- **Java基础（珊珊老师，海阳老师）**
  - Java概述
    - 何为编程
    - 什么是Java
    - jdk1.5之后的三大版本
    - JVM、JRE和JDK的关系
    - 什么是跨平台性？原理是什么
    - Java语言有哪些特点
    - 什么是字节码？采用字节码的大好处是什么
    - 什么是Java程序的主类？应用程序和小程序的主类有何不同？
    - Java应用程序与小程序之间有那些差别？
    - Java和C++的区别
    - Oracle JDK 和 OpenJDK 的对比
  - 基础语法
    - 数据类型
      - Java有哪些数据类型
      - switch 是否能作用在 byte 上，是否能作用在 long 上，是否 能作用在 String 上
      - 用最有效率的方法计算 2 乘以 8
      - Math.round(11.5) 等于多少？Math.round(-11.5) 等于多少
      - loat f=3.4;是否正确
      - short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗
    - 编码
      - Java语言采用何种编码方案？有何特点？
    - 注释
      - 什么Java注释
        - 定义：用于解释说明程序的文字
        - 分类
        - 作用
      - 访问修饰符 public,private,protected,以及不写（默认）时的 区别
    - 运算符
      - &和&&的区别
    - 关键字
      - Java 有没有 goto
      - final 有什么用?
      - final finally finalize区别
      - this关键字的用法
      - super关键字的用法
      - this与super的区别
      - static存在的主要意义
      - static的独特之处
      - static应用场景
      - static注意事项
    - 流程控制语句
      - break ,continue ,return 的区别及作用
      - 在 Java 中，如何跳出当前的多重嵌套循环
  - 面向对象
    - 面向对象概述
      - 面向对象和面向过程的区别
        - 面向过程：
        - 面向对象：
    - 面向对象三大特性
      - 面向对象的特征有哪些方面
      - 什么是多态机制？Java语言是如何实现多态的？
    - 面向对象五大基本原则是什么（可选）
  - 类与接口

- 抽象类和接口的对比
  - 普通类和抽象类有哪些区别?
  - 抽象类能使用 final 修饰吗?
  - 创建一个对象用什么关键字？对象实例与对象引用有何不同?
- 变量与方法
  - 成员变量与局部变量的区别有哪些
    - 作用域
    - 存储位置
    - 生命周期
    - 初始值
  - 在Java中定义一个不做事且没有参数的构造方法的作用
  - 在调用子类构造方法之前会先调用父类没有参数的构造方法，其 目的是?
  - 一个类的构造方法的作用是什么？若一个类没有声明构造方法， 改程序能正确执行吗？为什么?
  - 构造方法有哪些特性?
  - 静态变量和实例变量区别
  - 静态变量与普通变量区别
  - 静态方法和实例方法有何不同?
  - 在一个静态方法内调用一个非静态成员为什么是非法的?
  - 什么是方法的返回值？返回值的作用是什么?
- 内部类
  - 什么是内部类?
  - 内部类的分类有哪些
    - 静态内部类
      - 成员内部类
      - 局部内部类
      - 匿名内部类
      - 匿名内部类创建方式：
  - 内部类的优点
  - 内部类有哪些应用场景
  - 局部内部类和匿名内部类访问局部变量的时候，为什么变量必须 要加上final?
  - 内部类相关，看程序说出运行结果
- 重写与重载
  - 构造器（constructor）是否可被重写（override）
  - 重载（Overload）和重写（Override）的区别。重载的方法能 否根据返回类型进行区分?
- 对象相等判断
  - == 和 equals 的区别是什么
  - hashCode 与 equals (重要)
    - hashCode()介绍
    - 为什么要有 hashCode
    - 我们以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode：
    - hashCode()与equals()的相关规定
    - 因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖
  - 对象的相等与指向他们的引用相等，两者有什么不同?
- 值传递
  - 当一个对象被当作参数传递到一个方法后，此方法可改变这个对 象的属性，并可返回变化后的结果，那么
  - 这里到底是值传递还是 引用传递
  - 为什么 Java 中只有值传递
  - 值传递和引用传递有什么区别
- Java包
  - JDK 中常用的包有哪些
  - import java和javax有什么区别
- IO流
  - java 中 IO 流分为几种?
  - BIO,NIO,AIO 有什么区别?
  - Files的常用方法都有哪些?
- 反射
  - 什么是反射机制?
    - 反射机制优缺点

反射机制的应用场景有哪些?
Java获取反射的三种方法
常用API
String相关
字符型常量和字符串常量的区别
什么是字符串常量池?
String 是最基本的数据类型吗
String有哪些特性
String为什么是不可变的吗?
String真的是不可变的吗?
是否可以继承 String 类
数组有没有 length()方法? String 有没有 length()方法
String 类的常用方法都有那些?
在使用 HashMap 的时候，用 String 做 key 有什么好处?
String和StringBuffer、StringBuilder的区别是什么? String 为什么是不可变的
Date相关
包装类相关
拆箱：将包装类型转换为基本数据类型； int 和 Integer 有什么区别
Integer a= 127 与 Integer b = 127相等吗
集合容器概述
什么是集合
集合的特点
集合和数组的区别
使用集合框架的好处
常用的集合类有哪些?
List，Set，Map三者的区别? List、Set、Map 是否继 承自 Collection 接口? List、Map、Set 三个接口存取 元素时，各有什么特点?
集合框架底层数据结构
哪些集合类是线程安全的?
Java集合的快速失败机制 “fail-fast”?
怎么确保一个集合不能被修改?
Collection接口
List接口
迭代器 Iterator 是什么?
Iterator 怎么使用? 有什么特点?
如何边遍历边移除 Collection 中的元素?
Iterator 和 ListIterator 有什么区别?
遍历一个 List 有哪些不同的方式? 每种方法的实现原理是什 么? Java 中 List 遍历的最佳实践是什么?
遍历方式有以下几种:
说一下 ArrayList 的优缺点
如何实现数组和 List 之间的转换?
ArrayList 和 LinkedList 的区别是什么?
ArrayList 和 Vector 的区别是什么?
插入数据时，ArrayList、LinkedList、Vector谁速度较快? 阐述 ArrayList、Vector、LinkedList 的存储性能和特性?
多线程场景下如何使用 ArrayList?
为什么 ArrayList 的 elementData 加上 transient 修饰? ArrayList 中的数组定义如下:
List 和 Set 的区别
Set接口
说一下 HashSet 的实现原理?
HashSet如何检查重复? HashSet是如何保证数据不可重复的?
HashSet与HashMap的区别
Queue
BlockingQueue是什么?
Map接口
说一下 HashMap 的实现原理?
HashMap在JDK1.7和JDK1.8中有哪些不同? HashMap的底层实现
JDK1.8之前

JDK1.8之后
JDK1.7 VS JDK1.8 比较
HashMap的put方法的具体流程?
HashMap的扩容操作是怎么实现的?
HashMap是怎么解决哈希冲突的?
什么是哈希冲突?
HashMap的数据结构
JDK1.8新增红黑树
能否使用任何类作为 Map 的 key?
为什么HashMap中String、Integer这样的包装类适合作为K?
如果使用Object作为HashMap的Key，应该怎么办呢?
HashMap为什么不直接使用hashCode()处理后的哈希 值直接作为table的下标?
HashMap 的长度为什么是2的幂次方
HashMap 与 HashTable 有什么区别?
如何决定使用 HashMap 还是TreeMap?
HashMap 和 ConcurrentHashMap 的区别
ConcurrentHashMap 和 Hashtable 的区别?
ConcurrentHashMap 底层具体实现知道吗？实现原理是什么?
辅助工具类
Array 和 ArrayList 有何区别?
如何实现 Array 和 List 之间的转换?
comparable 和 comparator的区别?
Collection 和 Collections 有什么区别?
TreeMap 和 TreeSet 在排序时如何比较元素？ Collections 工具类中的 sort()方法如何比较元素?
Vector,ArrayList, LinkedList的区别是什么?
HashTable, HashMap，TreeMap区别?
HashMap的数据结构
HashMap的扩容因子
多线程修改HashMap
说说你知道的几个Java集合类：list、set、queue、map实现类
Java中的队列都有哪些，有什么区别
反射中，Class.forName和classloader的区别
讲讲IO里面的常见类，字节流、字符流、接口、实现类、方法阻塞
Java IO流详解（二）——IO流的框架体系
一、IO流的概念
二、IO流的分类
1.输入流和输出流
2.字节流和字符流
3.节点流和处理流
三、IO流的四大基类
1.InputStream
2.Reader
3.OutputStream
4.Writer
讲讲NIO
NIO技术概览
IO模型的分类
两种IO多路复用方案：Reactor和Proactor
传统BIO模型
NIO的实现原理
Reactor模型
Reactor单线程模型
Reactor多线程模型
主从Reactor多线程模型
AIO
select和epoll的区别
NIO与epoll
Zero Copy

传统的数据传输方式
Zero Copy的数据传输方式
NIO存在的问题
总结
三个channel使用 ServerSocketChannel||SocketChannel||FileChannel
Java NIO系列教程 FileChannel
打开FileChannel
从FileChannel读取数据
向FileChannel写数据
关闭FileChannel
FileChannel的position方法
FileChannel的size方法
FileChannel的truncate方法
FileChannel的force方法
Java NIO系列教程 SocketChannel
打开 SocketChannel
关闭 SocketChannel
从 SocketChannel 读取数据
写入 SocketChannel
非阻塞模式
connect()
write()
read()
非阻塞模式与选择器
Java NIO系列教程 ServerSocketChannel
打开 ServerSocketChannel
关闭 ServerSocketChannel
监听新进来的连接
非阻塞模式
String 编码UTF-8 和GBK的区别
什么时候使用字节流、什么时候使用字符流
什么时候使用字节流、什么时候使用字符流，二者的区别
递归读取文件夹下的文件，代码怎么实现
SynchronousQueue实现原理
前言
SynchronousQueue简单使用
SynchronousQueue实现原理
公平模式下的模型：
非公平模式下的模型：
总结
自定义类加载器
面向对象和面向过程的区别
面向过程
面向对象
Java 语言有哪些特点
关于 JVM JDK 和 JRE 最详细通俗的解答
JVM
JDK 和 JRE
Oracle JDK 和 OpenJDK 的对比
Java 和 C++的区别
什么是 Java 程序的主类 应用程序和小程序的主类有何不同
Java 应用程序与小程序之间有那些差别
字符型常量和字符串常量的区别
构造器 Constructor 是否可被 override
重载和重写的区别
Java 面向对象编程三大特性: 封装 继承 多态
封装
继承

多态
String StringBuffer 和 StringBuilder 的区别是什么 String 为什么是不可变的
自动装箱与拆箱
在一个静态方法内调用一个非静态成员为什么是非法的
在 Java 中定义一个不做事且没有参数的构造方法的作用
import java 和 javax 有什么区别
接口和抽象类的区别是什么
成员变量与局部变量的区别有那些
创建一个对象用什么运算符?对象实体与对象引用有何不同?
什么是方法的返回值?返回值在类的方法里的作用是什么?
一个类的构造方法的作用是什么 若一个类没有声明构造方法,该程序能正确执行吗 ?为什么?
构造方法有哪些特性
静态方法和实例方法有何不同
对象的相等与指向他们的引用相等，两者有什么不同?
在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是?
== 与 equals(重要)
hashCode 与 equals（重要）
hashCode（）介绍
为什么要有 hashCode
hashCode（）与 equals（）的相关规定
关于final关键字的一些总结
Java 中的异常处理
Java 异常类层次结构图
Throwable 类常用方法
异常处理总结
Java 序列化中如果有些字段不想进行序列化 怎么办
获取用键盘输入常用的的两种方法
接口继承关系和实现
List
Set
Map
TreeSet 和 TreeMap 的关系
TreeMap 的添加节点
图 1. 排序二叉树
TreeMap 的删除节点
图 2. 被删除节点只有左子树
图 3. 被删除节点只有右子树
图 4. 被删除节点既有左子树，又有右子树
图 5. 被删除节点既有左子树，又有右子树
红黑树
图 6. Java 红黑树的示意
添加节点后的修复
图 7. 插入节点后进行颜色调换
图 8. 插入节点后的树旋转
图 9. 插入节点后的颜色调整、树旋转
删除节点后的修复
检索节点
JAVA 异常分类及处理
JAVA 反射
JAVA 注解
JAVA 内部类
JAVA 泛型
JAVA 序列化**(**创建可复用的 **Java** 对象**)**
JAVA 复制
Java9比Java8改进了什么；
JAVA8 十大新特性详解
一、接口的默认方法
二、Lambda 表达式

三、函数式接口
四、方法与构造函数引用
五、Lambda 作用域
六、访问局部变量
七、访问对象字段与静态变量
八、访问接口的默认方法
九、Date API
十、Annotation 注解

Java 9 逆天的十大新特性

HashMap内部的数据结构是什么？底层是怎么实现的?

延伸考察ConcurrentHashMap与HashMap、HashTable等，考察 对技术细节的深入了解程度；

老生常谈，HashMap的死循环
问题
原因分析
实现
案例分析
总结

ConcurrentHashMap在jdk1.8中的改进

谈谈ConcurrentHashMap1.7和1.8的不同实现
ConcurrentHashMap
JDK1.7
JDK1.8

深入分析ConcurrentHashMap1.8的扩容实现
什么情况会触发扩容
transfer实现

深入浅出ConcurrentHashMap1.8
前言
JDK1.7分析
JDK1.8分析
重要概念
实例初始化
table初始化
table扩容
红黑树构造
get操作
总结

ConcurrentHashMap的红黑树实现分析
红黑树
ConcurrentHashMap红黑树实现
红黑树构造过程

说说反射的用途及实现，反射是不是很慢，我们在项目中是否要避免使 用反射；

一、用途 反射被广泛地用于那些需要在运行时检测或修改程序行为的程序中。
二、实现方式 Foo foo = new Foo();
三、缺点

说说自定义注解的场景及实现；

List 和 Map 区别
一、概述
二、明细
List
Map
Set（问题扩展）
三、场景（问题扩展）

List、Set、Map的区别
1.面试题：你说说collection里面有什么子类。
（其实面试的时候听到这个问题的时候，你要知道，面试官是想考察List，Set）
正如图一，list和set是实现了collection接口的。
List：1.可以允许重复的对象。
2.可以插入多个null元素。

3.是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。
4.常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。
Set：1.不允许重复对象
2. 无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或者 Comparable 维护了一个排序顺序。
3. 只允许一个 null 元素
4.Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。
1.Map不是collection的子接口或者实现类。Map是一个接口。
3. TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。
4. Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。
5.Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）
2.面试题：什么场景下使用list，set，map呢？
（或者会问为什么这里要用list、或者set、map，这里回答它们的优缺点就可以了）
答：
大家可以跟着下面的步骤一起尝试一下。
1.我们知道了列表要实现排序，需要重写comparable接口的compareTo的方法。
但是是我不知道comparaTo里面要怎么写呢，它有传入参数吗？它有返回值吗？如果有事什么类型的呢？ok，下面一起来做一下。先把这个链接的帮助文档下载下来。下载完之后，打开帮助文档，

Arraylist 与 LinkedList 区别，ArrayList 与 Vector 区别；

Java异常架构与异常关键字

- Java异常简介
- Java异常架构
  - 1. Throwable
  - 2. Error（错误）
  - 3. Exception（异常）
    - 运行时异常
    - 编译时异常
  - 4. 受检异常与非受检异常
    - 受检异常
    - 非受检异常
- Java异常关键字

Java异常处理

- 声明异常
- 抛出异常
- 捕获异常
- 常见异常处理方式直接抛出异常
  - 封装异常再抛出
  - 捕获异常
  - 自定义异常
  - try-catch-finally
  - try-with-resource

Java异常常见面试题

1. Error 和 Exception 区别是什么?
2. 运行时异常和一般异常(受检异常)区别是什么?
3. JVM 是如何处理异常的?
4. throw 和 throws 的区别是什么?
5. final、finally、finalize 有什么区别?
6. NoClassDefFoundError 和 ClassNotFoundException 区别?
7. try-catch-finally 中哪个部分可以省略?
8. try-catch-finally 中，如果 catch 中 return 了， finally 还会执行吗?
9. 类 ExampleA 继承 Exception，类 ExampleB 继承 ExampleA。
10. 常见的 RuntimeException 有哪些?
11. Java常见异常有哪些

Java异常处理最佳实践

1. 在 finally 块中清理资源或者使用 try-with-resource 语句
1.1 使用 finally 代码块
1.2 Java 7 的 try-with-resource 语法
2. 优先明确的异常
3. 对异常进行文档说明
4. 使用描述性消息抛出异常
5. 优先捕获最具体的异常
6. 不要捕获 Throwable 类
7. 不要忽略异常
8. 不要记录并抛出异常
9. 包装异常时不要抛弃原始的异常
10. 不要使用异常控制程序的流程
11. 使用标准异常
12. 异常会影响性能
13. 总结
异常处理-阿里巴巴Java开发手册
Tomcat是什么?
Tomcat的缺省端口是多少，怎么修改
tomcat 有哪几种Connector 运行模式(优化)?
Tomcat有几种部署方式?
tomcat容器是如何创建servlet类实例？用到了什么 原理?
Tomcat工作模式
Tomcat顶层架构
Connector和Container的微妙关系
Container架构分析
Container如何处理请求的
JVM（马老师，黄老师）
说一下 JVM 的主要组成部分及其作用?
说一下 JVM 运行时数据区
深拷贝和浅拷贝
说一下堆栈的区别?
队列和栈是什么？有什么区别?
对象的创建
为对象分配内存
处理并发安全问题
对象的访问定位
句柄访问
直接指针
内存溢出异常 Java会存在内存泄漏吗？请简单描述
内存溢出异常
JVM内存模型，GC机制和原理；
GC分哪两种，Minor GC 和Full GC有什么区别？什么时候会触发Full GC？分别采用什么算法?
JVM里的有几种classloader，为什么会有多种?
什么是双亲委派机制？介绍一些运作过程，双亲委派模型的好处；
常见的JVM调优方法有哪些？可以具体到调整哪个参数，调成什么值?
JVM虚拟机内存划分、类加载器、垃圾收集算法、垃圾收集器、class 文件结构是如何解析的；
类（class）文件结构
User user = new User() 做了什么操作，申请了哪些内存?
Java的内存模型以及GC算法
JVM内存模型与GC算法
1.JVM内存模型
1.1程序计数器
1.2 Java虚拟机栈
1.3 本地方法栈
1.4 Java堆
1.5 方法区
1.6 运行时常量池
1.7 直接内存

2.GC算法
2.1 标记-清除算法
2.2 复制算法
2.3 标记-整理算法
3. 总结
jvm性能调优都做了什么
JVM性能调优
java classload 机制 详解
类加载器基本概念
java.lang.ClassLoader 类介绍
类加载器的树状组织结构
jvm 如何分配直接内存， new 对象如何不分配在堆而是栈上，常量池解析
JVM直接内存
浅谈HotSpot逃逸分析
JIT
逃逸分析
同步消除
标量替换
栈上分配
编译阈值
触摸java常量池
数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold，问通常多大，没做过一问便知）
深入理解JVM : Java对象内存分配策略
1. **对象优先在Eden分配**
2. **大对象直接进入老年代**
3. **长期存活的对象将进入老年代**
4. **动态对象年龄判定**
5. **空间分配担保**
老年代中数组的访问方式
GC算法，永久代对象如何GC，GC有环怎么处理
谁会被GC，什么时候GC
Java JVM：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）
如果想在 GC 中生存 1 次怎么办
分析System.gc()方法
JVM源码分析之SystemGC完全解读
概述
JDK里的System.gc的实现
Hotspot里System.gc的实现
如何找到native里的实现
DisableExplicitGC参数
ExplicitGCInvokesConcurrent参数
并行Full GC相对正常的Full GC效率高在哪里
stop the world
CMS GC
正常的Full GC
并行的Full GC
堆外内存常配合使用System GC
JVM 选项 -XX:+UseCompressedOops 有什么作用？为什么要使用？
写代码分别使得JVM的堆、栈和持久代发生内存溢出(栈溢出)
JVM内存溢出详解（栈溢出，堆溢出，持久代溢出以及无法创建本地线程）
写在前面
内存溢出和内存泄漏的区别：
全文简短总结，具体内容可以看下文。
栈溢出(StackOverflowError)
堆溢出(OutOfMemoryError:java heap space)
持久带溢出(OutOfMemoryError: PermGen space)
OutOfMemoryError:unable to create native thread
为什么jdk8用metaspace数据结构用来替代perm?

元空间（Metaspace）：
PermGen 空间的状况
Metaspace 内存分配模型
（最大区别）大部分类元数据都在本地内存中分配。
Metaspace 容量
Metaspace 垃圾回收
Java 堆内存的影响
Metaspace 监控
参数设置
更新原因
简单谈谈堆外内存以及你的理解和认识
JVM源码分析之堆外内存完全解读
概述
广义的堆外内存
狭义的堆外内存
JDK/JVM里DirectByteBuffer的实现
堆外内存默认是多大
为什么要主动调用System.gc
为什么要使用堆外内存
为什么不能大面积使用堆外内存
threadlocal使用场景及注意事项
JVM老年代和新生代的比例?
JVM-堆学习之新生代老年代持久带的使用关系
1.设置JVM启动参数
测试类
栈是运行时的单位，而堆是存储的单位。
为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗?
为什么不把基本类型放堆中呢?
堆中存什么？栈中存什么?
Java中的参数传递时传值呢？还是传引用?
java中四种引用类型（对象的强、软、弱和虚引用）
讲一讲内存分代及生命周期。
管中窥豹——从对象的生命周期梳理JVM内存结构、GC、类加载、AOP编程及性能监控
Java对象的创建
class文件载入过程
加载
连接
拓展：双亲委派机制与AOP面向切面编程原理
对象创建及方法调用
内存集中管理(模型及GC机制)
回收算法
回收器
新生代收集器
老年代收集器
G1收集器
调优
命令行工具(以Linux场景进行详解)
图形化工具jvisualvm.exe
优化策略使用
什么情况下触发垃圾回收?
Java JVM 8：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）
如何选择合适的垃圾收集算法?
深入理解垃圾收集器和收集器的选择策略
StringTable
探索StringTable提升YGC性能
验证问题
String.intern() 250w个String对象：
String.intern() 500w个String对象：

PrintStringTableStatistics结果解读：
String.intern() 250w个String对象&-XX:StringTableSize=2500000：
String.intern() 500w个String对象&-XX:StringTableSize=5000000：
PrintStringTableStatistics&StringTableSize结果解读：
引申问题
结论
JVM源码分析之String.intern()导致的YGC不断变长
概述
Demo
String.intern方法
StringTable为什么会影响YGC
YGC过程扫描StringTable对CPU影响大吗
StringTable什么时候清理
JVM中最大堆大小有没有限制?
JVM调优常用参数配置
堆配置
收集器设置：
垃圾回收统计信息
并行收集器设置
典型配置举例
堆大小设置
回收器的选择
调优总结：
调优方法
如何调优
内存泄露检查
如何进行JVM调优？有哪些方法?
如何合理的规划一次jvm性能调优
一、性能调优的层次
二、jvm调优流程
1、性能定义
2、性能调优原则
3、性能调优流程
三、确定内存占用
1、运行阶段
2、jvm内存分配&参数
3、计算活跃数据大小
四、延迟调优
1、系统延迟需求
2、优化新生代的大小
3、优化老年代的大小
对象提升率
五、吞吐量调优
六、最后
内存模型以及分区，需要详细到每个区放什么。
堆里面的分区：Eden，survival （from+ to），老年代，各自的特点。
对象创建方法，对象的内存分配，对象的访问定位。
GC 的两种判定方法：
SafePoint 是什么
GC 的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路?
GC 收集器有哪些？CMS 收集器与 G1 收集器的特点。
Minor GC 与 Full GC 分别在什么时候发生?
几种常用的内存调试工具：jmap、jstack、jconsole、jhat jstack
类加载的几个过程：
JVM 内存分哪几个区，每个区的作用是什么?
如和判断一个对象是否存活?(或者 GC 对象的判定方法)
简述 java 垃圾回收机制?

- **java 中垃圾收集的方法有哪些?**
- **java 内存模型**
- **java 类加载过程?**
- **简述 java 类加载机制?**
- **类加载器双亲委派模型机制?**
- **什么是类加载器，类加载器有哪些?**
- **简述 java 内存分配与回收策率以及 Minor GC 和Major GC**
- JVM
- **JVM** 内存区域
  - **程序计数器(线程私有)**
  - **虚拟机栈(线程私有)**
  - **本地方法区(线程私有)**
  - **堆（Heap-线程共享）-运行时数据区**
  - **方法区/永久代（线程共享）**
- **JVM 运行时内存**
  - **新生代**
  - **老年代**
  - **永久代**
  - **JAVA8与元数据**
- 垃圾回收与算法
  - **如何确定垃圾**
  - **标记清除算法（Mark-Sweep）**
- **JAVA四中引用类型**
- **GC 分代收集算法 VS 分区收集算法**
- **GC垃圾收集器**
  - **G1** 收集器
- **JAVA IO/NIO**
  - **JAVA IO** 包
  - **JAVA NIO**
- **JVM** 类加载机制
  - 类加载器
  - 双亲委派
  - **动态改变构造**

多线程（马老师，黄老师）

- **简述线程，程序、进程的基本概念。以及他们之间关系是什么**
- **线程有哪些基本状态?**
- 如何理解内存泄漏问题？有哪些情况会导致内存泄露？如何解决?
  - 详解java内存泄露和如何避免内存泄漏
- 线程池的原理，为什么要创建线程池？创建线程池的方式;
  - 原理:
    - 线程池的优点
    - 线程池的创建
    - 线程池的实现原理
    - RejetedExecutionHandler：饱和策略
    - Executor框架的两级调度模型
    - Callable、Future、FutureTash详解
    - Callable与FutureTask
    - Callable与Future
  - 创建线程池的几种方式:
- 线程的生命周期，什么时候会出现僵死进程;
- 说说线程安全问题，什么是线程安全，如何实现线程安全;
  - Java 多线程安全机制
    - 1.互斥同步锁（悲观锁）
    - 2.非阻塞同步锁
    - 3.无同步方案
- 创建线程池有哪几个核心参数？ 如何合理配置线程池的大小?
  - 关于线程池的执行原则及配置参数详解
    - 线程池的好处

创建线程池
具体参数介绍
线程池工作原则
任务队列 BlockingQueue
任务拒绝类型
java线程池如何合理的设置大小
volatile、ThreadLocal的使用场景和原理；
volatile的适用场景
模式 #1：状态标志
模式 #2：一次性安全发布（one-time safe publication）
模式 #3：独立观察（independent observation）
模式 #4：“volatile bean” 模式
模式 #5：开销较低的“读 - 写锁”策略
正确使用 Volatile 变量
Volatile 变量
正确使用 volatile 变量的条件
清单 1. 非线程安全的数值范围类
性能考虑
正确使用 volatile 的模式
模式 #1：状态标志
清单 2. 将 volatile 变量作为状态标志使用
模式 #2：一次性安全发布（one-time safe publication）
清单 3. 将 volatile 变量用于一次性安全发布
模式 #3：独立观察（independent observation）
清单 4. 将 volatile 变量用于多个独立观察结果的发布
模式 #4：“volatile bean” 模式
清单 5. 遵守 volatile bean 模式的 Person 对象
volatile 的高级模式
模式 #5：开销较低的读 - 写锁策略
清单 6. 结合使用 volatile 和 synchronized 实现 “开销较低的读 - 写锁”
结束语
ThreadLocal什么时候会出现OOM的情况？为什么？
ThreadLocal全部方法和内部类
ThreadLocal源码分析
ThreadLocal的set方法
ThreadLocal的get方法
ThreadLocal的内存回收
ThreadLocal可能引起的OOM内存溢出问题简要分析
总结
synchronized、volatile区别
synchronized和volatile区别
synchronized锁粒度、模拟死锁场景；
三大性质总结
1. 三大性质简介
2. 原子性
3. 有序性
4. 可见性
5. 总结
基于synchronized的对象锁，类锁以及死锁模拟
Java并发和并行
怎么提高并发量，请列举你所知道的方案？
高并发解决方案——提升高并发量服务器性能解决思路
系统的用户量有多少？多用户并发访问时如何解决？
大型网站是怎样解决多用户高并发访问的
说说阻塞队列的实现：可以参考ArrayBlockingQueue的底层实现（锁和同步都行）
Java阻塞队列ArrayBlockingQueue和LinkedBlockingQueue实现原理分析
进程通讯的方式：消息队列，共享内存，信号量，socket通讯等
Linux进程间通信方式--信号，管道，消息队列，信号量，共享内存

1、概述
2、信号
3、管道
4、消息队列
5、共享内存
6、信号量
为什么要用线程池
线程池的基础概念
自带线程池的各种坑
volatile关键字的用法：使多线程中的变量可见
Java并发编程：volatile关键字解析
一.内存模型的相关概念
二.并发编程中的三个概念
三.Java内存模型
四.深入剖析volatile关键字
五.使用volatile关键字的场景
线程的几种状态
常用的线程池模式以及不同线程池的使用场景
java线程池与五种常用线程池策略使用与解析
一.线程池
二.可选择的阻塞队列BlockingQueue详解
2.1 无界队列
2.2 有界队列
2.3 同步移交
2.4 几种BlockingQueue的具体实现原理
三.可选择的饱和策略RejectedExecutionHandler详解
3.1 AbortPolicy中止策略
3.2 DiscardPolicy抛弃策略
3.3 DiscardOldestPolicy抛弃旧任务策略
3.4 CallerRunsPolicy调用者运行
四.java提供的四种常用线程池解析
4.1 newCachedThreadPool
4.2 newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
4.3 newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
4.4 newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。
4.5 newWorkStealingPool创建一个拥有多个任务队列（以便减少连接数）的线程池。
线程间通信，wait和notify
wait和notify的理解与使用
java线程池主线程等待子线程执行完成
Java如何等待子线程执行结束
进程和线程的区别
什么叫线程安全？举例说明
并发、同步的接口或方法
Java并发编程的类、接口和方法
HashMap 是否线程安全，为何不安全。 ConcurrentHashMap，线程安全，为何安全。底层实现是怎么样的。
深入浅出ConcurrentHashMap1.8
前言
JDK1.7分析
JDK1.8分析
重要概念
实例初始化
table初始化
table扩容
红黑树构造
get操作

总结
谈谈ConcurrentHashMap1.7和1.8的不同实现
ConcurrentHashMap
JDK1.7
数据结构
put实现
size实现
JDK1.8
数据结构
put实现
size实现
ConcurrentHashMap的红黑树实现分析
红黑树
ConcurrentHashMap红黑树实现
红黑树构造过程
深入分析ConcurrentHashMap1.8的扩容实现
什么情况会触发扩容
transfer实现
老生常谈，HashMap的死循环
问题
原因分析
实现
案例分析
总结
volatile的理解
volatile特性
volatile的使用场景
如何保证内存可见性?
线程
JAVA 多线程并发
JAVA 线程实现/创建方式
4种线程池
线程生命周期(状态)
终止线程 4 种方式
leep 与 wait 区别
start 与 run 区别
JAVA 后台线程
JAVA 锁
乐观锁
悲观锁
自旋锁
Synchronized 同步锁
java 中的锁 -- 偏向锁、轻量级锁、自旋锁、重量级锁
理解锁的基础知识
基础知识之一：锁的类型
基础知识之二：java线程阻塞的代价
基础知识之三：markword
小结
java中的锁
自旋锁
重量级锁Synchronized
偏向锁
轻量级锁
总结
锁优化
减少锁的时间
减少锁的粒度
ConcurrentHashMap

LongAdder
LinkedBlockingQueue
锁粗化
使用读写锁
读写分离
使用cas
消除缓存行的伪共享
ReentrantLock
ReentrantLock 与**synchronized**
Condition 类和**Object** 类锁方法区别区别
tryLock和**lock**和**lockInterruptibly** 的区别
Semaphore 信号量
AtomicInteger
可重入锁（递归锁）
公平锁与非公平锁
ReadWriteLock 读写锁
共享锁和独占锁 java
重量级锁（**Mutex Lock**）
轻量级锁
偏向锁
分段锁
锁优化
线程基本方法
线程上下文切换
同步锁与死锁
线程池原理
JAVA 阻塞队列原理
CyclicBarrier、**CountDownLatch**、**Semaphore** 的用法
volatile 关键字的作用（变量可见性、禁止重排序）
如何在两个线程之间共享数据
ThreadLocalMap（线程的一个属性）
synchronized 和 **ReentrantLock** 的区别
ConcurrentHashMap 并发
Java 中用到的线程调度
进程调度算法
什么是 **CAS**（**比较并交换-乐观锁机制-锁自旋**）
什么是 **AQS**（**抽象的队列同步器**）
NIO是什么？适用于何种场景？
NIO技术概览
IO模型的分类
两种IO多路复用方案：Reactor和Proactor
传统BIO模型
NIO的实现原理
Reactor模型
Reactor单线程模型
Reactor多线程模型
主从Reactor多线程模型
AIO
select和epoll的区别
NIO与epoll
Zero Copy
传统的数据传输方式
Zero Copy的数据传输方式
NIO存在的问题
总结

Java基础（珊珊老师，海阳老师）

Java概述

何为编程

编程就是让计算机为解决某个问题而使用某种程序设计语言编写程序代码，并 终得到结果的过程。

为了使计算机能够理解人的意图，人类就必须要将需解决的问题的思路、方法、 和手段通过计算机能够理解的形式告诉计算机，使得计算机能够根据人的指令一 步一步去工作，完成某种特定的任务。这种人和计算机之间交流的过程就是编 程。

什么是Java

Java是一门面向对象编程语言，不仅吸收了C++语言的各种优点，还摒弃了 C++里难以理解的多继承、指针等概念，因此Java语言具有功能强大和简单易 用两个特征。Java语言作为静态面向对象编程语言的代表，极好地实现了面向对 象理论，允许程序员以优雅的思维方式进行复杂的编程 。

jdk1.5之后的三大版本

- Java SE（J2SE，Java 2 Platform Standard Edition，标准版） Java SE 以前称为 J2SE。它允许开发和部署在桌面、服务器、嵌入式环境和实时环境中使 用的 Java 应用程序。Java SE 包含了支持 Java Web 服务开发的类，并为Java EE和Java ME提供基础。
- Java EE（J2EE，Java 2 Platform Enterprise Edition，企业版） Java EE 以前称为 J2EE。企业版本帮助开发和部署可移植、健壮、可伸缩且安全的服务器 端Java 应用程序。Java EE 是在 Java SE 的基础上构建的，它提供 Web 服务、组件模型、 管理和通信 API，可以用来实现企业级的面向服务体系结构（service-oriented architecture，SOA）和 Web2.0应用程序。2018年2月，Eclipse 宣布正式将 JavaEE 更名 为 JakartaEE
- Java ME（J2ME，Java 2 Platform Micro Edition，微型版） Java ME 以前称为 J2ME。Java ME 为在移动设备和嵌入式设备（比如手机、PDA、电视 机顶盒和打印机）上运行的应用程序提供一个健壮且灵活的环境。Java ME 包括灵活的用 户界面、健壮的安全模型、许多内置的网络协议以及对可以动态下载的连网和离线应用程序 的丰富支持。基于 Java ME 规范的应用程序只需编写一次，就可以用于许多设备，而且可 以利用每个设备的本机功能。

JVM、JRE和JDK的关系

JVM

Java Virtual Machine是Java虚拟机，Java程序需要运行在虚拟机上，不同的平 台有自己的虚拟机，因此Java语言可以实现跨平台。

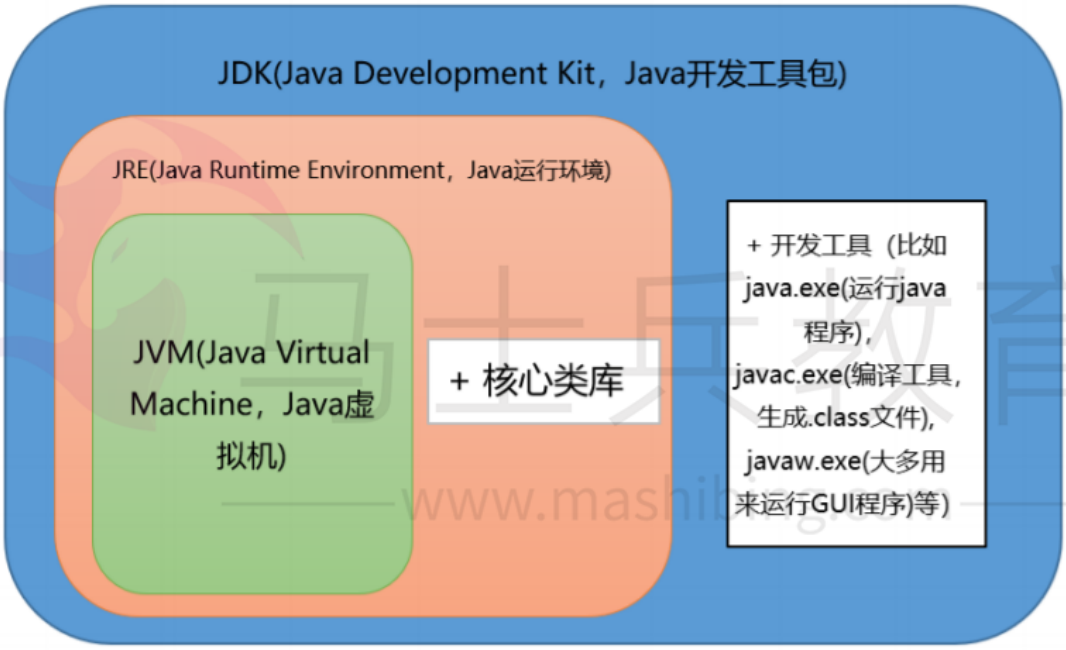
JRE

Java Runtime Environment包括Java虚拟机和Java程序所需的核心类库等。核 心类库主要是java.lang包：包含了运行Java程序必不可少的系统类，如基本数 据类型、基本数学函数、字符串处理、线程、异常处理类等，系统缺省加载这个包

 如果想要运行一个开发好的Java程序，计算机中只需要安装JRE即可。

JDK

Java Development Kit是提供给Java开发人员使用的，其中包含了Java的开发 工具，也包括了JRE。所以安装了JDK，就无需再单独安装JRE了。其中的开发工 具：编译工具(javac.exe)，打包工具(jar.exe)等



什么是跨平台性？原理是什么

所谓跨平台性，是指java语言编写的程序，一次编译后，可以在多个系统平台上 运行。

实现原理：Java程序是通过java虚拟机在系统平台上运行的，只要该系统可以安 装相应的java虚拟机，该系统就可以运行java程序。

Java语言有哪些特点

简单易学（Java语言的语法与C语言和C++语言很接近）

面向对象（封装，继承，多态）

平台无关性（Java虚拟机实现平台无关性）

支持网络编程并且很方便（Java语言诞生本身就是为简化网络编程设计的）

支持多线程（多线程机制使应用程序在同一时间并行执行多项任）

健壮性（Java语言的强类型机制、异常处理、垃圾的自动收集等）

安全性

什么是字节码？采用字节码的大好处是什么

字节码：Java源代码经过虚拟机编译器编译后产生的文件（即扩展为.class的文 件），它不面向任何特定的处理器，只面向虚拟机。

采用字节码的好处：

Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的 问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效， 而且，由于字节码并不专对一种特定的机器，因此，Java程序无须重新编译便可 在多种不同的计算机上运行。

先看下java中的编译器和解释器：

Java中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟机 器。这台虚拟的机器在任何平台上都提供给编译程序一个的共同的接口。编译程 序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代 码转换为特定系统的机器码执行。在Java中，这种供虚拟机理解的代码叫做字节 码（即扩展为.class的文件），它不面向任何特定的处理器，只面向虚拟机。每 一种平台的

解释器是不同的，但是实现的虚拟机是相同的。Java源程序经过编译 器编译后变成字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节 码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运 行，这就是上面提到的Java的特点的编译与解释并存的解释。

```
Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)---->jvm---->jvm中 解释器----->机器可执行的二进制机器码---->程序运行。
```

什么是Java程序的主类？应用程序和小程序的主类有何不同?

一个程序中可以有多个类，但只能有一个类是主类。在Java应用程序中，这个主 类是指包含main()方法的类。而在Java小程序中，这个主类是一个继承自系统 类Applet或Applet的子类。应用程序的主类不一定要求是public类，但小程序 的主类要求必须是public类。主类是Java程序执行的入口点。

Java应用程序与小程序之间有那些差别?

简单说应用程序是从主线程启动(也就是main()方法)。applet小程序没有main 方法，主要是嵌在浏览器页面上运行(调用init()线程或者run()来启动)，嵌入浏 览器这点跟flash的小游戏类似。

Java和C++的区别

我知道很多人没学过C++，但是面试官就是没事喜欢拿咱们Java和C++比呀！ 没办法！！！ 就算没学过C++，也要记下来！

- 都是面向对象的语言，都支持封装、继承和多态
- Java不提供指针来直接访问内存，程序内存更加安全
- Java的类是单继承的，C++支持多重继承；虽然Java的类不可以多继承，但是 接口可以多继承。
- Java有自动内存管理机制，不需要程序员手动释放无用内存

Oracle JDK 和 OpenJDK 的对比

1. Oracle JDK版本将每三年发布一次，而OpenJDK版本每三个月发布一 次；
2. OpenJDK 是一个参考模型并且是完全开源的，而Oracle JDK是 OpenJDK的一个实现，并不是完全开源的；
3. Oracle JDK 比 OpenJDK 更稳定。OpenJDK和Oracle JDK的代码几乎 相同，但Oracle JDK有更多的类和一些错误修复。因此，如果您想开发企 业/商业软件，我建议您选择Oracle JDK，因为它经过了彻底的测试和稳 定。某些情况下，有些人提到在使用OpenJDK 可能会遇到了许多应用程 序崩溃的问题，但是，只需切换到Oracle JDK就可以解决问题；
4. 在响应性和JVM性能方面，Oracle JDK与OpenJDK相比提供了更好的 性能；
5. Oracle JDK不会为即将发布的版本提供长期支持，用户每次都必须通过 更新到最新版本获得支持来获取最新版本；
6. Oracle JDK根据二进制代码许可协议获得许可，而OpenJDK根据GPL v2许可获得许可。

基础语法

数据类型

Java有哪些数据类型

定义：Java语言是强类型语言，对于每一种数据都定义了明确的具体的数据类 型，在内存中分配了不同大小的内存空间。

分类：

基本数据类型

- 数值型
    整数类型(byte,short,int,long)
    浮点类型(float,double)
- 字符型(char)
- 布尔型(boolean)

引用数据类型

- 类(class)
- 接口(interface)
- 数组([])

Java基本数据类型图

类型	类型名称	关键字	占用内存	取值范围	作为成员变量的默认值
整形	字节型	byte	1 字节	-128(-2^7) ~ 127(2^7-1)	0
	短整型	short	2 字节	-32,768(-2^15) ~ 32,767(2^15-1)	0
	整型	int	4 字节	-2,147,483,648(-2^31) ~ 2,147,483,647(2^31-1)	0
	长整型	long	8 字节	-9,223,372,036,854,775,808(-2^63) ~ 9,223,372,036,854,775,807(2^63-1)	0L
浮点型	单精度浮点型	float	4 字节	-3.403E38 ~ 3.403E38	0.0F
	双精度浮点型	double	8 字节	-1.798E308 ~ 1.798E308	0.0D
字符型	字符型	char	2 字节	表示一个字符，如('a','A','家')	'\u0000'
布尔型	布尔型	boolean	1 字节	只有两个值，true 或 false	false

switch 是否能作用在 byte 上，是否能作用在 long 上，是否 能作用在 String 上

在 Java 5 以前，switch(expr)中，expr 只能是 byte、short、char、int。从 Java5 开始，Java 中引入了枚举类型，expr 也可以是 enum 类型，从 Java 7 开始，expr 还可以是字符串（String），但是长整型（long）在目前所有的版 本中都是不可以的

用最有效率的方法计算 2 乘以 8

2 << 3（左移 3 位相当于乘以 2 的 3 次方，右移 3 位相当于除以 2 的 3 次 方）。

Math.round(11.5) 等于多少？ Math.round(-11.5) 等于多少

Math.round(11.5)的返回值是 12，Math.round(-11.5)的返回值是-11。四舍 五入的原理是在参数上加 0.5 然后进行下取整。

loat f=3.4;是否正确

不正确。3.4 是双精度数，将双精度型（double）赋值给浮点型（float）属于 下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转 换float f =(float)3.4; 或者写成 float f =3.4F;。

short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗

对于 short s1 = 1; s1 = s1 + 1;由于 1 是 int 类型，因此 s1+1 运算结果也是 int型，需要强制转换类型才能赋值给 short 型。 而 short s1 = 1; s1 += 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short(s1 + 1);其中有隐含的强制类型转换。

编码

Java语言采用何种编码方案？有何特点?

Java语言采用Unicode编码标准，Unicode（标准码），它为每个字符制订了一 个唯一的数值，因此在任何的语言，平台，程序都可以放心的使用。

注释

什么Java注释

定义：用于解释说明程序的文字

分类

- 单行注释 格式： // 注释文字
- 多行注释 格式： /* 注释文字 */
- 文档注释 格式：/** 注释文字 */

作用

在程序中，尤其是复杂的程序中，适当地加入注释可以增加程序的可读性，有利 于程序的修改、调试和交流。注释的内容在程序编译的时候会被忽视，不会产生 目标代码，注释的部分不会对程序的执行结果产生任何影响。 注意事项：多行和文档注释都不能嵌套使用。 访问修饰符

访问修饰符 public,private,protected,以及不写（默认）时的 区别

定义：Java中，可以使用访问修饰符来保护对类、变量、方法和构造方法的访 问。Java 支持 4 种不同的访问权限。

分类:

- private : 在同一类内可见。使用对象：变量、方法。 注意：不能修饰类（外部 类）
- default (即缺省，什么也不写，不使用任何关键字）: 在同一包内可见，不使用 任何修饰符。使用对象：类、接口、变量、方法。
- protected : 对同一包内的类和所有子类可见。使用对象：变量、方法。 注意： 不能修饰类（外部类）。
- public : 对所有类可见。使用对象：类、接口、变量、方法

访问修饰符图

修饰符	当前类	同 包	子 类	其他包
private	√	×	×	×
default	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

马士兵教育 www.mashibing.com

运算符

&和&&的区别

&运算符有两种用法：(1)按位与；(2)逻辑与。

&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要 求运算符左右两端的布尔值都是true 整个表达式的值才是 true。&&之所以称
为短路运算，是因为如果&&左边的表达式的值是 false，右边的表达式会被直 接短路掉，不会进行运算。

注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

关键字

Java 有没有 goto

goto 是 Java 中的保留字，在目前版本的 Java 中没有使用。

final 有什么用?

用于修饰类、属性和方法；

- 被final修饰的类不可以被继承
- 被final修饰的方法不可以被重写
- 被final修饰的变量不可以被改变，被final修饰不可变的是变量的引用，而不是引用指向的内容，引用指向的内容是可以改变的

final finally finalize区别

- final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。
- finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。
- finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用System.gc() 方法的时候，由垃圾回收器调用finalize()，回收垃圾，一个对象是否可回收的最后判断。

this关键字的用法

this是自身的一个对象，代表对象本身，可以理解为：指向对象本身的一个指针。

this的用法在java中大体可以分为3种：

1.普通的直接引用，this相当于是指向当前对象本身。

2.形参与成员名字重名，用this来区分：

```
1 public Person(String name, int age) {
2 this.name = name;
3 this.age = age;
4 }
```

3.引用本类的构造函数

```
1 class Person{
2 private String name;
3 private int age;
4
5 public Person() {
6 }
7
8 public Person(String name) {
9 this.name = name;
10 }
11 public Person(String name, int age) {
```

```
12 this(name);
13 this.age = age;
14 }
15 }
```

super关键字的用法

super可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己最近的一个父类。

super也有三种用法：

1.普通的直接引用
与this类似，super相当于是指向当前对象的父类的引用，这样就可以用super.xxx来引用父类的成员。
2.子类中的成员变量或方法与父类中的成员变量或方法同名时，用super进行区分

```
1 class Person{
2 protected String name;
3
4 public Person(String name) {
5 this.name = name;
6 }
7
8 }
9
10 class Student extends Person{
11 private String name;
12
13 public Student(String name, String name1) {
14 super(name);
15 this.name = name1;
16 }
17
18 public void getInfo(){
19 System.out.println(this.name); //Child
20 System.out.println(super.name); //Father
21 }
22
23 }
24
25 public class Test {
26 public static void main(String[] args) {
27 Student s1 = new Student("Father","Child");
28 s1.getInfo();
29
30 }
31 }
```

3.引用父类构造函数

super（参数）：调用父类中的某一个构造函数（应该为构造函数中的第一条语句）。

this（参数）：调用本类中另一种形式的构造函数（应该为构造函数中的第一条语句）。

this与super的区别

- super： 它引用当前对象的直接父类中的成员（用来访问直接父类中被隐藏的父类中成员数据或函数，基类与派生类中有相同成员定义时如：super.变量名 super.成员函数据名（实参）
- this：它代表当前对象名（在程序中易产生二义性之处，应使用this来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用this来指明成员变量名）
- super()和this()类似,区别是，super()在子类中调用父类的构造方法，this()在本类内调用本类的其它构造方法。
- super()和this()均需放在构造方法内第一行。
- 尽管可以用this调用一个构造器，但却不能调用两个。
- this和super不能同时出现在一个构造函数里面，因为this必然会调用其它的构造函数，其它的构造函数必然也会有super语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。
- this()和super()都指的是对象，所以，均不可以在static环境中使用。包括：static变量,static方法，static语句块。
- 从本质上讲，this是一个指向本对象的指针, 然而super是一个Java关键字。

static存在的主要意义

static的主要意义是在于创建独立于具体对象的域变量或者方法。以致于即使没有创建对象，也能使用属性和调用方法！

static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。为什么说static块可以用来优化程序性能，是因为它的特性:只会在类加载的时候
执行一次。因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。

static的独特之处

1、被static修饰的变量或者方法是独立于该类的任何对象，也就是说，这些变量和方法不属于任何一个实例对象，而是被类的实例对象所共享。

> 怎么理解 “被类的实例对象所共享” 这句话呢？就是说，一个类的静态成员，它是属于大伙的【大伙指的是这个类的多个对象实例，我们都知道一个类可以创建多个实例！】，所有的类对象共享的，不像成员变量是自个的【自个指的是这个类的单个实例对象】...我觉得我已经讲的很通俗了，你明白了咩？

2、在该类被第一次加载的时候，就会去加载被static修饰的部分，而且只在类第一次使用时加载并进行初始化，注意这是第一次用就要初始化，后面根据需要是可以再次赋值的。

3、static变量值在类加载的时候分配空间，以后创建类对象的时候不会重新分配。赋值的话，是可以任意赋值的！

4、被static修饰的变量或者方法是优先于对象存在的，也就是说当一个类加载完毕之后，即便没有创建对象，也可以去访问。

static应用场景

因为static是被类的实例对象所共享，因此如果某个成员变量是被所有对象所共享的，那么这个成员变量就应该定义为静态变量。

因此比较常见的static应用场景有：

> 1、修饰成员变量
> 2、修饰成员方法
> 3、静态代码块
> 4、修饰类【只能修饰内部类也就是静态内部类】
> 5、静态导包

static注意事项

1、静态只能访问静态。 2、非静态既可以访问非静态的，也可以访问静态的。

流程控制语句

break ,continue ,return 的区别及作用

break 跳出总上一层循环，不再执行循环(结束当前的循环体)continue 跳出本次循环，继续执行下次循环(结束正在执行的循环 进入下一个循环条件)

return 程序返回，不再执行下面的代码(结束当前的方法 直接返回)

在 Java 中，如何跳出当前的多重嵌套循环

在Java中，要想跳出多重循环，可以在外面的循环语句前定义一个标号，然后在里层循环体的代码中使用带有标号的break 语句，即可跳出外层循环。例如：

```
1 public static void main(String[] args) {
2 ok:
3 for (int i = 0; i < 10; i++) {
4 for (int j = 0; j < 10; j++) {
5 System.out.println("i=" + i + ",j=" + j);
6 if (j == 5) {
7 break ok;
8 }
9
10 }
11 }
12 }
```

面向对象

面向对象概述

面向对象和面向过程的区别

面向过程：

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，能是最重要的因素。
缺点：没有面向对象易维护、易复用、易扩展

面向对象：

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

缺点：性能比面向过程低
面向过程是具体化的，流程化的，解决一个问题，你需要一步一步的分析，一步
一步的实现。

面向对象是模型化的，你只需抽象出一个类，这是一个封闭的盒子，在这里你拥有数据也拥有解决问题的方法。需要什么功能直接使用就可以了，不必去一步一步的实现，至于这个功能是如何实现的，管我们什么事？我们会用就可以了。

面向对象的底层其实还是面向过程，把面向过程抽象成类，然后封装，方便我们使用的就是面向对象了。

面向对象三大特性

面向对象的特征有哪些方面

面向对象的特征主要有以下几个方面：

抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行 为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是 什么。

- 封装 : 封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如 果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没 有提供给外界访问的方法，那么这个类也没有什么意义了。
- 继承 : 继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新 的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用 继承我们能够非常方便地复用以前的代码。

  关于继承如下 3 点请记住：

  1.子类拥有父类非 private 的属性和方法。

  2.子类可以拥有自己属性和方法，即子类可以对父类进行扩展。

  3.子类可以用自己的方式实现父类的方法。
- 多态 : 所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出 的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到 底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的 方法，必须在由程序运行期间才能决定。

  在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口 （实现接口并覆盖接口中同一方法）。

其中Java 面向对象编程三大特性：封装 继承 多态

封装：隐藏对象的属性和实现细节，仅对外提供公共访问方式，将变化隔离，便 于使用，提高复用性和安全性。

继承：继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以 增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通 过使用继承可以提高代码复用性。继承是多态的前提。

关于继承如下 3 点请记住：

1. 子类拥有父类非 private 的属性和方法。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。

多态性：父类或接口定义的引用变量可以指向子类或具体实现类的实例对象。提 高了程序的拓展性。

在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口 （实现接口并覆盖接口中同一方法）。

方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重 写（override）实现的是运行时的多态性（也称为后绑定）。

一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是 哪个类中实现的方法，必须在由程序运行期间才能决定。运行时的多态是面向对 象精髓的东西，要实现多态需要做两件事：

- 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；
- 对象造型（用父类型引用子类型对象，这样同样的引用调用同样的方法就会根据 子类对象的不同而表现出不同的行为）。

什么是多态机制？Java语言是如何实现多态的？

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出 的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒 底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的 方法，必须在由程序运行期间才能决定。因为在程序运行时才确定具体的类，这 样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而 导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时 所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。 多态分为编译时多态和运行时多态。其中编辑时多态是静态的，主要是指方法的 重载，它是根据参数列表的不同来区分不同的函数，通过编辑之后会变成两个不 同的函数，在运行时谈不上多态。而运行时多态是动态的，它是通过动态绑定来 实现的，也就是我们所说的多态性

多态的实现

Java实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的子类和父类。

重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的 方法。

向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具 备技能调用父类的方法和子类的方法。

只有满足了上述三个条件，我们才能够在同一个继承结构中使用统一的逻辑实现 代码处理不同的对象，从而达到执行不同的行为。

对于Java而言，它多态的实现机制遵循一个原则：当超类对象引用变量引用子类 对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但 是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。

面向对象五大基本原则是什么（可选）

- 单一职责原则SRP(Single Responsibility Principle)类的功能要单一，不能包罗万象，跟杂货铺似的。
  开放封闭原则OCP(Open - Close Principle)
- 一个模块对于拓展是开放的，对于修改是封闭的，想要增加功能热烈欢迎，想要修改，哼，一万个不乐意。
- 里式替换原则LSP(the Liskov Substitution Principle LSP)子类可以替换父类出现在父类能够出现的任何地方。比如你能代表你爸去你姥姥家干活。哈哈~~
- 依赖倒置原则DIP(the Dependency Inversion Principle DIP)高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。抽象不应该依赖于具体实现，具体实现应该依赖于抽象。就是你出国要说你是中国人，而不能说你是哪个村子的。比如说中国人是抽象的，下面有具体的xx省，xx市，xx县。你要依赖的抽象是中国人，而不是你是xx村的。
- 接口分离原则ISP(the Interface Segregation Principle ISP)
  设计时采用多个与特定客户类有关的接口比采用一个通用的接口要好。就比如一个手机拥有打电话，看视频，玩游戏等功能，把这几个功能拆分成不同的接口，比在一个接口里要好的多。

类与接口

抽象类和接口的对比

抽象类是用来捕捉子类的通用特性的。接口是抽象方法的集合。
从设计层面来说，抽象类是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

相同点

- 接口和抽象类都不能实例化
- 都位于继承的顶端，用于被其他实现或继承
- 都包含抽象方法，其子类都必须覆写这些抽象方法

不同点

参数	抽象类	接口
声明	抽象类使用abstract关键字声明	接口使用interface关键字声明
实现	子类使用extends关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现	子类使用implements关键字来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
访问修饰符	抽象类中的方法可以是任意访问修饰符	接口方法默认修饰符是public。并且不允许定义为 private 或者 protected
多继承	一个类最多只能继承一个抽象类	一个类可以实现多个接口
字段声明	抽象类的字段声明可以是任意的	接口的字段默认都是 static 和 final 的

备注：Java8中接口中引入默认方法和静态方法，以此来减少抽象类和接口之间 的差异。

现在，我们可以为接口提供默认实现的方法了，并且不用强制子类来实现它。 接口和抽象类各有优缺点，在接口和抽象类的选择上，必须遵守这样一个原则：

- 行为模型应该总是通过接口而不是抽象类定义，所以通常是优先选用接口，尽量 少用抽象类。
- 选择抽象类的时候通常是如下情况：需要定义子类的行为，又要为子类提供通用 的功能。

普通类和抽象类有哪些区别?

- 普通类不能包含抽象方法，抽象类可以包含抽象方法。
- 抽象类不能直接实例化，普通类可以直接实例化。

抽象类能使用 final 修饰吗?

不能，定义抽象类就是让其他类继承的，如果定义为 final 该类就不能被继承， 这样彼此就会产生矛盾，所以 final 不能修饰抽象类

创建一个对象用什么关键字？对象实例与对象引用有何不同?

new关键字，new创建对象实例（对象实例在堆内存中），对象引用指向对象实 例（对象引用存放在栈内存中）。一个对象引用可以指向0个或1个对象（一根 绳子可以不系气球，也可以系一个气球）;一个对象可以有n个引用指向它（可以 用n条绳子系住一个气球）

变量与方法

成员变量与局部变量的区别有哪些

变量：在程序执行的过程中，在某个范围内其值可以发生改变的量。从本质上 讲，变量其实是内存中的一小块区域 成员变量：方法外部，类内部定义的变量 局部变量：类的方法中的变量。 成员变量和局部变量的区别

作用域

成员变量：针对整个类有效。

局部变量：只在某个范围内有效。(一般指的就是方法,语句体内)

存储位置

成员变量：随着对象的创建而存在，随着对象的消失而消失，存储在堆内存中。

局部变量：在方法被调用，或者语句被执行的时候存在，存储在栈内存中。当方法调用完，或者语句结束后，就自动释放。

生命周期

成员变量：随着对象的创建而存在，随着对象的消失而消失

局部变量：当方法调用完，或者语句结束后，就自动释放。

初始值

成员变量：有默认初始值。

局部变量：没有默认初始值，使用前必须赋值。

使用原则

在使用变量时需要遵循的原则为：就近原则 首先在局部范围找，有就使用；接着在成员位置找。

在Java中定义一个不做事且没有参数的构造方法的作用

Java程序在执行子类的构造方法之前，如果没有用super()来调用父类特定的构 造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了 有参数的构造方法，而在子类的构造方法中又没有用super()来调用父类中特定 的构造方法，则编译时将发生错误，因为Java程序在父类中找不到没有参数的构 造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

在调用子类构造方法之前会先调用父类没有参数的构造方法，其 目的是?

帮助子类做初始化工作。

一个类的构造方法的作用是什么？若一个类没有声明构造方法， 改程序能正确执行吗？为什么?

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构 造方法也会有默认的不带参数的构造方法。

构造方法有哪些特性?

名字与类名相同；

没有返回值，但不能用void声明构造函数；

生成类的对象时自动执行，无需调用。

静态变量和实例变量区别

静态变量： 静态变量由于不属于任何实例对象，属于类的，所以在内存中只会 有一份，在类的加载过程中，JVM只为静态变量分配一次内存空间。

实例变量： 每次创建对象，都会为每个对象分配成员变量内存空间，实例变量 是属于实例对象的，在内存中，创建几次对象，就有几份成员变量。

静态变量与普通变量区别

static变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有 的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始 化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副 本，各个对象拥有的副本互不影响。

还有一点就是static成员变量的初始化顺序按照定义的顺序进行初始化。

静态方法和实例方法有何不同?

静态方法和实例方法的区别主要体现在两个方面：

1. 在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使 用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调 用静态方法可以无需创建对象。
2. 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量 和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此 限制

在一个静态方法内调用一个非静态成员为什么是非法的?

由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静 态变量，也不可以访问非静态变量成员。

什么是方法的返回值？返回值的作用是什么?

方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果！（前提 是该方法可能产生结果）。返回值的作用:接收出结果，使得它可以用于其他的 操作!

内部类

什么是内部类?

在Java中，可以将一个类的定义放在另外一个类的定义内部，这就是内部类。内 部类本身就是类的一个属性，与其他属性定义方式一致。

内部类的分类有哪些

内部类可以分为四种：**成员内部类、局部内部类、匿名内部类和静态内部类。**

静态内部类

定义在类内部的静态类，就是静态内部类。

```
1 public class Outer {
2
3 private static int radius = 1;
4
5 static class StaticInner {
6 public void visit() {
7 System.out.println("visit outer static variable:" + radius);
8 }
9 }
10 }
```

静态内部类可以访问外部类所有的静态变量，而不可访问外部类的非静态变量； 静态内部类的创建方式，new 外部类.静态内部类()，如下：

```
1 Outer.StaticInner inner = new Outer.StaticInner();
2 inner.visit
```

成员内部类

定义在类内部，成员位置上的非静态类，就是成员内部类。

```
1 public class Outer {
2
3 private static int radius = 1;
4 private int count =2;
5
6 class Inner {
7 public void visit() {
8 System.out.println("visit outer static variable:" + radius);
9 System.out.println("visit outer variable:" + count);
10 }
11 }
12 }
```

成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和公 有。成员内部类依赖于外部类的实例，它的创建方式外部类实例.new 内部类()，如 下：

```
1 Outer outer = new Outer();
2 Outer.Inner inner = outer.new Inner();
3 inner.visit();
```

局部内部类

定义在方法中的内部类，就是局部内部类

```
1 public class Outer {
2
3 private int out_a = 1;
4 private static int STATIC_b = 2;
5
6 public void testFunctionClass(){
7 int inner_c =3;
8 class Inner {
9 private void fun(){
10 System.out.println(out_a);
11 System.out.println(STATIC_b);
12 System.out.println(inner_c);
13 }
14 }
15 Inner inner = new Inner();
16 inner.fun();
17 }
18 public static void testStaticFunctionClass(){
19 int d =3;
20 class Inner {
21 private void fun(){
22 // System.out.println(out_a); 编译错误，定义在静态方法中的局部类不可以访问外
部类的实例变量
23 System.out.println(STATIC_b);
24 System.out.println(d);
25 }
26 }
27 Inner inner = new Inner();
28 inner.fun();
29 }
30 }
```

定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法 中的局部类只能访问外部类的静态变量和方法。局部内部类的创建方式，在对应 方法内，new 内部类()，如下：

```
1 public static void testStaticFunctionClass(){
2 class Inner {
3 }
4 Inner inner = new Inner();
5 }
```

匿名内部类

匿名内部类就是没有名字的内部类，日常开发中使用的比较多。

```
1 public class Outer {
2
3 private void test(final int i) {
4 new Service() {
5 public void method() {
6 for (int j = 0; j < i; j++) {
7 System.out.println("匿名内部类" );
```

```
8 }
9 }
10 }.method();
11 }
12 }
13 //匿名内部类必须继承或实现一个已有的接口
14 interface Service{
15 void method();
16 }
```

除了没有名字，匿名内部类还有以下特点：

- 匿名内部类必须继承一个抽象类或者实现一个接口。
- 匿名内部类不能定义任何静态成员和静态方法。
- 当所在的方法的形参需要被匿名内部类使用时，必须声明为 final。
- 匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方 法。

匿名内部类创建方式：

```
1 new 类/接口{
2 //匿名内部类实现部分
3 }
```

内部类的优点

我们为什么要使用内部类呢？因为它有以下优点：

- 一个内部类对象可以访问创建它的外部类对象的内容，包括私有数据！
- 内部类不为同一包的其他类所见，具有很好的封装性；
- 内部类有效实现了“多重继承”，优化 java 单继承的缺陷。
- 匿名内部类可以很方便的定义回调。

内部类有哪些应用场景

1. 一些多算法场合
2. 解决一些非面向对象的语句块。
3. 适当使用内部类，使得代码更加灵活和富有扩展性。
4. 当某个类除了它的外部类，不再被其他的类使用时。

局部内部类和匿名内部类访问局部变量的时候，为什么变量必须 要加上final?

局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上final呢？ 它内部原理是什么呢？

先看这段代码：

```
1 public class Outer {
2
3 void outMethod(){
4 final int a =10;
5 class Inner {
6 void innerMethod(){
7 System.out.println(a);
8 }
9
10 }
11 }
12 }
```

以上例子，为什么要加final呢？是因为生命周期不一致， 局部变量直接存储在 栈中，当方法执行结束后，非final的局部变量就被销毁。而局部内部类对局部变 量的引用依然存在，如果局部内部类要调用局部变量时，就会出错。加了final， 可以确保局部内部类使用的变量与外层的局部变量区分开，解决了这个问题。

内部类相关，看程序说出运行结果

```
1 public class Outer {
2 private int age = 12;
3
4 class Inner {
5 private int age = 13;
6 public void print() {
7 int age = 14;
8 System.out.println("局部变量：" + age);
9 System.out.println("内部类变量：" + this.age);
10 System.out.println("外部类变量：" + Outer.this.age);
11 }
12 }
13
14 public static void main(String[] args) {
15 Outer.Inner in = new Outer().new Inner();
16 in.print();
17 }
18
19 }
```

运行结果：

```
1 局部变量：14
2 内部类变量：13
3 外部类变量：12
```

重写与重载

构造器（constructor）是否可被重写（override）

构造器不能被继承，因此不能被重写，但可以被重载。

重载（Overload）和重写（Override）的区别。重载的方法能 否根据返回类型进行区分?

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态 性，而后者实现的是运行时的多态性。

重载：发生在同一个类中，方法名相同参数列表不同（参数类型不同、个数不 同、顺序不同），与方法返回值和访问修饰符无关，即重载的方法不能根据返回 类型进行区分

重写：发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类，抛 出的异常小于等于父类，访问修饰符大于等于父类（里氏代换原则）；如果父类 方法访问修饰符为private则子类中就不是重写。

对象相等判断

== 和 equals 的区别是什么

== : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同 一个对象。(基本数据类型 == 比较的是值，引用数据类型 == 比较的是内存地址)

equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况： 情况1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时， 等价于通过“==”比较这两个对象。

情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象 的内容相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

举个例子：

```
1 public class test1 {
2 public static void main(String[] args) {
3 String a = new String("ab"); // a 为一个引用
4 String b = new String("ab"); // b为另一个引用,对象的内容一样
5 String aa = "ab"; // 放在常量池中
6 String bb = "ab"; // 从常量池中查找
7 if (aa == bb) // true
8 System.out.println("aa==bb");
9 if (a == b) // false，非同一对象
10 System.out.println("a==b");
11 if (a.equals(b)) // true
12 System.out.println("aEQb");
13 if (42 == 42.0) { // true
14 System.out.println("true");
15 }
16 }
17 }
```

说明：

- String中的equals方法是被重写过的，因为object的equals方法是比较的对象的 内存地址，而String的equals方法比较的是对象的值。
- 当创建String类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要 创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建 一个String对象。

hashCode 与 equals (重要)

HashSet如何检查重复

两个对象的 hashCode() 相同，则 equals() 也一定为 true，对吗?

hashCode和equals方法的关系

面试官可能会问你："你重写过 hashcode 和 equals 么，为什么重写equals时 必须重写hashCode方法？"

hashCode()介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int整 数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义 在JDK的Object.java中，这就意味着Java中的任何类都包含有hashCode()函数。

散列表存储的是键值对(key-value)，它的特点是：能根据"键"快速的检索出 对应的"值"。这其中就利用到了散列码！（可以快速找到所需要的对象）

为什么要有 hashCode

我们以"HashSet 如何检查重复"为例子来说明为什么要有 hashCode：

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对 象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果 没有相符的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相 同 hashcode 值的对象，这时会调用 equals()方法来检查 hashcode 相等的对 象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不 同的话，就会重新散列到其他位置。（摘自我的Java启蒙书《Head first java》 第二版）。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速 度。

hashCode()与equals()的相关规定

如果两个对象相等，则hashcode一定也是相同的 两个对象相等，对两个对象分别调用equals方法都返回true 两个对象有相同的hashcode值，它们也不一定是相等的

因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖

hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写

hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指 向相同的数据）

对象的相等与指向他们的引用相等，两者有什么不同?

对象的相等 比的是内存中存放的内容是否相等而 引用相等 比较的是他们指向的 内存地址是否相等。

值传递

当一个对象被当作参数传递到一个方法后，此方法可改变这个对 象的属性，并可返回变化后的结果，那么这里到底是值传递还是 引用传递

是值传递。Java 语言的方法调用只支持参数的值传递。当一个对象实例作为一 个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被 调用过程中被改变，但对对象引用的改变是不会影响到调用者的

为什么 Java 中只有值传递

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术 语。**按值调用(call by value)表示方法接收的是调用者提供的值，而按引用调用 （call by reference)表示方法接收的是调用者提供的变量地址。一个方法可以 修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。** 它用 来描述各种程序设计语言（不只是Java)中方法参数传递方式。

Java程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的 一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

下面通过 3 个例子来给大家说明

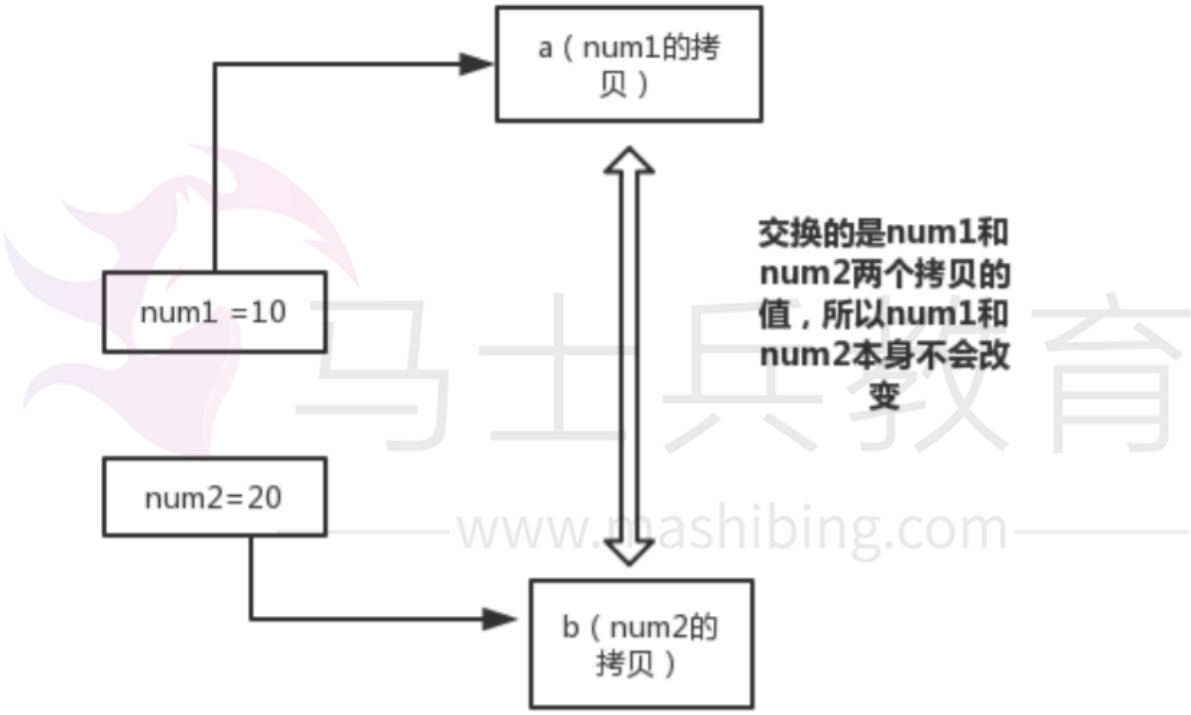
example 1

```
1 public static void main(String[] args) {
2 int num1 = 10;
3 int num2 = 20;
4
5 swap(num1, num2);
6
7 System.out.println("num1 = " + num1);
8 System.out.println("num2 = " + num2);
9 }
10
11 public static void swap(int a, int b) {
12 int temp = a;
13 a = b;
14 b = temp;
15
16 System.out.println("a = " + a);
17 System.out.println("b = " + b);
18 }
```

结果：

```
1 a = 20
2 b = 10
3 num1 = 10
4 num2 = 20
```

解析：



在swap方法中，a、b的值进行交换，并不会影响到 num1、num2。因为，a、 b中的值，只是从 num1、num2 的复制过来的。也就是说，a、b相当于 num1、num2 的副本，副本的内容无论怎么修改，都不会影响到原件本身。

通过上面例子，我们已经知道了一个方法不能修改一个基本数据类型的参数，而 对象引用作为参数就不一样，请看 example2.

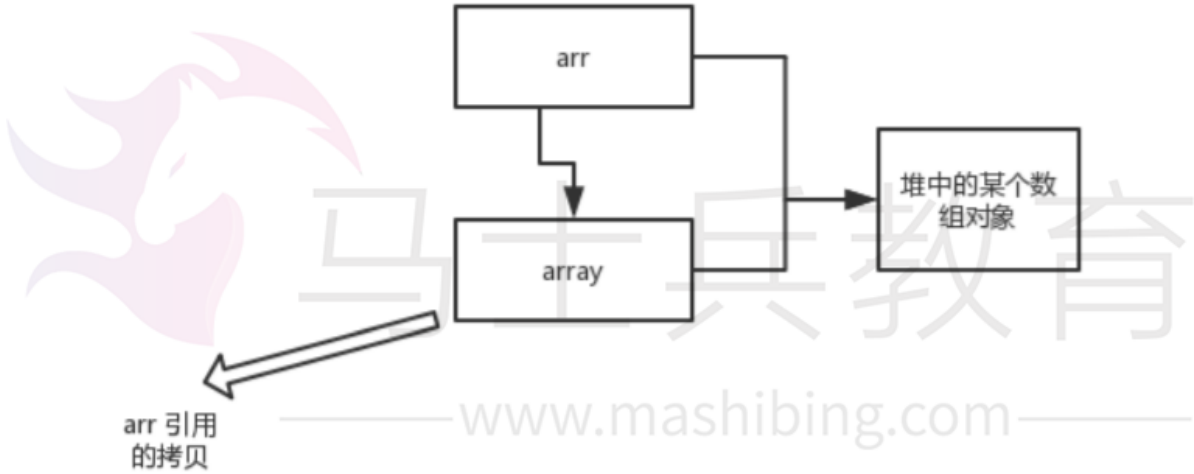
example 2

```
1 public static void main(String[] args) {
2 int[] arr = { 1, 2, 3, 4, 5 };
3 System.out.println(arr[0]);
4 change(arr);
5 System.out.println(arr[0]);
6 }
7
8 public static void change(int[] array) {
9 // 将数组的第一个元素变为0
10 array[0] = 0;
11 }
```

结果：

```
1 1
2 0
```

解析：



马士兵教育
www.mashibing.com

array 被初始化 arr 的拷贝也就是一个对象的引用，也就是说 array 和 arr 指向 的时同一个数组对象。因此，外部对引用对象的改变会反映到所对应的对象 上。

通过 example2 我们已经看到，实现一个改变对象参数状态的方法并不是一件 难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时 引用同一个对象。

很多程序设计语言（特别是，C++和Pascal)提供了两种参数传递的方式：值调 用和引用调用。有些程序员（甚至本书的作者）认为Java程序设计语言对对象 采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍 性，所以下面给出一个反例来详细地阐述一下这个问题。

example 3

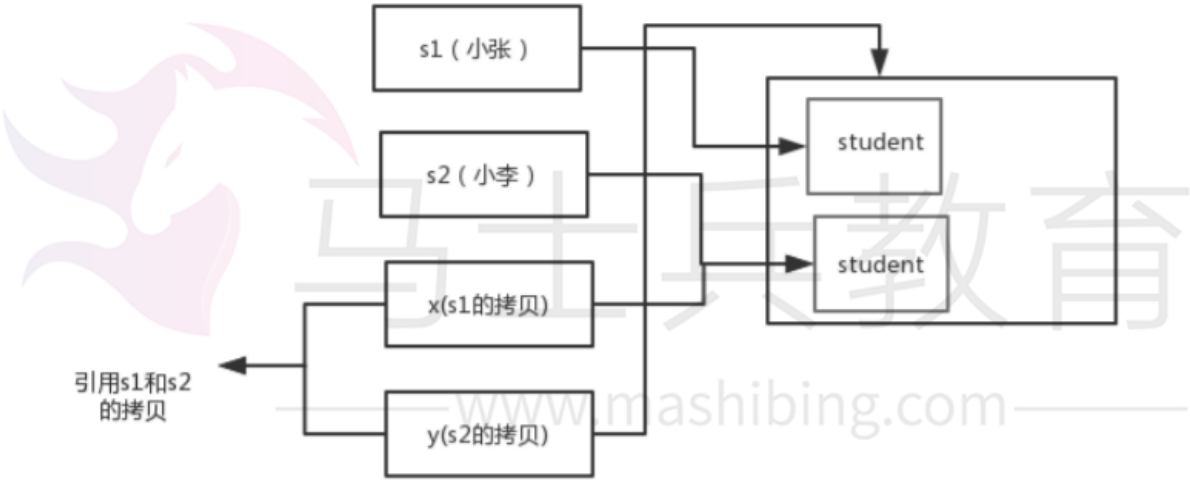
```
1 public class Test {
2
3 public static void main(String[] args) {
4 // TODO Auto-generated method stub
5 Student s1 = new Student("小张");
6 Student s2 = new Student("小李");
7 Test.swap(s1, s2);
8 System.out.println("s1:" + s1.getName());
9 System.out.println("s2:" + s2.getName());
10 }
11
12 public static void swap(Student x, Student y) {
13 Student temp = x;
14 x = y;
15 y = temp;
16 System.out.println("x:" + x.getName());
17 System.out.println("y:" + y.getName());
18 }
19 }
```

结果：

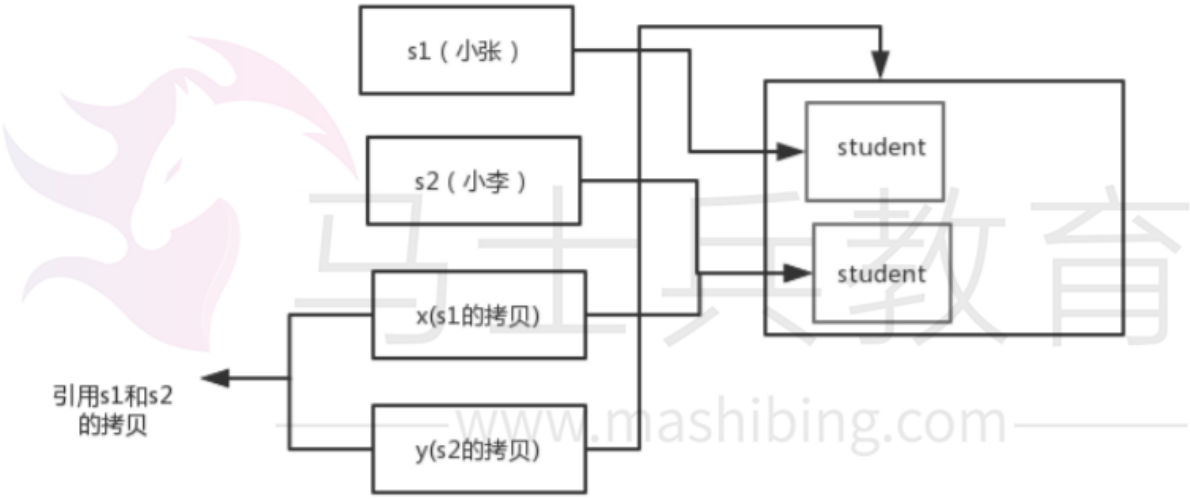
```
1 x:小李
2 y:小张
3 s1:小张
4 s2:小李
```

解析：

交换之前：



交换之后：



通过上面两张图可以很清晰的看出： 方法并没有改变存储在变量 s1 和 s2 中的 对象引用。swap方法的参数x和y被初始化为两个对象引用的拷贝，这个方法交 换的是这两个拷贝

总结

Java程序设计语言对对象采用的不是引用调用，实际上，对象引用是按值传递的。

下面再总结一下Java中方法参数的使用情况：

- 一个方法不能修改一个基本数据类型的参数（即数值型或布尔型》
- 一个方法可以改变一个对象参数的状态。
- 一个方法不能让对象参数引用一个新的对象。

值传递和引用传递有什么区别

值传递：指的是在方法调用时，传递的参数是按值的拷贝传递，传递的是值的拷 贝，也就是说传递后就互不相关了。 引用传递：指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引 用的地址，也就是变量所对应的内存空间的地址。传递的是值的引用，也就是说 传递前和传递后都指向同一个引用（也就是同一个内存空间）。

Java包

JDK 中常用的包有哪些

- java.lang：这个是系统的基础类；
- java.io：这里面是所有输入输出有关的类，比如文件操作等；
- java.nio：为了完善 io 包中的功能，提高 io 包中性能而写的一个新包；
- java.net：这里面是与网络有关的类；
- java.util：这个是系统辅助类，特别是集合类；
- java.sql：这个是数据库操作的类。

import java和javax有什么区别

刚开始的时候 JavaAPI 所必需的包是 java 开头的包，javax 当时只是扩展 API 包来说使用。然而随着时间的推移，javax 逐渐的扩展成为 Java API 的组成部 分。但是，将扩展从 javax 包移动到 java 包将是太麻烦了，最终会破坏一堆现 有的代码。因此，最终决定 javax 包将成为标准API的一部分。 所以，实际上java和javax没有区别。这都是一个名字。

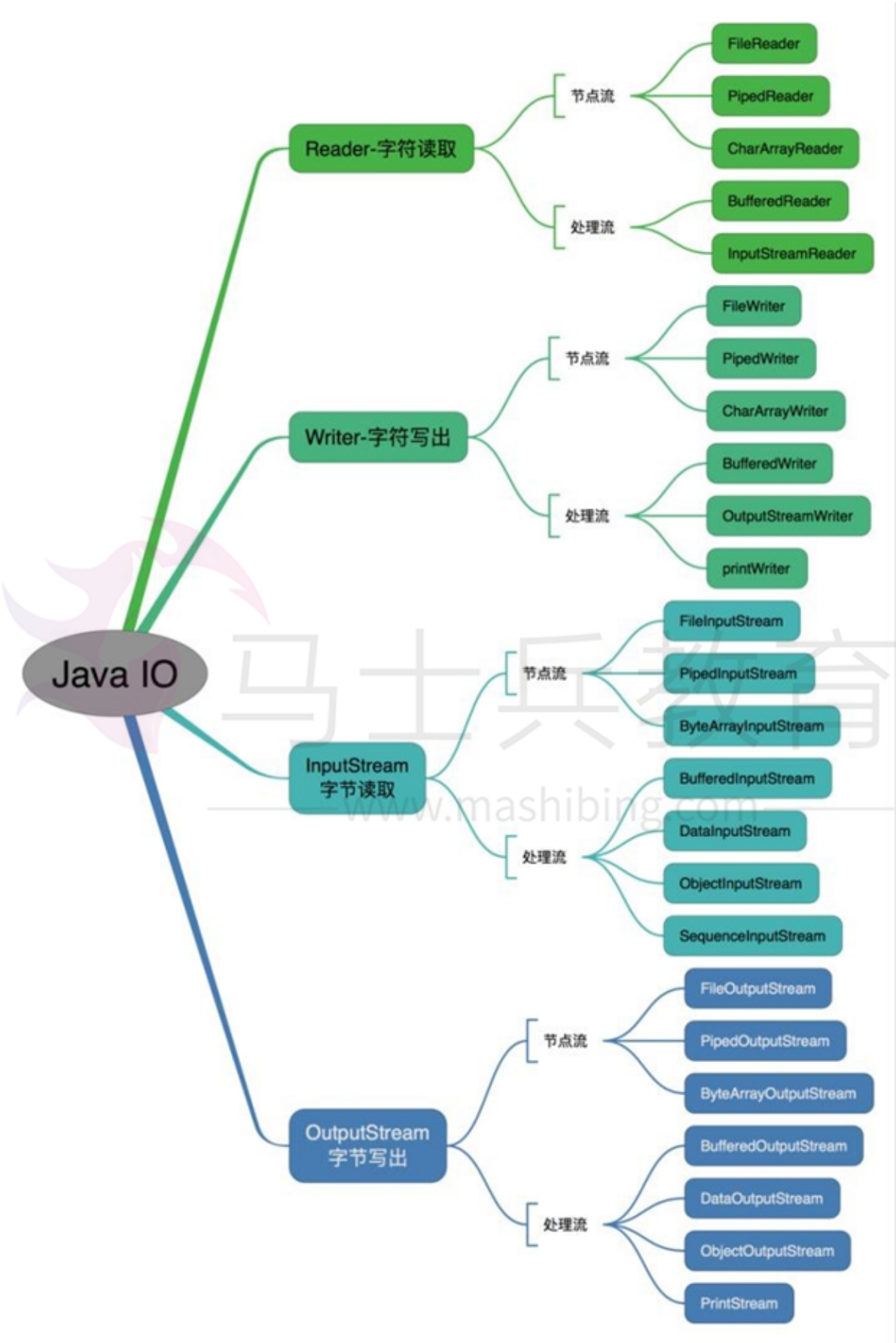
IO流

java 中 IO 流分为几种?

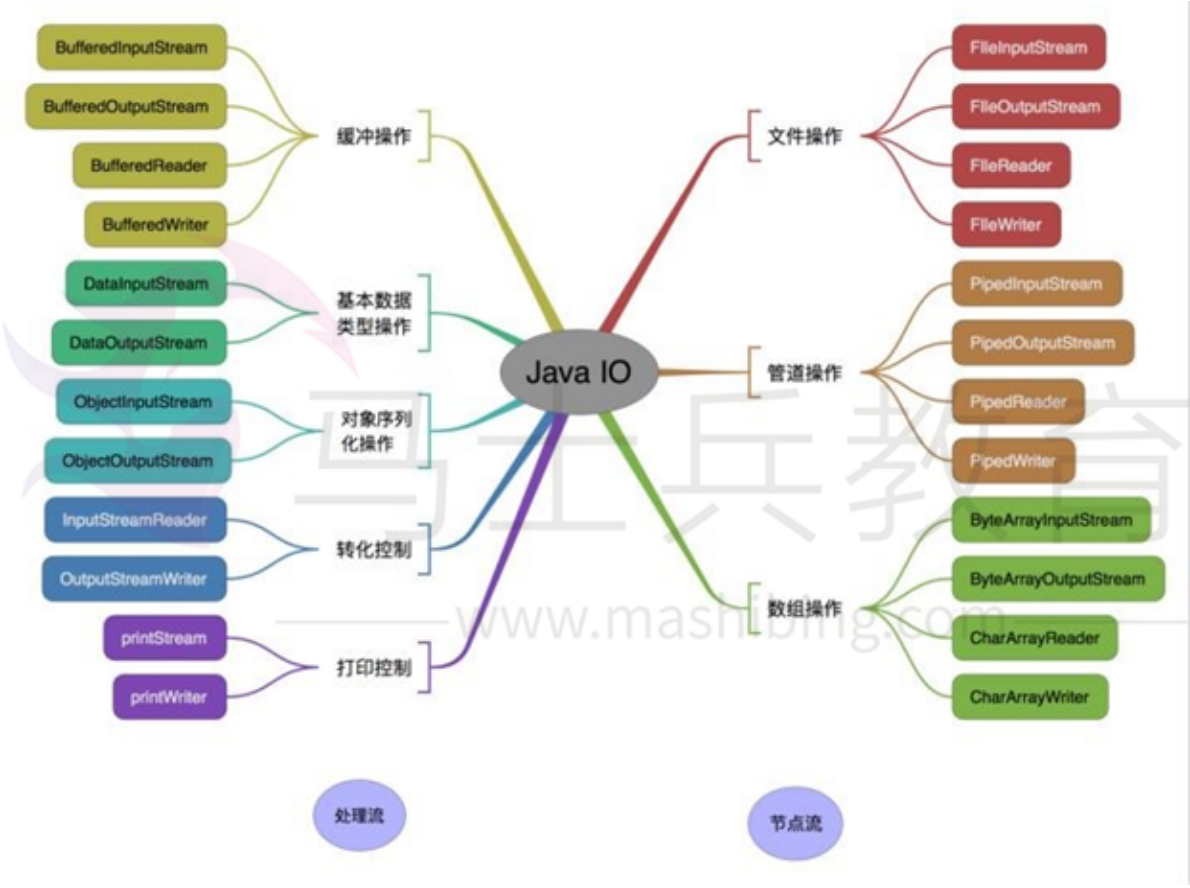
按照流的流向分，可以分为输入流和输出流； 按照操作单元划分，可以划分为字节流和字符流； 按照流的角色划分为节点流和处理流。 Java Io流共涉及40多个类，这些类看上去很杂乱，但实际上很有规则，而且彼 此之间存在非常紧密的联系， Java I0流的40多个类都是从如下4个抽象类基类 中派生出来的。

- InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符 输入流。
- OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输 出流。

按操作方式分类结构图：



IO操作方式分类按操作对象分类结构图：



BIO,NIO,AIO 有什么区别?

简答

- BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。
- NIO：Non IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。
- AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

详细回答

- BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- NIO (New I/O): NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的 N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。 NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
- AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操

作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

Files的常用方法都有哪些?

- Files. exists()：检测文件路径是否存在。
- Files. createFile()：创建文件。
- Files. createDirectory()：创建文件夹。
- Files. delete()：删除一个文件或目录。
- Files. copy()：复制文件。
- Files. move()：移动文件。
- Files. size()：查看文件个数。
- Files. read()：读取文件。
- Files. write()：写入文件。

反射

什么是反射机制?

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

静态编译和动态编译

静态编译：在编译时确定类型，绑定对象

动态编译：运行时确定类型，绑定对象

反射机制优缺点

优点： 运行期类型的判断，动态加载类，提高代码灵活度。

缺点： 性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能

比直接的java代码要慢很多。

反射机制的应用场景有哪些?

反射是框架设计的灵魂。

在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反射机制没有用，实际上有很多设计、开发都与反射机制有关，例如模块化的开发，通过反射去调用对应的字节码；动态代理设计模式也采用了反射机制，还有我们日常使用的 Spring / Hibernate 等框架也大量使用到了反射机制。

举例：①我们在使用JDBC连接数据库时使用Class.forName()通过反射加载数据库的驱动程序；②Spring框架也用到很多反射机制， 经典的就是xml的配置模式。Spring 通过 XML 配置模式装载 Bean 的过程：1) 将程序内所有 XML 或 Properties 配置文件加载入内存中; 2)Java类里面解析xml或properties里面的内容，得到对应实体类的字节码字符串以及相关的属性信息; 3)使用反射机制，根据这个字符串获得某个类的Class实例; 4)动态配置实例的属性

Java获取反射的三种方法

1.通过new对象实现反射机制

2.通过路径实现反射机制

3.通过类名实现反射机制

```
1  public class Student {
2  private int id;
3  String name;
4  protected boolean sex;
5  public float score;
6  }
1   public class Get {
2   //获取反射机制三种方式
3   public static void main(String[] args) throws ClassNotFoundException {
4   //方式一(通过建立对象)
5   Student stu = new Student();
6   Class classobj1 = stu.getClass();
7   System.out.println(classobj1.getName());
8   //方式二（所在通过路径-相对路径）
9   Class classobj2 = Class.forName("fanshe.Student");
10  System.out.println(classobj2.getName());
11  //方式三（通过类名）
12  Class classobj3 = Student.class;
13  System.out.println(classobj3.getName());
14  }
15  }
```

常用API

String相关

字符型常量和字符串常量的区别

1. 形式上: 字符常量是单引号引起的一个字符 字符串常量是双引号引起的若干个字符
2. 含义上: 字符常量相当于一个整形值(ASCII值),可以参加表达式运算 字符串常量代表一个地址值(该字符串在内存中存放位置)
3. 占内存大小 字符常量只占一个字节 字符串常量占若干个字节(至少一个字符结束标志)

什么是字符串常量池?

字符串常量池位于堆内存中，专门用来存储字符串常量，可以提高内存的使用率，避免开辟多块空间存储相同的字符串，在创建字符串时 JVM 会首先检查字符串常量池，如果该字符串已经存在池中，则返回它的引用，如果不存在，则实例化一个字符串放到池中，并返回其引用。

String 是最基本的数据类型吗

不是。Java 中的基本数据类型只有 8 个： byte、short、int、long、float、 double、char、boolean；除了基本类型（primitive type），剩下的都是引用类型（referencetype），Java 5 以后引入的枚举类型也算是一种比较特殊的引用类型。

这是很基础的东西，但是很多初学者却容易忽视，Java 的 8 种基本数据类型中不包括 String，基本数据类型中用来描述文本数据的是 char，但是它只能表示单个字符，比如 ‘a’,‘好’ 之类的，如果要描述一段文本，就需要用多个

char 类型的变量，也就是一个 char 类型数组，比如“你好” 就是长度为2的数组 char[] chars = {‘你’,‘好’};

但是使用数组过于麻烦，所以就有了 String，String 底层就是一个 char 类型的数组，只是使用的时候开发者不需要直接操作底层数组，用更加简便的方式即可完成对字符串的使用。

String有哪些特性

- 不变性：String 是只读字符串，是一个典型的 immutable 对象，对它进行任何操作，其实都是创建一个新的对象，再把引用指向该对象。不变模式的主要作用在于当一个对象需要被多线程共享并频繁访问时，可以保证数据的一致性。
- 常量池优化：String 对象创建之后，会在字符串常量池中进行缓存，如果下次创建同样的对象时，会直接返回缓存的引用。
- final：使用 final 来定义 String 类，表示 String 类不能被继承，提高了系统的安全性。

String为什么是不可变的吗?

简单来说就是String类利用了final修饰的char类型数组存储字符，源码如下图所以：

```
1  /** The value is used for character storage. */

2  private final char value[];
```

String真的是不可变的吗?

我觉得如果别人问这个问题的话，回答不可变就可以了。 下面只是给大家看两个有代表性的例子：

1) String不可变但不代表引用不可以变

```
1  String str = "Hello";

2  str = str + " World";

3  System.out.println("str=" + str);
```

结果：

```
1 str=Hello World
```

解析：

实际上，原来String的内容是不变的，只是str由原来指向"Hello"的内存地址转为指向"Hello World"的内存地址而已，也就是说多开辟了一块内存区域给"Hello World"字符串。

2) 通过反射是可以修改所谓的“不可变”对象

```
1  // 创建字符串"Hello World"， 并赋给引用s

2  String s = "Hello World";

3

4 System.out.println("s = " + s); // Hello World
```

```
5
6  // 获取String类中的value字段
7  Field valueFieldOfString = String.class.getDeclaredField("value");
8
9  // 改变value属性的访问权限
10  valueFieldOfString.setAccessible(true);
11
12  // 获取s对象上的value属性的值
13  char[] value = (char[]) valueFieldOfString.get(s);
14
15  // 改变value所引用的数组中的第5个字符
16  value[5] = '_';
17
18 System.out.println("s = " + s); // Hello_World
```

结果：

```
1  s = Hello World
2  s = Hello_World
```

解析：

用反射可以访问私有成员， 然后反射出String对象中的value属性， 进而改变通过获得的value引用改变数组的结构。但是一般我们不会这么做，这里只是简单提一下有这个东西。

是否可以继承 String 类

String 类是 final 类，不可以被继承。

String str="i"与 String str=new String(“i”)一样吗？不一样，因为内存的分配方式不一样。String str="i"的方式，java 虚拟机会将其分配到常量池中；而 String str=new String(“i”) 则会被分到堆内存中。 String s = new String(“xyz”);创建了几个字符串对象两个对象，一个是静态区的"xyz"，一个是用new创建在堆上的对象。

```
String str1 = "hello"; //str1指向静态区
String str2 = new String("hello"); //str2指向堆上的对象
String str3 = "hello";
String str4 = new String("hello");
System.out.println(str1.equals(str2)); //true
System.out.println(str2.equals(str4)); //true
System.out.println(str1 == str3); //true
System.out.println(str1 == str2); //false
System.out.println(str2 == str4); //false
System.out.println(str2 == "hello"); //false
str2 = str1;
System.out.println(str2 == "hello"); //true
```

如何将字符串反转？使用 StringBuilder 或者 stringBuffer 的 reverse() 方法。示例代码：

```
// StringBuffer reverse
StringBuffer stringBuffer = new StringBuffer();
stringBuffer. append("abcdefg");
System. out. println(stringBuffer. reverse()); // gfedcba
// StringBuilder reverse
StringBuilder stringBuilder = new StringBuilder();
stringBuilder. append("abcdefg");
System. out. println(stringBuilder. reverse()); // gfedcba
```

数组有没有 length()方法？String 有没有 length()方法

数组没有 length()方法 ，有 length 的属性。String 有 length()方法。 JavaScript中，获得字符串的长度是通过 length 属性得到的，这一点容易和 Java 混淆。

String 类的常用方法都有那些?

- indexOf()：返回指定字符的索引。
- charAt()：返回指定索引处的字符。
- replace()：字符串替换。
- trim()：去除字符串两端空白。
- split()：分割字符串，返回一个分割后的字符串数组。
- getBytes()：返回字符串的 byte 类型数组。
- length()：返回字符串长度。
- toLowerCase()：将字符串转成小写字母。
- toUpperCase()：将字符串转成大写字符。
- substring()：截取字符串。
- equals()：字符串比较。

在使用 HashMap 的时候，用 String 做 key 有什么好处?

HashMap 内部实现是通过 key 的 hashcode 来确定 value 的存储位置，因为字符串是不可变的，所以当创建字符串时，它的 hashcode 被缓存下来，不需要再次计算，所以相比于其他对象更快。

String和StringBuffer、StringBuilder的区别是什么？String 为什么是不可变的

可变性

String类中使用字符数组保存字符串，private final char value[]，所以 string对象是不可变的。StringBuilder与StringBuffer都继承自

AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，char[] value，这两种对象都是可变的。线程安全性

String中的对象是不可变的，也就可以理解为常量，线程安全。

AbstractStringBuilder是StringBuilder与StringBuffer的公共父类，定义了一些字符串的基本操作，如expandCapacity、append、insert、indexOf等公共方法。StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder并没有对方法进行加同步锁，所以是非线程安全的。

性能

每次对String 类型进行改变的时候，都会生成一个新的String对象，然后将指针指向新的String 对象。StringBuffer每次都会对StringBuffer对象本身进行操

作，而不是生成新的对象并改变对象引用。相同情况下使用StirngBuilder 相比使用StringBuffer 仅能获得10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结

如果要操作少量的数据用 = String 单线程操作字符串缓冲区 下操作大量数据 = StringBuilder 多线程操作字符串缓冲区 下操作大量数据 = StringBuffer

Date相关

包装类相关

自动装箱与拆箱

装箱：将基本类型用它们对应的引用类型包装起来；

拆箱：将包装类型转换为基本数据类型； int 和 Integer 有什么区别

Java 是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

原始类型: boolean，char，byte，short，int，long，float，double 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，

Double

Integer a= 127 与 Integer b = 127相等吗

对于对象引用类型：==比较的是对象的内存地址。

对于基本数据类型：==比较的是值。如果整型字面量的值在-128到127之间，那么自动装箱时不会new新的Integer 对象，而是直接引用常量池中的Integer对象，超过范围 a1==b1的结果是false

```
1   public static void main(String[] args) {
2   Integer a = new Integer(3);
3   Integer b = 3; // 将3自动装箱成Integer类型
4   int c = 3;
5   System.out.println(a == b); // false 两个引用没有引用同一对象
6   System.out.println(a == c); // true a自动拆箱成int类型再和c比较
7   System.out.println(b == c); // true
8
```

```
9   Integer a1 = 128;
10  Integer b1 = 128;
11  System.out.println(a1 == b1); // false
12
13  Integer a2 = 127;
14  Integer b2 = 127;
15  System.out.println(a2 == b2); // true
16  }
```

集合容器概述

什么是集合

集合框架：用于存储数据的容器。

集合框架是为表示和操作集合而规定的一种统一的标准的体系结构。 任何集合框架都包含三大块内容：对外的接口、接口的实现和对集合运算的算 法。

接口：表示集合的抽象数据类型。接口允许我们操作集合时不必关注具体实现， 从而达到“多态”。在面向对象编程语言中，接口通常用来形成规范。

实现：集合接口的具体实现，是重用性很高的数据结构。
算法：在一个实现了某个集合框架中的接口的对象身上完成某种有用的计算的方 法，例如查找、排序等。这些算法通常是多态的，因为相同的方法可以在同一个 接口被多个类实现时有不同的表现。事实上，算法是可复用的函数。 它减少了程序设计的辛劳。

集合框架通过提供有用的数据结构和算法使你能集中注意力于你的程序的重要部 分上，而不是为了让程序能正常运转而将注意力于底层设计上。

通过这些在无关API之间的简易的互用性，使你免除了为改编对象或转换代码以 便联合这些API而去写大量的代码。 它提高了程序速度和质量。

集合的特点

集合的特点主要有如下两点：

- 对象封装数据，对象多了也需要存储。集合用于存储对象。
- 对象的个数确定可以使用数组，对象的个数不确定的可以用集合。因 为集合是可变长度的。

集合和数组的区别

- 数组是固定长度的；集合可变长度的。
- 数组可以存储基本数据类型，也可以存储引用数据类型；集合只能存 储引用数据类型。
- 数组存储的元素必须是同一个数据类型；集合存储的对象可以是不同 数据类型。

数据结构：就是容器中存储数据的方式。

对于集合容器，有很多种。因为每一个容器的自身特点不同，其实原理在于每个 容器的内部数据结构不同。

集合容器在不断向上抽取过程中，出现了集合体系。**在使用一个体系的原则：参 阅顶层内容。建立底层对象。**

使用集合框架的好处

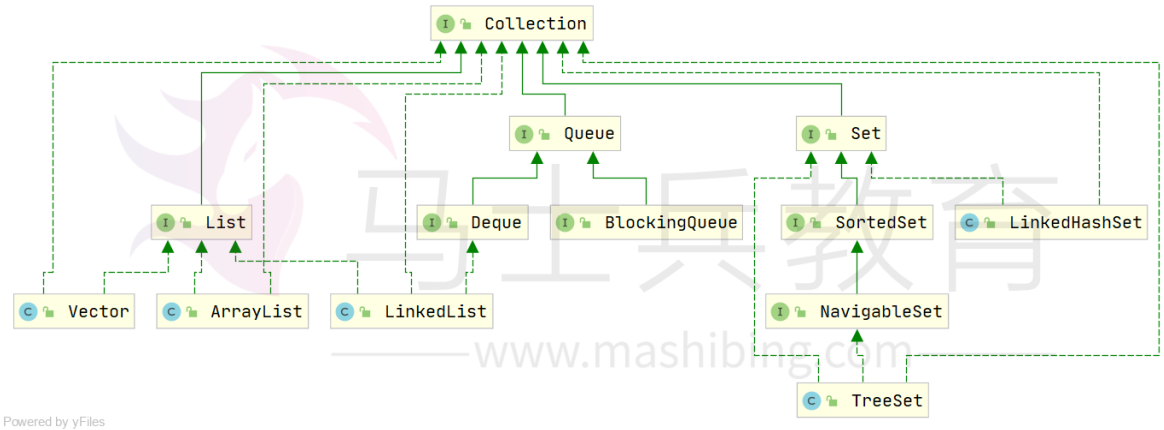
1. 容量自增长；
2. 提供了高性能的数据结构和算法，使编码更轻松，提高了程序速度和质量；
3. 允许不同 API 之间的互操作，API之间可以来回传递集合；
4. 可以方便地扩展或改写集合，提高代码复用性和可操作性。
5. 通过使用JDK自带的集合类，可以降低代码维护和学习新API成本。

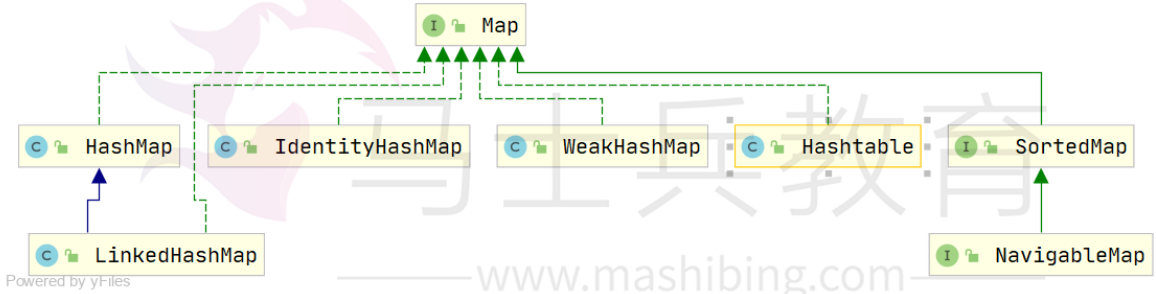
常用的集合类有哪些?

Map接口和Collection接口是所有集合框架的父接口：

1. Collection接口的子接口包括：Set接口和List接口
2. Map接口的实现类主要有：HashMap、TreeMap、Hashtable、 ConcurrentHashMap以及Properties等
3. Set接口的实现类主要有：HashSet、TreeSet、LinkedHashSet等
4. List接口的实现类主要有：ArrayList、LinkedList、Stack以及Vector等

List，Set，Map三者的区别? List、Set、Map 是否继 承自 Collection 接口? List、Map、Set 三个接口存取 元素时，各有什么特点?





Java 容器分为 Collection 和 Map 两大类，Collection集合的子接口有Set、 List、Queue三种子接口。我们比较常用的是Set、List，Map接口不是 collection的子接口。

Collection集合主要有List和Set两大接口

- List：一个有序（元素存入集合的顺序和取出的顺序一致）容器，元素可以重 复，可以插入多个null元素，元素都有索引。常用的实现类有 ArrayList、LinkedList 和 Vector。
- Set：一个无序（存入和取出顺序有可能不一致）容器，不可以存储重复元素， 只允许存入一个null元素，必须保证元素唯一性。Set 接口常用实现类是 HashSet、 LinkedHashSet 以及 TreeSet。

Map是一个键值对集合，存储键、值和之间的映射。 Key无序，唯一；value 不 要求有序，允许重复。Map没有继承于Collection接口，从Map集合中检索元 素时，只要给出键对象，就会返回对应的值对象。

Map 的常用实现类：HashMap、TreeMap、HashTable、LinkedHashMap、 ConcurrentHashMap

集合框架底层数据结构

Collection

List

- Arraylist： Object数组
- Vector： Object数组
- LinkedList： 双向循环链表

Set

- HashSet（无序，唯一）：基于 HashMap 实现的，底层采用 HashMap 来保存元素
- LinkedHashSet： LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的LinkedHashMap 其内部是基 于 Hashmap 实现一样，不过还是有一点点区别的。
- TreeSet（有序，唯一）： 红黑树(自平衡的排序二叉树。) Map
- HashMap： JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主 体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）.JDK1.8以后
  在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转 化为红黑树，以减少搜索时间
- LinkedHashMap：LinkedHashMap 继承自 HashMap，所以它的底层仍然是 基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面 结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。 同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。
- HashTable： 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为 了解决哈希冲突而存在的
- TreeMap： 红黑树（自平衡的排序二叉树）

哪些集合类是线程安全的?

- vector：就比arraylist多了个同步化机制（线程安全），因为效率较低，现在已 经不太建议使用。在web应用中，特别是前台页面，往往效率（页面响应速度）是优 先考虑的。
- statck：堆栈类，先进后出。
- hashtable：就比hashmap多了个线程安全。
- enumeration：枚举，相当于迭代器。

Java集合的快速失败机制 “fail-fast”?

是java集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作 时，有可能会产生 fail-fast 机制。

例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中 的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简 单的修改集合元素的内容），那么这个时候程序就会抛出ConcurrentModificationException 异常，从而产生fail-fast机制。

原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变modCount 的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测 modCount变量是否为expectedmodCount值，是的话就返回遍历；否则抛出异常，终止遍历。

解决办法：

1. 在遍历过程中，所有涉及到改变modCount值得地方全部加上 synchronized。
2. 使用CopyOnWriteArrayList来替换ArrayList

怎么确保一个集合不能被修改?

可以使用 Collections. unmodifiableCollection(Collection c) 方法来创建一个 只读集合，这样改变集合的任何操作都会抛出 Java. lang. UnsupportedOperationException 异常。 示例代码如下：

```
List<String> list = new ArrayList<>();
list. add("x");
Collection<String> clist = Collections. unmodifiableCollection(list);
clist. add("y"); // 运行时此行报错
System. out. println(list. size());
```

Collection接口

List接口

迭代器 Iterator 是什么?

Iterator 接口提供遍历任何 Collection 的接口。我们可以从一个 Collection 中使用迭代器方法来获取迭代器实例。迭代器取代了 Java 集合框架中的 Enumeration，迭代器允许调用者在迭代过程中移除元素。

Iterator 怎么使用？有什么特点?

Iterator 使用代码如下：

```
1   List<String> list = new ArrayList
2   Iterator<String> it = list. iterator
3   while(it. hasNext()){
4   String obj = it. next();
5   System. out. println(obj);
6   }
```

Iterator 的特点是只能单向遍历，但是更加安全，因为它可以确保，在当前遍历

的集合元素被更改的时候，就会抛出 ConcurrentModificationException 异常。

如何边遍历边移除 Collection 中的元素?

边遍历边修改 Collection 的唯一正确方式是使用 Iterator.remove() 方法，如下：

```
1   Iterator<Integer> it = list.iterator();
2   while(it.hasNext()){
3   *// do something*
4   it.remove();5   }
```

一种 常见的错误代码如下：

```
1   for(Integer i : list){
2   list.remove(i)
3   }
```

运行以上错误代码会报 ConcurrentModificationException 异常。这是因为当使用 foreach(for(Integer i : list)) 语句时，会自动生成一个iterator 来遍历该 list，但同时该 list 正在被 Iterator.remove() 修改。Java 一般不允许一个线程在遍历 Collection 时另一个线程修改它。

Iterator 和 ListIterator 有什么区别?

- Iterator 可以遍历 Set 和 List 集合，而 ListIterator 只能遍历 List。
- Iterator 只能单向遍历，而 ListIterator 可以双向遍历（向前/后遍历）。
- ListIterator 实现 Iterator 接口，然后添加了一些额外的功能，比如添加一个元 素、替换一个元素、获取前面或后面元素的索引位置。

遍历一个 List 有哪些不同的方式？每种方法的实现原理是什 么？Java 中 List 遍历的最佳实践是什么?

遍历方式有以下几种：

1. for 循环遍历，基于计数器。在集合外部维护一个计数器，然后依次读 取每一个位置的元素，当读取到后一个元素后停止。
2. 迭代器遍历，Iterator。Iterator 是面向对象的一个设计模式，目的是屏 蔽不同数据集合的特点，统一遍历集合的接口。Java 在 Collections 中支 持了 Iterator 模式。
3. foreach 循环遍历。foreach 内部也是采用了 Iterator 的方式实现，使 用时不需要显式声明 Iterator 或计数器。优点是代码简洁，不易出错；缺 点是只能做简单的遍历，不能在遍历过程中操作数据集合，例如删除、替 换。

最佳实践：Java Collections 框架中提供了一个 RandomAccess 接口，用来标 记 List 实现是否支持 Random Access。

- 如果一个数据集合实现了该接口，就意味着它支持 Random Access，按位置读 取元素的平均时间复杂度为 O(1)，如ArrayList。
- 如果没有实现该接口，表示不支持 Random Access，如LinkedList。 推荐的做法就是，支持 Random Access 的列表可用 for 循环遍历，否则建议 用 Iterator 或 foreach 遍历。

说一下 ArrayList 的优缺点

ArrayList的优点如下：

- ArrayList 底层以数组实现，是一种随机访问模式。ArrayList 实现了 RandomAccess 接口，因此查找的时候非常快。
- ArrayList 在顺序添加一个元素的时候非常方便。

ArrayList 的缺点如下：

- 删除元素的时候，需要做一次元素复制操作。如果要复制的元素很多，那么就会比较耗费性能。
- 插入元素的时候，也需要做一次元素复制操作，缺点同上。

ArrayList 比较适合顺序添加、随机访问的场景。

如何实现数组和 List 之间的转换?

数组转 List：使用 Arrays. asList(array) 进行转换。

List 转数组：使用 List 自带的 toArray() 方法。代码示例：

```
// list to array
List<String> list = new ArrayList<String>();
list.add("123");
list.add("456");
list.toArray();

// array to list
String[] array = new String[]{"123","456"};
Arrays.asList(array);
```

ArrayList 和 LinkedList 的区别是什么?

- 数据结构实现：ArrayList 是动态数组的数据结构实现，而 LinkedList 是双向链表的数据结构实现。
- 随机访问效率：ArrayList 比 LinkedList 在随机访问的时候效率要高，因为 LinkedList 是线性的数据存储方式，所以需要移动指针从前往后依次查找。
- 增加和删除效率：在非首尾的增加和删除操作，LinkedList 要比 ArrayList 效率要高，因为 ArrayList 增删操作要影响数组内的其他数据的下标。
- 内存空间占用：LinkedList 比 ArrayList 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。
- 线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

综合来说，在需要频繁读取集合中的元素时，更推荐使用 ArrayList，而在插入和删除操作较多时，更推荐使用 LinkedList。

补充：数据结构基础之双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。

ArrayList 和 Vector 的区别是什么?

这两个类都实现了 List 接口（List 接口继承了 Collection 接口），他们都是有序集合

```
线程安全：Vector 使用了 Synchronized 来实现线程同步，是线程安全的，而
```

ArrayList 是非线程安全的。

性能：ArrayList 在性能方面要优于 Vector。

扩容：ArrayList 和 Vector 都会根据实际的需要动态的调整容量，只不过在

Vector 扩容每次会增加 1 倍，而 ArrayList 只会增加 50%。

Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对

象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。

Arraylist不是同步的，所以在不需要保证线程安全时时建议使用Arraylist。

插入数据时，ArrayList、LinkedList、Vector谁速度较快？阐述 ArrayList、Vector、LinkedList 的存储性能和特性?

ArrayList、LinkedList、Vector 底层的实现都是使用数组方式存储数据。数组

元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢。

Vector 中的方法由于加了 synchronized 修饰，因此 Vector是线程安全容器，但性能上较ArrayList差。

LinkedList 使用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，但插入数据时只需要记录当前项的前后项即可，所以 LinkedList插入速度较快。

多线程场景下如何使用 ArrayList?

ArrayList 不是线程安全的，如果遇到多线程场景，可以通过 Collections 的

synchronizedList 方法将其转换成线程安全的容器后再使用。例如像下面这样：

```
1   List<String> synchronizedList = Collections.synchronizedList(list);
2   synchronizedList.add("aaa");
3   synchronizedList.add("bbb");
4
5   for (int i = 0; i < synchronizedList.size(); i++) {
6   System.out.println(synchronizedList.get(i));
7   }
```

为什么 ArrayList 的 elementData 加上 transient 修饰？ ArrayList 中的数组定义如下：

```
1 private transient Object[] elementData;
```

再看一下 ArrayList 的定义：

```
1   public class ArrayList<E> extends AbstractList<E>
2   implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

可以看到 ArrayList 实现了 Serializable 接口，这意味着 ArrayList 支持序列

化。transient 的作用是说不希望 elementData 数组被序列化，重写了 writeObject 实现：

```
1 private void writeObject(java.io.ObjectOutputStream s) throws java.io.IOE
xception{
2  *// Write out element count, and any hidden stuff*
3  int expectedModCount = modCount;
4  s.defaultWriteObject();
5  *// Write out array length*
6  s.writeInt(elementData.length);
7  *// Write out all elements in the proper order.*
8  for (int i=0; i<size; i++)
9  s.writeObject(elementData[i]);
10  if (modCount != expectedModCount) {
11  throw new ConcurrentModificationException();
12 }
```

每次序列化时，先调用 defaultWriteObject() 方法序列化 ArrayList 中的非transient 元素，然后遍历 elementData，只序列化已存入的元素，这样既加快了序列化的速度，又减小了序列化之后的文件大小。

List 和 Set 的区别

List , Set 都是继承自Collection 接口

List 特点：一个有序（元素存入集合的顺序和取出的顺序一致）容器，元素可以重复，可以插入多个null元素，元素都有索引。常用的实现类有 ArrayList、

LinkedList 和 Vector。

Set 特点：一个无序（存入和取出顺序有可能不一致）容器，不可以存储重复元素，只允许存入一个null元素，必须保证元素唯一性。Set 接口常用实现类是

HashSet、LinkedHashSet 以及 TreeSet。

另外 List 支持for循环，也就是通过下标来遍历，也可以用迭代器，但是set只能用迭代，因为他无序，无法用下标来取得想要的值。

Set和List对比

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

List：和数组类似，List可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变

Set接口

说一下 HashSet 的实现原理?

HashSet 是基于 HashMap 实现的，HashSet的值存放于HashMap的key上，HashMap的value统一为PRESENT，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，HashSet 不允许重复的值。

HashSet如何检查重复？HashSet是如何保证数据不可重复的?

向HashSet 中add ()元素时，判断元素是否存在的依据，不仅要比较hash值，同时还要结合equles 方法比较。

HashSet 中的add ()方法会使用HashMap 的put()方法。

HashMap 的 key 是唯一的，由源码可以看出 HashSet 添加进去的值就是作为 HashMap 的key，并且在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V。所以不会重复（ HashMap 比较key是否相等是先比较 hashcode 再比较equals ）。

以下是HashSet 部分源码：

```
1   private static final Object PRESENT = new Object();
2   private transient HashMap<E,Object> map;
3
4   public HashSet() {
<>
5   map = new HashMap ();
6   }
7
8   public boolean add(E e) {
9   // 调用HashMap的put方法,PRESENT是一个至始至终都相同的虚值
10  return map.put(e, PRESENT)==null;
11  }
```

hashCode（）与equals（）的相关规定：

1. 如果两个对象相等，则hashcode一定也是相同的
2. 两个对象相等,对两个equals方法返回true
3. 两个对象有相同的hashcode值，它们也不一定是相等的
4. 综上，equals方法被覆盖过，则hashCode方法也必须被覆盖
5. hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

==与equals的区别

1. ==是判断两个变量或实例是不是指向同一个内存空间 equals是判断两个变量或实例所指向的内存空间的值是不是相同
2. ==是指对内存地址进行比较 equals()是对字符串的内容进行比较3.== 指引用是否相同 equals()指的是值是否相同

HashSet与HashMap的区别

HashMap	HashSet
实现了Map接口	实现了Set接口
存储键值对	仅存储对象
调用 put（）向 map中添加元素	调用 add（） 方法向Set 中添加元素
HashMap 使用键（Key）计算 Hashcode	HashSet 使用成员对象来计 算 hashcode 值，对于两个对象 来说 hashcode 可能相 同，所以 equals()方法用来判断对象的相等性，如果两个对象不同的话，那 么返回 false
HashMap 相对于 HashSet 较快，因为它是使用唯一的键获取对象	HashSet 较 HashMap 来说比较慢

Queue

BlockingQueue是什么?

Java.util.concurrent.BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间。BlockingQueue接口是Java集合框架的一部分，主要用于实现生产者-消费者模式。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在BlockingQueue的实现类中被处理了。Java提供了集中 BlockingQueue的实现，比如ArrayBlockingQueue、

LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue等。在 Queue 中 poll()和 remove()有什么区别?

- 相同点：都是返回第一个元素，并在队列中删除返回的对象。
- 不同点：如果没有元素 poll()会返回 null，而 remove()会直接抛出 NoSuchElementException 异常。

代码示例：

```
1   Queue<String> queue = new LinkedList<String>();
2   queue. offer("string"); // add
3   System. out. println(queue. poll());
4   System. out. println(queue. remove());
5   System. out. println(queue. size());
```

Map接口

说一下 HashMap 的实现原理?

HashMap概述： HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap的数据结构： 在Java编程语言中， 基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

HashMap 基于 Hash 算法实现的

1. 当我们往Hashmap中put元素时，利用key的hashCode重新hash计算出当前对象的元素在数组中的下标
2. 存储时，如果出现hash值相同的key，此时有两种情况。(1)如果key相

同，则覆盖原始值；(2)如果key不同（出现冲突），则将当前的key-value 放入链表中

3. 获取时，直接找到hash值对应的下标，在进一步判断key是否相同，从而找到对应值。
4. 理解了以上过程就不难明白HashMap是如何解决hash冲突的问题，核心就是使用了数组的存储方式，然后将冲突的key的对象放入链表中，一旦发现冲突就在链表中做进一步的对比。

需要注意Jdk 1.8中对HashMap的实现做了优化，当链表中的节点数据超过八个

之后，该链表会转为红黑树来提高查询效率，从原来的O(n)到O(logn)

HashMap在JDK1.7和JDK1.8中有哪些不同？ HashMap的底层实现

在Java中，保存数据有两种比较简单的数据结构：数组和链表。数组的特点是：寻址容易，插入和删除困难；链表的特点是：寻址困难，但插入和删除容易；所以我们将数组和链表结合在一起，发挥两者各自的优势，使用一种叫做拉链法的方式可以解决哈希冲突。

JDK1.8之前

JDK1.8之前采用的是拉链法。拉链法：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

JDK1.8之后

相比于之前的版本，jdk1.8在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。

JDK1.7 VS JDK1.8 比较

JDK1.8主要解决或优化了一下问题：

1. resize 扩容优化
2. 引入了红黑树，目的是避免单条链表过长而影响查询效率，红黑树算法请参考
3. 解决了多线程死循环问题，但仍是非线程安全的，多线程时可能会造成数据丢失问题。

不同	JDK 1.7	JDK 1.8
存储结构	数组 + 链表	数组 + 链表 + 红黑树
初始化方式	单独函数： inflateTab le()	直接集成到了扩容 函数 resize()中
hash值计算方式	扰动处理 = 9次扰动 = 4次位运算 + 5次异或运算	扰动处理 = 2次扰动 = 1次位运算 + 1次异或运算
存放数据的规则	无冲突 时，存放 数组；冲 突时，存 放链表	无冲突时，存放 数组；冲 突 & 链表 长度 < 8：存放单 链表；冲 突 & 链表 长度 > 8：树化并 存放红黑 树
插入数据方式	头插法 （先讲原 位置的数 据移到后1 位，再插 入数据到 该位置）	尾插法 （直接插 入到链表 尾部/红黑 树）
扩容后存储位置的计算方式	全部按照 原来方法 进行计算 （即 hashCode ->> 扰动 函数 ->> (h&lengt h-1)）	按照扩容 后的规律 计算（即 扩容后的 位置= 原位 置 or 原位 置 + 旧容 量）

HashMap的put方法的具体流程?

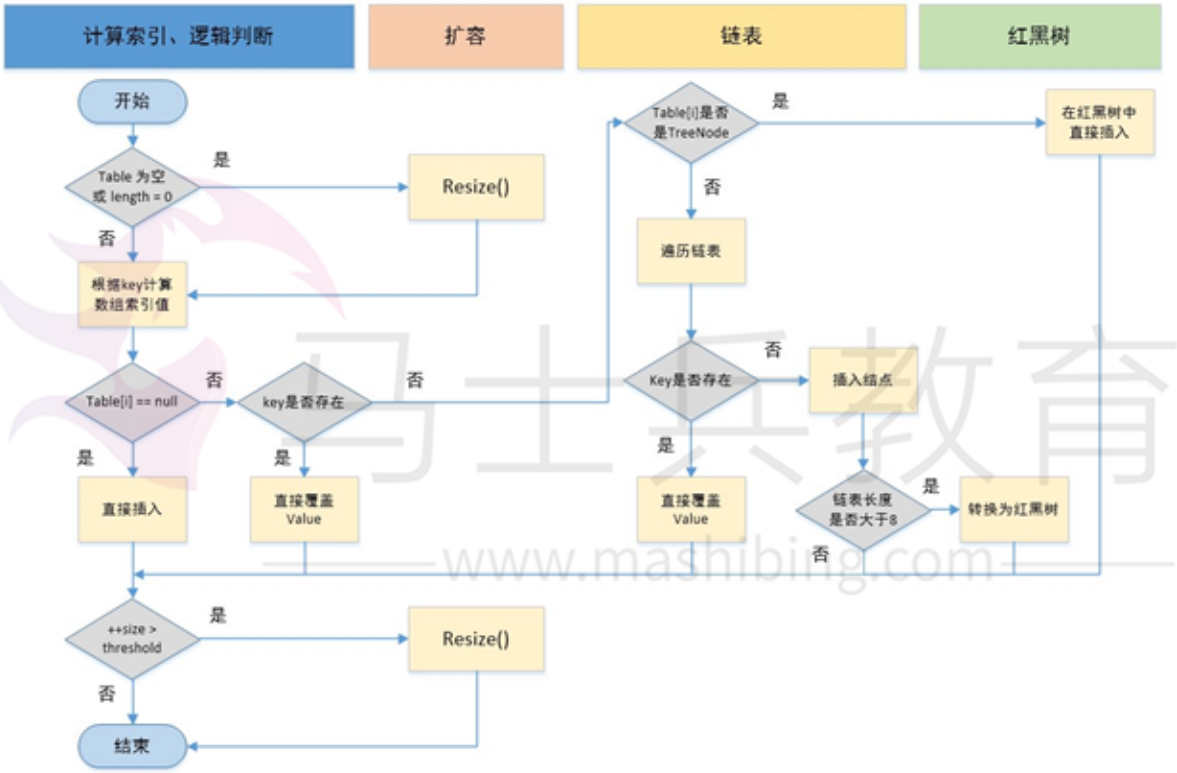
当我们put的时候，首先计算 key的hash值，这里调用了 hash方法，hash方法实际是让key.hashCode()与key.hashCode()>>>16进行异或操作，高16bit补0，一个数和0异或不变，所以 hash 函数大概的作用就是：高16bit不变，低16bit和高

16bit做了一个异或，目的是减少碰撞。按照函数注释，因为bucket数组大小是

2的幂，计算下标index = (table.length - 1) & hash，如果不做 hash 处理，相当于散列生效的只有几个低 bit 位，为了减少散列的碰撞，设计者综合考虑了速度、作用、质量之后，使用高16bit和低16bit异或来简单处理减少碰撞，而且

JDK8中用了复杂度 O（logn）的树结构来提升碰撞下的性能。

putVal方法执行流程图




```
1 public V put(K key, V value) {
2 return putVal(hash(key), key, value, false, true);
3 }
4
5 static final int hash(Object key) {
6 int h;
7 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
8 }
9
10 //实现Map.put和相关方法
11 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
12 boolean evict) {
13 Node<K,V>[] tab; Node<K,V> p; int n, i;
14 // 步骤①：tab为空则创建
15 // table未初始化或者长度为0，进行扩容
16 if ((tab = table) == null || (n = tab.length) == 0)
17 n = (tab = resize()).length;
18 // 步骤②：计算index，并对null做处理
19 // (n - 1) & hash 确定元素存放在哪个桶中，桶为空，新生成结点放入桶中(此时，这
个结点是放在数组中)
20 if ((p = tab[i = (n - 1) & hash]) == null)
21 tab[i] = newNode(hash, key, value, null);
22 // 桶中已经存在元素
23 else {
24 Node<K,V> e; K k;
25 // 步骤③：节点key存在，直接覆盖value
26 // 比较桶中第一个元素(数组中的结点)的hash值相等，key相等
27 if (p.hash == hash &&
28 ((k = p.key) == key || (key != null && key.equals(k))))
29 // 将第一个元素赋值给e，用e来记录
30 e = p;
31 // 步骤④：判断该链为红黑树
32 // hash值不相等，即key不相等；为红黑树结点
33 // 如果当前元素类型为TreeNode，表示为红黑树，putTreeVal返回待存放的node, e可
```

```
能为null
34 else if (p instanceof TreeNode)
35 // 放入树中
36 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
37 // 步骤⑥：该链为链表
38 // 为链表结点
39 else {
40 // 在链表最末插入结点
41 for (int binCount = 0; ; ++binCount) {
42 // 到达链表的尾部
43
44 //判断该链表尾部指针是不是空的
45 if ((e = p.next) == null) {
46 // 在尾部插入新结点
47 p.next = newNode(hash, key, value, null);
48 //判断链表的长度是否达到转化红黑树的临界值，临界值为8
49 if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
50 //链表结构转树形结构
51 treeifyBin(tab, hash);
52 // 跳出循环
53 break;
54 }
55 // 判断链表中结点的key值与插入的元素的key值是否相等
56 if (e.hash == hash &&
57 ((k = e.key) == key || (key != null && key.equals(k))))
58 // 相等，跳出循环
59 break;
60 // 用于遍历桶中的链表，与前面的e = p.next组合，可以遍历链表
61 p = e;
62 }
63 }
64 //判断当前的key已经存在的情况下，再来一个相同的hash值、key值时，返回新来的val
ue这个值
65 if (e != null) {
66 // 记录e的value
67 V oldValue = e.value;
68 // onlyIfAbsent为false或者旧值为null
69 if (!onlyIfAbsent || oldValue == null)
70 //用新值替换旧值
71 e.value = value;
72 // 访问后回调
73 afterNodeAccess(e);
74 // 返回旧值
75 return oldValue;
76 }
77 }
78 // 结构性修改
79 ++modCount;
80 // 步骤⑥：超过最大容量就扩容
81 // 实际大小大于阈值则扩容
82 if (++size > threshold)
83 resize();
84 // 插入后回调
85 afterNodeInsertion(evict);
86 return null;
87 }
```

①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向

④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了 大容量threshold，如果超过，进行扩容。

HashMap的扩容操作是怎么实现的?

①.在jdk1.8中，resize方法是在hashmap中的键值对大于阀值时或者初始化时，就调用resize方法进行扩容；

②.每次扩展的时候，都是扩展2倍；

③.扩展后Node对象的位置要么在原位置，要么移动到原偏移量两倍的位置。在putVal()中，我们看到在这个函数里面使用到了2次resize()方法，resize()方法表示的在进行第一次初始化时会对其进行扩容，或者当该数组的实际大小大于其临界值值(第一次为12),这个时候在扩容的同时也会伴随的桶上面的元素进行重新分发，这也是JDK1.8版本的一个优化的地方，在1.7中，扩容之后需要重新去计算其Hash值，根据Hash值对其进行分发，但在1.8版本中，则是根据在同一个桶的位置中进行判断(e.hash & oldCap)是否为0，重新进行hash分配后，该元素的位置要么停留在原始位置，要么移动到原始位置+增加的数组大小这个位置上

```
1 final Node<K,V>[] resize() {
2 Node<K,V>[] oldTab = table;//oldTab指向hash桶数组
3 int oldCap = (oldTab == null) ? 0 : oldTab.length;
4 int oldThr = threshold;
5 int newCap, newThr = 0;
6 if (oldCap > 0) {//如果oldCap不为空的话，就是hash桶数组不为空
7 if (oldCap >= MAXIMUM_CAPACITY) {//如果大于最大容量了，就赋值为整数最大的阀
值
8 threshold = Integer.MAX_VALUE;
9 return oldTab;//返回
10 }//如果当前hash桶数组的长度在扩容后仍然小于最大容量 并且oldCap大于默认值16
11 else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
12 oldCap >= DEFAULT_INITIAL_CAPACITY)
13 newThr = oldThr << 1; // double threshold 双倍扩容阀值threshold
14 }
15 // 旧的容量为0，但threshold大于零，代表有参构造有cap传入，threshold已经被初
始化成最小2的n次幂
16 // 直接将该值赋给新的容量
17 else if (oldThr > 0) // initial capacity was placed in threshold
18 newCap = oldThr;
19 // 无参构造创建的map，给出默认容量和threshold 16, 16*0.75
20 else { // zero initial threshold signifies using defaults
21 newCap = DEFAULT_INITIAL_CAPACITY;
22 newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
23 }
24 // 新的threshold = 新的cap * 0.75
25 if (newThr == 0) {
```

```
26 float ft = (float)newCap * loadFactor;
27 newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
28 (int)ft : Integer.MAX_VALUE);
29 }
30 threshold = newThr;
31 // 计算出新的数组长度后赋给当前成员变量table
32 @SuppressWarnings({"rawtypes","unchecked"})
33 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];//新建hash桶数组
34 table = newTab;//将新数组的值复制给旧的hash桶数组
35 // 如果原先的数组没有初始化，那么resize的初始化工作到此结束，否则进入扩容元素
重排逻辑，使其均匀的分散
36 if (oldTab != null) {
37 // 遍历新数组的所有桶下标
38 for (int j = 0; j < oldCap; ++j) {
39 Node<K,V> e;
40 if ((e = oldTab[j]) != null) {
41 // 旧数组的桶下标赋给临时变量e，并且解除旧数组中的引用，否则就数组无法被GC回收
42 oldTab[j] = null;
43 // 如果e.next==null，代表桶中就一个元素，不存在链表或者红黑树
44 if (e.next == null)
45 // 用同样的hash映射算法把该元素加入新的数组
46 newTab[e.hash & (newCap - 1)] = e;
47 // 如果e是TreeNode并且e.next!=null，那么处理树中元素的重排
48 else if (e instanceof TreeNode)
49 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
50 // e是链表的头并且e.next!=null，那么处理链表中元素重排
51 else { // preserve order
52 // loHead,loTail 代表扩容后不用变换下标，见注1
53 Node<K,V> loHead = null, loTail = null;
54 // hiHead,hiTail 代表扩容后变换下标，见注1
55 Node<K,V> hiHead = null, hiTail = null;
56 Node<K,V> next;
57 // 遍历链表
58 do {
59 next = e.next;
60 if ((e.hash & oldCap) == 0) {
61 if (loTail == null)
62 // 初始化head指向链表当前元素e，e不一定是链表的第一个元素，初始化后loHead
63 // 代表下标保持不变的链表的头元素
64 loHead = e;
65 else
66 // loTail.next指向当前e
67 loTail.next = e;
68 // loTail指向当前的元素e
69 // 初始化后，loTail和loHead指向相同的内存，所以当loTail.next指向下一个元素
时，
70 // 底层数组中的元素的next引用也相应发生变化，造成lowHead.next.next.....
71 // 跟随loTail同步，使得lowHead可以链接到所有属于该链表的元素。
72 loTail = e;
73 }
74 else {
75 if (hiTail == null)
76 // 初始化head指向链表当前元素e，初始化后hiHead代表下标更改的链表头元素
77 hiHead = e;
78 else
79 hiTail.next = e;
80 hiTail = e;
81 }
```

```
82 } while ((e = next) != null);
83 // 遍历结束，将tail指向null，并把链表头放入新数组的相应下标，形成新的映射。
84 if (loTail != null) {
85 loTail.next = null;
86 newTab[j] = loHead;
87 }
88 if (hiTail != null) {
89 hiTail.next = null;
90 newTab[j + oldCap] = hiHead;
91 }
92 }
93 }
94 }
95 }
96 return newTab;
97 }
```

HashMap是怎么解决哈希冲突的?

答：在解决这个问题之前，我们首先需要知道什么是哈希冲突，而在了解哈希冲突之前我们还要知道什么是哈希才行；什么是哈希？

Hash，一般翻译为“散列”，也有直接音译为“哈希”的，这就是把任意长度的输入通过散列算法，变换成固定长度的输出，该输出就是散列值（哈希值）；这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

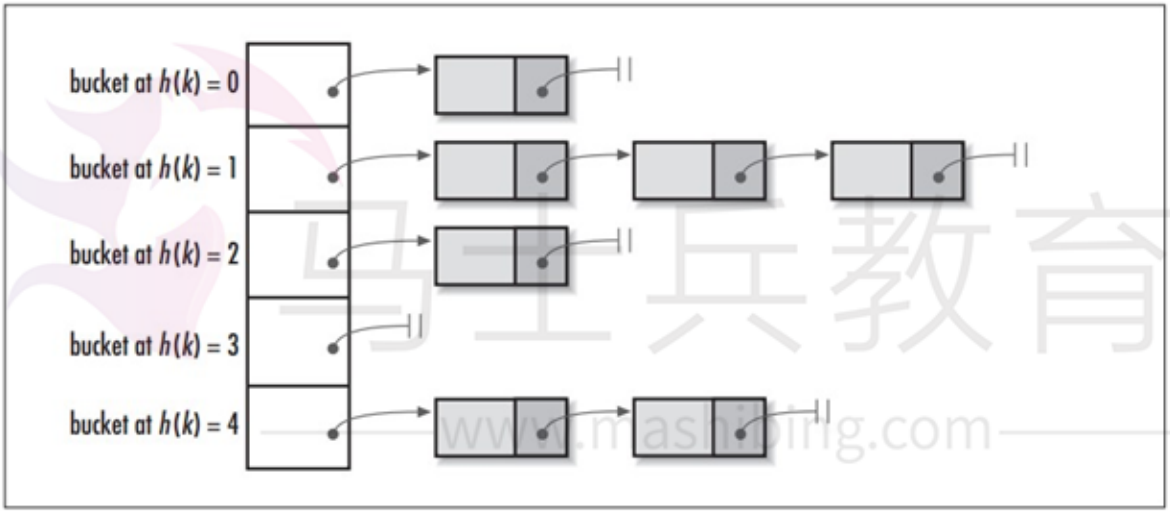
所有散列函数都有如下一个基本特性：**根据同一散列函数计算出的散列值如果不同，那么输入值肯定也不同。但是，根据同一散列函数计算出的散列值如果相同，输入值不一定相同。**

什么是哈希冲突?

当两个不同的输入值，根据同一散列函数计算出相同的散列值的现象，我们就把它叫做碰撞（哈希碰撞）。

HashMap的数据结构

在Java中，保存数据有两种比较简单的数据结构：数组和链表。数组的特点是：寻址容易，插入和删除困难；链表的特点是：寻址困难，但插入和删除容易；所以我们将数组和链表结合在一起，发挥两者各自的优势，使用一种叫做链地址法的方式可以解决哈希冲突：



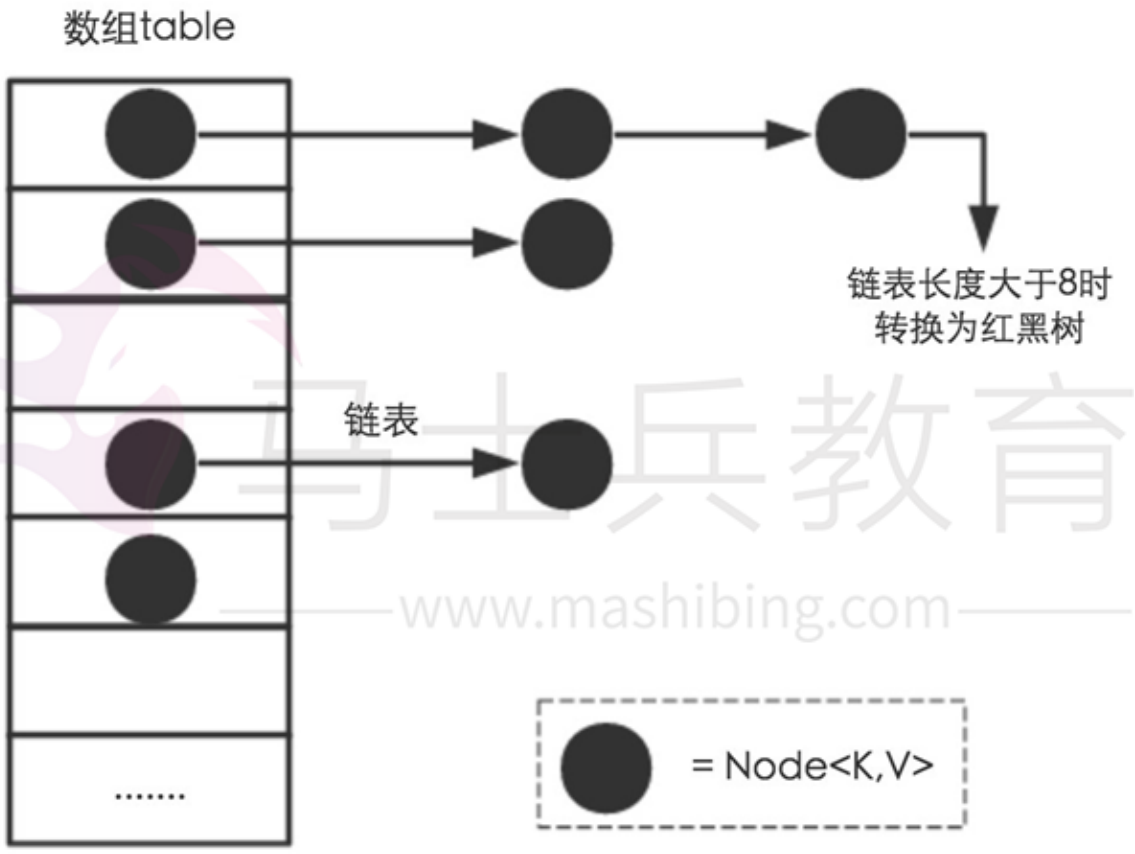
这样我们就可以将拥有相同哈希值的对象(img)组织成一个链表放在hash值所对应的 bucket下，但相比于hashCode返回的int类型，我们HashMap初始的容量大小DEFAULT_INITIAL_CAPACITY = 1 << 4（即2的四次方16）要远小于int类型的范围，所以我们如果只是单纯的用hashCode取余来获取对应的bucket这将会大大增加哈希碰撞的概率，并且最坏情况下还会将HashMap变成一个单链表，所以我们还需要对hashCode作一定的优化 hash()函数

上面提到的问题，主要是因为如果使用hashCode取余，那么相当于参与运算的只有hashCode的低位，高位是没有起到任何作用的，所以我们的思路就是让 hashCode取值出的高位也参与运算，进一步降低hash碰撞的概率，使得数据分布更平均，我们把这样的操作称为扰动，在JDK 1.8中的hash()函数如下：

```
1   static final int hash(Object key) {
2   int h;
3   return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);// 与自己右移16位进行异或运算（高低位异或）
4   }
```

这比在JDK 1.7中，更为简洁，相比在1.7中的4次位运算，5次异或运算（9次扰动），在1.8中，只进行了1次位运算和1次异或运算（2次扰动）；

JDK1.8新增红黑树



马士兵教育
www.mashibing.com

通过上面的链地址法（使用散列表）和扰(img)动函数我们成功让我们的数据分布更平均，哈希碰撞减少，但是当我们的HashMap中存在大量数据时，加入我们某个 bucket下对应的链表有n个元素，那么遍历时间复杂度就为O(n)，为了针对这个问题，JDK1.8在HashMap中新增了红黑树的数据结构，进一步使得遍历复杂度降低至O(logn)；总结

简单总结一下HashMap是使用了哪些方法来有效解决哈希冲突的：

1. 使用链地址法（使用散列表）来链接拥有相同hash值的数据；
2. 使用2次扰动函数（hash函数）来降低哈希冲突的概率，使得数据分布更平均；
3. 引入红黑树进一步降低遍历的时间复杂度，使得遍历更快；

能否使用任何类作为 Map 的 key?

可以使用任何类作为 Map 的 key，然而在使用之前，需要考虑以下几点： 如果类重写了 equals() 方法，也应该重写 hashCode() 方法。类的所有实例需要遵循与 equals() 和 hashCode() 相关的规则。

如果一个类没有使用 equals()，不应该在 hashCode() 中使用它。

用户自定义 Key 类 佳实践是使之为不可变的，这样 hashCode() 值可以被缓存起来，拥有更好的性能。不可变的类也可以确保 hashCode() 和 equals() 在未来不会改变，这样就会解决与可变相关的问题了。

为什么HashMap中String、Integer这样的包装类适合作为K?

答：String、Integer等包装类的特性能够保证Hash值的不可更改性和计算准确性，能够有效的减少Hash碰撞的几率

1. 都是final类型，即不可变性，保证key的不可更改性，不会存在获取 hash值不同的情况

内部已重写了equals()、hashCode()等方法，遵守了HashMap内部的规范（不清楚可以去上面看看putValue的过程），不容易出现Hash值计算错误的情况

如果使用Object作为HashMap的Key，应该怎么办呢?

答：重写hashCode()和equals()方法

1. 重写hashCode()是因为需要计算存储数据的存储位置，需要注意不要试图从散列码计算中排除掉一个对象的关键部分来提高性能，这样虽然能更快但可能会导致更多的Hash碰撞；
2. 重写equals()方法，需要遵守自反性、对称性、传递性、一致性以及对

于任何非null的引用值x，x.equals(null)必须返回false的这几个特性，目的是为了保证key在哈希表中的唯一性

HashMap为什么不直接使用hashCode()处理后的哈希 值直接作为table的下标?

答：hashCode()方法返回的是int整数类型，其范围为-(2 ^ 31)~(2 ^ 31 - 1)，约有40亿个映射空间，而HashMap的容量范围是在16（初始化默认值）~2 ^ 30，HashMap通常情况下是取不到 大值的，并且设备上也难以提供这么多的存储空间，从而导致通过hashCode()计算出的哈希值可能不在数组大小范围内，进而无法匹配存储位置；

那怎么解决呢?

1. HashMap自己实现了自己的hash()方法，通过两次扰动使得它自己的哈希值高低位自行进行异或运算，降低哈希碰撞概率也使得数据分布更平均；
2. 在保证数组长度为2的幂次方的时候，使用hash()运算之后的值与运算

（&）（数组长度 - 1）来获取数组下标的方式进行存储，这样一来是比取

余操作更加有效率，二来也是因为只有当数组长度为2的幂次方时，h&

(length-1)才等价于h%length，三来解决了“哈希值与数组大小范围不匹配"的问题

HashMap 的长度为什么是2的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀，每个链表/红黑树长度大致相同。这个实现就是把数据存到哪个链表/红黑树中的算法。

这个算法应该如何设计呢？我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操

作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说

hash%length==hash&(length-1)的前提是 length 是2的 n 次方；）。” 并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

那为什么是两次扰动呢？答：这样就是加大哈希值低位的随机性，使得分布更均匀，从而提高对应数组存储下标位置的随机性&均匀性， 终减少Hash冲突，两次就够了，已经达到了高位低位同时参与运算的目的

HashMap 与 HashTable 有什么区别?

1. 线程安全： HashMap 是非线程安全的，HashTable 是线程安全的；

HashTable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

2. 效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；
3. 对Null key 和Null value的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。但是在

HashTable 中 put 进的键值只要有一个 null，直接抛

NullPointerException。

4. **初始容量大小和每次扩充容量大小的不同** ： ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小。也就是说 HashMap 总是使用2的幂作为哈希表的大小，后面会介绍到为什么是2 的幂次方。
5. 底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。
6. 推荐使用：在 Hashtable 的类注释可以看到，Hashtable 是保留类不建议使用，推荐在单线程环境下使用 HashMap 替代，如果需要多线程使用则用 ConcurrentHashMap 替代。

如何决定使用 HashMap 还是TreeMap?

对于在Map中插入、删除和定位元素这类操作，HashMap是 好的选择。然

而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历

HashMap 和 ConcurrentHashMap 的区别

1. ConcurrentHashMap对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用lock锁进行保护，相对于HashTable的synchronized 锁的粒度更精细了一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。（JDK1.8之后ConcurrentHashMap启了一种全新的方式实现,利用CAS算法。）
2. HashMap的键值对允许有null，但是ConCurrentHashMap都不允许。

ConcurrentHashMap 和 Hashtable 的区别?

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

底层数据结构： JDK1.7的 ConcurrentHashMap 底层采用 分段的数组

+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑

二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

```
实现线程安全的方式（重要）： ① 在JDK1.7的时候，
```

ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配16个Segment，比Hashtable效率提高16

倍。） 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用

Node 数组+链表+红黑树的数据结构来实现，并发控制使用

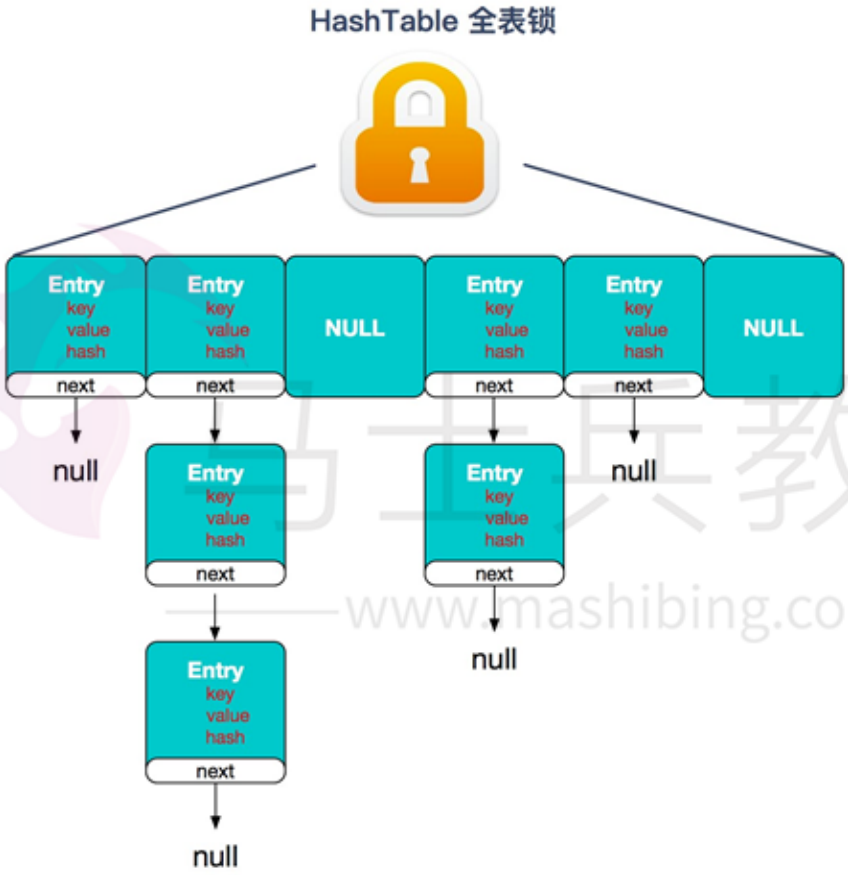
synchronized 和 CAS 来操作。 （JDK1.6以后 对 synchronized锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还

能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；②

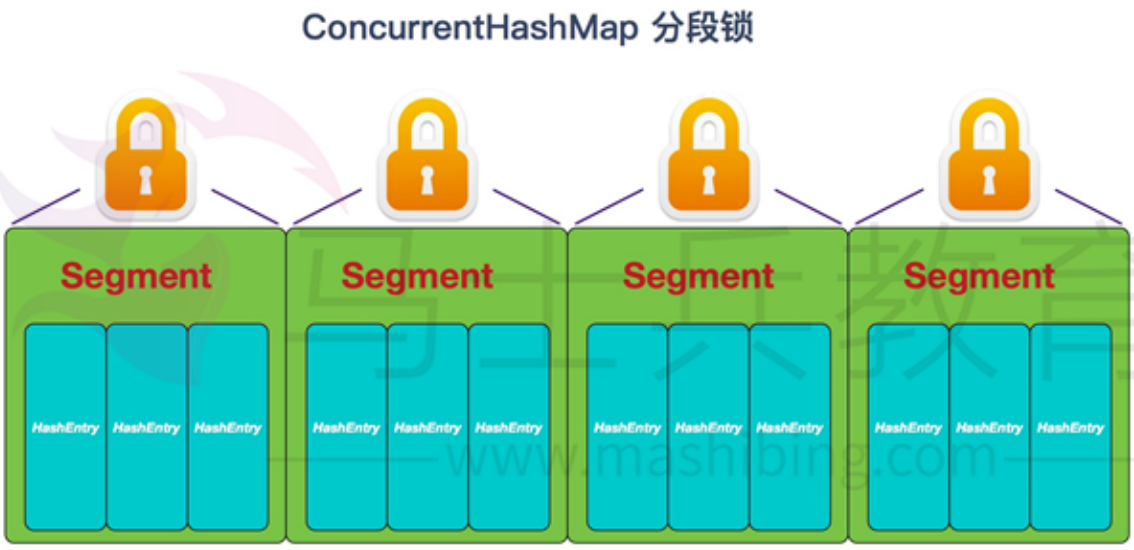
Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

两者的对比图：

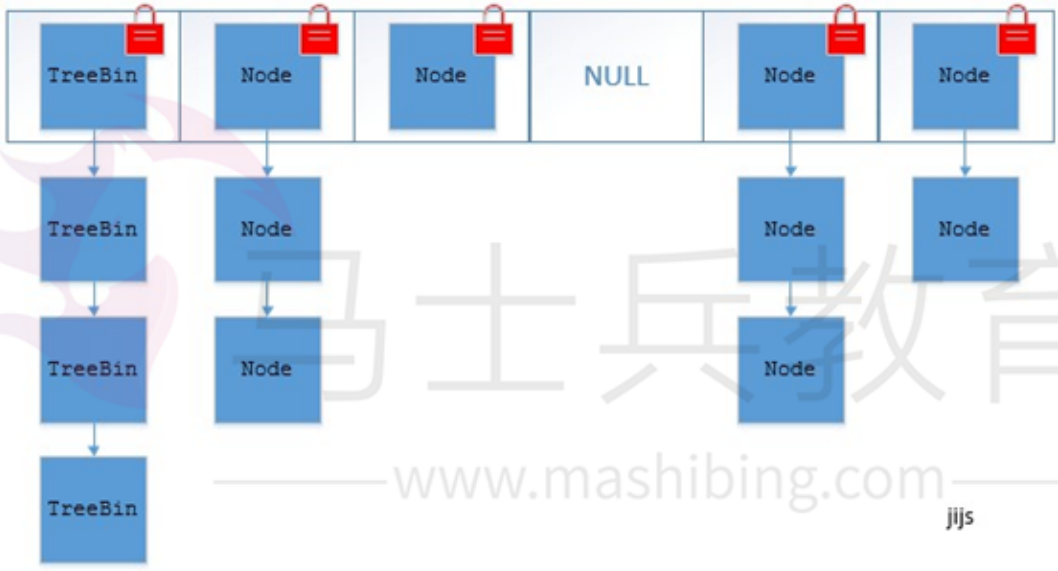
HashTable:



JDK1.7的ConcurrentHashMap：



JDK1.8的ConcurrentHashMap（TreeBi(img)n: 红黑二叉树节点 Node: 链表节点）：



答：ConcurrentHashMap 结合了 Hash(img)Map 和 HashTable 二者的优势。 HashMap 没有考虑同步，HashTable 考虑了同步的问题。但是 HashTable 在每次同步执行时都要锁住整个结构。 ConcurrentHashMap 锁的方式是稍微细粒度的。

ConcurrentHashMap 底层具体实现知道吗？实现原理是什么？

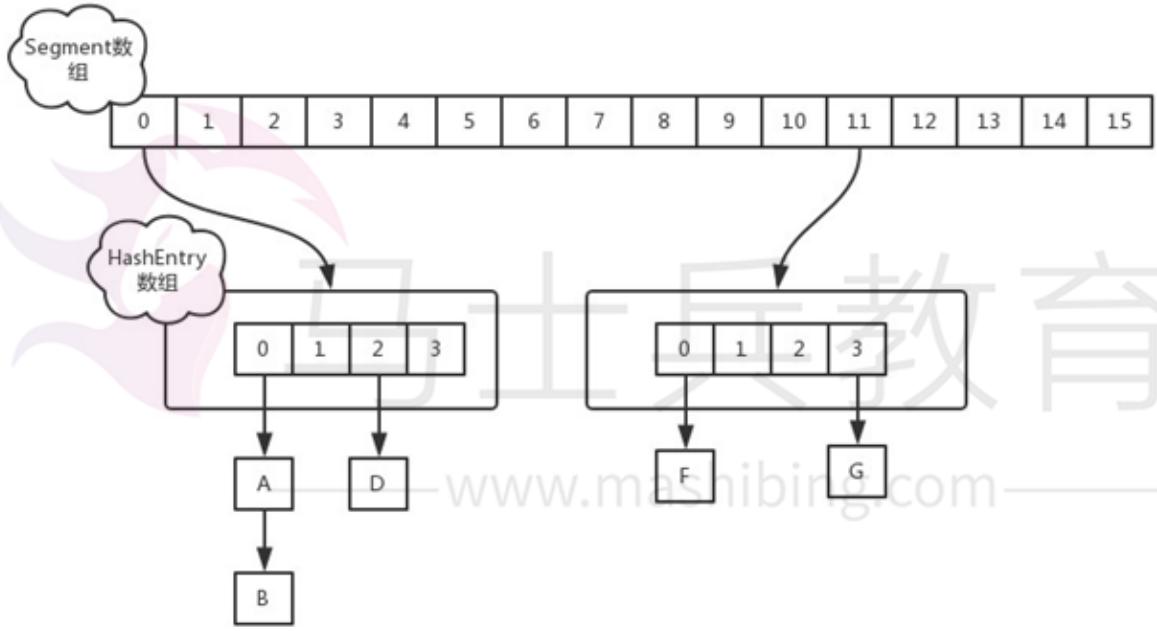
JDK1.7

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

在JDK1.7中，ConcurrentHashMap采用Segment + HashEntry的方式进行实

现，结构如下：

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个 HashEntry数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment的锁。

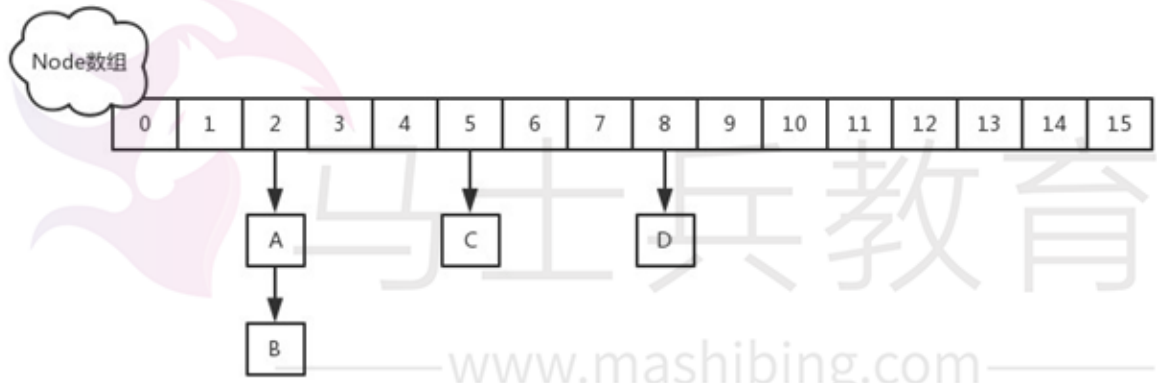


1. 该类包含两个静态内部类 HashE(img)ntry 和 Segment ； 前者用来封装映射表的键值对，后者用来充当锁的角色；
2. Segment 是一种可重入的锁 ReentrantLock，每个 Segment 守护一个HashEntry 数组里得元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 锁。

JDK1.8

在JDK1.8中，放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全进行实现，synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N 倍。

结构如下：



看插入元素过程（建议去看看源码）：

如果相应位置的Node还没有初始化，则调用CAS插入相应的数据；

```
1   else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
2   if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
3   break; // no lock when adding to empty bin
4   }
```

如果相应位置的Node不为空，且当前该节点不处于移动状态，则对该节点加

synchronized锁，如果该节点的hash不小于0，则遍历链表更新节点或插入新节点；

```
1 if (fh >= 0) {
```

```
2 binCount = 1;
3   for (Node<K,V> e = f;; ++binCount) {
4   K ek;
5   if (e.hash == hash &&
6   ((ek = e.key) == key ||
7   (ek != null && key.equals(ek)))) {
8   oldVal = e.val;
9   if (!onlyIfAbsent)
10  e.val = value;
11  break;
12  }
13  Node<K,V> pred = e;
14  if ((e = e.next) == null) {
15  pred.next = new Node<K,V>(hash, key, value, null);
16  break;
17  }
18  }
19  }
```

1. 如果该节点是TreeBin类型的节点，说明是红黑树结构，则通过 putTreeVal方法往红黑树中插入节点；如果binCount不为0，说明put操作对数据产生了影响，如果当前链表的个数达到8个，则通过treeifyBin方法转化为红黑树，如果oldVal不为空，说明是一次更新操作，没有对元素个数产生影响，则直接返回旧值；
2. 如果插入的是一个新节点，则执行addCount()方法尝试更新元素个数 baseCount；

辅助工具类

Array 和 ArrayList 有何区别?

- Array 可以存储基本数据类型和对象，ArrayList 只能存储对象。
- Array 是指定固定大小的，而 ArrayList 大小是自动扩展的。
- Array 内置方法没有 ArrayList 多，比如 addAll、removeAll、iteration 等方法只有 ArrayList 有。

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

如何实现 Array 和 List 之间的转换?

- Array 转 List： Arrays. asList(array)；
- List 转 Array：List 的 toArray() 方法。

comparable 和 comparator的区别?

- comparable接口实际上是出自java.lang包，它有一个 compareTo(Object obj)方法用来排序
- comparator接口实际上是出自 java.util 包，它有一个compare(Object obj1, Object obj2)方法用来排序

一般我们需要对一个集合使用自定义排序时，我们就要重写compareTo方法或 compare方法，当我们需要对某一个集合实现两种排序方式，比如一个song对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写compareTo方法和使用自制的Comparator方法或者以两个Comparator来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的Collections.sort().

Collection 和 Collections 有什么区别?

- java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了 大化的统一操作方式，其直接继承接口有List与Set。
- Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

TreeMap 和 TreeSet 在排序时如何比较元素？ Collections 工具类中的 sort()方法如何比较元素?

TreeSet 要求存放的对象所属的类必须实现 Comparable 接口，该接口提供了比较元素的 compareTo()方法，当插入元素时会回调该方法比较元素的大小。

TreeMap 要求存放的键值对映射的键必须实现 Comparable 接口从而根据键对元素进 行排 序。

Collections 工具类的 sort 方法有两种重载的形式，

第一种要求传入的待排序容器中存放的对象比较实现 Comparable 接口以实现元素的比较；

第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是Comparator 接口的子类型（需要重写 compare 方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java 中对函数式编程的支持）。

Vector,ArrayList, LinkedList的区别是什么?

答:

1. Vector、ArrayList都是以类似数组的形式存储在内存中，LinkedList则以链表的形式进行存储。
2. List中的元素有序、允许有重复的元素，Set中的元素无序、不允许有重复元素。
3. Vector线程同步，ArrayList、LinkedList线程不同步。
4. LinkedList适合指定位置插入、删除操作，不适合查找；ArrayList、Vector适合查找，不适合指定位置的插入、删除操作。
5. ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间。

HashTable, HashMap，TreeMap区别?

答:

1. HashTable线程同步，HashMap非线程同步。
2. HashTable不允许<键,值>有空值，HashMap允许<键,值>有空值。
3. HashTable使用Enumeration，HashMap使用Iterator。
4. HashTable中hash数组的默认大小是11，增加方式的old*2+1，HashMap中hash数组的默认大小是16，增长方式一定是2的指数倍。
5. TreeMap能够把它保存的记录根据键排序，默认是按升序排序。

HashMap的数据结构

jdk1.8之前list + 链表
jdk1.8之后list + 链表（当链表长度到8时，转化为红黑树）

HashMap的扩容因子

默认0.75，也就是会浪费1/4的空间，达到扩容因子时，会将list扩容一倍，0.75 是时间与空间一个平衡值；

多线程修改HashMap

多线程同时写入，同时执行扩容操作，多线程扩容可能死锁、丢数据；可以对HashMap 加入同步锁 Collections.synchronizedMap(hashMap)，但是效率很低，因为该锁是互斥锁，同一时刻只能有一个线程执行读写操作，这时候应该使用ConcurrentHashMap

注意：在使用Iterator遍历的时候，LinkedHashMap会产生`java.util.ConcurrentModificationException`。

> 扩展HashMap增加双向链表的实现，号称是最占内存的数据结构。支持iterator()时按Entry的插入顺序来排序(但是更新不算， 如果设置accessOrder属性为true，则所有读写访问都算)。实现上是在Entry上再增加属性before/after指针，插入时把自己加到Header Entry的前面去。如果所有读写访问都要排序，还要把前后Entry的before/after拼接起来以在链表中删除掉自己。

说说你知道的几个Java集合类：list、set、queue、map实现类

Java中的队列都有哪些，有什么区别

1. ArrayDeque, （数组双端队列）
2. PriorityQueue, （优先级队列）
3. ConcurrentLinkedQueue, （基于链表的并发队列）
4. DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）
5. ArrayBlockingQueue, （基于数组的并发阻塞队列）
6. LinkedBlockingQueue, （基于链表的FIFO阻塞队列）
7. LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）
8. PriorityBlockingQueue, （带优先级的无界阻塞队列）
9. SynchronousQueue （并发同步阻塞队列）

反射中，Class.forName和classloader的区别

java中class.forName()和classLoader都可用来对类进行加载。
class.forName()前者除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块。
而classLoader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容,只有在newInstance才会去执行static块。
Class.forName(name, initialize, loader)带参函数也可控制是否加载static块。并且只有调用了newInstance()方法采用调用构造函数，创建类的对象

看下Class.forName()源码

```
    //Class.forName(String className)  这是1.8的源码
    public static Class<?> forName(String className) throws
ClassNotFoundException {
        Class<?> caller = Reflection.getCallerClass();
        return forName0(className, true, ClassLoader.getClassLoader(caller),
caller);
    }
    //注意第二个参数，是指Class被loading后是不是必须被初始化。 不初始化就是不执行static的代码即静态代码
```

然后就是，测试代码证明上面的结论是OK的，如下：

```
package com.lxk.Reflect;

/**
 * Created by lxk on 2017/2/21
 */
public class Line {
    static {
        System.out.println("静态代码块执行：loading line");
    }
}
```

```
package com.lxk.Reflect;

/**
 * Created by lxk on 2017/2/21
 */
public class Point {
    static {
        System.out.println("静态代码块执行：loading point");
    }
}
```

```
package com.lxk.Reflect;

/**
 * Class.forName和classloader的区别
 * <p>
 * Created by lxk on 2017/2/21
 */
public class ClassloaderAndForNameTest {
    public static void main(String[] args) {
        String wholeNameLine = "com.lxk.Reflect.Line";
        String wholeNamePoint = "com.lxk.Reflect.Point";
        System.out.println("下面是测试Classloader的效果");
        testClassloader(wholeNameLine, wholeNamePoint);
        System.out.println("----------------------------------");
        System.out.println("下面是测试Class.forName的效果");
        testForName(wholeNameLine, wholeNamePoint);

    }

    /**
     * classloader
     */

    private static void testClassloader(String wholeNameLine, String
wholeNamePoint) {
        Class<?> line;
```

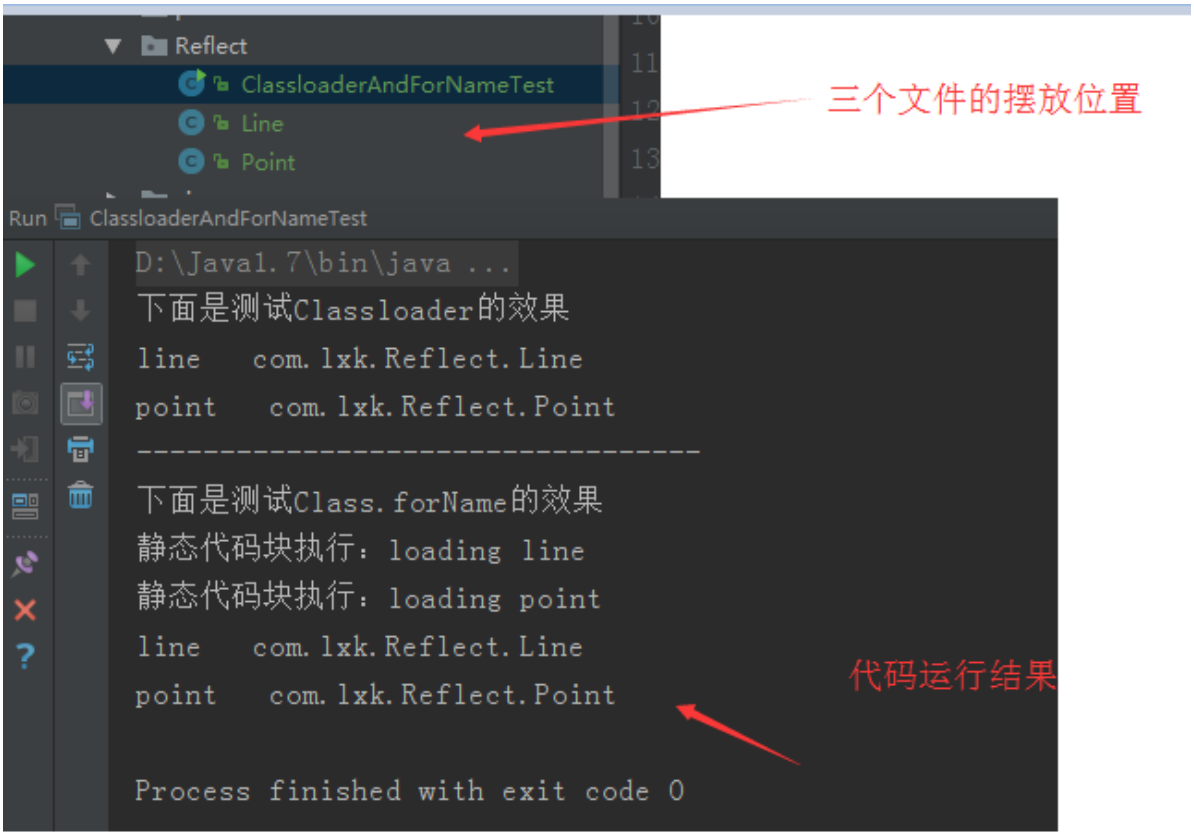
```
        Class<?> point;
        ClassLoader loader = ClassLoader.getSystemClassLoader();
        try {
            line = loader.loadClass(wholeNameLine);
            point = loader.loadClass(wholeNamePoint);
            //demo =
ClassloaderAndForNameTest.class.getClassLoader().loadClass(wholeNamePoint);//这个
也是可以的
            System.out.println("line   " + line.getName());
            System.out.println("point   " + point.getName());
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * Class.forName
     */
    private static void testForName(String wholeNameLine, String wholeNamePoint)
{

        try {
            Class line = Class.forName(wholeNameLine);
            Class point = Class.forName(wholeNamePoint);
            System.out.println("line   " + line.getName());
            System.out.println("point   " + point.getName());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }


}
```

执行结果如下：



备注：

根据运行结果，可以看到，classloader并没有执行静态代码块，如开头的理论所说。

而下面的Class.forName则是夹在完之后，就里面执行了静态代码块，可以看到，2个类，line和point的静态代码块执行结果是一起的，然后才是各自的打印结果。

也说明上面理论是OK的。

更新于2017/06/20

因为看到有小伙伴有疑问，我就把自己以前的代码拿出来再次测试一遍，发现结果仍然是相同的。

但是，因为我的Javabean model又经历了其他的测试，所以，两个model内部的代码稍有变化，

然后，还真就测试出来了不一样的地方。

这估计是其他理论所没有的。具体看下面的代码吧。

只是修改了Line的代码，添加了几个静态的方法和变量。

```
package com.lxk.reflect;

/**
 * Created by lxk on 2017/2/21
 */
public class Line {
    static {
        System.out.println("静态代码块执行：loading line");
    }

    public static String s = getString();

    private static String getString() {
        System.out.println("给静态变量赋值的静态方法执行：loading line");
```

```
        return "ss";
    }

    public static void test() {
        System.out.println("普通静态方法执行：loading line");
    }

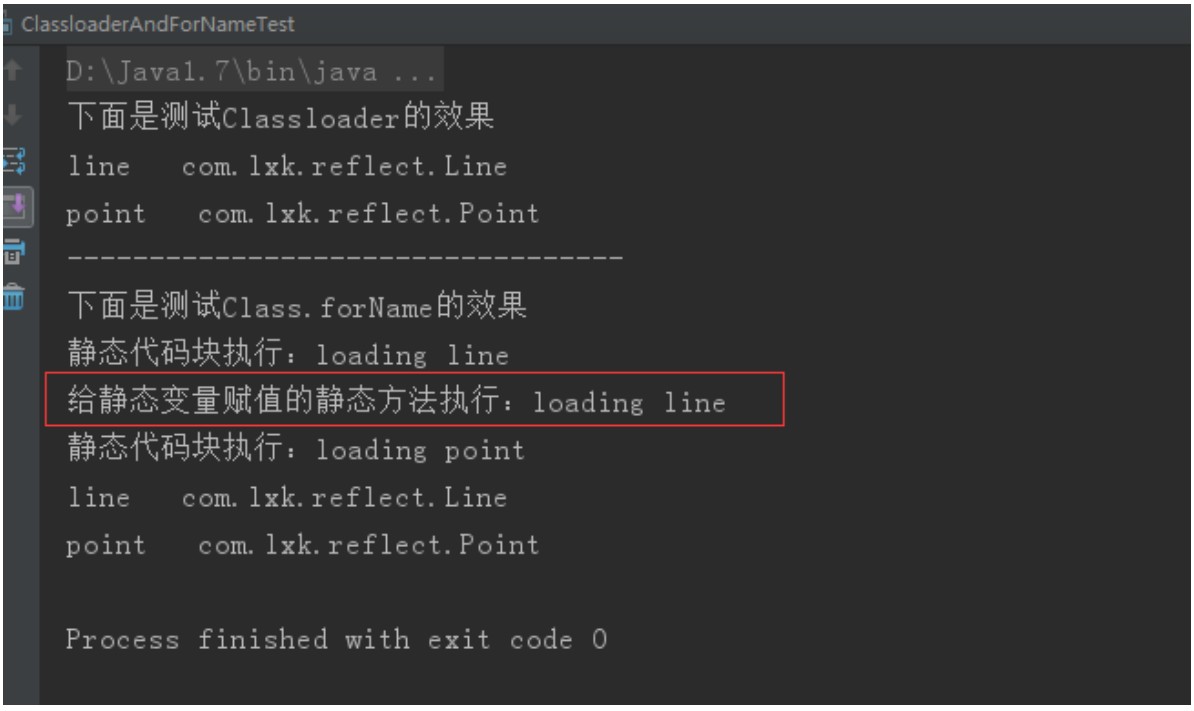
    {
        System.out.println("要是普通的代码块呢？");
    }

    public Line() {
        System.out.println("构造方法执行");
    }
}
```

可以看到，除了原来的简单的一个静态代码块以外，我又添加了构造方法，静态方法，以及静态变量，且，静态变量被一个静态方法赋值。

然后，看执行结果。

稍有不同。



除了，静态代码块的执行外，竟然还有一个静态方法被执行，就是给静态变量赋值的静态方法被执行了。

这个估计是以前没人发现的吧。

所以

上面的结论，就可以进一步的修改啦。

也许，这个执行的也叫，static块呢。

讲讲IO里面的常见类，字节流、字符流、接口、实现类、方法阻塞

Java IO流详解（二）——IO流的框架体系

一、IO流的概念

Java的IO流是实现输入/输出的基础，它可以方便地实现数据的输入/输出操作，在Java中把不同的输入/输出源抽象表述为"流"。流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流，流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。

流有输入和输出，输入时是流从数据源流向程序。输出时是流从程序传向数据源，而数据源可以是内存，文件，网络或程序等。

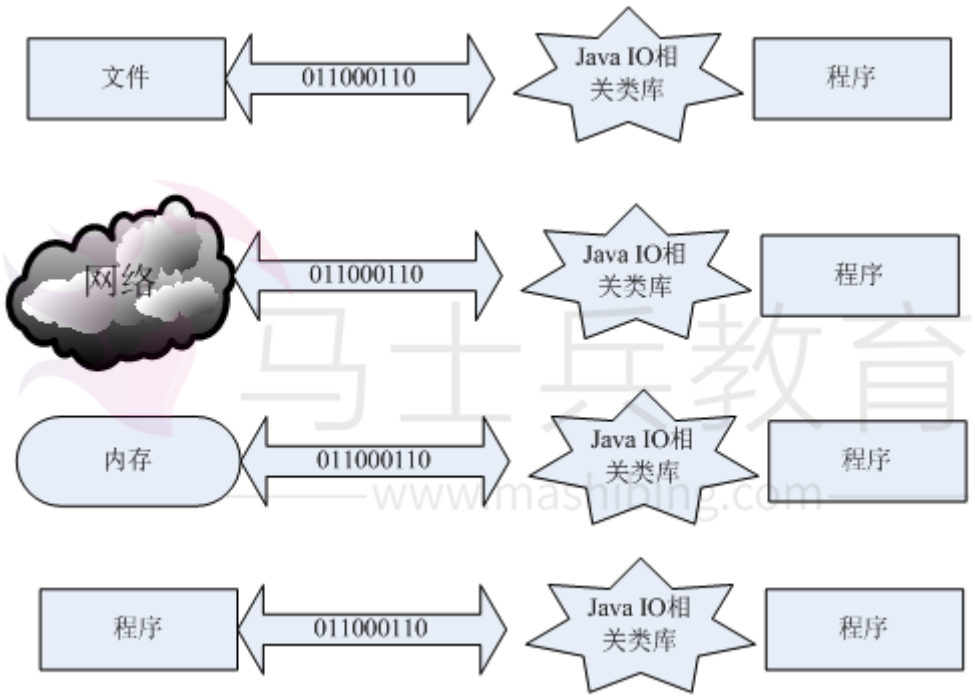
二、IO流的分类

1.输入流和输出流

根据数据流向不同分为：输入流和输出流。

> 输入流:只能从中读取数据，而不能向其写入数据。
> 输出流：只能向其写入数据，而不能从中读取数据。

如下如所示：对程序而言，向右的箭头，表示输入，向左的箭头，表示输出。



2.字节流和字符流

字节流和字符流和用法几乎完全一样，区别在于字节流和字符流所操作的数据单元不同。

字符流的由来： 因为数据编码的不同，而有了对字符进行高效操作的流对象。本质其实就是基于字节流读取时，去查了指定的码表。字节流和字符流的区别：

（1）读写单位不同：字节流以字节（8bit）为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节。

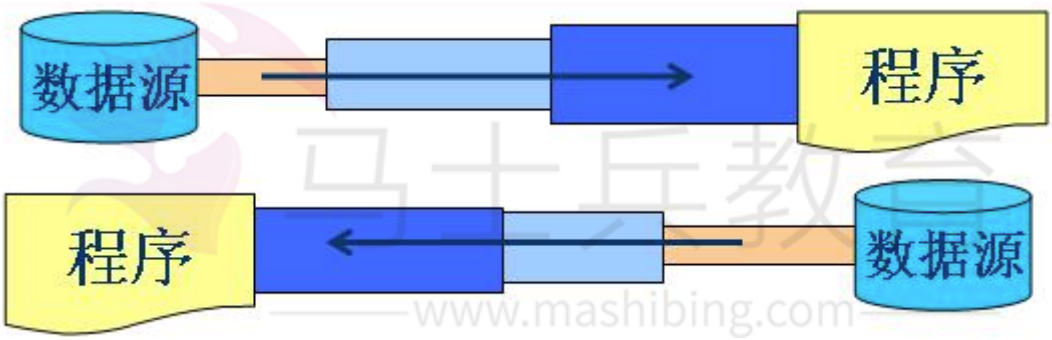
（2）处理对象不同：字节流能处理所有类型的数据（如图片、avi等），而字符流只能处理字符类型的数据。

只要是处理纯文本数据，就优先考虑使用字符流。 除此之外都使用字节流。

3.节点流和处理流

按照流的角色来分，可以分为节点流和处理流。
可以从/向一个特定的IO设备（如磁盘、网络）读/写数据的流，称为节点流，节点流也被成为低级流。
处理流是对一个已存在的流进行连接或封装，通过封装后的流来实现数据读/写功能，处理流也被称为高级流。

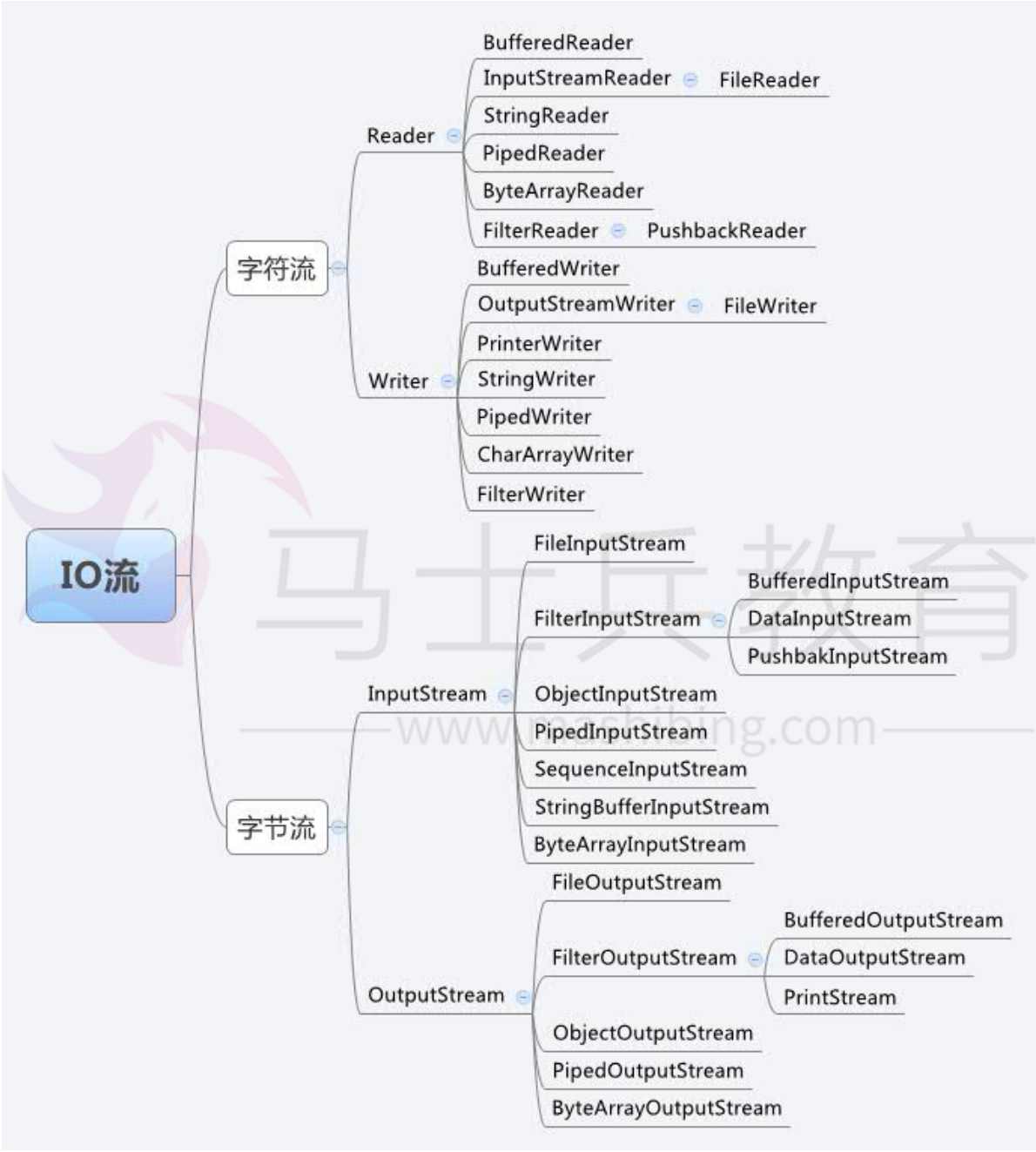
```
//节点流，直接传入的参数是IO设备
FileInputStream fis = new FileInputStream("test.txt");
//处理流，直接传入的参数是流对象
BufferedInputStream bis = new BufferedInputStream(fis);
```



当使用处理流进行输入/输出时，程序并不会直接连接到实际的数据源，没有和实际的输入/输出节点连接。使用处理流的一个明显好处是，只要使用相同的处理流，程序就可以采用完全相同的输入/输出代码来访问不同的数据源，随着处理流所包装节点流的变化，程序实际所访问的数据源也相应地发生变化。
实际上，Java使用处理流来包装节点流是一种典型的装饰器设计模式，通过使用处理流来包装不同的节点流，既可以消除不同节点流的实现差异，也可以提供更方便的方法来完成输入/输出功能。

三、IO流的四大基类

根据流的流向以及操作的数据单元不同，将流分为了四种类型，每种类型对应一种抽象基类。这四种抽象基类分别为：InputStream,Reader,OutputStream以及Writer。四种基类下，对应不同的实现类，具有不同的特性。在这些实现类中，又可以分为节点流和处理流。下面就是整个由着四大基类支撑下，整个IO流的框架图。



InputStream,Reader,OutputStream以及Writer，这四大抽象基类，本身并不能创建实例来执行输入/输出，但它们将成为所有输入/输出流的模版，所以它们的方法是所有输入/输出流都可以使用的方法。类似于集合中的Collection接口。

1.InputStream

InputStream 是所有的输入字节流的父类，它是一个抽象类，主要包含三个方法：

```
//读取一个字节并以整数的形式返回(0~255),如果返回-1已到输入流的末尾。
int read() ;
//读取一系列字节并存储到一个数组buffer，返回实际读取的字节数，如果读取前已到输入流的末尾返回-1。
int read(byte[] buffer) ;
//读取length个字节并存储到一个字节数组buffer，从off位置开始存,最多len， 返回实际读取的字节数，如果读取前以到输入流的末尾返回-1。
int read(byte[] buffer, int off, int len) ;
```

2.Reader

Reader 是所有的输入字符流的父类，它是一个抽象类，主要包含三个方法：

```
//读取一个字符并以整数的形式返回(0~255),如果返回-1已到输入流的末尾。
int read() ;
//读取一系列字符并存储到一个数组buffer，返回实际读取的字符数，如果读取前已到输入流的末尾返回-1。
int read(char[] cbuf) ;
//读取length个字符,并存储到一个数组buffer，从off位置开始存,最多读取len，返回实际读取的字符数，如果读取前以到输入流的末尾返回-1。
int read(char[] cbuf, int off, int len)
```

对比InputStream和Reader所提供的方法，就不难发现两个基类的功能基本一样的，只不过读取的数据单元不同。

在执行完流操作后，要调用`close()`方法来关系输入流，因为程序里打开的IO资源不属于内存资源，垃圾回收机制无法回收该资源，所以应该显式关闭文件IO资源。

除此之外，InputStream和Reader还支持如下方法来移动流中的指针位置：

```
//在此输入流中标记当前的位置
//readlimit - 在标记位置失效前可以读取字节的最大限制。
void mark(int readlimit)
// 测试此输入流是否支持 mark 方法
boolean markSupported()
// 跳过和丢弃此输入流中数据的 n 个字节/字符
long skip(long n)
//将此流重新定位到最后一次对此输入流调用 mark 方法时的位置
void reset()
```

3.OutputStream

OutputStream 是所有的输出字节流的父类，它是一个抽象类，主要包含如下四个方法：

```
//向输出流中写入一个字节数据,该字节数据为参数b的低8位。
void write(int b) ;
//将一个字节类型的数组中的数据写入输出流。
void write(byte[] b);
//将一个字节类型的数组中的从指定位置（off）开始的,len个字节写入到输出流。
void write(byte[] b, int off, int len);
//将输出流中缓冲的数据全部写出到目的地。
void flush();
```

4.Writer

Writer 是所有的输出字符流的父类，它是一个抽象类,主要包含如下六个方法：

```
//向输出流中写入一个字符数据,该字节数据为参数b的低16位。
void write(int c);
//将一个字符类型的数组中的数据写入输出流，
void write(char[] cbuf)
//将一个字符类型的数组中的从指定位置（offset）开始的,length个字符写入到输出流。
void write(char[] cbuf, int offset, int length);
//将一个字符串中的字符写入到输出流。
void write(String string);
//将一个字符串从offset开始的length个字符写入到输出流。
void write(String string, int offset, int length);
//将输出流中缓冲的数据全部写出到目的地。
void flush()
```

可以看出，Writer比OutputStream多出两个方法，主要是支持写入字符和字符串类型的数据。

使用Java的IO流执行输出时，不要忘记关闭输出流，关闭输出流除了可以保证流的物理资源被回收之外，还能将输出流缓冲区的数据flush到物理节点里（因为在执行close()方法之前，自动执行输出流的flush()方法）

以上内容就是整个IO流的框架介绍。

讲讲NIO

NIO技术概览

NIO（Non-blocking I/O，在Java领域，也称为New I/O），是一种同步非阻塞的I/O模型，也是I/O多路复用的基础，已经被越来越多地应用到大型应用服务器，成为解决高并发与大量连接、I/O处理问题的有效方式。

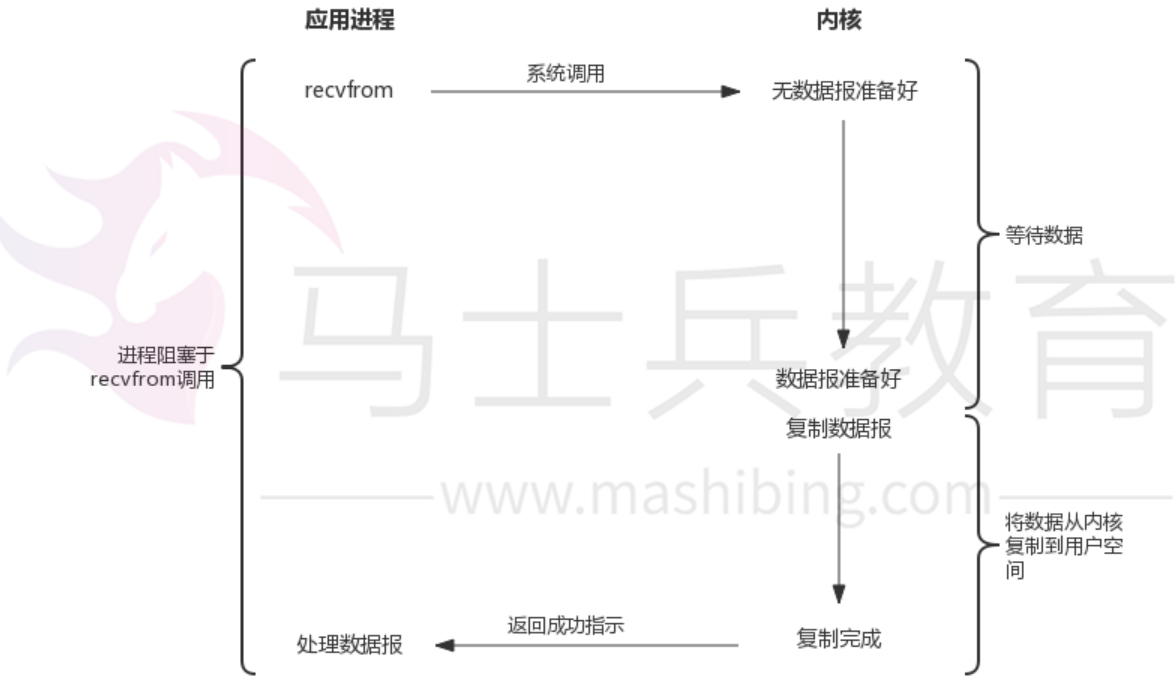
IO模型的分类

按照《Unix网络编程》的划分，I/O模型可以分为：阻塞I/O模型、非阻塞I/O模型、I/O复用模型、信号驱动式I/O模型和异步I/O模型，按照POSIX标准来划分只分为两类：同步I/O和异步I/O。

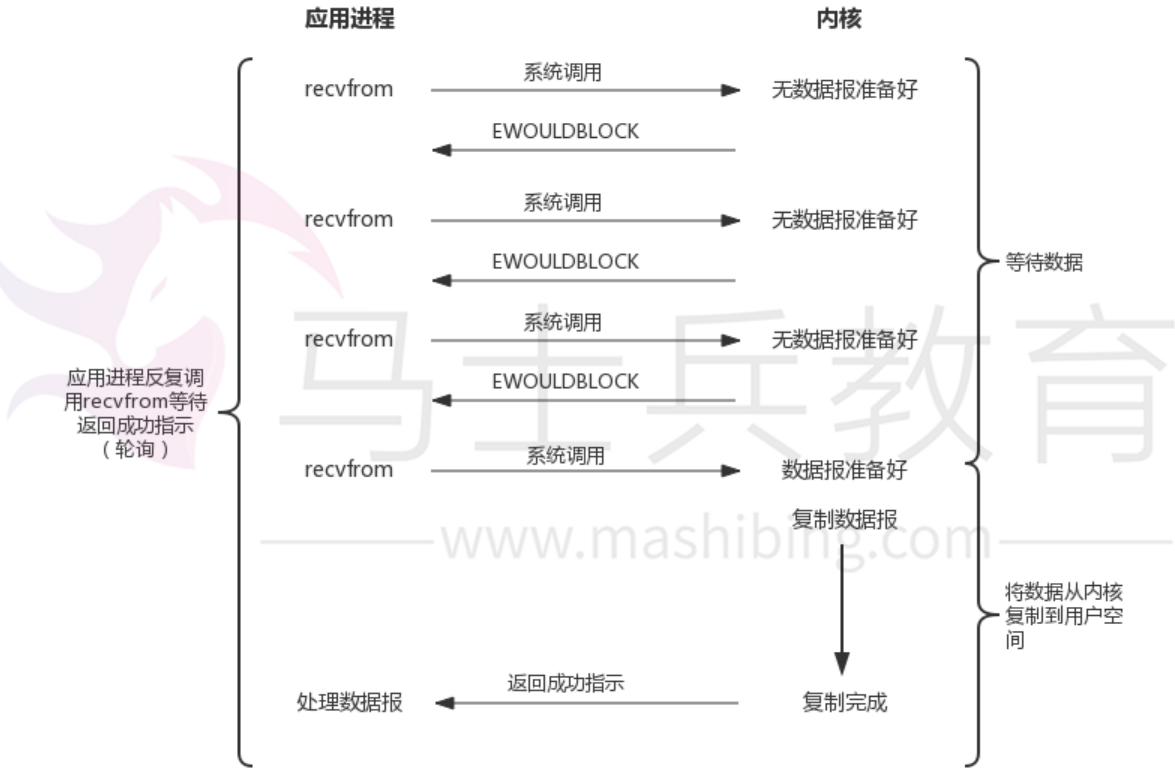
如何区分呢？首先一个I/O操作其实分成了两个步骤：**发起IO请求和实际的IO操作。**同步I/O和异步I/O的区别就在于第二个步骤是否阻塞，如果实际的I/O读写阻塞请求进程，那么就是同步I/O，因此阻塞I/O、非阻塞I/O、I/O复用、信号驱动I/O都是同步I/O，如果不阻塞，而是操作系统帮你做完I/O操作再将结果返回给你，那么就是异步I/O。

阻塞I/O和非阻塞I/O的区别在于第一步，发起I/O请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞I/O，如果不阻塞，那么就是非阻塞I/O。

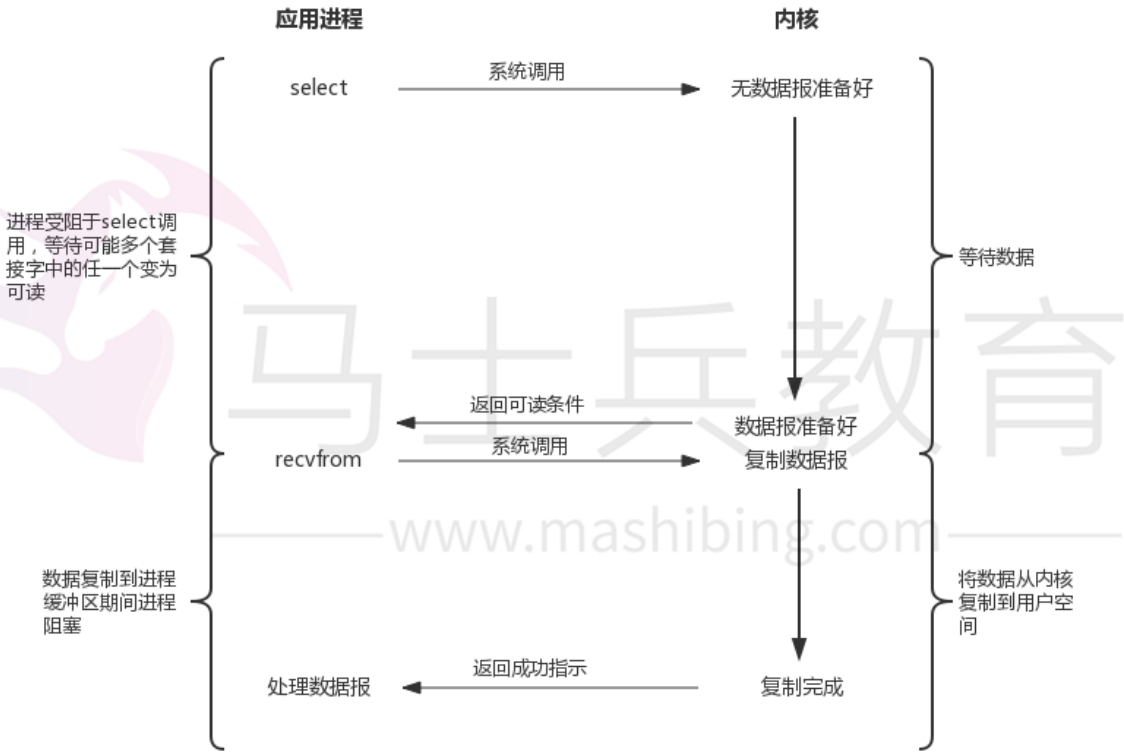
- 阻塞I/O模型：在linux中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：



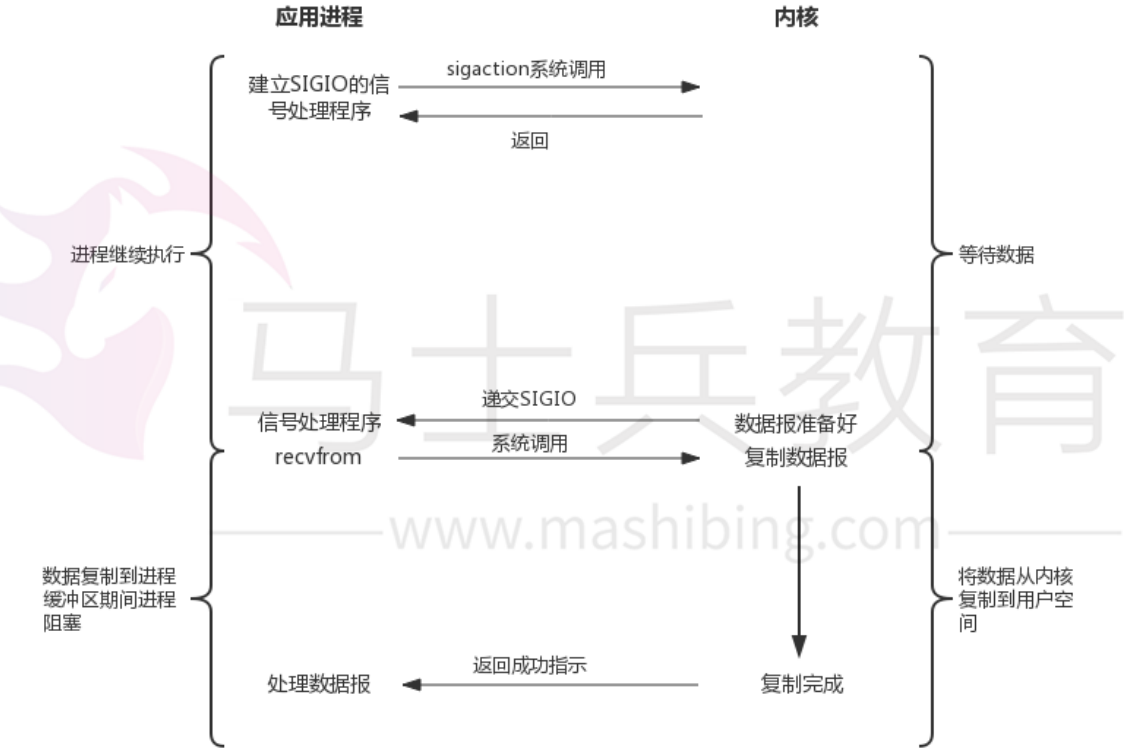
- 非阻塞I/O模型：linux下，可以通过设置socket使其变为non-blocking。当对一个non-blocking socket执行读操作时，流程是这个样子：



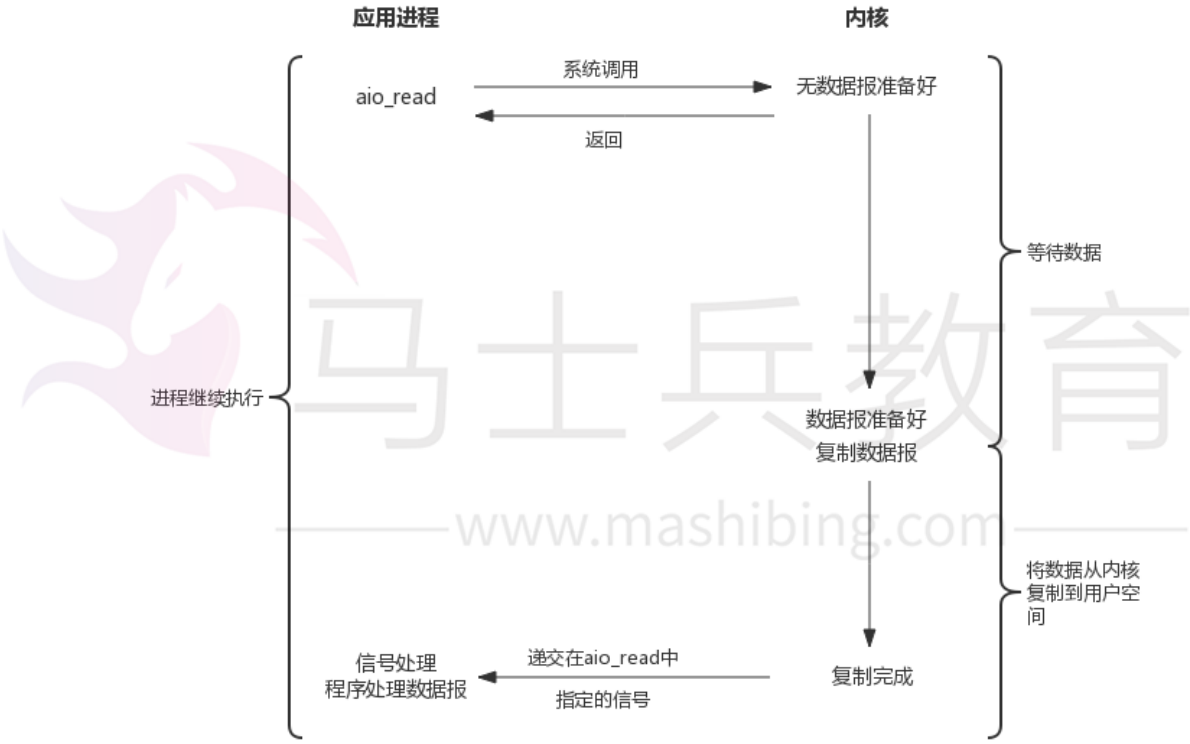
- I/O复用模型：我们可以调用`select`或`poll`，阻塞在这两个系统调用中的某一个之上，而不是真正的IO系统调用上：



- 信号驱动式I/O模型：我们可以用信号，让内核在描述符就绪时发送SIGIO信号通知我们：



- 异步I/O模型：用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个asynchronousread之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，内核会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，内核会给用户进程发送一个signal，告诉它read操作完成了：



以上参考自：《UNIX网络编程》

从前面 I/O 模型的分类中，我们可以看出 AIO 的动机。阻塞模型需要在 I/O 操作开始时阻塞应用程序。这意味着不可能同时重叠进行处理和 I/O 操作。非阻塞模型允许处理和 I/O 操作重叠进行，但是这需要应用程序来检查 I/O 操作的状态。对于异步I/O ，它允许处理和 I/O 操作重叠进行，包括 I/O 操作完成的通知。除了需要阻塞之外，select 函数所提供的功能（异步阻塞 I/O）与 AIO 类似。不过，它是对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。

参考下知乎上的回答：

> - **同步与异步**：同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)。所谓同步，就是在发出一个**调用**时，在没有得到结果之前，该**调用**就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由**调用者**主动等待这个**调用**的结果；
> - **阻塞与非阻塞**：阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回；而非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

两种IO多路复用方案：Reactor和Proactor

一般地，I/O多路复用机制都依赖于一个**事件多路分离器(Event Demultiplexer)**。分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write**事件处理器(Event Handler)**。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。

两个与事件分离器有关的模式是Reactor和Proactor。Reactor模式采用同步I/O，而Proactor采用异步I/O。在Reactor中，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

而在Proactor模式中，处理器或者兼任处理器的事件分离器，只负责发起异步读写操作。I/O操作本身由操作系统来完成。传递给操作系统的参数需要包括用户定义的数据缓冲区地址和数据大小，操作系统才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获I/O操作完成事件，然后将事件传递给对应处理器。比如，在windows上，处理器发起一个异步I/O操作，再由事件分离器等待

IOCompletion事件。典型的异步模式实现，都建立在操作系统支持异步API的基础之上，我们将这种实现称为“系统级”异步或“真”异步，因为应用程序完全依赖操作系统执行真正的I/O工作。

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（写操作类似）。

在Reactor中实现读：

- 注册读就绪事件和相应的事件处理器；
- 事件分离器等待事件；
- 事件到来，激活分离器，分离器调用事件对应的处理器；
- 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

在Proactor中实现读：

- 处理器发起异步读操作（注意：操作系统必须支持异步I/O）。在这种情况下，处理器无视I/O就绪事件，它关注的是完成事件；
- 事件分离器等待操作完成事件；
- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成；
- 事件分离器呼唤处理器；
- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

可以看出，两个模式的相同点，都是对某个I/O事件的事件通知（即告诉某个模块，这个I/O操作可以进行或已经完成）。在结构上，两者的相同点和不同点如下：

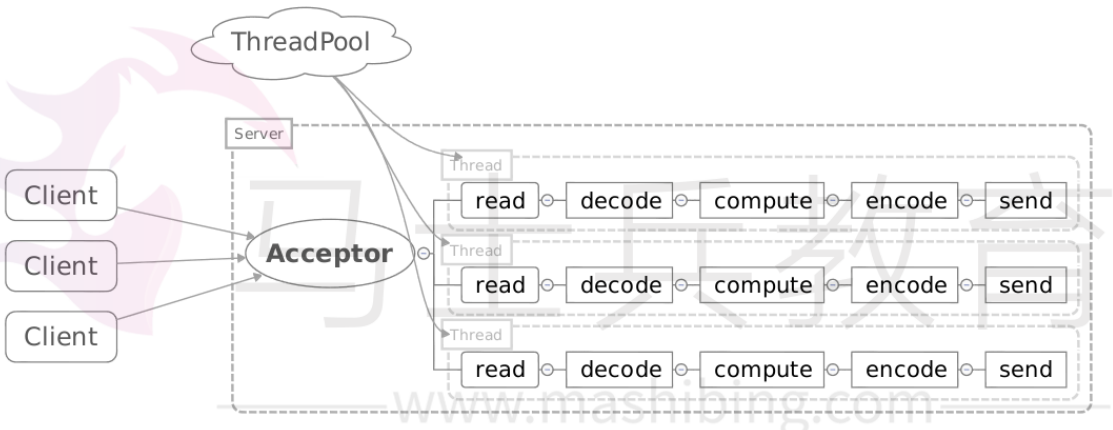
- 相同点：demultiplexor负责提交I/O操作（异步）、查询设备是否可操作（同步），然后当条件满足时，就回调handler；
- 不同点：异步情况下（Proactor），当回调handler时，表示I/O操作已经完成；同步情况下（Reactor），回调handler时，表示I/O设备可以进行某个操作（can read or can write）。

传统BIO模型

BIO是同步阻塞式IO，通常在while循环中服务端会调用accept方法等待接收客户端的连接请求，一旦接收到一个连接请求，就可以建立通信套接字在这个通信套接字上进行读写操作，此时不能再接收其他客户端连接请求，只能等待同当前连接的客户端的操作执行完成。

如果BIO要能够同时处理多个客户端请求，就必须使用多线程，即每次accept阻塞等待来自客户端请求，一旦受到连接请求就建立通信套接字同时开启一个新的线程来处理这个套接字的数据读写请求，然后立刻又继续accept等待其他客户端连接请求，即为每一个客户端连接请求都创建一个线程来单独处理。

我们看下传统的BIO方式下的编程模型大致如下：




```
public class BIODemo {
```

```
    public static void main(String[] args) throws IOException {

        ExecutorService executor = Executors.newFixedThreadPool(128);

        ServerSocket serverSocket = new ServerSocket();
        serverSocket.bind(new InetSocketAddress(1234));
        // 循环等待新连接
        while (true) {
            Socket socket = serverSocket.accept();
            // 为新的连接创建线程执行任务
            executor.submit(new ConnectionTask(socket));
        }
    }

}

class ConnectionTask extends Thread {
    private Socket socket;

    public ConnectionTask(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        while (true) {
            InputStream inputStream = null;
            OutputStream outputStream = null;
            try {
                inputStream = socket.getInputStream();

                // read from socket...

                inputStream.read();

                outputStream = socket.getOutputStream();

                // write to socket...

                outputStream.write();

            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                // 关闭资源...

            }

        }
    }
}
```

这里之所以使用多线程，是因为socket.accept()、inputStream.read()、outputStream.write()都是同步阻塞的，当一个连接在处理I/O的时候，系统是阻塞的，如果是单线程的话在阻塞的期间不能接受任何请求。所以，使用多线程，就可以让CPU去处理更多的事情。其实这也是所有使用多线程的本质：

- 利用多核。

- 当I/O阻塞系统，但CPU空闲的时候，可以利用多线程使用CPU资源。

使用线程池能够让线程的创建和回收成本相对较低。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的I/O并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池可以缓冲一些过多的连接或请求。

但这个模型最本质的问题在于，严重依赖于线程。但线程是很"贵"的资源，主要表现在：

1. 线程的创建和销毁成本很高，在Linux这样的操作系统中，线程本质上就是一个进程。创建和销毁都是重量级的系统函数；
2. 线程本身占用较大内存，像Java的线程栈，一般至少分配512K～1M的空间，如果系统中的线程数过千，恐怕整个JVM的内存都会被吃掉一半；
3. 线程的切换成本是很高的。操作系统发生线程切换的时候，需要保留线程的上下文，然后执行系统调用。如果线程数过高，可能执行线程切换的时间甚至会大于线程执行的时间，这时候带来的表现往往是系统load偏高、CPU sy使用率特别高（超过20%以上），导致系统几乎陷入不可用的状态；
4. 容易造成锯齿状的系统负载。因为系统负载是用活动线程数或CPU核心数，一旦线程数量高但外部网络环境不是很稳定，就很容易造成大量请求的结果同时返回，激活大量阻塞线程从而使系统负载压力过大。

所以，当面对十万甚至百万级连接的时候，传统的BIO模型是无能为力的。随着移动端应用的兴起和各种网络游戏的盛行，百万级长连接日趋普遍，此时，必然需要一种更高效的I/O处理模型。

NIO的实现原理

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题，即在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知应用程序进行处理，应用再将流读取到缓冲区或写入操作系统。

也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

下面看下代码的实现：

NIO服务端代码（新建连接）：

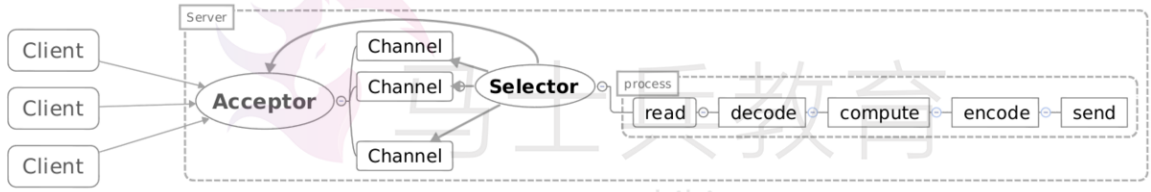
```
//获取一个ServerSocket通道
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
serverChannel.socket().bind(new InetSocketAddress(port));
//获取通道管理器
selector = Selector.open();
//将通道管理器与通道绑定，并为该通道注册SelectionKey.OP_ACCEPT事件，
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

NIO服务端代码（监听）：

```
while(true){
    //当有注册的事件到达时，方法返回，否则阻塞。
   selector.select();
   for(SelectionKey key : selector.selectedKeys()){
       if(key.isAcceptable()){
```

```
            ServerSocketChannel server =
                 (ServerSocketChannel)key.channel();
            SocketChannel channel = server.accept();
            channel.write(ByteBuffer.wrap(
             new String("send message to client").getBytes()));
            //在与客户端连接成功后，为客户端通道注册SelectionKey.OP_READ事件。
            channel.register(selector, SelectionKey.OP_READ);
        }else if(key.isReadable()){//有可读数据事件
            SocketChannel channel = (SocketChannel)key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(10);
            int read = channel.read(buffer);
            byte[] data = buffer.array();
            String message = new String(data);
            System.out.println("receive message from client, size:"
                + buffer.position() + " msg: " + message);
        }
    }
}
```

NIO模型示例如下：



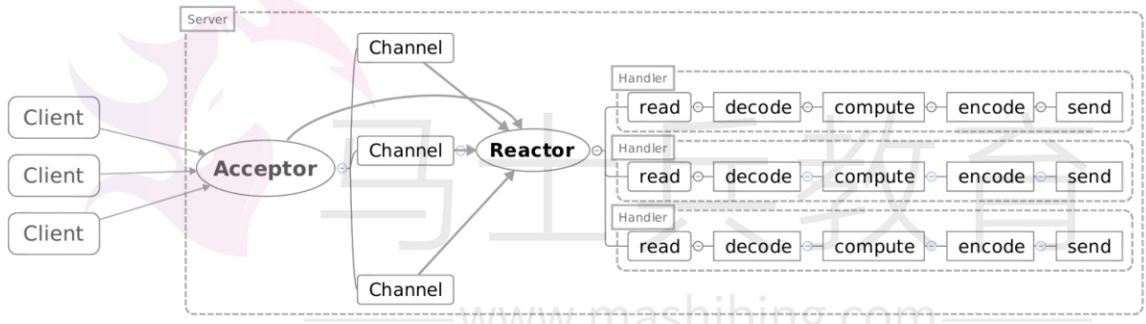
- Acceptor注册Selector，监听accept事件；
- 当客户端连接后，触发accept事件；
- 服务器构建对应的Channel，并在其上注册Selector，监听读写事件；
- 当发生读写事件后，进行相应的读写处理。

Reactor模型

有关Reactor模型结构，可以参考Doug Lea在 Scalable IO in Java中的介绍。这里简单介绍一下Reactor模式的典型实现：

Reactor单线程模型

这是最简单的单Reactor单线程模型。Reactor线程负责多路分离套接字、accept新连接，并分派请求到处理器链中。该模型适用于处理器链中业务处理组件能快速完成的场景。不过，这种单线程模型不能充分利用多核资源，所以实际使用的不多。



这个模型和上面的NIO流程很类似，只是将消息相关处理独立到了Handler中去了。

代码实现如下：

```
public class Reactor implements Runnable {

    final Selector selector;

    final ServerSocketChannel serverSocketChannel;

    public static void main(String[] args) throws IOException {
        new Thread(new Reactor(1234)).start();
    }

    public Reactor(int port) throws IOException {
        selector = Selector.open();
        serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new InetSocketAddress(port));
        serverSocketChannel.configureBlocking(false);
        SelectionKey key = serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
        key.attach(new Acceptor());
    }

    @Override
    public void run() {
        while (!Thread.interrupted()) {
            try {
                selector.select();
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                for (SelectionKey selectionKey : selectionKeys) {
                    dispatch(selectionKey);
                }
                selectionKeys.clear();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }

    private void dispatch(SelectionKey selectionKey) {
        Runnable run = (Runnable) selectionKey.attachment();
        if (run != null) {
            run.run();
        }
    }

    class Acceptor implements Runnable {

        @Override
        public void run() {
            try {
                SocketChannel channel = serverSocketChannel.accept();
                if (channel != null) {
                    new Handler(selector, channel);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        }

    }


}

class Handler implements Runnable {

    private final static int DEFAULT_SIZE = 1024;

    private final SocketChannel socketChannel;

    private final SelectionKey seletionKey;

    private static final int READING = 0;

    private static final int SENDING = 1;

    private int state = READING;

    ByteBuffer inputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    ByteBuffer outputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    public Handler(Selector selector, SocketChannel channel) throws IOException
{
        this.socketChannel = channel;
        socketChannel.configureBlocking(false);
        this.seletionKey = socketChannel.register(selector, 0);
        seletionKey.attach(this);
        seletionKey.interestOps(SelectionKey.OP_READ);
        selector.wakeup();
    }

    @Override
    public void run() {
        if (state == READING) {
            read();
        } else if (state == SENDING) {
            write();
        }

    }

    class Sender implements Runnable {

        @Override
        public void run() {
            try {
                socketChannel.write(outputBuffer);
            } catch (IOException e) {
                e.printStackTrace();
            }
            if (outIsComplete()) {
                seletionKey.cancel();
            }
        }
```

```
    }

    private void write() {
        try {
            socketChannel.write(outputBuffer);
        } catch (IOException e) {
            e.printStackTrace();
        }
        while (outIsComplete()) {
            seletionKey.cancel();
        }

    }

    private void read() {
        try {
            socketChannel.read(inputBuffer);
            if (inputIsComplete()) {
                process();
                System.out.println("接收到来自客户端（" +
socketChannel.socket().getInetAddress().getHostAddress()
                        + "）的消息：" + new String(inputBuffer.array()));
                seletionKey.attach(new Sender());
                seletionKey.interestOps(SelectionKey.OP_WRITE);
                seletionKey.selector().wakeup();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }


    }

    public boolean inputIsComplete() {
        return true;
    }

    public boolean outIsComplete() {
        return true;

    }

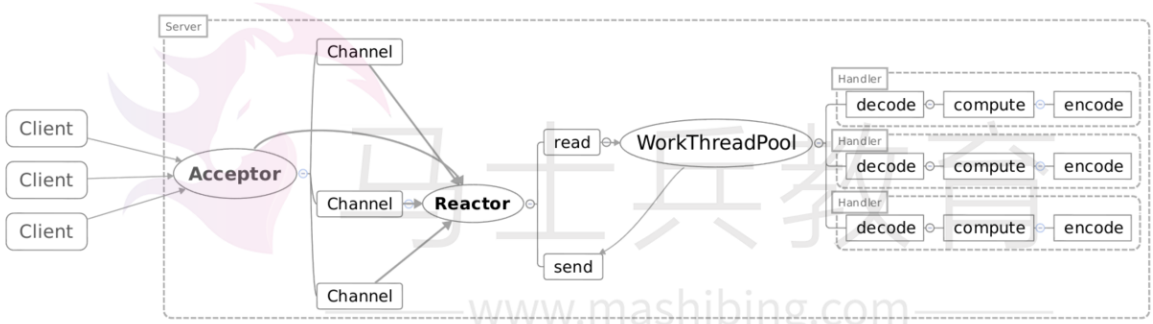
    public void process() {
        // do something...
    }

}
```

虽然上面说到NIO一个线程就可以支持所有的IO处理。但是瓶颈也是显而易见的。我们看一个客户端的情况，如果这个客户端多次进行请求，如果在Handler中的处理速度较慢，那么后续的客户端请求都会被积压，导致响应变慢！所以引入了Reactor多线程模型。

Reactor多线程模型

相比上一种模型，该模型在处理器链部分采用了多线程（线程池）：



Reactor多线程模型就是将Handler中的IO操作和非IO操作分开，操作IO的线程称为IO线程，非IO操作的线程称为工作线程。这样的话，客户端的请求会直接被丢到线程池中，客户端发送请求就不会堵塞。

可以将Handler做如下修改：

```
class Handler implements Runnable {

    private final static int DEFAULT_SIZE = 1024;

    private final SocketChannel socketChannel;

    private final SelectionKey seletionKey;

    private static final int READING = 0;

    private static final int SENDING = 1;

    private int state = READING;

    ByteBuffer inputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    ByteBuffer outputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    private Selector selector;

    private static ExecutorService executorService =
Executors.newFixedThreadPool(Runtime.getRuntime()
            .availableProcessors());
    private static final int PROCESSING = 3;

    public Handler(Selector selector, SocketChannel channel) throws IOException
{
        this.selector = selector;
        this.socketChannel = channel;
        socketChannel.configureBlocking(false);
        this.seletionKey = socketChannel.register(selector, 0);
        seletionKey.attach(this);
        seletionKey.interestOps(SelectionKey.OP_READ);
        selector.wakeup();
    }

    @Override
    public void run() {
        if (state == READING) {
```

```
            read();
        } else if (state == SENDING) {
            write();
        }

    }

    class Sender implements Runnable {

        @Override
        public void run() {
            try {
                socketChannel.write(outputBuffer);
            } catch (IOException e) {
                e.printStackTrace();
            }
            if (outIsComplete()) {
                seletionKey.cancel();
            }
        }

    }

    private void write() {
        try {
            socketChannel.write(outputBuffer);
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (outIsComplete()) {
            seletionKey.cancel();
        }

    }

    private void read() {
        try {
            socketChannel.read(inputBuffer);
            if (inputIsComplete()) {
                process();
                executorService.execute(new Processer());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }


    }

    public boolean inputIsComplete() {
        return true;
    }

    public boolean outIsComplete() {
        return true;

    }
```

```
    public void process() {

    }

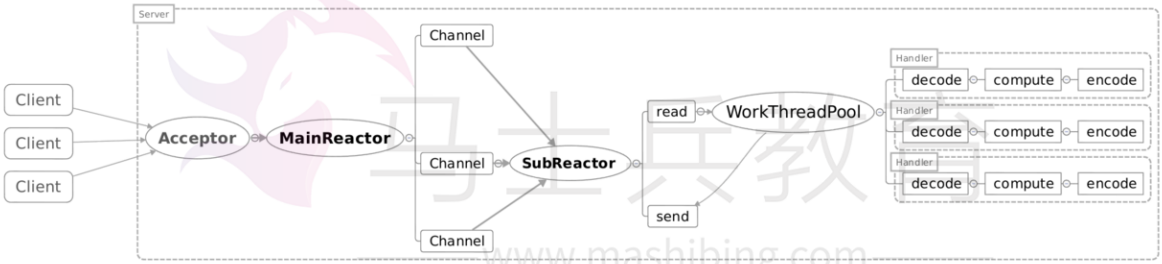
    synchronized void processAndHandOff() {
        process();
        state = SENDING; // or rebind attachment
        seletionKey.interestOps(SelectionKey.OP_WRITE);
        selector.wakeup();
    }
    class Processer implements Runnable {
        public void run() {
            processAndHandOff();
        }
    }

}
```

但是当用户进一步增加的时候，Reactor会出现瓶颈！因为Reactor既要处理IO操作请求，又要响应连接请求。为了分担Reactor的负担，所以引入了主从Reactor模型。

主从Reactor多线程模型

主从Reactor多线程模型是将Reactor分成两部分，mainReactor负责监听server socket，accept新连接，并将建立的socket分派给subReactor。subReactor负责多路分离已连接的socket，读写网络数据，对业务处理功能，其扔给worker线程池完成。通常，subReactor个数上可与CPU个数等同：



这时可以把Reactor做如下修改：

```
public class Reactor {


    final ServerSocketChannel serverSocketChannel;

    Selector[] selectors; // also create threads
    AtomicInteger next = new AtomicInteger(0);
    ExecutorService sunReactors =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

    public static void main(String[] args) throws IOException {
        new Reactor(1234);
    }

    public Reactor(int port) throws IOException {
        serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new InetSocketAddress(port));
        serverSocketChannel.configureBlocking(false);
        selectors = new Selector[4];
        for (int i = 0; i < 4; i++) {
```

```
            Selector selector = Selector.open();
            selectors[i] = selector;
            SelectionKey key = serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
            key.attach(new Acceptor());
            new Thread(() -> {
                while (!Thread.interrupted()) {
                    try {
                        selector.select();
                        Set<SelectionKey> selectionKeys =
selector.selectedKeys();
                        for (SelectionKey selectionKey : selectionKeys) {
                            dispatch(selectionKey);
                        }
                        selectionKeys.clear();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }).start();
        }

    }



    private void dispatch(SelectionKey selectionKey) {
        Runnable run = (Runnable) selectionKey.attachment();
        if (run != null) {
            run.run();
        }
    }

    class Acceptor implements Runnable {

        @Override
        public void run() {
            try {
                SocketChannel connection = serverSocketChannel.accept();
                if (connection != null)
                    sunReactors.execute(new
Handler(selectors[next.getAndIncrement() % selectors.length], connection));
            } catch (IOException e) {
                e.printStackTrace();
            }

        }

    }


}
```

可见，主Reactor用于响应连接请求，从Reactor用于处理IO操作请求。

AIO

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，
对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；
对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。
即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。
在JDK1.7中，这部分内容被称作NIO.2，主要在java.nio.channels包下增加了下面四个异步通道：

- AsynchronousSocketChannel
- AsynchronousServerSocketChannel
- AsynchronousFileChannel
- AsynchronousDatagramChannel

我们看一下AsynchronousSocketChannel中的几个方法：

```
public abstract class AsynchronousSocketChannel
    implements AsynchronousByteChannel, NetworkChannel
{
    public abstract Future<Integer> read(ByteBuffer dst);

    public abstract <A> void read(ByteBuffer[] dsts,
                                  int offset,
                                  int length,
                                  long timeout,
                                  TimeUnit unit,
                                  A attachment,
                                  CompletionHandler<Long,? super A> handler);

    public abstract <A> void write(ByteBuffer src,
                                   long timeout,
                                   TimeUnit unit,
                                   A attachment,
                                   CompletionHandler<Integer,? super A>
handler);

    public final <A> void write(ByteBuffer src,
                                A attachment,
                                CompletionHandler<Integer,? super A> handler)

    {
        write(src, 0L, TimeUnit.MILLISECONDS, attachment, handler);
    }

    public abstract Future<Integer> write(ByteBuffer src);

    public abstract <A> void write(ByteBuffer[] srcs,
                                   int offset,
                                   int length,
                                   long timeout,
                                   TimeUnit unit,
                                   A attachment,
                                   CompletionHandler<Long,? super A> handler);
}
```

其中的read/write方法，有的会返回一个`Future`对象，有的需要传入一个`CompletionHandler`对象，该对象的作用是当执行完读取/写入操作后，直接该对象当中的方法进行回调。

对于`AsynchronousSocketChannel`而言，在windows和linux上的实现类是不一样的。

在windows上，AIO的实现是通过IOCP来完成的，实现类是：

```
WindowsAsynchronousSocketChannelImpl
```

实现的接口是：

```
Iocp.OverlappedChannel
```

而在linux上，实现类是：

```
UnixAsynchronousSocketChannelImpl
```

实现的接口是：

```
Port.PollableChannel
```

AIO是一种接口标准，各家操作系统可以实现也可以不实现。在不同操作系统上在高并发情况下最好都采用操作系统推荐的方式。Linux上还没有真正实现网络方式的AIO。

select和epoll的区别

当需要读两个以上的I/O的时候，如果使用阻塞式的I/O，那么可能长时间的阻塞在一个描述符上面，另外的描述符虽然有数据但是不能读出来，这样实时性不能满足要求，大概的解决方案有以下几种：

1. 使用多进程或者多线程，但是这种方法会造成程序的复杂，而且对与进程与线程的创建维护也需要很多的开销（Apache服务器是用的子进程的方式，优点可以隔离用户）；
2. 用一个进程，但是使用非阻塞的I/O读取数据，当一个I/O不可读的时候立刻返回，检查下一个是否可读，这种形式的循环为轮询（polling），这种方法比较浪费CPU时间，因为大多数时间是不可读，但是仍花费时间不断反复执行read系统调用；
3. 异步I/O，当一个描述符准备好的时候用一个信号告诉进程，但是由于信号个数有限，多个描述符时不适用；
4. 一种较好的方式为I/O多路复用，先构造一张有关描述符的列表（epoll中为队列），然后调用一个函数，直到这些描述符中的一个准备好时才返回，返回时告诉进程哪些I/O就绪。select和epoll这两个机制都是多路I/O机制的解决方案，select为POSIX标准中的，而epoll为Linux所特有的。

它们的区别主要有三点：

1. select的句柄数目受限，在linux/posix_types.h头文件有这样的声明：`#define __FD_SETSIZE 1024` 表示select最多同时监听1024个fd。而epoll没有，它的限制是最大的打开文件句柄数目；
2. epoll的最大好处是不会随着FD的数目增长而降低效率，在selec中采用轮询处理，其中的数据结构类似一个数组的数据结构，而epoll是维护一个队列，直接看队列是不是空就可以了。epoll只会对”活跃”的socket进行操作—这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有”活跃”的socket才会主动的去调用 callback函数（把这个句柄加入队列），其他idle状态句柄则不会，在这点上，epoll实现了一个”伪”AIO。但是如果绝大部分的I/O都是“活跃的”，每个I/O端口使用率很高的话，epoll效率不一定比select高（可能是要维护队列复杂）；
3. 使用mmap加速内核与用户空间的消息传递。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。

NIO与epoll

上文说到了select与epoll的区别，再总结一下Java NIO与select和epoll：

- Linux2.6之后支持epoll
- windows支持select而不支持epoll
- 不同系统下nio的实现是不一样的，包括Sunos linux 和windows
- select的复杂度为O(N)
- select有最大fd限制，默认为1024
- 修改sys/select.h可以改变select的fd数量限制
- epoll的事件模型，无fd数量限制，复杂度O(1),不需要遍历fd

以下代码基于Java 8。

下面看下在NIO中Selector的open方法：

```
public static Selector open() throws IOException {
    return SelectorProvider.provider().openSelector();
}
```

这里使用了SelectorProvider去创建一个Selector，看下provider方法的实现：

```
public static SelectorProvider provider() {
    synchronized (lock) {
        if (provider != null)
            return provider;
        return AccessController.doPrivileged(
            new PrivilegedAction<SelectorProvider>() {
                public SelectorProvider run() {
                        if (loadProviderFromProperty())
                            return provider;
                        if (loadProviderAsService())
                            return provider;
                        provider = sun.nio.ch.DefaultSelectorProvider.create();
                        return provider;
                    }
                });
    }
}
```

看下`sun.nio.ch.DefaultSelectorProvider.create()`方法，该方法在不同的操作系统中的代码是不同的，在windows中的实现如下：

```
public static SelectorProvider create() {
    return new WindowsSelectorProvider();
}
```

在Mac OS中的实现如下：

```
public static SelectorProvider create() {
    return new KQueueSelectorProvider();
}
```

在linux中的实现如下：

```
public static SelectorProvider create() {
    String str = (String)AccessController.doPrivileged(new
GetPropertyAction("os.name"));

    if (str.equals("SunOS"))
        return createProvider("sun.nio.ch.DevPollSelectorProvider");
    if (str.equals("Linux"))
        return createProvider("sun.nio.ch.EPollSelectorProvider");
    return new PollSelectorProvider();
}
```

我们看到create方法中是通过区分操作系统来返回不同的Provider的。其中SunOs就是Solaris返回的是DevPollSelectorProvider，对于Linux，返回的Provder是EPollSelectorProvider，其余操作系统，返回的是PollSelectorProvider。

Zero Copy

许多web应用都会向用户提供大量的静态内容，这意味着有很多数据从硬盘读出之后，会原封不动的通过socket传输给用户。

这种操作看起来可能不会怎么消耗CPU，但是实际上它是低效的：

1. kernel把从disk读数据；
2. 将数据传输给application；
3. application再次把同样的内容再传回给处于kernel级的socket。

这种场景下，application实际上只是作为一种低效的中间介质，用来把磁盘文件的数据传给socket。

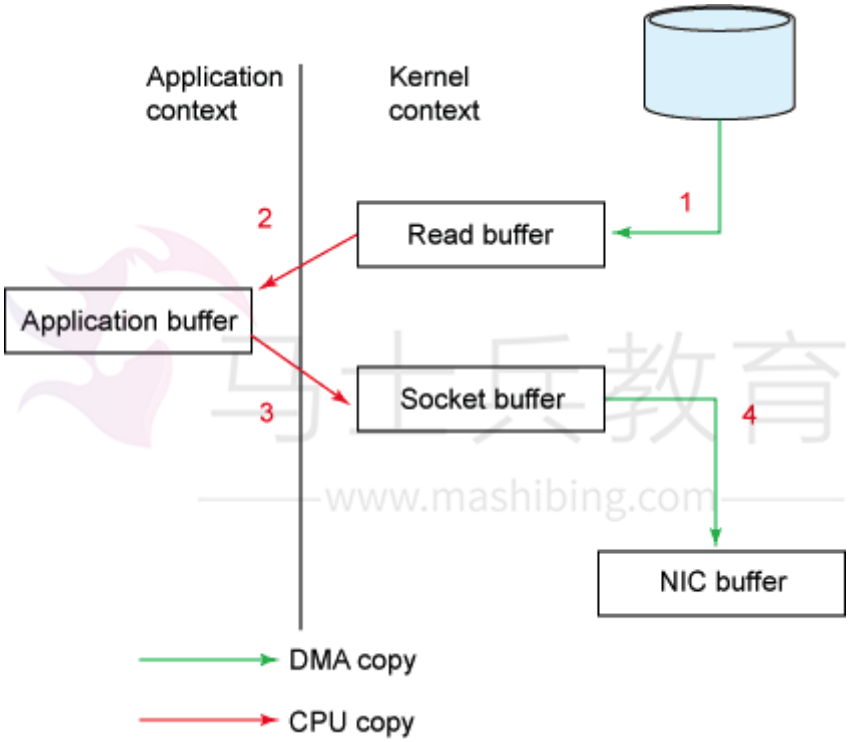
数据每次传输都会经过user和kernel空间都会被copy，这会消耗cpu，并且占用RAM的带宽。

传统的数据传输方式

像这种从文件读取数据然后将数据通过网络传输给其他的程序的方式其核心操作就是如下两个调用：

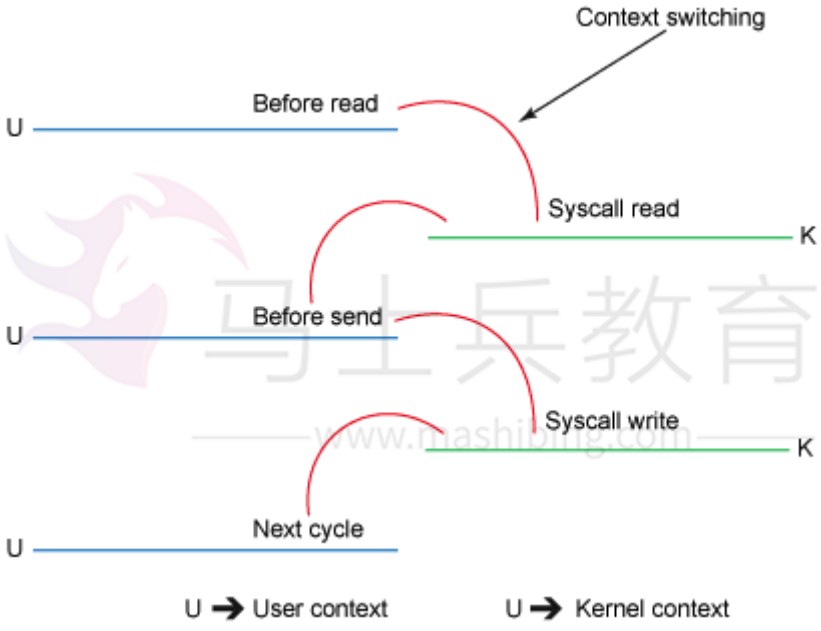
```
File.read(fileDesc,buf,len);
Socket.send(socket,buf,len);
```

其上操作看上去只有两个简单的调用，但是其内部过程却要经历四次用户态和内核态的切换以及四次的数据复制操作：



上图展示了数据从文件到socket的内部流程。

下面看下用户态和内核态的切换过程：



步骤如下：

1. read()的调用引起了从用户态到内核态的切换（看图二），内部是通过sys_read()（或者类似的方法）发起对文件数据的读取。数据的第一次复制是通过DMA（直接内存访问）将磁盘上的数据复制到内核空间的缓冲区中；
2. 数据从内核空间的缓冲区复制到用户空间的缓冲区后，read()方法也就返回了。此时内核态又切换回用户态，现在数据也已经复制到了用户地址空间的缓存中；
3. socket的send()方法的调用又会引起用户态到内核的切换，第三次数据复制又将数据从用户空间缓冲区复制到了内核空间的缓冲区，这次数据被放在了不同于之前的内核缓冲区中，这个缓冲区与数据将要被传输到的socket关联；
4. send()系统调用返回后，就产生了第四次用户态和内核态的切换。随着DMA单独异步的将数据从内核态的缓冲区中传输到协议引擎发送到网络上，有了第四次数据复制。

Zero Copy的数据传输方式

`java.nio.channels.FileChannel`中定义了两个方法：transferTo( )和 transferFrom( )。

transferTo( )和 transferFrom( )方法允许将一个通道交叉连接到另一个通道，而不需要通过一个中间缓冲区来传递数据。只有 FileChannel 类有这两个方法，因此 **channel-to-channel** 传输中通道之一必须是 FileChannel。您不能在 socket 通道之间直接传输数据，不过 socket 通道实现 `WritableByteChannel` 和 `ReadableByteChannel` 接口，因此文件的内容可以用 `transferTo( )` 方法传输给一个 socket 通道，或者也可以用 transferFrom( )方法将数据从一个 socket 通道直接读取到一个文件中。

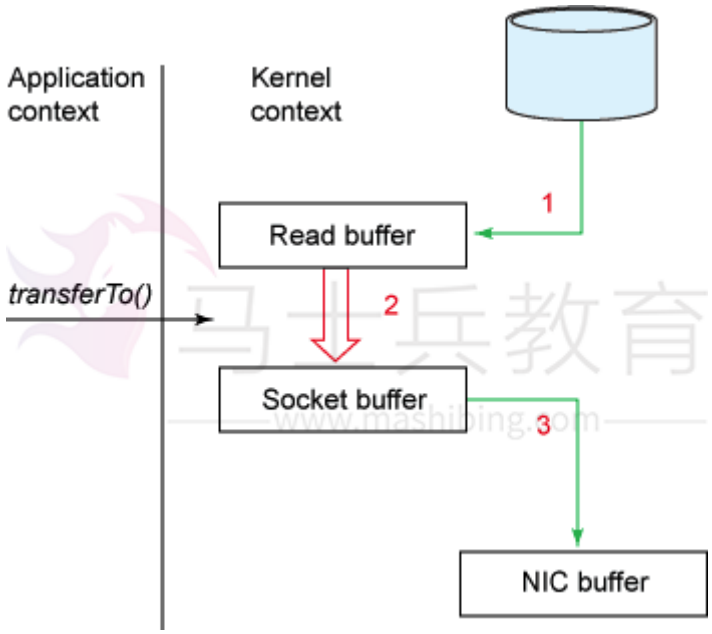
下面根据`transferTo()`方法来说明。

根据上文可知，`transferTo()`方法可以把bytes直接从调用它的channel传输到另一个WritableByteChannel，中间不经过应用程序。

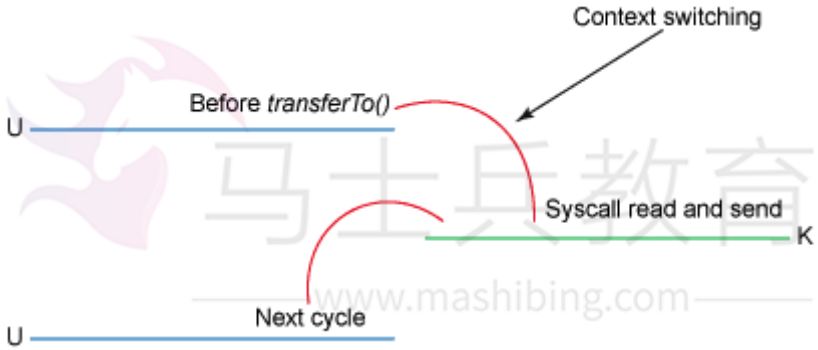
下面看下该方法的定义：

```
public abstract long transferTo(long position, long count,
                                WritableByteChannel target)
        throws IOException;
```

下图展示了通过transferTo实现数据传输的路径：



下图展示了内核态、用户态的切换情况：



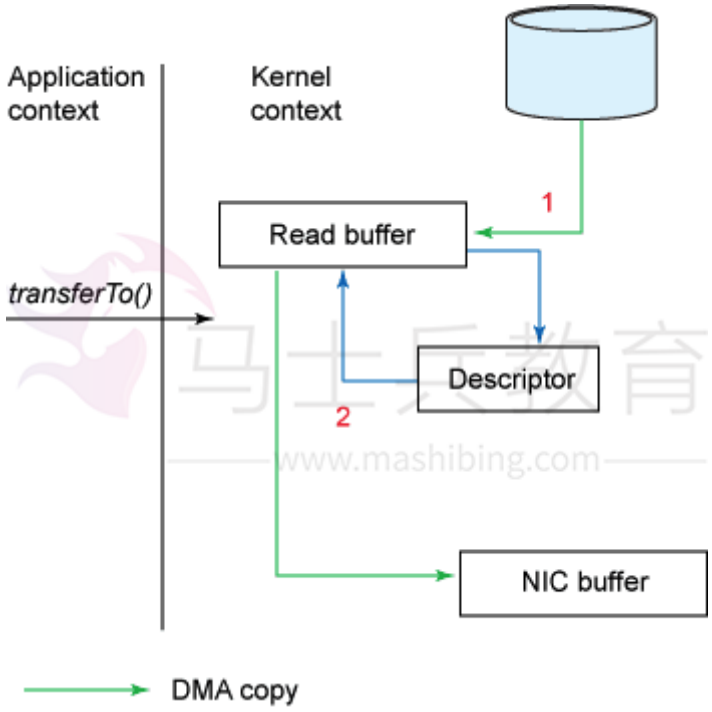
使用transferTo()方式所经历的步骤：

1. transferTo调用会引起DMA将文件内容复制到读缓冲区(内核空间的缓冲区)，然后数据从这个缓冲区复制到另一个与socket输出相关的内核缓冲区中；

2. 第三次数据复制就是DMA把socket关联的缓冲区中的数据复制到协议引擎上发送到网络上。

这次改善，我们是通过将内核、用户态切换的次数从四次减少到两次，将数据的复制次数从四次减少到三次(只有一次用到cpu资源)。但这并没有达到我们零复制的目标。如果底层网络适配器支持收集操作的话，我们可以进一步减少内核对数据的复制次数。在内核为2.4或者以上版本的linux系统上，socket缓冲区描述符将被用来满足这个需求。这个方式不仅减少了内核用户态间的切换，而且也省去了那次需要cpu参与的复制过程。从用户角度来看依旧是调用transferTo()方法，但是其本质发生了变化：

1. 调用transferTo方法后数据被DMA从文件复制到了内核的一个缓冲区中；
2. 数据不再被复制到socket关联的缓冲区中了，仅仅是将一个描述符（包含了数据的位置和长度等信息）追加到socket关联的缓冲区中。DMA直接将内核中的缓冲区中的数据传输给协议引擎，消除了仅剩的一次需要cpu周期的数据复制。



NIO存在的问题

使用NIO != 高性能，当连接数<1000，并发程度不高或者局域网环境下NIO并没有显著的性能优势。

NIO并没有完全屏蔽平台差异，它仍然是基于各个操作系统的I/O系统实现的，差异仍然存在。使用NIO做网络编程构建事件驱动模型并不容易，陷阱重重。

推荐大家使用成熟的NIO框架，如Netty，MINA等。解决了很多NIO的陷阱，并屏蔽了操作系统的差异，有较好的性能和编程模型。

总结

最后总结一下NIO有哪些优势：

- 事件驱动模型
- 避免多线程
- 单线程处理多任务
- 非阻塞I/O，I/O读写不再阻塞
- 基于block的传输，通常比基于流的传输更高效
- 更高级的IO函数，Zero Copy
- I/O多路复用大大提高了Java网络应用的可伸缩性和实用性

三个channel使用ServerSocketChannel||SocketChannel||FileChannel

Java NIO系列教程 FileChannel

Java NIO中的FileChannel是一个连接到文件的通道。可以通过文件通道读写文件。

FileChannel无法设置为非阻塞模式，它总是运行在阻塞模式下。

打开FileChannel

在使用FileChannel之前，必须先打开它。但是，我们无法直接打开一个FileChannel，需要通过使用一个InputStream、OutputStream或RandomAccessFile来获取一个FileChannel实例。下面是通过RandomAccessFile打开FileChannel的示例：

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
```

从FileChannel读取数据

调用多个read()方法之一从FileChannel中读取数据。如：

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);
```

首先，分配一个Buffer。从FileChannel中读取的数据将被读到Buffer中。

然后，调用FileChannel.read()方法。该方法将数据从FileChannel读取到Buffer中。read()方法返回的int值表示了有多少字节被读到了Buffer中。如果返回-1，表示到了文件末尾。

向FileChannel写数据

使用FileChannel.write()方法向FileChannel写数据，该方法的参数是一个Buffer。如：

```
String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {
    channel.write(buf);
}
```

注意FileChannel.write()是在while循环中调用的。因为无法保证write()方法一次能向FileChannel写入多少字节，因此需要重复调用write()方法，直到Buffer中已经没有尚未写入通道的字节。

关闭FileChannel

用完FileChannel后必须将其关闭。如：

```
channel.close();
```

FileChannel的position方法

有时可能需要在FileChannel的某个特定位置进行数据的读/写操作。可以通过调用position()方法获取FileChannel的当前位置。

也可以通过调用position(long pos)方法设置FileChannel的当前位置。

这里有两个例子:

```
long pos = channel.position();
channel.position(pos +123);
```

如果将位置设置在文件结束符之后，然后试图从文件通道中读取数据，读方法将返回-1 —— 文件结束标志。

如果将位置设置在文件结束符之后，然后向通道中写数据，文件将撑大到当前位置并写入数据。这可能导致“文件空洞”，磁盘上物理文件中写入的数据间有空隙。

FileChannel的size方法

FileChannel实例的size()方法将返回该实例所关联文件的大小。如:

```
long fileSize = channel.size();
```

FileChannel的truncate方法

可以使用FileChannel.truncate()方法截取一个文件。截取文件时，文件将中指定长度后面的部分将被删除。如：

```
channel.truncate(``1024``);
```

这个例子截取文件的前1024个字节。

FileChannel的force方法

FileChannel.force()方法将通道里尚未写入磁盘的数据强制写到磁盘上。出于性能方面的考虑，操作系统会将数据缓存在内存中，所以无法保证写入到FileChannel里的数据一定会即时写到磁盘上。要保证这一点，需要调用force()方法。

force()方法有一个boolean类型的参数，指明是否同时将文件元数据（权限信息等）写到磁盘上。

下面的例子同时将文件数据和元数据强制写到磁盘上：

```
channel.force(``true``);
```

Java NIO系列教程 SocketChannel

Java NIO中的SocketChannel是一个连接到TCP网络套接字的通道。可以通过以下2种方式创建SocketChannel：

1. 打开一个SocketChannel并连接到互联网上的某台服务器。
2. 一个新连接到达ServerSocketChannel时，会创建一个SocketChannel。

打开 SocketChannel

下面是SocketChannel的打开方式：

```
SocketChannel socketChannel = SocketChannel.open();
```

```
socketChannel.connect(``new InetSocketAddress(``"http://jenkov.com"``, ``80``));
```

关闭 SocketChannel

当用完SocketChannel之后调用SocketChannel.close()关闭SocketChannel：

```
socketChannel.close();
```

从 SocketChannel 读取数据

要从SocketChannel中读取数据，调用一个read()的方法之一。以下是例子：

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);
```

首先，分配一个Buffer。从SocketChannel读取到的数据将会放到这个Buffer中。

然后，调用SocketChannel.read()。该方法将数据从SocketChannel 读到Buffer中。read()方法返回的int值表示读了多少字节进Buffer里。如果返回的是-1，表示已经读到了流的末尾（连接关闭了）。

写入 SocketChannel

写数据到SocketChannel用的是SocketChannel.write()方法，该方法以一个Buffer作为参数。示例如下：

```
String newData = "New String to write to file..." +System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {
    channel.write(buf);
}
```

注意SocketChannel.write()方法的调用是在一个while循环中的。Write()方法无法保证能写多少字节到SocketChannel。所以，我们重复调用write()直到Buffer没有要写的字节为止。

非阻塞模式

可以设置 SocketChannel 为非阻塞模式（non-blocking mode）.设置之后，就可以在异步模式下调用connect(), read() 和write()了。

connect()

如果SocketChannel在非阻塞模式下，此时调用connect()，该方法可能在连接建立之前就返回了。为了确定连接是否建立，可以调用finishConnect()的方法。像这样：

```
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));
    while(! socketChannel.finishConnect() ){
        //wait, or do something else...
    }
```

write()

非阻塞模式下，write()方法在尚未写出任何内容时可能就返回了。所以需要在循环中调用write()。前面已经有例子了，这里就不赘述了。

read()

非阻塞模式下,read()方法在尚未读取到任何数据时可能就返回了。所以需要关注它的int返回值，它会告诉你读取了多少字节。

非阻塞模式与选择器

非阻塞模式与选择器搭配会工作的更好，通过将一或多个SocketChannel注册到Selector，可以询问选择器哪个通道已经准备好了读取，写入等。Selector与SocketChannel的搭配使用会在后面详讲。

Java NIO系列教程 ServerSocketChannel

Java NIO中的 ServerSocketChannel 是一个可以监听新进来的TCP连接的通道, 就像标准IO中的ServerSocket一样。ServerSocketChannel类在 java.nio.channels包中。

这里有个例子：

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));

while(true){
  SocketChannel socketChannel =
        serverSocketChannel.accept();

  //do something with socketChannel...
}
```

打开 ServerSocketChannel

通过调用 ServerSocketChannel.open() 方法来打开ServerSocketChannel.如：

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
```

关闭 ServerSocketChannel

通过调用ServerSocketChannel.close() 方法来关闭ServerSocketChannel. 如：

```
serverSocketChannel.close();
```

监听新进来的连接

通过 ServerSocketChannel.accept() 方法监听新进来的连接。当 accept()方法返回的时候,它返回一个包含新进来的连接的 SocketChannel。因此, accept()方法会一直阻塞到有新连接到达。

通常不会仅仅只监听一个连接,在while循环中调用 accept()方法. 如下面的例子：

```
while(true){
  SocketChannel socketChannel =
        serverSocketChannel.accept();

  //do something with socketChannel...
}
```

当然,也可以在while循环中使用除了true以外的其它退出准则。

非阻塞模式

ServerSocketChannel可以设置成非阻塞模式。在非阻塞模式下，accept() 方法会立刻返回，如果还没有新进来的连接,返回的将是null。 因此，需要检查返回的SocketChannel是否是null.如：

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);

while(true){
    SocketChannel socketChannel =
        serverSocketChannel.accept();
    if(socketChannel != null){
        //do something with socketChannel...
    }
}
```

String 编码UTF-8 和GBK的区别

- GBK编码：是指中国的中文字符，其实它包含了简体中文与繁体中文字符，另外还有一种字符“gb2312”，这种字符仅能存储简体中文字符。
- UTF-8编码：它是一种全国家通过的一种编码，如果你的网站涉及到多个国家的语言，那么建议你选择UTF-8编码。

GBK和UTF8有什么区别?

UTF8编码格式很强大，支持所有国家的语言，正是因为它的强大，才会导致它占用的空间大小要比GBK大，对于网站打开速度而言，也是有一定影响的。
GBK编码格式，它的功能少，仅限于中文字符，当然它所占用的空间大小会随着它的功能而减少，打开网页的速度比较快。

什么时候使用字节流、什么时候使用字符流

什么时候使用字节流、什么时候使用字符流，二者的区别

先来看一下流的概念：
在程序中所有的数据都是以流的方式进行传输或保存的，程序需要数据的时候要使用输入流读取数据，而当程序需要将一些数据保存起来的时候，就要使用输出流完成。

InputStream 和OutputStream,两个是为**字节流**设计的,主要用来处理字节或二进制对象,
Reader和 Writer.两个是为**字符流**（一个字符占两个字节）设计的,主要用来处理字符或字符串.

字符流处理的单元为2个字节的Unicode字符，操作字符、字符数组或字符串，
字节流处理单元为1个字节，操作字节和字节数组。
所以字符流是由Java虚拟机将字节转化为2个字节的Unicode字符为单位的字符而成的，
所以它对多国语言支持性比较好！
如果是音频文件、图片、歌曲，就用字节流好点，
如果是关系到中文（文本）的，用字符流好点

所有文件的储存是都是字节（byte）的储存，在磁盘上保留的并不是文件的字符而是先把字符编码成字节，再储存这些字节到磁盘。在读取文件（特别是文本文件）时，也是一个字节一个字节地读取以形成字节序列
字节流可用于任何类型的对象，包括二进制对象，而字符流只能处理字符或者字符串；
字节流提供了处理任何类型的IO操作的功能，但它不能直接处理Unicode字符，而字符流就可以
字节流是最基本的，所有的InputStrem和OutputStream的子类都是,主要用在处理二进制数据，它是按字节来处理的
但实际中很多的数据是文本，
又提出了字符流的概念，
它是按虚拟机的encode来处理，也就是要进行字符集的转化
这两个之间通过 InputStreamReader,OutputStreamWriter来关联，
实际上是通过byte[]和String来关联
在实际开发中出现的汉字问题实际上都是在字符流和字节流之间转化不统一而造成的

Reader类的read()方法返回类型为int： 作为整数读取的字符（占两个字节共16位），范围在 0 到 65535 之间 (0x00-0xffff)，如果已到达流的末尾，则返回 -1

inputStream的read()虽然也返回int，但由于此类是面向字节流的，一个字节占8个位，所以返回 0 到 255 范围内的 int 字节值。如果因为已经到达流末尾而没有可用的字节，则返回值 -1。因此对于不能用0-255来表示的值就得用字符流来读取！比如说汉字.

字节流和字符流的主要区别是什么呢?

> **一.字节流在操作时不会用到缓冲区（内存），是直接对文件本身进行操作的。而字符流在操作时使用了缓冲区，通过缓冲区再操作文件。**

二.在硬盘上的所有文件都是以字节形式存在的（图片，声音，视频），而字符值在内存中才会形成。

上面两点能说明什么呢?

针对第一点，

我们知道，如果一个程序频繁对一个资源进行IO操作，效率会非常低。此时，通过缓冲区，先把需要操作的数据暂时放入内存中，以后直接从内存中读取数据，则可以避免多次的IO操作，提高效率

针对第二点，

真正存储和传输数据时都是以字节为单位的，字符只是存在与内存当中的，所以，字节流适用范围更为宽广

递归读取文件夹下的文件，代码怎么实现

```
/**
 * 递归读取文件夹下的 所有文件
 *
 * @param testFileDir 文件名或目录名
 */
private static void testLoopOutAllFileName(String testFileDir) {
    if (testFileDir == null) {
        //因为new File(null)会空指针异常,所以要判断下
        return;
    }
    File[] testFile = new File(testFileDir).listFiles();
    if (testFile == null) {
        return;
    }
    for (File file : testFile) {
        if (file.isFile()) {
            System.out.println(file.getName());
        } else if (file.isDirectory()) {
            System.out.println("-------this is a directory, and its files are as
follows:-------");
            testLoopOutAllFileName(file.getPath());
        } else {
            System.out.println("文件读入有误! ");
        }
    }
```

```
}
```

SynchronousQueue实现原理

前言

SynchronousQueue是一个比较特别的队列，由于在线程池方面有所应用，为了更好的理解线程池的实现原理，笔者花了些时间学习了一下该队列源码(JDK1.8)，此队列源码中充斥着大量的CAS语句，理解起来是有些难度的，为了方便日后回顾，本篇文章会以简洁的图形化方式展示该队列底层的实现原理。

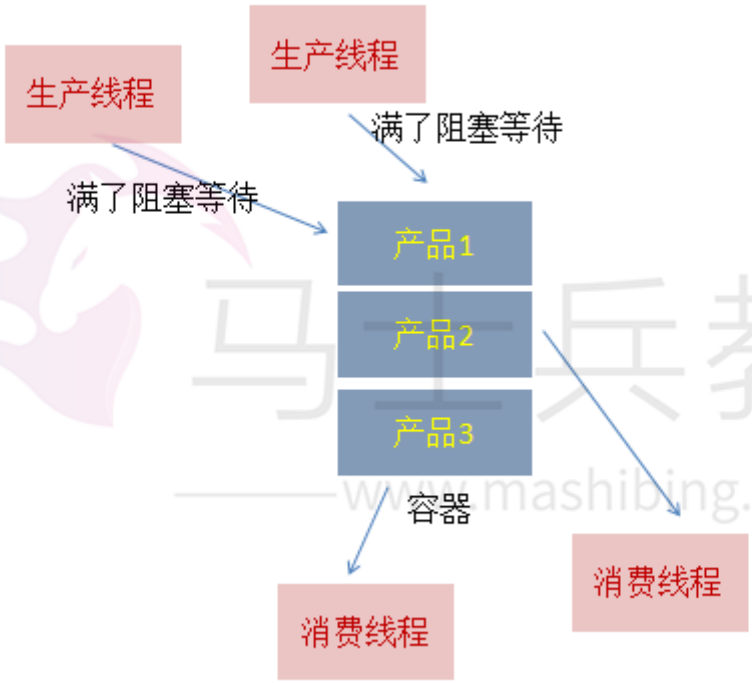
SynchronousQueue简单使用

经典的生产者-消费者模式，操作流程是这样的：

有多个生产者，可以并发生产产品，把产品置入队列中，如果队列满了，生产者就会阻塞；

有多个消费者，并发从队列中获取产品，如果队列空了，消费者就会阻塞；

如下面的示意图所示：



SynchronousQueue
也是一个队列来的，但它的特别之处在于它内部没有容器，一个生产线程，当它生产产品（即put的时候），如果当前没有人想要消费产品(即当前没有线程执行take)，此生产线程必须阻塞，等待一个消费线程调用take操作，take操作将会唤醒该生产线程，同时消费线程会获取生产线程的产品（即数据传递），这样的一个过程称为一次配对过程(当然也可以先take后put,原理是一样的)。

我们用一个简单的代码来验证一下，如下所示：

```
package com.concurrent;
```

```
import java.util.concurrent.SynchronousQueue;

public class SynchronousQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        final SynchronousQueue<Integer> queue = new SynchronousQueue<Integer>();

        Thread putThread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("put thread start");
                try {
                    queue.put(1);
                } catch (InterruptedException e) {
                }
                System.out.println("put thread end");
            }
        });

        Thread takeThread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("take thread start");
                try {
                    System.out.println("take from putThread: " + queue.take());
                } catch (InterruptedException e) {
                }
                System.out.println("take thread end");
            }
        });

        putThread.start();
        Thread.sleep(1000);
        takeThread.start();
    }
}
```

一种输出结果如下:

```
put thread start
take thread start
take from putThread: 1
put thread end
take thread end
```

从结果可以看出，put线程执行queue.put(1) 后就被阻塞了，只有take线程进行了消费，put线程才可以返回。可以认为这是一种线程与线程间一对一传递消息的模型。

SynchronousQueue实现原理

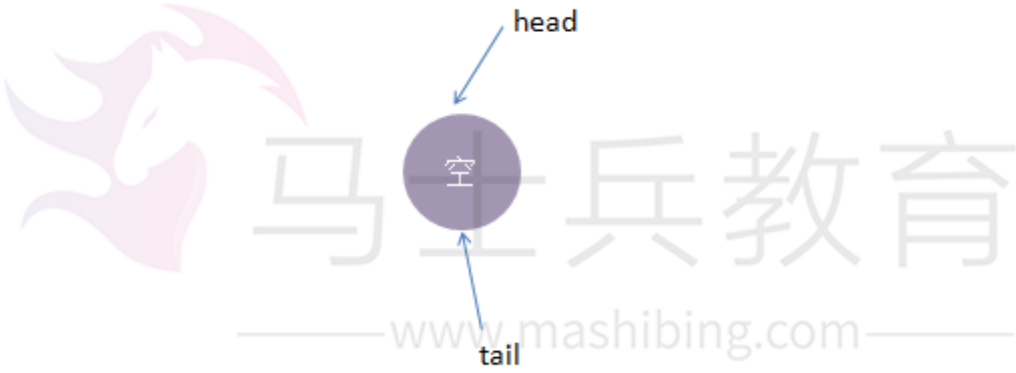
不像ArrayBlockingQueue、LinkedBlockingDeque之类的阻塞队列依赖AQS实现并发操作，SynchronousQueue直接使用CAS实现线程的安全访问。由于源码中充斥着大量的CAS代码，不易于理解，所以按照笔者的风格，接下来会使用简单的示例来描述背后的实现模型。

队列的实现策略通常分为公平模式和非公平模式，接下来将分别进行说明。

公平模式下的模型：

公平模式下，底层实现使用的是TransferQueue这个内部队列，它有一个head和tail指针，用于指向当前正在等待匹配的线程节点。

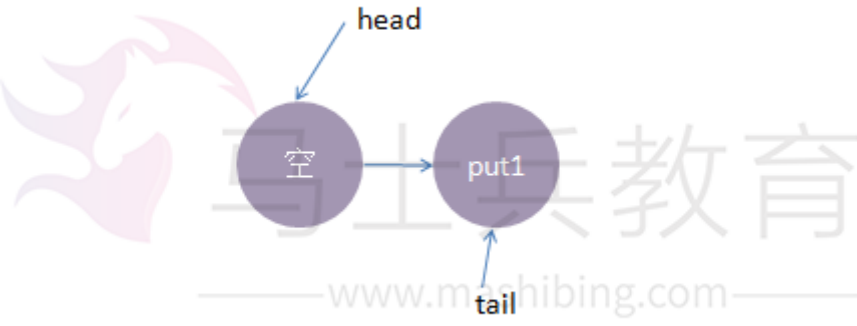
初始化时，TransferQueue的状态如下：



马士兵教育
www.mashibing.com

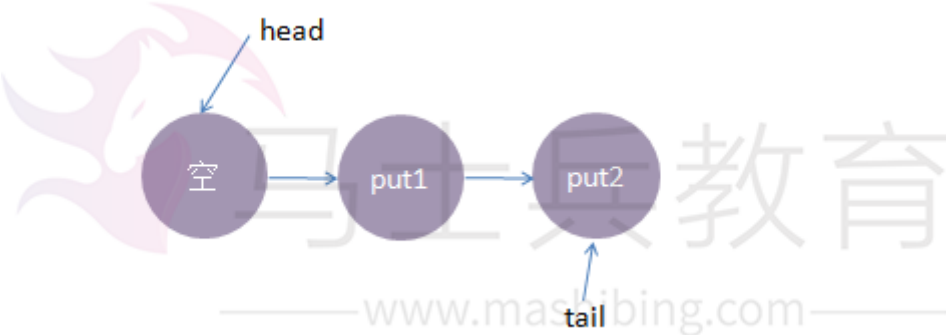
接着我们进行一些操作：

1、线程put1执行 put(1)操作，由于当前没有配对的消费线程，所以put1线程入队列，自旋一小会后睡眠等待，这时队列状态如下：



马士兵教育
www.mashibing.com

2、接着，线程put2执行了put(2)操作，跟前面一样，put2线程入队列，自旋一小会后睡眠等待，这时队列状态如下：

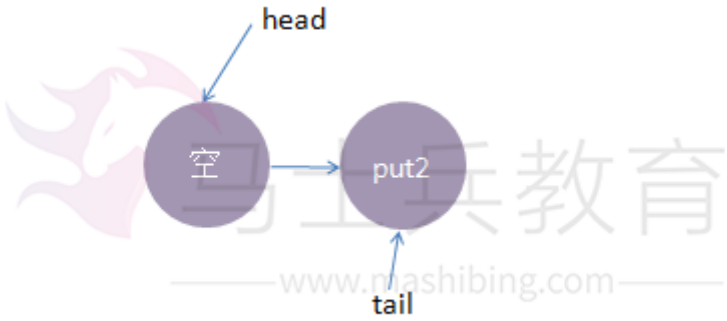


3、这时候，来了一个线程take1，执行了take操作，由于tail指向put2线程，put2线程跟take1线程配对了(一put一take)，这时take1线程不需要入队，但是请注意了，这时候，要唤醒的线程并不是put2，而是put1。为何?

大家应该知道我们现在讲的是公平策略，所谓公平就是谁先入队了，谁就优先被唤醒，我们的例子明显是put1应该优先被唤醒。至于读者可能会有一个疑问，明明是take1线程跟put2线程匹配上了，结果是put1线程被唤醒消费，怎么确保take1线程一定可以和次首节点(head.next)也是匹配的呢？其实大家可以拿个纸画一画，就会发现真的就是这样的。

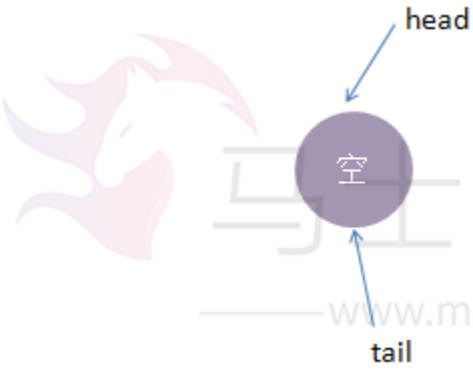
公平策略总结下来就是：队尾匹配队头出队。

执行后put1线程被唤醒，take1线程的 take()方法返回了1(put1线程的数据)，这样就实现了线程间的一对一通信，这时候内部状态如下：



4、最后，再来一个线程take2，执行take操作，这时候只有put2线程在等候，而且两个线程匹配上了，线程put2被唤醒，

take2线程take操作返回了2(线程put2的数据)，这时候队列又回到了起点，如下所示：



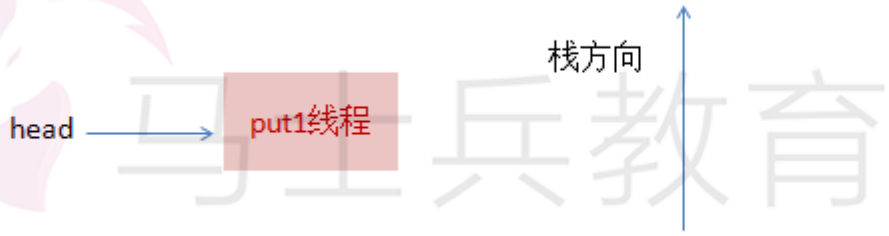
以上便是公平模式下，SynchronousQueue的实现模型。总结下来就是：队尾匹配队头出队，先进先出，体现公平原则。

非公平模式下的模型：

我们还是使用跟公平模式下一样的操作流程，对比两种策略下有何不同。非公平模式底层的实现使用的是TransferStack，

一个栈，实现中用head指针指向栈顶，接着我们看看它的实现模型:

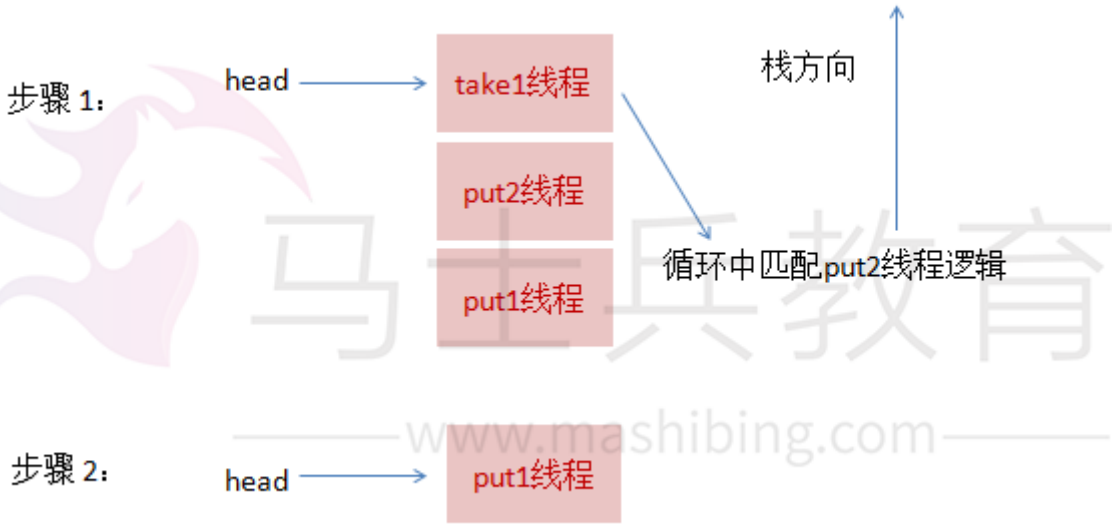
1、线程put1执行 put(1)操作，由于当前没有配对的消费线程，所以put1线程入栈，自旋一小会后睡眠等待，这时栈状态如下：



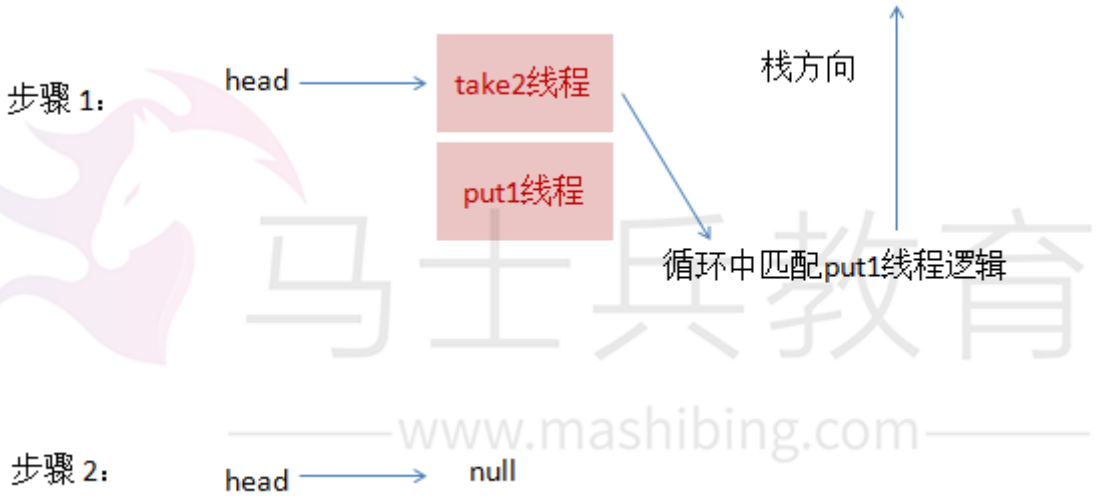
2、接着，线程put2再次执行了put(2)操作，跟前面一样，put2线程入栈，自旋一小会后睡眠等待，这时栈状态如下：



3、这时候，来了一个线程take1，执行了take操作，这时候发现栈顶为put2线程，匹配成功，但是实现会先把take1线程入栈，然后take1线程循环执行匹配put2线程逻辑，一旦发现没有并发冲突，就会把栈顶指针直接指向 put1线程



4、最后，再来一个线程take2，执行take操作，这跟步骤3的逻辑基本是一致的，take2线程入栈，然后在循环中匹配put1线程，最终全部匹配完毕，栈变为空，恢复初始状态，如下图所示：



可以从上面流程看出，虽然put1线程先入栈了，但是却是后匹配，这就是非公平的由来。

总结

SynchronousQueue由于其独有的线程一一配对通信机制，在大部分平常开发中，可能都不太会用到，但线程池技术中会有所使用，由于内部没有使用AQS，而是直接使用CAS，所以代码理解起来会比较困难，但这并不妨碍我们理解底层的实现模型，在理解了模型的基础上，有兴趣的话再查阅源码，就会有方向感，看起来也会比较容易，希望本文有所借鉴意义。

自定义类加载器

为什么需要自定义类加载器

网上的大部分自定义类加载器文章，几乎都是贴一段实现代码，然后分析一两句自定义ClassLoader的原理。但是我觉得首先得把为什么需要自定义加载器这个问题搞清楚，因为如果不明白它的作用的情况下，还要去学习它显然是很让人困惑的。

首先介绍自定义类的应用场景：

（1）加密：Java代码可以轻易的被反编译，如果你需要把自己的代码进行加密以防止反编译，可以先将编译后的代码用某种加密算法加密，类加密后就不能再用Java的ClassLoader去加载类了，这时就需要自定义ClassLoader在加载类的时候先解密类，然后再加载。

（2）从非标准的来源加载代码：如果你的字节码是放在数据库、甚至是在云端，就可以自定义类加载器，从指定的来源加载类。

（3）以上两种情况在实际中的综合运用：比如你的应用需要通过网络来传输 Java 类的字节码，为了安全性，这些字节码经过了加密处理。这个时候你就需要自定义类加载器来从某个网络地址上读取加密后的字节代码，接着进行解密和验证，最后定义出在Java虚拟机中运行的类。

1. 双亲委派模型

在实现自己的ClassLoader之前，我们先了解一下系统是如何加载类的，那么就不得不介绍双亲委派模型的实现过程。

```
//双亲委派模型的工作过程源码
protected synchronized Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException{
    // First, check if the class has already been loaded
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClassOrNull(name);
            }
        }
        catch (ClassNotFoundException e) {
            // ClassNotFoundException thrown if class not found
            // from the non-null parent class loader
            //父类加载器无法完成类加载请求
        }

        if (c == null) {
            // If still not found, then invoke findClass in order to find the
class
            //子加载器进行类加载
            c = findClass(name);
        }
    }

    if (resolve) {
        //判断是否需要链接过程，参数传入
        resolveClass(c);
    }
```

```
    return c;
}
```

双亲委派模型的工作过程如下：

（1）当前类加载器从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

（2）如果没有找到，就去委托父类加载器去加载（如代码c = parent.loadClass(name, false)所示）。父类加载器也会采用同样的策略，查看自己已经加载过的类中是否包含这个类，有就返回，没有就委托父类的父类去加载，一直到启动类加载器。因为如果父加载器为空了，就代表使用启动类加载器作为父加载器去加载。

（3）如果启动类加载器加载失败（例如在$JAVA_HOME/jre/lib里未查找到该class），则会抛出一个异常ClassNotFoundException，然后再调用当前加载器的findClass()方法进行加载。

双亲委派模型的好处：

（1）主要是为了安全性，避免用户自己编写的类动态替换 Java的一些核心类，比如 String。

（2）同时也避免了类的重复加载，因为 JVM中区分不同类，不仅仅是根据类名，相同的 class文件被不同的 ClassLoader加载就是不同的两个类。

2. 自定义类加载器

（1）从上面源码看出，调用loadClass时会先根据委派模型在父加载器中加载，如果加载失败，则会调用当前加载器的findClass来完成加载。

（2）因此我们自定义的类加载器只需要继承ClassLoader，并覆盖findClass方法，下面是一个实际例子，在该例中我们用自定义的类加载器去加载我们事先准备好的class文件。

2.1自定义一个People.java类做例子

```
public class People {
//该类写在记事本里，在用javac命令行编译成class文件，放在d盘根目录下
    private String name;

    public People() {}

    public People(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString() {
        return "I am a people, my name is " + name;
```

```
    }

}
```

2.2自定义类加载器

自定义一个类加载器，需要继承ClassLoader类，并实现findClass方法。其中defineClass方法可以把二进制流字节组成的文件转换为一个java.lang.Class（只要二进制字节流的内容符合Class文件规范）。

```
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.nio.ByteBuffer;
import java.nio.channels.Channels;
import java.nio.channels.FileChannel;
import java.nio.channels.WritableByteChannel;

public class MyClassLoader extends ClassLoader
{
    public MyClassLoader()
    {


    }

    public MyClassLoader(ClassLoader parent)
    {
        super(parent);
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException
    {
        File file = new File("D:/People.class");
        try{
            byte[] bytes = getClassBytes(file);
            //defineClass方法可以把二进制流字节组成的文件转换为一个java.lang.Class
            Class<?> c = this.defineClass(name, bytes, 0, bytes.length);
            return c;
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        return super.findClass(name);
    }

    private byte[] getClassBytes(File file) throws Exception
    {
        // 这里要读入.class的字节，因此要使用字节流
        FileInputStream fis = new FileInputStream(file);
        FileChannel fc = fis.getChannel();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        WritableByteChannel wbc = Channels.newChannel(baos);
        ByteBuffer by = ByteBuffer.allocate(1024);

        while (true){
```

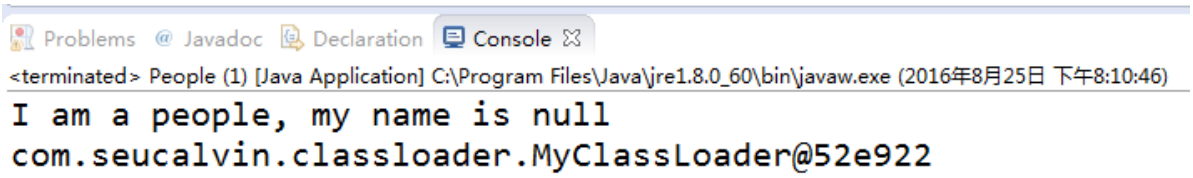
```
            int i = fc.read(by);
            if (i == 0 || i == -1)
            break;
            by.flip();
            wbc.write(by);
            by.clear();
        }
        fis.close();
        return baos.toByteArray();
    }
}
```

2.3在主函数里使用

```
MyClassLoader mcl = new MyClassLoader();
Class<?> clazz = Class.forName("People", true, mcl);
Object obj = clazz.newInstance();

System.out.println(obj);
System.out.println(obj.getClass().getClassLoader());//打印出我们的自定义类加载器
```

2.4运行结果



至此关于自定义ClassLoader的内容总结完毕。

面向对象和面向过程的区别

面向过程

优点： 性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗

资源;比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发，性能是最重要的因素。

缺点： 没有面向对象易维护、易复用、易扩展

面向对象

优点： 易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

缺点： 性能比面向过程低

Java 语言有哪些特点

1. 简单易学；
2. 面向对象（封装，继承，多态）；
3. 平台无关性（ Java 虚拟机实现平台无关性）；
4. 可靠性；
5. 安全性；
6. 支持多线程（ C++ 语言没有内置的多线程机制，因此必须调用操作系

统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；

7. 支持网络编程并且很方便（ Java 语言诞生本身就是为简化网络编程设

计的，因此 Java 语言不仅支持网络编程而且很方便）；

8. 编译与解释并存；

关于 JVM JDK 和 JRE 最详细通俗的解答

JVM

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特

定实现（Windows，Linux，macOS），目的是使用相同的字节码，它们都会给出相同的结果。

什么是字节码?采用字节码的好处是什么?

在 Java 中，JVM 可以理解的代码就叫做字节码（即扩展名为 .class 的文

件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码

并不专对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同的计算机上运行。

Java 程序从源代码到运行一般有下面 3 步：

我们需要格外注意的是 .class->机器码 这一步。在这一步 jvm 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。而且，有些方法和代码块是经常需要被调用的，也就是所谓的热点代码，所以后面引进了 JIT 编译器，JIT 属于运行时编译。当 JIT 编译器完成第一

次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什

么经常会说 Java 是编译与解释共存的语言。

HotSpot 采用了惰性评估(Lazy Evaluation)的做法，根据二八定律，消耗大部分系统资源的只有那一小部分的代码（热点代码），而这也就是 JIT 所需要编译

的部分。JVM 会根据代码每次被执行的情况收集信息并相应地做出一些优化，

因此执行的次数越多，它的速度就越快。JDK 9 引入了一种新的编译模式

AOT(Ahead of Time Compilation)，它是直接将字节码编译成机器码，这样就

避免了 JIT 预热等各方面的开销。JDK 支持分层编译和 AOT 协作使用。但是，

AOT 编译器的编译质量是肯定比不上 JIT 编译器的。

总结：Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系

统的特定实现（Windows，Linux，macOS），目的是使用相同的字节码，它们

都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，随处可以运行”的关键所在。

JDK 和 JRE

JDK 是 Java Development Kit，它是功能齐全的 Java SDK。它拥有 JRE 所拥有

的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，

包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。

如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用

程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译

servlet。

Oracle JDK 和 OpenJDK 的对比

可能在看这个问题之前很多人和我一样并没有接触和使用过 OpenJDK 。那么

Oracle 和 OpenJDK 之间是否存在重大差异？下面通过我通过我收集到一些资料对你解答这个被很多人忽视的问题。

对于 Java 7，没什么关键的地方。OpenJDK 项目主要基于 Sun 捐赠的 HotSpot

源代码。此外，OpenJDK 被选为 Java 7 的参考实现，由 Oracle 工程师维护。

关于 JVM，JDK，JRE 和 OpenJDK 之间的区别，Oracle 博客帖子在 2012 年有一个更详细的答案：

问：OpenJDK 存储库中的源代码与用于构建 Oracle JDK 的代码之间有什么区别？

答：非常接近 - 我们的 Oracle JDK 版本构建过程基于 OpenJDK 7 构建，只添

加了几个部分，例如部署代码，其中包括 Oracle 的 Java 插件和 Java WebStart 的实现，以及一些封闭的源代码派对组件，如图形光栅化器，一些开源的第三方组件，如 Rhino，以及一些零碎的东西，如附加文档或第三方字体。展望未

来，我们的目的是开源 Oracle JDK 的所有部分，除了我们考虑商业功能的部

分。

总结：

1. Oracle JDK 版本将每三年发布一次，而 OpenJDK 版本每三个月发布一

次；

2. OpenJDK 是一个参考模型并且是完全开源的，而 Oracle JDK 是

OpenJDK 的一个实现，并不是完全开源的；

3. Oracle JDK 比 OpenJDK 更稳定。OpenJDK 和 Oracle JDK 的代码几乎相同，但 Oracle JDK 有更多的类和一些错误修复。因此，如果您想开发

企业/商业软件，我建议您选择 Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用 OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到 Oracle JDK 就可以解决问题；

4. 顶级公司正在使用 Oracle JDK，例如 Android Studio，Minecraft 和

IntelliJ IDEA 开发工具，其中 Open JDK 不太受欢迎；

5. 在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能；
6. Oracle JDK 不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；
7. Oracle JDK 根据二进制代码许可协议获得许可，而 OpenJDK 根据 GPL v2 许可获得许可。

Java 和 C++的区别

我知道很多人没学过 C++，但是面试官就是没事喜欢拿咱们 Java 和 C++ 比呀！没办法！！！就算没学过 C++，也要记下来！

- 都是面向对象的语言，都支持封装、继承和多态
- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多

继承，但是接口可以多继承。

- Java 有自动内存管理机制，不需要程序员手动释放无用内存

什么是 Java 程序的主类 应用程序和小程序的主类有何不同

一个程序中可以有多个类，但只能有一个类是主类。在 Java 应用程序中，这

个主类是指包含 main（）方法的类。而在 Java 小程序中，这个主类是一个继

承自系统类 JApplet 或 Applet 的子类。应用程序的主类不一定要求是 public

类，但小程序的主类要求必须是 public 类。主类是 Java 程序执行的入口点。

Java 应用程序与小程序之间有那些差别

简单说应用程序是从主线程启动(也就是 main() 方法)。applet 小程序没有

main 方法，主要是嵌在浏览器页面上运行(调用 init()线程或者 run()来启动)，嵌

入浏览器这点跟 flash 的小游戏类似。

字符型常量和字符串常量的区别

1. 形式上: 字符常量是单引号引起的一个字符 字符串常量是双引号引起的

若干个字符

2. 含义上: 字符常量相当于一个整形值(ASCII 值),可以参加表达式运算 字符串常量代表一个地址值(该字符串在内存中存放位置)
3. 占内存大小 字符常量只占 2 个字节 字符串常量占若干个字节(至少一个

字符结束标志) (**注意： char 在 Java 中占两个字节**)

java 编程思想第四版：2.2.2 节

Java要确定每种基本类型所占存储空间的大小。它们的大小并不像其机器硬件架构的变化而变化。这种所占存储空间大小的不变性是Java程序编写的程序更具可移植性的原因之一。

基本类型	大小	最小值	最大值	包装器类
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2^{15}	$+2^{15}-1$	Short
int	32 bits	-2^{31}	$+2^{31}-1$	Integer
long	64 bits	-2^{63}	$+2^{63}-1$	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void	—	—	—	Void

构造器 Constructor 是否可被 override

在讲继承的时候我们就知道父类的私有属性和构造方法并不能被继承，所以

Constructor 也就不能被 override（重写）,但是可以 overload（重载）,所以你可以看到一个类中有多个构造函数的情况。

重载和重写的区别

重载： 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

重写： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

Java 面向对象编程三大特性: 封装 继承 多态

封装

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

继承

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。**关于继承如下 3 点请记住：**

1. 子类拥有父类非 private 的属性和方法。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。（以后介绍）。

多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在 Java 中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

String StringBuffer 和 StringBuilder 的区别是什么 String 为什么是不可变的

可变性

简单的来说：String 类中使用 final 关键字字符数组保存字符串，private

final　char　value[]，所以 String 对象是不可变的。而 StringBuilder 与

StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中

也是使用字符数组保存字符串char[]value 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

StringBuilder 与 StringBuffer 的构造方法都是调用父类构造方法也就是 AbstractStringBuilder 实现的，大家可以自行查阅源码。

AbstractStringBuilder.java

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
char[] value;
    int count;
    AbstractStringBuilder() {
    }
    AbstractStringBuilder(int capacity) {
    value = new char[capacity];
    }
```

线程安全性

String 中的对象是不可变的，也就可以理解为常量，线程安全。

AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共方法。StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。　　**性能**

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用

StringBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。**对于三者使用的总结：**

1. 操作少量的数据 = String
2. 单线程操作字符串缓冲区下操作大量数据 = StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据 = StringBuffer

自动装箱与拆箱

装箱：将基本类型用它们对应的引用类型包装起来；**拆箱**：将包装类型转换为基本数据类型；

在一个静态方法内调用一个非静态成员为什么是非法的

由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静态变量，也不可以访问非静态变量成员。

在 Java 中定义一个不做事且没有参数的构造方法的作用

Java 程序在执行子类的构造方法之前，如果没有用 super() 来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 super() 来调用父类

中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

import java 和 javax 有什么区别

刚开始的时候 JavaAPI 所必需的包是 java 开头的包，javax 当时只是扩展

API 包来说使用。然而随着时间的推移，javax 逐渐的扩展成为 Java API 的组

成部分。但是，将扩展从 javax 包移动到 java 包将是太麻烦了，最终会破坏

一堆现有的代码。因此，最终决定 javax 包将成为标准 API 的一部分。

所以，实际上 java 和 javax 没有区别。这都是一个名字。

接口和抽象类的区别是什么

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始

接口方法可以有默认实现），抽象类可以有非抽象的方法

2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定
3. 一个类可以实现多个接口，但最多只能实现一个抽象类
4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定
5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

成员变量与局部变量的区别有那些

1. 从语法形式上，看成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public,private,static 等修饰符所

修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员

变量和局部变量都能被 final 所修饰；

2. 从变量在内存中的存储方式来看，成员变量是对象的一部分，而对象存在于堆内存，局部变量存在于栈内存
3. 从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值，则会自动以类型的默认值而赋值（一种情况例外被 final 修饰的成员变量也必须显示地赋值）；而局部变量则不

会自动赋值。

创建一个对象用什么运算符?对象实体与对象引用有何不同?

new 运算符，new 创建对象实例（对象实例在堆内存中），对象引用指向对象实例（对象引用存放在栈内存中）。一个对象引用可以指向 0 个或 [1] 个对象

（一根绳子可以不系气球，也可以系一个气球）;一个对象可以有 n 个引用指向

它（可以用 n 条绳子系住一个气球）。

1 . 名字与类名相同;

什么是方法的返回值?返回值在类的方法里的作用是什么?

方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果！（前提是该方法可能产生结果）。返回值的作用:接收出结果，使得它可以用于其他的操作！

一个类的构造方法的作用是什么 若一个类没有声明构造方法,该程序能正确执行吗 ?为什么?

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。

构造方法有哪些特性

1. 名字与类名相同
2. 没有返回值，但不能用 void 声明构造函数;
3. 生成类的对象时自动执行，无需调用。

静态方法和实例方法有何不同

1. 在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对

象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用

静态方法可以无需创建对象。

2. 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制.

对象的相等与指向他们的引用相等，两者有什么不同?

对象的相等，比的是内存中存放的内容是否相等。而引用相等，比较的是他们指向的内存地址是否相等。

在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是?

帮助子类做初始化工作。

== 与 equals(重要)

== : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同

一个对象。(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)

equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

- 情况 1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个

对象时，等价于通过“==”比较这两个对象。

- 情况 2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来

两个对象的内容相等；若它们的内容相等，则返回 true (即，认为这两

个对象相等)。

举个例子：

```
public class test1 {
    public static void main(String[] args) {
    String a = new String("ab"); // a 为一个引用
        String b = new String("ab"); // b为另一个引用,对象的内容一样
        String aa = "ab"; // 放在常量池中
        String bb = "ab"; // 从常量池中查找
        if (aa == bb) // true
            System.out.println("aa==bb");
            if (a == b) // false，非同一对象
            System.out.println("a==b");
            if (a.equals(b)) // true
            System.out.println("aEQb");
            if (42 == 42.0) { // true
            System.out.println("true");
        }
    }
}
```

说明：

- String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是

比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。

- 当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存

在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

hashCode 与 equals（重要)

面试官可能会问你：“你重写过 hashcode 和 equals 么，为什么重写 equals

时必须重写 hashCode 方法？”

hashCode（）介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义

在 JDK 的 Object.java 中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

为什么要有 hashCode

我们以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode：

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断

对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如

果没有相符的 hashcode，HashSet 会假设对象没有重复出现。但是如果发现有

相同 hashcode 值的对象，这时会调用 equals（）方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。

如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head

first java》第二版）。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

hashCode（）与 equals（）的相关规定

1. 如果两个对象相等，则 hashcode 一定也是相同的
2. 两个对象相等,对两个对象分别调用 equals 方法都返回 true
3. 两个对象有相同的 hashcode 值，它们也不一定是相等的
4. **因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖**
5. hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

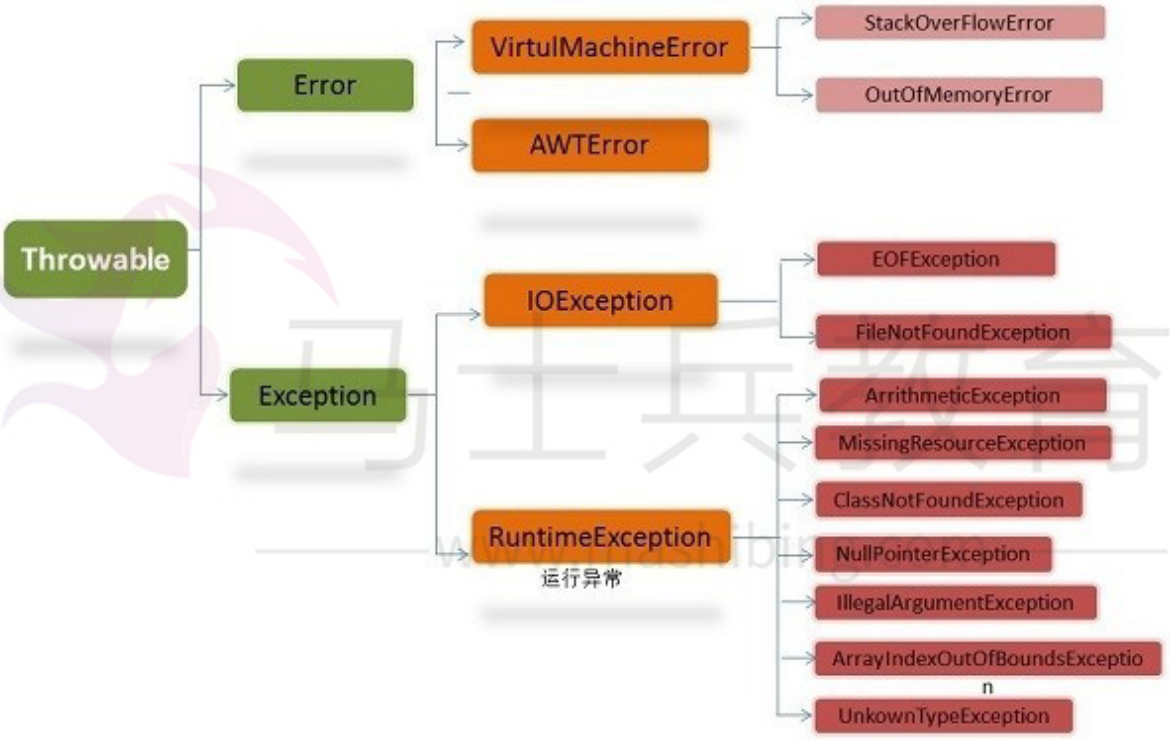
关于final关键字的一些总结

final 关键字主要用在三个地方：变量、方法、类。

1. 对于一个 final 变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
2. 当用 final 修饰一个类时，表明这个类不能被继承。final 类中的所有成员方法都会被隐式地指定为 final 方法。
3. 使用 final 方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的 Java 实现版本中，会将final 方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的 Java 版本已经不需要使用 final 方法进行这些优化了）。类中所有的 private 方法都隐式地指定为 final。

Java 中的异常处理

Java 异常类层次结构图



在 Java 中，所有的异常都有一个共同的祖先 java.lang 包中的 **Throwable类**。Throwable： 有两个重要的子类：**Exception（异常）** 和 **Error（错误）** ，二者都是 Java 异常处理的重要子类，各自都包含大量子类。

Error（错误）:是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java 虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java 虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 Error 的子类描述。

Exception（异常）:是程序本身可以处理的异常。Exception 类有一个重要的子类 **RuntimeException。**RuntimeException 异常由 Java 虚拟机抛出。

NullPointerException（要访问的变量没有引用任何对象时，抛出该异常）、**ArithmeticException**（算术运算异常，一个整数除以 0 时，抛出该异常）和 **ArrayIndexOutOfBoundsException** （下标越界异常）。

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

Throwable 类常用方法

- **public string getMessage()**:返回异常发生时的详细信息
- **public string toString()**:返回异常发生时的简要描述
- **public string getLocalizedMessage()**:返回异常对象的本地化信息。使用 Throwable 的子类覆盖这个方法，可以声称本地化信息。如果子类没有覆盖该方法，则该方法返回的信息与 getMessage（）返回的结果相同
- **public void printStackTrace()**:在控制台上打印 Throwable 对象封装的异常信息

异常处理总结

- try 块：用于捕获异常。其后可接零个或多个 catch 块，如果没有 catch

块，则必须跟一个 finally 块。

- catch 块：用于处理 try 捕获到的异常。
- finally 块：无论是否捕获或处理异常，finally 块里的语句都会被执行。

当在 try 块或 catch 块中遇到 return 语句时，finally 语句块将在方法返回之前被执行。**在以下 4 种特殊情况下，finally 块不会被执行：**

1. 在 finally 语句块中发生了异常。
2. 在前面的代码中用了 System.exit()退出程序。
3. 程序所在的线程死亡。
4. 关闭 CPU。

Java 序列化中如果有些字段不想进行序列化 怎么办

对于不想进行序列化的变量，使用 transient 关键字修饰。

transient 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被 transient 修饰的变量值不会被持久化和恢复。 transient 只能修饰变量，不能修饰类和方法。

获取用键盘输入常用的的两种方法

方法 1：通过 Scanner

```
Scanner input = new Scanner(System.in);
```

```
String s = input.nextLine(); input.close();
```

方法 2：通过 BufferedReader

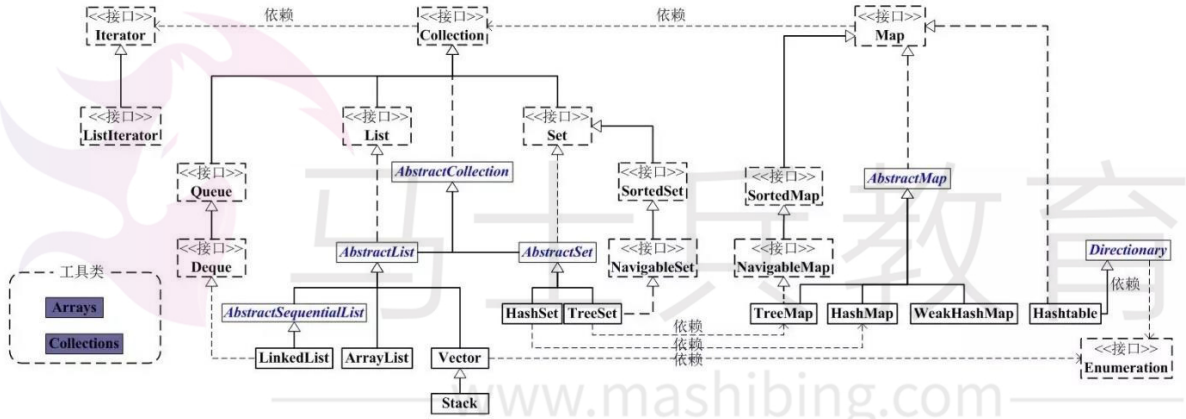
```
BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
```

```
String s = input.readLine();
```

接口继承关系和实现

集合类存放于 Java.util 包中，主要有 3 种：set(集）、list(列表包含 Queue）和 map(映射)。

1. Collection：Collection 是集合 List、Set、Queue 的最基本的接口。
2. Iterator：迭代器，可以通过迭代器遍历集合中的数据
3. Map：是映射表的基础接口

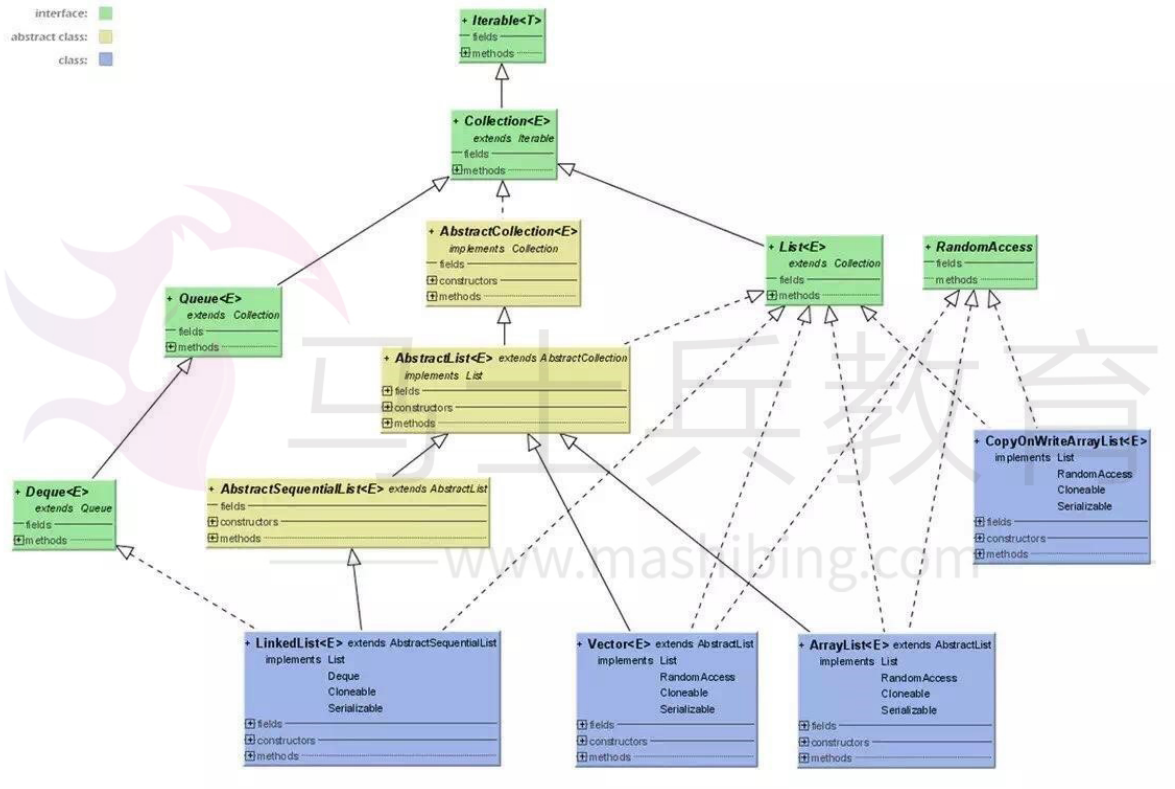




马士兵教育
www.mashibing.com

List

Java 的 List 是非常常用的数据类型。List 是有序的 Collection。Java List 一共三个实现类：分别是 ArrayList、Vector 和 LinkedList。



ArrayList（数组）

ArrayList 是最常用的 List 实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要将已经有数组的数据复制到新的存储空间中。当从 ArrayList 的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。

Vector（数组实现、线程同步）

Vector 与 ArrayList 一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一

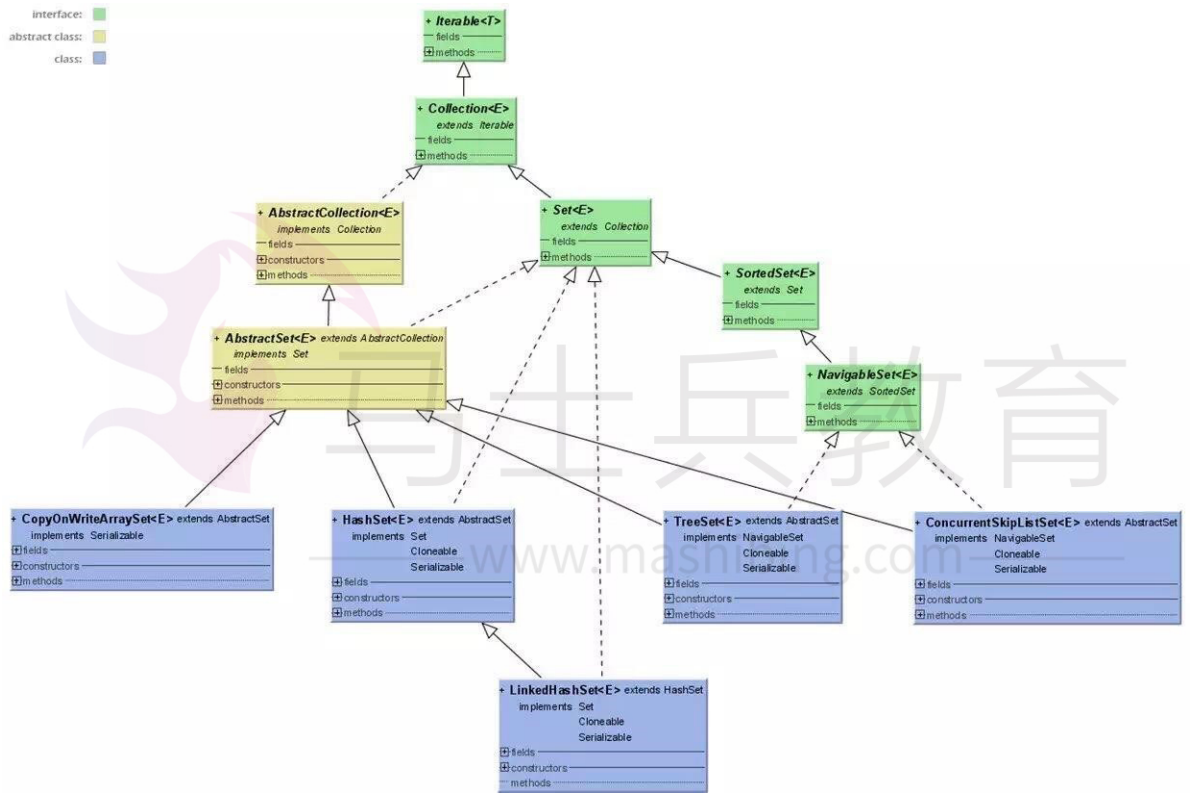
个线程能够写 Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问 ArrayList 慢。

LinkList（链表）

LinkedList 是用链表结构存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较慢。另外，他还提供了 List 接口中没有定义的方法，专门用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。

Set

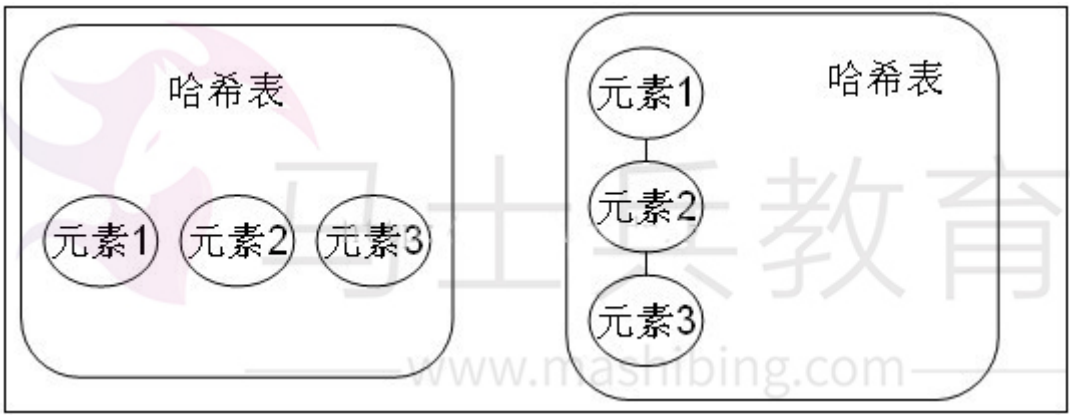
Set 注重独一无二的性质,该体系集合用于存储无序(存入和取出的顺序不一定相同)元素，值不能重复。对象的相等性本质是对象 hashCode 值（java 是依据对象的内存地址计算出的此序号）判断的，如果想要让两个不同的对象视为相等的，就必须覆盖 Object 的 hashCode 方法和 equals 方法。



HashSet（Hash 表）

哈希表边存放的是哈希值。HashSet 存储元素的顺序并不是按照存入时的顺序（和 List 显然不同） 而是按照哈希值来存的所以取数据也是按照哈希值取得。元素的哈希值是通过元素的hashcode 方法来获取的, HashSet 首先判断两个元素的哈希值，如果哈希值一样，接着会比较equals 方法 如果 equls 结果为 true ， HashSet 就视为同一个元素。如果 equals 为 false 就不是同一个元素。

哈希值相同 equals 为 false 的元素是怎么存储呢,就是在同样的哈希值下顺延（可以认为哈希值相同的元素放在一个哈希桶中）。也就是哈希一样的存一列。如图 1 表示 hashCode 值不相同的情况；图 2 表示 hashCode 值相同，但 equals 不相同的情况。



HashSet 通过 hashCode 值来确定元素在内存中的位置。一个 hashCode 位置上可以存放多个元素。

TreeSet（二叉树）

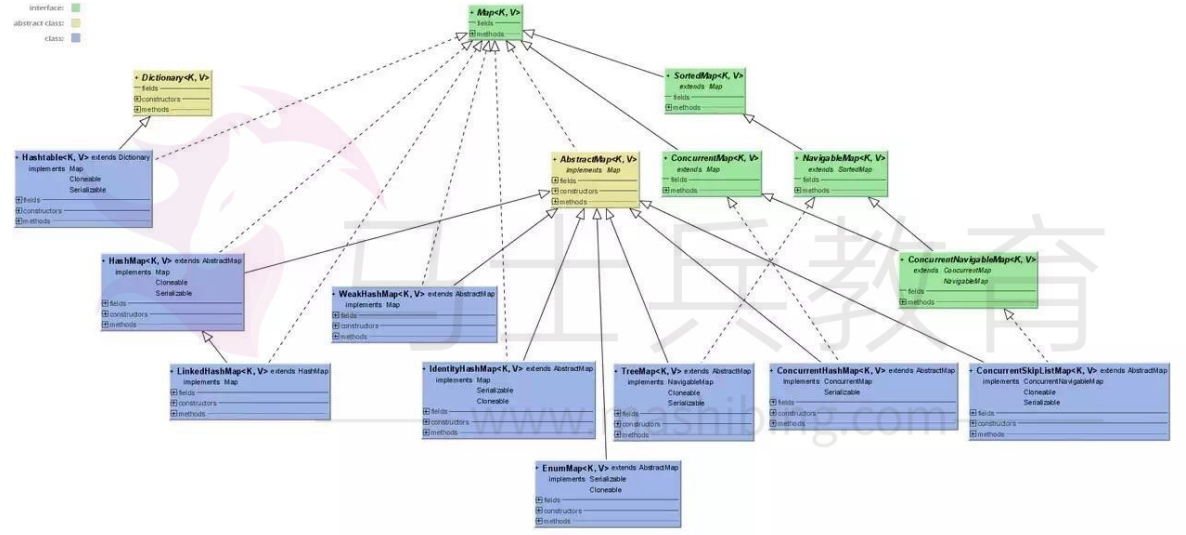
1. TreeSet()是使用二叉树的原理对新 add()的对象按照指定的顺序排序（升序、降序），每增加一个对象都会进行排序，将对象插入的二叉树指定的位置。
2. Integer 和 String 对象都可以进行默认的 TreeSet 排序，而自定义类的对象是不可以的，自己定义的类必须实现 Comparable 接口，并且覆写相应的 compareTo()函数，才可以正常使用。
3. 在覆写 compare()函数时，要返回相应的值才能使 TreeSet 按照一定的规则来排序
4. 比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数。

LinkHashSet（HashSet+LinkedHashMap）

对于 LinkedHashSet 而言，它继承与 HashSet、又基于 LinkedHashMap 来实现的。

LinkedHashSet 底层使用 LinkedHashMap 来保存所有元素，它继承与 HashSet，其所有的方法操作上又与 HashSet 相同，因此 LinkedHashSet 的实现上非常简单，只提供了四个构造方法，并通过传递一个标识参数，调用父类的构造器，底层构造一个 LinkedHashMap 来实现，在相关操作上与父类 HashSet 的操作相同，直接调用父类 HashSet 的方法即可。

Map



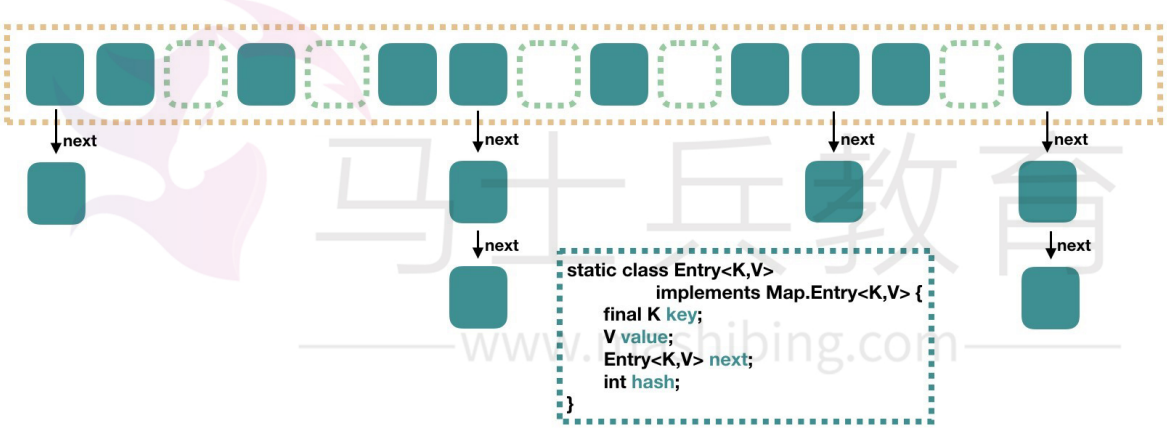
HashMap（数组+链表+红黑树）

HashMap 根据键的 hashCode 值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。 HashMap最多只允许一条记录的键为null，允许多条记录的值为null。HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。我们用下面这张图来介绍

HashMap 的结构。

JAVA7 实现

Java7 HashMap 结构



大方向上，HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。上图中，每个绿色

的实体是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

1. capacity：当前数组容量，始终保持 2^n，可以扩容，扩容后数组大小为当前的 2 倍。
2. loadFactor：负载因子，默认为 0.75。

3. threshold：扩容的阈值，等于 capacity * loadFactor

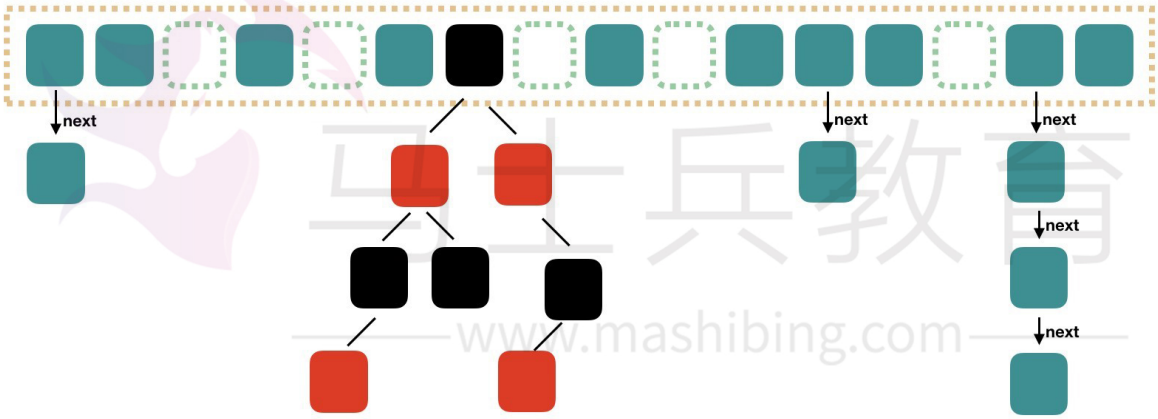
JAVA8实现

Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树 组成。

根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决

于链表的长度，为 O(n)。为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。

Java8 HashMap 结构



马士兵教育
www.mashibing.com

ConcurrentHashMap

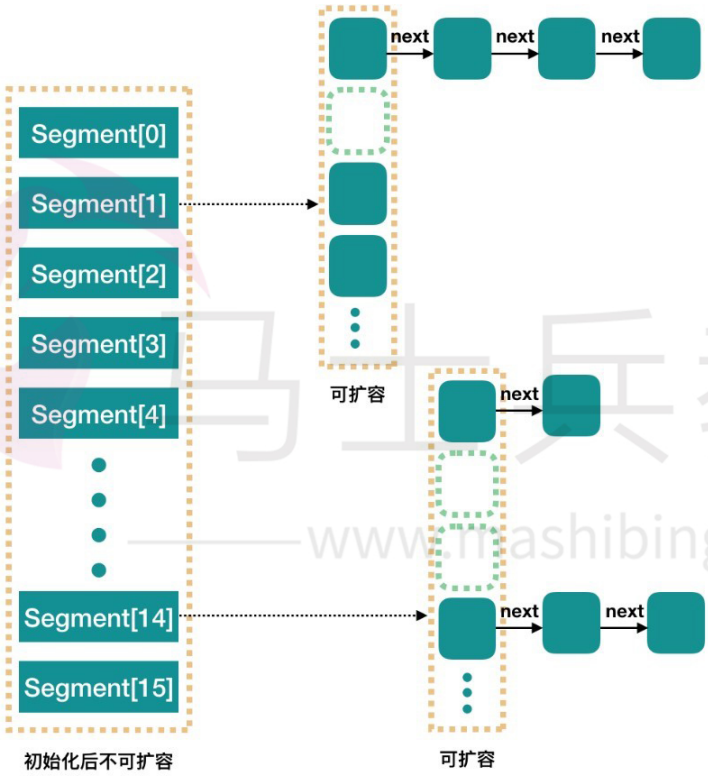
Segment 段

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表"部分"或"一段"的意思，所以很多地方都会将其描述为分段锁。注意，行文中，我很多地方用了"槽"来代表一个 segment。

线程安全（**Segment** 继承 **ReentrantLock** 加锁）

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

Java7 ConcurrentHashMap 结构



马士兵教育
www.mashibing.com

并行度 concurrencyLevel：并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

Java8 实现 （引入了红黑树）

Java8 对 ConcurrentHashMap 进行了比较大的改动,Java8 也引入了红黑树。

Java8实现(../../../../../0马士兵/新建文件夹/BAT面试突击资料(1)/整理/BAT面试突击资料/06-JAVA面试核心知识点整理(时间较多的同学全面复习).assets/Java8实现(引入了红黑树).jpg)

HashTable（线程安全）

Hashtable 是遗留类，很多映射的常用功能与 HashMap 类似，不同的是它承自 Dictionary 类，并且是线程安全的，任一时间只有一个线程能写 Hashtable，并发性不如 ConcurrentHashMap，因为 ConcurrentHashMap 引入了分段锁。Hashtable 不建议在新代码中使用，不需要线程安全的场合可以用 HashMap 替换，需要线程安全的场合可以用 ConcurrentHashMap 替换。

3.4.4. TreeMap（可排序）

TreeMap 实现 SortedMap 接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 TreeMap 时，得到的记录是排过序的。如果使用排序的映射，建议使用 TreeMap。

在使用 TreeMap 时，key 必须实现 Comparable 接口或者在构造 TreeMap 传入自定义的 Comparator，否则会在运行时抛出 java.lang.ClassCastException 类型的异常。

LinkHashMap（记录插入顺序）

LinkedHashMap 是 HashMap 的一个子类，保存了记录的插入顺序，在用 Iterator 遍历 LinkedHashMap 时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

TreeMap 的实现就是红黑树数据结构，也就说是一棵自平衡的排序二叉树，这样就可以保证当需要快速检索指定节点。

TreeSet 和 TreeMap 的关系

为了让大家了解 TreeMap 和 TreeSet 之间的关系，下面先看 TreeSet 类的部分源代码：

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
 {
    // 使用 NavigableMap 的 key 来保存 Set 集合的元素
    private transient NavigableMap<E,Object> m;
    // 使用一个 PRESENT 作为 Map 集合的所有 value。
    private static final Object PRESENT = new Object();
    // 包访问权限的构造器，以指定的 NavigableMap 对象创建 Set 集合
    TreeSet(NavigableMap<E,Object> m)
    {
        this.m = m;
    }
    public TreeSet()                                        // ①
    {
        // 以自然排序方式创建一个新的 TreeMap，
        // 根据该 TreeSet 创建一个 TreeSet，
        // 使用该 TreeMap 的 key 来保存 Set 集合的元素
        this(new TreeMap<E,Object>());
    }
    public TreeSet(Comparator<? super E> comparator)     // ②
    {
        // 以定制排序方式创建一个新的 TreeMap，
        // 根据该 TreeSet 创建一个 TreeSet，
        // 使用该 TreeMap 的 key 来保存 Set 集合的元素
        this(new TreeMap<E,Object>(comparator));
    }
    public TreeSet(Collection<? extends E> c)
    {
        // 调用①号构造器创建一个 TreeSet，底层以 TreeMap 保存集合元素
        this();
        // 向 TreeSet 中添加 Collection 集合 c 里的所有元素
        addAll(c);
    }
    public TreeSet(SortedSet<E> s)
    {
        // 调用②号构造器创建一个 TreeSet，底层以 TreeMap 保存集合元素
        this(s.comparator());
        // 向 TreeSet 中添加 SortedSet 集合 s 里的所有元素
        addAll(s);
    }
    //TreeSet 的其他方法都只是直接调用 TreeMap 的方法来提供实现
    ...
    public boolean addAll(Collection<? extends E> c)
    {
        if (m.size() == 0 && c.size() > 0 &&
            c instanceof SortedSet &&
            m instanceof TreeMap)
        {
```

```
            // 把 c 集合强制转换为 SortedSet 集合
            SortedSet<? extends E> set = (SortedSet<? extends E>) c;
            // 把 m 集合强制转换为 TreeMap 集合
            TreeMap<E,Object> map = (TreeMap<E, Object>) m;
            Comparator<? super E> cc = (Comparator<? super E>) set.comparator();
            Comparator<? super E> mc = map.comparator();
            // 如果 cc 和 mc 两个 Comparator 相等
            if (cc == mc || (cc != null && cc.equals(mc)))
            {
                // 把 Collection 中所有元素添加成 TreeMap 集合的 key
                map.addAllForTreeSet(set, PRESENT);
                return true;
            }
        }
        // 直接调用父类的 addAll() 方法来实现
        return super.addAll(c);
    }
    ...
}
```

显示更多

从上面代码可以看出，TreeSet 的 ① 号、② 号构造器的都是新建一个 TreeMap 作为实际存储 Set 元素的容器，而另外 2 个构造器则分别依赖于 ① 号和 ② 号构造器，由此可见，TreeSet 底层实际使用的存储容器就是 TreeMap。

与 HashSet 完全类似的是，TreeSet 里绝大部分方法都是直接调用 TreeMap 的方法来实现的，这一点读者可以自行参阅 TreeSet 的源代码，此处就不再给出了。

对于 TreeMap 而言，它采用一种被称为"红黑树"的排序二叉树来保存 Map 中每个 Entry —— 每个 Entry 都被当成"红黑树"的一个节点对待。例如对于如下程序而言：

```
public class TreeMapTest
{
    public static void main(String[] args)
    {
        TreeMap<String , Double> map =
            new TreeMap<String , Double>();
        map.put("ccc" , 89.0);
        map.put("aaa" , 80.0);
        map.put("zzz" , 80.0);
        map.put("bbb" , 89.0);
        System.out.println(map);
    }
}
```

显示更多

当程序执行 map.put("ccc" , 89.0); 时，系统将直接把 "ccc"-89.0 这个 Entry 放入 Map 中，这个 Entry 就是该"红黑树"的根节点。接着程序执行 map.put("aaa" , 80.0); 时，程序会将 "aaa"-80.0 作为新节点添加到已有的红黑树中。

以后每向 TreeMap 中放入一个 key-value 对，系统都需要将该 Entry 当成一个新节点，添加成已有红黑树中，通过这种方式就可保证 TreeMap 中所有 key 总是由小到大地排列。例如我们输出上面程序，将看到如下结果（所有 key 由小到大地排列）：

```
{aaa=80.0, bbb=89.0, ccc=89.0, zzz=80.0}
```

显示更多

TreeMap 的添加节点

对于 TreeMap 而言，由于它底层采用一棵”红黑树”来保存集合中的 Entry，这意味这 TreeMap 添加元素、取出元素的性能都比 HashMap 低：当 TreeMap 添加元素时，需要通过循环找到新增 Entry 的插入位置，因此比较耗性能；当从 TreeMap 中取出元素时，需要通过循环才能找到合适的 Entry，也比较耗性能。但 TreeMap、TreeSet 比 HashMap、HashSet 的优势在于：TreeMap 中的所有 Entry 总是按 key 根据指定排序规则保持有序状态，TreeSet 中所有元素总是根据指定排序规则保持有序状态。

> **红黑树**
>
> 红黑树是一种自平衡排序二叉树，树中每个节点的值，都大于或等于在它的左子树中的所有节点的值，并且小于或等于在它的右子树中的所有节点的值，这确保红黑树运行时可以快速地在树中查找和定位的所需节点。

为了理解 TreeMap 的底层实现，必须先介绍排序二叉树和红黑树这两种数据结构。其中红黑树又是一种特殊的排序二叉树。

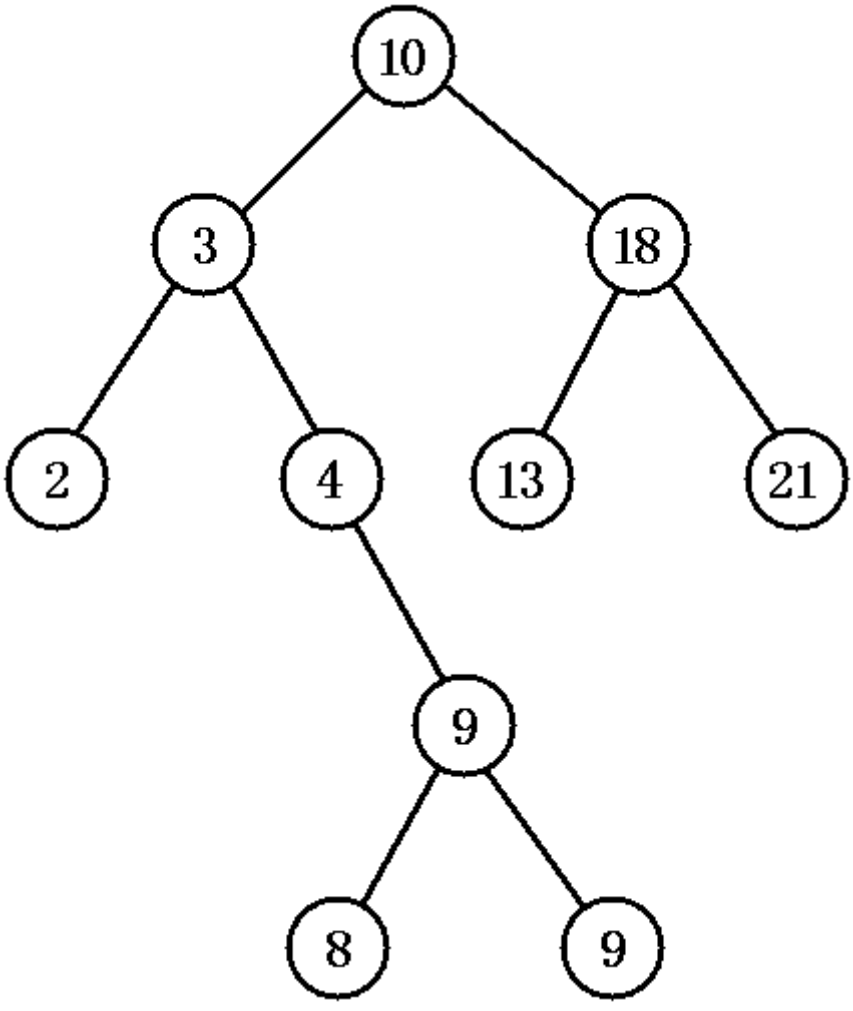
排序二叉树是一种特殊结构的二叉树，可以非常方便地对树中所有节点进行排序和检索。

排序二叉树要么是一棵空二叉树，要么是具有下列性质的二叉树：

- 若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
- 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
- 它的左、右子树也分别为排序二叉树。

图 1 显示了一棵排序二叉树：

图 1. 排序二叉树



对排序二叉树，若按中序遍历就可以得到由小到大的有序序列。如图 1 所示二叉树，中序遍历得：

```
{2，3，4，8，9，9，10，13，15，18}
```

创建排序二叉树的步骤，也就是不断地向排序二叉树添加节点的过程，向排序二叉树添加节点的步骤如下：

1. 以根节点当前节点开始搜索。
2. 拿新节点的值和当前节点的值比较。
3. 如果新节点的值更大，则以当前节点的右子节点作为新的当前节点；如果新节点的值更小，则以当前节点的左子节点作为新的当前节点。
4. 重复 2、3 两个步骤，直到搜索到合适的叶子节点为止。
5. 将新节点添加为第 4 步找到的叶子节点的子节点；如果新节点更大，则添加为右子节点；否则添加为左子节点。

掌握上面理论之后，下面我们来分析 TreeMap 添加节点（TreeMap 中使用 Entry 内部类代表节点）的实现，TreeMap 集合的 put(K key, V value) 方法实现了将 Entry 放入排序二叉树中，下面是该方法的源代码：

```
public V put(K key, V value)
 {
    // 先以 t 保存链表的 root 节点
    Entry<K,V> t = root;
    // 如果 t==null，表明是一个空链表，即该 TreeMap 里没有任何 Entry
    if (t == null)
    {
        // 将新的 key-value 创建一个 Entry，并将该 Entry 作为 root
        root = new Entry<K,V>(key, value, null);
        // 设置该 Map 集合的 size 为 1，代表包含一个 Entry
        size = 1;
        // 记录修改次数为 1
        modCount++;
        return null;
    }
    int cmp;
    Entry<K,V> parent;
    Comparator<? super K> cpr = comparator;
    // 如果比较器 cpr 不为 null，即表明采用定制排序
    if (cpr != null)
    {
        do {
            // 使用 parent 上次循环后的 t 所引用的 Entry
            parent = t;
            // 拿新插入 key 和 t 的 key 进行比较
            cmp = cpr.compare(key, t.key);
            // 如果新插入的 key 小于 t 的 key，t 等于 t 的左边节点
            if (cmp < 0)
                t = t.left;
            // 如果新插入的 key 大于 t 的 key，t 等于 t 的右边节点
            else if (cmp > 0)
                t = t.right;
            // 如果两个 key 相等，新的 value 覆盖原有的 value，
            // 并返回原有的 value
            else
                return t.setValue(value);
        } while (t != null);
```

```
    }
    else
    {
        if (key == null)
            throw new NullPointerException();
        Comparable<? super K> k = (Comparable<? super K>) key;
        do {
            // 使用 parent 上次循环后的 t 所引用的 Entry
            parent = t;
            // 拿新插入 key 和 t 的 key 进行比较
            cmp = k.compareTo(t.key);
            // 如果新插入的 key 小于 t 的 key，t 等于 t 的左边节点
            if (cmp < 0)
                t = t.left;
            // 如果新插入的 key 大于 t 的 key，t 等于 t 的右边节点
            else if (cmp > 0)
                t = t.right;
            // 如果两个 key 相等，新的 value 覆盖原有的 value，
            // 并返回原有的 value
            else
                return t.setValue(value);
        } while (t != null);
    }
    // 将新插入的节点作为 parent 节点的子节点
    Entry<K,V> e = new Entry<K,V>(key, value, parent);
    // 如果新插入 key 小于 parent 的 key，则 e 作为 parent 的左子节点
    if (cmp < 0)
        parent.left = e;
    // 如果新插入 key 小于 parent 的 key，则 e 作为 parent 的右子节点
    else
        parent.right = e;
    // 修复红黑树
    fixAfterInsertion(e);                                    // ①
    size++;
    modCount++;
    return null;
}
```

显示更多

上面程序中粗体字代码就是实现"排序二叉树"的关键算法，每当程序希望添加新节点时：系统总是从树的根节点开始比较 —— 即将根节点当成当前节点，如果新增节点大于当前节点、并且当前节点的右子节点存在，则以右子节点作为当前节点；如果新增节点小于当前节点、并且当前节点的左子节点存在，则以左子节点作为当前节点；如果新增节点等于当前节点，则用新增节点覆盖当前节点，并结束循环 —— 直到找到某个节点的左、右子节点不存在，将新节点添加该节点的子节点 —— 如果新节点比该节点大，则添加为右子节点；如果新节点比该节点小，则添加为左子节点。

TreeMap 的删除节点

当程序从排序二叉树中删除一个节点之后，为了让它依然保持为排序二叉树，程序必须对该排序二叉树进行维护。维护可分为如下几种情况：

（1）被删除的节点是叶子节点，则只需将它从其父节点中删除即可。

（2）被删除节点 p 只有左子树，将 p 的左子树 pL 添加成 p 的父节点的左子树即可；被删除节点 p 只有右子树，将 p 的右子树 pR 添加成 p 的父节点的右子树即可。

（3）若被删除节点 p 的左、右子树均非空，有两种做法：

- 将 pL 设为 p 的父节点 q 的左或右子节点（取决于 p 是其父节点 q 的左、右子节点），将 pR 设为 p 节点的中序前趋节点 s 的右子节点（s 是 pL 最右下的节点，也就是 pL 子树中最大的节点）。
- 以 p 节点的中序前趋或后继替代 p 所指节点，然后再从原排序二叉树中删去中序前趋或后继节点即可。（也就是用大于 p 的最小节点或小于 p 的最大节点代替 p 节点即可）。

图 2 显示了被删除节点只有左子树的示意图：

图 2. 被删除节点只有左子树

图 3 显示了被删除节点只有右子树的示意图：

图 3. 被删除节点只有右子树

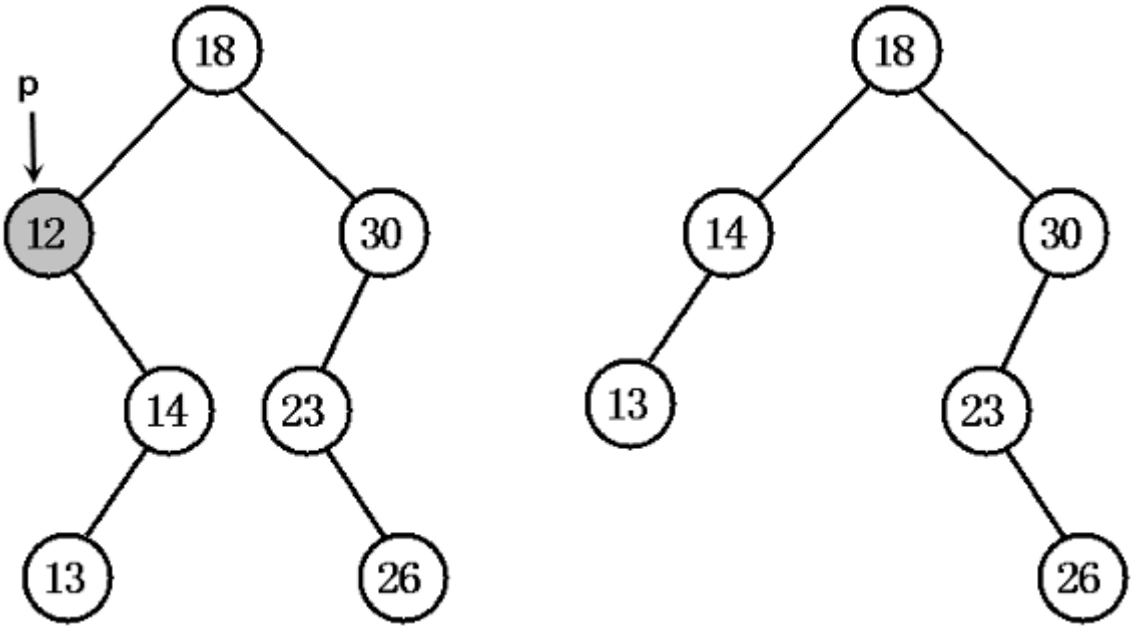


图 4 显示了被删除节点既有左子节点，又有右子节点的情形，此时我们采用到是第一种方式进行维护：

图 4. 被删除节点既有左子树，又有右子树

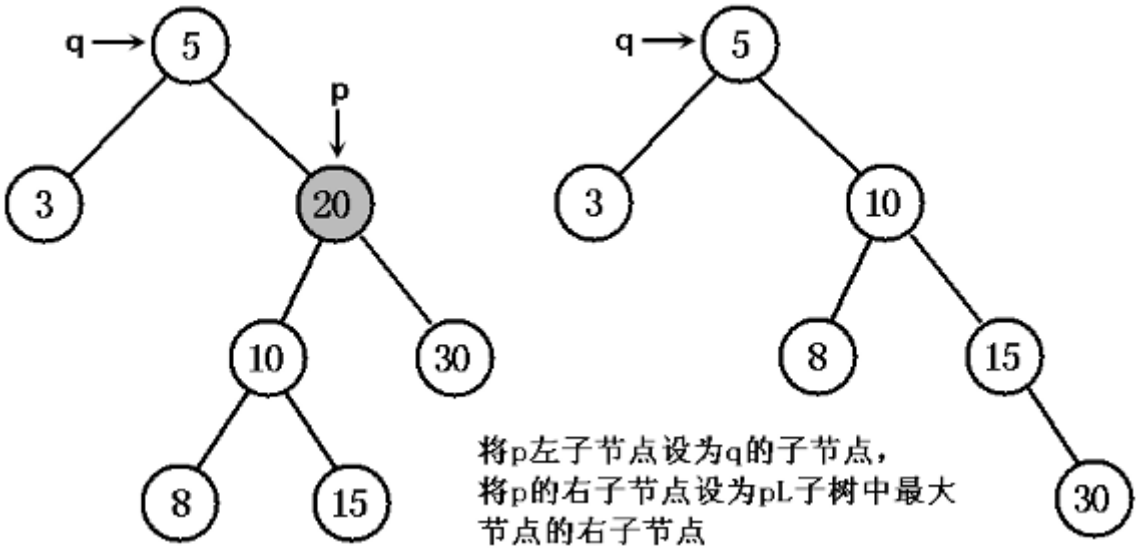
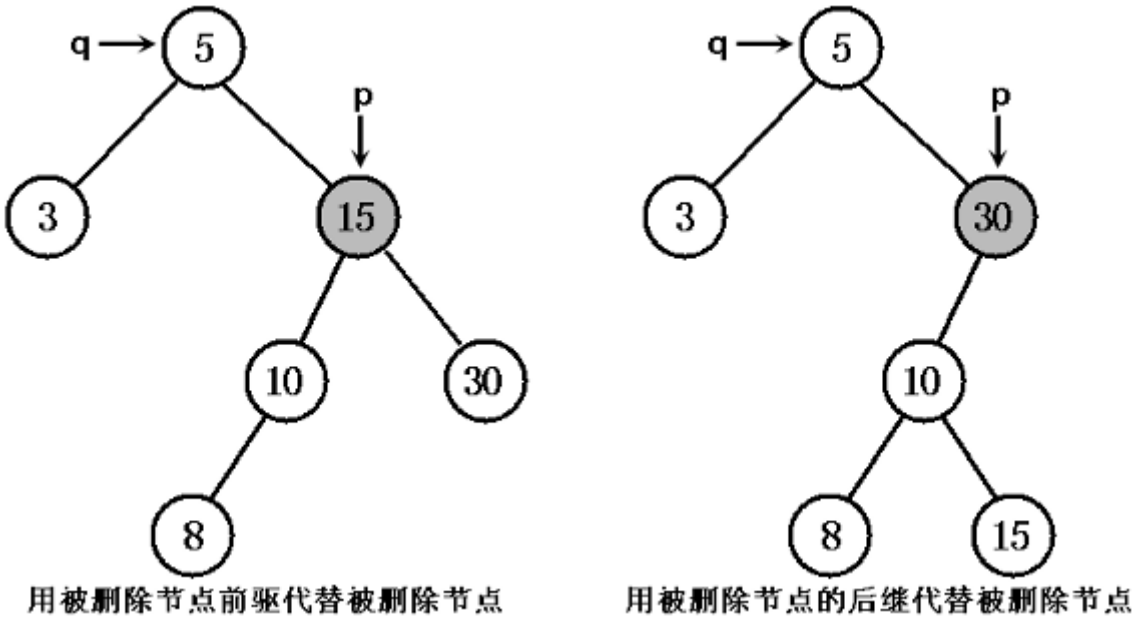


图 5 显示了被删除节点既有左子树，又有右子树的情形，此时我们采用到是第二种方式进行维护：

图 5. 被删除节点既有左子树，又有右子树



TreeMap 删除节点采用图 5 所示右边的情形进行维护——也就是用被删除节点的右子树中最小节点与被删节点交换的方式进行维护。

TreeMap 删除节点的方法由如下方法实现：

```
private void deleteEntry(Entry<K,V> p)
 {
    modCount++;
    size--;
    // 如果被删除节点的左子树、右子树都不为空
    if (p.left != null && p.right != null)
    {
        // 用 p 节点的中序后继节点代替 p 节点
        Entry<K,V> s = successor (p);
        p.key = s.key;
        p.value = s.value;
        p = s;
    }
    // 如果 p 节点的左节点存在，replacement 代表左节点；否则代表右节点。
    Entry<K,V> replacement = (p.left != null ? p.left : p.right);
    if (replacement != null)
    {
        replacement.parent = p.parent;
        // 如果 p 没有父节点，则 replacemment 变成父节点
        if (p.parent == null)
            root = replacement;
        // 如果 p 节点是其父节点的左子节点
        else if (p == p.parent.left)
            p.parent.left  = replacement;
        // 如果 p 节点是其父节点的右子节点
        else
            p.parent.right = replacement;
        p.left = p.right = p.parent = null;
        // 修复红黑树
        if (p.color == BLACK)
            fixAfterDeletion(replacement);       // ①
    }
    // 如果 p 节点没有父节点
```

```
    else if (p.parent == null)
    {
        root = null;
    }
    else
    {
        if (p.color == BLACK)
            // 修复红黑树
            fixAfterDeletion(p);                    // ②
        if (p.parent != null)
        {
            // 如果 p 是其父节点的左子节点
            if (p == p.parent.left)
                p.parent.left = null;
            // 如果 p 是其父节点的右子节点
            else if (p == p.parent.right)
                p.parent.right = null;
            p.parent = null;
        }
    }
}
```

显示更多

红黑树

排序二叉树虽然可以快速检索，但在最坏的情况下：如果插入的节点集本身就是有序的，要么是由小到大排列，要么是由大到小排列，那么最后得到的排序二叉树将变成链表：所有节点只有左节点（如果插入节点集本身是大到小排列）；或所有节点只有右节点（如果插入节点集本身是小到大排列）。在这种情况下，排序二叉树就变成了普通链表，其检索效率就会很差。

为了改变排序二叉树存在的不足，Rudolf Bayer 与 1972 年发明了另一种改进后的排序二叉树：红黑树，他将这种排序二叉树称为"对称二叉 B 树"，而红黑树这个名字则由 Leo J. Guibas 和 Robert Sedgewick 于 1978 年首次提出。

红黑树是一个更高效的检索二叉树，因此常常用来实现关联数组。典型地，JDK 提供的集合类 TreeMap 本身就是一个红黑树的实现。

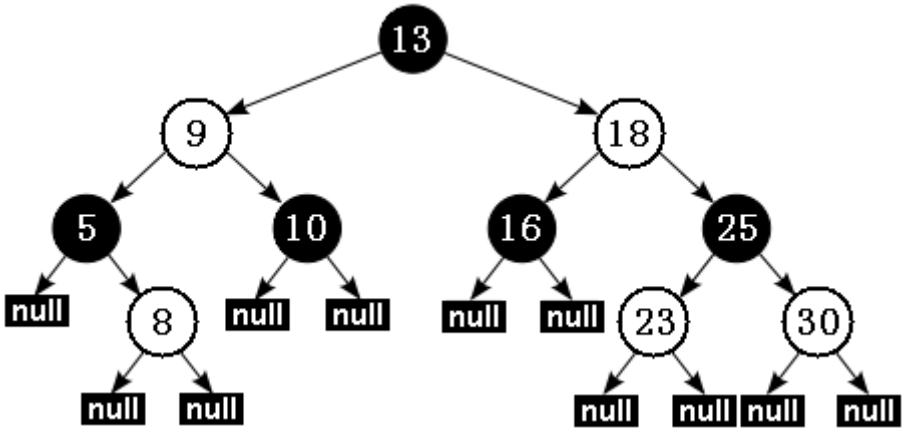
红黑树在原有的排序二叉树增加了如下几个要求：

- 性质 1：每个节点要么是红色，要么是黑色。
- 性质 2：根节点永远是黑色的。
- 性质 3：所有的叶节点都是空节点（即 null），并且是黑色的。
- 性质 4：每个红色节点的两个子节点都是黑色。（从每个叶子到根的路径上不会有两个连续的红色节点）
- 性质 5：从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点。

> **Java 实现的红黑树**
>
> 上面的性质 3 中指定红黑树的每个叶子节点都是空节点，而且并叶子节点都是黑色。但 Java 实现的红黑树将使用 null 来代表空节点，因此遍历红黑树时将看不到黑色的叶子节点，反而看到每个叶子节点都是红色的。

Java 中实现的红黑树可能有如图 6 所示结构：

图 6. Java 红黑树的示意



备注：本文中所有关于红黑树中的示意图采用白色代表红色。黑色节点还是采用了黑色表示。

根据性质 5：红黑树从根节点到每个叶子节点的路径都包含相同数量的黑色节点，因此从根节点到叶子节点的路径中包含的黑色节点数被称为树的"黑色高度（black-height）"。

性质 4 则保证了从根节点到叶子节点的最长路径的长度不会超过任何其他路径的两倍。假如有一棵黑色高度为 3 的红黑树：从根节点到叶节点的最短路径长度是 2，该路径上全是黑色节点（黑节点 – 黑节点 – 黑节点）。最长路径也只可能为 4，在每个黑色节点之间插入一个红色节点（黑节点 – 红节点 – 黑节点 – 红节点 – 黑节点），性质 4 保证绝不可能插入更多的红色节点。由此可见，红黑树中最长路径就是一条红黑交替的路径。

由此我们可以得出结论：对于给定的黑色高度为 N 的红黑树，从根到叶子节点的最短路径长度为 N-1，最长路径长度为 2 * (N-1)。

提示：排序二叉树的深度直接影响了检索的性能，正如前面指出，当插入节点本身就是由小到大排列时，排序二叉树将变成一个链表，这种排序二叉树的检索性能最低：N 个节点的二叉树深度就是 N-1。

> **红黑树和平衡二叉树**
>
> 红黑树并不是真正的平衡二叉树，但在实际应用中，红黑树的统计性能要高于平衡二叉树，但极端性能略差。

红黑树通过上面这种限制来保证它大致是平衡的——因为红黑树的高度不会无限增高，这样保证红黑树在最坏情况下都是高效的，不会出现普通排序二叉树的情况。

由于红黑树只是一个特殊的排序二叉树，因此对红黑树上的只读操作与普通排序二叉树上的只读操作完全相同，只是红黑树保持了大致平衡，因此检索性能比排序二叉树要好很多。

但在红黑树上进行插入操作和删除操作会导致树不再符合红黑树的特征，因此插入操作和删除操作都需要进行一定的维护，以保证插入节点、删除节点后的树依然是红黑树。

添加节点后的修复

上面 `put(K key, V value)` 方法中① 号代码处使用 `fixAfterInsertion(e)` 方法来修复红黑树——因此每次插入节点后必须进行简单修复，使该排序二叉树满足红黑树的要求。

> **插入后的修复**
>
> 在插入操作中，红黑树的性质 1 和性质 3 两个永远不会发生改变，因此无需考虑红黑树的这两个特性。

插入操作按如下步骤进行：

1. 以排序二叉树的方法插入新节点，并将它设为红色。
2. 进行颜色调换和树旋转。

这种颜色调用和树旋转就比较复杂了，下面将分情况进行介绍。在介绍中，我们把新插入的节点定义为 N 节点，N 节点的父节点定义为 P 节点，P 节点的兄弟节点定义为 U 节点，P 节点父节点定义为 G 节点。

下面分成不同情形来分析插入操作

情形 1：新节点 N 是树的根节点，没有父节点

在这种情形下，直接将它设置为黑色以满足性质 2。

情形 2：新节点的父节点 P 是黑色

在这种情况下，新插入的节点是红色的，因此依然满足性质 4。而且因为新节点 N 有两个黑色叶子节点；但是由于新节点 N 是红色，通过它的每个子节点的路径依然保持相同的黑色节点数，因此依然满足性质 5。

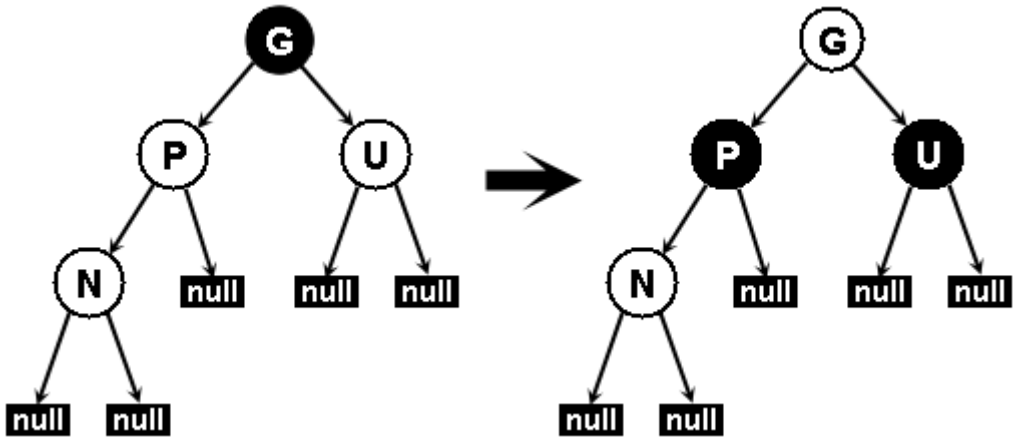
情形 3：如果父节点 P 和父节点的兄弟节点 U 都是红色

在这种情况下，程序应该将 P 节点、U 节点都设置为黑色，并将 P 节点的父节点设为红色（用来保持性质 5）。现在新节点 N 有了一个黑色的父节点 P。由于从 P 节点、U 节点到根节点的任何路径都必须通过 G 节点，在这些路径上的黑节点数目没有改变（原来有叶子和 G 节点两个黑色节点，现在有叶子和 P 两个黑色节点）。

经过上面处理后，红色的 G 节点的父节点也有可能是红色的，这就违反了性质 4，因此还需要对 G 节点递归地进行整个过程（把 G 当成是新插入的节点进行处理即可）。

图 7 显示了这种处理过程：

图 7. 插入节点后进行颜色调换



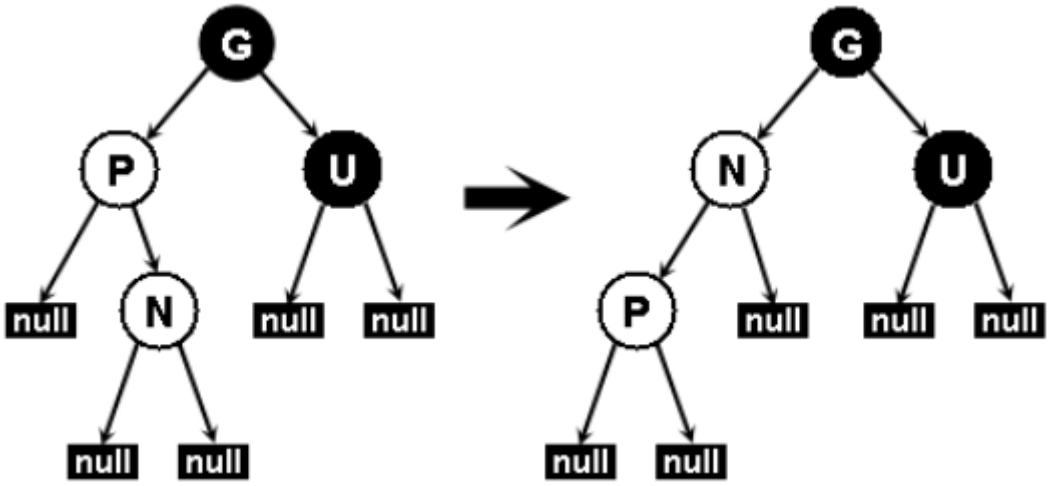
备注：虽然图 11.28 绘制的是新节点 N 作为父节点 P 左子节点的情形，其实新节点 N 作为父节点 P 右子节点的情况与图 11.28 完全相同。

情形 4：父节点 P 是红色、而其兄弟节点 U 是黑色或缺少；且新节点 N 是父节点 P 的右子节点，而父节点 P 又是其父节点 G 的左子节点。

在这种情形下，我们进行一次左旋转对新节点和其父节点进行，接着按情形 5 处理以前的父节点 P（也就是把 P 当成新插入的节点即可）。这导致某些路径通过它们以前不通过的新节点 N 或父节点 P 的其中之一，但是这两个节点都是红色的，因此不会影响性质 5。

图 8 显示了对情形 4 的处理：

图 8. 插入节点后的树旋转



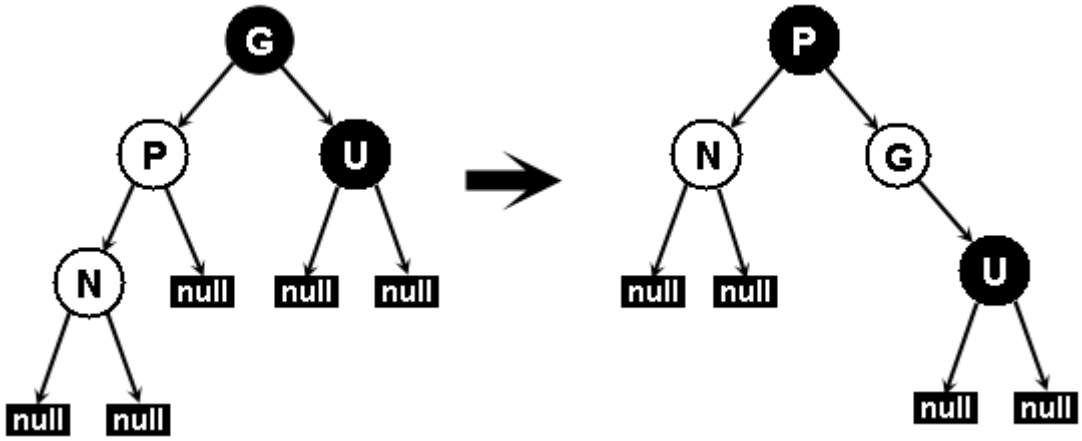
备注：图 11.29 中 P 节点是 G 节点的左子节点，如果 P 节点是其父节点 G 节点的右子节点，那么上 **面的处理情况应该左、右对调一下。**

情形 5：父节点 P 是红色、而其兄弟节点 U 是黑色或缺少；且新节点 N 是其父节点的左子节点，而父节点 P 又是其父节点 G 的左子节点。

在这种情形下，需要对节点 G 的一次右旋转，在旋转产生的树中，以前的父节点 P 现在是新节点 N 和节点 G 的父节点。由于以前的节点 G 是黑色，否则父节点 P 就不可能是红色，我们切换以前的父节点 P 和节点 G 的颜色，使之满足性质 4，性质 5 也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过节点 G，现在它们都通过以前的父节点 P。在各自的情形下，这都是三个节点中唯一的黑色节点。

图 9 显示了情形 5 的处理过程：

图 9. 插入节点后的颜色调整、树旋转



备注：图 11.30 中 P 节点是 G 节点的左子节点，如果 P 节点是其父节点 G 节点的右子节点，那么上面的处理情况应该左、右对调一下。

TreeMap 为插入节点后的修复操作由 fixAfterInsertion(Entry<k,v> x) 方法提供，该方法的源代码如下：

```
// 插入节点后修复红黑树
 private void fixAfterInsertion(Entry<K,V> x)
 {
    x.color = RED;
    // 直到 x 节点的父节点不是根，且 x 的父节点不是红色
    while (x != null && x != root
        && x.parent.color == RED)
```

```
    {
        // 如果 x 的父节点是其父节点的左子节点
        if (parentOf(x) == leftOf(parentOf(parentOf(x))))
        {
            // 获取 x 的父节点的兄弟节点
            Entry<K,V> y = rightOf(parentOf(parentOf(x)));
            // 如果 x 的父节点的兄弟节点是红色
            if (colorOf(y) == RED)
            {
                // 将 x 的父节点设为黑色
                setColor(parentOf(x), BLACK);
                // 将 x 的父节点的兄弟节点设为黑色
                setColor(y, BLACK);
                // 将 x 的父节点的父节点设为红色
                setColor(parentOf(parentOf(x)), RED);
                x = parentOf(parentOf(x));
            }
            // 如果 x 的父节点的兄弟节点是黑色
            else
            {
                // 如果 x 是其父节点的右子节点
                if (x == rightOf(parentOf(x)))
                {
                    // 将 x 的父节点设为 x
                    x = parentOf(x);
                    rotateLeft(x);
                }
                // 把 x 的父节点设为黑色
                setColor(parentOf(x), BLACK);
                // 把 x 的父节点的父节点设为红色
                setColor(parentOf(parentOf(x)), RED);
                rotateRight(parentOf(parentOf(x)));
            }
        }
        // 如果 x 的父节点是其父节点的右子节点
        else
        {
            // 获取 x 的父节点的兄弟节点
            Entry<K,V> y = leftOf(parentOf(parentOf(x)));
            // 如果 x 的父节点的兄弟节点是红色
            if (colorOf(y) == RED)
            {
                // 将 x 的父节点设为黑色。
                setColor(parentOf(x), BLACK);
                // 将 x 的父节点的兄弟节点设为黑色
                setColor(y, BLACK);
                // 将 x 的父节点的父节点设为红色
                setColor(parentOf(parentOf(x)), RED);
                // 将 x 设为 x 的父节点的节点
                x = parentOf(parentOf(x));
            }
            // 如果 x 的父节点的兄弟节点是黑色
            else
            {
                // 如果 x 是其父节点的左子节点
                if (x == leftOf(parentOf(x)))
                {
                    // 将 x 的父节点设为 x
```

```
                    x = parentOf(x);
                    rotateRight(x);
                }
                // 把 x 的父节点设为黑色
                setColor(parentOf(x), BLACK);
                // 把 x 的父节点的父节点设为红色
                setColor(parentOf(parentOf(x)), RED);
                rotateLeft(parentOf(parentOf(x)));
            }
        }
    }
    // 将根节点设为黑色
    root.color = BLACK;
 }
```

显示较少

删除节点后的修复

与添加节点之后的修复类似的是，TreeMap 删除节点之后也需要进行类似的修复操作，通过这种修复来保证该排序二叉树依然满足红黑树特征。大家可以参考插入节点之后的修复来分析删除之后的修复。TreeMap 在删除之后的修复操作由 fixAfterDeletion(Entry<k,v> x) 方法提供，该方法源代码如下：

```
// 删除节点后修复红黑树
 private void fixAfterDeletion(Entry<K,V> x)
 {
    // 直到 x 不是根节点，且 x 的颜色是黑色
    while (x != root && colorOf(x) == BLACK)
    {
        // 如果 x 是其父节点的左子节点
        if (x == leftOf(parentOf(x)))
        {
            // 获取 x 节点的兄弟节点
            Entry<K,V> sib = rightOf(parentOf(x));
            // 如果 sib 节点是红色
            if (colorOf(sib) == RED)
            {
                // 将 sib 节点设为黑色
                setColor(sib, BLACK);
                // 将 x 的父节点设为红色
                setColor(parentOf(x), RED);
                rotateLeft(parentOf(x));
                // 再次将 sib 设为 x 的父节点的右子节点
                sib = rightOf(parentOf(x));
            }
            // 如果 sib 的两个子节点都是黑色
            if (colorOf(leftOf(sib)) == BLACK
                && colorOf(rightOf(sib)) == BLACK)
            {
                // 将 sib 设为红色
                setColor(sib, RED);
                // 让 x 等于 x 的父节点
                x = parentOf(x);
            }
            else
            {
                // 如果 sib 的只有右子节点是黑色
```

```
                if (colorOf(rightOf(sib)) == BLACK)
                {
                    // 将 sib 的左子节点也设为黑色
                    setColor(leftOf(sib), BLACK);
                    // 将 sib 设为红色
                    setColor(sib, RED);
                    rotateRight(sib);
                    sib = rightOf(parentOf(x));
                }
                // 设置 sib 的颜色与 x 的父节点的颜色相同
                setColor(sib, colorOf(parentOf(x)));
                // 将 x 的父节点设为黑色
                setColor(parentOf(x), BLACK);
                // 将 sib 的右子节点设为黑色
                setColor(rightOf(sib), BLACK);
                rotateLeft(parentOf(x));
                x = root;
            }
        }
        // 如果 x 是其父节点的右子节点
        else
        {
            // 获取 x 节点的兄弟节点
            Entry<K,V> sib = leftOf(parentOf(x));
            // 如果 sib 的颜色是红色
            if (colorOf(sib) == RED)
            {
                // 将 sib 的颜色设为黑色
                setColor(sib, BLACK);
                // 将 sib 的父节点设为红色
                setColor(parentOf(x), RED);
                rotateRight(parentOf(x));
                sib = leftOf(parentOf(x));
            }
            // 如果 sib 的两个子节点都是黑色
            if (colorOf(rightOf(sib)) == BLACK
                && colorOf(leftOf(sib)) == BLACK)
            {
                // 将 sib 设为红色
                setColor(sib, RED);
                // 让 x 等于 x 的父节点
                x = parentOf(x);
            }
            else
            {
                // 如果 sib 只有左子节点是黑色
                if (colorOf(leftOf(sib)) == BLACK)
                {
                    // 将 sib 的右子节点也设为黑色
                    setColor(rightOf(sib), BLACK);
                    // 将 sib 设为红色
                    setColor(sib, RED);
                    rotateLeft(sib);
                    sib = leftOf(parentOf(x));
                }
                // 将 sib 的颜色设为与 x 的父节点颜色相同
                setColor(sib, colorOf(parentOf(x)));
                // 将 x 的父节点设为黑色
```

```
                setColor(parentOf(x), BLACK);
                // 将 sib 的左子节点设为黑色
                setColor(leftOf(sib), BLACK);
                rotateRight(parentOf(x));
                x = root;
            }
        }
    }
    setColor(x, BLACK);
 }
```

检索节点

当 TreeMap 根据 key 来取出 value 时，TreeMap 对应的方法如下：

```
public V get(Object key)
 {
    // 根据指定 key 取出对应的 Entry
    Entry>K,V< p = getEntry(key);
    // 返回该 Entry 所包含的 value
    return (p==null ? null : p.value);
 }
```

显示较少

从上面程序的粗体字代码可以看出，get(Object key) 方法实质是由于 getEntry() 方法实现的，这个 getEntry() 方法的代码如下：

```
final Entry<K,V> getEntry(Object key)
 {
    // 如果 comparator 不为 null，表明程序采用定制排序
    if (comparator != null)
        // 调用 getEntryUsingComparator 方法来取出对应的 key
        return getEntryUsingComparator(key);
    // 如果 key 形参的值为 null，抛出 NullPointerException 异常
    if (key == null)
        throw new NullPointerException();
    // 将 key 强制类型转换为 Comparable 实例
    Comparable<? super K> k = (Comparable<? super K>) key;
    // 从树的根节点开始
    Entry<K,V> p = root;
    while (p != null)
    {
        // 拿 key 与当前节点的 key 进行比较
        int cmp = k.compareTo(p.key);
        // 如果 key 小于当前节点的 key，向"左子树”搜索
        if (cmp < 0)
            p = p.left;
        // 如果 key 大于当前节点的 key，向"右子树”搜索
        else if (cmp > 0)
            p = p.right;
        // 不大于、不小于，就是找到了目标 Entry
        else
            return p;
    }
    return null;
```

```
}
```

显示较少

上面的 getEntry(Object obj) 方法也是充分利用排序二叉树的特征来搜索目标 Entry，程序依然从二叉树的根节点开始，如果被搜索节点大于当前节点，程序向”右子树”搜索；如果被搜索节点小于当前节点，程序向”左子树”搜索；如果相等，那就是找到了指定节点。

当 TreeMap 里的 comparator != null 即表明该 TreeMap 采用了定制排序，在采用定制排序的方式下，TreeMap 采用 getEntryUsingComparator(key) 方法来根据 key 获取 Entry。下面是该方法的代码：

```
final Entry<K,V> getEntryUsingComparator(Object key)
 {
    K k = (K) key;
    // 获取该 TreeMap 的 comparator
    Comparator<? super K> cpr = comparator;
    if (cpr != null)
    {
        // 从根节点开始
        Entry<K,V> p = root;
        while (p != null)
        {
            // 拿 key 与当前节点的 key 进行比较
            int cmp = cpr.compare(k, p.key);
            // 如果 key 小于当前节点的 key，向"左子树”搜索
            if (cmp < 0)
                p = p.left;
            // 如果 key 大于当前节点的 key，向"右子树”搜索
            else if (cmp > 0)
                p = p.right;
            // 不大于、不小于，就是找到了目标 Entry
            else
                return p;
        }
    }
    return null;
 }
```

显示较少

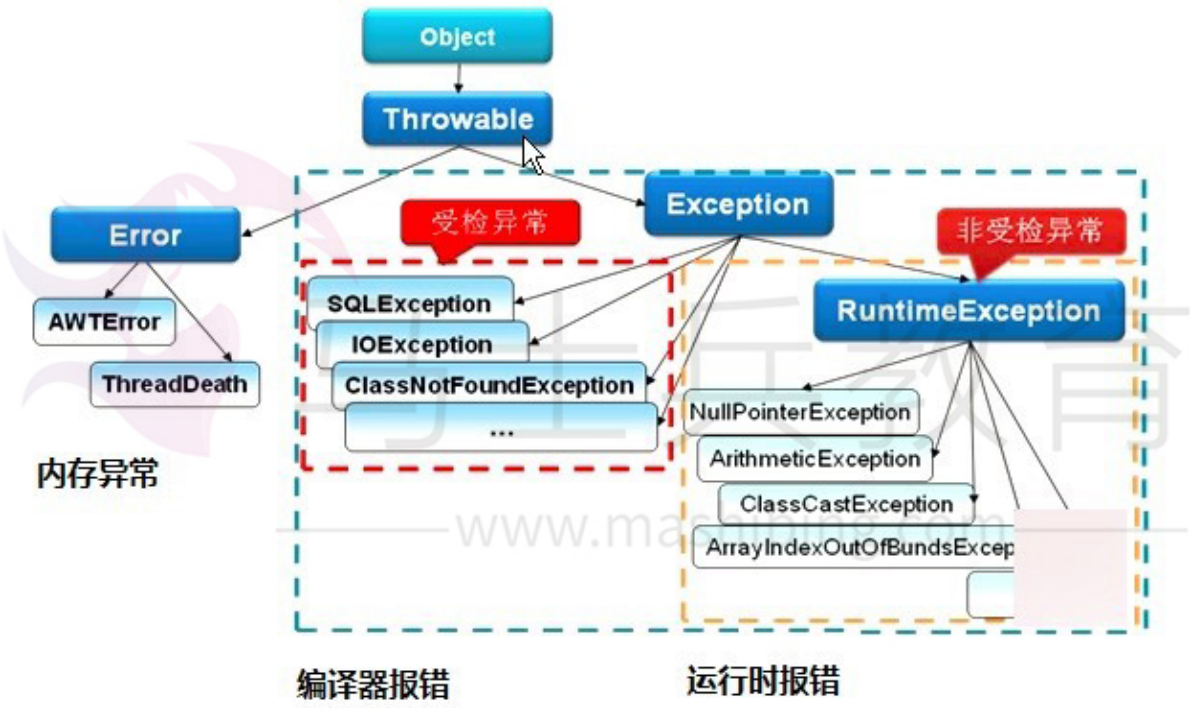
其实 getEntry、getEntryUsingComparator 两个方法的实现思路完全类似，只是前者对自然排序的 TreeMap 获取有效，后者对定制排序的 TreeMap 有效。

通过上面源代码的分析不难看出，TreeMap 这个工具类的实现其实很简单。或者说：从内部结构来看，TreeMap 本质上就是一棵”红黑树”，而 TreeMap 的每个 Entry 就是该红黑树的一个节点。

JAVA 异常分类及处理

概念

如果某个方法不能按照正常的途径完成任务，就可以通过另一种路径退出方法。在这种情况下会抛出一个封装了错误信息的对象。此时，这个方法会立刻退出同时不返回任何值。另外，调用这个方法的其他代码也无法继续执行，异常处理机制会将代码执行交给异常处理器。



异常分类

Throwable 是 Java 语言中所有错误或异常的超类。下一层分为 Error 和 Exception

Error

1. Error 类是指 java 运行时系统的内部错误和资源耗尽错误。应用程序不会抛出该类对象。如果出现了这样的错误，除了告知用户，剩下的就是尽力使程序安全的终止。

Exception（RuntimeException、CheckedException）

2. Exception 又有两个分支，一个是运行时异常 RuntimeException ， 一个是 ckedException。

RuntimeException 如： NullPointerException 、 ClassCastException； 一个是检查异常 CheckedException，如 I/O 错误导致的 IOException、SQLException。 RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。 如果出现 RuntimeException，那么一定是程序员的错误.

检查异常 CheckedException：一般是外部错误，这种异常都发生在编译阶段，Java 编译器会强制程序去捕获此类异常，即会出现要求你把这段可能出现异常的程序进行 try catch，该类异常一般包括几个方面：

1. 试图在文件尾部读取数据
2. 试图打开一个错误格式的 URL
3. 试图根据给定的字符串查找 class 对象，而这个字符串表示的类并不存在

异常的处理方式

遇到问题不进行具体处理，而是继续抛给调用者 （**throw,throws**）抛出异常有三种形式，一是 throw，一个 throws，还有一种系统自动抛异常。

```
public static void main(String[] args) {
String s = "abc";
if(s.equals("abc")) {
throw new NumberFormatException();
    } else {
      System.out.println(s);
    }
}
int div(int a,int b) throws Exception{ return a/b;}
```

try catch 捕获异常针对性处理方式

Throw和 throws 的区别：

位置不同

1. throws 用在函数上，后面跟的是异常类，可以跟多个；而 throw 用在函数内，后面跟的是异常对象。

功能不同：

2. throws 用来声明异常，让调用者只知道该功能可能出现的问题，可以给出预先的处理方式；throw 抛出具体的问题对象，执行到throw，功能就已经结束了，跳转到调用者，并将具体的问题对象抛给调用者。也就是说 throw 语句独立存在时，下面不要定义其他语句，因为执行不到。
3. throws 表示出现异常的一种可能性，并不一定会发生这些异常；throw 则是抛出了异常，执行 throw 则一定抛出了某种异常对象。
4. 两者都是消极处理异常的方式，只是抛出或者可能抛出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

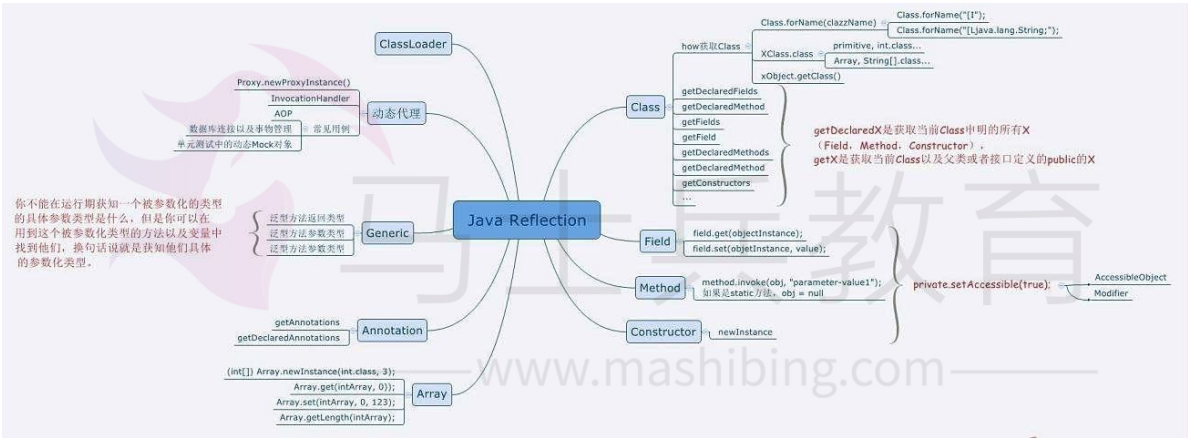
JAVA 反射

动态语言

动态语言，是指程序在运行时可以改变其结构：新的函数可以引进，已有的函数可以被删除等结

构上的变化。比如常见的 JavaScript 就是动态语言，除此之外 Ruby,Python 等也属于动态语言，而 C、C++则不属于动态语言。从反射角度说 JAVA 属于半动态语言。

反射机制概念 （运行状态中知道类所有的属性和方法）



在 Java 中的反射机制是指在运行状态中，对于任意一个类都能够知道这个类所有的属性和方法；并且对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息以及动态调用对象方法的功能成为 Java 语言的反射机制。

反射的应用场合

编译时类型和运行时类型

在 Java 程序中许多对象在运行是都会出现两种类型：编译时类型和运行时类型。 编译时的类型由声明对象时实用的类型来决定，运行时的类型由实际赋值给对象的类型决定 。如：

Person p=new Student(); 其中编译时类型为 Person，运行时类型为 Student。

的编译时类型无法获取具体方法

程序在运行时还可能接收到外部传入的对象，该对象的编译时类型为 Object,但是程序有需要调用该对象的运行时类型的方法。为了解决这些问题，程序需要在运行时发现对象和类的真实信息。然而，如果编译时根本无法预知该对象和类属于哪些类，程序只能依靠运行时信息来发现该对象和类的真实信息，此时就必须使用到反射了。

Java反射 API

反射**API**用来生成**JVM**中的类、接口或则对象的信息。

1. Class 类：反射的核心类，可以获取类的属性，方法等信息。
2. Field 类：Java.lang.reflec 包中的类，表示类的成员变量，可以用来获取和设置类之中的属性值。
3. Method 类： Java.lang.reflec 包中的类，表示类的方法，它可以用来获取类中的方法信息或者执行方法。
4. Constructor 类： Java.lang.reflec 包中的类，表示类的构造方法。

反射使用步骤（获取 Class 对象、调用对象方法）

1. 获取想要操作的类的 Class 对象，他是反射的核心，通过 Class 对象我们可以任意调用类的方法。
2. 调用 Class 类中的方法，既就是反射的使用阶段。
3. 使用反射 API 来操作这些信息。

获取 Class 对象的 3 种方法

调用某个对象的**getClass()**方法

Person p=new Person();

Class clazz=p.getClass();

调用某个类的**class**属性来获取该类对应的**Class**对象

Class clazz=Person.class;

使用**Class**类中的**forName()**静态方法**(**最安全**/**性能最好**)**

Class clazz=Class.forName("类的全路径"); (最常用)

当我们获得了想要操作的类的 Class 对象后，可以通过 Class 类中的方法获取并查看该类中的方法和属性。

```
  //获取 Person 类的 Class 对象
            Class clazz=Class.forName("reflection.Person");
               //获取 Person 类的所有方法信息
                  Method[] method=clazz.getDeclaredMethods();
for(Method m:method){
                System.out.println(m.toString());
            }
            //获取 Person 类的所有成员属性信息
            Field[] field=clazz.getDeclaredFields();            for(Field
f:field){
                System.out.println(f.toString());
```

```
            }
            //获取 Person 类的所有构造方法信息
            Constructor[] constructor=clazz.getDeclaredConstructors();
  for(Constructor c:constructor){
                System.out.println(c.toString());
            }
```

创建对象的两种方法

Class对象的**newInstance()**

1. 使用 Class 对象的 newInstance()方法来创建该 Class 对象对应类的实例，但是这种方法要求该 Class 对象对应的类有默认的空构造器。

调用**Constructor** 对象的**newInstance()**

2. 先使用 Class 对象获取指定的 Constructor 对象，再调用 Constructor 对象的 newInstance() 方法来创建 Class 对象对应类的实例,通过这种方法可以选定构造方法创建实例。

```
  //获取 Person 类的 Class 对象
            Class clazz=Class.forName("reflection.Person");
            //使用.newInstane 方法创建对象
            Person p=(Person) clazz.newInstance();
       //获取构造方法并创建对象
            Constructor
c=clazz.getDeclaredConstructor(String.class,String.class,int.class);
            //创建对象并设置属性
  Person p1=(Person) c.newInstance("李四","男",20);
```

JAVA 注解

概念

Annotation（注解）是 Java 提供的一种对元程序中元素关联信息和元数据（metadata）的途径和方法。Annatation(注解)是一个接口，程序可以通过反射来获取指定程序中元素的 Annotation 对象，然后通过该 Annotation 对象来获取注解中的元数据信息。

4 种标准元注解

元注解的作用是负责注解其他注解。 Java5.0 定义了 4 个标准的 meta-annotation 类型，它们被用来提供对其它 annotation 类型作说明。

@Target 修饰的对象范围

@Target说明了Annotation所修饰的对象范围： Annotation可被用于 packages、types（类、接口、枚举、Annotation 类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch 参数）。在 Annotation 类型的声明中使用了 target 可更加明晰其修饰的目标

@Retention 定义 被保留的时间长短

Retention 定义了该 Annotation 被保留的时间长短：表示需要在什么级别保存注解信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效），取值（RetentionPoicy）由：

n SOURCE:在源文件中有效（即源文件保留）

n CLASS:在 class 文件中有效（即 class 保留）

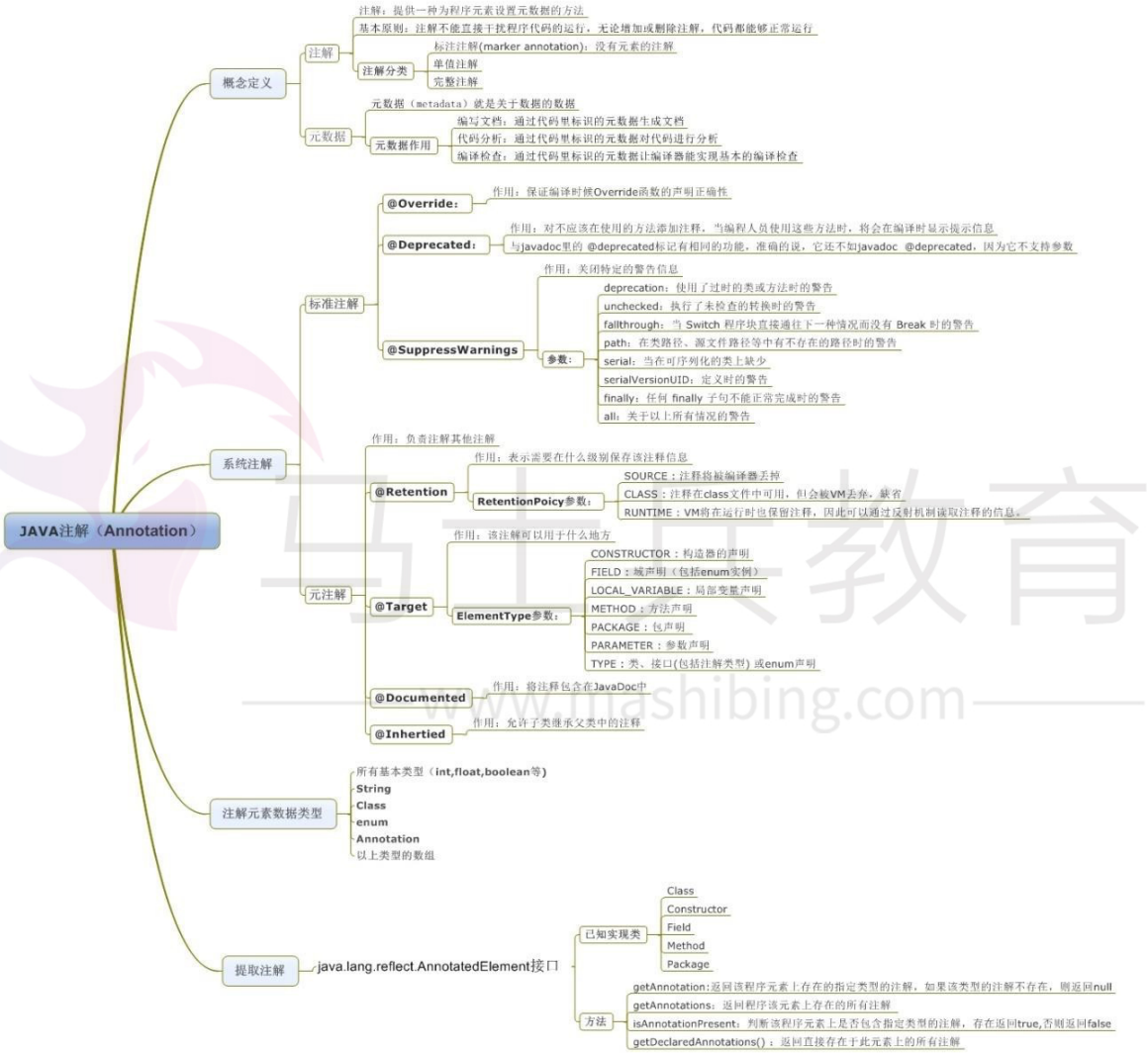
n RUNTIME:在运行时有效（即运行时保留）

@Documented 描述-javadoc

@ Documented 用于描述其它类型的 annotation 应该被作为被标注的程序成员的公共 API，因此可以被例如 javadoc 此类的工具文档化。

@Inherited 阐述了某个被标注的类型是被继承的

@Inherited 元注解是一个标记注解，@Inherited 阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited 修饰的 annotation 类型被用于一个 class，则这个 annotation 将被用于该 class 的子类。



注解处理器

如果没有用来读取注解的方法和工作，那么注解也就不会比注释更有用处了。使用注解的过程中，很重要的一部分就是创建于使用注解处理器。Java SE5扩展了反射机制的API，以帮助程序员快速的构造自定义注解处理器。下面实现一个注解处理器。

```
 /1：** 定义注解*/
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface FruitProvider {
    /**供应商编号*/
public int id() default -1;
/** 供应商名称*/
    public String name() default ""
```

```
   /** * 供应商地址*/
    public String address() default "";
}
//2：注解使用
public class Apple {
    @FruitProvider(id = 1, name = "陕西红富士集团", address = "陕西省西安市延安路")
    private String appleProvider;     public void setAppleProvider(String
appleProvider) {         this.appleProvider = appleProvider;
    }
    public String getAppleProvider() {         return appleProvider;
    }
}
/3：******** 注解处理器 **********/
public class FruitInfoUtil {     public static void getFruitInfo(Class<?> clazz)
{
        String strFruitProvicer = "供应商信息：";
        Field[] fields = clazz.getDeclaredFields();//通过反射获取处理注解
        for (Field field : fields) {              if
(field.isAnnotationPresent(FruitProvider.class)) {
       FruitProvider fruitProvider = (FruitProvider)
field.getAnnotation(FruitProvider.class);
 //注解信息的处理地方
strFruitProvicer = " 供应商编号：" + fruitProvider.id() + " 供应商名称："
                        + fruitProvider.name() + " 供应商地址："+
fruitProvider.address();                 System.out.println(strFruitProvicer);
            }
        }
    }
}
public class FruitRun {     public static void main(String[] args) {
        FruitInfoUtil.getFruitInfo(Apple.class);
    /********输出结果**********/
// 供应商编号：1 供应商名称：陕西红富士集团 供应商地址：陕西省西安市延
    }
}
```

JAVA 内部类

Java 类中不仅可以定义变量和方法，还可以定义类，这样定义在类内部的类就被称为内部类。根据定义的方式不同，内部类分为静态内部类，成员内部类，局部内部类，匿名内部类四种。

静态内部类

定义在类内部的静态类，就是静态内部类。

```
public class Out {
private static int a;
private int b;
public static class Inner {
public void print() {
           System.out.println(a);
        }
    } }
```

1. 静态内部类可以访问外部类所有的静态变量和方法，即使是 private 的也一样。

2. 静态内部类和一般类一致，可以定义静态变量、方法，构造方法等。
3. 其它类使用静态内部类需要使用“外部类.静态内部类”方式，如下所示：Out.Inner inner = new Out.Inner();inner.print();
4. Java集合类HashMap内部就有一个静态内部类Entry。Entry是HashMap存放元素的抽象，HashMap 内部维护 Entry 数组用了存放元素，但是 Entry 对使用者是透明的。像这种和外部类关系密切的，且不依赖外部类实例的，都可以使用静态内部类。

成员内部类

定义在类内部的非静态类，就是成员内部类。成员内部类不能定义静态方法和变量（final 修饰的除外）。这是因为成员内部类是非静态的，类初始化的时候先初始化静态成员，如果允许成员内部类定义静态变量，那么成员内部类的静态变量初始化顺序是有歧义的。

```
 public class Out {
 private static int a;
 private int b;
 public class Inner {
 public void print() {
 System.out.println(a);
            System.out.println(b);
        }
    }
}
```

局部内部类（定义在方法中的类）

定义在方法中的类，就是局部类。如果一个类只在某个方法中使用，则可以考虑使用局部类。

```
 public class Out {
 private static int a;
 private int b;
 public void test(final int c) {
 final int d = 1;
 class Inner {
 public void print() {
                      System.out.println(c);
               }
        }
    }
}
```

匿名内部类（要继承一个父类或者实现一个接口、直接使用 **new** 来生成一个对象的引用）

匿名内部类我们必须要继承一个父类或者实现一个接口，当然也仅能只继承一个父类或者实现一个接口。同时它也是没有class关键字，这是因为匿名内部类是直接使用new来生成一个对象的引用。

```
  public abstract class Bird {
  private String name;
  public String getName() {
  return name;
    }
     public void setName(String name) {
     this.name = name;
```

```
    }
     public abstract int fly();
}

 public class Test {
 public void test(Bird bird){
        System.out.println(bird.getName() + "能够飞 " + bird.fly() + "米");
    }
     public static void main(String[] args) {
     Test test = new Test();
     test.test(new Bird() {
     public int fly() {
     return 10000;
            }
             public String getName() {
             return "大雁";
            }
        });
    }
}
```

JAVA 泛型

泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。比如我们要写一个排序方法，能够对整型数组、字符串数组甚至其他任何类型的数组进行排序，我们就可以使用 Java 泛型。

泛型方法（）

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

```
// 泛型方法 printArray
  public static < E > void printArray( E[] inputArray )
  {
        for ( E element : inputArray ){
           System.out.printf( "%s ", element );
        }
       }
```

1. <? extends T>表示该通配符所代表的类型是 T 类型的子类。
2. <? super T>表示该通配符所代表的类型是 T 类型的父类。

泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

```
public class Box<T> {
  private T t;
   public void add(T t) {
   this.t = t;
  }
   public T get() {
   return t;
  }
```

类型通配符**?**

类型通配符一般是使用?代替具体的类型参数。例如List<?>在逻辑上是

List,List等所有List<具体类型实参>的父类。

类型擦除

Java 中的泛型基本上都是在编译器这个层次来实现的。在生成的 Java 字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉。这个过程就称为类型擦除。如在代码中定义的 List和 List等类型，在编译之后都会变成 List。JVM 看到的只是 List，而由泛型附加的类型信息对 JVM 来说是不可见的。

类型擦除的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是 Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换成具体的类。

JAVA 序列化(创建可复用的 Java 对象)

保存**(**持久化**)**对象及其状态到内存或者磁盘

Java 平台允许我们在内存中创建可复用的 Java 对象，但一般情况下，只有当 JVM 处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比 JVM 的生命周期更长。但在现实应用中，就可能要求在JVM停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。

Java 对象序列化就能够帮助我们实现该功能。

序列化对象以字节数组保持-静态成员不保存

使用 Java 对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的"状态"，即它的成员变量。由此可知，对象序列化不会关注类中的静态变量。

序列化用户远程对象传输

除了在持久化对象时会用到对象序列化之外，当使用 RMI(远程方法调用)，或在网络中传递对象时，都会用到对象序列化。Java序列化API为处理对象序列化提供了一个标准机制，该API简单易用。

Serializable 实现序列化

在 Java 中，只要一个类实现了 java.io.Serializable 接口，那么它就可以被序列化。

ObjectOutputStream 和**ObjectInputStream** 对对象进行序列化及反序列化

通过 ObjectOutputStream 和 ObjectInputStream 对对象进行序列化及反序列化。

writeObject 和 **readObject** 自定义序列化策略

在类中增加 writeObject 和 readObject 方法可以实现自定义序列化策略。

序列化 **ID**

虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（就是 private static final long serialVersionUID）

序列化并不保存静态变量序列化子父类说明

要想将父类对象也序列化，就需要让父类也实现 Serializable 接口。

Transient 关键字阻止该变量被序列化到文件中

1. 在变量声明前加上 Transient 关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。
2. 服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全。

JAVA 复制

将一个对象的引用复制给另外一个对象，一共有三种方式。第一种方式是直接赋值，第二种方式是浅拷贝，第三种是深拷贝。所以大家知道了哈，这三种概念实际上都是为了拷贝对象。

直接赋值复制

直接赋值。在 Java 中，A a1 = a2，我们需要理解的是这实际上复制的是引用，也就是说 a1 和 a2 指向的是同一个对象。因此，当 a1 变化的时候，a2 里面的成员变量也会跟着变化。

浅复制（复制引用但不复制引用的对象）

创建一个新对象，然后将当前对象的非静态字段复制到该新对象，如果字段是值类型的，那么对该字段执行复制；如果该字段是引用类型的话，则复制引用但不复制引用的对象。

因此，原始对象及其副本引用同一个对象。

```
 class Resume  implements Cloneable{
     public Object clone() {
        try {
            return (Resume)super.clone();
        } catch (Exception e) {
        e.printStackTrace();
        return null;
        }
    }
}
```

深复制（复制对象和其应用对象）

深拷贝不仅复制对象本身，而且复制对象包含的引用指向的所有对象。

```
 class Student implements Cloneable {
 String name;
 int age;
    Professor p;
    Student(String name, int age, Professor p) {
    this.name = name;
    this.age = age;
    this.p = p;
    }
     public Object clone() {
     Student o = null;
```

```
        try {
            o = (Student) super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println(e.toString());
        }
        o.p = (Professor) p.clone();
        return o;
    }
}
```

序列化（深 clone 一中实现）

在Java 语言里深复制一个对象，常常可以先使对象实现 Serializable 接口，然后把对象（实际上只是对象的一个拷贝）写到一个流里，再从流里读出来，便可以重建对象。

Java9比Java8改进了什么；

1）引入了模块系统，采用模块化系统的应用程序只需要这些应用程序所需的那部分JDK模块， 而非是整个JDK框架了，减少了内存的开销。

2）引入了一个新的package:java.net.http，里面提供了对Http访问很好的支持，不仅支持 Http1.1而且还支持HTTP2。

3）引入了jshell这个交互性工具，让Java也可以像脚本语言一样来运行，可以从控制台启动 jshell ，在 jshell 中直接输入表达式并查看其执行结果。

4）增加了List.of()、Set.of()、Map.of()和Map.ofEntries()等工厂方法来创建不可变集合

5）HTML5风格的Java帮助文档

6）多版本兼容 JAR 功能，能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本。

7）统一 JVM 日志
可以使用新的命令行选项-Xlog 来控制JVM 上 所有组件的日志记录。该日志记录系统可以设置 输出的日志消息的标签、级别、修饰符和输出目标等。

8）垃圾收集机制 Java 9 移除了在 Java 8 中 被废弃的垃圾回收器配置组合，同时把G1设为默认的垃圾回收器实 现.因为相对于Parallel来说，G1会在应用线程上做更多的事情，而Parallel几乎没有在应用线程上做任何事情，它基本上完全依赖GC线程完成所有的内存管理。这意味着切换到G1将会为应用 线程带来额外的工作，从而直接影响到应用的性能

9）I/O 流新特性 java.io.InputStream 中增加了新的方法来读取和复制 InputStream 中包含的数据。 readAllBytes：读取 InputStream 中的所有剩余字节。 readNBytes： 从 InputStream 中读取指定数量的字节到数组中。 transferTo：读取 InputStream 中的全部字节并写入到指定的 OutputStream 中 。

JAVA8 十大新特性详解

本教程将Java8的新特新逐一列出，并将使用简单的代码示例来指导你如何使用默认接口方法，lambda表达式，方法引用以及多重Annotation，之后你将会学到最新的API上的改进，比如流，函数式接口，Map以及全新的日期API

“Java is still not dead—and people are starting to figure that out.”

本教程将用带注释的简单代码来描述新特性，你将看不到大片吓人的文字。

一、接口的默认方法

Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法，示例如下：

代码如下:

```
interface Formula {
  double calculate(int a);


  default double sqrt(int a) {
    return Math.sqrt(a);
  }
}
```

Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。

代码如下:

```
Formula formula = new Formula() {
  @Override
  public double calculate(int a) {
    return sqrt(a * 100);
  }
};


formula.calculate(100);   // 100.0
formula.sqrt(16);      // 4.0
```

文中的formula被实现为一个匿名类的实例，该代码非常容易理解，6行代码实现了计算 sqrt(a * 100)。在下一节中，我们将会看到实现单方法接口的更简单的做法。

译者注： 在Java中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现，在C++中支持多继承，允许一个子类同时具有多个父类的接口与功能，在其他语言中，让一个类同时具有其他的可复用代码的方法叫做mixin。新的Java 8 的这个特新在编译器实现的角度上来说更加接近Scala的trait。 在C#中也有名为扩展方法的概念，允许给已存在的类型扩展方法，和Java 8的这个在语义上有差别。

二、Lambda 表达式

首先看看在老版本的Java中是如何排列字符串的：

代码如下:

```
List names = Arrays.asList("peterF", "anna", "mike", "xenia");


Collections.sort(names, new Comparator() {
  @Override
  public int compare(String a, String b) {
    return b.compareTo(a);
  }
});
```

只需要给静态方法 Collections.sort 传入一个List对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给sort方法。

在Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8提供了更简洁的语法，lambda表达式：

代码如下:

```
Collections.sort(names, (String a, String b) -> {
  return b.compareTo(a);
});
```

看到了吧，代码变得更段且更具有可读性，但是实际上还可以写得更短：

代码如下:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

对于函数体只有一行代码的，你可以去掉大括号{}以及return关键字，但是你还可以写得更短点：

代码如下:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Java编译器可以自动推导出参数类型，所以你可以不用再写一次类型。接下来我们看看lambda表达式还能作出什么更方便的东西来：

三、函数式接口

Lambda表达式是如何在java的类型系统中表示的呢？每一个lambda表达式都对应一个类型，通常是接口类型。而“函数式接口”是指仅仅只包含一个抽象方法的接口，每一个该类型的lambda表达式都会被匹配到这个抽象方法。因为 默认方法 不算抽象方法，所以你也可以给你的函数式接口添加默认方法。

我们可以将lambda表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 @FunctionalInterface 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

示例如下：

代码如下:

```
@FunctionalInterface
interface Converter<F, T> {
  T convert(F from);
}
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);  // 123
```

需要注意如果@FunctionalInterface如果没有指定，上面的代码也是对的。

译者注 将lambda表达式映射到一个单方法的接口上，这种做法在Java 8之前就有别的语言实现，比如Rhino JavaScript解释器，如果一个函数参数接收一个单方法的接口而你传递的是一个function，Rhino解释器会自动做一个单接口的实例到function的适配器，典型的应用场景有org.w3c.dom.events.EventTarget 的addEventListener 第二个参数 EventListener。

四、方法与构造函数引用

前一节中的代码还可以通过静态方法引用来表示：

代码如下:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);  // 123
```

Java 8 允许你使用 :: 关键字来传递方法或者构造函数引用，上面的代码展示了如何引用一个静态方法，我们也可以引用一个对象的方法：

代码如下:

```
 converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);  // "J"
```

接下来看看构造函数是如何使用::关键字来引用的，首先我们定义一个包含多个构造函数的简单类：

代码如下:

```
class Person {
  String firstName;
  String lastName;


  Person() {}

  Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

接下来我们指定一个用来创建Person对象的对象工厂接口：

代码如下:

```
interface PersonFactory
```

```
{
  P create(String firstName, String lastName);
}
```

这里我们使用构造函数引用来将他们关联起来，而不是实现一个完整的工厂：

代码如下:

```
PersonFactory personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

我们只需要使用 Person::new 来获取Person类构造函数的引用，Java编译器会自动根据PersonFactory.create方法的签名来选择合适的构造函数。

五、Lambda 作用域

在lambda表达式中访问外层作用域和老版本的匿名对象中的方式很相似。你可以直接访问标记了final的外层局部变量，或者实例的字段以及静态变量。

六、访问局部变量

我们可以直接在lambda表达式中访问外层的局部变量：

代码如下:

```
final int num = 1;
Converter<Integer, String> stringConverter =
    (from) -> String.valueOf(from + num);


stringConverter.convert(2);   // 3
```

但是和匿名对象不同的是，这里的变量num可以不用声明为final，该代码同样正确：

代码如下:

```
int num = 1;
Converter<Integer, String> stringConverter =
    (from) -> String.valueOf(from + num);


stringConverter.convert(2);   // 3
```

不过这里的num必须不可被后面的代码修改（即隐性的具有final的语义），例如下面的就无法编译：

代码如下:

```
int num = 1;
Converter<Integer, String> stringConverter =
    (from) -> String.valueOf(from + num);
num = 3;
```

在lambda表达式中试图修改num同样是不允许的。

七、访问对象字段与静态变量

和本地变量不同的是，lambda内部对于实例的字段以及静态变量是即可读又可写。该行为和匿名对象是一致的：

代码如下:

```
class Lambda4 {
  static int outerStaticNum;
  int outerNum;


  void testScopes() {
    Converter<Integer, String> stringConverter1 = (from) -> {
      outerNum = 23;
      return String.valueOf(from);
    };

    Converter<Integer, String> stringConverter2 = (from) -> {
      outerStaticNum = 72;
      return String.valueOf(from);
    };

  }
}
```

八、访问接口的默认方法

还记得第一节中的formula例子么，接口Formula定义了一个默认方法sqrt可以直接被formula的实例包括匿名对象访问到，但是在lambda表达式中这个是不行的。
Lambda表达式中是无法访问到默认方法的，以下代码将无法编译：

代码如下:

```
Formula formula = (a) -> sqrt( a * 100);
```

Built-in Functional Interfaces

JDK 1.8 API包含了很多内建的函数式接口，在老Java中常用到的比如Comparator或者Runnable接口，这些接口都增加了@FunctionalInterface注解以便能用在lambda上。
Java 8 API同样还提供了很多全新的函数式接口来让工作更加方便，有一些接口是来自Google Guava库里的，即便你对这些很熟悉了，还是有必要看看这些是如何扩展到lambda上使用的。

Predicate接口

Predicate 接口只有一个参数，返回boolean类型。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）：

代码如下:

```
Predicate predicate = (s) -> s.length() > 0;
```

```
predicate.test("foo");       // true
predicate.negate().test("foo");   // false

Predicate nonNull = Objects::nonNull;
Predicate isNull = Objects::isNull;

Predicate isEmpty = String::isEmpty;
Predicate isNotEmpty = isEmpty.negate();
```

Function 接口

Function 接口有一个参数并且返回一个结果，并附带了一些可以和其他函数组合的默认方法（compose, andThen）：

代码如下:

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);


backToString.apply("123");   // "123"
```

**Supplier 接口

**
Supplier 接口返回一个任意范型的值，和Function接口不同的是该接口没有任何参数

代码如下:

```
Supplier personSupplier = Person::new;
personSupplier.get();  // new Person
```

**Consumer 接口

**
Consumer 接口表示执行在单个参数上的操作。

代码如下:

```
Consumer greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

**Comparator 接口

**
Comparator 是老Java中的经典接口， Java 8在此之上添加了多种默认方法：

代码如下:

```
Comparator comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);
```

```
Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);       // > 0
comparator.reversed().compare(p1, p2); // < 0
```

Optional 接口

Optional 不是函数是接口，这是个用来防止NullPointerException异常的辅助类型，这是下一届中将要用到的重要概念，现在先简单的看看这个接口能干什么：

Optional 被定义为一个简单的容器，其值可能是null或者不是null。在Java 8之前一般某个函数应该返回非空对象但是偶尔却可能返回了null，而在Java 8中，不推荐你返回null而是返回Optional。

代码如下:

```
Optional optional = Optional.of("bam");


optional.isPresent();      // true
optional.get();         // "bam"
optional.orElse("fallback");  // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));   // "b"
```

Stream 接口

java.util.Stream 表示能应用在一组元素上一次执行的操作序列。Stream 操作分为中间操作或者最终操作两种，最终操作返回一特定类型的计算结果，而中间操作返回Stream本身，这样你就可以将多个操作依次串起来。Stream 的创建需要指定一个数据源，比如 java.util.Collection的子类，List或者Set，Map不支持。Stream的操作可以串行执行或者并行执行。

首先看看Stream是怎么用，首先创建实例代码的用到的数据List：

代码如下:

```
List stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

Java 8扩展了集合类，可以通过 Collection.stream() 或者 Collection.parallelStream() 来创建一个Stream。下面几节将详细解释常用的Stream操作：

Filter 过滤

过滤通过一个predicate接口来过滤并只保留符合条件的元素，该操作属于中间操作，所以我们可以在过滤后的结果来应用其他Stream操作（比如forEach）。forEach需要一个函数来对过滤后的元素依次执行。forEach是一个最终操作，所以我们不能在forEach之后来执行其他Stream操作。

代码如下:

```
stringCollection
  .stream()
  .filter((s) -> s.startsWith("a"))
  .forEach(System.out::println);


// "aaa2", "aaa1"
```

Sort 排序

排序是一个中间操作，返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序。

代码如下:

```
stringCollection
  .stream()
  .sorted()
  .filter((s) -> s.startsWith("a"))
  .forEach(System.out::println);


// "aaa1", "aaa2"
```

需要注意的是，排序只创建了一个排列好后的Stream，而不会影响原有的数据源，排序之后原数据stringCollection是不会被修改的：

代码如下:

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

**Map 映射

**
中间操作map会将元素根据指定的Function接口来依次将元素转成另外的对象，下面的示例展示了将字符串转换为大写字符串。你也可以通过map来讲对象转换成其他类型，map返回的Stream类型是根据你map传递进去的函数的返回值决定的。

代码如下:

```
stringCollection
  .stream()
  .map(String::toUpperCase)
  .sorted((a, b) -> b.compareTo(a))
  .forEach(System.out::println);
```

```
// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match 匹配

Stream提供了多种匹配操作，允许检测指定的Predicate是否匹配整个Stream。所有的匹配操作都是最终操作，并返回一个boolean类型的值。

代码如下:

```
boolean anyStartsWithA =
  stringCollection
    .stream()
    .anyMatch((s) -> s.startsWith("a"));


System.out.println(anyStartsWithA);   // true

boolean allStartsWithA =
  stringCollection
    .stream()
    .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);   // false

boolean noneStartsWithZ =
  stringCollection
    .stream()
    .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);   // true
```

**Count 计数

**
计数是一个最终操作，返回Stream中元素的个数，返回值类型是long。

代码如下:

```
long startsWithB =
  stringCollection
    .stream()
    .filter((s) -> s.startsWith("b"))
    .count();


System.out.println(startsWithB);  // 3
```

Reduce 规约

这是一个最终操作，允许通过指定的函数来讲stream中的多个元素规约为一个元素，规越后的结果是通过Optional接口表示的:

代码如下:

```
Optional reduced =
  stringCollection
    .stream()
    .sorted()
    .reduce((s1, s2) -> s1 + "#" + s2);


reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

并行Streams

前面提到过Stream有串行和并行两种，串行Stream上的操作是在一个线程中依次完成，而并行Stream则是在多个线程上同时执行。

下面的例子展示了是如何通过并行Stream来提升性能：

首先我们创建一个没有重复元素的大表：

代码如下:

```
int max = 1000000;
List values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
  UUID uuid = UUID.randomUUID();
  values.add(uuid.toString());
}
```

然后我们计算一下排序这个Stream要耗时多久，
串行排序：

代码如下:

```
long t0 = System.nanoTime();


long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms", millis));


// 串行耗时: 899 ms
```

并行排序：

代码如下:

```
long t0 = System.nanoTime();
```

```
long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms", millis));


// 并行排序耗时: 472 ms
```

上面两个代码几乎是一样的，但是并行版的快了50%之多，唯一需要做的改动就是将stream()改为parallelStream()。

Map

前面提到过，Map类型不支持stream，不过Map提供了一些新的有用的方法来处理一些日常任务。

代码如下:

```
Map<Integer, String> map = new HashMap<>();


for (int i = 0; i < 10; i++) {
  map.putIfAbsent(i, "val" + i);
}


map.forEach((id, val) -> System.out.println(val));
```

以上代码很容易理解， putIfAbsent 不需要我们做额外的存在性检查，而forEach则接收一个Consumer接口来对map里的每一个键值对进行操作。

下面的例子展示了map上的其他有用的函数：

代码如下:

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);       // val33


map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9);   // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);  // true

map.computeIfAbsent(3, num -> "bam");
map.get(3);       // val33
```

接下来展示如何在Map里删除一个键值全都匹配的项：

代码如下:

```
map.remove(3, "val3");
map.get(3);       // val33


map.remove(3, "val33");
map.get(3);       // null
```

另外一个有用的方法：

代码如下:

```
map.getOrDefault(42, "not found"); // not found
```

对Map的元素做合并也变得很容易了：

代码如下:

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9);       // val9


map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9);       // val9concat
```

Merge做的事情是如果键名不存在则插入，否则则对原键对应的值做合并操作并重新插入到map中。

九、Date API

Java 8 在包java.time下包含了一组全新的时间日期API。新的日期API和开源的Joda-Time库差不多，但又不完全一样，下面的例子展示了这组新API里最重要的一些部分：

Clock 时钟

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象。

代码如下:

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();


Instant instant = clock.instant();
Date legacyDate = Date.from(instant);  // legacy java.util.Date
```

Timezones 时区

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of来获取到。 时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

代码如下:

```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids


ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());
```

```
// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime 本地时间

LocalTime 定义了一个没有时区信息的时间，例如 晚上10点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差：

代码如下:

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);


System.out.println(now1.isBefore(now2)); // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween);    // -3
System.out.println(minutesBetween);   // -239
```

LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串。

代码如下:

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);    // 23:59:59


DateTimeFormatter germanFormatter =
  DateTimeFormatter
    .ofLocalizedTime(FormatStyle.SHORT)
    .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);  // 13:37
```

LocalDate 本地日期

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和LocalTime基本一致。下面的例子展示了如何给Date对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。

代码如下:

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);


LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
```

```
System.out.println(dayOfWeek);  // FRIDAY
```

从字符串解析一个LocalDate类型和解析LocalTime一样简单：

代码如下:

```
DateTimeFormatter germanFormatter =
  DateTimeFormatter
    .ofLocalizedDate(FormatStyle.MEDIUM)
    .withLocale(Locale.GERMAN);


LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
System.out.println(xmas);  // 2014-12-24
```

LocalDateTime 本地日期时间

LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。LocalDateTime和LocalTime还有LocalDate一样，都是不可变的。LocalDateTime提供了一些能访问具体字段的方法。

代码如下:

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);


DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek);   // WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month);     // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);  // 1439
```

只要附加上时区信息，就可以将其转换为一个时间点Instant对象，Instant时间点对象可以很容易的转换为老式的java.util.Date。

代码如下:

```
Instant instant = sylvester
    .atZone(ZoneId.systemDefault())
    .toInstant();


Date legacyDate = Date.from(instant);
System.out.println(legacyDate);   // Wed Dec 31 23:59:59 CET 2014
```

格式化LocalDateTime和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式：

代码如下:

```
DateTimeFormatter formatter =
  DateTimeFormatter
    .ofPattern("MMM dd, yyyy - HH:mm");
```

```
LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);
String string = formatter.format(parsed);
System.out.println(string);   // Nov 03, 2014 - 07:13
```

和java.text.NumberFormat不一样的是新版的DateTimeFormatter是不可变的，所以它是线程安全的。

十、Annotation 注解

在Java 8中支持多重注解了，先看个例子来理解一下是什么意思。
首先定义一个包装类Hints注解用来放置一组具体的Hint注解：

代码如下:

```
@interface Hints {
  Hint[] value();
}


@Repeatable(Hints.class)
@interface Hint {
  String value();
}
```

Java 8允许我们把同一个类型的注解使用多次，只需要给该注解标注一下@Repeatable即可。

例 1: 使用包装类当容器来存多个注解（老方法）

代码如下:

```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

例 2：使用多重注解（新方法）

代码如下:

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

第二个例子里java编译器会隐性的帮你定义好@Hints注解，了解这一点有助于你用反射来获取这些信息：

代码如下:

```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint);          // null


Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value().length); // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length);     // 2
```

即便我们没有在Person类上定义@Hints注解，我们还是可以通过 getAnnotation(Hints.class) 来获取@Hints注解，更加方便的方法是使用 getAnnotationsByType 可以直接获取到所有的@Hint注解。
另外Java 8的注解还增加到两种新的target上了：

代码如下:

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

关于Java 8的新特性就写到这了，肯定还有更多的特性等待发掘。JDK 1.8里还有很多很有用的东西，比如Arrays.parallelSort, StampedLock和CompletableFuture等等。

Java 9 逆天的十大新特性

在介绍java9之前，我们先来看看java成立到现在的所有版本。

1990年初，最初被命名为Oak；

1995年5月23日，Java语言诞生；

1996年1月，第一个JDK-JDK1.0诞生；

1996年4月，10个最主要的操作系统供应商申明将在其产品中嵌入Java技术；

1996年9月，约8.3万个网页应用了Java技术来制作；

1997年2月18日，JDK1.1发布；

1997年4月2日，JavaOne会议召开，参与者逾一万人，创当时全球同类会议纪录；

1997年9月，JavaDeveloperConnection社区成员超过十万；

1998年2月，JDK1.1被下载超过2,000,000次；

1998年12月8日，Java 2企业平台J2EE发布；

1999年6月，SUN公司发布Java三个版本：标准版（J2SE）、企业版（J2EE）和微型版（J2ME）；

2000年5月8日，JDK1.3发布；

2000年5月29日，JDK1.4发布；

2001年6月5日，Nokia宣布到2003年将出售1亿部支持Java的手机；

2001年9月24日，J2EE1.3发布；

2002年2月26日，J2SE1.4发布，此后Java的计算能力有了大幅提升；

2004年9月30日，J2SE1.5发布，成为Java语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5更名为Java SE 5.0；

2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名，以取消其中的数字“2”：J2EE更名为Java EE，J2SE更名为Java SE，J2ME更名为Java ME；

2006年12月，SUN公司发布JRE6.0；

2009年4月20日，甲骨文以74亿美元的价格购SUN公司，取得java的版权，业界传闻说这对Java程序员是个坏消息（其实恰恰相反）；

2010年11月，由于甲骨文对Java社区的不友善，因此Apache扬言将退出JCP；

2011年7月28日，甲骨文发布Java SE 7；

2014年3月18日，甲骨文发表Java SE 8；

2017年7月，甲骨文发表Java SE 9。

写在前面

**modularity System 模块系统

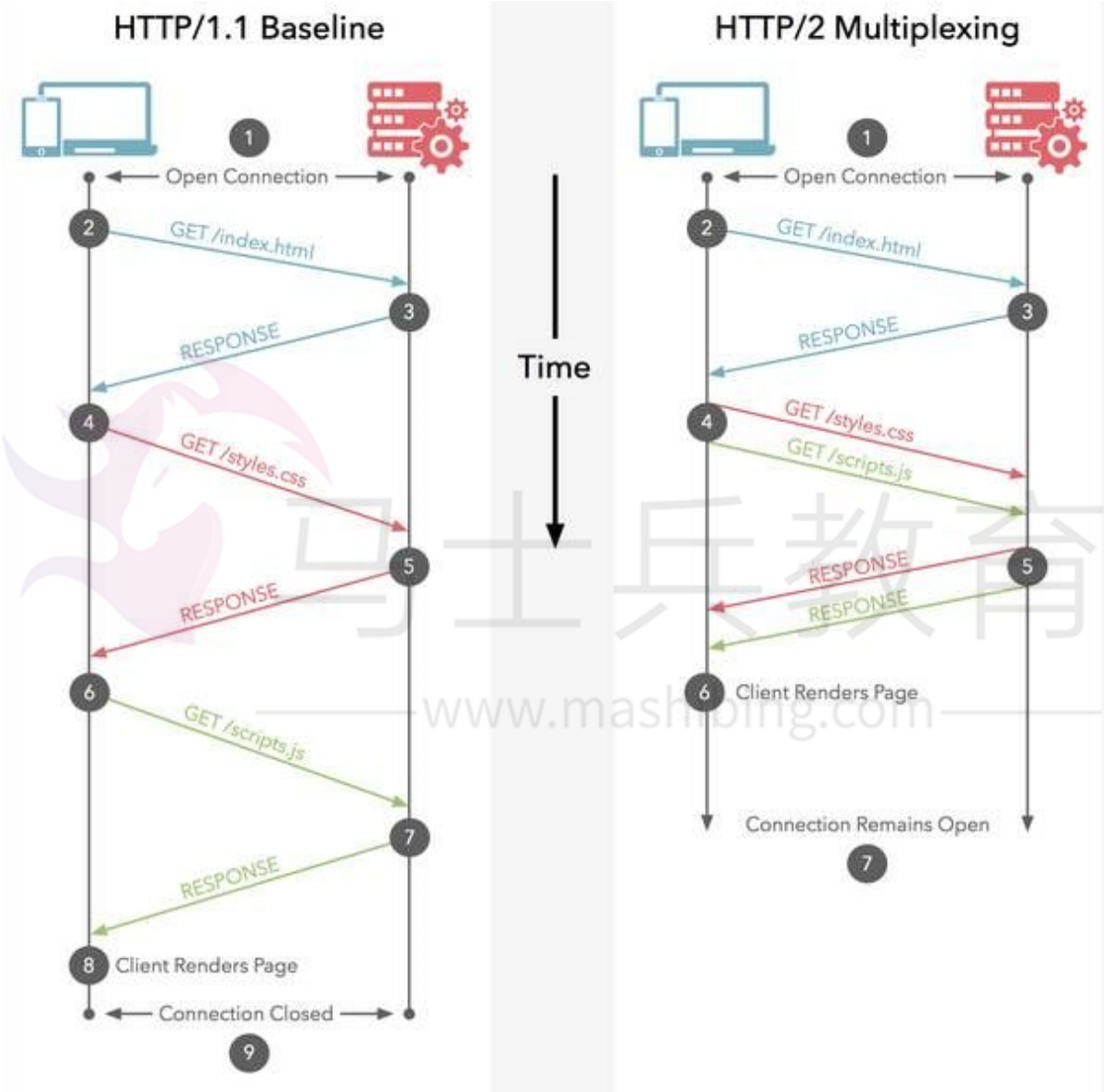
Java 9中主要的变化是已经实现的模块化系统。

Modularity提供了类似于OSGI框架的功能，模块之间存在相互的依赖关系，可以导出一个公共的API，并且隐藏实现的细节，Java提供该功能的主要的动机在于，减少内存的开销，在JVM启动的时候，至少会有30～60MB的内存加载，主要原因是JVM需要加载rt.jar，不管其中的类是否被classloader加载，第一步整个jar都会被JVM加载到内存当中去，模块化可以根据模块的需要加载程序运行需要的class。

在引入了模块系统之后，JDK 被重新组织成 94 个模块。Java 应用可以通过新增的 jlink 工具，创建出只包含所依赖的 JDK 模块的自定义运行时镜像。这样可以极大的减少 Java 运行时环境的大小。使得JDK可以在更小的设备中使用。采用模块化系统的应用程序只需要这些应用程序所需的那部分JDK模块，而非是整个JDK框架了。

HTTP/2

JDK9之前提供HttpURLConnection API来实现Http访问功能，但是这个类基本很少使用，一般都会选择Apache的Http Client，此次在Java 9的版本中引入了一个新的package:java.net.http，里面提供了对Http访问很好的支持，不仅支持Http1.1而且还支持HTTP2，以及WebSocket，据说性能特别好。



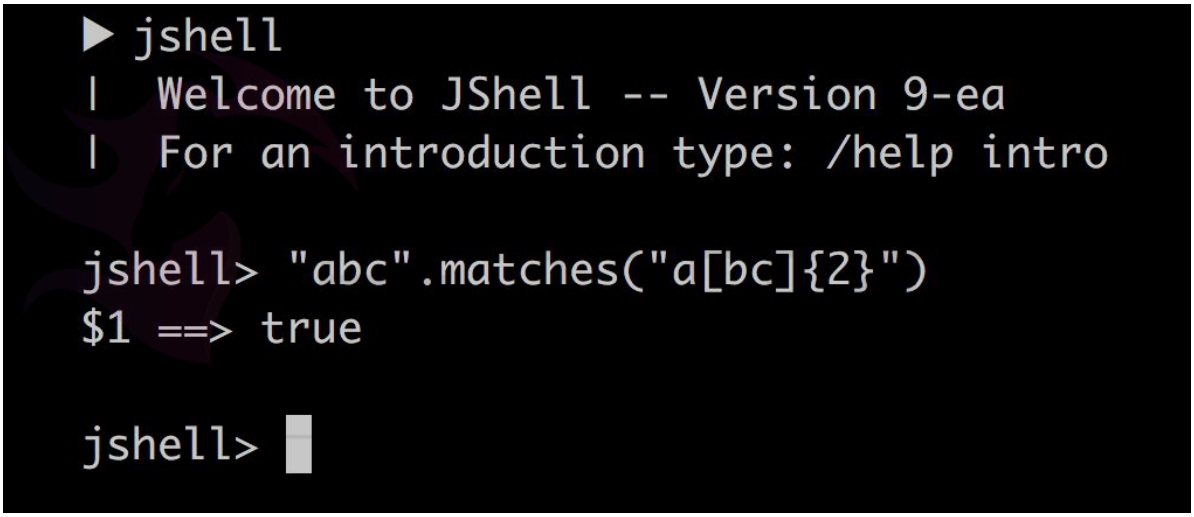
注意：新的 HttpClient API 在 Java 9 中以所谓的孵化器模块交付。也就是说，这套 API 不能保证 100% 完成。

JShell

用过Python的童鞋都知道，Python 中的读取-求值-打印循环（ Read-Evaluation-Print Loop ）很方便。它的目的在于以即时结果和反馈的形式。

java9引入了jshell这个交互性工具，让Java也可以像脚本语言一样来运行，可以从控制台启动 jshell，在 jshell 中直接输入表达式并查看其执行结果。当需要测试一个方法的运行效果，或是快速的对表达式进行求值时，jshell 都非常实用。

除了表达式之外，还可以创建 Java 类和方法。jshell 也有基本的代码完成功能。我们在教人们如何编写 Java 的过程中，不再需要解释 “public static void main（String [] args）” 这句废话。



不可变集合工厂方法

Java 9增加了List.of()、Set.of()、Map.of()和Map.ofEntries()等工厂方法来创建不可变集合。

```
List strs = List.of("Hello", "World");
List strs List.of(1, 2, 3);
Set strs = Set.of("Hello", "World");
Set ints = Set.of(1, 2, 3);
Map maps = Map.of("Hello", 1, "World", 2);
```

除了更短和更好阅读之外，这些方法也可以避免您选择特定的集合实现。在创建后，继续添加元素到这些集合会导致 “UnsupportedOperationException” 。

私有接口方法

Java 8 为我们提供了接口的默认方法和静态方法，接口也可以包含行为，而不仅仅是方法定义。

默认方法和静态方法可以共享接口中的私有方法，因此避免了代码冗余，这也使代码更加清晰。如果私有方法是静态的，那这个方法就属于这个接口的。并且没有静态的私有方法只能被在接口中的实例调用。

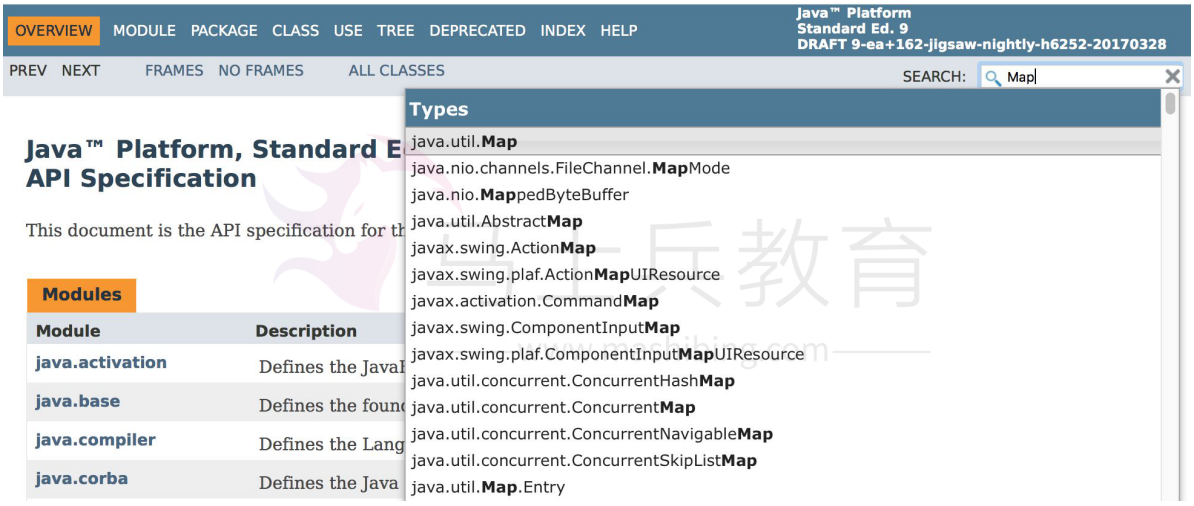
```
interface InterfaceWithPrivateMethods {
 private static String staticPrivate() {
  return "static private";
 }
 private String instancePrivate() {
```

```
        return "instance private";
    }
    default void check() {
        String result = staticPrivate();
        InterfaceWithPrivateMethods pvt = new InterfaceWithPrivateMethods() {
          // anonymous class 匿名类
        };
        result = pvt.instancePrivate();
    }
}
```

HTML5风格的Java帮助文档

Java 8之前的版本生成的Java帮助文档是在HTML 4中。在Java 9中，Javadoc 的输出现在符合兼容 HTML5 标准。现在HTML 4是默认的输出标记语言，但是在之后发布的JDK中，HTML 5将会是默认的输出标记语言。

Java帮助文档还是由三个框架组成的结构构成，这是不会变的，并且以HTML 5输出的Java帮助文档也保持相同的结构。每个 Javadoc 页面都包含有关 JDK 模块类或接口来源的信息。



多版本兼容 JAR

当一个新版本的 Java 出现的时候，你的库用户要花费很长时间才会切换到这个新的版本。这就意味着库要去向后兼容你想要支持的最老的 Java 版本 (许多情况下就是 Java 6 或者 7)。这实际上意味着未来的很长一段时间，你都不能在库中运用 Java 9 所提供的新特性。幸运的是，多版本兼容 JAR 功能能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本：

multirelease.jar

├── META-INF

│ └── versions

│ └── 9

│ └── multirelease

│ └── Helper.class

├── multirelease

├── Helper.class

└── Main.class

在上述场景中， multirelease.jar 可以在 Java 9 中使用, 不过 Helper 这个类使用的不是顶层的 multirelease.Helper 这个 class, 而是处在“META-INF/versions/9”下面的这个。这是特别为 Java 9 准备的 class 版本，可以运用 Java 9 所提供的特性和库。同时，在早期的 Java 诸版本中使用这个 JAR 也是能运行的，因为较老版本的 Java 只会看到顶层的这个 Helper 类。

统一 JVM 日志

Java 9 中，JVM 有了统一的日志记录系统，可以使用新的命令行选项-Xlog 来控制 JVM 上 所有组件的日志记录。该日志记录系统可以设置输出的日志消息的标签、级别、修饰符和输出目标等。

java9的垃圾收集机制

Java 9 移除了在 Java 8 中 被废弃的垃圾回收器配置组合，同时把G1设为默认的垃圾回收器实现。替代了之前默认使用的Parallel GC，对于这个改变，evens的评论是酱紫的：这项变更是很重要的，因为相对于Parallel来说，G1会在应用线程上做更多的事情，而Parallel几乎没有在应用线程上做任何事情，它基本上完全依赖GC线程完成所有的内存管理。这意味着切换到G1将会为应用线程带来额外的工作，从而直接影响到应用的性能

I/O 流新特性

java.io.InputStream 中增加了新的方法来读取和复制 InputStream 中包含的数据。

readAllBytes：读取 InputStream 中的所有剩余字节。

readNBytes： 从 InputStream 中读取指定数量的字节到数组中。

transferTo：读取 InputStream 中的全部字节并写入到指定的 OutputStream 中 。

除了上面这些以外，还有以下这么多的新特性，不再一一介绍。

102: Process API Updates

110: HTTP 2 Client

143: Improve Contended Locking

158: Unified JVM Logging

165: Compiler Control

193: Variable Handles

197: Segmented Code Cache

199: Smart Java Compilation, Phase Two

200: The Modular JDK

201: Modular Source Code

211: Elide Deprecation Warnings on Import Statements

212: Resolve Lint and Doclint Warnings

213: Milling Project Coin

214: Remove GC Combinations Deprecated in JDK 8

215: Tiered Attribution for javac

216: Process Import Statements Correctly

217: Annotations Pipeline 2.0

219: Datagram Transport Layer Security (DTLS)

220: Modular Run-Time Images

221: Simplified Doclet API

222: jshell: The Java Shell (Read-Eval-Print Loop)

223: New Version-String Scheme

224: HTML5 Javadoc

225: Javadoc Search

226: UTF-8 Property Files

227: Unicode 7.0

228: Add More Diagnostic Commands

229: Create PKCS12 Keystores by Default

231: Remove Launch-Time JRE Version Selection

232: Improve Secure Application Performance

233: Generate Run-Time Compiler Tests Automatically

235: Test Class-File Attributes Generated by javac

236: Parser API for Nashorn

237: Linux/AArch64 Port

238: Multi-Release JAR Files

240: Remove the JVM TI hprof Agent

241: Remove the jhat Tool

243: Java-Level JVM Compiler Interface

244: TLS Application-Layer Protocol Negotiation Extension

245: Validate JVM Command-Line Flag Arguments

246: Leverage CPU Instructions for GHASH and RSA

247: Compile for Older Platform Versions

248: Make G1 the Default Garbage Collector

249: OCSP Stapling for TLS

250: Store Interned Strings in CDS Archives

251: Multi-Resolution Images

252: Use CLDR Locale Data by Default

253: Prepare JavaFX UI Controls & CSS APIs for Modularization

254: Compact Strings

255: Merge Selected Xerces 2.11.0 Updates into JAXP

256: BeanInfo Annotations

257: Update JavaFX/Media to Newer Version of GStreamer

258: HarfBuzz Font-Layout Engine

259: Stack-Walking API

260: Encapsulate Most Internal APIs

261: Module System

262: TIFF Image I/O

263: HiDPI Graphics on Windows and Linux

264: Platform Logging API and Service

265: Marlin Graphics Renderer

266: More Concurrency Updates

267: Unicode 8.0

268: XML Catalogs

269: Convenience Factory Methods for Collections

270: Reserved Stack Areas for Critical Sections

271: Unified GC Logging

272: Platform-Specific Desktop Features

273: DRBG-Based SecureRandom Implementations

274: Enhanced Method Handles

275: Modular Java Application Packaging

276: Dynamic Linking of Language-Defined Object Models

277: Enhanced Deprecation

278: Additional Tests for Humongous Objects in G1

279: Improve Test-Failure Troubleshooting

280: Indify String Concatenation

281: HotSpot C++ Unit-Test Framework

282: jlink: The Java Linker

283: Enable GTK 3 on Linux

284: New HotSpot Build System

285: Spin-Wait Hints

287: SHA-3 Hash Algorithms

288: Disable SHA-1 Certificates

289: Deprecate the Applet API

290: Filter Incoming Serialization Data

292: Implement Selected ECMAScript 6 Features in Nashorn

294: Linux/s390x Port

295: Ahead-of-Time Compilation

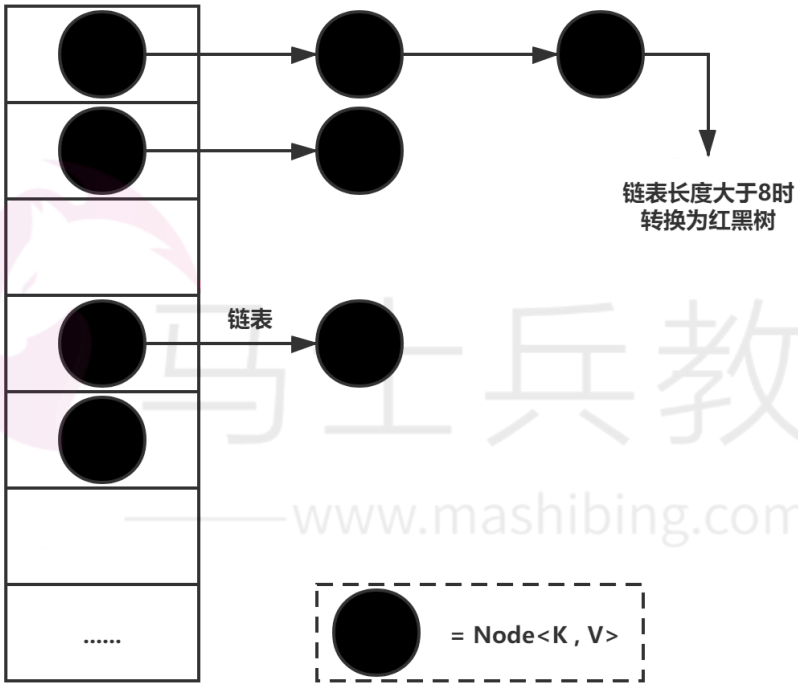
HashMap内部的数据结构是什么？底层是怎么实现的？

HashMap内部结构

jdk8以前：数组+链表

jdk8以后：数组+链表 （当链表长度到8时，转化为红黑树）

在并发的情况，发生扩容时，可能会产生循环链表，在执行get的时候，会触发死循环，引起 CPU的100%问题，所以一定要避免在并发环境下使用HashMap。



码士兵教育
www.mashibing.com

延伸考察ConcurrentHashMap与HashMap、HashTable等，考察 对技术细节的深入了解程度；

老生常谈，HashMap的死循环

问题

最近的几次面试中，我都问了是否了解HashMap在并发使用时可能发生死循环，导致cpu100%，结果让我很意外，都表示不知道有这样的问题，让我意外的是面试者的工作年限都不短。

由于HashMap并非是线程安全的，所以在高并发的情况下必然会出现问题，这是一个普遍的问题，虽然网上分析的文章很多，还是觉得有必须写一篇文章，让关注我公众号的同学能够意识到这个问题，并了解这个死循环是如何产生的。

如果是在单线程下使用HashMap，自然是没有问题的，如果后期由于代码优化，这段逻辑引入了多线程并发执行，在一个未知的时间点，会发现CPU占用100%，居高不下，通过查看堆栈，你会惊讶的发现，线程都Hang在hashMap的get()方法上，服务重启之后，问题消失，过段时间可能又复现了。

这是为什么?

原因分析

在了解来龙去脉之前，我们先看看HashMap的数据结构。

在内部，HashMap使用一个Entry数组保存key、value数据，当一对key、value被加入时，会通过一个hash算法得到数组的下标index，算法很简单，根据key的hash值，对数组的大小取模 hash & (length-1)，并把结果插入数组该位置，如果该位置上已经有元素了，就说明存在hash冲突，这样会在index位置生成链表。

如果存在hash冲突，最惨的情况，就是所有元素都定位到同一个位置，形成一个长长的链表，这样get一个值时，最坏情况需要遍历所有节点，性能变成了O(n)，所以元素的hash值算法和HashMap的初始化大小很重要。

当插入一个新的节点时，如果不存在相同的key，则会判断当前内部元素是否已经达到阈值（默认是数组大小的0.75），如果已经达到阈值，会对数组进行扩容，也会对链表中的元素进行rehash。

实现

HashMap的put方法实现：

1、判断key是否已经存在

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    // 如果key已经存在，则替换value，并返回旧值
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    // key不存在，则插入新的元素
    addEntry(hash, key, value, i);
    return null;
}
```

2、检查容量是否达到阈值threshold

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length);
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }

    createEntry(hash, key, value, bucketIndex);
}
```

如果元素个数已经达到阈值，则扩容，并把原来的元素移动过去。

3、扩容实现

```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    ...

    Entry[] newTable = new Entry[newCapacity];
    ...
    transfer(newTable, rehash);
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
}
```

这里会新建一个更大的数组，并通过transfer方法，移动元素。

```
void transfer(Entry[] newTable, boolean rehash) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            if (rehash) {
                e.hash = null == e.key ? 0 : hash(e.key);
            }
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```

移动的逻辑也很清晰，遍历原来table中每个位置的链表，并对每个元素进行重新hash，在新的newTable找到归宿，并插入。

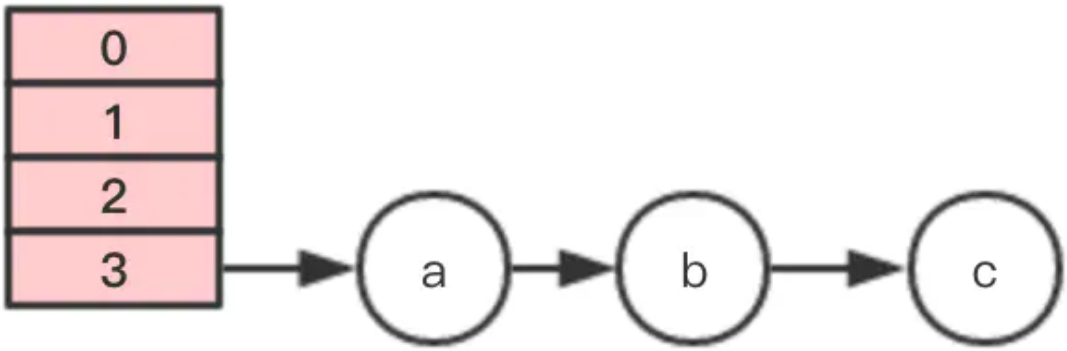
案例分析

假设HashMap初始化大小为4，插入个3节点，不巧的是，这3个节点都hash到同一个位置，如果按照默认的负载因子的话，插入第3个节点就会扩容，为了验证效果，假设负载因子是1.

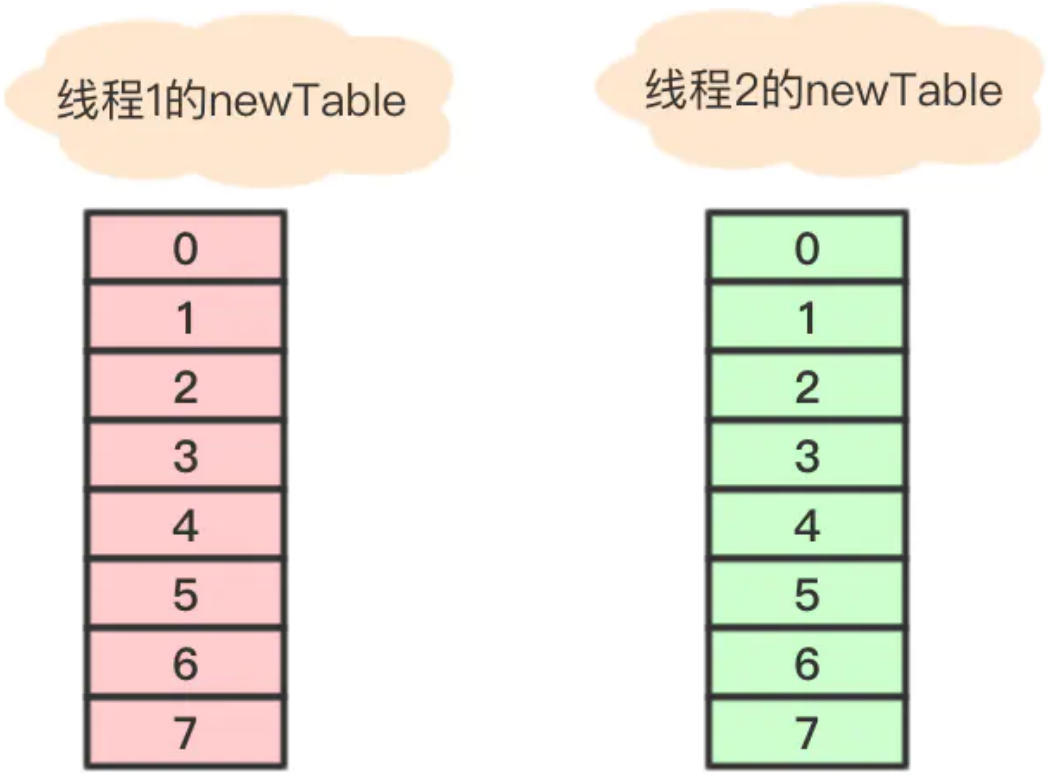
```
void transfer(Entry[] newTable, boolean rehash) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            if (rehash) {
                e.hash = null == e.key ? 0 : hash(e.key);
            }
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
```

```
}
```

以上是节点移动的相关逻辑。



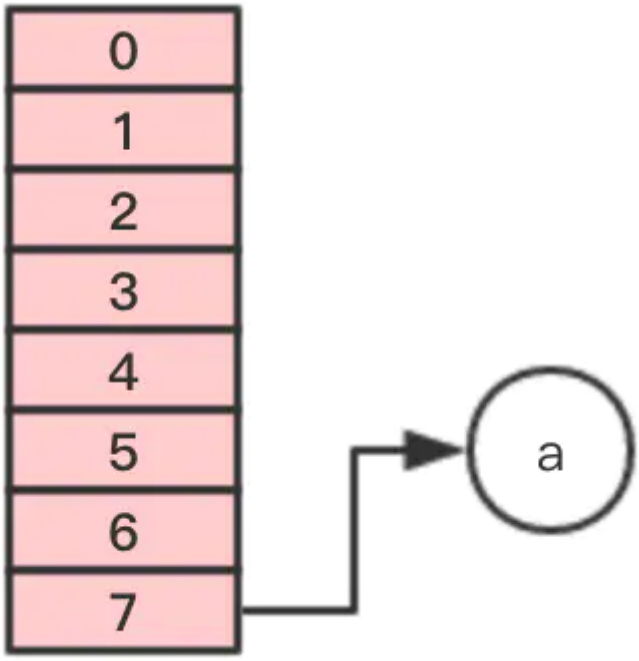
插入第4个节点时，发生rehash，假设现在有两个线程同时进行，线程1和线程2，两个线程都会新建新的数组。



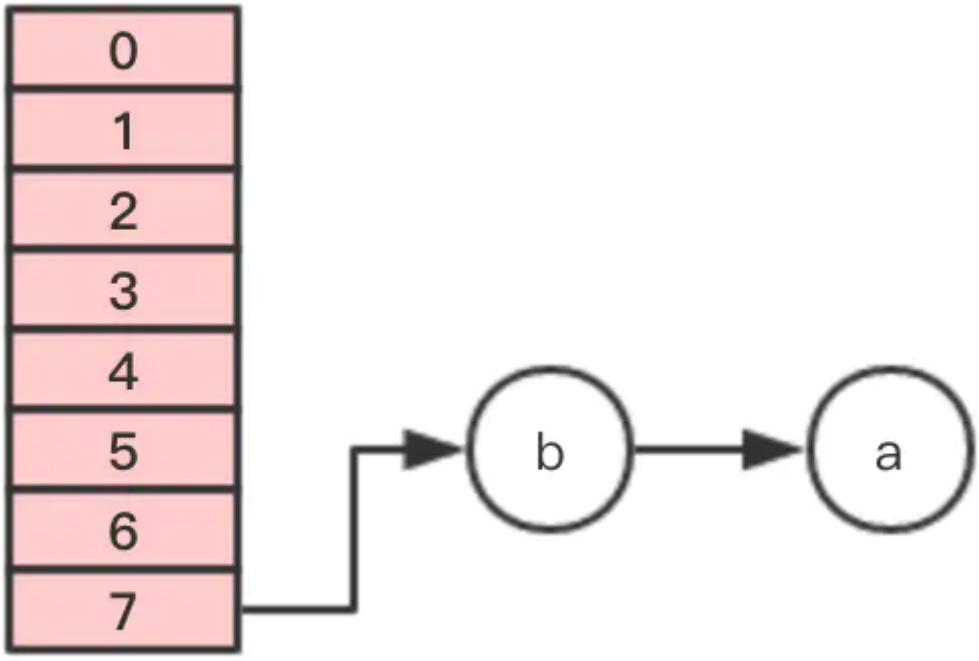
假设 **线程2** 在执行到`Entry<K,V> next = e.next;`之后，cpu时间片用完了，这时变量e指向节点a，变量next指向节点b。

线程1继续执行，很不巧，a、b、c节点rehash之后又是在同一个位置7，开始移动节点

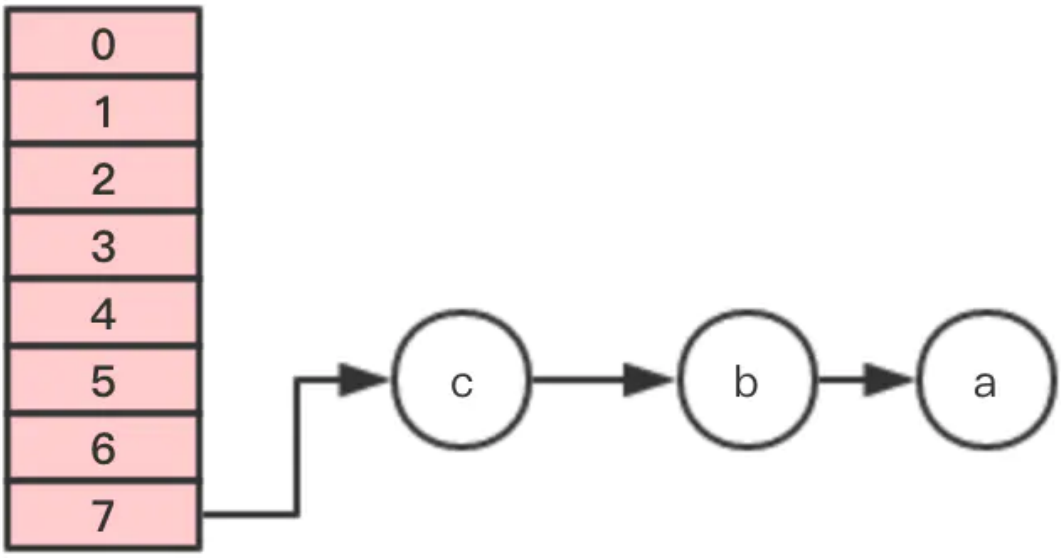
第一步，移动节点a



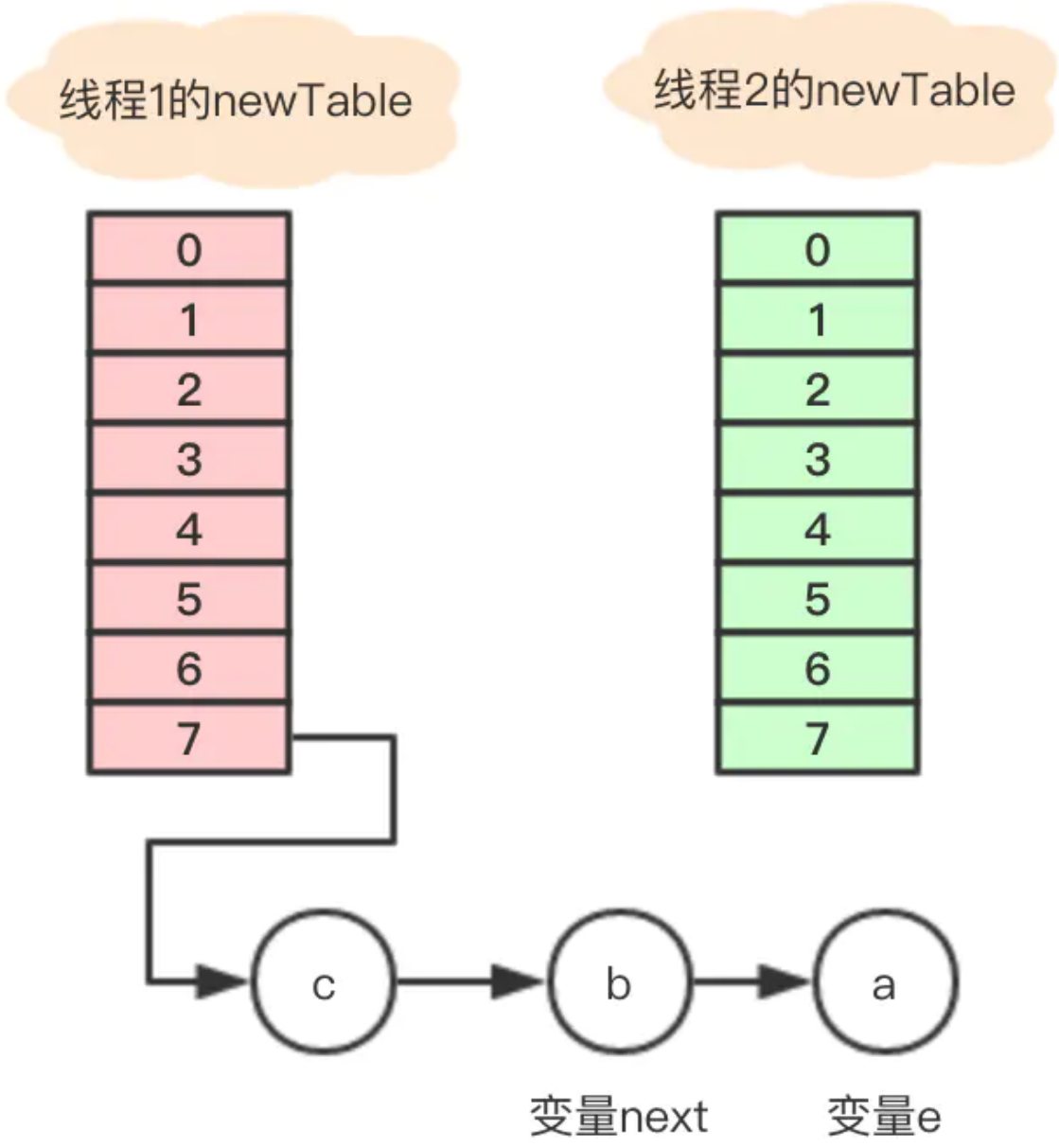
第二步，移动节点b



注意，这里的顺序是反过来的，继续移动节点c



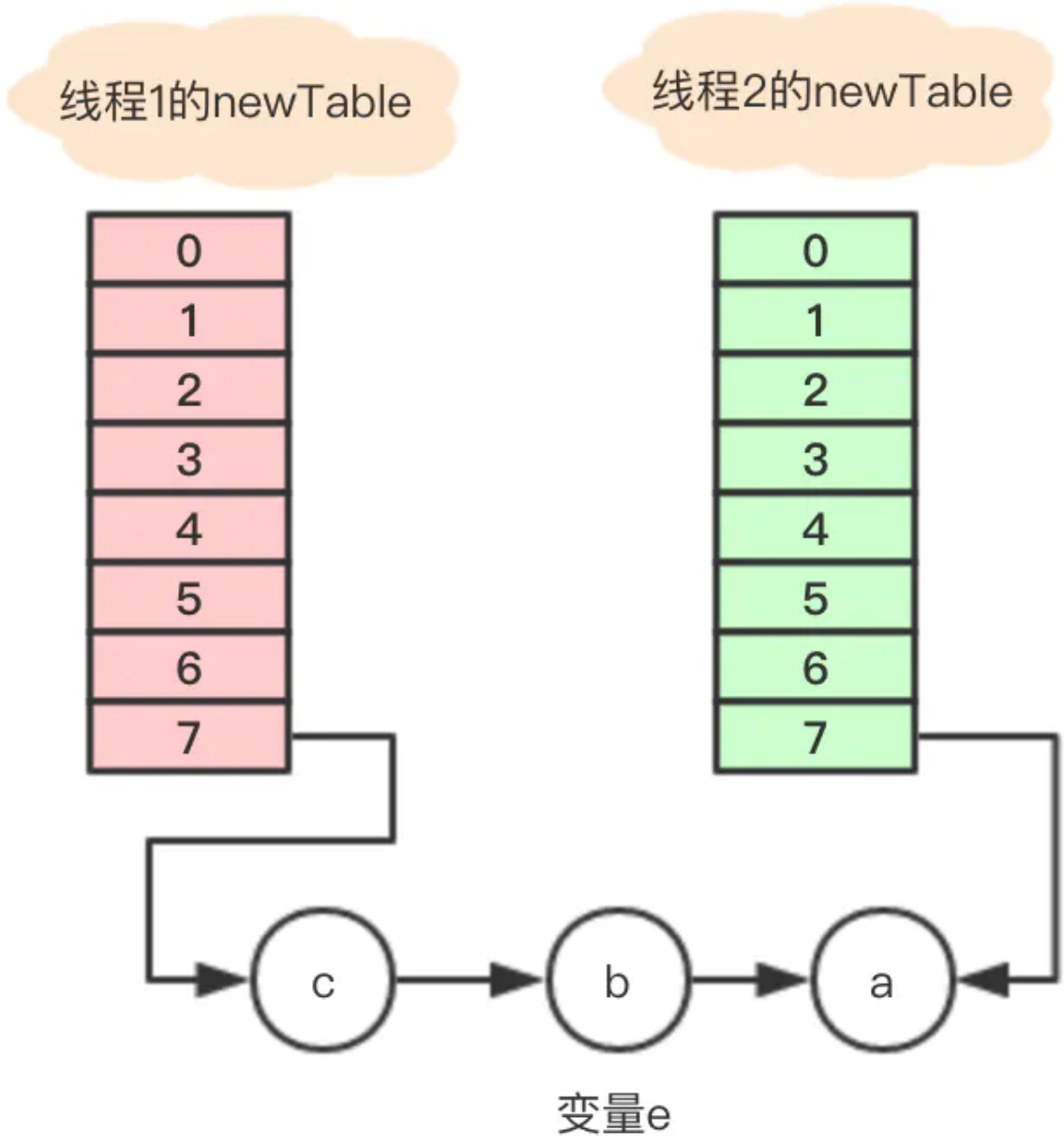
这个时候 **线程1** 的时间片用完，内部的table还没有设置成新的newTable， **线程2** 开始执行，这时内部的引用关系如下：



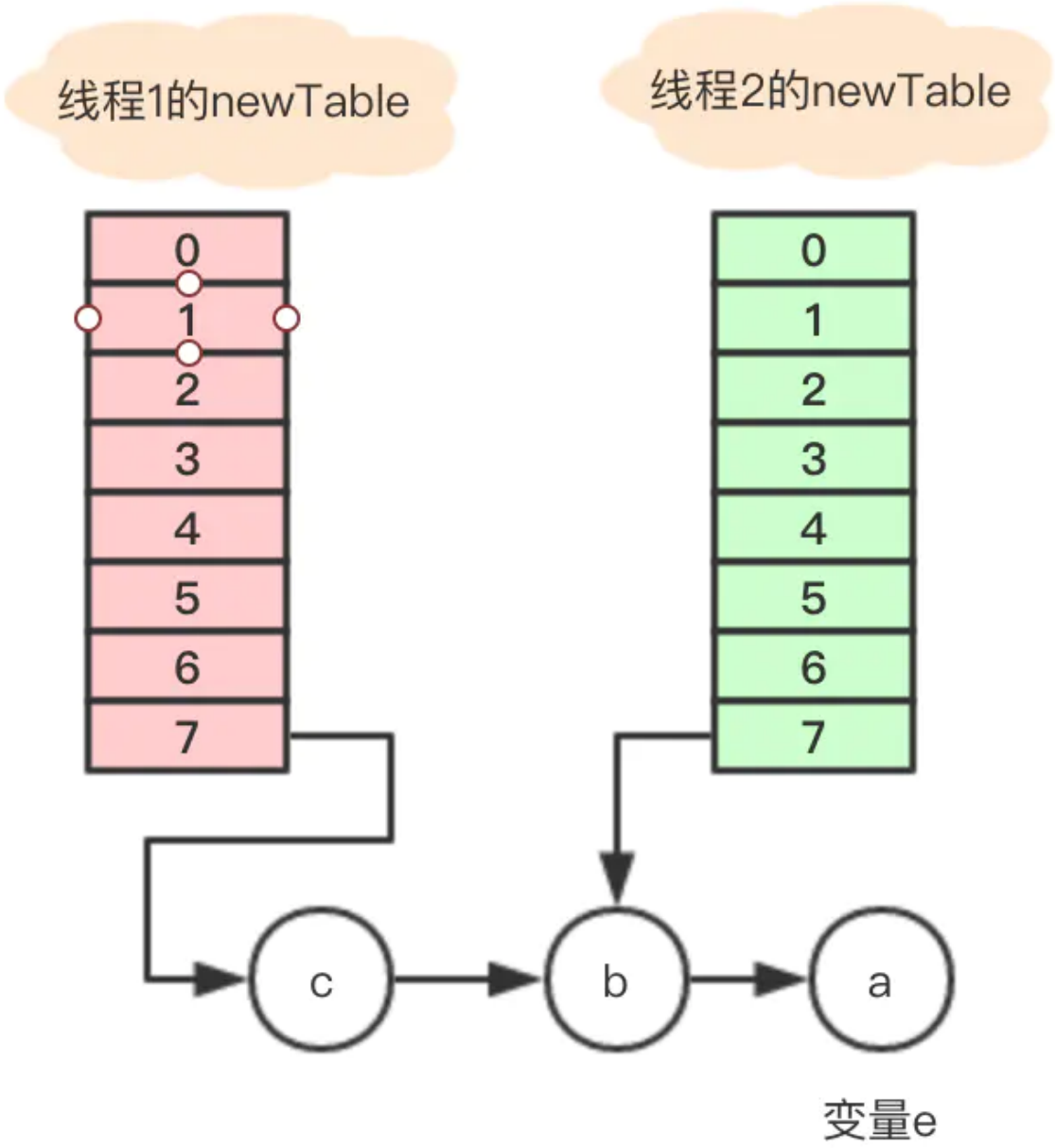
这时，在 **线程2** 中，变量e指向节点a，变量next指向节点b，开始执行循环体的剩余逻辑。

```
Entry<K,V> next = e.next;
int i = indexFor(e.hash, newCapacity);
e.next = newTable[i];
newTable[i] = e;
e = next;
```

执行之后的引用关系如下图



执行后，变量e指向节点b，因为e不是null，则继续执行循环体，执行后的引用关系



变量e又重新指回节点a，只能继续执行循环体，这里仔细分析下：

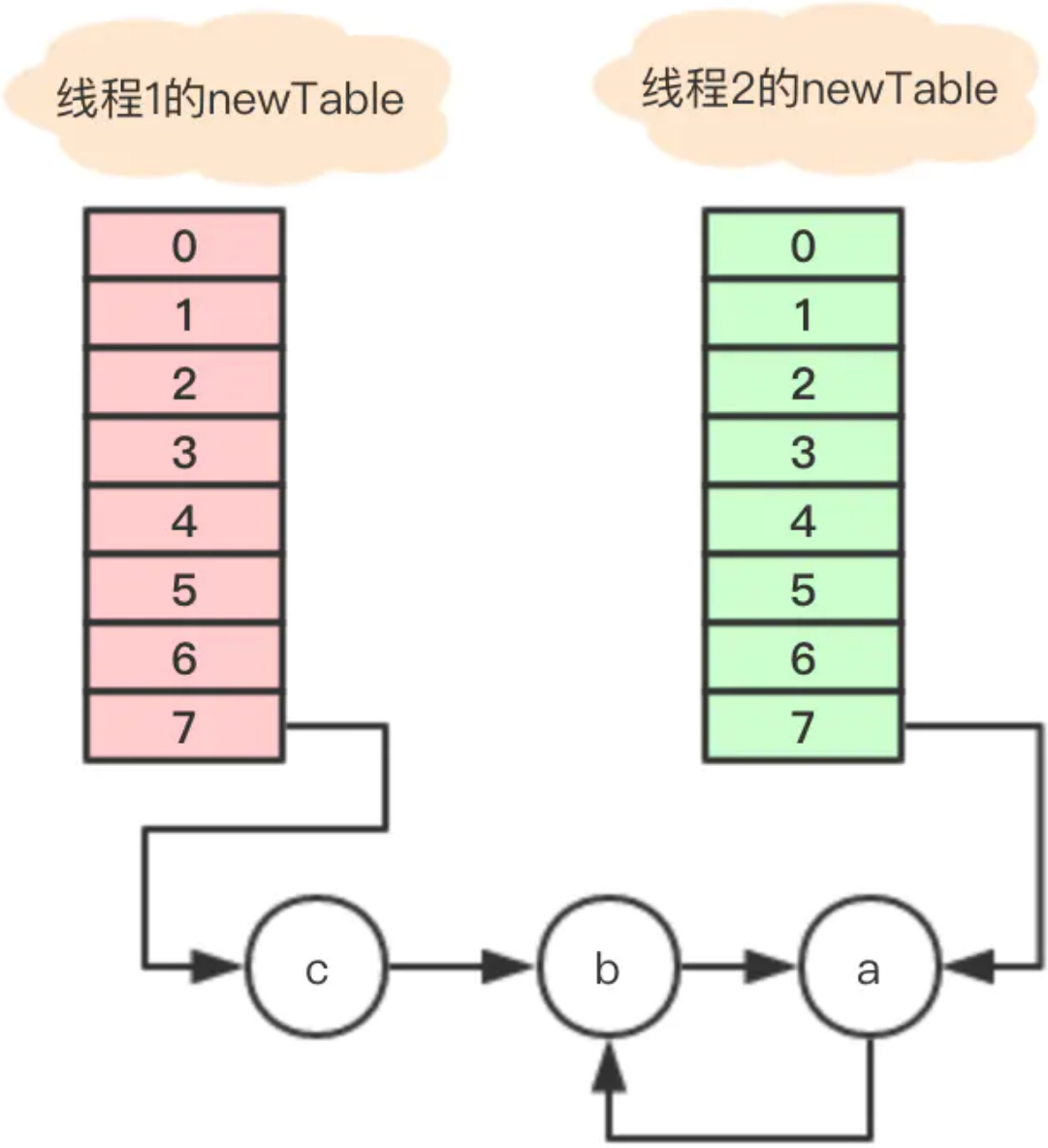
1、执行完 `Entry<K,V> next = e.next;`，目前节点a没有next，所以变量next指向null；

2、 `e.next = newTable[i];` 其中 newTable[i] 指向节点b，那就是把a的next指向了节点b，这样a和b就相互引用了，形成了一个环；

3、 `newTable[i] = e` 把节点a放到了数组i位置；

4、 `e = next;` 把变量e赋值为null，因为第一步中变量next就是指向null；

所以最终的引用关系是这样的：



节点a和b互相引用，形成了一个环，当在数组该位置get寻找对应的key时，就发生了死循环。

另外，如果线程2把newTable设置成到内部的table，节点c的数据就丢了，看来还有数据遗失的问题。

总结

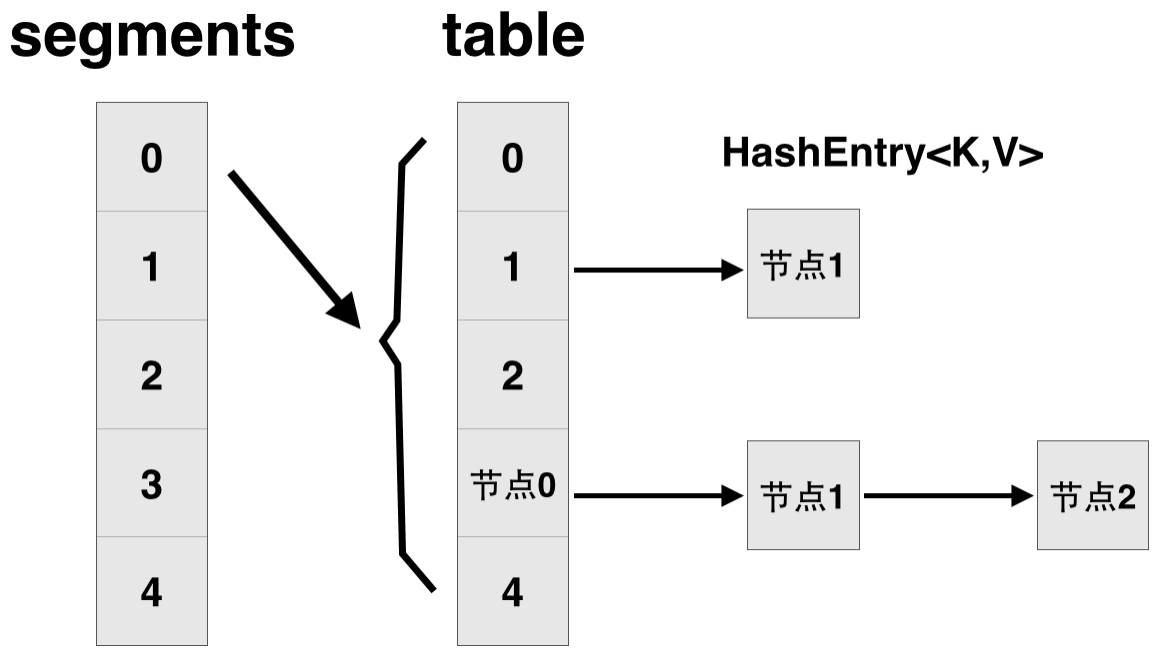
所以在并发的情况，发生扩容时，可能会产生循环链表，在执行get的时候，会触发死循环，引起CPU的100%问题，所以一定要避免在并发环境下使用HashMap。

曾经有人把这个问题报给了Sun，不过Sun不认为这是一个bug，因为在HashMap本来就不支持多线程使用，要并发就用ConcurrentHashmap。

ConcurrentHashMap在jdk1.8中的改进

一、简单回顾ConcurrentHashMap在jdk1.7中的设计

先简单看下ConcurrentHashMap类在jdk1.7中的设计，其基本结构如图所示：



每一个segment都是一个HashEntry<K,V>[] table， table中的每一个元素本质上都是一个HashEntry的单向队列。比如table[3]为首节点，table[3]->next为节点1，之后为节点2，依次类推。

```
public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
        implements ConcurrentMap<K, V>, Serializable {

    // 将整个hashmap分成几个小的map，每个segment都是一个锁；与hashtable相比，这么设计的目的是对于put, remove等操作，可以减少并发冲突，对
    // 不属于同一个片段的节点可以并发操作，大大提高了性能
    final Segment<K,V>[] segments;

    // 本质上Segment类就是一个小的hashmap，里面table数组存储了各个节点的数据，继承了ReentrantLock，可以作为互拆锁使用
    static final class Segment<K,V> extends ReentrantLock implements Serializable {
        transient volatile HashEntry<K,V>[] table;
        transient int count;
    }

    // 基本节点，存储Key， Value值
    static final class HashEntry<K,V> {
        final int hash;
        final K key;
        volatile V value;
        volatile HashEntry<K,V> next;
    }
}
```

二、在jdk1.8中主要做了2方面的改进

改进一：取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

改进二：将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。

为了说明以上2个改动，看一下put操作是如何实现的。

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 如果table为空，初始化；否则，根据hash值计算得到数组索引i，如果tab[i]为空，直接新建节点Node即可。注：tab[i]实质为链表或者红黑树的首节点。
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        // 如果tab[i]不为空并且hash值为MOVED，说明该链表正在进行transfer操作，返回扩容完成后的table。
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            // 针对首个节点进行加锁操作，而不是segment，进一步减少线程冲突
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            // 如果在链表中找到值为key的节点e，直接设置e.val = value即可。
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            // 如果没有找到值为key的节点，直接新建Node并加入链表即可。
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
```

```
                        }
                    }
                    // 如果首节点为TreeBin类型，说明为红黑树结构，执行putTreeVal操作。
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            if (binCount != 0) {
                // 如果节点数>＝8，那么转换链表结构为红黑树结构。
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    // 计数增加1，有可能触发transfer操作(扩容)。
    addCount(1L, binCount);
    return null;
}
```

另外，在其他方面也有一些小的改进，比如新增字段 transient volatile CounterCell[] counterCells; 可方便的计算hashmap中所有元素的个数，性能大大优于jdk1.7中的size()方法。

三、ConcurrentHashMap jdk1.7、jdk1.8性能比较

测试程序如下：

```
public class CompareConcurrentHashMap {
    private static ConcurrentHashMap<String, Integer> map = new
ConcurrentHashMap<String, Integer>(40000);


    public static void putPerformance(int index, int num) {
        for (int i = index; i < (num + index) ; i++)
            map.put(String.valueOf(i), i);
    }
public static void getPerformance2() {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 400000; i++)
            map.get(String.valueOf(i));
        long end = System.currentTimeMillis();
        System.out.println("get: it costs " + (end - start) + " ms");
    }
```

```
    public static void main(String[] args) throws InterruptedException {
        long start = System.currentTimeMillis();
        final CountDownLatch cdLatch = new CountDownLatch(4);
        for (int i = 0; i < 4; i++) {
            final int finalI = i;
            new Thread(new Runnable() {
                public void run() {
                    CompareConcurrentHashMap.putPerformance(100000 * finalI,
100000);
                    cdLatch.countDown();
                }
            }).start();
        }
        cdLatch.await();
        long end = System.currentTimeMillis();
        System.out.println("put: it costs " + (end - start) + " ms");
        CompareConcurrentHashMap.getPerformance2();
    }
}
```

程序运行多次后取平均值，结果如下：

	put	get
jdk1.7	330ms	242ms
jdk1.8	345ms	120ms

四、Collections.synchronizedList和CopyOnWriteArrayList性能分析

CopyOnWriteArrayList在线程对其进行变更操作的时候，会拷贝一个新的数组以存放新的字段，因此写操作性能很差；而Collections.synchronizedList读操作采用了synchronized，因此读性能较差。以下为测试程序：

```
public class App {
    private static List<String> arrayList = Collections.synchronizedList(new
ArrayList<String>());
    private static List<String> copyOnWriteArrayList = new
CopyOnWriteArrayList<String>();
    private static CountDownLatch cdl1 = new CountDownLatch(2);
    private static CountDownLatch cdl2 = new CountDownLatch(2);
    private static CountDownLatch cdl3 = new CountDownLatch(2);
    private static CountDownLatch cdl4 = new CountDownLatch(2);

    static class Thread1 extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++)
```

```
                arrayList.add(String.valueOf(i));
            cdl1.countDown();
        }
    }

    static class Thread2 extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++)
                copyOnWriteArrayList.add(String.valueOf(i));
            cdl2.countDown();
        }
    }

    static class Thread3 extends Thread1 {
        @Override
        public void run() {
            int size = arrayList.size();
            for (int i = 0; i < size; i++)
                arrayList.get(i);
            cdl3.countDown();
        }
    }

    static class Thread4 extends Thread1 {
        @Override
        public void run() {
            int size = copyOnWriteArrayList.size();
            for (int i = 0; i < size; i++)
                copyOnWriteArrayList.get(i);
            cdl4.countDown();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        long start1 = System.currentTimeMillis();
        new Thread1().start();
        new Thread1().start();
        cdl1.await();
        System.out.println("arrayList add: " + (System.currentTimeMillis() -
start1));

        long start2 = System.currentTimeMillis();
        new Thread2().start();
        new Thread2().start();
        cdl2.await();
        System.out.println("copyOnWriteArrayList add: " +
(System.currentTimeMillis() - start2));

        long start3 = System.currentTimeMillis();
        new Thread3().start();
        new Thread3().start();
        cdl3.await();
        System.out.println("arrayList get: " + (System.currentTimeMillis() -
start3));

        long start4 = System.currentTimeMillis();
        new Thread4().start();
```

```
        new Thread4().start();
        cdl4.await();
        System.out.println("copyOnWriteArrayList get: " +
(System.currentTimeMillis() - start4));
    }
}
```

结果如下：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java ...
arrayList add: 18
copyOnWriteArrayList add: 356
arrayList get: 9
copyOnWriteArrayList get: 5
```

谈谈ConcurrentHashMap1.7和1.8的不同实现

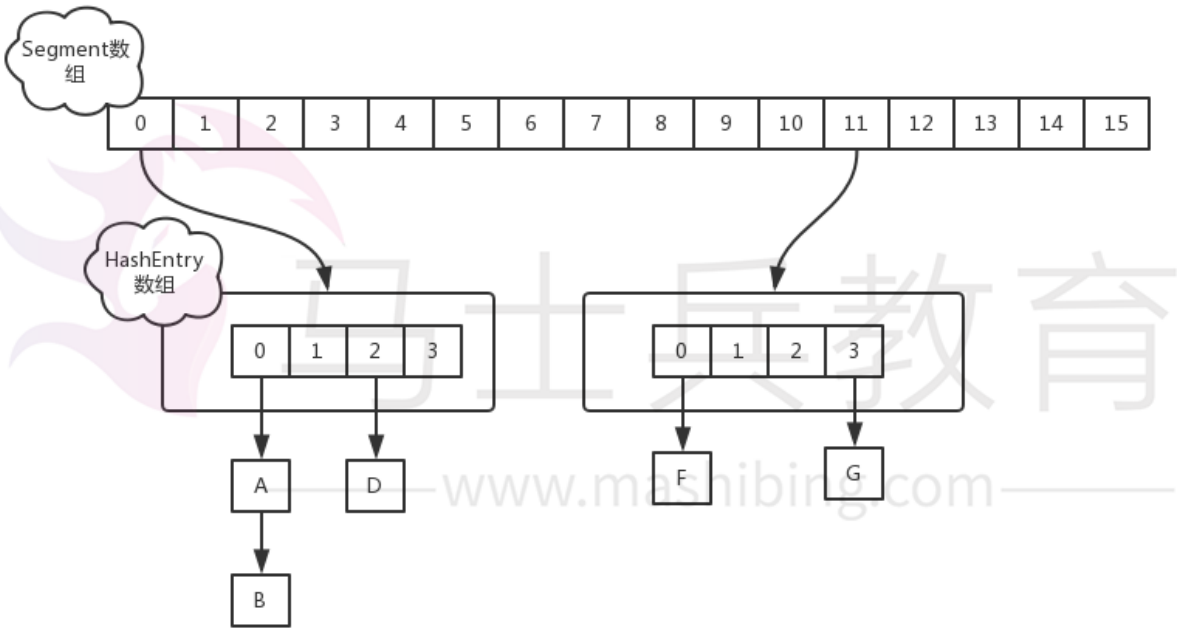
ConcurrentHashMap

在多线程环境下，使用`HashMap`进行`put`操作时存在丢失数据的情况，为了避免这种bug的隐患，强烈建议使用`ConcurrentHashMap`代替`HashMap`，为了对更深入的了解，本文将对JDK1.7和1.8的不同实现进行分析。

JDK1.7

数据结构

jdk1.7中采用`Segment` + `HashEntry`的方式进行实现，结构如下：



`ConcurrentHashMap`初始化时，计算出`Segment`数组的大小`ssize`和每个`Segment`中`HashEntry`数组的大小`cap`，并初始化`Segment`数组的第一个元素；其中`ssize`大小为2的幂次方，默认为16，`cap`大小也是2的幂次方，最小值为2，最终结果根据根据初始化容量`initialCapacity`进行计算，计算过程如下：

```
if (c * ssize < initialCapacity)
    ++c;
int cap = MIN_SEGMENT_TABLE_CAPACITY;
while (cap < c)
    cap <<= 1;
```

其中`Segment`在实现上继承了`ReentrantLock`，这样就自带了锁的功能。

put实现

当执行`put`方法插入数据时，根据key的hash值，在`Segment`数组中找到相应的位置，如果相应位置的`Segment`还未初始化，则通过CAS进行赋值，接着执行`Segment`对象的`put`方法通过加锁机制插入数据，实现如下：

场景：线程A和线程B同时执行相同`Segment`对象的`put`方法

1、线程A执行`tryLock()`方法成功获取锁，则把`HashEntry`对象插入到相应的位置；
2、线程B获取锁失败，则执行`scanAndLockForPut()`方法，在`scanAndLockForPut`方法中，会通过重复执行`tryLock()`方法尝试获取锁，在多处理器环境下，重复次数为64，单处理器重复次数为1，当执行`tryLock()`方法的次数超过上限时，则执行`lock()`方法挂起线程B；
3、当线程A执行完插入操作时，会通过`unlock()`方法释放锁，接着唤醒线程B继续执行；

size实现

因为`ConcurrentHashMap`是可以并发插入数据的，所以在准确计算元素时存在一定的难度，一般的思路是统计每个`Segment`对象中的元素个数，然后进行累加，但是这种方式计算出来的结果并不一样的准确的，因为在计算后面几个`Segment`的元素个数时，已经计算过的`Segment`同时可能有数据的插入或则删除，在1.7的实现中，采用了如下方式：

```
try {
    for (;;) {
        if (retries++ == RETRIES_BEFORE_LOCK) {
            for (int j = 0; j < segments.length; ++j)
                ensureSegment(j).lock(); // force creation
        }
        sum = 0L;
        size = 0;
        overflow = false;
        for (int j = 0; j < segments.length; ++j) {
            Segment<K,V> seg = segmentAt(segments, j);
            if (seg != null) {
                sum += seg.modCount;
                int c = seg.count;
                if (c < 0 || (size += c) < 0)
                    overflow = true;
            }
        }
        if (sum == last)
            break;
        last = sum;
    }
} finally {
    if (retries > RETRIES_BEFORE_LOCK) {
```

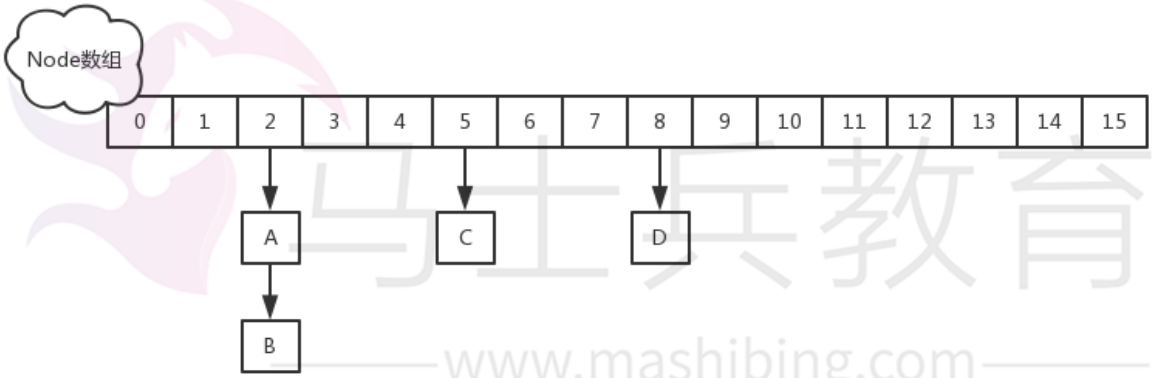
```
        for (int j = 0; j < segments.length; ++j)
            segmentAt(segments, j).unlock();
    }
}
```

先采用不加锁的方式，连续计算元素的个数，最多计算3次：
1、如果前后两次计算结果相同，则说明计算出来的元素个数是准确的；
2、如果前后两次计算结果都不同，则给每个`Segment`进行加锁，再计算一次元素的个数；

JDK1.8

数据结构

1.8中放弃了`Segment`臃肿的设计，取而代之的是采用`Node` + `CAS` + `Synchronized`来保证并发安全进行实现，结构如下：



马士兵教育
www.mashibing.com

只有在执行第一次`put`方法时才会调用`initTable()`初始化`Node`数组，实现如下：

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

put实现

当执行`put`方法插入数据时，根据key的hash值，在`Node`数组中找到相应的位置，实现如下：

1、如果相应位置的`Node`还未初始化，则通过CAS插入相应的数据；

```
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
        break;                   // no lock when adding to empty bin
}
```

2、如果相应位置的`Node`不为空，且当前该节点不处于移动状态，则对该节点加`synchronized`锁，如果该节点的`hash`不小于0，则遍历链表更新节点或插入新节点；

```
if (fh >= 0) {
    binCount = 1;
    for (Node<K,V> e = f;; ++binCount) {
        K ek;
        if (e.hash == hash &&
            ((ek = e.key) == key ||
             (ek != null && key.equals(ek)))) {
            oldVal = e.val;
            if (!onlyIfAbsent)
                e.val = value;
            break;
        }
        Node<K,V> pred = e;
        if ((e = e.next) == null) {
            pred.next = new Node<K,V>(hash, key, value, null);
            break;
        }
    }
}
```

3、如果该节点是`TreeBin`类型的节点，说明是红黑树结构，则通过`putTreeVal`方法往红黑树中插入节点；

```
else if (f instanceof TreeBin) {
    Node<K,V> p;
    binCount = 2;
    if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key, value)) != null) {
        oldVal = p.val;
        if (!onlyIfAbsent)
            p.val = value;
    }
}
```

4、如果`binCount`不为0，说明`put`操作对数据产生了影响，如果当前链表的个数达到8个，则通过`treeifyBin`方法转化为红黑树，如果`oldVal`不为空，说明是一次更新操作，没有对元素个数产生影响，则直接返回旧值；

```
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
```

5、如果插入的是一个新节点，则执行`addCount()`方法尝试更新元素个数`baseCount`；

size实现

1.8中使用一个`volatile`类型的变量`baseCount`记录元素的个数，当插入新数据或则删除数据时，会通过`addCount()`方法更新`baseCount`，实现如下：

```
if ((as = counterCells) != null ||
    !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
    CounterCell a; long v; int m;
    boolean uncontended = true;
    if (as == null || (m = as.length - 1) < 0 ||
        (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
        !(uncontended =
          U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
        fullAddCount(x, uncontended);
        return;
    }
    if (check <= 1)
        return;
    s = sumCount();
}
```

1、初始化时`counterCells`为空，在并发量很高时，如果存在两个线程同时执行`CAS`修改`baseCount`值，则失败的线程会继续执行方法体中的逻辑，使用`CounterCell`记录元素个数的变化；

2、如果`CounterCell`数组`counterCells`为空，调用`fullAddCount()`方法进行初始化，并插入对应的记录数，通过`CAS`设置cellsBusy字段，只有设置成功的线程才能初始化`CounterCell`数组，实现如下：

```
else if (cellsBusy == 0 && counterCells == as &&
         U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
    boolean init = false;
    try {                           // Initialize table
        if (counterCells == as) {
            CounterCell[] rs = new CounterCell[2];
            rs[h & 1] = new CounterCell(x);
            counterCells = rs;
            init = true;
        }
    } finally {
        cellsBusy = 0;
    }
    if (init)
        break;
```

```
}
```

3、如果通过 CAS 设置cellsBusy字段失败的话，则继续尝试通过 CAS 修改 baseCount 字段，如果修改 baseCount 字段成功的话，就退出循环，否则继续循环插入 CounterCell 对象；

```
else if (U.compareAndSwapLong(this, BASECOUNT, v = baseCount, v + x))
    break;
```

所以在1.8中的 size 实现比1.7简单多，因为元素个数保存 baseCount 中，部分元素的变化个数保存在 CounterCell 数组中，实现如下：

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}

final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

通过累加 baseCount 和 CounterCell 数组中的数量，即可得到元素的总个数；

深入分析ConcurrentHashMap1.8的扩容实现

ConcurrentHashMap相关的文章写了不少，有个遗留问题一直没有分析，也被好多人请教过，被搁置在一旁，即如何在并发的情况下实现数组的扩容。

什么情况会触发扩容

当往hashMap中成功插入一个key/value节点时，有可能触发扩容动作：
1、如果新增节点之后，所在链表的元素个数达到了阈值 **8**，则会调用 treeifyBin 方法把链表转换成红黑树，不过在结构转换之前，会对数组长度进行判断，实现如下：

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

如果数组长度n小于阈值`MIN_TREEIFY_CAPACITY`，默认是64，则会调用`tryPresize`方法把数组长度扩大到原来的两倍，并触发`transfer`方法，重新调整节点的位置。

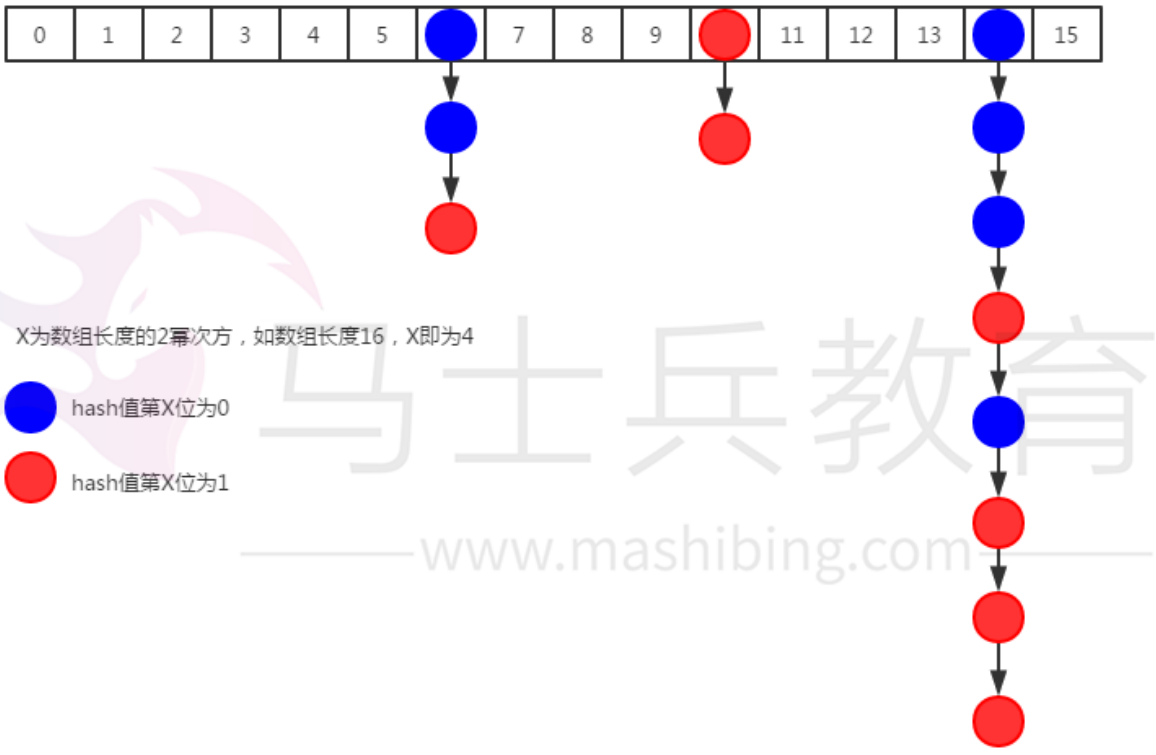
```
else if (tab == table) {
    int rs = resizeStamp(n);
    if (sc < 0) {
        Node<K,V>[] nt;
        if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
            sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
            transferIndex <= 0)
            break;
        if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
            transfer(tab, nt);
    }
    else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                 (rs << RESIZE_STAMP_SHIFT) + 2))
        transfer(tab, null);
}
```

2、新增节点之后，会调用`addCount`方法记录元素个数，并检查是否需要进行扩容，当数组元素个数达到阈值时，会触发`transfer`方法，重新调整节点的位置。

```
if (check >= 0) {
    Node<K,V>[] tab, nt; int n, sc;
    while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
           (n = tab.length) < MAXIMUM_CAPACITY) {
        int rs = resizeStamp(n);
        if (sc < 0) {
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                transferIndex <= 0)
                break;
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                transfer(tab, nt);
        }
        else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                     (rs << RESIZE_STAMP_SHIFT) + 2))
            transfer(tab, null);
        s = sumCount();
    }
}
```

transfer实现

`transfer`方法实现了在并发的情况下，高效的从原始组数往新数组中移动元素，假设扩容之前节点的分布如下，这里区分蓝色节点和红色节点，是为了后续更好的分析：



在上图中，第14个槽位插入新节点之后，链表元素个数已经达到了8，且数组长度为16，优先通过扩容来缓解链表过长的问题，实现如下：

1、根据当前数组长度n，新建一个两倍长度的数组`nextTable`；

```
if (nextTab == null) {            // initiating
    try {
        /unchecked/
        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
        nextTab = nt;
    } catch (Throwable ex) {      // try to cope with OOME
        sizeCtl = Integer.MAX_VALUE;
        return;
    }
    nextTable = nextTab;
    transferIndex = n;
}
```

2、初始化`ForwardingNode`节点，其中保存了新数组`nextTable`的引用，在处理完每个槽位的节点之后当做占位节点，表示该槽位已经处理过了；

```
int nextn = nextTab.length;
ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
boolean advance = true;
boolean finishing = false; // to ensure sweep before committing nextTab
```

3、通过`for`自循环处理每个槽位中的链表元素，默认`advace`为真，通过CAS设置`transferIndex`属性值，并初始化`i`和`bound`值，`i`指当前处理的槽位序号，`bound`指需要处理的槽位边界，先处理槽位15的节点；

```
for (int i = 0, bound = 0;;) {
    Node<K,V> f; int fh;
    while (advance) {
        int nextIndex, nextBound;
        if (--i >= bound || finishing)
            advance = false;
        else if ((nextIndex = transferIndex) <= 0) {
            i = -1;
            advance = false;
        }
        else if (U.compareAndSwapInt
                 (this, TRANSFERINDEX, nextIndex,
                  nextBound = (nextIndex > stride ?
                               nextIndex - stride : 0))) {
            bound = nextBound;
            i = nextIndex - 1;
            advance = false;
        }
    }
```

4、在当前假设条件下，槽位15中没有节点，则通过CAS插入在第二步中初始化的`ForwardingNode`节点，用于告诉其它线程该槽位已经处理过了；

```
else if ((f = tabAt(tab, i)) == null)
    advance = casTabAt(tab, i, null, fwd);
```

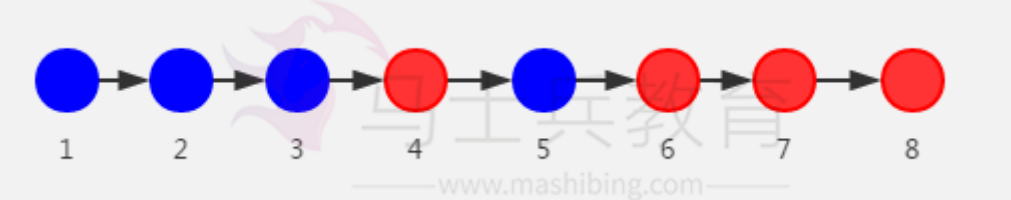
5、如果槽位15已经被线程A处理了，那么线程B处理到这个节点时，取到该节点的hash值应该为`MOVED`，值为`-1`，则直接跳过，继续处理下一个槽位14的节点；

```
else if ((fh = f.hash) == MOVED)
    advance = true; // already processed
```

6、处理槽位14的节点，是一个链表结构，先定义两个变量节点`ln`和`hn`，按我的理解应该是`lowNode`和`highNode`，分别保存hash值的第X位为0和1的节点，具体实现如下：

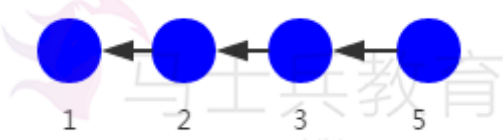
```
synchronized (f) {
    if (tabAt(tab, i) == f) {
        Node<K,V> ln, hn;
        if (fh >= 0) {
            int runBit = fh & n;
            Node<K,V> lastRun = f;
            for (Node<K,V> p = f.next; p != null; p = p.next) {
                int b = p.hash & n;
                if (b != runBit) {
                    runBit = b;
                    lastRun = p;
                }
            }
            if (runBit == 0) {
                ln = lastRun;
                hn = null;
            }
            else {
                hn = lastRun;
                ln = null;
            }
            for (Node<K,V> p = f; p != lastRun; p = p.next) {
                int ph = p.hash; K pk = p.key; V pv = p.val;
                if ((ph & n) == 0)
                    ln = new Node<K,V>(ph, pk, pv, ln);
                else
                    hn = new Node<K,V>(ph, pk, pv, hn);
            }
            setTabAt(nextTab, i, ln);
            setTabAt(nextTab, i + n, hn);
            setTabAt(tab, i, fwd);
            advance = true;
        }
```

使用`fn&n`可以快速把链表中的元素区分成两类，A类是hash值的第X位为0，B类是hash值的第X位为1，并通过`lastRun`记录最后需要处理的节点，A类和B类节点可以分散到新数组的槽位14和30中，在原数组的槽位14中，蓝色节点第X为0，红色节点第X为1，把链表拉平显示如下：

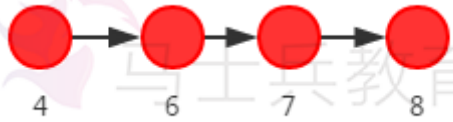


1、通过遍历链表，记录`runBit`和`lastRun`，分别为1和节点6，所以设置`hn`为节点6，`ln`为null；
2、重新遍历链表，以`lastRun`节点为终止条件，根据第X位的值分别构造ln链表和hn链表：

ln链：和原来链表相比，顺序已经不一样了



hn链：



通过CAS把ln链表设置到新数组的i位置，hn链表设置到i+n的位置；

7、如果该槽位是红黑树结构，则构造树节点 `lo` 和 `hi`，遍历红黑树中的节点，同样根据 `hash&n` 算法，把节点分为两类，分别插入到 `lo` 和 `hi` 为头的链表中，根据 `lo` 和 `hi` 链表中的元素个数分别生成 `ln` 和 `hn` 节点，其中 `ln` 节点的生成逻辑如下：

（1）如果 `lo` 链表的元素个数小于等于 `UNTREEIFY_THRESHOLD`，默认为6，则通过 `untreeify` 方法把树节点链表转化成普通节点链表；

（2）否则判断 `hi` 链表中的元素个数是否等于0：如果等于0，表示 `lo` 链表中包含了所有原始节点，则设置原始红黑树给 `ln`，否则根据 `lo` 链表重新构造红黑树。

```
else if (f instanceof TreeBin) {
    TreeBin<K,V> t = (TreeBin<K,V>)f;
    TreeNode<K,V> lo = null, loTail = null;
    TreeNode<K,V> hi = null, hiTail = null;
    int lc = 0, hc = 0;
    for (Node<K,V> e = t.first; e != null; e = e.next) {
        int h = e.hash;
        TreeNode<K,V> p = new TreeNode<K,V>
            (h, e.key, e.val, null, null);
        if ((h & n) == 0) {
            if ((p.prev = loTail) == null)
                lo = p;
            else
                loTail.next = p;
            loTail = p;
            ++lc;
        }
        else {
            if ((p.prev = hiTail) == null)
                hi = p;
            else
                hiTail.next = p;
            hiTail = p;
            ++hc;
        }
    }
    ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
        (hc != 0) ? new TreeBin<K,V>(lo) : t;
    hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
        (lc != 0) ? new TreeBin<K,V>(hi) : t;
    setTabAt(nextTab, i, ln);
    setTabAt(nextTab, i + n, hn);
    setTabAt(tab, i, fwd);
    advance = true;
}
```

最后，同样的通过CAS把 `ln` 设置到新数组的 `i` 位置，`hn` 设置到 `i+n` 位置。

深入浅出ConcurrentHashMap1.8

前言

HashMap是我们平时开发过程中用的比较多的集合，但它是非线程安全的，在涉及到多线程并发的情况，进行get操作有可能会引起死循环，导致CPU利用率接近100%。

```
final HashMap<String, String> map = new HashMap<String, String>(2);
for (int i = 0; i < 10000; i++) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            map.put(UUID.randomUUID().toString(), "");
        }
    }).start();
}
```

解决方案有Hashtable和Collections.synchronizedMap(hashMap)，不过这两个方案基本上是对读写进行加锁操作，一个线程在读写元素，其余线程必须等待，性能可想而知。

所以，Doug Lea给我们带来了并发安全的ConcurrentHashMap，它的实现是依赖于 Java 内存模型，所以我们在了解 ConcurrentHashMap 的之前必须了解一些底层的知识：

1. java内存模型
2. java中的Unsafe
3. java中的CAS
4. 深入浅出java同步器
5. 深入浅出ReentrantLock

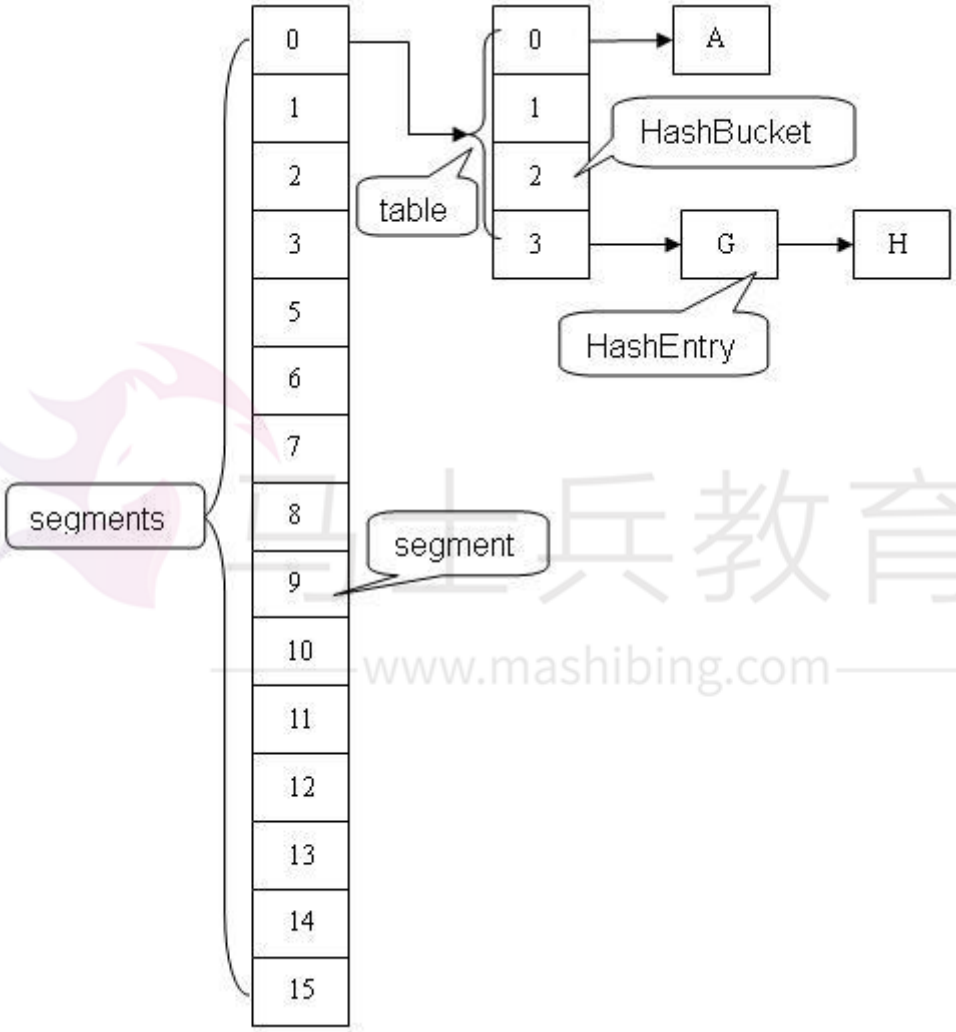
本文源码是JDK8的版本，与之前的版本有较大差异。

JDK1.7分析

ConcurrentHashMap采用 分段锁的机制，实现并发的更新操作，底层采用数组+链表的存储结构。
其包含两个核心静态内部类 Segment和HashEntry。

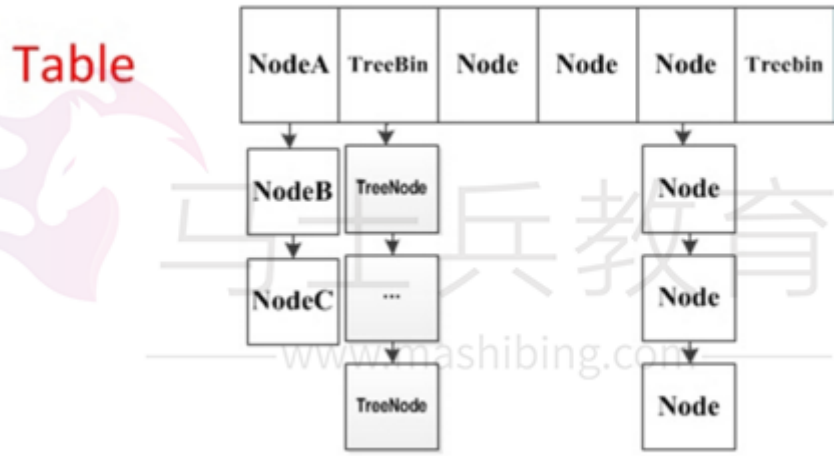
1. Segment继承ReentrantLock用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶。
2. HashEntry 用来封装映射表的键 / 值对；
3. 每个桶是由若干个 HashEntry 对象链接起来的链表。

一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组，下面我们通过一个图来演示一下 ConcurrentHashMap 的结构：



JDK1.8分析

1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层采用数组+链表+红黑树的存储结构。



重要概念

在开始之前，有些重要的概念需要介绍一下：

1. **table**：默认为null，初始化发生在第一次插入操作，默认大小为16的数组，用来存储Node节点数据，扩容时大小总是2的幂次方。
2. **nextTable**：默认为null，扩容时新生成的数组，其大小为原数组的两倍。

- **sizeCtl** ：默认为0，用来控制table的初始化和扩容操作，具体应用在后续会体现出来。
- **-1** 代表table正在初始化
- **-N** 表示有N-1个线程正在进行扩容操作
- 其余情况：
  1、如果table未初始化，表示table需要初始化的大小。
  2、如果table初始化完成，表示table的容量，默认是table大小的0.75倍，居然用这个公式算0.75（n - (n >>> 2)）。
- **Node**：保存key，value及key的hash值的数据结构。

```
class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
    ... 省略部分代码
}
```

其中value和next都用volatile修饰，保证并发的可见性。

- **ForwardingNode**：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用。

```
final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
}
```

只有table发生扩容的时候，ForwardingNode才会发挥作用，作为一个占位符放在table中表示当前节点为null或则已经被移动。

实例初始化

实例化ConcurrentHashMap时带参数时，会根据参数调整table的大小，假设参数为100，最终会调整成256，确保table的大小总是2的幂次方，算法如下：

```
ConcurrentHashMap<String, String> hashMap = new ConcurrentHashMap<>(100);
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

注意，ConcurrentHashMap在构造函数中只会初始化sizeCtl值，并不会直接初始化table，而是延缓到第一次put操作。

table初始化

前面已经提到过，table初始化操作会延缓到第一次put行为。但是put是可以并发执行的，Doug Lea是如何实现table只初始化一次的？让我们来看看源码的实现。

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
//如果一个线程发现sizeCtl<0，意味着另外的线程执行CAS操作成功，当前线程只需要让出cpu时间片
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

sizeCtl默认为0，如果ConcurrentHashMap实例化时有传参数，sizeCtl会是一个2的幂次方的值。所以执行第一次put操作的线程会执行Unsafe.compareAndSwapInt方法修改sizeCtl为-1，有且只有一个线程能够修改成功，其它线程通过Thread.yield()让出CPU时间片等待table初始化完成。

put操作

假设table已经初始化完成，put操作采用CAS+synchronized实现并发插入或更新操作，具体实现如下。

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
```

```
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        ...省略部分代码
    }
    addCount(1L, binCount);
    return null;
}
```

1. hash算法

```
static final int spread(int h) {return (h ^ (h >>> 16)) & HASH_BITS;}
```

1. table中定位索引位置，n是table的大小

```
int index = (n - 1) & hash
```

1. 获取table中对应索引的元素f。
   Doug Lea采用Unsafe.getObjectVolatile来获取，也许有人质疑，直接table[index]不可以么，为什么要这么复杂?
   在java内存模型中，我们已经知道每个线程都有一个工作内存，里面存储着table的副本，虽然table是volatile修饰的，但不能保证线程每次都拿到table中的最新元素，
   Unsafe.getObjectVolatile可以直接获取指定内存的数据，保证了每次拿到数据都是最新的。
2. 如果f为null，说明table中这个位置第一次插入元素，利用Unsafe.compareAndSwapObject方法插入Node节点。

- 如果CAS成功，说明Node节点已经插入，随后addCount(1L, binCount)方法会检查当前容量是否需要进行扩容。
- 如果CAS失败，说明有其它线程提前插入了节点，自旋重新尝试在这个位置插入节点。

1. 如果f的hash值为-1，说明当前f是ForwardingNode节点，意味有其它线程正在扩容，则一起进行扩容操作。
2. 其余情况把新的Node节点按链表或红黑树的方式插入到合适的位置，这个过程采用同步内置锁实现并发，代码如下:

```
synchronized (f) {
    if (tabAt(tab, i) == f) {
        if (fh >= 0) {
            binCount = 1;
            for (Node<K,V> e = f;; ++binCount) {
                K ek;
```

```
                if (e.hash == hash &&
                    ((ek = e.key) == key ||
                     (ek != null && key.equals(ek)))) {
                    oldVal = e.val;
                    if (!onlyIfAbsent)
                        e.val = value;
                    break;
                }
                Node<K,V> pred = e;
                if ((e = e.next) == null) {
                    pred.next = new Node<K,V>(hash, key,
                                              value, null);
                    break;
                }
            }
        }
        else if (f instanceof TreeBin) {
            Node<K,V> p;
            binCount = 2;
            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                           value)) != null) {
                oldVal = p.val;
                if (!onlyIfAbsent)
                    p.val = value;
            }
        }
    }
}
```

在节点f上进行同步，节点插入之前，再次利用tabAt(tab, i) == f判断，防止被其它线程修改。

1. 如果f.hash >= 0，说明f是链表结构的头结点，遍历链表，如果找到对应的node节点，则修改value，否则在链表尾部加入节点。
2. 如果f是TreeBin类型节点，说明f是红黑树根节点，则在树结构上遍历元素，更新或增加节点。
3. 如果链表中节点数binCount >= TREEIFY_THRESHOLD(默认是8)，则把链表转化为红黑树结构。

table扩容

当table容量不足的时候，即table的元素数量达到容量阈值sizeCtl，需要对table进行扩容。
整个扩容分为两部分：

1. 构建一个nextTable，大小为table的两倍。
2. 把table的数据复制到nextTable中。

这两个过程在单线程下实现很简单，但是ConcurrentHashMap是支持并发插入的，扩容操作自然也会有并发的出现，这种情况下，第二步可以支持节点的并发复制，这样性能自然提升不少，但实现的复杂度也上升了一个台阶。

先看第一步，构建nextTable，毫无疑问，这个过程只能只有单个线程进行nextTable的初始化，具体实现如下：

```
private final void addCount(long x, int check) {
    ... 省略部分代码
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
```

```
                (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

通过Unsafe.compareAndSwapInt修改sizeCtl值，保证只有一个线程能够初始化nextTable，扩容后的数组长度为原来的两倍，但是容量是原来的1.5。

节点从table移动到nextTable，大体思想是遍历、复制的过程。

1. 首先根据运算得到需要遍历的次数i，然后利用tabAt方法获得i位置的元素f，初始化一个forwardNode实例fwd。
2. 如果f == null，则在table中的i位置放入fwd，这个过程是采用Unsafe.compareAndSwapObjectf方法实现的，很巧妙的实现了节点的并发移动。
3. 如果f是链表的头节点，就构造一个反序链表，把他们分别放在nextTable的i和i+n的位置上，移动完成，采用Unsafe.putObjectVolatile方法给table原位置赋值fwd。
4. 如果f是TreeBin节点，也做一个反序处理，并判断是否需要untreeify，把处理的结果分别放在nextTable的i和i+n的位置上，移动完成，同样采用Unsafe.putObjectVolatile方法给table原位置赋值fwd。

遍历过所有的节点以后就完成了复制工作，把table指向nextTable，并更新sizeCtl为新数组大小的0.75倍 ，扩容完成。

红黑树构造

注意：如果链表结构中元素超过TREEIFY_THRESHOLD阈值，默认为8个，则把链表转化为红黑树，提高遍历查询效率。

```
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
```

接下来我们看看如何构造树结构，代码如下：

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
```

```
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

可以看出，生成树节点的代码块是同步的，进入同步代码块之后，再次验证table中index位置元素是否被修改过。
1、根据table中index位置Node链表，重新生成一个hd为头结点的TreeNode链表。
2、根据hd头结点，生成TreeBin树结构，并把树结构的root节点写到table的index位置的内存中，具体实现如下：

```
TreeBin(TreeNode<K,V> b) {
    super(TREEBIN, null, null, null);
    this.first = b;
    TreeNode<K,V> r = null;
    for (TreeNode<K,V> x = b, next; x != null; x = next) {
        next = (TreeNode<K,V>)x.next;
        x.left = x.right = null;
        if (r == null) {
            x.parent = null;
            x.red = false;
            r = x;
        }
        else {
            K k = x.key;
            int h = x.hash;
            Class<?> kc = null;
            for (TreeNode<K,V> p = r;;) {
                int dir, ph;
                K pk = p.key;
                if ((ph = p.hash) > h)
                    dir = -1;
                else if (ph < h)
                    dir = 1;
                else if ((kc == null &&
                          (kc = comparableClassFor(k)) == null) ||
```

```
                        (dir = compareComparables(kc, k, pk)) == 0)
                    dir = tieBreakOrder(k, pk);
                    TreeNode<K,V> xp = p;
                if ((p = (dir <= 0) ? p.left : p.right) == null) {
                    x.parent = xp;
                    if (dir <= 0)
                        xp.left = x;
                    else
                        xp.right = x;
                    r = balanceInsertion(r, x);
                    break;
                }
            }
        }
    }
    this.root = r;
    assert checkInvariants(root);
}
```

主要根据Node节点的hash值大小构建二叉树。这个红黑树的构造过程实在有点复杂，感兴趣的同学可以看看源码。

get操作

get操作和put操作相比，显得简单了许多。

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

1. 判断table是否为空，如果为空，直接返回null。
2. 计算key的hash值，并获取指定table中指定位置的Node节点，通过遍历链表或则树结构找到对应的节点，返回value值。

总结

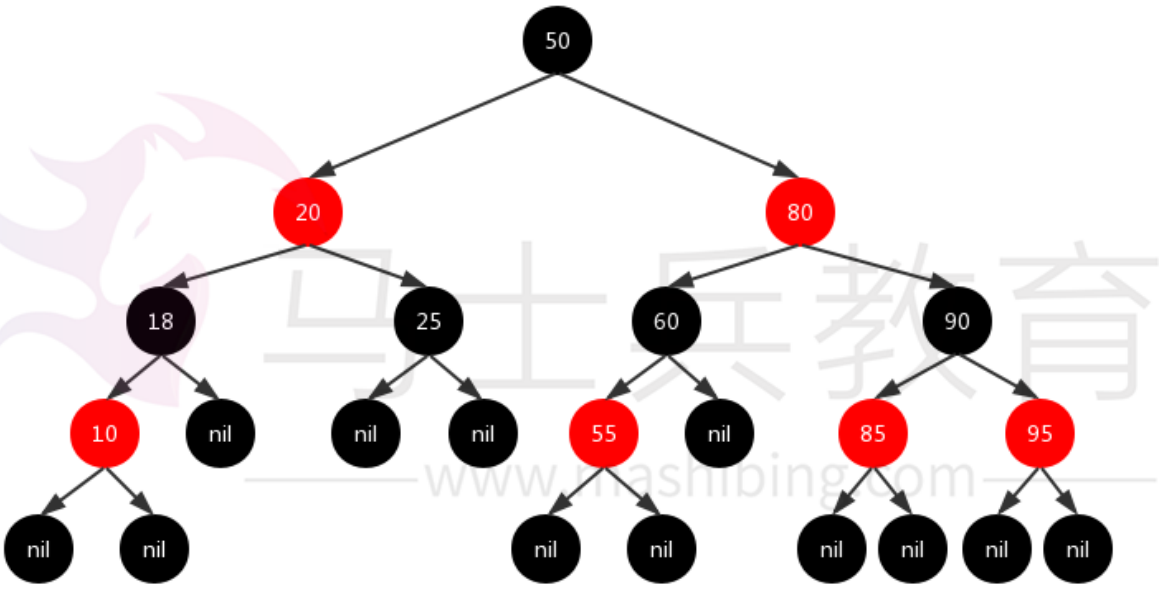
ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和同步包装器包装的 HashMap，使用一个全局的锁来同步不同线程间的并发访问，同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器，这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。
1.6中采用ReentrantLock 分段锁的方式，使多个线程在不同的segment上进行写操作不会发现阻塞行为;1.8中直接采用了内置锁synchronized，难道是因为1.8的虚拟机对内置锁已经优化的足够快了?

ConcurrentHashMap的红黑树实现分析

红黑树

红黑树是一种特殊的二叉树，主要用它存储有序的数据，提供高效的数据检索，时间复杂度为O(lgn)，每个节点都有一个标识位表示颜色，红色或黑色，有如下5种特性：
1、每个节点要么红色，要么是黑色；
2、根节点一定是黑色的；
3、每个空叶子节点必须是黑色的；
4、如果一个节点是红色的，那么它的子节点必须是黑色的；
5、从一个节点到该节点的子孙节点的所有路径包含相同个数的黑色节点；

结构示意图



只要满足以上5个特性的二叉树都是红黑树，当有新的节点加入时，有可能会破坏其中一些特性，需要通过左旋或右旋操作调整树结构，重新着色，使之重新满足所有特性。

ConcurrentHashMap红黑树实现

谈谈ConcurrentHashMap1.7和1.8的不同实现中已经提到，在1.8的实现中，当一个链表中的元素达到8个时，会调用`treeifyBin()`方法把链表结构转化成红黑树结构，实现如下：

```
/**
 * Replaces all linked nodes in bin at given index unless table is
 * too small, in which case resizes instead.
 */
```

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

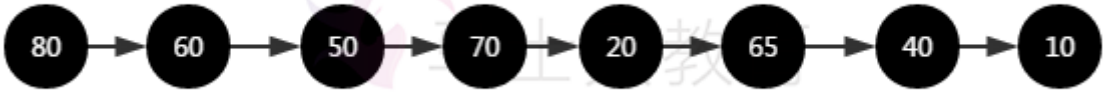
从上述实现可以看出：并非一开始就创建红黑树结构，如果当前`Node`数组长度小于阈值`MIN_TREEIFY_CAPACITY`，默认为64，先通过扩大数组容量为原来的两倍以缓解单个链表元素过大的性能问题。

红黑树构造过程

下面对红黑树的构造过程进行分析：
1、通过遍历`Node`链表，生成对应的`TreeNode`链表，其中`TreeNode`在实现上继承了`Node`类；

```
class TreeNode<K,V> extends Node<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;
    // needed to unlink next upon deletion
    boolean red;
}
```

假设`TreeNode`链表如下，其中节点中的数值代表`hash`值：



www.mashibing.com

2、根据`TreeNode`链表初始化`TreeBin`类对象，`TreeBin`在实现上同样继承了`Node`类，所以初始化完成的`TreeBin`类对象可以保持在`Node`数组中；

```
class TreeBin<K,V> extends Node<K,V> {
    TreeNode<K,V> root;
    volatile TreeNode<K,V> first;
    volatile Thread waiter;
    volatile int lockState;
    // values for lockState
    // set while holding write lock
    static final int WRITER = 1;
    // set when waiting for write lock
    static final int WAITER = 2;
    // increment value for setting read lock
    static final int READER = 4;
}
```

3、遍历 `TreeNode` 链表生成红黑树，一开始二叉树的根节点 `root` 为空，则设置链表中的第一个节点80为 `root`，并设置其 `red` 属性为 `false`，因为在红黑树的特性1中，明确规定根节点必须是黑色的；

```
for (TreeNode<K,V> x = b, next; x != null; x = next) {
    next = (TreeNode<K,V>)x.next;
    x.left = x.right = null;
    if (r == null) {
        x.parent = null;
        x.red = false;
        r = x;
    }
    ...
```

二叉树结构：



4、加入节点60，如果 `root` 不为空，则通过比较节点 `hash` 值的大小将新节点插入到指定位置，实现如下：

```
K k = x.key;
int h = x.hash;
Class<?> kc = null;
for (TreeNode<K,V> p = r;;) {
    int dir, ph;
    K pk = p.key;
    if ((ph = p.hash) > h)
        dir = -1;
    else if (ph < h)
        dir = 1;
    else if ((kc == null &&
              (kc = comparableClassFor(k)) == null) ||
             (dir = compareComparables(kc, k, pk)) == 0)
        dir = tieBreakOrder(k, pk);
```

```
            TreeNode<K,V> xp = p;
        if ((p = (dir <= 0) ? p.left : p.right) == null) {
            x.parent = xp;
            if (dir <= 0)
                xp.left = x;
            else
                xp.right = x;
            r = balanceInsertion(r, x);
            break;
        }
    }
```

其中`x`代表即将插入到红黑树的节点，`p`指向红黑树中当前遍历到的节点，从根节点开始递归遍历，`x`的插入过程如下：

1)、如果`x`的`hash`值小于`p`的`hash`值，则判断`p`的左节点是否为空，如果不为空，则把`p`指向其左节点，并继续和`p`进行比较，如果`p`的左节点为空，则把`x`指向的节点插入到该位置；

2)、如果`x`的`hash`值大于`p`的`hash`值，则判断`p`的右节点是否为空，如果不为空，则把`p`指向其右节点，并继续和`p`进行比较，如果`p`的右节点为空，则把`x`指向的节点插入到该位置；

3)、如果`x`的`hash`值和`p`的`hash`值相等，怎么办？
解决：首先判断节点中的`key`对象的类是否实现了`Comparable`接口，如果实现`Comparable`接口，则调用`compareTo`方法比较两者`key`的大小，但是如果`key`对象没有实现`Comparable`接口，或则`compareTo`方法返回了0，则继续调用`tieBreakOrder`方法计算`dir`值，`tieBreakOrder`方法实现如下：

```
static int tieBreakOrder(Object a, Object b) {
    int d;
    if (a == null || b == null ||
        (d = a.getClass().getName().
         compareTo(b.getClass().getName())) == 0)
        d = (System.identityHashCode(a) <= System.identityHashCode(b) ?
             -1 : 1);
    return d;
}
```

最终比较`key`对象的默认`hashCode()`方法的返回值，因为`System.identityHashCode(a)`调用的是对象`a`默认的`hashCode()`；

插入节点60之后的二叉树：

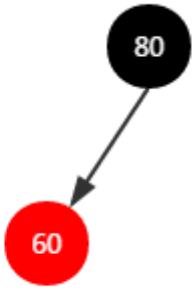


5、当有新节点加入时，可能会破坏红黑树的特性，需要执行`balanceInsertion()`方法调整二叉树，使之重新满足特性，方法中的变量`xp`指向`x`的父节点，`xpp`指向`xp`父节点，`xppl`和`xppr`分别指向`xpp`的左右子节点，`balanceInsertion()`方法首先会把新加入的节点设置成红色。

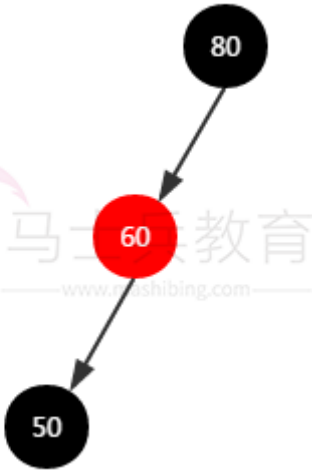
①、加入节点60之后，此时`xp`指向节点80，其父节点为空，直接返回。

```
if ((xp = x.parent) == null) {
    x.red = false;
    return x;
}
else if (!xp.red || (xpp = xp.parent) == null)
    return root;
```

调整之后的二叉树：



②、加入节点50，二叉树如下：



继续执行`balanceInsertion()`方法调整二叉树，此时节点50的父节点60是左儿子，走如下逻辑：

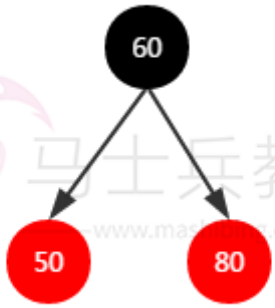
```
if (xp == (xppl = xpp.left)) {
    if ((xppr = xpp.right) != null && xppr.red) {
        xppr.red = false;
        xp.red = false;
        xpp.red = true;
        x = xpp;
    }
```

```
    else {
        if (x == xp.right) {
            root = rotateLeft(root, x = xp);
            xpp = (xp = x.parent) == null ? null : xp.parent;
        }
        if (xp != null) {
            xp.red = false;
            if (xpp != null) {
                xpp.red = true;
                root = rotateRight(root, xpp);
            }
        }
    }
}
```

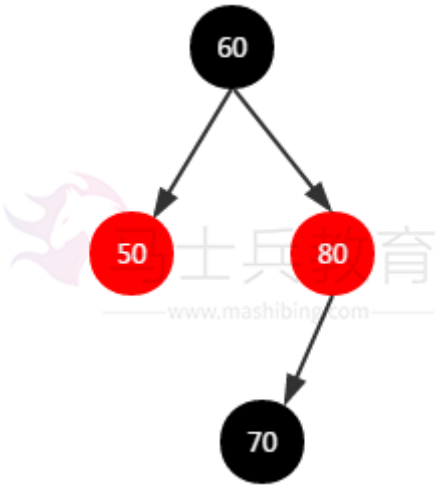
根据上述逻辑，把节点60设置成黑色，把节点80设置成红色，并对节点80执行右旋操作，右旋实现如下：

```
static <K,V> TreeNode<K,V> rotateRight(TreeNode<K,V> root,
                                       TreeNode<K,V> p) {
    TreeNode<K,V> l, pp, lr;
    if (p != null && (l = p.left) != null) {
        if ((lr = p.left = l.right) != null)
            lr.parent = p;
        if ((pp = l.parent = p.parent) == null)
            (root = l).red = false;
        else if (pp.right == p)
            pp.right = l;
        else
            pp.left = l;
        l.right = p;
        p.parent = l;
    }
    return root;
}
```

右旋之后的红黑树如下：



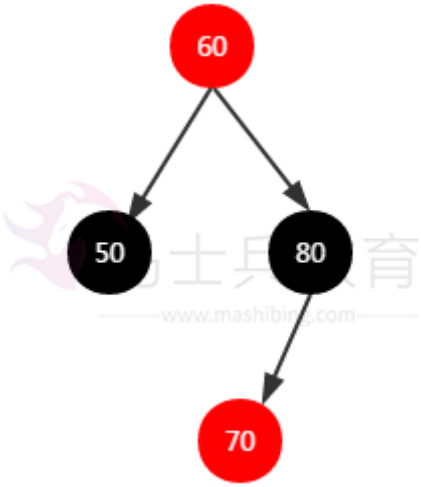
③、加入节点70，二叉树如下：



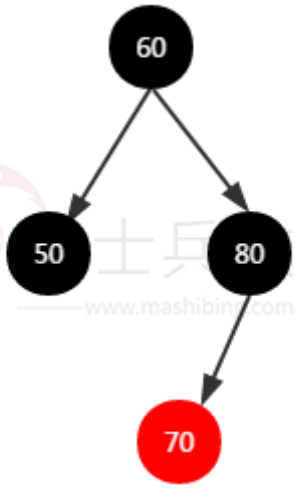
继续执行`balanceInsertion()`方法调整二叉树，此时父节点80是个右儿子，节点70是左儿子，且叔节点50不为空，且是红色的，则执行如下逻辑：

```
if (xppl != null && xppl.red) {
    xppl.red = false;
    xp.red = false;
    xpp.red = true;
    x = xpp;
}
```

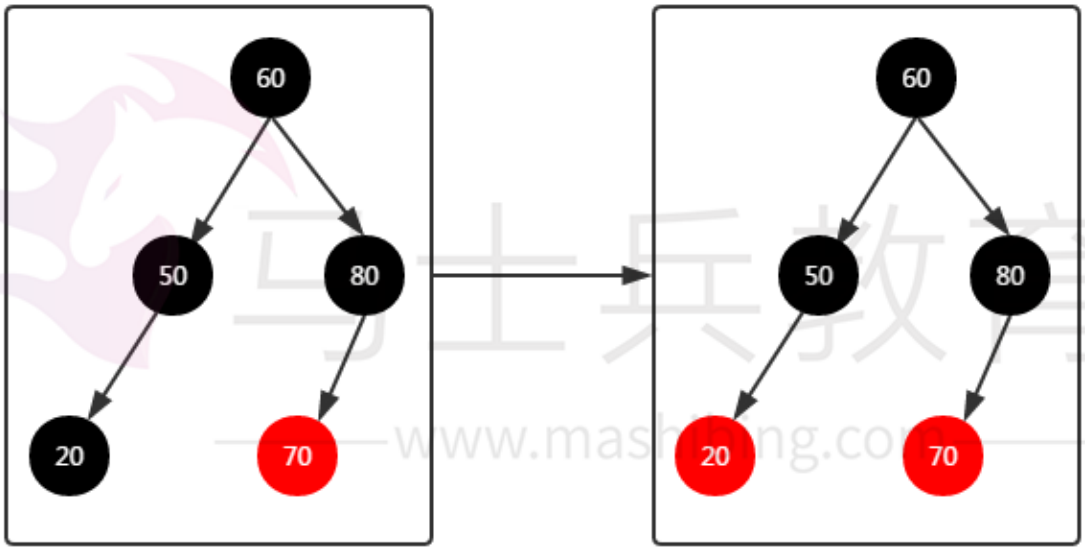
此时二叉树如下：



此时`x`指向`xpp`，即节点60，继续循环处理`x`，设置其颜色为黑色，最终二叉树如下：

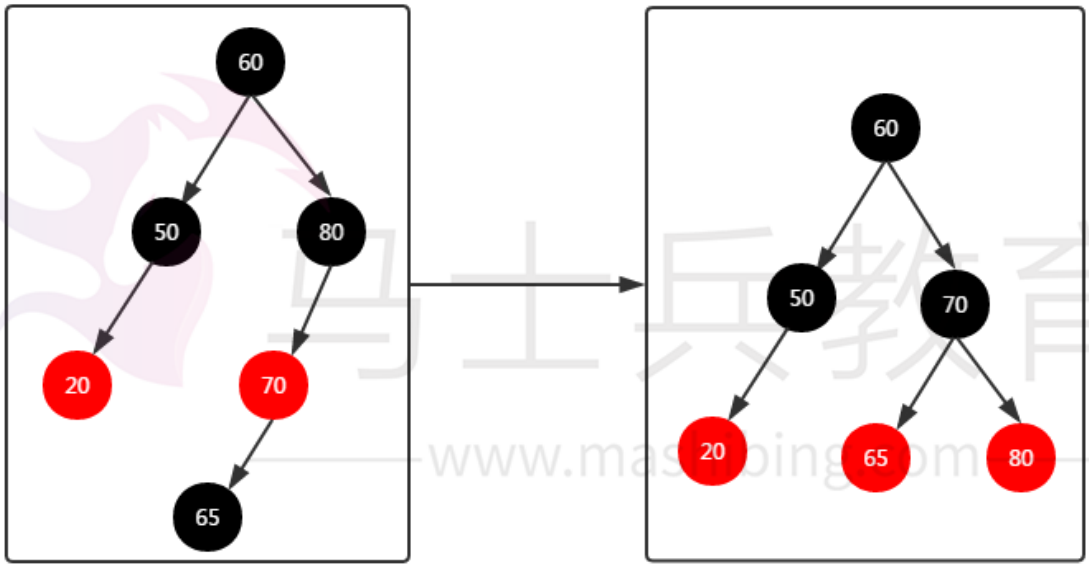


④、加入节点20，二叉树变化如下：



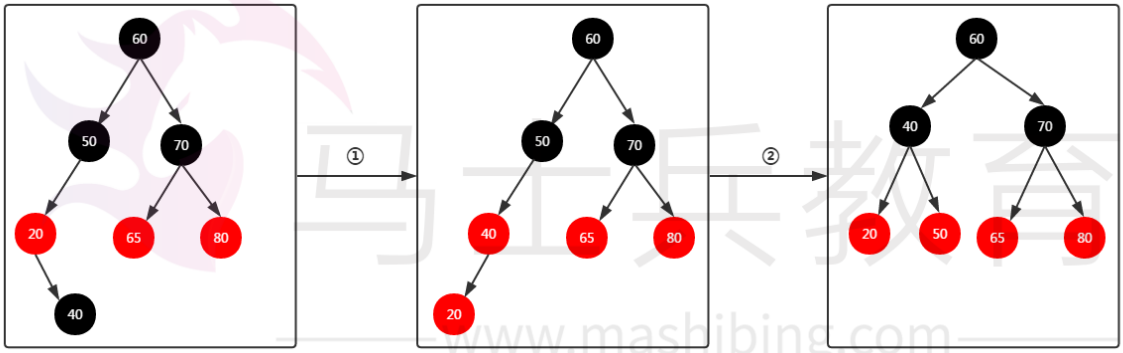
因为节点20的父节点50是一个黑色的节点，不需要进行调整；

⑤、加入节点65，二叉树变化如下：



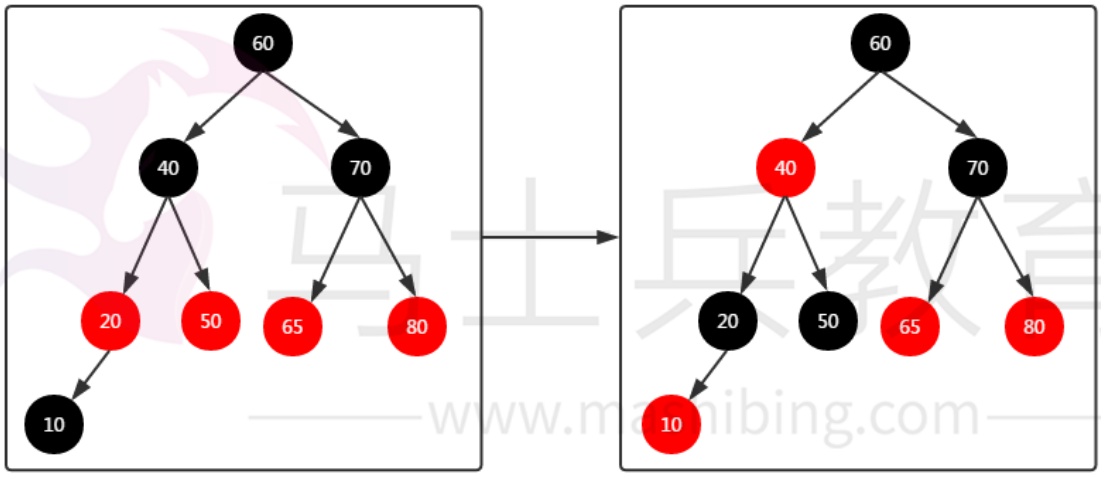
对节点80进行右旋操作。

⑥、加入节点40，二叉树变化如下：



1、对节点20执行左旋操作；
2、对节点50执行右旋操作；

最后加入节点10，二叉树变化如下：



重新对节点进行着色，到此为止，红黑树已经构造完成；

说说反射的用途及实现，反射是不是很慢，我们在项目中是否要避免使 用反射；

一、用途 反射被广泛地用于那些需要在运行时检测或修改程序行为的程序中。

二、实现方式 Foo foo = new Foo();

第一种：通过Object类的getClass方法 Class cla = foo.getClass();

第二种：通过对象实例方法获取对象 Class cla = foo.class;

第三种：通过Class.forName方式 Class cla = Class.forName("xx.xx.Foo");

三、缺点

1）影响性能 反射包括了一些动态类型，所以 JVM 无法对这些代码进行优化。因此，反射操作的效 率要比那些非反射操作低得多。我们应该避免在经常被执行的代码或对性能要求很高的程 序中使用反射。

2）安全限制 使用反射技术要求程序必须在一个没有安全限制的环境中运行。

3）内部暴露 由于反射允许代码执行一些在正常情况下不被允许的操作（比如访问私有的属性和方法），所以 使用反射可能会导致意料之外的副作用 - - 代码有功能上的错误，降低可移植性。 反射代码破坏 了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化。

说说自定义注解的场景及实现；

利用自定义注解,结合SpringAOP可以完成权限控制、日志记录、统一异常处理、数字签名、数 据加解密等功能。
实现场景（API接口数据加解密）

1）自定义一个注解，在需要加解密的方法上添加该注解

2）配置SringAOP环绕通知

3）截获方法入参并进行解密

4）截获方法返回值并进行加密

List 和 Map 区别

一、概述

List是存储单列数据的集合，Map是存储键和值这样的双列数据的集合， List中存储的数据是有顺序，并且允许重复，值允许有多个null； Map中存储的数据是没有顺序的，键不能重复，值是可以有重复的，key最多有一个null。

二、明细

List

1）可以允许重复的对象。

2）可以插入多个null元素。

3）是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。

4）常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引 的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

Map

1）Map不是collection的子接口或者实现类。Map是一个接口。

2）Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象 但键对象必须是唯一的。

3）TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。

4）Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。

5）Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）

Set（问题扩展）

1）不允许重复对象

2）无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或Comparable 维 护了一个排序顺序。

3）只允许一个 null 元素

4）Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

三、场景（问题扩展）

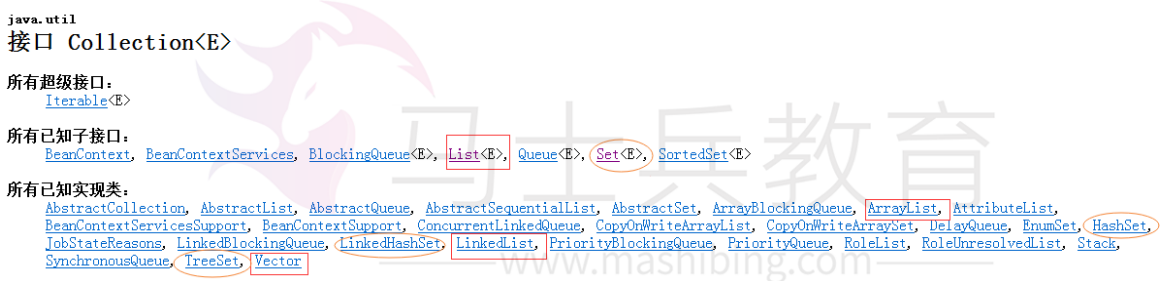
1）如果你经常会使用索引来对容器中的元素进行访问，那么 List 是你的正确的选择。如果你已 经知道索引了的话，那么 List 的实现类比如 ArrayList 可以提供更快速的访问,如果经常添加删除 元素的，那么肯定要选择LinkedList。

2）如果你想容器中的元素能够按照它们插入的次序进行有序存储，那么还是 List，因为 List 是 一个有序容器，它按照插入顺序进行存储。

3）如果你想保证插入元素的唯一性，也就是你不想有重复值的出现，那么可以选择一个 Set 的 实现类，比如 HashSet、LinkedHashSet 或者 TreeSet。所有 Set 的实现类都遵循了统一约束 比如唯一性，而且还提供了额外的特性比如 TreeSet 还是一个 SortedSet，所有存储于 TreeSet 中的元素可以使用 Java 里的 Comparator 或者 Comparable 进行排序。LinkedHashSet 也按 照元素的插入顺序对它们进行存储。

4）如果你以键和值的形式进行数据存储那么 Map 是你正确的选择。你可以根据你的后续需要 从 Hashtable、HashMap、TreeMap 中进行选择。

List、Set、Map的区别



马士兵教育 www.mashibing.com

（图一）

1.面试题：你说说collection里面有什么子类。

（其实面试的时候听到这个问题的时候，你要知道，面试官是想考察List，Set）

正如图一，list和set是实现了collection接口的。



java.util
接口 List<E>

所有超级接口：
Collection<E>, Iterable<E>

所有已知实现类：
AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

public interface List<E>
extends Collection<E>

有序的 collection（也称为*序列*）。此接口的用户可以对列表中每个元素的插入位置进行精确地控制。用户可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。

与 set 不同，列表通常允许重复的元素。更正式地说，列表通常允许满足 e1.equals(e2) 的元素对 e1 和 e2，并且如果列表本身允许 null 元素的话，通常它们允许多个 null 元素。难免有人希望通过在用户尝试插入重复元素时抛出运行时异常的方法来禁止重复的列表，但我们希望这种用法越少越好。

List 接口在 iterator、add、remove、equals 和 hashCode 方法的协定上加了一些其他约定，超过了 Collection 接口中指定的约定。为方便起见，这里也包括了其他继承方法的声明。

List 接口提供了 4 种对列表元素进行定位（索引）访问方法。列表（像 Java 数组一样）是基于 0 的。注意，这些操作可能在和某些实现（例如 LinkedList 类）的索引值成比例的时间内执行。因此，如果调用方不知道实现，那么在列表元素上迭代通常优于用索引遍历列表。

List 接口提供了特殊的迭代器，称为 ListIterator，除了允许 Iterator 接口提供的正常操作外，该迭代器还允许元素插入和替换，以及双向访问。还提供了一个方法来获取从列表中指定位置开始的列表迭代器。

List 接口提供了两种搜索指定对象的方法。从性能的观点来看，应该小心使用这些方法。在很多实现中，它们将执行高开销的线性搜索。

List 接口提供了两种在列表的任意位置高效插入和移除多个元素的方法。

注意：尽管列表允许把自身作为元素包含在内，但建议要特别小心：在这样的列表上，equals 和 hashCode 方法不再是定义良好的。

某些列表实现对列表可能包含的元素有限制。例如，某些实现禁止 null 元素，而某些实现则对元素的类型有限制。试图添加不合格的元素会抛出未经检查的异常，通常是 NullPointerException 或 ClassCastException。试图查询不合格的元素是否存在可能会抛出异常，也可能简单地返回 false；某些实现会采用前一种行为，而某些则采用后者。概括地说，试图对不合格元素执行操作时，如果完成该操作后不会导致在列表中插入不合格的元素，则该操作可能抛出一个异常，也可能成功，这取决于实现的选择。此接口的规范中将这样的异常标记为“可选”。

此接口是 Java Collections Framework 的成员。

（图二）

List：1.可以允许重复的对象。

2.可以插入多个null元素。

3.是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。

4.常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

java.util
接口 Set<E>

所有超级接口：
Collection<E>, Iterable<E>

所有已知子接口：
SortedSet<E>

所有已知实现类：
AbstractSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

---

public interface **Set<E>**
extends Collection<E>

一个不包含重复元素的 collection。更正式地说，set 不包含满足 e1.equals(e2) 的元素对 e1 和 e2，并且最多包含一个 null 元素。正如其名称所暗示的，此接口模仿了数学上的 *set* 抽象。

在所有构造方法以及 add、equals 和 hashCode 方法的协定上，Set 接口还加入了其他规定，这些规定超出了从 Collection 接口所继承的内容。出于方便考虑，它还包括了其他继承方法的声明（这些声明的规范已经专门针对 Set 接口进行了修改，但是没有包含任何其他的规定）。

对这些构造方法的其他规定是（不要奇怪），所有构造方法必须创建一个不包含重复元素的 set（正如上面所定义的）。

注：如果将可变对象用作 set 元素，那么必须极其小心。如果对象是 set 中某个元素，以一种影响 equals 比较的方式改变对象的值，那么 set 的行为就是不确定的。此项禁止的一个特殊情况是不允许某个 set 包含其自身作为元素。

某些 set 实现对其所包含的元素有所限制。例如，某些实现禁止 null 元素，而某些则对其元素的类型所有限制。试图添加不合格的元素会抛出未经检查的异常，通常是 NullPointerException 或 ClassCastException。试图查询不合格的元素是否存在可能会抛出异常，也可能简单地返回 false；某些实现会采用前一种行为，而某些则采用后者。概括地说，试图对不合格元素执行操作时，如果完成该操作后不会导致在 set 中插入不合格的元素，则该操作可能抛出一个异常，也可能成功，这取决于实现的选择。此接口的规范中将这样的异常标记为“可选”。

此接口是 Java Collections Framework 的成员。

（图三）

Set：1.不允许重复对象

2. 无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或者 Comparable 维护了一个排序顺序。

3. 只允许一个 null 元素

4.Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

java.util
接口 Map<K,V>

所有已知子接口：
ConcurrentMap<K,V>, SortedMap<K,V>

所有已知实现类：
AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

---

public interface **Map<K,V>**

将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射一个值。

此接口代替 Dictionary 类，后者完全是一个抽象类，而不是一个接口。

Map 接口提供三种*collection 视图*，允许以键集、值集合或键-值映射关系集的形式查看某个映射的内容。映射的*顺序* 定义为迭代器在映射的 collection 视图中返回其元素的顺序。某些映射实现可明确保证其顺序，如 TreeMap 类；某些映射实现则不保证顺序，如 HashMap 类。

注：将可变对象用作映射键时必须格外小心。当对象是映射中某个键时，如果以影响 equals 比较的方式更改了对象的值，则映射的行为就不是特定的。此项禁止的一个特殊情况是不允许某个映射包含其自身作为键。虽然允许某个映射包含其自身作为值，但建议要格外小心，在这样的映射上将无法再定义 equals 和 hashCode 方法。

所有通用的映射实现类应该提供两个“标准的”构造方法：一个 void（无参数）构造方法，用于创建空映射，另一个是带有 Map 类型单参数的构造方法，用于创建一个与其参数具有相同键-值映射关系的新映射。实际上，后一个构造方法允许用户复制任意映射，生成所需类的一个等价映射。尽管无法强制执行该建议（因为接口不能包含构造方法），但是 JDK 中所有通用的映射实现都遵从它。

该接口中包含的“破坏性”方法可修改其所操作的映射，如果此映射不支持该操作，则指定这些方法抛出 UnsupportedOperationException。如果是这样，那么在调用对该映射无效时，这些方法可能，但并不一定抛出 UnsupportedOperationException。例如，如果某个不可修改的映射（其映射关系是“重叠”的）为空，则对该映射调用 putAll(Map) 方法时，可能但并不一定抛出异常。

某些映射实现对可能包含的键和值有所限制。例如，某些实现禁止空键和空值，而某些实现则对其键的类型有限制。尝试插入不合格的键或值会抛出一个未经检查的异常，通常是 NullPointerException 或 ClassCastException。试图查询是否存在不合格的键或值可能抛出异常，或者只是简单地返回 false；某些实现将表现出前一种行为，而某些实现则表现后一种。较为常见的是，试图对不合格的键或值执行操作且该操作的完成不会导致不合格的元素被插入映射中时，将可能抛出一个异常，也可能操作成功，这取决于实现本身。这样的异常在此接口的规范中标记为“可选”。

（图四）

1.Map不是collection的子接口或者实现类。Map是一个接口。

2.Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。

3. TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。

4. Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。

5.Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）

2.面试题：什么场景下使用list，set，map呢?

（或者会问为什么这里要用list、或者set、map，这里回答它们的优缺点就可以了）

答:

1. **如果你经常会使用索引来对容器中的元素进行访问，那么 List 是你的正确的选择。如果你已经知道索引了的话，那么 List 的实现类比如 ArrayList 可以提供更快速的访问,如果经常添加删除元素的，那么肯定要选择LinkedList。**
2. **如果你想容器中的元素能够按照它们插入的次序进行有序存储，那么还是 List，因为 List 是一个有序容器，它按照插入顺序进行存储。**
3. **如果你想保证插入元素的唯一性，也就是你不想有重复值的出现，那么可以选择一个 Set 的实现类，比如 HashSet、LinkedHashSet 或者 TreeSet。所有 Set 的实现类都遵循了统一约束比如唯一性，而且还提供了额外的特性比如 TreeSet 还是一个 SortedSet，所有存储于 TreeSet 中的元素可以使用 Java 里的 Comparator 或者 Comparable 进行排序。LinkedHashSet 也按照元素的插入顺序对它们进行存储。**
4. **如果你以键和值的形式进行数据存储那么 Map 是你正确的选择。你可以根据你的后续需要从 Hashtable、HashMap、TreeMap 中进行选择。**

```
Student stu1 = new Student("廖超伟", 21, '男', 172, "13602312113");
Student stu2 = new Student("陈卫泽", 25, '男', 158, "13709394767");
Student stu3 = new Student("刘德华", 18, '男', 180, "18976341237");
Student stu4 = new Student("陈晓东", 20, '男', 170, "13602003113");
Student stu5 = new Student("苏亦晨", 19, '男', 185, "15521134916");
// collections的sort方法只能传递list类型的，所以就用arraylist
ArrayList<Student> list = new ArrayList<Student>();
list.add(stu1);List: 1.可以允许重复的对象。2.可以插入多个null元素。
list.add(stu3);      3.是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺
list.add(stu2); 序。 4.常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提
list.add(stu5); 供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素
list.add(stu4); 的场合更为合适。
```

```
System.out.println("------------直接输出-------------");
for (Object object : list) {
    System.out.println(object);
}
/**
 * 使用collections的sort方法让集合排序， 使用其方法必须要重写Comparable接口的compareTo方
 */
Collections.sort(list);
System.out.println("\n----使用了sort输出，必须要重写Comparable接口的compareTo方法--------
//因为用了泛型集合，所以这里可以使用student类型
for (Student object : list) {
    System.out.println(object);
}
```

```
------------直接输出-------------
姓名:廖超伟       年龄：21 性别：男 身高:172 手机:13602312113
姓名:刘德华       年龄：18 性别：男 身高:180 手机:18976341237
姓名:陈卫泽       年龄：25 性别：男 身高:158 手机:13709394767
姓名:苏亦晨       年龄：19 性别：男 身高:185 手机:15521134916
姓名:陈晓东       年龄：20 性别：男 身高:170 手机:13602003113

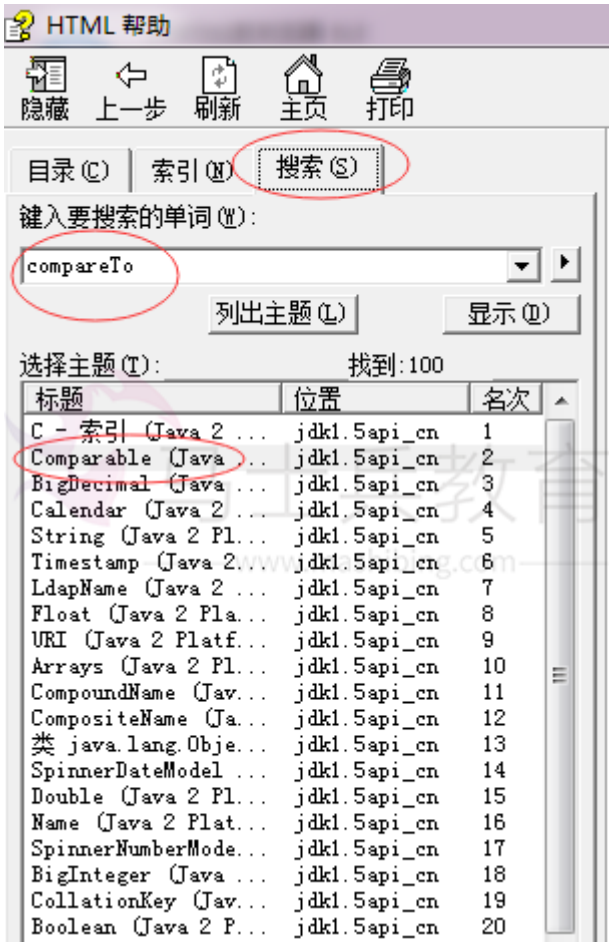
----使用了sort输出，必须要重写Comparable接口的compareTo方法--------
姓名:刘德华       年龄：18 性别：男 身高:180 手机:18976341237
姓名:苏亦晨       年龄：19 性别：男 身高:185 手机:15521134916
姓名:陈晓东       年龄：20 性别：男 身高:170 手机:13602003113
姓名:廖超伟       年龄：21 性别：男 身高:172 手机:13602312113
姓名:陈卫泽       年龄：25 性别：男 身高:158 手机:13709394767
```

按照年龄排序，如果我要按照身高排序呢，要怎么写呢？

大家可以跟着下面的步骤一起尝试一下。

1.我们知道了列表要实现排序，需要重写comparable接口的compareTo的方法。

但是是我不知道comparaTo里面要怎么写呢，它有传入参数吗？它有返回值吗？如果有事什么类型的呢？ok，下面一起来做一下。先把这个链接的帮助文档下载下来。下载完之后，打开帮助文档，



java.lang

接口 Comparable<T>

所有已知子接口：

Delayed, Name, ScheduledFuture<V>

所有已知实现类：

Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, Character, CharBuffer, Charset, CollationKey, CompositeName, CompoundName, Date, Date, Double, DoubleBuffer, ElementType, Enum, File, Float, FloatBuffer, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType, GregorianCalendar, IntBuffer, Integer, JTable.PrintMode, KeyRep.Type, LdapName, Long, LongBuffer, MappedByteBuffer, MemoryType, ObjectStreamField, Proxy.Type, Rdn, RetentionPolicy, RoundingMode, Short, ShortBuffer, SSLEngineResult.HandshakeStatus, SSLEngineResult.Status, String, Thread.State, Time, Timestamp, TimeUnit, URI, UUID

---

public interface **Comparable<T>**

此接口强行对实现它的每个类的对象进行整体排序。此排序被称为该类的*自然排序*，类的 compareTo 方法被称为它的*自然比较方法*。

实现此接口的对象列表（和数组）可以通过 Collections.sort（和 Arrays.sort）进行自动排序。实现此接口的对象可以用作有序映射表中的键或有序集合中的元素，无需指定比较器。

对于类 C 的每一个 e1 和 e2 来说，当且仅当 (e1.compareTo((Object)e2) == 0) 与 e1.equals((Object)e2) 具有相同的布尔值时，类 C 的自然排序才叫做*与 equals 一致*。注意，null 不是任何类的实例，即使 e.equals(null) 返回 false，e.compareTo(null) 也会抛出 NullPointerException。

强烈推荐（虽然不是必需的）使自然排序与 equals 一致。这是因为在使用其自然排序与 equals 不一致的元素（或键）时，没有显式比较器的有序集合（和有序映射表）行为表现“怪异”。尤其是，这样的有序集合（或有序映射表）违背了根据 equals 方法定义的集合（或映射表）的常规协定。

例如，如果将两个键 a 和 b 添加到一个没有使用显式比较器的有序集合中，使得 (!a.equals((Object)b) && a.compareTo((Object)b) == 0)，则第二个 add 操作返回 false（有序集合的大小没有增加），因为从有序集合的角度来看，a 和 b 是等效的。

实际上，所有执行比较的 Java 核心类都具有与 equals 一致的自然排序。java.math.BigDecimal 是个例外，它的自然排序把值相等但精确度不同的 BigDecimal 对象（比如 4.0 和 4.00）等同起来。

为了向数学上倾斜，在给定 C 类的基础上定义自然排序的*关系*如下：

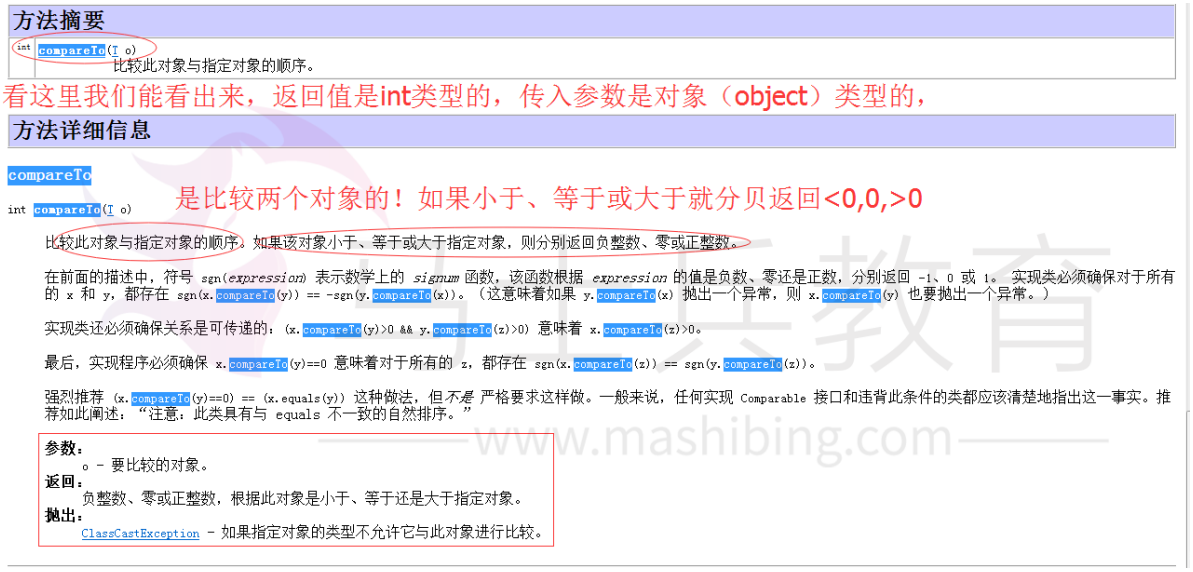
{(x, y) such that x.compareTo((Object)y) <= 0}。

整体排序的 *quotient* 是：

{(x, y) such that x.compareTo((Object)y) == 0}。

它直接遵循 compareTo 的协定，商是 C 的*等价关系*，自然排序是 C的*整体排序*。当我们说类的自然排序*与 equals 一致*时，是指自然排序的商是由类的 equals(Object) 方法定义的等价关系。

{(x, y) such that x.equals((Object)y)}。



2.看完了帮助文档是不是心里稍微有点底气了呢，那现在打开eclipse我们一起来写一写吧。

首先我们要比较对象的哪个属性呢。年龄？身高？还是体重？刚刚看帮助文档已经知道了，所以下面大家一起来写一下。

```
@Override
public int compareTo(Object o) {
    //使用当前对象的年龄和其他对象的年龄比较，如果小于0则返回负整数，-2，-3都可以，随你，下同
    int z=this.age-((Student)o).getAge();
    if(z<0)
        return -1;
    else if (z==0) {
        return 0;
    }
    else {
        return 1;//这里返回2,3,4,5...都可以，随意
    }
}
```

如果大家也是像上图这种写法，那么再想一想有没有更好的办法。(我这样吻是肯定有的，好好看看帮助文档，你就知道了，我知道你只要用心想想，肯定想出来的！)

好了，写完年龄，不去继续花几分钟把按照身高来排序也写一下吧。

Arraylist 与 LinkedList 区别，ArrayList 与 Vector 区别;

1）数据结构 Vector、ArrayList内部使用数组，而LinkedList内部使用双向链表，由数组和链表的特性知： LinkedList适合指定位置插入、删除操作，不适合查找； ArrayList、Vector适合查找，不适合指定位置的插入删除操作。 但是ArrayList越靠近尾部的元素进行增删时，其实效率比LinkedList要高

2)线程安全 Vector线程安全，ArrayList、LinkedList线程不安全。

3）空间 ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间。

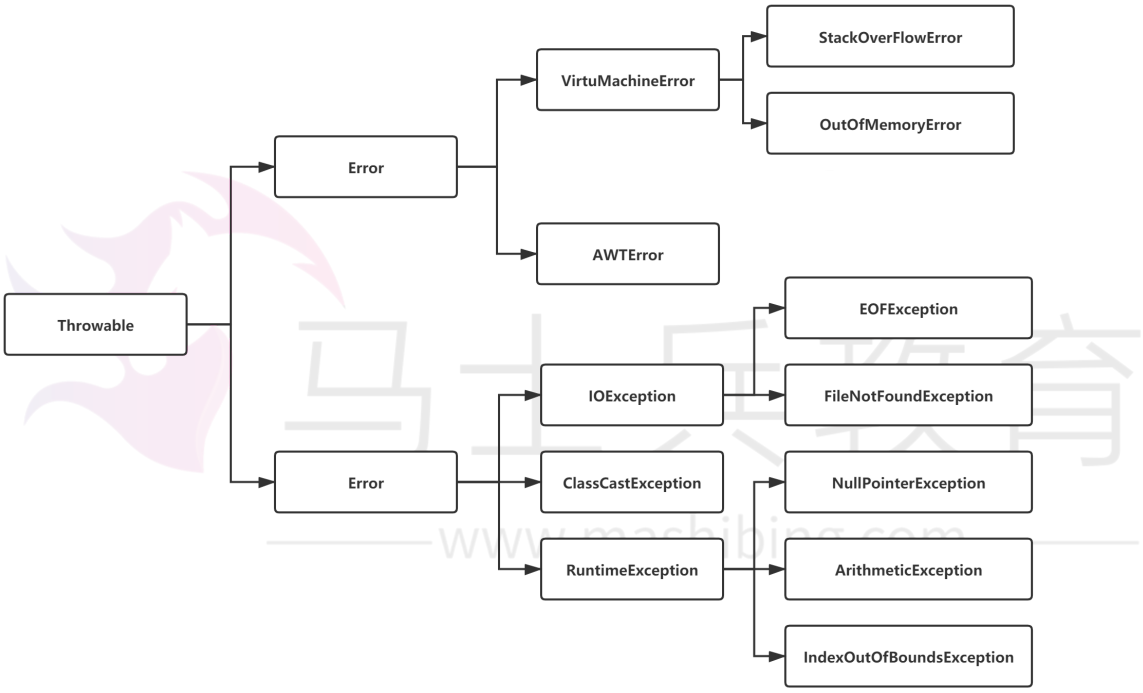
Java异常架构与异常关键字

Java异常简介

Java异常是Java提供的一种识别及响应错误的一致性机制。

Java异常机制可以使程序中异常处理代码和正常业务代码分离，保证程序代码更加优雅，并提高程序健壮性。在有效使用异常的情况下，异常能清晰的回答 what, where, why这3个问题：异常类型回答了“什么”被抛出，异常堆栈跟踪回答了“在哪”抛出，异常信息回答了“为什么”会抛出。

Java异常架构



1. Throwable

Throwable 是 Java 语言中所有错误与异常的超类。

Throwable 包含两个子类：Error（错误）和 Exception（异常），它们通常用于指示发生了异常情况。

Throwable 包含了其线程创建时线程执行堆栈的快照，它提供了 printStackTrace() 等接口用于获取堆栈跟踪数据等信息。

2. Error（错误）

定义：Error 类及其子类。程序中无法处理的错误，表示运行应用程序中出现了严重的错误。

特点：此类错误一般表示代码运行时 JVM 出现问题。通常有 Virtual

MachineError（虚拟机运行错误）、NoClassDefFoundError（类定义错误）

等。比如 OutOfMemoryError：内存不足错误；StackOverflowError：栈溢出错误。此类错误发生时，JVM 将终止线程。这些错误是不受检异常，非代码性错误。因此，当此类错误发生时，应用程序不

应该去处理此类错误。按照Java惯例，我们是不应该实现任何新的Error子类的！

3. Exception（异常）

程序本身可以捕获并且可以处理的异常。Exception 这种异常又分为两类：运行时异常和编译时异常。

运行时异常

定义：RuntimeException 类及其子类，表示 JVM 在运行期间可能出现的异常。

特点：Java 编译器不会检查它。也就是说，当程序中可能出现这类异常时，倘若既"没有通过throws声明抛出它"，也"没有用try-catch语句捕获它"，还是会编译通过。比如NullPointerException空指针异常、

ArrayIndexOutBoundException数组下标越界异常、ClassCastException类型转换异常、ArithmeticExecption算术异常。此类异常属于不受检异常，一般是由程序逻辑错误引起的，在程序中可以选择捕获处理，也可以不处理。虽然 Java 编译器不会检查运行时异常，但是我们也可以通过 throws 进行声明抛

出，也可以通过 try-catch 对它进行捕获处理。如果产生运行时异常，则需要通过修改代码来进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

RuntimeException 异常会由 Java 虚拟机自动抛出并自动捕获（就算我们没写异常捕获语句运行时也会抛出错误！！），此类异常的出现绝大数情况是代码本身有问题应该从逻辑上去解决并改进代码。

编译时异常

定义: Exception 中除 RuntimeException 及其子类之外的异常。

特点: Java 编译器会检查它。如果程序中出现此类异常，比如

ClassNotFoundException（没有找到指定的类异常），IOException（IO流异常），要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。在程序中，通常不会自定义该类异常，而是直接使用系统提供的异常类。该异常我们必须手动在代码里添加捕获语句来处理该异常。

4. 受检异常与非受检异常

Java 的所有异常可以分为受检异常（checked exception）和非受检异常

（unchecked exception）。

受检异常

编译器要求必须处理的异常。正确的程序在运行过程中，经常容易出现的、符合

预期的异常情况。一旦发生此类异常，就必须采用某种方式进行处理。除

RuntimeException 及其子类外，其他的 Exception 异常都属于受检异常。编译器会检查此类异常，也就是说当编译器检查到应用中的某处可能会此类异常时，将会提示你处理本异常——要么使用try-catch捕获，要么使用方法签名中用 throws 关键字抛出，否则编译不通过。

非受检异常

编译器不会进行检查并且不要求必须处理的异常，也就说当程序中出现此类异常时，即使我们没有try-catch捕获它，也没有使用throws抛出该异常，编译也会正常通过。该类异常包括运行时异常（RuntimeException极其子类）和错误(Error)

Java异常关键字

• try – 用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。

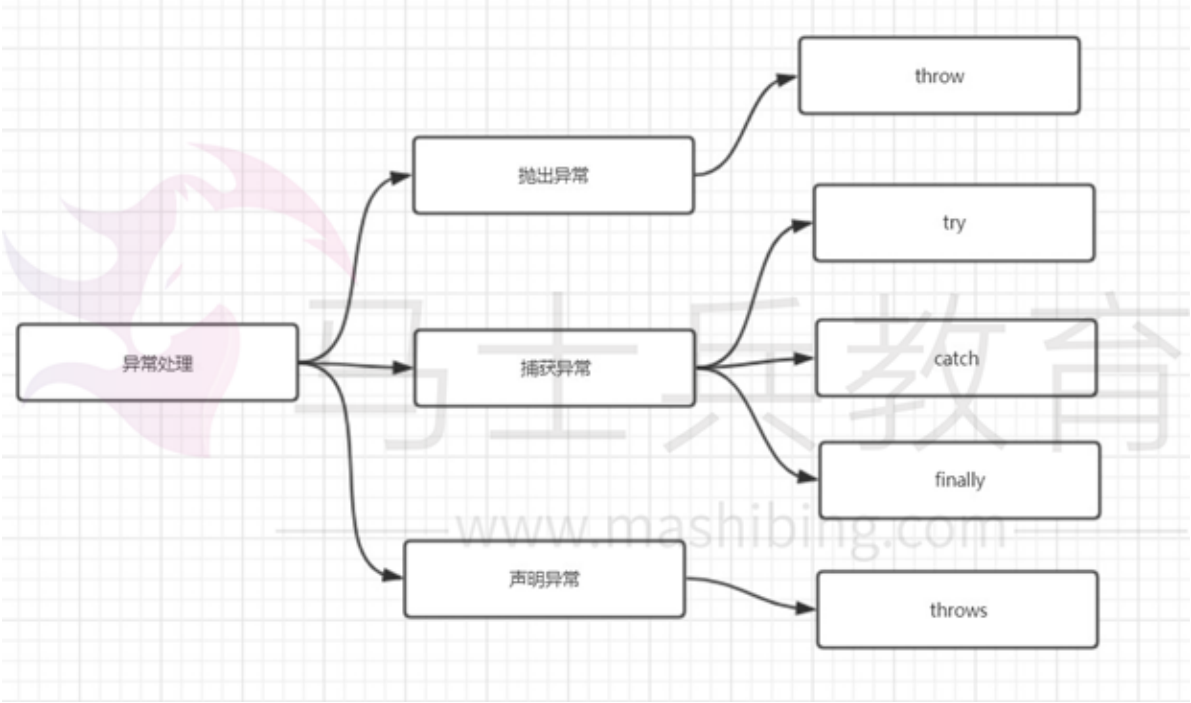
• catch – 用于捕获异常。catch用来捕获try语句块中发生的异常。

• finally – finally语句块总是会被执行。它主要用于回收在try块里打开的物力资源(如数据库连接、网络连接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了 return或者throw等终止方法的语句，则就不会跳回执行，直接停止。

• throw – 用于抛出异常。

• throws – 用在方法签名中，用于声明该方法可能抛出的异常。

Java异常处理



Java 通过面向对象的方法进行异常处理，一旦方法抛出异常，系统自动根据该异常对象寻找合适异常处理器（Exception Handler）来处理该异常，把各种不同的异常进行分类，并提供了良好的接口。在 Java 中，每个异常都是一个对

象，它是 Throwable 类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。Java 的异常处理是通过 5 个关键词来实现的：try、 catch、throw、throws 和 finally。

在Java应用中，异常的处理机制分为声明异常，抛出异常和捕获异常。

声明异常

通常，应该捕获那些知道如何处理的异常，将不知道如何处理的异常继续传递下

去。传递异常可以在方法签名处使用 throws 关键字声明可能会抛出的异常。注意

非检查异常（Error、RuntimeException 或它们的子类）不可使用 throws 关键字来声明要抛出的异常。

```
    ![img](03-Java异常面试题（2020最新版）.assets/clip_image002-1604914706296.gif)
一个方法出现编译时异常，就需要 try-catch/ throws 处理，否则会导致编译错误
```

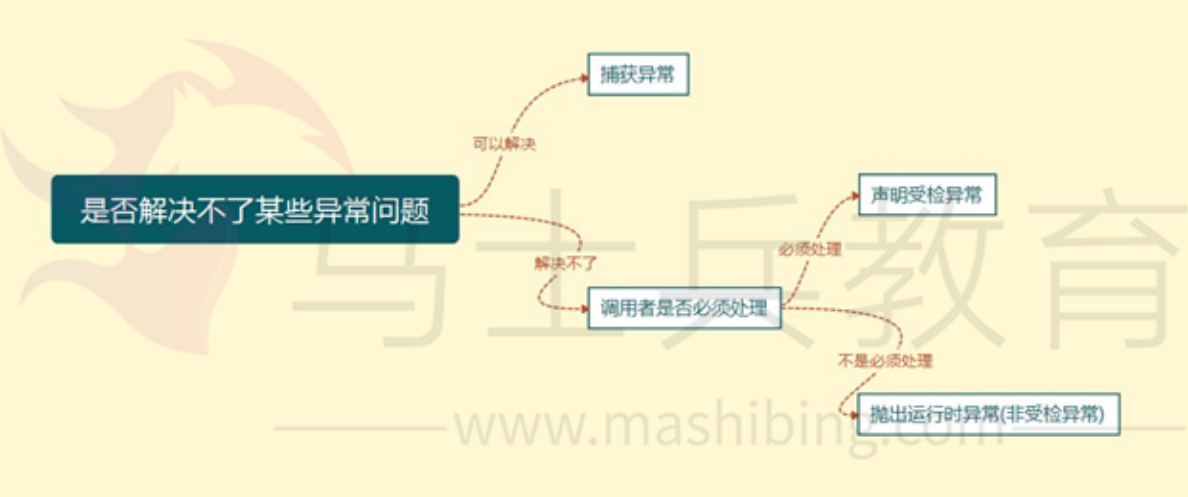
抛出异常

如果你觉得解决不了某些异常问题，且不需要调用者处理，那么你可以抛出异常。 throw关键字作用是在方法内部抛出一个Throwable类型的异常。任何Java代码都可以通过throw语句抛出异常。

捕获异常

程序通常在运行之前不报错，但是运行后可能会出现某些未知的错误，但是还不想直接抛出到上一级，那么就需要通过try...catch...的形式进行异常捕获，之后根据不同的异常情况来进行相应的处理。如何选择异常类型

可以根据下图来选择是捕获异常，声明异常还是抛出异常



常见异常处理方式直接抛出异常

通常，应该捕获那些知道如何处理的异常，将不知道如何处理的异常继续传递下去。传递异常可以在方法签名处使用 throws 关键字声明可能会抛出的异常。

```
1 private static void readFile(String filePath) throws IOException {
2 File file = new File(filePath);
3 String result;
4 BufferedReader reader = new BufferedReader(new FileReader(file));
5 while((result = reader.readLine())!=null) {
6 System.out.println(result);
7   }
8   reader.close();
9   }
```

封装异常再抛出

有时我们会从 catch 中抛出一个异常，目的是为了改变异常的类型。多用于在多系统集成时，当某个子系统故障，异常类型可能有多种，可以用统一的异常类型向外暴露，不需暴露太多内部异常细节。

```
1 private static void readFile(String filePath) throws MyException {
2   try {
3   // code
4 } catch (IOException e) {
5   MyException ex = new MyException("read file failed.");
6   ex.initCause(e);
7   throw ex;
8   }
9   }
```

捕获异常

在一个 try-catch 语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理

```
1   private static void readFile(String filePath) {
2   try {
3   // code
4   } catch (FileNotFoundException e) {
5   // handle FileNotFoundException
6 } catch (IOException e){
7   // handle IOException
8   }
9   }
```

同一个 catch 也可以捕获多种类型异常，用 | 隔开

```
1   private static void readFile(String filePath) {
2   try {
3   // code
4   } catch (FileNotFoundException | UnknownHostException e) {
5   // handle FileNotFoundException or UnknownHostException
6 } catch (IOException e){
7   // handle IOException
8   }
9   }
```

自定义异常

习惯上，定义一个异常类应包含两个构造函数，一个无参构造函数和一个带有详

细描述信息的构造函数（Throwable 的 toString 方法会打印这些详细信息，调试时很有用）

```
1   public class MyException extends Exception {
2   public MyException(){ }
3   public MyException(String msg){
4   super(msg);
5   }
6   // ...
7   }
```

try-catch-finally

当方法中发生异常，异常处之后的代码不会再执行，如果之前获取了一些本地资源需要释放，则需要在方法正常结束时和 catch 语句中都调用释放本地资源的代码，显得代码比较繁琐，finally 语句可以解决这个问题。

```
1 private static void readFile(String filePath) throws MyException {
2  File file = new File(filePath);
3  String result;
4  BufferedReader reader = null;
5  try {
6  reader = new BufferedReader(new FileReader(file));
7  while((result = reader.readLine())!=null) {
8  System.out.println(result);
9  }
10  } catch (IOException e) {
11  System.out.println("readFile method catch block.");
12  MyException ex = new MyException("read file failed.");
13  ex.initCause(e);
14  throw ex;
15  } finally {
16  System.out.println("readFile method finally block.");
17  if (null != reader) {
18  try {
19  reader.close();
20  } catch (IOException e) {
21  e.printStackTrace();
22  }
23  }
24  }
25 }
```

调用该方法时，读取文件时若发生异常，代码会进入 catch 代码块，之后进入 finally 代码块；若读取文件时未发生异常，则会跳过 catch 代码块直接进入 finally 代码块。所以无论代码中是否发生异常，fianlly 中的代码都会执行。 若 catch 代码块中包含 return 语句，finally 中的代码还会执行吗？将以上代码 中的 catch 子句修改如下：

```
1 catch (IOException e) {
2  System.out.println("readFile method catch block.");
3  return;
4 }
```

调用 readFile 方法，观察当 catch 子句中调用 return 语句时，finally 子句是 否执行

```
1 readFile method catch block.
2 readFile method finally block.
```

可见，即使 catch 中包含了 return 语句，finally 子句依然会执行。若 finally 中也包含 return 语句，finally 中的 return 会覆盖前面的 return.

try-with-resource

上面例子中，finally 中的 close 方法也可能抛出 IOException, 从而覆盖了原始异常。JAVA 7 提供了更优雅的方式来实现资源的自动释放，自动释放的资源需要是实现了 AutoCloseable 接口的类。

```
1   private static void tryWithResourceTest(){
2   try (Scanner scanner = new Scanner(new FileInputStream("c:/abc"),"UTF8")){
3   // code
4 } catch (IOException e){
5   // handle exception
6   }
7   }
```

try 代码块退出时，会自动调用 scanner.close 方法，和把 scanner.close 方法放在 finally 代码块中不同的是，若 scanner.close 抛出异常，则会被抑制，抛出的仍然为原始异常。被抑制的异常会由 addSusppressed 方法添加到原来的异常，如果想要获取被抑制的异常列表，可以调用 getSuppressed 方法来获取。

Java异常常见面试题

1. Error 和 Exception 区别是什么?

Error 类型的错误通常为虚拟机相关错误，如系统崩溃，内存不足，堆栈溢出 等，编译器不会对这类错误进行检测，JAVA 应用程序也不应对这类错误进行捕 获，一旦这类错误发生，通常应用程序会被终止，仅靠应用程序本身无法恢复；

Exception 类的错误是可以在应用程序中进行捕获并处理的，通常遇到这种错 误，应对其进行处理，使应用程序可以继续正常运行。

2. 运行时异常和一般异常(受检异常)区别是什么?

运行时异常包括 RuntimeException 类及其子类，表示 JVM 在运行期间可能出 现的异常。 Java 编译器不会检查运行时异常。

受检异常是Exception 中除 RuntimeException 及其子类之外的异常。 Java 编 译器会检查受检异常。

RuntimeException异常和受检异常之间的区别：是否强制要求调用者必须处 理此异常，如果强制要求调用者必须进行处理，那么就使用受检异常，否则就选择非受检异常(RuntimeException)。一般来讲，如果没有特殊的要求，我们建 议使用RuntimeException异常。

3. JVM 是如何处理异常的?

在一个方法中如果发生异常，这个方法会创建一个异常对象，并转交给 JVM， 该异常对象包含异常名称，异常描述以及异常发生时应用程序的状态。创建异常 对象并转交给 JVM 的过程称为抛出异常。可能有一系列的方法调用，终才进 入抛出异常的方法，这一系列方法调用的有序列表叫做调用栈。

JVM 会顺着调用栈去查找看是否有可以处理异常的代码，如果有，则调用异常 处理代码。当 JVM 发现可以处理异常的代码时，会把发生的异常传递给它。如 果 JVM 没有找到可以处理该异常的代码块，JVM 就会将该异常转交给默认的异 常处理器（默认处理器为 JVM 的一部分），默认异常处理器打印出异常信息并 终止应用程序。

4. throw 和 throws 的区别是什么?

Java 中的异常处理除了包括捕获异常和处理异常之外，还包括声明异常和抛出 异常，可以通过 throws 关键字在方法上声明该方法要抛出的异常，或者在方法 内部通过 throw 抛出异常对象。

throws 关键字和 throw 关键字在使用上的几点区别如下：

- throw 关键字用在方法内部，只能用于抛出一种异常，用来抛出方法或代码块中 的异常，受查异常和非受查异常都可以被抛出。
- throws 关键字用在方法声明上，可以抛出多个异常，用来标识该方法可能抛出 的异常列表。一个方法用 throws 标识了可能抛出的异常列表，调用该方法的方法中 必须包含可处理异常的代码，否则也要在方法签名中用 throws 关键字声明相应的异 常。

5. final、finally、finalize 有什么区别?

- final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方 法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。
- finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执 行的代码方法 finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用 来存放一些关闭资源的代码。
- finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类， Java 中允许使用 finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的 清理工作。

6. NoClassDefFoundError 和 ClassNotFoundException 区别?

NoClassDefFoundError 是一个 Error 类型的异常，是由 JVM 引起的，不应该 尝试捕获这个异常。

引起该异常的原因是 JVM 或 ClassLoader 尝试加载某类时在内存中找不到该类 的定义，该动作发生在运行期间，即编译时该类存在，但是在运行时却找不到 了，可能是变异后被删除了等原因导致；

ClassNotFoundException 是一个受查异常，需要显式地使用 try-catch 对其 进行捕获和处理，或在方法签名中用 throws 关键字进行声明。当使用 Class.forName, ClassLoader.loadClass 或 ClassLoader.findSystemClass 动 态加载类到内存的时候，通过传入的类路径参数没有找到该类，就会抛出该异常；另一种抛出该异常的可能原因是某个类已经由一个类加载器加载至内存中， 另一个加载器又尝试去加载它。

7. try-catch-finally 中哪个部分可以省略?

答：catch 可以省略

原因

更为严格的说法其实是：try只适合处理运行时异常，try+catch适合处理运行时 异常+普通异常。也就是说，如果你只用try去处理普通异常却不加以catch处 理，编译是通不过的，因为编译器硬性规定，普通异常如果选择捕获，则必须用 catch显示声明以便进一步处理。而运行时异常在编译时没有如此规定，所以 catch可以省略，你加上catch编译器也觉得无可厚非。

理论上，编译器看任何代码都不顺眼，都觉得可能有潜在的问题，所以你即使对 所有代码加上try，代码在运行期时也只不过是在正常运行的基础上加一层皮。 但是你一旦对一段代码加上try，就等于显示地承诺编译器，对这段代码可能抛 出的异常进行捕获而非向上抛出处理。如果是普通异常，编译器要求必须用 catch捕获以便进一步处理；如果运行时异常，捕获然后丢弃并且+finally扫尾 处理，或者加上catch捕获以便进一步处理。

至于加上finally，则是在不管有没捕获异常，都要进行的“扫尾”处理。

8. try-catch-finally 中，如果 catch 中 return 了， finally 还会执行吗?

答：会执行，在 return 前执行。

注意：在 finally 中改变返回值的做法是不好的，因为如果存在 finally 代码块， try中的 return 语句不会立马返回调用者，而是记录下返回值待 finally 代码块 执行完毕之后再向调用者返回其值，然后如果在 finally 中修改了返回值，就会 返回修改后的值。显然，在 finally 中返回或者修改返回值会对程序造成很大的 困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java 中也 可以通过提升编译器的语法检查级别来产生警告或错误。

代码示例1：

```
1 public static int getInt() {
2  int a = 10;
3  try {
4  System.out.println(a / 0);
5  a = 20;
6  } catch (ArithmeticException e) {
7  a = 30;
8  return a;
9  /*
10  * return a 在程序执行到这一步的时候，这里不是return a 而是 return 30；这个返 回路径就形成了
11  * 但是呢，它发现后面还有finally，所以继续执行finally的内容，a=40
12  * 再次回到以前的路径,继续走return 30，形成返回路径之后，这里的a就不是a变量 了，而是常量30
13  */
14  } finally {
15  a = 40;
16  }
17  return a;
18 }
```

执行结果：30

代码示例2：

```
1 public static int getInt() {
2  int a = 10;
3  try {
4  System.out.println(a / 0);
5  a = 20;
6  } catch (ArithmeticException e) {
7  a = 30;
8  return a;
9  } finally {
10  a = 40;
11  //如果这样，就又重新形成了一条返回路径，由于只能通过1个return返回，所以这里直 接返回40
12  return a;
13  }
14
15 }
```

执行结果：40

9. 类 ExampleA 继承 Exception，类 ExampleB 继承 ExampleA。

有如下代码片断：

```
try {
 throw new ExampleB("b")
} catch（ExampleA e）{
 System.out.println("ExampleA");
} catch（Exception e）{
 System.out.println("Exception");
}
```

请问执行此段代码的输出是什么？

答：

输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类 型]，抓取 ExampleA 类型异常的 catch 块能够抓住 try 块中抛出的 ExampleB 类型的异常）

面试题 - 说出下面代码的运行结果。（此题的出处是《Java 编程思想》一书）

```
class Annoyance extends Exception {
}
class Sneeze extends Annoyance {
}
class Human {
 public static void main(String[] args)
 throws Exception {
 try {
 try {
  throw new Sneeze();
  } catch ( Annoyance a ) {
  System.out.println("Caught Annoyance");
  throw a;
  }
  } catch ( Sneeze s ) {
  System.out.println("Caught Sneeze");
  return ;
  } finally {
  System.out.println("Hello World!");
  }
  }
}
```

结果

```
Caught Annoyance
Caught Sneeze
Hello World!
```

10. 常见的 RuntimeException 有哪些?

- ClassCastException(类转换异常)
- IndexOutOfBoundsException(数组越界)
- NullPointerException(空指针)
- ArrayStoreException(数据存储异常，操作数组时类型不一致)
- 还有IO操作的BufferOverflowException异常

11. Java常见异常有哪些

java.lang.IllegalAccessError：违法访问错误。当一个应用试图访问、修改某个类的域（Field）或者调用其方法，但是又违反域或方法的可见性声明，则抛出该异常。

java.lang.InstantiationError：实例化错误。当一个应用试图通过Java的new操作符构造一个抽象类或者接口时抛出该异常.

java.lang.OutOfMemoryError：内存不足错误。当可用内存不足以让Java虚拟机分配给一个对象时抛出该错误。

java.lang.StackOverflowError：堆栈溢出错误。当一个应用递归调用的层次太深而导致堆栈溢出或者陷入死循环时抛出该错误。

java.lang.ClassCastException：类造型异常。假设有类A和B（A不是B的父类或子类），O是A的实例，那么当强制将O构造为类B的实例时抛出该异常。该异常经常被称为强制类型转换异常。

java.lang.ClassNotFoundException：找不到类异常。当应用试图根据字符串形式的类名构造类，而在遍历CLASSPAH之后找不到对应名称的class文件时，抛出该异常。 java.lang.ArithmeticException：算术条件异常。譬如：整数除零等。

java.lang.ArrayIndexOutOfBoundsException：数组索引越界异常。当对数组的索引值为负数或大于等于数组大小时抛出。

java.lang.IndexOutOfBoundsException：索引越界异常。当访问某个序列的索引值小于0或大于等于序列大小时，抛出该异常。 java.lang.InstantiationException：实例化异常。当试图通过newInstance()方法创建某个类的实例，而该类是一个抽象类或接口时，抛出该异常。

java.lang.NoSuchFieldException：属性不存在异常。当访问某个类的不存在的属性时抛出该异常。

java.lang.NoSuchMethodException：方法不存在异常。当访问某个类的不存在的方法时抛出该异常。

java.lang.NullPointerException：空指针异常。当应用试图在要求使用对象的地方使用了null时，抛出该异常。譬如：调用null对象的实例方法、访问null对象的属性、计算null对象的长度、使用throw语句抛出null等等。

java.lang.NumberFormatException：数字格式异常。当试图将一个String转换为指定的数字类型，而该字符串确不满足数字类型要求的格式时，抛出该异常。

java.lang.StringIndexOutOfBoundsException：字符串索引越界异常。当使用索引值访问某个字符串中的字符，而该索引值小于0或大于等于序列大小时,抛出该异常

Java异常处理最佳实践

在 Java 中处理异常并不是一个简单的事情。不仅仅初学者很难理解，即使一些有经验的开发者也需要花费很多时间来思考如何处理异常，包括需要处理哪些异常，怎样处理等等。这也是绝大多数开发团队都会制定一些规则来规范进行异常处理的原因。而团队之间的这些规范往往是截然不同的。本文给出几个被很多团队使用的异常处理 佳实践。

1. 在 finally 块中清理资源或者使用 try-with-resource 语句

当使用类似InputStream这种需要使用后关闭的资源时，一个常见的错误就是在 try块的 后关闭资源。

```
1   public void doNotCloseResourceInTry() {
2   FileInputStream inputStream = null;
3   try {
4   File file = new File("./tmp.txt");
5   inputStream = new FileInputStream(file);
6   // use the inputStream to read a file
7   // do NOT do this
8   inputStream.close();
9   } catch (FileNotFoundException e) {
10  log.error(e);
11  } catch (IOException e) {
12  log.error(e);
13  }
14  }
```

问题就是，只有没有异常抛出的时候，这段代码才可以正常工作。try 代码块内代码会正常执行，并且资源可以正常关闭。但是，使用 try 代码块是有原因的，一般调用一个或多个可能抛出异常的方法，而且，你自己也可能会抛出一个异常，这意味着代码可能不会执行到 try 代码块的 后部分。结果就是，你并没有关闭资源。

所以，你应该把清理工作的代码放到 finally 里去，或者使用 try-with-resource 特性。

1.1 使用 finally 代码块

与前面几行 try 代码块不同，finally 代码块总是会被执行。不管 try 代码块成功

执行之后还是你在 catch 代码块中处理完异常后都会执行。因此，你可以确保你清理了所有打开的资源。

```
1 public void closeResourceInFinally() {
2  FileInputStream inputStream = null;
3  try {
4  File file = new File("./tmp.txt");
5  inputStream = new FileInputStream(file);
6  // use the inputStream to read a file
7  } catch (FileNotFoundException e) {
8  log.error(e);
9  } finally {
10  if (inputStream != null) {
11  try {
12  inputStream.close();
13  } catch (IOException e) {
14  log.error(e);
15  }
16  }
17  }
18 }
```

1.2 Java 7 的 try-with-resource 语法

如果你的资源实现了 AutoCloseable 接口，你可以使用这个语法。大多数的 Java 标准资源都继承了这个接口。当你在 try 子句中打开资源，资源会在 try 代码块执行后或异常处理后自动关闭。

```
1   public void automaticallyCloseResource() {
2   File file = new File("./tmp.txt");
3   try (FileInputStream inputStream = new FileInputStream(file);) {
4   // use the inputStream to read a file
5   } catch (FileNotFoundException e) {
6   log.error(e);
7 } catch (IOException e) {
8   log.error(e);
9   }
10  }
11
```

2. 优先明确的异常

你抛出的异常越明确越好，永远记住，你的同事或者几个月之后的你，将会调用你的方法并且处理异常。

因此需要保证提供给他们尽可能多的信息。这样你的 API 更容易被理解。你的方法的调用者能够更好的处理异常并且避免额外的检查。因此，总是尝试寻找 适合你的异常事件的类，例如，抛出一个

NumberFormatException 来替换一个 IllegalArgumentException 。避免抛出一个不明确的异常。

```
1 public void doNotDoThis() throws Exception {
2 ...
3   }
4   public void doThis() throws NumberFormatException {
5 ...
6 }
7
```

3. 对异常进行文档说明

当在方法上声明抛出异常时，也需要进行文档说明。目的是为了给调用者提供尽可能多的信息，从而可以更好地避免或处理异常。

在 Javadoc 添加 @throws 声明，并且描述抛出异常的场景。

```
1 public void doSomething(String input) throws MyBusinessException {
2 ...
3 }
```

4. 使用描述性消息抛出异常

在抛出异常时，需要尽可能精确地描述问题和相关信息，这样无论是打印到日志中还是在监控工具中，都能够更容易被人阅读，从而可以更好地定位具体错误信息、错误的严重程度等。

但这里并不是说要对错误信息长篇大论，因为本来 Exception 的类名就能够反映错误的原因，因此只需要用一到两句话描述即可。

如果抛出一个特定的异常，它的类名很可能已经描述了这种错误。所以，你不需要提供很多额外的信息。一个很好的例子是 NumberFormatException 。当你以错误的格式提供 String 时，它将被 java.lang.Long 类的构造函数抛出。

```
1   try {
2   new Long("xyz");
3 } catch (NumberFormatException e) {
4   log.error(e);
5   }
```

5. 优先捕获最具体的异常

大多数 IDE 都可以帮助你实现这个 佳实践。当你尝试首先捕获较不具体的异常时，它们会报告无法访问的代码块。

但问题在于，只有匹配异常的第一个 catch 块会被执行。 因此，如果首先捕获

IllegalArgumentException ，则永远不会到达应该处理更具体的 NumberFormatException 的 catch 块，因为它是 IllegalArgumentException 的子类。总是优先捕获 具体的异常类，并将不太具体的 catch 块添加到列表的末尾。你可以在下面的代码片断中看到这样一个 try-catch 语句的例子。 第一个 catch 块处理所有 NumberFormatException 异常，第二个处理所有非

NumberFormatException 异常的IllegalArgumentException 异常。

```
1 public void catchMostSpecificExceptionFirst() {
2  try {
3  doSomething("A message");
4   } catch (NumberFormatException e) {
5   log.error(e);
6   } catch (IllegalArgumentException e) {
7   log.error(e)
8   }
9   }
```

6. 不要捕获 Throwable 类

Throwable 是所有异常和错误的超类。你可以在 catch 子句中使用它，但是你

永远不应该这样做!

如果在 catch 子句中使用 Throwable ，它不仅会捕获所有异常，也将捕获所有的错误。JVM 抛出错误，指出不应该由应用程序处理的严重问题。 典型的例子是 OutOfMemoryError 或者 StackOverflowError 。两者都是由应用程序控制之外的情况引起的，无法处理。

所以， 好不要捕获 Throwable ，除非你确定自己处于一种特殊的情况下能够处理错误。

```
1   public void doNotCatchThrowable() {
2   try {
3   // do something
4   } catch (Throwable t) {
5 // don't do this!
6   }
7   }
```

7. 不要忽略异常

很多时候，开发者很有自信不会抛出异常，因此写了一个catch块，但是没有做任何处理或者记录日志。

```
1   public void doNotIgnoreExceptions() {
2   try {
3   // do something
4 } catch (NumberFormatException e) {
5   // this will never happen
6   }
7   }
```

但现实是经常会出现无法预料的异常，或者无法确定这里的代码未来是不是会改动(删除了阻止异常抛出的代码)，而此时由于异常被捕获，使得无法拿到足够的错误信息来定位问题。合理的做法是至少要记录异常的信息。

```
1   public void logAnException() {
2   try {
3   // do something
4 } catch (NumberFormatException e) {
5   log.error("This should never happen: " + e);
6   }
7   }
```

8. 不要记录并抛出异常

这可能是本文中 常被忽略的 佳实践。可以发现很多代码甚至类库中都会有捕获异常、记录日志并再次抛出的逻辑。如下：

```
1   try {
2   new Long("xyz");
3 } catch (NumberFormatException e) {
4   log.error(e);
5   throw e;
6   }
```

这个处理逻辑看着是合理的。但这经常会给同一个异常输出多条日志。如下：

```
1 17:44:28,945 ERROR TestExceptionHandling:65 - java.lang.NumberFormatExcep tion:
For input string: "xyz"
2 Exception in thread "main" java.lang.NumberFormatException: For input str ing:
"xyz"
3 at java.lang.NumberFormatException.forInputString(NumberFormatException.j
ava:65)
4 at java.lang.Long.parseLong(Long.java:589)
5 at java.lang.Long.(Long.java:965)
6 at com.stackify.example.TestExceptionHandling.logAndThrowException(TestEx
ceptionHandling.java:63)
7 at
com.stackify.example.TestExceptionHandling.main(TestExceptionHandling.java:5
8)
```

如上所示，后面的日志也没有附加更有用的信息。如果想要提供更加有用的信息，那么可以将异常包装为自定义异常。

```
 public void wrapException(String input) throws MyBusinessException {
3   // do something
4   } catch (NumberFormatException e) {
5   throw new MyBusinessException("A message that describes the error.", e);
6   }
7   }
```

因此，仅仅当想要处理异常时才去捕获，否则只需要在方法签名中声明让调用者去处理。

9. 包装异常时不要抛弃原始的异常

捕获标准异常并包装为自定义异常是一个很常见的做法。这样可以添加更为具体的异常信息并能够做针对的异常处理。

在你这样做时，请确保将原始异常设置为原因（注：参考下方代码

NumberFormatException e 中的原始异常 e ）。Exception 类提供了特殊的

构造函数方法，它接受一个 Throwable 作为参数。否则，你将会丢失堆栈跟踪和原始异常的消息，这将会使分析导致异常的异常事件变得困难。

```
1 public void wrapException(String input) throws MyBusinessException {
2   try {
3   // do something
4 } catch (NumberFormatException e) {
5   throw new MyBusinessException("A message that describes the error.", e);
6   }
7   }
8
```

10. 不要使用异常控制程序的流程

不应该使用异常控制应用的执行流程，例如，本应该使用if语句进行条件判断的情况下，你却使用异常处理，这是非常不好的习惯，会严重影响应用的性能。

11. 使用标准异常

如果使用内建的异常可以解决问题，就不要定义自己的异常。Java API 提供了上百种针对不同情况的异常类型，在开发中首先尽可能使用 Java API 提供的异常，如果标准的异常不能满足你的要求，这时候创建自己的定制异常。尽可能得使用标准异常有利于新加入的开发者看懂项目代码。

12. 异常会影响性能

异常处理的性能成本非常高，每个 Java 程序员在开发时都应牢记这句话。创建一个异常非常慢，抛出一个异常又会消耗1~5ms，当一个异常在应用的多个层级之间传递时，会拖累整个应用的性能。

- 仅在异常情况下使用异常；
- 在可恢复的异常情况下使用异常；尽管使用异常有利于 Java 开发，但是在应用中 好不要捕获太多的调用栈，因为在很多情况下都不需要打印调用栈就知道哪里出错了。因此，异常消息应该提供恰到好处的信息。

13. 总结

综上所述，当你抛出或捕获异常的时候，有很多不同的情况需要考虑，而且大部分事情都是为了改善代码的可读性或者 API 的可用性。

异常不仅仅是一个错误控制机制，也是一个通信媒介。因此，为了和同事更好的合作，一个团队必须要制定出一个 佳实践和规则，只有这样，团队成员才能理解这些通用概念，同时在工作中使用它。

异常处理-阿里巴巴Java开发手册

1. 【强制】Java 类库中定义的可以通过预检查方式规避的

RuntimeException异常不应该通过catch 的方式来处理，比如：

NullPointerException，IndexOutOfBoundsException等等。 说明：无法通过预检查的异常除外，比如，在解析字符串形式的数字时，可能存在数字格式错误，不得不通过catch NumberFormatException来实现。 正例：if (obj != null) {...} 反例：try { obj.method(); } catch

(NullPointerException e) {...}

2. 【强制】异常不要用来做流程控制，条件控制。 说明：异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式要低很多。
3. 【强制】catch时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的catch尽可能进行区分异常类型，

再做对应的异常处理。 说明：对大段代码进行try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利于定位问题，这是一种不负责任的表现。 正例：用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于简单，在程序上作出分门别类的判断，并提示给用户。

4. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。 外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。
5. 【强制】有try块放到了事务代码中，catch异常后，如果需要回滚事务，一定要注意手动回滚事务。
6. 【强制】finally块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。 说明：如果JDK7及以上，可以使用try-with-resources方式。
7. 【强制】不要在finally块中使用return。 说明：try块中的return语句执行成功后，并不马上返回，而是继续执行finally块中的语句，如果此处存在return语句，则在此直接返回，无情丢弃掉try块中的返回点。 反例：

```
1   private int x = 0;
2   public int checkReturn() {
3   try {
4   // x等于1，此处不返回
5   return ++x;
6   } finally {
7   // 返回的结果是2
8   return ++x;
9   }
10  }
```

1. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。 说明：如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。
2. 【强制】在调用RPC、二方包、或动态生成类的相关方法时，捕捉异常必须使用Throwable类来进行拦截。 说明：通过反射机制来调用方法，如果找不到方法，抛出NoSuchMethodException。什么情况会抛出

NoSuchMethodError呢？二方包在类冲突时，仲裁机制可能导致引入非预期的版本使类的方法签名不匹配，或者在字节码修改框架（比如：

ASM）动态创建或修改类时，修改了相应的方法签名。这些情况，即使代码编译期是正确的，但在代码运行期时，会抛出NoSuchMethodError。

3. 【推荐】方法的返回值可以为null，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回null值。 说明：本手册明确防止NPE是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回null的情况。
4. 【推荐】防止NPE，是程序员的基本修养，注意NPE产生的场景： 1） 返回类型为基本数据类型，return包装数据类型的对象时，自动拆箱有可能产生NPE。 反例：public int f() { return Integer对象}， 如果为null，自动解箱抛NPE。 2） 数据库的查询结果可能为null。 3） 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。 4） 远程调用返回对象时，一律要求进行空指针判断，防止NPE。 5） 对于Session中获取的数据，建议进行NPE检查，避免空指针。 6） 级联调用

obj.getA().getB().getC()；一连串调用，易产生NPE。

正例：使用JDK8的Optional类来防止NPE问题。

5. 【推荐】定义时区分unchecked / checked 异常，避免直接抛出new

RuntimeException()，更不允许抛出Exception或者Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：

DAOException / ServiceException等。

6. 【参考】对于公司外的http/api开放接口必须使用“错误码”；而应用内部推荐异常抛出；跨应用间RPC调用优先考虑使用Result方式，封装 isSuccess()方法、“错误码”、“错误简短信息”。 说明：关于RPC方法返回方式使用Result方式的理由： 1）使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。 2）如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。
7. 【参考】避免出现重复的代码（Don’t Repeat Yourself），即DRY原则。 说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。 正例：一个类中有多个public方法，都需要进行数行相同的参数校验操作，这个时候请抽取： private boolean checkParam(DTO dto) {...}

Tomcat是什么?

Tomcat 服务器Apache软件基金会项目中的一个核心项目，是一个免费的开放 源代码的Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问 用户不是很多的场合下被普遍使用，是开发和调试JSP 程序的首选。

Tomcat的缺省端口是多少，怎么修改

找到Tomcat目录下的conf文件夹

进入conf文件夹里面找到server.xml文件

打开server.xml文件

在server.xml文件里面找到下列信息

把Connector标签的8080端口改成你想要的端口

```
1 <Service name="Catalina">
2 <Connector port="8080" protocol="HTTP/1.1"
3  connectionTimeout="20000"
4  redirectPort="8443" />
```

tomcat 有哪几种Connector 运行模式(优化)?

下面，我们先大致了解Tomcat Connector的三种运行模式。

- **BIO：同步并阻塞** 一个线程处理一个请求。缺点：并发量高时，线程数较 多，浪费资源。
  Tomcat7或以下，在Linux系统中默认使用这种方式。
  配制项：protocol=”HTTP/1.1”
- **NIO：同步非阻塞IO** 利用Java的异步IO处理，可以通过少量的线程处理大量的请求，可以复用同一个线程处理多个connection(多路复用)。
  Tomcat8在Linux系统中默认使用这种方式。
  Tomcat7必须修改Connector配置来启动。
  配制项：protocol=”org.apache.coyote.http11.Http11NioProtocol”
  备注：我们常用的Jetty，Mina，ZooKeeper等都是基于java nio实现.
- **APR：**即Apache Portable Runtime，从操作系统层面解决io阻塞问 题。**AIO方式，异步非阻塞IO**(Java NIO2又叫AIO) 主要与NIO的区别 主要是操作系统的底层区别.可以做个比喻:比作快递，NIO就是网购后要自 己到官网查下快递是否已经到了(可能是多次)，然后自己去取快递；AIO就 是快递员送货上门了(不用关注快递进度)。
  配制项：protocol=”org.apache.coyote.http11.Http11AprProtocol”
  备注：需在本地服务器安装APR库。Tomcat7或Tomcat8在Win7或以上的系统 中启动默认使用这种方式。Linux如果安装了apr和native，Tomcat直接启动就 支持apr。

Tomcat有几种部署方式?

在Tomcat中部署Web应用的方式主要有如下几种：

- 利用Tomcat的自动部署。
  把web应用拷贝到webapps目录。Tomcat在启动时会加载目录下的应用，并将 编译后的结果放入work目录下。
- 使用Manager App控制台部署。
  在tomcat主页点击“Manager App” 进入应用管理控制台，可以指定一个 web应用的路径或war文件。
- 修改conf/server.xml文件部署。
  修改conf/server.xml文件，增加Context节点可以部署应用。
- 增加自定义的Web部署文件。
  在conf/Catalina/localhost/ 路径下增加 xyz.xml文件，内容是Context节点， 可以部署应用

tomcat容器是如何创建servlet类实例？用到了什么 原理?

1. 当容器启动时，会读取在webapps目录下所有的web应用中的web.xml 文件，然后对 xml文件进行解析，并读取servlet注册信息。然后，将每个 应用中注册的servlet类都进行加载，并通过 反射的方式实例化。（有时候 也是在第一次请求时实例化）
2. 在servlet注册时加上1如果为正数，则在一开始就实例化，如果不写或 为负数，则第一次请求实例化。

Tomcat工作模式

Tomcat作为servlet容器，有三种工作模式：

1. 独立的servlet容器，servlet容器是web服务器的一部分；
2. 进程内的servlet容器，servlet容器是作为web服务器的插件和java容器的实 现，web服务器插件在内部地址空间打开一个jvm使得java容器在内部得以运行。反 应速度快但伸缩性不足；
3. 进程外的servlet容器，servlet容器运行于web服务器之外的地址空间，并作 为web服务器的插件和java容器实现的结合。反应时间不如进程内但伸缩性和稳定性 比进程内优； 进入Tomcat的请求可以根据Tomcat的工作模式分为如下两类：

- Tomcat作为应用程序服务器：请求来自于前端的web服务器，这可能是 Apache, IIS, Nginx等；
- Tomcat作为独立服务器：请求来自于web浏览器；

面试时问到Tomcat相关问题的几率并不高，正式因为如此，很多人忽略了对 Tomcat相关技能的掌握，下面这一篇文章整理了Tomcat相关的系统架构，介 绍了Server、Service、Connector、Container之间的关系，各个模块的功 能，可以说把这几个掌握住了，Tomcat相关的面试题你就不会有任何问题了！

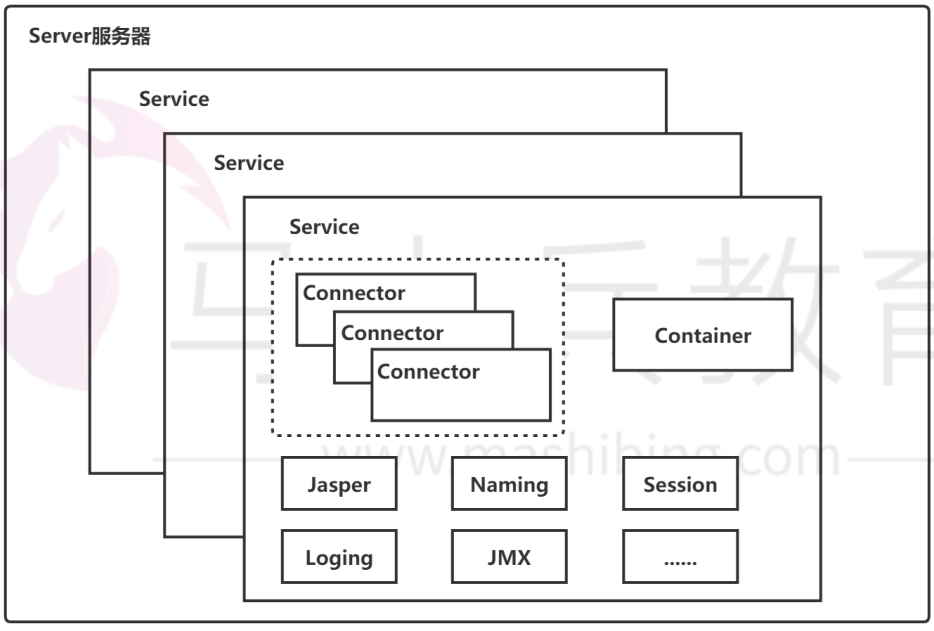
另外，在面试的时候你还要有意识无意识的往Tomcat这个地方引，就比如说常 见的Spring MVC的执行流程，一个URL的完整调用链路，这些相关的题目你是 可以往Tomcat处理请求的这个过程去说的！掌握了Tomcat这些技能，面试官 一定会佩服你的！

学了本章之后你应该明白的是：

- Server、Service、Connector、Container四大组件之间的关系和联系，以及他 们的主要功能点；
- Tomcat执行的整体架构，请求是如何被一步步处理的；
- Engine、Host、Context、Wrapper相关的概念关系；
- Container是如何处理请求的；
- Tomcat用到的相关设计模式；

Tomcat顶层架构

俗话说，站在巨人的肩膀上看世界，一般学习的时候也是先总览一下整体，然后 逐个部分个个击破，最后形成思路，了解具体细节，Tomcat的结构很复杂，但 是 Tomcat 非常的模块化，找到了 Tomcat 最核心的模块，问题才可以游刃而 解，了解了 Tomcat 的整体架构对以后深入了解 Tomcat 来说至关重要！ 先上一张Tomcat的顶层结构图（图A），如下：

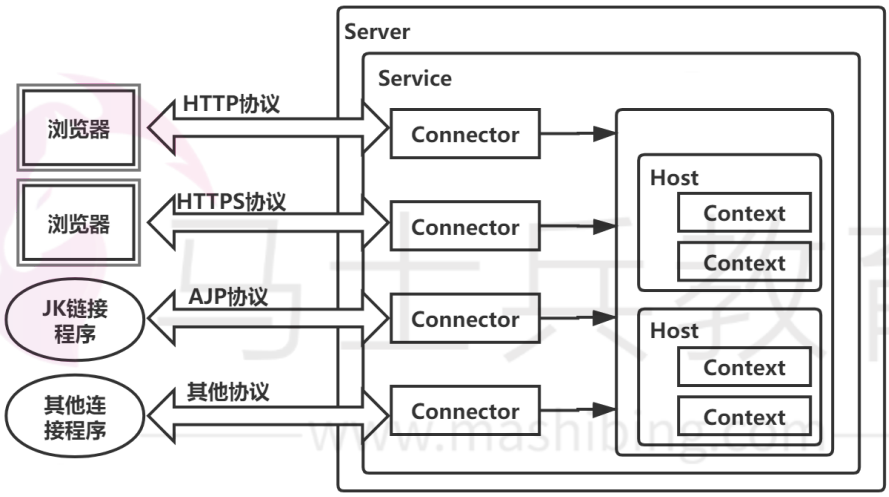


Tomcat中最顶层的容器是Server，代表着整个服务器，从上图中可以看出，一 个Server可以包含至少一个Service，即可以包含多个Service，用于具体提供服 务。

Service主要包含两个部分：Connector和Container。从上图中可以看出 Tomcat 的心脏就是这两个组件，他们的作用如下：

- Connector用于处理连接相关的事情，并提供Socket与Request请求和 Response响应相关的转化;
- Container用于封装和管理Servlet，以及具体处理Request请求;

一个Tomcat中只有一个Server，一个Server可以包含多个Service，一个 Service只有一个Container，但是可以有多个Connectors，这是因为一个服务 可以有多个连接，如同时提供Http和Https链接，也可以提供向相同协议不同端 口的连接，示意图如下（Engine、Host、Context下面会说到）：



多个 Connector 和一个 Container 就形成了一个 Service，有了 Service 就可 以对外提供服务了，但是 Service 还要一个生存的环境，必须要有人能够给她生 命、掌握其生死大权，那就非 Server 莫属了！所以整个 Tomcat 的生命周期由 Server 控制。

另外，上述的包含关系或者说是父子关系，都可以在tomcat的conf目录下的 server.xml配置文件中看出，下图是删除了注释内容之后的一个完整的 server.xml配置文件（Tomcat版本为8.0）

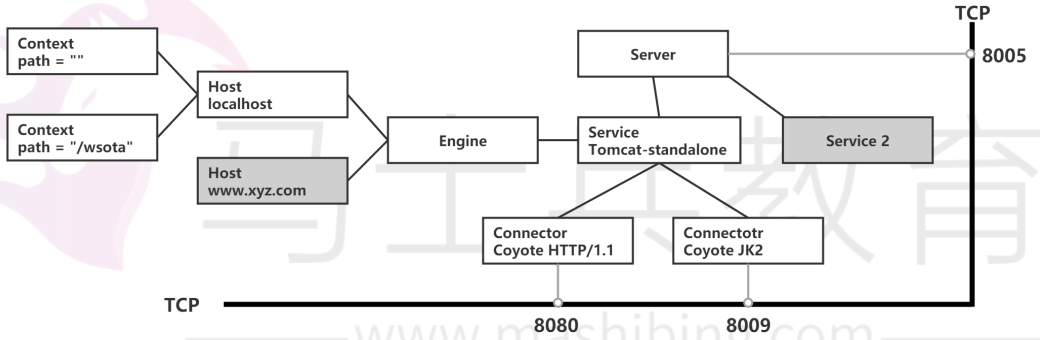
```
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
  <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
  <Listener className="org.apache.catalina.core.JasperListener" />
  <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />

  <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
  </GlobalNamingResources>

  <Service name="Catalina">
    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
    <Engine name="Catalina" defaultHost="localhost">
      <Realm className="org.apache.catalina.realm.LockOutRealm">
        <Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="UserDatabase"/>
      </Realm>
      <Host name="localhost"  appBase="webapps" unpackWARs="true" autoDeploy="true">
        <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
               prefix="localhost_access_log." suffix=".txt"
               pattern="%h %l %u %t &amp;quot;%r&amp;quot; %s %b" />
      </Host>
    </Engine>
  </Service>
</Server>
```

详细的配置文件内容可以到Tomcat官网查看：Tomcat配置文件

上边的配置文件，还可以通过下边的一张结构图更清楚的理解：



Server标签设置的端口号为8005，shutdown=”SHUTDOWN”，表示在 8005端口监听“SHUTDOWN”命令，如果接收到了就会关闭Tomcat。一个 Server有一个Service，当然还可以进行配置，一个Service有多个Connector， Service左边的内容都属于Container的，Service下边是Connector。

Tomcat顶层架构小结

1. Tomcat中只有一个Server，一个Server可以有多个Service，一个 Service可以有多个Connector和一个Container；
2. Server掌管着整个Tomcat的生死大权；
3. Service 是对外提供服务的；
4. Connector用于接受请求并将请求封装成Request和Response来具体 处理；
5. Container用于封装和管理Servlet，以及具体处理request请求；

知道了整个Tomcat顶层的分层架构和各个组件之间的关系以及作用，对于绝大 多数的开发人员来说Server和Service对我们来说确实很远，而我们开发中绝大 部分进行配置的内容是属于Connector和Container的，所以接下来介绍一下 Connector和Container。

Connector和Container的微妙关系

由上述内容我们大致可以知道一个请求发送到Tomcat之后，首先经过Service然 后会交给我们的Connector，Connector用于接收请求并将接收的请求封装为 Request和Response来具体处理，Request和Response封装完之后再交由 Container进行处理，Container处理完请求之后再返回给Connector，最后在 由Connector通过Socket将处理的结果返回给客户端，这样整个请求的就处理 完了！

Connector最底层使用的是Socket来进行连接的，Request和Response是按照 HTTP协议来封装的，所以Connector同时需要实现TCP/IP协议和HTTP协议！

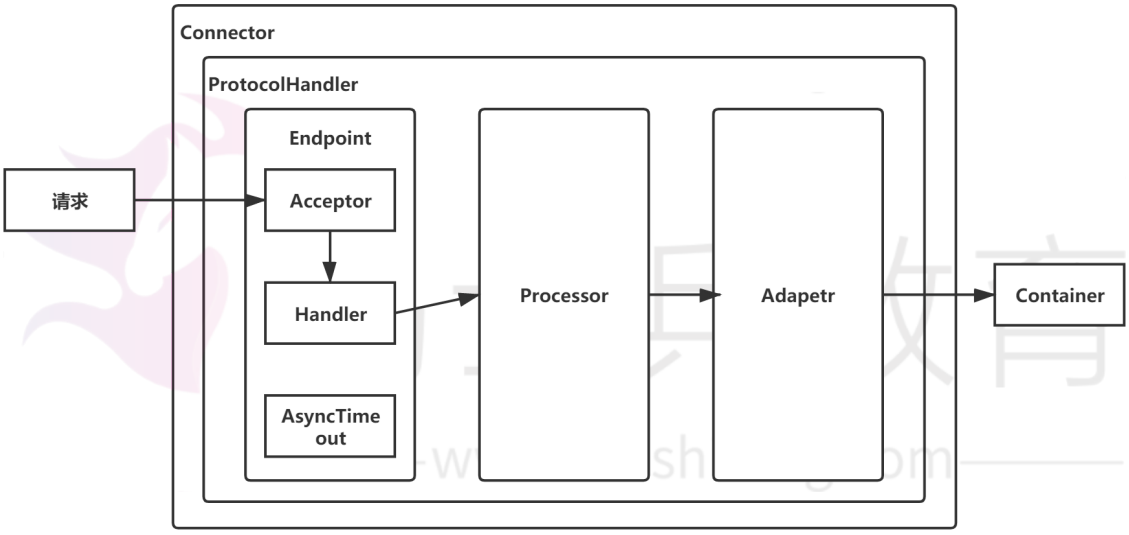
Tomcat既然需要处理请求，那么肯定需要先接收到这个请求，接收请求这个东 西我们首先就需要看一下Connector！

Connector架构分析

Connector用于接受请求并将请求封装成Request和Response，然后交给 Container进行处理，Container处理完之后在交给Connector返回给客户端。 因此，我们可以把Connector分为四个方面进行理解：

1. Connector如何接受请求的?
2. 如何将请求封装成Request和Response的?
3. 封装完之后的Request和Response如何交给Container进行处理的?

4. Container处理完之后如何交给Connector并返回给客户端的？ 首先看一下Connector的结构图，如下所示：



Connector就是使用ProtocolHandler来处理请求的，不同的ProtocolHandler 代表不同的连接类型，比如：Http11Protocol使用的是普通Socket来连接的， Http11NioProtocol使用的是NioSocket来连接的。

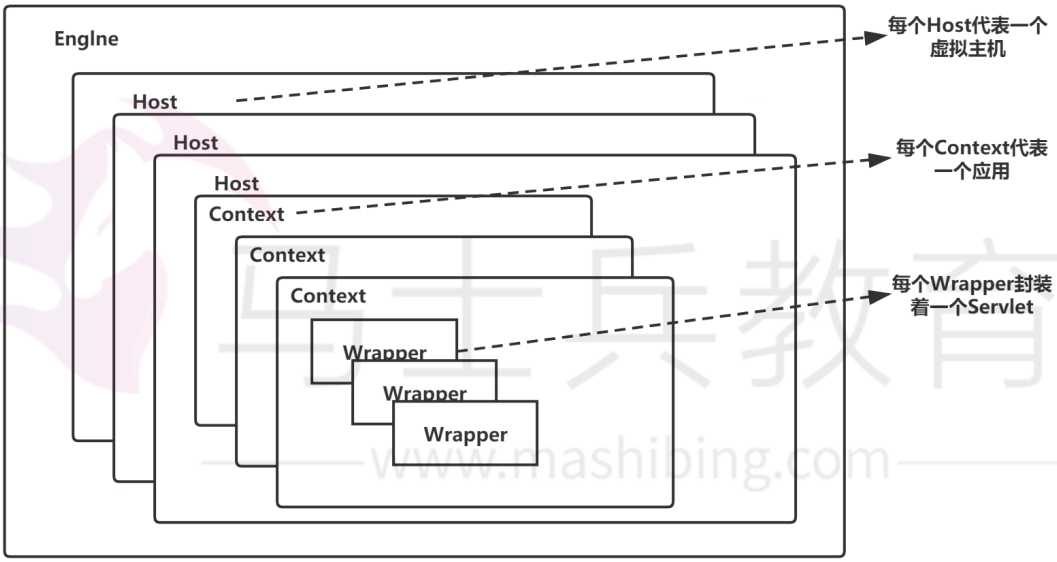
其中ProtocolHandler由包含了三个部件：Endpoint、Processor、Adapter。

1. Endpoint用来处理底层Socket的网络连接，Processor用于将 Endpoint接收到的Socket封装成Request，Adapter用于将Request交给 Container进行具体的处理。
2. Endpoint由于是处理底层的Socket网络连接，因此Endpoint是用来实 现TCP/IP协议的，而Processor用来实现HTTP协议的，Adapter将请求适 配到Servlet容器进行具体的处理。
3. Endpoint的抽象实现AbstractEndpoint里面定义的Acceptor和 AsyncTimeout两个内部类和一个Handler接口。Acceptor用于监听请 求，AsyncTimeout用于检查异步Request的超时，Handler用于处理接收 到的Socket，在内部调用Processor进行处理。

至此，我们应该很轻松的回答1，2，3的问题了，但是4还是不知道，那么我们 就来看一下Container是如何进行处理的以及处理完之后是如何将处理完的结果 返回给Connector的?

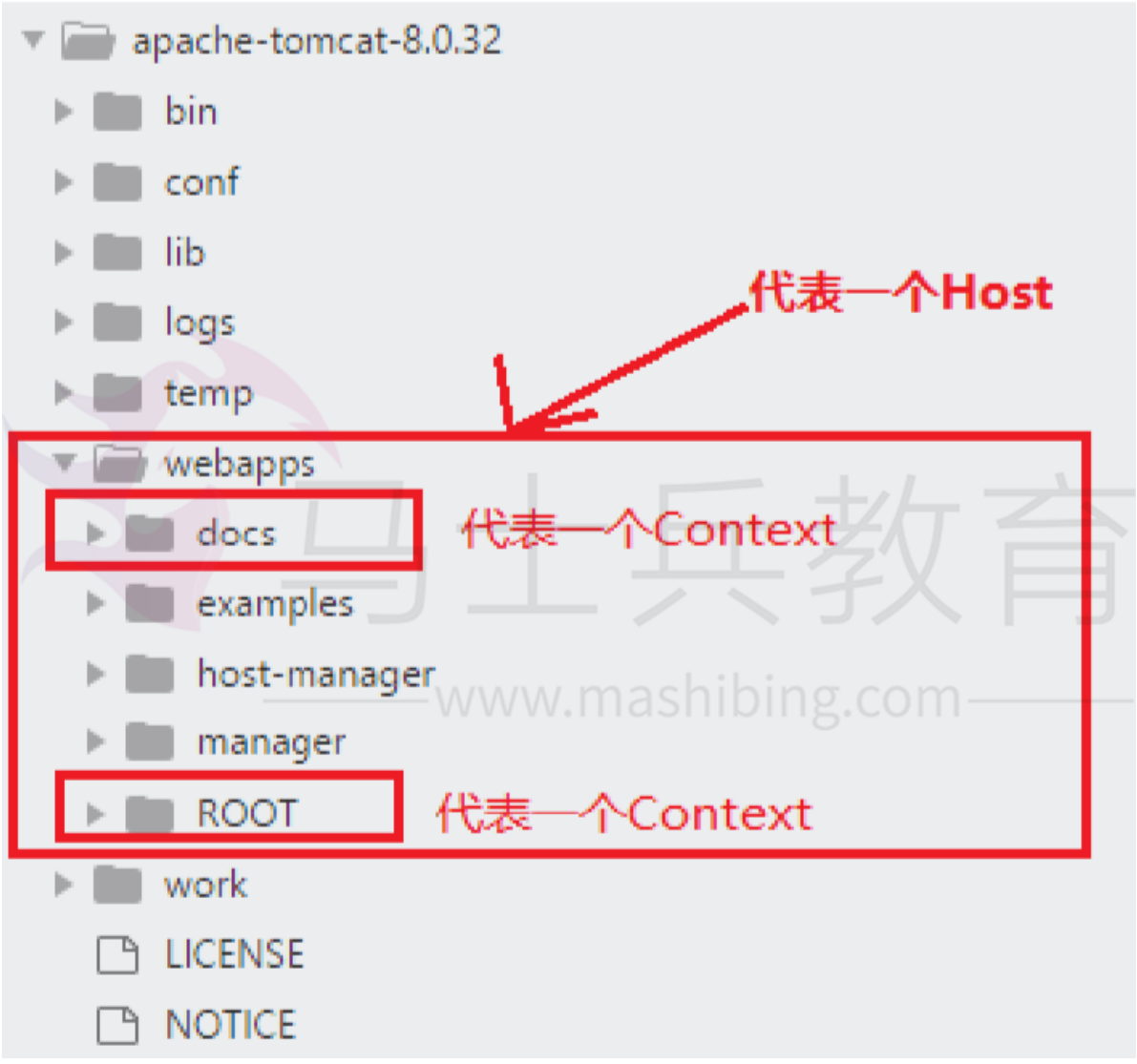
Container架构分析

Container用于封装和管理Servlet，以及具体处理Request请求，在Container 内部包含了4个子容器，结构图如下：



4个子容器的作用分别是：

1. Engine：引擎，用来管理多个站点，一个Service最多只能有一个 Engine；
2. Host：代表一个站点，也可以叫虚拟主机，通过配置Host就可以添加 站点；
3. Context：代表一个应用程序，对应着平时开发的一套程序，或者一个 WEB-INF目录以及下面的web.xml文件；
4. Wrapper：每一Wrapper封装着一个Servlet； 下面找一个Tomcat的文件目录对照一下，如下图所示：



Context和Host的区别是Context表示一个应用，我们的Tomcat中默认的配置 下webapps下的每一个文件夹目录都是一个Context，其中ROOT目录中存放着 主应用，其他目录存放着子应用，而整个webapps就是一个Host站点。

我们访问应用Context的时候，如果是ROOT下的则直接使用域名就可以访问， 例如：www.baidu.com，如果是Host（webapps）下的其他应用，则可以使 用www.baidu.com/docs进行访问，当然默认指定的根应用（ROOT）是可以 进行设定的，只不过Host站点下默认的主应用是ROOT目录下的。

看到这里我们知道Container是什么，但是还是不知道Container是如何进行请 求处理的以及处理完之后是如何将处理完的结果返回给Connector的？别急！下 边就开始探讨一下Container是如何进行处理的！

Container如何处理请求的

Container处理请求是使用Pipeline-Valve管道来处理的！（Valve是阀门之 意）

Pipeline-Valve是**责任链模式**，责任链模式是指在一个请求处理的过程中有很多 处理者依次对请求进行处理，每个处理者负责做自己相应的处理，处理完之后将 处理后的结果返回，再让下一个处理者继续处理。

但是！Pipeline-Valve使用的责任链模式和普通的责任链模式有些不同！区别主 要有以下两点：

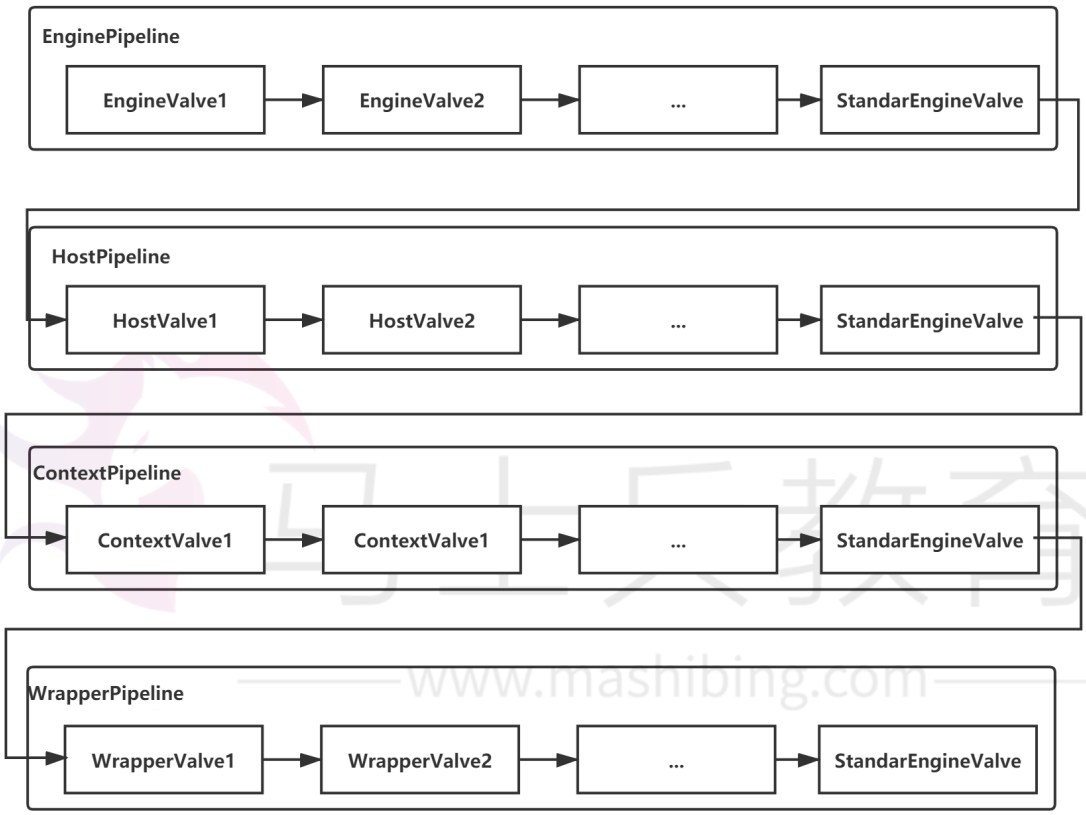
每个Pipeline都有特定的Valve，而且是在管道的最后一个执行，这个 Valve叫做BaseValve，BaseValve是不可删除的；

在上层容器的管道的BaseValve中会调用下层容器的管道。

我们知道Container包含四个子容器，而这四个子容器对应的BaseValve分别

在：StandardEngineValve、StandardHostValve、StandardContextValve、StandardWrapperValve。

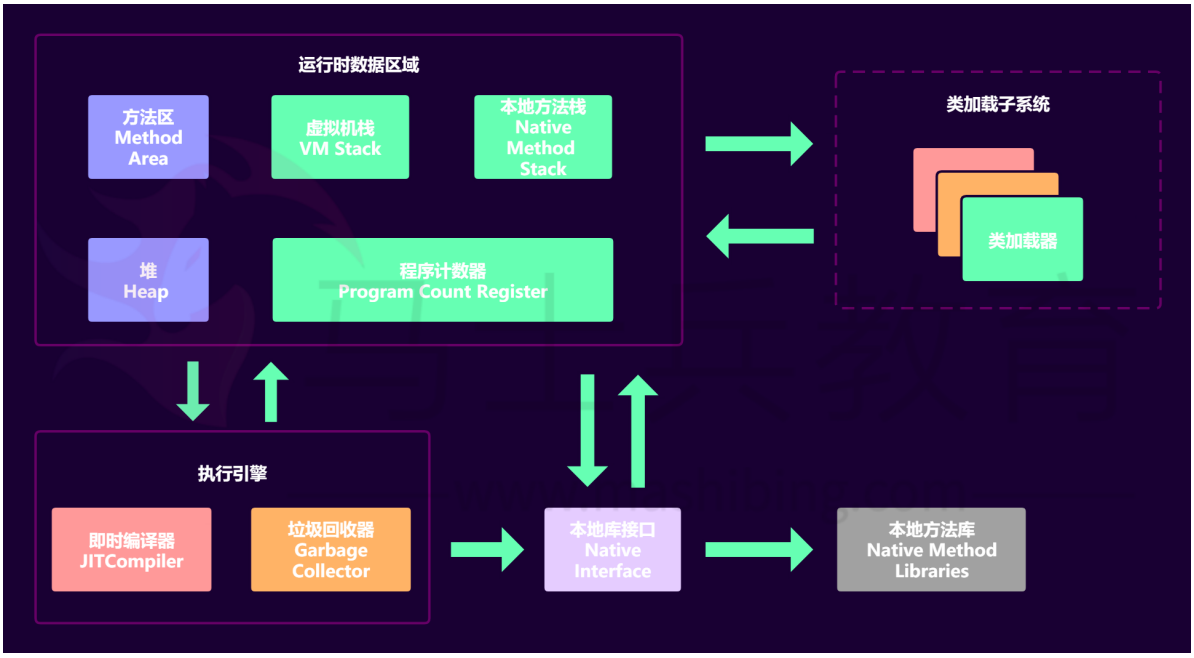
Pipeline的处理流程图如下：



- Connector在接收到请求后会首先调用最顶层容器的Pipeline来处 理，这里的最顶层容器的Pipeline就是EnginePipeline（Engine的管 道）；
- 在Engine的管道中依次会执行EngineValve1、EngineValve2等等， 最后会执行StandardEngineValve，在StandardEngineValve中会调用 Host管道，然后再依次执行Host的HostValve1、HostValve2等，最后在 执行StandardHostValve，然后再依次调用Context的管道和Wrapper的 管道，最后执行到StandardWrapperValve。
- 当执行到StandardWrapperValve的时候，会在 StandardWrapperValve中创建FilterChain，并调用其doFilter方法来处 理请求，这个FilterChain包含着我们配置的与请求相匹配的Filter和Servlet，其doFilter方法会依次调用所有的Filter的doFilter方法和Servlet 的service方法，这样请求就得到了处理！
- 当所有的Pipeline-Valve都执行完之后，并且处理完了具体的请求， 这个时候就可以将返回的结果交给Connector了，Connector在通过 Socket的方式将结果返回给客户端。

JVM（马老师，黄老师）

说一下 JVM 的主要组成部分及其作用?



JVM包含两个子系统和两个组件，两个子系统为Class loader(类装载)、 Execution engine(执行引擎)；两个组件为Runtime data area(运行时数据 区)、Native Interface(本地接口)。

- Class loader(类装载)：根据给定的全限定名类名(如： java.lang.Object)来装载class文件到Runtime data area中的method area。

- Execution engine（执行引擎）：执行classes中的指令。
- Native Interface(本地接口)：与native libraries交互，是其它编程语 言交互的接口。
- Runtime data area(运行时数据区域)：这就是我们常说的JVM的内 存。

作用： 首先通过编译器把 Java 代码转换成字节码，类加载器（ClassLoader） 再把字节码加载到内存中，将其放在运行时数据区（Runtime data area）的方 法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作 系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将 字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他 语言的本地库接口（Native Interface）来实现整个程序的功能。

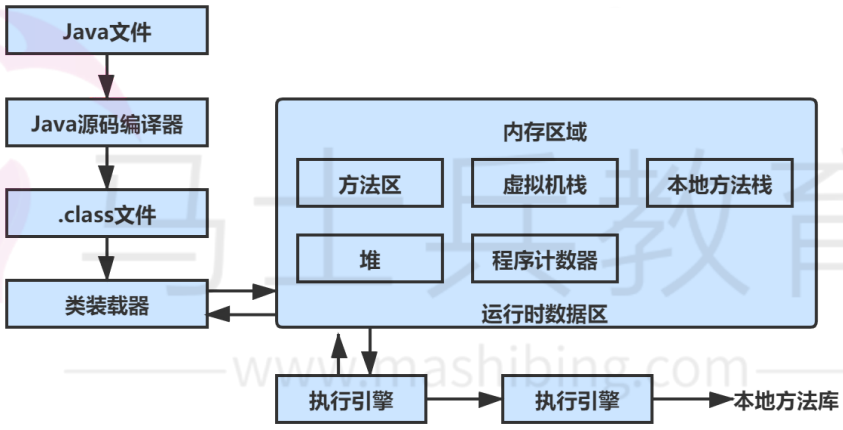
下面是Java程序运行机制详细说明

Java程序运行机制步骤

首先利用IDE集成开发工具编写Java源代码，源文件的后缀为.java；

再利用编译器(javac命令)将源代码编译成字节码文件，字节码文件的后缀名 为.class；

运行字节码的工作是由解释器(java命令)来完成的。

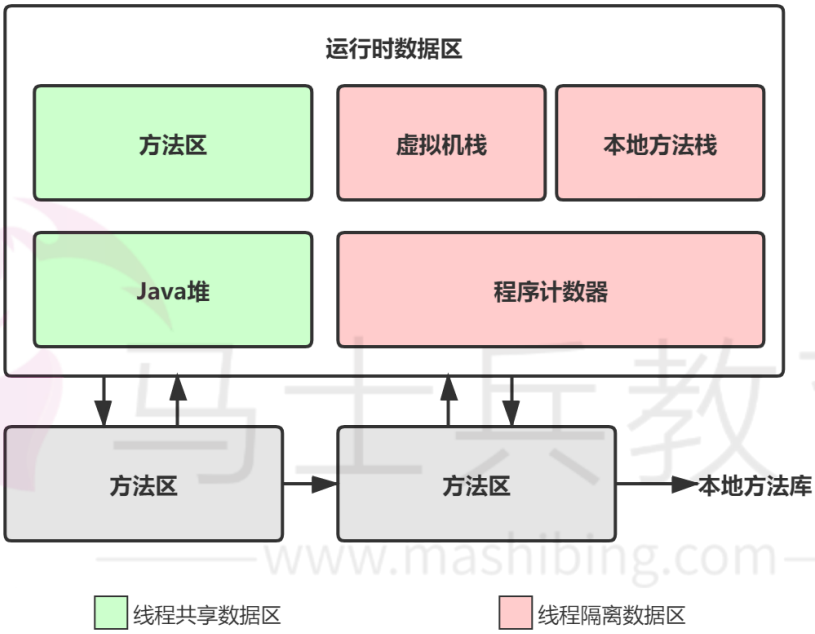


从上图可以看，java文件通过编译器变成了.class文件，接下来类加载器又将这 些.class文件加载到JVM中。

其实可以一句话来解释：类的加载指的是将类的.class文件中的二进制数据读入 到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 java.lang.Class对象，用来封装类在方法区内的数据结构。

说一下 JVM 运行时数据区

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存区域划分为若干个 不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有些区域 随着虚拟机进程的启动而存在，有些区域则是依赖线程的启动和结束而建立和销 毁。Java 虚拟机所管理的内存被划分为如下几个区域：



不同虚拟机的运行时数据区可能略微有所不同，但都会遵从 Java 虚拟机规范， Java 虚拟机规范规定的区域分为以下 5 个部分：

- 程序计数器（Program Counter Register）：当前线程所执行的字节码的行号 指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的 字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个 计数器来完成；
- Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作 数栈、动态链接、方法出口等信息；
- 本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚 拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；
- Java 堆（Java Heap）：Java 虚拟机中内存大的一块，是被所有线程共享 的，几乎所有的对象实例都在这里分配内存；
- 方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变 量、即时编译后的代码等数据。

深拷贝和浅拷贝

浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址，

深拷贝（deepCopy）是增加了一个指针并且申请了一个新的内存，使这个增加 的指针指向这个新的内存，

使用深拷贝的情况下，释放内存的时候不会因为出现浅拷贝时释放同一个内存的 错误。

浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来 的对象也会相应的改变。

深复制：在计算机中开辟一块**新的内存地址**用于存放复制的对象。

说一下堆栈的区别?

物理地址

堆的物理地址分配对对象是不连续的。因此性能慢些。在GC的时候也要考虑到 不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代 （即新生代使用复制算法，老年代使用标记——压缩） 栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性 能快。

内存分别

堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般 堆大小远远大于栈。

栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。

存放的内容

堆存放的是对象的实例和数组。因此该区更关注的是数据的存储

栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。
PS：

1. 静态变量放在方法区
2. 静态的对象还是放在堆。

程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。

队列和栈是什么？有什么区别？

队列和栈都是被用来预存储数据的。

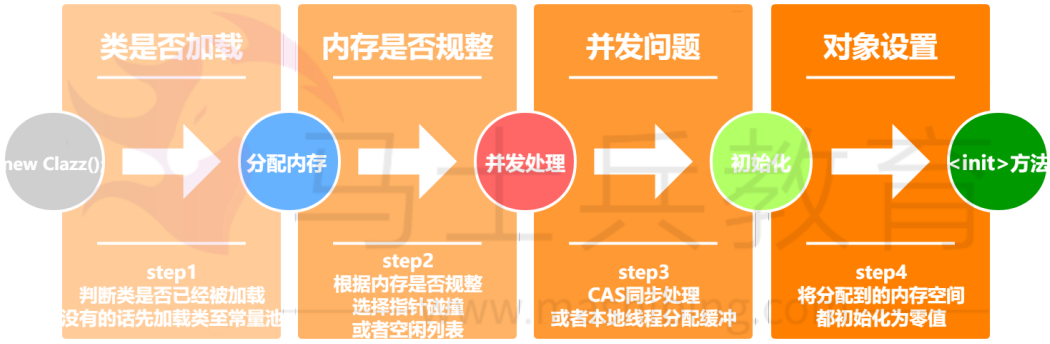
- 操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进 栈，栈的删除称为出栈。
- 可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进 栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。
- 操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原 则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列 头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当 前栈中新的元素，即后插入（进栈）的元素，而先插入的被放在栈的底部，要到后才能删除。

对象的创建

说到对象的创建，首先让我们看看 Java 中提供的几种对象创建方式：

Header	解释
使用new关键字	调用了构造函数
使用Class的newInstance方法	调用了构造函数
使用Constructor类的newInstance方法	调用了构造函数
使用clone方法	没有调用构造函数
使用反序列化	没有调用构造函数

下面是对象创建的主要流程:



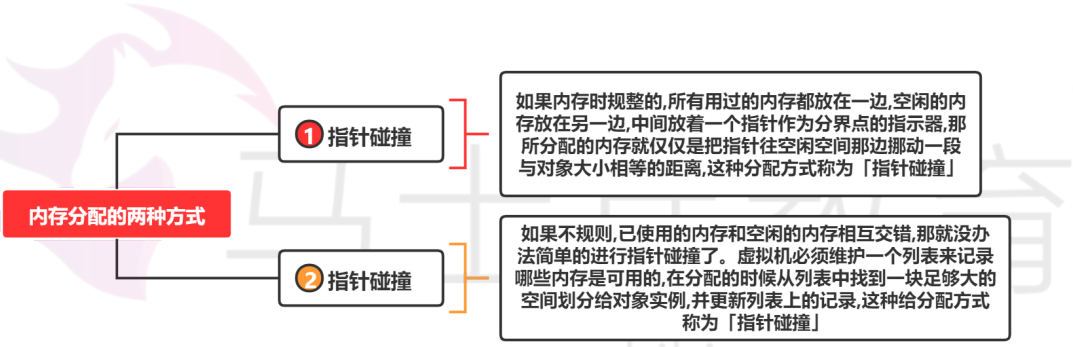
虚拟机遇到一条new指令时，先检查常量池是否已经加载相应的类，如果没有， 必须先执行相应的类加载。类加载通过后，接下来分配内存。若Java堆中内存是 绝对规整的，使用"指针碰撞"方式分配内存；如果不是规整的，就从空闲列表 中分配，叫做"空闲列表"方式。划分内存时还需要考虑一个问题-并发，也有 两种方式: CAS同步处理，或者本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。然后内存空间初始化操作，接着是做一些必要的对象设置(元信 息、哈希码...)，后执行方法。

为对象分配内存

类加载完成后，接着会在Java堆中划分一块内存分配给对象。内存分配根据Java 堆是否规整，有两种方式：

- 指针碰撞：如果Java堆的内存是规整，即所有用过的内存放在一边，而空闲的的 放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小 相等的距离，这样便完成分配内存工作。
- 空闲列表：如果Java堆的内存不是规整的，则需要由虚拟机维护一个列表来记录 那些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对 象，并在分配后更新列表记录。

选择哪种分配方式是由 Java 堆是否规整来决定的，而 Java 堆是否规整又由所 采用的垃圾收集器是否带有压缩整理功能决定。

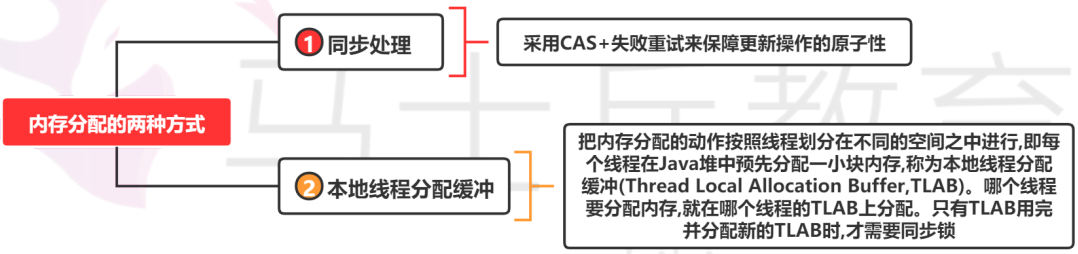


处理并发安全问题

对象的创建在虚拟机中是一个非常频繁的行为，哪怕只是修改一个指针所指向的 位置，在并发情况下也是不安全的，可能出现正在给对象 A 分配内存，指针还 没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。解决这个 问题有两种方案：

对分配内存空间的动作进行同步处理（采用 CAS + 失败重试来保障更新操作的 原子性）；

把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆 中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB）。哪个线程要分配内存，就在哪个线程的 TLAB 上分配。只有 TLAB 用完并 分配新的 TLAB 时，才需要同步锁。通过-XX:+/-UserTLAB参数来设定虚拟机是否使 用TLAB。



www.mashibing.com

对象的访问定位

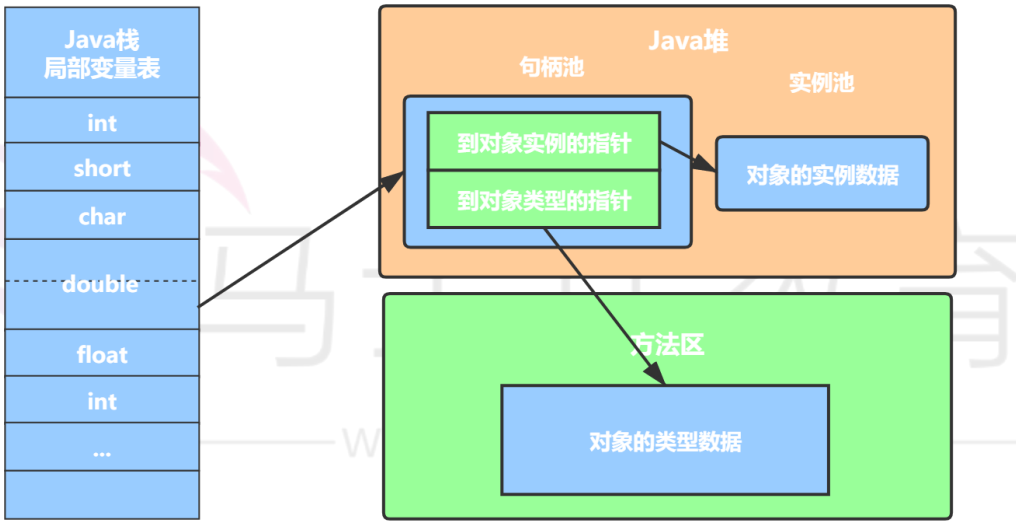
Java程序需要通过 **JVM** 栈上的引用访问堆中的具体对象。对象的访问方式取决 于 JVM 虚拟机的实现。目前主流的访问方式有 **句柄** 和 **直接指针** 两种方式。

指针： 指向对象，代表一个对象在内存中的起始地址。

句柄： 可以理解为指向指针的指针，维护着对象的指针。句柄不直接指向对象，而是 指向对象的指针（句柄不发生变化，指向固定内存地址），再由对象的指针指向对象的 真实内存地址。

句柄访问

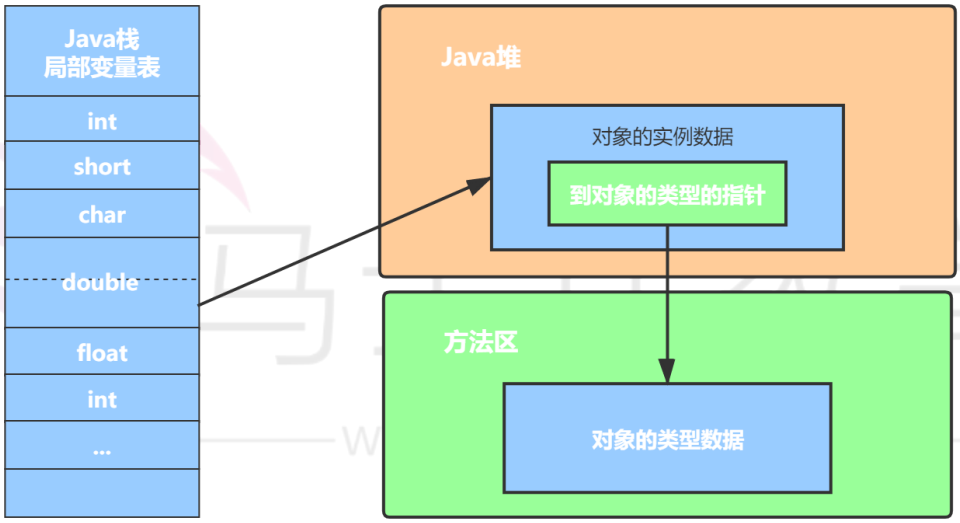
Java堆中划分出一块内存来作为**句柄池**，引用中存储对象的**句柄地址**，而句柄中 包含了**对象实例数据**与**对象类型数据**各自的**具体地址**信息，具体构造如下图所 示：



优势：引用中存储的是**稳定**的句柄地址，在对象被移动（垃圾收集时移动对象是 非常普遍的行为）时只会改变**句柄**中的**实例数据指针**，而**引用**本身不需要修改。

直接指针

如果使用**直接指针**访问，**引用** 中存储的直接就是**对象地址**，那么Java堆对象内 部的布局中就必须考虑如何放置访问**类型数据**的相关信息。



优势：速度更快，节省了一次指针定位的时间开销。由于对象的访问在Java中非 常频繁，因此这类开销积少成多后也是非常可观的执行成本。HotSpot 中采用 的就是这种方式。

内存溢出异常 Java会存在内存泄漏吗？请简单描述

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说， Java是有GC垃圾回收机制的，也就是说，不再被使用的对象，会被GC自动回收 掉，自动从内存中清除。

但是， 即使这样，Java也还是存在着内存泄漏的情况，java导致内存泄露的原因 很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露， 尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导 致不能被回收，这就是java中内存泄露的发生场景。

内存溢出异常

Java会存在内存泄漏吗？请简单描述

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说， Java是有GC垃圾回收机制的，也就是说，不再被使用的对象，会被GC自动回收 掉，自动从内存中清除。

但是，即使这样，Java也还是存在着内存泄漏的情况，java导致内存泄露的原因 很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露， 尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导 致不能被回收，这就是java中内存泄露的发生场景。

垃圾收集器

简述Java垃圾回收机制

在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行 执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会 执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没 有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

GC是什么？为什么要GC

GC 是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问 题的地方，忘记或者错误的内存

回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测 对象是否超过作用域从而达到自动

回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

垃圾回收的优点和原理。并考虑2种回收机制

java语言最显著的特点就是引入了垃圾回收机制，它使java程序员在编写程序时 不再考虑内存管理的问题。

由于有这个垃圾回收机制，java中的对象不再有“作用域”的概念，只有引用的 对象才有“作用域”。

垃圾回收机制有效的防止了内存泄露，可以有效的使用可使用的内存。

垃圾回收器通常作为一个单独的低级别的线程运行，在不可预知的情况下对内存 堆中已经死亡的或很长时间没有用过的对象进行清除和回收。

程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。 垃圾回收有分代复制垃圾回收、标记垃圾回收、增量垃圾回收。

垃圾回收器的基本原理是什么？垃圾回收器可以马上回收 内存吗？有什么办法主动通知虚拟机进行垃圾回收？ 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及 使用情况。

通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式 确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可 达"时，GC就有责任回收这些内存空间。

可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不 保证GC一定会执行。

Java 中都有哪些引用类型?

- 强引用：发生 gc 的时候不会被回收。
- 软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。
- 弱引用：有用但不是必须的对象，在下一次GC时会被回收。
- 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用

PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

怎么判断对象是否可以被回收?

垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收 的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需 要被回收。

一般有两种方法来判断：

- 引用计数器法：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用 被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用 的问题；
- 可达性分析算法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。 当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

在Java中，对象什么时候可以被垃圾回收

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被 回收了。

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全 垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代 也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原 因。

JVM中的永久代中会发生垃圾回收吗**

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全 垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代 也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原 因。请参考下Java8：从永久代到元数据区

(译者注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存 区)

说一下 JVM 有哪些垃圾回收算法?

- 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清 除垃圾碎片。

- 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的 对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不 高，只有原来的一半。
- 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清 除掉端边界以外的内存。
- 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年 代，新生代基本采用复制算法，老年代采用标记整理算法。

标记-清除算法**

标记无用对象，然后进行清除回收。

标记-清除算法（Mark-Sweep）是一种常见的基础垃圾收集算法，它将垃圾收 集分为两个阶段：

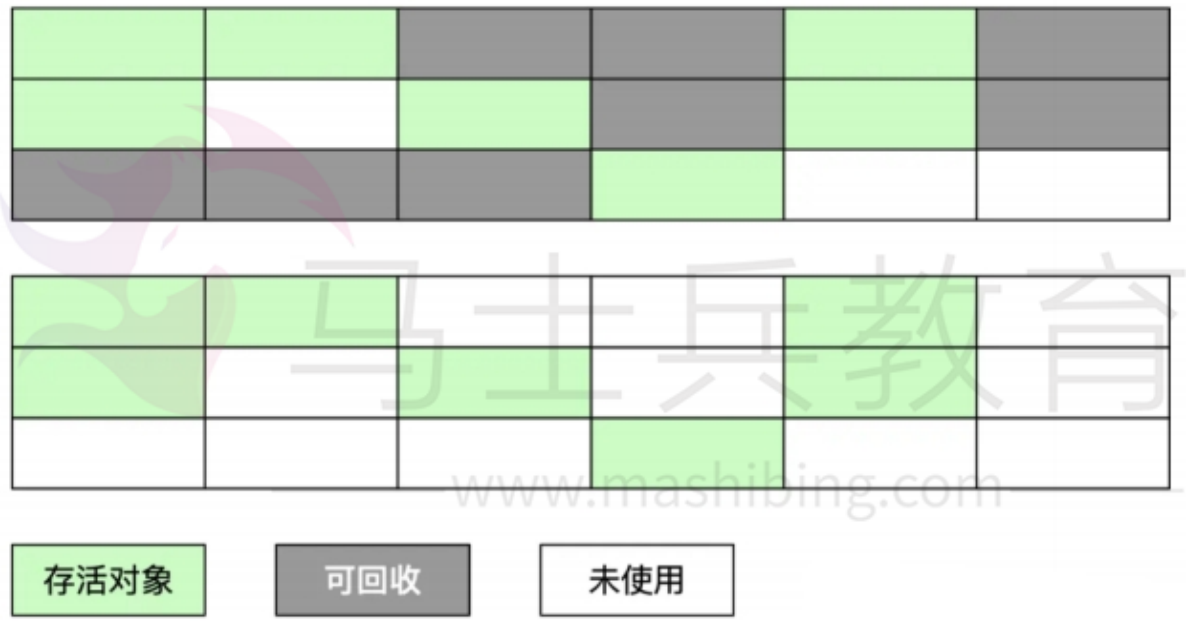
- 标记阶段：标记出可以回收的对象。
- 清除阶段：回收被标记的对象所占用的空间。

标记-清除算法之所以是基础的，是因为后面讲到的垃圾收集算法都是在此算法 的基础上进行改进的。

优点**：实现简单，不需要对象进行移动。

缺点**：标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的 频率。

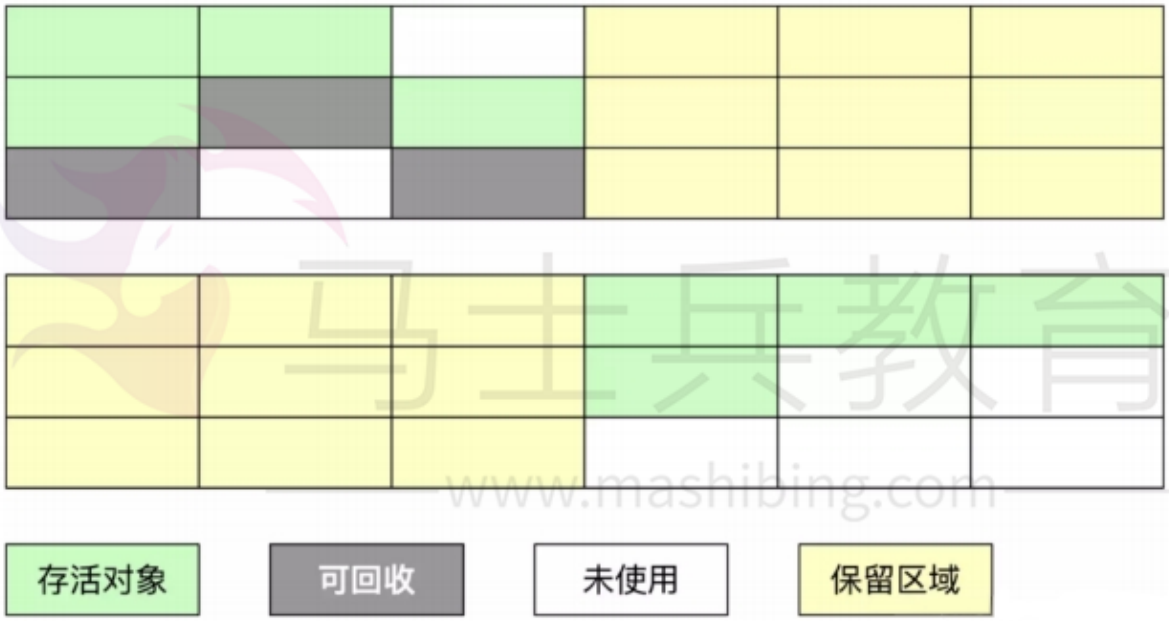
标记-清除算法的执行的过程如下图所示



复制算法**

为了解决标记-清除算法的效率不高的问题，产生了复制算法。它把内存空间划 为两个相等的区域，每次只使用其中一个区域。垃圾收集时，遍历当前使用的区 域，把存活对象复制到另外一个区域中，最后将当前使用的区域的可回收的对象 进行回收。

优点：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。 缺**点**：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。 复制算法的执行过程如下图所示



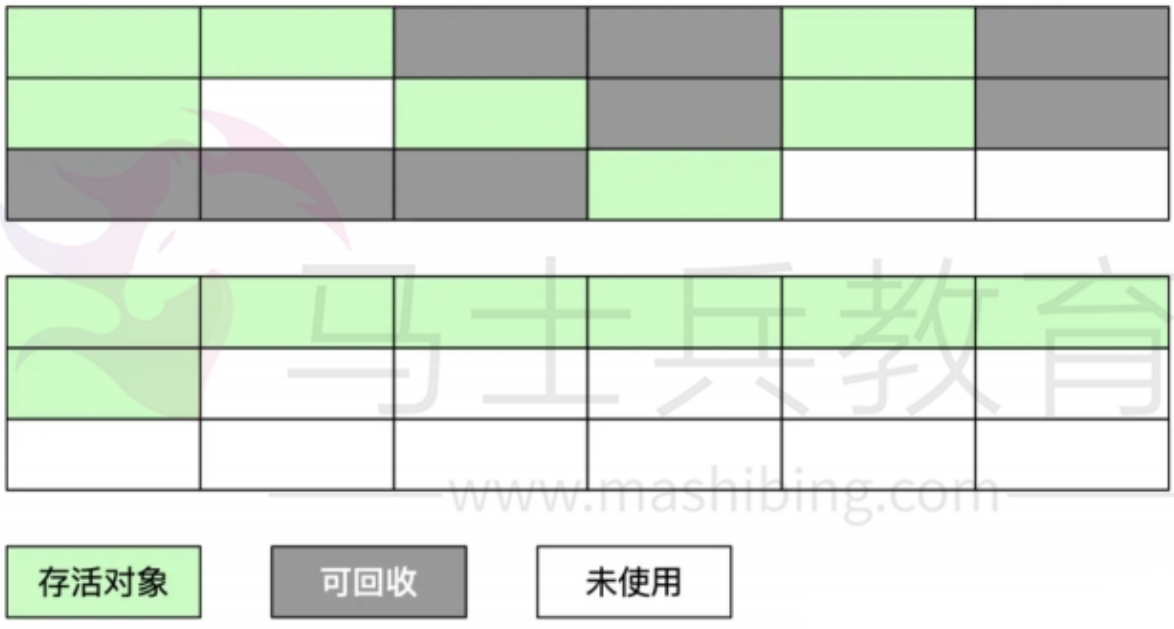
标记-整理算法

在新生代中可以使用复制算法，但是在老年代就不能选择复制算法了，因为老年 代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记-清除 算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎 片。因此就出现了一种标记-整理算法（Mark-Compact）算法，与标记-整理 算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使 他们紧凑的排列在一起，然后对端边界以外的内存进行回收。回收后，已用和未 用的内存都各自一边。

优点：解决了标记-清理算法存在的内存碎片问题。

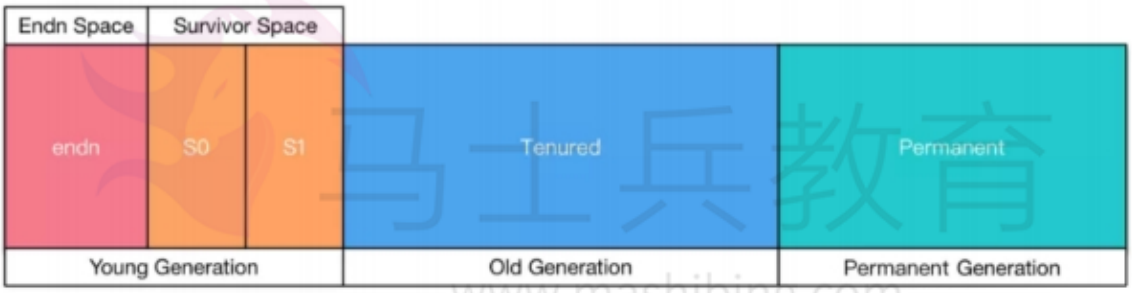
缺点：仍需要进行局部对象移动，一定程度上降低了效率。

标记-整理算法的执行过程如下图所示



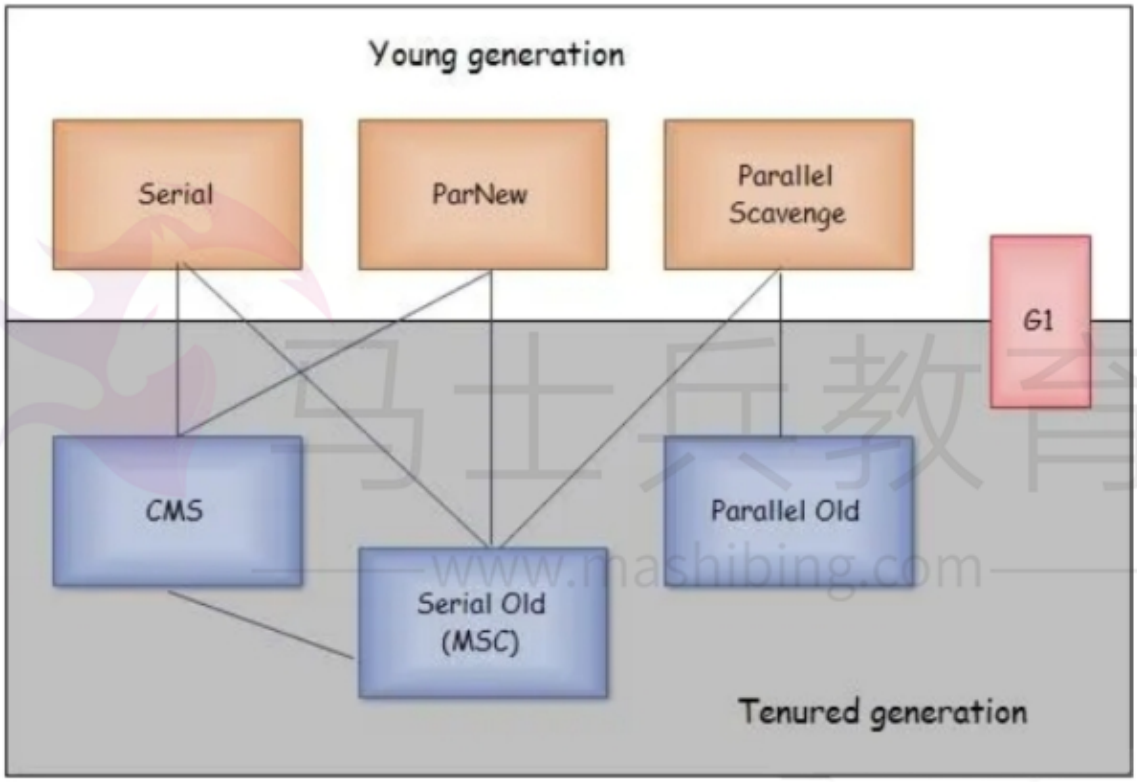
分代收集算法

当前商业虚拟机都采用**分**代**收**集**的垃圾收集算法。分代收集算法，顾名思义是根 据对象的**存活周期**将内存划分为几块。一般包括**年轻代、老年代 和 **永久代**，如 图所示：



说一下 JVM 有哪些垃圾回收器?

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体 实现。下图展示了7种作用于不同分代的收集器，其中用于回收新生代的收集器 包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器 之间的连线表示它们可以搭配使用。



- Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点 是简单高效；
- ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程 版本，在多核CPU环境下有着比Serial更好的表现；
- Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效 利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高 效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不 高的场景；
- Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年 代版本；
- Parallel Old收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先， Parallel Scavenge收集器的老年代版本；
- CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集 器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最 短GC回收停顿时间。
- G1(Garbage First)收集器 (标记-整理算法)： Java堆并行收集器，G1收集器是 JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会 产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代 或老年代。

详细介绍一下 CMS 垃圾回收器?

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得 最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾 回收器非常适合。在启动 JVM 的参数加上"-XX:+UseConcMarkSweepGC"来指定使用 CMS 垃圾回收器。 CMS 使用的是标记-清除的算法实现的，所以在 gc的时候回产生大量的内存碎 片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会 被降低。

新生代垃圾回收器和老年代垃圾回收器都有哪些？有什么 区别?

- 新生代回收器：Serial、ParNew、Parallel Scavenge
- 老年代回收器：Serial Old、Parallel Old、CMS
- 整堆回收器：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内 存利用率低；老年代回收器一般采用的是标记-整理的算法进行垃圾回收。

简述分代垃圾回收器是怎么工作的?

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

- 把 Eden + From Survivor 存活的对象放入 To Survivor 区；
- 清空 Eden 和 From Survivor 分区；
- From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年 龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的 执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

内存分配策略**

简述java内存分配与回收策率以及Minor GC和Major GC

所谓自动内存管理，最终要解决的也就是内存分配和内存回收两个问题。前面我 们介绍了内存回收，这里我们再来聊聊内存分配。

对象的内存分配通常是在 Java 堆上分配（随着虚拟机优化技术的诞生，某些场 景下也会在栈上分配，后面会详细介绍），对象主要分配在新生代的 Eden 区， 如果启动了本地线程缓冲，将按照线程优先在 TLAB 上分配。少数情况下也会直 接在老年代上分配。总的来说分配规则不是百分百固定的，其细节取决于哪一种 垃圾收集器组合以及虚拟机相关参数有关，但是虚拟机对于内存的分配还是会遵 循以下几种「普世」规则：

对象优先在 Eden 区分配

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行 分配时，虚拟机将会发起一次 Minor GC。如果本次 GC后还是没有足够的空 间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中 发现 Major GC/Full GC。

- **Minor GC** 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所 有 Minor GC 非常频繁，一般回收速度也非常快；

- **Major GC/Full GC** 是指发生在老年代的 GC，出现了 Major GC 通常会伴 随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

大对象直接进入老年代

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导 致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对 象。

前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象 直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

长期存活对象将进入老年代

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应 该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对 象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被 移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬 过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15） 就会被晋升 到老年代。

虚拟机类加载机制

简述java类加载机制?

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，解析和初 始化，最终形成可以被虚拟机直接使用的java类型。

描述一下JVM加载Class文件的原理机制

Java中的所有类，都需要由类加载器装载到JVM中才能运行。类加载器本身也 是一个类，而它的工作就是把class文件从硬盘读取到内存中。在写程序的时 候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊 的用法，像是反射，就需要显式的加载所需要的类。

类装载方式，有两种：

1.隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用 类装载器加载对应的类到jvm中，

2.显式装载， 通过class.forname() 等方法，显式加载需要的类

Java类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证 程序运行的基础类(像是基类)完全加载到jvm中，至于其他类，则在需要的时候 才加载。这当然就是为了节省内存开销。

什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。 主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被 java程序直接引用。
2. 扩展类加载器(extensions class loader): 它用来加载 Java 的扩展库。 Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找 并加载 Java 类。
3. 系统类加载器（system class loader ）：它根据 Java 应用的类路径 （CLASSPATH ）来加载 Java 类。一般来说，Java 应用的类都是由它来 完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取 它。
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实

现。

说一下类装载的执行过程?

类装载分为以下 5个步骤：

加载：根据查找路径找到相应的 class 文件然后导入；

验证：检查加载的 class 文件的正确性；

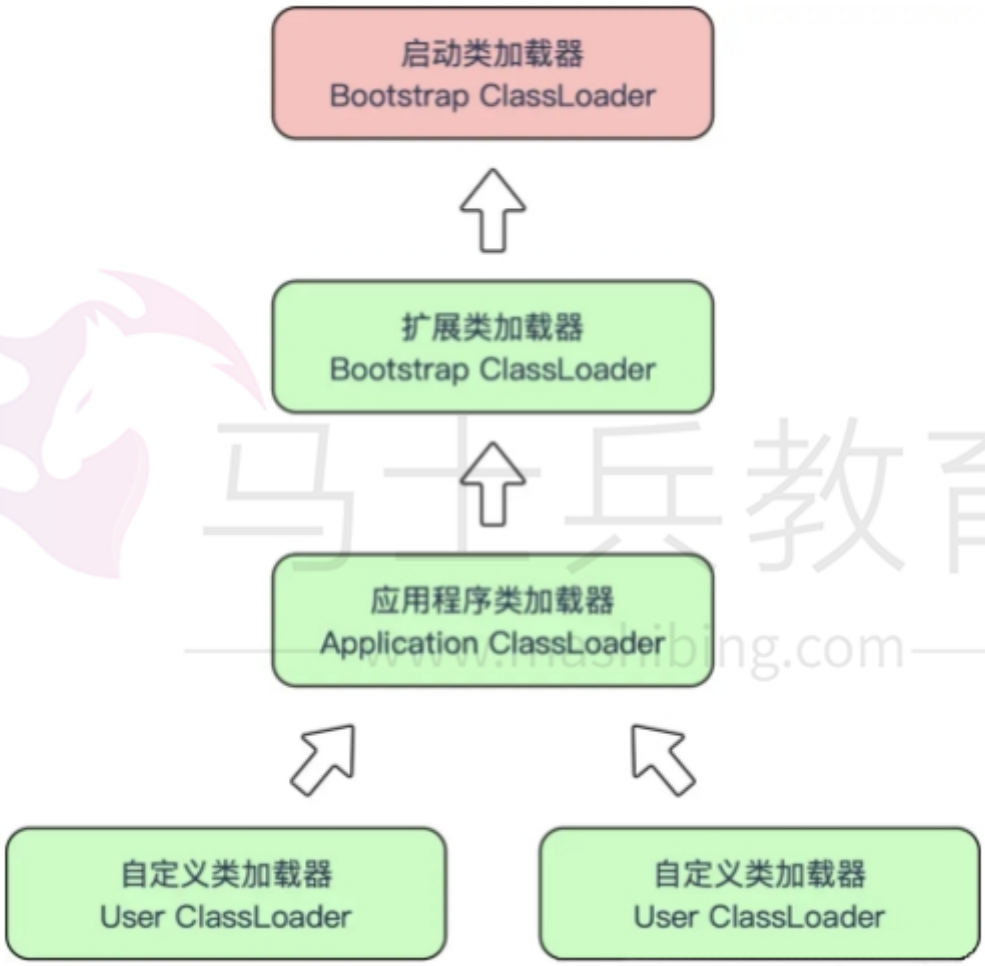
准备：给类中的静态变量分配内存空间；

解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为 一个标示，而在直接引用直接指向内存中的地址；

初始化：对静态变量和静态代码块执行初始化工作。

什么是双亲委派模型?

在介绍双亲委派模型之前先说下类加载器。对于任意一个类，都需要由加载它的 类加载器和这个类本身一同确立在 JVM 中的唯一性，每一个类加载器，都有一 个独立的类名称空间。类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。



类加载器分类：

- 启动类加载器（Bootstrap ClassLoader），是虚拟机自身的一部分，用来加载 Java_HOME/lib/目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚 拟机识别的类库；
- 其他类加载器：
    - 扩展类加载器（Extension ClassLoader）：负责加载\lib\ext目录或Java. ext. dirs系统变量指定的路径中的所有类库；
    - 应用程序类加载器（Application ClassLoader）。负责加载用户类路径 （classpath）上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我 们没有自定义类加载器默认就是用这个加载器。

双亲委派模型：如果一个类加载器收到了类加载的请求，它首先不会自己去加载 这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如 此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无 法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加 载类。

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父 类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加 载。

JVM调优

说一下 JVM 调优的工具?

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

- jconsole：用于对 JVM 中的内存、线程和类等进行监控；
- jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序 死锁、监控内存的变化、gc 变化等。

常用的 JVM 调优的参数都有哪些?

- -Xms2g：初始化推大小为 2g；
- -Xmx2g：堆最大内存为 2g；
- -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4； -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2； –XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合； -XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组
- 合；
- -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组 合；
- -XX:+PrintGC：开启打印 gc 信息；
- -XX:+PrintGCDetails：打印 gc 详细信息。

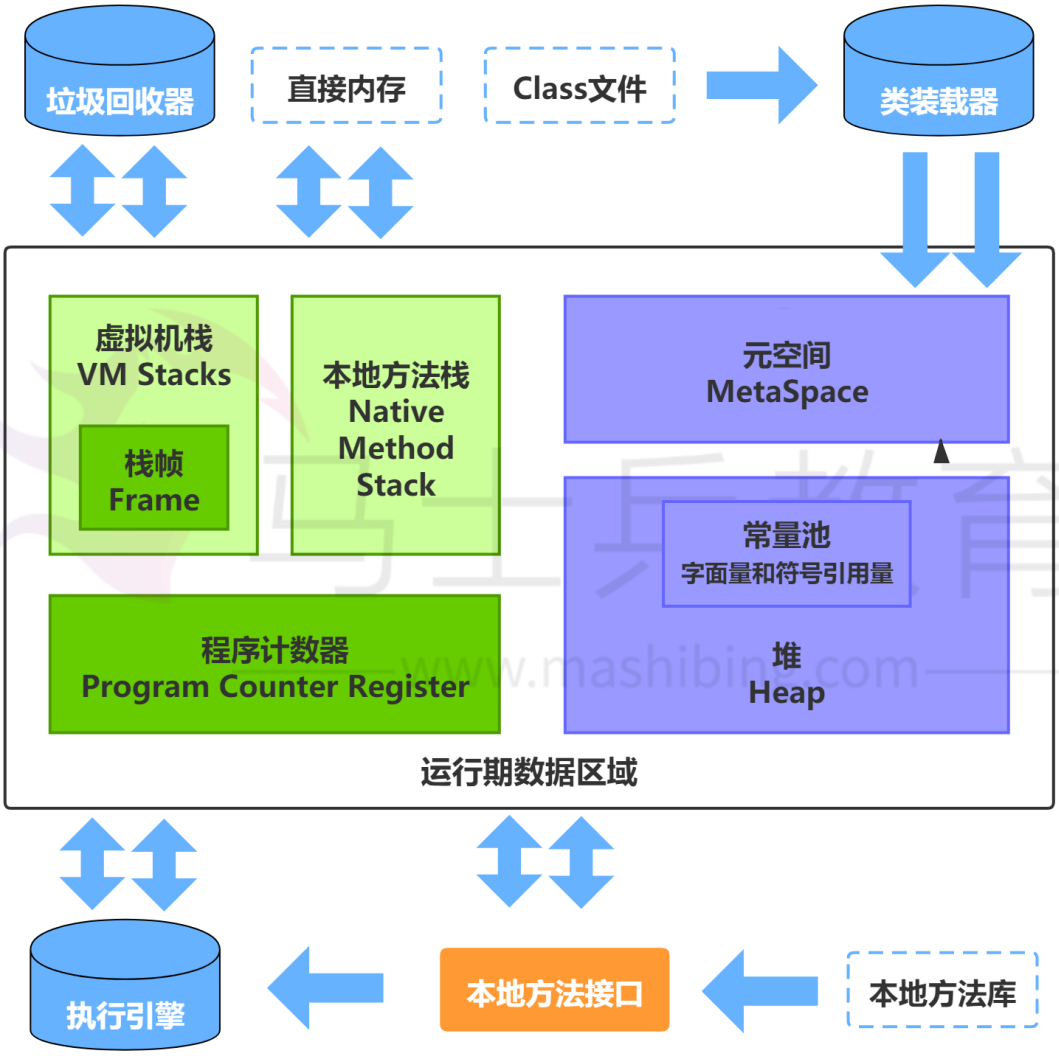
JVM内存模型，GC机制和原理；

内存模型

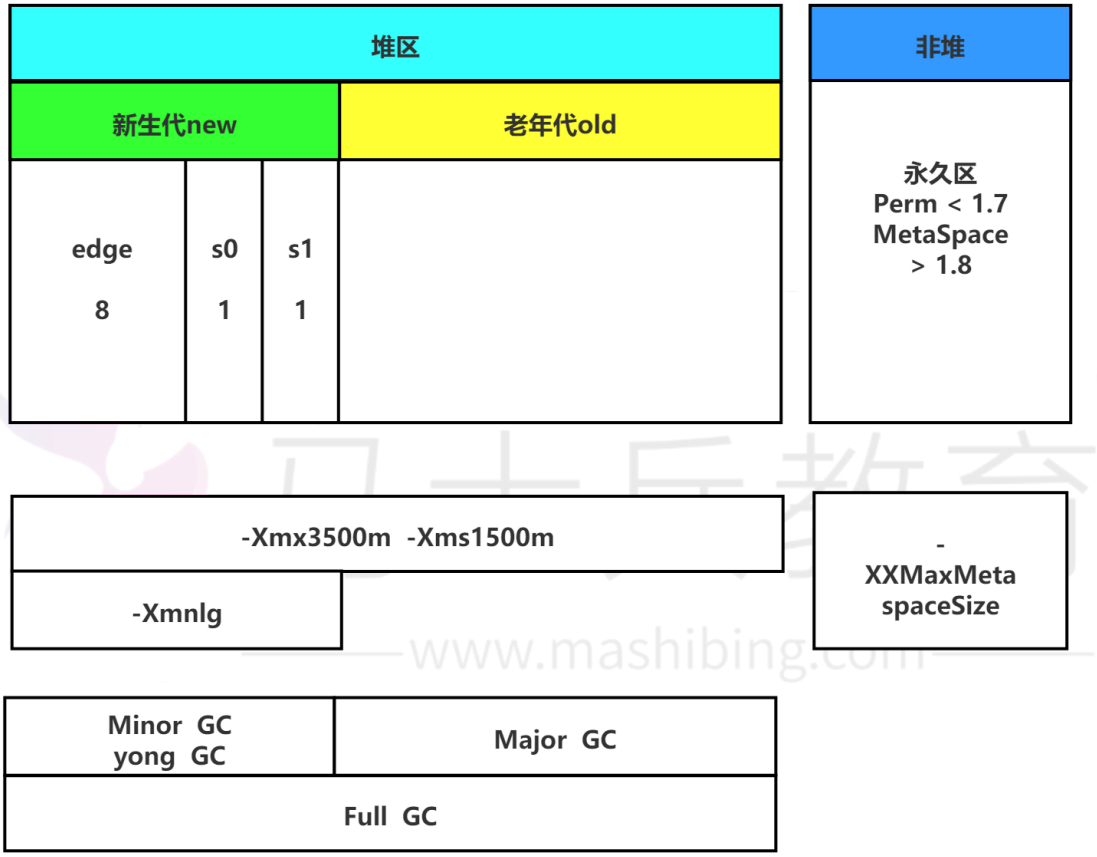
Jdk1.6及之前：有永久代, 常量池在方法区

Jdk1.7：有永久代，但已经逐步“去永久代”，常量池在堆

Jdk1.8及之后： 无永久代，常量池在元空间



GC分哪两种，Minor GC 和Full GC有什么区别？什么时候会触发Full GC？分别采用什么算法？



对象从新生代区域消失的过程，我们称之为 "minor GC"

对象从老年代区域消失的过程，我们称之为 "major GC"

Minor GC
清理整个YouGen的过程，eden的清理，S0\S1的清理都会由于MinorGC Allocation Failure(YoungGen区内存不足），而触发minorGC

Major GC
OldGen区内存不足，触发Major GC

Full GC
Full GC 是清理整个堆空间—包括年轻代和永久代

Full GC 触发的场景

1）System.gc

2）promotion failed (年代晋升失败,比如eden区的存活对象晋升到S区放不下，又尝试直接晋 升到Old区又放不下，那么Promotion Failed,会触发FullGC)

3）CMS的Concurrent-Mode-Failure 由于CMS回收过程中主要分为四步:
1.CMS initial mark
2.CMS Concurrent mark
3.CMS remark
4.CMS Concurrent sweep。在2中gc线程与用户线程同时执行，那么用户线程依旧可 能同时产生垃圾，如果这个垃圾较多无法放入预留的空间就会产生CMS-Mode-Failure， 切换 为SerialOld单线程做mark-sweep-compact。

4）新生代晋升的平均大小大于老年代的剩余空间 （为了避免新生代晋升到老年代失败） 当使用G1,CMS 时，FullGC发生的时候 是 Serial+SerialOld。 当使用ParalOld时，FullGC发生的时候是ParallNew +ParallOld.

JVM里的有几种classloader，为什么会有多种?

启动类加载器：负责加载JRE的核心类库，如jre目标下的rt.jar,charsets.jar等

扩展类加载器：负责加载JRE扩展目录ext中JAR类包

系统类加载器：负责加载ClassPath路径下的类包

用户自定义加载器：负责加载用户自定义路径下的类包

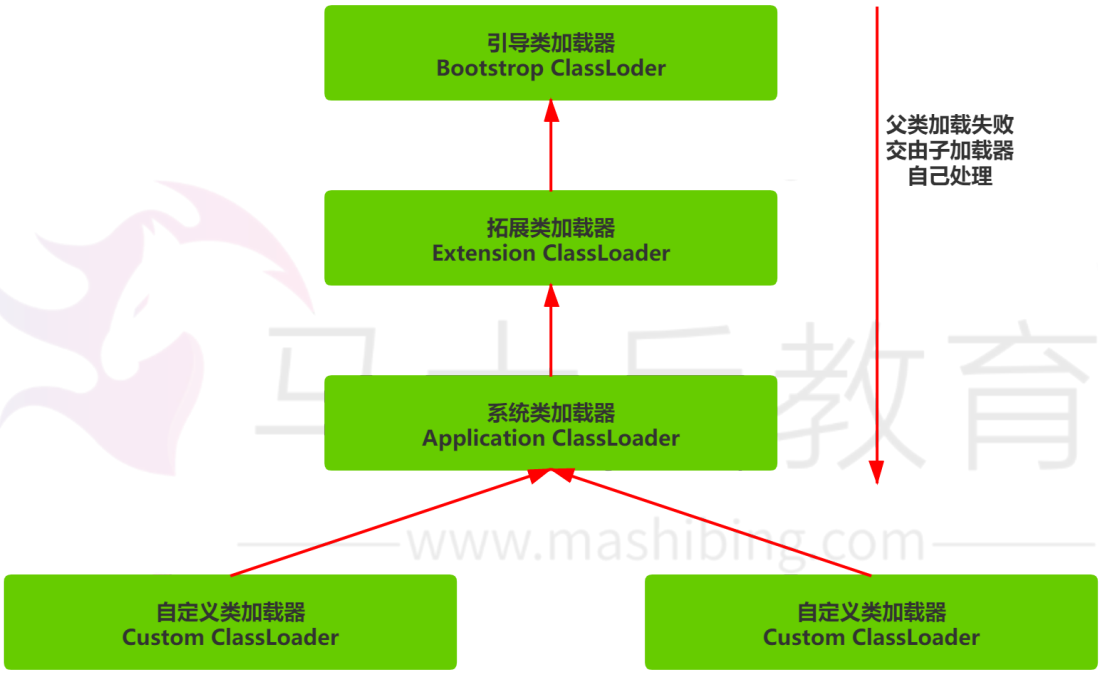
为什么会有多种：

1）分工，各自负责各自的区块

2）为了实现委托模型

什么是双亲委派机制？介绍一些运作过程，双亲委派模型的好处；

如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的 加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终 将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载 器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每个儿子都不 愿意干活，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己想办法去完 成，这不就是传说中的双亲委派模式。



好处

沙箱安全机制：自己写的String.class类不会被加载，这样便可以防止核心API库被随意篡改 避免类的重复加载：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次

常见的JVM调优方法有哪些？可以具体到调整哪个参数，调成什么值?

调优工具

console，jProfile，VisualVM

Dump线程详细信息：查看线程内部运行情况

死锁检查

查看堆内类、对象信息查看：数量、类型等

线程监控

线程信息监控：系统线程数量。

线程状态监控：各个线程都处在什么样的状态下

热点分析

CPU热点：检查系统哪些方法占用的大量CPU时间

内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

内存泄漏检查

JVM虚拟机内存划分、类加载器、垃圾收集算法、垃圾收集器、class 文件结构是如何解析的；

JVM虚拟机内存划分（重复）

类加载器（重复）

垃圾收集算法：标记-清除算法、复制算法、标记-整理算法、分代收集算法

垃圾收集器： Serial收集器、ParNew收集器、Parallel Scavenge收集器、Serial Old收集器、 Parallel Old收集器、CMS收集器、G1收集器、Z垃圾收集器

class文件结构是如何解析的

```
ClassFile {
    u4 magic;                魔数
    u2 minor_version;        副版本号
    u2 major_version;        主版本号
    u2 constant_pool_count;  常量池计数器
    cp_info constant_pool[constant_pool_count-1]; 常量池
    u2 access_flags;         访问标志
    u2 this_class;           类索引
    u2 super_class;          父类索引
    u2 interfaces_count;     接口计数器
    u2 interfaces[interfaces_count];    接口表
    u2 fields_count;         字段计数器
    field_info fields[fields_count];    字段表
    u2 methouds_count;       方法计数器
    method_info method[methods_count];  方法表
    u2 attributes_count;     属性计数器
    attributes_info attributes[attributes_count];  属性表
}
```

马士兵教育 www.mashibing.com

类（class）文件结构

前面的内容我们了解到jvm的内存结构。所有java文件必须经过“编译”转成class文件之后才会被jvm所识别和运用。那么我们开始了解一下类文件也就是class文件的结构。也就是我们写的java文件最终会被编译成什么样？那种格式？

本文讲解内容借鉴了《Java 虚拟机规范（Java SE 7 版）》第四章。如果有兴趣可以自行阅读

1、类文件介绍

每一个 Class 文件都对应着唯一一个类或接口的定义信息，但是相对地，类或接口并不一定都得定义在文件里（譬如类或接口也可以通过类加载器直接生成）。

本节中，我们只是通俗地将任意一个有效的类或接口所应当满足的格式称为“Class 文件格式”，即使它不一定以磁盘文件的形式存在。

每个 Class 文件都是由 8 字节为单位的字节流组成，所有的 16 位、32 位和 64 位长度的数据将被构造成 2 个、4 个和 8 个 8 字节单位来表示。多字节数据项总是按照 Big-Endian 的顺序进行存储。

注意：Big-Endian 顺序是指按高位字节在地址最低位，最低字节在地址最高位来存储数据，它是SPARC、PowerPC等处理器的默认多字节存储顺序，而x86等处理器则是使用了相反的Little-Endian顺序来存储数据。为了保证 Class 文件在不同硬件上具备同样的含义，因此在 Java 虚拟机规范中是有必要严格规定了数据存储顺序的。

在 Java SDK 中，访问这种格式的数据可以使用 java.io.DataInput、java.io.DataOutput 等接口和 java.io.DataInputStream 和java.io.DataOutputStream 等类来实现。

本节内容，还定义了一组私有数据类型来表示 Class 文件的内容，它们包括 u1，u2 和 u4，分别代表了 1、2 和 4 个字节的无符号数。

在 Java SDK 中这些类型的数据可以通过实现接口java.io.DataInput 中的 readUnsignedByte、readUnsignedShort 和 readInt 方法进行读取。

本节将采用类似 C 语言结构体的伪结构来描述 Class 文件格式。为了避免与类的字段、类的实例等概念产生混淆，在此把用于描述类结构格式的内容定义为项（Item）。

在 Class 文件中，各项按照严格顺序连续存放的，它们之间没有任何填充或对齐作为各项间的分隔符号。

表（Table）是由任意数量的可变长度的项组成，用于表示 Class 文件内容的一系列复合结构。尽管我们采用类似 C 语言的数组语法来表示表中的项，但是读者应当清楚意识到，表是由可变长数据组成的复合结构（表中每项的长度不固定），因此无法直接将字节偏移量来作为索引对表进行访问。

而我们描述一个数据结构为数组（Array）时，就意味着它含有零至多个长度固定的项组成，这个时候则可以采用数组索引的方式来访问它 。

注意：虽然原文中在此定义了“表”和“数组”的关系，但在后文中依然存在表和数组混用的情况。译文中作了一些修正，把各个数据项结构不一致的数据集合用“表”那表示，譬如“constant_pool 表”、 “attributes 表”，而把数据项结构一致的数据集合用“数组”来表示，譬如“code[]数组“、“fields[]数组”。

2、ClassFile 结构

每一个 Class 文件对应于一个如下所示的 ClassFile 结构体。

```
ClassFile {
    u4 magic;                魔数
    u2 minor_version;        副版本号
    u2 major_version;        主版本号
    u2 constant_pool_count;  常量池计数器
    cp_info constant_pool[constant_pool_count-1]; 常量池
    u2 access_flags;         访问标志
    u2 this_class;           类索引
    u2 super_class;          父类索引
    u2 interfaces_count;     接口计数器
    u2 interfaces[interfaces_count];    接口表
    u2 fields_count;         字段计数器
    field_info fields[fields_count];    字段表
    u2 methouds_count;       方法计数器
    method_info method[methods_count]; 方法表
    u2 attributes_count;属性计数器
    attributes_info attributes[attributes_count];  属性表
}
```

马士兵教育 www.mashibing.com

ClassFile 结构体中，各项的含义描述如下：

1，无符号数，以u1、u2、u4、u8分别代表1个字节、2个字节、4个字节、8个字节的无符号数

2，表是由多个无符号数或者其它表作为数据项构成的复合数据类型，所以表都以“_info”结尾，由多个无符号数或其它表构成的复合数据类型

每个部分出现的次数和数量见下表(Class文件格式)：

类型	名称	数量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fileds	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

magic

魔数，魔数的唯一作用是确定这个文件是否为一个能被虚拟机所接受的 Class 文件。魔数值固定为 0xCAFEBABE，不会改变。

minor_version、major_version

副版本号和主版本号，minor_version 和 major_version 的值分别表示 Class 文件的副、主版本。它们共同构成了 Class 文件的格式版本号。譬如某个 Class 文件的主版本号为 M，副版本号为 m，那么这个 Class 文件的格式版本号就确定为 M.m。Class 文件格式版本号大小的顺序为：1.5 < 2.0 < 2.1。

一个 Java 虚拟机实例只能支持特定范围内的主版本号（Mi 至 Mj）和 0 至特定范围内（0 至 m）的副版本号。假设一个 Class 文件的格式版本号为 V，仅当 Mi.0 ≤ v ≤ Mj.m

成立时，这个 Class 文件才可以被此 Java 虚拟机支持。不同版本的 Java 虚拟机实现支持的版本号也不同，高版本号的 Java 虚拟机实现可以支持低版本号的 Class 文件，反之则不成立 。

注意：Oracle 的 JDK 在 1.0.2 版本时，支持的 Class 格式版本号范围是 45.0 至 45.3；JDK 版本在 1.1.x 时，支持的 Class 格式版本号范围扩展至 45.0 至 45.65535；JDK 版本为 1. k 时（k ≥2）时，对应的 Class文件格式版本号的范围是 45.0 至 44+k.0

下表列举了Class文件版本号

编译器版本	-target参数	十六进制版本号	十进制版本号
JDK 1.1.8	不能带 target 参数	00 03 00 2D	45.3
JDK 1.2.2	不带 (默认为 -target 1.1)	00 03 00 2D	45.3
JDK 1.2.2	-target 1.2	00 00 00 2E	46.0
JDK 1.3.1_19	不带 (默认为 -target 1.1)	00 03 00 2D	45.3
JDK 1.3.1_19	-target 1.3	00 00 00 2F	47.0
JDK 1.4.2_10	不带 (默认为 -target 1.2)	00 00 00 2E	46.0
JDK 1.4.2_10	-target 1.4	00 00 00 30	48.0
JDK 1.5.0_11	不带 (默认为 -target 1.5)	00 00 00 31	49.0
JDK 1.5.0_11	-target 1.2 -source 1.4	00 00 00 30	48.0
JDK 1.6.0_01	不带 (默认为 -target 1.6)	00 00 00 32	50.0
JDK 1.6.0_01	-target 1.5	00 00 00 31	49.0
JDK 1.6.0_01	-target 1.4 -source 1.4	00 00 00 30	48.0
JDK 1.7.0	不带 (默认为 -target 1.7)	00 00 00 33	51.0
JDK 1.7.0	-target 1.6	00 00 00 32	50.0
JDK 1.7.0	-target 1.4 -source 1.4	00 00 00 30	48.0

onstant_pool_count
常量池计数器，constant_pool_count的值等于constant_pool表中的成员数加1。constant_pool 表的索引值只有在大于 0 且小于 constant_pool_count 时才会被
认为是有效的，对于 long 和 double 类型有例外情况，后续在讲解。

注意：虽然值为 0 的 constant_pool 索引是无效的，但其他用到常量池的数据结构可以使用索引 0 来表示“不引用任何一个常量池项”的意思。

constant_pool[ ]
常量池，constant_pool 是一种表结构（这里需要列举一下表就会明白，这个在下面的例子中会有讲解这个结构，返回来在读就会明白），它包含 Class 文件结构及其子结构中引用的所有字符串常量、类或接口名、字段名和其它常量。常量池中的每一项都具备相同的格式特征——第一个字节作为类型标记用于识别该项是哪种类型的常量，称为“tagbyte”。常量池的索引范围是 1 至 constant_pool_count−1。

1常量池的项目类型

类型	标志	描述
CONSTANT_Utf8_info	1	UTF-8编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldre_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_IngerfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	标识方法类型
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

2每一种类型的格式特征：这里用CONSTANT_Class_info举个例子：

类型	名称	数量
u1	tag	1
u2	name_index	1

常量池主要存放两大类常量：字面量（literal）和符号引用。字面量比较接近java语言层面的常量概念，比如文本字符串、声明的final的常量值等。符号引用属于编译原理方面概念。包括下面三类常量：类和接口的全局限定名。字段的名称和描述符。方法的名称和描述符。这三类稍后在详细讲解。

```
access_flags
```

访问标志，access_flags 是一种掩码标志，用于表示某个类或者接口的访问权限及基础属性。
access_flags 的取值范围和相应含义见表(访问和修饰符标志)所示。

标记名	值	含义
ACC_PUBLIC	0x0001	可以被包的类外访问
ACC_FINAL	0x0010	不允许有子类
ACC_SUPER	0x0020	当用到invokespecial指令时,需要特殊处理的父类方法
ACC_INTERFACE	0x0200	标识定义的是接口而不是类
ACC_ABSTRACT	0x0400	不能被实例化
ACC_SYNTHETIC	0x1000	标识并非Java源码生成的代码
ACC_ANNOTATION	0x2000	标识注解类型
ACC_ENUM	0x4000	标识枚举类型

带有 ACC_SYNTHETIC 标志的类，意味着它是由编译器自己产生的而不是由程序员编写的源代码生成的。

带有 ACC_ENUM 标志的类，意味着它或它的父类被声明为枚举类型。

带有 ACC_INTERFACE 标志的类，意味着它是接口而不是类，反之是类而不是接口。如果一个 Class 文件被设置了 ACC_INTERFACE 标志，那么同时也得设置ACC_ABSTRACT 标志（JLS §9.1.1.1）。同时它不能再设置 ACC_FINAL、ACC_SUPER 和 ACC_ENUM 标志。

注解类型必定带有 ACC_ANNOTATION 标记，如果设置了 ANNOTATION 标记，ACC_INTERFACE 也必须被同时设置。如果没有同时设置 ACC_INTERFACE 标记，那么这个Class文件可以具有表4.1中的除ACC_ANNOTATION外的所有其它标记。当然 ACC_FINAL 和 ACC_ABSTRACT 这类互斥的标记除外（JLS §8.1.1.2）。

ACC_SUPER 标志用于确定该 Class 文件里面的 invokespecial 指令使用的是哪一种执行语义。目前 Java 虚拟机的编译器都应当设置这个标志。ACC_SUPER 标记是为了向后兼容旧编译器编译的 Class 文件而存在的，在 JDK1.0.2 版本以前的编译器产生的 Class 文件中，access_flag 里面没有 ACC_SUPER 标志。同时，JDK1.0.2 前的 Java 虚拟机遇到 ACC_SUPER 标记会自动忽略它。

在表 4.1 中没有使用的 access_flags 标志位是为未来扩充而预留的，这些预留的标志为在编译器中会被设置为 0， Java 虚拟机实现也会自动忽略它们。

this_class

类索引，this_class 的值必须是对 constant_pool 表中项目的一个有效索引值。constant_pool 表在这个索引处的项必须为 CONSTANT_Class_info 类型常量，表示这个 Class 文件所定义的类或接口。

super_class

父类索引，对于类来说，super_class 的值必须为 0 或者是对 constant_pool 表中项目的一个有效索引值。如果它的值不为 0，那 constant_pool 表在这个索引处的项

必须为 CONSTANT_Class_info 类型常量（§4.4.1），表示这个 Class 文件所定义的类的直接父类。当前类的直接父类，以及它所有间接父类的 access_flag 中都不能带有 ACC_FINAL 标记。对于接口来说，它的 Class 文件的 super_class 项的值必须是对 constant_pool 表中项目的一个有效索引值。
constant_pool 表在这个索引处的
项必须为代表 java.lang.Object 的 CONSTANT_Class_info 类型常量（§4.4.1）。如果 Class 文件的 super_class 的值为 0，那这个 Class 文件只可能是定义的是
java.lang.Object 类，只有它是唯一没有父类的类。

interfaces_count

接口计数器，interfaces_count 的值表示当前类或接口的直接父接口数量。

interfaces[]

接口表，interfaces[]数组中的每个成员的值必须是一个对 constant_pool 表中项目的一个有效索引值，它的长度为 interfaces_count。每个成员 interfaces[i] 必须为 CONSTANT_Class_info 类型常量（§4.4.1），其中 0 ≤ i <interfaces_count。在 interfaces[]数组中，成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样，即 interfaces[0]对应的是源代码中最左边的接口。

fields_count

字段计数器，fields_count 的值表示当前 Class 文件 fields[]数组的成员个数。fields[]数组中每一项都是一个 field_info 结构（§4.5）的数据项，它用于表示该类或接口声明的类字段或者实例字段 。

注意：：类字段即被声明为 static 的字段，也称为类变量或者类属性，同样，实例字段是指未被声明为 static 的字段。由于《Java 虚拟机规范》中，“Variable”和“Attribute”出现频率很高且在大多数场景中具备其他含义，所以译文中统一把“Field”翻译为“字段”，即“类字段”、“实例字段”。

fields[]

字段表，fields[]数组中的每个成员都必须是一个 fields_info 结构（§4.5）的数据项，用于表示当前类或接口中某个字段的完整描述。fields[]数组描述当前类或接口
声明的所有字段，但不包括从父类或父接口继承的部分。

methods_count

方法计数器，methods_count 的值表示当前 Class 文件 methods[]数组的成员个数。Methods[]数组中每一项都是一个 method_info 结构（§4.5）的数据项。

methods[]

方法表，methods[]数组中的每个成员都必须是一个 method_info 结构（§4.6）的数据项，用于表示当前类或接口中某个方法的完整描述。如果某个 method_info 结构
的 access_flags 项既没有设置 ACC_NATIVE 标志也没有设置 ACC_ABSTRACT 标志，那么它所对应的方法体就应当可以被 Java 虚拟机直接从当前类加载，而不需要引用其它类。method_info 结构可以表示类和接口中定义的所有方法，包括实例方法、类方法、实例初始化方法方法（§2.9）和类或接口初始化方法方法（§2.9）。methods[]数组
只描述当前类或接口中声明的方法，不包括从父类或父接口继承的方法。

attributes_count

属性计数器，attributes_count 的值表示当前 Class 文件 attributes 表的成员个数。attributes 表中每一项都是一个 attribute_info 结构（§4.7）的数据项。

attributes[]

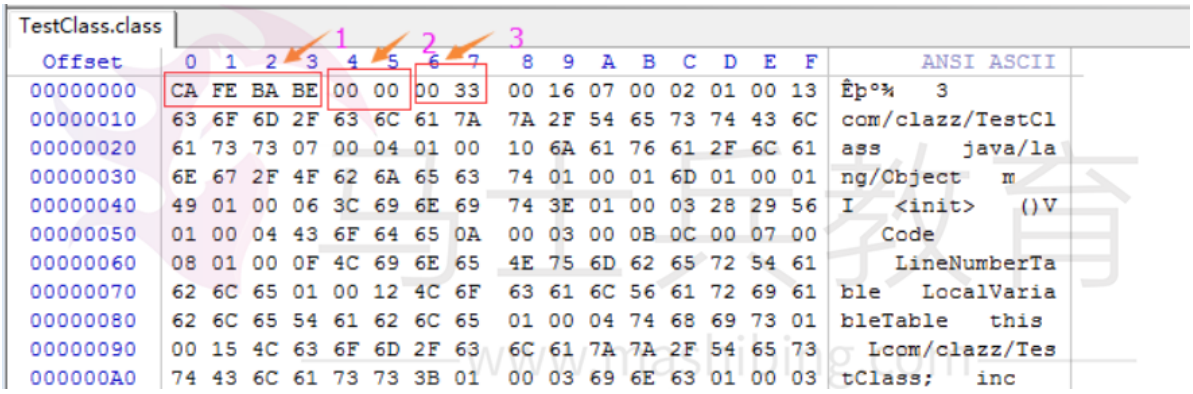
属性表，attributes 表的每个项的值必须是 attribute_info 结构（§4.7）。在本规范里，Class 文件结构中的 attributes 表的项包括下列定义的属性：
InnerClasses（§4.7.6）、EnclosingMethod（§4.7.7）、Synthetic（§4.7.8）、
Signature（§4.7.9）、SourceFile（§4.7.10），SourceDebugExtension（§4.7.11）、
Deprecated（§4.7.15）、RuntimeVisibleAnnotations（§4.7.16）、
RuntimeInvisibleAnnotations（§4.7.17）以及BootstrapMethods（§4.7.21）属性。对于支持 Class 文件格式版本号为 49.0 或更高的 Java 虚拟机实现，必须正确识别并读取 attributes 表中的
Signature（§4.7.9）、RuntimeVisibleAnnotations（§4.7.16）和
RuntimeInvisibleAnnotations（§4.7.17）属性。对于支持 Class 文件格式版本号为 51.0 或更高的 Java 虚拟机实现，必须正确识别并读取 attributes 表中的
BootstrapMethods（§4.7.21）属性。本规范要求任一 Java 虚拟机实现可以自动忽略 Class 文件的 attributes 表中的若干（甚至全部）它不可识别的属性项。任何本规范未定义的属性不能影响 Class 文件的语义，只能提供附加的描述信息（§4.7.1）。

这16个部分需要在实际应用中仔细研究一下。

列如下面代码：

```
1. package com.clazz;
2.
3. public class TestClass {
4.
5.  private int m;
6.  public int inc(){
7.      return m+1;
8.  }
9. }
```

经过编译之后class文件用WinHex编辑器打开你会发现



解读：

1>前4个字节16进制表示0xCAFEBABE 固定不变的魔数。

2>看到第5字节和6字节表示副版本号0x0000和主版本号0x0033也就是十进制51。查找class版本号可知这个class文件可以被JDK1.7.0 或者以上的虚拟机执行的class文件。

3>常量池计数器是从1开始计数，即上图中的偏移地址（0x00000008）即数字16那，换做十进制为22，也就是常量池中有21个常量。索引值范围1~21，

这里注意:将索引值设置为0时有特殊含义，不引用任何一个常量池项目的含义。Class文件中只有常量池的容量是从1计数开始。其它一般从0开始

4>常量池：常量池第一项，（偏移地址0x0000000A）是0x07，查看表6-3的标志发现它属于CONSTANT_Class_info类型。此类型的结构如上面的常量池的6-4图，其中tag是标志位已经说过了用于区分常量类型，name_index是一个索引值，它指向常量池中一个CONSTANT_Utf8_info类型的常量，这里name_index的值（偏移地址0x0000000B）为0x0002也指向了常量池的第二项。然后依次继续查找。。。

分析了TestClass.class常量池中的两个，其余19个常量也可以继续计算出来。这里我们可以借助jdk的计算器帮我们完成。在JDK的bin目录中有一个专门分析class文件的字节码工具：javap，使用命令 javap -verbose TestClass 参数输出文件字节码内容。如下：

```
F:\Java\jdk\jdk1.7.0_60\bin>javap -verbose TestClass.class
Classfile /F:/Java/jdk/jdk1.7.0_60/bin/TestClass.class
  Last modified 2017-10-16; size 373 bytes
  MD5 checksum 7d19b6fe8101f913758048f3529eaee4
  Compiled from "TestClass.java"
public class com.clazz.TestClass
  SourceFile: "TestClass.java"
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Class              #2             //  com/clazz/TestClass
   #2 = Utf8               com/clazz/TestClass
   #3 = Class              #4             //  java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               m
   #6 = Utf8               I
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Methodref          #3.#11         //  java/lang/Object."<init>":()V
  #11 = NameAndType        #7:#8          //  "<init>":()V
  #12 = Utf8               LineNumberTable
  #13 = Utf8               LocalVariableTable
  #14 = Utf8               this
  #15 = Utf8               Lcom/clazz/TestClass;
  #16 = Utf8               inc
  #17 = Utf8               ()I
  #18 = Fieldref           #1.#19         //  com/clazz/TestClass.m:I
  #19 = NameAndType        #5:#6          //  m:I
  #20 = Utf8               SourceFile
  #21 = Utf8               TestClass.java
{
  public com.clazz.TestClass();
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #10                 // Method java/lang/Object."<init>
":()V
         4: return
```

5>常量池结束后，紧接着的两个字节代表访问标识（access_flags）偏移量（0x000000DC）为0x0021=0x0001

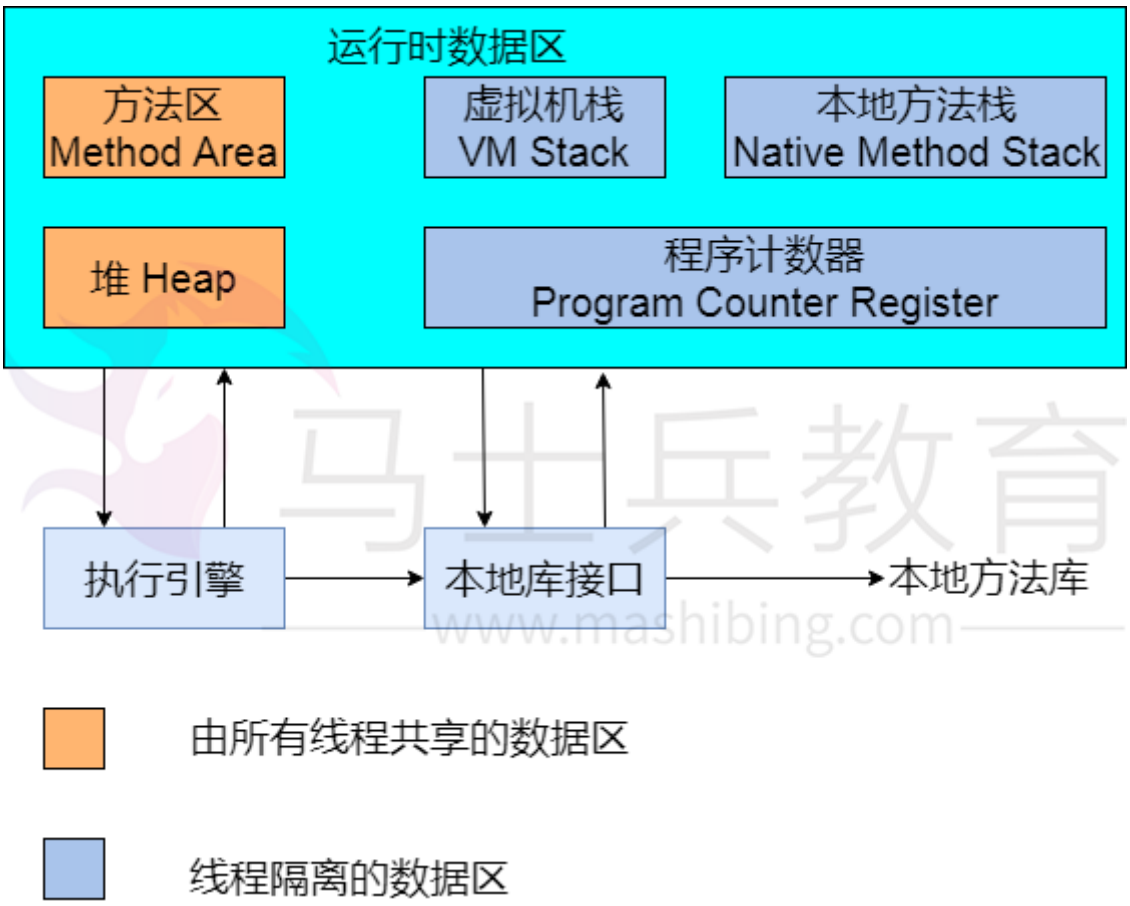
User user = new User() 做了什么操作，申请了哪些内存?

1. new User(); 创建一个User对象，内存分配在堆上
2. User user; 创建一个引用，内存分配在栈上
3. = 将User对象地址赋值给引用

Java的内存模型以及GC算法

JVM内存模型与GC算法

1.JVM内存模型



马士兵教育
www.mashibing.com

JVM内存模型如上图，需要声明一点，这是《Java虚拟机规范（Java SE 7版）》规定的内容，实际区域由各JVM自己实现，所以可能略有不同。以下对各区域进行简短说明。

1.1程序计数器

程序计数器是众多编程语言都共有的一部分，作用是标示下一条需要执行的指令的位置，分支、循环、跳转、异常处理、线程恢复等基础功能都是依赖程序计数器完成的。

对于Java的多线程程序而言，不同的线程都是通过轮流获得cpu的时间片运行的，这符合计算机组成原理的基本概念，因此不同的线程之间需要不停的获得运行，挂起等待运行，所以各线程之间的计数器互不影响，独立存储。这些数据区属于线程私有的内存。

1.2 Java虚拟机栈

VM虚拟机栈也是线程私有的，生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧(Stack Frame)用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法调用直至执行完的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

有人将java内存区域划分为栈与堆两部分，在这种粗略的划分下，栈标示的就是当前讲的虚拟机栈，或者是虚拟机栈对应的局部变量表。之所以说这种划分比较粗略是角度不同，这种划分方法关心的是新申请内存的存在空间，而我们目前谈论的是JVM整体的内存划分，由于角度不同，所以划分的方法不同，没有对与错。

局部变量表存放了编译期可知的各种基本类型，对象引用，和returnAddress。其中64位长的long和double占用了2个局部变量空间(slot)，其他类型都占用1个。这也从存储的角度上说明了long与double本质上的非原子性。局部变量表所需的内存在编译期间完成分配，当进入一个方法时，这个方法在栈帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表大小。

由于栈帧的进出栈，显而易见的带来了空间分配上的问题。如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverFlowError异常；如果虚拟机栈可以扩展，扩展时无法申请到足够的内存，将会抛出OutOfMemoryError。显然，这种情况大多数是由于循环调用与递归带来的。

1.3 本地方法栈

本地方法栈与虚拟机栈的作用十分类似，不过本地方法是为native方法服务的。部分虚拟机（比如 Sun HotSpot虚拟机）直接将本地方法栈与虚拟机栈合二为一。与虚拟机栈一样，本地方法栈也会抛出StactOverFlowError与OutOfMemoryError异常。

至此，线程私有数据区域结束，下面开始线程共享数据区。

1.4 Java堆

Java堆是虚拟机所管理的内存中最大的一块，在虚拟机启动时创建，此块内存的唯一目的就是存放对象实例，几乎所有的对象实例都在对上分配内存。JVM规范中的描述是：所有的对象实例以及数据都要在堆上分配。但是随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配(对象只存在于某方法中，不会逃逸出去，因此方法出栈后就会销毁，此时对象可以在栈上分配，方便销毁)，标量替换(新对象拥有的属性可以由现有对象替换拼凑而成，就没必要真正生成这个对象)等优化技术带来了一些变化，目前并非所有的对象都在堆上分配了。

当java堆上没有内存完成实例分配，并且堆大小也无法扩展是，将会抛出OutOfMemoryError异常。Java堆是垃圾收集器管理的主要区域。

1.5 方法区

方法区与java堆一样，是线程共享的数据区，用于存储被虚拟机加载的类信息、常量、静态变量、即时编译的代码。JVM规范将方法与堆区分开，但是HotSpot将方法区作为永久代(Permanent Generation)实现。这样方便将GC分代手机方法扩展至方法区，HotSpot的垃圾收集器可以像管理Java堆一样管理方法区。但是这种方向已经逐步在被HotSpot替换中，在JDK1.7的版本中，已经把原本存放在方法区的字符串常量区移出。

至此，JVM规范所声明的内存模型已经分析完毕，下面将分析一些经常提到的与内存相关的区域。

1.6 运行时常量池

运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等信息外，还有一项信息是常量池(Constant Poll Table)用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池存放。

其中字符串常量池属于运行时常量池的一部分，不过在HotSpot虚拟机中，JDK1.7将字符串常量池移到了java堆中，通过下面的实验可以很容易看到。

```
import java.util.ArrayList;
import java.util.List;

/**
 * Created by shining.cui on 2017/7/23.
 */
public class RunTimeContantPoolOOM {
    public static void main(String[] args) {
        List list = new ArrayList();
        int i = 0;
        while(true){
            list.add(String.valueOf(i++).intern());
        }
    }
}
```

在jdk1.6中，字符串常量区是在Perm Space中的，所以可以将Perm Spacce设置的小一些，XX:MaxPermSize=10M可以很快抛出异常：java.lang.OutOfMemoryError:Perm Space。

在jdk1.7以上，字符串常量区已经移到了Java堆中，设置-Xms:64m -Xmx:64m，很快就可以抛出异常java.lang.OutOfMemoryError:java.heap.space。

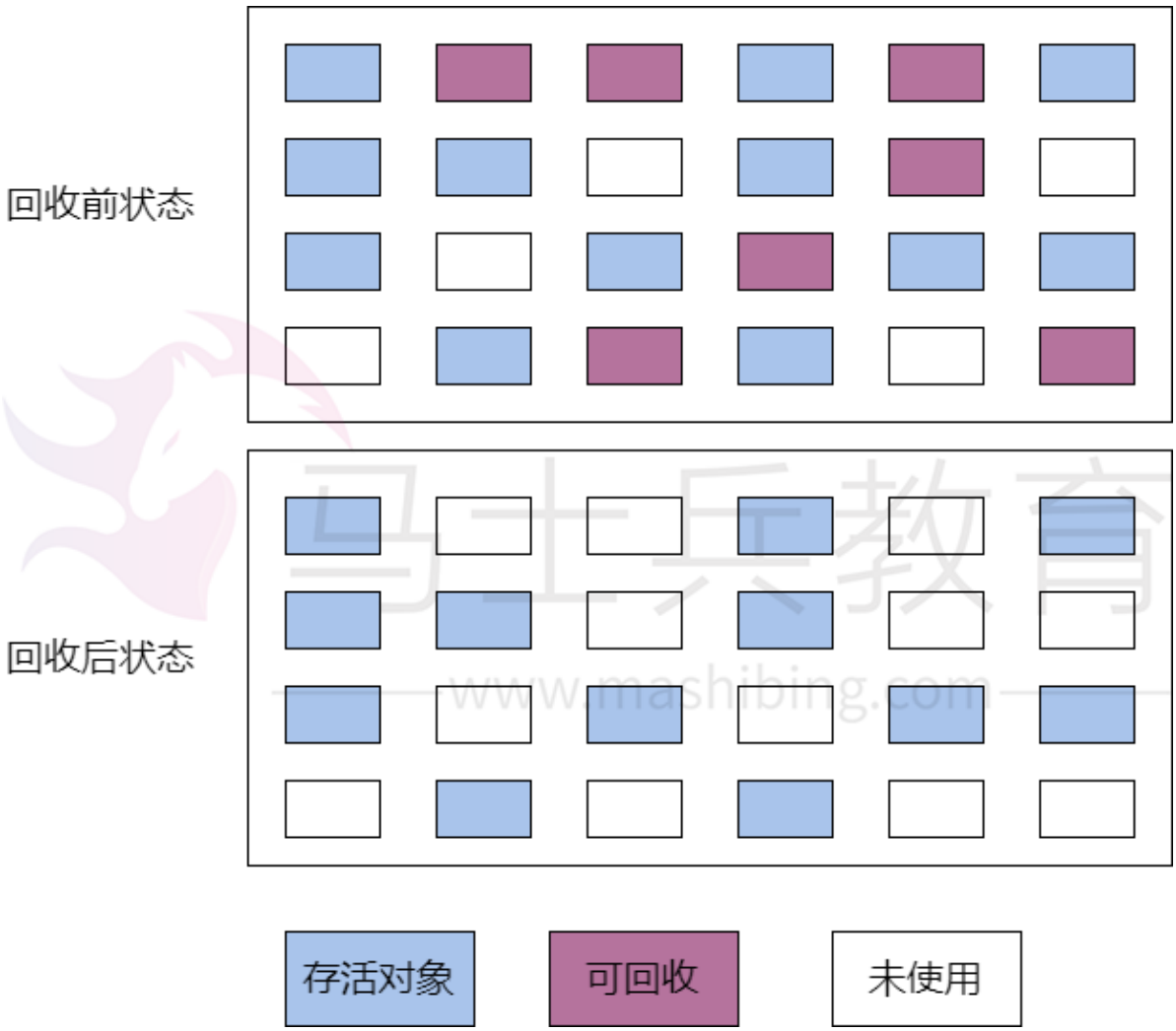
1.7 直接内存

直接内存不是JVM运行时的数据区的一部分，也不是Java虚拟机规范中定义的内存区域。在JDK1.4中引入了NIO(New Input/Output)类，引入了一种基于通道(Chanel)与缓冲区(Buffer)的I/O方式，他可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java中的DirectByteBuffer对象作为对这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java对和Native对中来回复制数据。

2.GC算法

2.1 标记-清除算法

最基础的垃圾收集算法是“标记-清除”(Mark Sweep)算法，正如名字一样，算法分为2个阶段：1.标记处需要回收的对象，2.回收被标记的对象。标记算法分为两种:1.引用计数算法(Reference Counting) 2.可达性分析算法(Reachability Analysis)。由于引用技术算法无法解决循环引用的问题，所以这里使用的标记算法均为可达性分析算法。



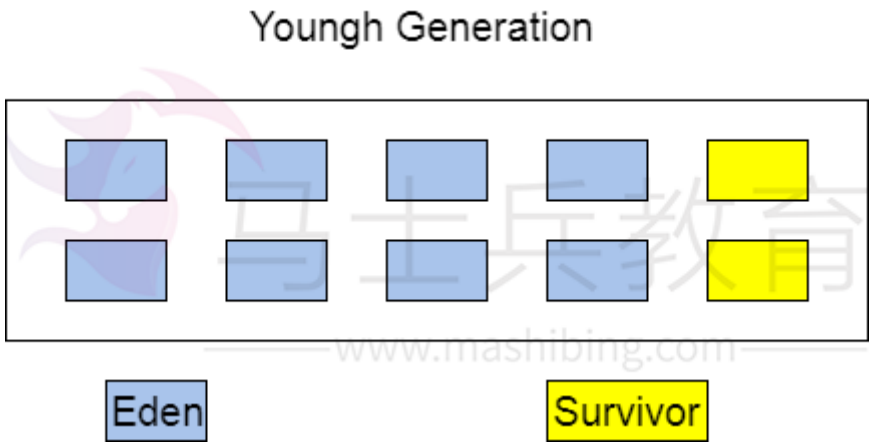
如图所示，当进行过标记清除算法之后，出现了大量的非连续内存。当java堆需要分配一段连续的内存给一个新对象时，发现虽然内存清理出了很多的空闲，但是仍然需要继续清理以满足“连续空间”的要求。所以说，这种方法比较基础，效率也比较低下。

2.2 复制算法

为了解决效率与内存碎片问题，复制(Copying)算法出现了，它将内存划分为两块相等的大小，每次使用一块，当这一块用完了，就讲还存活的对象复制到另外一块内存区域中，然后将当前内存空间一次性清理掉。这样的对整个半区进行回收，分配时按照顺序从内存顶端依次分配，这种实现简单，运行高效。不过这种算法将原有的内存空间减少为实际的一半，代价比较高。



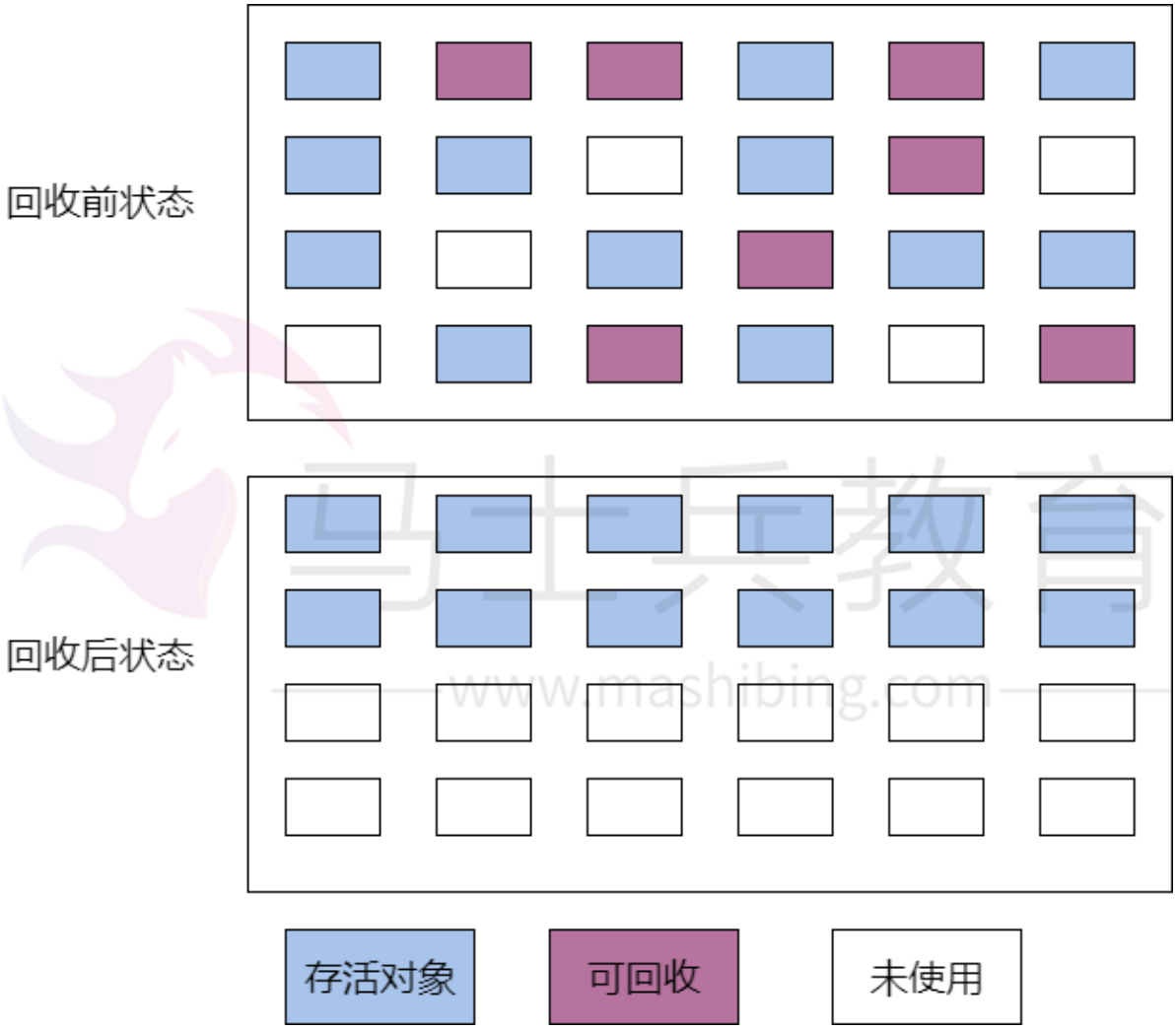
从图中可以看出，整理后的内存十分规整，但是白白浪费一般的内存成本太高。然而这其实是很重要的一个收集算法，因为现在的商业虚拟机都采用这种算法来回收新生代。IBM公司的专门研究表明，新生代中的对象98%都是“朝生夕死”的，所以不需要按照1:1的比例来划分内存。HotSpot虚拟机将Java堆划分为年轻代(Young Generation)、老年代(Tenured Generation），其中年轻代又分为一块Eden和两块Survivor。



所有的新建对象都放在年轻代中，年轻代使用的GC算法就是复制算法。其中Eden与Survivor的内存大小比例为8:2，其中Eden由1大块组成，Survivor由2小块组成。每次使用内存为1Eden+1Survivor，即90%的内存。由于年轻代中的对象生命周期往往很短，所以当需要进行GC的时候就将当前90%中存活的对象复制到另外一块Survivor中，原来的Eden与Survivor将被清空。但是这就有一个问题，我们无法保证每次年轻代GC后存活的对象都不高于10%。所以在当活下来的对象高于10%的时候，这部分对象将由Tenured进行担保，即无法复制到Survivor中的对象将移动到老年代。

2.3 标记-整理算法

复制算法在极端情况下(存活对象较多)效率变得很低，并且需要有额外的空间进行分配担保。所以在老年代中这种情况一般是不适合的。



所以就出现了标记-整理(Mark-Compact)算法。与标记清除算法一样，首先是标记对象，然而第二步是将存货的对象向内存一段移动，整理出一块较大的连续内存空间。

3. 总结

1. Java虚拟机规范中规定了对内存的分配，其中程序计数器、本地方法栈、虚拟机栈属于线程私有数据区，Java堆与方法区属于线程共享数据。
2. Jdk从1.7开始将字符串常量区由方法区(永久代)移动到了Java堆中。
3. Java从NIO开始允许直接操纵系统的直接内存，在部分场景中效率很高，因为避免了在Java堆与Native堆中来回复制数据。
4. Java堆分为年轻代有年老代，其中年轻代分为1个Eden与2个Survior，同时只有1个Eden与1个Survior处于使用中状态，又有年轻代的对象生存时间为往往很短，因此使用复制算法进行垃圾回收。
5. 年老代由于对象存活期比较长，并且没有可担保的数据区，所以往往使用标记-清除与标记-整理算法进行垃圾回收。

jvm性能调优都做了什么

JVM性能调优

一、JVM内存模型及垃圾收集算法

1.根据Java虚拟机规范，**JVM将内存划分为：**

- New（年轻代）
- Tenured（年老代）
- 永久代（Perm）

其中New和Tenured属于堆内存，堆内存会从JVM启动参数（-Xmx:3G）指定的内存中分配，**Perm不属于堆内存，有虚拟机直接分配**，但可以通过**-XX:PermSize -XX:MaxPermSize** 等参数调整其大小。

- 年轻代（New）：**年轻代用来存放JVM刚分配的Java对象**
- 年老代（Tenured）：年轻代中经过垃圾回收没有回收掉的对象将被Copy到年老代
- 永久代（Perm）：永久代存放Class、Method元信息，其大小跟项目的规模、类、方法的量有关，一般设置为128M就足够，设置原则是预留30%的空间。

New又分为几个部分：

- Eden：**Eden用来存放JVM刚分配的对象**
- Survivor1
- Survivro2：**两个Survivor空间一样大**，当Eden中的对象经过垃圾回收没有被回收掉时，会在两个Survivor之间来回Copy，当满足某个条件，比如Copy次数，就会被Copy到Tenured。显然，**Survivor只是增加了对象在年轻代中的逗留时间**，增加了被垃圾回收的可能性。

2.垃圾回收算法

垃圾回收算法可以分为三类，都基于标记-清除（复制）算法：

- Serial算法（单线程）
- 并行算法
- 并发算法

JVM会根据机器的硬件配置对每个内存代选择适合的回收算法，比如，如果机器多于1个核，会对年轻代选择并行算法，关于选择细节请参考JVM调优文档。

稍微解释下的是，**并行算法是用多线程进行垃圾回收，回收期间会暂停程序的执行，而并发算法，也是多线程回收，但期间不停止应用执行。**所以，并发算法适用于交互性高的一些程序。经过观察，并发算法会减少年轻代的大小，其实就是使用了一个大的年老代，这反过来跟并行算法相比吞吐量相对较低。

还有一个问题是，垃圾回收动作何时执行？

- 当年轻代内存满时，会引发一次普通GC，该GC仅回收年轻代。**需要强调的时，年轻代满是指Eden代满，Survivor满不会引发GC**
- 当年老代满时会引发Full GC，Full GC将会同时回收年轻代、年老代
- 当永久代满时也会引发Full GC，会导致Class、Method元信息的卸载

另一个问题是，何时会抛出OutOfMemoryException，**并不是内存被耗空的时候才抛出**

- JVM98%的时间都花费在内存回收
- 每次回收的内存小于2%

满足这两个条件将触发OutOfMemoryException，这将会留给系统一个微小的间隙以做一些Down之前的操作，比如手动打印Heap Dump。

二、内存泄漏及解决方法

1.系统崩溃前的一些现象：

- 每次垃圾回收的时间越来越长，由之前的10ms延长到50ms左右，FullGC的时间也有之前的0.5s延长到4、5s
- FullGC的次数越来越多，最频繁时隔不到1分钟就进行一次FullGC
- 年老代的内存越来越大并且每次FullGC后年老代没有内存被释放

之后系统会无法响应新的请求，逐渐到达OutOfMemoryError的临界值。

2.生成堆的dump文件

通过JMX的MBean生成当前的Heap信息，大小为一个3G（整个堆的大小）的hprof文件，如果没有启动JMX可以通过Java的jmap命令来生成该文件。

3.分析dump文件

下面要考虑的是如何打开这个3G的堆信息文件，显然一般的Window系统没有这么大的内存，必须借助高配置的Linux。当然我们可以借助X-Window把Linux上的图形导入到Window。我们考虑用下面几种工具打开该文件：

1. Visual VM
2. IBM HeapAnalyzer
3. JDK 自带的Hprof工具

使用这些工具时为了确保加载速度，建议设置最大内存为6G。使用后发现，这些工具都无法直观地观察到内存泄漏，Visual VM虽能观察到对象大小，但看不到调用堆栈；HeapAnalyzer虽然能看到调用堆栈，却无法正确打开一个3G的文件。因此，我们又选用了Eclipse专门的静态内存分析工具：Mat。

4.分析内存泄漏

通过Mat我们能清楚地看到，哪些对象被怀疑为内存泄漏，哪些对象占的空间最大及对象的调用关系。针对本案，在ThreadLocal中有很多的JbpmContext实例，经过调查是JBPM的Context没有关闭所致。

另，通过Mat或JMX我们还可以分析线程状态，可以观察到线程被阻塞在哪个对象上，从而判断系统的瓶颈。

5.回归问题

Q：为什么崩溃前垃圾回收的时间越来越长?

A:根据内存模型和垃圾回收算法，垃圾回收分两部分：内存标记、清除（复制），标记部分只要内存大小固定时间是不变的，变的是复制部分，因为每次垃圾回收都有一些回收不掉的内存，所以增加了复制量，导致时间延长。所以，垃圾回收的时间也可以作为判断内存泄漏的依据

Q：为什么Full GC的次数越来越多?

A：因此内存的积累，逐渐耗尽了年老代的内存，导致新对象分配没有更多的空间，从而导致频繁的垃圾回收

Q:为什么年老代占用的内存越来越大?

A:因为年轻代的内存无法被回收，越来越多地被Copy到年老代

三、性能调优

除了上述内存泄漏外，我们还发现CPU长期不足3%，系统吞吐量不够，针对8core×16G、64bit的Linux服务器来说，是严重的资源浪费。

在CPU负载不足的同时，偶尔会有用户反映请求的时间过长，我们意识到必须对程序及JVM进行调优。从以下几个方面进行：

- **线程池**：解决用户响应时间长的问题
- **连接池**
- **JVM启动参数**：调整各代的内存比例和垃圾回收算法，提高吞吐量
- **程序算法**：改进程序逻辑算法提高性能

1.Java线程池（java.util.concurrent.ThreadPoolExecutor）

大多数JVM6上的应用采用的线程池都是JDK自带的线程池，之所以把成熟的Java线程池进行罗嗦说明，是因为该线程池的行为与我们想象的有点出入。Java线程池有几个重要的配置参数：

- corePoolSize：核心线程数（最新线程数）
- maximumPoolSize：最大线程数，超过这个数量的任务会被拒绝，用户可以通过RejectedExecutionHandler接口自定义处理方式
- keepAliveTime：线程保持活动的时间
- workQueue：工作队列，存放执行的任务

  Java线程池需要传入一个Queue参数（workQueue）用来存放执行的任务，而对Queue的不同选择，线程池有完全不同的行为：

- `SynchronousQueue：` ``一个无容量的等待队列，一个线程的`insert`操作必须等待另一线程的`remove`操作，采用这个`Queue`线程池将会为每个任务分配一个新线程
- `LinkedBlockingQueue ：` 无界队列，采用该`Queue`，线程池将忽略 maximumPoolSize参数，仅用corePoolSize的线程处理所有的任务，未处理的任务便在`LinkedBlockingQueue`中排队
- `ArrayBlockingQueue：` 有界队列，在有界队列和 maximumPoolSize的作用下，程序将很难被调优：更大的Queue和小的maximumPoolSize将导致CPU的低负载；小的Queue和大的池，Queue就没起动应有的作用。

  其实我们的要求很简单，希望线程池能跟连接池一样，能设置最小线程数、最大线程数，当最小数<任务<最大数时，应该分配新的线程处理；当任务>最大数时，应该等待有空闲线程再处理该任务。

  但线程池的设计思路是，任务应该放到Queue中，当Queue放不下时再考虑用新线程处理，如果Queue满且无法派生新线程，就拒绝该任务。设计导致“先放等执行”、“放不下再执行”、“拒绝不等待”。所以，根据不同的Queue参数，要提高吞吐量不能一味地增大maximumPoolSize。

  当然，要达到我们的目标，必须对线程池进行一定的封装，幸运的是ThreadPoolExecutor中留了足够的自定义接口以帮助我们达到目标。我们封装的方式是：

- 以SynchronousQueue作为参数，使maximumPoolSize发挥作用，以防止线程被无限制的分配，同时可以通过提高maximumPoolSize来提高系统吞吐量
- 自定义一个RejectedExecutionHandler，当线程数超过maximumPoolSize时进行处理，处理方式为隔一段时间检查线程池是否可以执行新Task，如果可以把拒绝的Task重新放入到线程池，检查的时间依赖keepAliveTime的大小。

2.连接池（org.apache.commons.dbcp.BasicDataSource）

在使用org.apache.commons.dbcp.BasicDataSource的时候，因为之前采用了默认配置，所以当访问量大时，通过JMX观察到很多Tomcat线程都阻塞在BasicDataSource使用的Apache ObjectPool的锁上，直接原因当时是因为BasicDataSource连接池的最大连接数设置的太小，默认的BasicDataSource配置，仅使用8个最大连接。

我还观察到一个问题，当较长的时间不访问系统，比如2天，DB上的Mysql会断掉所以的连接，导致连接池中缓存的连接不能用。为了解决这些问题，我们充分研究了BasicDataSource，发现了一些优化的点：

- Mysql默认支持100个链接，所以每个连接池的配置要根据集群中的机器数进行，如有2台服务器，可每个设置为60
- initialSize：参数是一直打开的连接数
- minEvictableIdleTimeMillis：该参数设置每个连接的空闲时间，超过这个时间连接将被关闭
- timeBetweenEvictionRunsMillis：后台线程的运行周期，用来检测过期连接
- maxActive：最大能分配的连接数
- maxIdle：最大空闲数，当连接使用完毕后发现连接数大于maxIdle，连接将被直接关闭。只有initialSize < x < maxIdle的连接将被定期检测是否超期。这个参数主要用来在峰值访问时提高吞吐量。
- initialSize是如何保持的？经过研究代码发现，BasicDataSource会关闭所有超期的连接，然后再打开initialSize数量的连接，这个特性与minEvictableIdleTimeMillis、timeBetweenEvictionRunsMillis一起保证了所有超期的initialSize连接都会被重新连接，从而避免了Mysql长时间无动作会断掉连接的问题。

3.JVM参数

在JVM启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置JVM会工作的很好，但对一些配置很好的Server和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标：

- **GC的时间足够的小**
- **GC的次数足够的少**
- **发生Full GC的周期足够的长**

前两个目前是相悖的，要想GC时间小必须要一个更小的堆，要保证GC次数足够少，必须保证一个更大的堆，我们只能取其平衡。

（1）针对JVM堆的设置，一般可以通过-Xms -Xmx限定其最小、最大值，**为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值**

（2）**年轻代和年老代将根据默认的比例（1：2）分配堆内存**，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小

（3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

- **更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC**
- **更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少Full GC的频率**
- 如何选择应该依赖应用程序**对象生命周期的分布情况**：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 （B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间

（4）**在配置较好的机器上（比如多核、大内存），可以为年老代选择并行收集算法**：**-XX:+UseParallelOldGC**，默认为Serial收集

（5）线程堆栈的设置：每个线程默认会开启1M的堆栈，用于存放栈帧、调用参数、局部变量等，对大多数应用而言这个默认值太了，一般256K就足用。理论上，在内存不变的情况下，减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

（4）可以通过下面的参数打Heap Dump信息

- -XX:HeapDumpPath
- -XX:+PrintGCDetails
- -XX:+PrintGCTimeStamps
- -Xloggc:/usr/aaa/dump/heap_trace.txt

  通过下面参数可以控制OutOfMemoryError时打印堆的信息
- -XX:+HeapDumpOnOutOfMemoryError

请看一下一个时间的Java参数配置：（服务器：Linux 64Bit，8Core×16G）

JAVA_OPTS="$JAVA_OPTS -server -Xms3G -Xmx3G -Xss256k -XX:PermSize=128m -XX:MaxPermSize=128m -XX:+UseParallelOldGC -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/aaa/dump -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:/usr/aaa/dump/heap_trace.txt -XX:NewSize=1G -XX:MaxNewSize=1G"

经过观察该配置非常稳定，每次普通GC的时间在10ms左右，Full GC基本不发生，或隔很长很长的时间才发生一次

通过分析dump文件可以发现，每个1小时都会发生一次Full GC，经过多方求证，只要在JVM中开启了JMX服务，JMX将会1小时执行一次Full GC以清除引用，关于这点请参考附件文档。

4.程序算法调优

==========

调优方法

一切都是为了这一步，调优，在调优之前，我们需要记住下面的原则：

1、多数的Java应用不需要在服务器上进行GC优化；

2、多数导致GC问题的Java应用，都不是因为我们参数设置错误，而是代码问题；

3、在应用上线之前，先考虑将机器的JVM参数设置到最优（最适合）；

4、减少创建对象的数量；

5、减少使用全局变量和大对象；

6、GC优化是到最后不得已才采用的手段；

7、在实际使用中，分析GC情况优化代码比优化GC参数要多得多；

GC优化的目的有两个：

1、将转移到老年代的对象数量降低到最小；

2、减少full GC的执行时间；

为了达到上面的目的，一般地，你需要做的事情有：

1、减少使用全局变量和大对象；

2、调整新生代的大小到最合适；

3、设置老年代的大小为最合适；

4、选择合适的GC收集器；

在上面的4条方法中，用了几个“合适”，那究竟什么才算合适，一般的，请参考上面“收集器搭配”和“启动内存分配”两节中的建议。但这些建议不是万能的，需要根据您的机器和应用情况进行发展和变化，实际操作中，可以将两台机器分别设置成不同的GC参数，并且进行对比，选用那些确实提高了性能或减少了GC时间的参数。

真正熟练的使用GC调优，是建立在多次进行GC监控和调优的实战经验上的，进行监控和调优的一般步骤为：

1，监控GC的状态

使用各种JVM工具，查看当前日志，分析当前JVM参数设置，并且分析当前堆内存快照和gc日志，根据实际的各区域内存划分和GC执行时间，觉得是否进行优化；

2，分析结果，判断是否需要优化

如果各项参数设置合理，系统没有超时日志出现，GC频率不高，GC耗时不高，那么没有必要进行GC优化；如果GC时间超过1-3秒，或者频繁GC，则必须优化；

注：**如果满足下面的指标，则一般不需要进行GC：**

Minor GC执行时间不到50ms；

Minor GC执行不频繁，约10秒一次；

Full GC执行时间不到1s；

Full GC执行频率不算频繁，不低于10分钟1次；

3，调整GC类型和内存分配

如果内存分配过大或过小，或者采用的GC收集器比较慢，则应该优先调整这些参数，并且先找1台或几台机器进行beta，然后比较优化过的机器和没有优化的机器的性能对比，并有针对性的做出最后选择；

4，不断的分析和调整

通过不断的试验和试错，分析并找到最合适的参数

5，全面应用参数

如果找到了最合适的参数，则将这些参数应用到所有服务器，并进行后续跟踪。

调优实例

上面的内容都是纸上谈兵，下面我们以一些真实例子来进行说明：

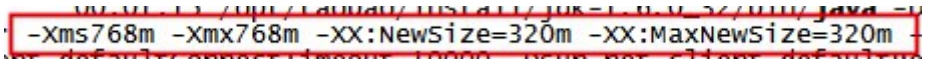
实例1：

笔者昨日发现部分开发测试机器出现异常：java.lang.OutOfMemoryError: GC overhead limit exceeded，这个异常代表：

GC为了释放很小的空间却耗费了太多的时间，其原因一般有两个：1，堆太小，2，有死循环或大对象；

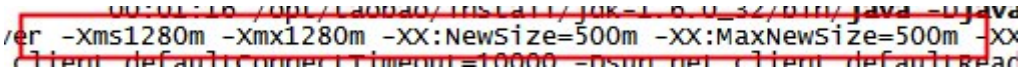
笔者首先排除了第2个原因，因为这个应用同时是在线上运行的，如果有问题，早就挂了。所以怀疑是这台机器中堆设置太小；

使用ps -ef |grep "java"查看，发现：



该应用的堆区设置只有768m，而机器内存有2g，机器上只跑这一个java应用，没有其他需要占用内存的地方。另外，这个应用比较大，需要占用的内存也比较多；

笔者通过上面的情况判断，只需要改变堆中各区域的大小设置即可，于是改成下面的情况：



跟踪运行情况发现，相关异常没有再出现；

实例2：

一个服务系统，**经常出现卡顿，分析原因，发现Full GC时间太长**：

jstat -gcutil:

S0 S1 E O P YGC YGCT FGC FGCT GCT

12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993

分析上面的数据，发现Young GC执行了54次，耗时2.047秒，每次Young GC耗时37ms，在正常范围，**而Full GC执行了5次，耗时6.946秒，每次平均1.389s，数据显示出来的问题是：Full GC耗时较长**，分析该系统的是指发现，NewRatio=9，也就是说，新生代和老生代大小之比为1:9，这就是问题的原因：

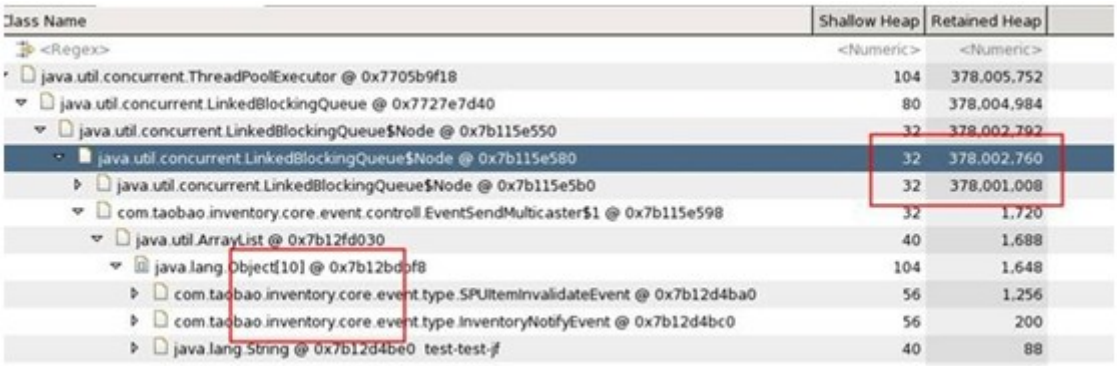
1，新生代太小，导致对象提前进入老年代，触发老年代发生Full GC；

2，老年代较大，进行Full GC时耗时较大；

优化的方法是调整NewRatio的值，调整到4，发现Full GC没有再发生，只有Young GC在执行。这就是把对象控制在新生代就清理掉，没有进入老年代（这种做法对一些应用是很有用的，但并不是对所有应用都要这么做）

实例3：

一应用在性能测试过程中，发现内存占用率很高，Full GC频繁，使用sudo -u admin -H jmap -dump:format=b,file=文件名.hprof pid 来dump内存，生成dump文件，并使用Eclipse下的mat差距进行分析，发现：



从图中可以看出，这个线程存在问题，队列LinkedBlockingQueue所引用的大量对象并未释放，导致整个线程占用内存高达378m，此时通知开发人员进行代码优化，将相关对象释放掉即可。

java classload 机制 详解

类加载器是 Java 语言的一个创新，也是 Java 语言流行的重要原因之一。它使得 Java 类可以被动态加载到 Java 虚拟机中并执行。类加载器从 JDK 1.0 就出现了，最初是为了满足 Java Applet 的需要而开发出来的。Java Applet 需要从远程下载 Java 类文件到浏览器中并执行。现在类加载器在 Web 容器和 OSGi 中得到了广泛的使用。一般来说，Java 应用的开发人员不需要直接同类加载器进行交互。Java 虚拟机默认的行为就已经足够满足大多数情况的需求了。不过如果遇到了需要与类加载器进行交互的情况，而对类加载器的机制又不是很了解的话，就很容易花大量的时间去调试

ClassNotFoundException 和 NoClassDefFoundError 等异常。本文将详细介绍 Java 的类加载器，帮助读者深刻理解 Java 语言中的这个重要概念。下面首先介绍一些相关的基本概念。

类加载器基本概念

类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 newInstance()方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。 基本上所有的类加载器都是 java.lang.ClassLoader 类的一个实例。下面详细介绍这个 Java 类。

java.lang.ClassLoader 类介绍

java.lang.ClassLoader 类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个ava 类，即 java.lang.Class 类的一个实例。除此之外，ClassLoader 还负责加载 Java
应用所需的资源，如图像文件和配置文件等。不过本文只讨论其加载类的功能。为了完成加载类的这个职责，ClassLoader 提供了一系列的方法，比较重要的方法如 表 1 所示。关于这些方法的细节会在下面进行介绍。

方法	说明
getParent()	返回该类加载器的父类加载器
loadClass(String name)	加载名称为 name 的类，返回的结果是 java.lang.Class 类的实例
findClass(String name)	查找名称为 name 的类，返回的结果是 java.lang.Class 类的实例
findLoadedClass(String name)	查找名称为 name 的已经被加载过的类，返回的结果是 java.lang.Class 类的实例。
defineClass(String name, byte[] b, int off, int len)	把字节数组 b 中的内容转换成 Java 类，返回的结果是 java.lang.Class 类的实例。这个方法被声明为 final 的
resolveClass(Class<?> c)	链接指定的 Java 类

表 1. ClassLoader 中与加载类相关的方法

对于 表 1 中给出的方法，表示类名称的 name 参数的值是类的二进制名称。需要注意的是内部类的表示，如
com.example.Sample$1 和 com.example.Sample$Inner 等表示方式。这些方法会在下面介绍类加载器的工作机制时，做进一步的说明。下面介绍类加载器的树状组织结构。

类加载器的树状组织结构

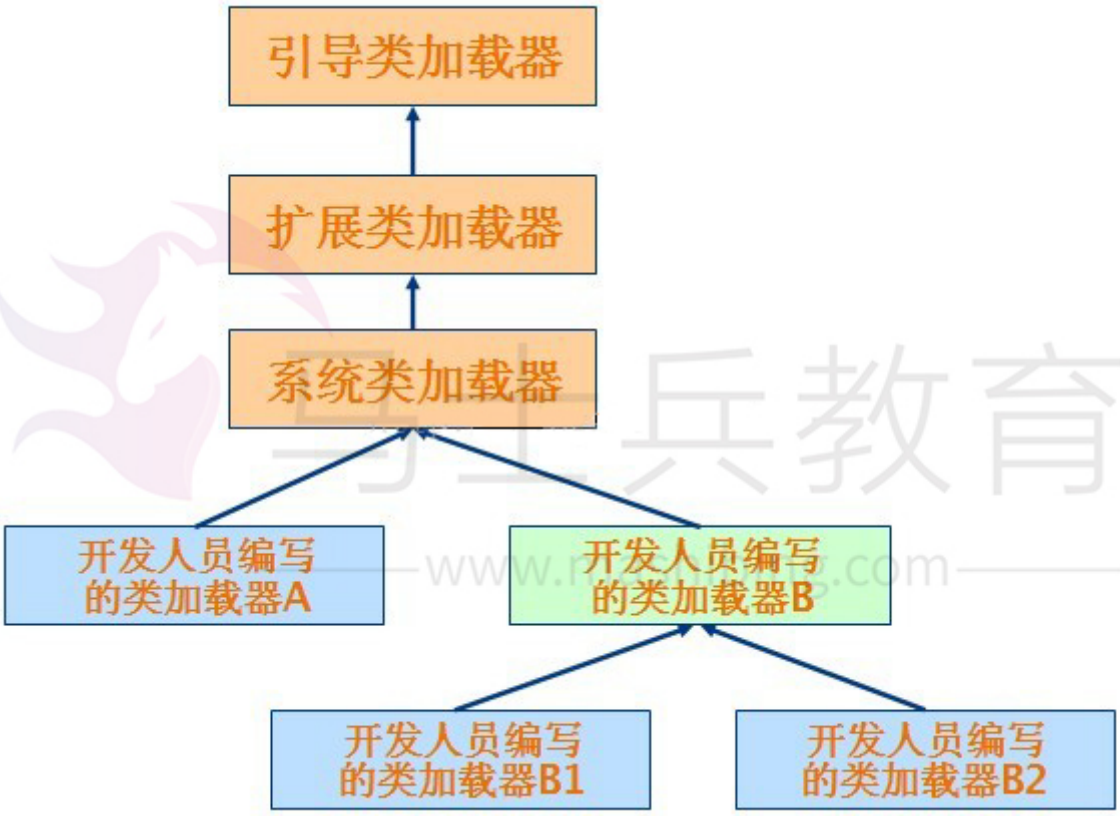
Java 中的类加载器大致可以分成两类，一类是系统提供的，另外一类则是由 Java 应用开发人员编写的。系统提供的类加载器主要有下面三个：

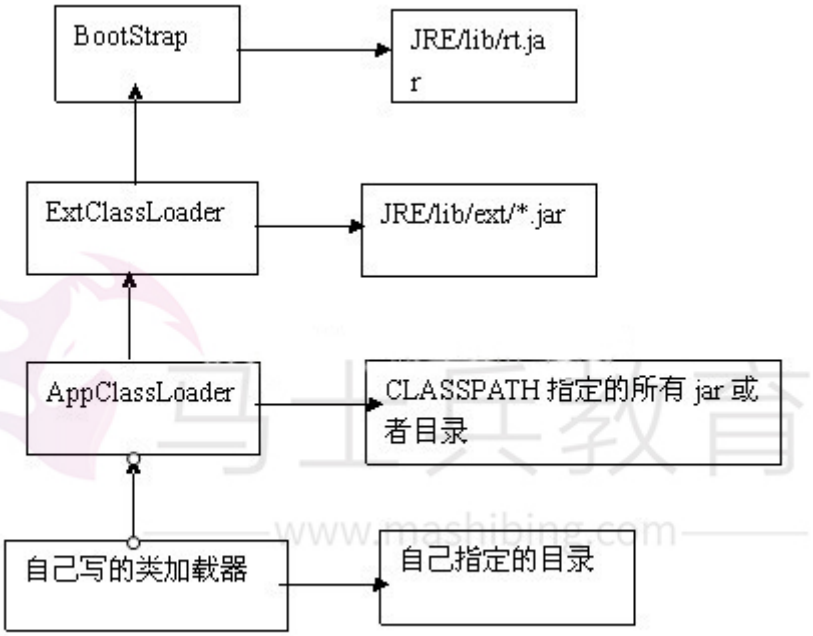
- **引导类加载器（bootstrap class loader）**：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自

java.lang.ClassLoader。

- **扩展类加载器（extensions class loader）**：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- **系统类加载器（system class loader/App classloader）**：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader() 来获取它。

  除了系统提供的类加载器以外，开发人员可以通过继承 java.lang.ClassLoader 类的方式实现自己的类加载器，以满足一些特殊的需求。
  除了引导类加载器之外，***所有的类加载器都有一个父类加载器。通过 表 1 中给出的 getParent() 方法可以得到***。对于系统提供的类加载器来说，system classloader的父类加载器是extensions cloassloader，而extensions cloassloader的父类加载器是bootstrap classloader；对于开发人员编写的类加载器来说，其父类加载器是加载此类加载器 Java 类的类加载器。因为类加载器 Java 类如同其它的 Java 类一样，也是要由类加载器来加载的。一般来说，开发人员编写的类加载器的父类加载器是system classloader。类加载器通过这种方式组织起来，形成树状结构。树的根节点就是bootstrap classloader。图 1 中给出了一个典型的类加载器树状组织结构示意图，其中的箭头指向的是父类加载器。





代码清单 1 演示了类加载器的树状组织结构。

```
<span style="font-size:18px;">public class ClassLoaderTree {

    public static void main(String[] args) {
        ClassLoader loader = ClassLoaderTree.class.getClassLoader();
        while (loader != null) {
            System.out.println(loader.getClass().getName());
            loader = loader.getParent();
        }
        System.out.println(loader);//最后当loader=null的时候。这个时候loader代表的是引导类加载器BootStrap
    }
 }</span>
```

输出结果：

```
sun.misc.Launcher$AppClassLoader
sun.misc.Launcher$ExtClassLoader
null
```

第一个输出的是 `ClassLoaderTree` 类的类加载器，即系统类加载器。它是 `sun.misc.Launcher$AppClassLoader` 类的实例；第二个输出的是扩展类加载器，是 `sun.misc.Launcher$ExtClassLoader` 类的实例。需要注意的是这里并没有输出引导类加载器，这是由于有些 JDK 的实现对于父类加载器是引导类加载器的情况， `getParent()` 方法返回 `null` 。

类加载器的代理模式

在了解了类加载器的树状组织结构之后，下面介绍类加载器的代理模式。

类加载器在尝试自己去查找某个类的字节代码并定义它时，会先代理给其父类加载器，由父类加载器先去尝试加载这个类，依次类推。在介绍代理模式背后的动机之前，首先需要说明一下 Java 虚拟机是如何判定两个 Java 类是相同的。***Java* *虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。***比如一个 Java 类 `com.tao.test.` ClassTest，编译之后生成了字节代码文件 `ClassTest.class` 。两个不同的类加载器 `ClassLoaderA` 和 `ClassLoaderB` 分别读取了这个 `ClassTest.class` 文件，并定义出两个 `java.lang.Class` 类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 `ClassCastException` 。下面通过上面一讲的示例来具体比较

```
public class Test {
    public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException {
        Class clazz1=new MyClassLoader().loadClass("com.tao.test.ClassTest");
        Class clazz2=new MyClassLoader().loadClass("com.tao.test.ClassTest");
        //这是由不同的类加载器加载的类生成的对象
        InterfaceTest classTest1=(InterfaceTest) clazz1.newInstance();
        InterfaceTest classTest2=(InterfaceTest) clazz2.newInstance();
        //这是Date对象，是由同一个类加载器加载的
        Date date1=new Date();
        Date date2=new Date();
        //比较他们是否是同一个Class类的实例
```

```
        System.out.println(date1.getClass().equals(date2.getClass()));
System.out.println(classTest1.getClass().equals(classTest2.getClass()));
    }
}
```

运行结果：

```
true
false
```

类加载器是怎么进行委托的呢，我们可以查看下ClassLoader的源码分析下。

因为，在加载一个类的时候，都是调用loadClass()方法，所以具体看下loadClass()方法。

```
//有两个loadClass()方法，不过最终都是调用第二个方法
public Class<?> loadClass(String name) throws ClassNotFoundException {
        return loadClass(name, false);
    }

//最终调用的loadClass()方法
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        //异步保护，防止重复加载一个相同的类
        synchronized (getClassLoadingLock(name)) {
            // 检测这个类是否被加载了
            Class c = findLoadedClass(name);
            //如果没有被加载，首先让父类去加载
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {//父类加载器不是null，也就是说不是BootStrap加
载器
                        c = parent.loadClass(name, false);
                    } else {//父类加载器是BootStrap加载器
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }
                 //如果经过父类加载器之后，还是null，也就是说父类无法加载，那么再由自己完成
加载
                  if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    long t1 = System.nanoTime();
                    c = findClass(name);//调用自己的findClass()方法

                    // this is the defining class loader; record the stats
                    sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 -
t0);

 sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
```

```
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }
```

所以，可以很清楚的看到，加载一个类的过程，它是层层的像父类委托，然后在层层的向下加载。用这样的一种委托机制，到底有什么好处呢？为什么要这样做呢？

原因如下：

1、节约系统资源。只要，这个类已经被加载过了，就不会在次加载。

2、保证 Java 核心库的类型安全。所有 Java 应用都至少需要引用 `java.lang.Object` 类，也就是说在运行的时候，`java.lang.Object` 这个类需要被加载到 Java 虚拟机中。如果这个加载过程由 Java 应用自己的类加载器来完成的话，很可能就存在多个版本的 `java.lang.Object` 类，而且这些类之间是不兼容的。通过代理模式，对于 Java 核心库的类的加载工作由引导类加载器来统一完成，保证了 Java 应用所使用的都是同一个版本的 Java 核心库的类，是互相兼容的。

> 下面讨论另外一种类加载器：线程上下文类加载器。

线程上下文类加载器

线程上下文类加载器（context class loader）是从 JDK 1.2 开始引入的。类 java.lang.Thread 中的方法 getContextClassLoader() 和 setContextClassLoader(ClassLoader cl) 用来获取和设置线程的上下文类加载器。如果没有通过 setContextClassLoader(ClassLoader cl)方法进行设置的话，线程将继承其父线程的上下文类加载器。**Java 应用运行的初始线程的上下文类加载器是系统类加载器。在线程中运行的代码可以通过此类加载器来加载类和资源。**

前面提到的类加载器的代理模式并不能解决 Java 应用开发中会遇到的类加载器的全部问题。Java 提供了很多服务提供者接口（Service Provider Interface，SPI），允许第三方为这些接口提供实现。常见的 SPI 有 JDBC、JCE、JNDI、JAXP 和 JBI 等。这些 SPI 的接口由 Java 核心库来提供，如 JAXP 的 SPI 接口定义包含在 javax.xml.parsers 包中。这些 SPI 的实现代码很可能是作为 Java 应用所依赖的 jar 包被包含进来，可以通过类路径（CLASSPATH）来找到，如实现了 JAXP SPI 的 Apache Xerces 所包含的 jar 包。SPI 接口中的代码经常需要加载具体的实现类。如 JAXP 中的 javax.xml.parsers.DocumentBuilderFactory 类中的 newInstance() 方法用来生成一个新的 DocumentBuilderFactory 的实例。这里的实例的真正的类是继承自 javax.xml.parsers.DocumentBuilderFactory，由 SPI 的实现所提供的。如在 Apache Xerces 中，实现的类是 org.apache.xerces.jaxp.DocumentBuilderFactoryImpl。而问题在于，SPI 的接口是 Java 核心库的一部分，是由引导类加载器(extension classloader)来加载的；SPI 实现的 Java 类一般是由系统类加载器(system classloader)来加载的。引导类加载器是无法找到 SPI 的实现类的，因为它只加载 Java 的核心库。它也不能代理给系统类加载器，因为它是系统类加载器的祖先类加载器。也就是说，类加载器的代理模式无法解决这个问题。 线程上下文类加载器正好解决了这个问题。如果不做任何的设置，Java 应用的线程的上下文类加载器默认就是系统上下文类加载器。在 SPI 接口的代码中使用线程上下文类加载器，就可以成功的加载到 SPI 实现的类。线程上下文类加载器在很多 SPI 的实现中都会用到。

下面介绍另外一种加载类的方法：Class.forName。

Class.forName

Class.forName 是一个静态方法，同样可以用来加载类。该方法有两种形式：

Class.forName(String name, boolean initialize, ClassLoader loader)

Class.forName(String className)。

第一种形式的参数 name 表示的是类的全名；initialize 表示是否初始化类；loader表示加载时使用的类加载器。第二种形式则相当于设置了参数 initialize 的值为 true，loader 的值为当前类的类加载器。Class.forName 的一个很常见的用法是在加载数据库驱动的时候。如 Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance() 用来加载 Apache Derby 数据库的驱动。

在介绍完类加载器相关的基本概念之后，下面介绍如何开发自己的类加载器。

开发自己的类加载器

虽然在绝大多数情况下，系统默认提供的类加载器实现已经可以满足需求。但是在某些情况下，您还是需要为应用开发出自己的类加载器。比如您的应用通过网络来传输 Java 类的字节代码，为了保证安全性，这些字节代码经过了加密处理。这个时候您就需要自己的类加载器来从某个网络地址上读取加密后的字节代码，接着进行解密和验证，最后定义出要在 Java 虚拟机中运行的类来。下面将通过两个具体的实例来说明类加载器的开发。

下面，详细的介绍自定义一个类加载器的过程。

一、首先，写出一个接口，然后用一个类实现该接口，该类作为测试类，即我们自定义ClassLoader要加载的类。

接口：

```
/**
 * 要加载类的接口，加载该接口的子类时，可以用接口引用，而不需要利用反射来实现。
 */
public interface InterfaceTest {
    public void name();
    public void age();
}
```

测试类：

```
/**
 * 测试类，自定义类加载器去加载该类
 */
public class ClassTest implements InterfaceTest{

    @Override
    public void name() {
        System.out.println("tao");
    }

    @Override
    public void age() {
        System.out.println("21");
    }

}
```

二、自定义一个加密类，用来加密测试类的字节码文件。

到我们要加密的.class文件的位置：
E:\workspace.fu\ClassLoaderTest\bin\com\tao\test\ClassTest.class

加密后的.class文件要存储的位置，这里将它直接放到E盘根目录：E:\ClassTest.class

```
/**
 * 用加密算法生成要隐藏的字节码文件
 */
public class ClassEncrypt extends MyClassLoader{
    public static void main(String[] args) throws IOException {
        //要加密的字节码.class文件
        String
srcPath="E:/workspace.fu/ClassLoaderTest/bin/com/tao/test/ClassTest.class";
        //加密之后输出的字节码.class文件的位置
        String destPath="E:/ClassTest.class";
        FileInputStream fis=new FileInputStream(srcPath);
        FileOutputStream ofs=new FileOutputStream(destPath);
        cypher(fis, ofs);//加密
        fis.close();
        ofs.close();
    }

    //简单的加密，用于测试。将所有二进制位取反，即0变成1，1变成0
    private static void cypher(InputStream in,OutputStream out) throws
IOException{
        int b=-1;
        while((b=in.read())!=-1){
            out.write(b^0xff);
        }
    }
}
```

运行该类，那么，我们就已经对ClassTest.class文件加密成功，打开E盘，可以发现根目录下已经有了一个ClassTest.class文件。

可以将E:\workspace.fu\ClassLoaderTest\bin\com\tao\test的ClassTest.class文件删除，并且删除ClassTest.java文件和加密类ClassEncrypt.java。

三、编写我们自己的类加载器，必须继承ClassLoader，然后覆盖findClass()方法。

ClassLoader超类的loadClass方法用于将类的加载操作委托给父类加载器去进行，只有该类尚未加载并且父类加载器也无法加载该类时，才调用findClass()方法。

如果要实现该方法，必须做到以下几点：

（1）、为来自本地文件系统或者其他来源的类加载其字节码

（2）、调用ClassLoader超类的defineClass()方法，向虚拟机提供字节码

```
/**
 * 自定义的类加载器
 */
public class MyClassLoader extends ClassLoader{

    /**
     * 因为类加载器是基于委托机制，所以我们只需要重写findClass方法。
     * 它会自动向父类加载器委托，如果父类没有找到，就会再去调用我们重写的findClass方法加载
     */
    @Override

    protected Class<?> findClass(String name) throws ClassNotFoundException {
```

```
        try {
            //需要加载的.class字节码的位置
            String classPath="E:/ClassTest.class";

            FileInputStream fis=new FileInputStream(classPath);
            ByteArrayOutputStream bos=new ByteArrayOutputStream();
            cypher(fis, bos);
            fis.close();
            byte[] bytes=bos.toByteArray();
            return defineClass(bytes, 0, bytes.length);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return super.findClass(name);
    }

    //相应的字节码解密类，在加载E盘根目录下的被加密过的ClassTest.class字节码的时候，进行相应的解密。
    private static void cypher(InputStream in,OutputStream out) throws IOException{
        int b=-1;
        while((b=in.read())!=-1){
            out.write(b^0xff);
        }
    }
}
```

四、最后写一个测试类，测试我们的类加载器

```
public class Test {
    public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException {
        Class clazz=new MyClassLoader().loadClass("com.tao.test.ClassTest");
            //这就是我们接口的作用。如果没有接口，就需要利用反射来实现了。
            InterfaceTest classTest=(InterfaceTest) clazz.newInstance();
        classTest.name();
        classTest.age();
    }
}
```

在介绍完如何开发自己的类加载器之后，下面说明类加载器和 web 容器的关系。
类加载器与 web 容器

对于运行在 Java EE? 容器中的 Web 应用来说，类加载器的实现方式与一般的 Java 应用有所不同。不同的 Web 容器的实现方式也会有所不同。以 Apache Tomcat 来说，每个 Web 应用都有一个对应的类加载器实例。**该类加载器也使用代理模式，所不同的是它是首先尝试去加载某个类，如果找不到再代理给父类加载器。这与一般类加载器的顺序是相反的。这是 *Java Servlet 规范中的推荐做法，其目的是使得 Web 应用自己的类的优先级高于 Web 容器提供的类。这种代理模式的一个例外是：Java* *核心库的类是不在查找范围之内的。这也是为了保证 Java 核心库的类型安全。*** 绝大多数情况下，Web 应用的开发人员不需要考虑与类加载器相关的细节。下面给出几条简单的原则：

- 每个 Web 应用自己的 Java 类文件和使用的库的 jar 包，分别放在 WEB-INF/classes 和 WEB-INF/lib 目录下面。

- 多个应用共享的 Java 类文件和 jar 包，分别放在 Web 容器指定的由所有 Web 应用共享的目录下面。
- 当出现找不到类的错误时，检查当前类的类加载器和当前线程的上下文类加载器是否正确。

在介绍完类加载器与 Web 容器的关系之后，下面介绍它与 OSGi 的关系。

类加载器与 OSGi

OSGi 是 Java 上的动态模块系统。它为开发人员提供了面向服务和基于组件的运行环境，并提供标准的方式用来管理软件的生命周期。OSGi 已经被实现和部署在很多产品上，在开源社区也得到了广泛的支持。Eclipse 就是基于 OSGi 技术来构建的。

OSGi 中的每个模块（bundle）都包含 Java 包和类。模块可以声明它所依赖的需要导入（import）的其它模块的 Java 包和类（通过 Import-Package），也可以声明导出（export）自己的包和类，供其它模块使用（通过 Export-Package）。也就是说需要能够隐藏和共享一个模块中的某些 Java 包和类。这是通过 OSGi 特有的类加载器机制来实现的***。OSGi 中的每个模块都有对应的一个类加载器。它负责加载模块自己包含的 Java 包和类。当它需要加载 Java* **核心库的类时（以 java 开头的包和类），它会代理给父类加载器（通常是启动类加载器）来完成。当它需要加载所导入的 Java 类时，它会代理给导出此** **Java 类的模块来完成加载。模块也可以显式的声明某些 Java 包和类，必须由父类加载器来加载。只需要设置系统属性** **org.osgi.framework.bootdelegation 的值即可。****

假设有两个模块 bundleA 和 bundleB，它们都有自己对应的类加载器 classLoaderA 和 classLoaderB。在 bundleA 中包含类 com.bundleA.Sample，并且该类被声明为导出的，也就是说可以被其它模块所使用的。bundleB 声明了导入 bundleA 提供的类 com.bundleA.Sample，并包含一个类 com.bundleB.NewSample 继承自 com.bundleA.Sample。在 bundleB 启动的时候，其类加载器 classLoaderB 需要加载类 com.bundleB.NewSample，进而需要加载类 com.bundleA.Sample。由于 bundleB 声明了类 com.bundleA.Sample 是导入的，classLoaderB 把加载类 com.bundleA.Sample 的工作代理给导出该类的 bundleA 的类加载器 classLoaderA。classLoaderA 在其模块内部查找类 com.bundleA.Sample 并定义它，所得到的类 com.bundleA.Sample 实例就可以被所有声明导入了此类的模块使用。对于以 java 开头的类，都是由父类加载器来加载的。如果声明了系统属性 org.osgi.framework.bootdelegation=com.example.core.*，那么对于包 com.example.core 中的类，都是由父类加载器来完成的。

OSGi 模块的这种类加载器结构，使得一个类的不同版本可以共存在 Java 虚拟机中，带来了很大的灵活性。不过它的这种不同，也会给开发人员带来一些麻烦，尤其当模块需要使用第三方提供的库的时候。下面提供几条比较好的建议：

- 如果一个类库只有一个模块使用，把该类库的 jar 包放在模块中，在 Bundle-ClassPath 中指明即可。
- 如果一个类库被多个模块共用，可以为这个类库单独的创建一个模块，把其它模块需要用到的 Java 包声明为导出的。其它模块声明导入这些类。

如果类库提供了 SPI 接口，并且利用线程上下文类加载器来加载 SPI 实现的 Java 类，有可能会找不到 Java 类。如果出现了 NoClassDefFoundError 异常，首先检查当前线程的上下文类加载器是否正确。通过 Thread.currentThread().getContextClassLoader() 就可以得到该类加载器。该类加载器应该是该模块对应的类加载器。如果不是的话，可以首先通过 class.getClassLoader() 来得到模块对应的类加载器，再通过 Thread.currentThread().setContextClassLoader() 来设置当前线程的上下文类加载器。

总结

类加载器是 Java 语言的一个创新。它使得动态安装和更新软件组件成为可能。本文详细介绍了类加载器的相关话题，包括基本概念、代理模式、线程上下文类加载器、与 Web 容器和 OSGi 的关系等。开发人员在遇到 ClassNotFoundException 和 NoClassDefFoundError 等异常的时候，应该检查抛出异常的类的类加载器和当前线程的上下文类加载器，从中可以发现问题的所在。在开发自己的类加载器的时候，需要注意与已有的类加载器组织结构的协调。

jvm 如何分配直接内存， new 对象如何不分配在堆而是栈上，常量池解析

JVM直接内存

概述

直接内存并不是虚拟机运行时数据区的一部分，也不是Java 虚拟机规范中农定义的内存区域。在JDK1.4中新加入了NIO(New Input/Output)类，引入了一种基于通道(Channel)与缓冲区（Buffer）的I/O 方式，它可以使用native 函数库直接分配堆外内存，然后通脱一个存储在Java堆中的DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。

- 本机直接内存的分配不会受到Java 堆大小的限制，受到本机总内存大小限制
- 配置虚拟机参数时，不要忽略直接内存 防止出现OutOfMemoryError异常

直接内存（堆外内存）与堆内存比较

1. 直接内存申请空间耗费更高的性能，当频繁申请到一定量时尤为明显
2. 直接内存IO读写的性能要优于普通的堆内存，在多次读写操作的情况下差异明显

代码验证：

```
package com.xnccs.cn.share;

import java.nio.ByteBuffer;


/**
 * 直接内存 与  堆内存的比较
 */
public class ByteBufferCompare {


    public static void main(String[] args) {
        allocateCompare();   //分配比较
        operateCompare();    //读写比较
    }

    /**
     * 直接内存 和 堆内存的 分配空间比较
     *
     * 结论： 在数据量提升时，直接内存相比非直接内的申请，有很严重的性能问题
     *
     */
    public static void allocateCompare(){
        int time = 10000000;    //操作次数


        long st = System.currentTimeMillis();
        for (int i = 0; i < time; i++) {

            //ByteBuffer.allocate(int capacity)    分配一个新的字节缓冲区。
```

```
            ByteBuffer buffer = ByteBuffer.allocate(2);      //非直接内存分配申请

        }
        long et = System.currentTimeMillis();

        System.out.println("在进行"+time+"次分配操作时，堆内存 分配耗时:" + (et-st)
+"ms" );

        long st_heap = System.currentTimeMillis();
        for (int i = 0; i < time; i++) {
            //ByteBuffer.allocateDirect(int capacity) 分配新的直接字节缓冲区。
            ByteBuffer buffer = ByteBuffer.allocateDirect(2); //直接内存分配申请
        }
        long et_direct = System.currentTimeMillis();

        System.out.println("在进行"+time+"次分配操作时，直接内存 分配耗时:" +
(et_direct-st_heap) +"ms" );

    }

    /**
     * 直接内存 和 堆内存的 读写性能比较
     *
     * 结论：直接内存在直接的IO 操作上，在频繁的读写时 会有显著的性能提升
     *
     */
    public static void operateCompare(){
        int time = 1000000000;

        ByteBuffer buffer = ByteBuffer.allocate(2*time);
        long st = System.currentTimeMillis();
        for (int i = 0; i < time; i++) {

            //  putChar(char value) 用来写入 char 值的相对 put 方法
            buffer.putChar('a');
        }
        buffer.flip();
        for (int i = 0; i < time; i++) {
            buffer.getChar();
        }
        long et = System.currentTimeMillis();

        System.out.println("在进行"+time+"次读写操作时，非直接内存读写耗时：" + (et-st)
+"ms");

        ByteBuffer buffer_d = ByteBuffer.allocateDirect(2*time);
        long st_direct = System.currentTimeMillis();
        for (int i = 0; i < time; i++) {

            //  putChar(char value) 用来写入 char 值的相对 put 方法
            buffer_d.putChar('a');
        }
        buffer_d.flip();
        for (int i = 0; i < time; i++) {
            buffer_d.getChar();
        }
        long et_direct = System.currentTimeMillis();
```

```
        System.out.println("在进行"+time+"次读写操作时，直接内存读写耗时:" + (et_direct
- st_direct) +"ms");
    }
}
```

输出：
在进行10000000次分配操作时，堆内存 分配耗时:12ms
在进行10000000次分配操作时，直接内存 分配耗时:8233ms
在进行1000000000次读写操作时，非直接内存读写耗时：4055ms
在进行1000000000次读写操作时，直接内存读写耗时:745ms

可以自己设置不同的time 值进行比较

分析

从数据流的角度，来看

非直接内存作用链:
本地IO –>直接内存–>非直接内存–>直接内存–>本地IO
直接内存作用链:
本地IO–>直接内存–>本地IO

直接内存使用场景

- 有很大的数据需要存储，它的生命周期很长
- 适合频繁的IO操作，例如网络并发场景

浅谈HotSpot逃逸分析

JIT

即时编译（Just-in-time Compilation，JIT）是一种通过在运行时将字节码翻译为机器码，从而改善字节码编译语言性能的技术。在HotSpot实现中有多种选择：C1、C2和C1+C2，分别对应client、server和分层编译。
1、C1编译速度快，优化方式比较保守；
2、C2编译速度慢，优化方式比较激进；
3、C1+C2在开始阶段采用C1编译，当代码运行到一定热度之后采用G2重新编译；
在1.8之前，分层编译默认是关闭的，可以添加`-server -XX:+TieredCompilation`参数进行开启。

逃逸分析

逃逸分析并不是直接的优化手段，而是一个代码分析，通过动态分析对象的作用域，为其它优化手段如栈上分配、标量替换和同步消除等提供依据，发生逃逸行为的情况有两种：方法逃逸和线程逃逸。
1、方法逃逸：当一个对象在方法中定义之后，作为参数传递到其它方法中；
2、线程逃逸：如类变量或实例变量，可能被其它线程访问到；

如果不存在逃逸行为，则可以对该对象进行如下优化：同步消除、标量替换和栈上分配。

同步消除

线程同步本身比较耗，如果确定一个对象不会逃逸出线程，无法被其它线程访问到，那该对象的读写就不会存在竞争，则可以消除对该对象的同步锁，通过`-XX:+EliminateLocks`可以开启同步消除。

标量替换

1、标量是指不可分割的量，如java中基本数据类型和reference类型，相对的一个数据可以继续分解，称为聚合量；
2、如果把一个对象拆散，将其成员变量恢复到基本类型来访问就叫做标量替换；
3、如果逃逸分析发现一个对象不会被外部访问，并且该对象可以被拆散，那么经过优化之后，并不直接生成该对象，而是在栈上创建若干个成员变量；
通过`-XX:+EliminateAllocations`可以开启标量替换，`-XX:+PrintEliminateAllocations`查看标量替换情况。

栈上分配

故名思议就是在栈上分配对象，其实目前Hotspot并没有实现真正意义上的栈上分配，实际上是标量替换。

```
  private static int fn(int age) {
        User user = new User(age);
        int i = user.getAge();
        return i;
    }
```

User对象的作用域局限在方法fn中，可以使用标量替换的优化手段在栈上分配对象的成员变量，这样就不会生成User对象，大大减轻GC的压力，下面通过例子看看逃逸分析的影响。

```
public class JVM {
    public static void main(String[] args) throws Exception {
        int sum = 0;
        int count = 1000000;
        //warm up
        for (int i = 0; i < count ; i++) {
            sum += fn(i);
        }

        Thread.sleep(500);

        for (int i = 0; i < count ; i++) {
            sum += fn(i);
        }

        System.out.println(sum);
        System.in.read();
    }

    private static int fn(int age) {
        User user = new User(age);
        int i = user.getAge();
        return i;
    }
}

class User {
    private final int age;
```

```
    public User(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }
}
```

分层编译和逃逸分析在1.8中是默认是开启的，例子中fn方法被执行了200w次，按理说应该在Java堆生成200w个User对象。

1、通过 `java -cp . -Xmx3G -Xmn2G -server -XX:-DoEscapeAnalysis JVM` 运行代码，`-XX:-DoEscapeAnalysis` 关闭逃逸分析，通过 `jps` 查看java进程的PID，接着通过 `jmap -histo [pid]` 查看java堆上的对象分布情况，结果如下：

```
D:\Users\j_zhan.CN1\Desktop\JAVA
λ jjps
42064 Jps
11700
38340 Launcher
38564 Launcher
25048 RemoteMavenServer
40888 AppMain
42668 JVM

D:\Users\j_zhan.CN1\Desktop\JAVA
λ jmap -histo 42668

 num     #instances         #bytes  class name
----------------------------------------------
   1:           413       63768632  [I
   2:       2000000       32000000  User
   3:          3178         578056  [C
   4:          2251          54024  java.lang.String
   5:           463          49216  java.lang.Class
   6:           102          39088  [B
   7:           786          31440  java.util.TreeMap$Entry
   8:           531          30896  [Ljava.lang.Object;
   9:           209           9272  [Ljava.lang.String;
  10:           122           8784  java.lang.reflect.Field
  11:            48           4224  java.lang.reflect.Method
```

可以发现：关闭逃逸分析之后，User对象一个不少的都在堆上进行分配。

2、通过 `java -cp . -Xmx3G -Xmn2G -server JVM` 运行代码，结果如下：

```
D:\Users\j_zhan.CN1\Desktop\JAVA
λ jps
42992 Jps
11700
38340 Launcher
38564 Launcher
42676 JVM
25048 RemoteMavenServer
40888 AppMain

D:\Users\j_zhan.CN1\Desktop\JAVA
λ jmap -histo 42676

 num     #instances         #bytes  class name
----------------------------------------------
   1:           413       89177392  [I
   2:        411950        6591200  User
   3:          3179         578080  [C
   4:          2252          54048  java.lang.String
   5:           463          49216  java.lang.Class
   6:           102          39088  [B
   7:           786          31440  java.util.TreeMap$Entry
   8:           531          30896  [Ljava.lang.Object;
   9:           209           9272  [Ljava.lang.String;
  10:           122           8784  java.lang.reflect.Field
  11:            48           4224  java.lang.reflect.Method
  12:             4           4160  [[C
```

可以发现：开启逃逸分析之后，只有41w左右的User对象在Java堆上分配，其余的对象已经通过标量替换优化了。

3、通过`java -cp . -Xmx3G -Xmn2G -server -XX:-TieredCompilation`运行代码，关闭分层编译，结果如下：

```
D:\Users\j_zhan.CN1\Desktop\JAVA
λ jps
41968 Jps
11700
38340 Launcher
38564 Launcher
25048 RemoteMavenServer
40888 AppMain
42456 JVM

D:\Users\j_zhan.CN1\Desktop\JAVA
λ jmap -histo 42456

 num     #instances         #bytes  class name
----------------------------------------------
   1:           412       63346000  [I
   2:          3179         578136  [C
   3:         13167         210672  User
   4:          2253          54072  java.lang.String
   5:           463          49216  java.lang.Class
   6:           102          39088  [B
   7:           786          31440  java.util.TreeMap$Entry
   8:           531          30896  [Ljava.lang.Object;
   9:           209           9272  [Ljava.lang.String;
  10:           122           8784  java.lang.reflect.Field
  11:            48           4224  java.lang.reflect.Method
```

可以发现：关闭了分层编译之后，在Java堆上分配的User对象降低到1w多个，分层编译对逃逸分析还是有影响的。

编译阈值

即时编译JIT只在代码段执行足够次数才会进行优化，在执行过程中不断收集各种数据，作为优化的决策，所以在优化完成之前，例子中的User对象还是在堆上进行分配。

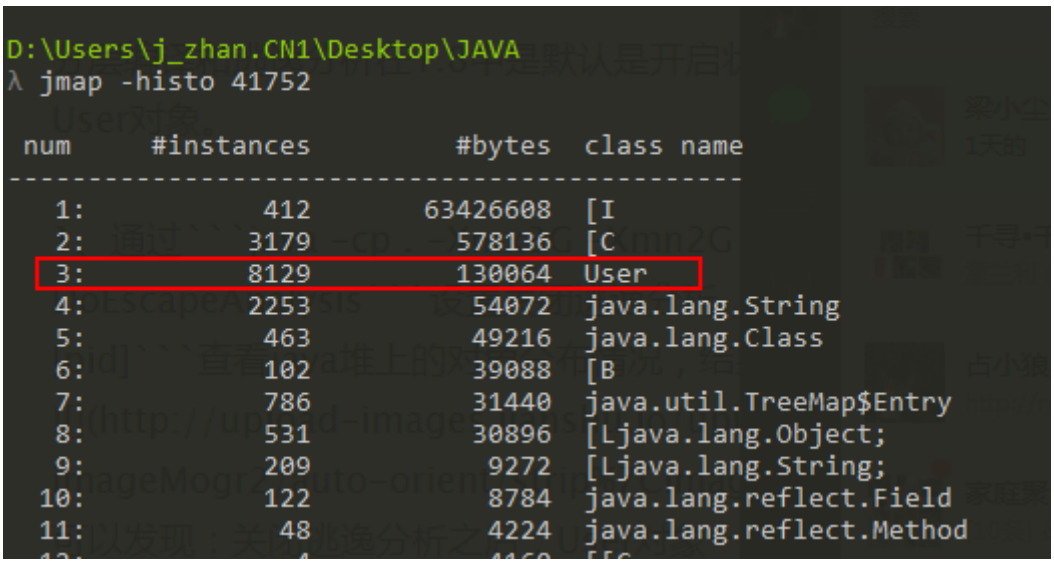
那么一段代码需要执行多少次才会触发JIT优化呢？通常这个值由`-XX:CompileThreshold`参数进行设置：
1、使用client编译器时，默认为1500；
2、使用server编译器时，默认为10000；
意味着如果方法调用次数或循环次数达到这个阈值就会触发标准编译，更改CompileThreshold标志的值，将使编译器提早（或延迟）编译。

除了标准编译，还有一个叫做OSR（On Stack Replacement）栈上替换的编译，如上述例子中的main方法，只执行一次，远远达不到阈值，但是方法体中执行了多次循环，OSR编译就是只编译该循环代码，然后将其替换，下次循环时就执行编译好的代码，不过触发OSR编译也需要一个阈值，可以通过以下公式得到。

```
-XX:CompileThreshold = 10000
-XX:OnStackReplacePercentage = 140
-XX:InterpreterProfilePercentage = 33
OSR trigger = (CompileThreshold * (OnStackReplacePercentage -
InterpreterProfilePercentage)) / 100 = 10700
```

其中trigger即为OSR编译的阈值。

那么如果把CompileThreshold设置适当小一点，是不是可以提早触发编译行为，减少在堆上生成User对象？我们可以进行通过不同参数验证一下：
1、`-XX:CompileThreshold = 5000`，结果如下：



2、`-XX:CompileThreshold = 2500`，结果如下：

```
D:\Users\j_zhan.CN1\Desktop\JAVA
λ jmap -histo 41864

 num     #instances         #bytes  class name
----------------------------------------------
   1:           412       63468624  [I
   2:          3179         578136  [C
   3:          5503          88048  User
   4:          2253          54072  java.lang.String
   5:           463          49216  java.lang.Class
   6:           102          39088  [B
   7:           786          31440  java.util.TreeMap$Entry
   8:           531          30896  [Ljava.lang.Object;
   9:           209           9272  [Ljava.lang.String;
  10:           122           8784  java.lang.reflect.Field
```

3、`-XX:CompileThreshold = 2000`，结果如下：

```
D:\Users\j_zhan.CN1\Desktop\JAVA
λ jmap -histo 42080

 num     #instances         #bytes  class name
----------------------------------------------
   1:           412       63483088  [I
   2:          3179         578136  [C
   3:          4599          73584  User
   4:          2253          54072  java.lang.String
   5:           463          49216  java.lang.Class
   6:           102          39088  [B
   7:           786          31440  java.util.TreeMap$Entry
   8:           531          30896  [Ljava.lang.Object;
   9:           209           9272  [Ljava.lang.String;
  10:           122           8784  java.lang.reflect.Field
```

4、`-XX:CompileThreshold = 1500`，结果如下：

```
D:\Users\j_zhan.CN1\Desktop\JAVA
λ jmap -histo 40188

 num     #instances         #bytes  class name
----------------------------------------------
   1:           412       62867912  [I
   2:         43002         688032  User
   3:          3190         578600  [C
   4:          2264          54336  java.lang.String
   5:           463          49216  java.lang.Class
   6:           102          39088  [B
   7:           786          31440  java.util.TreeMap$Entry
   8:           531          30896  [Ljava.lang.Object;
   9:           209           9272  [Ljava.lang.String;
  10:           122           8784  java.lang.reflect.Field
  11:            48           4224  java.lang.reflect.Method
  12:             4           4160  [[C
```

在我的机器中，当设置到1500时，在堆上生成的User对象反而升到4w个，目前还不清楚原因是啥...

JIT编译在默认情况是异步进行的，当触发某方法或某代码块的优化时，先将其放入编译队列，然后由编译线程进行编译，编译之后的代码放在CodeCache中，CodeCache的大小也是有限的，通过`-XX:-BackgroundCompilation`参数可以关闭异步编译，我们可以通过执行`java -cp . -Xmx3G -Xmn2G -server -XX:CompileThreshold=1 -XX:-TieredCompilation -XX:-BackgroundCompilation JVM`命令看看同步编译的效果：在java堆上只生成了2个对象。

```
138:             1             40  sun.nio.cs.ext.ExtendedCharsets
139:             1             32  [Ljava.lang.OutOfMemoryError;
140:             2             32  [Ljava.lang.StackTraceElement;
141:             1             32  [Ljava.lang.ThreadGroup;
142:             2             32  User
143:             2             32  java.io.FilePermission$1
144:             1             32  java.io.WinNTFileSystem
145:             1             32  java.lang.ArithmeticException
146:             2             32  java.lang.Boolean
147:             1             32  java.lang.NullPointerException
148:             1             32  java.lang.StringCoding$StringDecoder
149:             1             32  java.lang.StringCoding$StringEncoder
```

当然了，这是为了好玩而进行的测试，生产环境不要随意修改这些参数：
1、热点代码的编译过程是有成本的，如果逻辑复杂，编程成本更高；
2、编译后的代码会被存放在有大小限制的CodeCache中，如果CompileThreshold设置的太低，JIT会将一大堆执行不那么频繁的代码进行编译，并放入CodeCache，导致之后真正执行频繁的代码没有足够的空间存放；

触摸java常量池

java常量池是一个经久不衰的话题，也是面试官的最爱，题目花样百出，这次好好总结一下。

理论

小菜先拙劣的表达一下jvm虚拟内存分布：



程序计数器是jvm执行程序的流水线，存放一些跳转指令，这个太高深，小菜不懂。

本地方法栈是jvm调用操作系统方法所使用的栈。

虚拟机栈是jvm执行java代码所使用的栈。

方法区存放了一些常量、静态变量、类信息等，可以理解成class文件在内存中的存放位置。

虚拟机堆是jvm执行java代码所使用的堆。

Java中的常量池，实际上分为两种形态：**静态常量池**和**运行时常量池**。

所谓**静态常量池**，即*.class文件中的常量池，class文件中的常量池不仅仅包含字符串(数字)字面量，还包含类、方法的信息，占用class文件绝大部分空间。

而**运行时常量池**，则是jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在**方法区**中，我们常说的常量池，就是指方法区中的运行时常量池。

接下来我们引用一些网络上流行的常量池例子，然后借以讲解。

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Hel" + "lo";
String s4 = "Hel" + new String("lo");
String s5 = new String("Hello");
String s6 = s5.intern();
String s7 = "H";
String s8 = "ello";
String s9 = s7 + s8;

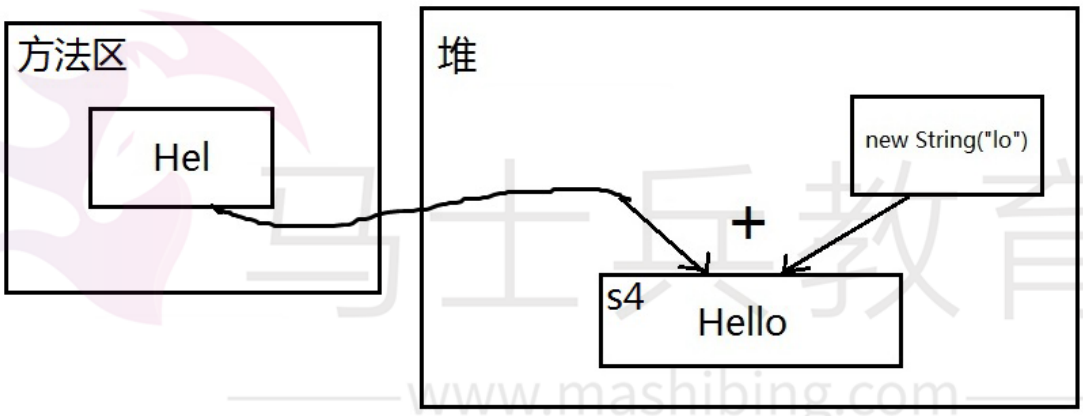
System.out.println(s1 == s2);  // true
System.out.println(s1 == s3);  // true
System.out.println(s1 == s4);  // false
System.out.println(s1 == s9);  // false
System.out.println(s4 == s5);  // false
System.out.println(s1 == s6);  // true
```

首先说明一点，在java 中，直接使用==操作符，比较的是两个字符串的引用地址，并不是比较内容，比较内容请用String.equals()。

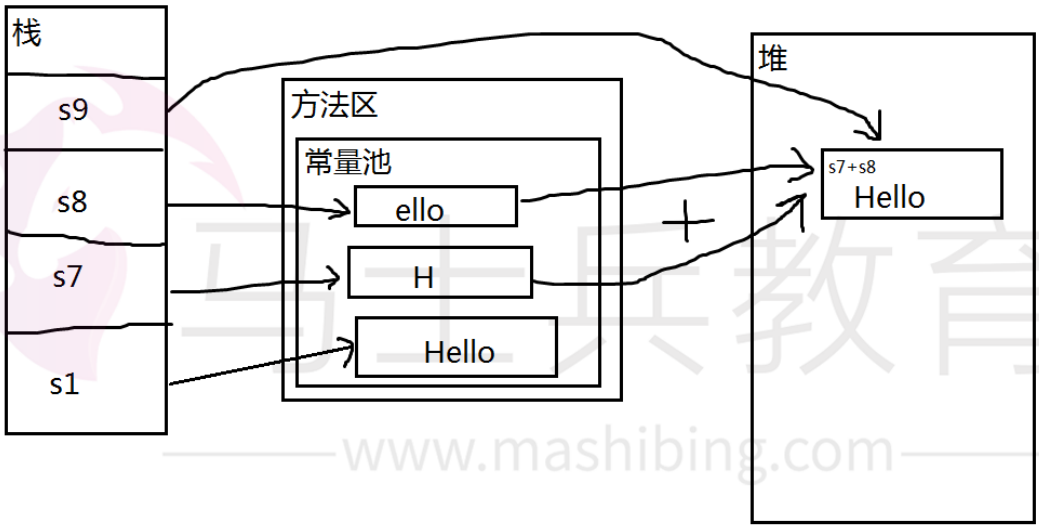
s1 == s2这个非常好理解，s1、s2在赋值时，均使用的字符串字面量，说白话点，就是直接把字符串写死，在编译期间，这种字面量会直接放入class文件的常量池中，从而实现复用，载入运行时常量池后，s1、s2指向的是同一个内存地址，所以相等。

s1 == s3这个地方有个坑，s3虽然是动态拼接出来的字符串，但是所有参与拼接的部分都是已知的字面量，在编译期间，这种拼接会被优化，编译器直接帮你拼好，因此String s3 = "Hel" + "lo";在class文件中被优化成String s3 = "Hello";，所以s1 == s3成立。

s1 == s4当然不相等，s4虽然也是拼接出来的，但new String("lo")这部分不是已知字面量，是一个不可预料的部分，编译器不会优化，必须等到运行时才可以确定结果，结合**字符串不变**定理，鬼知道s4被分配到哪去了，所以地址肯定不同。配上一张简图理清思路：



s1 == s9也不相等，道理差不多，虽然s7、s8在赋值的时候使用的字符串字面量，但是拼接成s9的时候，s7、s8作为两个变量，都是不可预料的，编译器毕竟是编译器，不可能当解释器用，所以不做优化，等到运行时，s7、s8拼接成的新字符串，在堆中地址不确定，不可能与方法区常量池中的s1地址相同。



s4 == s5已经不用解释了，绝对不相等，二者都在堆中，但地址不同。

s1 == s6这两个相等完全归功于intern方法，s5在堆中，内容为Hello ，intern方法会尝试将Hello字符串添加到常量池中，并返回其在常量池中的地址，因为常量池中已经有了Hello字符串，所以intern方法直接返回地址；而s1在编译期就已经指向常量池了，因此s1和s6指向同一地址，相等。

至此，我们可以得出三个非常重要的结论：

> **必须要关注编译期的行为，才能更好的理解常量池。**
>
> **运行时常量池中的常量，基本来源于各个class文件中的常量池。**
>
> **程序运行时，除非手动向常量池中添加常量(比如调用intern方法)，否则jvm不会自动添加常量到常量池。**

以上所讲仅涉及字符串常量池，实际上还有整型常量池、浮点型常量池等等，但都大同小异，只不过数值类型的常量池不可以手动添加常量，程序启动时常量池中的常量就已经确定了，比如整型常量池中的常量范围：-128~127，只有这个范围的数字可以用到常量池。

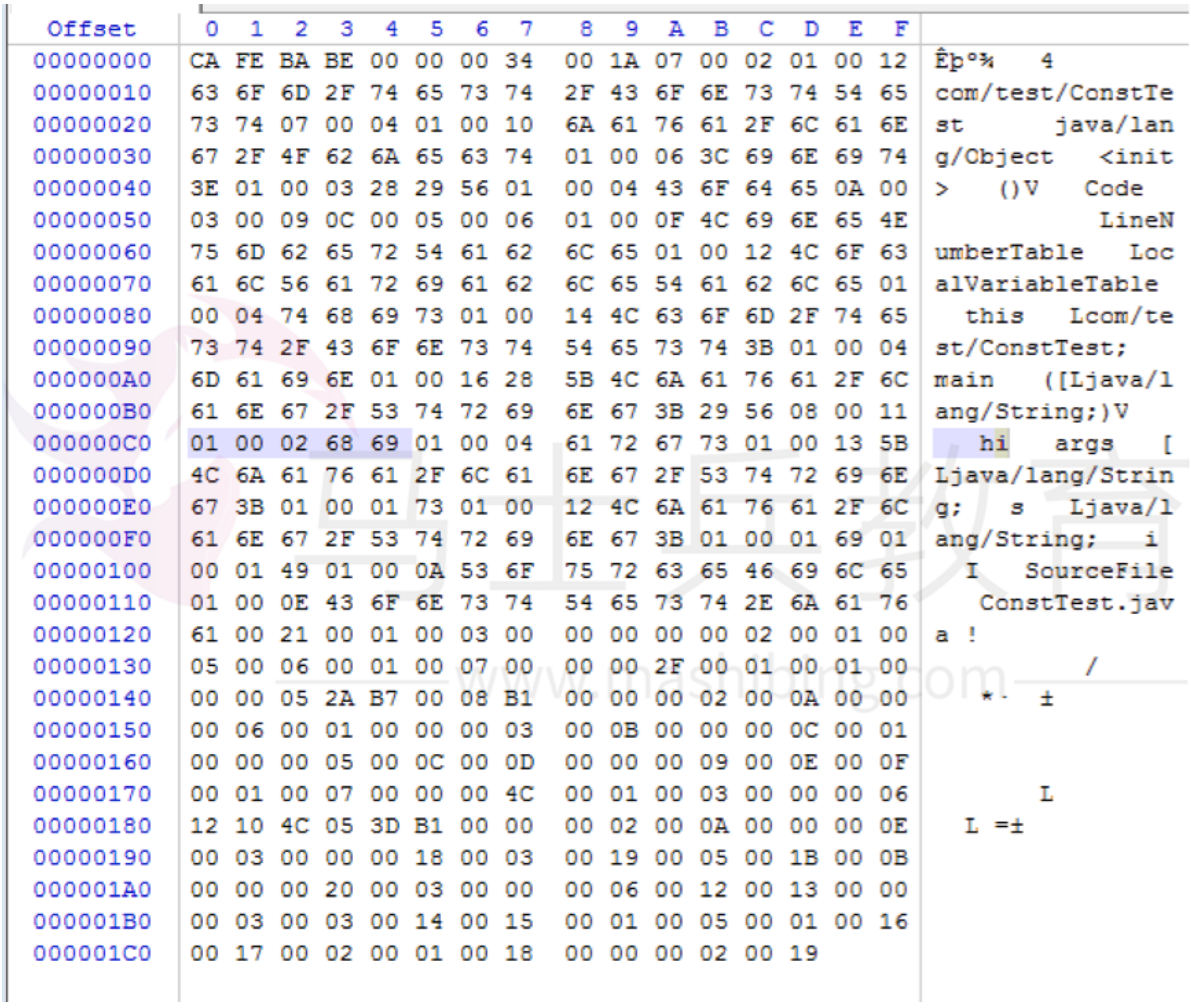
实践

说了这么多理论，接下来让我们触摸一下真正的常量池。

前文提到过，class文件中存在一个静态常量池，这个常量池是由编译器生成的，用来存储java源文件中的字面量(本文仅仅关注字面量)，假设我们有如下java代码：

```
1 String s = "hi";
```

为了方便起见，就这么简单，没错！将代码编译成class文件后，用winhex打开二进制格式的class文件。如图：



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	CA	FE	BA	BE	00	00	00	34	00	1A	07	00	02	01	00	12	Êþº¾ 4
00000010	63	6F	6D	2F	74	65	73	74	2F	43	6F	6E	73	74	54	65	com/test/ConstTe
00000020	73	74	07	00	04	01	00	10	6A	61	76	61	2F	6C	61	6E	st java/lan
00000030	67	2F	4F	62	6A	65	63	74	01	00	06	3C	69	6E	69	74	g/Object <init
00000040	3E	01	00	03	28	29	56	01	00	04	43	6F	64	65	0A	00	> ()V Code
00000050	03	00	09	0C	00	05	00	06	01	00	0F	4C	69	6E	65	4E	LineN
00000060	75	6D	62	65	72	54	61	62	6C	65	01	00	12	4C	6F	63	umberTable Loc
00000070	61	6C	56	61	72	69	61	62	6C	65	54	61	62	6C	65	01	alVariableTable
00000080	00	04	74	68	69	73	01	00	14	4C	63	6F	6D	2F	74	65	this Lcom/te
00000090	73	74	2F	43	6F	6E	73	74	54	65	73	74	3B	01	00	04	st/ConstTest;
000000A0	6D	61	69	6E	01	00	16	28	5B	4C	6A	61	76	61	2F	6C	main ([Ljava/l
000000B0	61	6E	67	2F	53	74	72	69	6E	67	3B	29	56	08	00	11	ang/String;)V
000000C0	01	00	02	68	69	01	00	04	61	72	67	73	01	00	13	5B	hi args [
000000D0	4C	6A	61	76	61	2F	6C	61	6E	67	2F	53	74	72	69	6E	Ljava/lang/Strin
000000E0	67	3B	01	00	01	73	01	00	12	4C	6A	61	76	61	2F	6C	g; s Ljava/l
000000F0	61	6E	67	2F	53	74	72	69	6E	67	3B	01	00	01	69	01	ang/String; i
00000100	00	01	49	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	I SourceFile
00000110	01	00	0E	43	6F	6E	73	74	54	65	73	74	2E	6A	61	76	ConstTest.jav
00000120	61	00	21	00	01	00	03	00	00	00	00	00	02	00	01	00	a !
00000130	05	00	06	00	01	00	07	00	00	00	2F	00	01	00	01	00	/
00000140	00	00	05	2A	B7	00	08	B1	00	00	00	02	00	0A	00	00	*· ±
00000150	00	06	00	01	00	00	00	03	00	0B	00	00	00	0C	00	01	
00000160	00	00	00	05	00	0C	00	0D	00	00	00	09	00	0E	00	0F	
00000170	00	01	00	07	00	00	00	4C	00	01	00	03	00	00	00	06	L
00000180	12	10	4C	05	3D	B1	00	00	00	02	00	0A	00	00	00	0E	L =±
00000190	00	03	00	00	00	18	00	03	00	19	00	05	00	1B	00	0B	
000001A0	00	00	00	20	00	03	00	00	00	06	00	12	00	13	00	00	
000001B0	00	03	00	03	00	14	00	15	00	01	00	05	00	01	00	16	
000001C0	00	17	00	02	00	01	00	18	00	00	00	02	00	19			

简单讲解一下class文件的结构，开头的4个字节是class文件魔数，用来标识这是一个class文件，说白话点就是文件头，既：CA FE BA BE。

紧接着4个字节是java的版本号，这里的版本号是34，因为笔者是用jdk8编译的，版本号的高低和jdk版本的高低相对应，高版本可以兼容低版本，但低版本无法执行高版本。所以，如果哪天读者想知道别人的class文件是用什么jdk版本编译的，就可以看这4个字节。

接下来就是常量池入口，入口处用2个字节标识常量池常量数量，本例中数值为00 1A，翻译成十进制是26，也就是有25个常量，其中第0个常量是特殊值，所以只有25个常量。

常量池中存放了各种类型的常量，他们都有自己的类型，并且都有自己的存储规范，本文只关注字符串常量，字符串常量以01开头(1个字节)，接着用2个字节记录字符串长度，然后就是字符串实际内容。本例中为：01 00 02 68 69。

接下来再说说运行时常量池，由于运行时常量池在方法区中，我们可以通过jvm参数：-XX:PermSize、-XX:MaxPermSize来设置方法区大小，从而间接限制常量池大小。

假设jvm启动参数为：-XX:PermSize = 2M -XX:MaxPermSize = 2M，然后运行如下代码：

```
1 //保持引用，防止自动垃圾回收
2 List<String> list = new ArrayList<String>();
3
4 int i = 0;
5
6 while(true){
7     //通过intern方法向常量池中手动添加常量
8     list.add(String.valueOf(i++).intern());
9 }
```

程序立刻会抛出：Exception in thread "main" java.lang.outOfMemoryError: PermGen space异常。PermGen space正是方法区，足以说明常量池在方法区中。

在jdk8中，移除了方法区，转而用Metaspace区域替代，所以我们需要使用新的jvm参数：-XX:MaxMetaspaceSize=2M，依然运行如上代码，抛出：java.lang.OutOfMemoryError: Metaspace异常。同理说明运行时常量池是划分在Metaspace区域中。具体关于Metaspace区域的知识，请读者自行搜索。

本文所有代码均在jdk7、jdk8下测试通过，其他版本jdk可能会略有差异，请读者自行探索。

数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold，问通常多大，没做过一问便知）

> 注意:PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效，Parallel Scavenge收集器不认识这个参数，Parallel Scavenge收集器一般并不需要设置。
> 如果遇到必须使用此参数的场合，可以考虑ParNew加CMS的收集器组合。

深入理解JVM : Java对象内存分配策略

Java技术体系中所提倡的自动内存管理最终可以归结为自动化地解决了两个问题：给对象分配内存以及回收分配给对象的内存。
对象的内存分配，往大方向讲，就是在堆上分配，对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

本文中的内存分配策略指的是Serial / Serial Old收集器下（ParNew / Serial Old收集器组合的规则也基本一致）的内存分配和回收的策略。

1. 对象优先在Eden分配

大多数情况下，对象在**新生代Eden区**中分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC。

- **代码示例：**

```
private static final int _1MB = 1024 * 1024;
/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -
XX:SurvivorRatio=8
  */
public static void testAllocation() {
     byte[] allocation1, allocation2, allocation3, allocation4;
     allocation1 = new byte[2 * _1MB];
     allocation2 = new byte[2 * _1MB];
     allocation3 = new byte[2 * _1MB];
     allocation4 = new byte[4 * _1MB];  // 出现一次Minor GC
 }
```

运行结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0070106 secs] 6651K->6292K(19456K), 0.0070426
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4326K [0x029d0000, 0x033d0000,
0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,  14% used [0x032d0000, 0x032f5370, 0x033d0000)
  to   space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
 tenured generation   total 10240K, used 6144K [0x033d0000, 0x03dd0000,
0x03dd0000)
   the space 10240K,  60% used [0x033d0000, 0x039d0030, 0x039d0200, 0x03dd0000)

 compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)

No shared spaces configured.
```

代码的`testAllocation()`方法中，尝试分配3个2MB大小和1个4MB大小的对象，在运行时通过-Xms20M、 -Xmx20M、 -Xmn10M这3个参数限制了Java堆大小为20MB，不可扩展，其中10MB分配给新生代，剩下的10MB分配给老年代。`-XX:SurvivorRatio=8`决定了新生代中Eden区与一个Survivor区的空间比例是8：1，从输出的结果也可以清晰地看到`eden space 8192K、from space 1024K、to space 1024K`的信息，新生代总可用空间为9216KB（Eden区+1个Survivor区的总容量）。

执行`testAllocation()`中分配allocation4对象的语句时会发生一次Minor GC，这次GC的结果是新生代6651KB变为148KB，而总内存占用量则几乎没有减少（因为allocation1、allocation2、allocation3三个对象都是存活的，虚拟机几乎没有找到可回收的对象）。这次GC发生的原因是给allocation4分配内存的时候，发现Eden已经被占用了6MB，剩余空间已不足以分配allocation4所需的4MB内存，因此发生Minor GC。GC期间虚拟机又发现已有的3个2MB大小的对象全部无法放入Survivor空间（Survivor空间只有1MB大小），所以只好通过分配担保机制提前转移到老年代去。

这次GC结束后，4MB的allocation4对象顺利分配在Eden中，因此程序执行完的结果是Eden占用4MB（被allocation4占用），Survivor空闲，老年代被占用6MB（被allocation1、allocation2、allocation3占用）。通过GC日志可以证实这一点。

2. 大对象直接进入老年代

所谓的大对象是指，需要**大量连续内存空间**的Java对象，最典型的大对象就是那种很长的字符串以及数组。

大对象对虚拟机的内存分配来说就是一个坏消息（比遇到一个大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，写程序的时候应当避免），经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

虚拟机提供了一个`-XX:PretenureSizeThreshold`参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制（复习一下：新生代采用复制算法收集内存）。

- **代码示例：**

```
private static final int _1MB = 1024 * 1024;
/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -
XX:SurvivorRatio=8
 * -XX:PretenureSizeThreshold=3145728
 */
public static void testPretenureSizeThreshold() {
  byte[] allocation;
  allocation = new byte[4 * _1MB];  //直接分配在老年代中
}
```

运行结果：

```
1.Heap
2.def new generation   total 9216K, used 671K [0x029d0000, 0x033d0000,
0x033d0000)
3.eden space 8192K,   8% used [0x029d0000, 0x02a77e98, 0x031d0000)
4.from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
5.to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
6.tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000,
0x03dd0000)
7.the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
8.compacting perm gen  total 12288K, used 2107K [0x03dd0000, 0x049d0000,
0x07dd0000)
9.the space 12288K,  17% used [0x03dd0000, 0x03fdefd0, 0x03fdf000, 0x049d0000)
10.No shared spaces configured.
```

执行代码中的`testPretenureSizeThreshold()`方法后，我们看到Eden空间几乎没有被使用，而老年代的10MB空间被使用了40%，也就是4MB的allocation对象直接就分配在老年代中，这是因为`PretenureSizeThreshold`被设置为3MB（就是3145728，这个参数不能像-Xmx之类的参数一样直接写3MB），因此超过3MB的对象都会直接在老年代进行分配。

注意:`PretenureSizeThreshold`参数只对Serial和ParNew两款收集器有效，Parallel Scavenge收集器不认识这个参数，Parallel Scavenge收集器一般并不需要设置。
如果遇到必须使用此参数的场合，可以考虑ParNew加CMS的收集器组合。

3. 长期存活的对象将进入老年代

为了在内存回收时能识别哪些对象应放在新生代，哪些对象应放在老年代中。虚拟机给每个对象定义了一个对象年龄（Age）计数器。

- **对象年龄的判定：**
  如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1。
  对象在Survivor区中每“熬过”一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁），就将会被晋升到老年代中。
  对象晋升老年代的年龄阈值，可以通过参数`-XX:MaxTenuringThreshold`设置。
- **代码示例**：

```
private static final int _1MB = 1024 * 1024;
/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -
XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=1
 * -XX:+PrintTenuringDistribution
 */
@SuppressWarnings("unused")
public static void testTenuringThreshold() {
  byte[] allocation1, allocation2, allocation3;
  allocation1 = new byte[_1MB / 4];
   // 什么时候进入老年代取决于XX:MaxTenuringThreshold设置
  allocation2 = new byte[4 * _1MB];
  allocation3 = new byte[4 * _1MB];
  allocation3 = null;
  allocation3 = new byte[4 * _1MB];
}
```

以MaxTenuringThreshold=1参数来运行的结果：

```
1.[GC [DefNew
2.Desired Survivor size 524288 bytes, new threshold 1 (max 1)
3.- age   1:     414664 bytes,     414664 total
4.: 4859K->404K(9216K), 0.0065012 secs] 4859K->4500K(19456K), 0.0065283 secs]
[Times: user=0.02 sys=0.00, real=0.02 secs]
5.[GC [DefNew
6.Desired Survivor size 524288 bytes, new threshold 1 (max 1)
7.: 4500K->0K(9216K), 0.0009253 secs] 8596K->4500K(19456K), 0.0009458 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
8.Heap
9.def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000,
0x033d0000)
10.eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
11.from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
12.to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
13.to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
14.to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
15.to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
16.tenured generation   total 10240K, used 4500K [0x033d0000, 0x03dd0000,
0x03dd0000)
17.the space 10240K,  43% used [0x033d0000, 0x03835348, 0x03835400, 0x03dd0000)

18.compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
```

```
19.the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)

20.No shared spaces configured.
```

以MaxTenuringThreshold=15参数来运行的结果：

```
1.[GC [DefNew
2.Desired Survivor size 524288 bytes, new threshold 15 (max 15)
3.- age   1:     414664 bytes,     414664 total
4.: 4859K->404K(9216K), 0.0049637 secs] 4859K->4500K(19456K), 0.0049932 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
5.[GC [DefNew
6.Desired Survivor size 524288 bytes, new threshold 15 (max 15)
7.- age   2:     414520 bytes,     414520 total
8.: 4500K->404K(9216K), 0.0008091 secs] 8596K->4500K(19456K), 0.0008305 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
9.Heap
10.def new generation   total 9216K, used 4582K [0x029d0000, 0x033d0000,
0x033d0000)
11.eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
12.from space 1024K,  39% used [0x031d0000, 0x03235338, 0x032d0000)
13.to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
14.tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000,
0x03dd0000)
15.the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)

16.compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
17.the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)

18.No shared spaces configured.
```

分别以`-XX:MaxTenuringThreshold=1`和`-XX:MaxTenuringThreshold=15`两种设置来执行代码清单3-7中的`testTenuringThreshold()`方法，此方法中的allocation1对象需要256KB内存，Survivor空间可以容纳。当MaxTenuringThreshold=1时，allocation1对象在第二次GC发生时进入老年代，新生代已使用的内存GC后非常干净地变成0KB。而MaxTenuringThreshold=15时，第二次GC发生后，allocation1对象则还留在新生代Survivor空间，这时新生代仍然有404KB被占用。

4. 动态对象年龄判定

为了能更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代.

- **动态对象年龄判定**
  如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到`MaxTenuringThreshold`中要求的年龄。
- **代码示例**

```
private static final int _1MB = 1024 * 1024;
/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -
XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=15
```

```
 * -XX:+PrintTenuringDistribution
 */
@SuppressWarnings("unused")
public static void testTenuringThreshold2() {
  byte[] allocation1, allocation2, allocation3, allocation4;
  allocation1 = new byte[_1MB / 4];
    // allocation1+allocation2大于survivo空间一半
  allocation2 = new byte[_1MB / 4];
  allocation3 = new byte[4 * _1MB];
  allocation4 = new byte[4 * _1MB];
  allocation4 = null;
  allocation4 = new byte[4 * _1MB];
}
```

```
1.[GC [DefNew
2.Desired Survivor size 524288 bytes, new threshold 1 (max 15)
3.- age   1:     676824 bytes,     676824 total
4.: 5115K->660K(9216K), 0.0050136 secs] 5115K->4756K(19456K), 0.0050443 secs]
[Times: user=0.00 sys=0.01, real=0.01 secs]
5.[GC [DefNew
6.Desired Survivor size 524288 bytes, new threshold 15 (max 15)
7.: 4756K->0K(9216K), 0.0010571 secs] 8852K->4756K(19456K), 0.0011009 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
8.Heap
9.def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000,
0x033d0000)
10.eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
11.from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
12.to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
13.tenured generation   total 10240K, used 4756K [0x033d0000, 0x03dd0000,
0x03dd0000)
14.the space 10240K,  46% used [0x033d0000, 0x038753e8, 0x03875400, 0x03dd0000)

15.compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
16.the space 12288K,  17% used [0x03dd0000, 0x03fe09a0, 0x03fe0a00, 0x049d0000)

17.No shared spaces configured.
```

执行代码中的`testTenuringThreshold2()`方法，并设置`-XX:MaxTenuringThreshold=15`，会发现运行结果中Survivor的空间占用仍然为0%，而老年代比预期增加了6%，也就是说，allocation1、allocation2对象都直接进入了老年代，而没有等到15岁的临界年龄。因为这两个对象加起来已经到达了512KB，并且它们是同年的，满足同年对象达到Survivor空间的一半规则。我们只要注释掉其中一个对象new操作，就会发现另外一个就不会晋升到老年代中去了。

5. 空间分配担保

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

下面解释一下“冒险”是冒了什么风险，前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代。与生活中的贷款担保类似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多少对象会活下来在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行Full GC来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说，如果某次Minor GC存活后的对象突增，远远高于平均值的话，依然会导致担保失败（Handle Promotion Failure）。如果出现了HandlePromotionFailure失败，那就只好在失败后重新发起一次Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将HandlePromotionFailure开关打开，避免Full GC过于频繁.

老年代中数组的访问方式

GC算法，永久代对象如何GC，GC有环怎么处理

永久代GC的原因：

- 永久代空间已经满了
- 调用了System.gc()

  注意： 这种GC是full GC 堆空间也会一并被GC一次

GC有环怎么处理

1. 根搜索算法

   什么是根搜索算法

   垃圾回收器从被称为GC Roots的点开始遍历遍历对象，凡是可以达到的点都会标记为存活，堆中不可到达的对象都会标记成垃圾，然后被清理掉。

   GC Roots有哪些

   - 类，由系统类加载器加载的类。这些类从不会被卸载，它们可以通过静态属性的方式持有对象的引用。

   > 注意，一般情况下由自定义的类加载器加载的类不能成为GC Roots

   - 线程，存活的线程
   - Java方法栈中的局部变量或者参数
   - JNI方法栈中的局部变量或者参数
   - JNI全局引用
   - 用做同步监控的对象

     被JVM持有的对象，这些对象由于特殊的目的不被GC回收。这些对象可能是系统的类加载器，一些重要的异常处理类，一些为处理异常预留的对象，以及一些正在执行类加载的自定义的类加载器。但是具体有哪些前面提到的对象依赖于具体的JVM实现。
2. 如何处理

   基于引用对象遍历的垃圾回收器可以处理循环引用，只要是涉及到的对象不能从GC Roots强引用可到达，垃圾回收器都会进行清理来释放内存。

谁会被GC，什么时候GC

Java JVM：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）

在什么时候：

首先需要知道，GC又分为minor GC 和 Full GC（major GC）。Java堆内存分为新生代和老年代，新生代

中又分为1个eden区和两个Survior区域。

一般情况下，新创建的对象都会被分配到eden区，这些对象经过一个minor gc后仍然存活将会被移动到

Survior区域中，对象在Survior中没熬过一个Minor GC，年龄就会增加一岁，当他的年龄到达一定程度时，

就会被移动到老年代中。

当eden区满时，还存活的对象将被复制到survior区，当一个survior区满时，此区域的存活对象将被复制到另外一个

survior区，当另外一个也满了的时候，从前一个Survior区复制过来的并且此时还存活的对象，将可能被复制到老年代

因为年轻代中的对象基本都是朝生夕死（80%以上），所以年轻代的垃圾回收算法使用的是复制算法，

复制算法的基本思想是将内存分为两块，每次只有其中一块，当这一块内存使用完，就将还活着的对象复制到

另一块上面。复制算法不会产生内存碎片。

在GC开始的时候，对象只会存在于eden区，和名为“From”的Survior区，Survior区“to”是空的。紧接着GC

eden区中所有存活的对象都会被复制到“To”,而在from区中，仍存活的对象会根据他们的年龄值来决定去向，

年龄到达一定只的对象会被复制到老年代,没有到达的对象会被复制到to survior中，经过这次gc后，eden区和from

survior区已经被清空。这个时候，from和to会交换他们的角色，也就是新的to就是上次GC前的from

Minor GC：从年轻代回收内存

当jvm无法为一个新的对象分配空间时会触发Minor GC，比如当Eden区满了。

当内存池被填满的时候，其中的内容全部会被复制，指针会从0开始跟踪空闲内存。Eden和Survior区不存在内存碎片

写指针总是停留在所使用内存池的顶部。执行minor操作时不会影响到永久代，从永久带到年轻代的引用被当成

GC roots，从年轻代到永久代的引用在标记阶段被直接忽略掉（永久代用来存放java的类信息）。如果eden区域中大部分

对象被认为是垃圾，永远也不会复制到Survior区域或者老年代空间。如果正好相反，eden区域大部分新生对象不符合GC

条件，Minor GC执行时暂停的线程时间将会长很多。Minor may call "stop the world";

Full GC：是清理整个堆空间包括年轻代和老年代。

那么对于Minor GC的触发条件：大多数情况下，直接在eden区中进行分配。如果eden区域没有足够的空间，

那么就会发起一次Minor GC;对于FullGC的触发条件：如果老年代没有足够的空间，那么就会进行一次FullGC

在发生MinorGC之前，虚拟机会先检查老年代最大可利用的连续空间是否大于新生代所有对象的总空间。

如果大于则进行Minor GC，如果小于则看HandlePromotionFailure设置是否是允许担保失败（不允许则直接FullGC）

如果允许，那么会继续检查老年代最大可利用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于

则尝试minor gc （如果尝试失败也会触发Full GC），如果小于则进行Full GC。

但是，具体什么时候执行，这个是由系统来进行决定的，是无法预测的。

对什么东西：

主要根据可达性分析算法，如果一个对象不可达,那么就是可以回收的，如果一个对象可达，那么这个对象就不可以回收，

对于可达性分析算法，它是通过一系列称为“GC Roots”的对象最为起始点，当一个对象GC Roots没有任何引用链相接的时候，

那么这个对象就是不可达，就可以被回收。

做了什么事情：

主要做了清理对象，整理内存的工作。Java堆分为新生代和老年代，采用了不同的回收方式。

例如新生代采用了复制算法，老年代采用了标记整理法。在新生代中，分为一个ede区域和两个Survior

区域，真正使用的是一个eden区域，和一个Survior区域，GC的时候，会把存活的对象放入到另一个Survior区域中，

然后再把这个eden区域和Survior区域清除。那么对于老年代，采用的是标记整理发，首先标记出存活对象，

然后在移动到一段。这样有利于减少内存碎片。

标记：标记的过程其实就是，遍历所有gc root 然后将所有gc root 可达的对象标记为存活对象

清除：清除的过程中将遍历堆中所有的对象，将没有标记的对象全部清除掉

主要缺点：标记和清除过程效率不高，标记清除之后会产生大量不连续的内存碎片

但是，老年代中因为对象存活率高，没有额外空间对他进行分配担保，就必须使用标记整理算法

标记整理算法 标记操作和"标记-清除"算法一致，后续操作不只是直接清理对象，而是在清理

无用对象完成后让所有存活的对象都向一段移动，并更新其引用对象的指针

主要缺点：在标记清除的基础上还需要进行对象的移动，成本相对比较高，成本相对较高，好处是不会产生内存碎片。

如果想在 GC 中生存 1 次怎么办

生存一次，释放掉对象的引用，但是在对象的finalize方法中重新建立引用，但是此方法只会被调用一次，所以能在GC中生存一次。

分析System.gc()方法

JVM源码分析之SystemGC完全解读

概述

JVM的GC一般情况下是JVM本身根据一定的条件触发的，不过我们还是可以做一些人为的触发，比如通过jvmti做强制GC，通过System.gc触发，还可以通过jmap来触发等，针对每个场景其实我们都可以写篇文章来做一个介绍，本文重点介绍下System.gc的原理

或许大家已经知道如下相关的知识

- system.gc其实是做一次full gc
- system.gc会暂停整个进程
- system.gc一般情况下我们要禁掉，使用-XX:+DisableExplicitGC
- system.gc在cms gc下我们通过-XX:+ExplicitGCInvokesConcurrent来做一次稍微高效点的GC(效果比Full GC要好些)
- system.gc最常见的场景是RMI/NIO下的堆外内存分配等

如果你已经知道上面这些了其实也说明你对System.gc有过一定的了解，至少踩过一些坑，但是你是否更深层次地了解过它，比如

- 为什么CMS GC下-XX:+ExplicitGCInvokesConcurrent这个参数加了之后会比真正的Full GC好?
- 它如何做到暂停整个进程?
- 堆外内存分配为什么有时候要配合System.gc?

如果你上面这些疑惑也都知道，那说明你很懂System.gc了，那么接下来的文字你可以不用看啦

JDK里的System.gc的实现

先贴段代码吧（java.lang.System）

```
    /**
     * Runs the garbage collector.
     * <p>
     * Calling the <code>gc</code> method suggests that the Java Virtual
     * Machine expend effort toward recycling unused objects in order to
```

```
     * make the memory they currently occupy available for quick reuse.
     * When control returns from the method call, the Java Virtual
     * Machine has made a best effort to reclaim space from all discarded
     * objects.
     * <p>
     * The call <code>System.gc()</code> is effectively equivalent to the
     * call:
     * <blockquote><pre>
     * Runtime.getRuntime().gc()
     * </pre></blockquote>
     *
     * @see     java.lang.Runtime#gc()
     */
    public static void gc() {
        Runtime.getRuntime().gc();
    }
```

发现主要调用的是Runtime里的gc方法（java.lang.Runtime）

```
    /**
     * Runs the garbage collector.
     * Calling this method suggests that the Java virtual machine expend
     * effort toward recycling unused objects in order to make the memory
     * they currently occupy available for quick reuse. When control
     * returns from the method call, the virtual machine has made
     * its best effort to recycle all discarded objects.
     * <p>
     * The name <code>gc</code> stands for "garbage
     * collector". The virtual machine performs this recycling
     * process automatically as needed, in a separate thread, even if the
     * <code>gc</code> method is not invoked explicitly.
     * <p>
     * The method {@link System#gc()} is the conventional and convenient
     * means of invoking this method.
     */
    public native void gc();
```

这里看到gc方法是native的，在java层面只能到此结束了，代码只有这么多，要了解更多，可以看方法上面的注释，不过我们需要更深层次地来了解其实现，那还是准备好进入到jvm里去看看

Hotspot里System.gc的实现

如何找到native里的实现

上面提到了Runtime.gc是一个本地方法，那需要先在jvm里找到对应的实现，这里稍微提一下jvm里native方法最常见的也是最简单的查找，jdk里一般含有native方法的类，一般都会有一个对应的c文件，比如上面的java.lang.Runtime这个类，会有一个Runtime.c的文件和它对应，native方法的具体实现都在里面了，如果你有source，可能会猜到和下面的方法对应

```
JNIEXPORT void JNICALL
Java_java_lang_Runtime_gc(JNIEnv *env, jobject this)
{
    JVM_GC();
}
```

其实没错的，就是这个方法，jvm要查找到这个native方法其实很简单的，看方法名可能也猜到规则了，Java_pkgName_className_methodName，其中pkgName里的"."替换成"_"，这样就能找到了，当然规则不仅仅只有这么一个，还有其他的，这里不细说了，有机会写篇文章详细介绍下其中细节

DisableExplicitGC参数

上面的方法里是调用JVM_GC()，实现如下

```
JVM_ENTRY_NO_ENV(void, JVM_GC(void))
  JVMWrapper("JVM_GC");
  if (!DisableExplicitGC) {
    Universe::heap()->collect(GCCause::_java_lang_system_gc);
  }
JVM_END
```

看到这里我们已经解释其中一个疑惑了，就是`DisableExplicitGC`这个参数是在哪里生效的，起的什么作用，如果这个参数设置为true的话，那么将直接跳过下面的逻辑，我们通过-XX:+ DisableExplicitGC就是将这个属性设置为true，而这个属性默认情况下是true还是false呢

```
product(bool, DisableExplicitGC, false,
          "Tells whether calling System.gc() does a full GC")
```

ExplicitGCInvokesConcurrent参数

这里主要针对CMSGC下来做分析，所以我们上面看到调用了heap的collect方法，我们找到对应的逻辑

```
void GenCollectedHeap::collect(GCCause::Cause cause) {
  if (should_do_concurrent_full_gc(cause)) {
#ifndef SERIALGC
    // mostly concurrent full collection
    collect_mostly_concurrent(cause);
#else  // SERIALGC
    ShouldNotReachHere();
#endif // SERIALGC
  } else {
#ifdef ASSERT
    if (cause == GCCause::_scavenge_alot) {
      // minor collection only
      collect(cause, 0);
    } else {
      // Stop-the-world full collection
      collect(cause, n_gens() - 1);
    }
#else
```

```
    // Stop-the-world full collection
    collect(cause, n_gens() - 1);
#endif
  }
}

bool GenCollectedHeap::should_do_concurrent_full_gc(GCCause::Cause cause) {
  return UseConcMarkSweepGC &&
         ((cause == GCCause::_gc_locker && GCLockerInvokesConcurrent) ||
          (cause == GCCause::_java_lang_system_gc &&
ExplicitGCInvokesConcurrent));
}
```

collect里一开头就有个判断，如果should_do_concurrent_full_gc返回true，那会执行collect_mostly_concurrent做并行的回收

其中should_do_concurrent_full_gc中的逻辑是如果使用CMS GC，并且是system gc且ExplicitGCInvokesConcurrent==true，那就做并行full gc，当我们设置-XX:+ExplicitGCInvokesConcurrent的时候，就意味着应该做并行Full GC了，不过要注意千万不要设置-XX:+DisableExplicitGC，不然走不到这个逻辑里来了

并行Full GC相对正常的Full GC效率高在哪里

stop the world

说到GC，这里要先提到VMThread，在jvm里有这么一个线程不断轮询它的队列，这个队列里主要是存一些VM_operation的动作，比如最常见的就是内存分配失败要求做GC操作的请求等，在对gc这些操作执行的时候会先将其他业务线程都进入到安全点，也就是这些线程从此不再执行任何字节码指令，只有当出了安全点的时候才让他们继续执行原来的指令，因此这其实就是我们说的stop the world(STW)，整个进程相当于静止了

CMS GC

这里必须提到CMS GC，因为这是解释并行Full GC和正常Full GC的关键所在，CMS GC我们分为两种模式background和foreground，其中background顾名思义是在后台做的，也就是可以不影响正常的业务线程跑，触发条件比如说old的内存占比超过多少的时候就可能触发一次background式的cms gc，这个过程会经历CMS GC的所有阶段，该暂停的暂停，该并行的并行，效率相对来说还比较高，毕竟有和业务线程并行的gc阶段；而foreground则不然，它发生的场景比如业务线程请求分配内存，但是内存不够了，于是可能触发一次cms gc，这个过程就必须是要等内存分配到了线程才能继续往下面走的，因此整个过程必须是STW的，因此CMS GC整个过程都是暂停应用的，但是为了提高效率，它并不是每个阶段都会走的，只走其中一些阶段，这些省下来的阶段主要是并行阶段，Precleaning、AbortablePreclean，Resizing这几个阶段都不会经历，其中sweep阶段是同步的，但不管怎么说如果走了类似foreground的cms gc，那么整个过程业务线程都是不可用的，效率会影响挺大。CMS GC具体的过程后面再写文章详细说，其过程确实非常复杂的

正常的Full GC

正常的Full GC其实是整个gc过程包括ygc和cms gc(这里说的是真正意义上的Full GC，还有些场景虽然调用Full GC的接口，但是并不会都做，有些时候只做ygc，有些时候只做cms gc)都是由VMThread来执行的，因此整个时间是ygc+cms gc的时间之和，其中CMS GC是上面提到的foreground式的，因此整个过程会比较长，也是我们要避免的

并行的Full GC

并行Full GC也通样会做YGC和CMS GC，但是效率高就搞在CMS GC是走的background的，整个暂停的过程主要是YGC+CMS_initMark+CMS_remark几个阶段

堆外内存常配合使用System GC

这里说的堆外内存主要针对java.nio.DirectByteBuffer，这些对象的创建过程会通过Unsafe接口直接通过os::malloc来分配内存，然后将内存的起始地址和大小存到java.nio.DirectByteBuffer对象里，这样就可以直接操作这些内存。这些内存只有在DirectByteBuffer回收掉之后才有机会被回收，因此如果这些对象大部分都移到了old，但是一直没有触发CMS GC或者Full GC，那么悲剧将会发生，因为你的物理内存被他们耗尽了，因此为了避免这种悲剧的发生，通过-XX:MaxDirectMemorySize来指定最大的堆外内存大小，当使用达到了阈值的时候将调用System.gc来做一次full gc，以此来回收掉没有被使用的堆外内存，具体堆外内存是如何回收的，其原理机制又是怎样的，还是后面写篇详细的文章吧

JVM 选项 -XX:+UseCompressedOops 有什么作用？为什么要使用?

当你将你的应用从 32 位的 JVM 迁移到 64 位的 JVM 时，由于对象的指针从 32 位增加到了 64 位，因此堆内存会突然增加，差不多要翻倍。这也会对 CPU 缓存（容量比内存小很多）的数据产生不利的影响。因为，迁移到 64 位的 JVM 主要动机在于可以指定最大堆大小，通过压缩 OOP 可以节省一定的内存。通过 -XX:+UseCompressedOops 选项，JVM 会使用 32 位的 OOP，而不是 64 位的 OOP。

写代码分别使得JVM的堆、栈和持久代发生内存溢出(栈溢出)

JVM内存溢出详解（栈溢出，堆溢出，持久代溢出以及无法创建本地线程)

写在前面

内存溢出和内存泄漏的区别：

内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。

内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

memory leak会最终会导致out of memory

内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

内存泄漏是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。一个盘子用尽各种方法只能装4个果子，你装了5个，结果掉倒地上不能吃了。这就是溢出！比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。就

是分配的内存不足以放下数据项序列,称为内存溢出.

全文简短总结，具体内容可以看下文。

栈内存溢出(StackOverflowError)：

程序所要求的栈深度过大导致，可以写一个死递归程序触发。

堆内存溢出(OutOfMemoryError:java heap space)

- 分清内存溢出还是内存泄漏
- 泄露则看对象如何被 GC Root 引用。
- 溢出则通过 调大 -Xms，-Xmx参数。

持久带内存溢出(OutOfMemoryError: PermGen space)

- 持久带中包含方法区，方法区包含常量池
- 因此持久带溢出有可能是运行时常量池溢出，也有可能是方法区中保存的class对象没有被及时回收掉或者class信息占用的内存超过了我们配置
- 用String.intern()触发常量池溢出
- Class对象未被释放，Class对象占用信息过多，有过多的Class对象。可以导致持久带内存溢出

无法创建本地线程

总容量不变，堆内存，非堆内存设置过大，会导致能给线程的内存不足。

以下是详细内容

栈溢出(StackOverflowError)

栈溢出抛出StackOverflowError错误，**出现此种情况是因为方法运行的时候栈的深度超过了虚拟机容许的最大深度所致。**出现这种情况，一般情况下是程序错误所致的，比如写了一个死递归，就有可能造成此种情况。 下面我们通过一段代码来模拟一下此种情况的内存溢出。

```
import java.util.*;
import java.lang.*;
public class OOMTest{
    public void stackOverFlowMethod(){
        stackOverFlowMethod();
    }
    public static void main(String... args){
        OOMTest oom = new OOMTest();
        oom.stackOverFlowMethod();
    }
}
```

运行上面的代码，会抛出如下的异常：

```
Exception in thread "main" java.lang.StackOverflowError
        at OOMTest.stackOverFlowMethod(OOMTest.java:6)
```

对于栈内存溢出，根据《Java 虚拟机规范》中文版：

> 如果线程请求的栈容量超过栈允许的最大容量的话，Java 虚拟机将抛出一个StackOverflow异常；如果Java虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么Java虚拟机将抛出一个OutOfMemory 异常。

堆溢出(OutOfMemoryError:java heap space)

堆内存溢出的时候，虚拟机会抛出java.lang.OutOfMemoryError:java heap space,出现此种情况的时候，我们需要根据内存溢出的时候产生的dump文件来具体分析（需要增加-XX:+HeapDumpOnOutOfMemoryErrorjvm启动参数）。**出现此种问题的时候有可能是内存泄露，也有可能是内存溢出了。**

- 如果内存泄露，我们要找出泄露的对象是怎么被GC ROOT引用起来，然后通过引用链来具体分析泄露的原因。
- 如果出现了内存溢出问题，这往往是程序本生需要的内存大于了我们给虚拟机配置的内存，这种情况下，我们可以采用调大-Xmx来解决这种问题。下面我们通过如下的代码来演示一下此种情况的溢出：

```
import java.util.*;
import java.lang.*;
public class OOMTest{
        public static void main(String... args){
                List<byte[]> buffer = new ArrayList<byte[]>();
                buffer.add(new byte[10*1024*1024]);
        }

}
```

我们通过如下的命令运行上面的代码：

```
java -verbose:gc -Xmn10M -Xms20M -Xmx20M -XX:+PrintGC OOMTest
```

程序输出如下的信息：

```
[GC 1180K->366K(19456K), 0.0037311 secs]
[Full GC 366K->330K(19456K), 0.0098740 secs]
[Full GC 330K->292K(19456K), 0.0090244 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at OOMTest.main(OOMTest.java:7)
```

从运行结果可以看出，JVM进行了一次Minor gc和两次的Major gc，从Major gc的输出可以看出，gc以后old区使用率为134K，而字节数组为10M，加起来大于了old generation的空间，所以抛出了异常，如果调整-Xms21M,-Xmx21M,那么就不会触发gc操作也不会出现异常了。

通过上面的实验其实也从侧面验证了一个结论：**对象大于新生代剩余内存的时候，将直接放入老年代，当老年代剩余内存还是无法放下的时候，触发垃圾收集，收集后还是不能放下就会抛出内存溢出异常了。**

持久带溢出(OutOfMemoryError: PermGen space)

我们知道Hotspot jvm通过持久带实现了Java虚拟机规范中的方法区，而运行时的常量池就是保存在方法区中的，因此持久带溢出有可能是运行时常量池溢出，也有可能是方法区中保存的class对象没有被及时回收掉或者class信息占用的内存超过了我们配置。
当持久带溢出的时候抛出java.lang.OutOfMemoryError: PermGen space。可能在如下几种场景下出现：

1. 使用一些应用服务器的热部署的时候，我们就会遇到热部署几次以后发现内存溢出了，这种情况就是因为每次热部署的后，原来的class没有被卸载掉。
2. 如果应用程序本身比较大，涉及的类库比较多，但是我们分配给持久带的内存（通过-XX:PermSize和-XX:MaxPermSize来设置）比较小的时候也可能出现此种问题。
3. 一些第三方框架，比如spring,hibernate都通过字节码生成技术（比如CGLib）来实现一些增强的功能，这种情况可能需要更大的方法区来存储动态生成的Class文件。
   我们知道Java中字符串常量是放在常量池中的，String.intern()这个方法运行的时候，会检查常量池中是否存和本字符串相等的对象，如果存在直接返回对常量池中对象的引用，不存在的话，先把此字符串加入常量池，然后再返回字符串的引用。那么我们就可以通过String.intern方法来模拟一下运行时常量区的溢出.下面我们通过如下的代码来模拟此种情况：

```
import java.util.*;
import java.lang.*;
public class OOMTest{
        public static void main(String... args){
                List<String> list = new ArrayList<String>();
                while(true){
                        list.add(UUID.randomUUID().toString().intern());
                }
        }
}
```

我们通过如下的命令运行上面代码：

```
java -verbose:gc -Xmn5M -Xms10M -Xmx10M -XX:MaxPermSize=1M -XX:+PrintGC OOMTest
```

运行后的输入如下图所示:

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
        at java.lang.String.intern(Native Method)
        at OOMTest.main(OOMTest.java:8)
```

通过上面的代码，我们成功模拟了运行时常量池溢出的情况，从输出中的PermGen space可以看出确实是持久带发生了溢出，这也验证了，我们前面说的Hotspot jvm通过持久带来实现方法区的说法。

OutOfMemoryError:unable to create native thread

最后我们在来看看java.lang.OutOfMemoryError:unable to create natvie thread这种错误。 出现这种情况的时候，一般是下面两种情况导致的：

1. 程序创建的线程数超过了操作系统的限制。对于Linux系统，我们可以通过ulimit -u来查看此限制。
2. 给虚拟机分配的内存过大，导致创建线程的时候需要的native内存太少。

我们都知道操作系统对每个进程的内存是有限制的，我们启动jvm,相当于启动了一个进程，假如我们一个进程占用了4G的内存，那么通过下面的公式计算出来的剩余内存就是建立线程栈的时候可以用的内存。*线程栈总可用内存=4G-（-Xmx的值）-（-XX:MaxPermSize的值）-程序计数器占用的内存*
通过上面的公式我们可以看出，-Xmx 和 MaxPermSize的值越大，那么留给线程栈可用的空间就越小，在-Xss参数配置的栈容量不变的情况下，可以创建的线程数也就越小。因此如果是因为这种情况导致的unable to create native thread,那么要么我们增大进程所占用的总内存，或者减少-Xmx或者-Xss来达到创建更多线程的目的。

为什么jdk8用metaspace数据结构用来替代perm?

元空间（Metaspace）：

一种新的内存空间的诞生。JDK8 HotSpot JVM 使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）；这与Oracle JRockit 和IBM JVM's很相似。这将是一个好消息：意味着不会再有java.lang.OutOfMemoryError: PermGen问题，也不再需要你进行调优及监控内存空间的使用，但是新特性不能消除类和类加载器导致的内存泄漏。你需要使用不同的方法以及遵守新的命名约定来追踪这些问题。

PermGen 空间的状况

这部分内存空间将全部移除。

JVM的参数：PermSize 和 MaxPermSize 会被忽略并给出警告（如果在启用时设置了这两个参数）。

Metaspace 内存分配模型

（最大区别）大部分类元数据都在本地内存中分配。

用于描述类元数据的"klasses"已经被移除。

Metaspace 容量

默认情况下，类元数据只受可用的本地内存限制（容量取决于是32位或是64位操作系统的可用虚拟内存大小）。

新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。

Metaspace 垃圾回收

对于僵死的类及类加载器的垃圾回收将在元数据使用达到"MaxMetaspaceSize"参数的设定值时进行。

适时地监控和调整元空间对于减小垃圾回收频率和减少延时是很有必要的。持续的元空间垃圾回收说明，可能存在类、类加载器导致的内存泄漏或是大小设置不合适。

Java 堆内存的影响

一些杂项数据已经移到Java堆空间中。升级到JDK8之后，会发现Java堆 空间有所增长。

Metaspace 监控

元空间的使用情况可以从HotSpot1.8的详细GC日志输出中得到。

Jstat 和 JVisualVM两个工具，在我们使用b75版本进行测试时，已经更新了，但是还是能看到老的PermGen空间的出现。

参数设置

默认情况下，元空间的大小仅受本地内存限制，但可以通过以下参数来指定元空间的大小：

-XX:MetaspaceSize，初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。

-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

除了上面两个指定大小的选项以外，还有两个与 GC 相关的属性：
-XX:MinMetaspaceFreeRatio，在GC之后，最小的Metaspace剩余空间容量的百分比，减少为分配空间所导致的垃圾收集

-XX:MaxMetaspaceFreeRatio，在GC之后，最大的Metaspace剩余空间容量的百分比，减少为释放空间所导致的垃圾收集

更新原因

1、字符串存在永久代中，容易出现性能问题和内存溢出。

2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。

3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

4、Oracle 可能会将HotSpot 与 JRockit 合二为一。

简单谈谈堆外内存以及你的理解和认识

JVM源码分析之堆外内存完全解读

概述

广义的堆外内存

说到堆外内存，那大家肯定想到堆内内存，这也是我们大家接触最多的，我们在jvm参数里通常设置-Xmx来指定我们的堆的最大值，不过这还不是我们理解的Java堆，-Xmx的值是新生代和老生代的和的最大值，我们在jvm参数里通常还会加一个参数-XX:MaxPermSize来指定持久代的最大值，那么我们认识的Java堆的最大值其实是-Xmx和-XX:MaxPermSize的总和，在分代算法下，新生代，老生代和持久代是连续的虚拟地址，因为它们是一起分配的，那么剩下的都可以认为是堆外内存(广义的)了，这些包括了jvm本身在运行过程中分配的内存，codecache，jni里分配的内存，DirectByteBuffer分配的内存等等

狭义的堆外内存

而作为java开发者，我们常说的堆外内存溢出了，其实是狭义的堆外内存，这个主要是指java.nio.DirectByteBuffer在创建的时候分配内存，我们这篇文章里也主要是讲狭义的堆外内存，因为它和我们平时碰到的问题比较密切

JDK/JVM里DirectByteBuffer的实现

DirectByteBuffer通常用在通信过程中做缓冲池，在mina，netty等nio框架中屡见不鲜，先来看看JDK里的实现：

```
    DirectByteBuffer(int cap) {                   // package-private

        super(-1, 0, cap, cap);
        boolean pa = VM.isDirectMemoryPageAligned();
        int ps = Bits.pageSize();
        long size = Math.max(1L, (long)cap + (pa ? ps : 0));
        Bits.reserveMemory(size, cap);

        long base = 0;
        try {
            base = unsafe.allocateMemory(size);
        } catch (OutOfMemoryError x) {
            Bits.unreserveMemory(size, cap);
            throw x;
        }
        unsafe.setMemory(base, size, (byte) 0);
        if (pa && (base % ps != 0)) {
            // Round up to page boundary
            address = base + ps - (base & (ps - 1));
        } else {
            address = base;
        }
        cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
        att = null;



    }
```

通过上面的构造函数我们知道，真正的内存分配是使用的Bits.reserveMemory方法

```
    static void reserveMemory(long size, int cap) {
        synchronized (Bits.class) {
            if (!memoryLimitSet && VM.isBooted()) {
                maxMemory = VM.maxDirectMemory();
                memoryLimitSet = true;
            }
            // -XX:MaxDirectMemorySize limits the total capacity rather than the
            // actual memory usage, which will differ when buffers are page
            // aligned.
            if (cap <= maxMemory - totalCapacity) {
                reservedMemory += size;
                totalCapacity += cap;
                count++;
                return;
            }
        }

        System.gc();
        try {
            Thread.sleep(100);
```

```
        } catch (InterruptedException x) {
            // Restore interrupt status
            Thread.currentThread().interrupt();
        }
        synchronized (Bits.class) {
            if (totalCapacity + cap > maxMemory)
                throw new OutOfMemoryError("Direct buffer memory");
            reservedMemory += size;
            totalCapacity += cap;
            count++;
        }

    }
```

通过上面的代码我们知道可以通过-XX:MaxDirectMemorySize来指定最大的堆外内存，那么我们首先引入两个问题

- 堆外内存默认是多大
- 为什么要主动调用System.gc()

堆外内存默认是多大

如果我们没有通过-XX:MaxDirectMemorySize来指定最大的堆外内存，那么默认的最大堆外内存是多少呢，我们还是通过代码来分析

上面的代码里我们看到调用了sun.misc.VM.maxDirectMemory()

```
private static long directMemory = 64 * 1024 * 1024;

    // Returns the maximum amount of allocatable direct buffer memory.
    // The directMemory variable is initialized during system initialization
    // in the saveAndRemoveProperties method.
    //
    public static long maxDirectMemory() {
        return directMemory;
    }
```

看到上面的代码之后是不是误以为默认的最大值是64M？其实不是的，说到这个值得从java.lang.System这个类的初始化说起

```
/**
     * Initialize the system class.  Called after thread initialization.
     */
    private static void initializeSystemClass() {

        // VM might invoke JNU_NewStringPlatform() to set those encoding
        // sensitive properties (user.home, user.name, boot.class.path, etc.)
        // during "props" initialization, in which it may need access, via
        // System.getProperty(), to the related system encoding property that
        // have been initialized (put into "props") at early stage of the
        // initialization. So make sure the "props" is available at the
        // very beginning of the initialization and all system properties to
        // be put into it directly.
        props = new Properties();
        initProperties(props);  // initialized by the VM

        // There are certain system configurations that may be controlled by
```

```
        // VM options such as the maximum amount of direct memory and
        // Integer cache size used to support the object identity semantics
        // of autoboxing.  Typically, the library will obtain these values
        // from the properties set by the VM.  If the properties are for
        // internal implementation use only, these properties should be
        // removed from the system properties.
        //
        // See java.lang.Integer.IntegerCache and the
        // sun.misc.VM.saveAndRemoveProperties method for example.
        //
        // Save a private copy of the system properties object that
        // can only be accessed by the internal implementation.  Remove
        // certain system properties that are not intended for public access.
        sun.misc.VM.saveAndRemoveProperties(props);

         ......

        sun.misc.VM.booted();
    }
```

上面这个方法在jvm启动的时候对System这个类做初始化的时候执行的，因此执行时间非常早，我们看到里面调用了`sun.misc.VM.saveAndRemoveProperties(props)`：

```
    public static void saveAndRemoveProperties(Properties props) {
        if (booted)
            throw new IllegalStateException("System initialization has
completed");

        savedProps.putAll(props);

        // Set the maximum amount of direct memory.  This value is controlled
        // by the vm option -XX:MaxDirectMemorySize=<size>.
        // The maximum amount of allocatable direct buffer memory (in bytes)
        // from the system property sun.nio.MaxDirectMemorySize set by the VM.
        // The system property will be removed.
        String s = (String)props.remove("sun.nio.MaxDirectMemorySize");
        if (s != null) {
            if (s.equals("-1")) {
                // -XX:MaxDirectMemorySize not given, take default
                directMemory = Runtime.getRuntime().maxMemory();
            } else {
                long l = Long.parseLong(s);
                if (l > -1)
                    directMemory = l;
            }
        }

        // Check if direct buffers should be page aligned
        s = (String)props.remove("sun.nio.PageAlignDirectMemory");
        if ("true".equals(s))
            pageAlignDirectMemory = true;

        // Set a boolean to determine whether ClassLoader.loadClass accepts
        // array syntax.  This value is controlled by the system property
        // "sun.lang.ClassLoader.allowArraySyntax".
        s = props.getProperty("sun.lang.ClassLoader.allowArraySyntax");
        allowArraySyntax = (s == null
```

```
                                ? defaultAllowArraySyntax
                                : Boolean.parseBoolean(s));

        // Remove other private system properties
        // used by java.lang.Integer.IntegerCache
        props.remove("java.lang.Integer.IntegerCache.high");

        // used by java.util.zip.ZipFile
        props.remove("sun.zip.disableMemoryMapping");

        // used by sun.launcher.LauncherHelper
        props.remove("sun.java.launcher.diag");
    }
```

如果我们通过-Dsun.nio.MaxDirectMemorySize指定了这个属性，只要它不等于-1，那效果和加了-XX:MaxDirectMemorySize一样的，如果两个参数都没指定，那么最大堆外内存的值来自于`directMemory = Runtime.getRuntime().maxMemory()`，这是一个native方法

```
JNIEXPORT jlong JNICALL
Java_java_lang_Runtime_maxMemory(JNIEnv *env, jobject this)
{
    return JVM_MaxMemory();
}

JVM_ENTRY_NO_ENV(jlong, JVM_MaxMemory(void))
  JVMWrapper("JVM_MaxMemory");
  size_t n = Universe::heap()->max_capacity();
  return convert_size_t_to_jlong(n);
JVM_END
```

其中在我们使用CMS GC的情况下的实现如下，其实是新生代的最大值-一个survivor的大小+老生代的最大值，也就是我们设置的-Xmx的值里除去一个survivor的大小就是默认的堆外内存的大小了

```
size_t GenCollectedHeap::max_capacity() const {
  size_t res = 0;
  for (int i = 0; i < _n_gens; i++) {
    res += _gens[i]->max_capacity();
  }
  return res;
}

size_t DefNewGeneration::max_capacity() const {
  const size_t alignment = GenCollectedHeap::heap()->collector_policy()-
>min_alignment();
  const size_t reserved_bytes = reserved().byte_size();
  return reserved_bytes - compute_survivor_size(reserved_bytes, alignment);
}

size_t Generation::max_capacity() const {
  return reserved().byte_size();
}
```

为什么要主动调用System.gc

既然要调用System.gc，那肯定是想通过触发一次gc操作来回收堆外内存，不过我想先说的是堆外内存不会对gc造成什么影响(这里的System.gc除外)，但是堆外内存的回收其实依赖于我们的gc机制，首先我们要知道在java层面和我们在堆外分配的这块内存关联的只有与之关联的DirectByteBuffer对象了，它记录了这块内存的基地址以及大小，那么既然和gc也有关，那就是gc能通过操作DirectByteBuffer对象来间接操作对应的堆外内存了。DirectByteBuffer对象在创建的时候关联了一个PhantomReference，说到PhantomReference它其实主要是用来跟踪对象何时被回收的，它不能影响gc决策，但是gc过程中如果发现某个对象除了只有PhantomReference引用它之外，并没有其他的地方引用它了，那将会把这个引用放到java.lang.ref.Reference.pending队列里，在gc完毕的时候通知ReferenceHandler这个守护线程去执行一些后置处理，而DirectByteBuffer关联的PhantomReference是PhantomReference的一个子类，在最终的处理里会通过Unsafe的free接口来释放DirectByteBuffer对应的堆外内存块

JDK里ReferenceHandler的实现：

```
private static class ReferenceHandler extends Thread {

        ReferenceHandler(ThreadGroup g, String name) {
            super(g, name);
        }

        public void run() {
            for (;;) {

                Reference r;
                synchronized (lock) {
                    if (pending != null) {
                        r = pending;
                        Reference rn = r.next;
                        pending = (rn == r) ? null : rn;
                        r.next = r;
                    } else {
                        try {
                            lock.wait();
                        } catch (InterruptedException x) { }
                        continue;
                    }
                }

                // Fast path for cleaners
                if (r instanceof Cleaner) {
                    ((Cleaner)r).clean();
                    continue;
                }

                ReferenceQueue q = r.queue;
                if (q != ReferenceQueue.NULL) q.enqueue(r);
            }
        }
    }
```

可见如果pending为空的时候，会通过lock.wait()一直等在那里，其中唤醒的动作是在jvm里做的，当gc完成之后会调用如下的方法VM_GC_Operation::doit_epilogue()，在方法末尾会调用lock的notify操作，至于pending队列什么时候将引用放进去的，其实是在gc的引用处理逻辑中放进去的，针对引用的处理后面可以专门写篇文章来介绍

```
void VM_GC_Operation::doit_epilogue() {
  assert(Thread::current()->is_Java_thread(), "just checking");
  // Release the Heap_lock first.
  SharedHeap* sh = SharedHeap::heap();
  if (sh != NULL) sh->_thread_holds_heap_lock_for_gc = false;
  Heap_lock->unlock();
  release_and_notify_pending_list_lock();
}

void VM_GC_Operation::release_and_notify_pending_list_lock() {
instanceRefKlass::release_and_notify_pending_list_lock(&_pending_list_basic_lock
);
}
```

对于System.gc的实现，它会对新生代的老生代都会进行内存回收，这样会比较彻底地回收DirectByteBuffer对象以及他们关联的堆外内存，我们dump内存发现DirectByteBuffer对象本身其实是很小的，但是它后面可能关联了一个非常大的堆外内存，因此我们通常称之为『冰山对象』，我们做ygc的时候会将新生代里的不可达的DirectByteBuffer对象及其堆外内存回收了，但是无法对old里的DirectByteBuffer对象及其堆外内存进行回收，这也是我们通常碰到的最大的问题，如果有大量的DirectByteBuffer对象移到了old，但是又一直没有做cms gc或者full gc，而只进行ygc，那么我们的物理内存可能被慢慢耗光，但是我们还不知道发生了什么，因为heap明明剩余的内存还很多(前提是我们禁用了System.gc)。

为什么要使用堆外内存

DirectByteBuffer在创建的时候会通过Unsafe的native方法来直接使用malloc分配一块内存，这块内存是heap之外的，那么自然也不会对gc造成什么影响(System.gc除外)，因为gc耗时的操作主要是操作heap之内的对象，对这块内存的操作也是直接通过Unsafe的native方法来操作的，相当于DirectByteBuffer仅仅是一个壳，还有我们通信过程中如果数据是在Heap里的，最终也还是会copy一份到堆外，然后再进行发送，所以为什么不直接使用堆外内存呢。对于需要频繁操作的内存，并且仅仅是临时存在一会的，都建议使用堆外内存，并且做成缓冲池，不断循环利用这块内存。

为什么不能大面积使用堆外内存

如果我们大面积使用堆外内存并且没有限制，那迟早会导致内存溢出，毕竟程序是跑在一台资源受限的机器上，因为这块内存的回收不是你直接能控制的，当然你可以通过别的一些途径，比如反射，直接使用Unsafe接口等，但是这些务必给你带来了一些烦恼，Java与生俱来的优势被你完全抛弃了---开发不需要关注内存的回收，由gc算法自动去实现。另外上面的gc机制与堆外内存的关系也说了，如果一直触发不了cms gc或者full gc，那么后果可能很严重。

threadlocal使用场景及注意事项

想必很多朋友对ThreadLocal并不陌生，今天我们就来一起探讨下ThreadLocal的使用方法和实现原理。首先，本文先谈一下对ThreadLocal的理解，然后根据ThreadLocal类的源码分析了其实现原理和使用需要注意的地方，最后给出了两个应用场景。

一.对ThreadLocal的理解

ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

这句话从字面上看起来很容易理解，但是真正理解并不是那么容易。

我们还是先来看一个例子：

Java代码

```
1. **class** ConnectionManager {
2.
3. **private** **static** Connection connect = **null**;
4.
5. **public** **static** Connection openConnection() {
6. **if**(connect == **null**){
7. connect = DriverManager.getConnection();
8. }
9. **return** connect;
10. }
11.
12. **public** **static** **void** closeConnection() {
13. **if**(connect!=**null**)
14. connect.close();
15. }
16. }
```

假设有这样一个数据库链接管理类，这段代码在单线程中使用是没有任何问题的，但是如果在多线程中使用呢？很显然，在多线程中使用会存在线程安全问题：第一，这里面的2个方法都没有进行同步，很可能在openConnection方法中会多次创建connect；第二，由于connect是共享变量，那么必然在调用connect的地方需要使用到同步来保障线程安全，因为很可能一个线程在使用connect进行数据库操作，而另外一个线程调用closeConnection关闭链接。

所以出于线程安全的考虑，必须将这段代码的两个方法进行同步处理，并且在调用connect的地方需要进行同步处理。

这样将会大大影响程序执行效率，因为一个线程在使用connect进行数据库操作的时候，其他线程只有等待。

那么大家来仔细分析一下这个问题，这地方到底需不需要将connect变量进行共享？事实上，是不需要的。假如每个线程中都有一个connect变量，各个线程之间对connect变量的访问实际上是没有依赖关系的，即一个线程不需要关心其他线程是否对这个connect进行了修改的。

到这里，可能会有朋友想到，既然不需要在线程之间共享这个变量，可以直接这样处理，在每个需要使用数据库连接的方法中具体使用时才创建数据库链接，然后在方法调用完毕再释放这个连接。比如下面这样：

Java代码

```
1. **class** ConnectionManager {
2.
3. **private** Connection connect = **null**;
4.
5. **public** Connection openConnection() {
6. **if**(connect == **null**){
7. connect = DriverManager.getConnection();
```

```
8. }
9. **return** connect;
10. }
11.
12. **public** **void** closeConnection() {
13. **if**(connect!=**null**)
14. connect.close();
15. }
16. }
17.
18. **class** Dao{
19. **public** **void** insert() {
20. ConnectionManager connectionManager = **new** ConnectionManager();
21. Connection connection = connectionManager.openConnection();
22.
23. //使用connection进行操作
24.
25. connectionManager.closeConnection();
26. }
27. }
```

这样处理确实也没有任何问题，由于每次都是在方法内部创建的连接，那么线程之间自然不存在线程安全问题。但是这样会有一个致命的影响：导致服务器压力非常大，并且严重影响程序执行性能。由于在方法中需要频繁地开启和关闭数据库连接，这样不仅严重影响程序执行效率，还可能导致服务器压力巨大。

那么这种情况下使用ThreadLocal是再适合不过的了，因为ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。

二.深入解析ThreadLocal类

> 在上面谈到了对ThreadLocal的一些理解，那我们下面来看一下具体ThreadLocal是如何实现的。
>
> 先了解一下ThreadLocal类提供的几个方法：

Java代码

1. **public** T get() { }
2. **public void** set(T value) { }
3. **public void** remove() { }
4. **protected** T initialValue() { }

get()方法是用来获取ThreadLocal在当前线程中保存的变量副本，set()用来设置当前线程中变量的副本，remove()用来移除当前线程中变量的副本，initialValue()是一个protected方法，一般是用来在使用时进行重写的，它是一个延迟加载方法，下面会详细说明。

首先我们来看一下ThreadLocal类是如何为每个线程创建一个变量的副本的。

先看下get方法的实现：

```
/**
 * Returns the value in the current thread's copy of this
 * thread-local variable.  If the variable has no value for the
 * current thread, it is first initialized to the value returned
 * by an invocation of the {@link #initialValue} method.
 *
 * @return the current thread's value of this thread-local
 */
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null)
            return (T)e.value;
    }
    return setInitialValue();
}
```

第一句是取得当前线程，然后通过getMap(t)方法获取到一个map，map的类型为ThreadLocalMap。然后接着下面获取到<key,value>键值对，注意这里获取键值对传进去的是 this，而不是当前线程t。

如果获取成功，则返回value值。

如果map为空，则调用setInitialValue方法返回value。

我们上面的每一句来仔细分析：

首先看一下getMap方法中做了什么：

```
/**
 * Get the map associated with a ThreadLocal. Overridden in
 * InheritableThreadLocal.
 *
 * @param  t the current thread
 * @return the map
 */
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

可能大家没有想到的是，在getMap中，是调用当期线程t，返回当前线程t中的一个成员变量threadLocals。

那么我们继续取Thread类中取看一下成员变量threadLocals是什么：

马士兵教育 www.mashibing.com

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;
```

实际上就是一个ThreadLocalMap，这个类型是ThreadLocal类的一个内部类，我们继续取看ThreadLocalMap的实现：

```
static class ThreadLocalMap {

    /**
     * The entries in this hash map extend WeakReference, using
     * its main ref field as the key (which is always a
     * ThreadLocal object).  Note that null keys (i.e. entry.get()
     * == null) mean that the key is no longer referenced, so the
     * entry can be expunged from table.  Such entries are referred to
     * as "stale entries" in the code that follows.
     */
    static class Entry extends WeakReference<ThreadLocal> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal k, Object v) {
            super(k);
            value = v;
        }
    }
```

可以看到ThreadLocalMap的Entry继承了WeakReference，并且使用ThreadLocal作为键值。

然后再继续看setInitialValue方法的具体实现：

```
/**
 * Variant of set() to establish initialValue. Used instead
 * of set() in case user has overridden the set() method.
 *
 * @return the initial value
 */
private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

很容易了解，就是如果map不为空，就设置键值对，为空，再创建Map，看一下createMap的实现：

```
/**
 * Create the map associated with a ThreadLocal. Overridden i
 * InheritableThreadLocal.
 *
 * @param t the current thread
 * @param firstValue value for the initial entry of the map
 * @param map the map to store.
 */
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

至此，可能大部分朋友已经明白了ThreadLocal是如何为每个线程创建变量的副本的：

首先，在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量）。

初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。

然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找。

下面通过一个例子来证明通过ThreadLocal能达到在每个线程中创建变量副本的效果：

Java代码

```
1. **package** com.bijian.study;
2.
3. **public** **class** Test {
4.
5. ThreadLocal<Long> longLocal = **new** ThreadLocal<Long>();
6. ThreadLocal<String> stringLocal = **new** ThreadLocal<String>();
7.
8. **public** **void** set() {
9. longLocal.set(Thread.currentThread().getId());
10. stringLocal.set(Thread.currentThread().getName());
11. }
12.
13. **public** **long** getLong() {
14. **return** longLocal.get();
15. }
16.
17. **public** String getString() {
18. **return** stringLocal.get();
19. }
20.
21. **public** **static** **void** main(String[] args) **throws**
InterruptedException {
22. **final** Test test = **new** Test();
23.
24. test.set();
25. System.out.println(test.getLong());
```

```
26. System.out.println(test.getString());
27.
28. Thread thread1 = **new** Thread() {
29. **public** **void** run() {
30. test.set();
31. System.out.println(test.getLong());
32. System.out.println(test.getString());
33. };
34. };
35. thread1.start();
36. thread1.join();
37.
38. System.out.println(test.getLong());
39. System.out.println(test.getString());
40. }
41. }
```

运行结果：

Text代码

```
1. 1
2. main
3. 11
4. Thread-0
5. 1
6. main
```

从这段代码的输出结果可以看出，在main线程中和thread1线程中，longLocal保存的副本值和stringLocal保存的副本值都不一样。最后一次在main线程再次打印副本值是为了证明在main线程中和thread1线程中的副本值确实是不同的。

总结一下：

1）实际的通过ThreadLocal创建的副本是存储在每个线程自己的threadLocals中的；

2）为何threadLocals的类型ThreadLocalMap的键值为ThreadLocal对象，因为每个线程中可有多个threadLocal变量，就像上面代码中的longLocal和stringLocal；

3）在进行get之前，必须先set，否则会报空指针异常；

如果想在get之前不需要调用set就能正常访问的话，必须重写initialValue()方法。

因为在上面的代码分析过程中，我们发现如果没有先set的话，即在map中查找不到对应的存储，则会通过调用setInitialValue方法返回i，而在setInitialValue方法中，有一个语句是T value = initialValue()，而默认情况下，initialValue方法返回的是null。

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null)
            return (T)e.value;
    }
    return setInitialValue();
}

/**
 * Variant of set() to establish initialValue. Used instead
 * of set() in case user has overridden the set() method.
 *
 * @return the initial value
 */
private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

马士兵教育 www.mashibing.com

看下面这个例子：

Java代码

```
1. **package** com.bijian.study;
2.
3. **public** **class** Test02 {
4.
5. ThreadLocal<Long> longLocal = **new** ThreadLocal<Long>();
6. ThreadLocal<String> stringLocal = **new** ThreadLocal<String>();
7.
8. **public** **void** set() {
9. longLocal.set(Thread.currentThread().getId());
10. stringLocal.set(Thread.currentThread().getName());
11. }
12.
13. **public** **long** getLong() {
14. **return** longLocal.get();
15. }
16.
17. **public** String getString() {
18. **return** stringLocal.get();
19. }
20.
21. **public** **static** **void** main(String[] args) **throws**
InterruptedException {
22. **final** Test02 test = **new** Test02();
```

```
23.
24. System.out.println(test.getLong());
25. System.out.println(test.getString());
26.
27. Thread thread1 = **new** Thread() {
28. **public** **void** run() {
29. test.set();
30. System.out.println(test.getLong());
31. System.out.println(test.getString());
32. };
33. };
34. thread1.start();
35. thread1.join();
36.
37. System.out.println(test.getLong());
38. System.out.println(test.getString());
39. }
40. }
```

运行结果：

Text代码

```
1. Exception in thread "main" java.lang.NullPointerException
2. at com.bijian.study.Test02.getLong(Test02.java:14)
3. at com.bijian.study.Test02.main(Test02.java:24)
```

在main线程中，没有先set，直接get的话，运行时会报空指针异常。

但是如果改成下面这段代码，即重写了initialValue方法：

Java代码

```
1. **package** com.bijian.study;
2.
3. **public** **class** Test03 {
4.
5. ThreadLocal<Long> longLocal = **new** ThreadLocal<Long>() {
6. **protected** Long initialValue() {
7. **return** Thread.currentThread().getId();
8. };
9. };
10.
11. ThreadLocal<String> stringLocal = **new** ThreadLocal<String>() {
12. **protected** String initialValue() {
13. **return** Thread.currentThread().getName();
14. };
15. };
16.
17. **public** **void** set() {
18. longLocal.set(Thread.currentThread().getId());
19. stringLocal.set(Thread.currentThread().getName());
20. }
21.
22. **public** **long** getLong() {
23. **return** longLocal.get();
24. }
```

```
25.
26. **public** String getString() {
27. **return** stringLocal.get();
28. }
29.
30. **public** **static** **void** main(String[] args) **throws**
InterruptedException {
31. **final** Test03 test = **new** Test03();
32.
33. //test.set();
34. System.out.println(test.getLong());
35. System.out.println(test.getString());
36.
37. Thread thread1 = **new** Thread() {
38. **public** **void** run() {
39. //test.set();
40. System.out.println(test.getLong());
41. System.out.println(test.getString());
42. };
43. };
44. thread1.start();
45. thread1.join();
46.
47. System.out.println(test.getLong());
48. System.out.println(test.getString());
49. }
50. }
```

运行结果：

Text代码

```
1. 1

2. main

3. 8

4. Thread-0

5. 1

6. main
```

就可以直接不用先set而直接调用get了。

三.ThreadLocal的应用场景

最常见的ThreadLocal使用场景为 用来解决数据库连接、Session管理等。如：

数据库连接：

Java代码

```
1. **private** **static** ThreadLocal<Connection> connectionHolder = **new**
ThreadLocal<Connection>() {

2. **public** Connection initialValue() {

3. **return** DriverManager.getConnection(DB_URL);

4. }

5. };

6.

7. **public** **static** Connection getConnection() {

8. **return** connectionHolder.get();

9. }
```

Session管理：

Java代码

```
1. **private** **static** **final** ThreadLocal threadSession = **new**
ThreadLocal();
2.
3. **public** **static** Session getSession() **throws** InfrastructureException
{
4. Session s = (Session) threadSession.get();
5. **try** {
6. **if** (s == **null**) {
7. s = getSessionFactory().openSession();
8. threadSession.set(s);
9. }
10. } **catch** (HibernateException ex) {
11. **throw** **new** InfrastructureException(ex);
12. }
13. **return** s;
14. }
```

JVM老年代和新生代的比例?

JVM-堆学习之新生代老年代持久带的使用关系

之前被问到一个问题，大意是这样的：假如jvm参数中，最大堆内存分配了800M，Eden区分配了200M，s0、s1各分配50M，持久带分配了100M，老年代分配了400M，问现在启动应用程序后，可使用的最大内存有多少?

这个问题的问本质其实是想考我们对JVM堆内存的GC回收原理的理解，我当时回答说是650M，我这样回答，当时是这样考虑的：持久带100M 是用来存放静态类、方法或者变量的，程序启动后，这部分内存不会再分配给新创建的对象，s0和S1总不能同时是满的，即总有一个是空的或者非满状态，所以减去这两部分剩下的部分就是可用的，即800-100-50 =650. 后来下面仔细看了看GC分代回收算法，我这个答案可能是错的。为了弄清楚这个问题，自己简单写个测试类，结合Jconsole，来进行分析，思路是这样的：

- **创建一个测试类，静态代码块中初始化一个List，观察持久带使用情况**
- **创建临时对象，调整对象个数，分别观察Eden、s0、S1内存交换情况**
- **塞满新生代，观察Eden区 s0、s1、old区的情况**
- ***手动执行GC，观察分析gc回收日志**

以上每一个步骤执行完系统挂起，观察jconsole，进行分析

1.设置JVM启动参数

这里为了方便跑程序，设置的参数较小

```
 -Xms50M -Xmx110M -Xmn20M -XX:PermSize=10M -XX:MaxPermSize=10M -
XX:SurvivorRatio=8 -XX:NewRatio=4 -XX:+PrintGCDetails1
```

参数分析如下

```
 * -Xms50M:初始堆大小 50M
 * -Xmx100M: 最大堆大小 100M
 * -Xmn20M: 新生代 20M （Eden + survivor0 + survivor1 ）
 * -XX:PermSize=10M :初始持久带10M

 * -XX:SurvivorRatio=8 : Eden区与Survivor区的大小比值=8,表示两个
   Survivor区之和与一个Eden区的比值为2:8,一个Survivor区占整个年轻代
   的1/10即Eden = 16M ,s0 = 2M ,s1=2M

 * -XX:NewRatio=4：表示新生代与年老代所占比值为1:4,新生代占整个堆栈的
   1/5,（110-10）*0.2 =20M，老年代占整个堆的4/5,(最大100*0.8=80M)

12345678910111213
```

总结推理：堆最大110M 持久带10M 新生代Eden=16M,S0=2M,s1=2M 老年代old区最大不超过80M，Eden区域到达16M时候触发幸存区复制，新生代达到20M的时候，触发YGC，old区域满的时候出发FGC，下面我们来验证这个理论。

测试类

```
/**
 * Created by zhugh on 17/6/16.
 */
public class JvmTest {

 List<DataObject> list = new ArrayList();
    static {
        //模拟持久带存储
        int index = 0;
        while(index < 10000){
            DataObject object = new DataObject();
            object.setAge(index + 30);
            object.setName("李四"+index);
```

```
            list.add(object);
            index ++;
        }
        System.out.println("生成持久带数据：" + list.size() + "条");
        System.out.println("[1]:生成持久带数据，请观察jconsole堆内存情况");
    }

    //模拟新生代存储
    public static void main(String[] agrs){
        try {
            System.out.print("请输入任意键开始给Eden区塞对象");
            System.in.read(); //模拟挂起，观察jconsole
            System.out.print("开始往Eden灌数据");
            List<DataObject> tempList = new ArrayList();
            int i = 100000;
            while(i > 0){
                DataObject object = new DataObject();
                object.setAge(i + 20);
                object.setName("张三"+i);
                i --;
                tempList.add(object);
            }
            System.out.println("生成新生代数据：" + tempList.size() + "条");
            System.out.println("[2]:生成新生代数据完成，请观察jconsole堆内存情况");
            System.in.read();
            System.out.println("输入任意键执行GC回收：");

            System.gc();
            System.out.println("[3]:GC回收完成，请观察jconsole堆内存情况");
            System.in.read();
            System.exit(0);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    static class DataObject{
        private String name;
        private int age;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

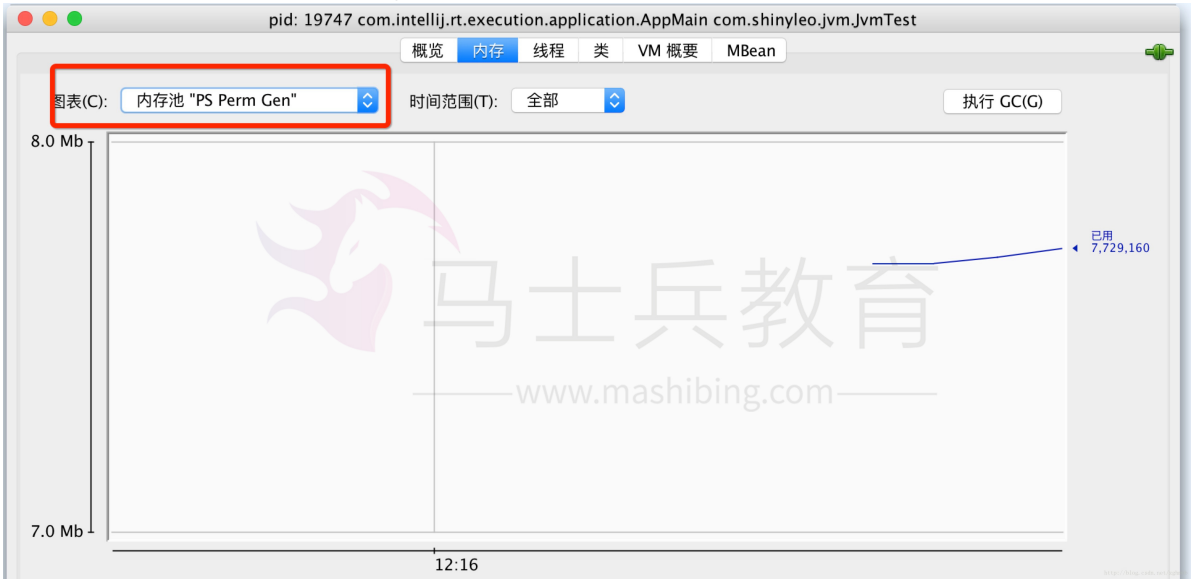
        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

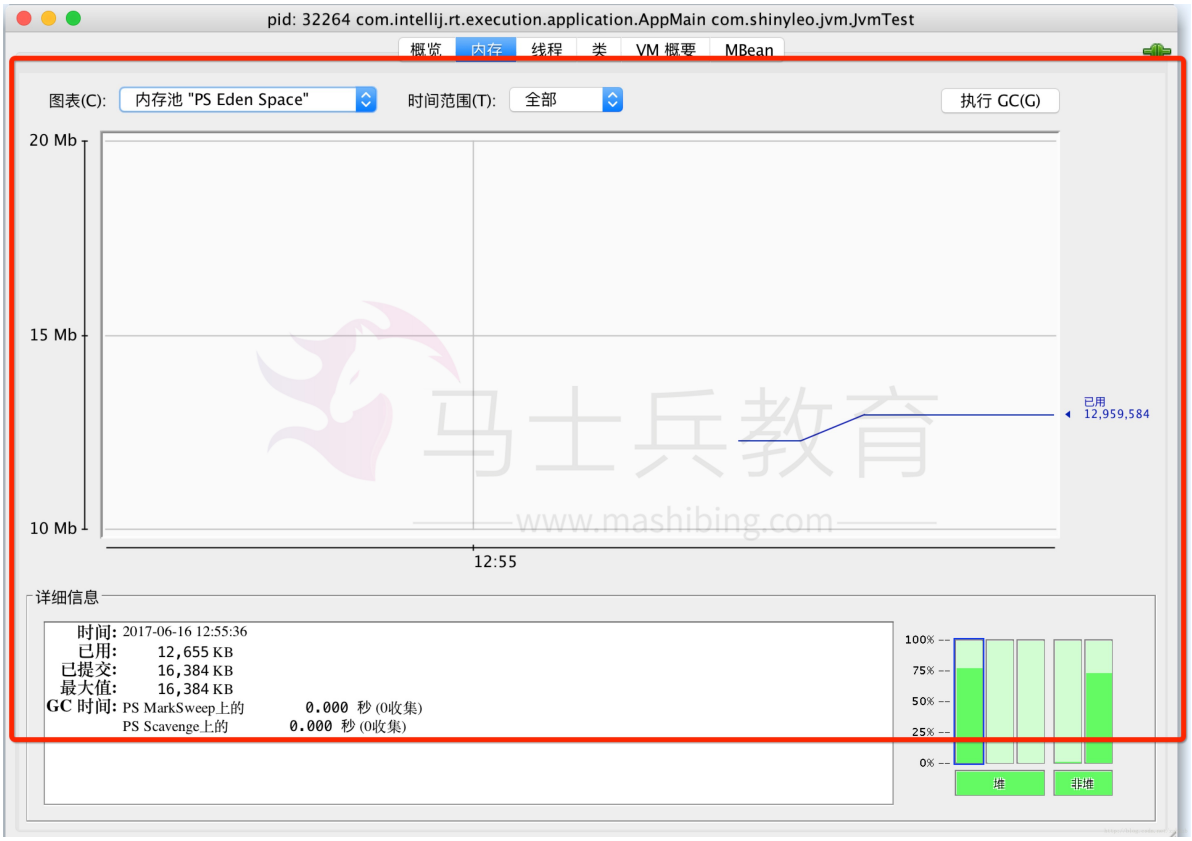
        @Override
        public String toString() {
```

```
            return "DataObject{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }
    }
}
```

[1]持久代先灌10000条记录，观察jconsole持久带占用情况如图1，使用不到8M：

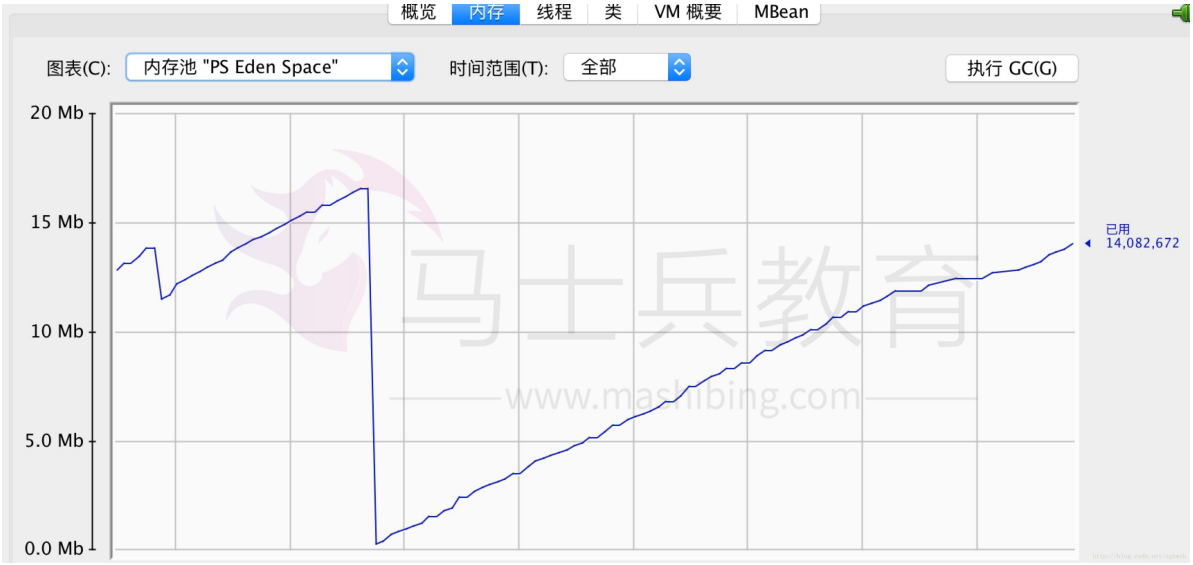


我们在观察Eden区使用的情况如下图2：



看到Eden区域已经使用了约12m，这是应为类加载过程中会创建一些对象，这些对象是非用户定义的。

我们开始给Eden区域第一次灌对象，之后效果如下图3:

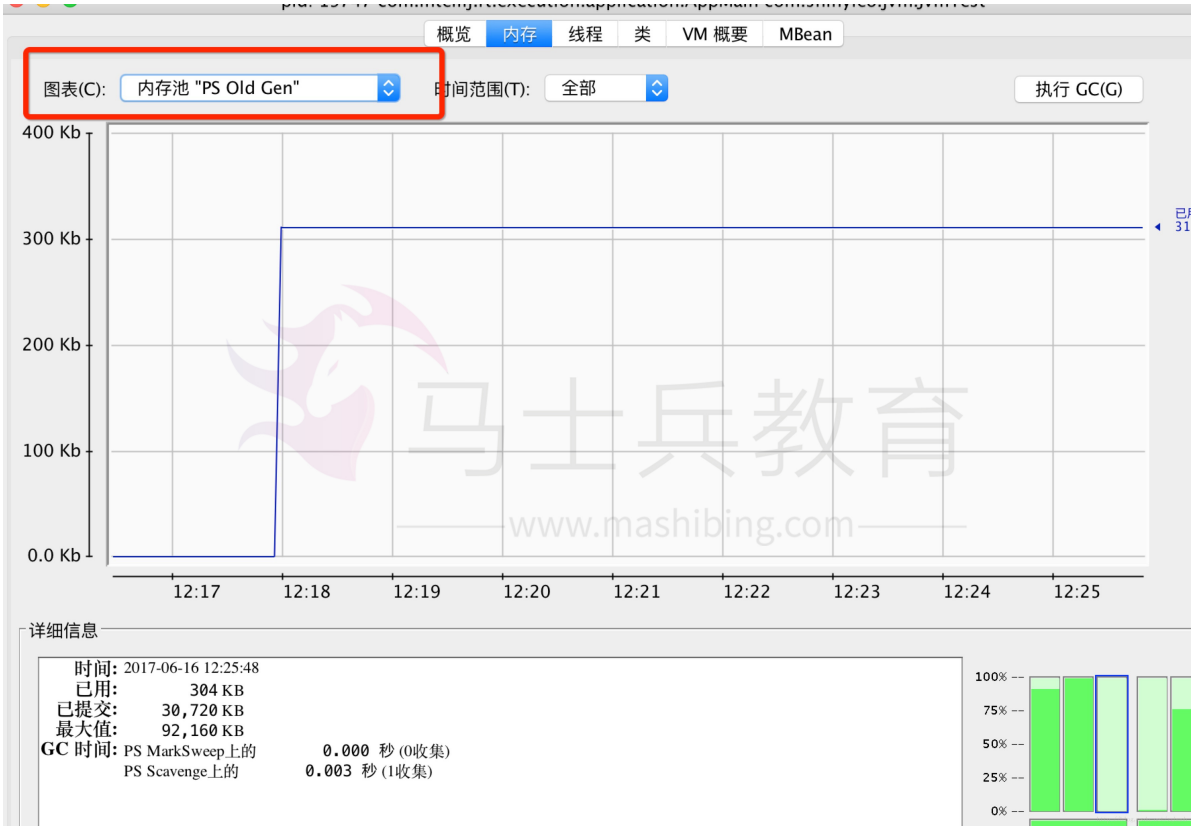


我们可以看到上图，Eden区域使用到16M的时候进行了一次从Eden区域往幸存区的复制，从Eden区复制了2M数据到了幸存区域。

再来看看幸存区的情况，如下图4：

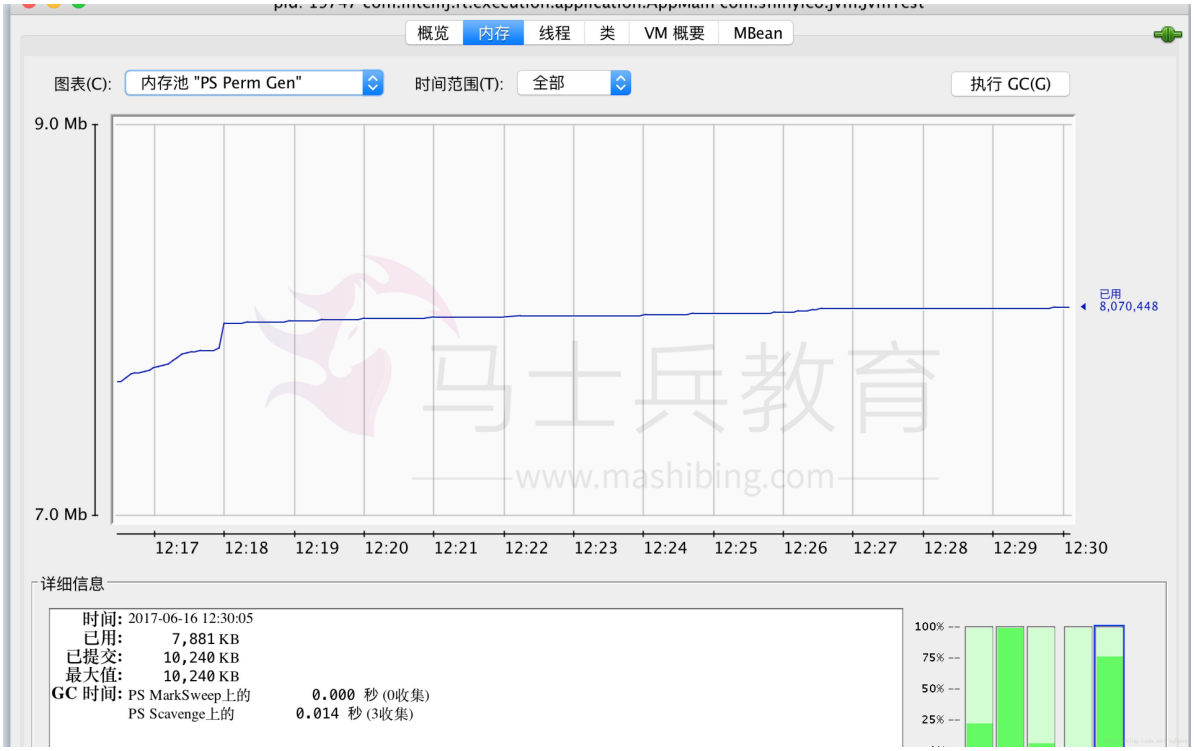


上图分析：由于触发了一次幸存区复制，所以幸存区使用了2M，目前为止符合我的推测。
我们再来看看old区现在的情况 图5：



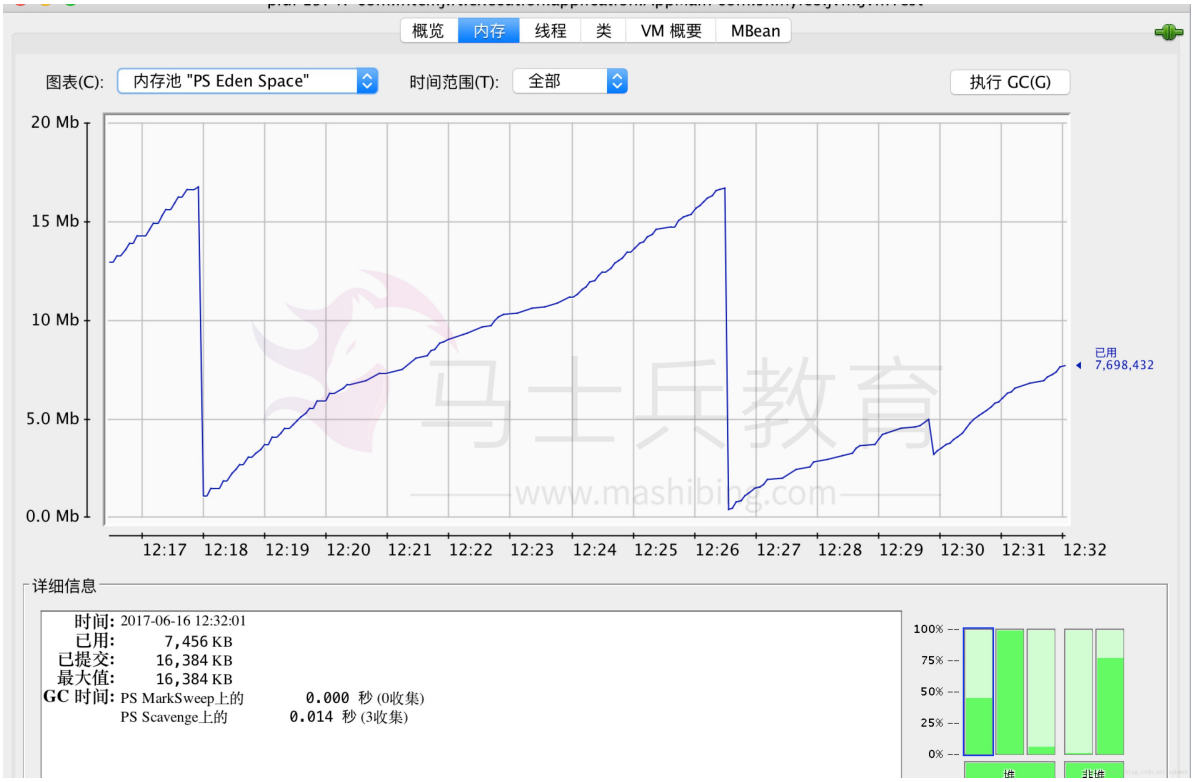
分析，由于现在还未触发YGC，old区域使用300Kb,这是程序无关外的其他额外内存占用，可以忽略，基本上还是满存状态。

现在我们把挂起的程序继续向下执行，继续往新生代灌注数据，再次触发Eden区域往幸存区的复制，看看此时持久带的情况如下图6：



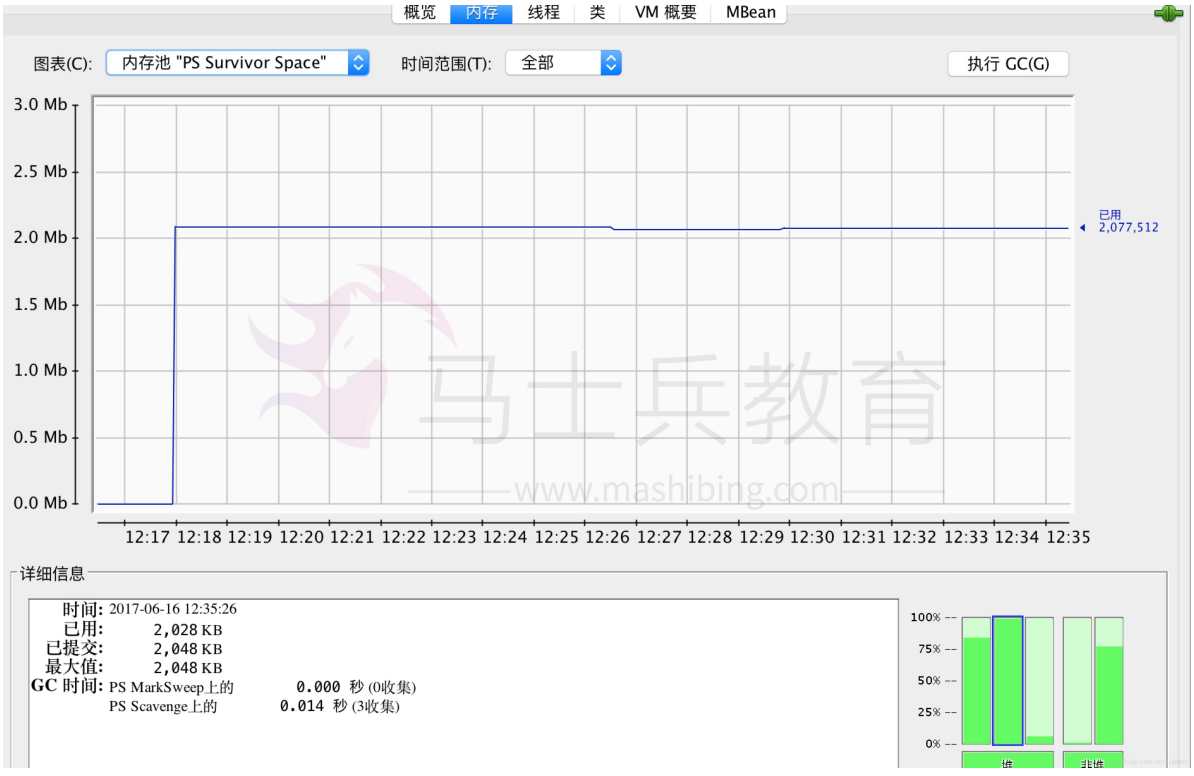
分析：持久带几乎没有变，这验证了临时对象的生成不占用持久带内存是正确的。

再来观察此时Eden区的情况图7：



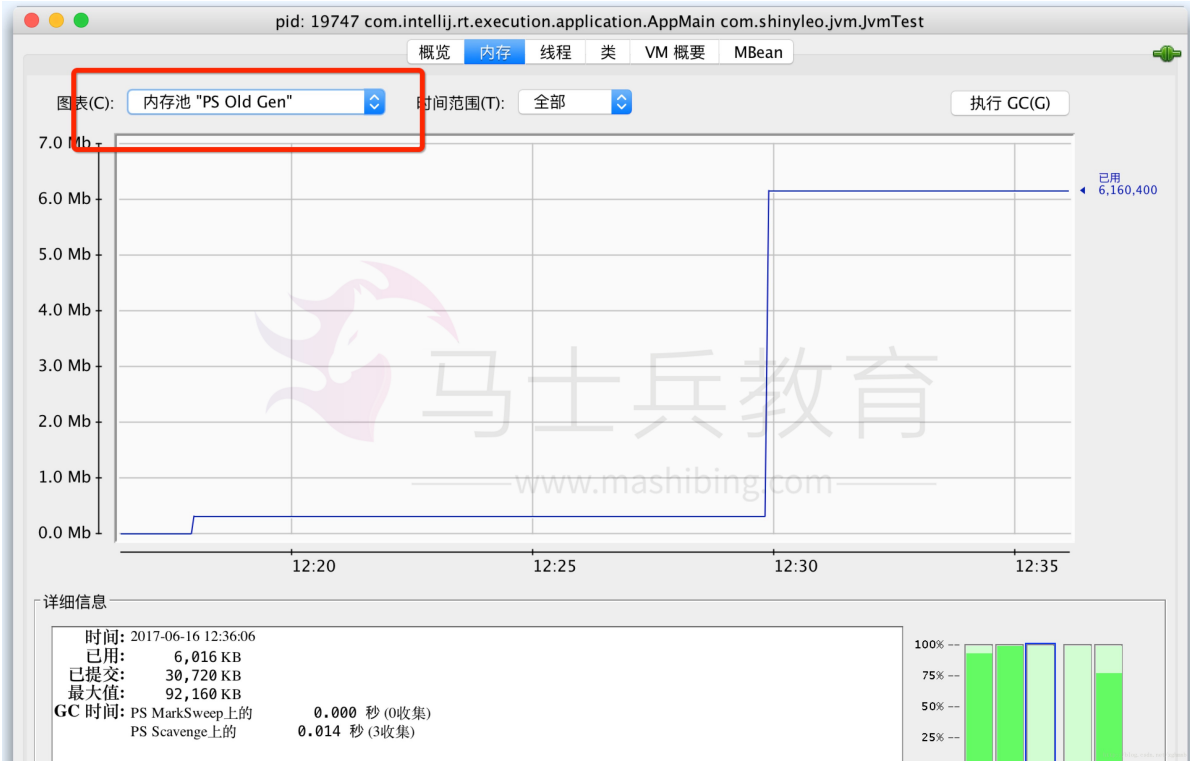
分析，此时Eden区第二次被灌满16M，部分数据复制到了s0或者s1区域，又触发了一次复制。

我们再来看看s0和s1的情况图8：



分析，果然，幸存区又一次进行了复制，至此幸存区已经占满，触发YGC。

再来看看此时老年代的情况图9：



分析，如图，老年代此时已经使用了6M内存，说明此时已经触发了YGC，
那么这6M是哪里来的呢？看到这里如果明白的人，就会懂了，让我们来一起来分析一下：
第二次往Eden区域灌数据的时候，如图7，触发第二次幸存区复制后，Eden区域剩余约8M数据，说明有另外8M被进行复制回收了，那么这8M回收到哪里去了呢？由于第二次触发幸存区复制时，幸存区s0或者s1还有2M可以用（第一次Eden往幸存区复制后已经使用了2M），所以这8m有2M复制到了幸存区，此时幸存区已4M满，剩下6M被复制到老年区，所以老年区的6M = Eden区被回收的8M-新生代承担的2M。当然这里有一个概念我需要强调下，老年区的数据并不是直接从Eden区进行复制，而是先从Eden区倒腾到幸存区，再从幸存区倒腾到old区。
程序运行结束，敢看GC回收日志如下

```
  Heap
 PSYoungGen      total 18432K, used 375K [0x00000007fec00000, 0x0000000800000000,
0x0000000800000000)
  eden space 16384K, 2% used
[0x00000007fec00000,0x00000007fec5dc38,0x00000007ffc00000)
  from space 2048K, 0% used
[0x00000007ffe00000,0x00000007ffe00000,0x0000000800000000)
  to   space 2048K, 0% used
[0x00000007ffc00000,0x00000007ffc00000,0x00000007ffe00000)
 ParOldGen       total 30720K, used 1947K [0x00000007f9200000,
0x00000007fb000000, 0x00000007fec00000)
  object space 30720K, 6% used
[0x00000007f9200000,0x00000007f93e6d80,0x00000007fb000000)
 PSPermGen       total 10240K, used 7762K [0x00000007f8800000,
0x00000007f9200000, 0x00000007f9200000)
  object space 10240K, 75% used
[0x00000007f8800000,0x00000007f8f94ab0,0x00000007f9200000)123456789
```

从GC汇总日志上看，新生代总共占用约20M，Eden占用约16M，s0、s1各2M，老年代总共30M左右，使用了6%，为什么现在是6%了？是因为代码中有人工进行GC回收的动作，老年代的6M，被回收释放了，剩余约1.8M，持久带约10M，使用了7M多点，占用约75%。

问题，和之前的结论大体一致，只有老年代为什么才30M，和预期的80M差这么多？我理解为老年代并不是初始化的时候就开辟80M的空间，而是当不断触发YGC往老年代复制数据的时候，一旦超过30M，会在开辟更多空间，最大不超过80M，只是推论，也希望同志们能够进行验证。

剩下如果想验证FGC的情况，需要继续往内存中灌数据，直到把old区域灌满，触发FGC，此时可以观察老年代的内存大小，这部分内容，就留给各位自己去验证吧。

栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗？

1. 从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。
2. 堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。
3. 栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。
4. 面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

为什么不把基本类型放堆中呢？

因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

堆中存什么？栈中存什么？

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处：））。

为什么不把基本类型放堆中呢？因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和

对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

Java中的参数传递时传值呢？还是传引用？

要说明这个问题，先要明确两点：

1. 不要试图与C进行类比，Java中没有指针的概念
2. 程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

明确以上两点后。Java在方法调用传递参数时，因为没有指针，所以它都是进行传值调用（这点可以参考C的传值调用）。因此，很多书里面都说Java是进行传值调用，这点没有问题，而且也简化的C中复杂性。

但是传引用的错觉是如何造成的呢？在运行栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的思想，才使得Java的垃圾回收成为可能。

Java中，栈的大小通过-Xss来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现java.lang.StackOverflowError异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点

java中四种引用类型（对象的强、软、弱和虚引用）

对象的强、软、弱和虚引用

在JDK 1.2以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及（reachable）状态，程序才能使用它。从JDK 1.2版本开始，把对象的引用分为4种级别，从而使程序能更加灵活地控制对象的生命周期。这4种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

⑴强引用（StrongReference）
强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。 ps：强引用其实也就是我们平时A a = new A()这个意思。

⑵软引用（SoftReference）
如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存（下文给出示例）。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

⑶弱引用（WeakReference）
弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。
弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

⑷虚引用（PhantomReference）
“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。
虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

ReferenceQueue queue = **new** ReferenceQueue ();

PhantomReference pr = **new** PhantomReference (object, queue);
程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

使用软引用构建敏感数据的缓存

1 为什么需要使用软引用
　首先，我们看一个雇员信息查询系统的实例。我们将使用一个Java语言实现的雇员信息查询系统查询存储在磁盘文件或者数据库中的雇员人事档案信息。作为一个用户，我们完全有可能需要回头去查看几分钟甚至几秒钟前查看过的雇员档案信息(同样，我们在浏览WEB页面的时候也经常会使用“后退”按钮)。这时我们通常会有两种程序实现方式:一种是把过去查看过的雇员信息保存在内存中，每一个存储了雇员档案信息的Java对象的生命周期贯穿整个应用程序始终;另一种是当用户开始查看其他雇员的档案信息的时候，把存储了当前所查看的雇员档案信息的Java对象结束引用，使得垃圾收集线程可以回收其所占用的内存空间，当用户再次需要浏览该雇员的档案信息的时候，重新构建该雇员的信息。很显然，第一种实现方法将造成大量的内存浪费，而第二种实现的缺陷在于即使垃圾收集线程还没有进行垃圾收集，包含雇员档案信息的对象仍然完好地保存在内存中，应用程序也要重新构建一个对象。我们知道，访问磁盘文件、访问网络资源、查询数据库等操作都是影响应用程序执行性能的重要因素，如果能重新获取那些尚未被回收的Java对象的引用，必将减少不必要的访问，大大提高程序的运行速度。

2 如果使用软引用
　SoftReference的特点是它的一个实例保存对一个Java对象的软引用，该软引用的存在不妨碍垃圾收集线程对该Java对象的回收。也就是说，一旦SoftReference保存了对一个Java对象的软引用后，在垃圾线程对这个Java对象回收前，SoftReference类所提供的get()方法返回Java对象的强引用。另外，一旦垃圾线程回收该Java对象之后，get()方法将返回null。
看下面代码:

MyObject aRef = **new** MyObject();

SoftReference aSoftRef = **new** SoftReference(aRef);

此时，对于这个MyObject对象，有两个引用路径，一个是来自SoftReference对象的软引用，一个来自变量aReference的强引用，所以这个MyObject对象是强可及对象。
随即，我们可以结束aReference对这个MyObject实例的强引用:

aRef = **null**;

此后，这个MyObject对象成为了软可及对象。如果垃圾收集线程进行内存垃圾收集，并不会因为有一个SoftReference对该对象的引用而始终保留该对象。Java虚拟机的垃圾收集线程对软可及对象和其他一般Java对象进行了区别对待:软可及对象的清理是由垃圾收集线程根据其特定算法按照内存需求决定的。也就是说，垃圾收集线程会在虚拟机抛出OutOfMemoryError之前回收软可及对象，而且虚拟机会尽可能优先回收长时间闲置不用的软可及对象，对那些刚刚构建的或刚刚使用过的"新"软可反对象会被虚拟机尽可能保留。在回收这些对象之前，我们可以通过:
MyObject anotherRef=(MyObject)aSoftRef.get();

重新获得对该实例的强引用。而回收之后，调用get()方法就只能得到null了。

3 使用ReferenceQueue清除失去了软引用对象的SoftReference
　作为一个Java对象，SoftReference对象除了具有保存软引用的特殊性之外，也具有Java对象的一般性。所以，当软可及对象被回收之后，虽然这个SoftReference对象的get()方法返回null,但这个SoftReference对象已经不再具有存在的价值，需要一个适当的清除机制，避免大量SoftReference对象带来的内存泄漏。在java.lang.ref包里还提供了ReferenceQueue。如果在创建SoftReference对象的时候，使用了一个ReferenceQueue对象作为参数提供给SoftReference的构造方法，如:

ReferenceQueue queue = **new** ReferenceQueue();

SoftReference ref = **new**
SoftReference(aMyObject, queue);

那么当这个SoftReference所软引用的aMyOhject被垃圾收集器回收的同时，ref所强引用的SoftReference对象被列入ReferenceQueue。也就是说，ReferenceQueue中保存的对象是Reference对象，而且是已经失去了它所软引用的对象的Reference对象。另外从ReferenceQueue这个名字也可以看出，它是一个队列，当我们调用它的poll()方法的时候，如果这个队列中不是空队列，那么将返回队列前面的那个Reference对象。
在任何时候，我们都可以调用ReferenceQueue的poll()方法来检查是否有它所关心的非强可及对象被回收。如果队列为空，将返回一个null,否则该方法返回队列中前面的一个Reference对象。利用这个方法，我们可以检查哪个SoftReference所软引用的对象已经被回收。于是我们可以把这些失去所软引用的对象的SoftReference对象清除掉。常用的方式为:

SoftReference ref = **null**;

while ((ref = (EmployeeRef) q.poll()) != **null**) {

// 清除ref

}

完整代码:

```
ReferenceQueue q = new ReferenceQueue();

// 获取数据并缓存
Object obj = new Object();
SoftReference sr = new SoftReference(obj, q);

// 下次使用时
Object obj = (Object)sr.get();
if (obj == null){
  // 当软引用被回收后才重新获取
  obj = new Object();
}

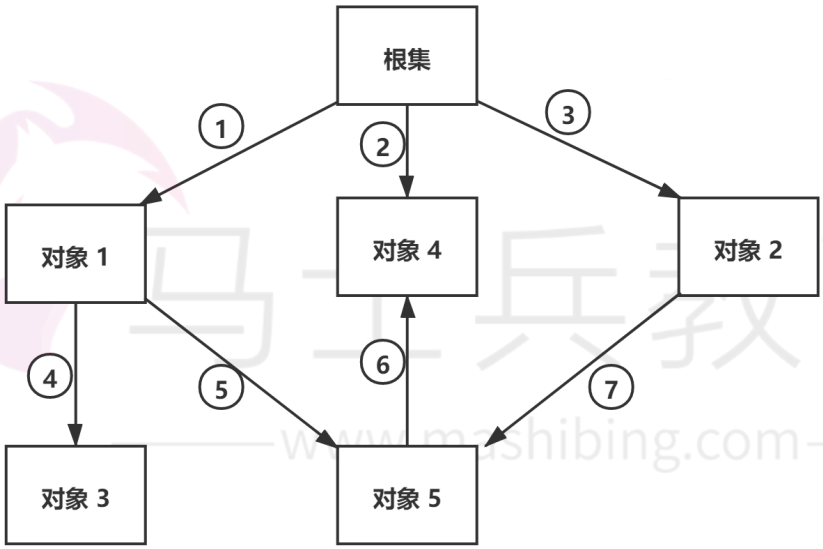
// 清理被收回后剩下来的软引用对象
```

```
SoftReference ref = null;
while((ref = q.poll()) != null){
  // 清理工作
}
```

疑问：

那现在问题来了，若一个对象的引用类型有多个，那到底如何判断它的可达性呢？其实规则如下：

```
1. 单条引用链的可达性以最弱的一个引用类型来决定；
2. 多条引用链的可达性以最强的一个引用类型来决定；
```



我们假设图2中引用①和③为强引用，⑤为软引用，⑦为弱引用，对于对象5按照这两个判断原则，路径①-⑤取最弱的引用⑤，因此该路径对对象5的引用为软引用。同样，③-⑦为弱引用。在这两条路径之间取最强的引用，于是对象5是一个软可及对象（当将要发生OOM时则会被回收掉）。

软引用、弱引用和虚引用均为抽象类 java.lang.ref.Reference 的子类，而与引用队列和GC相关的操作大多在抽象类Reference中实现。

讲一讲内存分代及生命周期。

管中窥豹——从对象的生命周期梳理JVM内存结构、GC、类加载、AOP编程及性能监控

如题，本文的宗旨既是透过对象的生命周期，来梳理JVM内存结构及GC相关知识，并辅以AOP及双亲委派机制原理，学习不仅仅是海绵式的吸收学习，还需要自己去分析why，加深对技术的理解和认知，祝大家早日走上自己的“成金之路”。

Java对象的创建

本部分，从攻城狮编写.java文件入手，详解了编译、载入、AOP原理。
读过《程序员的自我修养》的朋友，对程序的编译及执行会有一个很清晰的认识：编译其实就是将人类能理解的代码文件转译为机器/CPU能执行的文件(包括数据段、代码段)，而执行的过程，则是根据文件头部字节的标识(简称魔数)，映射为对应的文件结构体，找到程序入口，当获取到CPU执行权限时，将方法压栈，执行对应的指令码，完成相应的逻辑操作。
而对应.java文件，则先需要使用javac进行编译，编译后的.class文件，此文件将java程序能读懂的数据段和代码段，之后用java执行文件，既是载入.class文件，找到程序入口，并根据要执行的方法，不停的压栈、出栈，进行逻辑处理。

class文件载入过程

加载

在加载阶段，虚拟机需要完成以下三件事情：

> - 通过一个类的全限定名来获取其定义的二进制字节流。
> - 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
> - 在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口。

即相当于在内存中将代码段和数据段关联起来，组织好Class对象的内存空间，作为对象成员和方法的引入入口，并将.class及方法载入Perm内存区。相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。

连接

分为四个阶段：

- 第一阶段是验证，确保被加载的类的正确性。

> 这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。验证阶段大致会完成4个阶段的检验动作：
> ***文件格式验证***验证字节流是否符合Class文件格式的规范；例如：是否以0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。
> ***元数据验证***对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了java.lang.Object之外。
> ***字节码验证***通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。
> ***符号引用验证***确保解析动作能正确执行。

> 验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用***-Xverifynone参数来关闭大部分的类验证措施***，以缩短虚拟机类加载的时间。

- 第二阶段是准备：为类的静态变量分配内存，并将其初始化为默认值

> 准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：
> 1、这时候进行内存分配的仅包括***类变量（static）***，而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。

> 2、这里所设置的初始值通常情况下是数据类型***默认的零值***（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。如public static int value = 3；那么变量value在准备阶段过后的初始值为0，而不是3，因为这时候尚***未开始执行任何ava方法***，而把value赋值为3的putstatic指令是在程序编译后，存放于类构造器()方法之中的，所以把value赋值为3的动作将在***初始化阶段才会执行***。这里还需要注意如下几点：

> i.对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于***局部变量来说，在使用前必须显式地为其赋值***，否则编译时不通过。
> ii.对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被final修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，***系统不会为final修饰的常量赋予默认零值***。
> iii.对于***引用数据类型*** reference来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。
> iv.如果在***数组***初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

> 3、如果类字段的字段属性表中存在***ConstantValue属性***，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值。如public static final int value = 3；编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为3。

- 第三阶段是解析:把类中的符号引用转换为直接引用

> 解析阶段是虚拟机将常量池内的***符号引用替换为直接引用***的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。符号引用就是一组符号来描述目标，可以是任何字面量。
> 直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

这个阶段的主要目的将编译后的虚拟地址(类似动态库，库数据段都是0x00开始，载入内存后需要与实际分配的地址关联起来)与实际运行的地址关联起来。

- 第四阶段是初始化，为类的静态变量赋予正确的初始值，JVM负责对类进行初始化，主要对类变量进行初始化，包括给声明类变量指定初始值，和为类static变量指定静态代码块地址。

> 1、假如这个类还没有被加载和连接，则程序先加载并连接该类
> 2、假如该类的直接父类还没有被初始化，则先初始化其直接父类
> 3、假如类中有初始化语句，则系统依次执行这些初始化语句

> 类初始化时机：只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：
> – 创建类的实例，也就是new的方式
> – 访问某个类或接口的静态变量，或者对该静态变量赋值
> – 调用类的静态方法
> – 反射（如Class.forName(“com.xxx.Test”)）
> – 初始化某个类的子类，则其父类也会被初始化
> – Java虚拟机启动时被标明为启动类的类（Java Test），直接使用java.exe命令来运行某个主类。

拓展：双亲委派机制与AOP面向切面编程原理

JVM最底层的载入是BootstrapClassLoader，为C语言编写，从Java中引用不到，其上是ExtClassLoader，然后是AppClassLoader，普通类中getContextClassLoader()得到的是AppClassLoader，getContextClassLoader().getParent()得到的是ExtClassLoader，而再往上getParent()为null，即引用不到BootstrapLoader。

> ***启动类加载器：BootstrapClassLoader***，负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如rt.jar，所有的java.*开头的类均被BootstrapClassLoader加载）。启动类加载器是无法被Java程序直接引用的。

> ***扩展类加载器：ExtensionClassLoader***，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载DK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.*开头的类），开发者可以直接使用扩展类加载器。

> ***应用程序类加载器：ApplicationClassLoader***，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

> 应用程序都是由这三种类加载器**互相配合进行加载**的，如果有必要，我们还可以加入自定义的类加载器。因为JVM自带的ClassLoader只是懂得从本地文件系统加载标准的java class文件，因此如果编写了自己的ClassLoader，便可以做到如下几点：
> 1）在执行非置信代码之前，自动验证数字签名。
> 2）动态地创建符合用户特定需要的定制化构建类。
> 3）从特定的场所取得java class，例如数据库中和网络中。

> **双亲委派机制**
> 简单来说既是先拿父类加载器加载class，父类加载失败后再使用本类加载器加载。当AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。当ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader去完成。如果BootStrapClassLoader加载失败（例如在$JAVA_HOME/jre/lib里未查找到该class），会使用ExtClassLoader来尝试加载；若ExtClassLoader也加载失败，则会使用AppClassLoader来加载，如果AppClassLoader也加载失败，则会报出异常ClassNotFoundException。

> ***其它相关点***
> 1.全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入。
> 2.父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时，才使用本类加载器从自己的类路径中加载该类。
> 3.缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效。

--

> **AOP面向切面编程**
> AOP 专门用于处理系统中分布于各个模块（不同方法）中的交叉关注点的问题，在 Java EE 应用中，常常通过AOP来处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等，在不改变已有代码的情况下，静态/动态的插入代码。

> 将AOP放到这里的主要原因是因为AOP改变的class文件，达到嵌入方法的目的，***静态模式***使用AspectJ进行由.java到.class文件编译。而***动态模式***时使用CGLIB载入使用javac编译的.class文件后，使用动态代理的方式，将要执行的方法嵌入到原有class方法中，完成在内存中对class对象的构造，这也就是所谓***动态代理技术***的内在原理。同时静态方式在载入前已经修好完.class文件，而动态方式在.class载入时需要做额外的处理，导致性能受到一定影响，但其优势是无须使用额外的编译器。总体的技术的切入点在于在修改机器执行码，达到增加执行方法的目的。

对象创建及方法调用

数据分类：基本类型与引用类型

> 基本类型包括：byte,short,int,long,char,float,double,boolean,returnAddress
> 引用类型包括：类类型，接口类型和数组。

数据存储：使用堆存储对象信息

方法调用：使用栈来解决方法嵌套调用，而栈内部由一个个栈帧构成，调用一个方法时，在当前栈上压入一个栈帧，此栈帧包含局部变量表，操作栈等子项，每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。表面上代码在运行时，是通过程序计数器不断执行下一条指令；而实际指令运算等操作是通过控制操作栈的操作数入栈和出栈，将操作数在局部变量表和操作栈之间转移。

> 这里对内存的分配做个深入的扩展，解释下基础类型的***自动装箱*** boxing
> 以"Object obj = new Object();"为例，一个空Object对象占用8byte空间，而obj引用占用4byte，此条语句执行后共占用12byte；而Java中***对象大小是8的整数倍***，则Boolean b = new Boolean(true)至少需要20byte(16byte+4byte)，而如果直接使用基本数据类型boolean b = true则仅仅需要1byte，在栈帧中存储；为优化此问题，JVM提出了基本类型的自动装载技术，来自动化进行基本类型与基本类型对象间的转换，来降低内存的使用量。

内存集中管理(模型及GC机制)

- JVM内存模型图如下

> 内存结构主要有三大块：堆内存、方法区和栈。
> 1.***堆内存***是JVM中最大的一块由Young Generation(年轻代、新生代)和Old Generation(年老代)组成，而Young Generation内存又被分成三部分，Eden空间、From Survivor空间、To Survivor空间，默认情况下年轻代按照8:1:1的比例来分配。
> 2.***方法区***存储类信息、常量、静态变量等数据，是线程共享的区域，为与Java堆区分，方法区还有一个别名Non-Heap(非堆)。
> 3.***栈***又分为java虚拟机栈(方法执行的内存区，每个方法执行时会在虚拟机栈中创建栈帧)和本地方法栈(虚拟机的Native方法执行的内存区)主要用于方法的执行。
> 4.内存设置参数：
> -Xms设置堆的最小空间大小。
> -Xmx设置堆的最大空间大小。
> -XX:NewSize设置新生代最小空间大小。
> -XX:MaxNewSize设置新生代最大空间大小。
> -XX:PermSize设置永久代最小空间大小。
> -XX:MaxPermSize设置永久代最大空间大小。
> -Xss设置每个线程的堆栈大小。
> -XX:SurvivorRatio=x #Eden区与Survivor区的大小比值，默认为8(Eden:From-Survivor=8:1)
> -XX:NewRatio=x #年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代)默认值为2，即年轻代:年老代=1:2(这里与数学的比值有差异)

- Minor GC

对象分配先从Eden空间与From Survivor空间获取内存，当两者中空间不足时，进行Minor GC，将Eden与From Survivor空间存活的对象，Copy到To Survivor空间，然后清空Eden与From Survivor空间，之后将To Survivor空间对象年龄加1，并将To Survivor空间设置为From Survivor空间，保证Minor GC时To Survivor空间始终为空；而当对象年龄为15后(默认是 15，可以通过参数-XX:MaxTenuringThreshold 来设定)，将存活对象放入老年代。

例外情况：
1.对于一些***较大的对象***(即需要分配一块较大的连续内存空间)则是直接进入到老年代。虚拟机提供了一个-XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在老年代分配。避免在新生代采用复制算法收集内存时，在Eden区及两个Survivor区之间发生大量的内存复制。
2.为了更好的适应不同的程序，虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中***相同年龄所有对象大小的总和大于Survivor空间的一半***，年龄大于或等于该年龄的对象可以直接进入Old Generation，无需等到MaxTenuringThreshold中要求的年龄。

> 这里的疑问，按照默认的8:1:1设置，一个To Survivor空间占10%空间，每次Minor GC能保证To Survivor空间够用吗？IBM的研究表明，98%的对象都是很快消亡的，大部分的对象在创建后很快就不再使用。这里可以根据GC detail来查看和分析比例设置是否合理。

- Full GC

工作：同时回收年轻代、年老代，按照配置的不同算法进行回收。
时机：在Minor GC触发时，会检测之前每次晋升到老年代的***平均大小***是否大于老年代的剩余空间，如果大于，改为直接进行一次Full GC；如果小于则查看HandlePromotionFailure设置(是否允许担保，使用Old Gerneration空间担保)，如果允许，那仍然进行Minor GC，如果不允许，则也要改为进行一次Full GC。
取平均值进行比较其实仍然是一种动态概率的手段，也就是说如果某次Minor GC存活后的对象突增，大大高于平均值的话，依然会导致担保失败，这样就只好在失败后重新进行一次Full GC。

回收算法

首先判断对象是否存活，一般有两种方式：

> - 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
> - 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的、不可达对象。

在Java语言中，GC Roots包括：
--虚拟机栈中引用的对象。
--方法区中类静态属性实体引用的对象。
--方法区中常量引用的对象。
--本地方法栈中JNI引用的对象。

标记-清除算法:首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。
缺点：一个是**效率**问题，标记和清除过程的效率都不高；另外一个是**空间**问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

复制算法：对**空间**问题的改进，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。
*缺点*是这种算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低。如果不想浪费50%的空间，就需要有额外的空间进行分配担保(HandlePromotionFailure设置为true)，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

标记-压缩/整理算法：对复制算法在老年代上的改进，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

分代收集算法：把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。
在***新生代***中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。
而***老年代***中，因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

回收器

新生代收集器

- Serial收集器

> 串行收集器是最古老，最稳定以及效率高的收集器，使用停止复制方法，只使用一个线程去串行回收；***垃圾收集的过程中会Stop The World***（服务暂停）；
> 参数控制：使用-XX:+UseSerialGC可以使用Serial+Serial Old模式运行进行内存回收（这也是虚拟机在Client模式下运行的默认值）
> ***缺点***是串行效率较低

- ParNew收集器

> ParNew收集器其实就是Serial收集器的***多线程版本***，使用停止复制方法。新生代***并行***，其它工作线程暂停。
> 参数控制：使用-XX:+UseParNewGC开关来控制使用ParNew+Serial Old收集器组合收集内存；使用-XX:ParallelGCThreads来设置执行内存回收的线程数。

- Parallel收集器

> Parallel Scavenge收集器类似ParNew收集器，***Parallel收集器更关注CPU吞吐量，即运行用户代码的时间/总时间***，使用停止复制算法。可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例。

> 参数控制：使用-XX:+UseParallelGC开关控制使用Parallel Scavenge+Serial Old收集器组合回收垃圾（这也是在Server模式下的默认值）；使用-XX:GCTimeRatio来设置用户执行时间占总时间的比例，默认99，即1%的时间用来进行垃圾回收。使用-XX:MaxGCPauseMillis设置GC的最大停顿时间（这个参数只对Parallel Scavenge有效），用开关参数-XX:+UseAdaptiveSizePolicy可以进行动态控制，如自动调整Eden/Survivor比例，老年代对象年龄，新生代大小等，这个参数在ParNew下没有。

老年代收集器

- Serial Old收集器：老年代收集器，单线程收集器，串行，使用"标记-整理"算法（整理的方法是Sweep（清理）和Compact（压缩），
- Parallel Old 收集器

> 多线程机制与Parallel Scavenge差不错，使用标记整理（与Serial Old不同，这里的整理是Summary（汇总）和Compact（压缩），汇总的意思就是将幸存的对象复制到预先准备好的区域，而不是像Sweep（清理）那样清理废弃的对象）算法，在Parallel Old执行时，仍然需要暂停其它线程。Parallel Old在多核计算中很有用。这个收集器是在JDK 1.6中，与Parallel Scavenge配合有很好的效果。

> 参数控制： 使用-XX:+UseParallelOldGC开关控制使用Parallel Scavenge +Parallel Old组合收集器进行收集。

- CMS收集器：老年代收集器（新生代使用ParNew）

> CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。
> 从名字（包含“Mark Sweep”）上就可以看出CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为6个步骤，其中***初始标记***、***重新标记***这两个步骤仍然需要“Stop The World”，包括：

1. 初始标记(CMS-initial-mark)：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。

2. 并发标记(CMS-concurrent-mark)：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用，此阶段会打印2条日志：CMS-concurrent-mark-start，CMS-concurrent-mark。
3. 并发预清理(CMS-concurrent-preclean)：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。 此阶段会打印2条日志：CMS-concurrent-preclean-start，CMS-concurrent-preclean。
4. 下一阶段是CMS-concurrent-abortable-preclean阶段，加入此阶段的目的是使cms gc更加可控一些，作用也是执行一些预清理，以减少Rescan阶段造成应用暂停的时间.
   通过两个参数来来控制是否进行下一阶段：
   -XX:CMSScheduleRemarkEdenSizeThreshold（默认2M）：即当eden使用小于此值时；
   -XX:CMSScheduleRemarkEdenPenetratio（默认50%）：当Eden区占用比例此比例时
   在concurrent preclean阶段之后，如果Eden占用率高于CMSScheduleRemarkEdenSizeThreshold，开启'concurrent abortable preclean'，并且持续的precleanig直到Eden占比超过CMSScheduleRemarkEdenPenetratio，之后，开启remark阶段，另外，-XX:CMSMaxAbortablePrecleanTime：当abortable-preclean阶段执行达到这个时间时会结束进入下一阶段。
5. 重标记CMS-concurrent-remark：由于上面三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。
6. 并发清理CMS-concurrent-sweep：所有不再被应用的对象将从堆里清除掉。
7. 并发重置CMS-concurrent-reset：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。
   尽管CMS收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过“stop-the-world”方法来进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻带的垃圾回收时间通常是相当短的。

优点：并发收集、低停顿
缺点：产生大量空间碎片、并发阶段会降低吞吐量

> - 堆碎片：CMS收集器并没有任何碎片整理的机制，可能出现总的堆大小远没有耗尽，但因为没有足够连续的空间却不能分配对象，只能触发Full GC来解决，造成应用停顿。

- 对象分配率高：如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败，从某种程度上说，老年代将没有足够的可用空间来容纳一个从年轻代提升过来的对象。经常被证实是老年代有大量不必要的对象。一个可行的办法就是增加***年轻代的堆大小***，以防止年轻代短生命的对象提前进入老年代。另一个办法就似乎利用分析器，快照运行系统的堆转储，并且***分析过度的对象分配***，找出这些对象，最终减少这些对象的申请。

参数控制

> -XX:+UseConcMarkSweepGC 使用CMS收集器
> 当使用-XX：+UseConcMarkSweepGC时，-XX：UseParNewGC会自动开启。因此，如果年轻代的并行GC不想开启，可以通过设置-XX：-UseParNewGC来关掉
> -XX:+CMSClassUnloadingEnabled相对于并行收集器，CMS收集器默认不会对永久代进行垃圾回收。
> -XX:+CMSConcurrentMTEnabled当该标志被启用时，并发的CMS阶段将以多线程执行(因此，多个GC线程会与所有的应用程序线程并行工作)。该标志已经默认开启，如果顺序执行更好，这取决于所使用的硬件，多线程执行可以通过-XX:-CMSConcurremntMTEnabled禁用(注意是-号)。
> -XX:+UseCMSCompactAtFullCollection Full GC后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长
> -XX:+CMSFullGCsBeforeCompaction 设置进行几次Full GC后，进行一次碎片整理
> -XX:ParallelCMSThreads 设定CMS的线程数量（一般情况约等于可用CPU数量）
> -XX:CMSMaxAbortablePrecleanTime：当abortable-preclean阶段执行达到这个时间时才会结束

> -XX:CMSInitiatingOccupancyFraction，-XX:+UseCMSInitiatingOccupancyOnly来决定什么时间开始垃圾收集；如果设置了***-XX:+UseCMSInitiatingOccupancyOnly***，那么只有当old代占用确实达到了-XX:CMSInitiatingOccupancyFraction参数所设定的比例时才会触发cms gc；如果没有设置-XX:+UseCMSInitiatingOccupancyOnly，那么系统会根据统计数据自行决定什么时候触发cms gc；因此有时会遇到设置了80%比例才cms gc，但是50%时就已经触发了，就是因为这个参数没有设置的原因.

G1收集器

G1 GC是Jdk7的新特性之一、Jdk7+版本都可以自主配置G1作为JVM GC选项；作为JVM GC算法的一次重大升级、DK7u后G1已相对稳定、且未来计划替代CMS。
不同于其他的分代回收算法、G1将堆空间划分成了互相独立的区块。每块区域既有可能属于O区、也有可能是Y区，且每类区域空间可以是不连续的（***对比CMS的O区和Y区都必须是连续的***）。区别如下：

> 1. G1在压缩空间方面有优势

1. G1通过将内存空间分成区域（Region）的方式避免内存碎片问题
2. Eden, Survivor, Old区不再固定、在内存使用效率上来说更灵活
3. G1可以通过设置预期停顿时间（Pause Time）来控制垃圾收集时间避免应用雪崩现象
4. G1在回收内存后会马上同时做合并空闲内存的工作、而CMS默认是在STW（stop the world）的时候做
5. G1会在Young GC中使用、而CMS只能在O区使用

就目前而言、CMS还是默认首选的GC策略、可能在以下场景下G1更适合：

> 1. 大量内存的系统提供一个保证GC低延迟的解决方案，也就是说堆内存在6GB及以上，稳定和可预测的暂停时间小于0.5秒。

1. Full GC 次数太频繁或者消耗时间太长
2. 应用在运行过程中会产生大量内存碎片、需要经常压缩空间
3. 想要更可控、可预期的GC停顿周期；防止高并发下应用雪崩现象
4. 对象分配的频率或代数提升(promotion)显著变化
5. 受够了太长的垃圾回收或内存整理时间(超过0.5~1秒)
   注意: 如果正在使用CMS或ParallelOldGC，而应用程序的垃圾收集停顿时间并不长，那么建议继续使用现在的垃圾收集器。使用最新的JDK时，并不要求切换到G1收集器。

- 内存模型：

> G1堆由多个区（region）组成，每个区大小1M~32M，逻辑上区有3种类型，包括（Eden、Survivor、Old），按分代划分包括：年轻代（Young Generation）和老年代（Old Generation）。

> 如果从 ParallelOldGC 或者 CMS收集器迁移到 G1，可能会看到JVM进程占用更多的内存(a larger JVM process size)。 这在很大程度上与 “accounting” 数据结构有关，如 Remembered Sets 和 Collection Sets。
> Remembered Sets 简称 RSets。跟踪指向某个heap区内的对象引用。 堆内存中的每个区都有一个 RSet。 RSet 使heap区能并行独立地进行垃圾集合。 RSets的总体影响小于5%。
> Collection Sets 简称 CSets。收集集合, 在一次GC中将执行垃圾回收的heap区。GC时在CSet中的所有存活数据(live data)都会被转移(复制/移动)。集合中的heap区可以是 Eden, survivor, 和/或 old generation。CSets所占用的JVM内存小于1%。

- Young GC

> Young GC是stop-the-world活动，会导致整个应用线程的停止。其过程如下：

1. 新对象进入Eden区
2. 存活对象拷贝到Survivor区；

3. 存活时间达到年龄阈值时，对象晋升到Old区。
   young gc是多线程并行执行的。

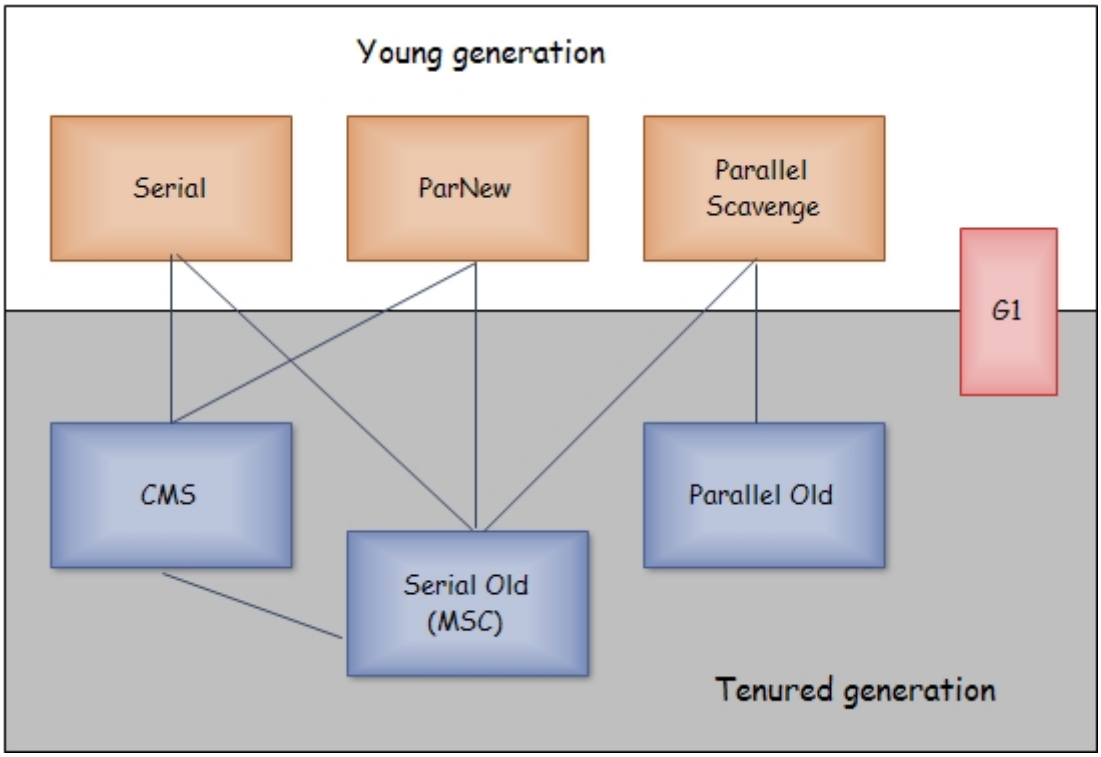
- Old GC阶段描述为：

> 1. 初始化标记(stop_the_world事件)：这是一个stop_the_world的过程，是随着年轻代GC做的，标记survivor区域(根区域),这些区域可能含有对老年代对象的引用。

1. 根区域扫描：扫描survivor区域中对老年代的引用，这个过程和应用程序一起执行的，这个阶段必须在年轻代GC发生之前完成。
2. 并发标记：查找整个堆中存活的对象，这也是和应用程序一起执行的。这个阶段可以被年轻代的垃圾收集打断。
3. 重新标记(stop-the-world事件)：完成堆内存活对象的标记。使用了一个叫开始前快照snapshot-at-the-beginning (SATB)的算法，这个会比CMS collector使用的算法快。
4. 清理(stop-the-world事件，并且是并发的)：对存活的对象和完全空的区域进行统计(stop-the-world)、刷新Remembered Sets(stop-the-world)、重置空的区域，把他们放到free列表(并发)(译者注：大体意思就是统计下哪些区域又空了，可以拿去重新分配了)
5. 复制(stop-the-world事件)：这个stop-the-world的阶段是来移动和复制存活对象到一个未被使用的区域，这个可以是年轻代区域,打日志的话就标记为 [GC pause (young)]。或者老年代和年轻代都用到了，打日志就会标记为[GC Pause (mixed)]。

- 参数分析

> -XX:+UseG1GC使用G1 GC
> -XX:MaxGCPauseMillis=n设置一个暂停时间期望目标，这是一个软目标，JVM会近可能的保证这个目标
> -XX:InitiatingHeapOccupancyPercent=n内存占用达到整个堆百分之多少的时候开启一个GC周期，G1 GC会根据整个栈的占用，而不是某个代的占用情况去触发一个并发GC周期，0表示一直在GC，默认值是45
> -XX:NewRatio=n年轻代和老年代大小的比例，默认是2
> -XX:SurvivorRatio=n eden和survivor区域空间大小的比例，默认是8
> -XX:MaxTenuringThreshold=n晋升的阈值，默认是15（译者注：一个存活对象经历多少次GC周期之后晋升到老年代)
> -XX:ParallelGCThreads=n GC在并行处理阶段试验多少个线程，默认值和平台有关。（译者注：和程序一起跑的时候，使用多少个线程)
> -XX:ConcGCThreads=n并发收集的时候使用多少个线程，默认值和平台有关。（译者注:stop-the-world的时候，并发处理的时候使用多少个线程)
> -XX:G1ReservePercent=n预留多少内存，防止晋升失败的情况，默认值是10
> -XX:G1HeapRegionSize=n G1 GC的堆内存会分割成均匀大小的区域，这个值设置每个划分区域的大小，这个值的默认值是根据堆的大小决定的。最小值是1Mb，最大值是32Mb



调优

命令行工具(以Linux场景进行详解)

- GC监控：

> ./jstat -gc pid 3s对pid GC每隔3s进行监控

- 谁动了我的CPU

1. top查看CPU使用情况，或通过CPU使用率收集，找到CPU占用率高Java进程，假设其进程编号为pid；
2. 使用top -Hp pid(或ps -Lfp pid或者ps -mp pid -o THREAD,tid,time)查看pid进程中占用CPU较高的线程，假设其编号为tid；
3. 使用Linux命令，将tid转换为16进制数，printf '%0x\n' tid，假设得到值txid；
4. 使用jstack pid | grep txid查看线程CPU占用代码，然后根据得到的对象信息，去追踪代码，定位问题。

- 死锁

> 另外，jstack -l long listings，会打印出额外的锁信息，在发生死锁时可以用jstack -l pid来观察锁持有情况。

- Memory

1. 查看java 堆（heap）使用情况jmap -J-d64 -heap pid，其中-J-d64为64位机器标识，启动
2. 查看堆内存(histogram)中的对象数量及大小jmap -J-d64 -histo pid；而查看存活对象-histo:live，这个命令执行，JVM会先触发gc，然后再统计信息。
   如果有***大量对象在持续被引用，并没有被释放掉***，那就产生了内存泄露，就要结合代码，把不用的对象释放掉。
   其中class name中B 代表 byte，C 代表 char，D 代表 double，F 代表 float，I 代表 int，J 代表 long，Z 代表 boolean，前边有 [代表数组， [I 就相当于 int[]，对象用 [L+ 类名表示
3. ***程序内存不足或者频繁GC，很可能存在内存泄露***将内存使用的详细情况输出到文件，执行命令：jmap -J-d64 -dump:format=b,file=heapDump pid；然后用jhat命令可以参看 jhat -port 5000 heapDump

--

> 内存泄漏OOM，通常做法：
> 方法1. 首先配置JVM启动参数，让JVM在遇到OutOfMemoryError时自动生成Dump文件-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path
> 方法2. 使用上面第3步进行进行分析；
> 方法3. 使用eclipse的MAT分析工具对dump文件进行分析

> 1. 发现问题

- 使用uptime命令查看CPU的Load情况，Load越高说明问题越严重；
- 使用jstat查看FGC发生的频率及FGC所花费的时间，FGC发生的频率越快、花费的时间越高，问题越严重；

1. 使用jmap -head 观察老年代大小，当快达到配置的阀值CMSInitiatingOccupancyFraction时，将对象导出进行分析；
2. 对head中对象进行命令行排序分析：./jmap -J-d64 -histo pid | grep java | sort -k 3 -n -r | more 按照bytes大小进行排序；数量最多排序将-k 3换成-k 2即可

图形化工具jvisualvm.exe

一般结合JMX分析，分析远程tomcat状态

- tomcat配置

> 修改远程tomcat的catalina.sh配置文件，在其中增加(不走权限校验。只是打开jmx端口)：
> JAVA_OPTS="$JAVA_OPTS -Djava.rmi.server.hostname=192.168.122.128 -
> Dcom.sun.management.jmxremote.port=18999 -
> Dcom.sun.management.jmxremote.ssl=false
> -Dcom.sun.management.jmxremote.authenticate=false"

--

> 如果连接的是公网上的Tomcat，那么就要注意安全性了，接下来看看使用用户名和密码连接
> JAVA_OPTS='-Xms128m -Xmx256m -XX:MaxPermSize=128m -
> Djava.rmi.server.hostname=10.10.23.10
> -Dcom.sun.management.jmxremote.port=9999
> -Dcom.sun.management.jmxremote.ssl=false
> -Dcom.sun.management.jmxremote.authenticate=true
> -Dcom.sun.management.jmxremote.password.file=/path/to/passwd/jmxremote.password
> -Dcom.sun.management.jmxremote.access.file=/path/to/passwd/jmxremote.access'

以下分别编辑jmxremote.password与jmxremote.access两个文件
jmxremote.password

> monitorRole 123456
> controlRole 123456789

jmxremote.access

> monitorRole readonly
> controlRole readwrite

完成这两个文件后修改jmxremote.password的权限chmod 600 jmxremote.password

- 工具使用
  配置后重启tomcat，然后从本机上使用jvisualvm.exe，按照配置输入远程地址、用户名密码，即可进行监控，包括CPU、Head、Thread、Objects等分析

优化策略使用

本质上是减少Full GC的次数(Minor GC很快基本不会有影响)

- 第一个问题既是目前年轻代、年老代分别使用什么收集器；
- 针对不同的收集器，进行优化：串行改并行、针对目前的Head对象情况，根据实际场景设置SurvivorRatio与NewRatio；
- 根据对象回收情况，判断是否需要压缩，减小碎片化。

> 如果是频繁创建对象的应用，可以适当增加新生代大小。常量较多可以增加持久代大小。对于单例较多的对象可以增加老生代大小。比如spring应用中。

GC选择，在JDK5.0以后，JVM会根据当前系统配置进行判断。一般执行-Server命令便可以。gc包括三种策略：串行，并行，并发(使用CMS收集器老年代)。

> 吞吐量大的应用，一般采用并行收集，开启多个线程，加快gc的速率。
> 响应速度高的应用，一般采用并发收集，比如应用服务器。
> 年老代建议配置为并发收集器，由于并发收集器不会压缩和整理磁盘碎片，因此建议配置：
> -XX:+UseConcMarkSweepGC #并发收集年老代
> -XX:CMSInitiatingOccupancyFraction=80 # 表示年老代空间到80%时就开始执行CMS
> -XX:+UseCMSInitiatingOccupancyOnly #是阀值生效
> -XX:+UseCMSCompactAtFullCollection#打开对年老代的压缩。可能会影响性能，但是可以消除内存碎片。
> -XX:CMSFullGCsBeforeCompaction=10 # 由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此参数设置运行次FullGC以后对内存空间进行压缩、整理。

什么情况下触发垃圾回收?

Java JVM 8：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）

在什么时候

首先需要知道，GC又分为 minor GC 和 Full GC (也称为 Major GC)。Java 堆内存分为新生代和老年代，新生代中又分为1个 Eden 区域 和两个 Survivor 区域。

那么对于 Minor GC 的触发条件：**大多数情况下，直接在 Eden 区中进行分配。**如果 Eden区域没有足够的空间，那么就会发起一次 Minor GC；对于 Full GC（Major GC）的触发条件：也是如果老年代没有足够空间的话，那么就会进行一次 Full GC。

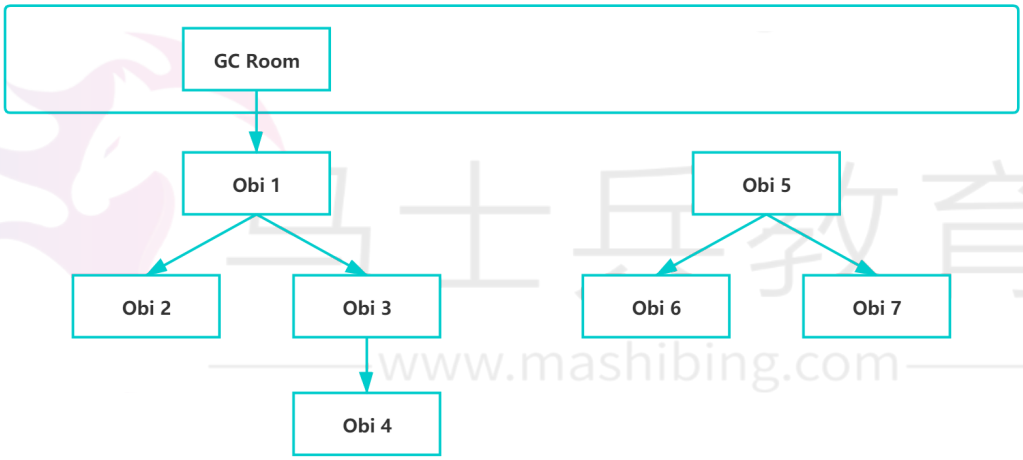
Ps：上面所说的只是一般情况下，实际上，需要考虑一个空间分配担保的问题：

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象的总空间。如果大于则进行Minor GC，如果小于则看HandlePromotionFailure设置是否允许担保失败（不允许则直接Full GC）。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于则尝试Minor GC（如果尝试失败也会触发Full GC），如果小于则进行Full GC。

但是，具体到什么时刻执行，这个是由系统来进行决定，是无法预测的。

对什么东西

主要根据可达性分析算法，如果一个对象不可达，那么就是可以回收的；如果一个对象可达，那么这个对象就不可以回收。对于可达性分析算法，它是通过一系列称为“GC Roots” 的对象作为起始点，当一个对象到 GC Roots 没有任何引用链相接的时候，那么这个对象就是不可达，就可以被回收。如下图:



马士兵教育
www.mashibing.com

这个GC Root 对象可以是一些静态的对象，Java方法的local变量或参数， native 方法引用的对象，活着的线程。

做了什么事情

主要做了清理对象，整理内存的工作。Java堆分为新生代和老年代，采用了不同的回收方式。例如新生代采用了复制算法，老年代采用了标记整理法。在新生代中，分为一个Eden 区域和两个Survivor区域，真正使用的是一个Eden区域和一个Survivor区域，GC的时候，会把存活的对象放入到另外一个Survivor区域中，然后再把这个Eden区域和Survivor区域清除。那么对于老年代，采用的是标记整理法，首先标记出存活的对象，然后再移动到一端。这样也有利于减少内存碎片。

如何选择合适的垃圾收集算法?

JVM给了三种选择：串行收集器、并行收集器、并发收集器 ，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置 进行判断。

吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

典型配置：

```
java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -
XX:ParallelGCThreads=20
```

-XX:+UseParallelGC ：选择垃圾收集器为并行收集器。 此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

```
-XX:ParallelGCThreads=20 ：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -
XX:ParallelGCThreads=20 -XX:+UseParallelOldGC
-XX:+UseParallelOldGC ：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -
XX:MaxGCPauseMillis=100
-XX:MaxGCPauseMillis=100 ： 设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -
XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy
-XX:+UseAdaptiveSizePolicy ：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。
```

响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置：

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -
XX:+UseConcMarkSweepGC -XX:+UseParNewGC
-XX:+UseConcMarkSweepGC ：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。
-XX:+UseParNewGC ：设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -
XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection
-XX:CMSFullGCsBeforeCompaction ：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。
-XX:+UseCMSCompactAtFullCollection ：打开对年老代的压缩。可能会影响性能，但是可以消除碎片
```

深入理解垃圾收集器和收集器的选择策略

前言：新生代的收集器有：Serial,ParNew,Parallel Scavenge等。老年代有：CMS,SerialOld,Paraller Old等。接下来将深入理解各个垃圾收集器的原理，以及它们如何在不同场景下进行搭配使用。
同时，先解释几个名次：
1，并行（Parallel）：多个垃圾收集线程并行工作，此时用户线程处于等待状态
2，并发（Concurrent）：用户线程和垃圾收集线程同时执行
3，吞吐量：运行用户代码时间 / （运行用户代码时间 + 垃圾回收时间）

(一) 新生代的收集器们
(1) ParNew
这是Serial收集器的多线程版本，使用多线程对垃圾收集，采用复制算法，同时需要暂时所有用户线程。除了使用多线程其他与Serial收集器相比并没有什么创新。但是为什么还要学习它，那是因为除了Serial

收集器，目前只有它能与CMS收集器配合工作。因此我们在Server中虚拟机首选项的新生代收集器还是它。可以使用的控制参数有：-XX:SurvivorRatio,-XX:PretenureSizeThreshold,-XX:HandlerPromotionFailure等（详情见官方手册）。

(2) Parallel Scavenge收集器
使用复制算法，同时也是并行收集器，相比ParNew，它更关注于达到一个可控制的吞吐量。高吞吐量可以高效率地利用CPU时间，尽快完成计算任务。所以这个收集器适合在后台运算而不需要很多交互的任务。接下来看看两个用于准备控制吞吐量的参数
1，-XX:MaxGCPauseMills（控制最大垃圾收集的时间）
设置一个大于0的毫秒数，收集器尽可能地保证内存回收不超过设定值。但是并不是设置地越小就越快。GC停顿时间缩短是以缩短吞吐量和新生代空间来换取的。
2，-XX:GCTimeRatio（设置吞吐量大小）
设置一个0 - 100的整数，也就是垃圾收集时间占总时间的比率，相当于吞吐量的倒数。

(二) 老年队的收集器们
(1) Serial Old
是Serial收集器的老年队版本，也是一个单线程收集器，使用标记 - 整理算法。这个收集器主要在于给Client模式下的虚拟机使用。如果在Server中，主要用途是：1，在JDK1.5前和Parallel Scavenge搭配使用。2，作为Concurrent Mode Failure时候使用。

(2)Parallel Old
这是Paraller Scanvenge收集器的老年队收集器，使用标记 - 整理方式。在这个方式没有产生之前，Parallel Scavenge只能选择Serial Old。由于被拖了后腿，那么Parallel Scavenge并不能在整体上获取吞吐量最大化的效果。甚至比不上CMS+ParNew的吞吐量。

(3)CMS收集器
这是一个以获取最短回收停顿时间为目标的并发收集器。对于重视服务响应时间，希望系统停顿时间尽可能短的，那么CMS就非常符合了。CMS收集器采用标记 - 清除实现。包括了四个步骤
1，初始标记：简单标记下GC Roots能直接关联到的对象，需要“Stop The World“
2，并发标记：进行GC Roots Tracing
3，重新标记：修正并发标记期间用户程序继续运行而导致标记发生变动那一部分对 象标记记录，需要“Stop The World“
4，并发清除

缺点：
1，无法处理浮动垃圾。由于并发清理阶段用户线程还在运行，程序自然会有新的垃圾产生，那么CMS将无法在这次收集中处理掉它们。只能等待下次GC再清理。由于垃圾回收阶段用户线程还需要运行。那么就需要预留足够的内存空间给用户线程使用，所以CMS不能等待老年队几乎完全快满了再进行收集。需要预留一部分空间提供并发收集时候的程序使用。如果运行期间预留的内存无法满足程序需要，那么就会出现“Concurrent Mode Failure“，这时候就启用Serial Old收集器进行老年代的收集。

2，对CPU资源敏感。在并发阶段虽然不会导致用户线程停顿，但是会因为占用一部分线程（或者说CPU资源）而导致应用程序变慢，吞吐量降低，默认是启动（CPU数量 + 3） / 4的线程数。

3，会产生大量的空间碎片。CMS是基于标记 - 清除算法实现的，那么收集结束时候会有大量空间碎片产生。这个时候就会给大对象分配带来麻烦，因为无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。那么我们可以使用:-XX:+UseCMSCompactAtFullCollection,在CMS收集器顶不住要进行FullGC时候开启内存碎片的合并整理过程，但是会加长停顿时间。还有一个参数 -XX:CMSFullGCsBeforeCompaction,表示用于设置执行多少次不压缩的Full GC后，跟着来一次带压缩的。

(三) 垃圾收集器选择的策略
(1) 吞吐量优先的并行收集器
参数配置：
1, -Xmx4g -Xms4g -Xmn2g -Xss200k -XX:+UseParallelGC -XX:ParallelGCThreads=8
说明：选择Parallel Scavenge收集器，然后配置多少个线程进行回收，最好与处理器数目相等。

2，-Xmx4g -Xms4g -Xmn2g -Xss200k -XX:+UseParallelGC -XX:ParallelGCThreads=8 -XX:+UseParallelOldGC
说明：配置老年代使用Parallel Old

3，-Xmx4g -Xms4g -Xmn2g -Xss200k -XX:+UseParallelGC -XX:MaxGCPauseMills=100
说明：设置每次年轻代垃圾回收的最长时间。如何不能满足，那么就会调整年轻代大小，满足这个设置

4，-Xmx4g -Xms4g -Xmn2g -Xss200k -XX:+UseParallelGC -XX:MaxGCPauseMills=100 -XX:+UseAdaptiveSizePolicy
说明：并行收集器会自动选择年轻代区大小和Survivor区的比例。

(2)响应时间优先的并发收集器
1， -Xmx4g -Xms4g -Xmn2g -Xss200k -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
说明：设置老年代的收集器是CMS，年轻代是ParNew

2，-Xmx4g -Xms4g -Xmn2g -Xss200k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection
说明：首先设置运行多少次GC后对内存空间进行压缩，整理。同时打开对年老代的压缩（会影响性能）

StringTable

探索StringTable提升YGC性能

很久很久以前看过笨神的一篇文章JVM源码分析之String.intern()导致的YGC不断变长，其原因是YGC过程需要对StringTable做扫描，而String.intern()就是在StringTable中保存这个对象的引用，如果String.intern()添加越来越多不同的对象，那么StringTable就越大，扫描StringTable的时间就越长，从而导致YGC耗时越长；那么如何确定YGC耗时越来越长是StringTable变大引起的呢?

介绍一个参数`-XX:+PrintStringTableStatistics`，看名字就知道这个参数的作用了：打印出StringTable的统计信息；再详细一点描述：`在JVM进程退出时，打印出StringTable的统计信息到标准日志输出目录中`。

> JDK版本要求：JDK 7u6 +

验证问题

验证代码如下：

```
import java.util.UUID;

/**
 * @author afei
 * @version 1.0.0
 * @since 2017年08月16日
 */
public class StringInternTest {
    public static void main(String[] args) throws Exception {
        for (int i=0; i<Integer.MAX_VALUE; i++){
            UUID.randomUUID().toString().intern();
            if (i>=100000 && i%100000==0){
```

```
                System.out.println("i="+i);
            }
        }
    }
}
```

执行命令如下：

```
java -verbose:gc -XX:+PrintGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC  -
XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -
XX:+PrintStringTableStatistics -Xmx1g -Xms1g -Xmn64m StringInternTest
```

从gc日志可以看出YGC时间越来越长：

```
[GC (Allocation Failure)  52480K->5691K(1042048K), 0.0109821 secs]
i=100000
[GC (Allocation Failure)  65261K->19731K(1042048K), 0.0233471 secs]
i=200000
[GC (Allocation Failure)  72211K->26796K(1042048K), 0.0266068 secs]
i=300000
[GC (Allocation Failure)  79276K->33860K(1042048K), 0.0262006 secs]
[GC (Allocation Failure)  86340K->40924K(1042048K), 0.0295842 secs]
... ...
[GC (Allocation Failure)  192868K->147456K(1042048K), 0.0661785 secs]
i=1400000
[GC (Allocation Failure)  199936K->154521K(1042048K), 0.0685919 secs]
[GC (Allocation Failure)  207001K->161585K(1042048K), 0.0707886 secs]
i=1500000
[GC (Allocation Failure)  214065K->168649K(1042048K), 0.0744149 secs]
[GC (Allocation Failure)  221129K->175714K(1042048K), 0.0766862 secs]
i=1600000
[GC (Allocation Failure)  228194K->182778K(1042048K), 0.0802783 secs]
```

String.intern() 250w个String对象：

> 执行命令：
> java -verbose:gc -XX:+PrintGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -
> XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -
> XX:+PrintStringTableStatistics -Xmx1g -Xms1g -Xmn64m StringInternTest

当i=2500000，即往StringTable添加了250w个引用时，kill调这个进程，能够看到`PrintStringTableStatistics`这个参数作用下输出的StringTable相关信息（输出信息中还有SymbolTable，这篇文章不做讨论）：

```
StringTable statistics:
Number of buckets       :     60013 =    480104 bytes, avg   8.000
Number of entries       :   2568786 =  61650864 bytes, avg  24.000
Number of literals      :   2568786 = 287662080 bytes, avg 111.984
Total footprint         :           = 349793048 bytes
Average bucket size     :    42.804
Variance of bucket size :    43.104
Std. dev. of bucket size:     6.565
Maximum bucket size     :        82
```

且这时候的YGC时间达到了0.12s：
i=2500000
[GC (Allocation Failure) 320041K->274625K(1042048K), 0.1268211 secs]
[GC (Allocation Failure) 327105K->281693K(1042048K), 0.1236515 secs]

String.intern() 500w个String对象：

> 执行命令：
> java -verbose:gc -XX:+PrintGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:+PrintStringTableStatistics -Xmx1g -Xms1g -Xmn64m StringInternTest

当i=5000000，即往StringTable添加了500w个引用时，kill调这个进程，输出结果如下：

```
StringTable statistics:
Number of buckets       :     60013 =    480104 bytes, avg   8.000
Number of entries       :   5082093 = 121970232 bytes, avg  24.000
Number of literals      :   5082093 = 569152512 bytes, avg 111.992
Total footprint         :           = 691602848 bytes
Average bucket size     :    84.683
Variance of bucket size :    85.084
Std. dev. of bucket size:     9.224
Maximum bucket size     :       123
```

YGC时间达到了0.24s：
i=5000000
[GC (Allocation Failure) 595600K->550184K(1042048K), 0.2425553 secs]

PrintStringTableStatistics结果解读：

从PrintStringTableStatistics输出信息可以看出StringTable的bucket数量默认为60013，且每个bucket占用8个字节（说明：如果是32位系统，那么每个bucket占用4个字节）；Number of entries即Hashtable的entry数量为2568786，因为我们String.intern( )了250w个不同的String对象；**Average bucket size**表示表示bucket中LinkedList的平均size，**Maximum bucket size** 表示bucket中LinkedList最大的size，Average bucket size越大，说明Hashtable碰撞越严重，由于bucket数量固定为60013，随着StringTable添加的引用越来越多，碰撞越来越严重，YGC时间越来越长。

String.intern() 250w个String对象&-XX:StringTableSize=2500000：

> 执行命令：
> java -verbose:gc -XX:+PrintGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:+PrintStringTableStatistics -Xmx1g -Xms1g -Xmn64m **-XX:StringTableSize=2500000** StringInternTest

当i=2500000，kill调这个进程，输出结果如下：

```
StringTable statistics:
Number of buckets       :   2500000 =  20000000 bytes, avg   8.000
Number of entries       :   2573556 =  61765344 bytes, avg  24.000
Number of literals      :   2573556 = 288196288 bytes, avg 111.984
Total footprint         :           = 369961632 bytes
Average bucket size     :     1.029
Variance of bucket size :     1.028
Std. dev. of bucket size:     1.014
Maximum bucket size     :        10
```

YGC时间从0.12s下降到了0.09s：
i=2500000
[GC (Allocation Failure) 320216K->274800K(1042048K), 0.0890073 secs]
[GC (Allocation Failure) 327280K->281865K(1042048K), 0.0926348 secs]

String.intern() 500w个String对象&-XX:StringTableSize=5000000：

> 执行命令：
> java -verbose:gc -XX:+PrintGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:+PrintStringTableStatistics -Xmx1g -Xms1g -Xmn64m **-XX:StringTableSize=5000000** StringInternTest

当i=5000000，即往StringTable添加了500w个引用时，kill调这个进程，输出结果如下：

```
StringTable statistics:
Number of buckets       :   5000000 =  40000000 bytes, avg   8.000
Number of entries       :   5151776 = 123642624 bytes, avg  24.000
Number of literals      :   5151776 = 576957008 bytes, avg 111.992
Total footprint         :           = 740599632 bytes
Average bucket size     :     1.030
Variance of bucket size :     1.030
Std. dev. of bucket size:     1.015
Maximum bucket size     :         9
```

YGC时间从0.24s降到了0.17s：
i=5000000
[GC (Allocation Failure) 595645K->550229K(1042048K), 0.1685862 secs]
[GC (Allocation Failure) 602709K->557293K(1042048K), 0.1706642 secs]

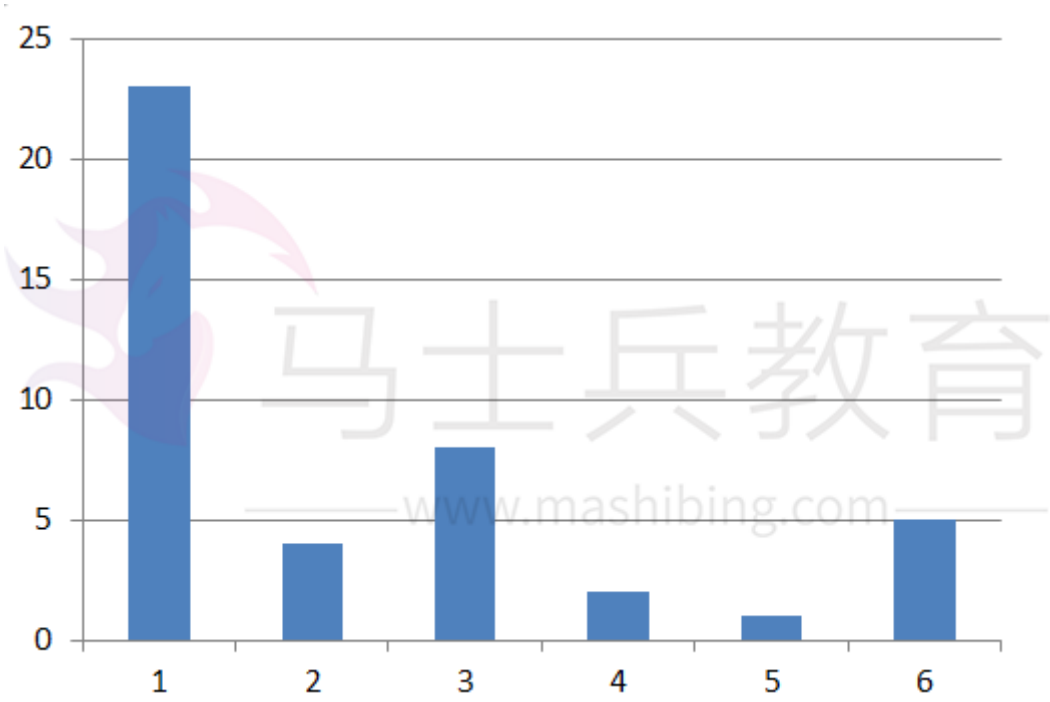
PrintStringTableStatistics&StringTableSize结果解读：

设置StringTableSize一个合适的值，即bucket数量为期望的数量后，碰撞的概率明显降低，由**Average bucket size**和**Maximum bucket size**的值明显小于未配置StringTableSize参数时的值可知，且YGC时间也明显降低。另外, 最好通过BTrace分析是哪里频繁调用String.intern(), 确实String.intern()没有滥用的前提下, 再增大StringTableSize的值。

引申问题

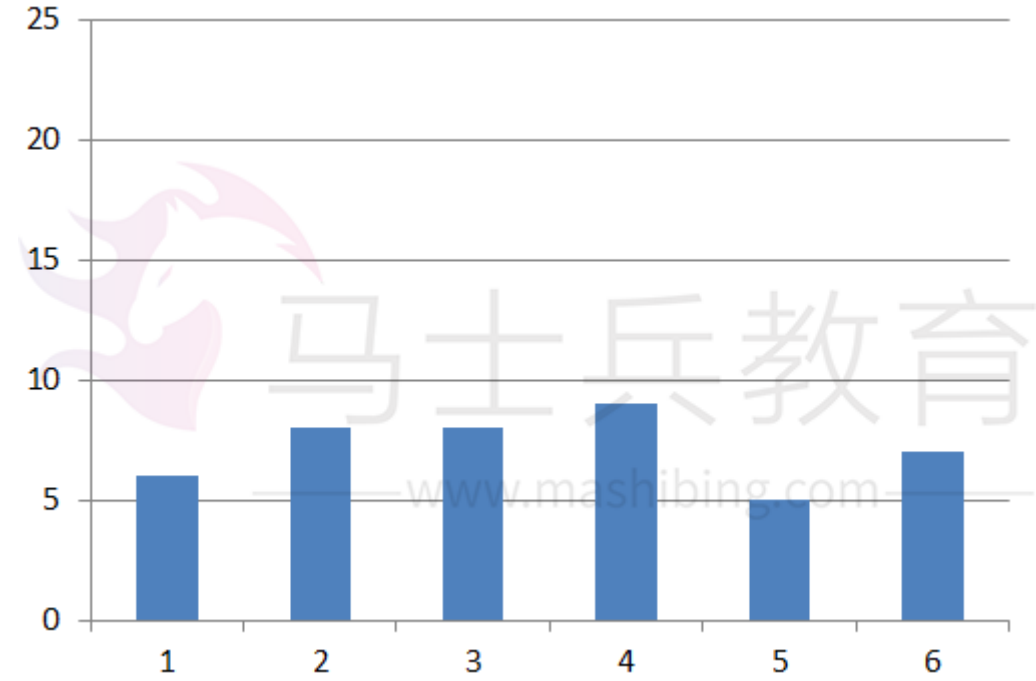
既然StringTable是Hashtable数据结构，那为什么不能自己通过rehash扩大bucket数量来提高性能呢？JVM中StringTable的rehash有点不一样, JVM中StringTable的rehash不会扩大bucket数量，而是在bucket不变的前提下，通过一个新的seed尝试摊平每个bucket中LinkedList的长度（想想也是，**如果StringTable能通过rehash扩大bucket数量，那还要StringTableSize干嘛**），rehash大概是一个如下图所示的过程，rehash前后bucket数量不变，这是重点：

假设reash前数据分布（23,4,8,2,1,5）：



StringTable rehash前.png

reash后可能数据分布（6,8,8,9,5,7）：



StringTable rehash后.png

对应的源码在hashtable.cpp中--第一行代码就是初始化一个新的`_seed`用于后面的hash值计算：

```
// Create a new table and using alternate hash code, populate the new table
// with the existing elements.   This can be used to change the hash code
// and could in the future change the size of the table.

template <class T, MEMFLAGS F> void Hashtable<T, F>::move_to(Hashtable<T, F>*
new_table) {

  // Initialize the global seed for hashing.
  _seed = AltHashing::compute_seed();
  assert(seed() != 0, "shouldn't be zero");

  int saved_entry_count = this->number_of_entries();

  // Iterate through the table and create a new entry for the new table
  for (int i = 0; i < new_table->table_size(); ++i) {
    for (HashtableEntry<T, F>* p = bucket(i); p != NULL; ) {
      HashtableEntry<T, F>* next = p->next();
      T string = p->literal();
      // Use alternate hashing algorithm on the symbol in the first table
      unsigned int hashValue = string->new_hash(seed());
      // Get a new index relative to the new table (can also change size)
      int index = new_table->hash_to_index(hashValue);
      p->set_hash(hashValue);
      // Keep the shared bit in the Hashtable entry to indicate that this entry
      // can't be deleted.   The shared bit is the LSB in the _next field so
      // walking the hashtable past these entries requires
      // BasicHashtableEntry::make_ptr() call.
      bool keep_shared = p->is_shared();
      this->unlink_entry(p);
      new_table->add_entry(index, p);
      if (keep_shared) {
        p->set_shared();
      }
      p = next;
    }
  }
  // give the new table the free list as well
  new_table->copy_freelist(this);
  assert(new_table->number_of_entries() == saved_entry_count, "lost entry on
dictionary copy?");

  // Destroy memory used by the buckets in the hashtable.  The memory
  // for the elements has been used in a new table and is not
  // destroyed.  The memory reuse will benefit resizing the SystemDictionary
  // to avoid a memory allocation spike at safepoint.
  BasicHashtable<F>::free_buckets();
}
```

结论

YGC耗时的问题确实比较难排查，遍历StringTable只是其中一部分，通过PrintStringTableStatistics参数可以了解我们应用的StringTable相关统计信息，且可以通过设置合理的StringTableSize值降低碰撞从而减少YGC时间。另一方面，增大StringTableSize的值有什么影响呢？需要多消耗一点内存，因为每一个bucket需要8个byte（64位系统）。与它带来的YGC性能提升相比，这点内存消耗还是非常值得的。

然而StringTable的统计信息需要在JVM退出时才输出，不得不说是一个遗憾

JVM源码分析之String.intern()导致的YGC不断变长

概述

之所以想写这篇文章，是因为YGC过程对我们来说太过于黑盒，如果对YGC过程不是很熟悉，这类问题基本很难定位，我们就算开了GC日志，也最多能看到类似下面的日志

```
                  [GC (Allocation Failure) [ParNew: 91807K->10240K(92160K),
0.0538384 secs] 91807K->21262K(2086912K), 0.0538680 secs] [Times: user=0.16
sys=0.06, real=0.06 secs]
```

只知道耗了多长时间，但是具体耗在了哪个阶段，是基本看不出来的，所以要么就是靠经验来定位，要么就是对代码相当熟悉，脑袋里过一遍整个过程，看哪个阶段最可能，今天要讲的这个大家可以当做今后排查这类问题的一个经验来使，这个当然不是唯一导致YGC过长的一个原因，但却是最近我帮忙定位碰到的发生相对来说比较多的一个场景

具体的定位是通过在JVM代码里进行了日志埋点确定的，这个问题其实最早的时候，是帮助毕玄毕大师定位到这块的问题，他也在公众号里对这个问题写了相关的一篇文章YGC越来越慢，为什么，大家可以关注下毕大师的公众号`HelloJava`，经常会发一些在公司碰到的诡异问题的排查，相信会让你涨姿势的

Demo

先上一个demo，来描述下问题的情况，代码很简单，就是不断创建UUID，其实就是一个字符串，并将这个字符串调用下intern方法

```
import java.util.UUID;
public class StringTableTest {
    public static void main(String args[]) {
        for (int i = 0; i < 10000000; i++) {
            uuid();
        }
    }
    public static void uuid() {
        UUID.randomUUID().toString().intern();
    }
}
```

我们使用的JVM参数如下：

```
-XX:+UseConcMarkSweepGC -XX:+PrintGCDetails -Xmx2G -Xms2G -Xmn100M
```

这里特意将新生代设置比较小，老生代设置比较大，让代码在执行过程中更容易突出问题来，大量做ygc，期间不做CMS GC，于是我们得到的输出结果类似下面的

```
[GC (Allocation Failure) [ParNew: 81920K->9887K(92160K), 0.0096027 secs] 81920K-
>9887K(2086912K), 0.0096428 secs] [Times: user=0.06 sys=0.01, real=0.01 secs]
```

```
[GC (Allocation Failure) [ParNew: 91807K->10240K(92160K), 0.0538384 secs] 91807K-
>21262K(2086912K), 0.0538680 secs] [Times: user=0.16 sys=0.06, real=0.06 secs]

[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0190535 secs]
103182K->32655K(2086912K), 0.0190965 secs] [Times: user=0.12 sys=0.01, real=0.02
secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0198259 secs]
114575K->44124K(2086912K), 0.0198558 secs] [Times: user=0.13 sys=0.01, real=0.02
secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0213643 secs]
126044K->55592K(2086912K), 0.0213930 secs] [Times: user=0.14 sys=0.01, real=0.02
secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0234291 secs]
137512K->67061K(2086912K), 0.0234625 secs] [Times: user=0.16 sys=0.01, real=0.03
secs]
[GC (Allocation Failure) [ParNew: 92160K->10238K(92160K), 0.0243691 secs]
148981K->78548K(2086912K), 0.0244041 secs] [Times: user=0.15 sys=0.01, real=0.03
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0235310 secs]
160468K->89998K(2086912K), 0.0235587 secs] [Times: user=0.17 sys=0.01, real=0.02
secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0255960 secs]
171918K->101466K(2086912K), 0.0256264 secs] [Times: user=0.18 sys=0.01, real=0.03
secs]
[GC (Allocation Failure) [ParNew: 92160K->10238K(92160K), 0.0287876 secs]
183386K->113770K(2086912K), 0.0288188 secs] [Times: user=0.20 sys=0.01, real=0.03
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0298405 secs]
195690K->125267K(2086912K), 0.0
298823 secs] [Times: user=0.20 sys=0.01, real=0.03 secs]                  [GC
(Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0310182 secs] 207187K-
>136742K(2086912K), 0.0311156 secs] [Times: user=0.22 sys=0.01, real=0.03 secs]

[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0321647 secs]
218662K->148210K(2086912K), 0.0321938 secs] [Times: user=0.22 sys=0.01, real=0.03
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0338090 secs]
230130K->159686K(2086912K), 0.0338446 secs] [Times: user=0.24 sys=0.01, real=0.03
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0326612 secs]
241606K->171159K(2086912K), 0.0326912 secs] [Times: user=0.23 sys=0.01, real=0.03
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0350578 secs]
253079K->182627K(2086912K), 0.0351077 secs] [Times: user=0.26 sys=0.01, real=0.04
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0346946 secs]
264547K->194096K(2086912K), 0.0347274 secs] [Times: user=0.25 sys=0.01, real=0.04
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0384091 secs]
276016K->205567K(2086912K), 0.0384401 secs] [Times: user=0.27 sys=0.01, real=0.04
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0394017 secs]
287487K->217035K(2086912K), 0.0394312 secs] [Times: user=0.29 sys=0.01, real=0.04
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0411447 secs]
298955K->228504K(2086912K), 0.0411748 secs] [Times: user=0.30 sys=0.01, real=0.04
secs]
```

```
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0393449 secs]
310424K->239972K(2086912K), 0.0393743 secs] [Times: user=0.29 sys=0.01, real=0.04
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0444541 secs]
321892K->251441K(2086912K), 0.0444887 secs] [Times: user=0.32 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0449196 secs]
333361K->262910K(2086912K), 0.0449557 secs] [Times: user=0.33 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0497517 secs]
344830K->274382K(2086912K), 0.0497946 secs] [Times: user=0.34 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0475741 secs]
356302K->285851K(2086912K), 0.0476130 secs] [Times: user=0.35 sys=0.01, real=0.04
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0461098 secs]
367771K->297320K(2086912K), 0.0461421 secs] [Times: user=0.34 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0508071 secs]
379240K->308788K(2086912K), 0.0508428 secs] [Times: user=0.38 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0494472 secs]
390708K->320257K(2086912K), 0.0494938 secs] [Times: user=0.36 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0531527 secs]
402177K->331725K(2086912K), 0.0531845 secs] [Times: user=0.39 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0543701 secs]
413645K->343194K(2086912K), 0.0544025 secs] [Times: user=0.41 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0528003 secs]
425114K->354663K(2086912K), 0.0528283 secs] [Times: user=0.39 sys=0.01, real=0.06
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0565080 secs]
436583K->366131K(2086912K), 0.0565394 secs] [Times: user=0.42 sys=0.01, real=0.06
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0597181 secs]
448051K->377600K(2086912K), 0.0597653 secs] [Times: user=0.44 sys=0.01, real=0.06
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0606671 secs]
459520K->389068K(2086912K), 0.0607423 secs] [Times: user=0.46 sys=0.01, real=0.06
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0590389 secs]
470988K->400539K(2086912K), 0.0590679 secs] [Times: user=0.43 sys=0.01, real=0.05
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0600462 secs]
482459K->412008K(2086912K), 0.0600757 secs] [Times: user=0.44 sys=0.01, real=0.06
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0608772 secs]
493928K->423476K(2086912K), 0.0609170 secs] [Times: user=0.45 sys=0.01, real=0.06
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0622107 secs]
505396K->434945K(2086912K), 0.0622391 secs] [Times: user=0.46 sys=0.00, real=0.06
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0626555 secs]
516865K->446413K(2086912K), 0.0626872 secs] [Times: user=0.47 sys=0.01, real=0.07
secs]
```

```
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0647713 secs]
528333K->457882K(2086912K), 0.0648013 secs] [Times: user=0.47 sys=0.00, real=0.07
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0747113 secs]
539802K->469353K(2086912K), 0.0747446 secs] [Times: user=0.51 sys=0.01, real=0.07
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0727498 secs]
551273K->480832K(2086912K), 0.0727899 secs] [Times: user=0.52 sys=0.01, real=0.07
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0734084 secs]
562752K->492300K(2086912K), 0.0734402 secs] [Times: user=0.54 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0766368 secs]
574220K->503769K(2086912K), 0.0766673 secs] [Times: user=0.55 sys=0.01, real=0.07
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0778940 secs]
585689K->515237K(2086912K), 0.0779250 secs] [Times: user=0.56 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0815513 secs]
597157K->526712K(2086912K), 0.0815824 secs] [Times: user=0.57 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0812080 secs]
608632K->538181K(2086912K), 0.0812406 secs] [Times: user=0.58 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0818790 secs]
620101K->549651K(2086912K), 0.0819155 secs] [Times: user=0.60 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0840677 secs]
631571K->561122K(2086912K), 0.0841000 secs] [Times: user=0.61 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0842462 secs]
643042K->572593K(2086912K), 0.0842785 secs] [Times: user=0.61 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0875011 secs]
654513K->584076K(2086912K), 0.0875416 secs] [Times: user=0.62 sys=0.01, real=0.08
secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0887645 secs]
665996K->595532K(2086912K), 0.0887956 secs] [Times: user=0.64 sys=0.01, real=0.09
secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0921844 secs]
677452K->607001K(2086912K), 0.0922153 secs] [Times: user=0.65 sys=0.01, real=0.09
secs]
[GC (Allocation Failure) [ParNew: 92160K->10238K(92160K), 0.0930053 secs]
688921K->618471K(2086912K), 0.0930380 secs] [Times: user=0.67 sys=0.01, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0955379 secs]
700391K->629942K(2086912K), 0.0955873 secs] [Times: user=0.69 sys=0.01, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0919127 secs]
711862K->641411K(2086912K), 0.0919528 secs] [Times: user=0.68 sys=0.01, real=0.09
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0942291 secs]
723331K->652879K(2086912K), 0.0942611 secs] [Times: user=0.70 sys=0.00, real=0.09
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0951904 secs]
734799K->664348K(2086912K), 0.0952265 secs] [Times: user=0.71 sys=0.00, real=0.10
secs]
```

```
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0963585 secs]
746268K->675816K(2086912K), 0.0963909 secs] [Times: user=0.72 sys=0.01, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0969504 secs]
757736K->687285K(2086912K), 0.0969843 secs] [Times: user=0.72 sys=0.01, real=0.09
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0999066 secs]
769205K->698753K(2086912K), 0.0999376 secs] [Times: user=0.75 sys=0.01, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0998371 secs]
780673K->710222K(2086912K), 0.0998835 secs] [Times: user=0.75 sys=0.00, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.1024616 secs]
792142K->721691K(2086912K), 0.1024927 secs] [Times: user=0.77 sys=0.01, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.1015041 secs]
803611K->733159K(2086912K), 0.1015378 secs] [Times: user=0.76 sys=0.01, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.1058214 secs]
815079K->744628K(2086912K), 0.1058532 secs] [Times: user=0.80 sys=0.01, real=0.10
secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.1061273 secs]
826548K->756096K(2086912K), 0.1061620 secs] [Times: user=0.80 sys=0.01, real=0.10
secs]
```

有没有发现YGC不断发生，并且发生的时间不断在增长，从10ms慢慢增长到了100ms，甚至还会继续涨下去

String.intern方法

从上面的demo我们能挖掘到的可能就是intern这个方法了，那我们先来了解下intern方法的实现，这是String提供的一个方法，jvm提供这个方法的目的是希望对于某个同名字符串使用非常多的场景，在jvm里只保留一份，比如我们不断new String("a")，其实在java heap里会有多个String的对象，并且值都是a，如果我们只希望内存里只保留一个a，或者希望我接下来用到的地方都返回同一个a，那就可以用String.intern这个方法了，用法如下

```
String a = "a".intern(); ... String b = a.intern();
```

这样b和a都是指向内存里的同一个String对象，那JVM里到底怎么做到的呢?

我们看到intern这个方法其实是一个native方法，具体对应到JVM里的逻辑是

```
oop StringTable::intern(oop string, TRAPS)
{
    if (string == NULL) return NULL;
    ResourceMark rm(THREAD);
    int length;
    Handle h_string (THREAD, string);
    jchar* chars = java_lang_String::as_unicode_string(string, length);
    oop result = intern(h_string, chars, length, CHECK_NULL);
    return result;
}

oop StringTable::intern(Handle string_or_null, jchar* name,
                        int len, TRAPS) {
    unsigned int hashValue = hash_string(name, len);
```

```
    int index = the_table()->hash_to_index(hashValue);
    oop found_string = the_table()->lookup(index, name, len, hashValue);
    // Found
    if (found_string != NULL) return found_string;

    debug_only(StableMemoryChecker smc(name, len * sizeof(name[0])));
assert(!Universe::heap()->is_in_reserved(name) || GC_locker::is_active(),

     "proposed name of symbol must be stable");
Handle string;
// try to reuse the string if possible
if (!string_or_null.is_null() && (!JavaObjectsInPerm || string_or_null()-
>is_perm())) {
    string = string_or_null;
} else {
    string = java_lang_String::create_tenured_from_unicode(name, len,
CHECK_NULL);
}
    // Grab the StringTable_lock before getting the_table() because it could
    // change at safepoint.  MutexLocker ml(StringTable_lock, THREAD);

    // Otherwise, add to symbol to table
    return the_table()->basic_add(index, string, name, len,
         hashValue, CHECK_NULL);
}
```

也就是说是其实在JVM里存在一个叫做StringTable的数据结构，这个数据结构是一个Hashtable，在我们调用String.intern的时候其实就是先去这个StringTable里查找是否存在一个同名的项，如果存在就直接返回对应的对象，否则就往这个table里插入一项，指向这个String对象，那么再下次通过intern再来访问同名的String对象的时候，就会返回上次插入的这一项指向的String对象

至此大家应该知道其原理了，另外我这里还想说个题外话，记得几年前tomcat里爆发的一个HashMap导致的hash碰撞的问题，这里其实也是一个Hashtable，所以也还是存在类似的风险，不过JVM里提供一个参数专门来控制这个table的size，`-XX:StringTableSize`，这个参数的默认值如下

```
product(uintx, StringTableSize, NOT_LP64(1009) LP64_ONLY(60013),
   "Number of buckets in the interned String table")
```

另外JVM还会根据hash碰撞的情况来决定是否做rehash，比如你从这个StringTable里查找某个字符串是否存在，如果对其对应的桶挨个遍历，超过了100个还是没有找到对应的同名的项，那就会设置一个flag，让下次进入到safepoint的时候做一次rehash动作，尽量减少碰撞的发生，但是当恶化到一定程度的时候，其实也没啥办法啦，因为你的数据量实在太大，桶子数就那么多，那每个桶再怎么均匀也会带着一个很长的链表，所以此时我们通过修改上面的StringTableSize将桶数变大，可能会一定程度上缓解，但是如果是java代码的问题导致泄露，那就只能定位到具体的代码进行改造了。

StringTable为什么会影响YGC

YGC的过程我不打算再这篇文章里细说，因为我希望尽量保持每篇文章的内容不过于臃肿，有机会可以单独写篇文章来介绍，我这里将列出ygc过程里StringTable这块的具体代码

```
    if (!_process_strong_tasks->is_task_claimed(SH_PS_StringTable_oops_do)) {
    if (so & SO_Strings || (!collecting_perm_gen && !JavaObjectsInPerm)) {
StringTable::oops_do(roots);
    }
    if (JavaObjectsInPerm) {
        // Verify the string table contents are in the perm gen
        NOT_PRODUCT(StringTable::oops_do(&assert_is_perm_closure));
    }
}
```

因为YGC过程不涉及到对perm做回收，因此`collecting_perm_gen`是false，而`JavaObjectsInPerm`默认情况下也是false，表示String.intern返回的字符串是不是在perm里分配，如果是false，表示是在heap里分配的，因此StringTable指向的字符串是在heap里分配的，所以ygc过程需要对StringTable做扫描，以保证处于新生代的String代码不会被回收掉

至此大家应该明白了为什么YGC过程会对StringTable扫描

有了这一层意思之后，YGC的时间长短和扫描StringTable有关也可以理解了，设想一下如果StringTable非常庞大，那是不是意味着YGC过程扫描的时间也会变长呢

YGC过程扫描StringTable对CPU影响大吗

这个问题其实是我写这文章的时候突然问自己的一个问题，于是稍微想了下来跟大家解释下，因为大家也可能会问这么个问题

要回答这个问题我首先得问你们的机器到底有多少个核，如果核数很多的话，其实影响不是很大，因为这个扫描的过程是单个GC线程来做的，所以最多消耗一个核，因此看起来对于核数很多的情况，基本不算什么

StringTable什么时候清理

YGC过程不会对StringTable做清理，这也就是我们demo里的情况会让Stringtable越来越大，因为到目前为止还只看到YGC过程，但是在Full GC或者CMS GC过程会对StringTable做清理，具体验证很简单，执行下`jmap -histo:live <pid>`，你将会发现YGC的时候又降下去了

JVM中最大堆大小有没有限制?

JVM调优常用参数配置

堆配置

```
-Xms:初始堆大小
-Xms：最大堆大小
-XX:NewSize=n:设置年轻代大小
-XX:NewRatio=n:设置年轻代和年老代的比值。如：为3表示年轻代和年老代比值为1：3，年轻代占整个年轻代年老代和的1/4
-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如3表示Eden： 3 Survivor：2，一个Survivor区占整个年轻代的1/5
-XX:MaxPermSize=n:设置持久代大小
```

> 说明：
> 1、一般初始堆和最大堆设置一样，因为：现在内存不是什么稀缺的资源，但是如果不一样，从初始堆到最大堆的过程会有一定的性能开销，所以一般设置为初始堆和最大堆一样。64位系统理论上可以设置为无限大，但是一般设置为**4G**,因为如果再大，JVM进行垃圾回收出现的暂停时间会比较长，这样全GC过长，影响JVM对外提供服务，所以不能太大。一般设置为4G。
> 2、-XX:NewRaio和-XX:SurvivorRatio这两个参数，都是设置年轻代和年老代的大小的，设置一个即可，第一是设置年轻代的大小，第二个是设置比值，理论上设置一个既可以满足需求

收集器设置：

```
-XX:+UseSerialGC:设置串行收集器
-XX:+UseParallelGC:设置并行收集器
-XX:+UseParalledlOldGC:设置并行年老代收集器
-XX:+UseConcMarkSweepGC:设置并发收集器
```

垃圾回收统计信息

打印GC回收的过程日志信息

```
-XX:+PrintGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Xloggc:filename
```

并行收集器设置

```
-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数
-XX:MaxGCPauseMillis=n:设置并行收集最大的暂停时间（如果到这个时间了，垃圾回收器依然没有回收完，也会停止回收）
-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为：1/(1+n)
-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况
-XX:ParallelGCThreads=n:设置并发收集器年轻代手机方式为并行收集时，使用的CPU数。并行收集线程数
```

典型配置举例

以下配置主要针对分代收集回收算法而言

堆大小设置

年轻代的设置很关键
JVM中最大堆大小有三方面限制：相关操作系统的数据模型(32bit还是64bit)限制：系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统下，一般限制在1.5G-2G;64位操作系统对内存没有限制。在Windows Server 2003系统，3.5G物理内存，JDK5.0下测试，最大设置为1478m。

典型设置：

```
java-Xmx3550m -Xms3550m-Xmn2g -Xss128k
-Xmx3550m:设置JVM最大可用内存为3550m
-Xms3550m:设置JVM初始内存为3550m，此值可以设置-Xmx相同，以避免每次垃圾回收完成以后JVM重新分配内存
-Xmn2g:设置年轻代大小为2G。整个堆大小=年轻代大小+年老代大小+持久代大小。持久代一般固定为64M,所以增大年轻代后，将会减少年老代大小，此值对系统性能影响比较大，Sun官方推荐配置为整个堆的3/8
-Xss128k:设置每个线程的堆栈大小。JDK5.0以后每个线程栈大小为1M，以前每个线程堆栈大小为256k。根据应用的线程所需要内存大小进行调整。在相同物理内存下，减少这个值能够生成更多的线程。但是操作系统对一个进程内的线程还是有限制的，不能无限生成，经验值在3000-5000左右。
```

```
java -Xmx3550m-Xms3550m-Xss128k-XX:NewRatio=4-XX:SurvivorRatio=4
-XX:MaxPermSize=16m-XX:MaxTenuringThreshold=0

-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代和年老代所占比值为1：4，年轻代占整个堆栈的1/5
-XX:SurvivorRatio=4:设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4,一个Survivor区占整个年轻代的1/6
-XX:MaxPermSize=16m:设置持久代大小为16m
-XX：MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，提高效率，如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间。
```

回收器的选择

JVM给了三种选择：串行收集器，并行收集器，并发收集器，但是串行收集器只适用于小数据量的情况，一般不考虑使用了，所以这里只针对并行收集器和并发收集器。默认情况下，JDK5.0以前是使用的串行收集器，如果想使用其他收集器需要在启动时加入相应的参数，**JDK5.0以后，JVM会根据系统当前的配置进行判断**

吞吐量优先的并行收集器

并行收集器主要以到达一定的吞吐量为目标，适用于后台处理

```
java -Xmx3550m-Xms3550m-Xss128k-XX:+UseParallelGC
-XX:ParallelGCThreads=20
-XX:+UseParallelGC:选择垃圾收集器为并行收集器。次配置仅对年轻代有效。即上述配置下，年轻代使用并行收集，而年老代仍旧使用串行收集。
-XX：PARALLELgcThreads=20:配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相同。
-XX:+UseParallelOldGC:配置年老代来及收集方式为并行收集，JDK6.0支持对年老代并行收集
-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值
-XX:+UseAdaptiveSizePolicy:设置此选项以后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低响应时间或者收集频率等，此值建议使用并行收集器时，一直打开
```

响应时间优先的并发收集器

并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

```
-XX:CMSFullGCsBeforeCompaction=5
-XX:+UseCMSCompactAtFullCollection
-XX:CMSFullGCsBeforeCompaction:由于并发收集器不对内存空间进行压缩、整理、所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理
-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片
```

调优总结：

年轻代大小选择

响应时间优先的应用：尽可能设置大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达Gbit的成都，因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8核CPU以上应用。

年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可能会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考一下数据获得：

1、并发垃圾收集信息

2、持久代并发收集次数

3、传统GC信息

4、花在年轻代和年老代回收上的时间比例减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用

一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期对象，而年老代尽存放长期存活的对象

较小堆引起的碎片问题

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

```
-XX:+UseCMSCompactAtFullCollection:使用并发收集器时，开启对年老代的压缩
-XX:CMSFullGCsBeforeCompaction=0:上面配置开启的情况下，这里设置多少次FullGc后，对年老代进行压缩
```

调优方法

调优工具

Jconsole,jProfile,VisualVM

Jconsole:jdk自带， 功能简单，但是可以再系统有一定负荷的情况下使用，对垃圾回收算法有很详细的跟踪。

JProfiler：商业软件，需要付费，但是功能强大

VisualVM:JDK自带，功能强大，与Jprofiler类似，推荐

如何调优

观察内存释放情况、集合类检查，对象树

上面这些调优工具都提供了强大的功能，但是总的来说一般分为以下几类功能：

> 1、堆的信息查看（年轻代、年老代、持久代分配）
> 2、提供即时的垃圾回收功能呢
> 3、垃圾监控，长时间监控

内存泄露检查

一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（**常见的集合对象引用**）分析，基本都可以找到泄漏点。

持久代沾满处理：
1、-XX:MaxPermSize=16m
2、换JDK比如：JRocket

系统内存被沾满：
一般是因为没有足够的资源产生线程造成的，系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大的一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，出现异常。
分配给Java虚拟机的内存越多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比。同事，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总共生产的线程数。

java程序内存问题的诊断方法：

> 1、jstat可以查看垃圾回收情况：jstat -gcutil pid
> 2、jmap可以将java内存dump出来
> 3、jstack -l 进程id
> 4、eclipse插件MAT可以有效的分析内存占用情况

```
jmap -h
```

查看jmap的命令参数，帮助查看堆信息

```
ss:~ wqp$ jmap -h
Usage:
    jmap [option] <pid>
        (to connect to running process)
    jmap [option] <executable <core>
        (to connect to a core file)
    jmap [option] [server_id@]<remote server IP or hostname>
        (to connect to remote debug server)

where <option> is one of:
    <none>               to print same info as Solaris pmap
    -heap                to print java heap summary
    -histo[:live]        to print histogram of java object heap; if the "live"
                         suboption is specified, only count live objects
    -permstat            to print permanent generation statistics
    -finalizerinfo       to print information on objects awaiting finalization
    -dump:<dump-options> to dump java heap in hprof binary format
                         dump-options:
                           live         dump only live objects; if not specified,
                                        all objects in the heap are dumped.
                           format=b     binary format
                           file=<file>  dump heap to <file>
                         Example: jmap -dump:live,format=b,file=heap.bin <pid>
    -F                   force. Use with -dump:<dump-options> <pid> or -histo
                         to force a heap dump or histogram when <pid> does not
                         respond. The "live" suboption is not supported
                         in this mode.
    -h | -help           to print this help message
    -J<flag>             to pass <flag> directly to the runtime system
ss:~ wqp$
```

最大线程数，一般的机器300-500不错了，淘宝曾经调优过1500，不过不清楚是如何调的。

如何进行JVM调优？有哪些方法？

如何合理的规划一次jvm性能调优

JVM性能调优涉及到方方面面的取舍，往往是牵一发而动全身，需要全盘考虑各方面的影响。但也有一些基础的理论和原则，理解这些理论并遵循这些原则会让你的性能调优任务将会更加轻松。为了更好的理解本篇所介绍的内容。你需要已经了解和遵循以下内容:

> 1、已了解jvm 垃圾收集器
>
> 2、已了解jvm 性能监控常用工具
>
> 3、能够读懂gc日志
>
> 4、确信不为了调优而调优，jvm调优不能解决一切性能问题

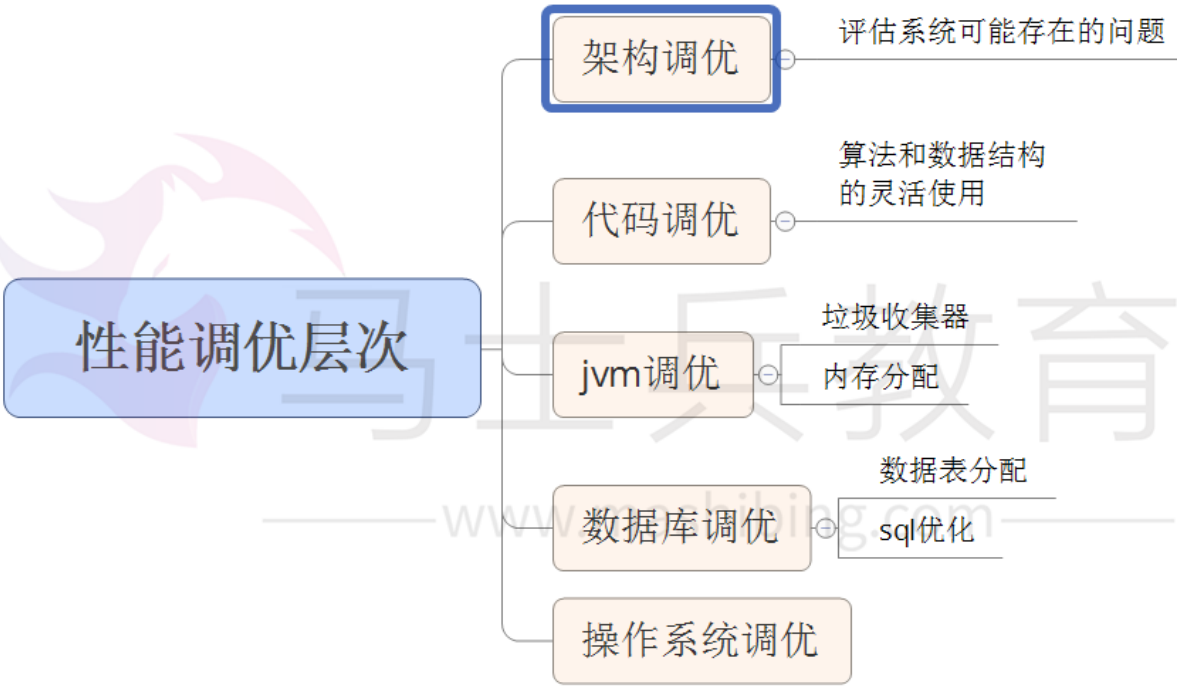
这些内容在之前的两篇文章已经介绍过了，如果有不了解的可以去点击上述连接进行回顾，如果对这些不了解不建议读本篇文章。

本篇文章基于jvm性能调优，结合jvm的各项参数对应用程序调优，主要内容有以下几个方面：

> 1、jvm调优的一般流程
>
> 2、jvm调优所要关注的几个性能指标
>
> 3、jvm调优需要掌握的一些原则
>
> 4、调优策略&示例

一、性能调优的层次

为了提升系统性能，我们需要对系统的各个角度和层次来进行优化，以下是需要优化的几个层次。



从上面我们可以看到，除了jvm调优以外，还有其他几个层面需要来处理，所以针对系统的调优不是只有jvm调优一项，而是需要针对系统来整体调优，才能提升系统的性能。本篇只针对jvm调优来讲解，其他几个方面，后续再介绍。

在进行jvm调优之前，我们假设项目的架构调优和代码调优已经进行过或者是针对当前项目是最优的。这两个是jvm调优的基础，并且架构调优是对系统影响最大的，我们不能指望一个系统架构有缺陷或者代码层次优化没有穷尽的应用，通过jvm调优令其达到一个质的飞跃，这是不可能的。

另外，在调优之前，必须得有明确的性能优化目标， 然后找到其性能瓶颈。之后针对瓶颈的优化，还需要对应用进行压力和基准测试，通过各种监控和统计工具，确认调优后的应用是否已经达到相关目标。

二、jvm调优流程

调优的最终目的都是为了令应用程序使用最小的硬件消耗来承载更大的吞吐。jvm的调优也不例外，jvm调优主要是针对垃圾收集器的收集性能优化，令运行在虚拟机上的应用能够使用更少的内存以及延迟获取更大的吞吐量。当然这里的最少是最优的选择，而不是越少越好。

1、性能定义

要查找和评估器性能瓶颈，首先要知道性能定义，对于jvm调优来说，我们需要知道以下三个定义属性，依作为评估基础:

> - 吞吐量：重要指标之一，是指不考虑垃圾收集引起的停顿时间或内存消耗，垃圾收集器能支撑应用达到的最高性能指标。
> - 延迟：其度量标准是缩短由于垃圾啊收集引起的停顿时间或者完全消除因垃圾收集所引起的停顿，避免应用运行时发生抖动。
> - 内存占用：垃圾收集器流畅运行所需要 的内存数量。

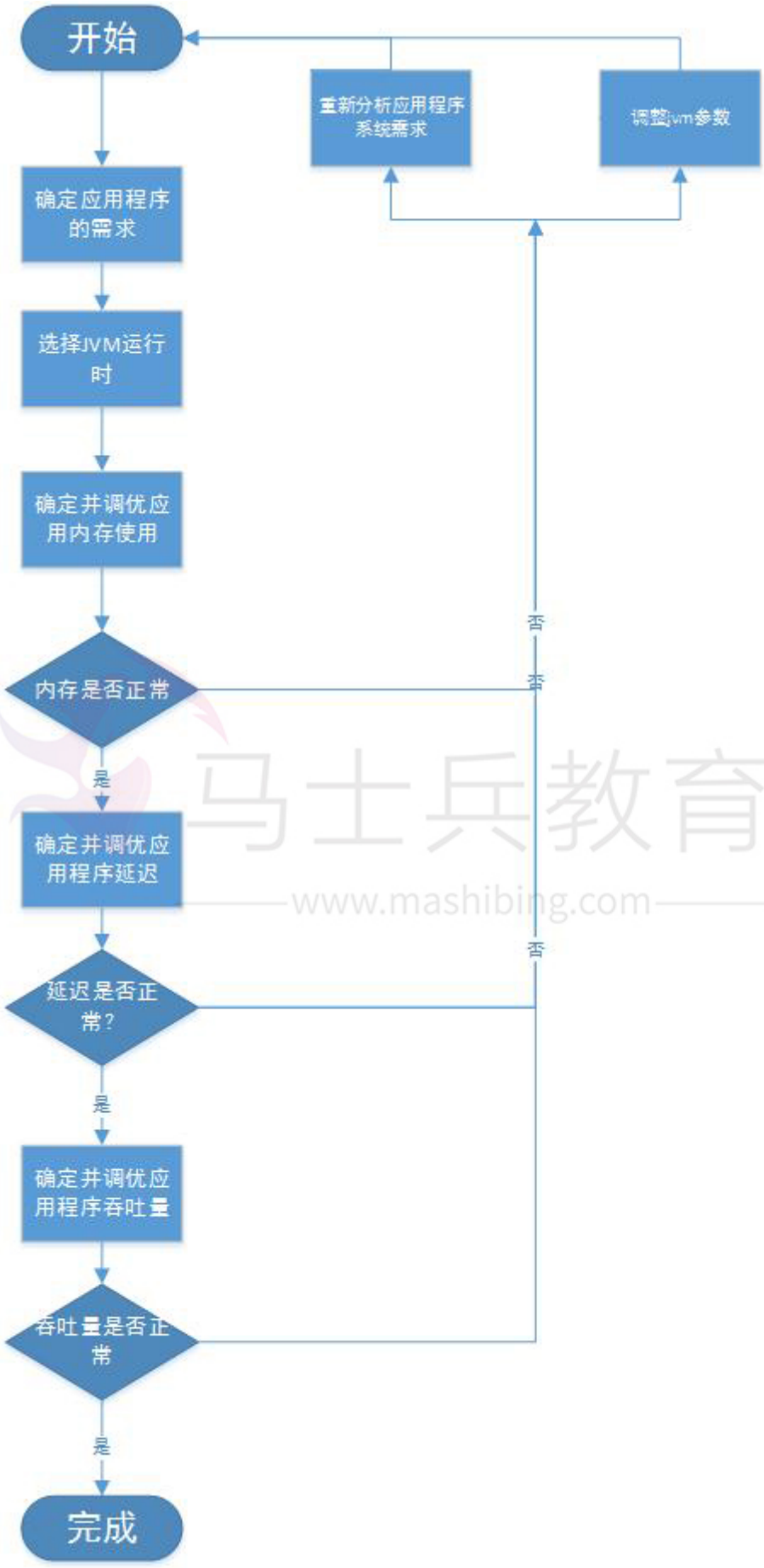
这三个属性中，其中一个任何一个属性性能的提高，几乎都是以另外一个或者两个属性性能的损失作代价，不可兼得，具体某一个属性或者两个属性的性能对应用来说比较重要，要基于应用的业务需求来确定。

2、性能调优原则

在调优过程中，我们应该谨记以下3个原则，以便帮助我们更轻松的完成垃圾收集的调优，从而达到应用程序的性能要求。

> 1. MinorGC回收原则： 每次minor GC 都要尽可能多的收集垃圾对象。以减少应用程序发生Full GC的频率。
> 2. GC内存最大化原则：处理吞吐量和延迟问题时候，垃圾处理器能使用的内存越大，垃圾收集的效果越好，应用程序也会越来越流畅。
> 3. GC调优3选2原则: 在性能属性里面，吞吐量、延迟、内存占用，我们只能选择其中两个进行调优，不可三者兼得。

3、性能调优流程



马士兵教育
www.mashibing.com

以上就是对应用程序进行jvm调优的基本流程，我们可以看到，jvm调优是根据性能测试结果不断优化配置而多次迭代的过程。在达到每一个系统需求指标之前，之前的每个步骤都有可能经历多次迭代。有时候为了达到某一方面的指标，有可能需要对之前的参数进行多次调整，进而需要把之前的所有步骤重新测试一遍。

另外调优一般是从满足程序的内存使用需求开始的，之后是时间延迟的要求，最后才是吞吐量的要求，要基于这个步骤来不断优化，每一个步骤都是进行下一步的基础，不可逆行之。以下我们针对每个步骤进行详细的示例讲解。

在JVM的运行模式方面，我们直接选择server模式，这也是jdk1.6以后官方推荐的模式。

在垃圾收集器方面，我们直接采用了jdk1.6-1.8 中默认的parallel收集器（新生代采用parallelGC,老生代采用parallelOldGC）。

三、确定内存占用

在确定内存占用之前，我们需要知道两个知识点：

> 1. 应用程序的运行阶段
> 2. jvm内存分配

1、运行阶段

应用程序的运行阶段，我可以划分为以下三个阶段:

> 1、初始化阶段 : jvm加载应用程序，初始化应用程序的主要模块和数据。
>
> 2、稳定阶段:应用在此时运行了大多数时间，经历过压力测试的之后，各项性能参数呈稳定状态。核心函数被执行，已经被jit编译预热过。
>
> 3、总结阶段:最后的总结阶段，进行一些基准测试，生成响应的策报告。这个阶段我们可以不关注。

确定内存占用以及活跃数据的大小，我们应该是在程序的稳定阶段来进行确定，而不是在项目起初阶段来进行确定，如何确定，我们先看以下jvm的内存分配。

2、jvm内存分配&参数



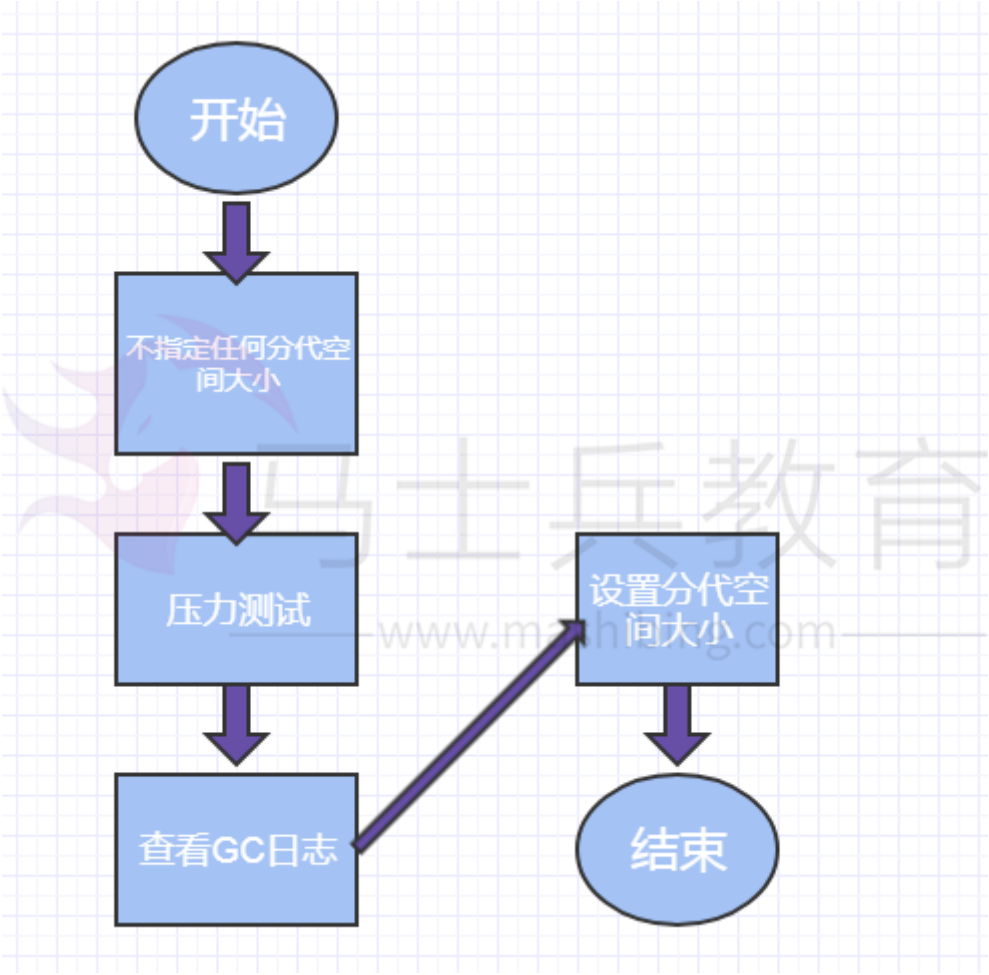
jvm堆中主要的空间，就是以上新生代、老生代、永久代组成,整个堆大小=新生代大小 + 老生代大小 + 永久代大小。 具体的对象提升方式，这里不再过多介绍了，我们看下一些jvm命令参数，对堆大小的指定。如果不采用以下参数进行指定的话，虚拟机会自动选择合适的值，同时也会基于系统的开销自动调整。

分代	参数	描述
堆大小	-Xms	初始堆大小，默认为物理内存的1/64(<1GB)
	-Xmx	最大堆大小，默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制
新生代	-XX:NewSize	新生代空间大小初始值
	-XX:MaxNewSize	新生代空间大小最大值
	-Xmn	新生代空间大小，此处的大小是(eden+2 survivor space)
永久代	-XX:PermSize	永久代空间的初始值&最小值
	-XX：MaxPermSize	永久代空间的最大值
老年代	老年代的空间大小会根据新生代的大小隐式设定	
	初始值=-Xmx减去-XX:NewSize的值	
	最小值=-Xmx值减去-XX:MaxNewSize的值	

在设置的时候，如果关注性能开销的话，应尽量把永久代的初始值与最大值设置为同一值，因为永久代的大小调整需要进行FullGC 才能实现。

3、计算活跃数据大小

计算活跃数据大小应该遵循以下流程:



如前所述，活跃数据应该是基于应用程序稳定阶段时，观察长期存活与对象在java堆中占用的空间大小。

计算活跃数据时应该确保以下条件发生：

> 1.测试时，启动参数采用jvm默认参数，不人为设置。
>
> 2.确保Full GC 发生时，应用程序正处于稳定阶段。

采用jvm默认参数启动，是为了观察应用程序在稳定阶段的所需要的内存使用。

如何才算稳定阶段?

一定得需要产生足够的压力，找到应用程序和生产环境高峰符合状态类似的负荷，在此之后达到峰值之后，保持一个稳定的状态，才算是一个稳定阶段。所以要达到稳定阶段，压力测试是必不可少的，具体如何如何对应用压力测试，本篇不过多说明,后期会有专门介绍的篇幅。

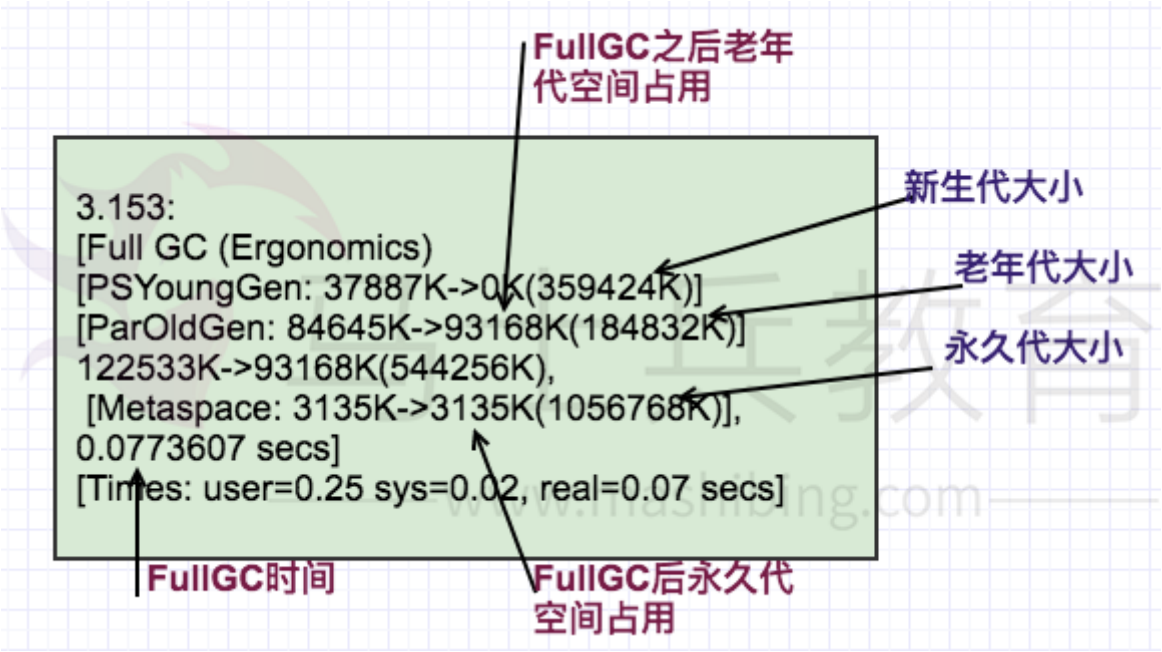
在确定了应用出于稳定阶段的时候，要注意观察应用的GC日志，特别是Full GC 日志。

> GC日志指令: -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:
>
> GC日志是收集调优所需信息的最好途径，即便是在生产环境，也可以开启GC日志来定位问题，开启GC日志对性能的影响极小，却可以提供丰富数据。

必须得有FullGC 日志，如果没有的话，可以采用监控工具强制调用一次，或者采用以下命令，亦可以触发

> jmap -histo:live pid

在稳定阶段触发了FullGC我们一般会拿到如下信息:



从以上gc日志中，我们大概可以分析到，在发生fullGC之时，整个应用的堆占用以及GC时间，当然了，为了更加精确，应该多收集几次，获取一个平均值。或者是采用耗时最长的一次FullGC来进行估算。

在上图中，fullGC之后，老年代空间占用在93168kb（约93MB），我们以此定为老年代空间的活跃数据。

其他堆空间的分配，基于以下规则来进行。

空间	命令参数	建议扩大倍数
java heap	-Xms和-Xmx	3-4倍FullGC后的老年代空间占用
永久代	-XX:PermSize-XX:MaxPermSize	1.2-1.5倍FullGc后的永久带空间占用
新生代	-Xmn	1-1.5倍FullGC之后的老年代空间占用
老年代		2-3倍FullGC后的老年代空间占用

基于以上规则和上图中的FullGC信息，我们现在可以规划的该应用堆空间为：

> java 堆空间: 373Mb (=老年代空间93168kb*4)
>
> 新生代空间:140Mb(=老年代空间93168kb*1.5)
>
> 永久代空间:5Mb(=永久代空间3135kb*1.5)
>
> 老年代空间: 233Mb=堆空间-新生代看空间=373Mb-140Mb

对应的应用启动参数应该为:

```
java -Xms373m -Xmx373m -Xmn140m -XX:PermSize=5m -XX:MaxPermSize=5m
```

四、延迟调优

在确定了应用程序的活跃数据大小之后，我们需要再进行延迟性调优，因为对于此时堆内存大小，延迟性需求无法达到应用的需要，需要基于应用的情况来进行调试。

在这一步进行期间，我们可能会再次优化堆大小的配置，评估GC的持续时间和频率、以及是否需要切换到不同的垃圾收集器上。

1、系统延迟需求

在调优之前，我们需要知道系统的延迟需求是那些，以及对应的延迟可调优指标是那些。

> - 应用程序可接受的平均停滞时间: 此时间与测量的Minor GC持续时间进行比较。
> - 可接受的Minor GC频率：Minor GC的频率与可容忍的值进行比较。
> - 可接受的最大停顿时间: 最大停顿时间与最差情况下FullGC的持续时间进行比较。
> - 可接受的最大停顿发生的频率：基本就是FullGC的频率。

以上中，平均停滞时间和最大停顿时间，对用户体验最为重要，可以多关注。

基于以上的要求，我们需要统计以下数据:

> - MinorGC的持续时间；
> - 统计MinorGC的次数；
> - FullGC的最差持续时间；
> - 最差情况下，FullGC的频率；

2、优化新生代的大小

```
0.388: [GC (Allocation Failure) [PSYoungGen: 40960K->4953K(47616K)] 40960K->4961K(157184K),
0.0040914 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]
0.505: [GC (Allocation Failure) [PSYoungGen: 45913K->6645K(88576K)] 45921K->8238K(198144K),
0.0196029 secs] [Times: user=0.01 sys=0.01, real=0.02 secs]
0.775: [GC (Allocation Failure) [PSYoungGen: 88565K->6653K(88576K)] 90158K->18846K(198144K),
0.0126382 secs] [Times: user=0.01 sys=0.02, real=0.01 secs]
1.025: [GC (Allocation Failure) [PSYoungGen: 88573K->6641K(170496K)] 100766K->28993K(280064K),
0.0144236 secs] [Times: user=0.02 sys=0.01, real=0.01 secs]
1.524: [GC (Allocation Failure)
[PSYoungGen: 170481K->6653K(170496K)] 192833K->50893K(280064K), 0.0220529 secs] [Times:
user=0.03 sys=0.02, real=0.02 secs]
2.081: [GC (Allocation Failure) [PSYoungGen: 170481K->30765K(353280K)] 214720K-
>75004K(462848K), 0.0172621 secs] [Times: user=0.03 sys=0.04, real=0.02 secs]
3.113: [GC (Allocation Failure) [PSYoungGen: 352301K->37887K(359424K)] 396540K-
>122533K(468992K), 0.0400571 secs] [Times: user=0.06 sys=0.08, real=0.04 secs]
3.153: [Full GC (Ergonomics) [PSYoungGen: 37887K->0K(359424K)] [ParOldGen: 84645K-
>93168K(184832K)] 122533K->93168K(544256K), [Metaspace: 3135K->3135K(1056768K)], 0.0773607
secs] [Times: user=0.25 sys=0.02, real=0.07 secs]
4.237: [GC (Allocation Failure) [PSYoungGen: 321536K->48606K(576512K)] 414704K-
>141775K(761344K), 0.0273516 secs] [Times: user=0.04 sys=0.05, real=0.03 secs]
5.927: [GC (Allocation Failure) [PSYoungGen: 576478K->70648K(686592K)] 669647K-
>245925K(871424K), 0.0780328 secs] [Times: user=0.10 sys=0.18, real=0.08 secs]
6.005: [Full GC (Ergonomics) [PSYoungGen: 70648K->0K(686592K)] [ParOldGen: 175277K-
>180379K(320000K)] 245925K->180379K(1006592K), [Metaspace: 3135K->3135K(1056768K)], 0.1059144
secs] [Times: user=0.39 sys=0.01, real=0.10 secs]
```

比如如上的gc日志中，我们可以看到Minor GC的平均持续时间=0.069秒，MinorGC 的频率为0.389秒一次。

> 如果，我们系统的设置的平均停滞时间为50ms，当前的69ms明显是太长了，就需要调整。

> 我们知道新生代空间越大，Minor GC的GC时间越长，频率越低。
>
> 如果想减少其持续时长，就需要减少其空间大小。
>
> 如果想减小其频率，就需要加大其空间大小。

为了降低改变新生代的大小对其他区域的最小影响。**在改变新生代空间大小的时候，尽量保持老年代空间的大小。**

比如此次减少了新生代空间10%的大小，应该保持老年代和持代的大小不变化，第一步调优后的参数如下变化:

```
java -Xms359m -Xmx359m -Xmn126m -XX:PermSize=5m -XX:MaxPermSize=5m

新生代的大小有140m变为126，堆大小顺应变化，此时老年代是没有变化的。
```

3、优化老年代的大小

同上一步一样，在优化之前，也需要采集gc日志的数据。此次我们关注的是FullGC的持续时间和频率。

```
3.153: [Full GC (Ergonomics) [PSYoungGen: 37887K->0K(359424K)] [ParOldGen: 84645K-
>93168K(184832K)] 122533K->93168K(544256K), [Metaspace: 3135K->3135K(1056768K)], 0.0773607
secs] [Times: user=0.25 sys=0.02, real=0.07 secs]
6.005: [Full GC (Ergonomics) [PSYoungGen: 70648K->0K(686592K)] [ParOldGen: 175277K-
>180379K(320000K)] 245925K->180379K(1006592K), [Metaspace: 3135K->3135K(1056768K)], 0.1059144
secs] [Times: user=0.39 sys=0.01, real=0.10 secs]
12.147: [Full GC (Ergonomics) [PSYoungGen: 85025K->0K(741888K)] [ParOldGen: 306915K-
>258866K(443392K)] 391941K->258866K(1185280K), [Metaspace: 3140K->3140K(1056768K)], 0.1950417
secs] [Times: user=0.68 sys=0.01, real=0.19 secs]
18.130: [Full GC (Ergonomics) [PSYoungGen: 54640K->0K(723456K)] [ParOldGen: 392363K-
>238559K(480768K)] 447004K->238559K(1204224K), [Metaspace: 3140K->3140K(1056768K)], 0.1633162
secs] [Times: user=0.56 sys=0.00, real=0.17 secs]
27.020: [Full GC (Ergonomics) [PSYoungGen: 117739K->0K(678400K)] [ParOldGen: 411151K-
>264441K(539136K)] 528890K->264441K(1217536K), [Metaspace: 3140K->3140K(1056768K)], 0.1842140
secs] [Times: user=0.63 sys=0.00, real=0.19 secs]
36.520: [Full GC (Ergonomics) [PSYoungGen: 62798K->0K(681984K)] [ParOldGen: 512757K-
>219019K(527872K)] 575555K->219019K(1209856K), [Metaspace: 3140K->3140K(1056768K)], 0.1652638
secs] [Times: user=0.55 sys=0.01, real=0.17 secs]
44.678: [Full GC (Ergonomics) [PSYoungGen: 70581K->0K(692224K)] [ParOldGen: 423465K-
>221793K(541184K)] 494046K->221793K(1233408K), [Metaspace: 3140K->3140K(1056768K)], 0.1546943
secs] [Times: user=0.58 sys=0.00, real=0.15 secs]
```

上图中，我们可以看到

```
FullGC 平均频率 =5.8s

FullGC 平均持续时间=0.14s

(以上为了测试，真实项目的fullGC 没有这么快)
```

如果没有FullGC的日志，有办法可以评估么?

> 我们可以通过对象提升率进行计算。

对象提升率

比如上述中启动参数中，我们的老年代大小=233Mb。

那么需要多久才能填满老年代中这233Mb的空闲空间取决于新生代到老年代的提升率。

> 每次提升老年代占用量=每次MinorGC 之后 java堆占用情况 减去 MinorGC后新生代的空间占用
>
> 对象提升率=平均值（每次提升老年代占用量) 除以 老年代空间

有了对象提升率，我们就可以算出填充满老年代空间需要多少次minorGC，大概一次fullGC的时间就可以计算出来了。

比如:

```
0.514: [GC (Allocation Failure) [PSYoungGen: 107520K->13732K(125440K)] 107520K-
>13740K(365056K), 0.0105175 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]
0.844: [GC (Allocation Failure) [PSYoungGen: 121252K->17905K(125440K)] 121260K-
>22394K(365056K), 0.0167029 secs] [Times: user=0.02 sys=0.02, real=0.02 secs]
1.171: [GC (Allocation Failure) [PSYoungGen: 125425K->17917K(125440K)] 129914K-
>34739K(365056K), 0.0146356 secs] [Times: user=0.03 sys=0.01, real=0.01 secs]
1.542: [GC (Allocation Failure) [PSYoungGen: 125437K->17913K(125440K)] 142259K-
>48143K(365056K), 0.0147920 secs] [Times: user=0.03 sys=0.01, real=0.02 secs]
1.874: [GC (Allocation Failure) [PSYoungGen: 125404K->17917K(125440K)] 155634K-
>62112K(365056K), 0.0173626 secs] [Times: user=0.03 sys=0.01, real=0.02 secs]
2.233: [GC (Allocation Failure) [PSYoungGen: 125379K->17909K(78336K)] 169574K-
>74568K(317952K), 0.0284603 secs] [Times: user=0.03 sys=0.01, real=0.03 secs]
2.445: [GC (Allocation Failure) [PSYoungGen: 78325K->15932K(76800K)] 134984K->81904K(316416K),
0.0121190 secs] [Times: user=0.02 sys=0.01, real=0.02 secs]
2.646: [GC (Allocation Failure) [PSYoungGen: 76330K->19826K(102400K)] 142302K-
>87774K(342016K), 0.0082700 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
2.855: [GC (Allocation Failure) [PSYoungGen: 80224K->18545K(101376K)] 148172K-
>93848K(340992K), 0.0095098 secs] [Times: user=0.03 sys=0.01, real=0.01 secs]
3.057: [GC (Allocation Failure) [PSYoungGen: 78930K->10683K(101376K)] 154233K-
>102604K(340992K), 0.0129099 secs] [Times: user=0.03 sys=0.01, real=0.01 secs]
3.261: [GC (Allocation Failure) [PSYoungGen: 72094K->5937K(67584K)] 164014K->109467K(307200K),
0.0081963 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]
3.461: [GC (Allocation Failure) [PSYoungGen: 67333K->6766K(101376K)] 170863K-
>116353K(340992K), 0.0062545 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
```

上图中:

```
第一次minor GC 之后，老年代空间:13740kb - 13732kb =8kb

第二次minor GC 之后，老年代空间:22394kb - 17905kb =4489kb

第三次minor GC 之后，老年代空间:34739kb - 17917kb =16822kb

第四次minor GC 之后，老年代空间:48143kb - 17913kb =30230kb

第五次minor GC 之后，老年代空间:62112kb - 17917kb =44195kb
```

老年代每次minorGC提升率

```
4481kb 第二次和第一次minorGC之间

12333kb 第3次和第2次minorGC之间

13408kb 第4次和第3次minorGC之间

13965kb 第5次和第4次minorGC之间
```

我们可以测算出:

```
每次minorGC 的平均提升为12211kb,约为12Mb

上图中，平均minorGC的频率为 213ms/次

提升率=12211kb/213ms=57kb/ms

老年代空间233Mb ,占满大概需要233*1024/57=4185ms 约为4.185s。
```

FullGC的预期最差频率时长可以通过以上两种方式估算出来，可以调整老年代的大小来调整FullGC的频率，当然了，如果FullGC持续时间过长，无法达到应用程序的最差延迟要求，就需要切换垃圾处理器了。具体如何切换，下篇再讲，比如切换为CMS，针对CMS的调优方式又有会细微的差别。

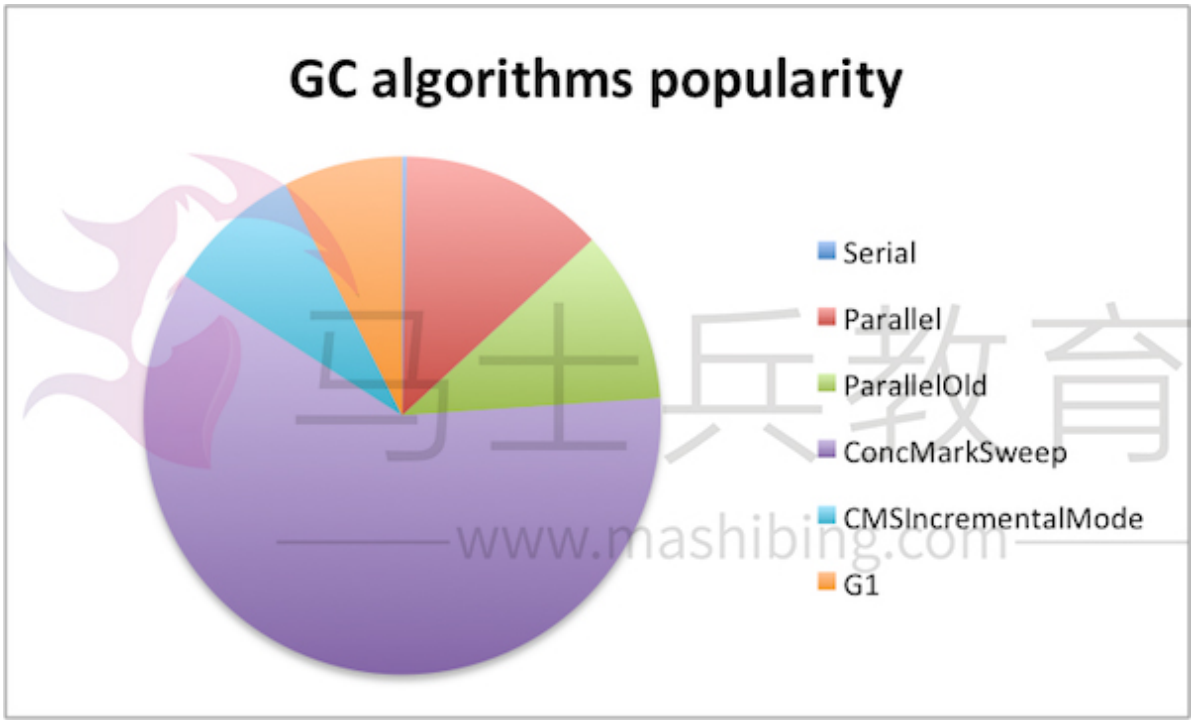
五、吞吐量调优

经过上述漫长 调优过程，最终来到了调优的最后一步，这一步对上述的结果进行吞吐量测试，并进行微调。

吞吐量调优主要是基于应用程序的吞吐量要求而来的，应用程序应该有一个综合的吞吐指标，这个指标基于真个应用的需求和测试而衍生出来的。当有应用程序的吞吐量达到或者超过预期的吞吐目标，整个调优过程就可以圆满结束了。

如果出现调优后依然无法达到应用程序的吞吐目标，需要重新回顾吞吐要求，评估当前吞吐量和目标差距是否巨大，如果在20%左右，可以修改参数，加大内存，再次从头调试，如果巨大就需要从整个应用层面来考虑，设计以及目标是否一致了，重新评估吞吐目标。

对于垃圾收集器来说，提升吞吐量的性能调优的目标就是就是尽可能避免或者很少发生FullGC 或者Stop-The-World压缩式垃圾收集（CMS），因为这两种方式都会造成应用程序吞吐降低。尽量在MinorGC 阶段回收更多的对象，避免对象提升过快到老年代。



马士兵教育
www.mashibing.com

六、最后

据Plumbr公司对特定垃圾收集器使用情况进行了一次调查研究，研究数据使用了84936个案例。在明确指定垃圾收集器的13%的案例中，并发收集器（CMS）使用次数最多；但大多数案例没有选择最佳垃圾收集器。这个比例占用在87%左右。

JVM调优是一个系统而又复杂的工作，目前jvm下的自动调整已经做的比较优秀，基本的一些初始参数都可以保证一般的应用跑的比较稳定了，对部分团队来说，程序性能可能优先级不高，默认垃圾收集器已经够用了。调优要基于自己的情况而来。

内存模型以及分区，需要详细到每个区放什么。

JVM 分为堆区和栈区，还有方法区，初始化的对象放在堆里面，引用放在栈里面，

class 类信息常量池（static 常量和 static 变量）等放在方法区new:

· 方法区：主要是存储类信息，常量池（static 常量和 static 变量），编译后的代码（字

节码）等数据

· 堆：初始化的对象，成员变量 （那种非 static 的变量），所有的对象实例和数组都要

在堆上分配

· 栈：栈的结构是栈帧组成的，调用一个方法就压入一帧，帧上面存储局部变量表，操

作数栈，方法出口等信息，局部变量表存放的是 8 大基础类型加上一个应用类型，所以还是一个指向地址的指针

· 本地方法栈：主要为 Native 方法服务

· 程序计数器：记录当前线程执行的行号

堆里面的分区：Eden，survival （from+ to），老年代，各自的特点。

堆里面分为新生代和老生代（java8 取消了永久代，采用了 Metaspace），新生代包含 Eden+Survivor 区，survivor 区里面分为 from 和 to 区，内存回收时，如果用的是复制算法，从 from 复制到 to，当经过一次或者多次 GC 之后，存活下来的对象会被移动到老年区，当 JVM 内存不够用的时候，会触发 Full GC，清理 JVM 老年区

当新生区满了之后会触发 YGC,先把存活的对象放到其中一个 Survice

区，然后进行垃圾清理。因为如果仅仅清理需要删除的对象，这样会导致内存碎

片，因此一般会把 Eden 进行完全的清理，然后整理内存。那么下次 GC 的时候，就会使用下一个 Survive，这样循环使用。如果有特别大的对象，新生代放不下，就会使用老年代的担保，直接放到老年代里面。因为 JVM 认为，一般大对象的存活时间一般比较久远。

对象创建方法，对象的内存分配，对象的访问定位。

new 一个对象

GC 的两种判定方法：

引用计数法：指的是如果某个地方引用了这个对象就+1，如果失效了就-1，当为 0 就会回收但是 JVM 没有用这种方式，因为无法判定相互循环引用（A 引用 B,B 引用 A）的情况

引用链法： 通过一种 GC ROOT 的对象（方法区中静态变量引用的对象等-static 变量）来判断，如果有一条链能够到达 GC ROOT 就说明，不能到达 GC ROOT 就说明可以回收

SafePoint 是什么

比如 GC 的时候必须要等到 Java 线程都进入到 safepoint 的时候 VMThread 才能开始

执行 GC，

1. 循环的末尾 (防止大循环的时候一直不进入 safepoint，而其他线程在等待它进入

safepoint)

2. 方法返回前
3. 调用方法的 call 之后
4. 抛出异常的位置

GC 的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路?

先标记，标记完毕之后再清除，效率不高，会产生碎片

复制算法：分为 8：1 的 Eden 区和 survivor 区，就是上面谈到的 YGC

标记整理：标记完毕之后，让所有存活的对象向一端移动

GC 收集器有哪些？CMS 收集器与 G1 收集器的特点。

并行收集器：串行收集器使用一个单独的线程进行收集，GC 时服务有停顿时间串行收集器：次要回收中使用多线程来执行

CMS 收集器是基于**“标记—清除”**算法实现的，经过多次标记才会被清除

G1 从**整体来看是基于“标记—整理”**算法实现的收集器，从**局部（两个 Region 之间）**

上来看是基于“复制”算法实现的

Minor GC 与 Full GC 分别在什么时候发生?

新生代内存不够用时候发生 MGC 也叫 YGC，JVM 内存不够的时候发生 FGC

几种常用的内存调试工具：jmap、jstack、jconsole、jhat jstack

可以看当前栈的情况，jmap 查看内存，jhat 进行 dump 堆的信息mat（eclipse 的也要了解一下）

类加载的几个过程：

加载、验证、准备、解析、初始化。然后是使用和卸载了

通过全限定名来加载生成 class 对象到内存中，然后进行验证这个 class 文件，包括文件格式校验、元数据验证，字节码校验等。准备是对这个对象分配内存。解析是将符号引用转化为直接引用（指针引用），初始化就是开始执行构造器的代码

JVM 内存分哪几个区，每个区的作用是什么?

java 虚拟机主要分为以下一个区:**方法区:**

1. 有时候也成为**永久代**，在该区内很少发生垃圾回收，但是并不代表不发生 GC，在这里进行的 GC 主要是对方法区里的常量池和对类型的卸载
2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后

的代码等数据。

3. 该区域是被线程共享的。
4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

虚拟机栈:

1. 虚拟机栈也就是我们平常所称的**栈内存**,它为 java 方法服务，每个方法在执行的时候都

会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。

2. 虚拟机栈是线程私有的，它的生命周期与线程相同。
3. 局部变量表里存储的是基本数据类型、returnAddress 类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定
4. 操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
5. 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

本地方法栈

本地方法栈和虚拟机栈类似，只不过本地方法栈为 Native 方法服务。

堆

java 堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

程序计数器

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个 java 虚拟机规范没有规定任何 OOM 情况的区域。

如和判断一个对象是否存活?(或者 GC 对象的判定方法)

判断一个对象是否存活有两种方法:

1. **引用计数法**

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象 A 引用对象 B，对象B 又引用者对象 A，那么此时 A,B 对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2. **可达性算法**(引用链法)

该算法的思想是：从一个被称为 **GC Roots** 的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。

在 java 中可以作为 GC Roots 的对象有以下几种:

· 虚拟机栈中引用的对象

· 方法区类静态属性引用的对象

· 方法区常量池引用的对象

· 本地方法栈 JNI 引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比**不一定会被回收。**当一个对象不可达 GC Root 时，这个对象并

不会立马被回收，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记

如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法或者已被虚拟机调用过，那么就认为是没必要的。

如果该对象有必要执行 finalize()方法，那么这个对象将会放在一个称为 F-Queue 的对队列中，虚拟机会触发一个 Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize()执行缓慢或者发生了死锁，那么就会造成 F- Queue 队列一直等待，造成了内存回收系统的崩溃。GC 对处于 F-Queue 中的对象进行

第二次被标记，这时，该对象将被移除”即将回收”集合，等待回收。

简述 java 垃圾回收机制?

在 java 中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM 中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

java 中垃圾收集的方法有哪些?

1. **标记-清除:**

这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次 GC 动作

2. 复制算法:**

为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，

内存的代价太高，每次基本上都要浪费一般的内存。

于是将该算法进行了改进，内存区域不再是按照 1：1 去划分，而是将内存划分为8:1:1 三部分，较大那份内存交 Eden 区，其余是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区，若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这些对象通过分配担保机制复制到老年代中。(java 堆又分为新生代和老年代)

3. **标记-整理**

该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了

4. **分代收集**

现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用**复制**算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用**标记-整理** 或者 **标记-清除。**

java 内存模型

java 内存模型(JMM)是线程间通信的控制机制.JMM 定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM 的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java 内存模型的抽象示意图如下：

从上图来看，线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：

1. 首先，线程 A 把本地内存 A 中更新过的共享变量刷新到主内存中去。
2. 然后，线程 B 到主内存中去读取线程 A 之前已更新过的共享变量。

java 类加载过程?

java 类加载需要经历一下 7 个过程：

加载

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名获取该类的二进制流。
2. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
3. 在内存中生成该类的 Class 对象，作为该类的数据访问入口。

验证

验证的目的是为了确保 Class 文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成

以下四钟验证:

1. 文件格式验证：验证字节流是否符合 Class 文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.
2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

public static int value=123;//***在准备阶段value初始值为0。在初始化阶段才会变为123**。

解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。

简述 java 类加载机制?

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的 java 类型。

类加载器双亲委派模型机制?

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类

去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载 java 核心类库，无法被 java 程序直接引用。
2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

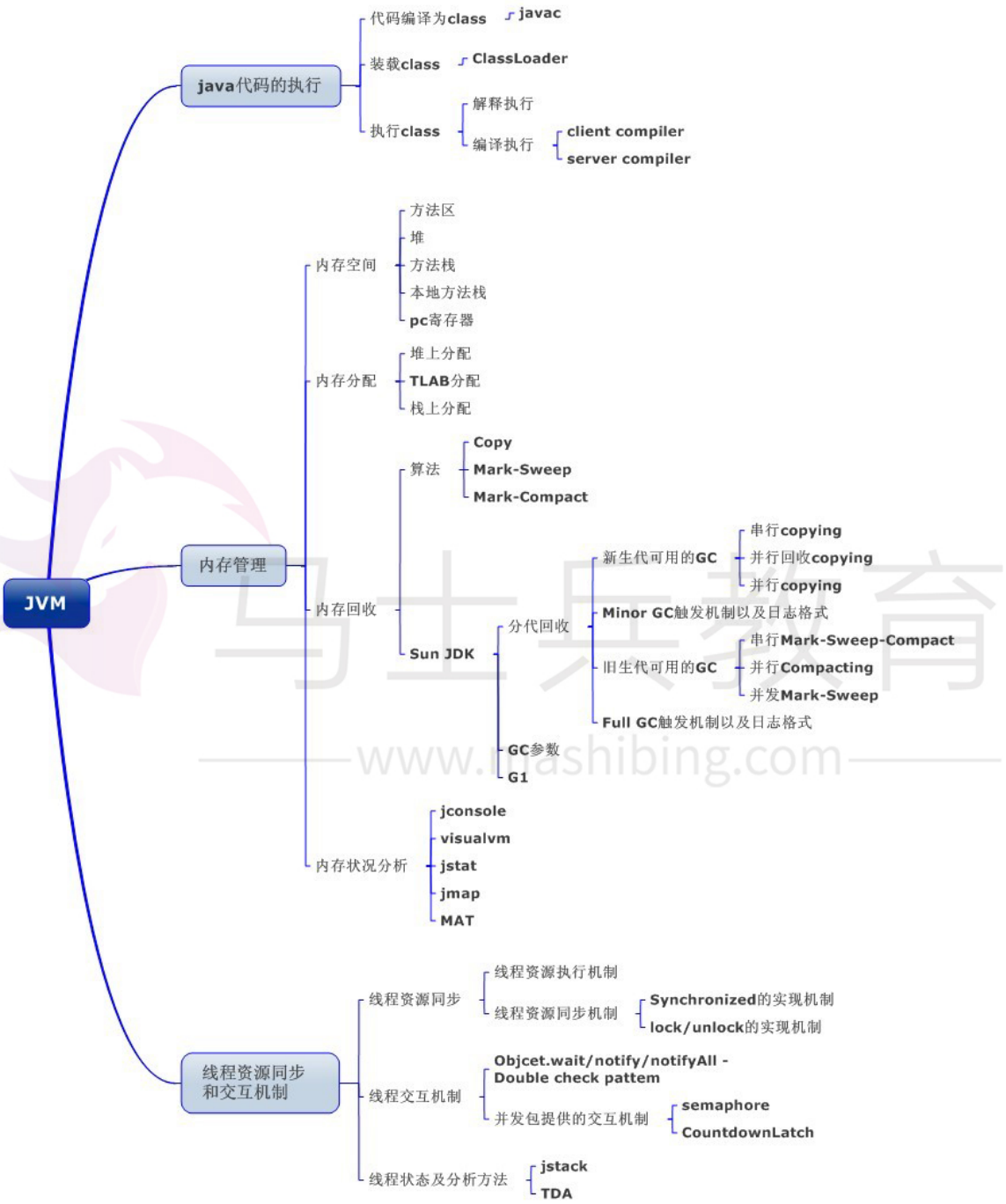
简述 java 内存分配与回收策率以及 Minor GC 和Major GC

1. 对象优先在堆的 Eden 区分配。
2. 大对象直接进入老年代.
3. 长期存活的对象将直接进入老年代.
   当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC.Minor Gc 通常发生在新生代的 Eden 区，在这个区的对象生存期短，往往发生 Gc 的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代 GC的时候不会触发 Minor GC,但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。

JVM

(1) 基本概念：

JVM是可运行Java代码的假想计算机，包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收，堆 和 一个存储方法域。JVM 是运行在操作系统之上的，它与硬件没有直接的交互。



(2) 运行过程：

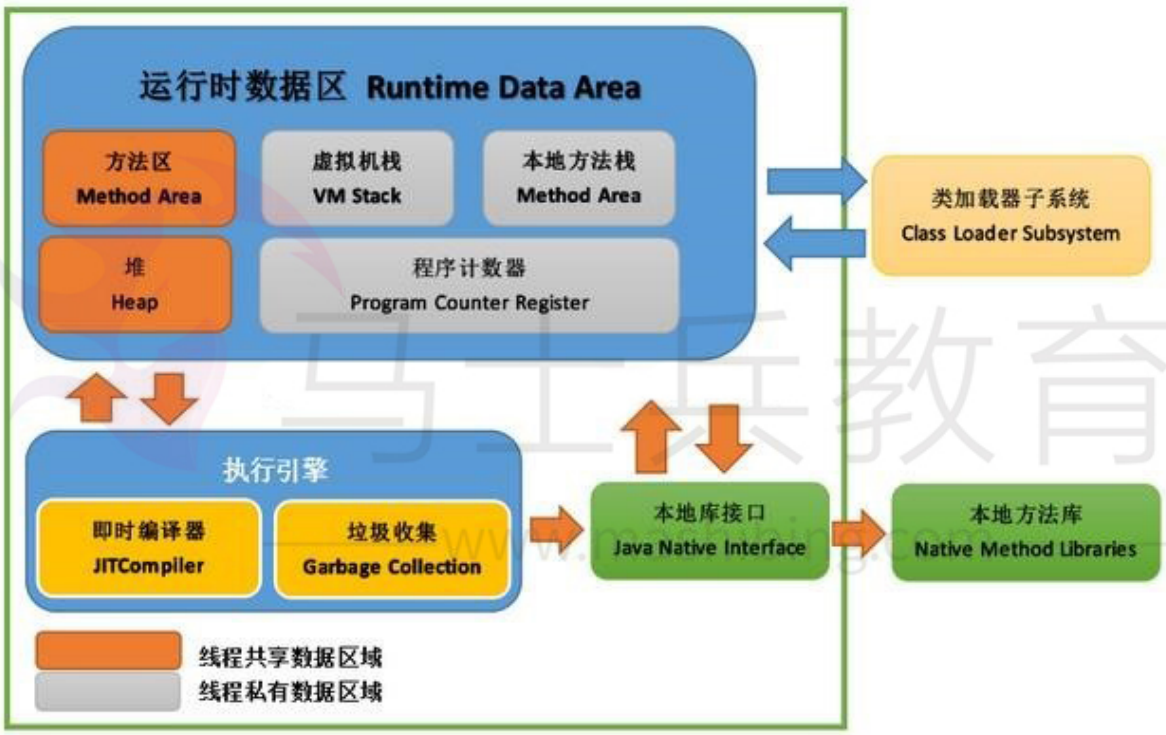
我们都知道 Java 源文件，通过编译器，能够生产相应的.Class 文件，也就是字节码文件，而字节码文件又通过 Java 虚拟机中的解释器，编译成特定机器上的机器码 。

也就是如下：

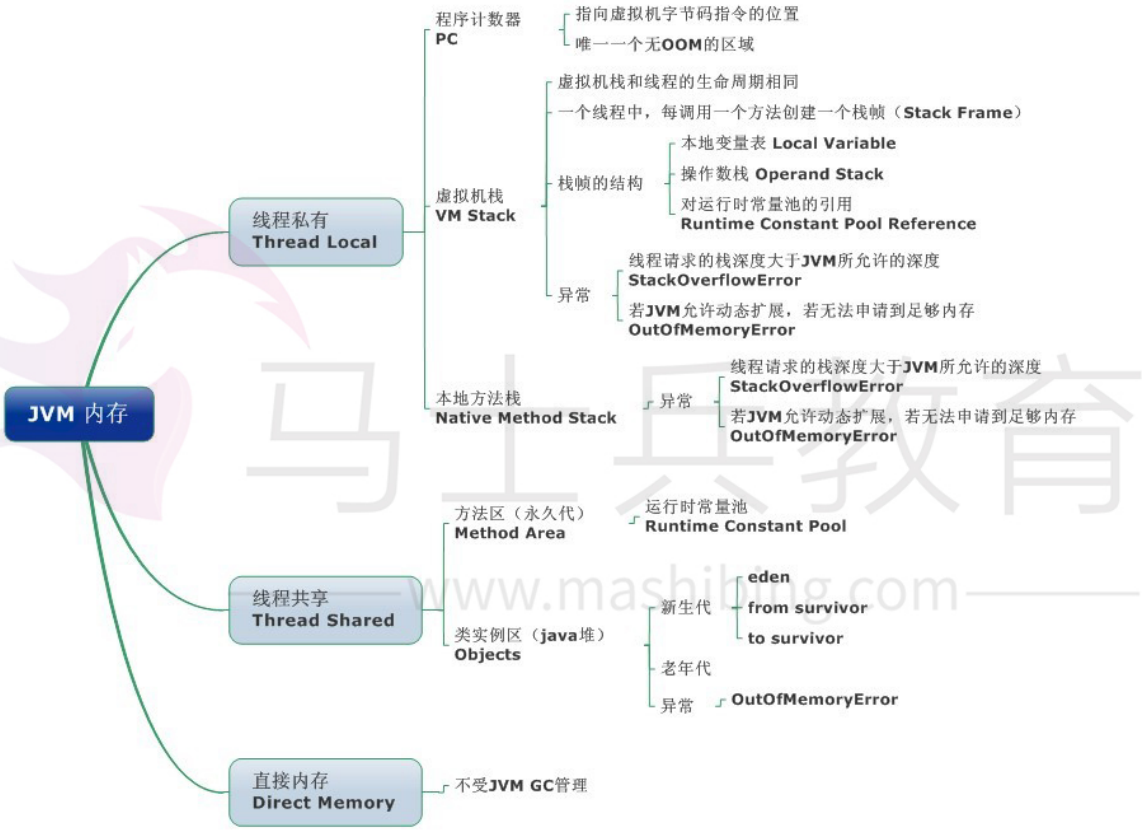
① Java 源文件—->编译器—->字节码文件

② 字节码文件—->JVM—->机器码

每一种平台的解释器是不同的，但是实现的虚拟机是相同的，这也就是 Java 为什么能够跨平台的原因了，当一个程序从开始运行，这时虚拟机就开始实例化了，多个程序启动就会存在多个虚拟机实例。程序退出或者关闭，则虚拟机实例消亡，多个虚拟机实例之间数据不能共享。



JVM 内存区域

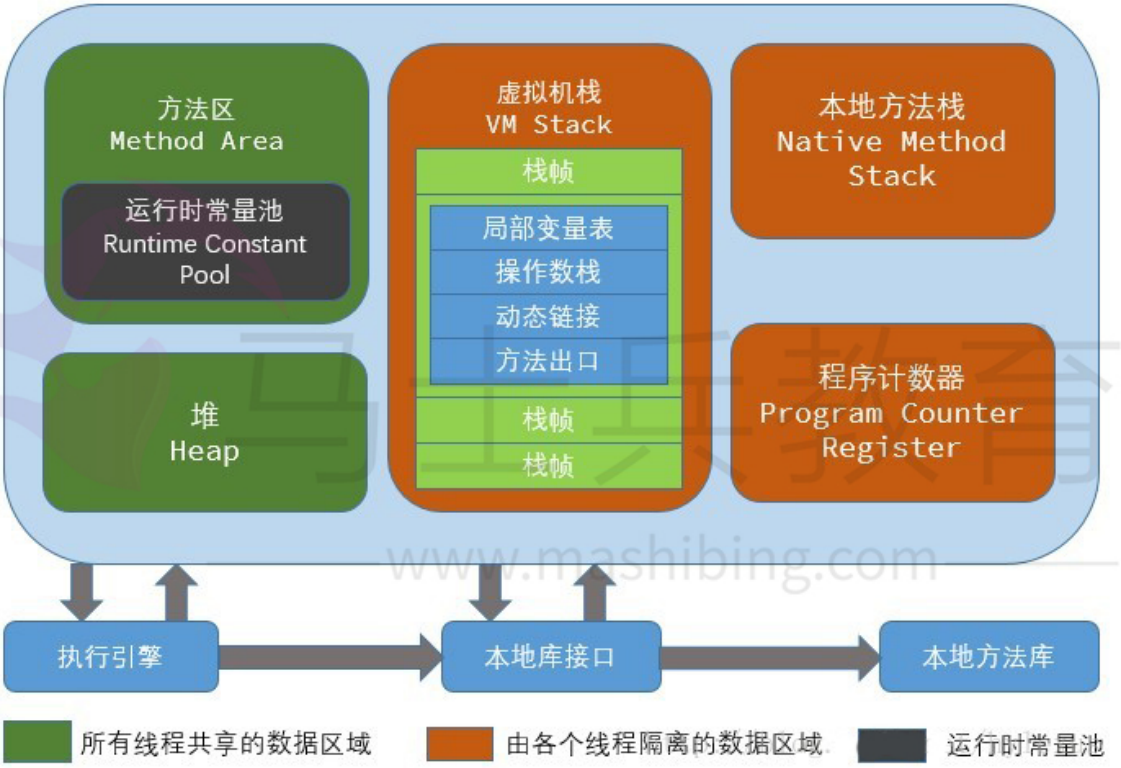


JVM 内存区域主要分为线程私有区域【程序计数器、虚拟机栈、本地方法区】、线程共享区域【JAVA 堆、方法区】、直接内存。

线程私有数据区域生命周期与线程相同, 依赖用户线程的启动/结束 而 创建/销毁(在 Hotspot VM 内, 每个线程都与操作系统的本地线程直接映射, 因此这部分内存区域的存/否跟随本地线程的生/死对应)。

线程共享区域随虚拟机的启动/关闭而创建/销毁。

直接内存并不是 JVM 运行时数据区的一部分, 但也会被频繁的使用: 在 JDK 1.4 引入的 NIO 提供了基于 Channel 与 Buffer 的 IO 方式, 它可以使用 Native 函数库直接分配堆外内存, 然后使用 DirectByteBuffer 对象作为这块内存的引用进行操作(详见:Java I/O 扩展, 这样就避免了在 Java 堆和 Native 堆中来回复制数据, 因此在一些场景中可以显著提高性能。



程序计数器(线程私有)

一块较小的内存空间, 是当前线程所执行的字节码的行号指示器，每条线程都要有一个独立的程序计数器，这类内存也称为“线程私有”的内存。

正在执行 java 方法的话，计数器记录的是虚拟机字节码指令的地址（当前指令的地址）。如果还是 Native 方法，则为空。

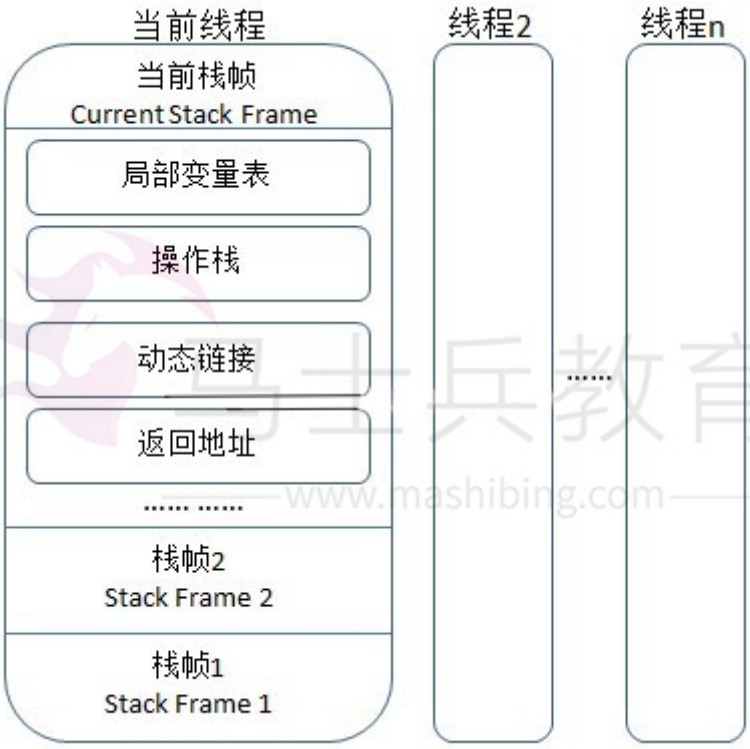
这个内存区域是唯一一个在虚拟机中没有规定任何 OutOfMemoryError 情况的区域。

虚拟机栈(线程私有)

是描述java方法执行的内存模型，每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

栈帧（ Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接

(Dynamic Linking)、 方法返回值和异常分派（ Dispatch Exception）。栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法结束。



本地方法区(线程私有)

本地方法区和 Java Stack 作用类似, 区别是虚拟机栈为执行 Java 方法服务, 而本地方法栈则为 Native 方法服务, 如果一个 VM 实现使用 C-linkage 模型来支持 Native 调用, 那么该栈将会是一个

C 栈，但 HotSpot VM 直接就把本地方法栈和虚拟机栈合二为一。

堆（Heap-线程共享）-运行时数据区

是被线程共享的一块内存区域，创建的对象和数组都保存在 Java 堆内存中，也是垃圾收集器进行垃圾收集的最重要的内存区域。由于现代 VM 采用**分代收集算法**, 因此 Java 堆从 GC 的角度还可以细分为: **新生代**(Eden区、From Survivor区和To Survivor区)和**老年代。**

方法区/永久代（线程共享）

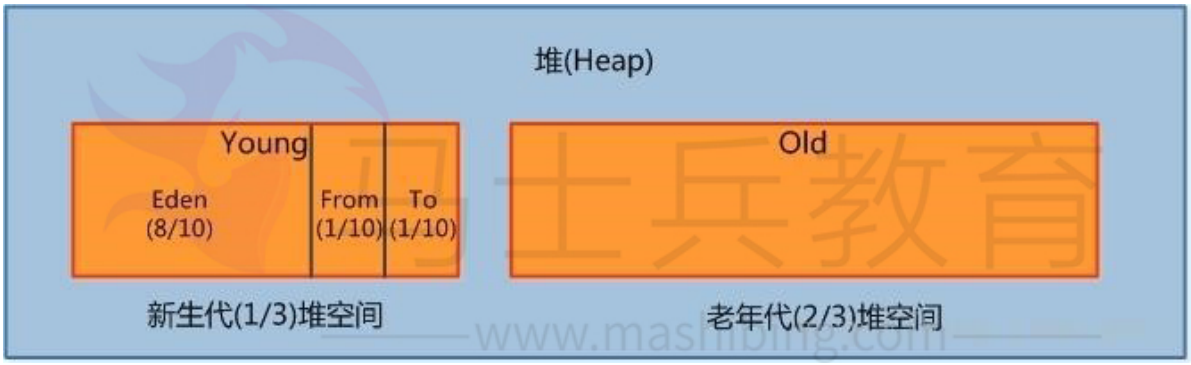
即我们常说的**永久代(Permanent Generation)**, 用于存储**被 JVM 加载的类信息、常量、静态变量、即时编译器编译后的代码**等数据. HotSpot VM把GC分代收集扩展至方法区, 即**使用Java 堆的永久代来实现方法区**, 这样 HotSpot 的垃圾收集器就可以像管理 Java 堆一样管理这部分内存, 而不必为方法区开发专门的内存管理器(永久带的内存回收的主要目标是针对**常量池的回收**和**类型的卸载**, 因此收益一般很小)。

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池

（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。 Java 虚拟机对 Class 文件的每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。

JVM 运行时内存

Java 堆从 GC 的角度还可以细分为: **新生代**(Eden区、From Survivor区和To Survivor区)和**老年代。**



新生代

是用来存放新生的对象。一般占据堆的1/3空间。由于频繁创建对象，所以新生代会频繁触发

MinorGC 进行垃圾回收。新生代又分为 Eden 区、ServivorFrom、ServivorTo 三个区。

1. Eden 区

Java新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当Eden区内存不够的时候就会触发MinorGC，对新生代区进行一次垃圾回收。

2. ServivorFrom

上一次 GC 的幸存者，作为这一次 GC 的被扫描者。

3. ServivorTo

保留了一次 MinorGC 过程中的幸存者。

4. MinorGC 的过程（复制->清空->互换）

MinorGC 采用复制算法。

1：eden、servicorFrom 复制到**ServicorTo**，年龄+1

首先，把Eden和ServivorFrom区域中存活的对象复制到ServicorTo区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果 ServicorTo 不够位置了就放到老年区）；

2：清空eden、servicorFrom

然后，清空 Eden 和 ServicorFrom 中的对象；

3：ServicorTo 和ServicorFrom 互换

最后，ServicorTo 和 ServicorFrom 互换，原 ServicorTo 成为下一次 GC 时的 ServicorFrom 区。

老年代

主要存放应用程序中生命周期长的内存对象。

老年代的对象比较稳定，所以 MajorGC 不会频繁执行。在进行 MajorGC 前一般都先进行了一次 MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次 MajorGC 进行垃圾回收腾出空间。

MajorGC 采用标记清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。MajorGC 的耗时比较长，因为要扫描再回收。MajorGC 会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。当老年代也满了装不下的时候，就会抛出 OOM（Out of Memory）异常。

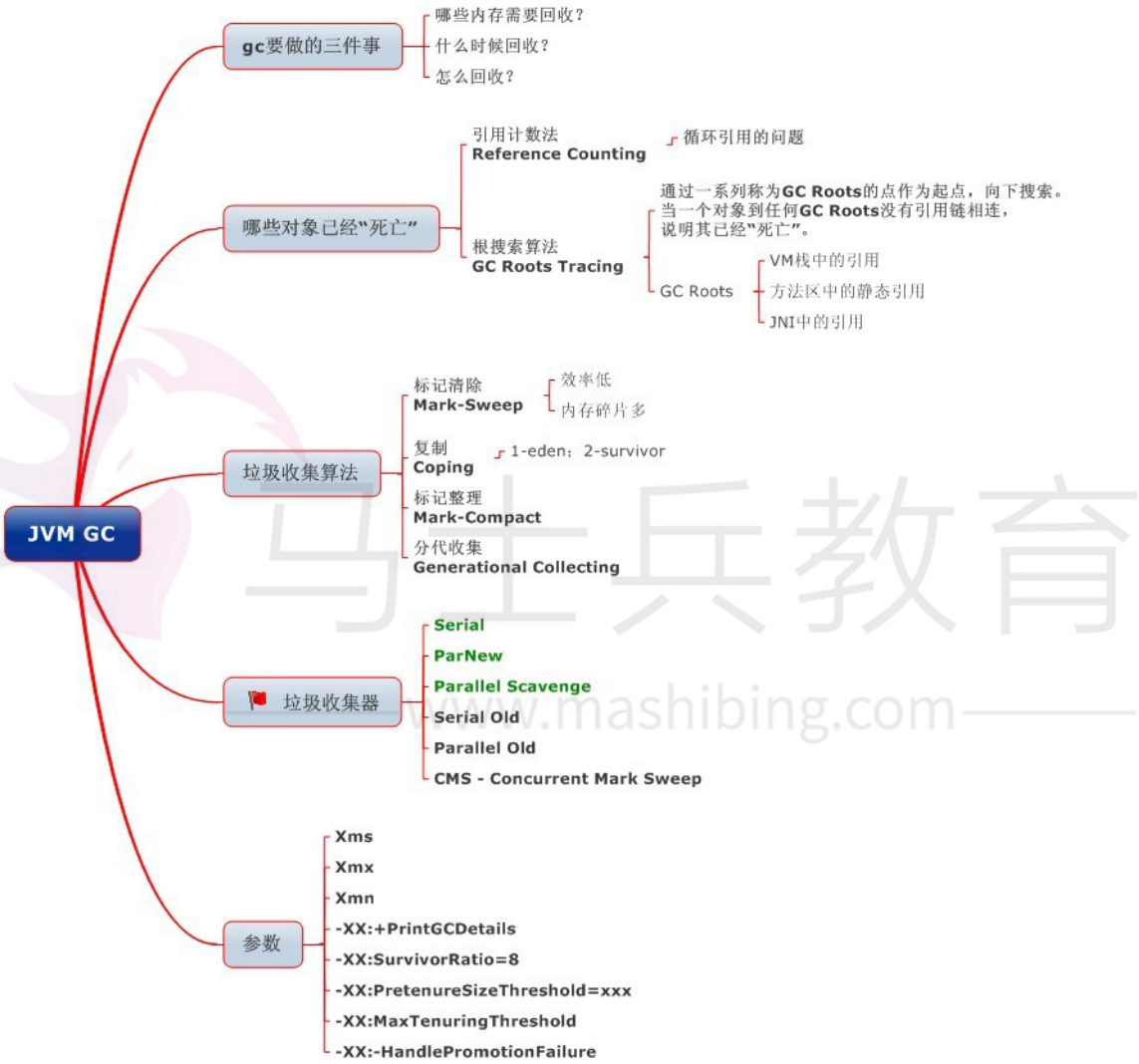
永久代

指内存的永久保存区域，主要存放 Class 和 Meta（元数据）的信息,Class 在被加载的时候被放入永久区域，它和和存放实例的区域不同,GC 不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的 Class 的增多而胀满，最终抛出 OOM 异常。

JAVA8与元数据

在Java8中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。元空间的本质和永久代类似，元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。类的元数据放入 native memory, 字符串池和类的静态变量放入 java 堆中，这样可以加载多少类的元数据就不再由MaxPermSize 控制, 而由系统的实际可用空间来控制。

垃圾回收与算法



如何确定垃圾

引用计数法

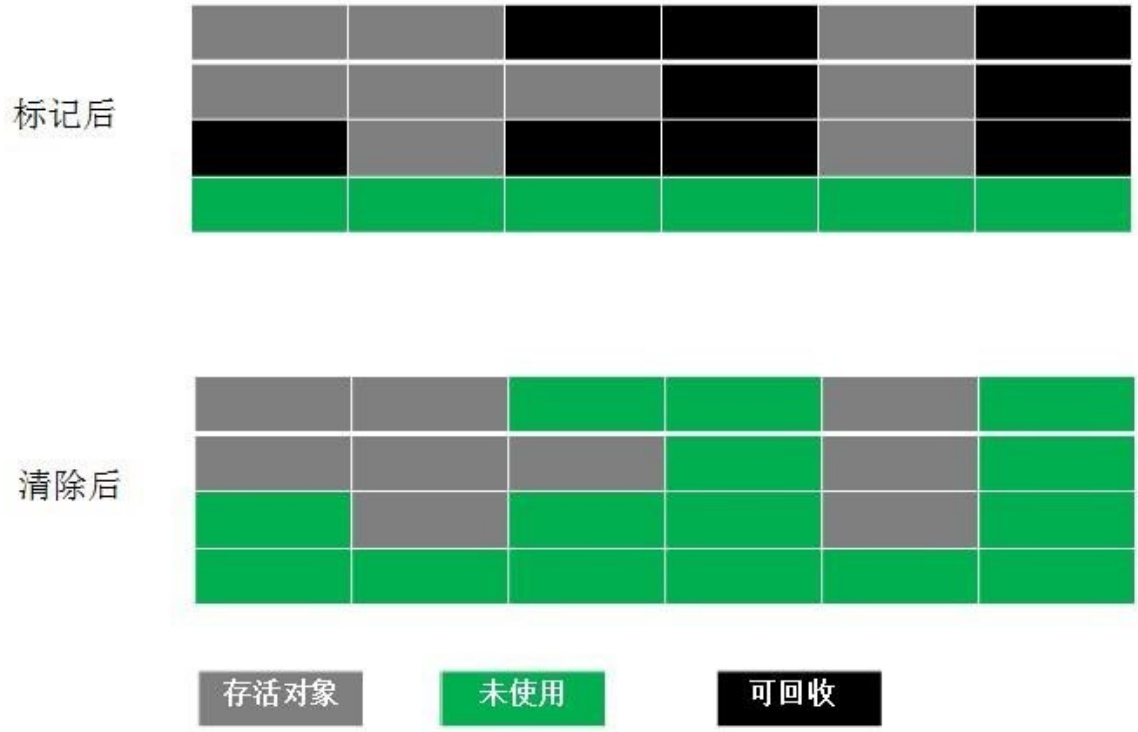
在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关联的引用，即他们的引用计数都不为 0，则说明对象不太可能再被用到，那么这个对象就是可回收对象。

可达性分析

为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。通过一系列的“GC roots” 对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记过程。两次标记后仍然是可回收对象，则将面临回收。

标记清除算法（Mark-Sweep）

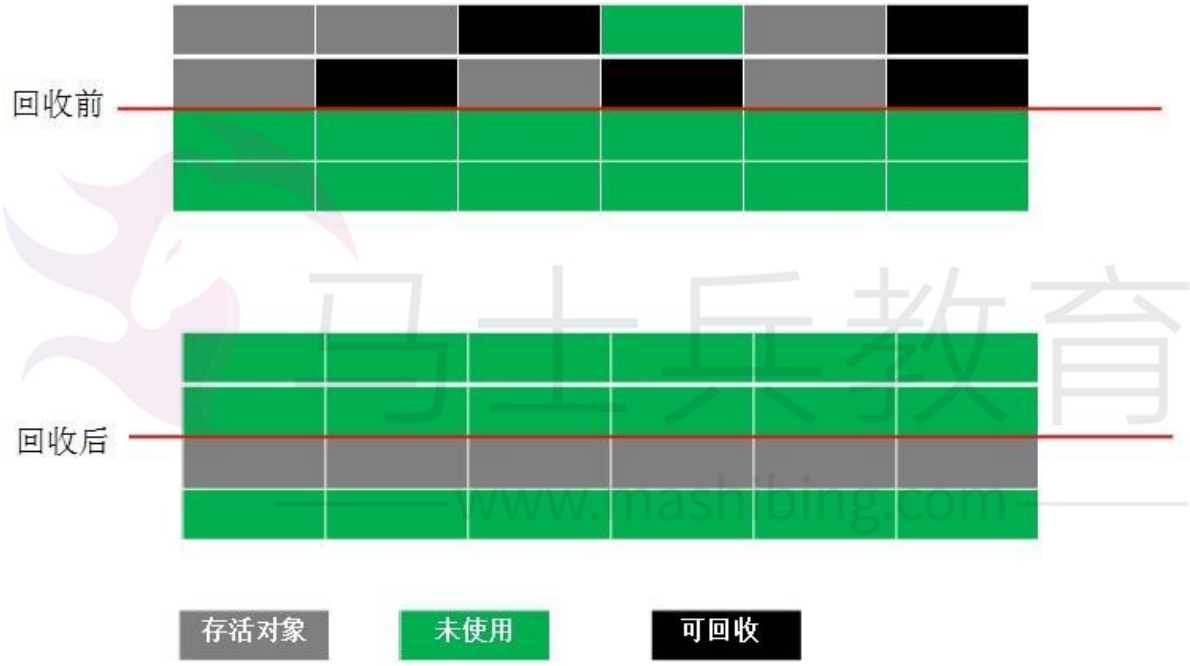
最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间。如图



我们就可以发现，该算法最大的问题是内存碎片化严重，后续可能发生大对象不能找到可利用空间的问题。

复制算法（copying）

为了解决 Mark-Sweep 算法内存碎片化的缺陷而被提出的算法。按内存容量将内存划分为等大小的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉，如图：



算法虽然实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原本的一半。且存活对象增多的话，Copying 算法的效率会大大降低。

标记整理算法(Mark-Compact)

结合了以上两个算法，为了避免缺陷而提出。标记阶段和 Mark-Sweep 算法相同，标记后不是清理对象，而是将存活对象移向内存的一端。然后清除端边界外的对象。如图：



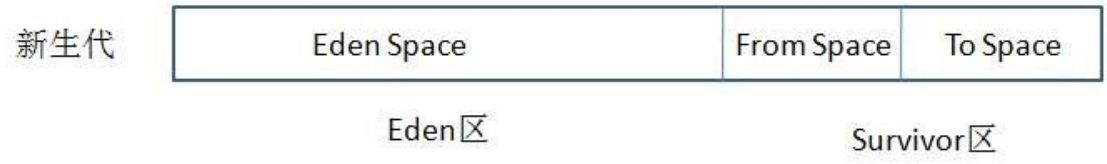
分代收集算法

分代收集法是目前大部分 JVM 所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将 GC 堆划分为老生代(Tenured/Old Generation)和新生代(Young Generation)。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

新生代与复制算法

目前大部分 JVM 的 GC 对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少，但通常并不是按照 1：1 来划分新生代。一般将新生代划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用

Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另一块 Survivor 空间中。



老年代与标记复制算法

而老年代因为每次只回收少量对象，因而采用 Mark-Compact 算法。

1. JAVA 虚拟机提到过的处于方法区的永生代(Permanet Generation)，它用来存储 class 类，常量，方法描述等。对永生代的回收主要包括废弃常量和无用的类。
2. 对象的内存分配主要在新生代的 Eden Space 和 Survivor Space 的 From Space(Survivor 目前存放对象的那一块)，少数情况会直接分配到老生代。
3. 当新生代的 Eden Space 和 From Space 空间不足时就会发生一次 GC，进行 GC 后，Eden Space 和 From Space 区的存活对象会被挪到 To Space，然后将 Eden Space 和 From Space 进行清理。
4. 如果 To Space 无法足够存储某个对象，则将这个对象存储到老生代。
5. 在进行 GC 后，使用的便是 Eden Space 和 To Space 了，如此反复循环。
6. 当对象在 Survivor 区躲过一次 GC 后，其年龄就会+1。默认情况下年龄到达 15 的对象会被移到老生代中。

JAVA四中引用类型

强引用

在 Java 中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引用。当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，即使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之一。

软引用

软引用需要用 SoftReference 类来实现，对于只有软引用的对象来说，当系统内存足够时它不会被回收，当系统内存空间不足时它会被回收。软引用通常用在对内存敏感的程序中。

弱引用

弱引用需要用 WeakReference 类来实现，它比软引用的生存期更短，对于只有弱引用的对象来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。

虚引用

虚引用需要 PhantomReference 类来实现，它不能单独使用，必须和引用队列联合使用。虚引用的主要作用是跟踪对象被垃圾回收的状态。

GC 分代收集算法 VS 分区收集算法

1. **分代收集算法**

当前主流 VM 垃圾收集都采用"分代收集"(Generational Collection)算法, 这种算法会根据对象存活周期的不同将内存划分为几块, 如 JVM 中的 新生代、老年代、永久代，这样就可以根据各年代特点分别采用最适当的 GC 算法

1.1. **在新生代-复制算法**

每次垃圾收集都能发现大批对象已死, 只有少量存活. 因此选用复制算法, 只需要付出少量存活对象的复制成本就可以完成收集.

1.2. **在老年代-标记整理算法**

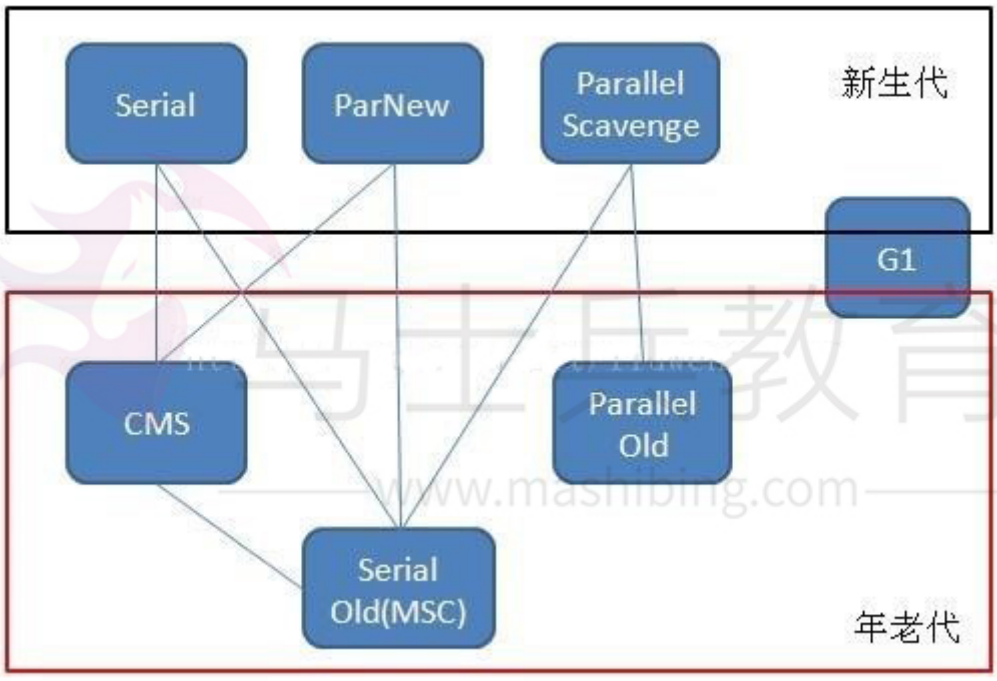
因为对象存活率高、没有额外空间对它进行分配担保, 就必须采用"标记—清理"或"标记—整理"算法来进行回收, 不必进行内存复制, 且直接腾出空闲内存.

2. **分区收集算法**

分区算法则将整个堆空间划分为连续的不同小区间, 每个小区间独立使用, 独立回收. 这样做的好处是可以控制一次回收多少个小区间 , 根据目标停顿时间, 每次合理地回收若干个小区间(而不是整个堆), 从而减少一次 GC 所产生的停顿。

GC垃圾收集器

Java 堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法年老代主要使用标记-整理垃圾回收算法，因此 java 虚拟中针对新生代和年老代分别提供了多种不同的垃圾收集器，JDK1.6 中 Sun HotSpot 虚拟机的垃圾收集器如下：



Serial 垃圾收集器（单线程、复制算法）

Serial（英文连续）是最基本垃圾收集器，使用复制算法，曾经是JDK1.3.1之前新生代唯一的垃圾收集器。Serial 是一个单线程的收集器，它不但只会使用一个 CPU 或一条线程去完成垃圾收集工作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。

Serial 垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个 CPU 环境来说，没有线程交互的开销，可以获得最高的单线程垃圾收集效率，因此 Serial 垃圾收集器依然是 java 虚拟机运行在 Client 模式下默认的新生代垃圾收集器。

ParNew 垃圾收集器（**Serial+**多线程）

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和 Serial 收集器完全一样，ParNew垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。

ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过-XX:ParallelGCThreads 参数来限制垃圾收集器的线程数。【Parallel：平行的】

ParNew虽然是除了多线程外和Serial收集器几乎完全一样，但是ParNew垃圾收集器是很多java

虚拟机运行在 Server 模式下新生代的默认垃圾收集器。

Parallel Scavenge 收集器（多线程复制算法、高效）

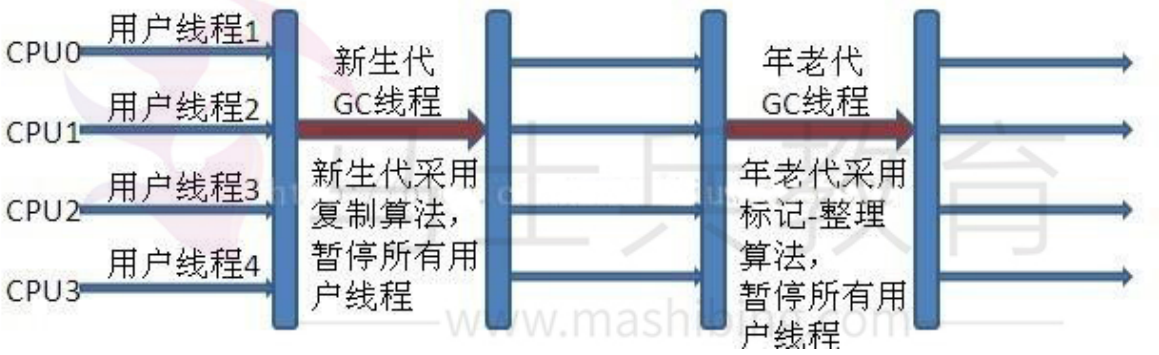
Parallel Scavenge 收集器也是一个新生代垃圾收集器，同样使用复制算法，也是一个多线程的垃圾收集器，它重点关注的是程序达到一个可控制的吞吐量（Thoughput，CPU 用于运行用户代码的时间/CPU 总消耗时间，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)），高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个重要区别。

Serial Old 收集器（单线程标记整理算法 ）

Serial Old 是 Serial 垃圾收集器年老代版本，它同样是个单线程的收集器，使用标记-整理算法，这个收集器也主要是运行在 Client 默认的 java 虚拟机默认的年老代垃圾收集器。

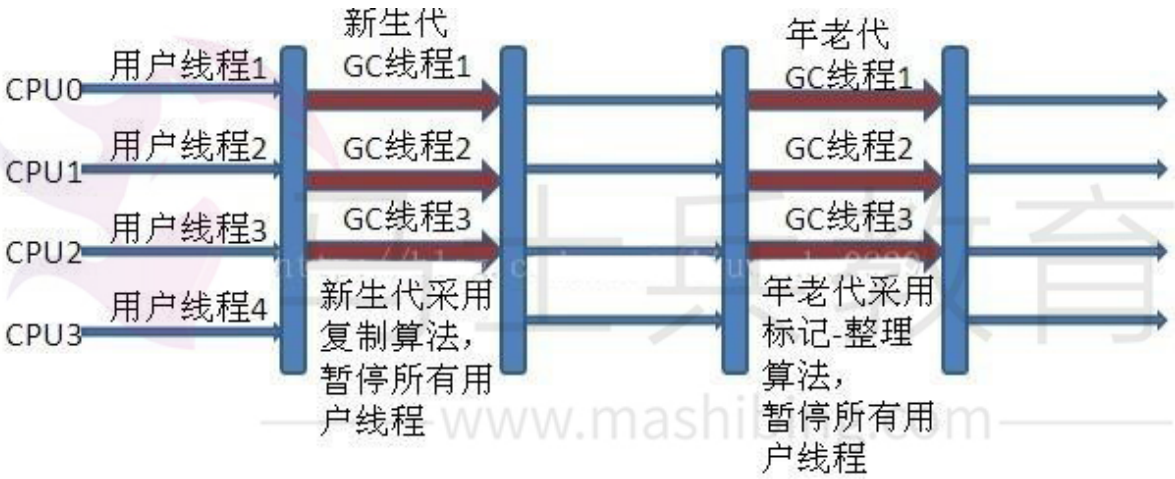
在 Server 模式下，主要有两个用途：

1. 在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。
2. 作为年老代中使用 CMS 收集器的后备垃圾收集方案。新生代 Serial 与年老代 Serial Old 搭配垃圾收集过程图：



代 Parallel Scavenge 收集器与 ParNew 收集器工作原理类似，都是多线程的收集器，都使

用的是复制算法，在垃圾收集过程中都需要暂停所有的工作线程。新生代 Parallel Scavenge/ParNew 与年老代 Serial Old 搭配垃圾收集过程图：



CMS 收集器（多线程标记清除算法）

Concurrent mark sweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。

最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验。

CMS 工作机制相比其他的垃圾收集器来说更复杂，整个过程分为以下 4 个阶段：

1. **初始标记**

只是标记一下 GC Roots 能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。

2. **并发标记**

进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。

3. **重新标记**

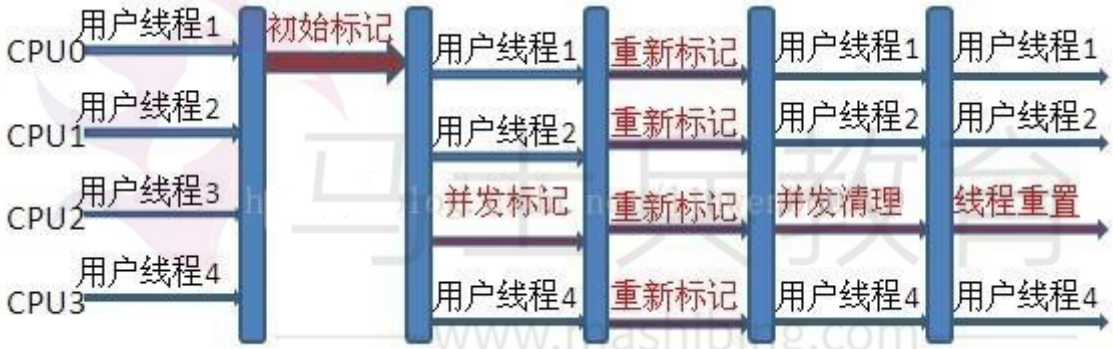
为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程。

4. **并发清除**

清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作，所以总体上来看

CMS 收集器的内存回收和用户线程是一起并发地执行。

CMS 收集器工作过程：



G1 收集器

Garbage first 垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比与 CMS 收集器，G1 收集器两个最突出的改进是：

1. 基于标记-整理算法，不产生内存碎片。
2. 可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾最多的区域。区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收集效率。

JAVA IO/NIO

1. 阻塞 **IO** 模型

最传统的一种 IO 模型，即在读写数据过程中会发生阻塞现象。当用户线程发出 IO 请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出 CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才解除 block 状态。典型的阻塞 IO 模型的例子为：data = socket.read();如果数据没有就绪，就会一直阻塞在 read 方法。

2. 非阻塞 **IO** 模型

当用户线程发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。如果结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦内核中的数据准备好了，并且又再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。

所以事实上，在非阻塞 IO 模型中，用户线程需要不断地询问内核数据是否就绪，也就说非阻塞 IO 不会交出 CPU，而会一直占用 CPU。典型的非阻塞 IO 模型一般如下：

```
while(true){
data = socket.read();
if(data!= error){处理数据
break;
}
}
```

但是对于非阻塞 IO 就有一个非常严重的问题，在 while 循环中需要不断地去询问内核数据是否就绪，这样会导致 CPU 占用率非常高，因此一般情况下很少使用 while 循环这种方式来读取数据。

3. 多路复用 **IO** 模型

多路复用 IO 模型是目前使用得比较多的模型。Java NIO 实际上就是多路复用 IO。在多路复用 IO 模型中，会有一个线程不断去轮询多个 socket 的状态，只有当 socket 真正有读写事件时，才真正调用实际的 IO 读写操作。因为在多路复用 IO 模型中，只需要使用一个线程就可以管理多个 socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有socket 读写事件进行时，才会使用 IO 资源，所以它大大减少了资源占用。在 Java NIO 中，是通过 selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。多路复用 IO 模式，通过一个线程就可以管理多个 socket，只有当 socket 真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用 IO 比较适合连接数比较多的情况。

另外多路复用 IO 为何比非阻塞 IO 模型的效率高是因为在非阻塞 IO 中，不断地询问 socket 状态时通过用户线程去进行的，而在多路复用 IO 中，轮询每个 socket 状态是内核在进行的，这个效率要比用户线程要高的多。

不过要注意的是，多路复用 IO 模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用 IO 模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

4. 信号驱动 **IO** 模型

在信号驱动 IO 模型中，当用户线程发起一个 IO 请求操作，会给对应的 socket 注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用 IO 读写操作来进行实际的 IO 请求操作。

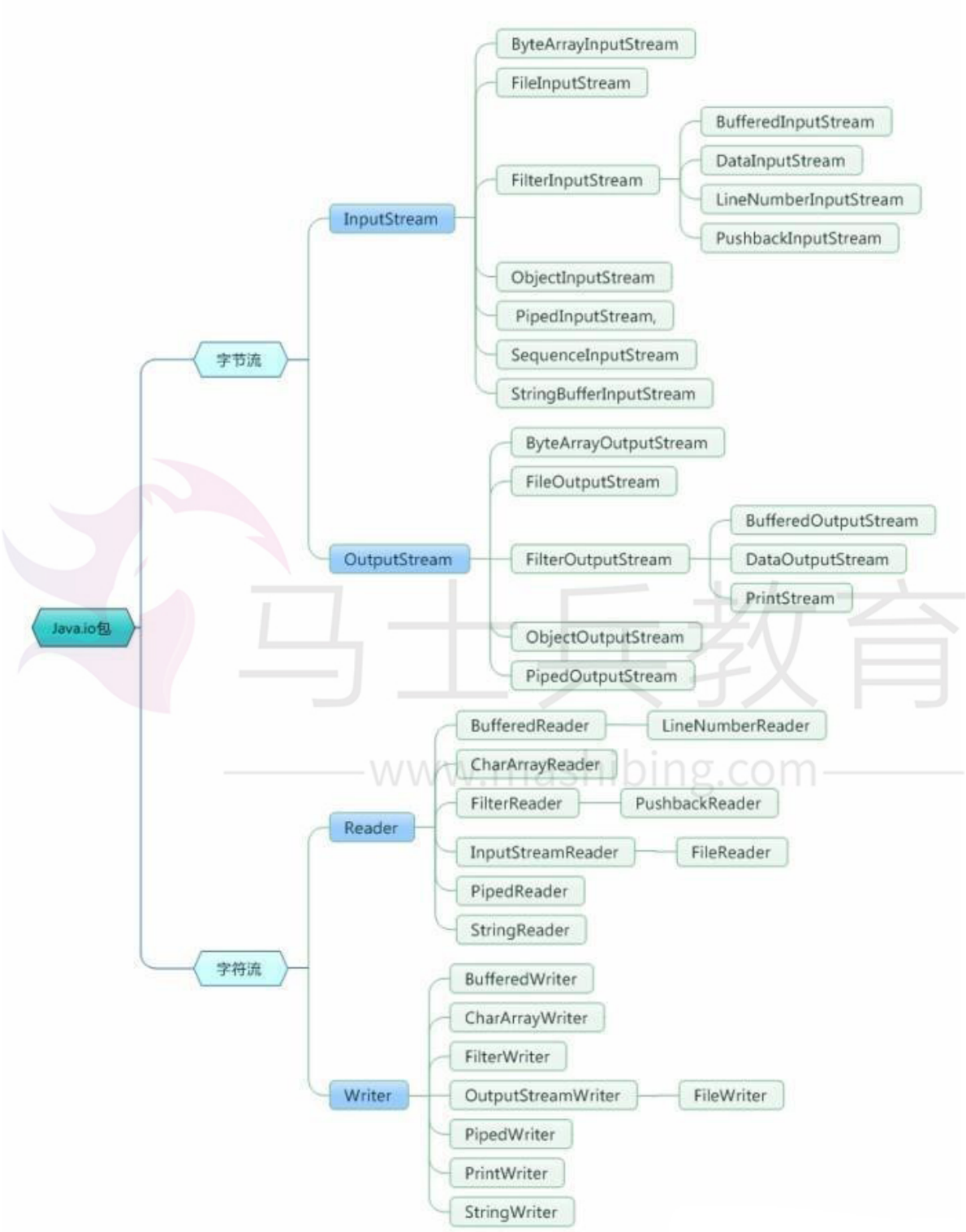
5. 异步 **IO** 模型

异步 IO 模型才是最理想的 IO 模型，在异步 IO 模型中，当用户线程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个 asynchronous read 之后，它会立刻返回，说明 read 请求已经成功发起了，因此不会对用户线程产生任何 block。然后，内核会等待数据准备完成，然后将数据拷贝到用户线程，当这一切都完成之后，内核会给用户线程发送一个信号，告诉它 read 操作完成了。也就说用户线程完全不需要实际的整个 IO 操作是如何进行的，只需要先发起一个请求，当接收内核返回的成功信号时表示 IO 操作已经完成，可以直接去使用数据了。

也就说在异步 IO 模型中，IO 操作的两个阶段都不会阻塞用户线程，这两个阶段都是由内核自动完成，然后发送一个信号告知用户线程操作已完成。用户线程中不需要再次调用 IO 函数进行具体的读写。这点是和信号驱动模型有所不同的，在信号驱动模型中，当用户线程接收到信号表示数据已经就绪，然后需要用户线程调用 IO 函数进行实际的读写操作；而在异步 IO 模型中，收到信号表示 IO 操作已经完成，不需要再在用户线程中调用 IO 函数进行实际的读写操作。

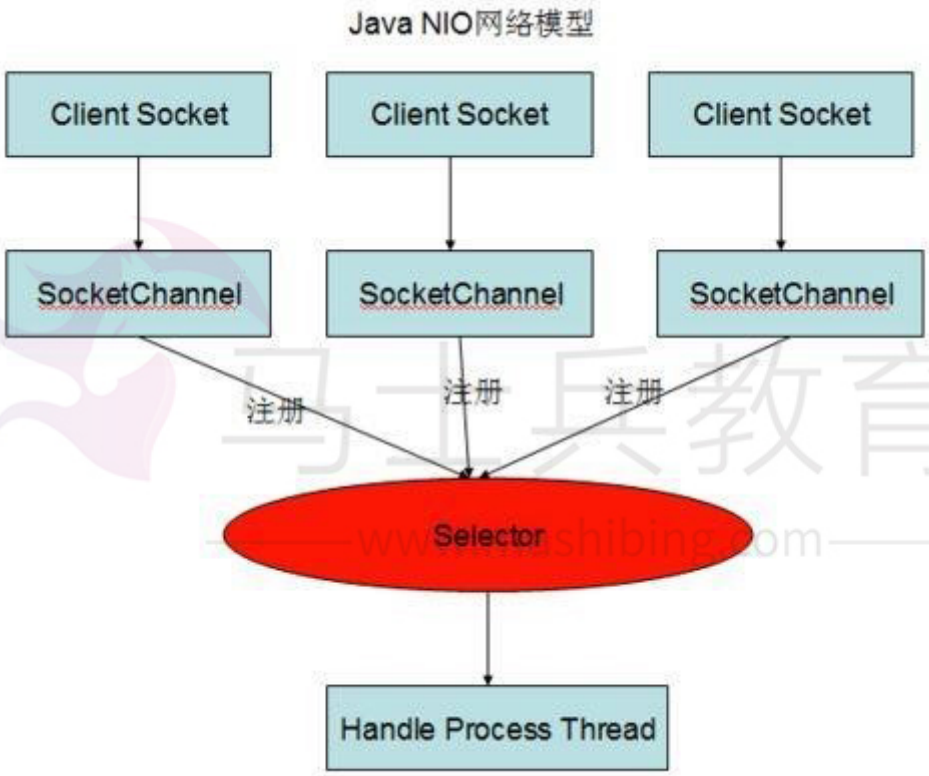
注意，异步 IO 是需要操作系统的底层支持，在 Java 7 中，提供了 Asynchronous IO。

JAVA IO 包



JAVA NIO

NIO 主要有三大核心部分：Channel(通道)，Buffer(缓冲区), Selector。传统 IO 基于字节流和字符流进行操作，而 NIO 基于 Channel 和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。



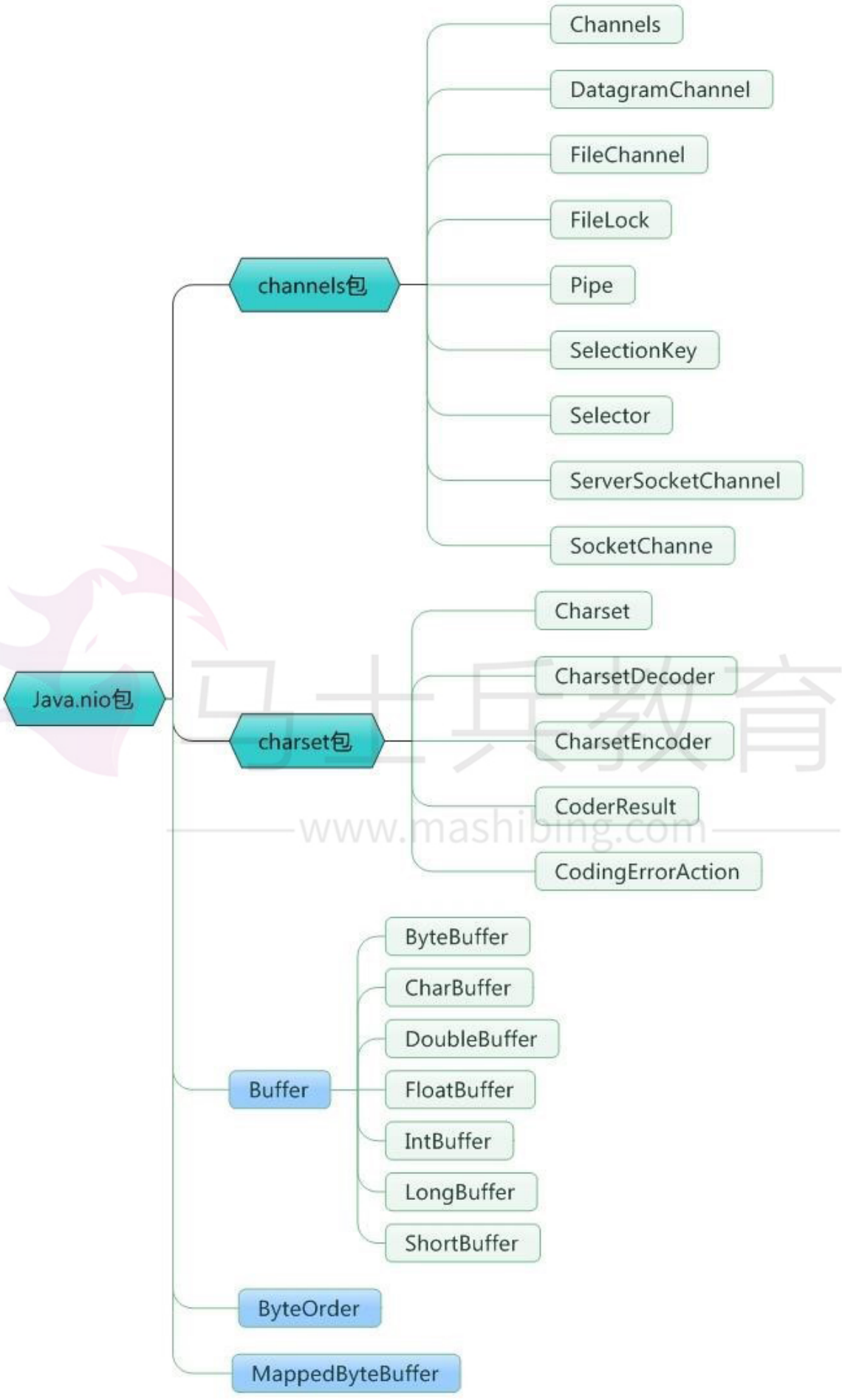
NIO 和传统 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。

NIO 的缓冲区

Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。NIO 的缓冲导向方法不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

NIO 的非阻塞

IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。



马士兵教育 www.mashibing.com
第七城市 WWW.THSF.CN

Channel

首先说一下 Channel，国内大多翻译成“通道”。Channel 和 IO 中的 Stream(流)是差不多一个

等级的。只不过 Stream 是单向的，譬如：InputStream, OutputStream，而 Channel 是双向的，既可以用来进行读操作，又可以用来进行写操作。

NIO 中的 Channel 的主要实现有：

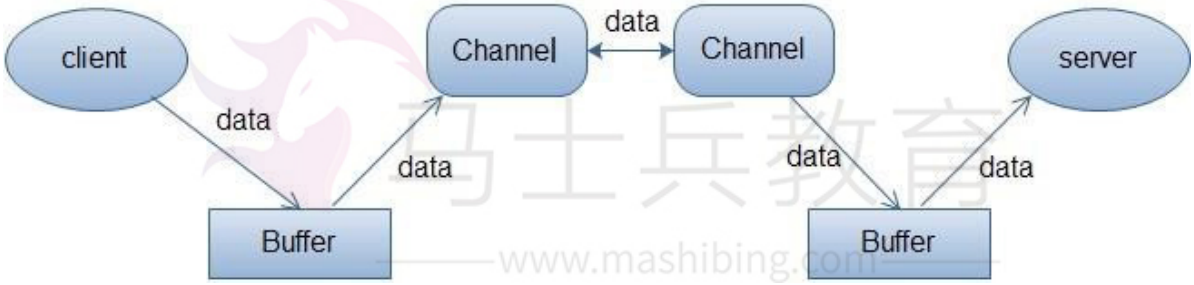
1. FileChannel
2. DatagramChannel
3. SocketChannel
4. ServerSocketChannel

这里看名字就可以猜出个所以然来：分别可以对应文件 IO、UDP 和 TCP（Server 和 Client）。

下面演示的案例基本上就是围绕这 4 个类型的 Channel 进行陈述的。

Buffer

Buffer，故名思意，缓冲区，实际上是一个容器，是一个连续数组。Channel 提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由 Buffer。



上面的图描述了从一个客户端向服务端发送数据，然后服务端接收数据的过程。客户端发送数据时，必须先将数据存入 Buffer 中，然后将 Buffer 中的内容写入通道。服务端这边接收数据必须通过 Channel 将数据读入到 Buffer 中，然后再从 Buffer 中取出数据来处理。

在 NIO 中，Buffer 是一个顶层父类，它是一个抽象类，常用的 Buffer 的子类有： ByteBuffer、IntBuffer、 CharBuffer、 LongBuffer、 DoubleBuffer、FloatBuffer、

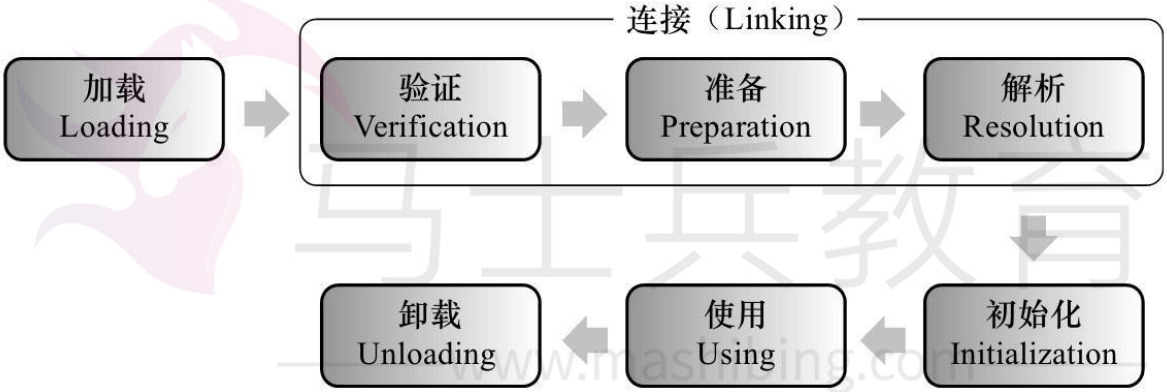
ShortBuffer

Selector

Selector 类是 NIO 的核心类，Selector 能够检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的响应处理。这样一来，只是用一个单线程就可以管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销。

JVM 类加载机制

JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化，下面我们就分别来看一下这五个过程。



加载

加载是类加载过程中的一个阶段，这个阶段会在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的入口。注意这里不一定非得要从一个 Class 文件获取，这里既可以从 ZIP 包中读取（比如从 jar 包和 war 包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将 JSP 文件转换成对应的 Class 类）。

验证

这一阶段的主要目的是为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

准备

准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

```
public static int v = 8080;
```

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080，将 v 赋值为 8080 的 put static 指令是程序被编译后，存放于类构造器方法之中。

但是注意如果声明为：

```
public static final int v = 8080;
```

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

```
1.  CONSTANT_Class_info
2.  CONSTANT_Field_info
3.  CONSTANT_Method_info 等类型的常量。
```

符号引用

n 符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在 Java 虚拟机规范的 Class 文件格式中。

直接引用

n 直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

初始化

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由 JVM 主导。到了初始阶段，才开始真正执行类中定义的 Java 程序代码。

类构造器

初始化阶段是执行类构造器方法的过程。方法是由编译器自动收集类中的类变量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证子方法执行之前，父类的方法已经执行完毕，如果一个类中没有对静态变量赋值也没有静态语句块，那么编译器可以不为这个类生成()方法。

注意以下几种情况不会执行类初始化：

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。
3. 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
4. 通过类名获取 Class 对象，不会触发类的初始化。
5. 通过 Class.forName 加载指定类时，如果指定参数 initialize 为 false 时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
6. 通过 ClassLoader 默认的 loadClass 方法，也不会触发初始化动作。

类加载器

虚拟机设计团队把加载动作放到 JVM 外部实现，以便让应用程序决定如何获取所需的类，JVM 提供了 3 种类加载器：

启动类加载器(Bootstrap ClassLoader)

1. 负责加载 JAVA_HOME\lib 目录中的，或通过-Xbootclasspath 参数指定路径中的，且被虚拟机认可（按文件名识别，如 rt.jar）的类。

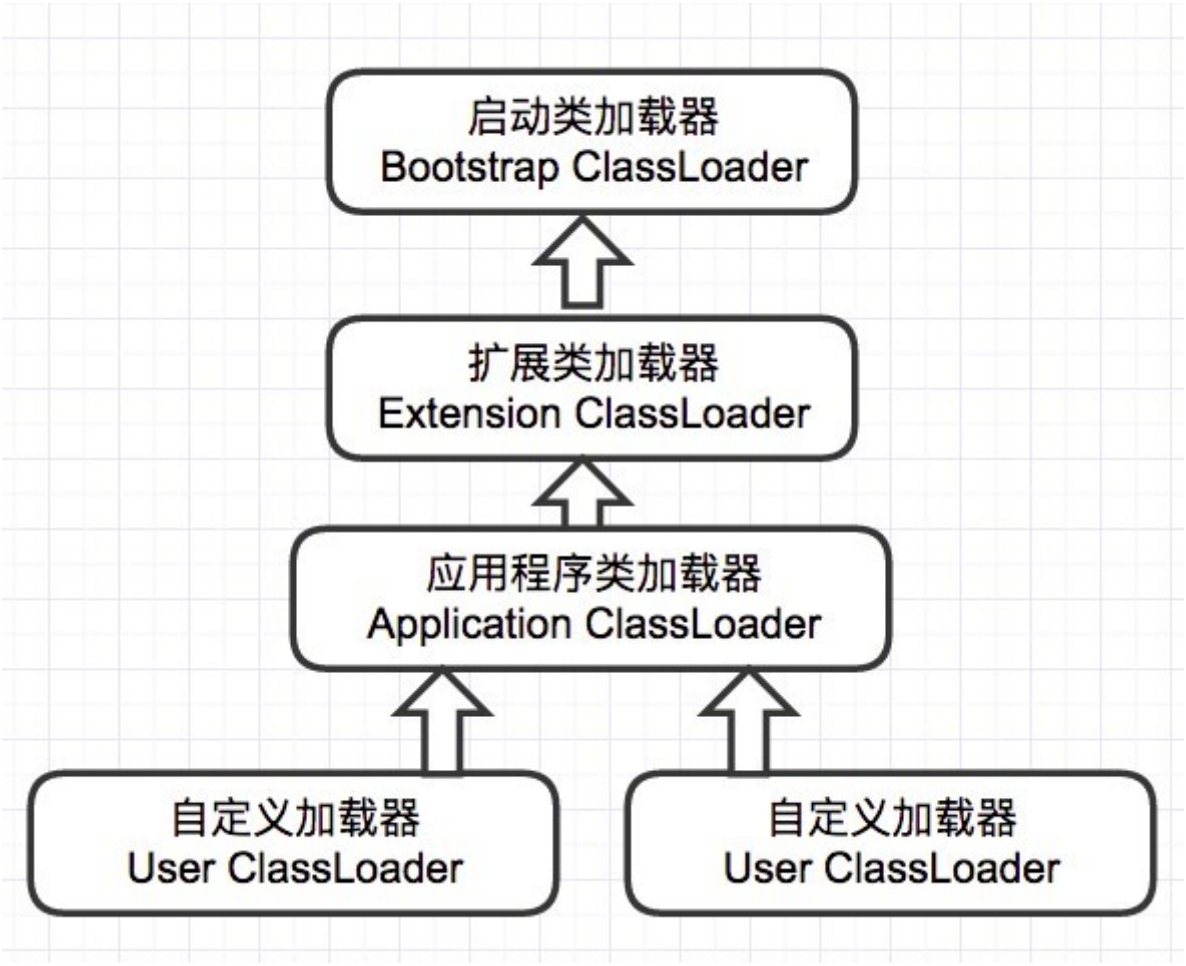
扩展类加载器(Extension ClassLoader)

2. 负责加载 JAVA_HOME\lib\ext 目录中的，或通过 java.ext.dirs 系统变量指定路径中的类库。

应用程序类加载器(Application ClassLoader)：

3. 负责加载用户路径（classpath）上的类库。

JVM 通过双亲委派模型进行类的加载，当然我们也可以通过继承 java.lang.ClassLoader 实现自定义的类加载器。



双亲委派

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的

Class），子类加载器才会尝试自己去加载。

采用双亲委派的一个好处是比如加载位于 rt.jar 包中的类 java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个 Object 对象。



OSGI（动态模型系统**）

OSGi(Open Service Gateway Initiative)，是面向 Java 的动态模型系统，是 Java 动态化模块化系统的一系列规范。

动态改变构造

OSGi 服务平台提供在多种网络设备上无需重启的动态改变构造的功能。为了最小化耦合度和促使这些耦合度可管理，OSGi 技术提供一种面向服务的架构，它能使这些组件动态地发现对方。

模块化编程与热插拔

OSGi 旨在为实现 Java 程序的模块化编程提供基础条件，基于 OSGi 的程序很可能可以实现模块级的热插拔功能，当程序升级更新时，可以只停用、重新安装然后启动程序的其中一部分，这对企业级程序开发来说是非常具有诱惑力的特性。

OSGi 描绘了一个很美好的模块化开发目标，而且定义了实现这个目标的所需要服务与架构，同时也有成熟的框架进行实现支持。但并非所有的应用都适合采用 OSGi 作为基础架构，它在提供强大功能同时，也引入了额外的复杂度，因为它不遵守了类加载的双亲委托模型。

多线程（马老师，黄老师）

简述线程，程序、进程的基本概念。以及他们之间关系是什么

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

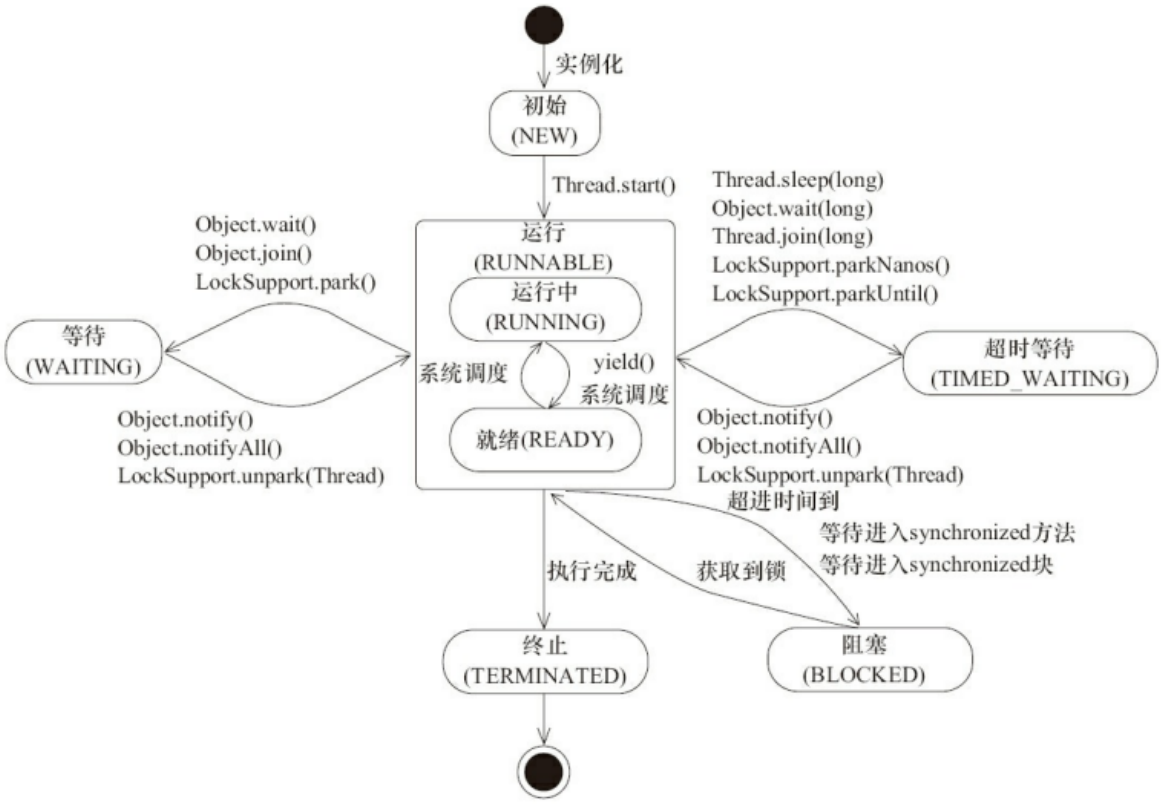
程序是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。 线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

线程有哪些基本状态?

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态。

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示：



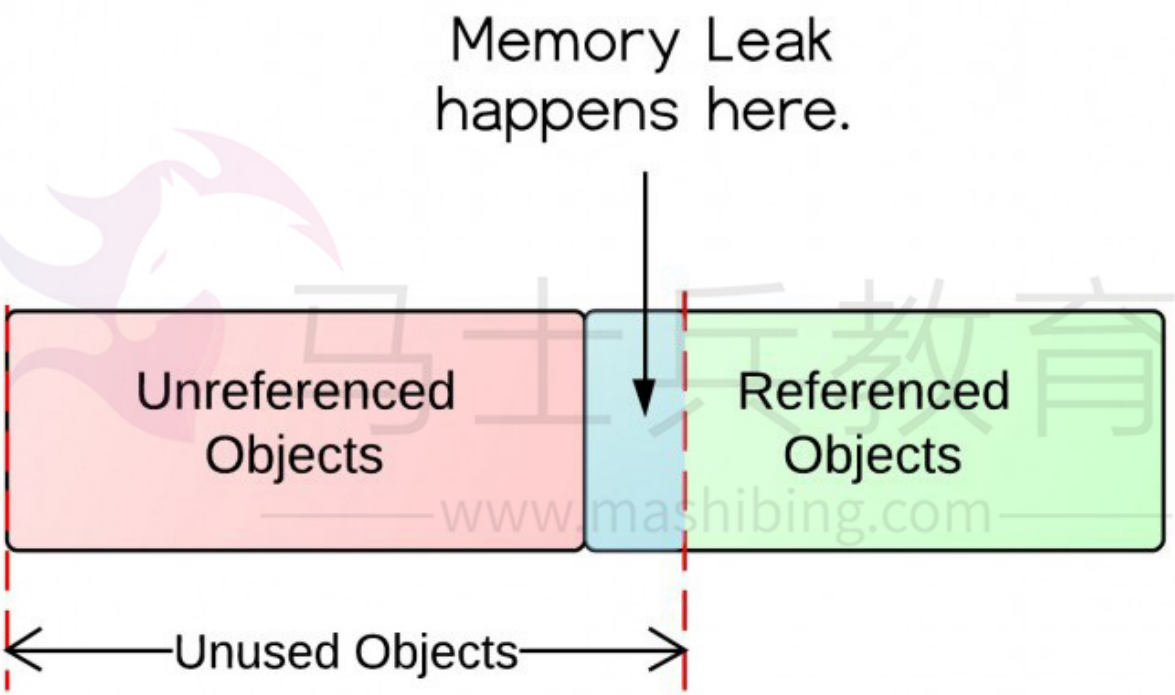
如何理解内存泄漏问题？有哪些情况会导致内存泄露？如何解决？

详解java内存泄露和如何避免内存泄漏

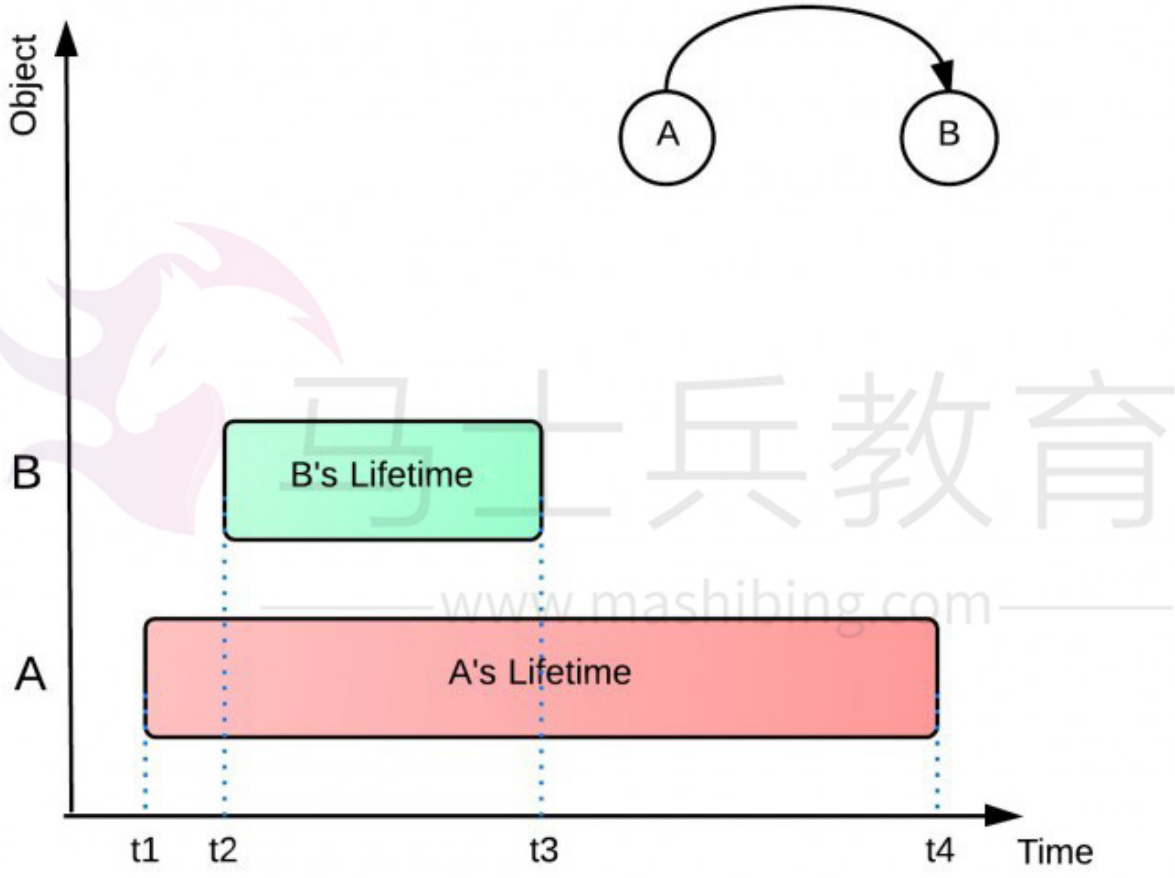
一直以来java都占据着语言排行榜的头把交椅。这是与java的设计密不可分的，其中最令大家喜欢的不是面向对象，而是垃圾回收机制。你只需要简单的创建对象而不需要负责释放空间，因为Java的垃圾回收器会负责内存的回收。然而，情况并不是这样简单，内存泄露还是经常会在Java应用程序中出现。
下面我们将详细的学习什么是内存泄露，为什么会发生，以及怎样阻止内存泄露。

什么是内存泄露

内存泄露的定义：对于应用程序来说，当对象已经不再被使用，但是Java的垃圾回收器不能回收它们的时候，就产生了内存泄露。
要理解这个定义，我们需要理解对象在内存中的状态。如下图所示，展示了哪些对象是无用对象，哪些是未被引用的对象；



上图中包含了未引用对象和引用对象。未引用对象将会被垃圾回收器回收，而引用对象却不会。未引用对象很显然是无用的对象。然而，无用的对象并不都是未引用对象，有一些无用对象也有可能是引用对象，这部分对象正是内存泄露的来源。



为什么内存泄露会发生

让我们用下面的例子来看看为什么会发生内存泄露。如下图所示，对象A引用对象B，A的生命周期（t1-t4）比B的生命周期（t2-t3）要长，当B在程序中不再被使用的时候，A仍然引用着B。在这种情况下，垃圾回收器是不会回收B对象的，这就可能造成了内存不足问题，因为A可能不止引用着B对象，还可能引用其它生命周期比A短的对象，这就造成了大量无用对象不能被回收，且占据了昂贵的内存资源。

同样的，B对象也可能引用着一大堆对象，这些被B对象引用着的对象也不能被垃圾回收器回收，所有的这些无用对象消耗了大量内存资源。

怎样阻止内存泄露

1.使用List、Map等集合时，在使用完成后赋值为null
2.使用大对象时，在用完后赋值为null
3.目前已知的jdk1.6的substring()方法会导致内存泄露
4.避免一些死循环等重复创建或对集合添加元素，撑爆内存
5.简洁数据结构、少用静态集合等
6.及时的关闭打开的文件，socket句柄等
7.多关注事件监听(listeners)和回调(callbacks)，比如注册了一个listener，当它不再被使用的时候，忘了注销该listener，可能就会产生内存泄露

线程池的原理，为什么要创建线程池？创建线程池的方式;

原理:

线程池的优点

1、线程是稀缺资源，使用线程池可以减少创建和销毁线程的次数，每个工作线程都可以重复使用。

2、可以根据系统的承受能力，调整线程池中工作线程的数量，防止因为消耗过多内存导致服务器崩溃。

线程池的创建

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)
```

corePoolSize：线程池核心线程数量

maximumPoolSize:线程池最大线程数量

keepAliverTime：当活跃线程数大于核心线程数时，空闲的多余线程最大存活时间

unit：存活时间的单位

workQueue：存放任务的队列

handler：超出线程范围和队列容量的任务的处理程序

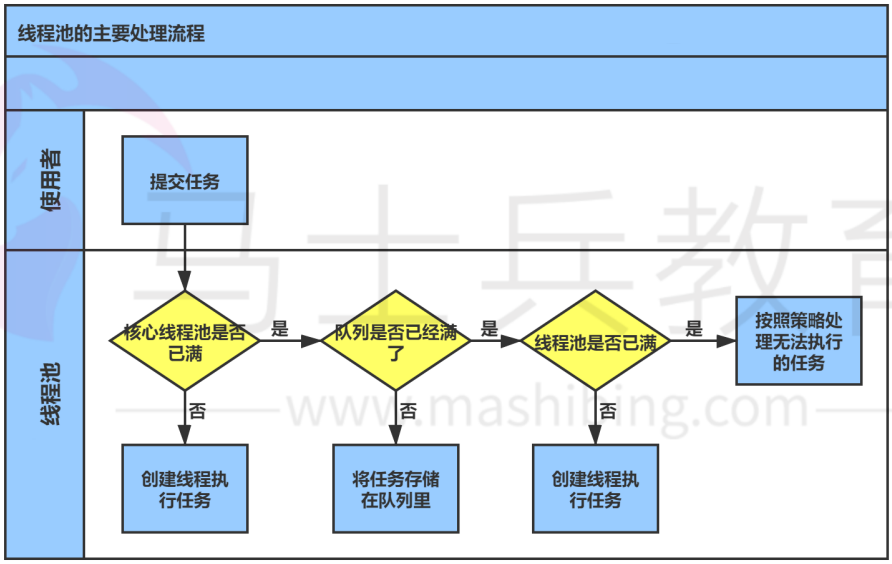
线程池的实现原理

提交一个任务到线程池中，线程池的处理流程如下：

1、判断**线程池里的核心线程**是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3、判断**线程池里的线程**是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。



线程池的源码解读

1、ThreadPoolExecutor的execute()方法

```
 1 public void execute(Runnable command) {
 2         if (command == null)
 3             throw new NullPointerException();           //如果线程数大于等于基本线
程数或者线程创建失败，将任务加入队列
 4         if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command))
{                 //线程池处于运行状态并且加入队列成功
 5             if (runState == RUNNING && workQueue.offer(command)) {
 6                 if (runState != RUNNING || poolSize == 0)
 7                     ensureQueuedTaskHandled(command);
 8             }               //线程池不处于运行状态或者加入队列失败，则创建线程（创建的
是非核心线程）
 9             else if (!addIfUnderMaximumPoolSize(command))                   //
创建线程失败，则采取阻塞处理的方式
10                 reject(command); // is shutdown or saturated
11         }
12     }
```

2、创建线程的方法：addIfUnderCorePoolSize(command)

```
 1 private boolean addIfUnderCorePoolSize(Runnable firstTask) {
```

```
2         Thread t = null;
3         final ReentrantLock mainLock = this.mainLock;
4         mainLock.lock();
5         try {
6             if (poolSize < corePoolSize && runState == RUNNING)
7                 t = addThread(firstTask);
8         } finally {
9             mainLock.unlock();
10        }
11        if (t == null)
12            return false;
13        t.start();
14        return true;
15    }
```

我们重点来看第7行：

```
1 private Thread addThread(Runnable firstTask) {
2         Worker w = new Worker(firstTask);
3         Thread t = threadFactory.newThread(w);
4         if (t != null) {
5             w.thread = t;
6             workers.add(w);
7             int nt = ++poolSize;
8             if (nt > largestPoolSize)
9                 largestPoolSize = nt;
10        }
11        return t;
12    }
```

这里将线程封装成工作线程worker，并放入工作线程组里，worker类的方法run方法：

```
public void run() {
            try {
                Runnable task = firstTask;
                firstTask = null;
                while (task != null || (task = getTask()) != null) {
                    runTask(task);
                    task = null;
                }
            } finally {
                workerDone(this);
            }
        }
```

worker在执行完任务后，还会通过getTask方法循环获取工作队里里的任务来执行。

我们通过一个程序来观察线程池的工作原理：

1、创建一个线程

```
1 public class ThreadPoolTest implements Runnable
2 {
3     @Override
4     public void run()
5     {
```

```
 6         try
 7         {
 8             Thread.sleep(300);
 9         }
10         catch (InterruptedException e)
11         {
12             e.printStackTrace();
13         }
14     }
15 }
```

2、线程池循环运行16个线程：

```
 1 public static void main(String[] args)
 2     {
 3         LinkedBlockingQueue<Runnable> queue =
 4             new LinkedBlockingQueue<Runnable>(5);
 5         ThreadPoolExecutor threadPool = new ThreadPoolExecutor(5, 10, 60,
TimeUnit.SECONDS, queue);
 6         for (int i = 0; i < 16 ; i++)
 7         {
 8             threadPool.execute(
 9                 new Thread(new ThreadPoolTest(), "Thread".concat(i + "")));
10             System.out.println("线程池中活跃的线程数：  " +
threadPool.getPoolSize());
11             if (queue.size() > 0)
12             {
13                 System.out.println("----------------队列中阻塞的线程数" +
queue.size());
14             }
15         }
16         threadPool.shutdown();
17     }
```

执行结果：

```
线程池中活跃的线程数：  1
线程池中活跃的线程数：  2
线程池中活跃的线程数：  3
线程池中活跃的线程数：  4
线程池中活跃的线程数：  5
线程池中活跃的线程数：  5
----------------队列中阻塞的线程数1
线程池中活跃的线程数：  5
----------------队列中阻塞的线程数2
线程池中活跃的线程数：  5
----------------队列中阻塞的线程数3
线程池中活跃的线程数：  5
----------------队列中阻塞的线程数4
线程池中活跃的线程数：  5
----------------队列中阻塞的线程数5
线程池中活跃的线程数：  6
----------------队列中阻塞的线程数5
线程池中活跃的线程数：  7
----------------队列中阻塞的线程数5
```

```
线程池中活跃的线程数： 8
----------------队列中阻塞的线程数5
线程池中活跃的线程数： 9
----------------队列中阻塞的线程数5
线程池中活跃的线程数： 10
----------------队列中阻塞的线程数5
Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task
Thread[Thread15,5,main] rejected from
java.util.concurrent.ThreadPoolExecutor@232204a1[Running, pool size = 10, active
threads = 10, queued tasks = 5, completed tasks = 0]
    at
java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPool
Executor.java:2047)
    at
java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:823)
    at
java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1369)
    at test.ThreadTest.main(ThreadTest.java:17)
```

从结果可以观察出：

1、创建的线程池具体配置为：核心线程数量为5个；全部线程数量为10个；工作队列的长度为5。

2、我们通过queue.size()的方法来获取工作队列中的任务数。

3、运行原理：

刚开始都是在创建新的线程，达到核心线程数量5个后，新的任务进来后不再创建新的线程，而是将任务加入工作队列，任务队列到达上线5个后，新的任务又会创建新的普通线程，直到达到线程池最大的线程数量10个，后面的任务则根据配置的饱和策略来处理。我们这里没有具体配置，使用的是默认的配置AbortPolicy:直接抛出异常。

当然，为了达到我需要的效果，上述线程处理的任务都是利用休眠导致线程没有释放！！！

RejetedExecutionHandler：饱和策略

当队列和线程池都满了，说明线程池处于饱和状态，那么必须对新提交的任务采用一种特殊的策略来进行处理。这个策略默认配置是AbortPolicy，表示无法处理新的任务而抛出异常。JAVA提供了4中策略：

1、AbortPolicy：直接抛出异常

2、CallerRunsPolicy：只用调用所在的线程运行任务

3、DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。

4、DiscardPolicy：不处理，丢弃掉。

我们现在用第四种策略来处理上面的程序：

```
 1 public static void main(String[] args)
 2     {
 3         LinkedBlockingQueue<Runnable> queue =
 4             new LinkedBlockingQueue<Runnable>(3);
 5         RejectedExecutionHandler handler = new
ThreadPoolExecutor.DiscardPolicy();
 6
 7         ThreadPoolExecutor threadPool = new ThreadPoolExecutor(2, 5, 60,
TimeUnit.SECONDS, queue,handler);
 8         for (int i = 0; i < 9 ; i++)
 9         {
```

```
10              threadPool.execute(
11                  new Thread(new ThreadPoolTest(), "Thread".concat(i + "")));
12              System.out.println("线程池中活跃的线程数： " +
threadPool.getPoolSize());
13              if (queue.size() > 0)
14              {
15                  System.out.println("----------------队列中阻塞的线程数" +
queue.size());
16              }
17          }
18          threadPool.shutdown();
19      }
```

执行结果：

```
线程池中活跃的线程数： 1
线程池中活跃的线程数： 2
线程池中活跃的线程数： 2
----------------队列中阻塞的线程数1
线程池中活跃的线程数： 2
----------------队列中阻塞的线程数2
线程池中活跃的线程数： 2
----------------队列中阻塞的线程数3
线程池中活跃的线程数： 3
----------------队列中阻塞的线程数3
线程池中活跃的线程数： 4
----------------队列中阻塞的线程数3
线程池中活跃的线程数： 5
----------------队列中阻塞的线程数3
线程池中活跃的线程数： 5
----------------队列中阻塞的线程数3
```

这里采用了丢弃策略后，就没有再抛出异常，而是直接丢弃。在某些重要的场景下，可以采用记录日志或者存储到数据库中，而不应该直接丢弃。

设置策略有两种方式：

1、

```
 RejectedExecutionHandler handler = new ThreadPoolExecutor.DiscardPolicy();
 ThreadPoolExecutor threadPool = new ThreadPoolExecutor(2, 5, 60,
TimeUnit.SECONDS, queue,handler);
```

2、

```
  ThreadPoolExecutor threadPool = new ThreadPoolExecutor(2, 5, 60,
TimeUnit.SECONDS, queue);
  threadPool.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy());
```

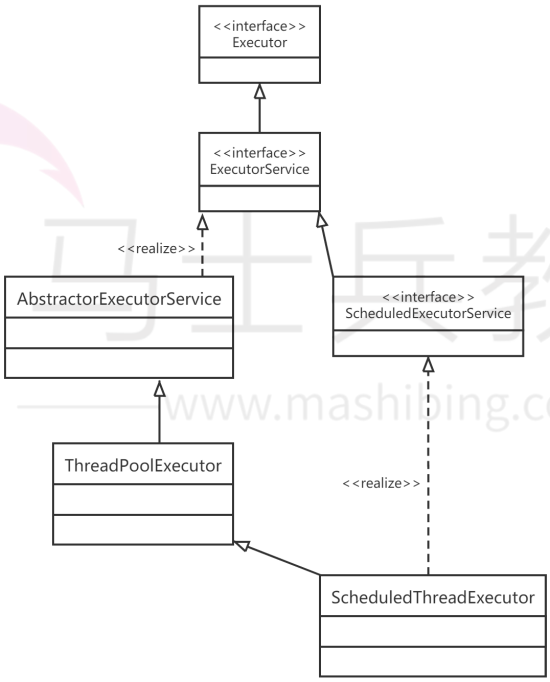
JAVA线程池原理详解二

Executor框架的两级调度模型

在HotSpot VM的模型中，JAVA线程被一对一映射为本地操作系统线程。JAVA线程启动时会创建一个本地操作系统线程，当JAVA线程终止时，对应的操作系统线程也被销毁回收，而操作系统会调度所有线程并将它们分配给可用的CPU。

在上层，JAVA程序会将应用分解为多个任务，然后使用应用级的调度器（Executor）将这些任务映射成固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。

Executor框架类图



在前面介绍的JAVA线程既是工作单元，也是执行机制。而在Executor框架中，我们将工作单元与执行机制分离开来。Runnable和Callable是工作单元（也就是俗称的任务），而执行机制由Executor来提供。这样一来Executor是基于生产者消费者模式的，提交任务的操作相当于生成者，执行任务的线程相当于消费者。

1、从类图上看，Executor接口是异步任务执行框架的基础，该框架能够支持多种不同类型的任务执行策略。

```
public interface Executor {

    void execute(Runnable command);
}
```

Executor接口就提供了一个执行方法，任务是Runnbale类型，不支持Callable类型。

2、ExecutorService接口实现了Executor接口，主要提供了关闭线程池和submit方法：

```
public interface ExecutorService extends Executor {

    List<Runnable> shutdownNow();


    boolean isTerminated();


    <T> Future<T> submit(Callable<T> task);

 }
```

另外该接口有两个重要的实现类：ThreadPoolExecutor与ScheduledThreadPoolExecutor。

其中ThreadPoolExecutor是线程池的核心实现类，用来执行被提交的任务；而ScheduledThreadPoolExecutor是一个实现类，可以在给定的延迟后运行任务，或者定期执行命令。

在上一篇文章中，我是使用ThreadPoolExecutor来通过给定不同的参数从而创建自己所需的线程池，但是在后面的工作中不建议这种方式，推荐使用Exectuors工厂方法来创建线程池

这里先来区别线程池和线程组（ThreadGroup与ThreadPoolExecutor）这两个概念：

a、线程组就表示一个线程的集合。

b、线程池是为线程的生命周期开销问题和资源不足问题提供解决方案，主要是用来管理线程。

Executors可以创建3种类型的ThreadPoolExecutor：SingleThreadExecutor、FixedThreadExecutor和CachedThreadPool

a、SingleThreadExecutor：单线程线程池

```
ExecutorService threadPool = Executors.newSingleThreadExecutor();
```

```
public static ExecutorService newSingleThreadExecutor() {
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>()));
    }
```

我们从源码来看可以知道，单线程线程池的创建也是通过ThreadPoolExecutor，里面的核心线程数和线程数都是1，并且工作队列使用的是无界队列。由于是单线程工作，每次只能处理一个任务，所以后面所有的任务都被阻塞在工作队列中，只能一个个任务执行。

b、FixedThreadExecutor：固定大小线程池

```
ExecutorService threadPool = Executors.newFixedThreadPool(5);
public static ExecutorService newFixedThreadPool(int nThreads) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
    }
```

这个与单线程类似，只是创建了固定大小的线程数量。

c、CachedThreadPool:无界线程池

```
ExecutorService threadPool = Executors.newCachedThreadPool();
public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
    }
```

无界线程池意味着没有工作队列，任务进来就执行，线程数量不够就创建，与前面两个的区别是：空闲的线程会被回收掉，空闲的时间是60s。这个适用于执行很多短期异步的小程序或者负载较轻的服务器。

Callable、Future、FutureTash详解

Callable与Future是在JAVA的后续版本中引入进来的，Callable类似于Runnable接口，实现Callable接口的类与实现Runnable的类都是可以被线程执行的任务。

三者之间的关系：

Callable是Runnable封装的异步运算任务。

Future用来保存Callable异步运算的结果

FutureTask封装Future的实体类

1、Callable与Runnbale的区别

a、Callable定义的方法是call，而Runnable定义的方法是run。

b、call方法有返回值，而run方法是没有返回值的。

c、call方法可以抛出异常，而run方法不能抛出异常。

2、Future

Future表示异步计算的结果，提供了以下方法，主要是判断任务是否完成、中断任务、获取任务执行结果

```
 1 public interface Future<V> {
 2 
 3     boolean cancel(boolean mayInterruptIfRunning);
 4 
 5     boolean isCancelled();
 6 
 7     boolean isDone();
 8 
 9     V get() throws InterruptedException, ExecutionException;
10 
11     V get(long timeout, TimeUnit unit)
12         throws InterruptedException, ExecutionException, TimeoutException;
13 }
```

3、FutureTask

可取消的异步计算，此类提供了对Future的基本实现，仅在计算完成时才能获取结果，如果计算尚未完成，则阻塞get方法。

```
public class FutureTask<V> implements RunnableFuture<V>
public interface RunnableFuture<V> extends Runnable, Future<V>
```

FutureTask不仅实现了Future接口，还实现了Runnable接口，所以不仅可以将FutureTask当成一个任务交给Executor来执行，还可以通过Thread来创建一个线程。

Callable与FutureTask

定义一个callable的任务：

```
 1 public class MyCallableTask implements Callable<Integer>
 2 {
 3     @Override
 4     public Integer call()
 5         throws Exception
 6     {
 7         System.out.println("callable do somothing");
 8         Thread.sleep(5000);
 9         return new Random().nextInt(100);
10     }
11 }
```

```
 1 public class CallableTest
 2 {
 3     public static void main(String[] args) throws Exception
 4     {
 5         Callable<Integer> callable = new MyCallableTask();
 6         FutureTask<Integer> future = new FutureTask<Integer>(callable);
 7         Thread thread = new Thread(future);
 8         thread.start();
 9         Thread.sleep(100);
10         //尝试取消对此任务的执行
11         future.cancel(true);
12         //判断是否在任务正常完成前取消
13         System.out.println("future is cancel:" + future.isCancelled());
14         if(!future.isCancelled())
15         {
16             System.out.println("future is cancelled");
17         }
18         //判断任务是否已完成
19         System.out.println("future is done:" + future.isDone());
20         if(!future.isDone())
21         {
22             System.out.println("future get=" + future.get());
23         }
24         else
25         {
26             //任务已完成
27             System.out.println("task is done");
28         }
29     }
30 }
```

执行结果：

```
callable do somothing
future is cancel:true
future is done:true
task is done
```

这个DEMO主要是通过调用FutureTask的状态设置的方法，演示了状态的变迁。

a、第11行，尝试取消对任务的执行，该方法如果由于任务已完成、已取消则返回false，如果能够取消还未完成的任务，则返回true，该DEMO中由于任务还在休眠状态，所以可以取消成功。

```
future.cancel(true);
```

b、第13行，判断任务取消是否成功：如果在任务正常完成前将其取消，则返回true

```
System.out.println("future is cancel:" + future.isCancelled());
```

c、第19行，判断任务是否完成：如果任务完成，则返回true，以下几种情况都属于任务完成：正常终止、异常或者取消而完成。

我们的DEMO中，任务是由于取消而导致完成。

```
 System.out.println("future is done:" + future.isDone());
```

d、在第22行，获取异步线程执行的结果，我这个DEMO中没有执行到这里，需要注意的是，future.get方法会阻塞当前线程， 直到任务执行完成返回结果为止。

```
System.out.println("future get=" + future.get());
```

Callable与Future

```
public class CallableThread implements Callable<String>
{
    @Override
    public String call()
        throws Exception
    {
        System.out.println("进入Call方法，开始休眠，休眠时间为：" +
System.currentTimeMillis());
        Thread.sleep(10000);
        return "今天停电";
    }

    public static void main(String[] args) throws Exception
    {
        ExecutorService es = Executors.newSingleThreadExecutor();
        Callable<String> call = new CallableThread();
        Future<String> fu = es.submit(call);
        es.shutdown();
        Thread.sleep(5000);
        System.out.println("主线程休眠5秒，当前时间" + System.currentTimeMillis());
        String str = fu.get();
        System.out.println("Future已拿到数据，str=" + str + ";当前时间为：" +
System.currentTimeMillis());
    }
}
```

执行结果：

```
进入Call方法，开始休眠，休眠时间为：1478606602676
主线程休眠5秒，当前时间1478606608676
Future已拿到数据，str=今天停电;当前时间为：1478606612677
```

这里的future是直接扔到线程池里面去执行的。由于要打印任务的执行结果，所以从执行结果来看，主线程虽然休眠了5s，但是从Call方法执行到拿到任务的结果，这中间的时间差正好是10s，说明get方法会阻塞当前线程直到任务完成。

**通过FutureTask也可以达到同样的效果：

```
public static void main(String[] args) throws Exception
    {
      ExecutorService es = Executors.newSingleThreadExecutor();
      Callable<String> call = new CallableThread();
      FutureTask<String> task = new FutureTask<String>(call);
      es.submit(task);
      es.shutdown();
      Thread.sleep(5000);
      System.out.println("主线程等待5秒，当前时间为：" + System.currentTimeMillis());
      String str = task.get();
      System.out.println("Future已拿到数据，str=" + str + ";当前时间为：" +
System.currentTimeMillis());
    }
```

以上的组合可以给我们带来这样的一些变化：

如有一种场景中，方法A返回一个数据需要10s,A方法后面的代码运行需要20s，但是这20s的执行过程中，只有后面10s依赖于方法A执行的结果。如果与以往一样采用同步的方式，势必会有10s的时间被浪费，如果采用前面两种组合，则效率会提高：

1、先把A方法的内容放到Callable实现类的call()方法中

2、在主线程中通过线程池执行A任务

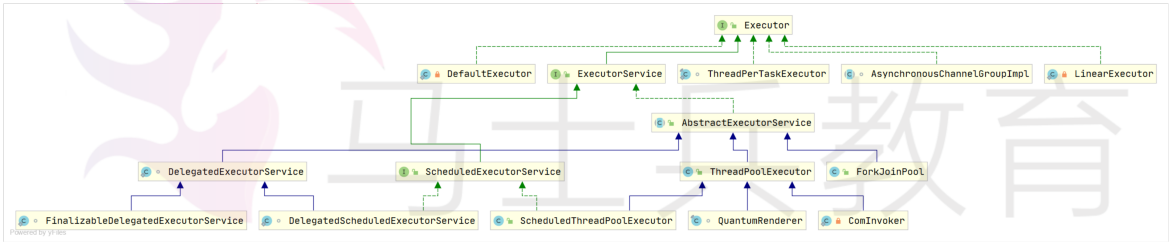
3、执行后面方法中10秒不依赖方法A运行结果的代码

4、获取方法A的运行结果，执行后面方法中10秒依赖方法A运行结果的代码

这样代码执行效率一下子就提高了，程序不必卡在A方法处。

创建线程池的几种方式:

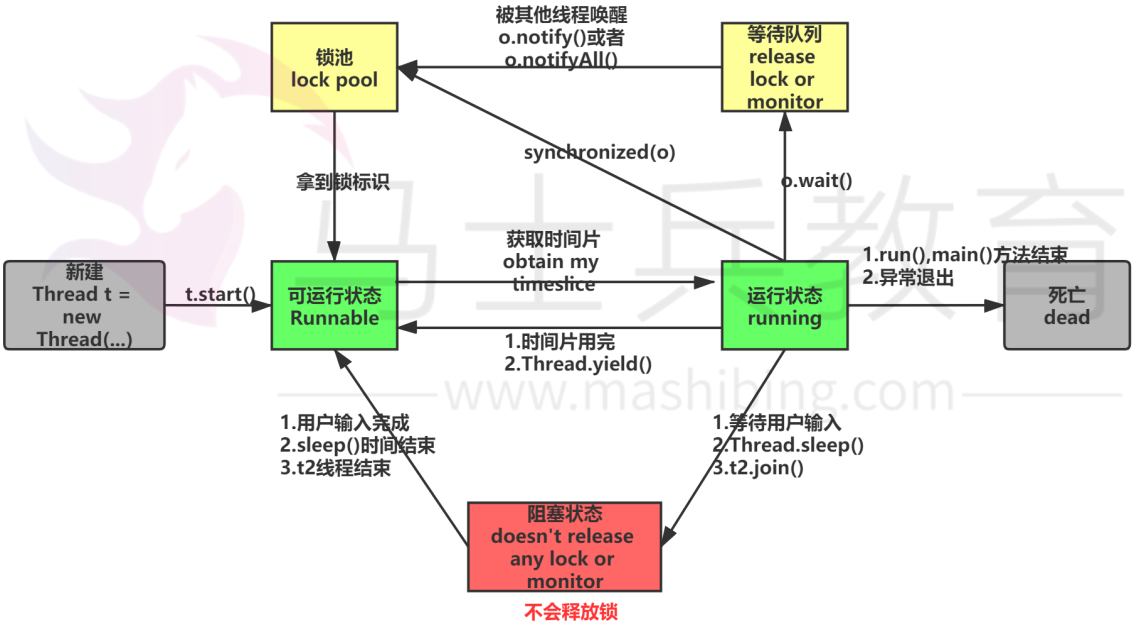
ThreadPoolExecutor、ThreadScheduledExecutor、ForkJoinPool



马士兵教育
www.mashibing.com

线程的生命周期，什么时候会出现僵死进程；

线程生命周期



僵死进程是指子进程退出时，父进程并未对其发出的SIGCHLD信号进行适当处理，导致子 进程停留在僵死状态等待其父进程为其收尸，这个状态下的子进程就是僵死进程。

说说线程安全问题，什么是线程安全，如何实现线程安全；

线程安全 - 如果线程执行过程中不会产生共享资源的冲突，则线程安全。

线程不安全 - 如果有多个线程同时在操作主内存中的变量，则线程不安全

实现线程安全的三种方式

1）互斥同步

临界区：syncronized、ReentrantLock
信号量 semaphore
互斥量　mutex

2）非阻塞同步

```
CAS（Compare And Swap）
```

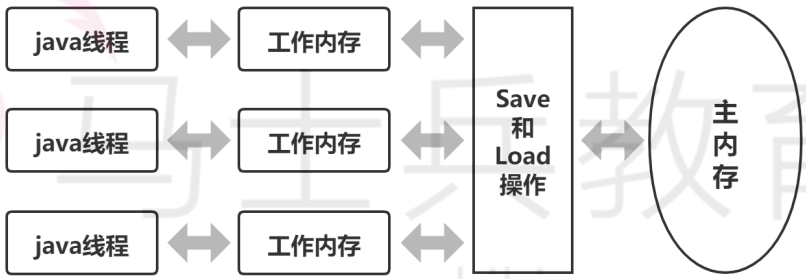
3）无同步方案

```
可重入代码
使用Threadlocal 类来包装共享变量，做到每个线程有自己的copy
线程本地存储
```

Java 多线程安全机制

在开始讨论java多线程安全机制之前，首先从内存模型来了解一下什么是多线程的安全性。

我们都知道java的内存模型中有主内存和线程的工作内存之分，主内存上存放的是线程共享的变量（实例字段，静态字段和构成数组的元素），线程的工作内存是线程私有的空间，存放的是线程私有的变量（方法参数与局部变量）。线程在工作的时候如果要操作主内存上的共享变量，为了获得更好的执行性能并不是直接去修改主内存而是会在线程私有的工作内存中创建一份变量的拷贝（缓存），在工作内存上对变量的拷贝修改之后再把修改的值刷回到主内存的变量中去，JVM提供了8中原子操作来完成这一过程：lock, unlock, read, load, use, assign, store, write。深入理解java虚拟机-jvm最高特性与实践这本书中有一个图很好的表示了线程，主内存和工作内存之间的关系：



马士兵教育
www.mashibing.com

如果只有一个线程当然不会有什么问题，但是如果有多个线程同时在操作主内存中的变量，因为8种操作的非连续性和线程抢占cpu执行的机制就会带来冲突的问题，也就是多线程的安全问题。线程安全的定义就是：如果线程执行过程中不会产生共享资源的冲突就是线程安全的。

Java里面一般用以下几种机制保证线程安全：

1.互斥同步锁（悲观锁）

1）Synchorized

2）ReentrantLock

互斥同步锁也叫做阻塞同步锁，特征是会对没有获取锁的线程进行阻塞。

要理解互斥同步锁，首选要明白什么是互斥什么是同步。简单的说互斥就是非你即我，同步就是顺序访问。互斥同步锁就是以互斥的手段达到顺序访问的目的。操作系统提供了很多互斥机制比如信号量，互斥量，临界区资源等来控制在某一个时刻只能有一个或者一组线程访问同一个资源。

Java里面的互斥同步锁就是Synchorized和ReentrantLock，前者是由语言级别实现的互斥同步锁，理解和写法简单但是机制笨拙，在JDK6之后性能优化大幅提升，即使在竞争激烈的情况下也能保持一个和ReentrantLock相差不多的性能，所以JDK6之后的程序选择不应该再因为性能问题而放弃synchorized。ReentrantLock是API层面的互斥同步锁，需要程序自己打开并在finally中关闭锁，和synchorized相比更加的灵活，体现在三个方面：等待可中断，公平锁以及绑定多个条件。但是如果程序猿对ReentrantLock理解不够深刻，或者忘记释放lock，那么不仅不会提升性能反而会带来额外的问题。另外synchorized是JVM实现的，可以通过监控工具来监控锁的状态，遇到异常JVM会自动释放掉锁。而ReentrantLock必须由程序主动的释放锁。

互斥同步锁都是可重入锁，好处是可以保证不会死锁。但是因为涉及到核心态和用户态的切换，因此比较消耗性能。JVM开发团队在JDK5-JDK6升级过程中采用了很多锁优化机制来优化同步无竞争情况下锁的性能。比如：自旋锁和适应性自旋锁，轻量级锁，偏向锁，锁粗化和锁消除。

2.非阻塞同步锁

1) 原子类（CAS）

非阻塞同步锁也叫乐观锁，相比悲观锁来说，它会先进行资源在工作内存中的更新，然后根据与主存中旧值的对比来确定在此期间是否有其他线程对共享资源进行了更新，如果旧值与期望值相同，就认为没有更新，可以把新值写回内存，否则就一直重试直到成功。它的实现方式依赖于处理器的机器指令：CAS（Compare And Swap）

JUC中提供了几个Automic类以及每个类上的原子操作就是乐观锁机制

不激烈情况下，性能比synchronized略逊，而激烈的时候，也能维持常态。激烈的时候，Atomic的性能会优于ReentrantLock一倍左右。但是其有一个缺点，就是只能同步一个值，一段代码中只能出现一个Atomic的变量，多于一个同步无效。因为他不能在多个Atomic之间同步。

非阻塞锁是不可重入的，否则会造成死锁。

3.无同步方案

1）可重入代码

在执行的任何时刻都可以中断-重入执行而不会产生冲突。特点就是不会依赖堆上的共享资源

2）ThreadLocal/Volaitile

线程本地的变量，每个线程获取一份共享变量的拷贝，单独进行处理。

3) 线程本地存储

如果一个共享资源一定要被多线程共享，可以尽量让一个线程完成所有的处理操作，比如生产者消费者模式中，一般会让一个消费者完成对队列上资源的消费。典型的应用是基于请求-应答模式的web服务器的设计

创建线程池有哪几个核心参数？ 如何合理配置线程池的大小?

1）核心参数

```
public ThreadPoolExecutor(int corePoolSize,  // 核心线程数量大小
            int maximumPoolSize, // 线程池最大容纳线程数
            long keepAliveTime, // 线程空闲后的存活时长
            TimeUnit unit,
            //缓存异步任务的队列 //用来构造线程池里的worker线程
            BlockingQueue<Runnable> workQueue,
            ThreadFactory threadFactory,
            //线程池任务满载后采取的任务拒绝策略
            RejectedExecutionHandler handler)
```

2) 核心说明

1. 当线程池中线程数量小于 corePoolSize 则创建线程，并处理请求。
2. 当线程池中线程数量大于等于 corePoolSize 时，则把请求放入 workQueue 中,随着线程池 中的核心线程们不断执行任务，只要线程池中有空闲的核心线程，线程池就从 workQueue 中取 任务并处理。

3. 当 workQueue 已存满，放不下新任务时则新建非核心线程入池，并处理请求直到线程数目 达到 maximumPoolSize（最大线程数量设置值）。
4. 如果线程池中线程数大于 maximumPoolSize 则使用 RejectedExecutionHandler 来进行任 务拒绝处理。

3)线程池大小分配

线程池究竟设置多大要看你的线程池执行的什么任务了，CPU密集型、IO密集型、混合型，任 务类型不同，设置的方式也不一样。

任务一般分为：CPU密集型、IO密集型、混合型，对于不同类型的任务需要分配不同大小的线 程池。

3.1）CPU密集型

尽量使用较小的线程池，一般Cpu核心数+1

3.2）IO密集型

方法一：可以使用较大的线程池，一般CPU核心数 * 2

方法二：（线程等待时间与线程CPU时间之比 + 1）* CPU数目

3.3）混合型　可以将任务分为CPU密集型和IO密集型，然后分别使用不同的线程池去处理，按情况而定

关于线程池的执行原则及配置参数详解

在软件开发中，池一直都是一种非常优秀的设计思想，通过建立池可以有效的利用系统资源，节约系统性能。Java 中的线程池就是一种非常好的实现，从 JDK 1.5 开始 Java 提供了一个线程工厂 Executors 用来生成线程池，通过 Executors 可以方便的生成不同类型的线程池。但是要更好的理解使用线程池，就需要了解线程池的配置参数意义以及线程池的具体工作机制。

下面先介绍一下线程池的好处以及创建方式，接着会着重介绍关于线程池的执行原则以及构造方法的参数详解。

线程池的好处

- 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

创建线程池

```
//参数初始化
private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();
//核心线程数量大小
private static final int corePoolSize = Math.max(2, Math.min(CPU_COUNT - 1, 4));
//线程池最大容纳线程数
private static final int maximumPoolSize = CPU_COUNT * 2 + 1;
//线程空闲后的存活时长
private static final int keepAliveTime = 30;

//任务过多后，存储任务的一个阻塞队列
BlockingQueue<Runnable>  workQueue = new SynchronousQueue<>();
```

```
//线程的创建工厂
ThreadFactory threadFactory = new ThreadFactory() {
    private final AtomicInteger mCount = new AtomicInteger(1);

    public Thread newThread(Runnable r) {
        return new Thread(r, "AdvacnedAsyncTask #" + mCount.getAndIncrement());
    }
};

//线程池任务满载后采取的任务拒绝策略
RejectedExecutionHandler rejectHandler = new
ThreadPoolExecutor.DiscardOldestPolicy();

//线程池对象，创建线程
ThreadPoolExecutor mExecute = new ThreadPoolExecutor(
        corePoolSize,
        maximumPoolSize,
        keepAliveTime,
        TimeUnit.SECONDS,
        workQueue,
        threadFactory,
        rejectHandler
);
```

具体参数介绍

- corePoolSize

> 线程池的核心线程数。在没有设置 allowCoreThreadTimeOut 为 true 的情况下，核心线程会在线程池中一直存活，即使处于闲置状态。

maximumPoolSize

> 线程池所能容纳的最大线程数。当活动线程(核心线程+非核心线程)达到这个数值后，后续任务将会根据 RejectedExecutionHandler 来进行拒绝策略处理。

- keepAliveTime

> `非核心线程`闲置时的超时时长。超过该时长，**非核心线程**就会被回收。若线程池通设置核心线程也允许 timeOut，即 allowCoreThreadTimeOut 为 true，则该时长同样会作用于核心线程，在超过 aliveTime 时，核心线程也会被回收，AsyncTask 配置的线程池就是这样设置的。

- unit

> keepAliveTime 时长对应的单位。

- workQueue

> 线程池中的任务队列，通过线程池的 execute() 方法提交的 Runnable 对象会存储在该队列中。

- ThreadFactory

> 线程工厂，功能很简单，就是为线程池提供创建新线程的功能。这是一个接口，可以通过自定义，做一些自定义线程名的操作。

- RejectedExecutionHandler

> 当任务无法被执行时(超过线程最大容量 maximum 并且 workQueue 已经被排满了)的处理策略，这里有四种任务拒绝类型。

线程池工作原则

- 1、当线程池中线程数量小于 corePoolSize 则创建线程，并处理请求。
- 2、当线程池中线程数量大于等于 corePoolSize 时，则把请求放入 workQueue 中,随着线程池中的核心线程们不断执行任务，只要线程池中有空闲的核心线程，线程池就从 workQueue 中取任务并处理。
- 3 、当 workQueue 已存满，放不下新任务时则新建非核心线程入池，并处理请求直到线程数目达到 maximumPoolSize（最大线程数量设置值）。
- 4、如果线程池中线程数大于 maximumPoolSize 则使用 RejectedExecutionHandler 来进行任务拒绝处理。

任务队列 BlockingQueue

任务队列 workQueue 是用于存放不能被及时处理掉的任务的一个队列，它是 一个 BlockingQueue 类型。

> 关于 BlockingQueue，虽然它是 Queue 的子接口，但是它的主要作用并不是容器，而是作为线程同步的工具，他有一个特征，当生产者试图向 BlockingQueue 放入(put)元素，如果队列已满，则该线程被阻塞；当消费者试图从 BlockingQueue 取出(take)元素，如果队列已空，则该线程被阻塞。(From 疯狂Java讲义)

任务拒绝类型

- ThreadPoolExecutor.AbortPolicy:

> 当线程池中的数量等于最大线程数时抛 java.util.concurrent.RejectedExecutionException 异常，涉及到该异常的任务也不会被执行，线程池默认的拒绝策略就是该策略。

- ThreadPoolExecutor.DiscardPolicy():

> 当线程池中的数量等于最大线程数时,默默丢弃不能执行的新加任务，不报任何异常。

- ThreadPoolExecutor.CallerRunsPolicy():

> 当线程池中的数量等于最大线程数时，重试添加当前的任务；它会自动重复调用execute()方法。

- ThreadPoolExecutor.DiscardOldestPolicy():

> 当线程池中的数量等于最大线程数时,抛弃线程池中工作队列头部的任务(即等待时间最久的任务)，并执行新传入的任务。

java线程池如何合理的设置大小

线程池究竟设置多大要看你的线程池执行的什么任务了，CPU密集型、IO密集型、混合型，任务类型不同，设置的方式也不一样

任务一般分为：CPU密集型、IO密集型、混合型，对于不同类型的任务需要分配不同大小的线程池

1、CPU密集型

尽量使用较小的线程池，一般Cpu核心数+1

因为CPU密集型任务CPU的使用率很高，若开过多的线程，只能增加线程上下文的切换次数，带来额外的开销

2、IO密集型

方法一：可以使用较大的线程池，一般CPU核心数 * 2

IO密集型CPU使用率不高，可以让CPU等待IO的时候处理别的任务，充分利用cpu时间

方法二：线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。

下面举个例子：
比如平均每个线程CPU运行时间为0.5s，而线程等待时间（非CPU运行时间，比如IO）为1.5s，CPU核心数为8，那么根据上面这个公式估算得到：((0.5+1.5)/0.5)*8=32。这个公式进一步转化为：

最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）* CPU数目

3、混合型

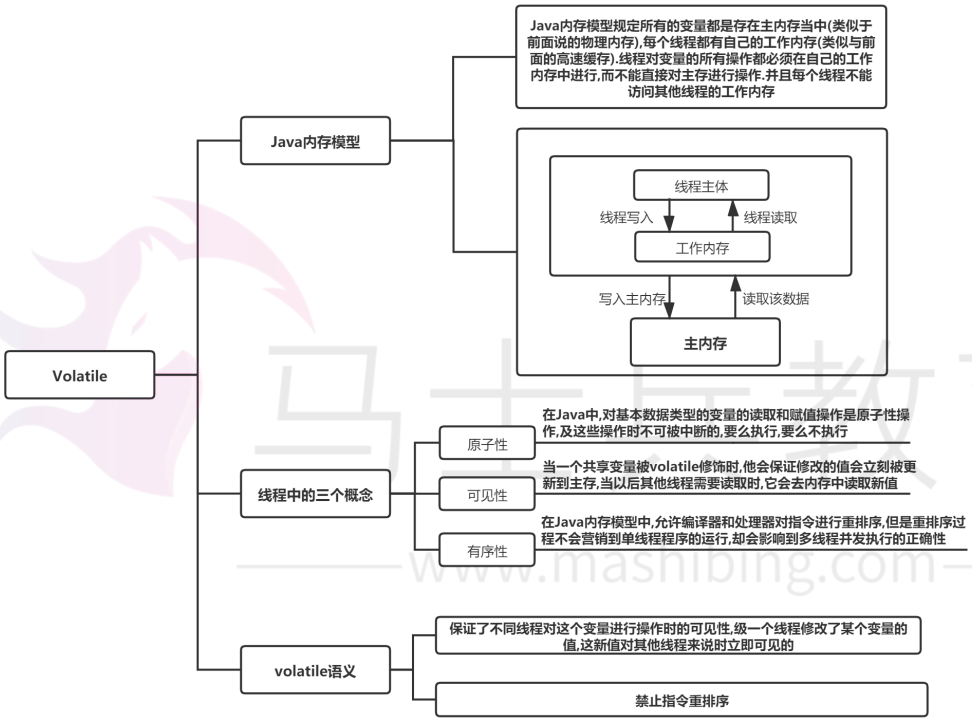
可以将任务分为CPU密集型和IO密集型，然后分别使用不同的线程池去处理，按情况而定

volatile、ThreadLocal的使用场景和原理;

volatile原理

volatile变量进行写操作时，JVM 会向处理器发送一条 Lock 前缀的指令，将这个变量所在缓 存行的数据写会到系统内存。

Lock 前缀指令实际上相当于一个内存屏障（也成内存栅栏），它确保指令重排序时不会把其 后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内 存屏障这句指令时，在它前面的操作已经全部完成。



volatile的适用场景

1）状态标志,如：初始化或请求停机

2）一次性安全发布，如：单列模式

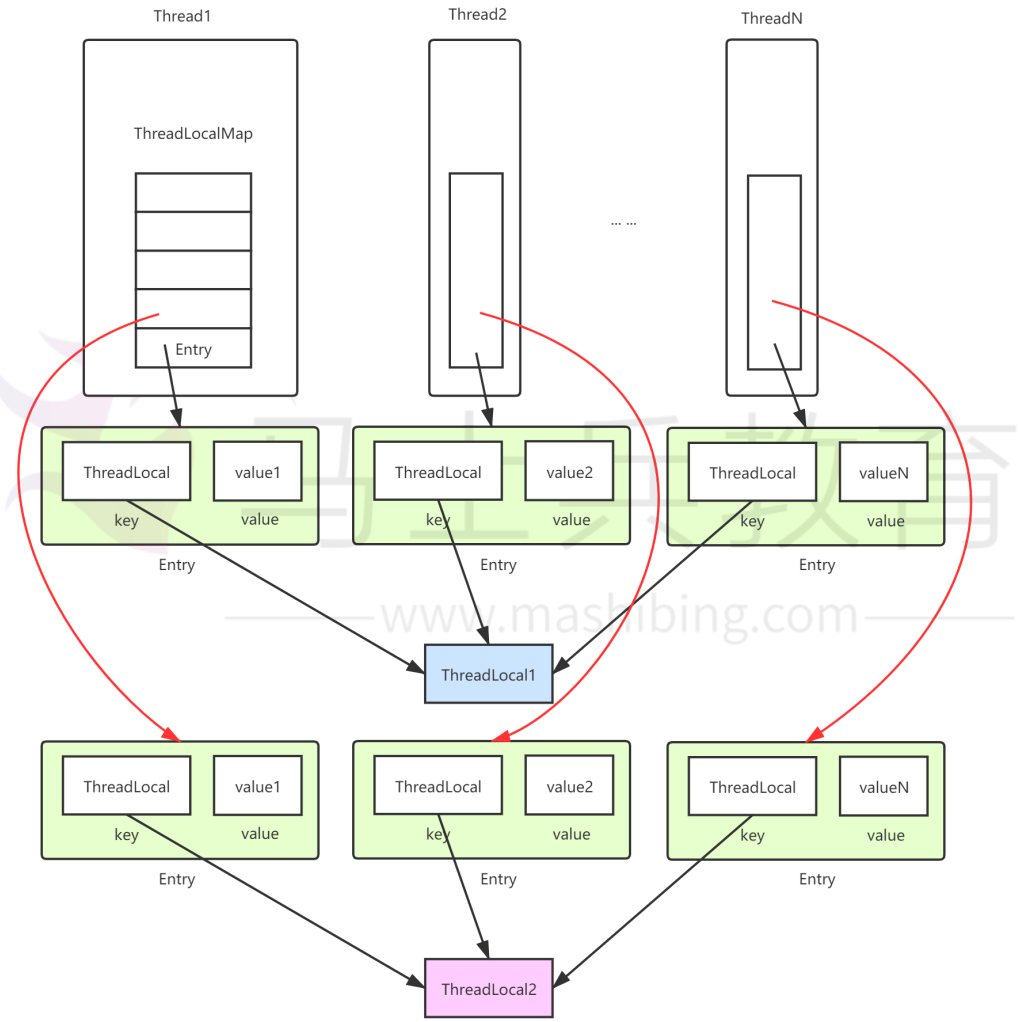
3）独立观察，如：定期更新某个值

4）“volatile bean” 模式

5) 开销较低的“读 - 写锁”策略，如：计数器

ThreadLocal原理

ThreadLocal是用来维护本线程的变量的，并不能解决共享变量的并发问题。ThreadLocal是 各线程将值存入该线程的map中，以ThreadLocal自身作为key，需要用时获得的是该线程之前 存入的值。如果存入的是共享变量，那取出的也是共享变量，并发问题还是存在的。



ThreadLocal的适用场景

场景：数据库连接、Session管理

volatile的适用场景

模式 #1：状态标志

也许实现 volatile 变量的规范使用仅仅是使用一个布尔状态标志，用于指示发生了一个重要的一次性事件，例如完成初始化或请求停机。

```
1. volatile boolean shutdownRequested;
2.
3. ...
4.
5. public void shutdown() {
6.   shutdownRequested = true;
7. }
8.
9. public void doWork() {
10.   while (!shutdownRequested) {
11.     // do stuff
12.   }
13. }
```

线程1执行doWork()的过程中，可能有另外的线程2调用了shutdown，所以boolean变量必须是volatile。

而如果使用 `synchronized` 块编写循环要比使用 volatile 状态标志编写麻烦很多。由于 volatile 简化了编码，并且状态标志并不依赖于程序内任何其他状态，因此此处非常适合使用 volatile。

这种类型的状态标记的一个公共特性是：**通常只有一种状态转换**；`shutdownRequested` 标志从 `false` 转换为 `true`，然后程序停止。这种模式可以扩展到来回转换的状态标志，但是只有在转换周期不被察觉的情况下才能扩展（从 `false` 到 `true`，再转换到 `false`）。此外，还需要某些原子状态转换机制，例如原子变量。

模式 #2：一次性安全发布（one-time safe publication）

在缺乏同步的情况下，可能会遇到某个对象引用的更新值（由另一个线程写入）和该对象状态的旧值同时存在。

这就是造成著名的双重检查锁定（double-checked-locking）问题的根源，其中对象引用在没有同步的情况下进行读操作，产生的问题是您可能会看到一个更新的引用，但是仍然会通过该引用看到不完全构造的对象。

```
1. //注意volatile！！！！！！！！！！！！！！！
2. private volatile static Singleton instace;
3.
4. public static Singleton getInstance(){
5.   //第一次null检查
6.   if(instance == null){
7.     synchronized(Singleton.class) {  //1
8.       //第二次null检查
9.       if(instance == null){     //2
10.         instance = new Singleton();//3
11.       }
12.     }
13.   }
14.   return instance;
```

如果不用volatile，则因为内存模型允许所谓的"无序写入"，可能导致失败。**——某个线程可能会获得一个未完全初始化的实例。**

考察上述代码中的 //3 行。此行代码创建了一个 Singleton 对象并初始化变量 instance 来引用此对象。这行代码的问题是：**在Singleton 构造函数体执行之前，变量instance 可能成为非 null 的！**什么？这一说法可能让您始料未及，但事实确实如此。

在解释这个现象如何发生前，请先暂时接受这一事实，我们先来考察一下双重检查锁定是如何被破坏的。假设上述代码执行以下事件序列：

1. 线程 1 进入 getInstance() 方法。
2. 由于 instance 为 null，线程 1 在 //1 处进入synchronized 块。
3. 线程 1 前进到 //3 处，但在构造函数执行之前，使实例成为非null。
4. 线程 1 被线程 2 预占。
5. 线程 2 检查实例是否为 null。因为实例不为 null，线程 2 将instance 引用返回，返回一个构造完整但**部分初始化**了的Singleton 对象。
6. 线程 2 被线程 1 预占。
7. 线程 1 通过运行 Singleton 对象的构造函数并将引用返回给它，来完成对该对象的初始化。

模式 #3：独立观察（independent observation）

安全使用 volatile 的另一种简单模式是：定期 “发布” 观察结果供程序内部使用。【例如】假设有一种环境传感器能够感觉环境温度。一个后台线程可能会每隔几秒读取一次该传感器，并更新包含当前文档的 volatile 变量。然后，其他线程可以读取这个变量，从而随时能够看到最新的温度值。

使用该模式的另一种应用程序就是收集程序的统计信息。【例】如下代码展示了身份验证机制如何记忆最近一次登录的用户的名字。将反复使用 `lastUser` 引用来发布值，以供程序的其他部分使用。

```
1. public class UserManager {
2.   public volatile String lastUser; //发布的信息
3.
4.   public boolean authenticate(String user, String password) {
5.     boolean valid = passwordIsValid(user, password);
6.     if (valid) {
7.       User u = new User();
8.       activeUsers.add(u);
9.       lastUser = user;
10.     }
11.     return valid;
12.   }
13. }
```

模式 #4：“volatile bean” 模式

volatile bean 模式的基本原理是：很多框架为易变数据的持有者（例如 `HttpSession`）提供了容器，但是放入这些容器中的对象必须是线程安全的。

在 volatile bean 模式中，JavaBean 的所有数据成员都是 volatile 类型的，并且 getter 和 setter 方法必须非常普通——即不包含约束！

```
1. @ThreadSafe
2. public class Person {
3.   private volatile String firstName;
4.   private volatile String lastName;
5.   private volatile int age;
6.
7.   public String getFirstName() { return firstName; }
8.   public String getLastName() { return lastName; }
```

```
9.   public int getAge() { return age; }
10.
11.   public void setFirstName(String firstName) {
12.     this.firstName = firstName;
13.   }
14.
15.   public void setLastName(String lastName) {
16.     this.lastName = lastName;
17.   }
18.
19.   public void setAge(int age) {
20.     this.age = age;
21.   }
22. }
```

模式 #5：开销较低的“读 - 写锁”策略

如果读操作远远超过写操作，您可以结合使用**内部锁**和 **volatile 变量**来减少公共代码路径的开销。

如下显示的线程安全的计数器，使用 `synchronized` 确保增量操作是原子的，并使用 `volatile` 保证当前结果的可见性。如果更新不频繁的话，该方法可实现更好的性能，因为读路径的开销仅仅涉及 volatile 读操作，这通常要优于一个无竞争的锁获取的开销。

```
1. @ThreadSafe
2. public class CheesyCounter {
3.   // Employs the cheap read-write lock trick
4.   // All mutative operations MUST be done with the 'this' lock held
5.   @GuardedBy("this") private volatile int value;
6.
7.   //读操作，没有synchronized，提高性能
8.   public int getValue() {
9.     return value;
10.   }
11.
12.   //写操作，必须synchronized。因为x++不是原子操作
13.   public synchronized int increment() {
14.     return value++;
15.   }
```

使用锁进行所有变化的操作，使用 volatile 进行只读操作。
其中，锁一次只允许一个线程访问值，volatile 允许多个线程执行读操作

正确使用 Volatile 变量

volatile 变量使用指南

Java 语言中的 volatile 变量可以被看作是一种 “程度较轻的 `synchronized`”；与 `synchronized` 块相比，volatile 变量所需的编码较少，并且运行时开销也较少，但是它所能实现的功能也仅是 `synchronized` 的一部分。本文介绍了几种有效使用 volatile 变量的模式，并强调了几种不适合使用 volatile 变量的情形。

锁提供了两种主要特性：*互斥（mutual exclusion）* 和 *可见性（visibility）*。互斥即一次只允许一个线程持有某个特定的锁，因此可使用该特性实现对共享数据的协调访问协议，这样，一次就只有一个线程能够使用该共享数据。可见性要更加复杂一些，它必须确保释放锁之前对共享数据做出的更改对于随后获得该锁的另一个线程是可见的 —— 如果没有同步机制提供的这种可见性保证，线程看到的共享变量可能是修改前的值或不一致的值，这将引发许多严重问题。

Volatile 变量

Volatile 变量具有 `synchronized` 的可见性特性，但是不具备原子特性。这就是说线程能够自动发现 volatile 变量的最新值。Volatile 变量可用于提供线程安全，但是只能应用于非常有限的一组用例：多个变量之间或者某个变量的当前值与修改后值之间没有约束。因此，单独使用 volatile 还不足以实现计数器、互斥锁或任何具有与多个变量相关的不变式（Invariants）的类（例如 “start <=end”）。

出于简易性或可伸缩性的考虑，您可能倾向于使用 volatile 变量而不是锁。当使用 volatile 变量而非锁时，某些习惯用法（idiom）更加易于编码和阅读。此外，volatile 变量不会像锁那样造成线程阻塞，因此也很少造成可伸缩性问题。在某些情况下，如果读操作远远大于写操作，volatile 变量还可以提供优于锁的性能优势。

正确使用 volatile 变量的条件

您只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

- 对变量的写操作不依赖于当前值。
- 该变量没有包含在具有其他变量的不变式中。

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

第一个条件的限制使 volatile 变量不能用作线程安全计数器。虽然增量操作（`x++`）看上去类似一个单独操作，实际上它是一个由读取 - 修改 - 写入操作序列组成的组合操作，必须以原子方式执行，而 volatile 不能提供必须的原子特性。实现正确的操作需要使 `x` 的值在操作期间保持不变，而 volatile 变量无法实现这点。（然而，如果将值调整为只从单个线程写入，那么可以忽略第一个条件。）

大多数编程情形都会与这两个条件的其中之一冲突，使得 volatile 变量不能像 `synchronized` 那样普遍适用于实现线程安全。清单 1 显示了一个非线程安全的数值范围类。它包含了一个不变式 —— 下界总是小于或等于上界。

清单 1. 非线程安全的数值范围类

```
@NotThreadSafe
public class NumberRange {
    private int lower, upper;

    public int getLower() { return lower; }
    public int getUpper() { return upper; }

    public void setLower(int value) {
        if (value > upper)
            throw new IllegalArgumentException(...);
        lower = value;
    }

    public void setUpper(int value) {
        if (value < lower)
            throw new IllegalArgumentException(...);
        upper = value;
```

```
    }
}
```

这种方式限制了范围的状态变量，因此将 `lower` 和 upper 字段定义为 volatile 类型不能够充分实现类的线程安全；从而仍然需要使用同步。否则，如果凑巧两个线程在同一时间使用不一致的值执行 `setLower` 和 `setUpper` 的话，则会使范围处于不一致的状态。例如，如果初始状态是 `(0, 5)`，同一时间内，线程 A 调用 `setLower(4)` 并且线程 B 调用 `setUpper(3)`，显然这两个操作交叉存入的值是不符合条件的，那么两个线程都会通过用于保护不变式的检查，使得最后的范围值是 `(4, 3)` —— 一个无效值。至于针对范围的其他操作，我们需要使 `setLower()` 和 `setUpper()` 操作原子化 —— 而将字段定义为 volatile 类型是无法实现这一目的的。

性能考虑

使用 volatile 变量的主要原因是其简易性：在某些情形下，使用 volatile 变量要比使用相应的锁简单得多。使用 volatile 变量次要原因是其性能：某些情况下，volatile 变量同步机制的性能要优于锁。

很难做出准确、全面的评价，例如 “X 总是比 Y 快”，尤其是对 JVM 内在的操作而言。（例如，某些情况下 VM 也许能够完全删除锁机制，这使得我们难以抽象地比较 `volatile` 和 `synchronized` 的开销。）就是说，在目前大多数的处理器架构上，volatile 读操作开销非常低 —— 几乎和非 volatile 读操作一样。而 volatile 写操作的开销要比非 volatile 写操作多很多，因为要保证可见性需要实现内存界定（Memory Fence），即便如此，volatile 的总开销仍然要比锁获取低。

volatile 操作不会像锁一样造成阻塞，因此，在能够安全使用 volatile 的情况下，volatile 可以提供一些优于锁的可伸缩特性。如果读操作的次数要远远超过写操作，与锁相比，volatile 变量通常能够减少同步的性能开销。

正确使用 volatile 的模式

很多并发性专家事实上往往引导用户远离 volatile 变量，因为使用它们要比使用锁更加容易出错。然而，如果谨慎地遵循一些良好定义的模式，就能够在很多场合内安全地使用 volatile 变量。要始终牢记使用 volatile 的限制 —— 只有在状态真正独立于程序内其他内容时才能使用 volatile —— 这条规则能够避免将这些模式扩展到不安全的用例。

模式 #1：状态标志

也许实现 volatile 变量的规范使用仅仅是使用一个布尔状态标志，用于指示发生了一个重要的一次性事件，例如完成初始化或请求停机。

很多应用程序包含了一种控制结构，形式为 “在还没有准备好停止程序时再执行一些工作”，如清单 2 所示：

清单 2. 将 volatile 变量作为状态标志使用

```
volatile boolean shutdownRequested;
...

public void shutdown() { shutdownRequested = true; }

public void doWork() {
    while (!shutdownRequested) {
        // do stuff
    }
}
```

很可能会从循环外部调用 `shutdown()` 方法 —— 即在另一个线程中 —— 因此，需要执行某种同步来确保正确实现 `shutdownRequested` 变量的可见性。（可能会从 JMX 侦听程序、GUI 事件线程中的操作侦听程序、通过 RMI 、通过一个 Web 服务等调用）。然而，使用 `synchronized` 块编写循环要比使用清单 2 所示的 volatile 状态标志编写麻烦很多。由于 volatile 简化了编码，并且状态标志并不依赖于程序内任何其他状态，因此此处非常适合使用 volatile。

这种类型的状态标记的一个公共特性是：通常只有一种状态转换；`shutdownRequested` 标志从 `false` 转换为 `true`，然后程序停止。这种模式可以扩展到来回转换的状态标志，但是只有在转换周期不被察觉的情况下才能扩展（从 `false` 到 `true`，再转换到 `false`）。此外，还需要某些原子状态转换机制，例如原子变量。

模式 #2：一次性安全发布（one-time safe publication）

缺乏同步会导致无法实现可见性，这使得确定何时写入对象引用而不是原语值变得更加困难。在缺乏同步的情况下，可能会遇到某个对象引用的更新值（由另一个线程写入）和该对象状态的旧值同时存在。（这就是造成著名的双重检查锁定（double-checked-locking）问题的根源，其中对象引用在没有同步的情况下进行读操作，产生的问题是您可能会看到一个更新的引用，但是仍然会通过该引用看到不完全构造的对象）。

实现安全发布对象的一种技术就是将对象引用定义为 volatile 类型。清单 3 展示了一个示例，其中后台线程在启动阶段从数据库加载一些数据。其他代码在能够利用这些数据时，在使用之前将检查这些数据是否曾经发布过。

清单 3. 将 volatile 变量用于一次性安全发布

```
public class BackgroundFloobleLoader {
    public volatile Flooble theFlooble;
    public void initInBackground() {
        // do lots of stuff
        theFlooble = new Flooble(); // this is the only write to theFlooble
    }
}

public class SomeOtherClass {
    public void doWork() {
        while (true) {
            // do some stuff...
            // use the Flooble, but only if it is ready
            if (floobleLoader.theFlooble != null)
doSomething(floobleLoader.theFlooble);
        }
    }
}
```

如果 `theFlooble` 引用不是 volatile 类型，`doWork()` 中的代码在解除对 `theFlooble` 的引用时，将会得到一个不完全构造的 `Flooble`。

该模式的一个必要条件是：被发布的对象必须是线程安全的，或者是有效的不可变对象（有效不可变意味着对象的状态在发布之后永远不会被修改）。volatile 类型的引用可以确保对象的发布形式的可见性，但是如果对象的状态在发布后将发生更改，那么就需要额外的同步。

模式 #3：独立观察（independent observation）

安全使用 volatile 的另一种简单模式是：定期 “发布” 观察结果供程序内部使用。例如，假设有一种环境传感器能够感觉环境温度。一个后台线程可能会每隔几秒读取一次该传感器，并更新包含当前文档的 volatile 变量。然后，其他线程可以读取这个变量，从而随时能够看到最新的温度值。

使用该模式的另一种应用程序就是收集程序的统计信息。清单 4 展示了身份验证机制如何记忆最近一次登录的用户的名字。将反复使用 `lastUser` 引用来发布值，以供程序的其他部分使用。

清单 4. 将 volatile 变量用于多个独立观察结果的发布

```
public class UserManager {
    public volatile String lastUser;

    public boolean authenticate(String user, String password) {
        boolean valid = passwordIsValid(user, password);
        if (valid) {
            User u = new User();
            activeUsers.add(u);
            lastUser = user;
        }
        return valid;
    }
}
```

该模式是前面模式的扩展；将某个值发布以在程序内的其他地方使用，但是与一次性事件的发布不同，这是一系列独立事件。这个模式要求被发布的值是有效不可变的 —— 即值的状态在发布后不会更改。使用该值的代码需要清楚该值可能随时发生变化。

模式 #4：“volatile bean” 模式

volatile bean 模式适用于将 JavaBeans 作为“荣誉结构”使用的框架。在 volatile bean 模式中，JavaBean 被用作一组具有 getter 和/或 setter 方法 的独立属性的容器。volatile bean 模式的基本原理是：很多框架为易变数据的持有者（例如 `HttpSession`）提供了容器，但是放入这些容器中的对象必须是线程安全的。

在 volatile bean 模式中，JavaBean 的所有数据成员都是 volatile 类型的，并且 getter 和 setter 方法必须非常普通 —— 除了获取或设置相应的属性外，不能包含任何逻辑。此外，对于对象引用的数据成员，引用的对象必须是有效不可变的。（这将禁止具有数组值的属性，因为当数组引用被声明为 `volatile` 时，只有引用而不是数组本身具有 volatile 语义）。对于任何 volatile 变量，不变式或约束都不能包含 JavaBean 属性。清单 5 中的示例展示了遵守 volatile bean 模式的 JavaBean：

清单 5. 遵守 volatile bean 模式的 Person 对象

```
@ThreadSafe
public class Person {
    private volatile String firstName;
    private volatile String lastName;
    private volatile int age;

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
```

```
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

volatile 的高级模式

前面几节介绍的模式涵盖了大部分的基本用例，在这些模式中使用 volatile 非常有用并且简单。这一节将介绍一种更加高级的模式，在该模式中，volatile 将提供性能或可伸缩性优势。

volatile 应用的的高级模式非常脆弱。因此，必须对假设的条件仔细证明，并且这些模式被严格地封装了起来，因为即使非常小的更改也会损坏您的代码！同样，使用更高级的 volatile 用例的原因是它能够提升性能，确保在开始应用高级模式之前，真正确定需要实现这种性能获益。需要对这些模式进行权衡，放弃可读性或可维护性来换取可能的性能收益 —— 如果您不需要提升性能（或者不能够通过一个严格的测试程序证明您需要它），那么这很可能是一次糟糕的交易，因为您很可能会得不偿失，换来的东西要比放弃的东西价值更低。

模式 #5：开销较低的读 - 写锁策略

目前为止，您应该了解了 volatile 的功能还不足以实现计数器。因为 `++x` 实际上是三种操作（读、添加、存储）的简单组合，如果多个线程凑巧试图同时对 volatile 计数器执行增量操作，那么它的更新值有可能会丢失。

然而，如果读操作远远超过写操作，您可以结合使用内部锁和 volatile 变量来减少公共代码路径的开销。清单 6 中显示的线程安全的计数器使用 `synchronized` 确保增量操作是原子的，并使用 `volatile` 保证当前结果的可见性。如果更新不频繁的话，该方法可实现更好的性能，因为读路径的开销仅仅涉及 volatile 读操作，这通常要优于一个无竞争的锁获取的开销。

清单 6. 结合使用 volatile 和 synchronized 实现 “开销较低的读 - 写锁”

```
@ThreadSafe
public class CheesyCounter {
    // Employs the cheap read-write lock trick
    // All mutative operations MUST be done with the 'this' lock held
@GuardedBy("this") private volatile int value;

    public int getValue() { return value; }

    public synchronized int increment() {
        return value++;
    }
}
```

之所以将这种技术称之为 “开销较低的读 - 写锁” 是因为您使用了不同的同步机制进行读写操作。因为本例中的写操作违反了使用 volatile 的第一个条件，因此不能使用 volatile 安全地实现计数器 —— 您必须使用锁。然而，您可以在读操作中使用 volatile 确保当前值的*可见性*，因此可以使用锁进行所有变化的操作，使用 volatile 进行只读操作。其中，锁一次只允许一个线程访问值，volatile 允许多个线程执行读操作，因此当使用 volatile 保证读代码路径时，要比使用锁执行全部代码路径获得更高的共享度 —— 就像读 - 写操作一样。然而，要随时牢记这种模式的弱点：如果超越了该模式的最基本应用，结合这两个竞争的同步机制将变得非常困难。

结束语

与锁相比，Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 volatile 的使用条件 —— 即变量真正独立于其他变量和自己以前的值 —— 在某些情况下可以使用 `volatile` 代替 `synchronized` 来简化代码。然而，使用 `volatile` 的代码往往比使用锁的代码更加容易出错。本文介绍的模式涵盖了可以使用 `volatile` 代替 `synchronized` 的最常见的一些用例。遵循这些模式（注意使用时不要超过各自的限制）可以帮助您安全地实现大多数用例，使用 volatile 变量获得更佳性能。

ThreadLocal什么时候会出现OOM的情况？为什么？

ThreadLocal变量是维护在Thread内部的，这样的话只要我们的线程不退出，对象的引用就会 一直存在。当线程退出时，Thread类会进行一些清理工作，其中就包含ThreadLocalMap， Thread调用exit方法如下：

```
/**
 * This method is called by the system to give a Thread
 * a chance to clean up before it actually exits.
 */
private void exit() {
    if (group != null) {
        group.threadTerminated(this);
        group = null;
    }
    /* Aggressively null out all reference fields: see bug 4006245 */
    target = null;
    /* Speed the release of some of these resources */
    threadLocals = null;
    inheritableThreadLocals = null;
    inheritedAccessControlContext = null;
    blocker = null;
    uncaughtExceptionHandler = null;
}
```

ThreadLocal在没有线程池使用的情况下，正常情况下不会存在内存泄露，但是如果使用了线程 池的话，就依赖于线程池的实现，如果线程池不销毁线程的话，那么就会存在内存泄露。

ThreadLocal可以说是笔试面试的常客，每逢面试基本都会问到，关于ThreadLocal的原理以及不正当的使用造成的OOM内存溢出的问题，值得花时间仔细研究一下其原理。这一篇主要学习一下ThreadLocal的原理，在下一篇会深入理解一下OOM内存溢出的原理和最佳实践。

ThreadLocal很容易让人望文生义，想当然地认为是一个“本地线程”。其实，ThreadLocal并不是一个Thread，而是Thread的一个局部变量，也许把它命名为`ThreadLocalVariable`更容易让人理解一些。

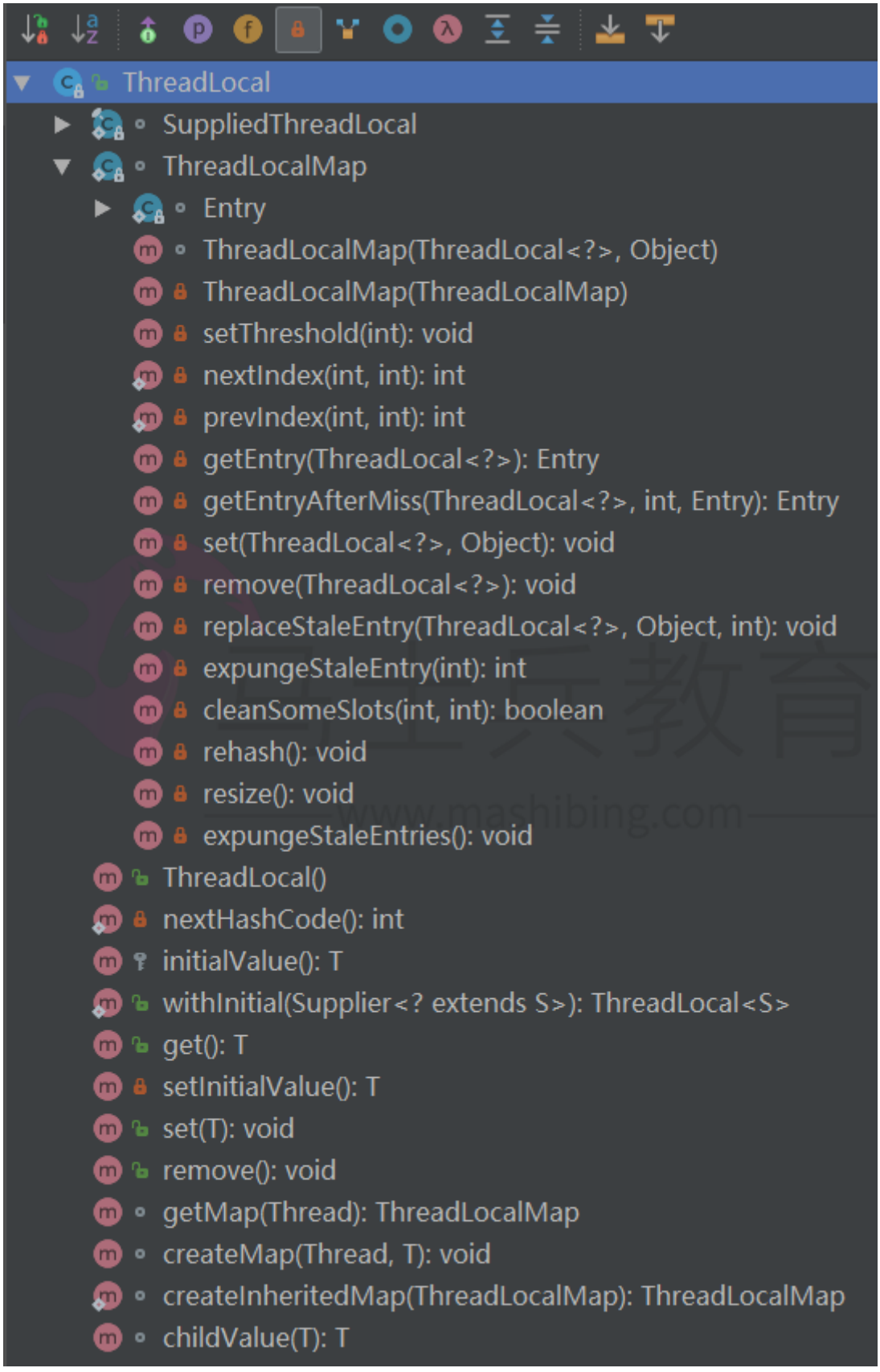
当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

ThreadLocal 的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。

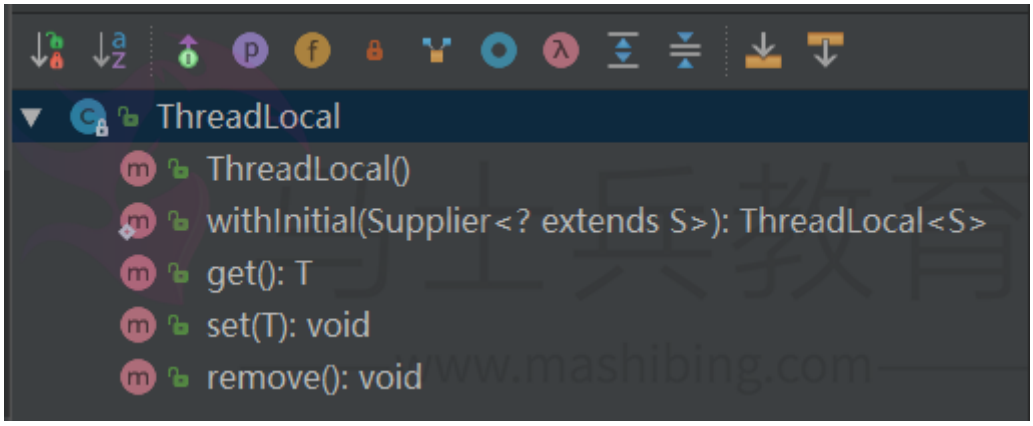
从线程的角度看，目标变量就像是线程的本地变量，这也是类名中“Local”所要表达的意思。

ThreadLocal全部方法和内部类

ThreadLocal全部方法和内部类结构如下：



ThreadLocal公有的方法就四个，分别为：**get、set、remove、intiValue**:



也就是说我们平时使用的时候关心的是这四个方法。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢?

其实实现的思路很简单：在ThreadLocal类中有一个static声明的Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本。我们自己就可以提供一个简单的实现版本：

```
public class SimpleThreadLocal<T> {

    /**
     * Key为线程对象，Value为传入的值对象
     */
    private static Map<Thread, T> valueMap = Collections.synchronizedMap(new
HashMap<Thread, T>());

    /**
     * 设值
     * @param value Map键值对的value
     */
    public void set(T value) {
        valueMap.put(Thread.currentThread(), value);
    }

    /**
     * 取值
     * @return
     */
    public T get() {
        Thread currentThread = Thread.currentThread();
        //返回当前线程对应的变量
        T t = valueMap.get(currentThread);
        //如果当前线程在Map中不存在，则将当前线程存储到Map中
        if (t == null && !valueMap.containsKey(currentThread)) {
            t = initialValue();
            valueMap.put(currentThread, t);
        }
        return t;
    }

    public void remove() {
        valueMap.remove(Thread.currentThread());
    }

    public T initialValue() {
```

```
        return null;
    }

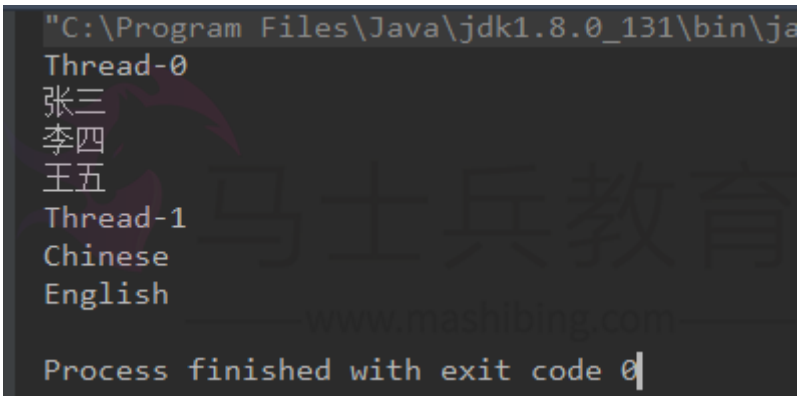
    public static void main(String[] args) {

        SimpleThreadLocal<List<String>> threadLocal = new SimpleThreadLocal<>();

        new Thread(() -> {
            List<String> params = new ArrayList<>(3);
            params.add("张三");
            params.add("李四");
            params.add("王五");
            threadLocal.set(params);
            System.out.println(Thread.currentThread().getName());
            threadLocal.get().forEach(param -> System.out.println(param));
        }).start();

        new Thread(() -> {
            try {
                Thread.sleep(1000);
                List<String> params = new ArrayList<>(2);
                params.add("Chinese");
                params.add("English");
                threadLocal.set(params);
                System.out.println(Thread.currentThread().getName());
                threadLocal.get().forEach(param -> System.out.println(param));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

运行结果：



虽然上面的代码清单中的这个ThreadLocal实现版本显得比较简单粗，但其目的主要在与呈现JDK中所提供的ThreadLocal类在实现上的思路。

ThreadLocal源码分析

1、线程局部变量在Thread中的位置

既然是**线程局部变量**，那么理所当然就应该存储在自己的线程对象中，我们可以从 Thread 的源码中找到**线程局部变量存储**的地方：

```
public class Thread implements Runnable {
```

```
    /* Make sure registerNatives is the first thing <clinit> does. */
    private static native void registerNatives();
    static {
        registerNatives();
    }

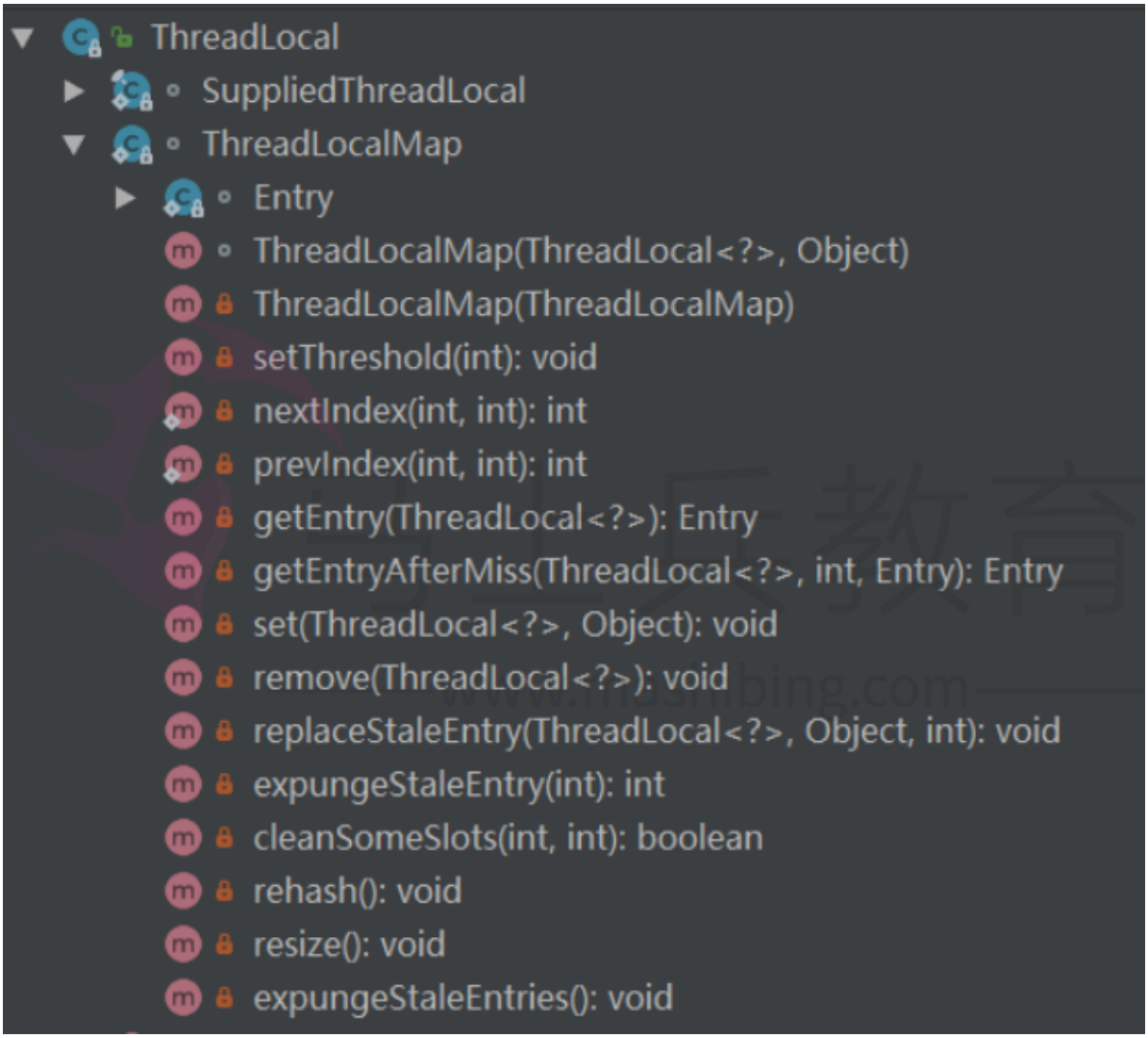
    //省略其他代码

    /* ThreadLocal values pertaining to this thread. This map is maintained
     * by the ThreadLocal class. */
    ThreadLocal.ThreadLocalMap threadLocals = null;

    /*
     * InheritableThreadLocal values pertaining to this thread. This map is
     * maintained by the InheritableThreadLocal class.
     */
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
}12345678910111213141516171819
```

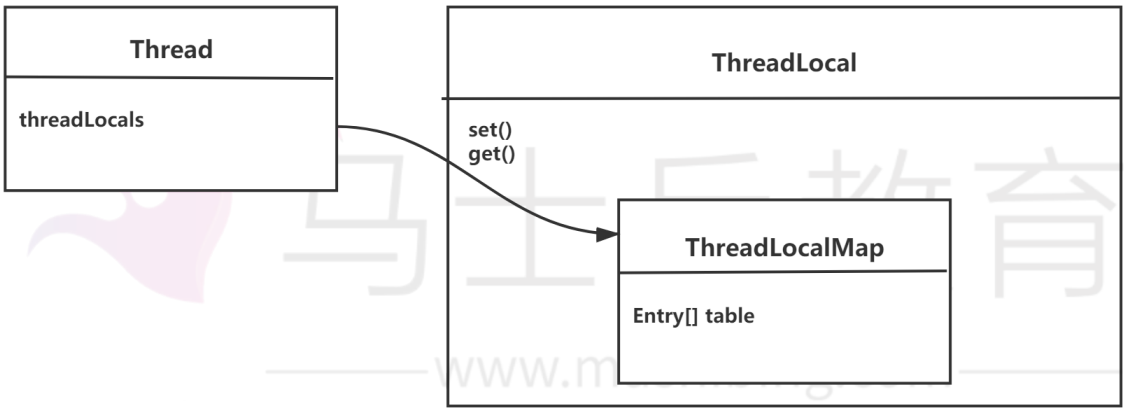
我们可以看到线程局部变量是存储在Thread对象的 `threadLocals` 属性中，而 `threadLocals` 属性是一个 `ThreadLocal.ThreadLocalMap` 对象。

ThreadLocalMap为ThreadLocal的**静态内部类**，如下图所示：

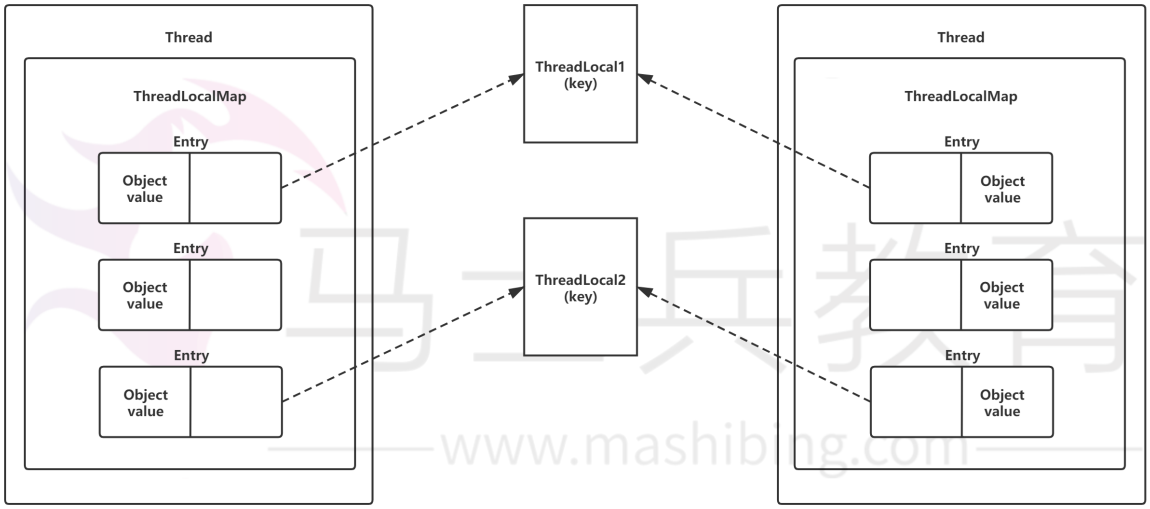


2、Thread和ThreadLocalMap的关系

Thread和ThreadLocalMap的关系，先看下边这个简单的图，可以看出Thread中的 `threadLocals` 就是ThreadLocal中的ThreadLocalMap：



到这里应该大致能够感受到上述三者之间微妙的关系，再看一个复杂点的图：

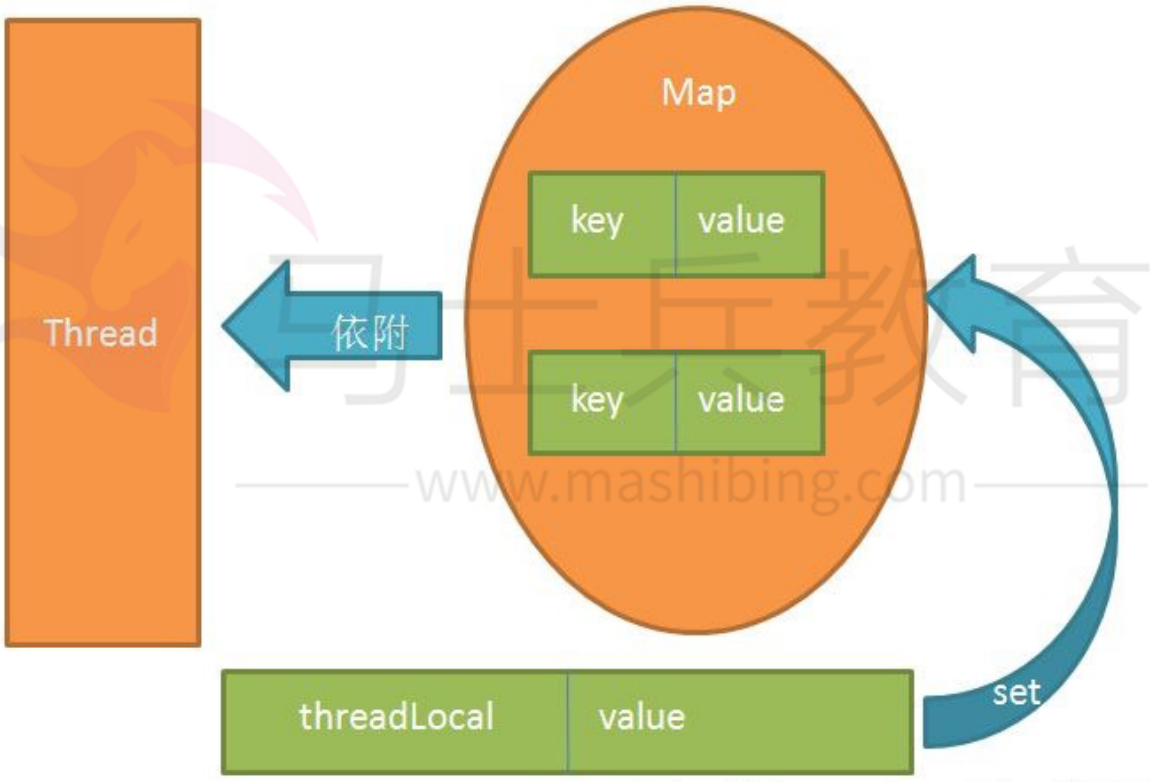


可以看出每个`thread`实例都有一个`ThreadLocalMap`。在上图中的一个Thread的这个ThreadLocalMap中分别存放了3个Entry，默认一个ThreadLocalMap初始化了16个Entry，每一个Entry对象存放的是一个ThreadLocal变量对象。

再简单一点的说就是：一个Thread中只有一个ThreadLocalMap，一个ThreadLocalMap中可以有多个ThreadLocal对象，其中一个ThreadLocal对象对应一个ThreadLocalMap中的一个Entry（也就是说：一个Thread可以依附有多个ThreadLocal对象）。

再看一张图片，应该可以更好的理解，如下图：

```
ThreadLocal<Test> threadLocal = new ThreadLocal<Test>();
threadLocal.set(value);
```



这里的Map其实是ThreadLocalMap。

3、ThreadLocalMap与WeakReference

`ThreadLocalMap`从字面上就可以看出这是一个保存`ThreadLocal`对象的map(其实是以它为Key)，不过是经过了两层包装的ThreadLocal对象：

（1）第一层包装是使用 `WeakReference<ThreadLocal<?>>` 将`ThreadLocal`对象变成一个弱引用的对象；

（2）第二层包装是定义了一个专门的类 Entry 来扩展 `WeakReference<ThreadLocal<?>>`：

```
static class ThreadLocalMap {

    /**
     * The entries in this hash map extend WeakReference, using
     * its main ref field as the key (which is always a
     * ThreadLocal object).  Note that null keys (i.e. entry.get()
     * == null) mean that the key is no longer referenced, so the
     * entry can be expunged from table.  Such entries are referred to
     * as "stale entries" in the code that follows.
     */
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
```

类 Entry 很显然是一个**保存map键值对**的实体，`ThreadLocal<?>`为key, 要保存的线程局部变量的值为`value`。`super(k)`调用的`WeakReference`的构造函数，表示将`ThreadLocal<?>`对象转换成弱引用对象，用做key。

4、ThreadLocalMap 的构造函数

```
/**
 * Construct a new map initially containing (firstKey, firstValue).
 * ThreadLocalMaps are constructed lazily, so we only create
 * one when we have at least one entry to put in it.
 */
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    table = new Entry[INITIAL_CAPACITY];
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
    table[i] = new Entry(firstKey, firstValue);
    size = 1;
    setThreshold(INITIAL_CAPACITY);
}
```

可以看出，ThreadLocalMap这个map的实现是使用一个数组`private Entry[] table`来保存键值对的实体，初始大小为16，`ThreadLocalMap`自己实现了如何从`key`到 value 的映射：

```
int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);1
```

使用一个`static`的原子属性`AtomicInteger nextHashCode`，通过每次增加`HASH_INCREMENT = 0x61c88647`，然后`& (INITIAL_CAPACITY - 1)`取得在数组`private Entry[] table`中的索引。

```
public class ThreadLocal<T> {
    /**
     * ThreadLocals rely on per-thread linear-probe hash maps attached
     * to each thread (Thread.threadLocals and
     * inheritableThreadLocals).  The ThreadLocal objects act as keys,
     * searched via threadLocalHashCode.  This is a custom hash code
     * (useful only within ThreadLocalMaps) that eliminates collisions
```

```
     * in the common case where consecutively constructed ThreadLocals
     * are used by the same threads, while remaining well-behaved in
     * less common cases.
     */
    private final int threadLocalHashCode = nextHashCode();

    /**
     * The next hash code to be given out. Updated atomically. Starts at
     * zero.
     */
    private static AtomicInteger nextHashCode =
        new AtomicInteger();

    /**
     * The difference between successively generated hash codes - turns
     * implicit sequential thread-local IDs into near-optimally spread
     * multiplicative hash values for power-of-two-sized tables.
     */
    private static final int HASH_INCREMENT = 0x61c88647;

    /**
     * Returns the next hash code.
     */
    private static int nextHashCode() {
        return nextHashCode.getAndAdd(HASH_INCREMENT);
    }
    //省略其它代码
}
```

总的来说，ThreadLocalMap是一个类似HashMap的集合，只不过自己实现了寻址，也没有HashMap中的put方法，而是set方法等区别。

ThreadLocal的set方法

```
/**
 * Sets the current thread's copy of this thread-local variable
 * to the specified value.  Most subclasses will have no need to
 * override this method, relying solely on the {@link #initialValue}
 * method to set the values of thread-locals.
 *
 * @param value the value to be stored in the current thread's copy of
 *        this thread-local.
 */
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

由于每个thread实例都有一个ThreadLocalMap，所以在进行set的时候，首先根据Thread.currentThread（）获取当前线程，然后根据当前线程t，调用getMap(t)获取ThreadLocalMap对象，
如果是第一次设置值，ThreadLocalMap对象是空值，所以会进行初始化操作，即调用

`createMap(t,value)`方法：

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    table = new Entry[INITIAL_CAPACITY];
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
    table[i] = new Entry(firstKey, firstValue);
    size = 1;
    setThreshold(INITIAL_CAPACITY);
}
```

```
/**
 * Set the resize threshold to maintain at worst a 2/3 load factor.
 */
private void setThreshold(int len) {
    threshold = len * 2 / 3;
}
```

即是调用上述的构造方法进行构造，这里仅仅是初始化了16个元素的引用数组，并没有初始化16个Entry 对象。而是一个线程中有多少个线程局部对象要保存，那么就初始化多少个 Entry 对象来保存它们。
到了这里，我们可以思考一下，为什么要这样实现了。

1、为什么要用 ThreadLocalMap 来保存线程局部对象呢?

原因是一个线程拥有的的局部对象可能有很多，这样实现的话，那么不管你一个线程拥有多少个局部变量，都是使用同一个 ThreadLocalMap 来保存的，ThreadLocalMap 中 `private Entry[] table` 的初始大小是16。超过容量的2/3时，会扩容。

```
/**
 * Sets the current thread's copy of this thread-local variable
 * to the specified value.  Most subclasses will have no need to
 * override this method, relying solely on the {@Link #initialValue}
 * method to set the values of thread-locals.
 *
 * @param value the value to be stored in the current thread's copy of
 *        this thread-local.
 */
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

然后在回到如果map不为空的情况，会调用`map.set(this, value);`方法，我们看到是以当前 thread 的引用为 key, 获得 `ThreadLocalMap` ，然后调用 `map.set(this, value);` 保存进 `private Entry[] table` ：

```
    private void set(ThreadLocal<?> key, Object value) {
```

```
        // We don't use a fast path as with get() because it is at
        // least as common to use set() to create new entries as
        // it is to replace existing ones, in which case, a fast
        // path would fail more often than not.

        Entry[] tab = table;
        int len = tab.length;
        int i = key.threadLocalHashCode & (len - 1);

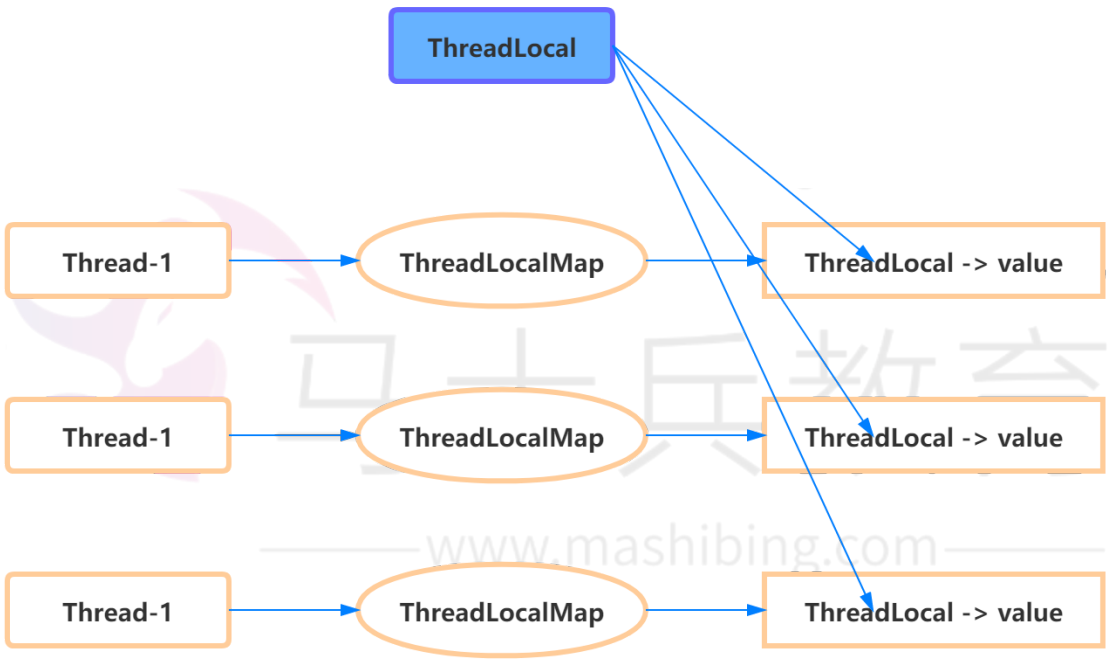
        for (Entry e = tab[i];
             e != null;
             e = tab[i = nextIndex(i, len)]) {
            ThreadLocal<?> k = e.get();

            if (k == key) {
                e.value = value;
                return;
            }

            if (k == null) {
                replaceStaleEntry(key, value, i);
                return;
            }
        }

        tab[i] = new Entry(key, value);
        int sz = ++size;
        if (!cleanSomeSlots(i, sz) && sz >= threshold)
            rehash();
    }
```

可以看到，`set(T value)`方法为每个Thread对象都创建了一个ThreadLocalMap，并且将value放入ThreadLocalMap中，ThreadLocalMap作为Thread对象的成员变量保存。那么可以用下图来表示ThreadLocal在存储value时的关系。



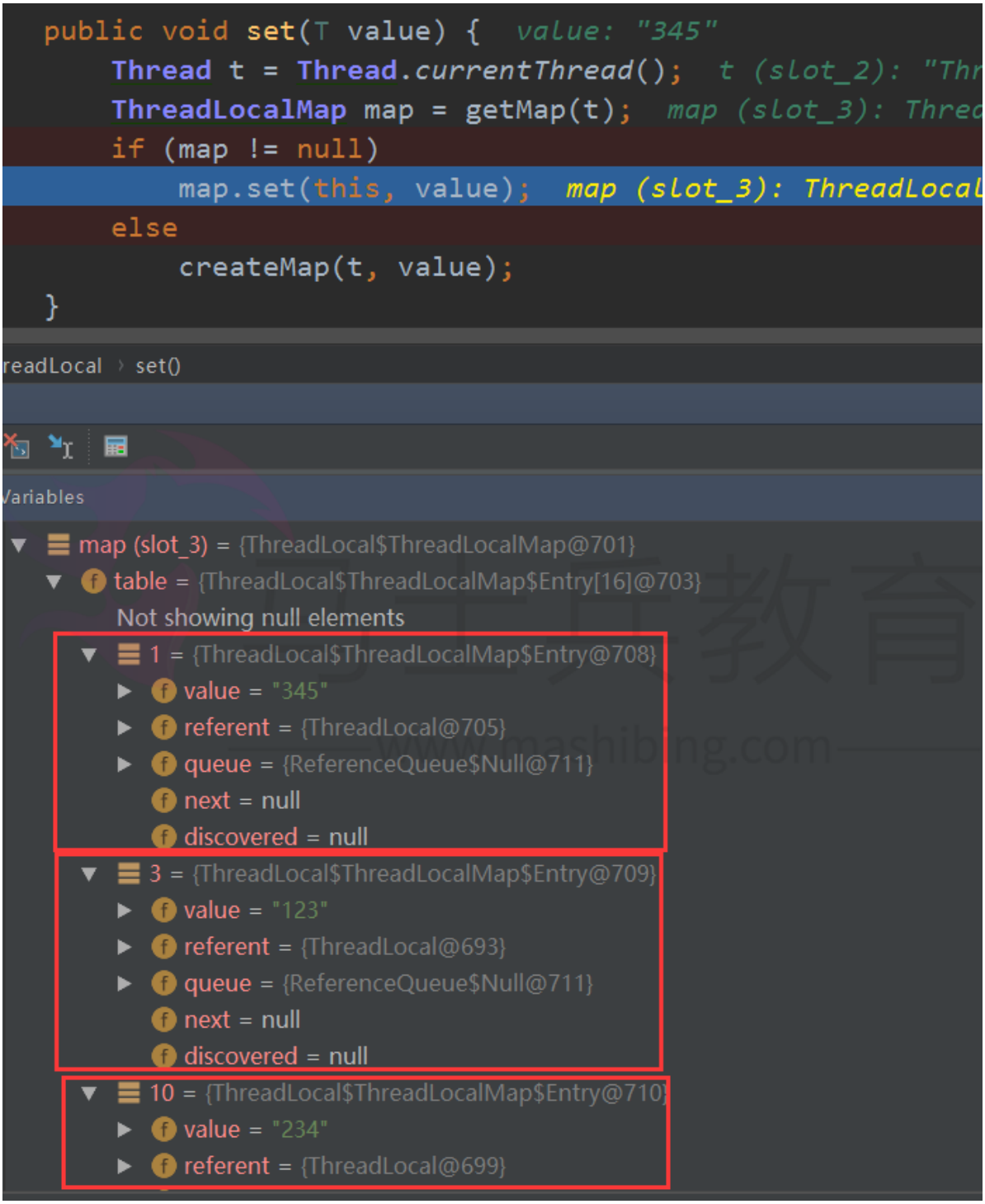
2、了解了set方法的大致原理之后，我们在研究一段程序如下：

```
/**
  * 三个ThreadLocal
  */
private static ThreadLocal<String> threadLocal1 = new ThreadLocal<>();
private static ThreadLocal<String> threadLocal2 = new ThreadLocal<>();
private static ThreadLocal<String> threadLocal3 = new ThreadLocal<>();

//线程池变量指定一个线程
ExecutorService executorService = Executors.newFixedThreadPool(1);

executorService.execute(() -> {
    threadLocal1.set("123");
    threadLocal2.set("234");
    threadLocal3.set("345");
    Thread t = Thread.currentThread();
    System.out.println(Thread.currentThread().getName());
});
123456789101112131415161718
```

这样的话就相当于**一个线程依附了三个ThreadLocal对象**,执行完最后一个set方法之后，调试过程如下：



可以看到table（Entry集合）中有三个对象，对象的值就是我们设置的三个threadLocal的对象值；

3、如果在修改一下代码，修改为两个线程：

```
private static final int THREAD_LOOP_SIZE = 2;

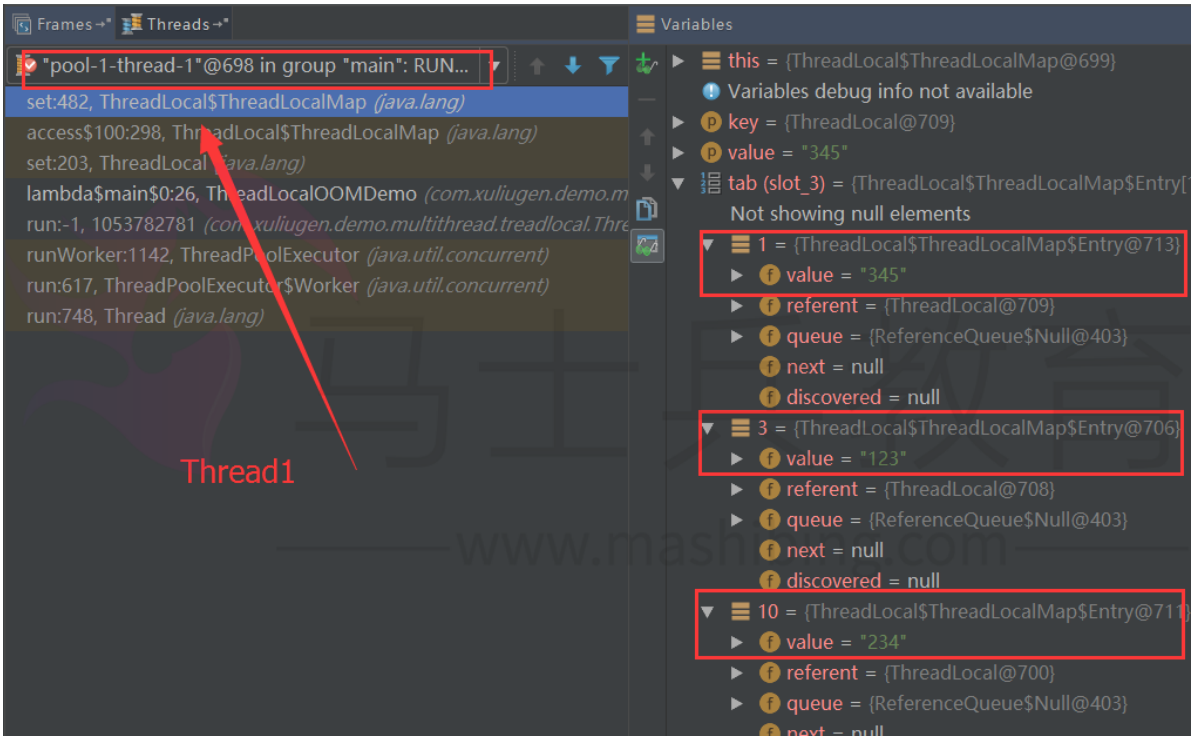
private static ThreadLocal<String> threadLocal1 = new ThreadLocal<>();
private static ThreadLocal<String> threadLocal2 = new ThreadLocal<>();
private static ThreadLocal<String> threadLocal3 = new ThreadLocal<>();

ExecutorService executorService =
Executors.newFixedThreadPool(THREAD_LOOP_SIZE);

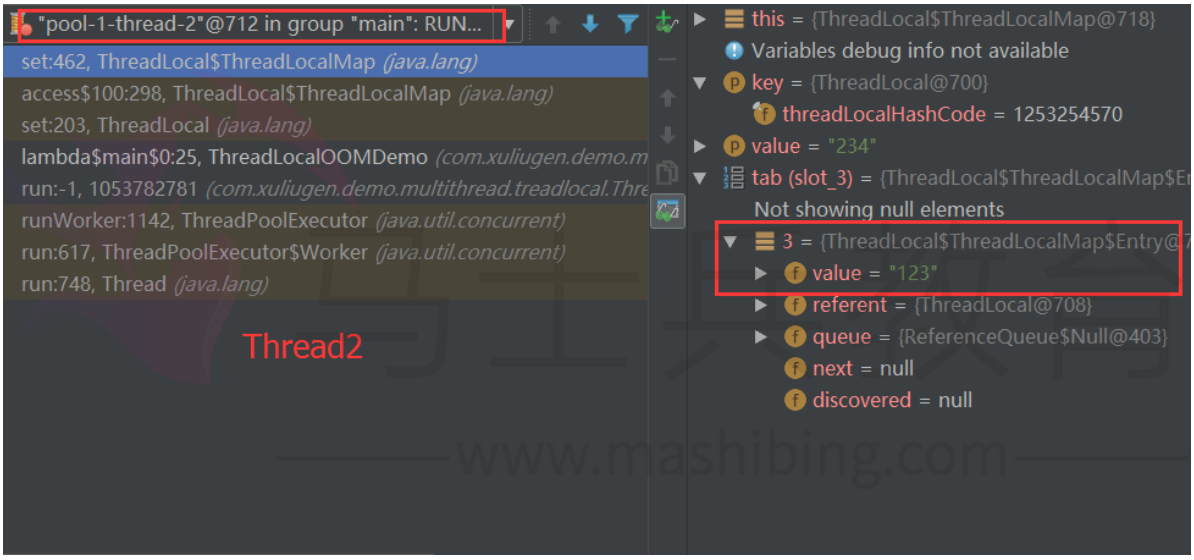
for (int i = 0; i < THREAD_LOOP_SIZE; i++) {
    executorService.execute(() -> {
        threadLocal1.set("123");
        threadLocal2.set("234");
```

```
            threadLocal3.set("345");
        });
    }
```

这样的话，可以看到运行调试图如下：



然后更改到Thread2，查看，由于多线程，线程1运行到上图情况，线程2运行到下图情况，也可以看出他们是不同的ThreadLocalMap：



那如果多个线程，只设置一个ThreadLocal变量那，结果可想而知，这里不再赘述！

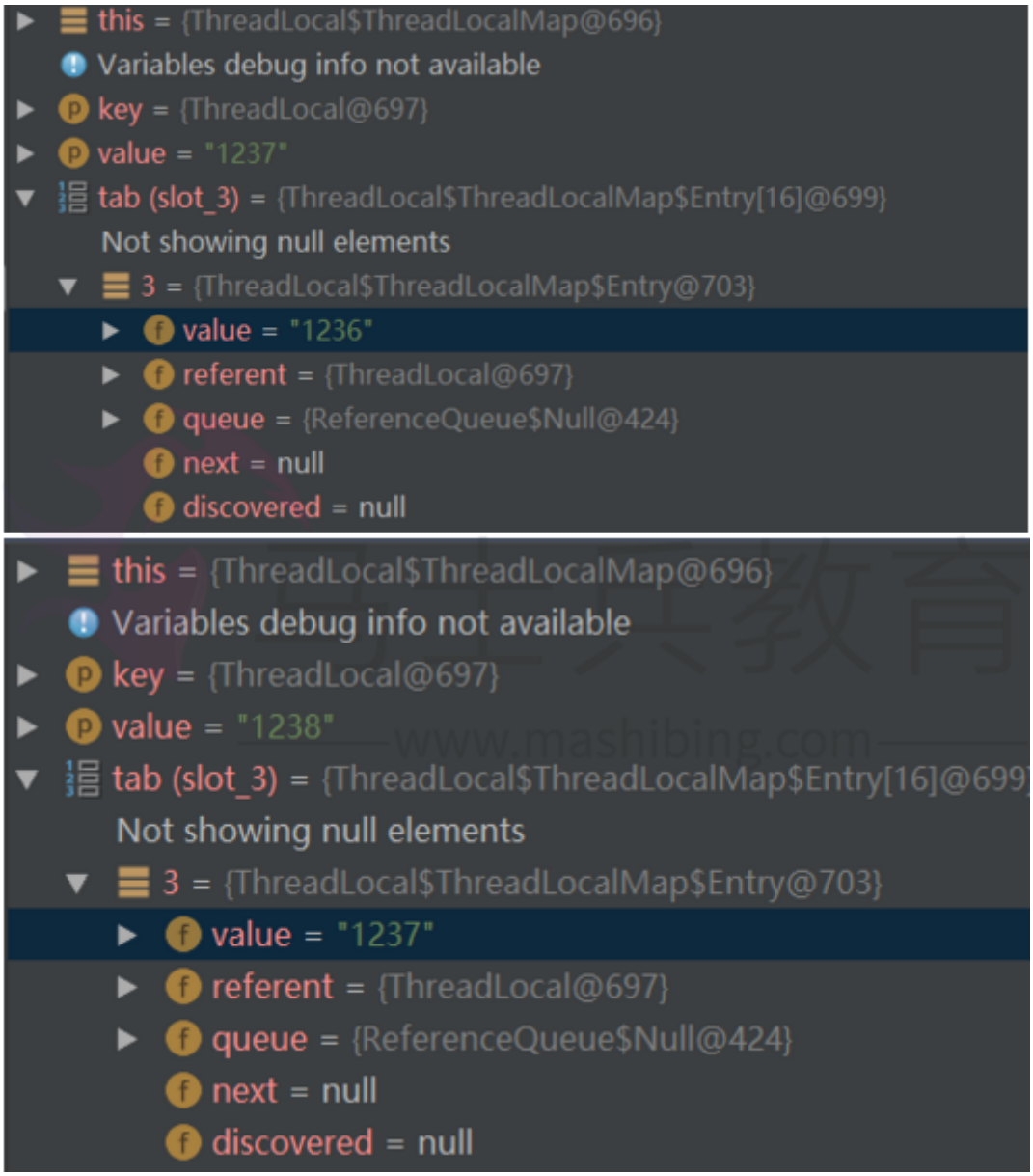
另外，有一点需要提示一下，代码如下：

```
private static final int THREAD_LOOP_SIZE = 1;
private static final int MOCK_DIB_DATA_LOOP_SIZE = 1000;
private static ThreadLocal<String> threadLocal = new ThreadLocal<>();

ExecutorService executorService =
Executors.newFixedThreadPool(THREAD_LOOP_SIZE);

for (int i = 0; i < THREAD_LOOP_SIZE; i++) {
    for (int j = 0; j < MOCK_DIB_DATA_LOOP_SIZE; j++) {
        executorService.execute(() -> threadLocal.set(("123" +
index).toString()));
    }
}1234567891011
```

运行结果：



可以看到，在这个线程中的ThreadLocal变量的值始终是只有一个的，即以前的值被覆盖了的！这里是因为Entry对象是以该ThreadLocal变量的引用为key的，所以多次赋值以前的值会被覆盖，特此注意！

到这里应该可以清楚了的了解Thread、ThreadLocal和ThreadLocalMap之间的关系了！

ThreadLocal的get方法

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

```
private Entry getEntry(ThreadLocal<?> key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
    if (e != null && e.get() == key)
        return e;
    else
        return getEntryAfterMiss(key, i, e);
}
```

经过上述set方法的分析，对于get方法应该理解起来轻松了许多，首先获取ThreadLocalMap对象，由于ThreadLocalMap使用的当前的ThreadLocal作为key，所以传入的参数为this，然后调用`getEntry()`方法，通过这个key构造索引，根据索引去table（Entry数组）中去查找线程本地变量，根据下边找到Entry对象，然后判断Entry对象e不为空并且e的引用与传入的key一样则直接返回，如果找不到则调用`getEntryAfterMiss()`方法。调用`getEntryAfterMiss`表示直接散列到的位置没找到，那么顺着hash表递增（循环）地往下找，从i开始，一直往下找，直到出现空的槽为止。

```
private Entry getEntryAfterMiss(ThreadLocal<?> key, int i, Entry e) {
    Entry[] tab = table;
    int len = tab.length;

    while (e != null) {
        ThreadLocal<?> k = e.get();
        if (k == key)
            return e;
        if (k == null)
            expungeStaleEntry(i);
        else
            i = nextIndex(i, len);
        e = tab[i];
    }
    return null;
}
```

ThreadLocal的内存回收

ThreadLocal 涉及到的两个层面的内存自动回收：

1）在 ThreadLocal 层面的内存回收：

```
 * <p>Each thread holds an implicit reference to its copy of a thread-local
 * variable as long as the thread is alive and the {@code ThreadLocal}
 * instance is accessible; after a thread goes away, all of its copies of
 * thread-local instances are subject to garbage collection (unless other
 * references to these copies exist).
 *
 * @author  Josh Bloch and Doug Lea
 * @since   1.2
 */
public class ThreadLocal<T> {
```

当线程死亡时，那么所有的保存在的线程局部变量就会被回收，其实这里是指线程Thread对象中的`ThreadLocal.ThreadLocalMap threadLocals`会被回收，这是显然的。

2）ThreadLocalMap 层面的内存回收：

```
/**
 * ThreadLocalMap is a customized hash map suitable only for
 * maintaining thread local values. No operations are exported
 * outside of the ThreadLocal class. The class is package private to
 * allow declaration of fields in class Thread.  To help deal with
 * very large and long-lived usages, the hash table entries use
 * WeakReferences for keys. However, since reference queues are not
 * used, stale entries are guaranteed to be removed only when
 * the table starts running out of space.
 */
static class ThreadLocalMap {
```

如果线程可以活很长的时间，并且该线程保存的线程局部变量有很多(也就是 Entry 对象很多)，那么就涉及到在线程的生命期内如何回收 ThreadLocalMap 的内存了，不然的话，Entry对象越多，那么 ThreadLocalMap 就会越来越大，占用的内存就会越来越多，所以对于已经不需要了的线程局部变量，就应该清理掉其对应的Entry对象。

使用的方式是，Entry对象的key是WeakReference 的包装，当ThreadLocalMap 的`private Entry[] table`，已经被占用达到了三分之二时`threshold = 2/3`(也就是线程拥有的局部变量超过了10个)，就会尝试回收 Entry 对象，我们可以看到`ThreadLocalMap.set()`方法中有下面的代码：

```
if (!cleanSomeSlots(i, sz) && sz >= threshold)
       rehash();12
```

cleanSomeSlots 就是进行回收内存：

```
private boolean cleanSomeSlots(int i, int n) {
    boolean removed = false;
    Entry[] tab = table;
    int len = tab.length;
    do {
        i = nextIndex(i, len);
        Entry e = tab[i];
        if (e != null && e.get() == null) {
            n = len;
            removed = true;
            i = expungeStaleEntry(i);
        }
    } while ( (n >>>= 1) != 0);
    return removed;
}
```

ThreadLocal可能引起的OOM内存溢出问题简要分析

我们知道ThreadLocal变量是维护在Thread内部的，这样的话只要我们的线程不退出，对象的引用就会一直存在。当线程退出时，Thread类会进行一些清理工作，其中就包含ThreadLocalMap，Thread调用exit方法如下：

```
/**
 * This method is called by the system to give a Thread
 * a chance to clean up before it actually exits.
 */
private void exit() {
    if (group != null) {
        group.threadTerminated(this);
        group = null;
    }
    /* Aggressively null out all reference fields: see bug 4006245 */
    target = null;
    /* Speed the release of some of these resources */
    threadLocals = null;
    inheritableThreadLocals = null;
    inheritedAccessControlContext = null;
    blocker = null;
    uncaughtExceptionHandler = null;
}
```

但是，当我们使用线程池的时候，就意味着当前线程未必会退出（比如固定大小的线程池，线程总是存在的）。如果这样的话，将一些很大的对象设置到ThreadLocal中（这个很大的对象实际保存在Thread的threadLocals属性中），这样的话就可能会出现内存溢出的情况。

一种场景就是说如果使用了线程池并且设置了固定的线程，处理一次业务的时候存放到ThreadLocalMap中一个大对象，处理另一个业务的时候，又一个线程存放到ThreadLocalMap中一个大对象，但是这个线程由于是线程池创建的他会一直存在，不会被销毁，这样的话，以前执行业务的时候存放到ThreadLocalMap中的对象可能不会被再次使用，但是由于线程不会被关闭，因此无法释放

Thread 中的ThreadLocalMap对象，造成内存溢出。

也就是说，ThreadLocal在没有线程池使用的情况下，正常情况下不会存在内存泄露，但是如果使用了线程池的话，就依赖于线程池的实现，如果线程池不销毁线程的话，那么就会存在内存泄露。所以我们在使用线程池的时候，使用ThreadLocal要格外小心！

总结

通过源代码可以看到每个线程都可以独立修改属于自己的副本而不会互相影响，从而隔离了线程和线程. 避免了线程访问实例变量发生安全问题. 同时我们也能得出下面的结论：

（1）ThreadLocal只是操作Thread中的ThreadLocalMap对象的集合；

（2）ThreadLocalMap变量属于线程的内部属性，不同的线程拥有完全不同的ThreadLocalMap变量；

（3）线程中的ThreadLocalMap变量的值是在ThreadLocal对象进行set或者get操作时创建的；

（4）使用当前线程的ThreadLocalMap的关键在于使用当前的ThreadLocal的实例作为key来存储value值；

（5） ThreadLocal模式至少从两个方面完成了数据访问隔离，即纵向隔离(线程与线程之间的ThreadLocalMap不同)和横向隔离(不同的ThreadLocal实例之间的互相隔离)；

（6）一个线程中的所有的局部变量其实存储在该线程自己的同一个map属性中；

（7）线程死亡时，线程局部变量会自动回收内存；

（8）线程局部变量时通过一个 Entry 保存在map中，该Entry 的key是一个 WeakReference包装的ThreadLocal, value为线程局部变量，key 到 value 的映射是通过：`ThreadLocal.threadLocalHashCode & (INITIAL_CAPACITY - 1)` 来完成的；

（9）当线程拥有的局部变量超过了容量的2/3(没有扩大容量时是10个)，会涉及到ThreadLocalMap中Entry的回收；

对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

synchronized、volatile区别

1) volatile主要应用在多个线程对实例变量更改的场合，刷新主内存共享变量的值从而使得各个 线程可以获得最新的值，线程读取变量的值需要从主存中读取；synchronized则是锁定当前变 量，只有当前线程可以访问该变量，其他线程被阻塞住。另外，synchronized还会创建一个内 存屏障，内存屏障指令保证了所有CPU操作结果都会直接刷到主存中（即释放锁前），从而保证 了操作的内存可见性，同时也使得先获得这个锁的线程的所有操作

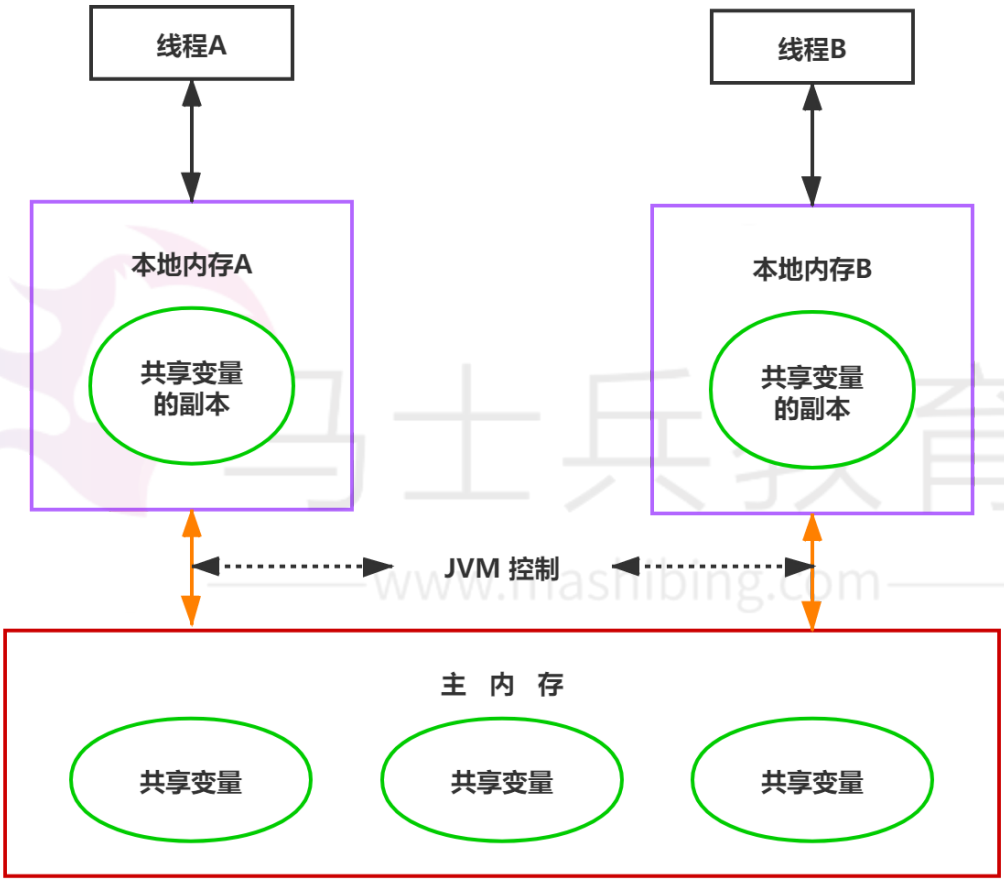
2) volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的。 volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞，比如多个线程争抢 synchronized锁对象时，会出现阻塞。

3) volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的 修改可见性和原子性，因为线程获得锁才能进入临界区，从而保证临界区中的所有语句全部得到 执行。

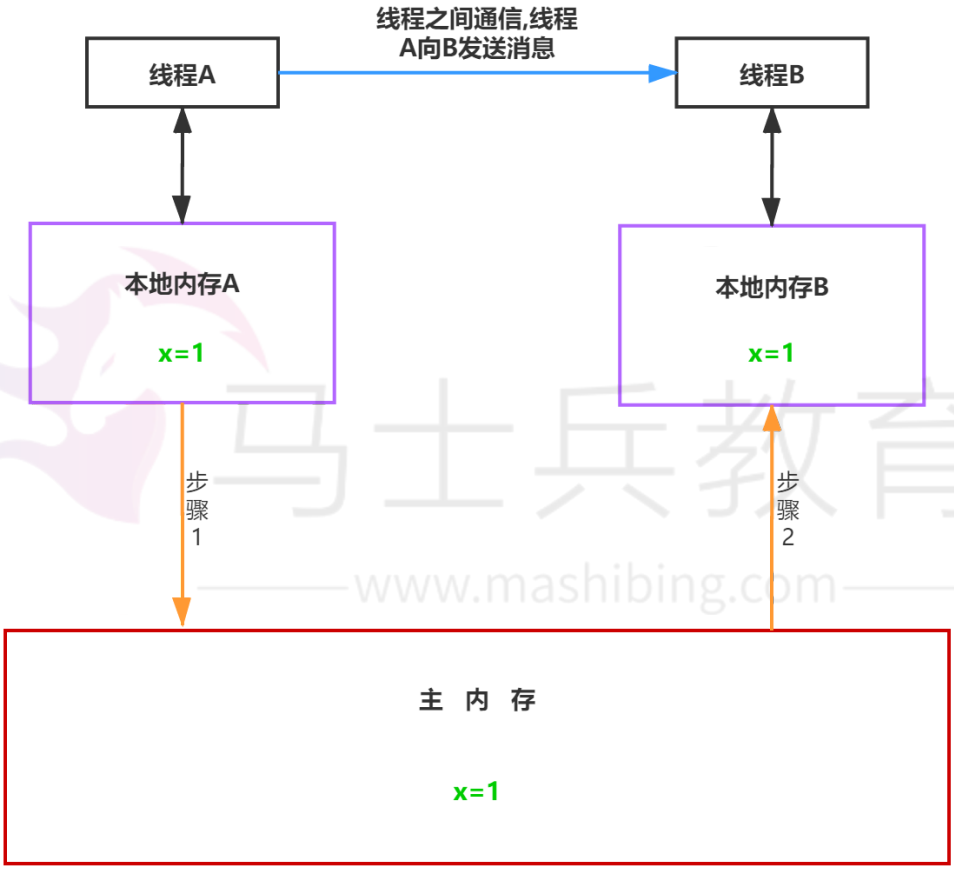
4) volatile标记的变量不会被编译器优化，可以禁止进行指令重排；synchronized标记的变量 可以被编译器优化。

synchronized和volatile区别

个人理解**JMM**：Java Memory Model(Java内存模型)，根据并发过程中如何处理、可见性、原子性和有序性这三个特性而建立的模型。
可见性：JMM提供了volatile变量定义、final、synchronized块来保证可见性。
原子性：个人理解是如果执行，就执行完，synchronized块来保证。
有序性：觉得有序是相对性的，根据从哪个线程观察，volatile和synchronized保证线程之间操作的有序性。
指令重排：处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证各个语句的执行顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。
JMM处理过程：JMM是通过禁止特定类型的编译器重排序和处理器重排序来为程序员提供一致的内存可见性保证。例如A线程具体什么时候刷新共享数据到主内存是不确定的，假设我们使用了同步原语(synchronized，volatile和final),那么刷新的时间是确定的。

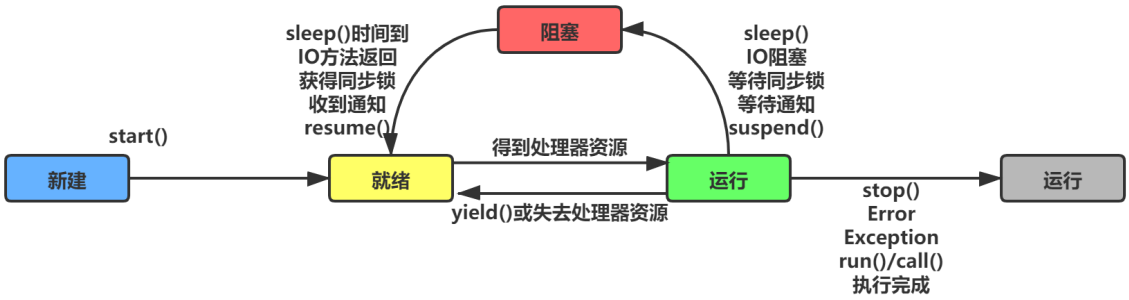


- 每个线程都有一个自己的本地内存空间–虚拟机栈线程空间。线程执行时，先把变量从主内存读取到线程自己的本地内存空间，然后再对该变量进行操作
- 对该变量操作完后，在某个时间再把变量刷新回主内存，所以线程A释放锁后会同步到主内存，线程B获取锁后会同步主内存数据，即“A线程释放锁–B线程获取锁”可以实现A，B线程之间的通信



稍微解释下：
假设本地内存A和B有主内存中共享变量x的副本，初始时这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。然后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

建议了解下线程的生命状态，这里就不多做解释，面试的时候很有可能会被问到。



重点：

1. volatile主要应用在多个线程对实例变量更改的场合，刷新主内存共享变量的值从而使得各个线程可以获得最新的值，线程读取变量的值需要从主存中读取；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。另外，synchronized还会创建一个内存屏障，内存屏障指令保证了所有CPU操作结果都会直接刷到主存中（即释放锁前），从而保证了操作的内存可见性，同时也使得先获得这个锁的线程的所有操作
2. volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的。
3. volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞，比如多个线程争抢synchronized锁对象时，会出现阻塞。
4. volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性，因为线程获得锁才能进入临界区，从而保证临界区中的所有语句全部得到执行。

5. volatile标记的变量不会被编译器优化，可以禁止进行指令重排；synchronized标记的变量可以被编译器优化。

synchronized锁粒度、模拟死锁场景；

synchronized：具有原子性，有序性和可见性

粒度：对象锁、类锁

三大性质总结

1. 三大性质简介

在并发编程中分析线程安全的问题时往往需要切入点，那就是**两大核心**：JMM抽象内存模型以及happens-before规则，三条性质：**原子性，有序性和可见性。**关于synchronized和volatile已经讨论过了，就想着将并发编程中这两大神器在 **原子性，有序性和可见性**上做一个比较，当然这也是面试中的高频考点，值得注意。

2. 原子性

原子性是指**一个操作是不可中断的，要么全部执行成功要么全部执行失败，有着"同生共死"的感觉。**及时在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程所干扰。我们先来看看哪些是原子操作，哪些不是原子操作，有一个直观的印象：

```
int a = 10;  //1
a++;  //2
int b=a; //3
a = a+1; //4
```

上面这四个语句中只**有第1个语句是原子操作**，将10赋值给线程工作内存的变量a,而语句2（a++），实际上包含了三个操作：1. 读取变量a的值；2：对a进行加一的操作；3.将计算后的值再赋值给变量a，而这三个操作无法构成原子操作。对语句3,4的分析同理可得这两条语句不具备原子性。当然，java内存模型中定义了8中操作都是原子的，不可再分的。

1. lock(锁定)：作用于主内存中的变量，它把一个变量标识为一个线程独占的状态；
2. unlock(解锁):作用于主内存中的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
3. read（读取）：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便后面的load动作使用；
4. load（载入）：作用于工作内存中的变量，它把read操作从主内存中得到的变量值放入工作内存中的变量副本
5. use（使用）：作用于工作内存中的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作；
6. assign（赋值）：作用于工作内存中的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作；
7. store（存储）：作用于工作内存的变量，它把工作内存中一个变量的值传送给主内存中以便随后的write操作使用；
8. write（操作）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

上面的这些指令操作是相当底层的，可以作为扩展知识面掌握下。那么如何理解这些指令了?比如，把一个变量从主内存中复制到工作内存中就需要执行read,load操作，将工作内存同步到主内存中就需要执行store,write操作。注意的是：**java内存模型只是要求上述两个操作是顺序执行的并不是连续执行的。**也就是说read和load之间可以插入其他指令，store和writer可以插入其他指令。比如对主内存中的a,b进行访问就可以出现这样的操作顺序：**read a,read b, load b,load a。**

由原子性变量操作read,load,use,assign,store,write，可以**大致认为基本数据类型的访问读写具备原子性**（例外就是long和double的非原子性协定）

> synchronized

上面一共有八条原子操作，其中六条可以满足基本数据类型的访问读写具备原子性，还剩下lock和unlock两条原子操作。如果我们需要更大范围的原子性操作就可以使用lock和unlock原子操作。尽管jvm没有把lock和unlock开放给我们使用，但jvm以更高层次的指令monitorenter和monitorexit指令开放给我们使用，反应到java代码中就是---synchronized关键字，也就是说**synchronized满足原子性。**

> volatile
> 我们先来看这样一个例子：

```
public class VolatileExample {
    private static volatile int counter = 0;

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 10000; i++)
                        counter++;
                }
            });
            thread.start();
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(counter);
    }
}
```

开启10个线程，每个线程都自加10000次，如果不出现线程安全的问题最终的结果应该就是：10*10000 = 100000;可是运行多次都是小于100000的结果，问题在于 **volatile并不能保证原子性**，在前面说过counter++这并不是一个原子操作，包含了三个步骤：1.读取变量counter的值；2.对counter加一；3.将新值赋值给变量counter。如果线程A读取counter到工作内存后，其他线程对这个值已经做了自增操作后，那么线程A的这个值自然而然就是一个过期的值，因此，总结果必然会是小于100000的。

如果让volatile保证原子性，必须符合以下两条规则：

1. **运算结果并不依赖于变量的当前值，或者能够确保只有一个线程修改变量的值；**
2. **变量不需要与其他的状态变量共同参与不变约束**

3. 有序性

> synchronized

synchronized语义表示锁在同一时刻只能由一个线程进行获取，当锁被占用后，其他线程只能等待。因此，synchronized语义就要求线程在访问读写共享变量时只能"串行"执行，因此**synchronized具有有序性。**

> volatile

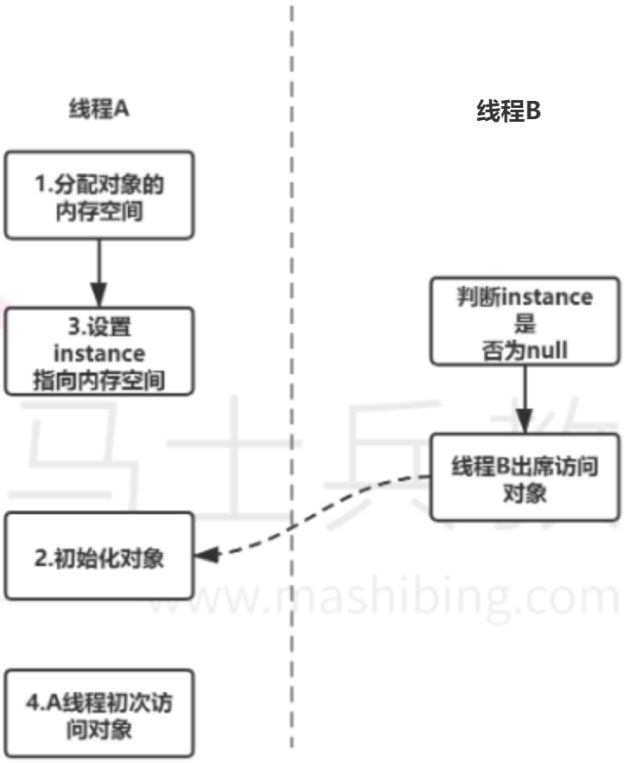
在java内存模型中说过，为了性能优化，编译器和处理器会进行指令重排序；也就是说java程序天然的有序性可以总结为：**如果在本线程内观察，所有的操作都是有序的；如果在一个线程观察另一个线程，所有的操作都是无序的。**在单例模式的实现上有一种双重检验锁定的方式（Double-checked Locking）。代码如下：

```
public class Singleton {
    private Singleton() { }
    private volatile static Singleton instance;
    public Singleton getInstance(){
        if(instance==null){
            synchronized (Singleton.class){
                if(instance==null){
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

这里为什么要加volatile了？我们先来分析一下不加volatile的情况，有问题的语句是这条：

> instance = new Singleton();

这条语句实际上包含了三个操作：1.分配对象的内存空间；2.初始化对象；3.设置instance指向刚分配的内存地址。但由于存在重排序的问题，可能有以下的执行顺序：



如果2和3进行了重排序的话，线程B进行判断if(instance==null)时就会为true，而实际上这个instance并没有初始化成功，显而易见对线程B来说之后的操作就会是错得。而**用volatile修饰**的话就可以禁止2和3操作重排序，从而避免这种情况。**volatile包含禁止指令重排序的语义，其具有有序性。**

4. 可见性

可见性是指当一个线程修改了共享变量后，其他线程能够立即得知这个修改。通过之前对synchronzed内存语义进行了分析，当线程获取锁时会从主内存中获取共享变量的最新值，释放锁的时候会将共享变量同步到主内存中。从而，**synchronized具有可见性。**同样的在volatile分析中，会通过在指令中添加**lock指令**，以实现内存可见性。因此, **volatile具有可见性**

5. 总结

通过这篇文章，主要是比较了synchronized和volatile在三条性质：原子性，可见性，以及有序性的情况，归纳如下：

> **synchronized: 具有原子性，有序性和可见性**;
> **volatile：具有有序性和可见性**

基于synchronized的对象锁，类锁以及死锁模拟

分为对象锁和类锁

```
public class T {
    public void test1() throws Exception{
     synchronized(this){ //对象锁
         System.out.println(Thread.currentThread().getName()+"---test1  Doing");
         Thread.currentThread().sleep(2000);
```

```
            test2();
        };
    }

    public synchronized void test2() throws Exception{ //同test1方法
        System.out.println(Thread.currentThread().getName()+"--- test2 Doing");
        Thread.currentThread().sleep(2000);
        test1();
    }

    public void testclass1() throws Exception{
         synchronized(T.class){//类锁
             System.out.println(Thread.currentThread().getName()+"---testclass1
Doing");
             Thread.currentThread().sleep(2000);
         };
    }

    public static synchronized void testclass2() throws Exception{ //同testclass1
方法
         System.out.println(Thread.currentThread().getName()+"---testclass2
Doing");
         Thread.currentThread().sleep(2000);
         testclass3();
    }

    public static synchronized void testclass3() throws Exception{ //同testclass1
方法
      System.out.println(Thread.currentThread().getName()+"---testclass3
Doing");
      Thread.currentThread().sleep(2000);
      testclass2();
    }
}
```

不同对象 访问类锁测试:

```
public class clent {

    public static void main(String[] args) throws Exception {
        final CountDownLatch c=new CountDownLatch(1);
        new Thread(new Runnable(){
            @Override
            public void run() {
                T t1=new T();
                try {
                    System.out.println(Thread.currentThread().getName()+"启动");
                    c.await();
                    t1.testclass3();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
       new Thread(new Runnable(){
```

```
                    @Override
                    public void run() {
                        T t1=new T();
                        try {

System.out.println(Thread.currentThread().getName()+"启动");
                            c.await();
                            t1.testclass2();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
        c.countDown();
        /* 可以发现 类的synchronized一个线程获得不释放，其他线程就不能访问 通过Jconsole可以
看到Thread0 一直在等待Thread1
        Thread-1启动
        Thread-1---testclass2  Doing
        Thread-0启动
        Thread-1---testclass3  Doing
        Thread-1---testclass2  Doing
        Thread-1---testclass2  Doing
        Thread-1---testclass3  Doing*/
    }
```

不同对象访问对象锁测试：

```
public class clent1 {
    public static void main(String[] args) {
        final CountDownLatch c=new CountDownLatch(1);
        new Thread(new Runnable(){
            @Override
            public void run() {
                T t1=new T();
                try {
                    System.out.println(Thread.currentThread().getName()+"启动");
                    c.await();
                    t1.test1();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable(){
                    @Override
                    public void run() {
                        T t1=new T();
                        try {

System.out.println(Thread.currentThread().getName()+"启动");
                            c.await();
                            t1.test2();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
```

```
                    }
                }).start();
        c.countDown();
       /* Thread-0启动
        Thread-0---test1  Doing
        Thread-1启动
        Thread-1--- test2 Doing
       */
    }
```

同一个对象，多个线程访问对象锁测试：

```
public class client2 {
    public static void main(String[] args) {
        final CountDownLatch c=new CountDownLatch(1);
        final T t1=new T();
        new Thread(new Runnable(){
            @Override
            public void run() {

                try {
                    System.out.println(Thread.currentThread().getName()+"启动");
                    c.await();
                    t1.test1();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
       new Thread(new Runnable(){
                    @Override
                    public void run() {


                        try {

System.out.println(Thread.currentThread().getName()+"启动");
                            c.await();
                            t1.test2();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
        c.countDown();
    }

    /*结果 同一个对象，当其中一个线程没有释放锁时，另一个会一直等待。
    Thread-1启动
    Thread-1--- test2 Doing
    Thread-0启动
    Thread-1---test1  Doing
    Thread-1--- test2 Doing
    Thread-1---test1  Doing
    Thread-1--- test2 Doing*/
```

```
}
```

以上演示的是阻塞等待，下面演示死锁。

```
public class E {
    public static synchronized void testclass() throws Exception{
     System.out.println(Thread.currentThread().getName()+"---E.method  Doing");
     Thread.currentThread().sleep(2000);
     E1.testclass();
    }
}
```

```
public class E1 {
    public static synchronized void testclass() throws Exception{
         System.out.println(Thread.currentThread().getName()+"---E1.method
Doing");
        Thread.currentThread().sleep(1000);
         E.testclass();
        }
}
public class SiSuo {


    public static void main(String[] args) {
        final CountDownLatch c=new CountDownLatch(1);
        new Thread(new Runnable(){
            @Override
            public void run() {
                try {
                    c.await();
                    E.testclass();
                    E1.testclass();
                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }

        }).start();

        new Thread(new Runnable(){
            @Override
            public void run() {
                try {
                    c.await();
                    E1.testclass();
                    E.testclass();
                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
```

```
        }).start();
        c.countDown();
    }
    /*执行结果，一直等待
    Thread-1---E1.method  Doing
    Thread-0---E.method  Doing*/

}
```

Java并发和并行

- 并发 : 是指两个或多个事件在同一时间间隔发生,在一台处理器上“同时”处理多个任务;
- 并行 : 是指两个或者多个事件在同一时刻发生,在多台处理器上同时处理多个任务。

怎么提高并发量，请列举你所知道的方案?

高并发解决方案——提升高并发量服务器性能解决思路

一个小型的网站，可以使用最简单的html静态页面就实现了，配合一些图片达到美化效果，所有的页面均存放在一个目录下，这样的网站对系统架构、性能的要求都很简单。随着互联网业务的不断丰富，网站相关的技术经过这些年的发展，已经细分到很细的方方面面，尤其对于大型网站来说，所采用的技术更是涉及面非常广，从硬件到软件、编程语言、数据库、WebServer、防火墙等各个领域都有了很高的要求，已经不是原来简单的html静态网站所能比拟的。

大型网站，比如门户网站，在面对大量用户访问、高并发请求方面，基本的解决方案集中在这样几个环节：使用高性能的服务器、高性能的数据库、高效率的编程语言、还有高性能的Web容器。这几个解决思路在一定程度上意味着更大的投入。

1、HTML静态化

其实大家都知道，效率最高、消耗最小的就是纯静态化的html页面，所以我们尽可能使我们的网站上的页面采用静态页面来实现，这个最简单的方法其实也是最有效的方法。但是对于大量内容并且频繁更新的网站，我们无法全部手动去挨个实现，于是出现了我们常见的信息发布系统CMS，像我们常访问的各个门户站点的新闻频道，甚至他们的其他频道，都是通过信息发布系统来管理和实现的，信息发布系统可以实现最简单的信息录入自动生成静态页面，还能具备频道管理、权限管理、自动抓取等功能，对于一个大型网站来说，拥有一套高效、可管理的CMS是必不可少的。

除了门户和信息发布类型的网站，对于交互性要求很高的社区类型网站来说，尽可能的静态化也是提高性能的必要手段，将社区内的帖子、文章进行实时的静态化、有更新的时候再重新静态化也是大量使用的策略，像Mop的大杂烩就是使用了这样的策略，网易社区等也是如此。

同时，html静态化也是某些缓存策略使用的手段，对于系统中频繁使用数据库查询但是内容更新很小的应用，可以考虑使用html静态化来实现。比如论坛中论坛的公用设置信息，这些信息目前的主流论坛都可以进行后台管理并且存储在数据库中，这些信息其实大量被前台程序调用，但是更新频率很小，可以考虑将这部分内容进行后台更新的时候进行静态化，这样避免了大量的数据库访问请求。

2、图片服务器分离

大家知道，对于Web服务器来说，不管是Apache、IIS还是其他容器，图片是最消耗资源的，于是我们有必要将图片与页面进行分离，这是基本上大型网站都会采用的策略，他们都有独立的、甚至很多台的图片服务器。这样的架构可以降低提供页面访问请求的服务器系统压力，并且可以保证系统不会因为图片问题而崩溃。

在应用服务器和图片服务器上，可以进行不同的配置优化，比如apache在配置ContentType的时候可以尽量少支持、尽可能少的LoadModule，保证更高的系统消耗和执行效率。

3、数据库集群、库表散列

大型网站都有复杂的应用，这些应用必须使用数据库，那么在面对大量访问的时候，数据库的瓶颈很快就能显现出来，这时一台数据库将很快无法满足应用，于是我们需要使用数据库集群或者库表散列。

在数据库集群方面，很多数据库都有自己的解决方案，Oracle、Sybase等都有很好的方案，常用的MySQL提供的Master/Slave也是类似的方案，您使用了什么样的DB，就参考相应的解决方案来实施即可。

上面提到的数据库集群由于在架构、成本、扩张性方面都会受到所采用DB类型的限制，于是我们需要从应用程序的角度来考虑改善系统架构，库表散列是常用并且最有效的解决方案。

我们在应用程序中安装业务和应用或者功能模块将数据库进行分离，不同的模块对应不同的数据库或者表，再按照一定的策略对某个页面或者功能进行更小的数据库散列，比如用户表，按照用户ID进行表散列，这样就能够低成本的提升系统的性能并且有很好的扩展性。

sohu的论坛就是采用了这样的架构，将论坛的用户、设置、帖子等信息进行数据库分离，然后对帖子、用户按照板块和ID进行散列数据库和表，最终可以在配置文件中进行简单的配置便能让系统随时增加一台低成本的数据库进来补充系统性能。

4、缓存

缓存一词搞技术的都接触过，很多地方用到缓存。网站架构和网站开发中的缓存也是非常重要。这里先讲述最基本的两种缓存。高级和分布式的缓存在后面讲述。

架构方面的缓存，对Apache比较熟悉的人都能知道Apache提供了自己的缓存模块，也可以使用外加的Squid模块进行缓存，这两种方式均可以有效的提高Apache的访问响应能力。

网站程序开发方面的缓存，Linux上提供的Memory Cache是常用的缓存接口，可以在web开发中使用，比如用Java开发的时候就可以调用MemoryCache对一些数据进行缓存和通讯共享，一些大型社区使用了这样的架构。另外，在使用web语言开发的时候，各种语言基本都有自己的缓存模块和方法，PHP有Pear的Cache模块，Java就更多了，.net不是很熟悉，相信也肯定有。

5、镜像

镜像是大型网站常采用的提高性能和数据安全性的方式，镜像的技术可以解决不同网络接入商和地域带来的用户访问速度差异，比如ChinaNet和EduNet之间的差异就促使了很多网站在教育网内搭建镜像站点，数据进行定时更新或者实时更新。在镜像的细节技术方面，这里不阐述太深，有很多专业的现成的解决架构和产品可选。也有廉价的通过软件实现的思路，比如Linux上的rsync等工具。

6、负载均衡

负载均衡将是大型网站解决高负荷访问和大量并发请求采用的高端解决办法。

负载均衡技术发展了多年，有很多专业的服务提供商和产品可以选择，我个人接触过一些解决方法，其中有两个架构可以给大家做参考。

（1）、硬件四层交换

第四层交换使用第三层和第四层信息包的报头信息，根据应用区间识别业务流，将整个区间段的业务流分配到合适的应用服务器进行处理。

第四层交换功能就像是虚IP，指向物理服务器。它传输的业务服从的协议多种多样，有HTTP、FTP、NFS、Telnet或其他协议。这些业务在物理服务器基础上，需要复杂的载量平衡算法。在IP世界，业务类型由终端TCP或UDP端口地址来决定，在第四层交换中的应用区间则由源端和终端IP地址、TCP和UDP端口共同决定。

在硬件四层交换产品领域，有一些知名的产品可以选择，比如Alteon、F5等，这些产品很昂贵，但是物有所值，能够提供非常优秀的性能和很灵活的管理能力。“Yahoo中国”当初接近2000台服务器，只使用了三、四台Alteon就搞定了。

(2)、软件四层交换

大家知道了硬件四层交换机的原理后，基于OSI模型来实现的软件四层交换也就应运而生，这样的解决方案实现的原理一致，不过性能稍差。但是满足一定量的压力还是游刃有余的，有人说软件实现方式其实更灵活，处理能力完全看你配置的熟悉能力。

软件四层交换我们可以使用Linux上常用的LVS来解决，LVS就是Linux Virtual Server，他提供了基于心跳线heartbeat的实时灾难应对解决方案，提高系统的强壮性，同时可供了灵活的虚拟VIP配置和管理功能，可以同时满足多种应用需求，这对于分布式的系统来说必不可少。

一个典型的使用负载均衡的策略就是，在软件或者硬件四层交换的基础上搭建squid集群，这种思路在很多大型网站包括搜索引擎上被采用，这样的架构低成本、高性能还有很强的扩张性，随时往架构里面增减节点都非常容易。

对于大型网站来说，前面提到的每个方法可能都会被同时使用到，这里介绍得比较浅显，具体实现过程中很多细节还需要大家慢慢熟悉和体会。有时一个很小的squid参数或者apache参数设置，对于系统性能的影响就会很大。

7、最新：CDN加速技术

什么是CDN?

CDN的全称是内容分发网络。其目的是通过在现有的Internet中增加一层新的网络架构，将网站的内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容，提高用户访问网站的响应速度。

CDN有别于镜像，因为它比镜像更智能，或者可以做这样一个比喻：CDN=更智能的镜像+缓存+流量导流。因而，CDN可以明显提高Internet网络中信息流动的效率。从技术上全面解决由于网络带宽小、用户访问量大、网点分布不均等问题，提高用户访问网站的响应速度。

CDN的类型特点

CDN的实现分为三类：镜像、高速缓存、专线。

镜像站点（Mirror Site），是最常见的，它让内容直接发布，适用于静态和准动态的数据同步。但是购买和维护新服务器的费用较高，还必须在各个地区设置镜像服务器，配备专业技术人员进行管理与维护。对于大型网站来说，更新所用的带宽成本也大大提高了。

高速缓存，成本较低，适用于静态内容。Internet的统计表明，超过80%的用户经常访问的是20%的网站的内容，在这个规律下，缓存服务器可以处理大部分客户的静态请求，而原始的服务器只需处理约20%左右的非缓存请求和动态请求，于是大大加快了客户请求的响应时间，并降低了原始服务器的负载。

CDN服务一般会在全国范围内的关键节点上放置缓存服务器。

专线，让用户直接访问数据源，可以实现数据的动态同步。

CDN的实例

举个例子来说，当某用户访问网站时，网站会利用全球负载均衡技术，将用户的访问指向到距离用户最近的正常工作的缓存服务器上，直接响应用户的请求。

当用户访问已经使用了CDN服务的网站时，其解析过程与传统解析方式的最大区别就在于网站的授权域名服务器不是以传统的轮询方式来响应本地DNS的解析请求，而是充分考虑用户发起请求的地点和当时网络的情况，来决定把用户的请求定向到离用户最近同时负载相对较轻的节点缓存服务器上。

通过用户定位算法和服务器健康检测算法综合后的数据，可以将用户的请求就近定向到分布在网络“边缘”的缓存服务器上，保证用户的访问能得到更及时可靠的响应。

由于大量的用户访问都由分布在网络边缘的CDN节点缓存服务器直接响应了，这就不仅提高了用户的访问质量，同时有效地降低了源服务器的负载压力。

附：某CDN服务商的服务说明

采用GCDN加速方式

采用了GCDN加速方式以后，系统会在浏览用户和您的服务器之间增加一台GCDN服务器。浏览用户访问您的服务器时，一般静态数据，如图片、多媒体资料等数据将直接从GCDN服务器读取，使得从主服务器上读取静态数据的交换量大大减少。

为VIP型虚拟主机而特加的VPN高速压缩通道，使用高速压缩的电信<==>网通、电信<==>国际（HK）、网通<==>国际（HK）等跨网专线通道，智能多线，自动获取最快路径，极速的动态实时并发响应速度，实现了网站的动态脚本实时同步，对动态网站有一个更加明显的加速效果。

每个网络运营商（电信、网通、铁通、教育网）均有您服务器的GCDN服务器，无论浏览用户是来自何处，GCDN都能让您的服务器展现最快的速度！另外，我们将对您的数据进行实时备份，让您的数据更安全！

系统的用户量有多少？多用户并发访问时如何解决？

大型网站是怎样解决多用户高并发访问的

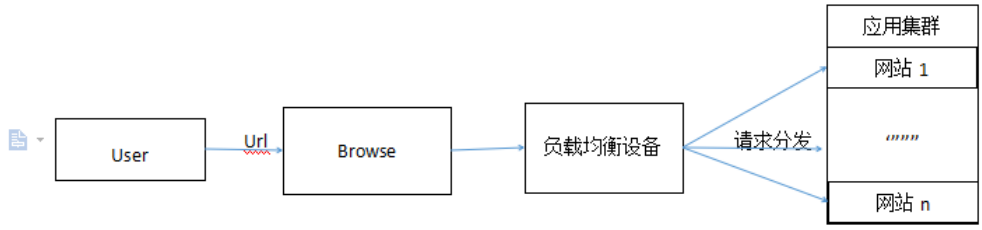
分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率。

集群主要分为：高可用集群(High Availability Cluster)，负载均衡集群(Load Balance Cluster，nginx即可实现)，科学计算集群(High Performance Computing Cluster)。

分布式是指将不同的业务分布在不同的地方；而集群指的是将几台服务器集中在一起，实现同一业务。分布式中的每一个节点，都可以做集群。 而集群并不一定就是分布式的。

为了解决大型网站的访问量大、并发量高、海量数据的问题，我们一般会考虑业务拆分和分布式部署。我们可以把那些关联不太大的业务独立出来，部署到不同的机器上，从而实现大规模的分布式系统。但这之中也有一个问题，那就是用户如何选择相应的机器的问题，这也被称为访问统一入口问题，而解决的方法是我们可以在集群机器的前面增加负载均衡设备，实现流量分发（总图如下）。

这里得先解释一下何为“负载均衡”，负载均衡就是将负载（工作任务、访问请求等）进行平衡、分摊到多个操作单元（服务器、组件等）上进行执行，是解决高性能，单点故障（高可用，如果你是单机版网络，一旦服务器挂掉了，那么用户就无法请求了，但对于集群来说，一台服务器挂掉了，负载均衡器会把用户的请求发送给其他的服务器进行处理），扩展性（这里主要是指水平伸缩）的终极解决方案。



在这里，本人主要讨论负载均衡设备为Nginx(至于为啥不讲讲F5，因为人家太贵了，不过人家比较稳定)，这是一款轻量级的Web服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，具有占用内存少、并发能力强等，中国大陆使用nginx网站用户有：百度、网易、新浪、腾讯等（该介绍来自百科）。

nginx大家可以上其 官网 去下载最新版，解压后复制到部署目录，对于Nginx的配置网上的资料很多，这里就不再赘述了，只总结一下Nginx使用的注意事项：

1.nginx的负载均衡配置中默认是采用轮询的方式，这种方式中，每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除，但存在各个服务器的session共享问题。

2.另外一种方式是ip_hash:每个请求按访问的ip的hash结果分配，如果访问的IP是固定的，那么在正常情况下，该用户的请求都会分配到后台的同一台服务器去处理，但是如果用户每次请求的IP都不同呢？所以这种方式也同1的方式一样都存在这么一个问题：session在各个服务器上的共享问题。

3.，如果集群中的服务器的性能不一，可以通过配置各个服务器的权值来实现资源利用率的最大化，即性能好的优先选择

也许你会问，既然IP可能变化，那么用户用页面请求时的cookie的ID应该是确定的吧！那么我们可以用cookie_id来进行hash，然后在通过负载均衡器分发到对应的服务器上，这样就可以解决session问题了，其实当初本人也有想到这个方案，但最后本人也放弃这个方案了，因为是根据cookid_id确实可以把该用户的请求唯一的分发到那台独一无二的服务器上，那如果这台服务器挂掉了，那么根据这种分发策略，岂不是在这服务器上请求资源的用户都不能访问了，你说是不是呢？

解决服务器共享session问题：使用redis来共享各个服务器的session，并同时通过redis来缓存一些常用的资源，加快用户获得请求资源的速度（个人比较喜欢redis，当然你们也可以使用memcache来实现，不过，memcache不能做到持久化，这样这台服务器一挂掉，那么所有的资源也都没有了……）。

不过，本人觉得这样进行集群部署，最好配上数据库的主从部署，因为如果在集群中只分配一个数据库服务器，那么这个系统的瓶颈将会出现在数据库的操作上，虽然redis能减轻这种负担，但对于数据量大的还是有一定影响的，而且数据库的主从部署也可以防止因某个数据库服务器的挂掉而丢失用户的信息。

说说阻塞队列的实现：可以参考ArrayBlockingQueue的底层实现（锁和同步都行）

Java阻塞队列ArrayBlockingQueue和LinkedBlockingQueue实现原理分析

Java中的阻塞队列接口BlockingQueue继承自Queue接口。

BlockingQueue接口提供了3个添加元素方法。

add：添加元素到队列里，添加成功返回true，由于容量满了添加失败会抛出IllegalStateException异常
offer：添加元素到队列里，添加成功返回true，添加失败返回false
put：添加元素到队列里，如果容量满了会阻塞直到容量不满
3个删除方法。

poll：删除队列头部元素，如果队列为空，返回null。否则返回元素。
remove：基于对象找到对应的元素，并删除。删除成功返回true，否则返回false
take：删除队列头部元素，如果队列为空，一直阻塞到队列有元素并删除
常用的阻塞队列具体类有ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue、LinkedBlockingDeque等。

本文以ArrayBlockingQueue和LinkedBlockingQueue为例，分析它们的实现原理。

ArrayBlockingQueue

ArrayBlockingQueue的原理就是使用一个可重入锁和这个锁生成的两个条件对象进行并发控制(classic two-condition algorithm)。

ArrayBlockingQueue是一个带有长度的阻塞队列，初始化的时候必须要指定队列长度，且指定长度之后不允许进行修改。

它带有的属性如下：

```
// 存储队列元素的数组，是个循环数组
final Object[] items;

// 拿数据的索引，用于take，poll，peek，remove方法
int takeIndex;

// 放数据的索引，用于put，offer，add方法
int putIndex;

// 元素个数
int count;

// 可重入锁
final ReentrantLock lock;
// notEmpty条件对象，由lock创建
private final Condition notEmpty;
// notFull条件对象，由lock创建
private final Condition notFull;
```

数据的添加

ArrayBlockingQueue有不同的几个数据添加方法，add、offer、put方法。

add方法：

```
public boolean add(E e) {
    if (offer(e))
        return true;
    else
        throw new IllegalStateException("Queue full");
}
```

add方法内部调用offer方法如下：

```
public boolean offer(E e) {
    checkNotNull(e); // 不允许元素为空
    final ReentrantLock lock = this.lock;
    lock.lock(); // 加锁，保证调用offer方法的时候只有1个线程
    try {
        if (count == items.length) // 如果队列已满
            return false; // 直接返回false，添加失败
        else {
            insert(e); // 数组没满的话调用insert方法
            return true; // 返回true，添加成功
        }
    } finally {
        lock.unlock(); // 释放锁，让其他线程可以调用offer方法
    }
}
```

insert方法如下：

```
private void insert(E x) {
    items[putIndex] = x; // 元素添加到数组里
    putIndex = inc(putIndex); // 放数据索引+1，当索引满了变成0
    ++count; // 元素个数+1
    notEmpty.signal(); // 使用条件对象notEmpty通知，比如使用take方法的时候队列里没有数据，被阻塞。这个时候队列insert了一条数据，需要调用signal进行通知
}
```

put方法：

```
public void put(E e) throws InterruptedException {
    checkNotNull(e); // 不允许元素为空
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly(); // 加锁，保证调用put方法的时候只有1个线程
    try {
        while (count == items.length) // 如果队列满了，阻塞当前线程，并加入到条件对象notFull的等待队列里
            notFull.await(); // 线程阻塞并被挂起，同时释放锁
        insert(e); // 调用insert方法
    } finally {
        lock.unlock(); // 释放锁，让其他线程可以调用put方法
    }
}
```

ArrayBlockingQueue的添加数据方法有add，put，offer这3个方法，总结如下：

add方法内部调用offer方法，如果队列满了，抛出IllegalStateException异常，否则返回true

offer方法如果队列满了，返回false，否则返回true

add方法和offer方法不会阻塞线程，put方法如果队列满了会阻塞线程，直到有线程消费了队列里的数据才有可能被唤醒。

这3个方法内部都会使用可重入锁保证原子性。

数据的删除

ArrayBlockingQueue有不同的几个数据删除方法，poll、take、remove方法。

poll方法：

```
public E poll() {
    final ReentrantLock lock = this.lock;
    lock.lock(); // 加锁，保证调用poll方法的时候只有1个线程
    try {
        return (count == 0) ? null : extract(); // 如果队列里没元素了，返回null，否则调用extract方法
    } finally {
        lock.unlock(); // 释放锁，让其他线程可以调用poll方法
    }
}
```

poll方法内部调用extract方法：

```
private E extract() {
    final Object[] items = this.items;
    E x = this.<E>cast(items[takeIndex]); // 得到取索引位置上的元素
    items[takeIndex] = null; // 对应取索引上的数据清空
    takeIndex = inc(takeIndex); // 取数据索引+1，当索引满了变成0
    --count; // 元素个数-1
    notFull.signal(); // 使用条件对象notFull通知，比如使用put方法放数据的时候队列已满，被阻塞。这个时候消费了一条数据，队列没满了，就需要调用signal进行通知
    return x; // 返回元素
}
```

take方法：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly(); // 加锁，保证调用take方法的时候只有1个线程
    try {
        while (count == 0) // 如果队列空，阻塞当前线程，并加入到条件对象notEmpty的等待队列里
            notEmpty.await(); // 线程阻塞并被挂起，同时释放锁
        return extract(); // 调用extract方法
    } finally {
        lock.unlock(); // 释放锁，让其他线程可以调用take方法
    }
}
```

remove方法：

```
public boolean remove(Object o) {
    if (o == null) return false;
    final Object[] items = this.items;
```

```
    final ReentrantLock lock = this.lock;
    lock.lock(); // 加锁，保证调用remove方法的时候只有1个线程
    try {
        for (int i = takeIndex, k = count; k > 0; i = inc(i), k--) { // 遍历元素
            if (o.equals(items[i])) { // 两个对象相等的话
                removeAt(i); // 调用removeAt方法
                return true; // 删除成功，返回true
            }
        }
        return false; // 删除成功，返回false
    } finally {
        lock.unlock(); // 释放锁，让其他线程可以调用remove方法
    }
}
```

removeAt方法：

```
void removeAt(int i) {
    final Object[] items = this.items;
    if (i == takeIndex) { // 如果要删除数据的索引是取索引位置，直接删除取索引位置上的数据，然后取索引+1即可
        items[takeIndex] = null;
        takeIndex = inc(takeIndex);
    } else { // 如果要删除数据的索引不是取索引位置，移动元素元素，更新取索引和放索引的值
        for (;;) {
            int nexti = inc(i);
            if (nexti != putIndex) {
                items[i] = items[nexti];
                i = nexti;
            } else {
                items[i] = null;
                putIndex = i;
                break;
            }
        }
    }
    --count; // 元素个数-1
    notFull.signal(); // 使用条件对象notFull通知，比如使用put方法放数据的时候队列已满，被阻塞。这个时候消费了一条数据，队列没满了，就需要调用signal进行通知
}
```

ArrayBlockingQueue的删除数据方法有poll，take，remove这3个方法，总结如下：

poll方法对于队列为空的情况，返回null，否则返回队列头部元素。

remove方法取的元素是基于对象的下标值，删除成功返回true，否则返回false。

poll方法和remove方法不会阻塞线程。

take方法对于队列为空的情况，会阻塞并挂起当前线程，直到有数据加入到队列中。

这3个方法内部都会调用notFull.signal方法通知正在等待队列满情况下的阻塞线程。

LinkedBlockingQueue

LinkedBlockingQueue是一个使用链表完成队列操作的阻塞队列。链表是单向链表，而不是双向链表。

内部使用放锁和拿锁，这两个锁实现阻塞(“two lock queue” algorithm)。

它带有的属性如下：

```
// 容量大小
private final int capacity;

// 元素个数，因为有2个锁，存在竞态条件，使用AtomicInteger
private final AtomicInteger count = new AtomicInteger(0);

// 头结点
private transient Node<E> head;

// 尾节点
private transient Node<E> last;

// 拿锁
private final ReentrantLock takeLock = new ReentrantLock();

// 拿锁的条件对象
private final Condition notEmpty = takeLock.newCondition();

// 放锁
private final ReentrantLock putLock = new ReentrantLock();

// 放锁的条件对象
private final Condition notFull = putLock.newCondition();
```

ArrayBlockingQueue只有1个锁，添加数据和删除数据的时候只能有1个被执行，不允许并行执行。

而LinkedBlockingQueue有2个锁，放锁和拿锁，添加数据和删除数据是可以并行进行的，当然添加数据和删除数据的时候只能有1个线程各自执行。

数据的添加

LinkedBlockingQueue有不同的几个数据添加方法，add、offer、put方法。

add方法内部调用offer方法：

```
public boolean offer(E e) {
    if (e == null) throw new NullPointerException(); // 不允许空元素
    final AtomicInteger count = this.count;
    if (count.get() == capacity) // 如果容量满了，返回false
        return false;
    int c = -1;
    Node<E> node = new Node(e); // 容量没满，以新元素构造节点
    final ReentrantLock putLock = this.putLock;
    putLock.lock(); // 放锁加锁，保证调用offer方法的时候只有1个线程
    try {
        if (count.get() < capacity) { // 再次判断容量是否已满，因为可能拿锁在进行消费数据，没满的话继续执行
            enqueue(node); // 节点添加到链表尾部
            c = count.getAndIncrement(); // 元素个数+1
            if (c + 1 < capacity) // 如果容量还没满
                notFull.signal(); // 在放锁的条件对象notFull上唤醒正在等待的线程，表示可以再次往队列里面加数据了，队列还没满
        }
    } finally {
        putLock.unlock(); // 释放放锁，让其他线程可以调用offer方法
    }
```

```
    if (c == 0) // 由于存在放锁和拿锁，这里可能拿锁一直在消费数据，count会变化。这里的if条件表示如果队列中还有1条数据
        signalNotEmpty(); // 在拿锁的条件对象notEmpty上唤醒正在等待的1个线程，表示队列里还有1条数据，可以进行消费
    return c >= 0; // 添加成功返回true，否则返回false
}
```

put方法：

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException(); // 不允许空元素
    int c = -1;
    Node<E> node = new Node(e); // 以新元素构造节点
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly(); // 放锁加锁，保证调用put方法的时候只有1个线程
    try {
        while (count.get() == capacity) { // 如果容量满了
            notFull.await(); // 阻塞并挂起当前线程
        }
        enqueue(node); // 节点添加到链表尾部
        c = count.getAndIncrement(); // 元素个数+1
        if (c + 1 < capacity) // 如果容量还没满
            notFull.signal(); // 在放锁的条件对象notFull上唤醒正在等待的线程，表示可以再次往队列里面加数据了，队列还没满
    } finally {
        putLock.unlock(); // 释放放锁，让其他线程可以调用put方法
    }
    if (c == 0) // 由于存在放锁和拿锁，这里可能拿锁一直在消费数据，count会变化。这里的if条件表示如果队列中还有1条数据
        signalNotEmpty(); // 在拿锁的条件对象notEmpty上唤醒正在等待的1个线程，表示队列里还有1条数据，可以进行消费
}
```

LinkedBlockingQueue的添加数据方法add，put，offer跟ArrayBlockingQueue一样，不同的是它们的底层实现不一样。

ArrayBlockingQueue中放入数据阻塞的时候，需要消费数据才能唤醒。

而LinkedBlockingQueue中放入数据阻塞的时候，因为它内部有2个锁，可以并行执行放入数据和消费数据，不仅在消费数据的时候进行唤醒插入阻塞的线程，同时在插入的时候如果容量还没满，也会唤醒插入阻塞的线程。

数据的删除

LinkedBlockingQueue有不同的几个数据删除方法，poll、take、remove方法。

poll方法：

```
public E poll() {
    final AtomicInteger count = this.count;
    if (count.get() == 0) // 如果元素个数为0
        return null; // 返回null
    E x = null;
    int c = -1;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock(); // 拿锁加锁，保证调用poll方法的时候只有1个线程
    try {
```

```
        if (count.get() > 0) { // 判断队列里是否还有数据
            x = dequeue(); // 删除头结点
            c = count.getAndDecrement(); // 元素个数-1
            if (c > 1) // 如果队列里还有元素
                notEmpty.signal(); // 在拿锁的条件对象notEmpty上唤醒正在等待的线程，表示队列里还有数据，可以再次消费
        }
    } finally {
        takeLock.unlock(); // 释放拿锁，让其他线程可以调用poll方法
    }
    if (c == capacity) // 由于存在放锁和拿锁，这里可能放锁一直在添加数据，count会变化。这里的if条件表示如果队列中还可以再插入数据
        signalNotFull(); // 在放锁的条件对象notFull上唤醒正在等待的1个线程，表示队列里还能再次添加数据
                return x;
}
```

take方法：

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly(); // 拿锁加锁，保证调用take方法的时候只有1个线程
    try {
        while (count.get() == 0) { // 如果队列里已经没有元素了
            notEmpty.await(); // 阻塞并挂起当前线程
        }
        x = dequeue(); // 删除头结点
        c = count.getAndDecrement(); // 元素个数-1
        if (c > 1) // 如果队列里还有元素
            notEmpty.signal(); // 在拿锁的条件对象notEmpty上唤醒正在等待的线程，表示队列里还有数据，可以再次消费
    } finally {
        takeLock.unlock(); // 释放拿锁，让其他线程可以调用take方法
    }
    if (c == capacity) // 由于存在放锁和拿锁，这里可能放锁一直在添加数据，count会变化。这里的if条件表示如果队列中还可以再插入数据
        signalNotFull(); // 在放锁的条件对象notFull上唤醒正在等待的1个线程，表示队列里还能再次添加数据
    return x;
}
```

remove方法：

```
public boolean remove(Object o) {
    if (o == null) return false;
    fullyLock(); // remove操作要移动的位置不固定，2个锁都需要加锁
    try {
        for (Node<E> trail = head, p = trail.next; // 从链表头结点开始遍历
             p != null;
             trail = p, p = p.next) {
            if (o.equals(p.item)) { // 判断是否找到对象
                unlink(p, trail); // 修改节点的链接信息，同时调用notFull的signal方法
                return true;
```

```
            }
        }
        return false;
    } finally {
        fullyUnlock(); // 2个锁解锁
    }
}
```

LinkedBlockingQueue的take方法对于没数据的情况下会阻塞，poll方法删除链表头结点，remove方法删除指定的对象。

需要注意的是remove方法由于要删除的数据的位置不确定，需要2个锁同时加锁。

进程通讯的方式：消息队列，共享内存，信号量，socket通讯等

Linux进程间通信方式--信号，管道，消息队列，信号量，共享内存

1、概述

通信方法	无法介于内核态与用户态的原因
管道（不包括命名管道）	局限于父子进程间的通信。
消息队列	在硬、软中断中无法无阻塞地接收数据。
信号量	无法介于内核态和用户态使用。
内存共享	需要信号量辅助，而信号量又无法使用。
套接字	在硬、软中断中无法无阻塞地接收数据。

2、信号

信号又称软终端，通知程序发生异步事件，程序执行中随时被各种信号中断，进程可以忽略该信号，也可以中断当前程序转而去处理信号，引起信号原因：

1).程序中执行错误码；

2).其他进程发送来的；

3).用户通过控制终端发送来；

4).子进程结束时向父进程发送SIGCLD；

5).定时器生产的SIGALRM；

3、管道

管道的优点是不需要加锁，缺点是默认缓冲区太小，只有4K，同时只适合父子进程间通信，而且一个管道只适合单向通信，如果要双向通信需要建立两个。而且不适合多个子进程，因为消息会乱，它的发送接收机制是用read/write这种适用流的，缺点是数据本身没有边界，需要应用程序自己解释，而一般消息大多是一个固定长的消息头，和一个变长的消息体，一个子进程从管道read到消息头后，消息体可能被别的子进程接收到

单向，一段输入，另一端输出，先进先出FIFO。管道也是文件。管道大小4096字节。

特点：管道满时，写阻塞；空时，读阻塞。

分类：普通管道（仅父子进程间通信）位于内存；命名管道位于文件系统，没有亲缘关系管道只要知道管道名也可以通讯。

管道是由内核管理的一个缓冲区(buffer)，相当于我们放入内存中的一个纸条。管道的一端连接一个进程的输出。这个进程会向管道中放入信息。管道的另一端连接一个进程的输入，这个进程取出被放入管道的信息。一个缓冲区不需要很大，它被设计成为环形的数据结构，以便管道可以被循环利用。当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。当管道被放满信息的时候，尝试放入信息的进程会等待，直到另一端的进程取出信息。当两个进程都终结的时候，管道也自动消失。

4、消息队列

消息队列也不要加锁，默认缓冲区和单消息上限都要大一些，在我的suse10上是64K，它并不局限于父子进程间通信，只要一个相同的key，就可以让不同的进程定位到同一个消息队列上，它也可以用来给双向通信，不过稍微加个标识，可以通过消息中的type进行区分，比如一个任务分派进程，创建了若干个执行子进程，不管是父进程发送分派任务的消息，还是子进程发送任务执行的消息，都将type设置为目标进程的pid，因为msgrcv可以指定只接收消息类型为type的消息，这样就实现了子进程只接收自己的任务，父进程只接收任务结果

消息队列是先进先出FIFO原则

消息结构模板

```
strut msgbuf
{
long int  mtype;//消息类型
char mtext[1];//消息内容
}
```

5、共享内存

共享内存的几乎可以认为没有上限，它也是不局限与父子进程，采用跟消息队列类似的定位方式，因为内存是共享的，不存在任何单向的限制，最大的问题就是需要应用程序自己做互斥，有如下几种方案

1 只适用两个进程共享，在内存中放一个标志位，一定要声明为volatile，大家基于标志位来互斥，例如为0时第一个可以写，第二个就等待，为1时第一个等待，第二个可以写/读

2 也只适用两个进程，是用信号，大家等待不同的信号，第一个写完了发送信号2，等待信号1，第二个等待信号2，收到后读取/写入完，发送信号1，它不是用更多进程是因为虽然父进程可以向不同子进程分别发送信号，但是子进程收到信号会同时访问共享内存，产生不同子进程间的竞态条件，如果用多块共享内存，又存在子进程发送结果通知信号时，父进程收到信号后，不知道是谁发送，也意味着不知道该

访问哪块共享内存，即使子进程发送不同的结果通知信号，因为等待信号的一定是阻塞的，如果某个子进程意外终止，父进程将永远阻塞下去，而不能超时处理

3 采用信号量或者msgctl自己的加锁、解锁功能，不过后者只适用于linux

6、信号量

信号量是一种用于提供不同进程间或一个进程间的不同线程间线程同步手段的原语，systemV信号量在内核中维护

二值信号量： 其值只有0、1 两种选择，0表示资源被锁，1表示资源可用；
计数信号量：其值在0 和某个限定值之间，不限定资源数只在0 1 之间；
计数信号量集； 多个信号量的集合组成信号量集

------总结

管道是最弱的，只适合有限场景；

消息队列能适合大部分场景，缺点是默认缓冲也比较小，不过这个可以调整，前提是你有管理员权限；

共享内存是最强大的，只是要做互斥

为什么要用线程池

在Java中，如果每当一个请求到达就创建一个新线程，开销是相当大的。在实际使用中，每个请求创建新线程的服务器在创建和销毁线程上花费的时间和消耗的系统资源，甚至可能要比花在处理实际的用户请求的时间和资源要多得多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个JVM里创建太多的线程，可能会导致系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，服务器应用程序需要一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务，这就是“池化资源”技术产生的原因。

线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使应用程序响应更快。另外，通过适当地调整线程池中的线程数目可以防止出现资源不足的情况。

创建一个线程池

一个比较简单的线程池至少应包含线程池管理器、工作线程、任务队列、任务接口等部分。其中线程池管理器（ThreadPool Manager）的作用是创建、销毁并管理线程池，将工作线程放入线程池中；工作线程是一个可以循环执行任务的线程，在没有任务时进行等待；任务队列的作用是提供一种缓冲机制，将没有处理的任务放在任务队列中；任务接口是每个任务必须实现的接口，主要用来规定任务的入口、任务执行完后的收尾工作、任务的执行状态等，工作线程通过该接口调度任务的执行。

线程池适合应用的场合

当一个Web服务器接受到大量短小线程的请求时，使用线程池技术是非常合适的，它可以大大减少线程的创建和销毁次数，提高服务器的工作效率。但如果线程要求的运行时间比较长，此时线程的运行时间比创建时间要长得多，单靠减少创建时间对系统效率的提高不明显，此时就不适合应用线程池技术，需要借助其它的技术来提高服务器的服务效率。

使用线程池的风险

虽然线程池是构建多线程应用程序的强大机制，但使用它并不是没有风险的。用线程池构建的应用程序容易遭受任何其它多线程应用程序容易遭受的所有并发风险，诸如同步错误和死锁，它还容易遭受特定于线程池的少数其它风险，诸如与池有关的死锁、资源不足和线程泄漏。

死锁

任何多线程应用程序都有死锁风险。当一组进程或线程中的每一个都在等待一个只有该组中另一个进程才能引起的事件时，我们就说这组进程或线程死锁了。死锁的最简单情形是：线程 A 持有对象 X 的独占锁，并且在等待对象 Y 的锁，而线程 B 持有对象 Y 的独占锁，却在等待对象 X 的锁。除非有某种方法来打破对锁的等待（Java 锁定不支持这种方法），否则死锁的线程将永远等下去。

虽然任何多线程程序中都有死锁的风险，但线程池却引入了另一种死锁可能，在那种情况下，所有池线程都在执行已阻塞的等待队列中另一任务的执行结果的任务，但这一任务却因为没有未被占用的线程而不能运行。当线程池被用来实现涉及许多交互对象的模拟，被模拟的对象可以相互发送查询，这些查询接下来作为排队的任务执行，查询对象又同步等待着响应时，会发生这种情况。

资源不足

线程池的一个优点在于：相对于其它替代调度机制（有些我们已经讨论过）而言，它们通常执行得很好。但只有恰当地调整了线程池大小时才是这样的。线程消耗包括内存和其它系统资源在内的大量资源。除了 Thread 对象所需的内存之外，每个线程都需要两个可能很大的执行调用堆栈。除此以外，JVM 可能会为每个 Java 线程创建一个本机线程，这些本机线程将消耗额外的系统资源。最后，虽然线程之间切换的调度开销很小，但如果有很多线程，环境切换也可能严重地影响程序的性能。

如果线程池太大，那么被那些线程消耗的资源可能严重地影响系统性能。在线程之间进行切换将会浪费时间，而且使用超出比您实际需要的线程可能会引起资源匮乏问题，因为池线程正在消耗一些资源，而这些资源可能会被其它任务更有效地利用。除了线程自身所使用的资源以外，服务请求时所做的工作可能需要其它资源，例如 JDBC 连接、套接字或文件。这些也都是有限资源，有太多的并发请求也可能引起失效，例如不能分配 JDBC 连接。

并发错误

线程池和其它排队机制依靠使用 wait() 和 notify() 方法，这两个方法都难于使用。如果编码不正确，那么可能丢失通知，导致线程保持空闲状态，尽管队列中有工作要处理。使用这些方法时，必须格外小心；即便是专家也可能在它们上面出错。而最好使用现有的、已经知道能工作的实现，例如在下面的无须编写您自己的池中讨论的 util.concurrent 包。

线程泄漏

各种类型的线程池中一个严重的风险是线程泄漏，当从池中除去一个线程以执行一项任务，而在任务完成后该线程却没有返回池时，会发生这种情况。发生线程泄漏的一种情形出现在任务抛出一个 RuntimeException 或一个 Error 时。如果池类没有捕捉到它们，那么线程只会退出而线程池的大小将会永久减少一个。当这种情况发生的次数足够多时，线程池最终就为空，而且系统将停止，因为没有可用的线程来处理任务。

有些任务可能会永远等待某些资源或来自用户的输入，而这些资源又不能保证变得可用，用户可能也已经回家了，诸如此类的任务会永久停止，而这些停止的任务也会引起和线程泄漏同样的问题。如果某个线程被这样一个任务永久地消耗着，那么它实际上就被从池除去了。对于这样的任务，应该要么只给予它们自己的线程，要么只让它们等待有限的时间。

请求过载

仅仅是请求就压垮了服务器，这种情况是可能的。在这种情形下，我们可能不想将每个到来的请求都排队到我们的工作队列，因为排在队列中等待执行的任务可能会消耗太多的系统资源并引起资源缺乏。在这种情形下决定如何做取决于您自己；在某些情况下，您可以简单地抛弃请求，依靠更高级别的协议稍后重试请求，您也可以用一个指出服务器暂时很忙的响应来拒绝请求。

有效使用线程池的准则

只要您遵循几条简单的准则，线程池可以成为构建服务器应用程序的极其有效的方法：

不要对那些同步等待其它任务结果的任务排队。这可能会导致上面所描述的那种形式的死锁，在那种死锁中，所有线程都被一些任务所占用，这些任务依次等待排队任务的结果，而这些任务又无法执行，因为所有的线程都很忙。

在为时间可能很长的操作使用合用的线程时要小心。如果程序必须等待诸如 I/O 完成这样的某个资源，那么请指定最长的等待时间，以及随后是失效还是将任务重新排队以便稍后执行。这样做保证了：通过将某个线程释放给某个可能成功完成的任务，从而将最终取得某些进展。

理解任务

要有效地调整线程池大小，您需要理解正在排队的任务以及它们正在做什么。它们是 CPU 限制的（CPU-bound）吗？它们是 I/O 限制的（I/O-bound）吗？您的答案将影响您如何调整应用程序。如果您有不同的任务类，这些类有着截然不同的特征，那么为不同任务类设置多个工作队列可能会有意义，这样可以相应地调整每个池。

调整池的大小

调整线程池的大小基本上就是避免两类错误：线程太少或线程太多。幸运的是，对于大多数应用程序来说，太多和太少之间的余地相当宽。

请回忆：在应用程序中使用线程有两个主要优点，尽管在等待诸如 I/O 的慢操作，但允许继续进行处理，并且可以利用多处理器。在运行于具有 N 个处理器机器上的计算限制的应用程序中，在线程数目接近 N 时添加额外的线程可能会改善总处理能力，而在线程数目超过 N 时添加额外的线程将不起作用。事实上，太多的线程甚至会降低性能，因为它会导致额外的环境切换开销。

线程池的最佳大小取决于可用处理器的数目以及工作队列中的任务的性质。若在一个具有 N 个处理器的系统上只有一个工作队列，其中全部是计算性质的任务，在线程池具有 N 或 N+1 个线程时一般会获得最大的 CPU 利用率。

对于那些可能需要等待 I/O 完成的任务（例如，从套接字读取 HTTP 请求的任务），需要让池的大小超过可用处理器的数目，因为并不是所有线程都一直在工作。通过使用概要分析，您可以估计某个典型请求的等待时间（WT）与服务时间（ST）之间的比例。如果我们将这一比例称之为 WT/ST，那么对于一个具有 N 个处理器的系统，需要设置大约 N*(1+WT/ST) 个线程来保持处理器得到充分利用。

处理器利用率不是调整线程池大小过程中的唯一考虑事项。随着线程池的增长，您可能会碰到调度程序、可用内存方面的限制，或者其它系统资源方面的限制，例如套接字、打开的文件句柄或数据库连接等的数目。

无须编写您自己的池

Doug Lea 编写了一个优秀的并发实用程序开放源码库 util.concurrent，它包括互斥、信号量、诸如在并发访问下执行得很好的队列和散列表之类集合类以及几个工作队列实现。该包中的 PooledExecutor 类是一种有效的、广泛使用的以工作队列为基础的线程池的正确实现。您无须尝试编写您自己的线程池，这样做容易出错，相反您可以考虑使用 util.concurrent 中的一些实用程序。
util.concurrent 库也激发了 JSR 166，JSR 166 是一个 Java 社区过程（Java Community Process (JCP)）工作组，他们正在打算开发一组包含在 java.util.concurrent 包下的 Java 类库中的并发实用程序，这个包应该用于 Java 开发工具箱 1.5 发行版。

UTIL.CONCURRENT包介绍

1概述

纽约奥斯维果州立大学的Doug Lea 编写了一个优秀的并发实用程序开放源码库 util.concurrent，它包括互斥、信号量、诸如在并发访问下执行得很好的队列和散列表之类集合类以及几个工作队列实现。该包中的 PooledExecutor 类是一种有效的、广泛使用的以工作队列为基础的线程池的正确实现。该工具包已经过了大量的测试和实践的考验，是个功能强大的成熟的工具包，在很多著名的服务器如oc4j、jboss里面您都可以找到util.concurrent的影子。Doug Lea 当时设计这个包的目标就是"简单的接口、高质量实现"。JSR 166（Java Community Process (JCP)）工作组已计划将 java.util.concurrent 包纳入 JDK 1.5 发行版，正在进行一些标准化的工作，让我们拭目以待。也许很多程序员愿意尝试自己写开发包，但是最终您会发现，您写的开发包也许并不比Doug Lea写的强大，甚至更容易出错。程序员的青春和生命是有限的，大量的时间和精力花在重复撰写已有的通用开发包，我认为这通常是不必要的，当然有志于挑战自我的和想深入研究该问题的读者除外。牛顿曾经很谦虚的说过，他的巨大的成功，是应为他站在巨人的肩膀上。所以，您并非必须编写您自己的线程池，这样做容易出错，相反您可以考虑使用 util.concurrent 中的一些实用程序。当然这里笔者并不是鼓励大家懒惰得思想，而是认为在具体的情况下，应该有所取舍，将宝贵的开发时间用在解决更重要的核心问题上来。在这里顺便提一下，Doug Lea 的 《Concurrent Programming in Java, Second Edition》是一本围绕 Java 应用程序中多线程编程方面可能出现的难解问题的权威著作，有时间的话您可找来读一读。

2　　框架与结构

下面让我们来看看util.concurrent的框架结构。关于这个工具包概述的e文原版链接地址是http://gee.cs.oswego.edu/dl/cpjslides/util.pdf。该工具包主要包括三大部分：同步、通道和线程池执行器。第一部分主要是用来定制锁，资源管理，其他的同步用途；通道则主要是为缓冲和队列服务的；线程池执行器则提供了一组完善的复杂的线程池实现。

--主要的结构如下图所示

2.1 Sync

acquire/release协议的主要接口

- 用来定制锁，资源管理，其他的同步用途
- 高层抽象接口
- 没有区分不同的加锁用法

实现

-Mutex, ReentrantLock, Latch, CountDown,Semaphore, WaiterPreferenceSemaphore, FIFOSemaphore, PrioritySemaphore

还有，有几个简单的实现，例如ObservableSync, LayeredSync

举例：如果我们要在程序中获得一独占锁，可以用如下简单方式：

```
try {
lock.acquire();
try {
action();
}
finally {
lock.release();
}
}catch（Exception e）{
}
```

程序中,使用lock对象的acquire()方法获得一独占锁，然后执行您的操作，锁用完后，使用release()方法释放之即可。呵呵，简单吧，想想看，如果您亲自撰写独占锁，大概会考虑到哪些问题？如果关键的锁得不到怎末办？用起来是不是会复杂很多？而现在，以往的很多细节和特殊异常情况在这里都无需多考虑，您尽可以把精力花在解决您的应用问题上去。

2.2 通道(Channel)
为缓冲，队列等服务的主接口

具体实现
LinkedQueue, BoundedLinkedQueue,BoundedBuffer, BoundedPriorityQueue,
SynchronousChannel, Slot

通道例子
```
class Service { // ...
final Channel msgQ = new LinkedQueue();
public void serve() throws InterruptedException {
String status = doService();
msgQ.put(status);
}
public Service() { // start background thread
Runnable logger = new Runnable() {
public void run() {
try {
for(;;)
System.out.println(msqQ.take());
}
catch(InterruptedException ie) {} }
};
new Thread(logger).start();
}
}
```
在后台服务器中，缓冲和队列都是最常用到的。试想，如果对所有远端的请求不排个队列，让它们一拥而上的去争夺cpu、内存、资源，那服务器瞬间不当掉才怪。而在这里，成熟的队列和缓冲实现已经提供，您只需要对其进行正确初始化并使用即可，大大缩短了开发时间。

2.3执行器(Executor)
Executor是这里最重要、也是我们往往最终写程序要用到的，下面重点对其进行介绍。
类似线程的类的主接口
- 线程池
- 轻量级运行框架
- 可以定制调度算法

只需要支持execute(Runnable r)
- 同Thread.start类似

实现
- PooledExecutor, ThreadedExecutor, QueuedExecutor, FJTaskRunnerGroup

PooledExecutor（线程池执行器）是个最常用到的类，以它为例：
可修改得属性如下：
- 任务队列的类型
- 最大线程数
- 最小线程数
- 预热(预分配)和立即(分配)线程
- 保持活跃直到工作线程结束

-- 以后如果需要可能被一个新的代替
- 饱和(Saturation)协议
-- 阻塞，丢弃，生产者运行，等等

可不要小看上面这数条属性，对这些属性的设置完全可以等同于您自己撰写的线程池的成百上千行代码。下面以笔者撰写过得一个GIS服务器为例：
该GIS服务器是一个典型的“请求 - 服务”类型的服务器，遵循后端程序设计的一般框架。首先对所有的请求按照先来先服务排入一个请求队列，如果瞬间到达的请求超过了请求队列的容量，则将溢出的请求转移至一个临时队列。如果临时队列也排满了，则对以后达到的请求给予一个“服务器忙”的提示后将其简单抛弃。这个就够忙活一阵的了。
然后，结合链表结构实现一个线程池，给池一个初始容量。如果该池满，以x2的策略将池的容量动态增加一倍，依此类推，直到总线程数服务达到系统能力上限，之后线程池容量不在增加，所有请求将等待一个空余的返回线程。每从池中得到一个线程，该线程就开始最请求进行GIS信息的服务，如取坐标、取地图，等等。服务完成后，该线程返回线程池继续为请求队列离地后续请求服务，周而复始。当时用矢量链表来暂存请求，用wait()、 notify() 和 synchronized等原语结合矢量链表实现线程池，总共约600行程序，而且在运行时间较长的情况下服务器不稳定，线程池被取用的线程有异常消失的情况发生。而使用util.concurrent相关类之后，仅用了几十行程序就完成了相同的工作而且服务器运行稳定，线程池没有丢失线程的情况发生。由此可见util.concurrent包极大的提高了开发效率，为项目节省了大量的时间。
使用PooledExecutor例子
```
import java.net.*;
/**
*

结束语

Title:
*

Description: 负责初始化线程池以及启动服务器
*

Copyright: Copyright (c) 2003
*

Company:
* @author not attributable
* @version 1.0
*/
public class MainServer {
//初始化常量
public static final int MAX_CLIENT=100; //系统最大同时服务客户数
//初始化线程池
public static final PooledExecutor pool =
new PooledExecutor(new BoundedBuffer(10), MAX_CLIENT); //chanel容量为10,
//在这里为线程池初始化了一个
//长度为10的任务缓冲队列。

public MainServer() {
//设置线程池运行参数
pool.setMinimumPoolSize(5); //设置线程池初始容量为5个线程
pool.discardOldestWhenBlocked();//对于超出队列的请求，使用了抛弃策略。
pool.createThreads(2); //在线程池启动的时候，初始化了具有一定生命周期的2个“热”线程
}
```

```
public static void main(String[] args) {
MainServer MainServer1 = new MainServer();
new HTTPListener().start();//启动服务器监听和处理线程
new manageServer().start();//启动管理线程
}
}
```

类HTTPListener

```
import java.net.*;
/**
*

Title:
*

Description: 负责监听端口以及将任务交给线程池处理
*

Copyright: Copyright (c) 2003
*

Company:
* @author not attributable
* @version 1.0
*/

public class HTTPListener extends Thread{
public HTTPListener() {
}
public void run(){
try{
ServerSocket server=null;
Socket clientconnection=null;
server = new ServerSocket(8008);//服务套接字监听某地址端口对
while(true){//无限循环
clientconnection =server.accept();
System.out.println("Client connected in!");
//使用线程池启动服务
MainServer.pool.execute(new HTTPRequest(clientconnection));//如果收到一个请求，则从线程池中取一个线程进行服务，任务完成后，该线程自动返还线程池
}
}catch(Exception e){
System.err.println("Unable to start serve listen:"+e.getMessage());
e.printStackTrace();
}
}
}
```

关于util.concurrent工具包就有选择的介绍到这，更详细的信息可以阅读这些java源代码的API文档。Doug Lea是个很具有“open”精神的作者，他将util.concurrent工具包的java源代码全部公布出来，有兴趣的读者可以下载这些源代码并细细品味。

线程池是组织服务器应用程序的有用工具。它在概念上十分简单，但在实现和使用一个池时，却需要注意几个问题，例如死锁、资源不足和 wait() 及 notify() 的复杂性。如果您发现您的应用程序需要线程池，那么请考虑使用 util.concurrent 中的某个 Executor 类，例如 PooledExecutor，而不用从头开始编写。如果您要自己创建线程来处理生存期很短的任务，那么您绝对应该考虑使用线程池来替代。

线程池的基础概念

core,maxPoolSize,keepalive

执行任务时

1. 如果线程池中线程数量 < core，新建一个线程执行任务；
2. 如果线程池中线程数量 >= core ,则将任务放入任务队列
3. 如果线程池中线程数量 >= core 且 < maxPoolSize，则创建新的线程；
4. 如果线程池中线程数量 > core ,当线程空闲时间超过了keepalive时，则会销毁线程；由此可见线程池的队列如果是无界队列，那么设置线程池最大数量是无效的；

自带线程池的各种坑

1. Executors.newFixedThreadPool(10);
   固定大小的线程池：
   它的实现new ThreadPoolExecutor(10, 10, 0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue());
   初始化一个指定线程数的线程池，其中corePoolSize == maximumPoolSize，使用LinkedBlockingQuene作为阻塞队列，当线程池没有可执行任务时，也不会释放线程。
   由于LinkedBlockingQuene的特性，这个队列是无界的，若消费不过来，会导致内存被任务队列占满，最终oom；
2. Executors.newCachedThreadPool();
   缓存线程池：
   它的实现new ThreadPoolExecutor(0,Integer.MAX_VALUE,60L, TimeUnit.SECONDS,new SynchronousQueue());
   初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到Integer.MAX_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销，因为线程池的最大值了Integer.MAX_VALUE，会导致无限创建线程；所以，使用该线程池时，一定要注意控制并发的任务数，否则创建大量的线程会导致严重的性能问题；
3. Executors.newSingleThreadExecutor()
   单线程线程池：
   同newFixedThreadPool线程池一样，队列用的是LinkedBlockingQueue无界队列，可以无限的往里面添加任务，直到内存溢出；

volatile关键字的用法：使多线程中的变量可见

Java并发编程：volatile关键字解析

volatile这个关键字可能很多朋友都听说过，或许也都用过。在Java 5之前，它是一个备受争议的关键字，因为在程序中使用它往往会导致出人意料的结果。在Java 5之后，volatile关键字才得以重获生机。

volatile关键字虽然从字面上理解起来比较简单，但是要用好不是一件容易的事情。由于volatile关键字是与Java的内存模型有关的，因此在讲述volatile关键之前，我们先来了解一下与内存模型相关的概念和知识，然后分析了volatile关键字的实现原理，最后给出了几个使用volatile关键字的场景。

以下是本文的目录大纲：

一.内存模型的相关概念

二.并发编程中的三个概念

三.Java内存模型

四..深入剖析volatile关键字

五.使用volatile关键字的场景

若有不正之处请多多谅解，并欢迎批评指正。

请尊重作者劳动成果，转载请标明原文链接：

一.内存模型的相关概念

大家都知道，计算机在执行程序时，每条指令都是在CPU中执行的，而执行指令过程中，势必涉及到数据的读取和写入。由于程序运行过程中的临时数据是存放在主存（物理内存）当中的，这时就存在一个问题，由于CPU执行速度很快，而从内存读取数据和向内存写入数据的过程跟CPU执行指令的速度比起来要慢的多，因此如果任何时候对数据的操作都要通过和内存的交互来进行，会大大降低指令执行的速度。因此在CPU里面就有了高速缓存。

也就是，当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。举个简单的例子，比如下面的这段代码：

```
i = i + 1;
```

当线程执行这个语句时，会先从主存当中读取i的值，然后复制一份到高速缓存当中，然后CPU执行指令对i进行加1操作，然后将数据写入高速缓存，最后将高速缓存中i最新的值刷新到主存当中。

这个代码在单线程中运行是没有任何问题的，但是在多线程中运行就会有问题了。在多核CPU中，每条线程可能运行于不同的CPU中，因此每个线程运行时有自己的高速缓存（对单核CPU来说，其实也会出现这种问题，只不过是以线程调度的形式来分别执行的）。本文我们以多核CPU为例。

比如同时有2个线程执行这段代码，假如初始时i的值为0，那么我们希望两个线程执行完之后i的值变为2。但是事实会是这样吗？

可能存在下面一种情况：初始时，两个线程分别读取i的值存入各自所在的CPU的高速缓存当中，然后线程1进行加1操作，然后把i的最新值1写入到内存。此时线程2的高速缓存当中i的值还是0，进行加1操作之后，i的值为1，然后线程2把i的值写入内存。

最终结果i的值是1，而不是2。这就是著名的缓存一致性问题。通常称这种被多个线程访问的变量为共享变量。

也就是说，如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题。

为了解决缓存不一致性问题，通常来说有以下2种解决方法：

1）通过在总线加LOCK#锁的方式

2）通过缓存一致性协议

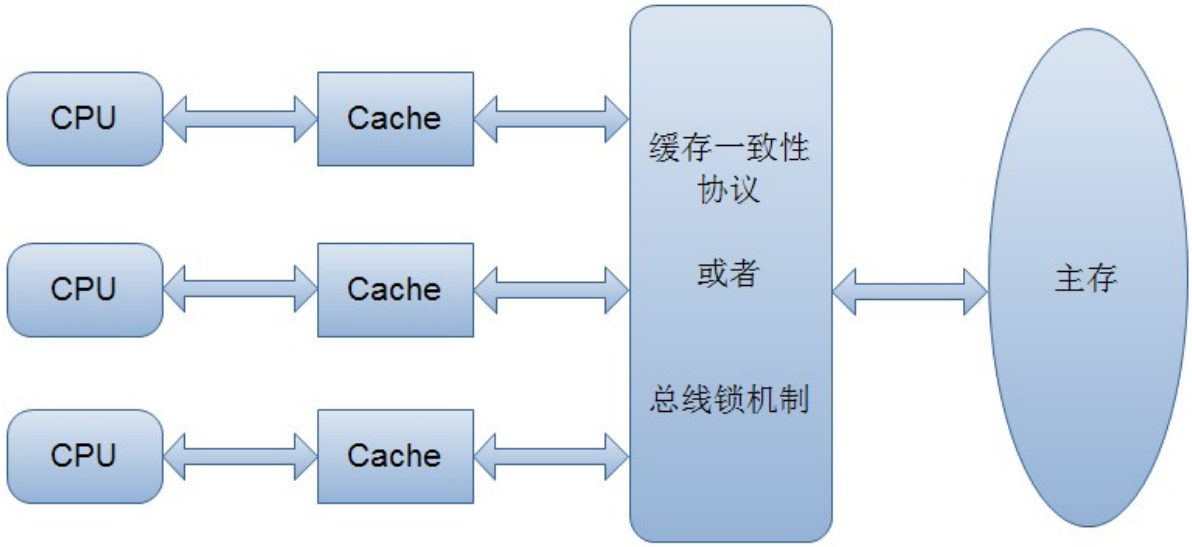
这2种方式都是硬件层面上提供的方式。

在早期的CPU当中，是通过在总线上加LOCK#锁的形式来解决缓存不一致的问题。因为CPU和其他部件进行通信都是通过总线来进行的，如果对总线加LOCK#锁的话，也就是说阻塞了其他CPU对其他部件访问（如内存），从而使得只能有一个CPU能使用这个变量的内存。比如上面例子中 如果一个线程在执行 i = i +1，如果在执行这段代码的过程中，在总线上发出了LCOK#锁的信号，那么只有等待这段代码

完全执行完毕之后，其他CPU才能从变量i所在的内存读取变量，然后进行相应的操作。这样就解决了缓存不一致的问题。

但是上面的方式会有一个问题，由于在锁住总线期间，其他CPU无法访问内存，导致效率低下。

所以就出现了缓存一致性协议。最出名的就是Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。



二.并发编程中的三个概念

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。我们先看具体看一下这三个概念：

1.原子性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

一个很经典的例子就是银行账户转账问题：

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。

试想一下，如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。然后又从B取出了500元，取出500元之后，再执行 往账户B加上1000元 的操作。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

所以这2个操作必须要具备原子性才能保证不出现一些意外的问题。

同样地反映到并发编程中会出现什么结果呢?

举个最简单的例子，大家想一下假如为一个32位的变量赋值过程不具备原子性的话，会发生什么后果?

```
i = 9;
```

假若一个线程执行到这个语句时，我暂且假设为一个32位的变量赋值包括两个过程：为低16位赋值，为高16位赋值。

那么就可能发生一种情况：当将低16位数值写入之后，突然被中断，而此时又有一个线程去读取i的值，那么读取到的就是错误的数据。

2.可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

举个简单的例子，看下面这段代码：

```
//线程1执行的代码
int i = 0;i = 10;

//线程2执行的代码
j = i;
```

假若执行线程1的是CPU1，执行线程2的是CPU2。由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到CPU1的高速缓存中，然后赋值为10，那么在CPU1的高速缓存当中i的值变为10了，却没有立即写入到主存当中。

此时线程2执行 j = i，它会先去主存读取i的值并加载到CPU2的缓存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10.

这就是可见性问题，线程1对变量i修改了之后，线程2没有立即看到线程1修改的值。

3.有序性

有序性：即程序执行的顺序按照代码的先后顺序执行。举个简单的例子，看下面这段代码：

```
int i = 0;
boolean flag = false;
i = 1;         //语句1
flag = true;      //语句2
```

上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操作。从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

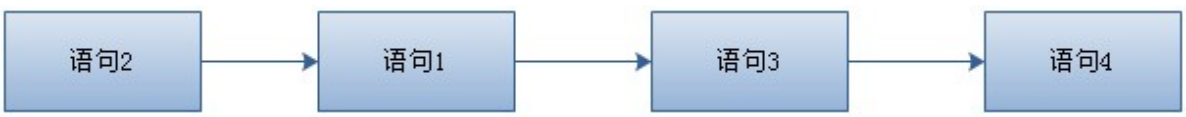
下面解释一下什么是指令重排序，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

但是要注意，虽然处理器会对指令进行重排序，但是它会保证程序最终结果会和代码顺序执行结果相同，那么它靠什么保证的呢？再看下面一个例子：

```
int a = 10;  //语句1
int r = 2;  //语句2
a = a + 3;  //语句3
r = a*a;   //语句4
```

这段代码有4个语句，那么可能的一个执行顺序是：



那么可不可能是这个执行顺序呢： 语句2 语句1 语句4 语句3

不可能，因为处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行。

虽然重排序不会影响单个线程内程序执行的结果，但是多线程呢？下面看一个例子：

```
//线程1:
context = loadContext();  //语句1
inited = true;        //语句2
//线程2:
while(!inited ){
    sleep()
}
doSomethingwithconfig(context);
```

上面代码中，由于语句1和语句2没有数据依赖性，因此可能会被重排序。假如发生了重排序，在线程1执行过程中先执行语句2，而此是线程2会以为初始化工作已经完成，那么就会跳出while循环，去执行doSomethingwithconfig(context)方法，而此时context并没有被初始化，就会导致程序出错。

从上面可以看出，指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。

也就是说，要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

三.Java内存模型

在前面谈到了一些关于内存模型以及并发编程中可能会出现的一些问题。下面我们来看一下Java内存模型，研究一下Java内存模型为我们提供了哪些保证以及在java中提供了哪些方法和机制来让我们在进行多线程编程时能够保证程序执行的正确性。

在Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽各个硬件平台和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。那么Java内存模型规定了哪些东西呢，它定义了程序中变量的访问规则，往大一点说是定义了程序执行的次序。注意，为了获得较好的执行性能，Java内存模型并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序。也就是说，在java内存模型中，也会存在缓存一致性问题和指令重排序的问题。

Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。

举个简单的例子：在java中，执行下面这个语句：

```
i = 10;
```

执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值10写入主存当中。

那么Java语言 本身对 原子性、可见性以及有序性提供了哪些保证呢？

1.原子性

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

上面一句话虽然看起来简单，但是理解起来并不是那么容易。看下面一个例子i：

请分析以下哪些操作是原子性操作：

```
x = 10;     //语句1
y = x;     //语句2
x++;      //语句3
x = x + 1;   //语句4
```

咋一看，有些朋友可能会说上面的4个语句中的操作都是原子性操作。其实只有语句1是原子性操作，其他三个语句都不是原子性操作。

语句1是直接将数值10赋值给x，也就是说线程执行这个语句的会直接将数值10写入到工作内存中。

语句2实际上包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及 将x的值写入工作内存 这2个操作都是原子性操作，但是合起来就不是原子性操作了。

同样的，x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。

所以上面4个语句只有语句1的操作具备原子性。

也就是说，只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。

不过这里有一点需要注意：在32位平台下，对64位数据的读取和赋值是需要通过两个操作来完成的，不能保证其原子性。但是好像在最新的JDK中，JVM已经保证对64位数据的读取和赋值也是原子性操作了。

从上面可以看出，Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。

2.可见性

对于可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

3.有序性

在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

下面就来具体介绍下happens-before原则（先行发生原则）：

- 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
- 锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作
- volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
- 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C
- 线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作
- 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生
- 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行
- 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

这8条原则摘自《深入理解Java虚拟机》。

这8条规则中，前4条规则是比较重要的，后4条规则都是显而易见的。

下面我们来解释一下前4条规则：

对于程序次序规则来说，我的理解就是一段程序代码的执行在单个线程中看起来是有序的。注意，虽然这条规则中提到“书写在前面的操作先行发生于书写在后面的操作”，这个应该是程序看起来执行的顺序是按照代码顺序执行的，因为虚拟机可能会对程序代码进行指令重排序。虽然进行重排序，但是最终执行的结果是与程序顺序执行的结果一致的，它只会对不存在数据依赖性的指令进行重排序。因此，在单个线程中，程序执行看起来是有序执行的，这一点要注意理解。事实上，这个规则是用来保证程序在单线程中执行结果的正确性，但无法保证程序在多线程中执行的正确性。

第二条规则也比较容易理解，也就是说无论在单线程中还是多线程中，同一个锁如果出于被锁定的状态，那么必须先对锁进行了释放操作，后面才能继续进行lock操作。

第三条规则是一条比较重要的规则，也是后文将要重点讲述的内容。直观地解释就是，如果一个线程先去写一个变量，然后一个线程去进行读取，那么写入操作肯定会先行发生于读操作。

第四条规则实际上就是体现happens-before原则具备传递性。

四.深入剖析volatile关键字

在前面讲述了很多东西，其实都是为讲述volatile关键字作铺垫，那么接下来我们就进入主题。

1.volatile关键字的两层语义

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序。

先看一段代码，假如线程1先执行，线程2后执行：

```
//线程1
boolean stop = false;
while(!stop){
    doSomething();
}
//线程2
stop = true;
```

这段代码是很典型的一段代码，很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

下面解释一下这段代码为何有可能导致无法中断线程。在前面已经解释过，每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，会将stop变量的值拷贝一份放在自己的工作内存当中。

那么当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

但是用volatile修饰之后就变得不一样了：

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

那么在线程2修改stop值时（当然这里包括2个操作，修改线程2工作内存中的值，然后将修改后的值写入内存），会使得线程1的工作内存中缓存变量stop的缓存行无效，然后线程1读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

那么线程1读取到的就是最新的正确的值。

2.volatile保证原子性吗?

从上面知道volatile关键字保证了操作的可见性，但是volatile能保证对变量的操作是原子性吗?

下面看一个例子：

```
public class Test {
public volatile int inc = 0;

public void increase() {
    inc++;
}

public static void main(String[] args) {
    final Test test = new Test();
    for(int i=0;i<10;i++){
        new Thread(){
            public void run() {
                for(int j=0;j<1000;j++)
                    test.increase();
            };
        }.start();
    }
    while(Thread.activeCount()>1) //保证前面的线程都执行完
        Thread.yield();
    System.out.println(test.inc);
    }
}
```

大家想一下这段程序的输出结果是多少？也许有些朋友认为是10000。但是事实上运行它会发现每次运行结果都不一致，都是一个小于10000的数字。

可能有的朋友就会有疑问，不对啊，上面是对变量inc进行自增操作，由于volatile保证了可见性，那么在每个线程中对inc自增完之后，在其他线程中都能看到修改后的值啊，所以有10个线程分别进行了1000次操作，那么最终inc的值应该是1000*10=10000。

这里面就有一个误区了，volatile关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性。

在前面已经提到过，自增操作是不具备原子性的，它包括读取变量的原始值、进行加1操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

假如某个时刻变量inc的值为10，

线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

那么两个线程分别进行了一次自增操作后，inc只增加了1。

解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happens-before规则中的volatile变量规则，但是要注意，线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是线程1没有进行修改，所以线程2根本就不会看到修改的值。

根源就在这里，自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的。

把上面的代码改成以下任何一种都可以达到效果：

采用synchronized：

View Code

采用Lock：

View Code

采用AtomicInteger：

View Code

在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。atomic是利用CAS来实现原子性操作的（Compare And Swap），CAS实际上是利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作。

3.volatile能保证有序性吗?

在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。

volatile关键字禁止指令重排序有两层意思：

1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

可能上面说的比较绕，举个简单的例子：

```
//x、y为非volatile变量
//flag为volatile变量

x = 2;    //语句1
y = 0;    //语句2
flag = true; //语句3
x = 4;     //语句4
y = -1;    //语句5
```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

并且volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

那么我们回到前面举的一个例子：

```
//线程1:
context = loadContext();  //语句1
inited = true;        //语句2

//线程2:
while(!inited ){
    sleep()
}
doSomethingwithconfig(context);
```

前面举这个例子的时候，提到有可能语句2会在语句1之前执行，那么久可能导致context还没被初始化，而线程2中就使用未初始化的context去进行操作，导致程序出错。

这里如果用volatile关键字对inited变量进行修饰，就不会出现这种问题了，因为当执行到语句2时，必定能保证context已经初始化完毕。

4.volatile的原理和实现机制

前面讲述了源于volatile关键字的一些使用，下面我们来探讨一下volatile到底如何保证可见性和禁止指令重排序的。

下面这段话摘自《深入理解Java虚拟机》：

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

2）它会强制将对缓存的修改操作立即写入主存；

3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

五.使用volatile关键字的场景

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

事实上，我的理解就是上面的2个条件需要保证操作是原子性操作，才能保证使用volatile关键字的程序在并发时能够正确执行。

下面列举几个Java中使用volatile的几个场景。

1.状态标记量

```
volatile boolean flag = false;

while(!flag){
    doSomething();
}

public void setFlag() {
    flag = true;
}
```

```
volatile boolean inited = false;
//线程1:
context = loadContext();
inited = true;

//线程2:
while(!inited ){
    sleep()
}
doSomethingwithconfig(context);
```

2.double check

```
class Singleton{
private volatile static Singleton instance = null;
private Singleton() {

}

    public static Singleton getInstance() {
        if(instance==null) {
            synchronized (Singleton.class) {
                if(instance==null)
                    instance = new Singleton();
```

```
            }
        }
        return instance;
    }
}
```

线程的几种状态

线程在一定条件下，状态会发生变化。线程一共有以下几种状态：

1. 新建状态(New)：新创建了一个线程对象。
2. 就绪状态(Runnable)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待获取CPU的使用权。即在就绪状态的进程除CPU之外，其它的运行所需资源都已全部获得。
3. 运行状态(Running)：就绪状态的线程获取了CPU，执行程序代码。
4. 阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

   阻塞的情况分三种：

   1. 等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，
   2. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。
   3. 其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。
5. 死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

常用的线程池模式以及不同线程池的使用场景

java线程池与五种常用线程池策略使用与解析

一.线程池

关于为什么要使用线程池久不赘述了，首先看一下java中作为线程池Executor底层实现类的ThredPoolExecutor的构造函数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
        ...
    }
```

其中各个参数含义如下：
corePoolSize- 池中所保存的线程数，包括空闲线程。需要注意的是在初创建线程池时线程不会立即启动，直到有任务提交才开始启动线程并逐渐时线程数目达到corePoolSize。若想一开始就创建所有核心线程需调用prestartAllCoreThreads方法。
maximumPoolSize-池中允许的最大线程数。需要注意的是当核心线程满且阻塞队列也满时才会判断当前线程数是否小于最大线程数，并决定是否创建新线程。
keepAliveTime - 当线程数大于核心时，多于的空闲线程最多存活时间
unit - keepAliveTime 参数的时间单位。
workQueue - 当线程数目超过核心线程数时用于保存任务的队列。主要有3种类型的BlockingQueue可供选择：无界队列，有界队列和同步移交。将在下文中详细阐述。从参数中可以看到，此队列仅保存实现Runnable接口的任务。
threadFactory - 执行程序创建新线程时使用的工厂。
handler - 阻塞队列已满且线程数达到最大值时所采取的饱和策略。java默认提供了4种饱和策略的实现方式：中止、抛弃、抛弃最旧的、调用者运行。将在下文中详细阐述。

二.可选择的阻塞队列BlockingQueue详解

首先看一下新任务进入时线程池的执行策略：
如果运行的线程少于corePoolSize，则 Executor始终首选添加新的线程，而不进行排队。（如果当前运行的线程小于corePoolSize，则任务根本不会存入queue中，而是直接运行）
如果运行的线程大于等于 corePoolSize，则 Executor始终首选将请求加入队列，而不添加新的线程。
如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。
主要有3种类型的BlockingQueue：

2.1 无界队列

队列大小无限制，常用的为无界的LinkedBlockingQueue，使用该队列做为阻塞队列时要尤其当心，当任务耗时较长时可能会导致大量新任务在队列中堆积最终导致OOM。最近工作中就遇到因为采用LinkedBlockingQueue作为阻塞队列，部分任务耗时80s + 且不停有新任务进来，导致cpu和内存飙升服务器挂掉。

2.2 有界队列

常用的有两类，一类是遵循FIFO原则的队列如ArrayBlockingQueue与有界的LinkedBlockingQueue，另一类是优先级队列如PriorityBlockingQueue。PriorityBlockingQueue中的优先级由任务的Comparator决定。
使用有界队列时队列大小需和线程池大小互相配合，线程池较小有界队列较大时可减少内存消耗，降低cpu使用率和上下文切换，但是可能会限制系统吞吐量。

2.3 同步移交

如果不希望任务在队列中等待而是希望将任务直接移交给工作线程，可使用SynchronousQueue作为等待队列。SynchronousQueue不是一个真正的队列，而是一种线程之间移交的机制。要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接收这个元素。只有在使用无界线程池或者有饱和策略时才建议使用该队列。

2.4 几种BlockingQueue的具体实现原理

关于上述几种BlockingQueue的具体实现原理与分析将在下篇博文中详细阐述。

三.可选择的饱和策略RejectedExecutionHandler详解

JDK主要提供了4种饱和策略供选择。4种策略都做为静态内部类在ThreadPoolExcutor中进行实现。

3.1 AbortPolicy中止策略

该策略是默认饱和策略。

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
          throw new RejectedExecutionException("Task " + r.toString() +
                                               " rejected from " +
                                               e.toString());
      }
```

使用该策略时在饱和时会抛出RejectedExecutionException（继承自RuntimeException），调用者可捕获该异常自行处理。

3.2 DiscardPolicy抛弃策略

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
      }
```

如代码所示，不做任何处理直接抛弃任务

3.3 DiscardOldestPolicy抛弃旧任务策略

```
 public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            e.getQueue().poll();
            e.execute(r);
        }
    }
```

如代码，先将阻塞队列中的头元素出队抛弃，再尝试提交任务。如果此时阻塞队列使用PriorityBlockingQueue优先级队列，将会导致优先级最高的任务被抛弃，因此不建议将该种策略配合优先级队列使用。

3.4 CallerRunsPolicy调用者运行

```
 public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
```

既不抛弃任务也不抛出异常，直接运行任务的run方法，换言之将任务回退给调用者来直接运行。使用该策略时线程池饱和后将由调用线程池的主线程自己来执行任务，因此在执行任务的这段时间里主线程无法再提交新任务，从而使线程池中工作线程有时间将正在处理的任务处理完成。

四.java提供的四种常用线程池解析

在JDK帮助文档中，有如此一段话：

> 强烈建议程序员使用较为方便的Executors工厂方法Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）、Executors.newFixedThreadPool(int)（固定大小线程池）Executors.newSingleThreadExecutor()（单个后台线程）它们均为大多数使用场景预定义了设置。

详细介绍一下上述四种线程池。

4.1 newCachedThreadPool

```
     public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
    }
```

在newCachedThreadPool中如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
初看该构造函数时我有这样的疑惑：核心线程池为0，那按照前面所讲的线程池策略新任务来临时无法进入核心线程池，只能进入 SynchronousQueue中进行等待，而SynchronousQueue的大小为1，那岂不是第一个任务到达时只能等待在队列中，直到第二个任务到达发现无法进入队列才能创建第一个线程？
这个问题的答案在上面讲SynchronousQueue时其实已经给出了，要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接收这个元素。因此即便SynchronousQueue一开始为空且大小为1，第一个任务也无法放入其中，因为没有线程在等待从SynchronousQueue中取走元素。因此第一个任务到达时便会创建一个新线程执行该任务。
这里引申出一个小技巧：有时我们可能希望线程池在没有任务的情况下销毁所有的线程，既设置线程池核心大小为0，但又不想使用SynchronousQueue而是想使用有界的等待队列。显然，不进行任何特殊设置的话这样的用法会发生奇怪的行为：直到等待队列被填满才会有新线程被创建，任务才开始执行。这并不是我们希望看到的，此时可通过allowCoreThreadTimeOut使等待队列中的元素出队被调用执行，详细原理和使用将会在后续博客中阐述。

4.2 newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

```
  public static ExecutorService newFixedThreadPool(int nThreads) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
    }
```

看代码一目了然了，使用固定大小的线程池并使用无限大的队列

4.3 newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

```
   public static ScheduledExecutorService newScheduledThreadPool(int
corePoolSize) {
        return new ScheduledThreadPoolExecutor(corePoolSize);
    }
```

在来看看ScheduledThreadPoolExecutor（）的构造函数

```
  public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue());
    }
```

ScheduledThreadPoolExecutor的父类即ThreadPoolExecutor，因此这里各参数含义和上面一样。值得关心的是DelayedWorkQueue这个阻塞对列，在上面没有介绍，它作为静态内部类就在ScheduledThreadPoolExecutor中进行了实现。具体分析讲会在后续博客中给出，在这里只进行简单说明：DelayedWorkQueue是一个无界队列，它能按一定的顺序对工作队列中的元素进行排列。因此这里设置的最大线程数 Integer.MAX_VALUE没有任何意义。关于ScheduledThreadPoolExecutor的具体使用将会在后续quartz的周期性任务实现原理中进行进一步分析。

4.4 newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

```
   public static ScheduledExecutorService newSingleThreadScheduledExecutor() {
        return new DelegatedScheduledExecutorService
            (new ScheduledThreadPoolExecutor(1));
    }
```

首先new了一个线程数目为1的ScheduledThreadPoolExecutor，再把该对象传入DelegatedScheduledExecutorService中，看看DelegatedScheduledExecutorService的实现代码：

```
DelegatedScheduledExecutorService(ScheduledExecutorService executor) {
            super(executor);
            e = executor;
        }
```

在看看它的父类

```
DelegatedExecutorService(ExecutorService executor) { e = executor; }
```

其实就是使用装饰模式增强了ScheduledExecutorService（1）的功能，不仅确保只有一个线程顺序执行任务，也保证线程意外终止后会重新创建一个线程继续执行任务。具体实现原理会在后续博客中讲解。

4.5 newWorkStealingPool创建一个拥有多个任务队列（以便减少连接数）的线程池。

这是jdk1.8中新增加的一种线程池实现，先看一下它的无参实现

```
    public static ExecutorService newWorkStealingPool() {
        return new ForkJoinPool
            (Runtime.getRuntime().availableProcessors(),
             ForkJoinPool.defaultForkJoinWorkerThreadFactory,
             null, true);
    }
```

返回的ForkJoinPool从jdk1.7开始引进，个人感觉类似于mapreduce的思想。这个线程池较为特殊，将在后续博客中给出详细的使用说明和原理。

线程间通信，wait和notify

wait和notify的理解与使用

1.对于wait()和notify()的理解

对于wait()和notify()的理解，还是要从jdk官方文档中开始，在Object类方法中有：

> **void notify()**
> Wakes up a single thread that is waiting on this object's monitor.
> 译：唤醒在此对象监视器上等待的单个线程
>
> **void notifyAll()**
> Wakes up all threads that are waiting on this object's monitor.
> 译：唤醒在此对象监视器上等待的所有线程
>
> **void wait( )**
> Causes the current thread to wait until another thread invokes the notify() method or the notifyAll( ) method for this object.
> 译：导致当前的线程等待，直到其他线程调用此对象的notify( ) 方法或 notifyAll( ) 方法
>
> **void wait(long timeout)**
> Causes the current thread to wait until either another thread invokes the notify( ) method or the notifyAll( ) method for this object, or a specified amount of time has elapsed.
> 译：导致当前的线程等待，直到其他线程调用此对象的notify() 方法或 notifyAll() 方法，或者指定的时间过完。
>
> **void wait(long timeout, int nanos)**
> Causes the current thread to wait until another thread invokes the notify( ) method or the notifyAll( ) method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.
> 译：导致当前的线程等待，直到其他线程调用此对象的notify( ) 方法或 notifyAll( ) 方法，或者其他线程打断了当前线程，或者指定的时间过完。

上面是官方文档的简介，下面我们根据官方文档总结一下：

- wait( )，notify( )，notifyAll( )都不属于Thread类，而是属于Object基础类，也就是每个对象都有wait( )，notify( )，notifyAll( ) 的功能，因为每个对象都有锁，锁是每个对象的基础，当然操作锁的方法也是最基础了。
- 当需要调用以上的方法的时候，一定要对竞争资源进行加锁，如果不加锁的话，则会报IllegalMonitorStateException 异常
- 当想要调用wait( )进行线程等待时，必须要取得这个锁对象的控制权（对象监视器），一般是放到synchronized(obj)代码中。
- 在while循环里而不是if语句下使用wait，这样，会在线程暂停恢复后都检查wait的条件，并在条件实际上并未改变的情况下处理唤醒通知
- 调用obj.wait( )释放了obj的锁，否则其他线程也无法获得obj的锁，也就无法在synchronized(obj){ obj.notify() } 代码段内唤醒A。
- notify( )方法只会通知等待队列中的第一个相关线程（不会通知优先级比较高的线程）
- notifyAll( )通知所有等待该竞争资源的线程（也不会按照线程的优先级来执行）
- 假设有三个线程执行了obj.wait( )，那么obj.notifyAll( )则能全部唤醒tread1，thread2，thread3，但是要继续执行obj.wait（）的下一条语句，必须获得obj锁，因此，tread1，

thread2，thread3只有一个有机会获得锁继续执行，例如tread1，其余的需要等待thread1释放obj锁之后才能继续执行。

- 当调用obj.notify/notifyAll后，调用线程依旧持有obj锁，因此，thread1，thread2，thread3虽被唤醒，但是仍无法获得obj锁。直到调用线程退出synchronized块，释放obj锁后，thread1，thread2，thread3中的一个才有机会获得锁继续执行。

2.wait和notify简单使用示例

```
public class WaitNotifyTest {

    // 在多线程间共享的对象上使用wait
    private String[] shareObj = { "true" };

    public static void main(String[] args) {
        WaitNotifyTest test = new WaitNotifyTest();
        ThreadWait threadWait1 = test.new ThreadWait("wait thread1");
        threadWait1.setPriority(2);
        ThreadWait threadWait2 = test.new ThreadWait("wait thread2");
        threadWait2.setPriority(3);
        ThreadWait threadWait3 = test.new ThreadWait("wait thread3");
        threadWait3.setPriority(4);

        ThreadNotify threadNotify = test.new ThreadNotify("notify thread");

        threadNotify.start();
        threadWait1.start();
        threadWait2.start();
        threadWait3.start();
    }

    class ThreadWait extends Thread {

        public ThreadWait(String name){
            super(name);
        }

        public void run() {
            synchronized (shareObj) {
                while ("true".equals(shareObj[0])) {
                    System.out.println("线程"+ this.getName() + "开始等待");
                    long startTime = System.currentTimeMillis();
                    try {
                        shareObj.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    long endTime = System.currentTimeMillis();
                    System.out.println("线程" + this.getName()
                            + "等待时间为：" + (endTime - startTime));
                }
            }
            System.out.println("线程" + getName() + "等待结束");
        }
    }

    class ThreadNotify extends Thread {
```

```
        public ThreadNotify(String name){
            super(name);
        }


        public void run() {
            try {
                // 给等待线程等待时间
                sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (shareObj) {
                System.out.println("线程" + this.getName() + "开始准备通知");
                shareObj[0] = "false";
                shareObj.notifyAll();
                System.out.println("线程" + this.getName() + "通知结束");
            }
            System.out.println("线程" + this.getName() + "运行结束");
        }
    }
}
```

运行结果：

```
线程wait thread1开始等待
线程wait thread3开始等待
线程wait thread2开始等待
线程notify thread开始准备通知
线程notify thread通知结束
线程notify thread运行结束
线程wait thread2等待时间为：2998
线程wait thread2等待结束
线程wait thread3等待时间为：2998
线程wait thread3等待结束
线程wait thread1等待时间为：3000
线程wait thread1等待结束
```

java线程池主线程等待子线程执行完成

Java如何等待子线程执行结束

今天讨论一个入门级的话题, 不然没东西更新对不起空间和域名~~

工作总往往会遇到异步去执行某段逻辑, 然后先处理其他事情, 处理完后再把那段逻辑的处理结果进行汇总的产景, 这时候就需要使用线程了.

一个线程启动之后, 是异步的去执行需要执行的内容的, 不会影响主线程的流程, 往往需要让主线程指定后, 等待子线程的完成. 这里有几种方式.

站在 主线程的角度, 我们可以分为主动式和被动式.

主动式指主线主动去检测某个标志位, 判断子线程是否已经完成. 被动式指主线程被动的等待子线程的结束, 很明显, 比较符合人们的胃口. 就是你事情做完了, 你告诉我, 我汇总一下, 哈哈.

那么主线程如何等待子线程工作完成呢. 很简单, Thread 类给我们提供了join 系列的方法, 这些方法的目的就是等待当前线程的die. 举个例子.

public class Threads **{**

```
**public**  **static**  **void**  main (String[]  args ) {

  SubThread **thread**  =  **new**  SubThread () ;

 thread . start () ;

 //主线程处理其他工作,让子线程异步去执行.

  *mainThreadOtherWork* () ;

  System . *out* . println ( “now waiting sub thread done.” ) ;

 //主线程其他工作完毕,等待子线程的结束, 调用join系列的方法即可(可以设置超时时间)

  **try**  {

   thread . join () ;

 }  **catch**  **(** InterruptedException **e) {**

   e . printStackTrace () ;

 }

 System . *out* . println ( “now all done.” ) ;
```

}

```
**private**  **static**  **void**  mainThreadOtherWork () {

 System . *out* . println ( “main thread work start” ) ;

  **try**  {

    Thread . *sleep* ( 3000L ) ;

 }  **catch**  **(** InterruptedException **e) {**

   e . printStackTrace () ;

 }

  System . *out* . println ( “main thread work done.” ) ;
```

}

```
**public**  **static**  **class**  SubThread  **extends**  Thread**{**
```

```
  @Override
  **public**  **void**  run () {
    working () ;
  }
  **private**  **void**  working () {
    System . *out* . println ( “sub thread start working.” ) ;
    busy () ;
    System . *out* . println ( “sub thread stop working.” ) ;
  }
  **private**  **void**  busy () {
    **try**  {
       *sleep* ( 5000L ) ;
    }  **catch**  **(** InterruptedException **e) {**
      e . printStackTrace () ;
    }
  }
```

```
  }
}
```

本程序的数据有可能是如下:

> 1. main thread work start
> 2. sub thread start working.
> 3. main thread work done.
> 4. now waiting sub thread done.
> 5. sub thread stop working.
> 6. now all done.

忽略标号, 当然输出也有可能是1和2调换位置了. 这个我们是无法控制的. 我们看下线程的join操作, 究竟干了什么.

**
**

```
  public final void join() throws InterruptedException {
  join(0) ;
 }
```

这里是调用了

public final synchronized void join(**long** millis)

方法, 参数为0, 表示没有超时时间, 等到线程结束为止. join(millis)方法里面有这么一段代码:

```
**while** (isAlive()) {

wait(0) ;

}
```

说明, 当线程处于活跃状态的时候, 会一直等待, 直到这里的isAlive方法返回false, 才会结束.isAlive方法是一个本地方法, 他的作用是判断线程是否已经执行结束. 注释是这么写的:

Tests if this thread is alive. A thread is alive if it has been started and has not yet died.

可见, join系列方法可以帮助我们等待一个子线程的结束.

那么要问, 有没有另外一种方法可以等待子线程结束? 当然有的, 我们可以使用并发包下面的Future模式.

Future是一个任务执行的结果, 他是一个将来时, 即一个任务执行, 立即异步返回一个Future对象, 等到任务结束的时候, 会把值返回给这个future对象里面. 我们可以使用ExecutorService接口来提交一个线程.

public class Threads {

```
**static** ExecutorService *executorService* = Executors .
*newFixedThreadPool* ( 1 ) ;
```

```
@SuppressWarnings ( “rawtypes” )

**public** **static** **void** main (String[] args ) **throws**
InterruptedException , ExecutionException {

SubThread thread = **new** SubThread () ;
```

// thread.start();

```
Future future = *executorService* . submit (thread) ;

*mainThreadOtherWork* () ;

System . *out* . println ( “now waiting sub thread done.” ) ;

future . get () ;
```

// try {

```
//     thread.join();
//   } catch (InterruptedException e) {
//     e.printStackTrace();
//   }
  System . *out* . println ( “now all done.” ) ;
   *executorService* . shutdown () ;
 }
**private**  **static**  **void**  mainThreadOtherWork () {
 System . *out* . println ( “main thread work start” ) ;
  **try**  {
   Thread . *sleep* ( 3000L ) ;
 }  **catch**  (InterruptedException e) {
   e . printStackTrace () ;
 }
 System . *out* . println ( “main thread work done.” ) ;
}
**public**  **static**  **class**  SubThread  **extends**  Thread{
  @Override
  **public**  **void**  run () {
    working () ;
 }
  **private**  **void**  working () {
   System . *out* . println ( “sub thread start working.” ) ;
    busy () ;
   System . *out* . println ( “sub thread stop working.” ) ;
 }
  **private**  **void**  busy () {
    **try**  {
       *sleep* ( 5000L ) ;
```

```
    }  **catch**  (InterruptedException e) {

       e . printStackTrace () ;

    }

  }
```

```
 }
```

}

这 里, ThreadPoolExecutor 是实现了 ExecutorService的方法, sumbit的过程就是把一个Runnable接口对象包装成一个 Callable接口对象, 然后放到 workQueue里等待调度执行. 当然, 执行的启动也是调用了thread的start来做到的, 只不过这里被包装掉了. 另外, 这里的thread是会被重复利用的, 所以这里要退出主线程, 需要执行以下shutdown方法以示退出使用线程池. 扯远了.

这 种方法是得益于Callable接口和Future模式, 调用future接口的get方法, 会同步等待该future执行结束, 然后获取到结果. Callbale接口的接口方法是 V call(); 是可以有返回结果的, 而Runnable的 void run(), 是没有返回结果的. 所以, 这里即使被包装成Callbale接口, future.get返回的结果也是null的.如果需要得到返回结果, 建议使用Callable接口.

通过队列来控制线程的进度, 是很好的一个理念. 我们完全可以自己搞个队列, 自己控制. 这样也可以实现. 不信看代码:

public class **Threads {**

// static ExecutorService executorService = Executors.newFixedThreadPool(1);

static final BlockingQueue < Integer > *queue* = **new** ArrayBlockingQueue < Integer > (1) ;

public static void main (String[] args) **throws** InterruptedException , ExecutionException {

```
  SubThread thread = **new** SubThread ( *queue* ) ;

  thread . start () ;
```

// Future future = executorService.submit(thread);

```
   *mainThreadOtherWork* () ;

  System . *out* . println ( “now waiting sub thread done.” ) ;
```

// future.get();

```
   *queue* . take () ;
```

// try {

// thread.join();

// } catch (InterruptedException e) {

// e.printStackTrace();

// }

```
System . *out* . println ( “now all done.” ) ;
```

// executorService.shutdown();

}

private static void mainThreadOtherWork () {

```
System . *out* . println ( “main thread work start” ) ;

 **try** {

  Thread . *sleep* ( 3000L ) ;

} **catch** (InterruptedException e) {

  e . printStackTrace () ;

}

System . *out* . println ( “main thread work done.” ) ;
```

}

public static class SubThread **extends** Thread{

```
 **private** BlockingQueue < Integer > queue ;
```

```
 /**

 \* **@param** queue

 */

 **public** SubThread ( BlockingQueue < Integer > queue ) {

   **this** . queue = queue ;

}

 @Override

 **public void** run () {

   **try** {

   working () ;

  } **finally** {
```

```
      **try** {
         queue . put ( 1 ) ;
      } **catch** (InterruptedException e) {
        e . printStackTrace () ;
      }
    }
```

```
  }
   **private void** working () {
    System . *out* . println ( “sub thread start working.” ) ;
     busy () ;
    System . *out* . println ( “sub thread stop working.” ) ;
  }
   **private void** busy () {
     **try** {
       *sleep* ( 5000L ) ;
    } **catch** (InterruptedException e) {
       e . printStackTrace () ;
    }
  }
```

```
 }
}
```

这 里是得益于我们用了一个阻塞队列, 他的put操作和take操作都会阻塞(同步), 在满足条件的情况下.当我们调用take()方法是, 由于子线程还没结束, 队列是空的, 所以这里的take操作会阻塞, 直到子线程结束的时候, 往队列里面put了个元素, 表明自己结束了. 这时候主线程的take()就会返回他拿到的数据. 当然, 他拿到什么我们是不必去关心的.

以上几种情况都是针对子线程只有1个的时候. 当子线程有多个的时候, 情况就不妙了.

第一种方法, 你要调用很多个线程的join, 特别是当你的线程不是for循环创建的, 而是一个一个创建的时候.

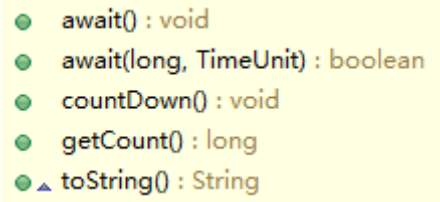
第二种方法, 要调用很多的future的get方法, 同第一种方法.

第三种方法, 比较方便一些, 只需要每个线程都在queue里面 put一个元素就好了.但是, 第三种方法, 这个队列里的对象, 对我们是毫无用处, 我们为了使用队列, 而要不明不白浪费一些内存, 那有没有更好的办法呢?

有的, concurrency包里面提供了好多有用的东东, 其中, CountDownLanch就是我们要用的.

CountDownLanch 是一个倒数计数器, 给一个初始值(>=0), 然后没countDown一次就会减1, 这很符合等待多个子线程结束的产景: 一个线程结束的时候, countDown一次, 直到所有都countDown了 , 那么所有子线程就都结束了.

先看看CountDownLanch有哪些方法:



await: 会阻塞等待计数器减少到0位置. 带参数的await是多了等待时间.

countDown: 将当前的技术减1

getCount(): 返回当前的计数

显而易见, 我们只需要在子线程执行之前, 赋予初始化countDownLanch, 并赋予线程数量为初始值.

每个线程执行完毕的时候, 就countDown一下.主线程只需要调用await方法, 可以等待所有子线程执行结束, 看代码:

public class Threads {

// static ExecutorService executorService = Executors.newFixedThreadPool(1);

static final BlockingQueue < Integer > *queue* = **new** ArrayBlockingQueue < Integer > (1) ;

public static void main (String[] args) **throws** InterruptedException , ExecutionException {

```
 **int** threads = 5 ;

CountDownLatch countDownLatch = **new** CountDownLatch (threads) ;

 **for** ( **int** i = 0 ; i < threads ; i ++ ){

  SubThread thread = **new** SubThread ( 2000 * (i + 1 ) , countDownLatch) ;

  thread . start () ;

}
```

// Future future = executorService.submit(thread);

```
 *mainThreadOtherWork* () ;

System . *out* . println ( “now waiting sub thread done.” ) ;
```

// future.get();

// queue.take();

```
countDownLatch . await () ;
```

// try {

// thread.join();

// } catch (InterruptedException e) {

// e.printStackTrace();

// }

```
System . *out* . println ( “now all done.” ) ;
```

// executorService.shutdown();

}

private static void mainThreadOtherWork () {

```
System . *out* . println ( “main thread work start” ) ;

 **try** {

  Thread . *sleep* ( 3000L ) ;

} **catch** (InterruptedException e) {

  e . printStackTrace () ;

}

System . *out* . println ( “main thread work done.” ) ;
```

}

public static class SubThread **extends** Thread{

// private BlockingQueue queue;

```
 **private** CountDownLatch countDownLatch ;

 **private** **long** work ;
```

```
 /**

 \* **@param** queue

 */
```

// public SubThread(BlockingQueue queue) {

// this.queue = queue;

```
//     this.work = 5000L;
//   }

   **public** SubThread ( **long** work , CountDownLatch countDownLatch ) {
//     this.queue = queue;
     **this** . countDownLatch = countDownLatch ;
     **this** . work = work ;
  }
   @Override
   **public** **void** run () {
     **try** {
     working () ;
    } **finally** {
//       try {
//         queue.put(1);
//       } catch (InterruptedException e) {
//         e.printStackTrace();
//       }
        countDownLatch . countDown () ;
    }

  }
   **private** **void** working () {
    System . *out* . println ( getName () + ” sub thread start working.” ) ;
     busy () ;
    System . *out* . println ( getName () + ” sub thread stop working.” ) ;
  }
   **private** **void** busy () {
     **try** {
```

```
      *sleep* ( work ) ;
    } **catch** (InterruptedException e) {
      e . printStackTrace () ;
    }
  }
```

```
  }
}
```

此种方法也适用于使用 ExecutorService summit 的任务的执行.

另外还有一个并发包的类CyclicBarrier, 这个是(子)线程之间的互相等待的利器. 栅栏, 就是把大家都在一个地方堵住, 就像水闸, 等大家都完成了之前的操作, 在一起继续下面的操作. 不过就不再本篇的讨论访问内了.

进程和线程的区别

- 进程：

是具有一定独立功能的程序、它是系统进行资源分配和调度的一个独立单位，重点在系统调度和单独的单位，也就是说进程是可以独立运行的一段程序(比如正在运行的某个java程序)。

- 线程：

他是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源(一个线程只能属于一个进程，而一个进程可以有多个线程)。

什么叫线程安全？举例说明

- java中的线程安全是什么：

就是线程同步的意思，就是当一个程序对一个线程安全的方法或者语句进行访问的时候，其他的不能再对他进行操作了，必须等到这次访问结束以后才能对这个线程安全的方法进行访问

- 什么叫线程安全：

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，
就是线程安全的。

或者说:一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。

线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。

存在竞争的线程不安全，不存在竞争的线程就是安全的

并发、同步的接口或方法

Java并发编程的类、接口和方法

1：线程池

与每次需要时都创建线程相比，线程池可以降低创建线程的开销，这也是因为线程池在线程执行结束后进行的是回收操作，而不是真正的

销毁线程。

2：ReentrantLock

ReentrantLock提供了tryLock方法，tryLock调用的时候，如果锁被其他线程持有，那么tryLock会立即返回，返回结果为false，如果锁没有被

其他线程持有，那么当前调用线程会持有锁，并且tryLock返回的结果是true,

```
lock.lock();
try {
  //do something
} finally {
  lock.unlock();
 }
```

3：volatile

保证了同一个变量在多线程中的可见性，所以它更多是用于修饰作为开关状态的变量，因为volatile保证了只有一份主存中的数据。

4：Atomics

```
public class Count {
  private AtomicInteger counter = new AtomicInteger();
  public int increase() {
    return counter.incrementAndGet();
  }
 public int decrease() {
    return counter.decrementAndGet();
 }
}
```

AtomicInteger内部通过JNI的方式使用了硬件支持的CAS指令。

5：CountDownLatch

它是java.util.concurrent包中的一个类，它主要提供的机制是当多个(具体数量等于初始化CountDown时的count参数的值)线程都到达了预期状态

或完成预期工作时触发事件，其他线程可以等待这个事件来出发自己后续的工作，等待的线程可以是多个，即CountDownLatch是可以唤醒多个等待

的线程的，到达自己预期状态的线程会调用CountDownLatch的countDown方法，而等待的线程会调用CountDownLatch的await方法

6：CyclicBarrier

循环屏障，CyclicBarrier可以协同多个线程，让多个线程在这个屏障前等待，直到所有线程都到达了这个屏障时，再一起继续执行后面的动作。

CyclicBarrier和CountDownLatch都是用于多个线程间的协调的，二者的一个很大的差别是，CountDownLatch是在多个线程都进行了latch.countDown

后才会触发事件，唤醒await在latch上的线程，而执行countDown的线程，执行完countDown后，会继续自己线程的工作；

CyclicBarrier是一个栅栏，用于同步所有调用await方法的线程，并且等所有线程都到了await方法，这些线程才一起返回继续各自的工作，因为使用CyclicBarrier的线程都会阻塞在await方法上，所以在线程池中使用CyclicBarrier时要特别小心，如果线程池的线程 数过少，那么就会发生死锁了，

CyclicBarrier可以循环使用，CountDownLatch不能循环使用。

7：Semaphore

是用于管理信号量的，构造的时候传入可供管理的信号量的数值，信号量对量管理的信号就像令牌，构造时传入个数，总数就是控制并发的数量。

semaphore.acquire();

try {

```
//调用远程通信的方法
```

} finally () {

```
semaphore.release();
```

}

8：Exchanger

Exchanger，从名字上讲就是交换，它用于在两个线程之间进行数据交换，线程会阻塞在Exchanger的exchange方法上，直到另一个线程也到了

同一个Exchanger的exchange方法时，二者进行交换，然后两个线程会继续执行自身相关的代码。

9：Future和FutureTask

Future future = getDataFromRemote2();

//do something

HashMap data = (HashMap)future.get();

private Future getDateFromRemote2() {

return threadPool.submit(new Callable() {

```
public HashMap call() {

    return getDataFromRemote();

}
```

});

}

思路：调用函数后马上返回，然后继续向下执行，急需要数据时再来用，或者说再来等待这个数据，具体实现方式有两种，一个是用Future，另一个

使用回调。

HashMap 是否线程安全，为何不安全。ConcurrentHashMap，线程安全，为何安全。底层实现是怎么样的。

深入浅出ConcurrentHashMap1.8

前言

HashMap是我们平时开发过程中用的比较多的集合，但它是非线程安全的，在涉及到多线程并发的情况，进行get操作有可能会引起死循环，导致CPU利用率接近100%。

```
final HashMap<String, String> map = new HashMap<String, String>(2);
for (int i = 0; i < 10000; i++) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            map.put(UUID.randomUUID().toString(), "");
        }
    }).start();
}
```

解决方案有Hashtable和Collections.synchronizedMap(hashMap)，不过这两个方案基本上是对读写进行加锁操作，一个线程在读写元素，其余线程必须等待，性能可想而知。

所以，Doug Lea给我们带来了并发安全的ConcurrentHashMap，它的实现是依赖于 Java 内存模型，所以我们在了解 ConcurrentHashMap 的之前必须了解一些底层的知识：

1. java内存模型

2. java中的Unsafe
3. java中的CAS
4. 深入浅出java同步器
5. 深入浅出ReentrantLock

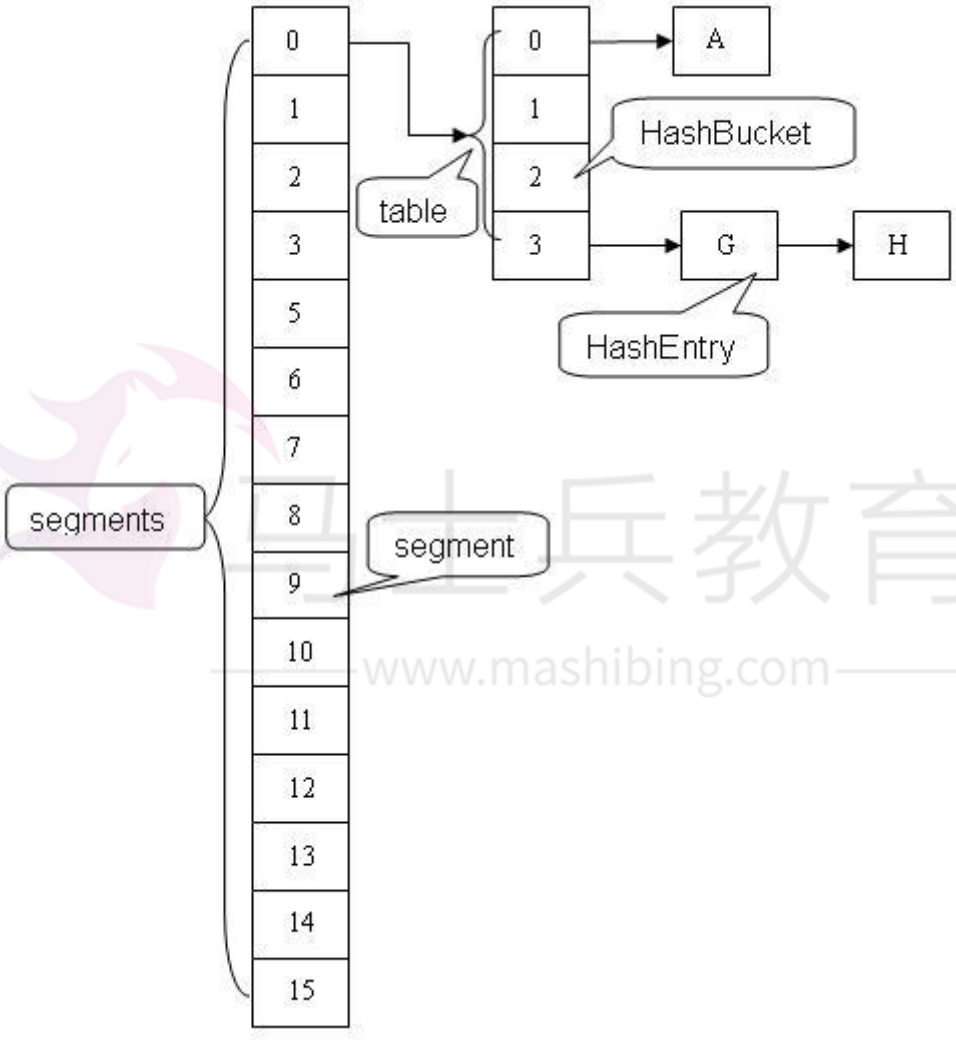
本文源码是JDK8的版本，与之前的版本有较大差异。

JDK1.7分析

ConcurrentHashMap采用 分段锁的机制，实现并发的更新操作，底层采用数组+链表的存储结构。
其包含两个核心静态内部类 Segment和HashEntry。

1. Segment继承ReentrantLock用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶。
2. HashEntry 用来封装映射表的键 / 值对；
3. 每个桶是由若干个 HashEntry 对象链接起来的链表。

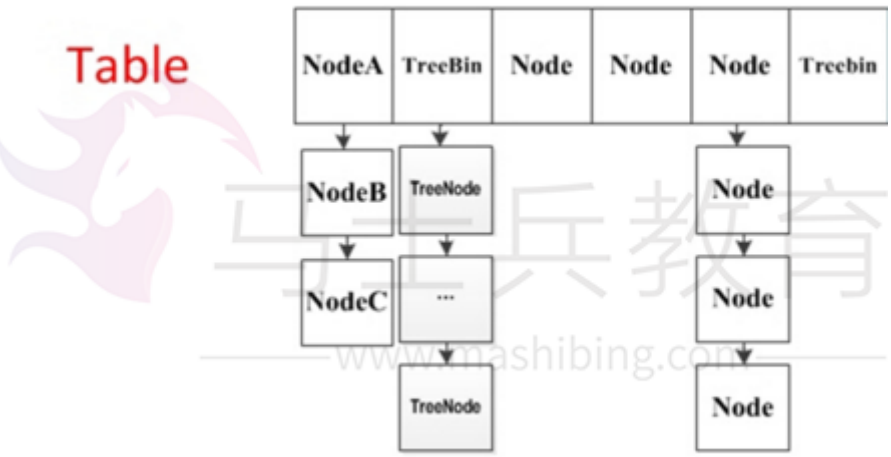
一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组，下面我们通过一个图来演示一下 ConcurrentHashMap 的结构：



马士兵教育
www.mashibing.com

JDK1.8分析

1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层采用数组+链表+红黑树的存储结构。



重要概念

在开始之前，有些重要的概念需要介绍一下：

1. **table**：默认为null，初始化发生在第一次插入操作，默认大小为16的数组，用来存储Node节点数据，扩容时大小总是2的幂次方。
2. **nextTable**：默认为null，扩容时新生成的数组，其大小为原数组的两倍。

- **sizeCtl** ：默认为0，用来控制table的初始化和扩容操作，具体应用在后续会体现出来。
- **-1** 代表table正在初始化
- **-N** 表示有N-1个线程正在进行扩容操作
- 其余情况：
  1、如果table未初始化，表示table需要初始化的大小。
  2、如果table初始化完成，表示table的容量，默认是table大小的0.75倍，居然用这个公式算0.75（n - (n >>> 2)）。
- **Node**：保存key，value及key的hash值的数据结构。

```
class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
    ... 省略部分代码
}
```

其中value和next都用volatile修饰，保证并发的可见性。

- **ForwardingNode**：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用。

```
final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
}
```

只有table发生扩容的时候，ForwardingNode才会发挥作用，作为一个占位符放在table中表示当前节点为null或则已经被移动。

实例初始化

实例化ConcurrentHashMap时带参数时，会根据参数调整table的大小，假设参数为100，最终会调整成256，确保table的大小总是2的幂次方，算法如下：

```
ConcurrentHashMap<String, String> hashMap = new ConcurrentHashMap<>(100);
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

注意，ConcurrentHashMap在构造函数中只会初始化sizeCtl值，并不会直接初始化table，而是延缓到第一次put操作。

table初始化

前面已经提到过，table初始化操作会延缓到第一次put行为。但是put是可以并发执行的，Doug Lea是如何实现table只初始化一次的？让我们来看看源码的实现。

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
//如果一个线程发现sizeCtl<0，意味着另外的线程执行CAS操作成功，当前线程只需要让出cpu时间片
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

sizeCtl默认为0，如果ConcurrentHashMap实例化时有传参数，sizeCtl会是一个2的幂次方的值。所以执行第一次put操作的线程会执行Unsafe.compareAndSwapInt方法修改sizeCtl为-1，有且只有一个线程能够修改成功，其它线程通过Thread.yield()让出CPU时间片等待table初始化完成。

put操作

假设table已经初始化完成，put操作采用CAS+synchronized实现并发插入或更新操作，具体实现如下。

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        ...省略部分代码
    }
    addCount(1L, binCount);
    return null;
}
```

1. hash算法

```
static final int spread(int h) {return (h ^ (h >>> 16)) & HASH_BITS;}
```

1. table中定位索引位置，n是table的大小

```
int index = (n - 1) & hash
```

1. 获取table中对应索引的元素f。
   Doug Lea采用Unsafe.getObjectVolatile来获取，也许有人质疑，直接table[index]不可以么，为什么要这么复杂?
   在java内存模型中，我们已经知道每个线程都有一个工作内存，里面存储着table的副本，虽然table是volatile修饰的，但不能保证线程每次都拿到table中的最新元素，
   Unsafe.getObjectVolatile可以直接获取指定内存的数据，保证了每次拿到数据都是最新的。
2. 如果f为null，说明table中这个位置第一次插入元素，利用Unsafe.compareAndSwapObject方法插入Node节点。

- 如果CAS成功，说明Node节点已经插入，随后addCount(1L, binCount)方法会检查当前容量是否需要进行扩容。
- 如果CAS失败，说明有其它线程提前插入了节点，自旋重新尝试在这个位置插入节点。

1. 如果f的hash值为-1，说明当前f是ForwardingNode节点，意味有其它线程正在扩容，则一起进行扩容操作。
2. 其余情况把新的Node节点按链表或红黑树的方式插入到合适的位置，这个过程采用同步内置锁实现并发，代码如下:

```
synchronized (f) {
    if (tabAt(tab, i) == f) {
        if (fh >= 0) {
            binCount = 1;
            for (Node<K,V> e = f;; ++binCount) {
                K ek;
                if (e.hash == hash &&
                    ((ek = e.key) == key ||
                     (ek != null && key.equals(ek)))) {
                    oldVal = e.val;
                    if (!onlyIfAbsent)
                        e.val = value;
                    break;
                }
                Node<K,V> pred = e;
                if ((e = e.next) == null) {
                    pred.next = new Node<K,V>(hash, key,
                                              value, null);
                    break;
                }
            }
        }
        else if (f instanceof TreeBin) {
            Node<K,V> p;
            binCount = 2;
            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                           value)) != null) {
                oldVal = p.val;
                if (!onlyIfAbsent)
                    p.val = value;
            }
        }
    }
}
```

在节点f上进行同步，节点插入之前，再次利用tabAt(tab, i) == f判断，防止被其它线程修改。

1. 如果f.hash >= 0，说明f是链表结构的头结点，遍历链表，如果找到对应的node节点，则修改value，否则在链表尾部加入节点。
2. 如果f是TreeBin类型节点，说明f是红黑树根节点，则在树结构上遍历元素，更新或增加节点。
3. 如果链表中节点数binCount >= TREEIFY_THRESHOLD(默认是8)，则把链表转化为红黑树结构。

table扩容

当table容量不足的时候，即table的元素数量达到容量阈值sizeCtl，需要对table进行扩容。
整个扩容分为两部分：

1. 构建一个nextTable，大小为table的两倍。
2. 把table的数据复制到nextTable中。

这两个过程在单线程下实现很简单，但是ConcurrentHashMap是支持并发插入的，扩容操作自然也会有并发的出现，这种情况下，第二步可以支持节点的并发复制，这样性能自然提升不少，但实现的复杂度也上升了一个台阶。

先看第一步，构建nextTable，毫无疑问，这个过程只能只有单个线程进行nextTable的初始化，具体实现如下：

```
private final void addCount(long x, int check) {
    ... 省略部分代码
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

通过Unsafe.compareAndSwapInt修改sizeCtl值，保证只有一个线程能够初始化nextTable，扩容后的数组长度为原来的两倍，但是容量是原来的1.5。

节点从table移动到nextTable，大体思想是遍历、复制的过程。

1. 首先根据运算得到需要遍历的次数i，然后利用tabAt方法获得i位置的元素f，初始化一个forwardNode实例fwd。
2. 如果f == null，则在table中的i位置放入fwd，这个过程是采用Unsafe.compareAndSwapObjectf方法实现的，很巧妙的实现了节点的并发移动。
3. 如果f是链表的头节点，就构造一个反序链表，把他们分别放在nextTable的i和i+n的位置上，移动完成，采用Unsafe.putObjectVolatile方法给table原位置赋值fwd。
4. 如果f是TreeBin节点，也做一个反序处理，并判断是否需要untreeify，把处理的结果分别放在nextTable的i和i+n的位置上，移动完成，同样采用Unsafe.putObjectVolatile方法给table原位置赋值fwd。

遍历过所有的节点以后就完成了复制工作，把table指向nextTable，并更新sizeCtl为新数组大小的0.75倍 ，扩容完成。

红黑树构造

注意：如果链表结构中元素超过TREEIFY_THRESHOLD阈值，默认为8个，则把链表转化为红黑树，提高遍历查询效率。

```
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
```

接下来我们看看如何构造树结构，代码如下：

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

可以看出，生成树节点的代码块是同步的，进入同步代码块之后，再次验证table中index位置元素是否被修改过。
1、根据table中index位置Node链表，重新生成一个hd为头结点的TreeNode链表。
2、根据hd头结点，生成TreeBin树结构，并把树结构的root节点写到table的index位置的内存中，具体实现如下：

```
TreeBin(TreeNode<K,V> b) {
    super(TREEBIN, null, null, null);
    this.first = b;
    TreeNode<K,V> r = null;
    for (TreeNode<K,V> x = b, next; x != null; x = next) {
        next = (TreeNode<K,V>)x.next;
        x.left = x.right = null;
        if (r == null) {
            x.parent = null;
            x.red = false;
```

```
            r = x;
        }
        else {
            K k = x.key;
            int h = x.hash;
            Class<?> kc = null;
            for (TreeNode<K,V> p = r;;) {
                int dir, ph;
                K pk = p.key;
                if ((ph = p.hash) > h)
                    dir = -1;
                else if (ph < h)
                    dir = 1;
                else if ((kc == null &&
                          (kc = comparableClassFor(k)) == null) ||
                         (dir = compareComparables(kc, k, pk)) == 0)
                    dir = tieBreakOrder(k, pk);
                    TreeNode<K,V> xp = p;
                if ((p = (dir <= 0) ? p.left : p.right) == null) {
                    x.parent = xp;
                    if (dir <= 0)
                        xp.left = x;
                    else
                        xp.right = x;
                    r = balanceInsertion(r, x);
                    break;
                }
            }
        }
    }
    this.root = r;
    assert checkInvariants(root);
}
```

主要根据Node节点的hash值大小构建二叉树。这个红黑树的构造过程实在有点复杂，感兴趣的同学可以看看源码。

get操作

get操作和put操作相比，显得简单了许多。

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
```

```
            }
        }
        return null;
    }
```

1. 判断table是否为空，如果为空，直接返回null。
2. 计算key的hash值，并获取指定table中指定位置的Node节点，通过遍历链表或则树结构找到对应的节点，返回value值。

总结

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和同步包装器包装的 HashMap，使用一个全局的锁来同步不同线程间的并发访问，同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器，这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。
1.6中采用ReentrantLock 分段锁的方式，使多个线程在不同的segment上进行写操作不会发现阻塞行为;1.8中直接采用了内置锁synchronized，难道是因为1.8的虚拟机对内置锁已经优化的足够快了?

谈谈ConcurrentHashMap1.7和1.8的不同实现

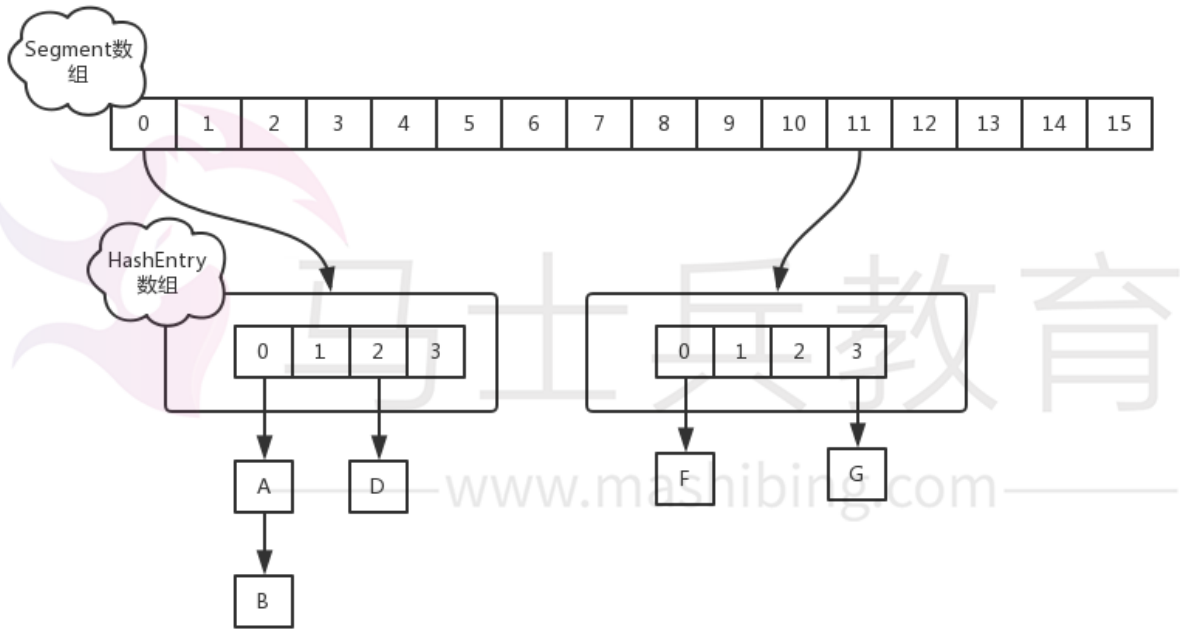
ConcurrentHashMap

在多线程环境下，使用`HashMap`进行`put`操作时存在丢失数据的情况，为了避免这种bug的隐患，强烈建议使用`ConcurrentHashMap`代替`HashMap`，为了对更深入的了解，本文将对JDK1.7和1.8的不同实现进行分析。

JDK1.7

数据结构

jdk1.7中采用`Segment` + `HashEntry`的方式进行实现，结构如下：



`ConcurrentHashMap`初始化时，计算出`Segment`数组的大小`ssize`和每个`Segment`中`HashEntry`数组的大小`cap`，并初始化`Segment`数组的第一个元素；其中`ssize`大小为2的幂次方，默认为16，`cap`大小也是2的幂次方，最小值为2，最终结果根据根据初始化容量`initialCapacity`进行计算，计算过程如下：

```
if (c * ssize < initialCapacity)
    ++c;
int cap = MIN_SEGMENT_TABLE_CAPACITY;
while (cap < c)
    cap <<= 1;
```

其中`Segment`在实现上继承了`ReentrantLock`，这样就自带了锁的功能。

put实现

当执行`put`方法插入数据时，根据key的hash值，在`Segment`数组中找到相应的位置，如果相应位置的`Segment`还未初始化，则通过CAS进行赋值，接着执行`Segment`对象的`put`方法通过加锁机制插入数据，实现如下：

场景：线程A和线程B同时执行相同`Segment`对象的`put`方法

1、线程A执行`tryLock()`方法成功获取锁，则把`HashEntry`对象插入到相应的位置；
2、线程B获取锁失败，则执行`scanAndLockForPut()`方法，在`scanAndLockForPut`方法中，会通过重复执行`tryLock()`方法尝试获取锁，在多处理器环境下，重复次数为64，单处理器重复次数为1，当执行`tryLock()`方法的次数超过上限时，则执行`lock()`方法挂起线程B；
3、当线程A执行完插入操作时，会通过`unlock()`方法释放锁，接着唤醒线程B继续执行；

size实现

因为`ConcurrentHashMap`是可以并发插入数据的，所以在准确计算元素时存在一定的难度，一般的思路是统计每个`Segment`对象中的元素个数，然后进行累加，但是这种方式计算出来的结果并不一样的准确的，因为在计算后面几个`Segment`的元素个数时，已经计算过的`Segment`同时可能有数据的插入或则删除，在1.7的实现中，采用了如下方式：

```
try {
    for (;;) {
        if (retries++ == RETRIES_BEFORE_LOCK) {
            for (int j = 0; j < segments.length; ++j)
                ensureSegment(j).lock(); // force creation
        }
        sum = 0L;
        size = 0;
        overflow = false;
        for (int j = 0; j < segments.length; ++j) {
            Segment<K,V> seg = segmentAt(segments, j);
            if (seg != null) {
                sum += seg.modCount;
                int c = seg.count;
                if (c < 0 || (size += c) < 0)
                    overflow = true;
            }
        }
```

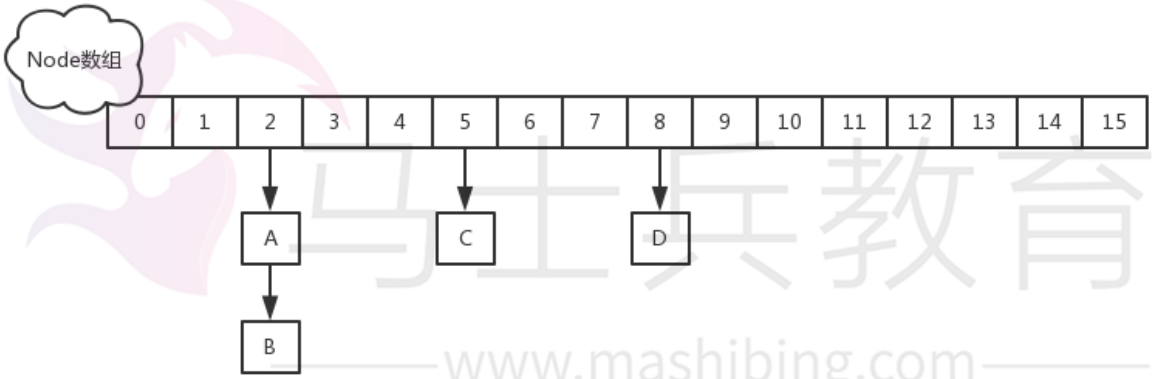
```
            if (sum == last)
                break;
            last = sum;
        }
    } finally {
        if (retries > RETRIES_BEFORE_LOCK) {
            for (int j = 0; j < segments.length; ++j)
                segmentAt(segments, j).unlock();
        }
    }
```

先采用不加锁的方式，连续计算元素的个数，最多计算3次：
1、如果前后两次计算结果相同，则说明计算出来的元素个数是准确的；
2、如果前后两次计算结果都不同，则给每个`Segment`进行加锁，再计算一次元素的个数；

JDK1.8

数据结构

1.8中放弃了`Segment`臃肿的设计，取而代之的是采用`Node` + `CAS` + `Synchronized`来保证并发安全进行实现，结构如下：



只有在执行第一次`put`方法时才会调用`initTable()`初始化`Node`数组，实现如下：

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
```

```
    return tab;
}
```

put实现

当执行`put`方法插入数据时，根据key的hash值，在`Node`数组中找到相应的位置，实现如下：

1、如果相应位置的`Node`还未初始化，则通过CAS插入相应的数据；

```
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
        break;                   // no lock when adding to empty bin
}
```

2、如果相应位置的`Node`不为空，且当前该节点不处于移动状态，则对该节点加`synchronized`锁，如果该节点的`hash`不小于0，则遍历链表更新节点或插入新节点；

```
if (fh >= 0) {
    binCount = 1;
    for (Node<K,V> e = f;; ++binCount) {
        K ek;
        if (e.hash == hash &&
            ((ek = e.key) == key ||
             (ek != null && key.equals(ek)))) {
            oldVal = e.val;
            if (!onlyIfAbsent)
                e.val = value;
            break;
        }
        Node<K,V> pred = e;
        if ((e = e.next) == null) {
            pred.next = new Node<K,V>(hash, key, value, null);
            break;
        }
    }
}
```

3、如果该节点是`TreeBin`类型的节点，说明是红黑树结构，则通过`putTreeVal`方法往红黑树中插入节点；

```
else if (f instanceof TreeBin) {
    Node<K,V> p;
    binCount = 2;
    if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key, value)) != null) {
        oldVal = p.val;
        if (!onlyIfAbsent)
            p.val = value;
    }
}
```

4、如果`binCount`不为0，说明`put`操作对数据产生了影响，如果当前链表的个数达到8个，则通过`treeifyBin`方法转化为红黑树，如果`oldVal`不为空，说明是一次更新操作，没有对元素个数产生影响，则直接返回旧值；

```
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
```

5、如果插入的是一个新节点，则执行`addCount()`方法尝试更新元素个数`baseCount`；

size实现

1.8中使用一个`volatile`类型的变量`baseCount`记录元素的个数，当插入新数据或则删除数据时，会通过`addCount()`方法更新`baseCount`，实现如下：

```
if ((as = counterCells) != null ||
    !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
    CounterCell a; long v; int m;
    boolean uncontended = true;
    if (as == null || (m = as.length - 1) < 0 ||
        (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
        !(uncontended =
          U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
        fullAddCount(x, uncontended);
        return;
    }
    if (check <= 1)
        return;
    s = sumCount();
}
```

1、初始化时`counterCells`为空，在并发量很高时，如果存在两个线程同时执行`CAS`修改`baseCount`值，则失败的线程会继续执行方法体中的逻辑，使用`CounterCell`记录元素个数的变化；

2、如果`CounterCell`数组`counterCells`为空，调用`fullAddCount()`方法进行初始化，并插入对应的记录数，通过`CAS`设置cellsBusy字段，只有设置成功的线程才能初始化`CounterCell`数组，实现如下：

```
else if (cellsBusy == 0 && counterCells == as &&
         U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
    boolean init = false;
    try {                           // Initialize table
        if (counterCells == as) {
            CounterCell[] rs = new CounterCell[2];
            rs[h & 1] = new CounterCell(x);
            counterCells = rs;
```

```
                init = true;
            }
        } finally {
            cellsBusy = 0;
        }
        if (init)
            break;
    }
```

3、如果通过 CAS 设置cellsBusy字段失败的话，则继续尝试通过 CAS 修改 baseCount 字段，如果修改 baseCount 字段成功的话，就退出循环，否则继续循环插入 CounterCell 对象；

```
else if (U.compareAndSwapLong(this, BASECOUNT, v = baseCount, v + x))
    break;
```

所以在1.8中的 size 实现比1.7简单多，因为元素个数保存 baseCount 中，部分元素的变化个数保存在 CounterCell 数组中，实现如下：

```
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}

final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

通过累加 baseCount 和 CounterCell 数组中的数量，即可得到元素的总个数；

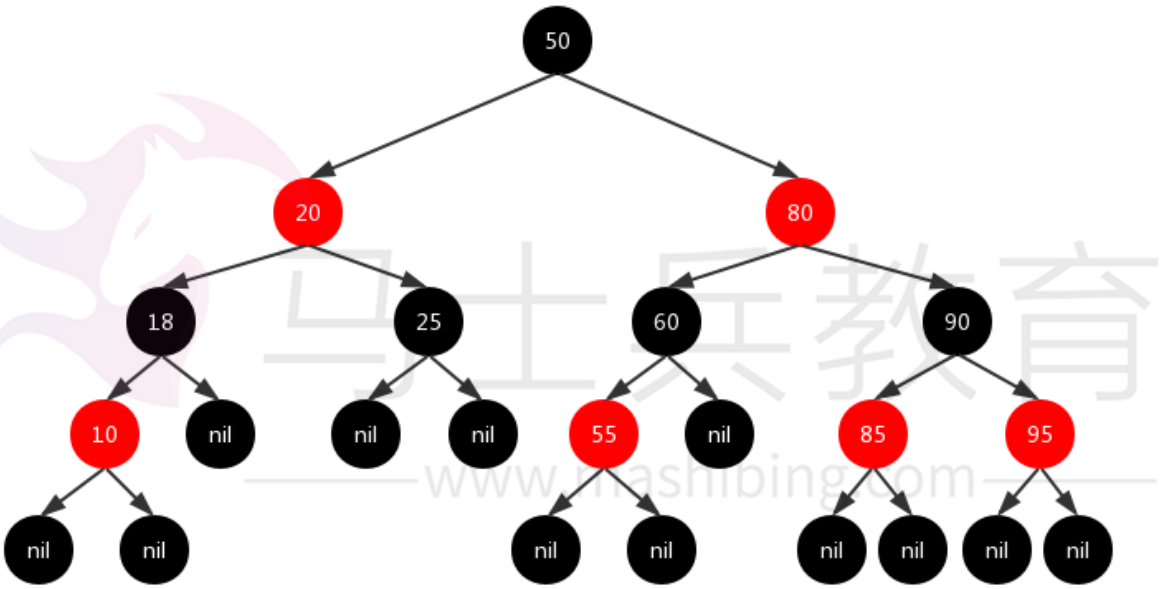
ConcurrentHashMap的红黑树实现分析

红黑树

红黑树是一种特殊的二叉树，主要用它存储有序的数据，提供高效的数据检索，时间复杂度为O(lgn)，每个节点都有一个标识位表示颜色，红色或黑色，有如下5种特性：

1、每个节点要么红色，要么是黑色；
2、根节点一定是黑色的；
3、每个空叶子节点必须是黑色的；
4、如果一个节点是红色的，那么它的子节点必须是黑色的；
5、从一个节点到该节点的子孙节点的所有路径包含相同个数的黑色节点；

结构示意图



只要满足以上5个特性的二叉树都是红黑树，当有新的节点加入时，有可能会破坏其中一些特性，需要通过左旋或右旋操作调整树结构，重新着色，使之重新满足所有特性。

ConcurrentHashMap红黑树实现

在1.8的实现中，当一个链表中的元素达到8个时，会调用`treeifyBin()`方法把链表结构转化成红黑树结构，实现如下：

```
/**
 * Replaces all linked nodes in bin at given index unless table is
 * too small, in which case resizes instead.
 */
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
```

```
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

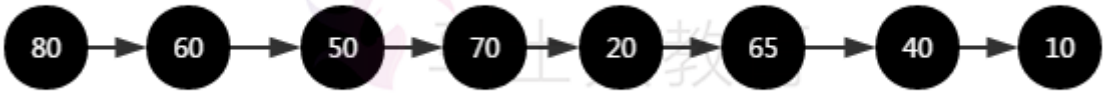
从上述实现可以看出：并非一开始就创建红黑树结构，如果当前`Node`数组长度小于阈值`MIN_TREEIFY_CAPACITY`，默认为64，先通过扩大数组容量为原来的两倍以缓解单个链表元素过大的性能问题。

红黑树构造过程

下面对红黑树的构造过程进行分析：
1、通过遍历`Node`链表，生成对应的`TreeNode`链表，其中`TreeNode`在实现上继承了`Node`类；

```
class TreeNode<K,V> extends Node<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;
    // needed to unlink next upon deletion
    boolean red;
}
```

假设`TreeNode`链表如下，其中节点中的数值代表`hash`值：



2、根据`TreeNode`链表初始化`TreeBin`类对象，`TreeBin`在实现上同样继承了`Node`类，所以初始化完成的`TreeBin`类对象可以保持在`Node`数组中；

```
class TreeBin<K,V> extends Node<K,V> {
    TreeNode<K,V> root;
    volatile TreeNode<K,V> first;
    volatile Thread waiter;
    volatile int lockState;
    // values for lockState
    // set while holding write lock
    static final int WRITER = 1;
    // set when waiting for write lock
    static final int WAITER = 2;
    // increment value for setting read lock
    static final int READER = 4;
}
```

3、遍历`TreeNode`链表生成红黑树，一开始二叉树的根节点`root`为空，则设置链表中的第一个节点80为`root`，并设置其`red`属性为`false`，因为在红黑树的特性1中，明确规定根节点必须是黑色的；

```
for (TreeNode<K,V> x = b, next; x != null; x = next) {
    next = (TreeNode<K,V>)x.next;
    x.left = x.right = null;
    if (r == null) {
        x.parent = null;
        x.red = false;
        r = x;
    }
    ...
```

二叉树结构：



4、加入节点60，如果`root`不为空，则通过比较节点`hash`值的大小将新节点插入到指定位置，实现如下：

```
K k = x.key;
int h = x.hash;
Class<?> kc = null;
for (TreeNode<K,V> p = r;;) {
    int dir, ph;
    K pk = p.key;
    if ((ph = p.hash) > h)
        dir = -1;
    else if (ph < h)
        dir = 1;
    else if ((kc == null &&
              (kc = comparableClassFor(k)) == null) ||
             (dir = compareComparables(kc, k, pk)) == 0)
        dir = tieBreakOrder(k, pk);
```

```
            TreeNode<K,V> xp = p;
        if ((p = (dir <= 0) ? p.left : p.right) == null) {
            x.parent = xp;
            if (dir <= 0)
                xp.left = x;
            else
                xp.right = x;
            r = balanceInsertion(r, x);
            break;
        }
    }
```

其中`x`代表即将插入到红黑树的节点，`p`指向红黑树中当前遍历到的节点，从根节点开始递归遍历，`x`的插入过程如下：

1)、如果`x`的`hash`值小于`p`的`hash`值，则判断`p`的左节点是否为空，如果不为空，则把`p`指向其左节点，并继续和`p`进行比较，如果`p`的左节点为空，则把`x`指向的节点插入到该位置；

2)、如果`x`的`hash`值大于`p`的`hash`值，则判断`p`的右节点是否为空，如果不为空，则把`p`指向其右节点，并继续和`p`进行比较，如果`p`的右节点为空，则把`x`指向的节点插入到该位置；

3)、如果`x`的`hash`值和`p`的`hash`值相等，怎么办？
解决：首先判断节点中的`key`对象的类是否实现了`Comparable`接口，如果实现`Comparable`接口，则调用`compareTo`方法比较两者`key`的大小，但是如果`key`对象没有实现`Comparable`接口，或则`compareTo`方法返回了0，则继续调用`tieBreakOrder`方法计算`dir`值，`tieBreakOrder`方法实现如下：

```
static int tieBreakOrder(Object a, Object b) {
    int d;
    if (a == null || b == null ||
        (d = a.getClass().getName().
         compareTo(b.getClass().getName())) == 0)
        d = (System.identityHashCode(a) <= System.identityHashCode(b) ?
             -1 : 1);
    return d;
}
```

最终比较`key`对象的默认`hashCode()`方法的返回值，因为`System.identityHashCode(a)`调用的是对象`a`默认的`hashCode()`；

插入节点60之后的二叉树：

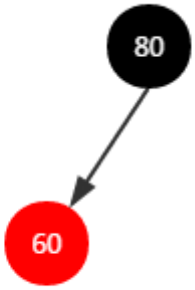


5、当有新节点加入时，可能会破坏红黑树的特性，需要执行`balanceInsertion()`方法调整二叉树，使之重新满足特性，方法中的变量`xp`指向`x`的父节点，`xpp`指向`xp`父节点，`xppl`和`xppr`分别指向`xpp`的左右子节点，`balanceInsertion()`方法首先会把新加入的节点设置成红色。

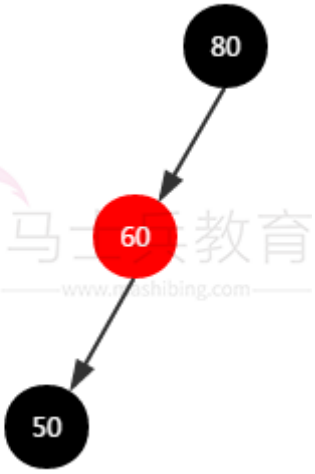
①、加入节点60之后，此时`xp`指向节点80，其父节点为空，直接返回。

```
if ((xp = x.parent) == null) {
    x.red = false;
    return x;
}
else if (!xp.red || (xpp = xp.parent) == null)
    return root;
```

调整之后的二叉树：



②、加入节点50，二叉树如下：



继续执行`balanceInsertion()`方法调整二叉树，此时节点50的父节点60是左儿子，走如下逻辑：

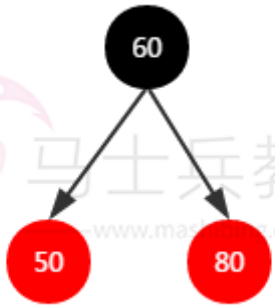
```
if (xp == (xppl = xpp.left)) {
    if ((xppr = xpp.right) != null && xppr.red) {
        xppr.red = false;
        xp.red = false;
        xpp.red = true;
        x = xpp;
    }
```

```
    else {
        if (x == xp.right) {
            root = rotateLeft(root, x = xp);
            xpp = (xp = x.parent) == null ? null : xp.parent;
        }
        if (xp != null) {
            xp.red = false;
            if (xpp != null) {
                xpp.red = true;
                root = rotateRight(root, xpp);
            }
        }
    }
}
```

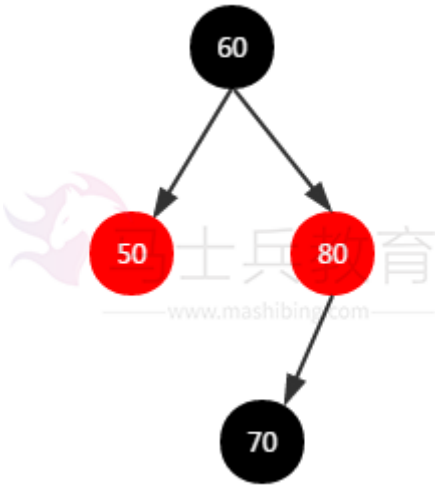
根据上述逻辑，把节点60设置成黑色，把节点80设置成红色，并对节点80执行右旋操作，右旋实现如下：

```
static <K,V> TreeNode<K,V> rotateRight(TreeNode<K,V> root,
                                       TreeNode<K,V> p) {
    TreeNode<K,V> l, pp, lr;
    if (p != null && (l = p.left) != null) {
        if ((lr = p.left = l.right) != null)
            lr.parent = p;
        if ((pp = l.parent = p.parent) == null)
            (root = l).red = false;
        else if (pp.right == p)
            pp.right = l;
        else
            pp.left = l;
        l.right = p;
        p.parent = l;
    }
    return root;
}
```

右旋之后的红黑树如下：



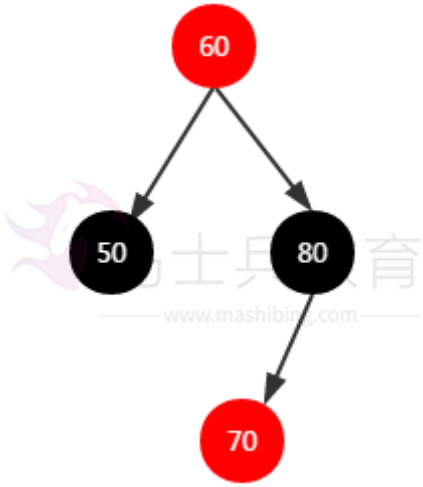
③、加入节点70，二叉树如下：



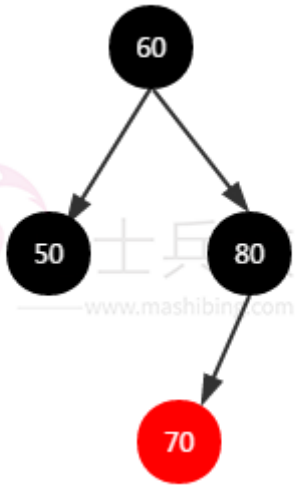
继续执行`balanceInsertion()`方法调整二叉树，此时父节点80是个右儿子，节点70是左儿子，且叔节点50不为空，且是红色的，则执行如下逻辑：

```
if (xppl != null && xppl.red) {
    xppl.red = false;
    xp.red = false;
    xpp.red = true;
    x = xpp;
}
```

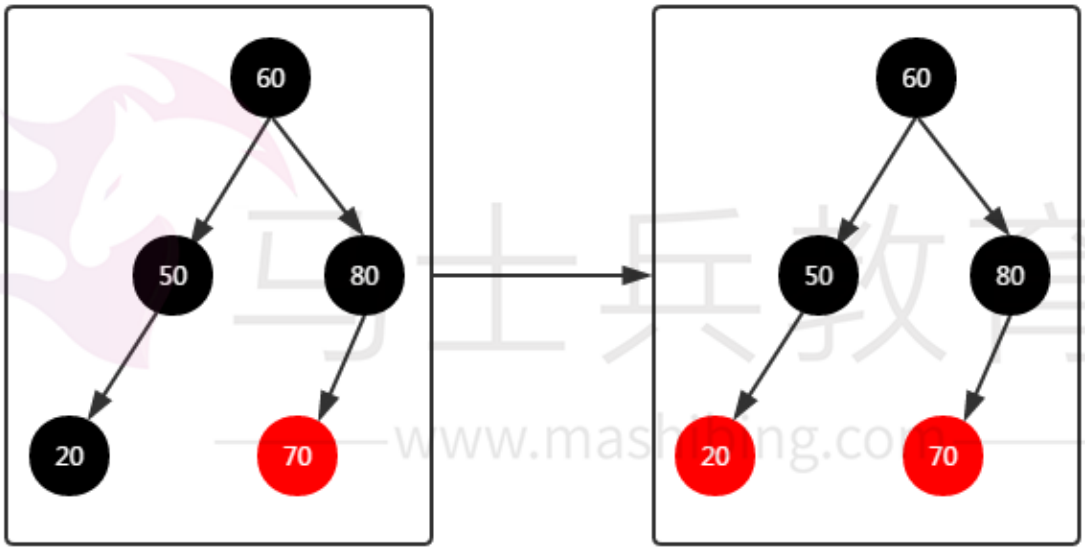
此时二叉树如下：



此时`x`指向`xpp`，即节点60，继续循环处理`x`，设置其颜色为黑色，最终二叉树如下：

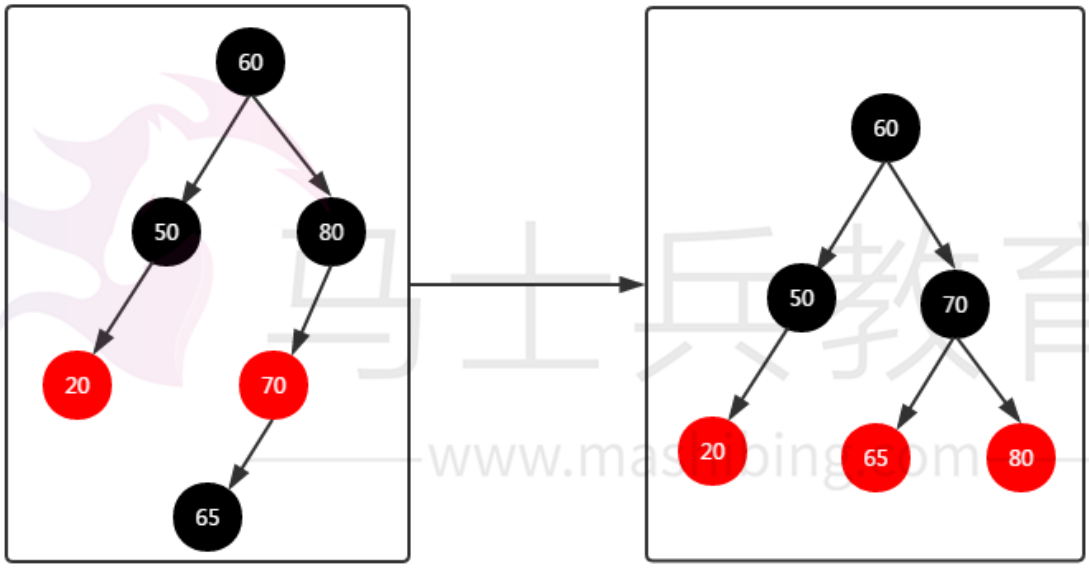


④、加入节点20，二叉树变化如下：



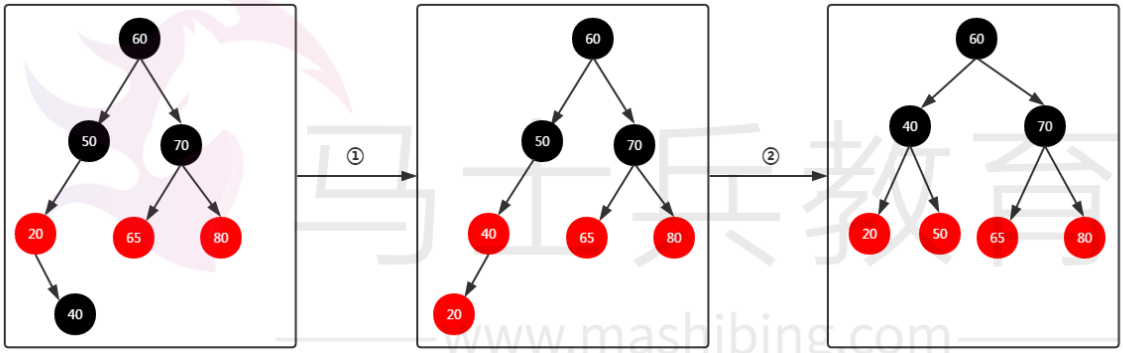
因为节点20的父节点50是一个黑色的节点，不需要进行调整；

⑤、加入节点65，二叉树变化如下：



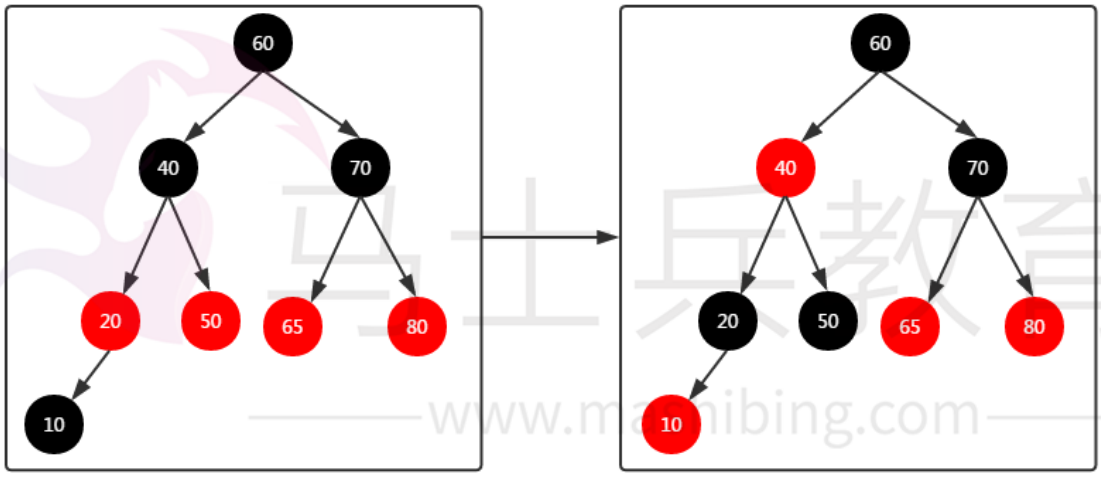
对节点80进行右旋操作。

⑥、加入节点40，二叉树变化如下：



1、对节点20执行左旋操作；
2、对节点50执行右旋操作；

最后加入节点10，二叉树变化如下：



重新对节点进行着色，到此为止，红黑树已经构造完成；

深入分析ConcurrentHashMap1.8的扩容实现

ConcurrentHashMap相关的文章写了不少，有个遗留问题一直没有分析，也被好多人请教过，被搁置在一旁，即如何在并发的情况下实现数组的扩容。

什么情况会触发扩容

当往hashMap中成功插入一个key/value节点时，有可能触发扩容动作：
1、如果新增节点之后，所在链表的元素个数达到了阈值 **8**，则会调用`treeifyBin`方法把链表转换成红黑树，不过在结构转换之前，会对数组长度进行判断，实现如下：

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.val,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

如果数组长度n小于阈值`MIN_TREEIFY_CAPACITY`，默认是64，则会调用`tryPresize`方法把数组长度扩大到原来的两倍，并触发`transfer`方法，重新调整节点的位置。

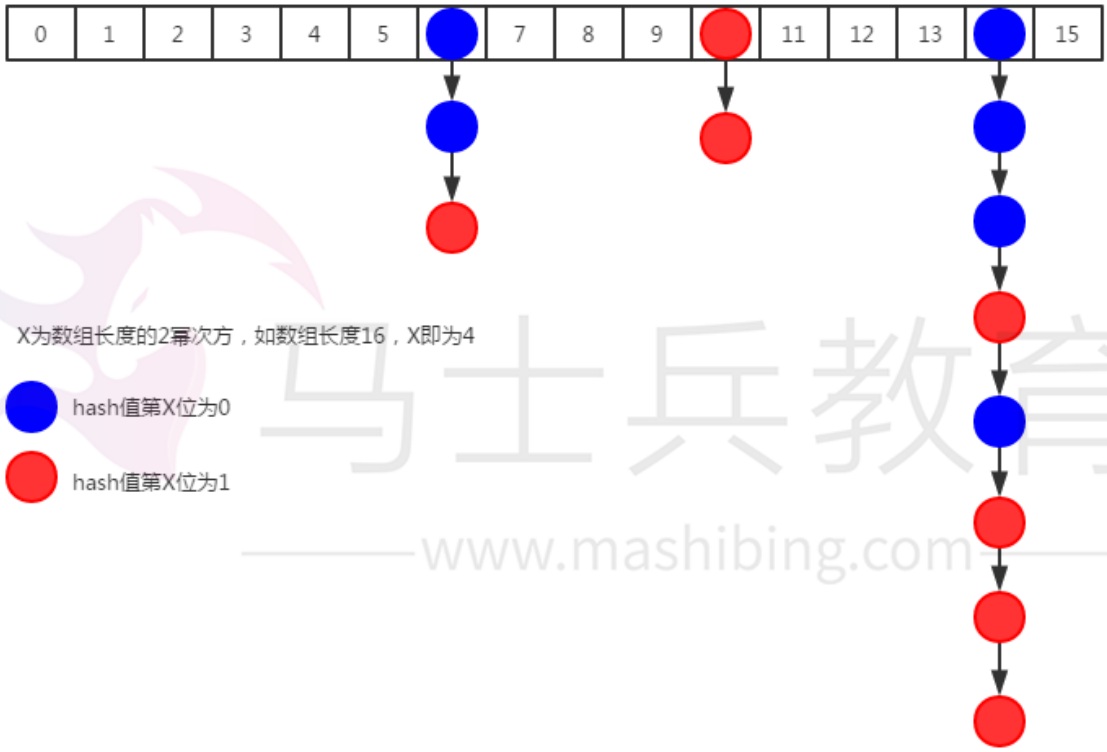
```
else if (tab == table) {
    int rs = resizeStamp(n);
    if (sc < 0) {
        Node<K,V>[] nt;
        if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
            sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
            transferIndex <= 0)
            break;
        if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
            transfer(tab, nt);
    }
    else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                 (rs << RESIZE_STAMP_SHIFT) + 2))
        transfer(tab, null);
}
```

2、新增节点之后，会调用`addCount`方法记录元素个数，并检查是否需要进行扩容，当数组元素个数达到阈值时，会触发`transfer`方法，重新调整节点的位置。

```
if (check >= 0) {
    Node<K,V>[] tab, nt; int n, sc;
    while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
           (n = tab.length) < MAXIMUM_CAPACITY) {
        int rs = resizeStamp(n);
        if (sc < 0) {
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                transferIndex <= 0)
                break;
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                transfer(tab, nt);
        }
        else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                     (rs << RESIZE_STAMP_SHIFT) + 2))
            transfer(tab, null);
        s = sumCount();
    }
}
```

transfer实现

`transfer`方法实现了在并发的情况下，高效的从原始组数往新数组中移动元素，假设扩容之前节点的分布如下，这里区分蓝色节点和红色节点，是为了后续更好的分析：



在上图中，第14个槽位插入新节点之后，链表元素个数已经达到了8，且数组长度为16，优先通过扩容来缓解链表过长的问题，实现如下：

1、根据当前数组长度n，新建一个两倍长度的数组`nextTable`；

```
if (nextTab == null) {            // initiating
    try {
        /unchecked/
        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
        nextTab = nt;
    } catch (Throwable ex) {      // try to cope with OOME
        sizeCtl = Integer.MAX_VALUE;
        return;
    }
    nextTable = nextTab;
    transferIndex = n;
}
```

2、初始化`ForwardingNode`节点，其中保存了新数组`nextTable`的引用，在处理完每个槽位的节点之后当做占位节点，表示该槽位已经处理过了；

```
int nextn = nextTab.length;
ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
boolean advance = true;
boolean finishing = false; // to ensure sweep before committing nextTab
```

3、通过`for`自循环处理每个槽位中的链表元素，默认`advace`为真，通过CAS设置`transferIndex`属性值，并初始化`i`和`bound`值，`i`指当前处理的槽位序号，`bound`指需要处理的槽位边界，先处理槽位15的节点；

```
for (int i = 0, bound = 0;;) {
    Node<K,V> f; int fh;
    while (advance) {
        int nextIndex, nextBound;
        if (--i >= bound || finishing)
            advance = false;
        else if ((nextIndex = transferIndex) <= 0) {
            i = -1;
            advance = false;
        }
        else if (U.compareAndSwapInt
                 (this, TRANSFERINDEX, nextIndex,
                  nextBound = (nextIndex > stride ?
                               nextIndex - stride : 0))) {
            bound = nextBound;
            i = nextIndex - 1;
            advance = false;
        }
    }
```

4、在当前假设条件下，槽位15中没有节点，则通过CAS插入在第二步中初始化的`ForwardingNode`节点，用于告诉其它线程该槽位已经处理过了；

```
else if ((f = tabAt(tab, i)) == null)
    advance = casTabAt(tab, i, null, fwd);
```

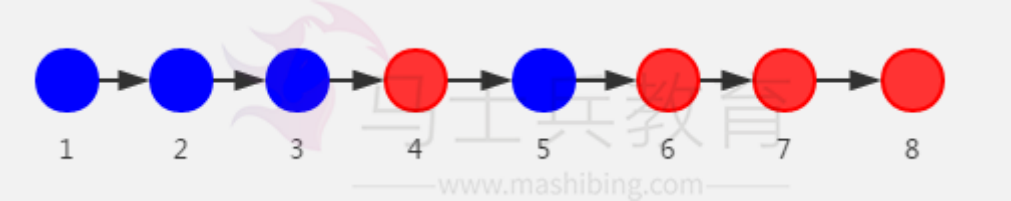
5、如果槽位15已经被线程A处理了，那么线程B处理到这个节点时，取到该节点的hash值应该为`MOVED`，值为`-1`，则直接跳过，继续处理下一个槽位14的节点；

```
else if ((fh = f.hash) == MOVED)
    advance = true; // already processed
```

6、处理槽位14的节点，是一个链表结构，先定义两个变量节点`ln`和`hn`，按我的理解应该是`lowNode`和`highNode`，分别保存hash值的第X位为0和1的节点，具体实现如下：

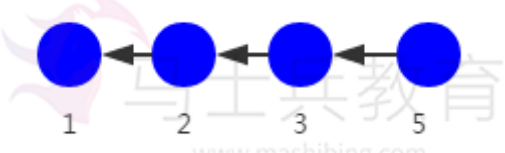
```
synchronized (f) {
    if (tabAt(tab, i) == f) {
        Node<K,V> ln, hn;
        if (fh >= 0) {
            int runBit = fh & n;
            Node<K,V> lastRun = f;
            for (Node<K,V> p = f.next; p != null; p = p.next) {
                int b = p.hash & n;
                if (b != runBit) {
                    runBit = b;
                    lastRun = p;
                }
            }
            if (runBit == 0) {
                ln = lastRun;
                hn = null;
            }
            else {
                hn = lastRun;
                ln = null;
            }
            for (Node<K,V> p = f; p != lastRun; p = p.next) {
                int ph = p.hash; K pk = p.key; V pv = p.val;
                if ((ph & n) == 0)
                    ln = new Node<K,V>(ph, pk, pv, ln);
                else
                    hn = new Node<K,V>(ph, pk, pv, hn);
            }
            setTabAt(nextTab, i, ln);
            setTabAt(nextTab, i + n, hn);
            setTabAt(tab, i, fwd);
            advance = true;
        }
```

使用`fn&n`可以快速把链表中的元素区分成两类，A类是hash值的第X位为0，B类是hash值的第X位为1，并通过`lastRun`记录最后需要处理的节点，A类和B类节点可以分散到新数组的槽位14和30中，在原数组的槽位14中，蓝色节点第X为0，红色节点第X为1，把链表拉平显示如下：

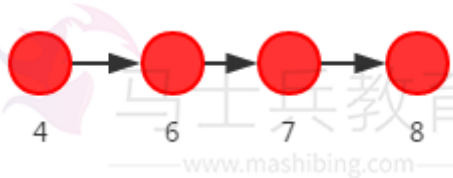


1、通过遍历链表，记录`runBit`和`lastRun`，分别为1和节点6，所以设置`hn`为节点6，`ln`为null；
2、重新遍历链表，以`lastRun`节点为终止条件，根据第X位的值分别构造ln链表和hn链表：

ln链：和原来链表相比，顺序已经不一样了



hn链：



通过CAS把ln链表设置到新数组的i位置，hn链表设置到i+n的位置；

7、如果该槽位是红黑树结构，则构造树节点 `lo` 和 `hi`，遍历红黑树中的节点，同样根据 `hash&n` 算法，把节点分为两类，分别插入到 `lo` 和 `hi` 为头的链表中，根据 `lo` 和 `hi` 链表中的元素个数分别生成 `ln` 和 `hn` 节点，其中 `ln` 节点的生成逻辑如下：

（1）如果 `lo` 链表的元素个数小于等于 `UNTREEIFY_THRESHOLD`，默认为6，则通过 `untreeify` 方法把树节点链表转化成普通节点链表；

（2）否则判断 `hi` 链表中的元素个数是否等于0：如果等于0，表示 `lo` 链表中包含了所有原始节点，则设置原始红黑树给 `ln`，否则根据 `lo` 链表重新构造红黑树。

```
else if (f instanceof TreeBin) {
    TreeBin<K,V> t = (TreeBin<K,V>)f;
    TreeNode<K,V> lo = null, loTail = null;
    TreeNode<K,V> hi = null, hiTail = null;
    int lc = 0, hc = 0;
    for (Node<K,V> e = t.first; e != null; e = e.next) {
        int h = e.hash;
        TreeNode<K,V> p = new TreeNode<K,V>
            (h, e.key, e.val, null, null);
        if ((h & n) == 0) {
            if ((p.prev = loTail) == null)
                lo = p;
            else
                loTail.next = p;
            loTail = p;
            ++lc;
        }
        else {
            if ((p.prev = hiTail) == null)
                hi = p;
            else
                hiTail.next = p;
            hiTail = p;
            ++hc;
        }
    }
    ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
        (hc != 0) ? new TreeBin<K,V>(lo) : t;
    hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
        (lc != 0) ? new TreeBin<K,V>(hi) : t;
    setTabAt(nextTab, i, ln);
    setTabAt(nextTab, i + n, hn);
    setTabAt(tab, i, fwd);
    advance = true;
}
```

最后，同样的通过CAS把 `ln` 设置到新数组的 `i` 位置，`hn` 设置到 `i+n` 位置。

老生常谈，HashMap的死循环

问题

最近的几次面试中，我都问了是否了解HashMap在并发使用时可能发生死循环，导致cpu100%，结果让我很意外，都表示不知道有这样的问题，让我意外的是面试者的工作年限都不短。

由于HashMap并非是线程安全的，所以在高并发的情况下必然会出现问题，这是一个普遍的问题，虽然网上分析的文章很多，还是觉得有必须写一篇文章，让关注我公众号的同学能够意识到这个问题，并了解这个死循环是如何产生的。

如果是在单线程下使用HashMap，自然是没有问题的，如果后期由于代码优化，这段逻辑引入了多线程并发执行，在一个未知的时间点，会发现CPU占用100%，居高不下，通过查看堆栈，你会惊讶的发现，线程都Hang在hashMap的get()方法上，服务重启之后，问题消失，过段时间可能又复现了。

这是为什么?

原因分析

在了解来龙去脉之前，我们先看看HashMap的数据结构。

在内部，HashMap使用一个Entry数组保存key、value数据，当一对key、value被加入时，会通过一个hash算法得到数组的下标index，算法很简单，根据key的hash值，对数组的大小取模 hash & (length-1)，并把结果插入数组该位置，如果该位置上已经有元素了，就说明存在hash冲突，这样会在index位置生成链表。

如果存在hash冲突，最惨的情况，就是所有元素都定位到同一个位置，形成一个长长的链表，这样get一个值时，最坏情况需要遍历所有节点，性能变成了O(n)，所以元素的hash值算法和HashMap的初始化大小很重要。

当插入一个新的节点时，如果不存在相同的key，则会判断当前内部元素是否已经达到阈值（默认是数组大小的0.75），如果已经达到阈值，会对数组进行扩容，也会对链表中的元素进行rehash。

实现

HashMap的put方法实现：

1、判断key是否已经存在

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    // 如果key已经存在，则替换value，并返回旧值
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
```

```
    modCount++;
    // key不存在，则插入新的元素
    addEntry(hash, key, value, i);
    return null;
}
```

2、检查容量是否达到阈值threshold

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length);
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }

    createEntry(hash, key, value, bucketIndex);
}
```

如果元素个数已经达到阈值，则扩容，并把原来的元素移动过去。

3、扩容实现

```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    ...

    Entry[] newTable = new Entry[newCapacity];
    ...
    transfer(newTable, rehash);
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
}
```

这里会新建一个更大的数组，并通过transfer方法，移动元素。

```
void transfer(Entry[] newTable, boolean rehash) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            if (rehash) {
                e.hash = null == e.key ? 0 : hash(e.key);
            }
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```

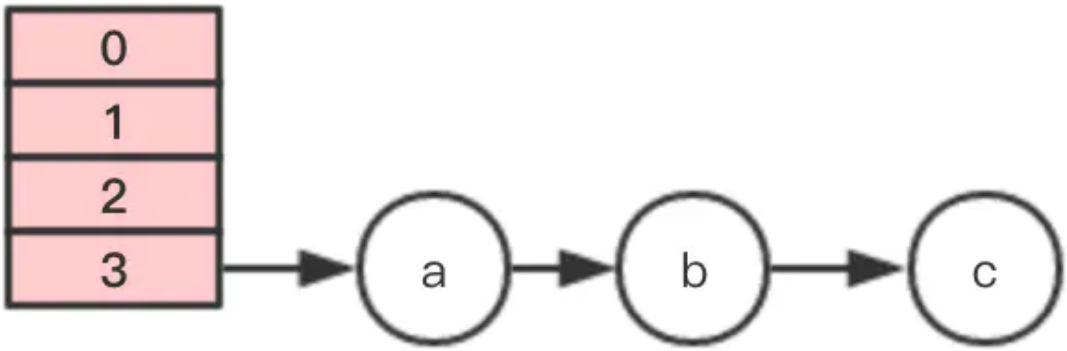
移动的逻辑也很清晰，遍历原来table中每个位置的链表，并对每个元素进行重新hash，在新的newTable找到归宿，并插入。

案例分析

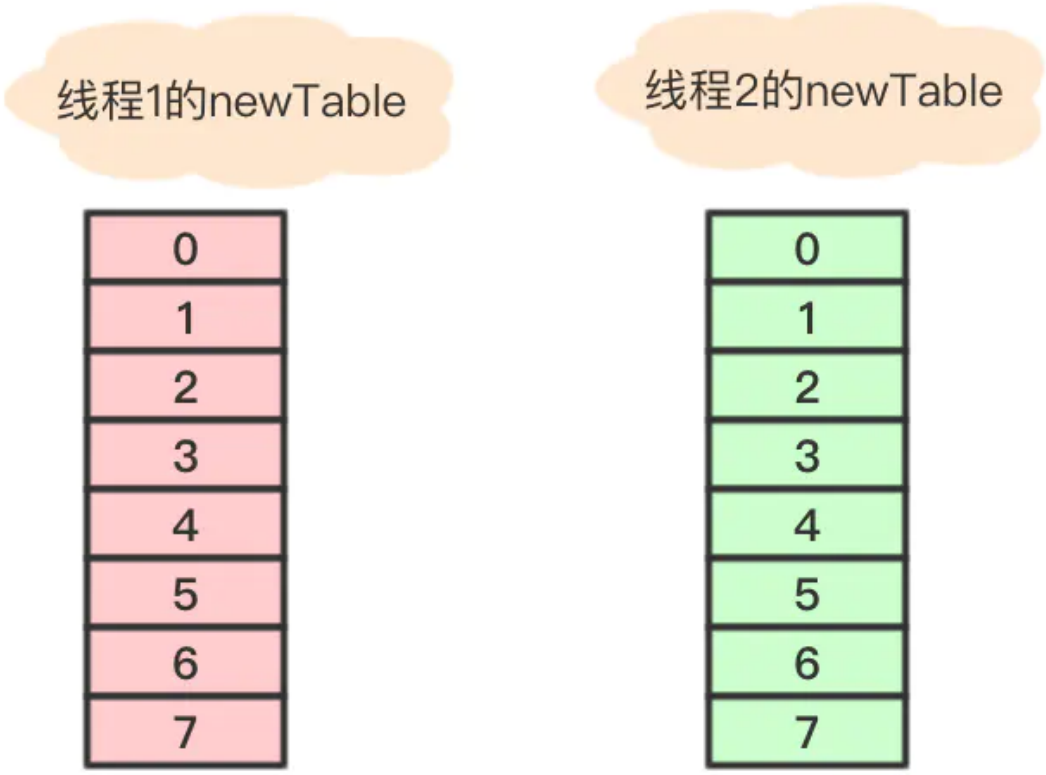
假设HashMap初始化大小为4，插入个3节点，不巧的是，这3个节点都hash到同一个位置，如果按照默认的负载因子的话，插入第3个节点就会扩容，为了验证效果，假设负载因子是1.

```
void transfer(Entry[] newTable, boolean rehash) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            if (rehash) {
                e.hash = null == e.key ? 0 : hash(e.key);
            }
            int i = indexFor(e.hash, newCapacity);
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```

以上是节点移动的相关逻辑。



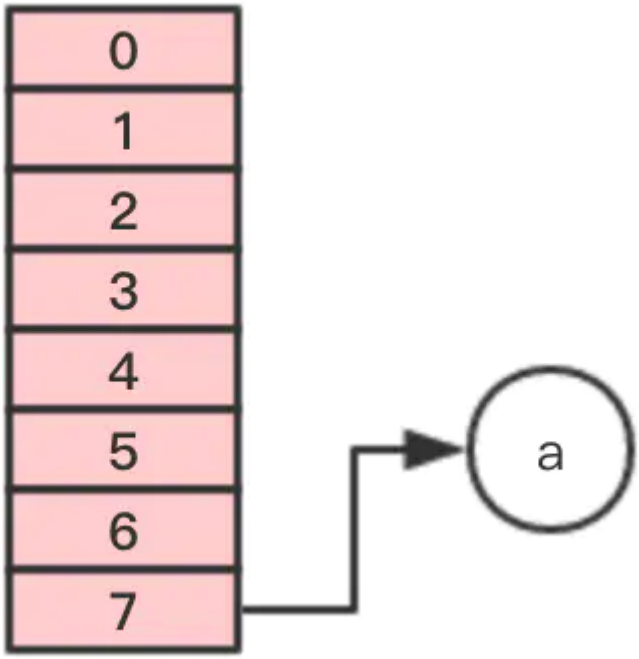
插入第4个节点时，发生rehash，假设现在有两个线程同时进行，线程1和线程2，两个线程都会新建新的数组。



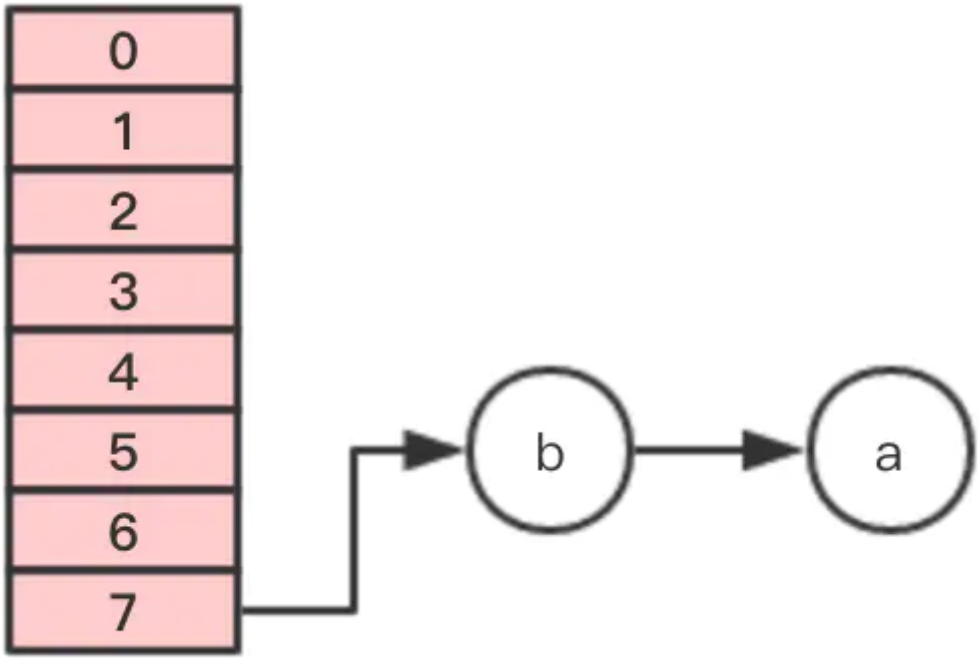
假设 **线程2** 在执行到 `Entry<K,V> next = e.next;` 之后，cpu时间片用完了，这时变量e指向节点a，变量next指向节点b。

线程1继续执行，很不巧，a、b、c节点rehash之后又是在同一个位置7，开始移动节点

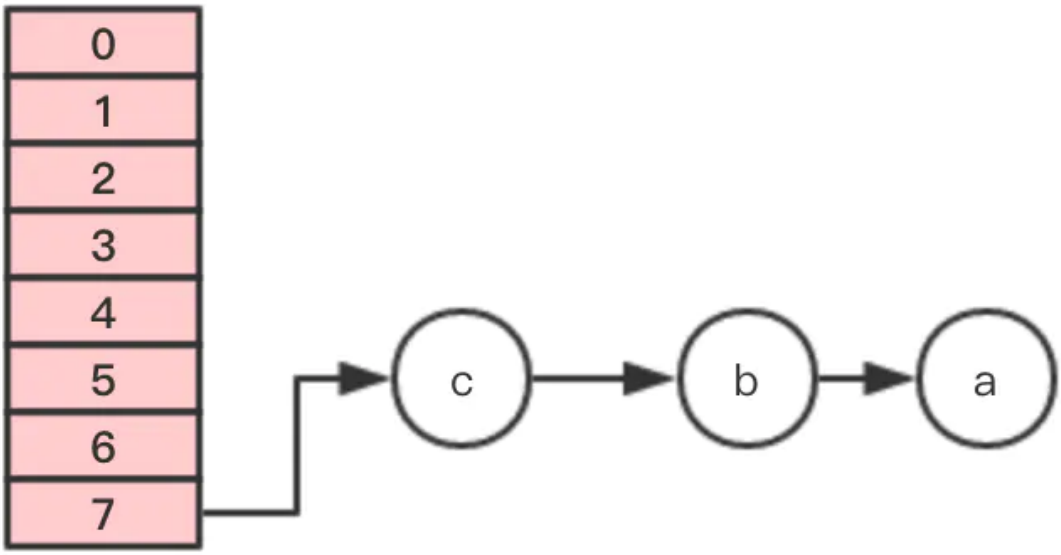
第一步，移动节点a



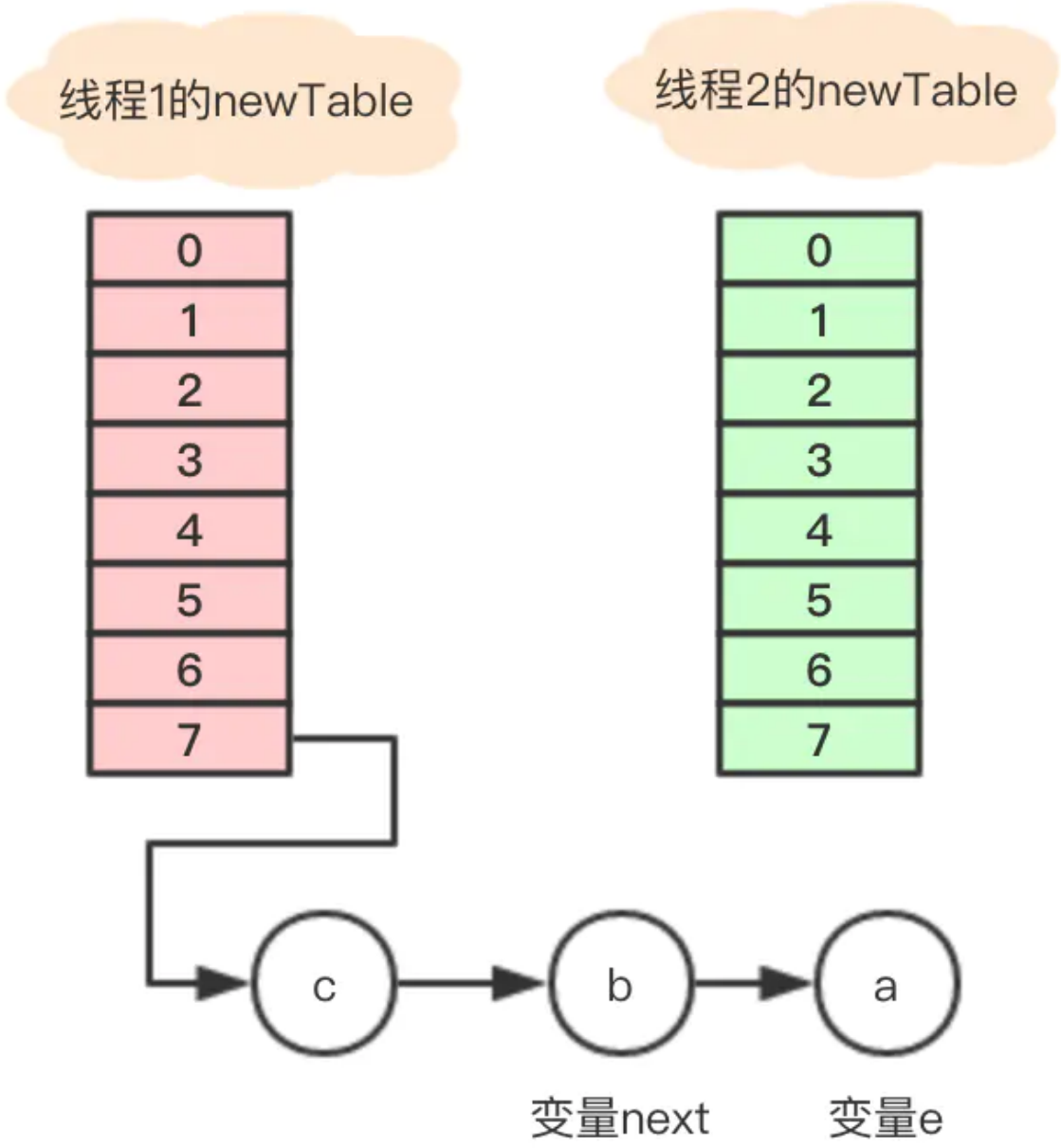
第二步，移动节点b



注意，这里的顺序是反过来的，继续移动节点c



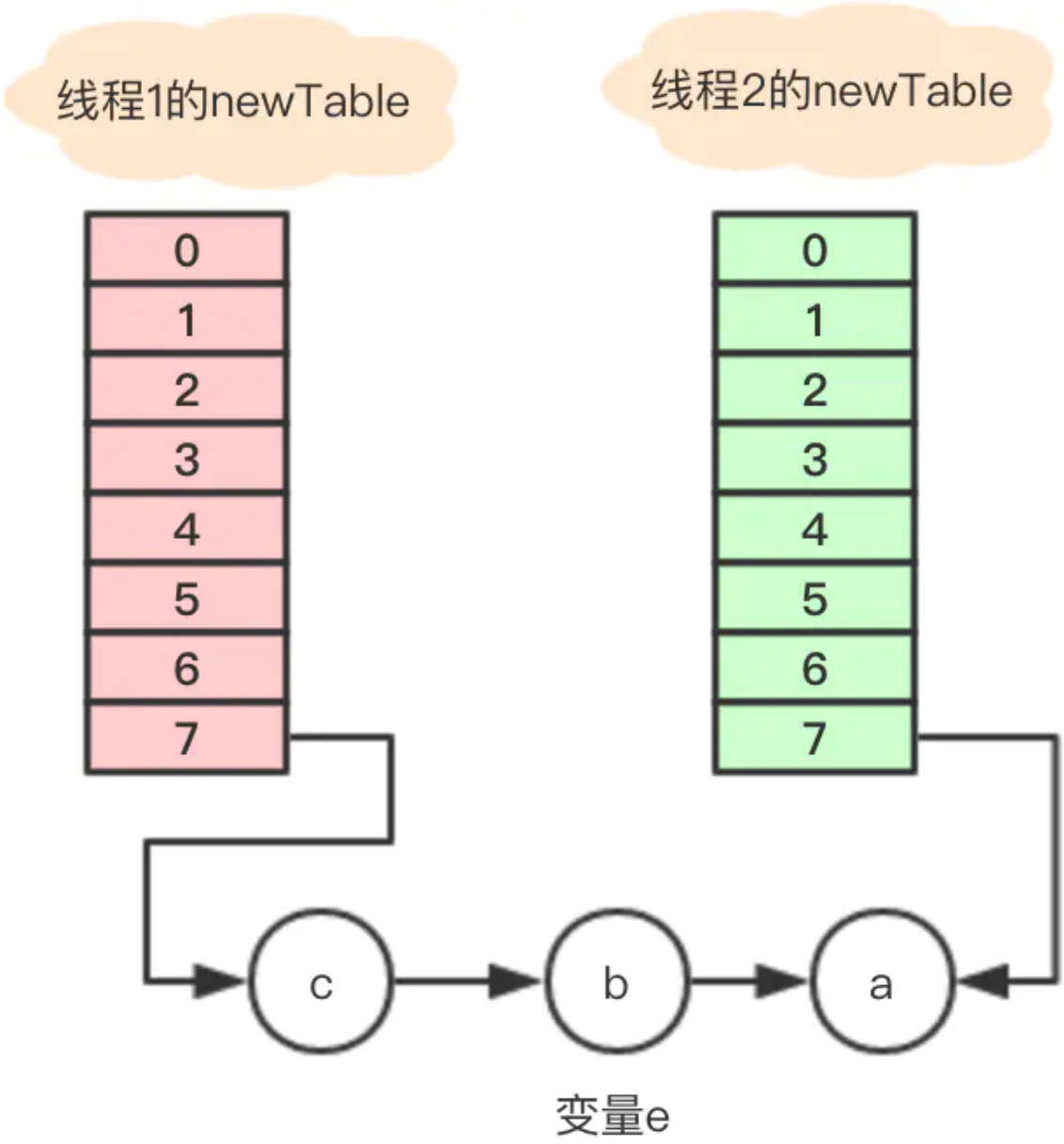
这个时候 **线程1** 的时间片用完，内部的table还没有设置成新的newTable， **线程2** 开始执行，这时内部的引用关系如下：



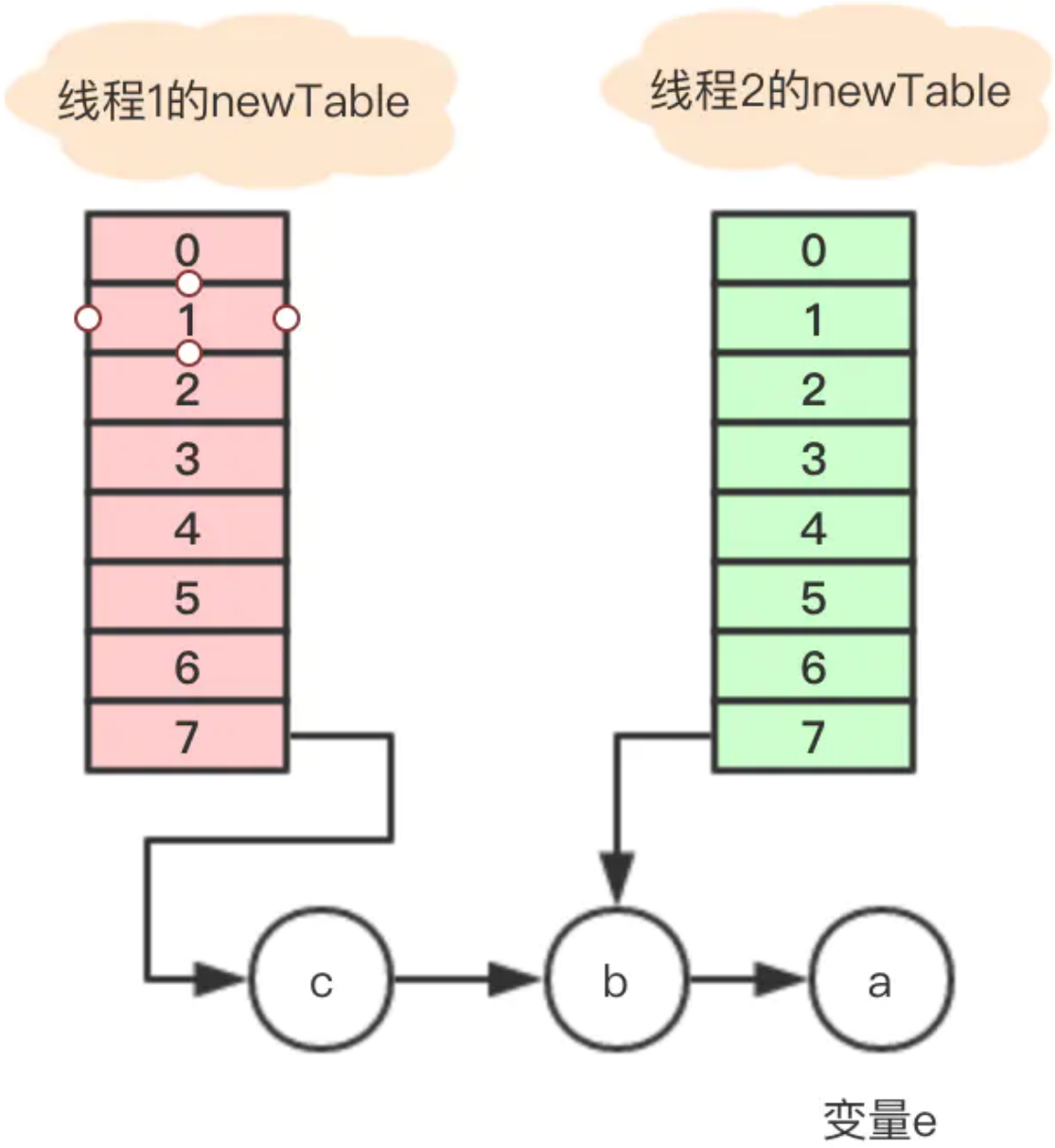
这时，在 **线程2** 中，变量e指向节点a，变量next指向节点b，开始执行循环体的剩余逻辑。

```
Entry<K,V> next = e.next;
int i = indexFor(e.hash, newCapacity);
e.next = newTable[i];
newTable[i] = e;
e = next;
```

执行之后的引用关系如下图



执行后，变量e指向节点b，因为e不是null，则继续执行循环体，执行后的引用关系



变量e又重新指回节点a，只能继续执行循环体，这里仔细分析下：

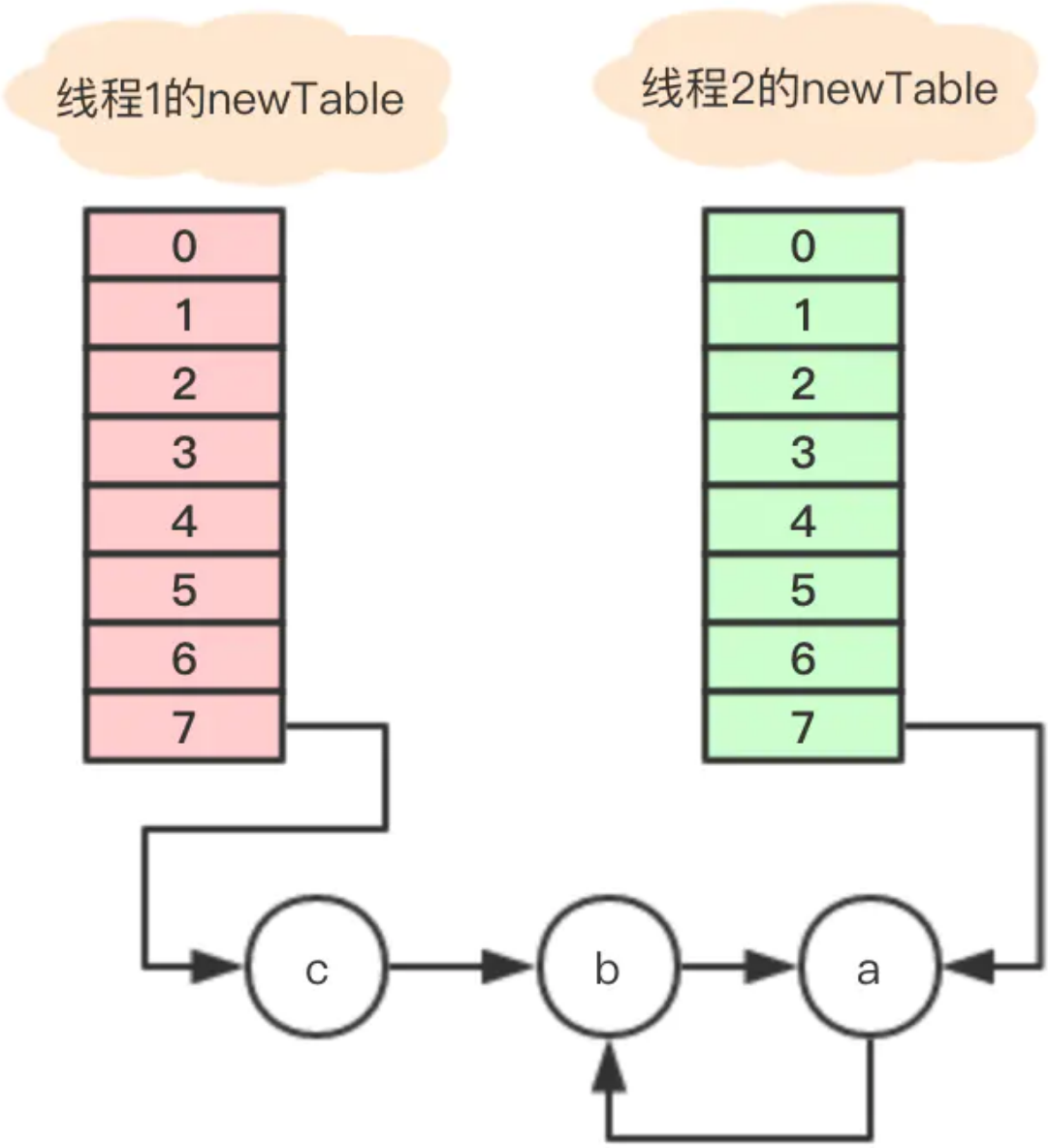
1、执行完 `Entry<K,V> next = e.next;`，目前节点a没有next，所以变量next指向null；

2、 `e.next = newTable[i];` 其中 newTable[i] 指向节点b，那就是把a的next指向了节点b，这样a和b就相互引用了，形成了一个环；

3、 `newTable[i] = e` 把节点a放到了数组i位置；

4、 `e = next;` 把变量e赋值为null，因为第一步中变量next就是指向null；

所以最终的引用关系是这样的：



节点a和b互相引用，形成了一个环，当在数组该位置get寻找对应的key时，就发生了死循环。

另外，如果线程2把newTable设置成到内部的table，节点c的数据就丢了，看来还有数据遗失的问题。

总结

所以在并发的情况，发生扩容时，可能会产生循环链表，在执行get的时候，会触发死循环，引起CPU的100%问题，所以一定要避免在并发环境下使用HashMap。

曾经有人把这个问题报给了Sun，不过Sun不认为这是一个bug，因为在HashMap本来就不支持多线程使用，要并发就用ConcurrentHashmap。

volatile的理解

volatile特性

内存可见性：通俗来说就是，线程A对一个volatile变量的修改，对于其它线程来说是可见的，即线程每次获取volatile变量的值都是最新的。

volatile的使用场景

通过关键字sychronize可以防止多个线程进入同一段代码，在某些特定场景中，volatile相当于一个轻量级的sychronize，因为不会引起线程的上下文切换，但是使用volatile必须满足两个条件：
1、对变量的写操作不依赖当前值，如多线程下执行a++，是无法通过volatile保证结果准确性的；
2、该变量没有包含在具有其它变量的不变式中，这句话有点拗口，看代码比较直观。

```
public class NumberRange {
    private volatile int lower = 0;
     private volatile int upper = 10;

    public int getLower() { return lower; }
    public int getUpper() { return upper; }

    public void setLower(int value) {
        if (value > upper)
            throw new IllegalArgumentException(...);
        lower = value;
    }

    public void setUpper(int value) {
        if (value < lower)
            throw new IllegalArgumentException(...);
        upper = value;
    }
}
```

上述代码中，上下界初始化分别为0和10，假设线程A和B在某一时刻同时执行了setLower(8)和setUpper(5)，且都通过了不变式的检查，设置了一个无效范围（8, 5），所以在这种场景下，需要通过sychronize保证方法setLower和setUpper在每一时刻只有一个线程能够执行。

下面是我们在项目中经常会用到volatile关键字的两个场景：

1、**状态标记量**
在高并发的场景中，通过一个boolean类型的变量isopen，控制代码是否走促销逻辑，该如何实现？

```
public class ServerHandler {
    private volatile isopen;
    public void run() {
        if (isopen) {
           //促销逻辑
        } else {
          //正常逻辑
        }
    }
    public void setIsopen(boolean isopen) {
        this.isopen = isopen
    }
}
```

场景细节无需过分纠结，这里只是举个例子说明volatile的使用方法，用户的请求线程执行run方法，如果需要开启促销活动，可以通过后台设置，具体实现可以发送一个请求，调用setIsopen方法并设置isopen为true，由于isopen是volatile修饰的，所以一经修改，其他线程都可以拿到isopen的最新值，用户请求就可以执行促销逻辑了。

2、**double check**
单例模式的一种实现方式，但很多人会忽略volatile关键字，因为没有该关键字，程序也可以很好的运行，只不过代码的稳定性总不是100%，说不定在未来的某个时刻，隐藏的bug就出来了。

```
class Singleton {
    private volatile static Singleton instance;
    public static Singleton getInstance() {
        if (instance == null) {
            syschronized(Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

不过在众多单例模式的实现中，我比较推荐懒加载的优雅写法Initialization on Demand Holder（IODH）。

```
public class Singleton {
    static class SingletonHolder {
        static Singleton instance = new Singleton();
    }

    public static Singleton getInstance(){
        return SingletonHolder.instance;
    }
}
```

当然，如果不需要懒加载的话，直接初始化的效果更好。

如何保证内存可见性?

在java虚拟机的内存模型中，有主内存和工作内存的概念，每个线程对应一个工作内存，并共享主内存的数据，下面看看操作普通变量和volatile变量有什么不同：

1、对于普通变量：读操作会优先读取工作内存的数据，如果工作内存中不存在，则从主内存中拷贝一份数据到工作内存中；写操作只会修改工作内存的副本数据，这种情况下，其它线程就无法读取变量的最新值。

2、对于volatile变量，读操作时JMM会把工作内存中对应的值设为无效，要求线程从主内存中读取数据；写操作时JMM会把工作内存中对应的数据刷新到主内存中，这种情况下，其它线程就可以读取变量的最新值。

volatile变量的内存可见性是基于内存屏障(Memory Barrier)实现的，什么是内存屏障？内存屏障，又称内存栅栏，是一个CPU指令。在程序运行时，为了提高执行性能，编译器和处理器会对指令进行重排序，JMM为了保证在不同的编译器和CPU上有相同的结果，通过插入特定类型的内存屏障来禁止特定类型的编译器重排序和处理器重排序，插入一条内存屏障会告诉编译器和CPU：不管什么指令都不能和这条Memory Barrier指令重排序。

这段文字显得有点苍白无力，不如来段简明的代码：

```
class Singleton {
    private volatile static Singleton instance;
    private int a;
    private int b;
    private int b;
    public static Singleton getInstance() {
        if (instance == null) {
            syschronized(Singleton.class) {
                if (instance == null) {
                    a = 1;  // 1
                     b = 2;  // 2
                    instance = new Singleton();  // 3
                    c = a + b;  // 4
                }
            }
        }
        return instance;
    }
}
```

1、如果变量instance没有volatile修饰，语句1、2、3可以随意的进行重排序执行，即指令执行过程可能是3214或1324。
2、如果是volatile修饰的变量instance，会在语句3的前后各插入一个内存屏障。

通过观察volatile变量和普通变量所生成的汇编代码可以发现，操作volatile变量会多出一个lock前缀指令：

```
Java代码：
instance = new Singleton();

汇编代码：
0x01a3de1d: movb $0x0,0x1104800(%esi);
0x01a3de24: **lock** addl $0x0,(%esp);
```

这个lock前缀指令相当于上述的内存屏障，提供了以下保证：
1、将当前CPU缓存行的数据写回到主内存；
2、这个写回内存的操作会导致在其它CPU里缓存了该内存地址的数据无效。

CPU为了提高处理性能，并不直接和内存进行通信，而是将内存的数据读取到内部缓存（L1，L2）再进行操作，但操作完并不能确定何时写回到内存，如果对volatile变量进行写操作，当CPU执行到Lock前缀指令时，会将这个变量所在缓存行的数据写回到内存，不过还是存在一个问题，就算内存的数据是最新的，其它CPU缓存的还是旧值，所以为了保证各个CPU的缓存一致性，每个CPU通过嗅探在总线上传播的数据来检查自己缓存的数据有效性，当发现自己缓存行对应的内存地址的数据被修改，就会将该缓存行设置成无效状态，当CPU读取该变量时，发现所在的缓存行被设置为无效，就会重新从内存中读取数据到缓存中。

线程

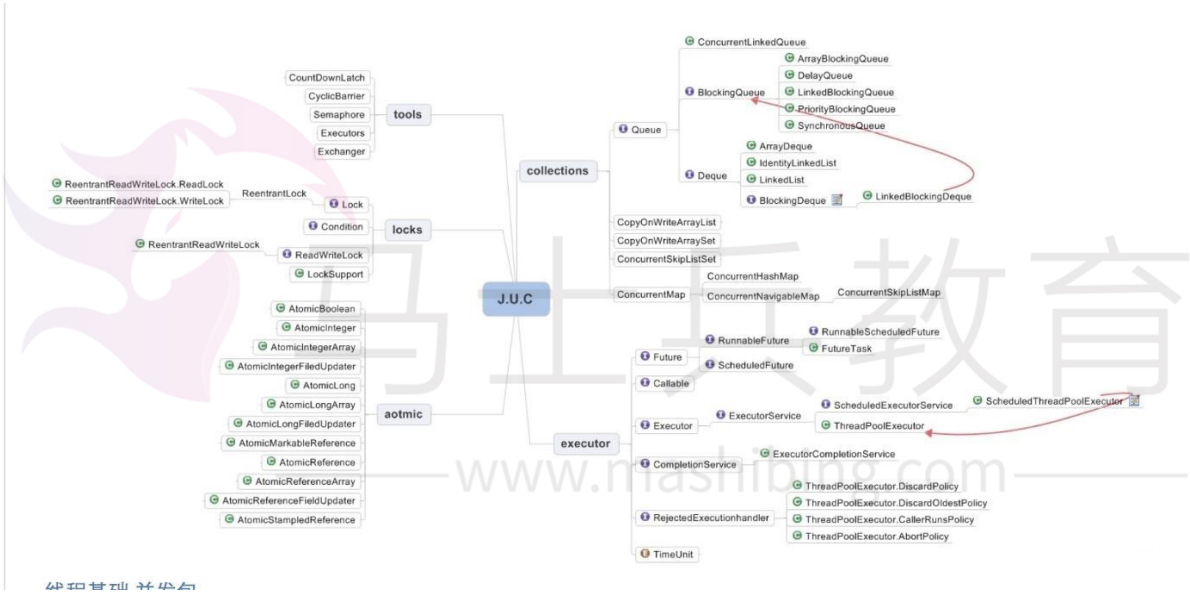
这里所说的线程指程序执行过程中的一个线程实体。JVM 允许一个应用并发执行多个线程。 Hotspot JVM 中的 Java 线程与原生操作系统线程有直接的映射关系。当线程本地存储、缓冲区分配、同步对象、栈、程序计数器等准备好以后，就会创建一个操作系统原生线程。

Java 线程结束，原生线程随之被回收。操作系统负责调度所有线程，并把它们分配到任何可用的 CPU 上。当原生线程初始化完毕，就会调用 Java 线程的 run() 方法。当线程结束时，会释放原生线程和 Java 线程的所有资源。

Hotspot JVM 后台运行的系统线程主要有下面几个：

虚拟机线程 （VM thread）	**这个线程等待 JVM 到达安全点操作出现。这些操作必须要在独立的线程里执行，因为当堆修改无法进行时，线程都需要 JVM 位于安全点。这些操作的类型有：stop-the- world 垃圾回收、线程栈 dump、线程暂停、线程偏向锁（biased locking）解除。**
周期性任务线程	这线程负责定时器事件（也就是中断），用来调度周期性操作的执行。
GC 线程	这些线程支持 JVM 中不同的垃圾回收活动。
编译器线程	这些线程在运行时将字节码动态编译成本地平台相关的机器码。
信号分发线程	这个线程接收发送到 JVM 的信号并调用适当的 JVM 方法处理。

JAVA 多线程并发



JAVA 线程实现/创建方式

继承 **Thread** 类

Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过 Thread 类的 start()实例方法。start()方法是一个 native 方法，它将启动一个新线程,并执行run()方法

```
public class MyThread extends Thread {
     public void run() {
     System.out.println("MyThread.run()");
     }
}
MyThread myThread1 = new MyThread();
myThread1.start();
```

实现 **Runnable** 接口。

如果自己的类已经 extends 另一个类，就无法直接 extends Thread，此时，可以实现一个 Runnable 接口。

```
public class MyThread extends OtherClass implements Runnable {
public void run() {
       System.out.println("MyThread.run()");
      }
}

//启动 MyThread，需要首先实例化一个 Thread，并传入自己的 MyThread 实例：
MyThread myThread = new MyThread();
Thread thread = new Thread(myThread);
thread.start();
//事实上，当传入一个 Runnable target 参数给 Thread 后，Thread 的 run()方法就会调用
target.run()
public void run() {
     if (target != null) {
     target.run();
     }
}
```

ExecutorService、Callable、Future 有返回值线程

有返回值的任务必须实现 Callable 接口，类似的，无返回值的任务必须 Runnable 接口。执行 Callable 任务后，可以获取一个 Future 的对象，在该对象上调用 get 就可以获取到 Callable 任务返回的 Object 了，再结合线程池接口 ExecutorService 就可以实现传说中有返回结果的多线程了。

```
//创建一个线程池
ExecutorService pool = Executors.newFixedThreadPool(taskSize);
// 创建多个有返回值的任务
List<Future> list = new ArrayList<Future>();
for (int i = 0; i < taskSize; i++) {
Callable c = new MyCallable(i + " ");
// 执行任务并获取 Future 对象
Future f = pool.submit(c);
list.add(f);
}
// 关闭线程池
pool.shutdown();
// 获取所有并发任务的运行结果
for (Future f : list) {
// 从 Future 对象上获取任务的返回值，并输出到控制台
System.out.println("res：" + f.get().toString());
}
```

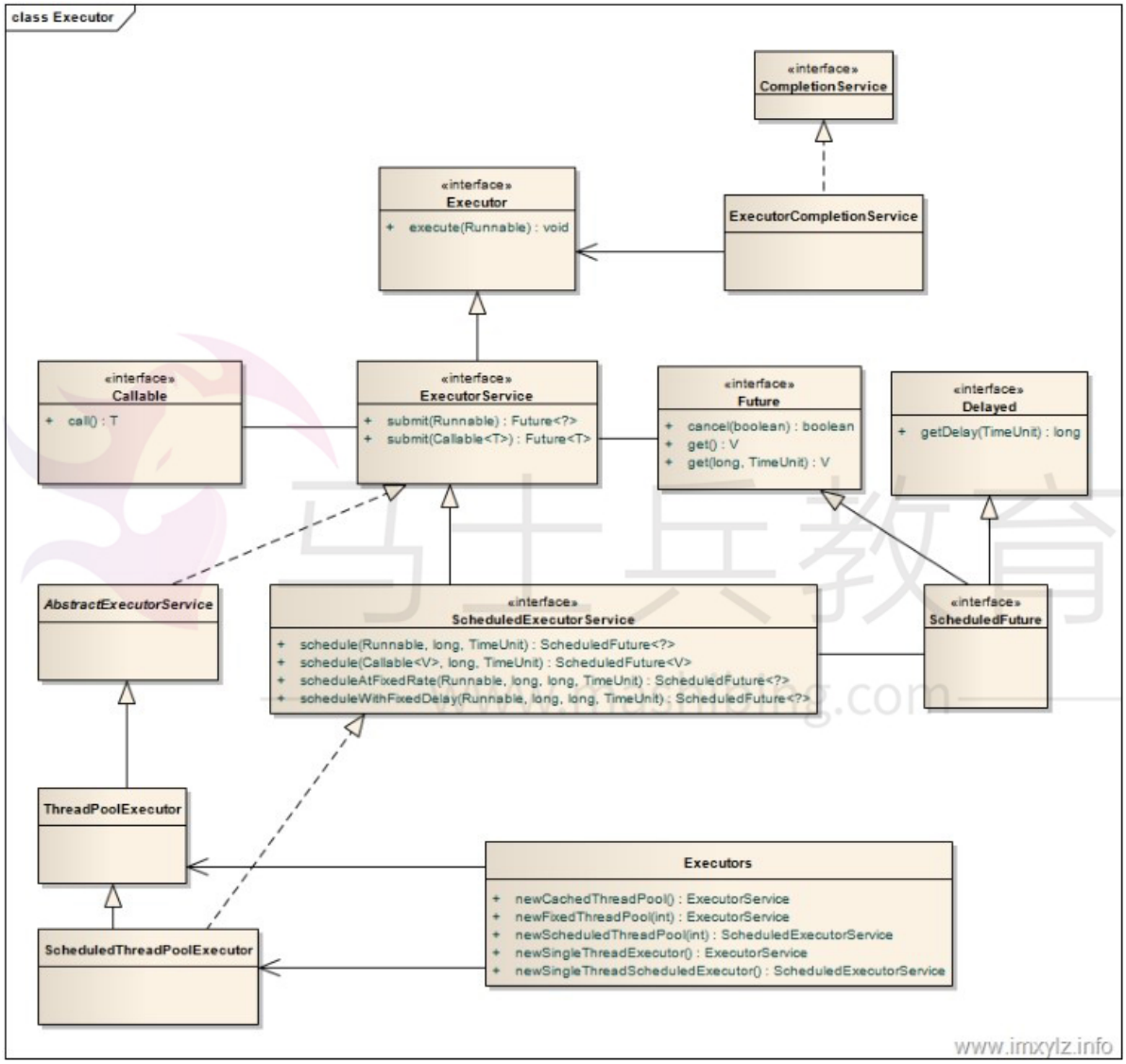
基于线程池的方式

线程和数据库连接这些资源都是非常宝贵的资源。那么每次需要的时候创建，不需要的时候销毁，是非常浪费资源的。那么我们就可以使用缓存的策略，也就是使用线程池。

```
 // 创建线程池
        ExecutorService threadPool = Executors.newFixedThreadPool(10);
        while(true) {
            threadPool.execute(new Runnable() { // 提交多个线程任务，并执行
                @Override
                public void run() {
                    System.out.println(Thread.currentThread().getName() + " is
running ..");
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
e.printStackTrace();
                    }
                }
            });
        }
}
```

4种线程池

Java 里面线程池的顶级接口是 **Executor**，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 **ExecutorService**。



newCachedThreadPool

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。

newFixedThreadPool

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

newScheduledThreadPool

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

```
\ScheduledExecutorService scheduledThreadPool=
Executors.newScheduledThreadPool(3);
scheduledThreadPool.schedule(newRunnable(){
@Override
public void run() { System.out.println("延迟三秒"); }
}, 3, TimeUnit.SECONDS);
scheduledThreadPool.scheduleAtFixedRate(newRunnable(){
@Override
public void run() {
System.out.println("延迟 1 秒后每三秒执行一次");
}
},1,3,TimeUnit.SECONDS);
```

newSingleThreadExecutor

Executors.newSingleThreadExecutor()返回一个线程池（这个线程池只有一个线程）,这个线程池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去!

线程生命周期(状态)

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换

新建状态（NEW）

当程序使用 new 关键字创建了一个线程之后，该线程就处于新建状态，此时仅由 JVM 为其分配内存，并初始化其成员变量的值

就绪状态（RUNNABLE**）：**

当线程对象调用了 start()方法之后，该线程处于就绪状态。Java 虚拟机会为其创建方法调用栈和程序计数器，等待调度运行。

阻塞状态（BLOCKED**）：**

阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得 cpu timeslice 转到运行(running)状

态。阻塞的情况分三种：

等待阻塞（o.wait->等待对列）：

运行(running)的线程执行 o.wait()方法，JVM 会把该线程放入等待队列(waitting queue)中。

同步阻塞(lock->锁池)

运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池(lock pool)中。

其他阻塞(sleep/join)

运行(running)的线程执行 Thread.sleep(long ms)或 t.join()方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O处理完毕时，线程重新转入可运行(runnable)状态。

线程死亡（DEAD）

线程会以下面三种方式结束，结束后就是死亡状态。

正常结束

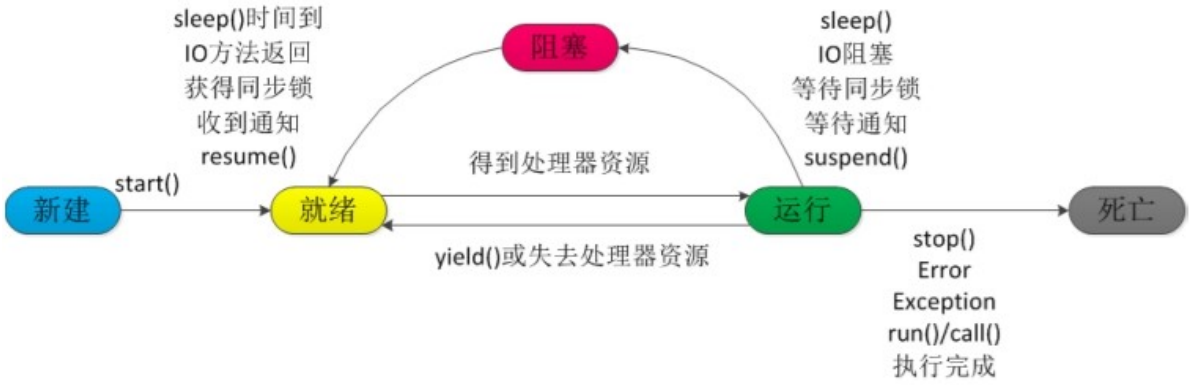
1. run()或 call()方法执行完成，线程正常结束。

异常结束

2. 线程抛出一个未捕获的 Exception 或 Error。

调用stop

3. 直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁，不推荐使用。



终止线程 4 种方式

正常运行结束

程序运行结束，线程自动结束。

使用退出标志退出线程

一般 run()方法执行完，线程就会正常结束，然而，常常有些线程是伺服线程。它们需要长时间的

运行，只有在外部某些条件满足的情况下，才能关闭这些线程。使用一个变量来控制循环，例如：

最直接的方法就是设一个 boolean 类型的标志，并通过设置这个标志为 true 或 false 来控制 while循环是否退出，代码示例：

```
public class ThreadSafe extends Thread {
public volatile boolean exit = false;
public void run() { while (!exit){
//do something
}
}
}
```

定义了一个退出标志 exit，当 exit 为 true 时，while 循环退出，exit 的默认值为 false.在定义 exit时，使用了一个 Java 关键字 volatile，这个关键字的目的是使 exit 同步，也就是说在同一时刻只能由一个线程来修改 exit 的值。

Interrupt 方法结束线程

使用 interrupt()方法来中断线程有两种情况：

1. 线程处于阻塞状态：如使用了 sleep,同步锁的 wait,socket 中的 receiver,accept 等方法时，会使线程处于阻塞状态。当调用线程的 interrupt()方法时，会抛出 InterruptException 异常。阻塞中的那个方法抛出这个异常，通过代码捕获该异常，然后 break 跳出循环状态，从而让我们有机会结束这个线程的执行。通常很多人认为只要调用 interrupt 方法线程就会结束，实际上是错的， 一定要先捕获 InterruptedException 异常之后通过 break 来跳出循环，才能正常结束 run 方法。

2. 线程未处于阻塞状态：使用 isInterrupted()判断线程的中断标志来退出循环。当使用interrupt()方法时，中断标志就会置 true，和使用自定义的标志来控制循环是一样的道理。

```
public class ThreadSafe extends Thread {
public void run() {
while (!isInterrupted()){ //非阻塞过程中通过判断中断标志来退出
try{
Thread.sleep(5*1000);//阻塞过程捕获中断异常来退出
}catch(InterruptedException e){
e.printStackTrace();
break;//捕获到异常之后，执行 break 跳出循环
}
}
}
}
```

stop 方法终止线程（线程不安全）

程序中可以直接使用 thread.stop()来强行终止线程，但是 stop 方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，不安全主要是：thread.stop()调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子线程所持有的所有锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。因此，并不推荐使用 stop 方法来终止线程。

leep 与 wait 区别

1. 对于 sleep()方法，我们首先要知道该方法是属于 Thread 类中的。而 wait()方法，则是属于 Object 类中的。
2. sleep()方法导致了程序暂停执行指定的时间，让出 cpu 该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。
3. 在调用 sleep()方法的过程中，线程不会释放对象锁。
4. 而当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用 notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

start 与 run 区别

1. **start**（）方法来启动线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码。
2. 通过调用 Thread 类的 start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。
3. 方法 run()称为线程体，它包含了要执行的这个线程的内容，线程就进入了运行状态，开始运行 run 函数当中的代码。 Run 方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

JAVA 后台线程

1. 定义：守护线程--也称"服务线程"，他是后台线程，它有一个特性，即为用户线程 提供 公共服务，在没有用户线程可服务时会自动离开。
2. 优先级：守护线程的优先级比较低，用于为系统中的其它对象和线程提供服务。
3. 设置：通过 setDaemon(true)来设置线程为"守护线程"；将一个用户线程设置为守护线程的方式是在 线程对象创建 之前 用线程对象的 setDaemon 方法。
4. 在 Daemon 线程中产生的新线程也是 Daemon 的。
5. 线程则是 JVM 级别的，以 Tomcat 为例，如果你在 Web 应用中启动一个线程，这个线程的生命周期并不会和 Web 应用程序保持同步。也就是说，即使你停止了 Web 应用，这个线程依旧是活跃

的。

6. example: 垃圾回收线程就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread, 程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是 JVM 上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。
7. 生命周期：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统“同生共死”。当 JVM 中所有的线程都是守护线程的时候，JVM 就可以退出了；如果还有一个或以上的非守护线程则 JVM 不会退出。

JAVA 锁

乐观锁

乐观锁是一种乐观思想，即认为读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读-比较-写的操作。

java 中的乐观锁基本都是通过 CAS 操作实现的，CAS 是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。

悲观锁

悲观锁是就是悲观思想，即认为写多，遇到并发写的可能性高，每次去拿数据的时候都认为别人会修改，所以每次在读写数据的时候都会上锁，这样别人想读写这个数据就会block直到拿到锁。 java中的悲观锁就是Synchronized,AQS框架下的锁则是先尝试cas乐观锁去获取锁，获取不到，才会转换为悲观锁，如 RetreenLock。

自旋锁

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。

线程自旋是需要消耗 cup 的，说白了就是让 cup 在做无用功，如果一直获取不到锁，那线程也不能一直占用 cup 自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用 cpu 做无用功，占着 XX 不 XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要 cup 的线程又不能获取到 cpu，造成 cpu 的浪费。所以这种情况下我们要关闭自旋锁；

自旋锁时间阈值（1.6引入了适应性自旋锁）

自旋锁的目的是为了占着 CPU 的资源不释放，等到获取到锁立即进行处理。但是如何去选择自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用 CPU 资源，进而会影响整体系统的性能。因此自旋的周期选的额外重要！

JVM 对于自旋周期的选择，jdk1.5 这个限度是一定的写死的，在 1.6 引入了适应性自旋锁，适应性自旋锁意味着自旋的时间不在是固定的了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定，基本认为一个线程上下文切换的时间是最佳的一个时间，同时 JVM 还针对当前 CPU 的负荷情况做了较多的优化，如果平均负载小于 CPUs 则一直自旋，如果有超过(CPUs/2) 个线程正在自旋，则后来线程直接阻塞，如果正在自旋的线程发现 Owner 发生了变化则延迟自旋时间（自旋计数）或进入阻塞，如果 CPU 处于节电模式则停止自旋，自旋时间的最坏情况是 CPU 的存储延迟（CPU A 存储了一个数据，到 CPU B 得知这个数据直接的时间差），自旋时会适当放弃线程优先级之间的差异。

自旋锁的开启

JDK1.6 中-XX:+UseSpinning 开启；

-XX:PreBlockSpin=10 为自旋次数；

JDK1.7 后，去掉此参数，由 jvm 控制；

Synchronized 同步锁

synchronized 它可以把任意一个非 NULL 的对象当作锁。他属于独占式的悲观锁，同时属于可重入锁。

Synchronized 作用范围

1. 作用于方法时，锁住的是对象的实例(this)；
2. 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带PermGen

（jdk1.8 则是 metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；

3. synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。它有多个队列，

当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

Synchronized 核心组件

1) Wait Set：哪些调用 wait 方法被阻塞的线程被放置在这里；

2) Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中；

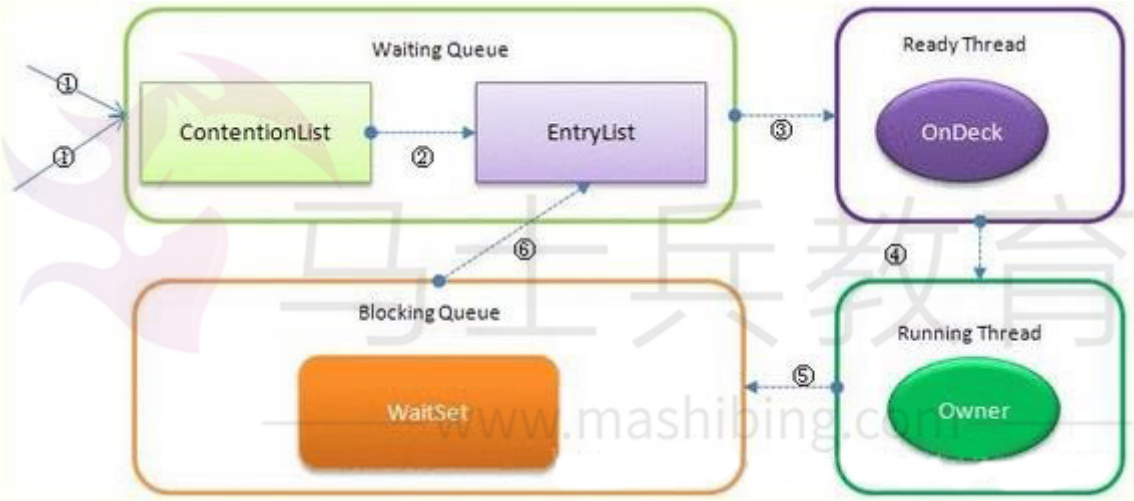
3) Entry List：Contention List 中那些有资格成为候选资源的线程被移动到 Entry List 中；

4) OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为 OnDeck；

5) Owner：当前已经获取到所资源的线程被称为 Owner；

6) !Owner：当前释放锁的线程。

Synchronized 实现



1. JVM 每次从队列的尾部取出一个数据用于锁竞争候选者（OnDeck），但是并发情况下，ContentionList 会被大量的并发线程进行 CAS 访问，为了降低对尾部元素的竞争，JVM 会将一部分线程移动到 EntryList 中作为候选竞争线程。
2. Owner 线程会在 unlock 时，将 ContentionList 中的部分线程迁移到 EntryList 中，并指定 EntryList 中的某个线程为 OnDeck 线程（一般是最先进去的那个线程）。
3. Owner 线程并不直接把锁传递给 OnDeck 线程，而是把锁竞争的权利交给 OnDeck，OnDeck需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在JVM 中，也把这种选择行为称之为“竞争切换”。
4. OnDeck 线程获取到锁资源后会变为 Owner 线程，而没有得到锁资源的仍然停留在 EntryList 中。如果Owner线程被wait方法阻塞，则转移到WaitSet队列中，直到某个时刻通过notify 或者 notifyAll 唤醒，会重新进去 EntryList 中。
5. 处于 ContentionList、EntryList、WaitSet 中的线程都处于阻塞状态，该阻塞是由操作系统来完成的（Linux 内核下采用pthread_mutex_lock 内核函数实现的）。
6. Synchronized 是非公平锁。 Synchronized 在线程进入 ContentionList 时，等待的线程会先尝试自旋获取锁，如果获取不到就进入 ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占 OnDeck 线程的锁资源。
7. 每个对象都有个 monitor 对象，加锁就是在竞争 monitor 对象，代码块加锁是在前后分别加上 monitorenter 和 monitorexit 指令来实现的，方法加锁是通过一个标记位来判断的
8. synchronized 是一个重量级操作，需要调用操作系统相关接口，性能是低效的，有可能给线程加锁消耗的时间比有用操作消耗的时间更多。
9. Java1.6，synchronized进行了很多的优化，有适应自旋、锁消除、锁粗化、轻量级锁及偏向锁等，效率有了本质上的提高。在之后推出的 Java1.7 与 1.8 中，均对该关键字的实现机理做了优化。引入了偏向锁和轻量级锁。都是在对象头中有标记位，不需要经过操作系统加锁。
10. 锁可以从偏向锁升级到轻量级锁，再升级到重量级锁。这种升级过程叫做锁膨胀；
11. JDK 1.6 中默认是开启偏向锁和轻量级锁，可以通过-XX:-UseBiasedLocking 来禁用偏向锁。

java 中的锁 -- 偏向锁、轻量级锁、自旋锁、重量级锁

之前做过一个测试，详情见这篇文章《多线程 +1操作的几种实现方式，及效率对比》，当时对这个测试结果很疑惑，反复执行过多次，发现结果是一样的:

1. 单线程下synchronized效率最高（当时感觉它的效率应该是最差才对）；
2. AtomicInteger效率最不稳定，不同并发情况下表现不一样：短时间低并发下，效率比synchronized高，有时甚至比LongAdder还高出一点，但是高并发下，性能还不如synchronized，不同情况下性能表现很不稳定；
3. LongAdder性能稳定，在各种并发情况下表现都不错，整体表现最好,短时间的低并发下比AtomicInteger性能差一点，长时间高并发下性能最高（可以让AtomicInteger下台了）；

这篇文章我们就去揭秘，为什么会是这个测试结果！

理解锁的基础知识

如果想要透彻的理解java锁的来龙去脉，需要先了解以下基础知识。

基础知识之一：锁的类型

锁从宏观上分类，分为悲观锁与乐观锁。

####### 乐观锁

乐观锁是一种乐观思想，即认为读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读-比较-写的操作。

java中的乐观锁基本都是通过CAS操作实现的，CAS是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。

####### 悲观锁

悲观锁是就是悲观思想，即认为写多，遇到并发写的可能性高，每次去拿数据的时候都认为别人会修改，所以每次在读写数据的时候都会上锁，这样别人想读写这个数据就会block直到拿到锁。java中的悲观锁就是Synchronized,AQS框架下的锁则是先尝试cas乐观锁去获取锁，获取不到，才会转换为悲观锁，如RetreenLock。

基础知识之二：java线程阻塞的代价

java的线程是映射到操作系统原生线程之上的，如果要阻塞或唤醒一个线程就需要操作系统介入，需要在户态与核心态之间切换，这种切换会消耗大量的系统资源，因为用户态与内核态都有各自专用的内存空间，专用的寄存器等，用户态切换至内核态需要传递给许多变量、参数给内核，内核也需要保护好用户态在切换时的一些寄存器值、变量等，以便内核态调用结束后切换回用户态继续工作。

1. 如果线程状态切换是一个高频操作时，这将会消耗很多CPU处理时间；
2. 如果对于那些需要同步的简单的代码块，获取锁挂起操作消耗的时间比用户代码执行的时间还要长，这种同步策略显然非常糟糕的。

synchronized会导致争用不到锁的线程进入阻塞状态，所以说它是java语言中一个重量级的同步操纵，被称为重量级锁，为了缓解上述性能问题，JVM从1.5开始，引入了轻量锁与偏向锁，默认启用了自旋锁，他们都属于乐观锁。

明确java线程切换的代价，是理解java中各种锁的优缺点的基础之一。

基础知识之三：markword

在介绍java锁之前，先说下什么是markword，markword是java对象数据结构中的一部分，要详细了解java对象的结构可以点击这里,这里只做markword的详细介绍，因为对象的markword和java各种类型的锁密切相关；

markword数据的长度在32位和64位的虚拟机（未开启压缩指针）中分别为32bit和64bit，它的**最后2bit是锁状态标志位**，用来标记当前对象的状态，对象的所处的状态，决定了markword存储的内容，如下表所示:

状态	标志位	存储内容
未锁定	01	对象哈希码、对象分代年龄
轻量级锁定	00	指向锁记录的指针
膨胀(重量级锁定)	10	执行重量级锁定的指针
GC标记	11	空(不需要记录信息)
可偏向	01	偏向线程ID、偏向时间戳、对象分代年龄

32位虚拟机在不同状态下markword结构如下图所示：

锁状态	25 bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01
无锁	对象的hashCode		对象分代年龄	0	01

了解了markword结构，有助于后面了解java锁的加锁解锁过程；

小结

前面提到了java的4种锁，他们分别是重量级锁、自旋锁、轻量级锁和偏向锁，
不同的锁有不同特点，每种锁只有在其特定的场景下，才会有出色的表现，java中没有哪种锁能够在所有情况下都能有出色的效率，引入这么多锁的原因就是为了应对不同的情况；

前面讲到了重量级锁是悲观锁的一种，自旋锁、轻量级锁与偏向锁属于乐观锁，所以现在你就能够大致理解了他们的适用范围，但是具体如何使用这几种锁呢，就要看后面的具体分析他们的特性；

java中的锁

自旋锁

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就**避免用户线程和内核的切换的消耗。**

但是线程自旋是需要消耗cup的，说白了就是让cup在做无用功，如果一直获取不到锁，那线程也不能一直占用cup自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

####### 自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用cpu做无用功，占着XX不XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要cup的线程又不能获取到cpu，造成cpu的浪费。所以这种情况下我们要关闭自旋锁；

####### 自旋锁时间阈值

自旋锁的目的是为了占着CPU的资源不释放，等到获取到锁立即进行处理。但是如何去选择自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用CPU资源，进而会影响整体系统的性能。因此自旋的周期选的额外重要！

JVM对于自旋周期的选择，jdk1.5这个限度是一定的写死的，在1.6引入了适应性自旋锁，适应性自旋锁意味着自旋的时间不在是固定的了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定，基本认为一个线程上下文切换的时间是最佳的一个时间，同时JVM还针对当前CPU的负荷情况做了较多的优化

1. 如果平均负载小于CPUs则一直自旋

2. 如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞
3. 如果正在自旋的线程发现Owner发生了变化则延迟自旋时间（自旋计数）或进入阻塞
4. 如果CPU处于节电模式则停止自旋
5. 自旋时间的最坏情况是CPU的存储延迟（CPU A存储了一个数据，到CPU B得知这个数据直接的时间差）
6. 自旋时会适当放弃线程优先级之间的差异

####### 自旋锁的开启

JDK1.6中-XX:+UseSpinning开启；
-XX:PreBlockSpin=10 为自旋次数；
JDK1.7后，去掉此参数，由jvm控制；

重量级锁Synchronized

####### Synchronized的作用

在JDK1.5之前都是使用synchronized关键字保证同步的，Synchronized的作用相信大家都已经非常熟悉了；

它可以把任意一个非NULL的对象当作锁。

1. 作用于方法时，锁住的是对象的实例(this)；
2. 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带PermGen（jdk1.8则是metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；
3. synchronized作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。

####### Synchronized的实现

它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

1. Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中；
2. Entry List：Contention List中那些有资格成为候选资源的线程被移动到Entry List中；
3. Wait Set：哪些调用wait方法被阻塞的线程被放置在这里；
4. OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为OnDeck；
5. Owner：当前已经获取到所资源的线程被称为Owner；
6. !Owner：当前释放锁的线程。

JVM每次从队列的尾部取出一个数据用于锁竞争候选者（OnDeck），但是并发情况下，ContentionList会被大量的并发线程进行CAS访问，为了降低对尾部元素的竞争，JVM会将一部分线程移动到EntryList中作为候选竞争线程。Owner线程会在unlock时，将ContentionList中的部分线程迁移到EntryList中，并指定EntryList中的某个线程为OnDeck线程（一般是最先进去的那个线程）。Owner线程并不直接把锁传递给OnDeck线程，而是把锁竞争的权利交给OnDeck，OnDeck需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在JVM中，也把这种选择行为称之为“竞争切换”。

OnDeck线程获取到锁资源后会变为Owner线程，而没有得到锁资源的仍然停留在EntryList中。如果Owner线程被wait方法阻塞，则转移到WaitSet队列中，直到某个时刻通过notify或者notifyAll唤醒，会重新进去EntryList中。

处于ContentionList、EntryList、WaitSet中的线程都处于阻塞状态，该阻塞是由操作系统来完成的（Linux内核下采用pthread_mutex_lock内核函数实现的）。

Synchronized是非公平锁。 Synchronized在线程进入ContentionList时，等待的线程会先尝试自旋获取锁，如果获取不到就进入ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占OnDeck线程的锁资源。

偏向锁

Java偏向锁(Biased Locking)是Java6引入的一项多线程优化。
偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，这种情况下，就会给线程加一个偏向锁。
如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

> *它通过消除资源无竞争情况下的同步原语，进一步提高了程序的运行性能。*

####### 偏向锁的实现

######## 偏向锁获取过程：

1. 访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01，确认为可偏向状态。
2. 如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤5，否则进入步骤3。
3. 如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行5；如果竞争失败，执行4。
4. 如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。（撤销偏向锁的时候会导致stop the word）
5. 执行同步代码。

> 注意：第四步中到达安全点safepoint会导致stop the word，时间很短。

######## 偏向锁的释放：

偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

####### 偏向锁的适用场景

始终只有一个线程在执行同步块，在它没有执行完释放锁之前，没有其它线程去执行同步块，在锁无竞争的情况下使用，一旦有了竞争就升级为轻量级锁，升级为轻量级锁的时候需要撤销偏向锁，撤销偏向锁的时候会导致stop the word操作；
在有锁的竞争时，偏向锁会多做很多额外操作，尤其是撤销偏向所的时候会导致进入安全点，安全点会导致stw，导致性能下降，这种情况下应当禁用；

######## 查看停顿–安全点停顿日志

要查看安全点停顿，可以打开安全点日志，通过设置JVM参数 -XX:+PrintGCApplicationStoppedTime 会打出系统停止的时间，添加-XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1 这两个参数会打印出详细信息，可以查看到使用偏向锁导致的停顿，时间非常短暂，但是争用严重的情况下，停顿次数也会非常多；

注意：安全点日志不能一直打开：

1. 安全点日志默认输出到stdout，一是stdout日志的整洁性，二是stdout所重定向的文件如果不在/dev/shm，可能被锁。
2. 对于一些很短的停顿，比如取消偏向锁，打印的消耗比停顿本身还大。
3. 安全点日志是在安全点内打印的，本身加大了安全点的停顿时间。

所以安全日志应该只在问题排查时打开。
如果在生产系统上要打开，再再增加下面四个参数：
-XX:+UnlockDiagnosticVMOptions -XX: -DisplayVMOutput -XX:+LogVMOutput -XX:LogFile=/dev/shm/vm.log

打开Diagnostic（只是开放了更多的flag可选，不会主动激活某个flag），关掉输出VM日志到stdout，输出到独立文件,/dev/shm目录（内存文件系统）。

```
         vmop                    [threads: total initially_running wait_to_block]    [time: spin block sync cleanup vmop] page_trap_count
0.105: EnableBiasedLocking              [       9          0              0    ]     [     0     0     0     0     0    ]  0
108
         vmop                    [threads: total initially_running wait_to_block]    [time: spin block sync cleanup vmop] page_trap_count
0.119: ParallelGCFailedAllocation       [       9          0              0    ]     [     0     0     0     0    12    ]  0
         vmop                    [threads: total initially_running wait_to_block]    [time: spin block sync cleanup vmop] page_trap_count
0.132: RevokeBias                       [       9          0              3    ]     [     0     0     0     0     0    ]  0
```

此日志分三部分：
第一部分是时间戳，VM Operation的类型
第二部分是线程概况，被中括号括起来
total: 安全点里的总线程数
initially_running: 安全点开始时正在运行状态的线程数
wait_to_block: 在VM Operation开始前需要等待其暂停的线程数

第三部分是到达安全点时的各个阶段以及执行操作所花的时间，其中最重要的是vmop

- spin: 等待线程响应safepoint号召的时间；
- block: 暂停所有线程所用的时间；
- sync: 等于 spin+block，这是从开始到进入安全点所耗的时间，可用于判断进入安全点耗时；
- cleanup: 清理所用时间；
- vmop: 真正执行VM Operation的时间。

可见，那些很多但又很短的安全点，全都是RevokeBias， 高并发的应用会禁用掉偏向锁。

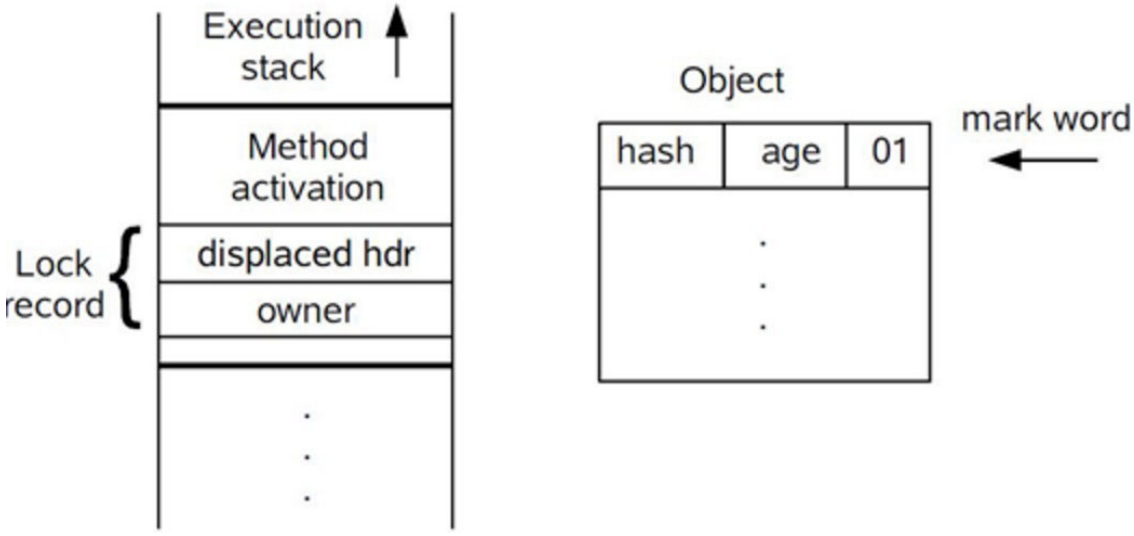
######## jvm开启/关闭偏向锁

- 开启偏向锁：-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0
- 关闭偏向锁：-XX:-UseBiasedLocking

轻量级锁

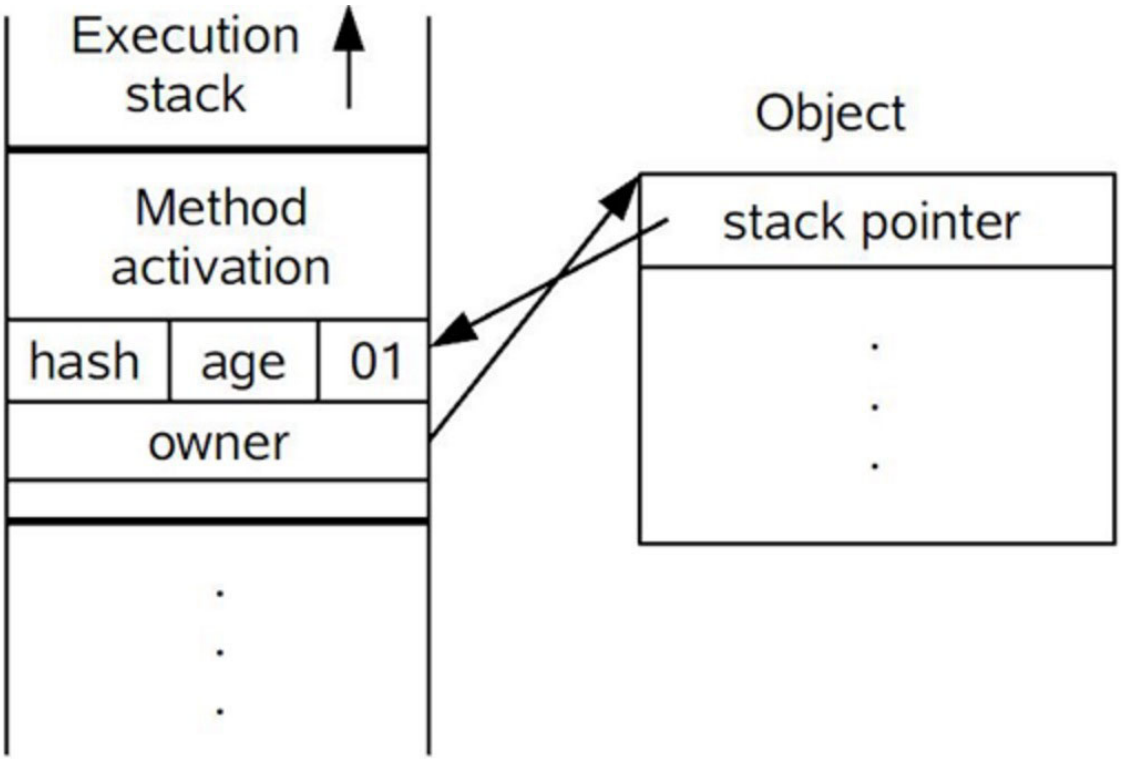
轻量级锁是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，偏向锁就会升级为轻量级锁；
轻量级锁的加锁过程：

1. 在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图：



2. 拷贝对象头中的Mark Word复制到锁记录中；

3. 拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤4，否则执行步骤5。
4. 如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如图所示。



5. 如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。 而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。

####### 轻量级锁的释放

释放锁线程视角：由轻量锁切换到重量锁，是发生在轻量锁释放锁的期间，之前在获取锁的时候它拷贝了锁对象头的markword，在释放锁的时候如果它发现在它持有锁的期间有其他线程来尝试获取锁了，并且该线程对markword做了修改，两者比对发现不一致，则切换到重量锁。

因为重量级锁被修改了，所有display mark word和原来的markword不一样了。

怎么补救，就是进入mutex前，compare一下obj的markword状态。确认该markword是否被其他线程持有。

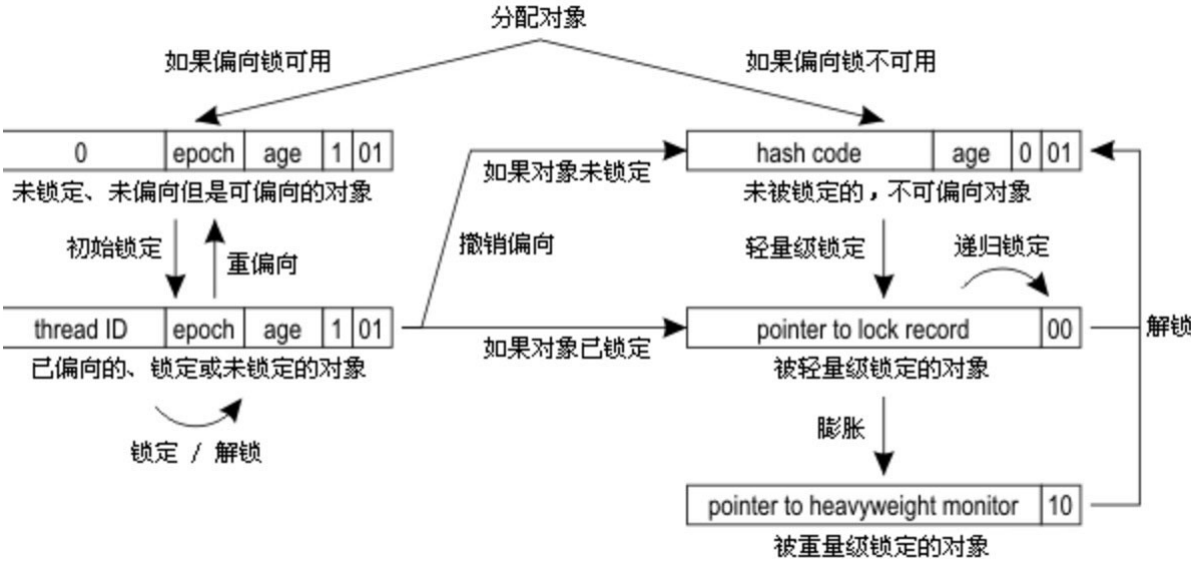
此时如果线程已经释放了markword，那么通过CAS后就可以直接进入线程，无需进入mutex，就这个作用。

尝试获取锁线程视角：如果线程尝试获取锁的时候，轻量锁正被其他线程占有，那么它就会修改markword，修改重量级锁，表示该进入重量锁了。

还有一个注意点：等待轻量锁的线程不会阻塞，它会一直自旋等待锁，并如上所说修改markword。

这就是自旋锁，尝试获取锁的线程，在没有获得锁的时候，不被挂起，而转而去执行一个空循环，即自旋。在若干个自旋后，如果还没有获得锁，则才被挂起，获得锁，则执行代码。

总结



synchronized的执行过程：

1. 检测Mark Word里面是不是当前线程的ID，如果是，表示当前线程处于偏向锁
2. 如果不是，则使用CAS将当前线程的ID替换Mard Word，如果成功则表示当前线程获得偏向锁，置偏向标志位1
3. 如果失败，则说明发生竞争，撤销偏向锁，进而升级为轻量级锁。
4. 当前线程使用CAS将对象头的Mark Word替换为锁记录指针，如果成功，当前线程获得锁
5. 如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。
6. 如果自旋成功则依然处于轻量级状态。
7. 如果自旋失败，则升级为重量级锁。

上面几种锁都是JVM自己内部实现，当我们执行synchronized同步块的时候jvm会根据启用的锁和当前线程的争用情况，决定如何执行同步操作；

在所有的锁都启用的情况下线程进入临界区时会先去获取偏向锁，如果已经存在偏向锁了，则会尝试获取轻量级锁，启用自旋锁，如果自旋也没有获取到锁，则使用重量级锁，没有获取到锁的线程阻塞挂起，直到持有锁的线程执行完同步块唤醒他们；

偏向锁是在无锁争用的情况下使用的，也就是同步开在当前线程没有执行完之前，没有其它线程会执行该同步块，一旦有了第二个线程的争用，偏向锁就会升级为轻量级锁，如果轻量级锁自旋到达阈值后，没有获取到锁，就会升级为重量级锁；

如果线程争用激烈，那么应该禁用偏向锁。

锁优化

以上介绍的锁不是我们代码中能够控制的，但是借鉴上面的思想，我们可以优化我们自己线程的加锁操作；

减少锁的时间

不需要同步执行的代码，能不放在同步快里面执行就不要放在同步快内，可以让锁尽快释放；

减少锁的粒度

它的思想是将物理上的一个锁，拆成逻辑上的多个锁，增加并行度，从而降低锁竞争。它的思想也是用空间来换时间；

java中很多数据结构都是采用这种方法提高并发操作的效率：

ConcurrentHashMap

java中的ConcurrentHashMap在jdk1.8之前的版本，使用一个Segment 数组

```
Segment< K,V >[] segments1
```

Segment继承自ReenTrantLock，所以每个Segment就是个可重入锁，每个Segment 有一个HashEntry< K,V >数组用来存放数据，put操作时，先确定往哪个Segment放数据，只需要锁定这个Segment，执行put，其它的Segment不会被锁定；所以数组中有多少个Segment就允许同一时刻多少个线程存放数据，这样增加了并发能力。

LongAdder

LongAdder 实现思路也类似ConcurrentHashMap，LongAdder有一个根据当前并发状况动态改变的Cell数组，Cell对象里面有一个long类型的value用来存储值;
开始没有并发争用的时候或者是cells数组正在初始化的时候，会使用cas来将值累加到成员变量的base上，在并发争用的情况下，LongAdder会初始化cells数组，在Cell数组中选定一个Cell加锁，数组有多少个cell，就允许同时有多少线程进行修改，最后将数组中每个Cell中的value相加，在加上base的值，就是最终的值；cell数组还能根据当前线程争用情况进行扩容，初始长度为2，每次扩容会增长一倍，直到扩容到大于等于cpu数量就不再扩容，这也就是为什么LongAdder比cas和AtomicInteger效率要高的原因，后面两者都是volatile+cas实现的，他们的竞争维度是1，LongAdder的竞争维度为“Cell个数+1”为什么要+1？因为它还有一个base，如果竞争不到锁还会尝试将数值加到base上；

LinkedBlockingQueue

LinkedBlockingQueue也体现了这样的思想，在队列头入队，在队列尾出队，入队和出队使用不同的锁，相对于LinkedBlockingArray只有一个锁效率要高；

拆锁的粒度不能无限拆，最多可以将一个锁拆为当前cup数量个锁即可；

锁粗化

大部分情况下我们是要让锁的粒度最小化，锁的粗化则是要增大锁的粒度;
在以下场景下需要粗化锁的粒度：
假如有一个循环，循环内的操作需要加锁，我们应该把锁放到循环外面，否则每次进出循环，都进出一次临界区，效率是非常差的；

使用读写锁

ReentrantReadWriteLock 是一个读写锁，读操作加读锁，可以并发读，写操作使用写锁，只能单线程写；

读写分离

CopyOnWriteArrayList 、CopyOnWriteArraySet
CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。
　CopyOnWrite并发容器用于读多写少的并发场景，因为，读的时候没有锁，但是对其进行更改的时候是会加锁的，否则会导致多个线程同时复制出多个副本，各自修改各自的；

使用cas

如果需要同步的操作执行速度非常快，并且线程竞争并不激烈，这时候使用cas效率会更高，因为加锁会导致线程的上下文切换，如果上下文切换的耗时比同步操作本身更耗时，且线程对资源的竞争不激烈，使用volatiled+cas操作会是非常高效的选择；

消除缓存行的伪共享

除了我们在代码中使用的同步锁和jvm自己内置的同步锁外，还有一种隐藏的锁就是缓存行，它也被称为性能杀手。
在多核cup的处理器中，每个cup都有自己独占的一级缓存、二级缓存，甚至还有一个共享的三级缓存，为了提高性能，cpu读写数据是以缓存行为最小单元读写的；32位的cpu缓存行为32字节，64位cup的缓存行为64字节，这就导致了一些问题。
例如，多个不需要同步的变量因为存储在连续的32字节或64字节里面，当需要其中的一个变量时，就将它们作为一个缓存行一起加载到某个cup-1私有的缓存中（虽然只需要一个变量，但是cpu读取会以缓存行为最小单位，将其相邻的变量一起读入），被读入cpu缓存的变量相当于是对主内存变量的一个拷贝，也相当于变相的将在同一个缓存行中的几个变量加了一把锁，这个缓存行中任何一个变量发生了变化，当cup-2需要读取这个缓存行时，就需要先将cup-1中被改变了的整个缓存行更新回主存（即使其它变量没有更改），然后cup-2才能够读取，而cup-2可能需要更改这个缓存行的变量与cpu-1已经更改的缓存行中的变量是不一样的，所以这相当于给几个毫不相关的变量加了一把同步锁；
为了防止伪共享，不同jdk版本实现方式是不一样的：

1. 在jdk1.7之前会 将需要独占缓存行的变量前后添加一组long类型的变量，依靠这些无意义的数组的填充做到一个变量自己独占一个缓存行；
2. 在jdk1.7因为jvm会将这些没有用到的变量优化掉，所以采用继承一个声明了好多long变量的类的方式来实现；
3. 在jdk1.8中通过添加sun.misc.Contended注解来解决这个问题，若要使该注解有效必须在jvm中添加以下参数：
   -XX:-RestrictContended

sun.misc.Contended注解会在变量**前面**添加**128字节**的padding将当前变量与其他变量进行隔离；
关于什么是缓存行，jdk是如何避免缓存行的，网上有非常多的解释，在这里就不再深入讲解了；

ReentrantLock

ReentantLock 继承接口 Lock 并实现了接口中定义的方法，他是一种可重入锁，除了能完成 synchronized 所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。

Lock接口的主要方法

1. void lock(): 执行此方法时, 如果锁处于空闲状态, 当前线程将获取到锁. 相反, 如果锁已经被其他线程持有, 将禁用当前线程, 直到当前线程获取到锁.
2. boolean tryLock()：如果锁可用, 则获取锁, 并立即返回 true, 否则返回 false. 该方法和 lock()的区别在于, tryLock()只是"试图"获取锁, 如果锁不可用, 不会导致当前线程被禁用, 当前线程仍然继续往下执行代码. 而 lock()方法则是一定要获取到锁, 如果锁不可用, 就一直等待, 在未获得锁之前,当前线程并不继续向下执行.

3. void unlock()：执行此方法时, 当前线程将释放持有的锁. 锁只能由持有者释放, 如果线程并不持有锁, 却执行该方法, 可能导致异常的发生.
4. Condition newCondition()：条件对象，获取等待通知组件。该组件和当前的锁绑定，

当前线程只有获取了锁，才能调用该组件的 await()方法，而调用后，当前线程将缩放锁。

5. getHoldCount()： 查询当前线程保持此锁的次数，也就是执行此线程执行lock方法的次数。
6. getQueueLength（）：返回正等待获取此锁的线程估计数，比如启动 10 个线程，1 个线程获得锁，此时返回的是 9
7. getWaitQueueLength：（Condition condition）返回等待与此锁相关的给定条件的线程估计数。比如 10 个线程，用同一个 condition 对象，并且此时这 10 个线程都执行了 condition 对象的 await 方法，那么此时执行此方法返回 10
8. hasWaiters(Condition condition)：查询是否有线程等待与此锁有关的给定条件

(condition)，对于指定 contidion 对象，有多少线程执行了 condition.await 方法

9. hasQueuedThread(Thread thread)：查询给定线程是否等待获取此锁
10. hasQueuedThreads()：是否有线程等待此锁
11. isFair()：该锁是否公平锁
12. isHeldByCurrentThread()： 当前线程是否保持锁锁定，线程的执行 lock 方法的前后分别是 false 和 true
13. isLock()：此锁是否有任意线程占用
14. lockInterruptibly（）：如果当前线程未被中断，获取锁
15. tryLock（）：尝试获得锁，仅在调用时锁未被线程占用，获得锁
16. tryLock(long timeout TimeUnit unit)：如果锁在给定等待时间内没有被另一个线程保持，则获取该锁。

非公平锁

JVM 按随机、就近原则分配锁的机制则称为不公平锁，ReentrantLock 在构造函数中提供了是否公平锁的初始化方式，默认为非公平锁。非公平锁实际执行的效率要远远超出公平锁，除非程序有特殊需要，否则最常用非公平锁的分配机制。

公平锁

公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁，

ReentrantLock 在构造函数中提供了是否公平锁的初始化方式来定义公平锁。

ReentrantLock 与synchronized

1. ReentrantLock 通过方法 lock()与 unlock()来进行加锁与解锁操作，与 synchronized 会被 JVM 自动解锁机制不同，ReentrantLock 加锁后需要手动进行解锁。为了避免程序出现异常而无法正常解锁的情况，使用 ReentrantLock 必须在 finally 控制块中进行解锁操作。
2. ReentrantLock 相比 synchronized 的优势是可中断、公平锁、多个锁。这种情况下需要使用 ReentrantLock。

ReentrantLock 实现

```
 public class MyService {             private Lock lock = new ReentrantLock();
    //Lock lock=new ReentrantLock(true);//公平锁
            //Lock lock=new ReentrantLock(false);//非公平锁             private
Condition condition=lock.newCondition();//创建 Condition     public void
testMethod() {
             try {
               lock.lock();//lock 加锁
     //1：wait 方法等待：
            //System.out.println("开始 wait");             condition.await();
```

```
//通过创建 Condition 对象来使线程 wait，必须先执行 lock.lock 方法获得锁
    //:2：signal 方法唤醒
    condition.signal();//condition 对象的 signal 方法可以唤醒 wait 线程
            for (int i = 0; i < 5; i++) {
 System.out.println("ThreadName=" + Thread.currentThread().getName()+ (" " + (i +
1)));
            }
        } catch (InterruptedException e) {
             e.printStackTrace();
        }
        finally
        {
        lock.unlock();
        }
    }
}
```

Condition 类和Object 类锁方法区别区别

1. Condition 类的 awiat 方法和 Object 类的 wait 方法等效
2. Condition 类的 signal 方法和 Object 类的 notify 方法等效
3. Condition 类的 signalAll 方法和 Object 类的 notifyAll 方法等效
4. ReentrantLock 类可以唤醒指定条件的线程，而 object 的唤醒是随机的

tryLock和lock和lockInterruptibly 的区别

1. tryLock 能获得锁就返回 true，不能就立即返回 false，tryLock(long timeout,TimeUnit unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回 false
2. lock 能获得锁就返回 true，不能的话一直等待获得锁
3. lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程， lock 不会抛出异常，而 lockInterruptibly 会抛出异常。

Semaphore 信号量

Semaphore 是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做完自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore 可以用来构建一些对象池，资源池之类的，比如数据库连接池实现互斥锁（计数器为**1**）

我们也可以创建计数为 1 的 Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态。

代码实现它的用法如下：

```
    // 创建一个计数阈值为 5 的信号量对象
        // 只能 5 个线程同时访问
        Semaphore semp = new Semaphore(5);
        try {   // 申请许可              semp.acquire();
            try {
                // 业务逻辑
            } catch (Exception e) {
            } finally {
                // 释放许可
                semp.release();
            }
        } catch (InterruptedException e) {
        }
```

Semaphore 与ReentrantLock

Semaphore 基本能完成 ReentrantLock 的所有工作，使用方法也与之类似，通过 acquire()与 release()方法来获得和释放临界资源。经实测，Semaphone.acquire()方法默认为可响应中断锁，与 ReentrantLock.lockInterruptibly()作用效果一致，也就是说在等待临界资源的过程中可以被 Thread.interrupt()方法中断。

此外，Semaphore 也实现了可轮询的锁请求与定时锁的功能，除了方法名 tryAcquire 与 tryLock 不同，其使用方法与ReentrantLock几乎一致。Semaphore也提供了公平与非公平锁的机制，也可在构造函数中进行设定。

Semaphore的锁释放操作也由手动进行，因此与ReentrantLock 一样，为避免线程因抛出异常而无法正常释放锁的情况发生，释放锁的操作也必须在 finally 代码块中完成。

AtomicInteger

首先说明，此处 AtomicInteger，一个提供原子操作的 Integer 的类，常见的还有AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference 等，他们的实现原理相同，区别在与运算对象类型的不同。令人兴奋地，还可以通过 AtomicReference将一个对象的所有操作转化成原子操作。

我们知道，在多线程程序中，诸如++i 或 i++等运算不具有原子性，是不安全的线程操作之一。通常我们会使用 synchronized 将该操作变成一个原子操作，但 JVM 为此类操作特意提供了一些同步类，使得使用更方便，且使程序运行效率变得更高。通过相关资料显示，通常AtomicInteger 的性能是 ReentantLock 的好几倍。

可重入锁（递归锁）

这里面讲的是广义上的可重入锁，而不是单指JAVA 下的ReentrantLock。可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。在 JAVA 环境下 ReentrantLock 和 synchronized 都是 可重入锁。

公平锁与非公平锁

公平锁（**Fair**）

加锁前检查是否有排队等待的线程，优先排队等待的线程，先来先得

非公平锁（**Nonfair**）

加锁时不考虑排队等待问题，直接尝试获取锁，获取不到自动到队尾等待

1. 非公平锁性能比公平锁高 5~10 倍，因为公平锁需要在多核的情况下维护一个队列
2. Java 中的 synchronized 是非公平锁，ReentrantLock 默认的 lock()方法采用的是非公平锁。

ReadWriteLock 读写锁

为了提高性能，Java 提供了读写锁，在读的地方使用读锁，在写的地方使用写锁，灵活控制，如果没有写锁的情况下，读是无阻塞的,在一定程度上提高了程序的执行效率。读写锁分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，这是由 jvm 自己控制的，你只要上好相应的锁即可。

读锁

如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁

写锁

如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁!

Java 中读写锁有个接口 java.util.concurrent.locks.ReadWriteLock，也有具体的实现 ReentrantReadWriteLock。

共享锁和独占锁 java

并发包提供的加锁模式分为独占锁和共享锁。

独占锁

独占锁模式下，每次只能有一个线程能持有锁，ReentrantLock 就是以独占方式实现的互斥锁。独占锁是一种悲观保守的加锁策略，它避免了读/读冲突，如果某个只读线程获取锁，则其他读线程都只能等待，这种情况下就限制了不必要的并发性，因为读操作并不会影响数据的一致性。

共享锁

共享锁则允许多个线程同时获取锁，并发访问 共享资源，如：ReadWriteLock。共享锁则是一种乐观锁，它放宽了加锁策略，允许多个执行读操作的线程同时访问共享资源。

1. AQS 的内部类 Node 定义了两个常量 SHARED 和 EXCLUSIVE，他们分别标识 AQS 队列中等待线程的锁获取模式。
2. java 的并发包中提供了 ReadWriteLock，读-写锁。它允许一个资源可以被多个读操作访问，或者被一个 写操作访问，但两者不能同时进行。

重量级锁（Mutex Lock）

Synchronized 是通过对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的 Mutex Lock 来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么 Synchronized 效率低的原因。因此，这种依赖于操作系统 Mutex Lock 所实现的锁我们称之为

“重量级锁”。JDK 中对 Synchronized 做的种种优化，其核心都是为了减少这种重量级锁的使用。

JDK1.6 以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和

“偏向锁”。

轻量级锁

锁的状态总共有四种：无锁状态、偏向锁、轻量级锁和重量级锁。

锁升级

随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁（但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级）。

“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

偏向锁

Hotspot 的作者经过以往的研究发现大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得。偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏护。引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次 CAS 原子指令，而偏向锁只需要在置换 ThreadID的时候依赖一次 CAS 原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须

小于节省下来的 CAS 原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

分段锁

分段锁也并非一种实际的锁，而是一种思想 ConcurrentHashMap 是学习分段锁的最好实践

锁优化

减少锁持有时间

只用在有线程安全要求的程序上加锁

减小锁粒度

将大对象（这个对象可能会被很多线程访问），拆成小对象，大大增加并行度，降低锁竞争。

降低了锁的竞争，偏向锁，轻量级锁成功率才会提高。最最典型的减小锁粒度的案例就是

ConcurrentHashMap。

锁分离

最常见的锁分离就是读写锁 ReadWriteLock，根据功能进行分离成读锁和写锁，这样读读不互斥，读写互斥，写写互斥，即保证了线程安全，又提高了性能，具体也请查看[高并发 Java 五]

JDK 并发包 1。读写分离思想可以延伸，只要操作互不影响，锁就可以分离。比如

LinkedBlockingQueue 从头部取出，从尾部放数据

锁粗化

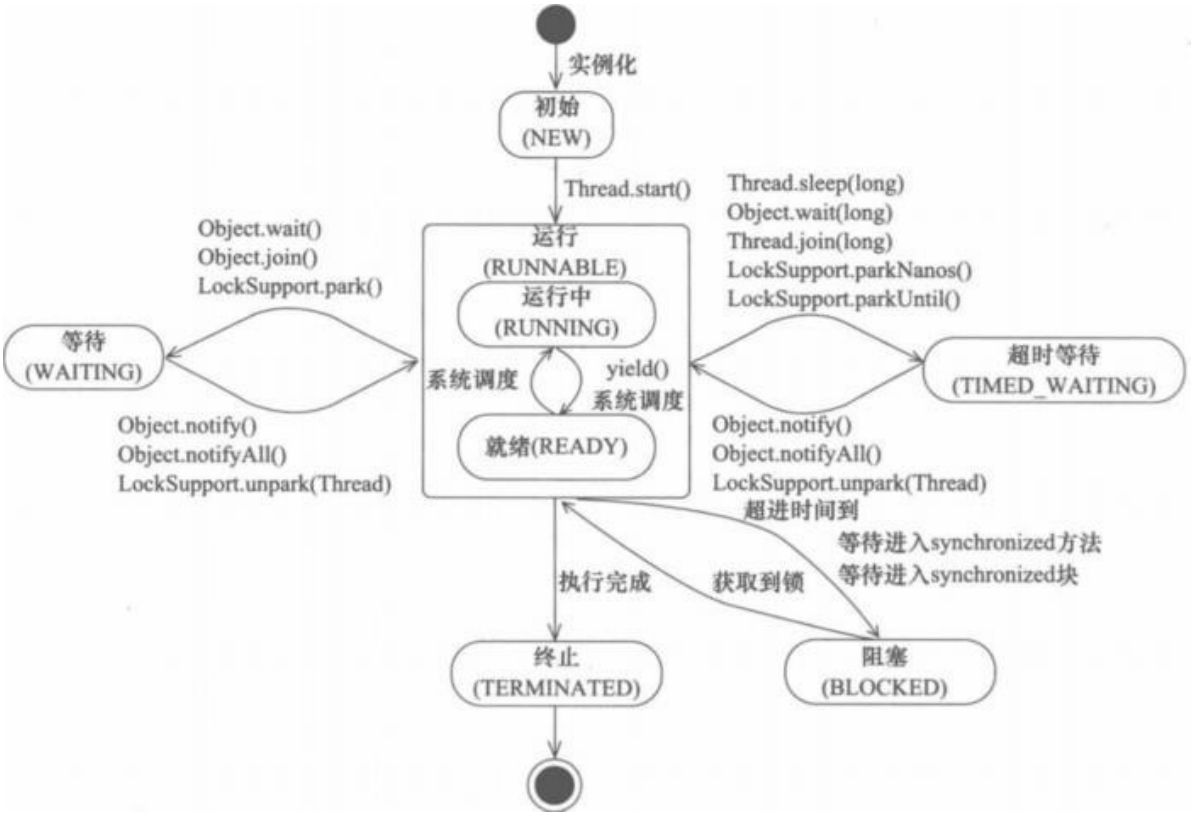
通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完公共资源后，应该立即释放锁。但是，凡事都有一个度，如果对同一个锁不停的进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化 。

锁消除

锁消除是在编译器级别的事情。在即时编译器时，如果发现不可能被共享的对象，则可以消除这些对象的锁操作，多数是因为程序员编码不规范引起。

线程基本方法

线程相关的基本方法有 wait，notify，notifyAll，sleep，join，yield 等。



线程等待（**wait**）

调用该方法的线程进入 WAITING 状态，只有等待另外线程的通知或被中断才会返回，需要注意的是调用 wait()方法后，会释放对象的锁。因此，wait 方法一般用在同步方法或同步代码块中。

线程睡眠（**sleep**）

sleep 导致当前线程休眠，与 wait 方法不同的是 sleep 不会释放当前占有的锁,sleep(long)会导致线程进入 TIMED-WATING 状态，而 wait()方法会导致当前线程进入 WATING 状态

线程让步（**yield**）

yield 会使当前线程让出 CPU 执行时间片，与其他线程一起重新竞争 CPU 时间片。一般情况下，优先级高的线程有更大的可能性成功竞争得到 CPU 时间片，但这又不是绝对的，有的操作系统对线程优先级并不敏感。

线程中断（**interrupt**）

中断一个线程，其本意是给这个线程一个通知信号，会影响这个线程内部的一个中断标识位。这个线程本身并不会因此而改变状态(如阻塞，终止等)。

1. 调用 interrupt()方法并不会中断一个正在运行的线程。也就是说处于 Running 状态的线程并不会因为被中断而被终止，仅仅改变了内部维护的中断标识位而已。
2. 若调用 sleep()而使线程处于 TIMED-WATING 状态，这时调用 interrupt()方法，会抛出

InterruptedException,从而使线程提前结束 TIMED-WATING 状态。

3. 许多声明抛出InterruptedException 的方法(如 Thread.sleep(long mills 方法))，抛出异

常前，都会清除中断标识位，所以抛出异常后，调用 isInterrupted()方法将会返回 false。

4. 中断状态是线程固有的一个标识位，可以通过此标识位安全的终止线程。比如,你想终止一个线程thread的时候，可以调用thread.interrupt()方法，在线程的run方法内部可以根据thread.isInterrupted()的值来优雅的终止线程。

Join 等待其他线程终止

join() 方法，等待其他线程终止，在当前线程中调用一个线程的 join() 方法，则当前线程转为阻塞状态，回到另一个线程结束，当前线程再由阻塞状态变为就绪状态，等待 cpu 的宠幸。

为什么要用 **join()**方法?

很多情况下，主线程生成并启动了子线程，需要用到子线程返回的结果，也就是需要主线程需要在子线程结束后再结束，这时候就要用到 join() 方法。

```
System.out.println(Thread.currentThread().getName() + "线程运行开始!");

    Thread6 thread1 = new Thread6();     thread1.setName("线程 B");
thread1.join();

System.out.println("这时 thread1 执行完毕之后才能执行主线程");
```

线程唤醒（**notify**）

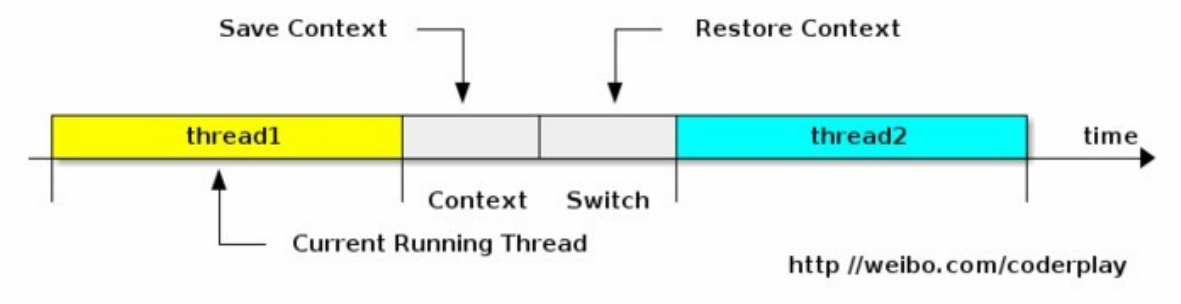
Object 类中的 notify() 方法，唤醒在此对象监视器上等待的单个线程，如果所有线程都在此对象上等待，则会选择唤醒其中一个线程，选择是任意的，并在对实现做出决定时发生，线程通过调用其中一个 wait() 方法，在对象的监视器上等待，直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程，被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争。类似的方法还有 notifyAll() ，唤醒再次监视器上等待的所有线程。

其他方法：

1. sleep()：强迫一个线程睡眠N毫秒。
2. isAlive()： 判断一个线程是否存活。
3. join()： 等待线程终止。
4. activeCount()： 程序中活跃的线程数。
5. enumerate()： 枚举程序中的线程。
6. currentThread()： 得到当前线程。
7. isDaemon()： 一个线程是否为守护线程。
8. setDaemon()： 设置一个线程为守护线程。(用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束)
9. setName()： 为线程设置一个名称。
10. wait()： 强迫一个线程等待。
11. notify()： 通知一个线程继续运行。
12. setPriority()： 设置一个线程的优先级。
13. getPriority()：: 获得一个线程的优先级。

线程上下文切换

巧妙地利用了时间片轮转的方式, CPU 给每个任务都服务一定的时间，然后把当前任务的状态保存下来，在加载下一任务的状态后，继续服务下一任务，任务的状态保存及再加载, 这段过程就叫做上下文切换。时间片轮转的方式使多个任务在同一颗 CPU 上执行变成了可能。



进程

（有时候也称做任务）是指一个程序运行的实例。在 Linux 系统中，线程就是能并行运行并且与他们的父进程（创建他们的进程）共享同一地址空间（一段内存区域）和其他资源的轻量级的进程。

上下文

是指某一时间点 CPU 寄存器和程序计数器的内容。

寄存器

是 CPU 内部的数量较少但是速度很快的内存（与之对应的是 CPU 外部相对较慢的 RAM 主内存）。寄存器通过对常用值（通常是运算的中间值）的快速访问来提高计算机程序运行的速度。

程序计数器

是一个专用的寄存器，用于表明指令序列中 CPU 正在执行的位置，存的值为正在执行的指令的位置或者下一个将要被执行的指令的位置，具体依赖于特定的系统。

PCB-“切换桢”

上下文切换可以认为是内核（操作系统的核心）在 CPU 上对于进程（包括线程）进行切换，上下文切换过程中的信息是保存在进程控制块（PCB, process control block）中的。PCB 还经常被称作“切换桢”（switchframe）。信息会一直保存到 CPU 的内存中，直到他们被再次使用。

上下文切换的活动：

1. 挂起一个进程，将这个进程在 CPU 中的状态（上下文）存储于内存中的某处。
2. 在内存中检索下一个进程的上下文并将其在 CPU 的寄存器中恢复。
3. 跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程在程序中。

引起线程上下文切换的原因

1. 当前执行任务的时间片用完之后，系统 CPU 正常调度下一个任务；
2. 当前执行任务碰到 IO 阻塞，调度器将此任务挂起，继续下一任务；
3. 多个任务抢占锁资源，当前任务没有抢到锁资源，被调度器挂起，继续下一任务；
4. 用户代码挂起当前任务，让出 CPU 时间；
5. 硬件中断；

同步锁与死锁

同步锁

当多个线程同时访问同一个数据时，很容易出现问题。为了避免这种情况出现，我们要保证线程同步互斥，就是指并发执行的多个线程，在同一时间内只允许一个线程访问共享数据。 Java 中可以使用 synchronized 关键字来取得一个对象的同步锁。

死锁

何为死锁，就是多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。

线程池原理

线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量超出数量的线程排队等候，等其它线程执行完毕，再从队列中取出任务来执行。他的主要特点为：线程复用；控制最大并发数；管理线程。

** **线程复用**

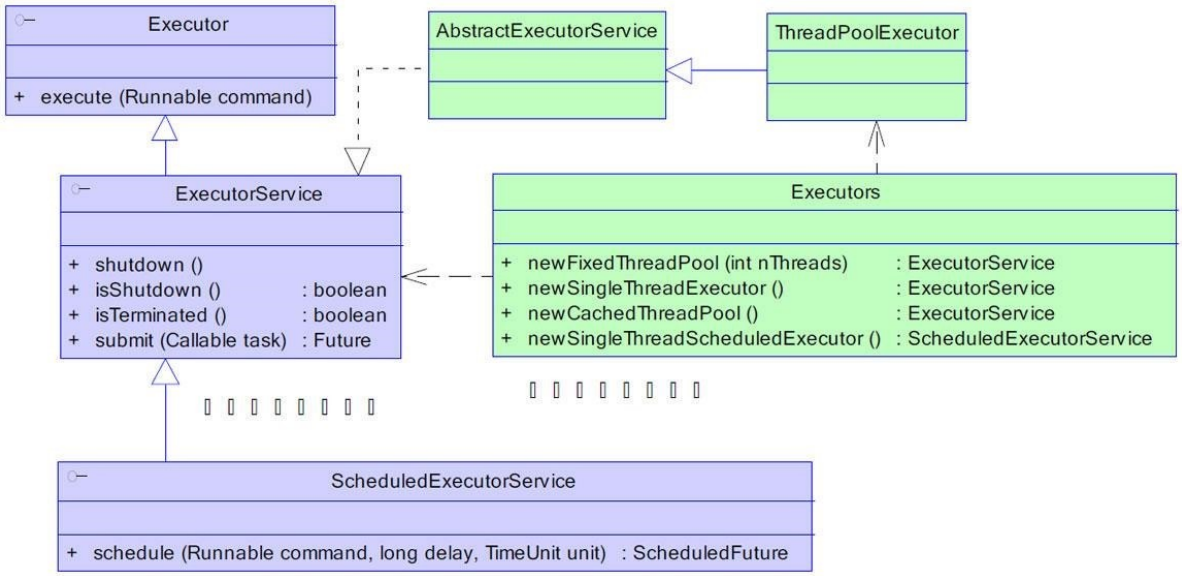
每一个 Thread 的类都有一个 start 方法。 当调用 start 启动线程时 Java 虚拟机会调用该类的 run 方法。 那么该类的 run() 方法中就是调用了 Runnable 对象的 run() 方法。 我们可以继承重写 Thread 类，在其 start 方法中添加不断循环调用传递过来的 Runnable 对象。 这就是线程池的实现原理。循环方法中不断获取 Runnable 是用 Queue 实现的，在获取下一个 Runnable 之前可以是阻塞的。

线程池的组成一般的线程池主要分为以下 4 个组成部分：

1. 线程池管理器：用于创建并管理线程池
2. 工作线程：线程池中的线程
3. 任务接口：每个任务必须实现的接口，用于工作线程调度其运行
4. 任务队列：用于存放待处理的任务，提供一种缓冲机制

Java 中的线程池是通过 Executor 框架实现的，该框架中用到了 Executor，Executors，

ExecutorService，ThreadPoolExecutor ，Callable 和 Future、FutureTask 这几个类。



ThreadPoolExecutor 的构造方法如下：

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue workQueue) {　this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,　Executors.defaultThreadFactory(), defaultHandler);

}

1. corePoolSize：指定了线程池中的线程数量。
2. maximumPoolSize：指定了线程池中的最大线程数量。
3. keepAliveTime：当前线程池数量超过 corePoolSize 时，多余的空闲线程的存活时间，即多次时间内会被销毁。
4. unit：keepAliveTime 的单位。
5. workQueue：任务队列，被提交但尚未被执行的任务。
6. threadFactory：线程工厂，用于创建线程，一般用默认的即可。
7. handler：拒绝策略，当任务太多来不及处理，如何拒绝任务。

拒绝策略

线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列也已经排满了，再也塞不下新任务了。这时候我们就需要拒绝策略机制合理的处理这个问题。

JDK 内置的拒绝策略如下：

1. AbortPolicy： 直接抛出异常，阻止系统正常运行。

2. CallerRunsPolicy： 只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。
3. DiscardOldestPolicy： 丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。
4. DiscardPolicy： 该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢失，这是最好的一种方案。

以上内置拒绝策略均实现了 RejectedExecutionHandler 接口，若以上策略仍无法满足实际需要，完全可以自己扩展 RejectedExecutionHandler 接口。

Java 线程池工作过程

1. 线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面有任务，线程池也不会马上执行它们。
2. 当调用 execute() 方法添加一个任务时，线程池会做如下判断：

   a) 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；

   b) 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；

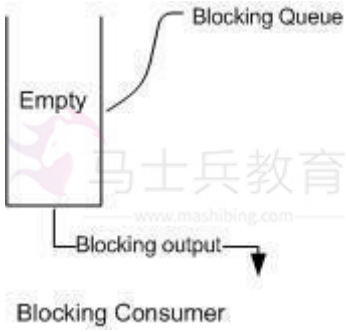
   c) 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；

   d) 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常 RejectExecutionException。
3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

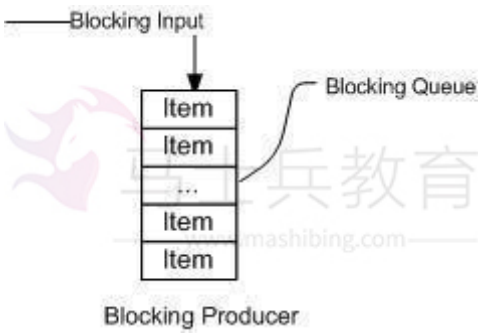
JAVA 阻塞队列原理

阻塞队列，关键字是阻塞，先理解阻塞的含义，在阻塞队列中，线程阻塞有这样的两种情况：

1. 当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。



2. 当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。



阻塞队列的主要方法

方法类型	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除	remove()	poll()	take()	poll(time,unit)
检查	element()	peek()	不可用	不可用

抛出异常：抛出一个异常；

特殊值：返回一个特殊值（null 或 false,视情况而定）

则塞：在成功操作之前，一直阻塞线程

超时：放弃前只在最大的时间内阻塞

插入操作：

1：public abstract boolean add(E paramE)：将指定元素插入此队列中（如果立即可行

且不会违反容量限制），成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。如果该元素是 NULL，则会抛出 NullPointerException 异常。

2：public abstract boolean offer(E paramE)：将指定元素插入此队列中（如果立即可行且不会违反容量限制），成功时返回 true，如果当前没有可用的空间，则返回 false。

3：public abstract void put(E paramE) throws InterruptedException： 将指定元素插入此队列中，将等待可用的空间（如果有必要）

public void put(E paramE) throws InterruptedException { checkNotNull(paramE);

```
ReentrantLock localReentrantLock = this.lock;
localReentrantLock.lockInterruptibly();     try {

  while (this.count == this.items.length)

     this.notFull.await();//如果队列满了，则线程阻塞等待        enqueue(paramE);

  localReentrantLock.unlock();

} finally {       localReentrantLock.unlock();

}
```

}

4：offer(E o, long timeout, TimeUnit unit)：可以设定等待的时间，如果在指定的时间内，还不能往队列中加入 BlockingQueue，则返回失败。 获取数据操作：

1：poll(time):取走 BlockingQueue 里排在首位的对象,若不能立即取出,则可以等 time 参数规定的时间,取不到时返回 null;

2：poll(long timeout, TimeUnit unit)：从 BlockingQueue 取出一个队首的对象，如果在

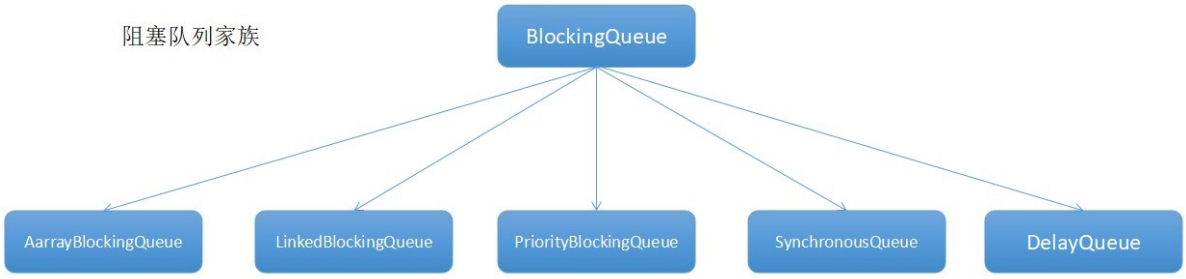
指定时间内，队列一旦有数据可取，则立即返回队列中的数据。否则直到时间超时还没有数据可取，返回失败。

3：take():取走 BlockingQueue 里排在首位的对象,若 BlockingQueue 为空,阻断进入等待状态直到 BlockingQueue 有新的数据被加入。

4.drainTo():一次性从 BlockingQueue 获取所有可用的数据对象（还可以指定获取数据的个数），通过该方法，可以提升获取数据效率；不需要多次分批加锁或释放锁。

Java 中的阻塞队列

1. ArrayBlockingQueue ： 由数组结构组成的有界阻塞队列。
2. LinkedBlockingQueue ： 由链表结构组成的有界阻塞队列。
3. PriorityBlockingQueue ： 支持优先级排序的无界阻塞队列。
4. DelayQueue：使用优先级队列实现的无界阻塞队列。
5. SynchronousQueue：不存储元素的阻塞队列。
6. LinkedTransferQueue：由链表结构组成的无界阻塞队列。
7. LinkedBlockingDeque：由链表结构组成的双向阻塞队列



ArrayBlockingQueue*（公平、非公平）

用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列：

ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(1000,true);

LinkedBlockingQueue*（两个独立锁提高并发）

基于链表的阻塞队列，同ArrayListBlockingQueue类似，此队列按照先进先出（FIFO）的原则对元素进行排序。而 LinkedBlockingQueue 之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

LinkedBlockingQueue 会默认一个类似无限大小的容量（Integer.MAX_VALUE）。

PriorityBlockingQueue*（*compareTo 排序实现优先*）

是一个支持优先级的无界队列。默认情况下元素采取自然顺序升序排列。可以自定义实现 compareTo() 方法来指定元素进行排序规则，或者初始化 PriorityBlockingQueue 时，指定构造参数 Comparator 来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

DelayQueue*（缓存失效、定时任务 ）

是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现 Delayed 接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。我们可以将 DelayQueue 运用在以下应用场景：

1. 缓存系统的设计：可以用 DelayQueue 保存缓存元素的有效期，使用一个线程循环查询

DelayQueue，一旦能从 DelayQueue 中获取元素时，表示缓存有效期到了。

2. 定时任务调度：使用 DelayQueue 保存当天将会执行的任务和执行时间，一旦从

DelayQueue 中获取到任务就开始执行，从比如 TimerQueue 就是使用 DelayQueue 实现的。

SynchronousQueue*（不存储数据、可用于传递数据）

是一个不存储元素的阻塞队列。每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。

SynchronousQueue 可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另 外 一 个 线 程 使 用 ， SynchronousQueue 的 吞 吐 量 高 于 LinkedBlockingQueue 和

ArrayBlockingQueue。

LinkedTransferQueue

是 一 个 由 链 表 结 构 组 成 的 无 界 阻 塞 TransferQueue 队 列 。 相 对 于 其 他 阻 塞 队 列 ，

LinkedTransferQueue 多了 tryTransfer 和 transfer 方法。

1. transfer 方法：如果当前有消费者正在等待接收元素（消费者使用 take()方法或带时间限制的 poll()方法时），transfer 方法可以把生产者传入的元素立刻 transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer 方法会将元素存放在队列的 tail 节点，并等到该元素被消费者消费了才返回。
2. tryTransfer 方法。则是用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回 false。和 transfer 方法的区别是 tryTransfer 方法无论消费者是否接收，方法立即返回。而 transfer 方法是必须等到消费者消费了才返回。

对于带有时间限制的 tryTransfer(E e, long timeout, TimeUnit unit)方法，则是试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回 false，如果在超时时间内消费了元素，则返回 true。

LinkedBlockingDeque

是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。

双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque 多了 addFirst，addLast，offerFirst，offerLast， peekFirst，peekLast 等方法，以 First 单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以 Last 单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法 add 等同于 addLast，移除方法 remove 等效于 removeFirst。但是 take 方法却等同于 takeFirst，不知道是不是 Jdk 的 bug，使用时还是用带有 First 和 Last 后缀的方法更清楚。

在初始化 LinkedBlockingDeque 时可以设置容量防止其过渡膨胀。另外双向阻塞队列可以运用在

“工作窃取”模式中。

CyclicBarrier、CountDownLatch、Semaphore 的用法

CountDownLatch*（线程计数器 ）

CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务 A，它要等待其他 4 个任务执行完毕之后才能执行，此时就可以利用 CountDownLatch 来实现这种功能了。

```
    final CountDownLatch latch = new CountDownLatch(2);           new Thread()
{public void run() {
System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");
```

```
                    Thread.sleep(3000);
                    System.out.println("子线
程"+Thread.currentThread().getName()+"执行完毕");
                    latch.countDown();
};}.start();
          new Thread(){ public void run() {
System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");
                    Thread.sleep(3000);
                    System.out.println("子线
程"+Thread.currentThread().getName()+"执行完毕");
                    latch.countDown();
 };}.start();
            System.out.println("等待 2 个子线程执行完毕...");
            latch.await();
            System.out.println("2 个子线程已经执行完毕");
            System.out.println("继续执行主线程");
           }
```

CyclicBarrier*（回环栅栏-等待至 barrier 状态再全部同时执行）

字面意思回环栅栏，通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier 可以被重用。我们暂且把这个状态就叫做 barrier，当调用 await()方法之后，线程就处于 barrier 了。

CyclicBarrier 中最重要的方法就是 await 方法，它有 2 个重载版本：

1. public int await()：用来挂起当前线程，直至所有线程都到达 barrier 状态再同时执行后续任务；
2. public int await(long timeout, TimeUnit unit)：让这些线程等待至一定的时间，如果还有线程没有到达 barrier 状态就直接让到达 barrier 的线程执行后续任务。

具体使用如下，另外 CyclicBarrier 是可以重用的。

```
        public static void main(String[] args) {         int N = 4;
        CyclicBarrier barrier  = new CyclicBarrier(N);         for(int
i=0;i<N;i++)             new Writer(barrier).start();
    }
    static class Writer extends Thread{         private CyclicBarrier
cyclicBarrier;         public Writer(CyclicBarrier cyclicBarrier) {
this.cyclicBarrier = cyclicBarrier;
        }
        @Override         public void run() {             try {
                Thread.sleep(5000);      //以睡眠来模拟线程需要预定写入数据操作
System.out.println("线程"+Thread.currentThread().getName()+"写入数据完毕，等待其他线程
写入完毕");
                cyclicBarrier.await();
            } catch (InterruptedException e) {
e.printStackTrace();
            }catch(BrokenBarrierException e){
e.printStackTrace();
            }
     System.out.println("所有线程写入完毕，继续处理其他任务，比如数据操作");
        }
    }
```

Semaphore*（信号量-*控制同时访问的线程个数***）**

Semaphore 翻译成字面意思为 信号量，Semaphore 可以控制同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

Semaphore 类中比较重要的几个方法：

1. public void acquire(): 用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许可。
2. public void acquire(int permits):获取 permits 个许可
3. public void release() { } :释放许可。注意，在释放许可之前，必须先获获得许可。
4. public void release(int permits) { }:释放 permits 个许可

上面 4 个方法都会被阻塞，如果想立即得到执行结果，可以使用下面几个方法

1. public boolean tryAcquire():尝试获取一个许可，若获取成功，则立即返回 true，若获取失败，则立即返回 false
2. public boolean tryAcquire(long timeout, TimeUnit unit):尝试获取一个许可，若在指定的时间内获取成功，则立即返回 true，否则则立即返回 false
3. public boolean tryAcquire(int permits):尝试获取 permits 个许可，若获取成功，则立即返回 true，若获取失败，则立即返回 false
4. public boolean tryAcquire(int permits, long timeout, TimeUnit unit): 尝试获取 permits 个许可，若在指定的时间内获取成功，则立即返回 true，否则则立即返回 false
5. 还可以通过 availablePermits()方法得到可用的许可数目。

例子：若一个工厂有5 台机器，但是有8个工人，一台机器同时只能被一个工人使用，只有使用完了，其他工人才能继续使用。那么我们就可以通过 Semaphore 来实现：

```
    int N = 8;            //工人数
        Semaphore semaphore = new Semaphore(5); //机器数目          for(int
i=0;i<N;i++)              new Worker(i,semaphore).start();
    }
    static class Worker extends Thread{          private int num;          private
Semaphore semaphore;          public Worker(int num,Semaphore semaphore){
    this.num = num;              this.semaphore = semaphore;
        }

        @Override          public void run() {              try {
                semaphore.acquire();
                System.out.println("工人"+this.num+"占用一个机器在生产...");
                Thread.sleep(2000);
                System.out.println("工人"+this.num+"释放出机器");
semaphore.release();                          } catch (InterruptedException e) {
              e.printStackTrace();
            }
        }
```

CountDownLatch 和 CyclicBarrier 都能够实现线程之间的等待，只不过它们侧重点不 同；CountDownLatch 一般用于某个线程 A 等待若干个其他线程执行完任务之后，它才

执行；而 CyclicBarrier 一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；另外，CountDownLatch 是不能够重用的，而 CyclicBarrier 是可以重用的。

> Semaphore 其实和锁有点类似，它一般用于控制对某组资源的访问权限。

volatile 关键字的作用（变量可见性、禁止重排序）

Java 语言提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他线程。volatile 变量具备两种特性，volatile 变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取 volatile 类型的变量时总会返回最新写入的值。

变量可见性其一是保证该变量对所有线程可见，这里的可见性指的是当一个线程修改了变量的值，那么新的值对于其他线程是可以立即获取的。

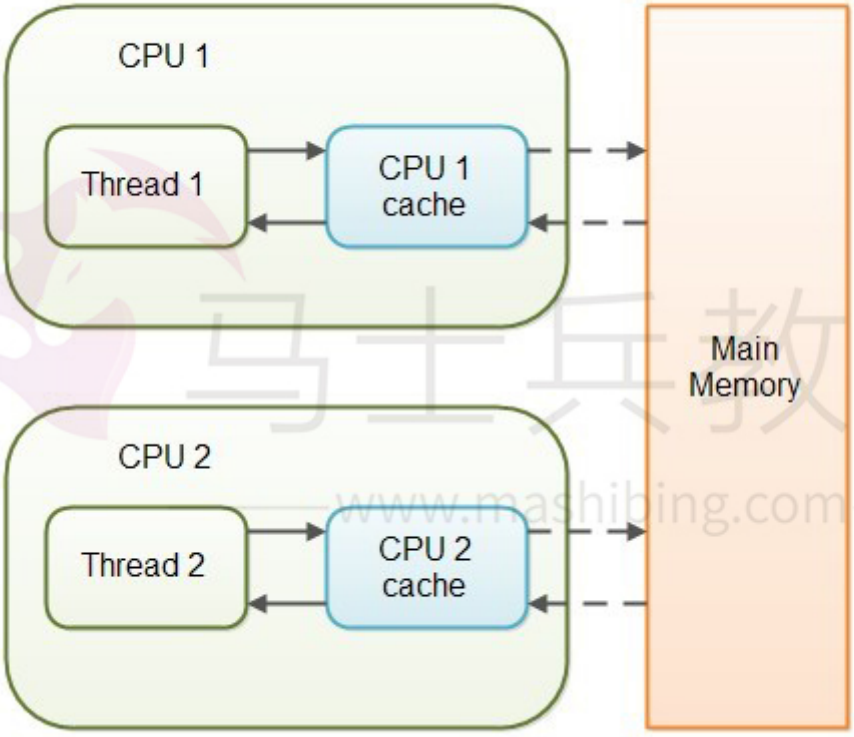
禁止重排序

volatile 禁止了指令重排。

比**sychronized** 更轻量级的同步锁

在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一

种比 sychronized 关键字更轻量级的同步机制。volatile 适合这种场景：一个变量被多个线程共享，线程直接给这个变量赋值。



当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到 CPU 缓存中。如果计算机有多个 CPU，每个线程可能在不同的 CPU 上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。

适用场景值得说明的是对 volatile 变量的单次读/写操作可以保证原子性的，如 long 和 double 类型变量，但是并不能保证 i++这种操作的原子性，因为本质上 i++是读、写两次操作。在某些场景下可以代替 Synchronized。但是,volatile 的不能完全取代 Synchronized 的位置，只有在一些特殊的场景下，才能适用 volatile。总的来说，必须同时满足下面两个条件才能保证在并发环境的线程安全：

（1） 对变量的写操作不依赖于当前值（比如 i++），或者说是单纯的变量赋值（boolean flag = true）。

（2） 该变量没有包含在具有其他变量的不变式中，也就是说，不同的 volatile 变量之间，不能互相依赖。只有在状态真正独立于程序内其他内容时才能使用 volatile。

如何在两个线程之间共享数据

Java 里面进行多线程通信的主要方式就是共享内存的方式，共享内存主要的关注点有两个：可见性和有序性原子性。Java 内存模型（JMM）解决了可见性和有序性的问题，而锁解决了原子性的问题，理想情况下我们希望做到“同步”和“互斥”。有以下常规实现方法：

将数据抽象成一个类，并将数据的操作作为这个类的方法

1. 将数据抽象成一个类，并将对这个数据的操作作为这个类的方法，这么设计可以和容易做到同步，只要在方法上加“synchronized“

```
public class MyData {
     private int j=0; public  synchronized void add(){
         j++;
System.out.println("线程"+Thread.currentThread().getName()+"j 为：“+j);
}
public  synchronized void dec(){
         j--;
       System.out.println("线程"+Thread.currentThread().getName()+"j 为：“+j);
     }
     public int getData(){          return j;
     }
}
public class AddRunnable implements Runnable{
    MyData data;     public AddRunnable(MyData data){         this.data= data;
    }
    public void run() {             data.add();
     }
 }
public class DecRunnable implements Runnable {
    MyData data;     public DecRunnable(MyData data){         this.data = data;
    }
    public void run() {             data.dec();
    }
}
 public static void main(String[] args) {
        MyData data = new MyData();
        Runnable add = new AddRunnable(data);         Runnable dec = new
DecRunnable(data);         for(int i=0;i<2;i++){             new
Thread(add).start();             new Thread(dec).start();
        }
```

Runnable 对象作为一个类的内部类

2. 将 Runnable 对象作为一个类的内部类，共享数据作为这个类的成员变量，每个线程对共享数据的操作方法也封装在外部类，以便实现对数据的各个操作的同步和互斥，作为内部类的各个 Runnable 对象调用外部类的这些方法。

```
public class MyData {
     private int j=0;      public  synchronized void add(){          j++;
     System.out.println("线程"+Thread.currentThread().getName()+"j 为：“+j);
     }
     public  synchronized void dec(){          j--;
      System.out.println("线程"+Thread.currentThread().getName()+"j 为：“+j);
     }
     public int getData(){
         return j;
     }
```

```
}
public class TestThread {
    public static void main(String[] args) {          final MyData data = new
MyData();          for(int i=0;i<2;i++){             new Thread(new Runnable(){
             public void run() {                      data.add();
                 }
             }).start();
            new Thread(new Runnable(){                    public void run() {
              data.dec();
                 }
             }).start();
        }
    }
}
```

ThreadLocal 作用（**线程本地存储**）

ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，ThreadLocal 的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。

ThreadLocalMap（线程的一个属性）

1. 每个线程中都有一个自己的 ThreadLocalMap 类对象，可以将线程自己的对象保持到其中，各管各的，线程可以正确的访问到自己的对象。
2. 将一个共用的 ThreadLocal 静态实例作为 key，将不同对象的引用保存到不同线程的

ThreadLocalMap 中，然后在线程执行的各处通过这个静态 ThreadLocal 实例的 get()方法取得自己线程保存的那个对象，避免了将这个对象作为参数传递的麻烦。

3. ThreadLocalMap 其实就是线程里面的一个属性，它在 Thread 类中定义

ThreadLocal.ThreadLocalMap threadLocals = null;



最常见的 ThreadLocal 使用场景为 用来解决 数据库连接、Session 管理等。

```
private static final ThreadLocal threadSession = new ThreadLocal();
public static Session getSession() throws InfrastructureException {
Session s = (Session) threadSession.get();
try {
if (s == null) {
s = getSessionFactory().openSession();               threadSession.set(s);
        }
    } catch (HibernateException ex) {
    throw new InfrastructureException(ex);
    }
    return s;
}
```

synchronized 和 ReentrantLock 的区别

两者的共同点：

1. 都是用来协调多线程对共享对象、变量的访问
2. 都是可重入锁，同一线程可以多次获得同一个锁
3. 都保证了可见性和互斥性

两者的不同点：

1. ReentrantLock 显示的获得、释放锁，synchronized 隐式获得释放锁
2. ReentrantLock 可响应中断、可轮回，synchronized 是不可以响应中断的，为处理锁的不可用性提供了更高的灵活性
3. ReentrantLock 是 API 级别的，synchronized 是 JVM 级别的
4. ReentrantLock 可以实现公平锁
5. ReentrantLock 通过 Condition 可以绑定多个条件
6. 底层实现不一样， synchronized 是同步阻塞，使用的是悲观并发策略，lock 是同步非阻塞，采用的是乐观并发策略
7. Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现。
8. synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象，因此使用 Lock 时需要在 finally 块中释放锁。
9. Lock 可以让等待锁的线程响应中断，而 synchronized 却不行，使用 synchronized 时，等待的线程会一直等待下去，不能够响应中断。
10. 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。
11. Lock 可以提高多个线程进行读操作的效率，既就是实现读写锁等。

ConcurrentHashMap 并发

减小锁粒度

减小锁粒度是指缩小锁定对象的范围，从而减小锁冲突的可能性，从而提高系统的并发能力。减小锁粒度是一种削弱多线程锁竞争的有效手段，这种技术典型的应用是 ConcurrentHashMap(高性能的 HashMap)类的实现。对于 HashMap 而言，最重要的两个方法是 get 与 set 方法，如果我们对整个 HashMap 加锁，可以得到线程安全的对象，但是加锁粒度太大。Segment 的大小也被称为 ConcurrentHashMap 的并发度。

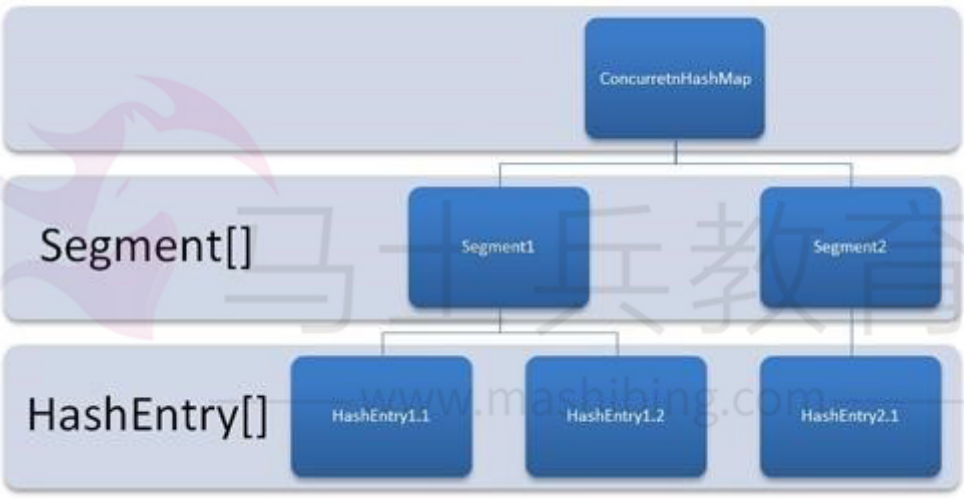
ConcurrentHashMap 分段锁

ConcurrentHashMap，它内部细分了若干个小的 HashMap，称之为段(Segment)。默认情况下一个 ConcurrentHashMap 被进一步细分为 16 个段，既就是锁的并发度。

如果需要在 ConcurrentHashMap 中添加一个新的表项，并不是将整个 HashMap 加锁，而是首先根据hashcode得到该表项应该存放在哪个段中，然后对该段加锁，并完成put操作。在多线程环境中，如果多个线程同时进行put操作，只要被加入的表项不存放在同一个段中，则线程间可以做到真正的并行。

ConcurrentHashMap 是由**Segment** 数组结构和**HashEntry** 数组结构组成

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。Segment 是一种可重入锁 ReentrantLock，在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构， 一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素， 每个 Segment 守护一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。



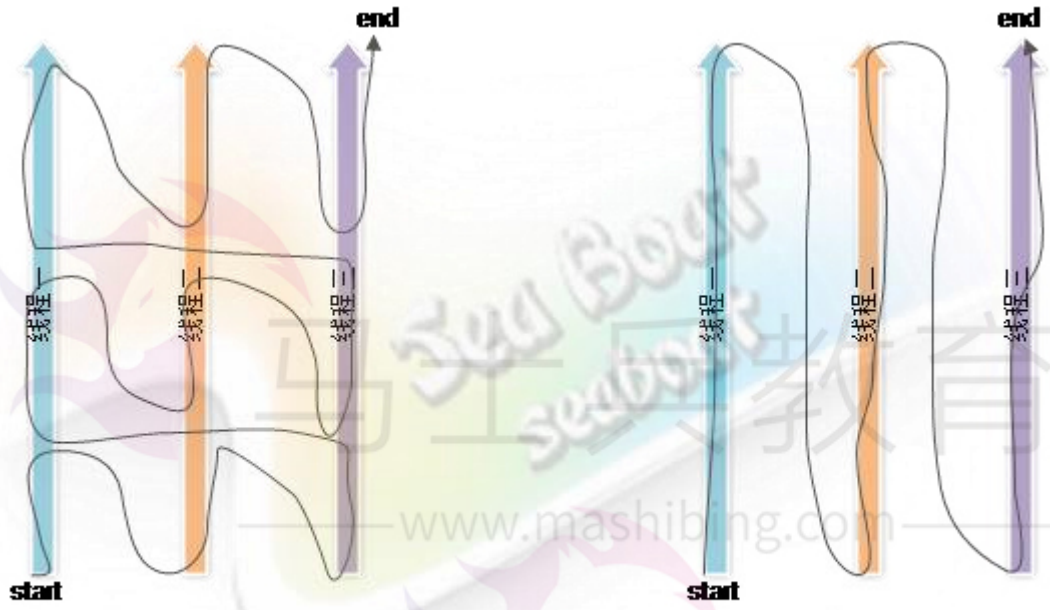
Java 中用到的线程调度

抢占式调度：

抢占式调度指的是每条线程执行的时间、线程的切换都由系统控制，系统控制指的是在系统某种运行机制下，可能每条线程都分同样的执行时间片，也可能是某些线程执行的时间片较长，甚至某些线程得不到执行的时间片。在这种机制下，一个线程的堵塞不会导致整个进程堵塞。

协同式调度：

协同式调度指某一线程执行完后主动通知系统切换到另一线程上执行，这种模式就像接力赛一样，一个人跑完自己的路程就把接力棒交接给下一个人，下个人继续往下跑。线程的执行时间由线程本身控制，线程切换可以预知，不存在多线程同步问题，但它有一个致命弱点：如果一个线程编写有问题，运行到一半就一直堵塞，那么可能导致整个系统崩溃。



抢占式线程调度

JVM 的线程调度实现（抢占式调度）

java 使用的线程调使用抢占式调度，Java 中线程会按优先级分配 CPU 时间片运行，且优先级越高越优先执行，但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间。

线程让出 cpu 的情况：

1. 当前运行线程主动放弃 CPU，JVM 暂时放弃 CPU 操作（基于时间片轮转调度的 JVM 操作系统不会让线程永久放弃 CPU，或者说放弃本次时间片的执行权），例如调用 yield()方法。
2. 当前运行线程因为某些原因进入阻塞状态，例如阻塞在 I/O 上。
3. 当前运行线程结束，即运行完 run()方法里面的任务。

进程调度算法

优先调度算法

1. 先来先服务调度算法（FCFS）

   当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用 FCFS 算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机，特点是：算法比较简单，可以实现基本上的公平。 2. 短作业(进程)优先调度算法

短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。该算法未照顾紧迫型作业。

高优先权优先调度算法

> 为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。

当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程。

1. 非抢占式优先权算法

在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2. 抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程) 的执行，重新将处理机分配给新到的优先权最高的进程。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

2. 高响应比优先调度算法

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：

$$R_{\mathrm{P}}=\frac{\text{等待时间}+\text{要求服务时间}}{\text{要求服务时间}}=\frac{\text{响应时间}}{\text{要求服务时间}}$$

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

基于时间片的轮转调度算法

\1. 时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。

2. 多级反馈队列调度算法

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第 i+1 个队列的时间片要比第 i 个队列的时间片长一倍。

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按 FCFS 原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按 FCFS 原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第 n 队列后，在第 n 队列便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第 1～(i-1)队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时，又有新进程进入优先权较高的队列(第 1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第 i 队列的末尾，把处理机分配给新到的高优先权进程。

在多级反馈队列调度算法中，如果规定第一个队列的时间片略大于多数人机交互所需之处理时间时，便能够较好的满足各种类型用户的需要。

什么是 CAS（比较并交换-乐观锁机制-锁自旋）

概念及特性

CAS（Compare And Swap/Set）比较并交换，CAS 算法的过程是这样：它包含 3 个参数 CAS(V,E,N)。V 表示要更新的变量(内存值)，E 表示预期值(旧的)，N 表示新值。当且仅当 V 值等于 E 值时，才会将 V 的值设为 N，如果 V 值和 E 值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS 返回当前 V 的真实值。

CAS 操作是抱着乐观的态度进行的(乐观锁)，它总是认为自己可以成功完成操作。当多个线程同时使用 CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，

CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

原子包 java.util.concurrent.atomic（锁自旋）

JDK1.5 的原子包：java.util.concurrent.atomic 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等

到该方法执行完成，才由 JVM 从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。

相对于对于 synchronized 这种阻塞算法，CAS 是非阻塞算法的一种常见实现。由于一般 CPU 切换时间比 CPU 指令集操作更加长， 所以 J.U.C 在性能上有了很大的提升。如下代码：

```
public class AtomicInteger extends Number implements java.io.Serializable {
        private volatile int value;
        public final int get() {
        return value;
     }
     public final int getAndIncrement() {
        for (;;) {
        //CAS 自旋，一直尝试，直达成功
           int current = get();
           int next = current + 1;
           if (compareAndSet(current, next))                    return current;

        }
    }
    public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
}
```

getAndIncrement 采用了 CAS 操作，每次从内存中读取数据然后将此数据和+1 后的结果进行 CAS 操作，如果成功就返回结果，否则重试直到成功为止。而 compareAndSet 利用 JNI 来完成CPU 指令的操作。

```
cmpxchg
/*
accumulator = AL, AX, or EAX, depending on whether
a byte, word, or doubleword comparison is being performed
*/
        更新的变量和旧的预期值是否相等
if(accumulator == Destination) {
    ZF = 1; 设置跳转标识
    Destination = Source;
}          原始数据设置到目标里面去
else {
    ZF = 0;  不设置值了
    accumulator = Destination;
}
```

马士兵教育 www.mashibing.com

ABA 问题

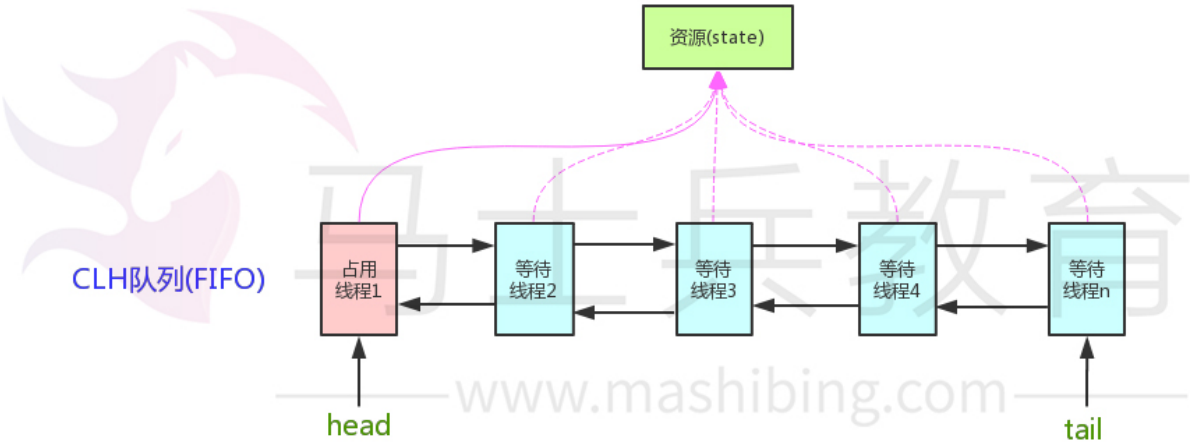
CAS 会导致“ABA 问题”。CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。

部分乐观锁的实现是通过版本号（version）的方式来解决 ABA 问题，乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问题，因为版本号只会增加不会减少。

什么是 AQS（抽象的队列同步器）

AbstractQueuedSynchronizer 类如其名，抽象的队列式的同步器，AQS 定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的 ReentrantLock/Semaphore/CountDownLatch。



它维护了一个 volatile int state（代表共享资源）和一个 FIFO 线程等待队列（多线程争用资源被阻塞时会进入此队列）。这里 volatile 是核心关键词，具体 volatile 的语义，在此不述。state 的访问方式有三种:

getState() setState()

compareAndSetState()

AQS 定义两种资源共享方式

Exclusive独占资源-ReentrantLock

Exclusive（独占，只有一个线程能执行，如 ReentrantLock）

Share共享资源-Semaphore/CountDownLatch

Share（共享，多个线程可同时执行，如 Semaphore/CountDownLatch）。

AQS只是一个框架，具体资源的获取/释放方式交由自定义同步器去实现，AQS这里只定义了一个接口，具体资源的获取交由自定义同步器去实现了（通过state的get/set/CAS)之所以没有定义成 abstract，是因为独占模式下只用实现 tryAcquire-tryRelease，而共享模式下只用实现 tryAcquireShared-tryReleaseShared。如果都定义成abstract，那么每个模式也要去实现另一模式下的接口。不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/ 唤醒出队等），AQS 已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

1． isHeldExclusively()：该线程是否正在独占资源。只有用到 condition 才需要去实现它。

2． tryAcquire(int)：独占方式。尝试获取资源，成功则返回 true，失败则返回 false。

3． tryRelease(int)：独占方式。尝试释放资源，成功则返回 true，失败则返回 false。

4． tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

5． tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回 true，否则返回 false。

同步器的实现是**ABS**核心（**state**资源状态计数）

同步器的实现是 ABS 核心，以 ReentrantLock 为例，state 初始化为 0，表示未锁定状态。A 线程 lock()时，会调用 tryAcquire()独占该锁并将 state+1。此后，其他线程再 tryAcquire()时就会失败，直到 A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的（state 会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证 state 是能回到零态的。

以 CountDownLatch 以例，任务分为 N 个子线程去执行，state 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 countDown()一次，state 会 CAS 减1。等到所有子线程都执行完后(即state=0)，会 unpark()主调用线程，然后主调用线程就会从 await()函数返回，继续后余动作。

ReentrantReadWriteLock 实现独占和共享两种方式

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 tryAcquiretryRelease、tryAcquireShared-tryReleaseShared 中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如 ReentrantReadWriteLock。

NIO是什么？适用于何种场景?

（New IO）为所有的原始类型（boolean类型除外）提供缓存支持的数据容器，使用它可以提 供非阻塞式的高伸缩性网络。

特性：I/O多路复用 + 非阻塞式I/O

NIO适用场景

服务器需要支持超大量的长时间连接。比如10000个连接以上，并且每个客户端并不会频繁地发 送太多数据。例如总公司的一个中心服务器需要收集全国便利店各个收银机的交易信息，只需要 少量线程按需处理维护的大量长期连接。

Jetty、Mina、Netty、ZooKeeper等都是基于NIO方式实现。

NIO技术概览

NIO（Non-blocking I/O，在Java领域，也称为New I/O），是一种同步非阻塞的I/O模型，也是I/O多路复用的基础，已经被越来越多地应用到大型应用服务器，成为解决高并发与大量连接、I/O处理问题的有效方式。

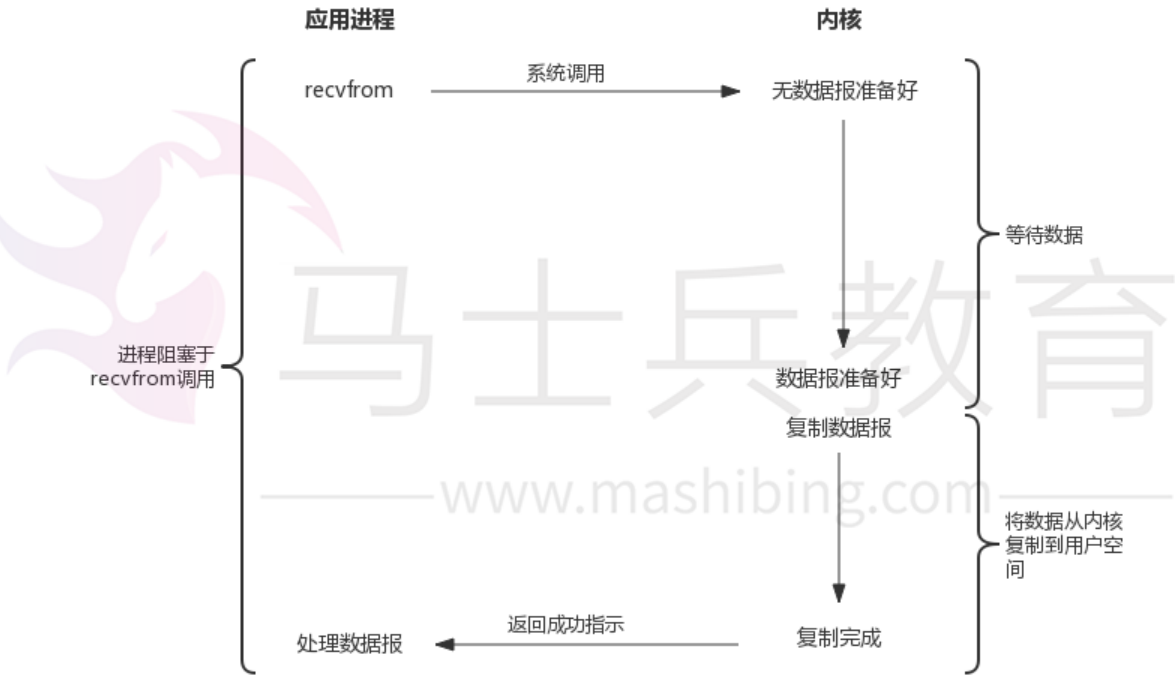
IO模型的分类

按照《Unix网络编程》的划分，I/O模型可以分为：阻塞I/O模型、非阻塞I/O模型、I/O复用模型、信号驱动式I/O模型和异步I/O模型，按照POSIX标准来划分只分为两类：同步I/O和异步I/O。

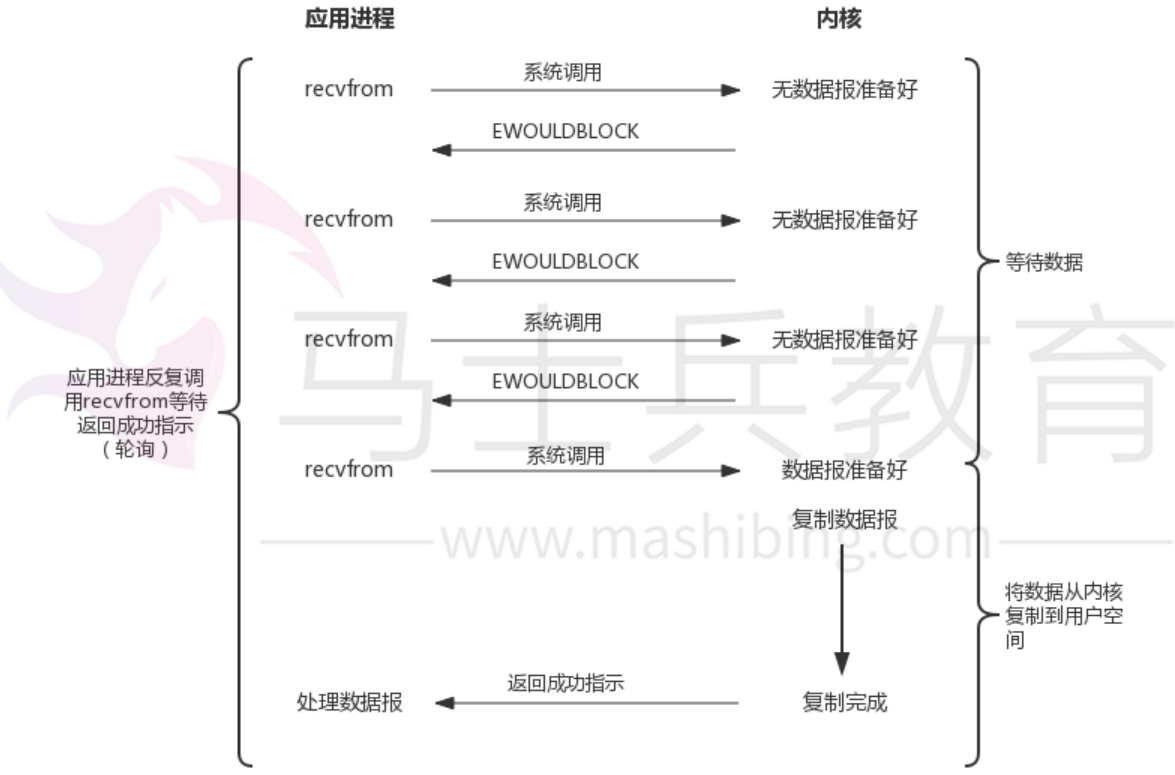
如何区分呢？首先一个I/O操作其实分成了两个步骤：**发起IO请求和实际的IO操作。**同步I/O和异步I/O的区别就在于第二个步骤是否阻塞，如果实际的I/O读写阻塞请求进程，那么就是同步I/O，因此阻塞I/O、非阻塞I/O、I/O复用、信号驱动I/O都是同步I/O，如果不阻塞，而是操作系统帮你做完I/O操作再将结果返回给你，那么就是异步I/O。

阻塞I/O和非阻塞I/O的区别在于第一步，发起I/O请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞I/O，如果不阻塞，那么就是非阻塞I/O。

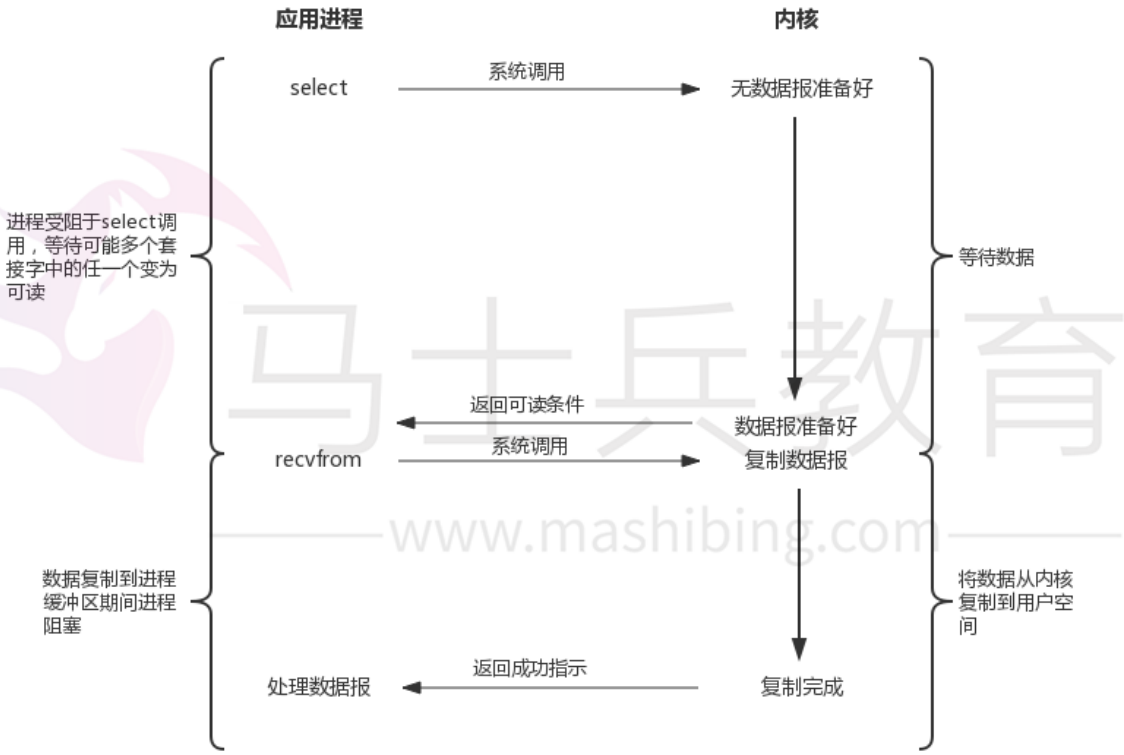
- 阻塞I/O模型 ： 在linux中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：



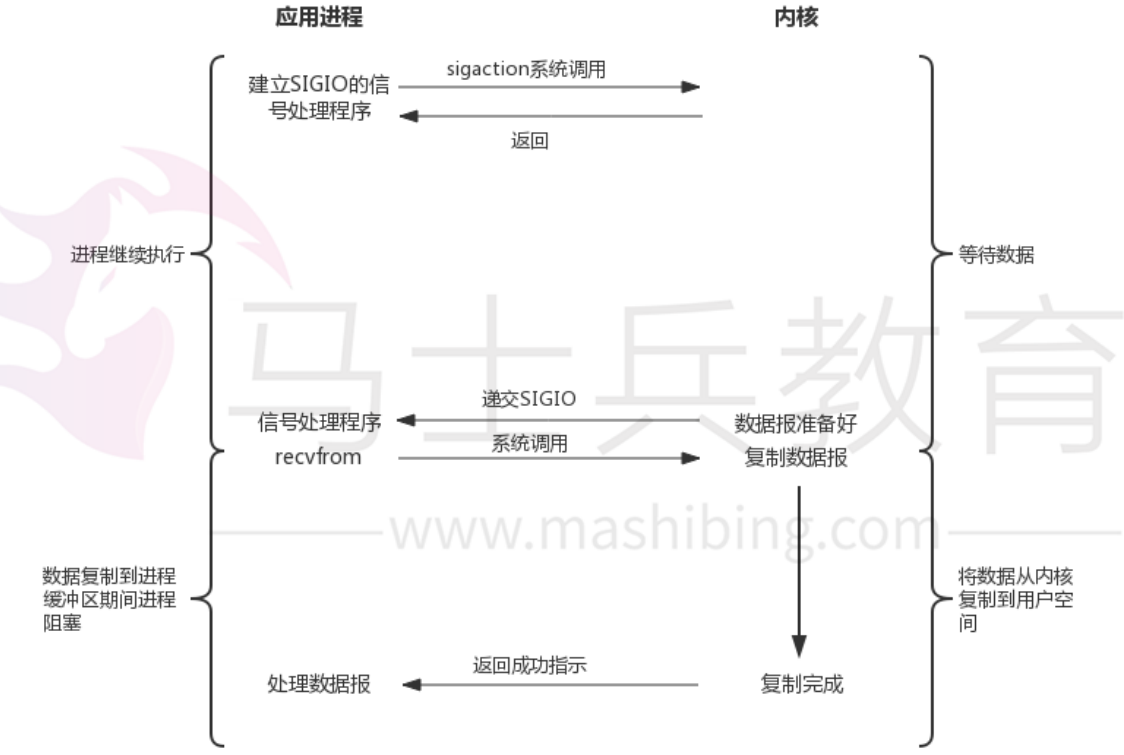
- 非阻塞I/O模型：linux下，可以通过设置socket使其变为non-blocking。当对一个non-blocking socket执行读操作时，流程是这个样子：



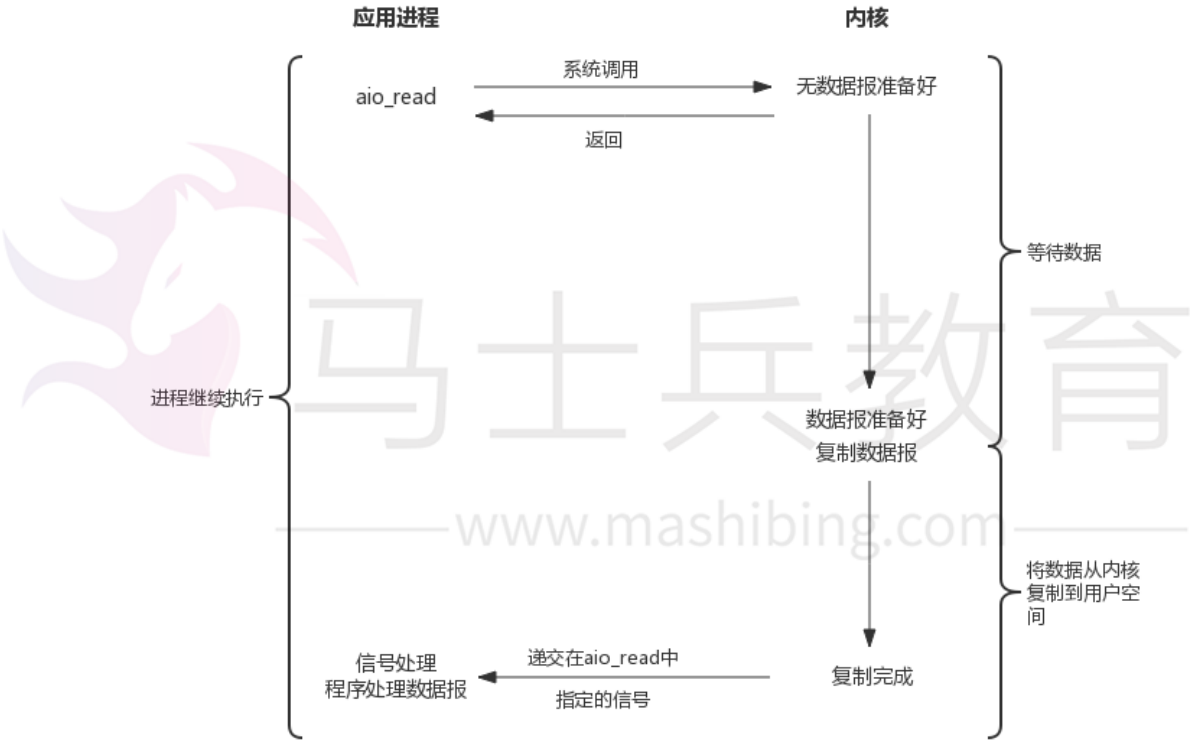
- I/O复用模型：我们可以调用`select`或`poll`，阻塞在这两个系统调用中的某一个之上，而不是真正的IO系统调用上：



- 信号驱动式I/O模型：我们可以用信号，让内核在描述符就绪时发送SIGIO信号通知我们：



- 异步I/O模型：用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个asynchronousread之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，内核会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，内核会给用户进程发送一个signal，告诉它read操作完成了：



以上参考自：《UNIX网络编程》

从前面 I/O 模型的分类中，我们可以看出 AIO 的动机。阻塞模型需要在 I/O 操作开始时阻塞应用程序。这意味着不可能同时重叠进行处理和 I/O 操作。非阻塞模型允许处理和 I/O 操作重叠进行，但是这需要应用程序来检查 I/O 操作的状态。对于异步I/O ，它允许处理和 I/O 操作重叠进行，包括 I/O 操作完成的通知。除了需要阻塞之外，select 函数所提供的功能（异步阻塞 I/O）与 AIO 类似。不过，它是对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。

参考下知乎上的回答：

> - **同步与异步**：同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)。所谓同步，就是在发出一个**调用**时，在没有得到结果之前，该**调用**就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由**调用者**主动等待这个**调用**的结果；
> - **阻塞与非阻塞**：阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回；而非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

两种IO多路复用方案：Reactor和Proactor

一般地，I/O多路复用机制都依赖于一个**事件多路分离器(Event Demultiplexer)**。分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write**事件处理器(Event Handler)**。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。

两个与事件分离器有关的模式是Reactor和Proactor。Reactor模式采用同步I/O，而Proactor采用异步I/O。在Reactor中，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

而在Proactor模式中，处理器或者兼任处理器的事件分离器，只负责发起异步读写操作。I/O操作本身由操作系统来完成。传递给操作系统的参数需要包括用户定义的数据缓冲区地址和数据大小，操作系统才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获I/O操作完成事件，然后将事件传递给对应处理器。比如，在windows上，处理器发起一个异步I/O操作，再由事件分离器等待

IOCompletion事件。典型的异步模式实现，都建立在操作系统支持异步API的基础之上，我们将这种实现称为“系统级”异步或“真”异步，因为应用程序完全依赖操作系统执行真正的I/O工作。

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（写操作类似）。

在Reactor中实现读：

- 注册读就绪事件和相应的事件处理器；
- 事件分离器等待事件；
- 事件到来，激活分离器，分离器调用事件对应的处理器；
- 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

在Proactor中实现读：

- 处理器发起异步读操作（注意：操作系统必须支持异步I/O）。在这种情况下，处理器无视I/O就绪事件，它关注的是完成事件；
- 事件分离器等待操作完成事件；
- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成；
- 事件分离器呼唤处理器；
- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

可以看出，两个模式的相同点，都是对某个I/O事件的事件通知（即告诉某个模块，这个I/O操作可以进行或已经完成）。在结构上，两者的相同点和不同点如下：

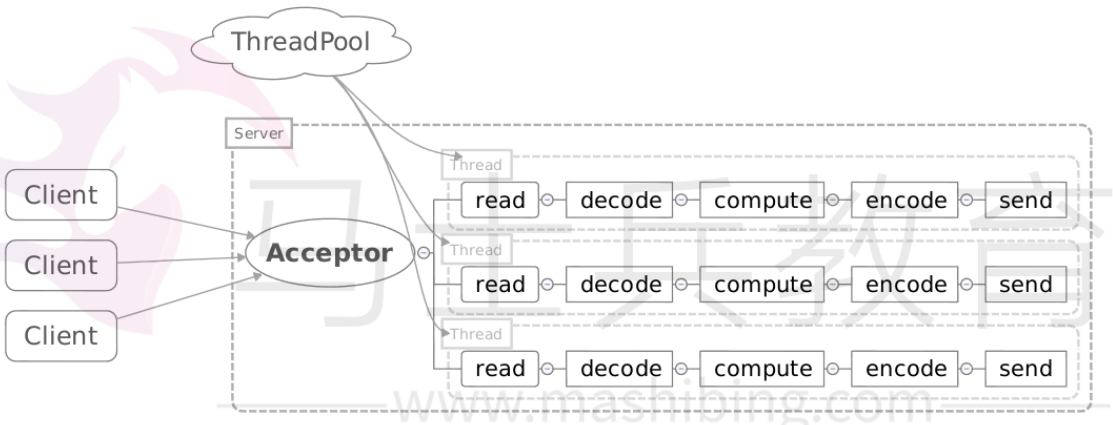
- 相同点：demultiplexor负责提交I/O操作（异步）、查询设备是否可操作（同步），然后当条件满足时，就回调handler；
- 不同点：异步情况下（Proactor），当回调handler时，表示I/O操作已经完成；同步情况下（Reactor），回调handler时，表示I/O设备可以进行某个操作（can read or can write）。

传统BIO模型

BIO是同步阻塞式IO，通常在while循环中服务端会调用accept方法等待接收客户端的连接请求，一旦接收到一个连接请求，就可以建立通信套接字在这个通信套接字上进行读写操作，此时不能再接收其他客户端连接请求，只能等待同当前连接的客户端的操作执行完成。

如果BIO要能够同时处理多个客户端请求，就必须使用多线程，即每次accept阻塞等待来自客户端请求，一旦受到连接请求就建立通信套接字同时开启一个新的线程来处理这个套接字的数据读写请求，然后立刻又继续accept等待其他客户端连接请求，即为每一个客户端连接请求都创建一个线程来单独处理。

我们看下传统的BIO方式下的编程模型大致如下：




```
public class BIODemo {
```

```
    public static void main(String[] args) throws IOException {

        ExecutorService executor = Executors.newFixedThreadPool(128);

        ServerSocket serverSocket = new ServerSocket();
        serverSocket.bind(new InetSocketAddress(1234));
        // 循环等待新连接
        while (true) {
            Socket socket = serverSocket.accept();
            // 为新的连接创建线程执行任务
            executor.submit(new ConnectionTask(socket));
        }
    }

}

class ConnectionTask extends Thread {
    private Socket socket;

    public ConnectionTask(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        while (true) {
            InputStream inputStream = null;
            OutputStream outputStream = null;
            try {
                inputStream = socket.getInputStream();

                // read from socket...

                inputStream.read();

                outputStream = socket.getOutputStream();

                // write to socket...

                outputStream.write();

            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                // 关闭资源...

            }

        }
    }
}
```

这里之所以使用多线程，是因为socket.accept()、inputStream.read()、outputStream.write()都是同步阻塞的，当一个连接在处理I/O的时候，系统是阻塞的，如果是单线程的话在阻塞的期间不能接受任何请求。所以，使用多线程，就可以让CPU去处理更多的事情。其实这也是所有使用多线程的本质：

- 利用多核。

- 当I/O阻塞系统，但CPU空闲的时候，可以利用多线程使用CPU资源。

使用线程池能够让线程的创建和回收成本相对较低。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的I/O并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池可以缓冲一些过多的连接或请求。

但这个模型最本质的问题在于，严重依赖于线程。但线程是很"贵"的资源，主要表现在：

1. 线程的创建和销毁成本很高，在Linux这样的操作系统中，线程本质上就是一个进程。创建和销毁都是重量级的系统函数；
2. 线程本身占用较大内存，像Java的线程栈，一般至少分配512K～1M的空间，如果系统中的线程数过千，恐怕整个JVM的内存都会被吃掉一半；
3. 线程的切换成本是很高的。操作系统发生线程切换的时候，需要保留线程的上下文，然后执行系统调用。如果线程数过高，可能执行线程切换的时间甚至会大于线程执行的时间，这时候带来的表现往往是系统load偏高、CPU sy使用率特别高（超过20%以上），导致系统几乎陷入不可用的状态；
4. 容易造成锯齿状的系统负载。因为系统负载是用活动线程数或CPU核心数，一旦线程数量高但外部网络环境不是很稳定，就很容易造成大量请求的结果同时返回，激活大量阻塞线程从而使系统负载压力过大。

所以，当面对十万甚至百万级连接的时候，传统的BIO模型是无能为力的。随着移动端应用的兴起和各种网络游戏的盛行，百万级长连接日趋普遍，此时，必然需要一种更高效的I/O处理模型。

NIO的实现原理

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题，即在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知应用程序进行处理，应用再将流读取到缓冲区或写入操作系统。

也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

下面看下代码的实现：

NIO服务端代码（新建连接）：

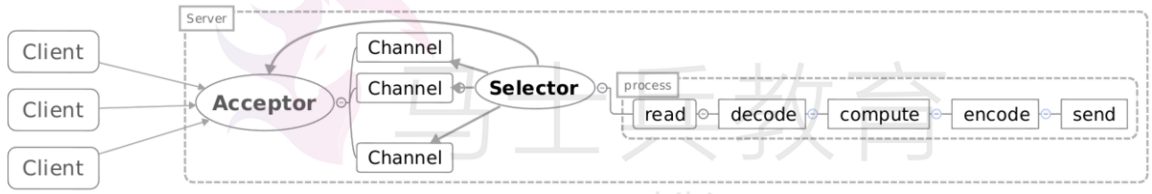
```
//获取一个ServerSocket通道
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
serverChannel.socket().bind(new InetSocketAddress(port));
//获取通道管理器
selector = Selector.open();
//将通道管理器与通道绑定，并为该通道注册SelectionKey.OP_ACCEPT事件，
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

NIO服务端代码（监听）：

```
while(true){
    //当有注册的事件到达时，方法返回，否则阻塞。
   selector.select();
   for(SelectionKey key : selector.selectedKeys()){
       if(key.isAcceptable()){
```

```
            ServerSocketChannel server =
                 (ServerSocketChannel)key.channel();
            SocketChannel channel = server.accept();
            channel.write(ByteBuffer.wrap(
             new String("send message to client").getBytes()));
            //在与客户端连接成功后，为客户端通道注册SelectionKey.OP_READ事件。
            channel.register(selector, SelectionKey.OP_READ);
        }else if(key.isReadable()){//有可读数据事件
            SocketChannel channel = (SocketChannel)key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(10);
            int read = channel.read(buffer);
            byte[] data = buffer.array();
            String message = new String(data);
            System.out.println("receive message from client, size:"
                + buffer.position() + " msg: " + message);
        }
    }
}
```

NIO模型示例如下：



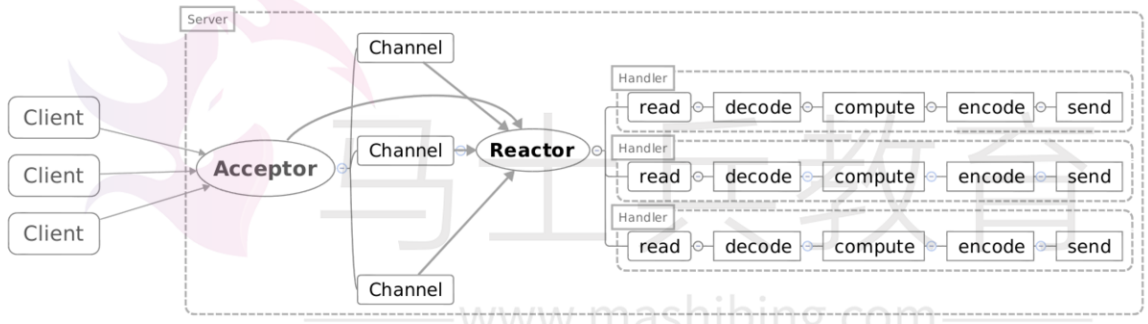
- Acceptor注册Selector，监听accept事件；
- 当客户端连接后，触发accept事件；
- 服务器构建对应的Channel，并在其上注册Selector，监听读写事件；
- 当发生读写事件后，进行相应的读写处理。

Reactor模型

有关Reactor模型结构，可以参考Doug Lea在 Scalable IO in Java中的介绍。这里简单介绍一下Reactor模式的典型实现：

Reactor单线程模型

这是最简单的单Reactor单线程模型。Reactor线程负责多路分离套接字、accept新连接，并分派请求到处理器链中。该模型适用于处理器链中业务处理组件能快速完成的场景。不过，这种单线程模型不能充分利用多核资源，所以实际使用的不多。



这个模型和上面的NIO流程很类似，只是将消息相关处理独立到了Handler中去了。

代码实现如下：

```
public class Reactor implements Runnable {

    final Selector selector;

    final ServerSocketChannel serverSocketChannel;

    public static void main(String[] args) throws IOException {
        new Thread(new Reactor(1234)).start();
    }

    public Reactor(int port) throws IOException {
        selector = Selector.open();
        serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new InetSocketAddress(port));
        serverSocketChannel.configureBlocking(false);
        SelectionKey key = serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
        key.attach(new Acceptor());
    }

    @Override
    public void run() {
        while (!Thread.interrupted()) {
            try {
                selector.select();
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                for (SelectionKey selectionKey : selectionKeys) {
                    dispatch(selectionKey);
                }
                selectionKeys.clear();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }

    private void dispatch(SelectionKey selectionKey) {
        Runnable run = (Runnable) selectionKey.attachment();
        if (run != null) {
            run.run();
        }
    }

    class Acceptor implements Runnable {

        @Override
        public void run() {
            try {
                SocketChannel channel = serverSocketChannel.accept();
                if (channel != null) {
                    new Handler(selector, channel);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        }

    }


}

class Handler implements Runnable {

    private final static int DEFAULT_SIZE = 1024;

    private final SocketChannel socketChannel;

    private final SelectionKey seletionKey;

    private static final int READING = 0;

    private static final int SENDING = 1;

    private int state = READING;

    ByteBuffer inputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    ByteBuffer outputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    public Handler(Selector selector, SocketChannel channel) throws IOException
{
        this.socketChannel = channel;
        socketChannel.configureBlocking(false);
        this.seletionKey = socketChannel.register(selector, 0);
        seletionKey.attach(this);
        seletionKey.interestOps(SelectionKey.OP_READ);
        selector.wakeup();
    }

    @Override
    public void run() {
        if (state == READING) {
            read();
        } else if (state == SENDING) {
            write();
        }

    }

    class Sender implements Runnable {

        @Override
        public void run() {
            try {
                socketChannel.write(outputBuffer);
            } catch (IOException e) {
                e.printStackTrace();
            }
            if (outIsComplete()) {
                seletionKey.cancel();
            }
        }
```

```
    }

    private void write() {
        try {
            socketChannel.write(outputBuffer);
        } catch (IOException e) {
            e.printStackTrace();
        }
        while (outIsComplete()) {
            seletionKey.cancel();
        }

    }

    private void read() {
        try {
            socketChannel.read(inputBuffer);
            if (inputIsComplete()) {
                process();
                System.out.println("接收到来自客户端（" +
socketChannel.socket().getInetAddress().getHostAddress()
                        + "）的消息：" + new String(inputBuffer.array()));
                seletionKey.attach(new Sender());
                seletionKey.interestOps(SelectionKey.OP_WRITE);
                seletionKey.selector().wakeup();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }


    }

    public boolean inputIsComplete() {
        return true;
    }

    public boolean outIsComplete() {
        return true;

    }

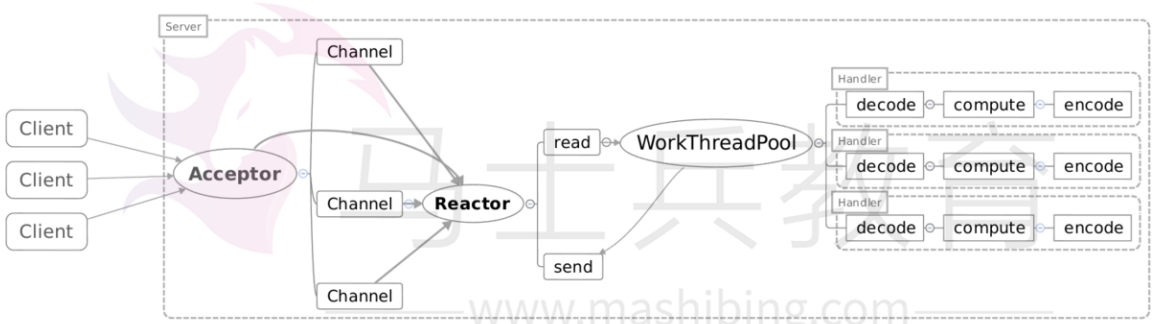
    public void process() {
        // do something...
    }

}
```

虽然上面说到NIO一个线程就可以支持所有的IO处理。但是瓶颈也是显而易见的。我们看一个客户端的情况，如果这个客户端多次进行请求，如果在Handler中的处理速度较慢，那么后续的客户端请求都会被积压，导致响应变慢！所以引入了Reactor多线程模型。

Reactor多线程模型

相比上一种模型，该模型在处理器链部分采用了多线程（线程池）：



Reactor多线程模型就是将Handler中的IO操作和非IO操作分开，操作IO的线程称为IO线程，非IO操作的线程称为工作线程。这样的话，客户端的请求会直接被丢到线程池中，客户端发送请求就不会堵塞。

可以将Handler做如下修改：

```
class Handler implements Runnable {

    private final static int DEFAULT_SIZE = 1024;

    private final SocketChannel socketChannel;

    private final SelectionKey seletionKey;

    private static final int READING = 0;

    private static final int SENDING = 1;

    private int state = READING;

    ByteBuffer inputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    ByteBuffer outputBuffer = ByteBuffer.allocate(DEFAULT_SIZE);

    private Selector selector;

    private static ExecutorService executorService =
Executors.newFixedThreadPool(Runtime.getRuntime()
            .availableProcessors());
    private static final int PROCESSING = 3;

    public Handler(Selector selector, SocketChannel channel) throws IOException
{
        this.selector = selector;
        this.socketChannel = channel;
        socketChannel.configureBlocking(false);
        this.seletionKey = socketChannel.register(selector, 0);
        seletionKey.attach(this);
        seletionKey.interestOps(SelectionKey.OP_READ);
        selector.wakeup();
    }

    @Override
    public void run() {
        if (state == READING) {
```

```
            read();
        } else if (state == SENDING) {
            write();
        }

    }

    class Sender implements Runnable {

        @Override
        public void run() {
            try {
                socketChannel.write(outputBuffer);
            } catch (IOException e) {
                e.printStackTrace();
            }
            if (outIsComplete()) {
                seletionKey.cancel();
            }
        }

    }

    private void write() {
        try {
            socketChannel.write(outputBuffer);
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (outIsComplete()) {
            seletionKey.cancel();
        }

    }

    private void read() {
        try {
            socketChannel.read(inputBuffer);
            if (inputIsComplete()) {
                process();
                executorService.execute(new Processer());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }


    }

    public boolean inputIsComplete() {
        return true;
    }

    public boolean outIsComplete() {
        return true;

    }
```

```
    public void process() {

    }

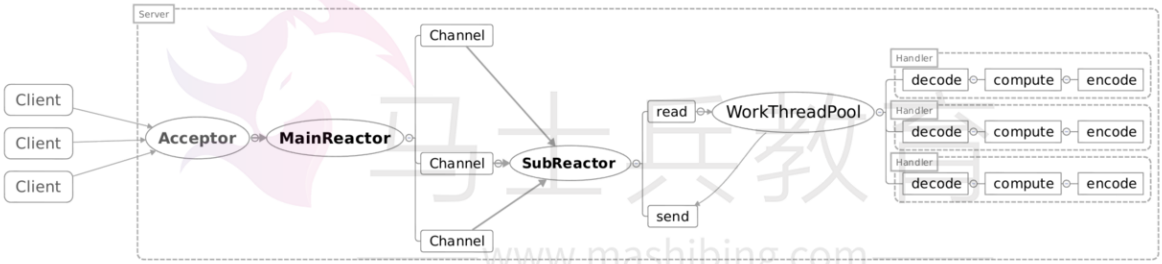
    synchronized void processAndHandOff() {
        process();
        state = SENDING; // or rebind attachment
        seletionKey.interestOps(SelectionKey.OP_WRITE);
        selector.wakeup();
    }
    class Processer implements Runnable {
        public void run() {
            processAndHandOff();
        }
    }

}
```

但是当用户进一步增加的时候，Reactor会出现瓶颈！因为Reactor既要处理IO操作请求，又要响应连接请求。为了分担Reactor的负担，所以引入了主从Reactor模型。

主从Reactor多线程模型

主从Reactor多线程模型是将Reactor分成两部分，mainReactor负责监听server socket，accept新连接，并将建立的socket分派给subReactor。subReactor负责多路分离已连接的socket，读写网络数据，对业务处理功能，其扔给worker线程池完成。通常，subReactor个数上可与CPU个数等同：



这时可以把Reactor做如下修改：

```
public class Reactor {


    final ServerSocketChannel serverSocketChannel;

    Selector[] selectors; // also create threads
    AtomicInteger next = new AtomicInteger(0);
    ExecutorService sunReactors =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

    public static void main(String[] args) throws IOException {
        new Reactor(1234);
    }

    public Reactor(int port) throws IOException {
        serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new InetSocketAddress(port));
        serverSocketChannel.configureBlocking(false);
        selectors = new Selector[4];
        for (int i = 0; i < 4; i++) {
```

```
            Selector selector = Selector.open();
            selectors[i] = selector;
            SelectionKey key = serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
            key.attach(new Acceptor());
            new Thread(() -> {
                while (!Thread.interrupted()) {
                    try {
                        selector.select();
                        Set<SelectionKey> selectionKeys =
selector.selectedKeys();
                        for (SelectionKey selectionKey : selectionKeys) {
                            dispatch(selectionKey);
                        }
                        selectionKeys.clear();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }).start();
        }

    }



    private void dispatch(SelectionKey selectionKey) {
        Runnable run = (Runnable) selectionKey.attachment();
        if (run != null) {
            run.run();
        }
    }

    class Acceptor implements Runnable {

        @Override
        public void run() {
            try {
                SocketChannel connection = serverSocketChannel.accept();
                if (connection != null)
                    sunReactors.execute(new
Handler(selectors[next.getAndIncrement() % selectors.length], connection));
            } catch (IOException e) {
                e.printStackTrace();
            }

        }

    }


}
```

可见，主Reactor用于响应连接请求，从Reactor用于处理IO操作请求。

AIO

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，
对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；
对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。
即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。
在JDK1.7中，这部分内容被称作NIO.2，主要在java.nio.channels包下增加了下面四个异步通道：

- AsynchronousSocketChannel
- AsynchronousServerSocketChannel
- AsynchronousFileChannel
- AsynchronousDatagramChannel

我们看一下AsynchronousSocketChannel中的几个方法：

```
public abstract class AsynchronousSocketChannel
    implements AsynchronousByteChannel, NetworkChannel
{
    public abstract Future<Integer> read(ByteBuffer dst);

    public abstract <A> void read(ByteBuffer[] dsts,
                                  int offset,
                                  int length,
                                  long timeout,
                                  TimeUnit unit,
                                  A attachment,
                                  CompletionHandler<Long,? super A> handler);

    public abstract <A> void write(ByteBuffer src,
                                   long timeout,
                                   TimeUnit unit,
                                   A attachment,
                                   CompletionHandler<Integer,? super A>
handler);

    public final <A> void write(ByteBuffer src,
                                A attachment,
                                CompletionHandler<Integer,? super A> handler)

    {
        write(src, 0L, TimeUnit.MILLISECONDS, attachment, handler);
    }

    public abstract Future<Integer> write(ByteBuffer src);

    public abstract <A> void write(ByteBuffer[] srcs,
                                   int offset,
                                   int length,
                                   long timeout,
                                   TimeUnit unit,
                                   A attachment,
                                   CompletionHandler<Long,? super A> handler);
}
```

其中的read/write方法，有的会返回一个`Future`对象，有的需要传入一个`CompletionHandler`对象，该对象的作用是当执行完读取/写入操作后，直接该对象当中的方法进行回调。

对于`AsynchronousSocketChannel`而言，在windows和linux上的实现类是不一样的。

在windows上，AIO的实现是通过IOCP来完成的，实现类是：

```
WindowsAsynchronousSocketChannelImpl
```

实现的接口是：

```
Iocp.OverlappedChannel
```

而在linux上，实现类是：

```
UnixAsynchronousSocketChannelImpl
```

实现的接口是：

```
Port.PollableChannel
```

AIO是一种接口标准，各家操作系统可以实现也可以不实现。在不同操作系统上在高并发情况下最好都采用操作系统推荐的方式。Linux上还没有真正实现网络方式的AIO。

select和epoll的区别

当需要读两个以上的I/O的时候，如果使用阻塞式的I/O，那么可能长时间的阻塞在一个描述符上面，另外的描述符虽然有数据但是不能读出来，这样实时性不能满足要求，大概的解决方案有以下几种：

1. 使用多进程或者多线程，但是这种方法会造成程序的复杂，而且对与进程与线程的创建维护也需要很多的开销（Apache服务器是用的子进程的方式，优点可以隔离用户）；
2. 用一个进程，但是使用非阻塞的I/O读取数据，当一个I/O不可读的时候立刻返回，检查下一个是否可读，这种形式的循环为轮询（polling），这种方法比较浪费CPU时间，因为大多数时间是不可读，但是仍花费时间不断反复执行read系统调用；
3. 异步I/O，当一个描述符准备好的时候用一个信号告诉进程，但是由于信号个数有限，多个描述符时不适用；
4. 一种较好的方式为I/O多路复用，先构造一张有关描述符的列表（epoll中为队列），然后调用一个函数，直到这些描述符中的一个准备好时才返回，返回时告诉进程哪些I/O就绪。select和epoll这两个机制都是多路I/O机制的解决方案，select为POSIX标准中的，而epoll为Linux所特有的。

它们的区别主要有三点：

1. select的句柄数目受限，在linux/posix_types.h头文件有这样的声明：`#define __FD_SETSIZE 1024` 表示select最多同时监听1024个fd。而epoll没有，它的限制是最大的打开文件句柄数目；
2. epoll的最大好处是不会随着FD的数目增长而降低效率，在selec中采用轮询处理，其中的数据结构类似一个数组的数据结构，而epoll是维护一个队列，直接看队列是不是空就可以了。epoll只会对”活跃”的socket进行操作—这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有”活跃”的socket才会主动的去调用 callback函数（把这个句柄加入队列），其他idle状态句柄则不会，在这点上，epoll实现了一个”伪”AIO。但是如果绝大部分的I/O都是“活跃的”，每个I/O端口使用率很高的话，epoll效率不一定比select高（可能是要维护队列复杂）；
3. 使用mmap加速内核与用户空间的消息传递。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。

NIO与epoll

上文说到了select与epoll的区别，再总结一下Java NIO与select和epoll：

- Linux2.6之后支持epoll
- windows支持select而不支持epoll
- 不同系统下nio的实现是不一样的，包括Sunos linux 和windows
- select的复杂度为O(N)
- select有最大fd限制，默认为1024
- 修改sys/select.h可以改变select的fd数量限制
- epoll的事件模型，无fd数量限制，复杂度O(1),不需要遍历fd

以下代码基于Java 8。

下面看下在NIO中Selector的open方法：

```
public static Selector open() throws IOException {
    return SelectorProvider.provider().openSelector();
}
```

这里使用了SelectorProvider去创建一个Selector，看下provider方法的实现：

```
public static SelectorProvider provider() {
    synchronized (lock) {
        if (provider != null)
            return provider;
        return AccessController.doPrivileged(
            new PrivilegedAction<SelectorProvider>() {
                public SelectorProvider run() {
                        if (loadProviderFromProperty())
                            return provider;
                        if (loadProviderAsService())
                            return provider;
                        provider = sun.nio.ch.DefaultSelectorProvider.create();
                        return provider;
                    }
                });
    }
}
```

看下`sun.nio.ch.DefaultSelectorProvider.create()`方法，该方法在不同的操作系统中的代码是不同的，在windows中的实现如下：

```
public static SelectorProvider create() {
    return new WindowsSelectorProvider();
}
```

在Mac OS中的实现如下：

```
public static SelectorProvider create() {
    return new KQueueSelectorProvider();
}
```

在linux中的实现如下：

```
public static SelectorProvider create() {
    String str = (String)AccessController.doPrivileged(new
GetPropertyAction("os.name"));

    if (str.equals("SunOS"))
        return createProvider("sun.nio.ch.DevPollSelectorProvider");
    if (str.equals("Linux"))
        return createProvider("sun.nio.ch.EPollSelectorProvider");
    return new PollSelectorProvider();
}
```

我们看到create方法中是通过区分操作系统来返回不同的Provider的。其中SunOs就是Solaris返回的是DevPollSelectorProvider，对于Linux，返回的Provder是EPollSelectorProvider，其余操作系统，返回的是PollSelectorProvider。

Zero Copy

许多web应用都会向用户提供大量的静态内容，这意味着有很多数据从硬盘读出之后，会原封不动的通过socket传输给用户。

这种操作看起来可能不会怎么消耗CPU，但是实际上它是低效的：

1. kernel把从disk读数据；
2. 将数据传输给application；
3. application再次把同样的内容再传回给处于kernel级的socket。

这种场景下，application实际上只是作为一种低效的中间介质，用来把磁盘文件的数据传给socket。

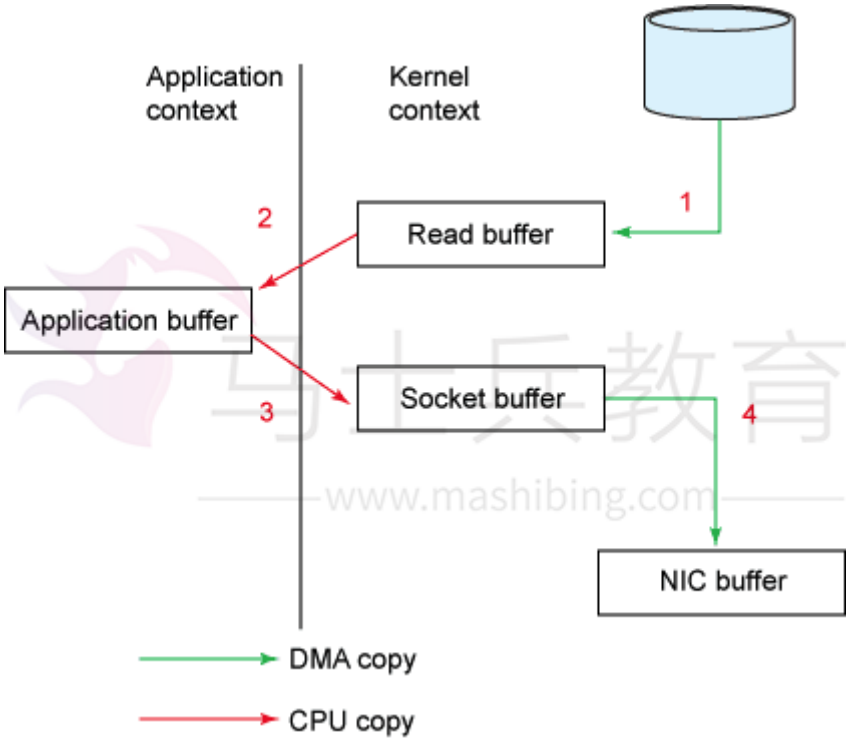
数据每次传输都会经过user和kernel空间都会被copy，这会消耗cpu，并且占用RAM的带宽。

传统的数据传输方式

像这种从文件读取数据然后将数据通过网络传输给其他的程序的方式其核心操作就是如下两个调用：

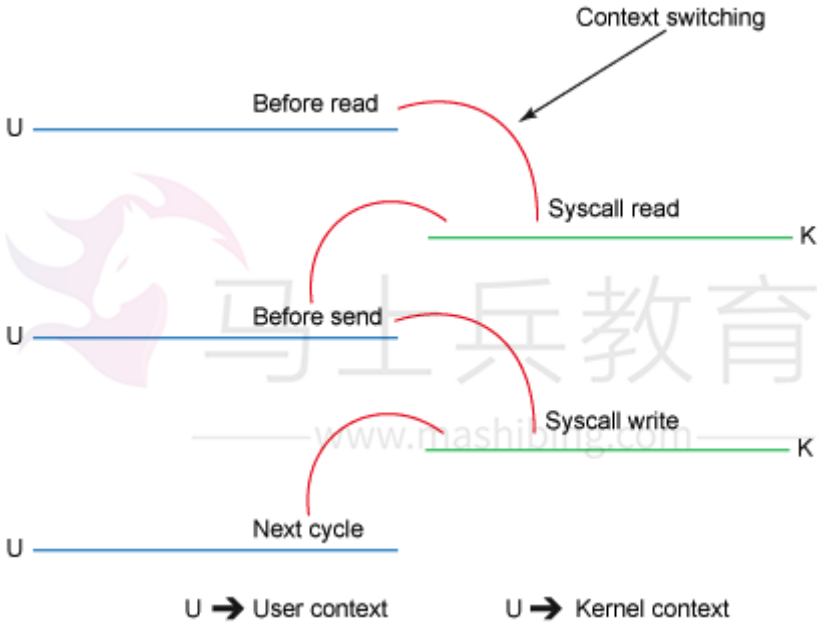
```
File.read(fileDesc,buf,len);
Socket.send(socket,buf,len);
```

其上操作看上去只有两个简单的调用，但是其内部过程却要经历四次用户态和内核态的切换以及四次的数据复制操作：



上图展示了数据从文件到socket的内部流程。

下面看下用户态和内核态的切换过程：



步骤如下：

1. read()的调用引起了从用户态到内核态的切换（看图二），内部是通过sys_read()（或者类似的方法）发起对文件数据的读取。数据的第一次复制是通过DMA（直接内存访问）将磁盘上的数据复制到内核空间的缓冲区中；
2. 数据从内核空间的缓冲区复制到用户空间的缓冲区后，read()方法也就返回了。此时内核态又切换回用户态，现在数据也已经复制到了用户地址空间的缓存中；
3. socket的send()方法的调用又会引起用户态到内核的切换，第三次数据复制又将数据从用户空间缓冲区复制到了内核空间的缓冲区，这次数据被放在了不同于之前的内核缓冲区中，这个缓冲区与数据将要被传输到的socket关联；
4. send()系统调用返回后，就产生了第四次用户态和内核态的切换。随着DMA单独异步的将数据从内核态的缓冲区中传输到协议引擎发送到网络上，有了第四次数据复制。

Zero Copy的数据传输方式

`java.nio.channels.FileChannel`中定义了两个方法：transferTo( )和 transferFrom( )。

transferTo( )和 transferFrom( )方法允许将一个通道交叉连接到另一个通道，而不需要通过一个中间缓冲区来传递数据。只有 FileChannel 类有这两个方法，因此 **channel-to-channel** 传输中通道之一必须是 FileChannel。您不能在 socket 通道之间直接传输数据，不过 socket 通道实现 `WritableByteChannel` 和 `ReadableByteChannel` 接口，因此文件的内容可以用 `transferTo( )` 方法传输给一个 socket 通道，或者也可以用 transferFrom( )方法将数据从一个 socket 通道直接读取到一个文件中。

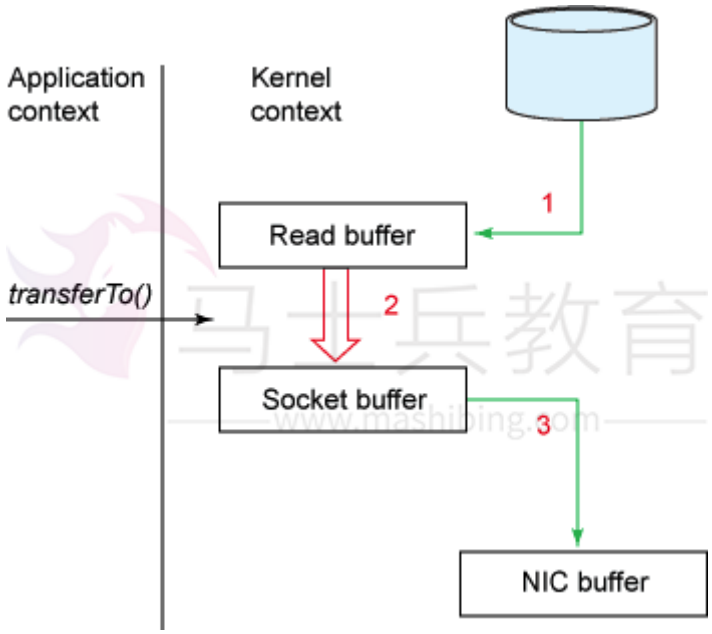
下面根据`transferTo()`方法来说明。

根据上文可知，`transferTo()`方法可以把bytes直接从调用它的channel传输到另一个WritableByteChannel，中间不经过应用程序。

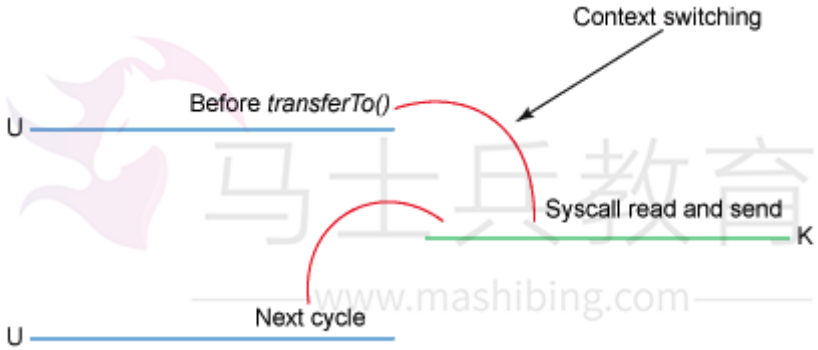
下面看下该方法的定义：

```
public abstract long transferTo(long position, long count,
                                WritableByteChannel target)
        throws IOException;
```

下图展示了通过transferTo实现数据传输的路径：



下图展示了内核态、用户态的切换情况：



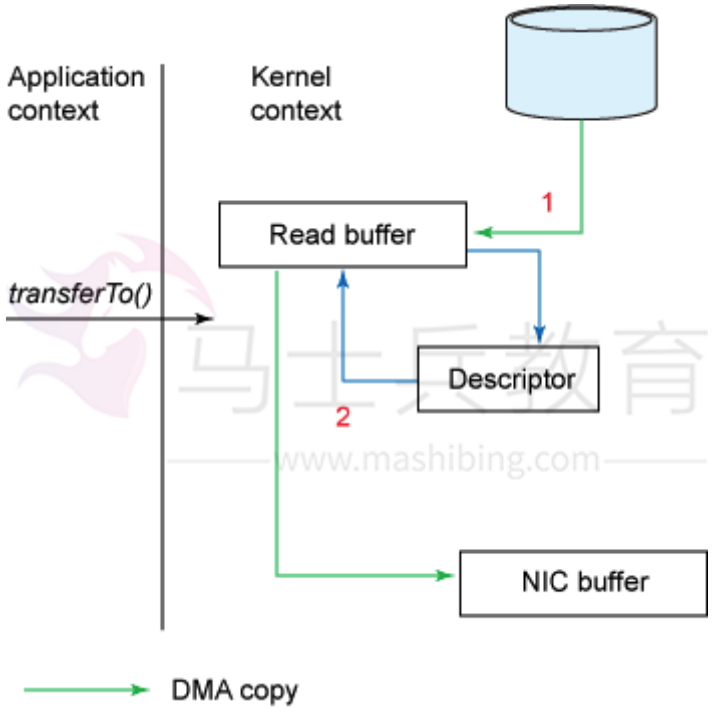
使用transferTo()方式所经历的步骤：

1. transferTo调用会引起DMA将文件内容复制到读缓冲区(内核空间的缓冲区)，然后数据从这个缓冲区复制到另一个与socket输出相关的内核缓冲区中；

2. 第三次数据复制就是DMA把socket关联的缓冲区中的数据复制到协议引擎上发送到网络上。

这次改善，我们是通过将内核、用户态切换的次数从四次减少到两次，将数据的复制次数从四次减少到三次(只有一次用到cpu资源)。但这并没有达到我们零复制的目标。如果底层网络适配器支持收集操作的话，我们可以进一步减少内核对数据的复制次数。在内核为2.4或者以上版本的linux系统上，socket缓冲区描述符将被用来满足这个需求。这个方式不仅减少了内核用户态间的切换，而且也省去了那次需要cpu参与的复制过程。从用户角度来看依旧是调用transferTo()方法，但是其本质发生了变化：

1. 调用transferTo方法后数据被DMA从文件复制到了内核的一个缓冲区中；
2. 数据不再被复制到socket关联的缓冲区中了，仅仅是将一个描述符（包含了数据的位置和长度等信息）追加到socket关联的缓冲区中。DMA直接将内核中的缓冲区中的数据传输给协议引擎，消除了仅剩的一次需要cpu周期的数据复制。



NIO存在的问题

使用NIO != 高性能，当连接数<1000，并发程度不高或者局域网环境下NIO并没有显著的性能优势。

NIO并没有完全屏蔽平台差异，它仍然是基于各个操作系统的I/O系统实现的，差异仍然存在。使用NIO做网络编程构建事件驱动模型并不容易，陷阱重重。

推荐大家使用成熟的NIO框架，如Netty，MINA等。解决了很多NIO的陷阱，并屏蔽了操作系统的差异，有较好的性能和编程模型。

总结

最后总结一下NIO有哪些优势：

- 事件驱动模型
- 避免多线程
- 单线程处理多任务
- 非阻塞I/O，I/O读写不再阻塞
- 基于block的传输，通常比基于流的传输更高效
- 更高级的IO函数，Zero Copy
- I/O多路复用大大提高了Java网络应用的可伸缩性和实用性

- **Mysql（连老师）**
  - 数据库基础知识
    - 为什么要使用数据库
      - 数据保存在内存
      - 数据保存在文件
      - 数据保存在数据库
    - 什么是SQL?
    - 什么是MySQL?
    - 数据库三大范式是什么
    - mysql有关权限的表都有哪几个
    - MySQL的binlog有有几种录入格式？分别有什么区别?
  - 数据类型
    - mysql有哪些数据类型
  - 引擎
    - MySQL存储引擎MyISAM与InnoDB区别
    - **MyISAM索引与InnoDB索引的区别?**
    - **InnoDB引擎的4大特性**
    - 存储引擎选择
  - 索引
    - 什么是索引?
    - 索引有哪些优缺点?
    - 索引使用场景（重点）
    - 索引有哪几种类型?
    - 索引的数据结构（b树，hash）
      - 索引的基本原理
      - 索引算法有哪些?
      - 索引设计的原则?
      - 创建索引的原则（重中之重）
      - 创建索引时需要注意什么?
      - 使用索引查询一定能提高查询的性能吗？为什么
    - 百万级别或以上的数据如何删除
    - 前缀索引
    - 什么是最左前缀原则？什么是最左匹配原则
    - B树和B+树的区别
    - 使用B树的好处
    - 使用B+树的好处
    - Hash索引和B+树所有有什么区别或者说优劣呢?
    - 数据库为什么使用B+树而不是B树
    - B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据
    - 什么是聚簇索引？何时使用聚簇索引与非聚簇索引
    - 非聚簇索引一定会回表查询吗?
    - 联合索引是什么？为什么需要注意联合索引中的顺序?
  - 事务
    - 什么是数据库事务?
    - 事物的四大特性(ACID)介绍一下?
    - 什么是脏读？幻读？不可重复读?
    - 什么是事务的隔离级别？MySQL的默认隔离级别是什么?
  - 锁
    - 对MySQL的锁了解吗
    - 隔离级别与锁的关系
    - 按照锁的粒度分数据库锁有哪些？锁机制与InnoDB锁算法
    - 从锁的类别上分MySQL都有哪些锁呢?
    - 像上面那样子进行锁定岂不是有点阻碍并发效率了
    - MySQL中InnoDB引擎的行锁是怎么实现的?
    - InnoDB存储引擎的锁的算法有三种

什么是死锁？怎么解决？
数据库的乐观锁和悲观锁是什么？怎么实现的？
视图
为什么要使用视图？什么是视图？
视图有哪些特点？
视图的使用场景有哪些？
视图的优点
视图的缺点
什么是游标？
存储过程与函数
什么是存储过程？有哪些优缺点？
触发器
什么是触发器？触发器的使用场景有哪些？
MySQL中都有哪些触发器？
常用SQL语句
SQL语句主要分为哪几类
超键、候选键、主键、外键分别是什么？
SQL约束有哪几种？
六种关联查询
什么是子查询
子查询的三种情况
mysql中in和exists区别
varchar与char的区别
varchar(50)中50的涵义
int(20)中20的涵义
mysql为什么这么设计
mysql中int(10)和char(10)以及varchar(10)的区别
FLOAT和DOUBLE的区别是什么？
drop、delete与truncate的区别
三者都表示删除，但是三者有一些差别：
UNION与UNIONALL的区别？
SQL优化
如何定位及优化SQL语句的性能问题？
创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因？
SQL的生命周期？
大表数据查询，怎么优化
超大分页怎么处理？
mysql分页
慢查询日志
关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？
慢查询都怎么优化过？
为什么要尽量设定一个主键？
主键使用自增ID还是UUID？
字段为什么要求定义为not null？
如果要存储用户的密码散列，应该使用什么字段进行存储？
优化查询过程中的数据访问
优化长难的查询语句
优化特定类型的查询语句
优化关联查询
优化子查询
优化LIMIT分页
优化UNION查询
优化WHERE子句
数据库优化
为什么要优化
数据库结构优化
MySQL数据库cpu飙升到500%的话他怎么处理？
大表怎么优化？某个表有近千万数据，

CRUD比较慢，如何优化？分库分表了是怎么做的？分表分库了有什么问题？有用到中间件么？他们的原理知道么？
垂直分表
适用场景
缺点
水平分表：
适用场景
MySQL的复制原理以及流程
读写分离有哪些解决方案？
备份计划，mysqldump以及xtranbackup的实现原理
数据表损坏的修复方式有哪些？
MySQL记录binlog的方式主要包括三种模式？每种模式的优缺点是什么？
MySQL锁，悲观锁、乐观锁、排它锁、共享锁、表级锁、行级锁；
MySQL InnoDB中，乐观锁、悲观锁、共享锁、排它锁、行锁、表锁、死锁概念的理解
乐观锁
悲观锁
共享锁
排它锁
行锁
表锁
死锁
分布式事务的原理2阶段提交，同步异步阻塞非阻塞；
数据库事务隔离级别，MySQL默认的隔离级别
Spring如何实现事务
spring事务管理(详解和实例)
1 初步理解
2 核心接口
2.1 事务管理器
2.2 基本事务属性的定义
2.3 事务状态
3 编程式事务
3.1 编程式和声明式事务的区别
3.2 如何实现编程式事务?
4 声明式事务
4.1 配置方式
4.2 一个声明式事务的实例
Spring事务管理及几种简单的实现
JDBC如何实现事务
嵌套事务实现
Spring事务管理--(二)嵌套事物详解
一、前言
二、spring嵌套事物
1、展示项目代码--简单测springboot项目
2、外部起事物，内部起事物，内外都无Try Catch
3、外部起事物，内部起事物，外部有Try Catch
4、外部起事物，内部起事物，内部有Try Catch
5、外部起事物，内部起事物，内外有Try Catch
三、嵌套事物总结
四、正确的嵌套事物实例
分布式事务实现；
分布式事务与解决方案
前言
产生原因
数据库分库分表：
应用SOA化：
应用场景
支付、转账：

在线下单：
电商场景：流量充值业务
数据库事务
事务类型：
隔离级别及引发现象：（略谈）
Spring事务传播行为：（略谈）
事务种类：
如何保证强一致性
本地事务（mysql 之 InnoDB）：
分布式事务：
实现分布式事务解决方案
基于XA协议的两阶段提交（2PC）
补偿事务（TCC）
本地消息表（MQ 异步确保）
MQ 事务消息
Sagas 事务模型
其他补偿方式
总结
SQL的整个解析、执行过程原理、SQL行转列；
mysql行转列、列转行
行转列
列转行
红黑树的实现原理和应用场景；
MySql的存储引擎的不同
MySQL存储引擎之Myisam和Innodb总结性梳理
Mysql优化系列--Innodb引擎下mysql自身配置优化
Mysql怎么分表，以及分表后如果想按条件分页查询怎么办(如果不是按分表字段来查询的话，几乎效率低下，无解)
mysql 数据库 分表后 怎么进行分页查询？Mysql分库分表方案?
mysql大数据量使用limit分页，随着页码的增大，查询效率越低下。
测试实验
1. 直接用limit start, count分页语句， 也是我程序中用的方法：
2. 对limit分页问题的性能优化方法
MySQL分表自增ID解决方案
理解分布式id生成算法SnowFlake
概述
Talk is cheap, show you the code
代码理解
负数的二进制表示
用位运算计算n个bit能表示的最大数值
用mask防止溢出
用位运算汇总结果
观察
扩展
MySql的主从实时备份同步的配置，以及原理(从库读主库的binlog)，读写分离
Mysql主从同步的实现原理
MySQL索引背后的数据结构及算法原理
摘要
数据结构及算法基础
索引的本质
B-Tree和B+Tree
B-Tree
B+Tree
带有顺序访问指针的B+Tree
为什么使用B-Tree（B+Tree）
主存存取原理
磁盘存取原理
局部性原理与磁盘预读

B-/+Tree索引的性能分析
MySQL索引实现
MyISAM索引实现
InnoDB索引实现
索引使用策略及优化
示例数据库
最左前缀原理与相关优化
情况一：全列匹配。
情况二：最左前缀匹配。
情况三：查询条件用到了索引中列的精确匹配，但是中间某个条件未提供。
情况四：查询条件没有指定索引第一列。
情况五：匹配某列的前缀字符串。
情况六：范围查询。
情况七：查询条件中含有函数或表达式。
索引选择性与前缀索引
InnoDB的主键选择与插入优化
后记
mysql的锁--行锁，表锁，乐观锁，悲观锁
Mysql中的MVCC
1. Innodb的事务相关概念
2. 行的更新过程
1. 初始数据行
2.事务1更改该行的各字段的值
3.事务2修改该行的值
4. 事务提交
5. Insert Undo log
3. 事务级别
4. MVCC
5.总结
mysql索引原理之聚簇索引
学习笔记_mysql索引原理之B+/-Tree
关系型和非关系型数据库区别
我的Mysql死锁排查过程（案例分析）
死锁起因
分析
阅读死锁日志
死锁形成流程图
拓展
总结
MySql优化
MySQL 对于千万级的大表要怎么优化?
产生死锁的必要条件
操作系统：死锁的产生、条件、和解锁
产生死锁的原因主要是：
产生死锁的四个必要条件：
死锁的解除与预防
死锁排除的方法
Spring（连老师）
Spring 原理
Spring 特点
Spring 核心组件
Spring 常用模块
Spring 主要包
Spring 常用注解
Spring 第三方结合
Spring IOC 原理
Spring APO 原理
Spring MVC 原理

Spring AOP的实现原理和场景；
一、场景
二、实现技术
Spring bean的作用域和生命周期；
作用域
生命周期
Spring 5比Spring4做了哪些改进；
Spring 4.x新特性
Spring 5.x新特性
Spring4新特性——泛型限定式依赖注入
Spring4新特性——**核心容器的其他改进**
Spring4新特性——Web开发的增强
Spring4新特性——集成Bean Validation 1.1(JSR-349)到SpringMVC
1、集成Bean Validation 1.1到SpringMVC
2、分组验证及分组顺序
3、消息中使用EL表达式
4、方法参数/返回值验证
5、自定义验证规则
6、类级别验证器
7、通过脚本验证
8、cross-parameter，跨参数验证
9、混合类级别验证器和跨参数验证器
10、组合验证注解
11、本地化
Spring4新特性——Groovy Bean定义DSL
一、对比
二、Groovy Bean定义
三、Groovy Bean定义 DSL语法
Spring4新特性——更好的Java泛型操作API
Spring4新特性——JSR310日期时间API的支持
Spring4新特性——注解、脚本、任务、MVC等其他特性改进
一、注解方面的改进
二、脚本的支持
三、Future增强
四、MvcUriComponentsBuilder
五、Socket支持
如何自定义一个Spring Boot Starter?
本章内容概览
版本信息
官方资料
设定实战目标
学习spring cloud的starter
如何建立对spring-cloud-netflix-eureka-client模块的依赖
为什么应用能自动注册到Eureka sever
实战的设计
自定义spring boot starter三部曲之二：实战开发
本章内容概述
创建工程customizestarter
创建模块customizeapi
创建模块addservice
创建模块minusservice
创建模块customizeservicestarter
构建工程customizestarter
创建工程customizestartertestdemo
构建工程customizestartertestdemo
验证支持负数的减法服务
验证不支持负数的减法服务
自定义spring boot starter三部曲之三：源码分析spring.factories加载过程

版本情况
初步分析
spring容器如何处理配置类
spring boot配置类的加载情况
Spring IOC是什么？优点是什么？
Spring IOC原理解读 面试必读
一、什么是Ioc/DI?
二、 Spring IOC体系结构?
三、IoC容器的初始化?
四、IOC容器的依赖注入
五、IoC容器的高级特性
SpringMVC
动态代理
反射
AOP原理
Spring事务；
springMVC的原理
Spring MVC工作原理 及注解说明
SpringMVC框架介绍
SpringMVC原理图
SpringMVC接口解释
SpringMVC运行原理
spring中beanFactory和ApplicationContext的联系和区别
Spring系列之beanFactory与ApplicationContext
spring注入的几种方式（循环注入）
Spring循环依赖的三种方式
第一种：构造器参数循环依赖
第二种：setter方式单例，默认方式
第三种：setter方式原型，prototype
springIOC
Spring IOC原理解读 面试必读
一、什么是Ioc/DI?
二、 Spring IOC体系结构?
三、IoC容器的初始化?
四、IOC容器的依赖注入
五、IoC容器的高级特性
spring AOP的原理
Spring AOP 实现原理与 CGLIB 应用
AOP 的存在价值
图 1.多个地方包含相同代码的软件
图 2 通过方法调用实现系统功能
使用 AspectJ 的编译时增强进行 AOP
清单 1.Hello.java
清单 2.TxAspect.java
清单 3.LogAspect.java
清单 4.Hello.class
使用 Spring AOP
清单 5.Chinese.java
清单 6.AfterReturningAdviceTest.java
清单 7.AfterReturningAdviceTest.java
清单 8.bean.xml
清单 9.BeanTest.java
清单 10.Person.java
清单 11.BeanTest.java
Spring AOP 原理剖析
图 3.AOP 代理的方法与目标对象的方法
使用 CGLIB 生成代理类
清单 12.AroundAdvice.java

清单 13.ChineseProxyFactory.java
清单 14.Main.java
结束语
spring AOP 两种代理方式
Spring 如何保证 Controller 并发的安全?
springMVC一个Controller处理所有用户请求的并发问题
Spring并发访问的线程安全性问题
spring中用到哪些设计模式?
Spring IOC 的理解，其初始化过程?
Spring的事务管理
1 初步理解
2 核心接口
2.1 事务管理器
2.1.1 JDBC事务
2.1.2 Hibernate事务
2.1.3 Java持久化API事务（JPA）
2.1.4 Java原生API事务
2.2 基本事务属性的定义
2.2.1 传播行为
2.2.2 隔离级别
2.2.3 只读
2.2.4 事务超时
2.2.5 回滚规则
2.3 事务状态
3 编程式事务
3.1 编程式和声明式事务的区别
3.2 如何实现编程式事务?
3.2.1 使用TransactionTemplate
3.2.2 使用PlatformTransactionManager
4 声明式事务
4.1 配置方式
4.2 一个声明式事务的实例
SpringMVC概述
什么是Spring MVC？简单介绍下你对Spring MVC的理解?
Spring MVC的优点
Spring MVC的主要组件?
什么是DispatcherServlet
什么是Spring MVC框架的控制器?
Spring MVC的控制器是不是单例模式,如果是,有什么问题,怎么解决?
工作原理
请描述Spring MVC的工作流程？描述一下 DispatcherServlet 的工作流程?
MVC框架
MVC是什么？MVC设计模式的好处有哪些
常用注解
注解原理是什么
Spring MVC常用的注解有哪些?
SpingMvc中的控制器的注解一般用哪个,有没有别的注解可以替代?
@Controller注解的作用
@RequestMapping注解的作用
@ResponseBody注解的作用
@PathVariable和@RequestParam的区别
其他
Spring MVC与Struts2区别
Spring MVC怎么样设定重定向和转发的?
Spring MVC怎么和AJAX相互调用的?
如何解决POST请求中文乱码问题，GET的又如何处理呢?
Spring MVC的异常处理?
如果在拦截请求中，我想拦截get方式提交的方法,怎么配置

        怎样在方法里面得到Request,或者Session?
            如果想在拦截的方法里面得到从前台传入的参数,怎么得到?
            如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?
        Spring MVC中函数的返回值是什么?
            Spring MVC用什么对象从后台向前台传递数据的?
            怎么样把ModelMap里面的数据放入Session里面?
        Spring MVC里面拦截器是怎么写的
        介绍一下 WebApplicationContext
Spring概述
    1. 什么是spring?
    2. Spring框架的设计目标，设计理念，和核心是什么
    3. Spring的优缺点是什么?
    4. Spring有哪些应用场景
    5. Spring由哪些模块组成?
    6. Spring 框架中都用到了哪些设计模式?
    7. 详细讲解一下核心容器（spring context应用上下文) 模 块
    8. Spring框架中有哪些不同类型的事件
    9. Spring 应用程序有哪些不同组件?
    10. 使用 Spring 有哪些方式?
Spring控制反转(IOC)
    什么是Spring IOC 容器?
    控制反转(IoC)有什么作用
    IOC的优点是什么?
    Spring IoC 的实现机制
    Spring 的 IoC支持哪些功能
    BeanFactory 和 ApplicationContext有什么区别?
    Spring 如何设计容器的，BeanFactory和 ApplicationContext的关系详解
    ApplicationContext通常的实现是什么?
        什么是Spring的依赖注入?
    依赖注入的基本原则
    依赖注入有什么优势
    有哪些不同类型的依赖注入实现方式?
    构造器依赖注入和 Setter方法注入的区别
    Spring Beans（19）什么是Spring beans?
    一个 Spring Bean 定义 包含什么?
        如何创建一个如何给Spring 容器提供配置元数据？Spring有几种配置方式
        Spring配置文件包含了哪些信息
    Spring基于xml注入bean的几种方式
    你怎样定义类的作用域?
    解释Spring支持的几种bean的作用域
        Spring框架中的单例bean是线程安全的吗?
    Spring如何处理线程并发问题?
    哪些是重要的bean生命周期方法？ 你能重载它们吗?
    什么是Spring的内部bean？什么是Spring inner beans?
    在 Spring中如何注入一个java集合?
    什么是bean装配?
    什么是bean的自动装配?
        解释不同方式的自动装配，spring 自动装配 bean 有哪些方式?
    使用@Autowired注解自动装配的过程是怎样的?
    自动装配有哪些局限性?
        你可以在Spring中注入一个null 和一个空字符串吗?
Spring注解
    什么是基于Java的Spring注解配置? 给一些注解的例子
    怎样开启注解装配?
    @Component, @Controller, @Repository, @Service 有何区别?
    @Required 注解有什么作用
    @Autowired 注解有什么作用
    @Autowired和@Resource之间的区别

@Qualifier 注解有什么作用
@RequestMapping 注解有什么用?
Spring数据访问
解释对象/关系映射集成模块
在Spring框架中如何更有效地使用JDBC?
解释JDBC抽象和DAO模块
spring DAO 有什么用?
spring JDBC API 中存在哪些类?
JdbcTemplate是什么
使用Spring通过什么方式访问Hibernate? 使用 Spring 访问 Hibernate 的方法有哪些?
如何通扩展 Hib过HibernateDaoSupport将Spring和Hibernate 结合起来?
Spring支持的事务管理类型， spring 事务实现方式有哪些?
Spring事务的实现方式和实现原理
说一下Spring的事务传播行为
说一下 spring 的事务隔离?
Spring框架的事务管理有哪些优点?
你更倾向用那种事务管理类型?
Spring面向切面编程(AOP)什么是AOP
Spring AOP and AspectJ AOP 有什么区别? AOP 有哪些实现方式?
JDK动态代理和CGLIB动态代理的区别
InvocationHandler 的 invoke(Object proxy,Method method,Object[] args):
如何理解 Spring 中的代理?
解释一下Spring AOP里面的几个名词
Spring在运行时通知对象
连接点的支持，而且它不支持构造器连接点。方法之外的连接点拦截功能，我们在Spring AOP 中，关注点
和横切关注的区别是什么? 在 spring aop 中 concern 和 cross-cutting concern 的不同之处
Spring通知有哪些类型?
什么是切面 Aspect?
解释基于XML Schema方式的切面实现
有几种不同类型的自动代理?
SpringMVC面试题
什么是 SpringMvc?
Spring MVC 的优点：答:
SpringMVC 工作原理? 答:
SpringMVC 流程? 答:
SpringMvc 的控制器是不是单例模式,如果是,有什么问题,怎么解决?
如果你也用过 struts2.简单介绍下 springMVC 和 struts2 的区别有哪些?
SpingMvc 中的控制器的注解一般用那个,有没有别的注解可以替代?
@RequestMapping 注解用在类上面有什么作用?
怎么样把某个请求映射到特定的方法上面?
如果在拦截请求中,我想拦截 get 方式提交的方法,怎么配置?
怎么样在方法里面得到 Request,或者 Session?
我想在拦截的方法里面得到从前台传入的参数,怎么得到?
如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?
SpringMvc 中函数的返回值是什么?
SpringMVC 怎么样设定重定向和转发的?
SpringMvc 用什么对象从后台向前台传递数据的?
SpringMvc 中有个类把视图和数据都合并的一起的,叫什么?
怎么样把 ModelMap 里面的数据放入 Session 里面?
SpringMvc 怎么和 AJAX 相互调用的? 答:
讲下 SpringMvc 的执行流程
Spring Boot（连老师）
Spring Boot概述
什么是 Spring Boot?
Spring Boot 有哪些优点?
Spring Boot 的核心注解是哪个? 它主要由哪几个注解组成的?
配置
什么是 JavaConfig?

Spring Boot 自动配置原理是什么?
你如何理解 Spring Boot 配置加载顺序?
4）命令行参数； 什么是 YAML?
YAML 配置的优势在哪里？
Spring Boot 是否可以使用 XML 配置？
什么是 Spring Profiles?
如何在自定义端口上运行 Spring Boot 应用程序?
安全
如何实现 Spring Boot 应用程序的安全性?
比较一下 Spring Security 和 Shiro 各自的优缺点？
Spring Boot 中如何解决跨域问题？
什么是 CSRF 攻击?
监视器
Spring Boot 中的监视器是什么?
我们如何监视所有 Spring Boot 微服务?
整合第三方项目
什么是 WebSockets?
什么是 Spring Batch?
什么是 FreeMarker 模板?
如何集成 Spring Boot 和 ActiveMQ?
什么是 Apache Kafka?
什么是 Swagger？你用 Spring Boot 实现了它吗?
其他
如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？Spring Boot项目如何热部署?
您使用了哪些 starter maven 依赖项?
运行 Spring Boot 有哪几种方式?
Spring Boot 需要独立的容器运行吗?
开启 Spring Boot 特性有哪几种方式?
如何使用 Spring Boot 实现异常处理?
如何使用 Spring Boot 实现分页和排序?
Spring Boot 中如何实现定时任务？
Spring Boot 原理
Spring Boot比Spring做了哪些改进?
Spring boot 热加载
Spring Boot设置有效时间和自动刷新缓存，时间支持在配置文件中配置
Spring Boot缓存实战 Redis 设置有效时间和自动刷新缓存，时间支持在配置文件中配置
问题描述
解决思路
CacheManager说明
Cache说明
请求步骤
代码分析
hibernate和ibatis的区别
Ibatis与Hibernate的区别
一、 hibernate与ibatis之间的比较：
二、hibernate与ibatis两者的对比：
三、iBatis与Hibernate区别?
四、选择Hibernate还是iBATIS都有它的道理：
讲讲mybatis的连接池。
Mybatis数据源与连接池
一、MyBatis数据源DataSource分类
二、数据源DataSource的创建过程
三、 DataSource什么时候创建Connection对象
四、不使用连接池的UnpooledDataSource
五、为什么要使用连接池?
六、使用了连接池的PooledDataSource
七、JNDI类型的数据源DataSource
经典面试题

问题一
问题二
问题三
问题四
问题五
问题六
问题七
问题八
问题九
问题十
问题十一
问题十二
问题十三
问题十四
问题十五
问题十六
问题十七
问题十八
问题十九
问题二十
问题二十一
问题二十二
Spring Cloud（一明老师，晁老师）
Spring Cloud熔断机制介绍；
Spring Cloud对比下Dubbo，什么场景下该使用Spring Cloud?
CAP原理和BASE理论
CAP原则(CAP定理)、BASE理论
一、CAP原则
一致性与可用性的决择编辑
取舍策略
BASE理论
基本可用
最终一致性
小结：
与NoSQL的关系编辑
CAP的是什么关系
为什么会是这样
选择权衡
延伸
分布式系统的典型应用
分布式事务、分布式锁
常用的分布式事务解决方案介绍有多少种?
一、一致性理论
二、一致性模型
三、分布式事务解决方案
四、选择建议
分布式锁的几种实现方式
1. 使用Redis实现分布式锁
* WATCH, MULTI, EXEC, DISCARD事务机制实现分布式锁
* SETNX实现分布式锁
* 锁的释放
2. 使用Memcached实现分布式锁
3. 使用ZooKeeper实现分布式锁
获取锁
释放锁
获取锁
释放锁
Spring Cloud面试题

        什么是 Spring Cloud?
        使用 Spring Cloud 有什么优势?
        服务注册和发现是什么意思？Spring Cloud 如何实现?
        负载平衡的意义什么?
        什么是 Hystrix？它如何实现容错?
        什么是Hystrix 断路器？我们需要它吗?
        什么是 Netflix Feign？它的优点是什么?
        什么是 Spring Cloud Bus？我们需要它吗?
    为什么需要学习Spring Cloud
    什么是Spring Cloud
    设计目标与优缺点
        优缺点
    Spring Cloud发展前景
    整体架构
    主要项目
        Spring Cloud Config
        Spring Cloud Netflix
        Spring Cloud Bus
        Spring Cloud Sleuth
        Spring Cloud Stream
        Spring Cloud Task
        Spring Cloud OpenFeign
    Spring Cloud的版本关系
        Spring Cloud和SpringBoot版本对应关系
        Spring Cloud和各子项目版本对应关系
    SpringBoot和SpringCloud的区别?
        使用 Spring Boot 开发分布式微服务时，我们面临以下问题
    服务注册和发现是什么意思？Spring Cloud 如何实现?
    负载平衡的意义什么?
    什么是 Hystrix？它如何实现容错?
    什么是 Hystrix 断路器？我们需要它吗?
        什么是 Netflix Feign？它的优点是什么?
        什么是 Spring Cloud Bus？我们需要它吗?
    Spring Cloud断路器的作用
        什么是Spring Cloud Config?
        什么是Spring Cloud Gateway?
        一. 分布式模型
            1. 节点
            异常
        副本
        3. 衡量分布式系统的指标

Mysql（连老师）

数据库基础知识

为什么要使用数据库

数据保存在内存

优点：存取速度快

缺点：数据不能永久保存

数据保存在文件

优点：数据永久保存

缺点：1）速度比内存操作慢，频繁的IO操作。

```
        2）查询数据不方便
```

数据保存在数据库

1）数据永久保存

2）使用SQL语句，查询方便效率高。

3）管理数据方便

什么是SQL?

结构化查询语言(StructuredQueryLanguage)简称SQL，是一种数据库查询语言。

作用：用于存取数据、查询、更新和管理关系数据库系统。

什么是MySQL?

MySQL是一个关系型数据库管理系统，由瑞典MySQLAB公司开发，属于Oracle旗下产品。MySQL是最流行的关系型数据库管理系统之一，在WEB应用方面，MySQL是最好的RDBMS(RelationalDatabaseManagementSystem，关系数据库管理系统)应用软件之一。在Java企业级开发中非常常用，因为MySQL是开源免费的，并且方便扩展。

数据库三大范式是什么

第一范式：每个列都不可以再拆分。

第二范式：在第一范式的基础上，非主键列完全依赖于主键，而不能是依赖于主键的一部分。

第三范式：在第二范式的基础上，非主键列只依赖于主键，不依赖于其他非主键。

在设计数据库结构的时候，要尽量遵守三范式，如果不遵守，必须有足够的理由。比如性能。事实上我们经常会为了性能而妥协数据库的设计。

mysql有关权限的表都有哪几个

MySQL服务器通过权限表来控制用户对数据库的访问，权限表存放在mysql数据库里，由mysql_install_db脚本初始化。这些权限表分别user，db，table_priv，columns_priv和host。下面分别介绍一下这些表的结构和内容：

user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。

db权限表：记录各个帐号在各个数据库上的操作权限。

table_priv权限表：记录数据表级的操作权限。

columns_priv权限表：记录数据列级的操作权限。

host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受GRANT和REVOKE语句的影响。

MySQL的binlog有有几种录入格式？分别有什么区别？

有三种格式，statement，row和mixed。

- statement模式下，每一条会修改数据的sql都会记录在binlog中。不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。由于sql的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制。
- row级别下，不记录sql语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是可以全部记下来但是由于很多操作，会导致大量行的改动(比如altertable)，因此这种模式的文件保存的信息太多，日志量太大。
- mixed，一种折中的方案，普通操作使用statement记录，当无法使用statement的时候使用row。

此外，新版的MySQL中对row级别也做了一些优化，当表结构发生变化的时候，会记录语句而不是逐行记录。

数据类型

mysql有哪些数据类型

分类	类型名称	说明
整数类型	tinyInt	很小的整数(8位二进制)
smallint	小的整数(16位二进制)	
mediumint	中等大小的整数(24位二进制)	
int(integer)	普通大小的整数(32位二进制)	
小数类型	float	单精度浮点数
double	双精度浮点数	
decimal(m,d)	压缩严格的定点数	
日期类型	year	YYYY1901~2155
time	HH:MM:SS-838:59:59~838:59:59	
date	YYYYMM-DD1000-01-01~9999-12-3	
datetime	YYYYMM-DDHH:MM:SS1000-01-0100:00:00~9999-12-3123:59:59	
timestamp	YYYYMM-DDHH:MM:SS1970010100:00:01UTC~2038-01-1903:14:07UTC	
文本、二进制类型	**CHAR(M)**	M为0~65535之间的整数
VARCHAR(M)	M为0~65535之间的整数	
TINYBLOB	允许长度0~255字节	
BLOB	允许长度0~65535字节	
MEDIUMBLOB	允许长度0~167772150字节	
LONGBLOB	允许长度0~4294967295字节	
TINYTEXT	允许长度0~255字节	
TEXT	允许长度0~65535字节	
MEDIUMTEXT	允许长度0~167772150字节	
LONGTEXT	允许长度0~4294967295字节	
VARBINARY(M)	允许长度0~M个字节的变长字节字符串	
BINARY(M)	允许长度0~M个字节的定长字节字符串	

整数类型，包括TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT，分别表示1字节、2字节、3字节、4字节、8字节整数。任何整数类型都可以加上UNSIGNED属性，表示数据是无符号的，即非负整数。

长度：整数类型可以被指定长度，例如：INT(11)表示长度为11的INT类型。长度在大多数场景是没有意义的，它不会限制值的合法范围，只会影响显示字符的个数，而且需要和UNSIGNEDZEROFILL属性配合使用才有意义。

例子，假定类型设定为INT(5)，属性为UNSIGNEDZEROFILL，如果用户插入的数据为12的话，那么数据库实际存储数据为00012。

实数类型，包括FLOAT、DOUBLE、DECIMAL。DECIMAL可以用于存储比BIGINT还大的整型，能存储精确的小数。而FLOAT和DOUBLE是有取值范围的，并支持使用标准的浮点进行近似计算。
计算时FLOAT和DOUBLE相比DECIMAL效率更高一些，DECIMAL你可以理解成是用字符串进行处理。

字符串类型，包括VARCHAR、CHAR、TEXT、BLOBVARCHAR用于存储可变长字符串，它比定长类型更节省空间。
VARCHAR使用额外1或2个字节存储字符串长度。列长度小于255字节时，使用1字节表示，否则使用2字节表示。
VARCHAR存储的内容超出设置的长度时，内容会被截断。
CHAR是定长的，根据定义的字符串长度分配足够的空间。
CHAR会根据需要使用空格进行填充方便比较。
CHAR适合存储很短的字符串，或者所有值都接近同一个长度。
CHAR存储的内容超出设置的长度时，内容同样会被截断。

使用策略：

对于经常变更的数据来说，CHAR比VARCHAR更好，因为CHAR不容易产生碎片。

对于非常短的列，CHAR比VARCHAR在存储空间上更有效率。

使用时要注意只分配需要的空间，更长的列排序时会消耗更多内存。尽量避免使用TEXT/BLOB类型，查询时会使用临时表，导致严重的性能开销。

枚举类型（ENUM），把不重复的数据存储为一个预定义的集合。有时可以使用ENUM代替常用的字符串类型。ENUM存储非常紧凑，会把列表值压缩到一个或两个字节。ENUM在内部存储时，其实存的是整数。尽量避免使用数字作为ENUM枚举的常量，因为容易混乱。排序是按照内部存储的整数

日期和时间类型，尽量使用timestamp，空间效率高于datetime，用整数保存时间戳通常不方便处理。如果需要存储微妙，可以使用bigint存储。

看到这里，这道真题是不是就比较容易回答了。

引擎

MySQL存储引擎MyISAM与InnoDB区别

存储引擎Storageengine：MySQL中的数据、索引以及其他对象是如何存储的，是一套文件系统的实现。

常用的存储引擎有以下：

Innodb引擎：Innodb引擎提供了对数据库ACID事务的支持。并且还提供了行级锁和外键的约束。它的设计的目标就是处理大数据容量的数据库系统。

MyIASM引擎(原本Mysql的默认引擎)：不提供事务的支持，也不支持行级锁和外键。

MEMORY引擎：所有的数据都在内存中，数据的处理速度快，但是安全性不高。

MyISAM与InnoDB区别

MyISAM	Innodb	
存储结构	每张表被存放在三个文件：frm-表格定义、MYD(MYData)-数据文件、MYI(MYIndex)-索引文件	所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB
存储空间	MyISAM可被压缩，存储空间较小	InnoDB的表需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引
可移植性、备份及恢复	由于MyISAM的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作	免费的方案可以是拷贝数据文件、备份binlog，或者用mysqldump，在数据量达到几十G的时候就相对痛苦了
文件格式	数据和索引是分别存储的，数据.MYD，索引.MYI	数据和索引是集中存储的，.ibd
记录存储顺序	按记录插入顺序保存	按主键大小有序插入
外键	不支持	支持
事务	不支持	支持
锁支持（锁是避免资源争用的一个机制，MySQL锁对用户几乎是透明的）	表级锁定	行级锁定、表级锁定，锁定力度小并发能力高
SELECT	MyISAM更优	
INSERT、UPDATE、DELETE	InnoDB更优	
selectcount(*)	myisam更快，因为myisam内部维护了一个计数器，可以直接调取。	
索引的实现方式	B+树索引，myisam是堆表	B+树索引，Innodb是索引组织表
哈希索引	不支持	支持
全文索引	支持	不支持

MyISAM索引与InnoDB索引的区别?

- InnoDB索引是聚簇索引，MyISAM索引是非聚簇索引。
- InnoDB的主键索引的叶子节点存储着行数据，因此主键索引非常高效。
- MyISAM索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。
- InnoDB非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

InnoDB引擎的4大特性

- 插入缓冲（insertbuffer)
- 二次写(doublewrite)
- 自适应哈希索引(ahi)
- 预读(readahead)

存储引擎选择

如果没有特别的需求，使用默认的Innodb即可。
MyISAM：以读写插入为主的应用程序，比如博客系统、新闻门户网站。
Innodb：更新（删除）操作频率也高，或者要保证数据的完整性；并发量高，
支持事务和外键。比如OA自动化办公系统。

索引

什么是索引?

索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它
们包含着对数据表里所有记录的引用指针。
索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种
B+树。
更通俗的说，索引就相当于目录。为了方便查找书中的内容，通过对内容建立索引形成目录。索引是一个文件，它是要占据物理空间的。

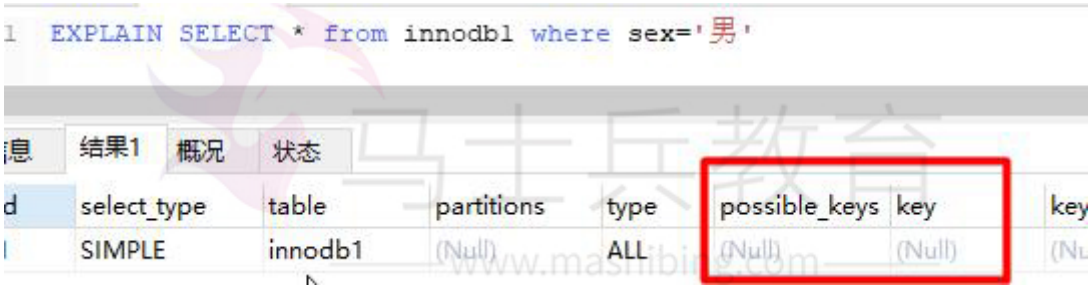
索引有哪些优缺点?

索引使用场景（重点）



上图中，根据id查询记录，因为id字段仅建立了主键索引，因此此SQL执行可选
的索引只有主键索引，如果有多个，最终会选一个较优的作为检索的依据。

```
1--增加一个没有建立索引的字段
2altertableinnodb1addsexchar(1);
3--按sex检索时可选的索引为null
4EXPLAINSELECT*frominnodb1wheresex='男';
```



> 可以尝试在一个字段未建立索引时，根据该字段查询的效率，然后对该字段建立索引（altertable表名addindex(字段名)），同样的SQL执行的效率，你会发现查询效率会有明显的提升（数据量越大越明显）。

orderby
当我们使用orderby将查询结果按照某个字段排序时，如果该字段没有建立索引，那么执行计划会将查询出的所有数据使用外部排序（将数据从硬盘分批读取到内存使用内部排序，最后合并排序结果），这个操作是很影响性能的，因为需要将查询涉及到的所有数据从磁盘中读到内存（如果单条数据过大或者数据量过多都会降低效率），更无论读到内存之后的排序了。
但是如果我们对该字段建立索引altertable表名addindex(字段名)，那么由于索引本身是有序的，因此直接按照索引的顺序和映射关系逐条取出数据即可。而且如果分页的，那么只用取出索引表某个范围内的索引对应的数据，而不用像上述那取出所有数据进行排序再返回某个范围内的数据。（从磁盘取数据是最影响性能的）
join
对join语句匹配关系（on）涉及的字段建立索引能够提高效率
索引覆盖
如果要查询的字段都建立过索引，那么引擎会直接在索引表中查询而不会访问原始数据（否则只要有一个字段没有建立索引就会做全表扫描），这叫索引覆盖。
因此我们需要尽可能的在select后只写必要的查询字段，以增加索引覆盖的几率。
这里值得注意的是不要想着为每个字段建立索引，因为优先使用索引的优势就在于其体积小。

索引有哪几种类型?

主键索引:数据列不允许重复，不允许为NULL，
一个表只能有一个主键。

唯一索引:数据列不允许重复，允许为NULL值，
一个表允许多个列创建唯一索引。

- 可以通过ALTERTABLEtable_nameADDUNIQUE(column);
  创建唯一索引
- 可以通过ALTERTABLEtable_nameADDUNIQUE(column1,column2);
  创建唯一组合索引

普通索引:基本的索引类型，没有唯一性的限制，允许为NULL值。

- 可以通过ALTERTABLEtable_nameADDINDEXindex_name(column);
  创建普通索引
- 可以通过ALTERTABLEtable_nameADDINDEXindex_name(column1,column2,column3);
  创建组合索引
- **全文索引**：是目前搜索引擎使用的一种关键技术。
  可以通过ALTERTABLEtable_nameADDFULLTEXT(column);
  创建全文索引

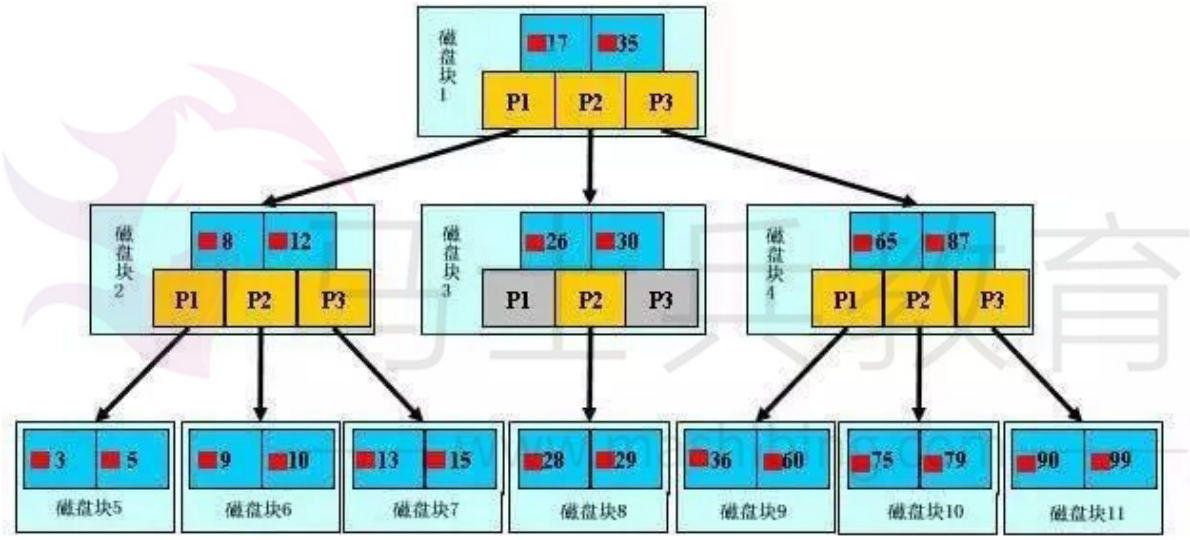
索引的数据结构（b树，hash）

索引的数据结构和具体存储引擎的实现有关，在MySQL中使用较多的索引有

Hash索引，B+树索引等，而我们经常使用的InnoDB存储引擎的默认索引实现为：B+树索引。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能快；其余大部分场景，建议选择BTree索引。

1）B树索引mysql通过存储引擎取数据，基本上90%的人用的就是InnoDB了，按照实现方式分，InnoDB的索引类型目前只有两种：BTREE（B树）索引和HASH索引。B树索引是Mysql数据库中使用频繁的索引类型，基本所有存储引擎都支持

BTree索引。通常我们说的索引不出意外指的就是（B树）索引（实际是用B+树实现的，因为在查看表索引时，mysql一律打印BTREE，所以简称为B树索引）



查询方式：

主键索引区:PI(关联保存的时数据的地址)按主键查询,

普通索引区:si(关联的id的地址,然后再到达上面的地址)。所以按主键查询,速度快

B+tree性质：

1.）n棵子tree的节点包含n个关键字，不用来保存数据而是保存数据的索引。

2.）所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

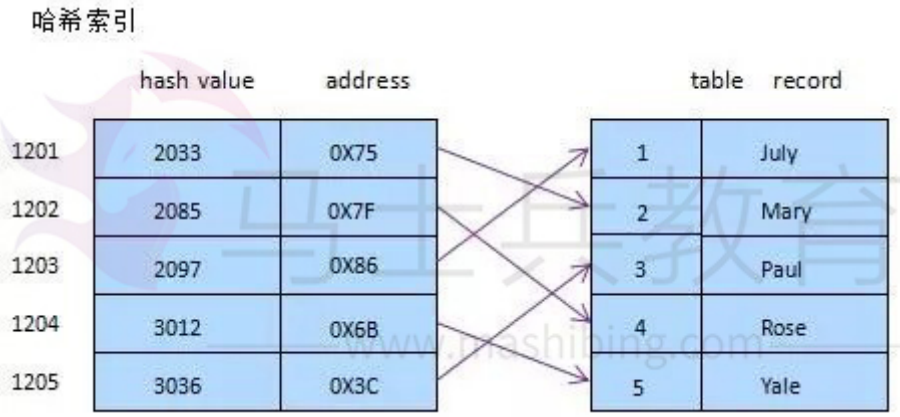
3.）所有的非终端结点可以看成是索引部分，结点中仅含其子树中的大（或小）关键字。

4.）B+树中，数据对象的插入和删除仅在叶节点上进行。

5.）B+树有2个头指针，一个是树的根节点，一个是小关键码的叶节点。2）哈希索引简要说下，类似于数据结构中简单实现的HASH表（散列表）一样，当我们在

mysql中用哈希索引时，主要就是通过Hash算法（常见的Hash算法有直接定址法、平方取中法、折叠法、除数取余法、随机数法），将数据库字段数据转换成定长的Hash值，与这条数据的行指针一并存入Hash表的对应位置；如果发生

Hash碰撞（两个不同关键字的Hash值相同），则在对应Hash键下以链表形式存储。当然这只是简略模拟图。



索引的基本原理

索引用来快速地寻找那些具有特定值的记录。如果没有索引，一般来说执行查询时遍历整张表。

索引的原理很简单，就是把无序的数据变成有序的查询

1. 把创建了索引的列的内容进行排序
2. 对排序结果生成倒排表
3. 在倒排表内容上拼上数据地址链
4. 在查询的时候，先拿到倒排表内容，再取出数据地址链，从而拿到具体数据

索引算法有哪些?

索引算法有BTree算法和Hash算法

BTree算法

BTree是常用的mysql数据库索引算法，也是mysql默认的算法。因为它不仅可以被用在=,>,>=,<,<=和between这些比较操作符上，而且还可以用于like操作符，只要它的查询条件是一个不以通配符开头的常量，例如：

```
1   --只要它的查询条件是一个不以通配符开头的常量
2   select*fromuserwherenamelike'jack%';
3   --如果一通配符开头，或者没有使用常量，则不会使用索引，例如：
4   select*fromuserwherenamelike'%jack';
```

Hash算法

HashHash索引只能用于对等比较，例如=,<=>（相当于=）操作符。由于是一次定位数据，不像BTree索引需要从根节点到枝节点，后才能访问到页节点这样多次IO访问，所以检索效率远高于BTree索引。

索引设计的原则?

1.适合索引的列是出现在where子句中的列，或者连接子句中指定的列

2.基数较小的类，索引效果较差，没有必要在此列建立索引

3.使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间

4.不要过度索引。索引需要额外的磁盘空间，并降低写操作的性能。在修改表内容的时候，索引会进行更新甚至重构，索引列越多，这个时间就会越长。所以只保持需要的索引有利于查询即可。

创建索引的原则（重中之重）

索引虽好，但也不是无限制的使用，好符合一下几个原则

1）左前缀匹配原则，组合索引非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a=1andb=2andc>3andd=4如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立

(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

2）较频繁作为查询条件的字段才去创建索引

3）更新频繁字段不适合创建索引

4）若是不能有效区分数据的列不适合做索引列(如性别，男女未知，多也就三种，区分度实在太低)

5）尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

6）定义有外键的数据列一定要建立索引。

7）对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。8）对于定义为text、image和bit的数据类型的列不要建立索引。创建索引的三种方式，删除索引

第一种方式：在执行CREATETABLE时创建索引

```
1   CREATETABLEuser_index2(
2   idINTauto_incrementPRIMARYKEY,
3   first_nameVARCHAR(16),
4   last_nameVARCHAR(16),
5   id_cardVARCHAR(18),
6   informationtext,
7   KEYname(first_name,last_name),
8   FULLTEXTKEY(information),
9   UNIQUEKEY(id_card));
```

第二种方式：使用ALTERTABLE命令去增加索引

```
1ALTERTABLEtable_nameADDINDEXindex_name(column_list);
```

ALTERTABLE用来创建普通索引、UNIQUE索引或PRIMARYKEY索引。

其中table_name是要增加索引的表名，column_list指出对哪些列进行索引，多列时各列之间用逗号分隔。

索引名index_name可自己命名，缺省时，MySQL将根据第一个索引列赋一个名称。另外，ALTERTABLE允许在单个语句中更改多个表，因此可以在同时创建多个索引。

第三种方式：使用CREATEINDEX命令创建

```
1CREATEINDEXindex_nameONtable_name(column_list);
```

CREATEINDEX可对表增加普通索引或UNIQUE索引。（但是，不能创建PRIMARYKEY索引）

删除索引

根据索引名删除普通索引、唯一索引、全文索引：altertable表名dropKEY索引名

```
1   altertableuser_indexdropKEYname;
2   altertableuser_indexdropKEYid_card;
3   altertableuser_indexdropKEYinformation;
```

删除主键索引：altertable表名dropprimarykey（因为主键只有一个）。这里值得注意的是，如果主键自增长，那么不能直接执行此操作（自增长依赖于主键索引）：

```
1  alter table user_index drop primary key
```

信息 概况 状态

[SQL]alter table user_index drop primary key

[Err] 1075 - Incorrect table definition; there can be only one auto column and it must be defined as a key

马士兵教育 www.mashibing.com

需要取消自增长再行删除：

```
altertableuser_index
--重新定义字段
MODIFYidint,
dropPRIMARYKEY
```

但通常不会删除主键，因为设计主键一定与业务逻辑无关。

创建索引时需要注意什么?

- 非空字段：应该指定列为NOTNULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；
- 取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；
- 索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。

使用索引查询一定能提高查询的性能吗？为什么

通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

索引需要空间来存储，也需要定期维护，每当有记录在表中增减或索引列被修改时，索引本身也会被修改。这意味着每条记录的INSERT，DELETE，UPDATE将为此多付出4，5次的磁盘I/O。因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反应时间变慢。使用索引查询不一定能提高查询性能，索引范围查询(INDEXRANGESCAN)适用于两种情况:

基于一个范围的检索，一般查询返回结果集小于表中记录数的30%

基于非唯一性索引的检索

百万级别或以上的数据如何删除

关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件,所以当我们对数据的增加,修改,删除,都会产生额外的对索引文件的操作,这些操作需要消耗额外的IO,会降低增/改/删的执行效率。所以，在我们删除数据库百万级别数据的时候，查询MySQL官方手册得知删除数据的速度和创建的索引数量是成正比的。

1. 所以我们想要删除百万数据的时候可以先删除索引（此时大概耗时三分多钟）
2. 然后删除其中无用数据（此过程需要不到两分钟）
3. 删除完成后重新创建索引(此时数据较少了)创建索引也非常快，约十分

钟左右。

4. 与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回滚。那更是坑了。

前缀索引

语法：index(field(10))，使用字段值的前10个字符建立索引，默认是使用字段的全部内容建立索引。

前提：前缀的标识度高。比如密码就适合建立前缀索引，因为密码几乎各不相

同。

实操的难度：在于前缀截取的长度。

我们可以利用selectcount(*)/count(distinctleft(password,prefixLen));，通过从调整prefixLen的值（从1自增）查看不同前缀长度的一个平均匹配度，接近1时就可以了（表示一个密码的前prefixLen个字符几乎能确定唯一一条记录）

什么是最左前缀原则？什么是最左匹配原则

顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where子句中

使用最频繁的一列放在最左边。

最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a=1andb=2andc>3andd=4如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

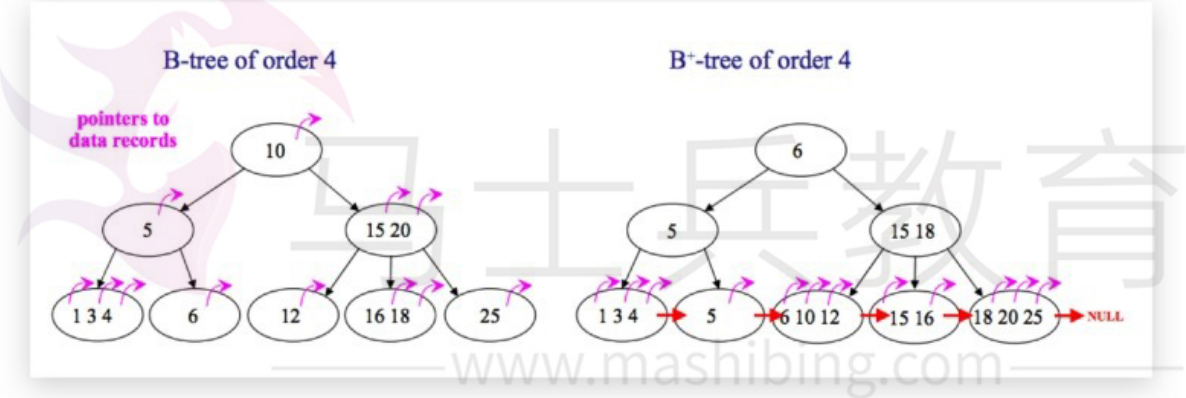
=和in可以乱序，比如a=1andb=2andc=3建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

B树和B+树的区别

在B树中，你可以将键和值存放在内部节点和叶子节点；但在B+树

中，内部节点都是键，没有值，叶子节点同时存放键和值。

B+树的叶子节点有一条链相连，而B树的叶子节点各自独立。



使用B树的好处

B树可以在内部节点同时存储键和值，因此，把频繁访问的数据放在靠近根节点的地方将会大大提高热点数据的查询效率。这种特性使得B树在特定数据重复多次查询的场景中更加高效。

使用B+树的好处

由于B+树的内部节点只存放键，不存放值，因此，一次读取，可以在内存页中获取更多的键，有利于更快地缩小查找范围。B+树的叶节点由一条链相连，因此，当需要进行一次全数据遍历的时候，B+树只需要使用O(logN)时间找到最小的一个节点，然后通过链进行O(N)的顺序遍历即可。而B树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

Hash索引和B+树所有有什么区别或者说优劣呢?

首先要知道Hash索引和B+树索引的底层实现原理：

hash索引底层就是hash表，进行查找时，调用一次hash函数就可以获取到相应

的键值，之后进行回表查询获得实际数据。B+树底层实现是多路平衡查找树。

对于每一次的查询都是从根节点出发，查找到叶子节点方可以获得所查键值，然

后根据查询判断是否需要回表查询数据。

么可以看出他们有以下的不同：

hash索引进行等值查询更快(一般情况下)，但是却无法进行范围查询。

因为在hash索引中经过hash函数建立索引之后，索引的顺序与原顺序无法保持一致，不能支持范围查询。而B+树的的所有节点皆遵循(左节点小于父节点，右节点大于父节点，多叉树也类似)，天然支持范围。

hash索引不支持使用索引进行排序，原理同上。

hash索引不支持模糊查询以及多列索引的最左前缀匹配。原理也是因为hash函

数的不可预测。AAAA和AAAAB的索引没有相关性。

hash索引任何时候都避免不了回表查询数据，而B+树在符合某些条件(聚簇索

引，覆盖索引等)的时候可以只通过索引完成查询。

hash索引虽然在等值查询上较快，但是不稳定。性能不可预测，当某个键值存在大量重复的时候，发生hash碰撞，此时效率可能极差。而B+树的查询效率比较稳定，对于所有的查询都是从根节点到叶子节点，且树的高度较低。

因此，在大多数情况下，直接选择B+树索引可以获得稳定且较好的查询速度。

而不需要使用hash索引。

数据库为什么使用B+树而不是B树

B树只适合随机检索，而B+树同时支持随机检索和顺序检索；

B+树空间利用率更高，可减少I/O次数，磁盘读写代价更低。一般来说，索引本

身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗。B+树的内部结点并没有指向关键字具体信息的指针，只是作为索引使用，其内部结点比B树小，盘块能容纳的结

点中关键字数量更多，一次性读入内存中可以查找的关键字也就越多，相对的，IO读

写次数也就降低了。而IO读写次数是影响索引检索效率的最大因素；

B+树的查询效率更加稳定。B树搜索有可能会在非叶子结点结束，越靠近根节点的记录查找时间越短，只要找到关键字即可确定记录的存在，其性能等价于在关键字全集内做一次二分查找。而在B+树中，顺序检索比较明显，随机检索时，任何关键字的查找都必须走一条从根节点到叶节点的路，所有关键字的查找路径长度相同，导致每一个关键字的查询效率相当。

B-树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。B+树的叶子节点使用指针顺序连接在一起，只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作。

增删文件（节点）时，效率更高。因为B+树的叶子节点包含所有关键字，并以

有序的链表结构存储，这样可很好提高增删效率。

B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据

在B+树的索引中，叶子节点可能存储了当前的key值，也可能存储了当前的key值以及整行的数据，这就是聚簇索引和非聚簇索引。在InnoDB中，只有主键索引是聚簇索引，如果没有主键，则挑选一个唯一键建立聚簇索引。如果没有唯一键，则隐式的生成一个键来建立聚簇索引。

当查询使用聚簇索引时，在对应的叶子节点，可以获取到整行数据，因此不用再

次进行回表查询。

什么是聚簇索引？何时使用聚簇索引与非聚簇索引

聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据

非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因

澄清一个概念：innodb中，在聚簇索引之上创建的索引称之为辅助索引，辅助索引访问数据总是需要二次查找，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引，辅助索引叶子节点存储的不再是行的物理位置，而是主键值

何时使用聚簇索引与非聚簇索引

动作	使用聚簇索引	使用非聚簇索引
列经常被分组排序	应	应
返回某范围内的数据	应	不应
一个或极少不同值	不应	不应
小数目的不同值	应	不应
大数目的不同值	不应	应
频繁更新的列	不应	应
外键列	应	应
主键列	应	应
频繁修改索引列	不应	应

马士兵教育 www.mashibing.com

非聚簇索引一定会回表查询吗?

不一定，这涉及到查询语句所要求的字段是否全部命中了索引，如果全部命中了

索引，那么就不必再进行回表查询

举个简单的例子，假设我们在员工表的年龄上建立了索引，那么当进行select age from employee where age < 20的查询时，在索引的叶子节点上，已经包含了age信息，不会再次进行回表查询。

联合索引是什么？为什么需要注意联合索引中的顺序?

MySQL可以使用多个字段同时建立一个索引，叫做联合索引。在联合索引中，如果想要命中索引，需要按照建立索引时的字段顺序挨个使用，否则无法命中索引。

具体原因为:

MySQL使用索引时需要索引有序，假设现在建立了"name，age，school"的联合索引，那么索引的排序为: 先按照name排序，如果name相同，则按照age排序，如果age的值也相等，则按照school进行排序。

当进行查询时，此时索引仅仅按照name严格有序，因此必须首先使用name字段进行等值查询，之后对于匹配到的列而言，其按照age字段严格有序，此时可以使用age字段用做索引查找，以此类推。因此在建立联合索引的时候应该注意索引列的顺序，一般情况下，将查询需求频繁或者字段选择性高的列放在前面。此外可以根据特例的查询或者表结构进行单独的调整。

事务

什么是数据库事务?

事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务最经典也经常被拿出来说例子就是转账了。

假如小明要给小红转账1000元，这个转账会涉及到两个关键操作就是：将小明的余额减少1000元，将小红的余额增加1000元。万一在这两个操作之间突然出现错误比如银行系统崩溃，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两个关键操作要么都成功，要么都要失败。

事物的四大特性(ACID)介绍一下?

关系性数据库需要遵循ACID规则，具体内容如下：



原子性： 事务是最小的执行单位，不允许分割。事务的原子性确保动作

要么全部完成，要么完全不起作用；

2. 一致性： 执行事务前后，数据保持一致，多个事务对同一个数据读取的

结果是相同的；

3. 隔离性： 并发访问数据库时，一个用户的事务不被其他事务所干扰，各

并发事务之间数据库是独立的；

4. 持久性： 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

什么是脏读？幻读？不可重复读?

脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这

可能是两次查询过程中间插入了一个事务更新的原有的数据。

幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

什么是事务的隔离级别？MySQL的默认隔离级别是什么?

为了达到事务的四大特性，数据库定义了4种不同的事务隔离级别，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

隔离级别	脏读	不可重复 读	幻影读
READ- UNCOM MITTED	√	√	√
READ- COMMIT TED	×	√	√
REPEATA BLE- READ	×	×	√
SERIALIZ ABLE	×	×	×

SQL 标准定义了四个隔离级别：

- READ-UNCOMMITTED(读取未提交)： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。
- READ-COMMITTED(读取已提交)： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。
- REPEATABLE-READ(可重复读)： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- SERIALIZABLE(可串行化)： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

这里需要注意的是：Mysql 默认采用的 REPEATABLE_READ隔离级别 Oracle默认采用的 READ_COMMITTED隔离级别

事务隔离机制的实现基于锁机制和并发调度。其中并发调度使用的是MVVC（多版本并发控制），通过保存修改的旧版本信息来支持并发一致性读和回滚等特性。

因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是READ-COMMITTED(读取提交内容):，但是你要知道的是InnoDB 存储引擎默认使用 **REPEATABLE-READ（可重读）**并不会有任何性能损失。

InnoDB 存储引擎在 分布式事务 的情况下一般会用到**SERIALIZABLE(可串行化)**隔离级别。

锁

对MySQL的锁了解吗

当数据库有并发事务的时候，可能会产生数据的不一致，这时候需要一些机制来

保证访问的次序，锁机制就是这样的一个机制。

就像酒店的房间，如果大家随意进出，就会出现多人抢夺同一个房间的情况，而在房间上装上锁，申请到钥匙的人才可以入住并且将房间锁起来，其他人只有等他使用完毕才可以再次使用。

隔离级别与锁的关系

在Read Uncommitted级别下，读取数据不需要加共享锁，这样就不会跟被修

改的数据上的排他锁冲突

在Read Committed级别下，读操作需要加共享锁，但是在语句执行完以后释放共享锁；

在Repeatable Read级别下，读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。

SERIALIZABLE 是限制性最强的隔离级别，因为该级别锁定整个范围的键，并一直持有锁，直到事务完成。

按照锁的粒度分数据库锁有哪些？锁机制与InnoDB锁算法

在关系型数据库中，可以按照锁的粒度把数据库锁分为行级锁(INNODB引擎)、表级锁(MYISAM引擎)和页级锁(BDB引擎)。

MyISAM和InnoDB存储引擎使用的锁：

- MyISAM采用表级锁(table-level locking)。
- InnoDB支持行级锁(row-level locking)和表级锁，默认为行级锁

行级锁，表级锁和页级锁对比

行级锁 行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁 和 排他锁。

特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，

并发度也最高。

表级锁 表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

特点：开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，

并发度最低。

页级锁 页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和

行锁之间，并发度一般

从锁的类别上分MySQL都有哪些锁呢?

像上面那样子进行锁定岂不是有点阻碍并发效率了

从锁的类别上来讲，有共享锁和排他锁。

共享锁: 又叫做读锁。 当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个。

排他锁: 又叫做写锁。 当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

用上面的例子来说就是用户的行为有两种，一种是来看房，多个用户一起看房是可以接受的。 一种是真正的入住一晚，在这期间，无论是想入住的还是想看房的都不可以。

锁的粒度取决于具体的存储引擎，InnoDB实现了行级锁，页级锁，表级锁。

他们的加锁开销从大到小，并发能力也是从大到小。

MySQL中InnoDB引擎的行锁是怎么实现的?

答：InnoDB是基于索引来完成行锁

例: select * from tab_with_index where id = 1 for update;

for update 可以根据条件来完成行锁锁定，并且 id 是有索引键的列，如果 id不是索引键那么InnoDB将完成表锁，并发将无从谈起

InnoDB存储引擎的锁的算法有三种

- Record lock：单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，不包括记录本身
- Next-key lock：record+gap 锁定一个范围，包含记录本身

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决Phantom Problem幻读问题
3. 当查询的索引含有唯一属性时，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生
5. 有两种方式显式关闭gap锁：（除了外键约束和唯一性检查外，其余情况仅使用record lock） A. 将事务隔离级别设置为RC B. 将参数innodb_locks_unsafe_for_binlog设置为1

什么是死锁？怎么解决?

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。

常见的解决死锁的方法

1、如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

2、在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

3、对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

如果业务处理不好可以用分布式事务锁或者使用乐观锁

数据库的乐观锁和悲观锁是什么？怎么实现的?

数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。乐观并发控制（乐观锁）和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。在修改数据的时候把事务锁起来，通过version的方式来进行锁定。实现方式：乐一般会使用版本号机制或CAS算法实现。

两种锁的使用场景从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。

但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行

retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

视图

为什么要使用视图？什么是视图?

为了提高复杂SQL语句的复用性和表操作的安全性，MySQL数据库管理系统提供了视图特性。所谓视图，本质上是一种虚拟表，在物理上是不存在的，其内容与真实的表相似，包含一系列带有名称的列和行数据。但是，视图并不在数据库中以储存的数据值形式存在。行和列数据来自定义视图的查询所引用基本表，并且在具体引用视图时动态生成。

视图使开发者只关心感兴趣的某些特定数据和所负责的特定任务，只能看到视图中所定义的数据，而不是视图所引用表中的数据，从而提高了数据库中数据的安全性

视图有哪些特点?

视图的特点如下:

- 视图的列可以来自不同的表，是表的抽象和在逻辑意义上建立的新关系。
- 视图是由基本表(实表)产生的表(虚表)。
- 视图的建立和删除不影响基本表。
- 对视图内容的更新(添加，删除和修改)直接影响基本表。
- 当视图来自多个基本表时，不允许添加和删除数据。视图的操作包括创建视图，查看视图，删除视图和修改视图。

视图的使用场景有哪些?

视图根本用途：简化sql查询，提高开发效率。如果说还有另外一个用途那就是兼容老的表结构。

下面是视图的常见使用场景：重用SQL语句；

- 简化复杂的SQL操作。在编写查询后，可以方便的重用它而不必知道它的基本查询细节；使用表的组成部分而不是整个表；
- 保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限；
- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据

视图的优点

1. 查询简单化。视图能简化用户的操作
2. 数据安全性。视图使用户能以多种角度看待同一数据，能够对机密数据提供安全保护

逻辑数据独立性。视图对重构数据库提供了一定程度的逻辑独立性

视图的缺点

1. 性能。数据库必须把视图的查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，即使是视图的一个简单查询，数据库也把它变成一个复杂的结合体，需要花费一定的时间。
2. 修改限制。当用户试图修改视图的某些行时，数据库必须把它转化为对基本表的某些行的修改。事实上，当从视图中插入或者删除时，情况也是这样。对于简单视图来说，这是很方便的，但是，对于比较复杂的视图，可能是不可修改的这些视图有如下特征：

    1. 有UNIQUE等集合操作符的视图。
    2.有GROUP BY子句的视图。
    3.有诸如AVG\SUM\MAX等聚合函数的视图。

4.使用DISTINCT关键字的视图。
5.连接表的视图（其中有些例外）

什么是游标?

游标是系统为用户开设的一个数据缓冲区，存放SQL语句的执行结果，每个游标区都有一个名字。用户可以通过游标逐一获取记录并赋给主变量，交由主语言进一步处理

存储过程与函数

什么是存储过程？有哪些优缺点?

存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需要创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。

优点

1） 存储过程是预编译过的，执行效率高。

2） 存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯。

3） 安全性高，执行存储过程需要有一定权限的用户。

4） 存储过程可以重复使用，减少数据库开发人员的工作量。

缺点

1） 调试麻烦，但是用 PL/SQL Developer 调试很方便！弥补这个缺点。

2） 移植问题，数据库端代码当然是与数据库相关的。但是如果是做工程型项目，基本不存在移植问题。

3） 重新编译问题，因为后端代码是运行前编译的，如果带有引用关系的对象发生改变时，受影响的存储过程、包将需要重新编译（不过也可以设置成运行时刻自动编译）。

4） 如果在一个程序系统中大量的使用存储过程，到程序交付使用的时候随着用户需求的增加会导致数据结构的变化，接着就是系统的相关问题了， 后如果用

户想维护该系统可以说是很难很难、而且代价是空前的，维护起来更麻烦。

触发器

什么是触发器？触发器的使用场景有哪些?

触发器是用户定义在关系表上的一类由事件驱动的特殊的存储过程。触发器是指一段代码，当触发某个事件时，自动执行这些代码。

使用场景

- 可以通过数据库中的相关表实现级联更改。
- 实时监控某张表中的某个字段的更改而需要做出相应的处理。
- 例如可以生成某些业务的编号。
- 注意不要滥用，否则会造成数据库及应用程序的维护困难。
- 大家需要牢记以上基础知识点，重点是理解数据类型CHAR和VARCHAR的差异，表存储引擎InnoDB和MyISAM的区别。

MySQL中都有哪些触发器?

- Before Insert
- After Insert
- Before Update
- After Update
- Before Delete
- After Delete

常用SQL语句

SQL语句主要分为哪几类

数据定义语言DDL（Data Ddefinition Language）CREATE，DROP，ALTER 主要为以上操作 即对逻辑结构等有操作的，其中包括表结构，视图和索引。

数据查询语言DQL（Data Query Language）SELECT

这个较为好理解 即查询操作，以select关键字。各种简单查询，连接查询等 都属于DQL。

数据操纵语言DML（Data Manipulation Language）INSERT，UPDATE，

DELETE

主要为以上操作 即对数据进行操作的，对应上面所说的查询操作 DQL与DML共同构建了多数初级程序员常用的增删改查操作。而查询是较为特殊的一种 被划分到DQL中。

数据控制功能DCL（Data Control Language）GRANT，REVOKE，

COMMIT，ROLLBACK

主要为以上操作 即对数据库安全性完整性等有操作的，可以简单的理解为权限控制等

超键、候选键、主键、外键分别是什么?

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

```
候选键：是　小超键，即没有冗余元素的超键。
```

主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。

一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

```
外键：在一个表中存在的另一个表的主键称此表的外键。
```

SQL约束有哪几种?

SQL 约束有哪几种?

NOT NULL: 用于控制字段的内容一定不能为空（NULL）。

UNIQUE: 控件字段内容不能重复，一个表允许有多个 Unique 约束。

PRIMARY KEY: 也是用于控件字段内容不能重复，但它在一个表只允许出现一个。

FOREIGN KEY: 用于预防破坏表之间连接的动作，也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。

CHECK： 用于控制字段的值范围。

六种关联查询

交叉连接（CROSS JOIN）

内连接（INNER JOIN）

外连接（LEFT JOIN/RIGHT JOIN）

联合查询（UNION与UNION ALL）

全连接（FULL JOIN）

交叉连接（CROSS JOIN）

SELECT * FROM A,B(,C)或者SELECT * FROM A CROSS JOIN B (CROSS JOIN C)#没有任何关联条件，结果是笛卡尔积，结果集会很大，没有意义，很少使用内连接（INNER

JOIN）SELECT * FROM A,B WHERE A.id=B.id或者SELECT * FROM A INNER JOIN B

ON A.id=B.id多表中同时符合某种条件的数据记录的集合，INNER JOIN可以缩写为JOIN

内连接分为三类

等值连接：ON A.id=B.id 不等值连接：ON A.id > B.id

自连接：SELECT * FROM A T1 INNER JOIN A T2 ON T1.id=T2.pid

外连接（LEFT JOIN/RIGHT JOIN）

左外连接：LEFT OUTER JOIN, 以左表为主，先查询出左表，按照ON后的关联

条件匹配右表，没有匹配到的用NULL填充，可以简写成LEFT JOIN

右外连接：RIGHT OUTER JOIN, 以右表为主，先查询出右表，按照ON后的关

联条件匹配左表，没有匹配到的用NULL填充，可以简写成RIGHT JOIN

联合查询（UNION与UNION ALL）

SELECT * FROM A UNION SELECT * FROM B UNION ...

1

就是把多个结果集集中在一起，UNION前的结果为基准，需要注意的是联合查

询的列数要相等，相同的记录行会合并如果使用UNION ALL，不会合并重复的记录行效率 UNION 高于 UNION ALL

全连接（FULL JOIN）

MySQL不支持全连接

可以使用LEFT JOIN 和UNION和RIGHT JOIN联合使用

SELECT * FROM A LEFT JOIN B ON A.id=B.id UNION SELECT * FROM A RIGHT JOIN B ON A.id=B.id

1

表连接面试题有2张表，1张R、1张S，R表有ABC三列，S表有CD两列，表中各有三条记录。

R表

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c3

S表

C	D
c1	d1
c2	d2
c4	d3

1. 交叉连接(笛卡尔积):

select r.*,s.* from r,s

A	B	C	C	D
a1	b1	c1	c1	d1
a2	b2	c2	c1	d1
a3	b3	c3	c1	d1
a1	b1	c1	c2	d2
a2	b2	c2	c2	d2
a3	b3	c3	c2	d2
a1	b1	c1	c4	d3
a2	b2	c2	c4	d3
a3	b3	c3	c4	d3

1. 内连接结果:

select r.*,s.* from r inner join s on r.c=s.c

A	B	C	C	D
a1	b1	c1	c1	d1
a2	b2	c2	c2	d2

1. 左连接结果:

select r.*,s.* from r left join s on r.c=s.c

A	B	C	C	D
a1	b1	c1	c1	d1
a2	b2	c2	c2	d2
a3	b3	c3		

1. 右连接结果：

select r.*,s.* from r right join s on r.c=s.c

A	B	C	C	D
a1	b1	c1	c1	d1
a2	b2	c2	c2	d2
c4	d3			

1. 全表连接的结果（MySql不支持，Oracle支持）：

select r.*,s.* from r full join s on r.c=s.c

A	B	C	C	D
a1	b1	c1	c1	d1
a2	b2	c2	c2	d2
a3	b3	c3		
c4	d3			

什么是子查询

1. 条件：一条SQL语句的查询结果做为另一条查询语句的条件或查询结果
2. 嵌套：多条SQL语句嵌套使用，内部的SQL查询语句称为子查询。

子查询的三种情况

1. 子查询是单行单列的情况：结果集是一个值，父查询使用：=、 <、 > 等运算符

```
1   -- 查询工资最高的员工是谁？
2   select * from employee where salary=(select max(salary) from employee);
```

2. 子查询是多行单列的情况：结果集类似于一个数组，父查询使用：in 运算符

```
1   -- 查询工资最高的员工是谁？
2   select * from employee where salary=(select max(salary) from employee);
```

3. 子查询是多行多列的情况：结果集类似于一张虚拟表，不能用于where 条件，用于select子句中做为子表

```
1   -- 1) 查询出2011年以后入职的员工信息
2   -- 2) 查询所有的部门信息，与上面的虚拟表中的信息比对，找出所有部门ID相等的员工。
3   select * from dept d, (select * from employee where join_date >'2011-1-
1') e where e.dept_id = d.id;
4   -- 使用表连接：
5   select d.*, e.* from dept d inner join employee e on d.id = e.dept_id whe re
e.join_date >'2011-1-1'
```

mysql中in和exists区别

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop

循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

1. 如果查询的两个表大小相当，那么用in和exists差别不大。
2. 如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in。
3. not in 和not exists：如果查询语句使用了not in，那么内外表都进行全表扫描，没有用到索引；而not extsts的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

varchar与char的区别

char的特点

char表示定长字符串，长度是固定的；如果插入数据的长度小于char的固定长度时，则用空格填充；

因为长度固定，所以存取速度要比varchar快很多，甚至能快50%，但正因为其长度固定，所以会占据多余的空间，是空间换时间的做法；

对于char来说， 多能存放的字符个数为255，和编码无关 varchar的特点

varchar表示可变长字符串，长度是可变的；

插入的数据是多长，就按照多长来存储；

varchar在存取方面与char相反，它存取慢，因为长度不固定，但正因如此，不占据多余的空间，是时间换空间的做法；

对于varchar来说， 多能存放的字符个数为65532 总之，结合性能角度（char更快）和节省磁盘空间角度（varchar更小），具体情况还需具体来设计数据库才是妥当的做法。

varchar(50)中50的涵义

多存放50个字符，varchar(50)和(200)存储hello所占空间一样，但后者在排

序时会消耗更多内存，因为order by col采用fixed_length计算col长度

(memory引擎也一样)。在早期 MySQL 版本中， 50 代表字节数，现在代表字符数。

int(20)中20的涵义

是指显示字符的长度。20表示 大显示宽度为20，但仍占4字节存储，存储范围不变；

不影响内部存储，只是影响带 zerofill 定义的 int 时，前面补多少个 0，易于报表展示

mysql为什么这么设计

对大多数应用没有意义，只是规定一些工具用来显示字符的个数；int(1)和 int(20)存储和计算均一样；

mysql中int(10)和char(10)以及varchar(10)的区别

- int(10)的10表示显示的数据的长度，不是存储数据的大小； chart(10)和varchar(10)的10表示存储数据的大小，即表示存储多少个字符。

int(10) 10位的数据长度 9999999999，占32个字节，int型4位 char(10) 10位固定字符串，不足补空格多10个字符

varchar(10) 10位可变字符串，不足补空格 多10个字符

- char(10)表示存储定长的10个字符，不足10个就用空格补齐，占用更多的存储空间
- varchar(10)表示存储10个变长的字符，存储多少个就是多少个，空格也按一个字符存储，这一点是和char(10)的空格不同的，char(10)的空格表示占位不算一个字符

FLOAT和DOUBLE的区别是什么?

FLOAT类型数据可以存储至多8位十进制数，并在内存中占4字节。

DOUBLE类型数据可以存储至多18位十进制数，并在内存中占8字节。

drop、delete与truncate的区别

三者都表示删除，但是三者有一些差别：

Delete	Truncate	Drop	
类型	属于DML	属于DDL	属于DDL
回滚	可回滚	不可回滚	不可回滚
删除内容	表结构还在，删除表的全部或者一部分数据行	表结构还在，删除表中的所有数据	从数据库中删除表，所有的数据行，索引和权限也会被删除
删除速度	删除速度慢，需要逐行删除	删除速度快	删除速度快

因此，在不再需要一张表的时候，用drop；在想删除部分数据行时候，用 delete；在保留表而删除所有数据的时候用truncate。

UNION与UNIONALL的区别?

如果使用UNION ALL，不会合并重复的记录行

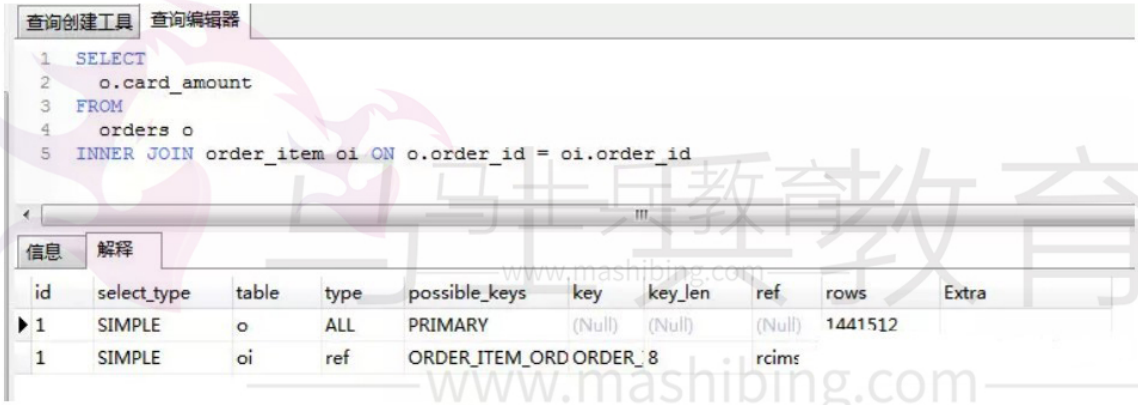
效率 UNION 高于 UNION ALL

SQL优化

如何定位及优化SQL语句的性能问题?

创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?

对于低性能的SQL语句的定位， 重要也是 有效的方法就是使用执行计划， MySQL提供了explain命令来查看语句的执行计划。 我们知道，不管是哪种数据库，或者是哪种数据库引擎，在对一条SQL语句进行执行的过程中都会做很多相关的优化，对于查询语句，最重要的优化方式就是使用索引。 而执行计划，就是显示数据库引擎对于SQL语句的执行的详细情况，其中包含了是否使用索引，使用什么索引，使用的索引的相关信息等。



执行计划包含的信息 id 有一组数字组成。表示一个查询中各个子查询的执行顺序;

- id相同执行顺序由上至下。
- id不同，id值越大优先级越高，越先被执行。
- id为null时表示一个结果集，不需要使用它查询，常出现在包含union等查询语句中。

select_type 每个子查询的查询类型，一些常见的查询类型。

id	select_ty pe	descripti on
1	SIMPLE	不包含任何子查询 或union等查询
2	PRIMARY	包含子查询 外层查询就显示为 PRIMARY
3	SUBQUER Y	在select或 where字句中包含的查询
4	DERIVED	from字句中包含的查询
5	UNION	出现在 union后的查询语句中
6	UNION RESULT	从UNION 中获取结果集，例如上文的
		第三个例子

table 查询的数据表，当从衍生表中查数据时会显示 x 表示对应的执行计划id

partitions 表分区、表创建的时候可以指定通过那个列进行表分区。 举个例子:

```
1   create table tmp (
2   id int unsigned not null AUTO_INCREMENT,
3   name varchar(255),
4   PRIMARY KEY(id))
5   engine=innodb partition by key(id) partitions 5;
```

type(非常重要，可以看到有没有走索引) 访问类型

ALL 扫描全表数据 index 遍历索引 range 索引范围查找 index_subquery 在子查询中使用 ref unique_subquery 在子查询中使用 eq_ref ref_or_null 对Null进行索引的优化的 ref fulltext 使用全文索引

ref 使用非唯一索引查找数据 eq_ref 在join查询中使用PRIMARY KEYorUNIQUE NOT NULL索引关联。

possible_keys 可能使用的索引，注意不一定会使用。查询涉及到的字段上若存在索引，则该索引将被列出来。当该列为 NULL时就要考虑当前的SQL是否需要优化了。

key 显示MySQL在查询中实际使用的索引，若没有使用索引，显示为NULL。 TIPS:查询中若使用了覆盖索引(覆盖索引：索引的数据覆盖了需要查询的所有数据)，则该索引仅出现在key列表中

key_length 索引长度

ref 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值 rows 返回估算的结果集数目，并不是一个准确的值。

extra 的信息非常丰富，常见的有：

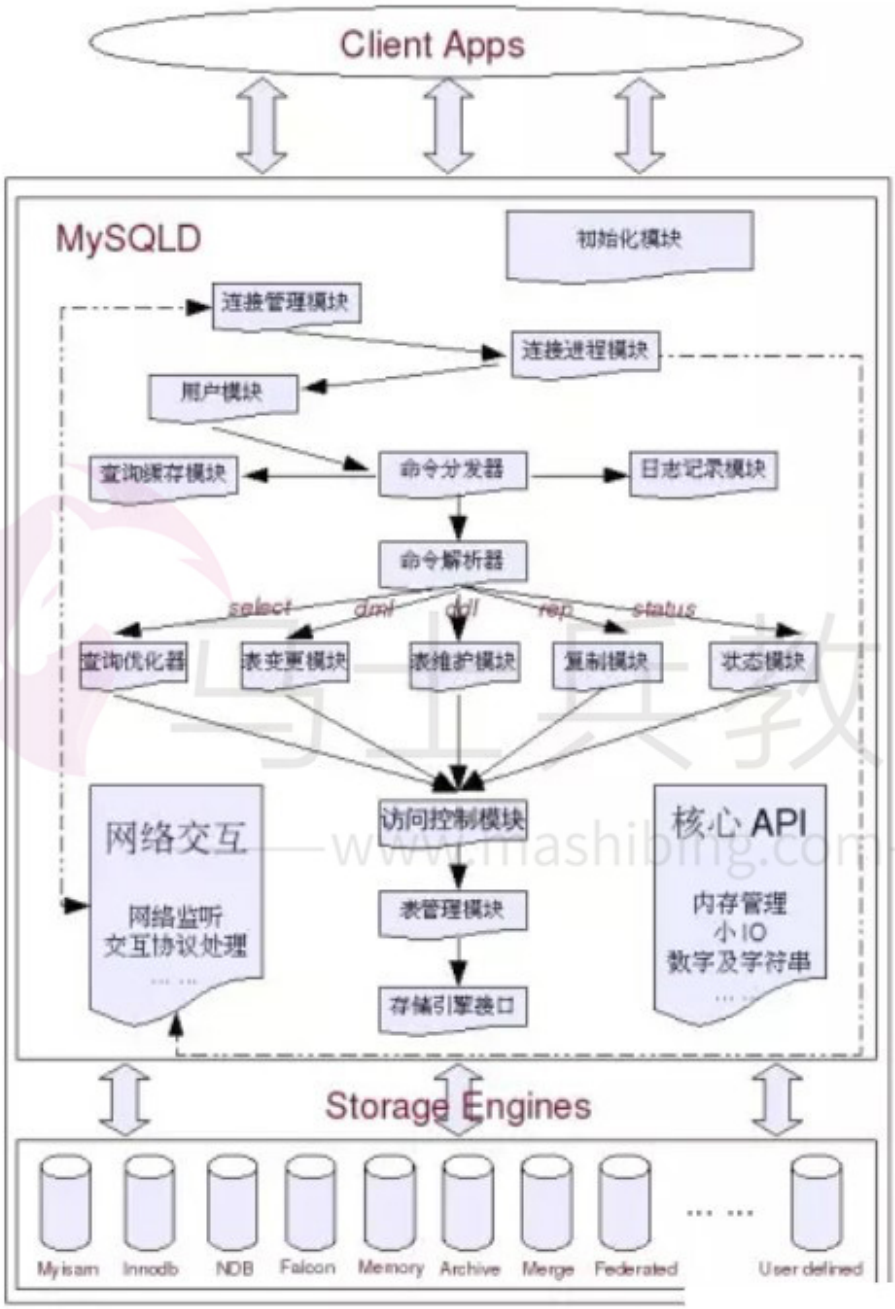
1. Using index 使用覆盖索引
2. Using where 使用了用where子句来过滤结果集
3. Using filesort 使用文件排序，使用非索引列进行排序时出现，非常消耗性能，尽量优化。
4. Using temporary 使用了临时表 sql优化的目标可以参考阿里开发手册

【推荐】SQL性能优化的目标：至少要达到 range 级别，要求是ref级别，如果可以是

consts 好。 说明： 1） consts 单表中 多只有一个匹配行（主键或者唯一索引），在优化阶段即可读取到数据。 2） ref 指的是使用普通的索引（normal index）。 3） range 对索引进行范围检索。 反例：explain表的结果，type=index，索引物理文件全扫描，速度非常慢，这个index级别比较range还低，与全表扫描是小巫见大巫。

SQL的生命周期?

1. 应用服务器与数据库服务器建立一个连接
2. 数据库进程拿到请求sql
3. 解析并生成执行计划，执行
4. 读取数据到内存并进行逻辑处理
5. 通过步骤一的连接，发送结果到客户端
6. 关掉连接，释放资源



大表数据查询，怎么优化

1. 优化shema、sql语句+索引；
2. 第二加缓存，memcached, redis；
3. 主从复制，读写分离；
4. 垂直拆分，根据你模块的耦合度，将一个大的系统分为多个小的系统，也就是分布式系统；
5. 水平切分，针对数据量大的表，这一步 麻烦， 能考验技术水平，要选择一个合理的sharding key，为了有好的查询效率，表结构也要改动，

做一定的冗余，应用也要改，sql中尽量带sharding key，将数据定位到限定的表上去查，而不是扫描全部的表；

超大分页怎么处理?

超大的分页一般从两个方向上来解决.

- 数据库层面,这也是我们主要集中关注的(虽然收效没那么大),类似于select * from table where age > 20 limit 1000000,10这种查询其实也是有可以优化的余地的. 这条语句需要load1000000数据然后基本上全部丢弃,只取10条当然比较慢. 当时我们可以修改为select * from table where id in (select id from table where age > 20 limit 1000000,10).这样虽然也load了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快. 同时如果ID连续的好,我们还可以select * from table where id > 1000000 limit 10,效率也是不错的,优化的可能性有许多种, 但是核心思想都一样,就是减少load的数据.
- 从需求的角度减少这种请求...主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止ID泄漏且连续被人恶意攻击.

解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至redis等k-V数据库中,直接返回即可.

在阿里巴巴《Java开发手册》中,对超大分页的解决办法是类似于上面提到的第一种.

【推荐】利用延迟关联或者子查询优化超多分页场景。 说明：MySQL并不是跳过offset

行，而是取offset+N行，然后返回放弃前offset行，返回N行，那当offset特别大的时候，效率就非常的低下，要么控制返回的总页数，要么对超过特定阈值的页数进行SQL改写。 正例：先快速定位需要获取的id段，然后再关联：

`SELECT a.* FROM 表1 a,(select id from 表1 where 条件 LIMIT 100000,20) b w here a.id=b.id`

mysql分页

LIMIT 子句可以被用于强制 SELECT 语句返回指定的记录数。LIMIT 接受一个或两个数字参数。参数必须是一个整数常量。如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的 大数目。初始记录行的偏移量是 0(而不是 1)

mysql> SELECT * FROM table LIMIT 5,10; // 检索记录行 6-15

```
        1
```

为了检索从某一个偏移量到记录集的结束所有的记录行，可以指定第二个参数为

-1：

mysql> SELECT * FROM table LIMIT 95,-1; // 检索记录行 96-last.

```
        1
```

如果只给定一个参数，它表示返回 大的记录行数目： mysql> SELECT * FROM table LIMIT 5; //检索前 5 个记录行

```
        1
```

换句话说，LIMIT n 等价于 LIMIT 0,n。

慢查询日志

用于记录执行时间超过某个临界值的SQL日志，用于快速定位慢查询，为我们的优化做参考。

开启慢查询日志

配置项：slow_query_log 可以使用show variables like ‘slov_query_log’查看是否开启，如果状态值为

OFF，可以使用set GLOBAL slow_query_log = on来开启，它会在datadir下产生一个xxx-slow.log的文件。

设置临界时间

配置项：long_query_time 查看：show VARIABLES like 'long_query_time'，单位秒设置：set long_query_time=0.5

实操时应该从长时间设置到短的时间，即将 慢的SQL优化掉

查看日志，一旦SQL超过了我们设置的临界时间就会被记录到xxx-slow.log中

关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？

慢查询都怎么优化过？

在业务系统中，除了使用主键进行的查询，其他的我都会在测试库上测试其耗时，慢查询的统计主要由运维在做，会定期将业务中的慢查询反馈给我们。慢查询的优化首先要搞明白慢的原因是什么？ 是查询条件没有命中索引？是 load了不需要的数据列？还是数据量太大？所以优化也是针对这三个方向来的，

首先分析语句，看看是否load了额外的数据，可能是查询了多余的行并且抛弃掉了，可能是加载了许多结果中并不需要的列，对语句进行分析以及重写。

分析语句的执行计划，然后获得其使用索引的情况，之后修改语句或者修改索引，使得语句可以尽可能的命中索引。

> 如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行横向或者纵向的分表。

为什么要尽量设定一个主键？

主键是数据库确保数据行在整张表唯一性的保障，即使业务上本张表没有主键，也建议添加一个自增长的ID列作为主键。设定了主键之后，在后续的删改查的时候可能更加快速以及确保操作数据范围安全。

主键使用自增ID还是UUID？

推荐使用自增ID，不要使用UUID。

因为在InnoDB存储引擎中，主键索引是作为聚簇索引存在的，也就是说，主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序)，如果主键索引是自增ID，那么只需要不断向后排列即可，如果是UUID，由于到来的ID与原来的大小不确定，会造成非常多的数据插入，数据移动，然后导致产生很多的内存碎片，进而造成插入性能的下降。

总之，在数据量大一些的情况下，用自增主键性能会好一些。

关于主键是聚簇索引，如果没有主键，InnoDB会选择一个唯一键来作为聚簇索引，如果没有唯一键，会生成一个隐式的主键。

字段为什么要求定义为not null?

null值会占用更多的字节，且会在程序中造成很多与预期不符的情况。

如果要存储用户的密码散列，应该使用什么字段进行存储?

密码散列，盐，用户身份证号等固定长度的字符串应该使用char而不是varchar 来存储，这样可以节省空间且提高检索效率。

优化查询过程中的数据访问

访问数据太多导致查询性能下降确定应用程序是否在检索大量超过需要的数据，可能是太多行或列确认MySQL服务器是否在分析大量不必要的数据行避免犯如下SQL语句错误

查询不需要的数据。解决办法：使用limit解决

多表关联返回全部列。解决办法：指定列名

总是返回全部列。解决办法：避免使用SELECT * 重复查询相同的数据。解决办法：可以缓存数据，下次直接读取缓存是否在扫描额外的记录。解决办法：

使用explain进行分析，如果发现查询需要扫描大量的数据，但只返回少数的行，可以通过如下技巧去优化：

使用索引覆盖扫描，把所有的列都放到索引中，这样存储引擎不需要回表获取对应行就可以返回结果。

改变数据库和表的结构，修改数据表范式

重写SQL语句，让优化器可以以更优的方式执行查询。

优化长难的查询语句

一个复杂查询还是多个简单查询

MySQL内部每秒能扫描内存中上百万行数据，相比之下，响应数据给客户端就要慢得多

使用尽可能小的查询是好的，但是有时将一个大的查询分解为多个小的查询是很有必要的。

切分查询将一个大的查询分为多个小的相同的查询

一次性删除1000万的数据要比一次删除1万，暂停一会的方案更加损耗服务器开销。

分解关联查询，让缓存的效率更高。

执行单个查询可以减少锁的竞争。

在应用层做关联更容易对数据库进行拆分。

查询效率会有大幅提升。

较少冗余记录的查询

优化特定类型的查询语句

count()*会忽略所有的列，直接统计所有列数，不要使用count(列名) MyISAM中，没有任何where条件的count()*非常快。

当有where条件时，MyISAM的count统计不一定比其它引擎快。

可以使用explain查询近似值，用近似值替代count(*) 增加汇总表使用缓存

优化关联查询

确定ON或者USING子句中是否有索引。

确保GROUP BY和ORDER BY只有一个表中的列，这样MySQL才有可能使用索引

优化子查询

用关联查询替代

优化GROUP BY和DISTINCT

这两种查询据可以使用索引来优化，是 有效的优化方法关联查询中，使用标识列分组的效率更高

如果不需要ORDER BY，进行GROUP BY时加ORDER BY NULL，MySQL不会再进行文件排序。

WITH ROLLUP超级聚合，可以挪到应用程序处理

优化LIMIT分页

LIMIT偏移量大的时候，查询效率较低

可以记录上次查询的 大ID，下次查询时直接根据该ID来查询

优化UNION查询

UNION ALL的效率高于UNION

优化WHERE子句

解题方法对于此类考题，先说明如何定位低效SQL语句，然后根据SQL语句可能低效的原因做排查，先从索引着手，如果索引没有问题，考虑以上几个方面，数据访问的问题，长难查询句的问题还是一些特定类型优化的问题，逐一回答。

SQL语句优化的一些方法?

1.对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

2.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如:

```
1 select id from t where num is null -- 可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：select id from t where num=0
```

3.应尽量避免在 where 子句中使用!=或<>操作符，否则引擎将放弃使用索引而进行全表扫描。

4.应尽量避免在 where 子句中使用or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如:

```
1 select id from t where num=10 or num=20 -- 可以这样查询：select id from t where num=10 union all select id from t where num=20
```

```
        5.in 和 not in 也要慎用，否则会导致全表扫描，如：
```

```
1 select id from t where num in(1,2,3) -- 对于连续的数值，能用 between 就不要用 in 了：select id from t where num between 1 and 3
```

6.下面的查询也将导致全表扫描：select id from t where name like '% 李%'若要提高效率，可以考虑全文检索。

7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

```
1 select id from t where num=@num --可以改为强制查询使用索引：select id from t with (index(索引名)) where num=@num
```

8.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
1 select id from t where num/2=100 -- 应改为:select id from t where num=100*2
```

9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
1 select id from t where substring(name,1,3)='abc' -- name以abc开头的id应改为: select id from t where name like 'abc%'
```

10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

数据库优化

为什么要优化

系统的吞吐量瓶颈往往出现在数据库的访问速度上随着应用程序的运行，数据库的中的数据会越来越多，处理时间会相应变慢数据是存放在磁盘上的，读写速度无法和内存相比优化原则：减少系统瓶颈，减少资源占用，增加系统的反应速度。

数据库结构优化

一个好的数据库设计方案对于数据库的性能往往会起到事半功倍的效果。

需要考虑数据冗余、查询和更新的速度、字段的数据类型是否合理等多方面的内容。

将字段很多的表分解成多个表对于字段较多的表，如果有些字段的使用频率很低，可以将这些字段分离出来形成新表。

因为当一个表的数据量很大时，会由于使用频率低的字段的存在而变慢。

增加中间表对于需要经常联合查询的表，可以建立中间表以提高查询效率。

通过建立中间表，将需要通过联合查询的数据插入到中间表中，然后将原来的联合查询改为对中间表的查询。

增加冗余字段设计数据表时应尽量遵循范式理论的规约，尽可能的减少冗余字段，让数据库设计看起来精致、优雅。但是，合理的加入冗余字段可以提高查询速度。

表的规范化程度越高，表和表之间的关系越多，需要连接查询的情况也就越多，性能也就越差。

注意：

冗余字段的值在一个表中修改了，就要想办法在其他表中更新，否则就会导致数据不一致的问题

MySQL数据库cpu飙升到500%的话他怎么处理?

当 cpu 飙升到 500%时，先用操作系统命令 top 命令观察是不是 mysqld 占用导致的，如果不是，找出占用高的进程，并进行相关处理。

如果是 mysqld 造成的， show processlist，看看里面跑的 session 情况，是不是有消耗资源的 sql 在运行。找出消耗高的 sql，看看执行计划是否准确， index 是否缺失，或者实在是数据量太大造成。

一般来说，肯定要 kill 掉这些线程(同时观察 cpu 使用率是否下降)，等进行相应的调整(比如说加索引、改 sql、改内存参数)之后，再重新跑这些 SQL。

也有可能是每个 sql 消耗资源并不多，但是突然之间，有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整,比如说限制连接数等

大表怎么优化？某个表有近千万数据，

CRUD比较慢，如何优化？分库分表了是怎么

做的？分表分库了有什么问题？有用到中间

件么？他们的原理知道么？

当MySQL单表记录数过大时，数据库的CRUD性能会明显下降，一些常见的优化措施如下：

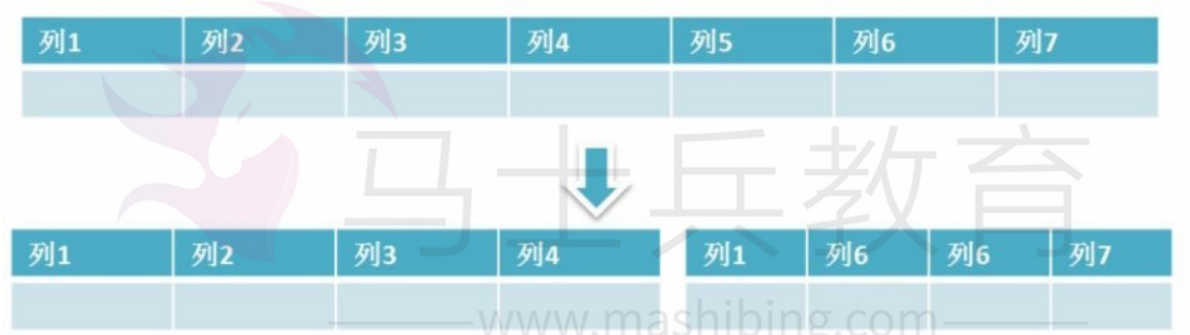
1. 限定数据的范围： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内。；
2. 读/写分离： 经典的数据库拆分方案，主库负责写，从库负责读；
3. 缓存： 使用MySQL的缓存，另外对重量级、更新少的数据可以考虑使用应用级别的缓存；

有就是通过分库分表的方式进行优化，主要有垂直分表和水平分表

1. 垂直分区：

根据数据库里面数据表的相关性进行拆分。 例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。 如下图所示，这样来说大家应该就更容易理解了。



马士兵教育
www.mashibing.com

垂直拆分的优点： 可以使得行数据变小，在查询时减少读取的Block数，减少

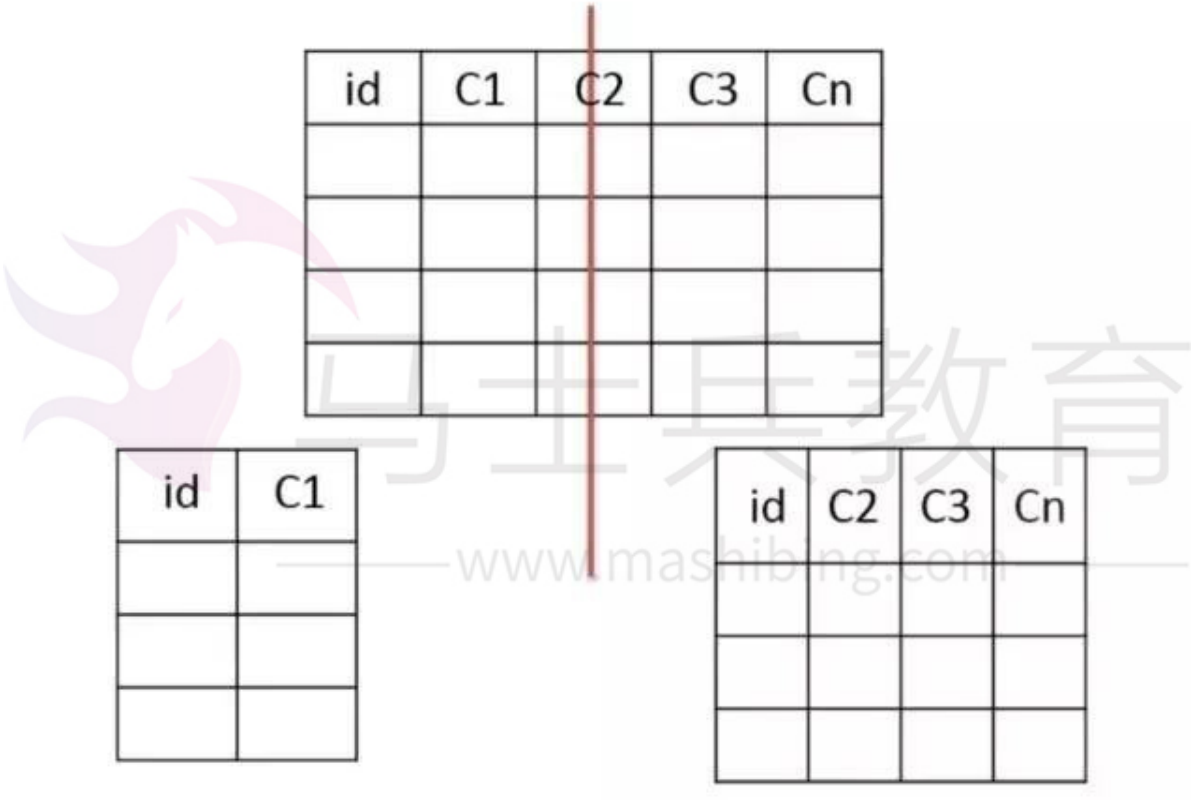
I/O次数。此外，垂直分区可以简化表的结构，易于维护。

垂直拆分的缺点： 主键会出现冗余，需要管理冗余列，并会引起Join操作，可

以通过在应用层进行Join来解决。此外，垂直分区会让事务变得更加复杂；

垂直分表

把主键和一些列放在一个表，然后把主键和另外的列放在另一个表中



适用场景

1、 如果一个表中某些列常用，另外一些列不常用

2、 可以使数据行变小，一个数据页能存储更多数据，查询时减少I/O次数

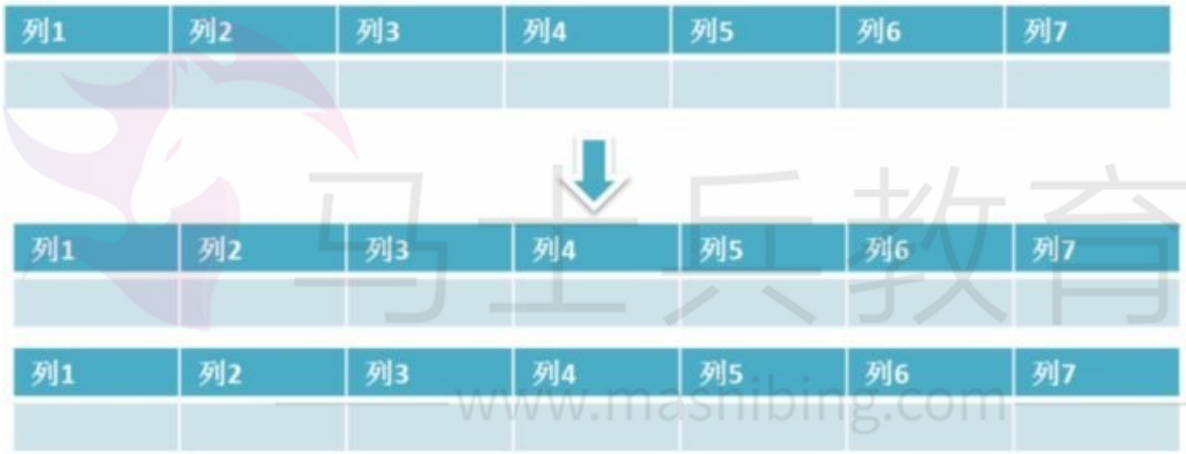
缺点

有些分表的策略基于应用层的逻辑算法，一旦逻辑算法改变，整个分表逻辑都会改变，扩展性较差 对于应用层来说，逻辑算法增加开发成本管理冗余列，查询所有数据需要join操作

2. 水平分区：

保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。 水平拆分可以支撑非常大的数据量。

水平拆分是指数据表行的拆分，表的行数超过200万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。



水品拆分可以支持非常大的数据量。需要注意的一点是:分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升

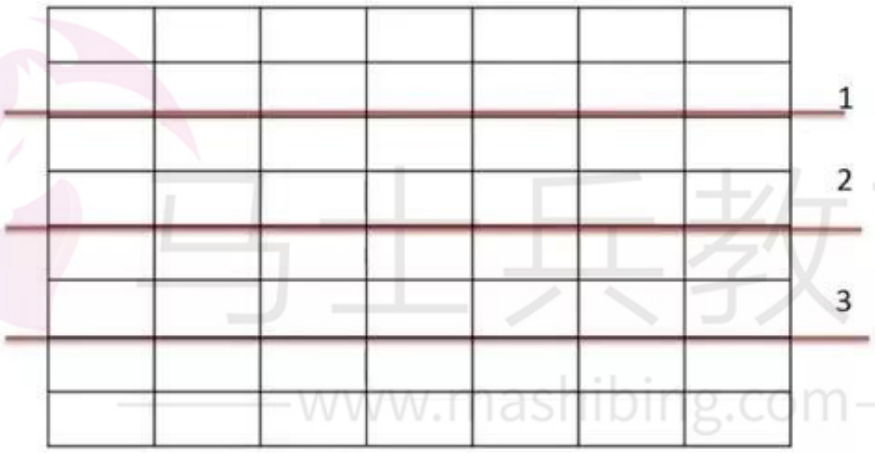
MySQL并发能力没有什么意义，所以 水平拆分最好分库 。

水平拆分能够 支持非常大的数据量存储，应用端改造也少，但 分片事务难以解决 ，跨界点Join性能较差，逻辑复杂。

《Java工程师修炼之道》的作者推荐 尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度 ，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络I/O。

水平分表:

表很大，分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询次数



马士兵教育
www.mashibing.com

适用场景

1、表中的数据本身就有独立性，例如表中分表记录各个地区的数据或者不同时期的数据，特别是有些数据常用，有些不常用。

2、需要把数据存放在多个介质上。

水平切分的缺点

1、给应用增加复杂度，通常查询时需要多个表名，查询所有数据都需UNION操作

2、在许多数据库应用中，这种复杂度会超过它带来的优点，查询时会增加读一个索引层的磁盘次数

下面补充一下数据库分片的两种常见方案:

客户端代理：分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。 当当网的Sharding-

JDBC 、阿里的TDDL是两种比较常用的实现。

中间件代理：在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。 我们现在谈的Mycat

、360的Atlas、网易的DDB等等都是这种架构的实现。

分库分表后面临的问题

事务支持 分库分表后，就成了分布式事务了。如果依赖数据库本身的分布式事务管理功能去执行事务，将付出高昂的性能代价； 如果由应用程序去协助控制，形成程序逻辑上的事务，又会造成编程方面的负担。

```
跨库join
```

只要是进行切分，跨节点Join的问题是不可避免的。但是良好的设计和切分却可以减少此类情况的发生。解决这一问题的普遍做法是分两次查询实现。在第一次查询的结果集中找出关联数据的id,根据这些id发起第二次请求得到关联数据。 分库分表方案产品

跨节点的count,order by,group by以及聚合函数问题 这些是一类问题，因为它们都需要基于全部数据集合进行计算。多数的代理都不会自动处理合并工作。解决方案：与解决跨节点join问题的类似，分别在各个节点上得到结果后在应用程序端进行合并。和join不同的是每个结点的查询可以并行执行，因此很多时候它的速度要比单一大表快很多。但如果结果集很大，对应用程序内存的消耗是一个问题。

数据迁移，容量规划，扩容等问题 来自淘宝综合业务平台团队，它利用对2的倍数取余具有向前兼容的特性（如对4取余得1的数对2取余也是 1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级别的迁移，同时对扩容规模和分表数量都有限制。总得来说，这些方案都不是

十分的理想，多多少少都存在一些缺点，这也从一个侧面反映出了

Sharding扩容的难度。

ID问题

一旦数据库被切分到多个物理结点上，我们将不能再依赖数据库自身的主键生成机制。一方面，某个分区数据库自生成的ID无法保证在全局上是唯一的；另一方面，应用程序在插入数据之前需要先获得ID,以便进行

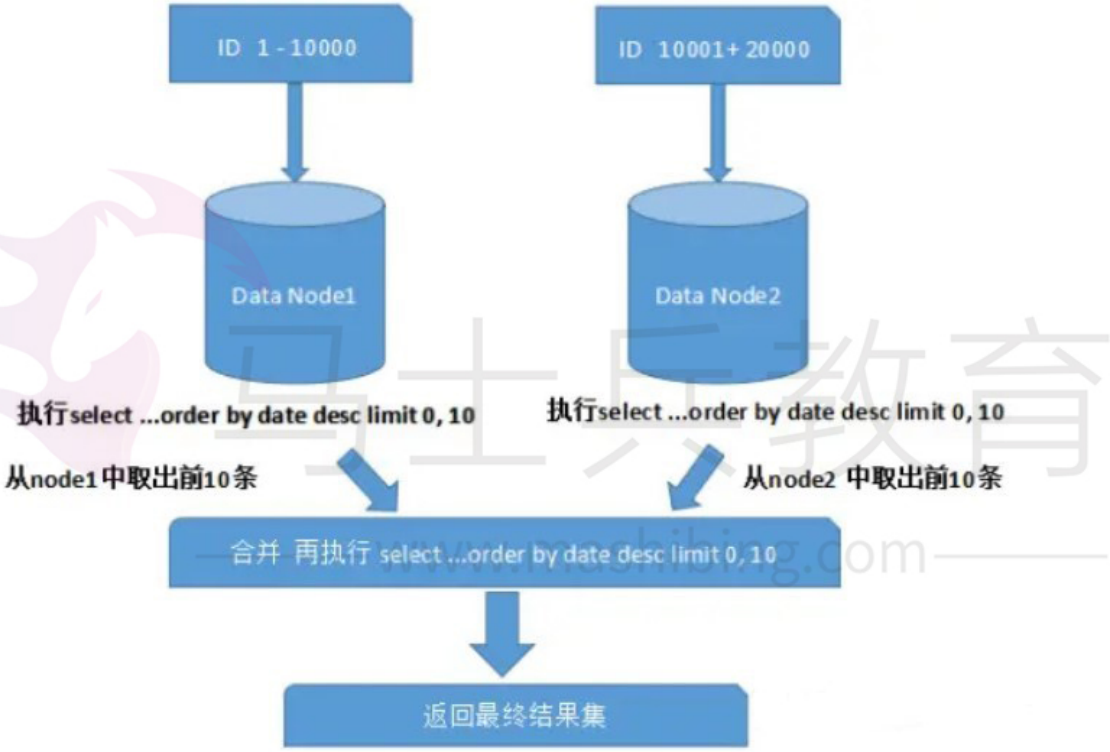
SQL路由. 一些常见的主键生成策略

UUID 使用UUID作主键是 简单的方案，但是缺点也是非常明显的。由于

UUID非常的长，除占用大量存储空间外， 主要的问题是在索引上，在建立索引和基于索引进行查询时都存在性能问题。 Twitter的分布式自增ID算法

Snowflake 在分布式系统中，需要生成全局UID的场合还是比较多的，twitter 的snowflake解决了这种需求，实现也还是很简单的，除去配置信息，核心代码就是毫秒级时间41位 机器ID 10位 毫秒内序列12位。

跨分片的排序分页般来讲，分页时需要按照指定字段进行排序。当排序字段就是分片字段的时候，我们通过分片规则可以比较容易定位到指定的分片，而当排序字段非分片字段的时候，情况就会变得比较复杂了。为了 终结果的准确性，我们需要在不同的分片节点中将数据进行排序并返回，并将不同分片返回的结果集进行汇总和再次排序， 后再返回给用户。如下图所示：



MySQL的复制原理以及流程

主从复制：将主数据库中的DDL和DML操作通过二进制日志（BINLOG）传输到从数据库上，然后将这些日志重新执行（重做）；从而使得从数据库的数据与主数据库保持一致。

主从复制的作用

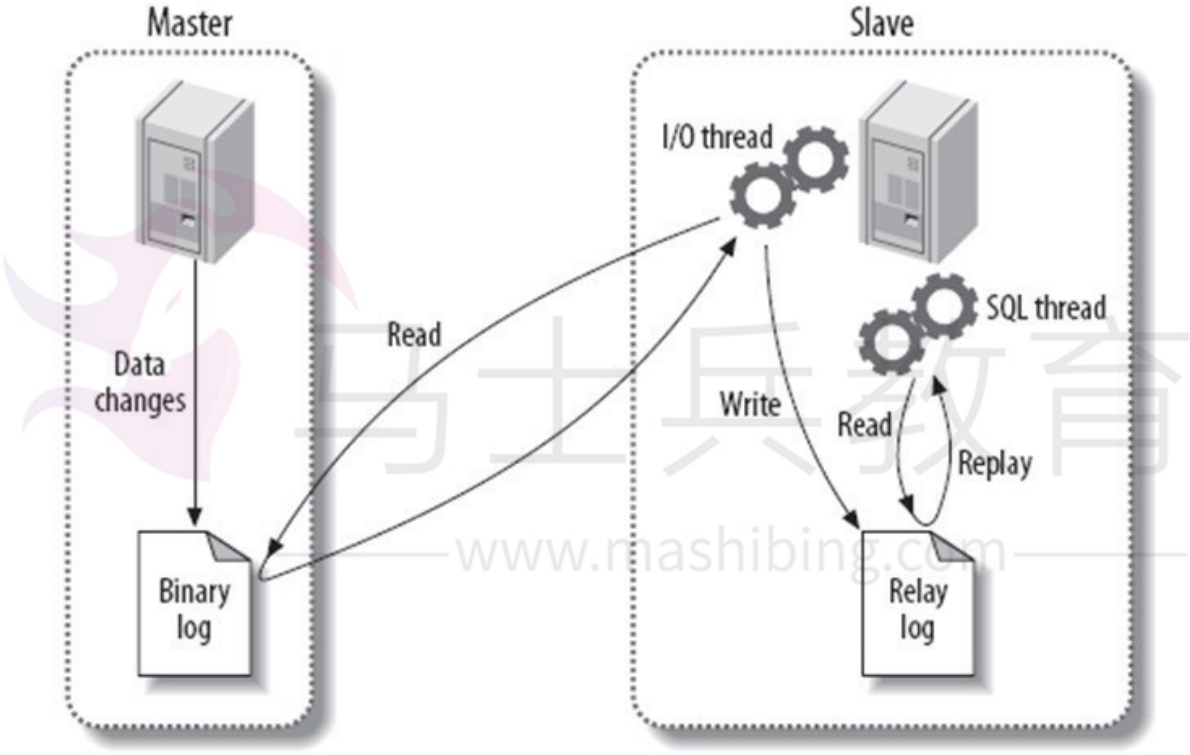
1. 主数据库出现问题，可以切换到从数据库。
2. 可以进行数据库层面的读写分离。
3. 可以在从数据库上进行日常备份。

MySQL主从复制解决的问题

- 数据分布：随意开始或停止复制，并在不同地理位置分布数据备份负载均衡：降低单个服务器的压力高可用和故障切换：帮助应用程序避免单点失败升级测试：可以用更高版本的MySQL作为从库
- MySQL主从复制工作原理
- 在主库上把数据更高记录到二进制日志从库将主库的日志复制到自己的中继日志
- 从库读取中继日志的事件，将其重放到从库数据中基本原理流程，3个线程以及之间的关联

主：binlog线程——记录下所有改变了数据库数据的语句，放进master上的 binlog中；从：io线程——在使用start slave 之后，负责从master上拉取 binlog 内容，放进自己的relay log中；

从：sql执行线程——执行relay log中的语句；



Binary log：主数据库的二进制日志

Relay log：从服务器的中继日志

第一步：master在每个事务更新数据完成之前，将该操作记录串行地写入到 binlog文件中。

第二步：salve开启一个I/O Thread，该线程在master打开一个普通连接，主要工作是binlog dump process。如果读取的进度已经跟上了master，就进入睡眠状态并等待master产生新的事件。I/O线程终的目的是将这些事件写入到中继日志中。

第三步：SQL Thread会读取中继日志，并顺序执行该日志中的SQL事件，从而与主数据库中的数据保持一致。

读写分离有哪些解决方案?

读写分离是依赖于主从复制，而主从复制又是为读写分离服务的。因为主从复制要求slave不能写只能读（如果对slave执行写操作，那么show slave status将会呈现Slave_SQL_Running=NO，此时你需要按照前面提到的手动同步一下slave）。

方案一

使用mysql-proxy代理

优点：直接实现读写分离和负载均衡，不用修改代码，master和slave用一样的帐号，mysql官方不建议实际生产中使用缺点：降低性能， 不支持事务方案二

使用AbstractRoutingDataSource+aop+annotation在dao层决定数据源。如果采用了mybatis， 可以将读写分离放在ORM层，比如mybatis可以通过

mybatis plugin拦截sql语句，所有的insert/update/delete都访问master库，所有的select 都访问salve库，这样对于dao层都是透明。 plugin实现时可以通过注解或者分析语句是读写方法来选定主从库。不过这样依然有一个问题， 也就是不支持事务， 所以我们还需要重写一下

DataSourceTransactionManager， 将read-only的事务扔进读库， 其余的有读有写的扔进写库。

方案三

使用AbstractRoutingDataSource+aop+annotation在service层决定数据源，可以支持事务. 缺点：类内部方法通过this.xx()方式相互调用时，aop不会进行拦截，需进行特殊处理

备份计划，mysqldump以及xtranbackup的实现原理

(1)备份计划视库的大小来定，一般来说 100G 内的库，可以考虑使用 mysqldump 来做，因为 mysqldump更加轻巧灵活，备份时间选在业务低峰期，可以每天进行都进行全量备份(mysqldump 备份出来的文件比较小，压缩之后更小)。

100G 以上的库，可以考虑用 xtranbackup 来做，备份速度明显要比

mysqldump 要快。一般是选择一周一个全备，其余每天进行增量备份，备份时间为业务低峰期。

(2)备份恢复时间

物理备份恢复快，逻辑备份恢复慢

这里跟机器，尤其是硬盘的速率有关系，以下列举几个仅供参考

20G的2分钟（mysqldump）

80G的30分钟(mysqldump)

111G的30分钟（mysqldump)

288G的3小时（xtra)

3T的4小时（xtra)

逻辑导入时间一般是备份时间的5倍以上

(3)备份恢复失败如何处理首先在恢复之前就应该做足准备工作，避免恢复的时候出错。比如说备份之后的有效性检查、权限检查、空间检查等。如果万一报错，再根据报错的提示来进行相应的调整。

(4)mysqldump和xtrabackup实现原理 mysqldump mysqldump 属于逻辑备份。加入–single-transaction 选项可以进行一致性备份。后台进程会先设置 session 的事务隔离级别为 RR(SET SESSION

TRANSACTION ISOLATION LEVELREPEATABLE READ)，之后显式开启一个

事务(START TRANSACTION /*!40100 WITH CONSISTENTSNAPSHOT */)，这样就保证了该事务里读到的数据都是事务事务时候的快照。之后再把表的数据读取出来。如果加上–master-data=1 的话，在刚开始的时候还会加一个数据库的读锁(FLUSH TABLES WITH READ LOCK),等开启事务后，再记录下数据库此时 binlog 的位置(showmaster status)，马上解锁，再读取表的数据。等所有的数据都已经导完，就可以结束事务

Xtrabackup:

xtrabackup 属于物理备份，直接拷贝表空间文件，同时不断扫描产生的 redo 日志并保存下来。 后完成 innodb 的备份后，会做一个 flush engine logs 的

操作(老版本在有 bug，在5.6 上不做此操作会丢数据)，确保所有的 redo log 都已经落盘(涉及到事务的两阶段提交

概念，因为 xtrabackup 并不拷贝 binlog，所以必须保证所有的 redo log 都落盘，否则可能会丢 后一组提交事务的数据)。这个时间点就是 innodb 完成备份的时间点，数据文件虽然不是一致性的，但是有这段时间的 redo 就可以让数据文件达到一致性(恢复的时候做的事

情)。然后还需要 flush tables with read lock，把 myisam 等其他引擎的表给备份出来，备份完后解锁。这样就做到了完美的热备。

数据表损坏的修复方式有哪些?

使用 myisamchk 来修复，具体步骤：

1） 修复前将mysql服务停止。

2） 打开命令行方式，然后进入到mysql的/bin目录。

3） 执行myisamchk –recover 数据库所在路径/*.MYI使用repair table 或者 OPTIMIZE table命令来修复，REPAIR TABLE

table_name 修复表 OPTIMIZE TABLE table_name 优化表 REPAIR TABLE 用于修复被破坏的表。
OPTIMIZE TABLE 用于回收闲置的数据库空间，当表上的数据行被删除时，所占据的磁盘空间并没有立即被回收，使用了OPTIMIZE

TABLE命令后这些空间将被回收，并且对磁盘上的数据行进行重排（注意：是磁盘上，而非数据库）

MySQL记录binlog的方式主要包括三种模式？每种模式的优缺点是什么?

mysql复制主要有三种方式：基于SQL语句的复制(statement-based replication, SBR)，基于行 的复制(row-based replication, RBR)，混合模式复制(mixed-based replication, MBR)。对应 的，binlog的格式也有三种：STATEMENT，ROW，MIXED。

① STATEMENT模式（SBR） 每一条会修改数据的sql语句会记录到binlog中。优点是并不需要记录每一条sql语句和每一行的 数据变化，减少了binlog日志量，节约IO，提高性能。缺点是在某些情况下会导致master-slave 中的数据不一致(如sleep()函数， last_insert_id()，以及user-defined functions(udf)等会出现 问题)

② ROW模式（RBR） 不记录每条sql语句的上下文信息，仅需记录哪条数据被修改了，修改成什么样了。而且不会出 现某些特定情况下的存储过程、或function、或trigger的调用和触发无法被正确复制的问题。缺 点是会产生大量的日志，尤其是alter table的时候会让日志暴涨。

③ MIXED模式（MBR） 以上两种模式的混合使用，一般的复制使用STATEMENT模式保存binlog，对于STATEMENT 模式无法复制的操作使用ROW模式保存binlog，MySQL会根据执行的SQL语句选择日志保存方 式。

MySQL锁，悲观锁、乐观锁、排它锁、共享锁、表级锁、行级锁；

乐观锁

用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即 为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实 现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加1。当我 们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比 对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期 数据。

悲观锁

在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中synchronized很 相似，共享锁（读锁）和排它锁（写锁）是悲观锁的不同的实现

共享锁（读锁）

共享锁又叫做读锁，所有的事务只能对其进行读操作不能写操作，加上共享锁后在事务结束之前 其他事务只能再加共享锁，除此之外其他任何类型的锁都不能再加了。

排它锁（写锁）

若某个事物对某一行加上了排他锁，只能这个事务对其进行读写，在此事务结束之前，其他事务 不能对其进行加任何锁，其他进程可以读取,不能进行写操作，需等待其释放。

表级锁

innodb 的行锁是在有索引的情况下,没有索引的表是锁定全表的

行级锁

行锁又分共享锁和排他锁,由字面意思理解，就是给某一行加上锁，也就是一条记录加上锁。 注意：行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级 锁。

MySQL InnoDB中，乐观锁、悲观锁、共享锁、排它锁、行锁、表锁、死锁概念的理解

MySQL/InnoDB的加锁，一直是一个面试中常问的话题。例如，数据库如果有高并发请求，如何保证数据完整性？产生死锁问题如何排查并解决？我在工作过程中，也会经常用到，乐观锁，排它锁，等。于是今天就对这几个概念进行学习，屡屡思路，记录一下。

注：MySQL是一个支持插件式存储引擎的数据库系统。本文下面的所有介绍，都是基于InnoDB存储引擎，其他引擎的表现，会有较大的区别。

存储引擎查看

MySQL给开发者提供了查询存储引擎的功能，我这里使用的是MySQL5.6.4，可以使用：

```
SHOW ENGINES
```

乐观锁

用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加1。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。

举例

1、数据库表设计

三个字段，分别是`id,value、version`

```
select id,value,version from TABLE where id=#{id}
```

2、每次更新表中的value字段时，为了防止发生冲突，需要这样操作

```
update TABLE
set value=2,version=version+1
where id=#{id} and version=#{version};
```

悲观锁

与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。

说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。**共享锁和排它锁是悲观锁的不同的实现**，它俩都属于悲观锁的范畴。

使用，排它锁 举例

要使用悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。

我们可以使用命令设置MySQL为非autocommit模式：

```
set autocommit=0;

# 设置完autocommit后，我们就可以执行我们的正常业务了。具体如下：

# 1. 开始事务

begin;/begin work;/start transaction; (三者选一就可以)

# 2. 查询表信息

select status from TABLE where id=1 for update;

# 3. 插入一条数据

insert into TABLE (id,value) values (2,2);

# 4. 修改数据为

update TABLE set value=2 where id=1;

# 5. 提交事务

commit;/commit work;
```

共享锁

共享锁又称**读锁 read lock**，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。

如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获得共享锁的事务只能读数据，不能修改数据

打开第一个查询窗口

```
begin;/begin work;/start transaction;  (三者选一就可以)

SELECT * from TABLE where id = 1  lock in share mode;
```

然后在另一个查询窗口中，对id为1的数据进行更新

```
update  TABLE set name="www.souyunku.com" where id =1;
```

此时，操作界面进入了卡顿状态，过了超时间，提示错误信息

如果在超时前，执行 `commit`，此更新语句就会成功。

```
[SQL]update  test_one set name="www.souyunku.com" where id =1;
[Err] 1205 - Lock wait timeout exceeded; try restarting transaction
```

加上共享锁后，也提示错误信息

```
update  test_one set name="www.souyunku.com" where id =1 lock in share mode;
[SQL]update  test_one set name="www.souyunku.com" where id =1 lock in share
mode;
[Err] 1064 - You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'lock
in share mode' at line 1
```

在查询语句后面增加 `LOCK IN SHARE MODE`，Mysql会对查询结果中的每行都加共享锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请共享锁，否则会被阻塞。其他线程也可以读取使用了共享锁的表，而且这些线程读取的是同一个版本的数据。

加上共享锁后，对于`update,insert,delete`语句会自动加排它锁。

排它锁

排他锁 exclusive lock（也叫writer lock）又称**写锁**。

排它锁是悲观锁的一种实现，在上面悲观锁也介绍过。

若事务 1 对数据对象A加上X锁，事务 1 可以读A也可以修改A，其他事务不能再对A加任何锁，直到事物 1 释放A上的锁。这保证了其他事务在事物 1 释放A上的锁之前不能再读取和修改A。排它锁会阻塞所有的排它锁和共享锁

读取为什么要加读锁呢：防止数据在被读取的时候被别的线程加上写锁，

使用方式：在需要执行的语句后面加上`for update`就可以了

行锁

行锁又分**共享锁**和**排他锁**,由字面意思理解，就是给某一行加上锁，也就是一条记录加上锁。

注意：行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁。

共享锁：

名词解释：共享锁又叫做读锁，所有的事务只能对其进行读操作不能写操作，加上共享锁后在事务结束之前其他事务只能再加共享锁，除此之外其他任何类型的锁都不能再加了。

```
SELECT * from TABLE where id = "1"  lock in share mode;  结果集的数据都会加共享锁
```

排他锁：

名词解释：若某个事物对某一行加上了排他锁，只能这个事务对其进行读写，在此事务结束之前，其他事务不能对其进行加任何锁，其他进程可以读取,不能进行写操作，需等待其释放。

```
select status from TABLE where id=1 for update;
```

可以参考之前演示的共享锁,排它锁语句

由于对于表中,id字段为主键，就也相当于索引。执行加锁时，会将id这个索引为1的记录加上锁，那么这个锁就是行锁。

表锁

如何加表锁

innodb 的行锁是在有索引的情况下,没有索引的表是锁定全表的.

Innodb中的行锁与表锁

前面提到过，在Innodb引擎中既支持行锁也支持表锁，那么什么时候会锁住整张表，什么时候或只锁住一行呢?
只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁!

在实际应用中，要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。

行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁。行级锁的缺点是：由于需要请求大量的锁资源，所以速度慢，内存消耗大。

死锁

死锁（Deadlock）
所谓死锁：是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

解除正在死锁的状态有两种方法：

第一种：

1.查询是否锁表

```
show OPEN TABLES where In_use > 0;
```

2.查询进程（如果您有SUPER权限，您可以看到所有线程。否则，您只能看到您自己的线程）

```
show processlist
```

3.杀死进程id（就是上面命令的id列）

```
kill id
```

第二种：

1：查看当前的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX;
```

2：查看当前锁定的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
```

3：查看当前等锁的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS;
```

杀死进程

```
kill 线程ID
```

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。
产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。
（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。
（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

虽然不能完全避免死锁，但可以使死锁的数量减至最少。将死锁减至最少可以增加事务的吞吐量并减少系统开销，因为只有很少的事务回滚，而回滚会取消事务执行的所有工作。由于死锁时回滚而由应用程序重新提交。

下列方法有助于最大限度地降低死锁：

（1）按同一顺序访问对象。
（2）避免事务中的用户交互。
（3）保持事务简短并在一个批处理中。
（4）使用低隔离级别。
（5）使用绑定连接。

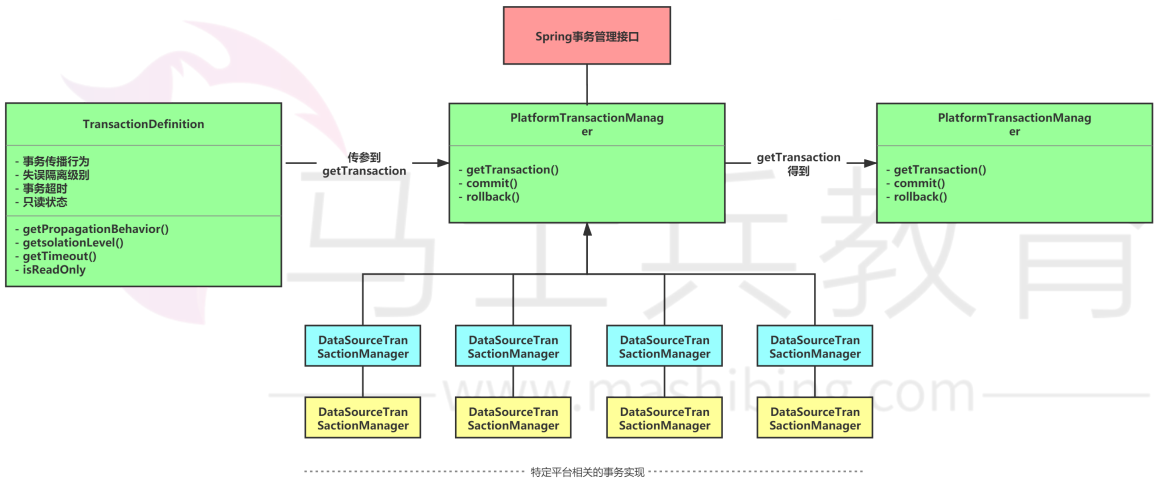
分布式事务的原理2阶段提交，同步异步阻塞非阻塞；

数据库事务隔离级别，MySQL默认的隔离级别

Mysql默认隔离级别：Repeatable Read

隔离级别	脏读	不可重复读	幻读
读未提交(Read uncommitted)	√	√	√
读已提交(Read committed)	×	√	√
可重复读(Repeatable read)	×	×	√
可串行化(Serializable)	×	×	×

Spring如何实现事务



Spring 事物四种实现方式：

- 基于编程式事务管理实现
- 基于TransactionProxyFactoryBean的声明式事务管理
- 基于AspectJ的XML声明式事务管理
- 基于注解的声明式事务管理

spring事务管理(详解和实例)

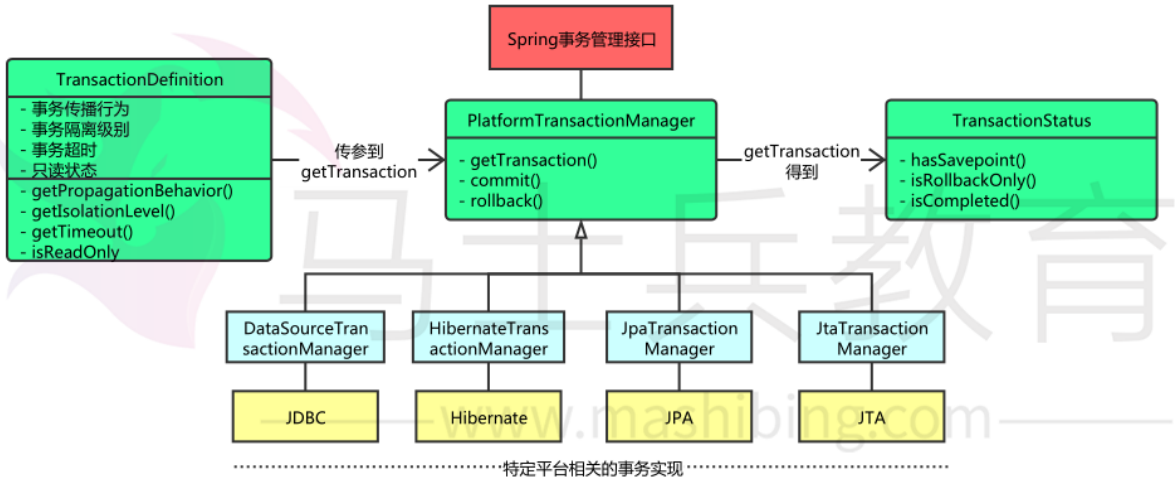
1 初步理解

理解事务之前，先讲一个你日常生活中最常干的事：取钱。
比如你去ATM机取1000块钱，大体有两个步骤：首先输入密码金额，银行卡扣掉1000元钱；然后ATM出1000元钱。这两个步骤必须是要么都执行要么都不执行。如果银行卡扣除了1000块但是ATM出钱失败的话，你将会损失1000元；如果银行卡扣钱失败但是ATM却出了1000块，那么银行将损失1000元。所以，如果一个步骤成功另一个步骤失败对双方都不是好事，如果不管哪一个步骤失败了以后，整个取钱过程都能回滚，也就是完全取消所有操作的话，这对双方都是极好的。
事务就是用来解决类似问题的。事务是一系列的动作，它们综合在一起才是一个完整的工作单元，这些动作必须全部完成，如果有一个失败的话，那么事务就会回滚到最开始的状态，仿佛什么都没发生过一样。
在企业级应用程序开发中，事务管理必不可少的技术，用来确保数据的完整性和一致性。
事务有四个特性：ACID

> - 原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。
> - 一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。
> - 隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
> - 持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

2 核心接口

Spring事务管理的实现有许多细节，如果对整个接口框架有个大体了解会非常有利于我们理解事务，下面通过讲解Spring的事务接口来了解Spring实现事务的具体策略。
Spring事务管理涉及的接口的联系如下：



2.1 事务管理器

Spring并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。
Spring事务管理器的接口是org.springframework.transaction.PlatformTransactionManager，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。此接口的内容如下：

```
Public interface PlatformTransactionManager()...{
    // 由TransactionDefinition得到TransactionStatus对象
    TransactionStatus getTransaction(TransactionDefinition definition) throws
TransactionException;
    // 提交
    Void commit(TransactionStatus status) throws TransactionException;
    // 回滚
    Void rollback(TransactionStatus status) throws TransactionException;
    }
```

从这里可知具体的具体的事务管理机制对Spring来说是透明的，它并不关心那些，那些是对应各个平台需要关心的，所以Spring事务管理的一个优点就是为不同的事务API提供一致的编程模型，如JTA、JDBC、Hibernate、JPA。下面分别介绍各个平台框架实现事务管理的机制。

2.1.1 JDBC事务

如果应用程序中直接使用JDBC来进行持久化，DataSourceTransactionManager会为你处理事务边界。为了使用DataSourceTransactionManager，你需要使用如下的XML将其装配到应用程序的上下文定义中：

```
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
```

实际上，DataSourceTransactionManager是通过调用java.sql.Connection来管理事务，而后者是通过DataSource获取到的。通过调用连接的commit()方法来提交事务，同样，事务失败则通过调用rollback()方法进行回滚。

2.1.2 Hibernate事务

如果应用程序的持久化是通过Hibernate实习的，那么你需要使用HibernateTransactionManager。对于Hibernate3，需要在Spring上下文定义中添加如下的`<bean>`声明：

```
    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
```

sessionFactory属性需要装配一个Hibernate的session工厂，HibernateTransactionManager的实现细节是它将事务管理的职责委托给org.hibernate.Transaction对象，而后者是从Hibernate Session中获取到的。当事务成功完成时，HibernateTransactionManager将会调用Transaction对象的commit()方法，反之，将会调用rollback()方法。

2.1.3 Java持久化API事务（JPA）

Hibernate多年来一直是事实上的Java持久化标准，但是现在Java持久化API作为真正的Java持久化标准进入大家的视野。如果你计划使用JPA的话，那你需要使用Spring的JpaTransactionManager来处理事务。你需要在Spring中这样配置JpaTransactionManager：

```
    <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
```

JpaTransactionManager只需要装配一个JPA实体管理工厂（javax.persistence.EntityManagerFactory接口的任意实现）。JpaTransactionManager将与由工厂所产生的JPA EntityManager合作来构建事务。

2.1.4 Java原生API事务

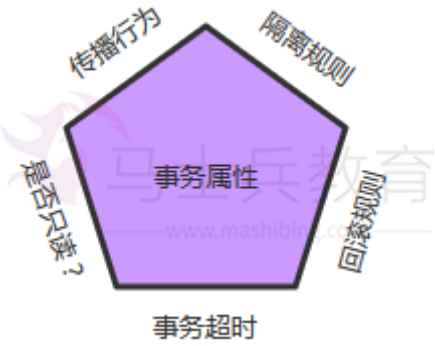
如果你没有使用以上所述的事务管理，或者是跨越了多个事务管理源（比如两个或者是多个不同的数据源），你就需要使用JtaTransactionManager：

```
    <bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
        <property name="transactionManagerName" value="java:/TransactionManager"
/>
    </bean>
```

JtaTransactionManager将事务管理的责任委托给javax.transaction.UserTransaction和javax.transaction.TransactionManager对象，其中事务成功完成通过UserTransaction.commit()方法提交，事务失败通过UserTransaction.rollback()方法回滚。

2.2 基本事务属性的定义

上面讲到的事务管理器接口PlatformTransactionManager通过getTransaction(TransactionDefinition definition)方法来得到事务，这个方法里面的参数是TransactionDefinition类，这个类就定义了一些基本的事务属性。
那么什么是事务属性呢？事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了5个方面，如图所示：



而TransactionDefinition接口内容如下：

```
public interface TransactionDefinition {
    int getPropagationBehavior(); // 返回事务的传播行为
    int getIsolationLevel(); // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据
    int getTimeout();  // 返回事务必须在多少秒内完成
    boolean isReadOnly(); // 事务是否只读，事务管理器能够根据这个返回值进行优化，确保事务是只读的
}
```

我们可以发现TransactionDefinition正好用来定义事务属性，下面详细介绍一下各个事务属性。

2.2.1 传播行为

事务的第一个方面是传播行为（propagation behavior）。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。Spring定义了七种传播行为：

传播行为	含义
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION_REQUIRED一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

（1）PROPAGATION_REQUIRED 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

```
//事务属性 PROPAGATION_REQUIRED
methodA{
    ……
    methodB();
    ……
}123456
//事务属性 PROPAGATION_REQUIRED
methodB{
   ……
}
```

使用spring声明式事务，spring使用AOP来支持声明式事务，会根据事务属性，自动在方法调用之前决定是否开启一个事务，并在方法执行之后决定事务提交或回滚事务。

单独调用methodB方法：

```
main{
    metodB();
}
```

相当于

```
Main{
    Connection con=null;
    try{
        con = getConnection();
        con.setAutoCommit(false);

        //方法调用
        methodB();

        //提交事务
        con.commit();
    } Catch(RuntimeException ex) {
        //回滚事务
        con.rollback();
    } finally {
        //释放资源
        closeCon();
    }
}
```

Spring保证在methodB方法中所有的调用都获得到一个相同的连接。在调用methodB时，没有一个存在的事务，所以获得一个新的连接，开启了一个新的事务。
单独调用MethodA时，在MethodA内又会调用MethodB.

执行效果相当于：

```
main{
    Connection con = null;
    try{
        con = getConnection();
        methodA();
        con.commit();
    } catch(RuntimeException ex) {
        con.rollback();
    } finally {
        closeCon();
    }
}
```

调用MethodA时，环境中没有事务，所以开启一个新的事务.当在MethodA中调用MethodB时，环境中已经有了一个事务，所以methodB就加入当前事务。

（2）PROPAGATION_SUPPORTS 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。但是对于事务同步的事务管理器，PROPAGATION_SUPPORTS与不使用事务有少许不同。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
  methodB();
}

//事务属性 PROPAGATION_SUPPORTS
methodB(){
  ……
}
```

单纯的调用methodB时，methodB方法是非事务的执行的。当调用methdA时,methodB则加入了methodA的事务中,事务地执行。

（3）PROPAGATION_MANDATORY 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    methodB();
}

//事务属性 PROPAGATION_MANDATORY
    methodB(){
    ……
}
```

当单独调用methodB时，因为当前没有一个活动的事务，则会抛出异常throw new IllegalTransactionStateException(“Transaction propagation ‘mandatory’ but no existing transaction found”);当调用methodA时，methodB则加入到methodA的事务中，事务地执行。

（4）PROPAGATION_REQUIRES_NEW 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    doSomeThingA();
    methodB();
    doSomeThingB();
}

//事务属性 PROPAGATION_REQUIRES_NEW
methodB(){
    ……
}
```

调用A方法：

```
main(){
    methodA();
}
```

相当于

```
main(){
    TransactionManager tm = null;
    try{
        //获得一个JTA事务管理器
        tm = getTransactionManager();
        tm.begin();//开启一个新的事务
        Transaction ts1 = tm.getTransaction();
        doSomeThing();
        tm.suspend();//挂起当前事务
        try{
            tm.begin();//重新开启第二个事务
            Transaction ts2 = tm.getTransaction();
```

```
            methodB();
            ts2.commit();//提交第二个事务
        } Catch(RunTimeException ex) {
            ts2.rollback();//回滚第二个事务
        } finally {
            //释放资源
        }
        //methodB执行完后，恢复第一个事务
        tm.resume(ts1);
        doSomeThingB();
        ts1.commit();//提交第一个事务
    } catch(RunTimeException ex) {
        ts1.rollback();//回滚第一个事务
    } finally {
        //释放资源
    }
}
```

在这里，我把ts1称为外层事务，ts2称为内层事务。从上面的代码可以看出，ts2与ts1是两个独立的事务，互不相干。Ts2是否成功并不依赖于 ts1。如果methodA方法在调用methodB方法后的doSomeThingB方法失败了，而methodB方法所做的结果依然被提交。而除了 methodB之外的其它代码导致的结果却被回滚了。使用PROPAGATION_REQUIRES_NEW,需要使用 JtaTransactionManager作为事务管理器。

（5）PROPAGATION_NOT_SUPPORTED 总是非事务地执行，并挂起任何存在的事务。使用PROPAGATION_NOT_SUPPORTED,也需要使用JtaTransactionManager作为事务管理器。（代码示例同上，可同理推出）

（6）PROPAGATION_NEVER 总是非事务地执行，如果存在一个活动事务，则抛出异常。

（7）PROPAGATION_NESTED如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按TransactionDefinition.PROPAGATION_REQUIRED 属性执行。这是一个嵌套事务,使用JDBC 3.0驱动时,仅仅支持DataSourceTransactionManager作为事务管理器。需要JDBC 驱动的java.sql.Savepoint类。有一些JTA的事务管理器实现可能也提供了同样的功能。使用PROPAGATION_NESTED，还需要把PlatformTransactionManager的nestedTransactionAllowed属性设为true;而 nestedTransactionAllowed属性值默认为false。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    doSomeThingA();
    methodB();
    doSomeThingB();
}

//事务属性 PROPAGATION_NESTED
methodB(){
    ……
}
```

如果单独调用methodB方法，则按REQUIRED属性执行。如果调用methodA方法，相当于下面的效果：

```
main(){
    Connection con = null;
    Savepoint savepoint = null;
    try{
        con = getConnection();
```

```
        con.setAutoCommit(false);
        doSomeThingA();
        savepoint = con2.setSavepoint();
        try{
            methodB();
        } catch(RuntimeException ex) {
            con.rollback(savepoint);
        } finally {
            //释放资源
        }
        doSomeThingB();
        con.commit();
    } catch(RuntimeException ex) {
        con.rollback();
    } finally {
        //释放资源
    }
}
```

当methodB方法调用之前，调用setSavepoint方法，保存当前的状态到savepoint。如果methodB方法调用失败，则恢复到之前保存的状态。但是需要注意的是，这时的事务并没有进行提交，如果后续的代码(doSomeThingB()方法)调用失败，则回滚包括methodB方法的所有操作。

嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

PROPAGATION_NESTED 与PROPAGATION_REQUIRES_NEW的区别:它们非常类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用 PROPAGATION_REQUIRES_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA事务管理器的支持。

使用PROPAGATION_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。DataSourceTransactionManager使用savepoint支持PROPAGATION_NESTED时，需要JDBC 3.0以上驱动及1.4以上的JDK版本支持。其它的JTA TrasactionManager实现可能有不同的支持方式。

PROPAGATION_REQUIRES_NEW 启动一个新的, 不依赖于环境的 “内部” 事务. 这个事务将被完全 commited 或 rolled back 而不依赖于外部事务, 它拥有自己的隔离范围, 自己的锁, 等等. 当内部事务开始执行时, 外部事务将被挂起, 内务事务结束时, 外部事务将继续执行。

另一方面, PROPAGATION_NESTED 开始一个 “嵌套的” 事务, 它是已经存在事务的一个真正的子事务. 潜套事务开始执行时, 它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 潜套事务是外部事务的一部分, 只有外部事务结束后它才会被提交。

由此可见, PROPAGATION_REQUIRES_NEW 和 PROPAGATION_NESTED 的最大区别在于, PROPAGATION_REQUIRES_NEW 完全是一个新的事务, 而 PROPAGATION_NESTED 则是外部事务的子事务, 如果外部事务 commit, 嵌套事务也会被 commit, 这个规则同样适用于 roll back.

PROPAGATION_REQUIRED应该是我们首先的事务传播行为。它能够满足我们大多数的事务需求。

2.2.2 隔离级别

事务的第二个维度就是隔离级别（isolation level）。隔离级别定义了一个事务可能受其他并发事务影响的程度。

（1）并发事务引起的问题

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。并发虽然是必须的，但可能会导致一下的问题。

> - 脏读（Dirty reads）——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。
> - 不可重复读（Nonrepeatable read）——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。
> - 幻读（Phantom read）——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

不可重复读与幻读的区别

不可重复读的重点是修改:
同样的条件, 你读取过的数据, 再次读取出来发现值不一样了
例如：在事务1中，Mary 读取了自己的工资为1000,操作并没有完成

```
    con1 = getConnection();
    select salary from employee empId ="Mary";
```

在事务2中，这时财务人员修改了Mary的工资为2000,并提交了事务.

```
    con2 = getConnection();
    update employee set salary = 2000;
    con2.commit();
```

在事务1中，Mary 再次读取自己的工资时，工资变为了2000

```
    //con1
    select salary from employee empId ="Mary";
```

在一个事务中前后两次读取的结果并不一致，导致了不可重复读。

幻读的重点在于新增或者删除:
同样的条件, 第1次和第2次读出来的记录数不一样
例如：目前工资为1000的员工有10人。事务1,读取所有工资为1000的员工。

```
    con1 = getConnection();
    Select * from employee where salary =1000;
```

共读取10条记录

这时另一个事务向employee表插入了一条员工记录，工资也为1000

```
    con2 = getConnection();
    Insert into employee(empId,salary) values("Lili",1000);
    con2.commit();
```

事务1再次读取所有工资为1000的员工

```
    //con1
    select * from employee where salary =1000;
```

共读取到了11条记录，这就产生了幻像读。

从总的结果来看, 似乎不可重复读和幻读都表现为两次读取的结果不一致。但如果你从控制的角度来看, 两者的区别就比较大。
对于前者, 只需要锁住满足条件的记录。
对于后者, 要锁住满足条件及其相近的记录。

（2）隔离级别

隔离级别	含义
ISOLATION_DEFAULT	使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED	最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
ISOLATION_READ_COMMITTED	允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
ISOLATION_REPEATABLE_READ	对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
ISOLATION_SERIALIZABLE	最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的

2.2.3 只读

事务的第三个特性是它是否为只读事务。如果事务只对后端的数据库进行该操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。

2.2.4 事务超时

为了使应用程序很好地运行，事务不能运行太长的时间。因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要的占用数据库资源。事务超时就是事务的一个定时器，在特定时间内事务如果没有执行完毕，那么就会自动回滚，而不是一直等待其结束。

2.2.5 回滚规则

事务五边形的最后一个方面是一组规则，这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）
但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

2.3 事务状态

上面讲到的调用PlatformTransactionManager接口的getTransaction()的方法得到的是TransactionStatus接口的一个实现，这个接口的内容如下：

```
public interface TransactionStatus{
    boolean isNewTransaction(); // 是否是新的事物
    boolean hasSavepoint(); // 是否有恢复点
    void setRollbackOnly();  // 设置为只回滚
    boolean isRollbackOnly(); // 是否为只回滚
    boolean isCompleted; // 是否已完成
}
```

可以发现这个接口描述的是一些处理事务提供简单的控制事务执行和查询事务状态的方法，在回滚或提交的时候需要应用对应的事务状态。

3 编程式事务

3.1 编程式和声明式事务的区别

Spring提供了对编程式事务和声明式事务的支持，编程式事务允许用户在代码中精确定义事务的边界，而声明式事务（基于AOP）有助于用户将操作与事务规则进行解耦。
简单地说，编程式事务侵入到了业务代码里面，但是提供了更加详细的事务管理；而声明式事务由于基于AOP，所以既能起到事务管理的作用，又可以不影响业务代码的具体实现。

3.2 如何实现编程式事务?

Spring提供两种方式的编程式事务管理，分别是：使用TransactionTemplate和直接使用PlatformTransactionManager。

3.2.1 使用TransactionTemplate

采用TransactionTemplate和采用其他Spring模板，如JdbcTempalte和HibernateTemplate是一样的方法。它使用回调方法，把应用程序从处理取得和释放资源中解脱出来。如同其他模板，TransactionTemplate是线程安全的。代码片段：

```
    TransactionTemplate tt = new TransactionTemplate(); // 新建一个
TransactionTemplate
    Object result = tt.execute(
        new TransactionCallback(){
            public Object doTransaction(TransactionStatus status){
                updateOperation();
                return resultOfUpdateOperation();
            }
    }); // 执行execute方法进行事务管理
```

使用TransactionCallback()可以返回一个值。如果使用TransactionCallbackWithoutResult则没有返回值。

3.2.2 使用PlatformTransactionManager

示例代码如下：

```
    DataSourceTransactionManager dataSourceTransactionManager = new
DataSourceTransactionManager(); //定义一个某个框架平台的TransactionManager，如JDBC、
Hibernate

dataSourceTransactionManager.setDataSource(this.getJdbcTemplate().getDataSource(
)); // 设置数据源
    DefaultTransactionDefinition transDef = new DefaultTransactionDefinition();
// 定义事务属性

transDef.setPropagationBehavior(DefaultTransactionDefinition.PROPAGATION_REQUIRE
D); // 设置传播行为属性
    TransactionStatus status =
dataSourceTransactionManager.getTransaction(transDef); // 获得事务状态
    try {
        // 数据库操作
        dataSourceTransactionManager.commit(status);// 提交
    } catch (Exception e) {
        dataSourceTransactionManager.rollback(status);// 回滚
    }
```

4 声明式事务

4.1 配置方式

根据代理机制的不同，总结了五种Spring事务的配置方式，配置文件如下：

（1）每个Bean都有一个代理

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <!-- 配置DAO -->
    <bean id="userDaoTarget" class="com.bluesky.spring.dao.UserDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
```

```
    <bean id="userDao"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

           <!-- 配置事务管理器 -->
           <property name="transactionManager" ref="transactionManager" />
        <property name="target" ref="userDaoTarget" />
         <property name="proxyInterfaces"
value="com.bluesky.spring.dao.GeneratorDao" />
        <!-- 配置事务属性 -->
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>
</beans>
```

(2) 所有Bean共享一个代理基类

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="transactionBase"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
            lazy-init="true" abstract="true">
        <!-- 配置事务管理器 -->
        <property name="transactionManager" ref="transactionManager" />
        <!-- 配置事务属性 -->
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
```

```
        </property>
    </bean>

    <!-- 配置DAO -->
    <bean id="userDaoTarget" class="com.bluesky.spring.dao.UserDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="userDao" parent="transactionBase" >
        <property name="target" ref="userDaoTarget" />
    </bean>
</beans>
```

(3) 使用拦截器

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="transactionInterceptor"

class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <property name="transactionManager" ref="transactionManager" />
        <!-- 配置事务属性 -->
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <bean
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames">
            <list>
                <value>*Dao</value>
```

```
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
            </list>
        </property>
    </bean>

    <!-- 配置DAO -->
    <bean id="userDao" class="com.bluesky.spring.dao.UserDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

(4) 使用tx标签配置的拦截器

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:annotation-config />
    <context:component-scan base-package="com.bluesky" />

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="*" propagation="REQUIRED" />
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <aop:pointcut id="interceptorPointCuts"
            expression="execution(* com.bluesky.spring.dao.*.*(..))" />
```

```
        <aop:advisor advice-ref="txAdvice"
            pointcut-ref="interceptorPointCuts" />
    </aop:config>
</beans>
```

(5) 全注解

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:annotation-config />
    <context:component-scan base-package="com.bluesky" />

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

</beans>
```

此时在DAO上需加上@Transactional注解，如下：

```
package com.bluesky.spring.dao;

import java.util.List;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.stereotype.Component;

import com.bluesky.spring.domain.User;

@Transactional
```

```
@Component("userDao")
public class UserDaoImpl extends HibernateDaoSupport implements UserDao {

    public List<User> listUsers() {
        return this.getSession().createQuery("from User").list();
    }
}
```

4.2 一个声明式事务的实例

首先是数据库表
book(isbn, book_name, price)
account(username, balance)
book_stock(isbn, stock)

然后是XML配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <import resource="applicationContext-db.xml" />

    <context:component-scan
        base-package="com.springinaction.transaction">
    </context:component-scan>

    <tx:annotation-driven transaction-manager="txManager"/>

    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

使用的类
BookShopDao

```
package com.springinaction.transaction;

public interface BookShopDao {
    // 根据书号获取书的单价
    public int findBookPriceByIsbn(String isbn);
    // 更新书的库存，使书号对应的库存-1
    public void updateBookStock(String isbn);
    // 更新用户的账户余额：account的balance-price
    public void updateUserAccount(String username, int price);
}
```

BookShopDaoImpl

```
package com.springinaction.transaction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository("bookShopDao")
public class BookShopDaoImpl implements BookShopDao {

    @Autowired
    private JdbcTemplate JdbcTemplate;

    @Override
    public int findBookPriceByIsbn(String isbn) {
        String sql = "SELECT price FROM book WHERE isbn = ?";

        return JdbcTemplate.queryForObject(sql, Integer.class, isbn);
    }

    @Override
    public void updateBookStock(String isbn) {
        //检查书的库存是否足够，若不够，则抛出异常
        String sql2 = "SELECT stock FROM book_stock WHERE isbn = ?";
        int stock = JdbcTemplate.queryForObject(sql2, Integer.class, isbn);
        if (stock == 0) {
            throw new BookStockException("库存不足！");
        }
        String sql = "UPDATE book_stock SET stock = stock - 1 WHERE isbn = ?";
        JdbcTemplate.update(sql, isbn);
    }

    @Override
    public void updateUserAccount(String username, int price) {
        //检查余额是否不足，若不足，则抛出异常
        String sql2 = "SELECT balance FROM account WHERE username = ?";
        int balance = JdbcTemplate.queryForObject(sql2, Integer.class,
username);
        if (balance < price) {
            throw new UserAccountException("余额不足！");
        }
        String sql = "UPDATE account SET balance = balance - ? WHERE username =
?";
        JdbcTemplate.update(sql, price, username);
```

```
    }

}
```

BookShopService

```
package com.springinaction.transaction;
public interface BookShopService {
     public void purchase(String username, String isbn);
}
```

BookShopServiceImpl

```
package com.springinaction.transaction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Service("bookShopService")
public class BookShopServiceImpl implements BookShopService {

    @Autowired
    private BookShopDao bookShopDao;

    /**
     * 1.添加事务注解
     * 使用propagation 指定事务的传播行为，即当前的事务方法被另外一个事务方法调用时如何使用事务。
     * 默认取值为REQUIRED，即使用调用方法的事务
     * REQUIRES_NEW：使用自己的事务，调用的事务方法的事务被挂起。
     *
     * 2.使用isolation 指定事务的隔离级别，最常用的取值为READ_COMMITTED
     * 3.默认情况下 Spring 的声明式事务对所有的运行时异常进行回滚，也可以通过对应的属性进行设置。通常情况下，默认值即可。
     * 4.使用readOnly 指定事务是否为只读。 表示这个事务只读取数据但不更新数据，这样可以帮助数据库引擎优化事务。若真的是一个只读取数据库值得方法，应设置readOnly=true
     * 5.使用timeOut 指定强制回滚之前事务可以占用的时间。
     */
    @Transactional(propagation=Propagation.REQUIRES_NEW,
            isolation=Isolation.READ_COMMITTED,
            noRollbackFor={UserAccountException.class},
            readOnly=true, timeout=3)
    @Override
    public void purchase(String username, String isbn) {
        //1.获取书的单价
        int price = bookShopDao.findBookPriceByIsbn(isbn);
        //2.更新书的库存
        bookShopDao.updateBookStock(isbn);
        //3.更新用户余额
        bookShopDao.updateUserAccount(username, price);
    }
}
```

Cashier

```
package com.springinaction.transaction;
import java.util.List;
public interface Cashier {
    public void checkout(String username, List<String>isbns);
}
```

CashierImpl：CashierImpl.checkout和bookShopService.purchase联合测试了事务的传播行为

```
package com.springinaction.transaction;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service("cashier")
public class CashierImpl implements Cashier {
    @Autowired
    private BookShopService bookShopService;

    @Transactional
    @Override
    public void checkout(String username, List<String> isbns) {
        for(String isbn : isbns) {
            bookShopService.purchase(username, isbn);
        }
    }
}
```

BookStockException

```
package com.springinaction.transaction;
public class BookStockException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public BookStockException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public BookStockException(String arg0, Throwable arg1, boolean arg2,
            boolean arg3) {
        super(arg0, arg1, arg2, arg3);
        // TODO Auto-generated constructor stub
    }

    public BookStockException(String arg0, Throwable arg1) {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    public BookStockException(String arg0) {
```

```
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public BookStockException(Throwable arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }
}
```

UserAccountException

```
package com.springinaction.transaction;
public class UserAccountException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public UserAccountException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(String arg0, Throwable arg1, boolean arg2,
            boolean arg3) {
        super(arg0, arg1, arg2, arg3);
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(String arg0, Throwable arg1) {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(String arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(Throwable arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }
}
```

测试类

```
package com.springinaction.transaction;

import java.util.Arrays;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringTransitionTest {
```

```
    private ApplicationContext ctx = null;
    private BookShopDao bookShopDao = null;
    private BookShopService bookShopService = null;
    private Cashier cashier = null;
    {
        ctx = new ClassPathXmlApplicationContext("config/transaction.xml");
        bookShopDao = ctx.getBean(BookShopDao.class);
        bookShopService = ctx.getBean(BookShopService.class);
        cashier = ctx.getBean(Cashier.class);
    }

    @Test
    public void testBookShopDaoFindPriceByIsbn() {
        System.out.println(bookShopDao.findBookPriceByIsbn("1001"));
    }

    @Test
    public void testBookShopDaoUpdateBookStock(){
        bookShopDao.updateBookStock("1001");
    }

    @Test
    public void testBookShopDaoUpdateUserAccount(){
        bookShopDao.updateUserAccount("AA", 100);
    }
    @Test
    public void testBookShopService(){
        bookShopService.purchase("AA", "1001");
    }

    @Test
    public void testTransactionPropagation(){
        cashier.checkout("AA", Arrays.asList("1001", "1002"));
    }
}
```

Spring事务管理及几种简单的实现

事务是逻辑上的一组操作，这组操作要么全部成功，要么全部失败，最为典型的就是银行转账的案例：

A要向B转账，现在A，B各自账户中有1000元，A要给B转200元，那么这个转账就必须保证是一个事务，防止中途因为各种原因导致A账户资金减少而B账户资金未添加，或者B账户资金添加而A账户资金未减少，这样不是用户有损失就是银行有损失，为了保证转账前后的一致性就必须保证转账操作是一个事务。

事务具有的ACID特性

首先，这篇文章先提及一些Spring中事务有关的API，然后分别实现编程式事务管理和声明式事务管理，其中声明式事务管理分别使用基于`TransactionProxyFactoryBean`的方式、基于AspectJ的XML方式、基于注解方式进行实现。

首先，我们简单看一下Spring事务管理需要提及的接口，Spring事务管理高层抽象主要包括3个接口

`PlatformTransactionManager` :事务管理器(用来管理事务，包含事务的提交，回滚)
`TransactionDefinition` :事务定义信息(隔离，传播，超时，只读)
`TransactionStatus` :事务具体运行状态

Spring根据事务定义信息(TransactionDefinition)由平台事务管理器(PlatformTransactionManager)真正进行事务的管理，在进行事务管理的过程中，事务会产生运行状态，状态保存在TransactionStatus中

PlatformTransactionManager:

Spring为不同的持久化框架提供了不同的PlatformTransactionManager如:
在使用Spring JDBC或iBatis进行持久化数据时，采用DataSourceTransactionManager
在使用Hibernate进行持久化数据时使用HibernateTransactionManager

TransactionDefinition:

`TransactionDefinition`接口中定义了一组常量，包括事务的隔离级别，事务的传播行为，超时信息，其中还定义了一些方法，可获得事务的隔离级别，超时信息，是否只读。

传播行为**主要解决**业务层方法之间的相互调用产生的事务应该如何传递的问题。

`TransactionDefinition`中定义的属性常量如下：

Field(属性)	**Description(描述)**
ISOLATION_DEFAULT	使用底层数据存储的默认隔离级别
ISOLATION_READ_COMMITTED	表示防止脏读;可能会发生不可重复的读取和幻像读取
ISOLATION_READ_UNCOMMITTED	表示可能会发生脏读，不可重复的读取和幻像读取
ISOLATION_REPEATABLE_READ	表示禁止脏读和不可重复读;可以发生幻影读取
ISOLATION_SERIALIZABLE	表示可以防止脏读，不可重复的读取和幻像读取
PROPAGATION_MANDATORY	支持当前交易;如果不存在当前事务，则抛出异常
PROPAGATION_NESTED	如果当前事务存在，则在嵌套事务中执行，其行为类似于PROPAGATION_REQUIRED
PROPAGATION_NEVER	不支持当前交易;如果当前事务存在，则抛出异常
PROPAGATION_NOT_SUPPORTED	不支持当前交易;而是总是非事务地执行
PROPAGATION_REQUIRED	支持当前交易;如果不存在，创建一个新的
PROPAGATION_REQUIRES_NEW	创建一个新的事务，挂起当前事务（如果存在）
PROPAGATION_SUPPORTS	支持当前交易;如果不存在，则执行非事务性的
TIMEOUT_DEFAULT	使用底层事务系统的默认超时，如果不支持超时，则为none

TransationStatus:

在该接口中提供了一些方法:

Method	Description
flush()	将基础会话刷新到数据存储（如果适用）：例如，所有受影响的Hibernate / JPA会话
hasSavepoint()	返回此事务是否内部携带保存点，也就是基于保存点创建为嵌套事务
isCompleted()	返回此事务是否完成，即是否已经提交或回滚
isNewTransaction()	返回当前交易是否是新的（否则首先参与现有交易，或者潜在地不会在实际交易中运行）
isRollbackOnly()	返回事务是否已被标记为仅回滚（由应用程序或由事务基础结构）
setRollbackOnly()	设置事务回滚

了解了上述接口，接下来我们通过转账案例来实现Spring的事务管理：

数据库中account表如下：

	id	name	money
1	1	aaa	1000
2	2	bbb	1000
3	3	ccc	1000
....	....	....	....

1.编程式事务管理实现：

AccountDao.java:

```
package com.spring.demo1;

/**
 * Created by zhuxinquan on 17-4-27.
 */
public interface AccountDao {
    public void outMoney(String out, Double money);
    public void inMoney(String in, Double money);
}
```

AccountDaoImp.java

```
package com.spring.demo1;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

/**
 * Created by zhuxinquan on 17-4-27.
 */
public class AccountDaoImp extends JdbcDaoSupport implements AccountDao {
```

```
    public void outMoney(String out, Double money) {
        String sql = "update account set money = money - ? where name = ?";
        this.getJdbcTemplate().update(sql, money, out);
    }

    public void inMoney(String in, Double money) {
        String sql = "update account set money = money + ? where name = ?";
        this.getJdbcTemplate().update(sql, money, in);
    }
}
```

AccountService.java

```
package com.spring.demo1;

/**
 * Created by zhuxinquan on 17-4-27.
 */
public interface AccountService {
    public void transfer(String out, String in, Double money);
}
```

AccountServiceImp.java

```
package com.spring.demo1;

import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

/**
 * Created by zhuxinquan on 17-4-27.
 */
public class AccountServiceImp implements AccountService{

    private AccountDao accountDao;

    //    注入事务管理的模板
    private TransactionTemplate transactionTemplate;

    public void setTransactionTemplate(TransactionTemplate transactionTemplate)
{
        this.transactionTemplate = transactionTemplate;
    }

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public void transfer(final String out, final String in, final Double money)
{
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus
transactionStatus) {
                accountDao.outMoney(out, money);
```

```
                //此处除0模拟转账发生异常
                int i = 1 / 0;
                accountDao.inMoney(in, money);
            }
        });
    }
}package com.spring.demo1;

import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

/**
 * Created by zhuxinquan on 17-4-27.
 */
public class AccountServiceImp implements AccountService{

    private AccountDao accountDao;

    //    注入事务管理的模板
    private TransactionTemplate transactionTemplate;

    public void setTransactionTemplate(TransactionTemplate transactionTemplate)
{
        this.transactionTemplate = transactionTemplate;
    }

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public void transfer(final String out, final String in, final Double money)
{
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus
transactionStatus) {
                accountDao.outMoney(out, money);
                int i = 1 / 0;
                accountDao.inMoney(in, money);
            }
        });
    }
}
```

创建Spring配置文件applicationContext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:property-placeholder location="classpath:jdbc.properties"/>
```

```
    <!--配置c3p0连接池-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${jdbc.Driver}"/>
        <property name="jdbcUrl" value="${jdbc.URL}"/>
        <property name="user" value="${jdbc.USERNAME}"/>
        <property name="password" value="${jdbc.PASSWD}"/>
    </bean>

    <!--配置业务层类-->
    <bean id="accountService" class="com.spring.demo1.AccountServiceImp">
        <property name="accountDao" ref="accountDao"/>
        <property name="transactionTemplate" ref="transactionTemplate"/>
    </bean>

    <!--配置Dao的类-->
    <bean id="accountDao" class="com.spring.demo1.AccountDaoImp">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置事务管理器-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置事务管理模板,Spring为了简化事务管理的代码而提供的类-->
    <bean id="transactionTemplate"
class="org.springframework.transaction.support.TransactionTemplate">
        <property name="transactionManager" ref="transactionManager"/>
    </bean>
</beans>
```

编写测试类如下:
SpringDemoTest1.java

```
import com.spring.demo1.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.context.annotation.Configuration;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.annotation.Resource;

/**
 * Created by zhuxinquan on 17-4-27.
 * Spring编程式事务管理
 */

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class SpringDemoTest1 {

    @Resource(name = "accountService")
    private AccountService accountService;
```

```
    @Test
    public void demo1(){
        accountService.transfer("aaa", "bbb", 200d);
    }
}
```

2.基于TransactionProxyFactoryBean的声明式事务管理

Dao与Service代码与1中相同，applicationContext2.xml如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--配置c3p0连接池-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${jdbc.Driver}"/>
        <property name="jdbcUrl" value="${jdbc.URL}"/>
        <property name="user" value="${jdbc.USERNAME}"/>
        <property name="password" value="${jdbc.PASSWD}"/>
    </bean>

    <!--配置业务层类-->
    <bean id="accountService" class="com.spring.demo2.AccountServiceImp">
        <property name="accountDao" ref="accountDao"/>
    </bean>

    <!--配置Dao的类-->
    <bean id="accountDao" class="com.spring.demo2.AccountDaoImp">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置事务管理器-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置业务层代理-->
    <bean id="accountServiceProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <!--配置目标对象-->
        <property name="target" ref="accountService"/>
        <!--注入事务管理器-->
        <property name="transactionManager" ref="transactionManager"/>
        <!--注入事务的属性-->
        <property name="transactionAttributes">
            <props>
                <!--
```

```
                prop格式
                    * PROPAGATION   :事务的传播行为
                    * ISOLATION     :事务的隔离级别
                    * readOnly      :只读(不可以进行修改，插入，删除的操作)
                    * -Exception    :发生哪些异常回滚事务
                    * +Exception    :发生哪些异常不回滚事务
                -->
                <prop key="transfer">PROPAGATION_REQUIRED</prop>
                <!--<prop key="transfer">PROPAGATION_REQUIRED,readOnly</prop>-->
                <!--<prop key="transfer">PROPAGATION_REQUIRED,
+java.lang.ArithmeticException</prop>-->
            </props>
        </property>
    </bean>
</beans>
```

此时注入时需要选择代理类，因为在代理类中进行增强操作，测试代码如下：
SpringDemoTest2.java

```
import com.spring.demo2.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.annotation.Resource;

/**
 * Created by zhuxinquan on 17-4-27.
 * Spring声明式事务管理:基于TransactionProxyFactoryBean的方式
 */

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext2.xml")
public class SpringDemoTest2 {


    /*
    此时需要注入代理类：因为代理类进行增强操作
     */
//    @Resource(name = "accountService")
    @Resource(name = "accountServiceProxy")
    private AccountService accountService;

    @Test
    public void demo1(){
        accountService.transfer("aaa", "bbb", 200d);
    }
}
```

3.基于AspectJ的XML声明式事务管理

在这种方式下Dao和Service的代码也没有改变，applicationContext3.xml如下：
applicationContext3.xml

```
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:task="http://www.springframework.org/schema/task"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.1.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
       http://www.springframework.org/schema/task
       http://www.springframework.org/schema/task/spring-task-3.1.xsd">

    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--配置c3p0连接池-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${jdbc.Driver}"/>
        <property name="jdbcUrl" value="${jdbc.URL}"/>
        <property name="user" value="${jdbc.USERNAME}"/>
        <property name="password" value="${jdbc.PASSWD}"/>
    </bean>

    <!--配置业务层类-->
    <bean id="accountService" class="com.spring.demo3.AccountServiceImp">
        <property name="accountDao" ref="accountDao"/>
    </bean>

    <!--配置Dao的类-->
    <bean id="accountDao" class="com.spring.demo3.AccountDaoImp">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置事务管理器-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置事务的通知：(事务的增强)-->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!--
                propagation     :事务传播行为
                isolation       :事务的隔离级别
                read-only       :只读
                rollback-for    :发生哪些异常回滚
                no-rollback-for :发生哪些异常不回滚
                timeout         :过期信息
            -->
            <tx:method name="transfer" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
```

```
    <!--配置切面-->
    <aop:config>
        <!--配置切入点-->
        <aop:pointcut id="pointcut1" expression="execution(*
com.spring.demo3.AccountService+.*(..))"/>
        <!--配置切面-->
        <aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut1"/>
    </aop:config>
</beans>
```

测试类与1中相同，增强是动态织入的，所以此时注入的还是accountService。

4.基于注解的声明式事务管理

基于注解的方式需要在业务层上添加一个@Transactional的注解。
如下：
AccountServiceImp.java

```
package com.spring.demo4;

import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

/**
 * Created by zhuxinquan on 17-4-27.
 * propagation  :事务的传播行为
 * isolation    :事务的隔离级别
 * readOnly     :只读
 * rollbackFor  :发生哪些异常回滚
 */
@Transactional(propagation = Propagation.REQUIRED)
public class AccountServiceImp implements AccountService {

    private AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public void transfer(String out, String in, Double money) {
        accountDao.outMoney(out, money);
        int i = 1 / 0;
        accountDao.inMoney(in, money);
    }
}
```

此时需要在Spring配置文件中开启注解事务，打开事务驱动
applicationContext4.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
```

```
        xmlns:tx="http://www.springframework.org/schema/tx"
        xmlns:task="http://www.springframework.org/schema/task"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-3.1.xsd">

    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--配置c3p0连接池-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${jdbc.Driver}"/>
        <property name="jdbcUrl" value="${jdbc.URL}"/>
        <property name="user" value="${jdbc.USERNAME}"/>
        <property name="password" value="${jdbc.PASSWD}"/>
    </bean>

    <!--配置业务层类-->
    <bean id="accountService" class="com.spring.demo4.AccountServiceImp">
        <property name="accountDao" ref="accountDao"/>
    </bean>

    <!--配置Dao的类-->
    <bean id="accountDao" class="com.spring.demo4.AccountDaoImp">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置事务管理器-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--开启注解事务  打开事务驱动-->
    <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

测试类与1中相同

JDBC如何实现事务

在JDBC中处理事务，都是通过Connection完成的。

同一事务中所有的操作，都在使用同一个Connection对象。

①JDBC中的事务

Connection的三个方法与事务有关：

- setAutoCommit（boolean）:设置是否为自动提交事务，如果true（默认值为true）表示自动提交，也就是每条执行的SQL语句都是一个单独的事务，如果设置为false，那么相当于开启了事务了； con.setAutoCommit(false) 表示开启事务。
- commit（）：提交结束事务。
- rollback（）：回滚结束事务。

```
示例代码
try{
    con.setAutoCommit(false);//开启事务
    ......
    con.commit();//try的最后提交事务
} catch（） {
    con.rollback();//回滚事务
}
```

嵌套事务实现

spring 事务嵌套:外层事务TraB,内层事务TraA、TraC

```
场景1：
TraA、TraC @Transactional（默认REQUIRED）
TraB:
    traA.update(order1); (traA.update throw new RuntimeException();)
traC.update(order2);
结果:内外层事务全部回滚;
```

```
场景2：
TraA、TraC @Transactional（默认REQUIRED）
TraB:
    traA.update(order1); (traA.update throw new RuntimeException();try catch
traC.update)
traC.update(order2);
结果:内外层事务全部不回滚，traA中try catch后的事务提交;
```

```
场景3：
TraA、TraC @Transactional（默认REQUIRED）
TraB: try{(traA.update throw new RuntimeException();
        在外层TraB try catch TraA)
            traA.update(order1);
            }catch(){}
        traC.update(order2);
结果:内外层事务全部回滚,内层的异常抛出到外层捕获也会回滚;
```

```
场景4：
TraA @Transactional(propagation=Propagation.REQUIRES_NEW)、TraC
@Transactional（默认REQUIRED）
TraB:
    traA.update(order1); (traA.update throw new RuntimeException();)
    traC.update(order2);
结果:内层事务回滚,外层事务继续提交;
```

```
场景5：
TraA @Transactional(propagation=Propagation.REQUIRES_NEW)、TraC
@Transactional（默认REQUIRED）
    TraB:
        traA.update(order1); (traA.update throw new RuntimeException();try catch
traC.update)
        traC.update(order2);
结果:内外层事务全部不回滚，traA中try catch后的事务提交,达到与场景2的同样效果;
```

```
 场景6：
 TraA @Transactional(propagation=Propagation.REQUIRES_NEW)、TraC
@Transactional（默认REQUIRED）
     TraB:
        try{ (traA.update throw new RuntimeException();在 外层TraB try catch TraA)
            traA.update(order1); }
        catch traC.update(order2);
 结果:内层事务回滚,外层事务不回滚;
```

Spring事务管理--(二)嵌套事物详解

一、前言

最近开发程序的时候，出现数据库自增id跳数字情况，无奈之下dba遍查操作日志，没有delete记录。才开始慢慢来查询事物问题。多久以来欠下的账，今天该还给spring事物。

二、spring嵌套事物

1、展示项目代码--简单测springboot项目

整体项目就这么简单，为了方便。这里就只有biz层与service层，主要作为两层嵌套，大家只要看看大概就ok。后面会给出git项目地址，下载下来看一看就明白，力求最简单。

下面我们分情况介绍异常。

Controller 调用层（没有使用它作为外层，因为controller作为外层要在servlet-mvc.xml 配置就ok。但是我觉得比较麻烦，一般也不推荐）

```
<pre name="code" class="html">package com.ycy.app;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * Created by ycy on 16/7/19.
 */
@RestController
@SpringBootApplication
```

```
@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class})
@ImportResource({"classpath:/applicationContext.xml"})
public class Application {
  @Autowired
  private TestBiz testBiz;

  @RequestMapping("/")
  String home() throws Exception {
    System.out.println("controller 正常执行");
    testBiz.insetTes();

    return " 正常返回Hello World!";
  }

  public static void main(String[] args) throws Exception {
    SpringApplication.run(Application.class, args);
  }
}
```

Biz层（外层）

```
<pre name="code" class="html">package com.ycy.app;

import com.ycy.service.TestService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

/**
 * Created by ycy on 16/7/20.
 */
@Component
public class TestBiz {
  @Autowired
  private TestService testService;

  @Transactional
  public void insetTes() {
    for (int j = 0; j < 8; j++) {
      testService.testInsert(j, j + "姓名");
    }
    System.out.println("biz层 正常执行");
  }
}
```

Service层 （内层）

```
<pre name="code" class="html"><pre name="code" class="html"><pre name="code"
class="html">package com.ycy.service.impl;

import com.ycy.center.dao.entity.YcyTable;
import com.ycy.center.dao.mapper.YcyTableMapper;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * Created by ycy on 16/7/19.
 */
@Service
public class TestServiceImpl implements com.ycy.service.TestService {
    @Autowired
   private YcyTableMapper ycyTableMapper;
    @Transactional
    public void  testInsert(int num,String name) {

            YcyTable ycyTable=new YcyTable();
            ycyTable.setName(name);
            ycyTable.setNum(num);
            ycyTableMapper.insert(ycyTable);
        System.out.println(num+"service正常执行");

    }
}
```

2、外部起事物，内部起事物，内外都无Try Catch

外部异常：

代码展示，修改外层Biz层代码如下

```
<pre name="code" class="html"><pre name="code" class="html">@Component
public class TestBiz {
  @Autowired
  private TestService testService;

  @Transactional
  public void insetTes() {
    for (int j = 0; j < 8; j++) {
      testService.testInsert(j, j + "姓名");
      if (j == 3) {
        int i = 1 / 0;// 此处会产生异常
      }
    }
    System.out.println("biz层 正常执行");
  }
}
```

打印执行结果：0-3service正常执行　　数据库结果：全部数据回滚

外部异常总结: 内外都无try Catch的时候，外部异常，全部回滚。

内部异常：

代码展示，修改service层代码

```
package com.ycy.service.impl;

import com.ycy.center.dao.entity.YcyTable;
import com.ycy.center.dao.mapper.YcyTableMapper;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * Created by ycy on 16/7/19.
 */
@Service
public class TestServiceImpl implements com.ycy.service.TestService {
  @Autowired
  private YcyTableMapper ycyTableMapper;

  @Transactional
  public void testInsert(int num, String name) {

    YcyTable ycyTable = new YcyTable();
    ycyTable.setName(name);
    ycyTable.setNum(num);
    ycyTableMapper.insert(ycyTable);
    if (num == 3) {
      int i = 1 / 0;// 此处会产生异常
    }
    System.out.println(num + "service正常执行");

  }
}
```

打印执行结果：0-3service正常执行　　数据库结果：全部数据回滚

内部异常总结： 内外都无try Catch的时候，内部异常，全部回滚。

3、外部起事物，内部起事物，外部有Try Catch

外部异常：

代码展示，修改biz层代码

```
@Component
public class TestBiz {
  @Autowired
  private TestService testService;
  @Transactional
  public void insetTes() {
    try {
```

```
      for (int j = 0; j < 8; j++) {
        testService.testInsert(j, j + "姓名");

        if (j == 3) {
          int i = 1 / 0;// 此处会产生异常
        }
      }
    } catch (Exception ex) {
      System.out.println("异常日志处理");
    }
    System.out.println("biz层 正常执行");
  }
}
```

打印结果：0-3执行正常数据库结果：4条数据

外部异常总结：外部有try Catch时候，外部异常，不能回滚（事物错误）

内部异常：

代码展示，修改service层代码：

```
@Service
public class TestServiceImpl implements com.ycy.service.TestService {
  @Autowired
  private YcyTableMapper ycyTableMapper;

  @Transactional
  public void testInsert(int num, String name) {

    YcyTable ycyTable = new YcyTable();
    ycyTable.setName(name);
    ycyTable.setNum(num);
    ycyTableMapper.insert(ycyTable);
    if (num == 3) {
      int i = 1 / 0;// 此处会产生异常
    }
    System.out.println(num + "service正常执行");
  }
}
```

打印结果：0-2打印正常　数据库结果：无数据，全部数据回滚

内部异常总结：外部有try Catch时候，内部异常，全部回滚

4、外部起事物，内部起事物，内部有Try Catch

外部异常:

代码展示，修改biz层：

```
@Component
public class TestBiz {
  @Autowired
  private TestService testService;
```

```
  @Transactional
  public void insetTes() {
    for (int j = 0; j < 8; j++) {
      testService.testInsert(j, j + "姓名");
      if (j == 3) {
        int i = 1 / 0;// 此处会产生异常
      }
    }
    System.out.println("biz层 正常执行");
  }
}
```

打印结果：0-3service打印正常 数据库结果：无数据，全部数据回滚

外部异常总结： 内部有try Catch，外部异常，全部回滚

内部异常：

修改service层代码：

```
@Service
public class TestServiceImpl implements com.ycy.service.TestService {
  @Autowired
  private YcyTableMapper ycyTableMapper;

  @Transactional
  public void testInsert(int num, String name) {
    try {
      YcyTable ycyTable = new YcyTable();
      ycyTable.setName(name);
      ycyTable.setNum(num);
      ycyTableMapper.insert(ycyTable);
      if (num == 3) {
        int i = 1 / 0;// 此处会产生异常
      }
    } catch (Exception ex) {
      System.out.println(num + "service异常日志");
    }
    System.out.println(num + "service正常执行");

  }
}
```

打印结果：0-0service打印正常 数据库结果：没有回滚

内部异常总结： 内部有try Catch，内部异常，全部不回滚(事物失败);

5、外部起事物，内部起事物，内外有Try Catch

外部异常：

代码展示，修改biz层：

```
@Component
```

```
public class TestBiz {
  @Autowired
  private TestService testService;

  @Transactional
  public void insetTes() {
    try {
      for (int j = 0; j < 8; j++) {
        testService.testInsert(j, j + "姓名");
        if (j == 3) {
          int i = 1 / 0;// 此处会产生异常
        }
      }
    } catch (Exception ex) {
      System.out.println("biz层异常日志处理");
    }
    System.out.println("biz层 正常执行");
  }
}
```

打印结果：0-3service打印正常 数据库结果：插入三条数据，没有回滚

外部异常总结： 内外都有try Catch，外部异常，事物执行一半（事物失败）

内部异常：

代码展示,修改service 层代码

```
@Service
public class TestServiceImpl implements com.ycy.service.TestService {
  @Autowired
  private YcyTableMapper ycyTableMapper;

  @Transactional
  public void testInsert(int num, String name) {
    try {
      YcyTable ycyTable = new YcyTable();
      ycyTable.setName(name);
      ycyTable.setNum(num);
      ycyTableMapper.insert(ycyTable);
      if (num == 3) {
        int i = 1 / 0;// 此处会产生异常
      }
    } catch (Exception ex) {
      System.out.println(num + "service异常日志处理");
    }
    System.out.println(num + "service正常执行");
  }
}
```

打印结果：0-7service打印正常，3异常日子好 数据库结果：插入全部，没有回滚

内部事物总结： 内外都有try Catch，内部异常，事物全部不会滚（事物失败）

三、嵌套事物总结

事物成功总结

1、内外都无try Catch的时候，外部异常，全部回滚。

2、内外都无try Catch的时候，内部异常，全部回滚。

3、外部有try Catch时候，内部异常，全部回滚

4、内部有try Catch，外部异常，全部回滚

5、友情提示：外层方法中调取其他接口，或者另外开启线程的操作，一定放到最后！！！(因为调取接口不能回滚，一定要最后来处理)

总结：由于上面的异常被捕获导致，很多事务回滚失败。如果一定要将捕获，请捕获后又抛出RuntimeException（默认为异常捕获RuntimeException）。

四、正确的嵌套事物实例

controller层

```
package com.ycy.app;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * Created by ycy on 16/7/19.
 */
@RestController
@SpringBootApplication
@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class})
@ImportResource({"classpath:/applicationContext.xml"})
public class Application {
  @Autowired
  private TestBiz testBiz;

  @RequestMapping("/")
  String home()  {
    System.out.println("controller 正常执行");
    try {
      testBiz.insetTes();
    } catch (Exception e) {
      System.out.println("controller 异常日志执行");
    }

    return " 正常返回Hello World!";
  }
```

```
  public static void main(String[] args) throws Exception {
    SpringApplication.run(Application.class, args);
  }
}
```

外层biz层:

```
package com.ycy.app;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

import com.ycy.service.TestService;

/**
 * Created by ycy on 16/7/20.
 */
@Component
public class TestBiz {
  @Autowired
  private TestService testService;

  @Transactional
  public void insetTes() throws Exception {
    try {
      for (int j = 0; j < 8; j++) {
        testService.testInsert(j, j + "姓名");
        if (j == 3) {
          int i = 1 / 0;// 此处会产生异常
        }
      }
    } catch (Exception ex) {
      System.out.println("biz层异常日志处理");
      throw new RuntimeException(ex);
    }

    System.out.println("biz层 正常执行");
  }
}
```

内层service层

```
package com.ycy.service.impl;

import com.ycy.center.dao.entity.YcyTable;
import com.ycy.center.dao.mapper.YcyTableMapper;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```java
/**
 * Created by ycy on 16/7/19.
 */
@Service
public class TestServiceImpl implements com.ycy.service.TestService {
  @Autowired
  private YcyTableMapper ycyTableMapper;
  @Transactional
  public void testInsert(int num, String name) throws Exception {
    try {
      YcyTable ycyTable = new YcyTable();
      ycyTable.setName(name);
      ycyTable.setNum(num);
      ycyTableMapper.insert(ycyTable);
      if (num== 3) {
        int i = 1 / 0;// 此处会产生异常
      }
    } catch (Exception ex) {
      System.out.println(num + "service异常日志处理");
        throw new RuntimeException(ex);
    }
    System.out.println(num + "service正常执行");
  }
}
```

分布式事务实现；

1）基于XA协议的两阶段提交（2PC） XA 规范主要 定义了 (全局) 事务管理器 (Transaction Manager) 和 (局部) 资源管理器 (Resource Manager) 之间的接口。

2）两阶段提交 事务的提交分为两个阶段： 预提交阶段(Pre-Commit Phase) 决策后阶段（Post-Decision Phase）

3）补偿事务（TCC） 针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段
Try 阶段主要是对业务系统做检测及资源预留 Confirm 阶段主要是对业务系统做确认提交，Try 阶段执行成功并开始执行 Confirm 阶段时， 默认Confirm 阶段是不会出错的。即：只要 Try 成功，Confirm 一定成功。 Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放

4）本地消息表（MQ 异步确保） 其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。

5）MQ 事务消息 有一些第三方的 MQ 是支持事务消息的，比如 RocketMQ，他们支持事务消息的方式也是类似 于采用的二阶段提交，但是市面上一些主流的 MQ 都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

6）Sagas 事务模型 该模型其核心思想就是拆分分布式系统中的长事务为多个短事务，或者叫多个本地事务，然后由 Sagas 工作流引擎负责协调，如果整个流程正常结束，那么就算是业务成功完成，如果在这过程 中实现失败，那么Sagas工作流引擎就会以相反的顺序调用补偿操作，重新进行业务回滚。

7）其他补偿方式 加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任 务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。 在某些特殊的情况下，还会有 "人工补偿" 的，这也是最后一道屏障。

分布式事务与解决方案

前言

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

产生原因

数据库分库分表：

当数据库单表一年产生的数据超过1000W，那么就要考虑分库分表（具体分库分表的原理在此不做解释），简单的说就是原来的一个数据库变成了多个数据库。这时候，如果一个操作既访问01库，又访问02库，而且要保证数据的一致性，那么就要用到分布式事务。

应用SOA化：

所谓的SOA化，就是业务的服务化。比如原来单机支撑了整个电商网站，现在对整个网站进行拆解，分离出了订单中心、用户中心、库存中心。对于订单中心，有专门的数据库存储订单信息，用户中心也有专门的数据库存储用户信息，库存中心也会有专门的数据库存储库存信息。这时候如果要同时对订单和库存进行操作，那么就会涉及到订单数据库和库存数据库，为了保证数据一致性，就需要用到分布式事务。

分布式事务是企业集成中的一个技术难点，也是每一个分布式系统架构中都会涉及到的一个东西，特别是在微服务架构中，几乎可以说是无法避免。

应用场景

支付、转账：

最经典的场景就是支付了，一笔支付，是对买家账户进行扣款，同时对卖家账户进行加钱，这些操作必须在一个事务里执行，要么全部成功，要么全部失败。而对于买家账户属于买家中心，对应的是买家数据库，而卖家账户属于卖家中心，对应的是卖家数据库，对不同数据库的操作必然需要引入分布式事务。

在线下单：

买家在电商平台下单，往往会涉及到两个动作，一个是扣库存，第二个是更新订单状态，库存和订单一般属于不同的数据库，需要使用分布式事务保证数据一致性。

电商场景：流量充值业务

中国移动-流量充值能力中心，核心业务流程为：

1、用户进入流量充值商品购买页面，选择流量商品；

2、购买流量充值商品，有库存限制则判断库存，生成流量购买订单；

3、选择对应的支付方式（和包、银联、支付宝、微信）进行支付操作；

4、支付成功后，近实时流量到账即可使用流量商品；

此业务流程看似不是很复杂对吧，不涉及到类似电商业务的实物购买，但是我认为其中的区别并不是很大，只是缺少电商中的物流发货流程，其他流程几乎是一样的，也有库存以及优惠折扣等业务存在。

数据库事务

事务是由一组SQL语句组成的逻辑处理单元，事务具有以下4个属性，通常简称为事务的ACID属性：

原子性（Atomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。

一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以保持数据的完整性；事务结束时，所有的内部数据结构（如B树索引或双向链表）也都必须是正确的。

隔离性（Isoation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的"独立"环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。

持久性（Durabe）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

事务类型：

JDBC事务：即为上面说的数据库事务中的本地事务，通过connection对象控制管理。

JTA事务：JTA指Java事务API(JavaTransaction API)，是Java EE数据库事务规范，JTA只提供了事务管理接口，由应用程序服务器厂商（如WebSphere Application Server）提供实现，JTA事务比JDBC更强大，支持分布式事务。

隔离级别及引发现象：（略谈）

隔离级别	脏读	非重复读	幻读
未提交读	Y	Y	Y
提交读	N	Y	Y
可重复读	N	N	Y
序列化	N	N	N

Spring事务传播行为：（略谈）

PROPAGATION_REQUIRED：支持当前事务，如当前没有事务，则新建一个。

PROPAGATION_SUPPORTS：支持当前事务，如当前没有事务，则已非事务性执行（源码中提示有个注意点，看不太明白，留待后面考究）。

PROPAGATION_MANDATORY：支持当前事务，如当前没有事务，则抛出异常（强制一定要在一个已经存在的事务中执行，业务方法不可独自发起自己的事务）。

PROPAGATION_REQUIRES_NEW：始终新建一个事务，如当前原来有事务，则把原事务挂起。

PROPAGATION_NOT_SUPPORTED：不支持当前事务，始终已非事务性方式执行，如当前事务存在，挂起该事务。

PROPAGATION_NEVER：不支持当前事务；如果当前事务存在，则引发异常。

PROPAGATION_NESTED：如果当前事务存在，则在嵌套事务中执行，如果当前没有事务，则执行与PROPAGATION_REQUIRED 类似的操作（注意：当应用到JDBC时，只适用JDBC 3.0以上驱动）。

事务种类：

本地事务：普通事务，独立一个数据库，能保证在该数据库上操作的ACID。

分布式事务：涉及两个或多个数据库源的事务，即跨越多台同类或异类数据库的事务（由每台数据库的本地事务组成的），分布式事务旨在保证这些本地事务的所有操作的ACID，使事务可以跨越多台数据库。

如何保证强一致性

本地事务（mysql 之 InnoDB）：

InnoDB支持事务，同Oracle类似，事务提交需要写redo、undo。采用日志先行的策略，将数据的变更在内存中完成，并且将事务记录成redo，顺序的写入redo日志中，即表示该事务已经完成，就可以返回给客户已提交的信息。但是实际上被更改的数据还在内存中，并没有刷新到磁盘，即还没有落地，当达到一定的条件，会触发checkpoint，将内存中的数据（page）合并写入到磁盘，这样就减少了离散写、IOPS，提高性能。

在这个过程中，如果服务器宕机了，内存中的数据丢失，当重启后，会通过redo日志进行recovery重做。确保不会丢失数据。因此只要redo能够实时的写入到磁盘，InnoDB就不会丢数据。

分布式事务：

多个数据库中的某个数据库在提交事务的时候突然断电，那么它是怎么样恢复的呢？ 这也是分布式系统复杂的地方，因为分布式的网络环境很复杂，这种“断电”故障要比单机多很多，所以我们在做分布式系统的时候，最先考虑的就是这种情况。这些异常可能有 机器宕机、网络异常、消息丢失、消息乱序、数据错误、不可靠的TCP、存储数据丢失、其他异常等等...

对分布式系统有过研究的读者， 听说过 "CAP定律"、"Base理论" 等，这里不对这些概念做过多的解释，有兴趣的读者可以查看相关参考资料 。

在分布式系统中，同时满足 "CAP定律" 中的 "一致性"、"可用性" 和 "分区容错性" 三者是不可能的， 根据不同的业务场景使用不同的方法实现最终一致性，可以根据业务的特性做部分取舍，在业务过程中可以容忍一定时间内的数据不一致。

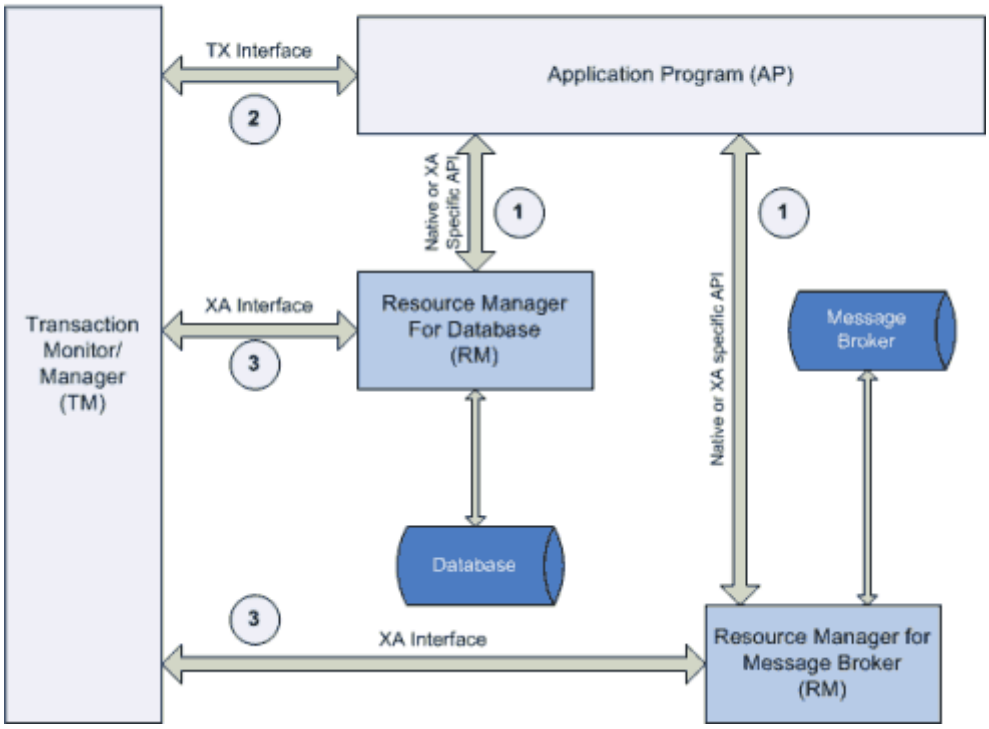
实现分布式事务解决方案

基于XA协议的两阶段提交（2PC）

XA 是由 X/Open 组织提出的**分布式事务的规范。**

XA 规范主要 定义了 (全局) 事务管理器 (Transaction Manager) 和 (局部) 资源管理器 (Resource Manager) 之间的接口。 XA 接口是双向的系统接口，在事务管理器（Transaction Manager）以及一个或多个资源管理器（Resource Manager）之间形成通信桥梁。 XA 之所以需要引入事务管理器是因为，在分布式系统中，从理论上讲（参考Fischer等的论文），两台机器理论上无 法达到一致的状态，需要引入一个单点进行协调。 事务管理器控制着全局事务，管理事务生命周期，并协调资源。资源管理器负责控制和管理实际资源（如数据库或 JMS队列）。下图说明了事务管理器、资源管理器，与应用程序之间的关系：

Conceptual View of XA/Open DTP Model



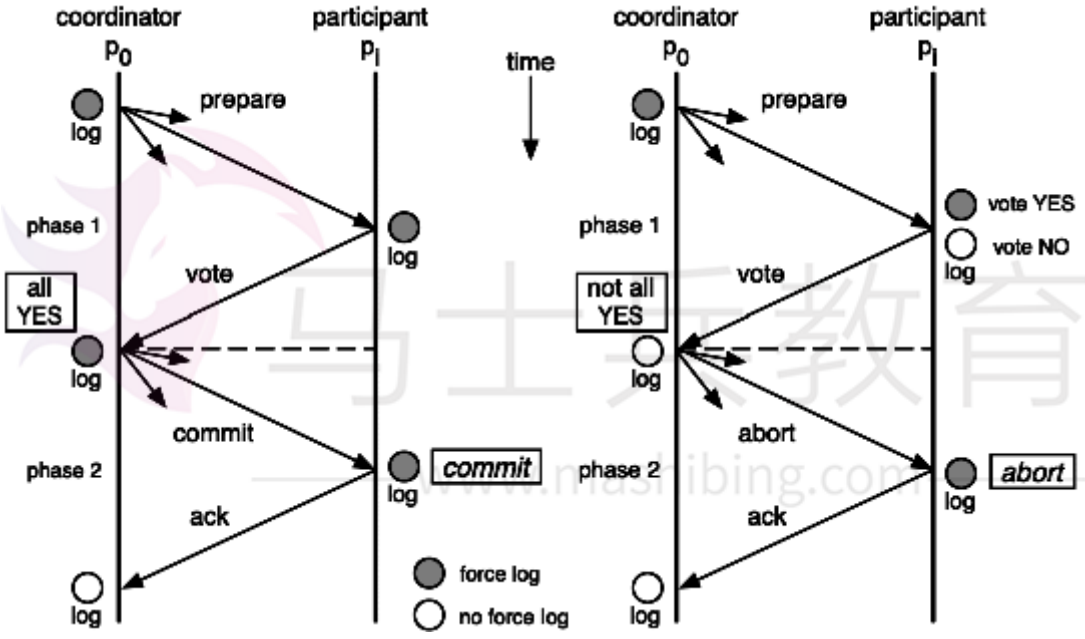
在 JavaEE 平台下，WebLogic、Webshare 等主流商用的应用服务器提供了 JTA 的实现和支持。而在 Tomcat 下是没有实现的（Tomcat 不能算是 JavaEE 应用服务器，比较轻量），这就需要借助第三方的框架 Jotm、Automikos 等来实现，两者均支持 Spring 事务整合。

在分布式事务的控制中采用了**两阶段提交协议（Two- Phase Commit Protocol）**。即事务的提交分为两个阶段：

预提交阶段(Pre-Commit Phase)　　决策后阶段（Post-Decision Phase）

为了支持两阶段提交，一个分布式更新事务中涉及到的服务器必须能够相互通信。一般来说一个服务器会被指定为"控制"或"提交"服务器并监控来自其它服务器的信息。

在一个分布式事务中，必须有一个场地的Server作为协调者(coordinator)，它能向 其它场地的Server发出请求，并对它们的回答作出响应，由它来控制一个分布式事务的提交或撤消。该分布式事务中涉及到的其它场地的Server称为参 与者（Participant）。





事务两阶段提交的过程如下：

● 两阶段提交在应用程序向协调者发出一个提交命令时被启动。这时提交进入第一阶段，即预提交阶段。在这一阶段中：

(1) 协调者准备局部（即在本地）提交并在日志中写入"预提交"日志项，并包含有该事务的所有参与者的名字。

(2) 协调者询问参与者能否提交该事务。一个参与者可能由于多种原因不能提交。例如，该Server提供的约束条件（Constraints）的延迟检查不符合 限制条件时，不能提交；参与者本身的Server进程或硬件发生故障，不能提交；或者协调者访问不到某参与者（网络故障），这时协调者都认为是收到了一个 否定的回答。

(3) 如果参与者能够提交，则在其本身的日志中写入"准备提交"日志项，该日志项立即写入硬盘，然后给协调者发回，已准备好提交"的回答。

(4) 协调者等待所有参与者的回答，如果有参与者发回否定的回答，则协调者撤消该事务并给所有参与者发出一个"撤消该事务"的消息，结束该分布式事务，撤消该事务的所有影响。

● 如果所有的参与者都送回"已准备好提交"的消息，则该事务的提交进入第二阶段，即决策后提交阶段。在这一阶段中：

(1) 协调者在日志中写入"提交"日志项，并立即写入硬盘。

(2) 协调者向参与者发出"提交该事务"的命令。各参与者接到该命令后，在各自的日志中写入"提交"日志项，并立即写入硬盘。然后送回"已提交"的消息，释放该事务占用的资源。

(3) 当所有的参与者都送回"已提交"的消息后，协调者在日志中写入"事务提交完成"日志项，释放协调者占用的资源 。这样，完成了该分布式事务的提交。

优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。

缺点： 实现复杂，牺牲了可用性，对性能影响较大，涉及多次节点间的网络通信，通信时间太长，不适合高并发高性能场景。

补偿事务（TCC）

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

- Try 阶段主要是对业务系统做检测及资源预留
- Confirm 阶段主要是对业务系统做确认提交，Try 阶段执行成功并开始执行 Confirm 阶段时，默认 Confirm 阶段是不会出错的。即：只要 Try 成功，Confirm 一定成功。
- Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

举个例子，假入 Bob 要向 Smith 转账，思路大概是：我们有一个本地方法，里面依次调用

1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。

2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。

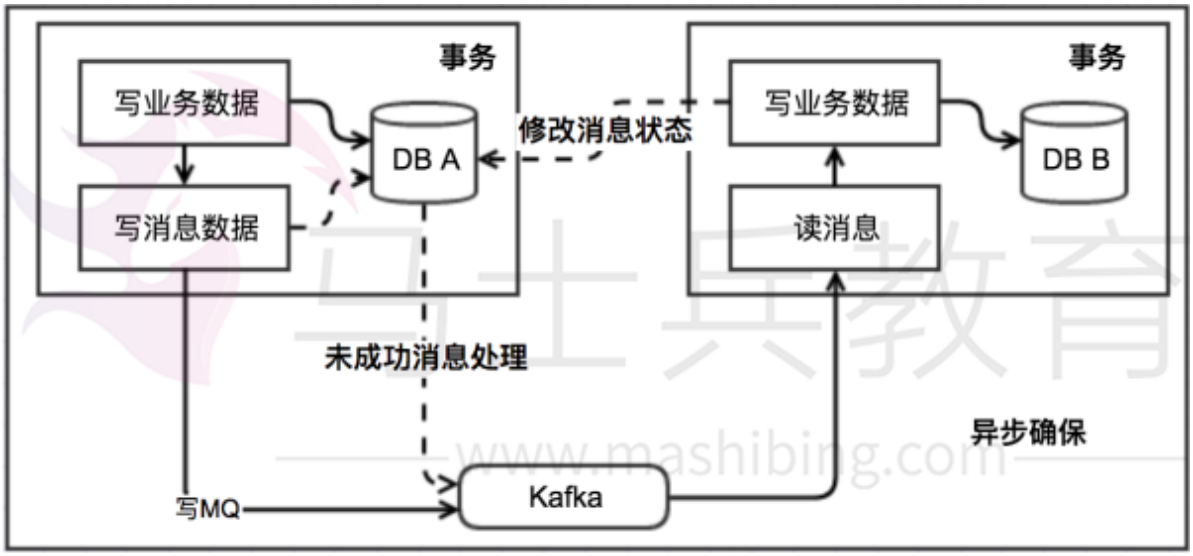
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

优点： 跟 2PC 比起来，实现以及流程相对简单了一些，但数据的一致性比 2PC 也要差一些。

缺点： 缺点还是比较明显的，在2,3步中都有可能失败。TCC 属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用 TCC 不太好定义及处理。

本地消息表（MQ 异步确保）

这种实现方式的思路，其实是源于ebay，后来通过支付宝等公司的布道，在业内广泛使用。其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。如果不考虑性能及设计优雅，借助关系型数据库中的表即可实现。



基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过 MQ 发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑（防止消息会被重复投递，增加消息应用状态表（message_apply），通俗来说就是个账本，用于记录消息的消费情况，每次来一个消息， 在真正执行之前，先去消息应用状态表中查询一遍，如果找到说明是重复消息，丢弃即可，如果没找到才执行，同时插入到消息应用状态表（同一事

务））， 这种方案还是非常实用的。

这种方案遵循 BASE 理论，采用的是最终一致性，比较适合实际业务场景的，即不会出现像 2PC 那样复杂的实现(当调用链很长的时候，2PC 的可用性是非常低的)，也不会像 TCC 那样可能出现确认或者回滚不了的情况。

优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理，而且，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。

MQ 事务消息

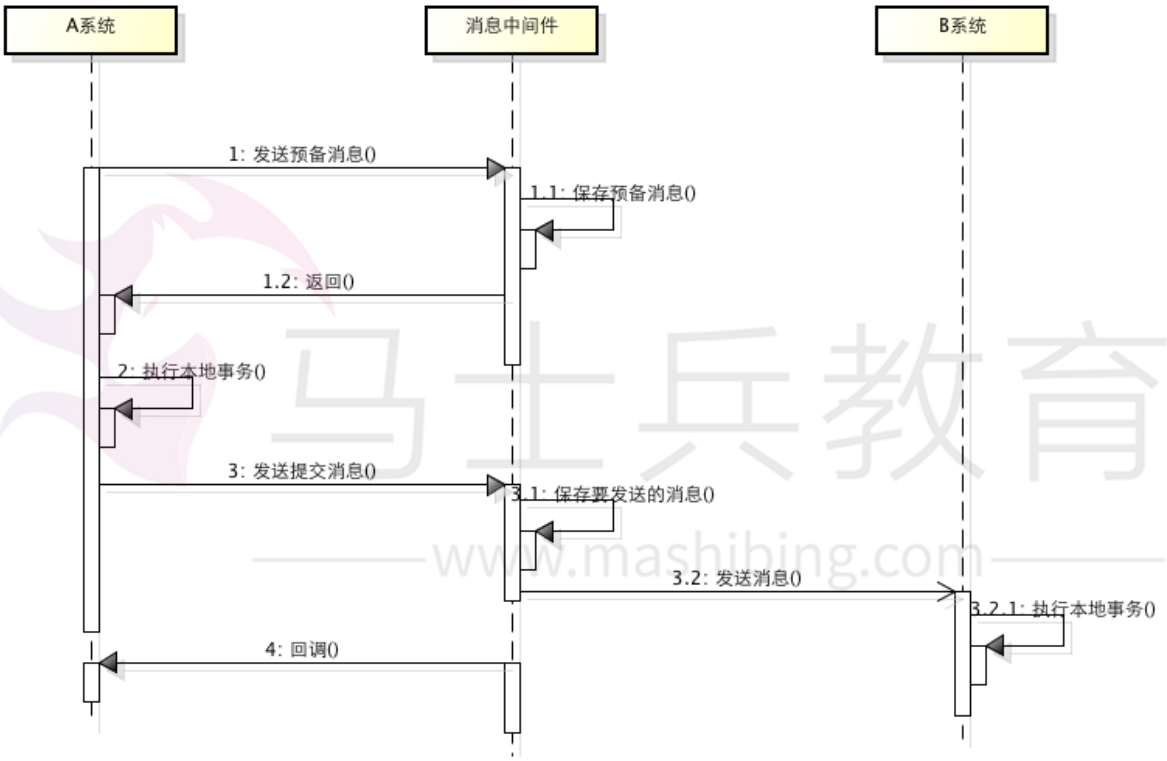
有一些第三方的 MQ 是支持事务消息的，比如 RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的 MQ 都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段 Prepared 消息，会拿到消息的地址。 第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了 RocketMQ 会定期扫描消息集群中的事务消息，这时候发现了 Prepared 消息，它会向消息发送者确认，所以生产方需要实现一个 check 接口，RocketMQ 会根据发送端设置的策略来决定是回滚还是继续发送确认消息。

这样就保证了消息发送与本地事务同时成功或同时失败，具体原理如下：



1、A系统向消息中间件发送一条预备消息

2、消息中间件保存预备消息并返回成功

3、A执行本地事务

4、A发送提交消息给消息中间件

通过以上4 步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

步骤一出错，则整个事务失败，不会执行A的本地操作

步骤二出错，则整个事务失败，不会执行A的本地操作

步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息

步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。

优点： 实现了最终一致性，不需要依赖本地数据库事务。

缺点： 实现难度大，主流MQ不支持，没有.NET客户端，RocketMQ事务消息部分代码也未开源。

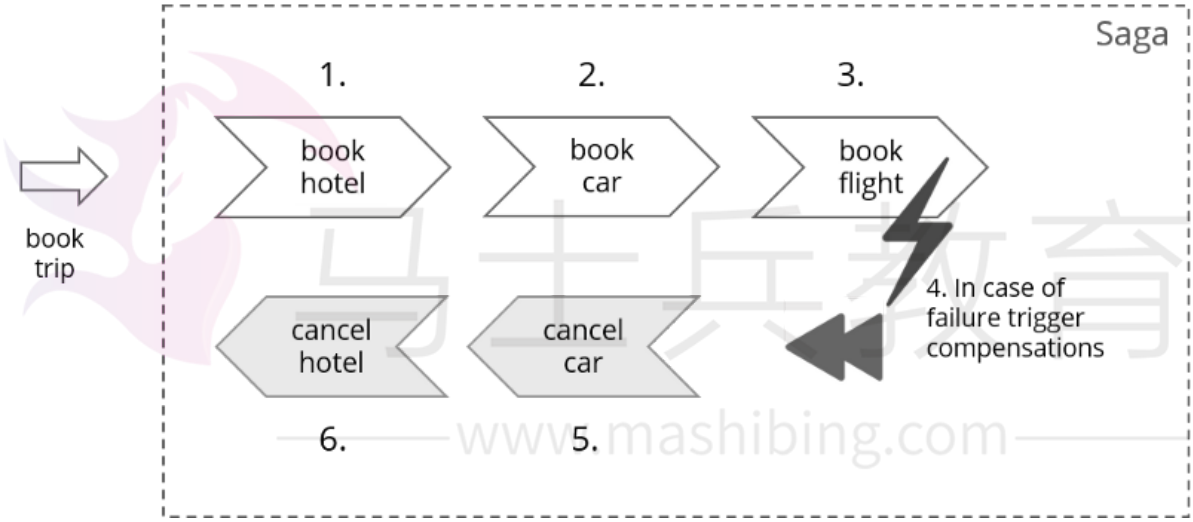
Sagas 事务模型

Saga事务模型又叫做长时间运行的事务（Long-running-transaction），它是由普林斯顿大学的H.Garcia-Molina等人提出，它描述的是另外一种在没有两阶段提交的的情况下解决分布式系统中复杂的业务事务问题。

我们这里说的是一种基于 Sagas 机制的工作流事务模型，这个模型的相关理论目前来说还是比较新的，以至于百度上几乎没有什么相关资料。

该模型其核心思想就是拆分分布式系统中的长事务为多个短事务，或者叫多个本地事务，然后由 Sagas 工作流引擎负责协调，如果整个流程正常结束，那么就算是业务成功完成，如果在这过程中实现失败，那么Sagas工作流引擎就会以相反的顺序调用补偿操作，重新进行业务回滚。

比如我们一次关于购买旅游套餐业务操作涉及到三个操作，他们分别是预定车辆，预定宾馆，预定机票，他们分别属于三个不同的远程接口。可能从我们程序的角度来说他们不属于一个事务，但是从业务角度来说是属于同一个事务的。



他们的执行顺序如上图所示，所以当发生失败时，会依次进行取消的补偿操作。

因为长事务被拆分了很多个业务流，所以 Sagas 事务模型最重要的一个部件就是工作流或者你也可以叫流程管理器（Process Manager），工作流引擎和Process Manager虽然不是同一个东西，但是在这里，他们的职责是相同的。

优缺点这里我们就不说了，因为这个理论比较新，目前市面上还没有什么解决方案，即使是 Java 领域，也没有搜索的太多有用的信息。

其他补偿方式

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了success字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。

其实这就是一个很典型的补偿例子，跟一些 MQ 重试补偿机制很类似。

一般成熟的系统中，对于级别较高的服务和接口，整体的可用性通常都会很高。如果有些业务由于瞬时的网络故障或调用超时等问题，那么这种重试机制其实是非常有效的。

当然，考虑个比较极端的场景，假如系统自身有bug或者程序逻辑有问题，那么重试1W次那也是无济于事的。那岂不是就发生了“明明已经付款，却显示未付款不发货”类似的悲剧？

其实为了交易系统更可靠，我们一般会在类似交易这种高级别的服务代码中，加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。

在某些特殊的情况下，还会有 "人工补偿" 的，这也是最后一道屏障。

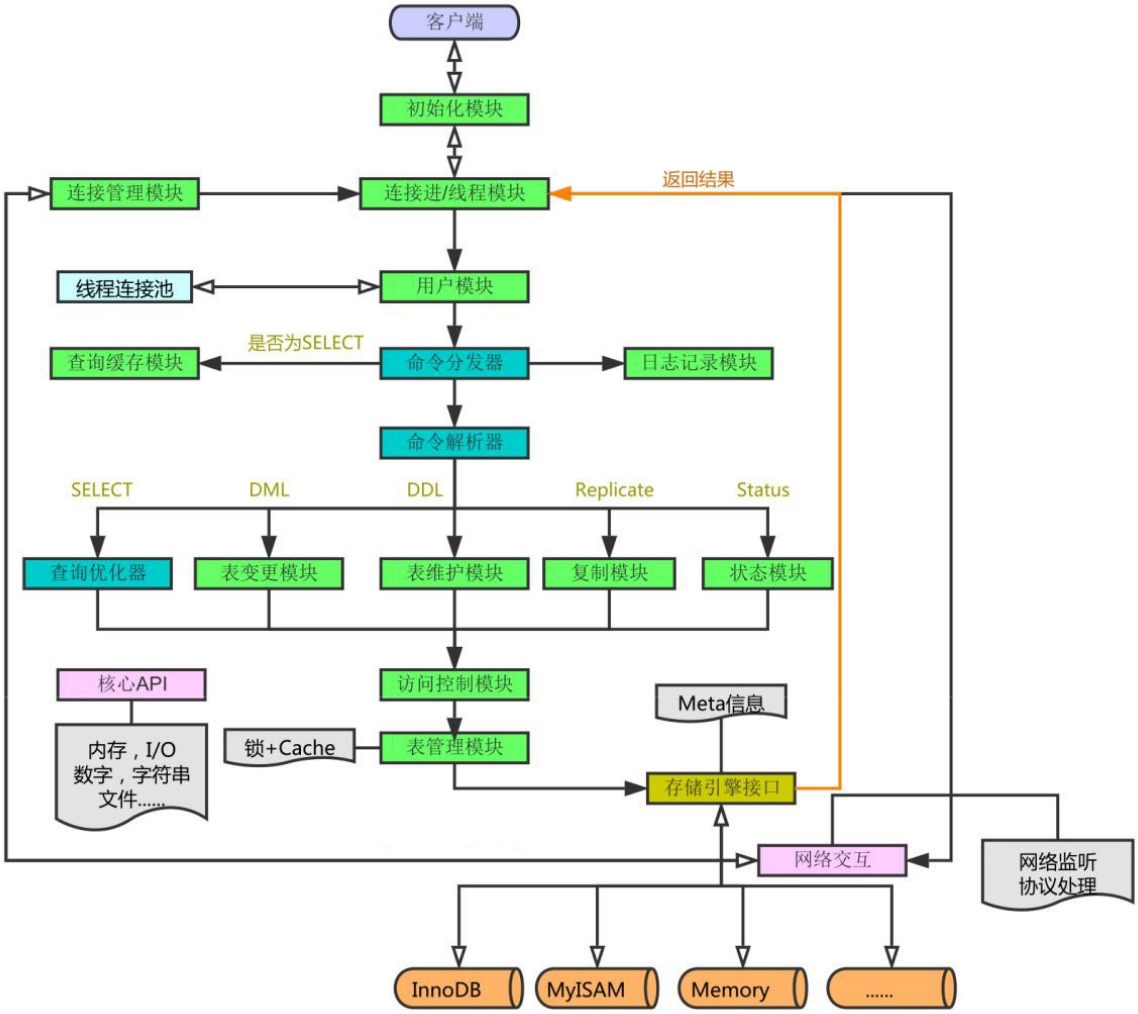
总结

分布式事务，本质上是对多个数据库的事务进行统一控制，按照控制力度可以分为：不控制、部分控制和完全控制。

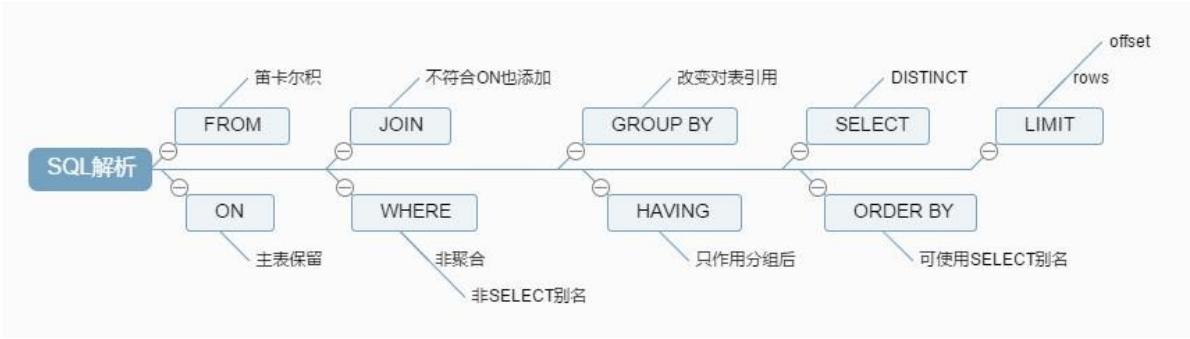
具体用哪种方式，最终还是取决于业务场景。作为技术人员，一定不能忘了技术是为业务服务的，不要为了技术而技术，针对不同业务进行技术选型也是一种很重要的能力！

SQL的整个解析、执行过程原理、SQL行转列；

整体架构



SQL解析



行转列、列转行

mysql行转列、列转行

语句不难，不做多余解释了，看语句时，从内往外一句一句剖析

行转列

有如图所示的表，现在希望查询的结果将行转成列

ID	USER_NAME	COURSE	SCORE
2	张三	数学	34
3	张三	语文	58
4	张三	英语	58
5	李四	数学	45
6	李四	语文	87
7	李四	英语	45
8	王五	数学	76
9	王五	语文	34
10	王五	英语	89

建表语句如下：

```
CREATE TABLE `TEST_TB_GRADE` (
  `ID` int(10) NOT NULL AUTO_INCREMENT,
  `USER_NAME` varchar(20) DEFAULT NULL,
  `COURSE` varchar(20) DEFAULT NULL,
  `SCORE` float DEFAULT '0',
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;1234567
insert into TEST_TB_GRADE(USER_NAME, COURSE, SCORE)  values
("张三", "数学", 34),
("张三", "语文", 58),
("张三", "英语", 58),
("李四", "数学", 45),
("李四", "语文", 87),
("李四", "英语", 45),
("王五", "数学", 76),
("王五", "语文", 34),
("王五", "英语", 89);12345678910
```

查询语句：

此处用之所以用MAX是为了将无数据的点设为0，防止出现NULL

```
SELECT user_name ,
    MAX(CASE course WHEN '数学' THEN score ELSE 0 END ) 数学,
    MAX(CASE course WHEN '语文' THEN score ELSE 0 END ) 语文,
    MAX(CASE course WHEN '英语' THEN score ELSE 0 END ) 英语
FROM test_tb_grade
GROUP BY USER_NAME;123456
```

结果展示：

信息 | 结果1 | 配置文件 | 状态

USER_NAME	数学	语文	英语
张三	34	58	58
李四	45	87	45
王五	76	34	89

列转行

有如图所示的表，现在希望查询的结果将列成行

ID	USER_NAME	CN_SCORE	MATH_SCORE	EN_SCORE
2	张三	34	58	58
3	李四	45	87	45
4	王五	76	34	89

建表语句如下：

```
CREATE TABLE `TEST_TB_GRADE2` (
  `ID` int(10) NOT NULL AUTO_INCREMENT,
  `USER_NAME` varchar(20) DEFAULT NULL,
  `CN_SCORE` float DEFAULT NULL,
  `MATH_SCORE` float DEFAULT NULL,
  `EN_SCORE` float DEFAULT '0',
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;12345678
insert into TEST_TB_GRADE2(USER_NAME, CN_SCORE, MATH_SCORE, EN_SCORE) values
("张三", 34, 58, 58),
("李四", 45, 87, 45),
("王五", 76, 34, 89);1234
```

查询语句：

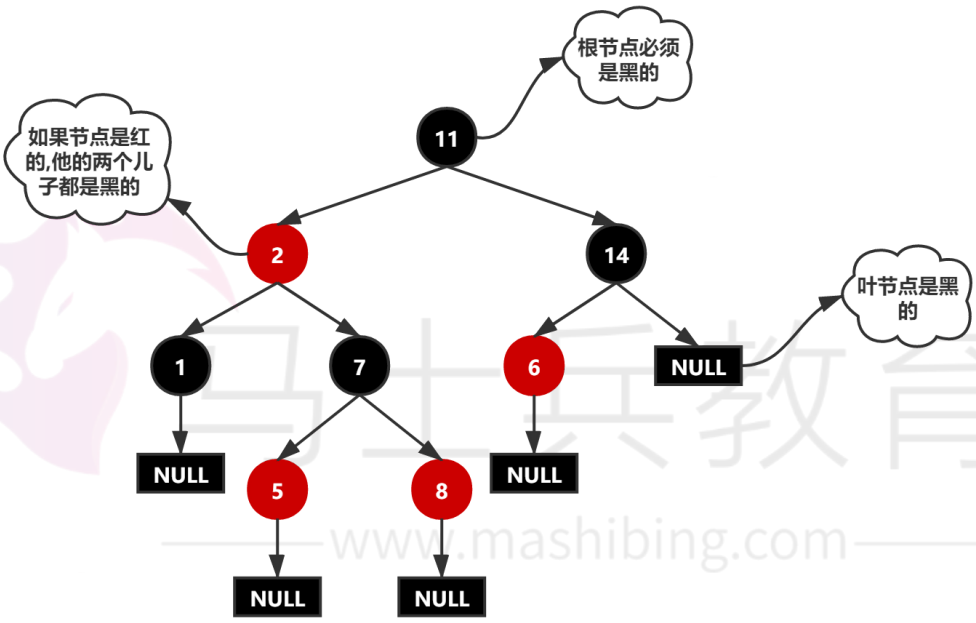
```
select user_name, '语文' COURSE , CN_SCORE as SCORE from test_tb_grade2
union select user_name, '数学' COURSE, MATH_SCORE as SCORE from test_tb_grade2
union select user_name, '英语' COURSE, EN_SCORE as SCORE from test_tb_grade2
order by user_name,COURSE;1234
```

结果展示：

信息 | 结果1 | 配置文件 | 状态

user_name	COURSE	SCORE
张三	数学	58
张三	英语	58
张三	语文	34
李四	数学	87
李四	英语	45
李四	语文	45
王五	数学	34
王五	英语	89
王五	语文	76

红黑树的实现原理和应用场景；



红黑树(一棵自平衡的排序二叉树)五大特性：

1）每个结点要么是红的，要么是黑的。

2）根结点是黑的。

3）每个叶结点，即空结点是黑的。

4）如果一个结点是红的，那么它的俩个儿子都是黑的。

5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

场景

1）广泛用于C++的STL中,map和set都是用红黑树实现的.

2）著名的linux进程调度Completely Fair Scheduler,用红黑树管理进程控制块,进程的虚拟内存 区域都存储在一颗红黑树上,每个虚拟地址区域都对应红黑树的一个节点,左指针指向相邻的地址 虚拟存储区域,右指针指向相邻的高地址虚拟地址空间.

3）IO多路复用epoll的实现采用红黑树组织管理sockfd，以支持快速的增删改查.

4）ngnix中,用红黑树管理timer,因为红黑树是有序的,可以很快的得到距离当前最小的定时器.

5）java中的TreeSet,TreeMap

MySql的存储引擎的不同

MySQL存储引擎之Myisam和Innodb总结性梳理

MyISAM是MySQL的默认数据库引擎（5.5版之前），由早期的ISAM（Indexed Sequential Access Method：有索引的顺序访问方法）所改良。虽然性能极佳，但却有一个缺点：不支持事务处理（transaction）。不过，在这几年的发展下，MySQL也导入了InnoDB（另一种数据库引擎），以强化参考完整性与并发违规处理机制，后来就逐渐取代MyISAM。

InnoDB，是MySQL的数据库引擎之一，为MySQL AB发布binary的标准之一。InnoDB由Innobase Oy公司所开发，2006年五月时由甲骨文公司并购。与传统的ISAM与MyISAM相比，InnoDB的最大特色就是支持了ACID兼容的事务（Transaction）功能，类似于PostgreSQL。目前InnoDB采用双轨制授权，一是GPL授权，另一是专有软件授权。

MyISAM和InnoDB两者之间有着明显区别，简单梳理如下:

1) 事务支持

MyISAM不支持事务，而InnoDB支持。InnoDB的AUTOCOMMIT默认是打开的，即每条SQL语句会默认被封装成一个事务，自动提交，这样会影响速度，所以最好是把多条SQL语句显示放在begin和commit之间，组成一个事务去提交。

MyISAM是非事务安全型的，而InnoDB是事务安全型的，默认开启自动提交，宜合并事务，一同提交，减小数据库多次提交导致的开销，大大提高性能。

2) 存储结构

MyISAM：每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。
InnoDB：所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。

3) 存储空间

MyISAM：可被压缩，存储空间较小。支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。
InnoDB：需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

4) 可移植性、备份及恢复

MyISAM：数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。
InnoDB：免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

5) 事务支持

MyISAM：强调的是性能，每次查询具有原子性,其执行数度比InnoDB类型更快，但是不提供事务支持。
InnoDB：提供事务支持事务，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

6) AUTO_INCREMENT

MyISAM：可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。
InnoDB：InnoDB中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

7) 表锁差异

MyISAM：只支持表级锁，用户在操作myisam表时，select，update，delete，insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。
InnoDB：支持事务和行级锁，是innodb的最大特色。行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。

MyISAM锁的粒度是表级，而InnoDB支持行级锁定。简单来说就是, InnoDB支持数据行锁定，而MyISAM不支持行锁定，只支持锁定整个表。即MyISAM同一个表上的读锁和写锁是互斥的，MyISAM并发读写时如果等待队列中既有读请求又有写请求，默认写请求的优先级高，即使读请求先到，所以MyISAM不适合于有大量查询和修改并存的情况，那样查询进程会长时间阻塞。因为MyISAM是锁表，所以某项读操作比较耗时会使其他写进程饿死。

8) 全文索引
MyISAM：支持(FULLTEXT类型的)全文索引
InnoDB：不支持(FULLTEXT类型的)全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。

全文索引是指对char、varchar和text中的每个词（停用词除外）建立倒排序索引。MyISAM的全文索引其实没啥用，因为它不支持中文分词，必须由使用者分词后加入空格再写到数据表里，而且少于4个汉字的词会和停用词一样被忽略掉。

另外，MyIsam索引和数据分离，InnoDB在一起，MyIsam天生非聚簇索引，最多有一个unique的性质，InnoDB的数据文件本身就是主键索引文件，这样的索引被称为“聚簇索引”

9) 表主键
MyISAM：允许没有任何索引和主键的表存在，索引都是保存行的地址。
InnoDB：如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。InnoDB的主键范围更大，最大是MyISAM的2倍。

10) 表的具体行数
MyISAM：保存有表的总行数，如果select count() *from table;会直接取出出该值。*
InnoDB：没有保存表的总行数(只能遍历)，如果使用select count() from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，myisam和innodb处理的方式都一样。

11) CURD操作
MyISAM：如果执行大量的SELECT，MyISAM是更好的选择。
InnoDB：如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。
DELETE 从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。

12) 外键
MyISAM：不支持
InnoDB：支持

13) 查询效率
没有where的count()*使用MyISAM要比InnoDB快得多。因为MyISAM内置了一个计数器，count()*时它直接从计数器中读，而InnoDB必须扫描全表。所以在InnoDB上执行count()*时一般要伴随where，且where中要包含主键以外的索引列。为什么这里特别强调"主键以外"？因为InnoDB中primary index是和raw data存放在一起的，而secondary index则是单独存放，然后有个指针指向primary key。所以只是count()*的话使用secondary index扫描更快，而primary key则主要在扫描索引同时要返回raw data时的作用较大。
MyISAM相对简单，所以在效率上要优于InnoDB，小型应用可以考虑使用MyISAM。

通过上述的分析，基本上可以考虑使用InnoDB来替代MyISAM引擎了，原因是InnoDB自身很多良好的特点，比如事务支持、存储 过程、视图、行级锁定等等，在并发很多的情况下，相信InnoDB的表现肯定要比MyISAM强很多。另外，任何一种表都不是万能的，只用恰当的针对业务类型来选择合适的表类型，才能最大的发挥MySQL的性能优势。如果不是很复杂的Web应用，非关键应用，还是可以继续考虑MyISAM的，这个具体情况可以自己斟酌。

MyISAM和InnoDB两者的应用场景：
1) MyISAM管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的SELECT查询，那么MyISAM是更好的选择。
2) InnoDB用于事务处理应用程序，具有众多特性，包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。

但是实际场景中，针对具体问题需要具体分析，一般而言可以遵循以下几个问题：
- 数据库是否有外键？
- 是否需要事务支持？
- 是否需要全文索引？
- 数据库经常使用什么样的查询模式？在写多读少的应用中还是Innodb插入性能更稳定，在并发情况下

也能基本，如果是对读取速度要求比较快的应用还是选MyISAM。
- 数据库的数据有多大？ 大尺寸倾向于innodb，因为事务日志，故障恢复。

Mysql优化系列--Innodb引擎下mysql自身配置优化

1.简单介绍
InnoDB给MySQL提供了具有提交，回滚和崩溃恢复能力的事务安全（ACID兼容）存储引擎。InnoDB锁定在行级并且也在SELECT语句提供一个Oracle风格一致的非锁定读。这些特色增加了多用户部署和性能。没有在InnoDB中扩大锁定的需要，因为在InnoDB中行级锁定适合非常小的空间。InnoDB也支持FOREIGN KEY强制。在SQL查询中，你可以自由地将InnoDB类型的表与其它MySQL的表的类型混合起来，甚至在同一个查询中也可以混合。

2.之所以选用innodb作为存储引擎的考虑
目前来说，InnoDB是为Mysql处理巨大数据量时的最大性能设计。它的CPU效率可能是任何其它基于磁盘的关系数据库引擎所不能匹敌的。在数据量大的网站或是应用中Innodb是倍受青睐的。
另一方面，在数据库的复制操作中Innodb也是能保证master和slave数据一致有一定的作用。

3.下面是对线上mysql5.6版本的数据库的配置进行的优化分析记录：
1）内存利用方面：
innodb_buffer_pool_size
这个是Innodb最重要的参数，和MyISAM的key_buffer_size有相似之处，但也是有差别的。
这个参数主要缓存innodb表的索引，数据，插入数据时的缓冲。
该参数分配内存的原则：
这个参数默认分配只有8M，可以说是非常小的一个值。
如果是一个专用DB服务器，那么他可以占到内存的70%-80%。
这个参数不能动态更改，所以分配需多考虑。分配过大，会使Swap占用过多，致使Mysql的查询特慢。
如果你的数据比较小，那么可分配是你的数据大小+10%左右做为这个参数的值。
例如：数据大小为50M,那么给这个值分配innodb_buffer_pool_size = 64M
设置方法，在my.cnf文件里：
innodb_buffer_pool_size=4G

注意：
在Mysql5.7版本之前，调整innodb_buffer_pool_size大小必须在my.cnf配置里修改，然后重启mysql进程才可以生效。
如今到了Mysql5.7版本，就可以直接动态调整这个参数，方便了很多。

尤其是在服务器内存增加之后，运维人员不能粗心大意，要记得调大Innodb_Buffer_Pool_size这个参数。
数据库配置后，要注意检查Innodb_Buffer_Pool_size这个参数的设置是否合理

需要注意的地方：
在调整innodb_buffer_pool_size 期间，用户的请求将会阻塞，直到调整完毕，所以请勿在白天调整，在凌晨3-4点低峰期调整。
调整时，内部把数据页移动到一个新的位置，单位是块。如果想增加移动的速度，需要调整innodb_buffer_pool_chunk_size参数的大小，默认是128M。

Mysql5.7中动态调整这个参数的操作记录（例如由128M增大为384M）：
134217728/1024*1024=128M
mysql> SELECT @@innodb_buffer_pool_size;

+----------------------------+

| @@innodb_buffer_pool_size |

+----------------------------+

| 134217728 |

+----------------------------+

1 row in set (0.00 sec)

mysql> SELECT @@innodb_buffer_pool_chunk_size;

+---------------------------------+

| @@innodb_buffer_pool_chunk_size |

+---------------------------------+

| 134217728 |

+---------------------------------+

1 row in set (0.00 sec)

mysql> SET GLOBAL innodb_buffer_pool_size=402653184;

Query OK, 0 rows affected (0.01 sec)

mysql> SELECT @@innodb_buffer_pool_size;

+----------------------------+

| @@innodb_buffer_pool_size |

+----------------------------+

| 402653184 |

+----------------------------+

1 row in set (0.00 sec)

innodb_buffer_pool_chunk_size的大小，计算公式是
innodb_buffer_pool_size/innodb_buffer_pool_instances

比如现在初始化innodb_buffer_pool_size为2G，innodb_buffer_pool_instances实例为4，
innodb_buffer_pool_chunk_size设置为1G，那么会自动把innodb_buffer_pool_chunk_size 1G调整为512M.
例：
./mysqld --innodb_buffer_pool_size=2147483648 --innodb_buffer_pool_instances=4
--innodb_buffer_pool_chunk_size=1073741824;

mysql> SELECT @@innodb_buffer_pool_size;

+----------------------------+

| @@innodb_buffer_pool_size |

+----------------------------+

| 2147483648 |

+---------------------------+

1 row in set (0.00 sec)

mysql> SELECT @@innodb_buffer_pool_instances;

+--------------------------------+

| @@innodb_buffer_pool_instances |

+--------------------------------+

| 4 |

+--------------------------------+

1 row in set (0.00 sec)

Chunk size was set to 1GB (1073741824 bytes) on startup but was

truncated to innodb_buffer_pool_size / innodb_buffer_pool_instances

mysql> SELECT @@innodb_buffer_pool_chunk_size;

+---------------------------------+

| @@innodb_buffer_pool_chunk_size |

+---------------------------------+

| 536870912 |

+---------------------------------+

1 row in set (0.00 sec)

监控Buffer Pool调整进程

mysql> SHOW STATUS WHERE Variable_name='InnoDB_buffer_pool_resize_status';

+----------------------------------+----------------------------------+

| Variable_name | Value |

+----------------------------------+----------------------------------+

| Innodb_buffer_pool_resize_status | Resizing also other hash tables. |

+----------------------------------+----------------------------------+

1 row in set (0.00 sec)

查看错误日志:
（增大）

[Note] InnoDB: Resizing buffer pool from 134217728 to 4294967296. (unit=134217728)

[Note] InnoDB: disabled adaptive hash index.

[Note] InnoDB: buffer pool 0 : 31 chunks (253952 blocks) was added.

[Note] InnoDB: buffer pool 0 : hash tables were resized.

[Note] InnoDB: Resized hash tables at lock_sys, adaptive hash index, dictionary.

[Note] InnoDB: completed to resize buffer pool from 134217728 to 4294967296.

[Note] InnoDB: re-enabled adaptive hash index.

（减少）

[Note] InnoDB: Resizing buffer pool from 4294967296 to 134217728. (unit=134217728)

[Note] InnoDB: disabled adaptive hash index.

[Note] InnoDB: buffer pool 0 : start to withdraw the last 253952 blocks.

[Note] InnoDB: buffer pool 0 : withdrew 253952 blocks from free list. tried to relocate 0 pages. (253952/253952)

[Note] InnoDB: buffer pool 0 : withdrawn target 253952 blocks.

[Note] InnoDB: buffer pool 0 : 31 chunks (253952 blocks) was freed.

[Note] InnoDB: buffer pool 0 : hash tables were resized.

[Note] InnoDB: Resized hash tables at lock_sys, adaptive hash index, dictionary.

[Note] InnoDB: completed to resize buffer pool from 4294967296 to 134217728.

[Note] InnoDB: re-enabled adaptive hash index.

---

innodb_additional_mem_pool_size
用来存放Innodb的内部目录，这个值不用分配太大，系统可以自动调。通常设置16M够用了，如果表比较多，可以适当的增大。
设置方法，在my.cnf文件里：
innodb_additional_mem_pool_size = 16M

2）关于日志方面：
innodb_log_file_size
作用：指定在一个日志组中，每个log的大小。
结合innodb_buffer_pool_size设置其大小，25%-100%。避免不需要的刷新。
注意：这个值分配的大小和数据库的写入速度，事务大小，异常重启后的恢复有很大的关系。一般取256M可以兼顾性能和recovery的速度。
分配原则：几个日值成员大小加起来差不多和你的innodb_buffer_pool_size相等。上限为每个日值上限大小为4G.一般控制在几个Log文件相加大小在2G以内为佳。具体情况还需要看你的事务大小，数据大小为依据。
说明：这个值分配的大小和数据库的写入速度，事务大小，异常重启后的恢复有很大的关系。
设置方法：在my.cnf文件里：
innodb_log_file_size = 256M

innodb_log_files_in_group
作用：指定你有几个日值组。
分配原则：　一般我们可以用2-3个日值组。默认为两个。
设置方法：在my.cnf文件里：
innodb_log_files_in_group=3

innodb_log_buffer_size：
作用：事务在内存中的缓冲，也就是日志缓冲区的大小， 默认设置即可，具有大量事务的可以考虑设置为16M。
如果这个值增长过快，可以适当的增加innodb_log_buffer_size
另外如果你需要处理大理的TEXT，或是BLOB字段，可以考虑增加这个参数的值。
设置方法：在my.cnf文件里：
innodb_log_buffer_size=3M

innodb_flush_logs_at_trx_commit
作用：控制事务的提交方式,也就是控制log的刷新到磁盘的方式。
分配原则：这个参数只有3个值（0，1，2）.默认为1，性能更高的可以设置为0或是2，这样可以适当的减少磁盘IO（但会丢失一秒钟的事务。），游戏库的MySQL建议设置为0。主库请不要更改了。
其中：
0：log buffer中的数据将以每秒一次的频率写入到log file中，且同时会进行文件系统到磁盘的同步操作，但是每个事务的commit并不会触发任何log buffer 到log file的刷新或者文件系统到磁盘的刷新操作；
1：（默认为1）在每次事务提交的时候将logbuffer 中的数据都会写入到log file，同时也会触发文件系统到磁盘的同步；
2：事务提交会触发log buffer 到log file的刷新，但并不会触发磁盘文件系统到磁盘的同步。此外，每秒会有一次文件系统到磁盘同步操作。
说明：
这个参数的设置对Ｉｎｎｏｄｂ的性能有很大的影响，所以在这里给多说明一下。
当这个值为1时：innodb 的事务LOG在每次提交后写入日值文件，并对日值做刷新到磁盘。这个可以做到不丢任何一个事务。
当这个值为2时：在每个提交，日志缓冲被写到文件，但不对日志文件做到磁盘操作的刷新,在对日志文件的刷新在值为2的情况也每秒发生一次。但需要注意的是，由于进程调用方面的问题，并不能保证每秒１００％的发生。从而在性能上是最快的。但操作系统崩溃或掉电才会删除最后一秒的事务。
当这个值为0时：日志缓冲每秒一次地被写到日志文件，并且对日志文件做到磁盘操作的刷新，但是在一个事务提交不做任何操作。mysqld进程的崩溃会删除崩溃前最后一秒的事务。
从以上分析，当这个值不为１时，可以取得较好的性能，但遇到异常会有损失，所以需要根据自已的情况去衡量。
设置方法：在my.cnf文件里：
innodb_flush_logs_at_trx_commit=1

3）文件IO分配，空间占用方面
innodb_file_per_table
作用：使每个Innodb的表，有自已独立的表空间。如删除文件后可以回收那部分空间。默认是关闭的，建议打开（innodb_file_per_table=1）
分配原则：只有使用不使用。但DB还需要有一个公共的表空间。
设置方法：在my.cnf文件里：
innodb_file_per_table=1

innodb_file_io_threads
作用：文件读写IO数，这个参数只在Windows上起作用。在Linux上只会等于4，默认即可！
设置方法：在my.cnf文件里：
innodb_file_io_threads=4

innodb_open_files
作用：限制Innodb能打开的表的数据。
分配原则：这个值默认是300。如果库里的表特别多的情况，可以适当增大为1000。innodb_open_files的大小对InnoDB效率的影响比较小。但是在InnoDBcrash的情况下，innodb_open_files设置过小会影响recovery的效率。所以用InnoDB的时候还是把innodb_open_files放大一些比较合适。
设置方法：在my.cnf文件里：
innodb_open_files=800

innodb_data_file_path
指定表数据和索引存储的空间，可以是一个或者多个文件。最后一个数据文件必须是自动扩充的，也只有最后一个文件允许自动扩充。这样，当空间用完后，自动扩充数据文件就会自动增长（以8MB为单位）以容纳额外的数据。
例如： innodb_data_file_path=/disk1/ibdata1:900M;/disk2/ibdata2:50M:autoextend 两个数据文件放在不同的磁盘上。数据首先放在ibdata1 中，当达到900M以后，数据就放在ibdata2中。
设置方法，在my.cnf文件里：

innodb_data_file_path
=ibdata1:1G;ibdata2:1G;ibdata3:1G;ibdata4:1G;ibdata5:1G;ibdata6:1G:autoextend

innodb_data_home_dir
放置表空间数据的目录，默认在mysql的数据目录，设置到和MySQL安装文件不同的分区可以提高性能。
设置方法，在my.cnf文件里：（比如mysql的数据目录是/data/mysql/data，这里可以设置到不通的分区/home/mysql下）
innodb_data_home_dir = /home/mysql

4）其它相关参数（适当的增加table_cache）
这里说明一个比较重要的参数：
innodb_flush_method
作用：Innodb和系统打交道的一个IO模型
分配原则：
Windows不用设置。
linux可以选择：O_DIRECT
直接写入磁盘，禁止系统Cache了
设置方法：在my.cnf文件里：
innodb_flush_method=O_DIRECT

innodb_max_dirty_pages_pct
作用：在buffer pool缓冲中，允许Innodb的脏页的百分比，值在范围1-100,默认为90，建议保持默认。
这个参数的另一个用处：当Innodb的内存分配过大，致使Swap占用严重时，可以适当的减小调整这个值，使达到Swap空间释放出来。建义：这个值最大在90%，最小在15%。太大，缓存中每次更新需要致换数据页太多，太小，放的数据页太小，更新操作太慢。
设置方法：在my.cnf文件里：
innodb_max_dirty_pages_pct = 90
动态更改需要有管理员权限：
set global innodb_max_dirty_pages_pct=50;

innodb_thread_concurrency
同时在Innodb内核中处理的线程数量。建议默认值。
设置方法，在my.cnf文件里：
innodb_thread_concurrency = 16

5）公共参数调优
skip-external-locking
MyISAM存储引擎也同样会使用这个参数，MySQL4.0之后，这个值默认是开启的。
作用是避免MySQL的外部锁定(老版本的MySQL此参数叫做skip-locking)，减少出错几率增强稳定性。建议默认值。
设置方法，在my.cnf文件里：
skip-external-locking

skip-name-resolve
禁止MySQL对外部连接进行DNS解析（默认是关闭此项设置的，即默认解析DNS），使用这一选项可以消除MySQL进行DNS解析的时间。
但需要注意，如果开启该选项，则所有远程主机连接授权都要使用IP地址方式，否则MySQL将无法正常处理连接请求！如果需要，可以设置此项。
设置方法，在my.cnf文件里：（我这线上mysql数据库中打开了这一设置）
skip-name-resolve

max_connections
设置最大连接（用户）数，每个连接MySQL的用户均算作一个连接，max_connections的默认值为100。此值需要根据具体的连接数峰值设定。
设置方法，在my.cnf文件里：
max_connections = 3000

query_cache_size
查询缓存大小，如果表的改动非常频繁，或者每次查询都不同，查询缓存的结果会减慢系统性能。可以设置为0。
设置方法，在my.cnf文件里：
query_cache_size = 512M

sort_buffer_size
connection级的参数，排序缓存大小。一般设置为2-4MB即可。
设置方法，在my.cnf文件里：
sort_buffer_size = 1024M

read_buffer_size
connection级的参数。一般设置为2-4MB即可。
设置方法，在my.cnf文件里：
read_buffer_size = 1024M

max_allowed_packet
网络包的大小，为避免出现较大的网络包错误，建议设置为16M
设置方法，在my.cnf文件里：
max_allowed_packet = 16M

table_open_cache
当某一连接访问一个表时，MySQL会检查当前已缓存表的数量。如果该表已经在缓存中打开，则会直接访问缓存中的表，以加快查询速度；如果该表未被缓存，则会将当前的表添加进缓存并进行查询。
通过检查峰值时间的状态值Open_tables和Opened_tables，可以决定是否需要增加table_open_cache的值。
如果发现open_tables等于table_open_cache，并且opened_tables在不断增长，那么就需要增加table_open_cache的值;设置为512即可满足需求。
设置方法，在my.cnf文件里：
table_open_cache = 512

myisam_sort_buffer_size
实际上这个myisam_sort_buffer_size参数意义不大，这是个字面上蒙人的参数，它用于ALTER TABLE, OPTIMIZE TABLE, REPAIR TABLE 等命令时需要的内存。默认值即可。
设置方法，在my.cnf文件里：
myisam_sort_buffer_size = 8M

thread_cache_size
线程缓存，如果一个客户端断开连接，这个线程就会被放到thread_cache_size中（缓冲池未满），SHOW STATUS LIKE 'threads%';如果 Threads_created 不断增大，那么当前值设置要改大，改到 Threads_connected 值左右。（通常情况下，这个值改善性能不大），默认8即可
设置方法，在my.cnf文件里：
thread_cache_size = 8

innodb_thread_concurrency
线程并发数，建议设置为CPU内核数*2
设置方法，在my.cnf文件里：
innodb_thread_concurrency = 8

key_buffer_size
仅作用于 MyISAM存储引擎，用来设置用于缓存 MyISAM存储引擎中索引文件的内存区域大小。如果我们有足够的内存，这个缓存区域最好是能够存放下我们所有的 MyISAM 引擎表的所有索引，以尽可能提高性能。不要设置超过可用内存的30%。即使不用MyISAM表，也要设置该值8-64M，用于临时表。
设置方法，在my.cnf文件里：
key_buffer_size = 8M

- - - - - - - - - - - - 影响InnoDB性能的一些重要参数 - - - - - - - - - - - - - -

1）InnoDB_buffer_pool_size
这个参数定义InnoDB存储引擎的表数据和索引数据的最大内存缓冲区,InnoDB_buffer_pool_size参数同时提供为数据块和索引块做缓存.这个值设置的越高,访问表中数据需要的磁盘IO就越少.

2）InnoDB_flush_log_at_trx_commit
这个参数控制缓冲区的数据写入到日志文件以及日志文件数据刷新到磁盘的操作时机.在正式环境中建议设置成1。
设置0时日志缓冲每秒一次被写到日志文件,并且对日志文件做向磁盘刷新的操作,但是在一个事物提交不做任何操作.
设置1时在每个事物提交时,日志缓冲被写到日志文件,并且对日志文件做向磁盘刷新的操作
设置2时在每个事物提交时,日志缓冲被写到日志文件,但不对日志文件做向磁盘刷新的操作,对日志文件每秒向磁盘做一次刷新操作.

3）InnoDB_additional_mem_pool_size
这个参数是InnoDB用来存储数据库结构和其他内部数据结构的内存池.应用程序的表越多,则需要从这里分配越多的内存,如果用光这个池,则会从OS层分配.

4）InnoDB_lock_wait_timeout
这个参数自动检测行锁导致的死锁并进行相应处理,但是对于表锁导致的死锁不能自动检测默认值为50秒.

5）InnoDB_support_xa
这个参数设置MySQL是否支持分布式事务

6）InnoDB_log_buffer_size
这个参数日志缓冲大小

7）InnoDB_log_file_size
这个参数是一个日志组中每个日志文件的大小,此参数在高写入负载尤其是大数据集的情况下很重要.这个值越大则性能相对越高,但好似副作用是一旦系统崩溃恢复的时间会加长.

8）Innodb_io_capacity
这个参数刷新脏页数量和合并插入数量，改善磁盘IO处理能力

9）Innodb_use_native_aio
异步I/O在一定程度上提高系统的并发能力，在Linux系统上，可以通过将MySQL的服务器此参数的值设定为ON设定InnoDB可以使用Linux的异步I/O子系统.

10）Innodb_read_io_threads
这个参数可调整的读请求的后台线程数

11）Innodb_write_io_threads
这个参数可调整的写请求的后台线程数

12）InnoDB_buffer_pool_instances
这个参数能较好的运行于多核处理器，支持使用 此参数对服务器变量建立多个缓冲池实例，每个缓冲池实例分别自我管理空闲列表、列表刷写、LRU以及其它跟缓冲池相关的数据结构，并通过各自的互斥锁进行保护

13）InnoDB_purge_threads
MySQL5.5以前碎片回收操作是主线程的一部分，这经定期调度的方式运行，但会阻塞数据库的其他操作.到5.5以后，可以将这个线程独立出来； 这个能让碎片回收得更及时而且不影响其他线程的操作

14）Innodb_flush_method
这个参数控制着innodb数据文件及redo log的打开、刷写模式，对于这个参数，文档上是这样描述的：
有三个值：fdatasync(默认)，O_DSYNC，O_DIRECT
默认是fdatasync，调用fsync()去刷数据文件与redo log的buffer
为O_DSYNC时，innodb会使用O_SYNC方式打开和刷写redo log,使用fsync()刷写数据文件
为O_DIRECT时，innodb使用O_DIRECT打开数据文件，使用fsync()刷写数据文件跟redo log
总结一下三者写数据方式：
fdatasync模式：写数据时，write这一步并不需要真正写到磁盘才算完成（可能写入到操作系统buffer中就会返回完成），真正完成是flush操作，buffer交给操作系统去flush,并且文件的元数据信息也都需要更新到磁盘。
O_DSYNC模式：写日志操作是在write这步完成，而数据文件的写入是在flush这步通过fsync完成
O_DIRECT模式：数据文件的写入操作是直接从mysql innodb buffer到磁盘的，并不用通过操作系统的缓冲，而真正的完成也是在flush这步,日志还是要经过OS缓冲

使用下面命令就可以查看到上面参数的设置：
mysql> show variables like "%innodb%";

--
下面是线上mysql（innodb）的my.cnf配置参考：

```
[client]
port = 3306
socket = /usr/local/mysql/var/mysql.sock

[mysqld]
port = 3306
socket = /usr/local/mysql/var/mysql.sock

basedir = /usr/local/mysql/
datadir = /data/mysql/data
pid-file = /data/mysql/data/mysql.pid
user = mysql
bind-address = 0.0.0.0
server-id = 1
sync_binlog=1
log_bin = mysql-bin

skip-name-resolve
back_log = 600

max_connections = 3000
max_connect_errors = 3000
table_open_cache = 512
max_allowed_packet = 16M
binlog_cache_size = 16M
max_heap_table_size = 16M
tmp_table_size = 256M

read_buffer_size = 1024M
read_rnd_buffer_size = 1024M
sort_buffer_size = 1024M
join_buffer_size = 1024M
```

```
key_buffer_size = 8192M

thread_cache_size = 8

query_cache_size = 512M
query_cache_limit = 1024M

ft_min_word_len = 4

binlog_format = mixed
expire_logs_days = 30

log_error = /data/mysql/data/mysql-error.log
slow_query_log = 1
long_query_time = 1
slow_query_log_file = /data/mysql/data/mysql-slow.log

performance_schema = 0
explicit_defaults_for_timestamp

skip-external-locking


default_storage_engine = InnoDB
innodb_file_per_table = 1
innodb_open_files = 500
innodb_buffer_pool_size = 1024M
innodb_write_io_threads = 1000
innodb_read_io_threads = 1000
innodb_thread_concurrency = 8
innodb_purge_threads = 1
innodb_flush_log_at_trx_commit = 2
innodb_log_buffer_size = 4M
innodb_log_file_size = 32M
innodb_log_files_in_group = 3
innodb_max_dirty_pages_pct = 90
innodb_lock_wait_timeout = 120

bulk_insert_buffer_size = 8M
myisam_sort_buffer_size = 8M
myisam_max_sort_file_size = 10G
myisam_repair_threads = 1

interactive_timeout = 28800
wait_timeout = 28800

[mysqldump]
quick
max_allowed_packet = 16M

[myisamchk]
key_buffer_size = 8M
sort_buffer_size = 8M
read_buffer = 4M
write_buffer = 4M

sql_mode=NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES
port = 3306
```

下面分享一个mysql5.6下my.cnf的优化配置，能使mysql性能大大提升：

```
# For advice on how to change settings please see
# http://dev.mysql.com/doc/refman/5.6/en/server-configuration-defaults.html
# * DO NOT EDIT THIS FILE. It's a template which will be copied to the
# * default location during install, and will be replaced if you
# *** upgrade to a newer version of MySQL.
[mysqld]
# Remove leading # and set to the amount of RAM for the most important data
# cache in MySQL. Start at 70% of total RAM for dedicated server, else 10%.
# innodb_buffer_pool_size = 128M
# Remove leading # to turn on a very important data integrity option: logging
# changes to the binary log between backups.
# log_bin
# These are commonly set, remove the # and set as required.
# basedir = .....
# datadir = .....
# port = .....
# server_id = .....
# socket = .....

# Remove leading # to set options mainly useful for reporting servers.
# The server defaults are faster for transactions and fast SELECTs.
# Adjust sizes as needed, experiment to find the optimal values.
##################################################
#innodb
user=mysql
innodb_buffer_pool_size=6G
innodb_log_file_size=4G
innodb_log_buffer_size = 8M
innodb_flush_log_at_trx_commit=2
innodb_file_per_table=1
innodb_file_io_threads=4
innodb_flush_method=O_DIRECT
innodb_io_capacity=2000
innodb_io_capacity_max=6000
innodb_lru_scan_depth=2000
innodb_thread_concurrency = 0
innodb_additional_mem_pool_size=16M
innodb_autoinc_lock_mode = 2
##################################################
# Binary log/replication
log-bin
sync_binlog=1
sync_relay_log=1
relay-log-info-repository=TABLE
master-info-repository=TABLE
expire_logs_days=7
binlog_format=ROW
transaction-isolation=READ-COMMITTED
##################################################
#cache
```

```
tmp_table_size=512M
character-set-server=utf8
collation-server=utf8_general_ci
skip-external-locking
back_log=1024
key_buffer_size=1024M
thread_stack=256k
read_buffer_size=8M
thread_cache_size=64
query_cache_size=128M
max_heap_table_size=256M
query_cache_type=1
binlog_cache_size = 2M
table_open_cache=128
thread_cache=1024
thread_concurrency=8
wait_timeout=30
join_buffer_size = 1024M
sort_buffer_size = 8M
read_rnd_buffer_size = 8M
#####################################################
#connect
max-connect-errors=100000
max-connections=1000
#####################################################
explicit_defaults_for_timestamp=true
sql_mode=NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES
#####################################################
```

参数解释：

```
# Binary log/replication(这里主要是复制功能，也就是主从,提前配置好，后面讲主从配置)

#二进制日志
log-bin
#为了在最大程序上保证复制的InnoDB事务持久性和一致性
sync_binlog=1
sync_relay_log=1
#启用此两项,可用于实现在崩溃时保证二进制及从服务器安全的功能
relay-log-info-repository=TABLE
master-info-repository=TABLE
#设置清除日志时间
expire_logs_days=7
#行复制
binlog_format=ROW
#mysql数据库事务隔离级别有四种(READ UNCOMMITTED,READ COMMITTED,REPEATABLE
READ,SERIALIZABLE)
transaction-isolation=READ-COMMITTED

#cache
#内部内存临时表的最大值
tmp_table_size=512M
character-set-server=utf8
collation-server=utf8_general_ci
```

```
#即跳过外部锁定
skip-external-locking
#MySQL能暂存的连接数量(根据实际设置)
back_log=1024
#指定索引缓冲区的大小,只对MyISAM表起作用,这里写上也没有关系
key_buffer_size=1024M
#这条指令限定用于每个数据库线程的栈大小
thread_stack=256k
#当一个查询不断地扫描某一个表,MySQL会为它分配一段内存缓冲区
read_buffer_size=8M
#线程缓存
thread_cache_size=64
#查询缓存大小
query_cache_size=128M
#内部内存临时表的最大值,每个线程都要分配
max_heap_table_size=256M
#将查询结果放入查询缓存中
query_cache_type=1
#代表在事务过程中容纳二进制日志SQL语句的缓存大小
binlog_cache_size = 2M
#同样是缓存表大小
table_open_cache=128
#缓存线程
thread_cache=1024
#推荐设置为服务器 CPU核数的2倍
thread_concurrency=8
wait_timeout=30
#表和表联接的缓冲区的大小
join_buffer_size = 1024M
#是一个connection级参数,在每个connection第一次需要使用这个buffer的时候,一次性分配设置的内存
sort_buffer_size=8M
#随机读取数据缓冲区使用内存
read_rnd_buffer_size = 8M

#connect
#是一个MySQL中与安全有关的计数器值,它负责阻止过多尝试失败的客户端以防止暴力破解密码
max-connect-errors=100000
#连接数
max-connections=1000
#开启查询缓存
explicit_defaults_for_timestamp=true
#mysql服务器能够工作在不同的模式下,并能针对不同的客户端以不同的方式应用这些模式
sql_mode=NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES
```

下面列出了对性能优化影响较大的主要变量，主要分为连接请求的变量和缓冲区变量。
1.连接请求的变量：
1) max_connections
MySQL的最大连接数，增加该值增加mysqld 要求的文件描述符的数量。如果服务器的并发连接请求量比较大，建议调高此值，以增加并行连接数量，当然这建立在机器能支撑的情况下，因为如果连接数越多， 介于MySQL会为每个连接提供连接缓冲区，就会开销越多的内存，所以要适当调整该值，不能盲目提高设值。
数值过小会经常出现ERROR 1040: Too many connections错误，可以过’conn%’通配符查看当前状态的连接数量，以定夺该值的大小。

show variables like ‘max_connections’ 最大连接数
show status like ‘max_used_connections’响应的连接数
如下：
mysql> show variables like ‘max_connections‘;
+——————————+——-+
| Variable_name | Value |
+——————————+——-+
| max_connections | 256 |
+——————————+——-+
mysql> show status like ‘max%connections‘;
+——————————+——-+
| Variable_name | Value |
+——————————+——-+
| max_used_connections | 256|
+——————————+——-+
max_used_connections / max_connections * 100% （理想值≈ 85%）
如果max_used_connections跟max_connections相同 那么就是max_connections设置过低或者超过服务器负载上限了，低于10%则设置过大。
2) back_log
MySQL能暂存的连接数量。当主要MySQL线程在一个很短时间内得到非常多的连接请求，这就起作用。如果MySQL的连接数据达到 max_connections时，新来的请求将会被存在堆栈中，以等待某一连接释放资源，该堆栈的数量即back_log，如果等待连接的数量超过 back_log，将不被授予连接资源。
back_log值指出在MySQL暂时停止回答新请求之前的短时间内有多少个请求可以被存在堆栈中。只有如果期望在一个短时间内有很多连接，你需要增加它，换句话说，这值对到来的TCP/IP连接的侦听队列的大小。
当观察你主机进程列表（mysql> show full processlist），发现大量264084 | unauthenticated user | xxx.xxx.xxx.xxx | NULL | Connect | NULL | login | NULL 的待连接进程时，就要加大back_log 的值了。
默认数值是50，可调优为128，对系统设置范围为小于512的整数。
3) interactive_timeout
一个交互连接在被服务器在关闭前等待行动的秒数。一个交互的客户被定义为对mysql_real_connect()使用CLIENT_INTERACTIVE 选项的客户。
默认数值是28800，可调优为7200。
\2. 缓冲区变量
全局缓冲：
4) key_buffer_size
key_buffer_size指定索引缓冲区的大小，它决定索引处理的速度，尤其是索引读的速度。通过检查状态值 Key_read_requests和Key_reads，可以知道key_buffer_size设置是否合理。比例key_reads / key_read_requests应该尽可能的低，至少是1:100，1:1000更好（上述状态值可以使用SHOW STATUS LIKE ‘key_read%’获得）。
key_buffer_size只对MyISAM表起作用。即使你不使用MyISAM表，但是内部的临时磁盘表是MyISAM表，也要使用该值。可以使用检查状态值created_tmp_disk_tables得知详情。
举例如下：
mysql> show variables like ‘key_buffer_size‘;
+——————-+————+
| Variable_name | Value |
+——————+————+
| key_buffer_size | 536870912 |
+———— ————-+————+
key_buffer_size为512MB，我们再看一下key_buffer_size的使用情况：
mysql> show global status like ‘key_read%‘;
+——————————+————-+

| Variable_name | Value |
+——————————+———————+
| Key_read_requests| 27813678764 |
| Key_reads | 6798830 |
+——————————+———————+
一共有27813678764个索引读取请求，有6798830个请求在内存中没有找到直接从硬盘读取索引，计算索引未命中缓存的概率：
key_cache_miss_rate = Key_reads / Key_read_requests * 100%，设置在1/1000左右较好
默认配置数值是8388600(8M)，主机有4GB内存，可以调优值为268435456(256MB)。
5) query_cache_size
使用查询缓冲，MySQL将查询结果存放在缓冲区中，今后对于同样的SELECT语句（区分大小写），将直接从缓冲区中读取结果。
通过检查状态值Qcache_*，可以知道query_cache_size设置是否合理（上述状态值可以使用SHOW STATUS LIKE ‘Qcache%’获得）。如果Qcache_lowmem_prunes的值非常大，则表明经常出现缓冲不够的情况，如果Qcache_hits的值也 非常大，则表明查询缓冲使用非常频繁，此时需要增加缓冲大小；如果Qcache_hits的值不大，则表明你的查询重复率很低，这种情况下使用查询缓冲反 而会影响效率，那么可以考虑不用查询缓冲。此外，在SELECT语句中加入SQL_NO_CACHE可以明确表示不使用查询缓冲。

与查询缓冲有关的参数还有query_cache_type、query_cache_limit、query_cache_min_res_unit。
query_cache_type指定是否使用查询缓冲，可以设置为0、1、2，该变量是SESSION级的变量。
query_cache_limit指定单个查询能够使用的缓冲区大小，缺省为1M。
query_cache_min_res_unit是在4.1版本以后引入的，它指定分配缓冲区空间的最小单位，缺省为4K。
检查状态值 Qcache_free_blocks，如果该值非常大，则表明缓冲区中碎片很多，这就表明查询结果都比较小，此时需要减小 query_cache_min_res_unit。
举例如下：
mysql> show global status like ‘qcache%’;
+———————————-+———————–+
| Variable_name | Value |
+———————————-+———————–+
Qcache_free_blocks	22756
Qcache_free_memory	76764704
Qcache_hits	213028692
Qcache_inserts	208894227
Qcache_lowmem_prunes	4010916
Qcache_not_cached	13385031
Qcache_queries_in_cache	43560
Qcache_total_blocks	111212
+———————————-+———————–+
mysql> show variables like ‘query_cache%’;
+——————————————–+———————–+
| Variable_name | Value |
+——————————————–+———————–+
query_cache_limit	2097152
query_cache_min_res_unit	4096
query_cache_size	203423744
query_cache_type	ON
query_cache_wlock_invalidate	OFF
+——————————————–+———————+
查询缓存碎片率= Qcache_free_blocks / Qcache_total_blocks * 100%
如果查询缓存碎片率超过20%，可以用FLUSH QUERY CACHE整理缓存碎片，或者试试减小query_cache_min_res_unit，如果你的查询都是小数据量的话。

查询缓存利用率= (query_cache_size – Qcache_free_memory) / query_cache_size * 100%
查询缓存利用率在25%以下的话说明query_cache_size设置的过大，可适当减小；查询缓存利用率在80%以上而且Qcache_lowmem_prunes > 50的话说明query_cache_size可能有点小，要不就是碎片太多。
查询缓存命中率= (Qcache_hits – Qcache_inserts) / Qcache_hits * 100%
示例服务器查询缓存碎片率 = 20.46%，查询缓存利用率 = 62.26%，查询缓存命中率 = 1.94%，命中率很差，可能写操作比较频繁吧，而且可能有些碎片。
每个连接的缓冲
6) record_buffer_size
每个进行一个顺序扫描的线程为其扫描的每张表分配这个大小的一个缓冲区。如果你做很多顺序扫描，你可能想要增加该值。
默认数值是131072(128K)，可改为16773120 (16M)
7) read_rnd_buffer_size
随机读缓冲区大小。当按任意顺序读取行时(例如，按照排序顺序)，将分配一个随机读缓存区。进行排序查询时，MySQL会首先扫描一遍该缓冲，以避 免磁盘搜索，提高查询速度，如果需要排序大量数据，可适当调高该值。但MySQL会为每个客户连接发放该缓冲空间，所以应尽量适当设置该值，以避免内存开销过大。
一般可设置为16M
8) sort_buffer_size
每个需要进行排序的线程分配该大小的一个缓冲区。增加这值加速ORDER BY或GROUP BY操作。
默认数值是2097144(2M)，可改为16777208 (16M)。
9) join_buffer_size
联合查询操作所能使用的缓冲区大小
record_buffer_size，read_rnd_buffer_size，sort_buffer_size，join_buffer_size为每个线程独占，也就是说，如果有100个线程连接，则占用为16M*100
10) table_cache
表高速缓存的大小。每当MySQL访问一个表时，如果在表缓冲区中还有空间，该表就被打开并放入其中，这样可以更快地访问表内容。通过检查峰值时间的状态值Open_tables和Opened_tables，可以决定是否需要增加table_cache的值。如 果你发现open_tables等于table_cache，并且opened_tables在不断增长，那么你就需要增加table_cache的值了 （上述状态值可以使用SHOW STATUS LIKE ‘Open%tables’获得）。注意，不能盲目地把table_cache设置成很大的值。如果设置得太高，可能会造成文件描述符不足，从而造成性能 不稳定或者连接失败。
1G内存机器，推荐值是128 - 256。内存在4GB左右的服务器该参数可设置为256M或384M。
11) max_heap_table_size
用户可以创建的内存表(memory table)的大小。这个值用来计算内存表的最大行数值。这个变量支持动态改变，即set @max_heap_table_size=#
这个变量和tmp_table_size一起限制了内部内存表的大小。如果某个内部heap（堆积）表大小超过tmp_table_size，MySQL可以根据需要自动将内存中的heap表改为基于硬盘的MyISAM表。
12) tmp_table_size
通过设置tmp_table_size选项来增加一张临时表的大小，例如做高级GROUP BY操作生成的临时表。如果调高该值，MySQL同时将增加heap表的大小，可达到提高联接查询速度的效果，建议尽量优化查询，要确保查询过程中生成的临时表在内存中，避免临时表过大导致生成基于硬盘的MyISAM表。
mysql> show global status like ‘created_tmp%’;

Variable_name	Value
Created_tmp_disk_tables	21197
Created_tmp_files	58
Created_tmp_tables	1771587

每次创建临时表，Created_tmp_tables增加，如果临时表大小超过tmp_table_size，则是在磁盘上创建临时 表，Created_tmp_disk_tables也增加,Created_tmp_files表示MySQL服务创建的临时文件文件

数，比较理想的配 置是：
Created_tmp_disk_tables / Created_tmp_tables * 100% <= 25%比如上面的服务器
Created_tmp_disk_tables / Created_tmp_tables * 100% = 1.20%，应该相当好了
默认为16M，可调到64-256最佳，线程独占，太大可能内存不够I/O堵塞
13) thread_cache_size
可以复用的保存在中的线程的数量。如果有，新的线程从缓存中取得，当断开连接的时候如果有空间，客户的线置在缓存中。如果有很多新的线程，为了提高性能可以这个变量值。
通过比较 Connections和Threads_created状态的变量，可以看到这个变量的作用。
默认值为110，可调优为80。
14) thread_concurrency
推荐设置为服务器 CPU核数的2倍，例如双核的CPU, 那么thread_concurrency的应该为4；2个双核的cpu, thread_concurrency的值应为8。默认为8
15) wait_timeout
指定一个请求的最大连接时间，对于4GB左右内存的服务器可以设置为5-10。
\3. 配置InnoDB的几个变量
innodb_buffer_pool_size
对于InnoDB表来说，innodb_buffer_pool_size的作用就相当于key_buffer_size对于MyISAM表的作用一样。InnoDB使用该参数指定大小的内存来缓冲数据和索引。对于单独的MySQL数据库服务器，最大可以把该值设置成物理内存的80%。
根据MySQL手册，对于2G内存的机器，推荐值是1G（50%）。

innodb_flush_log_at_trx_commit
主要控制了innodb将log buffer中的数据写入日志文件并flush磁盘的时间点，取值分别为0、1、2三个。0，表示当事务提交时，不做日志写入操作，而是每秒钟将log buffer中的数据写入日志文件并flush磁盘一次；1，则在每秒钟或是每次事物的提交都会引起日志文件写入、flush磁盘的操作，确保了事务的 ACID；设置为2，每次事务提交引起写入日志文件的动作，但每秒钟完成一次flush磁盘操作。
实际测试发现，该值对插入数据的速度影响非常大，设置为2时插入10000条记录只需要2秒，设置为0时只需要1秒，而设置为1时则需要229秒。因此，MySQL手册也建议尽量将插入操作合并成一个事务，这样可以大幅提高速度。
根据MySQL手册，在允许丢失最近部分事务的危险的前提下，可以把该值设为0或2。

innodb_log_buffer_size
log缓存大小，一般为1-8M，默认为1M，对于较大的事务，可以增大缓存大小。
可设置为4M或8M。

innodb_additional_mem_pool_size
该参数指定InnoDB用来存储数据字典和其他内部数据结构的内存池大小。缺省值是1M。通常不用太大，只要够用就行，应该与表结构的复杂度有关系。如果不够用，MySQL会在错误日志中写入一条警告信息。
根据MySQL手册，对于2G内存的机器，推荐值是20M，可适当增加。

innodb_thread_concurrency=8
推荐设置为 2*(NumCPUs+NumDisks)，默认一般为8

MySQL 5.6相比于前代GA版本性能提升显著，但默认缓存设置对于小型站点并不合理。通过修改my.ini文件中的performance_schema_max_table_instances参数，能够有效降低内存占用。
以下是5.6默认的设置
performance_schema_max_table_instances 12500
table_definition_cache 1400
table_open_cache 2000
可以调成，或者在小点都可以。

performance_schema_max_table_instances=600
table_definition_cache=400
table_open_cache=256

performance_schema_max_table_instances
The maximum number of instrumented table objects 检测的表对象的最大数目。
table_definition_cache
The number of table definitions (from .frm files) that can be stored in the definition cache. If you use a large number of tables, you can create a large table definition cache to speed up opening of tables. The table definition cache takes less space and does not use file descriptors, unlike the normal table cache. The minimum and default values are both 400.
缓存frm文件

table_open_cache
The number of open tables for all threads. Increasing this value increases the number of file descriptors that mysqld requires.
table_open_cache 指的是缓存数据文件的描述符(Linux/Unix)相关信息
这个很重要啊，之前mount个单独的文件，数据库一直不成功，原来是这个在作怪啊。
chcon -R -t mysqld_db_t /home/myusqldata

mysql> show variables;
一、慢查询
mysql> show variables like '%slow%';
+------------------+-------+
| Variable_name | Value |
+------------------+-------+
| log_slow_queries | ON |
| slow_launch_time | 2 |
+------------------+-------+

mysql> show global status like '%slow%';
+---------------------+-------+
| Variable_name | Value |
+---------------------+-------+
| Slow_launch_threads | 0 |
| Slow_queries | 4148 |
+---------------------+-------+
配置中打开了记录慢查询，执行时间超过2秒的即为慢查询，系统显示有4148个慢查询，你可以分析慢查询日志，找出有问题的SQL语句，慢查询时间不宜设置过长，否则意义不大，最好在5秒以内，如果你需要微秒级别的慢查询，可以考虑给MySQL打补丁，记得找对应的版本。
打开慢查询日志可能会对系统性能有一点点影响，如果你的MySQL是主 - 从结构，可以考虑打开其中一台从服务器的慢查询日志，这样既可以监控慢查询，对系统性能影响又小。
二、连接数
经常会遇见”MySQL: ERROR 1040: Too manyconnections”的情况，一种是访问量确实很高，MySQL服务器抗不住，这个时候就要考虑增加从服务器分散读压力，另外一种情况是MySQL配置文件中max_connections值过小：
mysql> show variables like 'max_connections';
+-----------------+-------+
| Variable_name | Value |
+-----------------+-------+
| max_connections | 256 |
+-----------------+-------+
这台MySQL服务器最大连接数是256，然后查询一下服务器响应的最大连接数：
mysql> show global status like 'Max_used_connections';
+----------------------+-------+
| Variable_name | Value |
+----------------------+-------+

| Max_used_connections | 245 |
+----------------------+-------+
MySQL服务器过去的最大连接数是245，没有达到服务器连接数上限256，应该没有出现1040错误，比较理想的设置是：
Max_used_connections / max_connections * 100% ≈ 85%
最大连接数占上限连接数的85%左右，如果发现比例在10%以下，MySQL服务器连接数上限设置的过高了。
三、Key_buffer_size
key_buffer_size是对MyISAM表性能影响最大的一个参数，下面一台以MyISAM为主要存储引擎服务器的配置：
mysql> show variables like 'key_buffer_size';
+-----------------+------------+
| Variable_name | Value |
+-----------------+------------+
| key_buffer_size | 536870912 |
+-----------------+------------+
分配了512MB内存给key_buffer_size，我们再看一下key_buffer_size的使用情况：
mysql> show global status like 'key_read%';
+-----------------------+-------------+
| Variable_name | Value |
+-----------------------+-------------+
| Key_read_requests | 27813678764 |
| Key_reads | 6798830 |
+-----------------------+-------------+
一共有27813678764个索引读取请求，有6798830个请求在内存中没有找到直接从硬盘读取索引，计算索引未命中缓存的概率：
key_cache_miss_rate = Key_reads / Key_read_requests * 100%
比 如上面的数据，key_cache_miss_rate为0.0244%，4000个索引读取请求才有一个直接读硬盘，已经很BT 了，key_cache_miss_rate在0.1%以下都很好（每1000个请求有一个直接读硬盘），如果key_cache_miss_rate在 0.01%以下的话，key_buffer_size分配的过多，可以适当减少。
MySQL服务器还提供了key_blocks_*参数：
mysql> show global status like 'key_blocks_u%';
+-----------------------+-------------+
| Variable_name | Value |
+-----------------------+-------------+
| Key_blocks_unused | 0 |
| Key_blocks_used | 413543 |
+-----------------------+-------------+
Key_blocks_unused 表示未使用的缓存簇(blocks)数，Key_blocks_used表示曾经用到的最大的blocks数，比如这台服务器，所有的缓存都用到了，要么 增加key_buffer_size，要么就是过渡索引了，把缓存占满了。比较理想的设置：
Key_blocks_used / (Key_blocks_unused + Key_blocks_used) * 100% ≈ 80%
四、临时表
mysql> show global status like 'created_tmp%';
+-------------------------+---------+
| Variable_name | Value |
+-------------------------+---------+
Created_tmp_disk_tables	21197
Created_tmp_files	58
Created_tmp_tables	1771587
+-------------------------+---------+
每次创建临时表，Created_tmp_tables增加，如果是在磁盘上创建临时表，Created_tmp_disk_tables

也增加,Created_tmp_files表示MySQL服务创建的临时文件文件数，比较理想的配置是：
Created_tmp_disk_tables / Created_tmp_tables * 100% <= 25%
比如上面的服务器Created_tmp_disk_tables / Created_tmp_tables * 100% = 1.20%，应该相当好了。我们再看一下MySQL服务器对临时表的配置：
mysql> show variables where Variable_name in ('tmp_table_size', 'max_heap_table_size');
+---------------------+-----------+
| Variable_name | Value |
+---------------------+-----------+
| max_heap_table_size | 268435456 |
| tmp_table_size | 536870912 |
+---------------------+-----------+
只有256MB以下的临时表才能全部放内存，超过的就会用到硬盘临时表。
五、Open Table情况
mysql> show global status like 'open%tables%';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables | 919 |
| Opened_tables | 1951 |
+---------------+-------+
Open_tables 表示打开表的数量，Opened_tables表示打开过的表数量，如果Opened_tables数量过大，说明配置中 table_cache(5.1.3之后这个值叫做table_open_cache)值可能太小，我们查询一下服务器table_cache值：
mysql> show variables like 'table_cache';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| table_cache | 2048 |
+---------------+-------+
比较合适的值为：
Open_tables / Opened_tables * 100% >= 85%
Open_tables / table_cache * 100% <= 95%

六、进程使用情况
mysql> show global status like 'Thread%';
+-------------------+-------+
| Variable_name | Value |
+-------------------+-------+
Threads_cached	46
Threads_connected	2
Threads_created	570
Threads_running	1
+-------------------+-------+
如 果我们在MySQL服务器配置文件中设置了thread_cache_size，当客户端断开之后，服务器处理此客户的线程将会缓存起来以响应下一个客户 而不是销毁（前提是缓存数未达上限）。Threads_created表示创建过的线程数，如果发现Threads_created值过大的话，表明 MySQL服务器一直在创建线程，这也是比较耗资源，可以适当增加配置文件中thread_cache_size值，查询服务器 thread_cache_size配置：
mysql> show variables like 'thread_cache_size';
+-------------------+-------+
| Variable_name | Value |
+-------------------+-------+
| thread_cache_size | 64 |

+-------------------+-------+
示例中的服务器还是挺健康的。
七、查询缓存(query cache)
mysql> show global status like 'qcache%';
+-------------------------+-----------+
| Variable_name | Value |
+-------------------------+-----------+
Qcache_free_blocks	22756
Qcache_free_memory	76764704
Qcache_hits	213028692
Qcache_inserts	208894227
Qcache_lowmem_prunes	4010916
Qcache_not_cached	13385031
Qcache_queries_in_cache	43560
Qcache_total_blocks	111212
+-------------------------+-----------+
MySQL查询缓存变量解释：
Qcache_free_blocks：缓存中相邻内存块的个数。数目大说明可能有碎片。FLUSH QUERY CACHE会对缓存中的碎片进行整理，从而得到一个空闲块。
Qcache_free_memory：缓存中的空闲内存。
Qcache_hits：每次查询在缓存中命中时就增大
Qcache_inserts：每次插入一个查询时就增大。命中次数除以插入次数就是不中比率。
Qcache_lowmem_prunes： 缓存出现内存不足并且必须要进行清理以便为更多查询提供空间的次数。这个数字最好长时间来看；如果这个数字在不断增长，就表示可能碎片非常严重，或者内存 很少。（上面的 free_blocks和free_memory可以告诉您属于哪种情况）
Qcache_not_cached：不适合进行缓存的查询的数量，通常是由于这些查询不是 SELECT 语句或者用了now()之类的函数。
Qcache_queries_in_cache：当前缓存的查询（和响应）的数量。
Qcache_total_blocks：缓存中块的数量。
我们再查询一下服务器关于query_cache的配置：
mysql> show variables like 'query_cache%';
+------------------------------+-----------+
| Variable_name | Value |
+------------------------------+-----------+
query_cache_limit	2097152
query_cache_min_res_unit	4096
query_cache_size	203423744
query_cache_type	ON
query_cache_wlock_invalidate	OFF
+------------------------------+-----------+
各字段的解释：
query_cache_limit：超过此大小的查询将不缓存
query_cache_min_res_unit：缓存块的最小大小
query_cache_size：查询缓存大小
query_cache_type：缓存类型，决定缓存什么样的查询，示例中表示不缓存 select sql_no_cache 查询
query_cache_wlock_invalidate：当有其他客户端正在对MyISAM表进行写操作时，如果查询在query cache中，是否返回cache结果还是等写操作完成再读表获取结果。
query_cache_min_res_unit的配置是一柄”双刃剑”，默认是4KB，设置值大对大数据查询有好处，但如果你的查询都是小数据查询，就容易造成内存碎片和浪费。
查询缓存碎片率 = Qcache_free_blocks / Qcache_total_blocks * 100%
如果查询缓存碎片率超过20%，可以用FLUSH QUERY CACHE整理缓存碎片，或者试试减小query_cache_min_res_unit，如果你的查询都是小数据量的话。

查询缓存利用率 = (query_cache_size - Qcache_free_memory) / query_cache_size * 100%
查询缓存利用率在25%以下的话说明query_cache_size设置的过大，可适当减小；查询缓存利用率在80%以上而且Qcache_lowmem_prunes > 50的话说明query_cache_size可能有点小，要不就是碎片太多。
查询缓存命中率 = (Qcache_hits - Qcache_inserts) / Qcache_hits * 100%
示例服务器 查询缓存碎片率 = 20.46%，查询缓存利用率 = 62.26%，查询缓存命中率 = 1.94%，命中率很差，可能写操作比较频繁吧，而且可能有些碎片。
八、排序使用情况

```
mysql> show global status like 'sort%';
+-------------------+------------+
| Variable_name | Value |
+-------------------+------------+
| Sort_merge_passes | 29 |
| Sort_range | 37432840 |
| Sort_rows | 9178691532 |
| Sort_scan | 1860569 |
+-------------------+------------+
```

Sort_merge_passes 包括两步。MySQL 首先会尝试在内存中做排序，使用的内存大小由系统变量 Sort_buffer_size 决定，如果它的大小不够把所有的记录都读到内存中，MySQL 就会把每次在内存中排序的结果存到临时文件中，等MySQL 找到所有记录之后，再把临时文件中的记录做一次排序。这再次排序就会增加 Sort_merge_passes。实际上，MySQL会用另一个临时文件来存再次排序的结果，所以通常会看到 Sort_merge_passes增加的数值是建临时文件数的两倍。因为用到了临时文件，所以速度可能会比较慢，增加 Sort_buffer_size 会减少Sort_merge_passes 和 创建临时文件的次数。但盲目的增加 Sort_buffer_size 并不一定能提高速度，
另外，增加read_rnd_buffer_size(3.2.3是record_rnd_buffer_size)的值对排序的操作也有一点的好处，
九、文件打开数(open_files)

```
mysql> show global status like 'open_files';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_files | 1410 |
+---------------+-------+

mysql> show variables like 'open_files_limit';
+------------------+-------+
| Variable_name | Value |
+------------------+-------+
| open_files_limit | 4590 |
+------------------+-------+
```

比较合适的设置：Open_files / open_files_limit * 100% <= 75%
十、表锁情况

```
mysql> show global status like 'table_locks%';
+----------------------+-----------+
| Variable_name | Value |
+----------------------+-----------+
| Table_locks_immediate | 490206328 |
| Table_locks_waited | 2084912 |
+----------------------+-----------+
```

Table_locks_immediate 表示立即释放表锁数，Table_locks_waited表示需要等待的表锁数，如果 Table_locks_immediate / Table_locks_waited >5000，最好采用InnoDB引擎，因为InnoDB是行锁而MyISAM是表锁，对于高并发写入的应用InnoDB效果会好些。示例中的服务 器Table_locks_immediate / Table_locks_waited = 235，MyISAM就足够了。

十一、表扫描情况
mysql> show global status like 'handler_read%';
+-----------------------+-------------+
| Variable_name | Value |
+-----------------------+-------------+
Handler_read_first	5803750
Handler_read_key	6049319850
Handler_read_next	94440908210
Handler_read_prev	34822001724
Handler_read_rnd	405482605
Handler_read_rnd_next	18912877839
+-----------------------+-------------+
各字段解释参见,调出服务器完成的查询请求次数:
mysql> show global status like 'com_select';
+---------------+-----------+
| Variable_name | Value |
+---------------+-----------+
| Com_select | 222693559 |
+---------------+-----------+
计算表扫描率:
表扫描率 = Handler_read_rnd_next / Com_select
如果表扫描率超过4000，说明进行了太多表扫描，很有可能索引没有建好，增加read_buffer_size值会有一些好处，但最好不要超过8MB。

要查看死锁，你要show engine innodb status\G;
在MySQL5.6版本，在my.cnf配置文件里，加入
innodb_print_all_deadlocks = 1
就可以把死锁信息打印到错误日志里

Mysql怎么分表，以及分表后如果想按条件分页查询怎么办(如果不是按分表字段来查询的话，几乎效率低下，无解)

如果按时间排序查询，使用limit n （不要使用limit m, n 页数多了之后效率低）然后记录最后一条的时间，下次从最后一条的时间开始查询

mysql 数据库 分表后 怎么进行分页查询？Mysql分库分表方案?

1.如果只是为了分页，可以考虑这种分表，就是表的id是范围性的，且id是连续的，比如第一张表id是1到10万，第二张是10万到20万，这样分页应该没什么问题。

2.如果是其他的分表方式，建议用sphinx先建索引，然后查询分页，我们公司现在就是这样干的

Mysql分库分表方案

1.为什么要分表：

当一张表的数据达到几千万时，你查询一次所花的时间会变多，如果有联合查询的话，我想有可能会死在那儿了。分表的目的就在于此，减小数据库的负担，缩短查询时间。

mysql中有一种机制是表锁定和行锁定，是为了保证数据的完整性。表锁定表示你们都不能对这张表进行操作，必须等我对表操作完才行。行锁定也一样，别的sql必须等我对这条数据操作完了，才能对这条数据进行操作。

2. mysql proxy：amoeba

做mysql集群,利用amoeba。

从上层的java程序来讲，不需要知道主服务器和从服务器的来源，即主从数据库服务器对于上层来讲是透明的。可以通过amoeba来配置。

3.大数据量并且访问频繁的表，将其分为若干个表

比如对于某网站平台的数据库表-公司表，数据量很大，这种能预估出来的大数据量表，我们就事先分出个N个表，这个N是多少，根据实际情况而定。

某网站现在的数据量至多是5000万条，可以设计每张表容纳的数据量是500万条，也就是拆分成10张表，

那么如何判断某张表的数据是否容量已满呢？可以在程序段对于要新增数据的表，在插入前先做统计表记录数量的操作，当<500万条数据，就直接插入，当已经到达阀值，可以在程序段新创建数据库表（或者已经事先创建好），再执行插入操作。

4. 利用merge存储引擎来实现分表

如果要把已有的大数据量表分开比较痛苦，最痛苦的事就是改代码，因为程序里面的sql语句已经写好了。用merge存储引擎来实现分表, 这种方法比较适合.

举例子：

```
mysql> CREATE TABLE IF NOT EXISTS `user1` (
 ->   `id` int(11) NOT NULL AUTO_INCREMENT,
 ->   `name` varchar(50) DEFAULT NULL,
 ->   `sex` int(1) NOT NULL DEFAULT '0',
 ->   PRIMARY KEY (`id`)
 -> ) ENGINE=MyISAM  DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
Query OK, 0 rows affected (0.05 sec)

mysql> CREATE TABLE IF NOT EXISTS `user2` (
 ->   `id` int(11) NOT NULL AUTO_INCREMENT,
 ->   `name` varchar(50) DEFAULT NULL,
 ->   `sex` int(1) NOT NULL DEFAULT '0',
 ->   PRIMARY KEY (`id`)
 -> ) ENGINE=MyISAM  DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO `user1` (`name`, `sex`) VALUES('张映', 0);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO `user2` (`name`, `sex`) VALUES('tank', 1);
Query OK, 1 row affected (0.00 sec)

mysql> CREATE TABLE IF NOT EXISTS `alluser` (
 ->   `id` int(11) NOT NULL AUTO_INCREMENT,
 ->   `name` varchar(50) DEFAULT NULL,
 ->   `sex` int(1) NOT NULL DEFAULT '0',
 ->   INDEX(id)
 -> ) TYPE=MERGE UNION=(user1,user2) INSERT_METHOD=LAST AUTO_INCREMENT=1 ;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> select id,name,sex from alluser;
+----+--------+-----+
| id | name   | sex |
+----+--------+-----+
|  1 | 张映 |   0 |
|  1 | tank   |   1 |
+----+--------+-----+
2 rows in set (0.00 sec)

mysql> INSERT INTO `alluser` (`name`, `sex`) VALUES('tank2', 0);
Query OK, 1 row affected (0.00 sec)

mysql> select id,name,sex from user2
 -> ;
+----+-------+-----+
| id | name  | sex |
+----+-------+-----+
|  1 | tank  |   1 |
|  2 | tank2 |   0 |
+----+-------+-----+
2 rows in set (0.00 sec)
```

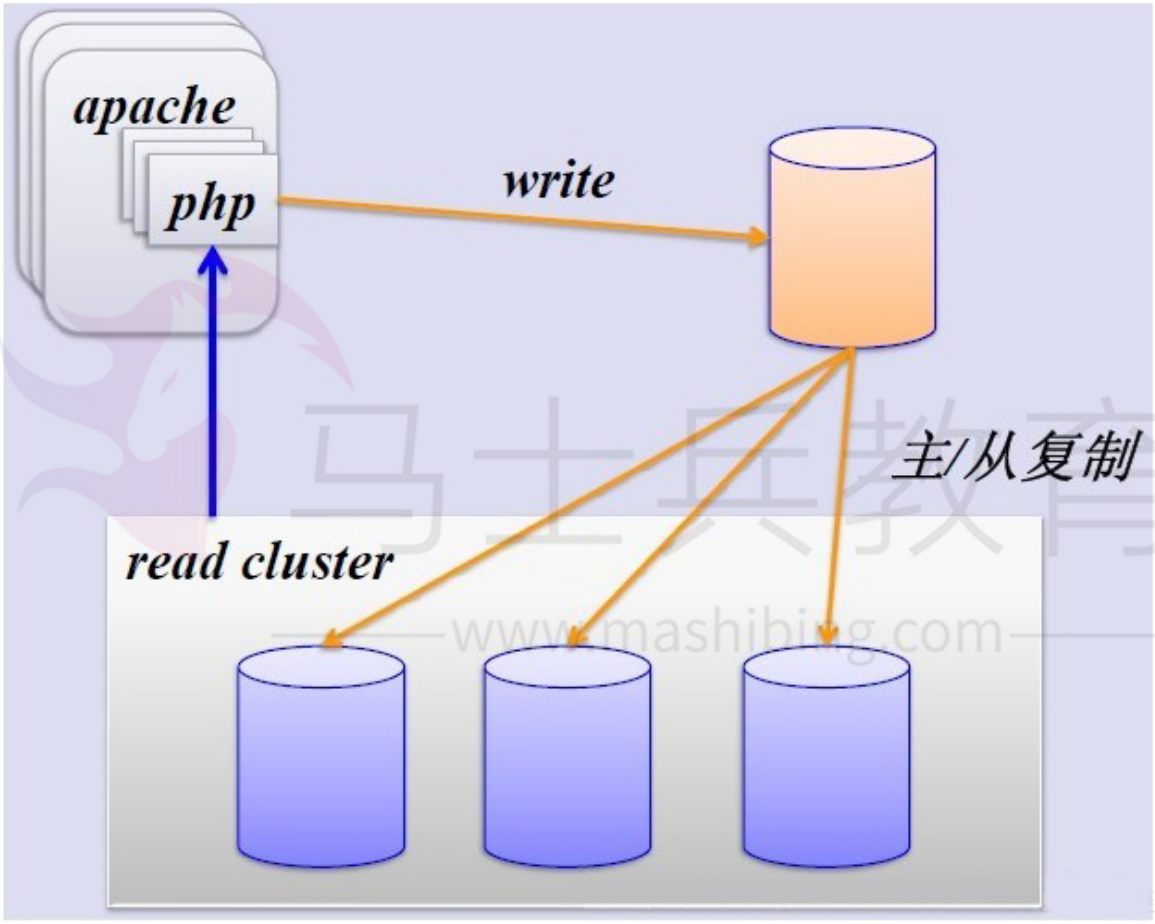
马士兵教育 www.mashibing.com

---------- ----------华丽的分割线----------

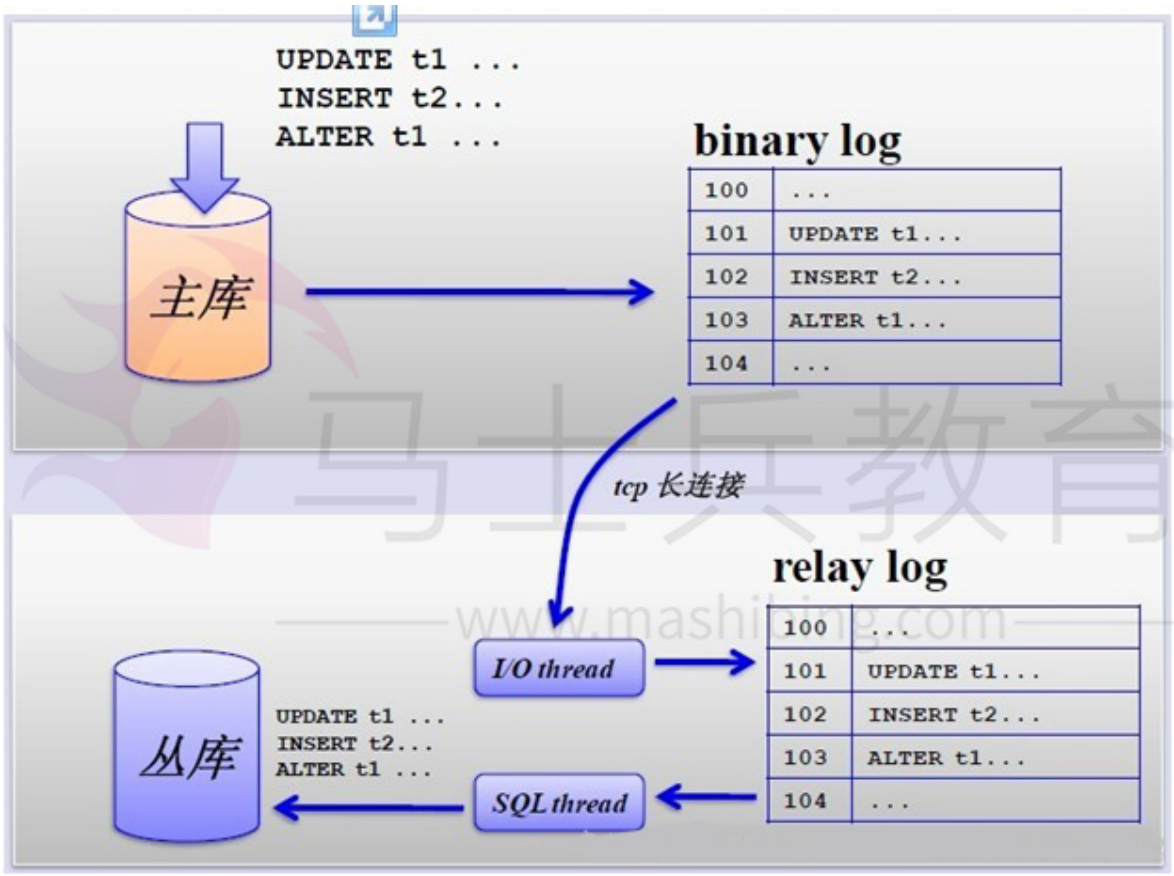
数据库架构

1、简单的MySQL主从复制:

MySQL的主从复制解决了数据库的读写分离，并很好的提升了读的性能，其图如下:



其主从复制的过程如下图所示：



但是，主从复制也带来其他一系列性能瓶颈问题：

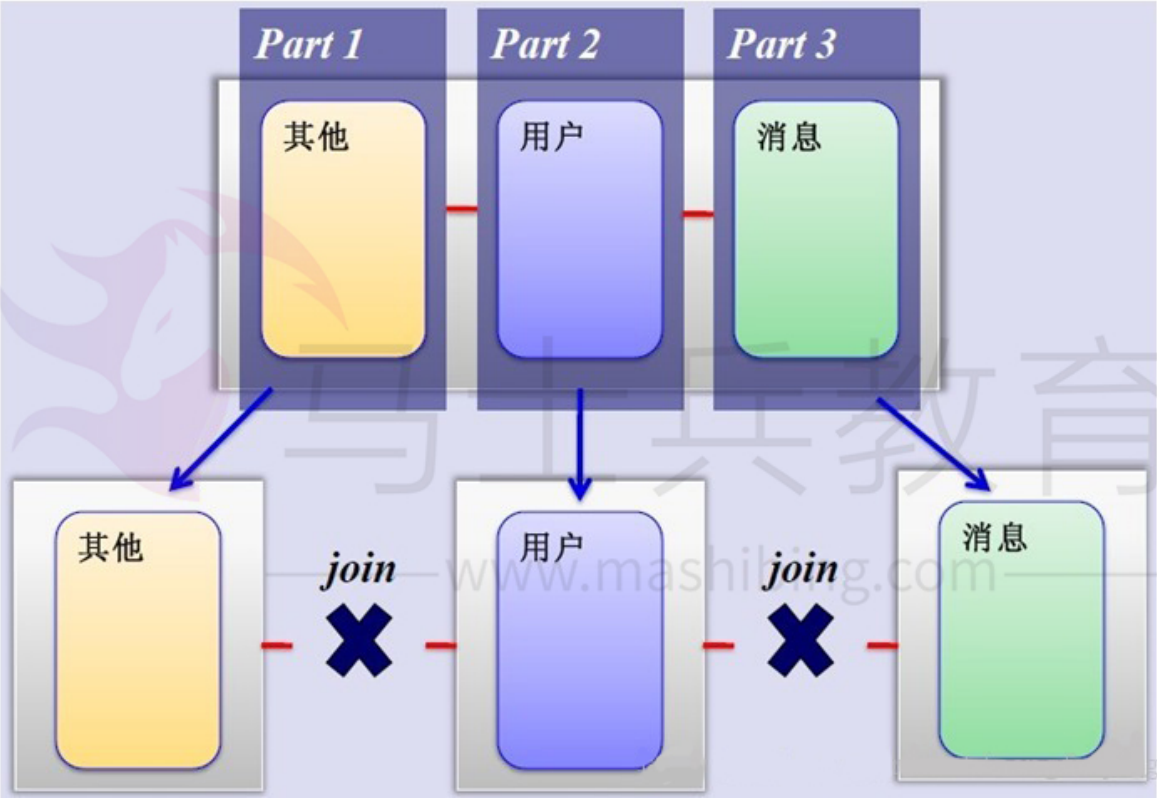
1. 写入无法扩展
2. 写入无法缓存

3. 复制延时
4. 锁表率上升
5. 表变大，缓存率下降

那问题产生总得解决的，这就产生下面的优化方案，一起来看看。

2、MySQL垂直分区

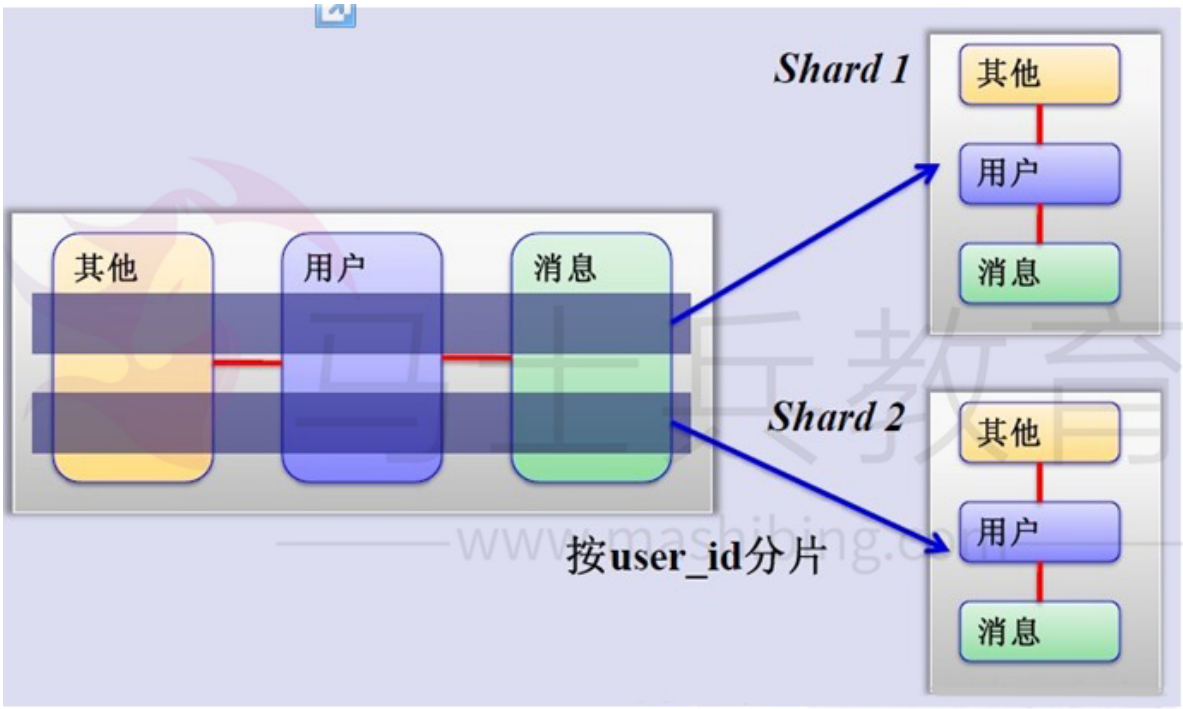
如果把业务切割得足够独立，那把不同业务的数据放到不同的数据库服务器将是一个不错的方案，而且万一其中一个业务崩溃了也不会影响其他业务的正常进行，并且也起到了负载分流的作用，大大提升了数据库的吞吐能力。经过垂直分区后的数据库架构图如下：



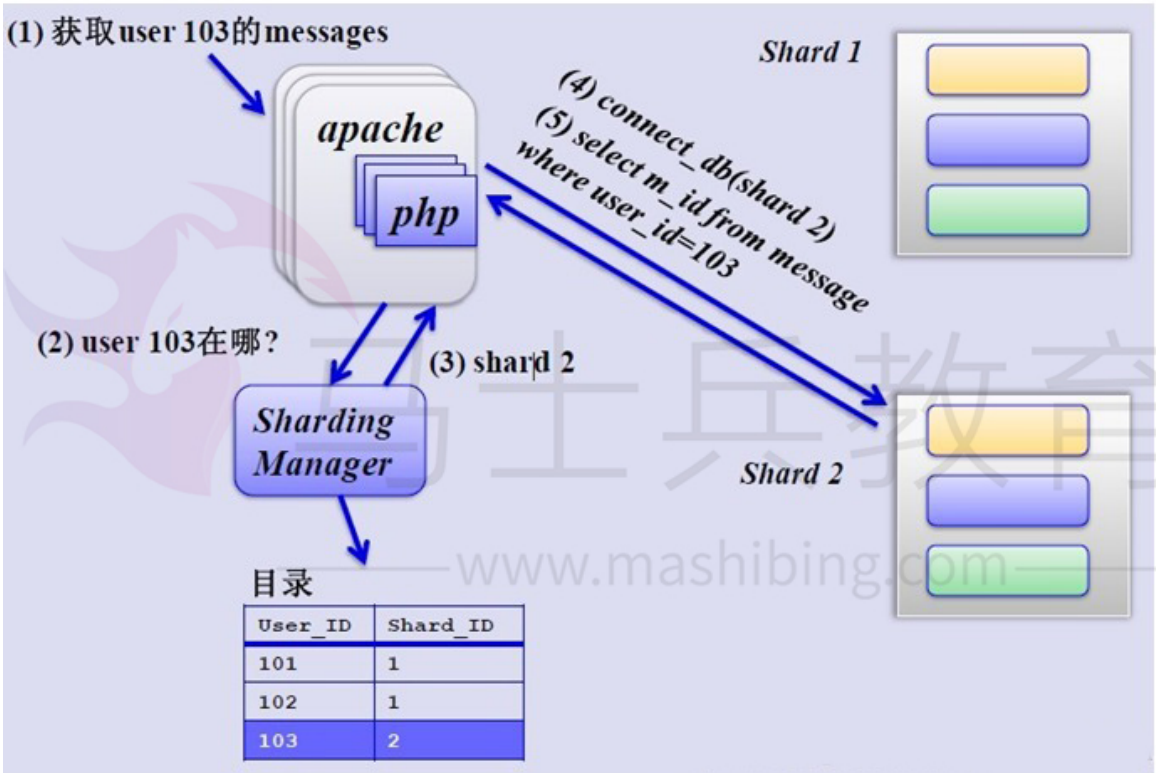
然而，尽管业务之间已经足够独立了，但是有些业务之间或多或少总会有点联系，如用户，基本上都会和每个业务相关联，况且这种分区方式，也不能解决单张表数据量暴涨的问题，因此为何不试试水平分割呢?

3、MySQL水平分片（Sharding）

这是一个非常好的思路，将用户按一定规则（按id哈希）分组，并把该组用户的数据存储到一个数据库分片中，即一个sharding，这样随着用户数量的增加，只要简单地配置一台服务器即可，原理图如下：



如何来确定某个用户所在的shard呢，可以建一张用户和shard对应的数据表，每次请求先从这张表找用户的shard id，再从对应shard中查询相关数据，如下图所示：



mysql大数据量使用limit分页，随着页码的增大，查询效率越低下。

测试实验

1. 直接用limit start, count分页语句， 也是我程序中用的方法：

select * from product limit start, count
当起始页较小时，查询没有性能问题，我们分别看下从10， 100， 1000， 10000开始分页的执行时间（每页取20条）， 如下：

select * from product limit 10, 20 0.016秒
select * from product limit 100, 20 0.016秒
select * from product limit 1000, 20 0.047秒
select * from product limit 10000, 20 0.094秒

我们已经看出随着起始记录的增加，时间也随着增大， 这说明分页语句limit跟起始页码是有很大关系的，那么我们把起始记录改为40w看下（也就是记录的一般左右） select * from product limit 400000, 20 3.229秒

再看我们取最后一页记录的时间
select * from product limit 866613, 20 37.44秒

难怪搜索引擎抓取我们页面的时候经常会报超时，像这种分页最大的页码页显然这种时间是无法忍受的。

从中我们也能总结出两件事情：
1）limit语句的查询时间与起始记录的位置成正比
2）mysql的limit语句是很方便，但是对记录很多的表并不适合直接使用。

2. 对limit分页问题的性能优化方法

利用表的覆盖索引来加速分页查询
我们都知道，利用了索引查询的语句中如果只包含了那个索引列（覆盖索引），那么这种情况会查询很快。

因为利用索引查找有优化算法，且数据就在查询索引上面，不用再去找相关的数据地址了，这样节省了很多时间。另外Mysql中也有相关的索引缓存，在并发高的时候利用缓存就效果更好了。

在我们的例子中，我们知道id字段是主键，自然就包含了默认的主键索引。现在让我们看看利用覆盖索引的查询效果如何：

这次我们之间查询最后一页的数据（利用覆盖索引，只包含id列），如下：
select id from product limit 866613, 20 0.2秒
相对于查询了所有列的37.44秒，提升了大概100多倍的速度

那么如果我们也要查询所有列，有两种方法，一种是id>=的形式，另一种就是利用join，看下实际情况：

SELECT * FROM product WHERE ID > =(select id from product limit 866613, 1) limit 20
查询时间为0.2秒，简直是一个质的飞跃啊，哈哈

另一种写法
SELECT * FROM product a JOIN (select id from product limit 866613, 20) b ON a.ID = b.id
查询时间也很短，赞！

其实两者用的都是一个原理嘛，所以效果也差不多

MySQL分表自增ID解决方案

当我们对MySQL进行分表操作后，将不能依赖MySQL的自动增量来产生唯一ID了，因为数据已经分散到多个表中。

应尽量避免使用自增IP来做为主键，为数据库分表操作带来极大的不便。

在postgreSQL、oracle、db2数据库中有一个特殊的特性---sequence。 任何时候数据库可以根据当前表中的记录数大小和步长来获取到该表下一条记录数。然而，MySQL是没有这种序列对象的。

可以通过下面的方法来实现sequence特性产生唯一ID：
1. 通过MySQL表生成ID
在《关于MySQL分表操作的研究》提到了一种方法：
对于插入也就是insert操作，首先就是获取唯一的id了，就需要一个表来专门创建id，插入一条记录，并获取最后插入的ID。代码如下：

```
CREATE TABLE `ttlsa_com`.`create_id` (
`id` BIGINT( 20 ) NOT NULL AUTO_INCREMENT PRIMARY KEY
) ENGINE = MYISAM
```

也就是说，当我们需要插入数据的时候，必须由这个表来产生id值，我的php代码的方法如下：

```
<?php
function get_AI_ID() {
    $sql = "insert into create_id (id) values('')";
    $this->db->query($sql);
    return $this->db->insertID();
}
?>
```

这种方法效果很好，但是在高并发情况下，MySQL的AUTO_INCREMENT将导致整个数据库慢。如果存在自增字段，MySQL会维护一个自增锁，innodb会在内存里保存一个计数器来记录auto_increment值，当插入一个新行数据时，就会用一个表锁来锁住这个计数器，直到插入结束。如果是一行一行的插入是没有问题的，但是在高并发情况下，那就悲催了，表锁会引起SQL阻塞，极大的影响性能，还可能会达到max_connections值。
innodb_autoinc_lock_mode：可以设定3个值：0、1、2
0：traditonal （每次都会产生表锁）
1：consecutive （默认,可预判行数时使用新方式，不可时使用表锁，对于simple insert会获得批量的锁，保证连续插入）
2：interleaved （不会锁表，来一个处理一个，并发最高）
对于myisam表引擎是traditional，每次都会进行表锁的。

2. 通过redis生成ID

```
function get_next_autoincrement_waitlock($timeout = 60){
    $count = $timeout > 0 ? $timeout : 60;

    while($r->get("serial:lock")){
        $count++;
        sleep(1);
```

```
        if($count >10)
            return false;
    }
    return true;
}
function get_next_autoincrement($timeout = 60){
    // first check if we are locked...
    if (get_next_autoincrement_waitlock($timeout) == false)
            return 0; $id = $r->incr("serial");
    if ( $id > 1 )
        return $id;
    // if ID == 1, we assume we do not have "serial" key...
    // first we need to get lock.
    if ($r->setnx("serial:lock"), 1){
        $r->expire("serial:lock", 60  5);
        // get max(id) from database.
        $id = select_db_query("select max(id) from user_posts");
        // or alternatively:
        // select id from user_posts order by id desc limit 1
        // increase it
        $id++;
        // update Redis key
        $r->set("serial", $id);
        // release the lock
        $r->del("serial:lock");
        return $id;
    }
    // can not get lock.
    return 0;
}
$r = new Redis();
$r->connect("127.0.0.1", "6379");

$id = get_next_autoincrement();
if ($id){
    $sql = "insert into
user_posts(id,user,message)values($id,'$user','$message')"
    $data = exec_db_query($sql);
}
```

3. 队列方式

使用队列服务，如redis、memcacheq等等，将一定量的ID预分配在一个队列里，每次插入操作，先从队列中获取一个ID，若插入失败的话，将该ID再次添加到队列中，同时监控队列数量，当小于阀值时，自动向队列中添加元素。

这种方式可以有规划的对ID进行分配，还会带来经济效应，比如ＱＱ号码，各种靓号，明码标价。如网站的userid, 允许uid登陆，推出各种靓号，明码标价，对于普通的ID打乱后再随机分配。

```
<?php
class common {

    private $r;

    function construct() {
        $this->__construct();
```

```
    }
    public function __construct(){
        $this->r=new Redis();
        $this->r->connect('127.0.0.1', 6379);
    }

    function set_queue_id($ids){
        if(is_array($ids) && isset($ids)){
            foreach ($ids as $id){
                $this->r->LPUSH('next_autoincrement',$id);
            }
        }
    }
    function get_next_autoincrement(){
        return $this->r->LPOP('next_autoincrement');
    }
}

$createid=array();
while(count($createid)<20){
    $num=rand(1000,4000);
    if(!in_array($num,$createid))
        $createid[]=$num;
}

$id=new common();
$id->set_queue_id($createid);

var_dump($id->get_next_autoincrement());
```

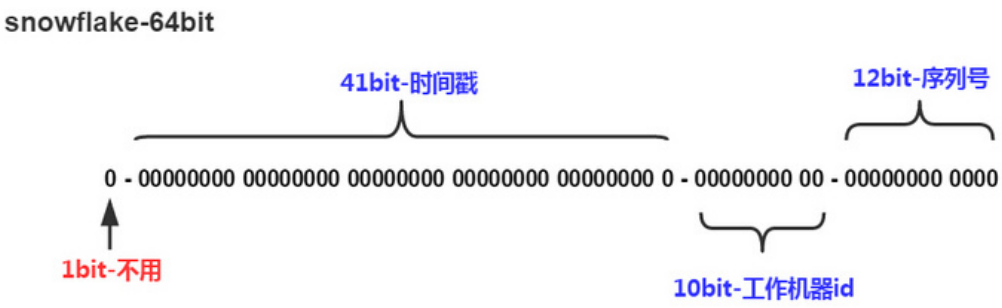
监控队列数量，并自动补充队列和取到id但并没有使用，相关代码没有贴出来。

理解分布式id生成算法SnowFlake

> 分布式id生成算法的有很多种，Twitter的SnowFlake就是其中经典的一种。

概述

SnowFlake算法生成id的结果是一个64bit大小的整数，它的结构如下图：



- `1位`，不用。二进制中最高位为1的都是负数，但是我们生成的id一般都使用整数，所以这个最高位固定是0
- `41位`，用来记录时间戳（毫秒）。
  - 41位可以表示2^{41}-1241−1个数字，
  - 如果只用来表示正整数（计算机中正数包含0），可以表示的数值范围是：0 至 2^{41}-1241−1，减1是因为可表示的数值范围是从0开始算的，而不是1。
  - 也就是说41位可以表示2^{41}-1241−1个毫秒的值，转化成单位年则是(2^{41}-1) / (1000 * 60 * 60 * 24 * 365) = 69(241−1)/(1000∗60∗60∗24∗365)=69年
- `10位`，用来记录工作机器id。
  - 可以部署在2^{10} = 1024210=1024个节点，包括`5位datacenterId`和`5位workerId`
  - `5位（bit）`可以表示的最大正整数是2^{5}-1 = 3125−1=31，即可以用0、1、2、3、....31这32个数字，来表示不同的datecenterId或workerId
- `12位`，序列号，用来记录同毫秒内产生的不同id。
  - `12位（bit）`可以表示的最大正整数是2^{12}-1 = 4095212−1=4095，即可以用0、1、2、3、....4094这4095个数字，来表示同一机器同一时间截（毫秒)内产生的4095个ID序号

由于在Java中64bit的整数是long类型，所以在Java中SnowFlake算法生成的id就是long来存储的。

SnowFlake可以保证：

- 所有生成的id按时间趋势递增
- 整个分布式系统内不会产生重复id（因为有datacenterId和workerId来做区分）

Talk is cheap, show you the code

以下是Twitter官方原版的，用Scala写的，（当成Java看即可）：

```
/** Copyright 2010-2012 Twitter, Inc.*/
package com.twitter.service.snowflake

import com.twitter.ostrich.stats.Stats
import com.twitter.service.snowflake.gen._
import java.util.Random
import com.twitter.logging.Logger

/**
 * An object that generates IDs.
 * This is broken into a separate class in case
 * we ever want to support multiple worker threads
 * per process
 */
class IdWorker(
    val workerId: Long,
    val datacenterId: Long,
    private val reporter: Reporter,
    var sequence: Long = 0L) extends Snowflake.Iface {

  private[this] def genCounter(agent: String) = {
    Stats.incr("ids_generated")
    Stats.incr("ids_generated_%s".format(agent))
  }
  private[this] val exceptionCounter = Stats.getCounter("exceptions")
  private[this] val log = Logger.get
  private[this] val rand = new Random
```

```
  val twepoch = 1288834974657L

  private[this] val workerIdBits = 5L
  private[this] val datacenterIdBits = 5L
  private[this] val maxWorkerId = -1L ^ (-1L << workerIdBits)
  private[this] val maxDatacenterId = -1L ^ (-1L << datacenterIdBits)
  private[this] val sequenceBits = 12L

  private[this] val workerIdShift = sequenceBits
  private[this] val datacenterIdShift = sequenceBits + workerIdBits
  private[this] val timestampLeftShift = sequenceBits + workerIdBits +
datacenterIdBits
  private[this] val sequenceMask = -1L ^ (-1L << sequenceBits)

  private[this] var lastTimestamp = -1L

  // sanity check for workerId
  if (workerId > maxWorkerId || workerId < 0) {
    exceptionCounter.incr(1)
    throw new IllegalArgumentException("worker Id can't be greater than %d or
less than 0".format(maxWorkerId))
  }

  if (datacenterId > maxDatacenterId || datacenterId < 0) {
    exceptionCounter.incr(1)
    throw new IllegalArgumentException("datacenter Id can't be greater than %d or
less than 0".format(maxDatacenterId))
  }

  log.info("worker starting. timestamp left shift %d, datacenter id bits %d,
worker id bits %d, sequence bits %d, workerid %d",
    timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId)

  def get_id(useragent: String): Long = {
    if (!validUseragent(useragent)) {
      exceptionCounter.incr(1)
      throw new InvalidUserAgentError
    }

    val id = nextId()
    genCounter(useragent)

    reporter.report(new AuditLogEntry(id, useragent, rand.nextLong))
    id
  }

  def get_worker_id(): Long = workerId
  def get_datacenter_id(): Long = datacenterId
  def get_timestamp() = System.currentTimeMillis

  protected[snowflake] def nextId(): Long = synchronized {
    var timestamp = timeGen()

    if (timestamp < lastTimestamp) {
      exceptionCounter.incr(1)
      log.error("clock is moving backwards.  Rejecting requests until %d.",
lastTimestamp);
```

```
      throw new InvalidSystemClock("Clock moved backwards.  Refusing to generate
id for %d milliseconds".format(
        lastTimestamp - timestamp))
    }

    if (lastTimestamp == timestamp) {
      sequence = (sequence + 1) & sequenceMask
      if (sequence == 0) {
        timestamp = tilNextMillis(lastTimestamp)
      }
    } else {
      sequence = 0
    }

    lastTimestamp = timestamp
    ((timestamp - twepoch) << timestampLeftShift) |
      (datacenterId << datacenterIdShift) |
      (workerId << workerIdShift) |
      sequence
  }

  protected def tilNextMillis(lastTimestamp: Long): Long = {
    var timestamp = timeGen()
    while (timestamp <= lastTimestamp) {
      timestamp = timeGen()
    }
    timestamp
  }

  protected def timeGen(): Long = System.currentTimeMillis()

  val AgentParser = """([a-zA-Z][a-zA-Z\-0-9]*)""".r

  def validUseragent(useragent: String): Boolean = useragent match {
    case AgentParser(_) => true
    case _ => false
  }
}
```

Scala是一门可以编译成字节码的语言，简单理解是在Java语法基础上加上了很多语法糖，例如不用每条语句后写分号，可以使用动态类型等等。抱着试一试的心态，我把Scala版的代码“翻译”成Java版本的，对scala代码改动的地方如下：

```
/** Copyright 2010-2012 Twitter, Inc.*/
package com.twitter.service.snowflake

import com.twitter.ostrich.stats.Stats
import com.twitter.service.snowflake.gen._
import java.util.Random
import com.twitter.logging.Logger

/**
 * An object that generates IDs.
 * This is broken into a separate class in case
 * we ever want to support multiple worker threads
 * per process
 */
```

```
class IdWorker(                                          // |
    val workerId: Long,                                  // |
    val datacenterId: Long,                              // |<--这部分改成Java的构造函
数形式
    private val reporter: Reporter,//日志相关，删       // |
    var sequence: Long = 0L)                             // |
       extends Snowflake.Iface { //接口找不到，删       // |

  private[this] def genCounter(agent: String) = {                     // |
    Stats.incr("ids_generated")                                       // |
    Stats.incr("ids_generated_%s".format(agent))                      // |<--错
误、日志处理相关，删
  }                                                                   // |
  private[this] val exceptionCounter = Stats.getCounter("exceptions") // |
  private[this] val log = Logger.get                                  // |
  private[this] val rand = new Random                                 // |

  val twepoch = 1288834974657L

  private[this] val workerIdBits = 5L
  private[this] val datacenterIdBits = 5L
  private[this] val maxWorkerId = -1L ^ (-1L << workerIdBits)
  private[this] val maxDatacenterId = -1L ^ (-1L << datacenterIdBits)
  private[this] val sequenceBits = 12L

  private[this] val workerIdShift = sequenceBits
  private[this] val datacenterIdShift = sequenceBits + workerIdBits
  private[this] val timestampLeftShift = sequenceBits + workerIdBits +
datacenterIdBits
  private[this] val sequenceMask = -1L ^ (-1L << sequenceBits)

  private[this] var lastTimestamp = -1L

  //------------------------------------------------------------------------------
-----------------------------------------------//
  // sanity check for workerId
                                                    //
  if (workerId > maxWorkerId || workerId < 0) {
                                                    //
    exceptionCounter.incr(1) //<--错误处理相关，删
                                                    //
    throw new IllegalArgumentException("worker Id can't be greater than %d or
less than 0".format(maxWorkerId))                  //这
    // |-->改成：throw new IllegalArgumentException
                                                     //部
    //            (String.format("worker Id can't be greater than %d or less than
0",maxWorkerId))                                   //分
  }
                                                    //放

                                                     //到
  if (datacenterId > maxDatacenterId || datacenterId < 0) {
                                                    //构
    exceptionCounter.incr(1) //<--错误处理相关，删
                                                    //造
    throw new IllegalArgumentException("datacenter Id can't be greater than %d or
less than 0".format(maxDatacenterId))          //函
```

```
    // |-->改成：throw new IllegalArgumentException
                                                //数
    //             (String.format("datacenter Id can't be greater than %d or less
than 0",maxDatacenterId))                       //中
  }
                                                //

                                                 //
  log.info("worker starting. timestamp left shift %d, datacenter id bits %d,
worker id bits %d, sequence bits %d, workerid %d", //
    timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId)
                                                //
  // |-->改成：System.out.printf("worker...%d...",timestampLeftShift,...);
                                               //
  //-------------------------------------------------------------------------------
-----------------------------------------------//

  //-------------------------------------------------------------------------//
  //这个函数删除错误处理相关的代码后，剩下一行代码：val id = nextId()      //
  //所以我们直接调用nextId()函数可以了，所以在“翻译”时可以删除这个函数      //
  def get_id(useragent: String): Long = {                                    //
    if (!validUseragent(useragent)) {                                        //
      exceptionCounter.incr(1)                                               //
      throw new InvalidUserAgentError                                        //删
    }                                                                        //除
                                                                             //
    val id = nextId()                                                        //
    genCounter(useragent)                                                    //
                                                                             //
    reporter.report(new AuditLogEntry(id, useragent, rand.nextLong))         //
    id                                                                       //
  }                                                                          //
  //-------------------------------------------------------------------------//

  def get_worker_id(): Long = workerId           // |
  def get_datacenter_id(): Long = datacenterId   // |<--改成Java函数
  def get_timestamp() = System.currentTimeMillis // |

  protected[snowflake] def nextId(): Long = synchronized { // 改成Java函数
    var timestamp = timeGen()

    if (timestamp < lastTimestamp) {
      exceptionCounter.incr(1) // 错误处理相关，删
      log.error("clock is moving backwards.  Rejecting requests until %d.",
lastTimestamp); // 改成System.err.printf(...)
      throw new InvalidSystemClock("Clock moved backwards.  Refusing to generate
id for %d milliseconds".format(
        lastTimestamp - timestamp)) // 改成RumTimeException
    }

    if (lastTimestamp == timestamp) {
      sequence = (sequence + 1) & sequenceMask
      if (sequence == 0) {
        timestamp = tilNextMillis(lastTimestamp)
      }
    } else {
      sequence = 0
    }
```

```
    lastTimestamp = timestamp
    ((timestamp - twepoch) << timestampLeftShift) | // |<--加上关键字return
      (datacenterId << datacenterIdShift) |         // |
      (workerId << workerIdShift) |                 // |
      sequence                                      // |
  }

  protected def tilNextMillis(lastTimestamp: Long): Long = { // 改成Java函数
    var timestamp = timeGen()
    while (timestamp <= lastTimestamp) {
      timestamp = timeGen()
    }
    timestamp // 加上关键字return
  }

  protected def timeGen(): Long = System.currentTimeMillis() // 改成Java函数

  val AgentParser = """([a-zA-Z][a-zA-Z\-0-9]*)""".r                  // |
                                                                      // |
  def validUseragent(useragent: String): Boolean = useragent match {  // |<--日志
相关，删
    case AgentParser(_) => true                                       // |
    case _ => false                                                   // |
  }                                                                   // |
}
```

改出来的Java版：

```
public class IdWorker{

    private long workerId;
    private long datacenterId;
    private long sequence;

    public IdWorker(long workerId, long datacenterId, long sequence){
        // sanity check for workerId
        if (workerId > maxWorkerId || workerId < 0) {
            throw new IllegalArgumentException(String.format("worker Id can't be
greater than %d or less than 0",maxWorkerId));
        }
        if (datacenterId > maxDatacenterId || datacenterId < 0) {
            throw new IllegalArgumentException(String.format("datacenter Id can't
be greater than %d or less than 0",maxDatacenterId));
        }
        System.out.printf("worker starting. timestamp left shift %d, datacenter
id bits %d, worker id bits %d, sequence bits %d, workerid %d",
                timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits,
workerId);

        this.workerId = workerId;
        this.datacenterId = datacenterId;
        this.sequence = sequence;
    }

    private long twepoch = 1288834974657L;
```

```
    private long workerIdBits = 5L;
    private long datacenterIdBits = 5L;
    private long maxWorkerId = -1L ^ (-1L << workerIdBits);
    private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);
    private long sequenceBits = 12L;

    private long workerIdShift = sequenceBits;
    private long datacenterIdShift = sequenceBits + workerIdBits;
    private long timestampLeftShift = sequenceBits + workerIdBits +
datacenterIdBits;
    private long sequenceMask = -1L ^ (-1L << sequenceBits);

    private long lastTimestamp = -1L;

    public long getWorkerId(){
        return workerId;
    }

    public long getDatacenterId(){
        return datacenterId;
    }

    public long getTimestamp(){
        return System.currentTimeMillis();
    }

    public synchronized long nextId() {
        long timestamp = timeGen();

        if (timestamp < lastTimestamp) {
            System.err.printf("clock is moving backwards.  Rejecting requests
until %d.", lastTimestamp);
            throw new RuntimeException(String.format("Clock moved backwards.
Refusing to generate id for %d milliseconds",
                    lastTimestamp - timestamp));
        }

        if (lastTimestamp == timestamp) {
            sequence = (sequence + 1) & sequenceMask;
            if (sequence == 0) {
                timestamp = tilNextMillis(lastTimestamp);
            }
        } else {
            sequence = 0;
        }

        lastTimestamp = timestamp;
        return ((timestamp - twepoch) << timestampLeftShift) |
                (datacenterId << datacenterIdShift) |
                (workerId << workerIdShift) |
                sequence;
    }

    private long tilNextMillis(long lastTimestamp) {
        long timestamp = timeGen();
        while (timestamp <= lastTimestamp) {
            timestamp = timeGen();
        }
```

```
        return timestamp;
    }

    private long timeGen(){
        return System.currentTimeMillis();
    }

    //---------------测试---------------
    public static void main(String[] args) {
        IdWorker worker = new IdWorker(1,1,1);
        for (int i = 0; i < 30; i++) {
            System.out.println(worker.nextId());
        }
    }

}
```

代码理解

上面的代码中，有部分位运算的代码，如：

```
sequence = (sequence + 1) & sequenceMask;

private long maxWorkerId = -1L ^ (-1L << workerIdBits);

return ((timestamp - twepoch) << timestampLeftShift) |
        (datacenterId << datacenterIdShift) |
        (workerId << workerIdShift) |
        sequence;
```

为了能更好理解，我对相关知识研究了一下。

负数的二进制表示

在计算机中，负数的二进制是用`补码`来表示的。
假设我是用Java中的int类型来存储数字的，
int类型的大小是32个二进制位（bit），即4个字节（byte）。（1 byte = 8 bit)
那么十进制数字`3`在二进制中的表示应该是这样的：

```
00000000 00000000 00000000 00000011
// 3的二进制表示，就是原码
```

那数字`-3`在二进制中应该如何表示？
我们可以反过来想想，因为-3+3=0，
在二进制运算中`把-3的二进制看成未知数x来求解`，
求解算式的二进制表示如下：

```
   00000000 00000000 00000000 00000011 //3，原码
+  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx //-3，补码
-----------------------------------------------
   00000000 00000000 00000000 00000000
```

反推x的值，3的二进制加上什么值才使结果变成`00000000 00000000 00000000 00000000`？：

```
   00000000 00000000 00000000 00000011 //3，原码
+  11111111 11111111 11111111 11111101 //-3，补码
-----------------------------------------------
 1 00000000 00000000 00000000 00000000
```

反推的思路是3的二进制数从最低位开始逐位加1，使溢出的1不断向高位溢出，直到溢出到第33位。然后由于int类型最多只能保存32个二进制位，所以最高位的1溢出了，剩下的32位就成了（十进制的）0。

补码的意义就是可以拿补码和原码（3的二进制）相加，最终加出一个"溢出的0"

以上是理解的过程，实际中记住**公式**就很容易算出来：

- 补码 = 反码 + 1
- 补码 = （原码 - 1）再取反码

因此`-1`的二进制应该这样算：

```
00000000 00000000 00000000 00000001 //原码：1的二进制
11111111 11111111 11111111 11111110 //取反码：1的二进制的反码
11111111 11111111 11111111 11111111 //加1：-1的二进制表示（补码）
```

用位运算计算n个bit能表示的最大数值

比如这样一行代码：

```
    private long workerIdBits = 5L;
    private long maxWorkerId = -1L ^ (-1L << workerIdBits);
```

上面代码换成这样看方便一点：
`long maxWorkerId = -1L ^ (-1L << 5L)`

咋一看真的看不准哪个部分先计算，于是查了一下Java运算符的优先级表：

优先级	运算符	结合性
1	() [] .	从左到右
2	! +(正) -(负) ~ ++ --	从右向左
3	* / %	从左向右
4	+(加) -(减)	从左向右
5	<< >> >>>	从左向右
6	< <= > >= instanceof	从左向右
7	== !=	从左向右
8	&(按位与)	从左向右
9	^	从左向右
10	|	从左向右
11	&&	从左向右
12	||	从左向右
13	?:	从右向左
14	= += -= *= /= %= &= |= ^= ~= <<= >>= >>>=	从右向左

所以上面那行代码中，运行顺序是：

- -1 左移 5，得结果a
- -1 异或 a

`long maxWorkerId = -1L ^ (-1L << 5L)`的二进制运算过程如下：

-1 左移 5，得结果a：

```
        11111111 11111111 11111111 11111111 //-1的二进制表示（补码）
  11111 11111111 11111111 11111111 11100000 //高位溢出的不要，低位补0
        11111111 11111111 11111111 11100000 //结果a
```

-1 异或 a：

```
        11111111 11111111 11111111 11111111 //-1的二进制表示（补码）
    ^   11111111 11111111 11111111 11100000 //两个操作数的位中，相同则为0，不同则为1
--------------------------------------------------------------------------------
        00000000 00000000 00000000 00011111 //最终结果31
```

最终结果是31，二进制`00000000 00000000 00000000 00011111`转十进制可以这么算：
2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 16 + 8 + 4 + 2 + 1 =3124+23+22+21+20=16+8+4+2+1=31

那既然现在知道算出来`long maxWorkerId = -1L ^ (-1L << 5L)`中的`maxWorkerId = 31`，有什么含义？为什么要用左移5来算？如果你看过`概述`部分，请找到这段内容看看：

> `5位（bit）`可以表示的最大正整数是2^{5}-1 = 3125−1=31，即可以用0、1、2、3、....31这32个数字，来表示不同的datecenterId或workerId

```
-1L ^ (-1L << 5L)`结果是`31`，2^{5}-125−1的结果也是`31`，所以在代码中，`-1L ^ (-1L << 5L)`的写法是`利用位运算计算出5位能表示的最大正整数是多少
```

用mask防止溢出

有一段有趣的代码：

```
sequence = (sequence + 1) & sequenceMask;
```

分别用不同的值测试一下，你就知道它怎么有趣了：

```
        long seqMask = -1L ^ (-1L << 12L); //计算12位能耐存储的最大正整数，相当于：
2^12-1 = 4095
        System.out.println("seqMask: "+seqMask);
        System.out.println(1L & seqMask);
        System.out.println(2L & seqMask);
        System.out.println(3L & seqMask);
        System.out.println(4L & seqMask);
        System.out.println(4095L & seqMask);
        System.out.println(4096L & seqMask);
        System.out.println(4097L & seqMask);
        System.out.println(4098L & seqMask);


        /**
        seqMask: 4095
        1
        2
```

```
        3
        4
        4095
        0
        1
        2
        */
```

这段代码通过`位与`运算保证计算的结果范围始终是 0-4095！

用位运算汇总结果

还有另外一段诡异的代码：

```
return ((timestamp - twepoch) << timestampLeftShift) |
        (datacenterId << datacenterIdShift) |
        (workerId << workerIdShift) |
        sequence;
```

为了弄清楚这段代码，

`首先` **需要计算一下相关的值：**

```
    private long twepoch = 1288834974657L; //起始时间戳，用于用当前时间戳减去这个时间戳，算出偏移量

    private long workerIdBits = 5L; //workerId占用的位数：5
    private long datacenterIdBits = 5L; //datacenterId占用的位数：5
    private long maxWorkerId = -1L ^ (-1L << workerIdBits);  // workerId可以使用的最大数值：31
    private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits); //datacenterId可以使用的最大数值：31
    private long sequenceBits = 12L;//序列号占用的位数：12

    private long workerIdShift = sequenceBits; // 12
    private long datacenterIdShift = sequenceBits + workerIdBits; // 12+5 = 17
    private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits; // 12+5+5 = 22
    private long sequenceMask = -1L ^ (-1L << sequenceBits);//4095

    private long lastTimestamp = -1L;
```

`其次` **写个测试，把参数都写死，并运行打印信息，方便后面来核对计算结果：**

```
    //---------------测试---------------
    public static void main(String[] args) {
        long timestamp = 1505914988849L;
        long twepoch = 1288834974657L;
        long datacenterId = 17L;
        long workerId = 25L;
        long sequence = 0L;

        System.out.printf("\ntimestamp: %d \n",timestamp);
        System.out.printf("twepoch: %d \n",twepoch);
        System.out.printf("datacenterId: %d \n",datacenterId);
        System.out.printf("workerId: %d \n",workerId);
```

```
        System.out.printf("sequence: %d \n",sequence);
        System.out.println();
        System.out.printf("(timestamp - twepoch): %d \n",(timestamp - twepoch));
        System.out.printf("((timestamp - twepoch) << 22L): %d \n",((timestamp -
twepoch) << 22L));
        System.out.printf("(datacenterId << 17L): %d \n" ,(datacenterId <<
17L));
        System.out.printf("(workerId << 12L): %d \n",(workerId << 12L));
        System.out.printf("sequence: %d \n",sequence);

        long result = ((timestamp - twepoch) << 22L) |
                (datacenterId << 17L) |
                (workerId << 12L) |
                sequence;
        System.out.println(result);

    }

    /** 打印信息:
        timestamp: 1505914988849
        twepoch: 1288834974657
        datacenterId: 17
        workerId: 25
        sequence: 0

        (timestamp - twepoch): 217080014192
        ((timestamp - twepoch) << 22L): 910499571845562368
        (datacenterId << 17L): 2228224
        (workerId << 12L): 102400
        sequence: 0
        910499571847892992
    */
```

代入位移的值得之后，就是这样：

```
return ((timestamp - 1288834974657) << 22) |
        (datacenterId << 17) |
        (workerId << 12) |
        sequence;
```

对于尚未知道的值，我们可以先看看`概述`中对SnowFlake结构的解释，再代入在合法范围的值(windows系统可以用计算器方便计算这些值的二进制)，来了解计算的过程。
当然，由于我的测试代码已经把这些值写死了，那直接用这些值来手工验证计算结果即可：

```
        long timestamp = 1505914988849L;
        long twepoch = 1288834974657L;
        long datacenterId = 17L;
        long workerId = 25L;
        long sequence = 0L;
```

设：timestamp = 1505914988849，twepoch = 1288834974657

1505914988849 - 1288834974657 = 217080014192 (timestamp相对于起始时间的毫秒偏移量)，其(a)二进制左移22位计算过程如下：

```
                          |<--这里开始左右22位
00000000 00000000 000000|00 00110010 10001010 11111010 00100101 01110000 // a = 217080014192
00001100 10100010 10111110 10001001 01011100 00|000000 00000000 00000000 // a左移22位后的值(la)
                                                 |<--这里后面的位补0
```

设：datacenterId = 17，其（b）二进制左移17位计算过程如下：

```
                    |<--这里开始左移17位
00000000 00000000 0|0000000 00000000 00000000 00000000 00000000 00010001 // b = 17
00000000 00000000 00000000 00000000 00000000 0010001|0 00000000 00000000 // b左移17位后的值(lb)
                                                      |<--这里后面的位补0
```

设：workerId = 25，其（c）二进制左移12位计算过程如下：

```
             |<--这里开始左移12位
00000000 0000|0000 00000000 00000000 00000000 00000000 00000000 00011001 // c = 25
00000000 00000000 00000000 00000000 00000000 00000001 1001|0000 00000000 // c左移12位后的值(lc)
                                                           |<--这里后面的位补0
```

设：sequence = 0，其二进制如下：

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 //
sequence = 0
```

现在知道了每个部分左移后的值(la,lb,lc)，代码可以简化成下面这样去理解：

```
return ((timestamp - 1288834974657) << 22) |
        (datacenterId << 17) |
        (workerId << 12) |
        sequence;
-----------------------------
           |
           |简化
          \|/
-----------------------------
return (la) |
        (lb) |
        (lc) |
        sequence;
```

上面的管道符号`|`在Java中也是一个位运算符。其含义是：
`x的第n位和y的第n位 只要有一个是1，则结果的第n位也为1，否则为0`，因此，我们对四个数的`位或运算`如下：

```
 1  |                   41                        |  5  |   5  |     12

   0|0001100 10100010 10111110 10001001 01011100 00|00000|0 0000|0000 00000000
//la
   0|0000000 00000000 00000000 00000000 00000000 00|10001|0 0000|0000 00000000
//lb
   0|0000000 00000000 00000000 00000000 00000000 00|00000|1 1001|0000 00000000
//lc
or 0|0000000 00000000 00000000 00000000 00000000 00|00000|0 0000|0000 00000000
//sequence
-------------------------------------------------------------------------------
---------
   0|0001100 10100010 10111110 10001001 01011100 00|10001|1 1001|0000 00000000 //
结果：910499571847892992
```

结果计算过程：
1) 从至左列出1出现的下标（从0开始算）：

```
0000  1   1   00  1   0  1  000  1   0  1  0  1  1  1  1  1  0 1   000 1 00 1  0
1  0   1  1  1  0000 1   000  1  1  1  00  1   0000 0000 0000
      59  58      55     53      49     47    45 44 43 42 41   39      35   32
30     28 27 26      21       17 16 15     12
```

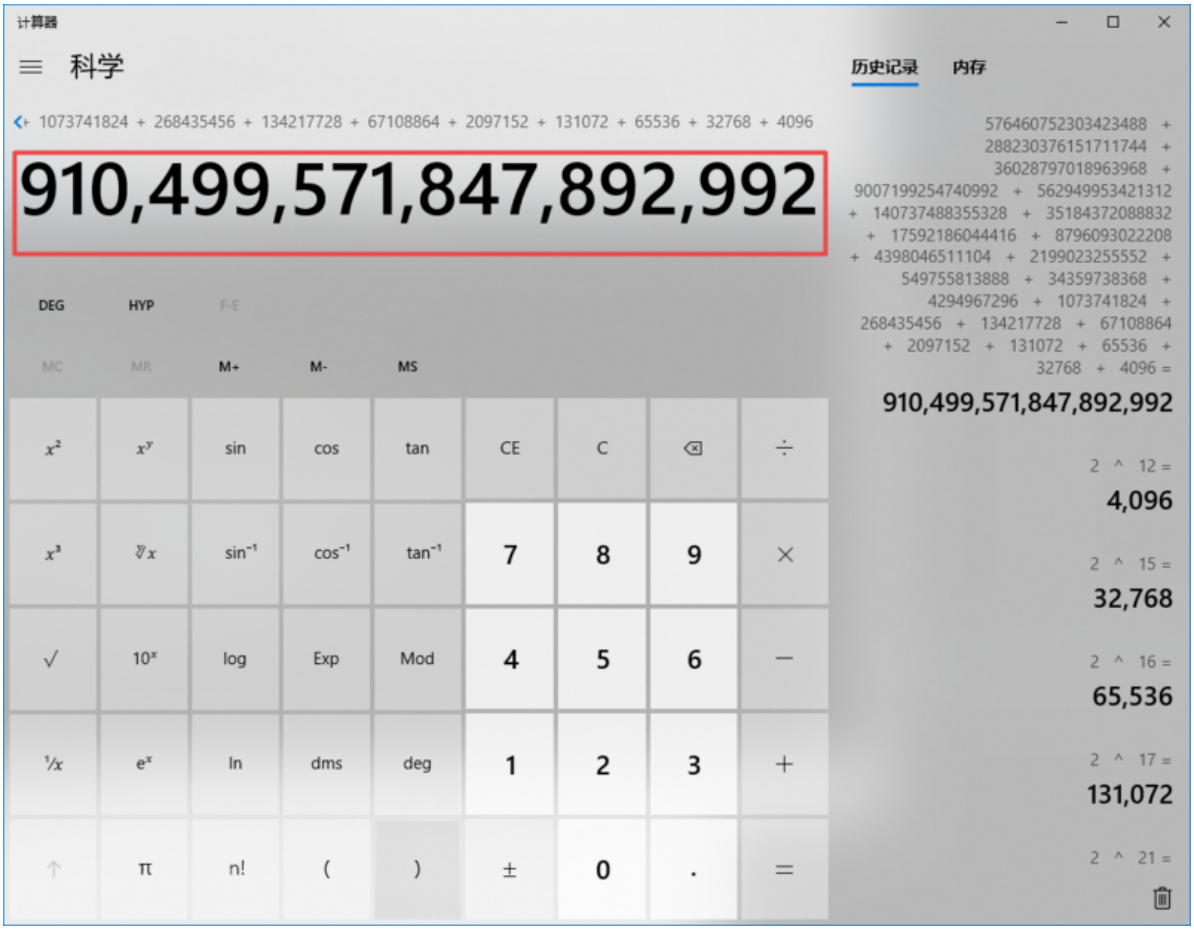
2) 各个下标作为2的幂数来计算，并相加：

> $ 2^{59}+2^{58}+2^{55}+2^{53}+2^{49}+2^{47}+2^{45}+2^{44}+2^{43}+
> 2^{42}+2^{41}+2^{39}+2^{35}+2^{32}+2^{30}+2^{28}+2^{27}+2^{26}+
> 2^{21}+2^{17}+2^{16}+2^{15}+2^{2} $

```
    2^59}  : 576460752303423488
    2^58}  : 288230376151711744
    2^55}  :  36028797018963968
    2^53}  :   9007199254740992
    2^49}  :    562949953421312
    2^47}  :    140737488355328
    2^45}  :     35184372088832
    2^44}  :     17592186044416
```

```
     2^43}   :       8796093022208
     2^42}   :       4398046511104
     2^41}   :       2199023255552
     2^39}   :        549755813888
     2^35}   :         34359738368
     2^32}   :          4294967296
     2^30}   :          1073741824
     2^28}   :           268435456
     2^27}   :           134217728
     2^26}   :            67108864
     2^21}   :             2097152
     2^17}   :              131072
     2^16}   :               65536
     2^15}   :               32768
+    2^12}   :                4096
----------------------------------------
              910499571847892992
```

计算截图：



跟测试程序打印出来的结果一样，手工验证完毕！

观察

```
 1  |                    41                          |  5  |   5  |     12

   0|0001100 10100010 10111110 10001001 01011100 00|     |      |
//la
   0|                                               |10001|      |
//lb
   0|                                               |     |1 1001|
//lc
or 0|                                               |     |      |0000 00000000
//sequence
-------------------------------------------------------------------------------
---------
   0|0001100 10100010 10111110 10001001 01011100 00|10001|1 1001|0000 00000000 //
结果：910499571847892992
```

上面的64位我按1、41、5、5、12的位数截开了，方便观察。

- `纵向`观察发现:
  - 在41位那一段，除了la一行有值，其它行（lb、lc、sequence）都是0，（我爸其它）
  - 在左起第一个5位那一段，除了lb一行有值，其它行都是0
  - 在左起第二个5位那一段，除了lc一行有值，其它行都是0
  - 按照这规律，如果sequence是0以外的其它值，12位那段也会有值的，其它行都是0
- `横向`观察发现:
  - 在la行，由于左移了5+5+12位，5、5、12这三段都补0了，所以la行除了41那段外，其它肯定都是0
  - 同理，lb、lc、sequnece行也以此类推
  - 正因为左移的操作，使四个不同的值移到了SnowFlake理论上相应的位置，然后四行做`位或`运算（只要有1结果就是1），就把4段的二进制数合并成一个二进制数。

结论：
所以，在这段代码中

```
return ((timestamp - 1288834974657) << 22) |
        (datacenterId << 17) |
        (workerId << 12) |
        sequence;
```

左移运算是为了将数值移动到对应的段(41、5、5，12那段因为本来就在最右，因此不用左移)。

然后对每个左移后的值(la、lb、lc、sequence)做位或运算，是为了把各个短的数据合并起来，合并成一个二进制数。

最后转换成10进制，就是最终生成的id

扩展

在理解了这个算法之后，其实还有一些扩展的事情可以做：

1. 根据自己业务修改每个位段存储的信息。算法是通用的，可以根据自己需求适当调整每段的大小以及存储的信息。
2. 解密id，由于id的每段都保存了特定的信息，所以拿到一个id，应该可以尝试反推出原始的每个段的信息。反推出的信息可以帮助我们分析。比如作为订单，可以知道该订单的生成日期，负责处理的数据中心等等。

MySql的主从实时备份同步的配置，以及原理(从库读主库的binlog)，读写分离

Mysql主从同步的实现原理

**1、什么是mysql主从同步?

**

当master(主)库的数据发生变化的时候，变化会实时的同步到slave(从)库。

2、主从同步有什么好处?

- 水平扩展数据库的负载能力。
- 容错，高可用。Failover(失败切换)/High Availability
- 数据备份。

3、主从同步的原理是什么?

首先我们来了解master-slave的体系结构。

如下图:



不管是delete、update、insert，还是创建函数、存储过程，所有的操作都在master上。当master有操作的时候,slave会快速的接收到这些操作，从而做同步。

但是，这个机制是怎么实现的呢?

在master机器上，主从同步事件会被写到特殊的log文件中(binary-log);在slave机器上，slave读取主从同步事件，并根据读取的事件变化，在slave库上做相应的更改。

如此，就实现了主从同步了!

下面我们来详细的了解。

**3.1主从同步事件有哪些

**

上面说到:

在master机器上，主从同步事件会被写到特殊的log文件中(binary-log);

主从同步事件有3种形式:statement、row、mixed。

1. statement：会将对数据库操作的sql语句写入到binlog中。
2. row：会将每一条数据的变化写入到binlog中。
3. mixed：statement与row的混合。Mysql决定什么时候写statement格式的，什么时候写row格式的binlog。

3.2在master机器上的操作

当master上的数据发生改变的时候，该事件(insert、update、delete)变化会按照顺序写入到binlog中。

binlog dump线程

当slave连接到master的时候，master机器会为slave开启binlog dump线程。当master 的 binlog发生变化的时候，binlog dump线程会通知slave，并将相应的binlog内容发送给slave。

3.3在slave机器上的操作

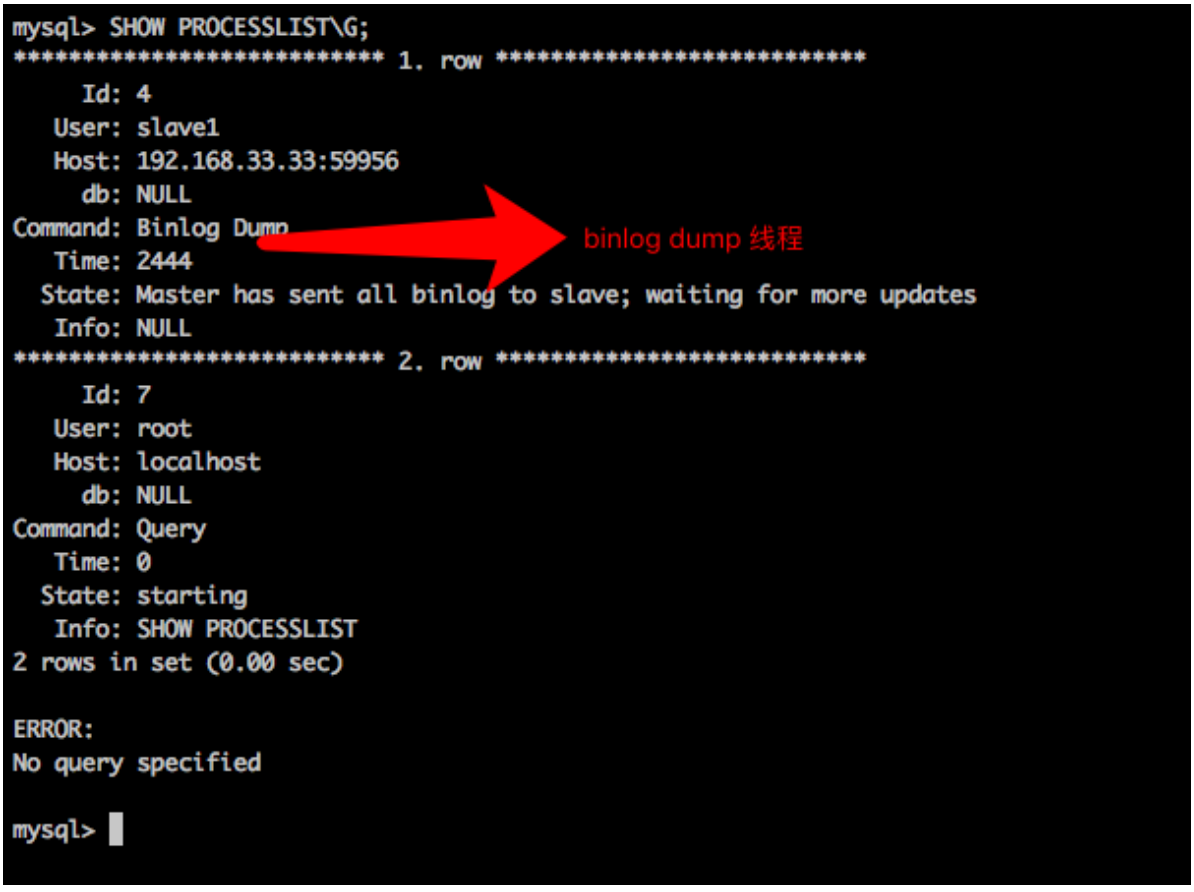
当主从同步开启的时候，slave上会创建2个线程。

- I/O线程。该线程连接到master机器，master机器上的binlog dump线程会将binlog的内容发送给该I/O线程。该I/O线程接收到binlog内容后，再将内容写入到本地的relay log。
- SQL线程。该线程读取I/O线程写入的relay log。并且根据relay log的内容对slave数据库做相应的操作。

3.4如何在master、slave上查看上述的线程?

使用SHOW PROCESSLIST命令可以查看。

如图，在master机器上查看binlog dump线程。



如图，在slave机器上查看I/O、SQL线程。

```
mysql> SHOW PROCESSLIST\G;
*************************** 1. row ***************************
     Id: 5
   User: root
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: starting
   Info: SHOW PROCESSLIST
*************************** 2. row ***************************
     Id: 8
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 15799
  State: Waiting for master to send event
   Info: NULL
*************************** 3. row ***************************
     Id: 9
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 2366
  State: Slave has read all relay log; waiting for more updates
   Info: NULL
3 rows in set (0.00 sec)

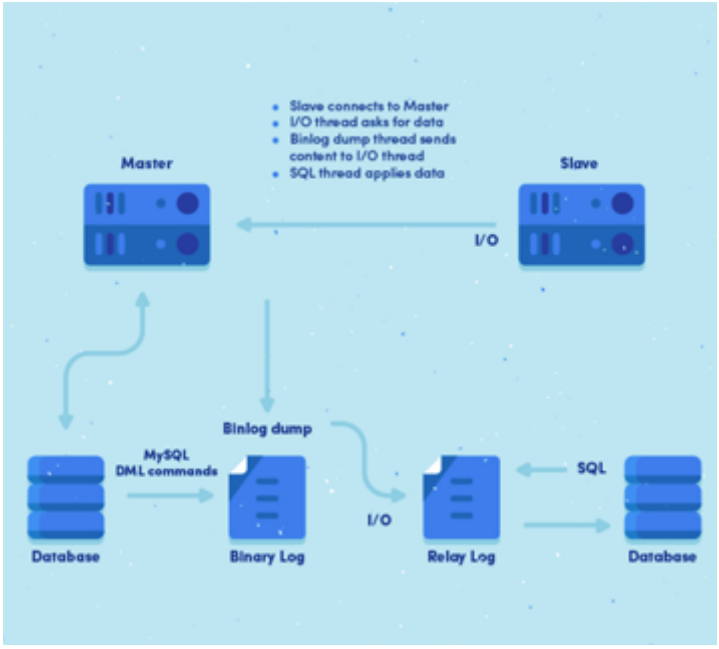
ERROR:
No query specified

mysql>
```

I/O threads

SQL threads

4、讲了这么多，一图以蔽之



MySQL索引背后的数据结构及算法原理

摘要

本文以MySQL数据库为研究对象，讨论与数据库索引相关的一些话题。特别需要说明的是，MySQL支持诸多存储引擎，而各种存储引擎对索引的支持也各不相同，因此MySQL数据库支持多种索引类型，如BTree索引，哈希索引，全文索引等等。为了避免混乱，本文将只关注于BTree索引，因为这是平常使用MySQL时主要打交道的索引，至于哈希索引和全文索引本文暂不讨论。

文章主要内容分为三个部分。

第一部分主要从数据结构及算法理论层面讨论MySQL数据库索引的数理基础。

第二部分结合MySQL数据库中MyISAM和InnoDB数据存储引擎中索引的架构实现讨论聚集索引、非聚集索引及覆盖索引等话题。

第三部分根据上面的理论基础，讨论MySQL中高性能使用索引的策略。

数据结构及算法基础

索引的本质

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：索引是数据结构。

我们知道，数据库查询是数据库的最主要功能之一。我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。最基本的查询算法当然是[顺序查找]（linear search），这种复杂度为O(n)的算法在数据量很大时显然是糟糕的，好在计算机科学的发展提供了很多更优秀的查找算法，例如[二分查找]（binary search）、[二叉树查找]（binary tree search）等。如果稍微分析一下会发现，每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织），所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

看一个例子：

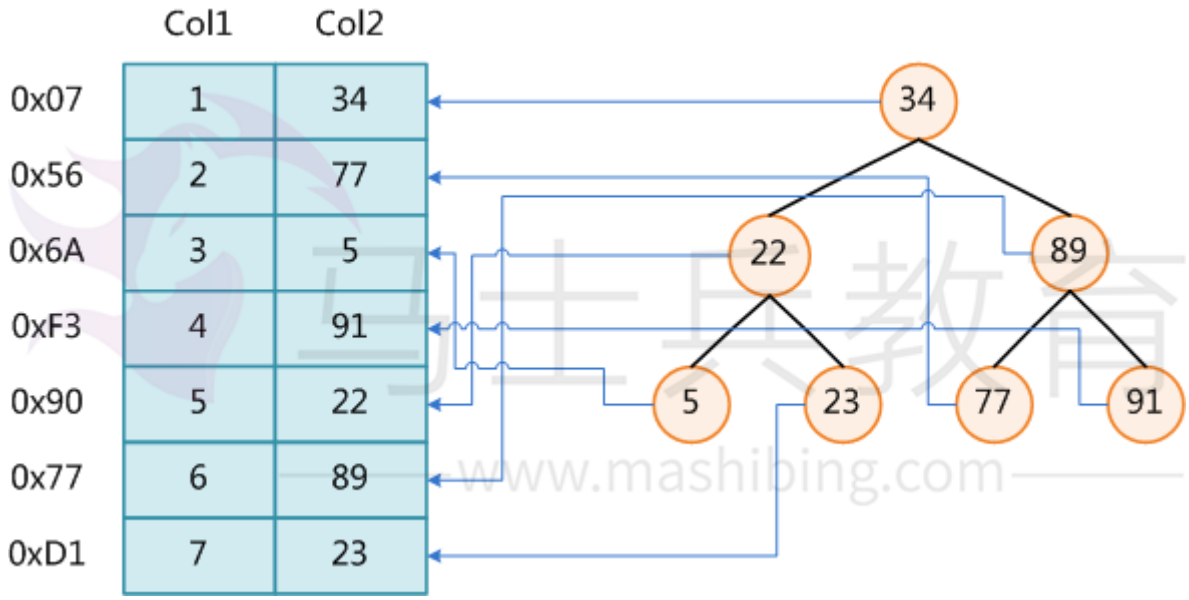


图1

图1展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)O(log2n)的复杂度内获取到相应数据。

虽然这是一个货真价实的索引，但是实际的数据库系统几乎没有使用二叉查找树或其进化品种红黑树（red-black tree）实现的，原因会在下文介绍。

B-Tree和B+Tree

目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构，在本文的下一节会结合存储器原理及计算机存取原理讨论为什么B-Tree和B+Tree在被如此广泛用于索引，这一节先单纯从数据结构角度描述它们。

B-Tree

为了描述B-Tree，首先定义一条数据记录为一个二元组[key, data]，key为记录的键值，对于不同数据记录，key是互不相同的；data为数据记录除key外的数据。那么B-Tree是满足下列条件的数据结构：

d为大于1的一个正整数，称为B-Tree的度。

h为一个正整数，称为B-Tree的高度。

每个非叶子节点由n-1个key和n个指针组成，其中d<=n<=2d。

每个叶子节点最少包含一个key和两个指针，最多包含2d-1个key和2d个指针，叶节点的指针均为null 。

所有叶节点具有相同的深度，等于树高h。

key和指针互相间隔，节点两端是指针。

一个节点中的key从左到右非递减排列。

所有节点组成树结构。

每个指针要么为null，要么指向另外一个节点。

如果某个指针在节点node最左边且不为null，则其指向节点的所有key小于v(key1)v(key1)，其中v(key1)v(key1)为node的第一个key的值。

如果某个指针在节点node最右边且不为null，则其指向节点的所有key大于v(keym)v(keym)，其中v(keym)v(keym)为node的最后一个key的值。

如果某个指针在节点node的左右相邻key分别是keyikeyi和keyi+1keyi+1且不为null，则其指向节点的所有key小于v(keyi+1)v(keyi+1)且大于v(keyi)v(keyi)。

图2是一个d=2的B-Tree示意图。

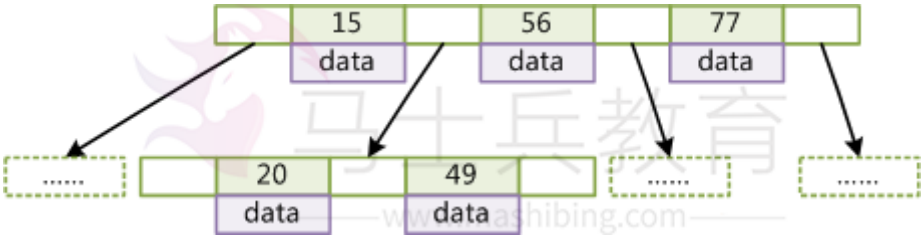


图2

由于B-Tree的特性，在B-Tree中按key检索数据的算法非常直观：首先从根节点进行二分查找，如果找到则返回对应节点的data，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。B-Tree上查找算法的伪代码如下：

```
BTree_Search(node, key) {    if(node == null) return null;    foreach(node.key)
{        if(node.key[i] == key) return node.data[i];            if(node.key[i] >
key) return BTree_Search(point[i]->node);    }    return BTree_Search(point[i+1]-
>node);}data = BTree_Search(root, my_key);
```

关于B-Tree有一系列有趣的性质，例如一个度为d的B-Tree，设其索引N个key，则其树高h的上限为logd((N+1)/2)logd((N+1)/2)，检索一个key，其查找节点个数的渐进复杂度为O(logdN)O(logdN)。从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

另外，由于插入删除新的数据记录会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质，本文不打算完整讨论B-Tree这些内容，因为已经有许多资料详细说明了B-Tree的数学性质及插入删除算法，有兴趣的朋友可以在本文末的参考文献一栏找到相应的资料进行阅读。

B+Tree

B-Tree有许多变种，其中最常见的是B+Tree，例如MySQL就普遍使用B+Tree实现其索引结构。

与B-Tree相比，B+Tree有以下不同点：

每个节点的指针上限为2d而不是2d+1。

内节点不存储data，只存储key；叶子节点不存储指针。

图3是一个简单的B+Tree示意。

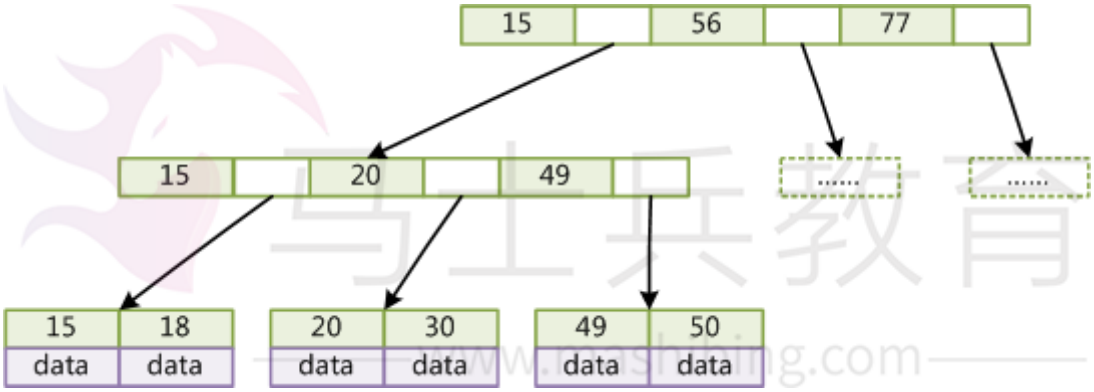


图3

由于并不是所有节点都具有相同的域，因此B+Tree中叶节点和内节点一般大小不同。这点与B-Tree不同，虽然B-Tree中不同节点存放的key和指针可能数量不一致，但是每个节点的域和上限是一致的，所以在实现中B-Tree往往对每个节点申请同等大小的空间。

一般来说，B+Tree比B-Tree更适合实现外存储索引结构，具体原因与外存储器原理及计算机存取原理有关，将在下面讨论。

带有顺序访问指针的B+Tree

一般在数据库系统或文件系统中使用的B+Tree结构都在经典B+Tree的基础上进行了优化，增加了顺序访问指针。

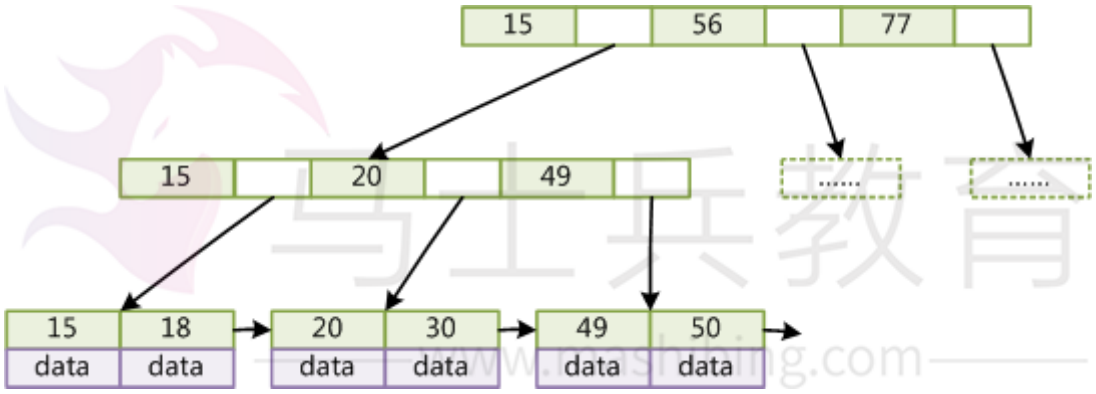


图4

如图4所示，在B+Tree的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的B+Tree。做这个优化的目的是为了提高区间访问的性能，例如图4中如果要查询key为从18到49的所有数据记录，当找到18后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提到了区间查询效率。

这一节对B-Tree和B+Tree进行了一个简单的介绍，下一节结合存储器存取原理介绍为什么目前B+Tree是数据库系统实现索引的首选数据结构。

为什么使用B-Tree（B+Tree）

上文说过，红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构，这一节将结合计算机组成原理相关知识讨论B-/+Tree作为索引的理论基础。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。下面先介绍内存和磁盘存取原理，然后再结合这些原理分析B-/+Tree作为索引的效率。

主存存取原理

目前计算机使用的主存基本都是随机读写存储器（RAM），现代RAM的结构和存取原理比较复杂，这里本文抛却具体差别，抽象出一个十分简单的存取模型来说明RAM的工作原理。

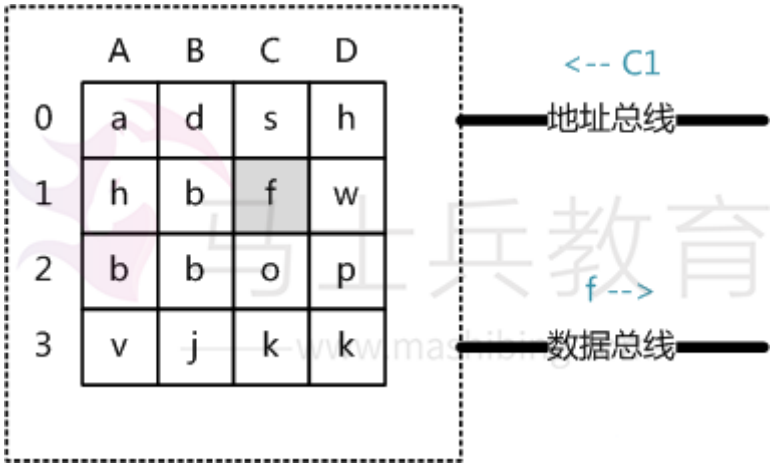


图5

从抽象角度看，主存是一系列的存储单元组成的矩阵，每个存储单元存储固定大小的数据。每个存储单元有唯一的地址，现代主存的编址规则比较复杂，这里将其简化成一个二维地址：通过一个行地址和一个列地址可以唯一定位到一个存储单元。图5展示了一个4 x 4的主存模型。

主存的存取过程如下：

当系统需要读取主存时，则将地址信号放到地址总线上传给主存，主存读到地址信号后，解析信号并定位到指定存储单元，然后将此存储单元数据放到数据总线上，供其它部件读取。

写主存的过程类似，系统将要写入单元地址和数据分别放在地址总线和数据总线上，主存读取两个总线的内容，做相应的写操作。

这里可以看出，主存存取的时间仅与存取次数呈线性关系，因为不存在机械操作，两次存取的数据的“距离”不会对时间有任何影响，例如，先取A0再取A1和先取A0再取D3的时间消耗是一样的。

磁盘存取原理

上文说过，索引一般以文件形式存储在磁盘上，索引检索需要磁盘I/O操作。与主存不同，磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。

图6是磁盘的整体结构示意图。

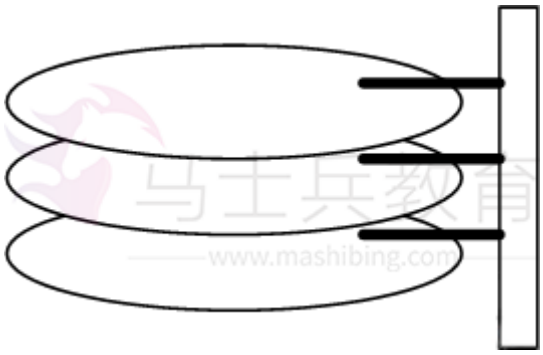


图6

一个磁盘由大小相同且同轴的圆形盘片组成，磁盘可以转动（各个磁盘必须同步转动）。在磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不能转动，但是可以沿磁盘半径方向运动（实际是斜切向运动），每个磁头同一时刻也必须是同轴的，即从正上方向下看，所有磁头任何时候都是重叠的（不过目前已经有多磁头独立技术，可不受此限制）。

图7是磁盘结构的示意图。

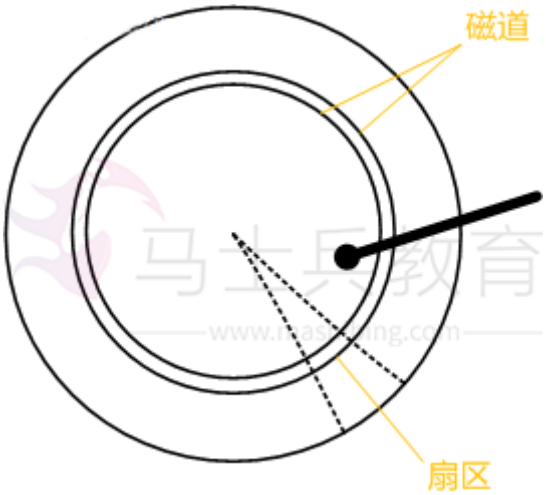


图7

盘片被划分成一系列同心环，圆心是盘片中心，每个同心环叫做一个磁道，所有半径相同的磁道组成一个柱面。磁道被沿半径线划分成一个个小的段，每个段叫做一个扇区，每个扇区是磁盘的最小存储单元。为了简单起见，我们下面假设磁盘只有一个盘片和一个磁头。

当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点，磁头需要移动对准相应磁道，这个过程叫做寻道，所耗费时间叫做寻道时间，然后磁盘旋转将目标扇区旋转到磁头下，这个过程耗费的时间叫做旋转时间。

局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：

当一个数据被用到时，其附近的数据也通常会马上被使用。

程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

B-/+Tree索引的性能分析

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

综上所述，用B-Tree作为索引结构效率是非常高的。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

上文还说过，B+Tree更适合外存索引，原因和内节点出度d有关。从上面分析可以看到，d越大索引的性能越好，而出度的上限取决于节点内key和data的大小：

dmax=floor(pagesize/(keysize+datasize+pointsize))dmax=floor(pagesize/(keysize+datasize+pointsize))

floor表示向下取整。由于B+Tree内节点去掉了data域，因此可以拥有更大的出度，拥有更好的性能。

这一章从理论角度讨论了与索引相关的数据结构与算法问题，下一章将讨论B+Tree是如何具体实现为MySQL中索引，同时将结合MyISAM和InnDB存储引擎介绍非聚集索引和聚集索引两种不同的索引实现形式。

MySQL索引实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式。

MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：

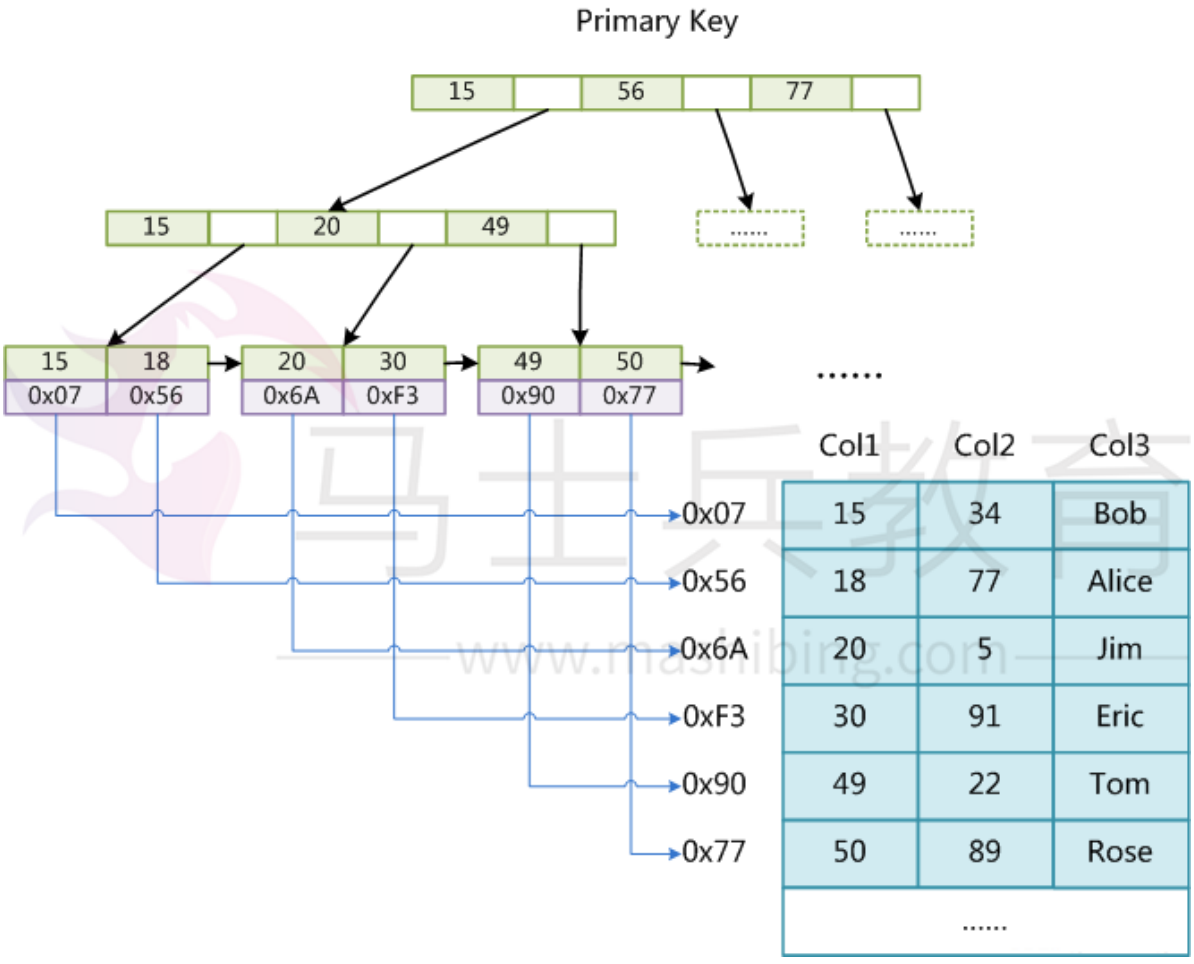


图8

这里设表一共有三列，假设我们以Col1为主键，则图8是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：

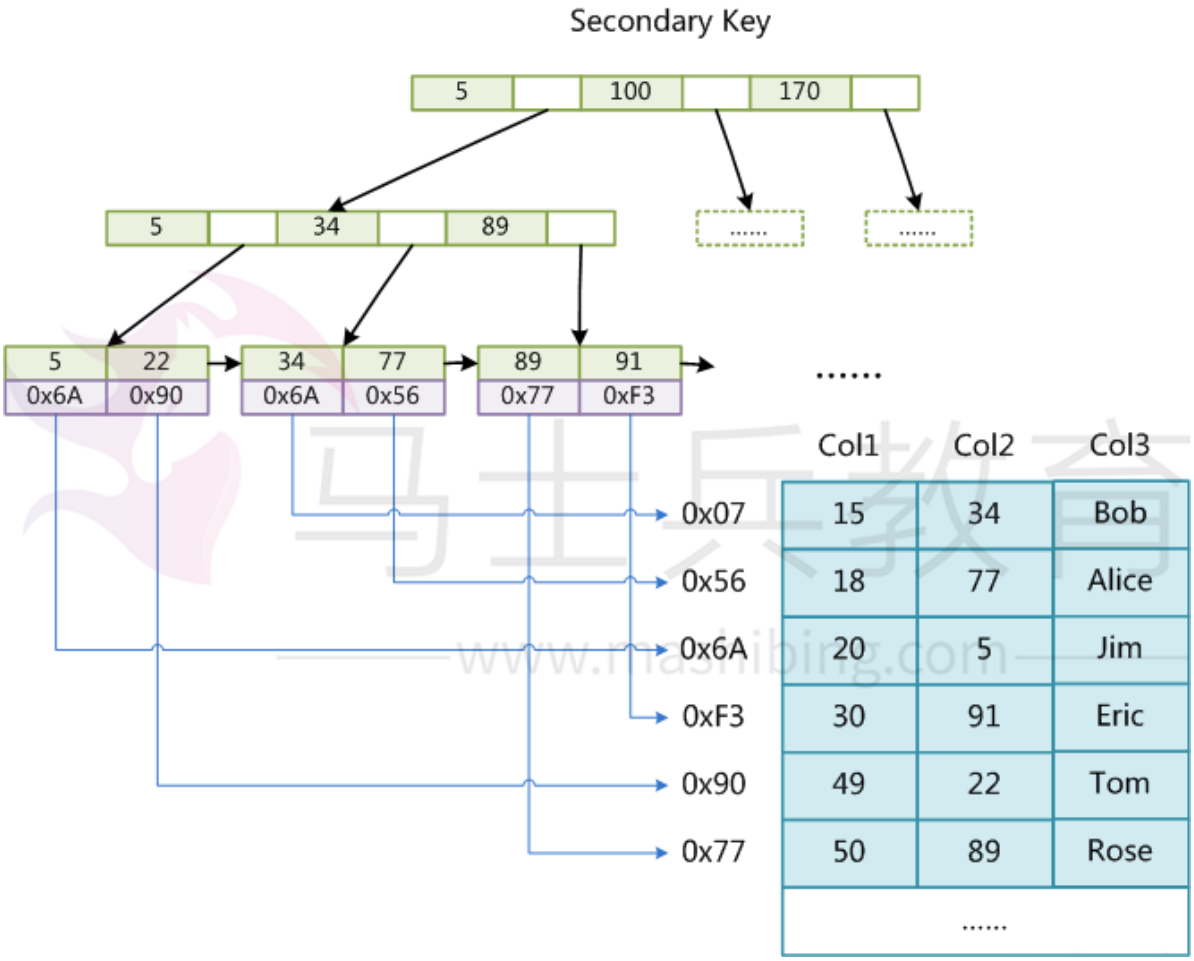


图9

同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

InnoDB索引实现

虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。

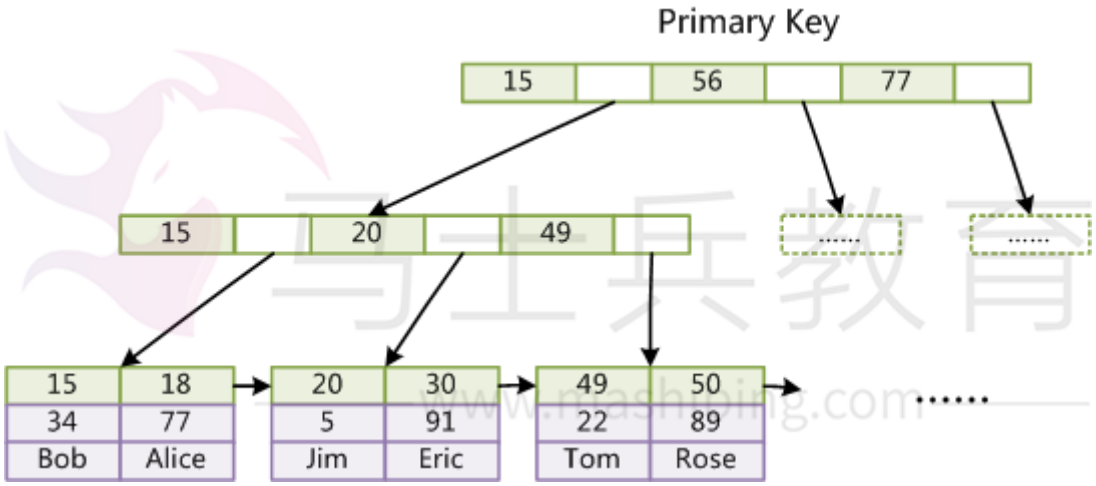


图10

图10是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，图11为定义在Col3上的一个辅助索引：

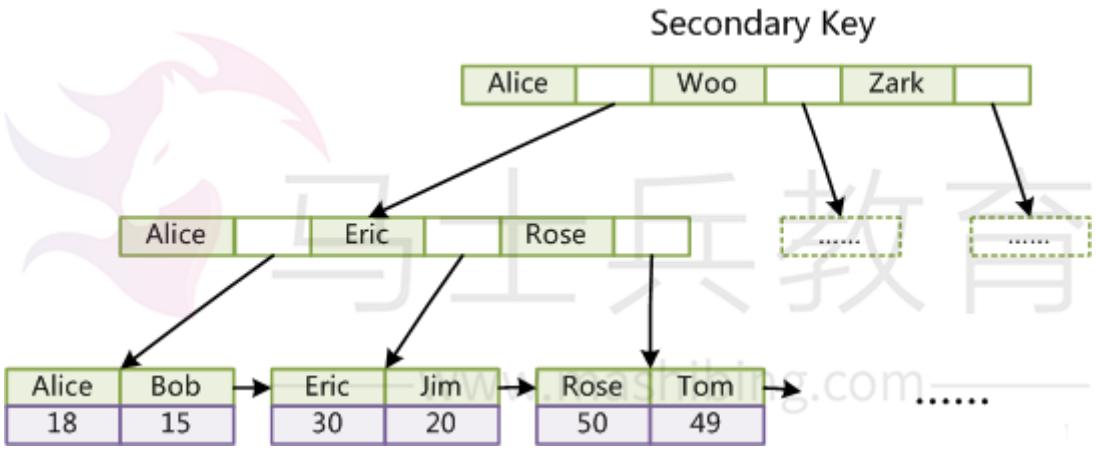


图11

这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

下一章将具体讨论这些与索引有关的优化策略。

索引使用策略及优化

MySQL的优化主要分为结构优化（Scheme optimization）和查询优化（Query optimization）。本章讨论的高性能索引策略主要属于结构优化范畴。本章的内容完全基于上文的理论基础，实际上一旦理解了索引背后的机制，那么选择高性能的策略就变成了纯粹的推理，并且可以理解这些策略背后的逻辑。

示例数据库

为了讨论索引策略，需要一个数据量不算小的数据库作为示例。本文选用MySQL官方文档中提供的示例数据库之一：employees。这个数据库关系复杂度适中，且数据量较大。下图是这个数据库的E-R关系图（引用自MySQL官方手册）：

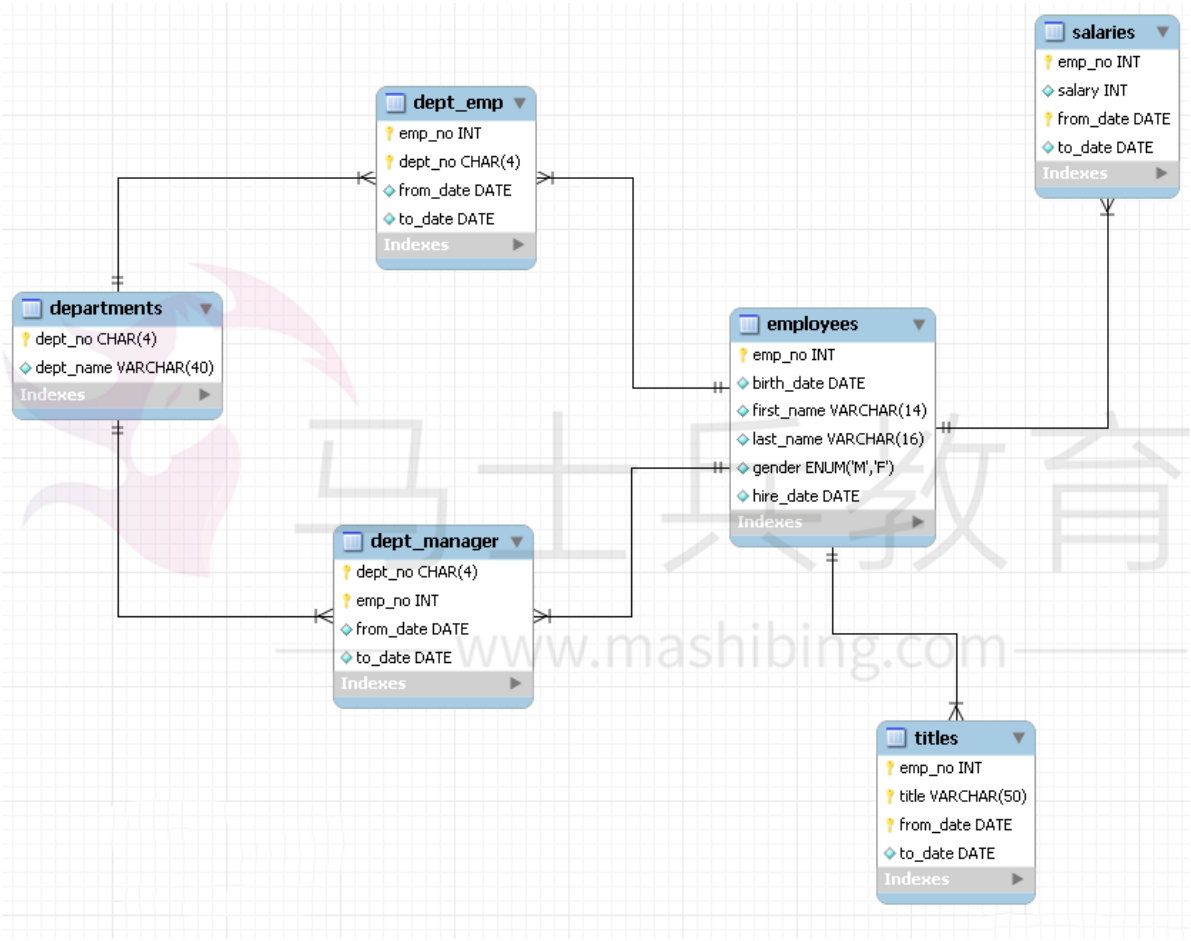


图12

MySQL官方文档中关于此数据库的页面为http://dev.mysql.com/doc/employee/en/employee.html。里面详细介绍了此数据库，并提供了下载地址和导入方法，如果有兴趣导入此数据库到自己的MySQL可以参考文中内容。

最左前缀原理与相关优化

高效使用索引的首要条件是知道什么样的查询会使用到索引，这个问题和B+Tree中的“最左前缀原理”有关，下面通过例子说明最左前缀原理。

这里先说一下联合索引的概念。在上文中，我们都是假设索引只引用了单个的列，实际上，MySQL中的索引可以以一定顺序引用多个列，这种索引叫做联合索引，一般的，一个联合索引是一个有序元组<a1, a2, ..., an>，其中各个元素均为数据表的一列，实际上要严格定义索引需要用到关系代数，但是这里我不想讨论太多关系代数的话题，因为那样会显得很枯燥，所以这里就不再做严格定义。另外，单列索引可以看成联合索引元素数为1的特例。

以employees.titles表为例，下面先查看其上都有哪些索引：

```
SHOW INDEX FROM employees.titles;+--------+------------+----------+-------------
+-------------+----------+-------------+------+------------+| Table  |
Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality |
Null | Index_type |+--------+------------+----------+--------------+-------------
+----------+-------------+------+------------+| titles |          0 | PRIMARY  |
           1 | emp_no      | A         |        NULL |      | BTREE      ||
titles |          0 | PRIMARY  |            2 | title       | A         |
NULL |      | BTREE      || titles |          0 | PRIMARY  |            3 |
from_date   | A         |      443308 |      | BTREE      || titles |          1
| emp_no   |            1 | emp_no      | A         |      443308 |      | BTREE
     |+--------+------------+----------+--------------+-------------+-----------
+-------------+------+------------+
```

从结果中可以到titles表的主索引为<emp_no, title, from_date>，还有一个辅助索引<emp_no>。为了避免多个索引使事情变复杂（MySQL的SQL优化器在多索引时行为比较复杂），这里我们将辅助索引drop掉：

```
ALTER TABLE employees.titles DROP INDEX emp_no;
```

这样就可以专心分析索引PRIMARY的行为了。

情况一：全列匹配。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title='Senior
Engineer' AND from_date='1986-06-26';+----+-------------+--------+-------+-------
--------+---------+---------+-------------------+------+-------+| id |
select_type | table  | type  | possible_keys | key     | key_len | ref
    | rows | Extra |+----+-------------+--------+-------+---------------+--------
-+---------+-------------------+------+-------+|  1 | SIMPLE      | titles |
const | PRIMARY       | PRIMARY | 59      | const,const,const |    1 |       |+--
--+-------------+--------+-------+---------------+---------+---------+-----------
--------+------+-------+
```

很明显，当按照索引中所有列进行精确匹配（这里精确匹配指“=”或“IN”匹配）时，索引可以被用到。这里有一点需要注意，理论上索引对顺序是敏感的，但是由于MySQL的查询优化器会自动调整where子句的条件顺序以使用适合的索引，例如我们将where中的条件顺序颠倒：

```
EXPLAIN SELECT * FROM employees.titles WHERE from_date='1986-06-26' AND
emp_no='10001' AND title='Senior Engineer';+----+-------------+--------+-------+-
--------------+---------+---------+-------------------+------+-------+| id |
select_type | table  | type  | possible_keys | key     | key_len | ref
    | rows | Extra |+----+-------------+--------+-------+---------------+--------
-+---------+-------------------+------+-------+|  1 | SIMPLE      | titles |
const | PRIMARY       | PRIMARY | 59      | const,const,const |    1 |       |+--
--+-------------+--------+-------+---------------+---------+---------+-----------
--------+------+-------+
```

效果是一样的。

情况二：最左前缀匹配。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001';+----+-------------+-
-------+------+---------------+---------+---------+-------+------+-------+| id |
select_type | table  | type | possible_keys | key     | key_len | ref   | rows |
Extra |+----+-------------+-------+------+---------------+---------+---------+---
-----+------+-------+|  1 | SIMPLE      | titles | ref  | PRIMARY       | PRIMARY
| 4       | const |    1 |       |+----+-------------+-------+------+-----------
----+---------+---------+-------+------+-------+
```

当查询条件精确匹配索引的左边连续一个或几个列时，如<emp_no>或<emp_no, title>，所以可以被用到，但是只能用到一部分，即条件所组成的最左前缀。上面的查询从分析结果看用到了PRIMARY索引，但是key_len为4，说明只用到了索引的第一列前缀。

情况三：查询条件用到了索引中列的精确匹配，但是中间某个条件未提供。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND from_date='1986-
06-26';+----+-------------+-------+------+---------------+---------+---------+--
-----+------+-------------+| id | select_type | table  | type | possible_keys |
key     | key_len | ref   | rows | Extra       |+----+-------------+-------+----
--+---------------+---------+---------+-------+------+-------------+|  1 | SIMPLE
     | titles | ref  | PRIMARY       | PRIMARY | 4       | const |    1 | Using
where |+----+-------------+-------+------+---------------+---------+---------+--
-----+------+-------------+
```

此时索引使用情况和情况二相同，因为title未提供，所以查询只用到了索引的第一列，而后面的from_date虽然也在索引中，但是由于title不存在而无法和左前缀连接，因此需要对结果进行扫描过滤from_date（这里由于emp_no唯一，所以不存在扫描）。如果想让from_date也使用索引而不是where过滤，可以增加一个辅助索引<emp_no, from_date>，此时上面的查询会使用这个索引。除此之外，还可以使用一种称之为“隔离列”的优化方法，将emp_no与from_date之间的“坑”填上。

首先我们看下title一共有几种不同的值：

```
SELECT DISTINCT(title) FROM employees.titles;+--------------------+| title
       |+--------------------+| Senior Engineer    || Staff              ||
Engineer           || Senior Staff       || Assistant Engineer || Technique
Leader   || Manager            |+--------------------+
```

只有7种。在这种成为“坑”的列值比较少的情况下，可以考虑用“IN”来填补这个“坑”从而形成最左前缀：

```
EXPLAIN SELECT * FROM employees.titlesWHERE emp_no='10001'AND title IN ('Senior
Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique
Leader', 'Manager')AND from_date='1986-06-26';+----+-------------+-------+------
-+---------------+---------+---------+------+------+-------------+| id |
select_type | table  | type  | possible_keys | key     | key_len | ref  | rows |
Extra       |+----+-------------+-------+-------+---------------+---------+-----
----+------+------+-------------+|  1 | SIMPLE      | titles | range | PRIMARY
    | PRIMARY | 59      | NULL |    7 | Using where |+----+-------------+--------
+-------+---------------+---------+---------+------+------+-------------+
```

这次key_len为59，说明索引被用全了，但是从type和rows看出IN实际上执行了一个range查询，这里检查了7个key。看下两种查询的性能比较：

```
SHOW PROFILES;+----------+------------+-----------------------------------------
-----------------------------------+| Query_ID | Duration   | Query
                                                                  |+----------+--------
----+-------------------------------------------------------------------------------
---+|       10 | 0.00058000 | SELECT * FROM employees.titles WHERE emp_no='10001'
AND from_date='1986-06-26'||       11 | 0.00052500 | SELECT * FROM
employees.titles WHERE emp_no='10001' AND title IN ...          |+----------+----
--------+-------------------------------------------------------------------------
-------+
```

“填坑”后性能提升了一点。如果经过emp_no筛选后余下很多数据，则后者性能优势会更加明显。当然，如果title的值很多，用填坑就不合适了，必须建立辅助索引。

情况四：查询条件没有指定索引第一列。

```
EXPLAIN SELECT * FROM employees.titles WHERE from_date='1986-06-26';+----+-------
------+--------+------+---------------+------+---------+------+--------+---------
----+| id | select_type | table  | type | possible_keys | key  | key_len | ref  |
rows   | Extra       |+----+-------------+--------+------+---------------+------
+---------+------+--------+-------------+|  1 | SIMPLE      | titles | ALL  |
NULL          | NULL | NULL    | NULL | 443308 | Using where |+----+-------------
+--------+------+---------------+------+---------+------+--------+-------------+
```

由于不是最左前缀，索引这样的查询显然用不到索引。

情况五：匹配某列的前缀字符串。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title LIKE
'Senior%';+----+-------------+--------+-------+---------------+---------+--------
-+------+------+-------------+| id | select_type | table  | type  | possible_keys
| key     | key_len | ref  | rows | Extra       |+----+-------------+--------+---
----+---------------+---------+---------+------+------+-------------+|  1 |
SIMPLE      | titles | range | PRIMARY       | PRIMARY | 56      | NULL |    1 |
Using where |+----+-------------+--------+-------+---------------+---------+-----
----+------+------+-------------+
```

此时可以用到索引，~~但是如果通配符不是只出现在末尾，则无法使用索引。~~（原文表述有误，如果通配符%不出现在开头，则可以用到索引，但根据具体情况不同可能只会用其中一个前缀）

情况六：范围查询。

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no < '10010' and title='Senior
Engineer';+----+-------------+--------+-------+---------------+---------+--------
-+------+------+-------------+| id | select_type | table  | type  | possible_keys
| key     | key_len | ref  | rows | Extra       |+----+-------------+--------+---
----+---------------+---------+---------+------+------+-------------+|  1 |
SIMPLE      | titles | range | PRIMARY       | PRIMARY | 4       | NULL |   16 |
Using where |+----+-------------+--------+-------+---------------+---------+-----
----+------+------+-------------+
```

范围列可以用到索引（必须是最左前缀），但是范围列后面的列无法用到索引。同时，索引最多用于一个范围列，因此如果查询条件中有两个范围列则无法全用到索引。

```
EXPLAIN SELECT * FROM employees.titlesWHERE emp_no < '10010'AND title='Senior
Engineer'AND from_date BETWEEN '1986-01-01' AND '1986-12-31';+----+-------------
+--------+-------+---------------+---------+---------+------+------+-----------
+| id | select_type | table  | type  | possible_keys | key     | key_len | ref  |
rows | Extra       |+----+-------------+--------+-------+---------------+--------
-+---------+------+------+-------------+|  1 | SIMPLE      | titles | range |
PRIMARY       | PRIMARY | 4       | NULL |   16 | Using where |+----+-----------
-+--------+-------+---------------+---------+---------+------+------+-----------
-+
```

可以看到索引对第二个范围索引无能为力。这里特别要说明MySQL一个有意思的地方，那就是仅用explain可能无法区分范围索引和多值匹配，因为在type中这两者都显示为range。同时，用了“between”并不意味着就是范围查询，例如下面的查询：

```
EXPLAIN SELECT * FROM employees.titlesWHERE emp_no BETWEEN '10001' AND '10010'AND
title='Senior Engineer'AND from_date BETWEEN '1986-01-01' AND '1986-12-31';+----
+-------------+--------+-------+---------------+---------+---------+------+------
+-------------+| id | select_type | table  | type  | possible_keys | key     |
key_len | ref  | rows | Extra       |+----+-------------+--------+-------+-------
--------+---------+---------+------+------+-------------+|  1 | SIMPLE      |
titles | range | PRIMARY       | PRIMARY | 59      | NULL |   16 | Using where
|+----+-------------+--------+-------+---------------+---------+---------+------
+------+-------------+
```

看起来是用了两个范围查询，但作用于emp_no上的“BETWEEN”实际上相当于“IN”，也就是说emp_no实际是多值精确匹配。可以看到这个查询用到了索引全部三个列。因此在MySQL中要谨慎地区分多值匹配和范围匹配，否则会对MySQL的行为产生困惑。

情况七：查询条件中含有函数或表达式。

很不幸，如果查询条件中含有函数或表达式，则MySQL不会为这列使用索引（虽然某些在数学意义上可以使用）。例如：

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND left(title,
6)='Senior';+----+-------------+--------+------+---------------+---------+-------
--+-------+------+-------------+| id | select_type | table  | type |
possible_keys | key     | key_len | ref   | rows | Extra       |+----+-----------
--+--------+------+---------------+---------+---------+-------+------+-----------
--+|  1 | SIMPLE      | titles | ref  | PRIMARY       | PRIMARY | 4       | const
|    1 | Using where |+----+-------------+--------+------+---------------+-------
--+---------+-------+------+-------------+
```

虽然这个查询和情况五中功能相同，但是由于使用了函数left，则无法为title列应用索引，而情况五中用LIKE则可以。再如：

```
EXPLAIN SELECT * FROM employees.titles WHERE emp_no - 1='10000';+----+-----------
--+--------+------+---------------+------+---------+------+--------+-------------
+| id | select_type | table  | type | possible_keys | key  | key_len | ref  |
rows   | Extra       |+----+-------------+--------+------+---------------+------
+---------+------+--------+-------------+|  1 | SIMPLE      | titles | ALL  |
NULL          | NULL | NULL    | NULL | 443308 | Using where |+----+-------------
+--------+------+---------------+------+---------+------+--------+-------------+
```

显然这个查询等价于查询emp_no为10001的函数，但是由于查询条件是一个表达式，MySQL无法为其使用索引。看来MySQL还没有智能到自动优化常量表达式的程度，因此在写查询语句时尽量避免表达式出现在查询中，而是先手工私下代数运算，转换为无表达式的查询语句。

索引选择性与前缀索引

既然索引可以加快查询速度，那么是不是只要是查询语句需要，就建上索引？答案是否定的。因为索引虽然加快了查询速度，但索引也是有代价的：索引文件本身要消耗存储空间，同时索引会加重插入、删除和修改记录时的负担，另外，MySQL在运行时也要消耗资源维护索引，因此索引并不是越多越好。一般两种情况下不建议建索引。

第一种情况是表记录比较少，例如一两千条甚至只有几百条记录的表，没必要建索引，让查询做全表扫描就好了。至于多少条记录才算多，这个个人有个人的看法，我个人的经验是以2000作为分界线，记录数不超过 2000可以考虑不建索引，超过2000条可以酌情考虑索引。

另一种不建议建索引的情况是索引的选择性较低。所谓索引的选择性（Selectivity），是指不重复的索引值（也叫基数，Cardinality）与表记录数（#T）的比值：

Index Selectivity = Cardinality / #T

显然选择性的取值范围为(0, 1]，选择性越高的索引价值越大，这是由B+Tree的性质决定的。例如，上文用到的employees.titles表，如果title字段经常被单独查询，是否需要建索引，我们看一下它的选择性：

```
SELECT count(DISTINCT(title))/count(*) AS Selectivity FROM employees.titles;+----
---------+| Selectivity |+-------------+|      0.0000 |+-------------+
```

title的选择性不足0.0001（精确值为0.00001579），所以实在没有什么必要为其单独建索引。

有一种与索引选择性有关的索引优化策略叫做前缀索引，就是用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。下面以employees.employees表为例介绍前缀索引的选择和使用。

从图12可以看到employees表只有一个索引<emp_no>，那么如果我们想按名字搜索一个人，就只能全表扫描了：

```
EXPLAIN SELECT * FROM employees.employees WHERE first_name='Eric' AND
last_name='Anido';+----+-------------+-----------+------+---------------+------+-
--------+------+--------+-------------+| id | select_type | table     | type |
possible_keys | key  | key_len | ref  | rows   | Extra       |+----+-------------
+-----------+------+---------------+------+---------+------+--------+------------
-+|  1 | SIMPLE      | employees | ALL  | NULL          | NULL | NULL    | NULL |
300024 | Using where |+----+-------------+-----------+------+---------------+----
--+---------+------+--------+-------------+
```

如果频繁按名字搜索员工，这样显然效率很低，因此我们可以考虑建索引。有两种选择，建<first_name>或<first_name, last_name>，看下两个索引的选择性：

```
SELECT count(DISTINCT(first_name))/count(*) AS Selectivity FROM
employees.employees;+-------------+| Selectivity |+-------------+|      0.0042
|+-------------+SELECT count(DISTINCT(concat(first_name, last_name)))/count(*) AS
Selectivity FROM employees.employees;+-------------+| Selectivity |+-------------
+|      0.9313 |+-------------+
```

<first_name>显然选择性太低，<first_name, last_name>选择性很好，但是first_name和last_name加起来长度为30，有没有兼顾长度和选择性的办法？可以考虑用first_name和last_name的前几个字符建立索引，例如<first_name, left(last_name, 3)>，看看其选择性：

```
SELECT count(DISTINCT(concat(first_name, left(last_name, 3))))/count(*) AS
Selectivity FROM employees.employees;+-------------+| Selectivity |+-------------
+|      0.7879 |+-------------+
```

选择性还不错，但离0.9313还是有点距离，那么把last_name前缀加到4：

```
SELECT count(DISTINCT(concat(first_name, left(last_name, 4))))/count(*) AS
Selectivity FROM employees.employees;+-------------+| Selectivity |+-------------
+|      0.9007 |+-------------+
```

这时选择性已经很理想了，而这个索引的长度只有18，比<first_name, last_name>短了接近一半，我们把这个前缀索引 建上：

```
ALTER TABLE employees.employeesADD INDEX `first_name_last_name4` (first_name,
last_name(4));
```

此时再执行一遍按名字查询，比较分析一下与建索引前的结果：

```
SHOW PROFILES;+----------+------------+-----------------------------------------
--------------------------------------+| Query_ID | Duration   | Query
                                      |+----------+----
--------+-------------------------------------------------------------------------
---------+|       87 | 0.11941700 | SELECT * FROM employees.employees WHERE
first_name='Eric' AND last_name='Anido' ||       90 | 0.00092400 | SELECT * FROM
employees.employees WHERE first_name='Eric' AND last_name='Anido' |+----------+--
----------+-------------------------------------------------------------------------
-----------+
```

性能的提升是显著的，查询速度提高了120多倍。

前缀索引兼顾索引大小和查询速度，但是其缺点是不能用于ORDER BY和GROUP BY操作，也不能用于Covering index（即当索引本身包含查询所需全部数据时，不再访问数据文件本身）。

InnoDB的主键选择与插入优化

在使用InnoDB存储引擎时，如果没有特别的需要，请永远使用一个与业务无关的自增字段作为主键。

经常看到有帖子或博客讨论主键选择问题，有人建议使用业务无关的自增主键，有人觉得没有必要，完全可以使用如学号或身份证号这种唯一字段作为主键。不论支持哪种论点，大多数论据都是业务层面的。如果从数据库索引优化角度看，使用InnoDB引擎而不使用自增主键绝对是一个糟糕的主意。

上文讨论过InnoDB的索引实现，InnoDB使用聚集索引，数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放，因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）。

如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。如下图所示：

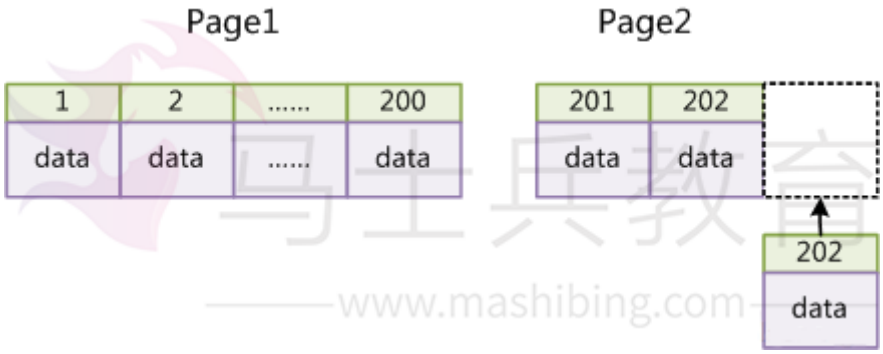


图13

这样就会形成一个紧凑的索引结构，近似顺序填满。由于每次插入时也不需要移动已有数据，因此效率很高，也不会增加很多开销在维护索引上。

如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置：

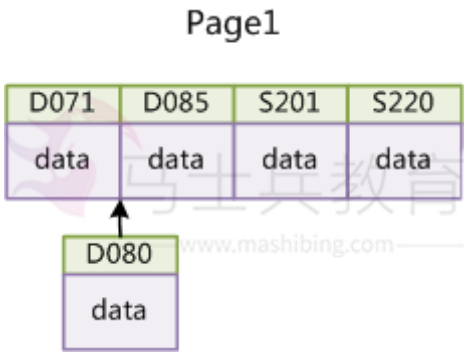


图14

此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

因此，只要可以，请尽量在InnoDB上采用自增字段做主键。

后记

其实数据库索引调优是一项技术活，不能仅仅靠理论，因为实际情况千变万化，而且MySQL本身存在很复杂的机制，如查询优化策略和各种引擎的实现差异等都会使情况变得更加复杂。但同时这些理论是索引调优的基础，只有在明白理论的基础上，才能对调优策略进行合理推断并了解其背后的机制，然后结合实践中不断的实验和摸索，从而真正达到高效使用MySQL索引的目的。

另外，MySQL索引及其优化涵盖范围非常广，本文只是涉及到其中一部分。如与排序（ORDER BY）相关的索引优化及覆盖索引（Covering index）的话题本文并未涉及，同时除B-Tree索引外MySQL还根据不同引擎支持的哈希索引、全文索引等等本文也并未涉及。如果有机会，希望再对本文未涉及的部分进行补充吧。

mysql的锁--行锁，表锁，乐观锁，悲观锁

一 引言--为什么mysql提供了锁

最近看到了mysql有行锁和表锁两个概念，越想越疑惑。为什么mysql要提供锁机制，而且这种机制不是一个摆设，还有很多人在用。在现代数据库里几乎有事务机制，acid的机制应该能解决并发调度的问题了，为什么还要主动加锁呢?

后来看到一篇，“防止更新丢失，并不能单靠数据库事务控制器来解决，需要应用程序对要更新的数据**加必要的锁**来解决”。瞬间，世界观都崩塌了。非常不敢相信，于是自己写了代码检验一下。

数据库表是这样的。用count字段来做100次累加。

id	count
1	0
2	0

为了保证实验的科学性，先确认了数据库是InnoDB的，这样才有事务机制；也确认了隔离性级别

```
mysql> SELECT @@session.tx_isolation;
+-------------------------+
| @@session.tx_isolation  |
+-------------------------+
| REPEATABLE-READ         |
+-------------------------+
```

定义一个任务，读count值--程序count++--写数据库

```
public class LostUpdate implements Runnable{
    private CountDownLatch countDown;
    public LostUpdate(CountDownLatch countDown){
        this.countDown = countDown;
    }

    @Override
    public void run() {
        Connection conn=null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=UTF-8",
                    "root", "123");
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        try {
            conn.setAutoCommit(false);
            //不加锁的情况
            PreparedStatement ps =conn.prepareStatement("select * from LostUpdate
where id =1");
            //加锁的情况
            //PreparedStatement ps =conn.prepareStatement("select * from
LostUpdate where id =1 for update");
            ResultSet rs=ps.executeQuery();
            int count = 0;
            while(rs.next()){
                count= rs.getInt("count");
```

```
            }

            count++;
            ps =conn.prepareStatement("update LostUpdate set count=? where id
=1");
            ps.setInt(1, count);
            ps.executeUpdate();

            conn.commit();
        } catch (Exception e) {
            try {
                conn.rollback();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        }
        //表示一次任务完成
        countDown.countDown();
    }
}
```

主线程下创建子线程，模拟多线程环境

```
public class TestLock {
    public static void main(String[] args) throws InterruptedException {
        //创建线程池，里面有10个线程，共执行100次+1操作
        final int THREAD_COUNT=10;
        final int RUN_TIME=100;

        ExecutorService threadPool=Executors.newFixedThreadPool(THREAD_COUNT);
        //用CountDownLatch保证主线程等待所有任务完成
        CountDownLatch count=new CountDownLatch(RUN_TIME);

        for(int i=0;i<RUN_TIME;i++)
            threadPool.execute(new LostUpdate(count));

        threadPool.shutdown();
        count.await();
        //提示所有任务执行完
        System.out.println("finish");
    }
}
```

运行结果是：

id	count
1	10
2	0

大概解释一下程序，就是创建了一个线程池，里面10个线程，执行100次任务。每个任务就是 读count值--程序count++--写数据库，经典的银行存款（丢失修改）问题。事实胜于雄辩，结论就是上面的橙色字，解决丢失修改不能靠事务，要加必要的锁，所以数据库提供的锁不是个摆设。

二 数据库事务机制

为了找到问题的根源，为了拯救我崩溃的世界观，我又去回顾了数据库事务的知识。

数据库的acid属性

- 原性性（Actomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
- 一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以操持完整性；事务结束时，所有的内部数据结构（如B树索引或双向链表）也都必须是正确的。
- 隔离性（Isolation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。
- 持久性（Durable）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

说好的一致性呢，童话里都是骗人的！！

事务并发调度的问题

1. 脏读（dirty read）：A事务读取B事务尚未提交的更改数据，并在这个数据基础上操作。如果B事务回滚，那么A事务读到的数据根本不是合法的，称为脏读。在oracle中，由于有version控制，不会出现脏读。
2. 不可重复读（unrepeatable read）：A事务读取了B事务已经提交的更改（或删除）数据。比如A事务第一次读取数据，然后B事务更改该数据并提交，A事务再次读取数据，两次读取的数据不一样。
3. 幻读（phantom read）：A事务读取了B事务已经提交的新增数据。注意和不可重复读的区别，这里是新增，不可重复读是更改（或删除）。这两种情况对策是不一样的，对于不可重复读，只需要采取行级锁防止该记录数据被更改或删除，然而对于幻读必须加表级锁，防止在这个表中新增一条数据。
4. 第一类丢失更新：A事务撤销时，把已提交的B事务的数据覆盖掉。
5. 第二类丢失更新：A事务提交时，把已提交的B事务的数据覆盖掉。

三级封锁协议

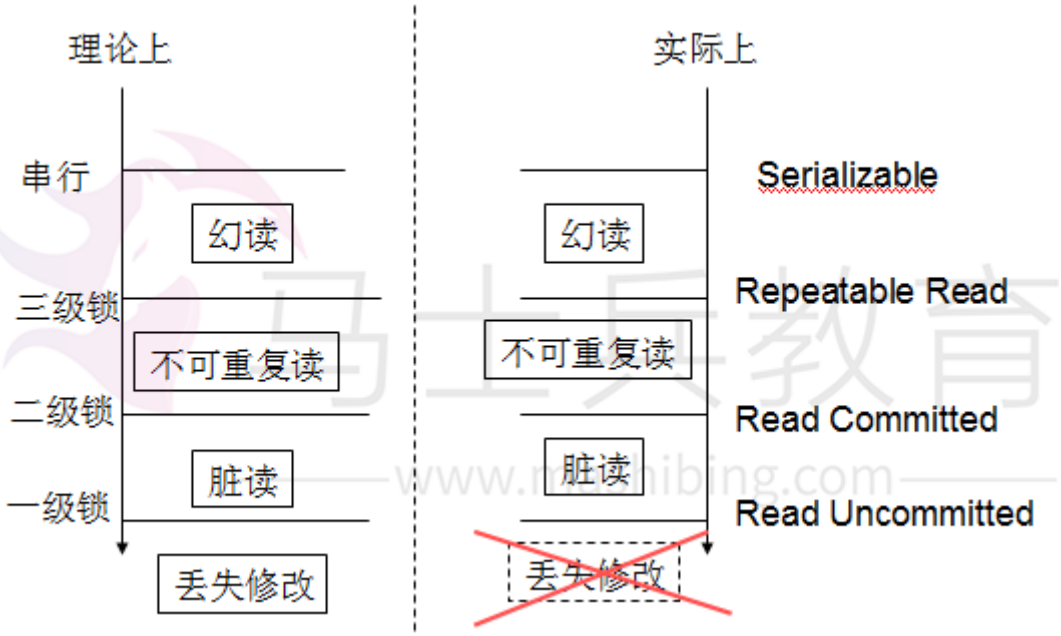
1. 一级封锁协议：事务T中如果对数据R有写操作，必须在这个事务中对R的第一次读操作前对它加X锁，直到事务结束才释放。事务结束包括正常结束（COMMIT）和非正常结束（ROLLBACK）。
2. 二级封锁协议：一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，读完后方可释放S锁。
3. 三级封锁协议：一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，直到事务结束才释放。

可见，三级锁操作一个比一个厉害（满足高级锁则一定满足低级锁）。但有个非常致命的地方，一级锁协议就要在第一次读加x锁，直到事务结束。几乎就要在整个事务加写锁了，效率非常低。三级封锁协议只是一个理论上的东西，实际数据库常用另一套方法来解决事务并发问题。

隔离性级别

mysql用意向锁（另一种机制）来解决事务并发问题，为了区别封锁协议，弄了一个新概念隔离性级别：包括Read Uncommitted、Read Committed、Repeatable Read、Serializable。mysql 一般默认Repeatable Read。

隔离级别/读数据一致性及允许的并发副作用	读数据一致性	脏读	不可重复读	幻读
未提交读（Read uncommitted）	最低级别，只能保证不读取物理上损坏的数据	是	是	是
已提交度（Read committed）	语句级	否	是	是
可重复读（Repeatable read）	事务级	否	否	是
可序列化（Serializable）	最高级别，事务级	否	否	否



终于发现自己为什么会误会事务能解决丢失修改了。至于为什么隔离性级别不解决丢失修改，我猜是有更好的解决方案吧。

总结一下，repeatable read能解决脏读和不可重复读，但不嗯呢该解决丢失修改。

三 mysql的行锁和表锁

说了那么久，终于入正题了，先来说说什么是行锁和表锁。

- 表级锁：每次操作锁住整张表。开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低；
- 行级锁：每次操作锁住一行数据。开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高；
- 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。没弄懂，有空再看。？

1 MyISAM的锁

稍微提一下MyISAM，只说和InnoDB不同的。

a. MyISAM只有表锁，锁又分为读锁和写锁。

当前锁模式/是否兼容/请求锁模式	None	读锁	写锁
读锁	是	是	否
写锁	是	否	否

b. 没有事务，不用考虑并发问题，世界和平~

c. 由于锁的粒度太大，所以当该表写并发量较高时，要等待的查询就会很多了。

2 InnoDB的行锁和表锁

没有特定的语法。mysql的行锁是通过索引体现的。

如果where条件中只用到索引项，则加的是行锁；否则加的是表锁。比如说主键索引，唯一索引和聚簇索引等。如果sql的where是全表扫描的，想加行锁也爱莫能助。

行锁和表锁对我们编程有什么影响，要在where中尽量只用索引项，否则就会触发表锁。另一个可能是，我们发疯了地想优化查询，但where子句中就是有非索引项，于是我们自己写连接？

行锁和表锁各适合怎么样的应用，待求证？。

3 读锁和写锁

InnoDB用意向锁？实现隔离性级别，原理未名，贴张图：

当前锁模式/是否兼容/请求锁模式	X	IX	S	IS
X	冲突	冲突	冲突	冲突
IX	冲突	兼容	冲突	兼容
S	冲突	冲突	兼容	兼容
IS	冲突	兼容	兼容	兼容

马士兵教育 www.mashibing.com

回想锁协议，对什么操作加什么锁是一个问题，加锁加到什么时候有是一个问题。锁协议里常常会看到“加锁直到事务结束”的烦心字样。而在InnoDB中，select,insert,update,delete等语句执行时都会自动加解锁。select的锁一般执行完就释放了，修改操作的X锁会持有到事务结束，效率高很多。至于详细的加锁原理，见这里，搜“InnoDB存储引擎中不同SQL在不同隔离级别下锁比较”

mysql也给用户提供了加锁的机会，只要在sql后加LOCK IN SHARE MODE 或FOR UPDATE

共享锁（S）：SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE
排他锁（X）：SELECT * FROM table_name WHERE ... FOR UPDATE

值得注意的是，自己加的锁没有释放锁的语句，所以锁会持有到事务结束。

mysql 还提供了LOCK TABLES，UNLOCK TABLES，用于加表锁，怎么用还不太清楚?

4 考察加锁的情况

加了读锁还是写锁，加了行锁还是表锁，说什么时候释放，可以从原理上分析。但刚开始时我不太懂原理，于是又写了个程序。

```
public class ForUpdate1  implements Runnable{
    private CountDownLatch countDown;
    public ForUpdate1(CountDownLatch countDown){
        this.countDown = countDown;
    }
    @Override
    public void run() {
        Connection conn=null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=UTF-8",
                    "root", "123");
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        try {
            conn.setAutoCommit(false);
            /*PreparedStatement ps =conn.prepareStatement("select * from
LostUpdate where id =1 for update");
            ps.executeQuery();*/
            PreparedStatement ps =conn.prepareStatement("update LostUpdate set
count =1 where id =1");
            ps.executeUpdate();
            Thread.sleep(10000);

            conn.commit();
            System.out.println("test 1 finish");
            countDown.countDown();
        } catch (Exception e) {
            try {
                conn.rollback();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
```

```
        }
    }
}
```

```
public class ForUpdate2  implements Runnable{
    private CountDownLatch countDown;
    public ForUpdate2(CountDownLatch countDown){
        this.countDown = countDown;
    }

    @Override
    public void run() {
        Connection conn=null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=UTF-8",
                    "root", "123");
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        try {
            Thread.sleep(2000);
            conn.setAutoCommit(false);
            PreparedStatement ps =conn.prepareStatement("select * from LostUpdate
where id =1 for update");
            ps.executeQuery();
            /*PreparedStatement ps =conn.prepareStatement("update LostUpdate set
count =1 where id =1");
            ps.executeUpdate();*/

            conn.commit();
            System.out.println("test 2 finish");
            countDown.countDown();
        } catch (Exception e) {
            try {
                conn.rollback();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        }
    }
}
```

```
public class TestForUpdate {
    public static void main(String[] args) throws InterruptedException {
```

```
        final int THREAD_COUNT=10;

        ExecutorService threadPool=Executors.newFixedThreadPool(THREAD_COUNT);
        CountDownLatch count=new CountDownLatch(2);

        threadPool.execute(new ForUpdate1(count));
        threadPool.execute(new ForUpdate2(count));

        threadPool.shutdown();
        count.await();
        System.out.println("finish");
    }
}
```

只有两个线程，ForUpdate1先执行sql语句之后等10s，ForUpdate2先等待2s再执行sql语句。所以如果ForUpdate1持有锁，而且ForUpdate2等待，输出就应该是test 1 finish->test 2 finish->finish；否则就是test 2 finish->test 1 finish->finish。

这个程序改一下能测试上面说的理论：

- repeatable read能解决脏读和不可重复读
- 比如行锁真的只锁住一行
- s，x，is和ix的关系

判断加锁情况，mysql应该有工具，但没找到?

可以通过检查InnoDB_row_lock状态变量来分析系统上的行锁的争夺情况：
mysql> show status like 'innodb_row_lock%';

```
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| InnoDB_row_lock_current_waits | 0     |
| InnoDB_row_lock_time          | 0     |
| InnoDB_row_lock_time_avg      | 0     |
| InnoDB_row_lock_time_max      | 0     |
| InnoDB_row_lock_waits         | 0     |
+-------------------------------+-------+
```

马士兵教育 www.mashibing.com

如果发现锁争用比较严重，如InnoDB_row_lock_waits和InnoDB_row_lock_time_avg的值比较高，还可以通过设置InnoDB Monitors来进一步观察发生锁冲突的表、数据行等，并分析锁争用的原因。不明觉厉？，看这篇

总结一下这一章，mysql提供了行锁和表锁，我们写语句时应该尽量启动行锁，以提高效率；另一方面，也说了一下读锁和写锁的原理。好了武器（原理）我们都懂了，那就看怎么优化了。

四 解决丢失修改--乐观锁和悲观锁

首先为什么要加锁，加锁就是为了解决丢失修改（也不知道这么说对不对）。如果一个事务中只有一句sql，数据库是可以保证它是并发安全的。丢失修改的特征就是在一个事务中先读P数据，再写P数据，注意是同一个数据（也不知道这么说对不对）。只是自己推理了一下，没有太强的理据。所谓丢失修改，一般是A事务有两个操作，后一个操作依赖于前一个操作，之后后一个操作覆盖了B事务的写操作，可以表示为这样。

session 1	session 2
Pro1	
	Write(P)
Write(P)	

pro1可能是Read(P),Write(P),Read(Q),Write(Q),其中P=2Q，数据库中的冗余导致的关联关系是很常见的。

1. pro1=Read(P)，就是我们结论中的情况；
2. pro1=Write(P)，pro1处会对P加IX锁？，IX锁会直至事务结束，不会丢失修改；
3. pro1=Read(Q)或Write(Q)，虽然语法上回发生这种情况，但既然数据时关联的，那在两个事务中都应该同时操作P，Q。这样就规范到第一种情况。

综上，如果一个事务先读后写**同一份数据**，就可能发生丢失修改，要做一些处理。可以用下面的乐观锁和悲观锁解决。

悲观锁和乐观锁的概念：

悲观锁（Pessimistic Concurrency Control，PCC）：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。至于怎么加锁，加锁的范围也没讲。

乐观锁（Optimistic Concurrency Control，OCC）：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。也没具体指定怎么检查。

就是这么概念，什么都不说清楚。毕竟乐观锁和悲观锁也不仅仅能用在数据库中，也能用在线程中。

悲观的缺陷是不论是页锁还是行锁，加锁的时间可能会很长，这样可能会长时间的限制其他用户的访问，也就是说悲观锁的并发访问性不好。

乐观锁不能解决脏读，加锁的时间要比悲观锁短（只是在执行sql时加了基本的锁保证隔离性级别），乐观锁可以用较大的锁粒度获得较好的并发访问性能。但是如果第二个用户恰好在第一个用户提交更改之前读取了该对象，那么当他完成了自己的更改进行提交时，数据库就会发现该对象已经变化了，这样，第二个用户不得不重新读取该对象并作出更改。

可见，乐观锁更适合解决冲突概率极小的情况；而悲观锁则适合解决并发竞争激烈的情况，尽量用行锁，缩小加锁粒度，以提高并发处理能力，即便加行锁的时间比加表锁的要长。

悲观锁的例子

并没有人说悲观锁要怎么加锁，加锁的范围如何。这里仅仅提供一种解决丢失修改的悲观锁例子。

丢失修改我们用第一章讲到的累积100次的例子。综合前面讲到的结论，丢失修改的特征就是在一个事务中先读P数据，再写P数据。而且一级锁协议能解决丢失修改，所以如果事务A 中写P，我们只要在A中第一次读P前加X锁。做法在第一章程序中有：

```
//把
PreparedStatement ps =conn.prepareStatement("select * from LostUpdate where id
=1");
//换成
PreparedStatement ps =conn.prepareStatement("select * from LostUpdate where id =1
for update");
```

乐观锁的例子

乐观锁也没有指定怎么检测并发冲突，下面是常见的两种做法：

1. 使用数据版本（Version）。在P数据上（通常每一行）加version字段(int)，A事务在读数据P 时同时读出版本号，在修改数据前检测最新版本号是否等于先前取出的版本号，如果是，则修改，同时把版本号+1；否则要么回滚，要么重新执行事务。另外，数据P的所有修改操作都要把版本号+1。有一个非常重要的点，版本号是用来查看被读的变量有无变化，而不是针对被写的变量，作用是防止被依赖的变量有修改。
2. 使用时间戳(TimeStamp)。做法类似于1中。

下面写两个例子，背景还是一开始的累积100次的丢失修改问题，都是用version解决的。

1 当发生冲突时回滚并抛异常

任务类

```
public class LostUpdateOccDiscard implements Runnable{
    private CountDownLatch countDown;
    public LostUpdateOccDiscard(CountDownLatch countDown){
        this.countDown = countDown;
    }

    @Override
    public void run() {
        Connection conn=null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=UTF-8",
                    "root", "123");
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        try {
            conn.setAutoCommit(false);
            //读的时候一并读出version
            PreparedStatement ps =conn.prepareStatement("select * from LostUpdate
where id =1");
            ResultSet rs=ps.executeQuery();
            int count = 0;
            int version = 0;
            while(rs.next()){
                count= rs.getInt("count");
                version= rs.getInt("version");
```

```
            }

            count++;

            //更新操作，用cas原子操作来更新
            ps =conn.prepareStatement("update LostUpdate set count=?,
version=version+1 where id =1 and version=?");
            ps.setInt(1, count);
            ps.setInt(2, version);
            int result = ps.executeUpdate();

            //检查有无因冲突导致执行失败
            //成功，则commit，完成任务
            if(result>0) {
                conn.commit();
            }
            //失败，回滚，抛异常提醒调用者出现冲突。
            else{
                conn.rollback();
                throw new Exception("更新count出现冲突");
            }
        } catch (SQLException e) {
            try {
                conn.rollback();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
        //表示一次任务完成
        countDown.countDown();
    }
}
```

主线程，和前面差不多，创建10个线程，执行100个任务。

```
public class TestLockOcc {
    public static void main(String[] args) throws InterruptedException {
        //创建线程池，里面有10个线程，共执行100次+1操作
        final int THREAD_COUNT=10;
        final int RUN_TIME=100;

        ExecutorService threadPool=Executors.newFixedThreadPool(THREAD_COUNT);
        //用CountDownLatch保证主线程等待所有任务完成
        CountDownLatch count=new CountDownLatch(RUN_TIME);

        for(int i=0;i<RUN_TIME;i++)
            threadPool.execute(new LostUpdateOccDiscard(count));

        threadPool.shutdown();
        count.await();
        //提示所有任务执行完
```

```
        System.out.println("finish");
    }
}
```

输出结果：在console里出了一堆异常，看数据库，大概累积了10-12次

不要怀疑，程序没有问题。

a. 对着上面说的version方法的原理，程序也比较好懂。

b. 更新时要用cas（compare and set）的原子操作，一步搞定。而不是先读一次version，比较完再执行依据update。想想也知道后者在多线程有问题。

至于为什么只累积了10-12次，原因是这个累加的并发量是10，就是有10个线程在争夺着修改权。九死一生啊，1个线程commit了，就意味着9个线程要rollback抛异常。

2 当发生冲突时重试，有时我们我们不希望程序里那么多异常

任务类

```
public class LostUpdateOcc implements Runnable{
    private CountDownLatch countDown;
    public LostUpdateOcc(CountDownLatch countDown){
        this.countDown = countDown;
    }

    @Override
    public void run() {
        Connection conn=null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=UTF-8",
                    "root", "123");
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        try {
            int try_times=100;
            int count;
            int version;
            PreparedStatement ps;
            ResultSet rs;

            //把循环条件放在里面if里
            while(try_times>0){
                //开始事务
                try_times--;
                conn.setAutoCommit(false);

                //读操作
                ps=conn.prepareStatement("select * from LostUpdate where id
=1");
```

```
                rs=ps.executeQuery();

                //判断事务执行的条件，首先是能执行，其次是需要执行
                if(rs.next()){
                    count= rs.getInt("count");
                    version= rs.getInt("version");

                    count++;

                    //更新操作，用cas原子操作来更新
                    ps =conn.prepareStatement("update LostUpdate set count=?,
version=version+1 where id =1 and version=?");
                    ps.setInt(1, count);
                    ps.setInt(2, version);
                    int result = ps.executeUpdate();

                    //每次执行完更新操作，检测一次冲突
                    //成功，则继续事务
                    //失败，回滚，睡100ms，避开竞争。结束这次循环，开启新事务。
                    if(result==0) {
                        conn.rollback();
                        Thread.sleep(100);
                        continue;
                    }

                    //事务一路顺风，没遇到冲突，事务提交，跳出while
                    conn.commit();
                    break;
                }
                //作为while条件不成立时的处理，比如该行数据被删除。
                else{
                    conn.rollback();
                    break;
                }
            }
            if(try_times<=0) throw new Exception("冲突重试的此时过多，事务失败");
            System.out.println(try_times);
        } catch (SQLException e) {
            try {
                conn.rollback();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        }catch (Exception e) {
            System.out.println(e.getMessage());
        }

        //表示一次任务完成
        countDown.countDown();
    }
}
```

主线程，和前面差不多，创建10个线程，执行100个任务。

```
public class TestLockOcc {
    public static void main(String[] args) throws InterruptedException {
        //创建线程池，里面有10个线程，共执行100次+1操作
        final int THREAD_COUNT=10;
        final int RUN_TIME=100;

        ExecutorService threadPool=Executors.newFixedThreadPool(THREAD_COUNT);
        //用CountDownLatch保证主线程等待所有任务完成
        CountDownLatch count=new CountDownLatch(RUN_TIME);

        for(int i=0;i<RUN_TIME;i++)
            threadPool.execute(new LostUpdateOcc(count));

        threadPool.shutdown();
        count.await();
        //提示所有任务执行完
        System.out.println("finish");
    }
}
```

任务类里就有比较多要注意的

a. 为了不断的重试，用了一个while。因为while的终止条件一般要读了数据后才知道，所以while只放了try_times，把结束条件放在了里面的if。

b. 在while里的每一次循环就重新起一个事务。因为更新失败我们要回滚的。下一次要重起一个。

c. 这里的事务执行条件，能执行且需要执行。比如id=1的记录被删掉了，那就不能执行了；需要执行，比如程序为了把商品记录status由未上架改为已上架，但发现已经被改了，那就不需要执行。可想而知，在多线程条件每次都要判断的。

d. try_times这个东西还是设置一下。至于设多少，要看并发量。

e. 每次更新，都要检测一次冲突

f. 冲突了，要睡一阵子再重试，避开冲突。怎么设置这个值，我突然想起计网的拥塞控制，说笑的~

顺手做了个小实验，还是执行100次，冲突睡眠100ms，

并发量	2	5	10	20
平均尝试次数	1.32	2.16	4.15	6.52
最多尝试次数	8	22	40	35

总结一下：

乐观锁更适合并发竞争少的情况，最好隔那么3-5分钟才有一次冲突。当并发量为10时就能明显感觉乐观锁更慢；

上面只是一读一写。考虑如果一个事务中有3个写，如果每次写都是九死一生，事务提交比小蝌蚪找妈妈还难，这时就更要考虑是不是要用乐观锁了。

但是，当分布式数据库规模大到一定程度后，又另说了。基于悲观锁的分布式锁在集群大到一定程度后（从几百台扩展到几千台时），性能开销就打得无法接受。所以目前的趋势是大规模的分布式数据库更倾向于用乐观锁来达成external consistency。

Mysql中的MVCC

Mysql到底是怎么实现MVCC的？这个问题无数人都在问，但google中并无答案，本文尝试从Mysql源码中寻找答案。

在Mysql中MVCC是在Innodb存储引擎中得到支持的，Innodb为每行记录都实现了三个隐藏字段：

- 6字节的事务ID（`DB_TRX_ID`）
- 7字节的回滚指针（**DB_ROLL_PTR**）
- 隐藏的ID

6字节的事物ID用来标识该行所述的事务，7字节的回滚指针需要了解下Innodb的事务模型。

1. Innodb的事务相关概念

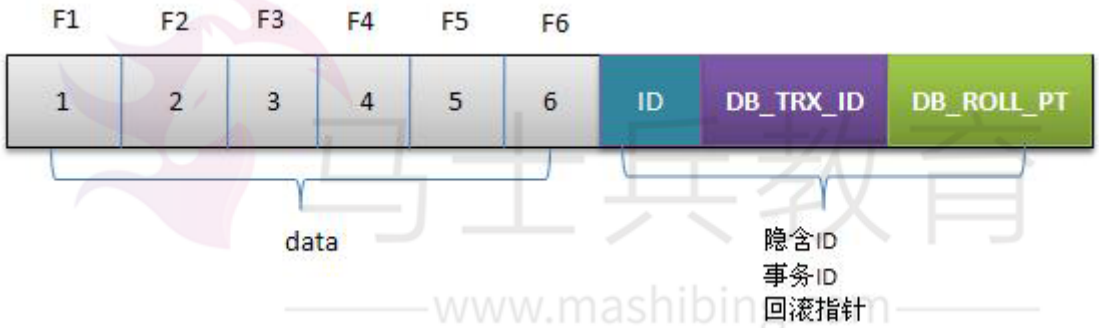
为了支持事务，Innbodb引入了下面几个概念：

- redo log
  redo log就是保存执行的SQL语句到一个指定的Log文件，当Mysql执行recovery时重新执行redo log记录的SQL操作即可。当客户端执行每条SQL（更新语句）时，redo log会被首先写入log buffer；当客户端执行COMMIT命令时，log buffer中的内容会被视情况刷新到磁盘。redo log在磁盘上作为一个独立的文件存在，即Innodb的log文件。
- undo log
  与redo log相反，undo log是为回滚而用，具体内容就是copy事务前的数据库内容（行）到undo buffer，在适合的时间把undo buffer中的内容刷新到磁盘。undo buffer与redo buffer一样，也是环形缓冲，但当缓冲满的时候，undo buffer中的内容会也会被刷新到磁盘；与redo log不同的是，磁盘上不存在单独的undo log文件，所有的undo log均存放在主ibd数据文件中（表空间），即使客户端设置了每表一个数据文件也是如此。
- rollback segment
  回滚段这个概念来自Oracle的事物模型，在Innodb中，undo log被划分为多个段，具体某行的undo log就保存在某个段中，称为回滚段。可以认为undo log和回滚段是同一意思。
- 锁
  Innodb提供了基于行的锁，如果行的数量非常大，则在高并发下锁的数量也可能会比较大，据Innodb文档说，Innodb对锁进行了空间有效优化，即使并发量高也不会导致内存耗尽。
  对行的锁有分两种：排他锁、共享锁。共享锁针对对，排他锁针对写，完全等同读写锁的概念。如果某个事务在更新某行（排他锁），则其他事物无论是读还是写本行都必须等待；如果某个事物读某行（共享锁），则其他读的事物无需等待，而写事物则需等待。通过共享锁，保证了多读之间的无等待性，但是锁的应用又依赖Mysql的事务隔离级别。
- 隔离级别
  隔离级别用来限制事务直接的交互程度，目前有几个工业标准：
  - READ_UNCOMMITTED：脏读
  - READ_COMMITTED：读提交
  - REPEATABLE_READ：重复读
  - SERIALIZABLE：串行化
  Innodb对四种类型都支持，脏读和串行化应用场景不多，读提交、重复读用的比较广泛，后面会介绍其实现方式。

2. 行的更新过程

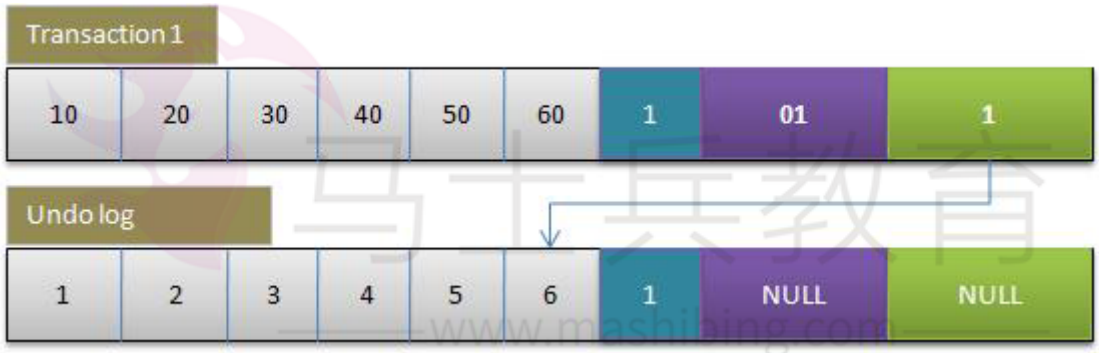
下面演示下事务对某行记录的更新过程：

1. 初始数据行



F1～F6是某行列的名字，1～6是其对应的数据。后面三个隐含字段分别对应该行的事务号和回滚指针，假如这条数据是刚INSERT的，可以认为ID为1，其他两个字段为空。

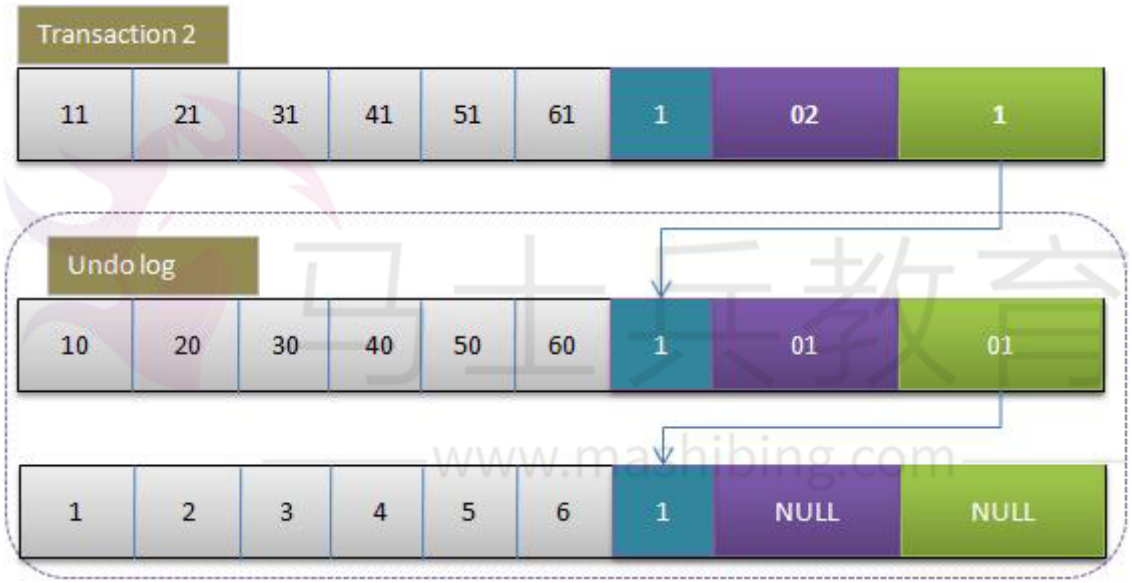
2.事务1更改该行的各字段的值



当事务1更改该行的值时，会进行如下操作：

- 用排他锁锁定该行
- 记录redo log
- 把该行修改前的值Copy到undo log，即上图中下面的行
- 修改当前行的值，填写事务编号，使回滚指针指向undo log中的修改前的行

3.事务2修改该行的值



与事务1相同，此时undo log，中有有两行记录，并且通过回滚指针连在一起。

因此，如果undo log一直不删除，则会通过当前记录的回滚指针回溯到该行创建时的初始内容，所幸的时在Innodb中存在purge线程，它会查询那些比现在最老的活动事务还早的undo log，并删除它们，从而保证undo log文件不至于无限增长。

4. 事务提交

当事务正常提交时Innbod只需要更改事务状态为COMMIT即可，不需做其他额外的工作，而Rollback则稍微复杂点，需要根据当前回滚指针从undo log中找出事务修改前的版本，并恢复。如果事务影响的行非常多，回滚则可能会变的效率不高，根据经验值没事务行数在1000～10000之间，Innodb效率还是非常高的。很显然，Innodb是一个COMMIT效率比Rollback高的存储引擎。据说，Postgress的实现恰好与此相反。

5. Insert Undo log

上述过程确切地说是描述了UPDATE的事务过程，其实undo log分insert和update undo log，因为insert时，原始的数据并不存在，所以回滚时把insert undo log丢弃即可，而update undo log则必须遵守上述过程。

3. 事务级别

众所周知地是更新（update、insert、delete）是一个事务过程，在Innodb中，查询也是一个事务，只读事务。当读写事务并发访问同一行数据时，能读到什么样的内容则依赖事务级别：

- READ_UNCOMMITTED
  读未提交时，读事务直接读取主记录，无论更新事务是否完成
- READ_COMMITTED
  读提交时，读事务每次都读取undo log中最近的版本，因此两次对同一字段的读可能读到不同的数据（幻读），但能保证每次都读到最新的数据。
- REPEATABLE_READ
  每次都读取指定的版本，这样保证不会产生幻读，但可能读不到最新的数据
- SERIALIZABLE
  锁表，读写相互阻塞，使用较少

读事务一般有SELECT语句触发，在Innodb中保证其非阻塞，但带FOR UPDATE的SELECT除外，带FOR UPDATE的SELECT会对行加排他锁，等待更新事务完成后读取其最新内容。就整个Innodb的设计目标来说，就是提供高效的、非阻塞的查询操作。

4. MVCC

上述更新前建立undo log，根据各种策略读取时非阻塞就是MVCC，undo log中的行就是MVCC中的多版本，这个可能与我们所理解的MVCC有较大的出入，一般我们认为MVCC有下面几个特点：

- 每行数据都存在一个版本，每次数据更新时都更新该版本
- 修改时Copy出当前版本随意修改，个事务之间无干扰
- 保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃copy（rollback）

就是每行都有版本号，保存时根据版本号决定是否成功，听起来含有乐观锁的味道。。。，而Innodb的实现方式是：

- 事务以排他锁的形式修改原始数据
- 把修改前的数据存放于undo log，通过回滚指针与主数据关联
- 修改成功（commit）啥都不做，失败则恢复undo log中的数据（rollback）

二者最本质的区别是，当修改数据时是否要排他锁定，如果锁定了还算不算是MVCC?

Innodb的实现真算不上MVCC，因为并没有实现核心的多版本共存，undo log中的内容只是串行化的结果，记录了多个事务的过程，不属于多版本共存。但理想的MVCC是难以实现的，当事务仅修改一行记录使用理想的MVCC模式是没有问题的，可以通过比较版本号进行回滚；但当事务影响到多行数据时，理想的MVCC据无能为力了。

比如，如果Transaciton1执行理想的MVCC，修改Row1成功，而修改Row2失败，此时需要回滚Row1，但因为Row1没有被锁定，其数据可能又被Transaction2所修改，如果此时回滚Row1的内容，则会破坏Transaction2的修改结果，导致Transaction2违反ACID。

理想MVCC难以实现的根本原因在于企图通过乐观锁代替二段提交。修改两行数据，但为了保证其一致性，与修改两个分布式系统中的数据并无区别，而二提交是目前这种场景保证一致性的唯一手段。二段提交的本质是锁定，乐观锁的本质是消除锁定，二者矛盾，故理想的MVCC难以真正在实际中被应用，Innodb只是借了MVCC这个名字，提供了读的非阻塞而已。

5.总结

也不是说MVCC就无处可用，对一些一致性要求不高的场景和对单一数据的操作的场景还是可以发挥作用的，比如多个事务同时更改用户在线数，如果某个事务更新失败则重新计算后重试，直至成功。这样使用MVCC会极大地提高并发数，并消除线程锁。

mysql索引原理之聚簇索引

索引分为聚簇索引和非聚簇索引。

以一本英文课本为例，要找第8课，直接翻书，若先翻到第5课，则往后翻，再翻到第10课，则又往前翻。这本书本身就是一个索引，即“聚簇索引”。

如果要找"fire”这个单词，会翻到书后面的附录，这个附录是按字母排序的，找到F字母那一块，再找到"fire”，对应的会是它在第几课。这个附录，为“非聚簇索引”。

由此可见，聚簇索引，索引的顺序就是数据存放的顺序，所以，很容易理解，一张数据表只能有一个聚簇索引。

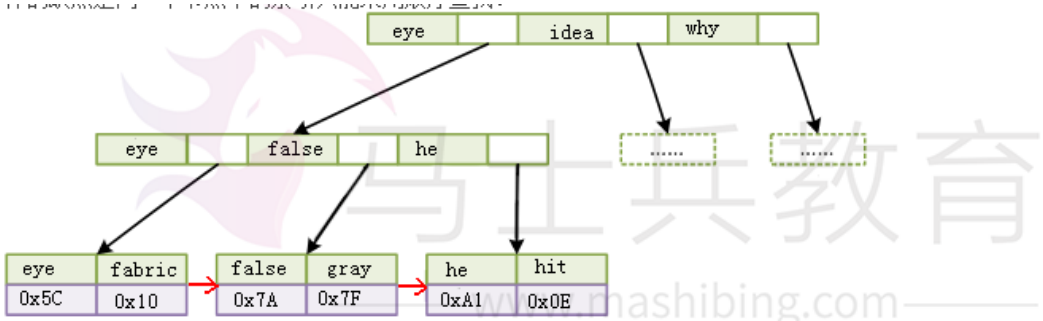
聚簇索引要比非聚簇索引查询效率高很多，特别是范围查询的时候。所以，至于聚簇索引到底应该为主键，还是其他字段，这个可以再讨论。

1、MYSQL的索引

mysql中，不同的存储引擎对索引的实现方式不同，大致说下MyISAM和InnoDB两种存储引擎。

MyISAM的B+Tree的叶子节点上的data，并不是数据本身，而是数据存放的地址。主索引和辅助索引没啥区别，只是主索引中的key一定得是唯一的。这里的索引都是非聚簇索引。

MyISAM还采用压缩机制存储索引，比如，第一个索引为“her”，第二个索引为“here”，那么第二个索引会被存储为“3,e”，这样的缺点是同一个节点中的索引只能采用顺序查找。



InnoDB的数据文件本身就是索引文件，B+Tree的叶子节点上的data就是数据本身，key为主键，这是聚簇索引。非聚簇索引，叶子节点上的data是主键(所以聚簇索引的key，不能过长)。为什么存放的主键，而不是记录所在地址呢，理由相当简单，因为记录所在地址并不能保证一定不会变，但主键可以保证。

至于为什么主键通常建议使用自增id呢?

2、聚簇索引

聚簇索引的数据的物理存放顺序与索引顺序是一致的，即：只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的。如果主键不是自增id，那么可以想象，它会干些什么，不断地调整数据的物理地址、分页，当然也有其他一些措施来减少这些操作，但却无法彻底避免。但，如果是自增的，那就简单了，它只需要一页一页地写，索引结构相对紧凑，磁盘碎片少，效率也高。

聚簇索引不但在检索上可以大大滴提高效率，在数据读取上也一样。比如：需要查询f~t的所有单词。

一个使用MyISAM的主索引，一个使用InnoDB的聚簇索引。两种索引的B+Tree检索时间一样，但读取时却有了差异。

因为MyISAM的主索引并非聚簇索引，那么他的数据的物理地址必然是凌乱的，拿到这些物理地址，按照合适的算法进行I/O读取，于是开始不停的寻道不停的旋转。聚簇索引则只需一次I/O。

不过，如果涉及到大数据量的排序、全表扫描、count之类的操作的话，还是MyISAM占优势些，因为索引所占空间小，这些操作是需要在内存中完成的。

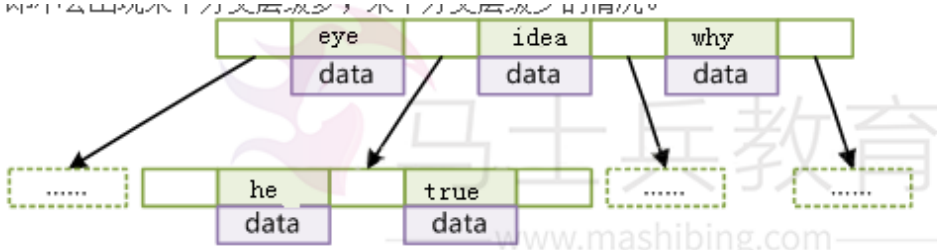
鉴于聚簇索引的范围查询效率，很多人认为使用主键作为聚簇索引太多浪费，毕竟几乎不会使用主键进行范围查询。但若再考虑到聚簇索引的存储，就不好定论了。

学习笔记_mysql索引原理之B+/-Tree

索引，是为了更快的查询数据，查询算法有很多，对应的数据结构也不少，数据库常用的索引数据结构一般为B+Tree。

1、B-Tree

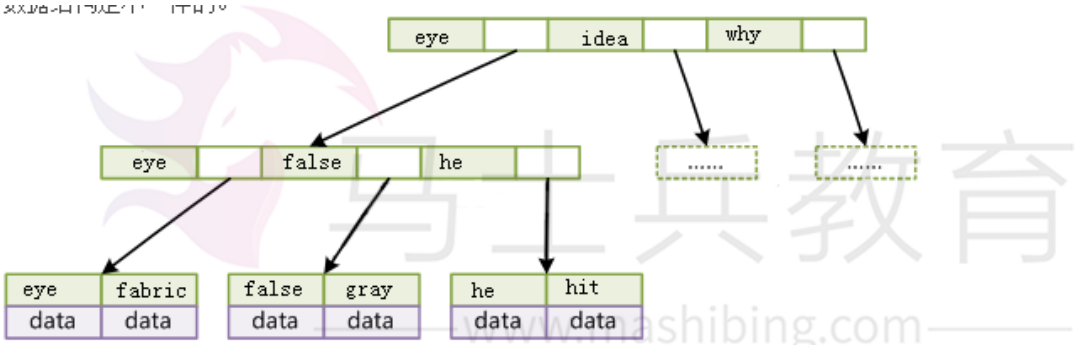
关于B-Tree的官方定义个人觉得比较难懂，通俗一点就是举个例子。假如：一本英文字典，单词+详细解释组成了一条记录，现在需要索引单词，那么以单词为key，单词+详细解释为data，B-Tree就是以一个二元组{key,data}来定义一条记录。如果一个节点有3条记录，那么会有对应的4个指针，用以指向下一个节点。B-Tree是有序且平衡的，所有叶子节点在同一层，即不会出现某个分支层级多，某个分支层级少的情况。



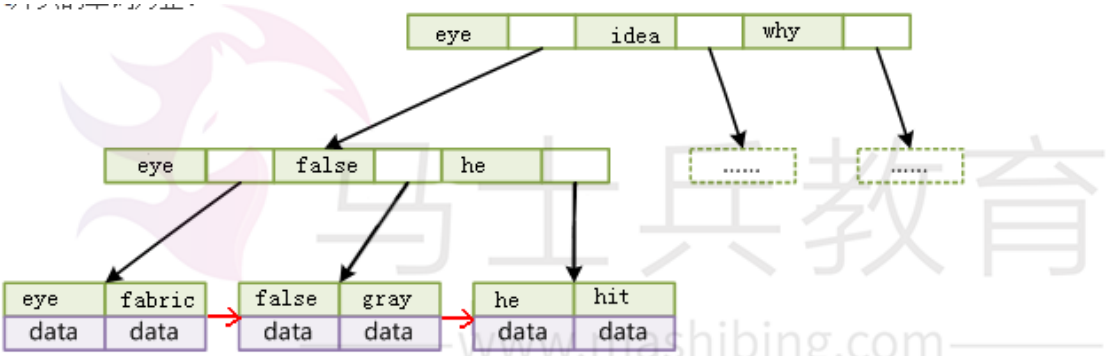
因为B-Tree是有序的，所以它的查找就简单了，先从根节点开始二分查找，找到则返回节点；否则沿着区间指针查找下一个节点。比如，查询false这个单词。

2、B+Tree

与B-Tree不同的是，B+Tree每个节点只有key，没有data；而且叶子节点没有指针。也就是说B+Tree的叶子节点和内节点的数据结构是不一样的。



一般数据库采用的是B+Tree，而且经过了一些优化，比如在叶子节点上增加了顺序访问指针，提高区间查询效率。比如：查询首字母为f~t的所有单词。那么只需查到f开头的第一个单词fabric，然后沿着叶子节点的开始遍历，直到找到最后一个以t开头的单词为止。



简单介绍了B-/+Tree，至于众多数据结构中，为何数据库索引选择BTree，而且选择B+Tree，下面从计算机存储原理方面简单说说。

3、读内存和读磁盘

内存读取和磁盘读取的效率是相差很大的。

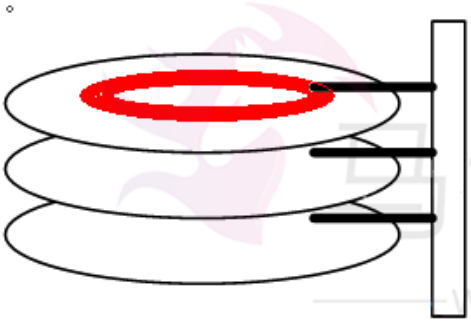
简单点说说内存读取，内存是由一系列的存储单元组成的，每个存储单元存储固定大小的数据，且有一个唯一地址。

当需要读内存时，将地址信号放到地址总线上传给内存，内存解析信号并定位到存储单元，然后把该存储单元上的数据放到数据总线上，回传。

写内存时，系统将要写入的数据和单元地址分别放到数据总线和地址总线上，内存读取两个总线的内容，做相应的写操作。

内存存取效率，跟次数有关，先读取A数据还是后读取A数据不会影响存取效率。而磁盘存取就不一样了，磁盘I/O涉及机械操作。

磁盘是由大小相同且同轴的圆形盘片组成，磁盘可以转动(各个磁盘须同时转动)。磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不动，磁盘转动，但磁臂可以前后动，用于读取不同磁道上的数据。磁道就是以盘片为中心划分出来的一系列同心环(如图标红那圈)。磁道又划分为一个个小段，叫扇区，是磁盘的最小存储单元。



码士兵教育
www.mashibing.com

磁盘读取时，系统将数据逻辑地址传给磁盘，磁盘的控制电路会解析出物理地址，即哪个磁道哪个扇区。于是磁头需要前后移动到对应的磁道，消耗的时间叫寻道时间，然后磁盘旋转将对应的扇区转到磁头下，消耗的时间叫旋转时间。所以，适当的操作顺序和数据存放可以减少寻道时间和旋转时间。

为了尽量减少I/O操作，磁盘读取每次都会预读，大小通常为页的整数倍。即使只需要读取一个字节，磁盘也会读取一页的数据(通常为4K)放入内存，内存与磁盘以页为单位交换数据。因为局部性原理认为，通常一个数据被用到，其附近的数据也会立马被用到。

4、检索性能分析

B-Tree：如果一次检索需要访问4个节点，数据库系统设计者利用磁盘预读原理，把节点的大小设计为一个页，那读取一个节点只需要一次I/O操作，完成这次检索操作，最多需要3次I/O(根节点常驻内存)。数据记录越小，每个节点存放的数据就越多，树的高度也就越小，I/O操作就少了，检索效率也就上去了。

B+Tree：内节点只存key，大大滴减少了内节点的大小，那么每个节点就可以存放更多的记录，树的更矮了，I/O操作更少了。所以B+Tree拥有更好的性能。

5、其他索引方式

散列索引：通过HASH来定位的一种索引，这种索引用的较少，通过用于单值查询。InnoDB的自适应索引就是HASH索引。

位图索引：字段值固定且少，比如性别、状态。在同时对多个这样的字段and/or查询时，效率极高，直接按位与/或就可以得到结果了。所以，应用范围局限。

关系型和非关系型数据库区别

- 关系型数据库：是指采用了关系模型(二维表格模型)来组织数据的数据库。
- 非关系型数据库：以键值对存储，且结构不固定.

我的Mysql死锁排查过程（案例分析）

以前接触到的数据库死锁，都是批量更新时加锁顺序不一致而导致的死锁，但是上周却遇到了一个很难理解的死锁。借着这个机会又重新学习了一下mysql的死锁知识以及常见的死锁场景。在多方调研以及和同事们的讨论下终于发现了这个死锁问题的成因，收获颇多。虽然是后端程序员，我们不需要像DBA一样深入地去分析与锁相关的源码，但是如果我们能够掌握基本的死锁排查方法，对我们的日常开发还是大有裨益的。

死锁起因

先介绍一下数据库和表情况，因为涉及到公司内部真是的数据，所以以下都做了模拟，不会影响具体的分析。

我们采用的是5.5版本的mysql数据库，事务隔离级别是默认的RR（Repeatable-Read），采用innodb引擎。假设存在test表：

```
CREATE TABLE `test` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `a` int(11) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `a` (`a`)
)
```

表的结构很简单，一个主键id，另一个唯一索引a。表里的数据如下：

```
mysql> select * from test;
+----+------+
| id | a    |
+----+------+
|  1 |    1 |
|  2 |    2 |
|  4 |    4 |
+----+------+
3 rows in set (0.00 sec)
```

出现死锁的操作如下：

步骤	事务1	事务2
1		begin
2		delete from test where a = 2;
3	begin	
4	delete from test where a = 2; （事务1卡住）	
5	提示出现死锁：ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction	insert into test (id, a) values (10, 2);

然后我们可以通过 SHOW ENGINE INNODB STATUS; 来查看死锁日志：

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
170219 13:31:31
*** (1) TRANSACTION:
TRANSACTION 2A8BD, ACTIVE 11 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s)
MySQL thread id 448218, OS thread handle 0x2abe5fb5d700, query id 18923238
renjun.fangcloud.net 121.41.41.92 root updating
delete from test where a = 2
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table
`oauthdemo`.`test` trx id 2A8BD lock_mode X waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** (2) TRANSACTION:
TRANSACTION 2A8BC, ACTIVE 18 sec inserting
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1248, 3 row lock(s), undo log entries 2
MySQL thread id 448217, OS thread handle 0x2abe5fd65700, query id 18923239
renjun.fangcloud.net 121.41.41.92 root update
insert into test (id,a) values (10,2)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table
`oauthdemo`.`test` trx id 2A8BC lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table
`oauthdemo`.`test` trx id 2A8BC lock mode S waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** WE ROLL BACK TRANSACTION (1)
```

分析

阅读死锁日志

遇到死锁，第一步就是阅读死锁日志。死锁日志通常分为两部分，上半部分说明了事务1在等待什么锁：

```
170219 13:31:31
*** (1) TRANSACTION:
TRANSACTION 2A8BD, ACTIVE 11 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s)
MySQL thread id 448218, OS thread handle 0x2abe5fb5d700, query id 18923238
renjun.fangcloud.net 121.41.41.92 root updating
delete from test where a = 2
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table
`oauthdemo`.`test` trx id 2A8BD lock_mode X waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;
```

从日志里我们可以看到事务1当前正在执行 delete from test where a = 2 ，该条语句正在申请索引a的X锁，所以提示 lock_mode X waiting 。

然后日志的下半部分说明了事务2当前持有的锁以及等待的锁：

```
*** (2) TRANSACTION:
TRANSACTION 2A8BC, ACTIVE 18 sec inserting
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1248, 3 row lock(s), undo log entries 2
MySQL thread id 448217, OS thread handle 0x2abe5fd65700, query id 18923239
renjun.fangcloud.net 121.41.41.92 root update
insert into test (id,a) values (10,2)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table
`oauthdemo`.`test` trx id 2A8BC lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 923 n bits 80 index `a` of table
`oauthdemo`.`test` trx id 2A8BC lock mode S waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 32
 0: len 4; hex 00000002; asc     ;;
 1: len 4; hex 00000002; asc     ;;
```

从日志的 HOLDS THE LOCKS(S) 块中我们可以看到事务2持有索引a的X锁，并且是记录锁（Record Lock）。该锁是通过事务2在步骤2执行的delete语句申请的。由于是RR隔离模式下的基于唯一索引的等值查询（Where a = 2），所以会申请一个记录锁，而非next-key锁。

从日志的 WAITING FOR THIS LOCK TO BE GRANTED 块中我们可以看到事务2正在申请S锁，也就是共享锁。该锁是 insert into test (id,a) values (10,2) 语句申请的。 insert语句在普通情况下是会申请排他锁，也就是X锁，但是这里出现了S锁。这是因为a字段是一个唯一索引，所以insert语句会在插入前进行一次duplicate key的检查，为了使这次检查成功，需要申请S锁防止其他事务对a字段进行修改。

那么为什么该S锁会失败呢？这是 **对同一个字段的锁的申请是需要排队的** 。S锁前面还有一个未申请成功的X锁，所以S锁必须等待，所以形成了循环等待，死锁出现了。

通过阅读死锁日志，我们可以清楚地知道两个事务形成了怎样的循环等待，再加以分析，就可以逆向推断出循环等待的成因，也就是死锁形成的原因。

死锁形成流程图

为了让大家更好地理解死锁形成的原因，我们再通过表格的形式阐述死锁形成的流程：

步骤	事务1	事务2
1		begin
2		delete from test where a = 2; 执行成功，事务2占有a=2下的X锁，类型为记录锁。
3	begin	
4	delete from test where a = 2; 事务1希望申请a=2下的X锁，但是由于事务2已经申请了一把X锁，两把X锁互斥，所以X锁申请进入锁请求队列。	
5	出现死锁，事务1权重较小，所以被选择回滚（成为牺牲品）。	insert into test (id, a) values (10, 2); 由于a字段建立了唯一索引，所以需要申请S锁以便检查duplicate key，由于插入的a的值还是2，所以排在X锁后面。但是前面的X锁的申请只有在事务2commit或者rollback之后才能成功，此时形成了循环等待，死锁产生。

拓展

在排查死锁的过程中，有个同事还发现了上述场景会产生另一种死锁，该场景无法通过手工复现，只有高并发场景下才有可能复现。

该死锁对应的日志这里就不贴出了，与上一个死锁的核心差别是事务2等待的锁从S锁换成了X锁，也就是 lock_mode X locks gap before rec insert intention waiting 。我们还是通过表格来详细说明该死锁产生的流程：

步骤	事务1	事务2
1		begin
2		delete from test where a = 2; 执行成功，事务2占有a=2下的X锁，类型为记录锁。
3	begin	
4		【insert第1阶段】insert into test (id, a) values (10, 2); 事务2申请S锁进行duplicate key进行检查。检查成功。
5	delete from test where a = 2; 事务1希望申请a=2下的X锁，但是由于事务2已经申请了一把X锁，两把X锁互斥，所以X锁申请进入锁请求队列。	
6	出现死锁，事务1权重较小，所以被选择回滚（成为牺牲品）。	【insert第2阶段】insert into test (id, a) values (10, 2); 事务2开始插入数据，S锁升级为X锁，类型为insert intention。同理，X锁进入队列排队，形成循环等待，死锁产生。

总结

排查死锁时，首先需要根据死锁日志分析循环等待的场景，然后根据当前各个事务执行的SQL分析出加锁类型以及顺序，逆向推断出如何形成循环等待，这样就能找到死锁产生的原因了。

MySql优化

MySQL 对于千万级的大表要怎么优化?

很多人第一反应是各种切分；我给的顺序是:
第一优化你的sql和索引；

第二加缓存，memcached,redis；

第三以上都做了后，还是慢，就做主从复制或主主复制，读写分离，可以在应用层做，效率高，也可以用三方工具，第三方工具推荐360的atlas,其它的要么效率不高，要么没人维护；

第四如果以上都做了还是慢，不要想着去做切分，mysql自带分区表，先试试这个，对你的应用是透明的，无需更改代码,但是sql语句是需要针对分区表做优化的，sql条件中要带上分区条件的列，从而使查询定位到少量的分区上，否则就会扫描全部分区，另外分区表还有一些坑，在这里就不多说了；

第五如果以上都做了，那就先做垂直拆分，其实就是根据你模块的耦合度，将一个大的系统分为多个小的系统，也就是分布式系统；

第六才是水平切分，针对数据量大的表，这一步最麻烦，最能考验技术水平，要选择一个合理的sharding key,为了有好的查询效率，表结构也要改动，做一定的冗余，应用也要改，sql中尽量带sharding key，将数据定位到限定的表上去查，而不是扫描全部的表；

mysql数据库一般都是按照这个步骤去演化的，成本也是由低到高；

有人也许要说第一步优化sql和索引这还用说吗？的确，大家都知道，但是很多情况下，这一步做的并不到位，甚至有的只做了根据sql去建索引，根本没对sql优化（中枪了没？），除了最简单的增删改查外，想实现一个查询，可以写出很多种查询语句，不同的语句，根据你选择的引擎、表中数据的分布情况、索引情况、数据库优化策略、查询中的锁策略等因素，最终查询的效率相差很大；优化要从整体去考虑，有时你优化一条语句后，其它查询反而效率被降低了，所以要取一个平衡点；即使精通mysql的话，除了纯技术面优化，还要根据业务面去优化sql语句，这样才能达到最优效果；你敢说你的sql和索引已经是最优了吗?

再说一下不同引擎的优化，myisam读的效果好，写的效率差，这和它数据存储格式，索引的指针和锁的策略有关的，它的数据是顺序存储的（innodb数据存储方式是聚簇索引），他的索引btree上的节点是一个指向数据物理位置的指针，所以查找起来很快，（innodb索引节点存的则是数据的主键，所以需要根据主键二次查找）；myisam锁是表锁，只有读读之间是并发的，写写之间和读写之间（读和插入之间是可以并发的，去设置concurrent_insert参数，定期执行表优化操作，更新操作就没有办法了）是串行的，所以写起来慢，并且默认的写优先级比读优先级高，高到写操作来了后，可以马上插入到读操作前面去，如果批量写，会导致读请求饿死，所以要设置读写优先级或设置多少写操作后执行读操作的策略;myisam不要使用查询时间太长的sql，如果策略使用不当，也会导致写饿死，所以尽量去拆分查询效率低的sql,

innodb一般都是行锁，这个一般指的是sql用到索引的时候，行锁是加在索引上的，不是加在数据记录上的，如果sql没有用到索引，仍然会锁定表,mysql的读写之间是可以并发的，普通的select是不需要锁的，当查询的记录遇到锁时，用的是一致性的非锁定快照读，也就是根据数据库隔离级别策略，会去读被锁定行的快照，其它更新或加锁读语句用的是当前读，读取原始行；因为普通读与写不冲突，所以innodb不会出现读写饿死的情况，又因为在使用索引的时候用的是行锁，锁的粒度小，竞争相同锁的情况就少，就增加了并发处理，所以并发读写的效率还是很优秀的，问题在于索引查询后的根据主键的二次查找导致效率低；

ps:很奇怪，为什innodb的索引叶子节点存的是主键而不是像mysism一样存数据的物理地址指针吗？如果存的是物理地址指针不就不需要二次查找了吗，这也是我开始的疑惑，根据mysism和innodb数据存储方式的差异去想，你就会明白了，我就不费口舌了！

所以innodb为了避免二次查找可以使用索引覆盖技术，无法使用索引覆盖的，再延伸一下就是基于索引覆盖实现延迟关联；不知道什么是索引覆盖的，建议你无论如何都要弄清楚它是怎么回事！

尽你所能去优化你的sql吧！说它成本低，却又是一项费时费力的活，需要在技术与业务都熟悉的情况下，用心去优化才能做到最优，优化后的效果也是立竿见影的！

产生死锁的必要条件

操作系统：死锁的产生、条件、和解锁

deadlocks（死锁）

所谓死锁: 是指两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去.此时称系统处于死锁状态或系统产生了死锁,这些永远在互相等竺的进程称为死锁进程.

由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

一种情形，此时执行程序中两个或多个线程发生永久堵塞（等待），每个线程都在等待被其他线程占用并堵塞了的资源。例如，如果线程A锁住了记录1并等待记录2，而线程B锁住了记录2并等待记录1，这样两个线程就发生了死锁现象。

计算机系统中,如果系统的资源分配策略不当，更常见的可能是程序员写的程序有错误等，则会导致进程因竞争资源不当而产生死锁的现象。

产生死锁的原因主要是：

（1） 因为系统资源不足。

（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则

就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在未使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之

一不满足，就不会发生死锁。

死锁的解除与预防

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源,在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配 。因此，对资源的分配要给予合理的规划。

一、有序资源分配法

这种算法资源按某种规则系统中的所有资源统一编号（例如打印机为1、磁带机为2、磁盘为3、等等），申请时必须以上升的次序。系统要求申请进程：

1、对它所必须使用的而且属于同一类的所有资源，必须一次申请完；

2、在申请不同类资源时，必须按各类设备的编号依次申请。例如：进程PA，使用资源的顺序是R1，R2； 进程PB，使用资源的顺序是R2，R1；若采用动态分配有可能形成环路条件，造成死锁。

采用有序资源分配法：R1的编号为1，R2的编号为2；

PA：申请次序应是：R1，R2

PB：申请次序应是：R1，R2

这样就破坏了环路条件，避免了死锁的发生

二、银行算法

避免死锁算法中最有代表性的算法是Dijkstra E.W 于1968年提出的银行家算法：

该算法需要检查申请者对资源的最大需求量，如果系统现存的各类资源可以满足申请者的请求，就满足申请者的请求。

这样申请者就可很快完成其计算，然后释放它占用的资源，从而保证了系统中的所有进程都能完成，所以可避免死锁的发生。

死锁排除的方法

1、撤消陷于死锁的全部进程；

2、逐个撤消陷于死锁的进程，直到死锁不存在；

3、从陷于死锁的进程中逐个强迫放弃所占用的资源，直至死锁消失。

4、从另外一些进程那

Spring（连老师）

Spring 原理

它是一个全面的、企业应用开发一站式的解决方案，贯穿表现层、业务层、持久层。但是 Spring 仍然可以和其他的框架无缝整合。

Spring 特点

轻量级

控制反转

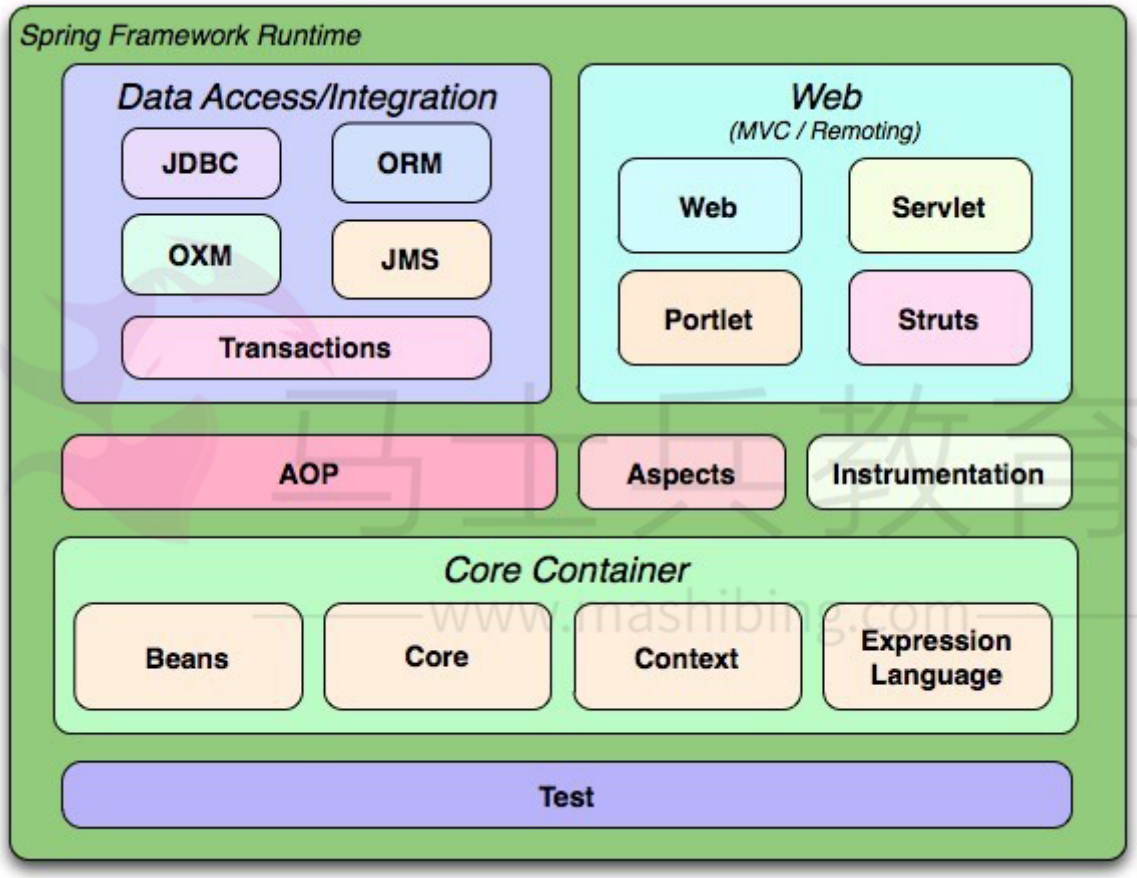
面向切面

容器

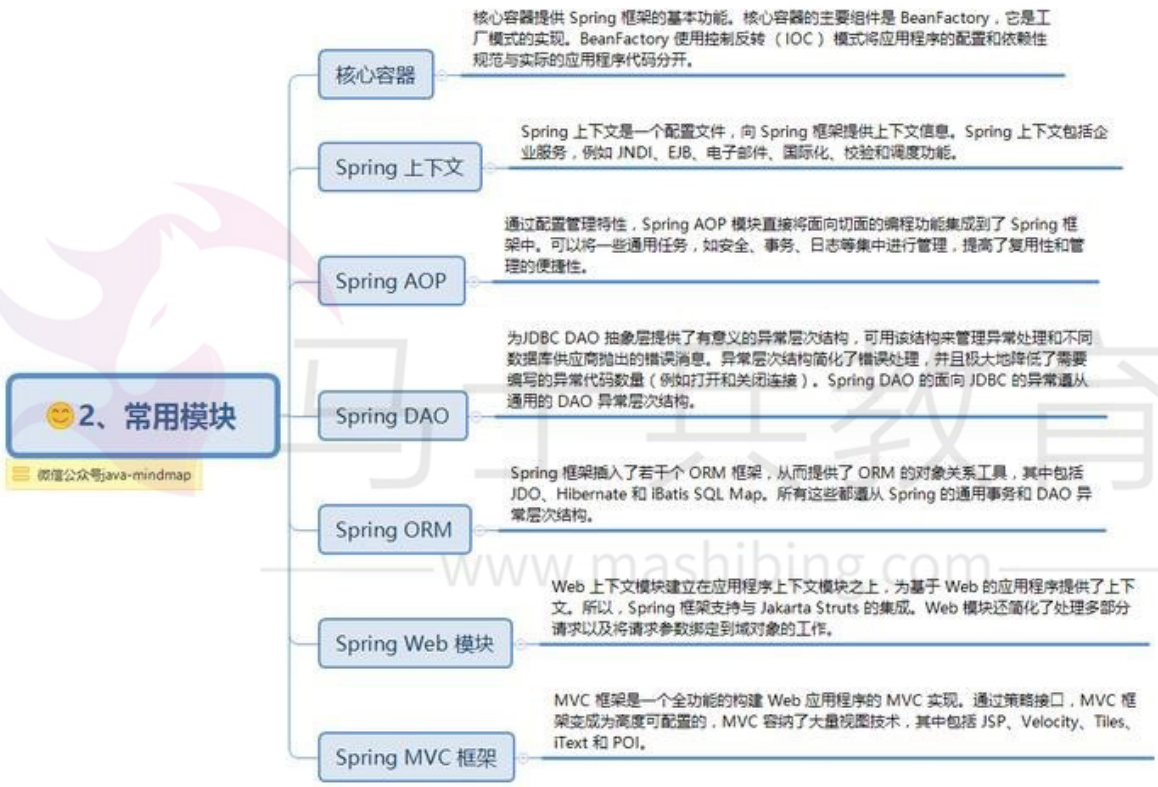
框架集合



Spring 核心组件



Spring 常用模块

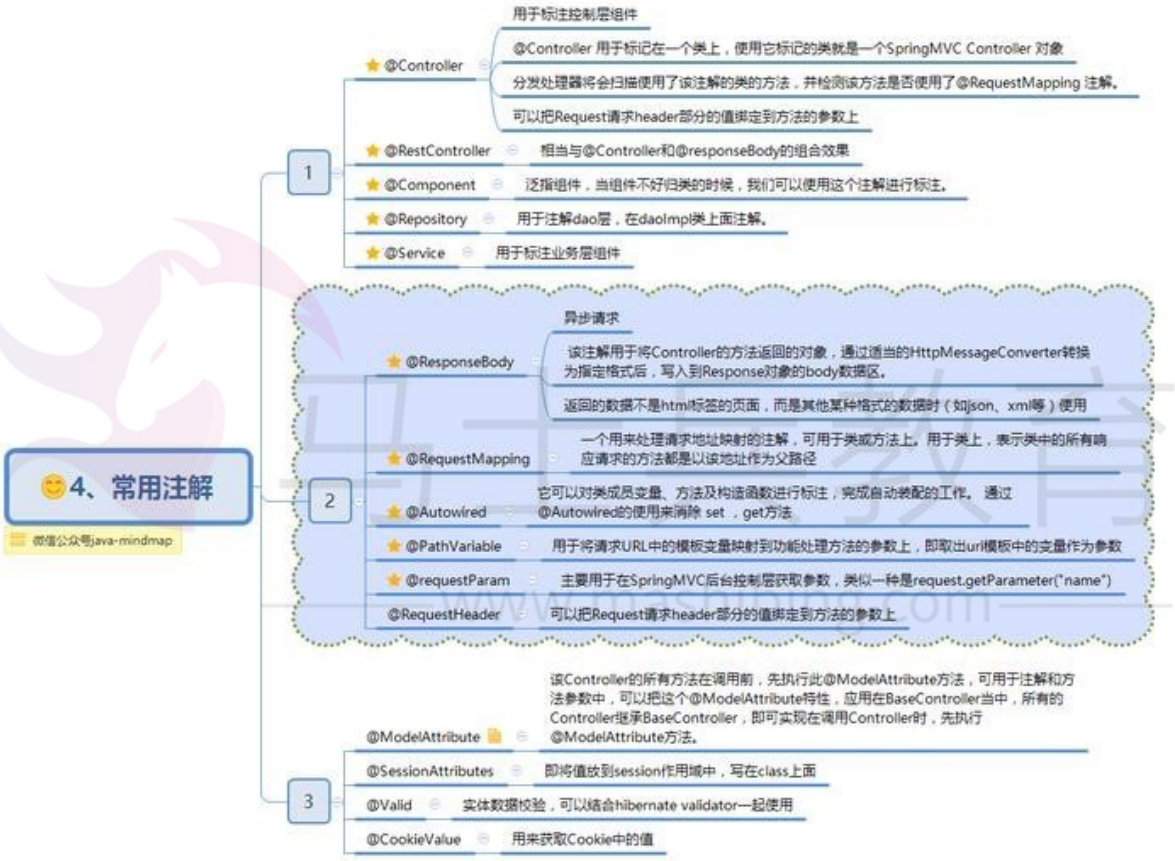

2、常用模块
微信公众号java-mindmap
核心容器
核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
Spring 上下文
Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
Spring AOP
通过配置管理特性，Spring AOP 模块直接将面向切面的编程功能集成到了 Spring 框架中。可以将一些通用任务，如安全、事务、日志等集中进行管理，提高了复用性和管理的便捷性。
Spring DAO
为JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。
Spring ORM
Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 IBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
Spring Web 模块
Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
Spring MVC 框架
MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

Spring 主要包

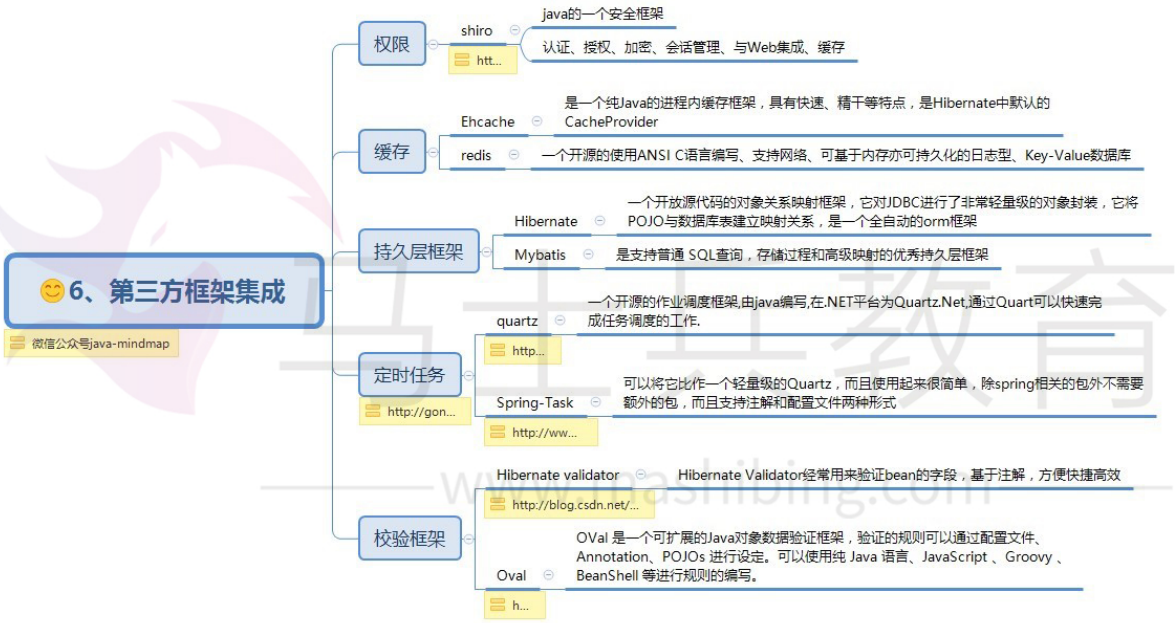


Spring 常用注解

bean 注入与装配的的方式有很多种，可以通过 xml，get set 方式，构造函数或者注解等。简单易用的方式就是使用 Spring 的注解了，Spring 提供了大量的注解方式。



Spring 第三方结合



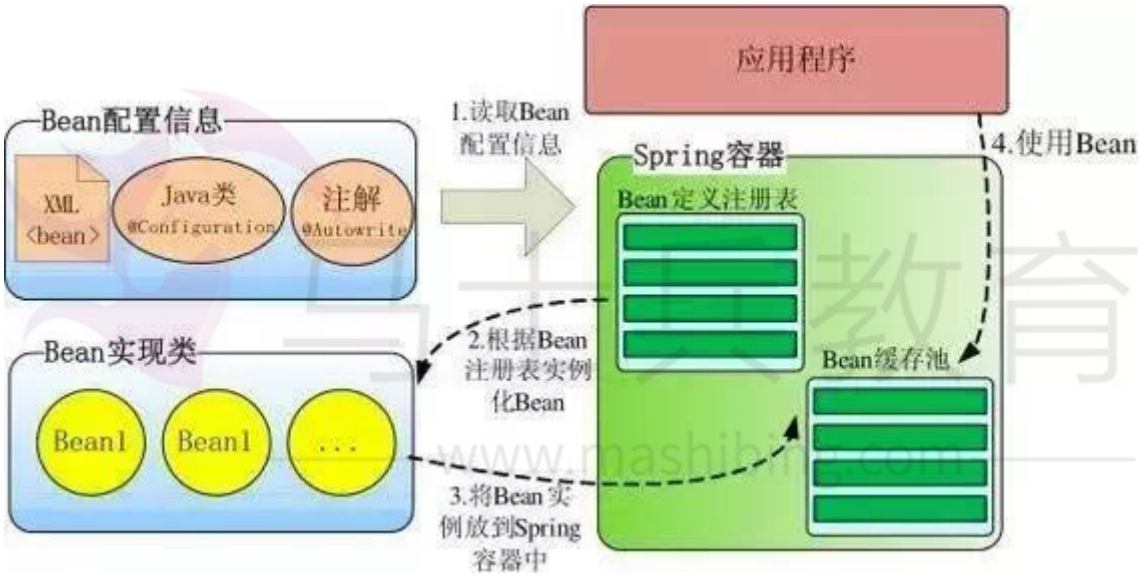
Spring IOC 原理

概念

Spring 通过一个配置文件描述 Bean 及 Bean 之间的依赖关系，利用 Java 语言的反射功能实例化 Bean 并建立 Bean 之间的依赖关系。 Spring 的 IoC 容器在完成这些底层工作的基础上，还提供了 Bean 实例缓存、生命周期管理、 Bean 实例代理、事件发布、资源装载等高级服务。

Spring容器高层视图

Spring 启动时读取应用程序提供的 Bean 配置信息，并在 Spring 容器中生成一份相应的 Bean 配置注册表，然后根据这张注册表实例化 Bean，装配好 Bean 之间的依赖关系，为上层应用提供准备就绪的运行环境。其中 Bean 缓存池为 HashMap 实现



IOC 容器实现

BeanFactory-框架基础设施

BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合我们都直接使用 ApplicationContext 而非底层的 BeanFactory。

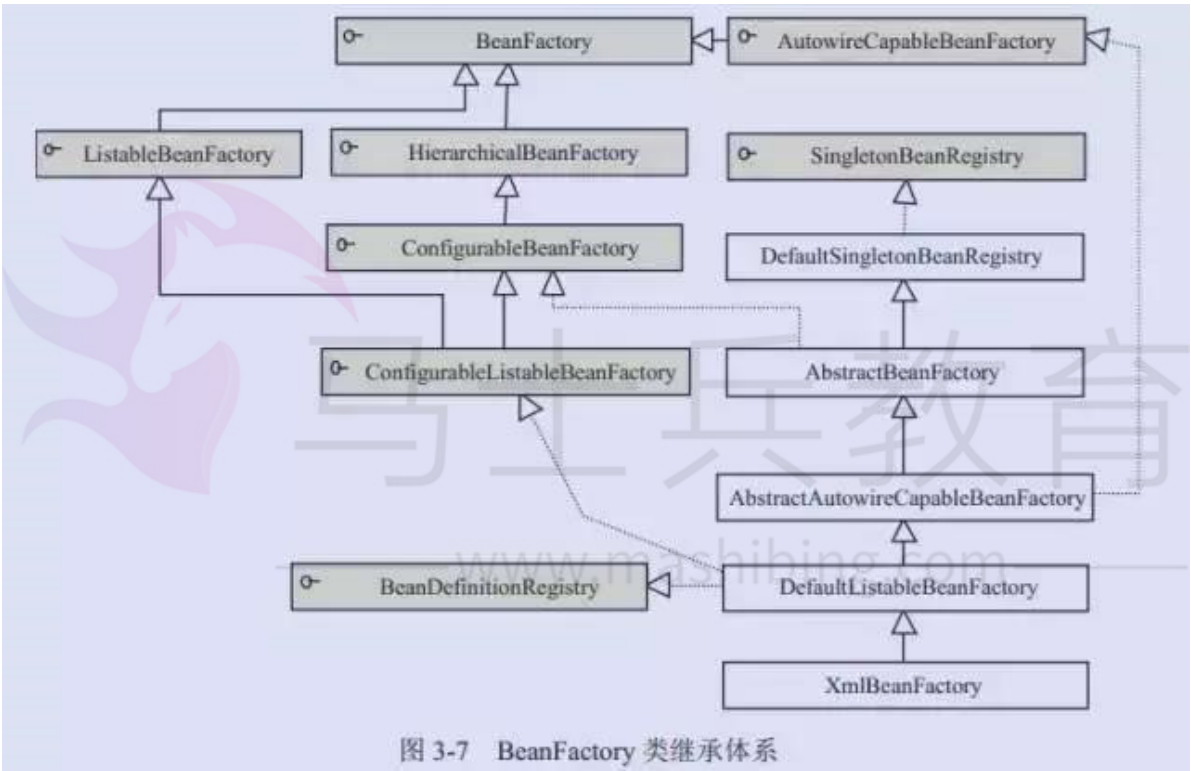


图 3-7　BeanFactory 类继承体系

*1　BeanDefinitionRegistry*注册表

1. Spring 配置文件中每一个节点元素在 Spring 容器里都通过一个 BeanDefinition 对象表示，它描述了 Bean 的配置信息。而 BeanDefinitionRegistry 接口提供了向容器手工注册 BeanDefinition 对象的方法。

2 BeanFactory 顶层接口

2. 位于类结构树的顶端 ，它最主要的方法就是 getBean(String beanName)，该方法从容器中返回特定名称的 Bean，BeanFactory 的功能通过其他的接口得到不断扩展：

3 ListableBeanFactory

3. 该接口定义了访问容器中 Bean 基本信息的若干方法，如查看 Bean 的个数、获取某一类型 Bean 的配置名、查看容器中是否包括某一 Bean 等方法；

*4 HierarchicalBeanFactory*父子级联

4. 父子级联 IoC 容器的接口，子容器可以通过接口方法访问父容器； 通过

HierarchicalBeanFactory 接口， Spring 的 IoC 容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的 Bean，但父容器不能访问子容器的 Bean。Spring 使用父子容器实现了很多功能，比如在 Spring MVC 中，展现层 Bean 位于一个子容器中，而业务层和持久层的 Bean 位于父容器中。这样，展现层 Bean 就可以引用业务层和持久层的 Bean，而业务层和持久层的 Bean 则看不到展现层的 Bean。

5 ConfigurableBeanFactory

5. 是一个重要的接口，增强了 IoC 容器的可定制性，它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法；

6 AutowireCapableBeanFactory 自动装配

6. 定义了将容器中的 Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法；

*7 SingletonBeanRegistry*运行期间注册单例*Bean*

7. 定义了允许在运行期间向容器注册单实例 Bean 的方法；对于单实例（ singleton）的 Bean 来说，BeanFactory 会缓存 Bean 实例，所以第二次使用 getBean() 获取 Bean 时将直接从 IoC 容器的缓存中获取 Bean 实例。Spring 在 DefaultSingletonBeanRegistry 类中提供了一个用于缓存单实例 Bean 的缓存器，它是一个用 HashMap 实现的缓存器，单实例的 Bean 以 beanName 为键保存在这个 HashMap 中。

8 依赖日志框框

8. 在初始化 BeanFactory 时，必须为其提供一种日志框架，比如使用 Log4J， 即在类路径下提供 Log4J 配置文件，这样启动 Spring 容器才不会报错。

ApplicationContext 面向开发应用

ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。

ApplicationContext 继承了 HierarchicalBeanFactory 和 ListableBeanFactory 接口，在此基础

上，还通过多个其他的接口扩展了 BeanFactory 的功能：

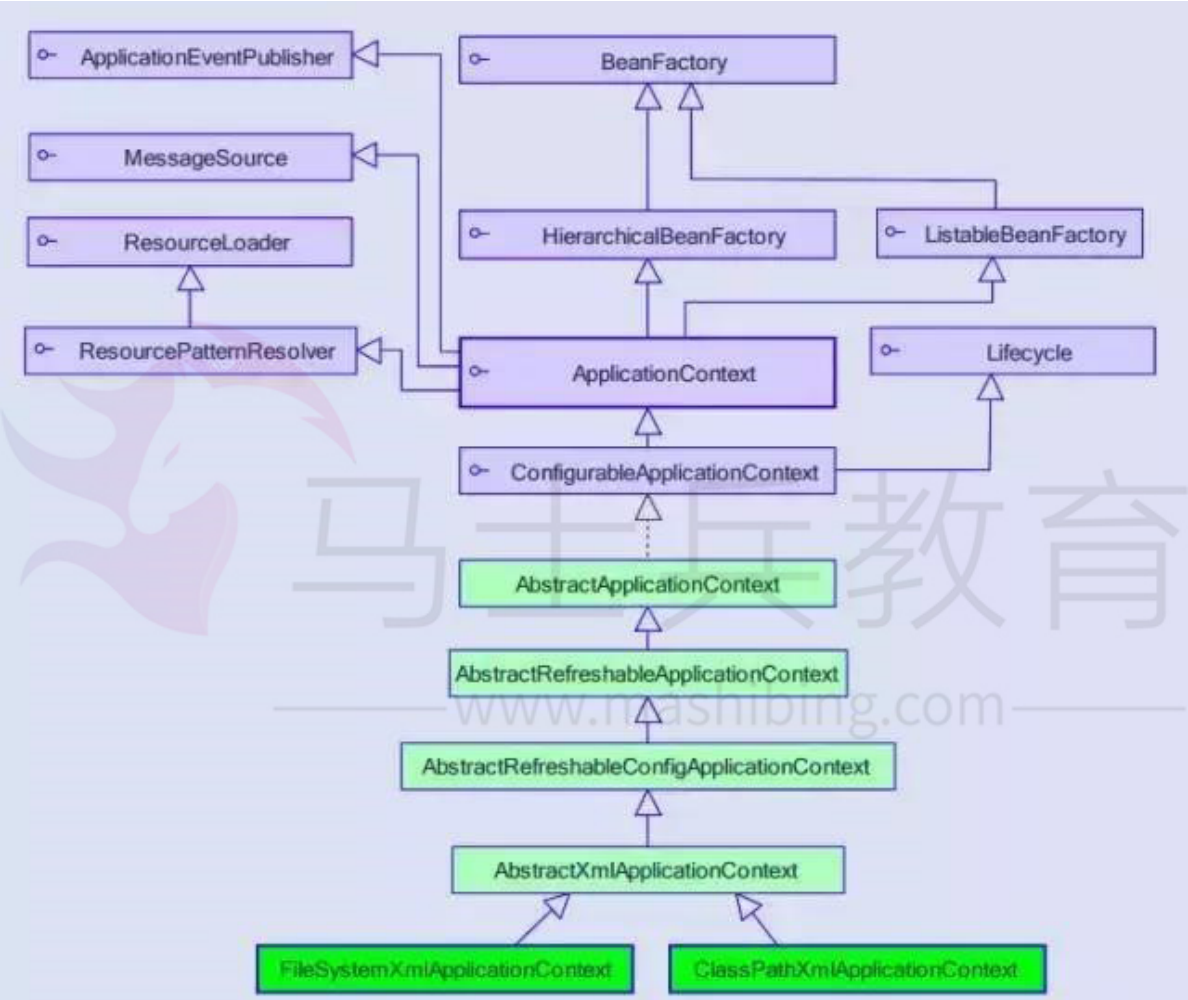
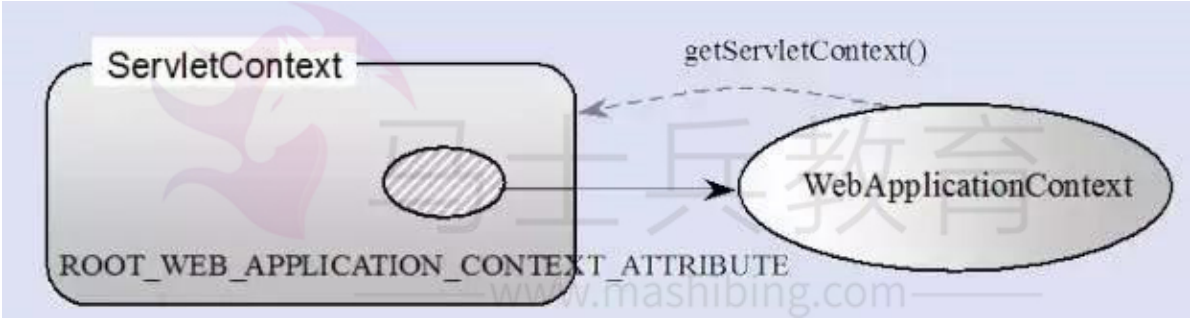


图 3-8 ApplicationContext 类继承体系

1. ClassPathXmlApplicationContext：默认从类路径加载配置文件
2. FileSystemXmlApplicationContext：默认从文件系统中装载配置文件
3. ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。
4. MessageSource：为应用提供 i18n 国际化消息访问的功能；
5. ResourcePatternResolver： 所有 ApplicationContext 实现类都实现了类似于 PathMatchingResourcePatternResolver 的功能，可以通过带前缀的 Ant 风格的资源文件路径装载 Spring 的配置文件。
6. LifeCycle：该接口是 Spring 2.0 加入的，该接口提供了 start()和 stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被 ApplicationContext 实现及具体 Bean 实现，ApplicationContext 会将 start/stop 的信息传递给容器中所有实现了该接口的 Bean，以达到管理和控制 JMX、任务调度等目的。
7. ConfigurableApplicationContext 扩展于 ApplicationContext，它新增加了两个主要的方法：refresh()和 close()，让 ApplicationContext 具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用 refresh()即可启动应用上下文，在已经启动的状态下，调用 refresh()则清除缓存并重新装载配置信息，而调用 close()则可关闭应用上下文。

WebApplication 体系架构

WebApplicationContext 是专门为 Web 应用准备的，它允许从相对于 Web 根目录的路径中装载配置文件完成初始化工作。从 WebApplicationContext 中可以获得 ServletContext 的引用，整个 Web 应用上下文对象将作为属性放置到 ServletContext 中，以便 Web 应用环境可以访问 Spring 应用上下文。



Spring Bean 作用域

Spring 3 中为 Bean 定义了 5 中作用域，分别为 singleton（单例）、prototype（原型）、 request、session 和 global session，5 种作用域说明如下： **singleton**：单例模式（多线程下不安全）

1. singleton：单例模式，Spring IoC 容器中只会存在一个共享的 Bean 实例，无论有多少个 Bean 引用它，始终指向同一对象。该模式在多线程下是不安全的。Singleton 作用域是 Spring 中的缺省作用域，也可以显示的将 Bean 定义为 singleton 模式，配置为：

prototype:原型模式每次使用时创建

2. prototype:原型模式，每次通过 Spring 容器获取 prototype 定义的 bean 时，容器都将创建一个新的 Bean 实例，每个 Bean 实例都有自己的属性和状态，而 singleton 全局只有一个对象。根据经验，对有状态的bean使用prototype作用域，而对无状态的bean使用singleton 作用域。

Request：一次**request** 一个实例

3. request：在一次 Http 请求中，容器会返回该 Bean 的同一实例。而对不同的 Http 请求则会产生新的 Bean，而且该 bean 仅在当前 Http Request 内有效,当前 Http 请求结束，该 bean 实例也将会被销毁。

session

4. session：在一次 Http Session 中，容器会返回该 Bean 的同一实例。而对不同的 Session 请求则会创建新的实例，该 bean 实例仅在当前 Session 内有效。同 Http 请求相同，每一次 session 请求创建新的实例，而不同的实例之间不共享属性，且实例仅在自己的 session 请求内有效，请求结束，则实例将被销毁。

global Session

5. global Session：在一个全局的 Http Session 中，容器会返回该 Bean 的同一个实例，仅在使用 portlet context 时有效。

Spring Bean 生命周期

实例化

1. 实例化一个 Bean，也就是我们常说的 new。

IOC 依赖注入

2. 按照 Spring 上下文对实例化的 Bean 进行配置，也就是 IOC 注入。

setBeanName 实现

3. 如果这个 Bean 已经实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String) 方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值

BeanFactoryAware 实现

4. 如果这个 Bean 已经实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory，setBeanFactory(BeanFactory)传递的是 Spring 工厂自身（可以用这个方式来获取其它 Bean，只需在 Spring 配置文件中配置一个普通的 Bean 就可以）。

ApplicationContextAware 实现

5. 如果这个 Bean 已经实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文（同样这个方式也可以实现步骤 4 的内容，但比 4 更好，因为 ApplicationContext 是 BeanFactory 的子接口，有更多的实现方法）

postProcessBeforeInitialization接口实现-初始化预处理

6. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用

postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor 经常被用

作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用那个的方法，也可以被应用于内存或缓存技术。

init-method

7. 如果 Bean 在 Spring 配置文件中配置了 init-method 属性会自动调用其配置的初始化方法。

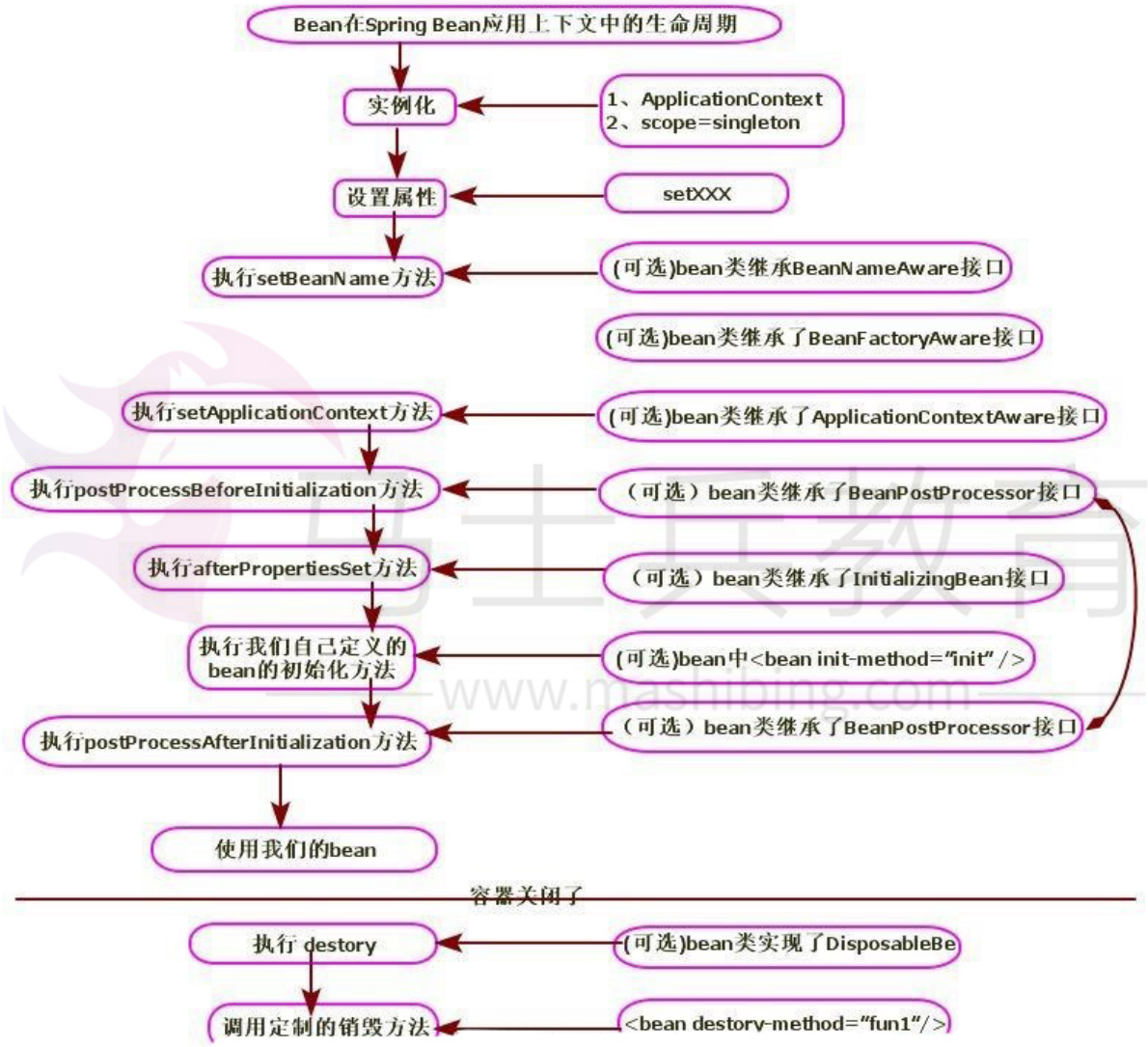
postProcessAfterInitialization

8. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法。

注：以上工作完成以后就可以应用这个 Bean 了，那这个 Bean 是一个 Singleton 的，所以一般情况下我们调用同一个 id 的 Bean 会是在内容地址相同的实例，当然在 Spring 配置文件中也可以配置非 Singleton。

Destroy 过期自动清理阶段

9. 当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 这个接口，会调用那个其实现的 destroy()方法； **destroy-method** 自配置清理
10. 最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。



11. bean 标签有两个重要的属性（init-method 和 destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct 和@PreDestroy）。

Spring 依赖注入四种方式

构造器注入

```
/*带参数，方便利用构造器进行注入*/
    public CatDaoImpl(String message){           this. message = message;
    }
<bean id="CatDaoImpl" class="com.CatDaoImpl">
<constructor-arg value=" message "></constructor-arg>
</bean>
```

setter方法注入

```
 public class Id {
 private int id;
 public int getId() {
 return id;
 }
 public void setId(int id) {
 this.id = id;
 }
}
 <bean id="id" class="com.id ">  <property name="id" value="123"></property>
</bean>
```

静态工厂注入

静态工厂顾名思义，就是通过调用静态工厂的方法来获取自己需要的对象，为了让 spring 管理所有对象，我们不能直接通过"工程类.静态方法()"来获取对象，而是依然通过 spring 注入的形式获取：

```
public class DaoFactory {   //静态工厂        public static final FactoryDao
getStaticFactoryDaoImpl(){           return new StaticFacotryDaoImpl();
    }
}
 public class SpringAction {             private FactoryDao staticFactoryDao;  //
注入对象
    //注入对象的 set 方法
    public void setStaticFactoryDao(FactoryDao staticFactoryDao) {
this.staticFactoryDao = staticFactoryDao;
    }
}
//factory-method="getStaticFactoryDaoImpl"指定调用哪个工厂方法
    <bean name="springAction" class=" SpringAction" >
        <!--使用静态工厂的方法注入对象,对应下面的配置文件-->
        <property name="staticFactoryDao" ref="staticFactoryDao"></property>
         </bean>
    <!--此处获取对象的方式是从工厂类中获取静态方法-->
<bean name="staticFactoryDao" class="DaoFactory"  factory-
method="getStaticFactoryDaoImpl"></bean>
```

实例工厂

实例工厂的意思是获取对象实例的方法不是静态的，所以你需要首先 new 工厂类，再调用普通的实例方法：

```
public class DaoFactory {   //实例工厂         public FactoryDao
getFactoryDaoImpl(){            return new FactoryDaoImpl();
    }
}
public class SpringAction {       private FactoryDao factoryDao;      //注入对象
      public void setFactoryDao(FactoryDao factoryDao) {
this.factoryDao = factoryDao;
    }
}
    <bean name="springAction" class="SpringAction">
        <!--使用实例工厂的方法注入对象,对应下面的配置文件-->
        <property name="factoryDao" ref="factoryDao"></property>       </bean>

     <!--此处获取对象的方式是从工厂类中获取实例方法-->
<bean name="daoFactory" class="com.DaoFactory"></bean>
<bean name="factoryDao" factory-bean="daoFactory"  factory-
method="getFactoryDaoImpl"></bean>
```

5 种不同方式的自动装配

Spring 装配包括手动装配和自动装配，手动装配是有基于 xml 装配、构造方法、setter 方法等自动装配有五种自动装配的方式，可以用来指导 Spring 容器用自动装配方式来进行依赖注入。

1. no：默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。
2. byName：通过参数名 自动装配，Spring容器在配置文件中发现bean的autowire 属性被设置成byname，之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。
3. byType：通过参数类型自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成byType，之后容器试图匹配、装配和该bean的属性具有相同类型的bean。如果有多个 bean 符合条件，则抛出错误。
4. constructor：这个方式类似于 byType， 但是要提供给构造器参数，如果没有确定的带参数的构造器参数类型，将会抛出异常。
5. autodetect：首先尝试使用constructor 来自动装配，如果无法工作，则使用byType方式。

Spring APO 原理

概念

"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，

并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

使用"横切"技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

AOP 主要应用场景有：

1. Authentication 权限
2. Caching 缓存
3. Context passing 内容传递
4. Error handling 错误处理
5. Lazy loading 懒加载
6. Debugging 调试

7. logging, tracing, profiling and monitoring 记录跟踪 优化 校准
8. Performance optimization 性能优化
9. Persistence 持久化
10. Resource pooling 资源池
11. Synchronization 同步
12. Transactions 事务

AOP 核心概念

1、切面（aspect）：类是对物体特征的抽象，切面就是对横切关注点的抽象

2、横切关注点：对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点。

3、连接点（joinpoint）：被拦截到的点，因为 Spring 只支持方法类型的连接点，所以在 Spring 中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器。

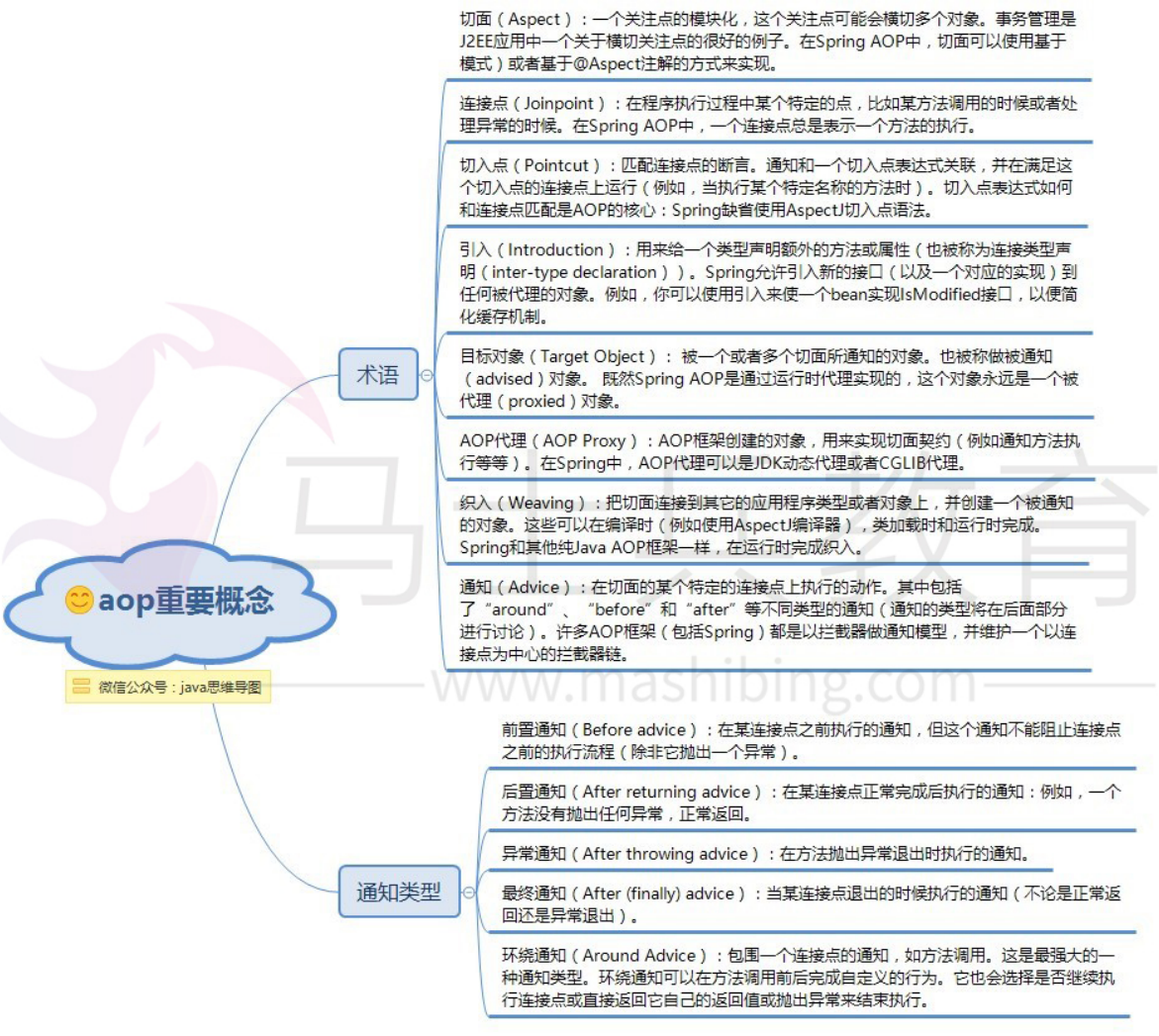
4、切入点（pointcut）：对连接点进行拦截的定义

5、通知（advice）：所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类。

6、目标对象：代理的目标对象

7、织入（weave）：将切面应用到目标对象并导致代理对象创建的过程 8、引入（introduction）：在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段。

[彻底征服 Spring AOP 之 理论篇]



AOP 两种代理方式

Spring 提供了两种方式来生成代理对象: JDKProxy 和 Cglib，具体使用哪种方式生成由 AopProxyFactory 根据 AdvisedSupport 对象的配置来决定。默认的策略是如果目标类是接口，则使用 JDK 动态代理技术，否则使用 Cglib 来生成代理。

JDK动态接口代理

1. JDK 动态代理主要涉及到 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。

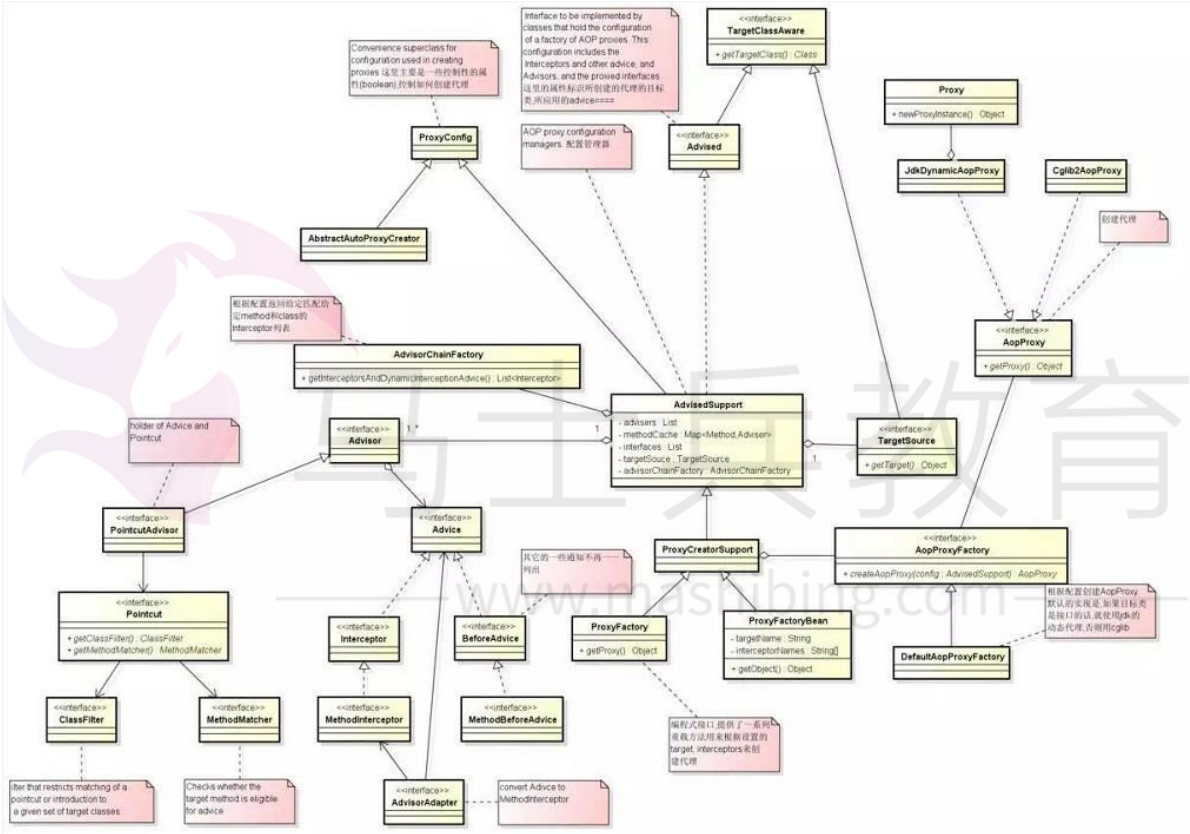
InvocationHandler是一个接口，通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态将横切逻辑和业务逻辑编制在一起。Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。

CGLib 动态代理

2. ： CGLib 全称为 Code Generation Library，是一个强大的高性能，高质量的代码生成类库，可以在运行期扩展 Java 类与实现 Java 接口，CGLib 封装了 asm，可以再运行期动态生成新的 class。和 JDK 动态代理相比较：JDK 创建代理有一个限制，就是只能为接口创建代理实例，而对于没有通过接口定义业务方法的类，则可以通过 CGLib 创建动态代理。

实现原理

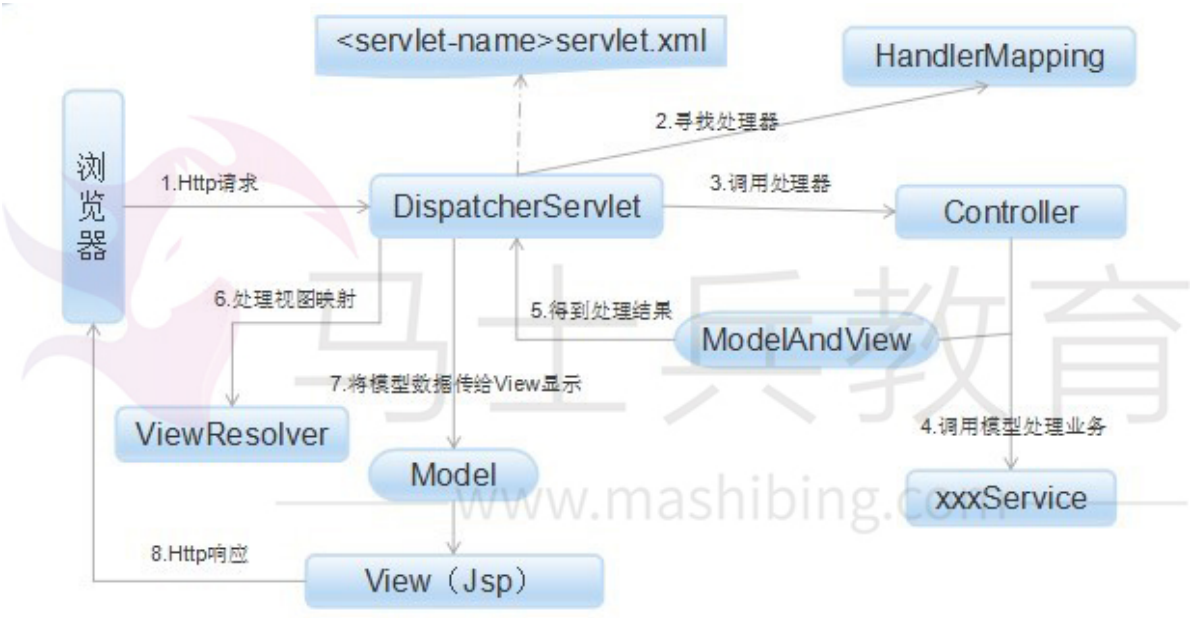
```
@Aspect public class TransactionDemo {
       @Pointcut(value="execution(* com.yangxin.core.service.*.*.*(..))")
public void point(){
      }
     @Before(value="point()")     public void before(){
         System.out.println("transaction begin");
    }
       @AfterReturning(value = "point()")     public void after(){
         System.out.println("transaction commit");
    }
        @Around("point()")
    public void around(ProceedingJoinPoint joinPoint) throws Throwable{
System.out.println("transaction begin");         joinPoint.proceed();
        System.out.println("transaction commit");
    }
}
```



Spring MVC 原理

Spring 的模型-视图-控制器（MVC）框架是围绕一个 DispatcherServlet 来设计的，这个 Servlet 会把请求分发给各个处理器，并支持可配置的处理器映射、视图渲染、本地化、时区与主题渲染等，甚至还能支持文件上传。

MVC 流程



Http 请求到**DispatcherServlet**

(1) 客户端请求提交到 DispatcherServlet。

HandlerMapping 寻找处理器

(2) 由 DispatcherServlet 控制器查询一个或多个 HandlerMapping，找到处理请求的 Controller。

调用处理器**Controller**

(3) DispatcherServlet 将请求提交到 Controller。

Controller 调用业务逻辑处理后，返回**ModelAndView**

(4)(5)调用业务处理和返回结果：Controller 调用业务逻辑处理后，返回 ModelAndView。

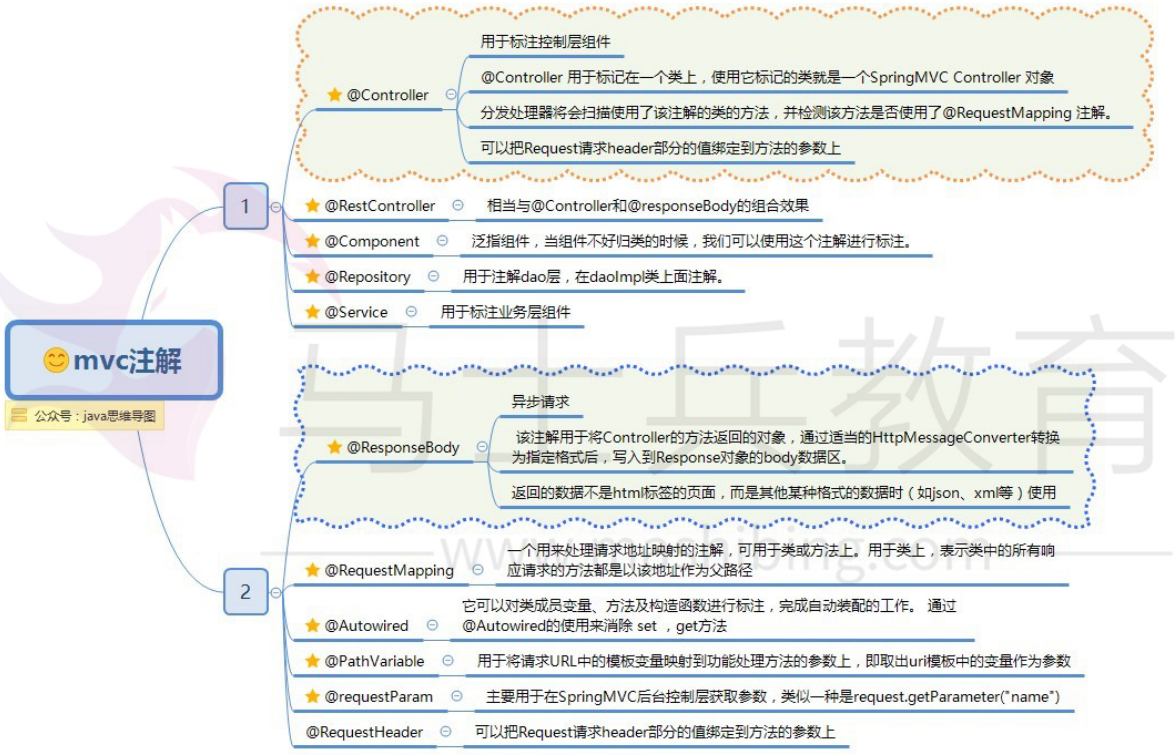
DispatcherServlet 查询**ModelAndView**

(6)(7)处理视图映射并返回模型： DispatcherServlet 查询一个或多个 ViewResoler 视图解析器，找到 ModelAndView 指定的视图。

ModelAndView 反馈浏览器**HTTP**

(8) Http 响应：视图负责将结果显示到客户端。

MVC 常用注解



Spring AOP的实现原理和场景；

AOP（Aspect Orient Programming），作为面向对象编程的一种补充，广泛应用于处理一 些具有横切性质的系统级服务。

一、场景

事务管理、安全检查、权限控制、数据校验、缓存、对象池管理等

二、实现技术

AOP（这里的AOP指的是面向切面编程思想，而不是Spring AOP）主要的的实现技术主要有 Spring AOP和AspectJ。

1）AspectJ的底层技术。 AspectJ的底层技术是静态代理，即用一种AspectJ支持的特定语言编写切面，通过一个命令来 编译，生成一个新的代理类，该代理类增强了业务类，这是在编译时增强，相对于下面说的运行 时增强，编译时增强的性能更好。

2）Spring AOP　Spring AOP采用的是动态代理，在运行期间对业务方法进行增强，所以不会生成新类，对于 动态代理技术，Spring AOP提供了对JDK动态代理的支持以及CGLib的支持。

JDK动态代理只能为接口创建动态代理实例，而不能对类创建动态代理。需要获得被目标类的 接口信息（应用Java的反射技术），生成一个实现了代理接口的动态代理类（字节码），再通过 反射机制获得动态代理类的构造函数，利用构造函数生成动态代理类的实例对象，在调用具体方 法前调用invokeHandler方法来处理。

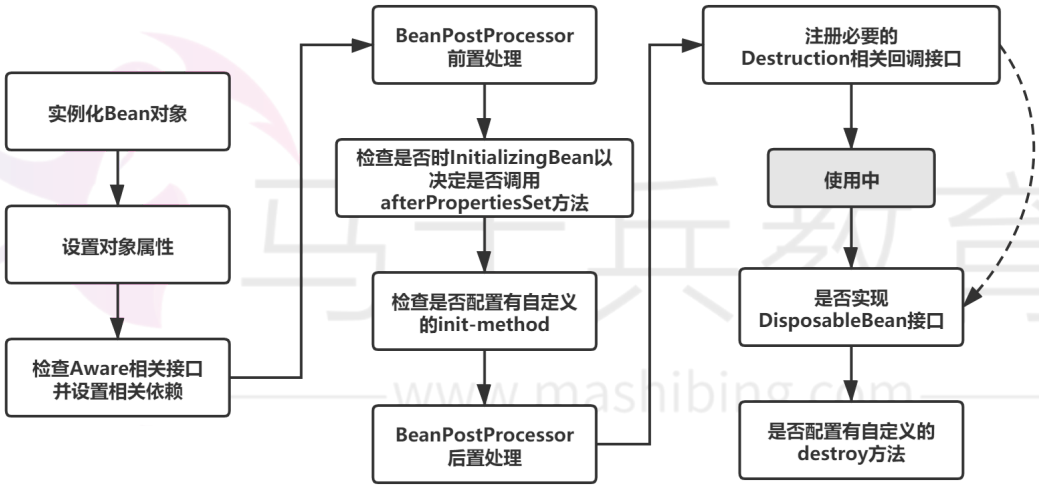
CGLib动态代理需要依赖asm包，把被代理对象类的class文件加载进来，修改其字节码生成 子类。　但是Spring AOP基于注解配置的情况下，需要依赖于AspectJ包的标准注解。

Spring bean的作用域和生命周期；

作用域

类别	说明
singleton	在Spring IOC容器中仅存在一个Bean实例,Bean以单例方式存在,默认值
prototype	每次从容器中调用Bean时,都返回一个新的实例,级每次调用getBean()时,相当于执行new XxxBean()
request	每次HTTP请求都会创建一个新的Bean,该作用域仅适用于WebApplicationContext环境
session	同一个HTTP Session共享一个Bean,不同Session使用不同Bean,仅适用于WebApplicationContext环境
globalSesison	一般用域Porlet应用环境,该作用域仅适用于WebApplicationContext环境

生命周期



Spring 5比Spring4做了哪些改进；

官网说明

Spring 4.x新特性

1. 泛型限定式依赖注入
2. 核心容器的改进
3. web开发增强
4. 集成Bean Validation 1.1（JSR-349）到SpringMVC
5. Groovy Bean定义DSL
6. 更好的Java泛型操作API
7. JSR310日期API的支持
8. 注解、脚本、任务、MVC等其他特性改进

Spring 5.x新特性

1. JDK8的增强
2. 核心容器的改进
3. 新的SpringWebFlux模块
4. 测试方面的改进

Spring FrameWork 5.0新的功能

JDK 8+和Java EE7+以上版本

- 整个框架的代码基于java8
- 通过使用泛型等特性提高可读性
- 对java8提高直接的代码支撑
- 运行时兼容JDK9
- Java EE 7API需要Spring相关的模块支持
- 运行时兼容Java EE8 API
- 取消的包,类和方法
- 包 beans.factory.access
- 包 dbc.support.nativejdbc
- 从spring-aspects 模块移除了包mock.staicmock,不在提AnnotationDrivenStaticEntityMockingControl支持
- 许多不建议使用的类和方法在代码库中删除

核心特性

JDK8的增强：

- 访问Resuouce时提供getFile或和isFile防御式抽象
- 有效的方法参数访问基于java 8反射增强
- 在Spring核心接口中增加了声明default方法的支持一贯使用JDK7 Charset和StandardCharsets的增强
- 兼容JDK9
- Spring 5.0框架自带了通用的日志封装
- 持续实例化via构造函数(修改了异常处理)

- Spring 5.0框架自带了通用的日志封装
- spring-jcl替代了通用的日志，仍然支持可重写
- 自动检测log4j 2.x, SLF4J, JUL（java.util.Logging）而不是其他的支持
- 访问Resuouce时提供getFile或和isFile防御式抽象
- 基于NIO的readableChannel也提供了这个新特性

核心容器

- 支持候选组件索引(也可以支持环境变量扫描)
- 支持@Nullable注解
- 函数式风格GenericApplicationContext/AnnotationConfigApplicationContext
- 基本支持bean API注册
- 在接口层面使用CGLIB动态代理的时候，提供事物，缓存，异步注解检测
- XML配置作用域流式
- Spring WebMVC
- 全部的Servlet 3.1 签名支持在Spring-provied Filter实现
- 在Spring MVC Controller方法里支持Servlet4.0 PushBuilder参数
- 多个不可变对象的数据绑定(Kotlin/Lombok/@ConstructorPorties)
- 支持jackson2.9
- 支持JSON绑定API
- 支持protobuf3
- 支持Reactor3.1 Flux和Mono

SpringWebFlux

- 新的spring-webflux模块，一个基于reactive的spring-webmvc，完全的异步非阻塞，旨在使用enent-loop执行模型和传统的线程池模型。
- Reactive说明在spring-core比如编码和解码
- spring-core相关的基础设施，比如Encode 和Decoder可以用来编码和解码数据流；DataBuffer 可以使用java ByteBuffer或者Netty ByteBuf;ReactiveAdapterRegistry可以对相关的库提供传输层支持。
- 在spring-web包里包含HttpMessageReade和HttpMessageWrite

测试方面的改进

- 完成了对JUnit 5’s Juptier编程和拓展模块在Spring TestContext框架
- SpringExtension:是JUnit多个可拓展API的一个实现，提供了对现存Spring TestContext Framework的支持，使用@ExtendWith(SpringExtension.class)注解引用。
- @SpringJunitConfig:一个复合注解
- @ExtendWith(SpringExtension.class) 来源于Junit Jupit
- @ContextConfiguration 来源于Srping TestContext框架
- @DisabledIf 如果提供的该属性值为true的表达或占位符，信号：注解的测试类或测试方法被禁用
- 在Spring TestContext框架中支持并行测试
- 具体细节查看Test 章节 通过SpringRunner在Sring TestContext框架中支持TestNG, Junit5,新的执行之前和之后测试回调。

- 在testexecutionlistener API和testcontextmanager新beforetestexecution()和aftertestexecution()回调。MockHttpServletRequest新增了getContentAsByteArray()和getContentAsString()方法来访问请求体
- 如果字符编码被设置为mock请求，在print()和log()方法中可以打印Spring MVC Test的redirectedUrl()和forwardedUrl()方法支持带变量表达式URL模板。
- XMLUnit 升级到了2.3版本。

Spring4新特性——泛型限定式依赖注入

Spring 4.0已经发布RELEASE版本，不仅支持Java8，而且向下兼容到JavaSE6/JavaEE6，并移出了相关废弃类，新添加如Java8的支持、Groovy式Bean定义DSL、对核心容器进行增强、对Web框架的增强、Websocket模块的实现、测试的增强等。其中两个我一直想要的增强就是：支持泛型依赖注入、对cglib类代理不再要求必须有空参构造器了。

1、相关代码：

1.1、实体

```
public class User implements Serializable {
    private Long id;
    private String name;
}

public class Organization implements Serializable {
    private Long id;
    private String name;
}
```

1.2、Repository

```
public abstract class BaseRepository<M extends Serializable> {
    public void save(M m) {
        System.out.println("=====repository save:" + m);
    }
}

@Repository
public class UserRepository extends BaseRepository<User> {
}

@Repository
public class OrganizationRepository extends BaseRepository<Organization> {
}
```

对于Repository，我们一般是这样实现的：首先写一个模板父类，把通用的crud等代码放在BaseRepository；然后子类继承后，只需要添加额外的实现。

1.3、Service

1.3.1、以前Service写法

```
public abstract class BaseService<M extends Serializable> {
    private BaseRepository<M> repository;
    public void setRepository(BaseRepository<M> repository) {
        this.repository = repository;
    }
    public void save(M m) {
        repository.save(m);
    }
}
@Service
public class UserService extends BaseService<User> {
    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        setRepository(userRepository);
    }
}

@Service
public class OrganizationService extends BaseService<Organization> {
    @Autowired
    public void setOrganizationRepository(OrganizationRepository
organizationRepository) {
        setRepository(organizationRepository);
    }
}
```

可以看到，以前必须再写一个setter方法，然后指定注入的具体类型，然后进行注入；

1.3.2、泛型Service的写法

```
public abstract class BaseService<M extends Serializable> {
    @Autowired
    protected BaseRepository<M> repository;

    public void save(M m) {
        repository.save(m);
    }
}

@Service
public class UserService extends BaseService<User> {
}

@Service
public class OrganizationService extends BaseService<Organization> {
}
```

大家可以看到，现在的写法非常简洁。支持泛型式依赖注入。

这个也是我之前非常想要的一个功能，这样对于那些基本的CRUD式代码，可以简化更多的代码。

如果大家用过Spring data jpa的话，以后注入的话也可以使用泛型限定式依赖注入：

```
@Autowired
private Repository<User> userRepository;
```

对于泛型依赖注入，最好使用setter注入，这样万一子类想变，比较容易切换。如果有多个实现时，子类可以使用@Qualifier指定使用哪一个。

Spring4新特性——核心容器的其他改进

1、Map依赖注入：

```
@Autowired
private Map<String, BaseService> map;
```

这样会注入：key是bean名字；value就是所有实现了BaseService的Bean，假设使用上一篇的例子，则会得到：

{organizationService=com.sishuok.spring4.service.OrganizationService@617029,
userService=com.sishuok.spring4.service.UserService@10ac73b}

2、List/数组注入：

```
@Autowired
private List<BaseService> list;
```

这样会注入所有实现了BaseService的Bean；但是顺序是不确定的，如果我们想要按照某个顺序获取；在Spring4中可以使用@Order或实现Ordered接口来实现，如：

```
@Order(value = 1)
@Service
public class UserService extends BaseService<User> {
}
```

这种方式在一些需要多态的场景下是非常有用的。

3、@Lazy可以延迟依赖注入：

```
@Lazy
@Service
public class UserService extends BaseService<User> {
}
```

```
    @Lazy
    @Autowired
    private UserService userService;
```

我们可以把@Lazy放在@Autowired之上，即依赖注入也是延迟的；当我们调用userService时才会注入。即延迟依赖注入到使用时。同样适用于@Bean。

4、@Conditional

@Conditional类似于`@Profile（一般用于如我们有开发环境、测试环境、正式机环境，为了方便切换不同的环境可以使用`@Profile指定各个环境的配置，然后通过某个配置来开启某一个环境，方便切换`）`，但是@Conditional的优点是允许自己定义规则。可以指定在如@Component、@Bean、@Configuration等注解的类上，以绝对Bean是否创建等。首先来看看使用@Profile的用例，假设我们有个用户模块：

```
public abstract class UserService extends BaseService<User> {
}

@Profile("local")
@Service
public class LocalUserService extends UserService {
}

@Profile("remote")
@Service
public class RemoteUserService extends UserService {
}
```

我们在写测试用例时，可以指定我们使用哪个Profile：

```
@ActiveProfiles("remote")
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations =  "classpath:spring-config.xml")
public class ServiceTest {

    @Autowired
    private UserService userService;
}
```

这种方式非常简单。如果想自定义如@Profile之类的注解等，那么@Conditional就派上用场了；假设我们系统中有好多本地/远程接口，那么我们定义两个注解@Local和@Remote注解要比使用@Profile方便的多；如：

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Conditional(CustomCondition.class)
public @interface Local {
}

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Conditional(CustomCondition.class)
public @interface Remote {
}
```

```
public class CustomCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
        boolean isLocalBean =
metadata.isAnnotated("com.sishuok.spring4.annotation.Local");
        boolean isRemoteBean =
metadata.isAnnotated("com.sishuok.spring4.annotation.Remote");
        //如果bean没有注解@Local或@Remote，返回true，表示创建Bean
        if(!isLocalBean && !isRemoteBean) {
            return true;
        }

        boolean isLocalProfile =
context.getEnvironment().acceptsProfiles("local");

        //如果profile=local 且 bean注解了@Local，则返回true 表示创建bean；
        if(isLocalProfile) {
            return isLocalBean;
        }

        //否则默认返回注解了@Remote或没有注解@Remote的Bean
        return isRemoteBean;
    }
}
```

然后我们使用这两个注解分别注解我们的Service：

```
@Local
@Service
public class LocalUserService extends UserService {
}
```

```
@Remote
@Service
public class RemoteUserService extends UserService {
}
```

首先在@Local和@Remote注解上使用@Conditional(CustomCondition.class)指定条件，然后使用@Local和@Remote注解我们的Service，这样当加载Service时，会先执行条件然后判断是否加载为Bean。@Profile就是这样实现的，其Condition是：org.springframework.context.annotation.ProfileCondition。可以去看下源码，很简单。

5、基于CGLIB的类代理不再要求类必须有空参构造器了：

这是一个很好的特性，使用构造器注入有很多好处，比如可以只在创建Bean时注入依赖，然后就不变了，如果使用setter注入，是允许别人改的。当然我们可以使用spring的字段级别注入。如果大家使用过如Shiro，我们可能要对Controller加代理。如果是类级别代理，此时要求Controller必须有空参构造器，有时候挺烦人的。spring如何实现的呢？其内联了objenesis类库，通过它来实现，可以去其官网看看介绍。这样就支持如下方式的构造器注入了：

```
@Controller
public class UserController {
    private UserService userService;
    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
}
```

Spring4新特性——Web开发的增强

从Spring4开始，Spring以Servlet3为进行开发，如果用Spring MVC 测试框架的话需要指定Servlet3兼容的jar包（因为其Mock的对象都是基于Servlet3的）。另外为了方便Rest开发，通过新的@RestController指定在控制器上，这样就不需要在每个@RequestMapping方法上加`@ResponseBody了。而且添加了一个``AsyncRestTemplate`，支持REST客户端的异步无阻塞支持。

```
@RestController
public class UserController {
    private UserService userService;
    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
    @RequestMapping("/test")
      public User view() {
        User user = new User();
        user.setId(1L);
        user.setName("haha");
        return user;
    }

    @RequestMapping("/test2")
```

```
    public String view2() {
        return "{\"id\" : 1}";
    }
}
```

其实现就是在@@RestController中加入@ResponseBody：

```
@org.springframework.stereotype.Controller
@org.springframework.web.bind.annotation.ResponseBody
public @interface RestController {
}
```

这样当你开发Rest服务器端的时候，spring-mvc配置文件需要的代码极少，可能就仅需如下一行：

```
    <context:component-scan base-package="com.sishuok.spring4"/>
    <mvc:annotation-driven/>
```

2、mvc:annotation-driven配置变化

统一风格；将 enableMatrixVariables改为enable-matrix-variables属性；将 ignoreDefaultModelOnRedirect改为ignore-default-model-on-redirect。

3、提供AsyncRestTemplate用于客户端非阻塞异步支持。

3.1、服务器端

对于服务器端的springmvc开发

```
@RestController
public class UserController {
    private UserService userService;
    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
    @RequestMapping("/api")
      public Callable<User> api() {
        System.out.println("=====hello");
        return new Callable<User>() {
            @Override
            public User call() throws Exception {
                Thread.sleep(10L * 1000); //暂停两秒
                User user = new User();
                user.setId(1L);
                user.setName("haha");
                return user;
            }
        };
    }
}
```

非常简单，服务器端暂停10秒再返回结果（但是服务器也是非阻塞的）。具体参考我github上的代码。

3.2、客户端

```
    public static void main(String[] args) {
        AsyncRestTemplate template = new AsyncRestTemplate();
        //调用完后立即返回（没有阻塞）
        ListenableFuture<ResponseEntity<User>> future =
template.getForEntity("http://localhost:9080/spring4/api", User.class);
        //设置异步回调
        future.addCallback(new ListenableFutureCallback<ResponseEntity<User>>()
{
            @Override
            public void onSuccess(ResponseEntity<User> result) {
                System.out.println("======client get result : " +
result.getBody());
            }

            @Override
            public void onFailure(Throwable t) {
                System.out.println("======client failure : " + t);
            }
        });
        System.out.println("==no wait");
    }
```

此处使用Future来完成非阻塞，这样的话我们也需要给它一个回调接口来拿结果； Future和Callable是一对，一个消费结果，一个产生结果。调用完模板后会立即返回，不会阻塞；有结果时会调用其回调。

AsyncRestTemplate默认使用SimpleClientHttpRequestFactory，即通过java.net.HttpURLConnection实现；另外我们也可以使用apache的http components；使用template.setAsyncRequestFactory(new HttpComponentsAsyncClientHttpRequestFactory());设置即可。

另外在开发时尽量不要自己注册如：

```
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandl
erMapping"/>
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandl
erAdapter">
```

尽量使用

```
<mvc:annotation-driven/>
```

它设计的已经足够好，使用子元素可以配置我们需要的配置。

且不要使用老版本的：

```
<bean
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMa
pping"/>
<bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAda
pter">
```

否则可能得到如下异常：

写道

Circular view path [login]: would dispatch back to the current handler URL [/spring4/login] again. Check your ViewResolver setup! (Hint: This may be the result of an unspecified view, due to default view name generation.)

Spring4新特性——集成Bean Validation 1.1(JSR-349)到SpringMVC

Bean Validation 1.1当前实现是Hibernate validator 5，且spring4才支持。接下来我们从以下几个方法讲解Bean Validation 1.1，当然不一定是新特性：

1. 集成Bean Validation 1.1到SpringMVC
2. 分组验证、分组顺序及级联验证
3. 消息中使用EL表达式
4. 方法参数/返回值验证
5. 自定义验证规则
6. 类级别验证器
7. 脚本验证器
8. cross-parameter，跨参数验证
9. 混合类级别验证器和跨参数验证器
10. 组合多个验证注解
11. 本地化

因为大多数时候验证都配合web框架使用，而且很多朋友都咨询过如分组/跨参数验证，所以本文介绍下这些，且是和SpringMVC框架集成的例子，其他使用方式（比如集成到JPA中）可以参考其官方文档：

规范：http://beanvalidation.org/1.1/spec/

hibernate validator文档：http://hibernate.org/validator/

1、集成Bean Validation 1.1到SpringMVC

1.1、项目搭建

首先添加hibernate validator 5依赖：

```
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>5.0.2.Final</version>
        </dependency>
```

如果想在消息中使用EL表达式，请确保EL表达式版本是 2.2或以上，如使用Tomcat6，请到Tomcat7中拷贝相应的EL jar包到Tomcat6中。

```
        <dependency>
            <groupId>javax.el</groupId>
            <artifactId>javax.el-api</artifactId>
            <version>2.2.4</version>
            <scope>provided</scope>
        </dependency>
```

请确保您使用的Web容器有相应版本的el jar包。

对于其他POM依赖请下载附件中的项目参考。

1.2、Spring MVC配置文件（spring-mvc.xml）：

```
    <!-- 指定自己定义的validator -->
    <mvc:annotation-driven validator="validator"/>

    <!-- 以下 validator  ConversionService 在使用 mvc:annotation-driven 会 自动注册-
->
    <bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
        <property name="providerClass"
value="org.hibernate.validator.HibernateValidator"/>
        <!-- 如果不加默认到 使用classpath下的 ValidationMessages.properties -->
        <property name="validationMessageSource" ref="messageSource"/>
    </bean>

    <!-- 国际化的消息资源文件（本系统中主要用于显示/错误消息定制） -->
    <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource
">
        <property name="basenames">
            <list>
                <!-- 在web环境中一定要定位到classpath 否则默认到当前web应用下找  -->
                <value>classpath:messages</value>

<value>classpath:org/hibernate/validator/ValidationMessages</value>
            </list>
        </property>
        <property name="useCodeAsDefaultMessage" value="false"/>
        <property name="defaultEncoding" value="UTF-8"/>
        <property name="cacheSeconds" value="60"/>
    </bean>
```

此处主要把bean validation的消息查找委托给spring的messageSource。

1.3、实体验证注解：

```
public class User implements Serializable {
    @NotNull(message = "{user.id.null}")
    private Long id;

    @NotEmpty(message = "{user.name.null}")
    @Length(min = 5, max = 20, message = "{user.name.length.illegal}")
    @Pattern(regexp = "[a-zA-Z]{5,20}", message = "{user.name.illegal}")
    private String name;

    @NotNull(message = "{user.password.null}")
    private String password;
}
```

对于验证规则可以参考官方文档，或者《第七章 注解式控制器的数据验证、类型转换及格式化》。

1.4、错误消息文件messages.properties：

```
user.id.null=用户编号不能为空
user.name.null=用户名不能为空
user.name.length.illegal=用户名长度必须在5到20之间
user.name.illegal=用户名必须是字母
user.password.null=密码不能为空
```

1.5、控制器

```
@Controller
public class UserController {

    @RequestMapping("/save")
    public String save(@Valid User user, BindingResult result) {
        if(result.hasErrors()) {
            return "error";
        }
        return "success";
    }
}
```

1.6、错误页面：

```
<spring:hasBindErrors name="user">
    <c:if test="${errors.fieldErrorCount > 0}">
        字段错误：<br/>
        <c:forEach items="${errors.fieldErrors}" var="error">
            <spring:message var="message" code="${error.code}"
arguments="${error.arguments}" text="${error.defaultMessage}"/>
            ${error.field}------${message}<br/>
        </c:forEach>
    </c:if>

    <c:if test="${errors.globalErrorCount > 0}">
```

```
        全局错误：<br/>
        <c:forEach items="${errors.globalErrors}" var="error">
            <spring:message var="message" code="${error.code}"
arguments="${error.arguments}" text="${error.defaultMessage}"/>
            <c:if test="${not empty message}">
                ${message}<br/>
            </c:if>
        </c:forEach>
    </c:if>
</spring:hasBindErrors>
```

大家以后可以根据这个做通用的错误消息显示规则。比如我前端页面使用validationEngine显示错误消息，那么我可以定义一个tag来通用化错误消息的显示：showFieldError.tag。

1.7、测试

输入如：http://localhost:9080/spring4/save?name=123 ， 我们得到如下错误：

```
name------用户名必须是字母
name------用户名长度必须在5到20之间
password------密码不能为空
id------用户编号不能为空
```

基本的集成就完成了。

如上测试有几个小问题：

1、错误消息顺序，大家可以看到name的错误消息顺序不是按照书写顺序的，即不确定；

2、我想显示如：用户名【zhangsan】必须在5到20之间；其中我们想动态显示：用户名、min，max；而不是写死了；

3、我想在修改的时候只验证用户名，其他的不验证怎么办。

接下来我们挨着试试吧。

2、分组验证及分组顺序

如果我们想在新增的情况验证id和name，而修改的情况验证name和password，怎么办？ 那么就需要分组了。

首先定义分组接口：

```
public interface First {
}

public interface Second {
}
```

分组接口就是两个普通的接口，用于标识，类似于java.io.Serializable。

接着我们使用分组接口标识实体：

```
public class User implements Serializable {

    @NotNull(message = "{user.id.null}", groups = {First.class})
    private Long id;

    @Length(min = 5, max = 20, message = "{user.name.length.illegal}", groups =
{Second.class})
    @Pattern(regexp = "[a-zA-Z]{5,20}", message = "{user.name.illegal}", groups =
{Second.class})
    private String name;

    @NotNull(message = "{user.password.null}", groups = {First.class,
Second.class})
    private String password;
}
```

验证时使用如：

```
    @RequestMapping("/save")
    public String save(@Validated({Second.class}) User user, BindingResult
result) {
        if(result.hasErrors()) {
            return "error";
        }
        return "success";
    }
```

即通过@Validate注解标识要验证的分组；如果要验证两个的话，可以这样@Validated({First.class, Second.class})。

接下来我们来看看通过分组来指定顺序；还记得之前的错误消息吗？ user.name会显示两个错误消息，而且顺序不确定；如果我们先验证一个消息；如果不通过再验证另一个怎么办？可以通过@GroupSequence指定分组验证顺序：

```
@GroupSequence({First.class, Second.class, User.class})
public class User implements Serializable {
    private Long id;

    @Length(min = 5, max = 20, message = "{user.name.length.illegal}", groups =
{First.class})
    @Pattern(regexp = "[a-zA-Z]{5,20}", message = "{user.name.illegal}", groups =
{Second.class})
    private String name;

    private String password;
}
```

通过@GroupSequence指定验证顺序：先验证First分组，如果有错误立即返回而不会验证Second分组，接着如果First分组验证通过了，那么才去验证Second分组，最后指定User.class表示那些没有分组的在最后。这样我们就可以实现按顺序验证分组了。

另一个比较常见的就是级联验证：

如：

```
public class User {

    @valid
    @ConvertGroup(from=First.class, to=Second.class)
    private Organization o;

}
```

1、级联验证只要在相应的字段上加@Valid即可，会进行级联验证；@ConvertGroup的作用是当验证o的分组是First时，那么验证o的分组是Second，即分组验证的转换。

3、消息中使用EL表达式

假设我们需要显示如：用户名[NAME]长度必须在[MIN]到[MAX]之间，此处大家可以看到，我们不想把一些数据写死，如NAME、MIN、MAX；此时我们可以使用EL表达式。

如：

```
@Length(min = 5, max = 20, message = "{user.name.length.illegal}", groups =
{First.class})
```

错误消息：

```
user.name.length.illegal=用户名长度必须在{min}到{max}之间
```

其中我们可以使用{验证注解的属性}得到这些值；如{min}得到@Length中的min值；其他的也是类似的。

到此，我们还是无法得到出错的那个输入值，如name=zhangsan。此时就需要EL表达式的支持，首先确定引入EL jar包且版本正确。然后使用如：

```
user.name.length.illegal=用户名[${validatedValue}]长度必须在5到20之间
```

使用如EL表达式：${validatedValue}得到输入的值，如zhangsan。当然我们还可以使用如${min > 1 ? '大于1' : '小于等于1'}，及在EL表达式中也能拿到如@Length的min等数据。

另外我们还可以拿到一个java.util.Formatter类型的formatter变量进行格式化：

```
${formatter.format("%04d", min)}
```

4、方法参数/返回值验证

这个可以参考《Spring3.1 对Bean Validation规范的新支持(方法级别验证)》，概念是类似的，具体可以参考Bean Validation 文档。

5、自定义验证规则

有时候默认的规则可能还不够，有时候还需要自定义规则，比如屏蔽关键词验证是非常常见的一个功能，比如在发帖时帖子中不允许出现admin等关键词。

1、定义验证注解

```
package com.sishuok.spring4.validator;

import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
/**
 * <p>User: Zhang Kaitao
 * <p>Date: 13-12-15
 * <p>Version: 1.0
 */

@Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
@Retention(RUNTIME)
//指定验证器
@Constraint(validatedBy = ForbiddenValidator.class)
@Documented
public @interface Forbidden {

    //默认错误消息
    String message() default "{forbidden.word}";

    //分组
    Class<?>[] groups() default { };

    //负载
    Class<? extends Payload>[] payload() default { };

    //指定多个时使用
    @Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Forbidden[] value();
    }
```

```
}
```

2、 定义验证器

```
package com.sishuok.spring4.validator;

import
org.hibernate.validator.internal.engine.constraintvalidation.ConstraintValidator
ContextImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.util.StringUtils;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import java.io.Serializable;

/**
 * <p>User: Zhang Kaitao
 * <p>Date: 13-12-15
 * <p>Version: 1.0
 */
public class ForbiddenValidator implements ConstraintValidator<Forbidden, String>
{

    private String[] forbiddenWords = {"admin"};

    @Override
    public void initialize(Forbidden constraintAnnotation) {
        //初始化，得到注解数据
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if(StringUtils.isEmpty(value)) {
            return true;
        }

        for(String word : forbiddenWords) {
            if(value.contains(word)) {
                return false;//验证失败
            }
        }
        return true;
    }
}
```

验证器中可以使用spring的依赖注入，如注入：@Autowired private ApplicationContext ctx;

3、使用

```
public class User implements Serializable {
    @Forbidden()
    private String name;
}
```

4、当我们在提交name中含有admin的时候会输出错误消息：

```
forbidden.word=您输入的数据中有非法关键词
```

问题来了，哪个词是非法的呢？bean validation 和 hibernate validator都没有提供相应的api提供这个数据，怎么办呢？通过跟踪代码，发现一种不是特别好的方法：我们可以覆盖org.hibernate.validator.internal.metadata.descriptor.ConstraintDescriptorImpl实现（即复制一份代码放到我们的src中），然后覆盖buildAnnotationParameterMap方法；

```
    private Map<String, Object> buildAnnotationParameterMap(Annotation
annotation) {
        ……
        //将Collections.unmodifiableMap( parameters );替换为如下语句
        return parameters;
    }
```

即允许这个数据可以修改；然后在ForbiddenValidator中：

```
        for(String word : forbiddenWords) {
            if(value.contains(word)) {

((ConstraintValidatorContextImpl)context).getConstraintDescriptor().getAttribute
s().put("word", word);
                return false;//验证失败
            }
        }
```

通过((ConstraintValidatorContextImpl)context).getConstraintDescriptor().getAttributes().put("word", word);添加自己的属性；放到attributes中的数据可以通过${} 获取。然后消息就可以变成：

```
forbidden.word=您输入的数据中有非法关键词【{word}】
```

这种方式不是很友好，但是可以解决我们的问题。

典型的如密码、确认密码的场景，非常常用；如果没有这个功能我们需要自己写代码来完成；而且经常重复自己。接下来看看bean validation 1.1如何实现的。

6、类级别验证器

6.1、定义验证注解

```
package com.sishuok.spring4.validator;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
/**
 * <p>User: Zhang Kaitao
 * <p>Date: 13-12-15
 * <p>Version: 1.0
 */

@Target({ TYPE, ANNOTATION_TYPE})
@Retention(RUNTIME)
//指定验证器
@Constraint(validatedBy = CheckPasswordValidator.class)
@Documented
public @interface CheckPassword {

    //默认错误消息
    String message() default "";

    //分组
    Class<?>[] groups() default { };

    //负载
    Class<? extends Payload>[] payload() default { };

    //指定多个时使用
    @Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        CheckPassword[] value();
    }
}
```

6.2、 定义验证器

```
package com.sishuok.spring4.validator;

import com.sishuok.spring4.entity.User;
import org.springframework.util.StringUtils;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

/**
 * <p>User: Zhang Kaitao
```

```
 * <p>Date: 13-12-15
 * <p>Version: 1.0
 */
public class CheckPasswordValidator implements ConstraintValidator<CheckPassword,
User> {

    @Override
    public void initialize(CheckPassword constraintAnnotation) {
    }

    @Override
    public boolean isValid(User user, ConstraintValidatorContext context) {
        if(user == null) {
            return true;
        }

        //没有填密码
        if(!StringUtils.hasText(user.getPassword())) {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate("{password.null}")
                    .addPropertyNode("password")
                    .addConstraintViolation();
            return false;
        }

        if(!StringUtils.hasText(user.getConfirmation())) {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate("
{password.confirmation.null}")
                    .addPropertyNode("confirmation")
                    .addConstraintViolation();
            return false;
        }

        //两次密码不一样
        if (!user.getPassword().trim().equals(user.getConfirmation().trim())) {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate("
{password.confirmation.error}")
                    .addPropertyNode("confirmation")
                    .addConstraintViolation();
            return false;
        }
        return true;
    }
}
```

其中我们通过disableDefaultConstraintViolation禁用默认的约束；然后通过buildConstraintViolationWithTemplate(消息模板)/addPropertyNode(所属属性)/addConstraintViolation定义我们自己的约束。

6.3、使用

```
@CheckPassword()
public class User implements Serializable {
}
```

放到类头上即可。

7、通过脚本验证

```
@ScriptAssert(script = "_this.password==_this.confirmation", lang = "javascript",
alias = "_this", message = "{password.confirmation.error}")
public class User implements Serializable {
}
```

通过脚本验证是非常简单而且强大的，lang指定脚本语言（请参考javax.script.ScriptEngineManager JSR-223），alias是在脚本验证中User对象的名字，但是大家会发现一个问题：错误消息怎么显示呢？在springmvc 中会添加到全局错误消息中，这肯定不是我们想要的，我们改造下吧。

7.1、定义验证注解

```
package com.sishuok.spring4.validator;

import
org.hibernate.validator.internal.constraintvalidators.ScriptAssertValidator;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({ TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {PropertyScriptAssertValidator.class})
@Documented
public @interface PropertyScriptAssert {

    String message() default "
{org.hibernate.validator.constraints.ScriptAssert.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    String lang();

    String script();
```

```
    String alias() default "_this";

    String property();

    @Target({ TYPE })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
        PropertyScriptAssert[] value();
    }
}
```

和ScriptAssert没什么区别，只是多了个property用来指定出错后给实体的哪个属性。

7.2、验证器

```
package com.sishuok.spring4.validator;

import javax.script.ScriptException;
import javax.validation.ConstraintDeclarationException;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

import com.sishuok.spring4.validator.PropertyScriptAssert;
import org.hibernate.validator.constraints.ScriptAssert;
import org.hibernate.validator.internal.util.Contracts;
import org.hibernate.validator.internal.util.logging.Log;
import org.hibernate.validator.internal.util.logging.LoggerFactory;
import org.hibernate.validator.internal.util.scriptengine.ScriptEvaluator;
import
org.hibernate.validator.internal.util.scriptengine.ScriptEvaluatorFactory;

import static org.hibernate.validator.internal.util.logging.Messages.MESSAGES;

public class PropertyScriptAssertValidator implements
ConstraintValidator<PropertyScriptAssert, Object> {

    private static final Log log = LoggerFactory.make();

    private String script;
    private String languageName;
    private String alias;
    private String property;
    private String message;

    public void initialize(PropertyScriptAssert constraintAnnotation) {
        validateParameters( constraintAnnotation );

        this.script = constraintAnnotation.script();
        this.languageName = constraintAnnotation.lang();
        this.alias = constraintAnnotation.alias();
        this.property = constraintAnnotation.property();
        this.message = constraintAnnotation.message();
    }
```

```
    public boolean isValid(Object value, ConstraintValidatorContext
constraintValidatorContext) {

        Object evaluationResult;
        ScriptEvaluator scriptEvaluator;

        try {
            ScriptEvaluatorFactory evaluatorFactory =
ScriptEvaluatorFactory.getInstance();
            scriptEvaluator = evaluatorFactory.getScriptEvaluatorByLanguageName(
languageName );
        }
        catch ( ScriptException e ) {
            throw new ConstraintDeclarationException( e );
        }

        try {
            evaluationResult = scriptEvaluator.evaluate( script, value, alias );
        }
        catch ( ScriptException e ) {
            throw log.getErrorDuringScriptExecutionException( script, e );
        }

        if ( evaluationResult == null ) {
            throw log.getScriptMustReturnTrueOrFalseException( script );
        }
        if ( !( evaluationResult instanceof Boolean ) ) {
            throw log.getScriptMustReturnTrueOrFalseException(
                    script,
                    evaluationResult,
                    evaluationResult.getClass().getCanonicalName()
            );
        }

        if(Boolean.FALSE.equals(evaluationResult)) {
            constraintValidatorContext.disableDefaultConstraintViolation();
            constraintValidatorContext
                    .buildConstraintViolationWithTemplate(message)
                    .addPropertyNode(property)
                    .addConstraintViolation();
        }

        return Boolean.TRUE.equals( evaluationResult );
    }

    private void validateParameters(PropertyScriptAssert constraintAnnotation) {
        Contracts.assertNotEmpty( constraintAnnotation.script(),
MESSAGES.parameterMustNotBeEmpty( "script" ) );
        Contracts.assertNotEmpty( constraintAnnotation.lang(),
MESSAGES.parameterMustNotBeEmpty( "lang" ) );
        Contracts.assertNotEmpty( constraintAnnotation.alias(),
MESSAGES.parameterMustNotBeEmpty( "alias" ) );
        Contracts.assertNotEmpty( constraintAnnotation.property(),
MESSAGES.parameterMustNotBeEmpty( "property" ) );
        Contracts.assertNotEmpty( constraintAnnotation.message(),
MESSAGES.parameterMustNotBeEmpty( "message" ) );
    }
}
```

和之前的类级别验证器类似，就不多解释了，其他代码全部拷贝自org.hibernate.validator.internal.constraintvalidators.ScriptAssertValidator。

7.3、使用

```
@PropertyScriptAssert(property = "confirmation", script =
"_this.password==_this.confirmation", lang = "javascript", alias = "_this",
message = "{password.confirmation.error}")
```

和之前的区别就是多了个property，用来指定出错时给哪个字段。 这个相对之前的类级别验证器更通用一点。

8、cross-parameter，跨参数验证

直接看示例；

8.1、首先注册MethodValidationPostProcessor，起作用请参考《Spring3.1 对Bean Validation规范的新支持(方法级别验证)》

```
    <bean
class="org.springframework.validation.beanvalidation.MethodValidationPostProcess
or">
        <property name="validator" ref="validator"/>
    </bean>
```

8.2、Service

```
@Validated
@Service
public class UserService {

    @CrossParameter
    public void changePassword(String password, String confirmation) {

    }
}
```

通过@Validated注解UserService表示该类中有需要进行方法参数/返回值验证； @CrossParameter注解方法表示要进行跨参数验证；即验证password和confirmation是否相等。

8.3、验证注解

```
package com.sishuok.spring4.validator;

//省略import
```

```
@Constraint(validatedBy = CrossParameterValidator.class)
@Target({ METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface CrossParameter {

    String message() default "{password.confirmation.error}";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };

}
```

8.4、验证器

```
package com.sishuok.spring4.validator;

//省略import

@SupportedValidationTarget(ValidationTarget.PARAMETERS)
public class CrossParameterValidator implements
ConstraintValidator<CrossParameter, Object[]> {

    @Override
    public void initialize(CrossParameter constraintAnnotation) {
    }

    @Override
    public boolean isValid(Object[] value, ConstraintValidatorContext context) {
        if(value == null || value.length != 2) {
            throw new IllegalArgumentException("must have two args");
        }
        if(value[0] == null || value[1] == null) {
            return true;
        }
        if(value[0].equals(value[1])) {
            return true;
        }
        return false;
    }
}
```

其中@SupportedValidationTarget(ValidationTarget.PARAMETERS)表示验证参数； value将是参数列表。

8.5、使用

```
    @RequestMapping("/changePassword")
    public String changePassword(
            @RequestParam("password") String password,
            @RequestParam("confirmation") String confirmation, Model model) {
        try {
            userService.changePassword(password, confirmation);
        } catch (ConstraintViolationException e) {
            for(ConstraintViolation violation : e.getConstraintViolations()) {
                System.out.println(violation.getMessage());
            }
        }
        return "success";
    }
```

调用userService.changePassword方法，如果验证失败将抛出ConstraintViolationException异常，然后得到ConstraintViolation，调用getMessage即可得到错误消息；然后到前台显示即可。

从以上来看，不如之前的使用方便，需要自己对错误消息进行处理。 下一节我们也写个脚本方式的跨参数验证器。

9、混合类级别验证器和跨参数验证器

9.1、验证注解

```
package com.sishuok.spring4.validator;

//省略import

@Constraint(validatedBy = {
        CrossParameterScriptAssertClassValidator.class,
        CrossParameterScriptAssertParameterValidator.class
})
@Target({ TYPE, FIELD, PARAMETER, METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface CrossParameterScriptAssert {
    String message() default "error";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };
    String script();
    String lang();
    String alias() default "_this";
    String property() default "";
    ConstraintTarget validationAppliesTo() default ConstraintTarget.IMPLICIT;
}
```

此处我们通过@Constraint指定了两个验证器，一个类级别的，一个跨参数的。validationAppliesTo指定为ConstraintTarget.IMPLICIT，表示隐式自动判断。

9.2、验证器

请下载源码查看

9.3、使用

9.3.1、类级别使用

```
@CrossParameterScriptAssert(property = "confirmation", script =
"_this.password==_this.confirmation", lang = "javascript", alias = "_this",
message = "{password.confirmation.error}")
```

指定property即可，其他和之前的一样。

9.3.2、跨参数验证

```
    @CrossParameterScriptAssert(script = "args[0] == args[1]", lang =
"javascript", alias = "args", message = "{password.confirmation.error}")
    public void changePassword(String password, String confirmation) {

    }
```

通过args[0]==args[1] 来判断是否相等。

这样，我们的验证注解就自动适应两种验证规则了。

10、组合验证注解

有时候，可能有好几个注解需要一起使用，此时就可以使用组合验证注解

```
@Target({ FIELD})
@Retention(RUNTIME)
@Documented
@NotNull(message = "{user.name.null}")
@Length(min = 5, max = 20, message = "{user.name.length.illegal}")
@Pattern(regexp = "[a-zA-Z]{5,20}", message = "{user.name.length.illegal}")
@Constraint(validatedBy = { })
public @interface Composition {
    String message() default "";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };
}
```

这样我们验证时只需要：

```
    @Composition()
    private String name;
```

简洁多了。

11、本地化

即根据不同的语言选择不同的错误消息显示。

1、本地化解析器

```
    <bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
        <property name="cookieName" value="locale"/>
        <property name="cookieMaxAge" value="-1"/>
        <property name="defaultLocale" value="zh_CN"/>
    </bean>
```

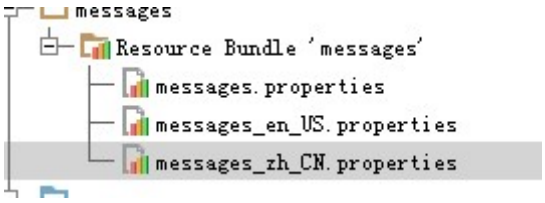
此处使用cookie存储本地化信息，当然也可以选择其他的，如Session存储。

2、设置本地化信息的拦截器

```
    <mvc:interceptors>
        <bean
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
            <property name="paramName" value="language"/>
        </bean>
    </mvc:interceptors>
```

即请求参数中通过language设置语言。

3、消息文件



4、 浏览器输入

http://localhost:9080/spring4/changePassword?password=1&confirmation=2&language=en_US

到此，我们已经完成大部分Bean Validation的功能实验了。对于如XML配置、编程式验证API的使用等对于我们使用SpringMVC这种web环境用处不大，所以就不多介绍了，有兴趣可以自己下载官方文档学习。

Spring4新特性——Groovy Bean定义DSL

Spring4支持使用Groovy DSL来进行Bean定义配置，其类似于XML，不过因为是Groovy DSL，可以实现任何复杂的语法配置，但是对于配置，我们需要那么复杂吗？本着学习的态度试用了下其Groovy DSL定义Bean，其主要缺点：

1、DSL语法规则不足，需要其后续维护；

2、编辑器的代码补全需要跟进，否则没有代码补全，写这个很痛苦；

3、出错提示不友好，排错难；

4、当前对于一些配置还是需要XML的支持，所以还不是100%的纯Groovy DSL；

5、目前对整个Spring生态支持还是不够的，比如Web，需要观望。

其优点就是其本质是Groovy脚本，所以可以做非常复杂的配置，如果以上问题能够解决，其也是一个不错的选择。在Groovy中的话使用这种配置感觉不会有什么问题，但是在纯Java开发环境下也是有它，给我的感觉是这个功能其目的是去推广它的groovy。比较怀疑它的动机。

一、对比

对于我来说，没有哪个好/坏，只有适用不适用；开发方便不方便。接下来我们来看一下各种类型的配置吧：

XML风格配置

```
    <context:component-scan base-package="com.sishuok.spring4"/>
    <bean
class="org.springframework.validation.beanvalidation.MethodValidationPostProcess
or">
        <property name="validator" ref="validator"/>
    </bean>
    <mvc:annotation-driven validator="validator"/>

    <bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
        <property name="providerClass"
value="org.hibernate.validator.HibernateValidator"/>
        <property name="validationMessageSource" ref="messageSource"/>
    </bean>
```

注解风格配置

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.sishuok.spring4")
public class MvcConfiguration extends WebMvcConfigurationSupport {
    @Override
    protected Validator getValidator() {
        LocalValidatorFactoryBean localValidatorFactoryBean =
                new LocalValidatorFactoryBean();
        localValidatorFactoryBean.setProviderClass(HibernateValidator.class);
        localValidatorFactoryBean.setValidationMessageSource(messageSource());
        return localValidatorFactoryBean;
    }
}
```

Groovy DSL风格配置

```
import org.hibernate.validator.HibernateValidator
import org.springframework.context.support.ReloadableResourceBundleMessageSource
import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean

beans {
    xmlns context: "http://www.springframework.org/schema/context"
    xmlns mvc: "http://www.springframework.org/schema/mvc"

    context.'component-scan'('base-package': "com,sishuok.spring4")
    mvc.'annotation-driven'('validator': "validator")

    validator(LocalValidatorFactoryBean) {
        providerClass = HibernateValidator.class
        validationMessageSource = ref("messageSource")
    }
}
```

因为Spring4 webmvc没有提供用于Web环境的Groovy DSL实现的WebApplicationContext，所以为了在web环境使用，单独写了一个WebGenricGroovyApplicationContext，可以到源码中查找。

可以看到，它们之前差别不是特别大；

对于注解风格的配置，如果在Servlet3容器中使用的话，可以借助WebApplicationInitializer实现无配置：

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(javax.servlet.ServletContext sc) throws
ServletException {

//        AnnotationConfigWebApplicationContext rootContext = new
AnnotationConfigWebApplicationContext();
```

```
//        rootContext.register(AppConfig.class);
//        sc.addListener(new ContextLoaderListener(rootContext));

        //2、springmvc上下文
        AnnotationConfigWebApplicationContext springMvcContext = new
AnnotationConfigWebApplicationContext();
        springMvcContext.register(MvcConfiguration.class);

        //3、DispatcherServlet
        DispatcherServlet dispatcherServlet = new
DispatcherServlet(springMvcContext);
        ServletRegistration.Dynamic dynamic = sc.addServlet("dispatcherServlet",
dispatcherServlet);
        dynamic.setLoadOnStartup(1);
        dynamic.addMapping("/");

        //4、CharacterEncodingFilter
        FilterRegistration filterRegistration =
                sc.addFilter("characterEncodingFilter",
CharacterEncodingFilter.class);

filterRegistration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
false, "/*");


    }
}
```

到底好还是不好，需要根据自己项目大小等一些因素来衡量。

对于Groovy风格配置，如果语法足够丰富、Spring内部支持完善，且编辑器支持也非常好的话，也是不错的选择。

二、Groovy Bean定义

接下来我们来看下groovy DSL的具体使用吧：

1、安装环境

```
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-all</artifactId>
            <version>${groovy.version}</version>
        </dependency>
```

我使用的groovy版本是2.2.1

2、相关组件类

此处使用Spring Framework官网的hello world

3、Groovy Bean定义配置文件

```
import com.sishuok.spring4.xml.MessageServiceImpl
import com.sishuok.spring4.xml.MessagePrinter

beans {
    messageService(MessageServiceImpl) {//名字(类型)
        message = "hello"  //注入的属性
    }

    messagePrinter(MessagePrinter, messageService) //名字（类型，构造器参数列表）

}
```

从此处可以看到 如果仅仅是简单的Bean定义，确实比XML简洁。

4、测试

如果不测试环境可以这样测试：

```
public class XmlGroovyBeanDefinitionTest1 {
    @Test
    public void test() {
        ApplicationContext ctx = new
GenericGroovyApplicationContext("classpath:spring-config-xml.groovy");
        MessagePrinter messagePrinter = (MessagePrinter)
ctx.getBean("messagePrinter");
        messagePrinter.printMessage();
    }
}
```

使用GenericGroovyApplicationContext加载groovy配置文件。

如果想集成到Spring Test中，可以这样：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring-config-xml.groovy", loader =
GenericGroovyContextLoader.class)
public class XmlGroovyBeanDefinitionTest2 {

    @Autowired
    private MessagePrinter messagePrinter;

    @Test
    public void test() {
        messagePrinter.printMessage();
    }
}
```

此处需要定义我们自己的bean loader，即从groovy配置文件加载：

```
public class GenericGroovyContextLoader extends AbstractGenericContextLoader {

    @Override
    protected String getResourceSuffix() {
        throw new UnsupportedOperationException(
                "GenericGroovyContextLoader does not support the
getResourceSuffix() method");
    }
    @Override
    protected BeanDefinitionReader
createBeanDefinitionReader(GenericApplicationContext context) {
        return new GroovyBeanDefinitionReader(context);
    }
}
```

使用GroovyBeanDefinitionReader来加载groovy配置文件。

到此基本的使用就结束了，还算是比较简洁，但是我们已经注意到了，在纯Java环境做测试还是比较麻烦的。 比如没有给我们写好相关的测试支撑类。

再看一下我们使用注解方式呢：

```
@Component
public class MessageServiceImpl implements MessageService {
    @Autowired
    @Qualifier("message")
    private String message;
    ……
}
```

```
@Component
public class MessagePrinter {
    private MessageService messageService;
    @Autowired
    public MessagePrinter(MessageService messageService) {
        this.messageService = messageService;
    }
……
}
```

Groovy配置文件：

```
beans {
    xmlns context: "http://www.springframework.org/schema/context"    //导入命名空间

    context.'component-scan'('base-package': "com.sishuok.spring4") {
        'exclude-filter'('type': "aspectj", 'expression': "com.sishuok.spring4.xml.*")
    }

    message(String, "hello") {}

}
```

在该配置文件中支持导入xml命名空间， 其中context.'component-scan'部分等价于XML中的：

```
    <context:component-scan base-package="com.sishuok.spring4">
        <context:exclude-filter type="aspectj" expression="com.sishuok.spring4.xml.*"/>
    </context:component-scan>
```

从这里可以看出，其还没能完全从XML风格配置中走出来，不是纯Groovy DSL。

测试方式和之前的一样

三、Groovy Bean定义 DSL语法

到目前为止，基本的helloworld就搞定了；接下来看看Groovy DSL都支持哪些配置吧：

创建Bean

构造器

```
    validator(LocalValidatorFactoryBean) { //名字（类型）
        providerClass = HibernateValidator.class  //属性=值
        validationMessageSource = ref("messageSource") //属性 = 引用，当然也支持如validationMessageSource=messageSource 但是这种方式缺点是messageSource必须在validator之前声明
    }
```

静态工厂方法

```
def bean = factory(StaticFactory) {
    prop = 1
}
bean.factoryMethod = "getInstance"
```

或者

```
bean(StaticFactory) { bean ->
    bean.factoryMethod = "getInstance"
    prop = 1
}
```

实例工厂方法

```
beanFactory(Factory)
bean(beanFactory : "newInstance", "args") {
    prop = 1
}
```

或者

```
beanFactory(Factory)
bean("bean"){bean ->
    bean.factoryBean="beanFactory"
    bean.factoryMethod="newInstance"
    prop = 1
}
```

依赖注入

属性注入

```
    beanName(BeanClass) { //名字（类型）
        str = "123" // 常量直接注入
        bean = ref("bean") //属性 = 引用 ref("bean", true) 这样的话是引用父容器的
        beans = [bean1, bean2] //数组/集合
        props = [key1:"value1", key2:"value2"] // Properties / Map
    }
```

构造器注入

```
bean(Bean, "args1", "args2")
```

静态工厂注入/实例工厂注入，请参考创建bean部分

匿名内部Bean

```
outer(OuterBean) {
   prop = 1
   inner =  { InnerBean bean ->  //匿名内部Bean
                            prop =2
            }
}
```

```
outer(OuterBean) {
   prop = 1
   inner =  { bean ->  //匿名内部Bean 通过实例工厂方法创建
                            bean.factoryBean = "innerBean"
                            bean.factoryMethod = "create"
                            prop = 2
            }
}
```

单例/非单例/作用域

```
singletonBean(Bean1) { bean ->
    bean.singleton = true
}
nonSingletonBean(Bean1) { bean ->
    bean.singleton = false
}
prototypeBean(Bean1) { bean ->
    bean.scope = "prototype"
}
```

其中bean可以理解为xml中的 标签，即bean定义。

父子Bean

```
parent(Bean1){ bean ->
    bean.'abstract' = true //抽象的
    prop = 123
}
child { bean ->
    bean.parent = parent //指定父bean
}
```

命名空间

```
    xmlns aop:"http://www.springframework.org/schema/aop"
    myAspect(MyAspect)
    aop {
        config("proxy-target-class":true) {
            aspect( id:"test",ref:"myAspect" ) {
                before method:"before", pointcut: "execution(void
com.sishuok.spring4..*.*(..))"
            }
        }
    }
```

以上是AOP的，可以自己推到其他相关的配置；

```
    xmlns context: "http://www.springframework.org/schema/context"
    context.'component-scan'('base-package': "com.sishuok.spring4") {
        'exclude-filter'('type': "aspectj", 'expression':
"com.sishuok.spring4.xml.*")
    }
```

以上是component-scan，之前介绍过了。

```
xmlns aop:"http://www.springframework.org/schema/aop"
scopedList(ArrayList) { bean ->
    bean.scope = "haha"
    aop.'scoped-proxy'()
}
```

等价于

```
    <bean id="scopedList" class="java.util.ArrayList" scope="haha">
        <aop:scoped-proxy/>
    </bean>
```

```
    xmlns util:"http://www.springframework.org/schema/util"
    util.list(id : 'list') {
        value 1
        value 2
    }
```

等价于XML：

```
    <util:list id="list">
        <value>1</value>
        <value>2</value>
    </util:list>
```

```
    xmlns util:"http://www.springframework.org/schema/util"
    util.map(id : 'map') {
        entry(key : 1, value :1)
        entry('key-ref' : "messageService", 'value-ref' : "messageService")
    }
```

等价于

```
    <util:map id="map">
        <entry key="1" value="1"/>
        <entry key-ref="messageService" value-ref="messageService"/>
    </util:map>
```

引入其他配置文件

```
importBeans "classpath:org/springframework/context/groovy/test.xml"
```

当然也能引入XML的。

再来看看groovy bean定义的另一个好处：

我们可以直接在groovy bean定义文件中声明类，然后使用

```
@Controller
def class GroovyController {
    @RequestMapping("/groovy")
    @ResponseBody
    public String hello() {
        return "hello";
    }
}

beans {

    groovyController(GroovyController)

}
```

另一种Spring很早就支持的方式是引入外部groovy文件，如：

```
xmlns lang: "http://www.springframework.org/schema/lang"
lang.'groovy'(id: 'groovyController2', 'script-source':
'classpath:com/sishuok/spring4/controller/GroovyController2.groovy')
```

使用其lang命名空间引入外部脚本文件。

到此，Groovy Bean定义DSL就介绍完了，其没有什么特别之处，只是换了种写法而已，我认为目前试试即可，还不能用到真实环境。

Spring4新特性——更好的Java泛型操作API

随着泛型用的越来越多，获取泛型实际类型信息的需求也会出现，如果用原生API，需要很多步操作才能获取到泛型，比如：

```
ParameterizedType parameterizedType =
    (ParameterizedType) ABService.class.getGenericInterfaces()[0];
Type genericType = parameterizedType.getActualTypeArguments()[1];
```

Spring提供的ResolvableType API，提供了更加简单易用的泛型操作支持，如：

```
ResolvableType resolvableType1 = ResolvableType.forClass(ABService.class);
resolvableType1.as(Service.class).getGeneric(1).resolve()
```

对于获取更复杂的泛型操作ResolvableType更加简单。

假设我们的API是：

```
public interface Service<N, M> {
}

@org.springframework.stereotype.Service
public class ABService implements Service<A, B> {
}

@org.springframework.stereotype.Service
public class CDService implements Service<C, D> {
}
```

如上泛型类非常简单。

1、得到类型的泛型信息

```
ResolvableType resolvableType1 = ResolvableType.forClass(ABService.class);
```

通过如上API，可以得到类型的ResolvableType，如果类型被Spring AOP进行了CGLIB代理，请使用ClassUtils.getUserClass(ABService.class)得到原始类型。

可以通过如下得到泛型参数的第1个位置（从0开始）的类型信息

```
resolvableType1.getInterfaces()[0].getGeneric(1).resolve()
```

因为我们泛型信息放在 Service<A, B> 上，所以需要resolvableType1.getInterfaces()[0]得到；

通过getGeneric（泛型参数索引）得到某个位置的泛型；

resolve()把实际泛型参数解析出来

2、得到字段级别的泛型信息

假设我们的字段如下：

```
   @Autowired
    private Service<A, B> abService;
    @Autowired
    private Service<C, D> cdService;

    private List<List<String>> list;

    private Map<String, Map<String, Integer>> map;

    private List<String>[] array;
```

通过如下API可以得到字段级别的ResolvableType

```
ResolvableType resolvableType2 =

ResolvableType.forField(ReflectionUtils.findField(GenricInjectTest.class,
"cdService"));
```

然后通过如下API得到Service<C, D>的第0个位置上的泛型实参类型，即C

```
resolvableType2.getGeneric(0).resolve()
```

比如 List<List> list;是一种嵌套的泛型用例，我们可以通过如下操作获取String类型：

```
ResolvableType resolvableType3 =

ResolvableType.forField(ReflectionUtils.findField(GenricInjectTest.class,
"list"));
resolvableType3.getGeneric(0).getGeneric(0).resolve();
```

更简单的写法

```
resolvableType3.getGeneric(0, 0).resolve(); //List<List<String>> 即String
```

比如Map<String, Map<String, Integer>> map;我们想得到Integer，可以使用：

```
ResolvableType resolvableType4 =

ResolvableType.forField(ReflectionUtils.findField(GenricInjectTest.class,
"map"));
resolvableType4.getGeneric(1).getGeneric(1).resolve();
```

更简单的写法

```
resolvableType4.getGeneric(1, 1).resolve()
```

3、得到方法返回值的泛型信息

假设我们的方法如下：

```
private HashMap<String, List<String>> method() {
    return null;
}
```

得到Map中的List中的String泛型实参：

```
ResolvableType resolvableType5 =

ResolvableType.forMethodReturnType(ReflectionUtils.findMethod(GenricInjectTest.c
lass, "method"));
resolvableType5.getGeneric(1, 0).resolve();
```

4、得到构造器参数的泛型信息

假设我们的构造器如下：

```
public Const(List<List<String>> list, Map<String, Map<String, Integer>> map) {
}
```

我们可以通过如下方式得到第1个参数（Map<String, Map<String, Integer>>）中的Integer：

```
ResolvableType resolvableType6 =

ResolvableType.forConstructorParameter(ClassUtils.getConstructorIfAvailable(Cons
t.class, List.class, Map.class), 1);
resolvableType6.getGeneric(1, 0).resolve();
```

5、得到数组组件类型的泛型信息

如对于private List[] array; 可以通过如下方式获取List的泛型实参String：

```
ResolvableType resolvableType7 =

ResolvableType.forField(ReflectionUtils.findField(GenricInjectTest.class,
"array"));
resolvableType7.isArray();//判断是否是数组
resolvableType7.getComponentType().getGeneric(0).resolve();
```

6、自定义泛型类型

```
ResolvableType resolvableType8 = ResolvableType.forClassWithGenerics(List.class,
String.class);
        ResolvableType resolvableType9 =
ResolvableType.forArrayComponent(resolvableType8);
resolvableType9.getComponentType().getGeneric(0).resolve();
```

ResolvableType.forClassWithGenerics(List.class, String.class)相当于创建一个List类型；

ResolvableType.forArrayComponent(resolvableType8);：相当于创建一个List[]数组；

resolvableType9.getComponentType().getGeneric(0).resolve()：得到相应的泛型信息；

7、泛型等价比较：

```
resolvableType7.isAssignableFrom(resolvableType9)
```

如下创建一个List[]数组，与之前的List[]数组比较，将返回false。

```
ResolvableType resolvableType10 = ResolvableType.forClassWithGenerics(List.class,
Integer.class);
ResolvableType resolvableType11=
ResolvableType.forArrayComponent(resolvableType10);
resolvableType11.getComponentType().getGeneric(0).resolve();
resolvableType7.isAssignableFrom(resolvableType11);
```

从如上操作可以看出其泛型操作功能十分完善，尤其在嵌套的泛型信息获取上相当简洁。目前整个Spring4环境都使用这个API来操作泛型信息。

如之前说的泛型注入：Spring4新特性——泛型限定式依赖注入，通过在依赖注入时使用如下类实现：

GenericTypeAwareAutowireCandidateResolver

QualifierAnnotationAutowireCandidateResolver

ContextAnnotationAutowireCandidateResolver

还有如Spring的核心BeanWrapperImpl，以及整个Spring/SpringWevMVC的泛型操作都是替换为这个API了：GenericCollectionTypeResolver和GenericTypeResolver都直接委托给ResolvableType这个API。

所以大家以后对泛型操作可以全部使用这个API了，非常好用。

Spring4新特性——JSR310日期时间API的支持

JSR310 日期与时间规范主要API如下：

Clock

时钟，类似于钟表的概念，提供了如系统时钟、固定时钟、特定时区的时钟

```
        //时钟提供给我们用于访问某个特定 时区的 瞬时时间、日期 和 时间的。
        Clock c1 = Clock.systemUTC(); //系统默认UTC时钟（当前瞬时时间
System.currentTimeMillis()）
        System.out.println(c1.millis()); //每次调用将返回当前瞬时时间（UTC）

        Clock c2 = Clock.systemDefaultZone(); //系统默认时区时钟（当前瞬时时间）

        Clock c31 = Clock.system(ZoneId.of("Europe/Paris")); //巴黎时区
        System.out.println(c31.millis()); //每次调用将返回当前瞬时时间（UTC）

        Clock c32 = Clock.system(ZoneId.of("Asia/Shanghai"));//上海时区
        System.out.println(c32.millis());//每次调用将返回当前瞬时时间（UTC）

        Clock c4 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Shanghai"));//固定上
海时区时钟
```

```
        System.out.println(c4.millis());
        Thread.sleep(1000);
        System.out.println(c4.millis()); //不变 即时钟时钟在那一个点不动

        Clock c5 = Clock.offset(c1, Duration.ofSeconds(2)); //相对于系统默认时钟两秒
的时钟
        System.out.println(c1.millis());
        System.out.println(c5.millis());
```

Instant

瞬时时间，等价于以前的System.currentTimeMillis()

```
        //瞬时时间 相当于以前的System.currentTimeMillis()
        Instant instant1 = Instant.now();
        System.out.println(instant1.getEpochSecond());//精确到秒 得到相对于1970-01-
01 00:00:00 UTC的一个时间
        System.out.println(instant1.toEpochMilli()); //精确到毫秒

        Clock clock1 = Clock.systemUTC(); //获取系统UTC默认时钟
        Instant instant2 = Instant.now(clock1);//得到时钟的瞬时时间
        System.out.println(instant2.toEpochMilli());

        Clock clock2 = Clock.fixed(instant1, ZoneId.systemDefault()); //固定瞬时时
间时钟
        Instant instant3 = Instant.now(clock2);//得到时钟的瞬时时间
        System.out.println(instant3.toEpochMilli());//equals instant1
```

LocalDateTime、LocalDate、LocalTime

提供了对java.util.Date的替代，另外还提供了新的DateTimeFormatter用于对格式化/解析的支持

```
        //使用默认时区时钟瞬时时间创建 Clock.systemDefaultZone() -->即相对于
ZoneId.systemDefault()默认时区
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);

        //自定义时区
        LocalDateTime now2= LocalDateTime.now(ZoneId.of("Europe/Paris"));
        System.out.println(now2);//会以相应的时区显示日期

        //自定义时钟
        Clock clock = Clock.system(ZoneId.of("Asia/Dhaka"));
        LocalDateTime now3= LocalDateTime.now(clock);
        System.out.println(now3);//会以相应的时区显示日期

        //不需要写什么相对时间 如java.util.Date 年是相对于1900 月是从0开始
        //2013-12-31 23:59
        LocalDateTime d1 = LocalDateTime.of(2013, 12, 31, 23, 59);

        //年月日 时分秒 纳秒
        LocalDateTime d2 = LocalDateTime.of(2013, 12, 31, 23, 59,59, 11);
```

```
        //使用瞬时时间 + 时区
        Instant instant = Instant.now();
        LocalDateTime d3 = LocalDateTime.ofInstant(Instant.now(),
ZoneId.systemDefault());
        System.out.println(d3);

        //解析String--->LocalDateTime
        LocalDateTime d4 = LocalDateTime.parse("2013-12-31T23:59");
        System.out.println(d4);

        LocalDateTime d5 = LocalDateTime.parse("2013-12-31T23:59:59.999");//999毫
秒 等价于999000000纳秒
        System.out.println(d5);

        //使用DateTimeFormatter API 解析 和 格式化
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd
HH:mm:ss");
        LocalDateTime d6 = LocalDateTime.parse("2013/12/31 23:59:59",
formatter);
        System.out.println(formatter.format(d6));


        //时间获取
        System.out.println(d6.getYear());
        System.out.println(d6.getMonth());
        System.out.println(d6.getDayOfYear());
        System.out.println(d6.getDayOfMonth());
        System.out.println(d6.getDayOfWeek());
        System.out.println(d6.getHour());
        System.out.println(d6.getMinute());
        System.out.println(d6.getSecond());
        System.out.println(d6.getNano());
        //时间增减
        LocalDateTime d7 = d6.minusDays(1);
        LocalDateTime d8 = d7.plus(1, IsoFields.QUARTER_YEARS);

        //LocalDate 即年月日 无时分秒
        //LocalTime即时分秒 无年月日
        //API和LocalDateTime类似就不演示了
```

ZonedDateTime

带有时区的date-time 存储纳秒、时区和时差（避免与本地date-time歧义）； API和LocalDateTime类似，只是多了时差(如2013-12-20T10:35:50.711+08:00[Asia/Shanghai])

```
        //即带有时区的date-time 存储纳秒、时区和时差（避免与本地date-time歧义）。
        //API和LocalDateTime类似，只是多了时差(如2013-12-
20T10:35:50.711+08:00[Asia/Shanghai])
        ZonedDateTime now = ZonedDateTime.now();
        System.out.println(now);

        ZonedDateTime now2= ZonedDateTime.now(ZoneId.of("Europe/Paris"));
        System.out.println(now2);

        //其他的用法也是类似的 就不介绍了

        ZonedDateTime z1 = ZonedDateTime.parse("2013-12-
31T23:59:59Z[Europe/Paris]");
        System.out.println(z1);
```

Duration

表示两个瞬时时间的时间段

```
        //表示两个瞬时时间的时间段
        Duration d1 =
Duration.between(Instant.ofEpochMilli(System.currentTimeMillis() - 12323123),
Instant.now());
        //得到相应的时差
        System.out.println(d1.toDays());
        System.out.println(d1.toHours());
        System.out.println(d1.toMinutes());
        System.out.println(d1.toMillis());
        System.out.println(d1.toNanos());

        //1天时差 类似的还有如ofHours()
        Duration d2 = Duration.ofDays(1);
        System.out.println(d2.toDays());
```

Chronology

用于对年历系统的支持，是java.util.Calendar的替代者

```
        //提供对java.util.Calendar的替换，提供对年历系统的支持
        Chronology c  = HijrahChronology.INSTANCE;
        ChronoLocalDateTime d = c.localDateTime(LocalDateTime.now());
        System.out.println(d);
```

其他

如果提供了年、年月、月日、周期的API支持

```
        Year year = Year.now();
        YearMonth yearMonth = YearMonth.now();
        MonthDay monthDay = MonthDay.now();

        System.out.println(year);//年
        System.out.println(yearMonth); //年-月
        System.out.println(monthDay); // 月-日

        //周期，如表示10天前  3年5个月钱
        Period period1 = Period.ofDays(10);
        System.out.println(period1);
        Period period2 = Period.of(3, 5, 0);
        System.out.println(period2);
```

我们只需要在实体/Bean上使用DateTimeFormat注解：

```
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime dateTime;

    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate date;

    @DateTimeFormat(pattern = "HH:mm:ss")
    private LocalTime time;
```

比如我们在springmvc中：

```
    @RequestMapping("/test")
    public String test(@ModelAttribute("entity") Entity entity) {
        return "test";
    }
```

当前端页面请求：

localhost:9080/spring4/test?dateTime=2013-11-11 11:11:11&date=2013-11-11&time=12:12:12

会自动进行类型转换。

另外spring4也提供了对TimeZone的支持；比如在springmvc中注册了LocaleContextResolver相应实现的话（如CookieLocaleResolver），我们就可以使用如下两种方式得到相应的TimeZone：

RequestContextUtils.getTimeZone(request)

LocaleContextHolder.getTimeZone()

不过目前的缺点是不能像Local那样自动的根据当前请求得到相应的TimeZone，如果需要这种功能需要覆盖相应的如CookieLocaleResolver中的如下方法来得到：

```
    protected TimeZone determineDefaultTimeZone(HttpServletRequest request) {
        return getDefaultTimeZone();
    }
```

另外还提供了DateTimeContextHolder，其用于线程绑定DateTimeContext；而DateTimeContext提供了如：Chronology、ZoneId、DateTimeFormatter等上下文数据，如果需要这种上下文信息的话，可以使用这个API进行绑定。比如在进行日期格式化时，就会去查找相应的DateTimeFormatter，因此如果想自定义相应的格式化格式，那么使用DateTimeContextHolder绑定即可。

spring4只是简单的对jsr310提供了相应的支持，没有太多的增强。

Spring4新特性——注解、脚本、任务、MVC等其他特性改进

一、注解方面的改进

spring4对注解API和ApplicationContext获取注解Bean做了一点改进。

获取注解的注解，如@Service是被@Compent注解的注解，可以通过如下方式获取@Componet注解实例：

```
Annotation service = AnnotationUtils.findAnnotation(ABService.class,
org.springframework.stereotype.Service.class);
Annotation component = AnnotationUtils.getAnnotation(service,
org.springframework.stereotype.Component.class);
```

获取重复注解：

比如在使用hibernate validation时，我们想在一个方法上加相同的注解多个，需要使用如下方式：

```
@Length.List(
        value = {
                @Length(min = 1, max = 2, groups = A.class),
                @Length(min = 3, max = 4, groups = B.class)
        }
)
public void test() {
```

可以通过如下方式获取@Length：

```
Method method = ClassUtils.getMethod(AnnotationUtilsTest.class, "test");
Set<Length> set = AnnotationUtils.getRepeatableAnnotation(method,
Length.List.class, Length.class);
```

当然，如果你使用Java8，那么本身就支持重复注解，比如spring的任务调度注解，

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repeatable(Schedules.class)
public @interface Scheduled {
```

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Schedules {

    Scheduled[] value();

}
```

这样的话，我们可以直接同时注解相同的多个注解：

```
@Scheduled(cron = "123")
@Scheduled(cron = "234")
public void test
```

但是获取的时候还是需要使用如下方式：

```
AnnotationUtils.getRepeatableAnnotation(ClassUtils.getMethod(TimeTest.class,
"test"), Schedules.class, Scheduled.class)
```

ApplicationContext和BeanFactory提供了直接通过注解获取Bean的方法：

```
    @Test
    public void test() {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
        ctx.register(GenericConfig.class);
        ctx.refresh();

        Map<String, Object> beans =
ctx.getBeansWithAnnotation(org.springframework.stereotype.Service.class);
        System.out.println(beans);
    }
```

这样可以实现一些特殊功能。

另外和提供了一个AnnotatedElementUtils用于简化java.lang.reflect.AnnotatedElement的操作，具体请参考其javadoc。

二、脚本的支持

spring4也提供了类似于javax.script的简单封装，用于支持一些脚本语言，核心接口是：

```
public interface ScriptEvaluator {
    Object evaluate(ScriptSource script) throws ScriptCompilationException;
    Object evaluate(ScriptSource script, Map<String, Object> arguments) throws
ScriptCompilationException;
}
```

比如我们使用groovy脚本的话，可以这样：

```
    @Test
    public void test() throws ExecutionException, InterruptedException {
        ScriptEvaluator scriptEvaluator = new GroovyScriptEvaluator();

        //ResourceScriptSource 外部的
        ScriptSource source = new StaticScriptSource("i+j");
        Map<String, Object> args = new HashMap<>();
        args.put("i", 1);
        args.put("j", 2);
        System.out.println(scriptEvaluator.evaluate(source, args));
    }
```

没什么很特别的地方。另外还提供了BeanShell（BshScriptEvaluator）和javax.script（StandardScriptEvaluator）的简单封装。

三、Future增强

提供了一个ListenableFuture，其是jdk的Future的封装，用来支持回调（成功/失败），其借鉴了com.google.common.util.concurrent.ListenableFuture。

```
    @Test
    public void test() throws Exception {
        ListenableFutureTask<String> task = new ListenableFutureTask<String>(new
Callable() {
            @Override
            public Object call() throws Exception {
                Thread.sleep(10 * 1000L);
                System.out.println("=======task execute");
                return "hello";
            }
        });

        task.addCallback(new ListenableFutureCallback<String>() {
            @Override
            public void onSuccess(String result) {
                System.out.println("===success callback 1");
            }

            @Override
            public void onFailure(Throwable t) {
            }
```

```
        });

        task.addCallback(new ListenableFutureCallback<String>() {
            @Override
            public void onSuccess(String result) {
                System.out.println("===success callback 2");
            }

            @Override
            public void onFailure(Throwable t) {
            }
        });

        ExecutorService executorService = Executors.newSingleThreadExecutor();
        executorService.submit(task);
        String result = task.get();
        System.out.println(result);

    }
```

可以通过addCallback添加一些回调，当执行成功/失败时会自动调用。

四、MvcUriComponentsBuilder

MvcUriComponentsBuilder类似于ServletUriComponentsBuilder，但是可以直接从控制器获取URI信息，如下所示：

假设我们的控制器是：

```
@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping("/{id}")
    public String view(@PathVariable("id") Long id) {
        return "view";
    }

    @RequestMapping("/{id}")
    public A getUser(@PathVariable("id") Long id) {
        return new A();
    }

}
```

注：如果在真实mvc环境，存在两个@RequestMapping("/{id}")是错误的。当前只是为了测试。

我们可以通过如下方式得到

```
    //需要静态导入 import static
org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder.*;
    @Test
    public void test() {
        MockHttpServletRequest req = new MockHttpServletRequest();
```

```
        RequestContextHolder.setRequestAttributes(new
ServletRequestAttributes(req));

        //MvcUriComponentsBuilder类似于ServletUriComponentsBuilder，但是直接从控制器
获取
        //类级别的
        System.out.println(
                fromController(UserController.class).build().toString()
        );

        //方法级别的
        System.out.println(
                fromMethodName(UserController.class, "view",
1L).build().toString()
        );

        //通过Mock方法调用得到
        System.out.println(
                fromMethodCall(on(UserController.class).getUser(2L)).build()
        );
    }
}
```

注意：当前MvcUriComponentsBuilder实现有问题，只有JDK环境支持，大家可以复制一份，然后修改：

method.getParameterCount() （Java 8才支持）

到

method.getParameterTypes().length

五、Socket支持

提供了获取Socket TCP/UDP可用端口的工具，如

SocketUtils.findAvailableTcpPort()

SocketUtils.findAvailableTcpPort(min, max)

SocketUtils.findAvailableUdpPort()

非常简单，就不用特别说明了。

如何自定义一个Spring Boot Starter?

从本章开始，一起实战一个自定义的spring boot starter，整个系列共三篇文章，内容如下：

1. 准备：了解基本概念、规划实战内容；
2. 实战：开发自定义starter，并在demo工程中使用它；
3. 深入：从spring和spring boot源码层面分析starter的原理；

本章内容概览

1. 查看官方资料；
2. 设定实战目标；
3. 学习spring cloud的starter，作为实战参考；
4. 实战内容的具体设计；

版本信息

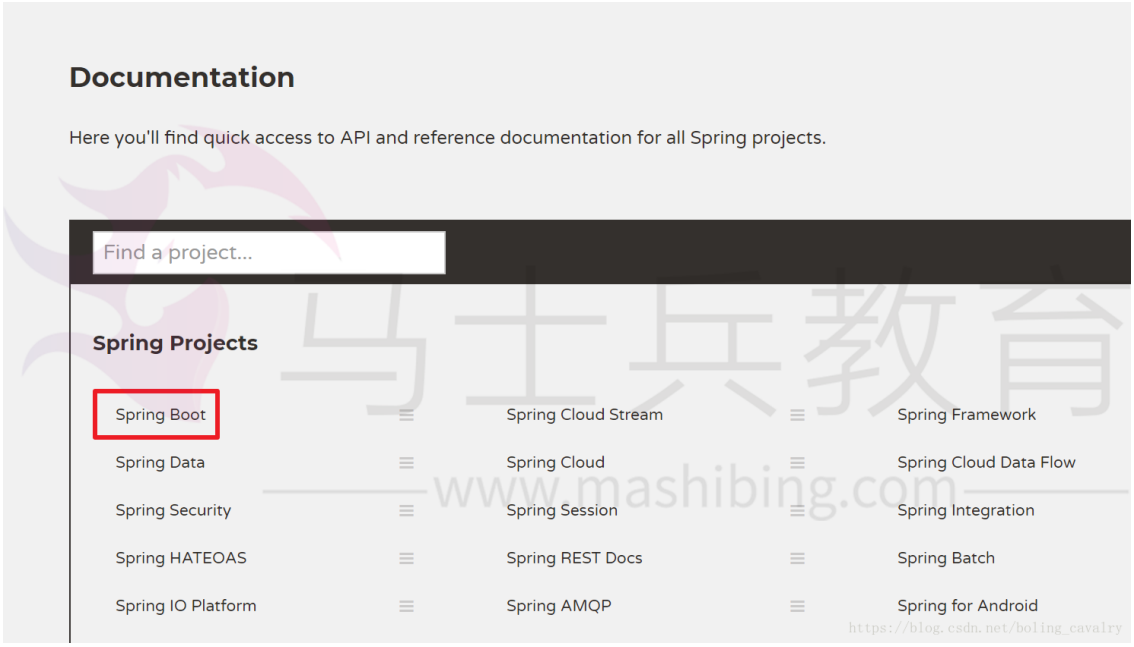
本次实战的版本信息：

1. java：1.8.0_144
2. spring boot：1.5.16.RELEASE
3. spring cloud：Edgware.RELEASE

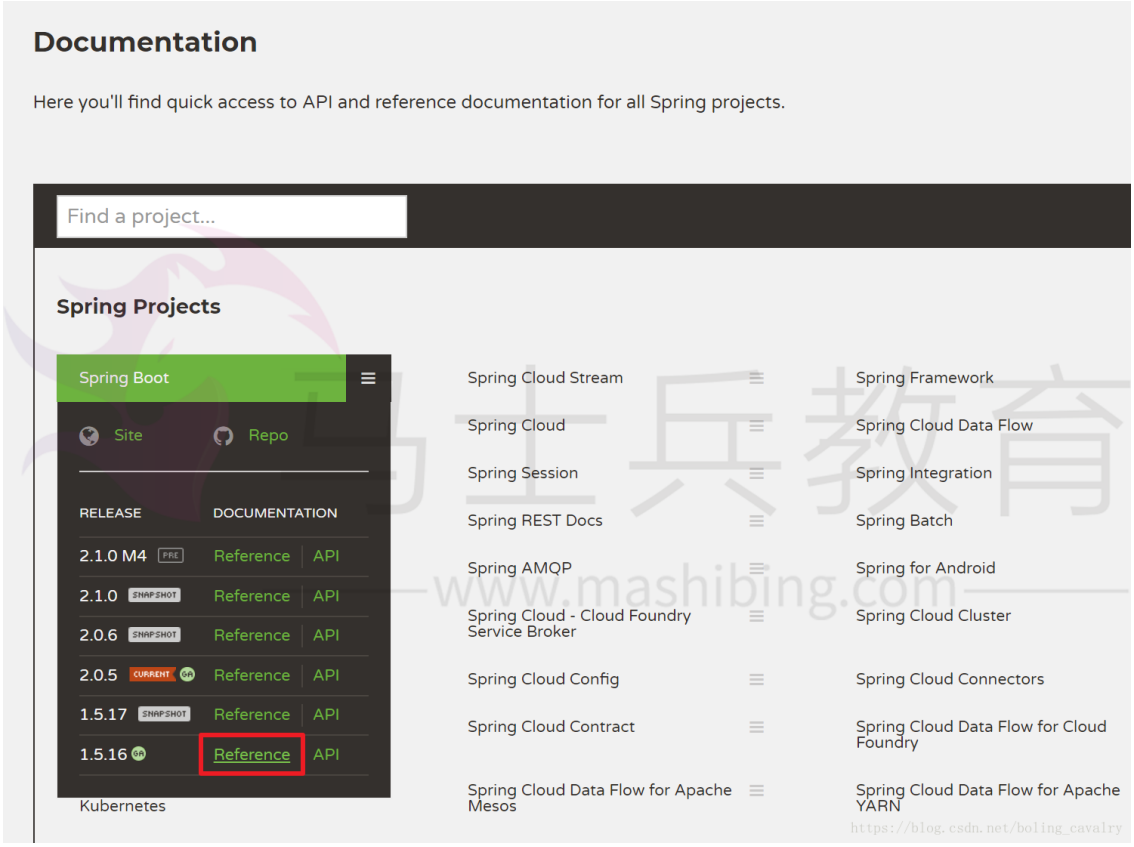
官方资料

为了有个初始印象，我们从spring官方文档看起吧：

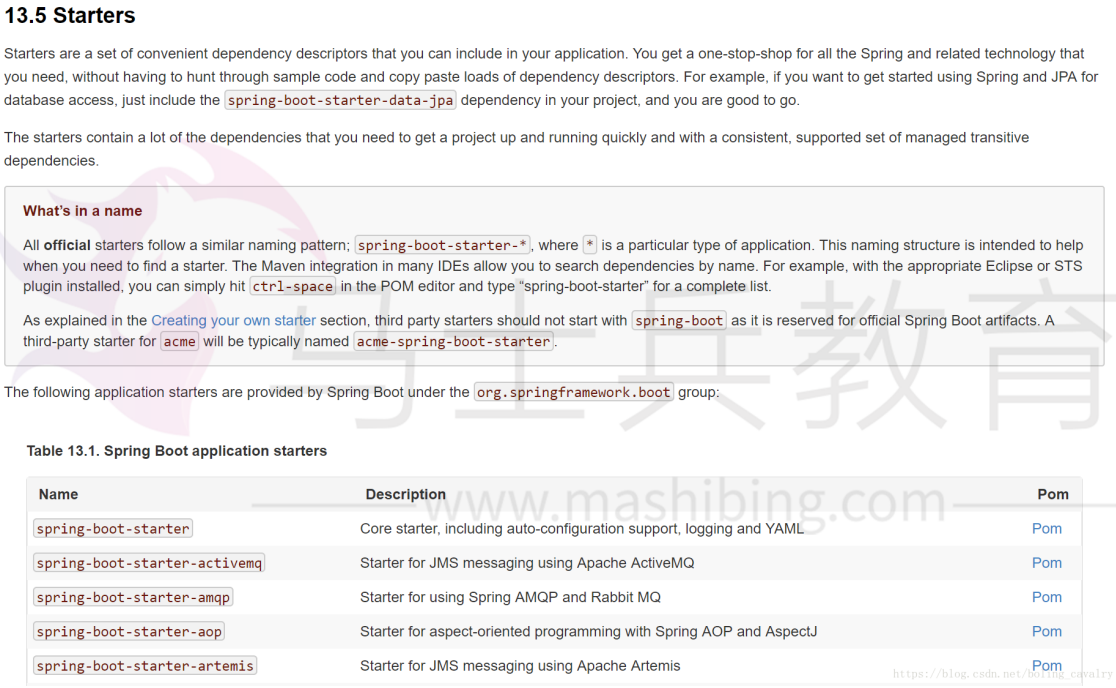
1. 官网：https://spring.io/docs/reference，点击下图红框位置：



2. 在弹出的列表中选择1.5.16版本的reference，如下图红框：



3. 在打开的文档目录中很容易找到starter的章节，地址是：https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/htmlsingle/#using-boot-starter，内容如下图：



13.5 Starters

Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, just include the `spring-boot-starter-data-jpa` dependency in your project, and you are good to go.

The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

What's in a name

All **official** starters follow a similar naming pattern; `spring-boot-starter-*`, where `*` is a particular type of application. This naming structure is intended to help when you need to find a starter. The Maven integration in many IDEs allow you to search dependencies by name. For example, with the appropriate Eclipse or STS plugin installed, you can simply hit `ctrl-space` in the POM editor and type "spring-boot-starter" for a complete list.

As explained in the Creating your own starter section, third party starters should not start with `spring-boot` as it is reserved for official Spring Boot artifacts. A third-party starter for `acme` will be typically named `acme-spring-boot-starter`.

The following application starters are provided by Spring Boot under the `org.springframework.boot` group:

Table 13.1. Spring Boot application starters

Name	Description	Pom
`spring-boot-starter`	Core starter, including auto-configuration support, logging and YAML	Pom
`spring-boot-starter-activemq`	Starter for JMS messaging using Apache ActiveMQ	Pom
`spring-boot-starter-amqp`	Starter for using Spring AMQP and Rabbit MQ	Pom
`spring-boot-starter-aop`	Starter for aspect-oriented programming with Spring AOP and AspectJ	Pom
`spring-boot-starter-artemis`	Starter for JMS messaging using Apache Artemis	Pom

我的理解：

第一. 在应用中可以用starter将依赖库问题变得简单，如果你想依赖Spring和JPA，只需在应用中依赖spring-boot-starter-data-jpa即可；

第二. 常用库的官方starter，其artifactId的格式类似"spring-boot-starter-*"， 对于非官方的starter，建议将业务名称放在"spring-boot-starter"前面，例如"acme-spring-boot-starter"；

第三. 已列举常用的官方starter，可用来参考；

设定实战目标

本次实战的目标如下：

1. A应用提供加法计算的服务；
2. B应用提供减法计算的服务；
3. C应用要使用加法计算和减法计算的服务，并且减法服务可以通过配置来实现是否支持负数；

学习spring cloud的starter

目标已定下，但是先不急着编码，我们去看下spring cloud的设计，用来作为借鉴参考；

回顾一下我们使用Spring cloud的时候，如果要把一个应用作为Eureka client注册到Eureka server，只需在应用的pom.xml中添加如下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

注册到Eureka server的工作，是由CloudEurekaClient类完成的，该类属于模块spring-cloud-netflix-eureka-client，因此我们要弄清楚以下两点：

1. 为什么不需要应用的pom.xml中依赖spring-cloud-netflix-eureka-client?
2. 为什么应用能自动注册到Eureka sever?

如何建立对spring-cloud-netflix-eureka-client模块的依赖

打开spring-cloud-starter-netflix-eureka-client模块的pom.xml文件就一目了然了，如下图，原来在这个pom.xml文件中已经依赖了spring-cloud-netflix-eureka-client模块，因此，我们的应用只需依赖spring-cloud-starter-netflix-eureka-client模块，就能间接依赖到spring-cloud-netflix-eureka-client模块：

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/mave
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix</artifactId>
        <version>1.4.0.RELEASE</version>
    </parent>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <name>Spring Cloud Starter Netflix Eureka Client</name>
    <description>Spring Cloud Starter Netflix Eureka Client</description>
    <url>https://projects.spring.io/spring-cloud</url>
    <organization>
        <name>Pivotal Software, Inc.</name>
        <url>https://www.spring.io</url>
    </organization>
    <properties>
        <main.basedir>${basedir}/../../..</main.basedir>
    </properties>
    <dependencies>
        <dependency>
            <!-- You actually can't use Eureka as a client unless you register so
                it has to be a webapp -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-netflix-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-netflix-eureka-client</artifactId>
        </dependency>
        <dependency>
            <groupId>com.netflix.eureka</groupId>
            <artifactId>eureka-client</artifactId>
        </dependency>
        <dependency>
            <groupId>com.netflix.eureka</groupId>
            <artifactId>eureka-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-archaius</artifactId>
        </dependency>
```

https://blog.csdn.net/boling_cavalry

再看看上图中其他的依赖，可以发现的确如官方文档所说，starter处理了复杂的依赖关系，我们只需要依赖starter即可，官方文档中还有一段话需要注意，如下图：

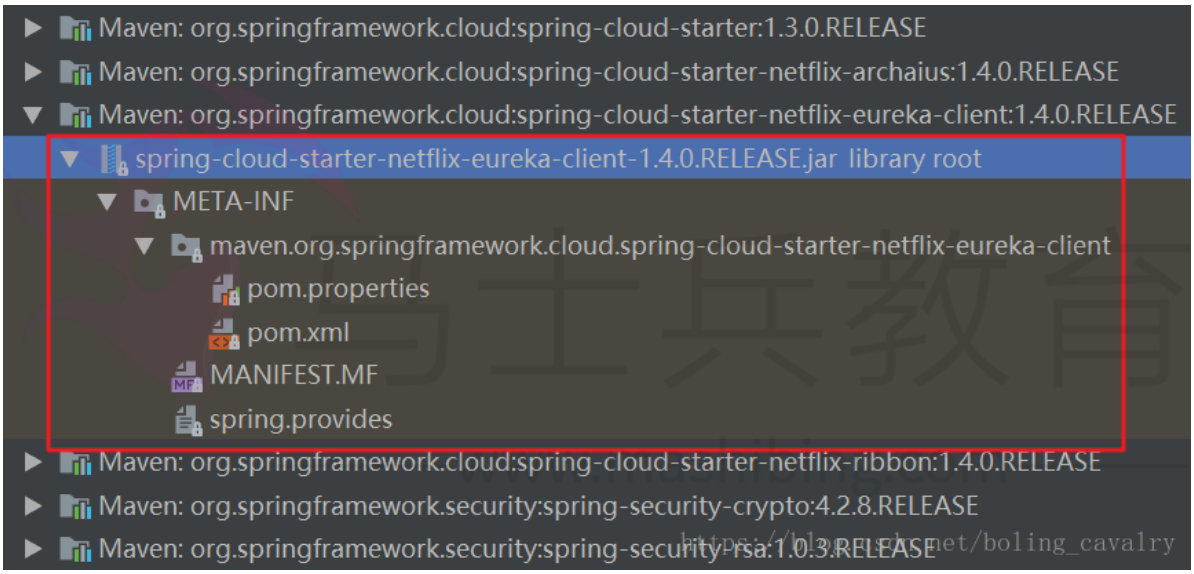
44.4.3 Starter module

The starter is an empty jar, really. Its only purpose is to provide the necessary dependencies to work with the library; see it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are auto-configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high as you should avoid bringing unnecessary dependencies for a typical usage of the library.

www.mashibing.com
https://blog.csdn.net/boling_cavalry

上图红框中说明starter是个空的jar，其作用就是用来提供必要的模块依赖的，来看看spring-cloud-starter-netflix-eureka-client模块是否遵守此规则，如下图，只有配置文件，没有任何class：



为什么应用能自动注册到Eureka sever

作为Eureka client的应用，在启动后就自动注册到Eureka server了，作为应用开发者的我们除了在pom.xml中依赖spring-cloud-starter-netflix-eureka-client模块，没有做其他设置，这是如何实现的呢?

1. 注册到Eureka server的工作，是CloudEurekaClient类在其父类的构造方法中完成的，搜索源码发现此类的在EurekaClientAutoConfiguration中被注册到spring容器，如下图红框所示：

```
@Configuration
@ConditionalOnRefreshScope
protected static class RefreshableEurekaClientConfiguration {

    @Autowired
    private ApplicationContext context;

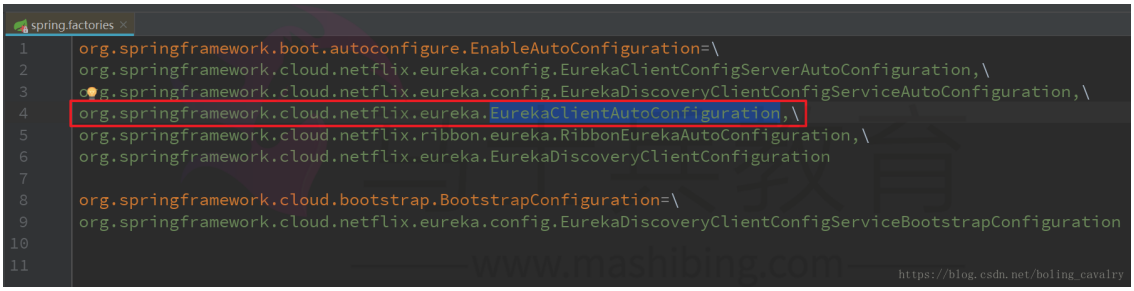
    @Autowired
    private AbstractDiscoveryClientOptionalArgs<?> optionalArgs;

    @Bean(destroyMethod = "shutdown")
    @ConditionalOnMissingBean(value = EurekaClient.class, search = SearchStrategy.CURRENT)
    @org.springframework.cloud.context.config.annotation.RefreshScope
    @Lazy
    public EurekaClient eurekaClient(ApplicationInfoManager manager, EurekaClientConfig config, EurekaInstanceConfig instance) {
        manager.getInfo(); // force initialization
        return new CloudEurekaClient(manager, config, this.optionalArgs,
                this.context);
    }

    @Bean
    @ConditionalOnMissingBean(value = ApplicationInfoManager.class, search = SearchStrategy.CURRENT)
    @org.springframework.cloud.context.config.annotation.RefreshScope
    @Lazy
    public ApplicationInfoManager eurekaApplicationInfoManager(EurekaInstanceConfig config) {
        InstanceInfo instanceInfo = new InstanceInfoFactory().create(config);
        return new ApplicationInfoManager(config, instanceInfo);
    }
}
```

所以，现在问题就变成了如何让EurekaClientAutoConfiguration类被实例化?

1. 在spring-cloud-netflix-eureka-client模块的spring.factories文件中，找到了EurekaClientAutoConfiguration：



这是个spring boot的扩展配置，在此文件中配置的bean都会被实例化，然后注册到spring容器，具体的细节，我们会在第三章结合spring boot源码详细分析，本章只要知道用法即可；

此处小结Eureka client自动注册到Eureka server的过程：
第一、spring-cloud-netflix-eureka-client模块的spring.factories文件中配置了EurekaClientAutoConfiguration，因此EurekaClientAutoConfiguration会被实例化并注册到Spring容器中；
第二、EurekaClientAutoConfiguration中配置了CloudEurekaClient，因此CloudEurekaClient会实例化，在构造方法中执行了注册；

实战的设计

参考了spring cloud的starter设计后，接下来的实战被设计成两个maven工程：customizestarter和customizestartertestdemo；

1. 工程customizestarter里面包含了四个模块，每个模块功能如下所示：

模块名称	作用	备注
customizeapi	包含了接口和异常的定义	实现和调用服务时用到的接口和异常都在此工程中
addservice	提供加法服务	普通的maven工程，里面加法接口的实现类
minusservice	提供减法服务	普通的maven工程，里面有两个减法接口的实现类，一个支持负数，另一个不支持
customizeservicestarter	自定义starter模块	pom.xml中依赖了customizeapi、addservice、minusservice，自身有个Configuration类，通过@Bean注解向spring容器注册AddService和MinusService的实例

1. 工程customizestartertestdemo在pom.xml中依赖了上述的customizeservicestarter模块，提供的web服务会用到addservice和minusservice的服务，并且在应用启动时设置环境变量来选择使用的减法服务是否支持负数；

至此，准备工作已经完成了，对基本原理和开发设计都已经清楚，接下来的章节我们来一起开发上述五个工程；

自定义spring boot starter三部曲之二：实战开发

本文是《自定义spring boot starter三部曲》的第二篇，上一篇中我们通过学习spring cloud的starter，对spring boot的starter有了初步了解，也设计好了实战内容，今天就来一起实现；

本章内容概述

1. 创建工程customizestarter；
2. 创建模块customizeapi；
3. 创建模块addservice；
4. 创建模块minusservice；
5. 创建模块customizeservicestarter；
6. 构建工程customizestarter，并安装到本地maven仓库；
7. 创建工程customizestartertestdemo；
8. 构建工程customizestartertestdemo，得到jar包；
9. 启动customizestartertestdemo工程的jar包，并带上一个启动参数，验证支持负数的减法服务；
10. 启动customizestartertestdemo工程的jar包，验证不支持服务的减法服务；

创建工程customizestarter

1. 创建一个名为customizestarter的maven工程，以spring-boot-starter-parent作为父工程，同时自身又是后续几个模块的父工程，pom.xml内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.bolingcavalry</groupId>
    <artifactId>customizestarter</artifactId>
    <packaging>pom</packaging>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
```

```
        <version>1.5.9.RELEASE</version>
        <relativePath/>
    </parent>

    <modules>
        <!--加法服务-->
        <module>addservice</module>
        <!--减法服务-->
        <module>minusservice</module>
        <!--接口和异常定义-->
        <module>customizeapi</module>
        <!--启动器-->
        <module>customizeservicestarter</module>
    </modules>
</project>
```

创建模块customizeapi

1. 在工程customizestarter下创建模块customizeapi，这是个java工程，里面是加法和减法服务的接口，和一个业务异常的定义；
2. customizeapi的pom.xml内容如下，很简单，只有基本定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>customizestarter</artifactId>
        <groupId>com.bolingcavalry</groupId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../</relativePath>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>customizeapi</artifactId>
</project>
```

1. 异常定义类：

```
package com.bolingcavalry.api.exception;

/**
 * @author wilzhao
 * @description 执行减法服务时抛出的异常
 * @email zq2599@gmail.com
 * @time 2018/10/13 14:20
 */
public class MinusException extends Exception{
    public MinusException(String message) {
        super(message);
    }
}
```

1. 加法服务的接口类AddService：

```
```

```
package com.bolingcavalry.api.service;

/**
 * @author wilzhao
 * @description 加法服务对应的接口
 * @email zq2599@gmail.com
 * @time 2018/10/13 10:07
 */
public interface AddService {
    /**
     * 普通加法
     * @param a
     * @param b
     * @return
     */
    int add(int a, int b);
}
```

1. 减法服务定义类，注意减法API声明了异常抛出，因为如果已经配置了不支持负数的减法服务，那么被减数如果小于减数就抛出异常：

```
package com.bolingcavalry.api.service;

import com.bolingcavalry.api.exception.MinusException;

/**
 * @author wilzhao
 * @description 减法服务
 * @email zq2599@gmail.com
 * @time 2018/10/13 12:07
 */
public interface MinusService {
    /**
     * 普通减法
     * @param minuend 减数
     * @param subtraction 被减数
     * @return 差
     */
    int minus(int minuend, int subtraction) throws MinusException;
}
```

创建模块addservice

1. 在工程customizestarter下创建模块addservice，这是个java工程，里面包含了加法相服务的实现，pom.xml内容如下，注意由于要实现加法接口，因此需要依赖模块customizeapi：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>customizestarter</artifactId>
        <groupId>com.bolingcavalry</groupId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../</relativePath>
```

```
    </parent>
    <artifactId>addservice</artifactId>
    <modelVersion>4.0.0</modelVersion>

    <dependencies>
        <dependency>
            <groupId>com.bolingcavalry</groupId>
            <artifactId>customizeapi</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>
</project>
```

1. 加法接口的实现类AddServiceImpl很简单，如下：

```
package com.bolingcavalry.addservice.service.impl;

import com.bolingcavalry.api.service.AddService;

/**
 * @author wilzhao
 * @description 加法服务的实现
 * @email zq2599@gmail.com
 * @time 2018/10/13 10:59
 */
public class AddServiceImpl implements AddService {
    public int add(int a, int b) {
        return a + b;
    }
}
```

创建模块minusservice

1. 在工程customizestarter下创建模块minusservice，这是个java工程，里面包含了减法相服务的实现，pom.xml内容如下，注意由于要实现减法接口，因此需要依赖模块customizeapi：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>customizestarter</artifactId>
        <groupId>com.bolingcavalry</groupId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../</relativePath>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>minusservice</artifactId>
    <dependencies>
        <dependency>
            <groupId>com.bolingcavalry</groupId>
            <artifactId>customizeapi</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>
```

```
</project>
```

1. 一共有两个减法接口的实现类，第一个不支持负数结果，如果被减数小于减数就抛出异常MinusException：

```
package com.bolingcavalry.minusservice.service.impl;

import com.bolingcavalry.api.exception.MinusException;
import com.bolingcavalry.api.service.MinusService;

/**
 * @author wilzhao
 * @description 减法服务的实现，不支持负数
 * @email zq2599@gmail.com
 * @time 2018/10/13 14:24
 */
public class MinusServiceNotSupportNegativeImpl implements MinusService {

    /**
     * 减法运算，不支持负数结果，如果被减数小于减数，就跑出MinusException
     * @param minuend 被减数
     * @param subtraction 减数
     * @return
     * @throws MinusException
     */
    public int minus(int minuend, int subtraction) throws MinusException {
        if(subtraction>minuend){
            throw new MinusException("not support negative!");
        }

        return minuend-subtraction;
    }
}
```

1. 第二个减法接口的实现类支持负数返回：

```
package com.bolingcavalry.minusservice.service.impl;

import com.bolingcavalry.api.exception.MinusException;
import com.bolingcavalry.api.service.MinusService;

/**
 * @author wilzhao
 * @description 支持负数结果的减法服务
 * @email zq2599@gmail.com
 * @time 2018/10/13 14:30
 */
public class MinusServiceSupportNegativeImpl implements MinusService {

    /**
     * 减法实现，支持负数
     * @param minuend 减数
     * @param subtraction 被减数
     * @return
     * @throws MinusException
     */
```

```
    public int minus(int minuend, int subtraction) throws MinusException {
        return minuend - subtraction;
    }
}
```

创建模块customizeservicestarter

1. 在工程customizestarter下创建模块customizeservicestarter，这是个java工程，里面需要依赖spring boot配置相关的库，由于要在配置中实例化加法和减法服务的实现，因此customizeapi、addservice、minusservice这些模块都要在pom.xml中声明依赖，如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>customizestarter</artifactId>
        <groupId>com.bolingcavalry</groupId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../</relativePath>
    </parent>
    <artifactId>customizeservicestarter</artifactId>
    <modelVersion>4.0.0</modelVersion>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-autoconfigure</artifactId>
            <!--仅编译时才需要-->
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.bolingcavalry</groupId>
            <artifactId>customizeapi</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>com.bolingcavalry</groupId>
            <artifactId>addservice</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>com.bolingcavalry</groupId>
            <artifactId>minusservice</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>
</project>
```

1. 创建配置类CustomizeConfiguration，注意getSupportMinusService和getNotSupportMinusService这两个方法上的注解配置，如果环境变量com.bolingcavalry.supportnegative存在并且等于true，那么getSupportMinusService方法就返回了MinusService接口的实例，如果当前环境没有MinusService接口的实例，就由getNotSupportMinusService方法就返回一个，并且有会在控制台打印创建了哪种实现：

```java
package com.bolingcavalry.customizeservicestarter;

import com.bolingcavalry.addservice.service.impl.AddServiceImpl;
import com.bolingcavalry.api.service.AddService;
import com.bolingcavalry.api.service.MinusService;
import
com.bolingcavalry.minusservice.service.impl.MinusServiceNotSupportNegativeImpl;
import
com.bolingcavalry.minusservice.service.impl.MinusServiceSupportNegativeImpl;
import org.springframework.beans.factory.annotation.Qualifier;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @author wilzhao
 * @description 一句话介绍
 * @email zq2599@gmail.com
 * @time 2018/10/13 14:36
 */
@Configuration
public class CustomizeConfiguration {

    @Bean
    public AddService getAddService(){
        System.out.println("create addService");
        return new AddServiceImpl();
    }

    /**
     * 如果配置了com.bolingcavalry.supportnegative=true，
     * 就实例化MinusServiceSupportNegativeImpl
     * @return
     */
    @Bean
    @ConditionalOnProperty(prefix="com.bolingcavalry",name = "supportnegative",
havingValue = "true")
    public MinusService getSupportMinusService(){
        System.out.println("create minusService support minus");
        return new MinusServiceSupportNegativeImpl();
    }

    /**
     * 如果没有配置com.bolingcavalry.supportnegative=true，
     * 就不会实例化MinusServiceSupportNegativeImpl，
     * 这里的条件是如果没有MinusService类型的bean，就在此实例化一个
     * @return
     */
    @Bean
    @ConditionalOnMissingBean(MinusService.class)
    public MinusService getNotSupportMinusService(){
        System.out.println("create minusService not support minus");
        return new MinusServiceNotSupportNegativeImpl();
    }
}
```

1. 在src\main\resources目录下创建一个目录META-INF，里面创建一个文件spring.factories，内容是如下，表示如果当前应用支持spring boot的自动配置，就会被spring boot框架实例化并注册到spring容器内：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.bolingcavalry
.customizeservicestarter.CustomizeConfiguration
```

构建工程customizestarter

1. 到这里customizestarter工程的编码就结束了，在工程内pom.xml所在目录（也就是customizestarter内的第一层目录），执行以下命令可以编译构建并安装到本地maven仓库：

```
mvn clean install -Dmaven.test.skip=true -U
```

1. 如果编译构建和安装都成功了，可以看到类似如下输出：

```
[INFO] Installing
C:\temp\201810\07\customizestarter\customizeservicestarter\pom.xml to
C:\Users\12167\.m2\repositor
y\com\bolingcavalry\customizeservicestarter\0.0.1-
SNAPSHOT\customizeservicestarter-0.0.1-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] customizestarter ................................... SUCCESS [  0.748 s]
[INFO] customizeapi ....................................... SUCCESS [  3.266 s]
[INFO] addservice ......................................... SUCCESS [  0.427 s]
[INFO] minusservice ....................................... SUCCESS [  0.344 s]
[INFO] customizeservicestarter ............................ SUCCESS [  0.495 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.954 s
[INFO] Finished at: 2018-10-14T00:17:46+08:00
[INFO] Final Memory: 29M/221M
[INFO] ------------------------------------------------------------------------
```

现在starter已经准备好了，我们做一个spring boot的web应用来验证一下；

创建工程customizestartertestdemo

1. 工程customizestartertestdemo是个简单的spring boot应用，pom.xml如下，可见并无特别之处，只是多了customizeservicestarter的依赖：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.bolingcavalry</groupId>
    <artifactId>customizestartertestdemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
```

```
    <name>customizestartertestdemo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>com.bolingcavalry</groupId>
            <artifactId>customizeservicestarter</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

1. 开发一个Controller类，用于调用AddService和MinusService对应的服务：

```
package com.bolingcavalry.customizestartertestdemo.controller;

import com.bolingcavalry.api.exception.MinusException;
import com.bolingcavalry.api.service.AddService;
import com.bolingcavalry.api.service.MinusService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
```

```
/**
 * @author wilzhao
 * @description 调用加法和减法服务的测试类
 * @email zq2599@gmail.com
 * @time 2018/10/13 16:00
 */
@RestController
public class CalculateController {

    @Autowired
    private AddService addService;

    @Autowired
    private MinusService minusService;

    @RequestMapping(value = "/add/{added}/{add}", method = RequestMethod.GET)
    public String add(@PathVariable("added") int added, @PathVariable("add") int
add){
        return added + " 加 " + add + " 等于 : " + addService.add(added, add);
    }

    @RequestMapping(value = "/minus/{minuend}/{subtraction}", method =
RequestMethod.GET)
    public String minus(@PathVariable("minuend") int minuend,
@PathVariable("subtraction") int subtraction) throws MinusException {
        return minuend + " 减 " + subtraction + " 等于 : " +
minusService.minus(minuend, subtraction);
    }
}
```

1. 启动类如下:

```
package com.bolingcavalry.customizestartertestdemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CustomizestartertestdemoApplication {

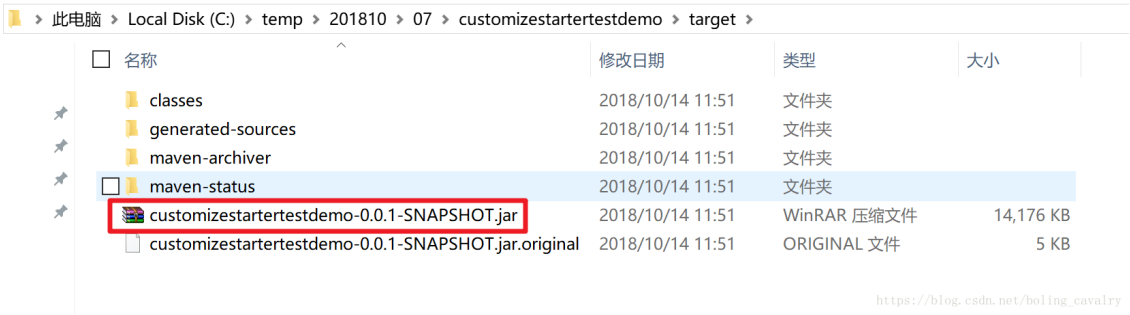
    public static void main(String[] args) {
        SpringApplication.run(CustomizestartertestdemoApplication.class, args);
    }
}
```

构建工程customizestartertestdemo

1. 在customizestartertestdemo工程的pom.xml所在目录下执行以下命令即可构建成功:

```
mvn clean package -Dmaven.test.skip=true
```

1. 命令执行成功后，即可在target目录下见到customizestartertestdemo-0.0.1-SNAPSHOT.jar文件，如下图:



现在编码和构建已经全部完成，我们可以来验证了；

验证支持负数的减法服务

1. 在customizeapi模块的CustomizeConfiguration类中，有如下方法和注解：

```
@Bean
    @ConditionalOnProperty(prefix="com.bolingcavalry",name = "supportnegative",
havingValue = "true")
    public MinusService getSupportMinusService(){
        System.out.println("create minusService support minus");
        return new MinusServiceSupportNegativeImpl();
    }
```

从上述代码可见，只要环境变量"com.bolingcavalry.supportnegative"等于true，注册到spring容器的就是MinusServiceSupportNegativeImpl类的实例；

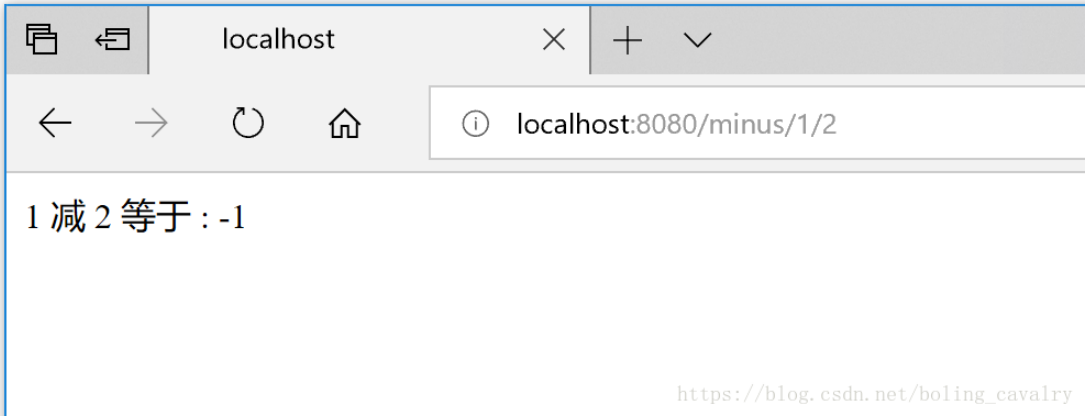
1. customizestartertestdemo-0.0.1-SNAPSHOT.jar文件所在目录下，执行以下命令启动应用：

```
java -Dcom.bolingcavalry.supportnegative=true -jar customizestartertestdemo-
0.0.1-SNAPSHOT.jar
```

1. 在控制台中可以看见create minusService support minus，表示注册到spring容器的是MinusServiceSupportNegativeImpl类的实例，如下所示：

```
2018-10-14 12:04:54.233  INFO 16588 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter'
 to: [/*]
create addService
create minusService support minus
2018-10-14 12:04:54.845  INFO 16588 --- [           main]
s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice:
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplication
Context@443b7951: startup date [Sun Oct 14 12:04:50 CST 2018]; root of context
hierarchy
```

1. 在浏览器访问http://localhost:8080/minus/1/2，可见返回计算结果为负数：



https://blog.csdn.net/boling_cavalry

验证不支持负数的减法服务

1. 前面已经分析过，CustomizeConfiguration类的getNotSupportMinusService方法执行的条件是环境变量"com.bolingcavalry.supportnegative"等于true，如果没有这个环境变量，getNotSupportMinusService方法就不会执行，spring容器中就没有MinusService接口的实例；
2. CustomizeConfiguration类中，有如下方法和注解：

```
    @Bean
    @ConditionalOnMissingBean(MinusService.class)
    public MinusService getNotSupportMinusService(){
        System.out.println("create minusService not support minus");
        return new MinusServiceNotSupportNegativeImpl();
    }
```

从上述代码可见，spring容器中如果没有MinusService接口的实例，getNotSupportMinusService方法就会被执行，在spring容器中注册MinusServiceNotSupportNegativeImpl实例；

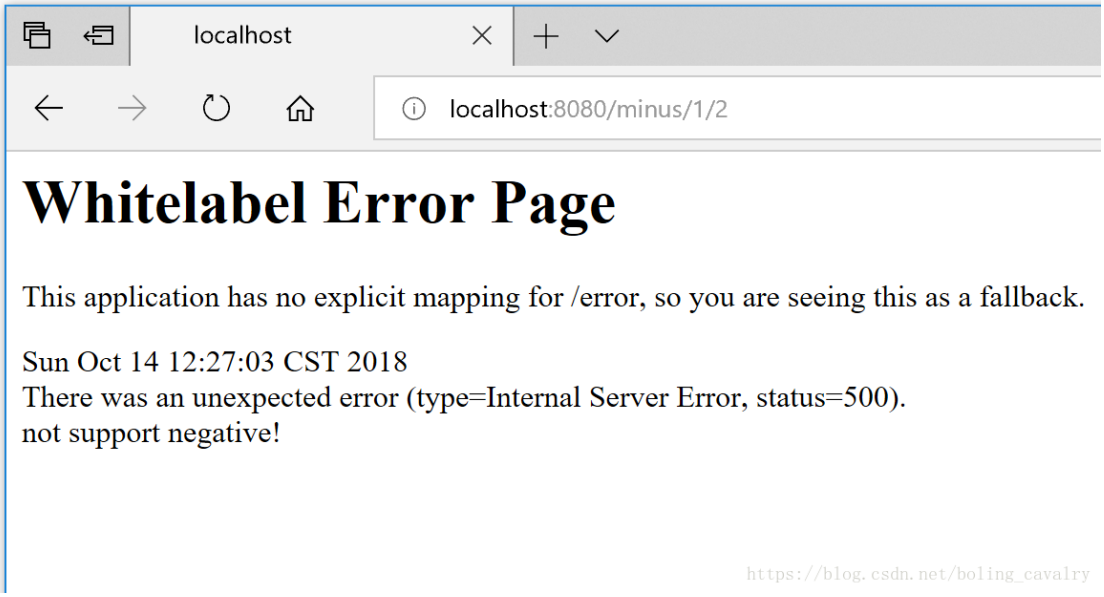
因此接下来的我们启动的应用如果没有环境变量"com.bolingcavalry.supportnegative"，就可以使用到不支持负数的减法服务了；
\2. 停掉之前启动的应用，然后执行以下命令启动应用：

```
java -jar customizestartertestdemo-0.0.1-SNAPSHOT.jar
```

1. 在控制台中可以看见create minusService not support minus，表示注册到spring容器的是MinusServiceNotSupportNegativeImpl类的实例，如下所示：

```
2018-10-14 12:15:05.994  INFO 16608 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter'
 to: [/*]
create addService
create minusService not support minus
2018-10-14 12:15:06.592  INFO 16608 --- [           main]
s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice:
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplication
Context@443b7951: startup date [Sun Oct 14 12:15:02 CST 2018]; root of context
hierarchy
```

1. 在浏览器访问http://localhost:8080/minus/1/2，由于MinusServiceNotSupportNegativeImpl实例不支持负数减法，会直接抛出异常，如下图：



至此，自定义spring boot starter的编码实战就完成了，希望本篇可以给您用来作参考，助您做出自己所需的starter；

下一篇我们一起去看看spring boot的源码，对这个高效的扩展功能做更深入的了解；

自定义spring boot starter三部曲之三：源码分析spring.factories加载过程

版本情况

本文中涉及到的库的版本：

1. Spring boot： 1.5.9.RELEASE；
2. JDK： 1.8.0_144

初步分析

先回顾customizeservicestarter模块中spring.factories文件的内容：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.bolingcavalry
.customizeservicestarter.CustomizeConfiguration
```

从上述内容可以确定今天源码学习目标：

1. spring容器如何处理配置类；
2. spring boot配置类的加载情况；
3. spring.factories中的EnableAutoConfiguration配置何时被加载？
4. spring.factories中的EnableAutoConfiguration配置被加载后做了什么处理；

spring容器如何处理配置类

1. ConfigurationClassPostProcessor类的职责是处理配置类；
2. ConfigurationClassPostProcessor是BeanDefinitionRegistryPostProcessor接口的实现类，它的postProcessBeanDefinitionRegistry方法在容器初始化阶段会被调用；
3. postProcessBeanDefinitionRegistry方法又调用processConfigBeanDefinitions方法处理具体业务；

4. processConfigBeanDefinitions方法中通过ConfigurationClassParser类来处理Configuration注解，如下图：



5. 如上图红框所示，所有被Configuration注解修饰过的类，都会被parser.parse(candidates)处理，即ConfigurationClassParser类的parse方法；
6. parse方法中调用processDeferredImportSelectors方法做处理：找到Configuration类中的Import注解，对于Import注解的值，如果实现了ImportSelector接口，就调用其selectImports方法，将返回的名称实例化：

```
private void processDeferredImportSelectors() {
        //这里就是Configuration注解中的Import注解的值，
        //例如EnableAutoConfiguration注解的源码中，Import注解的值是
EnableAutoConfigurationImportSelector.class
        List<DeferredImportSelectorHolder> deferredImports =
this.deferredImportSelectors;
        this.deferredImportSelectors = null;
        Collections.sort(deferredImports, DEFERRED_IMPORT_COMPARATOR);

        for (DeferredImportSelectorHolder deferredImport : deferredImports) {
            ConfigurationClass configClass =
deferredImport.getConfigurationClass();
            try {
                //以EnableAutoConfiguration注解为例，其Import注解的值为
EnableAutoConfigurationImportSelector.class，
                //那么此处就是在调用EnableAutoConfigurationImportSelector的
selectImports方法，返回了一个字符串数组
                String[] imports =
deferredImport.getImportSelector().selectImports(configClass.getMetadata());
                //字符串数组中的每个字符串都代表一个类，此处做实例化
                processImports(configClass, asSourceClass(configClass),
asSourceClasses(imports), false);
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                        "Failed to process import candidates for configuration
class [" +
                        configClass.getMetadata().getClassName() + "]", ex);
            }
        }
    }
```

小结一下spring容器配置类的逻辑：

1. 找出配置类；
2. 找出配置类中的Import注解；

3. Import注解的值是class，如果该class实现了ImportSelector接口，就调用其selectImports方法，将返回的名称实例化；

有了上面的结论就可以结合Spring boot的源码来分析加载了哪些数据了；

spring boot配置类的加载情况

1. 我们的应用使用了SpringBootApplication注解，看此注解的源码，使用了EnableAutoConfiguration注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
    ......
```

2. EnableAutoConfiguration注解中，通过Import注解引入了EnableAutoConfigurationImportSelector.class：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    ......
```

3. 看EnableAutoConfigurationImportSelector的源码：

```
/**
 * {@link DeferredImportSelector} to handle {@link EnableAutoConfiguration
 * auto-configuration}. This class can also be subclassed if a custom variant of
 * {@link EnableAutoConfiguration @EnableAutoConfiguration}. is needed.
 *
 * @deprecated as of 1.5 in favor of {@link AutoConfigurationImportSelector}
 * @author Phillip Webb
 * @author Andy Wilkinson
 * @author Stephane Nicoll
 * @author Madhura Bhave
 * @since 1.3.0
 * @see EnableAutoConfiguration
 */
@Deprecated
public class EnableAutoConfigurationImportSelector
        extends AutoConfigurationImportSelector {

    @Override
    protected boolean isEnabled(AnnotationMetadata metadata) {
```

```
        if (getClass().equals(EnableAutoConfigurationImportSelector.class)) {
            return getEnvironment().getProperty(
                    EnableAutoConfiguration.ENABLED_OVERRIDE_PROPERTY,
Boolean.class,
                    true);
        }
        return true;
    }

}
```

上述源码有三处重点需要关注：
第一，EnableAutoConfigurationImportSelector是AutoConfigurationImportSelector的子类；
第二，EnableAutoConfigurationImportSelector已经被废弃了，不推荐使用；
第三，文档中已经写明废弃原因：从1.5版本开始，其特性由父类AutoConfigurationImportSelector实现；

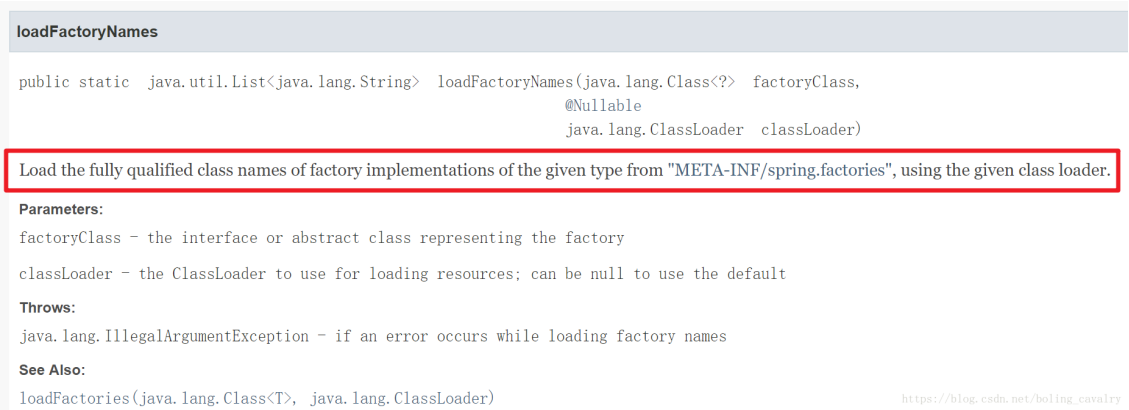
4. 查看AutoConfigurationImportSelector的源码，重点关注selectImports方法，该方法的返回值表明了哪些类会被实例化：

```
@Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return NO_IMPORTS;
        }
        try {
            //将所有spring-autoconfigure-metadata.properties文件中的键值对保存在
autoConfigurationMetadata中
            AutoConfigurationMetadata autoConfigurationMetadata =
AutoConfigurationMetadataLoader
                    .loadMetadata(this.beanClassLoader);
            AnnotationAttributes attributes = getAttributes(annotationMetadata);
            //取得所有配置类的名称
            List<String> configurations =
getCandidateConfigurations(annotationMetadata,
                    attributes);
            configurations = removeDuplicates(configurations);
            configurations = sort(configurations, autoConfigurationMetadata);
            Set<String> exclusions = getExclusions(annotationMetadata,
attributes);
            checkExcludedClasses(configurations, exclusions);
            configurations.removeAll(exclusions);
            configurations = filter(configurations, autoConfigurationMetadata);
            fireAutoConfigurationImportEvents(configurations, exclusions);
            return configurations.toArray(new String[configurations.size()]);
        }
        catch (IOException ex) {
            throw new IllegalStateException(ex);
        }
    }
```

5. 通过上述代码可以发现，getCandidateConfigurations方法的调用是个关键，它返回的字符串都是即将被实例化的类名，来看此方法源码：

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
            AnnotationAttributes attributes) {
        List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
                getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
        Assert.notEmpty(configurations,
                "No auto configuration classes found in META-
INF/spring.factories. If you "
                        + "are using a custom packaging, make sure that file is
correct.");
        return configurations;
    }
```

6. getCandidateConfigurations方法中，调用了静态方法SpringFactoriesLoader.loadFactoryNames，上面提到的SpringFactoriesLoader.loadFactoryNames方法是关键，看看官方文档对此静态方法的描述，如下图红框所示，该方法会在spring.factories文件中寻找指定接口对应的实现类的全名（包名+实现类）：



loadFactoryNames

public static java.util.List<java.lang.String> loadFactoryNames(java.lang.Class<?> factoryClass,
@Nullable
java.lang.ClassLoader classLoader)

Load the fully qualified class names of factory implementations of the given type from "META-INF/spring.factories", using the given class loader.

Parameters:

factoryClass - the interface or abstract class representing the factory

classLoader - the ClassLoader to use for loading resources; can be null to use the default

Throws:

java.lang.IllegalArgumentException - if an error occurs while loading factory names

See Also:

loadFactories(java.lang.Class<T>, java.lang.ClassLoader)

https://blog.csdn.net/boling_cavalry

7. 在getCandidateConfigurations方法中，调用SpringFactoriesLoader.loadFactoryNames的时候传入的指定类型是getSpringFactoriesLoaderFactoryClass方法的返回值：

```
protected Class<?> getSpringFactoriesLoaderFactoryClass() {
        return EnableAutoConfiguration.class;
    }
```

现在可以梳理一下了：

1. spring boot应用启动时使用了EnableAutoConfiguration注解；
2. EnableAutoConfiguration注解通过import注解将EnableAutoConfigurationImportSelector类实例化，并且将其selectImports方法返回的类名实例化后注册到spring容器；
3. EnableAutoConfigurationImportSelector的selectImports方法返回的类名，来自spring.factories文件内的配置信息，这些配置信息的key等于EnableAutoConfiguration；

现在真相大白了：只要我们在spring.factories文件内配置了EnableAutoConfiguration，那么对于的类就会被实例化后注册到spring容器；

至此，《自定义spring boot starter三部曲》系列就完结了，希望实战加源码分析的三篇文章，能帮助您理解和实现自定义starter这种简单快捷的扩展方式；

Spring IOC是什么？优点是什么？

IoC文英全称Inversion of Control，即控制反转，我么可以这么理解IoC容器：“把某些业务对 象的的控制权交给一个平台或者框架来同一管理，这个同一管理的平台可以称为IoC容器。”

ioc的思想最核心的地方在于，资源不由使用资源的双方管理，而由不使用资源的第三方管理， 这可以带来很多好处：

1）资源集中管理，实现资源的可配置和易管理

2）降低了使用资源双方的依赖程度，也就是我们说的耦合度

Spring IOC原理解读 面试必读

一、什么是Ioc/DI?

IoC 容器：最主要是完成了完成对象的创建和依赖的管理注入等等。

先从我们自己设计这样一个视角来考虑：

所谓控制反转，就是把原先我们代码里面需要实现的对象创建、依赖的代码，反转给容器来帮忙实现。那么必然的我们需要创建一个容器，同时需要一种描述来让容器知道需要创建的对象与对象的关系。这个描述最具体表现就是我们可配置的文件。

对象和对象关系怎么表示?

可以用 xml， properties 文件等语义化配置文件表示。

描述对象关系的文件存放在哪里?

可能是 classpath， filesystem，或者是 URL 网络资源， servletContext 等。

回到正题，有了配置文件，还需要对配置文件解析。

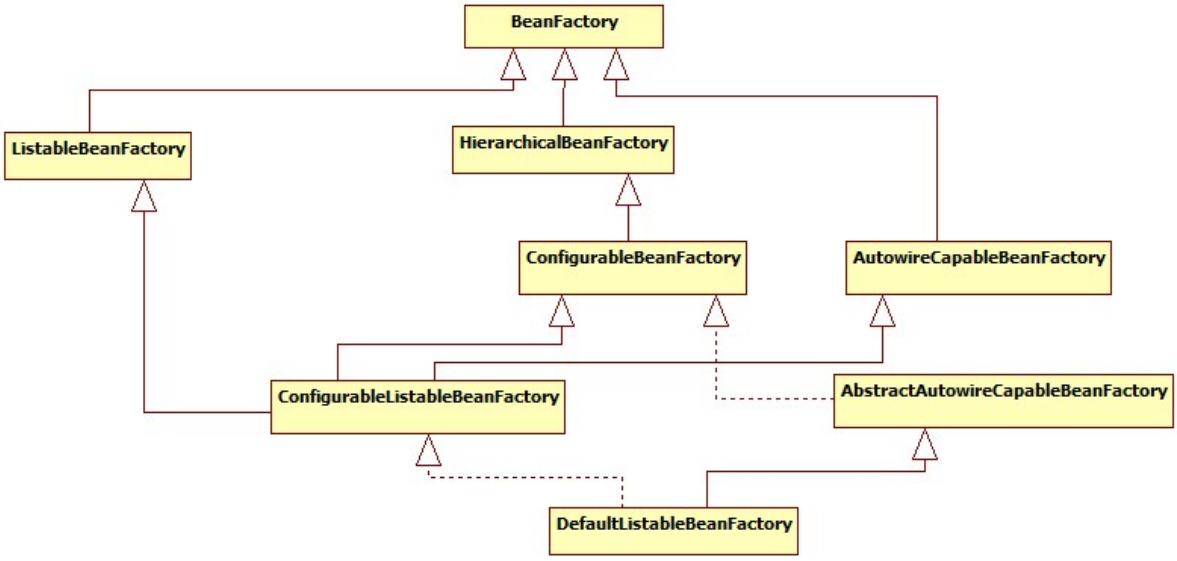
不同的配置文件对对象的描述不一样，如标准的，自定义声明式的，如何统一？ 在内部需要有一个统一的关于对象的定义，所有外部的描述都必须转化成统一的描述定义。

如何对不同的配置文件进行解析？需要对不同的配置文件语法，采用不同的解析器

二、 Spring IOC体系结构?

(1) BeanFactory

> Spring Bean的创建是典型的工厂模式，这一系列的Bean工厂，也即IOC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IOC容器的实现供用户选择和使用，其相互关系如下：



其中BeanFactory作为最顶层的一个接口类，它定义了IOC容器的基本功能规范，BeanFactory 有三个子类：ListableBeanFactory、HierarchicalBeanFactory 和AutowireCapableBeanFactory。但是从上图中我们可以发现最终的默认实现类是 DefaultListableBeanFactory，他实现了所有的接口。那为何要定义这么多层次的接口呢？查阅这些接口的源码和说明发现，每个接口都有他使用的场合，它主要是为了区分在 Spring 内部在操作过程中对象的传递和转化过程中，对对象的数据访问所做的限制。例如 ListableBeanFactory 接口表示这些 Bean 是可列表的，而 HierarchicalBeanFactory 表示的是这些 Bean 是有继承关系的，也就是每个Bean 有可能有父 Bean。AutowireCapableBeanFactory 接口定义 Bean 的自动装配规则。这四个接口共同定义了 Bean 的集合、Bean 之间的关系、以及 Bean 行为.

最基本的IOC容器接口BeanFactory

```
1 public interface BeanFactory {
2
3       //对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成的对象，
4       //如果需要得到工厂本身，需要转义
5       String FACTORY_BEAN_PREFIX = "&";
6
7       //根据bean的名字，获取在IOC容器中得到bean实例
8       Object getBean(String name) throws BeansException;
9
10      //根据bean的名字和Class类型来得到bean实例，增加了类型安全验证机制。
11       Object getBean(String name, Class requiredType) throws BeansException;

12
13      //提供对bean的检索，看看是否在IOC容器有这个名字的bean
14       boolean containsBean(String name);
15
16      //根据bean名字得到bean实例，并同时判断这个bean是不是单例
17      boolean isSingleton(String name) throws NoSuchBeanDefinitionException;

18
19      //得到bean实例的Class类型
20      Class getType(String name) throws NoSuchBeanDefinitionException;
21
22      //得到bean的别名，如果根据别名检索，那么其原名也会被检索出来
```

```
23    String[] getAliases(String name);
24
 }
```

在BeanFactory里只对IOC容器的基本行为作了定义，根本不关心你的bean是如何定义怎样加载的。正如我们只关心工厂里得到什么的产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心。

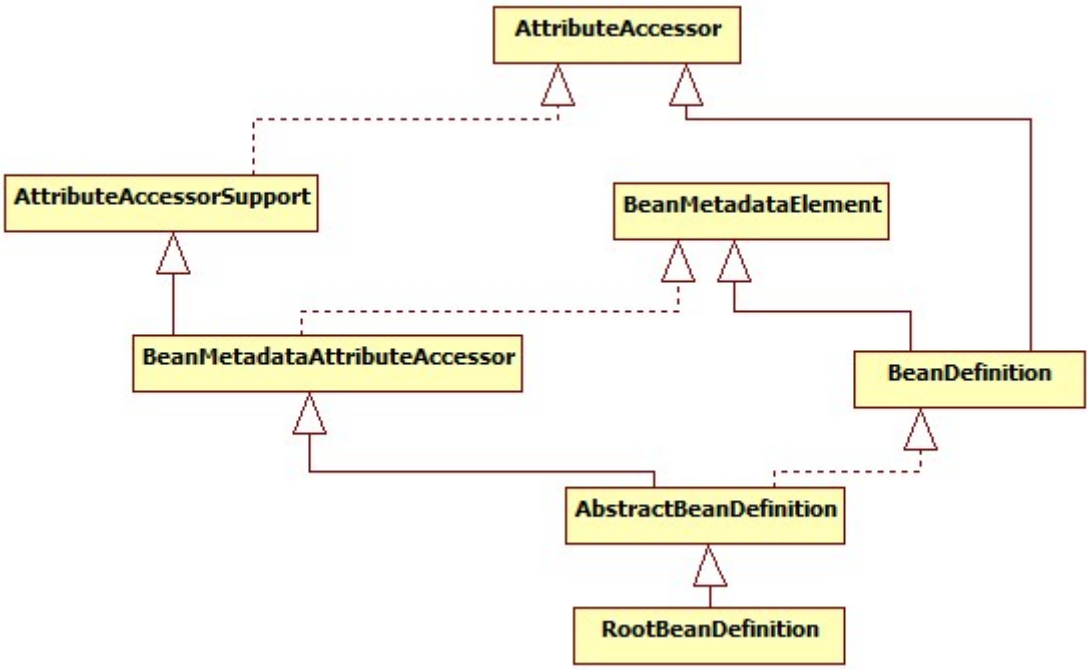
> 而要知道工厂是如何产生对象的，我们需要看具体的IOC容器实现，spring提供了许多IOC容器的实现。比如XmlBeanFactory，ClasspathXmlApplicationContext等。其中XmlBeanFactory就是针对最基本的ioc容器的实现，这个IOC容器可以读取XML文件定义的BeanDefinition（XML文件中对bean的描述）,如果说XmlBeanFactory是容器中的屌丝，ApplicationContext应该算容器中的高帅富.
>
> ApplicationContext是Spring提供的一个高级的IoC容器，它除了能够提供IoC容器的基本功能外，还为用户提供了以下的附加服务。

从ApplicationContext接口的实现，我们看出其特点：

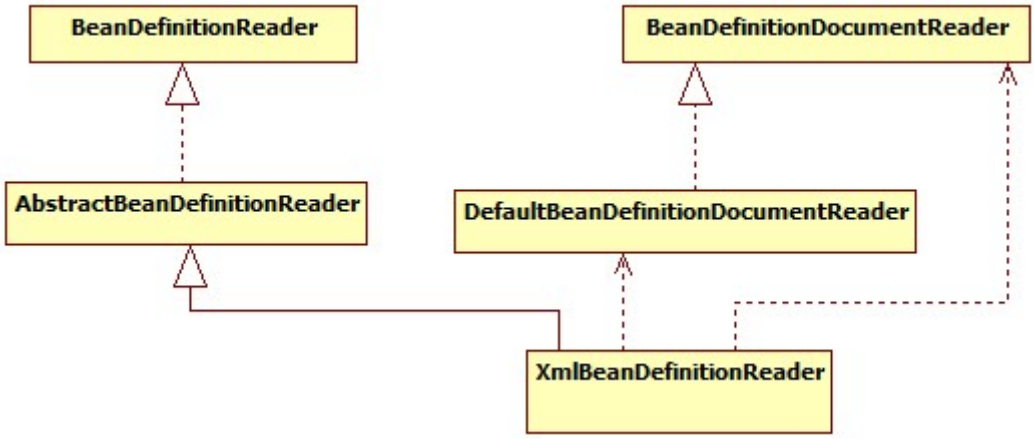
> 1. 支持信息源，可以实现国际化。（实现MessageSource接口）
>
> 2. 访问资源。(实现ResourcePatternResolver接口，这个后面要讲)
>
> 3. 支持应用事件。(实现ApplicationEventPublisher接口)

(2) BeanDefinition

> SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Bean对象在Spring实现中是以BeanDefinition来描述的，其继承体系如下：

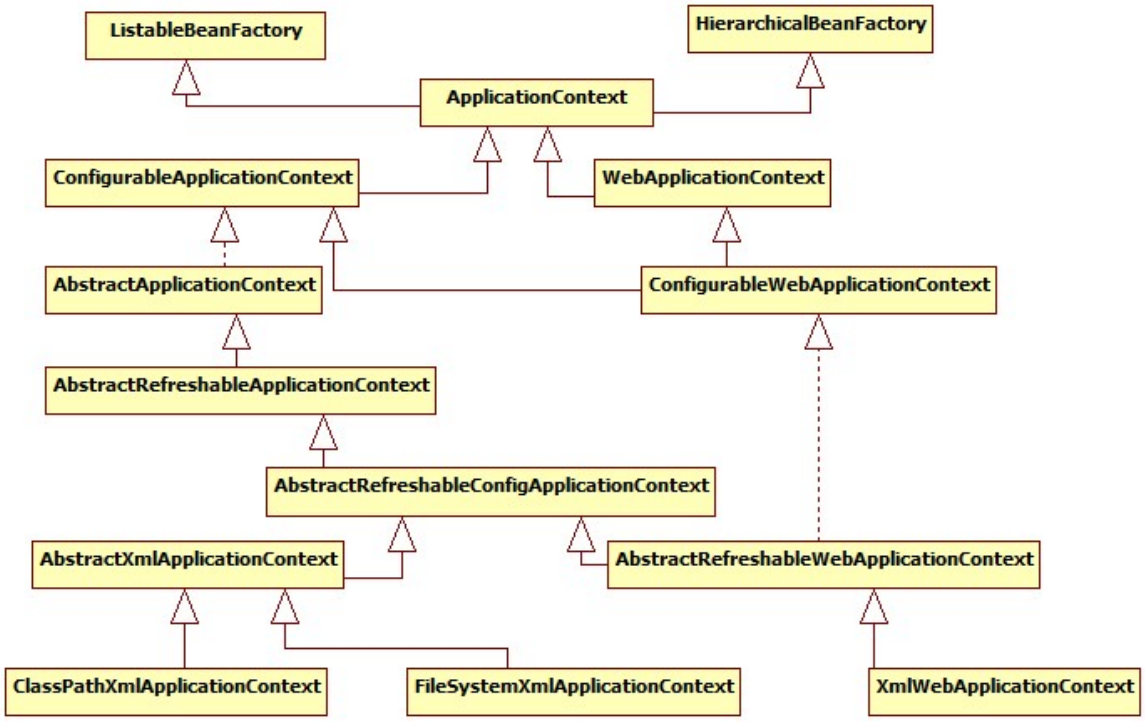


Bean 的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean 的解析主要就是对 Spring 配置文件的解析。这个解析过程主要通过下图中的类完成：



三、IoC容器的初始化?

IoC容器的初始化包括BeanDefinition的Resource定位、载入和注册这三个基本的过程。我们以ApplicationContext为例讲解，ApplicationContext系列容器也许是我们最熟悉的，因为web项目中使用的XmlWebApplicationContext就属于这个继承体系，还有ClasspathXmlApplicationContext等，其继承体系如下图所示：



ApplicationContext允许上下文嵌套，通过保持父上下文可以维持一个上下文体系。对于bean的查找可以在这个上下文体系中发生，首先检查当前上下文，其次是父上下文，逐级向上，这样为不同的Spring应用提供了一个共享的bean定义环境。

下面我们分别简单地演示一下两种ioc容器的创建过程

1、XmlBeanFactory(屌丝IOC)的整个流程

通过XmlBeanFactory的源码，我们可以发现:

```
public class XmlBeanFactory extends DefaultListableBeanFactory{

    private final XmlBeanDefinitionReader reader;
```

```
    public XmlBeanFactory(Resource resource)throws BeansException{
        this(resource, null);
    }

    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory)
         throws BeansException{
        super(parentBeanFactory);
        this.reader = new XmlBeanDefinitionReader(this);
        this.reader.loadBeanDefinitions(resource);
    }
}
```

```
//根据Xml配置文件创建Resource资源对象，该对象中包含了BeanDefinition的信息
 ClassPathResource resource =new ClassPathResource("application-context.xml");
//创建DefaultListableBeanFactory
 DefaultListableBeanFactory factory =new DefaultListableBeanFactory();
//创建XmlBeanDefinitionReader读取器，用于载入BeanDefinition。之所以需要BeanFactory作为参数，是因为会将读取的信息回调配置给factory
 XmlBeanDefinitionReader reader =new XmlBeanDefinitionReader(factory);
//XmlBeanDefinitionReader执行载入BeanDefinition的方法，最后会完成Bean的载入和注册。完成后Bean就成功的放置到IOC容器当中，以后我们就可以从中取得Bean来使用
 reader.loadBeanDefinitions(resource);
```

通过前面的源码，this.reader = new XmlBeanDefinitionReader(this); 中其中this 传的是factory对象

2、FileSystemXmlApplicationContext 的IOC容器流程

1、高富帅IOC解剖

1 ApplicationContext =new FileSystemXmlApplicationContext(xmlPath);

先看其构造函数：

调用构造函数：

```
/**
* Create a new FileSystemXmlApplicationContext, loading the definitions
* from the given XML files and automatically refreshing the context.
* @param configLocations array of file paths
* @throws BeansException if context creation failed
 */public FileSystemXmlApplicationContext(String... configLocations) throws
BeansException {
        this(configLocations, true, null);
    }
```

实际调用

```
public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh,
ApplicationContext parent)
            throws BeansException {
        super(parent);
        setConfigLocations(configLocations);
        if (refresh) {
            refresh();
        }
    }
```

2、设置资源加载器和资源定位

通过分析FileSystemXmlApplicationContext的源代码可以知道，在创建FileSystemXmlApplicationContext容器时，构造方法做以下两项重要工作：

首先，调用父类容器的构造方法(super(parent)方法)为容器设置好Bean资源加载器。

然后，再调用父类AbstractRefreshableConfigApplicationContext的setConfigLocations(configLocations)方法设置Bean定义资源文件的定位路径。

通过追踪FileSystemXmlApplicationContext的继承体系，发现其父类的父类AbstractApplicationContext中初始化IoC容器所做的主要源码如下：

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader
        implements ConfigurableApplicationContext, DisposableBean {
    //静态初始化块，在整个容器创建过程中只执行一次
    static {
        //为了避免应用程序在Weblogic8.1关闭时出现类加载异常加载问题，加载IoC容
       //器关闭事件(ContextClosedEvent)类
        ContextClosedEvent.class.getName();
    }
    //FileSystemXmlApplicationContext调用父类构造方法调用的就是该方法
    public AbstractApplicationContext(ApplicationContext parent) {
        this.parent = parent;
        this.resourcePatternResolver = getResourcePatternResolver();
    }
    //获取一个Spring Source的加载器用于读入Spring Bean定义资源文件
    protected ResourcePatternResolver getResourcePatternResolver() {
        // AbstractApplicationContext继承DefaultResourceLoader，也是一个S
        //Spring资源加载器，其getResource(String location)方法用于载入资源
        return new PathMatchingResourcePatternResolver(this);
    }
……
}
```

AbstractApplicationContext构造方法中调用PathMatchingResourcePatternResolver的构造方法创建Spring资源加载器：

```
public PathMatchingResourcePatternResolver(ResourceLoader resourceLoader) {
        Assert.notNull(resourceLoader, "ResourceLoader must not be null");
        //设置Spring的资源加载器
        this.resourceLoader = resourceLoader;
}
```

在设置容器的资源加载器之后，接下来FileSystemXmlApplicationContet执行setConfigLocations方法通过调用其父类AbstractRefreshableConfigApplicationContext的方法进行对Bean定义资源文件的定位，该方法的源码如下：

```
    //处理单个资源文件路径为一个字符串的情况
    public void setConfigLocation(String location) {
       //String CONFIG_LOCATION_DELIMITERS = ",; /t/n";
       //即多个资源文件路径之间用” ,; /t/n”分隔，解析成数组形式
        setConfigLocations(StringUtils.tokenizeToStringArray(location,
CONFIG_LOCATION_DELIMITERS));
    }
    //解析Bean定义资源文件的路径，处理多个资源文件字符串数组
     public void setConfigLocations(String[] locations) {
        if (locations != null) {
            Assert.noNullElements(locations, "Config locations must not be
null");
            this.configLocations = new String[locations.length];
            for (int i = 0; i < locations.length; i++) {
                // resolvePath为同一个类中将字符串解析为路径的方法
                this.configLocations[i] = resolvePath(locations[i]).trim();
            }
        }
        else {
            this.configLocations = null;
        }  www
    }
```

通过这两个方法的源码我们可以看出，我们既可以使用一个字符串来配置多个Spring Bean定义资源文件，也可以使用字符串数组，即下面两种方式都是可以的：

a. ClasspathResource res = new ClasspathResource(“a.xml,b.xml,……”);

多个资源文件路径之间可以是用” ,; /t/n”等分隔。

b. ClasspathResource res = new ClasspathResource(newString[]{“a.xml”,”b.xml”,……});

至此，Spring IoC容器在初始化时将配置的Bean定义资源文件定位为Spring封装的Resource。

3、AbstractApplicationContext的refresh函数载入Bean定义过程：

Spring IoC容器对Bean定义资源的载入是从refresh()函数开始的，refresh()是一个模板方法，refresh()方法的作用是：在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器。refresh的作用类似于对IoC容器的重启，在新建立好的容器中对容器进行初始化，对Bean定义资源进行载入

FileSystemXmlApplicationContext通过调用其父类AbstractApplicationContext的refresh()函数启动整个IoC容器对Bean定义的载入过程：

```
1      public void refresh() throws BeansException, IllegalStateException {
2        synchronized (this.startupShutdownMonitor) {
3            //调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识
4            prepareRefresh();
5            //告诉子类启动refreshBeanFactory()方法，Bean定义资源文件的载入从
6           //子类的refreshBeanFactory()方法启动
7            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();
8            //为BeanFactory配置容器特性，例如类加载器、事件处理器等
9            prepareBeanFactory(beanFactory);
10            try {
11                //为容器的某些子类指定特殊的BeanPost事件处理器
12                postProcessBeanFactory(beanFactory);
13                //调用所有注册的BeanFactoryPostProcessor的Bean
14                invokeBeanFactoryPostProcessors(beanFactory);
15                //为BeanFactory注册BeanPost事件处理器.
16                //BeanPostProcessor是Bean后置处理器，用于监听容器触发的事件
17                registerBeanPostProcessors(beanFactory);
18                //初始化信息源，和国际化相关.
19                initMessageSource();
20                //初始化容器事件传播器.
21                initApplicationEventMulticaster();
22                //调用子类的某些特殊Bean初始化方法
23                onRefresh();
24                //为事件传播器注册事件监听器.
25                registerListeners();
26                //初始化所有剩余的单态Bean.
27                finishBeanFactoryInitialization(beanFactory);
28                //初始化容器的生命周期事件处理器，并发布容器的生命周期事件
29                finishRefresh();
30            }
31            catch (BeansException ex) {
32                //销毁以创建的单态Bean
33                destroyBeans();
34                //取消refresh操作，重置容器的同步标识.
35                cancelRefresh(ex);
36                throw ex;
37            }
38        }
39    }
```

refresh()方法主要为IoC容器Bean的生命周期管理提供条件，Spring IoC容器载入Bean定义资源文件从其子类容器的refreshBeanFactory()方法启动，所以整个refresh()中“ConfigurableListableBeanFactory beanFactory =obtainFreshBeanFactory();”这句以后代码的都是注册容器的信息源和生命周期事件，载入过程就是从这句代码启动。

refresh()方法的作用是：在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器。refresh的作用类似于对IoC容器的重启，在新建立好的容器中对容器进行初始化，对Bean定义资源进行载入

AbstractApplicationContext的obtainFreshBeanFactory()方法调用子类容器的refreshBeanFactory()方法，启动容器载入Bean定义资源文件的过程，代码如下：

```
    protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
        //这里使用了委派设计模式，父类定义了抽象的refreshBeanFactory()方法，具体实现调用子类容器的refreshBeanFactory()方法
         refreshBeanFactory();
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
        if (logger.isDebugEnabled()) {
            logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
        }
        return beanFactory;
    }
```

AbstractApplicationContext子类的refreshBeanFactory()方法：

AbstractApplicationContext类中只抽象定义了refreshBeanFactory()方法，容器真正调用的是其子类AbstractRefreshableApplicationContext实现的 refreshBeanFactory()方法，方法的源码如下：

```
1    protected final void refreshBeanFactory() throws BeansException {
2        if (hasBeanFactory()) {//如果已经有容器，销毁容器中的bean，关闭容器
3            destroyBeans();
4            closeBeanFactory();
5        }
6        try {
7             //创建IoC容器
8             DefaultListableBeanFactory beanFactory = createBeanFactory();
9             beanFactory.setSerializationId(getId());
10            //对IoC容器进行定制化，如设置启动参数，开启注解的自动装配等
11            customizeBeanFactory(beanFactory);
12            //调用载入Bean定义的方法，主要这里又使用了一个委派模式，在当前类中只定义了抽象的loadBeanDefinitions方法，具体的实现调用子类容器
13            loadBeanDefinitions(beanFactory);
14            synchronized (this.beanFactoryMonitor) {
15                this.beanFactory = beanFactory;
16            }
17        }
18        catch (IOException ex) {
19            throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
20        }
21    }
```

在这个方法中，先判断BeanFactory是否存在，如果存在则先销毁beans并关闭beanFactory，接着创建DefaultListableBeanFactory，并调用loadBeanDefinitions(beanFactory)装载bean

定义。

5、AbstractRefreshableApplicationContext子类的loadBeanDefinitions方法：

AbstractRefreshableApplicationContext中只定义了抽象的loadBeanDefinitions方法，容器真正调用的是其子类AbstractXmlApplicationContext对该方法的实现，AbstractXmlApplicationContext的主要源码如下：

loadBeanDefinitions方法同样是抽象方法，是由其子类实现的，也即在AbstractXmlApplicationContext中。

```
1  public abstract class AbstractXmlApplicationContext extends
AbstractRefreshableConfigApplicationContext {
2      ……
3      //实现父类抽象的载入Bean定义方法
4      @Override
5      protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
throws BeansException, IOException {
6          //创建XmlBeanDefinitionReader，即创建Bean读取器，并通过回调设置到容器中去，容
器使用该读取器读取Bean定义资源
7          XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);
8          //为Bean读取器设置Spring资源加载器，AbstractXmlApplicationContext的
9          //祖先父类AbstractApplicationContext继承DefaultResourceLoader，因此，容器本
身也是一个资源加载器
10         beanDefinitionReader.setResourceLoader(this);
11         //为Bean读取器设置SAX xml解析器
```

```
12        beanDefinitionReader.setEntityResolver(new
ResourceEntityResolver(this));
13        //当Bean读取器读取Bean定义的Xml资源文件时，启用Xml的校验机制
14        initBeanDefinitionReader(beanDefinitionReader);
15        //Bean读取器真正实现加载的方法
16        loadBeanDefinitions(beanDefinitionReader);
17    }
18    //Xml Bean读取器加载Bean定义资源
19    protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
BeansException, IOException {
20        //获取Bean定义资源的定位
21        Resource[] configResources = getConfigResources();
22        if (configResources != null) {
23            //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位
24            //的Bean定义资源
25            reader.loadBeanDefinitions(configResources);
26        }
27        //如果子类中获取的Bean定义资源定位为空，则获取FileSystemXmlApplicationContext
构造方法中setConfigLocations方法设置的资源
28        String[] configLocations = getConfigLocations();
29        if (configLocations != null) {
30            //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位
31            //的Bean定义资源
32            reader.loadBeanDefinitions(configLocations);
33        }
34    }
35    //这里又使用了一个委托模式，调用子类的获取Bean定义资源定位的方法
36    //该方法在ClassPathXmlApplicationContext中进行实现，对于我们
37    //举例分析源码的FileSystemXmlApplicationContext没有使用该方法
38    protected Resource[] getConfigResources() {
39        return null;
40    }   ……
41}
```

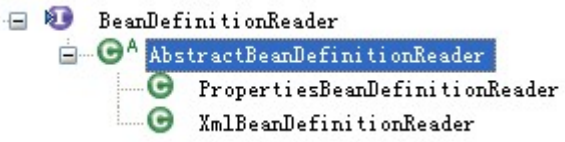
Xml Bean读取器(XmlBeanDefinitionReader)调用其父类AbstractBeanDefinitionReader的reader.loadBeanDefinitions方法读取Bean定义资源。

由于我们使用FileSystemXmlApplicationContext作为例子分析，因此getConfigResources的返回值为null，因此程序执行reader.loadBeanDefinitions(configLocations)分支。

6、AbstractBeanDefinitionReader读取Bean定义资源：

AbstractBeanDefinitionReader的loadBeanDefinitions方法源码如下：

可以到org.springframework.beans.factory.support看一下BeanDefinitionReader的结构



在其抽象父类AbstractBeanDefinitionReader中定义了载入过程

```
1    //重载方法，调用下面的loadBeanDefinitions(String, Set<Resource>);方法
2    public int loadBeanDefinitions(String location) throws
BeanDefinitionStoreException {
3        return loadBeanDefinitions(location, null);
4    }
5    public int loadBeanDefinitions(String location, Set<Resource>
actualResources) throws BeanDefinitionStoreException {
6        //获取在IoC容器初始化过程中设置的资源加载器
7        ResourceLoader resourceLoader = getResourceLoader();
8        if (resourceLoader == null) {
9            throw new BeanDefinitionStoreException(
10                   "Cannot import bean definitions from location [" + location
+ "]: no ResourceLoader available");
11        }
12        if (resourceLoader instanceof ResourcePatternResolver) {
13            try {
14                //将指定位置的Bean定义资源文件解析为Spring IoC容器封装的资源
15                //加载多个指定位置的Bean定义资源文件
16                Resource[] resources = ((ResourcePatternResolver)
resourceLoader).getResources(location);
17                //委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能
18                int loadCount = loadBeanDefinitions(resources);
19                if (actualResources != null) {
20                    for (Resource resource : resources) {
21                        actualResources.add(resource);
22                    }
23                }
24                if (logger.isDebugEnabled()) {
25                    logger.debug("Loaded " + loadCount + " bean definitions
from location pattern [" + location + "]");
26                }
27                return loadCount;
28            }
```

```
29            catch (IOException ex) {
30                throw new BeanDefinitionStoreException(
31                        "Could not resolve bean definition resource pattern ["
+ location + "]", ex);
32            }
33        }
34        else {
35            //将指定位置的Bean定义资源文件解析为Spring IoC容器封装的资源
36            //加载单个指定位置的Bean定义资源文件
37            Resource resource = resourceLoader.getResource(location);
38            //委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能
39            int loadCount = loadBeanDefinitions(resource);
40            if (actualResources != null) {
41                actualResources.add(resource);
42            }
43            if (logger.isDebugEnabled()) {
44                logger.debug("Loaded " + loadCount + " bean definitions from
location [" + location + "]");
45            }
46            return loadCount;
47        }
48    }
49    //重载方法，调用loadBeanDefinitions(String);
50    public int loadBeanDefinitions(String... locations) throws
BeanDefinitionStoreException {
51        Assert.notNull(locations, "Location array must not be null");
52        int counter = 0;
53        for (String location : locations) {
54            counter += loadBeanDefinitions(location);
55        }
56        return counter;
    }
```

loadBeanDefinitions(Resource…resources)方法和上面分析的3个方法类似，同样也是调用XmlBeanDefinitionReader的loadBeanDefinitions方法。

从对AbstractBeanDefinitionReader的loadBeanDefinitions方法源码分析可以看出该方法做了以下两件事：

首先，调用资源加载器的获取资源方法resourceLoader.getResource(location)，获取到要加载的资源。

其次，真正执行加载功能是其子类XmlBeanDefinitionReader的loadBeanDefinitions方法。

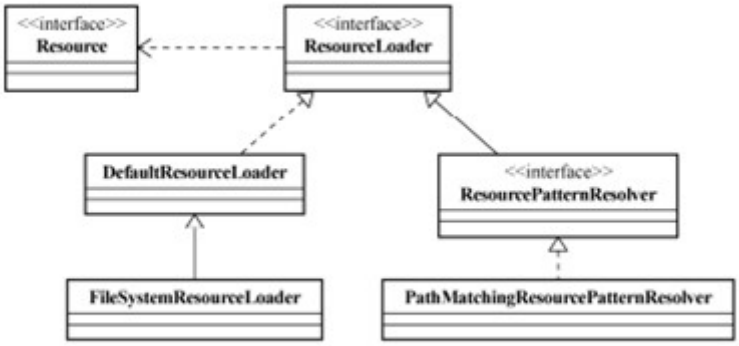
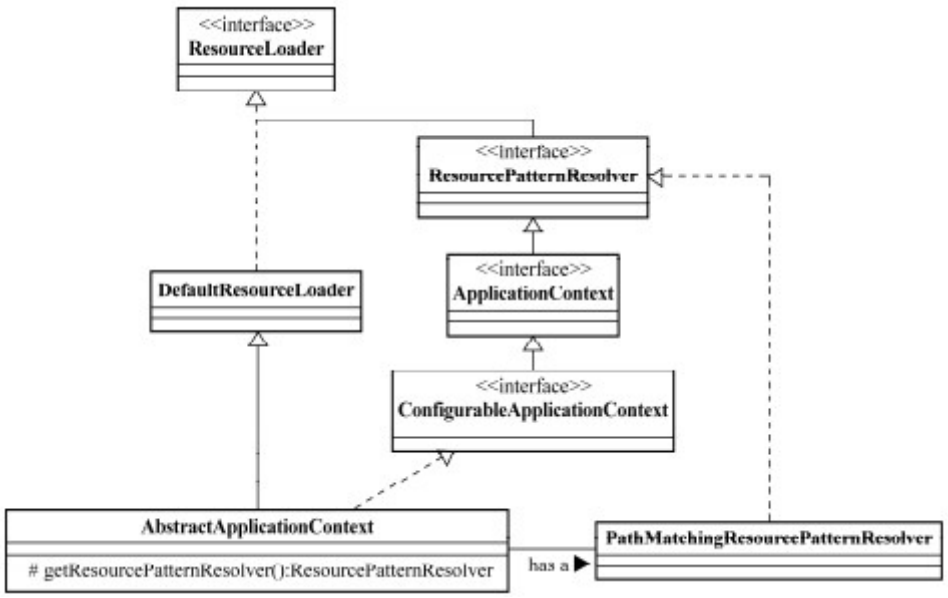


图5-1 Resource和ResourceLoader类层次图



看到第8、16行，结合上面的ResourceLoader与ApplicationContext的继承关系图，可以知道此时调用的是DefaultResourceLoader中的getSource()方法定位Resource，因为FileSystemXmlApplicationContext本身就是DefaultResourceLoader的实现类，所以此时又回到了FileSystemXmlApplicationContext中来。

7、资源加载器获取要读入的资源：

XmlBeanDefinitionReader通过调用其父类DefaultResourceLoader的getResource方法获取要加载的资源，其源码如下

```
1    //获取Resource的具体实现方法
2    public Resource getResource(String location) {
3        Assert.notNull(location, "Location must not be null");
4        //如果是类路径的方式，那需要使用ClassPathResource 来得到bean 文件的资源对象
5        if (location.startsWith(CLASSPATH_URL_PREFIX)) {
```

```
6            return new
ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()),
getClassLoader());
7        }
8         try {
9              // 如果是URL 方式，使用UrlResource 作为bean 文件的资源对象
10             URL url = new URL(location);
11             return new UrlResource(url);
12            }
13            catch (MalformedURLException ex) {
14            }
15            //如果既不是classpath标识，又不是URL标识的Resource定位，则调用
16            //容器本身的getResourceByPath方法获取Resource
17            return getResourceByPath(location);
18
19    }
```

FileSystemXmlApplicationContext容器提供了getResourceByPath方法的实现，就是为了处理既不是classpath标识，又不是URL标识的Resource定位这种情况。

```
protected Resource getResourceByPath(String path) {
   if (path != null && path.startsWith("/")) {
        path = path.substring(1);
    }
    //这里使用文件系统资源对象来定义bean 文件
    return new FileSystemResource(path);
}
```

这样代码就回到了 FileSystemXmlApplicationContext 中来，他提供了FileSystemResource 来完成从文件系统得到配置文件的资源定义。

这样，就可以从文件系统路径上对IOC 配置文件进行加载 - 当然我们可以按照这个逻辑从任何地方加载，在Spring 中我们看到它提供 的各种资源抽象，比如ClassPathResource, URLResource,FileSystemResource 等来供我们使用。上面我们看到的是定位Resource 的一个过程，而这只是加载过程的一部分.

8、XmlBeanDefinitionReader加载Bean定义资源：

Bean定义的Resource得到了

继续回到XmlBeanDefinitionReader的loadBeanDefinitions(Resource ...)方法看到代表bean文件的资源定义以后的载入过程。

```
1    //XmlBeanDefinitionReader加载资源的入口方法
2    public int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException {
3        //将读入的XML资源进行特殊编码处理
4        return loadBeanDefinitions(new EncodedResource(resource));
5    }
     //这里是载入XML形式Bean定义资源文件方法
6    public int loadBeanDefinitions(EncodedResource encodedResource) throws
BeanDefinitionStoreException {
7    .......
8    try {
9         //将资源文件转为InputStream的IO流
10        InputStream inputStream =
encodedResource.getResource().getInputStream();
11        try {
12           //从InputStream中得到XML的解析源
13            InputSource inputSource = new InputSource(inputStream);
14            if (encodedResource.getEncoding() != null) {
15                inputSource.setEncoding(encodedResource.getEncoding());
16            }
17            //这里是具体的读取过程
18            return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
19        }
20        finally {
21            //关闭从Resource中得到的IO流
22            inputStream.close();
23        }
24    }
25       .........
26}
27    //从特定XML文件中实际载入Bean定义资源的方法
28    protected int doLoadBeanDefinitions(InputSource inputSource, Resource
resource)
29        throws BeanDefinitionStoreException {
30    try {
31        int validationMode = getValidationModeForResource(resource);
32        //将XML文件转换为DOM对象，解析过程由documentLoader实现
33        Document doc = this.documentLoader.loadDocument(
34                inputSource, this.entityResolver, this.errorHandler,
validationMode, this.namespaceAware);
35        //这里是启动对Bean定义解析的详细过程，该解析过程会用到Spring的Bean配置规则
36        return registerBeanDefinitions(doc, resource);
37      }
38      .......
     }
```

通过源码分析，载入Bean定义资源文件的最后一步是将Bean定义资源转换为Document对象，该过程由documentLoader实现

9、DocumentLoader将Bean定义资源转换为Document对象：

DocumentLoader将Bean定义资源转换成Document对象的源码如下：

```
1    //使用标准的JAXP将载入的Bean定义资源转换成document对象
2    public Document loadDocument(InputSource inputSource, EntityResolver
entityResolver,
3            ErrorHandler errorHandler, int validationMode, boolean
namespaceAware) throws Exception {
4        //创建文件解析器工厂
5        DocumentBuilderFactory factory =
createDocumentBuilderFactory(validationMode, namespaceAware);
6        if (logger.isDebugEnabled()) {
7            logger.debug("Using JAXP provider [" + factory.getClass().getName()
+ "]");
8        }
9        //创建文档解析器
10        DocumentBuilder builder = createDocumentBuilder(factory,
entityResolver, errorHandler);
11        //解析Spring的Bean定义资源
12        return builder.parse(inputSource);
13    }
14    protected DocumentBuilderFactory createDocumentBuilderFactory(int
validationMode, boolean namespaceAware)
15            throws ParserConfigurationException {
16        //创建文档解析工厂
17        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

18        factory.setNamespaceAware(namespaceAware);
19        //设置解析XML的校验
20        if (validationMode != XmlValidationModeDetector.VALIDATION_NONE) {
21            factory.setValidating(true);
22            if (validationMode == XmlValidationModeDetector.VALIDATION_XSD) {

23                factory.setNamespaceAware(true);
24                try {
25                    factory.setAttribute(SCHEMA_LANGUAGE_ATTRIBUTE,
XSD_SCHEMA_LANGUAGE);
26                }
27                catch (IllegalArgumentException ex) {
28                    ParserConfigurationException pcex = new
ParserConfigurationException(
29                            "Unable to validate using XSD: Your JAXP provider
[" + factory +
30                            "] does not support XML Schema. Are you running on
Java 1.4 with Apache Crimson? " +
31                            "Upgrade to Apache Xerces (or Java 1.5) for full
XSD support.");
32                    pcex.initCause(ex);
33                    throw pcex;
34                }
35            }
36        }
37        return factory;
```

```
38     }
```

该解析过程调用JavaEE标准的JAXP标准进行处理。

至此Spring IoC容器根据定位的Bean定义资源文件，将其加载读入并转换成为Document对象过程完成。

接下来我们要继续分析Spring IoC容器将载入的Bean定义资源文件转换为Document对象之后，是如何将其解析为Spring IoC管理的Bean对象并将其注册到容器中的。

10、XmlBeanDefinitionReader解析载入的Bean定义资源文件：

XmlBeanDefinitionReader类中的doLoadBeanDefinitions方法是从特定XML文件中实际载入Bean定义资源的方法，该方法在载入Bean定义资源之后将其转换为Document对象，接下来调用registerBeanDefinitions启动Spring IoC容器对Bean定义的解析过程，registerBeanDefinitions方法源码如下：

```
1    //按照Spring的Bean语义要求将Bean定义资源解析并转换为容器内部数据结构
2    public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
3        //得到BeanDefinitionDocumentReader来对xml格式的BeanDefinition解析
4        BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
5        //获得容器中注册的Bean数量
6        int countBefore = getRegistry().getBeanDefinitionCount();
7        //解析过程入口，这里使用了委派模式，BeanDefinitionDocumentReader只是个接口，//具
体的解析实现过程有实现类DefaultBeanDefinitionDocumentReader完成
8        documentReader.registerBeanDefinitions(doc,
createReaderContext(resource));
9        //统计解析的Bean数量
10        return getRegistry().getBeanDefinitionCount() - countBefore;
11    }
12    //创建BeanDefinitionDocumentReader对象，解析Document对象
13    protected BeanDefinitionDocumentReader createBeanDefinitionDocumentReader()
{
14        return
BeanDefinitionDocumentReader.class.cast(BeanUtils.instantiateClass(this.document
ReaderClass));
      }
```

Bean定义资源的载入解析分为以下两个过程：

首先，通过调用XML解析器将Bean定义资源文件转换得到Document对象，但是这些Document对象并没有按照Spring的Bean规则进行解析。这一步是载入的过程

其次，在完成通用的XML解析之后，按照Spring的Bean规则对Document对象进行解析。

按照Spring的Bean规则对Document对象解析的过程是在接口BeanDefinitionDocumentReader的实现类DefaultBeanDefinitionDocumentReader中实现的。

11、DefaultBeanDefinitionDocumentReader对Bean定义的Document对象解析：

BeanDefinitionDocumentReader接口通过registerBeanDefinitions方法调用其实现类DefaultBeanDefinitionDocumentReader对Document对象进行解析，解析的代码如下：

```
1     //根据Spring DTD对Bean的定义规则解析Bean定义Document对象
2     public void registerBeanDefinitions(Document doc, XmlReaderContext
readerContext) {
3         //获得XML描述符
4         this.readerContext = readerContext;
5         logger.debug("Loading bean definitions");
6         //获得Document的根元素
7         Element root = doc.getDocumentElement();
8         //具体的解析过程由BeanDefinitionParserDelegate实现，
9         //BeanDefinitionParserDelegate中定义了Spring Bean定义XML文件的各种元素
10        BeanDefinitionParserDelegate delegate = createHelper(readerContext,
root);
11        //在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
12        preProcessXml(root);
13        //从Document的根元素开始进行Bean定义的Document对象
14        parseBeanDefinitions(root, delegate);
15        //在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性
16        postProcessXml(root);
17    }
18    //创建BeanDefinitionParserDelegate，用于完成真正的解析过程
19    protected BeanDefinitionParserDelegate createHelper(XmlReaderContext
readerContext, Element root) {
20        BeanDefinitionParserDelegate delegate = new
BeanDefinitionParserDelegate(readerContext);
21        //BeanDefinitionParserDelegate初始化Document根元素
22        delegate.initDefaults(root);
23        return delegate;
24    }
25    //使用Spring的Bean规则从Document的根元素开始进行Bean定义的Document对象
26    protected void parseBeanDefinitions(Element root,
BeanDefinitionParserDelegate delegate) {
27        //Bean定义的Document对象使用了Spring默认的XML命名空间
28        if (delegate.isDefaultNamespace(root)) {
29            //获取Bean定义的Document对象根元素的所有子节点
30            NodeList nl = root.getChildNodes();
31            for (int i = 0; i < nl.getLength(); i++) {
32                Node node = nl.item(i);
33                //获得Document节点是XML元素节点
34                if (node instanceof Element) {
35                    Element ele = (Element) node;
36                //Bean定义的Document的元素节点使用的是Spring默认的XML命名空间
37                    if (delegate.isDefaultNamespace(ele)) {
38                        //使用Spring的Bean规则解析元素节点
```

```
                        parseDefaultElement(ele, delegate);
                    }
                    else {
                        //没有使用Spring默认的XML命名空间，则使用用户自定义的解//析规则
解析元素节点
                        delegate.parseCustomElement(ele);
                    }
                }
            }
        }
        else {
            //Document的根节点没有使用Spring默认的命名空间，则使用用户自定义的
            //解析规则解析Document根节点
            delegate.parseCustomElement(root);
        }
    }
    //使用Spring的Bean规则解析Document元素节点
    private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate
delegate) {
        //如果元素节点是<Import>导入元素，进行导入解析
        if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
            importBeanDefinitionResource(ele);
        }
        //如果元素节点是<Alias>别名元素，进行别名解析
        else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
            processAliasRegistration(ele);
        }
        //元素节点既不是导入元素，也不是别名元素，即普通的<Bean>元素，
        //按照Spring的Bean规则解析元素
        else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
            processBeanDefinition(ele, delegate);
        }
    }
    //解析<Import>导入元素，从给定的导入路径加载Bean定义资源到Spring IoC容器中
    protected void importBeanDefinitionResource(Element ele) {
        //获取给定的导入元素的location属性
        String location = ele.getAttribute(RESOURCE_ATTRIBUTE);
        //如果导入元素的location属性值为空，则没有导入任何资源，直接返回
        if (!StringUtils.hasText(location)) {
            getReaderContext().error("Resource location must not be empty",
ele);
            return;
        }
        //使用系统变量值解析location属性值
        location = SystemPropertyUtils.resolvePlaceholders(location);
        Set<Resource> actualResources = new LinkedHashSet<Resource>(4);
        //标识给定的导入元素的location是否是绝对路径
        boolean absoluteLocation = false;
        try {
            absoluteLocation = ResourcePatternUtils.isUrl(location) ||
ResourceUtils.toURI(location).isAbsolute();
        }
        catch (URISyntaxException ex) {
            //给定的导入元素的location不是绝对路径
        }
        //给定的导入元素的location是绝对路径
        if (absoluteLocation) {
            try {
```

```
93                //使用资源读入器加载给定路径的Bean定义资源
94                int importCount =
getReaderContext().getReader().loadBeanDefinitions(location, actualResources);
95                if (logger.isDebugEnabled()) {
96                    logger.debug("Imported " + importCount + " bean definitions
from URL location [" + location + "]");
97                }
98            }
99            catch (BeanDefinitionStoreException ex) {
100                getReaderContext().error(
101                        "Failed to import bean definitions from URL location
[" + location + "]", ele, ex);
102            }
103        }
104        else {
105            //给定的导入元素的location是相对路径
106            try {
107                int importCount;
108                //将给定导入元素的location封装为相对路径资源
109                Resource relativeResource =
getReaderContext().getResource().createRelative(location);
110                //封装的相对路径资源存在
111                if (relativeResource.exists()) {
112                    //使用资源读入器加载Bean定义资源
113                    importCount =
getReaderContext().getReader().loadBeanDefinitions(relativeResource);
114                    actualResources.add(relativeResource);
115                }
116                //封装的相对路径资源不存在
117                else {
118                    //获取Spring IoC容器资源读入器的基本路径
119                    String baseLocation =
getReaderContext().getResource().getURL().toString();
120                    //根据Spring IoC容器资源读入器的基本路径加载给定导入
121                    //路径的资源
122                    importCount =
getReaderContext().getReader().loadBeanDefinitions(
123                            StringUtils.applyRelativePath(baseLocation,
location), actualResources);
124                }
125                if (logger.isDebugEnabled()) {
126                    logger.debug("Imported " + importCount + " bean
definitions from relative location [" + location + "]");
127                }
128            }
129            catch (IOException ex) {
130                getReaderContext().error("Failed to resolve current resource
location", ele, ex);
131            }
132            catch (BeanDefinitionStoreException ex) {
133                getReaderContext().error("Failed to import bean definitions
from relative location [" + location + "]",
134                        ele, ex);
135            }
136        }
137        Resource[] actResArray = actualResources.toArray(new
Resource[actualResources.size()]);
138        //在解析完<Import>元素之后，发送容器导入其他资源处理完成事件
```

```
139         getReaderContext().fireImportProcessed(location, actResArray,
extractSource(ele));
140     }
141     //解析<Alias>别名元素，为Bean向Spring IoC容器注册别名
142     protected void processAliasRegistration(Element ele) {
143         //获取<Alias>别名元素中name的属性值
144         String name = ele.getAttribute(NAME_ATTRIBUTE);
145         //获取<Alias>别名元素中alias的属性值
146         String alias = ele.getAttribute(ALIAS_ATTRIBUTE);
147         boolean valid = true;
148         //<alias>别名元素的name属性值为空
149         if (!StringUtils.hasText(name)) {
150             getReaderContext().error("Name must not be empty", ele);
151             valid = false;
152         }
153         //<alias>别名元素的alias属性值为空
154         if (!StringUtils.hasText(alias)) {
155             getReaderContext().error("Alias must not be empty", ele);
156             valid = false;
157         }
158         if (valid) {
159             try {
160                 //向容器的资源读入器注册别名
161                 getReaderContext().getRegistry().registerAlias(name, alias);

162             }
163             catch (Exception ex) {
164                 getReaderContext().error("Failed to register alias '" + alias
+
165                         "' for bean with name '" + name + "'", ele, ex);
166             }
167             //在解析完<Alias>元素之后，发送容器别名处理完成事件
168             getReaderContext().fireAliasRegistered(name, alias,
extractSource(ele));
169         }
170     }
171     //解析Bean定义资源Document对象的普通元素
172     protected void processBeanDefinition(Element ele,
BeanDefinitionParserDelegate delegate) {
173         // BeanDefinitionHolder是对BeanDefinition的封装，即Bean定义的封装类
174         //对Document对象中<Bean>元素的解析由BeanDefinitionParserDelegate实现
BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
175         if (bdHolder != null) {
176             bdHolder = delegate.decorateBeanDefinitionIfRequired(ele,
bdHolder);
177             try {
178                //向Spring IoC容器注册解析得到的Bean定义，这是Bean定义向IoC容器注册的入
口
BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
179             }
180             catch (BeanDefinitionStoreException ex) {
181                 getReaderContext().error("Failed to register bean definition
with name '" +
182                         bdHolder.getBeanName() + "'", ele, ex);
183             }
184             //在完成向Spring IoC容器注册解析得到的Bean定义之后，发送注册事件
```

```
185             getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
186         }
187     }
```

通过上述Spring IoC容器对载入的Bean定义Document解析可以看出，我们使用Spring时，在Spring配置文件中可以使用元素来导入IoC容器所需要的其他资源，Spring IoC容器在解析时会首先将指定导入的资源加载进容器中。使用别名时，Spring IoC容器首先将别名元素所定义的别名注册到容器中。

对于既不是元素，又不是元素的元素，即Spring配置文件中普通的元素的解析由BeanDefinitionParserDelegate类的parseBeanDefinitionElement方法来实现。

12、BeanDefinitionParserDelegate解析Bean定义资源文件中的元素：

Bean定义资源文件中的和元素解析在DefaultBeanDefinitionDocumentReader中已经完成，对Bean定义资源文件中使用最多的元素交由BeanDefinitionParserDelegate来解析，其解析实现的源码如下：

```
1     //解析<Bean>元素的入口
2     public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
3         return parseBeanDefinitionElement(ele, null);
4     }
5     //解析Bean定义资源文件中的<Bean>元素，这个方法中主要处理<Bean>元素的id，name
6     //和别名属性
7     public BeanDefinitionHolder parseBeanDefinitionElement(Element ele,
BeanDefinition containingBean) {
8         //获取<Bean>元素中的id属性值
9         String id = ele.getAttribute(ID_ATTRIBUTE);
10         //获取<Bean>元素中的name属性值
11         String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
12         ////获取<Bean>元素中的alias属性值
13         List<String> aliases = new ArrayList<String>();
14         //将<Bean>元素中的所有name属性值存放到别名中
15         if (StringUtils.hasLength(nameAttr)) {
16             String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr,
BEAN_NAME_DELIMITERS);
17             aliases.addAll(Arrays.asList(nameArr));
18         }
19         String beanName = id;
20         //如果<Bean>元素中没有配置id属性时，将别名中的第一个值赋值给beanName
21         if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
22             beanName = aliases.remove(0);
23             if (logger.isDebugEnabled()) {
24                 logger.debug("No XML 'id' specified - using '" + beanName +
25                         "' as bean name and " + aliases + " as aliases");
26             }
27         }
28         //检查<Bean>元素所配置的id或者name的唯一性，containingBean标识<Bean>
29         //元素中是否包含子<Bean>元素
30         if (containingBean == null) {
```

```
31              //检查<Bean>元素所配置的id、name或者别名是否重复
32              checkNameUniqueness(beanName, aliases, ele);
33          }
34          //详细对<Bean>元素中配置的Bean定义进行解析的地方
35          AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele,
beanName, containingBean);
36          if (beanDefinition != null) {
37              if (!StringUtils.hasText(beanName)) {
38                  try {
39                      if (containingBean != null) {
40                          //如果<Bean>元素中没有配置id、别名或者name，且没有包含
子//<Bean>元素，为解析的Bean生成一个唯一beanName并注册
41                          beanName = BeanDefinitionReaderUtils.generateBeanName(

42                                  beanDefinition,
this.readerContext.getRegistry(), true);
43                      }
44                      else {
45                          //如果<Bean>元素中没有配置id、别名或者name，且包含了
子//<Bean>元素，为解析的Bean使用别名向IoC容器注册
46                          beanName =
this.readerContext.generateBeanName(beanDefinition);
47                          //为解析的Bean使用别名注册时，为了向后兼容
                  //Spring1.2/2.0，给别名添加类名后缀
48                          String beanClassName =
beanDefinition.getBeanClassName();
49                          if (beanClassName != null &&
50                                  beanName.startsWith(beanClassName) &&
beanName.length() > beanClassName.length() &&
51
!this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
52                              aliases.add(beanClassName);
53                          }
54                      }
55                      if (logger.isDebugEnabled()) {
56                          logger.debug("Neither XML 'id' nor 'name' specified - "
+
57                                  "using generated bean name [" + beanName +
"]");
58                      }
59                  }
60                  catch (Exception ex) {
61                      error(ex.getMessage(), ele);
62                      return null;
63                  }
64              }
65              String[] aliasesArray = StringUtils.toStringArray(aliases);
66              return new BeanDefinitionHolder(beanDefinition, beanName,
aliasesArray);
67          }
68          //当解析出错时，返回null
69          return null;
70      }
71      //详细对<Bean>元素中配置的Bean定义其他属性进行解析，由于上面的方法中已经对//Bean的id、
name和别名等属性进行了处理，该方法中主要处理除这三个以外的其他属性数据
72      public AbstractBeanDefinition parseBeanDefinitionElement(
73              Element ele, String beanName, BeanDefinition containingBean) {
74          //记录解析的<Bean>
```

```
75          this.parseState.push(new BeanEntry(beanName));
76          //这里只读取<Bean>元素中配置的class名字，然后载入到BeanDefinition中去
77          //只是记录配置的class名字，不做实例化，对象的实例化在依赖注入时完成
78          String className = null;
79          if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
80              className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
81          }
82          try {
83              String parent = null;
84              //如果<Bean>元素中配置了parent属性，则获取parent属性的值
85              if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
86                  parent = ele.getAttribute(PARENT_ATTRIBUTE);
87              }
88              //根据<Bean>元素配置的class名称和parent属性值创建BeanDefinition
89              //为载入Bean定义信息做准备
90              AbstractBeanDefinition bd = createBeanDefinition(className,
parent);
91              //对当前的<Bean>元素中配置的一些属性进行解析和设置，如配置的单态(singleton)
属性等
92              parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);

93              //为<Bean>元素解析的Bean设置description信息
bd.setDescription(DomUtils.getChildElementValueByTagName(ele,
DESCRIPTION_ELEMENT));
94              //对<Bean>元素的meta(元信息)属性解析
95              parseMetaElements(ele, bd);
96              //对<Bean>元素的lookup-method属性解析
97              parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
98              //对<Bean>元素的replaced-method属性解析
99              parseReplacedMethodSubElements(ele, bd.getMethodOverrides());
100             //解析<Bean>元素的构造方法设置
101             parseConstructorArgElements(ele, bd);
102             //解析<Bean>元素的<property>设置
103             parsePropertyElements(ele, bd);
104             //解析<Bean>元素的qualifier属性
105             parseQualifierElements(ele, bd);
106             //为当前解析的Bean设置所需的资源和依赖对象
107             bd.setResource(this.readerContext.getResource());
108             bd.setSource(extractSource(ele));
109             return bd;
110         }
111         catch (ClassNotFoundException ex) {
112             error("Bean class [" + className + "] not found", ele, ex);
113         }
114         catch (NoClassDefFoundError err) {
115             error("Class that bean class [" + className + "] depends on not
found", ele, err);
116         }
117         catch (Throwable ex) {
118             error("Unexpected failure during bean definition parsing", ele,
ex);
119         }
120         finally {
121             this.parseState.pop();
122         }
123         //解析<Bean>元素出错时，返回null
124         return null;
125     }
```

只要使用过Spring，对Spring配置文件比较熟悉的人，通过对上述源码的分析，就会明白我们在Spring配置文件中元素的中配置的属性就是通过该方法解析和设置到Bean中去的。

注意：在解析元素过程中没有创建和实例化Bean对象，只是创建了Bean对象的定义类BeanDefinition，将元素中的配置信息设置到BeanDefinition中作为记录，当依赖注入时才使用这些记录信息创建和实例化具体的Bean对象。

上面方法中一些对一些配置如元信息(meta)、qualifier等的解析，我们在Spring中配置时使用的也不多，我们在使用Spring的元素时，配置最多的是属性，因此我们下面继续分析源码，了解Bean的属性在解析时是如何设置的。

13、BeanDefinitionParserDelegate解析元素：

BeanDefinitionParserDelegate在解析调用parsePropertyElements方法解析元素中的属性子元素，解析源码如下：

```
1    //解析<Bean>元素中的<property>子元素
2    public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
3        //获取<Bean>元素中所有的子元素
4        NodeList nl = beanEle.getChildNodes();
5        for (int i = 0; i < nl.getLength(); i++) {
6            Node node = nl.item(i);
7            //如果子元素是<property>子元素，则调用解析<property>子元素方法解析
8            if (isCandidateElement(node) && nodeNameEquals(node,
PROPERTY_ELEMENT)) {
9                parsePropertyElement((Element) node, bd);
10            }
11        }
12    }
13    //解析<property>元素
14    public void parsePropertyElement(Element ele, BeanDefinition bd) {
15        //获取<property>元素的名字
16        String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
17        if (!StringUtils.hasLength(propertyName)) {
18            error("Tag 'property' must have a 'name' attribute", ele);
19            return;
```

```
20        }
21        this.parseState.push(new PropertyEntry(propertyName));
22        try {
23            //如果一个Bean中已经有同名的property存在，则不进行解析，直接返回。
24            //即如果在同一个Bean中配置同名的property，则只有第一个起作用
25            if (bd.getPropertyValues().contains(propertyName)) {
26                error("Multiple 'property' definitions for property '" +
propertyName + "'", ele);
27                return;
28            }
29            //解析获取property的值
30            Object val = parsePropertyValue(ele, bd, propertyName);
31            //根据property的名字和值创建property实例
32            PropertyValue pv = new PropertyValue(propertyName, val);
33            //解析<property>元素中的属性
34            parseMetaElements(ele, pv);
35            pv.setSource(extractSource(ele));
36            bd.getPropertyValues().addPropertyValue(pv);
37        }
38        finally {
39            this.parseState.pop();
40        }
41    }
42    //解析获取property值
43    public Object parsePropertyValue(Element ele, BeanDefinition bd, String
propertyName) {
44        String elementName = (propertyName != null) ?
45                        "<property> element for property '" + propertyName +
"'" :
46                        "<constructor-arg> element";
47        //获取<property>的所有子元素，只能是其中一种类型:ref,value,list等
48        NodeList nl = ele.getChildNodes();
49        Element subElement = null;
50        for (int i = 0; i < nl.getLength(); i++) {
51            Node node = nl.item(i);
52            //子元素不是description和meta属性
53            if (node instanceof Element && !nodeNameEquals(node,
DESCRIPTION_ELEMENT) &&
54                    !nodeNameEquals(node, META_ELEMENT)) {
55                if (subElement != null) {
56                    error(elementName + " must not contain more than one sub-
element", ele);
57                }
58                else {//当前<property>元素包含有子元素
59                    subElement = (Element) node;
60                }
61            }
62        }
63        //判断property的属性值是ref还是value，不允许既是ref又是value
64        boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
65        boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
66        if ((hasRefAttribute && hasValueAttribute) ||
67                ((hasRefAttribute || hasValueAttribute) && subElement != null))
{
68            error(elementName +
69                    " is only allowed to contain either 'ref' attribute OR
'value' attribute OR sub-element", ele);
70        }
```

```
71          //如果属性是ref，创建一个ref的数据对象RuntimeBeanReference，这个对象
72          //封装了ref信息
73          if (hasRefAttribute) {
74              String refName = ele.getAttribute(REF_ATTRIBUTE);
75              if (!StringUtils.hasText(refName)) {
76                  error(elementName + " contains empty 'ref' attribute", ele);
77              }
78              //一个指向运行时所依赖对象的引用
79              RuntimeBeanReference ref = new RuntimeBeanReference(refName);
80              //设置这个ref的数据对象是被当前的property对象所引用
81              ref.setSource(extractSource(ele));
82              return ref;
83          }
84           //如果属性是value，创建一个value的数据对象TypedStringValue，这个对象
85          //封装了value信息
86          else if (hasValueAttribute) {
87              //一个持有String类型值的对象
88              TypedStringValue valueHolder = new
TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE));
89              //设置这个value数据对象是被当前的property对象所引用
90              valueHolder.setSource(extractSource(ele));
91              return valueHolder;
92          }
93          //如果当前<property>元素还有子元素
94          else if (subElement != null) {
95              //解析<property>的子元素
96              return parsePropertySubElement(subElement, bd);
97          }
98          else {
99              //propery属性中既不是ref，也不是value属性，解析出错返回null
error(elementName + " must specify a ref or value", ele);
100             return null;
101         }
    }
```

通过对上述源码的分析，我们可以了解在Spring配置文件中，元素中元素的相关配置是如何处理的：

a. ref被封装为指向依赖对象一个引用。

b.value配置都会封装成一个字符串类型的对象。

c.ref和value都通过“解析的数据类型属性值.setSource(extractSource(ele));”方法将属性值/引用与所引用的属性关联起来。

在方法的最后对于元素的子元素通过parsePropertySubElement 方法解析，我们继续分析该方法的源码，了解其解析过程。

14、解析元素的子元素：

在BeanDefinitionParserDelegate类中的parsePropertySubElement方法对中的子元素解析，源码如下：

```
1    //解析<property>元素中ref,value或者集合等子元素
2    public Object parsePropertySubElement(Element ele, BeanDefinition bd, String
defaultValueType) {
3        //如果<property>没有使用Spring默认的命名空间，则使用用户自定义的规则解析//内嵌元素

4        if (!isDefaultNamespace(ele)) {
5            return parseNestedCustomElement(ele, bd);
6        }
7        //如果子元素是bean，则使用解析<Bean>元素的方法解析
8        else if (nodeNameEquals(ele, BEAN_ELEMENT)) {
9            BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);

10            if (nestedBd != null) {
11                nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);

12            }
13            return nestedBd;
14        }
15        //如果子元素是ref，ref中只能有以下3个属性：bean、local、parent
16        else if (nodeNameEquals(ele, REF_ELEMENT)) {
17            //获取<property>元素中的bean属性值，引用其他解析的Bean的名称
18            //可以不再同一个Spring配置文件中，具体请参考Spring对ref的配置规则
19            String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
20            boolean toParent = false;
21            if (!StringUtils.hasLength(refName)) {
22                //获取<property>元素中的local属性值，引用同一个Xml文件中配置
23                //的Bean的id，local和ref不同，local只能引用同一个配置文件中的Bean
24                refName = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
25                if (!StringUtils.hasLength(refName)) {
26                    //获取<property>元素中parent属性值，引用父级容器中的Bean
27                    refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
28                    toParent = true;
29                    if (!StringUtils.hasLength(refName)) {
30                        error("'bean', 'local' or 'parent' is required for
<ref> element", ele);
31                        return null;
32                    }
33                }
34            }
35            //没有配置ref的目标属性值
36            if (!StringUtils.hasText(refName)) {
37                error("<ref> element contains empty target attribute", ele);
38                return null;
39            }
40            //创建ref类型数据，指向被引用的对象
41            RuntimeBeanReference ref = new RuntimeBeanReference(refName,
toParent);
```

```
42              //设置引用类型值是被当前子元素所引用
43              ref.setSource(extractSource(ele));
44              return ref;
45          }
46          //如果子元素是<idref>，使用解析ref元素的方法解析
47          else if (nodeNameEquals(ele, IDREF_ELEMENT)) {
48              return parseIdRefElement(ele);
49          }
50          //如果子元素是<value>，使用解析value元素的方法解析
51          else if (nodeNameEquals(ele, VALUE_ELEMENT)) {
52              return parseValueElement(ele, defaultValueType);
53          }
54          //如果子元素是null，为<property>设置一个封装null值的字符串数据
55          else if (nodeNameEquals(ele, NULL_ELEMENT)) {
56              TypedStringValue nullHolder = new TypedStringValue(null);
57              nullHolder.setSource(extractSource(ele));
58              return nullHolder;
59          }
60          //如果子元素是<array>，使用解析array集合子元素的方法解析
61          else if (nodeNameEquals(ele, ARRAY_ELEMENT)) {
62              return parseArrayElement(ele, bd);
63          }
64          //如果子元素是<list>，使用解析list集合子元素的方法解析
65          else if (nodeNameEquals(ele, LIST_ELEMENT)) {
66              return parseListElement(ele, bd);
67          }
68          //如果子元素是<set>，使用解析set集合子元素的方法解析
69          else if (nodeNameEquals(ele, SET_ELEMENT)) {
70              return parseSetElement(ele, bd);
71          }
72          //如果子元素是<map>，使用解析map集合子元素的方法解析
73          else if (nodeNameEquals(ele, MAP_ELEMENT)) {
74              return parseMapElement(ele, bd);
75          }
76          //如果子元素是<props>，使用解析props集合子元素的方法解析
77          else if (nodeNameEquals(ele, PROPS_ELEMENT)) {
78              return parsePropsElement(ele);
79          }
80          //既不是ref，又不是value，也不是集合，则子元素配置错误，返回null
81          else {
82              error("Unknown property sub-element: [" + ele.getNodeName() + "]",
ele);
83              return null;
84          }
    }
```

通过上述源码分析，我们明白了在Spring配置文件中，对元素中配置的Array、List、Set、Map、Prop等各种集合子元素的都通过上述方法解析，生成对应的数据对象，比如ManagedList、ManagedArray、ManagedSet等，这些Managed类是Spring对象BeanDefiniton的数据封装，对集合数据类型的具体解析有各自的解析方法实现，解析方法的命名非常规范，一目了然，我们对集合元素的解析方法进行源码分析，了解其实现过程。

15、解析子元素：

在BeanDefinitionParserDelegate类中的parseListElement方法就是具体实现解析元素中的集合子元素，源码如下：

```
1    //解析<list>集合子元素
2    public List parseListElement(Element collectionEle, BeanDefinition bd) {
3        //获取<list>元素中的value-type属性，即获取集合元素的数据类型
4        String defaultElementType =
collectionEle.getAttribute(VALUE_TYPE_ATTRIBUTE);
5        //获取<list>集合元素中的所有子节点
6        NodeList nl = collectionEle.getChildNodes();
7        //Spring中将List封装为ManagedList
8        ManagedList<Object> target = new ManagedList<Object>(nl.getLength());
9        target.setSource(extractSource(collectionEle));
10        //设置集合目标数据类型
11        target.setElementTypeName(defaultElementType);
12        target.setMergeEnabled(parseMergeAttribute(collectionEle));
13        //具体的<list>元素解析
14        parseCollectionElements(nl, target, bd, defaultElementType);
15        return target;
16    }
17    //具体解析<list>集合元素，<array>、<list>和<set>都使用该方法解析
18    protected void parseCollectionElements(
19            NodeList elementNodes, Collection<Object> target, BeanDefinition
bd, String defaultElementType) {
20        //遍历集合所有节点
21        for (int i = 0; i < elementNodes.getLength(); i++) {
22            Node node = elementNodes.item(i);
23            //节点不是description节点
24            if (node instanceof Element && !nodeNameEquals(node,
DESCRIPTION_ELEMENT)) {
25                //将解析的元素加入集合中，递归调用下一个子元素
26                target.add(parsePropertySubElement((Element) node, bd,
defaultElementType));
27            }
28        }
    }
```

经过对Spring Bean定义资源文件转换的Document对象中的元素层层解析，Spring IoC现在已经将XML形式定义的Bean定义资源文件转换为Spring IoC所识别的数据结构——BeanDefinition，它是Bean定义资源文件中配置的POJO对象在Spring IoC容器中的映射，我们可以通过AbstractBeanDefinition为入口，荣IoC容器进行索引、查询和操作。

通过Spring IoC容器对Bean定义资源的解析后，IoC容器大致完成了管理Bean对象的准备工作，即初始化过程，但是最为重要的依赖注入还没有发生，现在在IoC容器中BeanDefinition存储的只是一些静态信息，接下来需要向容器注册Bean定义信息才能全部完成IoC容器的初始化过程

16、解析过后的BeanDefinition在IoC容器中的注册：

让我们继续跟踪程序的执行顺序，接下来会到我们第3步中分析DefaultBeanDefinitionDocumentReader对Bean定义转换的Document对象解析的流程中，在其parseDefaultElement方法中完成对Document对象的解析后得到封装BeanDefinition的BeanDefinitionHold对象，然后调用BeanDefinitionReaderUtils的registerBeanDefinition方法向IoC容器注册解析的Bean，BeanDefinitionReaderUtils的注册的源码如下：

```
//将解析的BeanDefinitionHold注册到容器中
public static void registerBeanDefinition(BeanDefinitionHolder definitionHolder,
BeanDefinitionRegistry registry)
    throws BeanDefinitionStoreException {
        //获取解析的BeanDefinition的名称
         String beanName = definitionHolder.getBeanName();
        //向IoC容器注册BeanDefinition
        registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());
        //如果解析的BeanDefinition有别名，向容器为其注册别名
         String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String aliase : aliases) {
                registry.registerAlias(beanName, aliase);
            }
        }
}
```

当调用BeanDefinitionReaderUtils向IoC容器注册解析的BeanDefinition时，真正完成注册功能的是DefaultListableBeanFactory。

17、DefaultListableBeanFactory向IoC容器注册解析后的BeanDefinition：

DefaultListableBeanFactory中使用一个HashMap的集合对象存放IoC容器中注册解析的BeanDefinition，向IoC容器注册的主要源码如下：



```
1    //存储注册的俄BeanDefinition
2    private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<String, BeanDefinition>();
3    //向IoC容器注册解析的BeanDefiniton
4    public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
5            throws BeanDefinitionStoreException {
6        Assert.hasText(beanName, "Bean name must not be empty");
7        Assert.notNull(beanDefinition, "BeanDefinition must not be null");
8        //校验解析的BeanDefiniton
9        if (beanDefinition instanceof AbstractBeanDefinition) {
10           try {
11               ((AbstractBeanDefinition) beanDefinition).validate();
12           }
13           catch (BeanDefinitionValidationException ex) {
14               throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,

15                       "Validation of bean definition failed", ex);
16           }
17       }
18       //注册的过程中需要线程同步，以保证数据的一致性
19       synchronized (this.beanDefinitionMap) {
20           Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
21           //检查是否有同名的BeanDefinition已经在IoC容器中注册，如果已经注册，
22           //并且不允许覆盖已注册的Bean，则抛出注册失败异常
23           if (oldBeanDefinition != null) {
24               if (!this.allowBeanDefinitionOverriding) {
25                   throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,

26                           "Cannot register bean definition [" +
beanDefinition + "] for bean '" + beanName +
27                           "': There is already [" + oldBeanDefinition + "]
bound.");
28               }
29               else {//如果允许覆盖，则同名的Bean，后注册的覆盖先注册的
30                   if (this.logger.isInfoEnabled()) {
31                       this.logger.info("Overriding bean definition for bean
'" + beanName +
32                               "': replacing [" + oldBeanDefinition + "] with
[" + beanDefinition + "]");
33                   }
34               }
35           }
36           //IoC容器中没有已经注册同名的Bean，按正常注册流程注册
37           else {
38               this.beanDefinitionNames.add(beanName);
39               this.frozenBeanDefinitionNames = null;
40           }
41           this.beanDefinitionMap.put(beanName, beanDefinition);
42           //重置所有已经注册过的BeanDefinition的缓存
43           resetBeanDefinition(beanName);
44       }
   }
```

至此，Bean定义资源文件中配置的Bean被解析过后，已经注册到IoC容器中，被容器管理起来，真正完成了IoC容器初始化所做的全部工作。现 在IoC容器中已经建立了整个Bean的配置信息，这些BeanDefinition信息已经可以使用，并且可以被检索，IoC容器的作用就是对这些注册的Bean定义信息进行处理和维护。这些的注册的Bean定义信息是IoC容器控制反转的基础，正是有了这些注册的数据，容器才可以进行依赖注入。

总结：

现在通过上面的代码，总结一下IOC容器初始化的基本步骤：

u 初始化的入口在容器实现中的 refresh()调用来完成

u 对 bean 定义载入 IOC 容器使用的方法是 loadBeanDefinition,其中的大致过程如下：通过 ResourceLoader 来完成资源文件位置的定位，DefaultResourceLoader 是默认的实现，同时上下文本身就给出了 ResourceLoader 的实现，可以从类路径，文件系统, URL 等方式来定为资源位置。如果是 XmlBeanFactory作为 IOC 容器，那么需要为它指定 bean 定义的资源，也就是说 bean 定义文件时通过抽象成 Resource 来被 IOC 容器处理的，容器通过 BeanDefinitionReader来完成定义信息的解析和 Bean 信息的注册,往往使用的是XmlBeanDefinitionReader 来解析 bean 的 xml 定义文件 - 实际的处理过程是委托给 BeanDefinitionParserDelegate 来完成的，从而得到 bean 的定义信息，这些信息在 Spring 中使用 BeanDefinition 对象来表示 - 这个名字可以让我们想到 loadBeanDefinition,RegisterBeanDefinition 这些相关的方法 - 他们都是为处理 BeanDefinitin 服务的， 容器解析得到 BeanDefinitionIoC 以后，需要把它在 IOC 容器中注册，这由 IOC 实现 BeanDefinitionRegistry 接口来实现。注册过程就是在 IOC 容器内部维护的一个HashMap 来保存得到的 BeanDefinition 的过程。这个 HashMap 是 IoC 容器持有 bean 信息的场所，以后对 bean 的操作都是围绕这个HashMap 来实现的.

u 然后我们就可以通过 BeanFactory 和 ApplicationContext 来享受到 Spring IOC 的服务了,在使用 IOC 容器的时候，我们注意到除了少量粘合代码，绝大多数以正确 IoC 风格编写的应用程序代码完全不用关心如何到达工厂，因为容器将把这些对象与容器管理的其他对象钩在一起。基本的策略是把工厂放到已知的地方，最好是放在对预期使用的上下文有意义的地方，以及代码将实际需要访问工厂的地方。Spring 本身提供了对声明式载入 web 应用程序用法的应用程序上下文,并将其存储在ServletContext 中的框架实现。具体可以参见以后的文章

在使用 Spring IOC 容器的时候我们还需要区别两个概念:

> Beanfactory 和 Factory bean，其中 BeanFactory 指的是 IOC 容器的编程抽象，比如 ApplicationContext， XmlBeanFactory 等，这些都是 IOC 容器的具体表现，需要使用什么样的容器由客户决定,但 Spring 为我们提供了丰富的选择。 FactoryBean 只是一个可以在 IOC而容器中被管理的一个 bean,是对各种处理过程和资源使用的抽象,Factory bean 在需要时产生另一个对象，而不返回 FactoryBean本身,我们可以把它看成是一个抽象工厂，对它的调用返回的是工厂生产的产品。所有的 Factory bean 都实现特殊的org.springframework.beans.factory.FactoryBean 接口，当使用容器中 factory bean 的时候，该容器不会返回 factory bean 本身,而是返回其生成的对象。Spring 包括了大部分的通用资源和服务访问抽象的 Factory bean 的实现，其中包括:对 JNDI 查询的处理，对代理对象的处理，对事务性代理的处理，对 RMI 代理的处理等，这些我们都可以看成是具体的工厂,看成是 SPRING 为我们建立好的工厂。也就是说 Spring 通过使用抽象工厂模式为我们准备了一系列工厂来生产一些特定的对象,免除我们手工重复的工作，我们要使用时只需要在 IOC 容器里配置好就能很方便的使用了

四、IOC容器的依赖注入

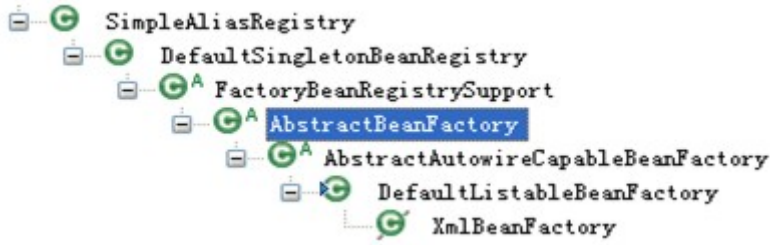
1、依赖注入发生的时间

当Spring IoC容器完成了Bean定义资源的定位、载入和解析注册以后，IoC容器中已经管理类Bean定义的相关数据，但是此时IoC容器还没有对所管理的Bean进行依赖注入，依赖注入在以下两种情况发生：

(1).用户第一次通过getBean方法向IoC容索要Bean时，IoC容器触发依赖注入。

(2).当用户在Bean定义资源中为元素配置了lazy-init属性，即让容器在解析注册Bean定义时进行预实例化，触发依赖注入。

BeanFactory接口定义了Spring IoC容器的基本功能规范，是Spring IoC容器所应遵守的最底层和最基本的编程规范。BeanFactory接口中定义了几个getBean方法，就是用户向IoC容器索取管理的Bean的方法，我们通过分析其子类的具体实现，理解Spring IoC容器在用户索取Bean时如何完成依赖注入。



在BeanFactory中我们看到getBean（String...）函数，它的具体实现在AbstractBeanFactory中

2、AbstractBeanFactory通过getBean向IoC容器获取被管理的Bean：

AbstractBeanFactory的getBean相关方法的源码如下：

```
1    //获取IoC容器中指定名称的Bean
2    public Object getBean(String name) throws BeansException {
3        //doGetBean才是真正向IoC容器获取被管理Bean的过程
4        return doGetBean(name, null, null, false);
5    }
6    //获取IoC容器中指定名称和类型的Bean
7    public <T> T getBean(String name, Class<T> requiredType) throws
BeansException {
```

```
8         //doGetBean才是真正向IoC容器获取被管理Bean的过程
9         return doGetBean(name, requiredType, null, false);
10    }
11    //获取IoC容器中指定名称和参数的Bean
12    public Object getBean(String name, Object... args) throws BeansException {

13        //doGetBean才是真正向IoC容器获取被管理Bean的过程
14        return doGetBean(name, null, args, false);
15    }
16    //获取IoC容器中指定名称、类型和参数的Bean
17    public <T> T getBean(String name, Class<T> requiredType, Object... args)
throws BeansException {
18    //doGetBean才是真正向IoC容器获取被管理Bean的过程
19        return doGetBean(name, requiredType, args, false);
20    }
21    //真正实现向IoC容器获取Bean的功能，也是触发依赖注入功能的地方
22    @SuppressWarnings("unchecked")
23    protected <T> T doGetBean(
24            final String name, final Class<T> requiredType, final Object[]
args, boolean typeCheckOnly)
25            throws BeansException {
26        //根据指定的名称获取被管理Bean的名称，剥离指定名称中对容器的相关依赖
27        //如果指定的是别名，将别名转换为规范的Bean名称
28        final String beanName = transformedBeanName(name);
29        Object bean;
30        //先从缓存中取是否已经有被创建过的单态类型的Bean，对于单态模式的Bean整
31        //个IoC容器中只创建一次，不需要重复创建
32        Object sharedInstance = getSingleton(beanName);
33        //IoC容器创建单态模式Bean实例对象
34        if (sharedInstance != null && args == null) {
35            if (logger.isDebugEnabled()) {
36                //如果指定名称的Bean在容器中已有单态模式的Bean被创建，直接返回
37                //已经创建的Bean
38                if (isSingletonCurrentlyInCreation(beanName)) {
39                    logger.debug("Returning eagerly cached instance of
singleton bean '" + beanName +
40                            "' that is not fully initialized yet - a
consequence of a circular reference");
41                }
42                else {
43                    logger.debug("Returning cached instance of singleton bean
'" + beanName + "'");
44                }
45            }
46            //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
47            //注意：BeanFactory是管理容器中Bean的工厂，而FactoryBean是
48            //创建创建对象的工厂Bean，两者之间有区别
49            bean = getObjectForBeanInstance(sharedInstance, name, beanName,
null);
50        }
51        else {//缓存没有正在创建的单态模式Bean
52            //缓存中已经有已经创建的原型模式Bean，但是由于循环引用的问题导致实
53            //例化对象失败
54            if (isPrototypeCurrentlyInCreation(beanName)) {
55                throw new BeanCurrentlyInCreationException(beanName);
56            }
57            //对IoC容器中是否存在指定名称的BeanDefinition进行检查，首先检查是否
58            //能在当前的BeanFactory中获取的所需要的Bean，如果不能则委托当前容器
```

```
59              //的父级容器去查找，如果还是找不到则沿着容器的继承体系向父级容器查找
60              BeanFactory parentBeanFactory = getParentBeanFactory();
61              //当前容器的父级容器存在，且当前容器中不存在指定名称的Bean
62              if (parentBeanFactory != null && !containsBeanDefinition(beanName))
{
63                  //解析指定Bean名称的原始名称
64                  String nameToLookup = originalBeanName(name);
65                  if (args != null) {
66                      //委派父级容器根据指定名称和显式的参数查找
67                      return (T) parentBeanFactory.getBean(nameToLookup, args);

68                  }
69                  else {
70                      //委派父级容器根据指定名称和类型查找
71                      return parentBeanFactory.getBean(nameToLookup,
requiredType);
72                  }
73              }
74              //创建的Bean是否需要进行类型验证，一般不需要
75              if (!typeCheckOnly) {
76                  //向容器标记指定的Bean已经被创建
77                  markBeanAsCreated(beanName);
78              }
79               //根据指定Bean名称获取其父级的Bean定义，主要解决Bean继承时子类
80              //合并父类公共属性问题
81              final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);
82              checkMergedBeanDefinition(mbd, beanName, args);
83              //获取当前Bean所有依赖Bean的名称
84              String[] dependsOn = mbd.getDependsOn();
85              //如果当前Bean有依赖Bean
86              if (dependsOn != null) {
87                  for (String dependsOnBean : dependsOn) {
88                      //递归调用getBean方法，获取当前Bean的依赖Bean
89                      getBean(dependsOnBean);
90                      //把被依赖Bean注册给当前依赖的Bean
91                      registerDependentBean(dependsOnBean, beanName);
92                  }
93              }
94              //创建单态模式Bean的实例对象
95              if (mbd.isSingleton()) {
96              //这里使用了一个匿名内部类，创建Bean实例对象，并且注册给所依赖的对象
97                  sharedInstance = getSingleton(beanName, new ObjectFactory() {

98                      public Object getObject() throws BeansException {
99                          try {
100                             //创建一个指定Bean实例对象，如果有父级继承，则合并子//类和
父类的定义
101                             return createBean(beanName, mbd, args);
102                          }
103                          catch (BeansException ex) {
104                             //显式地从容器单态模式Bean缓存中清除实例对象
105                             destroySingleton(beanName);
106                             throw ex;
107                          }
108                      }
109                  });
110                  //获取给定Bean的实例对象
```

```
111                 bean = getObjectForBeanInstance(sharedInstance, name,
beanName, mbd);
112             }
113             //IoC容器创建原型模式Bean实例对象
114             else if (mbd.isPrototype()) {
115                 //原型模式(Prototype)是每次都会创建一个新的对象
116                 Object prototypeInstance = null;
117                 try {
118                     //回调beforePrototypeCreation方法，默认的功能是注册当前创//建的
原型对象
119                     beforePrototypeCreation(beanName);
120                     //创建指定Bean对象实例
121                     prototypeInstance = createBean(beanName, mbd, args);
122                 }
123                 finally {
124                     //回调afterPrototypeCreation方法，默认的功能告诉IoC容器指//定
Bean的原型对象不再创建了
125                     afterPrototypeCreation(beanName);
126                 }
127                 //获取给定Bean的实例对象
128                 bean = getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
129             }
130             //要创建的Bean既不是单态模式，也不是原型模式，则根据Bean定义资源中
131             //配置的生命周期范围，选择实例化Bean的合适方法，这种在Web应用程序中
132             //比较常用，如：request、session、application等生命周期
133             else {
134                 String scopeName = mbd.getScope();
135                 final Scope scope = this.scopes.get(scopeName);
136                 //Bean定义资源中没有配置生命周期范围，则Bean定义不合法
137                 if (scope == null) {
138                     throw new IllegalStateException("No Scope registered for
scope '" + scopeName + "'");
139                 }
140                 try {
141                     //这里又使用了一个匿名内部类，获取一个指定生命周期范围的实例
142                     Object scopedInstance = scope.get(beanName, new
ObjectFactory() {
143                         public Object getObject() throws BeansException {
144                             beforePrototypeCreation(beanName);
145                             try {
146                                 return createBean(beanName, mbd, args);
147                             }
148                             finally {
149                                 afterPrototypeCreation(beanName);
150                             }
151                         }
152                     });
153                     //获取给定Bean的实例对象
154                     bean = getObjectForBeanInstance(scopedInstance, name,
beanName, mbd);
155                 }
156                 catch (IllegalStateException ex) {
157                     throw new BeanCreationException(beanName,
158                             "Scope '" + scopeName + "' is not active for the
current thread; " +
159                             "consider defining a scoped proxy for this bean if
you intend to refer to it from a singleton",
```

```
160                             ex);
161                 }
162             }
163         }
164         //对创建的Bean实例对象进行类型检查
165         if (requiredType != null && bean != null &&
!requiredType.isAssignableFrom(bean.getClass())) {
166             throw new BeanNotOfRequiredTypeException(name, requiredType,
bean.getClass());
167         }
168         return (T) bean;
169     }
```

通过上面对向IoC容器获取Bean方法的分析，我们可以看到在Spring中，如果Bean定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果Bean定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。

上面的源码只是定义了根据Bean定义的模式，采取的不同创建Bean实例对象的策略，具体的Bean实例对象的创建过程由实现了ObejctFactory接口的匿名内部类的createBean方法完成，ObejctFactory使用委派模式，具体的Bean实例创建过程交由其实现类AbstractAutowireCapableBeanFactory完成，我们继续分析AbstractAutowireCapableBeanFactory的createBean方法的源码，理解其创建Bean实例的具体实现过程。

3、AbstractAutowireCapableBeanFactory创建Bean实例对象：

AbstractAutowireCapableBeanFactory类实现了ObejctFactory接口，创建容器指定的Bean实例对象，同时还对创建的Bean实例对象进行初始化处理。其创建Bean实例对象的方法源码如下：

```
1     //创建Bean实例对象
2     protected Object createBean(final String beanName, final RootBeanDefinition
mbd, final Object[] args)
3             throws BeanCreationException {
4         if (logger.isDebugEnabled()) {
5             logger.debug("Creating instance of bean '" + beanName + "'");
6         }
7         //判断需要创建的Bean是否可以实例化，即是否可以通过当前的类加载器加载
8         resolveBeanClass(mbd, beanName);
9         //校验和准备Bean中的方法覆盖
10         try {
11             mbd.prepareMethodOverrides();
12         }
13         catch (BeanDefinitionValidationException ex) {
14             throw new
BeanDefinitionStoreException(mbd.getResourceDescription(),
15                     beanName, "Validation of method overrides failed", ex);
16         }
17         try {
18             //如果Bean配置了初始化前和初始化后的处理器，则试图返回一个需要创建//Bean的代理
对象
```

```
19            Object bean = resolveBeforeInstantiation(beanName, mbd);
20            if (bean != null) {
21                return bean;
22            }
23        }
24        catch (Throwable ex) {
25            throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
26                    "BeanPostProcessor before instantiation of bean failed",
ex);
27        }
28        //创建Bean的入口
29        Object beanInstance = doCreateBean(beanName, mbd, args);
30        if (logger.isDebugEnabled()) {
31            logger.debug("Finished creating instance of bean '" + beanName +
"'");
32        }
33        return beanInstance;
34    }
35    //真正创建Bean的方法
36    protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final Object[] args) {
37        //封装被创建的Bean对象
38        BeanWrapper instanceWrapper = null;
39        if (mbd.isSingleton()){//单态模式的Bean，先从容器中缓存中获取同名Bean
40            instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);

41        }
42        if (instanceWrapper == null) {
43            //创建实例对象
44            instanceWrapper = createBeanInstance(beanName, mbd, args);
45        }
46        final Object bean = (instanceWrapper != null ?
instanceWrapper.getWrappedInstance() : null);
47        //获取实例化对象的类型
48        Class beanType = (instanceWrapper != null ?
instanceWrapper.getWrappedClass() : null);
49        //调用PostProcessor后置处理器
50        synchronized (mbd.postProcessingLock) {
51            if (!mbd.postProcessed) {
52                applyMergedBeanDefinitionPostProcessors(mbd, beanType,
beanName);
53                mbd.postProcessed = true;
54            }
55        }
56        // Eagerly cache singletons to be able to resolve circular references

57        //向容器中缓存单态模式的Bean对象，以防循环引用
58        boolean earlySingletonExposure = (mbd.isSingleton() &&
this.allowCircularReferences &&
59                isSingletonCurrentlyInCreation(beanName));
60        if (earlySingletonExposure) {
61            if (logger.isDebugEnabled()) {
62                logger.debug("Eagerly caching bean '" + beanName +
63                        "' to allow for resolving potential circular
references");
64            }
65            //这里是一个匿名内部类，为了防止循环引用，尽早持有对象的引用
```

```
66            addSingletonFactory(beanName, new ObjectFactory() {
67                public Object getObject() throws BeansException {
68                    return getEarlyBeanReference(beanName, mbd, bean);
69                }
70            });
71        }
72        //Bean对象的初始化，依赖注入在此触发
73        //这个exposedObject在初始化完成之后返回作为依赖注入完成后的Bean
74        Object exposedObject = bean;
75        try {
76            //将Bean实例对象封装，并且Bean定义中配置的属性值赋值给实例对象
77            populateBean(beanName, mbd, instanceWrapper);
78            if (exposedObject != null) {
79                //初始化Bean对象
80                exposedObject = initializeBean(beanName, exposedObject, mbd);

81            }
82        }
83        catch (Throwable ex) {
84            if (ex instanceof BeanCreationException &&
beanName.equals(((BeanCreationException) ex).getBeanName())) {
85                throw (BeanCreationException) ex;
86            }
87            else {
88                throw new BeanCreationException(mbd.getResourceDescription(),
beanName, "Initialization of bean failed", ex);
89            }
90        }
91        if (earlySingletonExposure) {
92            //获取指定名称的已注册的单态模式Bean对象
93            Object earlySingletonReference = getSingleton(beanName, false);
94            if (earlySingletonReference != null) {
95                //根据名称获取的以注册的Bean和正在实例化的Bean是同一个
96                if (exposedObject == bean) {
97                    //当前实例化的Bean初始化完成
98                    exposedObject = earlySingletonReference;
99                }
100                //当前Bean依赖其他Bean，并且当发生循环引用时不允许新创建实例对象
101                else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
102                    String[] dependentBeans = getDependentBeans(beanName);
103                    Set<String> actualDependentBeans = new
LinkedHashSet<String>(dependentBeans.length);
104                    //获取当前Bean所依赖的其他Bean
105                    for (String dependentBean : dependentBeans) {
106                        //对依赖Bean进行类型检查
107                        if
(!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
108                            actualDependentBeans.add(dependentBean);
109                        }
110                    }
111                    if (!actualDependentBeans.isEmpty()) {
112                        throw new BeanCurrentlyInCreationException(beanName,

113                                "Bean with name '" + beanName + "' has been
injected into other beans [" +
114
StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
```

```
115                                 "] in its raw version as part of a circular
reference, but has eventually been " +
116                                 "wrapped. This means that said other beans do
not use the final version of the " +
117                                 "bean. This is often the result of over-eager
type matching - consider using " +
118                                 "'getBeanNamesOfType' with the
'allowEagerInit' flag turned off, for example.");
119                     }
120                 }
121             }
122         }
123         //注册完成依赖注入的Bean
124         try {
125             registerDisposableBeanIfNecessary(beanName, bean, mbd);
126         }
127         catch (BeanDefinitionValidationException ex) {
128             throw new BeanCreationException(mbd.getResourceDescription(),
beanName, "Invalid destruction signature", ex);
129         }
130         return exposedObject;
    }
```

通过对方法源码的分析，我们看到具体的依赖注入实现在以下两个方法中：

(1).createBeanInstance：生成Bean所包含的java对象实例。

(2).populateBean ： 对Bean属性的依赖注入进行处理。

下面继续分析这两个方法的代码实现。

4、createBeanInstance方法创建Bean的java实例对象：

在createBeanInstance方法中，根据指定的初始化策略，使用静态工厂、工厂方法或者容器的自动装配特性生成java实例对象，创建对象的源码如下：

```
1    //创建Bean的实例对象
2    protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition
mbd, Object[] args) {
3        //检查确认Bean是可实例化的
4        Class beanClass = resolveBeanClass(mbd, beanName);
5        //使用工厂方法对Bean进行实例化
6        if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) &&
!mbd.isNonPublicAccessAllowed()) {
7            throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
8                    "Bean class isn't public, and non-public access not allowed:
" + beanClass.getName());
```

```
9        }
10        if (mbd.getFactoryMethodName() != null)  {
11            //调用工厂方法实例化
12            return instantiateUsingFactoryMethod(beanName, mbd, args);
13        }
14        //使用容器的自动装配方法进行实例化
15        boolean resolved = false;
16        boolean autowireNecessary = false;
17        if (args == null) {
18            synchronized (mbd.constructorArgumentLock) {
19                if (mbd.resolvedConstructorOrFactoryMethod != null) {
20                    resolved = true;
21                    autowireNecessary = mbd.constructorArgumentsResolved;
22                }
23            }
24        }
25        if (resolved) {
26            if (autowireNecessary) {
27                //配置了自动装配属性，使用容器的自动装配实例化
28                //容器的自动装配是根据参数类型匹配Bean的构造方法
29                return autowireConstructor(beanName, mbd, null, null);
30            }
31            else {
32                //使用默认的无参构造方法实例化
33                return instantiateBean(beanName, mbd);
34            }
35        }
36        //使用Bean的构造方法进行实例化
37        Constructor[] ctors =
determineConstructorsFromBeanPostProcessors(beanClass, beanName);
38        if (ctors != null ||
39                mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
40                mbd.hasConstructorArgumentValues() ||
!ObjectUtils.isEmpty(args))  {
41            //使用容器的自动装配特性，调用匹配的构造方法实例化
42            return autowireConstructor(beanName, mbd, ctors, args);
43        }
44        //使用默认的无参构造方法实例化
45        return instantiateBean(beanName, mbd);
46    }
47    //使用默认的无参构造方法实例化Bean对象
48    protected BeanWrapper instantiateBean(final String beanName, final
RootBeanDefinition mbd) {
49        try {
50            Object beanInstance;
51            final BeanFactory parent = this;
52            //获取系统的安全管理接口，JDK标准的安全管理API
53            if (System.getSecurityManager() != null) {
54                //这里是一个匿名内置类，根据实例化策略创建实例对象
55                beanInstance = AccessController.doPrivileged(new
PrivilegedAction<Object>() {
56                    public Object run() {
57                        return getInstantiationStrategy().instantiate(mbd,
beanName, parent);
58                    }
59                }, getAccessControlContext());
60            }
```

```
61            else {
62                //将实例化的对象封装起来
63                beanInstance = getInstantiationStrategy().instantiate(mbd,
beanName, parent);
64            }
65            BeanWrapper bw = new BeanWrapperImpl(beanInstance);
66            initBeanWrapper(bw);
67            return bw;
68        }
69        catch (Throwable ex) {
70            throw new BeanCreationException(mbd.getResourceDescription(),
beanName, "Instantiation of bean failed", ex);
71        }
72    }
```

经过对上面的代码分析，我们可以看出，对使用工厂方法和自动装配特性的Bean的实例化相当比较清楚，调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略(JDK的反射机制或者CGLIB)来进行初始化了，在方法getInstantiationStrategy().instantiate中就具体实现类使用初始策略实例化对象。

5、SimpleInstantiationStrategy类使用默认的无参构造方法创建Bean实例化对象：

在使用默认的无参构造方法创建Bean的实例化对象时，方法getInstantiationStrategy().instantiate调用了SimpleInstantiationStrategy类中的实例化Bean的方法，其源码如下：

```
1    //使用初始化策略实例化Bean对象
2    public Object instantiate(RootBeanDefinition beanDefinition, String
beanName, BeanFactory owner) {
3        //如果Bean定义中没有方法覆盖，则就不需要CGLIB父类类的方法
4        if (beanDefinition.getMethodOverrides().isEmpty()) {
5            Constructor<?> constructorToUse;
6            synchronized (beanDefinition.constructorArgumentLock) {
7                //获取对象的构造方法或工厂方法
8                constructorToUse = (Constructor<?>)
beanDefinition.resolvedConstructorOrFactoryMethod;
9                //如果没有构造方法且没有工厂方法
10                if (constructorToUse == null) {
11                    //使用JDK的反射机制，判断要实例化的Bean是否是接口
12                    final Class clazz = beanDefinition.getBeanClass();
13                    if (clazz.isInterface()) {
14                        throw new BeanInstantiationException(clazz, "Specified
class is an interface");
15                    }
16                    try {
17                        if (System.getSecurityManager() != null) {
18                        //这里是一个匿名内置类，使用反射机制获取Bean的构造方法
```

```
                            constructorToUse =
AccessController.doPrivileged(new PrivilegedExceptionAction<Constructor>() {
                                public Constructor run() throws Exception {
                                    return
clazz.getDeclaredConstructor((Class[]) null);
                                }
                            });
                        }
                        else {
                            constructorToUse =
clazz.getDeclaredConstructor((Class[]) null);
                        }
                        beanDefinition.resolvedConstructorOrFactoryMethod =
constructorToUse;
                    }
                    catch (Exception ex) {
                        throw new BeanInstantiationException(clazz, "No default
constructor found", ex);
                    }
                }
            }
            //使用BeanUtils实例化，通过反射机制调用”构造方法.newInstance(arg)”来进行实
例化
            return BeanUtils.instantiateClass(constructorToUse);
        }
        else {
            //使用CGLIB来实例化对象
            return instantiateWithMethodInjection(beanDefinition, beanName,
owner);
        }
    }
```

通过上面的代码分析，我们看到了如果Bean有方法被覆盖了，则使用JDK的反射机制进行实例化，否则，使用CGLIB进行实例化。

instantiateWithMethodInjection方法调用SimpleInstantiationStrategy的子类CglibSubclassingInstantiationStrategy使用CGLIB来进行初始化，其源码如下：

```
    //使用CGLIB进行Bean对象实例化
    public Object instantiate(Constructor ctor, Object[] args) {
            //CGLIB中的类
            Enhancer enhancer = new Enhancer();
            //将Bean本身作为其基类
            enhancer.setSuperclass(this.beanDefinition.getBeanClass());
            enhancer.setCallbackFilter(new CallbackFilterImpl());
            enhancer.setCallbacks(new Callback[] {
                    NoOp.INSTANCE,
                     new LookupOverrideMethodInterceptor(),
                     new ReplaceOverrideMethodInterceptor()
```

```
12            });
13            //使用CGLIB的create方法生成实例对象
14            return (ctor == null) ?
15                    enhancer.create() :
16                    enhancer.create(ctor.getParameterTypes(), args);
17        }
```

CGLIB是一个常用的字节码生成器的类库，它提供了一系列API实现java字节码的生成和转换功能。我们在学习JDK的动态代理时都知道，JDK的动态代理只能针对接口，如果一个类没有实现任何接口，要对其进行动态代理只能使用CGLIB。

6、populateBean方法对Bean属性的依赖注入：

在第3步的分析中我们已经了解到Bean的依赖注入分为以下两个过程：

(1).createBeanInstance：生成Bean所包含的java对象实例。

(2).populateBean ： 对Bean属性的依赖注入进行处理。

第4、5步中我们已经分析了容器初始化生成Bean所包含的Java实例对象的过程，现在我们继续分析生成对象后，Spring IoC容器是如何将Bean的属性依赖关系注入Bean实例对象中并设置好的，属性依赖注入的代码如下：

```
1    //将Bean属性设置到生成的实例对象上
2    protected void populateBean(String beanName, AbstractBeanDefinition mbd,
BeanWrapper bw) {
3        //获取容器在解析Bean定义资源时为BeanDefiniton中设置的属性值
4        PropertyValues pvs = mbd.getPropertyValues();
5        //实例对象为null
6        if (bw == null) {
7            //属性值不为空
8            if (!pvs.isEmpty()) {
9                throw new BeanCreationException(
10                        mbd.getResourceDescription(), beanName, "Cannot apply
property values to null instance");
11            }
12            else {
13                //实例对象为null，属性值也为空，不需要设置属性值，直接返回
14                return;
15            }
16        }
17        //在设置属性之前调用Bean的PostProcessor后置处理器
18        boolean continueWithPropertyPopulation = true;
19        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

20            for (BeanPostProcessor bp : getBeanPostProcessors()) {
21                if (bp instanceof InstantiationAwareBeanPostProcessor) {
```

```
22                    InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
23                    if
(!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
24                        continueWithPropertyPopulation = false;
25                        break;
26                    }
27                }
28            }
29        }
30        if (!continueWithPropertyPopulation) {
31            return;
32        }
33        //依赖注入开始，首先处理autowire自动装配的注入
34        if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME ||
35                mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
36            MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
37            //对autowire自动装配的处理，根据Bean名称自动装配注入
38            if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME) {
39                autowireByName(beanName, mbd, bw, newPvs);
40            }
41            //根据Bean类型自动装配注入
42            if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
43                autowireByType(beanName, mbd, bw, newPvs);
44            }
45            pvs = newPvs;
46        }
47        //检查容器是否持有用于处理单态模式Bean关闭时的后置处理器
48        boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

49        //Bean实例对象没有依赖，即没有继承基类
50        boolean needsDepCheck = (mbd.getDependencyCheck() !=
RootBeanDefinition.DEPENDENCY_CHECK_NONE);
51        if (hasInstAwareBpps || needsDepCheck) {
52            //从实例对象中提取属性描述符
53            PropertyDescriptor[] filteredPds =
filterPropertyDescriptorsForDependencyCheck(bw);
54            if (hasInstAwareBpps) {
55                for (BeanPostProcessor bp : getBeanPostProcessors()) {
56                    if (bp instanceof InstantiationAwareBeanPostProcessor) {
57                        InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
58                        //使用BeanPostProcessor处理器处理属性值
59                        pvs = ibp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(), beanName);
60                        if (pvs == null) {
61                            return;
62                        }
63                    }
64                }
65            }
66            if (needsDepCheck) {
67                //为要设置的属性进行依赖检查
68                checkDependencies(beanName, mbd, filteredPds, pvs);
```

```
69            }
70        }
71        //对属性进行注入
72        applyPropertyValues(beanName, mbd, bw, pvs);
73    }
74    //解析并注入依赖属性的过程
75    protected void applyPropertyValues(String beanName, BeanDefinition mbd,
BeanWrapper bw, PropertyValues pvs) {
76        if (pvs == null || pvs.isEmpty()) {
77            return;
78        }
79        //封装属性值
80        MutablePropertyValues mpvs = null;
81        List<PropertyValue> original;
82        if (System.getSecurityManager()!= null) {
83            if (bw instanceof BeanWrapperImpl) {
84                //设置安全上下文，JDK安全机制
85                ((BeanWrapperImpl)
bw).setSecurityContext(getAccessControlContext());
86            }
87        }
88        if (pvs instanceof MutablePropertyValues) {
89            mpvs = (MutablePropertyValues) pvs;
90            //属性值已经转换
91            if (mpvs.isConverted()) {
92                try {
93                    //为实例化对象设置属性值
94                    bw.setPropertyValues(mpvs);
95                    return;
96                }
97                catch (BeansException ex) {
98                    throw new BeanCreationException(
99                            mbd.getResourceDescription(), beanName, "Error
setting property values", ex);
100                }
101            }
102            //获取属性值对象的原始类型值
103            original = mpvs.getPropertyValueList();
104        }
105        else {
106            original = Arrays.asList(pvs.getPropertyValues());
107        }
108        //获取用户自定义的类型转换
109        TypeConverter converter = getCustomTypeConverter();
110        if (converter == null) {
111            converter = bw;
112        }
113        //创建一个Bean定义属性值解析器，将Bean定义中的属性值解析为Bean实例对象
114        //的实际值
115        BeanDefinitionValueResolver valueResolver = new
BeanDefinitionValueResolver(this, beanName, mbd, converter);
116        //为属性的解析值创建一个拷贝，将拷贝的数据注入到实例对象中
117        List<PropertyValue> deepCopy = new ArrayList<PropertyValue>
(original.size());
118        boolean resolveNecessary = false;
119        for (PropertyValue pv : original) {
120            //属性值不需要转换
121            if (pv.isConverted()) {
```

```
122                 deepCopy.add(pv);
123             }
124             //属性值需要转换
125             else {
126                 String propertyName = pv.getName();
127                 //原始的属性值，即转换之前的属性值
128                 Object originalValue = pv.getValue();
129                 //转换属性值，例如将引用转换为IoC容器中实例化对象引用
130                 Object resolvedValue =
valueResolver.resolveValueIfNecessary(pv, originalValue);
131                 //转换之后的属性值
132                 Object convertedValue = resolvedValue;
133                 //属性值是否可以转换
134                 boolean convertible = bw.isWritableProperty(propertyName) &&

135
!PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
136                 if (convertible) {
137                     //使用用户自定义的类型转换器转换属性值
138                     convertedValue = convertForProperty(resolvedValue,
propertyName, bw, converter);
139                 }
140                 //存储转换后的属性值，避免每次属性注入时的转换工作
141                 if (resolvedValue == originalValue) {
142                     if (convertible) {
143                         //设置属性转换之后的值
144                         pv.setConvertedValue(convertedValue);
145                     }
146                     deepCopy.add(pv);
147                 }
148                 //属性是可转换的，且属性原始值是字符串类型，且属性的原始类型值不是
149                 //动态生成的字符串，且属性的原始值不是集合或者数组类型
150                 else if (convertible && originalValue instanceof
TypedStringValue &&
151                         !((TypedStringValue) originalValue).isDynamic() &&
152                         !(convertedValue instanceof Collection ||
ObjectUtils.isArray(convertedValue))) {
153                     pv.setConvertedValue(convertedValue);
154                     deepCopy.add(pv);
155                 }
156                 else {
157                     resolveNecessary = true;
158                     //重新封装属性的值
159                     deepCopy.add(new PropertyValue(pv, convertedValue));
160                 }
161             }
162         }
163         if (mpvs != null && !resolveNecessary) {
164             //标记属性值已经转换过
165             mpvs.setConverted();
166         }
167         //进行属性依赖注入
168         try {
169             bw.setPropertyValues(new MutablePropertyValues(deepCopy));
170         }
171         catch (BeansException ex) {
172             throw new BeanCreationException(
```

```
173                     mbd.getResourceDescription(), beanName, "Error setting
property values", ex);
174         }
    }
```

分析上述代码，我们可以看出，对属性的注入过程分以下两种情况：

(1).属性值类型不需要转换时，不需要解析属性值，直接准备进行依赖注入。

(2).属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

对属性值的解析是在BeanDefinitionValueResolver类中的resolveValueIfNecessary方法中进行的，对属性值的依赖注入是通过bw.setPropertyValues方法实现的，在分析属性值的依赖注入之前，我们先分析一下对属性值的解析过程。

7、BeanDefinitionValueResolver解析属性值：

当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个Bean实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由resolveValueIfNecessary方法实现，其源码如下：

```
1    //解析属性值，对注入类型进行转换
2    public Object resolveValueIfNecessary(Object argName, Object value) {
3        //对引用类型的属性进行解析
4        if (value instanceof RuntimeBeanReference) {
5            RuntimeBeanReference ref = (RuntimeBeanReference) value;
6            //调用引用类型属性的解析方法
7            return resolveReference(argName, ref);
8        }
9        //对属性值是引用容器中另一个Bean名称的解析
10        else if (value instanceof RuntimeBeanNameReference) {
11            String refName = ((RuntimeBeanNameReference) value).getBeanName();

12            refName = String.valueOf(evaluate(refName));
13            //从容器中获取指定名称的Bean
14            if (!this.beanFactory.containsBean(refName)) {
15                throw new BeanDefinitionStoreException(
16                        "Invalid bean name '" + refName + "' in bean reference
for " + argName);
17            }
18            return refName;
19        }
20        //对Bean类型属性的解析，主要是Bean中的内部类
21        else if (value instanceof BeanDefinitionHolder) {
22            BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
23            return resolveInnerBean(argName, bdHolder.getBeanName(),
bdHolder.getBeanDefinition());
```

```
24          }
25          else if (value instanceof BeanDefinition) {
26              BeanDefinition bd = (BeanDefinition) value;
27              return resolveInnerBean(argName, "(inner bean)", bd);
28          }
29          //对集合数组类型的属性解析
30          else if (value instanceof ManagedArray) {
31              ManagedArray array = (ManagedArray) value;
32              //获取数组的类型
33              Class elementType = array.resolvedElementType;
34              if (elementType == null) {
35                  //获取数组元素的类型
36                  String elementTypeName = array.getElementTypeName();
37                  if (StringUtils.hasText(elementTypeName)) {
38                      try {
39                          //使用反射机制创建指定类型的对象
40                          elementType = ClassUtils.forName(elementTypeName,
this.beanFactory.getBeanClassLoader());
41                          array.resolvedElementType = elementType;
42                      }
43                      catch (Throwable ex) {
44                          throw new BeanCreationException(
45                                  this.beanDefinition.getResourceDescription(),
this.beanName,
46                                  "Error resolving array type for " + argName,
ex);
47                      }
48                  }
49                  //没有获取到数组的类型，也没有获取到数组元素的类型，则直接设置数
50                  //组的类型为Object
51                  else {
52                      elementType = Object.class;
53                  }
54              }
55              //创建指定类型的数组
56              return resolveManagedArray(argName, (List<?>) value, elementType);

57          }
58          //解析list类型的属性值
59          else if (value instanceof ManagedList) {
60              return resolveManagedList(argName, (List<?>) value);
61          }
62          //解析set类型的属性值
63          else if (value instanceof ManagedSet) {
64              return resolveManagedSet(argName, (Set<?>) value);
65          }
66          //解析map类型的属性值
67          else if (value instanceof ManagedMap) {
68              return resolveManagedMap(argName, (Map<?, ?>) value);
69          }
70          //解析props类型的属性值，props其实就是key和value均为字符串的map
71          else if (value instanceof ManagedProperties) {
72              Properties original = (Properties) value;
73              //创建一个拷贝，用于作为解析后的返回值
74              Properties copy = new Properties();
75              for (Map.Entry propEntry : original.entrySet()) {
76                  Object propKey = propEntry.getKey();
77                  Object propValue = propEntry.getValue();
```

```
78                if (propKey instanceof TypedStringValue) {
79                    propKey = evaluate((TypedStringValue) propKey);
80                }
81                if (propValue instanceof TypedStringValue) {
82                    propValue = evaluate((TypedStringValue) propValue);
83                }
84                copy.put(propKey, propValue);
85            }
86            return copy;
87        }
88        //解析字符串类型的属性值
89        else if (value instanceof TypedStringValue) {
90            TypedStringValue typedStringValue = (TypedStringValue) value;
91            Object valueObject = evaluate(typedStringValue);
92            try {
93                //获取属性的目标类型
94                Class<?> resolvedTargetType =
resolveTargetType(typedStringValue);
95                if (resolvedTargetType != null) {
96                    //对目标类型的属性进行解析，递归调用
97                    return this.typeConverter.convertIfNecessary(valueObject,
resolvedTargetType);
98                }
99                //没有获取到属性的目标对象，则按Object类型返回
100                else {
101                    return valueObject;
102                }
103            }
104            catch (Throwable ex) {
105                throw new BeanCreationException(
106                        this.beanDefinition.getResourceDescription(),
this.beanName,
107                        "Error converting typed String value for " + argName,
ex);
108            }
109        }
110        else {
111            return evaluate(value);
112        }
113    }
114    //解析引用类型的属性值
115    private Object resolveReference(Object argName, RuntimeBeanReference ref)
{
116        try {
117            //获取引用的Bean名称
118            String refName = ref.getBeanName();
119            refName = String.valueOf(evaluate(refName));
120            //如果引用的对象在父类容器中，则从父类容器中获取指定的引用对象
121            if (ref.isToParent()) {
122                if (this.beanFactory.getParentBeanFactory() == null) {
123                    throw new BeanCreationException(
124                            this.beanDefinition.getResourceDescription(),
this.beanName,
125                            "Can't resolve reference to bean '" + refName +
126                            "' in parent factory: no parent factory
available");
127                }
```

```
128                 return
this.beanFactory.getParentBeanFactory().getBean(refName);
129             }
130             //从当前的容器中获取指定的引用Bean对象，如果指定的Bean没有被实例化
131             //则会递归触发引用Bean的初始化和依赖注入
132             else {
133                 Object bean = this.beanFactory.getBean(refName);
134                 //将当前实例化对象的依赖引用对象
135                 this.beanFactory.registerDependentBean(refName,
this.beanName);
136                 return bean;
137             }
138         }
139         catch (BeansException ex) {
140             throw new BeanCreationException(
141                     this.beanDefinition.getResourceDescription(),
this.beanName,
142                     "Cannot resolve reference to bean '" + ref.getBeanName() +
"' while setting " + argName, ex);
143         }
144     }
145     //解析array类型的属性
146     private Object resolveManagedArray(Object argName, List<?> ml, Class
elementType) {
147         //创建一个指定类型的数组，用于存放和返回解析后的数组
148         Object resolved = Array.newInstance(elementType, ml.size());
149         for (int i = 0; i < ml.size(); i++) {
150         //递归解析array的每一个元素，并将解析后的值设置到resolved数组中，索引为i
151             Array.set(resolved, i,
152                 resolveValueIfNecessary(new KeyedArgName(argName, i),
ml.get(i)));
153         }
154         return resolved;
155     }
156     //解析list类型的属性
157     private List resolveManagedList(Object argName, List<?> ml) {
158         List<Object> resolved = new ArrayList<Object>(ml.size());
159         for (int i = 0; i < ml.size(); i++) {
160             //递归解析list的每一个元素
161             resolved.add(
162                 resolveValueIfNecessary(new KeyedArgName(argName, i),
ml.get(i)));
163         }
164         return resolved;
165     }
166     //解析set类型的属性
167     private Set resolveManagedSet(Object argName, Set<?> ms) {
168         Set<Object> resolved = new LinkedHashSet<Object>(ms.size());
169         int i = 0;
170         //递归解析set的每一个元素
171         for (Object m : ms) {
172             resolved.add(resolveValueIfNecessary(new KeyedArgName(argName, i),
m));
173             i++;
174         }
175         return resolved;
176     }
177     //解析map类型的属性
```

```
178    private Map resolveManagedMap(Object argName, Map<?, ?> mm) {
179        Map<Object, Object> resolved = new LinkedHashMap<Object, Object>
(mm.size());
180        //递归解析map中每一个元素的key和value
181        for (Map.Entry entry : mm.entrySet()) {
182            Object resolvedKey = resolveValueIfNecessary(argName,
entry.getKey());
183            Object resolvedValue = resolveValueIfNecessary(
184                    new KeyedArgName(argName, entry.getKey()),
entry.getValue());
185            resolved.put(resolvedKey, resolvedValue);
186        }
187        return resolved;
188    }
```

通过上面的代码分析，我们明白了Spring是如何将引用类型，内部类以及集合类型等属性进行解析的，属性值解析完成后就可以进行依赖注入了，依赖注入的过程就是Bean对象实例设置到它所依赖的Bean对象属性上去，在第7步中我们已经说过，依赖注入是通过bw.setPropertyValues方法实现的，该方法也使用了委托模式，在BeanWrapper接口中至少定义了方法声明，依赖注入的具体实现交由其实现类BeanWrapperImpl来完成，下面我们就分析依BeanWrapperImpl中赖注入相关的源码。

8、BeanWrapperImpl对Bean属性的依赖注入：

BeanWrapperImpl类主要是对容器中完成初始化的Bean实例对象进行属性的依赖注入，即把Bean对象设置到它所依赖的另一个Bean的属性中去，依赖注入的相关源码如下：

```
1    //实现属性依赖注入功能
2    private void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv)
throws BeansException {
3        //PropertyTokenHolder主要保存属性的名称、路径，以及集合的size等信息
4        String propertyName = tokens.canonicalName;
5        String actualName = tokens.actualName;
6        //keys是用来保存集合类型属性的size
7        if (tokens.keys != null) {
8            //将属性信息拷贝
9            PropertyTokenHolder getterTokens = new PropertyTokenHolder();
10            getterTokens.canonicalName = tokens.canonicalName;
11            getterTokens.actualName = tokens.actualName;
12            getterTokens.keys = new String[tokens.keys.length - 1];
```

```
13            System.arraycopy(tokens.keys, 0, getterTokens.keys, 0,
tokens.keys.length - 1);
14            Object propValue;
15            try {
16                //获取属性值，该方法内部使用JDK的内省( Introspector)机制，调用属性//的
getter(readerMethod)方法，获取属性的值
17                propValue = getPropertyValue(getterTokens);
18            }
19            catch (NotReadablePropertyException ex) {
20                throw new NotWritablePropertyException(getRootClass(),
this.nestedPath + propertyName,
21                        "Cannot access indexed value in property referenced " +

22                        "in indexed property path '" + propertyName + "'", ex);

23            }
24            //获取集合类型属性的长度
25            String key = tokens.keys[tokens.keys.length - 1];
26            if (propValue == null) {
27                throw new NullValueInNestedPathException(getRootClass(),
this.nestedPath + propertyName,
28                        "Cannot access indexed value in property referenced " +

29                        "in indexed property path '" + propertyName + "':
returned null");
30            }
31            //注入array类型的属性值
32            else if (propValue.getClass().isArray()) {
33                //获取属性的描述符
34                PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
35                //获取数组的类型
36                Class requiredType = propValue.getClass().getComponentType();

37                //获取数组的长度
38                int arrayIndex = Integer.parseInt(key);
39                Object oldValue = null;
40                try {
41                    //获取数组以前初始化的值
42                    if (isExtractOldValueForEditor()) {
43                        oldValue = Array.get(propValue, arrayIndex);
44                    }
45                    //将属性的值赋值给数组中的元素
46                    Object convertedValue = convertIfNecessary(propertyName,
oldValue, pv.getValue(), requiredType,
47                            new PropertyTypeDescriptor(pd, new
MethodParameter(pd.getReadMethod(), -1), requiredType));
48                    Array.set(propValue, arrayIndex, convertedValue);
49                }
50                catch (IndexOutOfBoundsException ex) {
51                    throw new InvalidPropertyException(getRootClass(),
this.nestedPath + propertyName,
52                            "Invalid array index in property path '" +
propertyName + "'", ex);
53                }
54            }
55            //注入list类型的属性值
56            else if (propValue instanceof List) {
```

```
57                PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
58                //获取list集合的类型
59                Class requiredType =
GenericCollectionTypeResolver.getCollectionReturnType(
60                        pd.getReadMethod(), tokens.keys.length);
61                List list = (List) propValue;
62                //获取list集合的size
63                int index = Integer.parseInt(key);
64                Object oldValue = null;
65                if (isExtractOldValueForEditor() && index < list.size()) {
66                    oldValue = list.get(index);
67                }
68                //获取list解析后的属性值
69                Object convertedValue = convertIfNecessary(propertyName,
oldValue, pv.getValue(), requiredType,
70                        new PropertyTypeDescriptor(pd, new
MethodParameter(pd.getReadMethod(), -1), requiredType));
71                if (index < list.size()) {
72                    //为list属性赋值
73                    list.set(index, convertedValue);
74                }
75                //如果list的长度大于属性值的长度，则多余的元素赋值为null
76                else if (index >= list.size()) {
77                    for (int i = list.size(); i < index; i++) {
78                        try {
79                            list.add(null);
80                        }
81                        catch (NullPointerException ex) {
82                            throw new InvalidPropertyException(getRootClass(),
this.nestedPath + propertyName,
83                                    "Cannot set element with index " + index +
" in List of size " +
84                                    list.size() + ", accessed using property
path '" + propertyName +
85                                    "': List does not support filling up gaps
with null elements");
86                        }
87                    }
88                    list.add(convertedValue);
89                }
90            }
91            //注入map类型的属性值
92            else if (propValue instanceof Map) {
93                PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
94                //获取map集合key的类型
95                Class mapKeyType =
GenericCollectionTypeResolver.getMapKeyReturnType(
96                        pd.getReadMethod(), tokens.keys.length);
97                //获取map集合value的类型
98                Class mapValueType =
GenericCollectionTypeResolver.getMapValueReturnType(
99                        pd.getReadMethod(), tokens.keys.length);
100                Map map = (Map) propValue;
101                //解析map类型属性key值
102                Object convertedMapKey = convertIfNecessary(null, null, key,
mapKeyType,
```

```
103                         new PropertyTypeDescriptor(pd, new
MethodParameter(pd.getReadMethod(), -1), mapKeyType));
104                 Object oldValue = null;
105                 if (isExtractOldValueForEditor()) {
106                     oldValue = map.get(convertedMapKey);
107                 }
108                 //解析map类型属性value值
109                 Object convertedMapValue = convertIfNecessary(
110                         propertyName, oldValue, pv.getValue(), mapValueType,

111                         new TypeDescriptor(new
MethodParameter(pd.getReadMethod(), -1, tokens.keys.length + 1)));
112                 //将解析后的key和value值赋值给map集合属性
113                 map.put(convertedMapKey, convertedMapValue);
114             }
115             else {
116                 throw new InvalidPropertyException(getRootClass(),
this.nestedPath + propertyName,
117                         "Property referenced in indexed property path '" +
propertyName +
118                         "' is neither an array nor a List nor a Map; returned
value was [" + pv.getValue() + "]");
119             }
120         }
121         //对非集合类型的属性注入
122         else {
123             PropertyDescriptor pd = pv.resolvedDescriptor;
124             if (pd == null ||
!pd.getWriteMethod().getDeclaringClass().isInstance(this.object)) {
125                 pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
126                 //无法获取到属性名或者属性没有提供setter(写方法)方法
127                 if (pd == null || pd.getWriteMethod() == null) {
128                     //如果属性值是可选的，即不是必须的，则忽略该属性值
129                     if (pv.isOptional()) {
130                         logger.debug("Ignoring optional value for property '"
+ actualName +
131                                 "' - property not found on bean class [" +
getRootClass().getName() + "]");
132                         return;
133                     }
134                     //如果属性值是必须的，则抛出无法给属性赋值，因为每天提供setter方法异
常
135                     else {
136                         PropertyMatches matches =
PropertyMatches.forProperty(propertyName, getRootClass());
137                         throw new NotWritablePropertyException(
138                                 getRootClass(), this.nestedPath +
propertyName,
139                                 matches.buildErrorMessage(),
matches.getPossibleMatches());
140                     }
141                 }
142                 pv.getOriginalPropertyValue().resolvedDescriptor = pd;
143             }
144             Object oldValue = null;
145             try {
146                 Object originalValue = pv.getValue();
```

```
147                 Object valueToApply = originalValue;
148                 if (!Boolean.FALSE.equals(pv.conversionNecessary)) {
149                     if (pv.isConverted()) {
150                         valueToApply = pv.getConvertedValue();
151                     }
152                     else {
153                         if (isExtractOldValueForEditor() && pd.getReadMethod()
!= null) {
154                             //获取属性的getter方法(读方法)，JDK内省机制
155                             final Method readMethod = pd.getReadMethod();
156                             //如果属性的getter方法不是public访问控制权限的，即访问控
制权限比较严格，
157                             //则使用JDK的反射机制强行访问非public的方法(暴力读取属性
值)
158                             if
(!Modifier.isPublic(readMethod.getDeclaringClass().getModifiers()) &&
159                                     !readMethod.isAccessible()) {
160                                 if (System.getSecurityManager()!= null) {
161                                     //匿名内部类，根据权限修改属性的读取控制限制
162                                     AccessController.doPrivileged(new
PrivilegedAction<Object>() {
163                                         public Object run() {
164                                             readMethod.setAccessible(true);
165                                             return null;
166                                         }
167                                     });
168                                 }
169                                 else {
170                                     readMethod.setAccessible(true);
171                                 }
172                             }
173                             try {
174                                 //属性没有提供getter方法时，调用潜在的读取属性值//的
方法，获取属性值
175                                 if (System.getSecurityManager() != null) {
176                                     oldValue =
AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
177                                         public Object run() throws Exception {

178                                             return readMethod.invoke(object);

179                                         }
180                                     }, acc);
181                                 }
182                                 else {
183                                     oldValue = readMethod.invoke(object);
184                                 }
185                             }
186                             catch (Exception ex) {
187                                 if (ex instanceof PrivilegedActionException) {

188                                     ex = ((PrivilegedActionException)
ex).getException();
189                                 }
190                                 if (logger.isDebugEnabled()) {
191                                     logger.debug("Could not read previous
value of property '" +
```

```
192                                                 this.nestedPath + propertyName +
"'", ex);
193                                 }
194                             }
195                         }
196                         //设置属性的注入值
197                         valueToApply = convertForProperty(propertyName,
oldValue, originalValue, pd);
198                     }
199                     pv.getOriginalPropertyValue().conversionNecessary =
(valueToApply != originalValue);
200                 }
201                 //根据JDK的内省机制，获取属性的setter(写方法)方法
202                 final Method writeMethod = (pd instanceof
GenericTypeAwarePropertyDescriptor ?
203                         ((GenericTypeAwarePropertyDescriptor)
pd).getWriteMethodForActualAccess() :
204                         pd.getWriteMethod());
205                 //如果属性的setter方法是非public，即访问控制权限比较严格，则使用JDK的反
射机制，
206                 //强行设置setter方法可访问(暴力为属性赋值)
207                 if
(!Modifier.isPublic(writeMethod.getDeclaringClass().getModifiers()) &&
!writeMethod.isAccessible()) {
208                     //如果使用了JDK的安全机制，则需要权限验证
209                     if (System.getSecurityManager()!= null) {
210                         AccessController.doPrivileged(new
PrivilegedAction<Object>() {
211                             public Object run() {
212                                 writeMethod.setAccessible(true);
213                                 return null;
214                             }
215                         });
216                     }
217                     else {
218                         writeMethod.setAccessible(true);
219                     }
220                 }
221                 final Object value = valueToApply;
222                 if (System.getSecurityManager() != null) {
223                     try {
224                         //将属性值设置到属性上去
225                         AccessController.doPrivileged(new
PrivilegedExceptionAction<Object>() {
226                             public Object run() throws Exception {
227                                 writeMethod.invoke(object, value);
228                                 return null;
229                             }
230                         }, acc);
231                     }
232                     catch (PrivilegedActionException ex) {
233                         throw ex.getException();
234                     }
235                 }
236                 else {
237                     writeMethod.invoke(this.object, value);
238                 }
239             }
```

```
240            catch (TypeMismatchException ex) {
241                throw ex;
242            }
243            catch (InvocationTargetException ex) {
244                PropertyChangeEvent propertyChangeEvent =
245                        new PropertyChangeEvent(this.rootObject,
this.nestedPath + propertyName, oldValue, pv.getValue());
246                if (ex.getTargetException() instanceof ClassCastException) {

247                    throw new TypeMismatchException(propertyChangeEvent,
pd.getPropertyType(), ex.getTargetException());
248                }
249                else {
250                    throw new MethodInvocationException(propertyChangeEvent,
ex.getTargetException());
251                }
252            }
253            catch (Exception ex) {
254                PropertyChangeEvent pce =
255                        new PropertyChangeEvent(this.rootObject,
this.nestedPath + propertyName, oldValue, pv.getValue());
256                throw new MethodInvocationException(pce, ex);
257            }
258        }
    }
```

通过对上面注入依赖代码的分析，我们已经明白了Spring IoC容器是如何将属性的值注入到Bean实例对象中去的：

(1).对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

(2).对于非集合类型的属性，大量使用了JDK的反射和内省机制，通过属性的getter方法(reader method)获取指定属性注入以前的值，同时调用属性的setter方法(writer method)为属性设置注入后的值。看到这里相信很多人都明白了Spring的setter注入原理。

至此Spring IoC容器对Bean定义资源文件的定位，载入、解析和依赖注入已经全部分析完毕，现在Spring IoC容器中管理了一系列靠依赖关系联系起来的Bean，程序不需要应用自己手动创建所需的对象，Spring IoC容器会在我们使用的时候自动为我们创建，并且为我们注入好相关的依赖，这就是Spring核心功能的控制反转和依赖注入的相关功能。

五、IoC容器的高级特性

1、介绍

通过前面4篇文章对Spring IoC容器的源码分析，我们已经基本上了解了Spring IoC容器对Bean定义资源的定位、读入和解析过程，同时也清楚了当用户通过getBean方法向IoC容器获取被管理的Bean时，IoC容器对Bean进行的初始化和依赖注入过程，这些是Spring IoC容器的基本功能特性。Spring IoC容器还有一些高级特性，如使用lazy-init属性对Bean预初始化、FactoryBean产生或者修饰Bean对象的生成、IoC容器初始化Bean过程中使用BeanPostProcessor后置处理器对Bean声明周期事件管理和IoC容器的autowiring自动装配功能等。

2、Spring IoC容器的lazy-init属性实现预实例化：

通过前面我们对IoC容器的实现和工作原理分析，我们知道IoC容器的初始化过程就是对Bean定义资源的定位、载入和注册，此时容器对Bean的依赖注入并没有发生，依赖注入主要是在应用程序第一次向容器索取Bean时，通过getBean方法的调用完成。

当Bean定义资源的元素中配置了lazy-init属性时，容器将会在初始化的时候对所配置的Bean进行预实例化，Bean的依赖注入在容器初始化的时候就已经完成。这样，当应用程序第一次向容器索取被管理的Bean时，就不用再初始化和对Bean进行依赖注入了，直接从容器中获取已经完成依赖注入的现成Bean，可以提高应用第一次向容器获取Bean的性能。

下面我们通过代码分析容器预实例化的实现过程：

(1).refresh()

先从IoC容器的初始会过程开始，通过前面文章分析，我们知道IoC容器读入已经定位的Bean定义资源是从refresh方法开始的，我们首先从AbstractApplicationContext类的refresh方法入手分析，源码如下：

```
1    //容器初始化的过程，读入Bean定义资源，并解析注册
2    public void refresh() throws BeansException, IllegalStateException {
3        synchronized (this.startupShutdownMonitor) {
4             //调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识
5             prepareRefresh();
6             //告诉子类启动refreshBeanFactory()方法，Bean定义资源文件的载入从
7             //子类的refreshBeanFactory()方法启动
8             ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();
9             //为BeanFactory配置容器特性，例如类加载器、事件处理器等
10            prepareBeanFactory(beanFactory);
11            try {
12                //为容器的某些子类指定特殊的BeanPost事件处理器
13                postProcessBeanFactory(beanFactory);
14                //调用所有注册的BeanFactoryPostProcessor的Bean
15                invokeBeanFactoryPostProcessors(beanFactory);
16                //为BeanFactory注册BeanPost事件处理器.
17                //BeanPostProcessor是Bean后置处理器，用于监听容器触发的事件
18                registerBeanPostProcessors(beanFactory);
19                //初始化信息源，和国际化相关.
20                initMessageSource();
21                //初始化容器事件传播器.
22                initApplicationEventMulticaster();
23                //调用子类的某些特殊Bean初始化方法
24                onRefresh();
25                //为事件传播器注册事件监听器.
26                registerListeners();
27                //这里是对容器lazy-init属性进行处理的入口方法
28                finishBeanFactoryInitialization(beanFactory);
29                //初始化容器的生命周期事件处理器，并发布容器的生命周期事件
30                finishRefresh();
31            }
32            catch (BeansException ex) {
33                //销毁以创建的单态Bean
34                destroyBeans();
35                //取消refresh操作，重置容器的同步标识.
36                cancelRefresh(ex);
37                throw ex;
```

```
            }
        }
    }
```

在refresh方法中ConfigurableListableBeanFactorybeanFactory = obtainFreshBeanFactory();启动了Bean定义资源的载入、注册过程，而finishBeanFactoryInitialization方法是对注册后的Bean定义中的预实例化(lazy-init=false，Spring默认就是预实例化，即为true)的Bean进行处理的地方。

(2).finishBeanFactoryInitialization处理预实例化Bean：

当Bean定义资源被载入IoC容器之后，容器将Bean定义资源解析为容器内部的数据结构BeanDefinition注册到容器中，AbstractApplicationContext类中的finishBeanFactoryInitialization方法对配置了预实例化属性的Bean进行预初始化过程，源码如下：

```
    //对配置了lazy-init属性的Bean进行预实例化处理
    protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
        //这是Spring3以后新加的代码，为容器指定一个转换服务(ConversionService)
        //在对某些Bean属性进行转换时使用
        if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
                beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class)) {
            beanFactory.setConversionService(
                    beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class));
        }
        //为了类型匹配，停止使用临时的类加载器
        beanFactory.setTempClassLoader(null);
        //缓存容器中所有注册的BeanDefinition元数据，以防被修改
        beanFactory.freezeConfiguration();
        //对配置了lazy-init属性的单态模式Bean进行预实例化处理
        beanFactory.preInstantiateSingletons();
    }
```

ConfigurableListableBeanFactory是一个接口，其preInstantiateSingletons方法由其子类DefaultListableBeanFactory提供。

(3)、DefaultListableBeanFactory对配置lazy-init属性单态Bean的预实例化：

```
1//对配置lazy-init属性单态Bean的预实例化
2public void preInstantiateSingletons() throws BeansException {
3        if (this.logger.isInfoEnabled()) {
4            this.logger.info("Pre-instantiating singletons in " + this);
5        }
6        //在对配置lazy-init属性单态Bean的预实例化过程中，必须多线程同步，以确保数据一致性
7        synchronized (this.beanDefinitionMap) {
8            for (String beanName : this.beanDefinitionNames) {
9                //获取指定名称的Bean定义
10                RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

11                //Bean不是抽象的，是单态模式的，且lazy-init属性配置为false
12                if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {

13                    //如果指定名称的bean是创建容器的Bean
14                    if (isFactoryBean(beanName)) {
15                    //FACTORY_BEAN_PREFIX=”&”，当Bean名称前面加”&”符号
16                   //时，获取的是产生容器对象本身，而不是容器产生的Bean.
17                   //调用getBean方法，触发容器对Bean实例化和依赖注入过程
18                        final FactoryBean factory = (FactoryBean)
getBean(FACTORY_BEAN_PREFIX + beanName);
19                        //标识是否需要预实例化
20                        boolean isEagerInit;
21                        if (System.getSecurityManager() != null && factory
instanceof SmartFactoryBean) {
22                            //一个匿名内部类
23                            isEagerInit = AccessController.doPrivileged(new
PrivilegedAction<Boolean>() {
24                                public Boolean run() {
25                                    return ((SmartFactoryBean)
factory).isEagerInit();
26                                }
27                            }, getAccessControlContext());
28                        }
29                        else {
30                            isEagerInit = factory instanceof SmartFactoryBean
&& ((SmartFactoryBean) factory).isEagerInit();
31                        }
32                        if (isEagerInit) {
33                           //调用getBean方法，触发容器对Bean实例化和依赖注入过程
34                            getBean(beanName);
35                        }
36                    }
37                    else {
38                         //调用getBean方法，触发容器对Bean实例化和依赖注入过程
39                        getBean(beanName);
40                    }
41                }
42            }
```

```
43         }
    }
```

通过对lazy-init处理源码的分析，我们可以看出，如果设置了lazy-init属性，则容器在完成Bean定义的注册之后，会通过getBean方法，触发对指定Bean的初始化和依赖注入过程，这样当应用第一次向容器索取所需的Bean时，容器不再需要对Bean进行初始化和依赖注入，直接从已经完成实例化和依赖注入的Bean中取一个线程的Bean，这样就提高了第一次获取Bean的性能。

3、FactoryBean的实现：

```
在Spring中，有两个很容易混淆的类：BeanFactory和FactoryBean。
```

BeanFactory：Bean工厂，是一个工厂(Factory)，我们Spring IoC容器的最顶层接口就是这个BeanFactory，它的作用是管理Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean：工厂Bean，是一个Bean，作用是产生其他bean实例。通常情况下，这种bean没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他bean实例。通常情况下，bean无须自己实现工厂模式，Spring容器担任工厂角色；但少数情况下，容器中的bean本身就是工厂，其作用是产生其它bean实例。

当用户使用容器本身时，可以使用转义字符”&”来得到FactoryBean本身，以区别通过FactoryBean产生的实例对象和FactoryBean对象本身。在BeanFactory中通过如下代码定义了该转义字符：

StringFACTORY_BEAN_PREFIX = "&";

如果myJndiObject是一个FactoryBean，则使用&myJndiObject得到的是myJndiObject对象，而不是myJndiObject产生出来的对象。

(1).FactoryBean的源码如下：

```
//工厂Bean，用于产生其他对象
public interface FactoryBean<T> {
   //获取容器管理的对象实例
    T getObject() throws Exception;
    //获取Bean工厂创建的对象的类型
    Class<?> getObjectType();
    //Bean工厂创建的对象是否是单态模式，如果是单态模式，则整个容器中只有一个实例
   //对象，每次请求都返回同一个实例对象
    boolean isSingleton();
}
```

(2). AbstractBeanFactory的getBean方法调用FactoryBean：

在前面我们分析Spring Ioc容器实例化Bean并进行依赖注入过程的源码时，提到在getBean方法触发容器实例化Bean的时候会调用AbstractBeanFactory的doGetBean方法来进行实例化的过程，源码如下：

```
1       //真正实现向IoC容器获取Bean的功能，也是触发依赖注入功能的地方
2       @SuppressWarnings("unchecked")
3       protected <T> T doGetBean(
4               final String name, final Class<T> requiredType, final Object[]
args, boolean typeCheckOnly)
5               throws BeansException {
6           //根据指定的名称获取被管理Bean的名称，剥离指定名称中对容器的相关依赖
7           //如果指定的是别名，将别名转换为规范的Bean名称
8           final String beanName = transformedBeanName(name);
9           Object bean;
10         //先从缓存中取是否已经有被创建过的单态类型的Bean，对于单态模式的Bean整
11             //个IoC容器中只创建一次，不需要重复创建
12          Object sharedInstance = getSingleton(beanName);
13          //IoC容器创建单态模式Bean实例对象
14          if (sharedInstance != null && args == null) {
15              if (logger.isDebugEnabled()) {
16              //如果指定名称的Bean在容器中已有单态模式的Bean被创建，直接返回
17                    //已经创建的Bean
18                  if (isSingletonCurrentlyInCreation(beanName)) {
19                      logger.debug("Returning eagerly cached instance of
singleton bean '" + beanName +
20                                  "' that is not fully initialized yet - a
consequence of a circular reference");
21                  }
22                  else {
23                      logger.debug("Returning cached instance of singleton bean
'" + beanName + "'");
24                  }
25              }
26              //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
27              bean = getObjectForBeanInstance(sharedInstance, name, beanName,
null);
28          }
29          ……
30      }
31     //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
32     protected Object getObjectForBeanInstance(
33             Object beanInstance, String name, String beanName,
RootBeanDefinition mbd) {
34         //容器已经得到了Bean实例对象，这个实例对象可能是一个普通的Bean，也可能是
35         //一个工厂Bean，如果是一个工厂Bean，则使用它创建一个Bean实例对象，如果
36         //调用本身就想获得一个容器的引用，则指定返回这个工厂Bean实例对象
37         //如果指定的名称是容器的解引用(dereference，即是对象本身而非内存地址)，
38         //且Bean实例也不是创建Bean实例对象的工厂Bean
39         if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance
instanceof FactoryBean)) {
40             throw new BeanIsNotAFactoryException(transformedBeanName(name),
beanInstance.getClass());
41         }
42         //如果Bean实例不是工厂Bean，或者指定名称是容器的解引用，调用者向获取对
43         //容器的引用，则直接返回当前的Bean实例
44         if (!(beanInstance instanceof FactoryBean) ||
BeanFactoryUtils.isFactoryDereference(name)) {
45             return beanInstance;
46         }
```

```
47    //处理指定名称不是容器的解引用，或者根据名称获取的Bean实例对象是一个工厂Bean
48    //使用工厂Bean创建一个Bean的实例对象
49        Object object = null;
50        if (mbd == null) {
51            //从Bean工厂缓存中获取给定名称的Bean实例对象
52            object = getCachedObjectForFactoryBean(beanName);
53        }
54        //让Bean工厂生产给定名称的Bean对象实例
55        if (object == null) {
56            FactoryBean factory = (FactoryBean) beanInstance;
57            //如果从Bean工厂生产的Bean是单态模式的，则缓存
58            if (mbd == null && containsBeanDefinition(beanName)) {
59                //从容器中获取指定名称的Bean定义，如果继承基类，则合并基类相关属性
60                mbd = getMergedLocalBeanDefinition(beanName);
61            }
62            //如果从容器得到Bean定义信息，并且Bean定义信息不是虚构的，则让工厂
63            //Bean生产Bean实例对象
64            boolean synthetic = (mbd != null && mbd.isSynthetic());
65            //调用FactoryBeanRegistrySupport类的getObjectFromFactoryBean
66            //方法，实现工厂Bean生产Bean对象实例的过程
67            object = getObjectFromFactoryBean(factory, beanName, !synthetic);

68        }
69        return object;
    }
```

在上面获取给定Bean的实例对象的getObjectForBeanInstance方法中，会调用FactoryBeanRegistrySupport类的getObjectFromFactoryBean方法，该方法实现了Bean工厂生产Bean实例对象。

Dereference(解引用)：一个在C/C++中应用比较多的术语，在C++中，”*”是解引用符号，而”&”是引用符号，解引用是指变量指向的是所引用对象的本身数据，而不是引用对象的内存地址。

(3)、AbstractBeanFactory生产Bean实例对象：

AbstractBeanFactory类中生产Bean实例对象的主要源码如下：

```
71    //Bean工厂生产Bean实例对象
72    protected Object getObjectFromFactoryBean(FactoryBean factory, String
beanName, boolean shouldPostProcess) {
73        //Bean工厂是单态模式，并且Bean工厂缓存中存在指定名称的Bean实例对象
74        if (factory.isSingleton() && containsSingleton(beanName)) {
```

```
75            //多线程同步，以防止数据不一致
76            synchronized (getSingletonMutex()) {
77                //直接从Bean工厂缓存中获取指定名称的Bean实例对象
78                Object object = this.factoryBeanObjectCache.get(beanName);
79                //Bean工厂缓存中没有指定名称的实例对象，则生产该实例对象
80                if (object == null) {
81                    //调用Bean工厂的getObject方法生产指定Bean的实例对象
82                    object = doGetObjectFromFactoryBean(factory, beanName,
shouldPostProcess);
83                    //将生产的实例对象添加到Bean工厂缓存中
84                    this.factoryBeanObjectCache.put(beanName, (object != null ?
object : NULL_OBJECT));
85                }
86                return (object != NULL_OBJECT ? object : null);
87            }
88        }
89        //调用Bean工厂的getObject方法生产指定Bean的实例对象
90        else {
91            return doGetObjectFromFactoryBean(factory, beanName,
shouldPostProcess);
92        }
93    }
94    //调用Bean工厂的getObject方法生产指定Bean的实例对象
95    private Object doGetObjectFromFactoryBean(
96            final FactoryBean factory, final String beanName, final boolean
shouldPostProcess)
97            throws BeanCreationException {
98        Object object;
99        try {
100            if (System.getSecurityManager() != null) {
101                AccessControlContext acc = getAccessControlContext();
102                try {
103                    //实现PrivilegedExceptionAction接口的匿名内置类
104                    //根据JVM检查权限，然后决定BeanFactory创建实例对象
105                    object = AccessController.doPrivileged(new
PrivilegedExceptionAction<Object>() {
106                        public Object run() throws Exception {
107                                //调用BeanFactory接口实现类的创建对象方法
108                                return factory.getObject();
109                            }
110                        }, acc);
111                }
112                catch (PrivilegedActionException pae) {
113                    throw pae.getException();
114                }
115            }
116            else {
117                //调用BeanFactory接口实现类的创建对象方法
118                object = factory.getObject();
119            }
120        }
121        catch (FactoryBeanNotInitializedException ex) {
122            throw new BeanCurrentlyInCreationException(beanName,
ex.toString());
123        }
124        catch (Throwable ex) {
125            throw new BeanCreationException(beanName, "FactoryBean threw
exception on object creation", ex);
```

```
126         }
127         //创建出来的实例对象为null，或者因为单态对象正在创建而返回null
128         if (object == null && isSingletonCurrentlyInCreation(beanName)) {
129             throw new BeanCurrentlyInCreationException(
130                     beanName, "FactoryBean which is currently in creation
returned null from getObject");
131         }
132         //为创建出来的Bean实例对象添加BeanPostProcessor后置处理器
133         if (object != null && shouldPostProcess) {
134             try {
135                 object = postProcessObjectFromFactoryBean(object, beanName);

136             }
137             catch (Throwable ex) {
138                 throw new BeanCreationException(beanName, "Post-processing of
the FactoryBean's object failed", ex);
139             }
140         }
141         return object;
    }
```

从上面的源码分析中，我们可以看出，BeanFactory接口调用其实现类的getObject方法来实现创建Bean实例对象的功能。

(4).工厂Bean的实现类getObject方法创建Bean实例对象：

FactoryBean的实现类有非常多，比如：Proxy、RMI、JNDI、ServletContextFactoryBean等等，FactoryBean接口为Spring容器提供了一个很好的封装机制，具体的getObject有不同的实现类根据不同的实现策略来具体提供，我们分析一个最简单的AnnotationTestFactoryBean的实现源码：

```
143    public class AnnotationTestBeanFactory implements
FactoryBean<IJmxTestBean> {
144         private final FactoryCreatedAnnotationTestBean instance = new
FactoryCreatedAnnotationTestBean();
145         public AnnotationTestBeanFactory() {
146             this.instance.setName("FACTORY");
147         }
148         //AnnotationTestBeanFactory产生Bean实例对象的实现
149         public IJmxTestBean getObject() throws Exception {
150             return this.instance;
151         }
152         public Class<? extends IJmxTestBean> getObjectType() {
153             return FactoryCreatedAnnotationTestBean.class;
154         }
155         public boolean isSingleton() {
156             return true;
157         }
     }
```

其他的Proxy，RMI，JNDI等等，都是根据相应的策略提供getObject的实现。这里不做一一分析，这已经不是Spring的核心功能，有需要的时候再去深入研究。

4.BeanPostProcessor后置处理器的实现：

BeanPostProcessor后置处理器是Spring IoC容器经常使用到的一个特性，这个Bean后置处理器是一个监听器，可以监听容器触发的Bean声明周期事件。后置处理器向容器注册以后，容器中管理的Bean就具备了接收IoC容器事件回调的能力。

BeanPostProcessor的使用非常简单，只需要提供一个实现接口BeanPostProcessor的实现类，然后在Bean的配置文件中设置即可。

(1).BeanPostProcessor的源码如下：

```
1    package org.springframework.beans.factory.config;
2    import org.springframework.beans.BeansException;
3    public interface BeanPostProcessor {
4        //为在Bean的初始化前提供回调入口
5        Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException;
6        //为在Bean的初始化之后提供回调入口
7        Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException;
     }
```

这两个回调的入口都是和容器管理的Bean的生命周期事件紧密相关，可以为用户提供在Spring IoC容器初始化Bean过程中自定义的处理操作。

(2).AbstractAutowireCapableBeanFactory类对容器生成的Bean添加后置处理器：

BeanPostProcessor后置处理器的调用发生在Spring IoC容器完成对Bean实例对象的创建和属性的依赖注入完成之后，在对Spring依赖注入的源码分析过程中我们知道，当应用程序第一次调用getBean方法(lazy-init预实例化除外)向Spring IoC容器索取指定Bean时触发Spring IoC容器创建Bean实例对象并进行依赖注入的过程，其中真正实现创建Bean对象并进行依赖注入的方法是AbstractAutowireCapableBeanFactory类的doCreateBean方法，主要源码如下：

```
1    //真正创建Bean的方法
2    protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final Object[] args) {
3        //创建Bean实例对象
4        ……
5        try {
6            //对Bean属性进行依赖注入
7            populateBean(beanName, mbd, instanceWrapper);
8            if (exposedObject != null) {
9                //在对Bean实例对象生成和依赖注入完成以后，开始对Bean实例对象
10               //进行初始化 ，为Bean实例对象应用BeanPostProcessor后置处理器
11               exposedObject = initializeBean(beanName, exposedObject, mbd);
12            }
13       }
14       catch (Throwable ex) {
```

```
15            if (ex instanceof BeanCreationException &&
beanName.equals(((BeanCreationException) ex).getBeanName())) {
16                throw (BeanCreationException) ex;
17            }
18        ……
19        //为应用返回所需要的实例对象
20        return exposedObject;
    }
```

从上面的代码中我们知道，为Bean实例对象添加BeanPostProcessor后置处理器的入口的是initializeBean方法。

(3).initializeBean方法为容器产生的Bean实例对象添加BeanPostProcessor后置处理器：

同样在AbstractAutowireCapableBeanFactory类中，initializeBean方法实现为容器创建的Bean实例对象添加BeanPostProcessor后置处理器，源码如下：

```
1    //初始容器创建的Bean实例对象，为其添加BeanPostProcessor后置处理器
2    protected Object initializeBean(final String beanName, final Object bean,
RootBeanDefinition mbd) {
3        //JDK的安全机制验证权限
4        if (System.getSecurityManager() != null) {
5            //实现PrivilegedAction接口的匿名内部类
6            AccessController.doPrivileged(new PrivilegedAction<Object>() {
7                public Object run() {
8                    invokeAwareMethods(beanName, bean);
9                    return null;
10                }
11            }, getAccessControlContext());
12        }
13        else {
14            //为Bean实例对象包装相关属性，如名称，类加载器，所属容器等信息
15            invokeAwareMethods(beanName, bean);
16        }
17        Object wrappedBean = bean;
18        //对BeanPostProcessor后置处理器的postProcessBeforeInitialization
19        //回调方法的调用，为Bean实例初始化前做一些处理
20        if (mbd == null || !mbd.isSynthetic()) {
21            wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
22        }
23        //调用Bean实例对象初始化的方法，这个初始化方法是在Spring Bean定义配置
24        //文件中通过init-method属性指定的
25        try {
26            invokeInitMethods(beanName, wrappedBean, mbd);
27        }
28        catch (Throwable ex) {
```

```
29            throw new BeanCreationException(
30                    (mbd != null ? mbd.getResourceDescription() : null),
31                    beanName, "Invocation of init method failed", ex);
32        }
33        //对BeanPostProcessor后置处理器的postProcessAfterInitialization
34        //回调方法的调用，为Bean实例初始化之后做一些处理
35        if (mbd == null || !mbd.isSynthetic()) {
36            wrappedBean =
applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
37        }
38        return wrappedBean;
39    }
40    //调用BeanPostProcessor后置处理器实例对象初始化之前的处理方法
41    public Object applyBeanPostProcessorsBeforeInitialization(Object
existingBean, String beanName)
42            throws BeansException {
43        Object result = existingBean;
44        //遍历容器为所创建的Bean添加的所有BeanPostProcessor后置处理器
45        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
46            //调用Bean实例所有的后置处理中的初始化前处理方法，为Bean实例对象在
47            //初始化之前做一些自定义的处理操作
48            result = beanProcessor.postProcessBeforeInitialization(result,
beanName);
49            if (result == null) {
50                return result;
51            }
52        }
53        return result;
54    }
55    //调用BeanPostProcessor后置处理器实例对象初始化之后的处理方法
56    public Object applyBeanPostProcessorsAfterInitialization(Object
existingBean, String beanName)
57            throws BeansException {
58        Object result = existingBean;
59        //遍历容器为所创建的Bean添加的所有BeanPostProcessor后置处理器
60        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
61            //调用Bean实例所有的后置处理中的初始化后处理方法，为Bean实例对象在
62            //初始化之后做一些自定义的处理操作
63            result = beanProcessor.postProcessAfterInitialization(result,
beanName);
64            if (result == null) {
65                return result;
66            }
67        }
68        return result;
    }
```

BeanPostProcessor是一个接口，其初始化前的操作方法和初始化后的操作方法均委托其实现子类来实现，在Spring中，BeanPostProcessor的实现子类非常的多，分别完成不同的操作，如：AOP面向切面编程的注册通知适配器、Bean对象的数据校验、Bean继承属性/方法的合并等等，我们以最简单的AOP切面织入来简单了解其主要的功能。

(4).AdvisorAdapterRegistrationManager在Bean对象初始化后注册通知适配器：

AdvisorAdapterRegistrationManager是BeanPostProcessor的一个实现类，其主要的作用为容器中管理的Bean注册一个面向切面编程的通知适配器，以便在Spring容器为所管理的Bean进行面向切面编程时提供方便，其源码如下：

```
1 //为容器中管理的Bean注册一个面向切面编程的通知适配器
2 public class AdvisorAdapterRegistrationManager implements BeanPostProcessor {

3    //容器中负责管理切面通知适配器注册的对象
4    private AdvisorAdapterRegistry advisorAdapterRegistry =
GlobalAdvisorAdapterRegistry.getInstance();
5    public void setAdvisorAdapterRegistry(AdvisorAdapterRegistry
advisorAdapterRegistry) {
6        this.advisorAdapterRegistry = advisorAdapterRegistry;
7    }
8    //BeanPostProcessor在Bean对象初始化前的操作
9    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
10        //没有做任何操作，直接返回容器创建的Bean对象
11        return bean;
12    }
13    //BeanPostProcessor在Bean对象初始化后的操作
14    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
15        if (bean instanceof AdvisorAdapter){
16            //如果容器创建的Bean实例对象是一个切面通知适配器，则向容器的注册
this.advisorAdapterRegistry.registerAdvisorAdapter((AdvisorAdapter) bean);
17        }
18        return bean;
19    }
}
```

其他的BeanPostProcessor接口实现类的也类似，都是对Bean对象使用到的一些特性进行处理，或者向IoC容器中注册，为创建的Bean实例对象做一些自定义的功能增加，这些操作是容器初始化Bean时自动触发的，不需要认为的干预。

5.Spring IoC容器autowiring实现原理：

Spring IoC容器提供了两种管理Bean依赖关系的方式：

a. 显式管理：通过BeanDefinition的属性值和构造方法实现Bean依赖关系管理。

b． autowiring：Spring IoC容器的依赖自动装配功能，不需要对Bean属性的依赖关系做显式的声明，只需要在配置好autowiring属性，IoC容器会自动使用反射查找属性的类型和名称，然后基于属性的类型或者名称来自动匹配容器中管理的Bean，从而自动地完成依赖注入。

通过对autowiring自动装配特性的理解，我们知道容器对Bean的自动装配发生在容器对Bean依赖注入的过程中。在前面对Spring IoC容器的依赖注入过程源码分析中，我们已经知道了容器对Bean实例对象的属性注入的处理发生在AbstractAutoWireCapableBeanFactory类中的populateBean方法中，我们通过程序流程分析autowiring的实现原理：

(1). AbstractAutoWireCapableBeanFactory对Bean实例进行属性依赖注入：

应用第一次通过getBean方法(配置了lazy-init预实例化属性的除外)向IoC容器索取Bean时，容器创建Bean实例对象，并且对Bean实例对象进行属性依赖注入，AbstractAutoWireCapableBeanFactory的populateBean方法就是实现Bean属性依赖注入的功能，其主要源码如下：

```
1    protected void populateBean(String beanName, AbstractBeanDefinition mbd,
BeanWrapper bw) {
2         //获取Bean定义的属性值，并对属性值进行处理
3         PropertyValues pvs = mbd.getPropertyValues();
4         ……
5         //对依赖注入处理，首先处理autowiring自动装配的依赖注入
6         if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME
||
7                 mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
8             MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
9             //根据Bean名称进行autowiring自动装配处理
10             if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME) {
11                 autowireByName(beanName, mbd, bw, newPvs);
12             }
13             //根据Bean类型进行autowiring自动装配处理
14             if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
15                 autowireByType(beanName, mbd, bw, newPvs);
16             }
17         }
18         //对非autowiring的属性进行依赖注入处理
19          ……
    }
```

(2).Spring IoC容器根据Bean名称或者类型进行autowiring自动依赖注入：

```
1    //根据名称对属性进行自动依赖注入
2    protected void autowireByName(
3            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw,
MutablePropertyValues pvs) {
4         //对Bean对象中非简单属性(不是简单继承的对象，如8中原始类型，字符串，URL等//都是简
单属性)进行处理
5        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
6        for (String propertyName : propertyNames) {
7            //如果Spring IoC容器中包含指定名称的Bean
8            if (containsBean(propertyName)) {
9                //调用getBean方法向IoC容器索取指定名称的Bean实例，迭代触发属性的//初始化
和依赖注入
10               Object bean = getBean(propertyName);
11               //为指定名称的属性赋予属性值
12               pvs.add(propertyName, bean);
13               //指定名称属性注册依赖Bean名称，进行属性依赖注入
14               registerDependentBean(propertyName, beanName);
15               if (logger.isDebugEnabled()) {
16                   logger.debug("Added autowiring by name from bean name '" +
beanName +
17                           "' via property '" + propertyName + "' to bean
named '" + propertyName + "'");
18               }
19           }
20           else {
21               if (logger.isTraceEnabled()) {
22                   logger.trace("Not autowiring property '" + propertyName +
"' of bean '" + beanName +
23                           "' by name: no matching bean found");
24               }
25           }
26       }
27   }
28   //根据类型对属性进行自动依赖注入
29   protected void autowireByType(
30           String beanName, AbstractBeanDefinition mbd, BeanWrapper bw,
MutablePropertyValues pvs) {
31       //获取用户定义的类型转换器
32       TypeConverter converter = getCustomTypeConverter();
33       if (converter == null) {
34           converter = bw;
35       }
36       //存放解析的要注入的属性
37       Set<String> autowiredBeanNames = new LinkedHashSet<String>(4);
38         //对Bean对象中非简单属性(不是简单继承的对象，如8中原始类型，字符
39        //URL等都是简单属性)进行处理
40       String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
41       for (String propertyName : propertyNames) {
42           try {
43               //获取指定属性名称的属性描述器
44               PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);

45               //不对Object类型的属性进行autowiring自动依赖注入
46               if (!Object.class.equals(pd.getPropertyType())) {
47                   //获取属性的setter方法
48                   MethodParameter methodParam =
BeanUtils.getWriteMethodParameter(pd);
```

```
49                      //检查指定类型是否可以被转换为目标对象的类型
50                      boolean eager =
!PriorityOrdered.class.isAssignableFrom(bw.getWrappedClass());
51                      //创建一个要被注入的依赖描述
52                      DependencyDescriptor desc = new
AutowireByTypeDependencyDescriptor(methodParam, eager);
53                      //根据容器的Bean定义解析依赖关系，返回所有要被注入的Bean对象
54                      Object autowiredArgument = resolveDependency(desc,
beanName, autowiredBeanNames, converter);
55                      if (autowiredArgument != null) {
56                          //为属性赋值所引用的对象
57                          pvs.add(propertyName, autowiredArgument);
58                      }
59                      for (String autowiredBeanName : autowiredBeanNames) {
60                          //指定名称属性注册依赖Bean名称，进行属性依赖注入
61                          registerDependentBean(autowiredBeanName, beanName);
62                          if (logger.isDebugEnabled()) {
63                              logger.debug("Autowiring by type from bean name '"
+ beanName + "' via property '" +
64                                      propertyName + "' to bean named '" +
autowiredBeanName + "'");
65                          }
66                      }
67                      //释放已自动注入的属性
68                      autowiredBeanNames.clear();
69                  }
70              }
71              catch (BeansException ex) {
72                  throw new
UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName,
propertyName, ex);
73              }
74          }
    }
```

通过上面的源码分析，我们可以看出来通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是DefaultSingletonBeanRegistry类的registerDependentBean方法。

(3).DefaultSingletonBeanRegistry的registerDependentBean方法对属性注入：

```
1    //为指定的Bean注入依赖的Bean
2    public void registerDependentBean(String beanName, String dependentBeanName)
{
3        //处理Bean名称，将别名转换为规范的Bean名称
4        String canonicalName = canonicalName(beanName);
5        //多线程同步，保证容器内数据的一致性
6        //先从容器中：bean名称-->全部依赖Bean名称集合找查找给定名称Bean的依赖Bean
7        synchronized (this.dependentBeanMap) {
8            //获取给定名称Bean的所有依赖Bean名称
9            Set<String> dependentBeans =
this.dependentBeanMap.get(canonicalName);
```

```
            if (dependentBeans == null) {
                //为Bean设置依赖Bean信息
                dependentBeans = new LinkedHashSet<String>(8);
                this.dependentBeanMap.put(canonicalName, dependentBeans);
            }
            //向容器中：bean名称-->全部依赖Bean名称集合添加Bean的依赖信息
            //即，将Bean所依赖的Bean添加到容器的集合中
            dependentBeans.add(dependentBeanName);
        }
          //从容器中：bean名称-->指定名称Bean的依赖Bean集合找查找给定名称
         //Bean的依赖Bean
        synchronized (this.dependenciesForBeanMap) {
            Set<String> dependenciesForBean =
this.dependenciesForBeanMap.get(dependentBeanName);
            if (dependenciesForBean == null) {
                dependenciesForBean = new LinkedHashSet<String>(8);
                this.dependenciesForBeanMap.put(dependentBeanName,
dependenciesForBean);
            }
            //向容器中：bean名称-->指定Bean的依赖Bean名称集合添加Bean的依赖信息
            //即，将Bean所依赖的Bean添加到容器的集合中
            dependenciesForBean.add(canonicalName);
        }
    }
```

通过对autowiring的源码分析，我们可以看出，autowiring的实现过程：

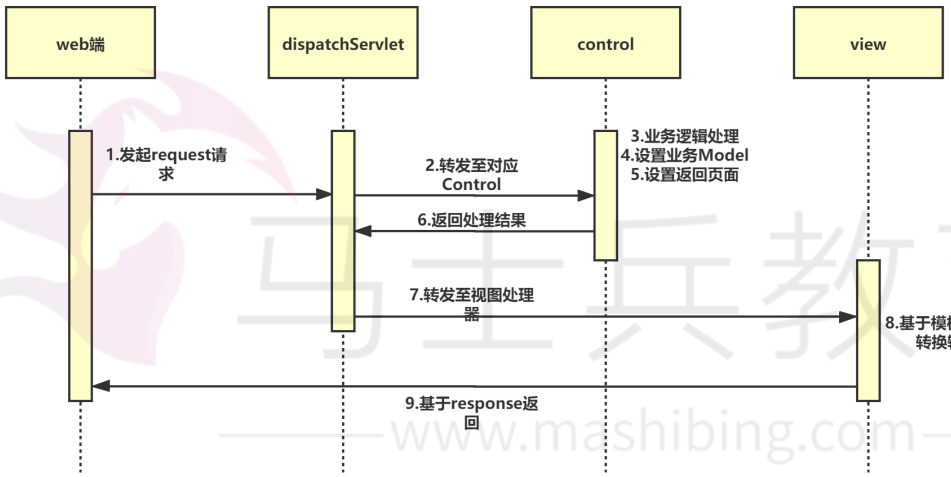
a. 对Bean的属性迭代调用getBean方法，完成依赖Bean的初始化和依赖注入。

b. 将依赖Bean的属性引用设置到被依赖的Bean属性上。

c. 将依赖Bean的名称和被依赖Bean的名称存储在IoC容器的集合中。

Spring IoC容器的autowiring属性自动依赖注入是一个很方便的特性，可以简化开发时的配置，但是凡是都有两面性，自动属性依赖注入也有不足，首先，Bean的依赖关系在配置文件中无法很清楚地看出来，对于维护造成一定困难。其次，由于自动依赖注入是Spring容器自动执行的，容器是不会智能判断的，如果配置不当，将会带来无法预料的后果，所以自动依赖注入特性在使用时还是综合考虑。

SpringMVC



马士兵教育
www.mashibing.com

动态代理

反射

AOP原理

Spring事务;

√ ： 可能出现　　× ： 不会出现

	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

一、spring事务

什么是事务: 事务逻辑上的一组操作,组成这组操作的各个逻辑单元,要么一起成功,要么一起失败.

二、事务特性（4种）:

- 原子性 （atomicity）:强调事务的不可分割.
- 一致性 （consistency）:事务的执行的前后数据的完整性保持一致.
- 隔离性 （isolation）:一个事务执行的过程中,不应该受到其他事务的干扰
- 持久性（durability） :事务一旦结束,数据就持久到数据库

如果不考虑隔离性引发安全性问题:

脏读 :一个事务读到了另一个事务的未提交的数据

不可重复读 :一个事务读到了另一个事务已经提交的 update 的数据导致多次查询结果不一致.

虚幻读 :一个事务读到了另一个事务已经提交的 insert 的数据导致多次查询结果不一致.

三、解决读问题: 设置事务隔离级别（5种）

DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔 离级别.

未提交读（read uncommited）:脏读，不可重复读，虚读都有可能发生

已提交读 （read commited）:避免脏读。但是不可重复读和虚读有可能发生

可重复读 （repeatable read）:避免脏读和不可重复读.但是虚读有可能发生.

串行化的 （serializable）:避免以上所有读问题.

Mysql 默认:可重复读

Oracle 默认:读已提交

四、事务的传播行为

PROPAGION_XXX :事务的传播行为

- 保证同一个事务中

PROPAGATION_REQUIRED 支持当前事务，如果不存在 就新建一个(默认)

PROPAGATION_SUPPORTS 支持当前事务，如果不存在，就不使用事务

PROPAGATION_MANDATORY 支持当前事务，如果不存在，抛出异常

- 保证没有在同一个事务中

PROPAGATION_REQUIRES_NEW 如果有事务存在，挂起当前事务，创建一个新的事务

PROPAGATION_NOT_SUPPORTED 以非事务方式运行，如果有事务存在，挂起当前事务

PROPAGATION_NEVER 以非事务方式运行，如果有事务存在，抛出异常

PROPAGATION_NESTED 如果当前事务存在，则嵌套事务执行

springMVC的原理

Spring MVC工作原理 及注解说明

SpringMVC框架介绍

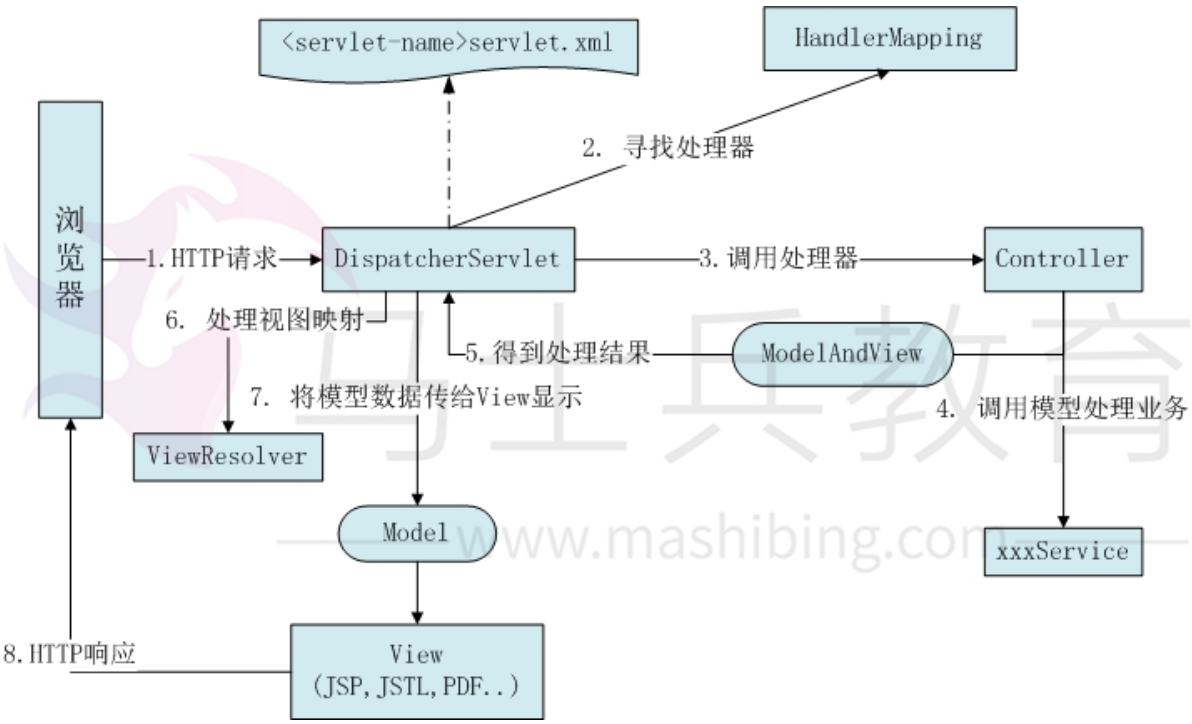
1) spring MVC属于SpringFrameWork的后续产品，已经融合在Spring Web Flow里面。

> Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，可以选择是使用内置的 Spring Web 框架还是 Struts 这样的 Web 框架。通过策略接口，Spring 框架是高度可配置的，而且包含多种视图技术，例如 JavaServer Pages（JSP）技术、Velocity、Tiles、iText 和 POI。Spring MVC 框架并不知道使用的视图，所以不会强迫您只使用 JSP 技术。

```
Spring MVC 分离了控制器、模型对象、分派器以及处理程序对象的角色，这种分离让它们更容易进行定制。
```

2) Spring的MVC框架主要由DispatcherServlet、处理器映射、处理器(控制器)、视图解析器、视图组成。

SpringMVC原理图



SpringMVC接口解释

DispatcherServlet接口：

> Spring提供的前端控制器，所有的请求都有经过它来统一分发。在DispatcherServlet将请求分发给Spring Controller之前，需要借助于Spring提供的HandlerMapping定位到具体的Controller。

HandlerMapping接口：

> 能够完成客户请求到Controller映射。

Controller接口：

> 需要为并发用户处理上述请求，因此实现Controller接口时，必须保证线程安全并且可重用。
>
> Controller将处理用户请求，这和Struts Action扮演的角色是一致的。一旦Controller处理完用户请求，则返回ModelAndView对象给DispatcherServlet前端控制器，ModelAndView中包含了模型（Model）和视图（View）。
>
> 从宏观角度考虑，DispatcherServlet是整个Web应用的控制器；从微观考虑，Controller是单个Http请求处理过程中的控制器，而ModelAndView是Http请求过程中返回的模型（Model）和视图（View）。

ViewResolver接口：

Spring提供的视图解析器（ViewResolver）在Web应用中查找View对象，从而将相应结果渲染给客户。

SpringMVC运行原理

> 1. 客户端请求提交到DispatcherServlet
> 2. 由DispatcherServlet控制器查询一个或多个HandlerMapping，找到处理请求的Controller
> 3. DispatcherServlet将请求提交到Controller
> 4. Controller调用业务逻辑处理后，返回ModelAndView

> 5. DispatcherServlet查询一个或多个ViewResoler视图解析器，找到ModelAndView指定的视图
> 6. 视图负责将结果显示到客户端

DispatcherServlet是整个Spring MVC的核心。它负责接收HTTP请求组织协调Spring MVC的各个组成部分。其主要工作有以下三项：

```
1. 截获符合特定格式的URL请求。
2. 初始化DispatcherServlet上下文对应的WebApplicationContext，并将其与业务层、持久化层的WebApplicationContext建立关联。
3. 初始化Spring MVC的各个组成组件，并装配到DispatcherServlet中。
```

<mvc:annotation-driven /> 说明:

是一种简写形式，可以让初学者快速成应用默认的配置方案，会默认注册DefaultAnnotationHandleMapping以及AnnotionMethodHandleAdapter 这两个 Bean, 这两个 Bean ，前者对应类级别， 后者对应到方法级别；

上在面的 DefaultAnnotationHandlerMapping和AnnotationMethodHandlerAdapter 是 Spring 为 @Controller 分发请求所必需的。

annotation-driven 扫描指定包中类上的注解，常用的注解有:

复制代码

@Controller 声明Action组件

@Service 声明Service组件 @Service("myMovieLister")

@Repository 声明Dao组件

@Component 泛指组件, 当不好归类时.

@RequestMapping("/menu") 请求映射

@Resource 用于注入，(j2ee提供的) 默认按名称装配，@Resource(name="beanName")

@Autowired 用于注入，(srping提供的) 默认按类型装配

@Transactional(rollbackFor={Exception.class}) 事务管理

@ResponseBody

@Scope("prototype") 设定bean的作用域

spring中beanFactory和ApplicationContext的联系和区别

Spring系列之beanFactory与ApplicationContext

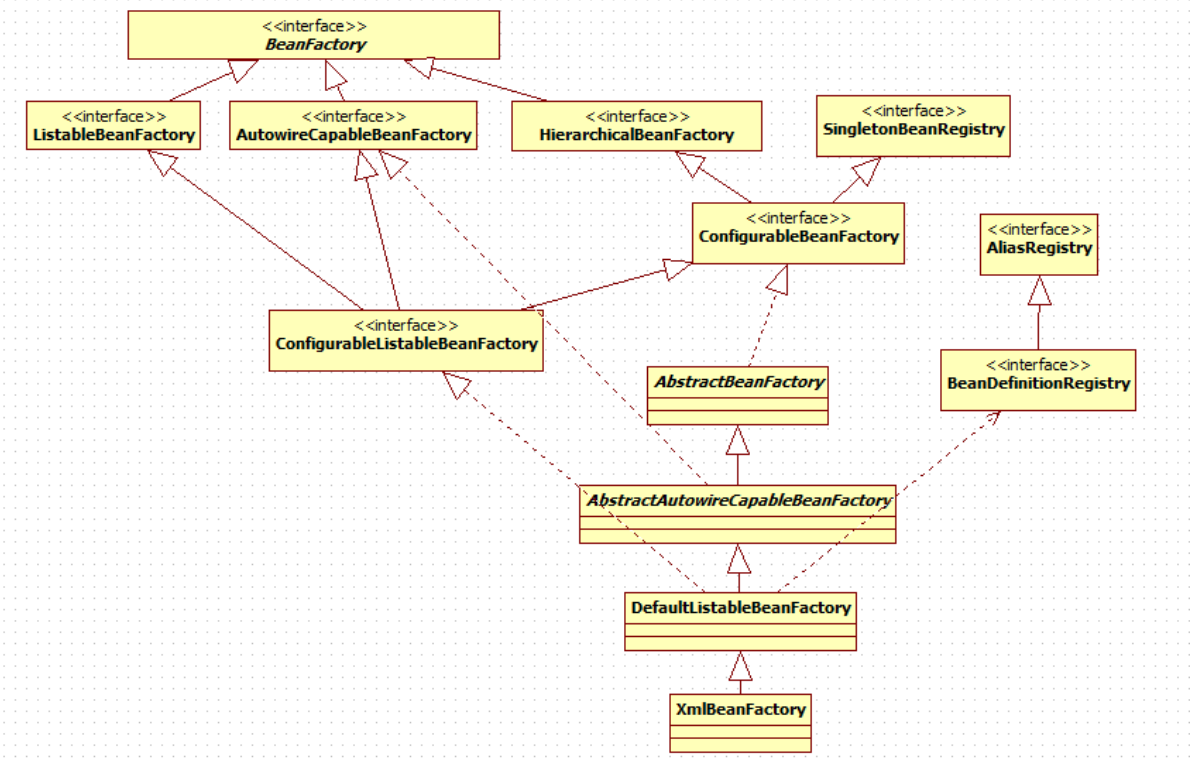
一、BeanFactory

BeanFactory 是 Spring 的“心脏”。它就是 Spring IoC 容器的真面目。Spring 使用 BeanFactory 来实例化、配置和管理 Bean。

BeanFactory：是IOC容器的核心接口， 它定义了IOC的基本功能，我们看到它主要定义了getBean方法。getBean方法是IOC容器获取bean对象和引发依赖注入的起点。方法的功能是返回特定的名称的Bean。

BeanFactory 是初始化 Bean 和调用它们生命周期方法的“吃苦耐劳者”。注意，BeanFactory 只能管理单例（Singleton）Bean 的生命周期。它不能管理原型(prototype,非单例)Bean 的生命周期。这是因为原型 Bean 实例被创建之后便被传给了客户端,容器失去了对它们的引用。

BeanFactory有着庞大的继承、实现体系，有众多的子接口、实现类。来看一下BeanFactory的基本类体系结构（接口为主）：



这是我画的BeanFactory基本的类体系结构，这里没有包括强大的ApplicationContext体系。
具体：

1、BeanFactory作为一个主接口不继承任何接口，暂且称为**一级接口。**

2、有3个子接口继承了它，进行功能上的增强。这3个子接口称为**二级接口。**

3、ConfigurableBeanFactory可以被称为**三级接口**，对二级接口HierarchicalBeanFactory进行了再次增强，它还继承了另一个外来的接口SingletonBeanRegistry

4、ConfigurableListableBeanFactory是一个更强大的接口，继承了上述的所有接口，无所不包，称为**四级接口。**

（这4级接口是BeanFactory的基本接口体系。继续，下面是继承关系的2个抽象类和2个实现类：）

5、AbstractBeanFactory作为一个抽象类，实现了三级接口ConfigurableBeanFactory大部分功能。

6、AbstractAutowireCapableBeanFactory同样是抽象类，继承自AbstractBeanFactory，并额外实现了二级接口AutowireCapableBeanFactory

7、DefaultListableBeanFactory继承自AbstractAutowireCapableBeanFactory，实现了最强大的四级接口ConfigurableListableBeanFactory，并实现了一个外来接口BeanDefinitionRegistry，它并非抽象类。

8、最后是最强大的XmlBeanFactory，继承自DefaultListableBeanFactory，重写了一些功能，使自己更强大。

总结：

BeanFactory的类体系结构看似繁杂混乱，实际上由上而下井井有条，非常容易理解。

再来看一下BeanFactory的源码：

```
package org.springframework.beans.factory;

public interface BeanFactory {

    /**
     * 用来引用一个实例，或把它和工厂产生的Bean区分开，就是说，如果一个FactoryBean的名字为a，那么，&a会得到那个Factory
     */
    String FACTORY_BEAN_PREFIX = "&";

    /*
     * 四个不同形式的getBean方法，获取实例
     */
    Object getBean(String name) throws BeansException;

    <T> T getBean(String name, Class<T> requiredType) throws BeansException;

    <T> T getBean(Class<T> requiredType) throws BeansException;

    Object getBean(String name, Object... args) throws BeansException;

    boolean containsBean(String name); // 是否存在

    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;// 是否为单实例

    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;// 是否为原型（多实例）

    boolean isTypeMatch(String name, Class<?> targetType)
            throws NoSuchBeanDefinitionException;// 名称、类型是否匹配

    Class<?> getType(String name) throws NoSuchBeanDefinitionException; // 获取类型

    String[] getAliases(String name);// 根据实例的名字获取实例的别名

}
```

具体：

1、4个获取实例的方法。getBean的重载方法。

2、4个判断的方法。判断是否存在，是否为单例、原型，名称类型是否匹配。

3、1个获取类型的方法、一个获取别名的方法。根据名称获取类型、根据名称获取别名。一目了然!

总结：

这10个方法，很明显，这是一个典型的工厂模式的工厂接口。

BeanFactory最常见的实现类为XmlBeanFactory，可以从classpath或文件系统等获取资源。

```
（1）File file = new File("fileSystemConfig.xml");
Resource resource = new FileSystemResource(file);
BeanFactory beanFactory = new XmlBeanFactory(resource);
（2）
Resource resource = new ClassPathResource("classpath.xml");
BeanFactory beanFactory = new XmlBeanFactory(resource);
```

XmlBeanFactory可以加载xml的配置文件。假设我们有一个Car类：

```
package spring.ioc.demo1;

public class Car {

    private String brand;

    private String color;

    private int maxSpeed;

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public String toString(){
```

```
        return "the car is:"+ getBrand() + ", color is:" +getColor() +", maxspeed
is:"+getMaxSpeed();
    }

    public Car() {

    }

    public Car(String brand, String color, int maxSpeed) {
        this.brand = brand;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }

    public void introduce() {
        System.out.println("brand:" + brand + ";color:" + color + ";maxSpeed:"
                + maxSpeed);
    }

}
```

我们通过在applicationContext.xml中配置：

```
<bean id="car1" class="spring.ioc.demo1.Car"
        p:brand="spring注入-红旗001"
        p:color="spring注入-紫色"
        p:maxSpeed="520" />
```

通过XmlBeanFactory实现启动Spring IoC容器：

```
public static void main(String[] args) {

      ResourcePatternResolver resolver = new
PathMatchingResourcePatternResolver();
      Resource res = resolver.getResource("classpath:applicationContext.xml");
      BeanFactory factory = new XmlBeanFactory(res);
       //ApplicationContext factory=new
ClassPathXmlApplicationContext("applicationContext.xml");

       Car car = factory.getBean("car1",Car.class);
       System.out.println("car对象已经初始化完成");
       System.out.println(car.getMaxSpeed());
}
```

1. XmlBeanFactory通过Resource装载Spring配置信息冰启动IoC容器，然后就可以通过factory.getBean从IoC容器中获取Bean了。
2. 通过BeanFactory启动IoC容器时，并不会初始化配置文件中定义的Bean，初始化动作发生在第一个调用时。
3. 对于单实例（singleton）的Bean来说，BeanFactory会缓存Bean实例，所以第二次使用getBean时直接从IoC容器缓存中获取Bean。

二、ApplicationContext

如果说BeanFactory是Spring的心脏，那么ApplicationContext就是完整的躯体了，ApplicationContext由BeanFactory派生而来，提供了更多面向实际应用的功能。在BeanFactory中，很多功能需要以编程的方式实现，而在ApplicationContext中则可以通过配置实现。

BeanFactorty接口提供了配置框架及基本功能，但是无法支持spring的aop功能和web应用。而ApplicationContext接口作为BeanFactory的派生，因而提供BeanFactory所有的功能。而且ApplicationContext还在功能上做了扩展，相较于BeanFactorty，ApplicationContext还提供了以下的功能：

（1）MessageSource, 提供国际化的消息访问
（2）资源访问，如URL和文件
（3）事件传播特性，即支持aop特性
（4）载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

ApplicationContext：是IOC容器另一个重要接口， 它继承了BeanFactory的基本功能， 同时也继承了容器的高级功能，如：MessageSource（国际化资源接口）、ResourceLoader（资源加载接口）、ApplicationEventPublisher（应用事件发布接口）等。

三、二者区别

1.BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化，这样，我们就不能发现一些存在的Spring的配置问题。而ApplicationContext则相反，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误。 相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

BeanFacotry延迟加载,如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常；而ApplicationContext则在初始化自身是检验，这样有利于检查所依赖属性是否注入；所以通常情况下我们选择使用 ApplicationContext。
应用上下文则会在上下文启动后预载入所有的单实例Bean。通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

2.BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。
（Applicationcontext比 beanFactory 加入了一些更好使用的功能。而且 beanFactory 的许多功能需要通过编程实现而 Applicationcontext 可以通过配置实现。比如后处理 bean ， Applicationcontext 直接配置在配置文件即可而 beanFactory 这要在代码中显示的写出来才可以被容器识别。 ）

3.beanFactory主要是面对与 spring 框架的基础设施，面对 spring 自己。而 Applicationcontex 主要面对与 spring 使用的开发者。基本都会使用 Applicationcontex 并非 beanFactory 。

四、总结

作用：
\1. BeanFactory负责读取bean配置文档，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的声明周期。

\2. ApplicationContext除了提供上述BeanFactory所能提供的功能之外，还提供了更完整的框架功能：
a. 国际化支持
b. 资源访问：Resource rs = ctx. getResource(“classpath:config.properties”), “file:c:/config.properties”
c. 事件传递：通过实现ApplicationContextAware接口

\3. 常用的获取ApplicationContext

FileSystemXmlApplicationContext：从文件系统或者url指定的xml配置文件创建，参数为配置文件名或文件名数组，有相对路径与绝对路径。

```
ApplicationContext factory=new
FileSystemXmlApplicationContext("src/applicationContext.xml");
ApplicationContext factory=new
FileSystemXmlApplicationContext("E:/Workspaces/MyEclipse
8.5/Hello/src/applicationContext.xml");
```

ClassPathXmlApplicationContext：从classpath的xml配置文件创建，可以从jar包中读取配置文件。
ClassPathXmlApplicationContext 编译路径总有三种方式：

```
ApplicationContext factory = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
ApplicationContext factory = new
ClassPathXmlApplicationContext("applicationContext.xml");
ApplicationContext factory = new
ClassPathXmlApplicationContext("file:E:/Workspaces/MyEclipse
8.5/Hello/src/applicationContext.xml");
```

XmlWebApplicationContext：从web应用的根目录读取配置文件，需要先在web.xml中配置，可以配置监听器或者servlet来实现

```
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

或

```
<servlet>
<servlet-name>context</servlet-name>
<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-
class>
<load-on-startup>1</load-on-startup>
</servlet>
```

这两种方式都默认配置文件为web-inf/applicationContext.xml，也可使用context-param指定配置文件

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/myApplicationContext.xml</param-value>
</context-param>
```

spring注入的几种方式（循环注入）

Spring循环依赖的三种方式

引言：循环依赖就是N个类中循环嵌套引用，如果在日常开发中我们用new 对象的方式发生这种循环依赖的话程序会在运行时一直循环调用，直至内存溢出报错。下面说一下Spring是如果解决循环依赖的。

第一种：构造器参数循环依赖

Spring容器会将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持
在这个池中，因此如果在创建Bean过程中发现自己已经在“当前创建Bean池”里时将抛出BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的Bean将从“当前创建Bean池”中清除掉。

首先我们先初始化三个Bean。

```
public class StudentA {
    private StudentB studentB ;
    public void setStudentB(StudentB studentB) {
        this.studentB = studentB;
    }
    public StudentA() {
    }
    public StudentA(StudentB studentB) {
        this.studentB = studentB;
    }
}
```

```
public class StudentB {
    private StudentC studentC ;
    public void setStudentC(StudentC studentC) {
        this.studentC = studentC;
    }

    public StudentB() {
    }
    public StudentB(StudentC studentC) {
        this.studentC = studentC;
    }
}
public class StudentC {
    private StudentA studentA ;
    public void setStudentA(StudentA studentA) {
        this.studentA = studentA;
    }

    public StudentC() {
    }
```

```
    public StudentC(StudentA studentA) {
        this.studentA = studentA;
    }
}
```

OK，上面是很基本的3个类，，StudentA有参构造是StudentB。StudentB的有参构造是StudentC，StudentC的有参构造是StudentA，这样就产生了一个循环依赖的情况，

我们都把这三个Bean交给Spring管理，并用有参构造实例化

```
<bean id="a" class="com.zfx.student.StudentA">
        <constructor-arg index="0" ref="b"></constructor-arg>
    </bean>
    <bean id="b" class="com.zfx.student.StudentB">
        <constructor-arg index="0" ref="c"></constructor-arg>
    </bean>
    <bean id="c" class="com.zfx.student.StudentC">
        <constructor-arg index="0" ref="a"></constructor-arg>
    </bean>
```

下面是测试类：

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("com/zfx/student/applicationContext.xml");
        //System.out.println(context.getBean("a", StudentA.class));
    }
}
```

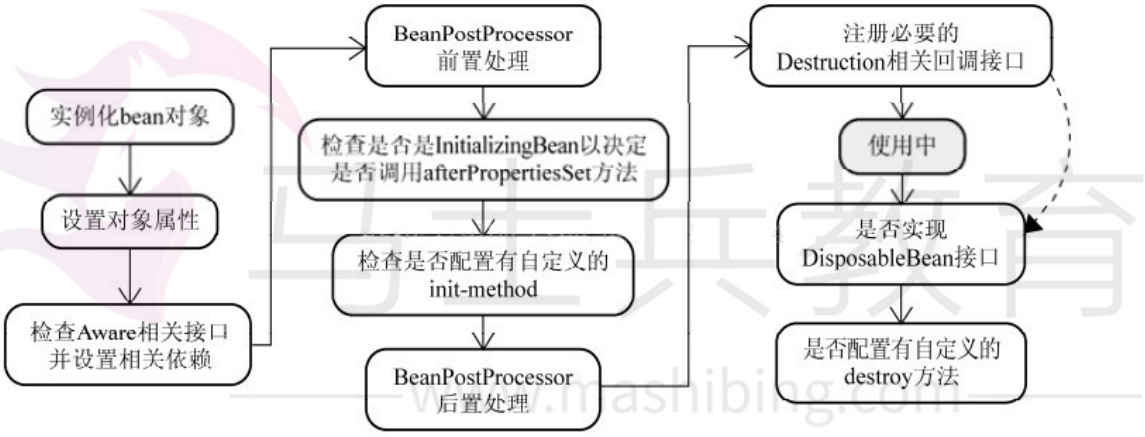
执行结果报错信息为：

```
Caused by: org.springframework.beans.factory.BeanCurrentlyInCreationException:
    Error creating bean with name 'a': Requested bean is currently in creation:
Is there an unresolvable circular reference?
```

如果大家理解开头那句话的话，这个报错应该不惊讶，Spring容器先创建单例StudentA，StudentA依赖StudentB，然后将A放在“当前创建Bean池”中，此时创建StudentB,StudentB依赖StudentC ,然后将B放在“当前创建Bean池”中,此时创建StudentC，StudentC又依赖StudentA， 但是，此时Student已经在池中，所以会报错，，因为在池中的Bean都是未初始化完的，所以会依赖错误，（初始化完的Bean会从池中移除）

第二种：setter方式单例，默认方式

如果要说setter方式注入的话，我们最好先看一张Spring中Bean实例化的图



如图中前两步骤得知：***Spring是先将Bean对象实例化之后再设置对象属性的***

*

*

修改配置文件为set方式注入

```
    <!--scope="singleton"(默认就是单例方式)  -->
    <bean id="a" class="com.zfx.student.StudentA" scope="singleton">
        <property name="studentB" ref="b"></property>
    </bean>
    <bean id="b" class="com.zfx.student.StudentB" scope="singleton">
        <property name="studentC" ref="c"></property>
    </bean>
    <bean id="c" class="com.zfx.student.StudentC" scope="singleton">
        <property name="studentA" ref="a"></property>
    </bean>
```

下面是测试类：

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("com/zfx/student/applicationContext.xml");
        System.out.println(context.getBean("a", StudentA.class));
    }
}
```

打印结果为：

```
com.zfx.student.StudentA@1fbfd6
```

为什么用set方式就不报错了呢？

我们结合上面那张图看，Spring先是用构造实例化Bean对象，此时Spring会将这个实例化结束的对象放到一个Map中，并且Spring提供了获取这个未设置属性的实例化对象引用的方法。 结合我们的实例来看，，当Spring实例化了StudentA、StudentB、StudentC后，紧接着会去设置对象的属性，此时StudentA依赖StudentB，就会去Map中取出存在里面的单例StudentB对象，以此类推，不会出来循环的问题喽、

**
**

下面是Spring源码中的实现方法，。以下的源码在Spring的Bean包中的
***DefaultSingletonBeanRegistry*.java**类中

```
/** Cache of singleton objects: bean name --> bean instance（缓存单例实例化对象的Map集合） */
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(64);

    /** Cache of singleton factories: bean name --> ObjectFactory（单例的工厂Bean缓存集合） */
    private final Map<String, ObjectFactory> singletonFactories = new HashMap<String, ObjectFactory>(16);

    /** Cache of early singleton objects: bean name --> bean instance（早期的单身对象缓存集合） */
    private final Map<String, Object> earlySingletonObjects = new HashMap<String, Object>(16);

    /** Set of registered singletons, containing the bean names in registration order（单例的实例化对象名称集合） */
    private final Set<String> registeredSingletons = new LinkedHashSet<String>(64);
    /**
     * 添加单例实例
     * 解决循环引用的问题
     * Add the given singleton factory for building the specified singleton
     * if necessary.
     * <p>To be called for eager registration of singletons, e.g. to be able to
     * resolve circular references.
     * @param beanName the name of the bean
     * @param singletonFactory the factory for the singleton object
     */
    protected void addSingletonFactory(String beanName, ObjectFactory singletonFactory) {
        Assert.notNull(singletonFactory, "Singleton factory must not be null");
        synchronized (this.singletonObjects) {
            if (!this.singletonObjects.containsKey(beanName)) {
                this.singletonFactories.put(beanName, singletonFactory);
                this.earlySingletonObjects.remove(beanName);
                this.registeredSingletons.add(beanName);
            }
        }
    }
```

第三种：setter方式原型，prototype

修改配置文件为：

```
<bean id="a" class="com.zfx.student.StudentA" scope="prototype">
        <property name="studentB" ref="b"></property>
    </bean>
    <bean id="b" class="com.zfx.student.StudentB" scope="prototype">
        <property name="studentC" ref="c"></property>
    </bean>
    <bean id="c" class="com.zfx.student.StudentC" scope="prototype">
        <property name="studentA" ref="a"></property>
    </bean>
```

scope="prototype" 意思是 每次请求都会创建一个实例对象。两者的区别是：有状态的bean都使用Prototype作用域，无状态的一般都使用singleton单例作用域。

测试用例：

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("com/zfx/student/applicationContext.xml");
        //此时必须要获取Spring管理的实例，因为现在scope="prototype" 只有请求获取的时候才会实例化对象
        System.out.println(context.getBean("a", StudentA.class));
    }
}
```

打印结果：

```
Caused by: org.springframework.beans.factory.BeanCurrentlyInCreationException:
    Error creating bean with name 'a': Requested bean is currently in creation:
Is there an unresolvable circular reference?
```

为什么原型模式就报错了呢？

对于“prototype”作用域Bean，Spring容器无法完成依赖注入，因为“prototype”作用域的Bean，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。

springIOC

Spring IOC原理解读 面试必读

一、什么是Ioc/DI?

IoC 容器：最主要是完成了完成对象的创建和依赖的管理注入等等。

先从我们自己设计这样一个视角来考虑：

所谓控制反转，就是把原先我们代码里面需要实现的对象创建、依赖的代码，反转给容器来帮忙实现。那么必然的我们需要创建一个容器，同时需要一种描述来让容器知道需要创建的对象与对象的关系。这个描述最具体表现就是我们可配置的文件。

对象和对象关系怎么表示?

可以用 xml， properties 文件等语义化配置文件表示。

描述对象关系的文件存放在哪里?

可能是 classpath， filesystem，或者是 URL 网络资源， servletContext 等。

回到正题，有了配置文件，还需要对配置文件解析。

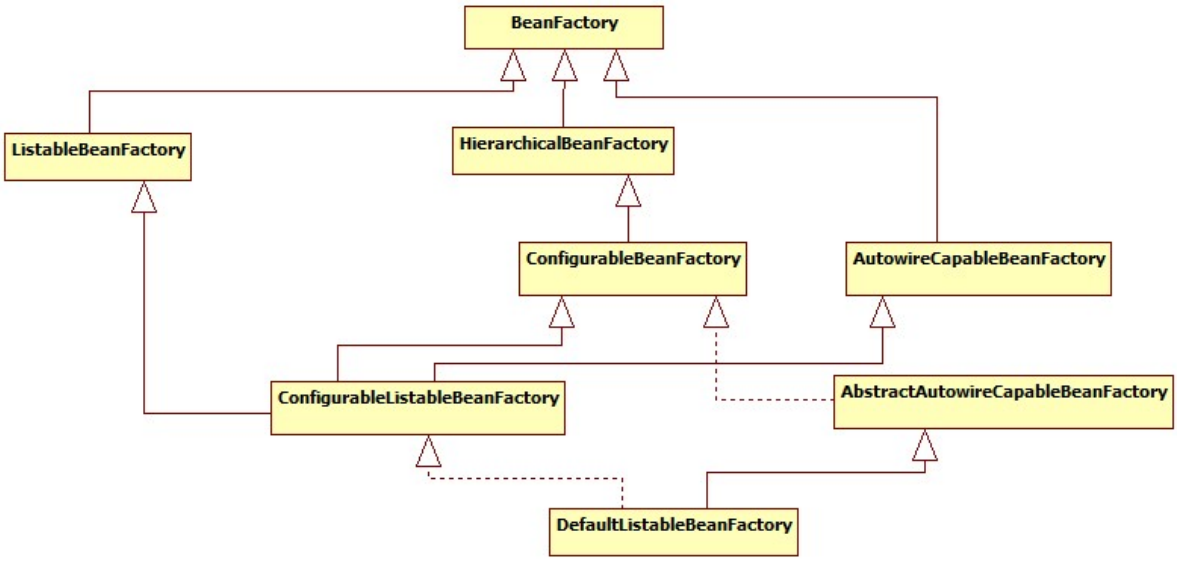
不同的配置文件对对象的描述不一样，如标准的，自定义声明式的，如何统一？ 在内部需要有一个统一的关于对象的定义，所有外部的描述都必须转化成统一的描述定义。

如何对不同的配置文件进行解析？需要对不同的配置文件语法，采用不同的解析器

二、 Spring IOC体系结构?

(1) BeanFactory

> Spring Bean的创建是典型的工厂模式，这一系列的Bean工厂，也即IOC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IOC容器的实现供用户选择和使用，其相互关系如下：



其中BeanFactory作为最顶层的一个接口类，它定义了IOC容器的基本功能规范，BeanFactory 有三个子类：ListableBeanFactory、HierarchicalBeanFactory 和AutowireCapableBeanFactory。但是从上图中我们可以发现最终的默认实现类是 DefaultListableBeanFactory，他实现了所有的接口。那为何要定义这么多层次的接口呢？查阅这些接口的源码和说明发现，每个接口都有他使用的场合，它主要是为了区分在 Spring 内部在操作过程中对象的传递和转化过程中，对对象的数据访问所做的限制。例如 ListableBeanFactory 接口表示这些 Bean 是可列表的，而 HierarchicalBeanFactory 表示的是这些 Bean 是有继承关系的，也就是每个Bean 有可能有父 Bean。AutowireCapableBeanFactory 接口定义 Bean 的自动装配规则。这四个接口共同定义了 Bean 的集合、Bean 之间的关系、以及 Bean 行为.

最基本的IOC容器接口BeanFactory

```
1 public interface BeanFactory {
2
3       //对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成的对象，
```

```
    //如果需要得到工厂本身，需要转义
    String FACTORY_BEAN_PREFIX = "&";

    //根据bean的名字，获取在IOC容器中得到bean实例
    Object getBean(String name) throws BeansException;

    //根据bean的名字和Class类型来得到bean实例，增加了类型安全验证机制。
     Object getBean(String name, Class requiredType) throws BeansException;


    //提供对bean的检索，看看是否在IOC容器有这个名字的bean
     boolean containsBean(String name);

    //根据bean名字得到bean实例，并同时判断这个bean是不是单例
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;


    //得到bean实例的Class类型
    Class getType(String name) throws NoSuchBeanDefinitionException;

    //得到bean的别名，如果根据别名检索，那么其原名也会被检索出来
   String[] getAliases(String name);

 }
```

在BeanFactory里只对IOC容器的基本行为作了定义，根本不关心你的bean是如何定义怎样加载的。正如我们只关心工厂里得到什么的产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心。

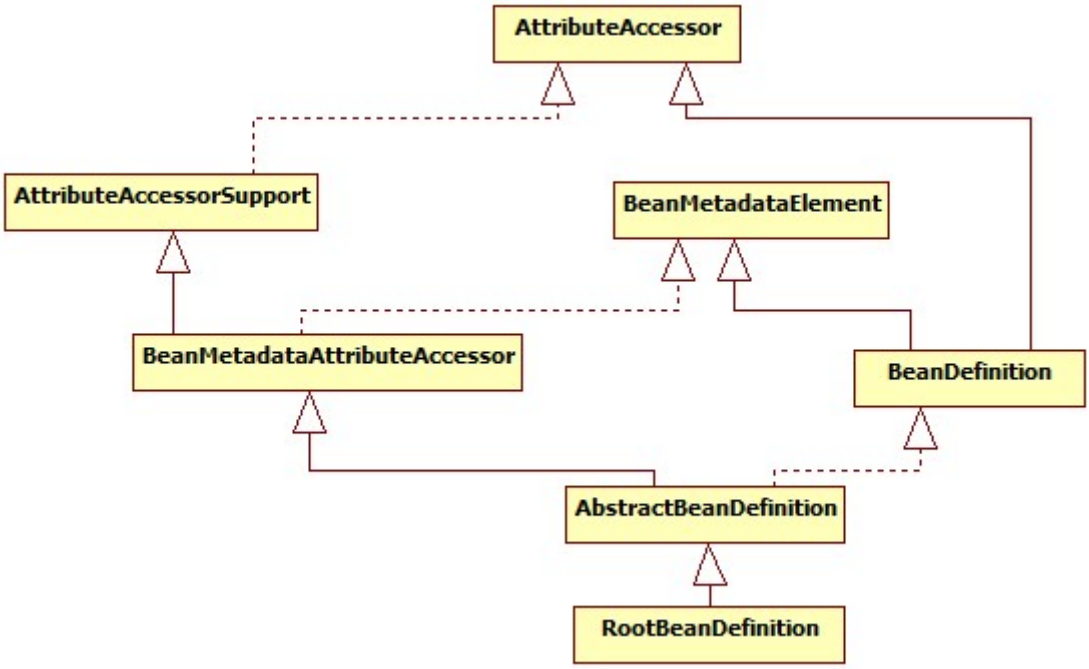
> 而要知道工厂是如何产生对象的，我们需要看具体的IOC容器实现，spring提供了许多IOC容器的实现。比如XmlBeanFactory，ClasspathXmlApplicationContext等。其中XmlBeanFactory就是针对最基本的ioc容器的实现，这个IOC容器可以读取XML文件定义的BeanDefinition（XML文件中对bean的描述）,如果说XmlBeanFactory是容器中的屌丝，ApplicationContext应该算容器中的高帅富.
>
> ApplicationContext是Spring提供的一个高级的IoC容器，它除了能够提供IoC容器的基本功能外，还为用户提供了以下的附加服务。

从ApplicationContext接口的实现，我们看出其特点：

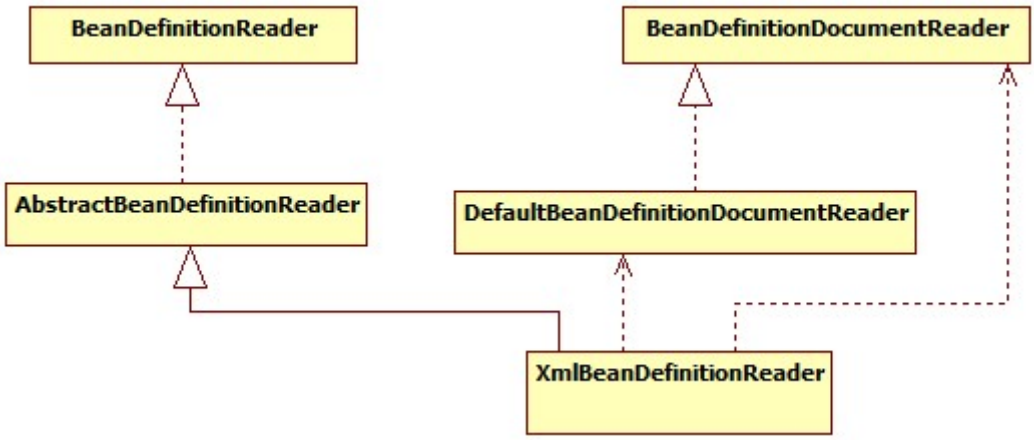
> 1. 支持信息源，可以实现国际化。（实现MessageSource接口）
>
> 2. 访问资源。(实现ResourcePatternResolver接口，这个后面要讲)
>
> 3. 支持应用事件。(实现ApplicationEventPublisher接口)

(2) BeanDefinition

> SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Bean对象在Spring实现中是以BeanDefinition来描述的，其继承体系如下：

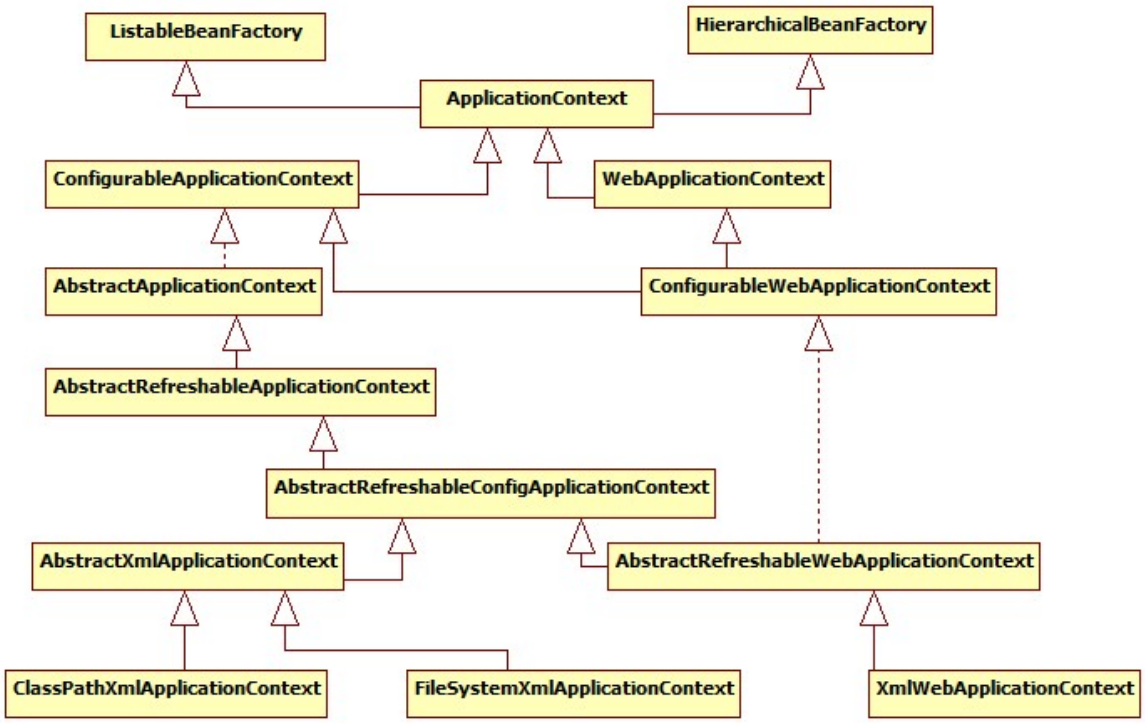


Bean 的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean 的解析主要就是对 Spring 配置文件的解析。这个解析过程主要通过下图中的类完成：



三、IoC容器的初始化?

IoC容器的初始化包括BeanDefinition的Resource定位、载入和注册这三个基本的过程。我们以ApplicationContext为例讲解，ApplicationContext系列容器也许是我们最熟悉的，因为web项目中使用的XmlWebApplicationContext就属于这个继承体系，还有ClasspathXmlApplicationContext等，其继承体系如下图所示：



ApplicationContext允许上下文嵌套，通过保持父上下文可以维持一个上下文体系。对于bean的查找可以在这个上下文体系中发生，首先检查当前上下文，其次是父上下文，逐级向上，这样为不同的Spring应用提供了一个共享的bean定义环境。

下面我们分别简单地演示一下两种ioc容器的创建过程

1、XmlBeanFactory(屌丝IOC)的整个流程

通过XmlBeanFactory的源码，我们可以发现:

```
public class XmlBeanFactory extends DefaultListableBeanFactory{

    private final XmlBeanDefinitionReader reader;

    public XmlBeanFactory(Resource resource)throws BeansException{
        this(resource, null);
    }

    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory)
         throws BeansException{
        super(parentBeanFactory);
        this.reader = new XmlBeanDefinitionReader(this);
        this.reader.loadBeanDefinitions(resource);
   }
}
```

```
//根据Xml配置文件创建Resource资源对象，该对象中包含了BeanDefinition的信息
 ClassPathResource resource =new ClassPathResource("application-context.xml");
//创建DefaultListableBeanFactory
 DefaultListableBeanFactory factory =new DefaultListableBeanFactory();
//创建XmlBeanDefinitionReader读取器，用于载入BeanDefinition。之所以需要BeanFactory作为参数，是因为会将读取的信息回调配置给factory
 XmlBeanDefinitionReader reader =new XmlBeanDefinitionReader(factory);
//XmlBeanDefinitionReader执行载入BeanDefinition的方法，最后会完成Bean的载入和注册。完成后Bean就成功的放置到IOC容器当中，以后我们就可以从中取得Bean来使用
 reader.loadBeanDefinitions(resource);
```

通过前面的源码，this.reader = new XmlBeanDefinitionReader(this); 中其中this 传的是factory对象

2、FileSystemXmlApplicationContext 的IOC容器流程

1、高富帅IOC解剖

1 ApplicationContext =new FileSystemXmlApplicationContext(xmlPath);

先看其构造函数：

调用构造函数：

```
/**
* Create a new FileSystemXmlApplicationContext, loading the definitions
* from the given XML files and automatically refreshing the context.
* @param configLocations array of file paths
* @throws BeansException if context creation failed
 */public FileSystemXmlApplicationContext(String... configLocations) throws
BeansException {
        this(configLocations, true, null);
    }
```

实际调用

```
public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh,
ApplicationContext parent)
            throws BeansException {
        super(parent);
        setConfigLocations(configLocations);
        if (refresh) {
            refresh();
        }
    }
```

2、设置资源加载器和资源定位

通过分析FileSystemXmlApplicationContext的源代码可以知道，在创建FileSystemXmlApplicationContext容器时，构造方法做以下两项重要工作：

首先，调用父类容器的构造方法(super(parent)方法)为容器设置好Bean资源加载器。

然后，再调用父类AbstractRefreshableConfigApplicationContext的setConfigLocations(configLocations)方法设置Bean定义资源文件的定位路径。

通过追踪FileSystemXmlApplicationContext的继承体系，发现其父类的父类AbstractApplicationContext中初始化IoC容器所做的主要源码如下：

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader
        implements ConfigurableApplicationContext, DisposableBean {
    //静态初始化块，在整个容器创建过程中只执行一次
    static {
        //为了避免应用程序在Weblogic8.1关闭时出现类加载异常加载问题，加载IoC容
       //器关闭事件(ContextClosedEvent)类
        ContextClosedEvent.class.getName();
    }
    //FileSystemXmlApplicationContext调用父类构造方法调用的就是该方法
    public AbstractApplicationContext(ApplicationContext parent) {
        this.parent = parent;
        this.resourcePatternResolver = getResourcePatternResolver();
    }
    //获取一个Spring Source的加载器用于读入Spring Bean定义资源文件
    protected ResourcePatternResolver getResourcePatternResolver() {
        // AbstractApplicationContext继承DefaultResourceLoader，也是一个S
        //Spring资源加载器，其getResource(String location)方法用于载入资源
        return new PathMatchingResourcePatternResolver(this);
    }
……
}
```

AbstractApplicationContext构造方法中调用PathMatchingResourcePatternResolver的构造方法创建Spring资源加载器：

```
public PathMatchingResourcePatternResolver(ResourceLoader resourceLoader) {
        Assert.notNull(resourceLoader, "ResourceLoader must not be null");
        //设置Spring的资源加载器
        this.resourceLoader = resourceLoader;
}
```

在设置容器的资源加载器之后，接下来FileSystemXmlApplicationContet执行setConfigLocations方法通过调用其父类AbstractRefreshableConfigApplicationContext的方法进行对Bean定义资源文件的定位，该方法的源码如下：

```
    //处理单个资源文件路径为一个字符串的情况
    public void setConfigLocation(String location) {
       //String CONFIG_LOCATION_DELIMITERS = ",; /t/n";
       //即多个资源文件路径之间用” ,; /t/n”分隔，解析成数组形式
        setConfigLocations(StringUtils.tokenizeToStringArray(location,
CONFIG_LOCATION_DELIMITERS));
    }
    //解析Bean定义资源文件的路径，处理多个资源文件字符串数组
     public void setConfigLocations(String[] locations) {
        if (locations != null) {
```

```
            Assert.noNullElements(locations, "Config locations must not be
null");
            this.configLocations = new String[locations.length];
            for (int i = 0; i < locations.length; i++) {
                // resolvePath为同一个类中将字符串解析为路径的方法
                this.configLocations[i] = resolvePath(locations[i]).trim();
            }
        }
        else {
            this.configLocations = null;
        }  www
    }
```

通过这两个方法的源码我们可以看出，我们既可以使用一个字符串来配置多个Spring Bean定义资源文件，也可以使用字符串数组，即下面两种方式都是可以的：

a. ClasspathResource res = new ClasspathResource(“a.xml,b.xml,……”);

多个资源文件路径之间可以是用” ,; /t/n”等分隔。

b. ClasspathResource res = new ClasspathResource(newString[]{“a.xml”,”b.xml”,……});

至此，Spring IoC容器在初始化时将配置的Bean定义资源文件定位为Spring封装的Resource。

3、AbstractApplicationContext的refresh函数载入Bean定义过程：

Spring IoC容器对Bean定义资源的载入是从refresh()函数开始的，refresh()是一个模板方法，refresh()方法的作用是：在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器。refresh的作用类似于对IoC容器的重启，在新建立好的容器中对容器进行初始化，对Bean定义资源进行载入

FileSystemXmlApplicationContext通过调用其父类AbstractApplicationContext的refresh()函数启动整个IoC容器对Bean定义的载入过程：

```
1      public void refresh() throws BeansException, IllegalStateException {
2        synchronized (this.startupShutdownMonitor) {
3            //调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识
4            prepareRefresh();
5            //告诉子类启动refreshBeanFactory()方法，Bean定义资源文件的载入从
6           //子类的refreshBeanFactory()方法启动
7            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();
8            //为BeanFactory配置容器特性，例如类加载器、事件处理器等
9            prepareBeanFactory(beanFactory);
10            try {
11                //为容器的某些子类指定特殊的BeanPost事件处理器
12                postProcessBeanFactory(beanFactory);
13                //调用所有注册的BeanFactoryPostProcessor的Bean
14                invokeBeanFactoryPostProcessors(beanFactory);
15                //为BeanFactory注册BeanPost事件处理器.
16                //BeanPostProcessor是Bean后置处理器，用于监听容器触发的事件
17                registerBeanPostProcessors(beanFactory);
18                //初始化信息源，和国际化相关.
19                initMessageSource();
```

```
20                //初始化容器事件传播器.
21                initApplicationEventMulticaster();
22                //调用子类的某些特殊Bean初始化方法
23                onRefresh();
24                //为事件传播器注册事件监听器.
25                registerListeners();
26                //初始化所有剩余的单态Bean.
27                finishBeanFactoryInitialization(beanFactory);
28                //初始化容器的生命周期事件处理器，并发布容器的生命周期事件
29                finishRefresh();
30            }
31            catch (BeansException ex) {
32                //销毁以创建的单态Bean
33                destroyBeans();
34                //取消refresh操作，重置容器的同步标识.
35                cancelRefresh(ex);
36                throw ex;
37            }
38        }
39    }
```

refresh()方法主要为IoC容器Bean的生命周期管理提供条件，Spring IoC容器载入Bean定义资源文件从其子类容器的refreshBeanFactory()方法启动，所以整个refresh()中“ConfigurableListableBeanFactory beanFactory =obtainFreshBeanFactory();”这句以后代码的都是注册容器的信息源和生命周期事件，载入过程就是从这句代码启动。

refresh()方法的作用是：在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器。refresh的作用类似于对IoC容器的重启，在新建立好的容器中对容器进行初始化，对Bean定义资源进行载入

AbstractApplicationContext的obtainFreshBeanFactory()方法调用子类容器的refreshBeanFactory()方法，启动容器载入Bean定义资源文件的过程，代码如下：

```
    protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
        //这里使用了委派设计模式，父类定义了抽象的refreshBeanFactory()方法，具体实现调用子类容器的refreshBeanFactory()方法
         refreshBeanFactory();
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
        if (logger.isDebugEnabled()) {
            logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
        }
        return beanFactory;
    }
```

AbstractApplicationContext子类的refreshBeanFactory()方法：

AbstractApplicationContext类中只抽象定义了refreshBeanFactory()方法，容器真正调用的是其子类AbstractRefreshableApplicationContext实现的 refreshBeanFactory()方法，方法的源码如下：

```
1    protected final void refreshBeanFactory() throws BeansException {
2        if (hasBeanFactory()) {//如果已经有容器，销毁容器中的bean，关闭容器
3            destroyBeans();
4            closeBeanFactory();
5        }
6        try {
7             //创建IoC容器
8             DefaultListableBeanFactory beanFactory = createBeanFactory();
9             beanFactory.setSerializationId(getId());
10            //对IoC容器进行定制化，如设置启动参数，开启注解的自动装配等
11            customizeBeanFactory(beanFactory);
12            //调用载入Bean定义的方法，主要这里又使用了一个委派模式，在当前类中只定义了抽象
的loadBeanDefinitions方法，具体的实现调用子类容器
13            loadBeanDefinitions(beanFactory);
14            synchronized (this.beanFactoryMonitor) {
15                this.beanFactory = beanFactory;
16            }
17        }
18        catch (IOException ex) {
19            throw new ApplicationContextException("I/O error parsing bean
definition source for " + getDisplayName(), ex);
20        }
21    }
```

在这个方法中，先判断BeanFactory是否存在，如果存在则先销毁beans并关闭beanFactory，接着创建DefaultListableBeanFactory，并调用loadBeanDefinitions(beanFactory)装载bean定义。

5、AbstractRefreshableApplicationContext子类的loadBeanDefinitions方法：

AbstractRefreshableApplicationContext中只定义了抽象的loadBeanDefinitions方法，容器真正调用的是其子类AbstractXmlApplicationContext对该方法的实现，AbstractXmlApplicationContext的主要源码如下：

loadBeanDefinitions方法同样是抽象方法，是由其子类实现的，也即在AbstractXmlApplicationContext中。

```
1  public abstract class AbstractXmlApplicationContext extends
AbstractRefreshableConfigApplicationContext {
2      ……
3      //实现父类抽象的载入Bean定义方法
4      @Override
5      protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
throws BeansException, IOException {
6          //创建XmlBeanDefinitionReader，即创建Bean读取器，并通过回调设置到容器中去，容
器使用该读取器读取Bean定义资源
7          XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);
```

```
8          //为Bean读取器设置Spring资源加载器，AbstractXmlApplicationContext的
9          //祖先父类AbstractApplicationContext继承DefaultResourceLoader，因此，容器本
身也是一个资源加载器
10         beanDefinitionReader.setResourceLoader(this);
11         //为Bean读取器设置SAX xml解析器
12         beanDefinitionReader.setEntityResolver(new
ResourceEntityResolver(this));
13         //当Bean读取器读取Bean定义的Xml资源文件时，启用Xml的校验机制
14         initBeanDefinitionReader(beanDefinitionReader);
15         //Bean读取器真正实现加载的方法
16         loadBeanDefinitions(beanDefinitionReader);
17     }
18     //Xml Bean读取器加载Bean定义资源
19     protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
BeansException, IOException {
20         //获取Bean定义资源的定位
21         Resource[] configResources = getConfigResources();
22         if (configResources != null) {
23             //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位
24             //的Bean定义资源
25             reader.loadBeanDefinitions(configResources);
26         }
27         //如果子类中获取的Bean定义资源定位为空，则获取FileSystemXmlApplicationContext
构造方法中setConfigLocations方法设置的资源
28         String[] configLocations = getConfigLocations();
29         if (configLocations != null) {
30             //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位
31             //的Bean定义资源
32             reader.loadBeanDefinitions(configLocations);
33         }
34     }
35     //这里又使用了一个委托模式，调用子类的获取Bean定义资源定位的方法
36     //该方法在ClassPathXmlApplicationContext中进行实现，对于我们
37     //举例分析源码的FileSystemXmlApplicationContext没有使用该方法
38     protected Resource[] getConfigResources() {
39         return null;
40     }   ……
41}
```

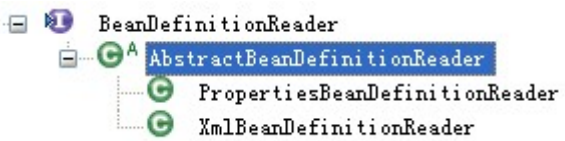
ml Bean读取器(XmlBeanDefinitionReader)调用其父类AbstractBeanDefinitionReader的reader.loadBeanDefinitions方法读取Bean定义资源

由于我们使用FileSystemXmlApplicationContext作为例子分析，因此getConfigResources的返回值为null，因此程序执行reader.loadBeanDefinitions(configLocations)分支。

6、AbstractBeanDefinitionReader读取Bean定义资源：

AbstractBeanDefinitionReader的loadBeanDefinitions方法源码如下：

可以到org.springframework.beans.factory.support看一下BeanDefinitionReader的结构



在其抽象父类AbstractBeanDefinitionReader中定义了载入过程

```
1    //重载方法，调用下面的loadBeanDefinitions(String, Set<Resource>);方法
2    public int loadBeanDefinitions(String location) throws
BeanDefinitionStoreException {
3        return loadBeanDefinitions(location, null);
4    }
5    public int loadBeanDefinitions(String location, Set<Resource>
actualResources) throws BeanDefinitionStoreException {
6        //获取在IoC容器初始化过程中设置的资源加载器
7        ResourceLoader resourceLoader = getResourceLoader();
8        if (resourceLoader == null) {
9            throw new BeanDefinitionStoreException(
10                   "Cannot import bean definitions from location [" + location
+ "]: no ResourceLoader available");
11        }
12        if (resourceLoader instanceof ResourcePatternResolver) {
13            try {
14                //将指定位置的Bean定义资源文件解析为Spring IoC容器封装的资源
15                //加载多个指定位置的Bean定义资源文件
16                Resource[] resources = ((ResourcePatternResolver)
resourceLoader).getResources(location);
17                //委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能
18                int loadCount = loadBeanDefinitions(resources);
19                if (actualResources != null) {
20                    for (Resource resource : resources) {
21                        actualResources.add(resource);
22                    }
23                }
24                if (logger.isDebugEnabled()) {
25                    logger.debug("Loaded " + loadCount + " bean definitions
from location pattern [" + location + "]");
26                }
27                return loadCount;
28            }
29            catch (IOException ex) {
30                throw new BeanDefinitionStoreException(
31                        "Could not resolve bean definition resource pattern ["
+ location + "]", ex);
32            }
33        }
34        else {
35            //将指定位置的Bean定义资源文件解析为Spring IoC容器封装的资源
36            //加载单个指定位置的Bean定义资源文件
37            Resource resource = resourceLoader.getResource(location);
38            //委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能
39            int loadCount = loadBeanDefinitions(resource);
40            if (actualResources != null) {
41                actualResources.add(resource);
42            }
43            if (logger.isDebugEnabled()) {
44                logger.debug("Loaded " + loadCount + " bean definitions from
location [" + location + "]");
45            }
46            return loadCount;
47        }
48    }
```

```
49    //重载方法，调用loadBeanDefinitions(String);
50    public int loadBeanDefinitions(String... locations) throws
BeanDefinitionStoreException {
51        Assert.notNull(locations, "Location array must not be null");
52        int counter = 0;
53        for (String location : locations) {
54            counter += loadBeanDefinitions(location);
55        }
56        return counter;
    }
```

loadBeanDefinitions(Resource…resources)方法和上面分析的3个方法类似，同样也是调用XmlBeanDefinitionReader的loadBeanDefinitions方法。

从对AbstractBeanDefinitionReader的loadBeanDefinitions方法源码分析可以看出该方法做了以下两件事：

首先，调用资源加载器的获取资源方法resourceLoader.getResource(location)，获取到要加载的资源。

其次，真正执行加载功能是其子类XmlBeanDefinitionReader的loadBeanDefinitions方法。

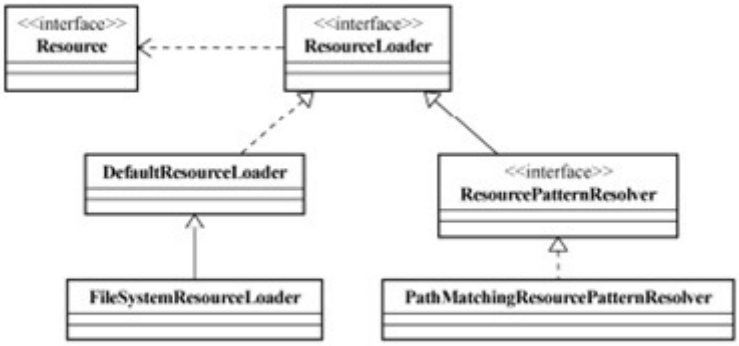
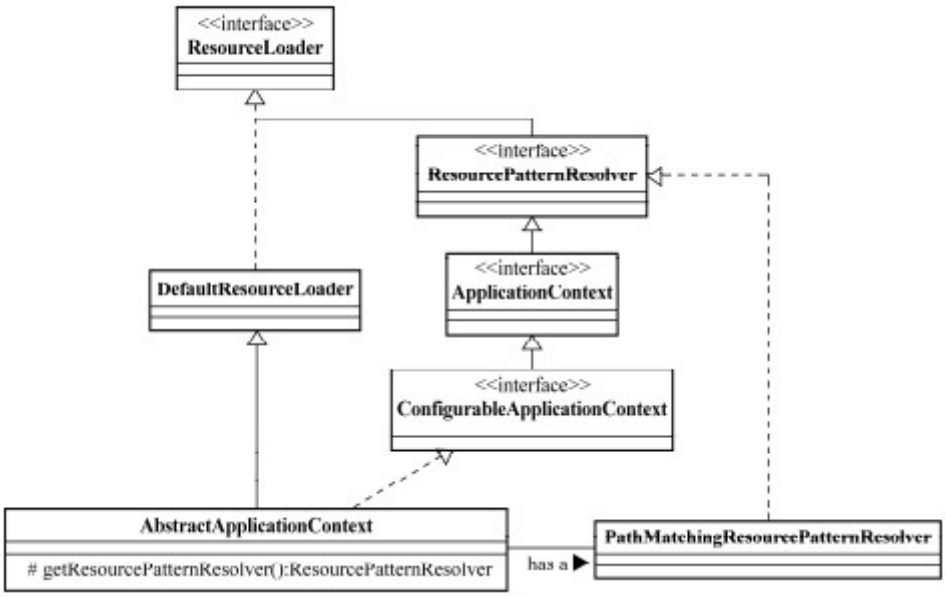


图5-1 Resource和ResourceLoader类层次图



看到第8、16行，结合上面的ResourceLoader与ApplicationContext的继承关系图，可以知道此时调用的是DefaultResourceLoader中的getSource()方法定位Resource，因为FileSystemXmlApplicationContext本身就是DefaultResourceLoader的实现类，所以此时又回到了FileSystemXmlApplicationContext中来。

7、资源加载器获取要读入的资源：

XmlBeanDefinitionReader通过调用其父类DefaultResourceLoader的getResource方法获取要加载的资源，其源码如下

```
1    //获取Resource的具体实现方法
2    public Resource getResource(String location) {
3        Assert.notNull(location, "Location must not be null");
4        //如果是类路径的方式，那需要使用ClassPathResource 来得到bean 文件的资源对象
5        if (location.startsWith(CLASSPATH_URL_PREFIX)) {
6            return new
ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()),
getClassLoader());
7        }
8         try {
9              // 如果是URL 方式，使用UrlResource 作为bean 文件的资源对象
10             URL url = new URL(location);
11             return new UrlResource(url);
12            }
13            catch (MalformedURLException ex) {
14            }
15            //如果既不是classpath标识，又不是URL标识的Resource定位，则调用
16            //容器本身的getResourceByPath方法获取Resource
17            return getResourceByPath(location);
18
19   }
```

FileSystemXmlApplicationContext容器提供了getResourceByPath方法的实现，就是为了处理既不是classpath标识，又不是URL标识的Resource定位这种情况。

```
protected Resource getResourceByPath(String path) {
   if (path != null && path.startsWith("/")) {
        path = path.substring(1);
    }
    //这里使用文件系统资源对象来定义bean 文件
    return new FileSystemResource(path);
}
```

这样代码就回到了 FileSystemXmlApplicationContext 中来，他提供了FileSystemResource 来完成从文件系统得到配置文件的资源定义。

这样，就可以从文件系统路径上对IOC 配置文件进行加载 - 当然我们可以按照这个逻辑从任何地方加载，在Spring 中我们看到它提供 的各种资源抽象，比如ClassPathResource, URLResource,FileSystemResource 等来供我们使用。上面我们看到的是定位Resource 的一个过程，而这只是加载过程的一部分.

8、XmlBeanDefinitionReader加载Bean定义资源：

Bean定义的Resource得到了

继续回到XmlBeanDefinitionReader的loadBeanDefinitions(Resource ...)方法看到代表bean文件的资源定义以后的载入过程。

```
1    //XmlBeanDefinitionReader加载资源的入口方法
2    public int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException {
3        //将读入的XML资源进行特殊编码处理
4        return loadBeanDefinitions(new EncodedResource(resource));
5    }
     //这里是载入XML形式Bean定义资源文件方法
6    public int loadBeanDefinitions(EncodedResource encodedResource) throws
BeanDefinitionStoreException {
7    .......
8    try {
9         //将资源文件转为InputStream的IO流
10        InputStream inputStream =
encodedResource.getResource().getInputStream();
11        try {
12           //从InputStream中得到XML的解析源
13            InputSource inputSource = new InputSource(inputStream);
14            if (encodedResource.getEncoding() != null) {
15                inputSource.setEncoding(encodedResource.getEncoding());
16            }
17            //这里是具体的读取过程
18            return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
19        }
20        finally {
21            //关闭从Resource中得到的IO流
22            inputStream.close();
23        }
24    }
25       .........
26}
27    //从特定XML文件中实际载入Bean定义资源的方法
28    protected int doLoadBeanDefinitions(InputSource inputSource, Resource
resource)
29        throws BeanDefinitionStoreException {
30    try {
31        int validationMode = getValidationModeForResource(resource);
32        //将XML文件转换为DOM对象，解析过程由documentLoader实现
33        Document doc = this.documentLoader.loadDocument(
34                inputSource, this.entityResolver, this.errorHandler,
validationMode, this.namespaceAware);
```

```
35          //这里是启动对Bean定义解析的详细过程，该解析过程会用到Spring的Bean配置规则
36          return registerBeanDefinitions(doc, resource);
37        }
38        .......
      }
```

通过源码分析，载入Bean定义资源文件的最后一步是将Bean定义资源转换为Document对象，该过程由documentLoader实现

9、DocumentLoader将Bean定义资源转换为Document对象：

DocumentLoader将Bean定义资源转换成Document对象的源码如下：

```
1     //使用标准的JAXP将载入的Bean定义资源转换成document对象
2     public Document loadDocument(InputSource inputSource, EntityResolver
entityResolver,
3             ErrorHandler errorHandler, int validationMode, boolean
namespaceAware) throws Exception {
4         //创建文件解析器工厂
5         DocumentBuilderFactory factory =
createDocumentBuilderFactory(validationMode, namespaceAware);
6         if (logger.isDebugEnabled()) {
7             logger.debug("Using JAXP provider [" + factory.getClass().getName()
+ "]");
8         }
9         //创建文档解析器
10         DocumentBuilder builder = createDocumentBuilder(factory,
entityResolver, errorHandler);
11         //解析Spring的Bean定义资源
12         return builder.parse(inputSource);
13     }
14     protected DocumentBuilderFactory createDocumentBuilderFactory(int
validationMode, boolean namespaceAware)
15             throws ParserConfigurationException {
16         //创建文档解析工厂
17         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

18         factory.setNamespaceAware(namespaceAware);
19         //设置解析XML的校验
20         if (validationMode != XmlValidationModeDetector.VALIDATION_NONE) {
21             factory.setValidating(true);
22             if (validationMode == XmlValidationModeDetector.VALIDATION_XSD) {

23                 factory.setNamespaceAware(true);
24                 try {
25                     factory.setAttribute(SCHEMA_LANGUAGE_ATTRIBUTE,
XSD_SCHEMA_LANGUAGE);
26                 }
27                 catch (IllegalArgumentException ex) {
```

```
28                    ParserConfigurationException pcex = new
ParserConfigurationException(
29                            "Unable to validate using XSD: Your JAXP provider
[" + factory +
30                            "] does not support XML Schema. Are you running on
Java 1.4 with Apache Crimson? " +
31                            "Upgrade to Apache Xerces (or Java 1.5) for full
XSD support.");
32                    pcex.initCause(ex);
33                    throw pcex;
34                }
35            }
36        }
37        return factory;
38    }
```

该解析过程调用JavaEE标准的JAXP标准进行处理。

至此Spring IoC容器根据定位的Bean定义资源文件，将其加载读入并转换成为Document对象过程完成。

接下来我们要继续分析Spring IoC容器将载入的Bean定义资源文件转换为Document对象之后，是如何将其解析为Spring IoC管理的Bean对象并将其注册到容器中的。

10、XmlBeanDefinitionReader解析载入的Bean定义资源文件：

XmlBeanDefinitionReader类中的doLoadBeanDefinitions方法是从特定XML文件中实际载入Bean定义资源的方法，该方法在载入Bean定义资源之后将其转换为Document对象，接下来调用registerBeanDefinitions启动Spring IoC容器对Bean定义的解析过程，registerBeanDefinitions方法源码如下：

```
1    //按照Spring的Bean语义要求将Bean定义资源解析并转换为容器内部数据结构
2    public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
3        //得到BeanDefinitionDocumentReader来对xml格式的BeanDefinition解析
4        BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
5        //获得容器中注册的Bean数量
6        int countBefore = getRegistry().getBeanDefinitionCount();
7        //解析过程入口，这里使用了委派模式，BeanDefinitionDocumentReader只是个接口，//具
体的解析实现过程有实现类DefaultBeanDefinitionDocumentReader完成
8        documentReader.registerBeanDefinitions(doc,
createReaderContext(resource));
9        //统计解析的Bean数量
10        return getRegistry().getBeanDefinitionCount() - countBefore;
11    }
12    //创建BeanDefinitionDocumentReader对象，解析Document对象
```

```
13    protected BeanDefinitionDocumentReader createBeanDefinitionDocumentReader()
{
14        return
BeanDefinitionDocumentReader.class.cast(BeanUtils.instantiateClass(this.document
ReaderClass));
      }
```

Bean定义资源的载入解析分为以下两个过程：

首先，通过调用XML解析器将Bean定义资源文件转换得到Document对象，但是这些Document对象并没有按照Spring的Bean规则进行解析。这一步是载入的过程

其次，在完成通用的XML解析之后，按照Spring的Bean规则对Document对象进行解析。

按照Spring的Bean规则对Document对象解析的过程是在接口BeanDefinitionDocumentReader的实现类DefaultBeanDefinitionDocumentReader中实现的。

11、DefaultBeanDefinitionDocumentReader对Bean定义的Document对象解析：

BeanDefinitionDocumentReader接口通过registerBeanDefinitions方法调用其实现类DefaultBeanDefinitionDocumentReader对Document对象进行解析，解析的代码如下：

```
1     //根据Spring DTD对Bean的定义规则解析Bean定义Document对象
2     public void registerBeanDefinitions(Document doc, XmlReaderContext
readerContext) {
3         //获得XML描述符
4         this.readerContext = readerContext;
5         logger.debug("Loading bean definitions");
6         //获得Document的根元素
7         Element root = doc.getDocumentElement();
8         //具体的解析过程由BeanDefinitionParserDelegate实现，
9         //BeanDefinitionParserDelegate中定义了Spring Bean定义XML文件的各种元素
10        BeanDefinitionParserDelegate delegate = createHelper(readerContext,
root);
11        //在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
12        preProcessXml(root);
13        //从Document的根元素开始进行Bean定义的Document对象
14        parseBeanDefinitions(root, delegate);
15        //在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性
16        postProcessXml(root);
17    }
18    //创建BeanDefinitionParserDelegate，用于完成真正的解析过程
```

```
19    protected BeanDefinitionParserDelegate createHelper(XmlReaderContext
readerContext, Element root) {
20        BeanDefinitionParserDelegate delegate = new
BeanDefinitionParserDelegate(readerContext);
21        //BeanDefinitionParserDelegate初始化Document根元素
22        delegate.initDefaults(root);
23        return delegate;
24    }
25    //使用Spring的Bean规则从Document的根元素开始进行Bean定义的Document对象
26    protected void parseBeanDefinitions(Element root,
BeanDefinitionParserDelegate delegate) {
27        //Bean定义的Document对象使用了Spring默认的XML命名空间
28        if (delegate.isDefaultNamespace(root)) {
29            //获取Bean定义的Document对象根元素的所有子节点
30            NodeList nl = root.getChildNodes();
31            for (int i = 0; i < nl.getLength(); i++) {
32                Node node = nl.item(i);
33                //获得Document节点是XML元素节点
34                if (node instanceof Element) {
35                    Element ele = (Element) node;
36                //Bean定义的Document的元素节点使用的是Spring默认的XML命名空间
37                    if (delegate.isDefaultNamespace(ele)) {
38                        //使用Spring的Bean规则解析元素节点
39                        parseDefaultElement(ele, delegate);
40                    }
41                    else {
42                        //没有使用Spring默认的XML命名空间，则使用用户自定义的解//析规则
解析元素节点
43                        delegate.parseCustomElement(ele);
44                    }
45                }
46            }
47        }
48        else {
49            //Document的根节点没有使用Spring默认的命名空间，则使用用户自定义的
50            //解析规则解析Document根节点
51            delegate.parseCustomElement(root);
52        }
53    }
54    //使用Spring的Bean规则解析Document元素节点
55    private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate
delegate) {
56        //如果元素节点是<Import>导入元素，进行导入解析
57        if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
58            importBeanDefinitionResource(ele);
59        }
60        //如果元素节点是<Alias>别名元素，进行别名解析
61        else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
62            processAliasRegistration(ele);
63        }
64        //元素节点既不是导入元素，也不是别名元素，即普通的<Bean>元素，
65        //按照Spring的Bean规则解析元素
66        else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
67            processBeanDefinition(ele, delegate);
68        }
69    }
70    //解析<Import>导入元素，从给定的导入路径加载Bean定义资源到Spring IoC容器中
71    protected void importBeanDefinitionResource(Element ele) {
```

```
72          //获取给定的导入元素的location属性
73          String location = ele.getAttribute(RESOURCE_ATTRIBUTE);
74          //如果导入元素的location属性值为空，则没有导入任何资源，直接返回
75          if (!StringUtils.hasText(location)) {
76              getReaderContext().error("Resource location must not be empty",
ele);
77              return;
78          }
79          //使用系统变量值解析location属性值
80          location = SystemPropertyUtils.resolvePlaceholders(location);
81          Set<Resource> actualResources = new LinkedHashSet<Resource>(4);
82          //标识给定的导入元素的location是否是绝对路径
83          boolean absoluteLocation = false;
84          try {
85              absoluteLocation = ResourcePatternUtils.isUrl(location) ||
ResourceUtils.toURI(location).isAbsolute();
86          }
87          catch (URISyntaxException ex) {
88              //给定的导入元素的location不是绝对路径
89          }
90          //给定的导入元素的location是绝对路径
91          if (absoluteLocation) {
92              try {
93                  //使用资源读入器加载给定路径的Bean定义资源
94                  int importCount =
getReaderContext().getReader().loadBeanDefinitions(location, actualResources);
95                  if (logger.isDebugEnabled()) {
96                      logger.debug("Imported " + importCount + " bean definitions
from URL location [" + location + "]");
97                  }
98              }
99              catch (BeanDefinitionStoreException ex) {
100                  getReaderContext().error(
101                          "Failed to import bean definitions from URL location
[" + location + "]", ele, ex);
102              }
103          }
104          else {
105              //给定的导入元素的location是相对路径
106              try {
107                  int importCount;
108                  //将给定导入元素的location封装为相对路径资源
109                  Resource relativeResource =
getReaderContext().getResource().createRelative(location);
110                  //封装的相对路径资源存在
111                  if (relativeResource.exists()) {
112                      //使用资源读入器加载Bean定义资源
113                      importCount =
getReaderContext().getReader().loadBeanDefinitions(relativeResource);
114                      actualResources.add(relativeResource);
115                  }
116                  //封装的相对路径资源不存在
117                  else {
118                      //获取Spring IoC容器资源读入器的基本路径
119                      String baseLocation =
getReaderContext().getResource().getURL().toString();
120                      //根据Spring IoC容器资源读入器的基本路径加载给定导入
121                      //路径的资源
```

```
122                     importCount =
getReaderContext().getReader().loadBeanDefinitions(
123                             StringUtils.applyRelativePath(baseLocation,
location), actualResources);
124                 }
125                 if (logger.isDebugEnabled()) {
126                     logger.debug("Imported " + importCount + " bean
definitions from relative location [" + location + "]");
127                 }
128             }
129             catch (IOException ex) {
130                 getReaderContext().error("Failed to resolve current resource
location", ele, ex);
131             }
132             catch (BeanDefinitionStoreException ex) {
133                 getReaderContext().error("Failed to import bean definitions
from relative location [" + location + "]",
134                         ele, ex);
135             }
136         }
137         Resource[] actResArray = actualResources.toArray(new
Resource[actualResources.size()]);
138         //在解析完<Import>元素之后，发送容器导入其他资源处理完成事件
139         getReaderContext().fireImportProcessed(location, actResArray,
extractSource(ele));
140     }
141     //解析<Alias>别名元素，为Bean向Spring IoC容器注册别名
142     protected void processAliasRegistration(Element ele) {
143         //获取<Alias>别名元素中name的属性值
144         String name = ele.getAttribute(NAME_ATTRIBUTE);
145         //获取<Alias>别名元素中alias的属性值
146         String alias = ele.getAttribute(ALIAS_ATTRIBUTE);
147         boolean valid = true;
148         //<alias>别名元素的name属性值为空
149         if (!StringUtils.hasText(name)) {
150             getReaderContext().error("Name must not be empty", ele);
151             valid = false;
152         }
153         //<alias>别名元素的alias属性值为空
154         if (!StringUtils.hasText(alias)) {
155             getReaderContext().error("Alias must not be empty", ele);
156             valid = false;
157         }
158         if (valid) {
159             try {
160                 //向容器的资源读入器注册别名
161                 getReaderContext().getRegistry().registerAlias(name, alias);

162             }
163             catch (Exception ex) {
164                 getReaderContext().error("Failed to register alias '" + alias
+
165                         "' for bean with name '" + name + "'", ele, ex);
166             }
167             //在解析完<Alias>元素之后，发送容器别名处理完成事件
168             getReaderContext().fireAliasRegistered(name, alias,
extractSource(ele));
169         }
```

```
170    }
171    //解析Bean定义资源Document对象的普通元素
172    protected void processBeanDefinition(Element ele,
BeanDefinitionParserDelegate delegate) {
173        // BeanDefinitionHolder是对BeanDefinition的封装，即Bean定义的封装类
174        //对Document对象中<Bean>元素的解析由BeanDefinitionParserDelegate实现
BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
175        if (bdHolder != null) {
176            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele,
bdHolder);
177            try {
178               //向Spring IoC容器注册解析得到的Bean定义，这是Bean定义向IoC容器注册的入
口
BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
179            }
180            catch (BeanDefinitionStoreException ex) {
181                getReaderContext().error("Failed to register bean definition
with name '" +
182                        bdHolder.getBeanName() + "'", ele, ex);
183            }
184            //在完成向Spring IoC容器注册解析得到的Bean定义之后，发送注册事件
185            getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
186        }
187    }
```

通过上述Spring IoC容器对载入的Bean定义Document解析可以看出，我们使用Spring时，在Spring配置文件中可以使用元素来导入IoC容器所需要的其他资源，Spring IoC容器在解析时会首先将指定导入的资源加载进容器中。使用别名时，Spring IoC容器首先将别名元素所定义的别名注册到容器中。

对于既不是元素，又不是元素的元素，即Spring配置文件中普通的元素的解析由BeanDefinitionParserDelegate类的parseBeanDefinitionElement方法来实现。

12、BeanDefinitionParserDelegate解析Bean定义资源文件中的元素：

Bean定义资源文件中的和元素解析在DefaultBeanDefinitionDocumentReader中已经完成，对Bean定义资源文件中使用最多的元素交由BeanDefinitionParserDelegate来解析，其解析实现的源码如下：

```
1    //解析<Bean>元素的入口
2    public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
3        return parseBeanDefinitionElement(ele, null);
4    }
5    //解析Bean定义资源文件中的<Bean>元素，这个方法中主要处理<Bean>元素的id，name
6    //和别名属性
7    public BeanDefinitionHolder parseBeanDefinitionElement(Element ele,
BeanDefinition containingBean) {
8        //获取<Bean>元素中的id属性值
9        String id = ele.getAttribute(ID_ATTRIBUTE);
10        //获取<Bean>元素中的name属性值
11        String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
```

```
        ////获取<Bean>元素中的alias属性值
        List<String> aliases = new ArrayList<String>();
        //将<Bean>元素中的所有name属性值存放到别名中
        if (StringUtils.hasLength(nameAttr)) {
            String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr,
BEAN_NAME_DELIMITERS);
            aliases.addAll(Arrays.asList(nameArr));
        }
        String beanName = id;
        //如果<Bean>元素中没有配置id属性时，将别名中的第一个值赋值给beanName
        if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
            beanName = aliases.remove(0);
            if (logger.isDebugEnabled()) {
                logger.debug("No XML 'id' specified - using '" + beanName +
                        "' as bean name and " + aliases + " as aliases");
            }
        }
        //检查<Bean>元素所配置的id或者name的唯一性，containingBean标识<Bean>
        //元素中是否包含子<Bean>元素
        if (containingBean == null) {
            //检查<Bean>元素所配置的id、name或者别名是否重复
            checkNameUniqueness(beanName, aliases, ele);
        }
        //详细对<Bean>元素中配置的Bean定义进行解析的地方
        AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele,
beanName, containingBean);
        if (beanDefinition != null) {
            if (!StringUtils.hasText(beanName)) {
                try {
                    if (containingBean != null) {
                        //如果<Bean>元素中没有配置id、别名或者name，且没有包含
子//<Bean>元素，为解析的Bean生成一个唯一beanName并注册
                        beanName = BeanDefinitionReaderUtils.generateBeanName(

                                beanDefinition,
this.readerContext.getRegistry(), true);
                    }
                    else {
                        //如果<Bean>元素中没有配置id、别名或者name，且包含了
子//<Bean>元素，为解析的Bean使用别名向IoC容器注册
                        beanName =
this.readerContext.generateBeanName(beanDefinition);
                        //为解析的Bean使用别名注册时，为了向后兼容
                //Spring1.2/2.0，给别名添加类名后缀
                        String beanClassName =
beanDefinition.getBeanClassName();
                        if (beanClassName != null &&
                                beanName.startsWith(beanClassName) &&
beanName.length() > beanClassName.length() &&

!this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
                            aliases.add(beanClassName);
                        }
                    }
                    if (logger.isDebugEnabled()) {
                        logger.debug("Neither XML 'id' nor 'name' specified - "
+
```

```
57                                  "using generated bean name [" + beanName +
"]");
58                      }
59                  }
60                  catch (Exception ex) {
61                      error(ex.getMessage(), ele);
62                      return null;
63                  }
64              }
65              String[] aliasesArray = StringUtils.toStringArray(aliases);
66              return new BeanDefinitionHolder(beanDefinition, beanName,
aliasesArray);
67          }
68          //当解析出错时，返回null
69          return null;
70      }
71      //详细对<Bean>元素中配置的Bean定义其他属性进行解析，由于上面的方法中已经对//Bean的id、
name和别名等属性进行了处理，该方法中主要处理除这三个以外的其他属性数据
72      public AbstractBeanDefinition parseBeanDefinitionElement(
73              Element ele, String beanName, BeanDefinition containingBean) {
74          //记录解析的<Bean>
75          this.parseState.push(new BeanEntry(beanName));
76          //这里只读取<Bean>元素中配置的class名字，然后载入到BeanDefinition中去
77          //只是记录配置的class名字，不做实例化，对象的实例化在依赖注入时完成
78          String className = null;
79          if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
80              className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
81          }
82          try {
83              String parent = null;
84              //如果<Bean>元素中配置了parent属性，则获取parent属性的值
85              if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
86                  parent = ele.getAttribute(PARENT_ATTRIBUTE);
87              }
88              //根据<Bean>元素配置的class名称和parent属性值创建BeanDefinition
89              //为载入Bean定义信息做准备
90              AbstractBeanDefinition bd = createBeanDefinition(className,
parent);
91              //对当前的<Bean>元素中配置的一些属性进行解析和设置，如配置的单态(singleton)
属性等
92              parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);

93              //为<Bean>元素解析的Bean设置description信息
bd.setDescription(DomUtils.getChildElementValueByTagName(ele,
DESCRIPTION_ELEMENT));
94              //对<Bean>元素的meta(元信息)属性解析
95              parseMetaElements(ele, bd);
96              //对<Bean>元素的lookup-method属性解析
97              parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
98              //对<Bean>元素的replaced-method属性解析
99              parseReplacedMethodSubElements(ele, bd.getMethodOverrides());
100              //解析<Bean>元素的构造方法设置
101              parseConstructorArgElements(ele, bd);
102              //解析<Bean>元素的<property>设置
103              parsePropertyElements(ele, bd);
104              //解析<Bean>元素的qualifier属性
105              parseQualifierElements(ele, bd);
106              //为当前解析的Bean设置所需的资源和依赖对象
```

```
107             bd.setResource(this.readerContext.getResource());
108             bd.setSource(extractSource(ele));
109             return bd;
110         }
111         catch (ClassNotFoundException ex) {
112             error("Bean class [" + className + "] not found", ele, ex);
113         }
114         catch (NoClassDefFoundError err) {
115             error("Class that bean class [" + className + "] depends on not
found", ele, err);
116         }
117         catch (Throwable ex) {
118             error("Unexpected failure during bean definition parsing", ele,
ex);
119         }
120         finally {
121             this.parseState.pop();
122         }
123         //解析<Bean>元素出错时，返回null
124         return null;
125     }
```

只要使用过Spring，对Spring配置文件比较熟悉的人，通过对上述源码的分析，就会明白我们在Spring配置文件中元素的中配置的属性就是通过该方法解析和设置到Bean中去的。

注意：在解析元素过程中没有创建和实例化Bean对象，只是创建了Bean对象的定义类BeanDefinition，将元素中的配置信息设置到BeanDefinition中作为记录，当依赖注入时才使用这些记录信息创建和实例化具体的Bean对象

上面方法中一些对一些配置如元信息(meta)、qualifier等的解析，我们在Spring中配置时使用的也不多，我们在使用Spring的元素时，配置最多的是属性，因此我们下面继续分析源码，了解Bean的属性在解析时是如何设置的。

13、BeanDefinitionParserDelegate解析元素：

DefinitionParserDelegate在解析调用parsePropertyElements方法解析元素中的属性子元素，解析源码如下：

```
1     //解析<Bean>元素中的<property>子元素
2     public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
3         //获取<Bean>元素中所有的子元素
4         NodeList nl = beanEle.getChildNodes();
5         for (int i = 0; i < nl.getLength(); i++) {
6             Node node = nl.item(i);
7             //如果子元素是<property>子元素，则调用解析<property>子元素方法解析
8             if (isCandidateElement(node) && nodeNameEquals(node,
PROPERTY_ELEMENT)) {
9                 parsePropertyElement((Element) node, bd);
10             }
11         }
12     }
13     //解析<property>元素
```

```
14    public void parsePropertyElement(Element ele, BeanDefinition bd) {
15        //获取<property>元素的名字
16        String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
17        if (!StringUtils.hasLength(propertyName)) {
18            error("Tag 'property' must have a 'name' attribute", ele);
19            return;
20        }
21        this.parseState.push(new PropertyEntry(propertyName));
22        try {
23            //如果一个Bean中已经有同名的property存在，则不进行解析，直接返回。
24            //即如果在同一个Bean中配置同名的property，则只有第一个起作用
25            if (bd.getPropertyValues().contains(propertyName)) {
26                error("Multiple 'property' definitions for property '" +
propertyName + "'", ele);
27                return;
28            }
29            //解析获取property的值
30            Object val = parsePropertyValue(ele, bd, propertyName);
31            //根据property的名字和值创建property实例
32            PropertyValue pv = new PropertyValue(propertyName, val);
33            //解析<property>元素中的属性
34            parseMetaElements(ele, pv);
35            pv.setSource(extractSource(ele));
36            bd.getPropertyValues().addPropertyValue(pv);
37        }
38        finally {
39            this.parseState.pop();
40        }
41    }
42    //解析获取property值
43    public Object parsePropertyValue(Element ele, BeanDefinition bd, String
propertyName) {
44        String elementName = (propertyName != null) ?
45                        "<property> element for property '" + propertyName +
"'" :
46                        "<constructor-arg> element";
47        //获取<property>的所有子元素，只能是其中一种类型:ref,value,list等
48        NodeList nl = ele.getChildNodes();
49        Element subElement = null;
50        for (int i = 0; i < nl.getLength(); i++) {
51            Node node = nl.item(i);
52            //子元素不是description和meta属性
53            if (node instanceof Element && !nodeNameEquals(node,
DESCRIPTION_ELEMENT) &&
54                    !nodeNameEquals(node, META_ELEMENT)) {
55                if (subElement != null) {
56                    error(elementName + " must not contain more than one sub-
element", ele);
57                }
58                else {//当前<property>元素包含有子元素
59                    subElement = (Element) node;
60                }
61            }
62        }
63        //判断property的属性值是ref还是value，不允许既是ref又是value
64        boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
65        boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
66        if ((hasRefAttribute && hasValueAttribute) ||
```

```
67                ((hasRefAttribute || hasValueAttribute) && subElement != null))
{
68            error(elementName +
69                    " is only allowed to contain either 'ref' attribute OR
'value' attribute OR sub-element", ele);
70        }
71        //如果属性是ref，创建一个ref的数据对象RuntimeBeanReference，这个对象
72        //封装了ref信息
73        if (hasRefAttribute) {
74            String refName = ele.getAttribute(REF_ATTRIBUTE);
75            if (!StringUtils.hasText(refName)) {
76                error(elementName + " contains empty 'ref' attribute", ele);
77            }
78            //一个指向运行时所依赖对象的引用
79            RuntimeBeanReference ref = new RuntimeBeanReference(refName);
80            //设置这个ref的数据对象是被当前的property对象所引用
81            ref.setSource(extractSource(ele));
82            return ref;
83        }
84         //如果属性是value，创建一个value的数据对象TypedStringValue，这个对象
85        //封装了value信息
86        else if (hasValueAttribute) {
87            //一个持有String类型值的对象
88            TypedStringValue valueHolder = new
TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE));
89            //设置这个value数据对象是被当前的property对象所引用
90            valueHolder.setSource(extractSource(ele));
91            return valueHolder;
92        }
93        //如果当前<property>元素还有子元素
94        else if (subElement != null) {
95            //解析<property>的子元素
96            return parsePropertySubElement(subElement, bd);
97        }
98        else {
99            //propery属性中既不是ref，也不是value属性，解析出错返回null
error(elementName + " must specify a ref or value", ele);
100            return null;
101        }
    }
```

通过对上述源码的分析，我们可以了解在Spring配置文件中，元素中元素的相关配置是如何处理的：

a. ref被封装为指向依赖对象一个引用。

b.value配置都会封装成一个字符串类型的对象。

c.ref和value都通过“解析的数据类型属性值.setSource(extractSource(ele));”方法将属性值/引用与所引用的属性关联起来。

在方法的最后对于元素的子元素通过parsePropertySubElement 方法解析，我们继续分析该方法的源码，了解其解析过程。

14、解析元素的子元素：

在BeanDefinitionParserDelegate类中的parsePropertySubElement方法对中的子元素解析，源码如下：

```
1    //解析<property>元素中ref,value或者集合等子元素
2    public Object parsePropertySubElement(Element ele, BeanDefinition bd, String
defaultValueType) {
3        //如果<property>没有使用Spring默认的命名空间，则使用用户自定义的规则解析//内嵌元素

4        if (!isDefaultNamespace(ele)) {
5            return parseNestedCustomElement(ele, bd);
6        }
7        //如果子元素是bean，则使用解析<Bean>元素的方法解析
8        else if (nodeNameEquals(ele, BEAN_ELEMENT)) {
9            BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);

10            if (nestedBd != null) {
11                nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);

12            }
13            return nestedBd;
14        }
15        //如果子元素是ref，ref中只能有以下3个属性：bean、local、parent
16        else if (nodeNameEquals(ele, REF_ELEMENT)) {
17            //获取<property>元素中的bean属性值，引用其他解析的Bean的名称
18            //可以不再同一个Spring配置文件中，具体请参考Spring对ref的配置规则
19            String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
20            boolean toParent = false;
21            if (!StringUtils.hasLength(refName)) {
22                //获取<property>元素中的local属性值，引用同一个Xml文件中配置
23                //的Bean的id，local和ref不同，local只能引用同一个配置文件中的Bean
24                refName = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
25                if (!StringUtils.hasLength(refName)) {
26                    //获取<property>元素中parent属性值，引用父级容器中的Bean
27                    refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
28                    toParent = true;
29                    if (!StringUtils.hasLength(refName)) {
30                        error("'bean', 'local' or 'parent' is required for
<ref> element", ele);
31                        return null;
32                    }
33                }
34            }
35            //没有配置ref的目标属性值
36            if (!StringUtils.hasText(refName)) {
37                error("<ref> element contains empty target attribute", ele);
38                return null;
39            }
40            //创建ref类型数据，指向被引用的对象
41            RuntimeBeanReference ref = new RuntimeBeanReference(refName,
toParent);
42            //设置引用类型值是被当前子元素所引用
43            ref.setSource(extractSource(ele));
44            return ref;
45        }
46        //如果子元素是<idref>，使用解析ref元素的方法解析
```

```
47          else if (nodeNameEquals(ele, IDREF_ELEMENT)) {
48              return parseIdRefElement(ele);
49          }
50          //如果子元素是<value>，使用解析value元素的方法解析
51          else if (nodeNameEquals(ele, VALUE_ELEMENT)) {
52              return parseValueElement(ele, defaultValueType);
53          }
54          //如果子元素是null，为<property>设置一个封装null值的字符串数据
55          else if (nodeNameEquals(ele, NULL_ELEMENT)) {
56              TypedStringValue nullHolder = new TypedStringValue(null);
57              nullHolder.setSource(extractSource(ele));
58              return nullHolder;
59          }
60          //如果子元素是<array>，使用解析array集合子元素的方法解析
61          else if (nodeNameEquals(ele, ARRAY_ELEMENT)) {
62              return parseArrayElement(ele, bd);
63          }
64          //如果子元素是<list>，使用解析list集合子元素的方法解析
65          else if (nodeNameEquals(ele, LIST_ELEMENT)) {
66              return parseListElement(ele, bd);
67          }
68          //如果子元素是<set>，使用解析set集合子元素的方法解析
69          else if (nodeNameEquals(ele, SET_ELEMENT)) {
70              return parseSetElement(ele, bd);
71          }
72          //如果子元素是<map>，使用解析map集合子元素的方法解析
73          else if (nodeNameEquals(ele, MAP_ELEMENT)) {
74              return parseMapElement(ele, bd);
75          }
76          //如果子元素是<props>，使用解析props集合子元素的方法解析
77          else if (nodeNameEquals(ele, PROPS_ELEMENT)) {
78              return parsePropsElement(ele);
79          }
80          //既不是ref，又不是value，也不是集合，则子元素配置错误，返回null
81          else {
82              error("Unknown property sub-element: [" + ele.getNodeName() + "]",
ele);
83              return null;
84          }
    }
```

通过上述源码分析，我们明白了在Spring配置文件中，对元素中配置的Array、List、Set、Map、Prop等各种集合子元素的都通过上述方法解析，生成对应的数据对象，比如ManagedList、ManagedArray、ManagedSet等，这些Managed类是Spring对象BeanDefiniton的数据封装，对集合数据类型的具体解析有各自的解析方法实现，解析方法的命名非常规范，一目了然，我们对集合元素的解析方法进行源码分析，了解其实现过程。

15、解析子元素：

在BeanDefinitionParserDelegate类中的parseListElement方法就是具体实现解析元素中的集合子元素，源码如下：

```
    //解析<list>集合子元素
    public List parseListElement(Element collectionEle, BeanDefinition bd) {
        //获取<list>元素中的value-type属性，即获取集合元素的数据类型
        String defaultElementType =
collectionEle.getAttribute(VALUE_TYPE_ATTRIBUTE);
        //获取<list>集合元素中的所有子节点
        NodeList nl = collectionEle.getChildNodes();
        //Spring中将List封装为ManagedList
        ManagedList<Object> target = new ManagedList<Object>(nl.getLength());
        target.setSource(extractSource(collectionEle));
         //设置集合目标数据类型
         target.setElementTypeName(defaultElementType);
         target.setMergeEnabled(parseMergeAttribute(collectionEle));
         //具体的<list>元素解析
         parseCollectionElements(nl, target, bd, defaultElementType);
         return target;
    }
    //具体解析<list>集合元素，<array>、<list>和<set>都使用该方法解析
    protected void parseCollectionElements(
            NodeList elementNodes, Collection<Object> target, BeanDefinition
bd, String defaultElementType) {
        //遍历集合所有节点
        for (int i = 0; i < elementNodes.getLength(); i++) {
            Node node = elementNodes.item(i);
            //节点不是description节点
            if (node instanceof Element && !nodeNameEquals(node,
DESCRIPTION_ELEMENT)) {
                //将解析的元素加入集合中，递归调用下一个子元素
                target.add(parsePropertySubElement((Element) node, bd,
defaultElementType));
            }
        }
    }
```

经过对Spring Bean定义资源文件转换的Document对象中的元素层层解析，Spring IoC现在已经将XML形式定义的Bean定义资源文件转换为Spring IoC所识别的数据结构——BeanDefinition，它是Bean定义资源文件中配置的POJO对象在Spring IoC容器中的映射，我们可以通过AbstractBeanDefinition为入口，荣IoC容器进行索引、查询和操作。

通过Spring IoC容器对Bean定义资源的解析后，IoC容器大致完成了管理Bean对象的准备工作，即初始化过程，但是最为重要的依赖注入还没有发生，现在在IoC容器中BeanDefinition存储的只是一些静态信息，接下来需要向容器注册Bean定义信息才能全部完成IoC容器的初始化过程

16、解析过后的BeanDefinition在IoC容器中的注册：

让我们继续跟踪程序的执行顺序，接下来会到我们第3步中分析DefaultBeanDefinitionDocumentReader对Bean定义转换的Document对象解析的流程中，在其parseDefaultElement方法中完成对Document对象的解析后得到封装BeanDefinition的BeanDefinitionHold对象，然后调用BeanDefinitionReaderUtils的registerBeanDefinition方法向IoC容器注册解析的Bean，BeanDefinitionReaderUtils的注册的源码如下：

```
//将解析的BeanDefinitionHold注册到容器中
public static void registerBeanDefinition(BeanDefinitionHolder definitionHolder,
BeanDefinitionRegistry registry)
```

```
    throws BeanDefinitionStoreException {
        //获取解析的BeanDefinition的名称
         String beanName = definitionHolder.getBeanName();
        //向IoC容器注册BeanDefinition
        registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());
        //如果解析的BeanDefinition有别名，向容器为其注册别名
         String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String aliase : aliases) {
                registry.registerAlias(beanName, aliase);
            }
        }
}
```

当调用BeanDefinitionReaderUtils向IoC容器注册解析的BeanDefinition时，真正完成注册功能的是DefaultListableBeanFactory。

17、DefaultListableBeanFactory向IoC容器注册解析后的BeanDefinition：

DefaultListableBeanFactory中使用一个HashMap的集合对象存放IoC容器中注册解析的BeanDefinition，向IoC容器注册的主要源码如下：




```
1    //存储注册的俄BeanDefinition
2    private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<String, BeanDefinition>();
3    //向IoC容器注册解析的BeanDefiniton
4    public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
5            throws BeanDefinitionStoreException {
6        Assert.hasText(beanName, "Bean name must not be empty");
7        Assert.notNull(beanDefinition, "BeanDefinition must not be null");
8        //校验解析的BeanDefiniton
9        if (beanDefinition instanceof AbstractBeanDefinition) {
10            try {
11                ((AbstractBeanDefinition) beanDefinition).validate();
12            }
13            catch (BeanDefinitionValidationException ex) {
14                throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
```

```
15                          "Validation of bean definition failed", ex);
16              }
17          }
18          //注册的过程中需要线程同步，以保证数据的一致性
19          synchronized (this.beanDefinitionMap) {
20              Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
21              //检查是否有同名的BeanDefinition已经在IoC容器中注册，如果已经注册，
22              //并且不允许覆盖已注册的Bean，则抛出注册失败异常
23              if (oldBeanDefinition != null) {
24                  if (!this.allowBeanDefinitionOverriding) {
25                      throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,

26                              "Cannot register bean definition [" +
beanDefinition + "] for bean '" + beanName +
27                              "': There is already [" + oldBeanDefinition + "]
bound.");
28                  }
29                  else {//如果允许覆盖，则同名的Bean，后注册的覆盖先注册的
30                      if (this.logger.isInfoEnabled()) {
31                          this.logger.info("Overriding bean definition for bean
'" + beanName +
32                                  "': replacing [" + oldBeanDefinition + "] with
[" + beanDefinition + "]");
33                      }
34                  }
35              }
36              //IoC容器中没有已经注册同名的Bean，按正常注册流程注册
37              else {
38                  this.beanDefinitionNames.add(beanName);
39                  this.frozenBeanDefinitionNames = null;
40              }
41              this.beanDefinitionMap.put(beanName, beanDefinition);
42              //重置所有已经注册过的BeanDefinition的缓存
43              resetBeanDefinition(beanName);
44          }
    }
```

至此，Bean定义资源文件中配置的Bean被解析过后，已经注册到IoC容器中，被容器管理起来，真正完成了IoC容器初始化所做的全部工作。现 在IoC容器中已经建立了整个Bean的配置信息，这些BeanDefinition信息已经可以使用，并且可以被检索，IoC容器的作用就是对这些注册的Bean定义信息进行处理和维护。这些的注册的Bean定义信息是IoC容器控制反转的基础，正是有了这些注册的数据，容器才可以进行依赖注入。

总结：

现在通过上面的代码，总结一下IOC容器初始化的基本步骤：

u 初始化的入口在容器实现中的 refresh()调用来完成

u 对 bean 定义载入 IOC 容器使用的方法是 loadBeanDefinition,其中的大致过程如下：通过 ResourceLoader 来完成资源文件位置的定位，DefaultResourceLoader 是默认的实现，同时上下文本身就给出了 ResourceLoader 的实现，可以从类路径，文件系统, URL 等方式来定为资源位置。如果是 XmlBeanFactory作为 IOC 容器，那么需要为它指定 bean 定义的资源，也就是说 bean 定义文件时通过抽象成 Resource 来被 IOC 容器处理的，容器通过 BeanDefinitionReader来完成定义信息的解析和 Bean 信息的注册,往往使用的是XmlBeanDefinitionReader 来解析 bean 的 xml 定义文件 - 实际的处理过程是委托给 BeanDefinitionParserDelegate 来完成的，从而得到 bean 的定义信息，这些信息在 Spring 中使用 BeanDefinition 对象来表示 - 这个名字可以让我们想到 loadBeanDefinition,RegisterBeanDefinition 这些相关的方法 - 他们都是为处理 BeanDefinitin 服务的， 容器解析得到 BeanDefinitionIoC 以后，需要把它在 IOC 容器中注册，这由 IOC 实现 BeanDefinitionRegistry 接口来实现。注册过程就是在 IOC 容器内部维护的一个HashMap 来保存得到的 BeanDefinition 的过程。这个 HashMap 是 IoC 容器持有 bean 信息的场所，以后对 bean 的操作都是围绕这个HashMap 来实现的.

u 然后我们就可以通过 BeanFactory 和 ApplicationContext 来享受到 Spring IOC 的服务了,在使用 IOC 容器的时候，我们注意到除了少量粘合代码，绝大多数以正确 IoC 风格编写的应用程序代码完全不用关心如何到达工厂，因为容器将把这些对象与容器管理的其他对象钩在一起。基本的策略是把工厂放到已知的地方，最好是放在对预期使用的上下文有意义的地方，以及代码将实际需要访问工厂的地方。Spring 本身提供了对声明式载入 web 应用程序用法的应用程序上下文,并将其存储在ServletContext 中的框架实现。具体可以参见以后的文章

在使用 Spring IOC 容器的时候我们还需要区别两个概念:

> Beanfactory 和 Factory bean，其中 BeanFactory 指的是 IOC 容器的编程抽象，比如 ApplicationContext， XmlBeanFactory 等，这些都是 IOC 容器的具体表现，需要使用什么样的容器由客户决定,但 Spring 为我们提供了丰富的选择。 FactoryBean 只是一个可以在 IOC而容器中被管理的一个 bean,是对各种处理过程和资源使用的抽象,Factory bean 在需要时产生另一个对象，而不返回 FactoryBean本身,我们可以把它看成是一个抽象工厂，对它的调用返回的是工厂生产的产品。所有的 Factory bean 都实现特殊的org.springframework.beans.factory.FactoryBean 接口，当使用容器中 factory bean 的时候，该容器不会返回 factory bean 本身,而是返回其生成的对象。Spring 包括了大部分的通用资源和服务访问抽象的 Factory bean 的实现，其中包括:对 JNDI 查询的处理，对代理对象的处理，对事务性代理的处理，对 RMI 代理的处理等，这些我们都可以看成是具体的工厂,看成是 SPRING 为我们建立好的工厂。也就是说 Spring 通过使用抽象工厂模式为我们准备了一系列工厂来生产一些特定的对象,免除我们手工重复的工作，我们要使用时只需要在 IOC 容器里配置好就能很方便的使用了

四、IOC容器的依赖注入

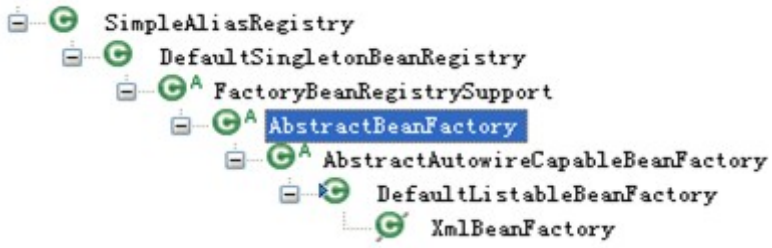
1、依赖注入发生的时间

当Spring IoC容器完成了Bean定义资源的定位、载入和解析注册以后，IoC容器中已经管理类Bean定义的相关数据，但是此时IoC容器还没有对所管理的Bean进行依赖注入，依赖注入在以下两种情况发生：

(1).用户第一次通过getBean方法向IoC容索要Bean时，IoC容器触发依赖注入。

(2).当用户在Bean定义资源中为元素配置了lazy-init属性，即让容器在解析注册Bean定义时进行预实例化，触发依赖注入。

BeanFactory接口定义了Spring IoC容器的基本功能规范，是Spring IoC容器所应遵守的最底层和最基本的编程规范。BeanFactory接口中定义了几个getBean方法，就是用户向IoC容器索取管理的Bean的方法，我们通过分析其子类的具体实现，理解Spring IoC容器在用户索取Bean时如何完成依赖注入。



在BeanFactory中我们看到getBean（String...）函数，它的具体实现在AbstractBeanFactory中

2、AbstractBeanFactory通过getBean向IoC容器获取被管理的Bean：

AbstractBeanFactory的getBean相关方法的源码如下：

```
1    //获取IoC容器中指定名称的Bean
2    public Object getBean(String name) throws BeansException {
3        //doGetBean才是真正向IoC容器获取被管理Bean的过程
4        return doGetBean(name, null, null, false);
5    }
6    //获取IoC容器中指定名称和类型的Bean
7    public <T> T getBean(String name, Class<T> requiredType) throws
BeansException {
8        //doGetBean才是真正向IoC容器获取被管理Bean的过程
9        return doGetBean(name, requiredType, null, false);
10    }
11    //获取IoC容器中指定名称和参数的Bean
12    public Object getBean(String name, Object... args) throws BeansException {

13        //doGetBean才是真正向IoC容器获取被管理Bean的过程
14        return doGetBean(name, null, args, false);
15    }
16    //获取IoC容器中指定名称、类型和参数的Bean
17    public <T> T getBean(String name, Class<T> requiredType, Object... args)
throws BeansException {
```

```
18    //doGetBean才是真正向IoC容器获取被管理Bean的过程
19        return doGetBean(name, requiredType, args, false);
20    }
21    //真正实现向IoC容器获取Bean的功能，也是触发依赖注入功能的地方
22    @SuppressWarnings("unchecked")
23    protected <T> T doGetBean(
24            final String name, final Class<T> requiredType, final Object[]
args, boolean typeCheckOnly)
25            throws BeansException {
26        //根据指定的名称获取被管理Bean的名称，剥离指定名称中对容器的相关依赖
27        //如果指定的是别名，将别名转换为规范的Bean名称
28        final String beanName = transformedBeanName(name);
29        Object bean;
30        //先从缓存中取是否已经有被创建过的单态类型的Bean，对于单态模式的Bean整
31        //个IoC容器中只创建一次，不需要重复创建
32        Object sharedInstance = getSingleton(beanName);
33        //IoC容器创建单态模式Bean实例对象
34        if (sharedInstance != null && args == null) {
35            if (logger.isDebugEnabled()) {
36                //如果指定名称的Bean在容器中已有单态模式的Bean被创建，直接返回
37                //已经创建的Bean
38                if (isSingletonCurrentlyInCreation(beanName)) {
39                    logger.debug("Returning eagerly cached instance of
singleton bean '" + beanName +
40                            "' that is not fully initialized yet - a
consequence of a circular reference");
41                }
42                else {
43                    logger.debug("Returning cached instance of singleton bean
'" + beanName + "'");
44                }
45            }
46            //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
47            //注意：BeanFactory是管理容器中Bean的工厂，而FactoryBean是
48            //创建创建对象的工厂Bean，两者之间有区别
49            bean = getObjectForBeanInstance(sharedInstance, name, beanName,
null);
50        }
51        else {//缓存没有正在创建的单态模式Bean
52            //缓存中已经有已经创建的原型模式Bean，但是由于循环引用的问题导致实
53            //例化对象失败
54            if (isPrototypeCurrentlyInCreation(beanName)) {
55                throw new BeanCurrentlyInCreationException(beanName);
56            }
57            //对IoC容器中是否存在指定名称的BeanDefinition进行检查，首先检查是否
58            //能在当前的BeanFactory中获取的所需要的Bean，如果不能则委托当前容器
59            //的父级容器去查找，如果还是找不到则沿着容器的继承体系向父级容器查找
60            BeanFactory parentBeanFactory = getParentBeanFactory();
61            //当前容器的父级容器存在，且当前容器中不存在指定名称的Bean
62            if (parentBeanFactory != null && !containsBeanDefinition(beanName))
{
63                //解析指定Bean名称的原始名称
64                String nameToLookup = originalBeanName(name);
65                if (args != null) {
66                    //委派父级容器根据指定名称和显式的参数查找
67                    return (T) parentBeanFactory.getBean(nameToLookup, args);

68                }
```

```
69                 else {
70                     //委派父级容器根据指定名称和类型查找
71                     return parentBeanFactory.getBean(nameToLookup,
requiredType);
72                 }
73             }
74             //创建的Bean是否需要进行类型验证，一般不需要
75             if (!typeCheckOnly) {
76                 //向容器标记指定的Bean已经被创建
77                 markBeanAsCreated(beanName);
78             }
79              //根据指定Bean名称获取其父级的Bean定义，主要解决Bean继承时子类
80             //合并父类公共属性问题
81             final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);
82             checkMergedBeanDefinition(mbd, beanName, args);
83             //获取当前Bean所有依赖Bean的名称
84             String[] dependsOn = mbd.getDependsOn();
85             //如果当前Bean有依赖Bean
86             if (dependsOn != null) {
87                 for (String dependsOnBean : dependsOn) {
88                     //递归调用getBean方法，获取当前Bean的依赖Bean
89                     getBean(dependsOnBean);
90                     //把被依赖Bean注册给当前依赖的Bean
91                     registerDependentBean(dependsOnBean, beanName);
92                 }
93             }
94             //创建单态模式Bean的实例对象
95             if (mbd.isSingleton()) {
96             //这里使用了一个匿名内部类，创建Bean实例对象，并且注册给所依赖的对象
97                 sharedInstance = getSingleton(beanName, new ObjectFactory() {

98                     public Object getObject() throws BeansException {
99                         try {
100                             //创建一个指定Bean实例对象，如果有父级继承，则合并子//类和
父类的定义
101                             return createBean(beanName, mbd, args);
102                          }
103                          catch (BeansException ex) {
104                             //显式地从容器单态模式Bean缓存中清除实例对象
105                             destroySingleton(beanName);
106                             throw ex;
107                          }
108                      }
109                  });
110                  //获取给定Bean的实例对象
111                  bean = getObjectForBeanInstance(sharedInstance, name,
beanName, mbd);
112              }
113              //IoC容器创建原型模式Bean实例对象
114              else if (mbd.isPrototype()) {
115                  //原型模式(Prototype)是每次都会创建一个新的对象
116                  Object prototypeInstance = null;
117                  try {
118                      //回调beforePrototypeCreation方法，默认的功能是注册当前创//建的
原型对象
119                      beforePrototypeCreation(beanName);
120                      //创建指定Bean对象实例
```

```
121                     prototypeInstance = createBean(beanName, mbd, args);
122                 }
123                 finally {
124                     //回调afterPrototypeCreation方法，默认的功能告诉IoC容器指//定
Bean的原型对象不再创建了
125                     afterPrototypeCreation(beanName);
126                 }
127                 //获取给定Bean的实例对象
128                 bean = getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
129             }
130             //要创建的Bean既不是单态模式，也不是原型模式，则根据Bean定义资源中
131             //配置的生命周期范围，选择实例化Bean的合适方法，这种在Web应用程序中
132             //比较常用，如：request、session、application等生命周期
133             else {
134                 String scopeName = mbd.getScope();
135                 final Scope scope = this.scopes.get(scopeName);
136                 //Bean定义资源中没有配置生命周期范围，则Bean定义不合法
137                 if (scope == null) {
138                     throw new IllegalStateException("No Scope registered for
scope '" + scopeName + "'");
139                 }
140                 try {
141                     //这里又使用了一个匿名内部类，获取一个指定生命周期范围的实例
142                     Object scopedInstance = scope.get(beanName, new
ObjectFactory() {
143                         public Object getObject() throws BeansException {
144                             beforePrototypeCreation(beanName);
145                             try {
146                                 return createBean(beanName, mbd, args);
147                             }
148                             finally {
149                                 afterPrototypeCreation(beanName);
150                             }
151                         }
152                     });
153                     //获取给定Bean的实例对象
154                     bean = getObjectForBeanInstance(scopedInstance, name,
beanName, mbd);
155                 }
156                 catch (IllegalStateException ex) {
157                     throw new BeanCreationException(beanName,
158                             "Scope '" + scopeName + "' is not active for the
current thread; " +
159                             "consider defining a scoped proxy for this bean if
you intend to refer to it from a singleton",
160                             ex);
161                 }
162             }
163         }
164         //对创建的Bean实例对象进行类型检查
165         if (requiredType != null && bean != null &&
!requiredType.isAssignableFrom(bean.getClass())) {
166             throw new BeanNotOfRequiredTypeException(name, requiredType,
bean.getClass());
167         }
168         return (T) bean;
169     }
```

通过上面对向IoC容器获取Bean方法的分析，我们可以看到在Spring中，如果Bean定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果Bean定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。

上面的源码只是定义了根据Bean定义的模式，采取的不同创建Bean实例对象的策略，具体的Bean实例对象的创建过程由实现了ObejctFactory接口的匿名内部类的createBean方法完成，ObejctFactory使用委派模式，具体的Bean实例创建过程交由其实现类AbstractAutowireCapableBeanFactory完成，我们继续分析AbstractAutowireCapableBeanFactory的createBean方法的源码，理解其创建Bean实例的具体实现过程。

3、AbstractAutowireCapableBeanFactory创建Bean实例对象：

AbstractAutowireCapableBeanFactory类实现了ObejctFactory接口，创建容器指定的Bean实例对象，同时还对创建的Bean实例对象进行初始化处理。其创建Bean实例对象的方法源码如下：

```
1    //创建Bean实例对象
2    protected Object createBean(final String beanName, final RootBeanDefinition
mbd, final Object[] args)
3            throws BeanCreationException {
4        if (logger.isDebugEnabled()) {
5            logger.debug("Creating instance of bean '" + beanName + "'");
6        }
7        //判断需要创建的Bean是否可以实例化，即是否可以通过当前的类加载器加载
8        resolveBeanClass(mbd, beanName);
9        //校验和准备Bean中的方法覆盖
10        try {
11            mbd.prepareMethodOverrides();
12        }
13        catch (BeanDefinitionValidationException ex) {
14            throw new
BeanDefinitionStoreException(mbd.getResourceDescription(),
15                    beanName, "Validation of method overrides failed", ex);
16        }
17        try {
18            //如果Bean配置了初始化前和初始化后的处理器，则试图返回一个需要创建//Bean的代理
对象
19            Object bean = resolveBeforeInstantiation(beanName, mbd);
20            if (bean != null) {
```

```
21                 return bean;
22             }
23         }
24         catch (Throwable ex) {
25             throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
26                     "BeanPostProcessor before instantiation of bean failed",
ex);
27         }
28         //创建Bean的入口
29         Object beanInstance = doCreateBean(beanName, mbd, args);
30         if (logger.isDebugEnabled()) {
31             logger.debug("Finished creating instance of bean '" + beanName +
"'");
32         }
33         return beanInstance;
34     }
35     //真正创建Bean的方法
36     protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final Object[] args) {
37         //封装被创建的Bean对象
38         BeanWrapper instanceWrapper = null;
39         if (mbd.isSingleton()){//单态模式的Bean，先从容器中缓存中获取同名Bean
40             instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);

41         }
42         if (instanceWrapper == null) {
43             //创建实例对象
44             instanceWrapper = createBeanInstance(beanName, mbd, args);
45         }
46         final Object bean = (instanceWrapper != null ?
instanceWrapper.getWrappedInstance() : null);
47         //获取实例化对象的类型
48         Class beanType = (instanceWrapper != null ?
instanceWrapper.getWrappedClass() : null);
49         //调用PostProcessor后置处理器
50         synchronized (mbd.postProcessingLock) {
51             if (!mbd.postProcessed) {
52                 applyMergedBeanDefinitionPostProcessors(mbd, beanType,
beanName);
53                 mbd.postProcessed = true;
54             }
55         }
56         // Eagerly cache singletons to be able to resolve circular references

57         //向容器中缓存单态模式的Bean对象，以防循环引用
58         boolean earlySingletonExposure = (mbd.isSingleton() &&
this.allowCircularReferences &&
59                 isSingletonCurrentlyInCreation(beanName));
60         if (earlySingletonExposure) {
61             if (logger.isDebugEnabled()) {
62                 logger.debug("Eagerly caching bean '" + beanName +
63                         "' to allow for resolving potential circular
references");
64             }
65             //这里是一个匿名内部类，为了防止循环引用，尽早持有对象的引用
66             addSingletonFactory(beanName, new ObjectFactory() {
67                 public Object getObject() throws BeansException {
```

```
68                      return getEarlyBeanReference(beanName, mbd, bean);
69                  }
70              });
71          }
72          //Bean对象的初始化，依赖注入在此触发
73          //这个exposedObject在初始化完成之后返回作为依赖注入完成后的Bean
74          Object exposedObject = bean;
75          try {
76              //将Bean实例对象封装，并且Bean定义中配置的属性值赋值给实例对象
77              populateBean(beanName, mbd, instanceWrapper);
78              if (exposedObject != null) {
79                  //初始化Bean对象
80                  exposedObject = initializeBean(beanName, exposedObject, mbd);

81              }
82          }
83          catch (Throwable ex) {
84              if (ex instanceof BeanCreationException &&
beanName.equals(((BeanCreationException) ex).getBeanName())) {
85                  throw (BeanCreationException) ex;
86              }
87              else {
88                  throw new BeanCreationException(mbd.getResourceDescription(),
beanName, "Initialization of bean failed", ex);
89              }
90          }
91          if (earlySingletonExposure) {
92              //获取指定名称的已注册的单态模式Bean对象
93              Object earlySingletonReference = getSingleton(beanName, false);
94              if (earlySingletonReference != null) {
95                  //根据名称获取的以注册的Bean和正在实例化的Bean是同一个
96                  if (exposedObject == bean) {
97                      //当前实例化的Bean初始化完成
98                      exposedObject = earlySingletonReference;
99                  }
100                  //当前Bean依赖其他Bean，并且当发生循环引用时不允许新创建实例对象
101                  else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
102                      String[] dependentBeans = getDependentBeans(beanName);
103                      Set<String> actualDependentBeans = new
LinkedHashSet<String>(dependentBeans.length);
104                      //获取当前Bean所依赖的其他Bean
105                      for (String dependentBean : dependentBeans) {
106                          //对依赖Bean进行类型检查
107                          if
(!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
108                              actualDependentBeans.add(dependentBean);
109                          }
110                      }
111                      if (!actualDependentBeans.isEmpty()) {
112                          throw new BeanCurrentlyInCreationException(beanName,

113                                  "Bean with name '" + beanName + "' has been
injected into other beans [" +
114
StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
115                                  "] in its raw version as part of a circular
reference, but has eventually been " +
```

```
116                                 "wrapped. This means that said other beans do
not use the final version of the " +
117                                 "bean. This is often the result of over-eager
type matching - consider using " +
118                                 "'getBeanNamesOfType' with the
'allowEagerInit' flag turned off, for example.");
119                     }
120                 }
121             }
122         }
123         //注册完成依赖注入的Bean
124         try {
125             registerDisposableBeanIfNecessary(beanName, bean, mbd);
126         }
127         catch (BeanDefinitionValidationException ex) {
128             throw new BeanCreationException(mbd.getResourceDescription(),
beanName, "Invalid destruction signature", ex);
129         }
130         return exposedObject;
    }
```

通过对方法源码的分析，我们看到具体的依赖注入实现在以下两个方法中：

(1).createBeanInstance：生成Bean所包含的java对象实例。

(2).populateBean ： 对Bean属性的依赖注入进行处理。

下面继续分析这两个方法的代码实现。

4、createBeanInstance方法创建Bean的java实例对象：

在createBeanInstance方法中，根据指定的初始化策略，使用静态工厂、工厂方法或者容器的自动装配特性生成java实例对象，创建对象的源码如下：

```
1    //创建Bean的实例对象
2    protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition
mbd, Object[] args) {
3        //检查确认Bean是可实例化的
4        Class beanClass = resolveBeanClass(mbd, beanName);
5        //使用工厂方法对Bean进行实例化
6        if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) &&
!mbd.isNonPublicAccessAllowed()) {
7            throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
8                    "Bean class isn't public, and non-public access not allowed:
" + beanClass.getName());
9        }
10        if (mbd.getFactoryMethodName() != null)  {
```

```
11              //调用工厂方法实例化
12              return instantiateUsingFactoryMethod(beanName, mbd, args);
13          }
14          //使用容器的自动装配方法进行实例化
15          boolean resolved = false;
16          boolean autowireNecessary = false;
17          if (args == null) {
18              synchronized (mbd.constructorArgumentLock) {
19                  if (mbd.resolvedConstructorOrFactoryMethod != null) {
20                      resolved = true;
21                      autowireNecessary = mbd.constructorArgumentsResolved;
22                  }
23              }
24          }
25          if (resolved) {
26              if (autowireNecessary) {
27                  //配置了自动装配属性，使用容器的自动装配实例化
28                  //容器的自动装配是根据参数类型匹配Bean的构造方法
29                  return autowireConstructor(beanName, mbd, null, null);
30              }
31              else {
32                  //使用默认的无参构造方法实例化
33                  return instantiateBean(beanName, mbd);
34              }
35          }
36          //使用Bean的构造方法进行实例化
37          Constructor[] ctors =
determineConstructorsFromBeanPostProcessors(beanClass, beanName);
38          if (ctors != null ||
39                  mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
40                  mbd.hasConstructorArgumentValues() ||
!ObjectUtils.isEmpty(args))  {
41              //使用容器的自动装配特性，调用匹配的构造方法实例化
42              return autowireConstructor(beanName, mbd, ctors, args);
43          }
44          //使用默认的无参构造方法实例化
45          return instantiateBean(beanName, mbd);
46      }
47      //使用默认的无参构造方法实例化Bean对象
48      protected BeanWrapper instantiateBean(final String beanName, final
RootBeanDefinition mbd) {
49          try {
50              Object beanInstance;
51              final BeanFactory parent = this;
52              //获取系统的安全管理接口，JDK标准的安全管理API
53              if (System.getSecurityManager() != null) {
54                  //这里是一个匿名内置类，根据实例化策略创建实例对象
55                  beanInstance = AccessController.doPrivileged(new
PrivilegedAction<Object>() {
56                      public Object run() {
57                          return getInstantiationStrategy().instantiate(mbd,
beanName, parent);
58                      }
59                  }, getAccessControlContext());
60              }
61              else {
62                  //将实例化的对象封装起来
```

```
63                beanInstance = getInstantiationStrategy().instantiate(mbd,
beanName, parent);
64            }
65            BeanWrapper bw = new BeanWrapperImpl(beanInstance);
66            initBeanWrapper(bw);
67            return bw;
68        }
69        catch (Throwable ex) {
70            throw new BeanCreationException(mbd.getResourceDescription(),
beanName, "Instantiation of bean failed", ex);
71        }
72    }
```

经过对上面的代码分析，我们可以看出，对使用工厂方法和自动装配特性的Bean的实例化相当比较清楚，调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略(JDK的反射机制或者CGLIB)来进行初始化了，在方法getInstantiationStrategy().instantiate中就具体实现类使用初始策略实例化对象。

5、SimpleInstantiationStrategy类使用默认的无参构造方法创建Bean实例化对象：

在使用默认的无参构造方法创建Bean的实例化对象时，方法getInstantiationStrategy().instantiate调用了SimpleInstantiationStrategy类中的实例化Bean的方法，其源码如下：

```
1    //使用初始化策略实例化Bean对象
2    public Object instantiate(RootBeanDefinition beanDefinition, String
beanName, BeanFactory owner) {
3        //如果Bean定义中没有方法覆盖，则就不需要CGLIB父类类的方法
4        if (beanDefinition.getMethodOverrides().isEmpty()) {
5            Constructor<?> constructorToUse;
6            synchronized (beanDefinition.constructorArgumentLock) {
7                //获取对象的构造方法或工厂方法
8                constructorToUse = (Constructor<?>)
beanDefinition.resolvedConstructorOrFactoryMethod;
9                //如果没有构造方法且没有工厂方法
10                if (constructorToUse == null) {
11                    //使用JDK的反射机制，判断要实例化的Bean是否是接口
12                    final Class clazz = beanDefinition.getBeanClass();
13                    if (clazz.isInterface()) {
14                        throw new BeanInstantiationException(clazz, "Specified
class is an interface");
15                    }
16                    try {
17                        if (System.getSecurityManager() != null) {
18                        //这里是一个匿名内置类，使用反射机制获取Bean的构造方法
19                            constructorToUse =
AccessController.doPrivileged(new PrivilegedExceptionAction<Constructor>() {
20                                public Constructor run() throws Exception {
```

```
21                                      return
clazz.getDeclaredConstructor((Class[]) null);
22                                  }
23                              });
24                          }
25                          else {
26                              constructorToUse =
clazz.getDeclaredConstructor((Class[]) null);
27                          }
28                          beanDefinition.resolvedConstructorOrFactoryMethod =
constructorToUse;
29                      }
30                      catch (Exception ex) {
31                          throw new BeanInstantiationException(clazz, "No default
constructor found", ex);
32                      }
33                  }
34              }
35              //使用BeanUtils实例化，通过反射机制调用”构造方法.newInstance(arg)”来进行实
例化
36              return BeanUtils.instantiateClass(constructorToUse);
37          }
38          else {
39              //使用CGLIB来实例化对象
40              return instantiateWithMethodInjection(beanDefinition, beanName,
owner);
41          }
    }
```

通过上面的代码分析，我们看到了如果Bean有方法被覆盖了，则使用JDK的反射机制进行实例化，否则，使用CGLIB进行实例化。

instantiateWithMethodInjection方法调用SimpleInstantiationStrategy的子类CglibSubclassingInstantiationStrategy使用CGLIB来进行初始化，其源码如下：

```
1    //使用CGLIB进行Bean对象实例化
2    public Object instantiate(Constructor ctor, Object[] args) {
3            //CGLIB中的类
4            Enhancer enhancer = new Enhancer();
5            //将Bean本身作为其基类
6            enhancer.setSuperclass(this.beanDefinition.getBeanClass());
7            enhancer.setCallbackFilter(new CallbackFilterImpl());
8            enhancer.setCallbacks(new Callback[] {
9                    NoOp.INSTANCE,
10                    new LookupOverrideMethodInterceptor(),
11                    new ReplaceOverrideMethodInterceptor()
12           });
13           //使用CGLIB的create方法生成实例对象
14           return (ctor == null) ?
```

```
15                      enhancer.create() :
16                      enhancer.create(ctor.getParameterTypes(), args);
17          }
```

CGLIB是一个常用的字节码生成器的类库，它提供了一系列API实现java字节码的生成和转换功能。我们在学习JDK的动态代理时都知道，JDK的动态代理只能针对接口，如果一个类没有实现任何接口，要对其进行动态代理只能使用CGLIB。

6、populateBean方法对Bean属性的依赖注入：

在第3步的分析中我们已经了解到Bean的依赖注入分为以下两个过程：

(1).createBeanInstance：生成Bean所包含的java对象实例。

(2).populateBean ： 对Bean属性的依赖注入进行处理。

第4、5步中我们已经分析了容器初始化生成Bean所包含的Java实例对象的过程，现在我们继续分析生成对象后，Spring IoC容器是如何将Bean的属性依赖关系注入Bean实例对象中并设置好的，属性依赖注入的代码如下：

```
1     //将Bean属性设置到生成的实例对象上
2     protected void populateBean(String beanName, AbstractBeanDefinition mbd,
BeanWrapper bw) {
3         //获取容器在解析Bean定义资源时为BeanDefiniton中设置的属性值
4         PropertyValues pvs = mbd.getPropertyValues();
5         //实例对象为null
6         if (bw == null) {
7             //属性值不为空
8             if (!pvs.isEmpty()) {
9                 throw new BeanCreationException(
10                          mbd.getResourceDescription(), beanName, "Cannot apply
property values to null instance");
11             }
12             else {
13                 //实例对象为null，属性值也为空，不需要设置属性值，直接返回
14                 return;
15             }
16         }
17         //在设置属性之前调用Bean的PostProcessor后置处理器
18         boolean continueWithPropertyPopulation = true;
19         if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

20             for (BeanPostProcessor bp : getBeanPostProcessors()) {
21                 if (bp instanceof InstantiationAwareBeanPostProcessor) {
22                     InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
23                     if
(!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
```

```
24                          continueWithPropertyPopulation = false;
25                          break;
26                      }
27                  }
28              }
29          }
30          if (!continueWithPropertyPopulation) {
31              return;
32          }
33          //依赖注入开始，首先处理autowire自动装配的注入
34          if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME ||
35                  mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
36              MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
37              //对autowire自动装配的处理，根据Bean名称自动装配注入
38              if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME) {
39                  autowireByName(beanName, mbd, bw, newPvs);
40              }
41              //根据Bean类型自动装配注入
42              if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
43                  autowireByType(beanName, mbd, bw, newPvs);
44              }
45              pvs = newPvs;
46          }
47          //检查容器是否持有用于处理单态模式Bean关闭时的后置处理器
48          boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

49          //Bean实例对象没有依赖，即没有继承基类
50          boolean needsDepCheck = (mbd.getDependencyCheck() !=
RootBeanDefinition.DEPENDENCY_CHECK_NONE);
51          if (hasInstAwareBpps || needsDepCheck) {
52              //从实例对象中提取属性描述符
53              PropertyDescriptor[] filteredPds =
filterPropertyDescriptorsForDependencyCheck(bw);
54              if (hasInstAwareBpps) {
55                  for (BeanPostProcessor bp : getBeanPostProcessors()) {
56                      if (bp instanceof InstantiationAwareBeanPostProcessor) {
57                          InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
58                          //使用BeanPostProcessor处理器处理属性值
59                          pvs = ibp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(), beanName);
60                          if (pvs == null) {
61                              return;
62                          }
63                      }
64                  }
65              }
66              if (needsDepCheck) {
67                  //为要设置的属性进行依赖检查
68                  checkDependencies(beanName, mbd, filteredPds, pvs);
69              }
70          }
71          //对属性进行注入
72          applyPropertyValues(beanName, mbd, bw, pvs);
```

```
73    }
74    //解析并注入依赖属性的过程
75    protected void applyPropertyValues(String beanName, BeanDefinition mbd,
BeanWrapper bw, PropertyValues pvs) {
76        if (pvs == null || pvs.isEmpty()) {
77            return;
78        }
79        //封装属性值
80        MutablePropertyValues mpvs = null;
81        List<PropertyValue> original;
82        if (System.getSecurityManager()!= null) {
83            if (bw instanceof BeanWrapperImpl) {
84                //设置安全上下文，JDK安全机制
85                ((BeanWrapperImpl)
bw).setSecurityContext(getAccessControlContext());
86            }
87        }
88        if (pvs instanceof MutablePropertyValues) {
89            mpvs = (MutablePropertyValues) pvs;
90            //属性值已经转换
91            if (mpvs.isConverted()) {
92                try {
93                    //为实例化对象设置属性值
94                    bw.setPropertyValues(mpvs);
95                    return;
96                }
97                catch (BeansException ex) {
98                    throw new BeanCreationException(
99                            mbd.getResourceDescription(), beanName, "Error
setting property values", ex);
100                }
101            }
102            //获取属性值对象的原始类型值
103            original = mpvs.getPropertyValueList();
104        }
105        else {
106            original = Arrays.asList(pvs.getPropertyValues());
107        }
108        //获取用户自定义的类型转换
109        TypeConverter converter = getCustomTypeConverter();
110        if (converter == null) {
111            converter = bw;
112        }
113        //创建一个Bean定义属性值解析器，将Bean定义中的属性值解析为Bean实例对象
114        //的实际值
115        BeanDefinitionValueResolver valueResolver = new
BeanDefinitionValueResolver(this, beanName, mbd, converter);
116        //为属性的解析值创建一个拷贝，将拷贝的数据注入到实例对象中
117        List<PropertyValue> deepCopy = new ArrayList<PropertyValue>
(original.size());
118        boolean resolveNecessary = false;
119        for (PropertyValue pv : original) {
120            //属性值不需要转换
121            if (pv.isConverted()) {
122                deepCopy.add(pv);
123            }
124            //属性值需要转换
125            else {
```

```
126                 String propertyName = pv.getName();
127                 //原始的属性值，即转换之前的属性值
128                 Object originalValue = pv.getValue();
129                 //转换属性值，例如将引用转换为IoC容器中实例化对象引用
130                 Object resolvedValue =
valueResolver.resolveValueIfNecessary(pv, originalValue);
131                 //转换之后的属性值
132                 Object convertedValue = resolvedValue;
133                 //属性值是否可以转换
134                 boolean convertible = bw.isWritableProperty(propertyName) &&

135
!PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
136                 if (convertible) {
137                     //使用用户自定义的类型转换器转换属性值
138                     convertedValue = convertForProperty(resolvedValue,
propertyName, bw, converter);
139                 }
140                 //存储转换后的属性值，避免每次属性注入时的转换工作
141                 if (resolvedValue == originalValue) {
142                     if (convertible) {
143                         //设置属性转换之后的值
144                         pv.setConvertedValue(convertedValue);
145                     }
146                     deepCopy.add(pv);
147                 }
148                 //属性是可转换的，且属性原始值是字符串类型，且属性的原始类型值不是
149                 //动态生成的字符串，且属性的原始值不是集合或者数组类型
150                 else if (convertible && originalValue instanceof
TypedStringValue &&
151                         !((TypedStringValue) originalValue).isDynamic() &&
152                         !(convertedValue instanceof Collection ||
ObjectUtils.isArray(convertedValue))) {
153                     pv.setConvertedValue(convertedValue);
154                     deepCopy.add(pv);
155                 }
156                 else {
157                     resolveNecessary = true;
158                     //重新封装属性的值
159                     deepCopy.add(new PropertyValue(pv, convertedValue));
160                 }
161             }
162         }
163         if (mpvs != null && !resolveNecessary) {
164             //标记属性值已经转换过
165             mpvs.setConverted();
166         }
167         //进行属性依赖注入
168         try {
169             bw.setPropertyValues(new MutablePropertyValues(deepCopy));
170         }
171         catch (BeansException ex) {
172             throw new BeanCreationException(
173                     mbd.getResourceDescription(), beanName, "Error setting
property values", ex);
174         }
    }
```

分析上述代码，我们可以看出，对属性的注入过程分以下两种情况：

(1).属性值类型不需要转换时，不需要解析属性值，直接准备进行依赖注入。

(2).属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

对属性值的解析是在BeanDefinitionValueResolver类中的resolveValueIfNecessary方法中进行的，对属性值的依赖注入是通过bw.setPropertyValues方法实现的，在分析属性值的依赖注入之前，我们先分析一下对属性值的解析过程。

7、BeanDefinitionValueResolver解析属性值：

当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个Bean实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由resolveValueIfNecessary方法实现，其源码如下：

```
1    //解析属性值，对注入类型进行转换
2    public Object resolveValueIfNecessary(Object argName, Object value) {
3        //对引用类型的属性进行解析
4        if (value instanceof RuntimeBeanReference) {
5            RuntimeBeanReference ref = (RuntimeBeanReference) value;
6            //调用引用类型属性的解析方法
7            return resolveReference(argName, ref);
8        }
9        //对属性值是引用容器中另一个Bean名称的解析
10        else if (value instanceof RuntimeBeanNameReference) {
11            String refName = ((RuntimeBeanNameReference) value).getBeanName();

12            refName = String.valueOf(evaluate(refName));
13            //从容器中获取指定名称的Bean
14            if (!this.beanFactory.containsBean(refName)) {
15                throw new BeanDefinitionStoreException(
16                        "Invalid bean name '" + refName + "' in bean reference
for " + argName);
17            }
18            return refName;
19        }
20        //对Bean类型属性的解析，主要是Bean中的内部类
21        else if (value instanceof BeanDefinitionHolder) {
22            BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
23            return resolveInnerBean(argName, bdHolder.getBeanName(),
bdHolder.getBeanDefinition());
24        }
25        else if (value instanceof BeanDefinition) {
26            BeanDefinition bd = (BeanDefinition) value;
27            return resolveInnerBean(argName, "(inner bean)", bd);
28        }
```

```
29          //对集合数组类型的属性解析
30          else if (value instanceof ManagedArray) {
31              ManagedArray array = (ManagedArray) value;
32              //获取数组的类型
33              Class elementType = array.resolvedElementType;
34              if (elementType == null) {
35                  //获取数组元素的类型
36                  String elementTypeName = array.getElementTypeName();
37                  if (StringUtils.hasText(elementTypeName)) {
38                      try {
39                          //使用反射机制创建指定类型的对象
40                          elementType = ClassUtils.forName(elementTypeName,
this.beanFactory.getBeanClassLoader());
41                          array.resolvedElementType = elementType;
42                      }
43                      catch (Throwable ex) {
44                          throw new BeanCreationException(
45                                  this.beanDefinition.getResourceDescription(),
this.beanName,
46                                  "Error resolving array type for " + argName,
ex);
47                      }
48                  }
49                  //没有获取到数组的类型，也没有获取到数组元素的类型，则直接设置数
50                  //组的类型为Object
51                  else {
52                      elementType = Object.class;
53                  }
54              }
55              //创建指定类型的数组
56              return resolveManagedArray(argName, (List<?>) value, elementType);

57          }
58          //解析list类型的属性值
59          else if (value instanceof ManagedList) {
60              return resolveManagedList(argName, (List<?>) value);
61          }
62          //解析set类型的属性值
63          else if (value instanceof ManagedSet) {
64              return resolveManagedSet(argName, (Set<?>) value);
65          }
66          //解析map类型的属性值
67          else if (value instanceof ManagedMap) {
68              return resolveManagedMap(argName, (Map<?, ?>) value);
69          }
70          //解析props类型的属性值，props其实就是key和value均为字符串的map
71          else if (value instanceof ManagedProperties) {
72              Properties original = (Properties) value;
73              //创建一个拷贝，用于作为解析后的返回值
74              Properties copy = new Properties();
75              for (Map.Entry propEntry : original.entrySet()) {
76                  Object propKey = propEntry.getKey();
77                  Object propValue = propEntry.getValue();
78                  if (propKey instanceof TypedStringValue) {
79                      propKey = evaluate((TypedStringValue) propKey);
80                  }
81                  if (propValue instanceof TypedStringValue) {
82                      propValue = evaluate((TypedStringValue) propValue);
```

```
83                }
84                copy.put(propKey, propValue);
85            }
86            return copy;
87        }
88        //解析字符串类型的属性值
89        else if (value instanceof TypedStringValue) {
90            TypedStringValue typedStringValue = (TypedStringValue) value;
91            Object valueObject = evaluate(typedStringValue);
92            try {
93                //获取属性的目标类型
94                Class<?> resolvedTargetType =
resolveTargetType(typedStringValue);
95                if (resolvedTargetType != null) {
96                    //对目标类型的属性进行解析，递归调用
97                    return this.typeConverter.convertIfNecessary(valueObject,
resolvedTargetType);
98                }
99                //没有获取到属性的目标对象，则按Object类型返回
100                else {
101                    return valueObject;
102                }
103            }
104            catch (Throwable ex) {
105                throw new BeanCreationException(
106                        this.beanDefinition.getResourceDescription(),
this.beanName,
107                        "Error converting typed String value for " + argName,
ex);
108            }
109        }
110        else {
111            return evaluate(value);
112        }
113    }
114    //解析引用类型的属性值
115    private Object resolveReference(Object argName, RuntimeBeanReference ref)
{
116        try {
117            //获取引用的Bean名称
118            String refName = ref.getBeanName();
119            refName = String.valueOf(evaluate(refName));
120            //如果引用的对象在父类容器中，则从父类容器中获取指定的引用对象
121            if (ref.isToParent()) {
122                if (this.beanFactory.getParentBeanFactory() == null) {
123                    throw new BeanCreationException(
124                            this.beanDefinition.getResourceDescription(),
this.beanName,
125                            "Can't resolve reference to bean '" + refName +
126                            "' in parent factory: no parent factory
available");
127                }
128                return
this.beanFactory.getParentBeanFactory().getBean(refName);
129            }
130            //从当前的容器中获取指定的引用Bean对象，如果指定的Bean没有被实例化
131            //则会递归触发引用Bean的初始化和依赖注入
132            else {
```

```
133                 Object bean = this.beanFactory.getBean(refName);
134                 //将当前实例化对象的依赖引用对象
135                 this.beanFactory.registerDependentBean(refName,
this.beanName);
136                 return bean;
137             }
138         }
139         catch (BeansException ex) {
140             throw new BeanCreationException(
141                     this.beanDefinition.getResourceDescription(),
this.beanName,
142                     "Cannot resolve reference to bean '" + ref.getBeanName() +
"' while setting " + argName, ex);
143         }
144     }
145     //解析array类型的属性
146     private Object resolveManagedArray(Object argName, List<?> ml, Class
elementType) {
147         //创建一个指定类型的数组，用于存放和返回解析后的数组
148         Object resolved = Array.newInstance(elementType, ml.size());
149         for (int i = 0; i < ml.size(); i++) {
150         //递归解析array的每一个元素，并将解析后的值设置到resolved数组中，索引为i
151             Array.set(resolved, i,
152                 resolveValueIfNecessary(new KeyedArgName(argName, i),
ml.get(i)));
153         }
154         return resolved;
155     }
156     //解析list类型的属性
157     private List resolveManagedList(Object argName, List<?> ml) {
158         List<Object> resolved = new ArrayList<Object>(ml.size());
159         for (int i = 0; i < ml.size(); i++) {
160             //递归解析list的每一个元素
161             resolved.add(
162                 resolveValueIfNecessary(new KeyedArgName(argName, i),
ml.get(i)));
163         }
164         return resolved;
165     }
166     //解析set类型的属性
167     private Set resolveManagedSet(Object argName, Set<?> ms) {
168         Set<Object> resolved = new LinkedHashSet<Object>(ms.size());
169         int i = 0;
170         //递归解析set的每一个元素
171         for (Object m : ms) {
172             resolved.add(resolveValueIfNecessary(new KeyedArgName(argName, i),
m));
173             i++;
174         }
175         return resolved;
176     }
177     //解析map类型的属性
178     private Map resolveManagedMap(Object argName, Map<?, ?> mm) {
179         Map<Object, Object> resolved = new LinkedHashMap<Object, Object>
(mm.size());
180         //递归解析map中每一个元素的key和value
181         for (Map.Entry entry : mm.entrySet()) {
```

```
182            Object resolvedKey = resolveValueIfNecessary(argName,
entry.getKey());
183            Object resolvedValue = resolveValueIfNecessary(
184                    new KeyedArgName(argName, entry.getKey()),
entry.getValue());
185            resolved.put(resolvedKey, resolvedValue);
186        }
187        return resolved;
188    }
```

通过上面的代码分析，我们明白了Spring是如何将引用类型，内部类以及集合类型等属性进行解析的，属性值解析完成后就可以进行依赖注入了，依赖注入的过程就是Bean对象实例设置到它所依赖的Bean对象属性上去，在第7步中我们已经说过，依赖注入是通过bw.setPropertyValues方法实现的，该方法也使用了委托模式，在BeanWrapper接口中至少定义了方法声明，依赖注入的具体实现交由其实现类BeanWrapperImpl来完成，下面我们就分析依BeanWrapperImpl中赖注入相关的源码。

8、BeanWrapperImpl对Bean属性的依赖注入：

BeanWrapperImpl类主要是对容器中完成初始化的Bean实例对象进行属性的依赖注入，即把Bean对象设置到它所依赖的另一个Bean的属性中去，依赖注入的相关源码如下：

```
1    //实现属性依赖注入功能
2    private void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv)
throws BeansException {
3        //PropertyTokenHolder主要保存属性的名称、路径，以及集合的size等信息
4        String propertyName = tokens.canonicalName;
5        String actualName = tokens.actualName;
6        //keys是用来保存集合类型属性的size
7        if (tokens.keys != null) {
8            //将属性信息拷贝
9            PropertyTokenHolder getterTokens = new PropertyTokenHolder();
10            getterTokens.canonicalName = tokens.canonicalName;
11            getterTokens.actualName = tokens.actualName;
12            getterTokens.keys = new String[tokens.keys.length - 1];
13            System.arraycopy(tokens.keys, 0, getterTokens.keys, 0,
tokens.keys.length - 1);
14            Object propValue;
15            try {
16                //获取属性值，该方法内部使用JDK的内省( Introspector)机制，调用属性//的
getter(readerMethod)方法，获取属性的值
```

```
17                propValue = getPropertyValue(getterTokens);
18            }
19            catch (NotReadablePropertyException ex) {
20                throw new NotWritablePropertyException(getRootClass(),
this.nestedPath + propertyName,
21                        "Cannot access indexed value in property referenced " +

22                        "in indexed property path '" + propertyName + "'", ex);

23            }
24            //获取集合类型属性的长度
25            String key = tokens.keys[tokens.keys.length - 1];
26            if (propValue == null) {
27                throw new NullValueInNestedPathException(getRootClass(),
this.nestedPath + propertyName,
28                        "Cannot access indexed value in property referenced " +

29                        "in indexed property path '" + propertyName + "':
returned null");
30            }
31            //注入array类型的属性值
32            else if (propValue.getClass().isArray()) {
33                //获取属性的描述符
34                PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
35                //获取数组的类型
36                Class requiredType = propValue.getClass().getComponentType();

37                //获取数组的长度
38                int arrayIndex = Integer.parseInt(key);
39                Object oldValue = null;
40                try {
41                    //获取数组以前初始化的值
42                    if (isExtractOldValueForEditor()) {
43                        oldValue = Array.get(propValue, arrayIndex);
44                    }
45                    //将属性的值赋值给数组中的元素
46                    Object convertedValue = convertIfNecessary(propertyName,
oldValue, pv.getValue(), requiredType,
47                            new PropertyTypeDescriptor(pd, new
MethodParameter(pd.getReadMethod(), -1), requiredType));
48                    Array.set(propValue, arrayIndex, convertedValue);
49                }
50                catch (IndexOutOfBoundsException ex) {
51                    throw new InvalidPropertyException(getRootClass(),
this.nestedPath + propertyName,
52                            "Invalid array index in property path '" +
propertyName + "'", ex);
53                }
54            }
55            //注入list类型的属性值
56            else if (propValue instanceof List) {
57                PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
58                //获取list集合的类型
59                Class requiredType =
GenericCollectionTypeResolver.getCollectionReturnType(
60                        pd.getReadMethod(), tokens.keys.length);
```

```
61                List list = (List) propValue;
62                //获取list集合的size
63                int index = Integer.parseInt(key);
64                Object oldValue = null;
65                if (isExtractOldValueForEditor() && index < list.size()) {
66                    oldValue = list.get(index);
67                }
68                //获取list解析后的属性值
69                Object convertedValue = convertIfNecessary(propertyName,
oldValue, pv.getValue(), requiredType,
70                        new PropertyTypeDescriptor(pd, new
MethodParameter(pd.getReadMethod(), -1), requiredType));
71                if (index < list.size()) {
72                    //为list属性赋值
73                    list.set(index, convertedValue);
74                }
75                //如果list的长度大于属性值的长度，则多余的元素赋值为null
76                else if (index >= list.size()) {
77                    for (int i = list.size(); i < index; i++) {
78                        try {
79                            list.add(null);
80                        }
81                        catch (NullPointerException ex) {
82                            throw new InvalidPropertyException(getRootClass(),
this.nestedPath + propertyName,
83                                    "Cannot set element with index " + index +
" in List of size " +
84                                    list.size() + ", accessed using property
path '" + propertyName +
85                                    "': List does not support filling up gaps
with null elements");
86                        }
87                    }
88                    list.add(convertedValue);
89                }
90            }
91            //注入map类型的属性值
92            else if (propValue instanceof Map) {
93                PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
94                //获取map集合key的类型
95                Class mapKeyType =
GenericCollectionTypeResolver.getMapKeyReturnType(
96                        pd.getReadMethod(), tokens.keys.length);
97                //获取map集合value的类型
98                Class mapValueType =
GenericCollectionTypeResolver.getMapValueReturnType(
99                        pd.getReadMethod(), tokens.keys.length);
100               Map map = (Map) propValue;
101               //解析map类型属性key值
102               Object convertedMapKey = convertIfNecessary(null, null, key,
mapKeyType,
103                       new PropertyTypeDescriptor(pd, new
MethodParameter(pd.getReadMethod(), -1), mapKeyType));
104               Object oldValue = null;
105               if (isExtractOldValueForEditor()) {
106                   oldValue = map.get(convertedMapKey);
107               }
```

```
108                 //解析map类型属性value值
109                 Object convertedMapValue = convertIfNecessary(
110                         propertyName, oldValue, pv.getValue(), mapValueType,
111                         new TypeDescriptor(new
MethodParameter(pd.getReadMethod(), -1, tokens.keys.length + 1)));
112                 //将解析后的key和value值赋值给map集合属性
113                 map.put(convertedMapKey, convertedMapValue);
114             }
115             else {
116                 throw new InvalidPropertyException(getRootClass(),
this.nestedPath + propertyName,
117                         "Property referenced in indexed property path '" +
propertyName +
118                         "' is neither an array nor a List nor a Map; returned
value was [" + pv.getValue() + "]");
119             }
120         }
121         //对非集合类型的属性注入
122         else {
123             PropertyDescriptor pd = pv.resolvedDescriptor;
124             if (pd == null ||
!pd.getWriteMethod().getDeclaringClass().isInstance(this.object)) {
125                 pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
126                 //无法获取到属性名或者属性没有提供setter(写方法)方法
127                 if (pd == null || pd.getWriteMethod() == null) {
128                     //如果属性值是可选的，即不是必须的，则忽略该属性值
129                     if (pv.isOptional()) {
130                         logger.debug("Ignoring optional value for property '"
+ actualName +
131                                 "' - property not found on bean class [" +
getRootClass().getName() + "]");
132                         return;
133                     }
134                     //如果属性值是必须的，则抛出无法给属性赋值，因为每天提供setter方法异
常
135                     else {
136                         PropertyMatches matches =
PropertyMatches.forProperty(propertyName, getRootClass());
137                         throw new NotWritablePropertyException(
138                                 getRootClass(), this.nestedPath +
propertyName,
139                                 matches.buildErrorMessage(),
matches.getPossibleMatches());
140                     }
141                 }
142                 pv.getOriginalPropertyValue().resolvedDescriptor = pd;
143             }
144             Object oldValue = null;
145             try {
146                 Object originalValue = pv.getValue();
147                 Object valueToApply = originalValue;
148                 if (!Boolean.FALSE.equals(pv.conversionNecessary)) {
149                     if (pv.isConverted()) {
150                         valueToApply = pv.getConvertedValue();
151                     }
152                     else {
```

```
153                         if (isExtractOldValueForEditor() && pd.getReadMethod()
!= null) {
154                             //获取属性的getter方法(读方法)，JDK内省机制
155                             final Method readMethod = pd.getReadMethod();
156                             //如果属性的getter方法不是public访问控制权限的，即访问控
制权限比较严格，
157                             //则使用JDK的反射机制强行访问非public的方法(暴力读取属性
值)
158                             if
(!Modifier.isPublic(readMethod.getDeclaringClass().getModifiers()) &&
159                                     !readMethod.isAccessible()) {
160                                 if (System.getSecurityManager()!= null) {
161                                     //匿名内部类，根据权限修改属性的读取控制限制
162                                     AccessController.doPrivileged(new
PrivilegedAction<Object>() {
163                                         public Object run() {
164                                             readMethod.setAccessible(true);
165                                             return null;
166                                         }
167                                     });
168                                 }
169                                 else {
170                                     readMethod.setAccessible(true);
171                                 }
172                             }
173                             try {
174                                 //属性没有提供getter方法时，调用潜在的读取属性值//的
方法，获取属性值
175                                 if (System.getSecurityManager() != null) {
176                                     oldValue =
AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
177                                         public Object run() throws Exception {

178                                             return readMethod.invoke(object);

179                                         }
180                                     }, acc);
181                                 }
182                                 else {
183                                     oldValue = readMethod.invoke(object);
184                                 }
185                             }
186                             catch (Exception ex) {
187                                 if (ex instanceof PrivilegedActionException) {

188                                     ex = ((PrivilegedActionException)
ex).getException();
189                                 }
190                                 if (logger.isDebugEnabled()) {
191                                     logger.debug("Could not read previous
value of property '" +
192                                             this.nestedPath + propertyName +
"'", ex);
193                                 }
194                             }
195                         }
196                         //设置属性的注入值
```

```
197                         valueToApply = convertForProperty(propertyName,
oldValue, originalValue, pd);
198                     }
199                     pv.getOriginalPropertyValue().conversionNecessary =
(valueToApply != originalValue);
200                 }
201                 //根据JDK的内省机制，获取属性的setter(写方法)方法
202                 final Method writeMethod = (pd instanceof
GenericTypeAwarePropertyDescriptor ?
203                         ((GenericTypeAwarePropertyDescriptor)
pd).getWriteMethodForActualAccess() :
204                         pd.getWriteMethod());
205                 //如果属性的setter方法是非public，即访问控制权限比较严格，则使用JDK的反
射机制，
206                 //强行设置setter方法可访问(暴力为属性赋值)
207                 if
(!Modifier.isPublic(writeMethod.getDeclaringClass().getModifiers()) &&
!writeMethod.isAccessible()) {
208                     //如果使用了JDK的安全机制，则需要权限验证
209                     if (System.getSecurityManager()!= null) {
210                         AccessController.doPrivileged(new
PrivilegedAction<Object>() {
211                             public Object run() {
212                                 writeMethod.setAccessible(true);
213                                 return null;
214                             }
215                         });
216                     }
217                     else {
218                         writeMethod.setAccessible(true);
219                     }
220                 }
221                 final Object value = valueToApply;
222                 if (System.getSecurityManager() != null) {
223                     try {
224                         //将属性值设置到属性上去
225                         AccessController.doPrivileged(new
PrivilegedExceptionAction<Object>() {
226                             public Object run() throws Exception {
227                                 writeMethod.invoke(object, value);
228                                 return null;
229                             }
230                         }, acc);
231                     }
232                     catch (PrivilegedActionException ex) {
233                         throw ex.getException();
234                     }
235                 }
236                 else {
237                     writeMethod.invoke(this.object, value);
238                 }
239             }
240             catch (TypeMismatchException ex) {
241                 throw ex;
242             }
243             catch (InvocationTargetException ex) {
244                 PropertyChangeEvent propertyChangeEvent =
```

```
245                         new PropertyChangeEvent(this.rootObject,
this.nestedPath + propertyName, oldValue, pv.getValue());
246                 if (ex.getTargetException() instanceof ClassCastException) {

247                     throw new TypeMismatchException(propertyChangeEvent,
pd.getPropertyType(), ex.getTargetException());
248                 }
249                 else {
250                     throw new MethodInvocationException(propertyChangeEvent,
ex.getTargetException());
251                 }
252             }
253             catch (Exception ex) {
254                 PropertyChangeEvent pce =
255                         new PropertyChangeEvent(this.rootObject,
this.nestedPath + propertyName, oldValue, pv.getValue());
256                 throw new MethodInvocationException(pce, ex);
257             }
258         }
    }
```

通过对上面注入依赖代码的分析，我们已经明白了Spring IoC容器是如何将属性的值注入到Bean实例对象中去的：

(1).对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

(2).对于非集合类型的属性，大量使用了JDK的反射和内省机制，通过属性的getter方法(reader method)获取指定属性注入以前的值，同时调用属性的setter方法(writer method)为属性设置注入后的值。看到这里相信很多人都明白了Spring的setter注入原理。

至此Spring IoC容器对Bean定义资源文件的定位，载入、解析和依赖注入已经全部分析完毕，现在Spring IoC容器中管理了一系列靠依赖关系联系起来的Bean，程序不需要应用自己手动创建所需的对象，Spring IoC容器会在我们使用的时候自动为我们创建，并且为我们注入好相关的依赖，这就是Spring核心功能的控制反转和依赖注入的相关功能。

五、IoC容器的高级特性

1、介绍

通过前面4篇文章对Spring IoC容器的源码分析，我们已经基本上了解了Spring IoC容器对Bean定义资源的定位、读入和解析过程，同时也清楚了当用户通过getBean方法向IoC容器获取被管理的Bean时，IoC容器对Bean进行的初始化和依赖注入过程，这些是Spring IoC容器的基本功能特性。Spring IoC容器还有一些高级特性，如使用lazy-init属性对Bean预初始化、FactoryBean产生或者修饰Bean对象的生成、IoC容器初始化Bean过程中使用BeanPostProcessor后置处理器对Bean声明周期事件管理和IoC容器的autowiring自动装配功能等。

2、Spring IoC容器的lazy-init属性实现预实例化：

通过前面我们对IoC容器的实现和工作原理分析，我们知道IoC容器的初始化过程就是对Bean定义资源的定位、载入和注册，此时容器对Bean的依赖注入并没有发生，依赖注入主要是在应用程序第一次向容器索取Bean时，通过getBean方法的调用完成。

当Bean定义资源的元素中配置了lazy-init属性时，容器将会在初始化的时候对所配置的Bean进行预实例化，Bean的依赖注入在容器初始化的时候就已经完成。这样，当应用程序第一次向容器索取被管理的Bean时，就不用再初始化和对Bean进行依赖注入了，直接从容器中获取已经完成依赖注入的现成Bean，可以提高应用第一次向容器获取Bean的性能。

下面我们通过代码分析容器预实例化的实现过程：

(1).refresh()

先从IoC容器的初始会过程开始，通过前面文章分析，我们知道IoC容器读入已经定位的Bean定义资源是从refresh方法开始的，我们首先从AbstractApplicationContext类的refresh方法入手分析，源码如下：

```
1    //容器初始化的过程，读入Bean定义资源，并解析注册
2    public void refresh() throws BeansException, IllegalStateException {
3        synchronized (this.startupShutdownMonitor) {
4             //调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识
5             prepareRefresh();
6             //告诉子类启动refreshBeanFactory()方法，Bean定义资源文件的载入从
7             //子类的refreshBeanFactory()方法启动
8             ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();
9             //为BeanFactory配置容器特性，例如类加载器、事件处理器等
10            prepareBeanFactory(beanFactory);
11            try {
12                //为容器的某些子类指定特殊的BeanPost事件处理器
13                postProcessBeanFactory(beanFactory);
14                //调用所有注册的BeanFactoryPostProcessor的Bean
15                invokeBeanFactoryPostProcessors(beanFactory);
16                //为BeanFactory注册BeanPost事件处理器.
17                //BeanPostProcessor是Bean后置处理器，用于监听容器触发的事件
18                registerBeanPostProcessors(beanFactory);
19                //初始化信息源，和国际化相关.
20                initMessageSource();
21                //初始化容器事件传播器.
22                initApplicationEventMulticaster();
23                //调用子类的某些特殊Bean初始化方法
24                onRefresh();
25                //为事件传播器注册事件监听器.
26                registerListeners();
27                //这里是对容器lazy-init属性进行处理的入口方法
28                finishBeanFactoryInitialization(beanFactory);
29                //初始化容器的生命周期事件处理器，并发布容器的生命周期事件
30                finishRefresh();
31            }
32            catch (BeansException ex) {
33                //销毁以创建的单态Bean
34                destroyBeans();
35                //取消refresh操作，重置容器的同步标识.
36                cancelRefresh(ex);
37                throw ex;
38            }
39        }
    }
```

在refresh方法中ConfigurableListableBeanFactorybeanFactory = obtainFreshBeanFactory();启动了Bean定义资源的载入、注册过程，而finishBeanFactoryInitialization方法是对注册后的Bean定义中的预实例化(lazy-init=false，Spring默认就是预实例化，即为true)的Bean进行处理的地方。

(2).finishBeanFactoryInitialization处理预实例化Bean：

当Bean定义资源被载入IoC容器之后，容器将Bean定义资源解析为容器内部的数据结构BeanDefinition注册到容器中，AbstractApplicationContext类中的finishBeanFactoryInitialization方法对配置了预实例化属性的Bean进行预初始化过程，源码如下：

```
1    //对配置了lazy-init属性的Bean进行预实例化处理
2    protected void
finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
3        //这是Spring3以后新加的代码，为容器指定一个转换服务(ConversionService)
4        //在对某些Bean属性进行转换时使用
5        if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
6                beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class)) {
7            beanFactory.setConversionService(
8                    beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class));
9        }
10        //为了类型匹配，停止使用临时的类加载器
11        beanFactory.setTempClassLoader(null);
12        //缓存容器中所有注册的BeanDefinition元数据，以防被修改
13        beanFactory.freezeConfiguration();
14        //对配置了lazy-init属性的单态模式Bean进行预实例化处理
15        beanFactory.preInstantiateSingletons();
    }
```

ConfigurableListableBeanFactory是一个接口，其preInstantiateSingletons方法由其子类DefaultListableBeanFactory提供。

(3)、DefaultListableBeanFactory对配置lazy-init属性单态Bean的预实例化：

```
1//对配置lazy-init属性单态Bean的预实例化
2public void preInstantiateSingletons() throws BeansException {
3        if (this.logger.isInfoEnabled()) {
4            this.logger.info("Pre-instantiating singletons in " + this);
5        }
6        //在对配置lazy-init属性单态Bean的预实例化过程中，必须多线程同步，以确保数据一致性
7        synchronized (this.beanDefinitionMap) {
8            for (String beanName : this.beanDefinitionNames) {
9                //获取指定名称的Bean定义
10                RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

11                //Bean不是抽象的，是单态模式的，且lazy-init属性配置为false
12                if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {

13                    //如果指定名称的bean是创建容器的Bean
14                    if (isFactoryBean(beanName)) {
15                    //FACTORY_BEAN_PREFIX=”&”，当Bean名称前面加”&”符号
16                   //时，获取的是产生容器对象本身，而不是容器产生的Bean.
17                   //调用getBean方法，触发容器对Bean实例化和依赖注入过程
18                        final FactoryBean factory = (FactoryBean)
getBean(FACTORY_BEAN_PREFIX + beanName);
19                        //标识是否需要预实例化
20                        boolean isEagerInit;
21                        if (System.getSecurityManager() != null && factory
instanceof SmartFactoryBean) {
22                            //一个匿名内部类
23                            isEagerInit = AccessController.doPrivileged(new
PrivilegedAction<Boolean>() {
24                                public Boolean run() {
25                                    return ((SmartFactoryBean)
factory).isEagerInit();
26                                }
27                            }, getAccessControlContext());
28                        }
29                        else {
30                            isEagerInit = factory instanceof SmartFactoryBean
&& ((SmartFactoryBean) factory).isEagerInit();
31                        }
32                        if (isEagerInit) {
33                           //调用getBean方法，触发容器对Bean实例化和依赖注入过程
34                            getBean(beanName);
35                        }
36                    }
37                    else {
38                         //调用getBean方法，触发容器对Bean实例化和依赖注入过程
39                        getBean(beanName);
40                    }
41                }
42            }
43        }
    }
```

通过对lazy-init处理源码的分析，我们可以看出，如果设置了lazy-init属性，则容器在完成Bean定义的注册之后，会通过getBean方法，触发对指定Bean的初始化和依赖注入过程，这样当应用第一次向容器索取所需的Bean时，容器不再需要对Bean进行初始化和依赖注入，直接从已经完成实例化和依赖注入的Bean中取一个线程的Bean，这样就提高了第一次获取Bean的性能。

3、FactoryBean的实现：

> 在Spring中，有两个很容易混淆的类：BeanFactory和FactoryBean。

BeanFactory：Bean工厂，是一个工厂(Factory)，我们Spring IoC容器的最顶层接口就是这个BeanFactory，它的作用是管理Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean：工厂Bean，是一个Bean，作用是产生其他bean实例。通常情况下，这种bean没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他bean实例。通常情况下，bean无须自己实现工厂模式，Spring容器担任工厂角色；但少数情况下，容器中的bean本身就是工厂，其作用是产生其它bean实例。

当用户使用容器本身时，可以使用转义字符”&”来得到FactoryBean本身，以区别通过FactoryBean产生的实例对象和FactoryBean对象本身。在BeanFactory中通过如下代码定义了该转义字符：

StringFACTORY_BEAN_PREFIX = "&";

如果myJndiObject是一个FactoryBean，则使用&myJndiObject得到的是myJndiObject对象，而不是myJndiObject产生出来的对象。

(1).FactoryBean的源码如下：

```
//工厂Bean，用于产生其他对象
public interface FactoryBean<T> {
   //获取容器管理的对象实例
    T getObject() throws Exception;
    //获取Bean工厂创建的对象的类型
    Class<?> getObjectType();
    //Bean工厂创建的对象是否是单态模式，如果是单态模式，则整个容器中只有一个实例
   //对象，每次请求都返回同一个实例对象
    boolean isSingleton();
}
```

(2). AbstractBeanFactory的getBean方法调用FactoryBean：

在前面我们分析Spring Ioc容器实例化Bean并进行依赖注入过程的源码时，提到在getBean方法触发容器实例化Bean的时候会调用AbstractBeanFactory的doGetBean方法来进行实例化的过程，源码如下：

```
1      //真正实现向IoC容器获取Bean的功能，也是触发依赖注入功能的地方
2      @SuppressWarnings("unchecked")
3      protected <T> T doGetBean(
4              final String name, final Class<T> requiredType, final Object[]
args, boolean typeCheckOnly)
5              throws BeansException {
```

```
6           //根据指定的名称获取被管理Bean的名称，剥离指定名称中对容器的相关依赖
7           //如果指定的是别名，将别名转换为规范的Bean名称
8           final String beanName = transformedBeanName(name);
9           Object bean;
10         //先从缓存中取是否已经有被创建过的单态类型的Bean，对于单态模式的Bean整
11             //个IoC容器中只创建一次，不需要重复创建
12           Object sharedInstance = getSingleton(beanName);
13           //IoC容器创建单态模式Bean实例对象
14           if (sharedInstance != null && args == null) {
15               if (logger.isDebugEnabled()) {
16               //如果指定名称的Bean在容器中已有单态模式的Bean被创建，直接返回
17                     //已经创建的Bean
18                   if (isSingletonCurrentlyInCreation(beanName)) {
19                       logger.debug("Returning eagerly cached instance of
singleton bean '" + beanName +
20                                    "' that is not fully initialized yet - a
consequence of a circular reference");
21                   }
22                   else {
23                       logger.debug("Returning cached instance of singleton bean
'" + beanName + "'");
24                   }
25               }
26               //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
27               bean = getObjectForBeanInstance(sharedInstance, name, beanName,
null);
28           }
29          ……
30      }
31     //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
32     protected Object getObjectForBeanInstance(
33             Object beanInstance, String name, String beanName,
RootBeanDefinition mbd) {
34         //容器已经得到了Bean实例对象，这个实例对象可能是一个普通的Bean，也可能是
35         //一个工厂Bean，如果是一个工厂Bean，则使用它创建一个Bean实例对象，如果
36         //调用本身就想获得一个容器的引用，则指定返回这个工厂Bean实例对象
37         //如果指定的名称是容器的解引用(dereference，即是对象本身而非内存地址)，
38         //且Bean实例也不是创建Bean实例对象的工厂Bean
39         if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance
instanceof FactoryBean)) {
40             throw new BeanIsNotAFactoryException(transformedBeanName(name),
beanInstance.getClass());
41         }
42         //如果Bean实例不是工厂Bean，或者指定名称是容器的解引用，调用者向获取对
43         //容器的引用，则直接返回当前的Bean实例
44         if (!(beanInstance instanceof FactoryBean) ||
BeanFactoryUtils.isFactoryDereference(name)) {
45             return beanInstance;
46         }
47     //处理指定名称不是容器的解引用，或者根据名称获取的Bean实例对象是一个工厂Bean
48     //使用工厂Bean创建一个Bean的实例对象
49         Object object = null;
50         if (mbd == null) {
51             //从Bean工厂缓存中获取给定名称的Bean实例对象
52             object = getCachedObjectForFactoryBean(beanName);
53         }
54         //让Bean工厂生产给定名称的Bean对象实例
55         if (object == null) {
```

```
56            FactoryBean factory = (FactoryBean) beanInstance;
57            //如果从Bean工厂生产的Bean是单态模式的，则缓存
58            if (mbd == null && containsBeanDefinition(beanName)) {
59                //从容器中获取指定名称的Bean定义，如果继承基类，则合并基类相关属性
60                mbd = getMergedLocalBeanDefinition(beanName);
61            }
62            //如果从容器得到Bean定义信息，并且Bean定义信息不是虚构的，则让工厂
63            //Bean生产Bean实例对象
64            boolean synthetic = (mbd != null && mbd.isSynthetic());
65            //调用FactoryBeanRegistrySupport类的getObjectFromFactoryBean
66            //方法，实现工厂Bean生产Bean对象实例的过程
67            object = getObjectFromFactoryBean(factory, beanName, !synthetic);

68        }
69        return object;
    }
```

在上面获取给定Bean的实例对象的getObjectForBeanInstance方法中，会调用FactoryBeanRegistrySupport类的getObjectFromFactoryBean方法，该方法实现了Bean工厂生产Bean实例对象。

Dereference(解引用)：一个在C/C++中应用比较多的术语，在C++中，”*”是解引用符号，而”&”是引用符号，解引用是指变量指向的是所引用对象的本身数据，而不是引用对象的内存地址。

(3)、AbstractBeanFactory生产Bean实例对象：

AbstractBeanFactory类中生产Bean实例对象的主要源码如下：

```
71    //Bean工厂生产Bean实例对象
72    protected Object getObjectFromFactoryBean(FactoryBean factory, String
beanName, boolean shouldPostProcess) {
73        //Bean工厂是单态模式，并且Bean工厂缓存中存在指定名称的Bean实例对象
74        if (factory.isSingleton() && containsSingleton(beanName)) {
75            //多线程同步，以防止数据不一致
76            synchronized (getSingletonMutex()) {
77                //直接从Bean工厂缓存中获取指定名称的Bean实例对象
78                Object object = this.factoryBeanObjectCache.get(beanName);
79                //Bean工厂缓存中没有指定名称的实例对象，则生产该实例对象
80                if (object == null) {
81                    //调用Bean工厂的getObject方法生产指定Bean的实例对象
82                    object = doGetObjectFromFactoryBean(factory, beanName,
shouldPostProcess);
```

```
83                    //将生产的实例对象添加到Bean工厂缓存中
84                    this.factoryBeanObjectCache.put(beanName, (object != null ?
object : NULL_OBJECT));
85                }
86                return (object != NULL_OBJECT ? object : null);
87            }
88        }
89        //调用Bean工厂的getObject方法生产指定Bean的实例对象
90        else {
91            return doGetObjectFromFactoryBean(factory, beanName,
shouldPostProcess);
92        }
93    }
94    //调用Bean工厂的getObject方法生产指定Bean的实例对象
95    private Object doGetObjectFromFactoryBean(
96            final FactoryBean factory, final String beanName, final boolean
shouldPostProcess)
97            throws BeanCreationException {
98        Object object;
99        try {
100            if (System.getSecurityManager() != null) {
101                AccessControlContext acc = getAccessControlContext();
102                try {
103                    //实现PrivilegedExceptionAction接口的匿名内置类
104                    //根据JVM检查权限，然后决定BeanFactory创建实例对象
105                    object = AccessController.doPrivileged(new
PrivilegedExceptionAction<Object>() {
106                        public Object run() throws Exception {
107                                //调用BeanFactory接口实现类的创建对象方法
108                                return factory.getObject();
109                            }
110                        }, acc);
111                }
112                catch (PrivilegedActionException pae) {
113                    throw pae.getException();
114                }
115            }
116            else {
117                //调用BeanFactory接口实现类的创建对象方法
118                object = factory.getObject();
119            }
120        }
121        catch (FactoryBeanNotInitializedException ex) {
122            throw new BeanCurrentlyInCreationException(beanName,
ex.toString());
123        }
124        catch (Throwable ex) {
125            throw new BeanCreationException(beanName, "FactoryBean threw
exception on object creation", ex);
126        }
127        //创建出来的实例对象为null，或者因为单态对象正在创建而返回null
128        if (object == null && isSingletonCurrentlyInCreation(beanName)) {
129            throw new BeanCurrentlyInCreationException(
130                    beanName, "FactoryBean which is currently in creation
returned null from getObject");
131        }
132        //为创建出来的Bean实例对象添加BeanPostProcessor后置处理器
133        if (object != null && shouldPostProcess) {
```

```
134             try {
135                 object = postProcessObjectFromFactoryBean(object, beanName);

136             }
137             catch (Throwable ex) {
138                 throw new BeanCreationException(beanName, "Post-processing of
the FactoryBean's object failed", ex);
139             }
140         }
141         return object;
    }
```

从上面的源码分析中，我们可以看出，BeanFactory接口调用其实现类的getObject方法来实现创建Bean实例对象的功能。

(4).工厂Bean的实现类getObject方法创建Bean实例对象：

FactoryBean的实现类有非常多，比如：Proxy、RMI、JNDI、ServletContextFactoryBean等等，FactoryBean接口为Spring容器提供了一个很好的封装机制，具体的getObject有不同的实现类根据不同的实现策略来具体提供，我们分析一个最简单的AnnotationTestFactoryBean的实现源码：

```
143    public class AnnotationTestBeanFactory implements
FactoryBean<IJmxTestBean> {
144         private final FactoryCreatedAnnotationTestBean instance = new
FactoryCreatedAnnotationTestBean();
145         public AnnotationTestBeanFactory() {
146             this.instance.setName("FACTORY");
147         }
148         //AnnotationTestBeanFactory产生Bean实例对象的实现
149         public IJmxTestBean getObject() throws Exception {
150             return this.instance;
151         }
152         public Class<? extends IJmxTestBean> getObjectType() {
153             return FactoryCreatedAnnotationTestBean.class;
154         }
155         public boolean isSingleton() {
156             return true;
157         }
     }
```

其他的Proxy，RMI，JNDI等等，都是根据相应的策略提供getObject的实现。这里不做一一分析，这已经不是Spring的核心功能，有需要的时候再去深入研究。

4.BeanPostProcessor后置处理器的实现：

BeanPostProcessor后置处理器是Spring IoC容器经常使用到的一个特性，这个Bean后置处理器是一个监听器，可以监听容器触发的Bean声明周期事件。后置处理器向容器注册以后，容器中管理的Bean就具备了接收IoC容器事件回调的能力。

BeanPostProcessor的使用非常简单，只需要提供一个实现接口BeanPostProcessor的实现类，然后在Bean的配置文件中设置即可。

(1).BeanPostProcessor的源码如下：

```
1    package org.springframework.beans.factory.config;
2    import org.springframework.beans.BeansException;
3    public interface BeanPostProcessor {
4        //为在Bean的初始化前提供回调入口
5        Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException;
6        //为在Bean的初始化之后提供回调入口
7        Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException;
     }
```

这两个回调的入口都是和容器管理的Bean的生命周期事件紧密相关，可以为用户提供在Spring IoC容器初始化Bean过程中自定义的处理操作。

(2).AbstractAutowireCapableBeanFactory类对容器生成的Bean添加后置处理器：

BeanPostProcessor后置处理器的调用发生在Spring IoC容器完成对Bean实例对象的创建和属性的依赖注入完成之后，在对Spring依赖注入的源码分析过程中我们知道，当应用程序第一次调用getBean方法(lazy-init预实例化除外)向Spring IoC容器索取指定Bean时触发Spring IoC容器创建Bean实例对象并进行依赖注入的过程，其中真正实现创建Bean对象并进行依赖注入的方法是AbstractAutowireCapableBeanFactory类的doCreateBean方法，主要源码如下：

```
1    //真正创建Bean的方法
2    protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final Object[] args) {
3        //创建Bean实例对象
4        ……
5        try {
6            //对Bean属性进行依赖注入
7            populateBean(beanName, mbd, instanceWrapper);
8            if (exposedObject != null) {
9                //在对Bean实例对象生成和依赖注入完成以后，开始对Bean实例对象
10               //进行初始化 ，为Bean实例对象应用BeanPostProcessor后置处理器
11               exposedObject = initializeBean(beanName, exposedObject, mbd);
12            }
13       }
14       catch (Throwable ex) {
15           if (ex instanceof BeanCreationException &&
beanName.equals(((BeanCreationException) ex).getBeanName())) {
16                throw (BeanCreationException) ex;
17           }
18       ……
19       //为应用返回所需要的实例对象
20       return exposedObject;
    }
```

从上面的代码中我们知道，为Bean实例对象添加BeanPostProcessor后置处理器的入口的是initializeBean方法。

(3).initializeBean方法为容器产生的Bean实例对象添加BeanPostProcessor后置处理器：

同样在AbstractAutowireCapableBeanFactory类中，initializeBean方法实现为容器创建的Bean实例对象添加BeanPostProcessor后置处理器，源码如下：

```
1    //初始容器创建的Bean实例对象，为其添加BeanPostProcessor后置处理器
2    protected Object initializeBean(final String beanName, final Object bean,
RootBeanDefinition mbd) {
3        //JDK的安全机制验证权限
4        if (System.getSecurityManager() != null) {
5            //实现PrivilegedAction接口的匿名内部类
6            AccessController.doPrivileged(new PrivilegedAction<Object>() {
7                public Object run() {
8                    invokeAwareMethods(beanName, bean);
9                    return null;
10                }
11            }, getAccessControlContext());
12        }
13        else {
14            //为Bean实例对象包装相关属性，如名称，类加载器，所属容器等信息
15            invokeAwareMethods(beanName, bean);
16        }
17        Object wrappedBean = bean;
18        //对BeanPostProcessor后置处理器的postProcessBeforeInitialization
19        //回调方法的调用，为Bean实例初始化前做一些处理
20        if (mbd == null || !mbd.isSynthetic()) {
21            wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
22        }
23        //调用Bean实例对象初始化的方法，这个初始化方法是在Spring Bean定义配置
24        //文件中通过init-method属性指定的
25        try {
26            invokeInitMethods(beanName, wrappedBean, mbd);
27        }
28        catch (Throwable ex) {
29            throw new BeanCreationException(
30                    (mbd != null ? mbd.getResourceDescription() : null),
31                    beanName, "Invocation of init method failed", ex);
32        }
33        //对BeanPostProcessor后置处理器的postProcessAfterInitialization
34        //回调方法的调用，为Bean实例初始化之后做一些处理
35        if (mbd == null || !mbd.isSynthetic()) {
36            wrappedBean =
applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
37        }
38        return wrappedBean;
```

```
39    }
40    //调用BeanPostProcessor后置处理器实例对象初始化之前的处理方法
41    public Object applyBeanPostProcessorsBeforeInitialization(Object
existingBean, String beanName)
42            throws BeansException {
43        Object result = existingBean;
44        //遍历容器为所创建的Bean添加的所有BeanPostProcessor后置处理器
45        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
46            //调用Bean实例所有的后置处理中的初始化前处理方法，为Bean实例对象在
47            //初始化之前做一些自定义的处理操作
48            result = beanProcessor.postProcessBeforeInitialization(result,
beanName);
49            if (result == null) {
50                return result;
51            }
52        }
53        return result;
54    }
55    //调用BeanPostProcessor后置处理器实例对象初始化之后的处理方法
56    public Object applyBeanPostProcessorsAfterInitialization(Object
existingBean, String beanName)
57            throws BeansException {
58        Object result = existingBean;
59        //遍历容器为所创建的Bean添加的所有BeanPostProcessor后置处理器
60        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
61            //调用Bean实例所有的后置处理中的初始化后处理方法，为Bean实例对象在
62            //初始化之后做一些自定义的处理操作
63            result = beanProcessor.postProcessAfterInitialization(result,
beanName);
64            if (result == null) {
65                return result;
66            }
67        }
68        return result;
    }
```

BeanPostProcessor是一个接口，其初始化前的操作方法和初始化后的操作方法均委托其实现子类来实现，在Spring中，BeanPostProcessor的实现子类非常的多，分别完成不同的操作，如：AOP面向切面编程的注册通知适配器、Bean对象的数据校验、Bean继承属性/方法的合并等等，我们以最简单的AOP切面织入来简单了解其主要的功能。

(4).AdvisorAdapterRegistrationManager在Bean对象初始化后注册通知适配器：

AdvisorAdapterRegistrationManager是BeanPostProcessor的一个实现类，其主要的作用为容器中管理的Bean注册一个面向切面编程的通知适配器，以便在Spring容器为所管理的Bean进行面向切面编程时提供方便，其源码如下：

```
1 //为容器中管理的Bean注册一个面向切面编程的通知适配器
2 public class AdvisorAdapterRegistrationManager implements BeanPostProcessor {

3    //容器中负责管理切面通知适配器注册的对象
4    private AdvisorAdapterRegistry advisorAdapterRegistry =
GlobalAdvisorAdapterRegistry.getInstance();
5    public void setAdvisorAdapterRegistry(AdvisorAdapterRegistry
advisorAdapterRegistry) {
6        this.advisorAdapterRegistry = advisorAdapterRegistry;
7    }
8    //BeanPostProcessor在Bean对象初始化前的操作
9    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
10        //没有做任何操作，直接返回容器创建的Bean对象
11        return bean;
12    }
13    //BeanPostProcessor在Bean对象初始化后的操作
14    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
15        if (bean instanceof AdvisorAdapter){
16            //如果容器创建的Bean实例对象是一个切面通知适配器，则向容器的注册
this.advisorAdapterRegistry.registerAdvisorAdapter((AdvisorAdapter) bean);
17        }
18        return bean;
19    }
}
```

其他的BeanPostProcessor接口实现类的也类似，都是对Bean对象使用到的一些特性进行处理，或者向IoC容器中注册，为创建的Bean实例对象做一些自定义的功能增加，这些操作是容器初始化Bean时自动触发的，不需要认为的干预。

5.Spring IoC容器autowiring实现原理：

Spring IoC容器提供了两种管理Bean依赖关系的方式：

a. 显式管理：通过BeanDefinition的属性值和构造方法实现Bean依赖关系管理。

b. autowiring：Spring IoC容器的依赖自动装配功能，不需要对Bean属性的依赖关系做显式的声明，只需要在配置好autowiring属性，IoC容器会自动使用反射查找属性的类型和名称，然后基于属性的类型或者名称来自动匹配容器中管理的Bean，从而自动地完成依赖注入。

通过对autowiring自动装配特性的理解，我们知道容器对Bean的自动装配发生在容器对Bean依赖注入的过程中。在前面对Spring IoC容器的依赖注入过程源码分析中，我们已经知道了容器对Bean实例对象的属性注入的处理发生在AbstractAutoWireCapableBeanFactory类中的populateBean方法中，我们通过程序流程分析autowiring的实现原理：

(1). AbstractAutoWireCapableBeanFactory对Bean实例进行属性依赖注入：

应用第一次通过getBean方法(配置了lazy-init预实例化属性的除外)向IoC容器索取Bean时，容器创建Bean实例对象，并且对Bean实例对象进行属性依赖注入，AbstractAutoWireCapableBeanFactory的populateBean方法就是实现Bean属性依赖注入的功能，其主要源码如下：

```
1    protected void populateBean(String beanName, AbstractBeanDefinition mbd,
BeanWrapper bw) {
2        //获取Bean定义的属性值，并对属性值进行处理
3        PropertyValues pvs = mbd.getPropertyValues();
4        ……
5        //对依赖注入处理，首先处理autowiring自动装配的依赖注入
6        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME
||
7                mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
8            MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
9            //根据Bean名称进行autowiring自动装配处理
10            if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME) {
11                autowireByName(beanName, mbd, bw, newPvs);
12            }
13            //根据Bean类型进行autowiring自动装配处理
14            if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
15                autowireByType(beanName, mbd, bw, newPvs);
16            }
17        }
18        //对非autowiring的属性进行依赖注入处理
19         ……
    }
```

(2).Spring IoC容器根据Bean名称或者类型进行autowiring自动依赖注入：

```
1    //根据名称对属性进行自动依赖注入
2    protected void autowireByName(
3            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw,
MutablePropertyValues pvs) {
4         //对Bean对象中非简单属性(不是简单继承的对象，如8中原始类型，字符串，URL等//都是简
单属性)进行处理
5        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
6        for (String propertyName : propertyNames) {
7            //如果Spring IoC容器中包含指定名称的Bean
8            if (containsBean(propertyName)) {
```

```
9                 //调用getBean方法向IoC容器索取指定名称的Bean实例，迭代触发属性的//初始化
和依赖注入
10                Object bean = getBean(propertyName);
11                //为指定名称的属性赋予属性值
12                pvs.add(propertyName, bean);
13                //指定名称属性注册依赖Bean名称，进行属性依赖注入
14                registerDependentBean(propertyName, beanName);
15                if (logger.isDebugEnabled()) {
16                    logger.debug("Added autowiring by name from bean name '" +
beanName +
17                            "' via property '" + propertyName + "' to bean
named '" + propertyName + "'");
18                }
19            }
20            else {
21                if (logger.isTraceEnabled()) {
22                    logger.trace("Not autowiring property '" + propertyName +
"' of bean '" + beanName +
23                            "' by name: no matching bean found");
24                }
25            }
26        }
27    }
28    //根据类型对属性进行自动依赖注入
29    protected void autowireByType(
30            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw,
MutablePropertyValues pvs) {
31        //获取用户定义的类型转换器
32        TypeConverter converter = getCustomTypeConverter();
33        if (converter == null) {
34            converter = bw;
35        }
36        //存放解析的要注入的属性
37        Set<String> autowiredBeanNames = new LinkedHashSet<String>(4);
38          //对Bean对象中非简单属性(不是简单继承的对象，如8中原始类型，字符
39         //URL等都是简单属性)进行处理
40        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
41        for (String propertyName : propertyNames) {
42            try {
43                //获取指定属性名称的属性描述器
44                PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);

45                //不对Object类型的属性进行autowiring自动依赖注入
46                if (!Object.class.equals(pd.getPropertyType())) {
47                    //获取属性的setter方法
48                    MethodParameter methodParam =
BeanUtils.getWriteMethodParameter(pd);
49                    //检查指定类型是否可以被转换为目标对象的类型
50                    boolean eager =
!PriorityOrdered.class.isAssignableFrom(bw.getWrappedClass());
51                    //创建一个要被注入的依赖描述
52                    DependencyDescriptor desc = new
AutowireByTypeDependencyDescriptor(methodParam, eager);
53                    //根据容器的Bean定义解析依赖关系，返回所有要被注入的Bean对象
54                    Object autowiredArgument = resolveDependency(desc,
beanName, autowiredBeanNames, converter);
55                    if (autowiredArgument != null) {
56                        //为属性赋值所引用的对象
```

```
57                          pvs.add(propertyName, autowiredArgument);
58                      }
59                      for (String autowiredBeanName : autowiredBeanNames) {
60                          //指定名称属性注册依赖Bean名称，进行属性依赖注入
61                          registerDependentBean(autowiredBeanName, beanName);
62                          if (logger.isDebugEnabled()) {
63                              logger.debug("Autowiring by type from bean name '"
+ beanName + "' via property '" +
64                                      propertyName + "' to bean named '" +
autowiredBeanName + "'");
65                          }
66                      }
67                      //释放已自动注入的属性
68                      autowiredBeanNames.clear();
69                  }
70              }
71              catch (BeansException ex) {
72                  throw new
UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName,
propertyName, ex);
73              }
74          }
    }
```

通过上面的源码分析，我们可以看出来通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是DefaultSingletonBeanRegistry类的registerDependentBean方法。

(3).DefaultSingletonBeanRegistry的registerDependentBean方法对属性注入：

```
1    //为指定的Bean注入依赖的Bean
2    public void registerDependentBean(String beanName, String dependentBeanName)
{
3        //处理Bean名称，将别名转换为规范的Bean名称
4        String canonicalName = canonicalName(beanName);
5        //多线程同步，保证容器内数据的一致性
6        //先从容器中：bean名称-->全部依赖Bean名称集合找查找给定名称Bean的依赖Bean
7        synchronized (this.dependentBeanMap) {
8            //获取给定名称Bean的所有依赖Bean名称
9            Set<String> dependentBeans =
this.dependentBeanMap.get(canonicalName);
10            if (dependentBeans == null) {
11                //为Bean设置依赖Bean信息
12                dependentBeans = new LinkedHashSet<String>(8);
13                this.dependentBeanMap.put(canonicalName, dependentBeans);
14            }
15            //向容器中：bean名称-->全部依赖Bean名称集合添加Bean的依赖信息
16            //即，将Bean所依赖的Bean添加到容器的集合中
17            dependentBeans.add(dependentBeanName);
18        }
19          //从容器中：bean名称-->指定名称Bean的依赖Bean集合找查找给定名称
20         //Bean的依赖Bean
```

```
21        synchronized (this.dependenciesForBeanMap) {
22            Set<String> dependenciesForBean =
this.dependenciesForBeanMap.get(dependentBeanName);
23            if (dependenciesForBean == null) {
24                dependenciesForBean = new LinkedHashSet<String>(8);
25                this.dependenciesForBeanMap.put(dependentBeanName,
dependenciesForBean);
26            }
27            //向容器中：bean名称-->指定Bean的依赖Bean名称集合添加Bean的依赖信息
28            //即，将Bean所依赖的Bean添加到容器的集合中
29            dependenciesForBean.add(canonicalName);
30        }
    }
```

通过对autowiring的源码分析，我们可以看出，autowiring的实现过程：

a. 对Bean的属性迭代调用getBean方法，完成依赖Bean的初始化和依赖注入。

b. 将依赖Bean的属性引用设置到被依赖的Bean属性上。

c. 将依赖Bean的名称和被依赖Bean的名称存储在IoC容器的集合中。

Spring IoC容器的autowiring属性自动依赖注入是一个很方便的特性，可以简化开发时的配置，但是凡是都有两面性，自动属性依赖注入也有不足，首先，Bean的依赖关系在配置文件中无法很清楚地看出来，对于维护造成一定困难。其次，由于自动依赖注入是Spring容器自动执行的，容器是不会智能判断的，如果配置不当，将会带来无法预料的后果，所以自动依赖注入特性在使用时还是综合考虑。

spring AOP的原理

Spring AOP 实现原理与 CGLIB 应用

AOP（Aspect Orient Programming），作为面向对象编程的一种补充，广泛应用于处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等。AOP 实现的关键就在于 AOP 框架自动创建的 AOP 代理，AOP 代理则可分为静态代理和动态代理两大类，其中静态代理是指使用 AOP 框架提供的命令进行编译，从而在编译阶段就可生成 AOP 代理类，因此也称为编译时增强；而动态代理则在运行时借助于 JDK 动态代理、CGLIB 等在内存中"临时"生成 AOP 动态代理类，因此也被称为运行时增强。

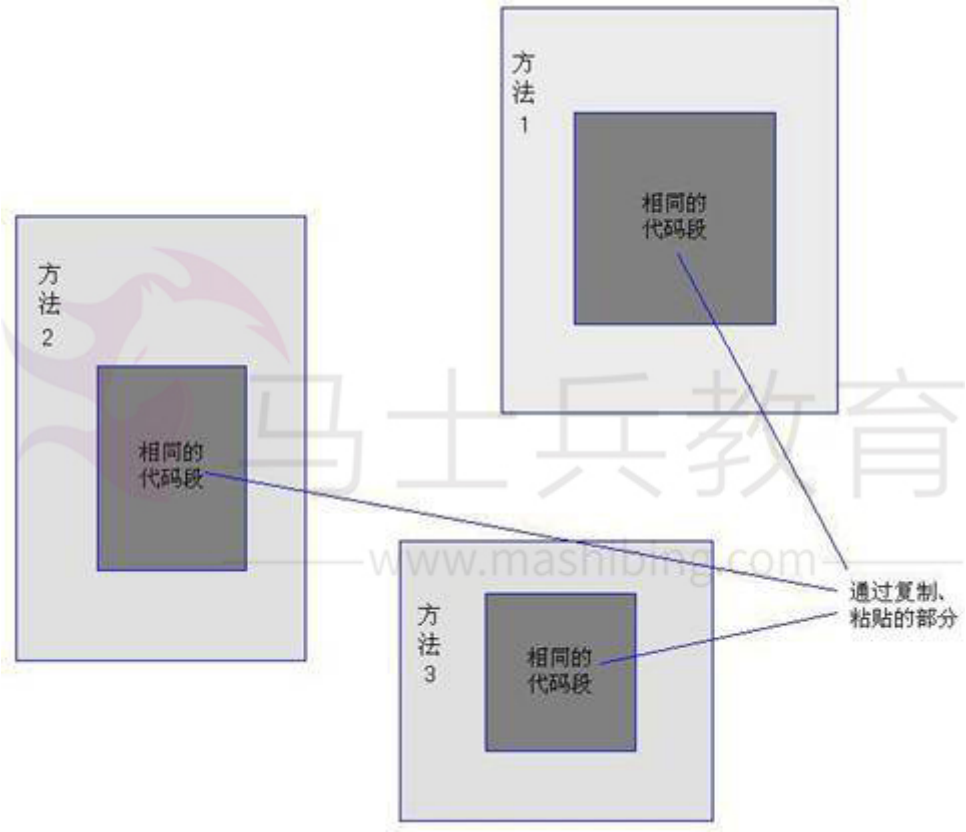
AOP 的存在价值

在传统 OOP 编程里以对象为核心，整个软件系统由系列相互依赖的对象所组成，而这些对象将被抽象成一个一个的类，并允许使用类继承来管理类与类之间一般到特殊的关系。随着软件规模的增大，应用的逐渐升级，慢慢出现了一些 OOP 很难解决的问题。

我们可以通过分析、抽象出一系列具有一定属性与行为的对象，并通过这些对象之间的协作来形成一个完整的软件功能。由于对象可以继承，因此我们可以把具有相同功能或相同特性的属性抽象到一个层次分明的类结构体系中。随着软件规范的不断扩大，专业化分工越来越系列，以及 OOP 应用实践的不断增多，随之也暴露出了一些 OOP 无法很好解决的问题。

现在假设系统中有 3 段完全相似的代码，这些代码通常会采用”复制”、”粘贴”方式来完成，通过这种”复制”、”粘贴”方式开发出来的软件如图 1 所示。

图 1.多个地方包含相同代码的软件

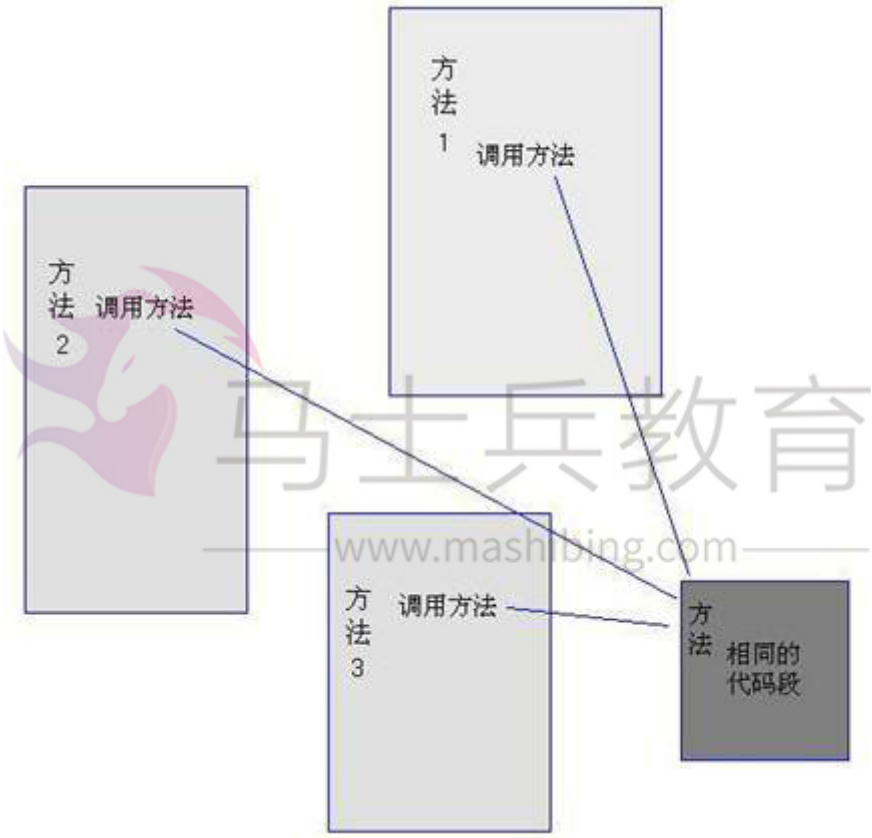


马士兵教育 www.mashibing.com

看到如图 1 所示的示意图，可能有的读者已经发现了这种做法的不足之处：如果有一天，图 1 中的深色代码段需要修改，那是不是要打开 3 个地方的代码进行修改？如果不是 3 个地方包含这段代码，而是 100 个地方，甚至是 1000 个地方包含这段代码段，那会是什么后果？

为了解决这个问题，我们通常会采用将如图 1 所示的深色代码部分定义成一个方法，然后在 3 个代码段中分别调用该方法即可。在这种方式下，软件系统的结构如图 2 所示。

图 2 通过方法调用实现系统功能



马士兵教育
www.mashibing.com

对于如图 2 所示的软件系统，如果需要修改深色部分的代码，只要修改一个地方即可，不管整个系统中有多少地方调用了该方法，程序无须修改这些地方，只需修改被调用的方法即可——通过这种方式，大大降低了软件后期维护的复杂度。

对于如图 2 所示的方法 1、方法 2、方法 3 依然需要显式调用深色方法，这样做能够解决大部分应用场景。但对于一些更特殊的情况：应用需要方法 1、方法 2、方法 3 彻底与深色方法分离——方法 1、方法 2、方法 3 无须直接调用深色方法，那如何解决？

因为软件系统需求变更是很频繁的事情，系统前期设计方法 1、方法 2、方法 3 时只实现了核心业务功能，过了一段时间，我们需要为方法 1、方法 2、方法 3 都增加事务控制；又过了一段时间，客户提出方法 1、方法 2、方法 3 需要进行用户合法性验证，只有合法的用户才能执行这些方法；又过了一段时间，客户又提出方法 1、方法 2、方法 3 应该增加日志记录；又过了一段时间，客户又提出……面对这样的情况，我们怎么办？通常有两种做法：

- 根据需求说明书，直接拒绝客户要求。
- 拥抱需求，满足客户的需求。

第一种做法显然不好，客户是上帝，我们应该尽量满足客户的需求。通常会采用第二种做法，那如何解决呢？是不是每次先定义一个新方法，然后修改方法 1、方法 2、方法 3，增加调用新方法？这样做的工作量也不小啊！我们希望有一种特殊的方法：我们只要定义该方法，无须在方法 1、方法 2、方法 3 中显式调用它，系统会”自动”执行该特殊方法。

上面想法听起来很神奇，甚至有一些不切实际，但其实是完全可以实现的，实现这个需求的技术就是 AOP。AOP 专门用于处理系统中分布于各个模块（不同方法）中的交叉关注点的问题，在 Java EE 应用中，常常通过 AOP 来处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等，AOP 已经成为一种非常常用的解决方案。

使用 AspectJ 的编译时增强进行 AOP

AspectJ 是一个基于 Java 语言的 AOP 框架，提供了强大的 AOP 功能，其他很多 AOP 框架都借鉴或采纳其中的一些思想。

AspectJ 是 Java 语言的一个 AOP 实现，其主要包括两个部分：第一个部分定义了如何表达、定义 AOP 编程中的语法规范，通过这套语言规范，我们可以方便地用 AOP 来解决 Java 语言中存在的交叉关注点问题；另一个部分是工具部分，包括编译器、调试工具等。

AspectJ 是最早、功能比较强大的 AOP 实现之一，对整套 AOP 机制都有较好的实现，很多其他语言的 AOP 实现，也借鉴或采纳了 AspectJ 中很多设计。在 Java 领域，AspectJ 中的很多语法结构基本上已成为 AOP 领域的标准。

下载、安装 AspectJ 比较简单，读者登录 AspectJ 官网，即可下载到一个可执行的 JAR 包，使用 java -jar aspectj-1.x.x.jar 命令、多次单击”Next”按钮即可成功安装 AspectJ。

成功安装了 AspectJ 之后，将会在 E:\Java\AOP\aspectj1.6 路径下（AspectJ 的安装路径）看到如下文件结构：

- bin：该路径下存放了 aj、aj5、ajc、ajdoc、ajbrowser 等命令，其中 ajc 命令最常用，它的作用类似于 javac，用于对普通 Java 类进行编译时增强。
- docs：该路径下存放了 AspectJ 的使用说明、参考手册、API 文档等文档。
- lib：该路径下的 4 个 JAR 文件是 AspectJ 的核心类库。
- 相关授权文件。

一些文档、AspectJ 入门书籍，一谈到使用 AspectJ，就认为必须使用 Eclipse 工具，似乎离开了该工具就无法使用 AspectJ 了。

虽然 AspectJ 是 Eclipse 基金组织的开源项目，而且提供了 Eclipse 的 AJDT 插件（AspectJ Development Tools）来开发 AspectJ 应用，但 AspectJ 绝对无须依赖于 Eclipse 工具。

实际上，AspectJ 的用法非常简单，就像我们使用 JDK 编译、运行 Java 程序一样。下面通过一个简单的程序来示范 AspectJ 的用法，并分析 AspectJ 如何在编译时进行增强。

首先编写一个简单的 Java 类，这个 Java 类用于模拟一个业务组件。

清单 1.Hello.java

```
public class Hello
 {
 // 定义一个简单方法，模拟应用中的业务逻辑方法
 public void sayHello(){System.out.println("Hello AspectJ!");}
 // 主方法，程序的入口
 public static void main(String[] args)
 {
 Hello h = new Hello();
 h.sayHello();
 }
 }
```

显示更多

上面 Hello 类模拟了一个业务逻辑组件，编译、运行该 Java 程序，这个结果是没有任何悬念的，程序将在控制台打印”Hello AspectJ”字符串。

假设现在客户需要在执行 sayHello() 方法之前启动事务，当该方法执行结束时关闭事务，在传统编程模式下，我们必须手动修改 sayHello() 方法——如果改为使用 AspectJ，则可以无须修改上面的 sayHello() 方法。

下面我们定义一个特殊的 Java 类。

清单 2.TxAspect.java

```
public aspect TxAspect
 {
 // 指定执行 Hello.sayHello() 方法时执行下面代码块
 void around():call(void Hello.sayHello()){System.out.println("开始事务
...");proceed();System.out.println("事务结束 ...");}
 }
```

显示更多

可能读者已经发现了，上面类文件中不是使用 class、interface、enum 在定义 Java 类，而是使用了 aspect ——难道 Java 语言又新增了关键字？没有！上面的 TxAspect 根本不是一个 Java 类，所以 aspect 也不是 Java 支持的关键字，它只是 AspectJ 才能识别的关键字。

上面粗体字代码也不是方法，它只是指定当程序执行 Hello 对象的 sayHello() 方法时，系统将改为执行粗体字代码的花括号代码块，其中 proceed() 代表回调原来的 sayHello() 方法。

正如前面提到的，Java 无法识别 TxAspect.java 文件的内容，所以我们要使用 ajc.exe 命令来编译上面的 Java 程序。为了能在命令行使用 ajc.exe 命令，需要把 AspectJ 安装目录下的 bin 路径（比如 E:\Java\AOP\aspectj1.6\bin 目录）添加到系统的 PATH 环境变量中。接下来执行如下命令进行编译：

ajc -d . Hello.java TxAspect.java

我们可以把 ajc.exe 理解成 javac.exe 命令，都用于编译 Java 程序，区别是 ajc.exe 命令可识别 AspectJ 的语法；从这个意义上看，我们可以将 ajc.exe 当成一个增强版的 javac.exe 命令。

运行该 Hello 类依然无须任何改变，因为 Hello 类位于 lee 包下。程序使用如下命令运行 Hello 类：

java lee.Hello

运行该程序，将看到一个令人惊喜的结果：

开始事务 ...

Hello AspectJ!

事务结束 ...

从上面运行结果来看，我们完全可以不对 Hello.java 类进行任何修改，同时又可以满足客户的需求：上面程序只是在控制台打印”开始事务 ...”、”结束事务 ...”来模拟了事务操作，实际上我们可用实际的事务操作代码来代替这两行简单的语句，这就可以满足客户需求了。

如果客户再次提出新需求，需要在 sayHello() 方法后增加记录日志的功能，那也很简单，我们再定义一个 LogAspect，程序如下：

清单 3.LogAspect.java

```
public aspect LogAspect
 {
 // 定义一个 PointCut，其名为 logPointcut
 // 该 PointCut 对应于指定 Hello 对象的 sayHello 方法
     pointcut logPointcut()
 :execution(void Hello.sayHello());
 // 在 logPointcut 之后执行下面代码块
     after():logPointcut()
     {
 System.out.println("记录日志 ...");
     }
 }
```

显示更多

上面程序的粗体字代码定义了一个 Pointcut：logPointcut – 等同于执行 Hello 对象的 sayHello() 方法，并指定在 logPointcut 之后执行简单的代码块，也就是说，在 sayHello() 方法之后执行指定代码块。使用如下命令来编译上面的 Java 程序：

ajc -d . *.java

再次运行 Hello 类，将看到如下运行结果：

开始事务 ...

Hello AspectJ!

记录日志 ...

事务结束 ...

从上面运行结果来看，通过使用 AspectJ 提供的 AOP 支持，我们可以为 sayHello() 方法不断增加新功能。

为什么在对 Hello 类没有任何修改的前提下，而 Hello 类能不断地、动态增加新功能呢？这看上去并不符合 Java 基本语法规则啊。实际上我们可以使用 Java 的反编译工具来反编译前面程序生成的 Hello.class 文件，发现 Hello.class 文件的代码如下：

清单 4.Hello.class

```
package lee;

 import java.io.PrintStream;
 import org.aspectj.runtime.internal.AroundClosure;

 public class Hello
 {
  public void sayHello()
  {
    try
    {
      System.out.println("Hello AspectJ!"); } catch (Throwable localThrowable) {
      LogAspect.aspectOf().ajc$after$lee_LogAspect$1$9fd5dd97(); throw
localThrowable; }
      LogAspect.aspectOf().ajc$after$lee_LogAspect$1$9fd5dd97();
  }

  ...

  private static final void sayHello_aroundBody1$advice(Hello target,
            TxAspect ajc$aspectInstance, AroundClosure ajc$aroundClosure)
  {
    System.out.println("开始事务 ...");
    AroundClosure localAroundClosure = ajc$aroundClosure;
sayHello_aroundBody0(target);
    System.out.println("事务结束 ...");
  }
 }
```

显示更多

不难发现这个 Hello.class 文件不是由原来的 Hello.java 文件编译得到的，该 Hello.class 里新增了很多内容——这表明 AspectJ 在编译时"自动"编译得到了一个新类，这个新类增强了原有的 Hello.java 类的功能，因此 AspectJ 通常被称为编译时增强的 AOP 框架。

提示：与 AspectJ 相对的还有另外一种 AOP 框架，它们不需要在编译时对目标类进行增强，而是运行时生成目标类的代理类，该代理类要么与目标类实现相同的接口，要么是目标类的子类——总之，代理类的实例可作为目标类的实例来使用。一般来说，编译时增强的 AOP 框架在性能上更有优势——因为运行时动态增强的 AOP 框架需要每次运行时都进行动态增强。

实际上，AspectJ 允许同时为多个方法添加新功能，只要我们定义 Pointcut 时指定匹配更多的方法即可。如下片段：

```
pointcut xxxPointcut()
     :execution(void H*.say*());
```

显示更多

上面程序中的 xxxPointcut 将可以匹配所有以 H 开头的类中、所有以 say 开头的方法，但该方法返回的必须是 void；如果不想匹配任意的返回值类型，则可将代码改为如下形式：

pointcut xxxPointcut()

:execution(*H*.say*());

关于如何定义 AspectJ 中的 Aspect、Pointcut 等，读者可以参考 AspectJ 安装路径下的 doc 目录里的 quick5.pdf 文件。

使用 Spring AOP

与 AspectJ 相同的是，Spring AOP 同样需要对目标类进行增强，也就是生成新的 AOP 代理类；与 AspectJ 不同的是，Spring AOP 无需使用任何特殊命令对 Java 源代码进行编译，它采用运行时动态地、在内存中临时生成"代理类"的方式来生成 AOP 代理。

Spring 允许使用 AspectJ Annotation 用于定义方面（Aspect）、切入点（Pointcut）和增强处理（Advice），Spring 框架则可识别并根据这些 Annotation 来生成 AOP 代理。Spring 只是使用了和 AspectJ 5 一样的注解，但并没有使用 AspectJ 的编译器或者织入器（ *Weaver* ），底层依然使用的是 Spring AOP，依然是在运行时动态生成 AOP 代理，并不依赖于 AspectJ 的编译器或者织入器。

简单地说，Spring 依然采用运行时生成动态代理的方式来增强目标对象，所以它不需要增加额外的编译，也不需要 AspectJ 的织入器支持；而 AspectJ 在采用编译时增强，所以 AspectJ 需要使用自己的编译器来编译 Java 文件，还需要织入器。

为了启用 Spring 对 @AspectJ 方面配置的支持，并保证 Spring 容器中的目标 Bean 被一个或多个方面自动增强，必须在 Spring 配置文件中配置如下片段：

```
<?xml version="1.0" encoding="GBK"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/aop
 http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
 <!-- 启动 @AspectJ 支持 -->
 <aop:aspectj-autoproxy/>
 </beans>
```

显示更多

当然，如果我们希望完全启动 Spring 的”零配置”功能，则还需要启用 Spring 的”零配置”支持，让 Spring 自动搜索指定路径下 Bean 类。

所谓自动增强，指的是 Spring 会判断一个或多个方面是否需要对指定 Bean 进行增强，并据此自动生成相应的代理，从而使得增强处理在合适的时候被调用。

如果不打算使用 Spring 的 XML Schema 配置方式，则应该在 Spring 配置文件中增加如下片段来启用 @AspectJ 支持。

```
<!-- 启动 @AspectJ 支持 -->
 <bean class="org.springframework.aop.aspectj.annotation.
     AnnotationAwareAspectJAutoProxyCreator"/>
```

显示更多

上面配置文件中的 AnnotationAwareAspectJAutoProxyCreator 是一个 Bean 后处理器（BeanPostProcessor），该 Bean 后处理器将会为容器中 Bean 生成 AOP 代理，

当启动了 @AspectJ 支持后，只要我们在 Spring 容器中配置一个带 @Aspect 注释的 Bean，Spring 将会自动识别该 Bean，并将该 Bean 作为方面 Bean 处理。

在 Spring 容器中配置方面 Bean（即带 @Aspect 注释的 Bean），与配置普通 Bean 没有任何区别，一样使用 元素进行配置，一样支持使用依赖注入来配置属性值；如果我们启动了 Spring 的”零配置”特性，一样可以让 Spring 自动搜索，并装载指定路径下的方面 Bean。

使用 @Aspect 标注一个 Java 类，该 Java 类将会作为方面 Bean，如下面代码片段所示：

```
// 使用 @Aspect 定义一个方面类
 @Aspect
 public class LogAspect
 {
 // 定义该类的其他内容
 ...
 }
```

显示更多

方面类（用 @Aspect 修饰的类）和其他类一样可以有方法、属性定义，还可能包括切入点、增强处理定义。

当我们使用 @Aspect 来修饰一个 Java 类之后，Spring 将不会把该 Bean 当成组件 Bean 处理，因此负责自动增强的后处理 Bean 将会略过该 Bean，不会对该 Bean 进行任何增强处理。

开发时无须担心使用 @Aspect 定义的方面类被增强处理，当 Spring 容器检测到某个 Bean 类使用了 @Aspect 标注之后，Spring 容器不会对该 Bean 类进行增强。

下面将会考虑采用 Spring AOP 来改写前面介绍的例子：

下面例子使用一个简单的 Chinese 类来模拟业务逻辑组件：

清单 5.Chinese.java

```
@Component
 public class Chinese
 {
 // 实现 Person 接口的 sayHello() 方法
     public String sayHello(String name)
```

```
    {
   System.out.println("-- 正在执行 sayHello 方法 --");
// 返回简单的字符串
        return name + " Hello , Spring AOP";
    }
// 定义一个 eat() 方法
    public void eat(String food)
    {
   System.out.println("我正在吃 :"+ food);
    }
}
```

显示更多

提供了上面 Chinese 类之后，接下来假设同样需要为上面 Chinese 类的每个方法增加事务控制、日志记录，此时可以考虑使用 Around、AfterReturning 两种增强处理。

先看 AfterReturning 增强处理代码。

清单 6.AfterReturningAdviceTest.java

```
// 定义一个方面
@Aspect
public class AfterReturningAdviceTest
{
// 匹配 org.crazyit.app.service.impl 包下所有类的、
// 所有方法的执行作为切入点
@AfterReturning(returning="rvt",
pointcut="execution(* org.crazyit.app.service.impl.*.*(..))")
public void log(Object rvt)
{
System.out.println("获取目标方法返回值 :" + rvt);
System.out.println("模拟记录日志功能 ...");
}
}
```

显示更多

上面 Aspect 类使用了 @Aspect 修饰，这样 Spring 会将它当成一个方面 Bean 进行处理。其中程序中粗体字代码指定将会在调用 org.crazyit.app.service.impl 包下的所有类的所有方法之后织入 log(Object rvt) 方法。

再看 Around 增强处理代码：

清单 7.AfterReturningAdviceTest.java

```
// 定义一个方面
@Aspect
public class AroundAdviceTest
{
// 匹配 org.crazyit.app.service.impl 包下所有类的、
// 所有方法的执行作为切入点
@Around("execution(* org.crazyit.app.service.impl.*.*(..))")
public Object processTx(ProceedingJoinPoint jp)
throws java.lang.Throwable
{
System.out.println("执行目标方法之前，模拟开始事务 ...");
// 执行目标方法，并保存目标方法执行后的返回值
```

```
Object rvt = jp.proceed(new String[]{"被改变的参数"});
System.out.println("执行目标方法之后，模拟结束事务 ...");
return rvt + " 新增的内容";
}
}
```

显示更多

与前面的 AfterReturning 增强处理类似的，此处同样使用了 @Aspect 来修饰前面 Bean，其中粗体字代码指定在调用 org.crazyit.app.service.impl 包下的所有类的所有方法的”前后（Around）” 织入 processTx(ProceedingJoinPoint jp) 方法

需要指出的是，虽然此处只介绍了 Spring AOP 的 AfterReturning、Around 两种增强处理，但实际上 Spring 还支持 Before、After、AfterThrowing 等增强处理，关于 Spring AOP 编程更多、更细致的编程细节，可以参考《轻量级 Java EE 企业应用实战》一书。

本示例采用了 Spring 的零配置来开启 Spring AOP，因此上面 Chinese 类使用了 @Component 修饰，而方面 Bean 则使用了 @Aspect 修饰，方面 Bean 中的 Advice 则分别使用了 @AfterReturning、@Around 修饰。接下来只要为 Spring 提供如下配置文件即可：

清单 8.bean.xml

```
<?xml version="1.0" encoding="GBK"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context-3.0.xsd
 http://www.springframework.org/schema/aop
 http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
 <!-- 指定自动搜索 Bean 组件、自动搜索方面类 -->
 <context:component-scan base-package="org.crazyit.app.service
 ,org.crazyit.app.advice">
 <context:include-filter type="annotation"
 expression="org.aspectj.lang.annotation.Aspect"/>
 </context:component-scan>
 <!-- 启动 @AspectJ 支持 -->
 <aop:aspectj-autoproxy/>
 </beans>
```

显示更多

接下来按传统方式来获取 Spring 容器中 chinese Bean、并调用该 Bean 的两个方法，程序代码如下：

清单 9.BeanTest.java

```
public class BeanTest
 {
 public static void main(String[] args)
 {
 // 创建 Spring 容器
 ApplicationContext ctx = new
 ClassPathXmlApplicationContext("bean.xml");
 Chinese p = ctx.getBean("chinese" ,Chinese.class);
 System.out.println(p.sayHello("张三"));
 p.eat("西瓜");
 }
 }
```

显示更多

从上面开发过程可以看出，对于 Spring AOP 而言，开发者提供的业务组件、方面 Bean 并没有任何特别的地方。只是方面 Bean 需要使用 @Aspect 修饰即可。程序不需要使用特别的编译器、织入器进行处理。

运行上面程序，将可以看到如下执行结果：

执行目标方法之前，模拟开始事务 ...

— 正在执行 sayHello 方法 —

执行目标方法之后，模拟结束事务 ...

获取目标方法返回值 : 被改变的参数 Hello , Spring AOP 新增的内容

模拟记录日志功能 ...

被改变的参数 Hello , Spring AOP 新增的内容

执行目标方法之前，模拟开始事务 ...

我正在吃 : 被改变的参数

执行目标方法之后，模拟结束事务 ...

获取目标方法返回值 :null 新增的内容

模拟记录日志功能 ...

虽然程序是在调用 Chinese 对象的 sayHello、eat 两个方法，但从上面运行结果不难看出：实际执行的绝对不是 Chinese 对象的方法，而是 AOP 代理的方法。也就是说，Spring AOP 同样为 Chinese 类生成了 AOP 代理类。这一点可通过在程序中增加如下代码看出：

System.out.println(p.getClass());

上面代码可以输出 p 变量所引用对象的实现类，再次执行程序将可以看到上面代码产生 class org.crazyit.app.service.impl.Chinese$$EnhancerByCGLIB$$290441d2 的输出，这才是 p 变量所引用的对象的实现类，这个类也就是 Spring AOP 动态生成的 AOP 代理类。从 AOP 代理类的类名可以看出，AOP 代理类是由 CGLIB 来生成的。

如果将上面程序程序稍作修改：只要让上面业务逻辑类 Chinese 类实现一个任意接口——这种做法更符合 Spring 所倡导的”面向接口编程”的原则。假设程序为 Chinese 类提供如下 Person 接口，并让 Chinese 类实现该接口：

清单 10.Person.java

```
public interface Person
 {
 String sayHello(String name);
 void eat(String food);
 }
```

显示更多

接下来让 BeanTest 类面向 Person 接口、而不是 Chinese 类编程。即将 BeanTest 类改为如下形式：

清单 11.BeanTest.java

```
public class BeanTest
 {
 public static void main(String[] args)
 {
 // 创建 Spring 容器
 ApplicationContext ctx = new
 ClassPathXmlApplicationContext("bean.xml");
 Person p = ctx.getBean("chinese" ,Person.class);
 System.out.println(p.sayHello("张三"));
 p.eat("西瓜");
 System.out.println(p.getClass());
 }
 }
```

显示更多

原来的程序是将面向 Chinese 类编程，现在将该程序改为面向 Person 接口编程，再次运行该程序，程序运行结果没有发生改变。只是 System.out.println(p.getClass()); 将会输出 class $Proxy7，这说明此时的 AOP 代理并不是由 CGLIB 生成的，而是由 JDK 动态代理生成的。

Spring AOP 框架对 AOP 代理类的处理原则是：如果目标对象的实现类实现了接口，Spring AOP 将会采用 JDK 动态代理来生成 AOP 代理类；如果目标对象的实现类没有实现接口，Spring AOP 将会采用 CGLIB 来生成 AOP 代理类——不过这个选择过程对开发者完全透明、开发者也无需关心。

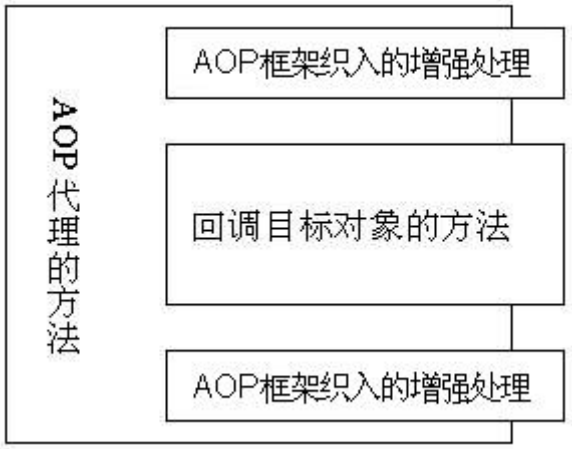
Spring AOP 会动态选择使用 JDK 动态代理、CGLIB 来生成 AOP 代理，如果目标类实现了接口，Spring AOP 则无需 CGLIB 的支持，直接使用 JDK 提供的 Proxy 和 InvocationHandler 来生成 AOP 代理即可。关于如何 Proxy 和 InvocationHandler 来生成动态代理不在本文介绍范围之内，如果读者对 Proxy 和 InvocationHandler 的用法感兴趣则可自行参考 Java API 文档或《疯狂 Java 讲义》。

Spring AOP 原理剖析

通过前面介绍可以知道：AOP 代理其实是由 AOP 框架动态生成的一个对象，该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但 AOP 代理中的方法与目标对象的方法存在差异：AOP 方法在特定切入点添加了增强处理，并回调了目标对象的方法。

AOP 代理所包含的方法与目标对象的方法示意图如图 3 所示。

图 3.AOP 代理的方法与目标对象的方法



Spring 的 AOP 代理由 Spring 的 IoC 容器负责生成、管理，其依赖关系也由 IoC 容器负责管理。因此，AOP 代理可以直接使用容器中的其他 Bean 实例作为目标，这种关系可由 IoC 容器的依赖注入提供。

纵观 AOP 编程，其中需要程序员参与的只有 3 个部分：

- 定义普通业务组件。
- 定义切入点，一个切入点可能横切多个业务组件。
- 定义增强处理，增强处理就是在 AOP 框架为普通业务组件织入的处理动作。

上面 3 个部分的第一个部分是最平常不过的事情，无须额外说明。那么进行 AOP 编程的关键就是定义切入点和定义增强处理。一旦定义了合适的切入点和增强处理，AOP 框架将会自动生成 AOP 代理，而 AOP 代理的方法大致有如下公式：

代理对象的方法 = 增强处理 + 被代理对象的方法

在上面这个业务定义中，不难发现 Spring AOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice 和回调目标对象的方法所组成。

对于前面提到的图 2 所示的软件调用结构：当方法 1、方法 2、方法 3……都需要去调用某个具有”横切”性质的方法时，传统的做法是程序员去手动修改方法 1、方法 2、方法 3……、通过代码来调用这个具有”横切”性质的方法，但这种做法的可扩展性不好，因为每次都要改代码。

于是 AOP 框架出现了，AOP 框架则可以”动态的”生成一个新的代理类，而这个代理类所包含的方法 1、方法 2、方法 3……也增加了调用这个具有”横切”性质的方法——但这种调用由 AOP 框架自动生成的代理类来负责，因此具有了极好的扩展性。程序员无需手动修改方法 1、方法 2、方法 3 的代码，程序员只要定义切入点即可—— AOP 框架所生成的 AOP 代理类中包含了新的方法 1、访法 2、方法 3，而 AOP 框架会根据切入点来决定是否要在方法 1、方法 2、方法 3 中回调具有”横切”性质的方法。

简而言之：AOP 原理的奥妙就在于动态地生成了代理类，这个代理类实现了图 2 的调用——这种调用无需程序员修改代码。接下来介绍的 CGLIB 就是一个代理生成库，下面介绍如何使用 CGLIB 来生成代理类。

使用 CGLIB 生成代理类

CGLIB（Code Generation Library），简单来说，就是一个代码生成类库。它可以在运行时候动态是生成某个类的子类。

此处使用前面定义的 Chinese 类，现在改为直接使用 CGLIB 来生成代理，这个代理类同样可以实现 Spring AOP 代理所达到的效果。

下面先为 CGLIB 提供一个拦截器实现类：

清单 12.AroundAdvice.java

```
public class AroundAdvice implements MethodInterceptor
 {
 public Object intercept(Object target, Method method
 , Object[] args, MethodProxy proxy)
 throws java.lang.Throwable
 {
 System.out.println("执行目标方法之前，模拟开始事务 ...");
 // 执行目标方法，并保存目标方法执行后的返回值
 Object rvt = proxy.invokeSuper(target, new String[]{"被改变的参数"});
 System.out.println("执行目标方法之后，模拟结束事务 ...");
 return rvt + " 新增的内容";
 }
 }
```

显示更多

上面这个 AroundAdvice.java 的作用就像前面介绍的 Around Advice，它可以在调用目标方法之前、调用目标方法之后织入增强处理。

接下来程序提供一个 ChineseProxyFactory 类，这个 ChineseProxyFactory 类会通过 CGLIB 来为 Chinese 生成代理类：

清单 13.ChineseProxyFactory.java

```
public class ChineseProxyFactory
 {
 public static Chinese getAuthInstance()
 {
 Enhancer en = new Enhancer();
 // 设置要代理的目标类
 en.setSuperclass(Chinese.class);
 // 设置要代理的拦截器
 en.setCallback(new AroundAdvice());
 // 生成代理类的实例
 return (Chinese)en.create();
 }
 }
```

显示更多

上面粗体字代码就是使用 CGLIB 的 Enhancer 生成代理对象的关键代码，此时的 Enhancer 将以 Chinese 类作为目标类，以 AroundAdvice 对象作为”Advice”，程序将会生成一个 Chinese 的子类，这个子类就是 CGLIB 生成代理类，它可作为 Chinese 对象使用，但它增强了 Chinese 类的方法。

测试 Chinese 代理类的主程序如下：

清单 14.Main.java

```
public class Main
 {
 public static void main(String[] args)
 {
 Chinese chin = ChineseProxyFactory.getAuthInstance();
 System.out.println(chin.sayHello("孙悟空"));
 chin.eat("西瓜");
 System.out.println(chin.getClass());
 }
 }
```

显示更多

运行上面主程序，看到如下输出结果：

执行目标方法之前，模拟开始事务 ...

— 正在执行 sayHello 方法 —

执行目标方法之后，模拟结束事务 ...

被改变的参数 Hello , CGLIB 新增的内容

执行目标方法之前，模拟开始事务 ...

我正在吃 : 被改变的参数

执行目标方法之后，模拟结束事务 ...

class lee.Chinese$$EnhancerByCGLIB$$4bd097d9

从上面输出结果来看，CGLIB 生成的代理完全可以作为 Chinese 对象来使用，而且 CGLIB 代理对象的 sayHello()、eat() 两个方法已经增加了事务控制（只是模拟），这个 CGLIB 代理其实就是 Spring AOP 所生成的 AOP 代理。

通过程序最后的输出，不难发现这个代理对象的实现类是 lee.Chinese$$EnhancerByCGLIB$$4bd097d9，这就是 CGLIB 所生成的代理类，这个代理类的格式与前面 Spring AOP 所生成的代理类的格式完全相同。

这就是 Spring AOP 的根本所在：Spring AOP 就是通过 CGLIB 来动态地生成代理对象，这个代理对象就是所谓的 AOP 代理，而 AOP 代理的方法则通过在目标对象的切入点动态地织入增强处理，从而完成了对目标方法的增强。

结束语

AOP 广泛应用于处理一些具有横切性质的系统级服务，AOP 的出现是对 OOP 的良好补充，它使得开发者能用更优雅的方式处理具有横切性质的服务。不管是那种 AOP 实现，不论是 AspectJ、还是 Spring AOP，它们都需要动态地生成一个 AOP 代理类，区别只是生成 AOP 代理类的时机不同：AspectJ 采用编译时生成 AOP 代理类，因此具有更好的性能，但需要使用特定的编译器进行处理；而 Spring AOP 则采用运行时生成 AOP 代理类，因此无需使用特定编译器进行处理。由于 Spring AOP 需要在每次运行时生成 AOP 代理，因此性能略差一些。

spring AOP 两种代理方式

回答为什么要用什么方法这种问题的时候，通常首先要回答两个问题，第一个就是，我要做什么事情，第二个就是，不同方法的优劣是什么。

首先，我要做什么事情。

这里的回答比较简单，就是代理Java类/接口。那么，两者在完成这件事情上，有什么差别呢

JDK Proxy	Cglib Proxy
只能代理接口	以继承的方式完成代理，不能代理被final修饰的类

实际上，大部分的Java类都会以接口-实现的方式来完成，因此，在这个方面上，JDK Proxy实际上是比Cglib Proxy要更胜一筹的。因为如果一个类被final修饰，则Cglib Proxy无法进行代理。

其次，两种方法的优劣又在什么地方呢?

我们可以参考一下来自bytebuddy的数据，这个是在代理一个实现了具有18个方法的接口的类，时间单位为ns。

	JDK Proxy	Cglib Proxy
生成代理类时间	1'060.766	960.527
方法调用时间	0.008	0.003
来源	JDK原生代码	第三方库，更新频率较低

不难看出，其实Cglib代理的性能是要远远好于JDK代理的。

其实从原理也能理解，直接通过类的方法调用，肯定要比通过反射调用的时间更短。但是从来源来看的话，一个是JDK原生代码，而另一个则是第三方的开源库。JDK原生代码无疑使用的人会更多范围也更广，会更佳稳定，而且还有可能在未来的JDK版本中不断优化性能。

而Cglib更新频率相对来说比较低了，一方面是因为这个代码库已经渐趋稳定，另一方面也表明后续这个库可能相对来说不会有大动作的优化维护。

对比完之后，再来回看这个问题，为什么要使用两种方式呢?

在功能上讲，实际上Cglib代理并不如JDK代理（如果大家都按接口-实现的方式来设计类）。但是从效率上将，Cglib远胜JDK代理啊！所以，为了提高效率，同时又保有在未来，当JDK代理的性能也能够同样好的时候，使用更佳稳定靠谱的JDK代码，这种可能，于是采取了这种设计。

Spring 如何保证 Controller 并发的安全?

springMVC一个Controller处理所有用户请求的并发问题

有状态和无状态的对象基本概念：
有状态对象(Stateful Bean)，就是有实例变量的对象 ，可以保存数据，是非线程安全的。一般是prototype scope。
无状态对象(Stateless Bean)，就是没有实例变量的对象，不能保存数据，是不变类，是线程安全的。一般是singleton scope。

如Struts2中的Action，假如内部有实例变量User，当调用新增用户方法时，user是用来保存数据，那么此action是有状态对象。多个线程同时访问此action时 会造成user变量的不一致。所以action的scope要设计成prototype，或者，User类放到threadLocal里来保持多个线程不会造成User变量的乱串（此种场景没必要放到threadLocal内）。

而Service内部一般只有dao实例变量 如userDao, 因为userDao是无状态的对象（内部无实例变量且不能保存数据），所以service也是无状态的对象。

public class XxxAction{

// 由于多线程环境下，user是引用对象，是非线程安全的

public User user;

.....

}

public class XxxService {

// 虽然有billDao属性，但billDao是没有状态信息的，是Stateless Bean.

BillDao billDao;

.....

}

对于那些会以多线程运行的单例类

局部变量不会受多线程影响，
成员变量会受到多线程影响。

多个线程调用同一个对象的同一个方法：
如果方法里无局部变量，那么不受任何影响；
如果方法里有局部变量，只有读操作，不受影响；存在写操作，考虑多线程影响值；

例如Web应用中的Servlet，每个方法中对局部变量的操作都是在线程自己独立的内存区域内完成的，所以是线程安全的。
对于成员变量的操作，可以使用ThreadLocal来保证线程安全。

springMVC中，一般Controller、service、DAO层的scope均是singleton；

每个请求都是单独的线程,即使同时访问同一个Controller对象，因为并没有修改Controller对象，相当于针对Controller对象而言，只是读操作，没有写操作，不需要做同步处理。

Service层、Dao层用默认singleton就行，虽然Service类也有dao这样的属性，但dao这些类都是没有状态信息的，也就是 相当于不变(immutable)类，所以不影响。

Struts2中的Action因为会有User、BizEntity这样的实例对象，是有状态信息 的，在多线程环境下是不安全的，所以Struts2默认的实现是Prototype模式。在Spring中，Struts2的Action中scope 要配成prototype作用域。

Spring并发访问的线程安全性问题

由于Spring MVC默认是Singleton的，所以会产生一个潜在的安全隐患。根本核心是instance变量保持状态的问题。这意味着每个request过来，系统都会用原有的instance去处理，这样导致了两个结果：
一是我们不用每次创建Controller，
二是减少了对象创建和垃圾收集的时间；
由于只有一个Controller的instance，当多个线程同时调用它的时候，它里面的instance变量就不是线程安全的了，会发生窜数据的问题。
当然大多数情况下，我们根本不需要考虑线程安全的问题，比如dao,service等，除非在bean中声明了实例变量。因此，我们在使用spring mvc 的contrller时，应避免在controller中定义实例变量。
如：

```
publicclassControllerextendsAbstractCommandController{
......
protectedModelAndView handle(HttpServletRequest request,HttpServletResponse
response,
Object command,BindException errors)throwsException{
company =.............;
}
protectedCompany company;
}
```

在这里有声明一个变量company，这里就存在并发线程安全的问题。
如果控制器是使用单例形式，且controller中有一个私有的变量a,所有请求到同一个controller时，使用的a变量是共用的，即若是某个请求中修改了这个变量a，则，在别的请求中能够读到这个修改的内容。。

有几种解决方法：
1、在控制器中不使用实例变量
2、将控制器的作用域从单例改为原型，即在spring配置文件Controller中声明 scope="prototype"，每次都创建新的controller
3、在Controller中使用ThreadLocal变量

这几种做法有好有坏，第一种，需要开发人员拥有较高的编程水平与思想意识，在编码过程中力求避免出现这种BUG，而第二种则是容器自动的对每个请求产生一个实例，由JVM进行垃圾回收，因此做到了线程安全。
使用第一种方式的好处是实例对象只有一个，所有的请求都调用该实例对象，速度和性能上要优于第二种，不好的地方，就是需要程序员自己去控制实例变量的状态保持问题。第二种由于每次请求都创建一个实例，所以会消耗较多的内存空间。
所以在使用spring开发web 时要注意，默认Controller、Dao、Service都是单例的

spring中用到哪些设计模式?

1.工厂模式，这个很明显，在各种BeanFactory以及ApplicationContext创建中都用到了；

2.模版模式，这个也很明显，在各种BeanFactory以及ApplicationContext实现中也都用到了；

3.代理模式，在Aop实现中用到了JDK的动态代理；

4.单例模式，这个比如在创建bean的时候。

5.Tomcat中有很多场景都使用到了外观模式，因为Tomcat中有很多不同的组件，每个组件需要相互通信，但又不能将自己内部数据过多地暴露给其他组件。用外观模式隔离数据是个很好的方法。

6.策略模式在Java中的应用，这个太明显了，因为Comparator这个接口简直就是为策略模式而生的。Comparable和Comparator的区别一文中，详细讲了Comparator的使用。比方说Collections里面有一个sort方法，因为集合里面的元素有可能是复合对象，复合对象并不像基本数据类型，可以根据大小排序，复合对象怎么排序呢？基于这个问题考虑，Java要求如果定义的复合对象要有排序的功能，就自行实现Comparable接口或Comparator接口.

7.原型模式：使用原型模式创建对象比直接new一个对象在性能上好得多，因为Object类的clone()方法是一个native方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。

8.迭代器模式：Iterable接口和Iterator接口 这两个都是迭代相关的接口，可以这么认为，实现了Iterable接口，则表示某个对象是可被迭代的；Iterator接口相当于是一个迭代器，实现了Iterator接口，等于具体定义了这个可被迭代的对象时如何进行迭代的

Spring IOC 的理解，其初始化过程?

IoC容器是什么?
IoC文英全称Inversion of Control，即控制反转，我么可以这么理解IoC容器：
“把某些业务对象的的控制权交给一个平台或者框架来同一管理，这个同一管理的平台可以称为IoC容器。”

我们刚开始学习spring的时候会经常看到的类似下面的这代码：

```
ApplicationContext appContext = new
ClassPathXmlApplicationContext("cjj/models/beans.xml");
Person p = (Person)appContext.getBean("person");
```

上面代码中，在创建ApplicationContext实例对象过程中会创建一个spring容器，该容器会读取配置文件"cjj/models/beans.xml",并统一管理由该文件中定义好的所有bean实例对象，如果要获取某个bean实例，使用getBean方法就行了。例如我们只需要将Person提前配置在beans.xml文件中（可以理解为注入），之后我们可以不需使用new Person()的方式创建实例，而是通过容器来获取Person实例，这就相当于将Person的控制权交由spring容器了，差不多这就是控制反转的概念。

那在创建IoC容器时经历了哪些呢？为此，先来了解下Spring中IoC容器分类，继而根据一个具体的容器来讲解IoC容器初始化的过程。
Spring中有两个主要的容器系列：

1. 实现BeanFactory接口的简单容器；
2. 实现ApplicationContext接口的高级容器。

ApplicationContext比较复杂，它不但继承了BeanFactory的大部分属性，还继承其它可扩展接口，扩展的了许多高级的属性，其接口定义如下：

```
public interface ApplicationContext extends EnvironmentCapable,
                                ListableBeanFactory,    //继承于
BeanFactory                                HierarchicalBeanFactory,//继承于
BeanFactory
                                MessageSource,
//
ApplicationEventPublisher,//                                ResourcePatternResolver
  //继承ResourceLoader，用于获取resource对象
```

在BeanFactory子类中有一个***DefaultListableBeanFactory***类，它包含了基本Spirng IoC容器所具有的重要功能，开发时不论是使用BeanFactory系列还是ApplicationContext系列来创建容器基本都会使用到DefaultListableBeanFactory类，可以这么说，在spring中实际上把它当成默认的IoC容器来使用。下文在源码实例分析时你将会看到这个类。

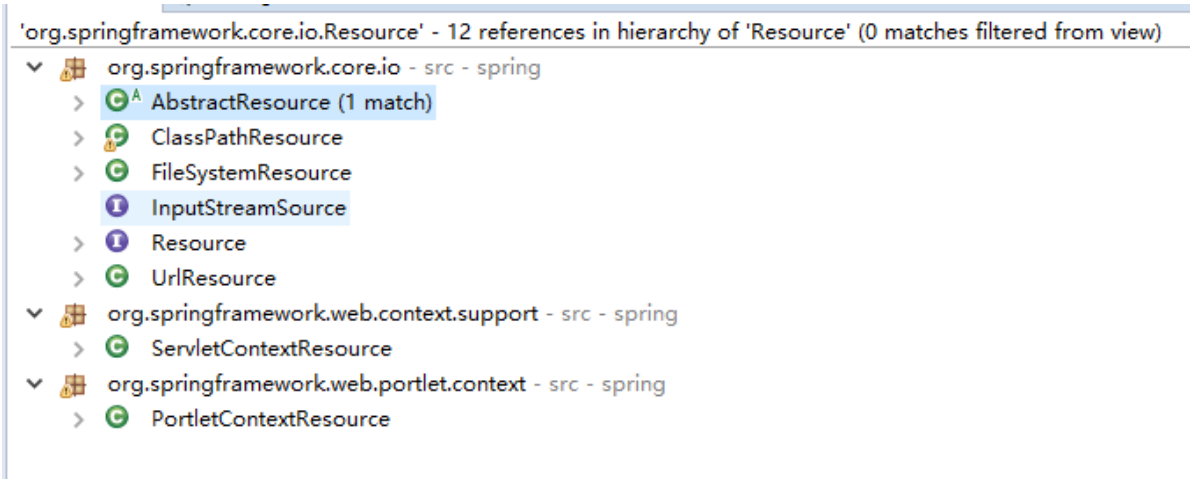
（注：文章有点长，需要细心阅读，不同版本的spring中源码可能不同，但逻辑几乎是一样的，如果可以建议还是看源码 ^_^）

回到本文正题上来，关于Spirng IoC容器的初始化过程在《Spirng技术内幕：深入解析Spring架构与设计原理》一书中有明确的指出，IoC容器的初始化过程可以分为三步：

1. Resource定位（Bean的定义文件定位）
2. 将Resource定位好的资源载入到BeanDefinition
3. 将BeanDefiniton注册到容器中

- **第一步 Resource定位**

Resource是Sping中用于封装I/O操作的接口。正如前面所见，在创建spring容器时，通常要访问XML配置文件，除此之外还可以通过访问文件类型、二进制流等方式访问资源，还有当需要网络上的资源时可以通过访问URL，Spring把这些文件统称为Resource，Resource的体系结构如下：



常用的resource资源类型如下：

FileSystemResource：以文件的绝对路径方式进行访问资源，效果类似于Java中的File;

ClassPathResourcee：以类路径的方式访问资源，效果类似于this.getClass().getResource("/").getPath();

ServletContextResource：web应用根目录的方式访问资源，效果类似于request.getServletContext().getRealPath("");

UrlResource：访问网络资源的实现类。例如file: http: ftp:等前缀的资源对象;

ByteArrayResource: 访问字节数组资源的实现类。

那如何获取上图中对应的各种Resource对象呢?
Spring提供了ResourceLoader接口用于实现不同的Resource加载策略，该接口的实例对象中可以获取一个resource对象，也就是说将不同Resource实例的创建交给ResourceLoader的实现类来处理。
ResourceLoader接口中只定义了两个方法：

```
Resource getResource(String location); //通过提供的资源location参数获取Resource实例
ClassLoader getClassLoader(); // 获取ClassLoader,通过ClassLoader可将资源载入JVM
```

注：ApplicationContext的所有实现类都实现RecourceLoader接口，因此可以直接调用getResource（参数）获取Resoure对象。不同的ApplicatonContext实现类使用getResource方法取得的资源类型不同，例如：FileSystemXmlApplicationContext.getResource获取的就是FileSystemResource实例；ClassPathXmlApplicationContext.gerResource获取的就是ClassPathResource实例；XmlWebApplicationContext.getResource获取的就是ServletContextResource实例，另外像不需要通过xml直接使用注解@Configuation方式加载资源的AnnotationConfigApplicationContext等等。

在资源定位过程完成以后，就为资源文件中的bean的载入创造了I/O操作的条件，如何读取资源中的数据将会在下一步介绍的BeanDefinition的载入过程中描述。

- ***第二步 通过返回的*resource对象，进行BeanDefinition的载入**

1.什么是BeanDefinition? BeanDefinition与Resource的联系呢?

官方文档中对BeanDefinition的解释如下：
A BeanDefinition describes a bean instance, which has property values, constructor argument values, and further information supplied by concrete implementations.
它们之间的联系从官方文档描述的一句话：Load bean definitions from the specified resource中可见一斑。

```
   /**
     * Load bean definitions from the specified resource.
     * @param resource the resource descriptor
     * @return the number of bean definitions found
     * @throws BeanDefinitionStoreException in case of loading or parsing errors
     */
    int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException;
```

总之，BeanDefinition相当于一个数据结构，这个数据结构的生成过程是根据定位的resource资源对象中的bean而来的，这些bean在Spirng IoC容器内部表示成了的BeanDefintion这样的数据结构，IoC容器对bean的管理和依赖注入的实现都是通过操作BeanDefinition来进行的。

2.如何将BeanDefinition载入到容器?
在Spring中配置文件主要格式是XML，对于用来读取XML型资源文件来进行初始化的IoC 容器而言，该类容器会使用到AbstractXmlApplicationContext类，该类定义了一个名为loadBeanDefinitions(DefaultListableBeanFactory beanFactory) 的方法用于获取BeanDefinition：

```
// 该方法属于AbstractXmlApplicationContect类
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws
BeansException, IOException {
    XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);
    beanDefinitionReader.setEnvironment(this.getEnvironment());
    beanDefinitionReader.setResourceLoader(this);
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
    this.initBeanDefinitionReader(beanDefinitionReader);
    // 用于获取BeanDefinition
    this.loadBeanDefinitions(beanDefinitionReader);
}
```

此方法在具体执行过程中首先会new一个与容器对应的BeanDefinitionReader型实例对象，然后将生成的BeanDefintionReader实例作为参数传入loadBeanDefintions(XmlBeanDefinitionReader)，继续往下执行载入BeanDefintion的过程。例如AbstractXmlApplicationContext有两个实现类：FileSystemXmlApplicationContext、ClassPathXmlApplicationContext，这些容器在调用此方法时会创建一个**XmlBeanDefinitionReader类**对象专门用来载入所有的BeanDefinition。

下面以XmlBeanDefinitionReader对象载入BeanDefinition为例，使用源码说明载入BeanDefinition的过程：

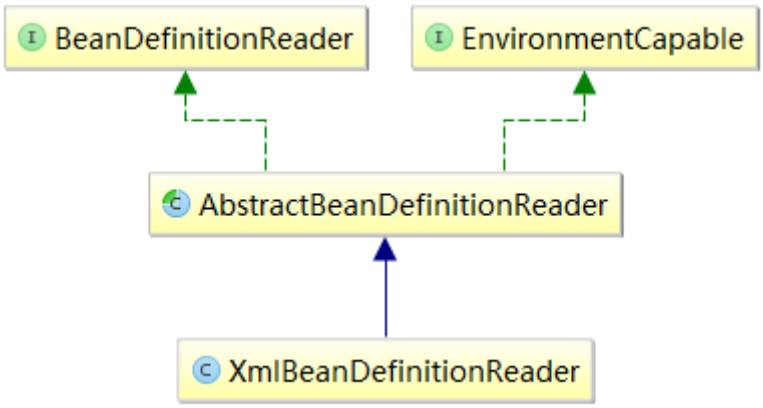
```
// 该方法属于AbstractXmlApplicationContect类protected void
loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException,
IOException {
        Resource[] configResources = getConfigResources();//获取所有定位到的
resource资源位置（用户定义）
        if (configResources != null) {
            reader.loadBeanDefinitions(configResources);//载入resources
        }
        String[] configLocations = getConfigLocations();//获取所有本地配置文件的位置
（容器自身）
        if (configLocations != null) {
            reader.loadBeanDefinitions(configLocations);//载入resources
        }
}
```

通过上面代码将用户定义的资源以及容器本身需要的资源全部加载到reader中，
reader.loadBeanDefinitions方法的源码如下：

```
// 该方法属于AbstractBeanDefinitionReader类，父接口BeanDefinitionReader
@Override
public int loadBeanDefinitions(Resource... resources) throws
BeanDefinitionStoreException {
    Assert.notNull(resources, "Resource array must not be null");
    int counter = 0;
    for (Resource resource : resources) {
        // 将所有资源全部加载，交给AbstractBeanDefinitionReader的实现子类处理这些
resource
        counter += loadBeanDefinitions(resource);
    }
    return counter;
}
```

BeanDefinitionReader接口定义了 int loadBeanDefinitions（Resource resource）方法：

```
int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException;
int loadBeanDefinitions(Resource... resources) throws
BeanDefinitionStoreException;
```



XmlBeanDefinitionReader 类实现了BeanDefinitionReader接口中的loadBeanDefinitions(Resource)方法，其继承关系如上图所示。XmlBeanDefinitionReader类中几主要对加载的所有resource开始进行处理，大致过程是，先将resource包装为EncodeResource类型，然后处理，为生成BeanDefinition对象做准备，其主要几个方法的源码如下：

```
    public int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException {
        // 包装resource为EncodeResource类型
        return loadBeanDefinitions(new EncodedResource(resource));
    }

    // 加载包装后的EncodeResource资源
    public int loadBeanDefinitions(EncodedResource encodedResource) throws
BeanDefinitionStoreException {
        Assert.notNull(encodedResource, "EncodedResource must not be null");
        if (logger.isInfoEnabled()) {
            logger.info("Loading XML bean definitions from " +
encodedResource.getResource());
        }
```

```
        try {
            // 通过resource对象得到XML文件内容输入流，并为IO的InputSource做准备
            InputStream inputStream =
encodedResource.getResource().getInputStream();
            try {
                // Create a new input source with a byte stream.
                InputSource inputSource = new InputSource(inputStream);
                if (encodedResource.getEncoding() != null) {
                    inputSource.setEncoding(encodedResource.getEncoding());
                }
                // 开始准备 load bean definitions from the specified XML file
                return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
            }
            finally {
                inputStream.close();
            }
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(
                    "IOException parsing XML document from " +
encodedResource.getResource(), ex);
        }
    }

    protected int doLoadBeanDefinitions(InputSource inputSource, Resource
resource)
            throws BeanDefinitionStoreException {
        try {
            // 获取指定资源的验证模式
            int validationMode = getValidationModeForResource(resource);

            // 从资源对象中加载DocumentL对象，大致过程为：将resource资源文件的内容读入到
document中
            // DocumentLoader在容器读取XML文件过程中有着举足轻重的作用!
            // XmlBeanDefinitionReader实例化时会创建一个DefaultDocumentLoader型的私有
属性,继而调用loadDocument方法
            // inputSource--要加载的文档的输入源
            Document doc = this.documentLoader.loadDocument(
                    inputSource, this.entityResolver, this.errorHandler,
validationMode, this.namespaceAware);

            // 将document文件的bean封装成BeanDefinition，并注册到容器
            return registerBeanDefinitions(doc, resource);
        }
        catch ...(略)
    }
```

DefaultDocumentLoader大致了解即可，感兴趣可继续深究，其源码如下：（看完收起，便于阅读下文）

⊞ View Code

上面代码分析到了registerBeanDefinitions(doc, resource)这一步，也就是准备将Document中的Bean按照Spring bean语义进行解析并转化为BeanDefinition类型，这个方法的具体过程如下：

```
/**
 * 属于XmlBeanDefinitionReader类
 * Register the bean definitions contained in the given DOM document.
 * @param doc the DOM document
 * @param resource
 * @return the number of bean definitions found
 * @throws BeanDefinitionStoreException
 */
public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
    // 获取到DefaultBeanDefinitionDocumentReader实例
    BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
    // 获取容器中bean的数量
    int countBefore = getRegistry().getBeanDefinitionCount();
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

通过 XmlBeanDefinitionReader 类中的私有属性 documentReaderClass 可以获得一个 DefaultBeanDefinitionDocumentReader 实例对象：

```
private Class<?> documentReaderClass =
DefaultBeanDefinitionDocumentReader.class;
protected BeanDefinitionDocumentReader createBeanDefinitionDocumentReader() {
   return
BeanDefinitionDocumentReader.class.cast(BeanUtils.instantiateClass(this.document
ReaderClass));
}
```

DefaultBeanDefinitionDocumentReader实现了BeanDefinitionDocumentReader接口，它的 registerBeanDefinitions方法定义如下：

```
// DefaultBeanDefinitionDocumentReader implements BeanDefinitionDocumentReader
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext)
{
    this.readerContext = readerContext;

    logger.debug("Loading bean definitions");
    // 获取doc的root节点，通过该节点能够访问所有的子节点
    Element root = doc.getDocumentElement();
    // 处理beanDefinition的过程委托给BeanDefinitionParserDelegate实例对象来完成
    BeanDefinitionParserDelegate delegate = createHelper(readerContext, root);

    // Default implementation is empty.
    // Subclasses can override this method to convert custom elements into
standard Spring bean definitions
    preProcessXml(root);
    // 核心方法，代理
    parseBeanDefinitions(root, delegate);
```

```
    postProcessXml(root);
}
```

上面出现的**BeanDefinitionParserDelegate类**非常非常重要（需要了解代理技术，如JDK动态代理、cglib动态代理等）。Spirng BeanDefinition的解析就是在这个代理类下完成的，此类包含了各种对符合Spring Bean语义规则的处理，比如、、等的检测。

parseBeanDefinitions(root, delegate)方法如下：

```
   protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate
delegate) {
        if (delegate.isDefaultNamespace(root)) {
            NodeList nl = root.getChildNodes();
            // 遍历所有节点，做对应解析工作
            // 如遍历到<import>标签节点就调用importBeanDefinitionResource(ele)方法对应
处理
            // 遍历到<bean>标签就调用processBeanDefinition(ele,delegate)方法对应处理
            for (int i = 0; i < nl.getLength(); i++) {
                Node node = nl.item(i);
                if (node instanceof Element) {
                    Element ele = (Element) node;
                    if (delegate.isDefaultNamespace(ele)) {
                        parseDefaultElement(ele, delegate);
                    }
                    else {
                        //对应用户自定义节点处理方法
                        delegate.parseCustomElement(ele);
                    }
                }
            }
        }
        else {
            delegate.parseCustomElement(root);
        }
    }

    private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate
delegate) {
        // 解析<import>标签
        if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
            importBeanDefinitionResource(ele);
        }
        // 解析<alias>标签
        else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
            processAliasRegistration(ele);
        }
        // 解析<bean>标签,最常用，过程最复杂
        else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
            processBeanDefinition(ele, delegate);
        }
        // 解析<beans>标签
        else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
            // recurse
            doRegisterBeanDefinitions(ele);
```

```
        }
    }
```

这里针对常用的标签中的方法做简单介绍，其他标签的加载方式类似：

```
    /**
     * Process the given bean element, parsing the bean definition
     * and registering it with the registry.
     */
    protected void processBeanDefinition(Element ele,
BeanDefinitionParserDelegate delegate) {
        // 该对象持有beanDefinition的name和alias，可以使用该对象完成beanDefinition向容
器的注册
        BeanDefinitionHolder bdHolder =
delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
            try {
                // 注册最终被修饰的bean实例，下文注册beanDefinition到容器会讲解该方法
                BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error("Failed to register bean definition with
name '" +
                        bdHolder.getBeanName() + "'", ele, ex);
            }
            // Send registration event.
            getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
        }
    }
```

parseBeanDefinitionElement(Element ele)方法会调用parseBeanDefinitionElement(ele, null)方法，并将值返回**BeanDefinitionHolder类**对象，这个方法将会对给定的标签进行解析，如果在解析标签的过程中出现错误则返回null。

需要强调一下的是parseBeanDefinitionElement(ele, null)方法中产生了一个抽象类型的BeanDefinition实例，这也是我们首次看到直接定义BeanDefinition的地方，这个方法里面会将标签中的内容解析到BeanDefinition中，之后再对BeanDefinition进行包装，将它与beanName,Alias等封装到BeanDefinitionHolder 对象中，该部分源码如下：

```
    public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
        return parseBeanDefinitionElement(ele, null);
    }

    public BeanDefinitionHolder parseBeanDefinitionElement(Element ele,
BeanDefinition containingBean) {
        String id = ele.getAttribute(ID_ATTRIBUTE);
```

```
        String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
            ...(略)
        String beanName = id;
           ...（略）
        // 从上面按过程走来，首次看到直接定义BeanDefinition ！！！
        // 该方法会对<bean>节点以及其所有子节点如<property>、<List>、<Set>等做出解析，具体过程本文不做分析（太多太长）
        AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);
        if (beanDefinition != null) {
            if (!StringUtils.hasText(beanName)) {
                ...(略)
            }
            String[] aliasesArray = StringUtils.toStringArray(aliases);
            return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
        }

        return null;
    }
```

- **第三步，将BeanDefiniton注册到容器中**

最终Bean配置会被解析成BeanDefinition并与beanName,Alias一同封装到**BeanDefinitionHolder类**中， 之后beanFactory.registerBeanDefinition(beanName, bdHolder.getBeanDefinition())，注册到**DefaultListableBeanFactory**.beanDefinitionMap中。之后客户端如果要获取Bean对象，Spring容器会根据注册的BeanDefinition信息进行实例化。

BeanDefinitionReaderUtils类：

```
public static void registerBeanDefinition(
            BeanDefinitionHolder bdHolder, BeanDefinitionRegistry beanFactory)
throws BeansException {

        // Register bean definition under primary name.
        String beanName = bdHolder.getBeanName();         // 注册beanDefinition!!!
        beanFactory.registerBeanDefinition(beanName,
bdHolder.getBeanDefinition());

        // 如果有别名的话也注册进去，Register aliases for bean name, if any.
        String[] aliases = bdHolder.getAliases();
        if (aliases != null) {
            for (int i = 0; i < aliases.length; i++) {
                beanFactory.registerAlias(beanName, aliases[i]);
            }
        }
    }
```

DefaultListableBeanFactory实现了上面调用BeanDefinitionRegistry接口的registerBeanDefinition(beanName, bdHolder.getBeanDefinition())方法，这一部分的主要逻辑是向DefaultListableBeanFactory对象的**beanDefinitionMap**中存放beanDefinition，当初始化容器进行bean初始化时，在bean的生命周期分析里必然会在这个beanDefinitionMap中获取beanDefition实例，有机会成文分析一下bean的生命周期，到时可以分析一下如何使用这个beanDefinitionMap。

registerBeanDefinition(beanName, bdHolder.getBeanDefinition())方法具体方法如下：

```
    /** Map of bean definition objects, keyed by bean name */
    private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<String, BeanDefinition>(256);
```

```
public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
            throws BeanDefinitionStoreException {

        Assert.hasText(beanName, "Bean name must not be empty");
        Assert.notNull(beanDefinition, "Bean definition must not be null");

        if (beanDefinition instanceof AbstractBeanDefinition) {
            try {
                ((AbstractBeanDefinition) beanDefinition).validate();
            }
            catch (BeanDefinitionValidationException ex) {
                throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                        "Validation of bean definition failed", ex);
            }
        }
              // beanDefinitionMap是个ConcurrentHashMap类型数据，用于存放
beanDefinition,它的key值是beanName
        Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
        if (oldBeanDefinition != null) {
            if (!this.allowBeanDefinitionOverriding) {
                throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                        "Cannot register bean definition [" + beanDefinition + "]
for bean '" + beanName +
                        "': there's already [" + oldBeanDefinition + "] bound");
            }
            else {
                if (logger.isInfoEnabled()) {
                    logger.info("Overriding bean definition for bean '" +
beanName +
                            "': replacing [" + oldBeanDefinition + "] with [" +
beanDefinition + "]");
                }
            }
        }
        else {
            this.beanDefinitionNames.add(beanName);
        }        // 将获取到的BeanDefinition放入Map中，容器操作使用bean时通过这个
HashMap找到具体的BeanDefinition
```

```
        this.beanDefinitionMap.put(beanName, beanDefinition);        // Remove
corresponding bean from singleton cache, if any.
        // Shouldn't usually be necessary, rather just meant for overriding
        // a context's default beans (e.g. the default StaticMessageSource
        // in a StaticApplicationContext).
        removeSingleton(beanName);
    }
```

Spring的事务管理

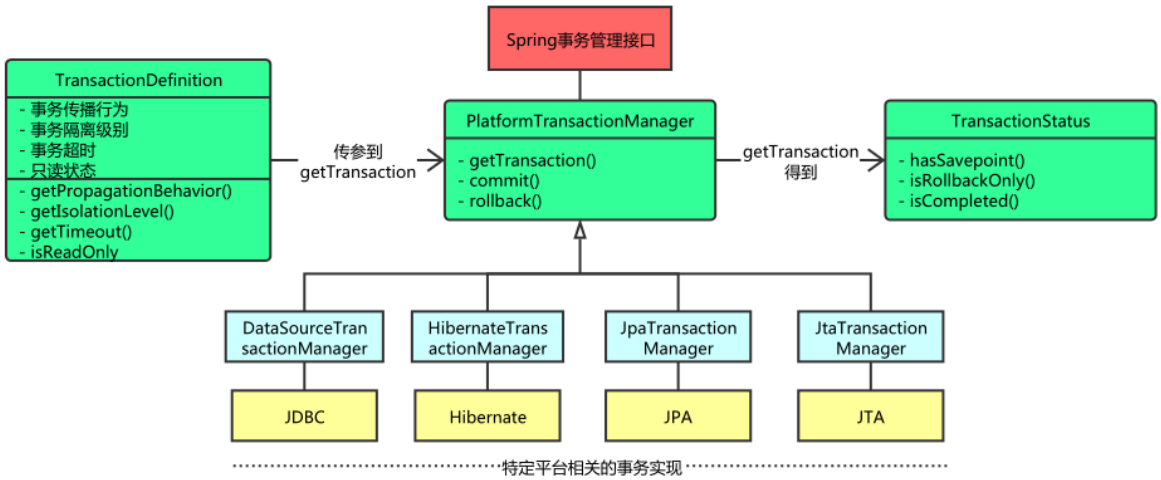
1 初步理解

理解事务之前，先讲一个你日常生活中最常干的事：取钱。
比如你去ATM机取1000块钱，大体有两个步骤：首先输入密码金额，银行卡扣掉1000元钱；然后ATM出1000元钱。这两个步骤必须是要么都执行要么都不执行。如果银行卡扣除了1000块但是ATM出钱失败的话，你将会损失1000元；如果银行卡扣钱失败但是ATM却出了1000块，那么银行将损失1000元。
所以，如果一个步骤成功另一个步骤失败对双方都不是好事，如果不管哪一个步骤失败了以后，整个取钱过程都能回滚，也就是完全取消所有操作的话，这对双方都是极好的。
事务就是用来解决类似问题的。事务是一系列的动作，它们综合在一起才是一个完整的工作单元，这些动作必须全部完成，如果有一个失败的话，那么事务就会回滚到最开始的状态，仿佛什么都没发生过一样。
在企业级应用程序开发中，事务管理必不可少的技术，用来确保数据的完整性和一致性。
事务有四个特性：ACID

> - 原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。
> - 一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。
> - 隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
> - 持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

2 核心接口

Spring事务管理的实现有许多细节，如果对整个接口框架有个大体了解会非常有利于我们理解事务，下面通过讲解Spring的事务接口来了解Spring实现事务的具体策略。
Spring事务管理涉及的接口的联系如下：



2.1 事务管理器

Spring并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。
Spring事务管理器的接口是org.springframework.transaction.PlatformTransactionManager，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。此接口的内容如下：

```
Public interface PlatformTransactionManager()...{
    // 由TransactionDefinition得到TransactionStatus对象
    TransactionStatus getTransaction(TransactionDefinition definition) throws
TransactionException;
    // 提交
    Void commit(TransactionStatus status) throws TransactionException;
    // 回滚
    Void rollback(TransactionStatus status) throws TransactionException;
    }
```

从这里可知具体的具体的事务管理机制对Spring来说是透明的，它并不关心那些，那些是对应各个平台需要关心的，所以Spring事务管理的一个优点就是为不同的事务API提供一致的编程模型，如JTA、JDBC、Hibernate、JPA。下面分别介绍各个平台框架实现事务管理的机制。

2.1.1 JDBC事务

如果应用程序中直接使用JDBC来进行持久化，DataSourceTransactionManager会为你处理事务边界。为了使用DataSourceTransactionManager，你需要使用如下的XML将其装配到应用程序的上下文定义中：

```
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
```

实际上，DataSourceTransactionManager是通过调用java.sql.Connection来管理事务，而后者是通过DataSource获取到的。通过调用连接的commit()方法来提交事务，同样，事务失败则通过调用rollback()方法进行回滚。

2.1.2 Hibernate事务

如果应用程序的持久化是通过Hibernate实习的，那么你需要使用HibernateTransactionManager。对于Hibernate3，需要在Spring上下文定义中添加如下的`<bean>`声明：

```
    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
```

sessionFactory属性需要装配一个Hibernate的session工厂，HibernateTransactionManager的实现细节是它将事务管理的职责委托给org.hibernate.Transaction对象，而后者是从Hibernate Session中获取到的。当事务成功完成时，HibernateTransactionManager将会调用Transaction对象的commit()方法，反之，将会调用rollback()方法。

2.1.3 Java持久化API事务（JPA）

Hibernate多年来一直是事实上的Java持久化标准，但是现在Java持久化API作为真正的Java持久化标准进入大家的视野。如果你计划使用JPA的话，那你需要使用Spring的JpaTransactionManager来处理事务。你需要在Spring中这样配置JpaTransactionManager：

```
    <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
```

JpaTransactionManager只需要装配一个JPA实体管理工厂（javax.persistence.EntityManagerFactory接口的任意实现）。JpaTransactionManager将与由工厂所产生的JPA EntityManager合作来构建事务。

2.1.4 Java原生API事务

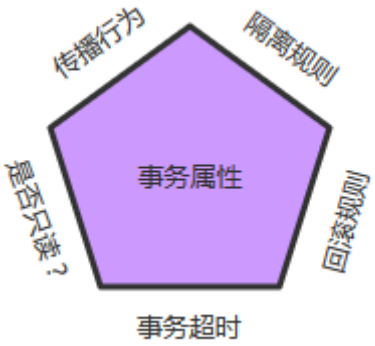
如果你没有使用以上所述的事务管理，或者是跨越了多个事务管理源（比如两个或者是多个不同的数据源），你就需要使用JtaTransactionManager：

```
    <bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
        <property name="transactionManagerName" value="java:/TransactionManager"
/>
    </bean>
```

JtaTransactionManager将事务管理的责任委托给javax.transaction.UserTransaction和javax.transaction.TransactionManager对象，其中事务成功完成通过UserTransaction.commit()方法提交，事务失败通过UserTransaction.rollback()方法回滚。

2.2 基本事务属性的定义

上面讲到的事务管理器接口PlatformTransactionManager通过getTransaction(TransactionDefinition definition)方法来得到事务，这个方法里面的参数是TransactionDefinition类，这个类就定义了一些基本的事务属性。
那么什么是事务属性呢？事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了5个方面，如图所示：



而TransactionDefinition接口内容如下：

```
public interface TransactionDefinition {
    int getPropagationBehavior(); // 返回事务的传播行为
    int getIsolationLevel(); // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据
    int getTimeout();  // 返回事务必须在多少秒内完成
    boolean isReadOnly(); // 事务是否只读，事务管理器能够根据这个返回值进行优化，确保事务是只读的
}
```

我们可以发现TransactionDefinition正好用来定义事务属性，下面详细介绍一下各个事务属性。

2.2.1 传播行为

事务的第一个方面是传播行为（propagation behavior）。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。Spring定义了七种传播行为：

传播行为	含义
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION_REQUIRED一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

*注：以下具体讲解传播行为的内容参考自Spring事务机制详解
（1）PROPAGATION_REQUIRED 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

```
//事务属性 PROPAGATION_REQUIRED
methodA{
    ……
    methodB();
    ……
}
//事务属性 PROPAGATION_REQUIRED
methodB{
   ……
}
```

使用spring声明式事务，spring使用AOP来支持声明式事务，会根据事务属性，自动在方法调用之前决定是否开启一个事务，并在方法执行之后决定事务提交或回滚事务。

单独调用methodB方法：

```
main{
    metodB();
}
```

相当于

```
Main{
    Connection con=null;
    try{
        con = getConnection();
        con.setAutoCommit(false);

        //方法调用
        methodB();

        //提交事务
        con.commit();
    } Catch(RuntimeException ex) {
        //回滚事务
        con.rollback();
    } finally {
        //释放资源
        closeCon();
    }
}
```

Spring保证在methodB方法中所有的调用都获得到一个相同的连接。在调用methodB时，没有一个存在的事务，所以获得一个新的连接，开启了一个新的事务。
单独调用MethodA时，在MethodA内又会调用MethodB.

执行效果相当于:

```
main{
    Connection con = null;
    try{
        con = getConnection();
        methodA();
        con.commit();
    } catch(RuntimeException ex) {
        con.rollback();
    } finally {
        closeCon();
    }
}
```

调用MethodA时，环境中没有事务，所以开启一个新的事务.当在MethodA中调用MethodB时，环境中已经有了一个事务，所以methodB就加入当前事务。

（2）PROPAGATION_SUPPORTS 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。但是对于事务同步的事务管理器，PROPAGATION_SUPPORTS与不使用事务有少许不同。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
  methodB();
}

//事务属性 PROPAGATION_SUPPORTS
methodB(){
  ……
}
```

单纯的调用methodB时，methodB方法是非事务的执行的。当调用methdA时,methodB则加入了methodA的事务中,事务地执行。

（3）PROPAGATION_MANDATORY 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    methodB();
}

//事务属性 PROPAGATION_MANDATORY
    methodB(){
    ……
}
```

当单独调用methodB时，因为当前没有一个活动的事务，则会抛出异常throw new IllegalTransactionStateException(“Transaction propagation ‘mandatory’ but no existing transaction found”);当调用methodA时，methodB则加入到methodA的事务中，事务地执行。

（4）PROPAGATION_REQUIRES_NEW 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    doSomeThingA();
    methodB();
    doSomeThingB();
}

//事务属性 PROPAGATION_REQUIRES_NEW
methodB(){
    ……
}
```

调用A方法：

```
main(){
    methodA();
}
```

相当于

```
main(){
    TransactionManager tm = null;
    try{
        //获得一个JTA事务管理器
        tm = getTransactionManager();
        tm.begin();//开启一个新的事务
        Transaction ts1 = tm.getTransaction();
        doSomeThing();
        tm.suspend();//挂起当前事务
        try{
            tm.begin();//重新开启第二个事务
            Transaction ts2 = tm.getTransaction();
```

```
            methodB();
            ts2.commit();//提交第二个事务
        } Catch(RunTimeException ex) {
            ts2.rollback();//回滚第二个事务
        } finally {
            //释放资源
        }
        //methodB执行完后，恢复第一个事务
        tm.resume(ts1);
        doSomeThingB();
        ts1.commit();//提交第一个事务
    } catch(RunTimeException ex) {
        ts1.rollback();//回滚第一个事务
    } finally {
        //释放资源
    }
}
```

在这里，我把ts1称为外层事务，ts2称为内层事务。从上面的代码可以看出，ts2与ts1是两个独立的事务，互不相干。Ts2是否成功并不依赖于 ts1。如果methodA方法在调用methodB方法后的doSomeThingB方法失败了，而methodB方法所做的结果依然被提交。而除了 methodB之外的其它代码导致的结果却被回滚了。使用PROPAGATION_REQUIRES_NEW,需要使用 JtaTransactionManager作为事务管理器。

（5）PROPAGATION_NOT_SUPPORTED 总是非事务地执行，并挂起任何存在的事务。使用PROPAGATION_NOT_SUPPORTED,也需要使用JtaTransactionManager作为事务管理器。（代码示例同上，可同理推出）

（6）PROPAGATION_NEVER 总是非事务地执行，如果存在一个活动事务，则抛出异常。

（7）PROPAGATION_NESTED如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按TransactionDefinition.PROPAGATION_REQUIRED 属性执行。这是一个嵌套事务,使用JDBC 3.0驱动时,仅仅支持DataSourceTransactionManager作为事务管理器。需要JDBC 驱动的java.sql.Savepoint类。有一些JTA的事务管理器实现可能也提供了同样的功能。使用PROPAGATION_NESTED，还需要把PlatformTransactionManager的nestedTransactionAllowed属性设为true;而 nestedTransactionAllowed属性值默认为false。

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
    doSomeThingA();
    methodB();
    doSomeThingB();
}

//事务属性 PROPAGATION_NESTED
methodB(){
    ……
}
```

如果单独调用methodB方法，则按REQUIRED属性执行。如果调用methodA方法，相当于下面的效果：

```
main(){
    Connection con = null;
    Savepoint savepoint = null;
    try{
        con = getConnection();
```

```
        con.setAutoCommit(false);
        doSomeThingA();
        savepoint = con2.setSavepoint();
        try{
            methodB();
        } catch(RuntimeException ex) {
            con.rollback(savepoint);
        } finally {
            //释放资源
        }
        doSomeThingB();
        con.commit();
    } catch(RuntimeException ex) {
        con.rollback();
    } finally {
        //释放资源
    }
}
```

当methodB方法调用之前，调用setSavepoint方法，保存当前的状态到savepoint。如果methodB方法调用失败，则恢复到之前保存的状态。但是需要注意的是，这时的事务并没有进行提交，如果后续的代码(doSomeThingB()方法)调用失败，则回滚包括methodB方法的所有操作。

嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

PROPAGATION_NESTED 与PROPAGATION_REQUIRES_NEW的区别:它们非常类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用 PROPAGATION_REQUIRES_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA事务管理器的支持。

使用PROPAGATION_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。DataSourceTransactionManager使用savepoint支持PROPAGATION_NESTED时，需要JDBC 3.0以上驱动及1.4以上的JDK版本支持。其它的JTA TrasactionManager实现可能有不同的支持方式。

PROPAGATION_REQUIRES_NEW 启动一个新的, 不依赖于环境的 “内部” 事务. 这个事务将被完全 commited 或 rolled back 而不依赖于外部事务, 它拥有自己的隔离范围, 自己的锁, 等等. 当内部事务开始执行时, 外部事务将被挂起, 内务事务结束时, 外部事务将继续执行。

另一方面, PROPAGATION_NESTED 开始一个 “嵌套的” 事务, 它是已经存在事务的一个真正的子事务. 潜套事务开始执行时, 它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 潜套事务是外部事务的一部分, 只有外部事务结束后它才会被提交。

由此可见, PROPAGATION_REQUIRES_NEW 和 PROPAGATION_NESTED 的最大区别在于, PROPAGATION_REQUIRES_NEW 完全是一个新的事务, 而 PROPAGATION_NESTED 则是外部事务的子事务, 如果外部事务 commit, 嵌套事务也会被 commit, 这个规则同样适用于 roll back.

PROPAGATION_REQUIRED应该是我们首先的事务传播行为。它能够满足我们大多数的事务需求。

2.2.2 隔离级别

事务的第二个维度就是隔离级别（isolation level）。隔离级别定义了一个事务可能受其他并发事务影响的程度。

（1）并发事务引起的问题

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。并发虽然是必须的，但可能会导致一下的问题。

> - 脏读（Dirty reads）——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。
> - 不可重复读（Nonrepeatable read）——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。
> - 幻读（Phantom read）——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

不可重复读与幻读的区别

不可重复读的重点是修改:
同样的条件, 你读取过的数据, 再次读取出来发现值不一样了
例如：在事务1中，Mary 读取了自己的工资为1000,操作并没有完成

```
    con1 = getConnection();
    select salary from employee empId ="Mary";
```

在事务2中，这时财务人员修改了Mary的工资为2000,并提交了事务.

```
    con2 = getConnection();
    update employee set salary = 2000;
    con2.commit();
```

在事务1中，Mary 再次读取自己的工资时，工资变为了2000

```
    //con1
    select salary from employee empId ="Mary";
```

在一个事务中前后两次读取的结果并不一致，导致了不可重复读。

幻读的重点在于新增或者删除:
同样的条件, 第1次和第2次读出来的记录数不一样
例如：目前工资为1000的员工有10人。事务1,读取所有工资为1000的员工。

```
    con1 = getConnection();
    Select * from employee where salary =1000;
```

共读取10条记录

这时另一个事务向employee表插入了一条员工记录，工资也为1000

```
    con2 = getConnection();
    Insert into employee(empId,salary) values("Lili",1000);
    con2.commit();
```

事务1再次读取所有工资为1000的员工

```
    //con1
    select * from employee where salary =1000;
```

共读取到了11条记录，这就产生了幻像读。

从总的结果来看, 似乎不可重复读和幻读都表现为两次读取的结果不一致。但如果你从控制的角度来看, 两者的区别就比较大。
对于前者, 只需要锁住满足条件的记录。
对于后者, 要锁住满足条件及其相近的记录。

（2）隔离级别

隔离级别	含义
ISOLATION_DEFAULT	使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED	最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
ISOLATION_READ_COMMITTED	允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
ISOLATION_REPEATABLE_READ	对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
ISOLATION_SERIALIZABLE	最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的

2.2.3 只读

事务的第三个特性是它是否为只读事务。如果事务只对后端的数据库进行该操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。

2.2.4 事务超时

为了使应用程序很好地运行，事务不能运行太长的时间。因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要的占用数据库资源。事务超时就是事务的一个定时器，在特定时间内事务如果没有执行完毕，那么就会自动回滚，而不是一直等待其结束。

2.2.5 回滚规则

事务五边形的最后一个方面是一组规则，这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）
但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

2.3 事务状态

上面讲到的调用PlatformTransactionManager接口的getTransaction()的方法得到的是TransactionStatus接口的一个实现，这个接口的内容如下：

```
public interface TransactionStatus{
    boolean isNewTransaction(); // 是否是新的事物
    boolean hasSavepoint(); // 是否有恢复点
    void setRollbackOnly();  // 设置为只回滚
    boolean isRollbackOnly(); // 是否为只回滚
    boolean isCompleted; // 是否已完成
}
```

可以发现这个接口描述的是一些处理事务提供简单的控制事务执行和查询事务状态的方法，在回滚或提交的时候需要应用对应的事务状态。

3 编程式事务

3.1 编程式和声明式事务的区别

Spring提供了对编程式事务和声明式事务的支持，编程式事务允许用户在代码中精确定义事务的边界，而声明式事务（基于AOP）有助于用户将操作与事务规则进行解耦。
简单地说，编程式事务侵入到了业务代码里面，但是提供了更加详细的事务管理；而声明式事务由于基于AOP，所以既能起到事务管理的作用，又可以不影响业务代码的具体实现。

3.2 如何实现编程式事务?

Spring提供两种方式的编程式事务管理，分别是：使用TransactionTemplate和直接使用PlatformTransactionManager。

3.2.1 使用TransactionTemplate

采用TransactionTemplate和采用其他Spring模板，如JdbcTempalte和HibernateTemplate是一样的方法。它使用回调方法，把应用程序从处理取得和释放资源中解脱出来。如同其他模板，TransactionTemplate是线程安全的。代码片段：

```
    TransactionTemplate tt = new TransactionTemplate(); // 新建一个
TransactionTemplate
    Object result = tt.execute(
        new TransactionCallback(){
            public Object doTransaction(TransactionStatus status){
                updateOperation();
                return resultOfUpdateOperation();
            }
    }); // 执行execute方法进行事务管理
```

使用TransactionCallback()可以返回一个值。如果使用TransactionCallbackWithoutResult则没有返回值。

3.2.2 使用PlatformTransactionManager

示例代码如下：

```
    DataSourceTransactionManager dataSourceTransactionManager = new
DataSourceTransactionManager(); //定义一个某个框架平台的TransactionManager，如JDBC、
Hibernate

dataSourceTransactionManager.setDataSource(this.getJdbcTemplate().getDataSource(
)); // 设置数据源
    DefaultTransactionDefinition transDef = new DefaultTransactionDefinition();
// 定义事务属性

transDef.setPropagationBehavior(DefaultTransactionDefinition.PROPAGATION_REQUIRE
D); // 设置传播行为属性
    TransactionStatus status =
dataSourceTransactionManager.getTransaction(transDef); // 获得事务状态
    try {
        // 数据库操作
        dataSourceTransactionManager.commit(status);// 提交
    } catch (Exception e) {
        dataSourceTransactionManager.rollback(status);// 回滚
    }
```

4 声明式事务

4.1 配置方式

*注：以下配置代码参考自Spring事务配置的五种方式

根据代理机制的不同，总结了五种Spring事务的配置方式，配置文件如下：

（1）每个Bean都有一个代理

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <!-- 配置DAO -->
    <bean id="userDaoTarget" class="com.bluesky.spring.dao.UserDaoImpl">
```

```
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="userDao"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

           <!-- 配置事务管理器 -->
           <property name="transactionManager" ref="transactionManager" />
        <property name="target" ref="userDaoTarget" />
         <property name="proxyInterfaces"
value="com.bluesky.spring.dao.GeneratorDao" />
        <!-- 配置事务属性 -->
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>
</beans>
```

(2) 所有Bean共享一个代理基类

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="transactionBase"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
            lazy-init="true" abstract="true">
        <!-- 配置事务管理器 -->
        <property name="transactionManager" ref="transactionManager" />
        <!-- 配置事务属性 -->
        <property name="transactionAttributes">
            <props>
```

```
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <!-- 配置DAO -->
    <bean id="userDaoTarget" class="com.bluesky.spring.dao.UserDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="userDao" parent="transactionBase" >
        <property name="target" ref="userDaoTarget" />
    </bean>
</beans>
```

(3) 使用拦截器

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="transactionInterceptor"

class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <property name="transactionManager" ref="transactionManager" />
        <!-- 配置事务属性 -->
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <bean
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames">
```

```
            <list>
                <value>*Dao</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
            </list>
        </property>
    </bean>

    <!-- 配置DAO -->
    <bean id="userDao" class="com.bluesky.spring.dao.UserDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

（4）使用tx标签配置的拦截器

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:annotation-config />
    <context:component-scan base-package="com.bluesky" />

    <bean id="sessionFactory"
           class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="*" propagation="REQUIRED" />
        </tx:attributes>
    </tx:advice>

    <aop:config>
```

```
        <aop:pointcut id="interceptorPointCuts"
            expression="execution(* com.bluesky.spring.dao.*.*(..))" />
        <aop:advisor advice-ref="txAdvice"
            pointcut-ref="interceptorPointCuts" />
    </aop:config>
</beans>
```

(5) 全注解

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:annotation-config />
    <context:component-scan base-package="com.bluesky" />

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>

    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

</beans>
```

此时在DAO上需加上@Transactional注解，如下：

```
package com.bluesky.spring.dao;

import java.util.List;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.stereotype.Component;

import com.bluesky.spring.domain.User;
```

```
@Transactional
@Component("userDao")
public class UserDaoImpl extends HibernateDaoSupport implements UserDao {

    public List<User> listUsers() {
        return this.getSession().createQuery("from User").list();
    }
}
```

4.2 一个声明式事务的实例

*注：该实例参考自Spring中的事务管理实例详解

首先是数据库表
book(isbn, book_name, price)
account(username, balance)
book_stock(isbn, stock)

然后是XML配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <import resource="applicationContext-db.xml" />

    <context:component-scan
        base-package="com.springinaction.transaction">
    </context:component-scan>

    <tx:annotation-driven transaction-manager="txManager"/>

    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>
```

使用的类
BookShopDao

```
package com.springinaction.transaction;

public interface BookShopDao {
    // 根据书号获取书的单价
    public int findBookPriceByIsbn(String isbn);
    // 更新书的库存，使书号对应的库存-1
    public void updateBookStock(String isbn);
    // 更新用户的账户余额：account的balance-price
    public void updateUserAccount(String username, int price);
}
```

BookShopDaoImpl

```
package com.springinaction.transaction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository("bookShopDao")
public class BookShopDaoImpl implements BookShopDao {

    @Autowired
    private JdbcTemplate JdbcTemplate;

    @Override
    public int findBookPriceByIsbn(String isbn) {
        String sql = "SELECT price FROM book WHERE isbn = ?";

        return JdbcTemplate.queryForObject(sql, Integer.class, isbn);
    }

    @Override
    public void updateBookStock(String isbn) {
        //检查书的库存是否足够，若不够，则抛出异常
        String sql2 = "SELECT stock FROM book_stock WHERE isbn = ?";
        int stock = JdbcTemplate.queryForObject(sql2, Integer.class, isbn);
        if (stock == 0) {
            throw new BookStockException("库存不足！");
        }
        String sql = "UPDATE book_stock SET stock = stock - 1 WHERE isbn = ?";
        JdbcTemplate.update(sql, isbn);
    }

    @Override
    public void updateUserAccount(String username, int price) {
        //检查余额是否不足，若不足，则抛出异常
        String sql2 = "SELECT balance FROM account WHERE username = ?";
        int balance = JdbcTemplate.queryForObject(sql2, Integer.class,
username);
        if (balance < price) {
            throw new UserAccountException("余额不足！");
        }
        String sql = "UPDATE account SET balance = balance - ? WHERE username =
?";
        JdbcTemplate.update(sql, price, username);
```

```
    }

}
```

BookShopService

```
package com.springinaction.transaction;
public interface BookShopService {
     public void purchase(String username, String isbn);
}
```

BookShopServiceImpl

```
package com.springinaction.transaction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Service("bookShopService")
public class BookShopServiceImpl implements BookShopService {

    @Autowired
    private BookShopDao bookShopDao;

    /**
     * 1.添加事务注解
     * 使用propagation 指定事务的传播行为，即当前的事务方法被另外一个事务方法调用时如何使用事务。
     * 默认取值为REQUIRED，即使用调用方法的事务
     * REQUIRES_NEW：使用自己的事务，调用的事务方法的事务被挂起。
     *
     * 2.使用isolation 指定事务的隔离级别，最常用的取值为READ_COMMITTED
     * 3.默认情况下 Spring 的声明式事务对所有的运行时异常进行回滚，也可以通过对应的属性进行设置。通常情况下，默认值即可。
     * 4.使用readOnly 指定事务是否为只读。 表示这个事务只读取数据但不更新数据，这样可以帮助数据库引擎优化事务。若真的是一个只读取数据库值得方法，应设置readOnly=true
     * 5.使用timeOut 指定强制回滚之前事务可以占用的时间。
     */
    @Transactional(propagation=Propagation.REQUIRES_NEW,
            isolation=Isolation.READ_COMMITTED,
            noRollbackFor={UserAccountException.class},
            readOnly=true, timeout=3)
    @Override
    public void purchase(String username, String isbn) {
        //1.获取书的单价
        int price = bookShopDao.findBookPriceByIsbn(isbn);
        //2.更新书的库存
        bookShopDao.updateBookStock(isbn);
        //3.更新用户余额
        bookShopDao.updateUserAccount(username, price);
    }
}
```

Cashier

```
package com.springinaction.transaction;
import java.util.List;
public interface Cashier {
    public void checkout(String username, List<String>isbns);
}
```

CashierImpl：CashierImpl.checkout和bookShopService.purchase联合测试了事务的传播行为

```
package com.springinaction.transaction;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service("cashier")
public class CashierImpl implements Cashier {
    @Autowired
    private BookShopService bookShopService;

    @Transactional
    @Override
    public void checkout(String username, List<String> isbns) {
        for(String isbn : isbns) {
            bookShopService.purchase(username, isbn);
        }
    }
}
```

BookStockException

```
package com.springinaction.transaction;
public class BookStockException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public BookStockException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public BookStockException(String arg0, Throwable arg1, boolean arg2,
            boolean arg3) {
        super(arg0, arg1, arg2, arg3);
        // TODO Auto-generated constructor stub
    }

    public BookStockException(String arg0, Throwable arg1) {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    public BookStockException(String arg0) {
```

```
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public BookStockException(Throwable arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }
}
```

UserAccountException

```
package com.springinaction.transaction;
public class UserAccountException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public UserAccountException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(String arg0, Throwable arg1, boolean arg2,
            boolean arg3) {
        super(arg0, arg1, arg2, arg3);
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(String arg0, Throwable arg1) {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(String arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public UserAccountException(Throwable arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }
}
```

测试类

```
package com.springinaction.transaction;

import java.util.Arrays;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringTransitionTest {
```

```
    private ApplicationContext ctx = null;
    private BookShopDao bookShopDao = null;
    private BookShopService bookShopService = null;
    private Cashier cashier = null;
    {
        ctx = new ClassPathXmlApplicationContext("config/transaction.xml");
        bookShopDao = ctx.getBean(BookShopDao.class);
        bookShopService = ctx.getBean(BookShopService.class);
        cashier = ctx.getBean(Cashier.class);
    }

    @Test
    public void testBookShopDaoFindPriceByIsbn() {
        System.out.println(bookShopDao.findBookPriceByIsbn("1001"));
    }

    @Test
    public void testBookShopDaoUpdateBookStock(){
        bookShopDao.updateBookStock("1001");
    }

    @Test
    public void testBookShopDaoUpdateUserAccount(){
        bookShopDao.updateUserAccount("AA", 100);
    }
    @Test
    public void testBookShopService(){
        bookShopService.purchase("AA", "1001");
    }

    @Test
    public void testTransactionPropagation(){
        cashier.checkout("AA", Arrays.asList("1001", "1002"));
    }
}
```

SpringMVC概述

什么是Spring MVC？简单介绍下你对Spring MVC的理解?

Spring MVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级 Web框架，通过把模型-视图-控制器分离，将web层进行职责解耦，把复杂的 web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间 的配合。

Spring MVC的优点

（1）可以支持各种视图技术,而不仅仅局限于JSP；

（2）与Spring框架集成（如IoC容器、AOP等）；

（3）清晰的角色分配：前端控制器(dispatcherServlet) , 请求到处理器映射 （handlerMapping), 处理器适配器（HandlerAdapter), 视图解析器 （ViewResolver）。

（4） 支持各种请求资源的映射策略。 ### 核心组件

Spring MVC的主要组件?

（1）前端控制器 DispatcherServlet（不需要程序员开发）
作用：接收请求、响应结果，相当于转发器，有了DispatcherServlet 就减少了 其它组件之间的耦合度。

（2）处理器映射器HandlerMapping（不需要程序员开发）
作用：根据请求的URL来查找Handler

（3）处理器适配器HandlerAdapter
注意：在编写Handler的时候要按照HandlerAdapter要求的规则去编写，这样适配器HandlerAdapter才可以正确的去执行Handler。

（4）处理器Handler（需要程序员开发）

（5）视图解析器 ViewResolver（不需要程序员开发）
作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）

（6）视图View（需要程序员开发jsp）
View是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf等 等）

什么是DispatcherServlet

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的 HTTP请求和响应。

什么是Spring MVC框架的控制器?

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解 析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽 象的方式实现了一个控制层，允许用户创建多种用途的控制器。

Spring MVC的控制器是不是单例模式,如果是,有什么问题,怎么解决?

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性 能的,解决方案是在控制器里面不能写字段。

工作原理

请描述Spring MVC的工作流程？描述一下 DispatcherServlet 的工作流程?

（1）用户发送请求至前端控制器DispatcherServlet；

（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器， 请求获取Handle；

（3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦 截器(如果有则生成)一并返回给DispatcherServlet；

（4）DispatcherServlet 调用 HandlerAdapter处理器适配器；

（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制 器)；

（6）Handler执行完成返回ModelAndView；

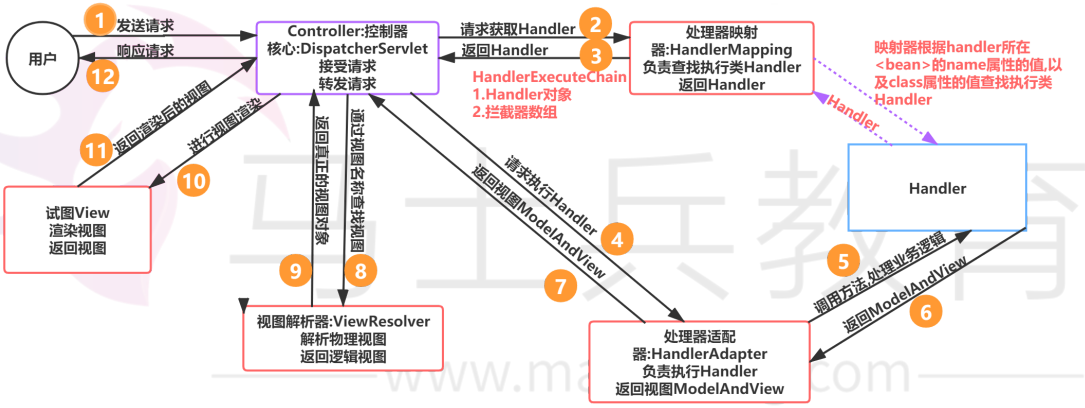
（7）HandlerAdapter将Handler执行结果ModelAndView返回给 DispatcherServlet；

（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行 解析；

（9）ViewResolver解析后返回具体View；

（10）DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）

（11）DispatcherServlet响应用户。



MVC框架

MVC是什么？MVC设计模式的好处有哪些

mvc是一种设计模式（设计模式就是日常开发中编写代码的一种好的方法和经验 的总结）。模型（model）-视图（view）-控制器（controller），三层架构的 设计模式。用于实现前端页面的展现与后端业务数据处理的分离。

mvc设计模式的好处

1.分层设计，实现了业务系统各个组件之间的解耦，有利于业务系统的可扩展 性，可维护性。

2.有利于系统的并行开发，提升开发效率。

常用注解

注解原理是什么

注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生 成的动态代理类。我们通过反射获取注解时，返回的是Java运行时生成的动态代 理对象。通过代理对象调用自定义注解的方法，会终调用 AnnotationInvocationHandler的invoke方法。该方法会从memberValues这 个Map中索引出对应的值。而memberValues的来源是Java常量池。

Spring MVC常用的注解有哪些?

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用 于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody：注解实现接收http请求的json数据，将json转换为java对 象。

@ResponseBody：注解实现将conreoller方法返回对象转化为json对象响应给 客户。

SpingMvc中的控制器的注解一般用哪个,有没有别的注解可以替代?

答：一般用@Controller注解,也可以使用@RestController,@RestController 注解相当于@ResponseBody + @Controller,表示是表现层,除此之外，一般不用别的注解代替。

@Controller注解的作用

在Spring MVC 中，控制器Controller 负责处理由DispatcherServlet 分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个Model ，然后再把该Model 返回给对应的View 进行展示。在 Spring MVC 中提供了一个非常简

便的定义Controller 的方法，你无需继承特定的类或实现特定的接口，只需使

用@Controller 标记一个类是Controller，然后使用@RequestMapping 和 @RequestParam 等一些注解用以定义URL 请求和Controller 方法之间的映射，这样的Controller 就能被外界访问到。此外Controller 不会直接依赖于

HttpServletRequest 和HttpServletResponse 等HttpServlet 对象，它们可以通过Controller 的方法参数灵活的获取到。

@Controller 用于标记在一个类上，使用它标记的类就是一个Spring MVC

Controller 对象。分发处理器将会扫描使用了该注解的类的方法，并检测该方

法是否使用了@RequestMapping 注解。@Controller 只是定义了一个控制器类，而使用@RequestMapping 注解的方法才是真正处理请求的处理器。单单使用@Controller 标记在一个类上还不能真正意义上的说它就是Spring MVC 的一个控制器类，因为这个时候Spring 还不认识它。那么要如何做Spring 才能认识它呢？这个时候就需要我们把这个控制器类交给Spring 来管理。有两种方式：

- 在Spring MVC 的配置文件中定义MyController 的bean 对象。
- 在Spring MVC 的配置文件中告诉Spring 该到哪里去找标记为@Controller 的Controller 控制器。

@RequestMapping注解的作用

RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。

用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

RequestMapping注解有六个属性，下面我们把她分成三类进行说明（下面有相应示例）。

value， method

value： 指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；

method： 指定请求的method类型， GET、POST、PUT、DELETE等； consumes，produces

consumes： 指定处理请求的提交内容类型（Content-Type），例如 application/json, text/html;

produces: 指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回；params，headers

params： 指定request中必须包含某些参数值是，才让该方法处理。

headers： 指定request中必须包含某些指定的header值，才能让该方法处理请求。

@ResponseBody注解的作用

作用： 该注解用于将Controller的方法返回的对象，通过适当的

HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。

使用时机：返回的数据不是html标签的页面，而是其他某种格式的数据时（如json、xml等）使用；

@PathVariable和@RequestParam的区别

请求路径上有个id的变量值，可以通过@PathVariable来获取

@RequestMapping(value = “/page/{id}”, method =

RequestMethod.GET)

@RequestParam用来获得静态的URL请求入参 spring注解时action里用到。

其他

Spring MVC与Struts2区别

相同点都是基于mvc的表现层框架，都用于web项目的开发。

不同点

1.前端控制器不一样。Spring MVC的前端控制器是servlet：

DispatcherServlet。struts2的前端控制器是filter：

StrutsPreparedAndExcutorFilter。

2.请求参数的接收方式不一样。Spring MVC是使用方法的形参接收请求的参数，基于方法的开发，线程安全，可以设计为单例或者多例的开发，推荐使用单例模式的开发（执行效率更高），默认就是单例开发模式。struts2是通过类的成员变量接收请求的参数，是基于类的开发，线程不安全，只能设计为多例的开发。

3.Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，Spring MVC 通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象， 后又将ModelAndView中的模型数据通过 reques域传输到页面。Jsp视图解析器默认使用jstl。

4.与spring整合不一样。Spring MVC是spring框架的一部分，不需要整合。在企业项目中，Spring MVC使用更多一些。

Spring MVC怎么样设定重定向和转发的?

（1） 转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"

（2） 重定向：在返回值前面加"redirect:"，譬如"redirect:http://www.baidu.com"

Spring MVC怎么和AJAX相互调用的?

通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象。

具体步骤如下：

（1） 加入Jackson.jar

（2） 在配置文件中配置json的映射

（3） 在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解。

如何解决POST请求中文乱码问题，GET的又如何处理呢?

（1） 解决post请求乱码问题：

在web.xml中配置一个CharacterEncodingFilter过滤器，设置成utf-8；

```
1 <filter>
2 <filter-name>CharacterEncodingFilter</filter-name>
3 <filter-class>org.springframework.web.filter.CharacterEncodingFilter</fi
lter-class>
4
5 <init-param>
6 <param-name>encoding</param-name>
7 <param-value>utf-8</param-value>
8 </init-param>
9 </filter>
10
```

```
11 <filter-mapping>
12 <filter-name>CharacterEncodingFilter</filter-name>
13 <url-pattern>/*</url-pattern>
14 </filter-mapping>
```

（2）get请求中文参数出现乱码解决方法有两个：

①修改tomcat配置文件添加编码与工程编码一致，如下：

```
1 <ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" proto
col="HTTP/1.1" redirectPort="8443"/>
```

②另外一种方法对参数进行重新编码：

String userName = new

String(request.getParamter(“userName”).getBytes(“ISO8859-

1”),“utf-8”)

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码。

Spring MVC的异常处理?

答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

如果在拦截请求中，我想拦截get方式提交的方法,怎么配置

答：可以在@RequestMapping注解里面加上 method=RequestMethod.GET。

怎样在方法里面得到Request,或者Session?

答：直接在方法的形参中声明request,Spring MVC就自动把request对象传入。

如果想在拦截的方法里面得到从前台传入的参数,怎么得到?

答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?

答：直接在方法中声明这个对象,Spring MVC就自动会把属性赋值到这个对象里面。

Spring MVC中函数的返回值是什么?

答：返回值可以有很多类型,有String, ModelAndView。ModelAndView类把视图和数据都合并的一起的，但一般用String比较好。

Spring MVC用什么对象从后台向前台传递数据的?

答：通过ModelMap对象,可以在这个对象里面调用put方法,把对象加到里面,前台就可以通过el表达式拿到。

怎么样把ModelMap里面的数据放入Session里面?

答：可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key。

Spring MVC里面拦截器是怎么写的

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在Spring MVC的配置文件中配置拦截器即可：

```
1 <!-- 配置Spring MVC的拦截器 -->
2 <mvc:interceptors>
3 <!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 -->
4 <bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor"></b
ean>
5 <!-- 只针对部分请求拦截 -->
6 <mvc:interceptor>
7 <mvc:mapping path="/modelMap.do" />
8 <bean class="com.zwp.action.MyHandlerInterceptorAdapter" />
9 </mvc:interceptor>
10 </mvc:interceptors>
```

介绍一下 WebApplicationContext

WebApplicationContext 继承了ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

Spring概述

1. 什么是spring?

Spring是**一个轻量级Java开发框架**，最早由**Rod Johnson**创建，目的是为了解 决企业级应用开发的业务逻辑层和其他各层的耦合问题。它是一个分层的 JavaSE/JavaEE full-stack（一站式）轻量级开源框架，为开发Java应用程序提 供全面的基础架构支持。Spring负责基础架构，因此Java开发者可以专注于应 用程序的开发。 Spring最根本的使命是**解决企业级应用开发的复杂性，即简化Java开发。**

Spring可以做很多事情，它为企业级开发提供给了丰富的功能，但是这些功能 的底层都依赖于它的两个核心特性，也就是**依赖注入（dependency injection，DI）和面向切面编程（aspect-oriented programming， AOP）。**

为了降低Java开发的复杂性，Spring采取了以下4种关键策略

- 基于POJO的轻量级和最小侵入性编程；
- 通过依赖注入和面向接口实现松耦合；
- 基于切面和惯例进行声明式编程；
- 通过切面和模板减少样板式代码。

2. Spring框架的设计目标，设计理念，和核心是什么

Spring设计目标：Spring为开发者提供一个一站式轻量级应用开发平台；

Spring设计理念：在JavaEE开发中，支持POJO和JavaBean开发方式，使应用 面向接口开发，充分支持OO（面向对象）设计方法；Spring通过IoC容器实现 对象耦合关系的管理，并实现依赖反转，将对象之间的依赖关系交给IoC容器， 实现解耦；

Spring框架的核心：IoC容器和AOP模块。通过IoC容器管理POJO对象以及他 们之间的耦合关系；通过AOP以动态非侵入的方式增强服务。 IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的 功能分离出来形成可重用的功能组件。

3. Spring的优缺点是什么?

优点

- 方便解耦，简化开发
  Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护，交给 Spring管理。
- AOP编程的支持
  Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等 功能。
- 声明式事务的支持
  只需要通过配置就可以完成对事务的管理，而无需手动编程。
- 方便程序的测试
  Spring对Junit4支持，可以通过注解方便的测试Spring程序。
- 方便集成各种优秀框架
  Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持 （如：Struts、Hibernate、MyBatis等）。
- 降低JavaEE API的使用难度
  Spring对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用 等），都提供了封装，使这些API应用难度大大降低。

缺点

- Spring明明一个很轻量级的框架，却给人感觉大而全
- Spring依赖反射，反射影响性能
- 使用门槛升高，入门Spring需要较长时间

4. Spring有哪些应用场景

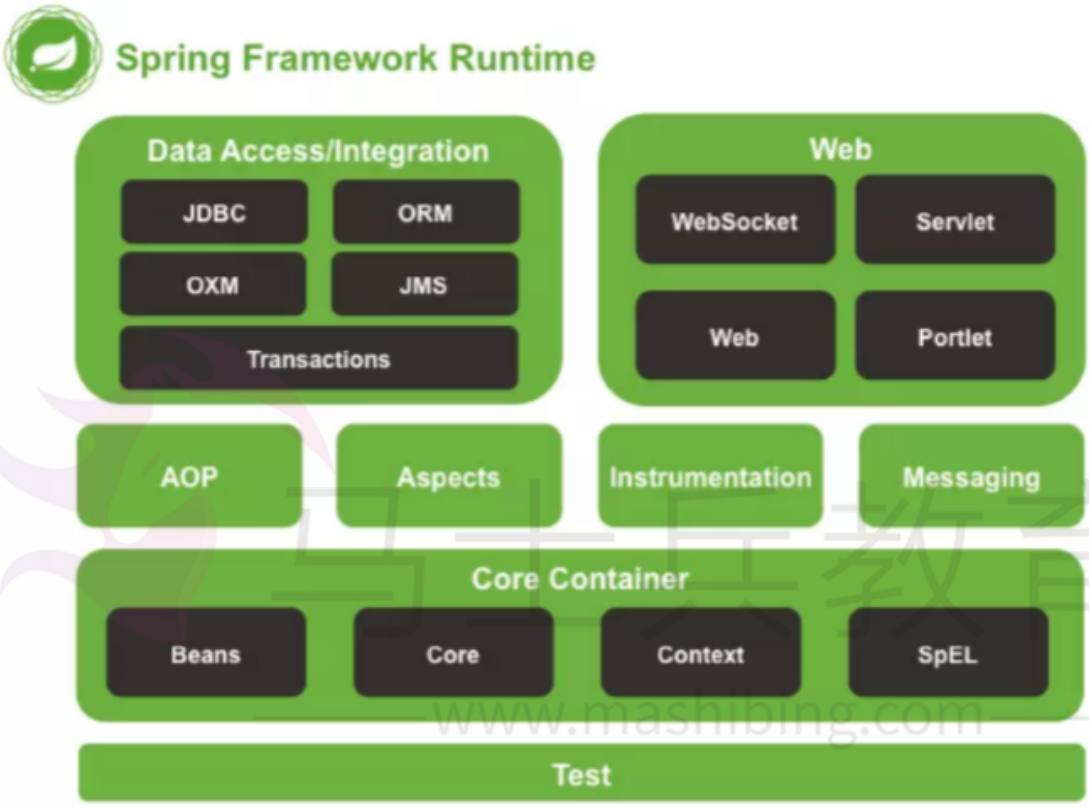
应用场景：JavaEE企业应用开发，包括SSH、SSM等

Spring价值：

- Spring是非侵入式的框架，目标是使应用程序代码对框架依赖最小化；
- Spring提供一个一致的编程模型，使应用直接使用POJO开发，与运行环境隔离 开来；
- Spring推动应用设计风格向面向对象和面向接口开发转变，提高了代码的重用性 和可测试性；

5. Spring由哪些模块组成?

Spring 总共大约有 20 个模块， 由 1300 多个不同的文件构成。 而这些组件被 分别整合在**核心容器（Core Container）、 AOP（Aspect Oriented Programming） 和设备支持（Instrmentation）、数据访问与集成（Data Access/Integeration）、 Web、 消息（Messaging）、 Test**等 6 个模块中。 以下是 Spring 5 的模块结构图：



- spring core：提供了框架的基本组成部分，包括控制反转（Inversion of Control，IOC）和依赖注入（Dependency Injection，DI）功能。
- spring beans：提供了BeanFactory，是工厂模式的一个经典实现，Spring将管 理对象称为Bean。
- spring context：构建于 core 封装包基础上的 context 封装包，提供了一种框 架式的对象访问方法。
- spring jdbc：提供了一个JDBC的抽象层，消除了烦琐的JDBC编码和数据库厂 商特有的错误代码解析， 用于简化JDBC。
- spring aop：提供了面向切面的编程实现，让你可以自定义拦截器、切点等。
- spring Web：提供了针对 Web 开发的集成特性，例如文件上传，利用 servlet listeners 进行 ioc 容器初始化和针对 Web 的 ApplicationContext。
- spring test：主要为测试提供支持的，支持使用JUnit或TestNG对Spring组件进 行单元测试和集成测试。

6. Spring 框架中都用到了哪些设计模式?

1. 工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实 例；
2. 单例模式：Bean默认为单例模式。
3. 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生 成技术；
4. 模板方法：用来解决代码重复的问题。比如. RestTemplate，JmsTemplate, JpaTemplate。
5. 观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发 生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中 listener的实现–ApplicationListener。

7. 详细讲解一下核心容器（spring context应用上下文) 模 块

这是基本的Spring模块，提供spring 框架的基础功能，BeanFactory 是 任何以 spring为基础的应用的核心。Spring 框架建立在此模块之上，它使Spring成为 一个容器。

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和 依赖从真正的应用代码中分离。最常用的就是 org.springframework.beans.factory.xml.XmlBeanFactory ，它根据XML文件中的定义加载beans。该容器从XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

8. Spring框架中有哪些不同类型的事件

Spring 提供了以下5种标准的事件：

1. 上下文更新事件（ContextRefreshedEvent）：在调用 ConfigurableApplicationContext 接口中的refresh()方法时被触发。
2. 上下文开始事件（ContextStartedEvent）：当容器调用 ConfigurableApplicationContext的Start()方法开始/重新开始容器时触 发该事件。
3. 上下文停止事件（ContextStoppedEvent）：当容器调用 ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。
4. 上下文关闭事件（ContextClosedEvent）：当ApplicationContext被 关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。
5. 请求处理事件（RequestHandledEvent）：在Web应用中，当一个 http请求（request）结束触发该事件。如果一个bean实现了 ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean 会自动被通知。

9. Spring 应用程序有哪些不同组件?

Spring 应用一般有以下组件：

- 接口 - 定义功能。
- Bean 类 - 它包含属性，setter 和 getter 方法，函数等。
- Bean 配置文件 - 包含类的信息以及如何配置它们。
- Spring 面向切面编程（AOP） - 提供面向切面编程的功能。
- 用户程序 - 它使用接口。

10. 使用 Spring 有哪些方式?

使用 Spring 有以下方式：

- 作为一个成熟的 Spring Web 应用程序。
- 作为第三方 Web 框架，使用 Spring Frameworks 中间层。
- 作为企业级 Java Bean，它可以包装现有的 POJO（Plain Old Java Objects）。
- 用于远程使用。

Spring控制反转(IOC)

什么是Spring IOC 容器?

控制反转即IoC (Inversion of Control)，它把传统上由程序代码直接操控的对

象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反

转"概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

控制反转(IoC)有什么作用

- 管理对象的创建和依赖关系的维护。对象的创建并不是一件简单的事，在对象关系比较复杂时，如果依赖关系需要程序猿来维护的话，那是相当头疼的
- 解耦，由容器去维护具体的对象
- 托管了类的产生过程，比如我们需要在类的产生过程中做一些处理，最直接的例子就是代理，如果有容器程序可以把这部分处理交给容器，应用程序则无需去关心类是如何完成代理的

IOC的优点是什么?

- IOC 或 依赖注入把应用的代码量降到最低。
- 它使应用容易测试，单元测试不再需要单例和JNDI查找机制。
- 最小的代价和最小的侵入性使松散耦合得以实现。
- IOC容器支持加载服务时的饿汉式初始化和懒加载。

Spring IoC 的实现机制

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

示例：

```
1 interface Fruit {
2 public abstract void eat();
3 }
4
5 class Apple implements Fruit {
6 public void eat(){
7 System.out.println("Apple");
8 }
9 }
10
11 class Orange implements Fruit {
12 public void eat(){
13 System.out.println("Orange");
14 }
15 }
16
17 class Factory {
18 public static Fruit getInstance(String ClassName) {
19 Fruit f=null;
20 try {
21 f=(Fruit)Class.forName(ClassName).newInstance();
22 } catch (Exception e) {
23 e.printStackTrace();
24 }
25 return f;
26 }
27 }
28
29 class Client {
30 public static void main(String[] a) {
31 Fruit f=Factory.getInstance("io.github.dunwu.spring.Apple");
32 if(f!=null){
33 f.eat();
34 }
35 }
36 }
```

Spring 的 IoC支持哪些功能

Spring 的 IoC 设计支持以下功能：依赖注入依赖检查自动装配支持集合指定初始化方法和销毁方法

支持回调某些方法（但是需要实现 Spring 接口，略有侵入）

其中，最重要的就是依赖注入，从 XML 的配置上说，即 ref 标签。对应 Spring

RuntimeBeanReference 对象。

对于 IoC 来说，最重要的就是容器。容器管理着 Bean 的生命周期，控制着 Bean 的依赖注入。

BeanFactory 和 ApplicationContext有什么区别?

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做 Spring的容器。其中ApplicationContext是BeanFactory的子接口。

依赖关系

BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取 bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean 之间的依赖关系。

ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具

有的功能外，还提供了更完整的框架功能：

- 继承MessageSource，因此支持国际化。统一的资源文件访问方式。
- 提供在监听器中注册bean的事件。
- 同时加载多个配置文件。
- 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

加载方式

BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean 时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所

依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

创建方式

BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

注册方式

BeanFactory和ApplicationContext都支持BeanPostProcessor、

BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

Spring 如何设计容器的，BeanFactory和 ApplicationContext的关系详解

Spring 作者 Rod Johnson 设计了两个接口用以表示容器。

- BeanFactory
- ApplicationContext

BeanFactory 简单粗暴，可以理解为就是个 HashMap，Key 是 BeanName，

Value 是 Bean 实例。通常只提供注册（put），获取（get）这两个功能。我们可以称之为 “低级容器”。

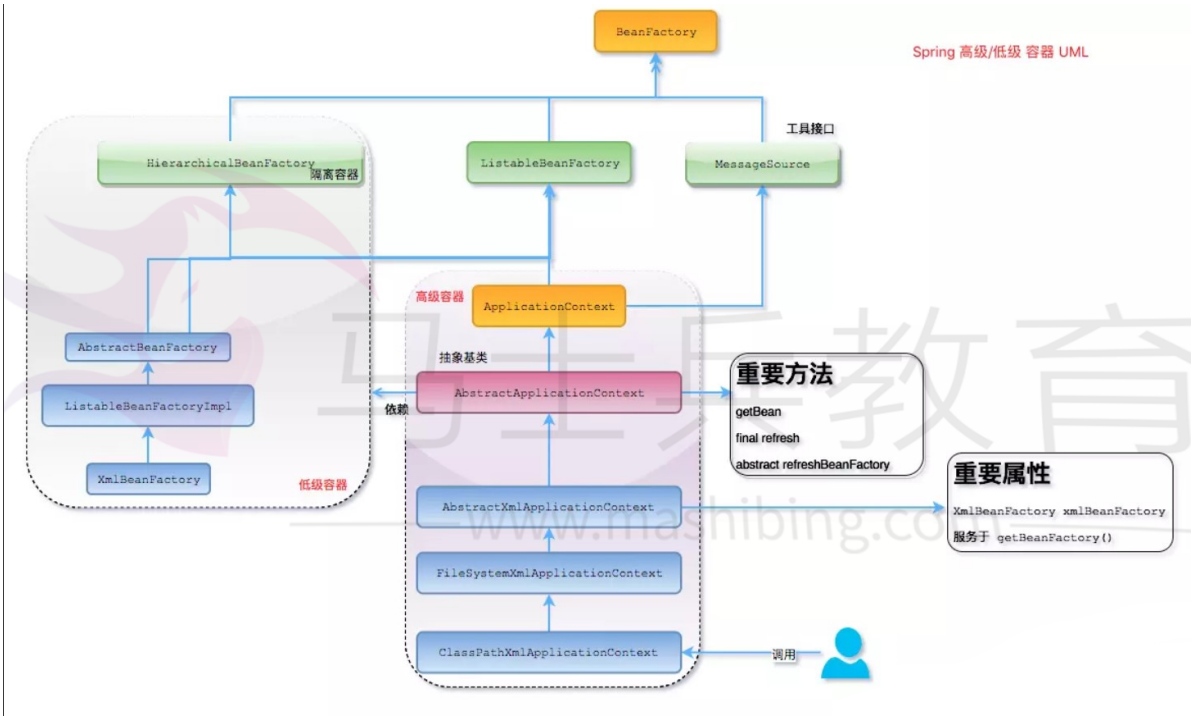
ApplicationContext 可以称之为 “高级容器”。因为他比 BeanFactory 多了更多的功能。他继承了多个接口。因此具备了更多的功能。例如资源的获取，支持多种消息（例如 JSP tag 的支持），对 BeanFactory 多了工具级别的支持等待。所以你看他的名字，已经不是 BeanFactory 之类的工厂了，而是 “应用上下文”， 代表着整个大容器的所有功能。该接口定义了一个 refresh 方法，此方法是所有阅读 Spring 源码的人的最熟悉的方法，用于刷新整个容器，即重新加载/刷新所有的 bean。

当然，除了这两个大接口，还有其他的辅助接口，这里就不介绍他们了。

BeanFactory和ApplicationContext的关系

为了更直观的展示 “低级容器” 和 “高级容器” 的关系，这里通过常用的

ClassPathXmlApplicationContext 类来展示整个容器的层级 UML 关系。



有点复杂？ 先不要慌，我来解释一下。

最上面的是 BeanFactory，下面的 3 个绿色的，都是功能扩展接口，这里就不展开讲。

看下面的隶属 ApplicationContext 粉红色的 “高级容器”，依赖着 “低级容器”，这里说的是依赖，不是继承哦。他依赖着 “低级容器” 的 getBean 功

能。而高级容器有更多的功能：支持不同的信息源头，可以访问文件资源，支持应用事件（Observer 模式）。

通常用户看到的就是 “高级容器”。 但 BeanFactory 也非常够用啦！左边灰色区域的是 “低级容器”， 只负载加载 Bean，获取 Bean。容器其他的高级功能是没有的。例如上图画的 refresh 刷新 Bean 工厂所有配置，生命周期事件回调等。

小结

说了这么多，不知道你有没有理解Spring IoC？ 这里小结一下：IoC 在 Spring 里，只需要低级容器就可以实现，2 个步骤：

1. 加载配置文件，解析成 BeanDefinition 放在 Map 里。
2. 调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出

Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法 —— 完成依赖注入。

上面就是 Spring 低级容器（BeanFactory）的 IoC。

至于高级容器 ApplicationContext，他包含了低级容器的功能，当他执行 refresh 模板方法的时候，将刷新整个容器的 Bean。同时其作为高级容器，包含了太多的功能。一句话，他不仅仅是 IoC。他支持不同信息源头，支持

BeanFactory 工具类，支持层级容器，支持访问文件资源，支持事件发布通知，支持接口回调等等。

ApplicationContext通常的实现是什么?

FileSystemXmlApplicationContext ： 此容器从一个XML文件中加载beans 的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

ClassPathXmlApplicationContext：此容器也从一个XML文件中加载beans 的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找 bean配置。

WebXmlApplicationContext：此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

什么是Spring的依赖注入?

控制反转IoC是一个很大的概念，可以用不同的方式来实现。其主要实现方式有两种：依赖注入和依赖查找依赖注入：相对于IoC而言，依赖注入(DI)更加准确地描述了IoC的设计理念。所谓依赖注入（Dependency Injection），即组件之间的依赖关系由容器在应用系统运行期来决定，也就是由容器动态地将某种依赖关系的目标对象实例注入到应用系统中的各个关联的组件之中。组件不做定位查询，只提供普通的Java方法让容器去决定依赖关系。

依赖注入的基本原则

依赖注入的基本原则是：应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由IoC容器负责，“查找资源”的逻辑应该从应用组件的代码中抽取出来，交给IoC容器负责。容器全权负责组件的装配，它会把符合依赖关系的对象通过属性（JavaBean中的setter）或者是构造器传递给需要的对象。

依赖注入有什么优势

依赖注入之所以更流行是因为它是一种更可取的方式：让容器全权负责依赖查询，受管组件只需要暴露JavaBean的setter方法或者带参数的构造器或者接

口，使容器可以在初始化时组装对象的依赖关系。其与依赖查找方式相比，主要优势为：

- 查找定位操作与应用代码完全无关。
- 不依赖于容器的API，可以很容易地在任何容器以外使用应用对象。
- 不需要特殊的接口，绝大多数对象可以做到完全不必依赖容器。

有哪些不同类型的依赖注入实现方式?

依赖注入是时下最流行的IoC实现方式，依赖注入分为接口注入（Interface Injection），Setter方法注入（Setter Injection）和构造器注入（Constructor

Injection）三种方式。其中接口注入由于在灵活性和易用性比较差，现在从Spring4开始已被废弃。

构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。

Setter方法注入：Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

构造器依赖注入和 Setter方法注入的区别

构造函数注入	setter注 入
没有部分注入	有部分注入
不会覆盖 setter 属 性	会覆盖 setter 属 性
任意修改都会创建一个新实例	任意修改不会创建一个新实例
适用于设置很多属性	适用于设置少量属性

两种依赖方式都可以使用，构造器注入和Setter方法注入。最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

Spring Beans（19）什么是Spring beans?

Spring beans 是那些形成Spring应用的主干的java对象。它们被Spring IOC容器初始化，装配，和管理。这些beans通过容器中配置的元数据创建。比如，以XML文件中 的形式定义。

一个 Spring Bean 定义 包含什么?

一个Spring Bean 的定义包含容器必知的所有配置元数据，包括bean，它的生命周期详情及它的依赖。

如何创建一个如何给Spring 容器提供配置元数据？Spring有几种配置方式

这里有三种重要的方法给Spring 容器提供配置元数据。

- XML配置文件。
- 基于注解的配置。
- 基于java的配置。

Spring配置文件包含了哪些信息

Spring配置文件是个XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

Spring基于xml注入bean的几种方式

1. Set方法注入；
2. 构造器注入：①通过index设置参数的位置；②通过type设置参数类型；
3. 静态工厂注入；
4. 实例工厂；

你怎样定义类的作用域?

当定义一个 在Spring里，我们还能给这个bean声明一个作用域。它可以通过 bean 定义中的scope属性来定义。如，当Spring要在需要的时候每次生产一个新的bean实例，bean的scope属性被指定为prototype。另一方面，一个bean 每次使用的时候必须返回同一个实例，这个bean的scope 属性 必须设为 singleton。

解释Spring支持的几种bean的作用域

Spring框架支持以下五种bean的作用域：

- singleton : bean在每个Spring ioc 容器中只有一个实例。
- prototype：一个bean的定义可以有多个实例。

- request：每次http请求都会创建一个bean，该作用域仅在基于web的SpringApplicationContext情形下有效。
- session：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
- global-session：在一个全局的HTTP Session中，一个bean定义对应一个

实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

注意： 缺省的Spring bean 的作用域是Singleton。使用 prototype 作用域需要慎重的思考，因为频繁创建和销毁 bean 会带来很大的性能开销。

Spring框架中的单例bean是线程安全的吗?

不是，Spring框架中的单例bean不是线程安全的。

spring 中的 bean 默认是单例模式，spring 框架并没有对单例 bean 进行多线程的封装处理。

实际上大部分时候 spring bean 无状态的（比如 dao 类），所有某种程度上来说 bean 也是安全的，但如果 bean 有状态的话（比如 view model 对象），那就要开发者自己去保证线程安全了，最简单的就是改变 bean 的作用域，

把“singleton”变更为“prototype”，这样请求 bean 相当于 new Bean() 了，所以就可以保证线程安全了。

- 有状态就是有数据存储功能。
- 无状态就是不会保存数据。

Spring如何处理线程并发问题?

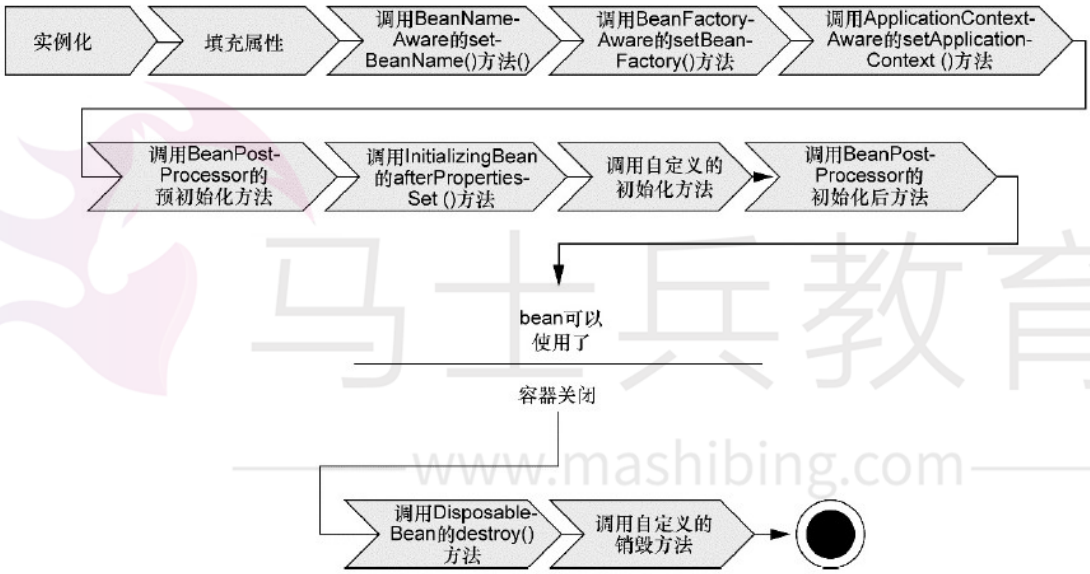
在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前

需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。解释Spring框架中bean的生命周期

在传统的Java应用中，bean的生命周期很简单。使用Java关键字new进行bean 实例化，然后该bean就可以使用了。一旦该bean不再被使用，则由Java自动进行垃圾回收。相比之下，Spring容器中的bean的生命周期就显得相对复杂多了。正确理解Spring bean的生命周期非常重要，因为你或许要利用Spring提供的扩展点来自定义bean的创建过程。下图展示了bean装载到Spring应用上下文中的一个典型的生命周期过程。



马士兵教育
www.mashibing.com

bean在Spring容器中从创建到销毁经历了若干阶段，每一阶段都可以针对 Spring如何管理bean进行个性化定制。

正如你所见，在bean准备就绪之前，bean工厂执行了若干启动步骤。

我们对上图进行详细描述：

Spring对bean进行实例化；

Spring将值和bean的引用注入到bean对应的属性中；

如果bean实现了BeanNameAware接口，Spring将bean的ID传递给setBeanName()方法；

如果bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()方

法，将BeanFactory容器实例传入；

如果bean实现了ApplicationContextAware接口，Spring将调用 setApplicationContext()方法，将bean所在的应用上下文的引用传入进来；

如果bean实现了BeanPostProcessor接口，Spring将调用它们的post-

ProcessBeforeInitialization()方法；

如果bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet()方法。类似地，如果bean使用initmethod声明了初始化方法，

该方法也会被调用；

如果bean实现了BeanPostProcessor接口，Spring将调用它们的post-

ProcessAfterInitialization()方法；此时，bean已经准备就绪，可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁；

如果bean实现了DisposableBean接口，Spring将调用它的destroy()接口方

法。同样，如果bean使用destroy-method声明了销毁方法，该方法也会被调用。

现在你已经了解了如何创建和加载一个Spring容器。但是一个空的容器并没有太大的价值，在你把东西放进去之前，它里面什么都没有。为了从Spring的

DI(依赖注入)中受益，我们必须将应用对象装配进Spring容器中。

哪些是重要的bean生命周期方法？ 你能重载它们吗？

有两个重要的bean 生命周期方法，第一个是setup ， 它是在容器加载bean的

时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

bean 标签有两个重要的属性（init-method和destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct和@PreDestroy）。

什么是Spring的内部bean？什么是Spring inner beans?

在Spring框架中，当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean。内部bean可以用setter注入“属性”和构造方法注入“构造参数”的方式来实现，内部bean通常是匿名的，它们的Scope一般是prototype。

在 Spring中如何注入一个java集合?

Spring提供以下几种集合的配置元素：类型用于注入一列值，允许有相同的值。

类型用于注入一组值，不允许有相同的值。

类型用于注入一组键值对，键和值都只能为String类型。

什么是bean装配?

装配，或bean 装配是指在Spring 容器中把bean组装到一起，前提是容器需要知道bean的依赖关系，如何通过依赖注入来把它们装配到一起。

什么是bean的自动装配?

在Spring框架中，在配置文件中设定bean的依赖关系是一个很好的机制，

Spring 容器能够自动装配相互合作的bean，这意味着容器不需要和配置，能通

过Bean工厂自动处理bean之间的协作。这意味着 Spring可以通过向Bean

Factory中注入的方式自动搞定bean之间的依赖关系。自动装配可以设置在每个bean上，也可以设定在特定的bean上。

解释不同方式的自动装配，spring 自动装配 bean 有哪些方式?

在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。

在Spring框架xml配置中共有5种自动装配：

- no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。
- byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。
- byType：通过参数的数据类型进行自动装配。
- constructor：利用构造函数进行装配，并且构造函数的参数通过 byType进行装配。
- autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

使用@Autowired注解自动装配的过程是怎样的?

使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，。

在启动spring IoC时，容器自动装载了一个

AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的 bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

- 如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；
- 如果查询的结果不止一个，那么@Autowired会根据名称来查找；
- 如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

自动装配有哪些局限性?

自动装配的局限性是：重写：你仍需用 和 配置来定义依赖，意味着总要重写自动装配。

基本数据类型：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。

模糊特性：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

你可以在Spring中注入一个null 和一个空字符串吗?

可以

Spring注解

什么是基于Java的Spring注解配置? 给一些注解的例子

基于Java的配置，允许你在少量的Java注解的帮助下，进行你的大部分Spring 配置而非通过XML文件。

以@Configuration 注解为例，它用来标记类可以当做一个bean的定义，被

Spring IOC容器使用。

另一个例子是@Bean注解，它表示此方法将要返回一个对象，作为一个bean注册进Spring应用上下文。

```
1   @Configuration
2   public class StudentConfig {
3   @Bean
4   public StudentBean myStudent() {
5   return new StudentBean();
6   }
7   }
```

怎样开启注解装配?

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在Spring配置文件中配置 元素。

@Component, @Controller, @Repository, @Service 有何区别?

@Component：这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

@Controller：这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

@Service：此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

@Repository：这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

@Required 注解有什么作用

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出

BeanInitializationException。示例：

```
1   public class Employee {
2   private String name;
3   @Required
4   public void setName(String name){
5   this.name=name;
6   }
7   public string getName(){
8   return name;
9   }
10  }
```

@Autowired 注解有什么作用

@Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰 setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

```
1   public class Employee {
2   private String name;
3   @Autowired
4   public void setName(String name) {
5   this.name=name;
6   }
7   public string getName(){
8   return name;
9   }
10  }
```

@Autowired和@Resource之间的区别

@Autowired可用于：构造函数、成员变量、Setter方法

@Autowired和@Resource之间的区别

@Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。

@Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

@Qualifier 注解有什么作用

当您创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，您可以使用@Qualifier 注解和 @Autowired 通过指定应该装配哪个确切的 bean 来消除歧义。

@RequestMapping 注解有什么用?

@RequestMapping 注解用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。此注释可应用于两个级别：

- 类级别：映射请求的 URL
- 方法级别：映射 URL 以及 HTTP 请求方法

Spring数据访问

解释对象/关系映射集成模块

Spring 通过提供ORM模块，支持我们在直接JDBC之上使用一个对象/关系映射映射(ORM)工具，Spring支持集成主流的ORM框架，如Hiberate，JDO和 iBATIS，JPA，TopLink，JDO，OJB 。Spring的事务管理同样支持以上所有ORM框架及JDBC。

在Spring框架中如何更有效地使用JDBC?

使用Spring JDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者

只需写statements 和 queries从数据存取数据，JDBC也可以在Spring框架提供的模板类的帮助下更有效地被使用，这个模板叫JdbcTemplate

解释JDBC抽象和DAO模块

通过使用JDBC抽象和DAO模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用Spring的AOP 模块给Spring应用中的对象提供事务管理服务。

spring DAO 有什么用?

Spring DAO（数据访问对象） 使得 JDBC，Hibernate 或 JDO 这样的数据访

问技术更容易以一种统一的方式工作。这使得用户容易在持久性技术之间切换。

它还允许您在编写代码时，无需考虑捕获每种技术不同的异常。

spring JDBC API 中存在哪些类?

JdbcTemplate

SimpleJdbcTemplate

NamedParameterJdbcTemplate

SimpleJdbcInsert

SimpleJdbcCall

JdbcTemplate是什么

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

使用Spring通过什么方式访问Hibernate？使用 Spring 访问 Hibernate 的方法有哪些?

在Spring中有两种方式访问Hibernate：

- 使用 Hibernate 模板和回调进行控制反转
- ernateDAOSupport 并应用 AOP 拦截器节点

如何通扩展 Hib过HibernateDaoSupport将Spring和Hibernate 结合起来?

用Spring的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

- 配置the Hibernate SessionFactory
- 继承HibernateDaoSupport实现一个DAO
- 在AOP支持的事务中装配

Spring支持的事务管理类型， spring 事务实现方式有哪些?

Spring支持两种类型的事务管理：编程式事务管理：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。

声明式事务管理：这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

Spring事务的实现方式和实现原理

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，

spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过 binlog或者redo log实现的。

说一下Spring的事务传播行为

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

① PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

③ PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

说一下 spring 的事务隔离?

spring 有五大隔离级别，默认值为 ISOLATION_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致：

1. ISOLATION_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；
2. ISOLATION_READ_UNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）； 3. ISOLATION_READ_COMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；
3. ISOLATION_REPEATABLE_READ：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；
4. ISOLATION_SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

脏读： 表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

不可重复读： 是指在一个事务内，多次读同一数据。

幻读： 指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

Spring框架的事务管理有哪些优点?

- 为不同的事务API 如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。
- 为编程式事务管理提供了一套简单的API而不是一些复杂的事务API 支持声明式事务管理。
- 和Spring各种数据访问抽象层很好得集成。

你更倾向用那种事务管理类型?

大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

Spring面向切面编程(AOP)什么是AOP

OOP(Object-Oriented Programming)面向对象编程，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP(Aspect-Oriented Programming)，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理等。

Spring AOP and AspectJ AOP 有什么区别？AOP 有哪些实现方式?

AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1） AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2） Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP

对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

JDK动态代理和CGLIB动态代理的区别

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

- JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。
- 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ 的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而

Spring AOP则无需特定的编译器处理。

InvocationHandler 的 invoke(Object proxy,Method method,Object[] args):

proxy是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

如何理解 Spring 中的代理?

将 Advice 应用于目标对象后创建的对象称为代理。在客户端对象的情况下，目标对象和代理对象是相同的。

Advice + Target Object = Proxy

解释一下Spring AOP里面的几个名词

（1） 切面（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。

（2） 连接点（Join point）：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。 应用可能有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

（3） 通知（Advice）：在AOP术语中，切面的工作被称为通知。

（4） 切入点（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

（5） 引入（Introduction）：引入允许我们向现有类添加新方法或属性。

（6） 目标对象（Target Object）： 被一个或者多个切面（aspect）所通知

（advise）的对象。它通常是一个代理对象。也有人把它叫做 被通知

（adviced） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7） 织入（Weaving）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有多少个点可以进行织入：

- 编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。
- 类加载期：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP 容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

Spring在运行时通知对象

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean

中。代理封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标 bean。当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。

直到应用需要被代理的bean时，Spring才创建代理对象。如果使用的是

ApplicationContext的话，在ApplicationContext从BeanFactory中加载所有 bean的时候，Spring才会创建被代理的对象。因为Spring运行时才创建代理对象，所以我们不需要特殊的编译器来织入SpringAOP的切面。

Spring只支持方法级别的连接点

因为Spring基于动态代理，所以Spring只支持方法连接点。Spring缺少对字段可以利用Aspect来补充。

连接点的支持，而且它不支持构造器连接点。方法之外的连接点拦截功能，我们在Spring AOP 中，关注点和横切关注的区别是什么？在 spring aop 中 concern 和 cross-cutting concern 的不同之处

关注点（concern）是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点（cross-cutting concern）是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

Spring通知有哪些类型?

在AOP术语中，切面的工作被称为通知，实际上是程序执行时要通过

SpringAOP框架触发的代码段。

Spring切面可以应用5种类型的通知：

1. 前置通知（Before）：在目标方法被调用之前调用通知功能；
2. 后置通知（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
3. 返回通知（After-returning ）：在目标方法成功执行之后调用通知；
4. 异常通知（After-throwing）：在目标方法抛出异常后调用通知；
5. 环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。

同一个aspect，不同advice的执行顺序：

①没有异常情况下的执行顺序：

around before advice before advice target method 执行 around after advice after advice

afterReturning

②有异常情况下的执行顺序： around before advice before advice target method 执行 around after advice after advice

afterThrowing:异常发生 java.lang.RuntimeException: 异常发生

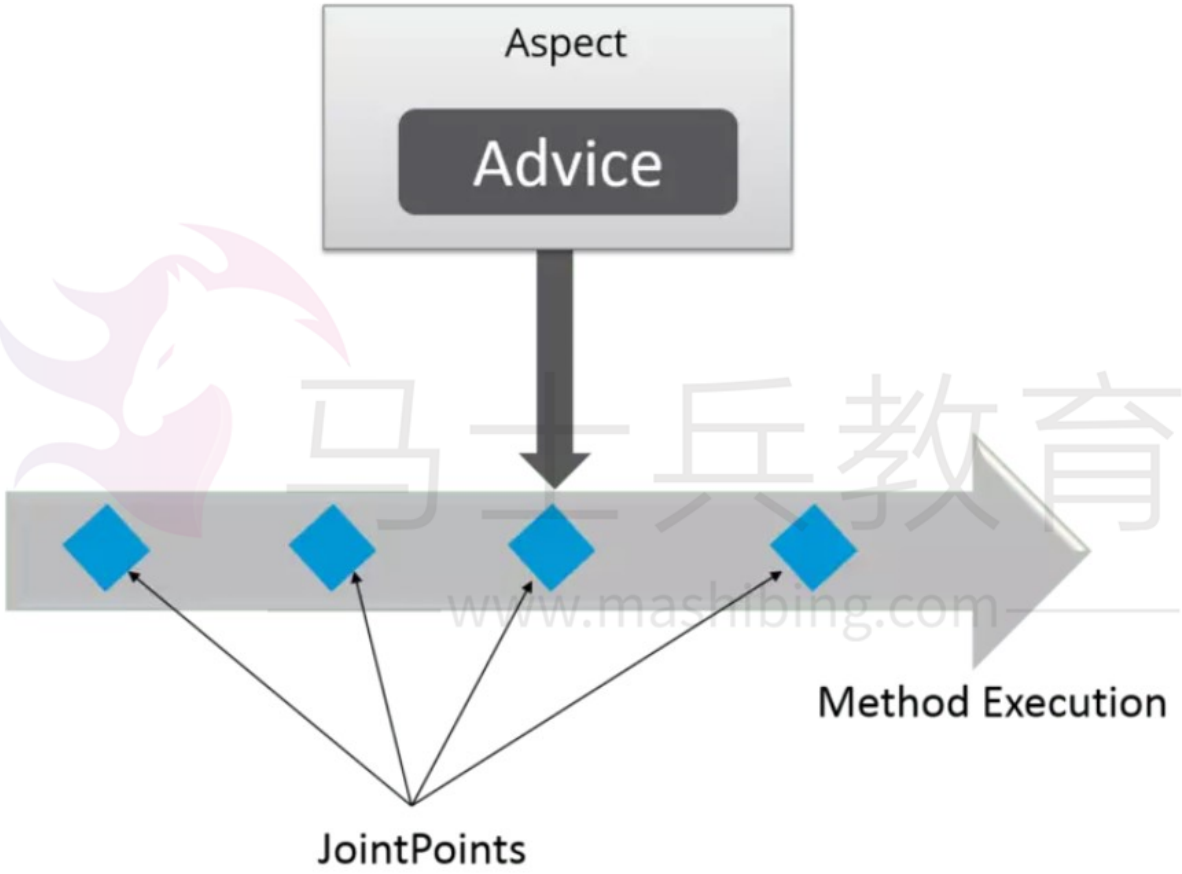
什么是切面 Aspect?

aspect 由 pointcount 和 advice 组成，切面是通知和切点的结合。 它既包含了横切逻辑的定义, 也包括了连接点的定义. Spring AOP 就是负责实施切面的框架, 它将切面所定义的横切逻辑编织到切面所指定的连接点中.

AOP 的工作重心在于如何将增强编织目标对象的连接点上, 这里包含两个工作:

- 如何通过 pointcut 和 advice 定位到特定的 joinpoint 上
- 如何在 advice 中编写切面代码.

可以简单地认为, 使用 @Aspect 注解的类就是切面.



解释基于XML Schema方式的切面实现

在这种情况下，切面由常规类以及基于XML的配置实现。解释基于注解的切面实现

在这种情况下(基于@AspectJ的实现)，涉及到的切面声明的风格与带有java5标注的普通java类一致。

有几种不同类型的自动代理?

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

SpringMVC面试题

什么是 SpringMvc?

答：SpringMvc 是 spring 的一个模块，基于 MVC 的一个框架，无需中间整合层来整合。

Spring MVC 的优点：答:

1）它是基于组件技术的.全部的应用对象,无论控制器和视图,还是业务对象之类的都是 java 组件.并且和 Spring 提供的其他基础结构紧密集成.

2）不依赖于 Servlet API(目标虽是如此,但是在实现的时候确实是依赖于 Servlet 的)

3）可以任意使用各种视图技术,而不仅仅局限于 JSP

4）支持各种请求资源的映射策略

5）它应是易于扩展的

SpringMVC 工作原理？答:

1）客户端发送请求到 DispatcherServlet

2）DispatcherServlet 查询 handlerMapping 找到处理请求的 Controller

3）Controller 调用业务逻辑后，返回 ModelAndView

4）DispatcherServlet 查询 ModelAndView，找到指定视图

5）视图将结果返回到客户端

SpringMVC 流程？答:

1） 用户发送请求至前端控制器 DispatcherServlet。

2） DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3） 处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4） DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5） HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

6） Controller 执行完成返回 ModelAndView。

7） HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

8） DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9） ViewReslover 解析后返回具体 View。

10）DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11）DispatcherServlet 响应用户。

SpringMvc 的控制器是不是单例模式,如果是,有什么问题,怎么解决?

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

如果你也用过 struts2.简单介绍下 springMVC 和 struts2 的区别有哪些?

答：

1）springmvc 的入口是一个 servlet 即前端控制器，而 struts2 入口是一个 filter 过虑器。

2）springmvc 是基于方法开发(一个 url 对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2 是基于类开发，传递参数是通过类的属性，只能设计为多例。

3）Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据，springmvc 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 reques 域传输到页面。Jsp 视图解析器默认使用 jstl。

SpingMvc 中的控制器的注解一般用那个,有没有别的注解可以替代?

答：一般用@Conntroller 注解,表示是表现层,不能用用别的注解代替。

@RequestMapping 注解用在类上面有什么作用?

答：是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

怎么样把某个请求映射到特定的方法上面?

答：直接在方法上面加上注解@RequestMapping,并且在这个注解里面写上要拦截的路径

如果在拦截请求中,我想拦截 get 方式提交的方法,怎么配置?

答：可以在@RequestMapping 注解里面加上 method=RequestMethod.GET

怎么样在方法里面得到 Request,或者 Session?

答：直接在方法的形参中声明 request,SpringMvc 就自动把 request 对象传入

我想在拦截的方法里面得到从前台传入的参数,怎么得到?

答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样

如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?

答：直接在方法中声明这个对象,SpringMvc 就自动会把属性赋值到这个对象里面。

SpringMvc 中函数的返回值是什么?

答：返回值可以有很多类型,有 String, ModelAndView,当一般用 String 比较好。

SpringMVC 怎么样设定重定向和转发的?

答：在返回值前面加"forward:"就可以让结果转发,譬如"forward:user.do?name=method4" 在返回值前面加"redirect:"就可以让返回值重定向,譬如"redirect:http://www.baidu.com"

SpringMvc 用什么对象从后台向前台传递数据的?

答：通过 ModelMap 对象,可以在这个对象里面用 put 方法,把对象加到里面,前台就可以通过 el 表达式拿到。

SpringMvc 中有个类把视图和数据都合并的一起的,叫什么?

答：叫 ModelAndView。

怎么样把 ModelMap 里面的数据放入 Session 里面?

答：可以在类上面加上@SessionAttributes 注解,里面包含的字符串就是要放入 session 里面的 key

SpringMvc 怎么和 AJAX 相互调用的? 答:

通过 Jackson 框架就可以把 Java 里面的对象直接转化成 Js 可以识别的 Json 对象。

具体步骤如下：

1）加入 Jackson.jar

2）在配置文件中配置 json 的映射

3）在接受 Ajax 方法里面可以直接返回 Object,List 等,但方法前面要加上@ResponseBody 注解

21、当一个方法向 AJAX 返回特殊对象,譬如 Object,List 等,需要做什么处理? 答：要加上 @ResponseBody 注解 22、SpringMvc 里面拦截器是怎么写的

答：有两种写法,一种是实现接口,另外一种是继承适配器类,然后在 SpringMvc 的配置文件中配置拦截器即可:

```
<!-- 配置 SpringMvc 的拦截器 -->

<mvc:interceptors>

   <!-- 配置一个拦截器的 Bean 就可以了 默认是对所有请求都拦截 -->

   <bean id="myInterceptor" class="com.et.action.MyHandlerInterceptor"></bean>


   <!-- 只针对部分请求拦截 -->

   <mvc:interceptor>

     <mvc:mapping path="/modelMap.do" />

     <bean class="com.et.action.MyHandlerInterceptorAdapter" />
```

```
    </mvc:interceptor>
</mvc:interceptors>
```

讲下 SpringMvc 的执行流程

答：系统启动的时候根据配置文件创建 spring 的容器, 首先是发送 http 请求到核心控制器

disPatherServlet，spring 容器通过映射器去寻找业务控制器，使用适配器找到相应的业务类，在进业务类时进行数据封装，在封装前可能会涉及到类型转换，执行完业务类后使用

ModelAndView 进行视图转发，数据放在 model 中，用 map 传递数据进行页面显示。

Spring Boot（连老师）

Spring Boot概述

什么是 Spring Boot?

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

Spring Boot 有哪些优点?

Spring Boot 主要有如下优点：

1. 容易上手，提升开发效率，为 Spring 开发提供一个更快、更广泛的入门体验。
2. 开箱即用，远离繁琐的配置。
3. 提供了一系列大型项目通用的非业务性功能，例如：内嵌服务器、安全管理、运行数据监控、运行状况检查和外部化配置等。
4. 没有代码生成，也不需要XML配置。
5. 避免大量的 Maven 导入和各种版本冲突。

Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的?

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude

= { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

配置

什么是 JavaConfig?

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

（1） 面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以

充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的

@Bean 方法等。

（2） 减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。

但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从

技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将JavaConfig 与 XML 混合匹配是理想的。（3）类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置

Spring容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

Spring Boot 自动配置原理是什么?

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，

@EnableAutoConfiguration 给容器导入META-INF/spring.factories 里定义的自动配置类。

筛选有效的自动配置类。

每一个自动配置类结合对应的 xxxProperties.java 读取配置文件进行自动配置功能

你如何理解 Spring Boot 配置加载顺序?

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

1） properties文件；

2） YAML文件；

3） 系统环境变量；

等等……

4）命令行参数； 什么是 YAML?

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

YAML 配置的优势在哪里？

YAML 现在可以算是非常流行的一种配置文件格式了，无论是前端还是后端，都可以见到 YAML 配置。那么 YAML 配置和传统的 properties 配置相比到底有哪些优势呢?

1. 配置有序，在一些特殊的场景下，配置有序很关键
2. 支持数组，数组中的元素可以是基本数据类型也可以是对象
3. 简洁

相比 properties 配置文件，YAML 还有一个缺点，就是不支持 @PropertySource 注解导入自定义的 YAML 配置。

Spring Boot 是否可以使用 XML 配置？

Spring Boot 推荐使用 Java 配置而非 XML 配置，但是 Spring Boot 中也可以使用 XML 配置，通过 @ImportResource 注解可以引入一个 XML 配置。 spring boot 核心配置文件是什么?

bootstrap.properties 和 application.properties 有何区别？

单纯做 Spring Boot 开发，可能不太容易遇到 bootstrap.properties 配置文

件，但是在结合 Spring Cloud 时，这个配置就会经常遇到了，特别是在需要加载一些远程配置文件的时侯。

spring boot 核心的两个配置文件:

bootstrap (. yml 或者 . properties)：boostrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud Config 或者 Nacos 中会用到它。且 boostrap 里面的属性不

能被覆盖;

application (. yml 或者 . properties)： 由ApplicatonContext 加载，用于 spring boot 项目的自动化配置。

什么是 Spring Profiles?

Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在

PRODUCTION中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

如何在自定义端口上运行 Spring Boot 应用程序?

为了在自定义端口上运行 Spring Boot 应用程序，您可以在

application.properties 中指定端口。server.port = 8090

安全

如何实现 Spring Boot 应用程序的安全性?

为了实现 Spring Boot 的安全性，我们使用 spring-boot-starter-security 依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展

WebSecurityConfigurerAdapter 并覆盖其方法。

比较一下 Spring Security 和 Shiro 各自的优缺点？

由于 Spring Boot 官方提供了大量的非常方便的开箱即用的 Starter ，包括

Spring Security 的 Starter ，使得在 Spring Boot 中使用 Spring Security 变得更加容易，甚至只需要添加一个依赖就可以保护所有的接口，所以，如果是

Spring Boot 项目，一般选择 Spring Security 。当然这只是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。Shiro 和 Spring

Security 相比，主要有如下一些特点：

1. Spring Security 是一个重量级的安全管理框架；Shiro 则是一个轻量级

的安全管理框架

2. Spring Security 概念复杂，配置繁琐；Shiro 概念简单、配置简单
3. Spring Security 功能强大；Shiro 功能简单

Spring Boot 中如何解决跨域问题？

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过 （CORS，Cross-origin resource sharing） 来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS ，现在可以通过实现 WebMvcConfigurer接口然后重写addCorsMappings方法解决跨域问题。

```
1  @Configuration
2  public class CorsConfig implements WebMvcConfigurer { 3
4   @Override
5   public void addCorsMappings(CorsRegistry registry) {
6   registry.addMapping("/**")
7   .allowedOrigins("*")
8   .allowCredentials(true)
9   .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
10   .maxAge(3600);
11   }
12
13  }
```

项目中前后端分离部署，所以需要解决跨域的问题。

我们使用cookie存放用户登录的信息，在spring拦截器进行权限控制，当权限不符合时，直接返回给用户固定的json结果。

当用户登录以后，正常使用；当用户退出登录状态时或者token过期时，由于拦截器和跨域的顺序有问题，出现了跨域的现象。

我们知道一个http请求，先走filter，到达servlet后才进行拦截器的处理，如果我们把cors放在filter里，就可以优先于权限拦截器执行。

```
1  @Configuration
2  public class CorsConfig { 3
4   @Bean
5   public CorsFilter corsFilter() {
6   CorsConfiguration corsConfiguration = new CorsConfiguration();
7   corsConfiguration.addAllowedOrigin("*");
8   corsConfiguration.addAllowedHeader("*");
```

```
9   corsConfiguration.addAllowedMethod("*");
10   corsConfiguration.setAllowCredentials(true);
11   UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new U
rlBasedCorsConfigurationSource();
12   urlBasedCorsConfigurationSource.registerCorsConfiguration("/**", corsCo
nfiguration);
13   return new CorsFilter(urlBasedCorsConfigurationSource);
14   }
15
16  }
```

什么是 CSRF 攻击?

CSRF 代表跨站请求伪造。这是一种攻击，迫使 终用户在当前通过身份验证的 Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

监视器

Spring Boot 中的监视器是什么?

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问 的REST 端点来检查状态。

如何在 Spring Boot 中禁用 Actuator 端点安全性?

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色

的用户才能访问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。 我们可以使用来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

我们如何监视所有 Spring Boot 微服务?

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。为了帮助我们处理这种情况，我们将使用位于的开源项目。 它建立在 Spring Boot Actuator 之上，它提供了一个 Web UI，使我们能够可视化多个应用程序的度量。

整合第三方项目

什么是 WebSockets?

WebSocket 是一种计算机通信协议，通过单个 TCP 连接提供全双工通信信道。

1、 WebSocket 是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。

2、 WebSocket 是全双工的 -客户端和服务器通信是相互独立的。

3、 单个 TCP 连接 -初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信

4、 Light -与 http 相比，WebSocket 消息数据交换要轻得多。

什么是 Spring Data ?

Spring Data 是 Spring 的一个子项目。用于简化数据库访问，支持NoSQL 和 关系数据存储。其主要目标是使数据库的访问变得方便快捷。Spring Data 具有如下特点：

SpringData 项目支持 NoSQL 存储：

1. MongoDB （文档数据库）
2. Neo4j（图形数据库）
3. Redis（键/值存储）
4. Hbase（列族数据库）

SpringData 项目所支持的关系数据存储技术：

1. JDBC
2. JPA

Spring Data Jpa 致力于减少数据访问层 (DAO) 的开发量. 开发者唯一要做的，就是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成！Spring Data JPA 通过规范方法的名字，根据符合规范的名字来确定方法需要实现什么样的逻辑。

什么是 Spring Batch?

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

什么是 FreeMarker 模板?

FreeMarker 是一个基于 Java 的模板引擎， 初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码，而设计人员可以处理 html 页面设计。 后使用 freemarker 可以将这些结合起来，给出 终的输出页面。

如何集成 Spring Boot 和 ActiveMQ?

对于集成 Spring Boot 和 ActiveMQ，我们使用依赖关系。 它只需要很少的配置，并且不需要样板代码。

什么是 Apache Kafka?

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的，容错的发布 - 订阅消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离线和在线消息消费。

什么是 Swagger？你用 Spring Boot 实现了它吗?

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用 少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger消除了调用服务时的猜测。

前后端分离，如何维护接口文档？

前后端分离开发日益流行，大部分情况下，我们都是通过 Spring Boot 做前后端分离开发，前后端分离一定会有接口文档，不然会前后端会深深陷入到扯皮中。一个比较笨的方法就是使用 word 或者 md 来维护接口文档，但是效率太低，接口一变，所有人手上的文档都得变。在 Spring Boot 中，这个问题常见

的解决方案是 Swagger ，使用 Swagger 我们可以快速生成一个接口文档网

站，接口一旦发生变化，文档就会自动更新，所有开发工程师访问这一个在线网站就可以获取到 新的接口文档，非常方便。

其他

如何重新加载 Spring Boot 上的更改，而无需重新启动服务器? Spring Boot项目如何热部署?

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式tomcat 将重新启动。Spring Boot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。

Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员 需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。

它还提供 H2 数据库控制台以更好地测试应用程序。

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-devtools</artifactId>
4  </dependency>
```

您使用了哪些 starter maven 依赖项?

使用了下面的一些依赖项

spring-boot-starter-activemq

spring-boot-starter-security

这有助于增加更少的依赖关系，并减少版本的冲突。

Spring Boot 中的 starter 到底是什么？

首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 XXXAutoConfiguration

，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是

Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。当然，开发者也可以自定义 Starter
spring-boot-starter-parent 有什么用？

我们都知道，新创建一个 Spring Boot 项目，默认都是有 parent 的，这个

parent 就是 spring-boot-starter-parent ，spring-boot-starter-parent 主要有如下作用：

1. 定义了 Java 编译版本为 1.8 。
2. 使用 UTF-8 格式编码。
3. 继承自 spring-boot-dependencies，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。
4. 执行打包操作的配置。
5. 自动化的资源过滤。
6. 自动化的插件配置。

7. 针对 application.properties 和 application.yml 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 applicationdev.properties 和 application-dev.yml。

Spring Boot 打成的 jar 和普通的 jar 有什么区别？

Spring Boot 项目 终打包成的 jar 是可执行 jar，这种 jar 可以直接通过 java jar xxx.jar 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

Spring Boot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot

打包成的可执行 jar 解压后，在 \BOOT-INF\classes 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar，一个可执行，一个可引用。

运行 Spring Boot 有哪几种方式?

1） 打包用命令或者放到容器中运行

2） 用 Maven/ Gradle 插件运行

3）直接执行 main 方法运行

Spring Boot 需要独立的容器运行吗?

开启 Spring Boot 特性有哪几种方式?

1） 继承spring-boot-starter-parent项目

2） 导入spring-boot-dependencies项目依赖

如何使用 Spring Boot 实现异常处理?

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。 我们通过实现一个 ControlerAdvice 类，来处理控制器类抛出的所有异常。

如何使用 Spring Boot 实现分页和排序?

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的传递给存储库方法。

微服务中如何实现 session 共享？

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经

常，我们需要在不同微服务之间共享 session，常见的方案就是 Spring

Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

Spring Boot 中如何实现定时任务？

定时任务也是一个常见的需求，Spring Boot 中对于定时任务的支持主要还是来自 Spring 框架。

在 Spring Boot 中使用定时任务主要有两种不同的方式，一个就是使用 Spring 中的 @Scheduled 注解，另一个则是使用第三方框架 Quartz。

使用 Spring 中的 @Scheduled 的方式主要通过 @Scheduled 注解来实现。

使用 Quartz，则按照 Quartz 的方式，定义 Job 和 Trigger 即可。

Spring Boot 原理

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot 致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。其特点如下：

1. 创建独立的 **Spring** 应用程序

2. 嵌入的 **Tomcat**，无需部署 **WAR** 文件

3. 简化 **Maven** 配置

4. 自动配置 **Spring**

5. 提供生产就绪型功能，如指标，健康检查和外部配置

6. 绝对没有代码生成和对 **XML** 没有要求配置 **[1]**

Spring Boot比Spring做了哪些改进?

1）Spring Boot可以建立独立的Spring应用程序；

2）内嵌了如Tomcat，Jetty和Undertow这样的容器，也就是说可以直接跑起来，用不着再做 部署工作了；

3）无需再像Spring那样搞一堆繁琐的xml文件的配置；

4）可以自动配置Spring。SpringBoot将原有的XML配置改为Java配置，将bean注入改为使 用注解注入的方式(@Autowire)，并将多个xml、properties配置浓缩在一个appliaction.yml 配置文件中。

5）提供了一些现有的功能，如量度工具，表单数据验证以及一些外部配置这样的一些第三方功 能；

6）整合常用依赖（开发库，例如spring-webmvc、jackson-json、validation-api和tomcat 等），提供的POM可以简化Maven的配置。当我们引入核心依赖时，SpringBoot会自引入其 他依赖。

Spring boot 热加载

在实际的开发中避免不了自己测试的时候修修改改，甚至有些源代码的修改是需要重启项目的，这个时候热加载就帮了大忙了，其会自动将修改的代码应用到部署的项目中去，而不用自己再次的去手动重启，大大的提高了我们开发的效率，实现了代码随时改效果立马生效的效果，好了废话不多说了，下面来介绍怎解嵌入热加载的实现。

在pom文件中添加依赖（optional-->true表示覆盖父级项目中的引用）：

```
 org.springframework.boot </ groupId>
 spring-boot-devtools </ artifactId>
 true </ optional>
</ dependency>
```

就是这么简单，这样就可以了，当然了，有些时候我们再修改一些文件时是并不希望其触发重启的，例如一些静态资源等，此时我们可以设置一些排序路径来将其排除出去，在此之前先来介绍一下触发重启的条件吧：当DevTools监视类路径资源时，触发重新启动的唯一方法是更新类路径。导致类路径更新的方式取决于您正在使用的IDE。在Eclipse中，保存修改的文件将导致类路径被更新并触发重新启动。在IntelliJ IDEA中，构建project（Build -> Make Project）将具有相同的效果。

如果要自定义一些排除项，您可以使用该spring.devtools.restart.exclude属性。例如，仅排除 /static和/public你设置如下：

```
spring.devtools.restart.exclude=static/,public/
```

当您对不在类路径中的文件进行更改时，可能需要重新启动或重新加载应用程序。为此，请使用该spring.devtools.restart.additional-paths属性配置其他路径来监视更改。

如果您不想使用重新启动功能，可以使用该spring.devtools.restart.enabled属性禁用它 。在大多数情况下，您可以将其设置为 application.properties（仍将初始化重新启动类加载器，但不会监视文件更改）。

例如，如果您需要完全禁用重新启动支持，因为它不适用于特定库，则需要System在调用之前设置属性 SpringApplication.run(...)。例如：

```
public static void main(String[] args) {
  System.setProperty("spring.devtools.restart.enabled", "false");
  SpringApplication.run(MyApp.class, args);
}
```

我不知道大家都使用的是什么IDE，我现在用的是IntelliJ IDEA，而这个工具有各特点就是编辑文件后其会立刻自动保存，并不需要手动的Ctrl+s来操作，造成的结果就是每当我修改一个触发启动的文件的时候其就会自动进行热加载，而我们并不想这样频繁的去热加载的话，可以进行一些特殊的设计实现仅在特定的时间去触发热加载：我们可以使用“触发文件”，这是一个特殊文件，当您要实际触发重新启动检查时，必须修改它。更改文件只会触发检查，只有在Devtools检测到它必须执行某些操作时才会重新启动。触发文件可以手动更新，也可以通过IDE插件更新。要使用触发器文件使用该spring.devtools.restart.trigger-file属性。

Spring Boot设置有效时间和自动刷新缓存，时间支持在配置文件中配置

Spring Boot缓存实战 Redis 设置有效时间和自动刷新缓存，时间支持在配置文件中配置

问题描述

Spring Cache提供的@Cacheable注解不支持配置过期时间，还有缓存的自动刷新。
我们可以通过配置CacheManneg来配置默认的过期时间和针对每个缓存容器（value）单独配置过期时间，但是总是感觉不太灵活。下面是一个示例:

```
@Bean



public CacheManager cacheManager(RedisTemplate redisTemplate) {
    RedisCacheManager cacheManager= new RedisCacheManager(redisTemplate);
    cacheManager.setDefaultExpiration(60);
    Map<String,Long> expiresMap=new HashMap<>();
    expiresMap.put("Product",5L);
    cacheManager.setExpires(expiresMap);
    return cacheManager;
}
```

我们想在注解上直接配置过期时间和自动刷新时间，就像这样:

```
@Cacheable(value = "people#120#90", key = "#person.id")
public Person findOne(Person person) {
    Person p = personRepository.findOne(person.getId());
    System.out.println("为id、key为:" + p.getId() + "数据做了缓存");
    return p;
}
```

value属性上用#号隔开，第一个是原始的缓存容器名称，第二个是缓存的有效时间，第三个是缓存的自动刷新时间，单位都是秒。

缓存的有效时间和自动刷新时间支持SpEl表达式，支持在配置文件中配置，如:

```
@Cacheable(value =
"people#${select.cache.timeout:1800}#${select.cache.refresh:600}", key =
"#person.id", sync = true)//3
public Person findOne(Person person) {
    Person p = personRepository.findOne(person.getId());
    System.out.println("为id、key为:" + p.getId() + "数据做了缓存");
    return p;
}
```

解决思路

查看源码你会发现缓存最顶级的接口就是CacheManager和Cache接口。

CacheManager说明

CacheManager功能其实很简单就是管理cache，接口只有两个方法，根据容器名称获取一个Cache。还有就是返回所有的缓存名称。

```
public interface CacheManager {

    /**
     * 根据名称获取一个Cache（在实现类里面是如果有这个Cache就返回，没有就新建一个Cache放到Map容器中）
     * @param name the cache identifier (must not be {@code null})
     * @return the associated cache, or {@code null} if none found
     */
    Cache getCache(String name);

    /**
     * 返回一个缓存名称的集合
     * @return the names of all caches known by the cache manager
     */
    Collection<String> getCacheNames();

}
```

Cache说明

Cache接口主要是操作缓存的。get根据缓存key从缓存服务器获取缓存中的值，put根据缓存key将数据放到缓存服务器，evict根据key删除缓存中的数据。

```
public interface Cache {

    ValueWrapper get(Object key);

    void put(Object key, Object value);

    void evict(Object key);

    ...
}
```

请求步骤

1. 请求进来，在方法上面扫描@Cacheable注解，那么会触发org.springframework.cache.interceptor.CacheInterceptor缓存的拦截器。
2. 然后会调用CacheManager的getCache方法，获取Cache，如果没有（第一次访问）就新建一Cache并返回。
3. 根据获取到的Cache去调用get方法获取缓存中的值。RedisCache这里有个bug，源码是先判断key是否存在，再去缓存获取值，在高并发下有bug。

代码分析

在最上面我们说了Spring Cache可以通过配置CacheManager来配置过期时间。那么这个过期时间是在哪里用的呢？设置默认的时间setDefaultExpiration，根据特定名称设置有效时间setExpires，获取一个缓存名称（value属性）的有效时间computeExpiration，真正使用有效时间是在createCache方法里面，而这个方法是在父类的getCache方法调用。通过RedisCacheManager源码我们看到：

```
// 设置默认的时间
public void setDefaultExpiration(long defaultExpireTime) {
    this.defaultExpiration = defaultExpireTime;
}

// 根据特定名称设置有效时间
public void setExpires(Map<String, Long> expires) {
    this.expires = (expires != null ? new ConcurrentHashMap<String, Long>
(expires) : null);
}
// 获取一个key的有效时间
protected long computeExpiration(String name) {
    Long expiration = null;
    if (expires != null) {
        expiration = expires.get(name);
    }
    return (expiration != null ? expiration.longValue() : defaultExpiration);
}

@SuppressWarnings("unchecked")
protected RedisCache createCache(String cacheName) {
    // 调用了上面的方法获取缓存名称的有效时间
    long expiration = computeExpiration(cacheName);
    // 创建了Cache对象，并使用了这个有效时间
    return new RedisCache(cacheName, (usePrefix ? cachePrefix.prefix(cacheName)
: null), redisOperations, expiration,
            cacheNullValues);
}

// 重写父类的getMissingCache。去创建Cache
@Override
protected Cache getMissingCache(String name) {
    return this.dynamic ? createCache(name) : null;
}
```

AbstractCacheManager父类源码：

```
// 根据名称获取Cache如果没有调用getMissingCache方法，生成新的Cache，并将其放到Map容器中去。
@Override
public Cache getCache(String name) {
    Cache cache = this.cacheMap.get(name);
    if (cache != null) {
        return cache;
    }
    else {
        // Fully synchronize now for missing cache creation...
        synchronized (this.cacheMap) {
            cache = this.cacheMap.get(name);
            if (cache == null) {
                // 如果没找到Cache调用该方法，这个方法默认返回值NULL由子类自己实现。上面的
就是子类自己实现的方法
                cache = getMissingCache(name);
                if (cache != null) {
                    cache = decorateCache(cache);
                    this.cacheMap.put(name, cache);
                    updateCacheNames(name);
                }
```

```
            }
            return cache;
        }
    }
}
```

由此这个有效时间的设置关键就是在getCache方法上，这里的name参数就是我们注解上的value属性。所以在这里解析这个特定格式的名称我就可以拿到配置的过期时间和刷新时间。getMissingCache方法里面在新建缓存的时候将这个过期时间设置进去，生成的Cache对象操作缓存的时候就会带上我们的配置的过期时间，然后过期就生效了。解析SpEL表达式获取配置文件中的时间也在也一步完成。

CustomizedRedisCacheManager源码：

```
package com.xiaolyuh.redis.cache;

import com.xiaolyuh.redis.cache.helper.SpringContextHolder;
import com.xiaolyuh.redis.utils.ReflectionUtils;
import org.apache.commons.lang3.StringUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.cache.Cache;
import org.springframework.data.redis.cache.RedisCacheManager;
import org.springframework.data.redis.core.RedisOperations;

import java.util.Collection;
import java.util.concurrent.ConcurrentHashMap;

/**
 * 自定义的redis缓存管理器
 * 支持方法上配置过期时间
 * 支持热加载缓存：缓存即将过期时主动刷新缓存
 *
 * @author yuhao.wang
 */
public class CustomizedRedisCacheManager extends RedisCacheManager {

    private static final Logger logger =
LoggerFactory.getLogger(CustomizedRedisCacheManager.class);

    /**
     * 父类cacheMap字段
     */
    private static final String SUPER_FIELD_CACHEMAP = "cacheMap";

    /**
     * 父类dynamic字段
     */
    private static final String SUPER_FIELD_DYNAMIC = "dynamic";
    /**
     * 父类cacheNullValues字段
     */
    private static final String SUPER_FIELD_CACHENULLVALUES = "cacheNullValues";

    /**
     * 父类updateCacheNames方法
```

```
     */
   private static final String SUPER_METHOD_UPDATECACHENAMES =
"updateCacheNames";

    /**
     * 缓存参数的分隔符
     * 数组元素0=缓存的名称
     * 数组元素1=缓存过期时间TTL
     * 数组元素2=缓存在多少秒开始主动失效来强制刷新
     */
    private static final String SEPARATOR = "#";

    /**
     * SpEL标示符
     */
    private static final String MARK = "$";

    RedisCacheManager redisCacheManager = null;

    @Autowired
    DefaultListableBeanFactory beanFactory;

    public CustomizedRedisCacheManager(RedisOperations redisOperations) {
        super(redisOperations);
    }

    public CustomizedRedisCacheManager(RedisOperations redisOperations,
Collection<String> cacheNames) {
        super(redisOperations, cacheNames);
    }

    public RedisCacheManager getInstance() {
        if (redisCacheManager == null) {
            redisCacheManager =
SpringContextHolder.getBean(RedisCacheManager.class);
        }
        return redisCacheManager;
    }

    @Override
    public Cache getCache(String name) {
        String[] cacheParams = name.split(SEPARATOR);
        String cacheName = cacheParams[0];

        if (StringUtils.isBlank(cacheName)) {
            return null;
        }

        // 有效时间，初始化获取默认的有效时间
        Long expirationSecondTime = getExpirationSecondTime(cacheName,
cacheParams);
        // 自动刷新时间，默认是0
        Long preloadSecondTime = getExpirationSecondTime(cacheParams);

        // 通过反射获取父类存放缓存的容器对象
        Object object = ReflectionUtils.getFieldValue(getInstance(),
SUPER_FIELD_CACHEMAP);
        if (object != null && object instanceof ConcurrentHashMap) {
```

```
            ConcurrentHashMap<String, Cache> cacheMap =
(ConcurrentHashMap<String, Cache>) object;
            // 生成Cache对象，并将其保存到父类的Cache容器中
            return getCache(cacheName, expirationSecondTime, preloadSecondTime,
cacheMap);
        } else {
            return super.getCache(cacheName);
        }

    }

    /**
     * 获取过期时间
     *
     * @return
     */
    private long getExpirationSecondTime(String cacheName, String[] cacheParams)
{
        // 有效时间，初始化获取默认的有效时间
        Long expirationSecondTime = this.computeExpiration(cacheName);

        // 设置key有效时间
        if (cacheParams.length > 1) {
            String expirationStr = cacheParams[1];
            if (!StringUtils.isEmpty(expirationStr)) {
                // 支持配置过期时间使用EL表达式读取配置文件时间
                if (expirationStr.contains(MARK)) {
                    expirationStr =
beanFactory.resolveEmbeddedValue(expirationStr);
                }
                expirationSecondTime = Long.parseLong(expirationStr);
            }
        }

        return expirationSecondTime;
    }

    /**
     * 获取自动刷新时间
     *
     * @return
     */
    private long getExpirationSecondTime(String[] cacheParams) {
        // 自动刷新时间，默认是0
        Long preloadSecondTime = 0L;
        // 设置自动刷新时间
        if (cacheParams.length > 2) {
            String preloadStr = cacheParams[2];
            if (!StringUtils.isEmpty(preloadStr)) {
                // 支持配置刷新时间使用EL表达式读取配置文件时间
                if (preloadStr.contains(MARK)) {
                    preloadStr = beanFactory.resolveEmbeddedValue(preloadStr);
                }
                preloadSecondTime = Long.parseLong(preloadStr);
            }
        }
        return preloadSecondTime;
    }
```

```
    /**
     * 重写父类的getCache方法，真假了三个参数
     *
     * @param cacheName             缓存名称
     * @param expirationSecondTime 过期时间
     * @param preloadSecondTime     自动刷新时间
     * @param cacheMap              通过反射获取的父类的cacheMap对象
     * @return Cache
     */
    public Cache getCache(String cacheName, long expirationSecondTime, long
preloadSecondTime, ConcurrentHashMap<String, Cache> cacheMap) {
        Cache cache = cacheMap.get(cacheName);
        if (cache != null) {
            return cache;
        } else {
            // Fully synchronize now for missing cache creation...
            synchronized (cacheMap) {
                cache = cacheMap.get(cacheName);
                if (cache == null) {
                    // 调用我们自己的getMissingCache方法创建自己的cache
                    cache = getMissingCache(cacheName, expirationSecondTime,
preloadSecondTime);
                    if (cache != null) {
                        cache = decorateCache(cache);
                        cacheMap.put(cacheName, cache);


                      // 反射去执行父类的updateCacheNames(cacheName)方法
                        Class<?>[] parameterTypes = {String.class};
                        Object[] parameters = {cacheName};
                        ReflectionUtils.invokeMethod(getInstance(),
SUPER_METHOD_UPDATECACHENAMES, parameterTypes, parameters);
                    }
                }
                return cache;
            }
        }
    }

    /**
     * 创建缓存
     *
     * @param cacheName             缓存名称
     * @param expirationSecondTime 过期时间
     * @param preloadSecondTime     制动刷新时间
     * @return
     */
    public CustomizedRedisCache getMissingCache(String cacheName, long
expirationSecondTime, long preloadSecondTime) {

        logger.info("缓存 cacheName：{}，过期时间:{}，自动刷新时间:{}", cacheName,
expirationSecondTime, preloadSecondTime);
        Boolean dynamic = (Boolean) ReflectionUtils.getFieldValue(getInstance(),
SUPER_FIELD_DYNAMIC);
        Boolean cacheNullValues = (Boolean)
ReflectionUtils.getFieldValue(getInstance(), SUPER_FIELD_CACHENULLVALUES);
```

```
        return dynamic ? new CustomizedRedisCache(cacheName, (this.isUsePrefix()
? this.getCachePrefix().prefix(cacheName) : null),
                this.getRedisOperations(), expirationSecondTime,
preloadSecondTime, cacheNullValues) : null;
    }
}
```

那自动刷新时间呢?

在RedisCache的属性里面没有刷新时间，所以我们继承该类重写我们自己的Cache的时候要多加一个属性preloadSecondTime来存储这个刷新时间。并在getMissingCache方法创建Cache对象的时候指定该值。

CustomizedRedisCache部分源码:

```
/**
 * 缓存主动在失效前强制刷新缓存的时间
 * 单位：秒
 */
private long preloadSecondTime = 0;

// 重写后的构造方法
public CustomizedRedisCache(String name, byte[] prefix, RedisOperations<?
extends Object, ? extends Object> redisOperations, long expiration, long
preloadSecondTime) {
    super(name, prefix, redisOperations, expiration);
    this.redisOperations = redisOperations;
    // 指定自动刷新时间
    this.preloadSecondTime = preloadSecondTime;
    this.prefix = prefix;
}

// 重写后的构造方法
public CustomizedRedisCache(String name, byte[] prefix, RedisOperations<?
extends Object, ? extends Object> redisOperations, long expiration, long
preloadSecondTime, boolean allowNullValues) {
    super(name, prefix, redisOperations, expiration, allowNullValues);
    this.redisOperations = redisOperations;
    // 指定自动刷新时间
    this.preloadSecondTime = preloadSecondTime;
    this.prefix = prefix;
}
```

那么这个自动刷新时间有了，怎么来让他自动刷新呢?

在调用Cache的get方法的时候我们都会去缓存服务查询缓存，这个时候我们在多查一个缓存的有效时间，和我们配置的自动刷新时间对比，如果缓存的有效时间小于这个自动刷新时间我们就去刷新缓存（这里注意一点在高并发下我们最好只放一个请求去刷新数据，尽量减少数据的压力，所以在这个位置加一个分布式锁）。所以我们重写这个get方法。

CustomizedRedisCache部分源码:

```
/**
 * 重写get方法，获取到缓存后再次取缓存剩余的时间，如果时间小余我们配置的刷新时间就手动刷新缓存。
 * 为了不影响get的性能，启用后台线程去完成缓存的刷。
 * 并且只放一个线程去刷新数据。
 *
```

```
     * @param key
     * @return
     */
    @Override
    public ValueWrapper get(final Object key) {
        RedisCacheKey cacheKey = getRedisCacheKey(key);
        String cacheKeyStr = new String(cacheKey.getKeyBytes());
        // 调用重写后的get方法
        ValueWrapper valueWrapper = this.get(cacheKey);

        if (null != valueWrapper) {
            // 刷新缓存数据
            refreshCache(key, cacheKeyStr);
        }
        return valueWrapper;
    }

    /**
     * 重写父类的get函数。
     * 父类的get方法，是先使用exists判断key是否存在，不存在返回null，存在再到redis缓存中去取值。这样会导致并发问题，
     * 假如有一个请求调用了exists函数判断key存在，但是在下一时刻这个缓存过期了，或者被删掉了。
     * 这时候再去缓存中获取值的时候返回的就是null了。
     * 可以先获取缓存的值，再去判断key是否存在。
     *
     * @param cacheKey
     * @return
     */
    @Override
    public RedisCacheElement get(final RedisCacheKey cacheKey) {

        Assert.notNull(cacheKey, "CacheKey must not be null!");

        // 根据key获取缓存值
        RedisCacheElement redisCacheElement = new RedisCacheElement(cacheKey,
    fromStoreValue(lookup(cacheKey)));
        // 判断key是否存在
        Boolean exists = (Boolean) redisOperations.execute(new
    RedisCallback<Boolean>() {

            @Override
            public Boolean doInRedis(RedisConnection connection) throws
    DataAccessException {
                return connection.exists(cacheKey.getKeyBytes());
            }
        });

        if (!exists.booleanValue()) {
            return null;
        }

        return redisCacheElement;
    }

    /**
     * 刷新缓存数据
     */
    private void refreshCache(Object key, String cacheKeyStr) {
```

```
    Long ttl = this.redisOperations.getExpire(cacheKeyStr);
    if (null != ttl && ttl <= CustomizedRedisCache.this.preloadSecondTime) {
        // 尽量少的去开启线程，因为线程池是有限的
        ThreadTaskHelper.run(new Runnable() {
            @Override
            public void run() {
                // 加一个分布式锁，只放一个请求去刷新缓存
                RedisLock redisLock = new RedisLock((RedisTemplate)
redisOperations, cacheKeyStr + "_lock");
                try {
                    if (redisLock.lock()) {
                        // 获取锁之后再判断一下过期时间，看是否需要加载数据
                        Long ttl =
CustomizedRedisCache.this.redisOperations.getExpire(cacheKeyStr);
                        if (null != ttl && ttl <=
CustomizedRedisCache.this.preloadSecondTime) {
                            // 通过获取代理方法信息重新加载缓存数据

 CustomizedRedisCache.this.getCacheSupport().refreshCacheByKey(CustomizedRedisCa
che.super.getName(), key.toString());
                        }
                    }
                } catch (Exception e) {
                    logger.info(e.getMessage(), e);
                } finally {
                    redisLock.unlock();
                }
            }
        });
    }
}
```

那么自动刷新肯定要掉用方法访问数据库，获取值后去刷新缓存。这时我们又怎么能去调用方法呢?

我们利用java的反射机制。所以我们要用一个容器来存放缓存方法的方法信息，包括对象，方法名称，参数等等。我们创建了CachedInvocation类来存放这些信息，再将这个类的对象维护到容器中。

CachedInvocation源码:

```
public final class CachedInvocation {


    private Object key;
    private final Object targetBean;
    private final Method targetMethod;
    private Object[] arguments;

    public CachedInvocation(Object key, Object targetBean, Method targetMethod,
Object[] arguments) {
        this.key = key;
        this.targetBean = targetBean;
        this.targetMethod = targetMethod;
        if (arguments != null && arguments.length != 0) {
            this.arguments = Arrays.copyOf(arguments, arguments.length);
        }
    }
```

```
    public Object[] getArguments() {
        return arguments;
    }

    public Object getTargetBean() {
        return targetBean;
    }

    public Method getTargetMethod() {
        return targetMethod;
    }

    public Object getKey() {
        return key;
    }

    /**
     * 必须重写equals和hashCode方法，否则放到set集合里没法去重
     * @param o
     * @return
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        CachedInvocation that = (CachedInvocation) o;

        return key.equals(that.key);
    }

    @Override
    public int hashCode() {
        return key.hashCode();
    }
}
```

维护缓存方法信息的容器和刷新缓存的类CacheSupportImpl 关键代码：

```
private final String SEPARATOR = "#";

/**
 * 记录缓存执行方法信息的容器。
 * 如果有很多无用的缓存数据的话，有可能会照成内存溢出。
 */
private Map<String, Set<CachedInvocation>> cacheToInvocationsMap = new
ConcurrentHashMap<>();

@Autowired
private CacheManager cacheManager;

// 刷新缓存
private void refreshCache(CachedInvocation invocation, String cacheName) {
```

```
    boolean invocationSuccess;
    Object computed = null;
    try {
        // 通过代理调用方法，并记录返回值
        computed = invoke(invocation);
        invocationSuccess = true;
    } catch (Exception ex) {
        invocationSuccess = false;
    }
    if (invocationSuccess) {
        if (!CollectionUtils.isEmpty(cacheToInvocationsMap.get(cacheName))) {
            // 通过cacheManager获取操作缓存的cache对象
            Cache cache = cacheManager.getCache(cacheName);
            // 通过Cache对象更新缓存
            cache.put(invocation.getKey(), computed);
        }
    }
}

private Object invoke(CachedInvocation invocation)
        throws ClassNotFoundException, NoSuchMethodException,
InvocationTargetException, IllegalAccessException {

    final MethodInvoker invoker = new MethodInvoker();
    invoker.setTargetObject(invocation.getTargetBean());
    invoker.setArguments(invocation.getArguments());
    invoker.setTargetMethod(invocation.getTargetMethod().getName());
    invoker.prepare();

    return invoker.invoke();
}

// 注册缓存方法的执行类信息
@Override
public void registerInvocation(Object targetBean, Method targetMethod, Object[]
arguments,
        Set<String> annotatedCacheNames, String cacheKey) {

    // 获取注解上真实的value值
    Collection<String> cacheNames = generateValue(annotatedCacheNames);

    // 获取注解上的key属性值
    Class<?> targetClass = getTargetClass(targetBean);
    Collection<? extends Cache> caches = getCache(cacheNames);
    Object key = generateKey(caches, cacheKey, targetMethod, arguments,
targetBean, targetClass,
            CacheOperationExpressionEvaluator.NO_RESULT);

    // 新建一个代理对象（记录了缓存注解的方法类信息）
    final CachedInvocation invocation = new CachedInvocation(key, targetBean,
targetMethod, arguments);
    for (final String cacheName : cacheNames) {
        if (!cacheToInvocationsMap.containsKey(cacheName)) {
            cacheToInvocationsMap.put(cacheName, new CopyOnWriteArraySet<>());
        }
        cacheToInvocationsMap.get(cacheName).add(invocation);
    }
```

```
}

@Override
public void refreshCache(String cacheName) {
    this.refreshCacheByKey(cacheName, null);
}


// 刷新特定key缓存
@Override
public void refreshCacheByKey(String cacheName, String cacheKey) {
    // 如果根据缓存名称没有找到代理信息类的set集合就不执行刷新操作。
    // 只有等缓存有效时间过了，再走到切面哪里然后把代理方法信息注册到这里来。
    if (!CollectionUtils.isEmpty(cacheToInvocationsMap.get(cacheName))) {
        for (final CachedInvocation invocation :
cacheToInvocationsMap.get(cacheName)) {
            if (!StringUtils.isBlank(cacheKey) &&
invocation.getKey().toString().equals(cacheKey)) {
                logger.info("缓存：{}-{}，重新加载数据", cacheName,
cacheKey.getBytes());
                refreshCache(invocation, cacheName);
            }
        }
    }
}
```

现在刷新缓存和注册缓存执行方法的信息都有了，我们怎么来把这个执行方法信息注册到容器里面呢？这里还少了触发点。

所以我们还需要一个切面，当执行@Cacheable注解获取缓存信息的时候我们还需要注册执行方法的信息，所以我们写了一个切面：

```
/**
 * 缓存拦截，用于注册方法信息
 * @author yuhao.wang
 */
@Aspect
@Component
public class CachingAnnotationsAspect {

    private static final Logger logger =
LoggerFactory.getLogger(CachingAnnotationsAspect.class);

    @Autowired
    private InvocationRegistry cacheRefreshSupport;

    private <T extends Annotation> List<T> getMethodAnnotations(AnnotatedElement
ae, Class<T> annotationType) {
        List<T> anns = new ArrayList<T>(2);
        // look for raw annotation
        T ann = ae.getAnnotation(annotationType);
        if (ann != null) {
            anns.add(ann);
        }
        // look for meta-annotations
        for (Annotation metaAnn : ae.getAnnotations()) {
```

```
            ann = metaAnn.annotationType().getAnnotation(annotationType);
            if (ann != null) {
                anns.add(ann);
            }
        }
        return (anns.isEmpty() ? null : anns);
    }

    private Method getSpecificmethod(ProceedingJoinPoint pjp) {
        MethodSignature methodSignature = (MethodSignature) pjp.getSignature();
        Method method = methodSignature.getMethod();
        // The method may be on an interface, but we need attributes from the
        // target class. If the target class is null, the method will be
        // unchanged.
        Class<?> targetClass =
AopProxyUtils.ultimateTargetClass(pjp.getTarget());
        if (targetClass == null && pjp.getTarget() != null) {
            targetClass = pjp.getTarget().getClass();
        }
        Method specificMethod = ClassUtils.getMostSpecificMethod(method,
targetClass);
        // If we are dealing with method with generic parameters, find the
        // original method.
        specificMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);
        return specificMethod;
    }

    @Pointcut("@annotation(org.springframework.cache.annotation.Cacheable)")
    public void pointcut() {
    }

    @Around("pointcut()")
    public Object registerInvocation(ProceedingJoinPoint joinPoint) throws
Throwable {

        Method method = this.getSpecificmethod(joinPoint);

        List<Cacheable> annotations = this.getMethodAnnotations(method,
Cacheable.class);

        Set<String> cacheSet = new HashSet<String>();
        String cacheKey = null;
        for (Cacheable cacheables : annotations) {
            cacheSet.addAll(Arrays.asList(cacheables.value()));
            cacheKey = cacheables.key();
        }
        cacheRefreshSupport.registerInvocation(joinPoint.getTarget(), method,
joinPoint.getArgs(), cacheSet, cacheKey);
        return joinPoint.proceed();

    }
}
```

注意：一个缓存名称（@Cacheable的value属性），也只能配置一个过期时间，如果配置多个以第一次配置的为准。

至此我们就把完整的设置过期时间和刷新缓存都实现了，当然还可能存在一定问题，希望大家多多指教。

使用这种方式有个不好的地方，我们破坏了Spring Cache的结构，导致我们切换Cache的方式的时候要改代码，有很大的依赖性。

下一篇我将对 redisCacheManager.setExpires()方法进行扩展来实现过期时间和自动刷新，进而不会去破坏Spring Cache的原有结构，切换缓存就不会有问题了。

hibernate和ibatis的区别

Ibatis与Hibernate的区别

一、 hibernate与ibatis之间的比较：

hibernate 是当前最流行的o/r mapping框架，它出身于sf.NET，现在已经成为jboss的一部分了。
ibatis 是另外一种优秀的o/r mapping框架，目前属于apache的一个子项目了。
相对hibernate“o/r”而言，ibatis是一种“sql mapping”的orm实现。
hibernate对数据库结构提供了较为完整的封装，hibernate的o/r mapping实现了pojo 和数据库表之间的映射，以及sql 的自动生成和执行。程序员往往只需定义好了pojo 到数据库表的映射关系，即可通过hibernate 提供的方法完成持久层操作。程序员甚至不需要对sql 的熟练掌握， hibernate/ojb 会根据制定的存储逻辑，自动生成对应的sql 并调用jdbc 接口加以执行。
而ibatis 的着力点，则在于pojo 与sql之间的映射关系。也就是说，ibatis并不会为程序员在运行期自动生成sql 执行。具体的sql 需要程序员编写，然后通过映射配置文件，将sql所需的参数，以及返回的结果字段映射到指定pojo。
使用ibatis 提供的orm机制，对业务逻辑实现人员而言，面对的是纯粹的Java对象。
这一层与通过hibernate 实现orm 而言基本一致，而对于具体的数据操作，hibernate会自动生成sql 语句，而ibatis 则要求开发者编写具体的sql 语句。相对hibernate而言，ibatis 以sql开发的工作量和数据库移植性上的让步，为系统设计提供了更大的自由空间。

二、hibernate与ibatis两者的对比：

1．ibatis非常简单易学，hibernate相对较复杂，门槛较高。
2．二者都是比较优秀的开源产品
3．当系统属于二次开发,无法对数据库结构做到控制和修改,那ibatis的灵活性将比hibernate更适合
4．系统数据处理量巨大，性能要求极为苛刻，这往往意味着我们必须通过经过高度优化的sql语句（或存储过程）才能达到系统性能设计指标。在这种情况下ibatis会有更好的可控性和表现。
5．ibatis需要手写sql语句，也可以生成一部分，hibernate则基本上可以自动生成，偶尔会写一些hql。同样的需求,ibatis的工作量比hibernate要大很多。类似的，如果涉及到数据库字段的修改，hibernate修改的地方很少，而ibatis要把那些sql mapping的地方一一修改。
6．以数据库字段一一对应映射得到的po和hibernte这种对象化映射得到的po是截然不同的，本质区别在于这种po是扁平化的，不像hibernate映射的po是可以表达立体的对象继承，聚合等等关系的，这将会直接影响到你的整个软件系统的设计思路。
7．hibernate现在已经是主流o/r mapping框架，从文档的丰富性，产品的完善性，版本的开发速度都要强于ibatis

三、iBatis与Hibernate区别?

1． iBatis 需要手写sql语句，也可以生成一部分，Hibernate则基本上可以自动生成，偶尔会写一些Hql。同样的需求,iBATIS的工作量比 Hibernate要大很多。类似的，如果涉及到数据库字段的修改，Hibernate修改的地方很少，而iBATIS要把那些sql mapping的地方一一修改。

2. iBatis 可以进行细粒度的优化

比如说我有一个表，这个表有几个或者几十个字段，我需要更新其中的一个字段，iBatis 很简单，执行一个sql UPDATE TABLE_A SET column_1=#column_1# WHERE id=#id# 但是用 Hibernate 的话就比较麻烦了,缺省的情况下 hibernate 会更新所有字段。 当然我记得 hibernate 有一个选项可以控制只保存修改过的字段，但是我不太确定这个功能的负面效果。

例如：我需要列出一个表的部分内容，用 iBatis 的时候，这里面的好处是可以少从数据库读很多数据，节省流量SELECT ID, NAME FROM TABLE_WITH_A_LOT_OF_COLUMN WHERE ...一般情况下Hibernate会把所有的字段都选出来。比如说有一个上面表有8个字段，其中有一两个比较大的字段，varchar(255)/text。上面的场景中我为什么要把他们 也选出来呢？用hibernate 的话，你又不能把这两个不需要的字段设置为lazy load，因为还有很多地方需要一次把整个 domain object 加载出来。这个时候就能显现出ibatis 的好处了。如果我需要更新一条记录（一个对象），如果使用 hibernate，需要现把对象 select 出来，然后再做 update。这对数据库来说就是两条sql。而iBatis只需要一条update的sql就可以了。减少一次与数据库的交互，对于性能的提升是非常重 要。

3. 开发方面：

开发效率上，我觉得两者应该差不多。可维护性方面，我觉得 iBatis 更好一些。因为 iBatis 的 sql 都保存到单独的文件中。而 Hibernate 在有些情况下可能会在 java 代码中保sql/hql。相对Hibernate“O/R”而言，iBATIS 是一种“Sql Mapping”的ORM实现。 而iBATIS 的着力点，则在于POJO 与SQL之间的映射关系。也就是说，iBATIS并不会为程序员在运行期自动生成SQL 执行。具体的SQL 需要程序员编写，然后通过映射配置文件，将SQL所需的参数，以及返回的结果字段映射到指定POJO。使用iBATIS 提供的ORM机制，对业务逻辑实现人员而言，面对的是纯粹的Java对象，这一层与通过Hibernate 实现ORM而言基本一致，而对于具体的数据操作，Hibernate会自动生成SQL 语句，而iBATIS 则要求开发者编写具体的SQL 语句。相对Hibernate而言，iBATIS 以SQL开发的工作量和数据库移植性上的让步，为系统设计提供了更大的自由空间。

4. 运行效率：在不考虑 cache 的情况下，iBatis 应该会比hibernate 快一些或者很多。

四、选择Hibernate还是iBATIS都有它的道理：

（1）、Hibernate功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码会相当简单，需要写的代码很少，开发速度很快，非常爽。

Hibernate的缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好Hibernate方面需要你的经验和能力都很强才行。

（2）、iBATiS入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

iBATIS的缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

建议：如果你的团队没有Hibernate高手，那么请用iBATIS。要把Hibernate用好，并不容易；否则你应该选择Hibernate，这样的话你的开发速度和代码简洁性都非常棒。

讲讲mybatis的连接池。

Mybatis数据源与连接池

对于ORM框架而言，数据源的组织是一个非常重要的一部分，这直接影响到框架的性能问题。本文将通过对MyBatis框架的数据源结构进行详尽的分析，并且深入解析MyBatis的连接池。

本文首先会讲述MyBatis的数据源的分类，然后会介绍数据源是如何加载和使用的。紧接着将分类介绍UNPOOLED、POOLED和JNDI类型的数据源组织；期间我们会重点讲解POOLED类型的数据源和其实现的连接池原理。

一、MyBatis数据源DataSource分类

MyBatis数据源实现是在以下四个包中：

```
org.apache.ibatis.datasource

org.apache.ibatis.datasource.jndi

org.apache.ibatis.datasource.pooled

org.apache.ibatis.datasource.unpooled
```

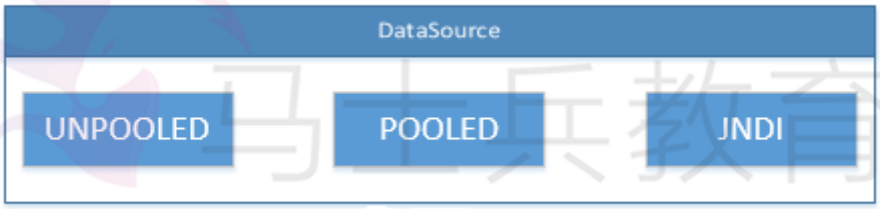
MyBatis把数据源DataSource分为三种：

ž UNPOOLED　不使用连接池的数据源

ž POOLED　使用连接池的数据源

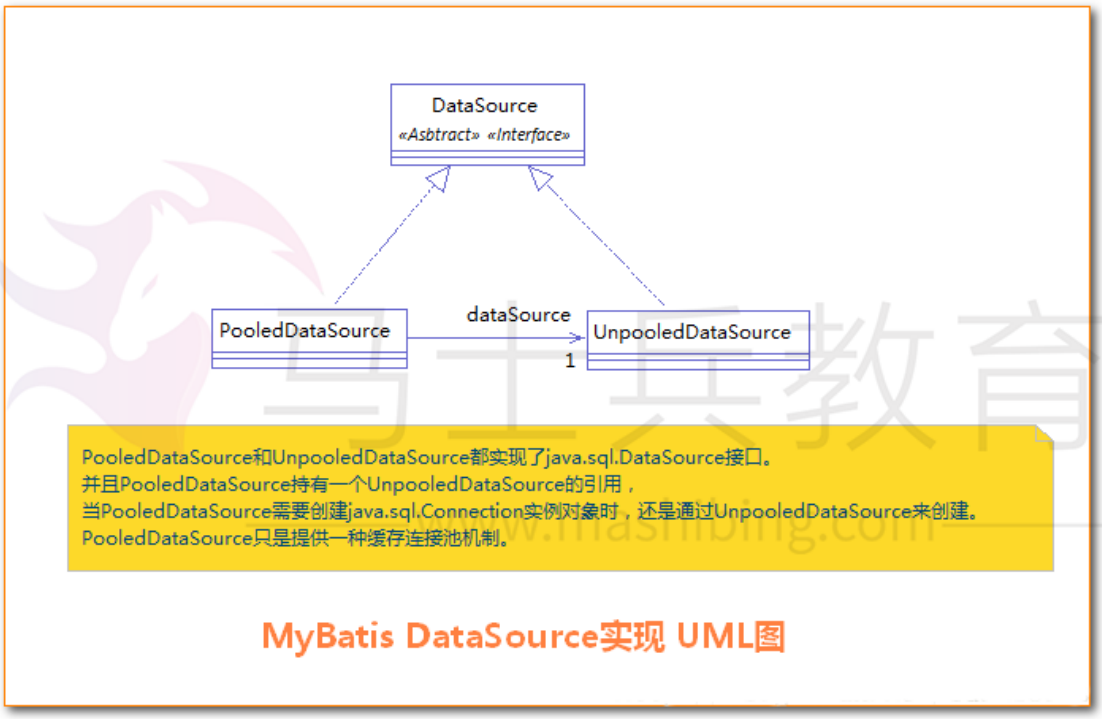
ž JNDI　　使用JNDI实现的数据源

即：



马士兵教育
www.mashibing.com

相应地，MyBatis内部分别定义了实现了java.sql.DataSource接口的UnpooledDataSource，PooledDataSource类来表示UNPOOLED、POOLED类型的数据源。



对于JNDI类型的数据源DataSource，则是通过JNDI上下文中取值。

二、数据源DataSource的创建过程

MyBatis*数据源DataSource**对象的创建发生在**MyBatis**初始化的过程中。***下面让我们一步步地了解MyBatis是如何创建数据源DataSource的。

在mybatis的XML配置文件中，使用元素来配置数据源：

1. MyBatis在初始化时，解析此文件，根据的type属性来创建相应类型的的数据源DataSource，即：

   - type="POOLED"：MyBatis会创建PooledDataSource实例
   - type="UNPOOLED"：MyBatis会创建UnpooledDataSource实例
   - type="JNDI" ：MyBatis会从JNDI服务上查找DataSource实例，然后返回使用

2. 顺便说一下，MyBatis是通过工厂模式来创建数据源DataSource对象的，MyBatis定义了抽象的工厂接口:***org.apache.ibatis.datasource.DataSourceFactory***,通过其getDataSource()方法返回数据源DataSource：

定义如下：

```
public interface DataSourceFactory {

 void setProperties(Properties props);
 //生产DataSource
 DataSource getDataSource();
}
```

上述三种不同类型的type，则有对应的以下dataSource工厂：

- POOLED PooledDataSourceFactory
- UNPOOLED UnpooledDataSourceFactory
- JNDI JndiDataSourceFactory

其类图如下所示：

3. MyBatis创建了DataSource实例后，会将其放到Configuration对象内的Environment对象中， 供以后使用。

三、 DataSource什么时候创建Connection对象

当我们需要创建SqlSession对象并需要执行SQL语句时，这时候MyBatis才会去调用dataSource对象来创建java.sql.Connection对象。也就是说，java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。

比如，我们有如下方法执行一个简单的SQL语句：

```
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
sqlSession.selectList("SELECT * FROM STUDENTS");
```

前4句都不会导致java.sql.Connection对象的创建，只有当第5句sqlSession.selectList("SELECT * FROM STUDENTS")，才会触发MyBatis在底层执行下面这个方法来创建java.sql.Connection对象：

```
protected void openConnection() throws SQLException {
if (log.isDebugEnabled()) {
log.debug("Opening JDBC Connection");
}
connection = dataSource.getConnection();
if (level != null) {
connection.setTransactionIsolation(level.getLevel());
}
setDesiredAutoCommit(autoCommmit);
}
```

而对于DataSource的UNPOOLED的类型的实现-UnpooledDataSource是怎样实现getConnection()方法的呢？请看下一节。

四、不使用连接池的UnpooledDataSource

当 的type属性被配置成了”UNPOOLED”，MyBatis首先会实例化一个UnpooledDataSourceFactory工厂实例，然后通过.getDataSource()方法返回一个UnpooledDataSource实例对象引用，我们假定为dataSource。

使用UnpooledDataSource的getConnection(),每调用一次就会产生一个新的Connection实例对象。

UnPooledDataSource的getConnection()方法实现如下：

```
/*
UnpooledDataSource的getConnection()实现
*/
public Connection getConnection() throws SQLException
{
return doGetConnection(username, password);
}

private Connection doGetConnection(String username, String password) throws
SQLException
{
//封装username和password成properties
Properties props = new Properties();
if (driverProperties != null)
{
props.putAll(driverProperties);
}
if (username != null)
{
props.setProperty("user", username);
}
if (password != null)
{
props.setProperty("password", password);
}
return doGetConnection(props);
}

/*
 *  获取数据连接
 */
private Connection doGetConnection(Properties properties) throws
SQLException
{
    //1.初始化驱动
    initializeDriver();
    //2.从DriverManager中获取连接，获取新的Connection对象
    Connection connection = DriverManager.getConnection(url, properties);
    //3.配置connection属性
    configureConnection(connection);
    return connection;
}
```

如上代码所示，UnpooledDataSource会做以下事情：

1. **初始化驱动：** 判断driver驱动是否已经加载到内存中，如果还没有加载，则会动态地加载driver类，并实例化一个Driver对象，使用DriverManager.registerDriver()方法将其注册到内存中，以供后续使用。
2. **创建Connection对象：** 使用DriverManager.getConnection()方法创建连接。
3. **配置Connection对象：** 设置是否自动提交autoCommit和隔离级别isolationLevel。
4. **返回Connection对象。**

上述的序列图如下所示：

总结：从上述的代码中可以看到，***我们每调用一次**getConnection()**方法，都会通过**DriverManager.getConnection()**返回新的**java.sql.Connection**实例。***

五、为什么要使用连接池?

1. 创建一个java.sql.Connection实例对象的代价

首先让我们来看一下创建一个java.sql.Connection对象的资源消耗。我们通过连接Oracle数据库，创建创建Connection对象，来看创建一个Connection对象、执行SQL语句各消耗多长时间。代码如下：

```
public static void main(String[] args) throws Exception
{

String sql = "select * from hr.employees where employee_id < ? and
employee_id >= ?";
PreparedStatement st = null;
ResultSet rs = null;

long beforeTimeOffset = -1L; //创建Connection对象前时间
long afterTimeOffset = -1L; //创建Connection对象后时间
long executeTimeOffset = -1L; //创建Connection对象后时间

Connection con = null;
Class.forName("oracle.jdbc.driver.OracleDriver");

beforeTimeOffset = new Date().getTime();
System.out.println("before:\t" + beforeTimeOffset);

con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
"louluan", "123456");

afterTimeOffset = new Date().getTime();
System.out.println("after:\t\t" + afterTimeOffset);
System.out.println("Create Costs:\t\t" + (afterTimeOffset -
beforeTimeOffset) + " ms");

st = con.prepareStatement(sql);
//设置参数
st.setInt(1, 101);
st.setInt(2, 0);
//查询，得出结果集
rs = st.executeQuery();
executeTimeOffset = new Date().getTime();
System.out.println("Exec Costs:\t\t" + (executeTimeOffset - afterTimeOffset)
+ " ms");

}
```

上述程序在我笔记本上的执行结果为：

```
before:          1404363138126
after:           1404363138376
Create Costs:    250 ms
Exec Costs:      170 ms
```

从此结果可以清楚地看出，创建一个Connection对象，用了**250 毫秒**；而执行SQL的时间用了**170 毫秒。**

创建一个Connection对象用了250毫秒！这个时间对计算机来说可以说是一个**非常奢侈**的！

这仅仅是一个Connection对象就有这么大的代价，设想一下另外一种情况：如果我们在Web应用程序中，为用户的每一个请求就操作一次数据库，当有10000个在线用户并发操作的话，对计算机而言，仅仅创建Connection对象不包括做业务的时间就要损耗10000×250ms= 250 0000 ms = 2500 s = 41.6667 min,竟然要***41***分钟！！！如果对高用户群体使用这样的系统，简直就是开玩笑!

2. 问题分析：

创建一个java.sql.Connection对象的代价是如此巨大，是因为创建一个Connection对象的过程，在底层就相当于和数据库建立的通信连接，在建立通信连接的过程，消耗了这么多的时间，而往往我们建立连接后（即创建Connection对象后），就执行一个简单的SQL语句，然后就要抛弃掉，这是一个非常大的资源浪费!

3.解决方案:

对于需要频繁地跟数据库交互的应用程序，可以在创建了Connection对象，并操作完数据库后，可以不释放掉资源，而是将它放到内存中，当下次需要操作数据库时，可以直接从内存中取出Connection对象，不需要再创建了，这样就极大地节省了创建Connection对象的资源消耗。由于内存也是有限和宝贵的，这又对我们对内存中的Connection对象怎么有效地维护提出了很高的要求。我们将在内存中存放Connection对象的容器称之为 连接池（Connection Pool）。下面让我们来看一下MyBatis的线程池是怎样实现的。

六、使用了连接池的PooledDataSource

同样地，我们也是使用PooledDataSource的getConnection()方法来返回Connection对象。现在让我们看一下它的基本原理：

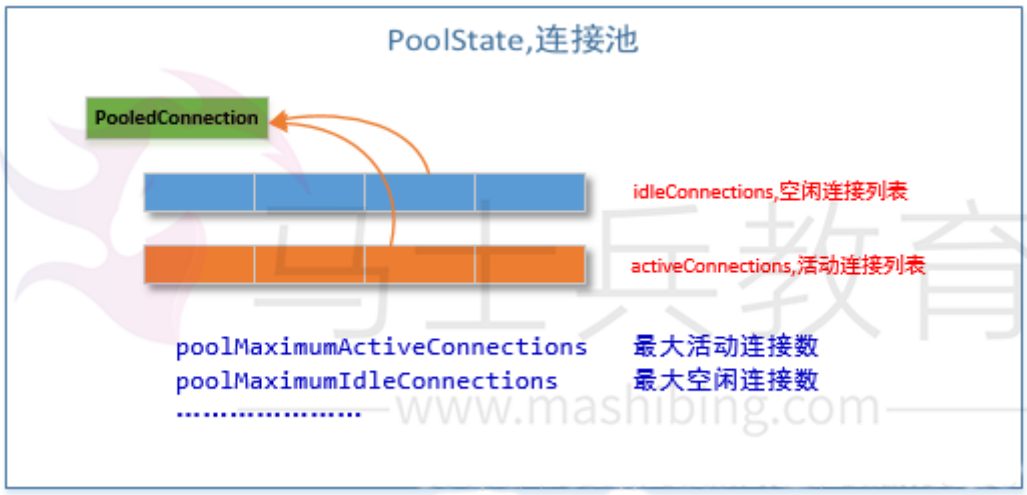
PooledDataSource将java.sql.Connection对象包裹成PooledConnection对象放到了PoolState类型的容器中维护。 MyBatis将连接池中的PooledConnection分为两种状态： 空闲状态（idle）和活动状态(active)，这两种状态的PooledConnection对象分别被存储到PoolState容器内的**idleConnections**和**activeConnections**两个List集合中：

idleConnections:空闲(idle)状态PooledConnection对象被放置到此集合中，表示当前闲置的没有被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从此集合中取PooledConnection对象。当用完一个java.sql.Connection对象时，MyBatis会将其包裹成PooledConnection对象放到此集合中。

activeConnections:活动(active)状态的PooledConnection对象被放置到名为activeConnections的ArrayList中，表示当前正在被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从idleConnections集合中取PooledConnection对象,如果没有，则看此集合是否已满，如果未满，PooledDataSource会创建出一个PooledConnection，添加到

此集合中，并返回。

PoolState连接池的大致结构如下所示：



6.1 获取java.sql.Connection对象的过程

下面让我们看一下PooledDataSource 的getConnection()方法获取Connection对象的实现：

```
public Connection getConnection() throws SQLException {
return popConnection(dataSource.getUsername(),
dataSource.getPassword()).getProxyConnection();
}

public Connection getConnection(String username, String password) throws
SQLException {
return popConnection(username, password).getProxyConnection();
}
```

上述的popConnection()方法，会从连接池中返回一个可用的PooledConnection对象，然后再调用getProxyConnection()方法最终返回Conection对象。（至于为什么会有getProxyConnection(),请关注下一节）

现在让我们看一下popConnection()方法到底做了什么：

1. 先看是否有空闲(idle)状态下的PooledConnection对象，如果有，就直接返回一个可用的PooledConnection对象；否则进行第2步。
2. 查看活动状态的PooledConnection池activeConnections是否已满；如果没有满，则创建一个新的PooledConnection对象，然后放到activeConnections池中，然后返回此PooledConnection对象；否则进行第三步；
3. 看最先进入activeConnections池中的PooledConnection对象是否已经过期：如果已经过期，从activeConnections池中移除此对象，然后创建一个新的PooledConnection对象，添加到activeConnections中，然后将此对象返回；否则进行第4步。

4. 线程等待，循环2步

```
/*
 * 传递一个用户名和密码，从连接池中返回可用的PooledConnection
 */
private PooledConnection popConnection(String username, String password)
throws SQLException
{
    boolean countedWait = false;
    PooledConnection conn = null;
    long t = System.currentTimeMillis();
    int localBadConnectionCount = 0;

    while (conn == null)
    {
        synchronized (state)
        {
            if (state.idleConnections.size() > 0)
            {
                // 连接池中有空闲连接，取出第一个
                conn = state.idleConnections.remove(0);
                if (log.isDebugEnabled())
                {
                    log.debug("Checked out connection " +
conn.getRealHashCode() + " from pool.");
                }
            }
            else
            {
                // 连接池中没有空闲连接，则取当前正在使用的连接数小于最大限定值，
                if (state.activeConnections.size() <
poolMaximumActiveConnections)
                {
                    // 创建一个新的connection对象
                    conn = new PooledConnection(dataSource.getConnection(),
this);
                    @SuppressWarnings("unused")
                    //used in logging, if enabled
                    Connection realConn = conn.getRealConnection();
                    if (log.isDebugEnabled())
                    {
                        log.debug("Created connection " +
conn.getRealHashCode() + ".");
                    }
                }
                else
                {
                    // Cannot create new connection 当活动连接池已满，不能创建
时，取出活动连接池的第一个，即最先进入连接池的PooledConnection对象
                    // 计算它的校验时间，如果校验时间大于连接池规定的最大校验时间，则认
为它已经过期了，利用这个PoolConnection内部的realConnection重新生成一个
PooledConnection
                    //
                    PooledConnection oldestActiveConnection =
state.activeConnections.get(0);
                    long longestCheckoutTime =
oldestActiveConnection.getCheckoutTime();
```

```
                    if (longestCheckoutTime > poolMaximumCheckoutTime)
                    {
                        // Can claim overdue connection
                        state.claimedOverdueConnectionCount++;
                        state.accumulatedCheckoutTimeOfOverdueConnections +=
longestCheckoutTime;
                        state.accumulatedCheckoutTime +=
longestCheckoutTime;

 state.activeConnections.remove(oldestActiveConnection);
                        if
(!oldestActiveConnection.getRealConnection().getAutoCommit())
                        {

 oldestActiveConnection.getRealConnection().rollback();
                        }
                        conn = new
PooledConnection(oldestActiveConnection.getRealConnection(), this);
                        oldestActiveConnection.invalidate();
                        if (log.isDebugEnabled())
                        {
                            log.debug("Claimed overdue connection " +
conn.getRealHashCode() + ".");
                        }
                    }
                    else
                    {

                        //如果不能释放，则必须等待有
                        // Must wait
                        try
                        {
                            if (!countedWait)
                            {
                                state.hadToWaitCount++;
                                countedWait = true;
                            }
                            if (log.isDebugEnabled())
                            {
                                log.debug("Waiting as long as " +
poolTimeToWait + " milliseconds for connection.");
                            }
                            long wt = System.currentTimeMillis();
                            state.wait(poolTimeToWait);
                            state.accumulatedWaitTime +=
System.currentTimeMillis() - wt;
                        }
                        catch (InterruptedException e)
                        {
                            break;
                        }
                    }
                }
            }

            //如果获取PooledConnection成功，则更新其信息

            if (conn != null)
```

```
            {
                if (conn.isValid())
                {
                    if (!conn.getRealConnection().getAutoCommit())
                    {
                        conn.getRealConnection().rollback();
                    }

 conn.setConnectionTypeCode(assembleConnectionTypeCode(dataSource.getUrl(),
username, password));
                    conn.setCheckoutTimestamp(System.currentTimeMillis());
                    conn.setLastUsedTimestamp(System.currentTimeMillis());
                    state.activeConnections.add(conn);
                    state.requestCount++;
                    state.accumulatedRequestTime +=
System.currentTimeMillis() - t;
                }
                else
                {
                    if (log.isDebugEnabled())
                    {
                        log.debug("A bad connection (" +
conn.getRealHashCode() + ") was returned from the pool, getting another
connection.");
                    }
                    state.badConnectionCount++;
                    localBadConnectionCount++;
                    conn = null;
                    if (localBadConnectionCount >
(poolMaximumIdleConnections + 3))
                    {
                        if (log.isDebugEnabled())
                        {
                            log.debug("PooledDataSource: Could not get a
good connection to the database.");
                        }
                        throw new SQLException("PooledDataSource: Could not
get a good connection to the database.");
                    }
                }
            }
        }

    }

    if (conn == null)
    {
        if (log.isDebugEnabled())
        {
            log.debug("PooledDataSource: Unknown severe error condition.
The connection pool returned a null connection.");
        }
        throw new SQLException("PooledDataSource: Unknown severe error
condition.  The connection pool returned a null connection.");
    }

    return conn;
}
```

对应的处理流程图如下所示：

如上所示,对于PooledDataSource的getConnection()方法内，先是调用类PooledDataSource的popConnection()方法返回了一个PooledConnection对象，然后调用了PooledConnection的getProxyConnection()来返回Connection对象。

6.2java.sql.Connection对象的回收

当我们的程序中使用完Connection对象时，如果不使用数据库连接池，我们一般会调用connection.close()方法，关闭connection连接，释放资源。如下所示：

```
private void test() throws ClassNotFoundException, SQLException
{
    String sql = "select * from hr.employees where employee_id < ? and
employee_id >= ?";
    PreparedStatement st = null;
    ResultSet rs = null;

    Connection con = null;
    Class.forName("oracle.jdbc.driver.OracleDriver");
    try
    {
        con =
DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
"louluan", "123456");
        st = con.prepareStatement(sql);
        //设置参数
        st.setInt(1, 101);
        st.setInt(2, 0);
        //查询，得出结果集
        rs = st.executeQuery();
        //取数据，省略
        //关闭，释放资源
        con.close();
    }
    catch (SQLException e)
    {
        con.close();
        e.printStackTrace();
    }
}
```

调用过close()方法的Connection对象所持有的资源会被全部释放掉，Connection对象也就不能再使用。

那么，如果我们使用了连接池，我们在用完了Connection对象时，需要将它放在连接池中，该怎样做呢?

可能大家第一个在脑海里闪现出来的想法就是：我在应该调用con.close()方法的时候，不调用close()f方法，将其换成将Connection对象放到连接池容器中的代码！

好，我们将上述的想法实现，首先定义一个简易连接池Pool，然后将上面的代码改写：

```
package com.foo.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Vector;

/**
 *
 * 一个线程安全的简易连接池实现，此连接池是单例的
 *  putConnection()将Connection添加到连接池中
 *  getConnection()返回一个Connection对象
 */
public class Pool {

    private static Vector<Connection> pool = new Vector<Connection>();

    private static int MAX_CONNECTION =100;

    private static String DRIVER="oracle.jdbc.driver.OracleDriver";
    private static String URL = "jdbc:oracle:thin:@127.0.0.1:1521:xe";
    private static String USERNAME = "louluan";
    private static String PASSWROD = "123456";

    static {
        try {
            Class.forName(DRIVER);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * 将一个Connection对象放置到连接池中
     */
    public static  void putConnection(Connection connection){

        synchronized(pool)
        {
            if(pool.size()<MAX_CONNECTION)
            {
                pool.add(connection);
            }
        }
    }


    /**
     * 返回一个Connection对象，如果连接池内有元素，则pop出第一个元素；
     * 如果连接池Pool中没有元素，则创建一个connection对象，然后添加到pool中
     * @return Connection
```

```
     */
    public static Connection getConnection(){
        Connection connection = null;
        synchronized(pool)
        {
            if(pool.size()>0)
            {
                connection = pool.get(0);
                pool.remove(0);
            }
            else
            {
                connection = createConnection();
                pool.add(connection);
            }
        }
        return connection;
    }

    /**
     * 创建一个新的Connection对象
     */
    private static Connection createConnection()
    {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(URL,
USERNAME,PASSWROD);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return connection;
    }

}
package com.foo.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Vector;

public class PoolTest
{

    private void test() throws ClassNotFoundException, SQLException
    {
        String sql = "select * from hr.employees where employee_id < ? and
employee_id >= ?";
        PreparedStatement st = null;
        ResultSet rs = null;
        Connection con = null;
        Class.forName("oracle.jdbc.driver.OracleDriver");
        try
        {
```

```
            con =
DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
"louluan", "123456");
            st = con.prepareStatement(sql);
            //设置参数
            st.setInt(1, 101);
            st.setInt(2, 0);
            //查询，得出结果集
            rs = st.executeQuery();
            //取数据，省略
            //将不再使用的Connection对象放到连接池中，供以后使用
            Pool.putConnection(con);
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}
```

上述的代码就是将我们使用过的Connection对象放到Pool连接池中，我们需要Connection对象的话，只需要使用Pool.getConnection()方法从里面取即可。

是的,上述的代码完全可以实现此能力，不过有一个很不优雅的实现：**就是我们需要手动地将Connection对象放到Pool连接池中，这是一个很傻的实现方式。这也和一般使用Connection对象的方式不一样：一般使用Connection的方式是使用完后，然后调用.close()方法释放资源。**

为了和一般的使用Conneciton对象的方式保持一致，我们希望当Connection使用完后，调用.close()方法，而实际上Connection资源并没有被释放，而实际上被添加到了连接池中。这样可以做到吗？答案是可以。上述的要求从另外一个角度来描述就是：能否提供一种机制，让我们知道Connection对象调用了什么方法，从而根据不同的方法自定义相应的处理机制。恰好代理机制就可以完成上述要求.

怎样实现Connection对象调用了close()方法，而实际是将其添加到连接池中

这是要使用代理模式，为真正的Connection对象创建一个代理对象，代理对象所有的方法都是调用相应的真正Connection对象的方法实现。当代理对象执行close()方法时，要特殊处理，不调用真正Connection对象的close()方法，而是将Connection对象添加到连接池中。

MyBatis的PooledDataSource的PoolState内部维护的对象是PooledConnection类型的对象，而PooledConnection则是对真正的数据库连接java.sql.Connection实例对象的包裹器。

PooledConnection对象内持有一个真正的数据库连接java.sql.Connection实例对象和一个java.sql.Connection的代理：

其部分定义如下：

```
class PooledConnection implements InvocationHandler {

  //......
  //所创建它的datasource引用
  private PooledDataSource dataSource;
  //真正的Connection对象
  private Connection realConnection;
  //代理自己的代理Connection
  private Connection proxyConnection;

  //......
}
```

PooledConenction

实现了

InvocationHandler

接口，并且，

proxyConnection

对象也是根据这个它来生成的代理对象：

```
public PooledConnection(Connection connection, PooledDataSource dataSource)
{
    this.hashCode = connection.hashCode();
    this.realConnection = connection;
    this.dataSource = dataSource;
    this.createdTimestamp = System.currentTimeMillis();
    this.lastUsedTimestamp = System.currentTimeMillis();
    this.valid = true;
    this.proxyConnection = (Connection)
Proxy.newProxyInstance(Connection.class.getClassLoader(), IFACES, this);
  }
```

实际上，我们调用PooledDataSource的getConnection()方法返回的就是这个proxyConnection对象。

当我们调用此proxyConnection对象上的任何方法时，都会调用PooledConnection对象内invoke()方法。

让我们看一下PooledConnection类中的invoke()方法定义：

```
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    String methodName = method.getName();
    //当调用关闭的时候，回收此Connection到PooledDataSource中
    if (CLOSE.hashCode() == methodName.hashCode() &&
CLOSE.equals(methodName)) {
      dataSource.pushConnection(this);
      return null;
    } else {
```

```
      try {
        if (!Object.class.equals(method.getDeclaringClass())) {
          checkConnection();
        }
        return method.invoke(realConnection, args);
      } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
      }
    }
  }
```

从上述代码可以看到，当我们使用了pooledDataSource.getConnection()返回的Connection对象的close()方法时，不会调用真正Connection的close()方法，而是将此Connection对象放到连接池中。

七、JNDI类型的数据源DataSource

对于JNDI类型的数据源DataSource的获取就比较简单，MyBatis定义了一个JndiDataSourceFactory工厂来创建通过JNDI形式生成的DataSource。

下面让我们看一下JndiDataSourceFactory的关键代码：

```
if (properties.containsKey(INITIAL_CONTEXT)
&& properties.containsKey(DATA_SOURCE))
{
//从JNDI上下文中找到DataSource并返回
Context ctx = (Context)
initCtx.lookup(properties.getProperty(INITIAL_CONTEXT));
dataSource = (DataSource) ctx.lookup(properties.getProperty(DATA_SOURCE));
}
else if (properties.containsKey(DATA_SOURCE))
{
// //从JNDI上下文中找到DataSource并返回
dataSource = (DataSource)
initCtx.lookup(properties.getProperty(DATA_SOURCE));
}
```

经典面试题

问题一

什么是 Spring Boot?

多年来，随着新功能的增加，spring 变得越来越复杂。只需访问 https://spring.io/projects

页面，我们就会看到可以在我们的应用程序中使用的所有 Spring 项目的不同功能。如果必须启动一个新的 Spring 项目，我们必须添加构建路径或添加 Maven 依赖关系，配置应用程序服务器，添加 spring 配置。因此，开始一个新的 spring 项目需要很多努力，因为我们现在必须从头开始做所有事情。

Spring Boot 是解决这个问题的方法。Spring Boot 已经建立在现有 spring 框架之上。使用 spring 启动，我们避免了之前我们必须做的所有样板代码和配置。因此，Spring Boot 可以帮助我们以最少的工作量，更加健壮地使用现有的 Spring 功能。

问题二

Spring Boot 有哪些优点?

Spring Boot 的优点有：

减少开发，测试时间和努力。

使用 JavaConfig 有助于避免使用 XML。

避免大量的 Maven 导入和各种版本冲突。

提供意见发展方法。

通过提供默认值快速开始开发。

没有单独的 Web 服务器需要。这意味着你不再需要启动 Tomcat，Glassfish 或其他任何东西。

需要更少的配置 因为没有 web.xml 文件。只需添加用@ Configuration 注释的类，然后添加用@Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。您甚至可以将

@Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。

基于环境的配置 使用这些属性，您可以将您正在使用的环境传递到应用程序：Dspring.profiles.active = {enviornment}。在加载主应用程序属性文件后，Spring 将在

（application{environment} .properties）中加载后续的应用程序属性文件。

问题三

什么是 JavaConfig?

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯 Java 方法。因此有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。

减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将 JavaConfig 与 XML 混合匹配是理想的。

类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring 容器。由于

Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

问题四

如何重新加载 Spring Boot 上的更改，而无需重新启动服务器?

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式 tomcat

将重新启动。Spring Boot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。

它还提供 H2 数据库控制台以更好地测试应用程序。

org.springframework.boot spring-boot-devtools true

问题五

Spring Boot 中的监视器是什么?

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的 REST 端点来检查状态。

问题六

如何在 Spring Boot 中禁用 Actuator 端点安全性?

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访

问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。 我们可以使用

management.security.enabled = false

来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

问题七

如何在自定义端口上运行 Spring Boot 应用程序?

为了在自定义端口上运行 Spring Boot 应用程序，您可以在 application.properties 中指定端口。

server.port = 8090

问题八

什么是 YAML?

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。

与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

问题九

如何实现 Spring Boot 应用程序的安全性?

为了实现 Spring Boot 的安全性，我们使用 spring-boot-starter-security 依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展 WebSecurityConfigurerAdapter 并覆盖其方法。

问题十

如何集成 Spring Boot 和 ActiveMQ?

对于集成 Spring Boot 和 ActiveMQ，我们使用

spring-boot-starter-activemq 依赖关系。 它只需要很少的配置，并且不需要样板代码。

问题十一

如何使用 Spring Boot 实现分页和排序?

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的

org.springframework.data.domain.Pageable 传递给存储库方法。

问题十二

什么是 Swagger？你用 Spring Boot 实现了它吗?

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻

辑来理解远程服务并与其进行交互。因此，Swagger 消除了调用服务时的猜测。

问题十三

什么是 Spring Profiles?

Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION 中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

问题十四

什么是 Spring Batch?

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单

以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

问题十五

什么是 FreeMarker 模板?

FreeMarker 是一个基于 Java 的模板引擎，最初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码，而设计人员可以处理 html 页面设计。最后使用 freemarker 可以将这些结合起来，给出最终的输出页面。

问题十六

如何使用 Spring Boot 实现异常处理?

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。 我们通过实现一个

ControlerAdvice 类，来处理控制器类抛出的所有异常。

问题十七

您使用了哪些 starter maven 依赖项?

使用了下面的一些依赖项

spring-boot-starter-activemq spring-boot-starter-security

spring-boot-starter-web

这有助于增加更少的依赖关系，并减少版本的冲突。

问题十八

什么是 CSRF 攻击?

CSRF 代表跨站请求伪造。这是一种攻击，迫使最终用户在当前通过身份验证的 Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

问题十九

什么是 WebSockets?

WebSocket 是一种计算机通信协议，通过单个 TCP 连接提供全双工通信信道。

WebSocket 是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。

WebSocket 是全双工的 -客户端和服务器通信是相互独立的。

单个 TCP 连接 -初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信

Light -与 http 相比，WebSocket 消息数据交换要轻得多。

问题二十

什么是 AOP?

在软件开发过程中，跨越应用程序多个点的功能称为交叉问题。这些交叉问题与应用程序的主要业务逻辑不同。因此，将这些横切关注与业务逻辑分开是面向方面编程（AOP）的地方。

问题二十一

什么是 Apache Kafka?

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的，容错的发布 - 订阅消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离线和在线消息消费。

问题二十二

我们如何监视所有 Spring Boot 微服务?

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。

Spring Cloud（一明老师，晁老师）

Spring Cloud熔断机制介绍；

在Spring Cloud框架里，熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失 败的调用到一定阈值，缺省是5秒内20次调用失败，就会启动熔断机制。熔断机制的注解是@HystrixCommand，Hystrix会找有这个注解的方法，并将这类方法关联到和熔断器连在一起 的代理上。当前，@HystrixCommand仅当类的注解为@Service或@Component时才会发挥 作用。

Spring Cloud对比下Dubbo，什么场景下该使用Spring Cloud?

两者所解决的问题域不一样：Dubbo的定位始终是一款RPC框架，**而Spring Cloud的目的是微 服务架构下的一站式解决方案。**

Spring Cloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式。

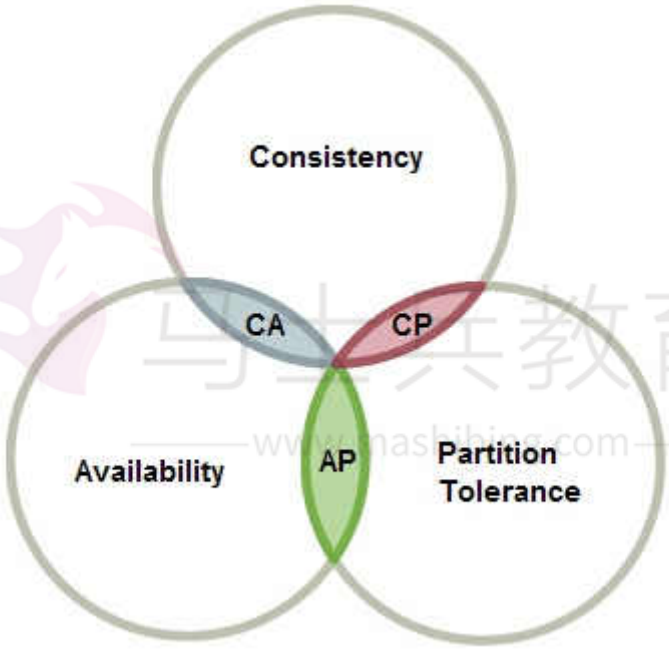
严格来说，这两种方式各有优劣。虽然在一定程度上来说，后者牺牲了服务调用的性能，但也避 免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖 只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合 适。

核心要素	Dubbo	Spring Cloud
服务注册中心	Zookeeper、Redis	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud
断路器	不完善	Spring Cloud Netiflix Hystrix
分布式配置	无	Spring Cloud Config
分布式追踪系统	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream 基于Redis,Kafka实现的消息微服务
批量任务	无	Spring Cloud Task

CAP原理和BASE理论

CAP原则(CAP定理)、BASE理论

一、CAP原则



CAP原则又称CAP定理，指的是在一个分布式系统中， Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。

CAP原则是NOSQL数据库的基石。

分布式系统的CAP理论：理论首先把分布式系统中的三个特性进行了如下归纳：

- 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）
- 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）
- 分区容忍性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

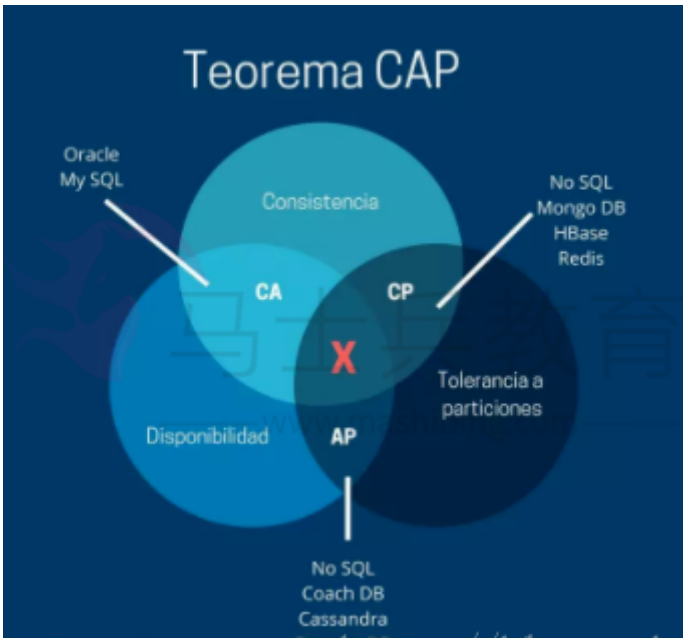
一致性与可用性的决择编辑

CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是我们必须需要实现的。所以我们只能在一致性和可用性之间进行权衡，没有NoSQL系统能同时保证这三点。

对于web2.0网站来说，关系数据库的很多主要特性却往往无用武之地

1. 数据库事务一致性需求
   很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求并不高。允许实现最终一致性。
2. 数据库的写实时性和读实时性需求
   对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之 后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。
3. 对复杂的SQL查询，特别是多表关联查询的需求
   任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计角 度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。

取舍策略



CAP三个特性只能满足其中两个，那么取舍的策略就共有三种：

CA without P：如果不要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。但放弃P的同时也就意味着放弃了系统的扩展性，也就是分布式节点受限，没办法部署子节点，这是违背分布式系统设计的初衷的。传统的关系型数据库RDBMS：Oracle、MySQL就是CA。

CP without A：如果不要求A（可用），相当于每个请求都需要在服务器之间保持强一致，而P（分区）会导致同步时间无限延长(也就是等待数据同步完才能正常访问服务)，一旦发生网络故障或者消息丢失等情况，就要牺牲用户的体验，等待所有数据全部一致了之后再让用户访问系统。设计成CP的系统其实不少，最典型的就是分布式数据库，如Redis、HBase等。对于这些分布式数据库来说，数据的一致性是最基本的要求，因为如果连这个标准都达不到，那么直接采用关系型数据库就好，没必要再浪费资源来部署分布式数据库。

AP wihtout C：要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。典型的应用就如某米的抢购手机场景，可能前几秒你浏览商品的时候页面提示是有库存的，当你选择完商品准备下单的时候，系统提示你下单失败，商品已售完。这其实就是先在 A（可用性）方面保证系统可以正常的服务，然后在数据的一致性方面做了些牺牲，虽然多少会影响一些用户体验，但也不至于造成用户购物流程的严重阻塞。

BASE理论

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的简写，BASE是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的结论，是基于CAP定理逐步演化而来的，其核心思想是即使无法做到强一致性（Strong consistency），但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。接下来我们着重对BASE中的三要素进行详细讲解。

基本可用

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性——但请注意，这绝不等价于系统不可用，以下两个就是“基本可用”的典型例子。

- 响应时间上的损失：正常情况下，一个在线搜索引擎需要0.5秒内返回给用户相应的查询结果，但由于出现异常（比如系统部分机房发生断电或断网故障），查询结果的响应时间增加到了1~2秒。

- 功能上的损失：正常情况下，在一个电子商务网站上进行购物，消费者几乎能够顺利地完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面。

弱状态也称为软状态，和硬状态相对，是指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据听不的过程存在延时。

最终一致性

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性

亚马逊首席技术官Werner Vogels在于2008年发表的一篇文章中对最终一致性进行了非常详细的介绍。他认为最终一致性时一种特殊的弱一致性：系统能够保证在没有其他新的更新操作的情况下，数据最终一定能够达到一致的状态，因此所有客户端对系统的数据访问都能够胡渠道最新的值。同时，在没有发生故障的前提下，数据达到一致状态的时间延迟，取决于网络延迟，系统负载和数据复制方案设计等因素。

在实际工程实践中，最终一致性存在以下五类主要变种。

因果一致性：

> 因果一致性是指，如果进程A在更新完某个数据项后通知了进程B，那么进程B之后对该数据项的访问都应该能够获取到进程A更新后的最新值，并且如果进程B要对该数据项进行更新操作的话，务必基于进程A更新后的最新值，即不能发生丢失更新情况。与此同时，与进程A无因果关系的进程C的数据访问则没有这样的限制。

读己之所写：

> 读己之所写是指，进程A更新一个数据项之后，它自己总是能够访问到更新过的最新值，而不会看到旧值。也就是说，对于单个数据获取者而言，其读取到的数据一定不会比自己上次写入的值旧。因此，读己之所写也可以看作是一种特殊的因果一致性。

会话一致性：

> 会话一致性将对系统数据的访问过程框定在了一个会话当中：系统能保证在同一个有效的会话中实现“读己之所写”的一致性，也就是说，执行更新操作之后，客户端能够在同一个会话中始终读取到该数据项的最新值。

单调读一致性：

> 单调读一致性是指如果一个进程从系统中读取出一个数据项的某个值后，那么系统对于该进程后续的任何数据访问都不应该返回更旧的值。

单调写一致性：

> 单调写一致性是指，一个系统需要能够保证来自同一个进程的写操作被顺序地执行。

以上就是最终一致性的五类常见的变种，在时间系统实践中，可以将其中的若干个变种互相结合起来，以构建一个具有最终一致性的分布式系统。事实上，可以将其中的若干个变种相互结合起来，以构建一个具有最终一致性特性的分布式系统。事实上，最终一致性并不是只有那些大型分布式系统才设计的特性，许多现代的关系型数据库都采用了最终一致性模型。在现代关系型数据库中，大多都会采用同步和异步方式来实现主备数据复制技术。在同步方式中，数据的复制国耻鞲通常是更新事务的一部分，因此在事务完成后，主备数据库的数据就会达到一致。而在异步方式中，备库的更新往往存在延时，这取决于事务日志在主备数据库之间传输的时间长短，如果传输时间过长或者甚至在日志传输过程中出现异常

导致无法及时将事务应用到备库上，那么狠显然，从备库中读取的的数据将是旧的，因此就出现了不一致的情况。当然，无论是采用多次重试还是认为数据订正，关系型数据库还是能搞保证最终数据达到一致——这就是系统提供最终一致性保证的经典案例。

总的来说，BASE理论面向的是大型高可用可扩展的分布式系统，和传统事务的ACID特性使相反的，它完全不同于ACID的强一致性模型，而是提出通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。但同时，在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中，ACID特性与BASE理论往往又会结合在一起使用。

小结：

计算机系统从集中式向分布式的变革随着包括分布式网络、分布式事务和分布式数据一致性等在内的一系列问题与挑战，同时也催生了一大批诸如ACID、CAP和BASE等经典理论的快速发展。

与NoSQL的关系编辑

传统的关系型数据库在功能支持上通常很宽泛，从简单的键值查询，到复杂的多表联合查询再到事务机制的支持。而与之不同的是，NoSQL系统通常注重性能和扩展性，而非事务机制（事务就是强一致性的体现）[2] 。

传统的SQL数据库的事务通常都是支持ACID的强事务机制。A代表原子性，即在事务中执行多个操作是原子性的，要么事务中的操作全部执行，要么一个都不执行;C代表一致性，即保证进行事务的过程中整个数据加的状态是一致的，不会出现数据花掉的情况;I代表隔离性，即两个事务不会相互影响，覆盖彼此数据等;D表示持久化，即事务一量完成，那么数据应该是被写到安全的，持久化存储的设备上（比如磁盘）。

NoSQL系统仅提供对行级别的原子性保证，也就是说同时对同一个Key下的数据进行的两个操作，在实际执行的时候是会串行的执行，保证了每一个Key-Value对不会被破坏。

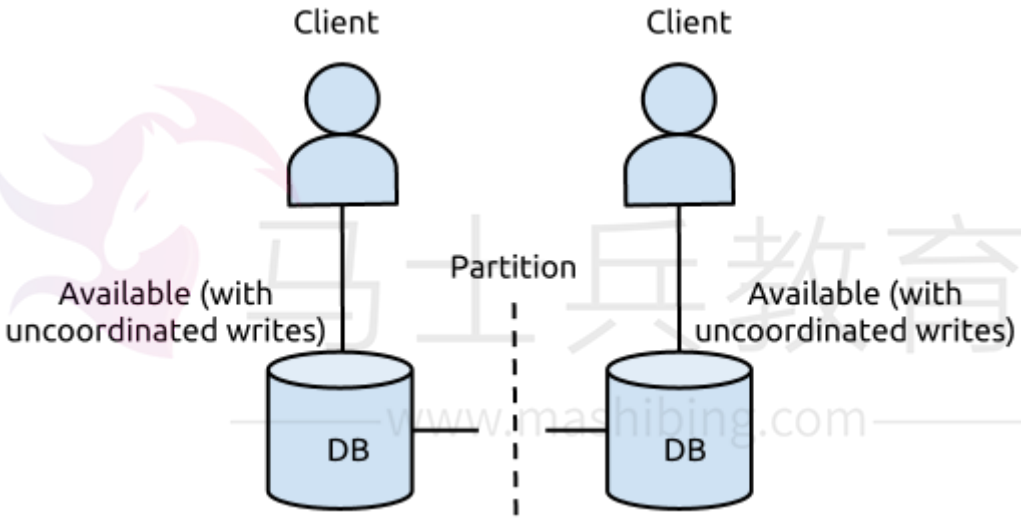
CAP的是什么关系

It states, that though its desirable to have Consistency, High-Availability and Partition-tolerance in every system, unfortunately no system can achieve all three at the same time.
在分布式系统的设计中，没有一种设计可以同时满足一致性，可用性，分区容错性 3个特性

注意：不要将弱一致性，最终一致性放到CAP理论里混为一谈（混淆概念的坑真多）
弱一致性，最终一致性 你可以认为和CAP的C一点关系也没有，因为CAP的C是更新操作完成后，任何节点看到的数据完全一致, 弱一致性。最终一致性本身和CAP的C一致性是违背的，所以你可以看到那些谎称自己系统同时具备CAP 3个特性是多么的可笑，可能国内更多的场景是：**一个开放人员一旦走上讲台演讲，就立马转变为了营销人员，连最基本的理念也不要了。**
这里有一篇标题很大的文章 cap-twelve-years-later-how-the-rules-have-changed ，实际上本文的changed更多的是在思考方式上，而本身CAP理论是没有changed的

为什么会是这样

我们来看一个简单的问题, 一个DB服务 搭建在两个机房（北京,广州），两个DB实例同时提供写入和读取



1. **假设DB的更新操作是同时写北京和广州的DB都成功才返回成功**
   在没有出现网络故障的时候，满足CA原则，C 即我的任何一个写入，更新操作成功并返回客户端完成后,分布式的所有节点在同一时间的数据完全一致， A 即我的读写操作都能够成功，但是当出现网络故障时，我不能同时保证CA，即P条件无法满足
2. **假设DB的更新操作是只写本地机房成功就返回，通过binlog/oplog回放方式同步至侧边机房**
   这种操作保证了在出现网络故障时,双边机房都是可以提供服务的，且读写操作都能成功，意味着他满足了AP ，但是它不满足C，因为更新操作返回成功后，双边机房的DB看到的数据会存在短暂不一致，且在网络故障时，不一致的时间差会很大（仅能保证最终一致性）
3. **假设DB的更新操作是同时写北京和广州的DB都成功才返回成功且网络故障时提供降级服务**
   降级服务，如停止写入，只提供读取功能，这样能保证数据是一致的，且网络故障时能提供服务，满足CP原则，但是他无法满足可用性原则

选择权衡

通过上面的例子，我们得知，我们永远无法同时得到CAP这3个特性，那么我们怎么来权衡选择呢?
选择的关键点取决于业务场景

对于大多数互联网应用来说（如网易门户），因为机器数量庞大，部署节点分散，网络故障是常态，可用性是必须需要保证的，所以只有设置一致性来保证服务的AP，通常常见的高可用服务吹嘘5个9 6个9服务SLA稳定性就本都是放弃C选择AP

对于需要确保强一致性的场景，如银行，通常会权衡CA和CP模型，CA模型网络故障时完全不可用，CP模型具备部分可用性，实际的选择需要通过业务场景来权衡（并不是所有情况CP都好于CA，只能查看信息不能更新信息有时候从产品层面还不如直接拒绝服务）

延伸

BASE(Basically Available, Soft State, Eventual Consistency 基本可用、软状态、最终一致性) 对CAP AP理论的延伸, Redis等众多系统构建与这个理论之上
ACID 传统数据库常用的设计理念, ACID和BASE代表了两种截然相反的设计哲学，分处一致性-可用性分布图谱的两极。

分布式系统的典型应用

分布式系统是一个非常广泛的概念，它最终要落实到解决实际问题上，不同的问题有不同的方法和架构。所有的开源软件都是以某个应用场景出现，而纯粹以“分布式”概念进行划分的比较少见。
但如果以算法划分，到能分出几类：
1.以Leader选举为主的一类算法，比如paxos、viewstamp，就是现在zookeeper、Chuby等工具的主体
2.以分布式事务为主的一类主要是二段提交，这些分布式数据库管理器及数据库都支持

3.以若一致性为主的，主要代表是Cassandra的W、R、N可调节的一致性
4.以租赁机制为主的，主要是一些分布式锁的概念，目前还没有看到纯粹“分布式”锁的实现
5.以失败探测为主的，主要是Gossip和phi失败探测算法，当然也包括简单的心跳
6.以弱一致性、因果一致性、顺序一致性为主的，开源尚不多，但大都应用在Linkedin、Twitter、Facebook等公司内部
7当然以异步解耦为主的，还有各类Queue

分布式事务、分布式锁

常用的分布式事务解决方案介绍有多少种?

关于分布式事务，工程领域主要讨论的是强一致性和最终一致性的解决方案。典型方案包括：

1. 两阶段提交（2PC, Two-phase Commit）方案
2. eBay 事件队列方案
3. TCC 补偿模式
4. 缓存数据最终一致性

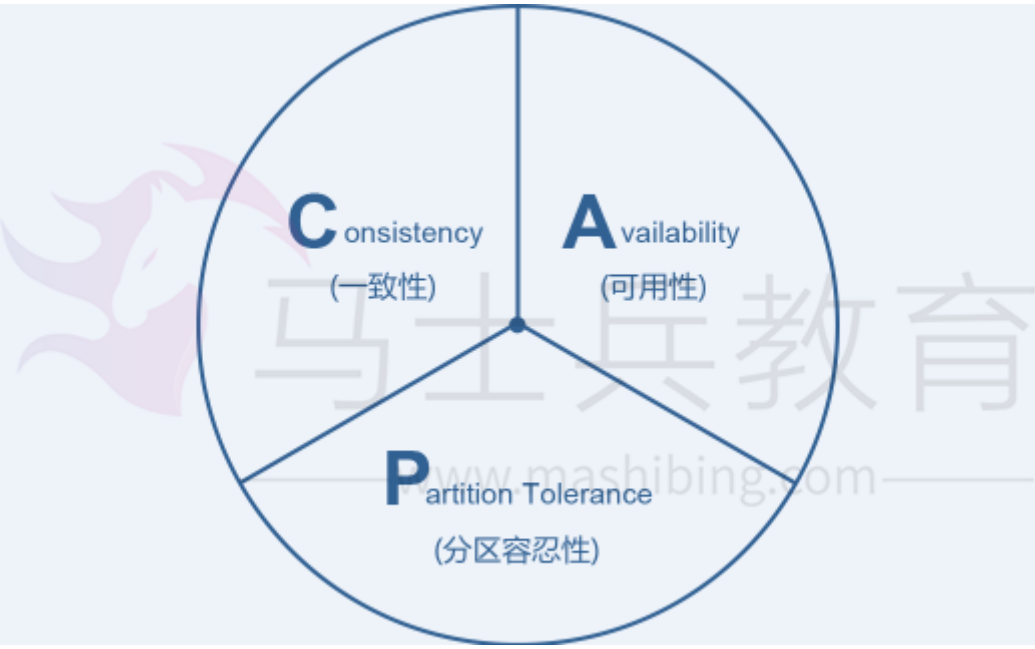
一、一致性理论

分布式事务的目的是保障分库数据一致性，而跨库事务会遇到各种不可控制的问题，如个别节点永久性宕机，像单机事务一样的ACID是无法奢望的。另外，业界著名的CAP理论也告诉我们，对分布式系统，需要将数据一致性和系统可用性、分区容忍性放在天平上一起考虑。

两阶段提交协议（简称2PC）是实现分布式事务较为经典的方案，但2PC 的可扩展性很差，在分布式架构下应用代价较大，eBay 架构师Dan Pritchett 提出了BASE 理论，用于解决大规模分布式系统下的数据一致性问题。BASE 理论告诉我们：可以通过放弃系统在每个时刻的强一致性来换取系统的可扩展性。

1、CAP理论

在分布式系统中，一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）3 个要素最多只能同时满足两个，不可兼得。其中，分区容忍性又是不可或缺的。



- 一致性：分布式环境下多个节点的数据是否强一致。
- 可用性：分布式服务能一直保证可用状态。当用户发出一个请求后，服务能在有限时间内返回结果。
- 分区容忍性：特指对网络分区的容忍性。

举例：Cassandra、Dynamo 等，默认优先选择AP，弱化C；HBase、MongoDB 等，默认优先选择CP，弱化A。

2、BASE 理论

核心思想：

- **基本可用（Basically Available）**：指分布式系统在出现故障时，允许损失部分的可用性来保证核心可用。
- **软状态（Soft State）**：指允许分布式系统存在中间状态，该中间状态不会影响到系统的整体可用性。
- **最终一致性（Eventual Consistency）**：指分布式系统中的所有副本数据经过一定时间后，最终能够达到一致的状态。

二、一致性模型

数据的一致性模型可以分成以下 3 类：

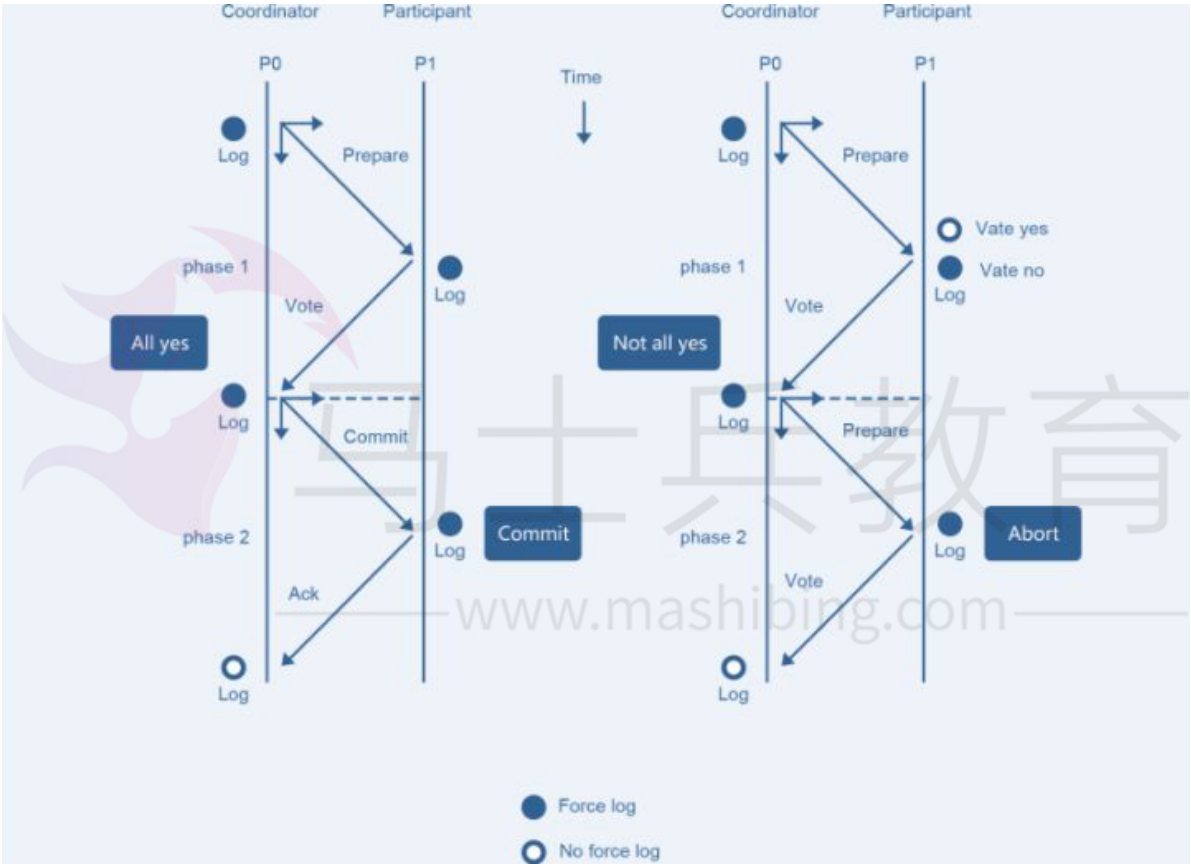
1. **强一致性**：数据更新成功后，任意时刻所有副本中的数据都是一致的，一般采用同步的方式实现。
2. **弱一致性**：数据更新成功后，系统不承诺立即可以读到最新写入的值，也不承诺具体多久之后可以读到。
3. **最终一致性**：弱一致性的一种形式，数据更新成功后，系统不承诺立即可以返回最新写入的值，但是保证最终会返回上一次更新操作的值。

分布式系统数据的强一致性、弱一致性和最终一致性可以通过Quorum NRW算法分析。

三、分布式事务解决方案

1、2PC方案——强一致性

2PC的核心原理是通过提交分阶段和记日志的方式，记录下事务提交所处的阶段状态，在组件宕机重启后，可通过日志恢复事务提交的阶段状态，并在这个状态节点重试，如Coordinator重启后，通过日志可以确定提交处于Prepare还是PrepareAll状态，若是前者，说明有节点可能没有Prepare成功，或所有节点Prepare成功但还没有下发Commit，状态恢复后给所有节点下发RollBack；若是PrepareAll状态，需要给所有节点下发Commit，数据库节点需要保证Commit幂等。



2PC方案的问题：

1. 同步阻塞。
2. 数据不一致。
3. 单点问题。

升级的3PC方案旨在解决这些问题，主要有两个改进：

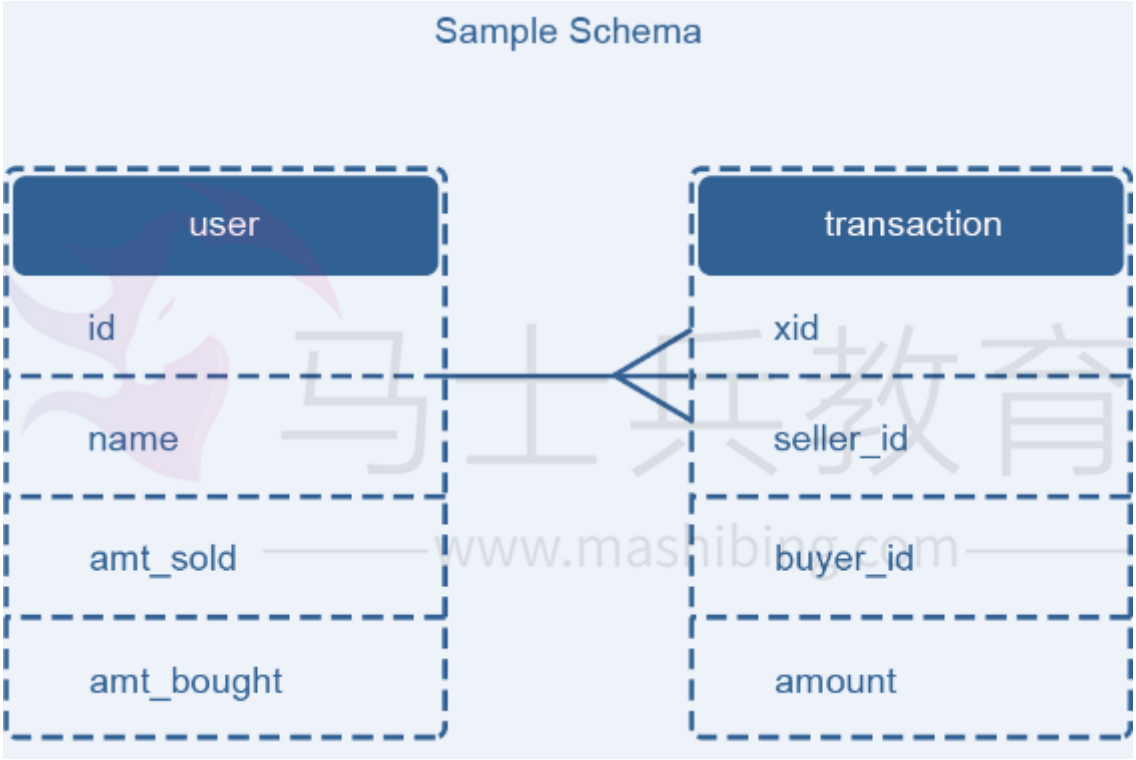
1. 增加超时机制。
2. 两阶段之间插入准备阶段。

但三阶段提交也存在一些缺陷，要彻底从协议层面避免数据不一致，可以采用Paxos或者Raft 算法。

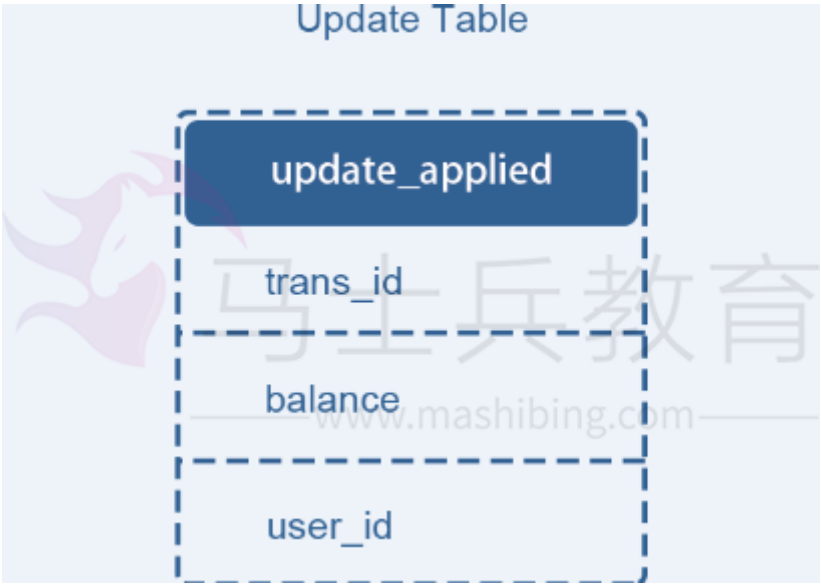
2、eBay 事件队列方案——最终一致性

eBay 的架构师Dan Pritchett，曾在一篇解释BASE 原理的论文《Base：An Acid Alternative》中提到一个eBay 分布式系统一致性问题的解决方案。它的核心思想是将需要分布式处理的任务通过消息或者日志的方式来异步执行，消息或日志可以存到本地文件、数据库或消息队列，再通过业务规则进行失败重试，它要求各服务的接口是幂等的。

描述的场景为，有用户表user 和交易表transaction，用户表存储用户信息、总销售额和总购买额，交易表存储每一笔交易的流水号、买家信息、卖家信息和交易金额。如果产生了一笔交易，需要在交易表增加记录，同时还要修改用户表的金额。



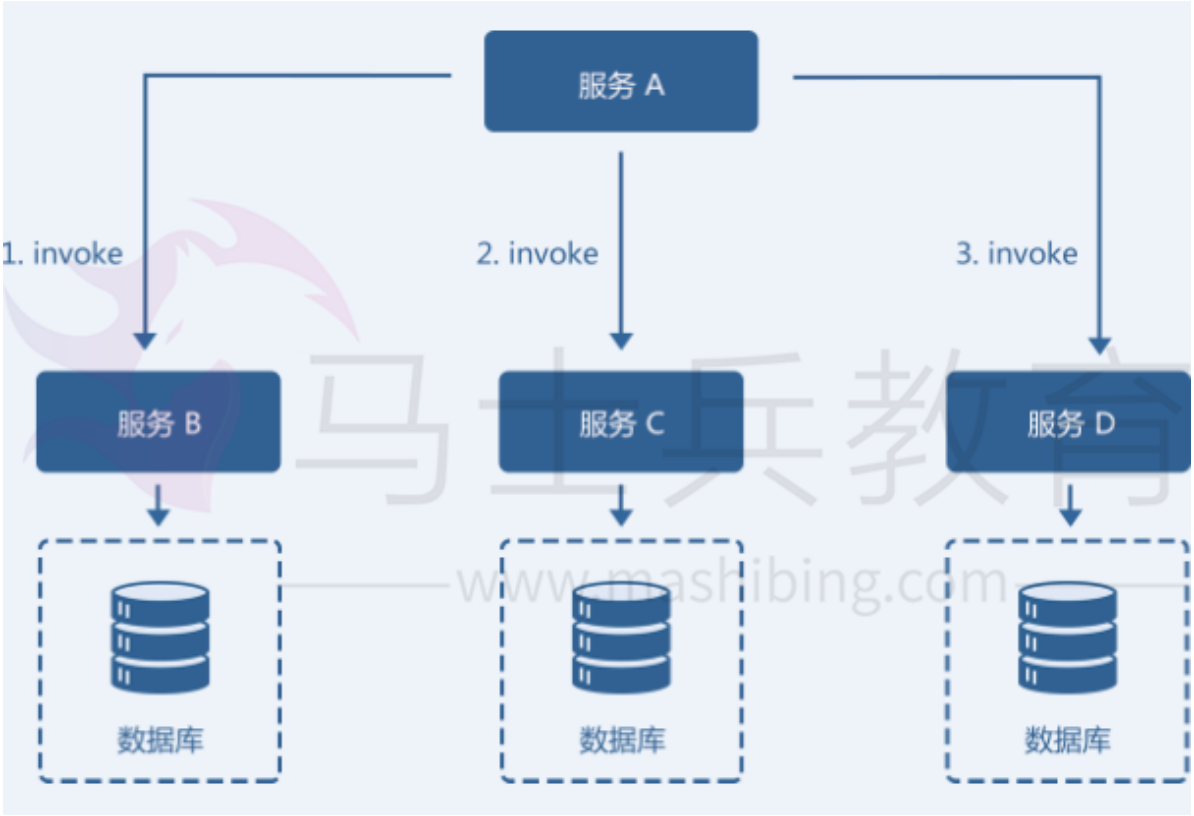
论文中提出的解决方法是将更新交易表记录和用户表更新消息放在一个本地事务来完成，为了避免重复消费用户表更新消息带来的问题，增加一个操作记录表updates_applied来记录已经完成的交易相关的信息。



这个方案的核心在于第二阶段的重试和幂等执行。失败后重试，这是一种补偿机制，它是能保证系统最终一致的关键流程。

3、TCC （Try-Confirm-Cancel）补偿模式——最终一致性

某业务模型如图，由服务 A、服务B、服务C、服务D 共同组成的一个微服务架构系统。服务A 需要依次调用服务B、服务C 和服务D 共同完成一个操作。当服务A 调用服务D 失败时，若要保证整个系统数据的一致性，就要对服务B 和服务C 的invoke 操作进行回滚，执行反向的revert 操作。回滚成功后，整个微服务系统是数据一致的。



实现关键要素：

1. 服务调用链必须被记录下来。
2. 每个服务提供者都需要提供一组业务逻辑相反的操作，互为补偿，同时回滚操作要保证幂等。
3. 必须按失败原因执行不同的回滚策略。

4、缓存数据最终一致性

在我们的业务系统中，缓存（Redis 或者Memcached）通常被用在数据库前面，作为数据读取的缓冲，使得I/O 操作不至于直接落在数据库上。以商品详情页为例，假如卖家修改了商品信息，并写回到数据库，但是这时候用户从商品详情页看到的信息还是从缓存中拿到的过时数据，这就出现了缓存系统和数据库系统中的数据不一致的现象。

要解决该场景下缓存和数据库数据不一致的问题我们有以下两种解决方案：

1. 为缓存数据设置过期时间。当缓存中数据过期后，业务系统会从数据库中获取数据，并将新值放入缓存。这个过期时间就是系统可以达到最终一致的容忍时间。
2. 更新数据库数据后同时清除缓存数据。数据库数据更新后，同步删除缓存中数据，使得下次对商品详情的获取直接从数据库中获取，并同步到缓存。

四、选择建议

在面临数据一致性问题的时候，首先要从业务需求的角度出发，确定我们对于3 种一致性模型的接受程度，再通过具体场景来决定解决方案。

从应用角度看，分布式事务的现实场景常常无法规避，在有能力给出其他解决方案前，2PC也是一个不错的选择。

对购物转账等电商和金融业务，中间件层的2PC最大问题在于业务不可见，一旦出现不可抗力或意想不到的一致性破坏，如数据节点永久性宕机，业务难以根据2PC的日志进行补偿。金融场景下，数据一致性是命根，业务需要对数据有百分之百的掌控力，建议使用TCC这类分布式事务模型，或基于消息队列的柔性事务框架，这两种方案都在业务层实现，业务开发者具有足够掌控力，可以结合SOA框架来架构，包括Dubbo、Spring Cloud等（题主的标签写了Dubbo）。

分布式锁的几种实现方式

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。
其典型的使用场景为：
不同系统或者是同一系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，需要通过一定的**互斥**手段来防止彼此的干扰，以保证一致性。

1. 使用Redis实现分布式锁

* WATCH, MULTI, EXEC, DISCARD事务机制实现分布式锁

Redis支持基本的事务操作

```
MULTI
some redis command
EXEC
```

以上被MULTI和EXEC包裹的redis命令，保证所有事务内的命令将会串行顺序执行，保证不会在事务的执行过程中被其他客户端打断。
而WATCH命令能够监视某个键，当事务执行时，如果被监视的键被其他客户端修改了值，事务运行失败，返回相应错误（被事务运行客户端在事务内修改了值，不会造成事务运行失败）。

运用Redis事务所支持的以上特性，可以实现一个分布式锁功能。Python代码如下：

```
# -*- coding: utf-8 -*-
def acqure_lock_with_watch(conn, lockname, acquire_timeout=10):
    pipe = conn.pipeline()
    end = time.time() + acquire_timeout
    lockname = 'lock:' + lockname

    while time.time() < end:
        try:
            pipe.watch(lockname)
            pipe.multi()  # 开启事务
            # 事务具体内容，对lockname的值进行更新
            pipe.execute()
            return True
        except redis.exceptions.WatchError:
            # 事务运行期间，有其他客户端改变了lockname的值，抛出异常，进行重试操作
            pass

    return False
```

通过WATCH命令监视某个键，当该键未被其他客户端修改值时，事务成功执行。当事务运行过程中，发现该值被其他客户端更新了值，任务失败，进行重试。

*** SETNX实现分布式锁**

SETNX：当指定键不存在时，向Redis中添加一个键值对。Redis客户端保证对统一键名称，多个客户端同时设置其值时，只有一个客户端能够设置成功的原子性。

```
# -*- coding: utf-8 -*-
def acquire_lock_with_timeout(
        conn, lockname, acquire_timeout=10, lock_timeout=10):
    identifire = str(uuid.uuid4())
    lockname = 'lock:' + lockname
    lock_timeout = int(math.ceil(lock_timeout))

    end = time.time() + acquire_timeout
    while time.time() < end:
        if conn.setnx(lockname, identifire):  # 以锁名称为键，uuid的值为值，redis服务器setnx保证了只能有一个客户端成功设置键的原子性
            conn.expire(lockname, lock_timeout)  # 设置键的过期时间，过期自动剔除，释放锁
            return identifire
        elif not conn.ttl(lockname):  # 当锁未被设置过期时间时，重新设置其过期时间
            conn.expire(lockname, lock_timeout)

        time.sleep(0.001)

    return False
```

以上，利用SETNX的原子特性，和Redis的键过期特性，实现了自动过期释放的分布式锁。

*** 锁的释放**

```
# -*- coding: utf-8 -*-
def release_lock(conn, lockname, identifire):
    pipe = conn.pipeline(True)
    lockname = 'lock:' + lockname

    while True:
        try:
            pipe.watch(lockname)  # 监视锁的键，在锁释放过程中改变了键的值时得到相应通知
            if pipe.get(lockname) == identifire:  # 检查客户端是否仍然持有该锁
                pipe.multi()
                pipe.delete(lockname)  # 删除键，释放锁
                pipe.execute()
                return True

            pipe.unwatch()
            break
        except redis.exceptions.WatchError:
            pass  # 释放锁期间，有其他客户端改变了键值对，锁释放失败，进行循环

    return False
```

锁释放的过程，首先检查客户端是否仍然持有该锁，如果持有，则在事务中删除键值对，释放锁的所有权。

2. 使用Memcached实现分布式锁

Memcached的add命令，当指定的key不存在时，进行添加，且保证了执行的原子性。利用该特性，可以实现一个分布式锁实现。

```
def acquire_lock_with__memcached_timeout(
        conn, lockname, acquire_timeout=10, lock_timeout=10):
    identifire = str(uuid.uuid4())
    lockname = 'lock:' + lockname
    lock_timeout = int(math.ceil(lock_timeout))

    end = time.time() + acquire_timeout
    while time.time() < end:
        # 过期时间保证了客户端崩溃时仍能在超过过期世家后正常释放锁
        # 以锁名称为键，uuid的值为值，memcached服务器add保证了只能有一个客户端成功设置键的原子性
        if conn.add(lockname, identifire, lock_timeout):
            return identifire
        time.sleep(0.001)

    return False
```

释放锁时，删除指定key的键即可。

3. 使用ZooKeeper实现分布式锁

相较于使用redis和memcached实现的锁，zk利用其高级特性，能够实现更复杂的锁特性。

- **排它锁**

排它锁，又称写锁或独占锁。如果事务T1对对象O1加了排它锁，那么整个加锁期间，只允许事务T1对O1进行读取和更新操作，其他任何事务都不能对该数据对象进行任务读写操作，直到T1释放了排它锁。
以上介绍的使用redis和memcached实现的分布式锁都属于排它锁。

获取锁

ZooKeeper实现分布式锁利用了其临时子节点的如下特性：
在/exclusive_lock节点下创建临时子节点/exclusive_lock/lock，zk会保证在所有的客户端中，最终只有一个客户端能够创建成功，即可以认为该客户端获得了锁。同时，虽有没有获取到所得客户端需要到/exclusive_lock节点上注册一个子节点变更的Watcher监听，以便实时监听到lock节点的变更情况。

释放锁

因为在获取锁时，创建的是一个临时节点/exclusive_lock/lock，因此在如下情况，都有可能释放锁：

- 获取锁的机器发生宕机，临时节点被zk移除
- 正常执行业务逻辑后，客户端主动删除临时节点

无论什么情况下，lock节点被移除，zk都会通知所有在/exclusive_lock节点上注册了子节点变更的Watcher监听的客户端。这些客户端在收到通知后，重新发起分布式锁的获取流程。

- **共享锁**

共享锁又称读锁。如果事务T1对数据对象O1加上了共享锁，那么当前事务只能对O1进行读取操作，其他事务也只能对这个数据对象加上共享锁，直到该数据对象上的所有共享锁都被释放。
而更新操作只能在当前没有任何事务进行读写操作的情况下进行。

获取锁

当需要获取共享锁是，所有客户端到/shared_lock节点下面创建一个临时顺序节点，/shared_lock/[hostname]-请求类型（W | R）-序号，该节点代编了一个共享锁。
如果是读请求，则创建/shared_lock/192.168.0.1-R-000000000001；
如果是写请求，则创建/shared_lock/192.168.0.1-W-000000000001；

判断读写顺序

1. 在创建完监听后，获取/shared_lock节点下的所有子节点，对该节点注册子节点变更的Watcher监听
2. 确定自己的节点序号在所有子节点的顺序
3. 对于读请求：

> 如果没有比自己序号小的子节点，或是所有比自己序号小的子节点都是读请求，那么表明自己已经成功获取到了共享锁，执行读取逻辑。
> 如果比自己序号小的子节点中有写请求，那么进入等待。

1. 对于写请求：

> 如果自己不是序号最小的子节点，那么进入等待。

1. 收到Watcher通知后，重复步骤1-4

释放锁

释放锁，删除对应的数据节点即可。

Spring Cloud面试题

什么是 Spring Cloud?

Spring cloud 流应用程序启动器是基于 Spring Boot 的 Spring 集成应用程序，提供与外部系统的集成。

Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。

使用 Spring Cloud 有什么优势?

使用 Spring Boot 开发分布式微服务时，我们面临以下问题

- 与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。
- 服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
- 冗余-分布式系统中的冗余问题。
- 负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。
- 性能-问题 由于各种运营开销导致的性能问题。

- 部署复杂性-Devops 技能的要求。

服务注册和发现是什么意思？Spring Cloud 如何实现？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。 Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。

负载平衡的意义什么？

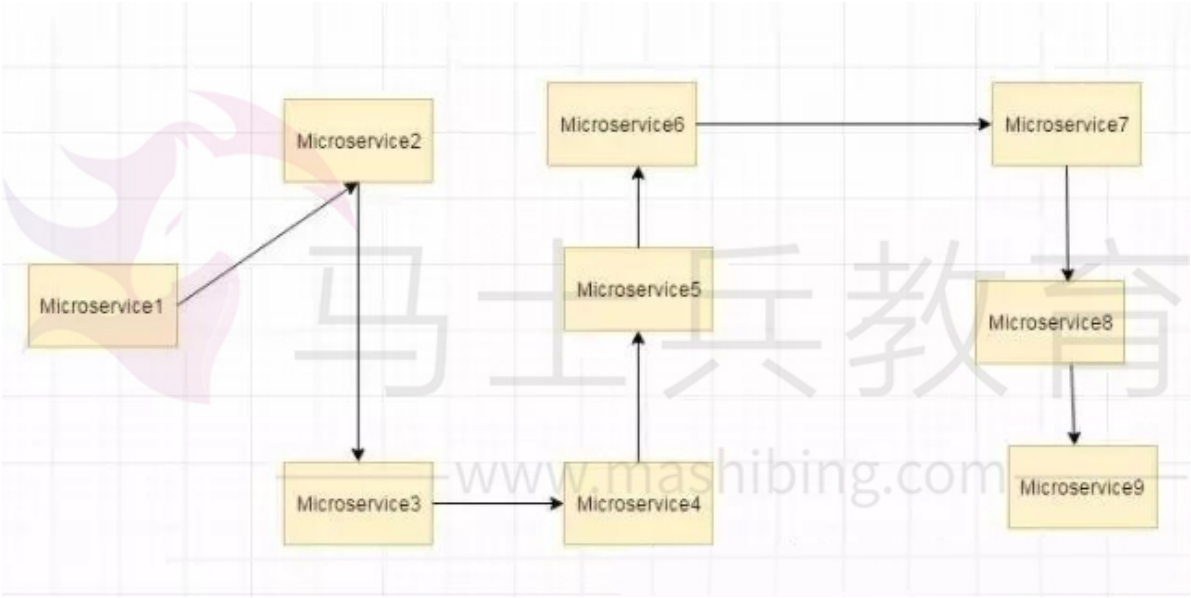
在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

什么是 Hystrix？它如何实现容错？

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

思考以下微服务



假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000.这是 hystrix 出现的地方我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-consumer 公开的服务。

简化图如下所示

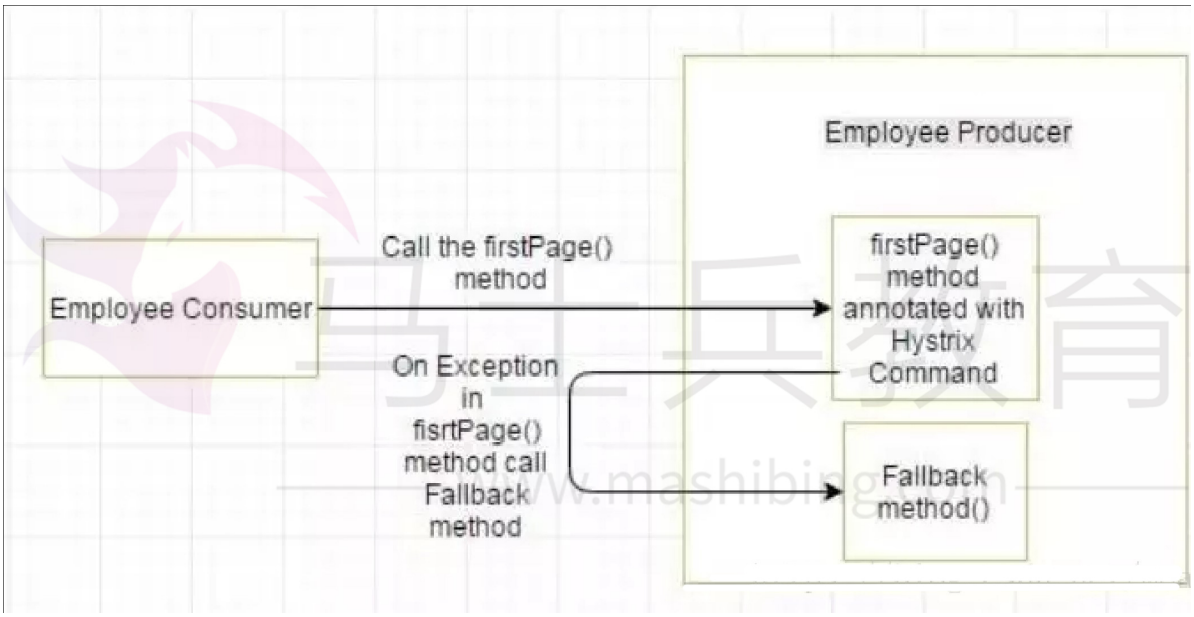


现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix

定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

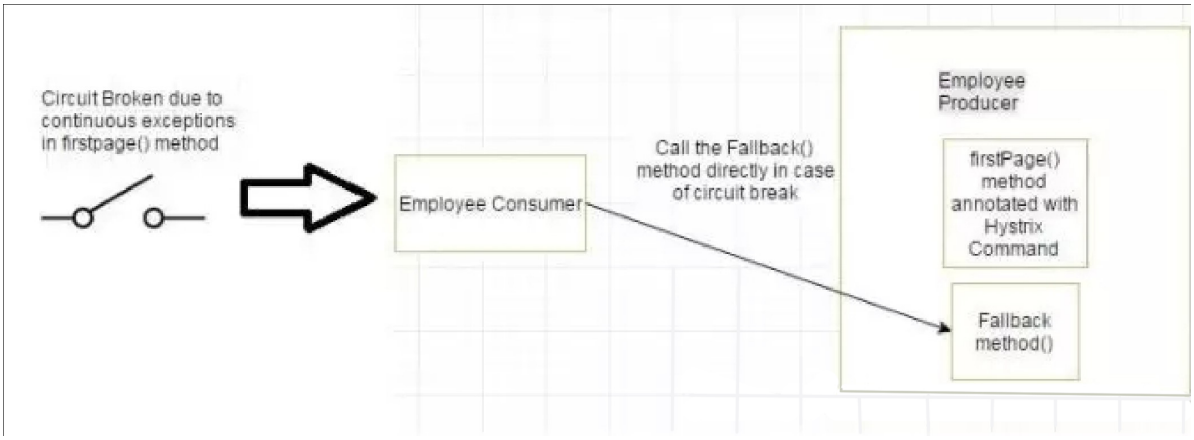
什么是Hystrix 断路器？我们需要它吗？

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用 Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。



如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过

firtsPage 方法，并直接调用回退方法。 断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会 。



什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤

· 使用功能区进行负载平衡。

· 获取服务实例，然后获取基本 URL。

· 利用 REST 模板来使用服务。 前面的代码如下

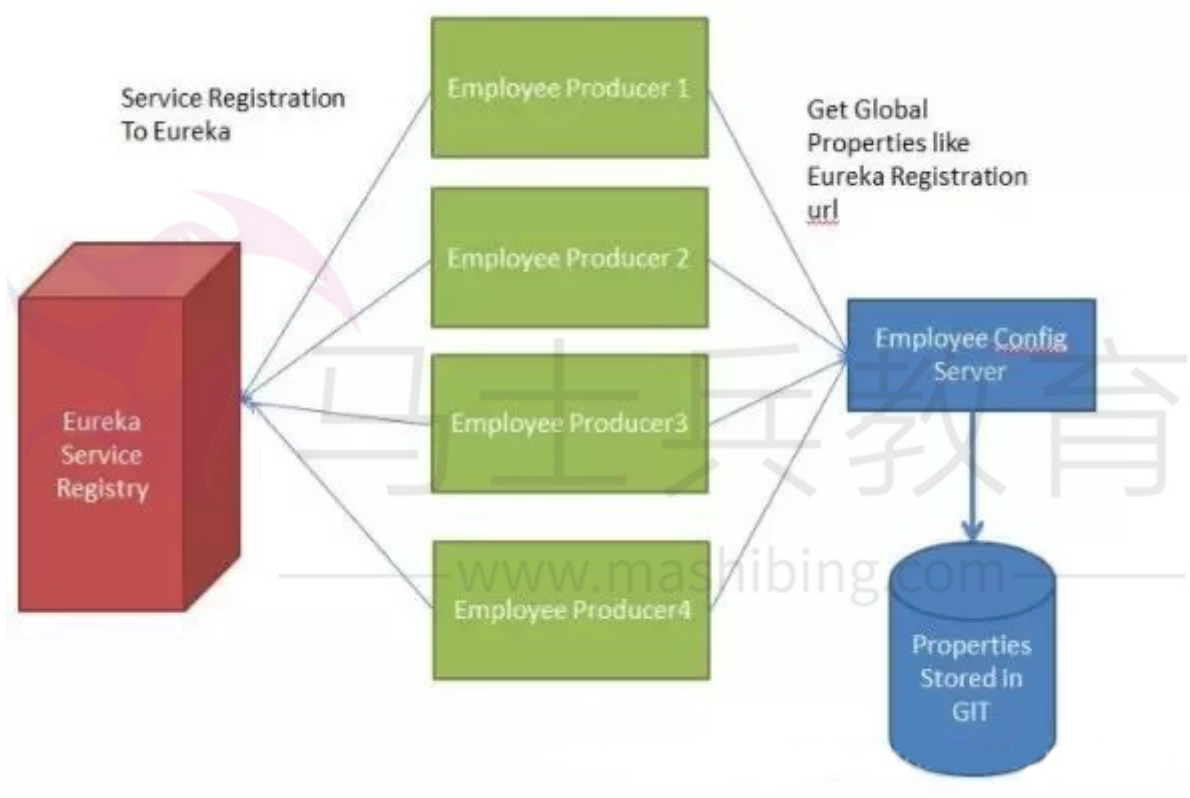
```
1.  @Controller
2.  public class ConsumerControllerClient { 3.
4.  @Autowired
5.  private LoadBalancerClient loadBalancer; 6.
7.  public void getEmployee() throws RestClientException, IOException {
8.
9.  ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");
10.
11.System.out.println(serviceInstance.getUri());
12.
13.String baseUrl=serviceInstance.getUri().toString();
14.
15.baseUrl=baseUrl+"/employee";
16.
17.RestTemplate restTemplate = new RestTemplate();
18. ResponseEntity<String> response=null;
19. try{
20. response=restTemplate.exchange(baseUrl,
21. HttpMethod.GET, getHeaders(),String.class);
22. }catch (Exception ex)
23. {
24. System.out.println(ex);
25. }
26. System.out.println(response.getBody());
27. }
```

之前的代码，有像 NullPointer 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

什么是 Spring Cloud Bus？我们需要它吗？

考虑以下情况：我们有多个应用程序使用 Spring Cloud Config 读取属性，而 Spring Cloud Config 从 GIT 读取这些属性。

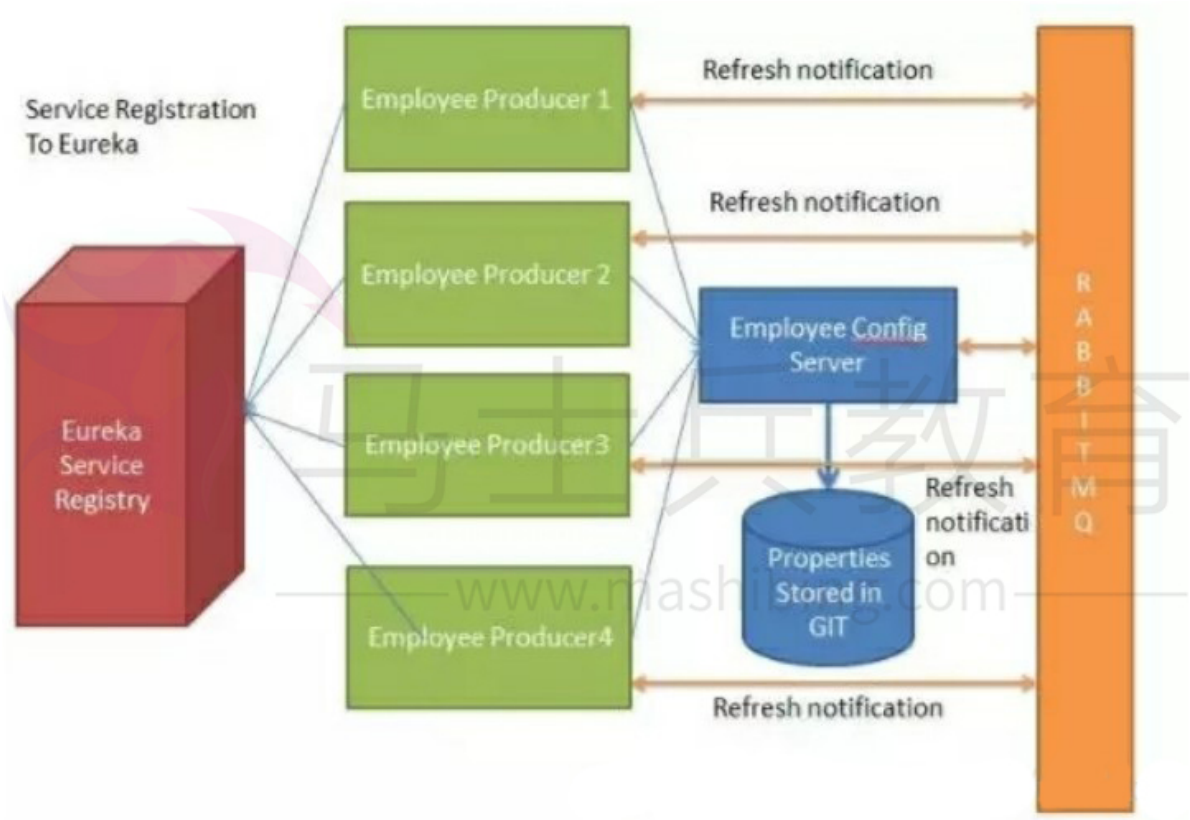
下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。



如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果

Employee Producer1 部署在端口 8080 上，则调用 http：// localhost：8080 / refresh。同样对于 Employee Producer2 http：// localhost：8081 / refresh 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。



Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

为什么需要学习Spring Cloud

不论是商业应用还是用户应用，在业务初期都很简单，我们通常会把它实现为单体结构的应用。但是，随着业务逐渐发展，产品思想会变得越来越复杂，单体结构的应用也会越来越复杂。这就会给应用带来如下的几个问题：

代码结构混乱：业务复杂，导致代码量很大，管理会越来越困难。同时，这也会给业务的快速迭代带来巨大挑战；

开发效率变低：开发人员同时开发一套代码，很难避免代码冲突。开发过程会伴随着不断解决冲突的过程，这会严重的影响开发效率；

排查解决问题成本高：线上业务发现 bug，修复 bug 的过程可能很简单。但是，由于只有一套代码，需要重新编译、打包、上线，成本很高。

由于单体结构的应用随着系统复杂度的增高，会暴露出各种各样的问题。近些年来，微服务架构逐渐取代了单体架构，且这种趋势将会越来越流行。Spring

Cloud是目前最常用的微服务开发框架，已经在企业级开发中大量的应用。

什么是Spring Cloud

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、智能路由、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

设计目标与优缺点

设计目标

协调各个微服务，简化分布式系统开发。

优缺点

微服务的框架那么多比如：dubbo、Kubernetes，为什么就要使用Spring

Cloud的呢?

优点：

- 产出于Spring大家族，Spring在企业级开发框架中无人能敌，来头很大，可以保证后续的更新、完善
- 组件丰富，功能齐全。Spring Cloud 为微服务架构提供了非常完整的支持。例如、配置管理、服务发现、断路器、微服务网关等；

- Spring Cloud 社区活跃度很高，教程很丰富，遇到问题很容易找到解决方案服务拆分粒度更细，耦合度比较低，有利于资源重复利用，有利于提高开发效率可以更精准的制定优化服务方案，提高系统的可维护性减轻团队的成本，可以并行开发，不用关注其他人怎么开发，先关注自己的开发微服务可以是跨平台的，可以用任何一种语言开发适于互联网时代，产品迭代周期更短

缺点：

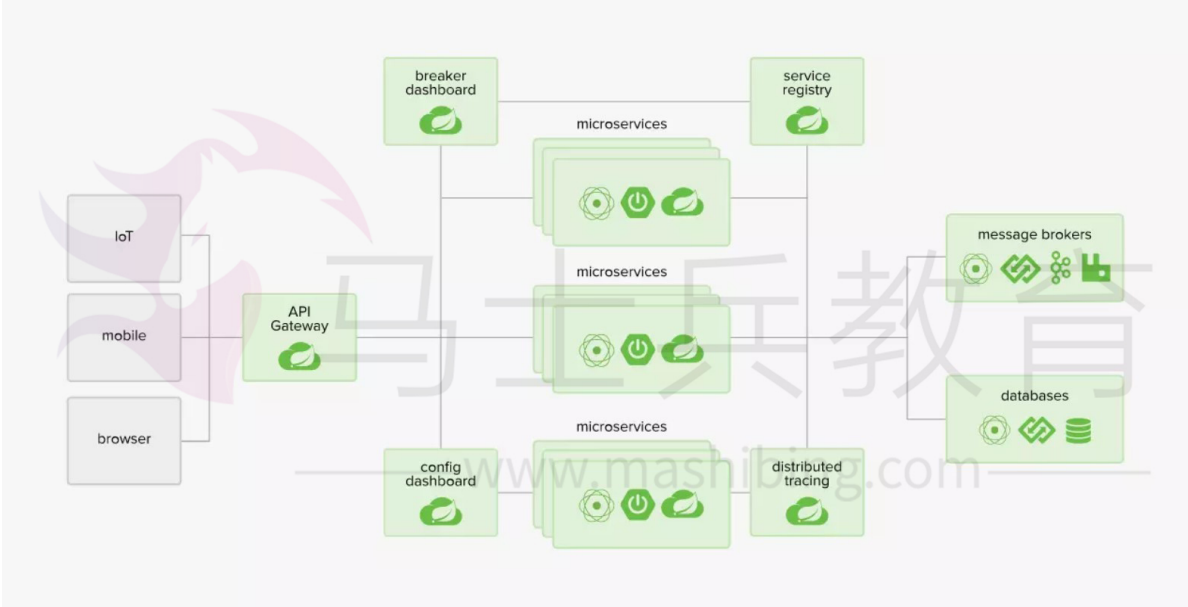
- 微服务过多，治理成本高，不利于维护系统
- 分布式系统开发的成本高（容错，分布式事务等）对团队挑战大总的来说优点大过于缺点，目前看来Spring Cloud是一套非常完善的分布式框架，目前很多企业开始用微服务、Spring Cloud的优势是显而易见的。因此对于想研究微服务架构的同学来说，学习Spring Cloud是一个不错的选择。

Spring Cloud发展前景

Spring Cloud对于中小型互联网公司来说是一种福音，因为这类公司往往没有实力或者没有足够的资金投入去开发自己的分布式系统基础设施，使用Spring

Cloud一站式解决方案能在从容应对业务发展的同时大大减少开发成本。同时，随着近几年微服务架构和Docker容器概念的火爆，也会让Spring Cloud在未来越来越“云”化的软件开发风格中立有一席之地，尤其是在五花八门的分布式解决方案中提供了标准化的、全站式的技术方案，意义可能会堪比当年Servlet规范的诞生，有效推进服务端软件系统技术水平的进步。

整体架构



主要项目

Spring Cloud的子项目，大致可分成两类，一类是对现有成熟框架"Spring

Boot化"的封装和抽象，也是数量最多的项目；第二类是开发了一部分分布式系统的基础设施的实现，如Spring Cloud Stream扮演的就是kafka, ActiveMQ这样的角色。

Spring Cloud Config

集中配置管理工具，分布式系统中统一的外部配置管理，默认使用Git来存储配置，可以支持客户端配置的刷新及加密、解密操作。

Spring Cloud Netflix

Netflix OSS 开源组件集成，包括Eureka、Hystrix、Ribbon、Feign、Zuul等核心组件。

Eureka：服务治理组件，包括服务端的注册中心和客户端的服务发现机制；

Ribbon：负载均衡的服务调用组件，具有多种负载均衡调用策略；

Hystrix：服务容错组件，实现了断路器模式，为依赖服务的出错和延迟提供了容错能力；

Feign：基于Ribbon和Hystrix的声明式服务调用组件；

Zuul：API网关组件，对请求提供路由及过滤功能。

Spring Cloud Bus

用于传播集群状态变化的消息总线，使用轻量级消息代理链接分布式系统中的节点，可以用来动态刷新集群中的服务配置。

Spring Cloud Consul

基于Hashicorp Consul的服务治理组件。

Spring Cloud Security

安全工具包，对Zuul代理中的负载均衡OAuth2客户端及登录认证进行支持。

Spring Cloud Sleuth

Spring Cloud应用程序的分布式请求链路跟踪，支持使用Zipkin、HTrace和基于日志（例如ELK）的跟踪。

Spring Cloud Stream

轻量级事件驱动微服务框架，可以使用简单的声明式模型来发送及接收消息，主要实现为Apache Kafka及RabbitMQ。

Spring Cloud Task

用于快速构建短暂、有限数据处理任务的微服务框架，用于向应用中添加功能性和非功能性的特性。

Spring Cloud Zookeeper

基于Apache Zookeeper的服务治理组件。

Spring Cloud Gateway

API网关组件，对请求提供路由及过滤功能。

Spring Cloud OpenFeign

基于Ribbon和Hystrix的声明式服务调用组件，可以动态创建基于Spring MVC 注解的接口实现用于服务调用，在Spring Cloud 2.0中已经取代Feign成为了一等公民。

Spring Cloud的版本关系

Spring Cloud是一个由许多子项目组成的综合项目，各子项目有不同的发布节奏。 为了管理Spring Cloud与各子项目的版本依赖关系，发布了一个清单，其中包括了某个Spring Cloud版本对应的子项目版本。 为了避免Spring Cloud版本号与子项目版本号混淆，Spring Cloud版本采用了名称而非版本号的命名，

这些版本的名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序，例如Angel是第一个版本，Brixton是第二个版本。 当Spring Cloud的发布

内容积累到临界点或者一个重大BUG被解决后，会发布一个"service

releases"版本，简称SRX版本，比如Greenwich.SR2就是Spring Cloud发布的 Greenwich版本的第2个SRX版本。目前Spring Cloud的最新版本是Hoxton。

Spring Cloud和SpringBoot版本对应关系

Spring Cloud Version	SpringBo ot Version
Hoxton	2.2.x
Greenwic h	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

Spring Cloud和各子项目版本对应关系

More Actions Compon ent	Edgware. SR6	Greenwic h.SR2
spring- cloud- bus	1.3.4.RELE ASE	2.1.2.RELE ASE
spring- cloud- commons	1.3.6.RELE ASE	2.1.2.RELE ASE
spring- cloud- config	1.4.7.RELE ASE	2.1.3.RELE ASE
spring- cloud- netflix	1.4.7.RELE ASE	2.1.2.RELE ASE
spring- cloud- security	1.2.4.RELE ASE	2.1.3.RELE ASE
spring- cloud- consul	1.3.6.RELE ASE	2.1.2.RELE ASE
spring- cloud- sleuth	1.3.6.RELE ASE	2.1.1.RELE ASE
spring- cloud- stream	Ditmars.S R5	Fishtown. SR3
spring- cloud- zookeepe r	1.2.3.RELE ASE	2.1.2.RELE ASE
spring- boot	1.5.21.REL EASE	2.1.5.RELE ASE
spring- cloud- task	1.2.4.RELE ASE	2.1.2.RELE ASE
spring- cloud- gateway	1.0.3.RELE ASE	2.1.2.RELE ASE
spring- cloud- openfeig n	暂无	2.1.2.RELE ASE

注意：Hoxton版本是基于SpringBoot 2.2.x版本构建的，不适用于1.5.x版本。随着2019年8月SpringBoot 1.5.x版本停止维护，Edgware版本也将停止维护。

SpringBoot和SpringCloud的区别?

SpringBoot专注于快速方便的开发单个个体微服务。

SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务

SpringBoot可以离开SpringCloud独立使用开发项目， 但是SpringCloud离不开SpringBoot ，属于依赖的关系

SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

使用 Spring Boot 开发分布式微服务时，我们面临以下问题

（1） 与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。

（2） 服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。

（3） 冗余-分布式系统中的冗余问题。

（4） 负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。

（5） 性能-问题 由于各种运营开销导致的性能问题。

（6）部署复杂性-Devops 技能的要求。

服务注册和发现是什么意思？Spring Cloud 如何实现?

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。 Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。

Spring Cloud 和dubbo区别?

（1）服务调用方式 dubbo是RPC springcloud Rest Api （2）注册中心,dubbo 是zookeeper springcloud是eureka，也可以是 zookeeper

（3）服务网关,dubbo本身没有实现，只能通过其他第三方技术整合， springcloud有Zuul路由网关，作为路由服务器，进行消费者的请求分

发,springcloud支持断路器，与git完美集成配置文件支持版本控制，事物总线实现配置文件的更新与服务自动装配等等一系列的微服务架构要素。

负载平衡的意义什么?

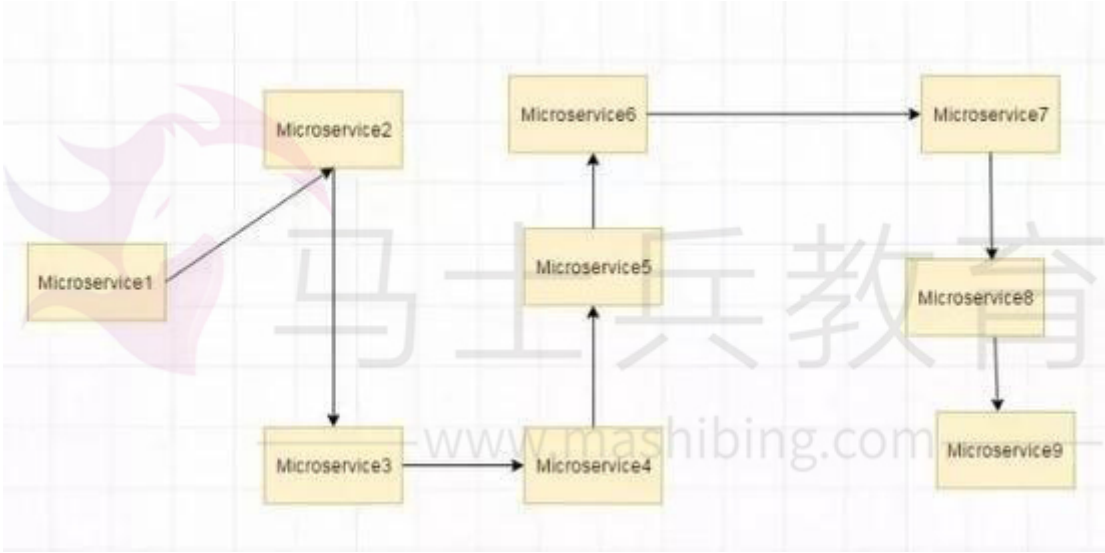
在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

什么是 Hystrix？它如何实现容错?

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

思考以下微服务



假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。

但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000.

这是 hystrix 出现的地方 我们将使用 Hystrix 在这种情况下的 Fallback 方法功

能。我们有两个服务 employee-consumer 使用由 employee-consumer 公开的服务。

简化图如下所示

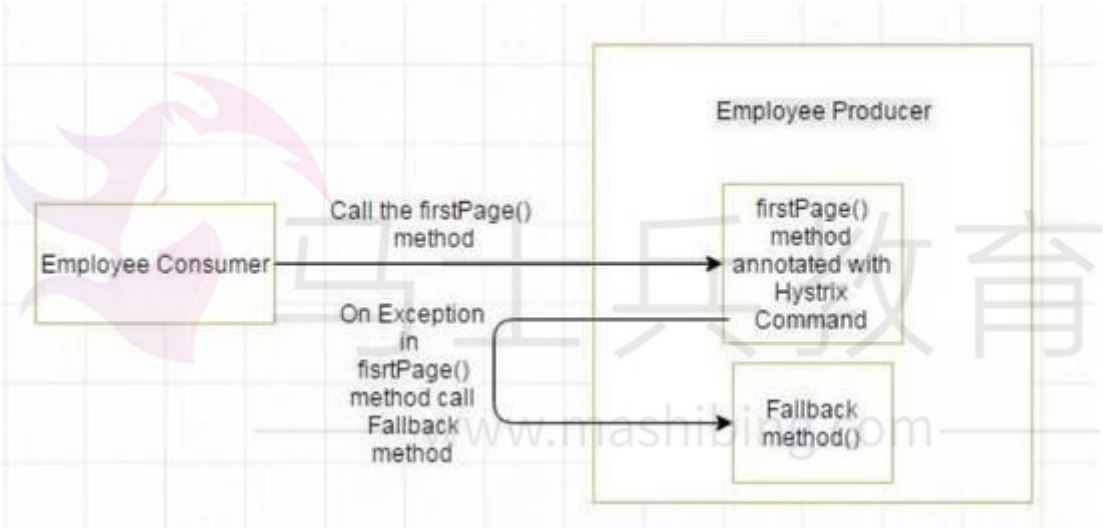


现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服

务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

什么是 Hystrix 断路器？我们需要它吗？

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。



如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员

工使用者将一起跳过 firtsPage 方法，并直接调用回退方法。 断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会 。

什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。

Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。

在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤（1）使用功能区进行负载平衡。

（2） 获取服务实例，然后获取基本 URL。

（3） 利用 REST 模板来使用服务。 前面的代码如下

```
@Controller
public class ConsumerControllerClient {
@Autowired
private LoadBalancerClient loadBalancer;
public void getEmployee() throws RestClientException, IOException {
 ServiceInstance serviceInstance=loadBalancer.choose("employee- producer");
 System.out.println(serviceInstance.getUri());
 String baseUrl=serviceInstance.getUri().toString();
 baseUrl=baseUrl+"/employee";
  RestTemplate restTemplate = new RestTemplate();
  ResponseEntity<String> response=null;
  try{
  response=restTemplate.exchange(baseUrl,
  HttpMethod.GET, getHeaders(),String.class);
  }
  catch (Exception ex)
  {
  System.out.println(ex);
  }
  System.out.println(response.getBody());
}
```

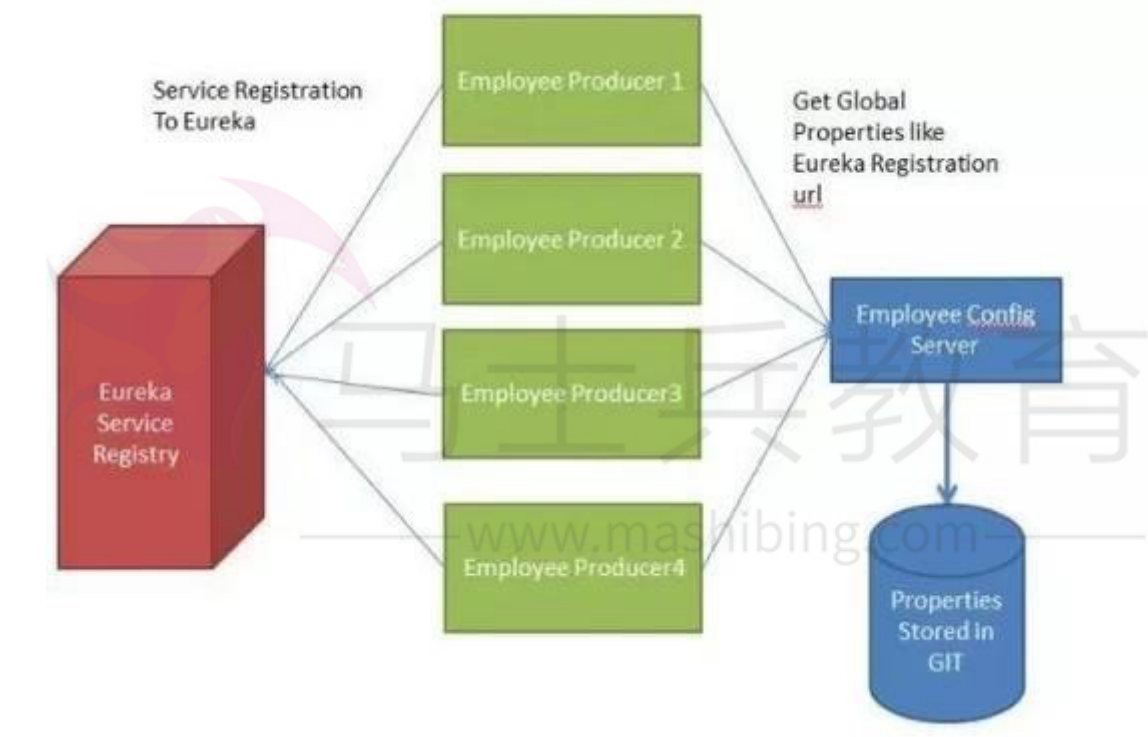
之前的代码，有像 NullPointer 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

什么是 Spring Cloud Bus？我们需要它吗？

考虑以下情况：我们有多个应用程序使用 Spring Cloud Config 读取属性，而

Spring Cloud Config 从 GIT 读取这些属性。

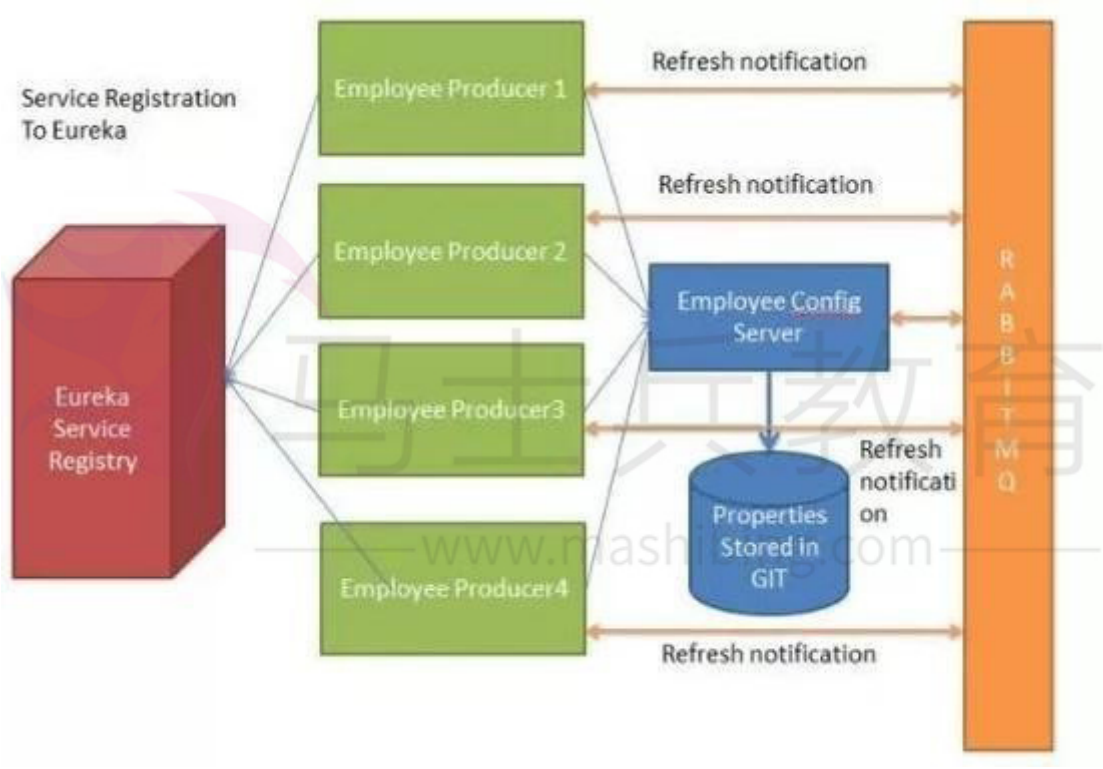
下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。



如果假设 GIT 中的 Eureka 注册属性更改(img)为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调

用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 http：// localhost：8080 / refresh。同样对于 Employee Producer2

http：//localhost：8081 / refresh 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。



Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模

块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

Spring Cloud断路器的作用

当一个服务调用另一个服务由于网络原因或自身原因出现问题，调用者就会等待被调用者的响应 当更多的服务请求到这些资源导致更多的请求等待，发生连锁效应（雪崩效应）断路器有完全打开状态:一段时间内 达到一定的次数无法调用 并且多次监测没有恢复的迹象 断路器完全打开 那么下次请求就不会请求到该服务半开:短时间内 有恢复迹象 断路器会将部分请求发给该服务，正常调用时 断路器关闭

关闭：当服务一直处于正常状态 能正常调用

什么是Spring Cloud Config?

在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在Spring Cloud中，有分布式配置中心组件spring cloud config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。在spring cloud config 组件中，分两个角色，一是 config server，二是config client。

使用：

（1） 添加pom依赖

（2） 配置文件添加相关配置

（3） 启动类添加注解@EnableConfigServer

什么是Spring Cloud Gateway?

Spring Cloud Gateway是Spring Cloud官方推出的第二代网关框架，取代Zuul 网关。网关作为流量的，在微服务系统中有着非常作用，网关常见的功能有路由转发、权限校验、限流控制等作用。

使用了一个RouteLocatorBuilder的bean去创建路由，除了创建路由

RouteLocatorBuilder可以让你添加各种predicates和filters，predicates断言的意思，顾名思义就是根据具体的请求的规则，由具体的route去处理，filters 是各种过滤器，用来对请求做各种判断和修改。

一. 分布式模型

1. 节点

在具体的工程项目中，一个节点往往是一个操作系统上的进程。在本文的模型中，认 为节点是一个完整的、不可分的整体，如果某个程序进程实际上由若干相对独立部分 构成，则在模型中可以将一个进程划分为多个节点。

异常

机器宕机：机器宕机是最常见的异常之一。在大型集群中每日宕机发生的概 率为千分之一左右，在实践中，一台宕机的机器恢复的时间通常认为是24 小 时，一般需要人工介入重启机器。

网络异常：消息丢失，两片节点之间彼此完全无法通信，即出现了“网络分 化”；消息乱序，有一定的概率不是按照发送时的顺序依次到达目的节点，考 虑使用序列号等机制处理网络消息的乱序问题，使得无效的、过期的网络消息 不影响系统的正确性；数据错误；不可靠的TCP，TCP 协议为应用层提供了可 靠的、面向连接的传输服务，但在分布式系统的协议设计中不能认为所有网络 通信都基于TCP 协议则通信就是可靠的。TCP协议只能保证同一个TCP 链接内 的网络消息不乱序，TCP 链接之间的网络消息顺序则无法保证。

分布式三态：如果某个节点向另一个节点发起RPC(Remote procedure call) 调用，即某个节点A 向另一个节点B 发送一个消息，节点B 根据收到的消息内 容完成某些操作，并将操作的结果通过另一个消息返回给节点A，那么这个RPC 执行的结果有三种状态：“成功”、“失败”、“超时（未知）”，称之为分 布式系统的三态。

存储数据丢失：对于有状态节点来说，数据丢失意味着状态丢失，通常只能从 其他节点读取、恢复存储的状态。

异常处理原则：被大量工程实践所检验过的异常处理黄金原则是：任何在设 计阶段考虑到的异常情况一定会在系统实际运行中发生，但在系统实际运行遇 到的异常却很有可能在设计时未能考虑，所以，除非需求指标允许，在系统设 计时不能放过任何异常情况。

副本

副本（replica/copy）指在分布式系统中为数据或服务提供的冗余。对于数据副本指 在不同的节点上持久化同一份数据，当出现某一个节点的存储的数据丢失时，可以从 副本上读到数据。数据副本是分布式系统解决数据丢失异常的唯一手段。另一类副本 是服务副本，指数个节点提供某种相同的服务，这种服务一般并不依赖于节点的本地 存储，其所需数据一般来自其他节点。

副本协议是贯穿整个分布式系统的理论核心。

副本一致性

分布式系统通过副本控制协议，使得从系统外部读取系统内部各个副本的数据在一定 的约束条件下相同，称之为副本一致性(consistency)。副本一致性是针对分布式系统 而言的，不是针对某一个副本而言。

1. 强一致性(strong consistency)：任何时刻任何用户或节点都可以读到最近 一次成功更新的副本数据。强一致性是程度最高的一致性要求，也是实践中最 难以实现的一致性。
2. 单调一致性(monotonic consistency)：任何时刻，任何用户一旦读到某个 数据在某次更新后的值，这个用户不会再读到比这个值更旧的值。单调一致性 是弱于强一致性却非常实用的一种一致性级别。因为通常来说，用户只关心从 己方视角观察到的一致性，而不会关注其他用户的一致性情况。
3. 会话一致性(session consistency)：任何用户在某一次会话内一旦读到某 个数据在某次更新后的值，这个用户在这次会话过程中不会再读到比这个值更 旧的值。会话一致性通过引入会话的概念，在单调一致性的基础上进一步放松 约束，会话一致性只保证单个用户单次会话内数据的单调修改，对于不同用户 间的一致性和同一用户不同会话间的一致性没有保障。实践中有许多机制正好 对应会话的概念，例如php 中的session 概念。
4. 最终一致性(eventual consistency)：最终一致性要求一旦更新成功，各个 副本上的数据最终将达到完全一致的状态，但达到完全一致状态所需要的时间 不能保障。对于最终一致性系统而言，一个用户只要始终读取某一个副本的数
   据，则可以实现类似单调一致性的效果，但一旦用户更换读取的副本，则无法 保障任何一致性。
5. 弱一致性(week consistency)：一旦某个更新成功，用户无法在一个确定 时间内读到这次更新的值，且即使在某个副本上读到了新的值，也不能保证在 其他副本上可以读到新的值。弱一致性系统一般很难在实际中使用，使用弱一 致性系统需要应用方做更多的工作从而使得系统可用。

3. 衡量分布式系统的指标

性能：系统的吞吐能力，指系统在某一时间可以处理的数据总量，通常可以 用系统每秒处理的总的数据量来衡量；系统的响应延迟，指系统完成某一功能 需要使用的时间；系统的并发能力，指系统可以同时完成某一功能的能力，通 常也用QPS(query per second)来衡量。上述三个性能指标往往会相互制约，追求高吞吐的系统，往往很难做到低延迟；系统平均响应时间较长时，也很难 提高QPS。

可用性：系统的可用性(availability)指系统在面对各种异常时可以正确提供 服务的能力。系统的可用性可以用系统停服务的时间与正常服务的时间的比例 来衡量，也可以用某功能的失败次数与成功次数的比例来衡量。可用性是分布 式的重要指标，衡量了系统的鲁棒性，是系统容错能力的体现。

可扩展性：系统的可扩展性(scalability)指分布式系统通过扩展集群机器规模 提高系统性能（吞吐、延迟、并发）、存储容量、计算能力的特性。好的分布 式系统总在追求“线性扩展性”，也就是使得系统的某一指标可以随着集群中 的机器数量线性增长。

一致性：分布式系统为了提高可用性，总是不可避免的使用副本的机制，从 而引发副本一致性的问题。越是强的一致的性模型，对于用户使用来说使用起 来越简单。

- **Dubbo(一明老师)**
  - 基础知识
    - 为什么要用 Dubbo?
    - Dubbo 是什么?
    - Dubbo 的使用场景有哪些?
    - Dubbo 核心功能有哪些?
  - 架构设计
    - Dubbo 核心组件有哪些?
    - Dubbo 服务器注册与发现的流程?
    - Dubbo 的整体架构设计有哪些分层?
    - Monitor 实现原理?
    - Dubbo 类似的分布式框架还有哪些?
    - Dubbo 和 Spring Cloud 有什么关系?
    - Dubbo 和 Spring Cloud 有什么哪些区别?
    - Dubbo 和 Dubbox 之间的区别?
  - 注册中心
    - Dubbo 有哪些注册中心?
    - Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么?
  - 集群
    - Dubbo集群提供了哪些负载均衡策略?
    - Dubbo的集群容错方案有哪些?
  - 配置
    - Dubbo 配置文件是如何加载到 Spring 中的?
    - 说说核心的配置有哪些?
    - Dubbo 超时设置有哪些方式?
    - 服务调用超时会怎么样?
  - 通信协议
    - Dubbo 使用的是什么通信框架?
    - Dubbo 支持哪些协议，它们的优缺点有哪些?
  - 设计模式
    - Dubbo 用到哪些设计模式?
  - 运维管理
    - 服务上线怎么兼容旧版本?
    - Dubbo telnet 命令能做什么?
    - Dubbo 支持服务降级吗?
    - Dubbo 如何优雅停机?
  - SPI
    - Dubbo SPI 和 Java SPI 区别?
    - Dubbo 支持分布式事务吗?
    - Dubbo 可以对结果进行缓存吗?
    - Dubbo 在安全方面有哪些措施?
    - 服务调用是阻塞的吗?
    - 服务提供者能实现失效踢出是什么原理?
    - 同一个服务多个注册的情况下可以直()连某一个服务吗?
    - Dubbo 服务降级，失败重试怎么做?
    - Dubbo 使用过程中都遇到了些什么问题?
  - Dubbo完整的一次调用链路介绍;
    - 框架设计
      - 整体设计
      - 各层说明
      - 关系说明
      - 模块分包
      - 依赖关系
      - 调用链
      - 暴露服务时序
      - 引用服务时序
      - 领域模型
      - 基本设计原则
  - Dubbo支持几种负载均衡策略?
    - 负载均衡
      - 负载均衡策略
        - Random LoadBalance
        - RoundRobin LoadBalance
        - LeastActive LoadBalance
        - ConsistentHash LoadBalance
    - 配置
      - 服务端服务级别
      - 客户端服务级别
      - 服务端方法级别
      - 客户端方法级别
    - dubbo负载均衡策略
  - Dubbo Provider服务提供者要控制执行并发请求上限，具体怎么做
    - 并发控制
      - 配置样例
        - 样例 1
        - 样例 2
        - 样例 3
        - 样例 4
  - Dubbo启动的时候支持几种配置方式?
    - XML 配置
      - provider.xml 示例
      - consumer.xml示例
      - 配置之间的关系
      - 不同粒度配置的覆盖关系

- 属性配置
- 映射规则
  - 重写与优先级
- API 配置
  - 服务提供者
  - 服务消费者
  - 特殊场景
  - 方法级设置
  - 点对点直连
- 注解配置
  - 服务提供方
    - `Service`注解暴露服务
    - 增加应用共享配置
    - 指定Spring扫描路径
  - 服务消费方
    - `Reference`注解引用服务
    - 增加应用共享配置
    - 指定Spring扫描路径
    - 调用服务

分布式系统原理
- 数据分布方式
  - 哈希方式
  - 按数据范围分布
  - 按数据量分布
  - 一致性哈希
  - 副本与数据分布
  - 本地化计算
  - 数据分布方式的选择
- 基本副本协议
  - 中心化副本控制协议
  - primary-secondary 协议
  - 数据读取方式
  - primary 副本的确定与切换
  - 数据同步
  - 去中心化副本控制协议
- Lease 机制
  - 基于lease 的分布式cache 系统
  - lease 机制的分析
  - 基于lease 机制确定节点状态
  - lease 的有效期时间选择
- Quorum 机制
  - write-all-read-one
  - Quorum 定义
  - 读取最新成功提交的数据
  - 基于Quorum 机制选择primary副本
- 日志技术
  - Redo Log 与Check point
- 两阶段提交协议
  - 流程描述
  - 异常处理
  - 协议分析
- MVCC
- Paxos协议
- CAP

Dubbo 支持哪些协议，每种协议的应用场景，优缺点?
Dubbo 超时时间怎样设置?
Dubbo 有些哪些注册中心?
Dubbo 集群的负载均衡有哪些策略
 Dubbo 是什么?
 Dubbo 的主要应用场景?
Dubbo 的核心功能?
Dubbo 的核心组件?
Dubbo 服务注册与发现的流程?
流程说明：
设计的原因：
Dubbo 的架构设计?
Dubbo 框架设计一共划分了 10 个层：
Dubbo 的服务调用流程?
Dubbo 支持哪些协议，每种协议的应用场景，优缺点?
Dubbo 有些哪些注册中心?
为什么需要服务治理?
Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么?
Dubbo 与 Spring 的关系?
Dubbo 集群提供了哪些负载均衡策略?
LeastActive LoadBalance: 最少活跃调用策略，解决慢提供者接收
Failover Cluster
Failfast Cluster
Failback Cluster
roadcast Cluster
Dubbo 支持哪些序列化方式?
Dubbo 超时时间怎样设置?
Dubbo 在安全机制方面是如何解决?
Dubbo 和 Dubbox 之间的区别?
Dubbo 和 Spring Cloud 的关系?
Dubbo 和 Spring Cloud 的区别?
Dubbo 中 zookeeper 做注册中心，如果注册中心集群都挂掉，发布者和订阅者之间还能通信么?
dubbo 服务负载均衡策略?

Dubbo 在安全机制方面是如何解决的
dubbo 连接注册中心和直连的区别
1. **dubbo 通信协议 dubbo 协议为什么要消费者比提供者个数多：**
2. **dubbo 通信协议 dubbo 协议为什么不能传大包：**
3. **dubbo 通信协议 dubbo 协议为什么采用异步单一长连接：**
4. **dubbo 通信协议 dubbo 协议适用范围和适用场景**
5. **RMI 协议**
6. **Hessian 协议**
7. **http**
8. **Webservice**
9. **Thrif**

Thrift
Protoclol Buffer
Mybatis（连老师）
JPA 原理
Mybatis 缓存
Mybatis 的一级缓存原理（**sqlsession级别**）
MyBatis缓存
MyBatis缓存机制的设计与实现
MyBatis数据源与连接池
一、MyBatis数据源DataSource分类
二、数据源DataSource的创建过程
三、 DataSource什么时候创建Connection对象
四、不使用连接池的UnpooledDataSource
五、为什么要使用连接池?
六、使用了连接池的PooledDataSource
七、JNDI类型的数据源DataSource
MyBatis简介
1. MyBatis是什么?
2. ORM是什么
3. 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里?
4. 传统JDBC开发存在的问题
5. JDBC编程有哪些不足之处，MyBatis是如何解决这些问题的?
6. Mybatis优缺点
7. MyBatis框架适用场景
8. Hibernate 和 MyBatis 的区别

MyBatis的解析和运行原理
9. MyBatis编程步骤是什么样的?
10. 请说说MyBatis的工作原理

Spring MVC 运行流程
什么是 MyBatis?
讲下 MyBatis 的缓存答
Mybatis 是如何进行分页的？分页插件的原理是什么？答:
简述 Mybatis 的插件运行原理，以及如何编写一个插件？答:
Mybatis 动态 sql 是做什么的？都有哪些动态 sql？能简述一下动态 sql 的执行原理不？答:
#{}和${}的区别是什么？答: 1）#{}是预编译处理，${}是字符串替换。
为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里?
Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？答:
MyBatis 与 Hibernate 有哪些不同？答:
MyBatis 的好处是什么？答:
简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系?
什么是 MyBatis 的接口绑定,有什么好处?
接口绑定有几种实现方式,分别是怎么实现的?
什么情况下用注解绑定,什么情况下用 xml 绑定?
MyBatis 实现一对一有几种方式?具体怎么操作的?
Mybatis 能执行一对一、一对多的关联查询吗？都有哪些实现方式，以及它们之间的区别?
MyBatis 里面的动态 Sql 是怎么设定的?用什么语法?
Mybatis 是如何将 sql 执行结果封装为目标对象并返回的？都有哪些映射形式？答:
Xml 映射文件中，除了常见的 select|insert|updae|delete 标签之外，还有哪些标签?
当实体类中的属性名和表中的字段名不一样，如果将查询的结果封装到指定 pojo？答:
模糊查询 like 语句该怎么写答:
通常一个 Xml 映射文件，都会写一个 Dao 接口与之对应, Dao 的工作原理，是否可以重载?
Mybatis 映射文件中，如果 A 标签通过 include 引用了 B 标签的内容，请问，B 标签能否定义在 A 标签的后面，还是说必须定义在 A 标签的前面?
Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？答：不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复；如果没有配置
Mybatis 中如何执行批处理?
Mybatis 都有哪些 Executor 执行器？它们之间的区别是什么?
Mybatis 中如何指定使用哪一种 Executor 执行器?
Mybatis 执行批量插入，能返回数据库主键列表吗?
Mybatis 是否可以映射 Enum 枚举类?
如何获取自动生成的(主)键值?
在 mapper 中如何传递多个参数？答:
resultType resultMap 的区别?
使用 MyBatis 的 mapper 接口调用时有哪些要求？答:
Mybatis 比 IBatis 比较大的几个改进是什么？答:
Redis（周老师）
Redis 概述
什么是Redis
Redis有哪些优缺点
为什么要用 Redis /为什么要用缓存
为什么要用 Redis 而不用 map/guava 做缓存?
Redis为什么这么快

数据类型
Redis有哪些数据类型
7Redis的应用场景
总结一
总结二

持久化
- 什么是Redis持久化?
- 9Redis 的持久化机制是什么？各自的优缺点?
  - RDB：是Redis DataBase缩写快照
    - 优点：
    - 缺点：
  - AOF：持久化
    - 优点：
    - 缺点：
  - 优缺点是什么?
- 如何选择合适的持久化方式
- Redis持久化数据和缓存怎么做扩容?

过期键的删除策略
- Redis的过期键的删除策略
- Redis key的过期时间和永久有效分别怎么设置?
  - 我们知道通过expire来设置key 的过期时间，那么对过期的数据怎么处理呢?

内存相关
- Redis的内存淘汰策略有哪些
  - Redis主要消耗什么物理资源?
- Redis的内存用完了会发生什么?
- Redis如何做内存优化?
- Redis线程模型
- 什么是事务?
- Redis事务的概念
- Redis事务的三个阶段
- Redis事务相关命令
- 事务管理（ACID）概述
- Redis事务支持隔离性吗
- Redis事务保证原子性吗，支持回滚吗
- Redis事务其他实现
  - 官方Redis Cluster 方案(服务端路由查询)
- Redis 主从架构
- Redis集群的主从复制模型是怎样的?
- 说说Redis哈希槽的概念?
- Redis集群会有写操作丢失吗？为什么?

分区
- Redis是单线程的，如何提高多核CPU的利用率?
- 为什么要做Redis分区?
- 你知道有哪些Redis分区实现方案?
- Redis分区有什么缺点?

分布式问题
- Redis实现分布式锁
- 如何解决 Redis 的并发竞争 Key 问题
- 分布式Redis是前期做还是后期规模上来了再做好？为什么?
- 什么是 RedLock
- 缓存穿透
- 缓存击穿
- 缓存降级
- 热点数据和冷数据
- 缓存热点key
- Redis和Redisson有什么关系?
- Jedis与Redisson对比有什么优缺点?

其他问题
- Redis与Memcached的区别
- 如何保证缓存与数据库双写时的数据一致性?
- Redis常见性能问题和解决方案?
- Redis官方为什么不提供Windows版本?
  - 一个字符串类型的值能存储最大容量是多少?
- 假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来?
- 使用Redis做过异步队列吗，是如何实现的
- Redis如何实现延时队列
- Redis回收进程如何工作的?
- Redis回收使用的是什么算法?

消息中间件如何保证消息的一致性

如何进行消息的重试机制?
- Rocket重试机制，消息模式，刷盘方式
  - 一、Consumer 批量消费（推模式）
  - 二、消息重试机制：消息重试分为2种
  - 三、消费模式
  - 四、conf下的配置文件说明
  - 五、刷盘方式

Redis数据类型
- 概述
- 字符串类型
  - 基本命令
    - 字符串操作
    - 位操作
- 散列类型
  - 基本命令
- 列表类型
  - 基本命令
- 集合类型
  - 基本命令
- 有序集合类型
  - 基本命令

redis集群如何同步;
- Redis 复制与集群

复制
同步方式
问题
主从切换
集群
key映射到节点的算法
握手
集群行为
redis的数据添加过程是怎样的：哈希槽；
redis集群实现（四） 数据的和槽位的分配
redis的淘汰策略有哪些；
redis 集群基础
redis Cluster主从模式
Redis为什么这么快?
redis采用多线程会有哪些问题?
阿里云Redis多线程性能提升思路解析
Redis支持哪几种数据结构；
Redis跳跃表的问题；
跳跃表以及跳跃表在redis中的实现
1 跳跃表的原理
2 跳跃表在redis中的实现
2.1 跳跃表节点定义和跳跃表描述符定义
2.2 新建跳跃表
2.3 插入新节点
2.4 删除指定节点
2.5 跳跃表的查询
Redis是单进程单线程的，如何能够高并发?
Redis如何使用Redis实现分布式锁?
Redis分布式锁的正确实现方式
前言
可靠性
代码实现
组件依赖
加锁代码
解锁代码
总结
Redis分布式锁操作的原子性，Redis内部是如何实现的?
网络（周老师，李老师）
计算机网络体系结构
网络协议是什么?
为什么要对网络协议分层?
TCP/IP 四层体系结构。 TCP/IP 协议族
应用层
运输层
网络层
数据链路层
物理层
TCP/IP 协议族
TCP的三次握手四次挥手
TCP报文的头部结构
三次握手
四次挥手
常见面试题
为什么TCP连接的时候是3次？2次不可以吗?
为什么TCP连接的时候是3次，关闭的时候却是4次?
为什么客户端发出第四次挥手的确认报文后要等2MSL的时间才能释放TCP连接?
如果已经建立了连接，但是客户端突然出现故障了怎么办?
常用HTTP状态码
GET和POST区别
什么是对称加密与非对称加密
Session、Cookie和Token的主要区别
Servlet是线程安全的吗
Servlet接口中有哪些方法及Servlet生命周期探秘
如果客户端禁止 cookie 能实现 session 还能用吗?
socket通信，以及长连接，分包，连接异常断开的处理。
Socket TCP/IP协议数据传输过程中的粘包和分包问题
深入理解socket网络异常
建立连接
1 正常情况下
2 客户端连接了服务端未监听的端口
3 客户端与服务器之间的网络不通
4 建立连接的过程中包丢失
通信过程
1 客户端与服务器的网络断开，双方不再发送数据
2 网络断开，一方给另一方发送数据
3 网络断开，一方等待着另一方发送数据
4 一方crash，另一方继续发送/接收数据
总结
http中，get post的区别
HTTP报文内容
HTTP请求报文和HTTP响应报文
1.请求头
2.请求头部
3.空行
4.请求数据
HTTP报文
关于HTTP请求GET和POST的区别

- 1.**GET提交，请求的数据会附在URL之后（就是把数据放置在HTTP协议头<request-line>中）**
- 说说浏览器访问http://www.taobao.com，经历了怎样的过程。
  - 访问 www.taobao.com过程
- HTTP协议、 HTTPS协议，SSL协议及完整交互过程；
  - HTTPS协议，SSL协议及完整交互过程
    - SSL
    - HTTPS
    - 交互过程
- Ping过程 原理 详解
- TCP/IP协议详解笔记——ARP协议和RARP协议
  - ARP：地址解析协议
    - 工作过程
    - 帧格式
    - ARP高速缓存
    - ARP代理
  - RARP：逆地址解析协议
    - 帧格式
    - 工作原理
- DNS域名解析的过程
- Http会话的四个过程
- 网络7层架构
- TCP/IP 原理
- **TCP** 三次握手**/**四次挥手
- **HTTP** 原理
- **CDN** 原理
- RPC
  - 为什么要有RPC
  - 什么是RPC
  - PRC架构组件
  - RPC和SOA、SOAP、REST的区别
  - RPC框架需要解决的问题?
  - RPC的实现基础?
  - RPC使用了哪些关键技术?
  - 主流RPC框架有哪些
  - RPC的实现原理架构图
- **Znode** 有四种形式的目录节点
- 负载均衡
- 四层负载均衡 **vs** 七层负载均衡
- 负载均衡算法**/**策略
- LVS
- Keepalive
- Nginx 反向代理负载均衡

Linux（李老师）

- Linux概述
  - 1. 什么是Linux
  - 2. Unix和Linux有什么区别?
  - 3. 什么是 Linux 内核?
  - 4. Linux的基本组件是什么?
  - 5. Linux 的体系结构
  - 6. BASH和DOS之间的基本区别是什么?
  - 7. Linux 开机启动过程?
  - 8. Linux系统缺省的运行级别?
  - 9. Linux 使用的进程间通信方式?
  - 10. Linux 有哪些系统日志文件?
- Linux系统安装多个桌面环境有帮助吗?
- 什么是交换空间?
- 什么是root帐户
- 什么是LILO?
- 什么是CLI?
- 什么是GUI?
- 开源的优势是什么?
- GNU项目的重要性是什么?
- 磁盘、目录、文件
  - 简单 Linux 文件系统?
  - Linux 的目录结构是怎样的?
  - 什么是 inode ?
  - 什么是硬链接和软链接?
- 安全
  - 一台 Linux 系统初始化环境后需要做一些什么安全工作?
  - 什么叫 CC 攻击? 什么叫 DDOS 攻击?
  - 什么是网站数据库注入?
- Shell
  - Shell 脚本是什么?
  - 语法级
  - 编程题(。)
  - 如何规划一台 Linux 主机，步骤是怎样?
  - 请问当用户反馈网站访问慢，你会如何处理?
  - Linux 性能调优都有哪几种方法?
  - 文件管理命令 cat 命令
  - chmod 命令
  - chown 命令
  - cp 命令
  - find 命令
  - head 命令
  - less 命令
  - ln 命令
  - locate 命令

- more 命令
- mv 命令
- rm 命令
- tail 命令
- （查看日志） touch 命令
- vim 命令
- whereis 命令
- which 命令
- wc 命令

磁盘管理命令 cd 命令
- df 命令
- du 命令
- ls命令
- mkdir 命令
- pwd 命令
- rmdir 命令
- 网络通讯命令 ifconfig 命令
- netstat 命令
- ping 命令
- telnet 命令
- 系统管理命令 date 命令
- free 命令
- kill 命令
- ps 命令
- rpm 命令
- top 命令
- yum 命令
- 备份压缩命令 bzip2 命令
- gzip 命令
- tar 命令
- unzip 命令

swap分区
- Swap交换分区概念
- Linux系统swappiness参数在内存与交换分区之间优化作用

overcommit_memory
- 又一次内存分配失败(关于overcommit_memory)

linux系统下查看CPU、内存负载情况

经典面试题
- **问题一：**绝对路径用什么符号表示？当前目录、上层目录用什么表示？主目录用什么表示？切换目录用什么命令？
- 答案：
- **问题二：**怎么查看当前进程？怎么执行退出？怎么查看当前路径？**答案：**
- **问题三：**怎么清屏？怎么退出当前命令？怎么执行睡眠？怎么查看当前用户 id？查看指定帮助用什么命令？
- **答案：**
- 问题四：Ls 命令执行什么功能？可以带哪些参数，有什么区别？
- **问题五：**建立软链接(快捷方式)，以及硬链接的命令。
- 答案：
- **问题六：**目录创建用什么命令？创建文件用什么命令？复制文件用什么命令？**答案：**
- **问题八：**查看文件内容有哪些命令可以使用？**答案：**
- **问题九：**
- 随意写文件命令？怎么向屏幕输出带空格的字符串，比如”hello world”？
- 答案：
- **问题十：**终端是哪个文件夹下的哪个文件？黑洞文件是哪个文件夹下的哪个命令？**答案：**
- **问题十一：**移动文件用哪个命令？改名用哪个命令？**答案：** mv mv
- 问题十二：
- **问题十三：**使用哪一个命令可以查看自己文件系统的磁盘空间配额呢？

MQ（一明老师，周老师）

为什么使用MQ？MQ的优点
- 简答
- 详答

消息队列有什么优缺点？RabbitMQ有什么优缺点？

你们公司生产环境用的是什么消息中间件？

Kafka、ActiveMQ、RabbitMQ、RocketMQ 有 什么优缺点？

MQ 有哪些常见问题？如何解决这些问题？

什么是RabbitMQ?

Rabbitmq 的使用场景

RabbitMQ基本概念

RabbitMQ的工作模式
- 一.simple模式（即最简单的收发模式）
- 二.work工作模式(资源的竞争)
- 三.publish/subscribe发布订阅(共享资源)
- 四.routing路由模式
- 五.topic 主题模式(路由模式的一种)

如何保证RabbitMQ消息的顺序性？

消息如何分发？

消息怎么路由？

消息基于什么传输？

如何保证消息不被重复消费？或者说，如何保证消息消费时的幂等性？

如何确保消息正确地发送至RabbitMQ？ 如何确保消息接收方消费了消息？

如何保证RabbitMQ消息的可靠传输？

为什么不应该对所有的 message 都使用持久化机制？

如何保证高可用的？RabbitMQ 的集群

设计MQ思路

消息中间件如何保证消息的一致性

如何进行消息的重试机制？
- Rocket重试机制，消息模式，刷盘方式
  - 一、Consumer 批量消费（推模式）
  - 二、消息重试机制：消息重试分为2种

三、消费模式
四、conf下的配置文件说明
五、刷盘方式
传递保证语义：
生产者的“Exactly once”语义方案
消费者的“Exactly once”语义方案
ISR集合
请说明什么是Apache Kafka?
请说明什么是传统的消息传递方法?
请说明Kafka相对传统技术有什么优势?
在Kafka中broker的意义是什么?
Kafka服务器能接收到的最大信息是多少?
解释Kafka的Zookeeper是什么?我们可以在没有Zookeeper的情况下使用Kafka吗?
解释Kafka的用户如何消费信息?
解释如何提高远程用户的吞吐量?
解释一下，在数据制作过程中，你如何能从Kafka得到准确的信息?
解释如何减少ISR中的扰动?broker什么时候离开ISR?
Kafka为什么需要复制?
如果副本在ISR中停留了很长时间表明什么?
请说明如果首选的副本不在ISR中会发生什么?
有可能在生产后发生消息偏移吗?
kafka与传统的消息中间件对比
kafka与传统的消息中间件对比
Kafka 对比 ActiveMQ
Kafka 的优点
KAFKA：如何做到1秒发布百万级条消息
KAFKA：如何做到1秒发布百万级条消息
kafka文件存储
Kafka深入理解-1：Kafka高效的文件存储设计
Kafka是什么
1.前言
2.Kafka文件存储机制
2.1 topic中partition存储分布
2.2 partiton中文件存储方式
2.3 partiton中segment文件存储结构
2.4 在partition中如何通过offset查找message
3 Kafka文件存储机制–实际运行效果
4.总结
kafka的log存储解析——topic的分区partition分段segment以及索引等
引言
Partition的数据文件
数据文件的分段
为数据文件建索引
小结
Kafka 的设计时什么样的呢?
数据传输的事物定义有哪三种?
Kafka 判断一个节点是否还活着有那两个条件?
producer 是否直接将数据发送到 broker 的 leader(主节点)?
Kafa consumer 是否可以消费指定分区消息?
Kafka 消息是采用 Pull 模式，还是 Push 模式?
Kafka 存储在硬盘上的消息格式是什么?
Kafka 高效文件存储设计特点：
Kafka 与传统消息系统之间有三个关键区别
Kafka 创建 Topic 时如何将分区放置到不同的 Broker 中
Kafka 新建的分区会在哪个目录下创建
Zookeeper（周老师）
ZooKeeper 是什么?
ZooKeeper 提供了什么?
Zookeeper 文件系统
Zookeeper 怎么保证主从节点的状态同步?
四种类型的数据节点 Znode
Zookeeper Watcher 机制 – 数据变更通知
客户端注册 Watcher 实现
服务端处理 Watcher 实现
客户端回调 Watcher
ACL 权限控制机制
Chroot 特性
会话管理
服务器角色
Zookeeper 下 Server 工作状态
数据同步
zookeeper 是如何保证事务的顺序一致性的?
分布式集群中为什么会有 Master主节点?
zk 节点宕机如何处理?
Zookeeper 有哪几种几种部署模式?
集群最少要几台机器，集群规则是怎样的?集群中有 3 台服务器，其中一个节点宕机，这个时候 Zookeeper 还可以使用吗?
集群支持动态添加机器吗?
Zookeeper 对节点的 watch 监听通知是永久的吗？为什么不是永久的?
Zookeeper 的 java 客户端都有哪些?
chubby 是什么，和 zookeeper 比你怎么看?
说几个 zookeeper 常用的命令。
ZAB 和 Paxos 算法的联系与区别?
Zookeeper 的典型应用场景
Zookeeper 都有哪些功能?
说一下 Zookeeper 的通知机制?
Zookeeper 和 Dubbo 的关系?
zookeeper集群之间如何通讯

- zookeeper系列之通信模型
- Zookeeper面试题
  - ZooKeeper 是什么?
  - ZooKeeper 提供了什么?
  - Zookeeper 文件系统
  - 四种类型的 znode
  - Zookeeper 通知机制
  - Zookeeper 做了什么?
  - zk 的命名服务（文件系统）
  - zk 的配置管理（文件系统、通知机制）
  - Zookeeper 集群管理（文件系统、通知机制）
  - Zookeeper 分布式锁（文件系统、通知机制）
  - 获取分布式锁的流程
  - Zookeeper 队列管理（文件系统、通知机制）
  - Zookeeper 数据复制
  - Zookeeper 工作原理
  - zookeeper 是如何保证事务的顺序一致性的?
  - Zookeeper 下 Server 工作状态
  - zookeeper 是如何选取主 leader 的?
- 微服务
- 服务注册发现
- **API 网关**
- 配置中心
- 事件调度（**kafka**）
- 服务跟踪（**starter-sleuth**）
- 服务熔断（**Hystrix**）
- **API** 管理
- Zookeeper概念
- Zookeeper角色
- **ZAB协议**
- **投票机制**
- **Zookeeper** 工作原理（原子广播）

Netty（周老师）

- Netty简介
  - JDK原生NIO程序的问题
  - Netty的特点
  - Netty常见使用场景
- Netty高性能设计
  - I/O模型
    - I/O复用模型
    - 基于buffer
    - 事件驱动模型
    - Reactor线程模型
    - Netty线程模型
  - 异步处理
- Netty架构设计
  - Bootstrap、ServerBootstrap
  - Future、ChannelFuture
  - Channel
  - Selector
  - NioEventLoop
  - NioEventLoopGroup
  - 工作原理架构
- Netty面试题
  - 1. Netty 是什么?
  - 2. Netty 特点是什么?
  - 3. Netty 的优势有哪些?
  - 4. Netty 的应用场景有哪些?
  - 5. Netty 高性能表现在哪些方面?
  - 6. BIO、NIO和AIO的区别?
  - 7. NIO的组成?
  - 8. Netty的线程模型?
  - 9. TCP 粘包/拆包的原因及解决方法?
  - 10. 什么是 Netty 的零拷贝?
  - 11.Netty 中有哪种重要组件?
  - 12.Netty 发送消息有几种方式?
  - 13.默认情况 Netty 起多少线程？何时启动?
  - 14.了解哪几种序列化协议?
  - 15.如何选择序列化协议?
  - 16.Netty 支持哪些心跳类型设置?
  - 17.Netty 和 Tomcat 的区别?
  - 18.NIOEventLoopGroup源码?
- Netty
  - 为什么选择 Netty
- Netty经典面试题
  - BIO、NIO 和 AIO 的区别?
  - NIO 的组成?
  - Netty 的特点?
  - Netty 的线程模型?
  - TCP 粘包/拆包的原因及解决方法?
  - 了解哪几种序列化协议?
  - 如何选择序列化协议?
  - Netty 的零拷贝实现?
  - Netty 的高性能表现在哪些方面?
  - NIOEventLoopGroup 源码?
- Netty 原理
- Netty 高性能

- **Netty RPC** 实现

大数据（周老师）

- Hadoop
  - 概念
  - **HDFS**
  - MapReduce
  - **Hadoop MapReduce** 作业的生命周期
- Spark
  - 概念
  - 核心架构
  - 核心组件
  - SPARK编程模型
  - **SPARK** 计算模型
  - **SPARK** 运行流程
  - **SPARK RDD** 流程
  - **SPARK RDD**
- Storm
  - 概念
  - 集群架构
  - **Nimbus**（**master**-代码分发给 **Supervisor**）
  - 编程模型（**spout->tuple->bolt**）
  - **Topology** 运行
- YARN
  - 概念
  - **ResourceManager**
  - NodeManager
  - **ApplicationMaster**
  - YARN运行流程
- 负载均衡的原理
  - 六大Web负载均衡原理与实现
    - 一、http重定向
    - 二、DNS负载均衡
    - 三、反向代理负载均衡
    - 四、IP负载均衡(LVS-NAT)
    - 五、直接路由(LVS-DR)
    - 六、IP隧道(LVS-TUN)
- Tomcat并发
  - Tomcat的性能与最大并发配置
- **Exchange** 类型
- base概念
- 列式存储
- **Hbase** 核心概念
- **Hbase** 核心架构
- **Hbase** 的写逻辑
- **HBase vs Cassandra**
- 概念
- 概念
- 数据模型
- **Cassandra** 一致 **Hash** 和虚拟节点
- Gossip** 协议
- 数据复制
- 数据写请求和协调者
- 数据读请求和后台修复
- 数据存储（**CommitLog**、**MemTable**、**SSTable**）
- 二级索引（对要索引的 **value** 摘要，生成 **RowKey**）
- 数据读写
- 存储引擎
- 索引
- 数据库三范式
- 数据库是事务
- 存储过程(特定功能的 SQL 语句集)
- 触发器(一段能自动执行的程序)
- 数据库并发策略
- 数据库锁
- Semphore、CountDownLatch、CyclicBarrier、Phaser
  - 并发工具类_同步屏障CyclicBarrier
    - 简介
    - CyclicBarrier的应用场景
    - CyclicBarrier和CountDownLatch的区别
  - 并发工具类_控制并发线程数的Semaphore
    - 简介
    - 应用场景
    - 其他方法
- CLH队列
  - 【Java并发编程实战】—– AQS(四)：CLH同步队列
    - 入列
    - 出列
    - 取消
    - 挂起
  - JAVA并发编程学习笔记之CLH队列锁
    - NUMA与SMP
    - CLH算法实现
    - CLH优缺点

算法（左老师-主要讲技巧）

- 红黑树的了解（平衡树，二叉搜索树），使用场景
  1. AVLtree
  2. 二叉搜索树

- 3. 红黑树的定义
- 4. B树定义
- 5. B+树
- 6. Trie树（字典树）

（2） 红黑树在STL上的应用

了解并查集吗？（低频）

（4） 贪心算法和动态规划的区别

（5） 判断一个链表是否有环，如何找到这个环的起点

实现一个strcpy函数（或者memcpy），如果内存可能重叠呢

- 内存重叠

快排存在的问题，如何优化

- 快排的时间复杂度

Top K问题（可以采取的方法有哪些，各自优点？）

Bitmap的使用，存储和插入方法

- BitMap从字面的意思
- bitmap应用
- bitmap排序复杂度分析
- BitMap算法流程
- BitMap算法评价

字典树的理解以及在统计上的应用

N个骰子出现和为m的概率

海量数据问题（可参考左神的书）

一致性哈希说明

如何给100亿个数字排序?

- 场景
- 实现思路
- 思维拓展

哈希分治法 - 统计海量数据中出现次数最多的前10个IP

- 场景
- 分析
- 实现思路
- MapReduce

排序算法时间复杂度

判断链表中是否有环 ----- 有关单链表中环的问题

常见hash算法的原理

七大查找算法

1. 顺序查找
2. 二分查找
3. 插值查找
4. 斐波那契查找
5. 树表查找

红黑树的平均高度大约为logn。

6. 分块查找

7. 哈希查找

一致性算法

- **Paxos**
- **Zab**
- **Raft**
- **NWR**
- **Gossip**
- 一致性 **Hash**

JAVA 算法

- 二分查找
- 冒泡排序算法
- **插入排序算法**
- 快速排序算法
- **希尔排序算法**
- 归并排序算法
- 桶排序算法
- 基数排序算法
- 剪枝算法
- 回溯算法
- 最短路径算法
- 最大子数组算法
- 最长公共子序算法
- 最小生成树算法

数据结构

- 栈（**stack**）
- 队列（**queue**）
- 链表（**Link**）
- 散列表（**Hash Table**）
- 排序二叉树
- 红黑树
- **B-TREE**
- 位图

加密算法

- **AES**
- **RSA**
- **CRC**
- **MD5**

项目方面（晁老师-主要讲经验）

- 日志
- Slf4j
- Log4j
- LogBack

设计模式（马老师）

- 单例模式

工厂模式
抽象工厂模式
建造者模式
原型模式
适配器模式
装饰器模式
代理模式
中介者模式
命令模式
责任链模式
策略模式
迭代器模式
组合模式
观察者模式
门面模式
备忘录模式
访问者模式
状态模式
解释器模式
享元模式
桥梁模式
模板方法模式
一千万的用户实时排名如何实现；
    海量积分数据实时排名处理
        需求概述
            常规积分排名处理
高效的排名算法
    如何高效地获得玩家排名？
基于redis的zset实现排行榜功能
五万人并发抢票怎么实现；
    Web大规模高并发请求和抢购的解决方案
集群服务器Session同步
大数据量高并发的数据库优化
如何提高服务器并发处理能力
    什么是服务器并发处理能力
    有什么方法衡量服务器并发处理能力
        1. 吞吐率
        2. 压力测试
    怎么提高服务器的并发处理能力
        1. 提高CPU并发计算能力
            多进程 & 多线程
            减少进程切换
            减少使用不必要的锁
            考虑进程优先级
            考虑系统负载
            考虑CPU使用率
        2. 考虑减少内存分配和释放
        3. 考虑使用持久连接
        4. 改进I/O 模型
            1. DMA技术
            2. 异步I/O
            3. I/O多路复用
            4. Sendfile
            5. 内存映射
            6. 直接I/O
        5. 改进服务器并发策略
            一个进程处理一个连接，非阻塞I/O
            一个线程处理一个连接，非阻塞IO
            一个进程处理多个连接，异步I/O
            一个线程处理多个连接，异步IO
        6. 改进硬件环境
假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题。
    三个方面解决性能问题的基本思路和方法
        发现问题、解决问题和预防问题三个方面
            1. 用户反馈（包括测试人员）
            2.在线监控
        优化成果
        总结
        问题
            (0)想问下你们 DB 操作这部分涉及到多线程读写是怎么处理的?
            (1)除了 sqlite 语句的优化之外，db 这部分还有没有其他方面的优化工作?
            (2) 请问你们选择用sqlite的考量是什么, 有没有考虑过使用其他的db如realm?
            (3)合理的使用线程，多线程之间的同步这块儿有什么方案或建议?
            (4) 我把我的问题编号为4吧，方便对应。业务场景里会不会涉及到有 读操作 依赖写操作完成的情况，否则会出现读操作的数据不准确的情况。FMDB 感觉不能很好的解决这个问题。
            数据库的记录-一般是在什么级别百千有没有尝试去做过一些压测数据量达到多少的时候会遇到瓶颈">(5)能不能问下 微信读书的数据库的记录 一般是在什么级别，百、千？有没有尝试去做过一些压测，数据量达到多少的时候会遇到瓶颈?
            (6) 微信读书这么成功，方便说下她的架构吗？我觉得架构好才是她可优化的第一步
            (7) 你们的 db 是只有一个文件，还是尝试分文件存储的?
造成性能瓶颈。
    五步定位性能瓶颈
    性能测试如何定位瓶颈
    性能测试如何定位瓶颈
你的项目中使用过缓存机制吗？有没用用户非本地缓存
    使用缓存的9大误区（上）
    使用缓存的9大误区（下）
        缓存大量的数据集合，而读取其中一部分

缓存大量具有图结构的对象导致内存浪费
缓存应用程序的配置信息
使用很多不同的键指向相同的缓存项
没有及时的更新或者删除再缓存中已经过期或者失效的数据
Tomcat 调优及 JVM 参数优化
设计模式

Dubbo(一明老师)

基础知识

为什么要用 Dubbo?

随着服务化的进一步发展，服务越来越多，服务之间的调用和依赖关系也越来越 复杂，诞生了面向服务的架构体系(SOA)，也因此衍生出了一系列相应的技术，如对服务提供、服务调用、连接处理、通信协议、序列化方式、服务发现、服务 路由、日志输出等行为进行封装的服务框架。就这样为分布式系统的服务治理框 架就出现了，Dubbo 也就这样产生了。

Dubbo 是什么?

Dubbo 是一款高性能、轻量级的开源 RPC 框架，提供服务自动注册、自动发 现等高效服务治理方案， 可以和 Spring 框架无缝集成。

Dubbo 的使用场景有哪些?

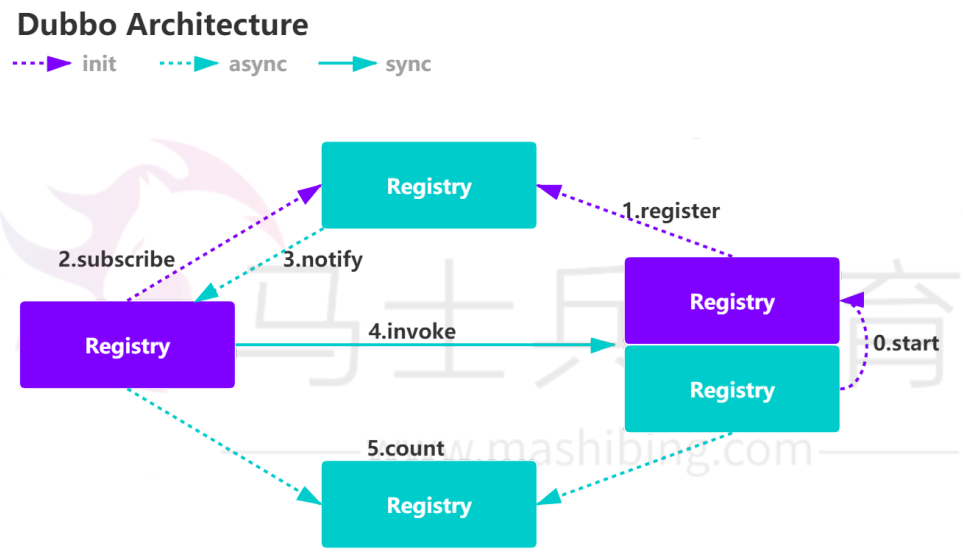
- 透明化的远程方法调用：就像调用本地方法一样调用远程方法，只需简单配置， 没有任何API侵入。
- 软负载均衡及容错机制：可在内网替代 F5 等硬件负载均衡器，降低成本，减少 单点。
- 服务自动注册与发现：不再需要写死服务提供方地址，注册中心基于接口名查询 服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

Dubbo 核心功能有哪些?

- Remoting：网络通信框架，提供对多种NIO框架抽象封装，包括“同步转异 步”和“请求-响应”模式的信息交换方式。
- Cluster：服务框架，提供基于接口方法的透明远程过程调用，包括多协议支 持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。
- Registry：服务注册，基于注册中心目录服务，使服务消费方能动态的查找服务 提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

架构设计

Dubbo 核心组件有哪些?



- Provider：暴露服务的服务提供方
- Consumer：调用远程服务消费方
- Registry：服务注册与发现注册中心
- Monitor：监控中心和访问调用统计
- Container：服务运行容器

Dubbo 服务器注册与发现的流程?

服务容器Container负责启动，加载，运行服务提供者。

服务提供者Provider在启动时，向注册中心注册自己提供的服务。

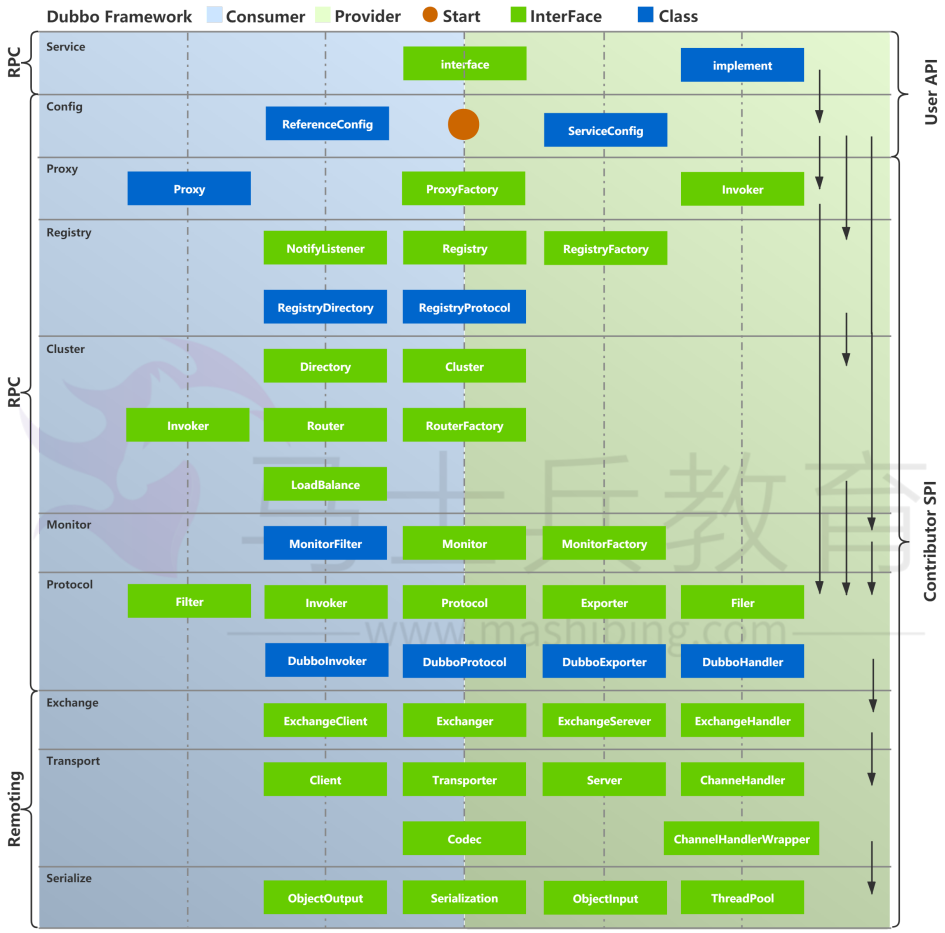
服务消费者Consumer在启动时，向注册中心订阅自己所需的服务。

中心Registry返回服务提供者地址列表给消费者，如果有变更，注册中心 将基于长连接推送变更数据给消费者。

服务消费者Consumer，从提供者地址列表中，基于软负载均衡算法，选一台 提供者进行调用，如果调用失败，再选另一台调用。

服务消费者Consumer和提供者Provider，在内存中累计调用次数和调用时 间，定时每分钟发送一次统计数据到监控中心Monitor。

Dubbo 的整体架构设计有哪些分层?



接口服务层（Service）：该层与业务逻辑相关，根据 provider 和 consumer 的业务设计对应的接口和实现

配置层（Config）：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为 中心

服务代理层（Proxy）：服务接口透明代理，生成服务的客户端 Stub 和 服务端 的 Skeleton，以 ServiceProxy 为中心，扩展接口为 ProxyFactory

服务注册层（Registry）：封装服务地址的注册和发现，以服务 URL 为中心， 扩展接口为 RegistryFactory、Registry、RegistryService

路由层（Cluster）：封装多个提供者的路由和负载均衡，并桥接注册中心，以 Invoker 为中心，扩展接口为 Cluster、Directory、Router 和 LoadBlancce

监控层（Monitor）：RPC 调用次数和调用时间监控，以 Statistics 为中心， 扩展接口为 MonitorFactory、Monitor 和 MonitorService

远程调用层（Protocal）：封装 RPC 调用，以 Invocation 和 Result 为中心， 扩展接口为 Protocal、Invoker 和 Exporter

信息交换层（Exchange）：封装请求响应模式，同步转异步。以 Request 和 Response 为中心，扩展接口为 Exchanger、ExchangeChannel、 ExchangeClient 和 ExchangeServer

网络传输层（Transport）：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel、Transporter、Client、Server 和 Codec

数据序列化层（Serialize）：可复用的一些工具，扩展接口为 Serialization、 ObjectInput、ObjectOutput 和 ThreadPool Dubbo

Monitor 实现原理?

Consumer 端在发起调用之前会先走 filter 链；provider 端在接收到请求时也 是先走 filter 链，然后才进行真正的业务逻辑处理。默认情况下，在 consumer 和 provider 的 filter 链中都会有 Monitorfilter。

1. MonitorFilter 向 DubboMonitor 发送数据
2. DubboMonitor 将数据进行聚合后（默认聚合 1min 中的统计数据）暂存 到ConcurrentMap statisticsMap，然后使用一个含有 3 个线程（线程名字： DubboMonitorSendTimer）的线程池每隔 1min 钟，调用 SimpleMonitorService 遍历发送 statisticsMap 中的统计数据，每发送完毕一 个，就重置当前的 Statistics 的 AtomicReference

3. SimpleMonitorService 将这些聚合数据塞入 BlockingQueue queue 中 （队列大写为 100000）
4. SimpleMonitorService 使用一个后台线程（线程名为： DubboMonitorAsyncWriteLogThread）将 queue 中的数据写入文件（该线程以死循环的形式来写）
5. SimpleMonitorService 还会使用一个含有 1 个线程（线程名字： DubboMonitorTimer）的线程池每隔 5min 钟，将文件中的统计数据画成图表

分布式框架

Dubbo 类似的分布式框架还有哪些?

比较著名的就是 Spring Cloud。

Dubbo 和 Spring Cloud 有什么关系?

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流 量监控和熔断。而 Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理 的方方面面，另外由于依托了 Spring、Spring Boot 的优势之上，两个框架在 开始目标就不一致，Dubbo 定位服务治理、Spring Cloud 是打造一个生态。

Dubbo 和 Spring Cloud 有什么哪些区别?

Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以

Hession 序列化完成 RPC 通信。

Spring Cloud 是基于 Http 协议 Rest 接口调用远程过程的通信，相对来说

Http 请求会有更大的报文，占的带宽也会更多。但是 REST 相比 RPC 更为灵

活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适，至于注重通信速度还是方便灵活性，具体情况具体考虑。

Dubbo 和 Dubbox 之间的区别?

Dubbox 是继 Dubbo 停止维护后，当当网基于 Dubbo 做的一个扩展项目，如加了服务可 Restful 调用，更新了开源组件等。

注册中心

Dubbo 有哪些注册中心?

- Multicast 注册中心：Multicast 注册中心不需要任何中心节点，只要广播地址，就能进行服务注册和发现,基于网络中组播传输实现。
- Zookeeper 注册中心：基于分布式协调系统 Zookeeper 实现，采用 Zookeeper 的 watch 机制实现数据变更。
- Redis 注册中心：基于 Redis 实现，采用 key/map 存储，key 存储服务名和类型，map 中 key 存储服务 url，value 服务过期时间。基于 Redis 的发布/订阅模式通知数据变更。
- Simple 注册中心。

推荐使用 Zookeeper 作为注册中心

Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么?

可以通讯。启动 Dubbo 时，消费者会从 Zookeeper 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用。

集群

Dubbo集群提供了哪些负载均衡策略?

- Random LoadBalance: 随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀。
- RoundRobin LoadBalance: 轮循选取提供者策略，平均分布，但是存在请求累积的问题。
- LeastActive LoadBalance: 少活跃调用策略，解决慢提供者接收更少的请求。
- ConstantHash LoadBalance: 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动。

默认为 Random 随机调用。

Dubbo的集群容错方案有哪些?

- Failover Cluster：失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。
- Failfast Cluster：快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。
- Failsafe Cluster：失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。
- Failback Cluster：失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。
- Forking Cluster：并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置 大并行数。
- Broadcast Cluster：广播调用所有提供者，逐个调用，任意一台报错则报错 。通常用于通知所有提供者更新缓存或日志等本地资源信息。

默认的容错方案是 Failover Cluster。

配置

Dubbo 配置文件是如何加载到 Spring 中的?

Spring 容器在启动的时候，会读取到 Spring 默认的一些 schema 以及 Dubbo 自定义的 schema，每个 schema 都会对应一个自己的 NamespaceHandler， NamespaceHandler 里面通过 BeanDefinitionParser 来解析配置信息并转化为需要加载的 bean 对象!

说说核心的配置有哪些?

标签	用途	解释
	服务配置	用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心
	引用配置	用于创建一个远程服务代理，一个引用可以指向多个注册中心
	协议配置	用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受
	应用配置	用于配置当前应用信息，不管该应用是提供者还是消费者
	模块配置	用于配置当前模块信息，可选
	模块配置	用于配置当前模块信息，可选
	注册中心配置	用于配置连接注册中心相关信息
	监控中心配置	用于配置连接监控中心相关信息，可选
	提供方配置	当 ProtocolC onfig 和 ServiceCo nfig 某属 性没有配置时，采用此缺省值，可选
	消费方配置	当 Reference Config 某属性没有配置时，采用此缺省值，可选
	方法配置	用于 ServiceCo nfig 和 Reference Config 指 定方法级的配置信息
	参数配置	用于指定方法参数配置

Dubbo 超时设置有哪些方式?

Dubbo 超时设置有两种方式：

- 服务提供者端设置超时时间，在Dubbo的用户文档中，推荐如果能在服务端多配置就尽量多配置，因为服务提供者比消费者更清楚自己提供的服务特性。
- 服务消费者端设置超时时间，如果在消费者端设置了超时时间，以消费者端为主，即优先级更高。因为服务调用方设置超时时间控制性更灵活。如果消费方超时，服务端线程不会定制，会产生警告。

服务调用超时会怎么样?

dubbo 在调用服务不成功时，默认是会重试两次。

通信协议

Dubbo 使用的是什么通信框架?

默认使用 Netty 作为通讯框架。

Dubbo 支持哪些协议，它们的优缺点有哪些?

- Dubbo： 单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步 Hessian 序列化。Dubbo推荐使用 dubbo协议。
- RMI： 采用 JDK 标准的 RMI 协议实现，传输参数和返回参数对象需要实现
- Serializable 接口，使用 Java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。 多个短连接 TCP
- 协议传输，同步传输，适用常规的远程服务调用和 RMI 互操作。在依赖低版本的 Common-Collections 包，Java 序列化存在安全漏洞。
- WebService：基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用。
- HTTP： 基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用。
- Hessian：集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件。
- Memcache：基于 Memcache实现的 RPC 协议。

Redis：基于 Redis 实现的RPC协议。

设计模式

Dubbo 用到哪些设计模式?

Dubbo 框架在初始化和通信过程中使用了多种设计模式，可灵活控制类加载、权限控制等功能。

工厂模式

Provider 在 export 服务时，会调用 ServiceConfig 的 export 方法。

ServiceConfig中有个字段：

```
1   private static final Protocol protocol =
2   ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtensi
3   on();
```

Dubbo 里有很多这种代码。这也是一种工厂模式，只是实现类的获取采用了

JDKSPI 的机制。这么实现的优点是可扩展性强，想要扩展实现，只需要在

classpath下增加个文件就可以了，代码零侵入。另外，像上面的 Adaptive 实现，可以做到调用时动态决定调用哪个实现，但是由于这种实现采用了动态代理，会造成代码调试比较麻烦，需要分析出实际调用的实现类。

装饰器模式

Dubbo 在启动和调用阶段都大量使用了装饰器模式。以 Provider 提供的调用链为例，具体的调用链代码是在 ProtocolFilterWrapper 的

buildInvokerChain 完成的，具体是将注解中含有 group=provider 的 Filter 实现，按照 order 排序， 后的调用顺序是：

```
1 EchoFilter -> ClassLoaderFilter -> GenericFilter -> ContextFilter ->
2 ExecuteLimitFilter -> TraceFilter -> TimeoutFilter -> MonitorFilter ->
3 ExceptionFilter
```

更确切地说，这里是装饰器和责任链模式的混合使用。例如，EchoFilter 的作用是判断是否是回声测试请求，是的话直接返回内容，这是一种责任链的体现。而像ClassLoaderFilter 则只是在主功能上添加了功能，更改当前线程的

ClassLoader，这是典型的装饰器模式。

观察者模式

Dubbo 的 Provider 启动时，需要与注册中心交互，先注册自己的服务，再订阅自己的服务，订阅时，采用了观察者模式，开启一个 listener。注册中心会每 5 秒定时检查是否有服务更新，如果有更新，向该服务的提供者发送一个 notify 消息，provider 接受到 notify 消息后，运行 NotifyListener 的 notify 方法，执行监听器方法。

动态代理模式

Dubbo 扩展 JDK SPI 的类 ExtensionLoader 的 Adaptive 实现是典型的动态代理实现。Dubbo 需要灵活地控制实现类，即在调用阶段动态地根据参数决定调用哪个实现类，所以采用先生成代理类的方法，能够做到灵活的调用。生成代理类的代码是 ExtensionLoader 的 createAdaptiveExtensionClassCode 方法。代理类主要逻辑是，获取 URL 参数中指定参数的值作为获取实现类的key。

运维管理

服务上线怎么兼容旧版本?

可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

Dubbo telnet 命令能做什么?

dubbo 服务发布之后，我们可以利用 telnet 命令进行调试、管理。

Dubbo2.0.5 以上版本服务提供端口支持 telnet 命令

Dubbo 支持服务降级吗?

以通过 dubbo:reference 中设置 mock="return null"。mock 的值也可以

修改为 true，然后再跟接口同一个路径下实现一个 Mock 类，命名规则是 "接口名称+Mock" 后缀。然后在 Mock 类里实现自己的降级逻辑

Dubbo 如何优雅停机?

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果使用kill

-9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

SPI

Dubbo SPI 和 Java SPI 区别?

JDK SPI：

JDK 标准的 SPI 会一次性加载所有的扩展实现，如果有的扩展很耗时，但也没用上，很浪费资源。所以只希望加载某个的实现，就不现实了

DUBBO SPI：

1、 对 Dubbo 进行扩展，不需要改动 Dubbo 的源码

2、 延迟加载，可以一次只加载自己想要加载的扩展实现。

3、 增加了对扩展点 IOC 和 AOP 的支持，一个扩展点可以直接 setter 注入其它扩展点。

4、 Dubbo 的扩展机制能很好的支持第三方 IoC 容器，默认支持 Spring

Bean。其他

Dubbo 支持分布式事务吗?

目前暂时不支持，可与通过 tcc-transaction 框架实现介绍：tcc-transaction 是开源的 TCC 补偿性分布式事务框架

TCC-Transaction 通过 Dubbo 隐式传参的功能，避免自己对业务代码的入侵。

Dubbo 可以对结果进行缓存吗?

为了提高数据访问的速度。Dubbo 提供了声明式缓存，以减少用户加缓存的工作量

其实比普通的配置文件就多了一个标签 cache="true"

Dubbo 必须依赖的包有哪些?

Dubbo 必须依赖 JDK，其他为可选。

Dubbo 支持哪些序列化方式?

默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。

Dubbo 在安全方面有哪些措施?

- Dubbo 通过 Token 令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。
- Dubbo 还提供服务黑白名单，来控制服务所允许的调用方。

服务调用是阻塞的吗?

默认是阻塞的，可以异步调用，没有返回值的可以这么做。Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小，异步调用会返回一个 Future 对象。

服务提供者能实现失效踢出是什么原理?

服务失效踢出基于 zookeeper 的临时节点原理。

同一个服务多个注册的情况下可以直()连某一个服务吗?

可以点对点直连，修改配置即可，也可以通过 telnet 直接某个服务。

Dubbo 服务降级，失败重试怎么做?

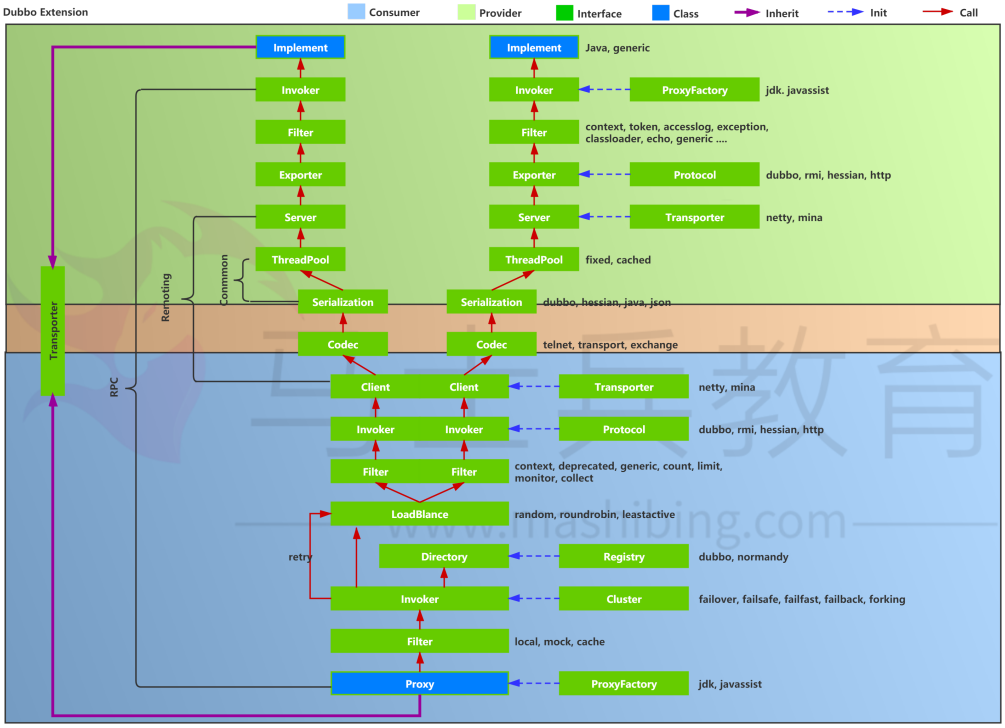
可以通过 dubbo:reference 中设置 mock="return null"。mock 的值也可以修改为 true，然后再跟接口同一个路径下实现一个 Mock 类，命名规则是

"接口名称+Mock" 后缀。然后在 Mock 类里实现自己的降级逻辑

Dubbo 使用过程中都遇到了些什么问题?

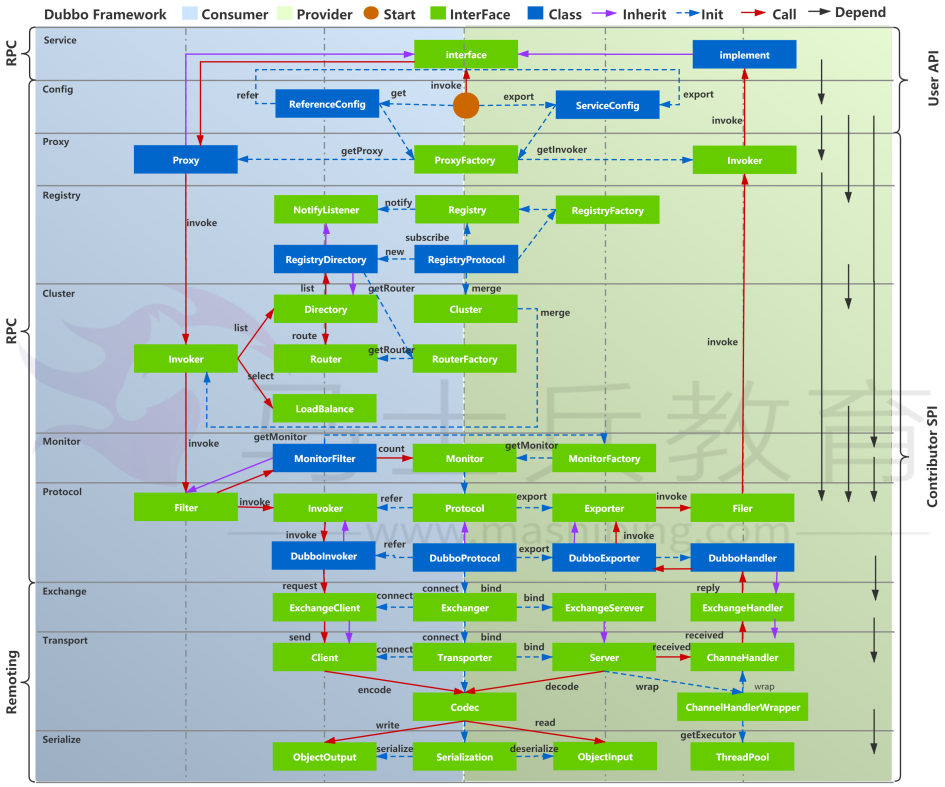
在注册中心找不到对应的服务,检查 service 实现类是否添加了@service 注解无法连接到注册中心,检查配置文件中的对应的测试 ip 是否正确

Dubbo完整的一次调用链路介绍;



框架设计

整体设计



图例说明：

- 图中左边淡蓝背景的为服务消费方使用的接口，右边淡绿色背景的为服务提供方使用的接口，位于中轴线上的为双方都用到的接口。
- 图中从下至上分为十层，各层均为单向依赖，右边的黑色箭头代表层之间的依赖关系，每一层都可以剥离上层被复用，其中，Service 和 Config 层为 API，其它各层均为 SPI。
- 图中绿色小块的为扩展接口，蓝色小块为实现类，图中只显示用于关联各层的实现类。
- 图中蓝色虚线为初始化过程，即启动时组装链，红色实线为方法调用过程，即运行时调时链，紫色三角箭头为继承，可以把子类看作父类的同一个节点，线上的文字为调用的方法。

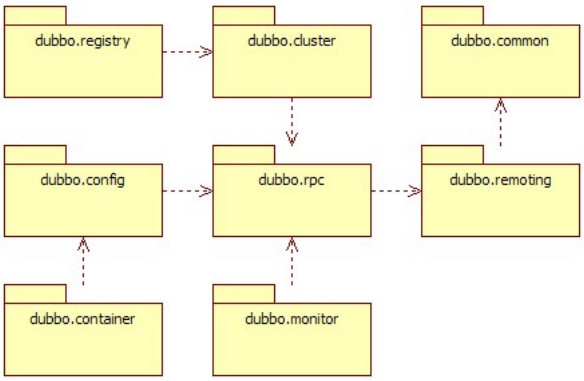
各层说明

- **config 配置层**：对外配置接口，以 `ServiceConfig`, `ReferenceConfig` 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类
- **proxy 服务代理层**：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton, 以 `ServiceProxy` 为中心，扩展接口为 `ProxyFactory`
- **registry 注册中心层**：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 `RegistryFactory`, `Registry`, `RegistryService`
- **cluster 路由层**：封装多个提供者的路由及负载均衡，并桥接注册中心，以 `Invoker` 为中心，扩展接口为 `Cluster`, `Directory`, `Router`, `LoadBalance`
- **monitor 监控层**：RPC 调用次数和调用时间监控，以 `Statistics` 为中心，扩展接口为 `MonitorFactory`, `Monitor`, `MonitorService`
- **protocol 远程调用层**：封装 RPC 调用，以 `Invocation`, `Result` 为中心，扩展接口为 `Protocol`, `Invoker`, `Exporter`
- **exchange 信息交换层**：封装请求响应模式，同步转异步，以 `Request`, `Response` 为中心，扩展接口为 `Exchanger`, `ExchangeChannel`, `ExchangeClient`, `ExchangeServer`
- **transport 网络传输层**：抽象 mina 和 netty 为统一接口，以 `Message` 为中心，扩展接口为 `Channel`, `Transporter`, `Client`, `Server`, `Codec`
- **serialize 数据序列化层**：可复用的一些工具，扩展接口为 `Serialization`, `ObjectInput`, `ObjectOutput`, `ThreadPool`

关系说明

- 在 RPC 中，Protocol 是核心层，也就是只要有 Protocol + Invoker + Exporter 就可以完成非透明的 RPC 调用，然后在 Invoker 的主过程上 Filter 拦截点。
- 图中的 Consumer 和 Provider 是抽象概念，只是想让看图者更直观的了解哪些类分属于客户端与服务器端，不用 Client 和 Server 的原因是 Dubbo 在很多场景下都使用 Provider, Consumer, Registry, Monitor 划分逻辑拓普节点，保持统一概念。
- 而 Cluster 是外围概念，所以 Cluster 的目的是将多个 Invoker 伪装成一个 Invoker，这样其它人只要关注 Protocol 层 Invoker 即可，加上 Cluster 或者去掉 Cluster 对其它层都不会造成影响，因为只有一个提供者时，是不需要 Cluster 的。
- Proxy 层封装了所有接口的透明化代理，而在其它层都以 Invoker 为中心，只有到了暴露给用户使用时，才用 Proxy 将 Invoker 转成接口，或将接口实现转成 Invoker，也就是去掉 Proxy 层 RPC 是可以 Run 的，只是不那么透明，不那么看起来像调本地服务一样调远程服务。
- 而 Remoting 实现是 Dubbo 协议的实现，如果你选择 RMI 协议，整个 Remoting 都不会用上，Remoting 内部再划为 Transport 传输层和 Exchange 信息交换层，Transport 层只负责单向消息传输，是对 Mina, Netty, Grizzly 的抽象，它也可以扩展 UDP 传输，而 Exchange 层是在传输层之上封装了 Request-Response 语义。
- Registry 和 Monitor 实际上不算一层，而是一个独立的节点，只是为了全局概览，用层的方式画在一起。

模块分包



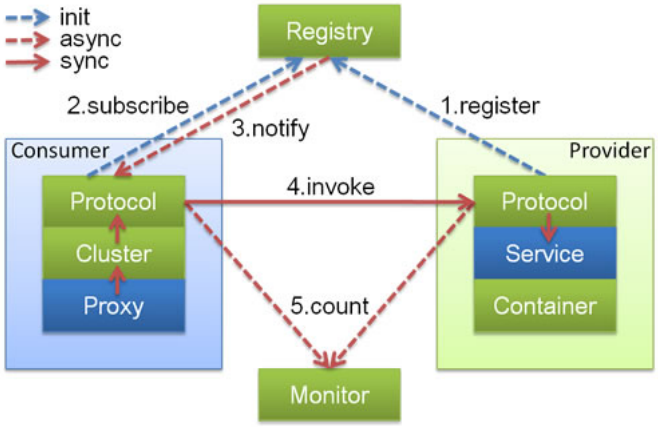
模块说明：

- **dubbo-common 公共逻辑模块**：包括 Util 类和通用模型。
- **dubbo-remoting 远程通讯模块**：相当于 Dubbo 协议的实现，如果 RPC 用 RMI协议则不需要使用此包。
- **dubbo-rpc 远程调用模块**：抽象各种协议，以及动态代理，只包含一对一的调用，不关心集群的管理。
- **dubbo-cluster 集群模块**：将多个服务提供方伪装为一个提供方，包括：负载均衡, 容错，路由等，集群的地址列表可以是静态配置的，也可以是由注册中心下发。
- **dubbo-registry 注册中心模块**：基于注册中心下发地址的集群方式，以及对各种注册中心的抽象。
- **dubbo-monitor 监控模块**：统计服务调用次数，调用时间的，调用链跟踪的服务。
- **dubbo-config 配置模块**：是 Dubbo 对外的 API，用户通过 Config 使用Dubbo，隐藏 Dubbo 所有细节。
- **dubbo-container 容器模块**：是一个 Standlone 的容器，以简单的 Main 加载 Spring 启动，因为服务通常不需要 Tomcat/JBoss 等 Web 容器的特性，没必要用 Web 容器去加载服务。

整体上按照分层结构进行分包，与分层的不同点在于：

- container 为服务容器，用于部署运行服务，没有在层中画出。
- protocol 层和 proxy 层都放在 rpc 模块中，这两层是 rpc 的核心，在不需要集群也就是只有一个提供者时，可以只使用这两层完成 rpc 调用。
- transport 层和 exchange 层都放在 remoting 模块中，为 rpc 调用的通讯基础。
- serialize 层放在 common 模块中，以便更大程度复用。

依赖关系

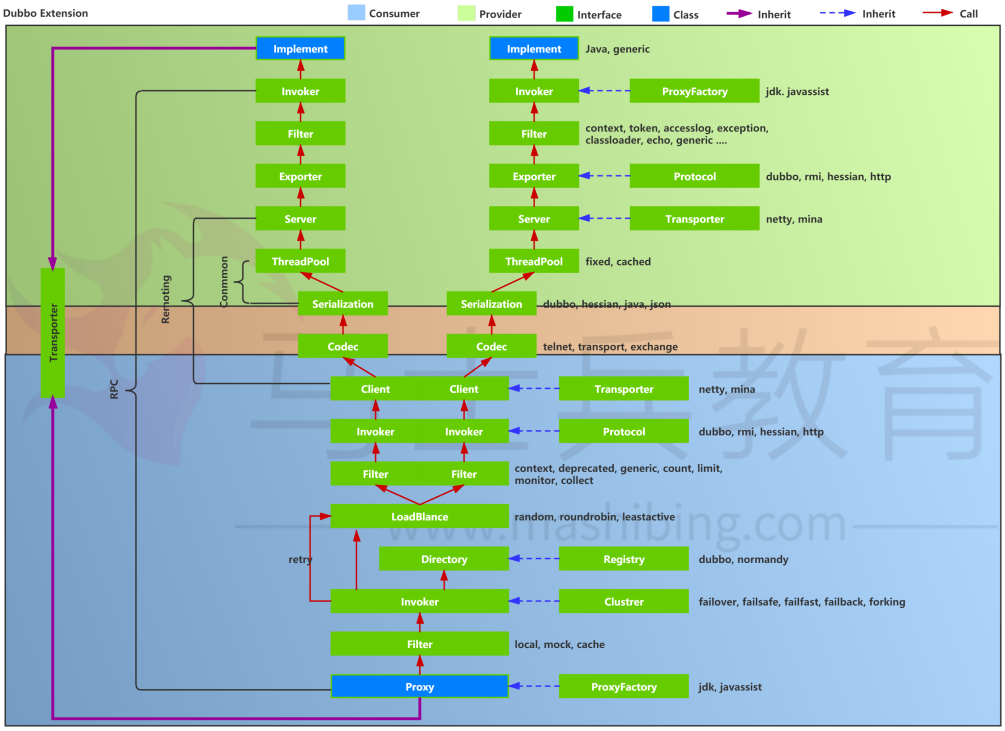


图例说明：

- 图中小方块 Protocol, Cluster, Proxy, Service, Container, Registry, Monitor 代表层或模块，蓝色的表示与业务有交互，绿色的表示只对 Dubbo 内部交互。
- 图中背景方块 Consumer, Provider, Registry, Monitor 代表部署逻辑拓扑节点。
- 图中蓝色虚线为初始化时调用，红色虚线为运行时异步调用，红色实线为运行时同步调用。
- 图中只包含 RPC 的层，不包含 Remoting 的层，Remoting 整体都隐含在 Protocol 中。

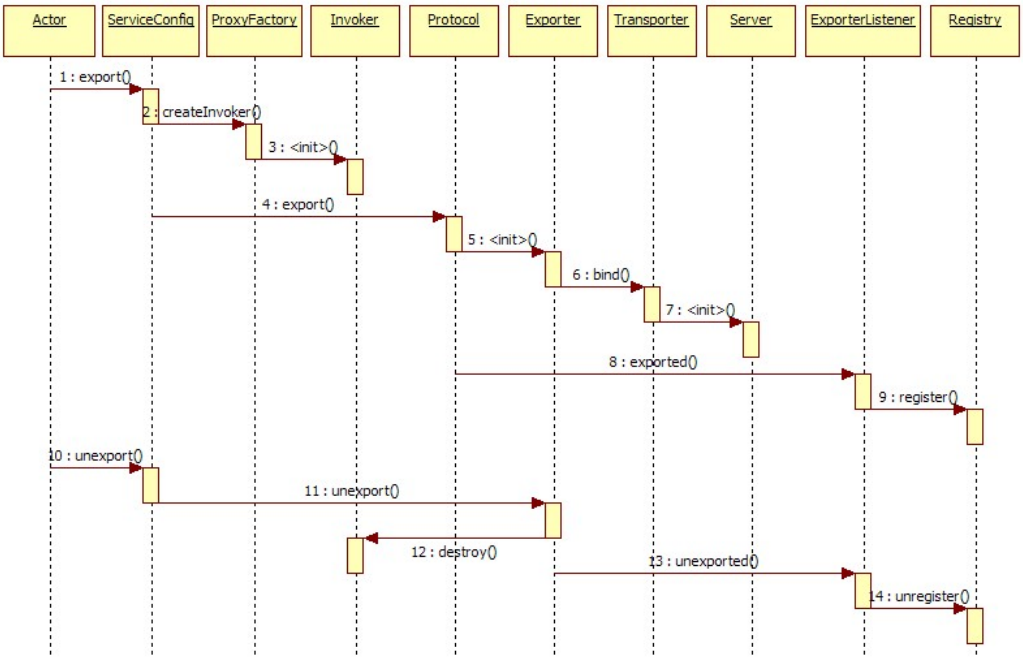
调用链

展开总设计图的红色调用链，如下：



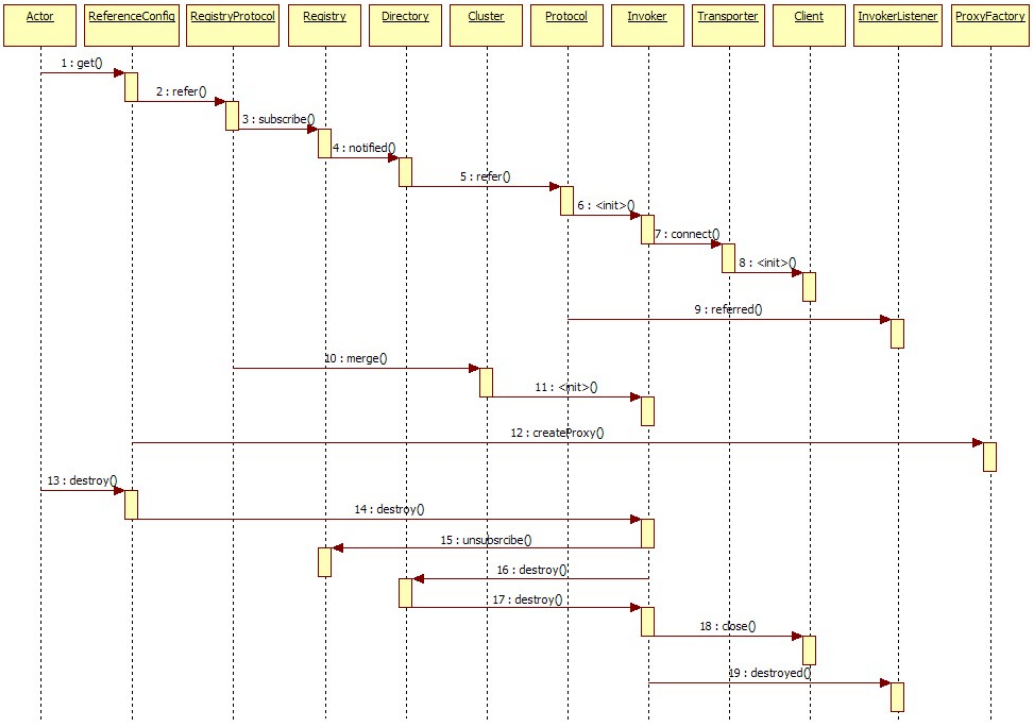
暴露服务时序

展开总设计图左边服务提供方暴露服务的蓝色初始化链，时序图如下：



引用服务时序

展开总设计图右边服务消费方引用服务的蓝色初始化链，时序图如下：



领域模型

在 Dubbo 的核心领域模型中：

- Protocol 是服务域，它是 Invoker 暴露和引用的主功能入口，它负责 Invoker 的生命周期管理。
- Invoker 是实体域，它是 Dubbo 的核心模型，其它模型都向它靠拢，或转换成它，它代表一个可执行体，可向它发起 invoke 调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。
- Invocation 是会话域，它持有调用过程中的变量，比如方法名，参数等。

基本设计原则

- 采用 Microkernel + Plugin 模式，Microkernel 只负责组装 Plugin，Dubbo 自身的功能也是通过扩展点实现的，也就是 Dubbo 的所有功能点都可被用户自定义扩展所替换。
- 采用 URL 作为配置信息的统一格式，所有扩展点都通过传递 URL 携带配置信息。

Dubbo支持几种负载均衡策略?

1）Random LoadBalance

随机，按权重设置随机概率。

在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有 利于动态调整提供者权重。

2）RoundRobin LoadBalance

轮询，按公约后的权重设置轮询比率。

存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在 那，久而久之，所有请求都卡在调到第二台上。

3）LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。

使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

4）ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发到同一提供者。

当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起 剧烈变动。

缺省只对第一个参数 Hash，如果要修改，请配置 <dubbo:parameter key="hash.arguments" value="0,1" />

缺省用 160 份虚拟节点，如果要修改，请配置 <dubbo:parameter key="hash.nodes" value="320" />

负载均衡

在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 `random` 随机调用。

可以自行扩展负载均衡策略

负载均衡策略

Random LoadBalance

- **随机**，按权重设置随机概率。
- 在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

RoundRobin LoadBalance

- **轮询**，按公约后的权重设置轮询比率。
- 存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

LeastActive LoadBalance

- **最少活跃调用数**，相同活跃数的随机，活跃数指调用前后计数差。
- 使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

ConsistentHash LoadBalance

- **一致性 Hash**，相同参数的请求总是发到同一提供者。
- 当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。
- 缺省只对第一个参数 Hash，如果要修改，请配置 `<dubbo:parameter key="hash.arguments" value="0,1" />`
- 缺省用 160 份虚拟节点，如果要修改，请配置 `<dubbo:parameter key="hash.nodes" value="320" />`

配置

服务端服务级别

```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

客户端服务级别

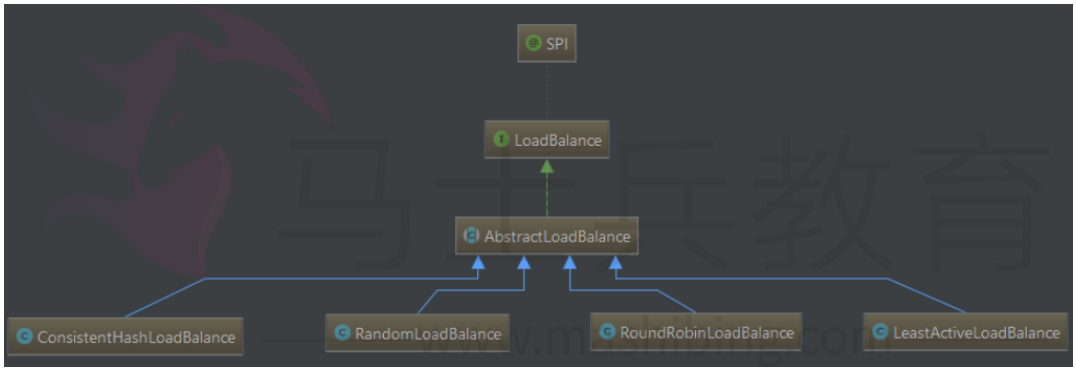
```
<dubbo:reference interface="..." loadbalance="roundrobin" />
```

服务端方法级别

```
<dubbo:service interface="...">
    <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:service>
```

客户端方法级别

```
<dubbo:reference interface="...">
    <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:reference>
```



dubbo负载均衡策略

前言：在上一篇博客中，介绍了zookeeper作为dubbo的注册中心是如何工作的，有一个很重要的点，我们的程序是分布式应用，服务部署在几个节点（服务器）上，当消费者调用服务时，zk返回给dubbo的是一个节点列表，但是dubbo只会选择一台服务器，那么它究竟会选择哪一台呢？这就是dubbo的负载均衡策略了，本篇博客就来聚焦dubbo的负载均衡策略。

本篇博客的目录

一：负载均衡介绍

1.1:负载均衡简介

以下是wikipedia对负载均衡的定义：

负载均衡改善了跨多个计算资源（例如计算机，计算机集群，网络链接，中央处理单元或磁盘驱动的的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间，并避免任何单个资源的过载。使用具有负载平衡而不是单个组件的多个组件可以通过冗余提高可靠性和可用性。负载平衡通常涉及专用软件或硬件

1.2：简单解释

这个概念如何理解呢？通俗点来说假如一个请求从客户端发起,比如(查询订单列表)，要选择服务器进行处理，但是我们的集群环境提供了5个服务器A\B\C\D\E，每个服务器都有处理这个请求的能力,此时客户端就必须选择一个服务器来进行处理(**不存在先选择A，处理一会又选择C,又跳到D**).说白了就是一个选择的问题。当请求多了的话，就要考虑各服务器的负载，一共5个服务器，不可能每次都让一个服务器都来处理吧，比如把让其他服务器来分压。这就是负载均衡的优点：避免单个服务器响应同一请求，容易造成服务器宕机、崩溃等问题。

二：dubbo的loadBalance接口

1.1：loadBalance

dubbo的负载均衡策略，主体向外暴露出来是一个接口，名字叫做loadBlace,位于com.alibaba.dubbo.rpc.cluster包下，很明显根据包名就可以看出它是用来管理集群的：

这个接口就一个方法，select方法的作用就是从众多的调用的List选择出一个调用者，Invoker可以理解为客户端的调用者，dubbo专门封装一个类来表示，URL就是调用者发起的URL请求链接，从这个URL中可以获取很多请求的具体信息,Invocation表示的是调用的具体过程，dubbo用这个类模拟调用具体细节过程：

1.2：AbstractLoadBlance

该接口在下面的子类都会对其进行实现。接口下是一个抽象类AbstractLoadBlance

```
package com.alibaba.dubbo.rpc.cluster;
public interface LoadBalance {@Adaptive("loadbalance")
    <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) throws RpcException;
}
```

AbstractLoadBlance抽象类继承自LoadBalance，其中有个static方法表明它在类加载的时候就会运行，它表示的含义是计算预热加载权重，参数是uptime，这里可以理解为服务启动的时间，warmup就是预热时间，weight是权重的值，下面会对比进行详细解释：

```
public abstract class AbstractLoadBalance implements LoadBalance{

   static int calculateWarmupWeight(int uptime, int warmup, int weight){
    //
  }
    @Override
    public <T> Invoker<T> select(List<Invoker<T>> invokers, URL url,
   Invocation invocation){
   //
   }
    protected abstract <T> Invoker<T> doSelect(List<Invoker<T>>
     invokers, URL url, Invocation invocation);
     protected int getWeight(Invoker<?> invoker, Invocation invocation) {
    }
}
```

1.2.1：select方法

抽象类方法中有个有方法体的方法select,先判断调用者组成的List是否为null，如果是null就返回null。再判断调用者的大小，如果只有一个就返回那个唯一的调用者(试想，如果服务调用另一个服务时，当服务的提供者机器只有一个，那么就可以返回那一个，因为没有选择了！)如果这些都不成立，就继续往下走，走doSelect方法：

```
@Override
    public <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        if (invokers == null || invokers.isEmpty())
            return null;
        if (invokers.size() == 1)
            return invokers.get(0);
        return doSelect(invokers, url, invocation);
    }
```

1.2.2:doSelect方法

该方法是抽象的，交给具体的子类去实现，由此也可以思考出一个问题就是：dubbo为什么要将一个接口首先做出一个实现抽象类，再由不同的子类去实现？原因是抽象类中的非抽象方法，再子类中都是必须要实现的，而他们子类的不同点就是具体做出选择的策略不同,将公共的逻辑提取出来放在抽象类里，子类不用写多余的代码，只用维护和实现最终要的自己的逻辑

```
 protected abstract <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation);
```

1.2.3：getWeight方法

顾名思义，这个方法的含义就是获取权重，首先通过URL(URL为dubbo封装的一个实体)获取基本的权重，如果权重大于0，会获取服务启动时间，再用当前的时间-启动时间就是服务到目前为止运行了多久，因此这个upTime就可以理解为服务启动时间，再获取配置的预热时间，如果启动时间小于预热时间，就会再次调用获取权重。这个预热的方法其实dubbo针对JVM做出的一个**很契合的优化**，因为JVM从启动到起来都运行到最佳状态是需要一点时间的，这个时间叫做warmup,而dubbo就会对这个时间进行设定，然后等到服务运行时间和warmup相等时再计算权重，这样就可以保障服务的最佳运行状态！

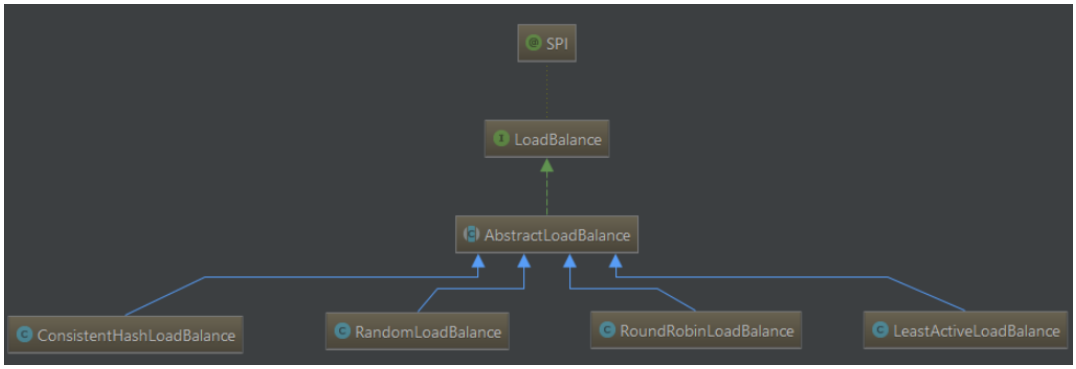
```
protected int getWeight(Invoker<?> invoker, Invocation invocation) {
        int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.WEIGHT_KEY,
Constants.DEFAULT_WEIGHT);
        if (weight > 0) {
            long timestamp = invoker.getUrl().getParameter(Constants.REMOTE_TIMESTAMP_KEY, 0L);
            if (timestamp > 0L) {
                int uptime = (int) (System.currentTimeMillis() - timestamp);
                int warmup = invoker.getUrl().getParameter(Constants.WARMUP_KEY, Constants.DEFAULT_WARMUP);
```

```
                if (uptime > 0 && uptime < warmup) {
                    weight = calculateWarmupWeight(uptime, warmup, weight);
                }
            }
        }
        return weight;
    }
```

三：dubbo的几种负载均衡策略

3.1：整体架构图

可以看出抽象的负载均衡下的类分为4个，这4个类表示了4种负载均衡策略，分别是一致性Hash均衡算法、随机调用法、轮询法、最少活动调用法



3.2:RandomLoadBalance

随机调用负载均衡，该类实现了抽象的AbstractLoadBalance接口,重写了doSelect方法，看方法的细节就是首先遍历每个提供服务的机器，获取每个服务的权重，然后累加权重值，判断每个服务的提供者权重是否相同，如果每个调用者的权重不相同，并且每个权重大于0，那么就会根据权重的总值生成一个随机数，再用这个随机数，根据调用者的数量每次减去调用者的权重，直到计算出当前的服务提供者随机数小于0，就选择那个提供者！另外，如果每个机器的权重的都相同，那么权重就不会参与计算，直接选择随机算法生成的某一个选择，完全随机。可以看出，随机调用法，

```
public class RandomLoadBalance extends AbstractLoadBalance {

    public static final String NAME = "random";

    private final Random random = new Random();

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        int length = invokers.size(); // Number of invokers
        int totalWeight = 0; // The sum of weights
        boolean sameWeight = true; // Every invoker has the same weight?
        for (int i = 0; i < length; i++) {
            int weight = getWeight(invokers.get(i), invocation);
            totalWeight += weight; // Sum
            if (sameWeight && i > 0
                    && weight != getWeight(invokers.get(i - 1), invocation)) {
                sameWeight = false;
            }
        }
        if (totalWeight > 0 && !sameWeight) {
            // If (not every invoker has the same weight & at least one invoker's weight>0), select randomly
based on totalWeight.
            int offset = random.nextInt(totalWeight);
            // Return a invoker based on the random value.
            for (int i = 0; i < length; i++) {
                offset -= getWeight(invokers.get(i), invocation);
                if (offset < 0) {
                    return invokers.get(i);
                }
            }
        }
        // If all invokers have the same weight value or totalWeight=0, return evenly.
        return invokers.get(random.nextInt(length));
    }
```

3.3：RoundRobinLoadBlance

轮询调用，轮询调用的过程主要是维护了局部变量的一个LinkdesHashMap（有顺序的Map）去存储调用者和权重值的对应关系，然后遍历每个调用者,把调用者和当前大于0的权重值放进去，再累加权重值。还有一个全局变量的map，找到第一个服务调用者，首先是找到每个服务的key值和method，这里可以理解为标识第一个调用者的唯一key，然后再给它对应的值保证原子性的+1（AtomicPositiveInteger是原子的），再对这个值取模总权重，再每次对其权重值-1，知道它取模与总权重值等于0就选择该调用者，可以称之为**"降权取模"**（只是一种的计算层面,而不是真正降权）。总结：**轮询调用并不是简单的一个接着一个依次调用，它是根据权重的值进行循环的。**

```
public class RoundRobinLoadBalance extends AbstractLoadBalance {

    public static final String NAME = "roundrobin";
```

```
    private final ConcurrentMap<String, AtomicPositiveInteger> sequences = new ConcurrentHashMap<String,
AtomicPositiveInteger>();

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();
        int length = invokers.size(); // Number of invokers
        int maxWeight = 0; // The maximum weight
        int minWeight = Integer.MAX_VALUE; // The minimum weight
        final LinkedHashMap<Invoker<T>, IntegerWrapper> invokerToWeightMap = new LinkedHashMap<Invoker<T>,
IntegerWrapper>();
        int weightSum = 0;
        for (int i = 0; i < length; i++) {
            int weight = getWeight(invokers.get(i), invocation);
            maxWeight = Math.max(maxWeight, weight); // Choose the maximum weight
            minWeight = Math.min(minWeight, weight); // Choose the minimum weight
            if (weight > 0) {
                invokerToWeightMap.put(invokers.get(i), new IntegerWrapper(weight));
                weightSum += weight;
            }
        }
        AtomicPositiveInteger sequence = sequences.get(key);
        if (sequence == null) {
            sequences.putIfAbsent(key, new AtomicPositiveInteger());
            sequence = sequences.get(key);
        }
        int currentSequence = sequence.getAndIncrement();
        if (maxWeight > 0 && minWeight < maxWeight) {
            int mod = currentSequence % weightSum;
            for (int i = 0; i < maxWeight; i++) {
                for (Map.Entry<Invoker<T>, IntegerWrapper> each : invokerToWeightMap.entrySet()) {
                    final Invoker<T> k = each.getKey();
                    final IntegerWrapper v = each.getValue();
                    if (mod == 0 && v.getValue() > 0) {
                        return k;
                    }
                    if (v.getValue() > 0) {
                        v.decrement();
                        mod--;
                    }
                }
            }
        }
        // Round robin
        return invokers.get(currentSequence % length);
    }
```

2.4：LeastActiveLoadBlance

最少活跃数调用法：这个方法的主要作用根据服务的提供者的运行状态去选择服务器,主要的思路就是遍历每个调用者，然后获取每个服务器的运行状态，如果当前运行的运行状态小于最小的状态-1，把它保存在leastIndexs中的第一个位置，并且认定所有的调用者权重都相同，然后直接返回那个调用者(这里的逻辑是：找到最少活跃数(在代码层反应就是：active的值))。如果计算出的权重值和最少的权重值相同，那么把它保存在leastIndexs数组里面，累加权重值，如果当前的权重值不等于初始值firstWeight，那么就认定不是所有的调用者的权重不同。然后再遍历lestIndexs，取权重累加值的随机数生成权重偏移量，在累减它，到它小于0的时候返回那个调用者。如果这些都不符合，就从leastIndexs随机选一个index，返回那个调用者！

```
public class LeastActiveLoadBalance extends AbstractLoadBalance {

    public static final String NAME = "leastactive";

    private final Random random = new Random();

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        int length = invokers.size(); // Number of invokers
        int leastActive = -1; // The least active value of all invokers
        int leastCount = 0; // The number of invokers having the same least active value (leastActive)
        int[] leastIndexs = new int[length]; // The index of invokers having the same least active value
(leastActive)
        int totalWeight = 0; // The sum of weights
        int firstWeight = 0; // Initial value, used for comparision
        boolean sameWeight = true; // Every invoker has the same weight value?
        for (int i = 0; i < length; i++) {
            Invoker<T> invoker = invokers.get(i);
            int active = RpcStatus.getStatus(invoker.getUrl(), invocation.getMethodName()).getActive(); //
Active number
            int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.WEIGHT_KEY,
Constants.DEFAULT_WEIGHT); // Weight
            if (leastActive == -1 || active < leastActive) { // Restart, when find a invoker having smaller
least active value.
                leastActive = active; // Record the current least active value
                leastCount = 1; // Reset leastCount, count again based on current leastCount
                leastIndexs[0] = i; // Reset
                totalWeight = weight; // Reset
                firstWeight = weight; // Record the weight the first invoker
                sameWeight = true; // Reset, every invoker has the same weight value?
```

```
            } else if (active == leastActive) { // If current invoker's active value equals with leaseActive,
then accumulating.
                leastIndexs[leastCount++] = i; // Record index number of this invoker
                totalWeight += weight; // Add this invoker's weight to totalWeight.
                // If every invoker has the same weight?
                if (sameWeight && i > 0
                        && weight != firstWeight) {
                    sameWeight = false;
                }
            }
        }
        // assert(leastCount > 0)
        if (leastCount == 1) {
            // If we got exactly one invoker having the least active value, return this invoker directly.
            return invokers.get(leastIndexs[0]);
        }
        if (!sameWeight && totalWeight > 0) {
            // If (not every invoker has the same weight & at least one invoker's weight>0), select randomly
based on totalWeight.
            int offsetWeight = random.nextInt(totalWeight);
            // Return a invoker based on the random value.
            for (int i = 0; i < leastCount; i++) {
                int leastIndex = leastIndexs[i];
                offsetWeight -= getWeight(invokers.get(leastIndex), invocation);
                if (offsetWeight <= 0)
                    return invokers.get(leastIndex);
            }
        }
        // If all invokers have the same weight value or totalWeight=0, return evenly.
        return invokers.get(leastIndexs[random.nextInt(leastCount)]);
    }
}
```

2.2.5：ConsistentHashLoadBalance

一致性Hash算法，doSelect方法进行选择。一致性Hash负载均衡涉及到两个主要的配置参数为hash.arguments与hash.nodes：当进行调用时候根据调用方法的哪几个参数生成key，并根据key来通过一致性hash算法来选择调用节点。例如调用方法invoke(Strings1,Strings2);若hash.arguments为1(默认值)，则仅取invoke的参数1（s1）来生成hashCode。

hash.nodes：节点的副本数。。dubbo的一致性哈希通过ConsistentHashLoadBalance类来实现。ConsistentHashLoadBalance内部定义ConsistentHashSelector类，最终通过该类进行结点选择。ConsistentHashLoadBalance实现的doSelect方法来利用所创建的ConsistentHashSelector对象选择结点。doSelect的实现如下。当调用该方法时，如果选择器不存在则去创建。随后通过ConsistentHashSelector的select方法选择结点。ConsistentHashSelector在构造函数内部会创建replicaNumber个虚拟结点，并将这些虚拟结点存储于TreeMap。随后根据调用方法的参数来生成key，并在TreeMap中选择一个结点进行调用。上述代码中hash(byte[]digest,intnumber)方法用来生成hashCode。该函数将生成的结果转换为long类，这是因为生成的结果是一个32位数，若用int保存可能会产生负数。而一致性hash生成的逻辑环其hashCode的范围是在0-MAX_VALUE之间。因此为正整数，所以这里要强制转换为long类型，避免出现负数。进行节点选择的方法为select,最后通过sekectForKey方法来选择结点。在进行选择时候若HashCode直接与某个虚拟结点的key一样，则直接返回该结点，如果hashCode落在某个节点上。若不在，找到一个最小上个的key所对应的结点。

```
public class ConsistentHashLoadBalance extends AbstractLoadBalance {

    private final ConcurrentMap<String, ConsistentHashSelector<?>> selectors = new ConcurrentHashMap<String,
ConsistentHashSelector<?>>();

    @SuppressWarnings("unchecked")
    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {
        String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();
        int identityHashCode = System.identityHashCode(invokers);
        ConsistentHashSelector<T> selector = (ConsistentHashSelector<T>) selectors.get(key);        //若选择器不
存在去创建
        if (selector == null || selector.identityHashCode != identityHashCode) {
            selectors.put(key, new ConsistentHashSelector<T>(invokers, invocation.getMethodName(),
identityHashCode));
            selector = (ConsistentHashSelector<T>) selectors.get(key);
        }
        return selector.select(invocation);
    }
    //私有内部类
    private static final class ConsistentHashSelector<T> {

        private final TreeMap<Long, Invoker<T>> virtualInvokers;

        private final int replicaNumber;

        private final int identityHashCode;

        private final int[] argumentIndex;

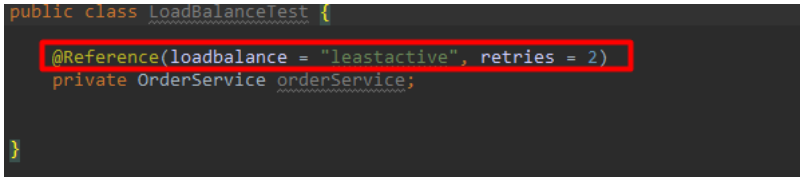
        ConsistentHashSelector(List<Invoker<T>> invokers, String methodName, int identityHashCode) {
            this.virtualInvokers = new TreeMap<Long, Invoker<T>>();
            this.identityHashCode = identityHashCode;
            URL url = invokers.get(0).getUrl();
            this.replicaNumber = url.getMethodParameter(methodName, "hash.nodes", 160);
```

```
            String[] index = Constants.COMMA_SPLIT_PATTERN.split(url.getMethodParameter(methodName,
"hash.arguments", "0"));
            argumentIndex = new int[index.length];
            for (int i = 0; i < index.length; i++) {
                argumentIndex[i] = Integer.parseInt(index[i]);
            }
            for (Invoker<T> invoker : invokers) {
                String address = invoker.getUrl().getAddress();
                for (int i = 0; i < replicaNumber / 4; i++) {
                    byte[] digest = md5(address + i);
                    for (int h = 0; h < 4; h++) {
                        long m = hash(digest, h);                          //虚拟调用者
                        virtualInvokers.put(m, invoker);
                    }
                }
            }
        }
         //选择调用
        public Invoker<T> select(Invocation invocation) {
            String key = toKey(invocation.getArguments());
            byte[] digest = md5(key);
            return selectForKey(hash(digest, 0));
        }
        //转化为服务的key值
        private String toKey(Object[] args) {
            StringBuilder buf = new StringBuilder();
            for (int i : argumentIndex) {
                if (i >= 0 && i < args.length) {
                    buf.append(args[i]);
                }
            }
            return buf.toString();
        }
        //
        private Invoker<T> selectForKey(long hash) {            //从TreeMap中去寻找
            Map.Entry<Long, Invoker<T>> entry = virtualInvokers.tailMap(hash, true).firstEntry();
            if (entry == null) {
                entry = virtualInvokers.firstEntry();
            }
            return entry.getValue();
        }
        //计算Hash值
        private long hash(byte[] digest, int number) {
            return (((long) (digest[3 + number * 4] & 0xFF) << 24)
                    | ((long) (digest[2 + number * 4] & 0xFF) << 16)
                    | ((long) (digest[1 + number * 4] & 0xFF) << 8)
                    | (digest[number * 4] & 0xFF))
                    & 0xFFFFFFFFL;
        }
        //md5加密
        private byte[] md5(String value) {
            MessageDigest md5;
            try {
                md5 = MessageDigest.getInstance("MD5");
            } catch (NoSuchAlgorithmException e) {
                throw new IllegalStateException(e.getMessage(), e);
            }
            md5.reset();
            byte[] bytes;
            try {
                bytes = value.getBytes("UTF-8");
            } catch (UnsupportedEncodingException e) {
                throw new IllegalStateException(e.getMessage(), e);
            }
            md5.update(bytes);
            return md5.digest();
        }
    }

}
```

三：dubbo的默认负载均衡策略

3.1：由@SPI注解可以看到，dubbo默认的负载均衡策略是随机调用法



3.2：如何改变dubbo的负载均衡策略?

3.2.1：如果是springboot项目,直接注解在@Reference中引用，然后注明loadblance="xx".其中xx为每个实现类中的name的值

3.2.2:xml配置的方式

```
<dubbo:serviceinterface="..."loadbalance="roundrobin"/>
```

四：总结

本篇博客讲述了dubbo的负载均衡机制，其中可以看到除了一致性Hash算法，其它都是根据权重进行计算的，在实际的分布式应用中,理解dubbo如何与zookeeper进行通信选择，如何实现负载均衡，如何维护服务的高可用性,理解负载均衡对于微服务的重要意义，将对于我们学习分布式的开发起着推波助澜的作用。

Dubbo Provider服务提供者要控制执行并发请求上限，具体怎么做

服务端并发限流：executes

客户端并发限流：actives

样例 1

限制 com.foo.BarService 的每个方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" executes="10" />
```

样例 2

限制 com.foo.BarService 的 sayHello 方法，服务器端并发执行（或占用线程池线程数）不能 超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" executes="10" />
</dubbo:service>
```

样例 3 限制 com.foo.BarService 的每个方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" actives="10" />
```

或

```
 <dubbo:reference interface="com.foo.BarService" actives="10" />
```

样例 4

限制 com.foo.BarService 的 sayHello 方法，每客户端并发执行（或占用连接的请求数）不能 超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

或

```
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

并发控制

配置样例

样例 1

限制 `com.foo.BarService` 的每个方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" executes="10" />
```

样例 2

限制 `com.foo.BarService` 的 `sayHello` 方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" executes="10" />
</dubbo:service>
```

样例 3

限制 `com.foo.BarService` 的每个方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" actives="10" />
```

或

```
<dubbo:reference interface="com.foo.BarService" actives="10" />
```

样例 4

限制 `com.foo.BarService` 的 `sayHello` 方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

或

```
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

如果 `<dubbo:service>` 和 `<dubbo:reference>` 都配了actives， `<dubbo:reference>` 优先

配置服务的客户端的 `loadbalance` 属性为 `leastactive`，此 Loadbalance 会调用并发数最小的 Provider（Consumer端并发数）。

```
<dubbo:reference interface="com.foo.BarService" loadbalance="leastactive" />
```

或

```
<dubbo:service interface="com.foo.BarService" loadbalance="leastactive" />
```

Dubbo启动的时候支持几种配置方式?

XML 配置

provider.xml 示例

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <dubbo:application name="demo-provider"/>
    <dubbo:registry address="zookeeper://127.0.0.1:2181"/>
    <dubbo:protocol name="dubbo" port="20890"/>
    <bean id="demoService" class="org.apache.dubbo.samples.basic.impl.DemoServiceImpl"/>
    <dubbo:service interface="org.apache.dubbo.samples.basic.api.DemoService" ref="demoService"/>
</beans>
```

consumer.xml示例

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <dubbo:application name="demo-consumer"/>
    <dubbo:registry group="aaa" address="zookeeper://127.0.0.1:2181"/>
    <dubbo:reference id="demoService" check="false"
interface="org.apache.dubbo.samples.basic.api.DemoService"/>
</beans>
```

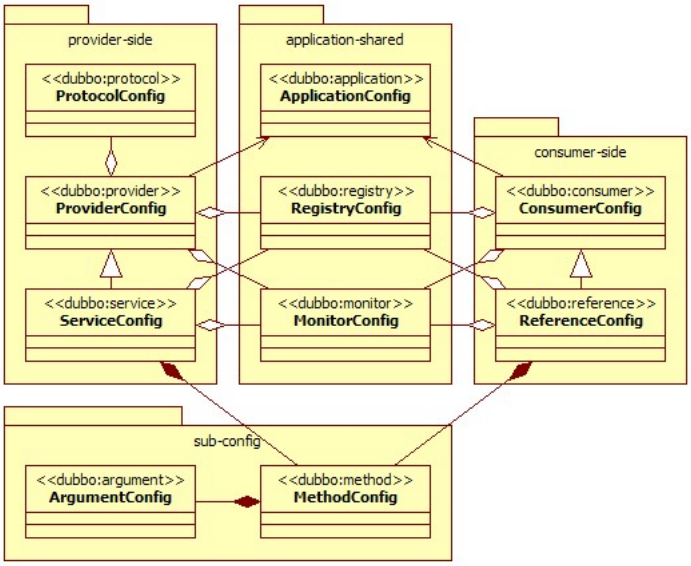
所有标签都支持自定义参数，用于不同扩展点实现的特殊配置，如：

```
<dubbo:protocol name="jms">
    <dubbo:parameter key="queue" value="your_queue" />
</dubbo:protocol>
```

或：[1]

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://dubbo.apache.org/schema/dubbo
http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <dubbo:protocol name="jms" p:queue="your_queue" />
</beans>
```

配置之间的关系



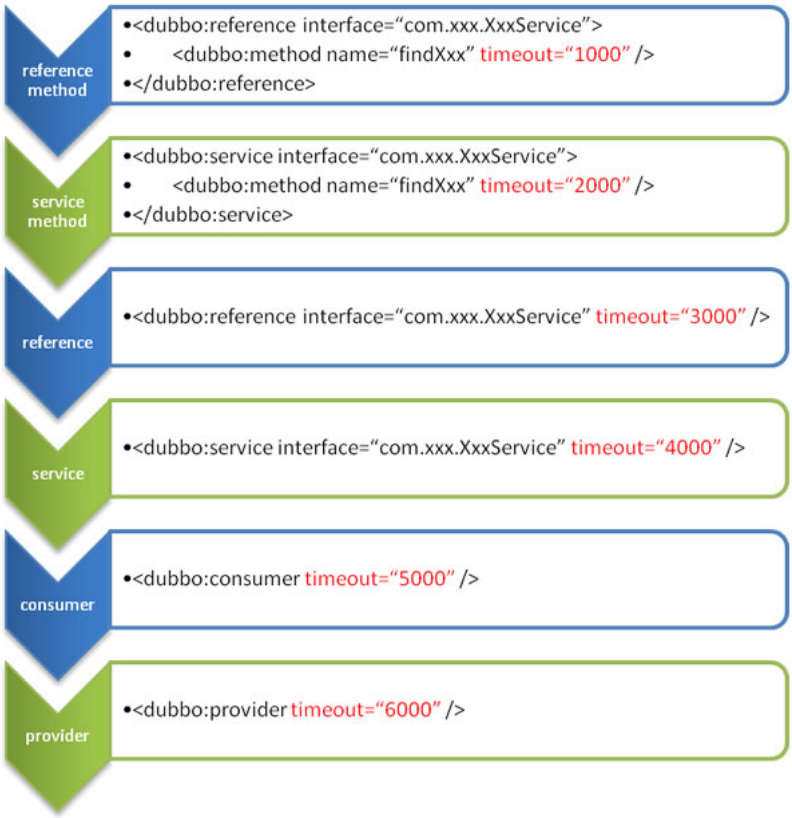
标签	用途	解释
`<dubbo:service/>`	服务配置	用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心
`<dubbo:reference/>`	引用配置	用于创建一个远程服务代理，一个引用可以指向多个注册中心
`<dubbo:protocol/>`	协议配置	用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受
`<dubbo:application/>`	应用配置	用于配置当前应用信息，不管该应用是提供者还是消费者
`<dubbo:module/>`	模块配置	用于配置当前模块信息，可选
`<dubbo:registry/>`	注册中心配置	用于配置连接注册中心相关信息
`<dubbo:monitor/>`	监控中心配置	用于配置连接监控中心相关信息，可选
`<dubbo:provider/>`	提供方配置	当 ProtocolConfig 和 ServiceConfig 某属性没有配置时，采用此缺省值，可选
`<dubbo:consumer/>`	消费方配置	当 ReferenceConfig 某属性没有配置时，采用此缺省值，可选
`<dubbo:method/>`	方法配置	用于 ServiceConfig 和 ReferenceConfig 指定方法级的配置信息
`<dubbo:argument/>`	参数配置	用于指定方法参数配置

不同粒度配置的覆盖关系

以 timeout 为例，下图显示了配置的查找顺序，其它 retries, loadbalance, actives 等类似：

- 方法级优先，接口级次之，全局配置再次之。
- 如果级别一样，则消费方优先，提供方次之。

其中，服务提供方配置，通过 URL 经由注册中心传递给消费方。



（建议由服务提供方设置超时，因为一个方法需要执行多长时间，服务提供方更清楚，如果一个消费方同时引用多个服务，就不需要关心每个服务的超时设置）。

理论上 ReferenceConfig 中除了`interface`这一项，其他所有配置项都可以缺省不配置，框架会自动使用ConsumerConfig，ServiceConfig, ProviderConfig等提供的缺省配置。

---

1. `2.1.0` 开始支持，注意声明：`xmlns:p="http://www.springframework.org/schema/p"` ↩
2. 引用缺省是延迟初始化的，只有引用被注入到其它 Bean，或被 `getBean()` 获取，才会初始化。如果需要饥饿加载，即没有人引用也立即生成动态代理，可以配置：`<dubbo:reference ... init="true" />`

属性配置

如果你的应用足够简单，例如，不需要多注册中心或多协议，并且需要在spring容器中共享配置，那么，我们可以直接使用`dubbo.properties`作为默认配置。

Dubbo可以自动加载classpath根目录下的dubbo.properties，但是你同样可以使用VM参数来指定路径：`-Ddubbo.properties.file=xxx.properties`。

映射规则

可以将xml的tag名和属性名组合起来，用'.'分隔。每行一个属性。

- `dubbo.application.name=foo` 相当于 `<dubbo:application name="foo" />`
- `dubbo.registry.address=10.20.153.10:9090` 相当于 `<dubbo:registry address="10.20.153.10:9090" />`

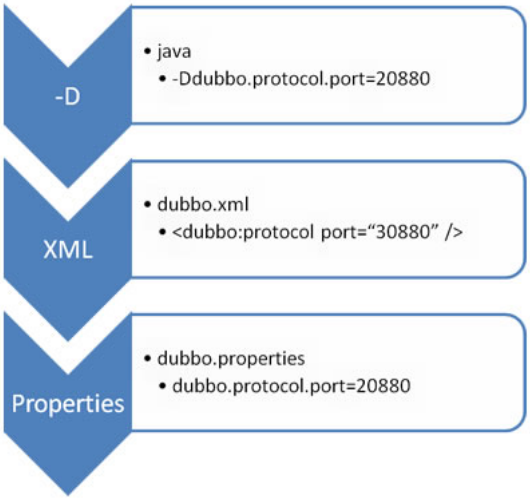
如果在xml配置中有超过一个的tag，那么你可以使用'id'进行区分。如果你不指定id，它将作用于所有tag。

- `dubbo.protocol.rmi.port=1099` 相当于 `<dubbo:protocol id="rmi" name="rmi" port="1099" />`
- `dubbo.registry.china.address=10.20.153.10:9090` 相当于 `<dubbo:registry id="china" address="10.20.153.10:9090" />`

如下，是一个典型的dubbo.properties配置样例。

```
dubbo.application.name=foo
dubbo.application.owner=bar
dubbo.registry.address=10.20.153.10:9090
```

重写与优先级



优先级从高到低：

- JVM -D参数，当你部署或者启动应用时，它可以轻易地重写配置，比如，改变dubbo协议端口；
- XML, XML中的当前配置会重写dubbo.properties中的；
- Properties，默认配置，仅仅作用于以上两者没有配置时。

1：如果在classpath下有超过一个dubbo.properties文件，比如，两个jar包都各自包含了dubbo.properties，dubbo将随机选择一个加载，并且打印错误日志。

2：如果 `id` 没有在 `protocol` 中配置，将使用 `name` 作为默认属性。

API 配置

服务提供者

```
import org.apache.dubbo.rpc.config.ApplicationConfig;
import org.apache.dubbo.rpc.config.RegistryConfig;
import org.apache.dubbo.rpc.config.ProviderConfig;
import org.apache.dubbo.rpc.config.ServiceConfig;
import com.xxx.XxxService;
import com.xxx.XxxServiceImpl;

// 服务实现
XxxService xxxService = new XxxServiceImpl();

// 当前应用配置
ApplicationConfig application = new ApplicationConfig();
application.setName("xxx");

// 连接注册中心配置
RegistryConfig registry = new RegistryConfig();
registry.setAddress("10.20.130.230:9090");
registry.setUsername("aaa");
registry.setPassword("bbb");

// 服务提供者协议配置
ProtocolConfig protocol = new ProtocolConfig();
protocol.setName("dubbo");
protocol.setPort(12345);
protocol.setThreads(200);

// 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口

// 服务提供者暴露服务配置
ServiceConfig<XxxService> service = new ServiceConfig<XxxService>(); // 此实例很重，封装了与注册中心的连接，请自行缓存，否则可能造成内存和连接泄漏
service.setApplication(application);
service.setRegistry(registry); // 多个注册中心可以用setRegistries()
service.setProtocol(protocol); // 多个协议可以用setProtocols()
service.setInterface(XxxService.class);
service.setRef(xxxService);
service.setVersion("1.0.0");

// 暴露及注册服务
service.export();
```

服务消费者

```
import org.apache.dubbo.rpc.config.ApplicationConfig;
import org.apache.dubbo.rpc.config.RegistryConfig;
import org.apache.dubbo.rpc.config.ConsumerConfig;
import org.apache.dubbo.rpc.config.ReferenceConfig;
```

```
import com.xxx.XxxService;

// 当前应用配置
ApplicationConfig application = new ApplicationConfig();
application.setName("yyy");

// 连接注册中心配置
RegistryConfig registry = new RegistryConfig();
registry.setAddress("10.20.130.230:9090");
registry.setUsername("aaa");
registry.setPassword("bbb");

// 注意：ReferenceConfig为重对象，内部封装了与注册中心的连接，以及与服务提供方的连接

// 引用远程服务
ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>(); // 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
reference.setApplication(application);
reference.setRegistry(registry); // 多个注册中心可以用setRegistries()
reference.setInterface(XxxService.class);
reference.setVersion("1.0.0");

// 和本地bean一样使用xxxService
XxxService xxxService = reference.get(); // 注意：此代理对象内部封装了所有通讯细节，对象较重，请缓存复用
```

特殊场景

下面只列出不同的地方，其它参见上面的写法

方法级设置

```
...

// 方法级配置
List<MethodConfig> methods = new ArrayList<MethodConfig>();
MethodConfig method = new MethodConfig();
method.setName("createXxx");
method.setTimeout(10000);
method.setRetries(0);
methods.add(method);

// 引用远程服务
ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>(); // 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
...
reference.setMethods(methods); // 设置方法级配置

...
```

点对点直连

```
...

ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>(); // 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
// 如果点对点直连，可以用reference.setUrl()指定目标地址，设置url后将绕过注册中心，
// 其中，协议对应provider.setProtocol()的值，端口对应provider.setPort()的值，
// 路径对应service.setPath()的值，如果未设置path，缺省path为接口名
reference.setUrl("dubbo://10.20.130.230:20880/com.xxx.XxxService");

...
```

---

1. API使用范围说明：API 仅用于 OpenAPI, ESB, Test, Mock 等系统集成，普通服务提供方或消费方，请采用XML 配置方式使用 Dubbo

注解配置

服务提供方

`Service`注解暴露服务

```
@Service
public class AnnotationServiceImpl implements AnnotationService {
    @Override
    public String sayHello(String name) {
        return "annotation: hello, " + name;
    }
}
```

增加应用共享配置

```
# dubbo-provider.properties
dubbo.application.name=annotation-provider
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.protocol.name=dubbo
dubbo.protocol.port=20880
```

指定Spring扫描路径

```
@Configuration
@EnableDubbo(scanBasePackages = "org.apache.dubbo.samples.simple.annotation.impl")
@PropertySource("classpath:/spring/dubbo-provider.properties")
static public class ProviderConfiguration {

}
```

服务消费方

`Reference`注解引用服务

```
@Component("annotationAction")
public class AnnotationAction {

    @Reference
    private AnnotationService annotationService;

    public String doSayHello(String name) {
        return annotationService.sayHello(name);
    }
}
```

增加应用共享配置

```
# dubbo-consumer.properties
dubbo.application.name=annotation-consumer
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.consumer.timeout=3000
```

指定Spring扫描路径

```
@Configuration
@EnableDubbo(scanBasePackages = "org.apache.dubbo.samples.simple.annotation.action")
@PropertySource("classpath:/spring/dubbo-consumer.properties")
@ComponentScan(value = {"org.apache.dubbo.samples.simple.annotation.action"})
static public class ConsumerConfiguration {

}
```

调用服务

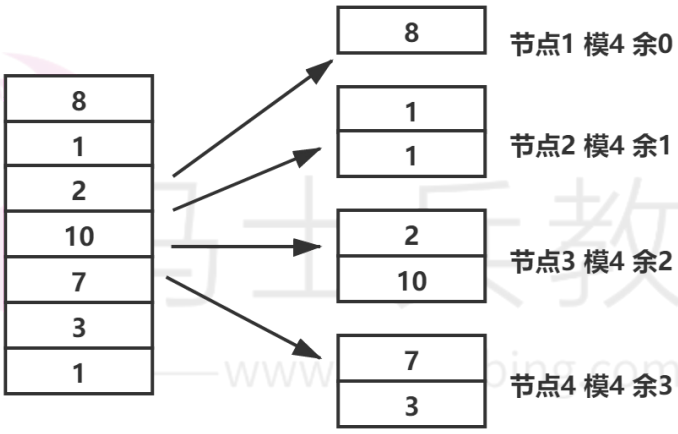
```
public static void main(String[] args) throws Exception {
    AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(ConsumerConfiguration.class);
    context.start();
    final AnnotationAction annotationAction = (AnnotationAction) context.getBean("annotationAction");
    String hello = annotationAction.doSayHello("world");
}
```

分布式系统原理

数据分布方式

所谓分布式系统顾名思义就是利用多台计算机协同解决单台计算机所不能解决的计 算、存储等问题。单机系统与分布式系统的最大的区别在于问题的规模，即计算、存
储的数据量的区别。将一个单机问题使用分布式解决，首先要解决的就是如何将问题 拆解为可以使用多机分布式解决，使得分布式系统中的每台机器负责原问题的一个子 集。由于无论是计算还是存储，其问题输入对象都是数据，所以如何拆解分布式系统 的输入数据成为分布式系统的基本问题。

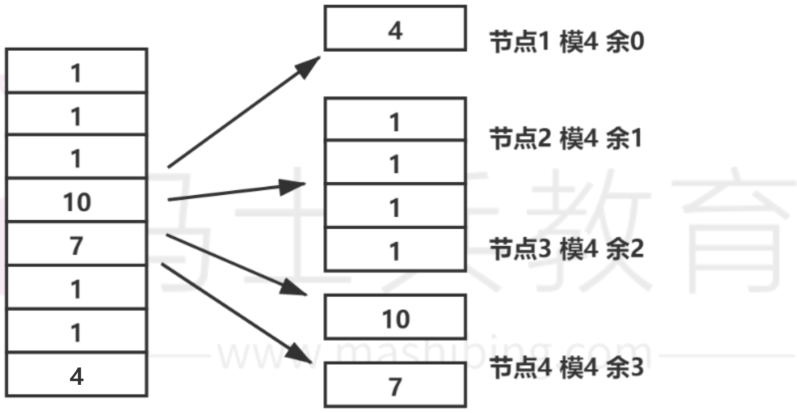
哈希方式



哈希分布数据的缺点同样明显，突出表现为可扩展性不高，一旦集群规模需要扩展， 则几乎所有的数据需要被迁移并重新分布。工程中，扩展哈希分布数据的系统时，往 往使得集群规模成倍扩展，按照数据重新计算哈希，这样原本一台机器上的数据只需 迁移一半到另一台对应的机器上即可完成扩展。

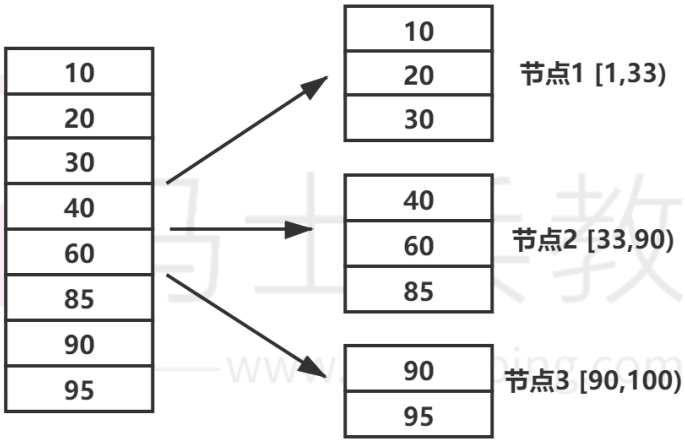
针对哈希方式扩展性差的问题，一种思路是不再简单的将哈希值与机器做除法取模映 射，而是将对应关系作为元数据由专门的元数据服务器管理.同时，哈希值取模个数往 往大于机器个数，这样同一台机器上需要负责多个哈希取模的余数。但需要以较复杂 的机制维护大量的元数据。哈希分布数据的另一个缺点是，一旦某数据特征值的数据 严重不均，容易出现“数据倾斜”（data skew）问题。

哈希分布数据的另一个缺点是，一旦某数据特征值的数据严重不均，容易出现“数据 倾斜”（data skew）问题



按数据范围分布

按数据范围分布是另一个常见的数据分布式，将数据按特征值的值域范围划分为不同 的区间，使得集群中每台（组）服务器处理不同区间的数据。



工程中，为了数据迁移等负载均衡操作的方便，往往利用动态划分区间的技术，使得 每个区间中服务的数据量尽量的一样多。当某个区间的数据量较大时，通过将区 间“分裂”的方式拆分为两个区间，使得每个数据区间中的数据量都尽量维持在一个 较为固定的阈值之下。

一般的，往往需要使用专门的服务器在内存中维护数据分布信息，称这种数据的分布 信息为一种元信息。甚至对于大规模的集群，由于元信息的规模非常庞大，单台 计算 机无法独立维护，需要使用多台机器作为元信息服务器。

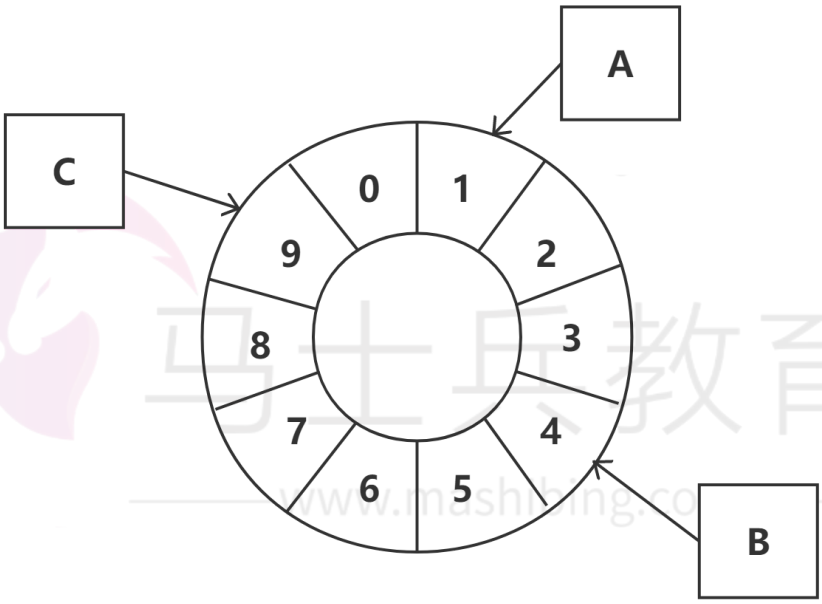
按数据量分布

数据量分布数据与具体的数据特征无关，而是将数据视为一个顺序增长的文件，并将 这个文件按照某一较为固定的大小划分为若干数据块（chunk），不同的数据块分布到不同的服务器上。与按数据范围分布数据的方式类似的是，按数据量分布数据也需 要记录数据块的具体分布情况，并将该分布信息作为元数据使用元数据服务器管理。

由于与具体的数据内容无关，按数据量分布数据的方式一般没有数据倾斜的问题，数 据总是被均匀切分并分布到集群中。当集群需要重新负载均衡时，只需通过迁移数据 块即可完成。集群扩容也没有太大的限制，只需将部分数据库迁移到新加入的机器上 即可以完成扩容。按数据量划分数据的缺点是需要管理较为复杂的元信息，与按范围 分布数据的方式类似，当集群规模较大时，元信息的数据量也变得很大，高效的管理 元信息成为新的课题。

一致性哈希

一致性哈希（consistent hashing）是另一个种在工程中使用较为广泛的数据分布方 式。一致性哈希最初在P2P 网络中作为分布式哈希表（DHT）的常用数据分布算法。 一致性哈希的基本方式是使用一个哈希函数计算数据或数据特征的哈希值，令该哈希 函数的输出值域为一个封闭的环，即哈希函数输出的最大值是最小值的前序。将节点 随机分布到这个环上，每个节点负责处理从自己开始顺时针至下一个节点的全部哈希 值域上的数据。



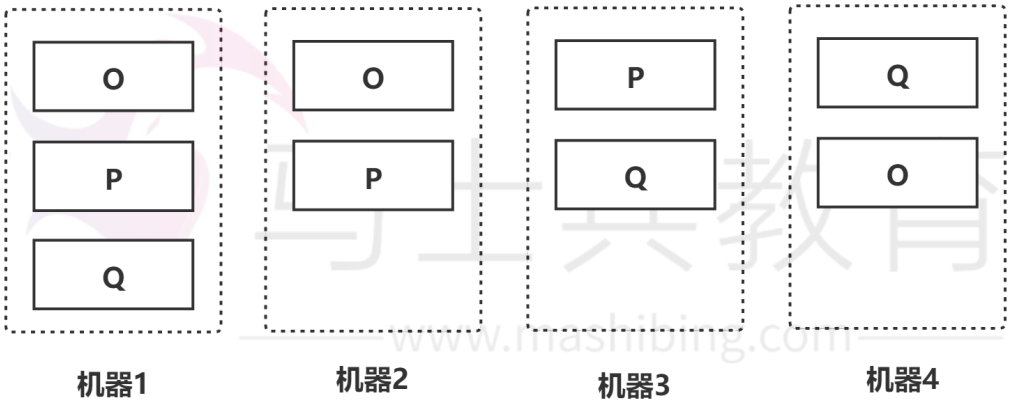
使用一致性哈希的方式需要将节点在一致性哈希环上的位置作为元信息加以管理，这 点比直接使用哈希分布数据的方式要复杂。然而，节点的位置信息只于集群中的机器 规模相关，其元信息的量通常比按数据范围分布数据和按数据量分布数据的元信息量 要小很多。

为此一种常见的改进算法是引入虚节点（virtual node）的概念，系统初始时就创建 许多虚节点，虚节点的个数一般远大于未来集群中机器的个数，将虚节点均匀分布到一致性哈希值域环上，其功能与基本一致性哈希算法中的节点相同。为每个节点分配 若干虚节点。操作数据时，首先通过数据的哈希值在环上找到对应的虚节点，进而查 找元数据找到对应的真实节点。使用虚节点改进有多个优点。首先，一旦某个节点不 可用，该节点将使得多个虚节点不可用，从而使得多个相邻的真实节点负载失效节点 的压里。同理，一旦加入一个新节点，可以分配多个虚节点，从而使得新节点可以 负 载多个原有节点的压力，从全局看，较容易实现扩容时的负载均衡。

副本与数据分布

分布式系统容错、提高可用性的基本手段就是使用副本。对于数据副本的分布方式主 要影响系统的可扩展性。一种基本的数据副本策略是以机器为单位，若干机器互为副 本，副本机器之间的数据完全相同。这种策略适用于上述各种数据分布方式。其优点 是非常简单，其缺点是恢复数据的效率不高、可扩展性也不高。

更合适的做法不是以机器作为副本单位，而是将数据拆为较合理的数据段，以数据段 为单位作为副本。实践中，常常使得每个数据段的大小尽量相等且控制在一定的大小 以内。数据段有很多不同的称谓，segment，fragment，chunk，partition 等等。 数据段的选择与数据分布方式直接相关。对于哈希分数据的方式，每个哈希分桶后的 余数可以作为一个数据段，为了控制数据段的大小，常常使得分桶个数大于集群规 模。一旦将数据分为数据段，则可以以数据段为单位管理副本，从而副本与机器不再 硬相关，每台机器都可以负责一定数据段的副本。



一旦副本分布与机器无关，数据丢失后的恢复效率将非常高。这是因为，一旦某台机 器的数据丢失，其上数据段的副本将分布在整个集群的所有机器中，而不是仅在几个 副本机器中，从而可以从整个集群同时拷贝恢复数据，而集群中每台数据源机器都可 以以非常低的资源做拷贝。作为恢复数据源的机器即使都限速1MB/s，若有100 台机 器参与恢复，恢复速度也能达到100MB/s。再者，副本分布与机器无关也利于集群容错。如果出现机器宕机，由于宕机机器上的副本分散于整个集群，其压力也自然分散 到整个集群。最后，副本分布与机器无关也利于集群扩展。理论上，设集群规模 为N 台机器，当加入一台新的机器时，只需从各台机器上迁移1/N – 1/N+1 比例的数据 段到新机器即实现了新的负载均衡。由于是从集群中各机器迁移数据，与数据恢复同 理，效率也较高。工程中，完全按照数据段建立副本会引起需要管理的元数据的开销 增大，副本维护的难度也相应增大。一种折中的做法是将某些数据段组成一个数据段 分组，按数据段分组为粒度进行副本管理。这样做可以将副本粒度控制在一个较为合 适的范围内。

本地化计算

在分布式系统中，数据的分布方式也深深影响着计算的分布方式。在分布式系统中计 算节点和保存计算数据的存储节点可以在同一台物理机器上，也可以位于不同的物理 机器。如果计算节点和存储节点位于不同的物理机器则计算的数据需要通过网络传 输，此种方式的开销很大，甚至网络带宽会成为系统的总体瓶颈。另一种思路是，将 计算尽量调度到与存储节点在同一台物理机器上的计算节点上进行，这称之为本地化 计算。本地化计算是计算调度的一种重要优化，其体现了一种重要的分布式调度思 想：“移动数据不如移动计算”。

数据分布方式的选择

在实际工程实践中，可以根据需求及实施复杂度合理选择数据分布方式。另外，数据 分布方式是可以灵活组合使用的，往往可以兼备各种方式的优点，收到较好的综合效 果。

例：数据倾斜问题，在按哈希分数据的基础上引入按数据量分布数据的方式，解决该 数据倾斜问题。按用户id 的哈希值分数据，当某个用户id 的数据量特别大时，该用 户的数据始终落在某一台机器上。此时，引入按数据量分布数据的方式，统计用户的 数据量，并按某一阈值将用户的数据切为多个均匀的数据段，将这些数据段分布到集 群中去。由于大部分用户的数据量不会超过阈值，所以元数据中仅仅保存超过阈值的 用户的数据段分布信息，从而可以控制元数据的规模。这种哈希分布数据方式与按数 据量分布数据方式组合使用的方案，在某真实系统中使用，取得了较好的效果。

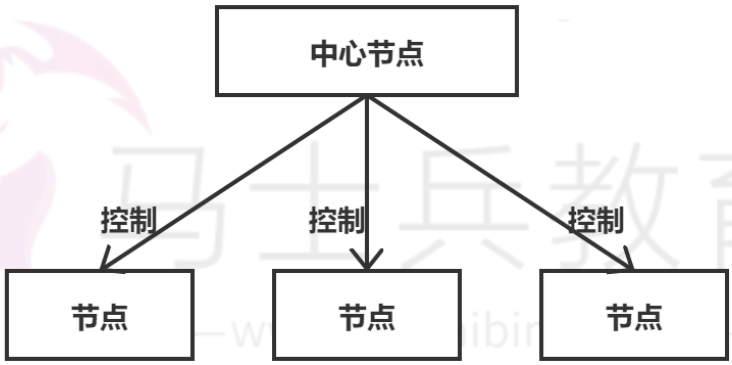
基本副本协议

副本控制协议指按特定的协议流程控制副本数据的读写行为，使得副本满足一定的可 用性和一致性要求的分布式协议。副本控制协议要具有一定的对抗异常状态的容错能 力，从而使得系统具有一定的可用性，同时副本控制协议要能提供一定一致性级别。 由CAP 原理（在2.9 节详细分析）可知，要设计一种满足强一致性，且在出现任何网 络异常时都可用的副本协议是不可能的。为此，实际中的副本控制协议总是在可用 性、一致性与性能等各要素之间按照具体需求折中。

副本控制协议可以分为两大类：“中心化(centralized)副本控制协议”和“去中心化 (decentralized)副本控制协议”。

中心化副本控制协议

中心化副本控制协议的基本思路是由一个中心节点协调副本数据的更新、维护副本之 间的一致性。图给出了中心化副本协议的通用架构。中心化副本控制协议的优点是协 议相对较为简单，所有的副本相关的控制交由中心节点完成。并发控制将由中心节点 完成，从而使得一个分布式并发控制问题，简化为一个单机并发控制问题。所谓并发 控制，即多个节点同时需要修改副本数据时，需要解决“写写”、“读写”等并发冲突。单机系统上常用加锁等方式进行并发控制。对于分布式并发控制，加锁也是一个 常用的方法，但如果没有中心节点统一进行锁管理，就需要完全分布式化的锁系统， 会使得协议非常复杂。中心化副本控制协议的缺点是系统的可用性依赖于中心化节 点，当中心节点异常或与中心节点通信中断时，系统将失去某些服务（通常至少失去 更新服务），所以中心化副本控制协议的缺点正是存在一定的停服务时间。



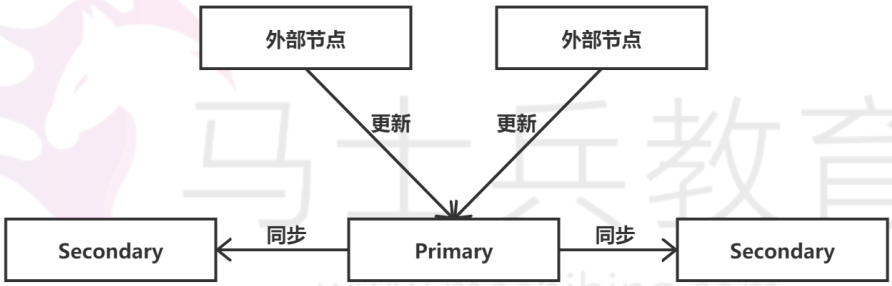
primary-secondary 协议

在primary-secondary 类型的协议中，副本被分为两大类，其中有且仅有一个副本作 为primary 副本，除primary 以外的副本都作为secondary 副本。维护primary 副本 的节点作为中心节点，中心节点负责维护数据的更新、并发控制、协调副本的一致 性。

Primary-secondary 类型的协议一般要解决四大类问题：数据更新流程、数据读取方 式、Primary 副本的确定和切换、数据同步（reconcile）。

数据更新基本流程

1. 数据更新都由primary 节点协调完成。
2. 外部节点将更新操作发给primary 节点
3. primary 节点进行并发控制即确定并发更新操作的先后顺序
4. primary 节点将更新操作发送给secondary 节点
5. primary 根据secondary 节点的完成情况决定更新是否成功并将结果返回外 部节点



在工程实践中，如果由primary 直接同时发送给其他N 个副本发送数据，则每个 secondary 的更新吞吐受限于primary 总的出口网络带宽，最大为primary 网络出口 带宽的1/N。为了解决这个问题，有些系统（例如，GFS），使用接力的方式同步数 据，即primary 将更新发送给第一 个secondary 副本，第一个secondary 副本发送 给第二secondary 副本，依次类推。

数据读取方式

数据读取方式也与一致性高度相关。如果只需要最终一致性，则读取任何副本都可以 满足需求。如果需要会话一致性，则可以为副本设置版本号，每次更新后递增版本 号，用户读取副本时验证版本号，从而保证用户读到的数据在会话范围内单调递增。 使用primary-secondary 比较困难的是实现强一致性。

1. 由于数据的更新流程都是由primary 控制的，primary 副本上的数据一定是 最新的，所以 如果始终只读primary 副本的数据，可以实现强一致性。如果只 读primary 副本，则secondary 副本将不提供读服务。实践中，如果副本不与 机器绑定，而是按照数据段为单位维护副本，仅有primary 副本提供读服务在 很多场景下并不会造出机器资源浪费。

将副本分散到集群中个，假设primary 也是随机的确定的，那么每台机器上都有一些 数据的primary 副本，也有另一些数据段的secondary 副本。从而某台服务器实际都 提供读写服务。

1. 由primary 控制节点secondary 节点的可用性。当primary 更新某个 secondary 副本不成功时，primary 将该secondary 副本标记为不可用，从而 用户不再读取该不可用的副本。不可用的 secondary 副本可以继续尝试与 primary 同步数据，当与primary 完成数据同步后，primary 可以副本标记为 可用。这种方式使得所有的可用的副本，无论是primary 还是secondary 都是 可读的，且在一个确定的时间内，某secondary 副本要么更新到与primary 一 致的最新状态，要么被标记为不可用，从而符合较高的一致性要求。这种方式依赖于一个中心元数据管理系统，用于记录哪些副本可用，哪些副本不可用。 某种意义上，该方式通过降低系统的可用性来提高系统的一致性。

primary 副本的确定与切换

在primary-secondary 类型的协议中，另一个核心的问题是如何确定primary 副本， 尤其是在原primary 副本所在机器出现宕机等异常时，需要有某种机制切换primary 副本，使得某个secondary 副本成为新的primary 副本。

通常的，在primary-secondary 类型的分布式系统中，哪个副本是primary 这一信息 都属于元信息，由专门的元数据服务器维护。执行更新操作时，首先查询元数据服务 器获取副本的primary 信息，从而进一步执行数据更新流程。

由于分布式系统中可靠的发现节点异常是需要一定的探测时间的，这样的探测时间通 常是10 秒级别，这也意味着一旦primary 异常，最多需要10 秒级别的发现时间，系统才能开始primary 的切换，在这10 秒时间内，由于没有primary，系统不能提供更 新服务，如果系统只能读primary 副本，则这段时间内甚至不能提供读服务。从这里 可以看到，primary-backup 类副本协议的最大缺点就是由于primary 切换带来的一 定的停服务时间。

数据同步

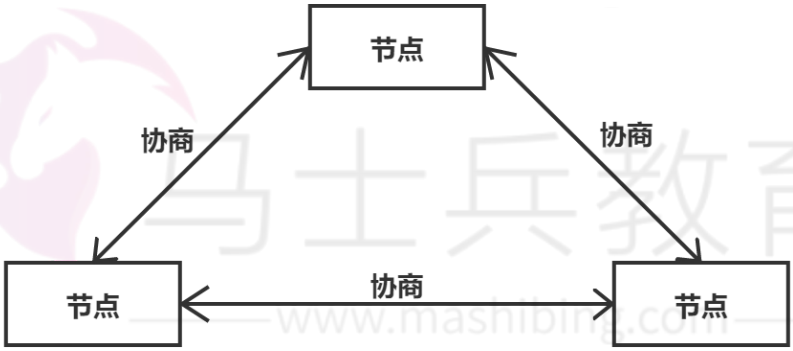
不一致的secondary 副本需要与primary 进行同步（reconcile）。

通常不一致的形式有三种：一、由于网络分化等异常，secondary 上的数据落后于 primary 上的数据。二、在某些协议下，secondary 上的数据有可能是脏数据，需要 被丢弃。所谓脏数据是由于primary 副本没有进行某一更新操作，而secondary 副本 上反而进行的多余的修改操作，从而造成secondary 副本数据错误。三、secondary 是一个新增加的副本，完全没有数据，需要从其他副本上拷贝数据。

对于第一种secondary 数据落后的情况，常见的同步方式是回放primary 上的操作日 志（通常是redo 日志），从而追上primary 的更新进度。对于脏数据的情况，较好 的做法是设计的分布式协议不产生脏数据。如果协议一定有产生脏数据的可能，则也 应该使得产生脏数据的概率降到非常低得情况，从而一旦发生脏数据的情况可以简单 的直接丢弃有脏数据的副本，这样相当于副本没有数据。另外，也可以设计一些基于 undo 日志的方式从而可以删除脏数据。如果secondary 副本完全没有数据，则常见 的做法是直接拷贝primary 副本的数据，这种方法往往比回放日志追更新进度的方法 快很多。但拷贝数据时primary 副本需要能够继续提供更新服务，这就要求primary 副本支持快照(snapshot)功能。即对某一刻的副本数据形成快照，然后拷贝快照，拷 贝完成后使用回放日志的方式追快照形成后的更新操作。

去中心化副本控制协议

去中心化副本控制协议没有中心节点，协议中所有的节点都是完全对等的，节点之间 通过平等协商达到一致。从而去中心化协议没有因为中心化节点异常而带来的停服务 等问题。 去中心化协议的最大的缺点是协议过程通常比较复杂。尤其当去中心化协议需要实现 强一致性时，协议流程变得复杂且不容易理解。由于流程的复杂，去中心化协议的效
率或者性能一般也较中心化协议低。一个不恰当的比方就是，中心化副本控制协议类 似专制制度，系统效率高但高度依赖于中心节点，一旦中心节点异常，系统受到的影 响较大；去中心化副本控制协议类似民主制度，节点集体协商，效率低下，但个别节 点的异常不会对系统总体造成太大影响。



Lease 机制

Lease 机制是最重要的分布式协议，广泛应用于各种实际的分布式系统中。

基于lease 的分布式cache 系统

基本的问题背景如下：在一个分布式系统中，有一个中心服务器节点，中心服务器存 储、维护着一些数据，这些数据是系统的元数据。系统中其他的节点通过访问中心服 务器节点读取、修改其上的元数据。由于系统中各种操作都依赖于元数据，如果每次 读取元数据的操作都访问中心服务器 节点，那么中心服务器节点的性能成为系统的瓶 颈。为此，设计一种元数据cache，在各个节点上 cache 元数据信息，从而减少对中 心服务器节点的访问，提高性能。另一方面，系统的正确运行严格依赖于元数据的正 确，这就要求各个节点上cache 的数据始终与中心服务器上的数据一致，cache 中的 数据不能是旧的脏数据。最后，设计的cache 系统要能最大可能的处理节点宕机、网 络中断等异常，最大程度的提高系统的可用性。

为此，利用lease 机制设计一套cache 系统，其基本原理为如下。中心服务器在向各 节点发送数据时同时向节点颁发一个lease。每个lease 具有一个有效期，和信用卡上 的有效期类似，lease 上的 有效期通常是一个明确的时间点，例如12:00:10，一旦真 实时间超过这个时间点，则lease 过期失效。这样lease 的有效期与节点收到lease 的 时间无关，节点可能收到lease 时该lease 就已经过期失效。这里首先假设中心服务 器与各节点的时钟是同步的，在下节中讨论时钟不同步对lease 的影响。中心服务器 发出的lease 的含义为：在lease 的有效期内，中心服务器保证不会修改对应数据的值。因此，节点收到数据和lease 后，将数据加入本地Cache，一旦对应的lease 超 时，节点将对应的本地cache 数据删除。中心服务器在修改数据时，首先阻塞所有新 的读请求，并等待之前为该数据发出的所有lease 超时过期，然后修改数据的值。
基于lease 的cache，客户端节点读取元数据

1. 判断元数据是否已经处于本地cache 且lease 处于有效期内1.1 是：直接返回 cache 中的元数据1.2 否：向中心服务器节点请求读取元数据信息1.2.1 服务器 收到读取请求后，返回元数据及一个对应的lease 1.2.2 客户端是否成功收到服 务器返回的数据 1.2.2.1 失败或超时：退出流程，读取失败，可重试1.2.2.2 成 功：将元数据与该元数据的lease 记录到内存中，返回元数据
2. 基于lease 的cache，客户端节点修改元数据流程2.1 节点向服务器发起修改 元数据请求。2.2 服务器收到修改请求后，阻塞所有新的读数据请求，即接收读 请求，但不返回数据。2.3 服务器等待所有与该元数据相关的lease 超时。2.4 服务器修改元数据并向客户端节点返回修改成功。

上述机制可以保证各个节点上的cache 与中心服务器上的中心始终一致。这是因为中 心服务器节点在发送数据的同时授予了节点对应的lease，在lease 有效期内，服务器 不会修改数据，从而客户端节点可以放心的在lease 有效期内cache 数据。上述lease 机制可以容错的关键是：服务器一旦 发出数据及lease，无论客户端是否收到，也无 论后续客户端是否宕机，也无论后续网络是否正常，服务器只要等待lease 超时，就 可以保证对应的客户端节点不会再继续cache 数据，从而可以放心的修改数据而不会 破坏cache 的一致性。

上述基础流程有一些性能和可用性上的问题，但可以很容易就优化改性。优化点一： 服务器在修改元数据时首先要阻塞所有新的读请求，造成没有读服务。这是为了防止 发出新的lease 从而引起不断有新客户端节点持有lease 并缓存着数据，形成“活 锁”。优化的方法很简单，服务器在进入修改数据流程后，一旦收到读请求则只返回 数据但不颁发lease。从而造成在修改流程执行的过程中，客户端可以读到元数据，只是不能缓存元数据。进一步的优化是，当进入修改流程，服务器颁发的lease 有效 期限选择为已发出的lease 的最大有效期限。这样做，客户端可以继续在服务器进入 修改流程后继续缓存元数据，但服务器的等待所有lease 过期的时间也不会因为颁发 新的lease 而不断延长。

最后，=cache 机制与多副本机制的区别。Cache 机制与多副本机制的相似之处都 是 将一份数据保存在多个节点上。但Cache 机制却要简单许多，对于cache 的数据，可 以随时删除丢弃，并命中cache 的后果仅仅是需要访问数据源读取数据；然而副本机 制却不一样，副本是不能随意丢弃的，每失去一个副本，服务质量都在下降，一旦副 本数下降到一定程度，则往往服务将不再可用。

lease 机制的分析

lease 的定义：Lease 是由颁发者授予的在某一有效期内的承诺。颁发者一旦发出 lease，则无论接受方是否收到，也无论后续接收方处于何种状态，只要lease 不过 期，颁发者一定严守承诺；另一方面，接收方在lease 的有效期内可以使用颁发者的 承诺，但一旦lease 过期，接收方一定不能继续使用颁发者的承诺。

Lease 机制具有很高的容错能力。首先，通过引入有效期，Lease 机制能否非常好的 容错网络异常。Lease 颁发过程只依赖于网络可以单向通信，即使接收方无法向颁发 者发送消息，也不影响lease 的颁发。由于lease 的有效期是一个确定的时间点， lease 的语义与发送lease 的具体时间无关，所以 同一个lease 可以被颁发者不断重 复向接受方发送。即使颁发者偶尔发送lease 失败，颁发者也可以 简单的通过重发的办法解决。一旦lease 被接收方成功接受，后续lease 机制不再依赖于网络通信，即 使网络完全中断lease 机制也不受影响。再者，Lease 机制能较好的容错节点宕机。 如果颁发者宕机，则宕机的颁发者通常无法改变之前的承诺，不会影响lease 的正确 性。在颁发者机恢复后，如果颁发者恢复出了之前的lease 信息，颁发者可以继续遵 守lease 的承诺。如果颁发者无法恢复lease 信息，则只需等待一个最大的lease 超时 时间就可以使得所有的lease 都失效，从而不破坏lease机制。

例如上节中的cache 系统的例子中，一旦服务器宕机，肯定不会修改元数据，重新恢 复后，只需等待一个最大的lease 超时时间，所有节点上的缓存信息都将被清空。对 于接受方宕机的情况，颁发者 不需要做更多的容错处理，只需等待lease 过期失效， 就可以收回承诺，实践中也就是收回之前赋予的权限、身份等。最后，lease 机制不 依赖于存储。颁发者可以持久化颁发过的lease 信息，从而在 宕机恢复后可以使得在 有效期的lease 继续有效。但这对于lease 机制只是一个优化，如之前的分析，即使颁发者没有持久化lease 信息，也可以通过等待一个最大的lease 时间的方式使得之 前所有颁发 的lease 失效，从而保证机制继续有效。

Lease 机制依赖于有效期，这就要求颁发者和接收者的时钟是同步的。一方面，如果 颁发者的 时钟比接收者的时钟慢，则当接收者认为 lease 已经过期的时候，颁发者依 旧认为lease 有效。接收者可以用在lease 到期前申请新的lease 的方式解决这个问 题。另一方面，如果颁发者的时钟比接收 者的时钟快，则当颁发者认为lease 已经过 期的时候，接收者依旧认为lease 有效，颁发者可能将lease 颁发给其他节点，造成 承诺失效，影响系统的正确性。对于这种时钟不同步，实践中的通常做法是将颁发者 的有效期设置得比接收者的略大，只需大过时钟误差就可以避免对lease 的有效性的 影响。

基于lease 机制确定节点状态

分布式协议依赖于对节点状态认知的全局一致性，即一旦节点Q 认为某个节点 A 异 常，则节点A 也必须认为自己异常，从而节点A 停止作为primary，避免“双主”问 题的出现。解决这种问题有两种思路，第一、设计的分布式协议可以容忍“双主”错 误，即不依赖于对节点状 态的全局一致性认识，或者全局一致性状态是全体协商后的 结果；第二、利用lease 机制。对于第一 种思路即放弃使用中心化的设计，而改用去 中心化设计，超过本节的讨论范畴。下面着重讨论利用 lease 机制确定节点状态。

由中心节点向其他节点发送lease，若某个节点持有有效的lease，则认为该节点正常 可以提供服 务。用于例2.3.1 中，节点A、B、C 依然周期性的发送heart beat 报告自 身状态，节点Q 收到heart beat 后发送一个lease，表示节点Q 确认了节点A、B、C 的状态，并允许节点在lease 有效期内正常工 作。节点Q 可以给primary 节点一个特 殊的lease，表示节点可以作为primary 工作。一旦节点Q 希望切换新的primary，则 只需等前一个primary 的lease 过期，则就可以安全的颁发新的lease 给新的 primary 节点，而不会出现“双主”问题。

在实际系统中，若用一个中心节点发送lease 也有很大的风险，一旦该中心节点宕机 或网络异常，则所有的节点没有lease，从而造成系统高度不可用。为此，实际系统 总是使用多个中心节点互为副本，成为一个小的集群，该小集群具有高可用性，对外 提供颁发lease 的功能。chubby 和zookeeper 都是基于这样的设计。

lease 的有效期时间选择

工程中，常选择的lease 时长是10 秒级别，这是一个经过验证的经验值，实践中可以 作为参考并综合选择合适的时长。

Quorum 机制

先做这样的约定：更新操作（write）是一系列顺序的过程，通过其他机制确定更新操 作的顺序（例如primary-secondary 架构中由primary 决定顺序），每个更新操作记 为wi，i 为更新操作单调递增的序号，每个wi 执行成功后副本数据都发生变化，称为 不同的数据版本，记 作vi。假设每个副本都保存了历史上所有版本的数据。

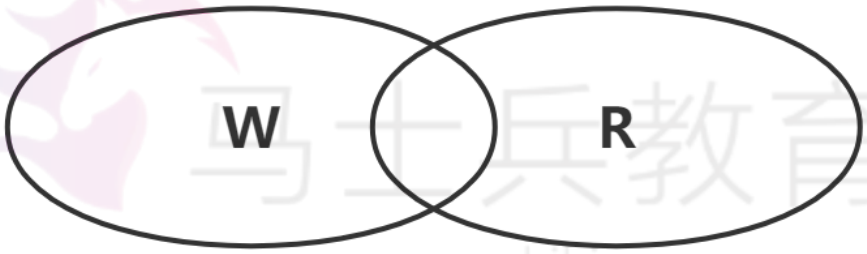
write-all-read-one

Write-all-read-one（简称WARO）是一种最简单的副本控制规则，顾名思义即在更 新时写所有的副本，只有在所有的副本上更新成功，才认为更新成功，从而保证所有 的副本一致，这样在读取数据时可以读任一副本上的数据。

由于更新操作需要在所有的N 个副本上都成功，更新操作才能成 功，所以一旦有一 个副本异常，更新操作失败，更新服务不可用。对于更新服务，虽然有N 个副本， 但系统无法容忍任何一个副本异常。另一方面，N 个副本中只要有一个副本正常，系 统就可以提供读服务。对于读服务而言，当有N 个副本时，系统可以容忍N-1 个副本 异常。从上述分析可以发现WARO 读服务的可用性较高，但更新服务的可用性不高，甚至虽然使用了副本，但更新服务的可用性等效于没有副本。

Quorum 定义

在Quorum 机制下，当某次更新操作wi 一旦在所有N 个副本中的W 个副本上都成 功，则就称 该更新操作为“成功提交的更新操作”，称对应的数据为“成功提交的数 据”。令R>N-W，由于更新 操作wi 仅在W 个副本上成功，所以在读取数据时，最 多需要读取R 个副本则一定能读到wi 更新后 的数据vi 。如果某次更新wi 在W 个副本 上成功，由于W+R>N，任意R 个副本组成的集合一定与 成功的W个副本组成的集合有交集，所以读取R 个副本一定能读到wi 更新后的数据vi。如图 2-10， Quorum 机 制的原理可以文森图表示。



某系统有5 个副本，W=3，R=3，最初5 个副本的数据一致，都是v1，某次更新操作 w2 在前3 副本上成功，副本情况变成（v2 v2 v2 v1 v1）。此时，任意3 个副本组成 的集合中一定包括 v2。在上述定义中，令W=N，R=1，就得到WARO，即WARO 是Quorum 机制的一种特例。与分析WARO 相似，分析Quorum 机制的可用性。限 制Quorum 参数为W+R=N+1。由于更新 操作需要在W 个副本上都成功，更新操作 才能成功，所以一旦N-W+1 个副本异常，更新操作始终无法在W 个副本上成功，更 新服务不可用。另一方面，一旦N-R+1 个副本异常，则无法保证一定可以读到与W 个副本有交集的副本集合，则读服务的一致性下降。

再次强调：仅仅依赖quorum 机制是无法保证强一致性的。因为仅有quorum 机制时 无法确定最新已成功提交的版本号，除非将最新已提交的版本号作为元数据由特定的 元数据服务器或元数据集群管理，否则很难确定最新成功提交的版本号。在下一节 中，将讨论在哪些情况下，可以仅仅 通过quorum 机制来确定最新成功提交的版本号。

Quorum 机制的三个系统参数N、W、R 控制了系统的可用性，也是系统对用户的服 务承诺：数据最多有N 个副本，但数据更新成功W 个副本即返回用户成功。对于一 致性要求较高的Quorum 系统，系统还应该承诺任何时候不读取未成功提交的数据， 即读取到的数据都是曾经在W 个副本上成功的数据。

读取最新成功提交的数据

Quorum 机制只需成功更新N 个副本中的W 个，在读取R 个副本时，一定可以读到 最新的成功提交的数据。但由于有不成功的更新情况存在，仅仅读取R 个副本却不一 定能确定哪个版本的数据 是最新的已提交的数据。对于一个强一致性Quorum 系 统，若存在个数据少于W 个，假设为X 个，则继续读取其他副本，直若成功读取到W 个 该版本的副本，则该数据为最新的成功提交的数据；如果在所有副本中该数据的个数肯定不满 足W 个，则R 中版本号第二大的为最新的成功提交的副本。例：在读取 到（v2 v1 v1）时，继续读取剩余的副本，若读到剩余两个副本 为（v2 v2）则v2 是 最新的已提交的副本；若读到剩余的两个副本为（v2 v1）或（v1 v1）则v1 是最新 成功提交的版本；若读取后续两个副本有任一超时或失败，则无法判断哪个版本是最 新的成功提交的版本。

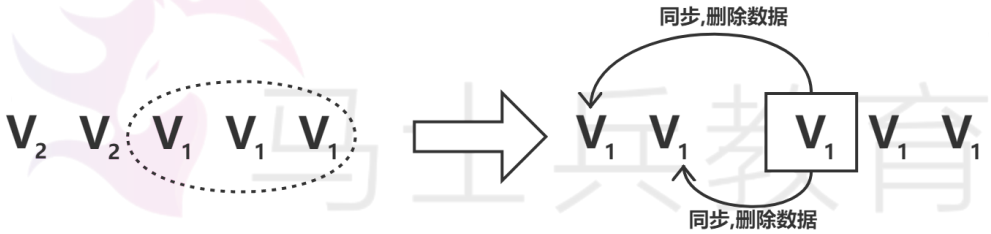
可以看出，在单纯使用Quorum 机制时，若要确定最新的成功提交的版本，最多需要 读取R+ （W-R-1）=N 个副本，当出现任一副本异常时，读最新的成功提交的版本 这一功能都有可能不可用。实际工程中，应该尽量通过其他技术手段，回避通过 Quorum 机制读取最新的成功提交的版本。例如，当quorum 机制与primarysecondary 控制协议结合使用时，可以通过读取primary 的方式读取到最新的已提交 的数据。

基于Quorum 机制选择primary副本

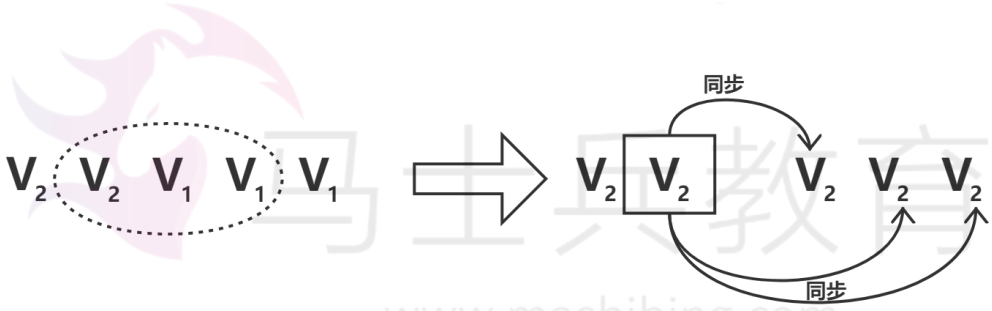
读取数据时依照一致性要求的不同可以有不同的做法：如果需要强一致性的立刻读取 到最新的成功提交的数据，则可以简单的只读取primary 副本上的数据即可，也可以 通过上节的方式读取；如果需要会话一致性，则可以根据之前已经读到的数据版本号 在各个副本上进行选择性读取；如果只需要弱一致性，则可以选择任意副本读取。

在primary-secondary 协议中，当primary 异常时，需要选择出一个新的primary， 之后secondary 副本与primary 同步数据。通常情况下，选择新的primary 的工作是 由某一中心节点完成的，在引入 quorum 机制后，常用的primary 选择方式与读取 数据的方式类似，即中心节点读取R 个副本，选择 R 个副本中版本号最高的副本作为 新的primary。新primary 与至少W 个副本完成数据同步后作为新的primary 提供读 写服务。首先，R 个副本中版本号最高的副本一定蕴含了最新的成功提交的数据。再 者，虽然不能确定最高版本号的数是一个成功提交的数据，但新的primary 在随后与 secondary 同 步数据，使得该版本的副本个数达到W，从而使得该版本的数据成为 成功提交的数据。

例：在N=5，W=3，R=3 的系统中，某时刻副本最大版本号为（v2 v2 v1 v1 v1），此时v1 是系统的最新的成功提交的数据，v2 是一个处于中间状态的未成功提交的数据。假设此刻原primary 副本异常，中心节点进行primary 切换工作。这 类“中间态”数据究竟作为“脏数据”被删除，还是作为新的数据被同步后成为生效 的数据，完全取决于这个数据能否参与新primary 的选举。下面分别分析这两种情 况。



第一、如图 2-12，若中心节点与其中3 个副本通信成功，读取到的版本号为（v1 v1 v1），则任 选一个副本作为primary，新primary 以v1 作为最新的成功提交的版本 并与其他副本同步，当与第1、第2 个副本同步数据时，由于第1、第2 个副本版本号 大于primary，属于脏数据，可以按照2.2.2.4 节中介绍的处理脏数据的方式解决。实 践中，新primary 也有可能与后两个副本完成同步后就提供数据服务，随后自身版本 号也更新到v2，如果系统不能保证之后的v2 与之前的v2 完全一样，则新 primary 在 与第1、2 个副本同步数据时不但要比较数据版本号还需要比较更新操作的具体内容 是否一样。



第二、若中心节点与其他3 个副本通信成功，读取到的版本号为（v2 v1 v1），则选 取版本号为 v2 的副本作为新的primary，之后，一旦新primary 与其他2 个副本完成 数据同步，则符合v2 的副 本个数达到W 个，成为最新的成功提交的副本，新 primary 可以提供正常的读写服务。

日志技术

日志技术是宕机恢复的主要技术之一。日志技术最初使用在数据库系统中。严格来说 日志技术不是一种分布式系统的技术，但在分布式系统的实践中，却广泛使用了日志 技术做宕机恢复，甚 至如BigTable 等系统将日志保存到一个分布式系统中进一步增 强了系统容错能力。

Redo Log 与Check point

设计一个高速的单机查询系统，将数据全部存放在内存中以实现高速的数据查询，每 次更新操作更新一小部分数据（例如 key-value 中的某一个key）。现在问题为利用 日志技术实现该内存查询系统的宕机恢复。与数据库的事务不同的是，这个问题模型 中的每个成功的更新操作都会生效。这也等效为数据库的每个事务只有一个更新操 作，且每次更新操作都可以也必须立即提交（Auto commit）。

- Redo Log

1. 将更新操作的结果（例如Set K1=1，则记录K1=1）以追加写（append）的 方式写入磁盘的 日志文件
2. 按更新操作修改内存中的数据
3. 返回更新成功

从Redo Log 的流程可以看出，Redo 写入日志的是更新操作完成后的结果（虽然本 文不讨论Undo Log，这点是与Undo Log 的区别之一），且由于是顺序追加写日志 文件，在磁盘等对顺序写有力的 存储设备上效率较高。

用Redo Log 进行宕机恢复非常简单，只需要“回放”日志即可。

流程2.5.2：Redo Log 的宕机恢复

从头读取日志文件中的每次更新操作的结果，用这些结果修改内存中的数 据。
从Redo Log 的宕机恢复流程也可以看出，只有写入日志文件的更新结果才能在宕机 后恢复。这也是为什么在Redo Log 流程中需要先更新日志文件再更新内存中的数据 的原因。假如先更新内存中的数据，那么用户立刻就能读到更新后的数据，一旦在完 成内存修改与写入日志之间发生宕机，那么最后一次更新操作无法恢复，但之前用户 可能已经读取到了更新后的数据，从而引起不一致的问题。

- Check point

。在简化的模型下，check point 技术的过程即将内存中的数据以某种易于重新加载 的数据组织方式完整的dump 到磁盘，从而减少宕机恢复时需要回放的日志数据。

流程：check point

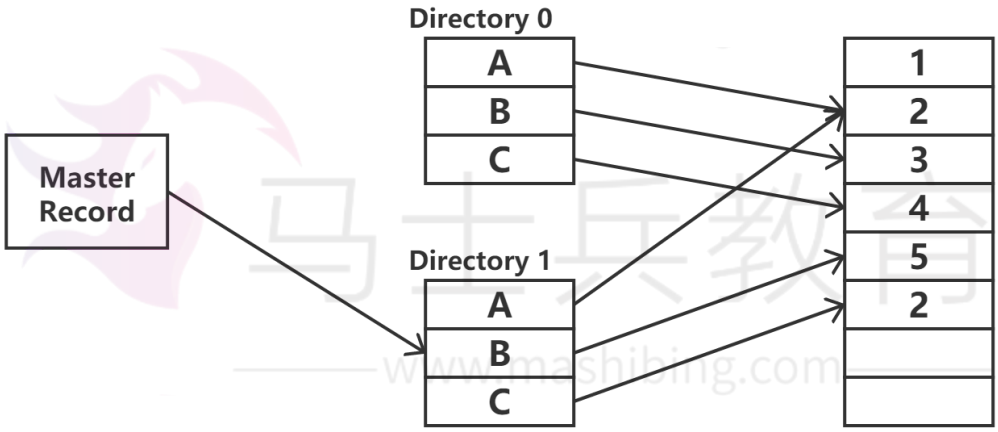
1. 向日志文件中记录“Begin Check Point”
2. 将内存中的数据以某种易于重新加载的数据组织方式dump 到磁盘上
3. 向日志文件中记录“End Check Point” 在check point 流程中，数据可以 继续按照流程2.5.1 被更新，这段过程中新更新的数据可以 dump 到磁盘也可以 不dump 到磁盘，具体取决于实现。例如，check point 开始时k1=v1，check point 过程 中某次更新为k1 = v2，那么dump 到磁盘上的k1 的值可以是v1 也 可以是v2。

流程：基于check point 的宕机恢复流程

1. 将dump 到磁盘的数据加载到内存。
2. 从后向前扫描日志文件，寻找最后一个“End Check Point”日志。
3. 从最后一个“End Check Point”日志向前找到最近的一个“Begin Check Point”日志，并回 放该日志之后的所有更新操作日志。

- No Undo/No Redo log

若数据维护在磁盘中，某批更新由若干个更新操作组成，这些更新操作需要原子生 效，即要么同时生效，要么都不生效。



0/1 目录技术中有两个目录结构，称为目录0(Directory 0)和目录1(Directory 1)。另 有一个结构称为主记录（Master record）记录当前正在使用的目录称为活动目录。 主记录中要么记录使用目录0，要么记录使用目录1。目录0 或目录1 中记录了各个数 据的在日志文件中的位置。0/1 目录的数据更新过程始终在非活动目录上进行，只是 在数据生效前，将主记录中的0、1 值反转，从而切换主记录。

流程：0/1 目录数据更新流程

1. 将活动目录完整拷贝到非活动目录。
2. 对于每个更新操作，新建一个日志项纪录操作后的值，并在非活动目录中将 相应数据的位置修改为新建的日志项的位置。
3. 原子性修改主记录：反转主记录中的值，使得非活动目录生效。

0/1 目录的更新流程非常简单，通过0、1 目录的主记录切换使得一批修改的生效是原 子的。0/1 目录将批量事务操作的原子性通过目录手段归结到主记录的原子切换。由 于多条记录的原子修改一般较难实现而单条记录的原子修改往往可以实现，从而降低 了问题实现的难度。在工程中0/1 目录的思想运用非常广泛，其形式也不局限在上述 流程中，可以是内存中的两个数据结构来回切换，也可以是磁盘上的两个文件目录来 回生效切换。

两阶段提交协议

两阶段提交协议是一种经典的强一致性中心化副本控制协议。虽然在工程中该协议有 较多的问题，但研究该协议能很好的理解分布式系统的几个典型问题。

流程描述

两阶段提交协议是一种典型的“中心化副本控制”协议。在该协议中，参与的节点分 为 两 类： 一 个 中 心 化 协 调 者 节 点 （ coordinator ） 和 N 个 参 与 者 节 点 （participant）。每个参与者节点即上文背景介绍中的管理数据库副本的节点。

两阶段提交的思路比较简单，在第一阶段，协调者询问所有的参与者是否可以提交事 务（请参与者投票），所有参与者向协调者投票。在第二阶段，协调者根据所有参与 者的投票结果做出是否事务可以全局提交的决定，并通知所有的参与者执行该决定。 在一个两阶段提交流程中，参与者不能改变自己的投票结果。两阶段提交协议的可以 全局提交的前提是所有的参与者都同意提交事务，只要有一个参与者投票选择放弃 (abort)事务，则事务必须被放弃。

流程：两阶段提交协调者流程

1. 写本地日志“begin_commit”，并进入WAIT 状态；
2. 向所有参与者发送“prepare 消息”；
3. 等待并接收参与者发送的对“prepare 消息”的响应；3.1 若收到任何一个 参与者发送的“vote-abort 消息”；3.1.1 写本地“global-abort”日志，进 入ABORT；3.1.2 向所有的参与者发送“global-abort 消息”；3.1.3 进入 ABORT 状态；3.2 若收到所有参与者发送的“vote-commit”消息；3.2.1 写 本地“global-commit”日志，进入COMMIT 状态；3.1.2 向所有的参与者发 送“global-commit 消息”； 4. 等待并接收参与者发送的对“global-abort 消息”或“global-commit 消 息”的确认响应消息，一旦收到所有参与者的确认消息，写本 地 “end_transaction” 日志流程结束。

流程：两阶段提交协调者流程

1. 写本地日志“init”记录，进入INIT 状态
2. 等待并接受协调者发送的“prepare 消息”，收到后 2.1 若参与者可以提交 本次事务 2.1.1 写本地日志“ready”，进入READY 状态 2.1.2 向协调者发 送“vote-commit”消息 2.1.4 等待协调者的消息2.1.4.1 若收到协调者 的“global-abort”消息2.1.4.1.1 写本地日志“abort”，进入ABORT 状态 2.1.4.1.2 向协调者发送对“global-abort”的确认消息 2.1.4.2 若收到协调者 的“global-commit”消息2.1.4.1.1 写本地日志“commit”，进入COMMIT 状态 2.1.4.1.2 向协调者发送对“global-commit”的确认消息 2.2 若参与 者无法提交本次事务 2.2.1 写本地日志“abort”，进入ABORT 状态 2.2.2 向协调者发送“vote-abort”消息 2.2.3 流程对该参与者结束 2.2.4 若后续收到 协调者的“global-abort”消息可以响应
3. 即使流程结束，但任何时候收到协调者发送的“global-abort”消息 或“global-commit”消息也都要发送一个对应的确认消息。

异常处理

宕机恢复

1. 协调者宕机恢复 协调者宕机恢复后，首先通过日志查找到宕机前的状态。如 果日志中最后是“begin_commit”记录，说明宕机前协调者处于WAIT 状态， 协调者可能已经发送过“prepare 消息”也可能还没发送，但协调者一定还没 有发送过“global-commit 消息”或“global-abort 消息”，即事务的全局状 态还没有确定。此时，协调者可以重新发送“prepare 消息” 继续两阶段提交 流程，即使参与者已经发送过对“prepare 消息”的响应，也不过是再次重传 之前的响应而不会影响协议的一致性。如果日志中最后是“globalcommit”或“global-abort”记录，说明宕机前协调者处于COMMIT 或 ABORT 状态。此时协调者只需重新向所有的参与者发送“global-commit 消 息”或“global-abort 消息”就可以继续两阶段提交流程。
2. 参与者宕机恢复参与者宕机恢复后，首先通过日志查找宕机前的状态。如果 日志中最后是“init”记录，说明参与者处于INIT 状态，还没有对本次事务做 出投票选择，参与者可以继续流程等待协调者发送的“prepare 消息”。如果 日志中最后是“ready”记录，说明参与者处于REDAY 状态，此时说明参与者 已经就本次 事务做出了投票选择，但宕机前参与者是否已经向协调者发 送“vote-commit”消息并不可知。所以此时参与者可以向协调者重发“votecommit”，并继续协议流程。如果日志中最后是“commit”或“abort”记 录，说明参与者已经收到过协调者的“global-commit 消息”（处于COMMIT 状态）或者“global-abort 消息”（处于ABORT 状态）。至于是否向协调者 发 送过对“global-commit”或“global-abort”的确认消息则未知。但即使 没有发送过确认消息，由于协调者会不断重发“global-commit”或“globalabort”，只需在收到这些消息时发送确认消息既可，不影响协议的全局一致 性。

协议分析

两阶段提交协议在工程实践中真正使用的较少，主要原因有以下几点：

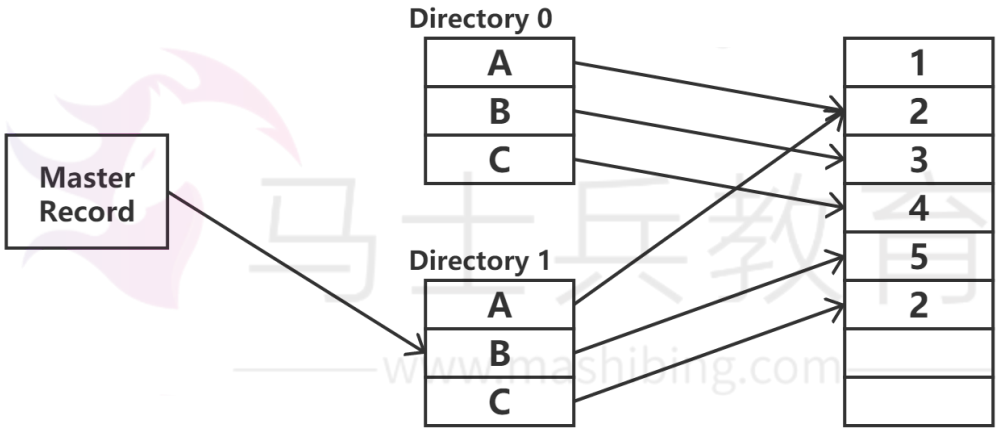
1. 两阶段提交协议的容错能力较差。从上文的分析可以看出，两阶段提交协议 在某些情况下存在流程无法执行下去的情况，且也无法判断流程状态。在工程 中好的分布式协议往往总是可以在即使发生异常的情况下也能执行下去。例 如，回忆Lease 机制（2.3 ），一旦 lease 发出，无论出现任何异常，Lease 服 务器节点总是可以通过时间判定出Lease 是否有效，也可以用等待Lease 超时 的方法收回 Lease 权限，整个Lease 协议的流程不存在任何流程被阻塞而无法 执行下去的情况。与Lease 机制的简单有效相比，两阶段提交的协议显得较为 复杂且容错能力差。
2. 两阶段提交协议的性能较差。一次成功的两阶段提交协议流程中，协调者与 每个参与者 之间至少需要两轮交互4 个消息“prepare”、“votecommit”、“global-commit”、“确认global-commit”。过多的交互次 数会降低性能。另一方面，协调者需要等待所有的参与者的投票结果，一旦存 在较慢的参与者，会影响全局流程执行速度。

虽然存在一些改进的两阶段提交协议可以提高容错能力和性能，然而这类协议依旧是 在工程中使用较少的一类协议，其理论价值大于实践意义。

MVCC

MVCC(Multi-version Cocurrent Control，多版本并发控制)技术。MVCC 技术最初 也是在数据库系统中被提出，但这种思想并不局限于单机的分布式系统，在分布式系 统中同样有效。

MVCC 即多个不同版本的数据实现并发控制的技术，其基本思想是为每次事务生成 一个新版本的数据，在读数据时选择不同版本的数据即可以实现对事务结果的完整性 读取。在使用MVCC 时，每个事务都是基于一个已生效的基础版本进行更新，事务可 以并行进行，从而可以产生一种图状结构。



基础数据的版本为1，同时产生了两个事务：事务A 与事务B。这两个事务都各自对数 据进行了一些本地修改（这些修改只有事务自己可见，不影响真正的数据），之后事 务A 首先提交，生成数据版本2；基于数据版本2，又发起了事务C，事务C 继续提 交，生成了数据版 本3；最后事务B 提交，此时事务B 的结果需要与事务C 的结果合 并，如果数据没有冲突，即事务 B 没有修改事务A 与事务C 修改过的变量，那么事务 B 可以提交，否则事务B 提交失败。MVCC 的流程过程非常类似于SVN 等版本控制 系统的流程，或者说SVN 等版本控制系统就是 使用的MVCC 思想。事务在基于基础 数据版本做本地修改时，为了不影响真正的数据，通常有两种做法，一是将基础数据 版本中的数据完全拷贝出来再修改，SVN 即使用了这种方法，SVN check out 即是 拷贝的过程；二是每个事务中只记录更新操作，而不记录完整的数据，读取数据时再 将更新操作应用到用基础版本的数据从而计算出结果，这个过程也类似SVN 的增量提 交。

Paxos协议

Paxos 协议是少数在工程实践中证实的强一致性、高可用的去中心化分布式协议。 Paxos 协议的流程较为复杂，但其基本思想却不难理解，类似于人类社会的投票过程。Paxos 协议中，有一组完全对等的参与节点（称为accpetor），这组节点各自就某一事件做出决议，如果某个决议获得了超过半数节点的同意则生效。Paxos 协议中只要有超过一半的节点正常，就可以工作，能很好对抗宕机、网络分化等异常情况。

角色

Proposer：提案者。Proposer 可以有多个，Proposer 提出议案（value）。所谓 value，在工程中可以是任何操作，例如“修改某个变量的值为某个值”、“设置当前primary 为某个节点”等等。Paxos 协议中统一将这些操作抽象为value。不同的 Proposer 可以提出不同的甚至矛盾的value，例如某个Proposer 提议“将变量X 设置为1”，另一个Proposer 提议“将变量X 设置为2”，但对同一轮Paxos 过程，最多只有一个 value 被批准。Acceptor：批准者。Acceptor 有N 个，Proposer 提出的value 必须获得超过半数(N/2+1)的Acceptor 批准后才能通过。Acceptor 之间完全对等独立。Learner：学习者。Learner 学习被批准的value。所谓学习就是通过读取各个Proposer 对value 的选择结果，如果某个value 被超过半数Proposer 通过，则Learner 学习到了这个value。回忆（2.4 ） 不难理解，这里类似Quorum 机制，某个 value 需要获得W=N/2 + 1 的Acceptor 批准，从而学习者需要至少读取N/2+1 个Accpetor，至多读取N 个Acceptor 的结果后，能学习到一个通过的value。上述三类角色只是逻辑上的划分，实践中一个节点可以同时充当这三类角色。

流程

Paxos 协议一轮一轮的进行，每轮都有一个编号。每轮Paxos 协议可能会批准一个 value，也可 能无法批准一个value。如果某一轮Paxos 协议批准了某个value，则以后各轮Paxos 只能批准这个 value。上述各轮协议流程组成了一个Paxos 协议实例，即一次Paxos 协议实例只能批准一个value，这也是Paxos 协议强一致性的重要体现。每轮Paxos 协议分为阶段，准备阶段和批准阶段，在这两个阶段Proposer 和 Acceptor 有各自的处理流程。

流程：Proposer 的流程 （准备阶段）

1. 向所有的Acceptor 发送消息“Prepare(b)”；这里b 是Paxos 的轮数，每

轮递增

2. 如果收到任何一个Acceptor 发送的消息“Reject(B)”，则对于这个 Proposer 而言本轮Paxos 失败，将轮数b 设置为B+1 后重新步骤1；（批准阶段，根据收到的Acceptor 的消息作出不同选择）
3. 如果接收到的Acceptor 的“Promise(b, v_i)”消息达到N/2+1 个（N 为 Acceptor 总数，除法取整， 下同）；v_i 表示Acceptor 最近一次在i 轮批准过 value v。3.1 如果收到的“Promise(b, v)”消息中，v 都为空，Proposer 选择一个value v，向所有Acceptor 广播Accept(b, v)；3.2 否则，在所有收到的“Promise(b, v_i)”消息中，选择i 最大的value v，向所有Acceptor 广播消息Accept(b，v)；
4. 如果收到Nack(B)，将轮数b 设置为B+1 后重新步骤1；流程：Accpetor 流程 （准备阶段）
5. 接受某个Propeser 的消息Prepare(b)。参数B 是该Acceptor 收到的最大

Paxos 轮数编号；V 是Acceptor 批准的value，可以为空 1.1 如果b>B，回复 Promise(b, V_B)，设置B=b; 表示保证不再接受编号小于b 的提案。1.2 否则，回复Reject(B) （批准阶段）

2. 接收Accept(b, v)， 2.1 如果b < B, 回复Nack(B)，暗示proposer 有一个更大编号的提案被这个Acceptor 接收了 2.2 否则设置V=v。表示这个Acceptor 批准的Value 是v。广播Accepted 消息。

例子

基本例子里有5 个Acceptor，1 个Proposer，不存在任何网络、宕机异常。我们着重考察各个Accpetor 上变量B 和变量V 的变化，及Proposer 上变量b 的变化。

1. 初始状态

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	0	0	0	0	0
V	NULL	NULL	NULL	NULL	NULL

	Proposer 1
b	1

2. Proposer 向所有Accpetor 发送“Prepare(1)”，所有Acceptor 正确处理，并回复Promise(1, NULL

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	NULL	NULL	NULL	NULL	NULL

	Proposer 1
b	1

3. Proposer 收到5 个Promise(1, NULL)，满足多余半数的Promise 的value 为空，此时发送 Accept(1, v1)，其中v1 是Proposer 选择的Value。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	v_1	v_1	v_1	v_1	v_1

4. 此时，v1 被超过半数的Acceptor 批准，v1 即是本次Paxos 协议实例批准的

Value。如果Learner 学习value，学到的只能是v1

在同一个Paxos 实例中，批准的Value 是无法改变的，即使后续Proposer 以更高的序号发起Paxos 协议也无法改变value。Paxos 协议的核心就在于“批准的value 无法改变”，这也是整个协议正确性的基础。

Paxos 协议是被人为设计出来，其设计过程也是协议的推导过程。Paxos 协议利用了 Quorom 机 制，选择的W=R=N/2+1。简单而言，协议就是Proposer 更新 Acceptor 的过程，一旦某个Acceptor 成功更新了超过半数的Acceptor，则更新成功。Learner 按Quorum 去读取Acceptor，一旦某个value 在超过半数的Proposer 上被成功读取，则说明这是一个被批准的value。协议通过引入轮次，使得高轮次的提议抢占低轮次的提议来避免死锁。协议设计关键点是如何满足“在一次Paxos 算法实例过程中只批准一个Value”这一约束条件。

CAP

CAP 理论的定义很简单，CAP 三个字母分别代表了分布式系统中三个相互矛盾的属性：

- Consistency (一致性)：CAP 理论中的副本一致性特指强一致性（1.3.4 ）；
- Availiablity(可用性)：指系统在出现异常时已经可以提供服务；
- Tolerance to the partition of network (分区容忍)：指系统可以对网络分区（1.1.4.2 ）这种异常情 况进行容错处理；

CAP 理论指出：无法设计一种分布式协议，使得同时完全具备CAP 三个属性，即1) 该种协议下的副本始终是强一致性，2)服务始终是可用的，3)协议可以容忍任何网络分区异常；分布式系统协议只能在CAP 这三者间所有折中。热力学第二定律说明了永动机是不可能存在的，不要去妄图设计永动机。与之类似， CAP 理论的意义就在于明确提出了不要去妄图设计一种对CAP 三大属性都完全拥有的完美系统，因为这种系统在理论上就已经被证明不存在。

- Lease 机制: Lease 机制牺牲了部分异常情况下的A，从而获得了完全的C 与很好的P。
- Quorum 机制: Quorum 机制，在CAP 三大因素中都各做了折中，有一定的C，有较好 的A，也有较好的P，是一种较为平衡的分布式协议。
- 两阶段提交协议: 两阶段提交系统具有完全的C，很糟糕的A，很糟糕的P。
- Paxos 协议：同样是强一致性协议，Paxos 在CAP 三方面较之两阶段提交协议要优秀得多。Paxos 协议具有 完全的C，较好的A，较好的P。Paxos 的A 与P 的属性与Quorum 机制类似，因为Paxos 的协议本 身就具有Quorum 机制的因素。

Dubbo 支持哪些协议，每种协议的应用场景，优缺点?

• **dubbo**：单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化；

• **rmi**：采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现

Serializable 接口，使用 java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。

多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，java 序列化存在安全漏洞；

• **webservice**：基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；

• **http**：基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；

• **hessian**：集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务， Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；

- **memcache：** 基于 memcached 实现的 RPC 协议
- **redis：** 基于 redis 实现的 RPC 协议

Dubbo 超时时间怎样设置?

Dubbo 超时时间设置有两种方式：

- 服务提供者端设置超时时间，在 Dubbo 的用户文档中，推荐如果能在服务端多配置就尽量多配置，因为服务提供者比消费者更清楚自己提供的服务特性。
- 服务消费者端设置超时时间，如果在消费者端设置了超时时间，以消费者端为主，即优先级更高。因为服务调用方设置超时时间控制性更灵活。如果消费方超时，服务端线程不会定制，会产生警告。

Dubbo 有些哪些注册中心?

- **Multicast 注册中心：** Multicast 注册中心不需要任何中心节点，只要广播地址，就能进行服务注册和发现。基于网络中组播传输实现；
- **Zookeeper 注册中心：** 基于分布式协调系统 Zookeeper 实现，采用

Zookeeper 的 watch 机制实现数据变更；

- **redis 注册中心：** 基于 redis 实现，采用 key/Map 存储，住 key 存储服务名和类型，Map 中 key 存储服务 URL，value 服务过期时间。基于 redis 的发布/订阅模式通知数据变更；
- **Simple 注册中心**

Dubbo 集群的负载均衡有哪些策略

Dubbo 提供了常见的集群策略实现，并预扩展点予以自行实现。

- **Random LoadBalance:** 随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀；
- **RoundRobin LoadBalance:** 轮循选取提供者策略，平均分布，但是存在请求累积的问题；
- **LeastActive LoadBalance:** 最少活跃调用策略，解决慢提供者接收更少的请求；
- **ConstantHash LoadBalance:** 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动；

□ Dubbo 是什么?

□ Dubbo 是一个分布式、高性能、透明化的 RPC 服务框架，提

供服务自动注册、自动发现等高效服务治理方案， 可以和

Spring 框架无缝集成。

□ Dubbo 的主要应用场景?

- 透明化的远程方法调用，就像调用本地方法一样调用远程方法，

只需简单配置，没有任何 API 侵入。

- 软负载均衡及容错机制，可在内网替代 F5 等硬件负载均衡器，

降低成本，减少单点。

- 服务自动注册与发现，不再需要写死服务提供方地址，注册中心

基于接口名查询服务提供者的 IP 地址，并且能够平滑添加或删

除服务提供者。

Dubbo 的核心功能?

主要就是如下 3 个核心功能：

§ **Remoting：**网络通信框架，提供对多种 NIO 框架抽象封装，包括

“同步转异步”和“请求-响应”模式的信息交换方式。

§ **Cluster：服务框架**，提供基于接口方法的透明远程过程调用，包括多

协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群

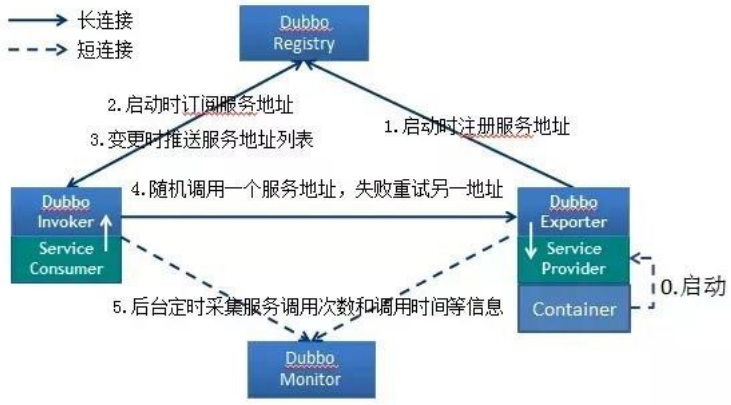
支持。

Registry：服务注册，基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

组件角色	说明
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Registry	服务注册与发现的注册中心
Monitor	统计服务的调用次调和调用时间的监控中心
Container	服务运行容器

Dubbo 的核心组件?

Dubbo 服务注册与发现的流程?



流程说明:

§ Provider(提供者)绑定指定端口并启动服务

§ 指供者连接注册中心，并发本机 IP、端口、应用信息和提供服务信息

发送至注册中心存储

§ Consumer(消费者），连接注册中心，并发送应用信息、所求服务信

息至注册中心

§ 注册中心根据 消费 者所求服务信息匹配对应的提供者列表发送至

Consumer 应用缓存。

§ Consumer 在发起远程调用时基于缓存的消费者列表择其一发起调

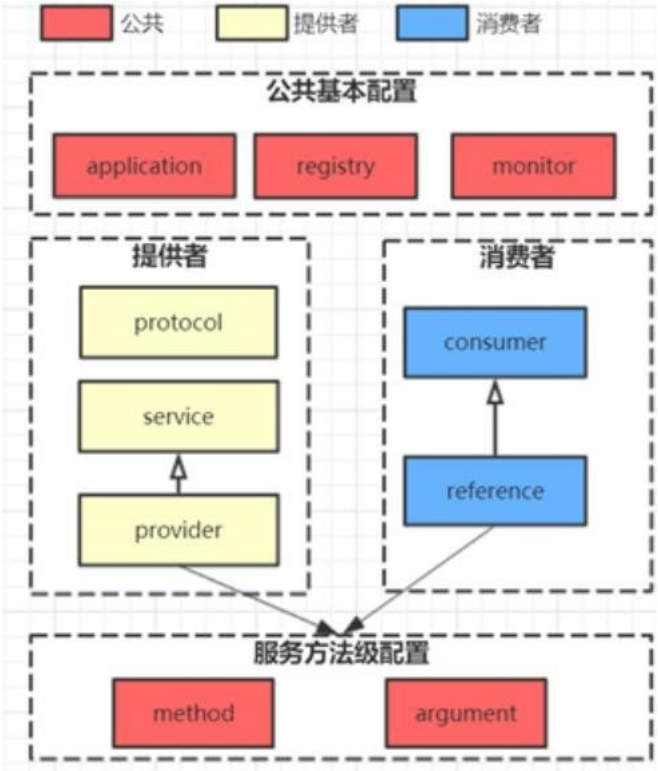
用。

Provider 状态变更会实时通知注册中心、在由注册中心实时推送至

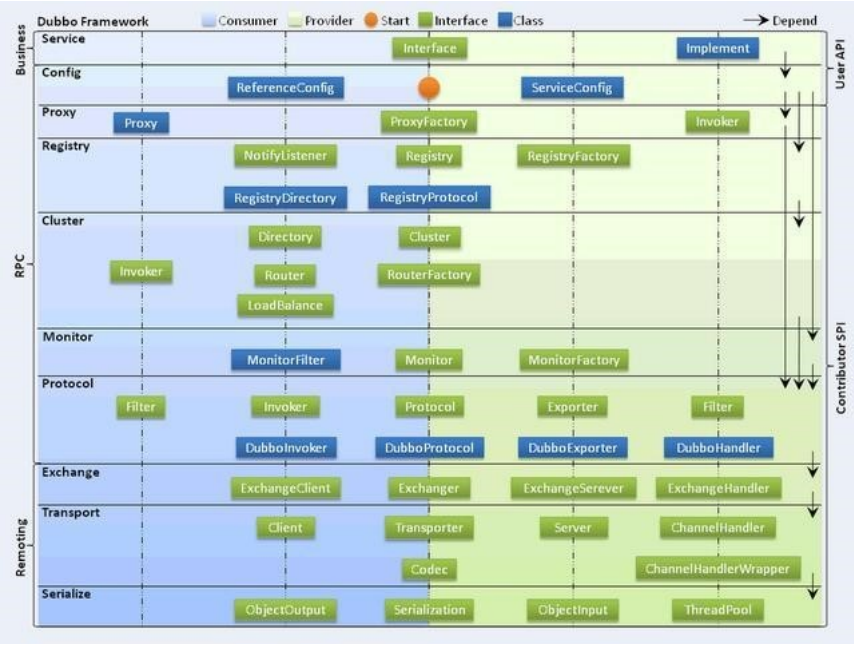
Consumer

设计的原因:

- Consumer 与 Provider 解偶，双方都可以横向增减节点数。
- 注册中心对本身可做对等集群，可动态增减节点，并且任意一台宕掉后，将自动切换到另一台
- 去中心化，双方不直接依懒注册中心，即使注册中心全部宕机短时间内也不会影响服务的调用
- 服务提供者无状态，任意一台宕掉后，不影响使用



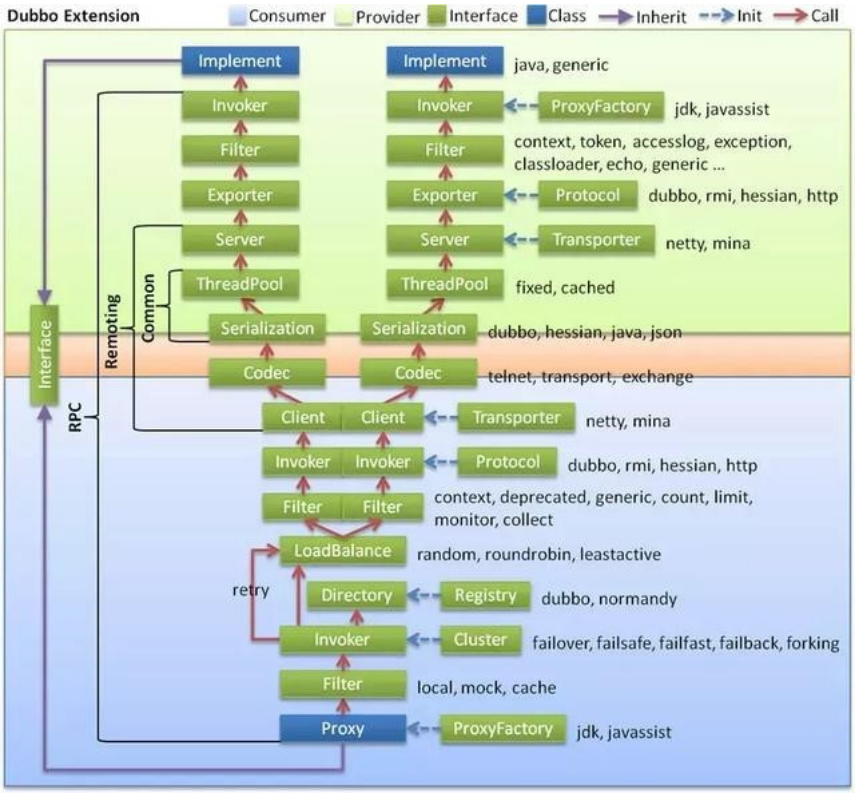
Dubbo 的架构设计?



Dubbo 框架设计一共划分了 10 个层:

- **服务接口层（Service）**：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。
- **配置层（Config）**：对外配置接口，以 ServiceConfig 和ReferenceConfig 为中心。
- **服务代理层（Proxy）**：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton。
- **服务注册层（Registry）**：封装服务地址的注册与发现，以服务 URL为中心。
- **集群层（Cluster）**：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心。
- **监控层（Monitor）**：RPC 调用次数和调用时间监控。
- **远程调用层（Protocol）**：封将 RPC 调用，以 Invocation 和 Result为中心，扩展接口为 Protocol、Invoker 和 Exporter。
- **信息交换层（Exchange）**：封装请求响应模式，同步转异步，以Request 和 Response 为中心。
- **网络传输层（Transport）**：抽象 mina 和 netty 为统一接口，以essage 为中心。

Dubbo 的服务调用流程?



Dubbo 支持哪些协议，每种协议的应用场景，优缺点?

- **dubbo：** 单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化； **rmi：** 采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。 多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections包，java 序列化存在安全漏洞；
- **webservice：** 基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；
- **http：** 基于 Http 表单提交的远程调用协议，使用 Spring 的HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；
- **hessian：** 集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；
- **memcache：** 基于 memcached 实现的 RPC 协议
- **redis：** 基于 redis 实现的 RPC 协议 **dubbo 推荐用什么协议?**
  默认使用 dubbo 协议

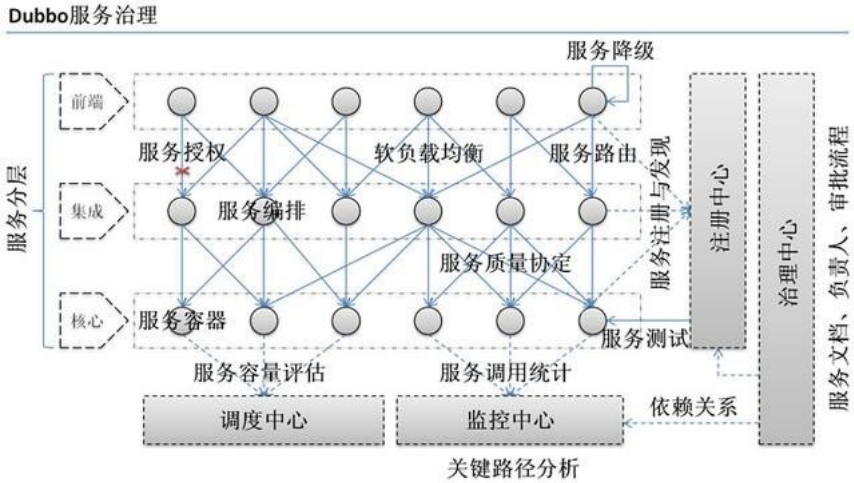
Dubbo 有些哪些注册中心?

- **Multicast 注册中心：** Multicast 注册中心不需要任何中心节点，只要广播地址，就能进行服务注册和发现。基于网络中组播传输实现；
- **Zookeeper 注册中心：** 基于分布式协调系统 Zookeeper 实现，采用Zookeeper 的 watch 机制实现数据变更；
- **redis 注册中心：** 基于 redis 实现，采用 key/Map 存储，住 key 存储服务名和类型，Map 中 key 存储服务 URL，value 服务过期时间。基于 redis 的发布/订阅模式通知数据变更；
- **Simple 注册中心**

Dubbo 默认采用注册中心?

采用 Zookeeper

为什么需要服务治理?



- 过多的服务 URL 配置困难
- 负载均衡分配节点压力过大的情况下也需要部署集群服务依赖混乱，启动顺序不清晰
- 过多服务导致性能指标分析难度较大，需要监控

Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么?

可以的，启动 dubbo 时，消费者会从 zookeeper 拉取注册的生产者的地址接口等数据，缓存在本地。

每次调用时，按照本地存储的地址进行调用。

Dubbo 与 Spring 的关系?

Dubbo 采用全 Spring 配置方式，透明化接入应用，对应用没有任何

API 侵入，只需用 Spring 加载 Dubbo 的配置即可，Dubbo 基于

Spring 的 Schema 扩展进行加载。

Dubbo 使用的是什么通信框架?

默认使用 NIO Netty 框架

Dubbo 集群提供了哪些负载均衡策略?

- **Random LoadBalance:** 随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀；
- **RoundRobin LoadBalance:** 轮循选取提供者策略，平均分布，但是存在请求累积的问题；

LeastActive LoadBalance: 最少活跃调用策略，解决慢提供者接收

更少的请求；

ConstantHash LoadBalance: 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动；

缺省时为 Random 随机调用

Dubbo 的集群容错方案有哪些?

Failover Cluster

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但

重试会带来更长延迟。

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

Forking Cluster 并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

roadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错 。通常用于通知所有提供者更新缓存或日志等本地资源信息。

Dubbo 的默认集群容错方案?

Failover Cluster

Dubbo 支持哪些序列化方式?

默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。

Dubbo 超时时间怎样设置?

Dubbo 超时时间设置有两种方式：

- 服务提供者端设置超时时间，在 Dubbo 的用户文档中，推荐如果能在服务端多配置就尽量多配置，因为服务提供者比消费者更清楚自己提供的服务特性。
- 服务消费者端设置超时时间，如果在消费者端设置了超时时间，以消费者端为主，即优先级更高。因为服务调用方设置超时时间控制性更灵活。如果消费方超时，服务端线程不会定制，会产生警告。

服务调用超时问题怎么解决?

dubbo 在调用服务不成功时，默认是会重试两次的。

Dubbo 在安全机制方面是如何解决?

Dubbo 通过 Token 令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo 还提供服务黑白名单，来控制服务所允许的调用方。

Dubbo 和 Dubbox 之间的区别?

dubbox 基于 dubbo 上做了一些扩展，如加了服务可 restful 调用，更新了开源组件等。

Dubbo 和 Spring Cloud 的关系?

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。而 Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依托了 Spirng、

Spirng Boot 的优势之上，两个框架在开始目标就不一致，Dubbo 定位服务治理、Spirng Cloud 是一个生态。

Dubbo 和 Spring Cloud 的区别?

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

最大的区别：Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于TCP 协议传输的，配合以 Hession 序列化完成 RPC 通信。

而 SpringCloud 是基于 Http 协议+Rest 接口调用远程过程的通信，相对来说，Http 请求会有更大的报文，占的带宽也会更多。但是REST 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖。

Dubbo 中 zookeeper 做注册中心，如果注册中心集群都挂掉，发布者和订阅者之间还能通信么?

可以通信的，启动 dubbo 时，消费者会从 zk 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用；

注册中心对等集群，任意一台宕机后，将会切换到另一台；注册中心全部宕机后，服务的提供者和消费者仍能通过本地缓存通讯。服务提供者无状态，任一台 宕机后，不影响使用；服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复；

挂掉是不要紧的，但前提是你没有增加新的服务，如果你要调用新的服务，则是不能办到的。

附文档截图：

dubbo 服务负载均衡策略?

l Random LoadBalance

随机，按权重设置随机概率。在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比

较均匀，有利于动态调整提供者权重。(权重可以在 dubbo 管控台配置)

1	**<dubbo:parameter key="hash.arguments" value="0,1" />**

1	**<dubbo:parameter key="hash.nodes" value="320" />**

l RoundRobin LoadBalance

轮循，按公约后的权重设置轮循比率。存在慢的提供者累积请求问题，比如：第二台机器很慢，但没挂，当请求调

到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

l LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。使慢的提供者收到更少请求，因为越慢的提供者的

调用前后计数差会越大。

l ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。缺省只对第一个参数 Hash，如果要修改，请配置

```
<dubbo:parameter key="hash.arguments" value="0,1" />
```

缺省用 160 份虚拟节点，如果要修改，请配置

```
<dubbo:parameter key="hash.nodes" value="320" />
```

Dubbo 在安全机制方面是如何解决的

Dubbo 通过 Token 令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo 还提供服务黑白名单，来控

制服务所允许的调用方。

dubbo 连接注册中心和直连的区别

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，

点对点直联方式，将以服务接口为单位，忽略注册中心的提供者列表，

服务注册中心，动态的注册和发现服务，使服务的位置透明，并通过在消费方获取服务提供方地址列表，实现软负载均衡和 Failover， 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外，注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者

注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

1. **dubbo 服务集群配置（集群容错模式）**

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。可以自行扩展集群容错策略

l Failover Cluster(默认)

失败自动切换，当出现失败，重试其它服务器。(缺省)通常用于读操作，但重试会带来更长延迟。可通过retries="2"来设置重试次数(不含第一次)。

```
<dubbo:service retries="2" cluster="failover"/>      或:          <dubbo:reference retries="2"
cluster="failover"/>         cluster="failover"可以不用写,因为默认就是 failover
```

l Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

```
dubbo:service cluster="failfast" />      或:
<dubbo:reference cluster="failfast" />
cluster="failfast"和 把 cluster="failover"、retries="0"是一样的效果,retries="0"就是不重试
```

l Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

```
<dubbo:service cluster="failsafe" />          或:
<dubbo:reference cluster="failsafe" />
```

l Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

```
<dubbo:service cluster="failback" />          或:
         <dubbo:reference cluster="failback" />
```

l Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过forks="2"来设置最大并行数。

```
<dubbo:service cluster="forking" forks="2"/>          或:
         <dubbo:reference cluster="forking" forks="2"/>
```

l 配置

```
服务端服务级别
<dubbo:service interface="..."  oadbalance="roundrobin" />
客户端服务级别
<dubbo:reference interface="..." loadbalance="roundrobin" />
服务端方法级别
<dubbo:service interface="...">   <dubbo:method name="..." loadbalance="roundrobin"/>
客户端方法级别
<dubbo:reference interface="..">    <dubbo:method name="..."  loadbalance="roundrobin"/>
```

1. dubbo 通信协议 dubbo 协议为什么要消费者比提供者个数多：

因 dubbo 协议采用单一长连接，假设网络为千兆网卡(1024Mbit=128MByte)，

根据测试经验数据每条连接最多只能压满 7MByte(不同的环境可能不一样，供参考)，理论上 1 个服务提供者需要 20

个服务消费者才能压满网卡。

2. dubbo 通信协议 dubbo 协议为什么不能传大包：

因 dubbo 协议采用单一长连接，

如果每次请求的数据包大小为 500KByte，假设网络为千兆网卡(1024Mbit=128MByte)，每条连接最大 7MByte(不同的环境可能不一样，供参考)，

单个服务提供者的 TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。

单个消费者调用单个服务提供者的 TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

3. dubbo 通信协议 dubbo 协议为什么采用异步单一长连接：

因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，

比如 Morgan 的提供者只有 6 台提供者，却有上百台消费者，每天有 1.5 亿次调用，如果采用常规的 hessian 服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，

并使用异步 IO，复用线程池，防止 C10K 问题。

4. dubbo 通信协议 dubbo 协议适用范围和适用场景

适用范围：传入传出参数数据包较小（建议小于100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用dubbo协议传输大文件或超大字符串。

适用场景：常规远程服务方法调用 dubbo协议补充：

连接个数：单连接连接方式：长连接传输协议：TCP

传输方式：NIO 异步传输

序列化：Hessian 二进制序列化

5. RMI 协议

RMI协议采用JDK标准的java.rmi.*实现，采用阻塞式短连接和JDK标准序列化方式，Java标准的远程调用协议。

连接个数：多连接连接方式：短连接传输协议：TCP 传输方式：同步传输序列化：Java标准二进制序列化

适用范围：传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。

适用场景：常规远程服务方法调用，与原生RMI服务互操作

6. Hessian 协议

Hessian协议用于集成Hessian的服务，Hessian底层采用Http通讯，采用

Servlet暴露服务，Dubbo缺省内嵌Jetty作为服务器实现基于Hessian的远程调用协议。

连接个数：多连接连接方式：短连接传输协议：HTTP 传输方式：同步传输

序列化：Hessian二进制序列化适用范围：传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。

适用场景：页面传输，文件传输，或与原生hessian服务互操作

7. http

采用Spring的HttpInvoker实现基于http表单的远程调用协议。

连接个数：多连接连接方式：短连接传输协议：HTTP 传输方式：同步传输序列化：表单序列化（JSON）

适用范围：传入传出参数数据包大小混合，提供者比消费者个数多，可用浏览器查看，可用表单或URL传入参数，暂不支持传文件。适用场景：需同时给应用程序和浏览器JS使用的服务。

8. Webservice

基于CXF的frontend-simple和transports-http实现基于WebService的远程调用协议。

连接个数：多连接连接方式：短连接传输协议：HTTP 传输方式：同步传输

序列化：SOAP文本序列化适用场景：系统集成，跨语言调用。

9. Thrif

Thrift是Facebook捐给Apache的一个RPC框架，当前 dubbo 支持的 thrift 协议是对 thrift 原生协议的扩展，在原生协议的基础上添加了一些额外的头信息，比如service name，magic number等

Thrift

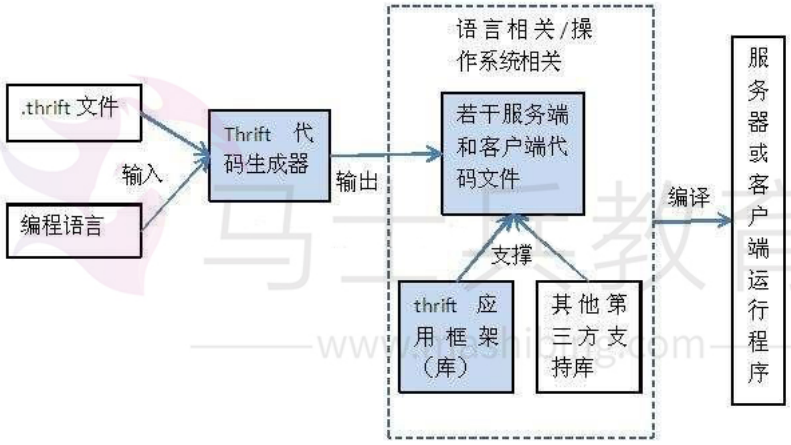
Apache Thrift 是 Facebook 实现的一种高效的、支持多种编程语言的远程服务调用的框架。本文将从 Java 开发人员角度详细介绍 Apache Thrift 的架构、开发和部署，并且针对不同的传输协议和服务类型给出相应的 Java 实例，同时详细介绍 Thrift 异步客户端的实现，最后提出使用 Thrift 需要注意的事项。

目前流行的服务调用方式有很多种，例如基于 SOAP 消息格式的 Web Service，基于 JSON 消息格式的 RESTful 服务等。其中所用到的数据传输方式包括 XML，JSON 等，然而 XML 相对体积太大，传输效率低，JSON 体积较小，新颖，但还不够完善。本文将介绍由 Facebook 开发的远程服务调用框架 Apache Thrift，它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk 等创建高效的、无缝的服务，其传输数据采用二进制格式，相对 XML 和 JSON 体积更小，对于高并发、大数据量和多语言的环境更有优势。本文将详细介绍 Thrift 的使用，并且提供丰富的实例代码加以解释说明，帮助使用者快速构建服务。

为什么要 Thrift：

1、多语言开发的需要

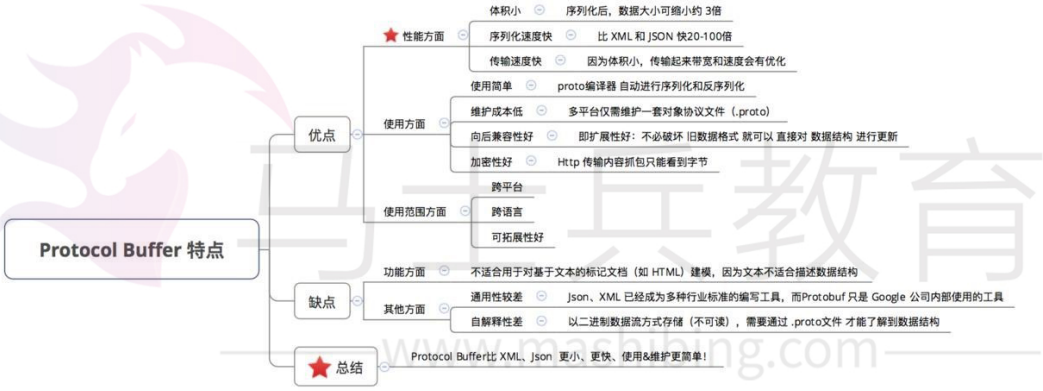
2、性能问题



Protoclol Buffer

protocol buffer 是 google 的一个开源项目,它是用于结构化数据串行化的灵活、高效、自动的方法，例如 XML，不过它比 xml 更小、更快、也更简单。你可以定义自己的数据结构，然后使用代码生成器生成的代码来读写这个数据结构。你甚至可以在无需重新部署程序的情况下更新数据结构。

特点



Protocol Buffer 的序列化 & 反序列化简单 & 速度快的原因是：

1. 编码 / 解码 方式简单（只需要简单的数学运算 = 位移等等）
2. 采用 Protocol Buffer 自身的框架代码 和 编译器 共同完成

Protocol Buffer 的数据压缩效果好（即序列化后的数据量体积小）的原因是：

1. a. 采用了独特的编码方式，如 Varint、Zigzag 编码方式等等
2. b. 采用 T - L - V 的数据存储方式：减少了分隔符的使用 & 数据存储得紧凑

Mybatis（连老师）

JPA 原理

事务

事务是计算机应用中不可或缺的组件模型，它保证了用户操作的原子性 (Atomicity)、一致性

(Consistency)、隔离性 (Isolation) 和持久性 (Durabilily)。

本地事务

紧密依赖于底层资源管理器（例如数据库连接），事务处理局限在当前事务资源内。此种事务处理方式不存在对应用服务器的依赖，因而部署灵活却无法支持多数据源的分布式事务。在数据库连接中使用本地事务示例如下：

```
  public void transferAccount() {
        Connection conn = null;         Statement stmt = null;
        try{
            conn = getDataSource().getConnection();
 // 将自动提交设置为 false，若设置为 true 则数据库将会把每一次数据更新认定为一个事务并自动提交
conn.setAutoCommit(false);              stmt = conn.createStatement();
            // 将 A 账户中的金额减少 500
        stmt.execute("update t_account set amount = amount - 500 where account_id = 'A'");
            // 将 B 账户中的金额增加 500
        stmt.execute("update t_account set amount = amount + 500 where account_id = 'B'");
            // 提交事务
            conn.commit();
               // 事务提交：转账的两步操作同时成功
        } catch(SQLException sqle){
                // 发生异常，回滚在本事务中的操做
conn.rollback();
               // 事务回滚：转账的两步操作完全撤销
stmt.close();                                                   conn.close();
        }
    }
```

分布式事务

Java 事务编程接口（JTA：Java Transaction API）和 Java 事务服务 (JTS；Java Transaction Service) 为 J2EE 平台提供了分布式事务服务。分布式事务（Distributed Transaction）包括事务管理器（Transaction Manager）和一个或多个支持 XA 协议的资源管理器 (Resource Manager)。我们可以将资源管理器看做任意类型的持久化数据存储；事务管理器承担着所有事务参与单元的协调与控制。

```
public void transferAccount() {
         UserTransaction userTx = null;
         Connection connA = null; Statement stmtA = null;            Connection connB = null; Statement stmtB
= null;
 try{
// 获得 Transaction 管理对象       userTx = (UserTransaction)getContext().lookup("java:comp/UserTransaction");
connA = getDataSourceA().getConnection();// 从数据库 A 中取得数据库连接       connB =
getDataSourceB().getConnection();// 从数据库 B 中取得数据库连接
userTx.begin();   // 启动事务  stmtA = connA.createStatement();// 将 A 账户中的金额减少 500
stmtA.execute("update t_account set amount = amount - 500 where account_id = 'A'");
             // 将 B 账户中的金额增加 500
             stmtB = connB.createStatement();
stmtB.execute("update t_account set amount = amount + 500 where account_id = 'B'");
             userTx.commit();// 提交事务
     // 事务提交：转账的两步操作同时成功（数据库 A 和数据库 B 中的数据被同时更新）
 } catch(SQLException sqle){
     // 发生异常，回滚在本事务中的操纵
                userTx.rollback();// 事务回滚：数据库 A 和数据库 B 中的数据更新被同时撤销      } catch(Exception ne){
}
}
```

两阶段提交

两阶段提交主要保证了分布式事务的原子性：即所有结点要么全做要么全不做，所谓的两个阶段是指：第一阶段：准备阶段；第二阶段：提交阶段。



1准备阶段

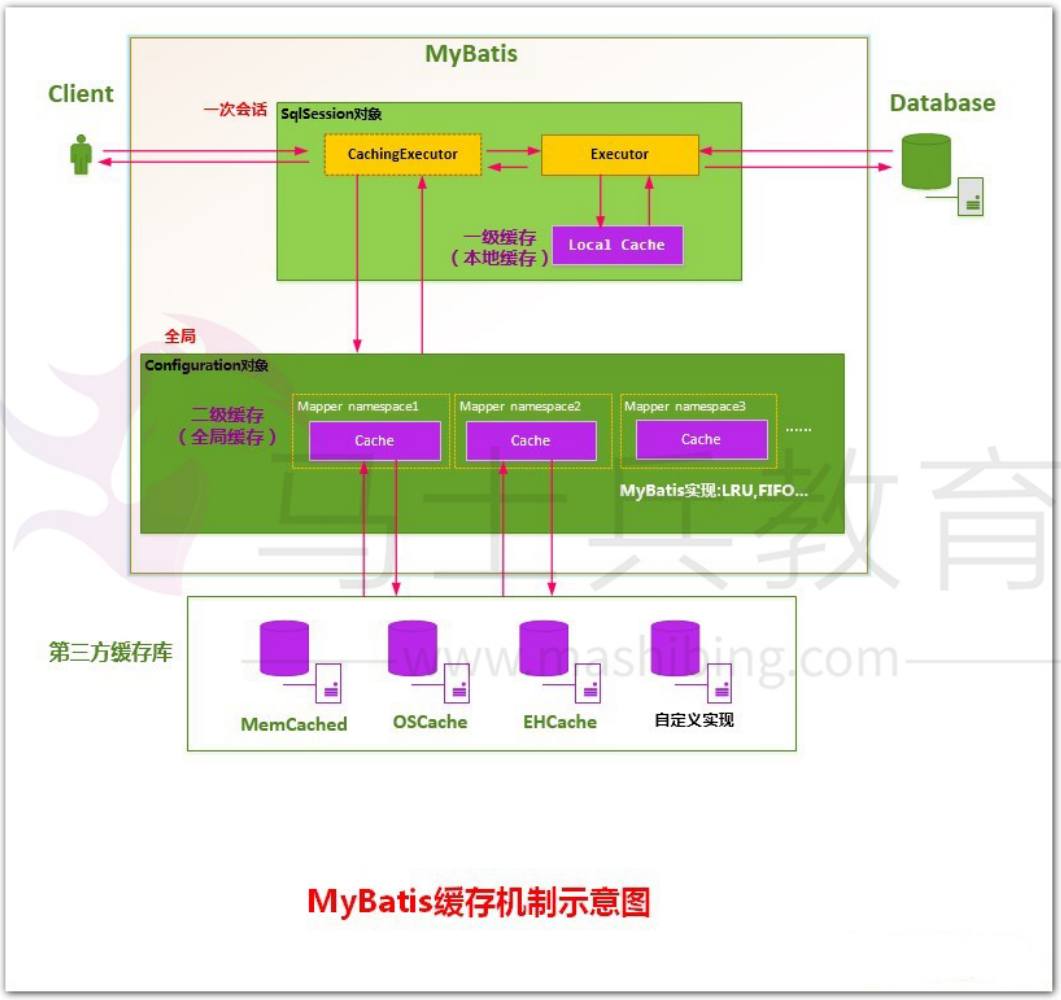
事务协调者(事务管理器)给每个参与者(资源管理器)发送 Prepare 消息，每个参与者要么直接返回失败(如权限验证失败)，要么在本地执行事务，写本地的 redo 和 undo 日志，但不提交，到达一种“万事俱备，只欠东风”的状态。

2提交阶段：

如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源) 将提交分成两阶段进行的目的很明确，就是尽可能晚地提交事务，让事务在提交前尽可能地完成所有能完成的工作。

Mybatis 缓存

Mybatis中有一级缓存和二级缓存，默认情况下一级缓存是开启的，而且是不能关闭的。一级缓存是指 SqlSession 级别的缓存，当在同一个 SqlSession 中进行相同的 SQL 语句查询时，第二次以后的查询不会从数据库查询，而是直接从缓存中获取，一级缓存最多缓存 1024 条 SQL。二级缓存是指可以跨 SqlSession 的缓存。是 mapper 级别的缓存，对于 mapper 级别的缓存不同的 sqlsession 是可以共享的。



Mybatis 的一级缓存原理（sqlsession级别）

第一次发出一个查询 sql，sql 查询结果写入 sqlsession 的一级缓存中，缓存使用的数据结构是一个 map。

key：MapperID+offset+limit+Sql+所有的入参

value：用户信息

同一个 sqlsession 再次发出相同的 sql，就从缓存中取出数据。如果两次中间出现 commit 操作（修改、添加、删除），本 sqlsession 中的一级缓存区域全部清空，下次再去缓存中查询不到所以要从数据库查询，从数据库查询到再写入缓存。

二级缓存原理（**mapper基本**）

二级缓存的范围是mapper 级别（mapper同一个命名空间），mapper以命名空间为单位创建缓存数据结构，结构是 map。mybatis 的二级缓存是通过 CacheExecutor 实现的。CacheExecutor 其实是 Executor 的代理对象。所有的查询操作，在 CacheExecutor 中都会先匹配缓存中是否存在，不存在则查询数据库。

key：MapperID+offset+limit+Sql+所有的入参

具体使用需要配置：

1. Mybatis 全局配置中启用二级缓存配置
2. 在对应的 Mapper.xml 中配置 cache 节点
3. 在对应的 select 查询节点中添加 useCache=true

MyBatis缓存

MyBatis缓存机制的设计与实现

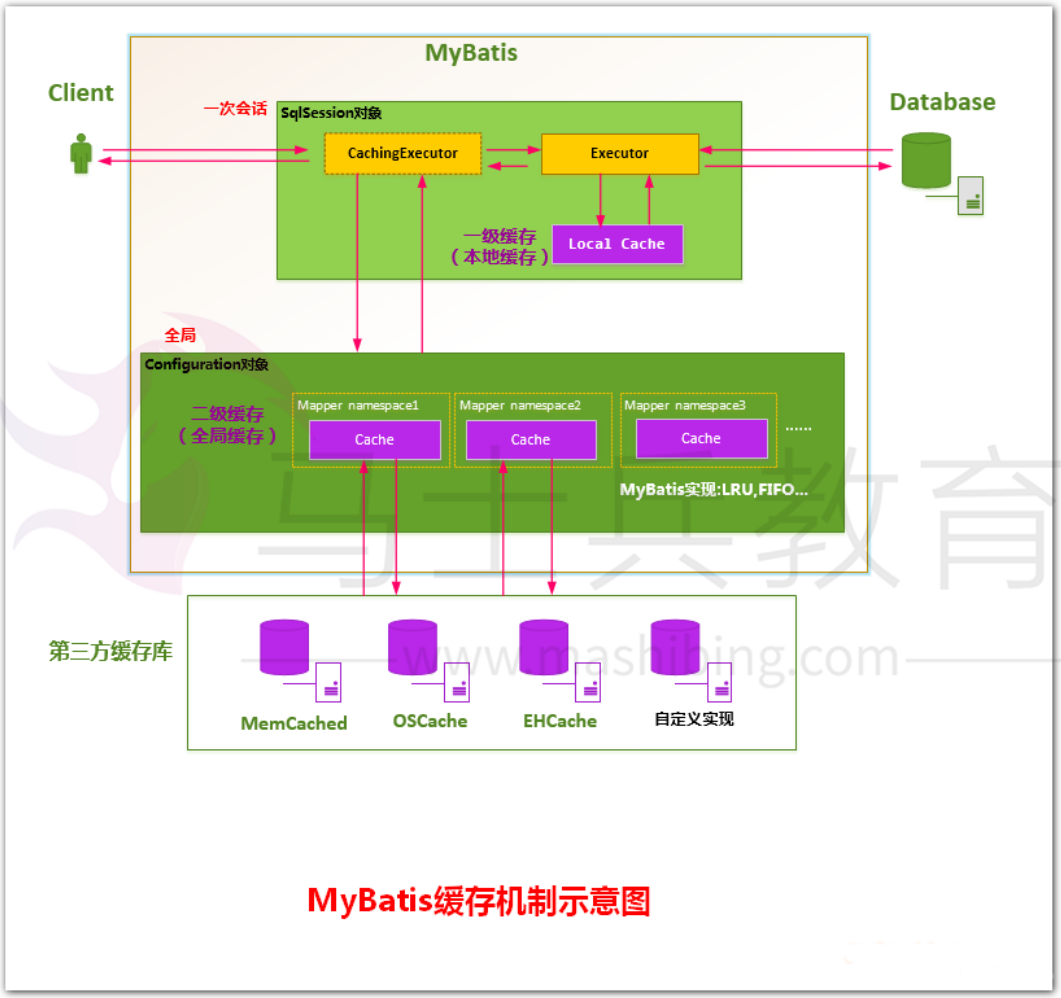
本文主要讲解***MyBatis***非常棒的缓存机制的设计原理，给读者们介绍一下***MyBatis***的缓存机制的轮廓，然后会分别针对缓存机制中的方方面面展开讨论。

MyBatis将数据缓存设计成两级结构，分为一级缓存、二级缓存：

一级缓存是Session**会话级别的缓存**，位于表示一次数据库会话的****SqlSession****对象之中，又被称之为本地缓存。一级缓存是****MyBatis****内部实现的一个特性，用户不能配置，默认情况下自动支持的缓存，用户没有定制它的权利（不过这也不是绝对的，可以通过开发插件对它进行修改）；

二级缓存是Application**应用级别的缓存**，它的是生命周期很长，跟****Application****的声明周期一样，也就是说它的作用范围是整个****Application****应用。

MyBatis中一级缓存和二级缓存的组织如下图所示：



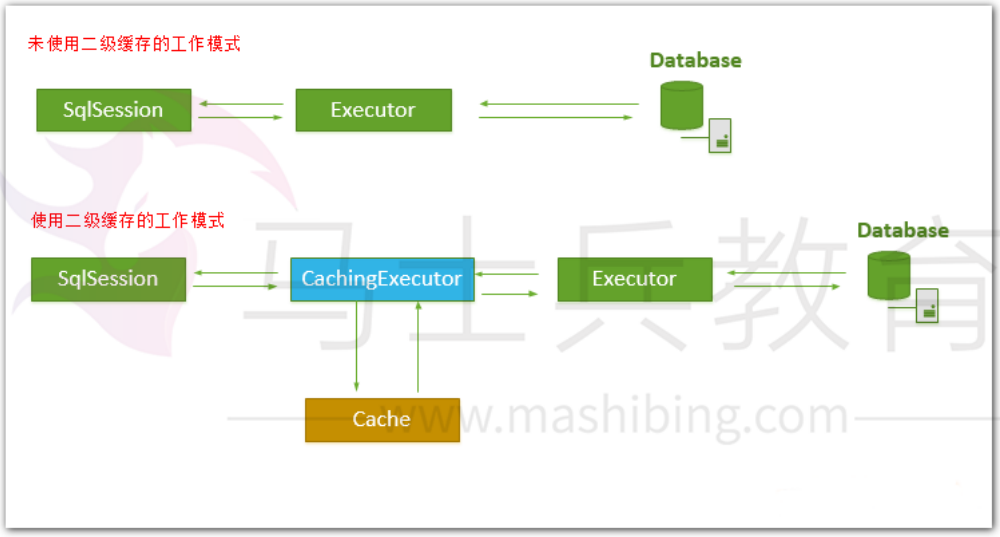
一级缓存的工作机制：

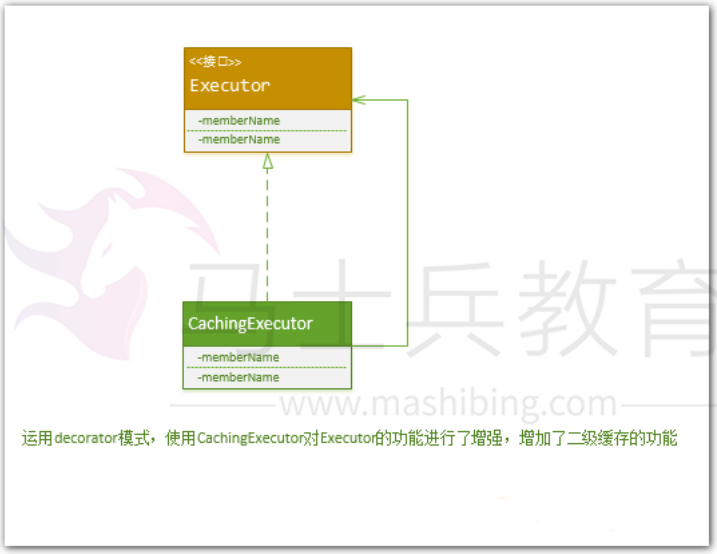
> 一级缓存是***Session***会话级别的，一般而言，一个***SqlSession***对象会使用一个***Executor***对象来完成会话操作，***Executor***对象会维护一个***Cache***缓存，以提高查询性能。

二级缓存的工作机制：

> 如上所言，一个***SqlSession***对象会使用一个***Executor***对象来完成会话操作，MyBatis的二级缓存机制的关键就是对这个***Executor***对象做文章。如果用户配置了"***cacheEnabled=true***"，那么MyBatis在为***SqlSession***对象创建***Executor***对象时，会对***Executor***对象加上一个装饰者：CachingExecutor，这时***SqlSession***使用CachingExecutor对象来完成操作请求。CachingExecutor对于查询请求，会先判断该查询请求在Application级别的二级缓存中是否有缓存结果，如果有查询结果，则直接返回缓存结果；如果缓存中没有，再交给真正的***Executor***对象来完成查询操作，之后CachingExecutor会将真正***Executor***返回的查询结果放置到缓存中，然后在返回给用户。

> **MyBatis**的二级缓存设计得比较灵活，你可以使用**MyBatis**自己定义的二级缓存实现；你也可以通过实现***org.apache.ibatis.cache.Cache***接口自定义缓存；也可以使用第三方内存缓存库，如***Memcached***等，这个我们会在后续的文章中详细讨论。





MyBatis数据源与连接池

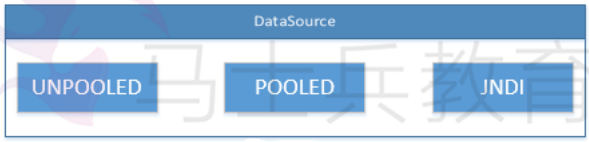
对于ORM框架而言，数据源的组织是一个非常重要的一部分，这直接影响到框架的性能问题。本文将通过对MyBatis框架的数据源结构进行详尽的分析，并且深入解析MyBatis的连接池。

本文首先会讲述MyBatis的数据源的分类，然后会介绍数据源是如何加载和使用的。紧接着将分类介绍UNPOOLED、POOLED和JNDI类型的数据源组织；期间我们会重点讲解POOLED类型的数据源和其实现的连接池原理。

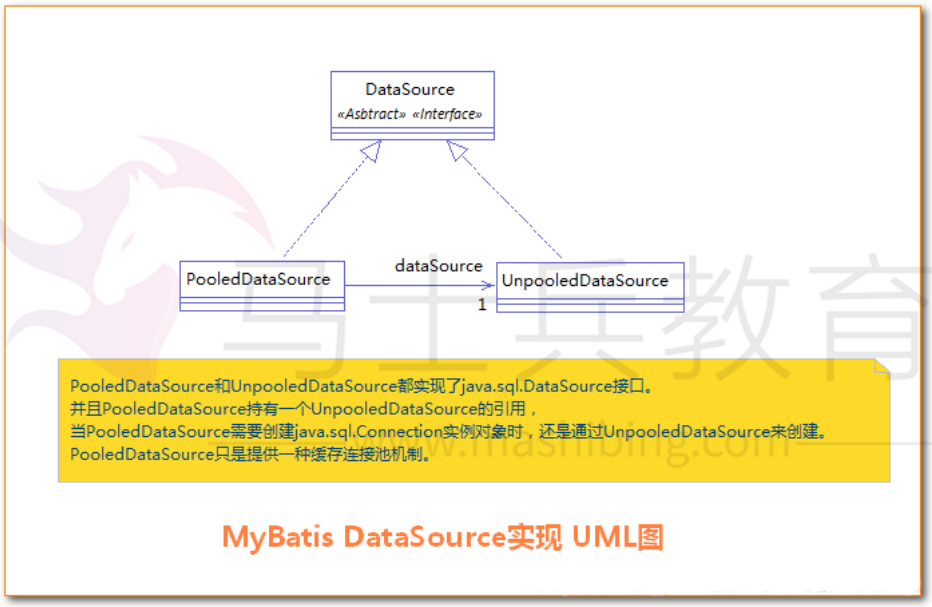
一、MyBatis数据源DataSource分类

> MyBatis数据源实现是在以下四个包中：
>
> ```
> org.apache.ibatis.datasource
>
> org.apache.ibatis.datasource.jndi
>
> org.apache.ibatis.datasource.pooled
>
> org.apache.ibatis.datasource.unpooled
> ```
>
> MyBatis把数据源DataSource分为三种：
>
> ž UNPOOLED　不使用连接池的数据源
>
> ž POOLED　使用连接池的数据源
>
> ž JNDI　　使用JNDI实现的数据源

即：



相应地，MyBatis内部分别定义了实现了java.sql.DataSource接口的UnpooledDataSource，PooledDataSource类来表示UNPOOLED、POOLED类型的数据源。



对于JNDI类型的数据源DataSource，则是通过JNDI上下文中取值。

二、数据源DataSource的创建过程

MyBatis*数据源DataSource**对象的创建发生在**MyBatis**初始化的过程中。***下面让我们一步步地了解MyBatis是如何创建数据源DataSource的。

在mybatis的XML配置文件中，使用元素来配置数据源：

1. MyBatis在初始化时，解析此文件，根据的type属性来创建相应类型的的数据源DataSource，即：

   - type="POOLED"：MyBatis会创建PooledDataSource实例
   - type="UNPOOLED"：MyBatis会创建UnpooledDataSource实例
   - type="JNDI" ：MyBatis会从JNDI服务上查找DataSource实例，然后返回使用

2. 顺便说一下，MyBatis是通过工厂模式来创建数据源DataSource对象的，MyBatis定义了抽象的工厂接口:***org.apache.ibatis.datasource.DataSourceFactory***,通过其getDataSource()方法返回数据源DataSource：

定义如下：

```
public interface DataSourceFactory {

 void setProperties(Properties props);
 //生产DataSource
 DataSource getDataSource();
}
```

上述三种不同类型的type，则有对应的以下dataSource工厂：

- POOLED PooledDataSourceFactory
- UNPOOLED UnpooledDataSourceFactory
- JNDI JndiDataSourceFactory

其类图如下所示：

3. MyBatis创建了DataSource实例后，会将其放到Configuration对象内的Environment对象中， 供以后使用。

三、 DataSource什么时候创建Connection对象

当我们需要创建SqlSession对象并需要执行SQL语句时，这时候MyBatis才会去调用dataSource对象来创建java.sql.Connection对象。也就是说，java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。

比如，我们有如下方法执行一个简单的SQL语句：

```
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
sqlSession.selectList("SELECT * FROM STUDENTS");
```

前4句都不会导致java.sql.Connection对象的创建，只有当第5句sqlSession.selectList("SELECT * FROM STUDENTS")，才会触发MyBatis在底层执行下面这个方法来创建java.sql.Connection对象：

```
protected void openConnection() throws SQLException {
if (log.isDebugEnabled()) {
log.debug("Opening JDBC Connection");
}
connection = dataSource.getConnection();
if (level != null) {
connection.setTransactionIsolation(level.getLevel());
}
setDesiredAutoCommit(autoCommmit);
}
```

而对于DataSource的UNPOOLED的类型的实现-UnpooledDataSource是怎样实现getConnection()方法的呢？请看下一节。

四、不使用连接池的UnpooledDataSource

当 的type属性被配置成了"UNPOOLED"，MyBatis首先会实例化一个UnpooledDataSourceFactory工厂实例，然后通过.getDataSource()方法返回一个UnpooledDataSource实例对象引用，我们假定为dataSource。

使用UnpooledDataSource的getConnection(),每调用一次就会产生一个新的Connection实例对象。

UnPooledDataSource的getConnection()方法实现如下：

```
/*
UnpooledDataSource的getConnection()实现
*/
public Connection getConnection() throws SQLException
{
return doGetConnection(username, password);
}

private Connection doGetConnection(String username, String password) throws SQLException
{
//封装username和password成properties
Properties props = new Properties();
if (driverProperties != null)
{
props.putAll(driverProperties);
}
if (username != null)
{
props.setProperty("user", username);
}
if (password != null)
{
props.setProperty("password", password);
}
return doGetConnection(props);
}

/*
 *  获取数据连接
 */
private Connection doGetConnection(Properties properties) throws SQLException
{
    //1.初始化驱动
    initializeDriver();
    //2.从DriverManager中获取连接，获取新的Connection对象
    Connection connection = DriverManager.getConnection(url, properties);
    //3.配置connection属性
    configureConnection(connection);
    return connection;
```

```
}
```

如上代码所示，UnpooledDataSource会做以下事情：

1. **初始化驱动：** 判断driver驱动是否已经加载到内存中，如果还没有加载，则会动态地加载driver类，并实例化一个Driver对象，使用DriverManager.registerDriver()方法将其注册到内存中，以供后续使用。
2. **创建Connection对象：** 使用DriverManager.getConnection()方法创建连接。
3. **配置Connection对象：** 设置是否自动提交autoCommit和隔离级别isolationLevel。
4. **返回Connection对象。**

上述的序列图如下所示：

总结：从上述的代码中可以看到，***我们每调用一次**getConnection()**方法，都会通过**DriverManager.getConnection()**返回新的**java.sql.Connection**实例。**

五、为什么要使用连接池?

1. 创建一个java.sql.Connection实例对象的代价

首先让我们来看一下创建一个java.sql.Connection对象的资源消耗。我们通过连接Oracle数据库，创建创建Connection对象，来看创建一个Connection对象、执行SQL语句各消耗多长时间。代码如下：

```
public static void main(String[] args) throws Exception
{

String sql = "select * from hr.employees where employee_id < ? and employee_id >= ?";
PreparedStatement st = null;
ResultSet rs = null;

long beforeTimeOffset = -1L; //创建Connection对象前时间
long afterTimeOffset = -1L; //创建Connection对象后时间
long executeTimeOffset = -1L; //创建Connection对象后时间

Connection con = null;
Class.forName("oracle.jdbc.driver.OracleDriver");

beforeTimeOffset = new Date().getTime();
System.out.println("before:\t" + beforeTimeOffset);

con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");

afterTimeOffset = new Date().getTime();
System.out.println("after:\t\t" + afterTimeOffset);
System.out.println("Create Costs:\t\t" + (afterTimeOffset - beforeTimeOffset) + " ms");

st = con.prepareStatement(sql);
//设置参数
st.setInt(1, 101);
st.setInt(2, 0);
//查询，得出结果集
rs = st.executeQuery();
executeTimeOffset = new Date().getTime();
System.out.println("Exec Costs:\t\t" + (executeTimeOffset - afterTimeOffset) + " ms");

}
```

上述程序在我笔记本上的执行结果为：

```
before:         1404363138126
after:          1404363138376
Create Costs:           250 ms
Exec Costs:             170 ms
```

从此结果可以清楚地看出，创建一个Connection对象，用了**250 毫秒**；而执行SQL的时间用了**170毫秒**。

创建一个Connection对象用了250毫秒！这个时间对计算机来说可以说是一个**非常奢侈**的！

这仅仅是一个Connection对象就有这么大的代价，设想一下另外一种情况：如果我们在Web应用程序中，为用户的每一个请求就操作一次数据库，当有10000个在线用户并发操作的话，对计算机而言，仅仅创建Connection对象不包括做业务的时间就要损耗10000×250ms= 250 0000 ms = 2500 s = 41.6667 min,竟然要***41***分钟！！！如果对高用户群体使用这样的系统，简直就是开玩笑！

2. 问题分析：

创建一个java.sql.Connection对象的代价是如此巨大，是因为创建一个Connection对象的过程，在底层就相当于和数据库建立的通信连接，在建立通信连接的过程，消耗了这么多的时间，而往往我们建立连接后（即创建Connection对象后），就执行一个简单的SQL语句，然后就要抛弃掉，这是一个非常大的资源浪费！

3.解决方案:

对于需要频繁地跟数据库交互的应用程序，可以在创建了Connection对象，并操作完数据库后，可以不释放掉资源，而是将它放到内存中，当下次需要操作数据库时，可以直接从内存中取出Connection对象，不需要再创建了，这样就极大地节省了创建Connection对象的资源消耗。由于内存也是有限和宝贵的，这又对我们对内存中的Connection对象怎么有效地维护提出了很高的要求。我们将在内存中存放Connection对象的容器称之为 连接池（Connection Pool）。下面让我们来看一下MyBatis的线程池是怎样实现的。

六、使用了连接池的PooledDataSource

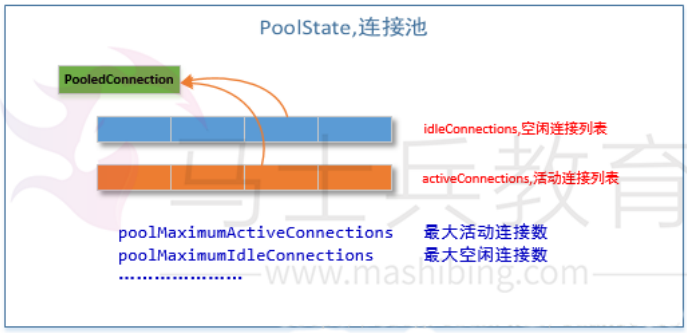
同样地，我们也是使用PooledDataSource的getConnection()方法来返回Connection对象。现在让我们看一下它的基本原理：

PooledDataSource将java.sql.Connection对象包裹成PooledConnection对象放到了PoolState类型的容器中维护。 MyBatis将连接池中的PooledConnection分为两种状态： 空闲状态（idle）和活动状态(active)，这两种状态的PooledConnection对象分别被存储到PoolState容器内的**idleConnections**和**activeConnections**两个List集合中：

idleConnections:空闲(idle)状态PooledConnection对象被放置到此集合中，表示当前闲置的没有被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从此集合中取PooledConnection对象。当用完一个java.sql.Connection对象时，MyBatis会将其包裹成PooledConnection对象放到此集合中。

activeConnections:活动(active)状态的PooledConnection对象被放置到名为activeConnections的ArrayList中，表示当前正在被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从idleConnections集合中取PooledConnection对象,如果没有，则看此集合是否已满，如果未满，PooledDataSource会创建出一个PooledConnection，添加到此集合中，并返回。

PoolState连接池的大致结构如下所示：



6.1 获取java.sql.Connection对象的过程

下面让我们看一下PooledDataSource 的getConnection()方法获取Connection对象的实现：

```
public Connection getConnection() throws SQLException {
return popConnection(dataSource.getUsername(), dataSource.getPassword()).getProxyConnection();
}

public Connection getConnection(String username, String password) throws SQLException {
return popConnection(username, password).getProxyConnection();
}
```

上述的popConnection()方法，会从连接池中返回一个可用的PooledConnection对象，然后再调用getProxyConnection()方法最终返回Conection对象。（至于为什么会有getProxyConnection(),请关注下一节）

现在让我们看一下popConnection()方法到底做了什么：

1. 先看是否有空闲(idle)状态下的PooledConnection对象，如果有，就直接返回一个可用的PooledConnection对象；否则进行第2步。
2. 查看活动状态的PooledConnection池activeConnections是否已满；如果没有满，则创建一个新的PooledConnection对象，然后放到activeConnections池中，然后返回此PooledConnection对象；否则进行第三步；
3. 看最先进入activeConnections池中的PooledConnection对象是否已经过期：如果已经过期，从activeConnections池中移除此对象，然后创建一个新的PooledConnection对象，添加到activeConnections中，然后将此对象返回；否则进行第4步。
4. 线程等待，循环2步

```
/*
 * 传递一个用户名和密码，从连接池中返回可用的PooledConnection
 */
private PooledConnection popConnection(String username, String password) throws SQLException
{
```

```
    boolean countedWait = false;
    PooledConnection conn = null;
    long t = System.currentTimeMillis();
    int localBadConnectionCount = 0;

    while (conn == null)
    {
        synchronized (state)
        {
            if (state.idleConnections.size() > 0)
            {
                // 连接池中有空闲连接，取出第一个
                conn = state.idleConnections.remove(0);
                if (log.isDebugEnabled())
                {
                    log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");
                }
            }
            else
            {
                // 连接池中没有空闲连接，则取当前正在使用的连接数小于最大限定值，
                if (state.activeConnections.size() < poolMaximumActiveConnections)
                {
                    // 创建一个新的connection对象
                    conn = new PooledConnection(dataSource.getConnection(), this);
                    @SuppressWarnings("unused")
                    //used in logging, if enabled
                    Connection realConn = conn.getRealConnection();
                    if (log.isDebugEnabled())
                    {
                        log.debug("Created connection " + conn.getRealHashCode() + ".");
                    }
                }
                else
                {
                    // Cannot create new connection 当活动连接池已满，不能创建时，取出活动连接池的第一个，即最先进入
连接池的PooledConnection对象
                    // 计算它的校验时间，如果校验时间大于连接池规定的最大校验时间，则认为它已经过期了，利用这个
PoolConnection内部的realConnection重新生成一个PooledConnection
                    //
                    PooledConnection oldestActiveConnection = state.activeConnections.get(0);
                    long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();
                    if (longestCheckoutTime > poolMaximumCheckoutTime)
                    {
                        // Can claim overdue connection
                        state.claimedOverdueConnectionCount++;
                        state.accumulatedCheckoutTimeOfOverdueConnections += longestCheckoutTime;
                        state.accumulatedCheckoutTime += longestCheckoutTime;
                        state.activeConnections.remove(oldestActiveConnection);
                        if (!oldestActiveConnection.getRealConnection().getAutoCommit())
                        {
                            oldestActiveConnection.getRealConnection().rollback();
                        }
                        conn = new PooledConnection(oldestActiveConnection.getRealConnection(), this);
                        oldestActiveConnection.invalidate();
                        if (log.isDebugEnabled())
                        {
                            log.debug("Claimed overdue connection " + conn.getRealHashCode() + ".");
                        }
                    }
                    else
                    {

                        //如果不能释放，则必须等待有
                        // Must wait
                        try
                        {
                            if (!countedWait)
                            {
                                state.hadToWaitCount++;
                                countedWait = true;
                            }
                            if (log.isDebugEnabled())
                            {
                                log.debug("Waiting as long as " + poolTimeToWait + " milliseconds for
connection.");
                            }
                            long wt = System.currentTimeMillis();
                            state.wait(poolTimeToWait);
                            state.accumulatedWaitTime += System.currentTimeMillis() - wt;
                        }
                        catch (InterruptedException e)
                        {
                            break;
                        }
                    }
                }
```

```
            }

            //如果获取PooledConnection成功，则更新其信息

            if (conn != null)
            {
                if (conn.isValid())
                {
                    if (!conn.getRealConnection().getAutoCommit())
                    {
                        conn.getRealConnection().rollback();
                    }
                    conn.setConnectionTypeCode(assembleConnectionTypeCode(dataSource.getUrl(), username,
password));
                    conn.setCheckoutTimestamp(System.currentTimeMillis());
                    conn.setLastUsedTimestamp(System.currentTimeMillis());
                    state.activeConnections.add(conn);
                    state.requestCount++;
                    state.accumulatedRequestTime += System.currentTimeMillis() - t;
                }
                else
                {
                    if (log.isDebugEnabled())
                    {
                        log.debug("A bad connection (" + conn.getRealHashCode() + ") was returned from
the pool, getting another connection.");
                    }
                    state.badConnectionCount++;
                    localBadConnectionCount++;
                    conn = null;
                    if (localBadConnectionCount > (poolMaximumIdleConnections + 3))
                    {
                        if (log.isDebugEnabled())
                        {
                            log.debug("PooledDataSource: Could not get a good connection to the
database.");
                        }
                        throw new SQLException("PooledDataSource: Could not get a good connection to the
database.");
                    }
                }
            }
        }

    }

    if (conn == null)
    {
        if (log.isDebugEnabled())
        {
            log.debug("PooledDataSource: Unknown severe error condition.  The connection pool returned a
null connection.");
        }
        throw new SQLException("PooledDataSource: Unknown severe error condition.  The connection pool
returned a null connection.");
    }

    return conn;
}
```

对应的处理流程图如下所示：

如上所示,对于PooledDataSource的getConnection()方法内，先是调用类PooledDataSource的popConnection()方法返回了一个PooledConnection对象，然后调用了PooledConnection的getProxyConnection()来返回Connection对象。

6.2java.sql.Connection对象的回收

当我们的程序中使用完Connection对象时，如果不使用数据库连接池，我们一般会调用 connection.close()方法，关闭connection连接，释放资源。如下所示：

```
private void test() throws ClassNotFoundException, SQLException
{
    String sql = "select * from hr.employees where employee_id < ? and employee_id >= ?";
    PreparedStatement st = null;
    ResultSet rs = null;

    Connection con = null;
    Class.forName("oracle.jdbc.driver.OracleDriver");
    try
    {
```

```
            con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan", "123456");
            st = con.prepareStatement(sql);
            //设置参数
            st.setInt(1, 101);
            st.setInt(2, 0);
            //查询，得出结果集
            rs = st.executeQuery();
            //取数据，省略
            //关闭，释放资源
            con.close();
        }
        catch (SQLException e)
        {
            con.close();
            e.printStackTrace();
        }
    }
```

调用过close()方法的Connection对象所持有的资源会被全部释放掉，Connection对象也就不能再使用。

那么，如果我们使用了连接池，我们在用完了Connection对象时，需要将它放在连接池中，该怎样做呢?

可能大家第一个在脑海里闪现出来的想法就是：我在应该调用con.close()方法的时候，不调用close()f方法，将其换成将Connection对象放到连接池容器中的代码！

好，我们将上述的想法实现，首先定义一个简易连接池Pool，然后将上面的代码改写：

```
package com.foo.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Vector;

/**
 *
 * 一个线程安全的简易连接池实现，此连接池是单例的
 *  putConnection()将Connection添加到连接池中
 *  getConnection()返回一个Connection对象
 */
public class Pool {

    private static Vector<Connection> pool = new Vector<Connection>();

    private static int MAX_CONNECTION =100;

    private static String DRIVER="oracle.jdbc.driver.OracleDriver";
    private static String URL = "jdbc:oracle:thin:@127.0.0.1:1521:xe";
    private static String USERNAME = "louluan";
    private static String PASSWROD = "123456";

    static {
        try {
            Class.forName(DRIVER);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * 将一个Connection对象放置到连接池中
     */
    public static  void putConnection(Connection connection){

        synchronized(pool)
        {
            if(pool.size()<MAX_CONNECTION)
            {
                pool.add(connection);
            }
        }
    }


    /**
     * 返回一个Connection对象，如果连接池内有元素，则pop出第一个元素：
     * 如果连接池Pool中没有元素，则创建一个connection对象，然后添加到pool中
     * @return Connection
     */
    public static Connection getConnection(){
        Connection connection = null;
        synchronized(pool)
        {
```

```
            if(pool.size()>0)
            {
                connection = pool.get(0);
                pool.remove(0);
            }
            else
            {
                connection = createConnection();
                pool.add(connection);
            }
        }
        return connection;
    }

    /**
     * 创建一个新的Connection对象
     */
    private static Connection createConnection()
    {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(URL, USERNAME,PASSWROD);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return connection;
    }

}
package com.foo.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Vector;

public class PoolTest
{

    private void test() throws ClassNotFoundException, SQLException
    {
        String sql = "select * from hr.employees where employee_id < ? and employee_id >= ?";
        PreparedStatement st = null;
        ResultSet rs = null;
        Connection con = null;
        Class.forName("oracle.jdbc.driver.OracleDriver");
        try
        {
            con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "louluan",
"123456");
            st = con.prepareStatement(sql);
            //设置参数
            st.setInt(1, 101);
            st.setInt(2, 0);
            //查询，得出结果集
            rs = st.executeQuery();
            //取数据，省略
            //将不再使用的Connection对象放到连接池中，供以后使用
            Pool.putConnection(con);
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}
```

上述的代码就是将我们使用过的Connection对象放到Pool连接池中，我们需要Connection对象的话，只需要使用Pool.getConnection()方法从里面取即可。

是的,上述的代码完全可以实现此能力，不过有一个很不优雅的实现：**就是我们需要手动地将Connection对象放到Pool连接池中，这是一个很傻的实现方式。这也和一般使用Connection对象的方式不一样：一般使用Connection的方式是使用完后，然后调用.close()方法释放资源。**

为了和一般的使用Conneciton对象的方式保持一致，我们希望当Connection使用完后，调用.close()方法，而实际上Connection资源并没有被释放，而实际上被添加到了连接池中。这样可以做到吗？答案是可以。上述的要求从另外一个角度来描述就是：能否提供一种机制，让我们知道Connection对象调用了什么方法，从而根据不同的方法自定义相应的处理机制。恰好代理机制就可以完成上述要求.

怎样实现Connection对象调用了close()方法，而实际是将其添加到连接池中

这是要使用代理模式，为真正的Connection对象创建一个代理对象，代理对象所有的方法都是调用相应的真正Connection对象的方法实现。当代理对象执行close()方法时，要特殊处理，不调用真正Connection对象的close()方法，而是将Connection对象添加到连接池中。

MyBatis的PooledDataSource的PoolState内部维护的对象是PooledConnection类型的对象，而PooledConnection则是对真正的数据库连接java.sql.Connection实例对象的包裹器。

PooledConnection对象内持有一个真正的数据库连接java.sql.Connection实例对象和一个java.sql.Connection的代理：

其部分定义如下：

```
class PooledConnection implements InvocationHandler {

  //......
  //所创建它的datasource引用
  private PooledDataSource dataSource;
  //真正的Connection对象
  private Connection realConnection;
  //代理自己的代理Connection
  private Connection proxyConnection;

  //......
}
```

PooledConenction

实现了

InvocationHandler

接口，并且，

proxyConnection

对象也是根据这个它来生成的代理对象：

```
public PooledConnection(Connection connection, PooledDataSource dataSource) {
    this.hashCode = connection.hashCode();
    this.realConnection = connection;
    this.dataSource = dataSource;
    this.createdTimestamp = System.currentTimeMillis();
    this.lastUsedTimestamp = System.currentTimeMillis();
    this.valid = true;
    this.proxyConnection = (Connection) Proxy.newProxyInstance(Connection.class.getClassLoader(), IFACES,
this);
  }
```

实际上，我们调用PooledDataSource的getConnection()方法返回的就是这个proxyConnection对象。

当我们调用此proxyConnection对象上的任何方法时，都会调用PooledConnection对象内invoke()方法。

让我们看一下PooledConnection类中的invoke()方法定义：

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String methodName = method.getName();
    //当调用关闭的时候，回收此Connection到PooledDataSource中
    if (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) {
      dataSource.pushConnection(this);
      return null;
    } else {
      try {
        if (!Object.class.equals(method.getDeclaringClass())) {
          checkConnection();
        }
        return method.invoke(realConnection, args);
      } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
      }
    }
  }
```

从上述代码可以看到，当我们使用了pooledDataSource.getConnection()返回的Connection对象的close()方法时，不会调用真正Connection的close()方法，而是将此Connection对象放到连接池中。

七、JNDI类型的数据源DataSource

对于JNDI类型的数据源DataSource的获取就比较简单，MyBatis定义了一个JndiDataSourceFactory工厂来创建通过JNDI形式生成的DataSource。

下面让我们看一下JndiDataSourceFactory的关键代码：

```
if (properties.containsKey(INITIAL_CONTEXT)
&& properties.containsKey(DATA_SOURCE))
{
//从JNDI上下文中找到DataSource并返回
Context ctx = (Context) initCtx.lookup(properties.getProperty(INITIAL_CONTEXT));
dataSource = (DataSource) ctx.lookup(properties.getProperty(DATA_SOURCE));
}
else if (properties.containsKey(DATA_SOURCE))
{
// //从JNDI上下文中找到DataSource并返回
dataSource = (DataSource) initCtx.lookup(properties.getProperty(DATA_SOURCE));
}
```

MyBatis简介

1. MyBatis是什么?

MyBatis 是一款优秀的持久层框架，一个半 ORM（对象关系映射）框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及 获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

2. ORM是什么

ORM（Object Relational Mapping），对象关系映射，是一种为了解决关系型数据库数 据与简单Java对象（POJO）的映射关系的技术。简单的说，ORM是通过使用描述对象和 数据库之间映射的元数据，将程序中的对象自动持久化到关系型数据库中。

3. 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里?

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时， 可以根据对象关系模型直接获取，所以它是全自动的。

而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半 自动ORM映射工具。

4. 传统JDBC开发存在的问题

频繁创建数据库连接对象、释放，容易造成系统资源浪费，影响系统性能。可以使用连接池 解决这个问题。但是使用jdbc需要自己实现连接池。

sql语句定义、参数设置、结果集处理存在硬编码。实际项目中sql语句变化的可能性较大， 一旦发生变化，需要修改java代码，系统需要重新编译，重新发布。不好维护。

使用preparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不一 定，可能多也可能少，修改sql还要修改代码，系统不易维护。

结果集处理存在重复代码，处理麻烦。如果可以映射成Java对象会比较方便。

5. JDBC编程有哪些不足之处，MyBatis是如何解决这些问题的?

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接 池可解决此问题。
   解决：在mybatis-config.xml中配置数据链接池，使用连接池管理数据库连接。
2. Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变 java代码。
   解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。
3. 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需 要和参数一一对应。
   解决： Mybatis自动将java对象映射至sql语句。
4. 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记 录封装成pojo对象解析比较方便。
   解决：Mybatis自动将sql执行结果映射至java对象。

6. Mybatis优缺点

优点

与传统的数据库访问技术相比，ORM有以下优点：

基于SQL语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在XML里，解除sql与程序代码的耦合，便于统一管理；提供XML标签，支持编写动态 SQL语句，并可重用

与JDBC相比，减少了50%以上的代码量，消除了JDBC大量冗余的代码，不需要手动开关连接

很好的与各种数据库兼容（因为MyBatis使用JDBC来连接数据库，所以只要JDBC支持的数 据库MyBatis都支持）

不同点

能够与Spring很好的集成

缺点

SQL语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写SQL语句的功底 有一定要求

SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库

7. MyBatis框架适用场景

MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。

对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

8. Hibernate 和 MyBatis 的区别

相同点

- 都是对jdbc的封装，都是持久层的框架，都用于dao层的开发。

不同点

映射关系

MyBatis 是一个半自动映射的框架，配置Java对象与sql语句执行结果的对应关系，多表关联关系配置简单

Hibernate 是一个全表映射的框架，配置Java对象与数据库表的对应关系，多表关联关系配置复杂

SQL优化和移植性

Hibernate 对SQL语句封装，提供了日志、缓存、级联（级联比 MyBatis 强大）等特性， 此外还提供 HQL（Hibernate Query Language）操作数据库，数据库无关性支持好，但 会多消耗性能。如果项目需要支持多种数据库，代码开发量少，但SQL语句优化困难。

MyBatis 需要手动编写 SQL，支持动态 SQL、处理列表、动态生成表名、支持存储过程。 开发工作量相对大些。直接使用SQL语句操作数据库，不支持数据库无关性，但sql语句优 化容易。

开发难易程度和学习成本

Hibernate 是重量级框架，学习使用门槛高，适合于需求相对稳定，中小型的项目，比如： 办公自动化系统

MyBatis 是轻量级框架，学习使用门槛低，适合于需求变化频繁，大型的项目，比如：互 联网电子商务系统

总结

MyBatis 是一个小巧、方便、高效、简单、直接、半自动化的持久层框架，

Hibernate 是一个强大、方便、高效、复杂、间接、全自动化的持久层框架。

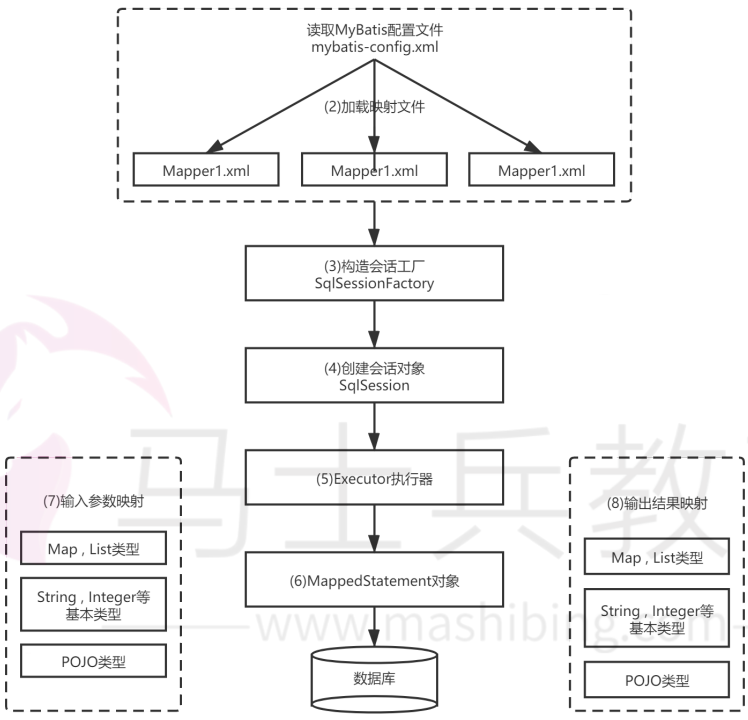
MyBatis的解析和运行原理

9. MyBatis编程步骤是什么样的?

1. 创建SqlSessionFactory
2. 通过SqlSessionFactory创建SqlSession
3. 通过sqlsession执行数据库操作
4. 调用session.commit()提交事务
5. 调用session.close()关闭会话

10. 请说说MyBatis的工作原理

在学习 MyBatis 程序之前，需要了解一下 MyBatis 工作原理，以便于理解程序。MyBatis 的工作原理如下图



1）读取 MyBatis 配置文件：mybatis-config.xml 为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，例如数据库连接信息。

2）加载映射文件。映射文件即 SQL 映射文件，该文件中配置了操作数据库的 SQL 语句， 需要在 MyBatis 配置文件 mybatis-config.xml 中加载。mybatis-config.xml 文件可以加 载多个映射文件，每个文件对应数据库中的一张表。

3）构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。

4）创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所 有方法。

5）Executor 执行器：MyBatis 底层定义了一个 Executor 接口来操作数据库，它将根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。

6）MappedStatement 对象：在 Executor 接口的执行方法中有一个 MappedStatement 类型的参数，该参数是对映射信息的封装，用于存储要映射的 SQL 语句的 id、参数等信 息。

7）输入参数映射：输入参数类型可以是 Map、List 等集合类型，也可以是基本数据类型和 POJO 类型。输入参数映射过程类似于 JDBC 对 preparedStatement 对象设置参数的过 程。

8）输出结果映射：输出结果类型可以是 Map、 List 等集合类型，也可以是基本数据类型 和 POJO 类型。输出结果映射过程类似于 JDBC 对结果集的解析过程。

MyBatis的功能架构是怎样的

我们把Mybatis的功能架构分为三层：

API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层 一接收到调用请求就会调用数据处理层来完成具体的数据处理。

数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的 目的是根据调用的请求完成一次数据库操作。

基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这 些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的 支撑。

MyBatis的框架架构设计是怎么样的

这张图从上往下看。MyBatis的初始化，会从mybatis-config.xml配置文件，解析构造成 Configuration这个类，就是图中的红框。

(1)加载配置：配置来源于两个地方，一处是配置文件，一处是Java代码的注解，将SQL的 配置信息加载成为一个个MappedStatement对象（包括了传入参数映射配置、执行的SQL 语句、结果映射配置），存储在内存中。

(2)SQL解析：当API接口层接收到调用请求时，会接收到传入SQL的ID和传入对象（可以是 Map、JavaBean或者基本数据类型），Mybatis会根据SQL的ID找到对应的 MappedStatement，然后根据传入参数对象对MappedStatement进行解析，解析后可以 得到最终要执行的SQL语句和参数。

(3)SQL执行：将最终得到的SQL和参数拿到数据库进行执行，得到操作数据库的结果。

(4)结果映射：将操作数据库的结果按照映射的配置进行转换，可以转换成HashMap、 JavaBean或者基本数据类型，并将最终结果返回。

为什么需要预编译

定义：

SQL 预编译指的是数据库驱动在发送 SQL 语句和参数给 DBMS 之前对 SQL 语句进行编 译，这样 DBMS 执行 SQL 时，就不需要重新编译。

为什么需要预编译

JDBC 中使用对象 PreparedStatement 来抽象预编译语句，使用预编译。预编译阶段可以 优化 SQL 的执行。预编译之后的 SQL 多数情况下可以直接执行，DBMS 不需要再次编 译，越复杂的SQL，编译的复杂度将越大，预编译阶段可以合并多次操作为一个操作。同时 预编译语句对象可以重复利用。把一个 SQL 预编译后产生的 PreparedStatement 对象缓 存下来，下次对于同一个SQL，可以直接使用这个缓存的 PreparedState 对象。Mybatis 默认情况下，将对所有的 SQL 进行预编译。

Mybatis都有哪些Executor执行器？它们之间的区别是什么?

Mybatis有三种基本的Executor执行器，SimpleExecutor、ReuseExecutor、 BatchExecutor。

ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使 用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。

BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添 加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个 Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行 executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。 **Mybatis中如何指定使用哪一种Executor执行器?**

在Mybatis配置文件中，在设置（settings）可以指定默认的ExecutorType执行器类型，也 可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参 数，如SqlSession openSession(ExecutorType execType) 。

配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句 （prepared statements）； BATCH 执行器将重用语句并执行批量更新。

Mybatis是否支持延迟加载？如果支持，它的实现原理是什么?

Mybatis仅支持association关联对象和collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在Mybatis配置文件中，可以配置是否 启用延迟加载lazyLoadingEnabled=true|false。

它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法， 比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独 发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b 属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。 **映射器**

#{}和${}的区别

#{}是占位符，预编译处理；${}是拼接符，字符串替换，没有预编译处理。

Mybatis在处理#{}时，#{}传入参数是以字符串传入，会将SQL中的#{}替换为?号，调用 PreparedStatement的set方法来赋值。

Mybatis在处理时，是原值传入，就是把 {}时，是原值传入，就是把时，是原值传入，就是 把{}替换成变量的值，相当于JDBC中的Statement编译

变量替换后，#{} 对应的变量自动加上单引号 ‘’；变量替换后，${} 对应的变量不会加上 单引号 ‘’

#{} 可以有效的防止SQL注入，提高系统安全性；${} 不能防止SQL 注入

#{} 的变量替换是在DBMS 中；${} 的变量替换是在 DBMS 外

模糊查询like语句该怎么写

（1）’%${question}%’ 可能引起SQL注入，不推荐

（2）"%"#{question}"%" 注意：因为#{...}解析成sql语句时候，会在变量外侧自动加单引 号’ ’，所以这里 % 需要使用双引号" "，不能使用单引号 ’ ’，不然会查不到任何结果。

（3）CONCAT(’%’,#{question},’%’) 使用CONCAT()函数，推荐

（4）使用bind标签

```
1 <select id="listUserLikeUsername" resultType="com.jourwon.pojo.User">
2 <bind name="pattern" value="'%' + username + '%'" />
3 select id,sex,age,username,password from person where username LIKE #
{pattern}
4 </select>
```

在mapper中如何传递多个参数

方法1：顺序传参法

```
1 public User selectUser(String name, int deptId);
2
3 <select id="selectUser" resultMap="UserResultMap">
4 select * from user
5 where user_name = #{0} and dept_id = #{1}
6 </select>
```

#{}里面的数字代表传入参数的顺序。

这种方法不建议使用，sql层表达不直观，且一旦顺序调整容易出错。

方法2：@Param注解传参法

```
1 public User selectUser(@Param("userName") String name, int @Param("deptI
d") deptId);
2
3 <select id="selectUser" resultMap="UserResultMap">
4 select * from user
5 where user_name = #{userName} and dept_id = #{deptId}
6 </select>
```

#{}里面的名称对应的是注解@Param括号里面修饰的名称。

这种方法在参数不多的情况还是比较直观的，推荐使用。

方法3：Map传参法

```
1 public User selectUser(Map<String, Object> params);
2
3 <select id="selectUser" parameterType="java.util.Map" resultMap="UserResu
ltMap">
4 select * from user
5 where user_name = #{userName} and dept_id = #{deptId}
6 </select>
```

#{}里面的名称对应的是Map里面的key名称。

这种方法适合传递多个参数，且参数易变能灵活传递的情况。

方法4：Java Bean传参法

```
1 public User selectUser(User user);
2
3 <select id="selectUser" parameterType="com.jourwon.pojo.User"
resultMap="UserResultMap">
4 select * from user
5 where user_name = #{userName} and dept_id = #{deptId}
6 </select>
7 <select id="selectUser" parameterType="com.jourwon.pojo.User"
resultMap="UserResultMap">
8
```

#{}里面的名称对应的是User类里面的成员属性。

这种方法直观，需要建一个实体类，扩展不容易，需要加属性，但代码可读性强，业务逻辑 处理方便，推荐使用。

Mybatis如何执行批量操作

使用foreach标签

foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。foreach标签的 属性主要有item，index，collection，open，separator，close。

item　　表示集合中每一个元素进行迭代时的别名，随便起的变量名；

index　　指定一个名字，用于表示在迭代过程中，每次迭代到的位置，不常用； open　　表示该语句以什么开始，常用“(”；

separator表示在每次进行迭代之间以什么符号作为分隔符，常用“,”；

close　　表示以什么结束，常用“)”。

在使用foreach的时候最关键的也是最容易出错的就是collection 属性，该属性是必须指定 的，但是在不同情况下，该属性的值是不一样的，主要有一下3种情况：

如果传入的是单参数且参数类型是一个List的时候，collection 属性值为list

如果传入的是单参数且参数类型是一个array数组的时候，collection 的属性值为array 如果传入的参数是多个的时候，我们就需要把它们封装成一个Map了，当然单参数也可以 封装成map，实际上如果你在传入参数的时候，在MyBatis里面也是会把它封装成一个 Map的，

map的key就是参数名，所以这个时候collection属性值就是传入的List或array对象在自己 封装的map里面的key

具体用法如下：

```
1 <!-- 批量保存(foreach插入多条数据两种方法)
2 int addEmpsBatch(@Param("emps") List<Employee> emps); -->
3 <!-- MySQL下批量保存，可以foreach遍历 mysql支持values(),(),()语法 --> //推荐
使用
4 <insert id="addEmpsBatch">
5 INSERT INTO emp(ename,gender,email,did)
6 VALUES
7 <foreach collection="emps" item="emp" separator=",">
8 (#{emp.eName},#{emp.gender},#{emp.email},#{emp.dept.id})
9 </foreach>
10 </insert>
```

```
1 <!-- 这种方式需要数据库连接属性allowMutiQueries=true的支持
2 如jdbc.url=jdbc:mysql://localhost:3306/mybatis?allowMultiQueries=true --
>
3 <insert id="addEmpsBatch">
4 <foreach collection="emps" item="emp" separator=";">
5 INSERT INTO emp(ename,gender,email,did)
6 VALUES(#{emp.eName},#{emp.gender},#{emp.email},#{emp.dept.id})
7 </foreach>
8 </insert>
```

使用ExecutorType.BATCH

Mybatis内置的ExecutorType有3种，默认为simple,该模式下它为每个语句的执行创建一 个新的预处理语句，单条提交sql；而batch模式重复使用已经预处理的语句，并且批量执行 所有更新语句，显然batch性能将更优； 但batch模式也有自己的问题，比如在Insert操作 时，在事务没有提交之前，是没有办法获取到自增的id，这在某型情形下是不符合业务要求 的

具体用法如下

```
1 //批量保存方法测试
2 @Test
3 public void testBatch() throws IOException{
4 SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();
5 //可以执行批量操作的sqlSession
6 SqlSession openSession = sqlSessionFactory.openSession(ExecutorType.BATC
H);
7
8 //批量保存执行前时间
9 long start = System.currentTimeMillis();
10 try {
11 EmployeeMapper mapper = openSession.getMapper(EmployeeMapper.class);
12 for (int i = 0; i < 1000; i++) {
13 mapper.addEmp(new Employee(UUID.randomUUID().toString().substring(0,
5), "b", "1"));
14 }
15
16 openSession.commit();
17 long end = System.currentTimeMillis();
18 //批量保存执行后的时间
19 System.out.println("执行时长" + (end - start));
20 //批量 预编译sql一次==》设置参数==》10000次==》执行1次 677
21 //非批量 （预编译=设置参数=执行 ）==》10000次 1121
22
23 } finally {
24 openSession.close();
25 }
26 }
```

mapper和mapper.xml如下

```
1 public interface EmployeeMapper {
2 //批量保存员工
3 Long addEmp(Employee employee);
4 }
1 <mapper namespace="com.jourwon.mapper.EmployeeMapper"
2 <!--批量保存员工 -->
3 <insert id="addEmp">
4 insert into employee(lastName,email,gender)
5 values(#{lastName},#{email},#{gender})
6 </insert>
7 </mapper>
```

如何获取生成的主键

对于支持主键自增的数据库（MySQL）

```
1 <insert id="insertUser" useGeneratedKeys="true" keyProperty="userId" >
2 insert into user(
3 user_name, user_password, create_time)
4 values(#{userName}, #{userPassword} , #{createTime, jdbcType=
TIMESTAMP})
5 </insert>
```

parameterType 可以不写，Mybatis可以推断出传入的数据类型。如果想要访问主键，那 么应当parameterType 应当是java实体或者Map。这样数据在插入之后 可以通过ava实体 或者Map 来获取主键值。通过 getUserId获取主键

不支持主键自增的数据库（Oracle）

对于像Oracle这样的数据，没有提供主键自增的功能，而是使用序列的方式获取自增主 键。

可以使用 < selectKey > 标签来获取主键的值，这种方式不仅适用于不提供主键自增功能的 数据库，也适用于提供主键自增功能的数据库

< selectKey > 一般的用法

```
1 <selectKey keyColumn="id" resultType="long" keyProperty="id" order="BEFOR
E">
2 </selectKey>
```

属性 描述

keyProperty selectKey 语句结果应该被设置的目标属性。如果希望得到多个生成的列， 也可以是逗号分隔的属性名称列表。

keyColmn

匹配属性的返回结果集中的列名称。如果希望得到多个生成的列，也可以

是逗号分隔的属性名称列表。

resultType

结果的类型，MyBatis 通常可以推算出来。MyBatis 允许任何简单类型用

作主键的类型，包括字符串。如果希望作用于多个生成的列，则可以使用一个包含期望属性 的 Object 或一个 Map。

order

值可为BEFORE 或 AFTER。如果是 BEFORE，那么它会先执行selectKey设置

keyProperty 然后执行插入语句。如果为AFTER则相反。

statementType

使用何种语句类型，默认PREPARED。 有STATEMENT，PREPARED

和 CALLABLE 语句的映射类型。

```
1 <insert id="insertUser" >
2 <selectKey keyColumn="id" resultType="long" keyProperty="userId"
order="BEFORE">
3 SELECT USER_ID.nextval as id from dual
4 </selectKey>
5 insert into user(
6 user_id,user_name, user_password, create_time)
7 values(#{userId},#{userName}, #{userPassword} , #{createTime, jdbcType=
TIMESTAMP})
8 </insert>
```

此时会将Oracle生成的主键值赋予userId变量。这个userId 就是USER对象的属性，这样就 可以将生成的主键值返回了。如果仅仅是在insert语句中使用但是不返回，此时 keyProperty=“任意自定义变量名”，resultType 可以不写。

Oracle 数据库中的值要设置为 BEFORE ，这是因为 Oracle中需要先从序列获取值，然后 将值作为主键插入到数据库中。

扩展

如果Mysql 使用selectKey的方式获取主键，需要注意下面两点：

order： AFTER

获取递增主键值： SELECT LAST_INSERT_ID()

当实体类中的属性名和表中的字段名不一样 ， 怎么办

第1种： 通过在查询的SQL语句中定义字段名的别名，让字段名的别名和实体类的属性名一 致。

```
1 <select id="getOrder" parameterType="int" resultType="com.jourwon.pojo.Or
der">
2 select order_id id, order_no orderno ,order_price price form orders wher
e order_id=#{id};
3 </select>
```

第2种： 通过来映射字段名和实体类属性名的一一对应的关系。

```
1 <select id="getOrder" parameterType="int" resultMap="orderResultMap">
2 select * from orders where order_id=#{id}
3 </select>
4
5 <resultMap type="com.jourwon.pojo.Order" id="orderResultMap">
6 <!–用id属性来映射主键字段–>
7 <id property="id" column="order_id">
8
9 <!–用result属性来映射非主键字段，property为实体类属性名，column为数据库表中的
属性–>
10 <result property ="orderno" column ="order_no"/>
11 <result property="price" column="order_price" />
12 </reslutMap>
```

Mapper 编写有哪几种方式?

第一种：接口实现类继承 SqlSessionDaoSupport：使用此种方法需要编写mapper 接 口，mapper 接口实现类、mapper.xml 文件。

（1）在 sqlMapConfig.xml 中配置 mapper.xml 的位置

```
1 <mappers>
2 <mapper resource="mapper.xml 文件的地址" />
3 <mapper resource="mapper.xml 文件的地址" />
4 </mappers>
```

（2）定义 mapper 接口

（3）实现类集成 SqlSessionDaoSupport

mapper 方法中可以 this.getSqlSession() 进行数据增删改查。

（4）spring 配置

```
1 <bean id=" " class="mapper 接口的实现">
2 <property name="sqlSessionFactory"
3 ref="sqlSessionFactory"></property>
4 </bean>
```

第二种：使用 org.mybatis.spring.mapper.MapperFactoryBean：

（1）在 sqlMapConfig.xml 中配置 mapper.xml 的位置，如果 mapper.xml 和mappre 接口的名称相同且在同一个目录，这里可以不用配置

```
1 <mappers>
2 <mapper resource="mapper.xml 文件的地址" />
3 <mapper resource="mapper.xml 文件的地址" />
4 </mappers>
```

（2）定义 mapper 接口：

（3）mapper.xml 中的 namespace 为 mapper 接口的地址

（4）mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致

（5）Spring 中定义

```
1 <bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean">
2 <property name="mapperInterface" value="mapper 接口地址" />
3 <property name="sqlSessionFactory" ref="sqlSessionFactory" />
4 </bean>
```

第三种：使用 mapper 扫描器：

（1）mapper.xml 文件编写：

mapper.xml 中的 namespace 为 mapper 接口的地址；

mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致；

如果将 mapper.xml 和 mapper 接口的名称保持一致则不用在 sqlMapConfig.xml中进行 配置。

（2）定义 mapper 接口：

注意 mapper.xml 的文件名和 mapper 的接口名称保持一致，且放在同一个目录 （3）配置 mapper 扫描器：

```
1 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
2 <property name="basePackage" value="mapper 接口包地址
3 "></property>
4 <property name="sqlSessionFactoryBeanName"
5 value="sqlSessionFactory"/>
6 </bean>
```

（4）使用扫描器后从 spring 容器中获取 mapper 的实现对象。

什么是MyBatis的接口绑定？有哪些实现方式?

接口绑定，就是在MyBatis中任意定义接口，然后把接口里面的方法和SQL语句绑定，我们 直接调用接口方法就可以，这样比起原来了SqlSession提供的方法我们可以有更加灵活的 选择和设置。

接口绑定有两种实现方式

通过注解绑定，就是在接口的方法上面加上 @Select、@Update等注解，里面包含Sql语 句来绑定；

通过xml里面写SQL来绑定， 在这种情况下，要指定xml映射文件里面的namespace必须为 接口的全路径名。当Sql语句比较简单时候，用注解绑定， 当SQL语句比较复杂时候，用 xml绑定，一般用xml绑定的比较多。

使用MyBatis的mapper接口调用时有哪些要求?

1、Mapper接口方法名和mapper.xml中定义的每个sql的id相同。

2、Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的 类型相同。

3、Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型 相同。

4、Mapper.xml文件中的namespace即是mapper接口的类路径。

最佳实践中，通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口 的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗

Dao接口，就是人们常说的Mapper接口，接口的全限名，就是映射文件中的namespace 的值，接口的方法名，就是映射文件中MappedStatement的id值，接口方法内的参数，就 是传递给sql的参数。Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法 名拼接字符串作为key值，可唯一定位一个MappedStatement，举例： com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到namespace为 com.mybatis3.mappers.StudentDao下面id = findStudentById的MappedStatement。 在Mybatis中，每一个 、、、 标签均会被解析为MappedStatement对象，标签内的sql会被解析为BoundSql 对象。

Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式?

第一种是使用标签，逐一定义列名和对象属性名之间的映射关系。

第二种是使用sql列的别名功能，将列别名书写为对象属性名，比如T_NAME AS NAME， 对象属性名一般是name，小写，但是列名不区分大小写，Mybatis会忽略列名大小写，智 能找到与之对应对象属性名，你甚至可以写成T_NAME AS NaMe，Mybatis一样可以正常 工作。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性 逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

Xml映射文件中，除了常见的select|insert|updae|delete标签之外，还有哪些标签? 还有很多其他的标签，、、、、、，加上动态sql的9个标签， trim|where|set|foreach|if|choose|when|otherwise|bind 等，其中为sql片段标签， 通过标签引入sql片段，为不支持自增的主键生成策略标签。

Mybatis映射文件中，如果A标签通过include引用了B标签的内容，请问，B标签能否定义 在A标签的后面，还是说必须定义在A标签的前面?

虽然Mybatis解析Xml映射文件是按照顺序解析的，但是，被引用的B标签依然可以定义在 任何地方，Mybatis都可以正确识别。

原理是，Mybatis解析A标签，发现A标签引用了B标签，但是B标签尚未解析到，尚不存 在，此时，Mybatis会将A标签标记为未解析状态，然后继续解析余下的标签，包含B标 签，待所有标签解析完毕，Mybatis会重新解析那些被标记为未解析的标签，此时再解析A 标签时，B标签已经存在，A标签也就可以正常解析完成了。

高级查询

MyBatis实现一对一，一对多有几种方式，怎么操作的?

有联合查询和嵌套查询。联合查询是几个表联合查询，只查询一次，通过在resultMap里面 的association，collection节点配置一对一，一对多的类就可以完成

嵌套查询是先查一个表，根据这个表里面的结果的外键id，去再另外一个表里面查询数据， 也是通过配置association，collection，但另外一个表的查询通过select节点配置。

Mybatis是否可以映射Enum枚举类?

Mybatis可以映射枚举类，不单可以映射枚举类，Mybatis可以映射任何对象到表的一列 上。映射方式为自定义一个TypeHandler，实现TypeHandler的setParameter()和 getResult()接口方法。

TypeHandler有两个作用，一是完成从javaType至jdbcType的转换，二是完成jdbcType至 javaType的转换，体现为setParameter()和getResult()两个方法，分别代表设置sql问号占 位符参数和获取列查询结果。

动态SQL

Mybatis动态sql是做什么的？都有哪些动态sql？能简述一下动态sql的执行原理不？ Mybatis动态sql可以让我们在Xml映射文件内，以标签的形式编写动态sql，完成逻辑判断 和动态拼接sql的功能，Mybatis提供了9种动态sql标签
trim|where|set|foreach|if|choose|when|otherwise|bind 。

其执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态sql的功能。

插件模块

Mybatis是如何进行分页的？分页插件的原理是什么?

Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非 物理分页，可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分 页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法 内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页 参数。

举例：select * from student，拦截sql后重写为：select t.* from (select * from student) t limit 0, 10

简述Mybatis的插件运行原理，以及如何编写一个插件。

Mybatis仅可以编写针对ParameterHandler、ResultSetHandler、StatementHandler、 Executor这4种接口的插件，Mybatis使用JDK的动态代理，为需要拦截的接口生成代理对 象以实现接口方法拦截功能，每当执行这4种接口对象的方法时，就会进入拦截方法，具体 就是InvocationHandler的invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

实现Mybatis的Interceptor接口并复写intercept()方法，然后在给插件编写注解，指定要 拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

缓存

Mybatis的一级、二级缓存

1）一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session， 当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓 存。

2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储， 不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默 认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接 口(可用来保存对象的状态),可在它的映射文件中配置；

3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的 进行了C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

BeanFactory 和 ApplicationContext 有什么区别

> BeanFactory 可以理解为含有bean集合的工厂类。BeanFactory 包含了种bean的定义，以便在接收到客户端请求时将对应的bean实例化。

> BeanFactory还能在实例化对象的时生成协作类之间的关系。此举将bean自身与bean客户端的配置中解放出来。BeanFactory还包含了bean生命周期的控制，调用客户端的初始化方法（initialization methods）和销毁方法（destruction methods）。

> 从表面上看，application context如同bean factory一样具有bean定义、bean关联关系的设置，根据请求分发bean的功能。但application context在此基础上还提供了其他的功能。

> 提供了支持国际化的文本消息

> 统一的资源文件读取方式

> 已在监听器中注册的bean的事件

Spring Bean 的生命周期

> Spring Bean的生命周期简单易懂。在一个bean实例被初始化时，需要执行一系列的初始化操作以达到可用的状态。同样的，当一个bean不在被调用时需要进行相关的析构操作，并从bean容器中移除。

> Spring bean factory 负责管理在spring容器中被创建的bean的生命周期。Bean的生命周期由两组回调（call back）方法组成。

> 初始化之后调用的回调方法。

> 销毁之前调用的回调方法。

> Spring框架提供了以下四种方式来管理bean的生命周期事件：

> InitializingBean和DisposableBean回调接口

> 针对特殊行为的其他Aware接口

> Bean配置文件中的Custom init()方法和destroy()方法

> @PostConstruct和@PreDestroy注解方式

Spring IOC 如何实现

> Spring中的 org.springframework.beans 包和 org.springframework.context包构成了

Spring框架IoC容器的基础。

> BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。

ApplicationContex接口对BeanFactory（是一个子接口）进行了扩展，在BeanFactory的基础上添加了其他功能，比如与Spring的AOP更容易集成，也提供了处理message resource的机制（用于国际化）、事件传播以及应用层的特别配置，比如针对Web应用的

WebApplicationContext。

> org.springframework.beans.factory.BeanFactory 是Spring IoC容器的具体实现，用来包装和管理前面提到的各种bean。BeanFactory接口是Spring IoC 容器的核心接口。

说说 Spring AOP

> 面向切面编程，在我们的应用中，经常需要做一些事情，但是这些事情与核心业务无关，比如，要记录所有update*方法的执行时间时间，操作人等等信息，记录到日志，

> 通过spring的AOP技术，就可以在不修改update*的代码的情况下完成该需求。

Spring AOP 实现原理

> Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理。JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。

> 如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB （Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

动态代理（cglib 与 JDK）

> JDK 动态代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用Java动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，就不能使用该机制。而CGLIB则可以实现对类的动态代理。

Spring 事务实现方式

> 1、编码方式

> 所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。

> 2、声明式事务管理方式

> 声明式事务管理又有两种实现方式：基于xml配置文件的方式；另一个实在业务方法上进行@Transaction注解，将事务规则应用到业务逻辑中

Spring 事务底层原理

> a、划分处理单元——IOC

> 由于spring解决的问题是对单个数据库进行局部事务处理的，具体的实现首相用spring 中的IOC划分了事务处理单元。并且将对事务的各种配置放到了ioc容器中（设置事务管理器，设置事务的传播特性及隔离机制）。

> b、AOP拦截需要进行事务处理的类

> Spring事务处理模块是通过AOP功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用TransactionProxyFactoryBean接口来使用AOP

功能，生成proxy代理对象，通过TransactionInterceptor完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。读取ioc容器事务配置属性，转化为spring事务处理

需要的内部数据结构（TransactionAttributeSourceAdvisor），转化为 TransactionAttribute表示的数据对象。

> c、对事物处理实现（事务的生成、提交、回滚、挂起）

> spring委托给具体的事务处理器实现。实现了一个抽象和适配。适配的具体事务处理器：DataSource数据源支持、hibernate数据源事务处理支持、JDO数据源事务处理支持，JPA、JTA数据源事务处理支持。这些支持都是通过设计

PlatformTransactionManager、AbstractPlatforTransaction一系列事务处理的支持。 为常用数据源支持提供了一系列的TransactionManager。

> d、结合

> PlatformTransactionManager实现了TransactionInterception接口，让其与

TransactionProxyFactoryBean结合起来，形成一个Spring声明式事务处理的设计体系。

如何自定义注解实现功能

> 创建自定义注解和创建一个接口相似，但是注解的interface关键字需要以@符号开头。

> 注解方法不能带有参数；

> 注解方法返回值类型限定为：基本类型、String、Enums、Annotation或者是这些类型的数组；

> 注解方法可以有默认值；

> 注解本身能够包含元注解，元注解被用来注解其它注解。

Spring MVC 运行流程

> 1.spring mvc将所有的请求都提交给DispatcherServlet,它会委托应用系统的其他模块负责对请求 进行真正的处理工作。

> 2.DispatcherServlet查询一个或多个HandlerMapping,找到处理请求的Controller.

> 3.DispatcherServlet请请求提交到目标Controller

> 4.Controller进行业务逻辑处理后，会返回一个ModelAndView

> 5.Dispathcher查询一个或多个ViewResolver视图解析器,找到ModelAndView对象指定的视图对象

> 6.视图对象负责渲染返回给客户端。

Spring MVC 启动流程

> 在 web.xml 文件中给 Spring MVC 的 Servlet 配置了 load-on-startup,所以程序启动的 > 时候会初始化 Spring MVC，在 HttpServletBean 中将配置的 contextConfigLocation

> 属性设置到 Servlet 中，然后在 FrameworkServlet 中创建了 WebApplicationContext,

> DispatcherServlet 根据 contextConfigLocation 配置的 classpath 下的 xml 文件初始化了

> Spring MVC 总的组件。

Spring 的单例实现原理

> Spring 对 Bean 实例的创建是采用单例注册表的方式进行实现的，而这个注册表的缓存是

ConcurrentHashMap 对象。

Spring 框架中用到了哪些设计模式

> 代理模式—在AOP和remoting中被用的比较多。

> 单例模式—在spring配置文件中定义的bean默认为单例模式。 > 模板方法—用来解决代码重复的问题。比如. RestTemplate, JmsTemplate,

JpaTemplate。

> 前端控制器—Spring提供了DispatcherServlet来对请求进行分发。

> 视图帮助(View Helper)—Spring提供了一系列的JSP标签，高效宏来辅助将分散的代码整合在视图里。

> 依赖注入—贯穿于BeanFactory / ApplicationContext接口的核心理念。

> 工厂模式—BeanFactory用来创建对象的实例。

什么是 MyBatis?

答：MyBatis 是一个可以自定义 SQL、存储过程和高级映射的持久层框架。

讲下 MyBatis 的缓存答

：MyBatis 的缓存分为一级缓存和二级缓存,一级缓存放在 session 里面,默认就有,二级缓存放在它的命名空间里,默认是不打开的,使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置

Mybatis 是如何进行分页的？分页插件的原理是什么？答：

1）Mybatis 使用 RowBounds 对象进行分页，也可以直接编写 sql 实现分页，也可以使用Mybatis 的分页插件。

2）分页插件的原理：实现 Mybatis 提供的接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql。

举例：select * from student，拦截 sql 后重写为：select t.* from （select * from student）t limit 0，10

简述 Mybatis 的插件运行原理，以及如何编写一个插件？答：

1）Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 通过动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

2）实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

Mybatis 动态 sql 是做什么的？都有哪些动态 sql？能简述一下动态 sql 的执行原理不？答：

1）Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能。

2）Mybatis 提供了 9 种动态 sql 标签：

trim|where|set|foreach|if|choose|when|otherwise|bind。

3）其执行原理为，使用 OGNL 从 sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

#{}和${}的区别是什么？答： 1）#{}是预编译处理，${}是字符串替换。

2）Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法来赋值；

3）Mybatis 在处理${}时，就是把${}替换成变量的值。

4）使用#{}可以有效的防止 SQL 注入，提高系统安全性。

为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里?

答：Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？答：

1） Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。

2） 它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 a.getB().getName()，拦截器 invoke()方法发现 a.getB()是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 a.setB(b)，于是 a 的对象 b 属性就有值了，接着完成 a.getB().getName()方法的调用。这就是延迟加载的基本原理。

MyBatis 与 Hibernate 有哪些不同？答：

1） Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句，不过 mybatis 可以通过 XML 或注解方式灵活配置要运行的 sql 语句，并将 java 对象和 sql 语句映射生成最终执行的 sql，最后将 sql 执行的结果再映射生成 java 对象。

2） Mybatis 学习门槛低，简单易学，程序员直接编写原生态 sql，可严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套 sql 映射文件，工作量大。

3） Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用 hibernate 开发可以节省很多代码，提高效率。但是Hibernate 的缺点是学习门槛高，要精通门槛更高，而且怎么设计 O/R 映射，在性能和对象模型之间如何权衡，以及怎样用好 Hibernate 需要具有很强的经验和能力才行。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

MyBatis 的好处是什么？答：

1） MyBatis 把 sql 语句从 Java 源程序中独立出来，放在单独的 XML 文件中编写，给程序的维护带来了很大便利。

2） MyBatis 封装了底层 JDBC API 的调用细节，并能自动将结果集转换成 Java Bean 对象，大大简化了 Java 数据库编程的重复工作。

3） 因为 MyBatis 需要程序员自己去编写 sql 语句，程序员可以结合数据库自身的特点灵活控制 sql 语句，因此能够实现比 Hibernate 等全自动 orm 框架更高的查询效率，能够完成复杂查询。

简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系?

答：Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在

Xml 映射文件中，标签会被解析为 ParameterMap 对象，其每个子元素会被解析为 ParameterMapping 对象。标签会被解析为 ResultMap 对象，其每个子

元素会被解析为 ResultMapping 对象。每一个

惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。

该策略可以 大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定

时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到 优的平衡效果。

(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该

Redis集群中保存的所有键。)

Redis中同时使用了惰性过期和定期过期两种过期策略。

Redis key的过期时间和永久有效分别怎么设置?

EXPIRE和PERSIST命令。

我们知道通过expire来设置key 的过期时间，那么对过期的数据怎么处理呢?

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

1. 定时去清理过期的缓存；
2. 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

内存相关

MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据

redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

Redis的内存淘汰策略有哪些

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

全局的键空间选择性移除

noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。

allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除 近 少使用的key。（这个是最常用的）

allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。

设置过期时间的键空间选择性移除

volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除 近 少使用的key。

volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。

volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

总结

Redis的内存淘汰策略的选取并不会影响过期的key的处理。内存淘汰策略用于处理内存不足时的需要申请额外空间的数据；过期策略用于处理过期的缓存数据。

Redis主要消耗什么物理资源?

内存。

Redis的内存用完了会发生什么?

如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以配置内存淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

Redis如何做内存优化?

可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。尽可能使用散列表

（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key，而是应该把这个用户的所有信息存储到一张散列表里面线程模型

Redis线程模型

Redis基于Reactor模式开发了网络事件处理器，这个处理器被称为文件事件处

理器（file event handler）。它的组成结构为4部分：多个套接字、IO多路复用程序、文件事件分派器、事件处理器。因为文件事件分派器队列的消费是单线程的，所以Redis才叫单线程模型。

文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。

```
    当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入
```

（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

虽然文件事件处理器以单线程方式运行， 但通过使用 I/O 多路复用程序来监听多个套接字， 文件事件处理器既实现了高性能的网络通信模型， 又可以很好地与 redis 服务器中其他同样以单线程方式运行的模块进行对接， 这保持了

Redis 内部单线程设计的简单性。

什么是事务?

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

Redis事务的概念

Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合。事务支

持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。

总结说：redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。

Redis事务的三个阶段

\1. 事务开始 MULTI

\2. 命令入队

\3. 事务执行 EXEC

事务执行过程中，如果服务端收到有EXEC、DISCARD、WATCH、MULTI之外的请求，将会把请求放入队列中排队

Redis事务相关命令

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的

Redis会将一个事务中的所有命令序列化，然后按顺序执行。

\1. redis 不支持回滚，“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的内部可以保持简单且快速。

\2. 如果在一个事务中的命令出现错误，那么所有的命令都不会执行；

\3. 如果在一个事务中出现运行错误，那么正确的命令会被执行。

WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。 可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。

MULTI命令用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。 当操作被打断时，返回空值 nil 。

通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中退出。

```
    UNWATCH命令可以取消watch对所有key的监控。
```

事务管理（ACID）概述

原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

一致性（Consistency）

事务前后数据的完整性必须保持一致。

隔离性（Isolation）

多个事务并发执行时，一个事务的执行不应影响其他事务的执行持久性（Durability）持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响

Redis的事务总是具有ACID中的一致性和隔离性，其他特性是不支持的。当服务器运行在AOF持久化模式下，并且appendfsync选项的值为always时，事务也具有耐久性。

Redis事务支持隔离性吗

Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis 的事务是总是带有隔离性的。

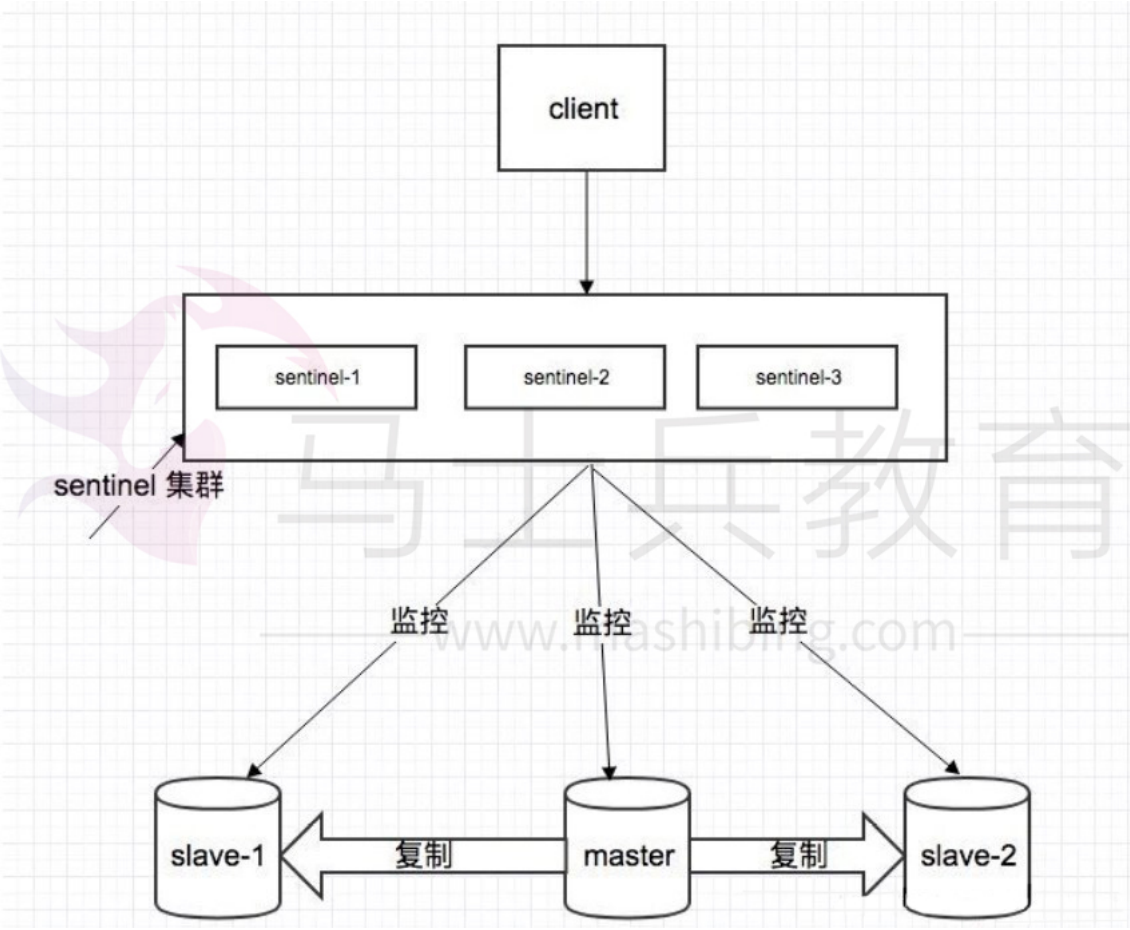
Redis事务保证原子性吗，支持回滚吗

Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

Redis事务其他实现

基于Lua脚本，Redis可以保证脚本内的命令一次性、按顺序地执行，其同时也不提供事务运行错误的回滚，执行过程中如果部分命令运行错误，剩下的命令还是会继续运行完

基于中间标记变量，通过另外的标记变量来标识事务是否执行完成，读取数据时先读取该标记变量判断是否事务执行完成。但这样会需要额外写代码实现，比较繁琐集群方案哨兵模式



哨兵的介绍

sentinel，中文名是哨兵。哨兵是 redis 集群机构中非常重要的一个组件，主要有以下功能：

集群监控：负责监控 redis master 和 slave 进程是否正常工作。

消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。

故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。

配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题。

即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了。

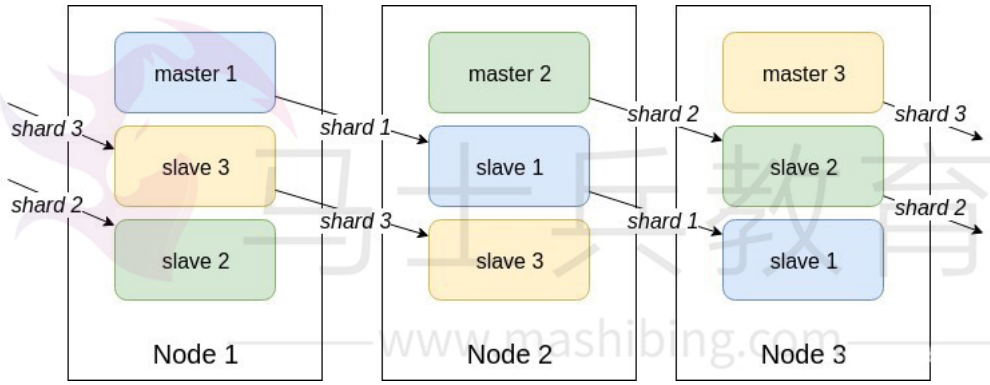
哨兵的核心知识

哨兵至少需要 3 个实例，来保证自己的健壮性。

哨兵 + redis 主从的部署架构，是不保证数据零丢失的，只能保证 redis 集群的高可用性。

对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

官方Redis Cluster 方案(服务端路由查询)



redis 集群模式的工作原理能说一下么？在集群模式下，redis 的 key 是如何寻址的？分布式寻址都有哪些算法？了解一致性 hash 算法吗？简介

Redis Cluster是一种服务端Sharding技术，3.0版本开始正式提供。Redis

Cluster并没有使用一致性hash，而是采用slot(槽)的概念，一共分成16384个

槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行

方案说明

1. 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值) 区间的数据，默认分配了16384 个槽位
2. 每份数据分片会存储在多个互为主从的多节点上
3. 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)
4. 同一分片多个节点间的数据不保持一致性
5. 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点
6. 扩容时时需要需要把旧节点的数据迁移一部分到新节点

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加1w 的端口号，比如 16379。

16379 端口号是用来进行节点间通信的，也就是 cluster bus 的东西，cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。cluster bus 用了另外一种二进制的协议，gossip 协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。

节点间的内部通信机制

基本通信原理

集群元数据的维护有两种方式：集中式、Gossip 协议。redis cluster 节点间采用 gossip 协议进行通信。

分布式寻址算法

hash 算法（大量缓存重建）

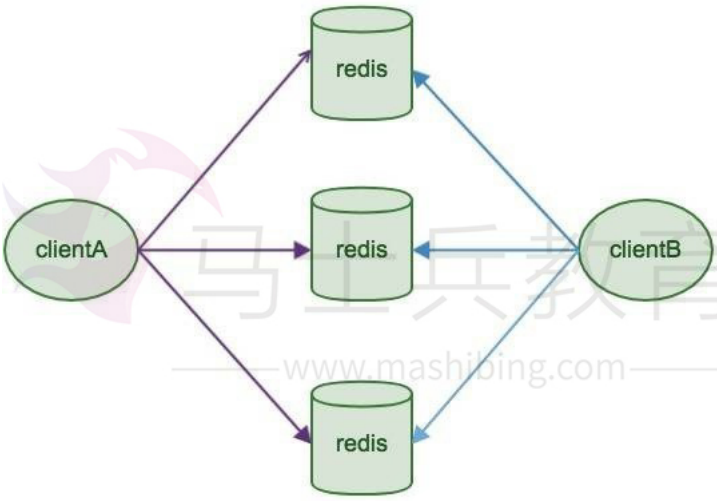
一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡） redis cluster 的 hash slot 算法优点

无中心架构，支持动态扩容，对业务透明

具备Sentinel的监控和自动Failover(故障转移)能力

客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可高性能，客户端直连redis运维也很复杂，数据迁移需要人工干预只能使用0号数据库

不支持批量操作(pipeline管道操作) 分布式逻辑和存储模块耦合等基于客户端分配



简介

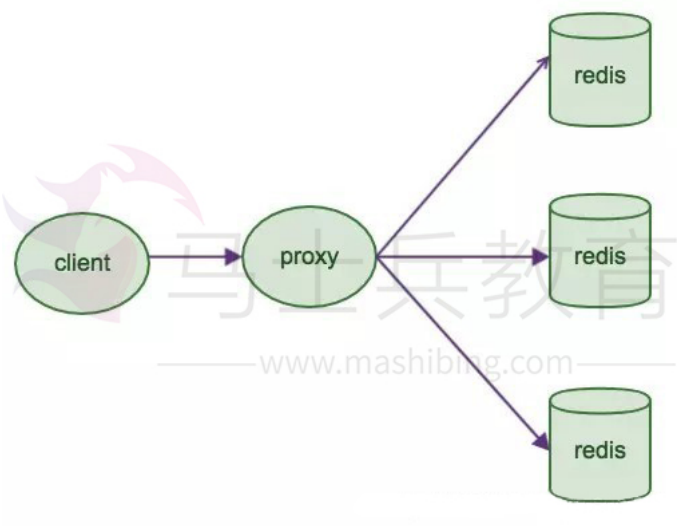
Redis Sharding是Redis Cluster出来之前，业界普遍使用的多Redis实例集群方法。其主要思想是采用哈希算法将Redis数据的key进行散列，通过hash函数，特定的key会映射到特定的Redis节点上。Java redis客户端驱动jedis，支持

Redis Sharding功能，即ShardedJedis以及结合缓存池的ShardedJedisPool 优点

优势在于非常简单，服务端的Redis实例彼此独立，相互无关联，每个Redis实例像单服务器一样运行，非常容易线性扩展，系统的灵活性很强缺点

由于sharding处理放到客户端，规模进一步扩大时给运维带来挑战。

客户端sharding不支持动态增删节点。服务端Redis实例群拓扑结构有变化时，每个客户端都需要更新调整。连接不能共享，当应用规模增大时，资源浪费制约优化基于代理服务器分片



简介客户端发送请求到一个代理组件，代理解析客户端的数据，并将请求转发至正确的节点， 后将结果回复给客户端特征

透明接入，业务程序不用关心后端Redis实例，切换成本低

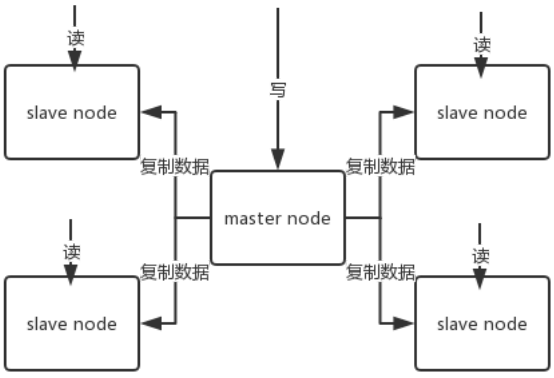
Proxy 的逻辑和存储的逻辑是隔离的代理层多了一次转发，性能有所损耗业界开源方案

Twtter开源的Twemproxy 豌豆荚开源的Codis

Redis 主从架构

单机的 redis，能够承载的 QPS 大概就在上万到几万不等。对于缓存来说，一般都是用来支撑读高并发的。因此架构做成主从(master-slave)架构，一主多

从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的读请求全部走从节点。这样也可以很轻松实现水平扩容，支撑读高



redis replication -> 主从架构 -> 读写分离 -(redismasterslave)> 水平扩容支撑读高并发 redis replication 的核心机制

redis 采用异步方式复制数据到 slave 节点，不过 redis2.8 开始，slave node 会周期性地确认自己每次复制的数据量；

一个 master node 是可以配置多个 slave node 的； slave node 也可以连接其他的 slave node；

slave node 做复制的时候，不会 block master node 的正常工作；

slave node 在做复制的时候，也不会 block 对自己的查询操作，它会用旧的数据集来提供服务；但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了；

slave node 主要用来进行横向扩容，做读写分离，扩容的 slave node 可以提高读的吞吐量。

注意，如果采用了主从架构，那么建议必须开启 master node 的持久化，不建议用 slave node 作为 master node 的数据热备，因为那样的话，如果你关掉 master 的持久化，可能在 master 宕机重启的时候数据是空的，然后可能一经过复制， slave node 的数据也丢了。

另外，master 的各种备份方案，也需要做。万一本地的所有文件丢失了，从备份中挑选一份 rdb 去恢复 master，这样才能确保启动的时候，是有数据的，即使采用了后续讲解的高可用机制，slave node 可以自动接管 master node，但也可能 sentinel 还没检测到 master failure，master node 就自动重启了，还是可能导致上面所有的 slave node 数据被清空。

redis 主从复制的核心原理当启动一个 slave node 的时候，它会发送一个 PSYNC 命令给 master node。

如果这是 slave node 初次连接到 master node，那么会触发一次 full

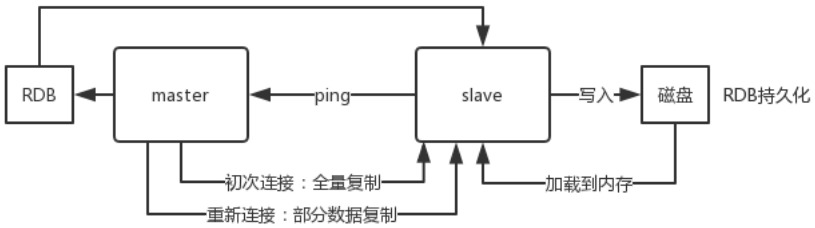
resynchronization 全量复制。此时 master 会启动一个后台线程，开始生成一份

RDB 快照文件，

同时还会将从客户端 client 新收到的所有写命令缓存在内存中。RDB 文件生成完毕后， master 会将这个 RDB 发送给 slave，slave 会先写入本地磁盘，然后再从本地磁盘加载到内存中，

接着 master 会将内存中缓存的写命令发送到 slave，slave 也会同步这些数据。

slave node 如果跟 master node 有网络故障，断开了连接，会自动重连，连接之后 master node 仅会复制给 slave 部分缺少的数据。



1. 当从库和主库建立MS关系后，会向主数据库发送SYNC命令
2. 主库接收到SYNC命令后会开始在后台保存快照(RDB持久化过程)，并将期间接收到的写命令存起来
3. 当快照完成后，主Redis会将快照文件和所有缓存的写命令发送给从

Redis

4. 从Redis接收到后，会载入快照文件并且执行收到的缓存的命令
5. 之后，主Redis每当接收到写命令时就会将命令发送从Redis，从而保证数据的一致

缺点

所有的slave节点数据的复制和同步都由master节点来处理，会照成master节点压力太大，使用主从从结构来解决

Redis集群的主从复制模型是怎样的?

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型，每个节点都会有N-1个复制品生产环境中的 redis 是怎么部署的?

redis cluster，10 台机器，5 台机器部署了 redis 主实例，另外 5 台机器部署

了 redis 的从实例，每个主实例挂了一个从实例，5 个节点对外提供读写服务，每个节点的读写高峰qps可能可以达到每秒 5 万，5 台机器 多是 25 万读写请求/s。

机器是什么配置？32G 内存+ 8 核 CPU + 1T 磁盘，但是分配给 redis 进程的

是10g内存，一般线上生产环境，redis 的内存尽量不要超过 10g，超过 10g 可能会有问题。

5 台机器对外提供读写，一共有 50g 内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis 从实例会自动变成主实例继续提供读写服务。

你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是 10kb。100 条数据是 1mb，10 万条数据是 1g。常驻内存的是 200 万条商品数据，占用内存是 20g，仅仅不到总内存的 50%。目前高峰期每秒就是 3500 左右的请求量。

其实大型的公司，会有基础架构的 team 负责缓存集群的运维。

说说Redis哈希槽的概念?

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384 个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

Redis集群会有写操作丢失吗？为什么?

Redis并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

Redis集群之间是如何复制的?

异步复制

Redis集群最大节点个数是多少?

16384个

Redis集群如何选择数据库?

Redis集群目前无法做数据库选择，默认在0数据库。

分区

Redis是单线程的，如何提高多核CPU的利用率?

可以在同一个服务器部署多个Redis的实例，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的， 所以，如果你想使用多个

CPU，你可以考虑一下分片（shard）。

为什么要做Redis分区?

分区可以让Redis管理更大的内存，Redis将可以使用所有机器的内存。如果没有分区，你 多只能使用一台机器的内存。分区使Redis的计算能力通过简单地增加计算机得到成倍提升，Redis的网络带宽也会随着计算机和网卡的增加而成倍增长。

你知道有哪些Redis分区实现方案?

客户端分区就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个 redis节点读取。大多数客户端已经实现了客户端分区。

代理分区 意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的响应结果返回给客户端。redis和memcached的一种代理实现就是Twemproxy

查询路由(Query routing) 的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。Redis Cluster实现了一种混合形式的查询路由，但并不是直接将请求从一个redis节点转发到另一个redis节点，而是在客户端的帮助下直接redirected到正确的redis节点。

Redis分区有什么缺点?

涉及多个key的操作通常不会被支持。例如你不能对两个集合求交集，因为他们可能被存储到不同的Redis实例（实际上这种情况也有办法，但是不能直接使用交集指令）。

同时操作多个key,则不能使用Redis事务.

分区使用的粒度是key，不能使用一个非常长的排序key存储一个数据集（The

partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set）

当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的Redis 实例和主机同时收集RDB / AOF文件。

分区时动态扩容或缩容可能非常复杂。Redis集群在运行时增加或者删除Redis 节点，能做到 大程度对用户透明地数据再平衡，但其他一些客户端分区或者代理分区方法则不支持这种特性。然而，有一种预分片的技术也可以较好的解决这个问题。

分布式问题

Redis实现分布式锁

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户

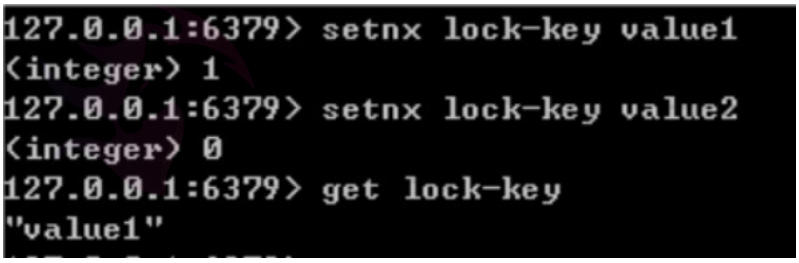
端对Redis的连接并不存在竞争关系Redis中可以使用SETNX命令实现分布式锁。

当且仅当 key 不存在，将 key 的值设为 value。 若给定的 key 已经存在，则

SETNX 不做任何动作

SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。

返回值：设置成功，返回 1 。设置失败，返回 0 。



使用SETNX完成同步锁的流程及事项如下(img)：使用SETNX命令获取锁，若返回0（key已存在，锁已存在）则获取失败，反之获取成功为了防止获取锁后程序出现异常，导致其他线程/进程调用SETNX命令总是返回

0而进入死锁状态，需要为该key设置一个“合理”的过期时间释放锁，使用DEL命令将锁数据删除

如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是 后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同!

推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）基于zookeeperl临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号 小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推Zookeeper。

分布式Redis是前期做还是后期规模上来了再做好？为什么?

既然Redis是如此的轻量（单实例只使用1M内存），为防止以后的扩容， 好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让Redis以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。

一开始就多设置几个Redis实例，例如32或者64个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。

这样的话，当你的数据不断增长，需要更多的Redis服务器时，你需要做的就是仅仅将Redis实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的Redis实例从第一台机器迁移到第二台机器。

什么是 RedLock

Redis 官方站提出了一种权威的基于 Redis 实现分布式锁的方式名叫

Redlock，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁： 终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务缓存异常缓存雪崩

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案

1. 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

2. 一般并发量不是特别多的时候，使用 多的解决方案是加锁排队。
3. 给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。

缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案

1. 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
2. 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-

value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

3. 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力附加

对于空间的利用到达了一种极致，那就是Bitmap和布隆过滤器(Bloom Filter)。

Bitmap： 典型的就是哈希表缺点是，Bitmap对于每个元素只能记录1bit信息，如果还想完成额外的功能，恐怕只能靠牺牲更多的空间、时间来完成了。

布隆过滤器（推荐）

就是引入了k(k>1)k(k>1)个相互独立的哈希函数，保证在给定的空间、误判率下，完成元素判重的过程。

它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

Bloom-Filter算法的核心思想就是利用多个不同的Hash函数来解决“冲突”。

Hash存在一个冲突（碰撞）的问题，用同一个Hash得到的两个URL的值有可能相同。为了减少冲突，我们可以多引入几个Hash，如果通过其中的一个Hash值我们得出某元素不在集合中，那么该元素肯定不在集合中。只有在所有的Hash

函数告诉我们该元素在集合中时，才能确定该元素存在于集合中。这便是

Bloom-Filter的基本思想。

Bloom-Filter一般用于在大数据量的集合中判定某元素是否存在。

缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案

1.设置热点数据永远不过期。

2.加互斥锁，互斥锁缓存预热

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！解决方案

1. 直接写个缓存刷新页面，上线时手工操作一下；
2. 数据量不大，可以在项目启动的时候自动进行加载；
3. 定时刷新缓存；

缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

缓存降级的 终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

1. 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
2. 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；
3. 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的 大阀值，此时可以根据情况自动降级或者人工降级；
4. 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

热点数据和冷数据

热点数据，缓存才有价值对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。频繁修改的数据，看情况考虑使用缓存对于热点数据，比如我们的某IM产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次。再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次。

数据更新前至少读取两次，缓存才有意义。这个是 基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。

那存不存在，修改频率很高，但是又不得不考虑缓存的场景呢？有！比如，这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享

数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到

Redis缓存，减少数据库压力。

缓存热点key

缓存中的一个Key(比如一个促销商品)，在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

解决方案对缓存查询加锁，如果KEY不存在，就加锁，然后查DB入缓存，然后解锁；其

他进程如果发现有锁就等待，然后等解锁后返回数据或者进入DB查询常用工具

Redis支持的Java客户端都有哪些？官方推荐用哪个？

Redisson、Jedis、lettuce等等，官方推荐使用Redisson。

Redis和Redisson有什么关系？

Redisson是一个高级的分布式协调Redis客服端，能帮助用户在分布式环境中轻松实现一些Java的对象 (Bloom filter, BitSet, Set, SetMultimap,

ScoredSortedSet, SortedSet, Map, ConcurrentMap, List, ListMultimap,

Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock,

ReadWriteLock, AtomicLong, CountDownLatch, Publish / Subscribe,

HyperLogLog)。

Jedis与Redisson对比有什么优缺点？

Jedis是Redis的Java实现的客户端，其API提供了比较全面的Redis命令的支

持；Redisson实现了分布式和可扩展的Java数据结构，和Jedis相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等Redis特性。

Redisson的宗旨是促进使用者对Redis的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

其他问题

Redis与Memcached的区别

两者都是非关系型内存键值数据库，现在公司一般都是用 Redis 来实现缓存，而且 Redis 自身也越来越强大了！Redis 与 Memcached 主要有以下不同：

对比参数	Redis	Memcac hed
类型	1. 支持内存 2. 非关系型数据库	1. 支持内存 2. 键值对形式 3. 缓存形式
数据存储类型	1. String 2. List 3. Set 4. Hash 5. Sort Set 【俗称 ZSet】	1. 文本型 2. 二进制类型
查询【操作】类型	1. 批量操作 2. 事务支持 3. 每个类型不同的 CRUD	1.常用的 CRUD 2. 少量的其他命令
附加功能	1. 发布/订阅模式 2. 主从分区 3. 序列化支持 4. 脚本支持 【Lua脚本】	1. 多线程服务支持
网络IO模型	1. 单线程的多路 IO 复用模型	1. 多线程，非阻塞IO模式
事件库	自封转简易事件库 AeEvent	贵族血统的 LibEvent 事件库
持久化支持	1. RDB 2. AOF	不支持
集群模式	原生支持 cluster 模式，可以实现主从复制，读写分离	没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据
内存管理机制	在 Redis 中，并不是所有数据都一直存储在内存中，可以将一些很久没用 的 value 交换到磁盘	Memcach ed 的数据则会一直在内存中， Memcach ed 将内存分割成特定长度的块来存储数据，以完全解决内存碎片的问题。但是这种方式会使得内存的利用率不高，例如块的大小为 128 bytes，只存储 100 bytes 的数据，那么剩下的 28 bytes 就浪费掉了。
	复杂数据	纯key- value，数据量非常大，并发量非常大的业务
适用场景	结构，有持久化，高可用需求，value 存储内容较大	

(1) memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型

(2) redis的速度比memcached快很多

(3) redis可以持久化其数据

如何保证缓存与数据库双写时的数据一致性？

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题？一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔有不一致的情况， 好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可以保证一定不会出现不一致的情况串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

还有一种方式就是可能会暂时产生不一致的情况，但是发生的几率特别小，就是先更新数据库，然后再删除缓存。

问题场景	描述	解决
先写缓存，再写数据库，缓存写成功，数据库写失败	缓存写成功，但写数据库失败或者响应延迟，则下次读取（并发读）缓存时，就出现脏读	这个写缓存的方式，本身就是错误的，需要改为先写数据库，把旧缓存置为失效；读取数据的时候，如果缓存不存在，则读取数据库再写缓存
先写数据库，再写缓存，数据库写成功，缓存写失败	写数据库成功，但写缓存失败，则下次读取（并发读）缓存时，则读不到数据	缓存使用时，假如读缓存失败，先读数据库，再回写缓存的方式实现
需要缓存异步刷新	指数据库操作和写缓存不在一个操作步骤中，比如在分布式场景下，无法做到同时写缓存或需要异步刷新（补	确定哪些数据适合此类场景，根据经验值确定合理的数据不一致时间，用户数据刷新的时间间隔
	救措施）时候	

Redis常见性能问题和解决方案?

1. Master 好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化。
2. 如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。
3. 为了主从复制的速度和连接的稳定性，Slave和Master 好在同一个局域网内。
4. 尽量避免在压力较大的主库上增加从库
5. Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。
6. 为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<–Slave1<–Slave2<–Slave3...，这样的结构也方便解决单点故障问题，实现Slave对Master的替换，也即，如果

Master挂了，可以立马启用Slave1做Master，其他不变。

Redis官方为什么不提供Windows版本?

因为目前Linux版本已经相当稳定，而且用户量很大，无需开发windows版本，反而会带来兼容性等问题。

一个字符串类型的值能存储最大容量是多少?

512M

Redis如何做大量数据插入？ Redis2.6开始redis-cli支持一种新的被称之为pipe mode的新模式用于执行大量数据插入工作。

假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来?

使用keys指令可以扫出指定模式的key列表。

对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

使用Redis做过异步队列吗，是如何实现的

使用list类型保存数据信息，rpush生产消息，lpop消费消息，当lpop没有消息时，可以sleep一段时间，然后再检查有没有信息，如果不想sleep的话，可以使用blpop, 在没有信息的时候，会一直阻塞，直到信息的到来。redis可以通过 pub/sub主题订阅模式实现一个生产者，多个消费者，当然也存在一定的缺点，当消费者下线时，生产的消息会丢失。

Redis如何实现延时队列

使用sortedset，使用时间戳做score, 消息内容作为key,调用zadd来生产消息，消费者使用zrangbyscore获取n秒之前的数据做轮询处理。

Redis回收进程如何工作的?

1. 一个客户端运行了新的命令，添加了新的数据。
2. Redis检查内存使用情况，如果大于maxmemory的限制， 则根据设定好的策略进行回收。
3. 一个新的命令被执行，等等。
4. 所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。

如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

Redis回收使用的是什么算法?

LRU算法

消息中间件如何保证消息的一致性

(1)主动方应用先把消息发送给消息中间件，消息状态标记为待确认;

(2)消息中间件收到消息之后，把消息持久化到消息存储中，但并不向被动方应用投递消息;

(3)消息中间件返回消息持久化结果（成功，或者失效），主动方应用根据返回结果进行判断如何

处理业务操作处理;

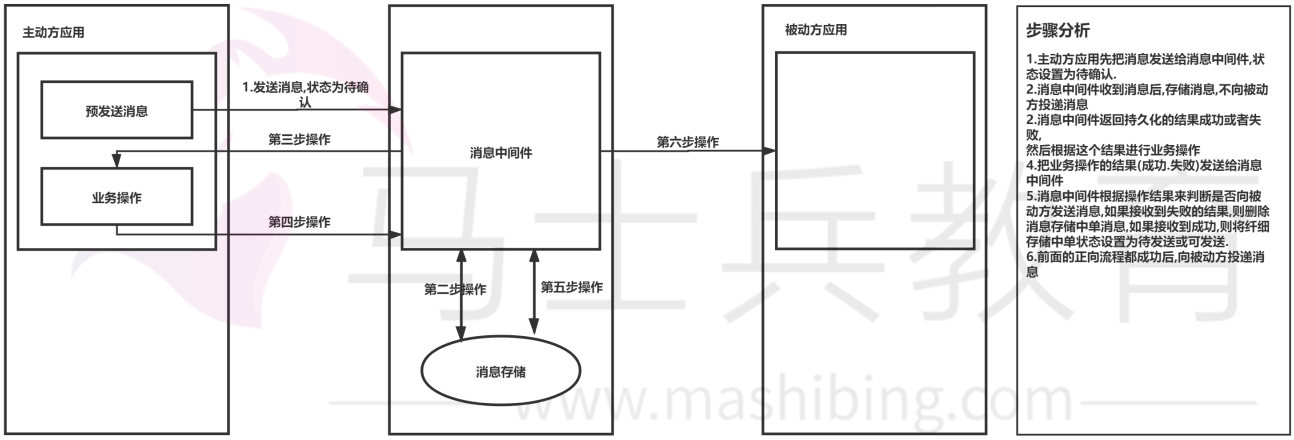
- ①失败：放弃业务操作处理，结束（必须向上层返回失败结果）
- ②成功：执行业务操作处理

(4)业务操作完成后，把业务操作结果（成功/失败）发送给消息中间件;

(5)消息中间件收到业务操作结果后，根据结果进行处理;

- ①失败：删除消息存储中的消息，结束;
- ②成功：更新消息存储中的消息状态为·待发送（可发送）·，紧接着执行消息投递;

(6)前面的正向流程都成功后，向被动方应用投递消息；



如何进行消息的重试机制?

Rocket重试机制，消息模式，刷盘方式

一、Consumer 批量消费（推模式）

可以通过

```
consumer.setConsumeMessageBatchMaxSize(10);//每次拉取10条
```

这里需要分为2种情况

- Consumer端先启动
- Consumer端后启动. 正常情况下：应该是Consumer需要先启动

注意：如果broker采用推模式的话，consumer先启动，会一条一条消息的消费，consumer后启动会才用批量消费

Consumer端先启动

1、Consumer.java

```
package quickstart;
import java.util.List;
import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;
/**
 * Consumer，订阅消息
 */
public class Consumer {
    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_4");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费 ,（消费顺序消息的时候设置）
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        consumer.subscribe("TopicTest", "*");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

                try {
                    System.out.println("msgs的长度" + msgs.size());
                    System.out.println(Thread.currentThread().getName() + " Receive New Messages: " + msgs);
                } catch (Exception e) {
                    e.printStackTrace();
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
```

```
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

由于这里是Consumer先启动，所以他回去轮询MQ上是否有订阅队列的消息，由于每次producer插入一条，Consumer就拿一条所以测试结果如下（每次size都是1）

2、Consumer端后启动，也就是Producer先启动

由于这里是Consumer后启动，所以MQ上也就堆积了一堆数据，Consumer的

```
consumer.setConsumeMessageBatchMaxSize(10);//每次拉取10条
```

二、消息重试机制：消息重试分为2种

- # 1、Producer端重试
- # 2、Consumer端重试

1、Producer端重试

也就是Producer往MQ上发消息没有发送成功，我们可以设置发送失败重试的次数,发送并触发回调函数

```
          //设置重试的次数
            producer.setRetryTimesWhenSendFailed(3);
            //开启生产者
            producer.start();
            //创建一条消息
            Message msg = new Message("PushTopic", "push", "1",   "我是一条普通消息".getBytes());
            //发送消息
            SendResult result = producer.send(msg);
            //发送，并触发回调函数
            producer.send(msg, new SendCallback() {

                @Override
                //成功的回调函数
                public void onSuccess(SendResult sendResult) {
                    System.out.println(sendResult.getSendStatus());
                    System.out.println("成功了");
                }

                @Override
                //出现异常的回调函数
                public void onException(Throwable e) {
                System.out.println("失败了"+e.getMessage());

                }
            });
```

2、Consumer端重试

2.1、exception的情况，一般重复16次 10s、30s、1分钟、2分钟、3分钟等等

上面的代码中消费异常的情况返回

return ConsumeConcurrentlyStatus.RECONSUME_LATER;//重试

正常则返回：

return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;//成功

```
package quickstart;


import java.util.List;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;

/**
 * Consumer，订阅消息
 */
```

```
public class Consumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_4");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "*");

        consumer.registerMessageListener(new MessageListenerConcurrently() {

            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

                try {
                    // System.out.println("msgs的长度" + msgs.size());
                    System.out.println(Thread.currentThread().getName() + " Receive New Messages: " + msgs);
                    for (MessageExt msg : msgs) {
                        String msgbody = new String(msg.getBody(), "utf-8");
                        if (msgbody.equals("Hello RocketMQ 4")) {
                            System.out.println("======错误=======");
                            int a = 1 / 0;
                        }
                    }

                } catch (Exception e) {
                    e.printStackTrace();
                    if(msgs.get(0).getReconsumeTimes()==3){
                        //记录日志

                        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
                    }else{

                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;// 重试
                    }
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
            }
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

假如超过了多少次之后我们可以让他不再重试记录 日志。

if(msgs.get(0).getReconsumeTimes()==3){
//记录日志
return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
}

2.2超时的情况，这种情况MQ会无限制的发送给消费端。

就是由于网络的情况，MQ发送数据之后，Consumer端并没有收到导致超时。也就是消费端没有给我返回return 任何状态，这样的就认为没有到达Consumer端。

这里模拟Producer只发送一条数据。consumer端暂停1分钟并且不发送接收状态给MQ

```
package model;

import java.util.List;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;

/**
 * Consumer，订阅消息
 */
public class Consumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
```

```
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("message_consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "*");

        consumer.registerMessageListener(new MessageListenerConcurrently() {

            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

                try {

                    // 表示业务处理时间
                    System.out.println("=========开始暂停==============");
                    Thread.sleep(60000);

                    for (MessageExt msg : msgs) {
                        System.out.println(" Receive New Messages: " + msg);
                    }

                } catch (Exception e) {
                    e.printStackTrace();
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;// 重试
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
            }
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

三、消费模式

1、集群消费

2、广播消费

rocketMQ默认是集群消费,我们可以通过在Consumer来支持广播消费

```
consumer.setMessageModel(MessageModel.BROADCASTING);// 广播消费
```

```
package model;

import java.util.List;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;
import com.alibaba.rocketmq.common.protocol.heartbeat.MessageModel;

/**
 * Consumer，订阅消息
 */
public class Consumer2 {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("message_consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        consumer.setMessageModel(MessageModel.BROADCASTING);// 广播消费

        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "*");

        consumer.registerMessageListener(new MessageListenerConcurrently() {

            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
```

```
                try {

                    for (MessageExt msg : msgs) {
                        System.out.println(" Receive New Messages: " + msg);
                    }

                } catch (Exception e) {
                    e.printStackTrace();
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;// 重试
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
            }
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

四、conf下的配置文件说明

异步复制和同步双写主要是主和从的关系。消息需要实时消费的，就需要采用主从模式部署

异步复制:比如这里有一主一从，我们发送一条消息到主节点之后，这样消息就算从producer端发送成功了，然后通过异步复制的方法将数据复制到从节点

同步双写:比如这里有一主一从，我们发送一条消息到主节点之后，这样消息就并不算从producer端发送成功了，需要通过同步双写的方法将数据同步到从节点后， 才算数据发送成功。

如果rocketMq才用双master部署，Producer往MQ上写入20条数据 其中Master1中拉取了12条 。Master2中拉取了8 条，这种情况下，Master1宕机，那么我们消费数据的时候，只能消费到Master2中的8条，Master1中的12条默认持久化，不会丢失消息，需要Master1恢复之后这12条数据才能继续被消费，如果想保证消息实时消费，就才用双Master双Slave的模式

五、刷盘方式

同步刷盘：在消息到达MQ后，RocketMQ需要将数据持久化，同步刷盘是指数据到达内存之后，必须刷到commitlog日志之后才算成功，然后返回producer数据已经发送成功。

异步刷盘：，同步刷盘是指数据到达内存之后,返回producer说数据已经发送成功。，然后再写入commitlog日志。

commitlog：

commitlog就是来存储所有的元信息，包含消息体，类似于MySQL、Oracle的redolog,所以主要有CommitLog在，Consume Queue即使数据丢失，仍然可以恢复出来。

consumequeue：记录数据的位置,以便Consume快速通过consumequeue找到commitlog中的数据

Redis数据类型

概述

Redis的键值可以使用物种数据类型：**字符串，散列表，列表，集合，有序集合。**本文详细介绍这五种数据类型的使用方法。本文命令介绍部分只是列举了基本的命令，至于具体的使用示例，可以参考Redis官方文档：http://redis.readthedocs.org/en/latest

字符串类型

字符串是Redis中最基本的数据类型，它能够存储任何类型的字符串，包含二进制数据。可以用于存储邮箱，JSON化的对象，甚至是一张图片，一个字符串允许存储的最大容量为**512MB**。字符串是其他四种类型的基础，与其他几种类型的区别从本质上来说只是组织字符串的方式不同而已。

基本命令

字符串操作

1. **SET** 赋值，用法： `SET key value`
2. **GET** 取值，用法： `GET key`
3. **INCR** 递增数字，仅仅对数字类型的键有用，相当于Java的i++运算，用法： `INCR key`
4. **INCRBY** 增加指定的数字，仅仅对数字类型的键有用，相当于Java的i+=3，用法：`INCRBY key increment`，意思是key自增increment，increment可以为负数，表示减少。
5. **DECR** 递减数字，仅仅对数字类型的键有用，相当于Java的i–，用法：`DECR key`
6. **DECRBY** 减少指定的数字，仅仅对数字类型的键有用，相当于Java的i-=3，用法：`DECRBY key decrement`，意思是key自减decrement，decrement可以为正数，表示增加。
7. **INCRBYFLOAT** 增加指定浮点数，仅仅对数字类型的键有用，用法：`INCRBYFLOAT key increment`
8. **APPEND** 向尾部追加值，相当于Java中的”hello”.append(“ world”)，用法：`APPEND key value`
9. **STRLEN** 获取字符串长度，用法：`STRLEN key`
10. **MSET** 同时设置多个key的值，用法：`MSET key1 value1 [key2 value2 ...]`
11. **MGET** 同时获取多个key的值，用法：`MGET key1 [key2 ...]`

位操作

1. **GETBIT** 获取一个键值的二进制位的指定位置的值(0/1)，用法：`GETBIT key offset`
2. **SETBIT** 设置一个键值的二进制位的指定位置的值(0/1)，用法：`SETBIT key offset value`
3. **BITCOUNT** 获取一个键值的一个范围内的二进制表示的1的个数，用法：`BITCOUNT key [start end]`
4. **BITOP** 该命令可以对多个字符串类型键进行位运算，并将结果存储到指定的键中，BITOP支持的运算包含：**OR,AND,XOR,NOT**，用法：`BITOP OP desKey key1 key2`
5. **BITPOS** 获取指定键的第一个位值为0或者1的位置，用法：`BITPOS key 0/1 [start， end]`

散列类型

散列类型相当于Java中的HashMap，他的值是一个字典，保存很多key，value对，每对key，value的值个键都是字符串类型，换句话说，散列类型不能嵌套其他数据类型。一个散列类型键最多可以包含2的32次方-1个字段。

基本命令

1. **HSET** 赋值，用法：`HSET key field value`
2. **HMSET** 一次赋值多个字段，用法：`HMSET key field1 value1 [field2 values]`
3. **HGET** 取值，用法：`HSET key field`
4. **HMGET** 一次取多个字段的值，用法：`HMSET key field1 [field2]`
5. **HGETALL** 一次取所有字段的值，用法：`HGETALL key`
6. **HEXISTS** 判断字段是否存在，用法：`HEXISTS key field`
7. **HSETNX** 当字段不存在时赋值，用法：`HSETNX key field value`
8. **HINCRBY** 增加数字，仅对数字类型的值有用，用法：`HINCRBY key field increment`
9. **HDEL** 删除字段，用法：`HDEL key field`
10. **HKEYS** 获取所有字段名，用法：`HKEYS key`
11. **HVALS** 获取所有字段值，用法：`HVALS key`
12. **HLEN** 获取字段数量，用法：`HLEN key`

列表类型

列表类型(list)用于存储一个有序的字符串列表，常用的操作是向队列两端添加元素或者获得列表的某一片段。列表内部使用的是双向链表（double linked list）实现的，所以向列表两端添加元素的时间复杂度是O(1),获取越接近列表两端的元素的速度越快。但是缺点是使用列表通过索引访问元素的效率太低（需要从端点开始遍历元素）。所以列表的使用场景一般如：朋友圈新鲜事，只关心最新的一些内容。借助列表类型，Redis还可以作为消息队列使用。

基本命令

1. **LPUSH** 向列表左端添加元素，用法：`LPUSH key value`
2. **RPUSH** 向列表右端添加元素，用法：`RPUSH key value`
3. **LPOP** 从列表左端弹出元素，用法：`LPOP key`
4. **RPOP** 从列表右端弹出元素，用法：`RPOP key`
5. **LLEN** 获取列表中元素个数，用法：`LLEN key`
6. **LRANGE** 获取列表中某一片段的元素，用法：`LRANGE key start stop`，index从0开始，-1表示最后一个元素
7. **LREM** 删除列表中指定的值，用法：`LREM key count value`，删除列表中前count个值为value的元素，当count>0时从左边开始数，count<0时从右边开始数，count=0时会删除所有值为value的元素
8. **LINDEX** 获取指定索引的元素值，用法：`LINDEX key index`
9. **LSET** 设置指定索引的元素值，用法：`LSET key index value`
10. **LTRIM** 只保留列表指定片段，用法：`LTRIM key start stop`，包含start和stop
11. **LINSERT** 像列表中插入元素，用法：`LINSERT key BEFORE|AFTER privot value`，从左边开始寻找值为privot的第一个元素，然后根据第二个参数是BEFORE还是AFTER决定在该元素的前面还是后面插入value
12. **RPOPLPUSH** 将元素从一个列表转义到另一个列表，用法：`RPOPLPUSH source destination`

集合类型

集合在概念在高中课本就学过，集合中每个元素都是不同的，集合中的元素个数最多为2的32次方-1个，集合中的元素师没有顺序的。

基本命令

1. **SADD** 添加元素，用法：`SADD key value1 [value2 value3 ...]`
2. **SREM** 删除元素，用法：`SREM key value2 [value2 value3 ...]`
3. **SMEMBERS** 获得集合中所有元素，用法：`SMEMBERS key`
4. **SISMEMBER** 判断元素是否在集合中，用法：`SISMEMBER key value`
5. **SDIFF** 对集合做差集运算，用法：`SDIFF key1 key2 [key3 ...]`，先计算key1和key2的差集，然后再用结果与key3做差集
6. **SINTER** 对集合做交集运算，用法：`SINTER key1 key2 [key3 ...]`
7. **SUNION** 对集合做并集运算，用法：`SUNION key1 key2 [key3 ...]`
8. **SCARD** 获得集合中元素的个数，用法：`SCARD key`
9. **SDIFFSTORE** 对集合做差集并将结果存储，用法：`SDIFFSTORE destination key1 key2 [key3 ...]`
10. **SINTERSTORE** 对集合做交集运算并将结果存储，用法：`SINTERSTORE destination key1 key2 [key3 ...]`
11. **SUNIONSTORE** 对集合做并集运算并将结果存储，用法：`SUNIONSTORE destination key1 key2 [key3 ...]`
12. **SRANDMEMBER** 随机获取集合中的元素，用法：`SRANDMEMBER key [count]`，当count>0时，会随机中集合中获取count个不重复的元素，当count<0时，随机中集合中获取|count|和可能重复的元素。
13. **SPOP** 从集合中随机弹出一个元素，用法：`SPOP key`

有序集合类型

有序集合类型与集合类型的区别就是他是有序的。有序集合是在集合的基础上为每一个元素关联一个分数，这就让有序集合不仅支持插入，删除，判断元素是否存在等操作外，还支持获取分数最高/最低的前N个元素。有序集合中的每个元素是不同的，但是分数却可以相同。有序集合使用散列表和跳跃表实现，即使读取位于中间部分的数据也很快，时间复杂度为O(log(N))，有序集合比列表更费内存。

基本命令

1. **ZADD** 添加元素，用法：`ZADD key score1 value1 [score2 value2 score3 value3 ...]`
2. **ZSCORE** 获取元素的分数，用法：`ZSCORE key value`
3. **ZRANGE** 获取排名在某个范围的元素，用法：`ZRANGE key start stop [WITHSCORE]`，按照元素从小到大的顺序排序，从0开始编号，包含start和stop对应的元素，WITHSCORE选项表示是否返回元素分数
4. **ZREVRANGE** 获取排名在某个范围的元素，用法：`ZREVRANGE key start stop [WITHSCORE]`，和上一个命令用法一样，只是这个倒序排序的。
5. **ZRANGEBYSCORE** 获取指定分数范围内的元素，用法：`ZRANGEBYSCORE key min max`，包含min和max，`(min` 表示不包含min，`(max` 表示不包含max，`+inf` 表示无穷大
6. **ZINCRBY** 增加某个元素的分数，用法：`ZINCRBY key increment value`
7. **ZCARD** 获取集合中元素的个数，用法：`ZCARD key`
8. **ZCOUNT** 获取指定分数范围内的元素个数，用法：`ZCOUNT key min max`，min和max的用法和5中的一样
9. **ZREM** 删除一个或多个元素，用法：`ZREM key value1 [value2 ...]`
10. **ZREMRANGEBYRANK** 按照排名范围删除元素，用法：`ZREMRANGEBYRANK key start stop`
11. **ZREMRANGEBYSCORE** 按照分数范围删除元素，用法：`ZREMRANGEBYSCORE key min max`，min和max的用法和4中的一样
12. **ZRANK** 获取正序排序的元素的排名，用法：`ZRANK key value`
13. **ZREVRANK** 获取逆序排序的元素的排名，用法：`ZREVRANK key value`
14. **ZINTERSTORE** 计算有序集合的交集并存储结果，用法：`ZINTERSTORE destination numbers key1 key2 [key3 key4 ...] WEIGHTS weight1 weight2 [weight3 weight4 ...] AGGREGATE SUM | MIN | MAX`，numbers表示参加运算的集合个数，weight表示权重，aggregate表示结果取值
15. **ZUNIONSTORE** 计算有序几个的并集并存储结果，用法和14一样，不再赘述。

redis集群如何同步；

Redis 复制与集群

复制

为提高高可用性，排除单点故障，redis支持主从复制功能。
其整体结构是一个有向无环图。

同步方式

分为两种：

- 全同步
  全同步是第一次从机连主机是进行的同步，主机会生成一个RDB文件给从机，然后从机加载该文件。
  并且如果从机掉线时间很长时也会触发这个同步，掉线时间短时使用另外的策略
- 部分同步
  当主机收到修改命令之后会把命令发给从机进行部分同步。
  这里会有一个缓存区，主要是用来，如果有从机掉线，再次连接的时候会优先使用缓存区中的数据进行同步，是在不行才使用全同步
- 同步过程

```
public void sync(String syncCommond){
    if(valid(masterid, offset)){
        // 表示能够通过部分同步找回
        sendCommandCache();
    }else{
        // 进行全同步
        generateRDB();
        transferRDB();
    }
}
```

问题

单主单从的情况下，读写分离很好，但是如果万一主挂了，这样就无法写了
或者单主多从时，如果主挂了，也无法进行同步了。这样就需要选举出一个新的主来作为主机。

主从切换

使用Sentinel，其包含如下功能：

- 监控
  监控服务器节点
- 提醒
  当监控的节点出现问题时，可以通过api通知其他应用等
- 故障转移
  当主挂掉时会选举新的从服务器为主服务器，代替原来主服务器的地位

集群

主从为了提高可用性，防止单点故障。
集群则是为了伸缩性了

key映射到节点的算法

对于集群的情况，经常会涉及一个key存在哪个节点中去。
一般有hash/mod的方式，但是有着增删节点重算的致命问题。
另外还有一致性hash算法，memcache客户端的算法就是这种算法，分成一个2^31的槽圆环，对节点进行hash,对key进行hash,选择顺时针离key最近的节点保存key-value，这样可以最少的影响原数据，还可以具有hash的平衡性等好的优点。
Redis使用的是另外一种：
内置16384 个hash槽，把这些槽大致均匀的分到节点中，每个节点都记录哪些槽给了自己以及给了别人。然后对key算hash/16384, 放到对应的节点中，如果新的节点来了，那么重新分割他一些槽同时更新各个节点中的记录，并且把槽中的记录同时也发给新的节点。

握手

节点信息的结构

```
class Node{
    private long time; // 创建时间
    private String name; // 名称
    private int flag;   // 标示主从，或者在线状态
    private String ip;
    private int port;
    private ClusterState clusterState; // 集群信息
}

class ClusterState{
    private Node myself;
    private int state;
    private int size; // 知道包含一个槽的节点数量
    private Dictionary nodes;  // 集群节点名单
}
```

size为0， 则state为集群下线。
握手就是节点加入到已有集群的一种方式，主要是为了丰富nodeList。握手的过程如下：

- B向A发送CLUSTER MEET
- A为B创建Node结构，并且加到clusterState.nodes中
- A向B发送MEET消息，B再未A创建Node,加入自己
- 然后B向A发送PONG,A向B返回PING完成握手

集群行为

当集群接收到请求之后：

- 一个节点接收到了请求，会检查是否自己的槽，不是则返回MOVED，告诉客户端去哪个槽
- 重新分片是由客户端去做的， 把一个槽的所有键值转到另外的槽
- 如果正在进行转移，客户请求没有命中，则会返回ask消息，让客户端去另外的节点去找
- 集群中如果有主从，那么从节点复制主节点，主几点下线之后，从节点升级代替主节点

redis的数据添加过程是怎样的：哈希槽；

redis集群实现（四） 数据的和槽位的分配

不知道有没有人思考过redis是如何把数据分配到集群中的每一个节点的，可能有人会说，把集群中的每一个节点编号，先放第一个节点，放满了就放第二个节点，以此类推。。如果真的是这样的话，服务器的利用率和性能就太低了，因为先放第一个，其他的服务器节点就闲置下来了，单个节点的压力就会非常的大，其实就相当于退化成为了单机服务器，从而违背了集群发挥每一个节点的性能的初衷。

在redis官方给出的集群方案中，数据的分配是按照槽位来进行分配的，每一个数据的键被哈希函数映射到一个槽位，redis-3.0.0规定一共有16384个槽位，当然这个可以根据用户的喜好进行配置。当用户put或者是get一个数据的时候，首先会查找这个数据对应的槽位是多少，然后查找对应的节点，然后才把数据放入这个节点。这样就做到了把数据均匀的分配到集群中的每一个节点上，从而做到了每一个节点的负载均衡，充分发挥了集群的威力。

在redis中，把一个key-value键值对放入的最简单的方式就是set key value，如下所示：

```
127.0.0.1:7000> set key value
-> Redirected to slot [12539] located at 192.168.39.153:7002
OK
192.168.39.153:7002> get key
"value"
192.168.39.153:7002>
```

可以看出，当我们把key的值设置成为value的时候，客户端被重定向到了另一个节点192.168.39.153:7002，这是因为key对应的槽位是12359，所以我们的key-value就被放到了槽12359对应的节点，192.168.39.153:7002了。接下来，我们来看看redis是怎么把一个key-value键值对映射成槽，然后又如何存放进集群中的。

首先在redis.c文件里定义了客户端命令和函数的对应关系，

```
    struct redisCommand redisCommandTable[] = {
    ------------------------
    {"set",setCommand,-3,"wm",0,NULL,1,1,1,0,0},
    ------------------------
```

可以看出，set命令会执行setCommand函数进行解析，继续进入setCommand函数查看

```
void setCommand(redisClient *c) {
    int j;
    robj *expire = NULL;
    int unit = UNIT_SECONDS;
    int flags = REDIS_SET_NO_FLAGS;

    ------------------------

    // 对value编码
    c->argv[2] = tryObjectEncoding(c->argv[2]);
    //真正执行set命令的地方

    setGenericCommand(c,flags,c->argv[1],c->argv[2],expire,unit,NULL,NULL);
}
```

继续进入setGenericCommand函数

```
void setGenericCommand(redisClient *c, int flags, robj *key, robj *val, robj *expire, int unit, robj *ok_reply, robj *abort_reply) {
    //参数检查和过期时间的检查
    ------------------------
    //在数据库里设置key  value
    setKey(c->db,key,val);
    //设置完成以后的时间通知
    ------------------------
}
```

接着看数据库的setKey函数

```
void setKey(redisDb *db, robj *key, robj *val) {

    // 添加或覆写数据库中的键值对
    if (lookupKeyWrite(db,key) == NULL) {
        dbAdd(db,key,val);
    } else {
        dbOverwrite(db,key,val);
    }
----------------------------------------------------------------------------------------------
```

当没有在数据库中发现key的时候，我们需要执行dbAdd函数把key-value添加到数据库里。

```
void dbAdd(redisDb *db, robj *key, robj *val) {

    // 赋值key的名字
    sds copy = sdsdup(key->ptr);

    // 添加键值对到字典中
    int retval = dictAdd(db->dict, copy, val);

    // 如果键已经存在，那么停止
    redisAssertWithInfo(NULL,key,retval == REDIS_OK);

    // 如果开启了集群模式，就把键保存到槽里面
    if (server.cluster_enabled) slotToKeyAdd(key);
 }
```

继续进入slotToKeyAdd函数

```
//把键key添加到槽里边
void slotToKeyAdd(robj *key) {

    // 通过字符串key计算出键对应的槽
    unsigned int hashslot = keyHashSlot(key->ptr,sdslen(key->ptr));

    // 将槽 slot 作为分数，键作为成员，添加到 slots_to_keys 跳跃表里面
    zslInsert(server.cluster->slots_to_keys,hashslot,key);

    incrRefCount(key);
}
```

keyHashSlot是一个哈希函数，通过key映射到一个0-16384的整数，我们来看一下实现

```
unsigned int keyHashSlot(char *key, int keylen) {

    //start 和end
    int s, e;
```

```
    for (s = 0; s < keylen; s++)
        if (key[s] == '{') break;

    /* 没有发现和{对应的}，就直接哈希整个字符串 */
    if (s == keylen) return crc16(key,keylen) & 0x3FFF;

    /* 如果发现了{，看看是不是又}匹配 */
    for (e = s+1; e < keylen; e++)
        if (key[e] == '}') break;

    /* 如果没有发现}，哈希函数就计算整个字符串. */
    if (e == keylen || e == s+1) return crc16(key,keylen) & 0x3FFF;

    /*如果{}在我们的两边，哈希中间的字符 */
    return crc16(key+s+1,e-s-1) & 0x3FFF;
}
```

计算key字符串对应的映射值，redis采用了crc16函数然后与0x3FFF取低16位的方法。crc16以及md5都是比较常用的根据key均匀的分配的函数，就这样，用户传入的一个key我们就映射到一个槽上，然后经过gossip协议，周期性的和集群中的其他节点交换信息，最终整个集群都会知道key在哪一个槽上。

redis的淘汰策略有哪些；

- volatile-lru -> 根据LRU算法删除带有过期时间的key。
- allkeys-lru -> 根据LRU算法删除任何key。
- volatile-random -> 根据过期设置来随机删除key, 具备过期时间的key。
- allkeys->random -> 无差别随机删, 任何一个key。
- volatile-ttl -> 根据最近过期时间来删除（辅以TTL）, 这是对于有过期时间的key
- noeviction -> 谁也不删，直接在写操作时返回错误。

redis 集群基础

- 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.
- 节点的fail是通过集群中超过半数的master节点检测失效时才生效.
- 客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可
- redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护node<->slot<->key.
- 如果存入一个值，按照redis cluster哈希槽的算法： CRC16('key')384 = 6782。 那么就会把这个key 的存储分配到对应的master上

redis Cluster主从模式

- 如果进群超过半数以上master挂掉，无论是否有slave集群进入fail状态,所以集群中至少应该有奇数个节点，所以至少有三个节点，每个节点至少有一个备份节点,
- redis cluster 为了保证数据的高可用性，加入了主从模式，一个主节点对应一个或多个从节点，主节点提供数据存取，从节点则是从主节点拉取数据备份，当这个主节点挂掉后，就会有这个从节点选取一个来充当主节点，从而保证集群不会挂掉。

Redis为什么这么快?

(1) 绝大部分请求是纯粹的内存操作（非常快速）

(2) 采用单线程,避免了不必要的上下文切换和竞争条件

(3) 非阻塞IO - IO多路复用

redis采用多线程会有哪些问题?

1）单线程的问题

无法发挥多核CPU性能，单进程单线程只能跑满一个CPU核

可以通过在单机开多个Redis实例来完善

可以通过数据分片来增加吞吐量，问题（不支持批量操作、扩缩容复杂等）

2）多线程的问题

多线程处理可能涉及到锁

多线程处理会涉及到线程切换而消耗CPU

阿里云Redis多线程性能提升思路解析

背景

众所周知redis是单进程单线程模型（不完全是单进程单线程，还有若干后端线程主要做刷脏数据，关闭文件描述符等后台清理工作）。redis中负责主要工作的是主线程，主线程的工作包括但不限：接收客户端连接，处理连接读写事件，解析请求，处理命令，处理定时器事件,数据同步等相关工作。单进程单线程只能跑满一个CPU核，在小包场景下，单个redis server的QPS在8~10万级别。如果QPS超过这个级别，单个redis server就无法满足需求。而常用的解决办法就是数据分片，采用多server的分布式架构予以解决。然而数据分片，多redis server方式也存在若干问题：redis server过多，难以管理；分片之后一些在单redis server上使用的命令无法支持；分片无法解决热点读写问题;分片后数据倾斜，数据重分布，数据扩缩容等也比较复杂。由于单进程单线程的局限，我们期望通过多线程的改造以期充分利用SMP多核架构的优势，从而达到提高单redis server吞吐的目的。对redis做多线程化，最容易想到的方案是每个线程既做IO又做命令处理等工作，但由于redis处理的数据结构相对比较复杂，多线程需要锁来保证线程安全性，而锁粒度处理不好性能反而可能会出现下降。

我们的思路是通过增加IO线程，将连接中数据的读写，命令的解析和数据包的回复放到单独的IO线程来处理，而对命令的处理，定时器事件的执行等仍让单一的线程来处理，以此达到提高单redis server吞吐的目的。

单进程单线程的优点和不足

优点

- 因为单进程单线程模型的限制，redis在实现上将耗时的操作分解成多步，多次来执行（例如dict rehash， 过期key删除等操作），尽量避免长时间执行一个操作，从而避免长时间阻塞在一个操作上。单进程单线程代码编写简单，可以减少多进程多线程导致的上下文切换和锁的争抢。

不足

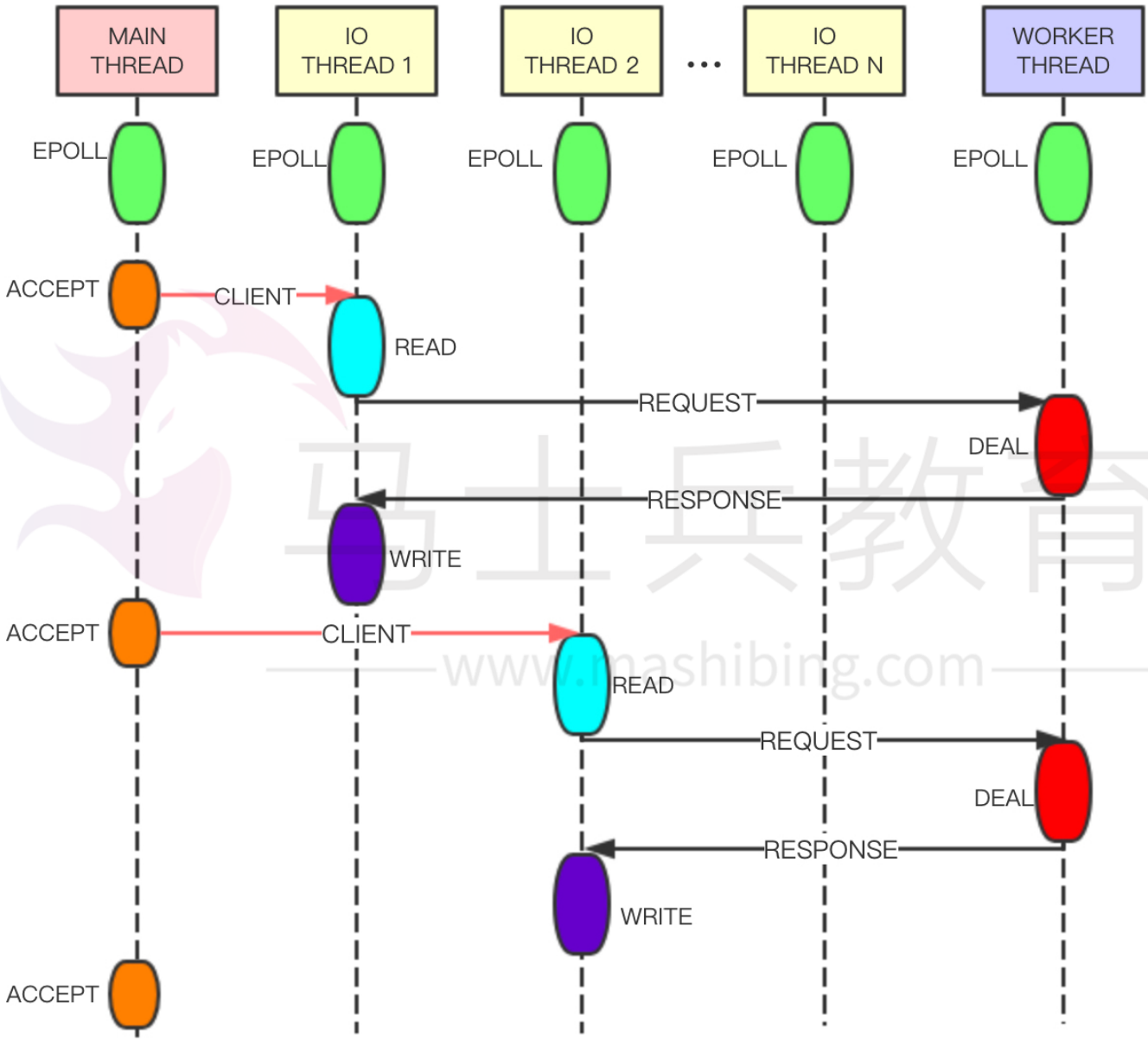
- 只能使用一个CPU核，无法发挥多核优势。
- 对于重IO应用来说，大量的cpu耗费在网络IO操作上。对于将redis做为缓存的应用，往往都是重IO的应用。这类应用基本上都是QPS很高，使用的命令相对比较简单（多为get，set，incr等操作），但是对RT响应很敏感。这类应用通常带宽占用很高，甚至会跑到百兆级别。当前由于万兆，25G网卡的普及，网络往往已不再是瓶颈，而如何发挥多核优势，充分发挥网卡性能成为需要考虑的事情。

实现

线程划分

- 主线程(MAIN THREAD)
- IO线程(IO THREAD)
- WORKER线程(WORKER THREAD)

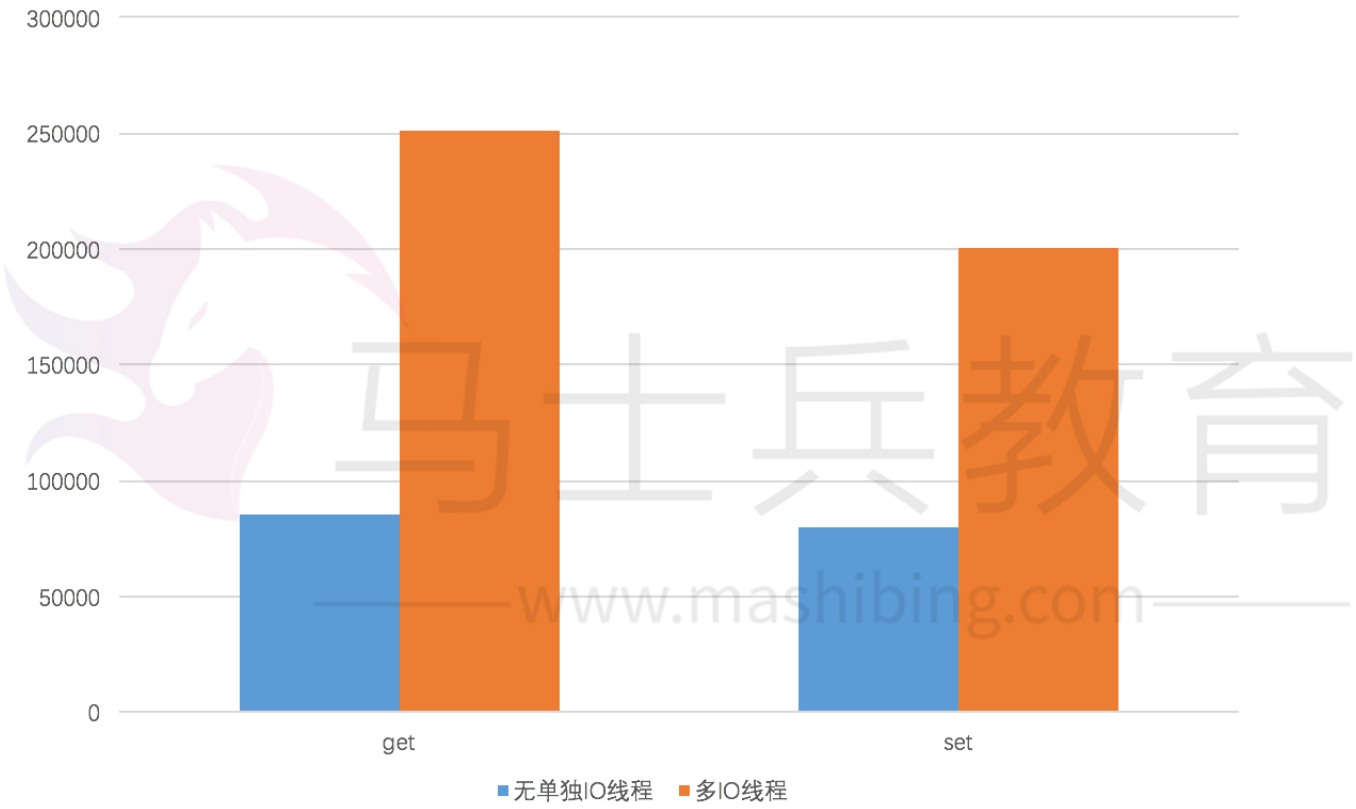
线程模型



- 主线程：接受连接，创建client，将连接转发给IO线程。
- IO线程：处理连接的读写事件，解析命令，将解析的完整命令转发给WORKER线程处理，发送response包，负责删除连接等。
- WORKER线程：负责命令的处理，生成客户端回包，定时器事件的执行等。
- 主线程，IO线程，WORKER线程都有单独的事件驱动。
- 线程之间通过无锁队列交换数据，通过管道进行消息通知。

收益

压测结果



马士兵教育
www.mashibing.com

- 从压测结果来看，小包场景下，读写性能差不多有三倍左右的性能提升。

主从同步速度提升

主从同步优化

- Master向Slave发送同步数据时，数据在IO线程中发送，Slave从主读取数据时，全量数据在WORKER线程中读取，增量数据在IO线程中读取，因此可以相对比较有效的增加同步的速度。

后续工作

- 现在所做的第一部分工作是增加IO线程，优化IO读写能力。进一步的优化可以考虑对WORKER线程进行拆分：每个线程既负责IO读取，也负责WORKER工作处理。

IO线程数设置

- 从测试结果来看，IO线程数最大不要超过6个。超过之后对简单操作来说，WORKER线程往往已经成为瓶颈。
- 进程在启动时需要设置IO线程的个数，在进程运行期间IO线程个数无法修改,按当前的连接分配策略，修改IO线程的个数涉及到连接的重新分配，处理相对比较复杂。

展望

- 随着万兆网卡，25G网卡的普及，如何充分利用硬件的性能需要充分的考虑。多网络IO线程，By pass内核的用户态协议栈等都是可利用的技术。
- 通过IO线程实现数据的迁移，可以无阻塞，IO线程对数据进程Encode，或者命令转发，目标节点实现数据Decode，或者命令执行。

Redis支持哪几种数据结构；

String 、List 、Set 、Hash 、ZSet

Redis跳跃表的问题；

Redis只在两个地方用到了跳跃表，一个是实现有序集合键，另一个是在集群结点中用作内部数 据结构

跳跃表以及跳跃表在redis中的实现

之前在阅读redis源码的时候了解到跳跃表这个数据结构，当时花了点时间了解了下，并做了记录；如今差不多一年过去了，被人问起，竟然一点印象都没有了。然后回头去看自己的注解，重新梳理下。

1 跳跃表的原理

跳跃表在redis中主要是有序表的一种底层实现。对于普通链表的查找，即使有序，我们也不能使用二分法，需要从头开始，一个一个找，时间复杂度为O(n)。而对于跳跃表，从名字可以看出跳跃表的优势就在于可以跳跃。如何做到呢？在于其特殊的层设计。比如我们查找46，普通链表只能从头开始查找，比对-3,2,17...直到46，要比对7次。但是对于跳跃表，我们可以从最高层开始查找：

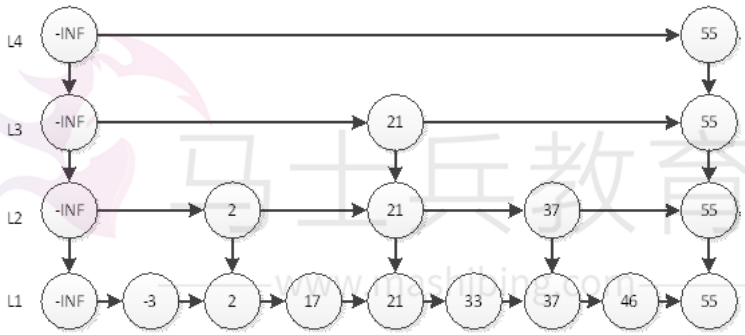
第一步：在L4层直接与55比对，发现大了，退回到第3层

第二步：在L3层与21比对，发现小了，继续往前比对55，发现大了，退回到第二层

第三步：在L2层与37比对，发现小了，往前，与55比对，发现大了，退回到第一层

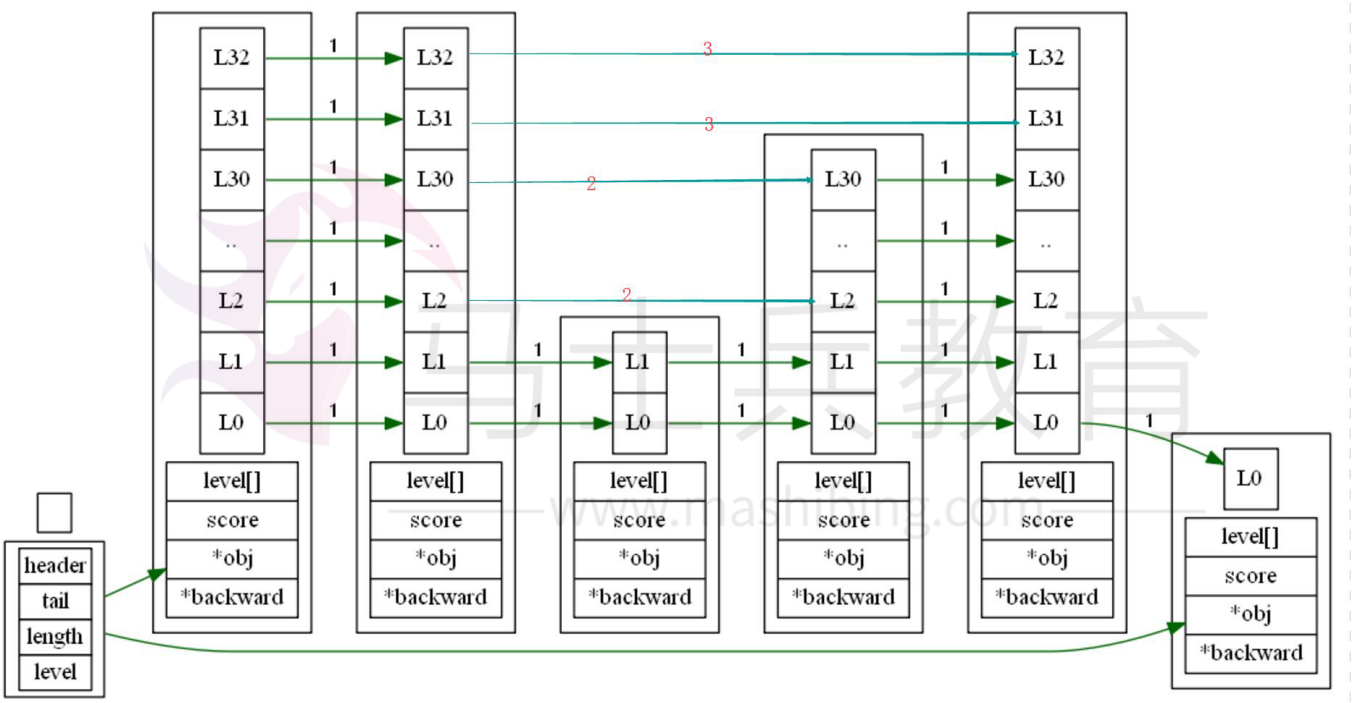
第四步：在第1层，与46比对，查找成功。

共比对了6次，比普通链表只节省了一次，似乎没什么优势。但如果细想，当链表比较长的时候，在高层查找时，跳过的元素数量将相当可观，提速的效果将非常明显。比如如果元素在55之后，在L4层，我们直接就跳过了7个元素，这是非常大的进步。



2 跳跃表在redis中的实现

2.1 跳跃表节点定义和跳跃表描述符定义




```
/* ZSETs use a specialized version of Skiplists */
typedef struct zskiplistNode {
    // member 对象
    robj *obj;
    // 分值
    double score;
    // 后退指针
    struct zskiplistNode *backward;
    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;
        // 节点在该层和前向节点的距离
        unsigned int span;
    } level[];
} zskiplistNode;

typedef struct zskiplist {
    // 头节点，尾节点
    struct zskiplistNode *header, *tail;
```

```
    // 节点数量
    unsigned long length;
    // 目前表内节点的最大层数
    int level;
} zskiplist;
```

2.2 新建跳跃表

主要是初始化描述符对象和初始化头结点。这里需要注意的是，头结点默认层数是为32的，但描述符中指示跳跃表层数初始化为1。

```
/*****************************************************************************
 * 函 数 名  : zslCreate
 * 函数功能  : 创建新的跳跃表
 * 输入参数  : void
 * 输出参数  : 无
 * 返 回 值  : zskiplist
 * 调用关系  :
 * 记     录
 * 1.日    期: 2018年05月08日
 *   作    者:
 *   修改内容: 新生成函数
*****************************************************************************/
zskiplist *zslCreate(void) {
    int j;
    zskiplist *zsl;

    // 申请内存
    zsl = zmalloc(sizeof(*zsl));
    // 初始化跳跃表属性,层数初始化为1，长度初始化为0
    zsl->level = 1;
    zsl->length = 0;

    // 创建一个层数为32，分值为0，成员对象为NULL的表头结点
    zsl->header = zslCreateNode(ZSKIPLIST_MAXLEVEL,0,NULL);
    for (j = 0; j < ZSKIPLIST_MAXLEVEL; j++) {

        // 设定每层的forward指针指向NULL
        zsl->header->level[j].forward = NULL;
        zsl->header->level[j].span = 0;
    }
    // 设定backward指向NULL
    zsl->header->backward = NULL;
    zsl->tail = NULL;
    return zsl;
}
```

2.3 插入新节点

分为三步：

第一步，查找每一层的插入点，所谓插入点指新节点插入后作为新节点前继的节点

redis使用了一个数组来记录

```
zskiplistNode *update[ZSKIPLIST_MAXLEVEL]
```

另外一个数组来记录插入点前继节点排名，所谓排名就是就链表中的位置，这个有什么用呢？主要是用来更新span字段

```
  unsigned int rank[ZSKIPLIST_MAXLEVEL];
```

第二步，新建插入点，层数随机

这一步，要注意的是，如果新节点的层数大于跳跃表的层数，需要更新跳跃表的层数外，还要做：

1 将大于原跳跃表层数的更新节点设置为头结点

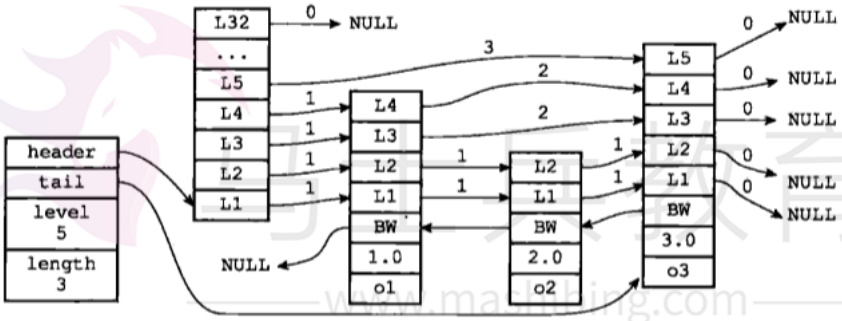
2 将大于原跳跃表层数的更新点前前继节点排名设置为0

3将更新点的前继节点设置为跳跃表长度，为啥是长度？因为整个层除了头结点没有其他节点

```
 update[i]->level[i].span = zsl->length;
```

第三步，修改节点指针和跨度之span

span指示节点与后继节点的距离，如下图，o1.L1 和o2.L1距离为1，o1.L3 和o3.L3距离为2



插入新节点必然涉及到插入处前继和后继 节点指针的改，这个跟普通链表没有什么区别。至于span值的修改，因为在两个节点之间插入新节点，那么原来两点的距离就会产生改变，新节点与后继节点的跨度span也需要设置，这个怎么理解呢?

比如对于上图，要在o2和o3之间插入新节点，

update[0]=o2, rank[0]=2

update[1]=o2, rank[1]=2

update[2]=o1,rank[2]=1

update[3]=o1,rank[3]=1

update[4]=header,rank[1]=0

要设置新节点在L4层与后继节点的距离：

在L4层，在插入新节点之前，update[3]即o1的后继节点为o3，o1.level[3].span=2,插入后，新节点的后继节点为o3,怎么计算新节点在L4层的span呢?

o新.level[3].span 就是**o新**和**o3**的距离

我们知道的是插入新节点前：

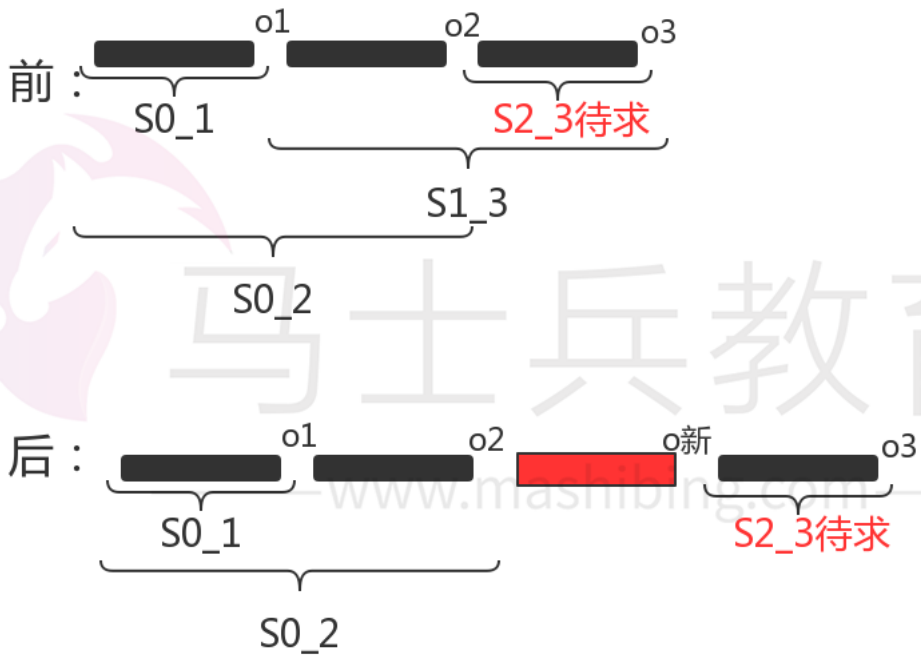
1） **o1**和**o3**的距离（**o1.level[3].span**），

2） **o1**和**o2**的排名（**rank[3]和*****rank[0]*****）

3）间接知道**o2**和**o1**的距离（**rank[0]-rank[3]**）

4）间接知道**o2**和**o3**的距离（**o1.level[3].span-（*rank[0]-rank[3]）***）

看图可知：插入节点后，**o新**和**o3**的距离，实质就是是插入节点前**o2**和**o3**的距离



下面是具体的源码实现：

```
/*****************************************************************************
 * 函 数 名  : zslInsert
 * 函数功能  : 插入新节点
 * 输入参数  : zskiplist *zsl  表头
               double score    节点分数
               robj *obj       待插入节点分数
 * 输出参数  : 无
 * 返 回 值  : zskiplistNode
 * 调用关系  :
```

```
 * 记    录
 * 1.日    期: 2017年06月13日
 *   作    者:
 *   修改内容: 新生成函数
*****************************************************************************/
zskiplistNode *zslInsert(zskiplist *zsl, double score, robj *obj) {
    // updata[]数组记录每一层位于插入节点的前一个节点
    zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;

    // rank[]记录每一层位于插入节点的前一个节点的排名
    //在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位
    unsigned int rank[ZSKIPLIST_MAXLEVEL];

    int i, level;

    serverAssert(!isnan(score));
    // 表头节点
    x = zsl->header;

    // 从最高层开始查找(最高层节点少，跳越快)
    for (i = zsl->level-1; i >= 0; i--) {
        /* store rank that is crossed to reach the insert position */

        //rank[i]用来记录第i层达到插入位置的所跨越的节点总数,也就是该层最接近(小于)给定score的排名
        //rank[i]初始化为上一层所跨越的节点总数,因为上一层已经加过
        rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];

        //后继节点不为空，并且后继节点的score比给定的score小
        while (x->level[i].forward &&
            (x->level[i].forward->score < score ||
                //score相同，但节点的obj比给定的obj小
                (x->level[i].forward->score == score &&
                compareStringObjects(x->level[i].forward->obj,obj) < 0))) {
            //记录跨越了多少个节点
            rank[i] += x->level[i].span;

            //查找下一个节点
            x = x->level[i].forward;
        }
        // 存储当前层上位于插入节点的前一个节点,找下一层的插入节点
        update[i] = x;
    }

    /* we assume the key is not already inside, since we allow duplicated
     * scores, and the re-insertion of score and redis object should never
     * happen since the caller of zslInsert() should test in the hash table
     * if the element is already inside or not. */

    // 此处假设插入节点的成员对象不存在于当前跳跃表内，即不存在重复的节点
    // 随机生成一个level值
    level = zslRandomLevel();



    // 如果level大于当前存储的最大level值
    // 设定rank数组中大于原level层以上的值为0--为什么设置为0
    // 同时设定update数组大于原level层以上的数据
    if (level > zsl->level) {

        for (i = zsl->level; i < level; i++) {

            //因为这一层没有节点，所以重置rank[i]为0
            rank[i] = 0;
            //因为这一层还没有节点，所以节点的前一个节点都是头节点
            update[i] = zsl->header;

            //在未添加新节点之前，需要更新的节点跨越的节点数目自然就是zsl->length---因为整个层只有一个头结点----->言外之意头结点的span都是链表长度
            update[i]->level[i].span = zsl->length;
        }
        // 更新level值（max层数）
        zsl->level = level;
    }

    // 创建插入节点
    x = zslCreateNode(level,score,obj);
    for (i = 0; i < level; i++) {

        // 针对跳跃表的每一层，改变其forward指针的指向
        x->level[i].forward = update[i]->level[i].forward;

         //插入位置节点的后继就是新节点
        update[i]->level[i].forward = x;

        /* update span covered by update[i] as x is inserted here */

        //rank[i]: 在第i层，update[i]->score的排名
        //rank[0] - rank[i]: update[0]->score与update[i]->score之间间隔了几个数
```

```
        // A3 ------------------------------ [I] -> F3
        // A2 ----------------> D2 -------- [I] -> F2
        // A1 ---------> C1 --> D1 -------- [I] -> F1
        // A0 --> B0 --> C0 --> D0 --> E0 - [I] -> F0

        //x->level[i].span = 从x到update[i]->forword的span数目，
        //原来的update[i]->level[i].span = 从update[i]到update[i]->level[i]->forward的span数目
        //所以x->level[i].span = 原来的update[i]->level[i].span - (rank[0] - rank[i]);
        x->level[i].span = update[i]->level[i].span - (rank[0] - rank[i]);


        //对于update[i]->level[i].span值的更新由于在update[i]与update[i]->level[i]->forward之间又添加了x，
        //update[i]->level[i].span = 从update[i]到x的span数目，
        //由于update[0]后面肯定是新添加的x，所以自然新的update[i]->level[i].span = (rank[0] - rank[i]) + 1;

        //提示： update[i]和x[i]之间肯定没有节点了
        update[i]->level[i].span = (rank[0] - rank[i]) + 1;
    }


    //
    //另外需要注意当level > zsl->level时，update[i] = zsl->header的span处理

    /* increment span for untouched levels */

    // 更新高层的span值
    for (i = level; i < zsl->level; i++) {
        //因为下层中间插入了x，而高层没有，所以多了一个跨度
        update[i]->level[i].span++;
    }

    // 设定插入节点的backward指针
    //如果插入节点的前一个节点都是头节点，则插入节点的后向指针为NULL?
    x->backward = (update[0] == zsl->header) ? NULL : update[0];

    //如果插入节点的0层存前向节点则前向节点的后向指针为插入节点
    if (x->level[0].forward)
        x->level[0].forward->backward = x;
    else
    //否则该节点为跳跃表的尾节点
        zsl->tail = x;

    // 跳跃表长度+1
    zsl->length++;

    //返回插入的节点
    return x;
}
```

2.4 删除指定节点

删除节点相对简单一些，提供了根据排名删除节点和根据分数删除节点两个API，主要涉及如下步骤：

1）找到待删除节点在每一层的前继节点，存在updatte数组中

2）调用**zslDeleteNode**处理因为删除节点而引发的指针修改和span修改，以及跳跃表层数和长度修改

3）释放待删除节点

下面是源码的实现：

```
/* Delete all the elements with rank between start and end from the skiplist.
 * Start and end are inclusive. Note that start and end need to be 1-based */
/*****************************************************************************
 * 函 数 名  : zslDeleteRangeByRank
 * 函数功能  : 根据提供的排名起始和结尾删除节点
 * 输入参数  : zskiplist *zsl      跳跃表指针
               unsigned int start  排名起始
               unsigned int end    排名结尾
               dict *dict          ?
 * 输出参数  : 无
 * 返 回 值  : removed
 * 调用关系  :
 * 记    录
 * 1.日    期: 2017年06月29日
 *   作    者: zyz
 *   修改内容: 新生成函数
*****************************************************************************/
unsigned long zslDeleteRangeByRank(zskiplist *zsl, unsigned int start, unsigned int end, dict *dict) {
    zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;
    unsigned long traversed = 0, removed = 0;
    int i;

    x = zsl->header;
    //寻找待更新的节点
    for (i = zsl->level-1; i >= 0; i--) {
```

```
        //指针前移的必要条件是前继指针不为空
        while (x->level[i].forward && (traversed + x->level[i].span) < start) {
            //排名累加
            traversed += x->level[i].span;
            x = x->level[i].forward;
        }
        update[i] = x;
    }

    //下面的节点排名肯定大于等于start
    traversed++;
    x = x->level[0].forward;
    while (x && traversed <= end) {
        //逐个删除后继节点,直到end为止
        zskiplistNode *next = x->level[0].forward;
        zslDeleteNode(zsl,x,update);
        dictDelete(dict,x->obj);
        zslFreeNode(x);
        removed++;
        //每删除一个节点,排名加1
        traversed++;
        x = next;
    }
    return removed;
}

/* Internal function used by zslDelete, zslDeleteByScore and zslDeleteByRank */
/*****************************************************************************
 * 函 数 名  : zslDeleteNode
 * 函数功能  : 内置功能函数，被zslDelete等函数调用
 * 输入参数  : zskiplist *zsl          链表头指针
               zskiplistNode *x        待删除节点指针
               zskiplistNode **update  带删除节点的前一节点地址的指针
 * 输出参数  : 无
 * 返 回 值  :
 * 调用关系  :
 * 记    录
 * 1.日    期: 2017年06月24日
 *   作    者:
 *   修改内容: 新生成函数
*****************************************************************************/

void zslDeleteNode(zskiplist *zsl, zskiplistNode *x, zskiplistNode **update) {
    int i;
    for (i = 0; i < zsl->level; i++) {
        //如果待更新节点（待删除节点的前一节点）的后继节点是待删除节点，则需要处理待更新节点的后继指针
        if (update[i]->level[i].forward == x) {
            update[i]->level[i].span += x->level[i].span - 1;

            //这里有可能为NULL，比如删除最后一个节点
            update[i]->level[i].forward = x->level[i].forward;


        //待删除节点没有出现在此层--跨度减1即可
        } else {
            update[i]->level[i].span -= 1;
        }
    }
    //处理待删除节点的后一节点（如果存在的话）
    if (x->level[0].forward) {
        x->level[0].forward->backward = x->backward;
    } else {
        zsl->tail = x->backward;
    }
    //跳跃表的层数处理，如果表头层级的前向指针为空，说明这一层已经没有元素，层数要减一
    while(zsl->level > 1 && zsl->header->level[zsl->level-1].forward == NULL)
        zsl->level--;

    //跳跃表长度减一
    zsl->length--;
}

/*****************************************************************************
 * 函 数 名  : zslDelete
 * 函数功能  : 根据给定分值和成员来删除节点
 * 输入参数  : zskiplist *zsl  表头指针
               double score    节点分数
               robj *obj       节点数据指针
 * 输出参数  : 无
 * 返 回 值  :
 * 调用关系  :
 * 记    录
 * 1.日    期: 2017年06月13日
 *   作    者: zyz
 *   修改内容: 新生成函数

*****************************************************************************/
```

```
int zslDelete(zskiplist *zsl, double score, robj *obj) {
    zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;
    int i;

    x = zsl->header;
    // 遍历所有层，记录删除节点后需要被修改的节点到 update 数组
    for (i = zsl->level-1; i >= 0; i--) {
        //指针前移首要条件是前向节点指针不为空，次要条件是分数小于指定分数，或即使分数相等，节点成员对象也不相等------> 前向指针前移的必要条件：分数小于或等于指定分数
        while (x->level[i].forward && //前向指针不为空
            (x->level[i].forward->score < score || //前向节点分数小于指定分数
                (x->level[i].forward->score == score &&  //前向节点分数等于指定分数
                compareStringObjects(x->level[i].forward->obj,obj) < 0)))//前向节点成员对象不相同
            x = x->level[i].forward;
        //保存待删除节点的前一节点指针
        update[i] = x;
    }
    // 因为多个不同的 member 可能有相同的 score
    // 所以要确保 x 的 member 和 score 都匹配时，才进行删除

    x = x->level[0].forward;

    if (x && score == x->score && equalStringObjects(x->obj,obj)) {
        zslDeleteNode(zsl, x, update);
        zslFreeNode(x);
        return 1;
    }
    return 0; /* not found */
}
```

2.5 跳跃表的查询

跳跃表提供了根据排名查询元素，以及根据分数或群排名的API，间接提供了根据分数获取元素的API，查询体现了跳跃表的优势，但实现相对简单，主要是判断在当前层比对的元素是否是否小于给定元素，如果小于，且其后继指针不为空，则继续往前查找（这效率是很高的），否则往下一层找（效率相对低一点）：

```
 /* Find the rank for an element by both score and key.
 * Returns 0 when the element cannot be found, rank otherwise.
 * Note that the rank is 1-based due to the span of zsl->header to the
 * first element. */
/*****************************************************************************
 * 函 数 名  : zslGetRank
 * 函数功能  : 获取指定分数和成员数据对象确定节点的排名
 * 输入参数  : zskiplist *zsl  跳跃表指针
               double score    节点分数
               robj *o         成员数据对象指针
 * 输出参数  : 无
 * 返 回 值  : unsigned
 * 调用关系  :
 * 记     录
 * 1.日    期: 2017年06月29日
 *   作    者: zyz
 *   修改内容: 新生成函数
*****************************************************************************/
unsigned long zslGetRank(zskiplist *zsl, double score, robj *o) {
    zskiplistNode *x;
    unsigned long rank = 0;
    int i;

    x = zsl->header;
    for (i = zsl->level-1; i >= 0; i--) {
        //指针前移的必要条件是后继指针不为空
        while (x->level[i].forward &&
            (x->level[i].forward->score < score ||
                (x->level[i].forward->score == score &&
                compareStringObjects(x->level[i].forward->obj,o) <= 0))) {
            rank += x->level[i].span;
            //排名累加
            x = x->level[i].forward;
        }

        /* x might be equal to zsl->header, so test if obj is non-NULL */
        if (x->obj && equalStringObjects(x->obj,o)) {
            return rank;
        }
    }
    return 0;
}

/* Finds an element by its rank. The rank argument needs to be 1-based. */
/*****************************************************************************
 * 函 数 名  : zslGetElementByRank
 * 函数功能  : 根据排名获取跳跃表元素
 * 输入参数  : zskiplist *zsl      跳跃表描述结构指针
               unsigned long rank  排名
 * 输出参数  : 无
 * 返 回 值  :
```

```
 * 调用关系   :
 * 记     录
 * 1.日     期: 2018年05月10日
 *   作     者:
 *   修改内容: 新生成函数
*******************************************************************************/
zskiplistNode* zslGetElementByRank(zskiplist *zsl, unsigned long rank) {
    zskiplistNode *x;
    unsigned long traversed = 0;
    int i;

    x = zsl->header;
    for (i = zsl->level-1; i >= 0; i--) {
        while (x->level[i].forward && (traversed + x->level[i].span) <= rank)
        {
            traversed += x->level[i].span;
            x = x->level[i].forward;
        }
        if (traversed == rank) {
            return x;
        }
    }
    return NULL;
}
```

总结：redis为了能根据排名查询元素，在每一个层维护了一个span字段，相对来说增加了不少实现的复杂度，在我看来是得不偿失的，不知道redis的在这个上面还有其他奥妙没有，或许我没有发现。

Redis是单进程单线程的，如何能够高并发?

采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消 耗）

Redis如何使用Redis实现分布式锁?

Redis分布式锁的正确实现方式

前言

分布式锁一般有三种实现方式：1. 数据库乐观锁；2. 基于Redis的分布式锁；3. 基于ZooKeeper的分布式锁。本篇博客将介绍第二种方式，基于Redis实现分布式锁。虽然网上已经有各种介绍Redis分布式锁实现的博客，然而他们的实现却有着各种各样的问题，为了避免误人子弟，本篇博客将详细介绍如何正确地实现Redis分布式锁。

可靠性

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1. 互斥性。在任意时刻，只有一个客户端能持有锁。
2. 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
4. 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

代码实现

组件依赖

首先我们要通过Maven引入`Jedis`开源组件，在`pom.xml`文件加入下面的代码：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

加锁代码

正确姿势

Talk is cheap, show me the code。先展示代码，再带大家慢慢解释为什么这样实现：

```
public class RedisTool {

    private static final String LOCK_SUCCESS = "OK";
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

    /**
```

```
     * 尝试获取分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {

        String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, expireTime);

        if (LOCK_SUCCESS.equals(result)) {
            return true;
        }
        return false;

    }

}
```

可以看到，我们加锁就一行代码：`jedis.set(String key, String value, String nxxx, String expx, int time)`，这个set()方法一共有五个形参：

- 第一个为key，我们使用key来当锁，因为key是唯一的。
- 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用`UUID.randomUUID().toString()`方法生成。
- 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
- 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
- 第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

错误示例1

比较常见的错误示例就是使用`jedis.setnx()`和`jedis.expire()`组合实现加锁，代码如下：

```
public static void wrongGetLock1(Jedis jedis, String lockKey, String requestId, int expireTime) {

    Long result = jedis.setnx(lockKey, requestId);
    if (result == 1) {
        // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁
        jedis.expire(lockKey, expireTime);
    }

}
```

setnx()方法作用就是SET IF NOT EXIST，expire()方法就是给锁加一个过期时间。乍一看好像和前面的set()方法结果一样，然而由于这是两条Redis命令，不具有原子性，如果程序在执行完setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。网上之所以有人这样实现，是因为低版本的jedis并不支持多参数的set()方法。

错误示例2

```
public static boolean wrongGetLock2(Jedis jedis, String lockKey, int expireTime) {

    long expires = System.currentTimeMillis() + expireTime;
    String expiresStr = String.valueOf(expires);

    // 如果当前锁不存在，返回加锁成功
    if (jedis.setnx(lockKey, expiresStr) == 1) {
        return true;
    }

    // 如果锁存在，获取锁的过期时间
    String currentValueStr = jedis.get(lockKey);
    if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {
        // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间
        String oldValueStr = jedis.getSet(lockKey, expiresStr);
        if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {
            // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才有权利加锁
            return true;
        }
    }

    // 其他情况，一律返回加锁失败
```

```
    return false;

}
```

这一种错误示例就比较难以发现问题，而且实现也比较复杂。实现思路：使用`jedis.setnx()`命令实现加锁，其中key是锁，value是锁的过期时间。执行过程：1. 通过setnx()方法尝试加锁，如果当前锁不存在，返回加锁成功。2. 如果锁已经存在则获取锁的过期时间，和当前时间比较，如果锁已经过期，则设置新的过期时间，返回加锁成功。代码如下：

那么这段代码问题在哪里？1. 由于是客户端自己生成过期时间，所以需要强制要求分布式下每个客户端的时间必须同步。 2. 当锁过期的时候，如果多个客户端同时执行`jedis.getSet()`方法，那么虽然最终只有一个客户端可以加锁，但是这个客户端的锁的过期时间可能被其他客户端覆盖。3. 锁不具备拥有者标识，即任何客户端都可以解锁。

解锁代码

正确姿势

还是先展示代码，再带大家慢慢解释为什么这样实现：

```
public class RedisTool {

    private static final Long RELEASE_SUCCESS = 1L;

    /**
     * 释放分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @return 是否释放成功
     */
    public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {

        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";
        Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));

        if (RELEASE_SUCCESS.equals(result)) {
            return true;
        }
        return false;

    }

}
```

可以看到，我们解锁只需要两行代码就搞定了！第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《黑客与画家》里，没想到这次居然用上了。第二行代码，我们将Lua代码传到`jedis.eval()`方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。那么为什么执行eval()方法可以确保原子性，源于Redis的特性，下面是官网对eval命令的部分解释：

简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命令。

错误示例1

最常见的解锁代码就是直接使用`jedis.del()`方法删除锁，这种不先判断锁的拥有者而直接解锁的方式，会导致任何客户端都可以随时进行解锁，即使这把锁不是它的。

```
public static void wrongReleaseLock1(Jedis jedis, String lockKey) {
    jedis.del(lockKey);
}
```

错误示例2

这种解锁代码乍一看也是没问题，甚至我之前也差点这样实现，与正确姿势差不多，唯一区别的是分成两条命令去执行，代码如下：

```
public static void wrongReleaseLock2(Jedis jedis, String lockKey, String requestId) {

    // 判断加锁与解锁是不是同一个客户端
    if (requestId.equals(jedis.get(lockKey))) {
        // 若在此时，这把锁突然不是这个客户端的，则会误解锁
        jedis.del(lockKey);
    }

}
```

如代码注释，问题在于如果调用`jedis.del()`方法的时候，这把锁已经不属于当前客户端的时候会解除他人加的锁。那么是否真的有这种场景？答案是肯定的，比如客户端A加锁，一段时间之后客户端A解锁，在执行`jedis.del()`之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

总结

本文主要介绍了如何使用Java代码正确实现Redis分布式锁，对于加锁和解锁也分别给出了两个比较经典的错误示例。其实想要通过Redis实现分布式锁并不难，只要保证能满足可靠性里的四个条件。互联网虽然给我们带来了方便，只要有问题就可以google，然而网上的答案一定是对的吗？其实不然，所以我们更应该时刻保持着质疑精神，多想多验证。

如果你的项目中Redis是多机部署的，那么可以尝试使用`Redisson`实现分布式锁，这是Redis官方提供的Java组件

Redis分布式锁操作的原子性，Redis内部是如何实现的?

```
setnx
Incrby\Decrby
```

网络（周老师，李老师）

计算机网络体系结构

在计算机网络的基本概念中，分层次的体系结构是 基本的。计算机网络体系结

构的抽象概念较多，在学习时要多思考。这些概念对后面的学习很有帮助。

网络协议是什么?

在计算机网络要做到有条不紊地交换数据，就必须遵守一些事先约定好的规则， 比如交换数据的格式、是否需要发送一个应答信息。这些规则被称为网络协议。

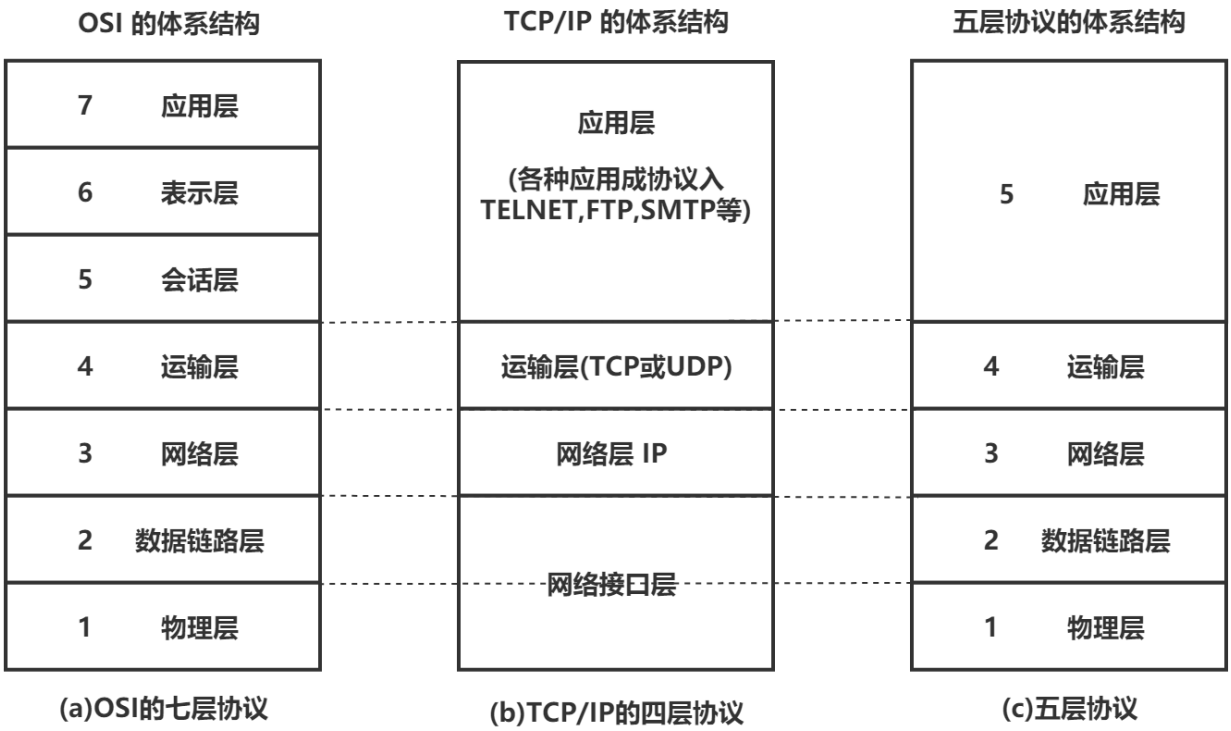
为什么要对网络协议分层?

- 简化问题难度和复杂度。由于各层之间独立，我们可以分割大问题为小问题。
- 灵活性好。当其中一层的技术变化时，只要层间接口关系保持不变，其他层不受 影响。
- 易于实现和维护。
- 促进标准化工作。分开后，每层功能可以相对简单地被描述。

网络协议分层的缺点： 功能可能出现在多个层里，产生了额外开销。 为了使不同体系结构的计算机网络都能互联，国际标准化组织 ISO 于1977年提 出了一个试图使各种计算机在世界范围内互联成网的标准框架，即著名的开放系统互联基本参考模型 OSI/RM，简称为OSI。

OSI 的七层协议体系结构的概念清楚，理论也较完整，但它既复杂又不实用， TCP/IP 体系结构则不同，但它现在却得到了非常广泛的应用。TCP/IP 是一个四 层体系结构，它包含应用层，运输层，网际层和网络接口层（用网际层这个名字 是强调这一层是为了解决不同网络的互连问题），不过从实质上讲，TCP/IP 只 有上面的三层，因为下面的网络接口层并没有什么具体内容，因此在学习计 算机网络的原理时往往采用折中的办法，即综合 OSI 和 TCP/IP 的优点，采用 一种只有五层协议的体系结构，这样既简洁又能将概念阐述清楚，有时为了方 便，也可把底下两层称为网络接口层。

- 四层协议，五层协议和七层协议的关系如下：
- TCP/IP是一个四层的体系结构，主要包括：应用层、运输层、网际层和网络接 口层。
- 五层协议的体系结构主要包括：应用层、运输层、网络层，数据链路层和物理层。
- OSI七层协议模型主要包括是：应用层（Application）、表示层 （Presentation）、会话层（Session）、运输层（Transport）、网络层 （Network）、数据链路层（Data Link）、物理层（Physical）。



注：五层协议的体系结构只是为了介绍网络原理而设计的，实际应用还是 TCP/IP 四层体系结构。

TCP/IP 四层体系结构。 TCP/IP 协议族

应用层

应用层(application-layer ）的任务是通过应用进程间的交互来完成特定网络应 用。应用层协议定义的是应用进程（进程：主机中正在运行的程序）间的通信和 交互的规则。

对于不同的网络应用需要不同的应用层协议。在互联网中应用层协议很多，如域 名系统 DNS，支持万维网应用的 HTTP 协议，支持电子邮件的 SMTP 协议等 等。

运输层

运输层(transport layer)的主要任务就是负责向两台主机进程之间的通信提供通 用的数据传输服务。应用进程利用该服务传送应用层报文。 运输层主要使用一下两种协议

1. 传输控制协议-TCP：提供面向连接的，可靠的数据传输服务。
2. 用户数据协议-UDP：提供无连接的，尽大努力的数据传输服务（不 保证数据传输的可靠性）。

UDP	TCP	
是否连接	无连接	面向连接
是否可靠	不可靠传 输，不使 用流量控 制和拥塞 控制	可靠传 输，使用 流量控制 和拥塞控 制
连接对象 个数	支持一对 一，一对 多，多对 一和多对 多交互通 信	只能是一 对一通信
传输方式	面向报文	面向字节 流
首部开销	首部开销 小，仅8字 节	首部小 20字节， 大60字 节
场景	适用于实 时应用 （IP电 话、视频会议、直 播等）	适用于要 求可靠传 输的应 用，例如 文件传输

每一个应用层（TCP/IP参考模型的最高层）协议一般都会使用到两个传输层协 议之一：

运行在TCP协议上的协议：

- **HTTP（Hypertext Transfer Protocol，超文本传输协议）**，主要用于普通浏 览。
- **HTTPS（HTTP over SSL，安全超文本传输协议）**,HTTP协议的安全版本。
- **FTP（File Transfer Protocol，文件传输协议）**，用于文件传输。
- **POP3（Post Office Protocol, version 3，邮局协议）**，收邮件用。
- **SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）**，用来发送电子 邮件。
- **TELNET（Teletype over the Network，网络电传）**，通过一个终端 （terminal）登陆到网络。
- **SSH（Secure Shell，用于替代安全性差的TELNET）**，用于加密安全登陆用。 运行在UDP协议上的协议：
- **BOOTP（Boot Protocol，启动协议）**，应用于无盘设备。
- **NTP（Network Time Protocol，网络时间协议）**，用于网络同步。
- **DHCP（Dynamic Host Configuration Protocol，动态主机配置协议）**，动态 配置IP地址。 运行在TCP和UDP协议上：
- **DNS（Domain Name Service，域名服务）**，用于完成地址查找，邮件转发等 工作。

网络层

网络层的任务就是选择合适的网间路由和交换结点，确保计算机通信的数据及时 传送。在发送数据时，网络层把运输层产生的报文段或用户数据报封装成分组和 包进行传送。在TCP/IP 体系结构中，由于网络层使用 IP 协议，因此分组也叫 IP 数据报 ，简称数据报。

互联网是由大量的异构（heterogeneous）网络通过路由器（router）相互连 接起来的。互联网使用的网络层协议是无连接的网际协议（Intert Prococol） 和许多路由选择协议，因此互联网的网络层也叫做网际层或 IP 层。

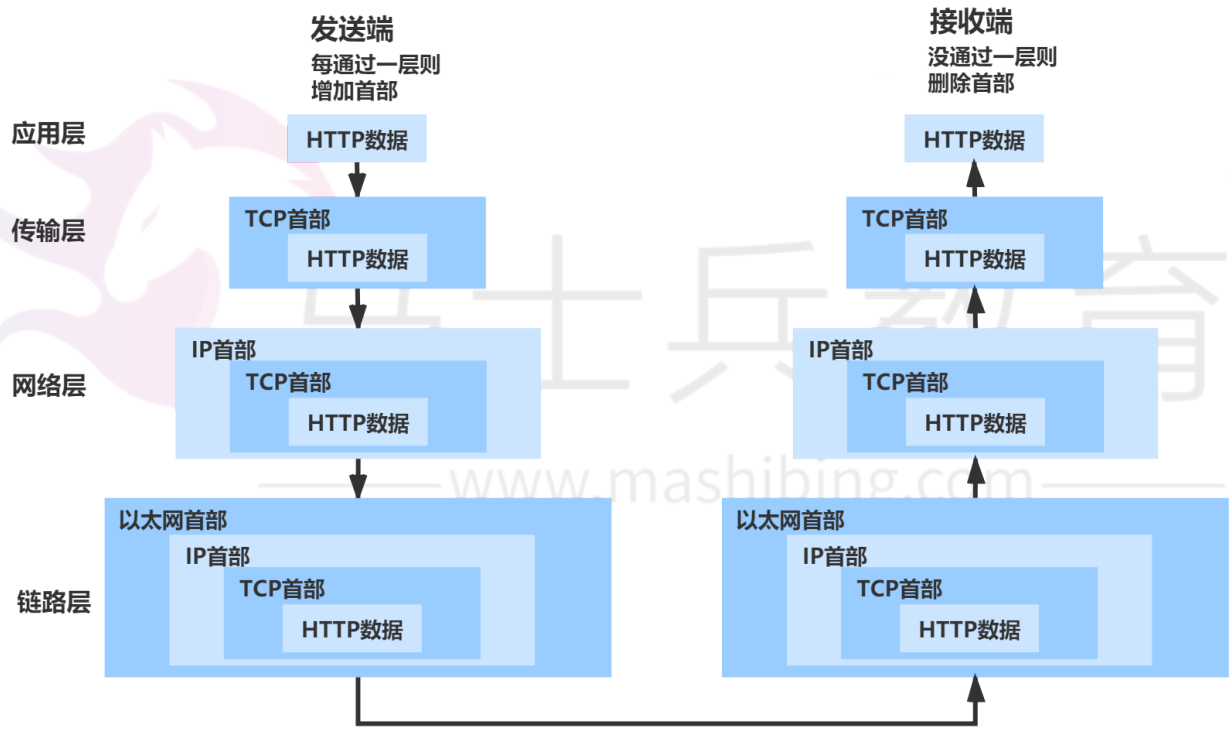
数据链路层

数据链路层(data link layer)通常简称为链路层。两台主机之间的数据传输，总 是在一段一段的链路上传送的，这就需要使用专门的链路层的协议。

在两个相邻节点之间传送数据时，数据链路层将网络层交下来的 IP 数据报组装 成帧，在两个相邻节点间的链路上传送帧。每一帧包括数据和必要的控制信息 （如同步信息，地址信息，差错控制等）。

在接收数据时，控制信息使接收端能够知道一个帧从哪个比特开始和到哪个比特 结束。

一般的web应用的通信传输流是这样的：



发送端在层与层之间传输数据时，每经过一层时会被打上一个该层所属的首部信 息。反之，接收端在层与层之间传输数据时，每经过一层时会把对应的首部信息 去除。

物理层

在物理层上所传送的数据单位是比特。 物理层(physical layer)的作用是实现相 邻计算机节点之间比特流的透明传送，尽可能屏蔽掉具体传输介质和物理设备的 差异。使其上面的数据链路层不必考虑网络的具体传输介质是什么。“透明传送 比特流”表示经实际电路传送后的比特流没有发生变化，对传送的比特流来说， 这个电路好像是看不见的。

TCP/IP 协议族

在互联网使用的各种协议中重要和著名的就是 TCP/IP 两个协议。现在人们 经常提到的 TCP/IP 并不一定是单指 TCP 和 IP 这两个具体的协议，而往往是表 示互联网所使用的整个 TCP/IP 协议族。

> 互联网协议套件（英语：Internet Protocol Suite，缩写**IPS**）是一个网络通讯模型， 以及一整个网络传输协议家族，为网际网络的基础通讯架构。它常被通称为TCP/IP协议族（英语：**TCP/IP Protocol Suite**，或**TCP/IP Protocols**），简称**TCP/IP**。因为该 协定家族的两个核心协定：**TCP（传输控制协议）和IP（网际协议）**，为该家族中早通过的标准。

划重点：

TCP（传输控制协议）和IP（网际协议） 是先定义的两个核心协议，所以才统称为**TCP/IP协议族**

TCP的三次握手四次挥手

TCP是一种面向连接的、可靠的、基于字节流的传输层通信协议，在发送数据 前，通信双方必须在彼此间建立一条连接。所谓的“连接”，其实是客户端和服 务端保存的一份关于对方的信息，如ip地址、端口号等。

TCP可以看成是一种字节流，它会处理IP层或以下的层的丢包、重复以及错误问 题。在连接的建立过程中，双方需要交换一些连接的参数。这些参数可以放在 TCP头部。

一个TCP连接由一个4元组构成，分别是两个IP地址和两个端口号。一个TCP连 接通常分为三个阶段：连接、数据传输、退出（关闭）。**通过三次握手建立一个 链接，通过四次挥手来关闭一个连接。**

当一个连接被建立或被终止时，交换的报文段只包含TCP头部，而没有数据。

TCP报文的头部结构

在了解TCP连接之前先来了解一下TCP报文的头部结构。

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port（16位源端口号）																Destination port（16位目标端口号）															
4	32	Sequence number （32位序列号）																															
8	64	Acknowledgment number (if ACK set) （32位确认号）																															
12	96	Data offset （4位头部长度）				Reserved（6位保留）				U R G	A C K	P S H	R S T	S Y N	F I N			Window Size （16位窗口大小）															
16	128	Checksum （16校验和）																Urgent pointer (if URG set)（16紧急指针）															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) 选项，长度可变，最多40个字节[kind(1字节)，length(1字节)，info(n字节)]																															

上图中有几个字段需要重点介绍下：

（1）序号：seq序号，占32位，用来标识从TCP源端向目的端发送的字节流， 发起方发送数据时对此进行标记。

（2）确认序号：ack序号，占32位，只有ACK标志位为1时，确认序号字段才有 效，ack=seq+1。

（3）标志位：共6个，即URG、ACK、PSH、RST、SYN、FIN等，具体含义如 下：

- ACK：确认序号有效。
- FIN：释放一个连接。
- PSH：接收方应该尽快将这个报文交给应用层。
- RST：重置连接。
- SYN：发起一个新连接。
- URG：紧急指针（urgent pointer）有效。

需要注意的是：

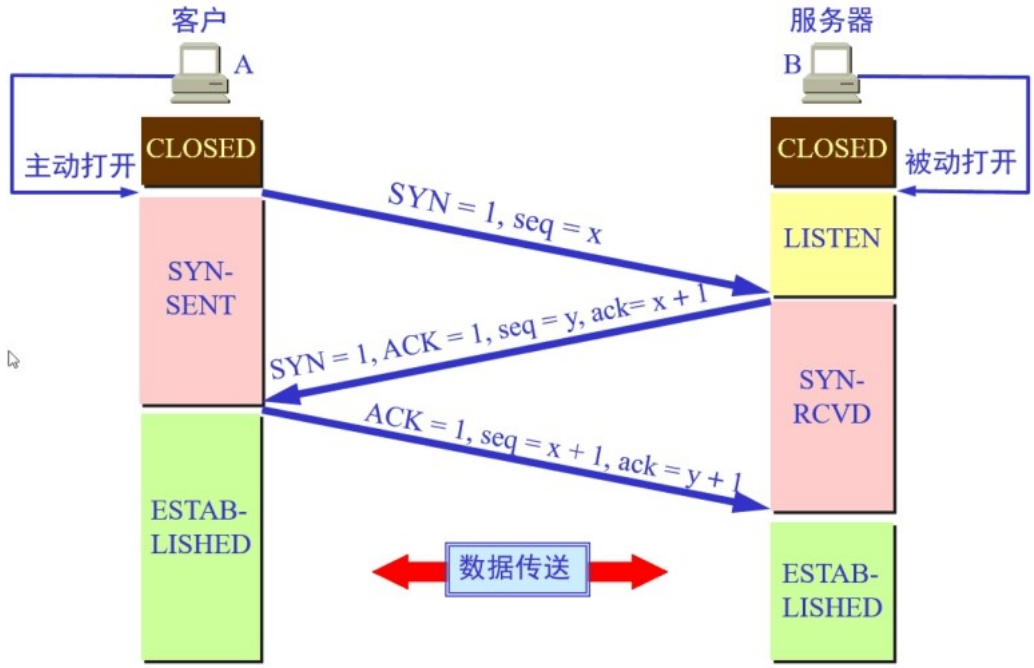
- 不要将确认序号ack与标志位中的ACK搞混了。
- 确认方ack=发起方seq+1，两端配对。

三次握手

三次握手的本质是确认通信双方收发数据的能力首先，我让信使运输一份信件给对方，对方收到了，那么他就知道了我的发件能力和他的收件能力是可以的。

于是他给我回信，我若收到了，我便知我的发件能力和他的收件能力是可以的，并且他的发件能力和我的收件能力是可以。

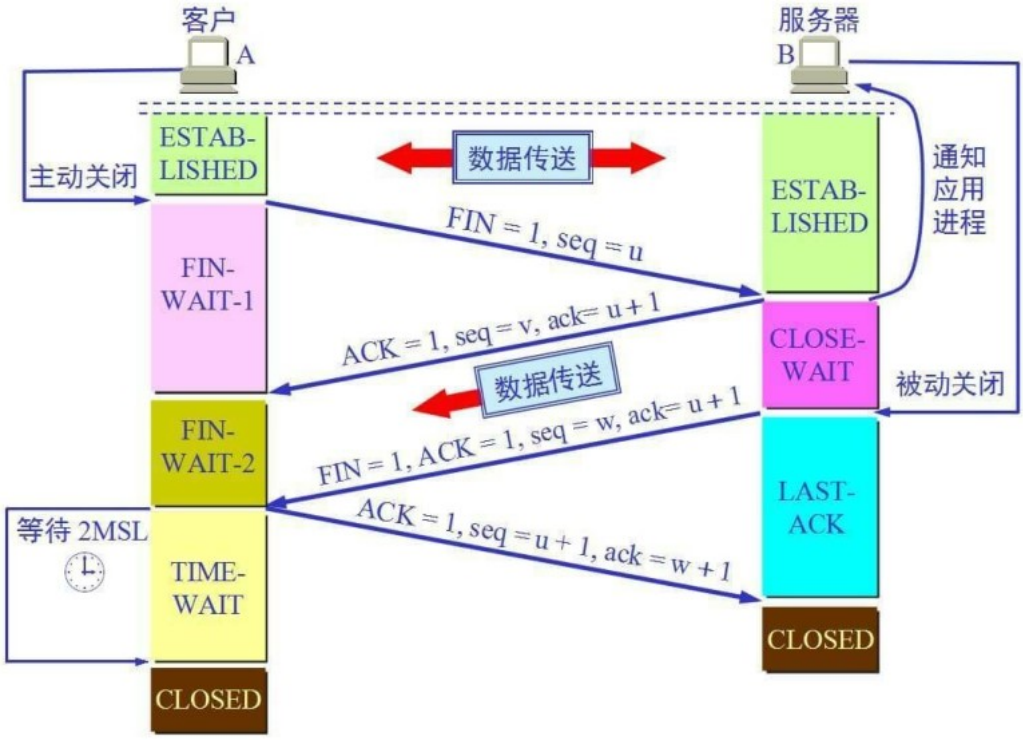
然而此时他还不知道他的发件能力和我的收件能力到底可不可以，于是我 后回馈一次，他若收到了，他便清楚了他的发件能力和我的收件能力是可以的。这，就是三次握手，这样说，你理解了吗？



- 第一次握手：客户端要向服务端发起连接请求，首先客户端随机生成一个起始序列号ISN(比如是100)，那客户端向服务端发送的报文段包含SYN标志位(也就是SYN=1)，序列号seq=100。
- 第二次握手：服务端收到客户端发过来的报文后，发现SYN=1，知道这是一个连接请求，于是将客户端的起始序列号100存起来，并且随机生成一个服务端的起始序列号(比如是300)。然后给客户端回复一段报文，回复报文包含SYN和ACK标志(也就是SYN=1,ACK=1)、序列号seq=300、确认号ack=101(客户端发过来的序列号+1)。
- 第三次握手：客户端收到服务端的回复后发现ACK=1并且ack=101,于是知道服务端已经收到了序列号为100的那段报文；同时发现SYN=1，知道了服务端同意了这次连接，于是就将服务端的序列号300给存下来。然后客户端再回复一段报文给服务端，报文包含ACK标志位(ACK=1)、ack=301(服务端序列号+1)、seq=101(第一次握手时发送报文是占据一个序列号的，所以这次seq就从101开始，需要注意的是不携带数据的ACK报文是不占据序列号的，所以后面第一次正式发送数据时seq还是101)。当服务端收到报文后发现ACK=1并且ack=301，就知道客户端收到序列号为300的报文了，就这样客户端和服务端通过TCP建立了连接。

四次挥手

四次挥手的目的是关闭一个连接



比如客户端初始化的序列号ISA=100，服务端初始化的序列号ISA=300。TCP连接成功后客户端总共发送了1000个字节的数据，服务端在客户端发FIN报文前总共回复了2000个字节的数据。

- 第一次挥手：当客户端的数据都传输完成后，客户端向服务端发出连接释放报文(当然数据没发完时也可以发送连接释放报文并停止发送数据)，释放连接报文包含FIN标志位(FIN=1)、序列号seq=1101(100+1+1000，其中的1是建立连接时占的一个序列号)。需要注意的是客户端发出FIN报文段后只是不能发数据了，但是还可以正常收数据；另外FIN报文段即使不携带数据也要占据一个序列号。
- 第二次挥手：服务端收到客户端发的FIN报文后给客户端回复确认报文，确认报文包含ACK标志位(ACK=1)、确认号ack=1102(客户端FIN报文序列号1101+1)、序列号seq=2300(300+2000)。此时服务端处于关闭等待状态，而不是立马给客户端发FIN报文，这个状态还要持续一段时间，因为服务端可能还有数据没发完。
- 第三次挥手：服务端将最后数据(比如50个字节)发送完毕后就向客户端发出连接释放报文，报文包含FIN和ACK标志位(FIN=1,ACK=1)、确认号和第二次挥手一样ack=1102、序列号seq=2350(2300+50)。
- 第四次挥手：客户端收到服务端发的FIN报文后，向服务端发出确认报文，确认报文包含ACK标志位(ACK=1)、确认号ack=2351、序列号seq=1102。注意客户端发出确认报文后不是立马释放TCP连接，而是要经过2MSL(最长报文段寿命的2倍时长)后才释放TCP连接。而服务端一旦收到客户端发出的确认报文就会立马释放TCP连接，所以服务端结束TCP连接的时间要比客户端早一些。

常见面试题

为什么TCP连接的时候是3次？2次不可以吗?

因为需要考虑连接时丢包的问题，如果只握手2次，第二次握手时如果服务端发给客户端的确认报文段丢失，此时服务端已经准备好了收发数(可以理解服务端已经连接成功)据，而客户端一直没收到服务端的确认报文，所以客户端就不知道服务端是否已经准备好了(可以理解为客户端未连接成功)，这种情况下客户端不会给服务端发数据，也会忽略服务端发过来的数据。

如果是三次握手，即便发生丢包也不会有问题，比如如果第三次握手客户端发的确认ack报文丢失，服务端在一段时间内没有收到确认ack报文的话就会重新进

行第二次握手，也就是服务端会重发SYN报文段，客户端收到重发的报文段后会再次给服务端发送确认ack报文。

为什么TCP连接的时候是3次，关闭的时候却是4次?

因为只有在客户端和服务端都没有数据要发送的时候才能断开TCP。而客户端发出FIN报文时只能保证客户端没有数据发了，服务端还有没有数据发客户端是不知道的。而服务端收到客户端的FIN报文后只能先回复客户端一个确认报文来告诉客户端我服务端已经收到你的FIN报文了，但我服务端还有一些数据没发完，等这些数据发完了服务端才能给客户端发FIN报文(所以不能一次性将确认报文和

FIN报文发给客户端，就是这里多出来了一次)。

为什么客户端发出第四次挥手的确认报文后要等2MSL的时间才能释放TCP连接?

这里同样是要考虑丢包的问题，如果第四次挥手的报文丢失，服务端没收到确认 ack报文就会重发第三次挥手的报文，这样报文一去一回 长时间就是2MSL，所以需要等这么长时间来确认服务端确实已经收到了。

如果已经建立了连接，但是客户端突然出现故障了怎么办?

TCP设有一个保活计时器，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。什么是HTTP，HTTP 与 HTTPS 的区别

HTTP 是一个在计算机世界里专门在两点之间传输文字、图片、音频、视频等超文本数据的约定和规范

区别	HTTP	HTTPS
协议	运行在 TCP 之上，明文传输，客户端与服务器端都无法验证对方的身份	身披 SSL(Secure Socket Layer)外壳的 HTTP，运行于 SSL 上，SSL 运行于 TCP 之 上， 是添加了加密和认证机制的 HTTP。
端口	80	443
资源消耗	较少	由于加解密处理，会消耗更 多的 CPU 和内存资源
开销	无需证书	需要证书，而证书一般需要向认证机构购买
加密机制	无	共享密钥加密和公开密钥加密并用的混合加密机制
安全性	弱	由于加密机制，安全性强

常用HTTP状态码

HTTP状态码表示客户端HTTP请求的返回结果、标识服务器处理是否正常、表明请求出现的错误等。

状态码的类别：

类别	原因短语
1XX	Informational（信息性状态码）接受的请求正在处理
2XX	Success（成功状态码）请求正常处理完毕
3XX	Redirection（重定向状态码）需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）服务器无法处理请求
5XX	Server Error（服务器错误状态码）服务器处理请求出错

常用HTTP状态码：

2XX	成功（这系列表明请求被正常处理了）
200	OK，表示从客户端发来的请求在服务器端被正确处理
204	No content，表示请求成功，但响应报文不含实体的主体部分
206	Partial Content ，进行范围请求成功
3XX	重定向 （表明浏览器要执行特殊处理）
301	moved permanently，永久 性重定向，表示资源已被分配了新的 URL
302	found，临时性重定向，表示资源临时被分配了新的 URL
303	see other，表示资源存在着另一个 URL， 应使用 GET 方法获取资源 （对于 301/302/ 303响应，几乎所有浏览器都会删除报文主体并 自动用 GET重新请求）
304	not modified ，表示服务器允许访问资源，但请求未满足条件的情况（与重定向无关）
307	temporary redirect，临时重定 向，和302 含义类似，但是期望客户端保持请求方法不变向新的地址发出请求
4XX	客户端错误
400	bad request，请求报文存在语法错误
401	unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息
403	forbidden ，表示对请求资源的访问被服务器拒绝，可在实体主体部分返回原因描述
404	not found，表示在服务器上没有找到请求的资源
5XX	服务器错误
500	internal sever error，表 示服务器端在执行请求时发生了错误
501	Not Implemented，表示服务器不支持当前请求所需要的某个功能
503	service unavailable，表明服务器暂时处于超负载或正在停机维护，无法处理请求

GET和POST区别

说道GET和POST，就不得不提HTTP协议，因为浏览器和服务器的交互是通过

HTTP协议执行的，而GET和POST也是HTTP协议中的两种方法。

HTTP全称为Hyper Text Transfer Protocol，中文翻译为超文本传输协议，目的是保证浏览器与服务器之间的通信。HTTP的工作方式是客户端与服务器之间的请求-应答协议。

HTTP协议中定义了浏览器和服务器进行交互的不同方法，基本方法有4种，分别是GET，POST，PUT，DELETE。这四种方法可以理解为，对服务器资源的查，改，增，删。

- GET：从服务器上获取数据，也就是所谓的查，仅仅是获取服务器资源，不进行修改。
- POST：向服务器提交数据，这就涉及到了数据的更新，也就是更改服务器的数据。
- PUT：英文含义是放置，也就是向服务器新添加数据，就是所谓的增。

- DELETE：从字面意思也能看出，这种方式就是删除服务器数据的过程。

GET和POST区别

1. Get是不安全的，因为在传输过程，数据被放在请求的URL中；Post的所有操作对用户来说都是不可见的。 但是这种做法也不时绝对的，大部分人的做法也是按照上面的说法来的，但是也可以在get请求加上 request body，给 post请求带上 URL 参数。
2. Get请求提交的url中的数据 多只能是2048字节，这个限制是浏览器或者服务器给添加的，http协议并没有对url长度进行限制，目的是为了保证服务器和浏览器能够正常运行，防止有人恶意发送请求。Post请求则没有大小限制。
3. Get限制Form表单的数据集的值必须为ASCII字符；而Post支持整个

ISO10646字符集。

4. Get执行效率却比Post方法好。Get是form提交的默认方法。
5. GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

什么是对称加密与非对称加密

对称密钥加密是指加密和解密使用同一个密钥的方式，这种方式存在的最大问题就是密钥发送问题，即如何安全地将密钥发给对方；而非对称加密是指使用一对非对称密钥，即公钥和私钥，公钥可以随意发布，但私钥只有自己知道。发送密文的一方使用对方的公钥进行加密处理，对方接收到加密信息后，使用自己的私钥进行解密。

由于非对称加密的方式不需要发送用来解密的私钥，所以可以保证安全性；但是和对称加密比起来，非常的慢什么是HTTP2

HTTP2 可以提高了网页的性能。

在 HTTP1 中浏览器限制了同一个域名下的请求数量（Chrome 下一般是六

个），当在请求很多资源的时候，由于队头阻塞当浏览器达到 大请求数量时，剩余的资源需等待当前的六个请求完成后才能发起请求。

HTTP2 中引入了多路复用的技术，这个技术可以只通过一个 TCP 连接就可以传输所有的请求数据。多路复用可以绕过浏览器限制同一个域名下的请求数量的问题，进而提高了网页的性能。

Session、Cookie和Token的主要区别

HTTP协议本身是无状态的。什么是无状态呢，即服务器无法判断用户身份。

什么是cookie

cookie是由Web服务器保存在用户浏览器上的小文件（key-value格式），包含用户相关的信息。客户端向服务器发起请求，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。客户端浏览器会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户身份。

什么是session

session是依赖Cookie实现的。session是服务器端对象

session 是浏览器和服务器会话过程中，服务器分配的一块储存空间。服务器默

认为浏览器在cookie中设置 sessionid，浏览器在向服务器请求过程中传输

cookie 包含 sessionid ，服务器根据 sessionid 获取出会话中存储的信息，然后确定会话的身份信息。

cookie与session区别

存储位置与安全性：cookie数据存放在客户端上，安全性较差，session数据放在服务器上，安全性相对更高；

存储空间：单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点多保存20个cookie，session无此限制

占用服务器资源：session一定时间内保存在服务器上，当访问增多，占用服务器性能，考虑到服务器性能方面，应当使用cookie。

什么是Token

Token的引入：Token是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。

Token的定义：Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

Token 是在服务端产生的。如果前端使用用户名/密码向服务端请求认证，服务端认证成功，那么在服务端会返回 Token 给前端。前端可以在每次请求的时候带上 Token 证明自己的合法地位 session与token区别

- session机制存在服务器压力增大，CSRF跨站伪造请求攻击，扩展性不强等问题；
- session存储在服务器端，token存储在客户端
- token提供认证和授权功能，作为身份认证，token安全性比session好；
- session这种会话存储方式方式只适用于客户端代码和服务端代码运行在同一台服务器上，token适用于项目级的前后端分离（前后端代码运行在不同的服务器下）

Servlet是线程安全的吗

Servlet不是线程安全的，多线程并发的读写会导致数据不同步的问题。解决的办法是尽量不要定义name属性，而是要把name变量分别定义在doGet() 和doPost()方法内。虽然使用synchronized(name){}语句块可以解决问题，但是会造成线程的等待，不是很科学的办法。

注意：多线程的并发的读写Servlet类属性会导致数据不同步。但是如果只是并发地读取属性而不写入，则不存在数据不同步的问题。因此Servlet里的只读属性最好定义为final类型的。

Servlet接口中有哪些方法及Servlet生命周期探秘

在Java Web程序中，Servlet主要负责接收用户请求HttpServletRequest，在 doGet()，doPost()中做相应的处理，并将回应HttpServletResponse反馈给用户。Servlet可以设置初始化参数，供Servlet内部使用。

Servlet接口定义了5个方法，其中前三个方法与Servlet生命周期相关：

- void init(ServletConfig config) throws ServletException
- void service(ServletRequest req, ServletResponse resp) throws ServletException, java.io.IOException

- void destory()
- java.lang.String getServletInfo()
- ServletConfig getServletConfig()

生命周期：

Web容器加载Servlet并将其实例化后，Servlet生命周期开始，容器运行其 init()方法进行Servlet的初始化；

请求到达时调用Servlet的service()方法，service()方法会根据需要调用与请求

对应的doGet或doPost等方法；

当服务器关闭或项目被卸载时服务器会将Servlet实例销毁，此时会调用Servlet 的destroy()方法。

init方法和destory方法只会执行一次，service方法客户端每次请求Servlet都会执行。Servlet中有时会用到一些需要初始化与销毁的资源，因此可以把初始化资源的代码放入init方法中，销毁资源的代码放入destroy方法中，这样就不需要每次处理客户端的请求都要初始化与销毁资源。

如果客户端禁止 cookie 能实现 session 还能用吗?

Cookie 与 Session，一般认为是两个独立的东西，Session采用的是在服务器端保持状态的方案，而Cookie采用的是在客户端保持状态的方案。

但为什么禁用Cookie就不能得到Session呢？因为Session是用Session ID来确定当前对话所对应的服务器Session，而Session ID是通过Cookie来传递的，禁用Cookie相当于失去了Session ID，也就得不到Session了。

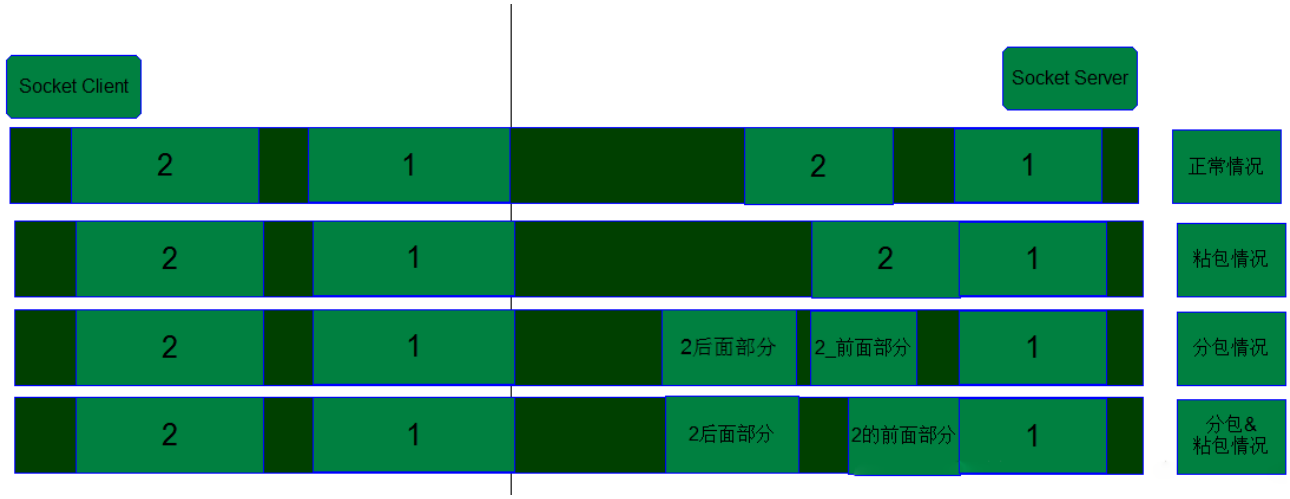
假定用户关闭Cookie的情况下使用Session，其实现途径有以下几种：

1. 手动通过URL传值、隐藏表单传递Session ID。
2. 用文件、数据库等形式保存Session ID，在跨页过程中手动调用。

socket通信，以及长连接，分包，连接异常断开的处理。

Socket TCP/IP协议数据传输过程中的粘包和分包问题

一：通过图解法来描述一下分包和粘包，这样客户更清晰直观的了解：



下面对上面的图进行解释：
1.正常情况：如果Socket Client 发送的数据包，在Socket Server端也是一个一个完整接收的，那个就不会出现粘包和分包情况，数据正常读取。
2.粘包情况：Socket Client发送的数据包，在客户端发送和服务器接收的情况下都有可能发送，因为客户端发送的数据都是发送的一个缓冲buffer，然后由缓冲buffer最后刷到数据链路层的，那么就有可能把数据包2的一部分数据结合数据包1的全部被一起发送出去了，这样在服务器端就有可能出现这样的情况，导致读取的数据包包含了数据包2的一部分数据，这就产生粘包，当然也有可能把数据包1和数据包2全部读取出来。
3.分包情况：意思就是把数据包2或者数据包1都有可能被分开一部分发送出去，接着发另外的部分，在服务器端有可能一次读取操作只读到一个完整数据包的一部分。
4.在数据包发送的情况下，有可能后面的数据包分开成2个或者多个，但是最前面的部分包，黏住在前面的一个完整或者部分包的后面，也就是粘包和分包同时产生了。

二：产生上情况的内部原因有下面几点：
1.数据发送端发送数据给缓冲buffer太大，导致发送一个完整的数据包被分几次发送给缓存buffer，然而缓冲buffer等到数据满了以后会自动把数据发送的数据链路层去，这样就导致分包了。
2.TCP协议定义有一个选项叫做最大报文段长度（MSS，Maximum Segment Size），该选项用于在TCP连接建立时，收发双方协商通信时每一个报文段所能承载的最大数据长度。在一定程度上MSS应该能尽可能多地承载用户数据，用于在传输通路上又可能避免分片，但是在复杂的网络环境下确定这个长度值非常困难，那么在这样的情况下在传输过程中产生分包，粘包就很常见了
3.以太网，IP,TCP,UDP数据包分析
4.数据帧的有效载荷(payload)比以太网的最大传输单元（MTU）大的时候，进行了IP分片。

三：解决数据分包和粘包的基本策略如下
1.消息定长，比如定一个100，那么读取端每次读取数据就截取100个长度的数据，然后交给业务成去做解析
2.在消息的尾部加一些特殊字符，那么在读取数据的时候，只要读到这个特殊字符，就认为已经可以截取一个完整的数据包了，这种情况在一定的业务情况下实用。
3.读取缓存的数据是不定长的，所以我们把读取到的数据添加到我们自己的一个byte[]数组中，然后根据我们的业务逻辑来找到指定的特殊协议头部，协议长度，协议尾部，然后从我们的byte[]中获取一个完整的数据包，然后再对数据包进行业务解析就可以得到正确结果。

深入理解socket网络异常

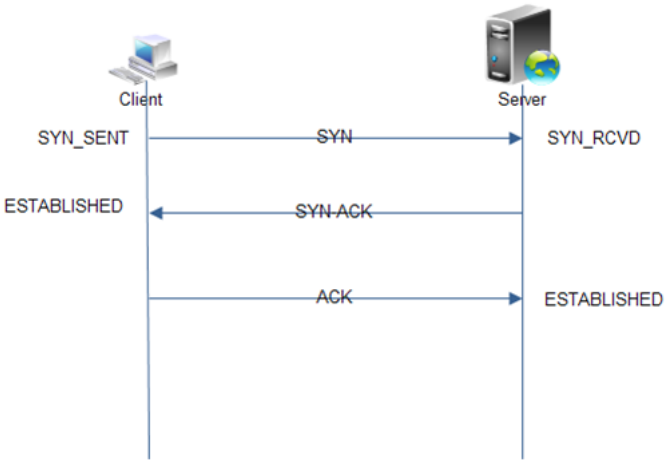
在各种网络异常情况的背后，TCP是怎么处理的？又是怎样把处理结果反馈给上层应用的？本文就来讨论这个问题。

分为两个场景来讨论

建立连接

1 正常情况下

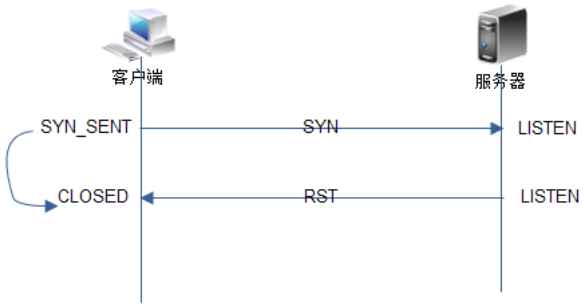
进过三次握手，客户端连接成功，服务端有一个新连接到来。



2 客户端连接了服务端未监听的端口

在这种情况下，服务端会对收到的SYN回应一个RST（RFC 793 3.4），客户端收到RST之后，终止连接，并进入CLOSED状态。

客户端的connect返回ECONNREFUSED 111 /* Connection refused */。



3 客户端与服务器之间的网络不通

这又分两种情况

1. connect返回主机不可达
   具体信息在不同系统上不一样，比如linux上的定义是EHOSTUNREACH 113 /* No route to host */。明显给出了一个不可访问的地址（例如，访问一个不存在的本地网络地址，或者DNS解析失败会导致这种情况。
2. connect返回连接超时
   这种情况下，客户端发送的SYN丢失在网络中，没有得到确认，客户端的TCP会超时重发SYN。以ubuntu 12.04为例，重发SYN的时间,系列是：0,1,3,7,15,31,63(2n-1-1)。即发送7个SYN后等待一个超时时间（例如：127秒），如果在这段时间内仍然没有收到ACK，则connect返回超时。

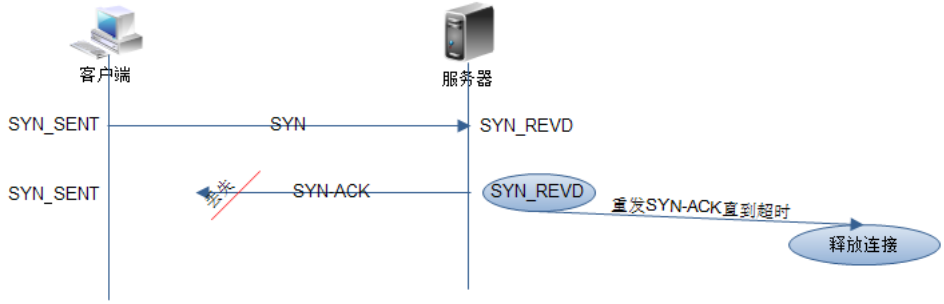
在这两种情况下， 服务端的状态没有变化，对服务端来讲什么也没发生。

4 建立连接的过程中包丢失

三次握手发送的包系列是SYN > SYN-ACK > ACK

1. SYN丢失
   这种情况就是3种的第2种情况。
2. SYN-ACK丢失
   从客户端的角度来讲以前面一种情况类似。从服务端的角度来讲，由LISTEN状态进入SYN_REVD状态。服务端的TCP会重发SYN-ACK，直到超时。SYN攻击正是利用这一原理，攻击方伪造大量的SYN包发送到服务器，服务器对收到的SYN包不断回应SYN-ACK，直到超时。这会浪费服务

器大量的资源，甚至导致奔溃。对服务端的应用层来讲，什么也没有发生。因为TCP只有在经过3次握手之后才回通知应用层，有新的连接到来。



3. ACK丢失
   这对服务端来讲与2相同。对于客户端来讲，由SYN_SENT状态进入了ESTABLISED状态，即连接成功了。连接成功后客户端就可以发送数据了。但实际上数据是发送不到服务端的（我们假设客户端收到SYN-ACK之后，客户端与服务端之间的网络就断开了），客户端发送出去的数据得不到确认，一般重发3次左右就会处于等待ACK的状态（win7）。而ubuntu 12.10下，调用send会返回成功，直到TCP的缓冲被填满（测试环境：局域网，感觉这个不是很合理，按照书上所说：应该是使用"指数退避"进行重传 -- TCP/IP协议详解， 大概是我的测试环境中有NAT所致吧）。最终，客户端产生一个复位信号并终止连接。返回给应用程序的结果是Connection time out（errno: 110）

通信过程

下面来看看连接建立成功后出现异常情况的处理

1 客户端与服务器的网络断开，双方不再发送数据

这样，双方都不知道网络已经不通，会一直保持ESTABLISHDED状态，除非打开了SO_KEEPALIVE选项。

2 网络断开，一方给另一方发送数据

这种情况下，接收一方不知道网络出问题，会一直等待数据到来。

对于发送方，理论上的情况是，重传一定次数后，返回连接超时。

不过实际，很可能是这样的情况，发送方显示发送数据成功（send返回发送的数据长度），但实际接收方还没有接收到数据。

对于已经发送成功的数据有3种可能情况：

1 在本机的TCP缓存中

2 在网络上的某个NAT的缓存中

3 对方已经成功接收到

在实验的过程中发现，即使网络断开了，发送方仍然收到了对数据的ACK（在有NAT的情况下），猜测是NAT把数据缓存起来并发送了ACK。

当网络恢复时，那些被缓存的数据会被发送到接收方。鉴于这样的结果，给我们一个提示：不能依赖于TCP的可靠性，认为我发送成功的数据，对方一定能收到。

TCP可以保证可靠、有序的传输，这意思是说保证收到的数据时有序正确的，并没有说已经发送成功的数据，对方一定就收到了。

在ubuntu 12.10上，发送方一直在发送数据，直到缓冲区满。而在win7下，重发3次就会停止，进入等待ACK状态。

解决的办法是：应用层对数据是否接收完成进行确认（需要的时候）。

3 网络断开，一方等待着另一方发送数据

这种情况下，等待数据的一方将一直等待下去。接收方无法直接知道网络已经断开，一般是设置一个超时时间，超时时间到就判断为网络已断开。发送数据的一方的反应如2所述。

4 一方crash，另一方继续发送/接收数据

这依赖于TCP协议栈对crash的反应。与系统相关性很大 例如：

在windows下：按ctrl+c结束程序，会发送RST段。而在linux下，按ctrl+c结束程序，会调用close。

在wind7下，如果没有调用close而结束程序，TCP会发送RST。而Ubuntu12.10上，则会发送FIN段。

1. crash的一端发送FIN，相当于调用了close
   没有crash的一端接收数据
   具体的反应与系统有关
   例如linux 3.8.0-29-generic调用recv返回-1，errno被设置为22， Invalid argument，而linux3.3.6-030306-generic调用recv返回0.
   在TCP内部，调用recv时，发送FIN，终止连接(Linux)。
   windows情况以此不同，recv返回0，表示对方调用了shutdown。TCP内部发送一个RST。

   但共同点是recv都会立即返回失败。
   没有crash的一端发送数据
   第一次调用send返回成功，数据会被发送到crash的一端，crash的一端会回应一个RST，再次调用send返回-1， errno被设置为32，Broken pipe。 注意：这会向应用程序发送SIGPIPE信号，你的程序会莫名其妙退出。这是因为程序对SIGPIPE的默认处理就是结束程序。
   这是编写服务器程序是最需要注意的一个问题。最简单的处理方法是忽略该信号 -- signal(SIGPIPE,SIG_IGN);
   windows下行为是一样的， 不同的是返回的错误是10053 - WSAECONNABORTED， 由于软件错误，造成一个已经建立的连接被取消。

   共同点第一次send成功，之后就出错。

2. crash的一端发送RST
   没有crash的一端接收数据
   调用recv返回-1，errno被设置为104， Connection reset by peer。在TCP内部，当收到RST时，把错误号设为ECONNRESET。
   没有crash的一端发送数据
   调用send返回-1，errno被设置为104， Connection reset by peer。在TCP内部，当收到RST时，把错误号设为ECONNRESET

3. crash的一端即没发送FIN也没发送RST
   没有crash的一端接收数据
   调用recv会一直阻塞等待数据到来
   没有crash的一端发送数据
   重传一定次数后，返回connection time out。

5 一端关闭连接

这种情况与一端crash并发送FIN 的情况相同，参看4.1

总结

上面分析的目的是：当程序出现网络异常时，能够知道问题的原因在哪?

作为开发者，我们主要关心应用层面的返回状态。一般出错的地方是调用connect, recv, send的时候。

下面做一个总结

connect函数返回状态及其原因

返回状态	可能原因
ECONNREFUSED	连接的端口不对；服务器不提供该服务（端口扫描）；
ETIMEDOUT	服务器繁忙；网络不通；网络很差，导致三次握手中包丢失；
ENETUNREACH	连接地址不对；网络不同；

recv函数返回状态及其原因

返回状态	ermo	可能原因
0	-	对方调用了shutdown(), 或close()。对方crash并发送了RST或FIN；总之表示对方不会再发送数据了。
-1	EAGIN	数据未到达；网络已断开；对方已奔溃；
-1	ECONNRESET	对方crash；
-1	EINVAL	对方关闭连接；

send函数返回状态及其原因

返回状态	errno	可能原因
0	-	发送数据长度为0；
-1	EAGIN	发送缓存已满；
-1	EPIPE	对方已经关闭连接；对方crash；
-1	ECONNRESET	对方crash；
-1	EINVAL	对方关闭连接；
-1	ETIMEDOUT	网络断开；对方崩溃即没发送FIN也没发送RST；

各种不同步的状态，都是通过发送RST来恢复的，理解这些状况的关键在于理解何时产生RST，以及在各种状态下，对RST段如何处理。

http中，get post的区别

- get： 从服务器上获取数据，也就是所谓的查，仅仅是获取服务器资源，不进行修改。
- post: 向服务器提交数据，这就涉及到了数据的更新，也就是更改服务器的数据。
- 请求方式的区别: get 请求的数据会附加在URL之后,特定的浏览器和服务器对URL的长度有限制. post 更加安全数据不会暴漏在url上,而且长度没有限制.

HTTP报文内容

HTTP请求报文和HTTP响应报文

简介:

HTTP报文是面向文本的，报文中的每一个字段都是一些ASCII码串，各个字段的长度是不确定的。HTTP有两类报文：请求报文和响应报文。

HTTP请求报文

一个HTTP请求报文由请求行（request line）、请求头部（header）、空行和请求数据4个部分组成，下面给出了请求报文的一般格式。

```
<request-line>
<headers>
<blank line>
[<request-body>
```

1.请求头

请求行由请求方法字段、URL字段和HTTP协议版本字段3个字段组成，它们用空格分隔。例如，GET /index.html HTTP/1.1。

HTTP协议的请求方法有GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT。

而常见的有如下几种：

1).GET

最常见的一种请求方式，当客户端要从服务器中读取文档时，当点击网页上的链接或者通过在浏览器的地址栏输入网址来浏览网页的，使用的都是GET方式。GET方法**要求服务器将URL定位的资源放在响应报文的数据部分，回送给客户端。**使用GET方法时，请求参数和对应的值附加在URL后面，利用一个问号（“?”）代表URL的结尾与请求参数的开始，**传递参数长度受限制。**例如，/index.jsp?id=100&op=bind,这样通过GET方式传递的数据直接表示在地址中，所以我们可以把请求结果以链接的形式发送给好友。以用google搜索domety为例，Request格式如下：

```
GET /search?hl=zh-CN&source=hp&q=domety&aq=f&oq= HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-silverlight, application/x-shockwave-flash, */*
Referer: <a href="http://www.google.cn/">http://www.google.cn/</a>
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; TheWorld)
Host: <a href="http://www.google.cn">www.google.cn</a>
Connection: Keep-Alive
Cookie: PREF=ID=80a06da87be9ae3c:U=f7167333e2c3b714:NW=1:TM=1261551909:LM=1261551917:S=ybYcq2wpfefs4V9g;
NID=31=ojj8d-IygaEtSxLgaJmqSjVhCspkviJrB6omjamNrSm8lZhKy_yMfO2M4QMRKcH1g0iQv9u-2hfBW7bUFwVh7pGaRUb0RnHcJU37y-
FxlRugatx63JLv7CWMD6UB_O_r
```

可以看到，GET方式的请求一般不包含“请求内容”部分，请求数据以地址的形式表现在请求行。地址链接如下：

```
<a href="http://www.google.cn/search?hl=zh-CN&source=hp&q=domety&aq=f&oq=">http://www.google.cn/search?hl=zh-CN&source=hp
&q=domety&aq=f&oq=</a>
```

地址中“?”之后的部分就是通过GET发送的请求数据，我们可以在地址栏中清楚的看到，各个数据之间用“&”符号隔开。显然，这种方式**不适合传送私密数据。**另外，由于不同的浏览器对地址的字符限制也有所不同，一般最多只能识别1024个字符，所以如果**需要传送大量数据的时候，也不适合使用GET方式。**

2).POST

对于上面提到的不适合使用GET方式的情况，可以考虑使用POST方式，因为使用POST方法可以允许客户端给服务器提供信息较多。POST方法将请求参数封装在HTTP请求数据中，以名称/值的形式出现，可以传输大量数据，这样POST方式对传送的数据大小没有限制，而且也不会显示在URL中。还以上面的搜索domety为例，如果使用POST方式的话，格式如下：

```
POST /search HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-silverlight, application/x-shockwave-flash, */*
Referer: <a href="http://www.google.cn/">http://www.google.cn/</a>
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; TheWorld)
Host: <a href="http://www.google.cn">www.google.cn</a>
Connection: Keep-Alive
Cookie: PREF=ID=80a06da87be9ae3c:U=f7167333e2c3b714:NW=1:TM=1261551909:LM=1261551917:S=ybYcq2wpfefs4V9g;
NID=31=ojj8d-IygaEtSxLgaJmqSjVhCspkviJrB6omjamNrSm8lZhKy_yMfO2M4QMRKcH1g0iQv9u-2hfBW7bUFwVh7pGaRUb0RnHcJU37y-
FxlRugatx63JLv7CWMD6UB_O_r

hl=zh-CN&source=hp&q=domety
```

可以看到，POST方式请求行中不包含数据字符串，这些数据保存在“请求内容”部分，各数据之间也是使用“&”符号隔开。POST方式大多用于页面的表单中。因为POST也能完成GET的功能，因此多数人在设计表单的时候一律都使用POST方式，其实这是一个误区。GET方式也有自己的特点和优势，我们应该根据不同的情况来选择是使用GET还是使用POST。

3).HEAD

HEAD就像GET，只不过服务端接受到HEAD请求后只返回响应头，而不会发送响应内容。当我们只需要查看某个页面的状态的时候，使用HEAD是非常高效的，因为在传输的过程中省去了页面内容。

2.请求头部

请求头部由**关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔。**请求头部通知服务器有关于客户端请求的信息，典型的请求头有：

User-Agent：产生请求的浏览器类型。

Accept：客户端可识别的内容类型列表。

Host：请求的主机名，允许多个域名同处一个IP地址，即虚拟主机。

3.空行

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

4.请求数据

请求数据不在GET方法中使用，而是在POST方法中使用。POST方法适用于需要客户填写表单的场合。与请求数据相关的最常使用的请求头是Content-Type和Content-Length。

HTTP报文

HTTP响应也由三个部分组成，分别是：**状态行、消息报头、响应正文。**

如下所示，HTTP响应的格式与请求的格式十分类似：

```
<status-line>
<headers>
<blank line>
[<response-body>]
```

正如你所见，在响应中唯一真正的区别在于第一行中用状态信息代替了请求信息。状态行（status line）通过提供一个状态码来说明所请求的资源情况。

状态行格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。状态代码由三位数字组成，第一个数字定义了响应的类别，且有五种可能取值。

- 1xx：指示信息--表示请求已接收，继续处理。
- 2xx：成功--表示请求已被成功接收、理解、接受。
- 3xx：重定向--要完成请求必须进行更进一步的操作。
- 4xx：客户端错误--请求有语法错误或请求无法实现。
- 5xx：服务器端错误--服务器未能实现合法的请求。

常见状态代码、状态描述的说明如下。

- 200 OK：客户端请求成功。
- 400 Bad Request：客户端请求有语法错误，不能被服务器所理解。
- 401 Unauthorized：请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用。
- 403 Forbidden：服务器收到请求，但是拒绝提供服务。
- 404 Not Found：请求资源不存在，举个例子：输入了错误的URL。
- 500 Internal Server Error：服务器发生不可预期的错误。
- 503 Server Unavailable：服务器当前不能处理客户端的请求，一段时间后可能恢复正常，举个例子：HTTP/1.1 200 OK（CRLF）。

下面给出一个HTTP响应报文例子

```
HTTP/1.1 200 OK
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 122

<html>
<head>
<title>Wrox Homepage</title>
</head>
<body>
<!-- body goes here -->
</body>
</html>
```

关于HTTP请求GET和POST的区别

1.GET提交，请求的数据会附在URL之后（就是把数据放置在HTTP协议头 < request-line > 中）

以?分割URL和传输数据，多个参数用&连接;例如：login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0 %E5%A5%BD。如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如： %E4%BD%A0%E5%A5%BD，其中%XX中的XX为该符号以16进制表示的ASCII。

POST提交：**把提交的数据放置在是HTTP包的包体 < request-body > 中。**

上文示例中加粗字体标明的就是实际的传输数据

因此，**GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变**

2.传输数据的大小：

首先声明,**HTTP协议没有对传输的数据大小进行限制，HTTP协议规范也没有对URL长度进行限制。** 而在实际开发中存在的限制主要有：

- GET:**特定浏览器和服务器对URL长度有限制**，例如IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。

  因此对于GET提交时，传输数据就会受到URL长度的限制。
- POST:由于不是通过URL传值，理论上数据不受限。但实际各个WEB服务器会规定对post提交数据大小进行限制，Apache、IIS6都有各自的配置。

3.安全性：

POST的安全性要比GET的安全性高。注意：这里所说的安全性和上面GET提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的Security的含义，比如：通过GET提交数据，用户名和密码将明文出现在URL上，因为(1)登录页面有可能被浏览器缓存， (2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了。

说说浏览器访问http://www.taobao.com，经历了怎样的过程。

访问 www.taobao.com过程

首先是查找浏览器缓存，浏览器会保存一段时间你之前访问过的一些网址的DNS信息，不同浏览器保存的时常不等。

如果没有找到对应的记录，这个时候浏览器会尝试调用系统缓存来继续查找这个网址的对应DNS信息。

如果还是没找到对应的IP，那么接着会发送一个请求到路由器上，然后路由器在自己的路由器缓存上查找记录，路由器一般也存有DNS信息。

如果还是没有，这个请求就会被发送到ISP（注：Internet Service Provider，互联网服务提供商，就是那些拉网线到你家里的运营商，中国电信中国移动什么的），ISP也会有相应的ISP DNS服务器，一听中国电信就知道这个DNS服务器的规模肯定不会小，所以基本上都能在这里找得到。题外话：会跑到这里进行查询是因为你没有改动过"网络中心"的"ipv4"的DNS地址，万恶的电信联通可以改动了这个DNS服务器，换句话说他们可以让你的浏览器跳转到他们设定的页面上，这也就是人尽皆知的DNS和HTTP劫持，ISP们还美名曰“免费推送服务”。强烈鄙视这种霸王行为。我们也可以自行修改DNS服务器来防止DNS被ISP污染。

如果还是没有的话， 你的ISP的DNS服务器会将请求发向根域名服务器进行搜索。根域名服务器就是面向全球的顶级DNS服务器，共有13台逻辑上的服务器，从A到M命名，真正的实体服务器则有几百台，分布于全球各大洲。所以这些服务器有真正完整的DNS数据库。如果到了这里还是找不到域名的对应信息，那只能说明一个问题：这个域名本来就不存在，它没有在网上正式注册过。或者卖域名的把它回收掉了（通常是因为欠费）。

这也就是为什么打开一个新页面会有点慢，因为本地没什么缓存，要这样递归地查询下去。

多说一句，例如"mp3.baidu.com"，域名先是解析出这是个.com的域名，然后跑到管理.com域名的服务器上进行进一步查询，然后是.baidu，最后是mp3，

所以域名结构为：三级域名.二级域名.一级域名。

浏览器终于得到了IP以后，浏览器接着给这个IP的服务器发送了一个http请求，方式为get，例如访问nbut.cn

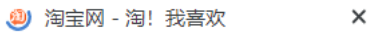


这个get请求包含了主机（host）、用户代理(User-Agent)，用户代理就是自己的浏览器，它是你的"代理人"，Connection（连接属性）中的keep-alive表示浏览器告诉对方服务器在传输完现在请求的内容后不要断开连接，不断开的话下次继续连接速度就很快了。其他的顾名思义就行了。还有一个重点是Cookies，Cookies保存了用户的登陆信息，在每次向服务器发送请求的时候会重复发送给服务器。Corome上的F12与Firefox上的firebug(快捷键shift+F5)均可查看这些信息。

发送完请求接下来就是等待回应了

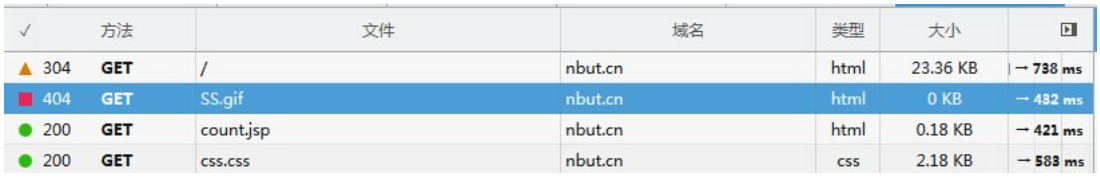
当然了，服务器收到浏览器的请求以后（其实是WEB服务器接收到了这个请求，WEB服务器有iis、apache等），它会解析这个请求（读请求头），然后生成一个响应头和具体响应内容。接着服务器会传回来一个响应头和一个响应，响应头告诉了浏览器一些必要的信息，例如重要的Status Code，2开头如200表示一切正常，3开头表示重定向，4开头，如404，呵呵。响应就是具体的页面编码，就是那个……，浏览器先读了关于这个响应的说明书（响应头），然后开始解析这个响应并在页面上显示出来。在下一次CF的时候（不是穿越火线，是http://codeforces.com/），由于经常难以承受几千人的同时访问，所以CF页面经常会出现崩溃页面，到时候可以点开火狐的firebug或是Chrome的F12看看状态，不过这时候一般都急着看题和提交代码，似乎根本就没心情理会这个状态吧-.-。

如果是个静态页面，那么基本上到这一步就没了，但是如今的网站几乎没有静态的了吧，基本全是动态的。所以这时候事情还没完，根据我们的经验，浏览器打开一个网址的时候会慢慢加载这个页面，一部分一部分的显示，直到完全显示，最后标签栏上的圈圈就不转了。





这是因为，主页（index）页面框架传送过来以后，浏览器还要继续向服务器发送请求，请求的内容是主页里面包含的一些资源，如图片，视频，css样式等等。这些"非静态"的东西要一点点地请求过来，所以标签栏转啊转，内容刷啊刷，最后全部请求并加载好了就终于好了。



✓	方法	文件	域名	类型	大小	
▲ 304	GET	/	nbut.cn	html	23.36 KB	→ 738 ms
■ 404	GET	SS.gif	nbut.cn	html	0 KB	→ 432 ms
● 200	GET	count.jsp	nbut.cn	html	0.18 KB	→ 421 ms
● 200	GET	css.css	nbut.cn	css	2.18 KB	→ 583 ms

需要说明的是，对于静态的页面内容，浏览器通常会进行缓存，而对于动态的内容，浏览器通常不会进行缓存。缓存的内容通常也不会保存很久，因为难保网站不会被改动。

HTTP协议、 HTTPS协议，SSL协议及完整交互过程；

HTTPS协议，SSL协议及完整交互过程

SSL

1. 安全套接字（Secure Socket Layer，SSL）协议是Web浏览器与Web服务器之间安全交换信息的协议。
2. SSL协议的三个特性

Ø 保密：在握手协议中定义了会话密钥后，所有的消息都被加密。

Ø 鉴别：可选的客户端认证，和强制的服务器端认证。

Ø 完整性：传送的消息包括消息完整性检查（使用MAC）。

3. SSL的位置

HTTPS

1. HTTPS基于SSL的HTTP协议。
2. HTTPS使用与HTTP不同的端口(，一个加密、身份验证层（HTTP与TCP之间））。
3. 提供了身份验证与加密通信方法，被广泛用于互联网上安全敏感的通信。

交互过程

客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤

1) 客户端请求建立SSL连接，并将自己支持的一套加密规则发送给网站。

2) 网站从中选出一组加密算法与HASH算法，并将自己的身份信息以证书的形式发回给浏览器。证书里面包含了网站地址，加密公钥，以及证书的颁发机构等信息

3) 获得网站证书之后浏览器要做以下工作：

Ø 验证证书的合法性

Ø 如果证书受信任，浏览器会生成一串随机数的密码，并用证书中提供的公钥加密。

Ø 使用约定好的HASH***计算握手消息***，

Ø 使用生成的随机数**对消息进行加密**，最后将之前生成的所有信息发送给网站。

4) 网站接收浏览器发来的数据之后要做以下的操作：

Ø 使用自己的私钥将信息解密**取出密码**

Ø 使用密码**解密**浏览器发来的握手消息，并验证HASH是否与浏览器发来的一致。

Ø ***使用密码加密***一段握手消息，发送给浏览器

5) 浏览器解密并计算握手消息的***HASH***，如果与服务端发来的HASH一致，此时握手结束。

6) 使用**随机密码**和**对称加密算法**对传输的数据加密，传输。

4. 密与HASH算法如下：

1) 非对称加密算法：RSA，DSA/DSS，用于在握手过程中加密生成的密码。

2) 对称加密算法：AES，RC4，3DES，用于对真正传输的数据进行加密。

3) HASH算法：MD5，SHA1，SHA256，验证数据的完整性。

5. HTTP与HTTPS的区别：

1) https协议需要申请证书。

2) http是超文本传输协议，明文传输；https使用的是具有安全性的SSL加密传输协议。

3) http端口80,；https端口443。

4) http连接简单无状态；https由SSL+HTTP协议构件的可进行加密传输、身份验证的网络协议。

Ping过程 原理 详解

如果你想了解PING的原理，就看我的文章，不要去网上找，找不到什么好的内容。看了我文章，也许你会从对网络一窍不通，到豁然开朗。

我在这里讲拼的两情况,一种是同一网段内,一种是跨网段的ping ....

首先，如果主机A，要去 ping主机B，那么主机A，就要封装二层报文，他会先查自己的MAC地址表，如果没有B的MAC地址，就会向外发送一个ARP广播包

其中ARP报文格式如下:

以太网目的MAC	以太网源MAC	帧类型	硬件类型	4	6	OP	发送端以太网MAC	发送端IP地址	目的MAC	目的IP
FF-FF-FF-FF-FF-FF	00-50-56-C0-00-01	0806	0800	**1**	00-50-56-C0-00-01	1.1.1.1	00-00-00-00-00-00	1.1.1.3		

其中OP

1 :表示ARP请求

2:表示ARP应答

```
***\*3:\**表示RARP请求**

***\*4:\**表示RARP应答**
```

首先,交换机会收到这个报文后，交换机有学习MAC地址的功能，所以他会检索自己有没有保存主机B有MAC，如果有，就返回给主机A， 如果没有，就会向所有端口发送ARP广播 ，其它主机收到后，发现不是在找自己，就纷纷丢弃了该报文，不去理会。。直到主机B收到了报文后，就立即响应，我的MAC地址是多少，同时学到主机A的MAC地址,并按同样的ARP报文格式返回给主机A

ARP报文格式:

以太网目的MAC	以太网源MAC	帧类型	硬件类型	4	6	OP	发送端以太网MAC	发送端IP地址	目的MAC	目的IP
00-50-56-C0-00-01	00-50-56-C0-00-03	0806	0800	2	00-50-56-C0-00-03	1.1.1.3	00-50-56-C0-00-01	1.1.1.1		

这时候主机A学到了主机B的MAC，就把这个MAC封装到ICMP协议的二层报文中向主机B发送,报文格式如下:

目的地址	源地址	...	源IP	目的IP	...	ICMP报文
00-50-56-C0-00-03	00-50-56-C0-00-01	1.1.1.1	1.1.1.3	Echo request		

当主机B收到了这个报文后，发现是主机A 的ICPM回显请求，就按同样的格式，返回一个值给主机A，这样就完成了同一网段内的ping过程...

目的地址	源地址	...	源IP	目的IP	...	ICMP报文
00-50-56-C0-00-01	00-50-56-C0-00-03	1.1.1.3	1.1.1.1	Echo answer		

在这里，我讲了这么久的 ***局域网内的PING*** ,实际过程的发生不到 **1毫秒....**

```
再继续...

如果主机A要ping主机C,那么主机A发现主机C的IP和自己不是同一网段,他就去找 ***\*网关\**** 转发,但是他也不知道网关的MAC情况下呢?他就会向之前那个步骤一样 **先发送一个ARP广播,学到网关的MAC,再发封装ICMP报文给网关路由器.**
```

报文格式如下:

目的地址	源地址	...	源IP	目的IP	...	ICMP报文
00-50-56-C0-00-02	00-50-56-C0-00-01	1.1.1.1	2.1.1.1	Echo request		

当路由器收到主机A发过来的ICMP报文,发现自己的目的地址是其本身MAC地址,根据目的的IP2.1.1.1,查路由表,发现2.1.1.1/24的路由表项,得到一个出口指针,去掉原来的MAC头部.加上自己的MAC地址向主机C转发...(**如果网关也没有主机C的MAC地址,还是要向前面一个步骤一样,ARP广播一下即可相互学到....路由器2端口能学到主机D的MAC,主机D也能学到路由器2端口的MAC.** .) ,报文格式如下:

目的地址	源地址	...	源IP	目的IP	...	ICMP报文
00-50-56-C0-00-05	00-50-56-C0-00-04	1.1.1.1	2.1.1.1	Echo request		

最后,在主机C已学到路由器2端口MAC,路由器2端口转发给路由器1端口,路由1端口学到主机A的MAC的情况下,他们就不需要再做ARP解析,就将ICMP的回显请求回复过来.. 报文格式大致如下:

目的地址	源地址	...	源IP	目的IP	...	ICMP报文
00-50-56-C0-00-04	00-50-56-C0-00-05	2.1.1.1	1.1.1.1	Echo Answer		

TCP/IP协议详解笔记——ARP协议和RARP协议

ARP：地址解析协议

对于以太网，数据链路层上是根据48bit的以太网地址来确定目的接口，设备驱动程序从不检查IP数据报中的目的IP地址。ARP协议为IP地址到对应的硬件地址之间提供动态映射。

工作过程

在以太网（ARP协议只适用于局域网）中，如果本地主机想要向某一个IP地址的主机（路由表中的下一跳路由器或者直连的主机，注意此处IP地址不一定是IP数据报中的目的IP）发包，但是并不知道其硬件地址，此时利用ARP协议提供的机制来获取硬件地址，具体过程如下：

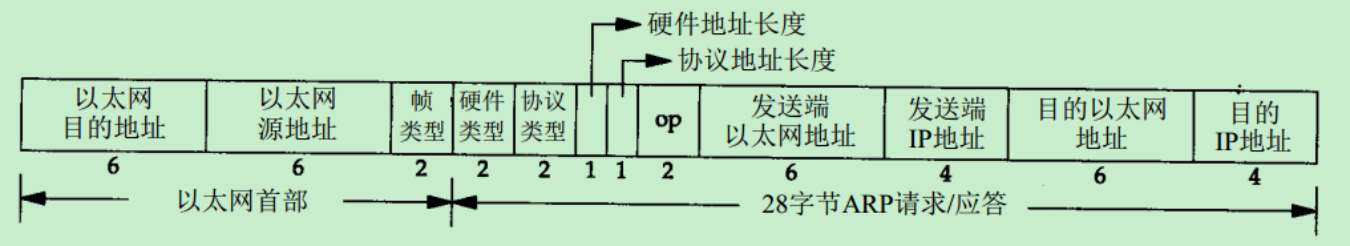
1) 本地主机在局域网中广播ARP请求，ARP请求数据帧中包含目的主机的IP地址。意思是“如果你是这个IP地址的拥有者，请回答你的硬件地址”。

2) 目的主机的ARP层解析这份广播报文，识别出是询问其硬件地址。于是发送ARP应答包，里面包含IP地址及其对应的硬件地址。

3) 本地主机收到ARP应答后，知道了目的地址的硬件地址，之后的数据报就可以传送了。

点对点链路不使用ARP协议。

帧格式



- 以太网目的地址：目的主机的硬件地址。目的地址全为1的特殊地址是广播地址。
- 以太网源地址：源主机的硬件地址。
- 帧类型：对于ARP协议，该字段为0x0806。对于RARP协议，该字段为0x8035。
- 硬件类型：表示硬件地址的类型。值为1时表示以太网地址。也就是说ARP协议不仅仅应用于以太网协议，还可以支持别的链路层协议。
- 协议类型：表示要映射的协议地址类型。值为0x0800时表示IP协议。
- 硬件地址长度：与硬件类型对应的硬件地址的长度，以字节为单位。如果是以太网，则是6字节（MAC长度）。
- 协议地址长度：与协议类型对应的协议地址长度，以字节为单位。如果是IP协议，则是4字节（IP地址长度）。
- 操作类型（op）：四中操作类型。ARP请求（1），ARP应答（2），RARP请求（3），RARP应答（4）。
- 发送端硬件地址：如果是以太网，则是源主机以太网地址，此处和以太网头中的源地址对应。
- 发送端协议地址：如果是IP协议，则表示源主机的IP地址。
- 目的端硬件地址：如果是以太网，则是目的以太网地址，和以太网头中的目的地址对应。
- 目的端协议地址：如果是IP协议，则表示源主机要请求硬件地址的IP地址。
- 对应ARP请求包来说，目的端的硬件地址字段无须填充，其他字段都需要填充。对于ARP回复包来说，所有字段都需要填充。

APR请求包是广播的，但是ARP应答帧是单播的。

以太网数据报最小长度是60字节（14字节的以太网头，不包含4字节的FCS），ARP数据包长度为42字节（14字节的以太网头和28字节的ARP数据），需要加入填充字符到以太网最小长度要求：60字节。

ARP高速缓存

每个主机都有一个ARP高速缓存表，这样避免每次发包时都需要发送ARP请求来获取硬件地址。默认老化时间是20分钟。利用arp -a命令可以查看显示系统中高速缓存的内容。

Windows下“arp -d”命令可以清除arp高速缓存表。

有时候需要手动清除arp缓存，曾经就是因为arp缓存没有做清理，导致迷惑了很久。遇到的问题：

1) 制作了一个写路由器MAC地址的工具，每次写完MAC地址，重启路由器，会发现无法telnet登陆路由器。IP地址没变，但是MAC地址更改了，而ARP缓存表中IP地址映射的仍然是旧的MAC地址。

2) 类似的问题，有两个路由器具有相同的IP地址。先连接一个路由器，登陆成功后，再去连接另一台路由器，却发现登陆不了。

ARP代理

如果ARP请求时从一个网络的主机发往另一个网络上的主机，那么连接这两个网络的路由器可以回答该请求，这个过程称作委托ARP或者ARP代理。这样可以欺骗发起ARP请求的发送端，使它误以为路由器就是目的主机。

RARP：逆地址解析协议

将局域网中某个主机的物理地址转换为IP地址，比如局域网中有一台主机只知道物理地址而不知道IP地址，那么可以通过RARP协议发出征求自身IP地址的广播请求，然后由RARP服务器负责回答。RARP协议广泛应用于无盘工作站引导时获取IP地址。

RARP允许局域网的物理机器从网管服务器ARP表或者缓存上请求其IP地址。

帧格式

帧格式同ARP协议，帧类型字段和操作类型不同，具体见ARP帧格式描述。

工作原理

1. 主机发送一个本地的RARP广播，在此广播包中，声明自己的MAC地址并且请求任何收到此请求的RARP服务器分配一个IP地址。
2. 本地网段上的RARP服务器收到此请求后，检查其RARP列表，查找该MAC地址对应的IP地址。
3. 如果存在，RARP服务器就给源主机发送一个响应数据包并将此IP地址提供给对方主机使用。
4. 如果不存在，RARP服务器对此不做任何的响应。
5. 源主机收到从RARP服务器的响应信息，就利用得到的IP地址进行通讯；如果一直没有收到RARP服务器的响应信息，表示初始化失败。

DNS域名解析的过程



一、主机解析域名的顺序

1、找缓存

2、找本机的hosts文件

3、找DNS服务器

注意：

配置IP和主机名时，要记得修改/etc/hosts文件，因为有些应用程序在主机内的进程之间通信的时候，会本机的主机名，如果主机名不能正确解析到一个正常的IP地址，那么就会导致进程通信有问题。

二、概念解释

DNS（Domain Name System，域名系统）

1、因特网上作为域名和IP(Internet Protocol Address)地址相互映射的一个分布式数据库，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。

2、通过主机名，最终得到该主机名对应的IP地址的过程叫做域名解析（或主机名解析）。

3、DNS协议运行在UDP协议之上，使用端口号53。

4、在整个互联网体系中，约定俗成的用于标识网络上设备的地址是IP，然而我们输入的是DNS，因为域名更方便人们记忆，不然那么多网站，人怎么可能记住所有的IP地址。

Q：浏览器如何通过域名去查询URL对应的IP（对应服务器地址）呢?

A：

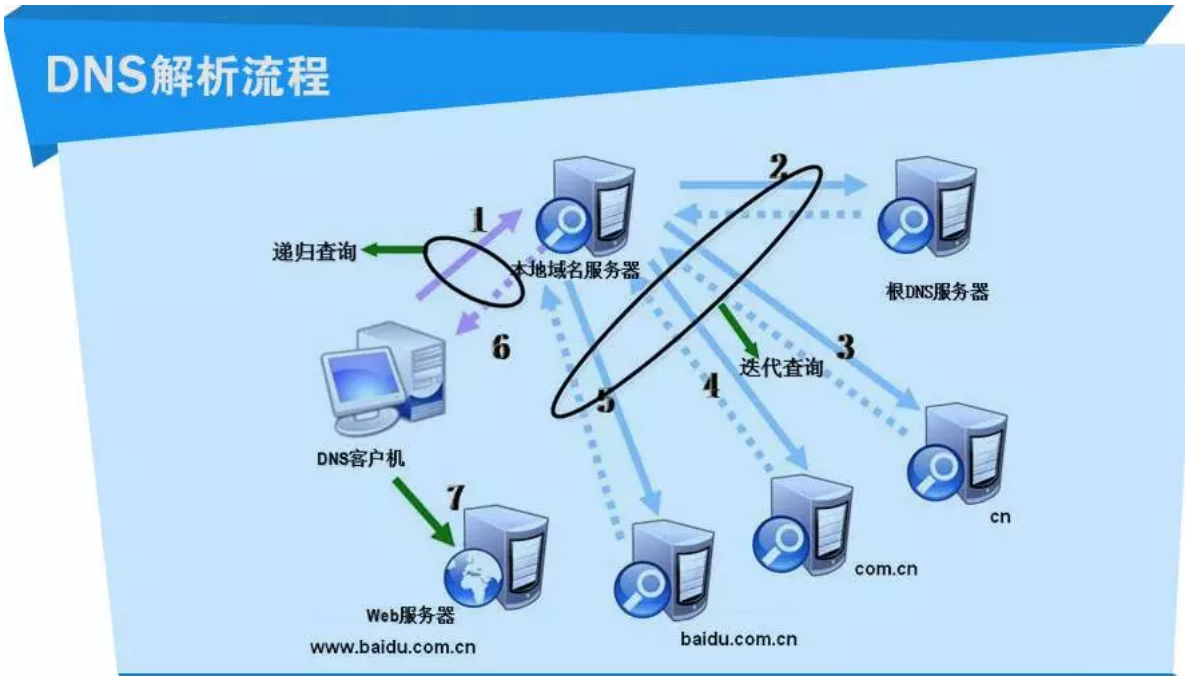
1、浏览器缓存：浏览器会按照一定的频率缓存DNS记录。

2、操作系统缓存：如果浏览器缓存中找不到需要的DNS记录，那就去操作系统中找。

3、路由缓存：路由器也有DNS缓存。

4、ISP的DNS服务器：ISP是互联网服务提供商(Internet Service Provider)的简称，ISP有专门的DNS服务器应对DNS查询请求。

5、根服务器：ISP的DNS服务器还找不到的话，它就会向根服务器发出请求，进行递归查询（DNS服务器先问根域名服务器.com域名服务器的IP地址，然后再问.com域名服务器，依次类推）。



Q：在网上查到某个网站的IP地址，在自己的浏览器上输入，却为什么连接不上？

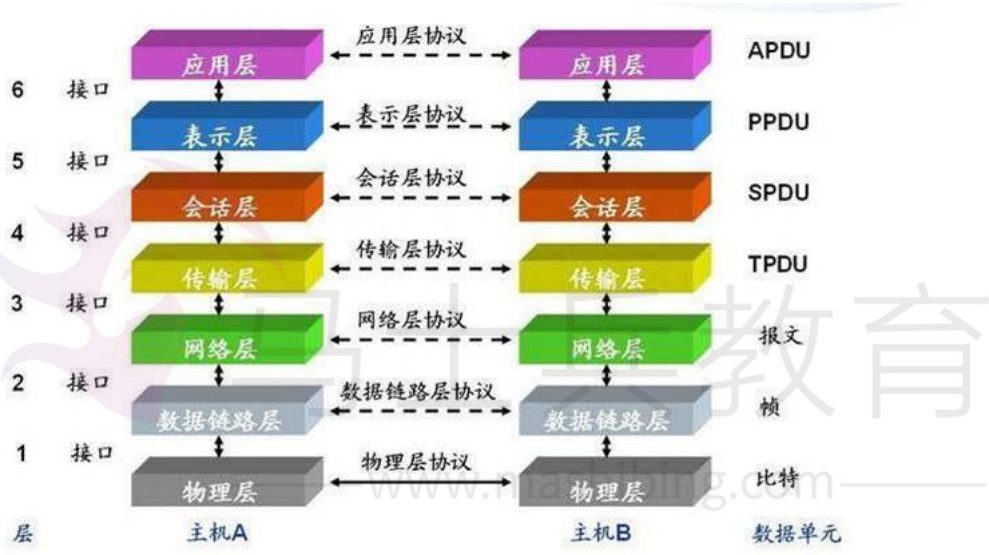
A：

大的高并发网站可能不止一个IP地址，根据不同的网络他们会有很多的IP来做集群。有的是通过DNS来实现负载均衡，有的是用squid来实现的。

Http会话的四个过程

建立连接，发送请求，返回响应，关闭连接。

网络7层架构



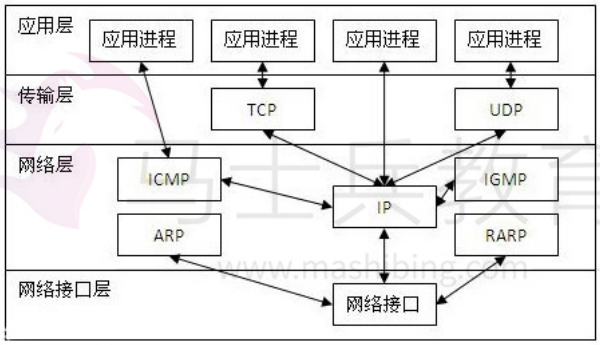
7 层模型主要包括：

1. 物理层：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由 1、0 转化为电流强弱来进行传输,到达目的地后在转化为1、0，也就是我们常说的模数转换与数模转换）。这一层的数据叫做比特。

2. 数据链路层：主要将从物理层接收的数据进行 MAC 地址（网卡的地址）的封装与解封装。常把这一层的数据叫做帧。在这一层工作的设备是交换机，数据通过交换机来传输。
3. 网络层：主要将从下层接收到的数据进行 IP 地址（例 192.168.0.1)的封装与解封装。在这一层工作的设备是路由器，常把这一层的数据叫做数据包。
4. 传输层：定义了一些传输数据的协议和端口号（WWW 端口 80 等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与 TCP 特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如 QQ 聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段进行传输，到达目的地址后在进行重组。常常把这一层数据叫做段。
5. 会话层：通过传输层（端口号：传输端口与接收端口）建立数据传输的通路。主要在你的系统之间发起会话或或者接受会话请求（设备之间需要互相认识可以是 IP 也可以是 MAC 或者是主机名）
6. 表示层：主要是进行对接收的数据进行解释、加密与解密、压缩与解压缩等（也就是把计算机能够识别的东西转换成人能够能识别的东西（如图片、声音等））
7. 应用层 主要是一些终端的应用，比如说FTP（各种文件下载），WEB（IE浏览），QQ之类的（你就把它理解成我们在电脑屏幕上可以看到的东西．就 是终端应用）。

TCP/IP 原理

TCP/IP 协议不是 TCP 和 IP 这两个协议的合称，而是指因特网整个 TCP/IP 协议族。从协议分层模型方面来讲，TCP/IP 由四个层次组成：网络接口层、网络层、传输层、应用层。



网络访问层**(Network Access Layer)**

1. 网络访问层(Network Access Layer)在 TCP/IP 参考模型中并没有详细描述，只是指出主机必须使用某种协议与网络相连。

网络层**(Internet Layer)**

2. 网络层(Internet Layer)是整个体系结构的关键部分，其功能是使主机可以把分组发往任何网络，并使分组独立地传向目标。这些分组可能经由不同的网络，到达的顺序和发送的顺序也可能不同。高层如果需要顺序收发，那么就必须自行处理对分组的排序。互联网层使用因特网协议(IP，Internet Protocol)。

传输层**(Tramsport Layer-TCP/UDP)**

3. 传输层(Tramsport Layer)使源端和目的端机器上的对等实体可以进行会话。在这一层定义了两个端到端的协议：传输控制协议(TCP，Transmission Control Protocol)和用户数据报协议(UDP，User Datagram Protocol)。TCP 是面向连接的协议，它提供可靠的报文传输和对上层应用的连接服务。为此，除了基本的数据传输外，它还有可靠性保证、流量控制、多路复用、优先权和安全性控制等功能。UDP 是面向无连接的不可靠传输的协议，主要用于不需要 TCP 的排序和流量控制等功能的应用程序。

应用层**(Application Layer)**

4. 应用层(Application Layer)包含所有的高层协议，包括：虚拟终端协议(TELNET，TELecommunications NETwork)、文件传输协议(FTP，File Transfer Protocol)、电子邮件传输协议(SMTP，Simple Mail Transfer Protocol)、域名服务(DNS，Domain Name Service)、网上新闻传输协议(NNTP，Net News Transfer Protocol)和超文本传送协议(HTTP，HyperText Transfer Protocol)等。

TCP 三次握手/四次挥手

TCP 三次握手/四次挥手

TCP 在传输之前会进行三次沟通，一般称为“三次握手”，传完数据断开的时候要进行四次沟通，一般称为“四次挥手”。

数据包说明

\1. 源端口号（ 16 位）：它（连同源主机 IP 地址）标识源主机的一个应用进程。

2. 目的端口号（ 16 位）：它（连同目的主机 IP 地址）标识目的主机的一个应用进程。这两个值加上 IP 报头中的源主机 IP 地址和目的主机 IP 地址唯一确定一个 TCP 连接。
3. 顺序号 seq（ 32 位）：用来标识从 TCP 源端向 TCP 目的端发送的数据字节流，它表示在这个报文段中的第一个数据字节的顺序号。如果将字节流看作在两个应用程序间的单向流动，则TCP 用顺序号对每个字节进行计数。序号是 32bit 的无符号数，序号到达 2 的 32 次方 - 1 后又从 0 开始。当建立一个新的连接时， SYN 标志变 1 ，顺序号字段包含由这个主机选择的该连接的初始顺序号 ISN （ Initial Sequence Number ）。
4. 确认号 ack（ 32 位）：包含发送确认的一端所期望收到的下一个顺序号。因此，确认序号应当是上次已成功收到数据字节顺序号加 1 。只有 ACK 标志为 1 时确认序号字段才有效。 TCP 为应用层提供全双工服务，这意味数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据顺序号。
5. TCP 报头长度（ 4 位）：给出报头中 32bit 字的数目，它实际上指明数据从哪里开始。需要这个值是因为任选字段的长度是可变的。这个字段占 4bit ，因此 TCP 最多有 60 字节的首部。然而，没有任选字段，正常的长度是 20 字节。
6. 保留位（ 6 位）：保留给将来使用，目前必须置为 0 。
7. 控制位（ control flags， 6 位）：在 TCP 报头中有 6 个标志比特，它们中的多个可同时被设置为 1 。依次为：

   § URG： 为 1 表示紧急指针有效，为 0 则忽略紧急指针值。

   § ACK： 为 1 表示确认号有效，为 0 表示报文中不包含确认信息，忽略确认号字段。

   § PSH： 为 1 表示是带有 PUSH 标志的数据，指示接收方应该尽快将这个报文段交给应用层而不用等待缓冲区装满。

   § RST： 用于复位由于主机崩溃或其他原因而出现错误的连接。它还可以用于拒绝非法的报文段和拒绝连接请求。一般情况下，如果收到一个 RST 为 1 的报文，那么一定发生了某些问题。

   § SYN： 同步序号，为 1 表示连接请求，用于建立连接和使顺序号同步（ synchronize ）。

   § FIN： 用于释放连接，为 1 表示发送方已经没有数据发送了，即关闭本方数据流。

8. 窗口大小（16 位）：数据字节数，表示从确认号开始，本报文的源方可以接收的字节数，即源方接收窗口大小。窗口大小是一个 16bit 字段，因而窗口大小最大为 65535 字节。
9. 校验和（16 位）：此校验和是对整个的 TCP 报文段，包括 TCP 头部和 TCP 数据，以 16 位字进行计算所得。这是一个强制性的字段，一定是由发送端计算和存储，并由接收端进行验证。
10. 紧急指针（16 位）：只有当 URG 标志置 1 时紧急指针才有效。TCP 的紧急方式是发送端向另一端发送紧急数据的一种方式。
11. 选项：最常见的可选字段是最长报文大小，又称为 MSS(Maximum Segment Size)。每个连接方通常都在通信的第一个报文段（为建立连接而设置 SYN 标志的那个段）中指明这个选项，它指明本端所能接收的最大长度的报文段。选项长度不一定是 32 位字的整数倍，所以要加填充位，使得报头长度成为整字数。
12. 数据： TCP 报文段中的数据部分是可选的。在一个连接建立和一个连接终止时，双方交换的报文段仅有 TCP 首部。如果一方没有数据要发送，也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段。

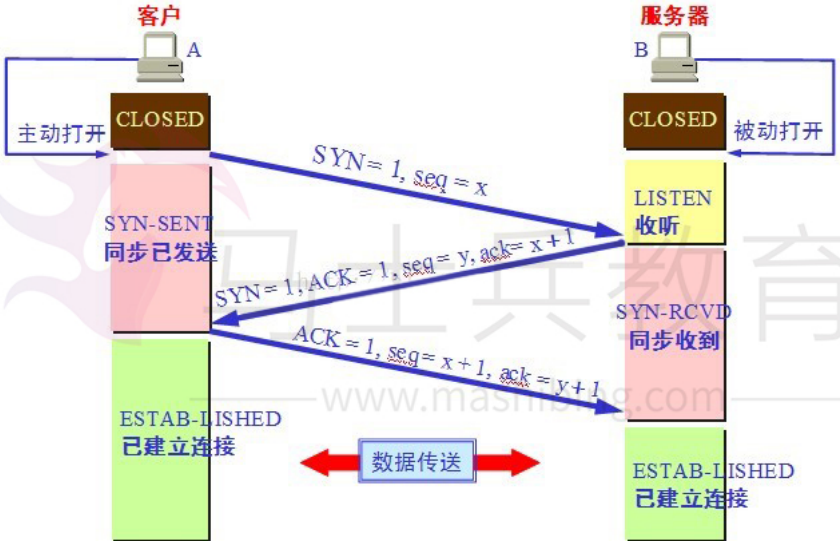


三次握手

第一次握手：主机 A 发送位码为 syn = 1,随机产生 seq number=1234567 的数据包到服务器，主机 B 由 SYN=1 知道，A 要求建立联机；

第二次握手：主机 B 收到请求后要确认联机信息，向 A 发送 ack number=(主机 A 的 seq+1),syn=1,ack=1,随机产生 seq=7654321 的包

第三次握手：主机 A 收到后检查 ack number 是否正确，即第一次发送的 seq number+1,以及位码ack 是否为 1，若正确，主机 A 会再发送 ack number=(主机 B 的 seq+1),ack=1，主机 B 收到后确认 seq 值与 ack=1 则连接建立成功。



四次挥手

TCP 建立连接要进行三次握手，而断开连接要进行四次。这是由于 TCP 的半关闭造成的。因为 TCP 连接是全双工的(即数据可在两个方向上同时传递)所以进行关闭时每个方向上都要单独进行关闭。这个单方向的关闭就叫半关闭。当一方完成它的数据发送任务，就发送一个 FIN 来向另一方通告将要终止这个方向的连接。

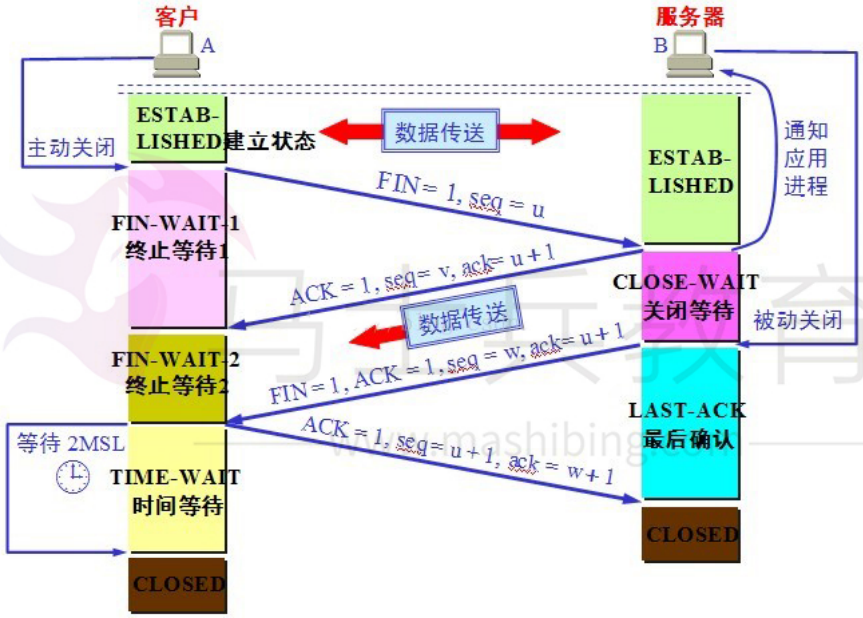
1） 关闭客户端到服务器的连接：首先客户端 A 发送一个 FIN，用来关闭客户到服务器的数据传送，然后等待服务器的确认。其中终止标志位 FIN=1，序列号 seq=u

2） 服务器收到这个 FIN，它发回一个 ACK，确认号 ack 为收到的序号加 1。

3） 关闭服务器到客户端的连接：也是发送一个 FIN 给客户端。

4） 客户段收到 FIN 后，并发回一个 ACK 报文确认，并将确认序号 seq 设置为收到序号加 1。

首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。



HTTP 原理

HTTP是一个无状态的协议。无状态是指客户机（Web浏览器）和服务器之间不需要建立持久的连接，这意味着当一个客户端向服务器端发出请求，然后服务器返回响应(response)，连接就被关闭了，在服务器端不保留连接的有关信息.HTTP 遵循请求(Request)/应答(Response)模型。客户机（浏览器）向服务器发送请求，服务器处理请求并返回适当的应答。所有 HTTP 连接都被构造成一套请求和应答。

传输流程

1：地址解析

如用客户端浏览器请求这个页面：http://localhost.com:8080/index.htm 从中分解出协议名、主机名、端口、对象路径等部分，对于我们的这个地址，解析得到的结果如下：
协议名：http

主机名：localhost.com 端口：8080

对象路径：/index.htm

在这一步，需要域名系统 DNS 解析域名 localhost.com,得主机的 IP 地址。

2：封装**HTTP**请求数据包

把以上部分结合本机自己的信息，封装成一个 HTTP 请求数据包

3：封装成**TCP**包并建立连接

封装成 TCP 包，建立 TCP 连接（TCP 的三次握手）

4：客户机发送请求命

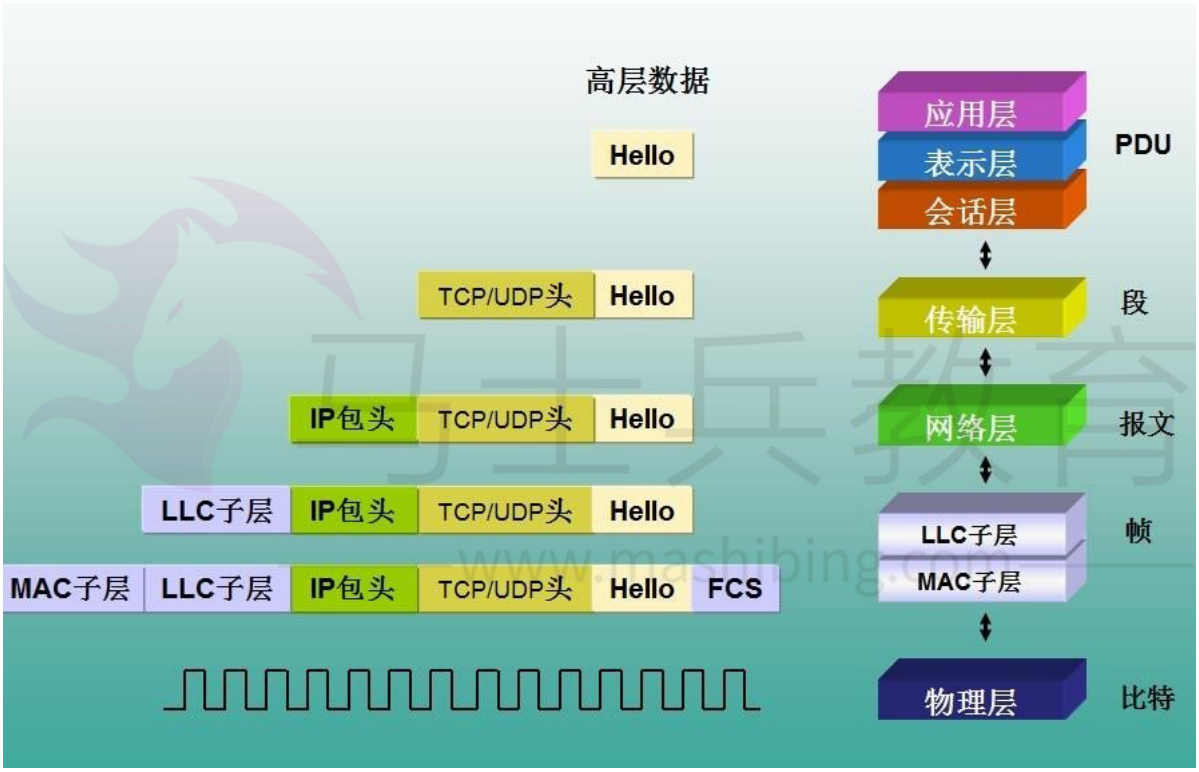
4）客户机发送请求命令：建立连接后，客户机发送一个请求给服务器，请求方式的格式为：统一资源标识符（URL）、协议版本号，后边是 MIME 信息包括请求修饰符、客户机信息和可内容。

5：服务器响应

服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息包括服务器信息、实体信息和可能的内容。

6：服务器关闭**TCP**连接

服务器关闭 TCP 连接：一般情况下，一旦 Web 服务器向浏览器发送了请求数据，它就要关闭 TCP 连接，然后如果浏览器或者服务器在其头信息加入了这行代码 Connection:keep-alive，TCP 连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。


高层数据
Hello
TCP/UDP头
Hello
IP包头
TCP/UDP头
Hello
LLC子层
IP包头
TCP/UDP头
Hello
MAC子层
LLC子层
IP包头
TCP/UDP头
Hello
FCS
应用层
表示层
会话层
PDU
传输层
段
网络层
报文
LLC子层
MAC子层
帧
物理层
比特

HTTP 状态

状态码	原因短语	
	消息响应	
100	Continue(继续)	
101	Switching Protocol(切换协议)	
	成功响应	
200	OK(成功)	
201	Created(已创建)	
202	Accepted(已创建)	
203	Non-Authoritative Information(未授权信息)	
204	No Content(无内容)	
205	Reset Content(重置内容)	
206	Partial Content(部分内容)	
	重定向	
300	Multiple Choice(多种选择)	
301	Moved Permanently(永久移动)	
302	Found(临时移动)	
303	See Other(查看其他位置)	
304	Not Modified(未修改)	
305	Use Proxy(使用代理)	
306	unused(未使用)	
307	Temporary Redirect(临时重定向)	
308	Permanent Redirect(永久重定向)	
	客户端错误	
400	Bad Request(错误请求)	
401	Unauthorized(未授权)	
402	Payment Required(需要付款)	
403	Forbidden(禁止访问)	
404	Not Found(未找到)	
405	Method Not Allowed(不允许使用该方法)	
406	Not Acceptable(无法接受)	
407	Proxy Authentication Required(要求代理身份验证)	
408	Request Timeout(请求超时)	
409	Conflict(冲突)	
410	Gone(已失效)	
411	Length Required(需要内容长度头)	
412	Precondition Failed(预处理失败)	
413	Request Entity Too Large(请求实体过长)	
414	Request-URI Too Long(请求网址过长)	
415	Unsupported Media Type(媒体类型不支持)	
416	Requested Range Not Satisfiable(请求范围不合要求)	
417	Expectation Failed(预期结果失败)	
	服务器端错误	
500	Internal Server Error(内部服务器错误)	
501	Implemented(未实现)	
502	Bad Gateway(网关错误)	
503	Service Unavailable(服务不可用)	
504	Gateway Timeout (网关超时)	

状态码	原因短语	
505	HTTP Version Not Supported(HTTP 版本不受支持)	

HTTPS

HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL。其所用的端口号是 443。 过程大致如下：

建立连接获取证书

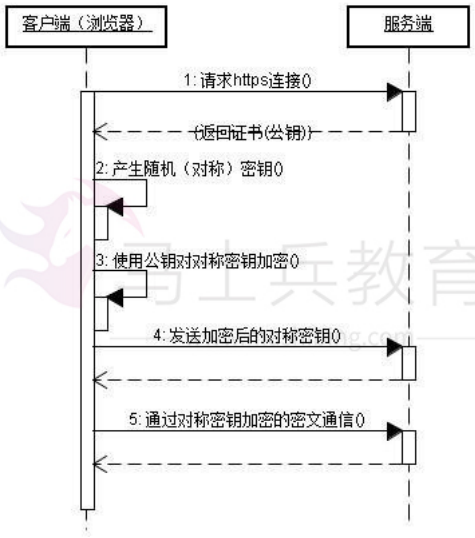
1） SSL 客户端通过 TCP 和服务器建立连接之后（443 端口），并且在一般的 tcp 连接协商（握手）过程中请求证书。即客户端发出一个消息给服务器，这个消息里面包含了自己可实现的算法列表和其它一些需要的消息，SSL 的服务器端会回应一个数据包，这里面确定了这次通信所需要的算法，然后服务器向客户端返回证书。（证书里面包含了服务器信息：域名。申请证书的公司，公共秘钥）。

证书验证

2） Client 在收到服务器返回的证书后，判断签发这个证书的公共签发机构，并使用这个机构的公共秘钥确认签名是否有效，客户端还会确保证书中列出的域名就是它正在连接的域名。

数据加密和传输

3） 如果确认证书有效，那么生成对称秘钥并使用服务器的公共秘钥进行加密。然后发送给服务器，服务器使用它的私钥对它进行解密，这样两台计算机可以开始进行对称加密进行通信。



CDN 原理

CND 一般包含分发服务系统、负载均衡系统和管理系统

分发服务系统

其基本的工作单元就是各个 Cache 服务器。负责直接响应用户请求，将内容快速分发到用户；同时还负责内容更新，保证和源站内容的同步。根据内容类型和服务种类的不同，分发服务系统分为多个子服务系统，如：网页加速服务、流媒体加速服务、应用加速服务等。每个子服务系统都是一个分布式的服务集群，由功能类似、地域接近的分布部署的 Cache 集群组成。

在承担内容同步、更新和响应用户请求之外，分发服务系统还需要向上层的管理调度系统反馈各个Cache 设备的健康状况、响应情况、内容缓存状况等，以便管理调度系统能够根据设定的策略决定由哪个 Cache 设备来响应用户的请求。

负载均衡系统：

负载均衡系统是整个 CDN 系统的中枢。负责对所有的用户请求进行调度，确定提供给用户的最终访问地址。

使用分级实现。最基本的两极调度体系包括全局负载均衡（GSLB）和本地负载均衡（SLB）。

GSLB 根据用户地址和用户请求的内容，主要根据就近性原则，确定向用户服务的节点。一般通过 DNS 解析或者应用层重定向（Http 3XX 重定向）的方式实现。

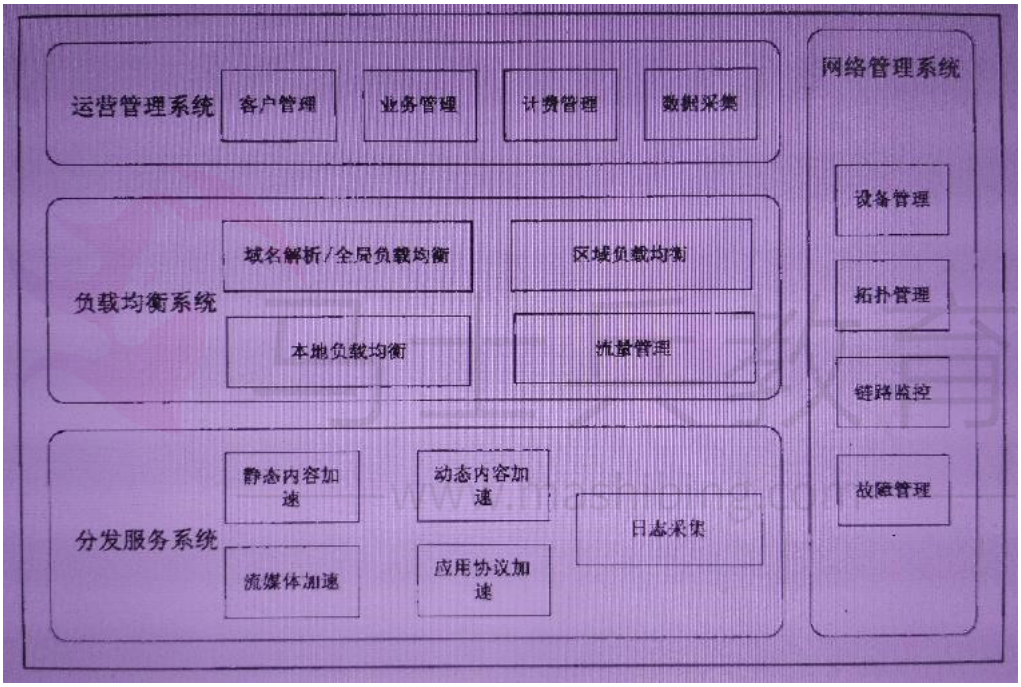
SLB 主要负责节点内部的负载均衡。当用户请求从 GSLB 调度到 SLB 时，SLB 会根据节点内各个

Cache 设备的工作状况和内容分布情况等对用户请求重定向。SLB 的实现有四层调度（LVS）、七层调度（Nginx）和链路负载调度等。

管理系统：分为运营管理和网络管理子系统。

网络管理系统实现对 CDN 系统的设备管理、拓扑管理、链路监控和故障管理，为管理员提供对全网资源的可视化的集中管理，通常用 web 方式实现。

运营管理是对 CDN 系统的业务管理，负责处理业务层面的与外界系统交互所必须的一些收集、整理、交付工作。包括用户管理、产品管理、计费管理、统计分析等。



RPC

为什么要有RPC

http接口是在接口不多、系统与系统交互较少的情况下，解决信息孤岛初期常使用的一种通信手段；优点就是简单、直接、开发方便。利用现成的http协议进行传输。但是如果是一个大型的网站，内部子系统较多、接口非常多的情况下，

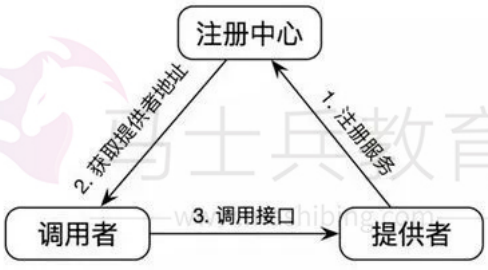
RPC框架的好处就显示出来了，首先就是长链接，不必每次通信都要像http一样去3次握手什么的，减少了网络开销；其次就是RPC框架一般都有注册中心，有丰富的监控管理；发布、下线接口、动态扩展等，对调用方来说是无感知、统一化的操作。第三个来说就是安全性。 后就是 近流行的服务化架构、服务化治理，RPC框架是一个强力的支撑。

socket只是一个简单的网络通信方式，只是创建通信双方的通信通道，而要实现rpc的功能，还需要对其进行封装，以实现更多的功能。

RPC一般配合netty框架、spring自定义注解来编写轻量级框架，其实netty内部是封装了socket的，较新的jdk的IO一般是NIO，即非阻塞IO，在高并发网站中，RPC的优势会很明显

什么是RPC

RPC（Remote Procedure Call Protocol）远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。简言之，RPC使得程序能够像访问本地系统资源一样，去访问远端系统资源。比较关键的一些方面包括：通讯协议、序列化、资源（接口）描述、服务框架、性能、语言支持等。



简单的说，RPC就是从一台机器(客户端)上通过参数传递的方式调用另一台机器

(服务器)上的一个函数或方法(可以统称为服务)并得到返回的结果。

PRC架构组件

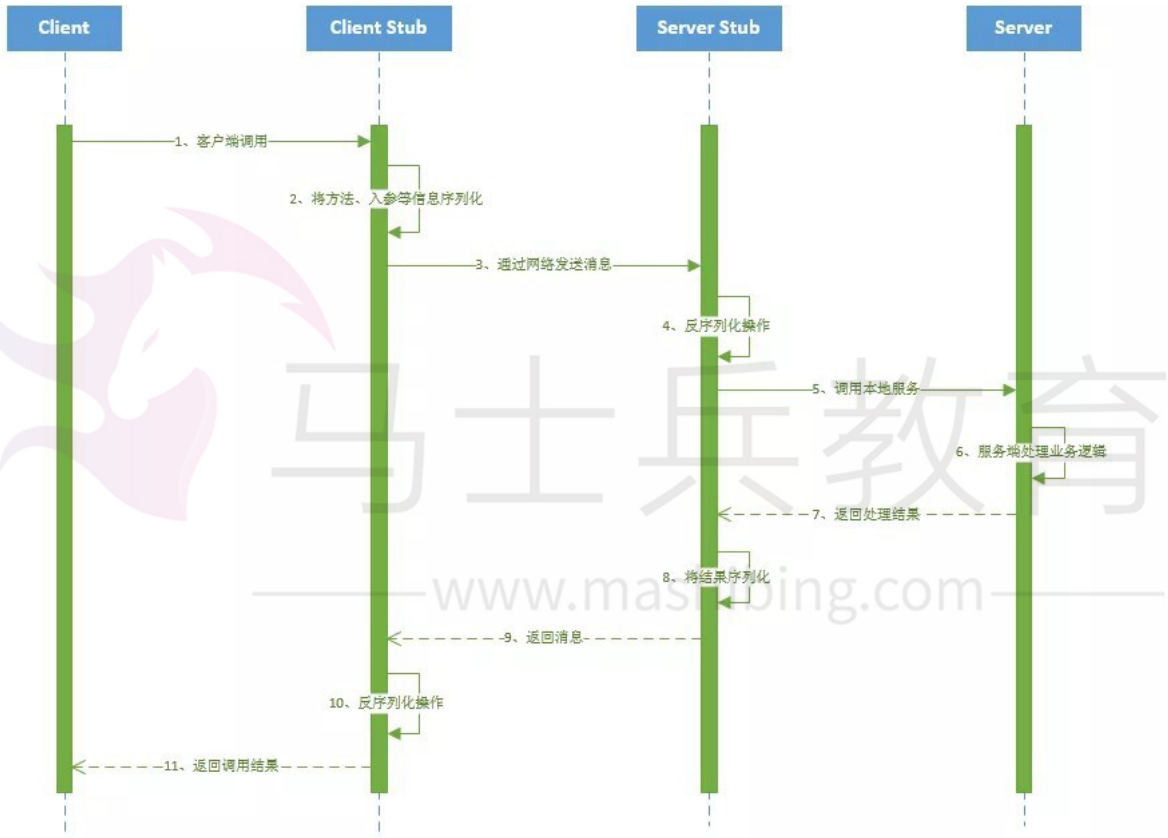
一个基本的RPC架构里面应该至少包含以下4个组件：

1、 客户端（Client）:服务调用方（服务消费者）

2、 客户端存根（Client Stub）:存放服务端地址信息，将客户端的请求参数数据信息打包成网络消息，再通过网络传输发送给服务端

3、 服务端存根（Server Stub）:接收客户端发送过来的请求消息并进行解包，然后再调用本地服务进行处理

4、 服务端（Server）:服务的真正提供者



具体调用过程：

1、 服务消费者（client客户端）通过调用本地服务的方式调用需要消费的服务；

2、 客户端存根（client stub）接收到调用请求后负责将方法、入参等信息序列化（组装）成能够进行网络传输的消息体；

3、 客户端存根（client stub）找到远程的服务地址，并且将消息通过网络发送给服务端；

4、 服务端存根（server stub）收到消息后进行解码（反序列化操作）；

5、 服务端存根（server stub）根据解码结果调用本地的服务进行相关处理；

6、 本地服务执行具体业务逻辑并将处理结果返回给服务端存根（server stub）；

7、 服务端存根（server stub）将返回结果重新打包成消息（序列化）并通过网络发送至消费方；

8、 客户端存根（client stub）接收到消息，并进行解码（反序列化）；

9、 服务消费方得到 终结果；而RPC框架的实现目标则是将上面的第2-10步完好地封装起来，也就是把调用、编码/解码的过程给封装起来，让用户感觉上像调用本地服务一样的调用远程服务。

RPC和SOA、SOAP、REST的区别

1、REST

可以看着是HTTP协议的一种直接应用，默认基于JSON作为传输格式,使用简单, 学习成本低效率高,但是安全性较低。

2、SOAP

SOAP是一种数据交换协议规范,是一种轻量的、简单的、基于XML的协议的规范。而SOAP可以看着是一个重量级的协议，基于XML、SOAP在安全方面是通过使用XML-Security和XML-Signature两个规范组成了WS-Security来实现安全控制的,当前已经得到了各个厂商的支持 。

它有什么优点？简单总结为：易用、灵活、跨语言、跨平台。

3、SOA

面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是SOA的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。

SOA是一种粗粒度、松耦合服务架构，服务之间通过简单、精确定义接口进行通讯，不涉及底层编程接口和通讯模型。SOA可以看作是B/S模型、XML（标准通用标记语言的子集）/Web Service技术之后的自然延伸。

4、REST 和 SOAP、RPC 有何区别呢?

没什么太大区别，他们的本质都是提供可支持分布式的基础服务， 大的区别在于他们各自的的特点所带来的不同应用场景 。

RPC框架需要解决的问题?

1、 如何确定客户端和服务端之间的通信协议?

2、 如何更高效地进行网络通信?

3、 服务端提供的服务如何暴露给客户端?

4、 客户端如何发现这些暴露的服务?

5、 如何更高效地对请求对象和响应结果进行序列化和反序列化操作?

RPC的实现基础?

1、 需要有非常高效的网络通信，比如一般选择Netty作为网络通信框架;

2、 需要有比较高效的序列化框架，比如谷歌的Protobuf序列化框架;

3、 可靠的寻址方式（主要是提供服务的发现），比如可以使用Zookeeper来注册服务等等;

4、 如果是带会话（状态）的RPC调用，还需要有会话和状态保持的功能;

RPC使用了哪些关键技术?

1、动态代理

生成Client Stub（客户端存根）和Server Stub（服务端存根）的时候需要用到 Java动态代理技术，可以使用JDK提供的原生的动态代理机制，也可以使用开源的：CGLib代理，Javassist字节码生成技术。

2、序列化和反序列化在网络中，所有的数据都将会被转化为字节进行传送，所以为了能够使参数对象在网络中进行传输，需要对这些参数进行序列化和反序列化操作。

- 序列化：把对象转换为字节序列的过程称为对象的序列化，也就是编码的过程。
- 反序列化：把字节序列恢复为对象的过程称为对象的反序列化，也就是解码的过程。

目前比较高效的开源序列化框架：如Kryo、FastJson和Protobuf等。

1、NIO通信

出于并发性能的考虑，传统的阻塞式 IO 显然不太合适，因此我们需要异步的

IO，即 NIO。Java 提供了 NIO 的解决方案，Java 7 也提供了更优秀的 NIO.2 支持。可以选择Netty或者MINA来解决NIO数据传输的问题。

2、服务注册中心

可选：Redis、Zookeeper、Consul 、Etcd。一般使用ZooKeeper提供服务注册与发现功能，解决单点故障以及分布式部署的问题(注册中心)。

主流RPC框架有哪些

1、RMI

利用java.rmi包实现，基于Java远程方法协议(Java Remote Method Protocol) 和java的原生序列化。

2、Hessian

是一个轻量级的remoting onhttp工具，使用简单的方法提供了RMI的功能。 基于HTTP协议，采用二进制编解码。

3、protobuf-rpc-pro是一个Java类库，提供了基于 Google 的 Protocol Buffers 协议的远程方法调用的框架。基于 Netty 底层的 NIO 技术。支持 TCP 重用/ keep-alive、SSL加密、RPC 调用取消操作、嵌入式日志等功能。

4、Thrift

是一种可伸缩的跨语言服务的软件框架。它拥有功能强大的代码生成引擎，无缝地支持C + +，C#，Java，Python和PHP和Ruby。thrift允许你定义一个描述

文件，描述数据类型和服务接口。依据该文件，编译器方便地生成RPC客户端和服务器通信代码。

初由facebook开发用做系统内个语言之间的RPC通信，2007年由facebook 贡献到apache基金 ，现在是apache下的opensource之一 。支持多种语言之间的RPC方式的通信：php语言client可以构造一个对象，调用相应的服务方法

来调用java语言的服务，跨越语言的C/S RPC调用。底层通讯基于SOCKET。

5、Avro

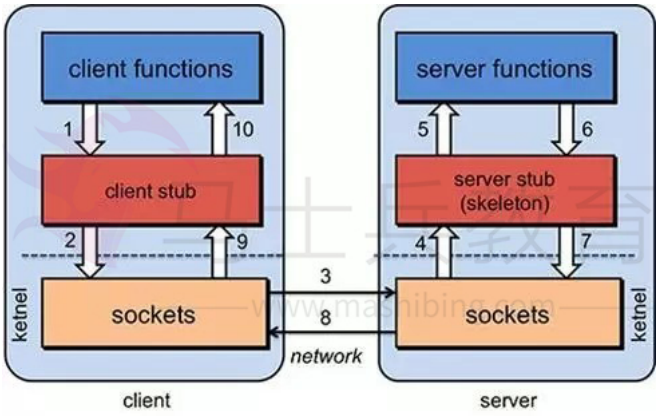
出自Hadoop之父Doug Cutting, 在Thrift已经相当流行的情况下推出Avro的目标不仅是提供一套类似Thrift的通讯中间件,更是要建立一个新的，标准性的云计算的数据交换和存储的Protocol。支持HTTP，TCP两种协议。

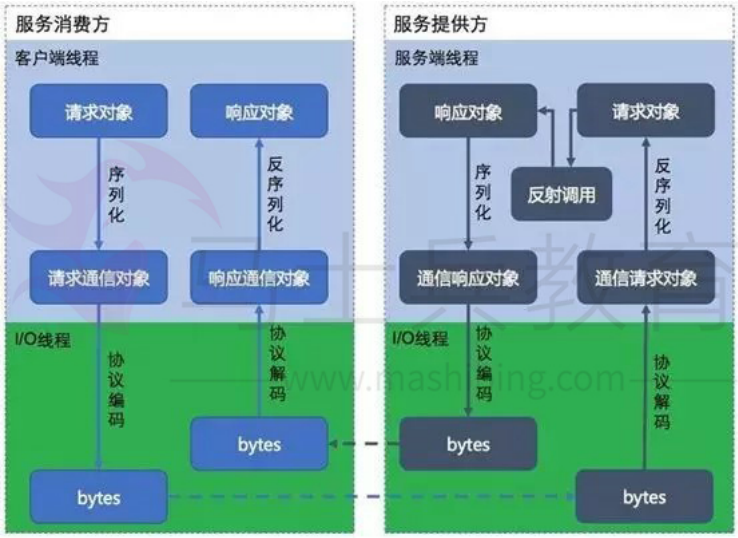
6、Dubbo

Dubbo是 阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高

性能的 RPC 实现服务的输出和输入功能，可以和 Spring框架无缝集成。

RPC的实现原理架构图





PS：这张图非常重点，是PRC的基本原理，请大家一定记住！也就是说两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数/方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。

比如说，A服务器想调用B服务器上的一个方法：

User getUserByName(String userName)

1、建立通信首先要解决通讯的问题：即A机器想要调用B机器，首先得建立起通信连接。

主要是通过在客户端和服务器之间建立TCP连接，远程过程调用的所有交换的数据都在这个连接里传输。连接可以是按需连接，调用结束后就断掉，也可以是长连接，多个远程过程调用共享同一个连接。

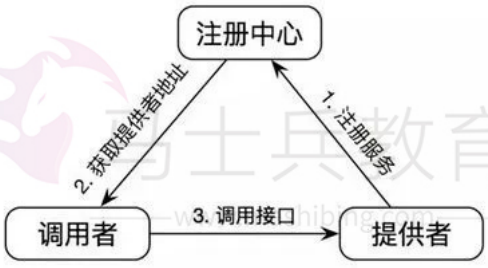
通常这个连接可以是按需连接（需要调用的时候就先建立连接，调用结束后就立马断掉），也可以是长连接（客户端和服务器建立起连接之后保持长期持有，不管此时有无数据包的发送，可以配合心跳检测机制定期检测建立的连接是否存活有效），多个远程过程调用共享同一个连接。

2、服务寻址要解决寻址的问题，也就是说，A服务器上的应用怎么告诉底层的RPC框架，如何连接到B服务器（如主机或IP地址）以及特定的端口，方法的名称名称是什么。

通常情况下我们需要提供B机器（主机名或IP地址）以及特定的端口，然后指定调用的方法或者函数的名称以及入参出参等信息，这样才能完成服务的一个调用。

可靠的寻址方式（主要是提供服务的发现）是RPC的实现基石，比如可以采用

Redis或者Zookeeper来注册服务等等。



2.1、 从服务提供者的角度看：当服务提供者启动的时候，需要将自己提供的服务注册到指定的注册中心，以便服务消费者能够通过服务注册中心进行查找；当服务提供者由于各种原因致使提供的服务停止时，需要向注册中心注销停止的服务；服务的提供者需要定期向服务注册中心发送心跳检测，服务注册中心如果一段时间未收到来自服务提供者的心跳后，认为该服务提供者已经停止服务，则将该服务从注册中心上去掉。

2.2、 从调用者的角度看：服务的调用者启动的时候根据自己订阅的服务向服务注册中心查找服务提供者的地址等信息；当服务调用者消费的服务上线或者下线的时候，注册中心会告知该服务的调用者；服务调用者下线的时候，则取消订阅。

3、网络传输3.1、序列化

当A机器上的应用发起一个RPC调用时，调用方法和其入参等信息需要通过底层的网络协议如TCP传输到B机器，由于网络协议是基于二进制的，所有我们传输的参数数据都需要先进行序列化（Serialize）或者编组（marshal）成二进制的形式才能在网络中进行传输。然后通过寻址操作和网络传输将序列化或者编组之后的二进制数据发送给B机器。

3.2、反序列化

当B机器接收到A机器的应用发来的请求之后，又需要对接收到的参数等信息进行反序列化操作（序列化的逆操作），即将二进制信息恢复为内存中的表达方

式，然后再找到对应的方法（寻址的一部分）进行本地调用（一般是通过生成代理Proxy去调用，

通常会有JDK动态代理、CGLIB动态代理、Javassist生成字节码技术等），之后得到调用的返回值。

4、服务调用

B机器进行本地调用（通过代理Proxy和反射调用）之后得到了返回值，此时还需要再把返回值发送回A机器，同样也需要经过序列化操作，然后再经过网络传输将二进制数据发送回A机器，而当A机器接收到这些返回值之后，则再次进行反序列化操作，恢复为内存中的表达方式， 后再交给A机器上的应用进行相关处理（一般是业务逻辑处理操作）。通常，经过以上四个步骤之后，一次完整的RPC调用算是完成了，另外可能因为网络抖动等原因需要重试等。

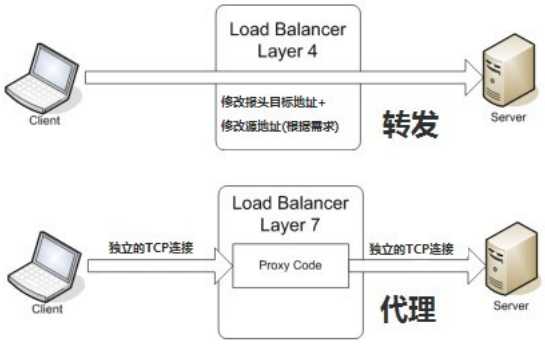
Znode 有四种形式的目录节点

1. PERSISTENT：持久的节点。
2. EPHEMERAL：暂时的节点。
3. PERSISTENT_SEQUENTIAL：持久化顺序编号目录节点。
4. EPHEMERAL_SEQUENTIAL：暂时化顺序编号目录节点。

负载均衡

负载均衡 建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

四层负载均衡 vs 七层负载均衡



四层负载均衡（目标地址和端口交换）

主要通过报文中的目标地址和端口，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

以常见的 TCP 为例，负载均衡设备在接收到第一个来自客户端的 SYN 请求时，即通过上述方式选择一个最佳的服务器，并对报文中目标 IP 地址进行修改(改为后端服务器 IP），直接转发给该服务器。TCP 的连接建立，即三次握手是客户端和服务器直接建立的，负载均衡设备只是起到一个类似路由器的转发动作。在某些部署情况下，为保证服务器回包可以正确返回给负载均衡设备，在转发报文的同时可能还会对报文原来的源地址进行修改。实现四层负载均衡的软件有：

F5：硬件负载均衡器，功能很好，但是成本很高。

lvs：重量级的四层负载软件。

nginx：轻量级的四层负载软件，带缓存功能，正则表达式较灵活。

haproxy：模拟四层转发，较灵活。

七层负载均衡（内容交换）

所谓七层负载均衡，也称为“内容交换”，也就是主要通过报文中的真正有意义的应用层内容，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。七层应用负载的好处，是使得整个网络更智能化。例如访问一个网站的用户流量，可以通过七层的方式，将对图片类的请求转发到特定的图片服务器并可以使用缓存技术；将对文字类的请求可以转发到特定的文字服务器并可以使用压缩技术。实现七层负载均衡的软件有：

haproxy：天生负载均衡技能，全面支持七层代理，会话保持，标记，路径转移； **nginx**：只在**http** 协议和**mail**协议上功能比较好，性能与**haproxy**差不多； **apache**：功能较差

Mysql proxy：功能尚可。

负载均衡算法/策略

轮循均衡（Round Robin）

每一次来自网络的请求轮流分配给内部中的服务器，从 1 至 N 然后重新开始。此种均衡算法适合于服务器组中的所有服务器都有相同的软硬件配置并且平均服务请求相对均衡的情况。

权重轮循均衡（Weighted Round Robin）

根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。例如：服务器 A 的权值被设计成 1，B 的权值是 3，C 的权值是 6，则服务器 A、B、C 将分别接受到 10%、30%、60%的服务请求。此种均衡算法能确保高性能的服务器得到更多的使用率，避免低性能的服务器负载过重。

随机均衡（Random）把来自网络的请求随机分配给内部中的多个服务器。

权重随机均衡（Weighted Random）

此种均衡算法类似于权重轮循算法，不过在处理请求分担时是个随机选择的过程。

响应速度均衡（Response Time 探测时间）

负载均衡设备对内部各服务器发出一个探测请求（例如 Ping），然后根据内部中各服务器对探测请求的最快响应时间来决定哪一台服务器来响应客户端的服务请求。此种均衡算法能较好的反映服务器的当前运行状态，但这最快响应时间仅仅指的是负载均衡设备与服务器间的最快响应时间，而不是客户端与服务器间的最快响应时间。

最少连接数均衡（Least Connection）

最少连接数均衡算法对内部中需负载的每一台服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使均衡更加符合实际情况，负载更加均衡。此种均衡算法适合长时处理的请求服务，如 FTP。

处理能力均衡（CPU、内存）

此种均衡算法将把服务请求分配给内部中处理负荷（根据服务器 CPU 型号、CPU 数量、内存大小及当前连接数等换算而成）最轻的服务器，由于考虑到了内部服务器的处理能力及当前网络运行状况，所以此种均衡算法相对来说更加精确，尤其适合运用到第七层（应用层）负载均衡的情况下。

DNS响应均衡（Flash DNS）

在此均衡算法下，分处在不同地理位置的负载均衡设备收到同一个客户端的域名解析请求，并在同一时间内把此域名解析成各自相对应服务器的 IP 地址并返回给客户端，则客户端将以最先收到的域名解析 IP 地址来继续请求服务，而忽略其它的 IP 地址响应。在种均衡策略适合应用在全局负载均衡的情况下，对本地负载均衡是没有意义的。

哈希算法

一致性哈希一致性 Hash，相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

IP地址散列（保证客户端服务器对应关系稳定）

通过管理发送方 IP 和目的地 IP 地址的散列，将来自同一发送方的分组(或发送至同一目的地的分组)统一转发到相同服务器的算法。当客户端有一系列业务需要处理而必须和一个服务器反复通信时，该算法能够以流(会话)为单位，保证来自相同客户端的通信能够一直在同一服务器中进行处理。

URL散列

通过管理客户端请求 URL 信息的散列，将发送至相同 URL 的请求转发至同一服务器的算法。

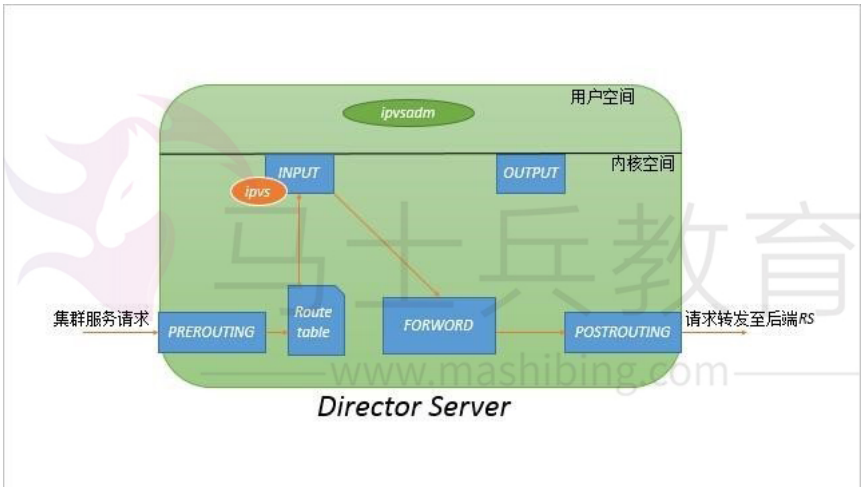
LVS

LVS 原理

IPVS

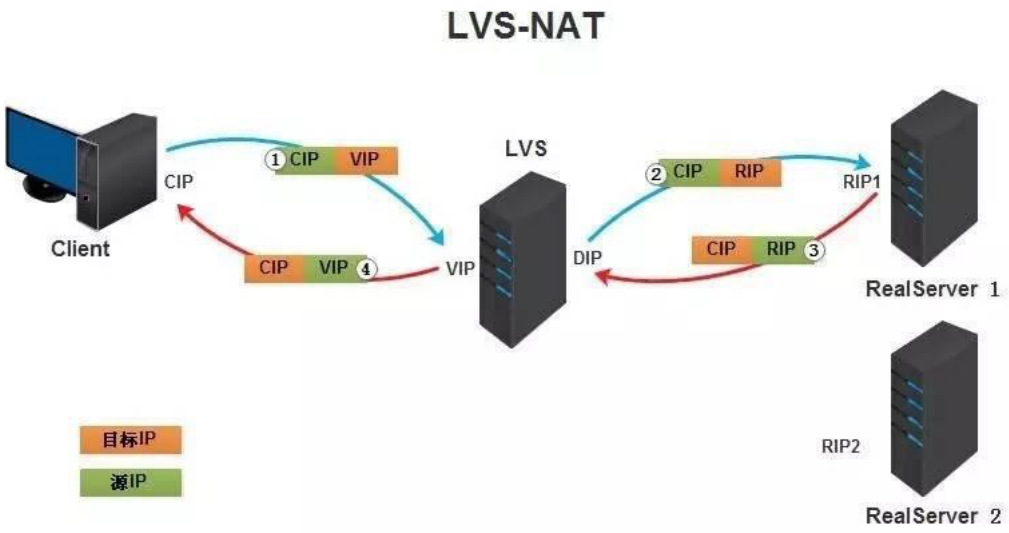
LVS 的 IP 负载均衡技术是通过 IPVS 模块来实现的，IPVS 是 LVS 集群系统的核心软件，它的主要作用是：安装在 Director Server 上，同时在 Director Server 上虚拟出一个 IP 地址，用户必须通过这个虚拟的 IP 地址访问服务器。这个虚拟 IP 一般称为 LVS 的 VIP，即 Virtual IP。访问的请求首先经过 VIP 到达负载调度器，然后由负载调度器从 Real Server 列表中选取一个服务节点响应用户的请求。 在用户的请求到达负载调度器后，调度器如何将请求发送到提供服务的 Real Server 节点，而 Real Server 节点如何返回数据给用户，是 IPVS 实现的重点技术。

ipvs： 工作于内核空间，主要用于使用户定义的策略生效 ipvsadm : 工作于用户空间，主要用于用户定义和管理集群服务的工具



ipvs 工作于内核空间的 INPUT 链上，当收到用户请求某集群服务时，经过 PREROUTING 链，经检查本机路由表，送往 INPUT 链；在进入 netfilter 的 INPUT 链时，ipvs 强行将请求报文通过ipvsadm 定义的集群服务策略的路径改为 FORWORD 链，将报文转发至后端真实提供服务的主机。

LVS NAT 模式



①.客户端将请求发往前端的负载均衡器，请求报文源地址是 CIP(客户端 IP),后面统称为 CIP)，目标地址为 VIP(负载均衡器前端地址，后面统称为 VIP)。

②.负载均衡器收到报文后，发现请求的是在规则里面存在的地址，那么它将客户端请求报文的目标地址改为了后端服务器的 RIP 地址并将报文根据算法发送出去。

③.报文送到 Real Server 后，由于报文的目标地址是自己，所以会响应该请求，并将响应报文返还给 LVS。

④.然后 lvs 将此报文的源地址修改为本机并发送给客户端。

注意：在 NAT 模式中，Real Server 的网关必须指向 LVS，否则报文无法送达客户端**特点：**

1、NAT 技术将请求的报文和响应的报文都需要通过 LB 进行地址改写，因此网站访问量比较大的时候 LB 负载均衡调度器有比较大的瓶颈，一般要求最多之能 10-20 台节点

2、只需要在 LB 上配置一个公网 IP 地址就可以了。

3、每台内部的 realserver 服务器的网关地址必须是调度器 LB 的内网地址。

4、NAT 模式支持对 IP 地址和端口进行转换。即用户请求的端口和真实服务器的端口可以不一致。

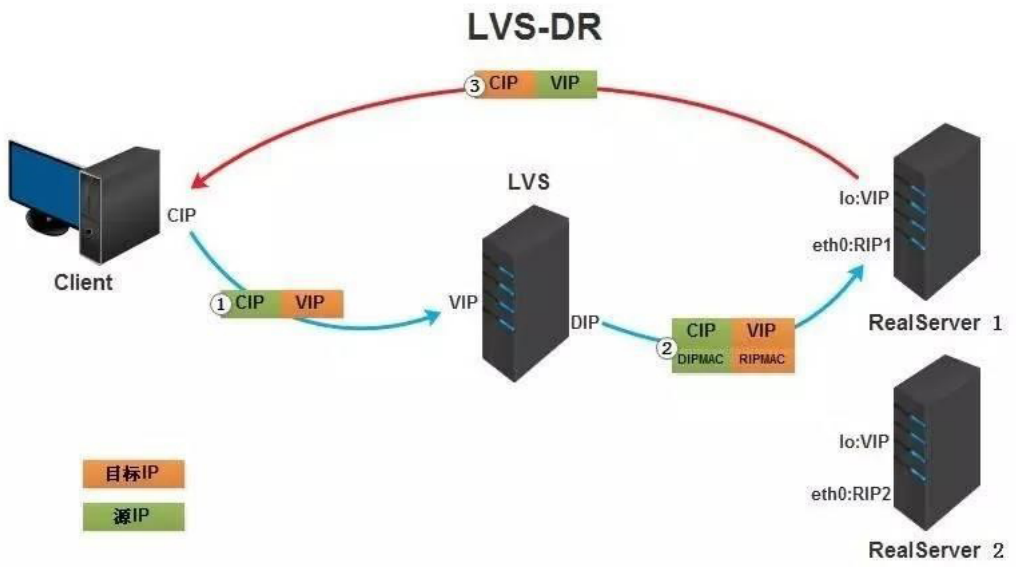
优点：

集群中的物理服务器可以使用任何支持 TCP/IP 操作系统，只有负载均衡器需要一个合法的 IP 地址。

缺点：

扩展性有限。当服务器节点（普通 PC 服务器）增长过多时,负载均衡器将成为整个系统的瓶颈，因为所有的请求包和应答包的流向都经过负载均衡器。当服务器节点过多时，大量的数据包都交汇在负载均衡器那，速度就会变慢！

LVS DR 模式（局域网改写 **mac** 地址）



①.客户端将请求发往前端的负载均衡器，请求报文源地址是 CIP，目标地址为 VIP。

②.负载均衡器收到报文后，发现请求的是在规则里面存在的地址，那么它将客户端请求报文的源 MAC地址改为自己DIP的MAC地址，目标MAC改为了RIP的MAC地址，并将此包发送给RS。 ③.RS 发现请求报文中的目的 MAC 是自己，就会将次报文接收下来，处理完请求报文后，将响应报文通过 lo 接口送给 eth0 网卡直接发送给客户端。

注意：需要设置 lo 接口的 VIP 不能响应本地网络内的 arp 请求。

总结：

1、通过在调度器 LB 上修改数据包的目的 MAC 地址实现转发。注意源地址仍然是 CIP，目的地址仍然是 VIP 地址。

2、请求的报文经过调度器，而 RS 响应处理后的报文无需经过调度器 LB，因此并发访问量大时使用效率很高（和 NAT 模式比）

3、因为 DR 模式是通过 MAC 地址改写机制实现转发，因此所有 RS 节点和调度器 LB 只能在一个局域网里面

4、RS 主机需要绑定 VIP 地址在 LO 接口（掩码 32 位）上，并且需要配置 ARP 抑制。

5、RS 节点的默认网关不需要配置成 LB，而是直接配置为上级路由的网关，能让 RS 直接出网就可以。

6、由于 DR 模式的调度器仅做 MAC 地址的改写，所以调度器 LB 就不能改写目标端口，那么 RS 服务器就得使用和 VIP 相同的端口提供服务。

7、直接对外的业务比如 WEB 等，RS 的 IP 最好是使用公网 IP。对外的服务，比如数据库等最好使用内网 IP。

优点：

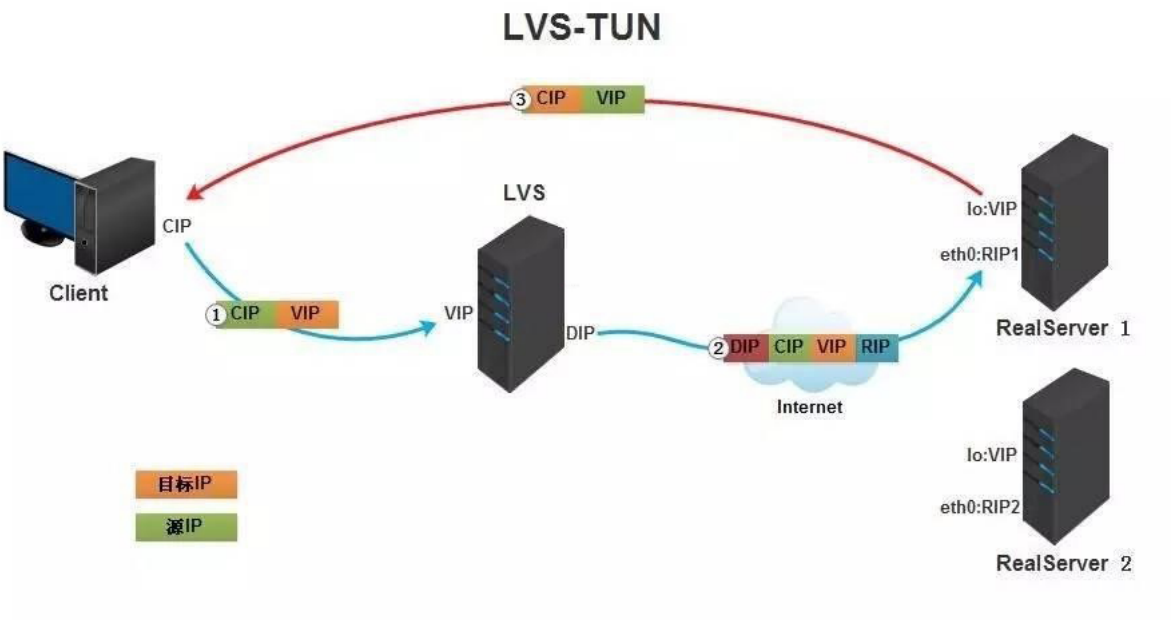
和 TUN（隧道模式）一样，负载均衡器也只是分发请求，应答包通过单独的路由方法返回给客户端。与 VS-TUN 相比，VS-DR 这种实现方式不需要隧道结构，因此可以使用大多数操作系统做为物理服务器。

DR 模式的效率很高，但是配置稍微复杂一点，因此对于访问量不是特别大的公司可以用

haproxy/nginx取代。日1000-2000W PV或者并发请求1万一下都可以考虑用haproxy/nginx。

缺点：所有 RS 节点和调度器 LB 只能在一个局域网里面

LVS TUN 模式（**IP** 封装、跨网段）



①.客户端将请求发往前端的负载均衡器，请求报文源地址是 CIP，目标地址为 VIP。

②.负载均衡器收到报文后，发现请求的是在规则里面存在的地址，那么它将在客户端请求报文的首部再封装一层 IP 报文,将源地址改为 DIP，目标地址改为 RIP,并将此包发送给 RS。

③.RS 收到请求报文后，会首先拆开第一层封装,然后发现里面还有一层 IP 首部的目标地址是自己 lo 接口上的 VIP，所以会处理次请求报文，并将响应报文通过 lo 接口送给 eth0 网卡直接发送给客户端。

注意：需要设置 lo 接口的 VIP 不能在共网上出现。

总结:

1.TUNNEL 模式必须在所有的 realserver 机器上面绑定 VIP 的 IP 地址

2.TUNNEL 模式的 vip ------>realserver 的包通信通过 TUNNEL 模式，不管是内网和外网都能通信，所以不需要 lvs vip 跟 realserver 在同一个网段内。

3.TUNNEL 模式 realserver 会把 packet 直接发给 client 不会给 lvs 了

4.TUNNEL 模式走的隧道模式，所以运维起来比较难，所以一般不用。

优点:

负载均衡器只负责将请求包分发给后端节点服务器，而 RS 将应答包直接发给用户。所以，减少了负载均衡器的大量数据流动，负载均衡器不再是系统的瓶颈，就能处理很巨大的请求量，这种方式，一台负载均衡器能够为很多 RS 进行分发。而且跑在公网上就能进行不同地域的分发。

缺点:

隧道模式的 RS 节点需要合法 IP，这种方式需要所有的服务器支持"IP Tunneling"(IP

Encapsulation)协议，服务器可能只局限在部分 Linux 系统上。

LVS FULLNAT 模式

无论是 DR 还是 NAT 模式，不可避免的都有一个问题：LVS 和 RS 必须在同一个 VLAN 下，否则

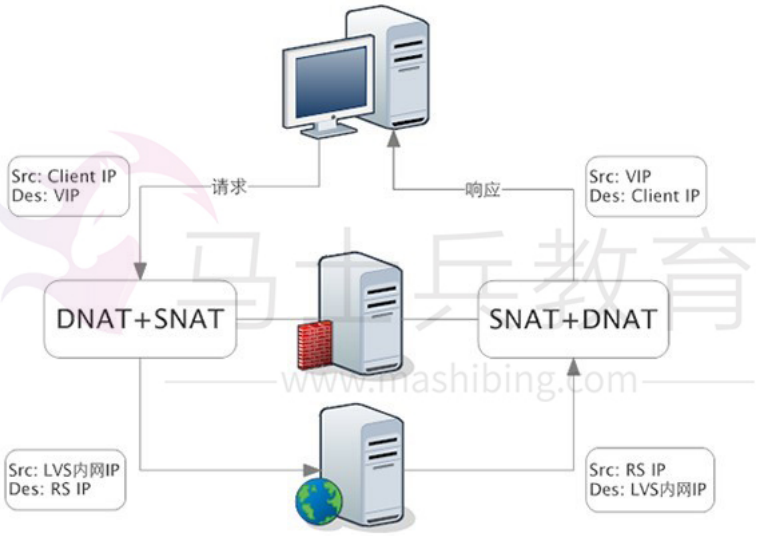
LVS 无法作为 RS 的网关。这引发的两个问题是:

1、同一个 VLAN 的限制导致运维不方便，跨 VLAN 的 RS 无法接入。

2、LVS 的水平扩展受到制约。当 RS 水平扩容时，总有一天其上的单点 LVS 会成为瓶颈。

Full-NAT 由此而生，解决的是 LVS 和 RS 跨 VLAN 的问题，而跨 VLAN 问题解决后，LVS 和 RS 不再存在 VLAN 上的从属关系，可以做到多个 LVS 对应多个 RS，解决水平扩容的问题。

Full-NAT 相比 NAT 的主要改进是，在 SNAT/DNAT 的基础上，加上另一种转换，转换过程如下:



1. 在包从 LVS 转到 RS 的过程中，源地址从客户端 IP 被替换成了 LVS 的内网 IP。内网 IP 之间可以通过多个交换机跨 VLAN 通信。目标地址从 VIP 修改为 RS IP.
2. 当 RS 处理完接受到的包，处理完成后返回时，将目标地址修改为 LVS ip，原地址修改为 RS IP，最终将这个包返回给 LVS 的内网 IP，这一步也不受限于 VLAN。
3. LVS 收到包后，在 NAT 模式修改源地址的基础上，再把 RS 发来的包中的目标地址从 LVS 内网 IP 改为客户端的 IP,并将原地址修改为 VIP。

Full-NAT 主要的思想是把网关和其下机器的通信，改为了普通的网络通信，从而解决了跨 VLAN 的问题。采用这种方式，LVS 和 RS 的部署在 VLAN 上将不再有任何限制，大大提高了运维部署的便利性。

总结

1. FULL NAT 模式不需要 LBIP 和 realserver ip 在同一个网段；
2. full nat 因为要更新 sorce ip 所以性能正常比 nat 模式下降 10%

Keepalive

keepalive 起初是为 LVS 设计的，专门用来监控 lvs 各个服务节点的状态，后来加入了 vrrp 的功能，因此除了 lvs，也可以作为其他服务（nginx，haproxy）的高可用软件。VRRP 是 virtual router redundancy protocal（虚拟路由器冗余协议）的缩写。VRRP 的出现就是为了解决静态路由出现的单点故障，它能够保证网络可以不间断的稳定的运行。所以 keepalive 一方面具有 LVS cluster node healthcheck 功能，另一方面也具有 LVS director failover。

Nginx 反向代理负载均衡

普通的负载均衡软件，如LVS，其实现的功能只是对请求数据包的转发、传递，从负载均衡下的节点服务器来看，接收到的请求还是来自访问负载均衡器的客户端的真实用户；而反向代理就不一样了，反向代理服务器在接收访问用户请求后，会代理用户 重新发起请求代理下的节点服务器，最后把数据返回给客户端用户。在节点服务器看来，访问的节点服务器的客户端用户就是反向代理服务器，而非真实的网站访问用户。

upstream_module 和健康检测

ngx_http_upstream_module 是负载均衡模块，可以实现网站的负载均衡功能即节点的健康检查，upstream 模块允许 Nginx 定义一组或多组节点服务器组，使用时可通过 proxy_pass 代理方式把网站的请求发送到事先定义好的对应 Upstream 组 的名字上。

upstream 模块内参数	参数说明
weight	服务器权重
max_fails	Nginx 尝试连接后端主机失败的此时，这是值是配合 proxy_next_upstream、 fastcgi_next_upstream 和 memcached_next_upstream 这三个参数来使用的。当 Nginx 接收后端服务器返回这三个参数定义的状态码时，会将这个请求转发给正常工作的的后端服务器。如 404、503、503,max_files=1
fail_timeout	max_fails 和 fail_timeout 一般会关联使用，如果某台 server 在 fail_timeout 时间内出现了 max_fails 次连接失败，那么 Nginx 会认为其已经挂掉，从而在 fail_timeout 时间内不再去请求它，fail_timeout 默认是 10s，max_fails 默认是 1，即默认情况只要是发生错误就认为服务器挂了，如果将 max_fails 设置为 0，则表示取消这项检查
backup	表示当前 server 是备用服务器，只有其它非 backup 后端服务器都挂掉了或很忙才会分配请求给它
down	标志服务器永远不可用，可配合 ip_hash 使用

```
upstream lvsServer{   server 191.168.1.11 weight=5 ;   server 191.168.1.22:82;   server example.com:8080 max_fails=2 fail_timeout=10s
backup;
  #域名的话需要解析的哦，内网记得 hosts
}
```

proxy_pass 请求转发

proxy_pass 指令属于 ngx_http_proxy_module 模块，此模块可以将请求转发到另一台服务器，在实际的反向代理工作中，会通过 location 功能匹配指定的 URI，然后把接收到服务匹配 URI 的请求通过 proyx_pass 抛给定义好的 upstream 节点池。

```
 location /download/ {   proxy_pass http://download/vedio/;
}
#这是前端代理节点的设置
```

#交给后端 upstream 为 download 的节点

proxy模块参数	说明
proxy_next_upstream	什么情况下将请求传递到下一个 upstream
proxy_limite_rate	限制从后端服务器读取响应的速率
proyx_set_header	设置 http 请求 header 传给后端服务器节点，如：可实现让代理后端的服务器节点获取访问客户端的这是 ip
client_body_buffer_size	客户端请求主体缓冲区大小
proxy_connect_timeout	代理与后端节点服务器连接的超时时间
proxy_send_timeout	后端节点数据回传的超时时间
proxy_read_timeout	设置 Nginx 从代理的后端服务器获取信息的时间，表示连接成功建立后，Nginx 等待后端服务器的响应时间
proxy_buffer_size	设置缓冲区大小
proxy_buffers	设置缓冲区的数量和大小
proyx_busy_buffers_size	用于设置系统很忙时可以使用的 proxy_buffers 大小，推荐为 proxy_buffers*2
proxy_temp_file_write_size	指定 proxy 缓存临时文件的大小

Linux（李老师）

Linux概述

1. 什么是Linux

Linux是一套免费使用和自由传播的类Unix操作系统，是一个基于POSIX和Unix 的多用户、多任务、支持多线程和多CPU的操作系统。它能运行主要的Unix工 具软件、应用程序和网络协议。它支持32位和64位硬件。Linux继承了Unix以网 络为核心的设计思想，是一个性能稳定的多用户网络操作系统。

2. Unix和Linux有什么区别?

Linux和Unix都是功能强大的操作系统，都是应用广泛的服务器操作系统，有很 多相似之处，甚至有一部分人错误地认为Unix和Linux操作系统是一样的，然 而，事实并非如此，以下是两者的区别。

1. 开源性 Linux是一款开源操作系统，不需要付费，即可使用；Unix是一款对源码实行知 识产权保护的传统商业软件，使用需要付费授权使用。
2. 跨平台性 Linux操作系统具有良好的跨平台性能，可运行在多种硬件平台上；Unix操作系 统跨平台性能较弱，大多需与硬件配套使用。
3. 可视化界面 Linux除了进行命令行操作，还有窗体管理系统；Unix只是命令行下的系统。
4. 硬件环境 Linux操作系统对硬件的要求较低，安装方法更易掌握；Unix对硬件要求比较苛 刻，按照难度较大。
5. 用户群体 Linux的用户群体很广泛，个人和企业均可使用；Unix的用户群体比较窄，多是 安全性要求高的大型企业使用，如银行、电信部门等，或者Unix硬件厂商使 用，如Sun等。

相比于Unix操作系统，Linux操作系统更受广大计算机爱好者的喜爱，主要原因 是Linux操作系统具有Unix操作系统的全部功能，并且能够在普通PC计算机上实 现全部的Unix特性，开源免费的特性，更容易普及使用!

3. 什么是 Linux 内核?

Linux 系统的核心是内核。内核控制着计算机系统上的所有硬件和软件，在必要 时分配硬件，并根据需要执行软件。

系统内存管理

应用程序管理

硬件设备管理

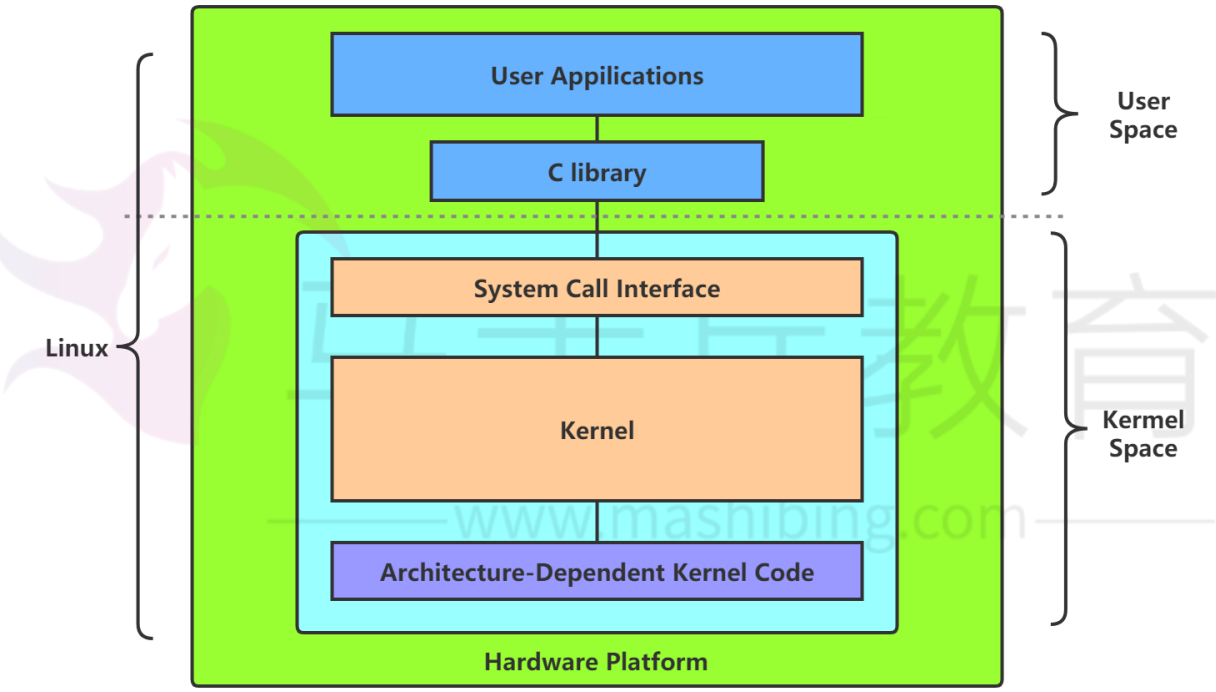
文件系统管理

4. Linux的基本组件是什么?

就像任何其他典型的操作系统一样，Linux拥有所有这些组件：内核，shell和 GUI，系统实用程序和应用程序。Linux比其他操作系统更具优势的是每个方面 都附带其他功能，所有代码都可以免费下载。

5. Linux 的体系结构

从大的方面讲，Linux 体系结构可以分为两块：



- 用户空间(User Space)： 用户空间又包括用户的应用程序(User Applications)、C 库(C Library) 。
- 内核空间(Kernel Space)： 内核空间又包括系统调用接口(System Call Interface)、内核(Kernel)、平台架构相关的代码(Architecture-Dependent Kernel Code) 。

为什么 Linux 体系结构要分为用户空间和内核空间的原因?

- 现代 CPU 实现了不同的工作模式，不同模式下 CPU 可以执行的指令和访问 的寄存器不同。
- Linux 从 CPU 的角度出发，为了保护内核的安全，把系统分成了两部分。

用户空间和内核空间是程序执行的**两种不同的状态**，我们可以通过两种方式完成 用户空间到内核空间的转移：1）系统调用；2）硬件中断。

6. BASH和DOS之间的基本区别是什么?

BASH和DOS控制台之间的主要区别在于3个方面：

- BASH命令区分大小写，而DOS命令则不区分;
- 在BASH下，/ character是目录分隔符，\作为转义字符。在DOS下，/用作命令 参数分隔符，\是目录分隔符
- DOS遵循命名文件中的约定，即8个字符的文件名后跟一个点，扩展名为3个字 符。BASH没有遵循这样的惯例。

7. Linux 开机启动过程?

了解即可。

1. 主机加电自检，加载 BIOS 硬件信息。
2. 读取 MBR 的引导文件(GRUB、LILO)。
3. 引导 Linux 内核。
4. 运行第一个进程 init (进程号永远为 1)。
5. 进入相应的运行级别。
6. 运行终端，输入用户名和密码。

8. Linux系统缺省的运行级别?

- 关机。
- 单机用户模式。
- 字符界面的多用户模式(不支持网络)。
- 字符界面的多用户模式。
- 未分配使用。
- 图形界面的多用户模式。
- 重启。

9. Linux 使用的进程间通信方式?

了解即可，不需要太深入。

1. 管道(pipe)、流管道(s_pipe)、有名管道(FIFO)。
2. 信号(signal) 。
3. 消息队列。
4. 共享内存。
5. 信号量。
6. 套接字(socket) 。

10. Linux 有哪些系统日志文件?

比较重要的是 `/var/log/messages` 日志文件。
该日志文件是许多进程日志文件的汇总，从该文件可以看出任何入侵企图或成功的入 侵。
另外，如果胖友的系统里有 ELK 日志集中收集，它也会被收集进去。

Linux系统安装多个桌面环境有帮助吗?

通常，一个桌面环境，如KDE或Gnome，足以在没有问题的情况下运行。尽管系统允许从一个环境切换到另一个环境，但这对用户来说都是优先考虑的问题。有些程序在一个环境中工作而在另一个环境中无法工作，因此它也可以被视为选择使用哪个环境的一个因素。

什么是交换空间?

交换空间是Linux使用的一定空间，用于临时保存一些并发运行的程序。当RAM 没有足够的内存来容纳正在执行的所有程序时，就会发生这种情况。

什么是root帐户

root帐户就像一个系统管理员帐户，允许你完全控制系统。你可以在此处创建和维护用户帐户，为每个帐户分配不同的权限。每次安装Linux时都是默认帐户。

什么是LILO?

LILO是Linux的引导加载程序。它主要用于将Linux操作系统加载到主内存中，以便它可以开始运行。什么是BASH?

BASH是Bourne Again SHell的缩写。它由Steve Bourne编写，作为原始

Bourne Shell（由/ bin / sh表示）的替代品。它结合了原始版本的Bourne

Shell的所有功能，以及其他功能，使其更容易使用。从那以后，它已被改编为运行Linux的大多数系统的默认shell。

什么是CLI?

命令行界面（英语：**command-line interface**，缩写]：CLI）是在图形用

户界面得到普及之前使用 为广泛的用户界面，它通常不支持鼠标，用户通过键

盘输入指令，计算机接收到指令后，予以执行。也有人称之为字符用户界面

（CUI）。

通常认为，命令行界面（CLI）没有图形用户界面（GUI）那么方便用户操作。

因为，命令行界面的软件通常需要用户记忆操作的命令，但是，由于其本身的特点，命令行界面要较图形用户界面节约计算机系统的资源。在熟记命令的前提下，使用命令行界面往往要较使用图形用户界面的操作速度要快。所以，图形用户界面的操作系统中，都保留着可选的命令行界面。

什么是GUI?

图形用户界面（Graphical User Interface，简称 GUI，又称图形用户接口）是指采用图形方式显示的计算机操作用户界面。

图形用户界面是一种人与计算机通信的界面显示格式，允许用户使用鼠标等输入设备操纵屏幕上的图标或菜单选项，以选择命令、调用文件、启动程序或执行其它一些日常任务。与通过键盘输入文本或字符命令来完成例行任务的字符界面相比，图形用户界面有许多优点。

开源的优势是什么?

开源允许你将软件（包括源代码）免费分发给任何感兴趣的人。然后，人们可以添加功能，甚至可以调试和更正源代码中的错误。它们甚至可以让它运行得更好，然后再次自由地重新分配这些增强的源代码。这 终使社区中的每个人受益。

GNU项目的重要性是什么?

这种所谓的自由软件运动具有多种优势，例如可以自由地运行程序以及根据你的需要自由学习和修改程序。它还允许你将软件副本重新分发给其他人，以及自由改进软件并将其发布给公众。

磁盘、目录、文件

简单 Linux 文件系统?

在 Linux 操作系统中，所有被操作系统管理的资源，例如网络接口卡、磁盘驱

动器、打印机、输入输出设备、普通文件或是目录都被看作是一个文件。

也就是说在 Linux 系统中有一个重要的概念：**一切都是文件**。其实这是

Unix 哲学的一个体现，而 Linux 是重写 Unix 而来，所以这个概念也就传承了下来。在 Unix 系统中，把一切资源都看作是文件，包括硬件设备。UNIX系统把每个硬件都看成是一个文件，通常称为设备文件，这样用户就可以用读写文件的方式实现对硬件的访问。

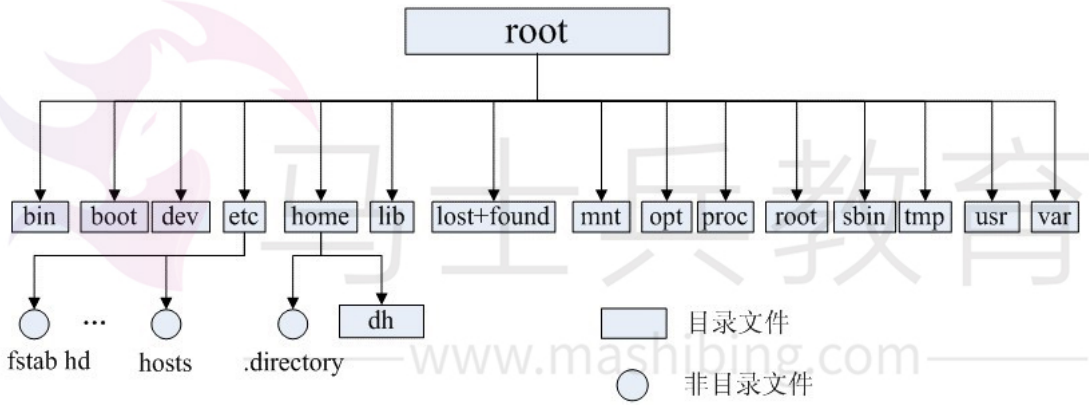
Linux 支持 5 种文件类型，如下图所示：

文件类型	描述	示例
普通文件	用来在辅助存储设备（如磁盘）上存储信息和数据	包含程序源代码（用C、C++、Java等语言所编写）、可执行程序、图片、声音、图像等
目录文件	用于表示和管理系统中的文件，目录文件中包含一些文件名和子目录名	/root、/home
链接文件	用于不同目录下文件的共享	当创建一个已存在文件的符号链接时，系统就创建一个链接文件，这个链接文件指向已存在的文件
设备文件	用来访问硬件设备	包括键盘、硬盘、光驱、打印机等
命名管道（FIFO）	是一种特殊类型的文件，Linux系统下，进程之间通信可以通过该文件完成	

Linux 的目录结构是怎样的?

这个问题，一般不会问。更多是实际使用时，需要知道。

Linux 文件系统的结构层次鲜明，就像一棵倒立的树， 顶层是其根目录：



常见目录说明：

- /bin： 存放二进制可执行文件(ls,cat,mkdir等)，常用命令一般都在这里；
- /etc： 存放系统管理和配置文件；
- /home： 存放所有用户文件的根目录，是用户主目录的基点，比如用户user的主目录就是/home/user，可以用~user表示；
- **/usr** ： 用于存放系统应用程序；
- /opt： 额外安装的可选应用程序包所放置的位置。一般情况下，我们可以把
- tomcat等都安装到这里；
- /proc： 虚拟文件系统目录，是系统内存的映射。可直接访问这个目录来获取系统信息；
- /root： 超级用户（系统管理员）的主目录（特权阶级o）；
- /sbin: 存放二进制可执行文件，只有root才能访问。这里存放的是系统管理员使用的系统级别的管理命令和程序。如ifconfig等；
- /dev： 用于存放设备文件；
- /mnt： 系统管理员安装临时文件系统的安装点，系统提供这个目录是让用户临时挂载其他的文件系统；
- /boot： 存放用于系统引导时使用的各种文件；
- **/lib** ： 存放着和系统运行相关的库文件 ；
- /tmp： 用于存放各种临时文件，是公用的临时文件存储点；
- /var： 用于存放运行时需要改变数据的文件，也是某些大文件的溢出区，比方说各种服务的日志文件（系统启动日志等。）等；
- /lost+found： 这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows下叫什么.chk）就在这里。

什么是 inode ?

一般来说，面试不会问 inode 。但是 inode 是一个重要概念，是理解 Unix/Linux 文件系统和硬盘储存的基础。

理解inode，要从文件储存说起。

文件储存在硬盘上，硬盘的 小存储单位叫做"扇区"（Sector）。每个扇区储存

512字节（相当于0.5KB）。

操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个"块"（block）。这种由多个扇区组成的"块"，是文件存取的 小单位。"块"的大小， 常见的是4KB，即连续八个 sector组成一个 block。

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做inode，中文译名为"索引节点"。

每一个文件都有对应的inode，里面包含了与该文件有关的一些信息。

简述 Linux 文件系统通过 i 节点把文件的逻辑结构和物理结构转换的工作过程?

如果看的一脸懵逼，也没关系。一般来说，面试官不太会问这个题目。

Linux 通过 inode 节点表将文件的逻辑结构和物理结构进行转换。

- inode 节点是一个 64 字节长的表，表中包含了文件的相关信息，其中有文件的大小、文件所有者、文件的存取许可方式以及文件的类型等重要信息。在 inode 节点表中 重要的内容是磁盘地址表。在磁盘地址表中有 13 个块号，文件将以块号在磁盘地址表中出现的顺序依次读取相应的块。
- Linux 文件系统通过把 inode 节点和文件名进行连接，当需要读取该文件时，文件系统在当前目录表中查找该文件名对应的项，由此得到该文件相对应的 inode 节点号，通过该 inode 节点的磁盘地址表把分散存放的文件物理块连接成文件的逻辑结构

什么是硬链接和软链接?

1)　硬链接

由于 Linux 下的文件是通过索引节点(inode)来识别文件，硬链接可以认为是一个指针，指向文件索引节点的指针，系统并不为它重新分配 inode 。每添加一个一个硬链接，文件的链接数就加 1 。

- 不足：1）不可以在不同文件系统的文件间建立链接；2）只有超级用户才可以为目录创建硬链接。

2)　软链接软链接克服了硬链接的不足，没有任何文件系统的限制，任何用户可以创建指向目录的符号链接。因而现在更为广泛使用，它具有更大的灵活性，甚至可以跨越不同机器、不同网络对文件进行链接。

- 不足：因为链接文件包含有原文件的路径信息，所以当原文件从一个目录下移到其他目录中，再访问链接文件，系统就找不到了，而硬链接就没有这个缺陷，你想怎么移就怎么移；还有它要系统分配额外的空间用于建立新的索引节点和保存原文件的路径。

实际场景下，基本是使用软链接。总结区别如下：

- 硬链接不可以跨分区，软件链可以跨分区。
- 硬链接指向一个 inode 节点，而软链接则是创建一个新的 inode 节点。
- 删除硬链接文件，不会删除原文件，删除软链接文件，会把原文件删除。

RAID 是什么?

RAID 全称为独立磁盘冗余阵列(Redundant Array of Independent Disks)，基本思想就是把多个相对便宜的硬盘组合起来，成为一个硬盘阵列组，使性能达到甚至超过一个价格昂贵、 容量巨大的硬盘。RAID 通常被用在服务器电脑上，使用完全相同的硬盘组成一个逻辑扇区，因此操作系统只会把它当做一个硬盘。

RAID 分为不同的等级，各个不同的等级均在数据可靠性及读写性能上做了不同的权衡。在实际应用中，可以依据自己的实际需求选择不同的 RAID 方案。

当然，因为很多公司都使用云服务，大家很难接触到 RAID 这个概念，更多的可能是普通云盘、SSD 云盘酱紫的概念。

安全

一台 Linux 系统初始化环境后需要做一些什么安全工作?

1、添加普通用户登陆，禁止 root 用户登陆，更改 SSH 端口号。

修改 SSH 端口不一定绝对哈。当然，如果要暴露在外网，建议改下。l

2、服务器使用密钥登陆，禁止密码登陆。

3、开启防火墙，关闭 SElinux ，根据业务需求设置相应的防火墙规则。

4、　装 fail2ban 这种防止 SSH 暴力破击的软件。

5、　设置只允许公司办公网出口 IP 能登陆服务器(看公司实际需要)

也可以安装 VPN 等软件，只允许连接 VPN 到服务器上。

6、　只允许有需要的服务器可以访问外网，其它全部禁止。

7、　做好软件层面的防护。

7.1 设置 nginx_waf 模块防止 SQL 注入。

7.2 把 Web 服务使用 www 用户启动，更改网站目录的所有者和所属组为 www 。

什么叫 CC 攻击? 什么叫 DDOS 攻击?

- CC 攻击，主要是用来攻击页面的，模拟多个用户不停的对你的页面进行访问，从而使你的系统资源消耗殆尽。
- DDOS 攻击，中文名叫分布式拒绝服务攻击，指借助服务器技术将多个计算机联合起来作为攻击平台，来对一个或多个目标发动 DDOS 攻击。

攻击，即是通过大量合法的请求占用大量网络资源，以达到瘫痪网络的目的。

怎么预防 CC 攻击和 DDOS 攻击? 防 CC、DDOS 攻击，这些只能是用硬件防火墙做流量清洗，将攻击流量引入黑洞。

流量清洗这一块，主要是买 ISP 服务商的防攻击的服务就可以，机房一般有空余流量，我们一般是买服务，毕竟攻击不会是持续长时间。

什么是网站数据库注入?

- 由于程序员的水平及经验参差不齐，大部分程序员在编写代码的时候，没有对用户输入数据的合法性进行判断。
- 应用程序存在安全隐患。用户可以提交一段数据库查询代码，根据程序返回的结果，获得某些他想得知的数据，这就是所谓的 SQL 注入。
- SQL注入，是从正常的 WWW 端口访问，而且表面看起来跟一般的 Web 页面访问没什么区别，如果管理员没查看日志的习惯，可能被入侵很长时间都不会发觉。

如何过滤与预防?

数据库网页端注入这种，可以考虑使用 nginx_waf 做过滤与预防。

Shell

本小节为选读。我也不太会写 Shell 脚本，都是写的时候，在网络上拼拼凑凑。。。

Shell 脚本是什么?

一个 Shell 脚本是一个文本文件，包含一个或多个命令。作为系统管理员，我们经常需要使用多个命令来完成一项任务，我们可以添加这些所有命令在一个文本文件(Shell 脚本)来完成这些日常工作任务。

什么是默认登录 Shell？在 Linux 操作系统，"/bin/bash" 是默认登录 Shell，是在创建用户时分配的。

使用 chsh 命令可以改变默认的 Shell 。示例如下所示：

```
1   ### chsh <用户名> -s <新shell>
2   ### chsh ThinkWon -s /bin/sh
```

在 Shell 脚本中，如何写入注释?

注释可以用来描述一个脚本可以做什么和它是如何工作的。每一行注释以 ## 开头。例子如下：

```
1 #!/bin/bash
2 ### This is a command
3  echo “I am logged in as $USER”
```

语法级

可以在 Shell 脚本中使用哪些类型的变量?

在 Shell 脚本，我们可以使用两种类型的变量：

- 系统定义变量
  系统变量是由系统系统自己创建的。这些变量通常由大写字母组成，可以通过 set 命令查看。
- 用户定义变量
  用户变量由系统用户来生成和定义，变量的值可以通过命令 "echo $<变量名>" 查看。
  Shell脚本中 $? 标记的用途是什么？在写一个 Shell 脚本时，如果你想要检查前一命令是否执行成功，在 if 条件中使用 $? 可以来检查前一命令的结束状态。
- 如果结束状态是 0 ，说明前一个命令执行成功。例如:

  ```
  1   root@localhost:~### ls /usr/bin/shar
  2   /usr/bin/shar
  3   root@localhost:~### echo $?
  4   0
  ```

- 如果结束状态不是0，说明命令执行失败。例如：

  ```
  1   root@localhost:~### ls /usr/bin/share
  2   ls: cannot access /usr/bin/share: No such file or directory 3 root@localhost:~### echo $?
  4 2
  ```

Bourne Shell(bash) 中有哪些特殊的变量？下面的表列出了 Bourne Shell 为命令行设置的特殊变量。

```
1   内建变量 解释
2   $0 命令行中的脚本名字
3   $1 第一个命令行参数
4   $2 第二个命令行参数
5   …... …….
6   $9 第九个命令行参数
7   $### 命令行参数的数量
8   $* 所有命令行参数，以空格隔开
```

如何取消变量或取消变量赋值？ unset 命令用于取消变量或取消变量赋值。语法如下所示：

```
1 ### unset <变量名>
```

Shell 脚本中 if 语法如何嵌套?

```
1  if [ 条件 ]
2  then
3  命令1
4  命令2
5 …..
6  else
7  if [ 条件 ]
8  then
9  命令1
10  命令2
11  ….
12  else
13  命令1
14  命令2
15  …..
16  fi
```

```
17  fi
```

在 Shell 脚本中如何比较两个数字？在 if-then 中使用测试命令（ -gt 等）来比较两个数字。例如：

```
1 #!/bin/bash
2 x=10
3 y=20
4 if [ $x -gt $y ]
5  then
6  echo “x is greater than y”
7  else
8  echo “y is greater than x”
9 fi
```

Shell 脚本中 case 语句的语法?

基础语法如下：

```
1   case 变量 in
2   值1)
3   命令1
4   命令2
5   …..
6   最后命令
!!
7
8   值2)
9   命令1
10  命令2
11  ……
12  最后命令
13  ;;
14  esac
```

Shell 脚本中 for 循环语法？基础语法如下：

```
1  for 变量 in 循环列表
2 do
3  命令1
4  命令2
5 ….
6  最后命令
7  done
```

Shell 脚本中 while 循环语法？如同 for 循环，while 循环只要条件成立就重复它的命令块。不同于 for循环，while 循环会不断迭代，直到它的条件不为真。基础语法：

```
1   while [ 条件 ]
2   do
3   命令…
4   done
```

do-while 语句的基本格式?

do-while 语句类似于 while 语句，但检查条件语句之前先执行命令（LCTT 译注：意即至少执行一次。）。下面是用 do-while 语句的语法：

```
1   do
2   {
3   命令
4   } while（条件）
```

Shell 脚本中 break 命令的作用?

break 命令一个简单的用途是退出执行中的循环。我们可以在 while 和 until 循环中使用 break 命令跳出循环。 Shell 脚本中 continue 命令的作用？ continue 命令不同于 break 命令，它只跳出当前循环的迭代，而不是整个循环。 continue 命令很多时候是很有用的，例如错误发生，但我们依然希望继续执行大循环的时候。

如何使脚本可执行?

使用 chmod 命令来使脚本可执行。例子如下：chmod a+x myscript.sh 。

#!/bin/bash 的作用?

#!/bin/bash 是 Shell 脚本的第一行，称为释伴（shebang）行。

这里 ## 符号叫做 hash ， 而 ! 叫做 bang。它的意思是命令通过 /bin/bash 来执行。

如何调试 Shell脚本?

使用 -x' 数（sh -x myscript.sh）可以调试 Shell脚本。另一个种方法是使用 -nv 参数(sh -nv myscript.sh)。

如何将标准输出和错误输出同时重定向到同一位置?

方法一：2>&1 (如### ls /usr/share/doc > out.txt 2>&1) 。

方法二：&> (如### ls /usr/share/doc &> out.txt) 。

在 Shell 脚本中，如何测试文件?

test 命令可以用来测试文件。基础用法如下表格：

```
Test 用法
-d 文件名 如果文件存在并且是目录，返回true
-e 文件名 如果文件存在，返回true
-f 文件名 如果文件存在并且是普通文件，返回true
-r 文件名 如果文件存在并可读，返回true
-s 文件名 如果文件存在并且不为空，返回true
-w 文件名 如果文件存在并可写，返回true
-x 文件名 如果文件存在并可执行，返回true
```

在 Shell 脚本如何定义函数呢?

函数是拥有名字的代码块。当我们定义代码块，我们就可以在我们的脚本调用函数名字，该块就会被执行。示例如下所示:

```
$ diskusage () { df -h ; }
译注：下面是我给的shell函数语法，原文没有
[ function ] 函数名 [()]
{
命令;
[return int;]
}
```

如何让 Shell 就脚本得到来自终端的输入?

read 命令可以读取来自终端（使用键盘）的数据。read 命令得到用户的输入并置于你给出的变量中。例子如下:

```
### vi /tmp/test.sh
!/
#bin/bash
echo ‘Please enter your name’
read name
echo “My Name is $name”
### ./test.sh
Please enter your name
ThinkWon
My Name is ThinkWon
```

如何执行算术运算?

有两种方法来执行算术运算:

1、使用 expr 命令：### expr 5 + 2 。

2、用一个美元符号和方括号（$[表达式]）： test=$[16 + 4] ; test=$[16 + 4]

编程题(。)

判断一文件是不是字符设备文件，如果是将其拷贝到 /dev 目录下?

```
#!/bin/bash
 read -p "Input file name: " FILENAME
if [ -c "$FILENAME" ];then
  cp $FILENAME /dev
fi
```

添加一个新组为 class1 ，然后添加属于这个组的 30 个用户，用户名的形式为 stdxx ， 其中 xx 从 01 到 30 ?

```
 #!/bin/bash
 groupadd class1
for((i=1;i<31;i++))
do
  if [ $i -le 10 ];then
  useradd -g class1 std0$i
  else
  useradd -g class1 std$i
  fi
  done
```

编写 Shell 程序，实现自动删除 50 个账号的功能，账号名为stud1 至 stud50?

```
#!/bin/bash
for((i=1;i<51;i++))
do
  userdel -r stud$i
 done
```

写一个 sed 命令，修改 /tmp/input.txt 文件的内容?

要求:

删除所有空行。

一行中，如果包含 “11111”，则在 “11111” 前面插入 “AAA”，在 “11111” 后面插入 “BBB” 。比如：将内容为 `000111112222 的一行改为 0000AAA11111BBB2222 。

```
[root@~]### cat -n /tmp/input.txt
1  000011111222
2
3  000011111222222
4  11111000000222
5
6
7  11111111111122222222222
8  2211111111
9  112222222
10 1122
11

### 删除所有空行命令
[root@~]### sed '/^$/d' /tmp/input.txt
000011111222
000011111222222
11111000000222
11111111111122222222222
2211111111
112222222
1122

### 插入指定的字符
[root@~]### sed 's#\(11111\)#AAA\1BBB#g' /tmp/input.txt
0000AAA11111BBB222
0000AAA11111BBB222222
AAA11111BBB000000222
AAA11111BBBAAA11111BBB11122222222222
22AAA11111BBB111
112222222
1122
```

实战如何选择 Linux 操作系统版本?

一般来讲，桌面用户首选 Ubuntu； 服务器首选 RHEL 或 CentOS ，两者中首选 CentOS 。

根据具体要求：

安全性要求较高，则选择 Debian 或者 FreeBSD 。

需要使用数据库高级服务和电子邮件网络应用的用户可以选择 SUSE 。

想要新技术新功能可以选择 Feddora ， Feddora 是 RHEL 和 CentOS 的一个测试版和预发布版本。

【重点】根据现有状况，绝大多数互联网公司选择 CentOS 。现在比较常用的是 6 系列，现在市场占有大概一半左右。另外的原因是 CentOS

更侧重服务器领域，并且无版权约束。

CentOS 7 系列，也慢慢使用的会比较多了。

如何规划一台 Linux 主机，步骤是怎样?

1、确定机器是做什么用的，比如是做 WEB 、DB、还是游戏服务器。

不同的用途，机器的配置会有所不同。

2、确定好之后，就要定系统需要怎么安装，默认安装哪些系统、分区怎么做。

3、需要优化系统的哪些参数，需要创建哪些用户等等的。

请问当用户反馈网站访问慢，你会如何处理?

有哪些方面的因素会导致网站网站访问慢?

1、服务器出口带宽不够用

- 本身服务器购买的出口带宽比较小。一旦并发量大的话，就会造成分给每个用户的出口带宽就小，访问速度自然就会慢。
- 跨运营商网络导致带宽缩减。例如，公司网站放在电信的网络上，那么客户这边对接是长城宽带或联通，这也可能导致带宽的缩减。

2、服务器负载过大，导致响应不过来

可以从两个方面入手分析：

- 分析系统负载，使用 w 命令或者 uptime 命令查看系统负载。如果负载很高，则使用 top 命令查看 CPU ， MEM 等占用情况，要么是 CPU 繁忙，要么是内存不够。
- 如果这二者都正常，再去使用 sar 命令分析网卡流量，分析是不是遭到了攻击。一旦分析出问题的原因，采取对应的措施解决，如决定要不要杀死一些进程，或者禁止一些访问等。

3、数据库瓶颈

- 如果慢查询比较多。那么就要开发人员或 DBA 协助进行 SQL 语句的优化。
- 如果数据库响应慢，考虑可以加一个数据库缓存，如 Redis 等。然后，也可以搭建 MySQL 主从，一台 MySQL 服务器负责写，其他几台从数据库负责读。

4、网站开发代码没有优化好

- 例如 SQL 语句没有优化，导致数据库读写相当耗时。

针对网站访问慢，怎么去排查?

1、首先要确定是用户端还是服务端的问题。当接到用户反馈访问慢，那边自己立即访问网站看看，如果自己这边访问快，基本断定是用户端问题，就需要耐心跟客户解释，协助客户解决问题。不要上来就看服务端的问题。一定要从源头开始，逐步逐步往下。

2、如果访问也慢，那么可以利用浏览器的调试功能，看看加载那一项数据消耗时间过多，是图片加载慢，还是某些数据加载慢。

3、针对服务器负载情况。查看服务器硬件(网络、CPU、内存)的消耗情况。如果是购买的云主机，比如阿里云，可以登录阿里云平台提供各方面的监控，比如 CPU、内存、带宽的使用情况。

4、如果发现硬件资源消耗都不高，那么就需要通过查日志，比如看看 MySQL慢查询的日志，看看是不是某条 SQL 语句查询慢，导致网站访问慢。怎么去解决?

1、 如果是出口带宽问题，那么久申请加大出口带宽。

2、 如果慢查询比较多，那么就要开发人员或 DBA 协助进行 SQL 语句的优化。

3、 如果数据库响应慢，考虑可以加一个数据库缓存，如 Redis 等等。然后也可以搭建MySQL 主从，一台 MySQL 服务器负责写，其他几台从数据库负责读。

4、 申请购买 CDN 服务，加载用户的访问。

5、 如果访问还比较慢，那就需要从整体架构上进行优化咯。做到专角色专用，多台服务器提供同一个服务。

Linux 性能调优都有哪几种方法?

1、Disabling daemons (关闭 daemons)。

2、Shutting down the GUI (关闭 GUI)。

3、Changing kernel parameters (改变内核参数)。

4、Kernel parameters (内核参数)。

5、Tuning the processor subsystem (处理器子系统调优)。

6、Tuning the memory subsystem (内存子系统调优)。

7、Tuning the file system (文件系统子系统调优)。

8、Tuning the network subsystem（网络子系统调优)。

文件管理命令 cat 命令

cat 命令用于连接文件并打印到标准输出设备上。

cat 主要有三大功能:

1.一次显示整个文件:

```
cat filename
```

2.从键盘创建一个文件

```
cat > filename
```

只能创建新文件,不能编辑已有文件

3.将几个文件合并为一个文件:

```
cat file1 file2 > file
```

- -b 对非空输出行号
- -n 输出所有行号实例:

（1） 把 log2012.log 的文件内容加上行号后输入 log2013.log 这个文件里

```
1  cat -n log2012.log log2013.log
```

（2） 把 log2012.log 和 log2013.log 的文件内容加上行号（空白行不加）之后将内容附加到 log.log 里

```
1  cat -b log2012.log log2013.log log.log
```

（3） 使用 here doc 生成新文件

```
1   cat >log.txt <<EOF
2   >Hello
3   >World
4   >PWD=$(pwd)
5   >EOF
6   ls -l log.txt
7   cat log.txt
8   Hello
9   World
=/
10  PWD opt/soft/test
```

（4） 反向列示

```
1   tac log.txt
=/
2   PWD opt/soft/test
3   World
4   Hello
```

chmod 命令

Linux/Unix 的文件调用权限分为三级 : 文件拥有者、群组、其他。利用 chmod 可以控制文件如何被他人所调用。

用于改变 linux 系统文件或目录的访问权限。用它控制文件或目录的访问权限。该命令有两种用法。一种是包含字母和操作符表达式的文字设定法；另一种是包含数字的数字设定法。

每一文件或目录的访问权限都有三组，每组用三位表示，分别为文件属主的读、写和执行权限；与属主同组的用户的读、写和执行权限；系统中其他用户的读、写和执行权限。可使用 ls -l test.txt 查找。

以文件 log2012.log 为例：

```
1 -rw-r--r-- 1 root root 296K 11-13 06:03 log2012.log
```

第一列共有 10 个位置，第一个字符指定了文件类型。在通常意义上，一个目录也是一个文件。如果第一个字符是横线，表示是一个非目录的文件。如果是 d，表示是一个目录。从第二个字符开始到第十个 9 个字符，3 个字符一组，分别表示了 3 组用户对文件或者目录的权限。权限字符用横线代表空许可，r 代表只读，w 代表写，x 代表可执行。常用参数：

```
1   -c 当发生改变时，报告处理信息
2   -R 处理指定目录以及其子目录下所有文件
```

权限范围：

```
1   u ：目录或者文件的当前的用户
2   g ：目录或者文件的当前的群组
3   o ：除了目录或者文件的当前用户或群组之外的用户或者群组
4   a ：所有的用户及群组
```

权限代号：

```
1   r ：读权限，用数字4表示
2   w ：写权限，用数字2表示
3   x ：执行权限，用数字1表示
4   - ：删除权限，用数字0表示
5   s ：特殊权限
```

实例：

（1） 增加文件 t.log 所有用户可执行权限

```
1  chmod a+x t.log
```

（2） 撤销原来所有的权限，然后使拥有者具有可读权限,并输出处理信息

```
1  chmod u=r t.log -c
```

（3） 给 file 的属主分配读、写、执行(7)的权限，给file的所在组分配读、执行

(5)的权限，给其他用户分配执行(1)的权限

```
1  chmod 751 t.log -c（或者：chmod u=rwx,g=rx,o=x t.log -c)
```

（4） 将 test 目录及其子目录所有文件添加可读权限

```
1  chmod u+r,g+r,o+r -R text/ -c
```

chown 命令

chown 将指定文件的拥有者改为指定的用户或组，用户可以是用户名或者用户 ID；组可以是组名或者组 ID；文件是以空格分开的要改变权限的文件列表，支持通配符。

```
1   -c 显示更改的部分的信息
2   -R 处理指定目录及子目录
```

实例：

（1） 改变拥有者和群组 并显示改变信息

```
1  chown -c mail:mail log2012.log
```

（1） 改变文件群组

```
1  chown -c :mail t.log
```

（2） 改变文件夹及子文件目录属主及属组为 mail

```
1  chown -cR mail: test/
```

cp 命令

将源文件复制至目标文件，或将多个源文件复制至目标目录。

注意：命令行复制，如果目标文件已经存在会提示是否覆盖，而在 shell 脚本中，如果不加 -i 参数，则不会提示，而是直接覆盖！

```
1   -i 提示
2   -r 复制目录及目录内所有项目
3   -a 复制的文件与原文件时间一样
```

实例：

（1） 复制 a.txt 到 test 目录下，保持原文件时间，如果原文件存在提示是否覆盖。

```
1  cp -ai a.txt test
```

（2） 为 a.txt 建议一个链接（快捷方式）

```
1  cp -s a.txt link_a.txt
```

find 命令

用于在文件树中查找文件，并作出相应的处理。

命令格式：

```
1 find pathname -options [-print -exec -ok ...]
```

命令参数：

```
1   pathname: find命令所查找的目录路径。例如用.来表示当前目录，用/来表示系统根目录。
2   -print： find命令将匹配的文件输出到标准输出。
3   -exec： find命令对匹配的文件执行该参数所给出的shell命令。相应命令的形式为'comm and' { } \;，注意{ }和\；之间的空格。
4   -ok： 和-exec的作用相同，只不过以一种更为安全的模式来执行该参数所给出的shell命令，在执行每一个命令之前，都会给出提示，让用户来确定是否执行。
```

命令选项：

```
1   -name 按照文件名查找文件
2   -perm 按文件权限查找文件
3   -user 按文件属主查找文件
4   -group 按照文件所属的组来查找文件。
5   -type 查找某一类型的文件，诸如：
6   b - 块设备文件
7   d - 目录
8   c - 字符设备文件
9   l - 符号链接文件
10  p - 管道文件
11  f - 普通文件
```

实例：

（1） 查找 48 小时内修改过的文件

```
1  find -atime -2
```

（2） 在当前目录查找 以 .log 结尾的文件。 . 代表当前目录

```
1  find ./ -name '*.log'
```

（3） 查找 /opt 目录下 权限为 777 的文件

```
1  find /opt -perm 777
```

（4） 查找大于 1K 的文件

```
1  find -size +1000c
```

查找等于 1000 字符的文件

```
1 find -size 1000c
```

-exec 参数后面跟的是 command 命令，它的终止是以 ; 为结束标志的，所以

这句命令后面的分号是不可缺少的，考虑到各个系统中分号会有不同的意义，所以前面加反斜杠。{} 花括号代表前面find查找出来的文件名。

head 命令

head 用来显示档案的开头至标准输出中，默认 head 命令打印其相应文件的开头 10 行。常用参数：

```
1 -n<行数> 显示的行数（行数为复数表示从最后向前数）
```

实例：

（1） 显示 1.log 文件中前 20 行

```
1  head 1.log -n 20
```

（2） 显示 1.log 文件前 20 字节

```
1  head -c 20 log2014.log
```

（3） 显示 t.log 后 10 行

```
1  head -n -10 t.log
```

less 命令

less 与 more 类似，但使用 less 可以随意浏览文件，而 more 仅能向前移动，却不能向后移动，而且 less 在查看之前不会加载整个文件。常用命令参数：

```
1   -i 忽略搜索时的大小写
2   -N 显示每行的行号
3   -o <文件名> 将less 输出的内容在指定文件中保存起来
4   -s 显示连续空行为一行
5   /字符串：向下搜索“字符串”的功能
```

```
6   ?字符串：向上搜索“字符串”的功能
7   n：重复前一个搜索（与 / 或 ? 有关）
8   N：反向重复前一个搜索（与 / 或 ? 有关）
9   -x <数字> 将“tab”键显示为规定的数字空格
10  b 向后翻一页
11  d 向后翻半页
12  h 显示帮助界面
13  Q 退出less 命令
14  u 向前滚动半页
15  y 向前滚动一行
16  空格键 滚动一行
17  回车键 滚动一页
18  [pagedown]： 向下翻动一页
19  [pageup]： 向上翻动一页
```

实例：

（1） ps 查看进程信息并通过 less 分页显示

```
1  ps -aux | less -N
```

（2） 查看多个文件

```
1  less 1.log 2.log
```

可以使用 n 查看下一个，使用 p 查看前一个。

ln 命令

功能是为文件在另外一个位置建立一个同步的链接，当在不同目录需要该问题时，就不需要为每一个目录创建同样的文件，通过 ln 创建的链接（link）减少磁盘占用量。

链接分类：软件链接及硬链接软链接：

1.软链接，以路径的形式存在。类似于Windows操作系统中的快捷方式

2.软链接可以 跨文件系统 ，硬链接不可以

3.软链接可以对一个不存在的文件名进行链接

4.软链接可以对目录进行链接硬链接:

1.硬链接，以文件副本的形式存在。但不占用实际空间。

2.不允许给目录创建硬链接

3.硬链接只有在同一个文件系统中才能创建需要注意：

第一：ln命令会保持每一处链接文件的同步性，也就是说，不论你改动了哪一处，其它的文件都会发生相同的变化；

第二：ln的链接又分软链接和硬链接两种，软链接就是ln –s 源文件 目标文件，它只会在你选定的位置上生成一个文件的镜像，不会占用磁盘空间，硬链接 ln 源文件 目标文件，没有参数-s， 它会在你选定的位置上生成一个和源文件大小相同的文件，无论是软链接还是硬链接，文件都保持同步变化。

第三：ln指令用在链接文件或目录，如同时指定两个以上的文件或目录，且 后的目的地是一个已经存在的目录，则会把前面指定的所有文件或目录复制到该目录中。若同时指定多个文件或目录，且 后的目的地并非是一个已存在的目录，则会出现错误信息。

常用参数：

```
1   -b 删除，覆盖以前建立的链接
2   -s 软链接（符号链接）
3   -v 显示详细处理过程
```

实例：

（1） 给文件创建软链接，并显示操作信息

```
1  ln -sv source.log link.log
```

（2） 给文件创建硬链接，并显示操作信息

```
1  ln -v source.log link1.log
```

（3） 给目录创建软链接

```
1  ln -sv /opt/soft/test/test3 /opt/soft/test/test5
```

locate 命令

locate 通过搜寻系统内建文档数据库达到快速找到档案，数据库由 updatedb

程序来更新，updatedb 是由 cron daemon 周期性调用的。默认情况下

locate 命令在搜寻数据库时比由整个由硬盘资料来搜寻资料来得快，但较差劲的是 locate 所找到的档案若是 近才建立或 刚更名的，可能会找不到，在内定值中，updatedb 每天会跑一次，可以由修改 crontab 来更新设定值

(etc/crontab)。

locate 与 find 命令相似，可以使用如 *、? 等进行正则匹配查找常用参数：

```
1   -l num（要显示的行数）
2   -f 将特定的档案系统排除在外，如将proc排除在外
3   -r 使用正则运算式做为寻找条件
```

实例：

（1） 查找和 pwd 相关的所有文件(文件名中包含 pwd)

```
1 locate pwd
```

（2） 搜索 etc 目录下所有以 sh 开头的文件

```
1 locate /etc/sh
```

（3） 查找 /var 目录下，以 reason 结尾的文件

```
1 locate -r '^/var.*reason$'（其中.表示一个字符，*表示任务多个；.*表示任意多个字符）
```

more 命令

功能类似于 cat, more 会以一页一页的显示方便使用者逐页阅读，而 基本的指令就是按空白键（space）就往下一页显示，按 b 键就会往回（back）一页显示。

命令参数：

```
1 +n 从第 n 行开始显示
2 -n 定义屏幕大小为n行
3 +/pattern 在每个档案显示前搜寻该字串（pattern），然后从该字串前两行之后开始显示
4 -c 从顶部清屏，然后显示
5 -d 提示“Press space to continue，’q’ to quit（按空格键继续，按q键退出）”，禁用响铃功能
6 -l 忽略Ctrl+l（换页）字符
7 -p 通过清除窗口而不是滚屏来对文件进行换页，与-c选项相似
8 -s 把连续的多个空行显示为一行
9 -u 把文件内容中的下画线去掉
```

常用操作命令：

```
1 Enter 向下 n 行，需要定义。默认为 1 行
2 Ctrl+F 向下滚动一屏
3 空格键 向下滚动一屏
4 Ctrl+B 返回上一屏
5 = 输出当前行的行号
6 :f 输出文件名和当前行的行号
7 V 调用vi编辑器
8 !命令 调用Shell，并执行命令
9 q 退出more
```

实例：

（1） 显示文件中从第3行起的内容

```
1 more +3 text.txt
```

（2） 在所列出文件目录详细信息，借助管道使每次显示 5 行

```
1 ls -l |more -5
```

按空格显示下 5 行。

mv 命令

移动文件或修改文件名，根据第二参数类型（如目录，则移动文件；如为文件则重命令该文件）。

当第二个参数为目录时，第一个参数可以是多个以空格分隔的文件或目录，然后移动第一个参数指定的多个文件到第二个参数指定的目录中。

实例：

（1） 将文件 test.log 重命名为 test1.txt

```
1 mv test.log test1.txt
```

（2） 将文件 log1.txt,log2.txt,log3.txt 移动到根的 test3 目录中

```
1 mv llog1.txt log2.txt log3.txt /test3
```

（3） 将文件 file1 改名为 file2，如果 file2 已经存在，则询问是否覆盖

```
1 mv -i log1.txt log2.txt
```

（4） 移动当前文件夹下的所有文件到上一级目录

```
1 mv * ../
```

rm 命令

删除一个目录中的一个或多个文件或目录，如果没有使用 -r 选项，则 rm 不会删除目录。如果使用 rm 来删除文件，通常仍可以将该文件恢复原状。

```
1 rm[选项] 文件…
```

实例：

（1） 删除任何 .log 文件，删除前逐一询问确认：

```
1 rm -i *.log
```

（2） 删除 test 子目录及子目录中所有档案删除，并且不用一一确认：

```
1 rm -rf test
```

（3） 删除以 -f 开头的文件

```
1  rm -- -f*
```

tail 命令

用于显示指定文件末尾内容，不指定文件时，作为输入信息进行处理。常用查看日志文件。常用参数：

```
1   -f 循环读取（常用于查看递增的日志文件）
2   -n<行数> 显示行数（从后向前）
```

（1）循环读取逐渐增加的文件内容

```
1 ping 127.0.0.1 > ping.log &
```

后台运行：可使用 jobs -l 查看，也可使用 fg 将其移到前台运行。

```
1 tail -f ping.log
```

（查看日志） touch 命令

Linux touch命令用于修改文件或者目录的时间属性，包括存取时间和更改时间。若文件不存在，系统会建立一个新的文件。

ls -l 可以显示档案的时间记录。

语法

```
1 touch [-acfm][-d<日期时间>][-r<参考文件或目录>] [-t<日期时间>][--help][--ver sion][文件或目录…]
```

参数说明：

- a 改变档案的读取时间记录。 m 改变档案的修改时间记录。 c 假如目的档案不存在，不会建立新的档案。与 --no-create 的效果一样。
- f 不使用，是为了与其他 unix 系统的相容性而保留。
- r 使用参考档的时间记录，与 --file 的效果一样。
- d 设定时间与日期，可以使用各种不同的格式。
- t 设定档案的时间记录，格式与 date 指令相同。
- –no-create 不会建立新档案。
- –help 列出指令格式。
- –version 列出版本讯息。

实例

使用指令"touch"修改文件"testfile"的时间属性为当前系统时间，输入如下命令：

```
1 $ touch testfile #修改文件的时间属性
```

首先，使用ls命令查看testfile文件的属性，如下所示：

```
1   $ ls -l testfile #查看文件的时间属性
2   #原来文件的修改时间为16:09
3   -rw-r--r-- 1 hdd hdd 55 2011-08-22 16:09 testfile
```

执行指令"touch"修改文件属性以后，并再次查看该文件的时间属性，如下所示：

```
1   $ touch testfile #修改文件时间属性为当前系统时间
2   $ ls -l testfile #查看文件的时间属性
3   #修改后文件的时间属性为当前系统时间
4   -rw-r--r-- 1 hdd hdd 55 2011-08-22 19:53 testfile
```

使用指令"touch"时，如果指定的文件不存在，则将创建一个新的空白文件。例如，在当前目录下，使用该指令创建一个空白文件"file"，输入如下命令：

```
1 $ touchfile#创建一个名为“file”的新的空白文件
```

vim 命令

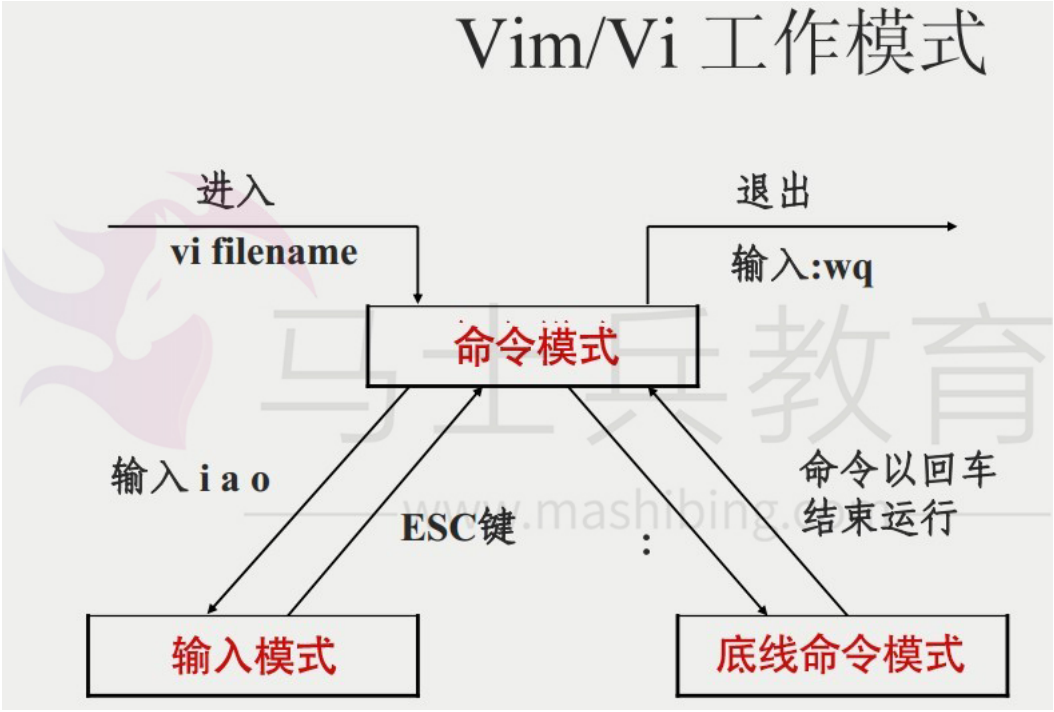
Vim是从 vi 发展出来的一个文本编辑器。代码补完、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。

打开文件并跳到第 10 行：vim +10 filename.txt 。

打开文件跳到第一个匹配的行：vim +/search-term filename.txt 。

以只读模式打开文件：vim -R /etc/passwd 。

基本上 vi/vim 共分为三种模式，分别是命令模式（Command mode），输入模式（Insert mode）和底线命令模式（Last line mode）。简单的说，我们可以将这三个模式想成底下的图标来表示：



whereis 命令

whereis 命令只能用于程序名的搜索，而且只搜索二进制文件（参数-b）、

man说明文件（参数-m）和源代码文件（参数-s）。如果省略参数，则返回所有信息。whereis 及 locate 都是基于系统内建的数据库进行搜索，因此效率很高，而find则是遍历硬盘查找文件。

常用参数：

```
1   -b 定位可执行文件。
2   -m 定位帮助文件。
3   -s 定位源代码文件。
4   -u 搜索默认路径下除可执行文件、源代码文件、帮助文件以外的其它文件。
```

实例：

（1） 查找 locate 程序相关文件

```
1  whereislocate
```

（2） 查找 locate 的源码文件

```
1  whereis -s locate
```

（3） 查找 lcoate 的帮助文件

```
1  whereis -m locate
```

which 命令

在 linux 要查找某个文件，但不知道放在哪里了，可以使用下面的一些命令来搜索：

```
1   which 查看可执行文件的位置。
2   whereis 查看文件的位置。
3   locate 配合数据库查看文件位置。
4   find 实际搜寻硬盘查询文件名称。
```

which 是在 PATH 就是指定的路径中，搜索某个系统命令的位置，并返回第一个搜索结果。使用 which 命令，就可以看到某个系统命令是否存在，以及执行的到底是哪一个位置的命令。

常用参数：

```
1 -n 指定文件名长度，指定的长度必须大于或等于所有文件中最长的文件名。
```

实例：

（1） 查看 ls 命令是否存在，执行哪个

```
1  whichls
```

（2） 查看 which

```
1  whichwhich
```

（3） 查看 cd

```
1  which cd（显示不存在，因为 cd 是内建命令，而 which 查找显示是 PATH 中的命令）
```

查看当前 PATH 配置：

`1 echo$PATH

或使用 env 查看所有环境变量及对应值文档编辑命令 grep 命令

强大的文本搜索命令，grep(Global Regular Expression Print) 全局正则表达式搜索。

grep 的工作方式是这样的，它在一个或多个文件中搜索字符串模板。如果模板包括空格，则必须被引用，模板后的所有字符串被看作文件名。搜索的结果被送到标准输出，不影响原文件内容。

命令格式：

```
grep [option] pattern file|dir
```

常用参数：

```
1   -A n --after-context显示匹配字符后n行
2   -B n --before-context显示匹配字符前n行
3   -C n --context 显示匹配字符前后n行
4   -c --count 计算符合样式的列数
5   -i 忽略大小写
6   -l 只列出文件内容符合指定的样式的文件名称
7   -f 从文件中读取关键词
8   -n 显示匹配内容的所在文件中行数
9   -R 递归查找文件夹
```

grep 的规则表达式:

```
 ^ #锚定行的开始 如：'^grep'匹配所有以grep开头的行。 2 $ #锚定行的结束 如：'grep$'匹配所有以grep结尾的行。
3   . #匹配一个非换行符的字符 如：'gr.p'匹配gr后接一个任意字符，然后是p。
4   * #匹配零个或多个先前字符 如：'*grep'匹配所有一个或多个空格后紧跟grep的行。
5   .* #一起用代表任意字符。
6   [] #匹配一个指定范围内的字符，如'[Gg]rep'匹配Grep和grep。
7   [^] #匹配一个不在指定范围内的字符，如：'[^A-FH-Z]rep'匹配不包含A-R和T-Z的一个
字母开头，紧跟rep的行。
8   \(..\) #标记匹配字符，如'\(love\)'，love被标记为1。
9   \< #锚定单词的开始，如:'\<grep'匹配包含以grep开头的单词的行。
10  \> #锚定单词的结束，如'grep\>'匹配包含以grep结尾的单词的行。
11  x\{m\} #重复字符x，m次，如：'O\{5\}'匹配包含5个o的行。
12  x\{m,\} #重复字符x,至少m次，如：'o\{5,\}'匹配至少有5个o的行。
13  x\{m,n\} #重复字符x，至少m次，不多于n次，如：'o\{5,10\}'匹配5--10个o的行。
14  \w #匹配文字和数字字符，也就是[A-Za-z0-9]，如：'G\w*p'匹配以G后跟零个或多个文字或数字字符，然后是p。
15  \W #\w的反置形式，匹配一个或多个非单词字符，如点号句号等。
16  \b #单词锁定符，如: '\bgrep\b'只匹配grep。
```

实例：（1）查找指定进程

```
1 ps -ef |grep svn
```

（2） 查找指定进程个数

```
1  ps -ef |grep svn -c
```

（3） 从文件中读取关键词

```
1  cat test1.txt |grep -f key.log
```

（4） 从文件夹中递归查找以grep开头的行，并只列出文件

```
1  grep -lR '^grep' /tmp
```

（5） 查找非x开关的行内容

```
1  grep'^[^x]' test.txt
```

（6） 显示包含 ed 或者 at 字符的内容行

```
1  grep -E 'ed|at' test.txt
```

wc 命令

wc(word count)功能为统计指定的文件中字节数、字数、行数，并将统计结果输出命令格式：

```
1 wc [option] file..
```

命令参数：

```
1   -c 统计字节数
2   -l 统计行数
3   -m 统计字符数
4   -w 统计词数，一个字被定义为由空白、跳格或换行字符分隔的字符串
```

实例：

（1） 查找文件的 行数 单词数 字节数 文件名

```
1  wc text.txt
```

结果：

```
1 7 8 70 test.txt
```

（2） 统计输出结果的行数

```
1  cat test.txt |wc -l
```

磁盘管理命令 cd 命令

cd(changeDirectory) 命令语法：

```
1 cd [目录名]
```

说明：切换当前目录至 dirName。

实例：（1）进入要目录

```
1 cd /
```

（2） 进入 “home” 目录

```
1  cd ~
```

（3） 进入上一次工作路径

```
1  cd -
```

（4） 把上个命令的参数作为cd参数使用。

```
1  cd!$
```

df 命令

显示磁盘空间使用情况。获取硬盘被占用了多少空间，目前还剩下多少空间等信息，如果没有文件名被指定，则所有当前被挂载的文件系统的可用空间将被显

示。默认情况下，磁盘空间将以 1KB 为单位进行显示，除非环境变量

POSIXLY_CORRECT 被指定，那样将以512字节为单位进行显示：

```
1   -a 全部文件系统列表
2   -h 以方便阅读的方式显示信息
3   -i 显示inode信息
4   -k 区块为1024字节
5   -l 只显示本地磁盘
6   -T 列出文件系统类型
```

实例：

（1） 显示磁盘使用情况

1 df -l

（2） 以易读方式列出所有文件系统及其类型

1 df -haT

du 命令

du 命令也是查看使用空间的，但是与 df 命令不同的是 Linux du 命令是对文件和目录磁盘使用的空间的查看：命令格式：

1 du [选项] [文件]

常用参数：

```
1   -a 显示目录中所有文件大小
2   -k 以KB为单位显示文件大小
3   -m 以MB为单位显示文件大小
4   -g 以GB为单位显示文件大小
5   -h 以易读方式显示文件大小
6   -s 仅显示总计
7   -c或--total 除了显示个别目录或文件的大小外，同时也显示所有目录或文件的总和
```

实例：

（1） 以易读方式显示文件夹内及子文件夹大小

```
1  du -h scf/
```

（2） 以易读方式显示文件夹内所有文件大小

```
1  du -ah scf/
```

（3） 显示几个文件或目录各自占用磁盘空间的大小，还统计它们的总和

```
1  du -hc test/ scf/
```

（4） 输出当前目录下各个子目录所使用的空间

```
1  du -hc --max-depth=1 scf/
```

ls命令

就是 list 的缩写，通过 ls 命令不仅可以查看 linux 文件夹包含的文件，而且可以查看文件权限(包括目录、文件夹、文件权限) 查看目录信息等等。

常用参数搭配：

```
ls -a 列出目录所有文件，包含以.开始的隐藏文件
ls -A 列出除.及..的其它文件
ls -r 反序排列
ls -t 以文件修改时间排序
ls -S 以文件大小排序
ls -h 以易读大小显示
ls -l 除了文件名之外，还将文件的权限、所有者、文件大小等信息详细列出来
```

实例：

(1) 按易读方式按时间反序排序，并显示文件详细信息

```
ls -lhrt
```

(2) 按大小反序显示文件详细信息

```
ls -lrS
```

(3)列出当前目录中所有以"t"开头的目录的详细内容

```
ls -l t*
```

(4) 列出文件绝对路径（不包含隐藏文件）

```
ls|sed"s:^: pwd /:"
```

(5) 列出文件绝对路径（包含隐藏文件）

```
find$pwd -maxdepth 1 |xargsls -ld
```

mkdir 命令

mkdir 命令用于创建文件夹。

可用选项：

-m: 对新建目录设置存取权限，也可以用 chmod 命令设置;

-p: 可以是一个路径名称。此时若路径中的某些目录尚不存在,加上此选项后，系统将自动建立好那些尚不在的目录，即一次可以建立多个目录。

实例：

（1） 当前工作目录下创建名为 t的文件夹

```
mkdir t
```

（2） 在 tmp 目录下创建路径为 test/t1/t 的目录，若不存在，则创建：

```
mkdir -p /tmp/test/t1/t
```

pwd 命令

pwd 命令用于查看当前工作目录路径。

实例：

（1） 查看当前路径

```
pwd
```

（2） 查看软链接的实际路径

```
pwd
```

rmdir 命令

从一个目录中删除一个或多个子目录项，删除某目录时也必须具有对其父目录的写权限。

注意：不能删除非空目录实例：

（1）当 parent 子目录被删除后使它也成为空目录的话，则顺便一并删除：

```
rmdir -p parent/child/child11
```

网络通讯命令 ifconfig 命令

- ifconfig 用于查看和配置 Linux 系统的网络接口。
- 查看所有网络接口及其状态：ifconfig -a 。
- 使用 up 和 down 命令启动或停止某个接口：ifconfig eth0 up 和 ifconfig eth0 down 。

iptables 命令

iptables ，是一个配置 Linux 内核防火墙的命令行工具。功能非常强大，对于

我们开发来说，主要掌握如何开放端口即可。例如：

把来源 IP 为 192.168.1.101 访问本机 80 端口的包直接拒绝： iptables -I INPUT -s 192.168.1.101 -p tcp --dport 80 -j REJECT 。

开启 80 端口，因为web对外都是这个端口

iptables -A INPUT -p tcp --dport 80 -j ACCEP

```
1
```

另外，要注意使用 iptables save 命令，进行保存。否则，服务器重启后，配置的规则将丢失。

netstat 命令

Linux netstat命令用于显示网络状态。

利用netstat指令可让你得知整个Linux系统的网络情况。语法

```
1 netstat [-acCeFghilMnNoprstuvVwx][-A<网络类型>][--ip]
```

参数说明：

- -a或–all 显示所有连线中的Socket。
- -A<网络类型>或–<网络类型> 列出该网络类型连线中的相关地址。
- -c或–continuous 持续列出网络状态。
- -C或–cache 显示路由器配置的快取信息。
- -e或–extend 显示网络其他相关信息。
- -F或–fib 显示FIB。
- -g或–groups 显示多重广播功能群组组员名单。
- -h或–help 在线帮助。
- -i或–interfaces 显示网络界面信息表单。
- -l或–listening 显示监控中的服务器的Socket。
- -M或–masquerade 显示伪装的网络连线。
- -n或–numeric 直接使用IP地址，而不通过域名服务器。
- -N或–netlink或–symbolic 显示网络硬件外围设备的符号连接名称。
- -o或–timers 显示计时器。
- -p或–programs 显示正在使用Socket的程序识别码和程序名称。
- -r或–route 显示Routing Table。
- -s或–statistice 显示网络工作信息统计表。
- -t或–tcp 显示TCP传输协议的连线状况。
- -u或–udp 显示UDP传输协议的连线状况。
- -v或–verbose 显示指令执行过程。
- -V或–version 显示版本信息。
- -w或–raw 显示RAW传输协议的连线状况。
- -x或–unix 此参数的效果和指定"-A unix"参数相同。
- –ip或–inet 此参数的效果和指定"-A inet"参数相同。

实例如何查看系统都开启了哪些端口?

```
1   [root@centos6 ~ 13:20 #55]## netstat -lnp
2   Active Internet connections (only servers)
3   Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name
4   tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN 1035/sshd
5   tcp 0 0 :::22 :::* LISTEN 1035/sshd
6   udp 0 0 0.0.0.0:68 0.0.0.0:* 931/dhclient 7 Active UNIX domain sockets (only servers)
8   Proto RefCnt Flags Type State I-Node PID/Program name Path
9   unix 2 [ ACC ] STREAM LISTENING 6825 1/init @/com/ubuntu/upstart
10  unix 2 [ ACC ] STREAM LISTENING 8429 1003/dbus-daemon /var/run/dbus/syst em_bus_socket
```

如何查看网络连接状况?

```
1   [root@centos6 ~ 13:22 #58]## netstat -an
2   Active Internet connections (servers and established)
3   Proto Recv-Q Send-Q Local Address Foreign Address State
4   tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN
5   tcp 0 0 192.168.147.130:22 192.168.147.1:23893 ESTABLISHED
6   tcp 0 0 :::22 :::* LISTEN
7   udp 0 0 0.0.0.0:68 0.0.0.0:*
```

如何统计系统当前进程连接数？输入命令 netstat -an | grep ESTABLISHED | wc -l 。输出结果 177 。一共有 177 连接数。用 netstat 命令配合其他命令，按照源 IP 统计所有到 80 端口的

ESTABLISHED 状态链接的个数？严格来说，这个题目考验的是对 awk 的使用。

首先，使用 netstat -an|grep ESTABLISHED 命令。结果如下：

```
1 tcp 0 0 120.27.146.122:80 113.65.18.33:62721 ESTABLISHED 2 tcp 0 0 120.27.146.122:80 27.43.83.115:47148 ESTABLISHED
3   tcp 0 0 120.27.146.122:58838 106.39.162.96:443 ESTABLISHED
4   tcp 0 0 120.27.146.122:52304 203.208.40.121:443 ESTABLISHED
5   tcp 0 0 120.27.146.122:33194 203.208.40.122:443 ESTABLISHED
6   tcp 0 0 120.27.146.122:53758 101.37.183.144:443 ESTABLISHED
7   tcp 0 0 120.27.146.122:27017 23.105.193.30:50556 ESTABLISHED
```

ping 命令

Linux ping命令用于检测主机。

执行ping指令会使用ICMP传输协议，发出要求回应的信息，若远端主机的网络功能没有问题，就会回应该信息，因而得知该主机运作正常。指定接收包的次数

```
1 ping -c 2 www.baidu.com
```

telnet 命令

Linux telnet命令用于远端登入。执行telnet指令开启终端机阶段作业，并登入远端主机。语法

```
1 telnet [-8acdEfFKLrx][-b<主机别名>][-e<脱离字符>][-k<域名>][-l<用户名称>][n<记录文件>][-S<服务类型>][-X<认证形态>][主机名称或IP地址<通信端口>]
```

参数说明:

- 8 允许使用8位字符资料，包括输入与输出。 -a 尝试自动登入远端系统。
- -b<主机别名> 使用别名指定远端主机名称。
- -c 不读取用户专属目录里的.telnetrc文件。
- -d 启动排错模式。
- -e<脱离字符> 设置脱离字符。
- -E 滤除脱离字符。
- -f 此参数的效果和指定"-F"参数相同。
- -F 使用Kerberos V5认证时，加上此参数可把本地主机的认证数据上传到远端主机。
- -k<域名> 使用Kerberos认证时，加上此参数让远端主机采用指定的领域名，而非该主机的域名。
- -K 不自动登入远端主机。
- -l<用户名称> 指定要登入远端主机的用户名称。
- -L 允许输出8位字符资料。
- -n<记录文件> 指定文件记录相关信息。
- -r 使用类似rlogin指令的用户界面。
- -S<服务类型> 设置telnet连线所需的IP TOS信息。
- -x 假设主机有支持数据加密的功能，就使用它。
- -X<认证形态> 关闭指定的认证形态。

实例

登录远程主机

```
1   ## 登录IP为 192.168.0.5 的远程主机
2   telnet 192.168.0.5
```

系统管理命令 date 命令

显示或设定系统的日期与时间。

命令参数:

```
1   -d<字符串>   显示字符串所指的日期与时间。字符串前后必须加上双引号。
2   -s<字符串>   根据字符串来设置日期与时间。字符串前后必须加上双引号。
3   -u   显示GMT。
4   %H 小时(00-23)
5   %I 小时(00-12)
6   %M 分钟(以00-59来表示)
7   %s 总秒数。起算时间为1970-01-01 00:00:00 UTC。
8   %S 秒(以本地的惯用法来表示) 9 %a 星期的缩写。
10  %A 星期的完整名称。
11  %d 日期(以01-31来表示)。
12  %D 日期(含年月日)。
13  %m 月份(以01-12来表示)。
14  %y 年份(以00-99来表示)。
15  %Y 年份(以四位数来表示)。
```

实例:

（1） 显示下一天

```
1 date +%Y%m%d --date="+1 day" //显示下一天的日期
```

（2） -d参数使用

```
1  date -d "nov 22" 今年的 11 月 22 日是星期三
2  date -d '2 weeks' 2周后的日期
3  date -d 'next monday' (下周一的日期)
4  date -d next-day +%Y%m%d（明天的日期）或者：date -d tomorrow +%Y%m%d
5  date -d last-day +%Y%m%d(昨天的日期) 或者：date -d yesterday +%Y%m%d
6  date -d last-month +%Y%m(上个月是几月)
7  date -d next-month +%Y%m(下个月是几月)
```

free 命令

显示系统内存使用情况，包括物理内存、交互区内存(swap)和内核缓冲区内存。

命令参数：

```
-b 以Byte显示内存使用情况
-k 以kb为单位显示内存使用情况
-m 以mb为单位显示内存使用情况
-g 以gb为单位显示内存使用情况
-s<间隔秒数> 持续显示内存
-t 显示内存使用总合
```

实例：

（1） 显示内存使用情况

```
free
free -k
free -m
```

（1） 以总和的形式显示内存的使用信息

```
free -t
```

（2） 周期性查询内存使用情况

```
free -s 10
```

kill 命令

发送指定的信号到相应进程。不指定型号将发送SIGTERM（15）终止指定进

程。如果任无法终止该程序可用"-KILL" 参数，其发送的信号为SIGKILL(9) ，将强制结束进程，使用ps命令或者jobs 命令可以查看进程号。root用户将影响用户的进程，非root用户只能影响自己的进程。

常用参数：

```
-l 信号，若果不加信号的编号参数，则使用“-l”参数会列出全部的信号名称
-a 当处理当前进程时，不限制命令名和进程号的对应关系
-p 指定kill 命令只打印相关进程的进程号，而不发送任何信号
-s 指定发送信号
-u 指定用户
```

实例：

（1）先使用ps查找进程pro1，然后用kill杀掉

1 kill -9 $(ps -ef |grep pro1)

ps 命令

ps(process status)，用来查看当前运行的进程状态，一次性查看，如果需要动态连续结果使用 top linux上进程有5种状态:

1. 运行(正在运行或在运行队列中等待)
2. 中断(休眠中, 受阻, 在等待某个条件的形成或接受到信号)
3. 不可中断(收到信号不唤醒和不可运行, 进程必须等待直到有中断发生)
4. 僵死(进程已终止, 但进程描述符存在, 直到父进程调用wait4()系统调用后释放)
5. 停止(进程收到SIGSTOP, SIGSTP, SIGTIN, SIGTOU信号后停止运行运

行)

ps 工具标识进程的5种状态码:

```
D 不可中断 uninterruptible sleep (usually IO)
R 运行 runnable (on run queue)
S 中断 sleeping
T 停止 traced or stopped
Z 僵死 a defunct (”zombie”) process
```

命令参数：

```
-A 显示所有进程
a 显示所有进程
-a 显示同一终端下所有进程
c 显示进程真实名称
e 显示环境变量
f 显示进程间的关系
r 显示当前终端运行的进程
-aux 显示所有包含其它使用的进程
```

实例：

（1） 显示当前所有进程环境变量及进程间关系

```
ps -ef
```

（2） 显示当前所有进程

```
ps -A
```

（3） 与grep联用查找某进程

```
ps -aux |grep apache
```

（4） 找出与 cron 与 syslog 这两个服务有关的 PID 号码

```
ps aux |grep'(cron|syslog)'
```

rpm 命令

Linux rpm 命令用于管理套件。

rpm(redhat package manager) 原本是 Red Hat Linux 发行版专门用来管理

Linux 各项套件的程序，由于它遵循 GPL 规则且功能强大方便，因而广受欢迎。逐渐受到其他发行版的采用。RPM 套件管理方式的出现，让 Linux 易于安装，升级，间接提升了 Linux 的适用度。

```
## 查看系统自带jdk
rpm -qa | grep jdk
## 删除系统自带jdk
rpm -e --nodeps 查看jdk显示的数据
## 安装jdk
rpm -ivh jdk-7u80-linux-x64.rpm
```

top 命令

显示当前系统正在执行的进程的相关信息，包括进程 ID、内存占用率、CPU 占用率等常用参数：

```
-c 显示完整的进程命令
-s 保密模式
-p <进程号> 指定进程显示
-n <次数>循环显示次数
```

实例：

```
top - 14:06:23 up 70 days, 16:44, 2 users, load average: 1.25, 1.32, 1.35
Tasks: 206 total, 1 running, 205 sleeping, 0 stopped, 0 zombie
Cpu(s): 5.9%us, 3.4%sy, 0.0%ni, 90.4%id, 0.0%wa, 0.0%hi, 0.2%si, 0.0%st
Mem: 32949016k total, 14411180k used, 18537836k free, 169884k buffers
Swap: 32764556k total, 0k used, 32764556k free, 3612636k cached
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
28894 root 22 0 1501m 405m 10m S 52.2 1.3 2534:16 java
```

前五行是当前系统情况整体的统计信息区。

第一行，任务队列信息，同 uptime 命令的执行结果，具体参数说明情况如下：

14:06:23 — 当前系统时间

up 70 days, 16:44 — 系统已经运行了70天16小时44分钟（在这期间系统没有重启过的吆！）

2 users — 当前有2个用户登录系统

load average: 1.15, 1.42, 1.44 — load average后面的三个数分别是1分钟、5 分钟、15分钟的负载情况。

load average数据是每隔5秒钟检查一次活跃的进程数，然后按特定算法计算出的数值。如果这个数除以逻辑CPU的数量，结果高于5的时候就表明系统在超负荷运转了。

第二行，Tasks — 任务（进程），具体信息说明如下：系统现在共有206个进程，其中处于运行中的有1个，205个在休眠（sleep）， stoped状态的有0个，zombie状态（僵尸）的有0个。

第三行，cpu状态信息，具体属性说明如下：

```
5.9%us — 用户空间占用CPU的百分比。
3.4% sy — 内核空间占用CPU的百分比。
0.0% ni — 改变过优先级的进程占用CPU的百分比
90.4% id — 空闲CPU百分比
0.0% wa — IO等待占用CPU的百分比
0.0% hi — 硬中断（Hardware IRQ）占用CPU的百分比
0.2% si — 软中断（Software Interrupts）占用CPU的百分比
```

备注：在这里CPU的使用比率和windows概念不同，需要理解linux系统用户空间和内核空间的相关知识！

第四行，内存状态，具体信息如下：

```
32949016k total — 物理内存总量（32GB）
14411180k used — 使用中的内存总量（14GB）
18537836k free — 空闲内存总量（18GB）
169884k buffers — 缓存的内存量 （169M）
```

第五行，swap交换分区信息，具体信息说明如下：

```
1  32764556k total — 交换区总量（32GB）
2  0k used — 使用的交换区总量（0K）
3  32764556k free — 空闲交换区总量（32GB）
4  3612636k cached — 缓冲的交换区总量（3.6GB）
```

第六行，空行。

第七行以下：各进程（任务）的状态监控，项目列信息说明如下：

```
1   PID — 进程id
2   USER — 进程所有者
3   PR — 进程优先级
4   NI — nice值。负值表示高优先级，正值表示低优先级
5   VIRT — 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES
6   RES — 进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA
7   SHR — 共享内存大小，单位kb
8   S — 进程状态。D=不可中断的睡眠状态 R=运行 S=睡眠 T=跟踪/停止 Z=僵尸进程
9   %CPU — 上次更新到现在的CPU时间占用百分比
10  %MEM — 进程使用的物理内存百分比
11  TIME+ — 进程使用的CPU时间总计，单位1/100秒
12  COMMAND — 进程名称（命令名/命令行）
```

top 交互命令

```
1    h 显示top交互命令帮助信息
2   c 切换显示命令名称和完整命令行
3   m 以内存使用率排序
4    P 根据CPU使用百分比大小进行排序
5   T 根据时间/累计时间进行排序
6  W 将当前设置写入~/.toprc文件中
7   o或者O 改变显示项目的顺序
```

yum 命令

yum（ Yellow dog Updater, Modified）是一个在Fedora和RedHat以及

SUSE中的Shell前端软件包管理器。

基於RPM包管理，能够从指定的服务器自动下载RPM包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软体包，无须繁琐地一次次下载、安装。

yum提供了查找、安装、删除某一个、一组甚至全部软件包的命令，而且命令简洁而又好记。

1.列出所有可更新的软件清单命令：yum check-update

2.更新所有软件命令：yum update

3.仅安装指定的软件命令：yum install

4.仅更新指定的软件命令：yum update

5.列出所有可安装的软件清单命令：yum list

6.删除软件包命令：

- yum remove 7.查找软件包 命令：
- yum search 8.清除缓存命令:
- yum clean packages: 清除缓存目录下的软件包 yum clean headers: 清除缓存目录下的 headers
- yum clean oldheaders: 清除缓存目录下旧的 headers
- yum clean, yum clean all (= yum clean packages; yum clean oldheaders) :清除缓存目录下的软件包及旧的headers

实例

安装 pam-devel

1 [root@www ~]## yum install pam-devel

备份压缩命令 bzip2 命令

创建 *.bz2 压缩文件：bzip2 test.txt 。解压 *.bz2 文件：bzip2 -d test.txt.bz2 。

gzip 命令

创建一个 *.gz 的压缩文件：gzip test.txt 。解压 *.gz 文件：gzip -d test.txt.gz 。

显示压缩的比率：gzip -l *.gz 。

tar 命令

用来压缩和解压文件。tar 本身不具有压缩功能，只具有打包功能，有关压缩及解压是调用其它的功能来完成。

弄清两个概念：打包和压缩。打包是指将一大堆文件或目录变成一个总的文件；压缩则是将一个大的文件通过一些压缩算法变成一个小文件常用参数：

```
-c 建立新的压缩文件
-f 指定压缩文件
-r 添加文件到已经压缩文件包中
-u 添加改了和现有的文件到压缩包中
-x 从压缩包中抽取文件
-t 显示压缩文件中的内容
-z 支持gzip压缩
-j 支持bzip2压缩
-Z 支持compress解压文件
-v 显示操作过程
```

有关 gzip 及 bzip2 压缩:

```
gzip 实例：压缩 gzip fileName .tar.gz 和.tgz 解压：gunzip filename.gz 或 gz ip -d filename.gz
对应：tar zcvf filename.tar.gz tar zxvf filename.tar.gz

bz2实例：压缩 bzip2 -z filename .tar.bz2 解压：bunzip filename.bz2或bzip -d filename.bz2
对应：tar jcvf filename.tar.gz 解压：tar jxvf filename.tar.bz2
```

实例：

（1） 将文件全部打包成 tar 包

```
tar -cvf log.tar 1.log,2.log 或tar -cvf log.*
```

（2） 将 /etc 下的所有文件及目录打包到指定目录，并使用 gz 压缩

```
tar -zcvf /tmp/etc.tar.gz /etc
```

（3） 查看刚打包的文件内容（一定加z，因为是使用 gzip 压缩的）

```
tar -ztvf /tmp/etc.tar.gz
```

（4） 要压缩打包 /home, /etc ，但不要 /home/dmtsai

```
tar --exclude /home/dmtsai -zcvf myfile.tar.gz /home/* /etc
```

unzip 命令

解压 *.zip 文件：unzip test.zip 。查看 *.zip 文件的内容：unzip -l jasper.zip 。

swap分区

Swap交换分区概念

什么是Linux swap space呢？我们先来看看下面两段关于Linux swap space的英文介绍资料:

Linux divides its physical RAM (random access memory) into chucks of memory called pages. Swapping is the process whereby a page of memory is copied to the preconfigured space on the hard disk, called swap space, to free up that page of memory. The combined sizes of the physical memory and the swap space is the amount of virtual memory available.

Swap space in Linux is used when the amount of physical memory (RAM) is full. If the system needs more memory resources and the RAM is full, inactive pages in memory are moved to the swap space. While swap space can help machines with a small amount of RAM, it should not be considered a replacement for more RAM. Swap space is located on hard drives, which have a slower access time than physical memory.Swap space can be a dedicated swap partition (recommended), a swap file, or a combination of swap partitions and swap files.

Linux内核为了提高读写效率与速度，会将文件在内存中进行缓存，这部分内存就是Cache Memory(缓存内存)。即使你的程序运行结束后，Cache Memory也不会自动释放。这就会导致你在Linux系统中程序频繁读写文件后，你会发现可用物理内存变少。当系统的物理内存不够用的时候，就需要将物理内存中的一部分空间释放出来，以供当前运行的程序使用。那些被释放的空间可能来自一些很长时间没有什么操作的程序，这些被释放的空间被临时保存到Swap空间中，等到那些程序要运行时，再从Swap分区中恢复保存的数据到内存中。这样，系统总是在物理内存不够时，才进行Swap交换。

关于Swap分区，其实我们有很多疑问,如果能弄清楚这些疑问，那么你对Swap的了解掌握就差不多了。如何查看Swap分区大小？ Swap分区大小应该如何设置？系统在什么时候会使用Swap分区? 是否可以调整？ 如何调整Swap分区的大小？Swap分区有什么优劣和要注意的地方？ Swap分区是否必要？那么我一个一个来看看这些疑问吧!

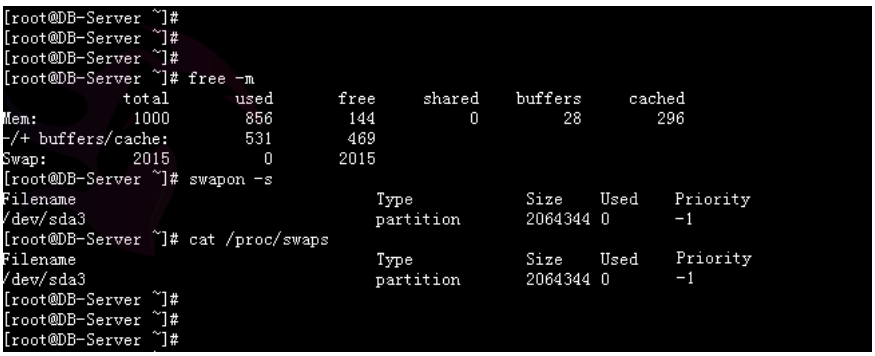
查看Swap分区大小

查看Swap分区的大小以及使用情况，一般使用free命令即可，如下所示，Swap大小为2015M，目前没有使用Swap分区

```
[root@DB-Server ~]# free -m
             total       used       free     shared    buffers     cached
Mem:          1000        855        145          0         28        296
-/+ buffers/cache:        530        470
Swap:         2015          0       2015
```

另外我们还可以使用swapon命令查看当前swap相关信息：例如swap空间是swap partition，Swap size，使用情况等详细信息

```
[root@DB-Server ~]# swapon -s
Filename                                Type            Size    Used    Priority
/dev/sda3                               partition       2064344 0       -1
[root@DB-Server ~]# cat /proc/swaps
Filename                                Type            Size    Used    Priority
/dev/sda3                               partition       2064344 0       -1
[root@DB-Server ~]#
```



Swap分区大小设置

系统的Swap分区大小设置多大才是最优呢？ 关于这个问题，应该说只能有一个统一的参考标准，具体还应该根据系统实际情况和内存的负荷综合考虑，像ORACLE的官方文档就推荐如下设置，这个是根据物理内存来做参考的。

RAM	Swap Space
Up to 512 MB	2 times the size of RAM
Between 1024 MB and 2048 MB	1.5 times the size of RAM
Between 2049 MB and 8192 MB	Equal to the size of RAM
More than 8192 MB	0.75 times the size of RAM

另外在其它博客中看到下面一个推荐设置，当然我不清楚其怎么得到这个标准的。是否合理也无从考证。可以作为一个参考。

4G以内的物理内存，SWAP 设置为内存的2倍。

4-8G的物理内存，SWAP 等于内存大小。

8-64G 的物理内存，SWAP 设置为8G。

64-256G物理内存，SWAP 设置为16G。

上下两个标准确实也很让人无所适从。我就有一次在一台ORACLE数据库服务器（64G的RAM），按照官方推荐设置了一个很大的Swap分区，但是我发现其实这个Swap几乎很少用到，其实是浪费了磁盘空间。所以如果根据系统实际情况和内存的负荷综合考虑，其实应该按照第二个参考标准设置为8G即可。当然这个只是个人的一些认知。

释放Swap分区空间

```
[root@testlnx ~]# free -m
             total       used       free     shared    buffers     cached
Mem:         64556      55368       9188          0        926      51405
-/+ buffers/cache:       3036      61520
Swap:        65535         13      65522
[root@testlnx ~]# swapon -s
Filename                                Type            Size    Used    Priority
/dev/mapper/VolGroup00-LogVol01         partition       67108856        14204   -1
```

使用swapoff关闭交换分区

[root@testlnx ~]# swapoff /dev/mapper/VolGroup00-LogVol01

使用swapon启用交换分区，此时查看交换分区的使用情况，你会发现used为0了

```
[root@testlnx ~]# swapon /dev/mapper/VolGroup00-LogVol01
[root@testlnx ~]# free -m
             total       used       free     shared    buffers     cached
Mem:         64556      55385       9171          0        926      51406
-/+ buffers/cache:       3052      61504
Swap:        65535          0      65535
[root@testlnx ~]#
```

Swap分区空间什么时候使用

系统在什么情况或条件下才会使用Swap分区的空间呢？ 其实是Linux通过一个参数swappiness来控制的。当然还涉及到复杂的算法。

这个参数值可为 0-100，控制系统 swap 的使用程度。高数值可优先系统性能，在进程不活跃时主动将其转换出物理内存。低数值可优先互动性并尽量避免将进程转换处物理内存，并降低反应延迟。默认值为 60。注意：这个只是一个权值，不是一个百分比值，涉及到系统内核复杂的算法。关于该参数请参考这篇文章[转载]调整虚拟内存，在此不做过多赘述。下面是关于swappiness的相关资料

The Linux 2.6 kernel added a new kernel parameter called swappiness to let administrators tweak the way Linux swaps. It is a number from 0 to 100. In essence, higher values lead to more pages being swapped, and lower values lead to more applications being kept in memory, even if they are idle. Kernel maintainer Andrew Morton has said that he runs his desktop machines with a swappiness of 100, stating that "My point is that decreasing the tendency of the kernel to swap stuff out is wrong. You really don't want hundreds of megabytes of BloatyApp's untouched memory floating about in the machine. Get it out on the disk, use the memory for something useful."

Swappiness is a property of the Linux kernel that changes the balance between swapping out runtime memory, as opposed to dropping pages from the system page cache. Swappiness can be set to values between 0 and 100 inclusive. A low value means the kernel will try to avoid swapping as much as possible where a higher value instead will make the kernel aggressively try to use swap space. The default value is 60, and for most desktop systems, setting it to 100 may affect the overall performance, whereas setting it lower (even 0) may improve interactivity (by decreasing response latency.

有两种临时修改swappiness参数的方法，系统重启后失效

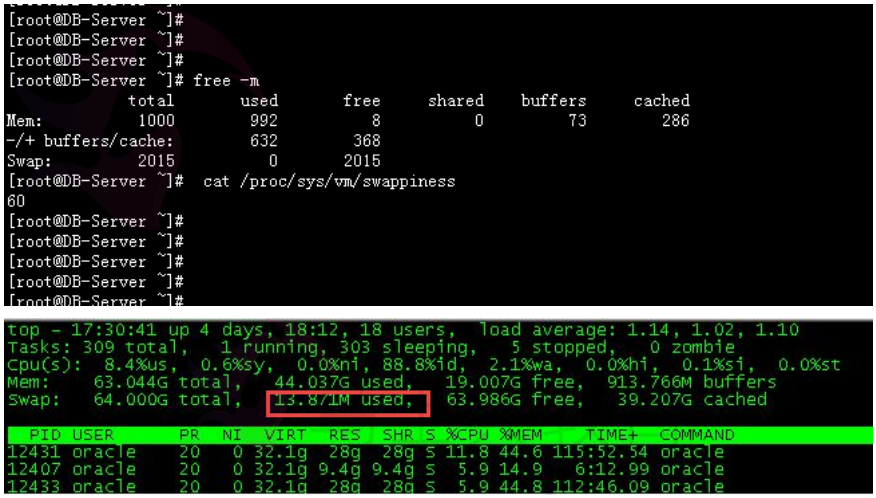
```
方法1：
[root@DB-Server ~]# more /proc/sys/vm/swappiness
60
[root@DB-Server ~]# echo 10 > /proc/sys/vm/swappiness
[root@DB-Server ~]# more /proc/sys/vm/swappiness
10

方法2
[root@DB-Server ~]#sysctl vm.swappiness=10
```

永久修改swappiness参数的方法就是在配置文件/etc/sysctl.conf里面修改vm.swappiness的值，然后重启系统

```
echo 'vm.swappiness=10' >>/etc/sysctl.conf
```

如果有人会问是否物理内存使用到某个百分比后才会使用Swap交换空间，可以明确的告诉你不是这样一个算法，如下截图所示，及时物理内存只剩下8M了，但是依然没有使用Swap交换空间，而另外一个例子，物理内存还剩下19G，居然用了一点点Swap交换空间。



另外调整/proc/sys/vm/swappiness这个参数，如果你没有绝对把握，就不要随便调整这个内核参数，这个参数符合大多数情况下的一个最优值。

Swap交换分区对性能的影响

我们知道Linux可以使用文件系统中的一个常规文件或独立分区作为Swap交换空间，相对而言，交换分区要快一些。但是和RAM比较而言，Swap交换分区的性能依然比不上物理内存，目前的服务器上RAM基本上都相当充足，那么是否可以考虑抛弃Swap交换分区，是否不需要保留Swap交换分区呢？这个其实是我的疑问之一。在这篇What Is a Linux SWAP Partition, And What Does It Do?博客中，作者给出了swap交换空间的优劣

Advantages:

1. Provides overflow space when your memory fills up completely
2. Can move rarely-needed items away from your high-speed memory
3. Allows you to hibernate

Disadvantages:

1. Takes up space on your hard drive as SWAP partitions do not resize dynamically

2. Can increase wear and tear to your hard drive
3. Does not necessarily improve performance (see below)

其实保留swap分区概括起来可以从下面来看：

首先，当物理内存不足以支撑系统和应用程序（进程）的运作时，这个Swap交换分区可以用作临时存放使用率不高的内存分页，把腾出的内存交给急需的应用程序（进程）使用。有点类似机房的UPS系统，虽然正常情况下不需要使用，但是异常情况下， Swap交换分区还是会发挥其关键作用。

其次，即使你的服务器拥有足够多的物理内存，也有一些程序会在它们初始化时残留的极少再用到的内存分页内容转移到 swap 空间，以此让出物理内存空间。对于有发生内存泄漏几率的应用程序（进程），Swap交换分区更是重要，因为谁也不想看到由于物理内存不足导致系统崩溃。

最后，现在很多个人用户在使用Linux，有些甚至是PC的虚拟机上跑Linux系统，此时可能常用到休眠（Hibernate），这种情况下也是推荐划分Swap交换分区的。

其实少量使用Swap交换空间是不会影响性能，只有当RAM资源出现瓶颈或者内存泄露，进程异常时导致频繁、大量使用交换分区才会导致严重性能问题。另外使用Swap交换分区频繁，还会引起kswapd0进程（虚拟内存管理中, 负责换页的）耗用大量CPU资源，导致CPU飙升。

关于Swap分区的优劣以及是否应该舍弃，我有点恶趣味的想到了这个事情：人身上的两个器官，阑尾和扁桃体。切除阑尾或扁桃体是否也是争论不休。另外，其实不要Swap交换分区，Linux也是可以正常运行的（有人提及过这个问题）

调整Swap分区的大小

如下测试案例所示，Swap分区大小为65535M，我现在想将Swap分区调整为8G，那么我们来看看具体操作吧

1：查看Swap的使用情况以及相关信息

```
[root@getlnx14uat ~]# swapon -s
Filename                                Type            Size    Used    Priority
/dev/mapper/VolGroup00-LogVol01         partition       67108856        878880  -1
[root@getlnx14uat ~]# free -m
             total       used       free     shared    buffers     cached
Mem:          3957       3920         36          0         39       3055
-/+ buffers/cache:        825       3132
Swap:        65535        858      64677
```

2： 关闭Swap交换分区

```
[root@getlnx14uat ~]# swapoff /dev/mapper/VolGroup00-LogVol01
[root@getlnx14uat ~]# swapon -s
Filename                                Type            Size    Used    Priority
```

3： 这里是缩小Swap分区大小，如果是增大Swap分区大小，那么就需要扩展正在使用的swap分区的逻辑卷,此处使用lvreduce命令收缩逻辑卷。

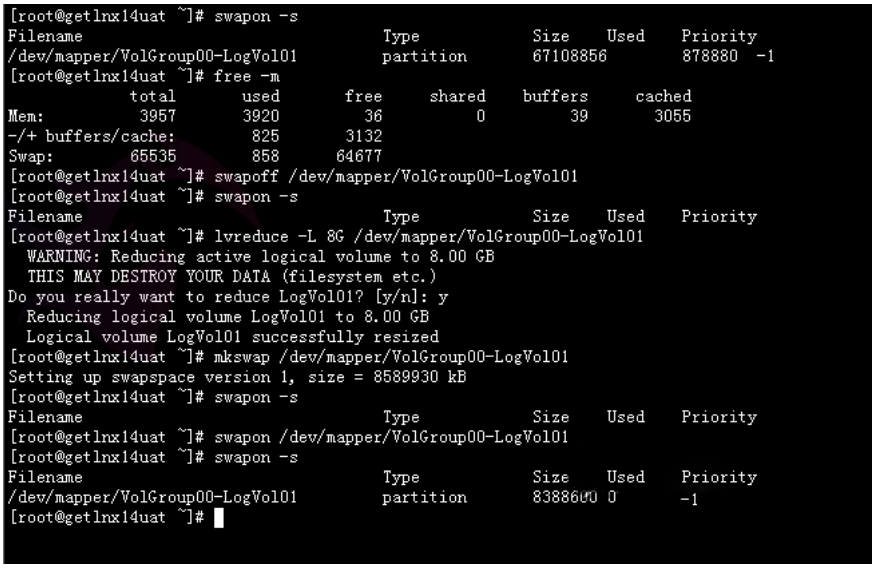
```
[root@getlnx14uat ~]# lvreduce -L 8G /dev/mapper/VolGroup00-LogVol01
  WARNING: Reducing active logical volume to 8.00 GB
  THIS MAY DESTROY YOUR DATA (filesystem etc.)
Do you really want to reduce LogVol01? [y/n]: y
  Reducing logical volume LogVol01 to 8.00 GB
  Logical volume LogVol01 successfully resized
```

4：格式化swap分区

```
[root@getlnx14uat ~]# mkswap /dev/mapper/VolGroup00-LogVol01
Setting up swapspace version 1, size = 8589930 kB
```

5:启动swap分区,并增加到/etc/fstab自动挂载

```
[root@getlnx14uat ~]# swapon -s
Filename                                Type            Size    Used    Priority
[root@getlnx14uat ~]# swapon /dev/mapper/VolGroup00-LogVol01
[root@getlnx14uat ~]# swapon -s
Filename                                Type            Size    Used    Priority
/dev/mapper/VolGroup00-LogVol01         partition       8388600 0       -1
```



Linux系统swappiness参数在内存与交换分区之间优化作用

swappiness的值的大小对如何使用swap分区是有着很大的联系的。swappiness=0的时候表示最大限度使用物理内存，然后才是 swap空间，swappiness = 100的时候表示积极的使用swap分区，并且把内存上的数据及时的搬运到swap空间里面。linux的基本默认设置为60，具体如下：

一般默认值都是60

```
[root@timeserver ~]# cat /proc/sys/vm/swappiness
60
```

也就是说，你的内存在使用到100-60=40%的时候，就开始出现有交换分区的使用。大家知道，内存的速度会比磁盘快很多，这样子会加大系统io，同时造的成大量页的换进换出，严重影响系统的性能，所以我们在操作系统层面，要尽可能使用内存，对该参数进行调整。

临时调整的方法如下，我们调成10：
[root@timeserver ~]# sysctl vm.swappiness=10
vm.swappiness = 10
[root@timeserver ~]# cat /proc/sys/vm/swappiness
10
这只是临时调整的方法，重启后会回到默认设置的

要想永久调整的话，需要将
需要在/etc/sysctl.conf修改，加上：
[root@timeserver ~]# cat /etc/sysctl.conf

# Controls the maximum number of shared memory segments, in pages
kernel.shmall = 4294967296
vm.swappiness=10

激活设置

[root@timeserver ~]# sysctl -p

在linux中，可以通过修改swappiness内核参数，降低系统对swap的使用，从而提高系统的性能。

遇到的问题是这样的，新版本产品发布后，每小时对内存的使用会有一个尖峰。虽然这个峰值还远没有到达服务器的物理内存，但确发现内存使用达到峰值时系统开始使用swap。在swap的过程中系统性能会有所下降，表现为较大的服务延迟。对这种情况，可以通过调节swappiness内核参数降低系统对swap的使用，从而避免不必要的swap对性能造成的影响。

简单地说这个参数定义了系统对swap的使用倾向，默认值为60，值越大表示越倾向于使用swap。可以设为0，这样做并不会禁止对swap的使用，只是最大限度地降低了使用swap的可能性。

通过sysctl -q vm.swappiness可以查看参数的当前设置。

修改参数的方法是修改/etc/sysctl.conf文件，加入vm.swappiness=xxx，并重起系统。这个操作相当于是修改虚拟系统中的/proc/sys/vm/swappiness文件，将值改为XXX数值。

如果不想重起，可以通过sysctl -p动态加载/etc/sysctl.conf文件，但建议这样做之前先清空swap。

overcommit_memory

取值为0，系统在为应用进程分配虚拟地址空间时，会判断当前申请的虚拟地址空间大小是否超过剩余内存大小，如果超过，则虚拟地址空间分配失败。因此，也就是如果进程本身占用的虚拟地址空间比较大或者剩余内存比较小时，fork、malloc等调用可能会失败。

取值为1，系统在为应用进程分配虚拟地址空间时，完全不进行限制，这种情况下，避免了fork可能产生的失败，但由于malloc是先分配虚拟地址空间，而后通过异常陷入内核分配真正的物理内存，在内存不足的情况下，这相当于完全屏蔽了应用进程对系统内存状态的感知，即malloc总是能成功，一旦内存不足，会引起系统OOM杀进程，应用程序对于这种后果是无法预测的

取值为2，则是根据系统内存状态确定了虚拟地址空间的上限，由于很多情况下，进程的虚拟地址空间占用远大小其实际占用的物理内存，这样一旦内存使用量上去以后，对于一些动态产生的进程(需要复制父进程地址空间)则很容易创建失败，如果业务过程没有过多的这种动态申请内存或者创建子进程，则影响不大，否则会产生比较大的影响

又一次内存分配失败(关于overcommit_memory)

1、问题现象和分析：
测试时发现当系统中空闲内存还有很多时，就报内存分配失败了，所有进程都报内存分配失败：
sshd@localhost:/var/log>free
total used free shared buffers cached
Mem: 12183700 8627972 3555728 0 289252 584444
-/+ buffers/cache: 7754276 4429424
Swap: 0 0 0
sshd@localhost:/var/log>free
-bash: fork: Cannot allocate memory
sshd@localhost:/var/log>cat /proc/meminfo
-bash: fork: Cannot allocate memory

而messages日志中，也没有OOM相关的记录。最后确认原因为：/proc/sys/vm/overcommit_memory参数导致。
该环境中该参数设置为2，表示“No overcommit”，即系统中所有进程占用的虚拟内存空间不能超过上限：
cat /proc/meminfo
CommitLimit: 12061860 kB //虚拟地址空间的上限
Committed_AS: 8625360 kB //当前的使用量

而该参数应该默认是0，这种情况下，只有还有空闲的物理内存，就可以继续分配，不受虚拟地址空间的限制。
echo 0 > /proc/sys/vm/overcommit_memory
如此修正后解决。

2、关于overcommit_memory说明：

取值为0，系统在为应用进程分配虚拟地址空间时，会判断当前申请的虚拟地址空间大小是否超过剩余内存大小，如果超过，则虚拟地址空间分配失败。因此，也就是如果进程本身占用的虚拟地址空间比较大或者剩余内存比较小时，fork、malloc等调用可能会失败。

取值为1，系统在为应用进程分配虚拟地址空间时，完全不进行限制，这种情况下，避免了fork可能产生的失败，但由于malloc是先分配虚拟地址空间，而后通过异常陷入内核分配真正的物理内存，在内存不足的情况下，这相当于完全屏蔽了应用进程对系统内存状态的感知，即malloc总是能成功，一旦内存不足，会引起系统OOM杀进程，应用程序对于这种后果是无法预测的

取值为2，则是根据系统内存状态确定了虚拟地址空间的上限，由于很多情况下，进程的虚拟地址空间占用远大小其实际占用的物理内存，这样一旦内存使用量上去以后，对于一些动态产生的进程(需要复制父进程地址空间)则很容易创建失败，如果业务过程没有过多的这种动态申请内存或者创建子进程，则影响不大，否则会产生比较大的影响

3、相应代码分析：

```
1. int __vm_enough_memory(struct mm_struct *mm, long pages, int cap_sys_admin)
2. {
3.   unsigned long free, allowed;
4.
5.   vm_acct_memory(pages);
6.
7.   /*
8.    \* Sometimes we want to use more memory than we have
9.    */
10.   if (sysctl_overcommit_memory == OVERCOMMIT_ALWAYS) //overcommit_memory=1，直接返回成功，不做任何限制。
11.     return 0;
12.
13.   if (sysctl_overcommit_memory == OVERCOMMIT_GUESS) { //overcommit_memory=0，启发式方式，根据当前系统中空闲内存状况来决定是否可以分配内存。
14.     unsigned long n;
```

```
15.
16.     free = global_page_state(NR_FILE_PAGES);
17.     free += nr_swap_pages;
18.
19.     /*
20.      \* Any slabs which are created with the
21.      \* SLAB_RECLAIM_ACCOUNT flag claim to have contents
22.      \* which are reclaimable, under pressure. The dentry
23.      \* cache and most inode caches should fall into this
24.      */
25.     free += global_page_state(NR_SLAB_RECLAIMABLE);
26.
27.     /*
28.      \* Leave the last 3% for root
29.      */
30.     if (!cap_sys_admin)
31.       free -= free / 32; //root用户可以在free更少(3%)的时候，分配内存。
32.
33.     if (free > pages) // pages为需要分配的内存大小，free为根据一定规则算出来的“空闲内存大小”，第一次free仅为NR_FILE_PAGES+NR_SLAB_RECLAIMABLE，由于直接或者系统中“实际空闲”内存代价比较大，所以进行分阶判断，提高效率。
34.       return 0;
35.
36.     /*
37.      \* nr_free_pages() is very expensive on large systems,
38.      \* only call if we're about to fail.
39.      */
40.     n = nr_free_pages(); //当第一次判断不满足内存分配条件时，再进行“实际空闲”内存的获取操作。
41.
42.     /*
43.      \* Leave reserved pages. The pages are not for anonymous pages.
44.      */
45.     if (n <= totalreserve_pages)
46.       goto error;
47.     else
48.       n -= totalreserve_pages;
49.
50.     /*
51.      \* Leave the last 3% for root
52.      */
53.     if (!cap_sys_admin)
54.       n -= n / 32;
55.     free += n;
56.
57.     if (free > pages)
58.       return 0;
59.
60.     goto error;
61.   }
62.
63.   allowed = (totalram_pages - hugetlb_total_pages()) //当overcommit_memory=2时，根据系统中虚拟地址空间的总量来进行限制。
64.      \* sysctl_overcommit_ratio / 100;
65.   /*
66.    \* Leave the last 3% for root
67.    */
68.   if (!cap_sys_admin)
69.     allowed -= allowed / 32;
70.   allowed += total_swap_pages;
71.
72.   /* Don't let a single process grow too big:
73.    leave 3% of the size of this process for other processes */
74.   if (mm)
75.     allowed -= mm->total_vm / 32;
76.
77.   if (percpu_counter_read_positive(&vm_committed_as) < allowed)
78.     return 0;
79. error:
80.   vm_unacct_memory(pages);
81.
82.   return -ENOMEM;
83. }
```

linux系统下查看CPU、内存负载情况

$ vmstat
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r b   swpd   free   buff cache   si   so   bi   bo   in   cs us sy id wa st
 1 4 329796 26040   4528 3379824   1   1   50   160   36   17 2 10 85 3 0

结果解释如下：

procs
r 列表示运行和等待cpu时间片的进程数，如果长期大于1，说明cpu不足，需要增加cpu。
b 列表示在等待资源的进程数，比如正在等待I/O、或者内存交换等。
cpu 表示cpu的使用状态
us 列显示了用户方式下所花费 CPU 时间的百分比。us的值比较高时，说明用户进程消耗的cpu时间多，但是如果长期大于50%，需要考虑优化用户的程序。
sy 列显示了内核进程所花费的cpu时间的百分比。这里us + sy的参考值为80%，如果us+sy 大于 80%说明可能存在CPU不足。
wa 列显示了IO等待所占用的CPU时间的百分比。这里wa的参考值为30%，如果wa超过30%，说明IO等待严重，这可能是磁盘大量随机访问造成的，也可能磁盘或者磁盘访问控制器的带宽瓶颈造成的(主要是块操作)。
id 列显示了cpu处在空闲状态的时间百分比
system 显示采集间隔内发生的中断数
in 列表示在某一时间间隔中观测到的每秒设备中断数。
cs列表示每秒产生的上下文切换次数，如当 cs 比磁盘 I/O 和网络信息包速率高得多，都应进行进一步调查。
memory
swpd 切换到内存交换区的内存数量(k表示)。如果swpd的值不为0，或者比较大，比如超过了100m，只要si、so的值长期为0，系统性能还是正常
free 当前的空闲页面列表中内存数量(k表示)
buff 作为buffer cache的内存数量，一般对块设备的读写才需要缓冲。
cache: 作为page cache的内存数量，一般作为文件系统的cache，如果cache较大，说明用到cache的文件较多，如果此时IO中bi比较小，说明文件系统效率比较好。
swap
si 由内存进入内存交换区数量。
so由内存交换区进入内存数量。
IO
bi 从块设备读入数据的总量（读磁盘）（每秒kb）。
bo 块设备写入数据的总量（写磁盘）（每秒kb）

经典面试题

问题一：绝对路径用什么符号表示？当前目录、上层目录用什么表示？主目录用什么表示？切换目录用什么命令？

答案：

绝对路径：如/etc/init.d

当前目录和上层目录：./ ../ 主目录：~/

切换目录：cd

问题二：怎么查看当前进程？怎么执行退出？怎么查看当前路径？答案：

查看当前进程：ps 执行退出：exit

查看当前路径：pwd

问题三：怎么清屏？怎么退出当前命令？怎么执行睡眠？怎么查看当前用户 id？查看指定帮助用什么命令？

答案：

清屏：clear

退出当前命令：ctrl+c 彻底退出执行睡眠： ctrl+z 挂起当前进程 fg 恢复后台查看当前用户 id：”id“：查看显示目前登陆账户的 uid 和 gid 及所属分组及用户名

查看指定帮助：如 man adduser 这个很全 而且有例子；adduser --help 这个告诉你一些常用参数；info adduesr；

问题四：Ls 命令执行什么功能？可以带哪些参数，有什么区别？

答案： ls 执行的功能：列出指定目录中的目录，以及文件哪些参数以及区别：a 所有文件 l 详细信息，包括大小字节数，可读可写可执行的权限等

问题五：建立软链接(快捷方式)，以及硬链接的命令。

答案：

软链接：ln -s slink source 硬链接：ln link source

问题六：目录创建用什么命令？创建文件用什么命令？复制文件用什么命令？答案：

创建目录：mkdir

创建文件：典型的如 touch，vi 也可以创建文件，其实只要向一个不存在的文件输出，都会创建文件复制文件：cp 7. 文件权限修改用什么命令？格式是怎么样的？文件权限修改：chmod

格式如下：

$ chmod u+x file 给 file 的属主增加执行权限

$ chmod 751 file 给 file 的属主分配读、写、执行(7)的权限，给 file 的所在组分配读、执行(5)的权限，给其他用户分配执行(1)的权限

$ chmod u=rwx,g=rx,o=x file 上例的另一种形式

$ chmod =r file 为所有用户分配读权限

$ chmod 444 file 同上例

$ chmod a-wx,a+r file 同上例

$ chmod -R u+r directory 递归地给 directory 目录下所有文件和子目录的属主分配读的权限

问题八：查看文件内容有哪些命令可以使用？答案：

vi 文件名 #编辑方式查看，可修改 cat 文件名 #显示全部文件内容 more 文件名 #分页显示文件内容 less 文件名 #与 more 相似，更好的是可以往前翻页 tail 文件名 #仅查看尾部，还可以指定行数 head 文件名 #仅查看头部,还可以指定行数

问题九：

随意写文件命令？怎么向屏幕输出带空格的字符串，比如”hello world”？

答案：

写文件命令：vi

向屏幕输出带空格的字符串:echo hello world

问题十：终端是哪个文件夹下的哪个文件？黑洞文件是哪个文件夹下的哪个命令？答案：

终端 /dev/tty

黑洞文件 /dev/null

问题十一：移动文件用哪个命令？改名用哪个命令？答案： mv mv

问题十二：

[root@localhost ~]## whatis zcat zcat [gzip] (1) – compress or expand files

问题十三：使用哪一个命令可以查看自己文件系统的磁盘空间配额呢？

答案：使用命令 repquota 能够显示出一个文件系统的配额信息

MQ（一明老师，周老师）

为什么使用MQ？MQ的优点

简答

- 异步处理 - 相比于传统的串行、并行方式，提高了系统吞吐量。
- 应用解耦 - 系统间通过消息通信，不用关心其他系统的处理。
- 流量削锋 - 可以通过消息队列长度控制请求量；可以缓解短时间内的高并发请 求。
- 日志处理 - 解决大量日志传输。
- 消息通讯 - 消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通 讯。比如实现点对点消息队列，或者聊天室等。

详答

主要是：解耦、异步、削峰。

解耦：A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要 这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃...A 系统 跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系 统都需要 A 系统将这个数据发送过来。如果使用 MQ，A 系统产生一条数据， 发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要 数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数 据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。 就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦。

异步：A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写 库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、 200ms。最终请求总延时是 3 + 300 + 450 + 200 = 953ms，接近 1s，用户 感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求。如果使用 MQ，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从 接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms。

削峰：减少高峰时期对服务器压力。

消息队列有什么优缺点？RabbitMQ有什么优缺点？

优点上面已经说了，就是**在特殊场景下有其对应的好处，解耦、异步、削峰。**

缺点有以下几个：

系统可用性降低

本来系统运行好好的，现在你非要加入个消息队列进去，那消息队列挂了，你的 系统不是呵呵了。因此，系统可用性会降低；

系统复杂度提高

加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息 不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，复杂 性增大。

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是， 要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了， 咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对 它带来的坏处做各种额外的技术方案和架构来规避掉，做好之后，你会发现，妈 呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用， 还是得用的。

你们公司生产环境用的是什么消息中间件?

这个首先你可以说下你们公司选用的是什么消息中间件，比如用的是 RabbitMQ，然后可以初步给一些你对不同MQ中间件技术的选型分析。

举个例子：比如说ActiveMQ是老牌的消息中间件，国内很多公司过去运用的还 是非常广泛的，功能很强大。

但是问题在于没法确认ActiveMQ可以支撑互联网公司的高并发、高负载以及高 吞吐的复杂场景，在国内互联网公司落地较少。而且使用较多的是一些传统企 业，用ActiveMQ做异步调用和系统解耦。

然后你可以说说RabbitMQ，他的好处在于可以支撑高并发、高吞吐、性能很 高，同时有非常完善便捷的后台管理界面可以使用。

另外，他还支持集群化、高可用部署架构、消息高可靠支持，功能较为完善。

而且经过调研，国内各大互联网公司落地大规模RabbitMQ集群支撑自身业务的 case较多，国内各种中小型互联网公司使用RabbitMQ的实践也比较多。

除此之外，RabbitMQ的开源社区很活跃，较高频率的迭代版本，来修复发现的 bug以及进行各种优化，因此综合考虑过后，公司采取了RabbitMQ。

但是RabbitMQ也有一点缺陷，就是他自身是基于erlang语言开发的，所以导致 较为难以分析里面的源码，也较难进行深层次的源码定制和改造，毕竟需要较为 扎实的erlang语言功底才可以。

然后可以聊聊RocketMQ，是阿里开源的，经过阿里的生产环境的超高并发、 高吞吐的考验，性能卓越，同时还支持分布式事务等特殊场景。

而且RocketMQ是基于Java语言开发的，适合深入阅读源码，有需要可以站在 源码层面解决线上生产问题，包括源码的二次开发和改造。

另外就是Kafka。Kafka提供的消息中间件的功能明显较少一些，相对上述几款 MQ中间件要少很多。

但是Kafka的优势在于专为超高吞吐量的实时日志采集、实时数据同步、实时数 据计算等场景来设计。

因此Kafka在大数据领域中配合实时计算技术（比如Spark Streaming、 Storm、Flink）使用的较多。但是在传统的MQ中间件使用场景中较少采用。

Kafka、ActiveMQ、RabbitMQ、RocketMQ 有 什么优缺点?

ActiveMQ	RabbitMQ	RocketMQ	Kafka	ZeroMQ	
单机吞吐量	比 RabbitM Q低	2.6w/s（ 消息做持 久化）	11.6w/s	17.3w/s	29w/s
开发语言	Java	Erlang	Java	Scala/Java	C
主要维护者	Apache	Mozilla/Spring	Alibaba	Apache	iMatix创始人已去世
成熟度	成熟	成熟	开源版本不够成熟	比较成熟	只有C、PHP等版本成熟
订阅形式	点对点 (p2p)、广 播（发布订阅）	提供了4 种： direct, topic,Headers 和 fanout。fanout就 是广播模 式	基于 topic/me ssageTag 以及按照消息类 型、属性 进行正则 匹配的发 布订阅模 式	基于topic 以及按照 topic进行正则匹配的发布订 阅模式	点对点 (P2P)
持久化	支持少量 堆积	支持少量 堆积	支持大量 堆积	支持大量 堆积	不支持
顺序消息	不支持	不支持	支持	支持	不支持
性能稳定性	好	好	一般	较差	很好
集群方式	支持简单 集群模 式，比如'主备'，对 高级集群 模式支持 不好。	支持简单 集群，'复 制'模 式，对高 级集群模 式支持不 好。	常用 多 对'Mast erSlave' 模 式，开源版本需手 动切换 Slave变成 Master	天然 的'Lead erSlave'无 状态集群，每台 服务器既 是Master 也是Slave	不支持
管理界面	一般	较好	一般	无	无

综上，各种对比之后，有如下建议：

一般的业务系统要引入 MQ，最早大家都用 ActiveMQ，但是现在确实大家用 的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，所以大家还是 算了吧，我个人不推荐用这个了；

后来大家开始用 RabbitMQ，但是确实 erlang 语言阻止了大量的 Java 工程师 去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开 源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司会去用 RocketMQ，确实很不错，毕竟是阿里出 品，但社区可能有突然黄掉的风险（目前 RocketMQ 已捐给 Apache，但 GitHub 上的活跃度其实不算高）对自己公司技术实力有绝对自信的，推荐用 RocketMQ，否则回去老老实实用 RabbitMQ 吧，人家有活跃的开源社区，绝 对不会黄。

所以**中小型公司**，技术实力较为一般，技术挑战不是特别高，用 RabbitMQ 是 不错的选择；**大型公司**，基础架构研发实力较强，用 RocketMQ 是很好的选 择。

如果是**大数据领域**的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝 对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性 规范。

MQ 有哪些常见问题？如何解决这些问题?

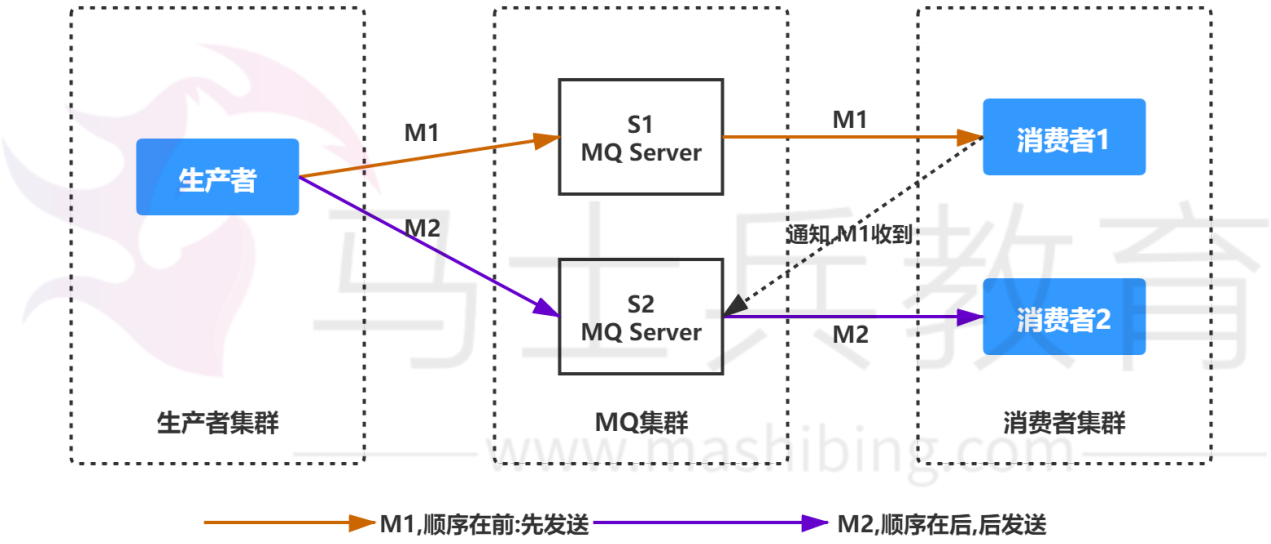
MQ 的常见问题有：

1. 消息的顺序问题
2. 消息的重复问题

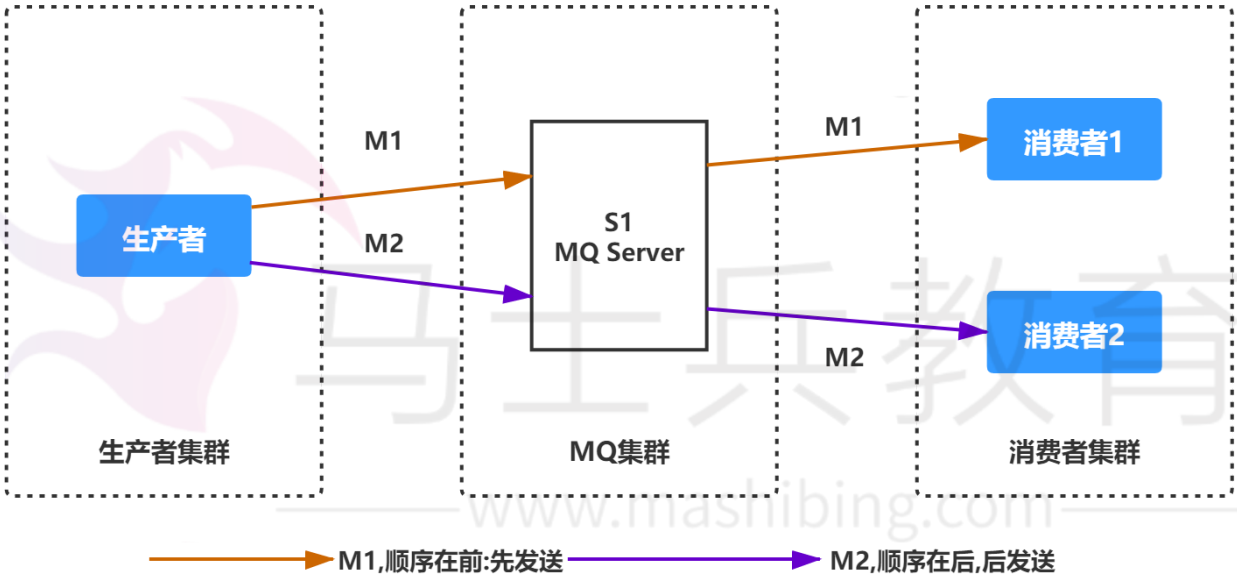
消息的顺序问题

消息有序指的是可以按照消息的发送顺序来消费。

假如生产者产生了 2 条消息：M1、M2，假定 M1 发送到 S1，M2 发送到 S2，如果要保证 M1 先于 M2 被消费，怎么做?



解决方案： （1）保证生产者 - MQServer - 消费者是一对一对一的关系



缺陷：

- 并行度就会成为消息系统的瓶颈（吞吐量不够）
- 更多的异常处理，比如：只要消费端出现问题，就会导致整个处理流程阻塞，我 们不得不花费更多的精力来解决阻塞的问题。（2）通过合理的设计或者将问题分解 来规避。
- 不关注乱序的应用实际大量存在
- 队列无序并不意味着消息无序 所以从业务层面来保证消息的顺序而不仅仅是依 赖于消息系统，是一种更合理的方式。

消息的重复问题

造成消息重复的根本原因是：网络不可达。

所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收 到两条一样的消息，应该怎样处理?

消费端处理消息的业务逻辑保持幂等性。只要保持幂等性，不管来多少条重复消 息，最后处理的结果都一样。保证每条消息都有唯一编号且保证消息处理成功与 去重表的日志同时出现。利用一张日志表来记录已经处理成功的消息的 ID，如 果新到的消息 ID 已经在日志表中，那么就不再处理这条消息。

什么是RabbitMQ?

RabbitMQ是一款开源的，Erlang编写的，基于AMQP协议的消息中间件

Rabbitmq 的使用场景

（1）服务间异步通信

（2）顺序消费

（3）定时任务

（4）请求削峰

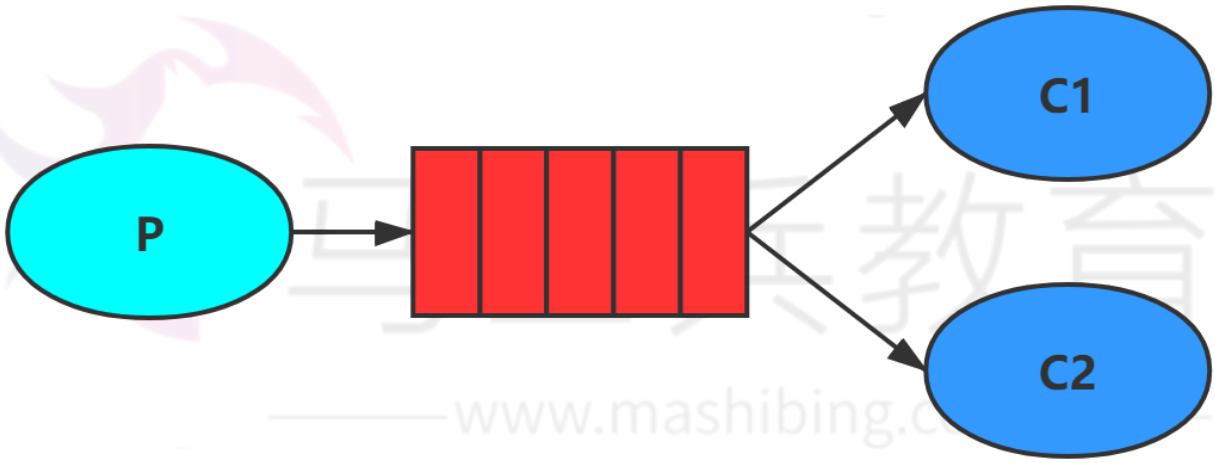
RabbitMQ基本概念

- Broker： 简单来说就是消息队列服务器实体
- Exchange： 消息交换机，它指定消息按什么规则，路由到哪个队列
- Queue： 消息队列载体，每个消息都会被投入到一个或多个队列
- Binding： 绑定，它的作用就是把exchange和queue按照路由规则绑定起来
- Routing Key： 路由关键字，exchange根据这个关键字进行消息投递
- VHost： vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部 均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的 权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度， vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同 的 vhost 中）。
- Producer： 消息生产者，就是投递消息的程序
- Consumer： 消息消费者，就是接受消息的程序
- Channel： 消息通道，在客户端的每个连接里，可建立多个channel，每个 channel代表一个会话任务

由Exchange、Queue、RoutingKey三个才能决定一个从Exchange到Queue的 唯一的线路。

RabbitMQ的工作模式

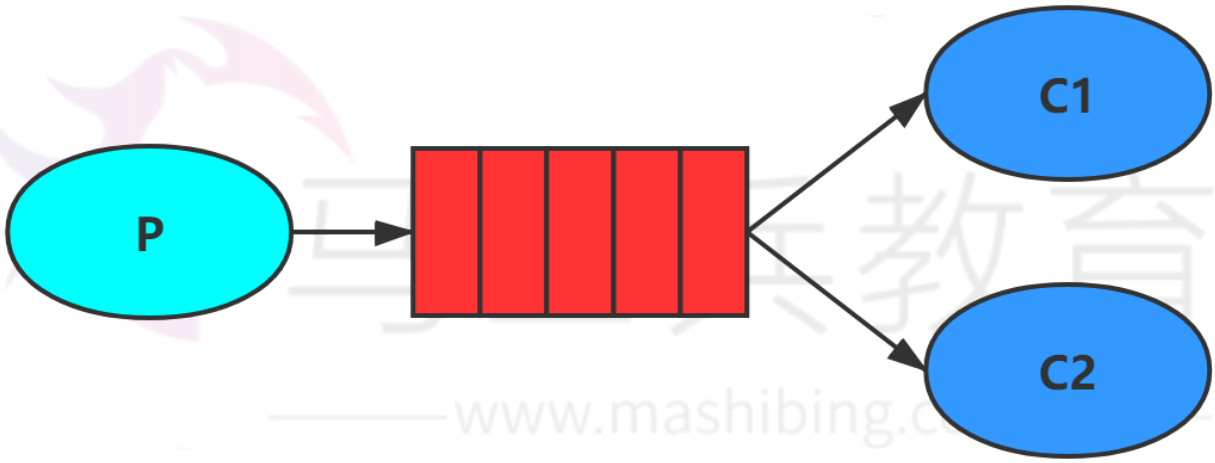
一.simple模式（即最简单的收发模式）



1.消息产生消息，将消息放入队列

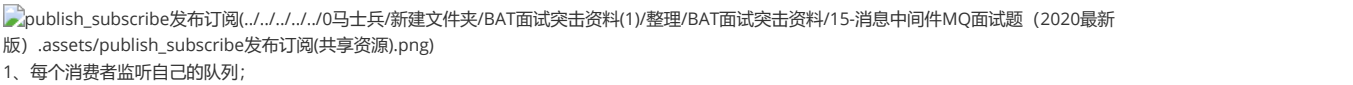
2.消息的消费者(consumer) 监听 消息队列,如果队列中有消息,就消费掉,消息被 拿走后,自动从队列中删除(隐患 消息可能没有被消费者正确处理,已经从队列中消失了,造成消息的丢失，这里可以设置成手动的ack,但如果设置成手动ack，处 理完后要及时发送ack消息给队列，否则会造成内存溢出)。

二.work工作模式(资源的竞争)



1.消息产生者将消息放入队列消费者可以有多个,消费者1,消费者2同时监听同一 个队列,消息被消费。C1 C2共同争抢当前的消息队列内容,谁先拿到谁负责消费 消息(隐患：高并发情况下,默认会产生某一个消息被多个消费者共同使用,可以设 置一个开关(syncronize) 保证一条消息只能被一个消费者使用)。

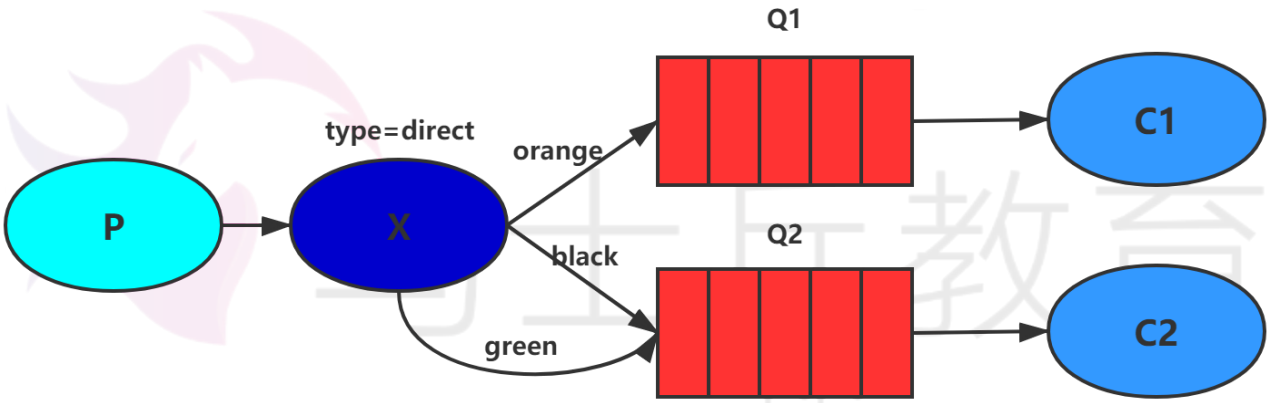
三.publish/subscribe发布订阅(共享资源)


publish_subscribe发布订阅(../../../../../0马士兵/新建文件夹/BAT面试突击资料(1)/整理/BAT面试突击资料/15-消息中间件MQ面试题（2020最新版）.assets/publish_subscribe发布订阅(共享资源).png)

1、每个消费者监听自己的队列；

2、生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队 列，每个绑定交换机的队列都将接收到消息。

四.routing路由模式



1.消息生产者将消息发送给交换机按照路由判断,路由是字符串(info) 当前产生的 消息携带路由字符(对象的方法),交换机根据路由的key,只能匹配上路由key对应的消息队列,对应的消费者才能消费消息;

2.根据业务功能定义路由字符串

3.从系统的代码逻辑中获取对应的功能字符串,将消息任务扔到对应的队列中。

4.业务场景:error 通知;EXCEPTION;错误通知的功能;传统意义的错误通知;客户 通知;利用key路由,可以将程序中的错误封装成消息传入到消息队列中,开发者可 以自定义消费者,实时接收错误;

五.topic 主题模式(路由模式的一种)

1.星号井号代表通配符

2.星号代表多个单词,井号代表一个单词

3.路由功能添加模糊匹配

4.消息产生者产生消息,把消息交给交换机

5.交换机根据key的规则模糊匹配到对应的队列,由队列的监听消费者接收消息消 费

（在我的理解看来就是routing查询的一种模糊匹配，就类似sql的模糊查询方 式）

如何保证RabbitMQ消息的顺序性?

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确 实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

消息如何分发?

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。通过路由可实现多消费的功能

消息怎么路由?

消息提供方->路由->一至多个队列消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。通过队列路由键，可以把队列绑定到交换器上。消息到达交换器后，RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）；常用的交换器主要分为一下三种： fanout：如果交换器收到消息，将会广播到所有绑定的队列上 direct：如果路由键完全匹配，消息就被投递到相应的队列

topic：可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时，可以使用通配符

消息基于什么传输?

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。

如何保证消息不被重复消费？或者说，如何保证消息消费时的幂等性?

先说为什么会重复消费：正常情况下，消费者在消费消息的时候，消费完毕后，会发送一个确认消息给消息队列，消息队列就知道该消息被消费了，就会将该消息从消息队列中删除；但是因为网络传输等等故障，确认信息没有传送到消息队列，导致消息队列不知道自己已经消费过该消息了，再次将消息分发给其他的消费者。

针对以上问题，一个解决思路是：保证消息的唯一性，就算是多次传输，不要让消息的多次消费带来影响；保证消息等幂性；比如：在写入消息队列的数据做唯一标示，消费消息时，根据唯一标识判断是否消费过；假设你有个系统，消费一条消息就往数据库里插入一条数据，要是你一个消息重复两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下是否已经消费过了，若是就直接扔了，这样不就保留了一条数据，从而保证了数据的正确性。

如何确保消息正确地发送至RabbitMQ？ 如何确保消息接收方消费了消息?

发送方确认模式将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。

一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。

如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（notacknowledged，未确认）消息。

发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。接收方确认机制消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。

这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了

Consumer 足够长的时间来处理消息。保证数据的最终一致性；下面罗列几种特殊情况

- 如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要去重）
- 如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。

如何保证RabbitMQ消息的可靠传输?

消息不可靠的情况可能是消息丢失，劫持等原因；丢失又分为：生产者丢失消息、消息列表丢失消息、消费者丢失消息；生产者丢失消息：从生产者弄丢数据这个角度来看，RabbitMQ提供 transaction和confirm模式来确保生产者不丢消息；

transaction机制就是说：发送消息前，开启事务（channel.txSelect()）,然后发送消息，如果发送过程中出现什么异常，事务就会回滚

（channel.txRollback()）,如果发送成功则提交事务

（channel.txCommit()）。然而，这种方式有个缺点：吞吐量下降；

confirm模式用的居多：一旦channel进入confirm模式，所有在该信道上发布的消息都将会被指派一个唯一的ID（从1开始），一旦消息被投递到所有匹配的队列之后；

rabbitMQ就会发送一个ACK给生产者（包含消息的唯一ID），这就使得生产者知道消息已经正确到达目的队列了；

如果rabbitMQ没能处理该消息，则会发送一个Nack消息给你，你可以进行重试操作。

消息队列丢数据：消息持久化。

处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。

这个持久化配置可以和confirm机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个Ack信号。

这样，如果消息持久化磁盘之前，rabbitMQ阵亡了，那么生产者收不到Ack信号，生产者会自动重发。

那么如何持久化呢?

这里顺便说一下吧，其实也很容易，就下面两步

\1. 将queue的持久化标识durable设置为true,则代表是一个持久的队列

\2. 发送消息的时候将deliveryMode=2

这样设置以后，即使rabbitMQ挂了，重启后也能恢复数据消费者丢失消息：消费者丢数据一般是因为采用了自动确认消息模式，改为手动确认消息即可！消费者在收到消息之后，处理消息之前，会自动回复RabbitMQ已收到消息；如果这时处理消息失败，就会丢失该消息；解决方案：处理消息成功后，手动回复确认消息。

为什么不应该对所有的 message 都使用持久化机制?

首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。

其次，message 的持久化机制用在 RabbitMQ 的内置 cluster 方案时会出现"坑爹"问题。矛盾点在于，若 message 设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的 owner node 出现异常后，在未重建该 queue 前，发往该 queue 的 message 将被 blackholed ；若 message 设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的 owner node 异常且无法重启的情况下，则该 queue 无法在其他 node 上重建，只能等待其 owner node 重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed 。

所以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想达到 100,000 条/秒以上的消息吞吐量（单 RabbitMQ 服务

器），则要么使用其他的方式来确保 message 的可靠 delivery ，要么使用非常快速的存储系统以支持全持久化（例如使用 SSD）。另外一种处理原则是：

仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

如何保证高可用的？RabbitMQ 的集群

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，

我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的?，没人生产用单机模式普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。

RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个queue 的完整数据。

如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

消息积压处理办法：临时紧急扩容：

先修复 consumer 的问题，确保其恢复消费速度，然后将现有 cnosumer 都停掉。

新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。

然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的

数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。

接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。

等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。

MQ中消息失效：假设你用的是 RabbitMQ，RabbtiMQ 是可以设置过期时间

的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是大量的数据会直接搞丢。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12 点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

mq消息队列块满了：如果消息积压在 mq 里，你很长时间都没有处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要

了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

设计MQ思路

比如说这个消息队列系统，我们从以下几个角度来考虑一下：首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，

broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。

如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有

磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。

其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。

能不能支持数据 0 丢失啊？可以的，参考我们之前说的那个 kafka 数据零丢失方案。

消息中间件如何保证消息的一致性

(1)主动方应用先把消息发送给消息中间件，消息状态标记为待确认；

(2)消息中间件收到消息之后，把消息持久化到消息存储中，但并不向被动方应用投递消息；

(3)消息中间件返回消息持久化结果（成功，或者失效），主动方应用根据返回结果进行判断如何

处理业务操作处理；

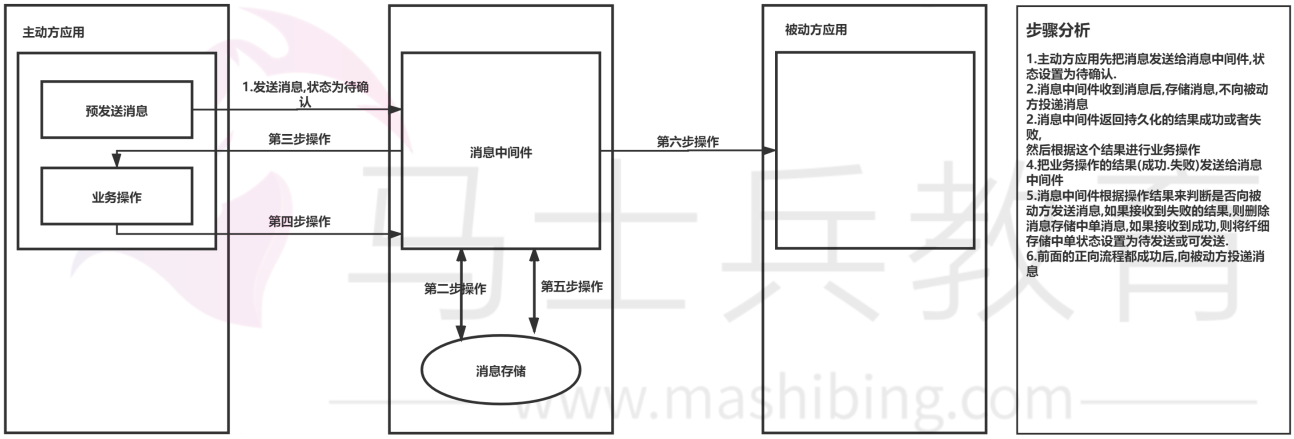
- ①失败：放弃业务操作处理，结束（必须向上层返回失败结果）
- ②成功：执行业务操作处理

(4)业务操作完成后，把业务操作结果（成功/失败）发送给消息中间件；

(5)消息中间件收到业务操作结果后，根据结果进行处理；

- ①失败：删除消息存储中的消息，结束；
- ②成功：更新消息存储中的消息状态为·待发送（可发送）·，紧接着执行消息投递；

(6)前面的正向流程都成功后，向被动方应用投递消息；



如何进行消息的重试机制?

Rocket重试机制，消息模式，刷盘方式

一、Consumer 批量消费（推模式）

可以通过

```
consumer.setConsumeMessageBatchMaxSize(10);//每次拉取10条
```

这里需要分为2种情况

- Consumer端先启动
- Consumer端后启动. 正常情况下：应该是Consumer需要先启动

注意：如果broker采用推模式的话，consumer先启动，会一条一条消息的消费，consumer后启动会才用批量消费

Consumer端先启动

1、Consumer.java

```
package quickstart;
import java.util.List;
import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;
/**
 * Consumer，订阅消息
 */
public class Consumer {
    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_4");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费 ,（消费顺序消息的时候设置）
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        consumer.subscribe("TopicTest", "*");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

                try {
                    System.out.println("msgs的长度" + msgs.size());
                    System.out.println(Thread.currentThread().getName() + " Receive New Messages: " + msgs);
                } catch (Exception e) {
                    e.printStackTrace();
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
```

```
        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

由于这里是Consumer先启动，所以他回去轮询MQ上是否有订阅队列的消息，由于每次producer插入一条，Consumer就拿一条所以测试结果如下（每次size都是1）

2、Consumer端后启动，也就是Producer先启动

由于这里是Consumer后启动，所以MQ上也就堆积了一堆数据，Consumer的

```
consumer.setConsumeMessageBatchMaxSize(10);//每次拉取10条
```

二、消息重试机制：消息重试分为2种

- ## 1、Producer端重试
- ## 2、Consumer端重试

1、Producer端重试

也就是Producer往MQ上发消息没有发送成功，我们可以设置发送失败重试的次数,发送并触发回调函数

```
          //设置重试的次数
            producer.setRetryTimesWhenSendFailed(3);
            //开启生产者
            producer.start();
            //创建一条消息
            Message msg = new Message("PushTopic", "push", "1",   "我是一条普通消息".getBytes());
            //发送消息
            SendResult result = producer.send(msg);
            //发送，并触发回调函数
            producer.send(msg, new SendCallback() {

                @Override
                //成功的回调函数
                public void onSuccess(SendResult sendResult) {
                    System.out.println(sendResult.getSendStatus());
                    System.out.println("成功了");
                }

                @Override
                //出现异常的回调函数
                public void onException(Throwable e) {
                System.out.println("失败了"+e.getMessage());

                }
            });
```

2、Consumer端重试

2.1、exception的情况，一般重复16次 10s、30s、1分钟、2分钟、3分钟等等

上面的代码中消费异常的情况返回

return ConsumeConcurrentlyStatus.RECONSUME_LATER;//重试

正常则返回：

return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;//成功

```
package quickstart;


import java.util.List;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;

/**
 * Consumer，订阅消息
 */
public class Consumer {
```

```
    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_4");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "*");

        consumer.registerMessageListener(new MessageListenerConcurrently() {

            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

                try {
                    // System.out.println("msgs的长度" + msgs.size());
                    System.out.println(Thread.currentThread().getName() + " Receive New Messages: " + msgs);
                    for (MessageExt msg : msgs) {
                        String msgbody = new String(msg.getBody(), "utf-8");
                        if (msgbody.equals("Hello RocketMQ 4")) {
                            System.out.println("======错误=======");
                            int a = 1 / 0;
                        }
                    }

                } catch (Exception e) {
                    e.printStackTrace();
                    if(msgs.get(0).getReconsumeTimes()==3){
                        //记录日志

                        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
                    }else{

                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;// 重试
                    }
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
            }
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

假如超过了多少次之后我们可以让他不再重试记录 日志。

if(msgs.get(0).getReconsumeTimes()==3){
//记录日志
return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
}

2.2超时的情况，这种情况MQ会无限制的发送给消费端。

就是由于网络的情况，MQ发送数据之后，Consumer端并没有收到导致超时。也就是消费端没有给我返回return 任何状态，这样的就认为没有到达Consumer端。

这里模拟Producer只发送一条数据。consumer端暂停1分钟并且不发送接收状态给MQ

```
package model;

import java.util.List;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;

/**
 * Consumer，订阅消息
 */
public class Consumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("message_consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
```

```
        consumer.setConsumeMessageBatchMaxSize(10);
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "*");

        consumer.registerMessageListener(new MessageListenerConcurrently() {

            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

                try {

                    // 表示业务处理时间
                    System.out.println("=========开始暂停===============");
                    Thread.sleep(60000);

                    for (MessageExt msg : msgs) {
                        System.out.println(" Receive New Messages: " + msg);
                    }

                } catch (Exception e) {
                    e.printStackTrace();
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;// 重试
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
            }
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

三、消费模式

1、集群消费

2、广播消费

rocketMQ默认是集群消费,我们可以通过在Consumer来支持广播消费

```
consumer.setMessageModel(MessageModel.BROADCASTING);// 广播消费
```

```
package model;

import java.util.List;

import com.alibaba.rocketmq.client.consumer.DefaultMQPushConsumer;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import com.alibaba.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import com.alibaba.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import com.alibaba.rocketmq.client.exception.MQClientException;
import com.alibaba.rocketmq.common.consumer.ConsumeFromWhere;
import com.alibaba.rocketmq.common.message.MessageExt;
import com.alibaba.rocketmq.common.protocol.heartbeat.MessageModel;

/**
 * Consumer，订阅消息
 */
public class Consumer2 {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("message_consumer");
        consumer.setNamesrvAddr("192.168.100.145:9876;192.168.100.146:9876");
        consumer.setConsumeMessageBatchMaxSize(10);
        consumer.setMessageModel(MessageModel.BROADCASTING);// 广播消费

        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "*");

        consumer.registerMessageListener(new MessageListenerConcurrently() {

            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {

                try {
```

```
                    for (MessageExt msg : msgs) {
                        System.out.println(" Receive New Messages: " + msg);
                    }

                } catch (Exception e) {
                    e.printStackTrace();
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;// 重试
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;// 成功
            }
        });

        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

四、conf下的配置文件说明

异步复制和同步双写主要是主和从的关系。消息需要实时消费的，就需要采用主从模式部署

异步复制:比如这里有一主一从，我们发送一条消息到主节点之后，这样消息就算从producer端发送成功了，然后通过异步复制的方法将数据复制到从节点

同步双写:比如这里有一主一从，我们发送一条消息到主节点之后，这样消息就并不算从producer端发送成功了，需要通过同步双写的方法将数据同步到从节点后， 才算数据发送成功。

如果rocketMq才用双master部署，Producer往MQ上写入20条数据 其中Master1中拉取了12条 。Master2中拉取了8 条，这种情况下，Master1宕机，那么我们消费数据的时候，只能消费到Master2中的8条，Master1中的12条默认持久化，不会丢失消息，需要Master1恢复之后这12条数据才能继续被消费，如果想保证消息实时消费，就才用双Master双Slave的模式

五、刷盘方式

同步刷盘：在消息到达MQ后，RocketMQ需要将数据持久化，同步刷盘是指数据到达内存之后，必须刷到commitlog日志之后才算成功，然后返回producer数据已经发送成功。

异步刷盘：，同步刷盘是指数据到达内存之后,返回producer说数据已经发送成功。，然后再写入commitlog日志。

commitlog：

commitlog就是来存储所有的元信息，包含消息体，类似于MySQL、Oracle的redolog,所以主要有CommitLog在，Consume Queue即使数据丢失，仍然可以恢复出来。

consumequeue：记录数据的位置,以便Consume快速通过consumequeue找到commitlog中的数据

传递保证语义：

- At most once：消息可能会丢，但绝不会重复传递。
- At least once：消息绝不会丢，但可能会重复传递。
- Exactly once： 每条消息只会被传递一次。

生产者的“Exactly once”语义方案

当生产者向Kafka发送消息，且正常得到响应的时候，可以确保生产者不会产生重复的消息。但是，如果生产者发送消息后，遇到网络问题，无法获取响应，生产者就无法判断该消息是否成功提交给了Kafka。根据生产者的机制，我们知道，当出现异常时，会进行消息重传，这就可能出现“At least one”语义。为了实现“Exactly once”语义，这里提供两个可选方案：

- 每个分区只有一个生产者写入消息，当出现异常或超时的情况时，生产者就要查询此分区的最后一个消息，用来决定后续操作是消息重传还是继续发送。
- 为每个消息添加一个全局唯一主键，生产者不做其他特殊处理，按照之前分析方式进行重传，由消费者对消息进行去重，实现“Exactly once”语义。

如果业务数据产生消息可以找到合适的字段作为主键，或是有一个全局ID生成器，可以优先考虑选用第二种方案。

消费者的“Exactly once”语义方案

为了实现消费者的“Exactly once”语义，在这里提供一种方案，供读者参考：消费者将关闭自动提交offset的功能且不再手动提交offset，这样就不使用Offsets Topic这个内部Topic记录其offset，而是由消费者自己保存offset。这里利用事务的原子性来实现“Exactly once”语义，我们将offset和消息处理结果放在一个事务中，事务执行成功则认为此消息被消费，否则事务回滚需要重新消费。当出现消费者宕机重启或Rebalance操作时，消费者可以从关系型数据库中找到对应的offset，然后调用KafkaConsumer.seek()方法手动设置消费位置，从此offset处开始继续消费。

ISR集合

ISR（In-SyncReplica）集合表示的是目前“可用”（alive）且消息量与Leader相差不多的副本集合，这是整个副本集合的一个子集。“可用”和“相差不多”都是很模糊的描述，其实际含义是ISR集合中的副本必须满足下面两个条件：

1. 副本所在节点必须维持着与ZooKeeper的连接。
2. 副本最后一条消息的offset与Leader副本的最后一条消息的offset之间的差值不能超出指定的阈值。

每个分区中的Leader副本都会维护此分区的ISR集合。写请求首先由Leader副本处理，之后Follower副本会从Leader上拉取写入的消息，这个过程会有一定的延迟，导致Follower副本中保存的消息略少于Leader副本，只要未超出阈值都是可以容忍的。如果一个Follower副本出现异常，比如：宕机，发生长时间GC而导致Kafka僵死或是网络断开连接导致长时间没有拉取消息进行同步，就会违反上面的两个条件，从而被Leader副本踢出ISR集合。当Follower副本从异常中恢复之后，会继续与Leader副本进行同步，当Follower副本"追上"（即最后一条消息的offset的差值小于指定阈值）Leader副本的时候，此Follower副本会被Leader副本重新加入到ISR中。

请说明什么是Apache Kafka?

Apache Kafka是由Apache开发的一种发布订阅消息系统，它是一个分布式的、分区的和重复的日志服务。

请说明什么是传统的消息传递方法?

传统的消息传递方法包括两种：

- 排队：在队列中，一组用户可以从服务器中读取消息，每条消息都发送给其中一个人。
- 发布-订阅：在这个模型中，消息被广播给所有的用户。

请说明Kafka相对传统技术有什么优势?

Apache Kafka与传统的消息传递技术相比优势之处在于：

- 快速:单一的Kafka代理可以处理成千上万的客户端，每秒处理数兆字节的读写操作。
- 可伸缩:在一组机器上对数据进行分区和简化，以支持更大的数据
- 持久:消息是持久性的，并在集群中进行复制，以防止数据丢失
- 设计:它提供了容错保证和持久性

在Kafka中broker的意义是什么?

在Kafka集群中，broker术语用于引用服务器。

Kafka服务器能接收到的最大信息是多少?

Kafka服务器可以接收到的消息的最大大小是1000000字节。

解释Kafka的Zookeeper是什么?我们可以在没有Zookeeper的情况下使用Kafka吗?

Zookeeper是一个开放源码的、高性能的协调服务，它用于Kafka的分布式应用。

不，不可能越过Zookeeper，直接联系Kafka broker。一旦Zookeeper停止工作，它就不能服务客户端请求。

Zookeeper主要用于在集群中不同节点之间进行通信

在Kafka中，它被用于提交偏移量，因此如果节点在任何情况下都失败了，它都可以从之前提交的偏移量中获取

除此之外，它还执行其他活动，如: leader检测、分布式同步、配置管理、识别新节点何时离开或连接、集群、节点实时状态等等。

解释Kafka的用户如何消费信息?

在Kafka中传递消息是通过使用sendfile API完成的。它支持将字节从套接口转移到磁盘，通过内核空间保存副本，并在内核用户之间调用内核。

解释如何提高远程用户的吞吐量?

如果用户位于与broker不同的数据中心，则可能需要调优套接口缓冲区大小，以对长网络延迟进行摊销。

解释一下，在数据制作过程中，你如何能从Kafka得到准确的信息?

在数据中，为了精确地获得Kafka的消息，你必须遵循两件事: 在数据消耗期间避免重复，在数据生产过程中避免重复。

这里有两种方法，可以在数据生成时准确地获得一个语义:

- 每个分区使用一个单独的写入器，每当你发现一个网络错误，检查该分区中的最后一条消息，以查看您的最后一次写入是否成功
- 在消息中包含一个主键(UUID或其他)，并在用户中进行反复制

解释如何减少ISR中的扰动?broker什么时候离开ISR?

ISR是一组与leaders完全同步的消息副本，也就是说ISR中包含了所有提交的消息。ISR应该总是包含所有的副本，直到出现真正的故障。如果一个副本从leader中脱离出来，将会从ISR中删除。

Kafka为什么需要复制?

Kafka的信息复制确保了任何已发布的消息不会丢失，并且可以在机器错误、程序错误或更常见些的软件升级中使用。

如果副本在ISR中停留了很长时间表明什么?

如果一个副本在ISR中保留了很长一段时间，那么它就表明，跟踪器无法像在leader收集数据那样快速地获取数据。

请说明如果首选的副本不在ISR中会发生什么?

如果首选的副本不在ISR中，控制器将无法将leadership转移到首选的副本。

有可能在生产后发生消息偏移吗?

在大多数队列系统中，作为生产者的类无法做到这一点，它的作用是触发并忘记消息。broker将完成剩下的工作，比如使用id进行适当的元数据处理、偏移量等。

作为消息的用户，你可以从Kafka broker中获得补偿。如果你注视SimpleConsumer类，你会注意到它会获取包括偏移量作为列表的MultiFetchResponse对象。此外，当你对Kafka消息进行迭代时，你会拥有包括偏移量和消息发送的MessageAndOffset对象。

kafka与传统的消息中间件对比

kafka与传统的消息中间件对比

RabbitMQ和kafka从几个角度简单的对比

业界对于消息的传递有多种方案和产品，本文就比较有代表性的两个MQ(rabbitMQ,kafka)进行阐述和做简单的对比，

在应用场景方面，

RabbitMQ,遵循AMQP协议，由内在高并发的erlanng语言开发，用在实时的对可靠性要求比较高的消息传递上。

kafka是Linkedin于2010年12月份开源的消息发布订阅系统,它主要用于处理活跃的流式数据,大数据量的数据处理上。

1)在架构模型方面，

RabbitMQ遵循AMQP协议，RabbitMQ的broker由Exchange,Binding,queue组成，其中exchange和binding组成了消息的路由键；客户端Producer通过连接channel和server进行通信，Consumer从queue获取消息进行消费（长连接，queue有消息会推送到consumer端，consumer循环从输入流读取数据）。rabbitMQ以broker为中心；有消息的确认机制。

kafka遵从一般的MQ结构，producer，broker，consumer，以consumer为中心，消息的消费信息保存的客户端consumer上，consumer根据消费的点，从broker上批量pull数据；无消息确认机制。

2)在吞吐量，

kafka具有高的吞吐量，内部采用消息的批量处理，zero-copy机制，数据的存储和获取是本地磁盘顺序批量操作，具有O(1)的复杂度，消息处理的效率很高。

rabbitMQ在吞吐量方面稍逊于kafka，他们的出发点不一样，rabbitMQ支持对消息的可靠的传递，支持事务，不支持批量的操作；基于存储的可靠性的要求存储可以采用内存或者硬盘。

3)在可用性方面，

rabbitMQ支持miror的queue，主queue失效，miror queue接管。

kafka的broker支持主备模式。

4)在集群负载均衡方面，

kafka采用zookeeper对集群中的broker、consumer进行管理，可以注册topic到zookeeper上；通过zookeeper的协调机制，producer保存对应topic的broker信息，可以随机或者轮询发送到broker上；并且producer可以基于语义指定分片，消息发送到broker的某分片上。

rabbitMQ的负载均衡需要单独的loadbalancer进行支持。

Kafka 对比 ActiveMQ

Kafka 是LinkedIn 开发的一个高性能、分布式的消息系统，广泛用于日志收集、流式数据处理、在线和离线消息分发等场景。虽然不是作为传统的MQ来设计，在大部分情况，Kafaka 也可以代替原先ActiveMQ 等传统的消息系统。

Kafka 将消息流按Topic 组织，保存消息的服务器称为Broker，消费者可以订阅一个或者多个Topic。为了均衡负载，一个Topic 的消息又可以划分到多个分区(Partition)，分区越多，Kafka并行能力和吞吐量越高。

Kafka 集群需要zookeeper 支持来实现集群，最新的kafka 发行包中已经包含了zookeeper，部署的时候可以在一台服务器上同时启动一个zookeeper Server 和 一个Kafka Server，也可以使用已有的其他zookeeper集群。

和传统的MQ不同，消费者需要自己保留一个offset，从kafka 获取消息时，只拉去当前offset 以后的消息。Kafka 的scala/java 版的client 已经实现了这部分的逻辑，将offset 保存到zookeeper 上。每个消费者可以选择一个id，同样id 的消费者对于同一条消息只会收到一次。一个Topic 的消费者如果都使用相同的id，就是传统的 Queue；如果每个消费者都使用不同的id, 就是传统的pub-sub.

ActiveMQ和Kafka，前者完全实现了JMS的规范，后者看上去有一些“野路子”，并没有纠结于JMS规范，剑走偏锋的设计了另一套吞吐非常高的分布式发布-订阅消息系统，目前非常流行。接下来我们结合三个点（消息安全性，服务器的稳定性容错性以及吞吐量）来分别谈谈这两个消息中间件。今天我们谈Kafka

01 性能怪兽Kafka

Kafka是LinkedIn开源的分布式发布-订阅消息系统，目前归属于Apache定级项目。”Apache Kafka is publish-subscribe messaging rethought as a distributed commit log.”，官网首页的一句话高度概括其职责。Kafka并没有遵守JMS规范，他只用文件系统来管理消息的生命周期。Kafka的设计目标是：

（1）以时间复杂度为O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间复杂度的访问性能。
（2）高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条以上消息的传输。
（3）支持Kafka Server间的消息分区，及分布式消费，同时保证每个Partition内的消息顺序传输。
（4）同时支持离线数据处理和实时数据处理。
（5）Scale out：支持在线水平扩展。

所以，不像AMQ，Kafka从设计开始极为高可用为目的，天然HA。broker支持集群，消息亦支持负载均衡，还有副本机制。同样，Kafka也是使用Zookeeper管理集群节点信息，包括consumer的消费信息也是保存在zk中，下面我们分话题来谈：

1）消息的安全性

Kafka集群中的Leader负责某一topic的某一partition的消息的读写，理论上consumer和producer只与该Leader 节点打交道，一个集群里的某一broker即是Leader的同时也可以担当某一partition的follower，即Replica。Kafka分配Replica的算法如下：

（1）将所有Broker（假设共n个Broker）和待分配的Partition排序
（2）将第i个Partition分配到第（i mod n）个Broker上
（3）将第i个Partition的第j个Replica分配到第（(i + j) mode n）个Broker上

同时，Kafka与Replica既非同步也不是严格意义上的异步。一个典型的Kafka发送-消费消息的过程如下：首先首先Producer消息发送给某Topic的某Partition的Leader，Leader先是将消息写入本地Log，同时follower（如果落后过多将会被踢出出 Replica列表）从Leader上pull消息，并且在未写入log的同时即向Leader发送ACK的反馈，所以对于某一条已经算作commit的消息来讲，在某一时刻，其存在于Leader的log中，以及Replica的内存中。这可以算作一个危险的情况（听起来吓人），因为如果此时集群挂了这条消息就算丢失了，但结合producer的属性（request.required.acks=2 当所有follower都收到消息后返回ack）可以保证在绝大多数情况下消息的安全性。当消息算作commit的时候才会暴露给consumer，并保证at-least-once的投递原则。

2）服务的稳定容错性
前面提到过，Kafka天然支持HA，整个leader/follower机制通过zookeeper调度，它在所有broker中选出一个 controller，所有Partition的Leader选举都由controller决定，同时controller也负责增删Topic以及 Replica的重新分配。如果Leader挂了，集群将在ISR（in-sync replicas）中选出新的Leader，选举基本原则是：新的Leader必须拥有原来的Leader commit过的所有消息。假如所有的follower都挂了，Kafka会选择第一个“活”过来的Replica（不一定是ISR中的）作为 Leader，因为如果此时等待ISR中的Replica是有风险的，假如所有的ISR都无法“活”，那此partition将会变成不可用。
3） 吞吐量
Leader节点负责某一topic（可以分成多个partition）的某一partition的消息的读写，任何发布到此partition的消息都会被直接追加到log文件的尾部，因为每条消息都被append到该partition中，是顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证），同时通过合理的partition，消息可以均匀的分布在不同的partition里面。 Kafka基于时间或者partition的大小来删除消息，同时broker是无状态的，consumer的消费状态(offset)是由 consumer自己控制的（每一个consumer实例只会消费某一个或多个特定partition的数据，而某个partition的数据只会被某一个特定的consumer实例所消费），也不需要broker通过锁机制去控制消息的消费，所以吞吐量惊人，这也是Kafka吸引人的地方。
最后说下由于zookeeper引起的脑裂（Split Brain）问题：每个consumer分别单独通过Zookeeper判断哪些partition down了，那么不同consumer从Zookeeper“看”到的view就可能不一样，这就会造成错误的reblance尝试。而且有可能所有的 consumer都认为rebalance已经完成了，但实际上可能并非如此。

如果在MQ的场景下，将Kafka 和 ActiveMQ 相比:

Kafka 的优点

分布式可高可扩展。Kafka 集群可以透明的扩展，增加新的服务器进集群。

高性能。Kafka 的性能大大超过传统的ActiveMQ、RabbitMQ等MQ 实现，尤其是Kafka 还支持batch 操作。下图是linkedin 的消费者性能压测结果:

容错。Kafka每个Partition的数据都会复制到几台服务器上。当某个Broker故障失效时，ZooKeeper服务将通知生产者和消费者，生产者和消费者转而使用其它Broker。

Kafka 的不利

重复消息。Kafka 只保证每个消息至少会送达一次，虽然几率很小，但一条消息有可能会被送达多次。
消息乱序。虽然一个Partition 内部的消息是保证有序的，但是如果一个Topic 有多个Partition，Partition 之间的消息送达不保证有序。
复杂性。Kafka需要zookeeper 集群的支持，Topic通常需要人工来创建，部署和维护较一般消息队列成本更高

KAFKA：如何做到1秒发布百万级条消息

KAFKA：如何做到1秒发布百万级条消息

KAFKA：如何做到1秒发布百万级条消息。KAFKA是分布式发布-订阅消息系统，是一个分布式的，可划分的，冗余备份的持久性的日志服务。它主要用于处理活跃的流式数据。

现在被广泛地应用于构建实时数据管道和流应用的场景中，具有横向扩展，容错，快等优点，并已经运行在众多大中型公司的生产环境中，成功应用于大数据领域，本文分享一下我所了解的KAFKA。

1 KAFKA高吞吐率性能揭秘

KAFKA的第一个突出特定就是“快”，而且是那种变态的“快”，在普通廉价的虚拟机器上，比如一般SAS盘做的虚拟机上，据LINDEDIN统计，最新的数据是每天利用KAFKA处理的消息超过1万亿条，在峰值时每秒钟会发布超过百万条消息，就算是在内存和CPU都不高的情况下，Kafka的速度最高可以达到每秒十万条数据，并且还能持久化存储。

作为消息队列，要承接读跟写两块的功能，首先是写，就是消息日志写入KAFKA，那么，KAFKA在“写”上是怎么做到写变态快呢?

1.1 KAFKA让代码飞起来之写得快

首先，可以使用KAFKA提供的生产端API发布消息到1个或多个Topic(主题)的一个(保证数据的顺序)或者多个分区(并行处理，但不一定保证数据顺序)。Topic可以简单理解成一个数据类别，是用来区分不同数据的。

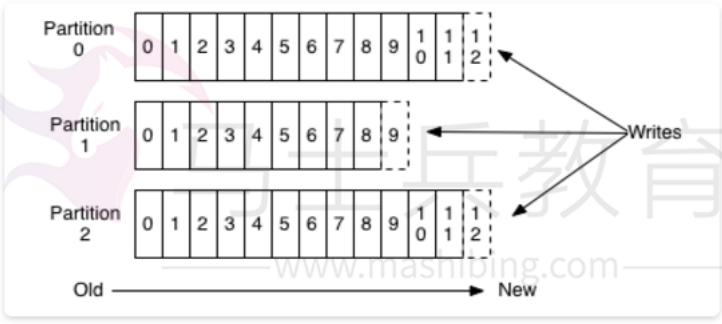
KAFKA维护一个Topic中的分区log，以顺序追加的方式向各个分区中写入消息，每个分区都是不可变的消息队列。分区中的消息都是以k-v形式存在。

? k表示offset，称之为偏移量，一个64位整型的唯一标识，offset代表了Topic分区中所有消息流中该消息的起始字节位置。

? v就是实际的消息内容，每个分区中的每个offset都是唯一存在的，所有分区的消息都是一次写入，在消息未过期之前都可以调整offset来实现多次读取。

以上提到KAFKA“快”的第一个因素：消息顺序写入磁盘。

我们知道现在的磁盘大多数都还是机械结构(SSD不在讨论的范围内)，如果将消息以随机写的方式存入磁盘，就会按柱面、磁头、扇区的方式进行(寻址过程)，缓慢的机械运动(相对内存)会消耗大量时间，导致磁盘的写入速度只能达到内存写入速度的几百万分之一，为了规避随机写带来的时间消耗，KAFKA采取顺序写的方式存储数据，如下图所示：



新来的消息只能追加到已有消息的末尾，并且已经生产的消息不支持随机删除以及随机访问，但是消费者可以通过重置offset的方式来访问已经消费过的数据。

即使顺序读写，过于频繁的大量小I/O操作一样会造成磁盘的瓶颈，所以KAFKA在此处的处理是把这些消息集合在一起批量发送，这样减少对磁盘IO的过度读写，而不是一次发送单个消息。

另一个是无效率的字节复制，尤其是在负载比较高的情况下影响是显着的。为了避免这种情况，KAFKA采用由Producer，broker和consumer共享的标准化二进制消息格式，这样数据块就可以在它们之间自由传输，无需转换，降低了字节复制的成本开销。

同时，KAFKA采用了MMAP(Memory Mapped Files，内存映射文件)技术。很多现代操作系统都大量使用主存做磁盘缓存，一个现代操作系统可以将内存中的所有剩余空间用作磁盘缓存，而当内存回收的时候几乎没有性能损失。

由于KAFKA是基于JVM的，并且任何与Java内存使用打过交道的人都知道两件事：

? 对象的内存开销非常高，通常是实际要存储数据大小的两倍;

? 随着数据的增加，java的垃圾收集也会越来越频繁并且缓慢。

基于此，使用文件系统，同时依赖页面缓存就比使用其他数据结构和维护内存缓存更有吸引力：

? 不使用进程内缓存，就腾出了内存空间，可以用来存放页面缓存的空间几乎可以翻倍。

? 如果KAFKA重启，进行内缓存就会丢失，但是使用操作系统的页面缓存依然可以继续使用。

可能有人会问KAFKA如此频繁利用页面缓存，如果内存大小不够了怎么办?

KAFKA会将数据写入到持久化日志中而不是刷新到磁盘。实际上它只是转移到了内核的页面缓存。

利用文件系统并且依靠页缓存比维护一个内存缓存或者其他结构要好，它可以直接利用操作系统的页缓存来实现文件到物理内存的直接映射。完成映射之后对物理内存的操作在适当时候会被同步到硬盘上。

1.2 KAFKA让代码飞起来之读得快

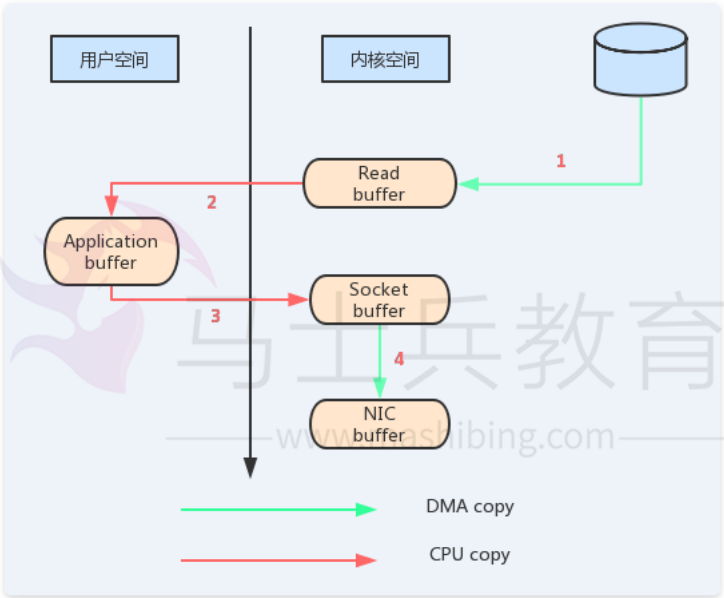
KAFKA除了接收数据时写得快，另外一个特点就是推送数据时发得快。

KAFKA这种消息队列在生产端和消费端分别采取的push和pull的方式，也就是你生产端可以认为KAFKA是个无底洞，有多少数据可以使劲往里面推送，消费端则是根据自己的消费能力，需要多少数据，你自己过来KAFKA这里拉取，KAFKA能保证只要这里有数据，消费端需要多少，都尽可以自己过来拿。

▲零拷贝

具体到消息的落地保存，broker维护的消息日志本身就是文件的目录，每个文件都是二进制保存，生产者和消费者使用相同的格式来处理。维护这个公共的格式并允许优化最重要的操作：网络传输持久性日志块。 现代的unix操作系统提供一个优化的代码路径，用于将数据从页缓存传输到socket;在Linux中，是通过sendfile系统调用来完成的。Java提供了访问这个系统调用的方法：FileChannel.transferTo API。

要理解senfile的影响，重要的是要了解将数据从文件传输到socket的公共数据路径，如下图所示，数据从磁盘传输到socket要经过以下几个步骤：



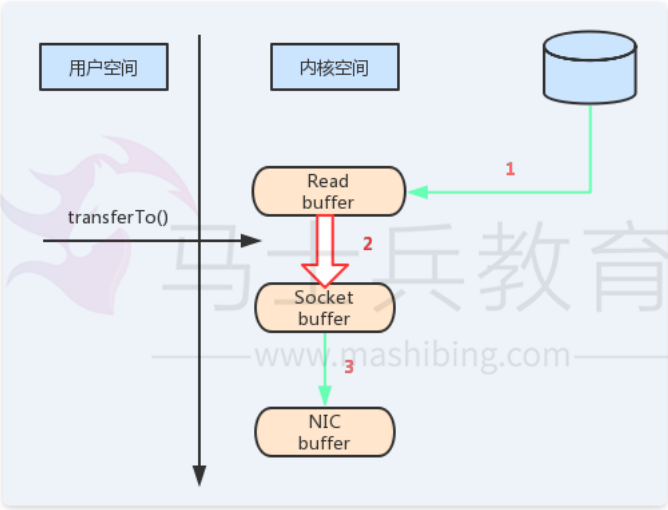
? 操作系统将数据从磁盘读入到内核空间的页缓存

? 应用程序将数据从内核空间读入到用户空间缓存中

? 应用程序将数据写回到内核空间到socket缓存中

? 操作系统将数据从socket缓冲区复制到网卡缓冲区，以便将数据经网络发出

这里有四次拷贝，两次系统调用，这是非常低效的做法。如果使用sendfile，只需要一次拷贝就行：允许操作系统将数据直接从页缓存发送到网络上。所以在这个优化的路径中，只有最后一步将数据拷贝到网卡缓存中是需要的。



常规文件传输和zeroCopy方式的性能对比：

File size	Normal file transfer (ms)	transferTo (ms)
7MB	156	45
21MB	337	128
63MB	843	387
98MB	1320	617
200MB	2124	1150
350MB	3631	1762
700MB	13498	4422
1GB	18399	8537

常规文件传输和zeroCopy方式的性能对比

假设一个Topic有多个消费者的情况， 并使用上面的零拷贝优化，数据被复制到页缓存中一次，并在每个消费上重复使用，而不是存储在存储器中，也不在每次读取时复制到用户空间。 这使得以接近网络连接限制的速度消费消息。

这种页缓存和sendfile组合，意味着KAFKA集群的消费者大多数都完全从缓存消费消息，而磁盘没有任何读取活动。

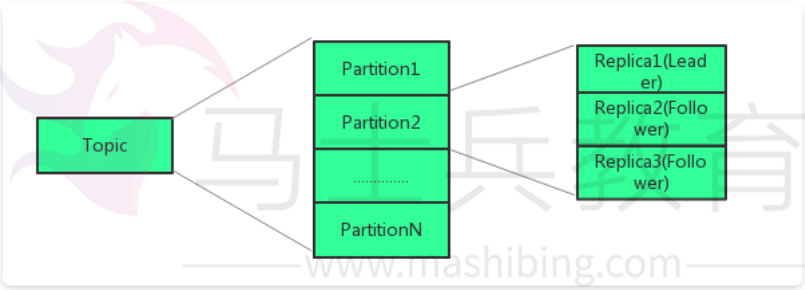
▲批量压缩

在很多情况下，系统的瓶颈不是CPU或磁盘，而是网络带宽，对于需要在广域网上的数据中心之间发送消息的数据流水线尤其如此。所以数据压缩就很重要。可以每个消息都压缩，但是压缩率相对很低。所以KAFKA使用了批量压缩，即将多个消息一起压缩而不是单个消息压缩。

KAFKA允许使用递归的消息集合，批量的消息可以通过压缩的形式传输并且在日志中也可以保持压缩格式，直到被消费者解压缩。

KAFKA支持Gzip和Snappy压缩协议。

2 KAFKA数据可靠性深度解读



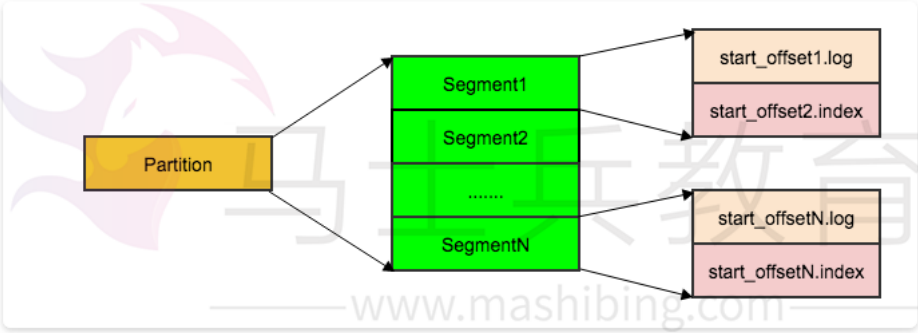
KAFKA的消息保存在Topic中，Topic可分为多个分区，为保证数据的安全性，每个分区又有多个Replia。

? 多分区的设计的特点：

1.为了并发读写，加快读写速度;

2.是利用多分区的存储，利于数据的均衡;

3.是为了加快数据的恢复速率，一但某台机器挂了，整个集群只需要恢复一部分数据，可加快故障恢复的时间。



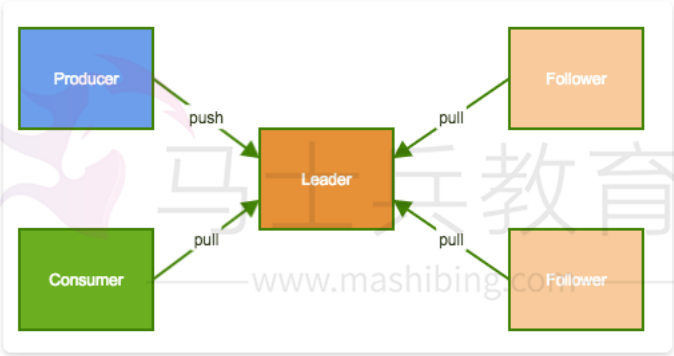
每个Partition分为多个Segment，每个Segment有.log和.index 两个文件，每个log文件承载具体的数据，每条消息都有一个递增的offset，Index文件是对log文件的索引，Consumer查找offset时使用的是二分法根据文件名去定位到哪个Segment，然后解析msg，匹配到对应的offset的msg。

2.1 Partition recovery过程

每个Partition会在磁盘记录一个RecoveryPoint,，记录已经flush到磁盘的最大offset。当broker 失败重启时，会进行loadLogs。首先会读取该Partition的RecoveryPoint，找到包含RecoveryPoint的segment及以后的segment， 这些segment就是可能没有完全flush到磁盘segments。然后调用segment的recover，重新读取各个segment的msg，并重建索引。每次重启KAFKA的broker时，都可以在输出的日志看到重建各个索引的过程。

2.2 数据同步

Producer和Consumer都只与Leader交互，每个Follower从Leader拉取数据进行同步。




```
Topic:light-ubas-raw    PartitionCount:1        ReplicationFactor:2     Configs:retention.ms=2592000000
        Topic: light-ubas-raw   Partition: 0    Leader: 2       Replicas: 2,4   Isr: 2,4
```

如上图所示，ISR是所有不落后的replica集合，不落后有两层含义：距离上次FetchRequest的时间不大于某一个值或落后的消息数不大于某一个值，Leader失败后会从ISR中随机选取一个Follower做Leader，该过程对用户是透明的。

当Producer向Broker发送数据时,可以通过request.required.acks参数设置数据可靠性的级别。

此配置是表明当一次Producer请求被认为完成时的确认值。特别是，多少个其他brokers必须已经提交了数据到它们的log并且向它们的Leader确认了这些信息。

?典型的值：

0： 表示Producer从来不等待来自broker的确认信息。这个选择提供了最小的时延但同时风险最大(因为当server宕机时，数据将会丢失)。

1：表示获得Leader replica已经接收了数据的确认信息。这个选择时延较小同时确保了server确认接收成功。

-1：Producer会获得所有同步replicas都收到数据的确认。同时时延最大，然而，这种方式并没有完全消除丢失消息的风险，因为同步replicas的数量可能是1。如果你想确保某些replicas接收到数据，那么你应该在Topic-level设置中选项min.insync.replicas设置一下。

仅设置 acks= -1 也不能保证数据不丢失,当ISR列表中只有Leader时,同样有可能造成数据丢失。要保证数据不丢除了设置acks=-1，还要保证ISR的大小大于等于2。

具体参数设置：

request.required.acks:设置为-1 等待所有ISR列表中的Replica接收到消息后采算写成功。

min.insync.replicas: 设置为>=2,保证ISR中至少两个Replica。

Producer：要在吞吐率和数据可靠性之间做一个权衡。

KAFKA作为现代消息中间件中的佼佼者，以其速度和高可靠性赢得了广大市场和用户青睐，其中的很多设计理念都是非常值得我们学习的，本文所介绍的也只是冰山一角，希望能够对大家了解KAFKA有一定的作用。

kafka文件存储

据kafka官网吹，如果随机写入磁盘，速度就只有100KB每秒。顺序写入的话，7200转/s的磁盘就能达到惊人的600MB每秒！

操作系统对文件访问做了优化，文件会在内核空间分页做缓存（pageCache）。写入时先写入pageCache。由操作系统来决定何时统一写入磁盘。操作系统会使用顺序写入。

Kafka深入理解-1：Kafka高效的文件存储设计

Kafka是什么

> Kafka是最初由Linkedin公司开发，是一个分布式、分区的、多副本的、多订阅者，基于zookeeper协调的分布式日志系统(也可以当做MQ系统)，常见可以用于web/nginx日志、访问日志，消息服务等等，Linkedin于2010年贡献给了Apache基金会并成为顶级开源项目。

1.前言

一个商业化消息队列的性能好坏，其文件存储机制设计是衡量一个消息队列服务技术水平和最关键指标之一。
下面将从Kafka文件存储机制和物理结构角度，分析Kafka是如何实现高效文件存储，及实际应用效果。

2.Kafka文件存储机制

Kafka部分名词解释如下：

- Broker：消息中间件处理结点，一个Kafka节点就是一个broker，多个broker可以组成一个Kafka集群。
- Topic：一类消息，例如page view日志、click日志等都可以以topic的形式存在，Kafka集群能够同时负责多个topic的分发。
- Partition：topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列。
- Segment：partition物理上由多个segment组成，下面2.2和2.3有详细说明。
- offset：每个partition都由一系列有序的、不可变的消息组成，这些消息被连续的追加到partition中。partition中的每个消息都有一个连续的序列号叫做offset,用于partition唯一标识一条消息.

分析过程分为以下4个步骤：

- topic中partition存储分布
- partiton中文件存储方式
- partiton中segment文件存储结构
- 在partition中如何通过offset查找message

通过上述4过程详细分析，我们就可以清楚认识到kafka文件存储机制的奥秘。

2.1 topic中partition存储分布

假设实验环境中Kafka集群只有一个broker，xxx/message-folder为数据文件存储根目录，在Kafka broker中server.properties文件配置(参数log.dirs=xxx/message-folder)，例如创建2个topic名称分别为report_push、launch_info, partitions数量都为partitions=4
存储路径和目录规则为：
xxx/message-folder

```
              |--report_push-0
              |--report_push-1
              |--report_push-2
              |--report_push-3
              |--launch_info-0
              |--launch_info-1
              |--launch_info-2
              |--launch_info-3
```

在Kafka文件存储中，同一个topic下有多个不同partition，每个partition为一个目录，partiton命名规则为topic名称+有序序号，第一个partiton序号从0开始，序号最大值为partitions数量减1。

2.2 partiton中文件存储方式

下面示意图形象说明了partition中文件存储方式:

- 每个partion(目录)相当于一个巨型文件被平均分配到多个大小相等segment(段)数据文件中。但每个段segment file消息数量不一定相等，这种特性方便old segment file快速被删除。
- 每个partiton只需要支持顺序读写就行了，segment文件生命周期由服务端配置参数决定。

这样做的好处就是能快速删除无用文件，有效提高磁盘利用率。

2.3 partiton中segment文件存储结构

读者从2.2节了解到Kafka文件系统partition存储方式，本节深入分析partion中segment file组成和物理结构。

- segment file组成：由2大部分组成，分别为index file和data file，此2个文件一一对应，成对出现，后缀".index"和“.log”分别表示为segment索引文件、数据文件.
- segment文件命名规则：partion全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值。数值最大为64位long大小，19位数字字符长度，没有数字用0填充。

下面文件列表是笔者在Kafka broker上做的一个实验，创建一个topicXXX包含1 partition，设置每个segment大小为500MB,并启动producer向Kafka broker写入大量数据,如下图2所示segment文件列表形象说明了上述2个规则：

以上述一对segment file文件为例，说明segment中index<—->data file对应关系物理结构如下：

上述索引文件存储大量元数据，数据文件存储大量消息，索引文件中元数据指向对应数据文件中message的物理偏移地址。
其中以索引文件中元数据3,497为例，依次在数据文件中表示第3个message(在全局partiton表示第368772个message)、以及该消息的物理偏移地址为497。

从上述了解到segment data file由许多message组成，下面详细说明message物理结构：
参数说明：

关键字	解释说明
8 byte offset	在parition(分区)内的每条消息都有一个有序的id号，这个id号被称为偏移(offset),它可以唯一确定每条消息在parition(分区)内的位置。即offset表示partiion的第多少message
4 byte message size	message大小
4 byte CRC32	用crc32校验message
1 byte “magic"	表示本次发布Kafka服务程序协议版本号
1 byte “attributes"	表示为独立版本、或标识压缩类型、或编码类型。
4 byte key length	表示key的长度,当key为-1时，K byte key字段不填
K byte key	可选
value bytes payload	表示实际消息数据。

2.4 在partition中如何通过offset查找message

例如读取offset=368776的message，需要通过下面2个步骤查找。

- 第一步查找segment file
  上述图2为例，其中00000000000000000000.index表示最开始的文件，起始偏移量(offset)为0.第二个文件00000000000000368769.index的消息量起始偏移量为368770 = 368769 + 1.同样，第三个文件00000000000000737337.index的起始偏移量为737338=737337 + 1，其他后续文件依次类推，以起始偏移量命名并排序这些文件，只要根据offset **二分查找**文件列表，就可以快速定位到具体文件。
  当offset=368776时定位到00000000000000368769.index|log
- 第二步通过segment file查找message
  通过第一步定位到segment file，当offset=368776时，依次定位到00000000000000368769.index的元数据物理位置和00000000000000368769.log的物理偏移地址，然后再通过00000000000000368769.log顺序查找直到offset=368776为止。

从上述图3可知这样做的优点，segment index file采取稀疏索引存储方式，它减少索引文件大小，通过mmap可以直接内存操作，稀疏索引为数据文件的每个对应message设置一个元数据指针,它比稠密索引节省了更多的存储空间，但查找起来需要消耗更多的时间。

3 Kafka文件存储机制–实际运行效果

实验环境：

- Kafka集群：由2台虚拟机组成
- cpu：4核
- 物理内存：8GB
- 网卡：千兆网卡
- jvm heap: 4GB

从上述可以看出，Kafka运行时很少有大量读磁盘的操作，主要是定期批量写磁盘操作，因此操作磁盘很高效。这跟Kafka文件存储中读写message的设计是息息相关的。Kafka中读写message有如下特点:

写message

- 消息从java堆转入page cache(即物理内存)。
- 由异步线程刷盘,消息从page cache刷入磁盘。

读message

- 消息直接从page cache转入socket发送出去。
- 当从page cache没有找到相应数据时，此时会产生磁盘IO,从磁盘Load消息到page cache,然后直接从socket发出去

4.总结

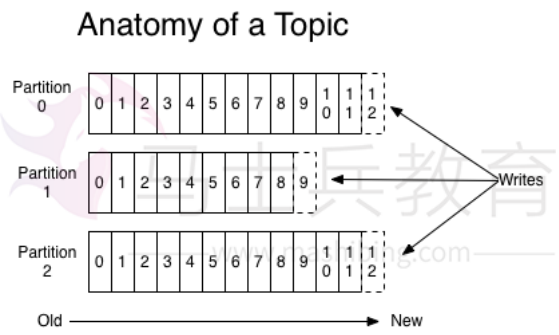
Kafka高效文件存储设计特点

- Kafka把topic中一个parition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。
- 通过索引信息可以快速定位message和确定response的最大大小。
- 通过index元数据全部映射到memory，可以避免segment file的IO磁盘操作。
- 通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小。

kafka的log存储解析——topic的分区partition分段segment以及索引等

引言

Kafka中的Message是以topic为基本单位组织的，不同的topic之间是相互独立的。每个topic又可以分成几个不同的partition(每个topic有几个partition是在创建topic时指定的)，每个partition存储一部分Message。借用官方的一张图，可以直观地看到topic和partition的关系。



partition是以文件的形式存储在文件系统中，比如，创建了一个名为page_visits的topic，其有5个partition，那么在Kafka的数据目录中(由配置文件中的log.dirs指定的)中就有这样5个目录: page_visits-0， page_visits-1，page_visits-2，page_visits-3，page_visits-4，其命名规则为<topic_name>-<partition_id>，里面存储的分别就是这5个partition的数据。

接下来，本文将分析partition目录中的文件的存储格式和相关的代码所在的位置。

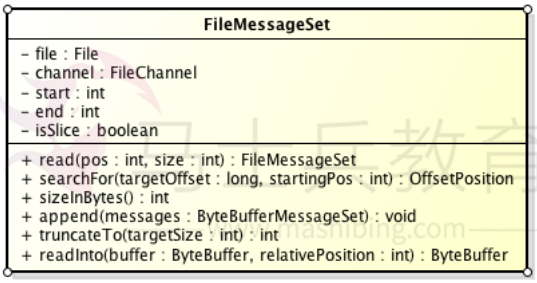
Partition的数据文件

Partition中的每条Message由offset来表示它在这个partition中的偏移量，这个offset不是该Message在partition数据文件中的实际存储位置，而是逻辑上一个值，它唯一确定了partition中的一条Message。因此，可以认为offset是partition中Message的id。partition中的每条Message包含了以下三个属性：

- offset
- MessageSize
- data

其中offset为long型，MessageSize为int32，表示data有多大，data为message的具体内容。它的格式和Kafka通讯协议中介绍的MessageSet格式是一致。

Partition的数据文件则包含了若干条上述格式的Message，按offset由小到大排列在一起。它的实现类为FileMessageSet，类图如下：



它的主要方法如下：

- append: 把给定的ByteBufferMessageSet中的Message写入到这个数据文件中。
- searchFor: 从指定的startingPosition开始搜索找到第一个Message其offset是大于或者等于指定的offset，并返回其在文件中的位置Position。它的实现方式是从startingPosition开始读取12个字节，分别是当前MessageSet的offset和size。如果当前offset小于指定的offset，那么将position向后移动LogOverHead+MessageSize（其中LogOverHead为offset+messagesize，为12个字节）。
- read：准确名字应该是slice，它截取其中一部分返回一个新的FileMessageSet。它不保证截取的位置数据的完整性。
- sizeInBytes: 表示这个FileMessageSet占有了多少字节的空间。
- truncateTo: 把这个文件截断，这个方法不保证截断位置的Message的完整性。
- readInto: 从指定的相对位置开始把文件的内容读取到对应的ByteBuffer中。

我们来思考一下，如果一个partition只有一个数据文件会怎么样?

1. 新数据是添加在文件末尾（调用FileMessageSet的append方法），不论文件数据文件有多大，这个操作永远都是O(1)的。
2. 查找某个offset的Message（调用FileMessageSet的searchFor方法）是顺序查找的。因此，如果数据文件很大的话，查找的效率就低。

那Kafka是如何解决查找效率的的问题呢？有两大法宝：1) 分段 2) 索引。

数据文件的分段

Kafka解决查询效率的手段之一是将数据文件分段，比如有100条Message，它们的offset是从0到99。假设将数据文件分成5段，第一段为0-19，第二段为20-39，以此类推，每段放在一个单独的数据文件里面，数据文件以该段中最小的offset命名。这样在查找指定offset的Message的时候，用二分查找就可以定位到该Message在哪个段中。

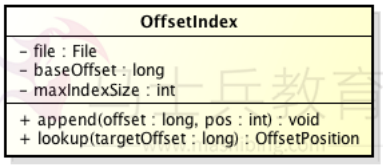
为数据文件建索引

数据文件分段使得可以在一个较小的数据文件中查找对应offset的Message了，但是这依然需要顺序扫描才能找到对应offset的Message。为了进一步提高查找的效率，Kafka为每个分段后的数据文件建立了索引文件，文件名与数据文件的名字是一样的，只是文件扩展名为.index。
索引文件中包含若干个索引条目，每个条目表示数据文件中一条Message的索引。索引包含两个部分（均为4个字节的数字），分别为相对offset和position。

- 相对offset：因为数据文件分段以后，每个数据文件的起始offset不为0，相对offset表示这条Message相对于其所属数据文件中最小的offset的大小。举例，分段后的一个数据文件的offset是从20开始，那么offset为25的Message在index文件中的相对offset就是25-20 = 5。存储相对offset可以减小索引文件占用的空间。
- position，表示该条Message在数据文件中的绝对位置。只要打开文件并移动文件指针到这个position就可以读取对应的Message了。

index文件中并没有为数据文件中的每条Message建立索引，而是采用了稀疏存储的方式，每隔一定字节的数据建立一条索引。这样避免了索引文件占用过多的空间，从而可以将索引文件保留在内存中。但缺点是没有建立索引的Message也不能一次定位到其在数据文件的位置，从而需要做一次顺序扫描，但是这次顺序扫描的范围就很小了。

在Kafka中，索引文件的实现类为OffsetIndex，它的类图如下：



主要的方法有：

- append方法，添加一对offset和position到index文件中，这里的offset将会被转成相对的offset。
- lookup, 用二分查找的方式去查找小于或等于给定offset的最大的那个offset

小结

我们以几张图来总结一下Message是如何在Kafka中存储的，以及如何查找指定offset的Message的。

Message是按照topic来组织，每个topic可以分成多个的partition，比如：有5个partition的名为为page_visits的topic的目录结构为：

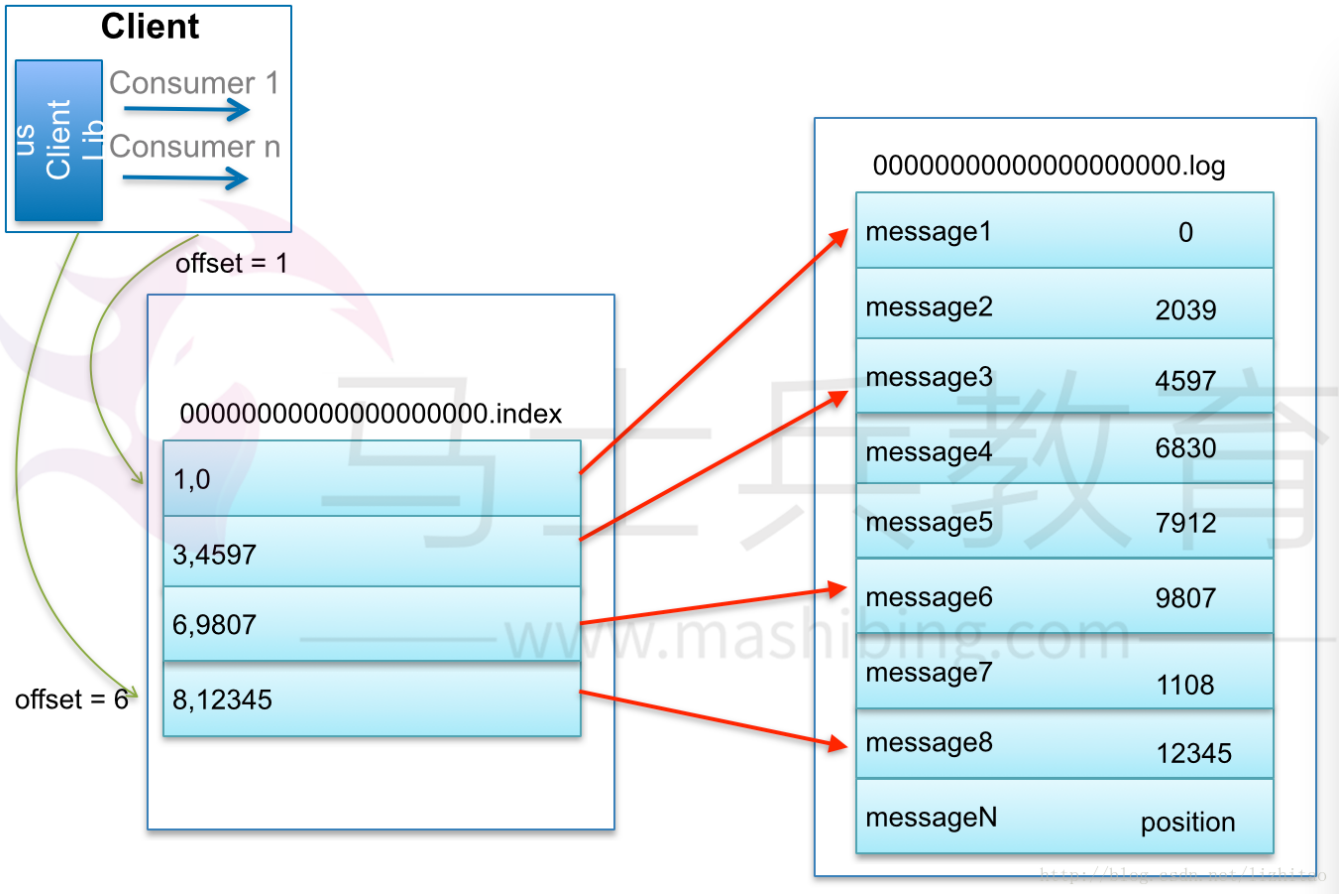
```
drwxrwxr-x 2 vagrant vagrant 4096 Jan 10 13:59 page_visits-0
drwxrwxr-x 2 vagrant vagrant 4096 Jan 20 08:51 page_visits-1
drwxrwxr-x 2 vagrant vagrant 4096 Jan 20 08:51 page_visits-2
drwxrwxr-x 2 vagrant vagrant 4096 Jan 20 08:51 page_visits-3
drwxrwxr-x 2 vagrant vagrant 4096 Jan 20 08:51 page_visits-4
```

partition是分段的，每个段叫LogSegment，包括了一个数据文件和一个索引文件，下图是某个partition目录下的文件：

```
00000000000000000000.index
00000000000000000000.log
00000000000000368769.index
00000000000000368769.log
00000000000000737337.index
00000000000000737337.log
00000000000001105814.index
00000000000001105814.log
```

可以看到，这个partition有4个LogSegment。

借用博主@lizhitao博客上的一张图来展示是如何查找Message的。



比如：要查找绝对offset为7的Message：

1. 首先是用二分查找确定它是在哪个LogSegment中，自然是在第一个Segment中。
2. 打开这个Segment的index文件，也是用二分查找找到offset小于或者等于指定offset的索引条目中最大的那个offset。自然offset为6的那个索引是我们要找的，通过索引文件我们知道offset为6的Message在数据文件中的位置为9807。
3. 打开数据文件，从位置为9807的那个地方开始顺序扫描直到找到offset为7的那条Message。

这套机制是建立在offset是有序的。索引文件被映射到内存中，所以查找的速度还是很快的。

一句话，Kafka的Message存储采用了分区(partition)，分段(LogSegment)和稀疏索引这几个手段来达到了高效性。

Kafka 的设计时什么样的呢?

Kafka 将消息以 topic 为单位进行归纳

将向 Kafka topic 发布消息的程序成为 producers.

将预订 topics 并消费消息的程序成为 consumer.

Kafka 以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个 broker.

producers 通过网络将消息发送到 Kafka 集群，集群向消费者提供消息

数据传输的事物定义有哪三种?

数据传输的事务定义通常有以下三种级别：

（1）最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输

（2）最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.

（3）精确的一次（Exactly once）:不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的

Kafka 判断一个节点是否还活着有那两个条件?

（1）节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接

（2）如果节点是个 follower,他必须能及时的同步 leader 的写操作，延时不能太久

producer 是否直接将数据发送到 broker 的 leader(主节点)?

producer 直接将数据发送到 broker 的 leader(主节点)，不需要在多个节点进行分发，为了帮助 producer 做到这点，所有的 Kafka 节点都可以及时的告知:哪些节点是活动的，目标topic 目标分区的 leader 在哪。这样 producer 就可以直接将消息发送到目的地了

Kafa consumer 是否可以消费指定分区消息?

Kafaconsumer 消费消息时，向 broker 发出"fetch"请求去消费特定分区的消息，consumer 指定消息在日志中的偏移量（offset），就可以消费从这个位置开始的消息，customer 拥有了 offset 的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的

Kafka 消息是采用 Pull 模式，还是 Push 模式?

Kafka 最初考虑的问题是，customer 应该从 brokes 拉取消息还是 brokers 将消息推送到

consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的设计：producer 将消息推送到 broker，consumer 从 broker 拉取消息

一些消息系统比如 Scribe 和 ApacheFlume 采用了 push 模式，将消息推送到下游的 consumer。这样做有好处也有坏处：由 broker 决定消息推送的速率，对于不同消费速率的 consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息，但不幸的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时， consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式

Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略

Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，

直到新消息到 t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞知道新消息到达

(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发

Kafka 存储在硬盘上的消息格式是什么?

消息由一个固定长度的头部和可变长度的字节数组组成。头部包含了一个版本号和 CRC32

校验码。

·消息长度: 4 bytes (value: 1+4+n)

·版本号: 1 byte

·CRC 校验码: 4 bytes

·具体的消息: n bytes

Kafka 高效文件存储设计特点：

(1).Kafka 把 topic 中一个 parition 大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

(2).通过索引信息可以快速定位 message 和确定 response 的最大大小。

(3).通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。(4).通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小。

Kafka 与传统消息系统之间有三个关键区别

(1).Kafka 持久化日志，这些日志可以被重复读取和无限期保留

(2).Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

(3).Kafka 支持实时的流式处理

Kafka 创建 Topic 时如何将分区放置到不同的 Broker 中

·副本因子不能大于 Broker 的个数；

·第一个分区（编号为 0）的第一个副本放置位置是随机从 brokerList 选择的；

·其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果我们有 5 个

Broker，5 个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个

Broker上；第三个分区将会放在第一个Broker上；第四个分区将会放在第二个Broker 上，依次类推；

·剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

Kafka 新建的分区会在哪个目录下创建

在启动 Kafka 集群之前，我们需要配置好 log.dirs 参数，其值是 Kafka 数据的存放目录， 这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。

Zookeeper（周老师）

ZooKeeper 是什么?

ZooKeeper 是一个开源的分布式协调服务。它是一个为分布式应用提供一致性 服务的软件，分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、 负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和 分布式队列等功能。

ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性 能高效、功能稳定的系统提供给用户。

Zookeeper 保证了如下分布式一致性特性：

（1）顺序一致性

（2）原子性

（3）单一视图

（4）可靠性

（5）实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了 监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这 些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。

因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞 吐会下降。

有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每 个更新都有一个唯一的时间戳，这个时间戳称为 zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中 会带有这个zookeeper 最新的 zxid。

ZooKeeper 提供了什么?

- 文件系统
- 通知机制

Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统 不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存 放数据而目录节点不行。

Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构， 这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限 为1M。

Zookeeper 怎么保证主从节点的状态同步?

Zookeeper 的核心是原子广播机制，这个机制保证了各个 server 之间的同步。 实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模 式和广播模式。

恢复模式

当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出 来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状 态同步保证了 leader 和 server 具有相同的系统状态。

广播模式

一旦 leader 已经和多数的 follower 进行了状态同步后，它就可以开始广播消息 了，即进入广播状态。这时候当一个 server 加入 ZooKeeper 服务中，它会在 恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。

四种类型的数据节点 Znode

（1）PERSISTENT-持久节点
除非手动删除，否则节点一直存在于 Zookeeper 上

（2）EPHEMERAL-临时节点
临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点 都会被移除。

（3）PERSISTENT_SEQUENTIAL-持久顺序节点
基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维 护的自增整型数字。

（4）EPHEMERAL_SEQUENTIAL-临时顺序节点
基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的 自增整型数字。

Zookeeper Watcher 机制 – 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服 务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事 件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类 型做出业务上的改变。

工作机制：

（1）客户端注册 watcher

（2）服务端处理 watcher

（3）客户端回调 watcher

Watcher 特性总结：

（1）一次性

无论是服务端还是客户端，一旦一个 Watcher 被 触 发 ，Zookeeper 都会将其 从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务 端的压力都非常大。

（2）客户端串行执行 客户端 Watcher 回调的过程是一个串行同步的过程。

（3）轻量

3.1、Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的 具体内容。

3.2、客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了 标记。

（4）watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是 异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通 信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知 它所监视 znode发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节 点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。

（5）注册 watcher getData、exists、getChildren

（6）触发 watcher create、delete、setData

（7）当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件 触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通 常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未 创建的 znode的 exist watch，如果在客户端断开连接期间被创建了，并且随后 在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

客户端注册 Watcher 实现

（1）调用 getData()/getChildren()/exist()三个 API，传入 Watcher 对象

（2）标记请求 request，封装 Watcher 到 WatchRegistration

（3）封装成 Packet 对象，发服务端发送 request

（4）收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管 理

（5）请求返回，完成注册。

服务端处理 Watcher 实现

（1）服务端接收 Watcher 并存储
接收到客户端请求，处理请求判断是否需要注册 Watcher，需要的话将数据节 点的节点路径和 ServerCnxn（ServerCnxn 代表一个客户端和服务端的连接， 实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象）存储在 WatcherManager 的 WatchTable 和 watch2Paths 中去。

（2）Watcher 触发
以服务端接收到 setData() 事务请求触发 NodeDataChanged 事件为例：

2.1 封装 WatchedEvent 将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点 路径封装成一个 WatchedEvent 对象

2.2 查询 Watcher 从 WatchTable 中根据节点路径查找 Watcher 2.3 没找到；说明没有客户端在该数据节点上注册过 Watcher 2.4 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从 这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了）

（3）调用 process 方法来触发 Watcher 这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件 通知。

客户端回调 Watcher

客户端 SendThread 线程接收事件通知，交由 EventThread 线程回调 Watcher。

客户端的 Watcher 机制同样是一次性的，一旦被触发后，该 Watcher 就失效 了。

ACL 权限控制机制

UGO（User/Group/Others）

目前在 Linux/Unix 文件系统中使用，也是使用最广泛的权限控制方式。是一种 粗粒度的文件系统权限控制模式。

ACL（Access Control List）访问控制列表

包括三个方面：

权限模式（Scheme）

（1）IP：从 IP 地址粒度进行权限控制

（2）Digest：最常用，用类似于 username:password 的权限标识来进行权限 配置，便于区分不同应用来进行权限控制

（3）World：最开放的权限控制方式，是一种特殊的 digest 模式，只有一个 权限标识“world:anyone”

（4）Super：超级用户

授权对象

授权对象指的是权限赋予的用户或一个指定实体，例如 IP 地址或是机器灯。

权限 Permission

（1）CREATE：数据节点创建权限，允许授权对象在该 Znode 下创建子节点

（2）DELETE：子节点删除权限，允许授权对象删除该数据节点的子节点

（3）READ：数据节点的读取权限，允许授权对象访问该数据节点并读取其数 据内容或子节点列表等

（4）WRITE：数据节点更新权限，允许授权对象对该数据节点进行更新操作

（5）ADMIN：数据节点管理权限，允许授权对象对该数据节点进行 ACL 相关 设置操作

Chroot 特性

3.2.0 版本后，添加了 Chroot 特性，该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了 Chroot，那么该客户端对服务器的任何操作，都将会被限制在其自己的命名空间下。

通过设置 Chroot，能够将一个客户端应用于 Zookeeper 服务端的一颗子树相对应，在那些多个应用公用一个 Zookeeper 进群的场景下，对实现不同应用间的相互隔离非常有帮助。

会话管理

分桶策略：将类似的会话放在同一区块中进行管理，以便于 Zookeeper 对会话进行不同区块的隔离处理以及同一区块的统一处理。分配原则：每个会话的“下次超时时间点”（ExpirationTime）

计算公式：

ExpirationTime_ = currentTime + sessionTimeout

ExpirationTime = (ExpirationTime_ / ExpirationInrerval + 1) *

ExpirationInterval , ExpirationInterval 是指 Zookeeper 会话超时检查时间间隔，默认 tickTime

服务器角色

Leader

（1） 事务请求的唯一调度和处理者，保证集群事务处理的顺序性

（2） 集群内部各服务的调度者

Follower

（1） 处理客户端的非事务请求，转发事务请求给 Leader 服务器

（2） 参与事务请求 Proposal 的投票

（3） 参与 Leader 选举投票

Observer

（1）3.0 版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础

上提升集群的非事务处理能力

（2） 处理客户端的非事务请求，转发事务请求给 Leader 服务器

（3） 不参与任何形式的投票

Zookeeper 下 Server 工作状态

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、 OBSERVING。

（1） LOOKING：寻 找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有 Leader，因此需要进入 Leader 选举状态。

（2） FOLLOWING：跟随者状态。表明当前服务器角色是 Follower。（3）LEADING：领导者状态。表明当前服务器角色是 Leader。

（4）OBSERVING：观察者状态。表明当前服务器角色是 Observer。

数据同步

整个集群完成 Leader 选举之后，Learner（Follower 和 Observer 的统称）回向Leader 服务器进行注册。当 Learner 服务器想 Leader 服务器完成注册后，进入数据同步环节。数据同步流程：（均以消息传递的方式进行）

Learner 向 Learder 注册

数据同步同步确认

Zookeeper 的数据同步通常分为四类：

（1） 直接差异化同步（DIFF 同步）

（2） 先回滚再差异化同步（TRUNC+DIFF 同步）

（3） 仅回滚同步（TRUNC 同步）（4）全量同步（SNAP 同步）

在进行数据同步前，Leader 服务器会完成数据同步初始化： peerLastZxid：

· 从 learner 服务器注册时发送的 ACKEPOCH 消息中提取 lastZxid（该Learner 服务器最后处理的 ZXID）

minCommittedLog：

· Leader 服务器 Proposal 缓存队列 committedLog 中最小

ZXIDmaxCommittedLog：

· Leader 服务器 Proposal 缓存队列 committedLog 中最大 ZXID直接差异化同步（DIFF 同步）

· 场景：peerLastZxid 介于 minCommittedLog 和 maxCommittedLog之间先回滚再差异化同步（TRUNC+DIFF 同步）

· 场景：当新的 Leader 服务器发现某个 Learner 服务器包含了一条自己没有的事务记录，那么就需要让该 Learner 服务器进行事务回滚–回滚到 Leader服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID仅回滚同步（TRUNC 同步）

· 场景：peerLastZxid 大于 maxCommittedLog 全量同步（SNAP 同步）

· 场景一：peerLastZxid 小于 minCommittedLog

· 场景二：Leader 服务器上没有 Proposal 缓存队列且 peerLastZxid 不等于 lastProcessZxid

zookeeper 是如何保证事务的顺序一致性的?

zookeeper 采用了全局递增的事务 Id 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是

epoch（ 时期; 纪元; 世; 新时代）用来标识 leader 周期，如果有新的 leader 产生出来，epoch会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

分布式集群中为什么会有 Master主节点?

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。

zk 节点宕机如何处理?

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；

如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在

ZK节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。

所以

3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)

2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1) 19. zookeeper 负载均衡和 nginx 负载均衡区别

zk 的负载均衡是可以调控，nginx 只是能调权重，其他需要可控的都需要自己写插件；但是 nginx 的吞吐量比 zk 大很多，应该说按业务选择用哪种方式。

Zookeeper 有哪几种几种部署模式?

Zookeeper 有三种部署模式：

1. 单机部署：一台集群上运行；
2. 集群部署：多台集群运行；
3. 伪集群部署：一台集群启动多个 Zookeeper 实例运行。

集群最少要几台机器，集群规则是怎样的？集群中有 3 台服务器，其中一个节点宕机，这个时候 Zookeeper 还可以使用吗?

集群规则为 2N+1 台，N>0，即 3 台。可以继续使用，单数服务器只要没超过一半的服务器宕机就可以继续使用。

集群支持动态添加机器吗?

其实就是水平扩容了，Zookeeper 在这方面不太好。两种方式：

全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。

逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5 版本开始支持动态扩容。

Zookeeper 对节点的 watch 监听通知是永久的吗？为什么不是永久的?

不是。官方声明：一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况

下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。

一般是客户端执行 getData(“/节点 A”,true)，如果节点 A 发生了变更或删除，客户端会得到它的 watch 事件，但是在之后节点 A 又发生了变更，而客户端又没有设置 watch 事件，就不再给客户端发送。在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。

Zookeeper 的 java 客户端都有哪些?

java 客户端：zk 自带的 zkclient 及 Apache 开源的 Curator。

chubby 是什么，和 zookeeper 比你怎么看?

chubby 是 google 的，完全实现 paxos 算法，不开源。zookeeper 是 chubby的开源实现，使用 zab 协议，paxos 算法的变种。

说几个 zookeeper 常用的命令。

常用命令：ls get set create delete 等。

ZAB 和 Paxos 算法的联系与区别?

相同点：

（1） 两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行

（2） Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交

（3） ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的

Leader周期，Paxos 中名字为 Ballot 不同点：

ZAB 用来构建高可用的分布式数据主备系统（Zookeeper），Paxos 是用来构建分布式一致性状态机系统。

Zookeeper 的典型应用场景

Zookeeper 是一个典型的发布/订阅模式的分布式数据管理与协调框架，开发人员可以使用它来进行分布式数据的发布和订阅。

通过对 Zookeeper 中丰富的数据节点进行交叉使用，配合 Watcher 事件通知机制，可以非常方便的构建一系列分布式应用中年都会涉及的核心功能，如：

（1） 数据发布/订阅

（2） 负载均衡

（3） 命名服务

（4） 分布式协调/通知

（5） 集群管理

（6） Master 选举

（7） 分布式锁

（8） 分布式队列数据发布/订阅介绍数据发布/订阅系统，即所谓的配置中心，顾名思义就是发布者发布数据供订阅者进行数据订阅。

目的动态获取数据（配置信息）实现数据（配置信息）的集中式管理和数据的动态更新设计模式

Push 模式

Pull 模式

数据（配置信息）特性

（1） 数据量通常比较小

（2） 数据内容在运行时会发生动态更新

（3） 集群中各机器共享，配置一致

如：机器列表信息、运行时开关配置、数据库配置信息等基于 Zookeeper 的实现方式

· 数据存储：将数据（配置信息）存储到 Zookeeper 上的一个数据节点

· 数据获取：应用在启动初始化节点从 Zookeeper 数据节点读取数据，并在该节点上注册一个数据变更 Watcher

· 数据变更：当变更数据时，更新 Zookeeper 对应节点数据，Zookeeper会将数据变更通知发到各客户端，客户端接到通知后重新读取变更后的数据即可。

负载均衡

zk 的命名服务命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。

对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

zk 的命名服务（文件系统）命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

zk 的配置管理（文件系统、通知机制）程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 watcher 通知给各个客户端，从而更改配置。

Zookeeper 集群管理（文件系统、通知机制）

所谓集群管理无在乎两点：是否有机器退出和加入、选举 master。

对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。

新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount 又有

了，对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。

Zookeeper 分布式锁（文件系统、通知机制）

有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。

对于第二类， /distribute_lock 已经预先存在，所有客户端在它下面创建临时顺

序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

Zookeeper 队列管理（文件系统、通知机制）两种类型的队列：

（1） 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

（2） 队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点，创建成功时 Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下

Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

Zookeeper 都有哪些功能?

1. 集群管理：监控节点存活状态、运行请求等；
2. 主节点选举：主节点挂掉了之后可以从备用的节点开始新一轮选主，主节点选举说的就是这个选举的过程，使用 Zookeeper 可以协助完成这个过程；
3. 分布式锁：Zookeeper 提供两种锁：独占锁、共享锁。独占锁即一次只能有一个线程使用资源，共享锁是读锁共享，读写互斥，即可以有多线

线程同时读同一个资源，如果要使用写锁也只能有一个线程使用。

Zookeeper 可以对分布式锁进行控制。

4. 命名服务：在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。

说一下 Zookeeper 的通知机制?

client 端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，这些 client 会收到 zk 的通知，然后 client 可以根据 znode 变化来做出业务上的改变等。

Zookeeper 和 Dubbo 的关系?

Zookeeper的作用：

zookeeper用来注册服务和进行负载均衡，哪一个服务由哪一个机器来提供必需让调用者知道，简单来说就是ip地址和服务名称的对应关系。当然也可以通过硬编码的方式把这种对应关系在调用方业务代码中实现，但是如果提供服务的机器

挂掉调用者无法知晓，如果不更改代码会继续请求挂掉的机器提供服务。

zookeeper通过心跳机制可以检测挂掉的机器并将挂掉机器的ip和服务对应关系从列表中删除。至于支持高并发，简单来说就是横向扩展，在不更改代码的情况通过添加机器来提高运算能力。通过添加新的机器向zookeeper注册服务，服务的提供者多了能服务的客户就多了。

dubbo：

是管理中间层的工具，在业务层到数据仓库间有非常多服务的接入和服务提供者需要调度，dubbo提供一个框架解决这个问题。

注意这里的dubbo只是一个框架，至于你架子上放什么是完全取决于你的，就像一个汽车骨架，你需要配你的轮子引擎。这个框架中要完成调度必须要有一个分布式的注册中心，储存所有服务的元数据，你可以用zk，也可以用别的，只是大家都用zk。

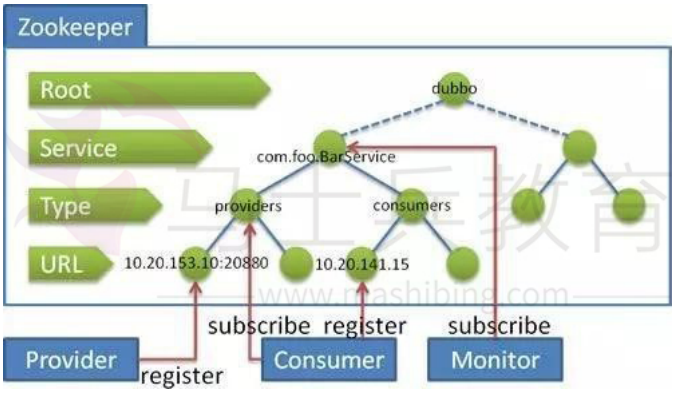
zookeeper和dubbo的关系：

Dubbo 的将注册中心进行抽象，它可以外接不同的存储媒介给注册中心提供服务，有 ZooKeeper，Memcached，Redis 等。

引入了 ZooKeeper 作为存储媒介，也就把 ZooKeeper 的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时 候就需要分流，负载均衡就是为了分流而存在的，一个 ZooKeeper 群配合相应的

Web 应用就可以很容易达到负载均衡；资源同步，单单有负载均衡还不 够，节点之间的数据和资源需要同步，ZooKeeper 集群就天然具备有这样的功能；命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动 的时

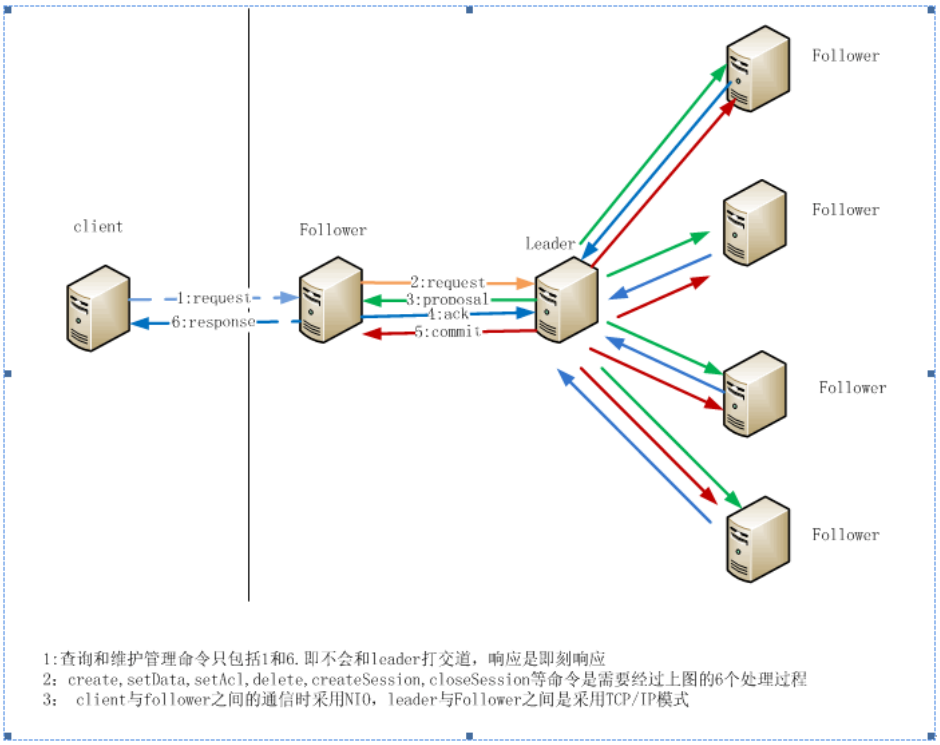
候，向 ZooKeeper 上的指定节点 /dubbo/${serviceName}/providers 目录下写入自己的 URL 地址，这个操作就完成了服务的发布。 其他特性还有 Mast 选举，分布式锁等。



zookeeper集群之间如何通讯

zookeeper系列之通信模型

本文的主题就是讲解Zookeeper通信模型，本节将通过一个概要图来说明Zookeeper的通信模型。

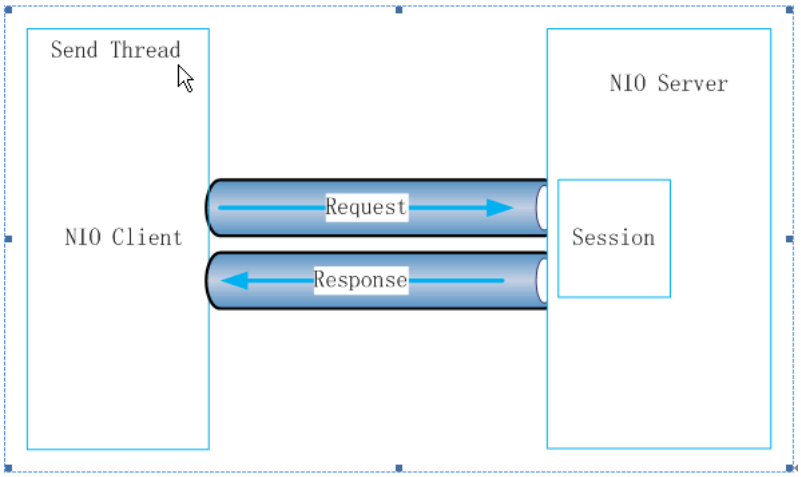


Zookeeper的通信架构

在Zookeeper整个系统中，有3中角色的服务，client、Follower、leader。其中client负责发起应用的请求，Follower接受client发起的请求，参与事务的确认过程，在leader crash后的leader选择。而leader主要承担事务的协调，当然leader也可以承担接收客户请求的功能，为了方便描述，后面的描述都是client与Follower之间的通信，如果Zookeeper的配置支持leader接收client的请求，client与leader的通信跟client与Follower的通信模式完全一样。Follower与leader之间的角色可能在某一时刻进行转换。一个Follower在leader crash掉以后可能被集群（Quorum）的Follower选举为leader。而一个leader在crash后，再次加入集群（Quorum）将作为Follower角色存在。在一个集群（Quorum）中,除了在选举leader的过程中没有Follower和leader的区分外，其他任何时刻都只有1个leader和多个Follower。Client、Follower和leader之间的通信架构如下：

Client与Follower之间

为了使客户端具有较高的吞吐量，Client与Follower之间采用NIO的通信方式。当client需要与Zookeeper service打交道时，首先读取配置文件确定集群内的所有server列表，按照一定的load balance算法选取一个Follower作为一个通信目标。这样client和Follower之间就有了一条由NIO模式构成的通信通道。这条通道会一直保持到client关闭session或者因为client或Follower任一方因某种原因异常中断通信连接。正常情况下, client与Follower在没有请求发起的时候都有心跳检测。



Follower与leader之间

Follower与leader之间的通信主要是因为Follower接收到像（create, delete, setData, setACL, createSession, closeSession, sync）这样一些需要让leader来协调最终结果的命令，将会导致Follower与leader之间产生通信。由于leader与Follower之间的关系式一对多的关系，非常适合client/server模式，因此他们之间是采用c/s模式，由leader创建一个socket server，监听各Follower的协调请求。

集群在选择leader过程中

由于在选择leader过程中没有leader，在集群中的任何一个成员都需要与其他所有成员进行通信，当集群的成员变得很大时，这个通信量是很大的。选择leader的过程发生在Zookeeper系统刚刚启动或者是leader失去联系后，选择leader过程中将不能处理用户的请求，为了提高系统的可用性，一定要尽量减少这个过程的时间。选择哪种方式让他们可用快速得到选择结果呢？Zookeeper在这个过程中采用了策略模式，可用动态插入选择leader的算法。系统默认提供了3种选择算法，AuthFastLeaderElection，FastLeaderElection，LeaderElection。其中AuthFastLeaderElection和LeaderElection采用UDP模式进行通信，而FastLeaderElection仍然采用tcp/ip模式。在Zookeeper新的版本中，新增了一个learner角色，减少选择leader的参与人数。使得选择过程更快。一般说来Zookeeper leader的选择过程都非常快，通常<200ms。

Zookeeper的通信流程

要详细了解Zookeeper的通信流程，我们首先得了解Zookeeper提供哪些客户端的接口，我们按照具有相同的通信流程的接口进行分组：

Zookeeper系统管理命令

Zookeeper的系统管理接口是指用来查看Zookeeper运行状态的一些命令，他们都是具有4字母构成的命令格式。主要包括：

1. ruok:发送此命令可以测试zookeeper是否运行正常。
2. dump：dump server端所有存活session的Ephemeral（临时）node信息。
3. stat：获取连接server的服务器端的状态及连接该server的所有客服端的状态信息。
4. reqs: 获取当前客户端已经提交但还未返回的请求。
5. stmk：开启或关闭Zookeeper的trace level.
6. gtmk：获取当前Zookeeper的trace level是否开启。
7. envi: 获取Zookeeper的java相关的环境变量。
8. srst：重置server端的统计状态

当用户发送这些命令的到server时，由于这些请求只与连接的server相关，没有业务处理逻辑，非常简单。Zookeeper对这些命令采用最快的效率进行处理。这些命令发送到server端只占用一个4字节的int类型来表示不同命令，没有采用字符串处理。当服务器端接收到这些命令，立刻返回结果。

Session创建

任何客户端的业务请求都是基于session存在的前提下。Session是维持client与Follower之间的一条通信通道，并维持他们之间从创建开始后的所有状态。当启动一个Zookeeper client的时候，首先按照一定的算法查找出follower, 然后与Follower建立起NIO连接。当连接建立好后，发送create session的命令，让server端为该连接创建一个维护该连接状态的对象session。当server收到create session命令，先从本地的session列表中查找看是否已经存在有相同sessionId，则关闭原session重新创建新的session。创建session的过程将需要发送到Leader，再由leader通知其他follower，大部分Follower都将此操作记录到本地日志再通知leader后，leader发送commit命令给所有Follower，连接客户端的Follower返回创建成功的session响应。Leader与Follower之间的协调过程将在后面的做详细讲解。当客户端成功创建好session后，其他的业务命令就可以正常处理了。

Zookeeper查询命令

Zookeeper查询命令主要用来查询服务器端的数据，不会更改服务器端的数据。所有的查询命令都可以即刻从client连接的server立即返回，不需要leader进行协调，因此查询命令得到的数据有可能是过期数据。但由于任何数据的修改，leader都会将更改的结果发布给所有的Follower，因此一般说来，Follower的数据是可以得到及时的更新。这些查询命令包括以下这些命令：

1. exists:判断指定path的node是否存在，如果存在则返回true，否则返回false.
2. *getData:**从指定**path**获取该**node**的数据*
3. *getACL:**获取指定**path**的**ACL**。*
4. *getChildren:**获取指定**path**的**node**的所有孩子结点。*

所有的查询命令都可以指定watcher，通过它来跟踪指定path的数据变化。一旦指定的数据发生变化（create,delete,modified,children_changed），服务器将会发送命令来回调注册的watcher. Watcher详细的讲解将在Zookeeper的Watcher中单独讲解。

Zookeeper修改命令

Zookeeper修改命令主要是用来修改节点数据或结构，或者权限信息。任何修改命令都需要提交到leader进行协调，协调完成后才返回。修改命令主要包括：

1. *createSession：请求**server**创建一个**session*
2. *create：创建一个节点*
3. *delete：删除一个节点*
4. *setData：修改一个节点的数据*
5. *setACL：修改一个节点的**ACL*
6. *closeSession：请求**server**关闭**session*

我们根据前面的通信图知道，任何修改命令都需要leader协调。 在leader的协调过程中，需要3次leader与Follower之间的来回请求响应。并且在此过程中还会涉及事务日志的记录，更糟糕的情况是还有take snapshot的操作。因此此过程可能比较耗时。但Zookeeper的通信中最大特点是异步的，如果请求是连续不断的，Zookeeper的处理是集中处理逻辑，然后批量发送，批量的大小也是有控制的。如果请求量不大，则即刻发送。这样当负载很大时也能保证很大的吞吐量，时效性也在一定程度上进行了保证。

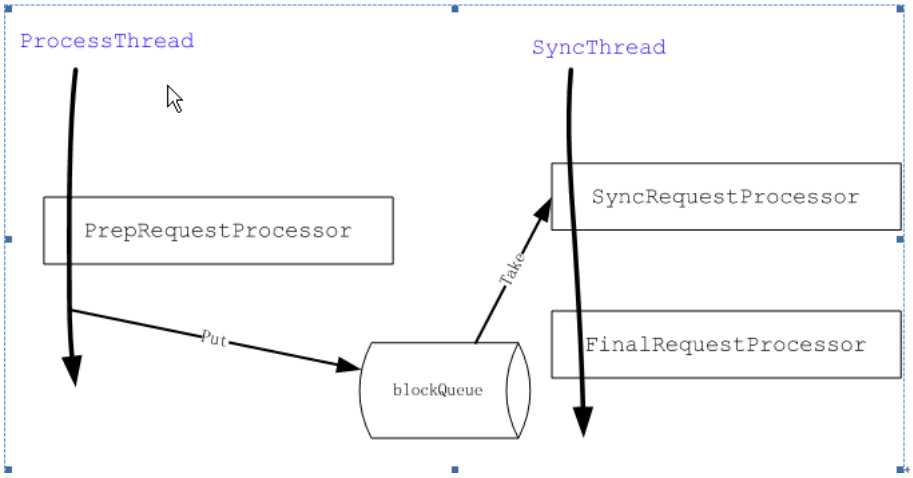
zookeeper server端的业务处理-processor链

Zookeeper通过链式的processor来处理业务请求，每个processor负责处理特定的功能。不同的Zookeeper角色的服务器processor链是不一样的，以下分别介绍standalone Zookeeper server, leader和Follower不同的processor链。

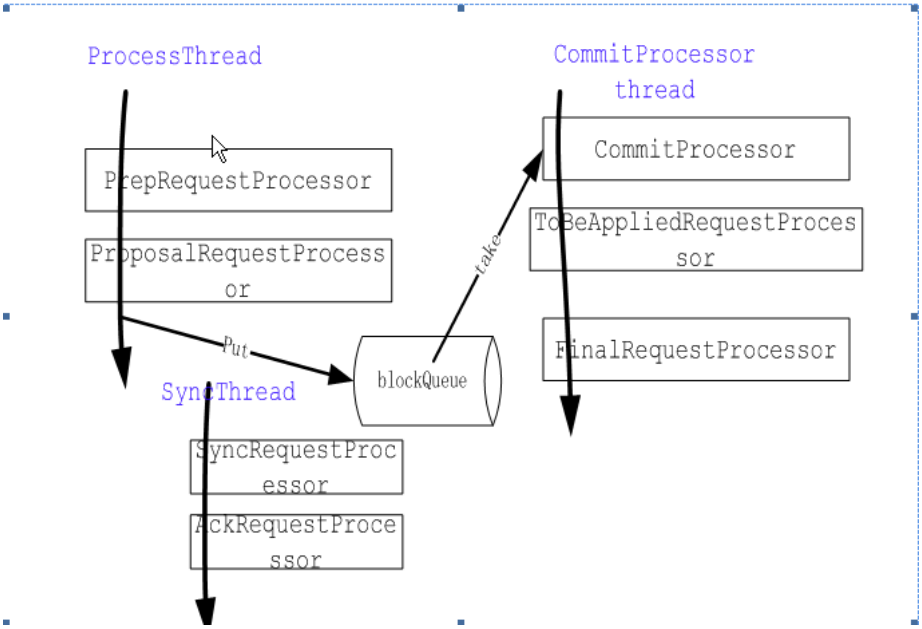
Zookeeper中的processor

1. AckRequestProcessor:当leader从向Follower发送proposal后，Follower将发送一个Ack响应，leader收到Ack响应后，将会调用这个Processor进行处理。它主要负责检查请求是否已经达到了多数Follower的确认，如果满足条件，则提交commitProcessor进行commit处理
2. CommitProcessor：从commited队列中处理已经由leader协调好并commit的请求或者从请求队列中取出那些无需leader协调的请求进行下一步处理。
3. FinalRequestProcessor：任何请求的处理都需要经过这个processor，这是请求处理的最后一个Processor，主要负责根据不同的请求包装不同的类型的响应包。当然Follower与leader之间协调后的请求由于没有client连接，将不需要发送响应（代码体现在if (request.cnxn == null) {return;}）。
4. FollowerRequestProcessor：Follower processor链上的第一个，主要负责将修改请求和同步请求发往leader进行协调。
5. PrepRequestProcessor：在leader和standalone server上作为第一Processor，主要作用对于所有的修改命令生成changelog。
6. ProposalRequestProcessor：leader用来将请求包装为proposal向Follower请求确认。
7. SendAckRequestProcessor：Follower用来向leader发送Ack响应的处理。
8. SyncRequestProcessor：负责将已经commit的事务写到事务日志以及take snapshot.
9. ToBeAppliedRequestProcessor:负责将tobeApplied队列的中request转移到下一个请求进行处理。

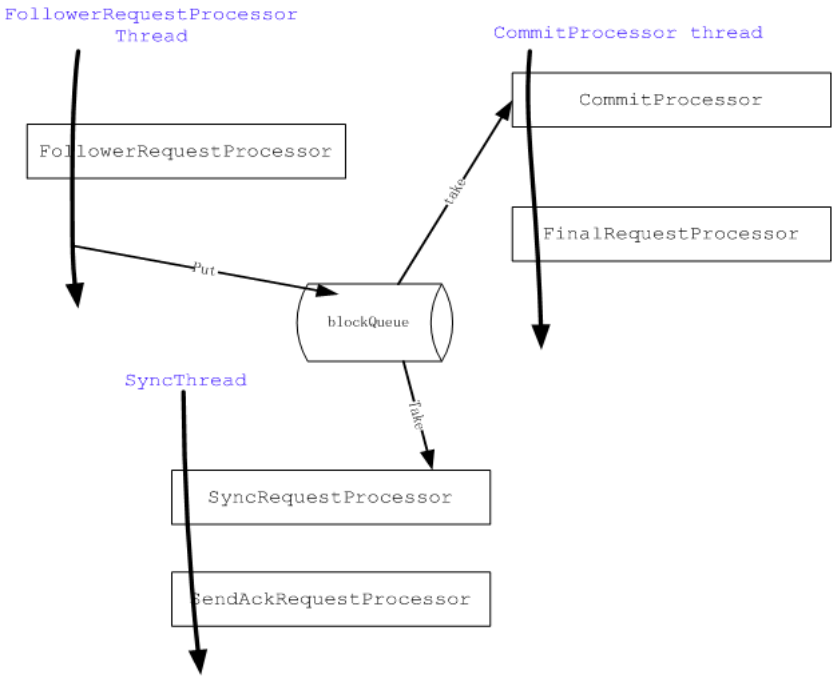
Standalone zookeeper processor链



Leader processor链



Follower processor链



Zookeeper面试题

ZooKeeper 是什么?

ZooKeeper 是一个***分布式***的，开放源码的分布式***应用程序协调服务***，是 Google 的 Chubby 一个开源的实现，它是***集群的管理者***，***监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作***。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

客户端的***读请求***可以被集群中的***任意一台机器处理***，如果读请求在节点上注册了监听器，这个监听器也是由所

连接的 zookeeper 机器来处理。对于***写请求***，这些请求会同***时发给其他zookeeper机器并且达成一致后，请求才会返回成功***。因此，随着***zookeeper*的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降***。有序性是 zookeeper 中非常重要的一个特性，所有的***更新都是全局有序的***，每个更新都有一个***唯一的时间戳***，这个时间戳称为***zxid*（Zookeeper Transaction Id）*。而*读请求只会相对于更新有序***，也就是读请求的返回结果中会带有这个***zookeeper*最新的zxid*。

ZooKeeper 提供了什么?

1、 ***文件系统***

2、 ***通知机制***

Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点***都可以设置关联的数据***，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper 为了保证高吞吐和低延

迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper***不能用于存放大量的数据***，每个节点的存放数据上限为***1M*。

四种类型的 znode

1、 ***PERSISTENT*-*持久化目录节点***

客户端与 zookeeper 断开连接后，该节点依旧存在

2、 ***PERSISTENT_SEQUENTIAL*-*持久化顺序编号目录节点***

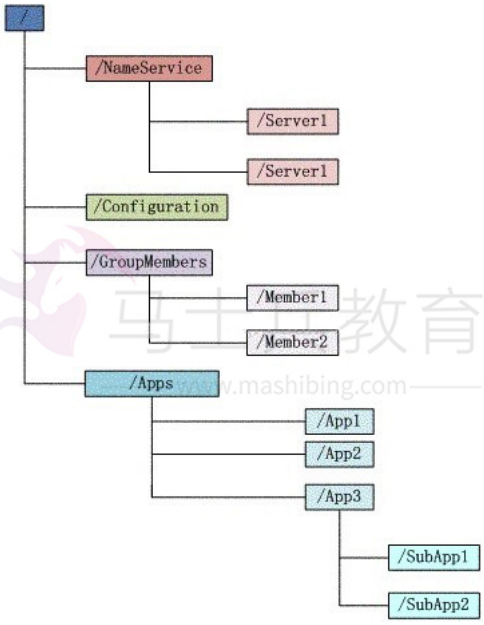
客户端与 zookeeper 断开连接后，该节点依旧存在，只是 Zookeeper 给该节点名称进行顺序编号

3、 ***EPHEMERAL*-*临时目录节点***

客户端与 zookeeper 断开连接后，该节点被删除

4、 ***EPHEMERAL_SEQUENTIAL*-*临时顺序编号目录节点***

客户端与 zookeeper 断开连接后，该节点被删除，只是 Zookeeper 给该节点名称进行顺序编号



Zookeeper 通知机制

client 端会对某个 znode 建立一个***watcher*事件***，当该 znode 发生变化时，这些 client 会收到 zk 的通知，然后 client 可以根据 znode 变化来做出业务上的改变等。

Zookeeper 做了什么?

1、命名服务

2、配置管理

3、集群管理

4、分布式锁

5、队列管理

zk 的命名服务（文件系统）

命名服务是指通过指定的名字来***获取资源***或者***服务的地址***，利用 zk 创建一个全局的路径，即是***唯一***的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

zk 的配置管理（文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的***znode***下，当有配置发生改变时，也就是

znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用***watcher***通知给各个客户端，从而更改配置。

Zookeeper 集群管理（文件系统、通知机制）

所谓集群管理无在乎两点：***是否有机器退出和加入、选举master*。**

对于第一点，所有机器约定在父目录下***创建临时目录节点***，然后监听父目录节点的子节点变化消息。一旦有机

器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，***所有其他机器都收到通知：某个兄弟目录被删除***，于是，所有人都知道：它上船了。

新机器加入也是类似，***所有机器收到通知：新兄弟目录加入***，highcount 又有了，对于第二点，我们稍微改变一下，***所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为master就好*。**

Zookeeper 分布式锁（文件系统、通知机制）

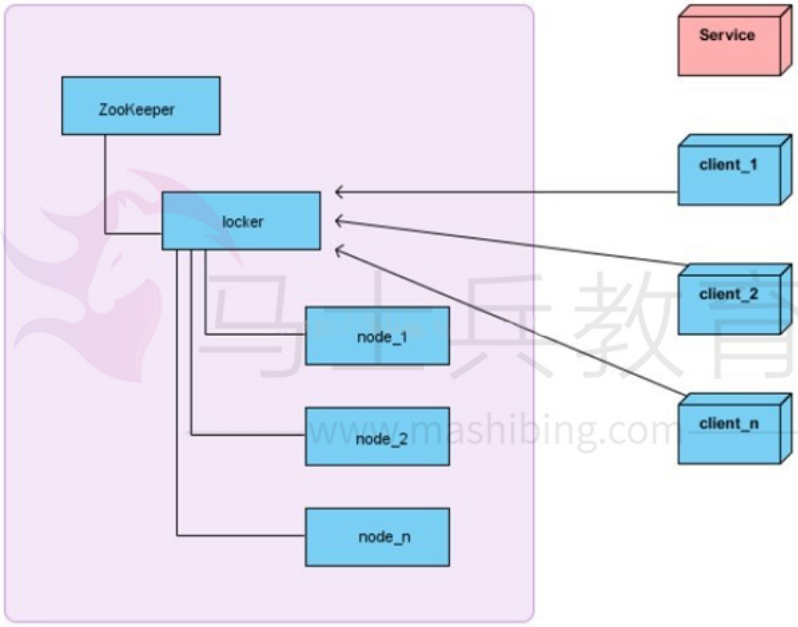
有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是***保持独占***，另一个是***控制时序*。**

对于第一类，我们将 zookeeper 上的一个***znode*看作是一把锁***，通过 createznode 的方式来实现。所有客户

端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。

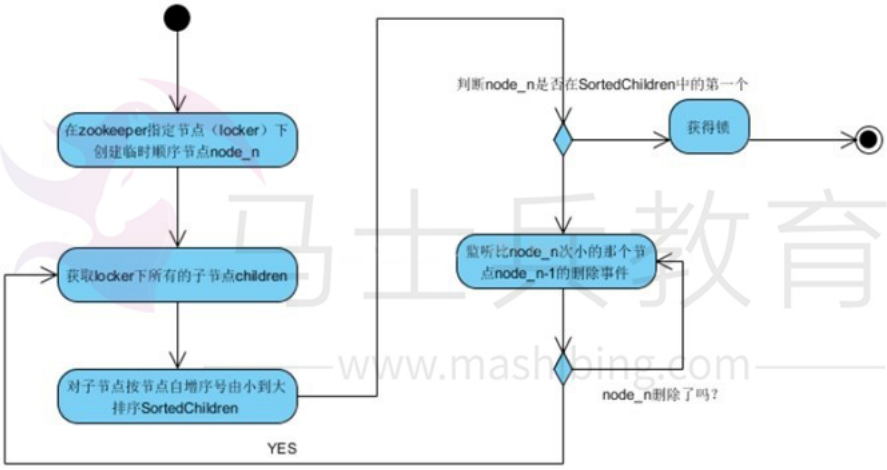
对于第二类， /distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，***编号最小的获得锁***，用完删除，依次方便。

获取分布式锁的流程



在获取分布式锁的时候在 locker 节点下创建临时顺序节点，释放锁的时候删除该临时节点。客户端调用 createNode 方法在 locker 下创建临时顺序节点，

然后调用 getChildren("locker")来获取 locker 下面的所有子节点，注意此时不用设置任何 Watcher。客户端获取到所有的子节点 path 之后，如果发现自己创建的节点在所有创建的子节点序号最小，那么就认为该客户端获取到了锁。如果发现自己创建的节点并非 locker 所有子节点中最小的，说明自己还没有获取到锁，此时客户端需要找到***比自己小的那个节点***，然后对其调用***exist()***方法，同时对其注册事件监听器。之后，让这个被关注的节点删除，则客户端的 Watcher 会收到相应通知，此时再次判断自己创建的节点是否是 locker 子节点中序号最小的，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听。当前这个过程中还需要许多的逻辑判断。



代码的实现主要是基于互斥锁，获取分布式锁的重点逻辑在于***BaseDistributedLock***，实现了基于Zookeeper 实现分布式锁的细节。

Zookeeper 队列管理（文件系统、通知机制）

两种类型的队列：

1、同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。

2、队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建

PERSISTENT_SEQUENTIAL节点，创建成功时***Watcher***通知等待的队列，队列删除***序列号最小的节点***用以消费。此场景下 Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，

SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以***不必担心队列消息的丢失问题***。

Zookeeper 数据复制

Zookeeper 作为一个集群提供一致的数据服务，自然，它要在***所有机器间***做数据复制。数据复制的好处：

1、容错：一个节点出错，不致于让整个系统停止工作，别的节点可以接管它的工作；

2、提高系统的扩展能力 ： 把负载分布到多个节点上，或者增加节点来提高系统的负载能力；

3、提高性能：让***客户端本地访问就近的节点，提高用户访问速度***。

从客户端读写访问的透明度来看，数据复制集群系统分下面两种：

1、 ***写主***(WriteMaster) ：对数据的***修改提交给指定的节点***。读无此限制，可以读取任何一个节点。这种情况下客户端需要对读与写进行区别，俗称***读写分离***；

2、 ***写任意***(Write Any)：对数据的***修改可提交给任意的节点***，跟读一样。这种情况下，客户端对集群节点的角色与变化透明。

对 zookeeper 来说，它采用的方式是***写任意***。通过增加机器，它的读吞吐能力和响应能力扩展性非常好，而写，随着机器的增多吞吐能力肯定下降（这也是它建立 observer 的原因），而响应能力则取决于具体实现方式，是***延迟复制保持最终一致性***，还是***立即复制快速响应***。

Zookeeper 工作原理

Zookeeper 的核心是***原子广播***，这个机制保证了***各个Server之间的同步***。实现这个机制的协议叫做***Zab*协议***。Zab 协议有两种模式，它们分别是***恢复模式（选主）***和***广播模式（同步）***。当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 Server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 Server 具有相同的系统状态。

zookeeper 是如何保证事务的顺序一致性的?

zookeeper 采用了***递增的事务Id***来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是 epoch（时期; 纪元; 世; 新时代）用来标识 leader 是否发生改变，如果有新的 leader 产生出来，epoch 会自增，***低32位用来递增计数***。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

Zookeeper 下 Server 工作状态

每个 Server 在工作过程中有三种状态：

LOOKING：当前 Server***不知道leader是谁***，正在搜寻

LEADING：当前 Server 即为选举出来的 leader

FOLLOWING：leader 已经选举出来，当前 Server 与之同步

zookeeper 是如何选取主 leader 的?

当 leader 崩溃或者 leader 失去大多数的 follower，这时 zk 进入恢复模式，恢复模式需要重新选举出一个新的 leader，让所有的 Server 都恢复到一个正确的状态。Zk 的选举算法有两种：一种是基于 basic paxos 实现的，另外一种是基于 fast paxos 算法实现的。系统默认的选举算法为***fast paxos***。

1、Zookeeper 选主流程(basic paxos)

（1） 选举线程由当前 Server 发起选举的线程担任，其主要功能是对投票结果进行统计，并选出推荐的Server；

（2） 选举线程首先向所有 Server 发起一次询问(包括自己)；

（3） 选举线程收到回复后，验证是否是自己发起的询问(验证 zxid 是否一致)，然后获取对方的 id(myid)，并存储到当前询问对象列表中，最后获取对方提议的 leader 相关信息(id,zxid)，并将这些信息存储到当次选举的投票记录表中；

（4） 收到所有 Server 回复以后，就计算出 zxid 最大的那个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server；

（5） 线程将当前 zxid 最大的 Server 设置为当前 Server 要推荐的 Leader，如果此时获胜的 Server 获得 n/2 + 1 的 Server 票数，设置当前推荐的 leader 为获胜的 Server，将根据获胜的 Server 相关信息设置自己的状态，否则，继续这个过程，直到 leader 被选举出来。 通过流程分析我们可以得出：要使 Leader 获得多数Server 的支持，则 Server 总数必须是奇数 2n+1，且存活的 Server 的数目不得少于 n+1. 每个 Server 启动后都会重复以上流程。在恢复模式下，如果是刚从崩溃状态恢复的或者刚启动的 server 还会从磁盘快照中恢复数据和会话信息，zk 会记录事务日志并定期进行快照，方便在恢复时进行状态恢复。

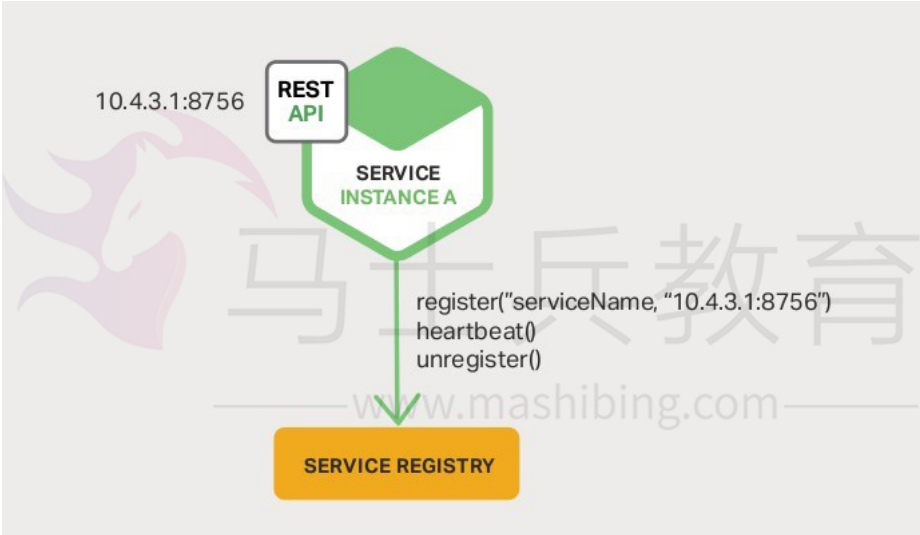
微服务

服务注册发现

服务注册就是维护一个登记簿，它管理系统内所有的服务地址。当新的服务启动后，它会向登记簿交待自己的地址信息。服务的依赖方直接向登记簿要Service Provider地址就行了。当下用于服务注册的工具非常多 ZooKeeper，Consul，Etcd, 还有 Netflix 家的 eureka 等。服务注册有两种形式：客户端注册和第三方注册。

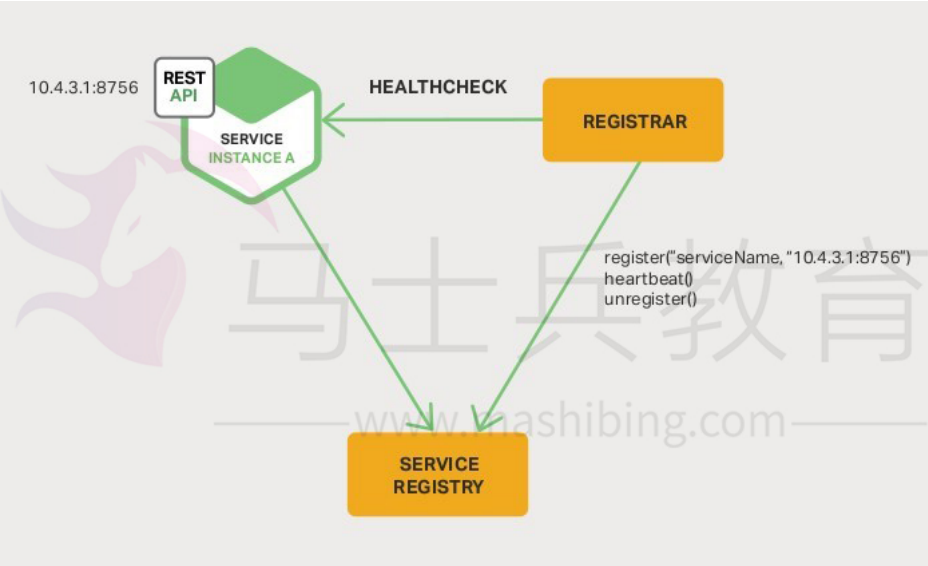
客户端注册（zookeeper）

客户端注册是服务自身要负责注册与注销的工作。当服务启动后向注册中心注册自身，当服务下线时注销自己。期间还需要和注册中心保持心跳。心跳不一定要客户端来做，也可以由注册中心负责（这个过程叫探活）。这种方式的缺点是注册工作与服务耦合在一起，不同语言都要实现一套注册逻辑。



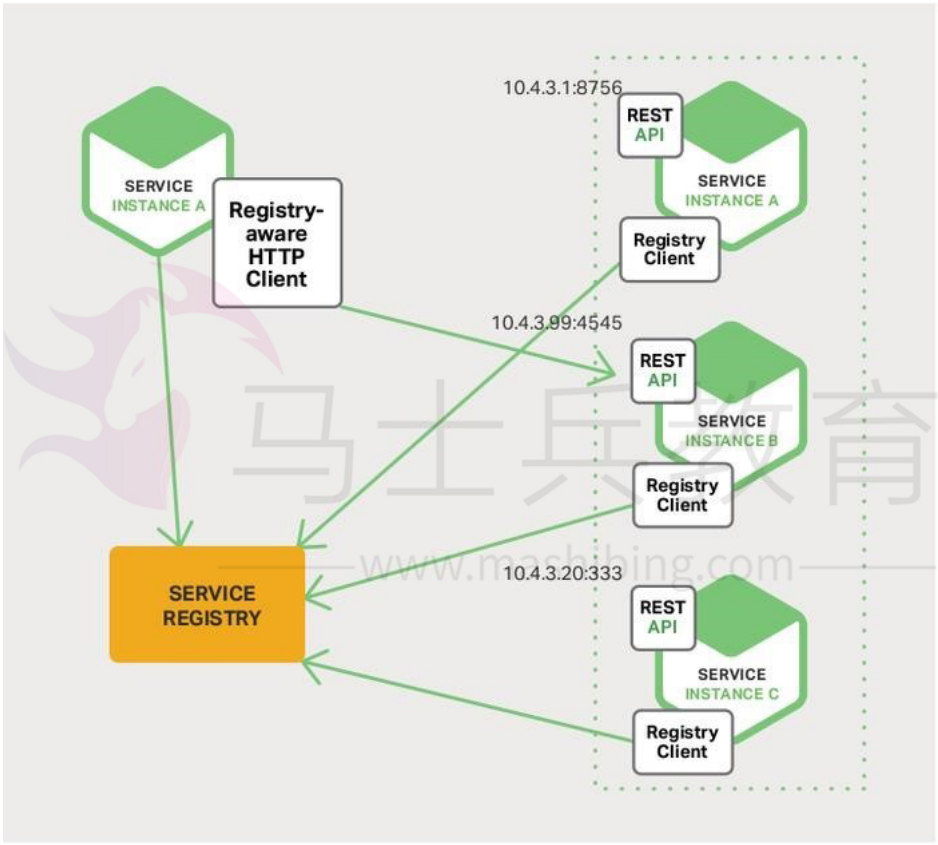
第三方注册（独立的服务 Registrar）

第三方注册由一个独立的服务Registrar负责注册与注销。当服务启动后以某种方式通知Registrar，然后 Registrar 负责向注册中心发起注册工作。同时注册中心要维护与服务之间的心跳，当服务不可用时，向注册中心注销服务。这种方式的缺点是 Registrar 必须是一个高可用的系统，否则注册工作没法进展。



客户端发现

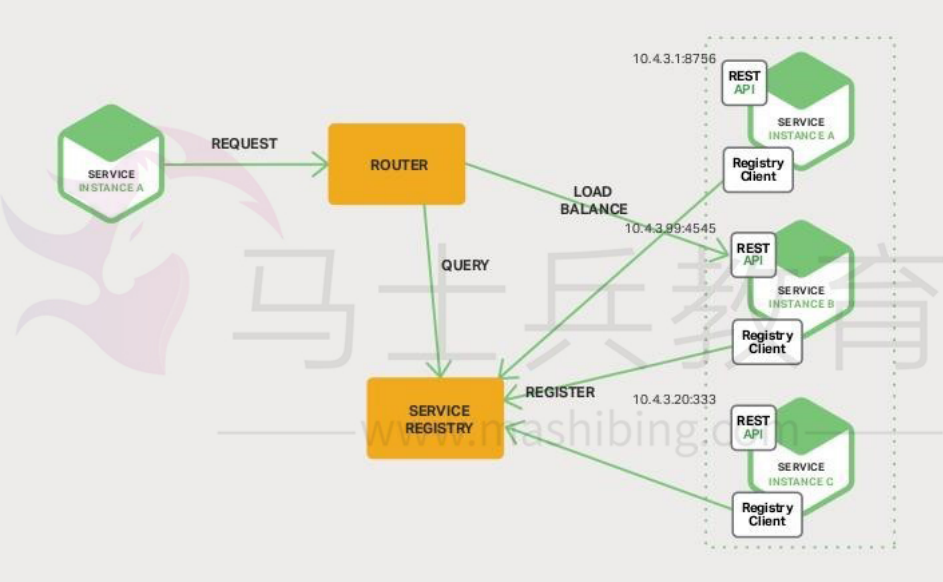
客户端发现是指客户端负责查询可用服务地址，以及负载均衡的工作。这种方式最方便直接，而且也方便做负载均衡。再者一旦发现某个服务不可用立即换另外一个，非常直接。缺点也在于多语言时的重复工作，每个语言实现相同的逻辑。



服务端发现

服务端发现需要额外的 Router 服务，请求先打到 Router，然后 Router 负责查询服务与负载均衡。

这种方式虽然没有客户端发现的缺点，但是它的缺点是保证 Router 的高可用。



Consul

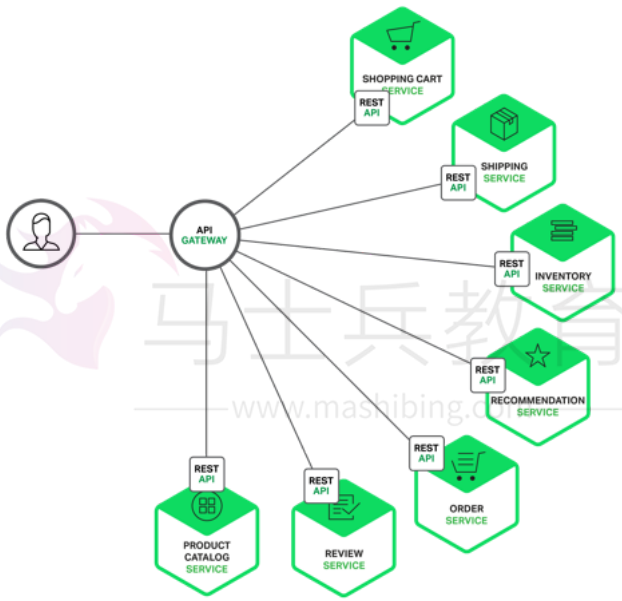
Eureka

SmartStack

Etcd

API 网关

API Gateway 是一个服务器，也可以说是进入系统的唯一节点。这跟面向对象设计模式中的 Facade 模式很像。API Gateway 封装内部系统的架构，并且提供 API 给各个客户端。它还可能有其他功能，如授权、监控、负载均衡、缓存、请求分片和管理、静态响应处理等。下图展示了一个适应当前架构的 API Gateway。



API Gateway 负责请求转发、合成和协议转换。所有来自客户端的请求都要先经过 API Gateway，

然后路由这些请求到对应的微服务。API Gateway 将经常通过调用多个微服务来处理一个请求以及聚合多个服务的结果。它可以在 web 协议与内部使用的非 Web 友好型协议间进行转换，如

HTTP 协议、WebSocket 协议。

请求转发

服务转发主要是对客户端的请求安装微服务的负载转发到不同的服务上

响应合并

把业务上需要调用多个服务接口才能完成的工作合并成一次调用对外统一提供服务。

协议转换

重点是支持 SOAP，JMS，Rest 间的协议转换。

数据转换

重点是支持 XML 和 Json 之间的报文格式转换能力（可选）

安全认证

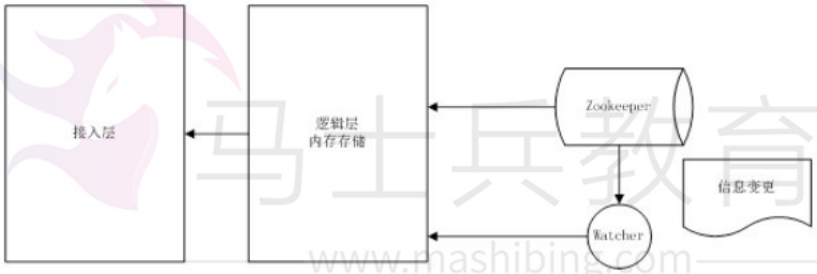
1. 基于 Token 的客户端访问控制和安全策略
2. 传输数据和报文加密，到服务端解密，需要在客户端有独立的 SDK 代理包
3. 基于 Https 的传输加密，客户端和服务端数字证书支持
4. 基于 OAuth2.0 的服务安全认证(授权码，客户端，密码模式等)

配置中心

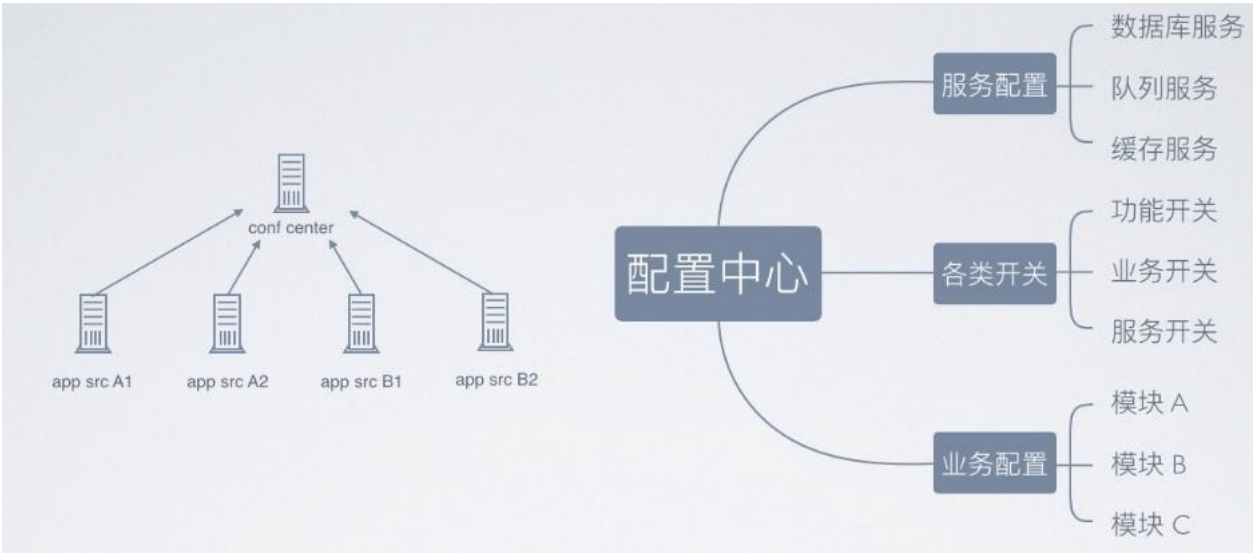
配置中心一般用作系统的参数配置，它需要满足如下几个要求：高效获取、实时感知、分布式访问。

zookeeper 配置中心

实现的架构图如下所示，采取数据加载到内存方式解决高效获取的问题，借助 zookeeper 的节点监听机制来实现实时感知。



配置中心数据分类



事件调度（kafka）

消息服务和事件的统一调度，常用用 kafka ， activemq 等。

服务跟踪（starter-sleuth）

随着微服务数量不断增长，需要跟踪一个请求从一个微服务到下一个微服务的传播过程， Spring Cloud Sleuth 正是解决这个问题，它在日志中引入唯一 ID，以保证微服务调用之间的一致性，这样你就能跟踪某个请求是如何从一个微服务传递到下一个。

1. 为了实现请求跟踪，当请求发送到分布式系统的入口端点时，只需要服务跟踪框架为该请求创建一个唯一的跟踪标识，同时在分布式系统内部流转的时候，框架始终保持传递该唯一标识，直到返回给请求方为止，这个唯一标识就是前文中提到的 Trace ID。通过 Trace ID 的记录，我们就能将所有请求过程日志关联起来。
2. 为了统计各处理单元的时间延迟，当请求达到各个服务组件时，或是处理逻辑到达某个状态时，也通过一个唯一标识来标记它的开始、具体过程以及结束，该标识就是我们前文中提到的 Span ID，对于每个 Span 来说，它必须有开始和结束两个节点，通过记录开始 Span 和结束 Span 的时间戳，就能统计出该 Span 的时间延迟，除了时间戳记录之外，它还可以包含一些其他元数据，比如：事件名称、请求信息等。
3. 在快速入门示例中，我们轻松实现了日志级别的跟踪信息接入，这完全归功于spring-cloudstarter-sleuth 组件的实现。在 Spring Boot 应用中，通过在工程中引入 spring-cloudstarter-sleuth 依赖之后， 它会自动的为当前应用构建起各通信通道的跟踪机制，比如：

   通过诸如 RabbitMQ、Kafka（或者其他任何 Spring Cloud Stream 绑定器实现的消息中间件）传递的请求。

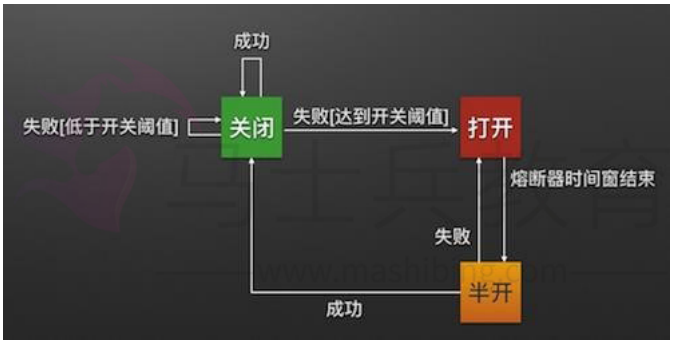
   通过 Zuul 代理传递的请求。

   通过 RestTemplate 发起的请求。

服务熔断（Hystrix）

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

熔断器的原理很简单，如同电力过载保护器。它可以实现快速失败，如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费 CPU 时间去等到长时间的超时产生。熔断器也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。



Hystrix 断路器机制

断路器很好理解, 当 Hystrix Command 请求后端服务失败数量超过一定比例(默认 50%), 断路器会切换到开路状态(Open). 这时所有请求会直接失败而不会发送到后端服务. 断路器保持在开路状态一段时间后(默认 5 秒), 自动切换到半开路状态(HALF-OPEN). 这时会判断下一次请求的返回情况, 如果请求成功, 断路器切回闭路状态(CLOSED), 否则重新切换到开路状态(OPEN). Hystrix 的断路器就像我们家庭电路中的保险丝, 一旦后端服务不可用, 断路器会直接切断请求链, 避免发送大量无效请求影响系统吞吐量, 并且断路器有自我检测并恢复的能力。

API 管理

SwaggerAPI 管理工具。

Zookeeper

Zookeeper概念

Zookeeper 是一个分布式协调服务，可用于服务发现，分布式锁，分布式领导选举，配置管理等。

Zookeeper 提供了一个类似于 Linux 文件系统的树形结构（可认为是轻量级的内存文件系统，但只适合存少量信息，完全不适合存储大量文件或者大文件），同时提供了对于每个节点的监控与通知机制。

Zookeeper角色

Zookeeper 集群是一个基于主从复制的高可用集群，每个服务器承担如下三种角色中的一种

Leader

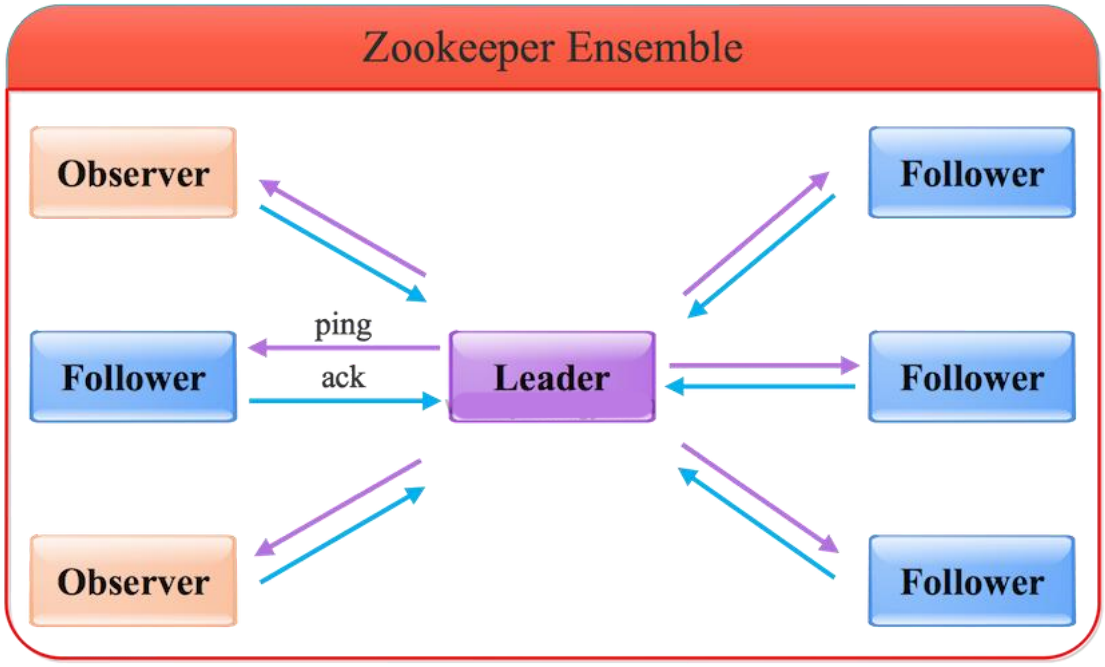
1. 一个 Zookeeper 集群同一时间只会有一个实际工作的 Leader，它会发起并维护与各 Follwer 及 Observer 间的心跳。
2. 所有的写操作必须要通过 Leader 完成再由 Leader 将写操作广播给其它服务器。只要有超过半数节点（不包括 observeer 节点）写入成功，该写请求就会被提交（类 2PC 协议）。

Follower

1. 一个 Zookeeper 集群可能同时存在多个 Follower，它会响应 Leader 的心跳，
2. Follower 可直接处理并返回客户端的读请求，同时会将写请求转发给 Leader 处理，
3. 并且负责在 Leader 处理写请求时对请求进行投票。

Observer

角色与 Follower 类似，但是无投票权。Zookeeper 需保证高可用和强一致性，为了支持更多的客户端，需要增加更多 Server；Server 增多，投票阶段延迟增大，影响性能；引入 Observer， Observer 不参与投票； Observers 接受客户端的连接，并将写请求转发给 leader 节点； 加入更多 Observer 节点，提高伸缩性，同时不影响吞吐率。



ZAB协议

事务编号 **Zxid**（事务请求计数器**+ epoch**）

在 ZAB (ZooKeeper Atomic Broadcast , ZooKeeper 原子消息广播协议） 协议的事务编号 Zxid 设计中，Zxid 是一个 64 位的数字，其中低 32 位是一个简单的单调递增的计数器，针对客户端每一个事务请求，计数器加 1；而高 32 位则代表 Leader 周期 epoch 的编号，每个当选产生一个新的 Leader 服务器，就会从这个 Leader 服务器上取出其本地日志中最大事务的 ZXID，并从中读取 epoch 值，然后加 1，以此作为新的 epoch，并将低 32 位从 0 开始计数。

Zxid（Transaction id）类似于 RDBMS 中的事务 ID，用于标识一次更新操作的 Proposal（提议）

ID。为了保证顺序性，该 zkid 必须单调递增。

epoch

epoch：可以理解为当前集群所处的年代或者周期，每个 leader 就像皇帝，都有自己的年号，所以每次改朝换代，leader 变更之后，都会在前一个年代的基础上加 1。这样就算旧的 leader 崩溃恢复之后，也没有人听他的了，因为 follower 只听从当前年代的 leader 的命令。

Zab 协议有两种模式-恢复模式（选主）、广播模式（同步）

Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 Server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 Server 具有相同的系统状态。

ZAB 协议**4**阶段

Leader election（选举阶段-选出准**Leader**）

1. Leader election（选举阶段）：节点在一开始都处于选举阶段，只要有一个节点得到超半数节点的票数，它就可以当选准 leader。只有到达 广播阶段（broadcast） 准 leader 才会成为真正的 leader。这一阶段的目的是就是为了选出一个准 leader，然后进入下一个阶段。

   Discovery（发现阶段-接受提议、生成**epoch**、接受**epoch**）

2. Discovery（发现阶段）：在这个阶段，followers 跟准 leader 进行通信，同步 followers 最近接收的事务提议。这个一阶段的主要目的是发现当前大多数节点接收的最新提议，并且准 leader 生成新的 epoch，让 followers 接受，更新它们的 accepted Epoch一个 follower 只会连接一个 leader，如果有一个节点 f 认为另一个 follower p 是 leader，f 在尝试连接 p 时会被拒绝，f 被拒绝之后，就会进入重新选举阶段。

   Synchronization（同步阶段-同步**follower**副本）

3. Synchronization（同步阶段）：同步阶段主要是利用 leader 前一阶段获得的最新提议历史，同步集群中所有的副本。只有当 大多数节点都同步完成，准 leader 才会成为真正的 leader。

   follower 只会接收 zxid 比自己的 lastZxid 大的提议。

Broadcast（广播阶段-**leader**消息广播）

4. Broadcast（广播阶段）：到了这个阶段，Zookeeper 集群才能正式对外提供事务服务，并且 leader 可以进行消息广播。同时如果有新的节点加入，还需要对新节点进行同步。

   ZAB 提交事务并不像 2PC 一样需要全部 follower 都 ACK，只需要得到超过半数的节点的 ACK 就可以了。

   ZAB 协议**JAVA** 实现（**FLE-发现阶段和同步合并为 Recovery Phase（恢复阶段）**协议的 Java 版本实现跟上面的定义有些不同，选举阶段使用的是 Fast Leader Election（FLE），它包含了 选举的发现职责。因为 FLE 会选举拥有最新提议历史的节点作为 leader，这样就省去了发现最新提议的步骤。实际的实现将 发现阶段 和 同步合并为 Recovery Phase（恢复阶段）。所以，ZAB 的实现只有三个阶段：Fast Leader Election；Recovery Phase；Broadcast Phase。

投票机制

每个 sever 首先给自己投票，然后用自己的选票和其他 sever 选票对比，权重大的胜出，使用权重较大的更新自身选票箱。具体选举过程如下：

1. 每个 Server 启动以后都询问其它的 Server 它要投票给谁。对于其他 server 的询问， server 每次根据自己的状态都回复自己推荐的 leader 的 id 和上一次处理事务的 zxid（系统启动时每个 server 都会推荐自己）
2. 收到所有 Server 回复以后，就计算出 zxid 最大的哪个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server。
3. 计算这过程中获得票数最多的的 sever 为获胜者，如果获胜者的票数超过半数，则改 server 被选为 leader。否则，继续这个过程，直到 leader 被选举出来
4. leader 就会开始等待 server 连接
5. Follower 连接 leader，将最大的 zxid 发送给 leader
6. Leader 根据 follower 的 zxid 确定同步点，至此选举阶段完成。
7. 选举阶段完成 Leader 同步后通知 follower 已经成为 uptodate 状态
8. Follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了

目前有 5 台服务器，每台服务器均没有数据，它们的编号分别是 1,2,3,4,5,按编号依次启动，它们的选择举过程如下：

1. 服务器 1 启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器 1 的状态一直属于 Looking。
2. 服务器 2 启动，给自己投票，同时与之前启动的服务器 1 交换结果，由于服务器 2 的编号大所以服务器 2 胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是 LOOKING。
3. 服务器 3 启动，给自己投票，同时与之前启动的服务器 1,2 交换信息，由于服务器 3 的编号最大所以服务器 3 胜出，此时投票数正好大于半数，所以服务器 3 成为领导者，服务器1,2 成为小弟。
4. 服务器 4 启动，给自己投票，同时与之前启动的服务器 1,2,3 交换信息，尽管服务器 4 的编号大，但之前服务器 3 已经胜出，所以服务器 4 只能成为小弟。
5. 服务器 5 启动，后面的逻辑同服务器 4 成为小弟。

Zookeeper 工作原理（原子广播）

1. Zookeeper 的核心是原子广播，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。
2. 当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 server 的完成了和 leader 的状态同步以后，恢复模式就结束了。
3. 状态同步保证了 leader 和 server 具有相同的系统状态
4. 一旦 leader 已经和多数的 follower 进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个 server 加入 zookeeper 服务中，它会在恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。Zookeeper服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。
5. 广播模式需要保证 proposal 被按顺序处理，因此 zk 采用了递增的事务 id 号(zxid)来保证。所有的提议(proposal)都在被提出的时候加上了 zxid。
6. 实现中 zxid 是一个 64 为的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch。低 32 位是个递增计数。
7. 当 leader 崩溃或者 leader 失去大多数的 follower，这时候 zk 进入恢复模式，恢复模式需要重新选举出一个新的 leader，让所有的 server 都恢复到一个正确的状态。

Netty（周老师）

Netty简介



Netty是 一个异步事件驱动的网络应用程(img)序框架，用于快速开发可维护的高性能协议服务器和客户端。

JDK原生NIO程序的问题

JDK原生也有一套网络应用程序API，但是存在一系列问题，主要如下：

- NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等
- 需要具备其它的额外技能做铺垫，例如熟悉Java多线程编程，因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序
- 可靠性能力补齐，开发工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等等，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大
- JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询， 终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到
- JDK1.7版本该问题仍旧存在，只不过该bug发生概率降低了一些而已，它并没有被根本解决

Netty的特点

- Netty的对JDK自带的NIO的API进行封装，解决上述问题，主要特点有：
- 设计优雅 适用于各种传输类型的统一API - 阻塞和非阻塞Socket 基于灵活且可扩展的事件模型，可以清晰地分离关注点 高度可定制的线程模型 - 单线程，一个或多个线程池 真正的无连接数据报套接字支持（自3.1起）
- 使用方便 详细记录的Javadoc，用户指南和示例 没有其他依赖项，JDK 5（Netty 3.x）或6（Netty 4.x）就足够了
- 高性能 吞吐量更高，延迟更低 减少资源消耗 小化不必要的内存复制安全 完整的SSL / TLS和StartTLS支持
- 社区活跃，不断更新 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会被加入

Netty常见使用场景

Netty常见的使用场景如下：

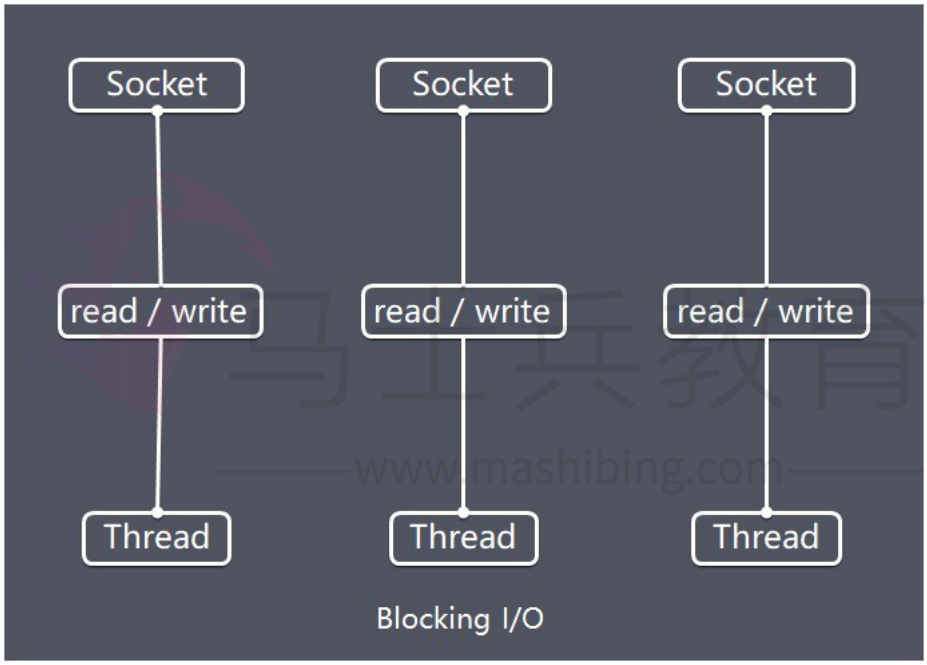
- 互联网行业 在分布式系统中，各个节点之间需要远程服务调用，高性能的RPC 框架必不可少，Netty作为异步高新能的通信框架,往往作为基础通信组件被这些RPC 框架使用。 典型的应用有：阿里分布式服务框架Dubbo的RPC框架使用Dubbo协议进行节点间通信，Dubbo协议默认使用Netty作为基础通信组件，用于实现各进程节点之间的内部通信。
- 游戏行业 无论是手游服务端还是大型的网络游戏，Java语言得到了越来越广泛的应用。Netty作为高性能的基础通信组件，它本身提供了TCP/UDP和HTTP协议栈。 非常方便定制和开发私有协议栈，账号登录服务器，地图服务器之间可以方便的通过Netty进行高性能的通信
- 大数据领域 经典的Hadoop的高性能通信和序列化组件Avro的RPC框架，默认采用Netty进行跨界点通信，它的Netty Service基于Netty框架二次封装实现有兴趣的读者可以了解一下目前有哪些开源项目使用了 Netty：Related projects

Netty高性能设计

Netty作为异步事件驱动的网络，高性能之处主要来自于其I/O模型和线程处理模型，前者决定如何收发数据，后者决定如何处理数据

I/O模型

用什么样的通道将数据发送给对方，BIO、NIO或者AIO，I/O模型在很大程度上决定了框架的性能阻塞I/O 传统阻塞型I/O(BIO)可以用下图表示：



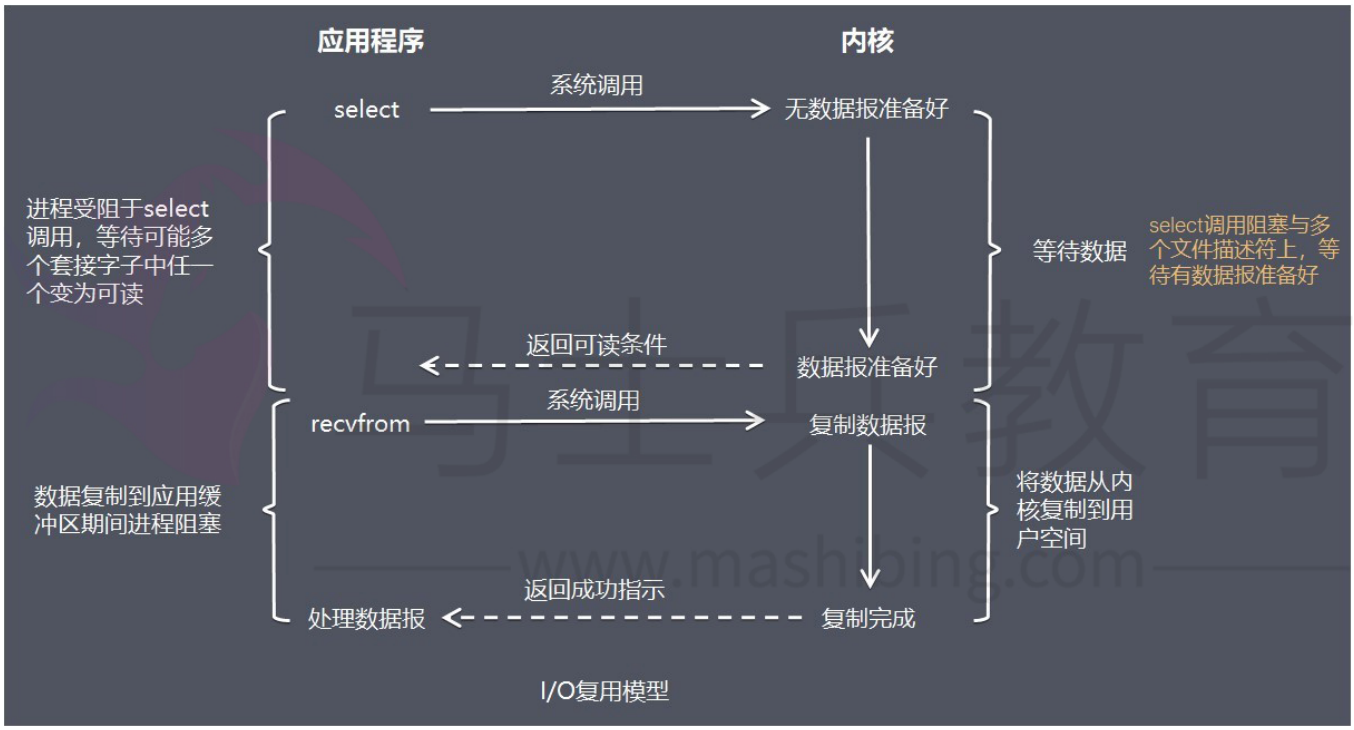
特点

每个请求都需要独立的线程完成数据read，业务处理，数据write的完整操作问题

当并发数较大时，需要创建大量线程来处理连接，系统资源占用较大

连接建立后，如果当前线程暂时没有数据可读，则线程就阻塞在read操作上，造成线程资源浪费

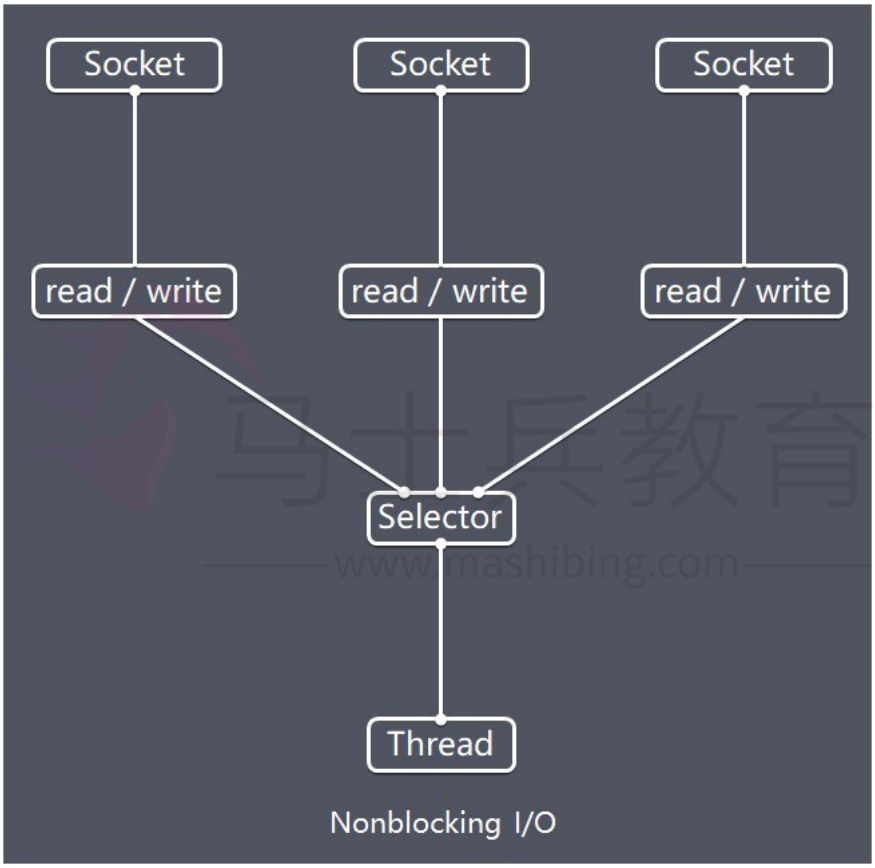
I/O复用模型



在I/O复用模型中，会用到select，这个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作，而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用

I/O操作函数

Netty的非阻塞I/O的实现关键是基于I/O复用模型，这里用Selector对象表示：



Netty的IO线程NioEventLoop由于聚合了多路复用器Selector，可以同时并发处理成百上千个客户端连接。当线程从某客户端Socket通道进行读写数据时，若没有数据可用时，该线程可以进行其他任务。线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管理多个输入和输出通道。

由于读写操作都是非阻塞的，这就可以充分提升IO线程的运行效率，避免由于频繁I/O阻塞导致的线程挂起，一个I/O线程可以并发处理N个客户端连接和读写操作，这从根本上解决了传统同步阻塞I/O一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

基于buffer

传统的I/O是面向字节流或字符流的，以流式的方式顺序地从一个Stream 中读取一个或多个字节, 因此也就不能随意改变读取指针的位置。

在NIO中, 抛弃了传统的 I/O流, 而是引入了Channel和Buffer的概念. 在NIO中,

只能从Channel中读取数据到Buffer中或将数据 Buffer 中写入到 Channel。

基于buffer操作不像传统IO的顺序操作, NIO 中可以随意地读取任意位置的数据线程模型

数据报如何读取？读取之后的编解码在哪个线程进行，编解码后的消息如何派发，线程模型的不同，对性能的影响也非常大。

事件驱动模型

通常，我们设计一个事件处理模型的程序有两种思路

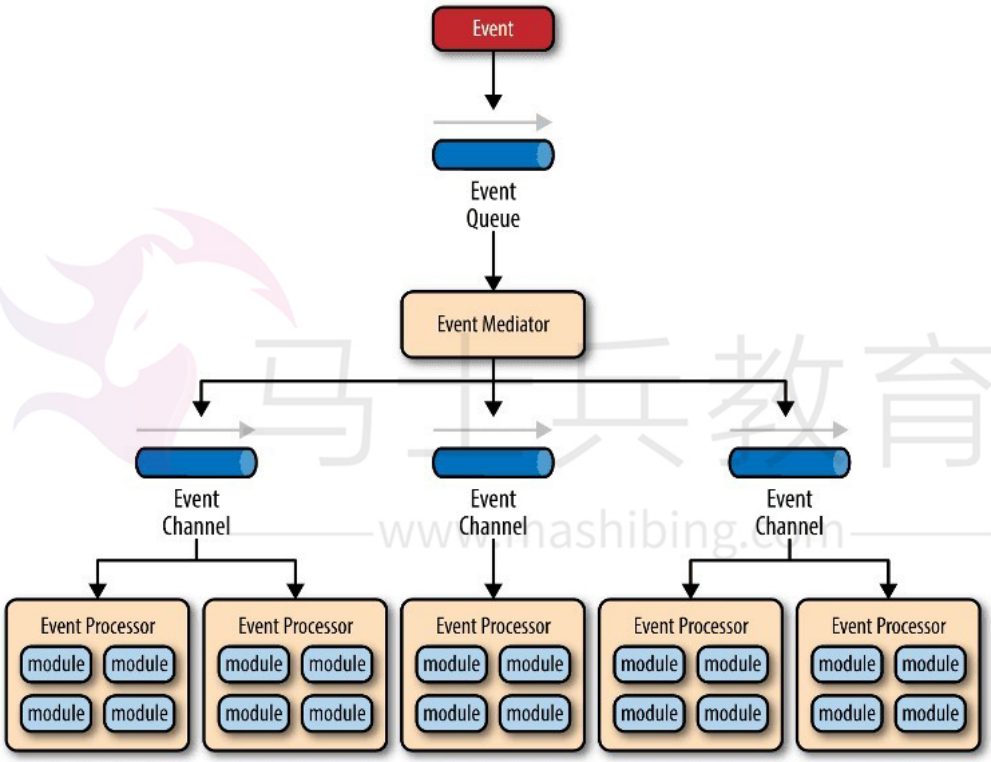
轮询方式 线程不断轮询访问相关事件发生源有没有发生事件，有发生事件就调用事件处理逻辑。

事件驱动方式 发生事件，主线程把事件放入事件队列，在另外线程不断循环消费事件列表中的事件，调用事件对应的处理逻辑处理事件。事件驱动方式也被称为消息通知方式，其实是设计模式中观察者模式的思路。

以GUI的逻辑处理为例，说明两种逻辑的不同：

轮询方式 线程不断轮询是否发生按钮点击事件，如果发生，调用处理逻辑

事件驱动方式 发生点击事件把事件放入事件队列，在另外线程消费的事件列表中的事件，根据事件类型调用相关事件处理逻辑这里借用O'Reilly 大神关于事件驱动模型解释图



主要包括4个基本组件：

- 事件队列（event queue）：接收事件的入口，存储待处理事件
- 分发器（event mediator）：将不同的事件分发到不同的业务逻辑单元
- 事件通道（event channel）：分发器与处理器之间的联系渠道
- 事件处理器（event processor）：实现业务逻辑，处理完成后会发出事件，触发下一步操作

可以看出，相对传统轮询模式，事件驱动有如下优点：

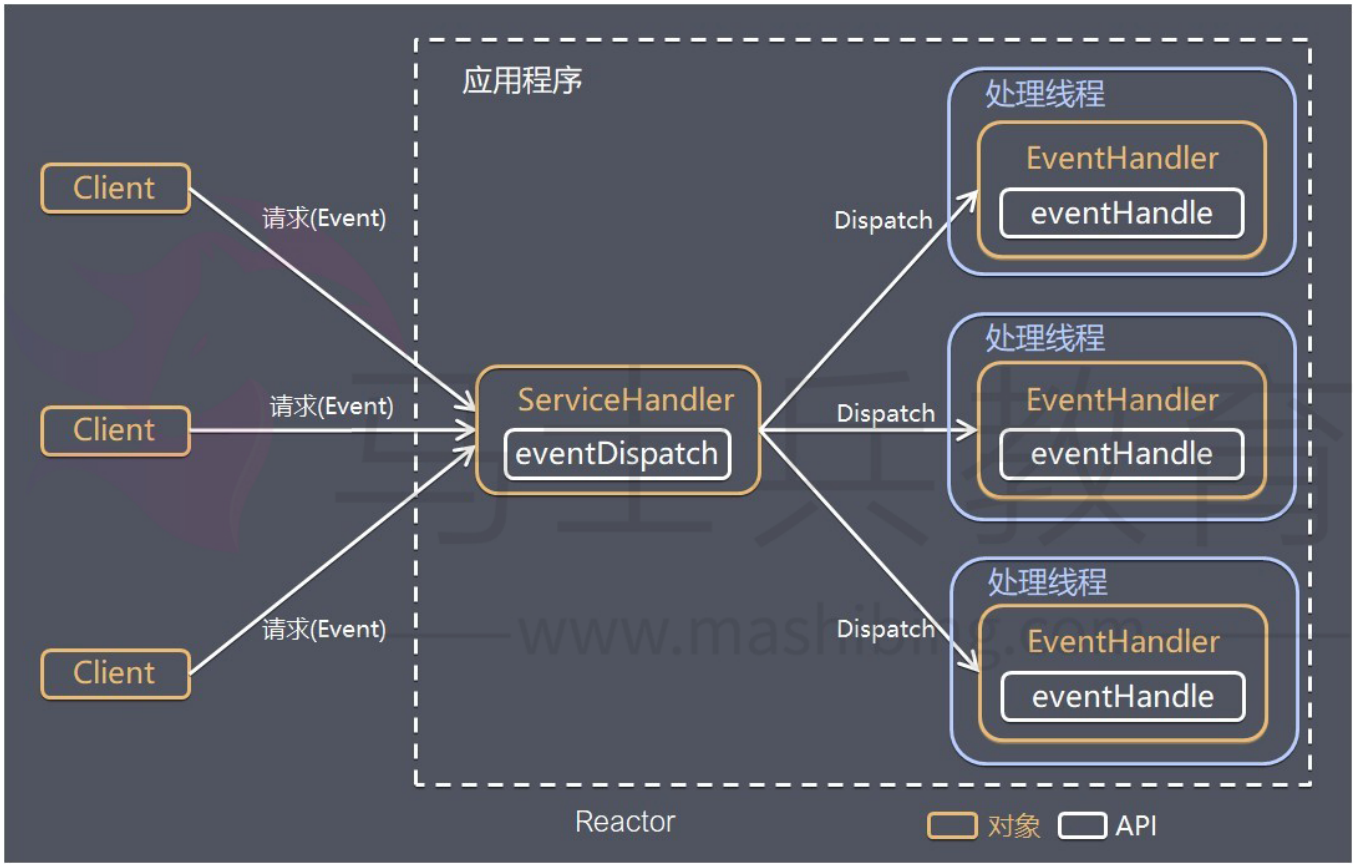
- 可扩展性好，分布式的异步架构，事件处理器之间高度解耦，可以方便扩展事件处理逻辑
- 高性能，基于队列暂存事件，能方便并行异步处理事件

Reactor线程模型

Reactor是反应堆的意思，Reactor模型，是指通过一个或多个输入同时传递给服务处理器的服务请求的事件驱动处理模式。 服务端程序处理传入多路请求，并将它们同步分派给请求对应的处理线程，Reactor模式也叫Dispatcher模式，即I/O多了复用统一监听事件，收到事件后分发(Dispatch给某进程)，是编写高性能网络服务器的必备技术之一。

Reactor模型中有2个关键组成：

- Reactor Reactor在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理程序来对IO事件做出反应。 它就像公司的电话接线员，它接听来自客户的电话并将线路转移到适当的联系人
- Handlers 处理程序执行I/O事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际官员。Reactor通过调度适当的处理程序来响应I/O事件，处理程序执行非阻塞操作



取决于Reactor的数量和Hanndler线程数量的不同，Reactor模型有3个变种

- 单Reactor单线程
- 单Reactor多线程
- 主从Reactor多线程

可以这样理解，Reactor就是一个执行while (true) { selector.select(); ...}循环的线程，会源源不断的产生新的事件，称作反应堆很贴切。

Netty线程模型

Netty主要基于主从Reactors多线程模型（如下图）做了一定的修改，其中主从

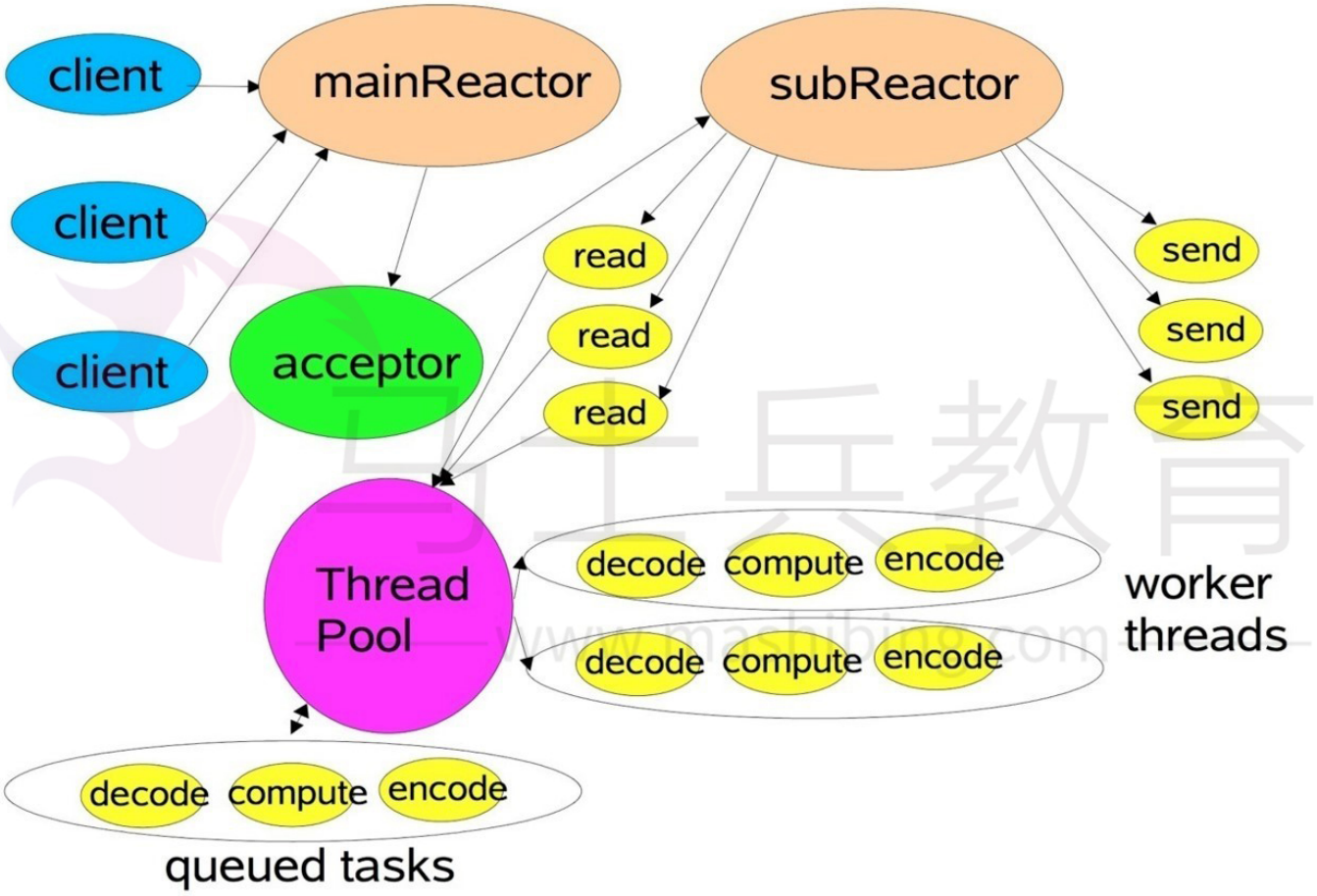
Reactor多线程模型有多个Reactor：MainReactor和SubReactor：

MainReactor负责客户端的连接请求，并将请求转交给SubReactor SubReactor负责相应通道的IO读写请求

非IO请求（具体逻辑处理）的任务则会直接写入队列，等待worker threads进

行处理

这里引用Doug Lee大神的Reactor介绍：Scalable IO in Java里面关于主从 Reactor多线程模型的图



特别说明的是： 虽然Netty的线程模型基于主从Reactor多线程，借用了

MainReactor和SubReactor的结构，但是实际实现上，SubReactor和Worker 线程在同一个线程池中：

```
1   EventLoopGroup bossGroup =newNioEventLoopGroup();
2   EventLoopGroup workerGroup =newNioEventLoopGroup();
3   ServerBootstrap server =newServerBootstrap();
4   server.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
```

- bossGroup线程池则只是在bind某个端口后，获得其中一个线程作为MainReactor，专门处理端口的accept事件，**每个端口对应一个boss线程**
- workerGroup线程池会被各个SubReactor和worker线程充分利用

异步处理

异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

Netty中的I/O操作是异步的，包括bind、write、connect等操作会简单的返回一个ChannelFuture，调用者并不能立刻获得结果，通过Future-Listener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果。

当future对象刚刚创建时，处于非完成状态，调用者可以通过返回的ChannelFuture来获取操作执行的状态，注册监听函数来执行完成后的操，常见有如下操作：

- 通过isDone方法来判断当前操作是否完成
- 通过isSuccess方法来判断已完成的当前操作是否成功
- 通过getCause方法来获取已完成的当前操作失败的原因
- 通过isCancelled方法来判断已完成的当前操作是否被取消
- 通过addListener方法来注册监听器，当操作已完成(isDone方法返回完成)，将会通知指定的监听器；如果future对象已完成，则理解通知指定的监听器

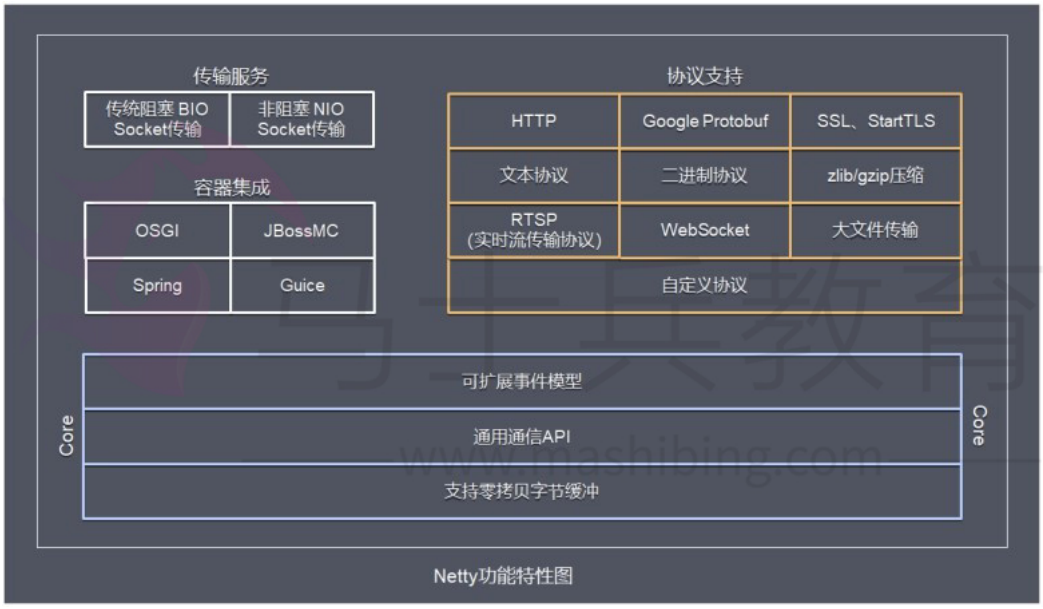
例如下面的的代码中绑定端口是异步操作，当绑定操作处理完，将会调用相应的监听器处理逻辑

```
1 serverBootstrap.bind(port).addListener(future -> {
2 if (future.isSuccess()) {
3 System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");
4 } else {
5 System.err.println("端口[" + port + "]绑定失败!");
6 }
7 });
```

相比传统阻塞I/O，执行I/O操作后线程会被阻塞住, 直到操作完成；异步处理的 好处是不会造成线程阻塞，线程在I/O操作期间可以执行别的程序，在高并发情 形下会更稳定和更高的吞吐量。

Netty架构设计

前面介绍完Netty相关一些理论介绍，下面从功能特性、模块组件、运作过程来介绍Netty的架构设计功能特性



- 传输服务 支持BIO和NIO
- 容器集成 支持OSGI、JBossMC、Spring、Guice容器
- 协议支持 HTTP、Protobuf、二进制、文本、WebSocket等一系列常见协议都支持。 还支持通过实行编码解码逻辑来实现自定义协议
- Core核心 可扩展事件模型、通用通信API、支持零拷贝的ByteBuf缓冲对象

模块组件

Bootstrap、ServerBootstrap

Bootstrap意思是引导，一个Netty应用通常由一个Bootstrap开始，主要作用是配置整个Netty程序，串联各个组件，Netty中Bootstrap类是客户端程序的启动引导类，ServerBootstrap是服务端启动引导类。

Future、ChannelFuture

正如前面介绍，在Netty中所有的IO操作都是异步的，不能立刻得知消息是否被正确处理，但是可以过一会等它执行完成或者直接注册一个监听，具体的实现就是通过Future和ChannelFutures，他们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册的监听事件。

Channel

Netty网络通信的组件，能够用于执行网络I/O操作。 Channel为用户提供：

- 当前网络连接的通道的状态（例如是否打开？是否已连接？）
- 网络连接的配置参数 （例如接收缓冲区大小）
- 提供异步的网络I/O操作(如建立连接，读写，绑定端口)，异步调用意味着任何I / O调用都将立即返回，并且不保证在调用结束时所请求的I / O操作已完成。调用立即返回一个ChannelFuture实例，通过注册监听器到ChannelFuture上，可以I / O操作成功、失败或取消时回调通知调用方。
- 支持关联I/O操作与对应的处理程序

不同协议、不同的阻塞类型的连接都有不同的 Channel 类型与之对应，下面是一些常用的 Channel 类型

- NioSocketChannel，异步的客户端 TCP Socket 连接
- NioServerSocketChannel，异步的服务器端 TCP Socket 连接
- NioDatagramChannel，异步的 UDP 连接
- NioSctpChannel，异步的客户端 Sctp 连接
- NioSctpServerChannel，异步的 Sctp 服务器端连接 这些通道涵盖了 UDP 和 TCP网络 IO以及文件 IO.

Selector

Netty基于Selector对象实现I/O多路复用，通过 Selector, 一个线程可以监听多个连接的Channel事件, 当向一个Selector中注册Channel 后，Selector 内部的机制就可以自动不断地查询(select) 这些注册的Channel是否有已就绪的I/O事件(例如可读, 可写, 网络连接完成等)，这样程序就可以很简单地使用一个线程高效地管理多个 Channel 。

NioEventLoop

NioEventLoop中维护了一个线程和任务队列，支持异步提交执行任务，线程启动时会调用NioEventLoop的run方法，执行I/O任务和非I/O任务：

- I/O任务 即selectionKey中ready的事件，如accept、connect、read、write等，由processSelectedKeys方法触发。
- 非IO任务 添加到taskQueue中的任务，如register0、bind0等任务，由runAllTasks方法触发。

两种任务的执行时间比由变量ioRatio控制，默认为50，则表示允许非IO任务执行的时间与IO任务的执行时间相等。

NioEventLoopGroup

NioEventLoopGroup，主要管理eventLoop的生命周期，可以理解为一个线程池，内部维护了一组线程，每个线程(NioEventLoop)负责处理多个Channel上的事件，而一个Channel只对应于一个线程。 ChannelHandler

ChannelHandler是一个接口，处理I / O事件或拦截I / O操作，并将其转发到其 ChannelPipeline(业务处理链)中的下一个处理程序。

ChannelHandler本身并没有提供很多方法，因为这个接口有许多的方法需要实现，方便使用期间，可以继承它的子类：

- ChannelInboundHandler用于处理入站I / O事件
- ChannelOutboundHandler用于处理出站I / O操作

或者使用以下适配器类：

- ChannelInboundHandlerAdapter用于处理入站I / O事件
- ChannelOutboundHandlerAdapter用于处理出站I / O操作
- ChannelDuplexHandler用于处理入站和出站事件

ChannelHandlerContext 保存Channel相关的所有上下文信息，同时关联一个ChannelHandler对象 ChannelPipline

保存ChannelHandler的List，用于处理或拦截Channel的入站事件和出站操作。 ChannelPipeline实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及Channel中各个的ChannelHandler如何相互交互。

下图引用Netty的Javadoc4.1中ChannelPipline的说明，描述了 ChannelPipeline中ChannelHandler通常如何处理I/O事件。 I/O事件由

ChannelInboundHandler或ChannelOutboundHandler处理，并通过调用

ChannelHandlerContext中定义的事件传播方法（例如

ChannelHandlerContext.fireChannelRead（Object）和

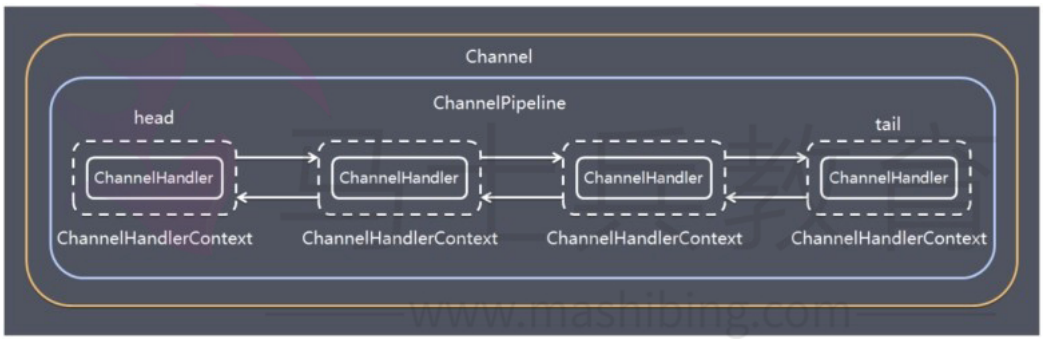
ChannelOutboundInvoker.write（Object））转发到其 近的处理程序。

```
1   I/O Request
2   via Channel or
3   ChannelHandlerContext
4   |
5   +---------------------------------------------------+---------------+
6   | ChannelPipeline | |
7   | \|/ |
8   | +--------------------+ +-----------+----------+ |
9   | | Inbound Handler N | | Outbound Handler 1 | |
10   | +----------+----------+ +-----------+----------+ |
11   | /|\ | |
12   | | \|/ |
13   | +----------+----------+ +-----------+----------+ |
14   | | Inbound Handler N-1 | | Outbound Handler 2 | |
15   | +----------+----------+ +-----------+----------+ |
16   | /|\ . |
17   | . . |
18   | ChannelHandlerContext.fireIN_EVT() ChannelHandlerContext.OUT_EVT()|
19   | [ method call] [method call] |
20   | . . |
21   | . \|/ |
22   | +----------+----------+ +-----------+----------+ |
23   | | Inbound Handler 2 | | Outbound Handler M-1 | |
24   | +----------+----------+ +-----------+----------+ |
25   | /|\ | |
26   | | \|/ |
27   | +----------+----------+ +-----------+----------+ |
28   | | Inbound Handler 1 | | Outbound Handler M | |
29   | +----------+----------+ +-----------+----------+
30   | /|\ | |
31   +---------------+-----------------------------------+---------------+
32   | \|/
33   +---------------+-----------------------------------+---------------+
34   | | | |
35   | [ Socket.read() ] [ Socket.write() ] |
36   | |
37   | Netty Internal I/O Threads (Transport Implementation) |
38   +-------------------------------------------------------------------+ 39
40
```

入站事件由自下而上方向的入站处理程序处理，如图左侧所示。 入站Handler处理程序通常处理由图底部的I / O线程生成的入站数据。 通常通过实际输入操作（例如SocketChannel.read（ByteBuffer））从远程读取入站数据。

出站事件由上下方向处理，如图右侧所示。 出站Handler处理程序通常会生成或转换出站传输，例如write请求。 I/O线程通常执行实际的输出操作，例如SocketChannel.write（ByteBuffer）。

在 Netty 中每个 Channel 都有且仅有一个 ChannelPipeline 与之对应, 它们的组成关系如下:



一个 Channel 包含了一个 ChannelPipel(img)ine, 而 ChannelPipeline 中又维护了

一个由 ChannelHandlerContext 组成的双向链表, 并且每个

ChannelHandlerContext 中又关联着一个 ChannelHandler。入站事件和出站

事件在一个双向链表中，入站事件会从链表head往后传递到 后一个入站的

handler，出站事件会从链表tail往前传递到 前一个出站的handler，两种类型的handler互不干扰。

工作原理架构

初始化并启动Netty服务端过程如下：

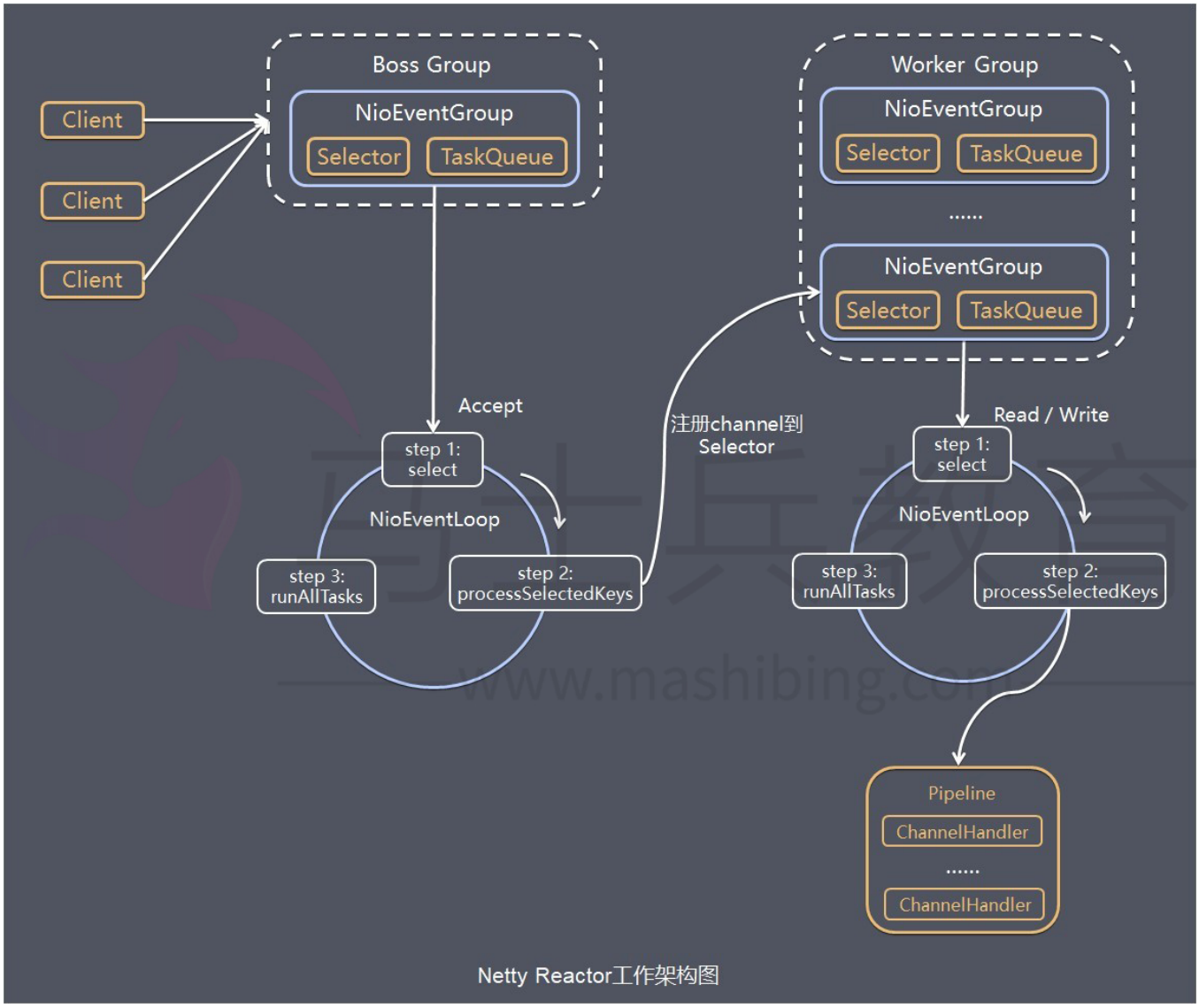
```
1   public static void main(String[] args) {
2   // 创建mainReactor
3   NioEventLoopGroup boosGroup = new NioEventLoopGroup();
4   // 创建工作线程组
5   NioEventLoopGroup workerGroup = new NioEventLoopGroup();
6
7   final ServerBootstrap serverBootstrap = new ServerBootstrap();
8   serverBootstrap
9   // 组装NioEventLoopGroup
10  .group(boosGroup, workerGroup)
11  // 设置channel类型为NIO类型
12  .channel(NioServerSocketChannel.class)
13  // 设置连接配置参数
14  .option(ChannelOption.SO_BACKLOG, 1024)
15  .childOption(ChannelOption.SO_KEEPALIVE, true)
16  .childOption(ChannelOption.TCP_NODELAY, true)
17  // 配置入站、出站事件handler
18  .childHandler(new ChannelInitializer<NioSocketChannel>() {
19  @Override
20  protected void initChannel(NioSocketChannel ch) {
21  // 配置入站、出站事件channel
22  ch.pipeline().addLast(...); 23 ch.pipeline().addLast(...);
24  }
25  });
26
27  // 绑定端口
28  int port = 8080;
->
29  serverBootstrap.bind(port).addListener(future  {
30  if (future.isSuccess()) {
31  System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");
32  } else {
33  System.err.println("端口[" + port + "]绑定失败!");
34  }
35  });36   }
```

基本过程如下：

1 初始化创建2个NioEventLoopGroup，其中boosGroup用于Accetpt连接建立事件并分发请求， workerGroup用于处理I/O读写事件和业务逻辑

2 基于ServerBootstrap(服务端启动引导类)，配置EventLoopGroup、Channel类型，连接参数、配置入站、出站事件handler

3 绑定端口，开始工作结合上面的介绍的Netty Reactor模型，介绍服务端Netty的工作架构图：



server端包含1个Boss NioEventLoopGroup和1个Worker

NioEventLoopGroup，NioEventLoopGroup相当于1个事件循环组，这个组

里包含多个事件循环NioEventLoop，每个NioEventLoop包含1个selector和1 个事件循环线程。

每个Boss NioEventLoop循环执行的任务包含3步：

1 轮询accept事件

2 处理accept I/O事件，与Client建立连接，生成NioSocketChannel，并将NioSocketChannel注册到某个Worker NioEventLoop的Selector上 *3 处理任务队列中的任务，runAllTasks。任务队列中的任务包括用户调用eventloop.execute或 schedule执行的任务，或者其它线程提交到该eventloop的任务。

每个Worker NioEventLoop循环执行的任务包含3步：

1 轮询read、write事件；

2 处I/O事件，即read、write事件，在NioSocketChannel可读、可写事件发生

时进行处理

```
        3 处理任务队列中的任务，runAllTasks。
```

其中任务队列中的task有3种典型使用场景

```
        1 用户程序自定义的普通任务
```

```
1   ctx.channel().eventLoop().execute(new Runnable() {
2   @Override
3   public void run() { 4 //...
5   }
6   });
```

2 非当前reactor线程调用channel的各种方法 例如在推送系统的业务线程里面，根据用户的标识，找到对应的channel引用，然后调用write类方法向该用户推送消息，就会进入到这种场景。 终的write会提交到任务队列中后被异步消费。

```
        3 用户自定义定时任务
```

```
ctx.channel().eventLoop().schedule(new Runnable() {
@Override
public void run() {

}
}, 60, TimeUnit.SECONDS);


```

总结

现在稳定推荐使用的主流版本还是Netty4，Netty5 中使用了 ForkJoinPool，

增加了代码的复杂度，但是对性能的改善却不明显，所以这个版本不推荐使用，官网也没有提供下载链接。

Netty 入门门槛相对较高，其实是因为这方面的资料较少，并不是因为他有多

难，大家其实都可以像搞透 Spring 一样搞透 Netty。在学习之前，建议先理解透整个框架原理结构，运行过程，可以少走很多弯路。

Netty面试题

1. Netty 是什么?

Netty是 一个异步事件驱动的网络应用程序框架，用于快速开发可维护的高性能 协议服务器和客户端。Netty是基于nio的，它封装了jdk的nio，让我们使用起 来更加方法灵活。

2. Netty 特点是什么?

- 高并发：Netty 是一款基于 NIO（Nonblocking IO，非阻塞IO）开发的网络通 信框架，对比于 BIO（Blocking I/O，阻塞IO），他的并发性能得到了很大提高。
- 传输快：Netty 的传输依赖于零拷贝特性，尽量减少不必要的内存拷贝，实现了 更高效率的传输。
- 封装好：Netty 封装了 NIO 操作的很多细节，提供了易于使用调用接口。

3. Netty 的优势有哪些?

- 使用简单：封装了 NIO 的很多细节，使用更简单。
- 功能强大：预置了多种编解码功能，支持多种主流协议。
- 定制能力强：可以通过 ChannelHandler 对通信框架进行灵活地扩展。
- 性能高：通过与其他业界主流的 NIO 框架对比，Netty 的综合性能优。
- 稳定：Netty 修复了已经发现的所有 NIO 的 bug，让开发人员可以专注于业务 本身。
- 社区活跃：Netty 是活跃的开源项目，版本迭代周期短，bug 修复速度快。

4. Netty 的应用场景有哪些?

典型的应用有：阿里分布式服务框架 Dubbo，默认使用 Netty 作为基础通信组 件，还有 RocketMQ 也是使用 Netty 作为通讯的基础。

5. Netty 高性能表现在哪些方面?

- IO 线程模型：同步非阻塞，用少的资源做更多的事。
- 内存零拷贝：尽量减少不必要的内存拷贝，实现了更高效率的传输。
- 内存池设计：申请的内存可以重用，主要指直接内存。内部实现是用一颗二叉查 找树管理内存分配情况。
- 串形化处理读写：避免使用锁带来的性能开销。
- 高性能序列化协议：支持 protobuf 等高性能序列化协议。

6. BIO、NIO和AIO的区别?

BIO：一个连接一个线程，客户端有连接请求时服务器端就需要启动一个线程进 行处理。线程开销大。

伪异步IO：将请求连接放入线程池，一对多，但线程还是很宝贵的资源。

NIO：一个请求一个线程，但客户端发送的连接请求都会注册到多路复用器上， 多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

AIO：一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务 器应用去启动线程进行处理，

BIO是面向流的，NIO是面向缓冲区的；BIO的各种流是阻塞的。而NIO是非阻 塞的；BIO的Stream是单向的，而NIO的channel是双向的。

NIO的特点：事件驱动模型、单线程处理多任务、非阻塞I/O，I/O读写不再阻 塞，而是返回0、基于block的传输比基于流的传输更高效、更高级的IO函数 zero-copy、IO多路复用大大提高了Java网络应用的可伸缩性和实用性。基于 Reactor线程模型。

在Reactor模式中，事件分发器等待某个事件或者可应用或个操作的状态发生， 事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来 做实际的读写操作。如在Reactor中实现读：注册读就绪事件和相应的事件处理 器、事件分发器等待事件、事件到来，激活分发器，分发器调用事件对应的处理 器、事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还 控制权。

7. NIO的组成?

Buffer：与Channel进行交互，数据是从Channel读入缓冲区，从缓冲区写入 Channel中的

flip方法： 反转此缓冲区，将position给limit，然后将position置为0，其实就 是切换读写模式

clear方法： 清除此缓冲区，将position置为0，把capacity的值给limit。

rewind方法： 重绕此缓冲区，将position置为0

DirectByteBuffer可减少一次系统空间到用户空间的拷贝。但Buffer创建和销毁 的成本更高，不可控，通常会用内存池来提高性能。直接缓冲区主要分配给那些易受基础系统的本机I/O 操作影响的大型、持久的缓冲区。如果数据量比较小的 中小应用情况下，可以考虑使用heapBuffer，由JVM进行管理。

Channel：表示 IO 源与目标打开的连接，是双向的，但不能直接访问数据，只 能与Buffer 进行交互。通过源码可知，FileChannel的read方法和write方法都 导致数据复制了两次！

Selector可使一个单独的线程管理多个Channel，open方法可创建Selector， register方法向多路复用器器注册通道，可以监听的事件类型：读、写、连接、 accept。注册事件后会产生一个SelectionKey：它表示SelectableChannel 和 Selector 之间的注册关系，wakeup方法：使尚未返回的第一个选择操作立即返 回，唤醒的

原因是：注册了新的channel或者事件；channel关闭，取消注册；优先级更高 的事件触发（如定时器事件），希望及时处理。

Selector在Linux的实现类是EPollSelectorImpl，委托给EPollArrayWrapper实 现，其中三个native方法是对epoll的封装，而EPollSelectorImpl.

implRegister方法，通过调用epoll_ctl向epoll实例中注册事件，还将注册的文 件描述符(fd)与SelectionKey的对应关系添加到fdToKey中，这个map维护了文 件描述符与SelectionKey的映射。

fdToKey有时会变得非常大，因为注册到Selector上的Channel非常多（百万连 接）；过期或失效的Channel没有及时关闭。fdToKey总是串行读取的，而读取 是在select方法中进行的，该方法是非线程安全的。

Pipe：两个线程之间的单向数据连接，数据会被写到sink通道，从source通道 读取

NIO的服务端建立过程：Selector.open()：打开一个Selector；

ServerSocketChannel.open()：创建服务端的Channel；bind()：绑定到某个 端口上。并配置非阻塞模式；register()：注册Channel和关注的事件到 Selector上；select()轮询拿到已经就绪的事件

8. Netty的线程模型?

Netty通过Reactor模型基于多路复用器接收并处理用户请求，内部实现了两个 线程池，boss线程池和work线程池，其中boss线程池的线程负责处理请求的 accept事件，当接收到accept事件的请求时，把对应的socket封装到一个NioSocketChannel中，并交给work线程池，其中work线程池负责请求的read 和write事件，由对应的Handler处理。

单线程模型：所有I/O操作都由一个线程完成，即多路复用、事件分发和处理都 是在一个Reactor线程上完成的。既要接收客户端的连接请求,向服务端发起连 接，又要发送/读取请求或应答/响应消息。一个NIO 线程同时处理成百上千的 链路，性能上无法支撑，速度慢，若线程进入死循环，整个程序不可用，对于高 负载、大并发的应用场景不合适。

多线程模型：有一个NIO 线程（Acceptor） 只负责监听服务端，接收客户端的 TCP 连接请求；NIO 线程池负责网络IO 的操作，即消息的读取、解码、编码和 发送；1 个NIO 线程可以同时处理N 条链路，但是1 个链路只对应1 个NIO 线 程，这是为了防止发生并发操作问题。但在并发百万客户端连接或需要安全认证 时，一个Acceptor 线程可能会存在性能不足问题。

主从多线程模型：Acceptor 线程用于绑定监听端口，接收客户端连接，将 SocketChannel 从主线程池的Reactor 线程的多路复用器上移除，重新注册到 Sub 线程池的线程上，用于处理I/O 的读写等操作，从而保证mainReactor只负 责接入认证、握手等操作；

9. TCP 粘包/拆包的原因及解决方法?

TCP是以流的方式来处理数据，一个完整的包可能会被TCP拆分成多个包进行发 送，也可能把小的封装成一个大的数据包发送。

TCP粘包/分包的原因：

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应 用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络 上，这将会发生粘包现象；

进行MSS大小的TCP分段，当TCP报文长度-TCP头部长度>MSS的时候将发生拆 包

以太网帧的payload（净荷）大于MTU（1500字节）进行ip分片。

解决方法

消息定长：FixedLengthFrameDecoder类

包尾增加特殊字符分割：

- 行分隔符类：LineBasedFrameDecoder
- 或自定义分隔符类 ： DelimiterBasedFrameDecoder

将消息分为消息头和消息体：LengthFieldBasedFrameDecoder类。分为有头 部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘 包。

10. 什么是 Netty 的零拷贝?

Netty 的零拷贝主要包含三个方面：

- Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进 行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存 （HEAP BUFFERS）进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内 存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓 冲区的内存拷贝。
- Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操 作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式 将几个小 Buffer 合并成一个大的 Buffer。
- Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发 送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

11.Netty 中有哪种重要组件?

- Channel：Netty 网络操作抽象类，它除了包括基本的 I/O 操作，如 bind、 connect、read、write 等。
- EventLoop：主要是配合 Channel 处理 I/O 操作，用来处理连接的生命周期中所发生的事情。
- ChannelFuture：Netty 框架中所有的 I/O 操作都为异步的，因此我们需要
- ChannelFuture 的 addListener()注册一个 ChannelFutureListener 监听事件，当操作执行成功或者失败时，监听就会自动触发返回结果。
- ChannelHandler：充当了所有处理入站和出站数据的逻辑容器。
- ChannelHandler 主要用来处理各种事件，这里的事件很广泛，比如可以是连接、数据接收、异常、数据转换等。
- ChannelPipeline：为 ChannelHandler 链提供了容器，当 channel 创建时，就会被自动分配到它专属的 ChannelPipeline，这个关联是永久性的。

12.Netty 发送消息有几种方式?

Netty 有两种发送消息的方式：

- 直接写入 Channel 中，消息从 ChannelPipeline 当中尾部开始移动；
- 写入和 ChannelHandler 绑定的 ChannelHandlerContext 中，消息从ChannelPipeline 中的下一个 ChannelHandler 中移动。

13.默认情况 Netty 起多少线程？何时启动?

Netty 默认是 CPU 处理器数的两倍，bind 完之后启动。

14.了解哪几种序列化协议?

序列化（编码）是将对象序列化为二进制形式（字节数组），主要用于网络传输、数据持久化等；而反序列化（解码）则是将从网络、磁盘等读取的字节数组还原成原始对象，主要用于网络传输对象的解码，以便完成远程调用。

影响序列化性能的关键因素：序列化后的码流大小（网络带宽的占用）、序列化的性能（CPU资源占用）；是否支持跨语言（异构系统的对接和开发语言切换）。

Java默认提供的序列化：无法跨语言、序列化后的码流太大、序列化的性能差 XML，优点：人机可读性好，可指定元素或特性的名称。缺点：序列化数据只包含数据本身以及类的结构，不包括类型标识和程序集信息；只能序列化公共属性和字段；不能序列化方法；文件庞大，文件格式复杂，传输占带宽。适用场景：当做配置文件存储数据，实时数据转换。

JSON，是一种轻量级的数据交换格式，优点：兼容性高、数据格式比较简单，易于读写、序列化后数据较小，可扩展性好，兼容性好、与XML相比，其协议比较简单，解析速度比较快。缺点：数据的描述性比XML差、不适合性能要求为ms级别的情况、额外空间开销比较大。适用场景（可替代ＸＭＬ）：跨防火墙访问、可调式性要求高、基于Web browser的Ajax请求、传输数据量相对小，实时性要求相对低（例如秒级别）的服务。

Fastjson，采用一种“假定有序快速匹配”的算法。优点：接口简单易用、目前 java语言中 快的json库。缺点：过于注重快，而偏离了“标准”及功能性、代码质量不高，文档不全。适用场景：协议交互、Web输出、Android客户端

Thrift，不仅是序列化协议，还是一个RPC框架。优点：序列化后的体积小, 速度快、支持多种语言和丰富的数据类型、对于数据字段的增删具有较强的兼容性、支持二进制压缩编码。缺点：使用者较少、跨防火墙访问时，不安全、不具有可读性，调试代码时相对困难、不能与其他传输层协议共同使用（例如HTTP）、无法支持向持久层直接读写数据，即不适合做数据持久化序列化协议。适用场景：分布式系统的RPC解决方案

Avro，Hadoop的一个子项目，解决了JSON的冗长和没有IDL的问题。优点：支持丰富的数据类型、简单的动态语言结合功能、具有自我描述属性、提高了数据解析速度、快速可压缩的二进制数据形式、可以实现远程过程调用RPC、支持跨编程语言实现。缺点：对于习惯于静态类型语言的用户不直观。适用场景：在

Hadoop中做Hive、Pig和MapReduce的持久化数据格式。

Protobuf，将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的POJO对象和Protobuf相关的方法和属性。优点：序列化后码流小，性能高、结构化数据存储格式（XML JSON等）、通过标识字段的顺序，可以实现协议的前向兼容、结构化的文档更容易管理和维护。缺点：需要依赖于工具生成代码、支持的语言相对较少，官方只支持Java 、C++ 、python。适用场景：对性能要求高的RPC调用、具有良好的跨防火墙的访问属性、适合应用层对象的持久化

其它

protostuff 基于protobuf协议，但不需要配置proto文件，直接导包即可

Jboss marshaling 可以直接序列化java类， 无须实java.io.Serializable接口

Message pack 一个高效的二进制序列化格式

Hessian 采用二进制协议的轻量级remoting onhttp工具

kryo 基于protobuf协议，只支持java语言,需要注册（Registration），然后序列化（Output），反序列化（Input）

15.如何选择序列化协议?

具体场景

对于公司间的系统调用，如果性能要求在100ms以上的服务，基于XML的SOAP 协议是一个值得考虑的方案。

基于Web browser的Ajax，以及Mobile app与服务端之间的通讯，JSON协议是首选。对于性能要求不太高，或者以动态类型语言为主，或者传输数据载荷很小的的运用场景，JSON也是非常不错的选择。

对于调试环境比较恶劣的场景，采用JSON或XML能够极大的提高调试效率，降低系统开发成本。当对性能和简洁性有极高要求的场景，Protobuf，Thrift，Avro之间具有一定的竞争关系。

对于T级别的数据的持久化应用场景，Protobuf和Avro是首要选择。如果持久化后的数据存储在hadoop子项目里，Avro会是更好的选择。

对于持久层非Hadoop项目，以静态类型语言为主的应用场景，Protobuf会更符合静态类型语言工程师的开发习惯。由于Avro的设计理念偏向于动态类型语言，对于动态语言为主的应用场景，Avro是更好的选择。

如果需要提供一个完整的RPC解决方案，Thrift是一个好的选择。

如果序列化之后需要支持不同的传输层协议，或者需要跨防火墙访问的高性能场景，Protobuf可以优先考虑。

protobuf的数据类型有多种：bool、double、float、int32、int64、string、 bytes、enum、message。protobuf的限定符：required: 必须赋值，不能为

空、optional:字段可以赋值，也可以不赋值、repeated: 该字段可以重复任意次数（包括0次）、枚举；只能用指定的常量集中的一个值作为其值；

protobuf的基本规则：每个消息中必须至少留有一个required类型的字段、包含0个或多个optional类型的字段；repeated表示的字段可以包含0个或多个数据；[1,15]之内的标识号在编码的时候会占用一个字节（常用），[16,2047]之内的标识号则占用2个字节，标识号一定不能重复、使用消息类型，也可以将消息嵌套任意多层，可用嵌套消息类型来代替组。

protobuf的消息升级原则：不要更改任何已有的字段的数值标识；不能移除已经存在的required字段，optional和repeated类型的字段可以被移除，但要保留标号不能被重用。新添加的字段必须是optional或repeated。因为旧版本程序无法读取或写入新增的required限定符的字段。

编译器为每一个消息类型生成了一个.java文件，以及一个特殊的Builder类（该类是用来创建消息类接口的）。如：UserProto.User.Builder builder =

UserProto.User.newBuilder();builder.build()；

Netty中的使用：ProtobufVarint32FrameDecoder 是用于处理半包消息的解码类；ProtobufDecoder(UserProto.User.getDefaultInstance())这是创建的 UserProto.java文件中的解码类；ProtobufVarint32LengthFieldPrepender 对protobuf协议的消息头上加上一个长度为32的整形字段，用于标志这个消息的长度的类；ProtobufEncoder 是编码类将StringBuilder转换为ByteBuf类型：copiedBuffer()方法

16.Netty 支持哪些心跳类型设置?

- readerIdleTime：为读超时时间（即测试端一定时间内未接受到被测试端消息）。
- writerIdleTime：为写超时时间（即测试端一定时间内向被测试端发送消息）。
- allIdleTime：所有类型的超时时间。

17.Netty 和 Tomcat 的区别?

- 作用不同：Tomcat 是 Servlet 容器，可以视为 Web 服务器，而 Netty 是异步事件驱动的网络应用程序框架和工具用于简化网络编程，例如TCP和UDP套接字服务器。
- 协议不同：Tomcat 是基于 http 协议的 Web 服务器，而 Netty 能通过编程自定义各种协议，因为 Netty 本身自己能编码/解码字节流，所有 Netty 可以实现， HTTP 服务器、FTP 服务器、UDP 服务器、RPC 服务器、WebSocket 服务器、 Redis 的 Proxy 服务器、MySQL 的 Proxy 服务器等等。

18.NIOEventLoopGroup源码?

NioEventLoopGroup(其实是MultithreadEventExecutorGroup) 内部维护一个类型为 EventExecutor children [], 默认大小是处理器核数 * 2, 这样就构成了一个线程池，初始化EventExecutor时NioEventLoopGroup重载newChild方法，所以children元素的实际类型为NioEventLoop。

线程启动时调用SingleThreadEventExecutor的构造方法，执行NioEventLoop 类的run方法，首先会调用hasTasks()方法判断当前taskQueue是否有元素。如果taskQueue中有元素，执行 selectNow() 方法， 终执行 selector.selectNow()，该方法会立即返回。如果taskQueue没有元素，执行 select(oldWakenUp) 方法

select (oldWakenUp) 方法解决了 Nio 中的 bug，selectCnt 用来记录 selector.select方法的执行次数和标识是否执行过selector.selectNow()，若触发了epoll的空轮询bug，则会反复执行selector.select(timeoutMillis)，变量 selectCnt 会逐渐变大，当selectCnt 达到阈值（默认512），则执行 rebuildSelector方法，进行selector重建，解决cpu占用100%的bug。

rebuildSelector方法先通过openSelector方法创建一个新的selector。然后将

old selector的selectionKey执行cancel。 后将old selector的channel重新注册到新的selector中。rebuild后，需要重新执行方法selectNow，检查是否有已ready的selectionKey。

接下来调用processSelectedKeys 方法（处理I/O任务），当selectedKeys != null时，调用processSelectedKeysOptimized方法，迭代 selectedKeys 获取就绪的 IO 事件的selectkey存放在数组selectedKeys中, 然后为每个事件都调用 processSelectedKey 来处理它，processSelectedKey 中分别处理OP_READ； OP_WRITE；OP_CONNECT事件。

后调用runAllTasks方法（非IO任务），该方法首先会调用

fetchFromScheduledTaskQueue方法，把scheduledTaskQueue中已经超过延迟执行时间的任务移到taskQueue中等待被执行，然后依次从taskQueue中取任务执行，每执行64个任务，进行耗时检查，如果已执行时间超过预先设定的执行时间，则停止执行非IO任务，避免非IO任务太多，影响IO任务的执行。每个NioEventLoop对应一个线程和一个Selector，NioServerSocketChannel 会主动注册到某一个NioEventLoop的Selector上，NioEventLoop负责事件轮询。

Outbound 事件都是请求事件, 发起者是 Channel，处理者是 unsafe，通过 Outbound 事件进行通知，传播方向是 tail到head。Inbound 事件发起者是 unsafe，事件的处理者是 Channel, 是通知事件，传播方向是从头到尾。内存管理机制，首先会预申请一大块内存Arena，Arena由许多Chunk组成，而每个Chunk默认由2048个page组成。Chunk通过AVL树的形式组织Page，每个叶子节点表示一个Page，而中间节点表示内存区域，节点自己记录它在整个 Arena中的偏移地址。当区域被分配出去后，中间节点上的标记位会被标记，这样就表示这个中间节点以下的所有节点都已被分配了。大于8k的内存分配在

poolChunkList中，而PoolSubpage用于分配小于8k的内存，它会把一个page 分割成多段，进行内存分配。

ByteBuf的特点：支持自动扩容（4M），保证put方法不会抛出异常、通过内置的复合缓冲类型，实现零拷贝（zero-copy）；不需要调用flip()来切换读/写模

式，读取和写入索引分开；方法链；引用计数基于AtomicIntegerFieldUpdater 用于内存回收；PooledByteBuf采用二叉树来实现一个内存池，集中管理内存的分配和释放，不用每次使用都新建一个缓冲区对象。UnpooledHeapByteBuf每次都会新建一个缓冲区对象。

Netty

为什么选择 Netty

> 1) API使用简单，开发门槛低；

> 2) 功能强大，预置了多种编解码功能，支持多种主流协议；

> 3) 定制能力强，可以通过 ChannelHandler 对通信框架进行灵活的扩展；

> 4) 性能高，通过与其它业界主流的NIO框架对比，Netty的综合性能最优；

> 5) 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为

NIO的BUG而烦恼；

> 6) 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会被加入；

> 7) 经历了大规模的商业应用考验，质量已经得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它可以完全满足不同行业的商业应用。

> 正是因为这些优点，Netty逐渐成为Java NIO编程的首选框架。

说说业务中，Netty 的使用场景

> 构建高性能、低时延的各种Java中间件，例如MQ、分布式服务框架、ESB消息总线等，

Netty主要作为基础通信框架提供高性能、低时延的通信服务；

> 公有或者私有协议栈的基础通信框架，例如可以基于Netty构建异步、高性能的

WebSocket协议栈；

> 各领域应用，例如大数据、游戏等，Netty作为高性能的通信框架用于内部各模块的数据分发、传输和汇总等，实现模块之间高性能通信。

原生的 NIO 在 JDK 1.7 版本存在 epoll bug

> 它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK 1.6版本的update18修复了该问题，但是直到JDK 1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有得到根本性解决。

什么是 TCP 粘包/拆包

> 1、要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包。

> 2、待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。

> 3、要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。

> 4、接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。

TCP 粘包/拆包的解决办法

> 1、发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。

> 2、发送端将每个数据包封装为固定长度（不够的可以通过补0填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。

> 3、可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

Netty 线程模型

> 首先，Netty使用EventLoop来处理连接上的读写事件，而一个连接上的所有请求都保证在一个EventLoop中被处理，一个EventLoop中只有一个Thread，所以也就实现了一个连接上的所有事件只会在一个线程中被执行。一个EventLoopGroup包含多个EventLoop，可以把一个EventLoop当做是Reactor线程模型中的一个线程，而一个EventLoopGroup类似于一个ExecutorService

说说 Netty 的零拷贝

> “零拷贝”是指计算机操作的过程中，CPU不需要为数据在内存之间的拷贝消耗资源。而

它通常是指计算机在网络上发送文件时，不需要将文件内容拷贝到用户空间（User

Space）而直接在内核空间（Kernel Space）中传输到网络的方式。

Netty 内部执行流程

> 1. Netty的接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行

Socket读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP

BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入

Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

> 2. Netty提供了组合Buffer对象，可以聚合多个ByteBuffer对象，用户可以像操作一

个Buffer那样方便的对组合Buffer进行操作，避免了传统通过内存拷贝的方式将几个小

Buffer合并成一个大的Buffer。

> 3. Netty的文件传输采用了transferTo方法，它可以直接将文件缓冲区的数据发送到

Netty经典面试题

BIO、NIO 和 AIO 的区别?

BIO：一个连接一个线程，客户端有连接请求时服务器端就需要启动一个线程进行处理。线程开销大。

伪异步 IO：将请求连接放入线程池，一对多，但线程还是很宝贵的资源。

NIO：一个请求一个线程，但客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。

AIO：一个有效请求一个线程，客户端的 I/O 请求都是由 OS 先完成了再通知服务器应用去启动线程进行处理，

BIO 是面向流的，NIO 是面向缓冲区的；BIO 的各种流是阻塞的。而 NIO 是非阻塞的；BIO 的 Stream 是单向的，而 NIO 的 channel 是双向的。

NIO 的特点：事件驱动模型、单线程处理多任务、非阻塞 I/O，I/O 读写不再阻塞，而是返回 0、基于 block 的传输比基于流的传输更高效、更高级的 IO 函数 zero-copy、IO 多路复用大大提高了 Java 网络应用的可伸缩性和实用性。基于 Reactor 线程模型。

在 Reactor 模式中，事件分发器等待某个事件或者可应用或个操作的状态发生，事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。如在 Reactor 中实现读：注册读就绪事件和相应的事件处理器、事件分发器等待事件、事件到来，激活分发器，分发器调用事件对应的处理器、事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

NIO 的组成?

Buffer：与 Channel 进行交互，数据是从 Channel 读入缓冲区，从缓冲区写入 Channel 中的

flip 方法： 反转此缓冲区，将 position 给 limit，然后将 position 置为 0，其实就是切换读写模式

clear 方法： 清除此缓冲区，将 position 置为 0，把 capacity 的值给 limit。

rewind 方法： 重绕此缓冲区，将 position 置为 0

DirectByteBuffer 可减少一次系统空间到用户空间的拷贝。但 Buffer 创建和销毁的成本更

高，不可控，通常会用内存池来提高性能。直接缓冲区主要分配给那些易受基础系统的本机 I/O 操作影响的大型、持久的缓冲区。如果数据量比较小的中小应用情况下，可以考虑使用 heapBuffer，由 JVM 进行管理。

Channel：表示 IO 源与目标打开的连接，是双向的，但不能直接访问数据，只能与 Buffer 进行交互。通过源码可知，FileChannel 的 read 方法和 write 方法都导致数据复制了两次!

Selector 可使一个单独的线程管理多个 Channel，open 方法可创建 Selector，register 方法向多路复用器器注册通道，可以监听的事件类型：读、写、连接、accept。注册事件后会产

生一个 SelectionKey：它表示 SelectableChannel 和 Selector 之间的注册关系，wakeup 方法：使尚未返回的第一个选择操作立即返回，唤醒的原因是：注册了新的 channel 或者事件；channel 关闭，取消注册；优先级更高的事件触发（如定时器事件），希望及时处理。

Selector 在 Linux 的实现类是 EPollSelectorImpl，委托给 EPollArrayWrapper 实现，其中三个 native 方法是对 epoll 的封装，而 EPollSelectorImpl. implRegister 方法，通过调用 epoll_ctl 向 epoll 实例中注册事件，还将注册的文件描述符(fd)与 SelectionKey 的对应关系添加到 fdToKey 中，这个 map 维护了文件描述符与 SelectionKey 的映射。

fdToKey 有时会变得非常大，因为注册到 Selector 上的 Channel 非常多（百万连接）；过期或失效的 Channel 没有及时关闭。fdToKey 总是串行读取的，而读取是在 select 方法中进行的，该方法是非线程安全的。

Pipe：两个线程之间的单向数据连接，数据会被写到 sink 通道，从 source 通道读取

NIO 的服务端建立过程：Selector.open()：打开一个 Selector；ServerSocketChannel.open()：创建服务端的 Channel；bind()：绑定到某个端口上。并配置非阻塞模式；register()：注册

Channel 和关注的事件到 Selector 上；select()轮询拿到已经就绪的事件

Netty 的特点?

一个高性能、异步事件驱动的 NIO 框架，它提供了对 TCP、UDP 和文件传输的支持使用更高效的 socket 底层，对 epoll 空轮询引起的 cpu 占用飙升在内部进行了处理，避免了直接使用 NIO 的陷阱，简化了 NIO 的处理方式。

采用多种 decoder/encoder 支持，对 TCP 粘包/分包进行自动化处理

可使用接受/处理线程池，提高连接效率，对重连、心跳检测的简单支持

可配置 IO 线程数、TCP 参数， TCP 接收和发送缓冲区使用直接内存代替堆内存，通过内存池的方式循环利用 ByteBuf

通过引用计数器及时申请释放不再引用的对象，降低了 GC 频率使用单线程串行化的方式，高效的 Reactor 线程模型大量使用了 volitale、使用了 CAS 和原子类、线程安全类的使用、读写锁的使用

Netty 的线程模型?

Netty 通过 Reactor 模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，

boss 线程池和 work 线程池，其中 boss 线程池的线程负责处理请求的 accept 事件，当接收到 accept 事件的请求时，把对应的 socket 封装到一个 NioSocketChannel 中，并交给 work

线程池，其中 work 线程池负责请求的 read 和 write 事件，由对应的 Handler 处理。

单线程模型：所有 I/O 操作都由一个线程完成，即多路复用、事件分发和处理都是在一个 Reactor 线程上完成的。既要接收客户端的连接请求,向服务端发起连接，又要发送/读取请求或应答/响应消息。一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，速度

慢，若线程进入死循环，整个程序不可用，对于高负载、大并发的应用场景不合适。

多线程模型：有一个 NIO 线程（Acceptor） 只负责监听服务端，接收客户端的 TCP 连接请求；NIO 线程池负责网络 IO 的操作，即消息的读取、解码、编码和发送；1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，这是为了防止发生并发操作问题。但在并发百万客户端连接或需要安全认证时，一个 Acceptor 线程可能会存在性能不足问题。

主从多线程模型：Acceptor 线程用于绑定监听端口，接收客户端连接，将 SocketChannel 从主线程池的 Reactor 线程的多路复用器上移除，重新注册到 Sub 线程池的线程上，用于处理 I/O 的读写等操作，从而保证 mainReactor 只负责接入认证、握手等操作；

TCP 粘包/拆包的原因及解决方法?

TCP 是以流的方式来处理数据，一个完整的包可能会被 TCP 拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送。

TCP 粘包/分包的原因：

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象；进行 MSS 大小的 TCP 分段，当 TCP 报文长度-TCP 头部长度>MSS 的时候将发生拆包以太网帧的 payload（净荷）大于 MTU（1500 字节）进行 ip 分片。

解决方法

消息定长：FixedLengthFrameDecoder 类

包尾增加特殊字符分割：行分隔符类：LineBasedFrameDecoder 或自定义分隔符类：

DelimiterBasedFrameDecoder

将消息分为消息头和消息体：LengthFieldBasedFrameDecoder 类。分为有头部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘包。

了解哪几种序列化协议?

序列化（编码）是将对象序列化为二进制形式（字节数组），主要用于网络传输、数据持久化等；而反序列化（解码）则是将从网络、磁盘等读取的字节数组还原成原始对象，主要用于网络传输对象的解码，以便完成远程调用。

影响序列化性能的关键因素：序列化后的码流大小（网络带宽的占用）、序列化的性能

（CPU 资源占用）；是否支持跨语言（异构系统的对接和开发语言切换）。

Java 默认提供的序列化：无法跨语言、序列化后的码流太大、序列化的性能差

XML，优点：人机可读性好，可指定元素或特性的名称。缺点：序列化数据只包含数据本身以及类的结构，不包括类型标识和程序集信息；只能序列化公共属性和字段；不能序列化方法；文件庞大，文件格式复杂，传输占带宽。适用场景：当做配置文件存储数据，实时数据转换。

JSON，是一种轻量级的数据交换格式，优点：兼容性高、数据格式比较简单，易于读写、序列化后数据较小，可扩展性好，兼容性好、与 XML 相比，其协议比较简单，解析速度比较快。缺点：数据的描述性比 XML 差、不适合性能要求为 ms 级别的情况、额外空间开销比较大。适用场景（可替代ＸＭＬ）：跨防火墙访问、可调式性要求高、基于 Web

browser 的 Ajax 请求、传输数据量相对小，实时性要求相对低（例如秒级别）的服务。

Fastjson，采用一种“假定有序快速匹配”的算法。优点：接口简单易用、目前 java 语言中最快的 json 库。缺点：过于注重快，而偏离了“标准”及功能性、代码质量不高，文档不全。适用场景：协议交互、Web 输出、Android 客户端 Thrift，不仅是序列化协议，还是一个 RPC 框架。优点：序列化后的体积小, 速度快、支持多种语言和丰富的数据类型、对于数据字段的增删具有较强的兼容性、支持二进制压缩编码。缺点：使用者较少、跨防火墙访问时，不安全、不具有可读性，调试代码时相对困难、不能与其他传输层协议共同使用（例如 HTTP）、无法支持向持久层直接读写数据，即

不适合做数据持久化序列化协议。适用场景：分布式系统的 RPC 解决方案

Avro，Hadoop 的一个子项目，解决了 JSON 的冗长和没有 IDL 的问题。优点：支持丰富的数据类型、简单的动态语言结合功能、具有自我描述属性、提高了数据解析速度、快速可压缩的二进制数据形式、可以实现远程过程调用 RPC、支持跨编程语言实现。缺点：对于习惯于静态类型语言的用户不直观。适用场景：在 Hadoop 中做 Hive、Pig 和 MapReduce 的持久化数据格式。

Protobuf，将数据结构以.proto 文件进行描述，通过代码生成工具可以生成对应数据结构的 POJO 对象和 Protobuf 相关的方法和属性。优点：序列化后码流小，性能高、结构化数据存储格式（XML JSON 等）、通过标识字段的顺序，可以实现协议的前向兼容、结构化的文档更容易管理和维护。缺点：需要依赖于工具生成代码、支持的语言相对较少，官方只支持

Java 、C++ 、python。适用场景：对性能要求高的 RPC 调用、具有良好的跨防火墙的访问属性、适合应用层对象的持久化

其它

protostuff 基于 protobuf 协议，但不需要配置 proto 文件，直接导包即可

Jboss marshaling 可以直接序列化 java 类， 无须实 java.io.Serializable 接口

Message pack 一个高效的二进制序列化格式

Hessian 采用二进制协议的轻量级 remoting onhttp 工具

kryo　　基于 protobuf 协议，只支持 java 语言,需要注册（Registration），然后序列化

（Output），反序列化（Input）

如何选择序列化协议?

具体场景

对于公司间的系统调用，如果性能要求在 100ms 以上的服务，基于 XML 的 SOAP 协议是一个值得考虑的方案。

基于 Web browser 的 Ajax，以及 Mobile app 与服务端之间的通讯，JSON 协议是首选。对于性能要求不太高，或者以动态类型语言为主，或者传输数据载荷很小的的运用场景，JSON 也是非常不错的选择。

对于调试环境比较恶劣的场景，采用 JSON 或 XML 能够极大的提高调试效率，降低系统开发成本。

当对性能和简洁性有极高要求的场景，Protobuf，Thrift，Avro 之间具有一定的竞争关系。对于 T 级别的数据的持久化应用场景，Protobuf 和 Avro 是首要选择。如果持久化后的数据存储在 hadoop 子项目里，Avro 会是更好的选择。

对于持久层非 Hadoop 项目，以静态类型语言为主的应用场景，Protobuf 会更符合静态类型语言工程师的开发习惯。由于 Avro 的设计理念偏向于动态类型语言，对于动态语言为主的应用场景，Avro 是更好的选择。

如果需要提供一个完整的 RPC 解决方案，Thrift 是一个好的选择。

如果序列化之后需要支持不同的传输层协议，或者需要跨防火墙访问的高性能场景，

Protobuf 可以优先考虑。

protobuf 的数据类型有多种：bool、double、float、int32、int64、string、bytes、enum、 message。protobuf 的限定符：required: 必须赋值，不能为空、optional:字段可以赋值，也可以不赋值、repeated: 该字段可以重复任意次数（包括 0 次）、枚举；只能用指定的常量集中的一个值作为其值；

protobuf 的基本规则：每个消息中必须至少留有一个 required 类型的字段、包含 0 个或多个 optional 类型的字段；repeated 表示的字段可以包含 0 个或多个数据；[1,15]之内的标识号在编码的时候会占用一个字节（常用），[16,2047]之内的标识号则占用 2 个字节，标识号一定不能重复、使用消息类型，也可以将消息嵌套任意多层，可用嵌套消息类型来代替组。

protobuf 的消息升级原则：不要更改任何已有的字段的数值标识；不能移除已经存在的 required 字段，optional 和 repeated 类型的字段可以被移除，但要保留标号不能被重用。新添加的字段必须是 optional 或 repeated。因为旧版本程序无法读取或写入新增的 required 限定符的字段。

编译器为每一个消息类型生成了一个.java 文件，以及一个特殊的 Builder 类（该类是用来创建消息类接口的）。如：UserProto.User.Builder builder =

UserProto.User.newBuilder();builder.build()；

Netty 中的使用：ProtobufVarint32FrameDecoder 是用于处理半包消息的解码类；

ProtobufDecoder(UserProto.User.getDefaultInstance())这是创建的 UserProto.java 文件中的解码类；ProtobufVarint32LengthFieldPrepender 对 protobuf 协议的消息头上加上一个长度为

32 的整形字段，用于标志这个消息的长度的类；ProtobufEncoder 是编码类

将 StringBuilder 转换为 ByteBuf 类型：copiedBuffer()方法

Netty 的零拷贝实现?

Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。堆内存多了一次内存拷贝，JVM 会将堆内存

Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。ByteBuffer 由 ChannelConfig 分配，

而 ChannelConfig 创建 ByteBufAllocator 默认使用 Direct Buffer

CompositeByteBuf 类可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf, 避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。addComponents 方法将 header 与 body 合并为一个逻辑上的 ByteBuf, 这两个 ByteBuf 在 CompositeByteBuf 内部都是单独存在的, CompositeByteBuf 只是逻辑上是一个整体

通过 FileRegion 包装的 FileChannel.tranferTo 方法 实现文件传输, 可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

通过 wrap 方法, 我们可以将 byte[] 数组、ByteBuf、ByteBuffer 等包装成一个 Netty

ByteBuf 对象, 进而避免了拷贝操作。

Selector BUG：若 Selector 的轮询结果为空，也没有 wakeup 或新消息处理，则发生空轮询，CPU 使用率 100%，

Netty 的解决办法：对 Selector 的 select 操作周期进行统计，每完成一次空的 select 操作进行一次计数，若在某个周期内连续发生 N 次空轮询，则触发了 epoll 死循环 bug。重建

Selector，判断是否是其他线程发起的重建请求，若不是则将原 SocketChannel 从旧的

Selector 上去除注册，重新注册到新的 Selector 上，并将原来的 Selector 关闭。

Netty 的高性能表现在哪些方面?

心跳，对服务端：会定时清除闲置会话 inactive(netty5)，对客户端:用来检测会话是否断

开，是否重来，检测网络延迟，其中 idleStateHandler 类 用来检测会话状态

串行无锁化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。表面上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，

这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。

可靠性，链路有效性检测：链路空闲检测机制，读/写空闲超时机制；内存保护机制：通过内存池重用 ByteBuf;ByteBuf 的解码保护；优雅停机：不再接收新消息、退出前的预处理操作、资源的释放操作。

Netty 安全性：支持的安全协议：SSL V2 和 V3，TLS，SSL 单向认证、双向认证和第三方 CA 认证。

高效并发编程的体现：volatile 的大量、正确使用；CAS 和原子类的广泛使用；线程安全容器的使用；通过读写锁提升并发性能。IO 通信性能三原则：传输（AIO）、协议（Http）、线程（主从多线程）

流量整型的作用（变压器）：防止由于上下游网元性能不均衡导致下游网元被压垮，业务

流中断；防止由于通信模块接受消息过快，后端业务线程处理不及时导致撑死问题。

TCP 参数配置：SO_RCVBUF 和 SO_SNDBUF：通常建议值为 128K 或者 256K；

SO_TCPNODELAY：NAGLE 算法通过将缓冲区内的小封包自动相连，组成较大的封包，阻止大量小封包的发送阻塞网络，从而提高网络应用效率。但是对于时延敏感的应用场景需要关闭该优化算法；

NIOEventLoopGroup 源码?

NioEventLoopGroup(其实是 MultithreadEventExecutorGroup)　　内部维护一个类型为

EventExecutor children [], 默认大小是处理器核数 * 2, 这样就构成了一个线程池，初始化 EventExecutor 时 NioEventLoopGroup 重载 newChild 方法，所以 children 元素的实际类型为

NioEventLoop。

线程启动时调用 SingleThreadEventExecutor 的构造方法，执行 NioEventLoop 类的 run 方法，首先会调用 hasTasks()方法判断当前 taskQueue 是否有元素。如果 taskQueue 中有元素，执行 selectNow() 方法，最终执行 selector.selectNow()，该方法会立即返回。如果 taskQueue 没有元素，执行 select(oldWakenUp) 方法

select (oldWakenUp) 方法解决了 Nio 中的 bug，selectCnt 用来记录 selector.select 方法的执行次数和标识是否执行过 selector.selectNow()，若触发了 epoll 的空轮询 bug，则会反复

执行 selector.select(timeoutMillis)，变量 selectCnt 会逐渐变大，当 selectCnt 达到阈值（默认 512），则执行 rebuildSelector 方法，进行 selector 重建，解决 cpu 占用 100%的 bug。

rebuildSelector 方法先通过 openSelector 方法创建一个新的 selector。然后将 old selector 的 selectionKey 执行 cancel。最后将 old　selector 的 channel 重新注册到新的 selector 中。 rebuild 后，需要重新执行方法 selectNow，检查是否有已 ready 的 selectionKey。

接下来调用 processSelectedKeys 方法（处理 I/O 任务），当 selectedKeys != null 时，调用 processSelectedKeysOptimized 方法，迭代 selectedKeys 获取就绪的 IO 事件的 selectkey 存放在数组 selectedKeys 中, 然后为每个事件都调用 processSelectedKey 来处理它， processSelectedKey 中分别处理 OP_READ；OP_WRITE；OP_CONNECT 事件。

最后调用 runAllTasks 方法（非 IO 任务），该方法首先会调用 fetchFromScheduledTaskQueue 方法，把 scheduledTaskQueue 中已经超过延迟执行时间的任务移到 taskQueue 中等待被执行，然后依次从 taskQueue 中取任务执行，每执行 64 个任务，进行耗时检查，如果已执行时间超过预先设定的执行时间，则停止执行非 IO 任务，避免非 IO 任务太多，影响 IO 任务的执行。

每个 NioEventLoop 对应一个线程和一个 Selector，NioServerSocketChannel 会主动注册到某一个 NioEventLoop 的 Selector 上，NioEventLoop 负责事件轮询。

Outbound 事件都是请求事件, 发起者是 Channel，处理者是 unsafe，通过 Outbound 事件进行通知，传播方向是 tail 到 head。Inbound 事件发起者是 unsafe，事件的处理者是 Channel, 是通知事件，传播方向是从头到尾。

内存管理机制，首先会预申请一大块内存 Arena，Arena 由许多 Chunk 组成，而每个 Chunk 默认由 2048 个 page 组成。Chunk 通过 AVL 树的形式组织 Page，每个叶子节点表示一个 Page，而中间节点表示内存区域，节点自己记录它在整个 Arena 中的偏移地址。当区域被分配出去后，中间节点上的标记位会被标记，这样就表示这个中间节点以下的所有节点都已被分配了。大于 8k 的内存分配在 poolChunkList 中，而 PoolSubpage 用于分配小于 8k 的内存，它会把一个 page 分割成多段，进行内存分配。

ByteBuf 的特点：支持自动扩容（4M），保证 put 方法不会抛出异常、通过内置的复合缓冲类型，实现零拷贝（zero-copy）；不需要调用 flip()来切换读/写模式，读取和写入索引分开；方法链；引用计数基于 AtomicIntegerFieldUpdater 用于内存回收；PooledByteBuf 采用二叉树来实现一个内存池，集中管理内存的分配和释放，不用每次使用都新建一个缓冲区对象。UnpooledHeapByteBuf 每次都会新建一个缓冲区对象。

Netty 原理

Netty 是一个高性能、异步事件驱动的 NIO 框架，基于 JAVA NIO 提供的 API 实现。它提供了对

TCP、UDP 和文件传输的支持，作为一个异步 NIO 框架，Netty 的所有 IO 操作都是异步非阻塞的，通过 Future-Listener 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果。

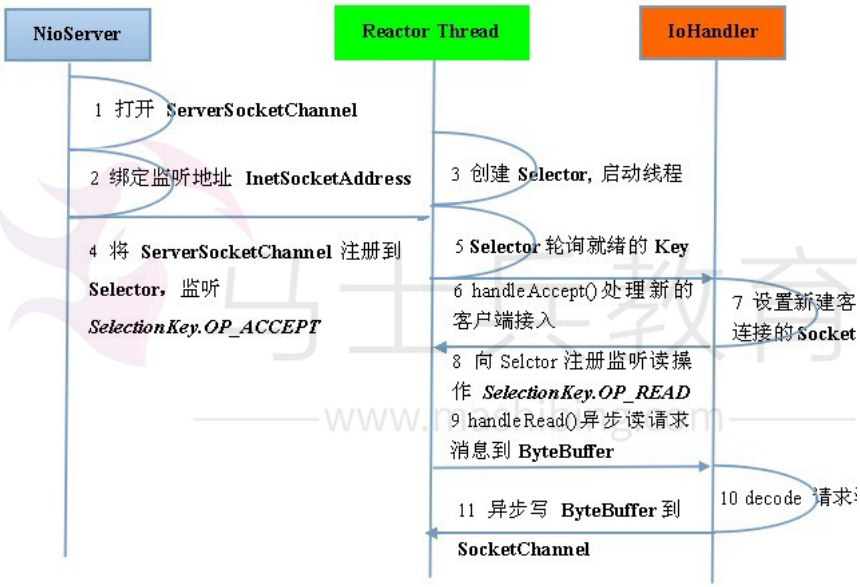
Netty 高性能

在IO编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者IO多路复用技术进行处理。IO多路复用技术通过把多个IO的阻塞复用到同一个select的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型比，I/O 多路复用的最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程的运行，降低了系统的维护工作量，节省了系统资源。

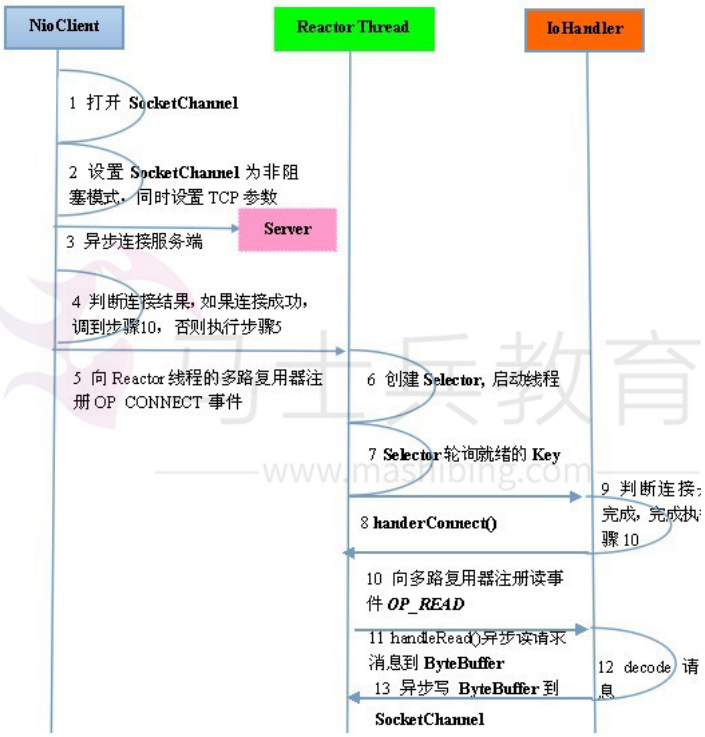
与Socket类和ServerSocket类相对应，NIO也提供了SocketChannel和ServerSocketChannel 两种不同的套接字通道实现。

多路复用通讯方式

Netty 架构按照 Reactor 模式设计和实现，它的服务端通信序列图如下：



客户端通信序列图如下：



Netty 的 IO 线程 NioEventLoop 由于聚合了多路复用器 Selector，可以同时并发处理成百上千个客户端 Channel，由于读写操作都是非阻塞的，这就可以充分提升 IO 线程的运行效率，避免由于频繁 IO 阻塞导致的线程挂起。

异步通讯NIO

由于 Netty 采用了异步通信模式，一个 IO 线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 IO 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

零拷贝（DIRECT BUFFERS 使用堆外直接内存）

1. Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行 Socket 读写，

JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

2. Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的

Buffer。

3. Netty的文件传输采用了transferTo方法，它可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环 write 方式导致的内存拷贝问题

内存池（基于内存池的缓冲区重用机制）

随着 JVM 虚拟机和 JIT 即时编译技术的发展，对象的分配和回收是个非常轻量级的工作。但是对于缓冲区 Buffer，情况却稍有不同，特别是对于堆外直接内存的分配和回收，是一件耗时的操作。为了尽量重用缓冲区，Netty 提供了基于内存池的缓冲区重用机制。

高效的 Reactor 线程模型

常用的 Reactor 线程模型有三种，Reactor 单线程模型, Reactor 多线程模型, 主从 Reactor 多线程模型。

Reactor单线程模型

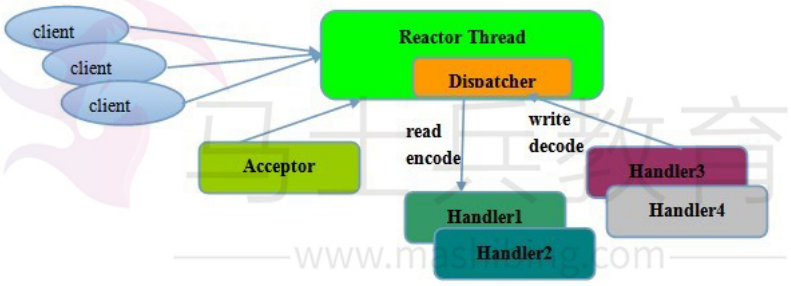
Reactor 单线程模型，指的是所有的 IO 操作都在同一个 NIO 线程上面完成，NIO 线程的职责如下：

1) 作为 NIO 服务端，接收客户端的 TCP 连接；

2) 作为 NIO 客户端，向服务端发起 TCP 连接；

3) 读取通信对端的请求或者应答消息；

4) 向通信对端发送消息请求或者应答消息。

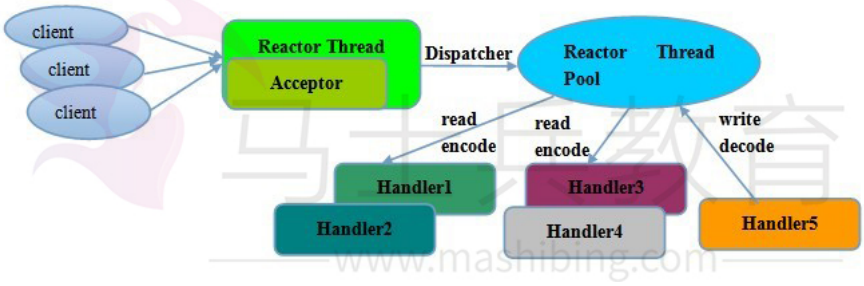


由于 Reactor 模式使用的是异步非阻塞 IO，所有的 IO 操作都不会导致阻塞，理论上一个线程可以独立处理所有 IO 相关的操作。从架构层面看，一个 NIO 线程确实可以完成其承担的职责。例如，通过

Acceptor 接收客户端的 TCP 连接请求消息，链路建立成功之后，通过 Dispatch 将对应的 ByteBuffer 派发到指定的 Handler 上进行消息解码。用户 Handler 可以通过 NIO 线程将消息发送给客户端。

Reactor多线程模型

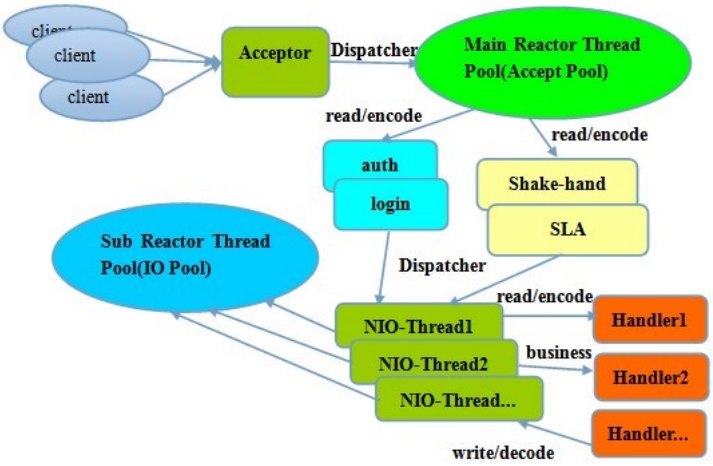
Rector 多线程模型与单线程模型最大的区别就是有一组 NIO 线程处理 IO 操作。 有专门一个NIO 线程-Acceptor 线程用于监听服务端，接收客户端的 TCP 连接请求； 网络 IO 操作-读、写等由一个 NIO 线程池负责，线程池可以采用标准的 JDK 线程池实现，它包含一个任务队列和 N 个可用的线程，由这些 NIO 线程负责消息的读取、解码、编码和发送；



主从**Reactor**多线程模型

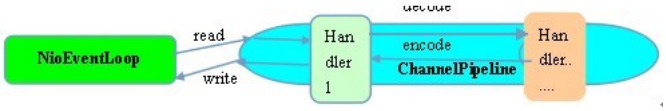
服务端用于接收客户端连接的不再是个 1 个单独的 NIO 线程，而是一个独立的 NIO 线程池。

Acceptor 接收到客户端 TCP 连接请求处理完成后（可能包含接入认证等），将新创建的SocketChannel 注册到 IO 线程池（sub reactor 线程池）的某个 IO 线程上，由它负责 SocketChannel 的读写和编解码工作。Acceptor 线程池仅仅只用于客户端的登陆、握手和安全认证，一旦链路建立成功，就将链路注册到后端 subReactor 线程池的 IO 线程上，由 IO 线程负责后续的 IO 操作。



无锁设计、线程绑定

Netty 采用了串行无锁化设计，在 IO 线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列多个工作线程模型性能更优。



高性能的序列化框架

Netty 默认提供了对 Google Protobuf 的支持，通过扩展 Netty 的编解码接口，用户可以实现其它的高性能序列化框架，例如 Thrift 的压缩二进制编解码框架。

1. SO_RCVBUF 和 SO_SNDBUF：通常建议值为 128K 或者 256K。

小包封大包，防止网络阻塞

2. SO_TCPNODELAY：NAGLE 算法通过将缓冲区内的小封包自动相连，组成较大的封包，阻止大量小封包的发送阻塞网络，从而提高网络应用效率。但是对于时延敏感的应用场景需要关闭该优化算法。

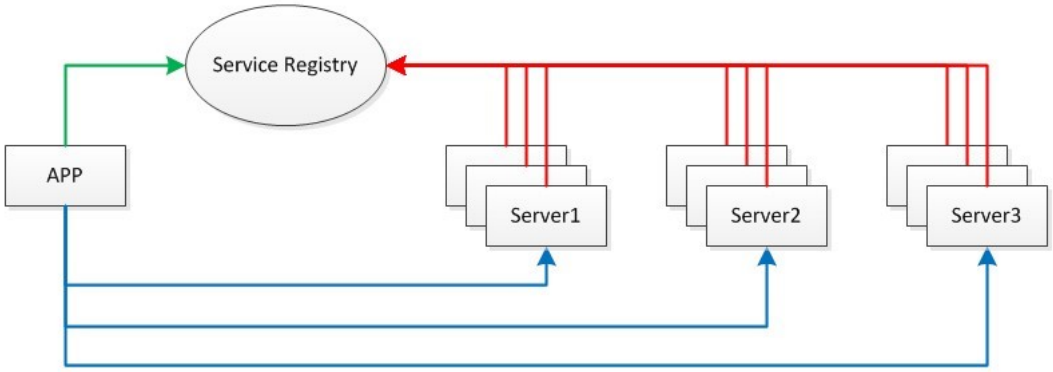
软中断**Hash** 值和**CPU**绑定

3. 软中断：开启 RPS 后可以实现软中断，提升网络吞吐量。RPS 根据数据包的源地址，目的地址以及目的和源端口，计算出一个 hash 值，然后根据这个 hash 值来选择软中断运行的 cpu，从上层来看，也就是说将每个连接和 cpu 绑定，并通过这个 hash 值，来均衡软中断在多个 cpu 上，提升网络并行处理性能。

Netty RPC 实现

概念

RPC，即 Remote Procedure Call（远程过程调用），调用远程计算机上的服务，就像调用本地服务一样。RPC 可以很好的解耦系统，如 WebService 就是一种基于 Http 协议的 RPC。这个 RPC 整体框架如下：



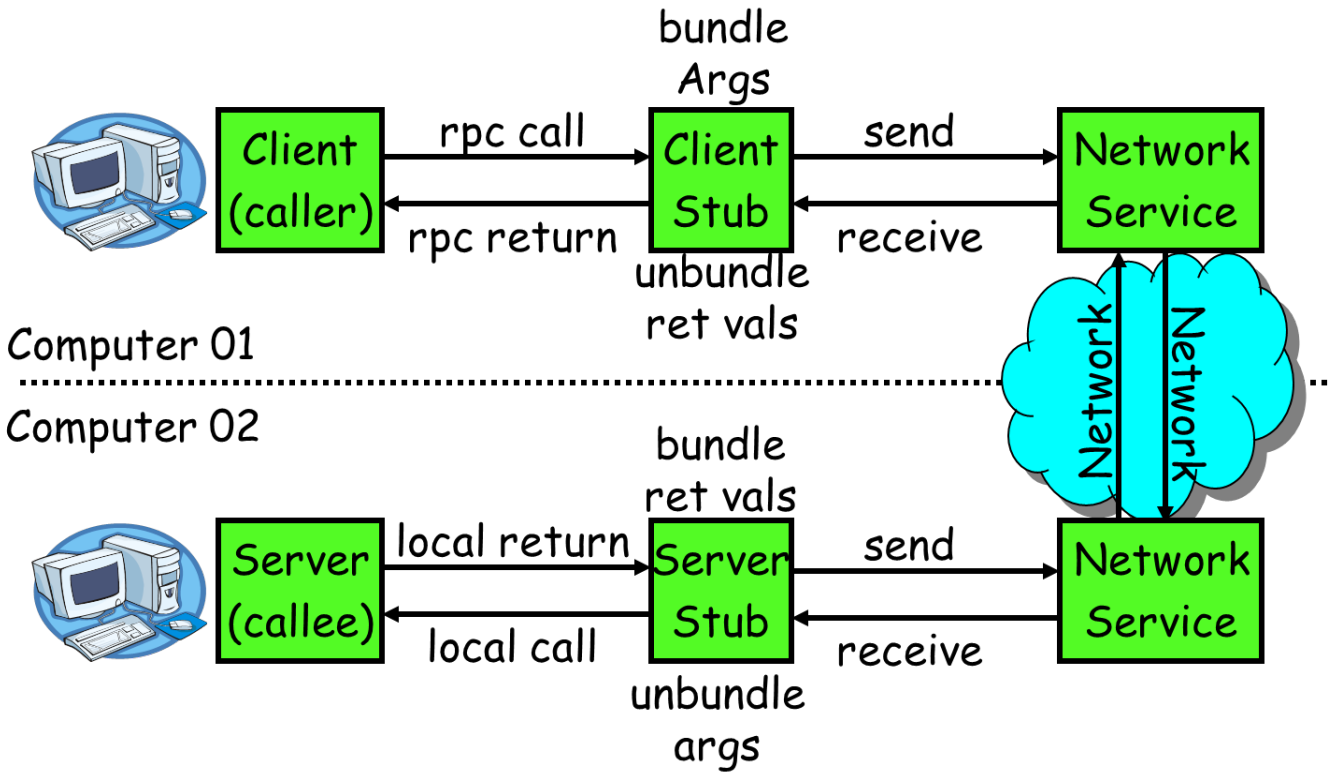
关键技术

1. 服务发布与订阅：服务端使用 Zookeeper 注册服务地址，客户端从 Zookeeper 获取可用的服务地址。
2. 通信：使用 Netty 作为通信框架。
3. Spring：使用 Spring 配置服务，加载 Bean，扫描注解。
4. 动态代理：客户端使用代理模式透明化服务调用。
5. 消息编解码：使用 Protostuff 序列化和反序列化消息。

核心流程

1. 服务消费方（client）调用以本地调用方式调用服务；
2. client stub 接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；
3. client stub 找到服务地址，并将消息发送到服务端；
4. server stub 收到消息后进行解码；
5. server stub 根据解码结果调用本地的服务；
6. 本地服务执行并将结果返回给 server stub；
7. server stub 将返回结果打包成消息并发送至消费方；
8. client stub 接收到消息，并进行解码；
9. 服务消费方得到最终结果。

RPC 的目标就是要 2~8 这些步骤都封装起来，让用户对这些细节透明。JAVA 一般使用动态代理方式实现远程调用。



消息编解码

息数据结构（接口名称+方法名+参数类型和参数值+超时时间+ **requestID**） 客户端的请求消息结构一般需要包括以下内容：

1. 接口名称：在我们的例子里接口名是“HelloWorldService”，如果不传，服务端就不知道调用哪个接口了；
2. 方法名：一个接口内可能有很多方法，如果不传方法名服务端也就不知道调用哪个方法；
3. 参数类型和参数值：参数类型有很多，比如有 bool、int、long、double、string、map、list，甚至如 struct（class）；以及相应的参数值；
4. 超时时间：
5. requestID，标识唯一请求 id，在下面一节会详细描述 requestID 的用处。
6. 服务端返回的消息： 一般包括以下内容。返回值+状态 code+requestID

序列化

目前互联网公司广泛使用 Protobuf、Thrift、Avro 等成熟的序列化解决方案来搭建 RPC 框架，这些都是久经考验的解决方案。

通讯过程

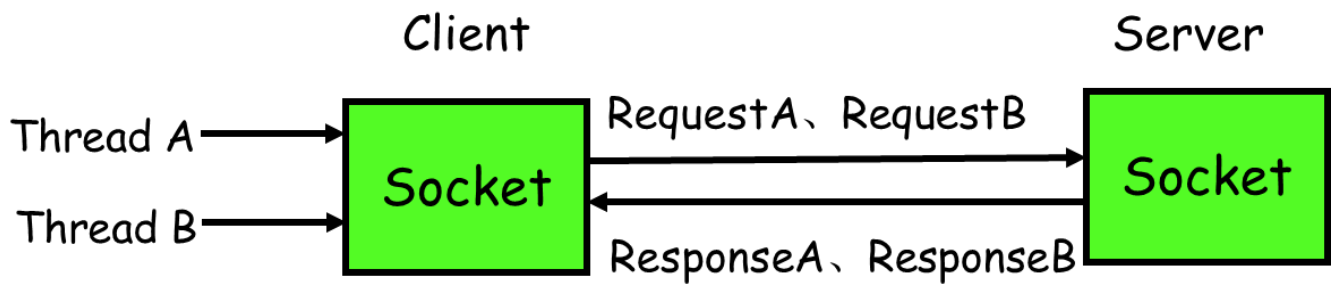
核心问题(线程暂停、消息乱序)

如果使用 netty 的话，一般会用 channel.writeAndFlush()方法来发送消息二进制串，这个方法调用后对于整个远程调用(从发出请求到接收到结果)来说是一个异步的，即对于当前线程来说，将请求发送出来后，线程就可以往后执行了，至于服务端的结果，是服务端处理完成后，再以消息的形式发送给客户端的。于是这里出现以下两个问题：

1. 怎么让当前线程“暂停”，等结果回来后，再向后执行?
2. 如果有多个线程同时进行远程方法调用，这时建立在 client server 之间的 socket 连接上会有很多双方发送的消息传递，前后顺序也可能是随机的，server 处理完结果后，将结果消息发送给 client，client 收到很多消息，怎么知道哪个消息结果是原先哪个线程调用

的? 如下图所示，线程 A 和线程 B 同时向 client socket 发送请求 requestA 和 requestB，

socket 先后将 requestB 和 requestA 发送至 server，而 server 可能将 responseB 先返回，尽管 requestB 请求到达时间更晚。我们需要一种机制保证 responseA 丢给 ThreadA，responseB 丢给 ThreadB。



通讯流程

requestID 生成-AtomicLong

1. client 线程每次通过 socket 调用一次远程接口前，生成一个唯一的 ID，即 requestID（requestID 必需保证在一个 Socket 连接里面是唯一的），一般常常使用 AtomicLong 从 0 开始累计数字生成唯一 ID；存放回调对象**callback**到全局**ConcurrentHashMap**
2. 将处理结果的回调对象 callback，存放到全局 ConcurrentHashMap 里面 put(requestID, callback)；**synchronized** 获取回调对象**callback**的锁并自旋**wait**
3. 当线程调用 channel.writeAndFlush()发送消息后，紧接着执行 callback 的 get()方法试图获取远程返回的结果。在 get()内部，则使用 synchronized 获取回调对象 callback 的锁，再先检测是否已经获取到结果，如果没有，然后调用 callback 的 wait()方法，释放 callback 上的锁，让当前线程处于等待状态。 监听消息的线程收到消息，找到**callback**上的锁并唤醒
4. 服务端接收到请求并处理后，将response结果（此结果中包含了前面的requestID）发送给客户端，客户端 socket 连接上专门监听消息的线程收到消息，分析结果，取到 requestID，再从前面的 ConcurrentHashMap 里面 get(requestID)，从而找到 callback 对象，再用 synchronized 获取 callback 上的锁，将方法调用结果设置到 callback 对象里，再调用 callback.notifyAll()唤醒前面处于等待状态的线程。

```
  public Object get() {         synchronized (this) { // 旋锁             while (true) { // 是否有结果了
If （!isDone）{
                        wait(); //没结果释放锁，让当前线程处于等待状态
}else{//获取数据并处理
}
                    }
        }
    }
    private void setDone(Response res) {
        this.res = res;         isDone = true;
        synchronized (this) { //获取锁，因为前面 wait()已经释放了 callback 的锁了             notifyAll(); // 唤醒处于等待的线程
        }
    }
```

大数据（周老师）

Hadoop

概念

就是一个大数据解决方案。它提供了一套分布式系统基础架构。 核心内容包含 hdfs 和 mapreduce。hadoop2.0 以后引入 yarn.

hdfs 是提供数据存储的，mapreduce 是方便数据计算的。

1. hdfs 又对应 namenode 和 datanode. namenode 负责保存元数据的基本信息， datanode 直接存放数据本身；
2. mapreduce 对应 jobtracker 和 tasktracker. jobtracker 负责分发任务，tasktracker 负责执行具体任务；
3. 对应到 master/slave 架构，namenode 和 jobtracker 就应该对应到 master, datanode 和 tasktracker 就应该对应到 slave.

HDFS

Client

Client（代表用 户） 通过与 NameNode 和 DataNode 交互访问 HDFS 中 的文件。 Client 提供了一个类似 POSIX 的文件系统接口供用户调用。

NameNode

整个 Hadoop 集群中只有一个 NameNode。 它是整个系统的“ 总管”， 负责管理 HDFS 的目录树和相关的文件元数据信息。 这些信息是以“ fsimage”（ HDFS 元数据镜像文件） 和 “ editlog”（HDFS 文件改动日志）两个文件形式存放在本地磁盘，当 HDFS 重启时重新构造出来的。此外， NameNode 还负责监控各个 DataNode 的健康状态， 一旦发现某个 DataNode 宕掉，则将该 DataNode 移出 HDFS 并重新备份其上面的数据。

Secondary NameNode

Secondary NameNode 最重要的任务并不是为 NameNode 元数据进行热备份， 而是定期合并 fsimage 和 edits 日志， 并传输给 NameNode。 这里需要注意的是，为了减小 NameNode 压力， NameNode 自己并不会合并 fsimage 和 edits， 并将文件存储到磁盘上， 而是交由

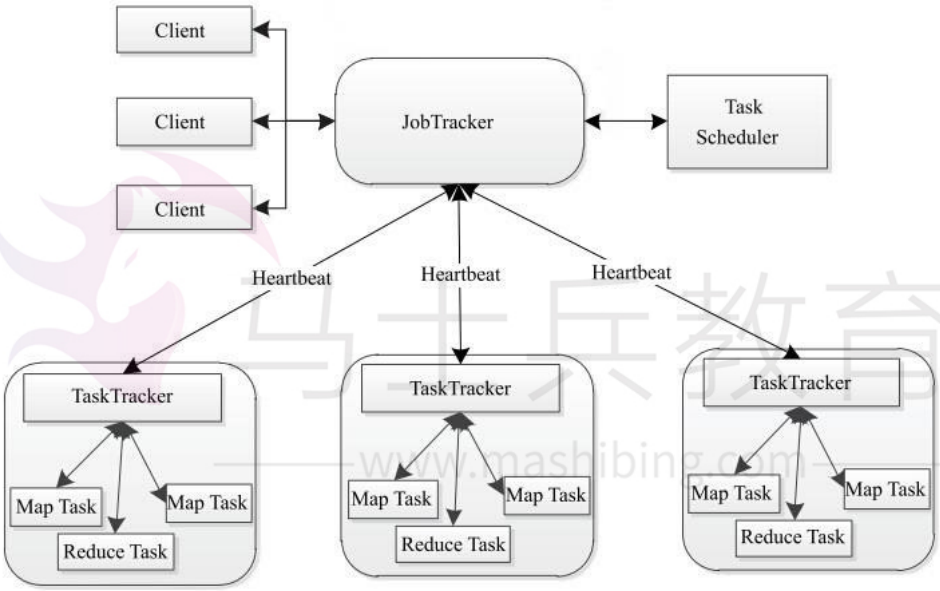
Secondary NameNode 完成。

DataNode

一般而言， 每个 Slave 节点上安装一个 DataNode， 它负责实际的数据存储， 并将数据信息定期汇报给 NameNode。 DataNode 以固定大小的 block 为基本单位组织文件内容， 默认情况下 block 大小为 64MB。 当用户上传一个大的文件到 HDFS 上时， 该文件会被切分成若干个 block，分别存储到不同的 DataNode ； 同时，为了保证数据可靠，会将同一个 block 以流水线方式写到若干个（默认是 3，该参数可配置）不同的 DataNode 上。 这种文件切割后存储的过程是对用户透明的。

MapReduce

同 HDFS 一样，Hadoop MapReduce 也采用了 Master/Slave（M/S）架构，具体如图所示。它主要由以下几个组件组成：Client、JobTracker、TaskTracker 和 Task。 下面分别对这几个组件进行介绍。



Hadoop MapReduce 架构图

Client

用户编写的 MapReduce 程序通过 Client 提交到 JobTracker 端； 同时， 用户可通过 Client 提供的一些接口查看作业运行状态。 在 Hadoop 内部用“作业”（Job） 表示 MapReduce 程序。一个 MapReduce 程序可对应若干个作业，而每个作业会被分解成若干个 Map/Reduce 任务

（Task）。

JobTracker

JobTracker 主要负责资源监控和作业调度。JobTracker 监控所有 TaskTracker 与作业的健康状况，一旦发现失败情况后，其会将相应的任务转移到其他节点；同时 JobTracker 会跟踪任务的执行进度、资源使用量等信息，并将这些信息告诉任务调度器，而调度器会在资源出现空闲时，选择合适的任务使用这些资源。在 Hadoop 中，任务调度器是一个可插拔的模块，用户可以根据自己的需要设计相应的调度器。

TaskTracker

TaskTracker 会周期性地通过 Heartbeat 将本节点上资源的使用情况和任务的运行进度汇报给 JobTracker， 同时接收 JobTracker 发送过来的命令并执行相应的操作（如启动新任务、 杀死任务等）。TaskTracker 使用“slot” 等量划分本节点上的资源量。“slot” 代表计算资源（CPU、内存等）。一个 Task 获取到一个 slot 后才有机会运行，而 Hadoop 调度器的作用就是将各个

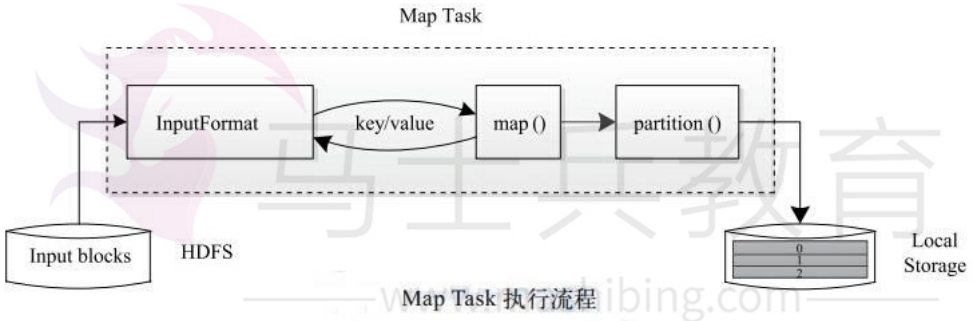
TaskTracker 上的空闲 slot 分配给 Task 使用。 slot 分为 Map slot 和 Reduce slot 两种，分别供 MapTask 和 Reduce Task 使用。 TaskTracker 通过 slot 数目（可配置参数）限定 Task 的并发度。

Task

Task 分为 Map Task 和 Reduce Task 两种， 均由 TaskTracker 启动。 HDFS 以固定大小的 block 为基本单位存储数据， 而对于 MapReduce 而言， 其处理单位是 split。split 与 block 的对应关系如图所示。 split 是一个逻辑概念， 它只包含一些元数据信息， 比如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自己决定。 但需要注意的是，split 的多少决定了 Map

Task 的数目 ，因为每个 split 会交由一个 Map Task 处理。

Map Task 执行过程如图所示。 由该图可知，Map Task 先将对应的 split 迭代解析成一个个 key/value 对，依次调用用户自定义的 map() 函数进行处理，最终将临时结果存放到本地磁盘上，其中临时数据被分成若干个 partition，每个 partition 将被一个 Reduce Task 处理。



Reduce Task执行过程

该过程分为三个阶段

1. 从远程节点上读取 MapTask 中间结果（称为“Shuffle 阶段”）；
2. 按照 key 对 key/value 对进行排序（称为“ Sort 阶段”）；
3. 依次读取<key, value list>，调用用户自定义的 reduce() 函数处理，并将最终结果存到 HDFS 上（称为“ Reduce 阶段”）。

Hadoop MapReduce 作业的生命周期

1.作业提交与初始化

1. 用户提交作业后， 首先由 JobClient 实例将作业相关信息， 比如将程序 jar 包、作业配置文件、 分片元信息文件等上传到分布式文件系统（ 一般为 HDFS）上，其中，分片元信息文件记录了每个输入分片的逻辑位置信息。 然后 JobClient 通过 RPC 通知 JobTracker。 JobTracker 收到新作业提交请求后， 由 作业调度模块对作业进行初始化：为作业创建一个 JobInProgress 对象以跟踪作业运行状况， 而 JobInProgress 则会为每个 Task 创建一个 TaskInProgress 对象以跟踪每个任务的运行状态，TaskInProgress 可能需要管理多个

“ Task 运行尝试”（ 称为“ Task Attempt”）。

2.任务调度与监控。

2. 前面提到，任务调度和监控的功能均由 JobTracker 完成。TaskTracker 周期性地通过

Heartbeat 向 JobTracker 汇报本节点的资源使用 情况， 一旦出 现空闲资源， JobTracker 会按照一定的策略选择一个合适的任务使用该空闲资源， 这由任务调度器完成。 任务调度器是一个可插拔的独立模块， 且为双层架构， 即首先选择作业， 然后从该作业中选择任务， 其中，选择任务时需要重点考虑数据本地性。 此外，JobTracker 跟踪作业的整个运行过程，并为作业的成功运行提供全方位的保障。 首先， 当 TaskTracker 或者 Task 失败时， 转移计算任务； 其次， 当某个 Task 执行进度远落后于同一作业的其他 Task 时，为之启动一个相同

Task， 并选取计算快的 Task 结果作为最终结果。

3.任务运行环境准备

3. 运行环境准备包括 JVM 启动和资源隔 离， 均由 TaskTracker 实现。 TaskTracker 为每个

Task 启动一个独立的 JVM 以避免不同 Task 在运行过程中相互影响； 同时，TaskTracker 使用了操作系统进程实现资源隔离以防止 Task 滥用资源。

4.任务执行

4. TaskTracker 为 Task 准备好运行环境后， 便会启动 Task。 在运行过程中， 每个 Task 的最

新进度首先由 Task 通过 RPC 汇报给 TaskTracker， 再由 TaskTracker 汇报给 JobTracker。

5.作业完成。

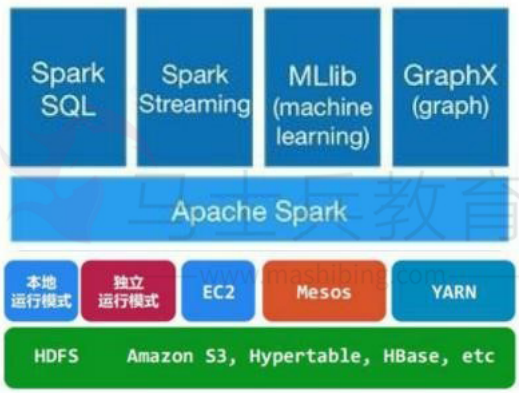
5. 待所有 Task 执行完毕后， 整个作业执行成功。

Spark

概念

Spark 提供了一个全面、统一的框架用于管理各种有着不同性质（文本数据、图表数据等）的数据集和数据源（批量数据或实时的流数据）的大数据处理的需求。

核心架构



Spark SQL

提供通过 Apache Hive 的 SQL 变体 Hive 查询语言（HiveQL）与 Spark 进行交互的 API。每个数据库表被当做一个 RDD，Spark SQL 查询被转换为 Spark 操作。

Spark Streaming

对实时数据流进行处理和控制。Spark Streaming 允许程序能够像普通 RDD 一样处理实时数据

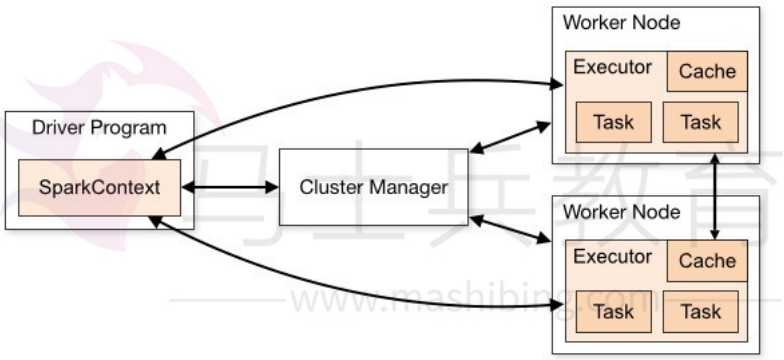
MIlib

一个常用机器学习算法库，算法被实现为对 RDD 的 Spark 操作。这个库包含可扩展的学习算法，比如分类、回归等需要对大量数据集进行迭代的操作。

GraphX

控制图、并行图操作和计算的一组算法和工具的集合。GraphX 扩展了 RDD API，包含控制图、创建子图、访问路径上所有顶点的操作

核心组件



Cluster Manager-制整个集群，监控worker

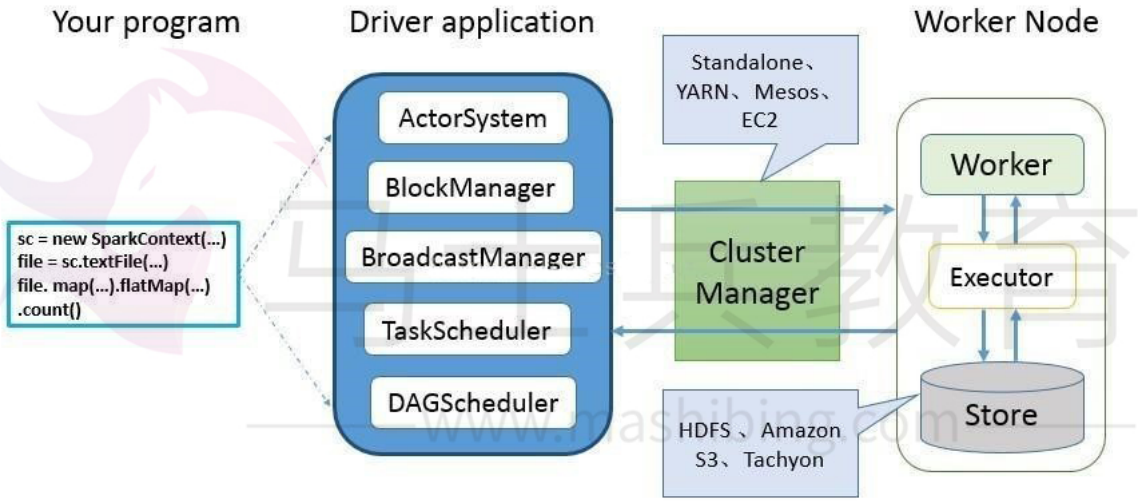
在 standalone 模式中即为 Master 主节点，控制整个集群，监控 worker。在 YARN 模式中为资源管理器

Worker节点-负责控制计算节点从节点，负责控制计算节点，启动 Executor 或者 Driver。

Driver： 运行**Application** 的**main()**函数

Executor：执行器，是为某个**Application** 运行在**worker node** 上的一个进程

SPARK编程模型

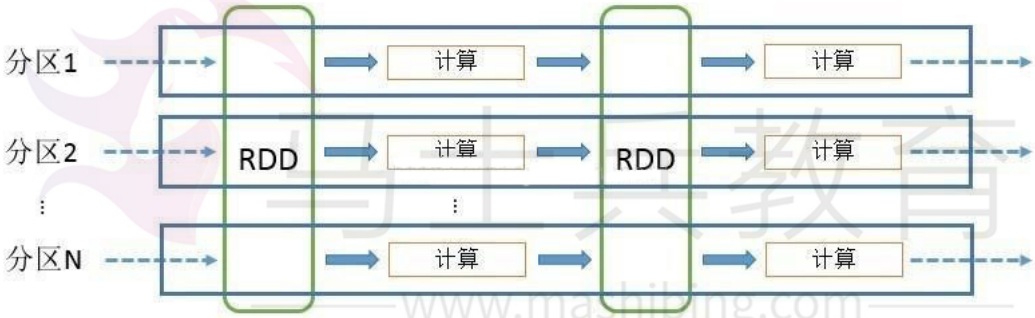


Spark 应用程序从编写到提交、执行、输出的整个过程如图所示，图中描述的步骤如下：

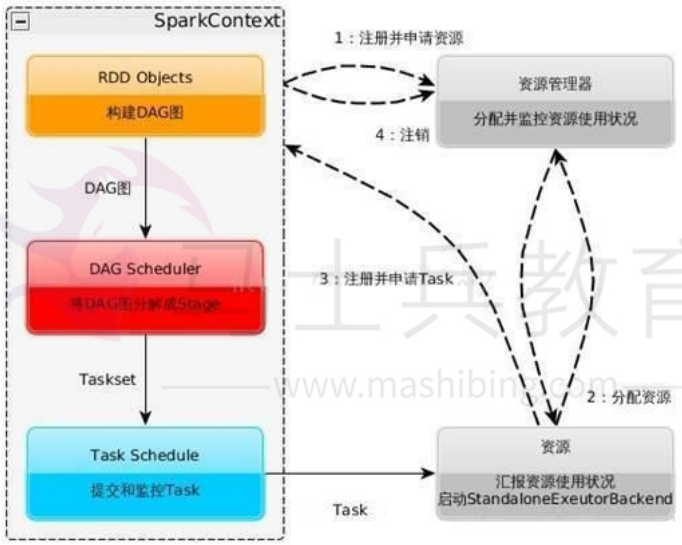
1. 用户使用SparkContext提供的API（常用的有textFile、sequenceFile、runJob、stop等）编写 Driver application 程序。此外 SQLContext、HiveContext 及 StreamingContext 对 SparkContext 进行封装，并提供了 SQL、Hive 及流式计算相关的 API。
2. 使用SparkContext提交的用户应用程序，首先会使用BlockManager和BroadcastManager 将任务的 Hadoop 配置进行广播。然后由 DAGScheduler 将任务转换为 RDD 并组织成 DAG， DAG 还将被划分为不同的 Stage。最后由 TaskScheduler 借助 ActorSystem 将任务提交给集群管理器（Cluster Manager）。
3. 集群管理器（ClusterManager）给任务分配资源，即将具体任务分配到Worker上，Worker 创建 Executor 来处理任务的运行。Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。

SPARK 计算模型

RDD 可以看做是对各种数据计算模型的统一抽象，Spark 的计算过程主要是 RDD 的迭代计算过程。RDD 的迭代计算过程非常类似于管道。分区数量取决于 partition 数量的设定，每个分区的数据只会在一个 Task 中计算。所有分区可以在多个机器节点的 Executor 上并行执行。

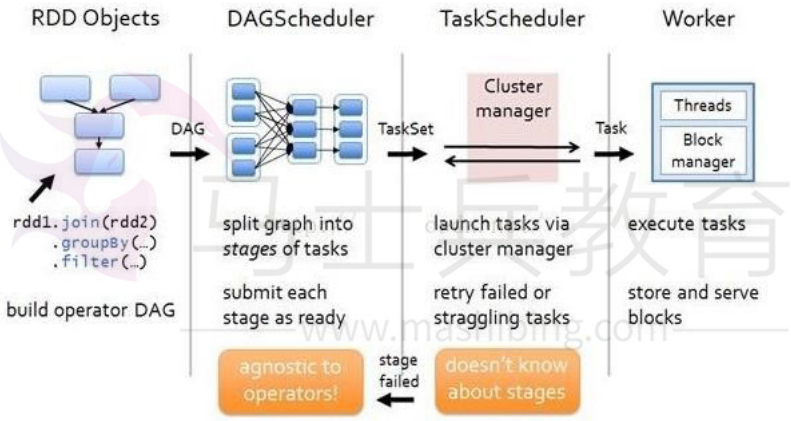


SPARK 运行流程



1. 构建**Spark Application** 的运行环境，启动**SparkContext**
2. **SparkContext** 向资源管理器（可以是**Standalone**，**Mesos**，**Yarn**）申请运行**Executor** 资源，并启动**StandaloneExecutorbackend**，
3. **Executor**向**SparkContext** 申请**Task**
4. **SparkContext** 将应用程序分发给**Executor**
5. **SparkContext** 构建成**DAG** 图，将**DAG**图分解成**Stage**、将**Taskset** 发送给**Task Scheduler**，最后由**Task Scheduler** 将**Task**发送给**Executor**运行
6. **Task**在**Executor** 上运行，运行完释放所有资源

SPARK RDD 流程



1. 创建 RDD 对象
2. DAGScheduler 模块介入运算，计算 RDD 之间的依赖关系，RDD 之间的依赖关系就形成了DAG
3. 每一个 Job 被分为多个 Stage。划分 Stage 的一个主要依据是当前计算因子的输入是否是确定的，如果是则将其分在同一个 Stage，避免多个 Stage 之间的消息传递开销

SPARK RDD

（1） **RDD** 的创建方式

1） 从Hadoop文件系统（或与Hadoop兼容的其他持久化存储系统，如Hive、Cassandra、 HBase）输入（例如 HDFS）创建。

2） 从父 RDD 转换得到新 RDD。

3） 通过 parallelize 或 makeRDD 将单机数据创建为分布式 RDD。

（2） **RDD** 的两种操作算子（**转换（Transformation）与行动（Action）**） 对于 RDD 可以有两种操作算子：转换（Transformation）与行动（Action）。

转换（Transformation）：Transformation操作是延迟计算的，也就是说从一个RDD转换生成另一个 RDD 的转换操作不是马上执行，需要等到有 Action 操作的时候才会真正触发运算。

map(func)	对调用map的RDD数据集中的每个element都使用func，然后返回一个新的RDD,这个返回的数据集是分布式的数据集
filter(func)	对调用filter的RDD数据集中的每个元素都使用func，然后返回一个包含使func为true的元素构成的RDD
flatMap(func)	和map差不多，但是flatMap生成的是多个结果
mapPartitions(*func*)	和map很像，但是map是每个element，而mapPartitions是每个partition
sample(*withReplacement*, *fraction*, *seed*)	抽样
union(*otherDataset*)	返回一个新的dataset，包含源dataset和给定dataset的元素的集合
distinct([*numTasks*]))	返回一个新的dataset，这个dataset含有的是源dataset中的distinct的element
groupByKey([*numTasks*])	返回(K,Seq[V])，也就是hadoop中reduce函数接受的key-valuelist
reduceByKey(*func*, [*numTasks*])	就是用一个给定的reducefunc再作用在groupByKey产生的(K,Seq[V]),比如求和，求平均数
sortByKey([ascending],[numTasks])	按照key来进行排序，是升序还是降序，ascending是boolean类型
join(*otherDataset*, [*numTasks*])	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,(V,W))的dataset,numTasks为并发的任务数
cogroup(*otherDataset*, [*numTasks*])	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,Seq[V],Seq[W])的dataset,numTasks为并发的任务数
cartesian(otherDataset)	笛卡尔积就是m*n

行动（Action）：Action 算子会触发 Spark 提交作业（Job），并将数据输出 Spark 系统。

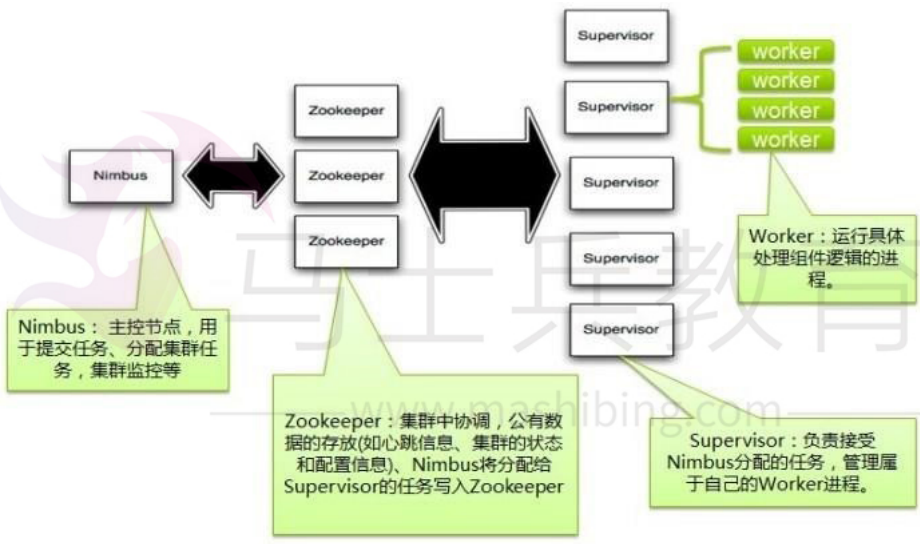
reduce(func)	聚集，但是传入的函数是两个参数输入返回一个值，这个函数必须是满足交换律和结合律的
collect()	一般在filter或者足够小的结果的时候，再用collect封装返回一个数组
count()	返回的是dataset中的element的个数
first()	返回的是dataset中的第一个元素
take(n)	返回前n个elements，这个士driverprogram返回的
takeSample(withReplacement，num，seed)	抽样返回一个dataset中的num个元素，随机种子seed
saveAsTextFile（path）	把dataset写到一个textfile中，或者hdfs，或者hdfs支持的文件系统中，spark把每条记录都转换为一行记录，然后写到file中
saveAsSequenceFile(path)	只能用在key-value对上，然后生成SequenceFile写到本地或者hadoop文件系统
saveAsObjectFile(*path*)	把dataset写到一个java序列化的文件中，用SparkContext.objectFile()装载
countByKey()	返回的是key对应的个数的一个map，作用于一个RDD
foreach(func)	对dataset中的每个元素都使用func

Storm

概念

Storm 是一个免费并开源的分布式实时计算系统。利用 Storm 可以很容易做到可靠地处理无限的数据流，像 Hadoop 批量处理大数据一样，Storm 可以实时处理数据。

集群架构



Nimbus（master-代码分发给 Supervisor）

Storm 集群的 Master 节点，负责分发用户代码，指派给具体的 Supervisor 节点上的 Worker 节点，去运行 Topology 对应的组件（Spout/Bolt）的 Task。

Supervisor（slave-管理 Worker 进程的启动和终止）

Storm 集群的从节点，负责管理运行在 Supervisor 节点上的每一个 Worker 进程的启动和终止。通过 Storm 的配置文件中的 supervisor.slots.ports 配置项，可以指定在一个 Supervisor 上最大允许多少个 Slot，每个 Slot 通过端口号来唯一标识，一个端口号对应一个 Worker 进程（如果该 Worker 进程被启动）。

Worker（具体处理组件逻辑的进程）

运行具体处理组件逻辑的进程。Worker 运行的任务类型只有两种，一种是 Spout 任务，一种是

Bolt 任务。

Task

worker中每一个spout/bolt的线程称为一个task. 在storm0.8之后，task不再与物理线程对应，不同 spout/bolt 的 task 可能会共享一个物理线程，该线程称为 executor。

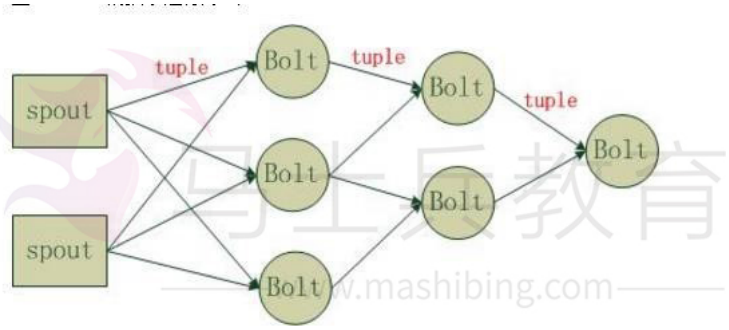
ZooKeeper

用来协调 Nimbus 和 Supervisor，如果 Supervisor 因故障出现问题而无法运行 Topology，

Nimbus 会第一时间感知到，并重新分配 Topology 到其它可用的 Supervisor 上运行

编程模型（spout->tuple->bolt）

strom 在运行中可分为 spout 与 bolt 两个组件，其中，数据源从 spout 开始，数据以 tuple 的方式发送到 bolt，多个 bolt 可以串连起来，一个 bolt 也可以接入多个 spot/bolt. 运行时原理如下图：



Topology

Storm 中运行的一个实时应用程序的名称。将 Spout、 Bolt 整合起来的拓扑图。定义了 Spout 和 Bolt 的结合关系、并发数量、配置等等。

Spout

在一个 topology 中获取源数据流的组件。通常情况下 spout 会从外部数据源中读取数据，然后转换为 topology 内部的源数据。

Bolt

接受数据然后执行处理的组件,用户可以在其中执行自己想要的操作。

Tuple

一次消息传递的基本单元，理解为一组消息就是一个 Tuple。

Stream

Tuple 的集合。表示数据的流向。

Topology 运行

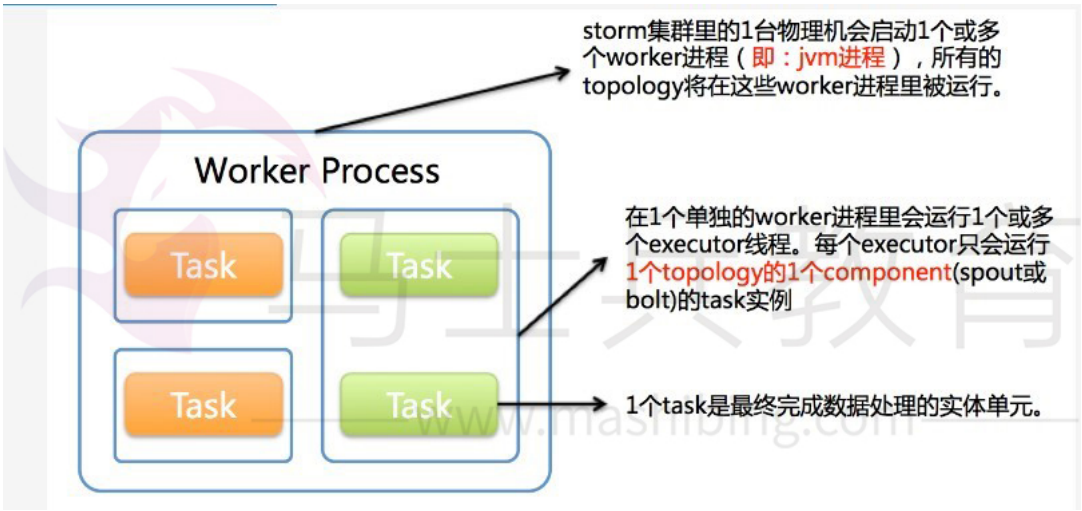
在 Storm 中,一个实时应用的计算任务被打包作为 Topology 发布，这同 Hadoop MapReduce 任务相似。但是有一点不同的是:在 Hadoop 中，MapReduce 任务最终会执行完成后结束；而在 Storm 中，Topology 任务一旦提交后永远不会结束，除非你显示去停止任务。计算任务

Topology 是由不同的 Spouts 和 Bolts，通过数据流（Stream）连接起来的图。一个 Storm 在集群上运行一个 Topology 时，主要通过以下 3 个实体来完成 Topology 的执行工作：

(1). Worker（进程）

(2). Executor（线程）

(3). Task



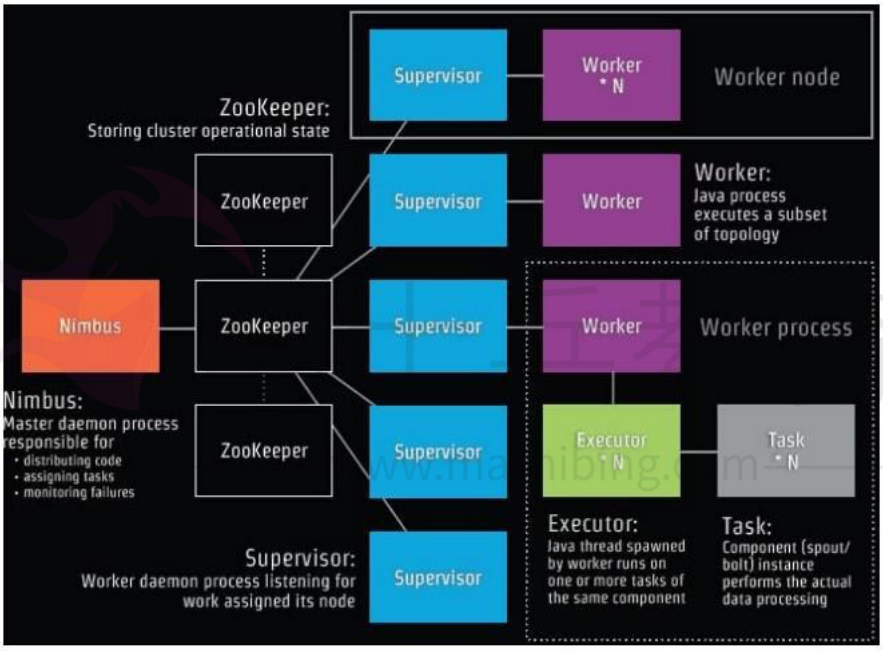
Worker(1个 worker 进程执行的是 1 个 topology 的子集) 1 个 worker 进程执行的是 1 个 topology 的子集（注：不会出现 1 个 worker 为多个 topology 服务）。1 个 worker 进程会启动 1 个或多个 executor 线程来执行 1 个 topology 的 component(spout 或 bolt)。因此，1 个运行中的 topology 就是由集群中多台物理机上的多个 worker 进程组成的。

Executor(executor是 1 个被 worker 进程启动的单独线程)

executor 是 1 个被 worker 进程启动的单独线程。每个 executor 只会运行 1 个 topology 的 1 个 component(spout 或 bolt)的 task（注：task 可以是 1 个或多个，storm 默认是 1 个 component 只生成 1 个 task，executor 线程里会在每次循环里顺序调用所有 task 实例）。

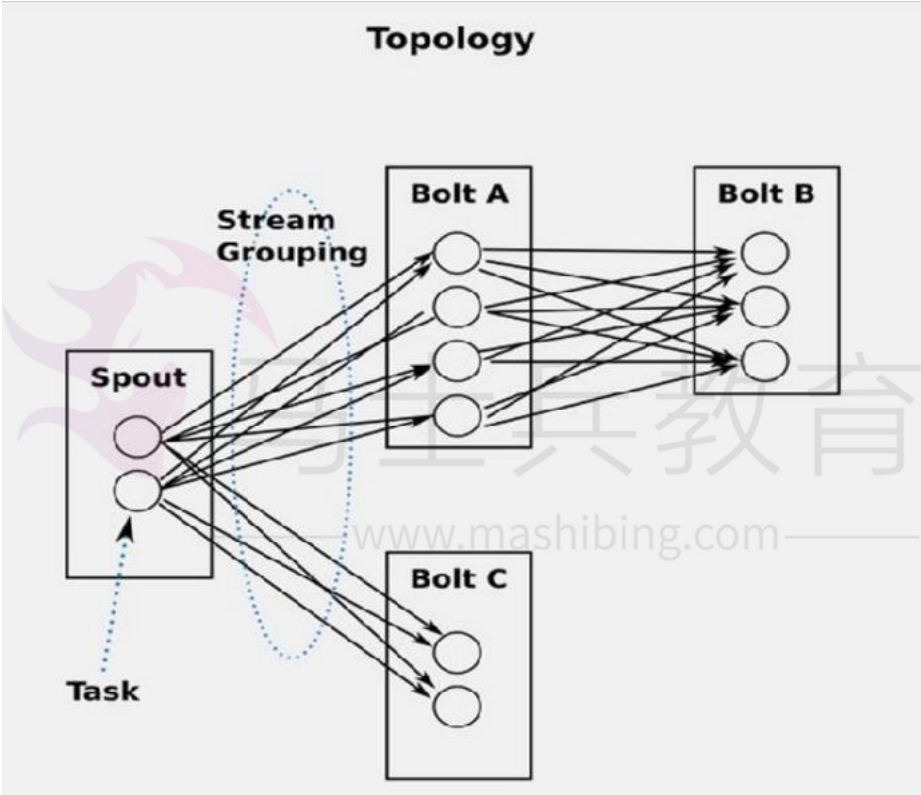
Task(最终运行 spout 或 bolt 中代码的单元)

是最终运行 spout 或 bolt 中代码的单元（注：1 个 task 即为 spout 或 bolt 的 1 个实例， executor 线程在执行期间会调用该 task 的 nextTuple 或 execute 方法）。topology 启动后，1 个 component(spout 或 bolt)的 task 数目是固定不变的，但该 component 使用的 executor 线程数可以动态调整（例如：1 个 executor 线程可以执行该 component 的 1 个或多个 task 实例）。这意味着，对于 1 个 component 存在这样的条件：#threads<=#tasks（即：线程数小于等于 task 数目）。默认情况下 task 的数目等于 executor 线程数目，即 1 个 executor 线程只运行 1 个 task。



Storm Streaming Grouping

Storm 中最重要的抽象，应该就是 Stream grouping 了，它能够控制 Spot/Bolt 对应的 Task 以什么样的方式来分发 Tuple，将 Tuple 发射到目的 Spot/Bolt 对应的 Task.



目前，Storm Streaming Grouping 支持如下几种类型：

huffle Grouping

随机分组，尽量均匀分布到下游 Bolt 中将流分组定义为混排。这种混排分组意味着来自 Spout 的输入将混排，或随机分发给此 Bolt 中的任务。shuffle grouping 对各个 task 的 tuple 分配的比较均匀。

Fields Grouping

按字段分组，按数据中 field 值进行分组；相同 field 值的 Tuple 被发送到相同的 Task 这种 grouping 机制保证相同 field 值的 tuple 会去同一个 task。

All grouping：广播

广播发送， 对于每一个 tuple 将会复制到每一个 bolt 中处理。

Global grouping

全局分组，Tuple 被分配到一个 Bolt 中的一个 Task，实现事务性的 Topology。Stream 中的所有的 tuple 都会发送给同一个 bolt 任务处理，所有的 tuple 将会发送给拥有最小 task_id 的 bolt 任务处理。

None grouping：不分组

不关注并行处理负载均衡策略时使用该方式，目前等同于 shuffle grouping,另外 storm 将会把 bolt 任务和他的上游提供数据的任务安排在同一个线程下。

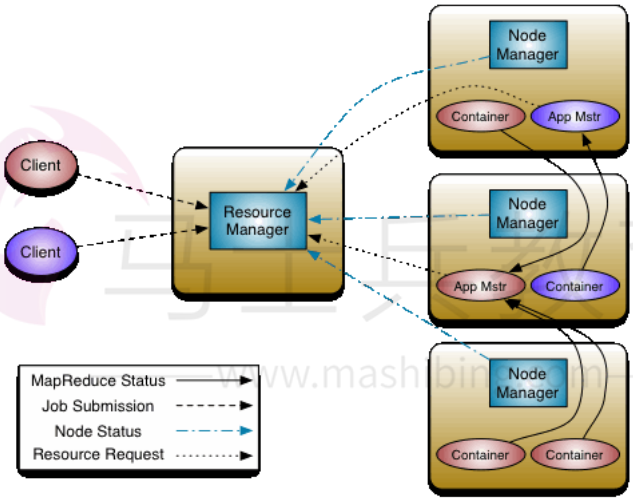
Direct grouping：直接分组 指定分组

由 tuple 的发射单元直接决定 tuple 将发射给那个 bolt，一般情况下是由接收 tuple 的 bolt 决定接收哪个 bolt 发射的 Tuple。这是一种比较特别的分组方法，用这种分组意味着消息的发送者指定由消息接收者的哪个 task 处理这个消息。 只有被声明为 Direct Stream 的消息流可以声明这种分组方法。而且这种消息 tuple 必须使用 emitDirect 方法来发射。消息处理者可以通过TopologyContext 来获取处理它的消息的 taskid (OutputCollector.emit 方法也会返回 taskid)。

YARN

概念

YARN 是一个资源管理、任务调度的框架，主要包含三大模块：ResourceManager（RM）、 NodeManager（NM）、ApplicationMaster（AM）。其中，ResourceManager 负责所有资源的监控、分配和管理；ApplicationMaster 负责每一个具体应用程序的调度和协调； NodeManager 负责每一个节点的维护。对于所有的applications，RM拥有绝对的控制权和对资源的分配权。而每个 AM 则会和 RM 协商资源，同时和 NodeManager 通信来执行和监控 task。几个模块之间的关系如图所示。



ResourceManager

1. ResourceManager 负责整个集群的资源管理和分配，是一个全局的资源管理系统。
2. NodeManager 以心跳的方式向 ResourceManager 汇报资源使用情况（目前主要是 CPU 和内存的使用情况）。RM 只接受 NM 的资源回报信息，对于具体的资源处理则交给 NM 自己处理。
3. YARN Scheduler 根据 application 的请求为其分配资源，不负责 application job 的监控、追踪、运行状态反馈、启动等工作。

NodeManager

1. NodeManager 是每个节点上的资源和任务管理器，它是管理这台机器的代理，负责该节点程序的运行，以及该节点资源的管理和监控。YARN集群每个节点都运行一个NodeManager。
2. NodeManager 定时向 ResourceManager 汇报本节点资源（CPU、内存）的使用情况和 Container 的运行状态。当 ResourceManager 宕机时 NodeManager 自动连接 RM 备用节点。
3. NodeManager 接收并处理来自 ApplicationMaster 的 Container 启动、停止等各种请求。

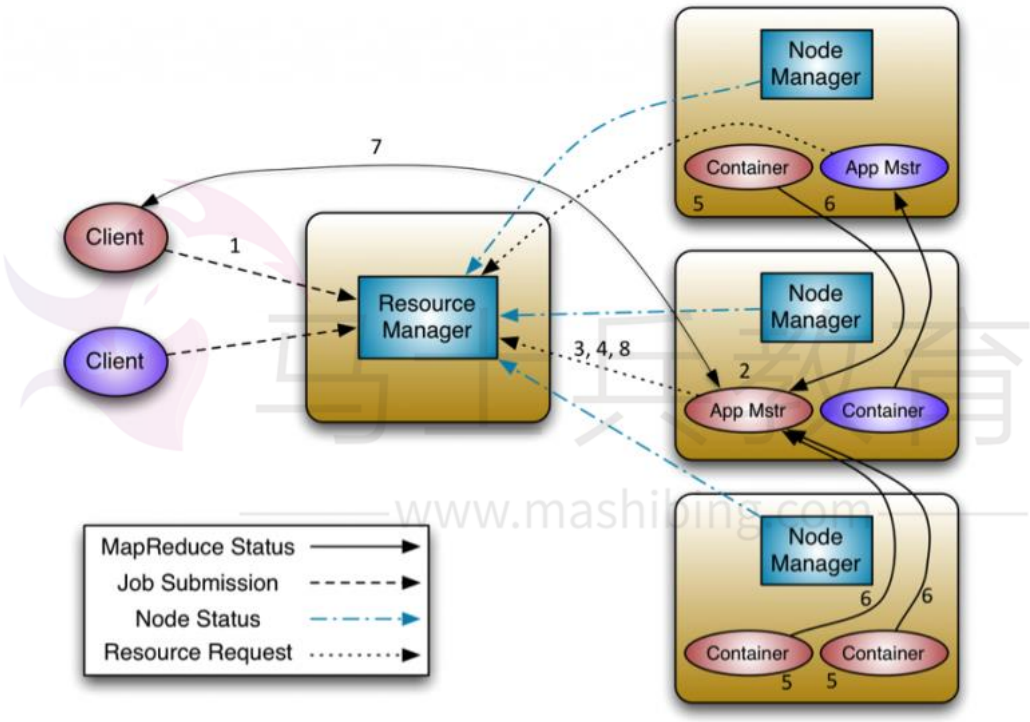
ApplicationMaster

用户提交的每个应用程序均包含一个ApplicationMaster，它可以运行在ResourceManager以外的机器上。

1. 负责与 RM 调度器协商以获取资源（用 Container 表示）。
2. 将得到的任务进一步分配给内部的任务(资源的二次分配)。
3. 与 NM 通信以启动/停止任务。
4. 监控所有任务运行状态，并在任务运行失败时重新为任务申请资源以重启任务。
5. 当前 YARN 自带了两个 ApplicationMaster 实现，一个是用于演示 AM 编写方法的实例程序 DistributedShell，它可以申请一定数目的 Container 以并行运行一个 Shell 命令或者 Shell 脚本；另一个是运行 MapReduce 应用程序的 AM—MRAppMaster。

注：RM 只负责监控 AM，并在 AM 运行失败时候启动它。RM 不负责 AM 内部任务的容错，任务的容错由 AM 完成。

YARN运行流程



1. client 向 RM 提交应用程序，其中包括启动该应用的 ApplicationMaster 的必须信息，例如 ApplicationMaster 程序、启动 ApplicationMaster 的命令、用户程序等。
2. ResourceManager 启动一个 container 用于运行 ApplicationMaster。
3. 启动中的ApplicationMaster向ResourceManager注册自己，启动成功后与RM保持心跳。
4. ApplicationMaster 向 ResourceManager 发送请求，申请相应数目的 container。
5. ResourceManager 返回 ApplicationMaster 的申请的 containers 信息。申请成功的 container，由 ApplicationMaster 进行初始化。container 的启动信息初始化后，AM 与对应的 NodeManager 通信，要求 NM 启动 container。AM 与 NM 保持心跳，从而对 NM 上运行的任务进行监控和管理。
6. container 运行期间，ApplicationMaster 对 container 进行监控。container 通过 RPC 协议向对应的 AM 汇报自己的进度和状态等信息。
7. 应用运行期间，client 直接与 AM 通信获取应用的状态、进度更新等信息。
8. 应用运行结束后，ApplicationMaster 向 ResourceManager 注销自己，并允许属于它的 container 被收回。

负载均衡的原理

六大Web负载均衡原理与实现

开头先理解一下所谓的"均衡"

不能狭义地理解为分配给所有实际服务器一样多的工作量，因为多台服务器的承载能力各不相同，这可能体现在硬件配置、网络带宽的差异，也可能因为某台服务器身兼多职，我们所说的"均衡"，也就是希望所有服务器都不要过载，并且能够最大程序地发挥作用。

一、http重定向

当http代理（比如浏览器）向web服务器请求某个URL后，web服务器可以通过http响应头信息中的Location标记来返回一个新的URL。这意味着HTTP代理需要继续请求这个新的URL，完成自动跳转。

性能缺陷：

1、吞吐率限制

主站点服务器的吞吐率平均分配到了被转移的服务器。现假设使用RR（Round Robin）调度策略，子服务器的最大吞吐率为1000reqs/s，那么主服务器的吞吐率要达到3000reqs/s才能完全发挥三台子服务器的作用，那么如果有100台子服务器，那么主服务器的吞吐率可想而知得有大？相反，如果主服务的最大吞吐率为6000reqs/s，那么平均分配到子服务器的吞吐率为2000reqs/s，而现子服务器的最大吞吐率为1000reqs/s，因此就得增加子服务器的数量，增加到6个才能满足。

2、重定向访问深度不同

有的重定向一个静态页面，有的重定向相比复杂的动态页面，那么实际服务器的负载差异是不可预料的，而主站服务器却一无所知。因此整站使用重定向方法做负载均衡不太好。

我们需要权衡转移请求的开销和处理实际请求的开销，前者相对于后者越小，那么重定向的意义就越大，例如下载。你可以去很多镜像下载网站试下，会发现基本下载都使用了Location做了重定向。

二、DNS负载均衡

DNS负责提供域名解析服务，当访问某个站点时，实际上首先需要通过该站点域名的DNS服务器来获取域名指向的IP地址，在这一过程中，DNS服务器完成了域名到IP地址的映射，同样，这样映射也可以是一对多的，这时候，DNS服务器便充当了负载均衡调度器，它就像http重定向转换策略一样，将用户的请求分散到多台服务器上，但是它的实现机制完全不同。

使用dig命令来看下"baidu"的DNS设置

```
[redacted] ~> dig baidu.com

; <<>> DiG 9.6-ESV-R5-P1 <<>> baidu.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34721
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;baidu.com.                     IN      A

;; ANSWER SECTION:
baidu.com.              584     IN      A       220.181.111.86
baidu.com.              584     IN      A       123.125.114.144
baidu.com.              584     IN      A       220.181.111.85

;; Query time: 300 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Thu Nov 20 18:00:08 2014
;; MSG SIZE  rcvd: 75
```

可见baidu拥有三个A记录

相比http重定向，基于DNS的负载均衡完全节省了所谓的主站点，或者说DNS服务器已经充当了主站点的职能。但不同的是，作为调度器，DNS服务器本身的性能几乎不用担心。因为DNS记录可以被用户浏览器或者互联网接入服务商的各级DNS服务器缓存，只有当缓存过期后才会重新向域名的DNS服务器请求解析。也说是DNS不存在http的吞吐率限制，理论上可以无限增加实际服务器的数量。

特性:

1、可以根据用户IP来进行智能解析。DNS服务器可以在所有可用的A记录中寻找离用记最近的一台服务器。

2、动态DNS：在每次IP地址变更时，及时更新DNS服务器。当然，因为缓存，一定的延迟不可避免。

不足:

1、没有用户能直接看到DNS解析到了哪一台实际服务器，加服务器运维人员的调试带来了不便。

2、策略的局限性。例如你无法将HTTP请求的上下文引入到调度策略中，而在前面介绍的基于HTTP重定向的负载均衡系统中，调度器工作在HTTP层面，它可以充分理解HTTP请求后根据站点的应用逻辑来设计调度策略，比如根据请求不同的URL来进行合理的过滤和转移。

3、如果要根据实际服务器的实时负载差异来调整调度策略，这需要DNS服务器在每次解析操作时分析各服务器的健康状态，对于DNS服务器来说，这种自定义开发存在较高的门槛，更何况大多数站点只是使用第三方DNS服务。

4、DNS记录缓存，各级节点的DNS服务器不同程序的缓存会让你晕头转向。

5、基于以上几点，DNS服务器并不能很好地完成工作量均衡分配，最后，是否选择基于DNS的负载均衡方式完全取决于你的需要。

三、反向代理负载均衡

这个肯定大家都有所接触，因为几乎所有主流的Web服务器都热衷于支持基于反向代理的负载均衡。它的核心工作就是转发HTTP请求。

相比前面的HTTP重定向和DNS解析，反向代理的调度器扮演的是用户和实际服务器中间人的角色:

1、任何对于实际服务器的HTTP请求都必须经过调度器

2、调度器必须等待实际服务器的HTTP响应，并将它反馈给用户（前两种方式不需要经过调度反馈，是实际服务器直接发送给用户）

特性:

1、调度策略丰富。例如可以为不同的实际服务器设置不同的权重，以达到能者多劳的效果。

2、对反向代理服务器的并发处理能力要求高，因为它工作在HTTP层面。

3、反向代理服务器进行转发操作本身是需要一定开销的，比如创建线程、与后端服务器建立TCP连接、接收后端服务器返回的处理结果、分析HTTP头部信息、用户空间和内核空间的频繁切换等，虽然这部分时间并不长，但是当后端服务器处理请求的时间非常短时，转发的开销就显得尤为突出。例如请求静态文件，更适合使用前面介绍的基于DNS的负载均衡方式。

4、反向代理服务器可以监控后端服务器，比如系统负载、响应时间、是否可用、TCP连接数、流量等，从而根据这些数据调整负载均衡的策略。

5、反射代理服务器可以让用户在一次会话周期内的所有请求始终转发到一台特定的后端服务器上（粘滞会话），这样的好处一是保持session的本地访问，二是防止后端服务器的动态内存缓存的资源浪费。

四、IP负载均衡(LVS-NAT)

因为反向代理服务器工作在HTTP层，其本身的开销就已经严重制约了可扩展性，从而也限制了它的性能极限。那能否在HTTP层面以下实现负载均衡呢?

NAT服务器:它工作在传输层，它可以修改发送来的IP数据包，将数据包的目标地址修改为实际服务器地址。

从Linux2.4内核开始，其内置的Neftilter模块在内核中维护着一些数据包过滤表，这些表包含了用于控制数据包过滤的规则。可喜的是，Linux提供了iptables来对过滤表进行插入、修改和删除等操作。更加令人振奋的是，Linux2.6.x内核中内置了IPVS模块，它的工作性质类型于Netfilter模块，不过它更专注于实现IP负载均衡。

想知道你的服务器内核是否已经安装了IPVS模块，可以

```
[ $ modprobe -l | grep ipvs
kernel/net/netfilter/ipvs/ip_vs.ko
kernel/net/netfilter/ipvs/ip_vs_rr.ko
kernel/net/netfilter/ipvs/ip_vs_wrr.ko
kernel/net/netfilter/ipvs/ip_vs_lc.ko
kernel/net/netfilter/ipvs/ip_vs_wlc.ko
kernel/net/netfilter/ipvs/ip_vs_lblc.ko
kernel/net/netfilter/ipvs/ip_vs_lblcr.ko
kernel/net/netfilter/ipvs/ip_vs_dh.ko
kernel/net/netfilter/ipvs/ip_vs_sh.ko
kernel/net/netfilter/ipvs/ip_vs_sed.ko
kernel/net/netfilter/ipvs/ip_vs_nq.ko
kernel/net/netfilter/ipvs/ip_vs_ftp.ko
kernel/net/netfilter/ipvs/ip_vs_pe_sip.ko
```

有输出意味着IPVS已经安装了。IPVS的管理工具是ipvsadm，它为提供了基于命令行的配置界面，可以通过它快速实现负载均衡系统。这就是大名鼎鼎的LVS(Linux Virtual Server，Linux虚拟服务器)。

1、打开调度器的数据包转发选项

echo 1 > /proc/sys/net/ipv4/ip_forward

2、检查实际服务器是否已经将NAT服务器作为自己的默认网关，如果不是，如添加

route add default gw xx.xx.xx.xx

3、使用ipvsadm配置

ipvsadm -A -t 111.11.11.11:80 -s rr

添加一台虚拟服务器，-t 后面是服务器的外网ip和端口，-s rr是指采用简单轮询的RR调度策略（这属于静态调度策略，除此之外，LVS还提供了系列的动态调度策略，比如最小连接（LC）、带权重的最小连接（WLC），最短期望时间延迟（SED）等）

ipvsadm -a -t 111.11.11.11:80 -r 10.10.120.210:8000 -m

ipvsadm -a -t 111.11.11.11:80 -r 10.10.120.211:8000 -m

添加两台实际服务器（不需要有外网ip），-r后面是实际服务器的内网ip和端口，-m表示采用NAT方式来转发数据包

运行ipvsadm -L -n可以查看实际服务器的状态。这样就大功告成了。

实验证明使用基于NAT的负载均衡系统。作为调度器的NAT服务器可以将吞吐率提升到一个新的高度，几乎是反向代理服务器的两倍以上，这大多归功于在内核中进行请求转发的较低开销。但是一旦请求的内容过大时，不论是基于反向代理还是NAT，负载均衡的整体吞吐量都差距不大，这说明对于一睦开销较大的内容，使用简单的反向代理来搭建负载均衡系统是值考虑的。

这么强大的系统还是有它的瓶颈，那就是NAT服务器的网络带宽，包括内部网络和外部网络。当然如果你不差钱，可以去花钱去购买千兆交换机或万兆交换机，甚至负载均衡硬件设备，但如果你是个屌丝，咋办?

一个简单有效的办法就是将基于NAT的集群和前面的DNS混合使用，比如 5 个100Mbps出口宽带的集群，然后通过DNS来将用户请求均衡地指向这些集群，同时，你还可以利用DNS智能解析实现地域就近访问。这样的配置对于大多数业务是足够了，但是对于提供下载或视频等服务的大规模站点，NAT服务器还是不够出色。

五、直接路由(LVS-DR)

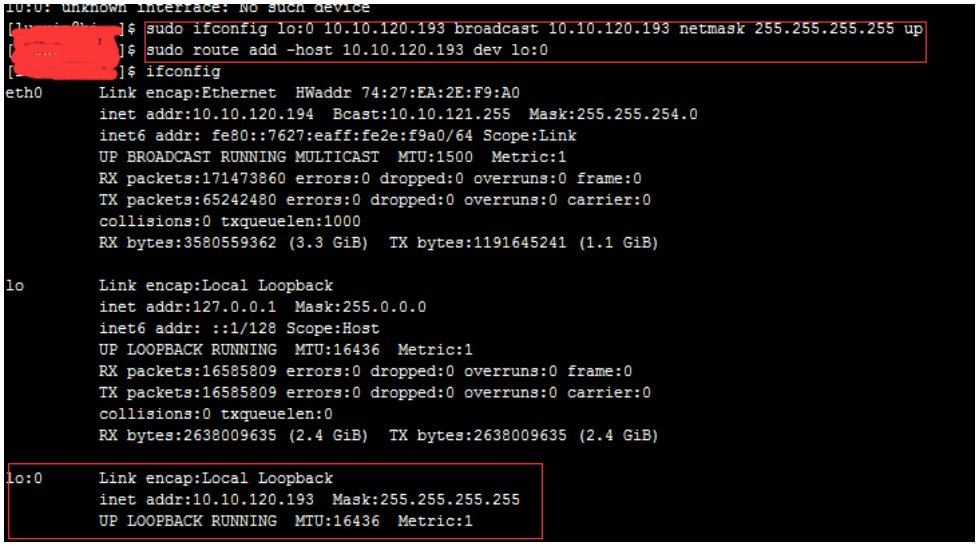
NAT是工作在网络分层模型的传输层（第四层），而直接路由是工作在数据链路层（第二层），貌似更屌些。它通过修改数据包的目标MAC地址（没有修改目标IP），将数据包转发到实际服务器上，不同的是，实际服务器的响应数据包将直接发送给客户崃，而不经过调度器。

1、网络设置

这里假设一台负载均衡调度器，两台实际服务器，购买三个外网ip，一台机一个，三台机的默认网关需要相同，最后再设置同样的ip别名，这里假设别名为10.10.120.193。这样一来，将通过10.10.120.193这个IP别名来访问调度器，你可以将站点的域名指向这个IP别名。

2、将ip别名添加到回环接口lo上

这是为了让实际服务器不要去寻找其他拥有这个IP别名的服务器，在实际服务器中运行：



另外还要防止实际服务器响应来自网络中针对IP别名的ARP广播，为此还要执行：

echo "1" > /proc/sys/net/ipv4/conf/lo/arp_ignore

echo "2" > /proc/sys/net/ipv4/conf/lo/arp_announce

echo "1" > /proc/sys/net/ipv4/conf/all/arp_ignore

echo "1" > /proc/sys/net/ipv4/conf/all/arp_announce

配置完了就可以使用ipvsadm配置LVS-DR集群了

ipvsadm -A -t 10.10.120.193:80 -s rr

ipvsadm -a -t 10.10.120.193:80 -r 10.10.120.210:8000 -g

ipvsadm -a -t 10.10.120.193:80 -r 10.10.120.211:8000 -g

-g 就意味着使用直接路由的方式转发数据包

LVS-DR 相较于LVS-NAT的最大优势在于LVS-DR不受调度器宽带的限制，例如假设三台服务器在WAN交换机出口宽带都限制为10Mbps，只要对于连接调度器和两台实际服务器的LAN交换机没有限速，那么，使用LVS-DR理论上可以达到20Mbps的最大出口宽带，因为它的实际服务器的响应数据包可以不经过调度器而直接发往用户端啊，所以它与调度器的出口宽带没有关系，只能自身的有关系。而如果使用LVS-NAT，集群只能最大使用10Mbps的宽带。所以，越是响应数据包远远超过请求数据包的服务，就越应该降低调度器转移请求的开销，也就越能提高整体的扩展能力，最终也就越依赖于WAN出口宽带。

总的来说，LVS-DR适合搭建可扩展的负载均衡系统，不论是Web服务器还是文件服务器，以及视频服务器，它都拥有出色的性能。前提是你必须为实际器购买一系列的合法IP地址。

六、IP隧道(LVS-TUN)

基于IP隧道的请求转发机制：将调度器收到的IP数据包封装在一个新的IP数据包中，转交给实际服务器，然后实际服务器的响应数据包可以直接到达用户端。目前Linux大多支持，可以用LVS来实现，称为LVS-TUN，与LVS-DR不同的是，实际服务器可以和调度器不在同一个WANt网段，调度器通过IP隧道技术来转发请求到实际服务器，所以实际服务器也必须拥有合法的IP地址。

总体来说，LVS-DR和LVS-TUN都适合响应和请求不对称的Web服务器，如何从它们中做出选择，取决于你的网络部署需要，因为LVS-TUN可以将实际服务器根据需要部署在不同的地域，并且根据就近访问的原则来转移请求，所以有类似这种需求的，就应该选择LVS-TUN。

Tomcat并发

Tomcat的性能与最大并发配置

当一个进程有 500 个线程在跑的话，那性能已经是很低很低了。Tomcat 默认配置的最大请求数是 150，也就是说同时支持 150 个并发，当然了，也可以将其改大。

当某个应用拥有 250 个以上并发的时候，应考虑应用服务器的集群。

具体能承载多少并发，需要看硬件的配置，CPU 越多性能越高，分配给 JVM 的内存越多性能也就越高，但也会加重 GC 的负担。
操作系统对于进程中的线程数有一定的限制：
Windows 每个进程中的线程数不允许超过 2000
Linux 每个进程中的线程数不允许超过 1000
另外，在 Java 中每开启一个线程需要耗用 1MB 的 JVM 内存空间用于作为线程栈之用。
Tomcat的最大并发数是可以配置的，实际运用中，最大并发数与硬件性能和CPU数量都有很大关系的。更好的硬件，更多的处理器都会使Tomcat支持更多的并发。

Tomcat 默认的 HTTP 实现是采用阻塞式的 Socket 通信，每个请求都需要创建一个线程处理。这种模式下的并发量受到线程数的限制，但对于 Tomcat 来说几乎没有 BUG 存在了。

Tomcat 还可以配置 NIO 方式的 Socket 通信，在性能上高于阻塞式的，每个请求也不需要创建一个线程进行处理，并发能力比前者高。但没有阻塞式的成熟。

这个并发能力还与应用的逻辑密切相关，如果逻辑很复杂需要大量的计算，那并发能力势必会下降。如果每个请求都含有很多的数据库操作，那么对于数据库的性能也是非常高的。

对于单台数据库服务器来说，允许客户端的连接数量是有限制的。

并发能力问题涉及整个系统架构和业务逻辑。

系统环境不同，Tomcat版本不同、JDK版本不同、以及修改的设定参数不同。并发量的差异还是满大的。

maxThreads="1000" 最大并发数

minSpareThreads="100"///初始化时创建的线程数

maxSpareThreads="500"///一旦创建的线程超过这个值，Tomcat就会关闭不再需要的socket线程。

acceptCount="700"// 指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理

配置实例：

<Connector port="8080" protocol="HTTP/1.1" minSpareThreads="100" maxSpareThreads="500" maxThreads="1000" acceptCount="700"

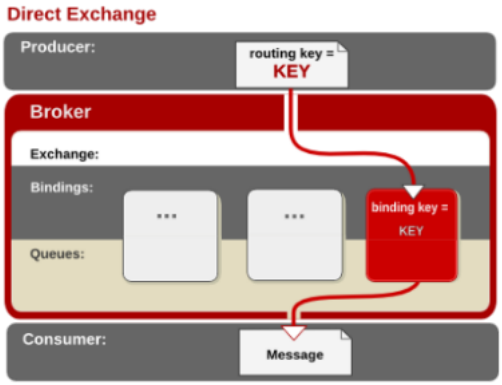
```
    connectionTimeout="20000"
    redirectPort="8443" />
```

Exchange 类型

Exchange 分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers 。headers 匹配 AMQP 消息的 header 而不是路由键，此外 headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以直接看另外三种类型：

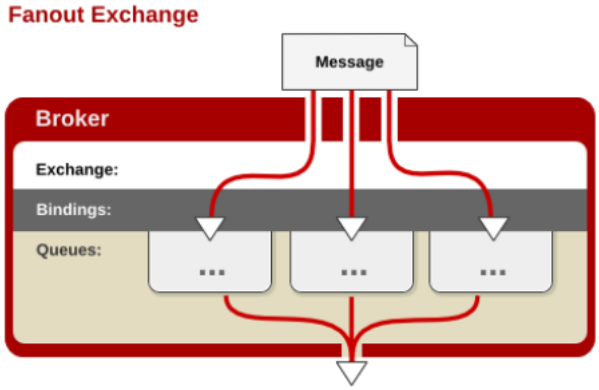
Direct键（routing key）分布：

1. Direct：消息中的路由键（routing key）如果和 Binding 中的 binding key 一致，交换器就将消息发到对应的队列中。它是完全匹配、单播的模式。



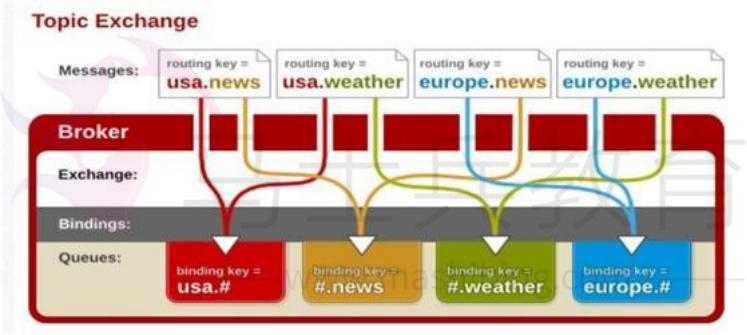
Fanout（广播分发）

2. Fanout：每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 类型转发消息是最快的。



topic交换器（模式匹配）

3. topic 交换器：topic 交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。它同样也会识别两个通配符：符号“#”和符号“”。#匹配 0 个或多个单词，匹配不多不少一个单词。



base概念

base 是分布式、面向列的开源数据库（其实准确的说是面向列族）。HDFS 为 Hbase 提供可靠的底层数据存储服务，MapReduce 为 Hbase 提供高性能的计算能力，Zookeeper 为 Hbase 提供稳定服务和 Failover 机制，因此我们说 Hbase 是一个通过大量廉价的机器解决海量数据的高速存储和读取的分布式数据库解决方案。

列式存储

列方式所带来的重要好处之一就是，由于查询中的选择规则是通过列来定义的，因此整个数据库是自动索引化的。

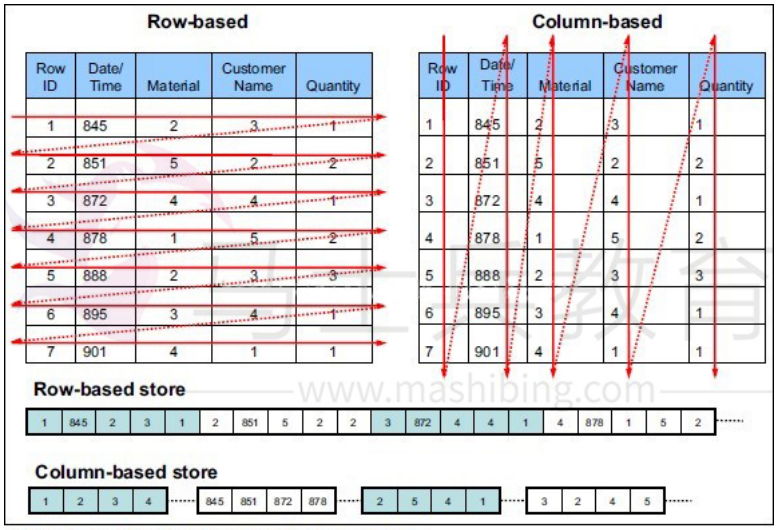


Figure 1-4 Row-based and column-based storage models

这里的列式存储其实说的是列族存储，Hbase 是根据列族来存储数据的。列族下面可以有非常多的列，列族在创建表的时候就必须指定。为了加深对 Hbase 列族的理解，下面是一个简单的关系型数据库的表和 Hbase 数据库的表：

RDBMS表			Hbase表		
Primary key	column1	column2	Rowkey	CF1	CF2
记录1	xxx	xxx	记录1	列1....列n	列1列2列3
记录2	xxx	xxx	记录2	列1 列2	
记录3	xxxx	xxxx	记录3	列1...列5	列1

Hbase 核心概念

Column Family列族

Column Family 又叫列族，Hbase 通过列族划分数据的存储，列族下面可以包含任意多的列，实现灵活的数据存取。Hbase 表的创建的时候就必须指定列族。就像关系型数据库创建的时候必须指定具体的列是一样的。Hbase的列族不是越多越好，官方推荐的是列族最好小于或者等于3。我们使用的场景一般是 1 个列族。

Rowkey（Rowkey查询，Rowkey 范围扫描，全表扫描）

Rowkey 的概念和 mysql 中的主键是完全一样的，Hbase 使用 Rowkey 来唯一的区分某一行的数据。Hbase 只支持 3 中查询方式：基于 Rowkey 的单行查询，基于 Rowkey 的范围扫描，全表扫描。

Region分区

§ **Region**：Region 的概念和关系型数据库的分区或者分片差不多。Hbase 会将一个大表的数据基于 Rowkey 的不同范围分配到不通的 Region 中，每个 Region 负责一定范围的数据访问和存储。这样即使是一张巨大的表，由于被切割到不通的 region，访问起来的时延也很低。

TimeStamp多版本

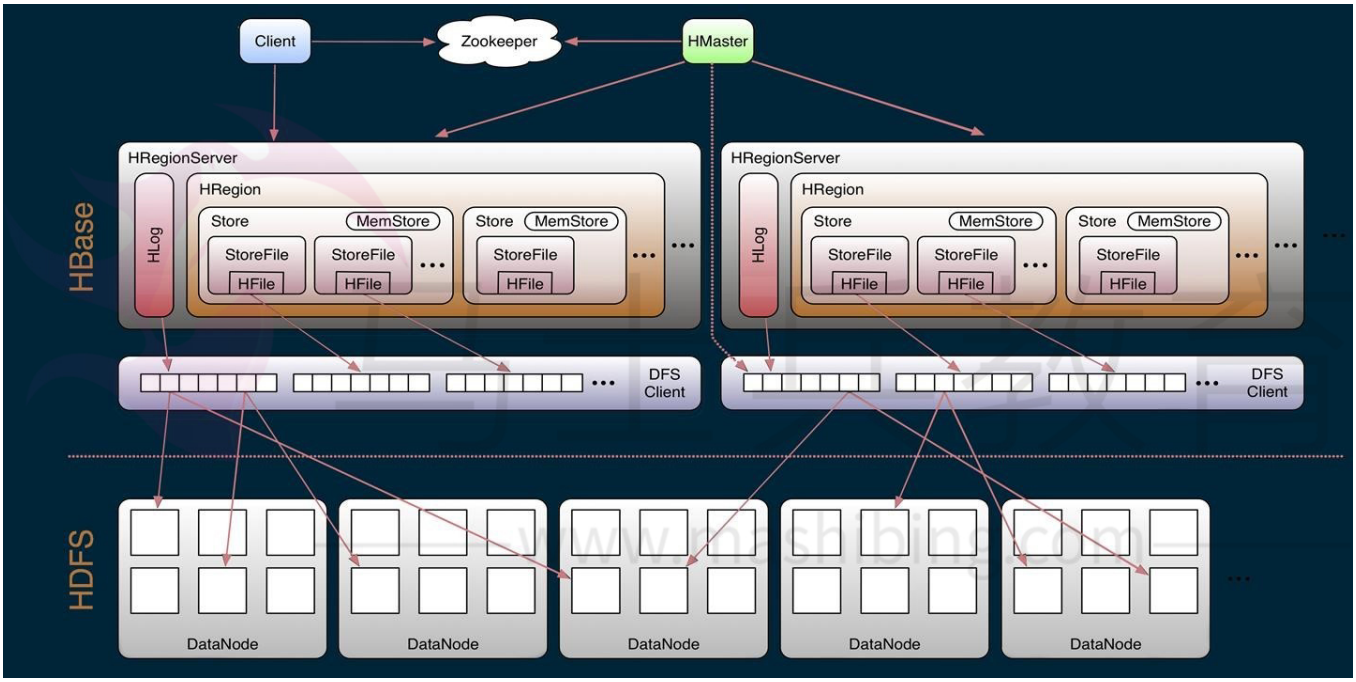
§ TimeStamp 是实现 Hbase 多版本的关键。在 Hbase 中使用不同的 timestame 来标识相同 rowkey 行对应的不通版本的数据。在写入数据的时候，如果用户没有指定对应的

timestamp，Hbase 会自动添加一个 timestamp，timestamp 和服务器时间保持一致。在

Hbase 中，相同 rowkey 的数据按照 timestamp 倒序排列。默认查询的是最新的版本，用户可同指定 timestamp 的值来读取旧版本的数据。

Hbase 核心架构

Hbase 是由 Client、Zookeeper、Master、HRegionServer、HDFS 等几个组建组成。



Client:

§ Client 包含了访问 Hbase 的接口，另外 Client 还维护了对应的 cache 来加速 Hbase 的访问，比如 cache 的.META.元数据的信息。

Zookeeper:

§ Hbase 通过 Zookeeper 来做 master 的高可用、RegionServer 的监控、元数据的入口以及集群配置的维护等工作。具体工作如下：

1. 通过 Zoopkeeper 来保证集群中只有 1 个 master 在运行，如果 master 异常，会通过竞争机制产生新的 master 提供服务
2. 通过 Zoopkeeper 来监控 RegionServer 的状态，当 RegionSevrer 有异常的时候，通过回调的形式通知 Master RegionServer 上下限的信息
3. 通过 Zoopkeeper 存储元数据的统一入口地址。

14.1.4.3.Hmaster

§ master 节点的主要职责如下：

1. 为 RegionServer 分配 Region
2. 维护整个集群的负载均衡
3. 维护集群的元数据信息发现失效的 Region，并将失效的 Region 分配到正常 RegionServer 上当 RegionSever 失效的时候，协调对应 Hlog 的拆分

HregionServer

§ HregionServer 直接对接用户的读写请求，是真正的"干活"的节点。它的功能概括如下：

\1. 管理 master 为其分配的 Region

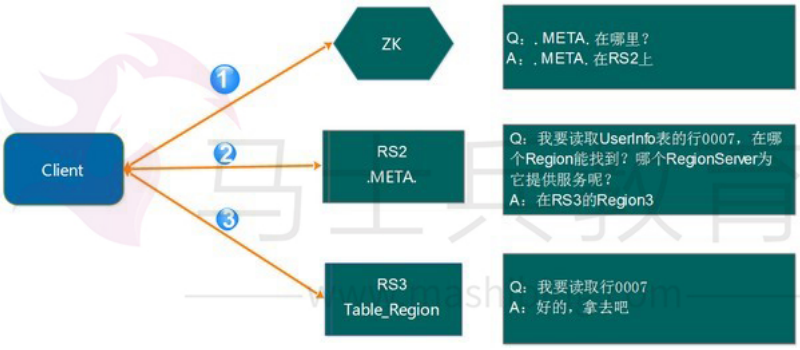
2. 处理来自客户端的读写请求
3. 负责和底层 HDFS 的交互，存储数据到 HDFS
4. 负责 Region 变大以后的拆分
5. 负责 Storefile 的合并工作

Region寻址方式（通过 zookeeper .META）

第 1 步：Client 请求 ZK 获取.META.所在的 RegionServer 的地址。

第 2 步：Client 请求.META.所在的 RegionServer 获取访问数据所在的 RegionServer 地址，client 会将.META.的相关信息 cache 下来，以便下一次快速访问。

第 3 步：Client 请求数据所在的 RegionServer，获取所需要的数据。

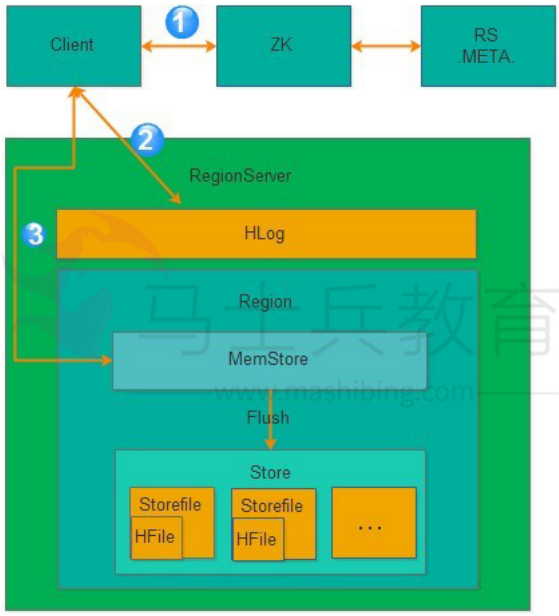


HDFS

§ HDFS 为 Hbase 提供最终的底层数据存储服务，同时为 Hbase 提供高可用（Hlog 存储在 HDFS）的支持。

Hbase 的写逻辑

Hbase的写入流程



从上图可以看出氛围 3 步骤：

获取**RegionServer** 第 1 步：Client 获取数据写入的 Region 所在的 RegionServer 请求写**Hlog** 第 2 步：请求写 Hlog, Hlog 存储在 HDFS，当 RegionServer 出现异常，需要使用 Hlog 来恢复数据。

请求写**MemStore** 第 3 步：请求写 MemStore,只有当写 Hlog 和写 MemStore 都成功了才算请求写入完成。

MemStore 后续会逐渐刷到 HDFS 中。

MemStore刷盘

为了提高 Hbase 的写入性能，当写请求写入 MemStore 后，不会立即刷盘。而是会等到一定的时候进行刷盘的操作。具体是哪些场景会触发刷盘的操作呢？总结成如下的几个场景：

全局内存控制

1. 这个全局的参数是控制内存整体的使用情况，当所有 memstore 占整个 heap 的最大比例的时候，会触发刷盘的操作。这个参数是 hbase.regionserver.global.memstore.upperLimit，默认为整个 heap 内存的 40%。

   但这并不意味着全局内存触发的刷盘操作会将所有的 MemStore 都进行输盘，而是通过

   另外一个参数 hbase.regionserver.global.memstore.lowerLimit 来控制，默认是整个 heap 内存的 35%。当 flush 到所有 memstore 占整个 heap 内存的比率为 35%的时候，就停止刷盘。这么做主要是为了减少刷盘对业务带来的影响，实现平滑系统负载的目的。

MemStore达到上限

2. 当 MemStore 的大小达到 hbase.hregion.memstore.flush.size 大小的时候会触发刷盘，默认 128M 大小

RegionServer 的**Hlog** 数量达到上限

3. 前面说到 Hlog 为了保证 Hbase 数据的一致性，那么如果 Hlog 太多的话，会导致故障恢复的时间太长，因此 Hbase 会对 Hlog 的最大个数做限制。当达到 Hlog 的最大个数的时候，会强制刷盘。这个参数是 hase.regionserver.max.logs，默认是 32 个。手工触发
4. 可以通过 hbase shell 或者 java api 手工触发 flush 的操作。

关闭**RegionServer**触发

5. 在正常关闭 RegionServer 会触发刷盘的操作，全部数据刷盘后就不需要再使用 Hlog 恢复数据。

Region 使用**HLOG** 恢复完数据后触发

6. ：当 RegionServer 出现故障的时候，其上面的 Region 会迁移到其他正常的 RegionServer 上，在恢复完 Region 的数据后，会触发刷盘，当刷盘完成后才会提供给业务访问。

HBase vs Cassandra

	HBase	Cassandra
语言	Java	Java
出发点	BigTable	BigTable and Dynamo
License	Apache	Apache
Protocol	HTTP/REST (also Thrift)	Custom, binary (Thrift)
数据分布	表划分为多个 region 存在不同 region server 上	改进的一致性哈希（虚拟节点）
存储目标	大文件	小文件
一致性	强一致性	最终一致性，Quorum NRW 策略
架构	master/slave	p2p
高可用性	NameNode 是 HDFS 的单点故障点	P2P 和去中心化设计，不会出现单点故障
伸缩性	Region Server 扩容，通过将自身发布到 Master，Master 均匀分布 Region	扩容需在 Hash Ring 上多个节点间调整数据分布
读写性能	数据读写定位可能要通过最多 6 次的网络 RPC，性能较低。	数据读写定位非常快
数据冲突处理	乐观并发控制（optimistic concurrency control）	向量时钟
临时故障处理	Region Server 宕机，重做 HLog	数据回传机制：某节点宕机，hash 到该节点的新数据自动路由到下一节点做 hinted handoff，源节点恢复后，推送回源节点。
永久故障恢复	Region Server 恢复，master 重新给其分配 region	Merkle 哈希树，通过 Gossip 协议同步 Merkle Tree，维 护集群节点间的数据一致性
成员通信及错误检测	Zookeeper	基于 Gossip
CAP	1，强一致性，0 数据丢失。2，可用性低。3，扩容方便。	1，弱一致性，数据可能丢失。2，可用性高。3，扩容方便。

概念

MongoDB 是由 C++语言编写的，是一个基于分布式文件存储的开源数据库系统。在高负载的情况下，添加更多的节点，可以保证服务器性能。MongoDB 旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。

MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数组。



特点

- MongoDB 是一个面向文档存储的数据库，操作起来比较简单和容易。
- 你可以在 MongoDB 记录中设置任何属性的索引 (如：FirstName="Sameer",Address="8 Ga ndhi Road")来实现更快的排序。
- 你可以通过本地或者网络创建数据镜像，这使得 MongoDB 有更强的扩展性。
- 如果负载的增加（需要更多的存储空间和更强的处理能力），它可以分布在计算机网络中的其他节点上这就是所谓的分片。
- Mongo 支持丰富的查询表达式。查询指令使用 JSON 形式的标记，可轻易查询文档中内嵌的对象及数组。
- MongoDb 使用 update()命令可以实现替换完成的文档（数据）或者一些指定的数据字段 。
- Mongodb 中的 Map/reduce 主要是用来对数据进行批量处理和聚合操作。
- Map 和 Reduce。Map 函数调用 emit(key,value)遍历集合中所有的记录，将 key 与 value 传给 Reduce 函数进行处理。
- Map 函数和 Reduce 函数是使用 Javascript 编写的，并可以通过 db.runCommand 或 mapre duce 命令来执行 MapReduce 操作。
- GridFS 是 MongoDB 中的一个内置功能，可以用于存放大量小文件。

MongoDB 允许在服务端执行脚本，可以用 Javascript 编写某个函数，直接在服务端执行，也可以把函数的定义存储在服务端，下次直接调用即可。

概念

Apache Cassandra 是高度可扩展的，高性能的分布式 NoSQL 数据库。 Cassandra 旨在处理许多商品服务器上的大量数据，提供高可用性而无需担心单点故障。

Cassandra 具有能够处理大量数据的分布式架构。 数据放置在具有多个复制因子的不同机器上，以获得高可用性，而无需担心单点故障。

数据模型

Key Space（对应 **SQL** 数据库中的 **database**）

1. 一个 Key Space 中可包含若干个 CF，如同 SQL 数据库中一个 database 可包含多个 table

Key（对应 **SQL** 数据库中的主键）

2. 在 Cassandra 中，每一行数据记录是以 key/value 的形式存储的，其中 key 是唯一标识。

column（对应 **SQL** 数据库中的列）

3. Cassandra 中每个 key/value 对中的 value 又称为 column，它是一个三元组，即：name， value 和 timestamp，其中 name 需要是唯一的。

super column（**SQL** 数据库不支持）

4. cassandra 允许 key/value 中的 value 是一个 map(key/value_list)，即某个 column 有多个子列。

Standard Column Family（相对应 **SQL** 数据库中的 **table**）

5. 每个 CF 由一系列 row 组成，每个 row 包含一个 key 以及其对应的若干 column。

Super Column Family（**SQL** 数据库不支持）

6. 每个 SCF 由一系列 row 组成，每个 row 包含一个 key 以及其对应的若干 super column。

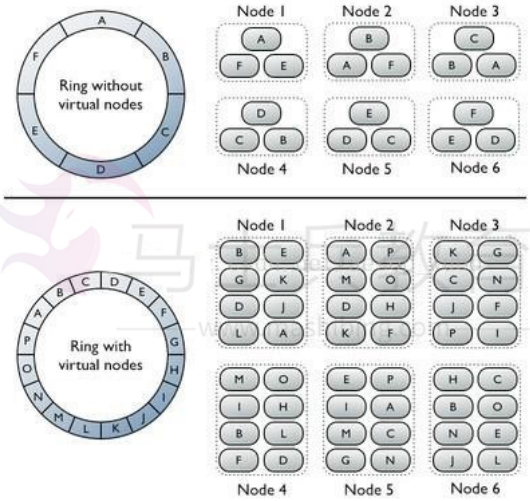
Cassandra 一致 Hash 和虚拟节点

一致性 **Hash**（多米诺 **down** 机）

为每个节点分配一个 token，根据这个 token 值来决定节点在集群中的位置以及这个节点所存储的数据范围。

虚拟节点（**down** 机多节点托管）

由于这种方式会造成数据分布不均的问题，在 Cassandra1.2 以后采用了虚拟节点的思想：不需要为每个节点分配 token，把圆环分成更多部分，让每个节点负责多个部分的数据，这样一个节点移除后，它所负责的多个 token 会托管给多个节点处理，这种思想解决了数据分布不均的问题。



如图所示，上面部分是标准一致性哈希，每个节点负责圆环中连续的一段，如果 Node2 突然 down 掉，Node2 负责的数据托管给 Node1，即 Node1 负责 EFAB 四段，如果 Node1 里面有很多热点用户产生的数据导致 Node1 已经有点撑不住了，恰巧 B 也是热点用户产生的数据，这样一来 Node1 可能会接着 down 机，Node1down 机，Node6 还 hold 住吗?

下面部分是虚拟节点实现，每个节点不再负责连续部分，且圆环被分为更多的部分。如果 Node2 突然down掉，Node2负责的数据不全是托管给Node1，而是托管给多个节点。而且也保持了一致性哈希的特点。

Gossip** 协议

Gossip 算法如其名，灵感来自办公室八卦，只要一个人八卦一下，在有限的时间内所有的人都会知道该八卦的信息，这种方式也与病毒传播类似，因此 Gossip 有众多的别名“闲话算法”、

“疫情传播算法”、“病毒感染算法”、“谣言传播算法”。 Gossip 的特点：在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致。因为 Gossip 不要求节点知道所有其他节点，因此又具有去中心化的特点，节点之间完全对等，

不需要任何的中心节点。实际上 Gossip 可以用于众多能接受“最终一致性”的领域：失败检测、路由同步、Pub/Sub、动态负载均衡。

Gossip 节点的通信方式及收敛性

Gossip 两个节点（**A**、**B**）之间存在三种通信方式（**push**、**pull**、**push&pull**）

1. push: A 节点将数据(key,value,version)及对应的版本号推送给 B 节点，B 节点更新 A 中比自己新的数据。
2. pull：A 仅将数据 key,version 推送给 B，B 将本地比 A 新的数据（Key,value,version）推送给 A，A 更新本地。
3. push/pull：与 pull 类似，只是多了一步，A 再将本地比 B 新的数据推送给 B，B 更新本地。

如果把两个节点数据同步一次定义为一个周期，则在一个周期内，push 需通信 1 次，pull 需 2 次， push/pull 则需 3 次，从效果上来讲，push/pull 最好，理论上一个周期内可以使两个节点完全一致。直观上也感觉，push/pull 的收敛速度是最快的。

gossip 的协议和**seed list**（防止集群分列）

cassandra 使用称为 gossip 的协议来发现加入 C 集群中的其他节点的位置和状态信息。gossip 进程每秒都在进行，并与至多三个节点交换状态信息。节点交换他们自己和所知道的信息，于是所有的节点很快就能学习到整个集群中的其他节点的信息。gossip 信息有一个相关的版本号，于是在一次gossip信息交换中，旧的信息会被新的信息覆盖重写。要阻止分区进行gossip交流，那么在集群中的所有节点中使用相同的 seed list，种子节点的指定除了启动起 gossip 进程外，没有其他的目的。种子节点不是一个单点故障，他们在集群操作中也没有其他的特殊目的，除了引导节点以外

数据复制

Partitioners（计算 **primary key token** 的 **hash** 函数）在 Cassandra 中，table 的每行由唯一的 primarykey 标识，partitioner 实际上为一 hash 函数用以计算 primary key 的 token。Cassandra 依据这个 token 值在集群中放置对应的行

两种可用的复制策略：

SimpleStrategy：仅用于单数据中心，

将第一个 replica 放在由 partitioner 确定的节点中，其余的 replicas 放在上述节点顺时针方向的后续节点中。

NetworkTopologyStrategy：可用于较复杂的多数据中心。可以指定在每个数据中心分别存储多少份 replicas。

复制策略在创建 keyspace 时指定，如

CREATE KEYSPACE Excelsior WITH REPLICATION = { 'class' : 'SimpleStrategy','replication_factor' : 3 };

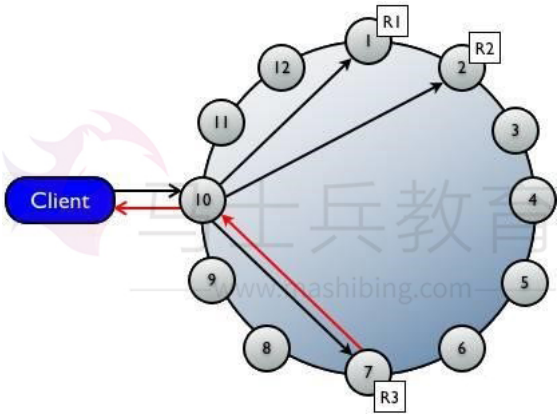
CREATE KEYSPACE Excalibur WITH REPLICATION = {'class' :'NetworkTopologyStrategy',

'dc1' : 3, 'dc2' : 2};

数据写请求和协调者

协调者**(coordinator)**

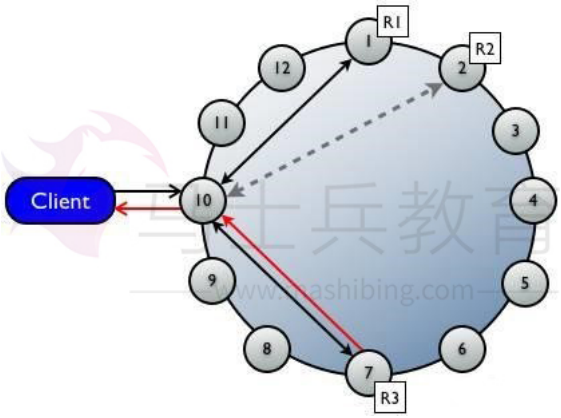
协调者(coordinator)将 write 请求发送到拥有对应row 的所有 replica 节点，只要节点可用便获取并执行写请求。写一致性级别(write consistency level)确定要有多少个 replica 节点必须返回成功的确认信息。成功意味着数据被正确写入了 commit log 和 memtable。



其中 dc1、dc2 这些数据中心名称要与 snitch 中配置的名称一致.上面的拓扑策略表示在 dc1 配置3 个副本,在 dc2 配置 2 个副本

数据读请求和后台修复

1. 协调者首先与一致性级别确定的所有 replica 联系，被联系的节点返回请求的数据。
2. 若多个节点被联系，则来自各 replica 的 row 会在内存中作比较，若不一致，则协调者使用含最新数据的 replica 向 client 返回结果。那么比较操作过程中只需要传递时间戳就可以,因为要比较的只是哪个副本数据是最新的。
3. 协调者在后台联系和比较来自其余拥有对应 row 的 replica 的数据，若不一致，会向过时的 replica 发写请求用最新的数据进行更新 read repair。



数据存储（CommitLog、MemTable、SSTable）

写请求分别到 CommitLog 和 MemTable, 并且 MemTable 的数据会刷写到磁盘 SSTable 上. 除了写数据,还有索引也会保存到磁盘上.

先将数据写到磁盘中的 commitlog，同时追加到中内存中的数据结构 memtable 。这个时候就会返回客户端状态， memtable 内容超出指定容量后会被放进将被刷入磁盘的队列

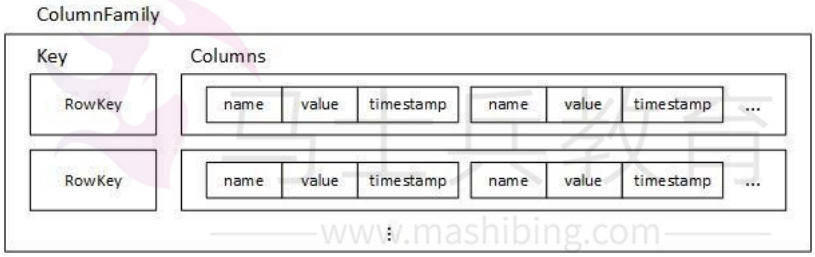
(memtable_flush_queue_size 配置队列长度)。若将被刷入磁盘的数据超出了队列长度，将内存数据刷进磁盘中的 SSTable,之后 commit log 被清空。

SSTable 文件构成（**BloomFilter**、**index**、**data**、**static**）

SSTable 文件有fileer（判断数据key是否存在，这里使用了BloomFilter提高效率），index（寻找对应 column 值所在 data 文件位置）文件，data（存储真实数据）文件，static（存储和统计 column 和 row 大小）文件。

二级索引（对要索引的 value 摘要，生成 RowKey）

在 Cassandra 中，数据都是以 Key-value 的形式保存的。



KeysIndex 所创建的二级索引也被保存在一张 ColumnFamily 中。在插入数据时，对需要进行索引的value进行摘要，生成独一无二的key，将其作为RowKey保存在索引的ColumnFamily中；同时在 RowKey 上添加一个 Column，将插入数据的 RowKey 作为 name 域的值，value 域则赋空值，timestamp 域则赋为插入数据的时间戳。

如果有相同的 value 被索引了，则会在索引 ColumnFamily 中相同的 RowKey 后再添加新的 Column。如果有新的 value 被索引，则会在索引 ColumnFamily 中添加新的 RowKey 以及对应新的 Column。

当对 value 进行查询时，只需计算该 value 的 RowKey，在索引 ColumnFamily 中的查找该

RowKey，对其 Columns 进行遍历就能得到该 value 所有数据的 RowKey。

数据读写

数据写入和更新（数据追加）

Cassandra 的设计思路与这些系统不同，无论是 insert 还是 remove 操作，都是在已有的数据后面进行追加，而不修改已有的数据。这种设计称为 Log structured 存储，顾名思义就是系统中的数据是以日志的形式存在的，所以只会将新的数据追加到已有数据的后面。Log structured 存储系统有两个主要优点：

数据的写和删除效率极高

- 传统的存储系统需要更新元信息和数据，因此磁盘的磁头需要反复移动，这是一个比较耗时

的操作，而 Log structured 的系统则是顺序写，可以充分利用文件系统的 cache，所以效率很高。

错误恢复简单

- 由于数据本身就是以日志形式保存，老的数据不会被覆盖，所以在设计 journal 的时候不需要考虑 undo，简化了错误恢复。

读的复杂度高

- 但是，Log structured 的存储系统也引入了一个重要的问题：读的复杂度和性能。理论上说，读操作需要从后往前扫描数据，以找到某个记录的最新版本。相比传统的存储系统，这是比较耗时的。

数据删除（**column** 的墓碑）

如果一次删除操作在一个节点上失败了（总共 3 个节点，副本为 3， RF=3).整个删除操作仍然被认为成功的（因为有两个节点应答成功，使用 CL.QUORUM 一致性）。接下来如果读发生在该节点上就会变的不明确，因为结果返回是空，还是返回数据，没有办法确定哪一种是正确的。

Cassandra 总是认为返回数据是对的，那就会发生删除的数据又出现了的事情，这些数据可以叫” 僵尸”，并且他们的表现是不可预见的。

墓碑

删除一个 column 其实只是插入一个关于这个 column 的墓碑（tombstone），并不直接删除原有的 column。该墓碑被作为对该 CF 的一次修改记录在 Memtable 和 SSTable 中。墓碑的内容是删除请求被执行的时间，该时间是接受客户端请求的存储节点在执行该请求时的本地时间（local delete time），称为本地删除时间。需要注意区分本地删除时间和时间戳，每个 CF 修改记录都有一个时间戳，这个时间戳可以理解为该 column 的修改时间，是由客户端给定的。

垃圾回收**compaction**

由于被删除的 column 并不会立即被从磁盘中删除，所以系统占用的磁盘空间会越来越大，这就需要有一种垃圾回收的机制，定期删除被标记了墓碑的 column。垃圾回收是在 compaction 的过程中完成的。

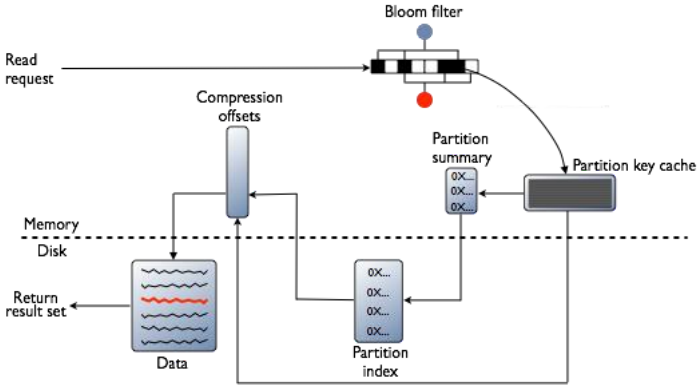
数据读取（**memtable+SStables**）

为了满足读 cassandra 读取的数据是 memtable 中的数据和 SStables 中数据的合并结果。读取 SSTables 中的数据就是查找到具体的哪些的 SSTables 以及数据在这些 SSTables 中的偏移量 (SSTables 是按主键排序后的数据块)。首先如果 row cache enable 了话，会检测缓存。缓存命中

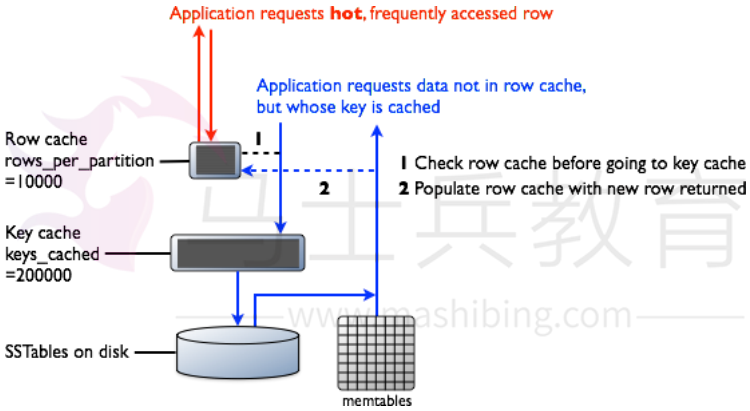
直接返回数据，没有则查找 Bloom filter，查找可能的 SSTable。然后有一层 Partition key cache，

找 partition key 的位置。如果有根据找到的 partition 去压缩偏移量映射表找具体的数据块。如果缓存没有，则要经过 Partition summary,Partition index 去找 partition key。然后经过压缩偏移量映射表找具体的数据块。

1. 检查 memtable
2. 如果 enabled 了,检查 row cache
3. 检查 Bloom filter
4. 如果 enable 了,检查 partition key 缓存
5. 如果在 partition key 缓存中找到了 partition key,直接去 compression offset 命中，如果没有，检查 partition summary
6. 根据 compression offset map 找到数据位置
7. 从磁盘的 SSTable 中取出数据



行缓存和键缓存请求流程图



MemTable：如果 memtable 有目标分区数据，这个数据会被读出来并且和从 SSTables 中读出来的数据进行合并。SSTable 的数据访问如下面所示的步骤。

Row Cache（**SSTables**中频繁被访问的数据）

在 Cassandra2.2+，它们被存储在堆外内存，使用全新的实现避免造成垃圾回收对 JVM 造成压力。

存在在 row cache 的子集数据可以在特定的一段时间内配置一定大小的内存。row cache 使用 LRU(least-recently-userd)进行回收在申请内存。存储在 row cache 中的数据是 SSTables 中频繁被访问的数据。存储到row cache中后，数据就可以被后续的查询访问。row cache不是写更新。如果写某行了，这行的缓存就会失效，并且不会被继续缓存，直到这行被读到。类似的，如果一个partition更新了，整个partition的cache都会被移除，但目标的数据在row cache中找不到，就会去检查 Bloom filter。

Bloom Filter（查找数据可能对应的**SSTable**）

首先，Cassandra 检查 Bloom filter 去发现哪个 SSTables 中有可能有请求的分区数据。Bloom filter 是存储在堆外内存。每个 SSTable 都有一个关联的 Bloom filter。一个 Bloom filter 可以建立一个 SSTable 没有包含的特定的分区数据。同样也可以找到分区数据存在 SSTable 中的可能性。它可以加速查找 partition key 的查找过程。然而，因为 Bloom filter 是一个概率函数，所以可能会得到错误的结果，并不是所有的 SSTables 都可以被 Bloom filter 识别出是否有数据。如果

Bloom filter 不能够查找到 SSTable，Cassandra 会检查 partition key cache。Bloom filter 大小增长很适宜，每 10 亿数据 1~2GB。在极端情况下，可以一个分区一行。都可以很轻松的将数十亿的 entries 存储在单个机器上。Bloom filter 是可以调节的，如果你愿意用内存来换取性能。

Partition Key Cache（查找数据可能对应的**Partition key**）

partition key 缓存如果开启了，将 partition index 存储在堆外内存。key cache 使用一小块可配置大小的内存。在读的过程中，每个"hit"保存一个检索。如果在 key cache 中找到了 partition key。就直接到 compression offset map 中招对应的块。partition key cache 热启动后工作的更好，相比较冷启动，有很大的性能提升。如果一个节点上的内存非常受限制，可能的话，需要限制保存在 key cache 中的 partition key 数目。如果一个在 key cache 中没有找到 partition key。就会去partition summary中去找。partition key cache 大小是可以配置的，意义就是存储在key cache 中的 partition keys 数目。

Partition Summary（内存中存储一些**partition index** 的样本）

partition summary 是存储在堆外内存的结构，存储一些 partition index 的样本。如果一个 partition index 包含所有的 partition keys。鉴于一个 partition summary 从每 X 个 keys 中取样，然后将每X个key map到index 文件中。例如，如果一个partition summary设置了20keys 进行取样。它就会存储 SSTable file 开始的一个 key,20th 个 key，以此类推。尽管并不知道 partition key 的具体位置，partition summary 可以缩短找到 partition 数据位置。当找到了 partition key 值可能的范围后，就会去找 partition index。通过配置取样频率，你可以用内存来换取性能，当 partition summary 包含的数据越多，使用的内存越多。可以通过表定义的 index interval 属性来改变样本频率。固定大小的内存可以通过 index_summary_capacity_in_mb 属性来设置，默认是堆大小的 5%。

Partition Index（磁盘中）

partition index 驻扎在磁盘中，索引所有 partition keys 和偏移量的映射。如果 partition summary 已经查到 partition keys 的范围，现在的检索就是根据这个范围值来检索目标 partition key。需要进行单次检索和顺序读。根据找到的信息。然后去 compression offset map 中去找磁盘中有这个数据的块。如果 partition index 必须要被检索，则需要检索两次磁盘去找到目标数据。

Compression offset map（磁盘中）

compression offset map 存储磁盘数据准确位置的指针。存储在堆外内存，可以被 partition key cache 或者 partition index 访问。一旦 compression offset map 识别出来磁盘中的数据位置，就会从正确的 SStable(s)中取出数据。查询就会收到结果集。

存储引擎

概念

数据库存储引擎是数据库底层软件组织，数据库管理系统（DBMS）使用数据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以 获得特定的功能。现在许多不同的数据库管理系统都支持多种不同的数据引擎。存储引擎主要有： 1. MyIsam , 2. InnoDB, 3. Memory, 4. Archive, 5. Federated 。

InnoDB（B+树）

InnoDB 底层存储结构为B+树， B树的每个节点对应innodb的一个page，page大小是固定的，一般设为 16k。其中非叶子节点只有键值，叶子节点包含完成数据。

适用场景：

1）经常更新的表，适合处理多重并发的更新请求。

2）支持事务。

3）可以从灾难中恢复（通过 bin-log 日志等）。

4）外键约束。只有他支持外键。

5）支持自动增加列属性 auto_increment。

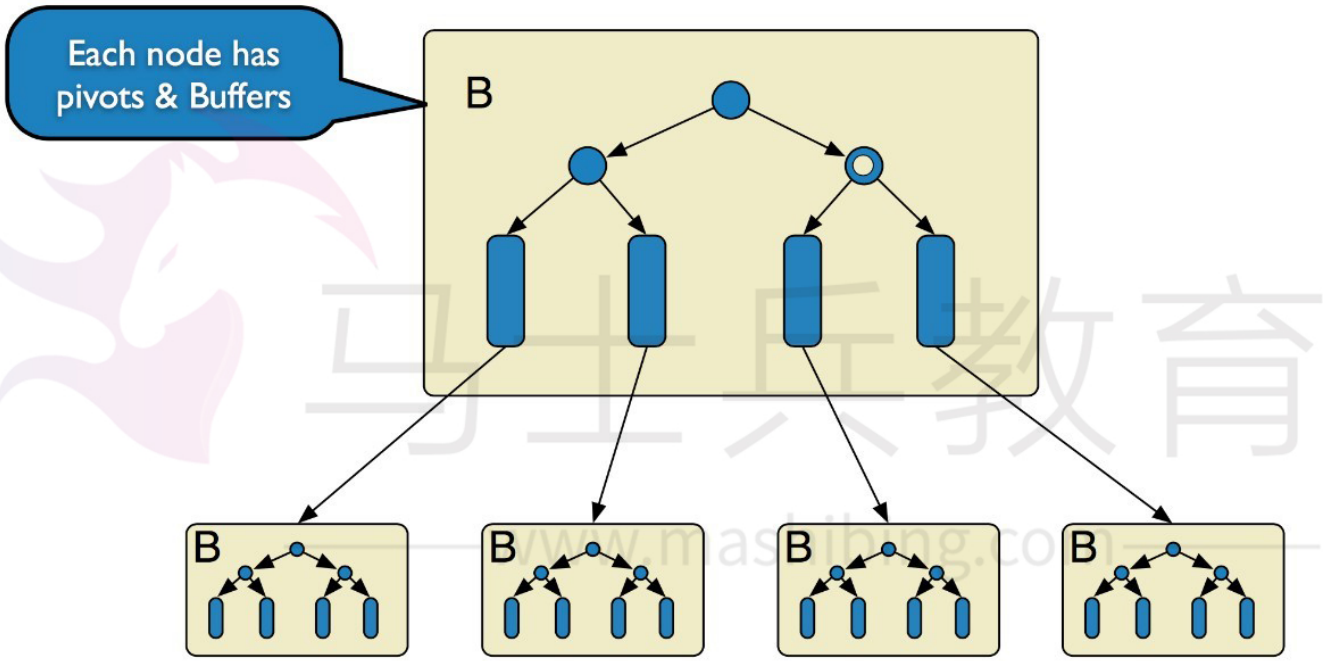
TokuDB（Fractal Tree-节点带数据）

TokuDB 底层存储结构为 Fractal Tree,Fractal Tree 的结构与 B+树有些类似, 在 Fractal Tree

中，每一个 child 指针除了需要指向一个 child 节点外，还会带有一个 Message Buffer ，这个

Message Buffer 是一个 FIFO 的队列，用来缓存更新操作。

例如，一次插入操作只需要落在某节点的 Message Buffer 就可以马上返回了，并不需要搜索到叶子节点。这些缓存的更新会在查询时或后台异步合并应用到对应的节点中。



TokuDB 在线添加索引，不影响读写操作, 非常快的写入性能， Fractal-tree 在事务实现上有优势。 他主要适用于访问频率不高的数据或历史数据归档。

MyIASM

MyIASM是MySQL默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁和外键，因此当 INSERT(插入)或 UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。

ISAM 执行读取操作的速度很快，而且不占用大量的内存和存储资源。在设计之初就预想数据组织成有固定长度的记录，按顺序存储的。---ISAM 是一种静态索引结构。

缺点是它不 支持事务处理。

Memory

Memory（也叫 HEAP）堆内存：使用存在内存中的内容来创建表。每个 MEMORY 表只实际对应一个磁盘文件。MEMORY 类型的表访问非常得快，因为它的数据是放在内存中的，并且默认使用 HASH 索引。但是一旦服务关闭，表中的数据就会丢失掉。 Memory 同时支持散列索引和 B 树索引，B树索引可以使用部分查询和通配查询，也可以使用<,>和>=等操作符方便数据挖掘，散列索引相等的比较快但是对于范围的比较慢很多。

索引

索引（Index）是帮助 MySQL 高效获取数据的数据结构。常见的查询算法,顺序查找,二分查找,二叉排序树查找,哈希散列法,分块查找,平衡多路搜索树 B 树（B-tree）

常见索引原则有

1.选择唯一性索引

1． 唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。

2.为经常需要排序、分组和联合操作的字段建立索引：

3．为常作为查询条件的字段建立索引。

4．限制索引的数目：

越多的索引，会使更新表变得很浪费时间。

尽量使用数据量少的索引

6. 如果索引的值很长，那么查询的速度会受到影响。

尽量使用前缀来索引

7. 如果索引字段的值很长，最好使用值的前缀来索引。

7. 删除不再使用或者很少使用的索引

8 . 最左前缀匹配原则，非常重要的原则。

10. 尽量选择区分度高的列作为索引

区分度的公式是表示字段不重复的比例

11.索引列不能参与计算，保持列“干净”：带函数的查询不参与索引。

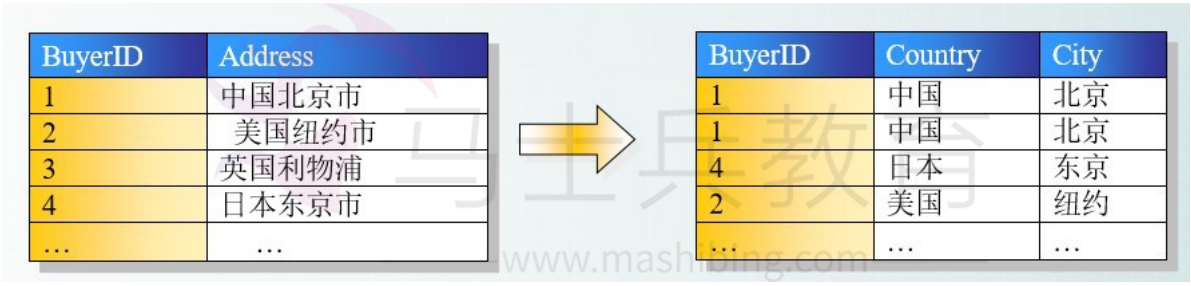
12.尽量的扩展索引，不要新建索引。

数据库三范式

范式是具有最小冗余的表结构。3 范式具体如下：

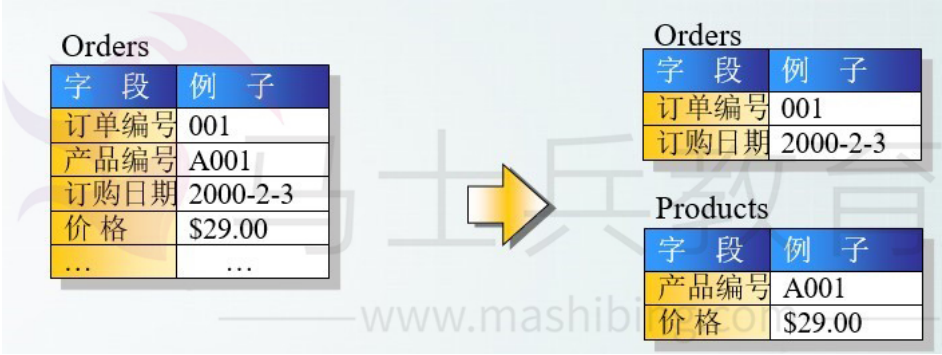
第一范式(1st NF - 列都是不可再分)

第一范式的目标是确保每列的原子性:如果每列都是不可再分的最小数据单元（也称为最小的原子单元），则满足第一范式（1NF）



第二范式(2nd NF - 每个表只描述一件事情)

首先满足第一范式，并且表中非主键列不存在对主键的部分依赖。 第二范式要求每个表只描述一件事情。



第三范式(3rd NF - 不存在对非主键列的传递依赖)

第三范式定义是，满足第二范式，并且表中的列不存在对非主键列的传递依赖。除了主键订单编号外，顾客姓名依赖于非主键顾客编号。



数据库是事务

事务(TRANSACTION)是作为单个逻辑工作单元执行的一系列操作，这些操作作为一个整体一起向系统提交，要么都执行、要么都不执行 。事务是一个不可分割的工作逻辑单元
事务必须具备以下四个属性，简称 ACID 属性：

原子性（**Atomicity**）

1. 事务是一个完整的操作。事务的各步操作是不可分的（原子的）；要么都执行，要么都不执行。

一致性（**Consistency**）

2. 当事务完成时，数据必须处于一致状态。

隔离性（**Isolation**）

3. 对数据进行修改的所有并发事务是彼此隔离的，这表明事务必须是独立的，它不应以任何方式依赖于或影响其他事务。

永久性（**Durability**）

4. 事务完成后，它对数据库的修改被永久保持，事务日志能够保持事务的永久性。

存储过程(特定功能的 SQL 语句集)

一组为了完成特定功能的 SQL 语句集，存储在数据库中，经过第一次编译后再次调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象。

存储过程优化思路：

1. 尽量利用一些 sql 语句来替代一些小循环，例如聚合函数，求平均函数等。
2. 中间结果存放于临时表，加索引。
3. 少使用游标。sql 是个集合语言，对于集合运算具有较高性能。而 cursors 是过程运算。比如对一个 100 万行的数据进行查询。游标需要读表 100 万次，而不使用游标则只需要少量几次读取。
4. 事务越短越好。sqlserver 支持并发操作。如果事务过多过长，或者隔离级别过高，都会造成并发操作的阻塞，死锁。导致查询极慢，cpu 占用率极地。
5. 使用 try-catch 处理错误异常。
6. 查找语句尽量不要放在循环内。

触发器(一段能自动执行的程序)

触发器是一段能自动执行的程序，是一种特殊的存储过程，触发器和普通的存储过程的区别是：触发器是当对某一个表进行操作时触发。诸如：update、insert、delete 这些操作的时候，系统会自动调用执行该表上对应的触发器。SQL Server 2005 中触发器可以分为两类：DML 触发器和 DDL 触发器，其中 DDL 触发器它们会影响多种数据定义语言语句而激发，这些语句有 create、 alter、drop 语句。

数据库并发策略

并发控制一般采用三种方法，分别是乐观锁和悲观锁以及时间戳。

乐观锁

乐观锁认为一个用户读数据的时候，别人不会去写自己所读的数据；悲观锁就刚好相反，觉得自己读数据库的时候，别人可能刚好在写自己刚读的数据，其实就是持一种比较保守的态度；时间戳就是不加锁，通过时间戳来控制并发出现的问题。

悲观锁

悲观锁就是在读取数据的时候，为了不让别人修改自己读取的数据，就会先对自己读取的数据加锁，只有自己把数据读完了，才允许别人修改那部分数据，或者反过来说，就是自己修改某条数据的时候，不允许别人读取该数据，只有等自己的整个事务提交了，才释放自己加上的锁，才允许其他用户访问那部分数据。

时间戳

时间戳就是在数据库表中单独加一列时间戳，比如“TimeStamp”，每次读出来的时候，把该字段也读出来，当写回去的时候，把该字段加1，提交之前 ，跟数据库的该字段比较一次，如果比数据库的值大的话，就允许保存，否则不允许保存，这种处理方法虽然不使用数据库系统提供的锁机制，但是这种方法可以大大提高数据库处理的并发量，

以上悲观锁所说的加“锁”，其实分为几种锁，分别是：排它锁（写锁）和共享锁（读锁）。

数据库锁

行级锁

行级锁是一种排他锁，防止其他事务修改此行；在使用以下语句时，Oracle 会自动应用行级锁：

1. INSERT、UPDATE、DELETE、SELECT ... FOR UPDATE [OF columns] [WAIT n | NOWAIT];
2. SELECT ... FOR UPDATE 语句允许用户一次锁定多条记录进行更新
3. 使用 COMMIT 或 ROLLBACK 语句释放锁。

表级锁

表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分 MySQL 引擎支持。最常使用的 MYISAM 与 INNODB 都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

页级锁

页级锁是 MySQL 中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB 支持页级锁

Semphore、CountDownLatch、CyclicBarrier、Phaser

并发工具类_同步屏障CyclicBarrier

简介

CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier默认的构造方法是CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。

实例代码如下：

```
public class CyclicBarrierTest {
    static CyclicBarrier c = new CyclicBarrier(2);

    public static void main(String[] args) {
    new Thread(new Runnable() {

    @Override
    public void run() {
        try {
            c.await();
        } catch (Exception e) {

        }
        System.out.println(1);
        }
        }).start();

        try {
            c.await();
        } catch (Exception e) {
        }
        System.out.println(2);
    }
}
输出
    2
    1
或输出
    1
    2
```

如果把new CyclicBarrier(2)修改成new CyclicBarrier(3)则主线程和子线程会永远等待，因为没有第三个线程执行await方法，即没有第三个线程到达屏障，所以之前到达屏障的两个线程都不会继续执行。

CyclicBarrier还提供一个更高级的构造函数CyclicBarrier(int parties，Runnable barrierAction)，用于在线程到达屏障时，优先执行barrierAction，方便处理更复杂的业务场景。代码如下：

```
public class CyclicBarrierTest2 {

    static CyclicBarrier c = new CyclicBarrier(2, new A());
    public static void main(String[] args) {
        new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                c.await();
            } catch (Exception e) {
            }
            System.out.println(1);
        }
        }).start();
        try {
            c.await();
        } catch (Exception e) {
        }
        System.out.println(2);
    }
    static class A implements Runnable {

        @Override
        public void run() {
            System.out.println(3);

        }
    }
}

输出

3
1
2
```

CyclicBarrier的应用场景

CyclicBarrier可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个Excel保存了用户所有银行流水，每个Sheet保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后，再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水。

CyclicBarrier和CountDownLatch的区别

- CountDownLatch的计数器只能使用一次。而CyclicBarrier的计数器可以使用reset() 方法重置。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。
- CyclicBarrier还提供其他有用的方法，比如getNumberWaiting方法可以获得CyclicBarrier阻塞的线程数量。isBroken方法用来知道阻塞的线程是否被中断。比如以下代码执行完之后会返回true。

isBroken的使用代码如下：

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierTest3 {

    static CyclicBarrier c = new CyclicBarrier(2);

    public static void main(String[] args) throws InterruptedException, BrokenBarrierException {
            Thread thread = new Thread(new Runnable() {

            @Override
            public void run() {
                    try {
                         c.await();
                    } catch (Exception e) {
                    }
             }
        });
        thread.start();
        thread.interrupt();
        try {
            c.await();
        } catch (Exception e) {
            System.out.println(c.isBroken());
        }
    }
}

输出
true
```

并发工具类_控制并发线程数的Semaphore

简介

Semaphore（信号量）是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源。很多年以来，我都觉得从字面上很难理解Semaphore所表达的含义，只能把它比作是控制流量的红绿灯，比如XX马路要限制流量，只允许同时有一百辆车在这条路上行使，其他的都必须在路口等待，所以前一百辆车会看到绿灯，可以开进这条马路，后面的车会看到红灯，不能驶入XX马路，但是如果前一百辆中有五辆车已经离开了XX马路，那么后面就允许有5辆车驶入马路，这个例子里说的车就是线程，驶入马路就表示线程在执行，离开马路就表示线程执行完成，看见红灯就表示线程被阻塞，不能执行。

应用场景

Semaphore可以用于做流量控制，特别公用资源有限的应用场景，比如数据库连接。假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发的读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有十个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，我们就可以使用Semaphore来做流控，代码如下：

```
public class SemaphoreTest {

    private static final int THREAD_COUNT = 30;

    private static ExecutorService threadPool = Executors
            .newFixedThreadPool(THREAD_COUNT);

    private static Semaphore s = new Semaphore(10);

    public static void main(String[] args) {
        for (int i = 0; i < THREAD_COUNT; i++) {
            threadPool.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        s.acquire();
                        System.out.println("save data");
                        s.release();
                    } catch (InterruptedException e) {
                    }
                }
            });
        }

        threadPool.shutdown();
```

```
    }
}
```

在代码中，虽然有30个线程在执行，但是只允许10个并发的执行。Semaphore的构造方法Semaphore(int permits) 接受一个整型的数字，表示可用的许可证数量。Semaphore(10)表示允许10个线程获取许可证，也就是最大并发数是10。Semaphore的用法也很简单，首先线程使用Semaphore的acquire()获取一个许可证，使用完之后调用release()归还许可证。还可以用tryAcquire()方法尝试获取许可证。

其他方法

Semaphore还提供一些其他方法：

- int availablePermits()：返回此信号量中当前可用的许可证数。
- int getQueueLength()：返回正在等待获取许可证的线程数。
- boolean hasQueuedThreads()：是否有线程正在等待获取许可证。
- void reducePermits(int reduction)：减少reduction个许可证。是个protected方法。
- Collection getQueuedThreads()：返回所有等待获取许可证的线程集合。是个protected方法。

CLH队列

【Java并发编程实战】—– AQS(四)：CLH同步队列

在【Java并发编程实战】—–"J.U.C"：CLH队列锁提过，AQS里面的CLH队列是CLH同步锁的一种变形。其主要从两方面进行了改造：节点的结构与节点等待机制。在结构上引入了头结点和尾节点，他们分别指向队列的头和尾，尝试获取锁、入队列、释放锁等实现都与头尾节点相关，并且每个节点都引入前驱节点和后后续节点的引用；在等待机制上由原来的自旋改成阻塞唤醒。

知道其结构了，我们再看看他的实现。在线程获取锁时会调用AQS的acquire()方法，该方法第一次尝试获取锁如果失败，会将该线程加入到CLH队列中：

```
public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

addWaiter：

```
private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        Node pred = tail;
        if (pred != null) {
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        enq(node);
        return node;
    }
```

这是addWaiter()的实现，在厘清这段代码之前我们要先看一个更重要的东东，Node,CLH队列的节点。其源码如下：

```
static final class Node {
            /** 线程已被取消 */
            static final int CANCELLED =  1;

            /** 当前线程的后继线程需要被unpark(唤醒) */
            static final int SIGNAL    = -1;

            /** 线程(处在Condition休眠状态)在等待Condition唤醒 */
            static final int CONDITION = -2;

            /** 共享锁 */
            static final Node SHARED = new Node();
            /** 独占锁  */
            static final Node EXCLUSIVE = null;

            volatile int waitStatus;

            /** 前继节点 */
            volatile Node prev;

            /** 后继节点 */
            volatile Node next;
```

```
            volatile Thread thread;

            Node nextWaiter;

            final boolean isShared() {

            return nextWaiter == SHARED;
            }


           /** 获取前继节点 */
            final Node predecessor() throws NullPointerException {
                Node p = prev;
                if (p == null)
                    throw new NullPointerException();
                else
                    return p;
            }

            /**
             * 三个构造函数
             */
            Node() {
            }

            Node(Thread thread, Node mode) {
                this.nextWaiter = mode;
                this.thread = thread;
            }

            Node(Thread thread, int waitStatus) {
                this.waitStatus = waitStatus;
                this.thread = thread;
            }
        }
```

在这个源代码中有三个值（CANCELLED、SIGNAL、CONDITION）要特别注意，前面提到过CLH队列的节点都有一个状态位，该状态位与线程状态密切相关：

CANCELLED = 1：因为超时或者中断，节点会被设置为取消状态，被取消的节点时不会参与到竞争中的，他会一直保持取消状态不会转变为其他状态；

SIGNAL = -1：其后继节点已经被阻塞了，到时需要进行唤醒操作；

CONDITION = -2：表示这个结点在条件队列中，因为等待某个条件而被阻塞；

0：新建节点一般都为0。

入列

在线程尝试获取锁的时候，如果失败了需要将该线程加入到CLH队列，入列中的主要流程是：tail执行新建node，然后将node的后继节点指向旧tail值。注意在这个过程中有一个CAS操作，采用自旋方式直到成功为止。其代码如下：

```
for(;;){
            Node t = tail;
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
```

其实这段代码在enq()方法中存在。

出列

当线程是否锁时，需要进行“出列”，出列的主要工作则是唤醒其后继节点（一般来说就是head节点），让所有线程有序地进行下去：

```
Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
```

取消

线程因为超时或者中断涉及到取消的操作，如果某个节点被取消了，那个该节点就不会参与到锁竞争当中，它会等待GC回收。取消的主要过程是将取消状态的节点移除掉，移除的过程还是比较简单的。先将其状态设置为CANCELLED,然后将其前驱节点的pred执行其后继节点，当然这个过程仍然会是一个CAS操作：

```
node.waitStatus = Node.CANCELLED;
Node pred = node.prev;
Node predNext = pred.next;
Node next = node.next;
```

挂起

我们了解了AQS的CLH队列相比原始的CLH队列锁，它采用了一种变形操作，将自旋机制改为阻塞机制。当前线程将首先检测是否为头结点且尝试获取锁，如果当前节点为头结点并成功获取锁则直接返回，当前线程不进入阻塞，否则将当前线程阻塞：

```
for (;;) {
    if (node.prev == head)
if(尝试获取锁成功){
         head=node;
         node.next=null;
         return;
     }
   阻塞线程
}
```

JAVA并发编程学习笔记之CLH队列锁

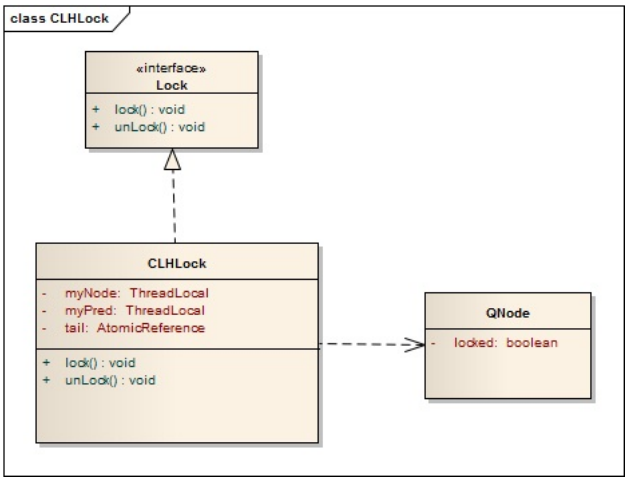
NUMA与SMP

SMP(Symmetric Multi-Processor)，即对称多处理器结构，指服务器中多个CPU对称工作，每个CPU访问内存地址所需时间相同。其主要特征是共享，包含对CPU，内存，I/O等进行共享。SMP的优点是能够保证内存一致性，缺点是这些共享的资源很可能成为性能瓶颈，随着CPU数量的增加，每个CPU都要访问相同的内存资源，可能导致内存访问冲突，可能会导致CPU资源的浪费。常用的PC机就属于这种。

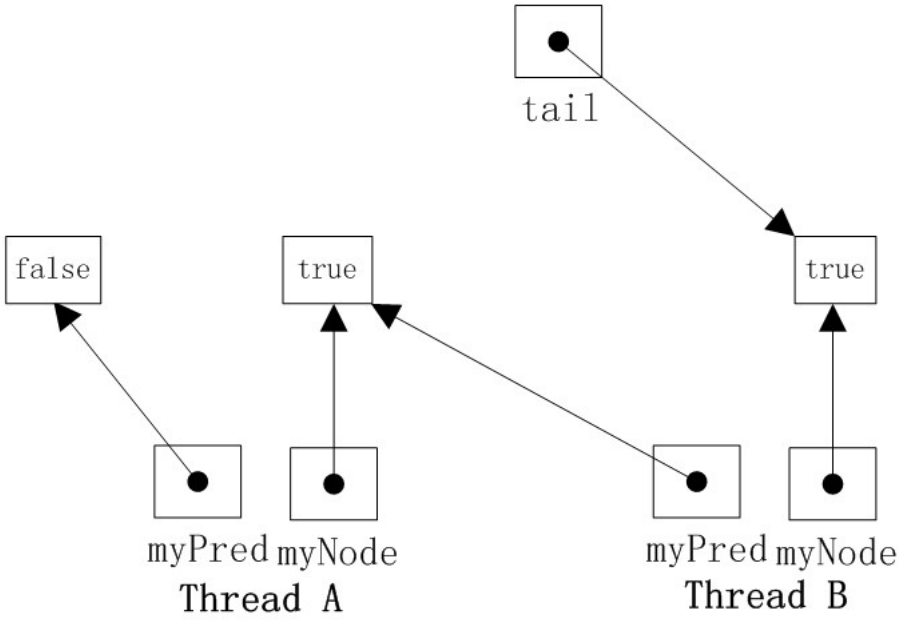
NUMA(Non-Uniform Memory Access)非一致存储访问， 将CPU分为CPU模块，每个CPU模块由多个CPU组成， 并且具有独立的本地内存、 I/O 槽口等，模块之间可以通过互联模块 相互访问 ，访问本地内存的速度将远远高于访问远地内存 (系统内其它节点的内存) 的速度，这也是非一致存储访问 NUMA 的由来。 NUMA优点是 可以较好地解决原来 SMP 系统的扩展问题，缺点是由于 访问远地内存的延时远远超过本地内存，因此当 CPU 数量增加时，系统性能无法线性增加。

CLH算法实现

CLH队列中的结点QNode中含有一个locked字段，该字段若为true表示该线程需要获取锁，且不释放锁，为false表示线程释放了锁。结点之间是通过隐形的链表相连，之所以叫隐形的链表是因为这些结点之间没有明显的next指针，而是通过myPred所指向的结点的变化情况来影响myNode的行为。CLHLock上还有一个尾指针，始终指向队列的最后一个结点。CLHLock的类图如下所示：



当一个线程需要获取锁时，会创建一个新的QNode，将其中的locked设置为true表示需要获取锁，然后线程对tail域调用getAndSet方法，使自己成为队列的尾部，同时获取一个指向其前趋的引用myPred,然后该线程就在前趋结点的locked字段上旋转，直到前趋结点释放锁。当一个线程需要释放锁时，将当前结点的locked域设置为false，同时回收前趋结点。如下图所示，线程A需要获取锁，其myNode域为true，些时tail指向线程A的结点，然后线程B也加入到线程A后面，tail指向线程B的结点。然后线程A和B都在它的myPred域上旋转，一量它的myPred结点的locked字段变为false，它就可以获取锁扫行。明显线程A的myPred locked域为false，此时线程A获取到了锁。



CLH lock示意图

整个CLH的代码如下，其中用到了ThreadLocal类，将QNode绑定到每一个线程上，同时用到了AtomicReference,对尾指针的修改正是调用它的getAndSet()操作来实现的，它能够保证以原子方式更新对象引用。

```
public class CLHLock implements Lock {
    AtomicReference<QNode> tail = new AtomicReference<QNode>(new QNode());
    ThreadLocal<QNode> myPred;
    ThreadLocal<QNode> myNode;

    public CLHLock() {
        tail = new AtomicReference<QNode>(new QNode());
        myNode = new ThreadLocal<QNode>() {
            protected QNode initialValue() {
                return new QNode();
            }
        };
        myPred = new ThreadLocal<QNode>() {
            protected QNode initialValue() {
                return null;
            }
        };
    }

    @Override
    public void lock() {
        QNode qnode = myNode.get();
        qnode.locked = true;
        QNode pred = tail.getAndSet(qnode);
        myPred.set(pred);
        while (pred.locked) {
        }
    }

    @Override
    public void unlock() {
        QNode qnode = myNode.get();
        qnode.locked = false;
        myNode.set(myPred.get());
    }
}
```

从代码中可以看出lock方法中有一个while循环，这 是在等待前趋结点的locked域变为false，这是一个自旋等待的过程。unlock方法很简单，只需要将自己的locked域设置为false即可。

CLH优缺点

CLH队列锁的优点是空间复杂度低（ 如果有n个线程，L个锁，每个线程每次只获取一个锁，那么需要的存储空间是O（L+n），n个线程有n个myNode，L个锁有L个tail），CLH的一种变体被应用在了JAVA并发框架中。唯一的缺点是在NUMA系统结构下性能很差，在这种系统结构下，每个线程有自己的内存，如果前趋结点的内存位置比较远，自旋判断前趋结点的locked域，性能将大打折扣，但是在SMP系统结构下该法还是非常有效的。一种解决NUMA系统结构的思路是MCS队列锁。

里强行剥夺足够数量的资源分配给死锁进程，以解除死锁状态

算法（左老师-主要讲技巧）

红黑树的了解（平衡树，二叉搜索树），使用场景

把数据结构上几种树集中的讨论一下：

1. AVLtree

定义：先发明的自平衡二叉查找树。在AVL树中任何节点的两个子树的高度 大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和坏情况 下都是O（log n）。增加和删除可能需要通过一次或多次树旋转来重新平衡这 个树。

节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。带有 平衡因子1、0或 -1的节点被认为是平衡的。带有平衡因子 -2或2的节点被认为 是不平衡的，并需要重新平衡这个树。平衡因子可以直接存储在每个节点中，或 从可能存储在节点中的子树高度计算出来。

一般我们所看见的都是排序平衡二叉树。

AVLtree使用场景：AVL树适合用于插入删除次数比较少，但查找多的情况。插 入删除导致很多的旋转，旋转是非常耗时的。AVL 在linux内核的vm area中使用。

2. 二叉搜索树

二叉搜索树也是一种树，适用与一般二叉树的全部操作，但二叉搜索树能够实现 数据的快速查找。

二叉搜索树满足的条件：

1. 非空左子树的所有键值小于其根节点的键值
2. 非空右子树的所有键值大于其根节点的键值
3. 左右子树都是二叉搜索树

二叉搜索树的应用场景：如果是没有退化称为链表的二叉树，查找效率就是 lgn，效率不错，但是一旦退换称为链表了，要么使用平衡二叉树，或者之后的 RB树，因为链表就是线性的查找效率。

3. 红黑树的定义

红黑树是一种二叉查找树，但在每个结点上增加了一个存储位表示结点的颜色， 可以是RED或者BLACK。通过对任何一条从根到叶子的路径上各个着色方式的 限制，红黑树确保没有一条路径会比其他路径长出两倍，因而是接近平衡的。 当二叉查找树的高度较低时，这些操作执行的比较快，但是当树的高度较高时， 这些操作的性能可能不比用链表好。红黑树（red-black tree）是一种平衡的二 叉查找树，它能保证在坏情况下，基本的动态操作集合运行时间为O(lgn)。 红黑树必须要满足的五条性质：

- 性质一：节点是红色或者是黑色； 在树里面的节点不是红色的就是黑色的，没有其他 颜色，要不怎么叫红黑树呢，是吧。
- 性质二：根节点是黑色； 根节点总是黑色的。它不能为红。
- 性质三：每个叶节点（NIL或空节点）是黑色；
- 性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节 点）； 就是连续的两个节点不能是连续的红色，连续的两个节点的意思就是父节点与子节点不能是连续的红色。
- 性质五：从任一节点到其每个叶节点的所有路径都包含相同数目的黑色节点。从根节点 到每一个NIL节点的路径中，都包含了相同数量的黑色节点。

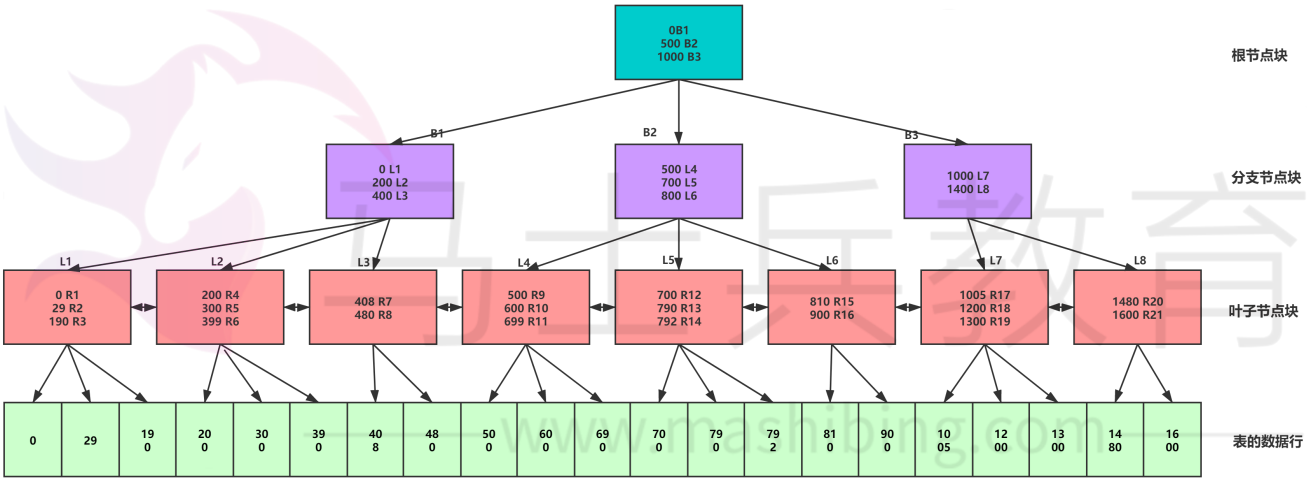
红黑树的应用场景：红黑树是一种不是非常严格的平衡二叉树，没有AVLtree那 么严格的平衡要求，所以它的平均查找，增添删除效率都还不错。广泛用在 C++的STL中。如map和set都是用红黑树实现的。

4. B树定义

B树和平衡二叉树稍有不同的是B树属于多叉树又名平衡多路查找树（查找路径 不只两个），不属于二叉搜索树的范畴，因为它不止两路，存在多路。

B树满足的条件：

- （1）树种的每个节点多拥有m个子节点且m>=2,空树除外（注：m阶代表一个树节 点多有多少个查找路径，m阶=m路,当m=2则是2叉树,m=3则是3叉）；
- （2）除根节点外每个节点的关键字数量大于等于ceil(m/2)-1个小于等于m-1个，非根 节点关键字数必须>=2;（注：ceil()是个朝正无穷方向取整的函数 如ceil(1.1)结果为2)
- （3）所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也 有指向其子节点的指针只不过其指针地址都为null对应下图后一层节点的空格子
- （4）如果一个非叶节点有N个子节点，则该节点的关键字数等于N-1;
- （5）所有节点关键字是按递增次序排列，并遵循左小右大原则；



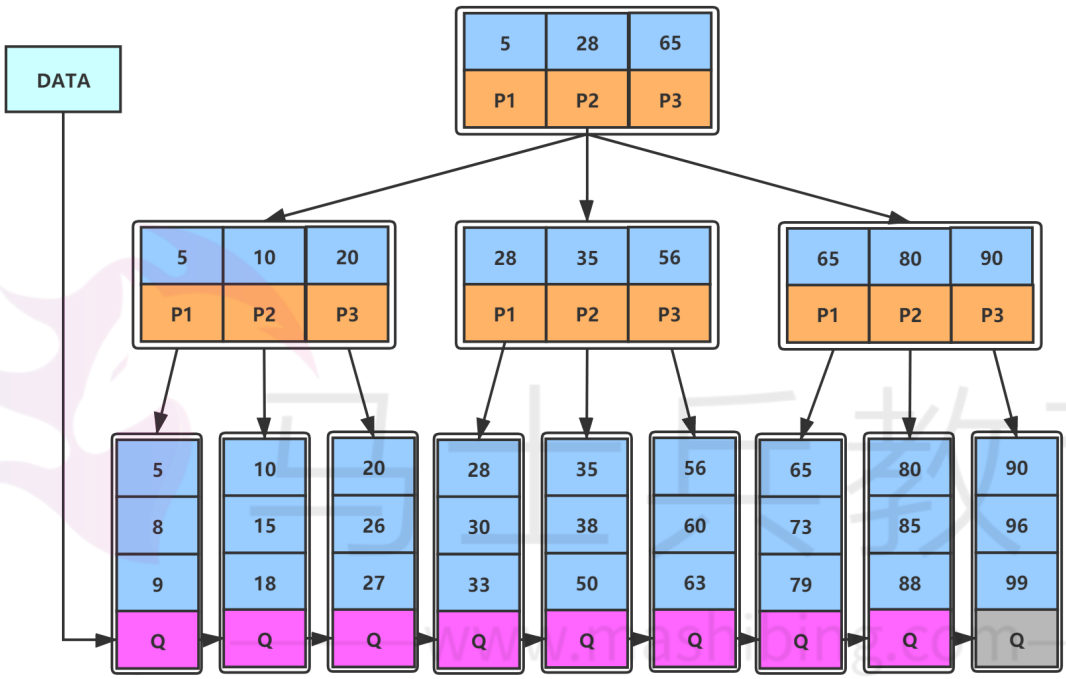
B树的应用场景：构造一个多阶的B类树，然后在尽量多的在结点上存储相关的信息， 保证层数尽量的少，以便后面我们可以更快的找到信息，磁盘的I/O操作也少一些，而 且B类树是平衡树，每个结点到叶子结点的高度都是相同，这也保证了每个查询是稳定 的。

5. B+树

B+树是B树的一个升级版，B+树是B树的变种树，有n棵子树的节点中含有n个 关键字，每个关键字不保存数据，只用来索引，数据都保存在叶子节点。是为文 件系统而生的。

相对于B树来说B+树更充分的利用了节点的空间，让查询速度更加稳定，其速度 完全接近于二分法查找。为什么说B+树查找的效率要比B树更高、更稳定；我们 先看看两者的区别

- （1）B+跟B树不同，B+树的非叶子节点不保存关键字记录的指针，这样使得B+树每 个节点所能保存的关键字大大增加；
- （2）B+树叶子节点保存了父节点的所有关键字和关键字记录的指针，每个叶子节点的 关键字从小到大链接；
- （3）B+树的根节点关键字数量和其子节点个数相等；
- （4）B+的非叶子节点只进行数据索引，不会存实际的关键字记录的指针，所有数据地 址必须要到叶子节点才能获取到，所以每次数据查询的次数都一样；



特点：

在B树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更 快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速 度更稳定；

应用场景： 用在磁盘文件组织 数据索引和数据库索引。

6. Trie树（字典树）

trie，又称前缀树，是一种有序树，用于保存关联数组，其中的键通常是字符 串。与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决 定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根 节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点 和部分内部节点所对应的键才有相关的值。

在图示中，键标注在节点中，值标注在节点之下。每一个完整的英文单词对应一 个特定的整数。Trie 可以看作是一个确定有限状态自动机，尽管边上的符号一 般是隐含在分支的顺序中的。

键不需要被显式地保存在节点中。图示中标注出完整的单词，只是为了演示 trie 的原理。

trie树的优点：利用字符串的公共前缀来节约存储空间，大限度地减少无谓的 字符串比较，查询效率比哈希表高。

缺点：Trie树是一种比较简单的数据结构.理 解起来比较简单,正所谓简单的东西也得付出代价.故Trie树也有它的缺点,Trie树 的内存消耗非常大.

其基本性质可以归纳为：

1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符都不相同。

典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经 常被搜索引擎系统用于文本词频统计。字典树与字典很相似,当你要查一个单词 是不是在字典树中,首先看单词的第一个字母是不是在字典的第一层,如果不在,说 明字典树里没有该单词,如果在就在该字母的孩子节点里找是不是有单词的第二 个字母,没有说明没有该单词,有的话用同样的方法继续查找.字典树不仅可以用来 储存字母,也可以储存数字等其它数据。

（2） 红黑树在STL上的应用

STL中set、multiset、map、multimap底层是红黑树实现的，而 unordered_map、unordered_set 底层是哈希表实现的。

multiset、multimap： 插入相同的key的时候，我们将后插入的key放在相等的 key的右边，之后不管怎么进行插入或删除操作，后加入的key始终被认为比之 前的大。

了解并查集吗？（低频）

什么是合并查找问题呢?

顾名思义，就是既有合并又有查找操作的问题。举个例子，有一群人，他们之间 有若干好友关系。如果两个人有直接或者间接好友关系，那么我们就说他们在同 一个朋友圈中，这里解释下，如果Alice是Bob好友的好友，或者好友的好友的 好友等等，即通过若干好友可以认识，那么我们说Alice和Bob是间接好友。随 着时间的变化，这群人中有可能会有新的朋友关系，这时候我们会对当中某些人 是否在同一朋友圈进行询问。这就是一个典型的合并 - 查找操作问题，既包含了 合并操作，又包含了查找操作。

并查集，在一些有N个元素的集合应用问题中，我们通常是在开始时让每个元素 构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并， 其间要反复查找一个元素在哪个集合中。

并查集是一种树型的数据结构，用于处理一些不相交集合（Disjoint Sets）的合 并及查询问题。

并查集也是使用树形结构实现。不过，不是二叉树。每个元素对应一个节点，每 个组对应一棵树。在并查集中，哪个节点是哪个节点的父亲以及树的形状等信息 无需多加关注，整体组成一个树形结构才是重要的。类似森林

（4） 贪心算法和动态规划的区别

贪心算法：局部优，划分的每个子问题都优，得到全局优，但是不能保证 是全局优解，所以对于贪心算法来说，解是从上到下的，一步一步优，直到 后。 动态规划：将问题分解成重复的子问题，每次都寻找左右子问题解中优的解， 一步步得到全局的优解.重复的子问题可以通过记录的方式，避免多次计算。 所以对于动态规划来说，解是从小到上，从底层所有可能性中找到优解，再一 步步向上。 分治法：和动态规划类似，将大问题分解成小问题，但是这些小问题是独立的， 没有重复的问题。独立问题取得解，再合并成大问题的解。 例子：比如钱币分为1元3元4元，要拿6元钱，贪心的话，先拿4，再拿两个1， 一共3张钱；实际优却是两张3元就够了。

（5） 判断一个链表是否有环，如何找到这个环的起点

给定一个单链表，只给出头指针h：

1. 如何判断是否存在环?
2. 如何知道环的长度?
3. 如何找出环的连接点在哪里?
4. 带环链表的长度是多少?

解法：

1. 对于问题1，使用追赶的方法，设定两个指针slow、fast，从头指针开始， 每次分别前进1步、2步。如存在环，则两者相遇；如不存在环，fast遇到NULL 退出。
2. 对于问题2，记录下问题1的碰撞点p，slow、fast从该点开始，再次碰撞所 走过的操作数就是环的长度s。
3. 问题3：有定理：碰撞点p到连接点的距离=头指针到连接点的距离，因此， 分别从碰撞点、头指针开始走，相遇的那个点就是连接点。(证明在后面附注)
4. 问题3中已经求出连接点距离头指针的长度，加上问题2中求出的环的长度， 二者之和就是带环单链表的长度

实现一个strcpy函数（或者memcpy），如果内存可能重叠呢

——大家一般认为名不见经传strcpy函数实现不是很难，流行的strcpy函数写法是：

```
1. char *my_strcpy(char *dst,const char *src)
2. {
3. assert(dst != NULL);
4. assert(src != NULL);
5. char *ret = dst;
6. while((* dst++ = * src++) != '\0')
7. ;
8. return ret;
9. }
```

如果注意到：

1，检查指针有效性；

2，返回目的指针des；

3，源字符串的末尾 '\0' 需要拷贝。

内存重叠

内存重叠：拷贝的目的地址在源地址范围内。所谓内存重叠就是拷贝的目的地址和源地址有重叠。

在函数strcpy和函数memcpy都没有对内存重叠做处理的，使用这两个函数的时候只有程序员自己保证源地址和目标地址不重叠，或者使用memmove函数进行内存拷贝。

memmove函数对内存重叠做了处理。

strcpy的正确实现应为：

```
1. char *my_strcpy(char *dst,const char *src)
2. {
3. assert(dst != NULL);
4. assert(src != NULL);
5. char *ret = dst;
6. memmove(dst,src,strlen(src)+1);
7. return ret;
8. }
```

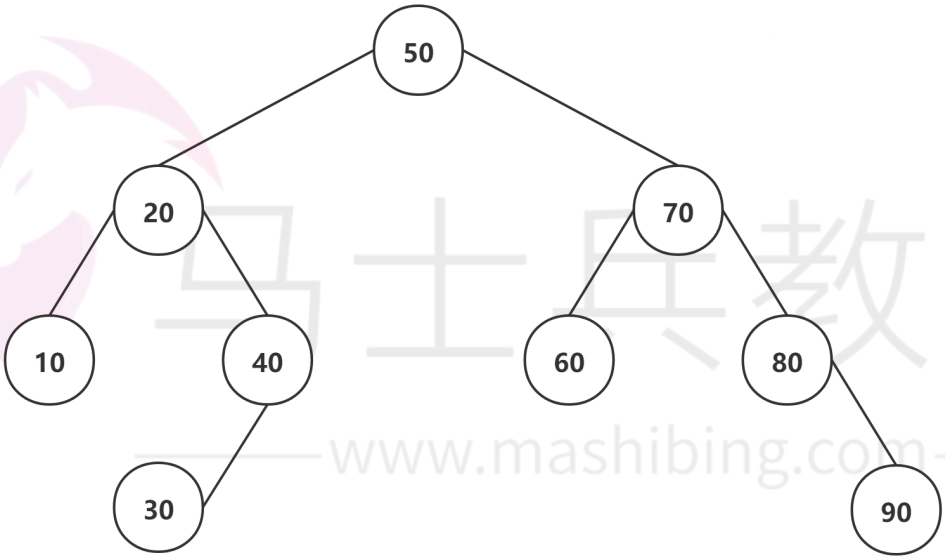
memmove函数实现时考虑到了内存重叠的情况，可以完成指定大小的内存拷贝

快排存在的问题，如何优化

快排的时间复杂度

时间复杂度 快平均是O（nlogn），慢的时候是O(n2);辅助空间也是

O(logn)； 开始学快排时 疑惑的就是这个东西不知道怎么得来的，一种是通过数学运算可以的出来，还有一种是通过递归树来理解就容易多了



所谓时间复杂度 理想的就是取到中位数情况，那么递归树就是一个完全二叉树，那么树的深度也就是 低为Logn，这个时候每一次又需要n次比较，所以时间复杂度nlogn，当快排为顺序或者逆序

时，这个数为一个斜二叉树，深度为n，同样每次需要n次比较，那那么 坏需要n2的时间优化：

1.当整个序列有序时退出算法；

2.当序列长度很小时（根据经验是大概小于 8），应该使用常数更小的算法，比如插入排序等；

3.随机选取分割位置；

4.当分割位置不理想时，考虑是否重新选取分割位置；

5.分割成两个序列时，只对其中一个递归进去，另一个序列仍可以在这一函数内继续划分，可以显著减小栈的大小（尾递归）：

6.将单向扫描改成双向扫描，可以减少划分过程中的交换次数优化1：当待排序序列的长度分割到一定大小后，使用插入排序原因：对于很小和部分有序的数组，快排不如插排好。当待排序序列的长度分割到一定大小后，继续分割的效率比插入排序要差，此时可以使用插排而不是快排优化2：在一次分割结束后，可以把与Key相等的元素聚在一起，继续下次分割时，不用再对与key相等元素分割优化3：优化递归操作快排函数在函数尾部有两次递归操作，我们可以对其使用尾递归优化优点：如果待排序的序列划分极端不平衡，递归的深度将趋近于n，而栈的大小是很有限的，每次递归调用都会耗费一定的栈空间，函数的参数越多，每次递归

耗费的空间也越多。优化后，可以缩减堆栈深度，由原来的O(n)缩减为

O(logn)，将会提高性能。

Top K问题（可以采取的方法有哪些，各自优点？）

1.将输入内容（假设用数组存放）进行完全排序，从中选出排在前K的元素即为所求。有了这个思路，我们可以选择相应的排序算法进行处理，目前来看快速排序，堆排序和归并排序都能达到O(nlogn)的时间复杂度。

2.对输入内容进行部分排序，即只对前K大的元素进行排序（这K个元素即为所求）。此时我们可以选择冒泡排序或选择排序进行处理，即每次冒泡（选择）都能找到所求的一个元素。这类策略的时间复杂度是O(Kn)。

3.对输入内容不进行排序，显而易见，这种策略将会有更好的性能开销。我们此时可以选择两种策略进行处理：用一个桶来装前k个数，桶里面可以按照 小堆来维护 a)利用 小堆维护一个大小为K的数组，目前该小根堆中的元素是排名前K的数，其中根是 小的数。此后，每次从原数组中取一个元素与根进行比较，如大于根的元素，则将根元素替换并进行堆调整（下沉），即保证小根堆中的元素仍然是排名前K的数，且根元素仍然 小；否则不予处理，取下一个数组元素继续该过程。该算法的时间复杂度是O(nlogK)，一般来说企业中都采用该策略处理 top-K问题，因为该算法不需要一次将原数组中的内容全部加载到内存中，而这正是海量数据处理必然会面临的一个关卡。

b)利用快速排序的分划函数找到分划位置K，则其前面的内容即为所求。该算法是一种非常有效的处理方式，时间复杂度是O(n)（证明可以参考算法导论书籍）。对于能一次加载到内存中的数组，该策略非常优秀。

Bitmap的使用，存储和插入方法

BitMap从字面的意思

很多人认为是位图，其实准确的来说，翻译成基于位的映射。

在所有具有性能优化的数据结构中，大家使用 多的就是hash表，是的，在具有定位查找上具有O(1)的常量时间，多么的简洁优美。但是数据量大了，内存就不够了。

当然也可以使用类似外排序来解决问题的，由于要走IO所以时间上又不行。

所谓的Bit-map就是用一个bit位来标记某个元素对应的Value， 而Key即是该元素。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

其实如果你知道计数排序的话（算法导论中有一节讲过），你就会发现这个和计数排序很像。

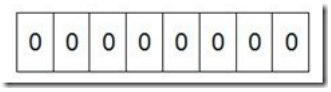
bitmap应用

```
1）可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下。
2）去重数据而达到压缩数据
```

还可以用于爬虫系统中url去重、解决全组合问题。

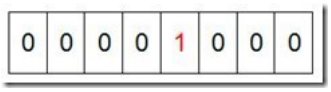
BitMap应用：排序示例

假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0(如下图：）

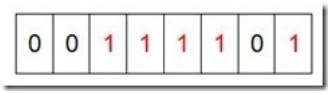


然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 p+(i/8)|(0×01<<(i%8)) 当然了这里的操作涉及到Big-ending和Littleending的情况，这里默认为Big-ending。不过计算机一般是小端存储的，如

intel。小端的话就是将倒数第5位置1）,因为是从零开始的，所以要把第五位置为一（如下图）：



然后再处理第二个元素7，将第八位置为1,，接着再处理第三个元素，一直到后处理完所有的元素，将相应的位置为1，这时候的内存的Bit位的状态如下：



然后我们现在遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的。

bitmap排序复杂度分析

Bitmap排序需要的时间复杂度和空间复杂度依赖于数据中 大的数字。

bitmap排序的时间复杂度不是O(N)的，而是取决于待排序数组中的大值

MAX，在实际应用上关系也不大，比如我开10个线程去读byte数组，那么复杂度为:O(Max/10)。也就是要是读取的，可以用多线程的方式去读取。时间复杂度方面也是O(Max/n)，其中Max为byte[]数组的大小，n为线程大小。空间复杂度应该就是O(Max/8)bytes吧

BitMap算法流程

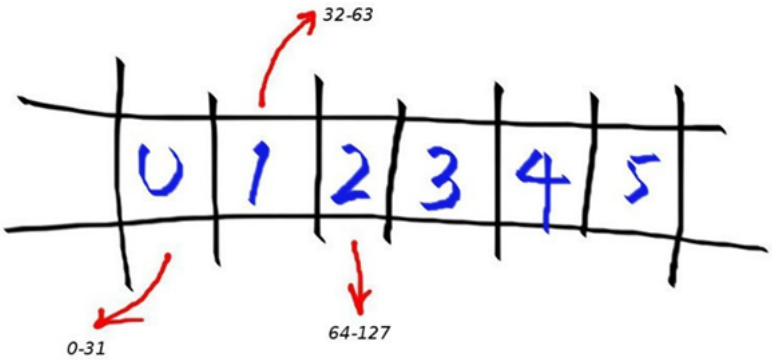
假设需要排序或者查找的 大数MAX=10000000（lz:这里MAX应该是 大的数而不是int数据的总数！），那么我们需要申请内存空间的大小为int a[1 +

MAX/32]。

其中：a[0]在内存中占32为可以对应十进制数0-31，依次类推： bitmap表为：

a[0]--------->0-31 a[1]--------->32-63 a[2]--------->64-95

a[3]--------->96-127



我们要把一个整数N映射到Bit-Map中去，首先要确定把这个N Mapping到哪一个数组元素中去，即确定映射元素的index。我们用int类型的数组作为map的元素，这样我们就知道了一个元素能够表示的数字个数(这里是32)。于是N/32 就可以知道我们需要映射的key了。所以余下来的那个N%32就是要映射到的位数。

1.求十进制数对应在数组a中的下标：

先由十进制数n转换为与32的余可转化为对应在数组a中的下标。

如十进制数0-31，都应该对应在a[0]中，比如n=24,那么 n/32=0，则24对应在数组a中的下标为0。又比如n=60,那么n/32=1，则60对应在数组a中的下标为 1，同理可以计算0-N在数组a中的下标。

i = N>>K % 结果就是N/(2^K)

Note: map的范围是[0, 原数组 大的数对应的2的整次方数-1]。

2.求十进制数对应数组元素a[i]在0-31中的位m：

十进制数0-31就对应0-31，而32-63则对应也是0-31，即给定一个数n可以通过模32求得对应0-31中的数。

m = n & ((1 << K) - 1) %结果就是n%(2^K)

3.利用移位0-31使得对应第m个bit位为1

如a[i]的第m位置1：a[i] = a[i] | (1<

如：将当前4对应的bit位置1的话，只需要1左移4位与B[0] | 即可。

Note:

1 p+(i/8)|(0×01<<(i%8))这样也可以?

2 同理将int型变量a的第k位清0，即a=a&~(1<

BitMap算法评价

优点：

1. 运算效率高，不进行比较和移位；
2. 占用内存少，比如 大的数MAX=10000000；只需占用内存为

MAX/8=1250000Byte=1.25M。

3.

缺点：

1. 所有的数据不能重复，即不可对重复的数据进行排序。（少量重复数据查找还是可以的，用2-bitmap）。
2. 当数据类似（1，1000，10万）只有3个数据的时候，用bitmap时间复杂度和空间复杂度相当大，只有当数据比较密集时才有优势。

字典树的理解以及在统计上的应用

Trie的核心思想是空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。Trie树也有它的缺点,Trie树的内存消耗非常大.当然,或许用左儿子右兄弟的方法建树的话,可能会好点. 就是在海量数据中找出某一个数，比如2亿QQ号中查找出某一个特定的QQ号

N个骰子出现和为m的概率

典型的可以用动态规划的思想来完成

1.现在变量有：骰子个数，点数和。当有k个骰子，点数和为n时，出现次数记为 f(k,n)。那与k-1个骰子阶段之间的关系是怎样的?

2.当我有k-1个骰子时，再增加一个骰子，这个骰子的点数只可能为1、2、3、

4、5或6。那k个骰子得到点数和为n的情况有：

(k-1,n-1)：第k个骰子投了点数1

(k-1,n-2)：第k个骰子投了点数2

(k-1,n-3)：第k个骰子投了点数3

...

(k-1,n-6)：第k个骰子投了点数6

在k-1个骰子的基础上，再增加一个骰子出现点数和为n的结果只有这6种情况！所以：f(k,n)=f(k-1,n-1)+f(k-1,n-2)+f(k-1,n-3)+f(k-1,n-4)+f(k-1,n-5)+f(k-1,n-6)

3.有1个骰子，f(1,1)=f(1,2)=f(1,3)=f(1,4)=f(1,5)=f(1,6)=1。

用递归就可以解决这个问题：

```
/*****************************
func:获取n个骰子指定点数和出现的次数
para:n:骰子个数;sum:指定的点数和
return:点数和为sum的排列数
*****************************/
int getNSumCount(int n, int sum){
    if(n<1||sum<n||sum>6*n){
        return 0;
    }
    if(n==1){
        return  1;
    }
    int resCount=0;
    resCount=getNSumCount(n-1,sum-1)+getNSumCount(n-1,sum-2)+
             getNSumCount(n-1,sum-3)+getNSumCount(n-1,sum-4)+
             getNSumCount(n-1,sum-5)+getNSumCount(n-1,sum-6);
    return resCount;
}

//验证
int main(){
    int n=0;
    while(true){
        cout<<"input dice num: ";
        cin>>n;
        for(int i=n;i<=6*n;++i)
            cout<<"f("<<n<<","<<i<<")="<<getNSumCount(n,i)<<endl;
    }
}
```

用迭代来完成

```
/****************************************
func:给定骰子数目n，求所有可能点数和的种类数
para: n:骰子个数;count:存放各种点数和的种类数，下标i表示点数和为（i+n）
return:出错返回-1，成功返回0
****************************************/
int getNSumCountNotRecusion(int n, int* count){
    if(n<1)
        return -1;
    //初始化最初状态
    count[0]=count[1]=count[2]=count[3]=count[4]=count[5]=1;
    if(n==1) return 0;

    for(int i=2;i<=n;++i){
        for(int sum=6*i;sum>=i;--sum){
            int tmp1=((sum-1-(i-1))>=0?count[sum-1-(i-1)]:0); //上一阶段点数和sum-1的排列总数
            int tmp2=(sum-2-(i-1)>=0?count[sum-2-(i-1)]:0);
            int tmp3=(sum-3-(i-1)>=0?count[sum-3-(i-1)]:0);
            int tmp4(sum-4-(i-1)>=0?count[sum-4-(i-1)]:0);
            int tmp5=(sum-5-(i-1)>=0?count[sum-5-(i-1)]:0);
            int tmp6=(sum-6-(i-1)>=0?count[sum-6-(i-1)]:0);
            count[sum-i]=tmp1+tmp2+tmp3+tmp4+tmp5+tmp6;
        }
    }
    return 0;
}
```

海量数据问题（可参考左神的书）

目前关于海量数据想到的解决办法：

1.bitmap

2.桶排序，外部排序，将需要排序的放到外存上，不用全部放到内存上

一致性哈希说明

优点

1.当后端是缓存服务器时，经常使用一致性哈希算法来进行负载均衡。使用一致性哈希的好处在于，增减集群的缓存服务器时，只有少量的缓存会失效，回源量较小。

2.尽量减少数据丢失问题，减少移动数据的风险

如何给100亿个数字排序?

场景

之前写过一篇海量数据中统计ip出现次数最多的博客，今天再写篇类似的，当然会有不同的地方，相同的地方我快速写过

今天要给100亿个数字排序，100亿个 int 型数字放在文件里面大概有 37.2GB，非常大，内存一次装不下了。那么肯定是要拆分成小的文件一个一个来处理，最终在合并成一个排好序的大文件。

实现思路

1.把这个37GB的大文件，用哈希分成1000个小文件，每个小文件平均38MB左右（理想情况），把100亿个数字对1000取模，模出来的结果在0到999之间，每个结果对应一个文件，所以我这里取的哈希函数是 h = x % 1000，哈希函数取得"好"，能使冲突减小，结果分布均匀。

2.拆分完了之后，得到一些几十MB的小文件，那么就可以放进内存里排序了，可以用快速排序，归并排序，堆排序等等。

3.1000个小文件内部排好序之后，就要把这些内部有序的小文件，合并成一个大的文件，可以用**二叉堆**来做1000路合并的操作，每个小文件是一路，合并后的大文件仍然有序。

- 首先遍历1000个文件，每个文件里面取第一个数字，组成 (数字, 文件号) 这样的组合加入到堆里（假设是从小到大排序，用小顶堆），遍历完后堆里有1000个 (数字，文件号) 这样的元素
- 然后不断从堆顶拿元素出来，每拿出一个元素，把它的文件号读取出来，然后去对应的文件里，加一个元素进入堆，直到那个文件被读取完。拿出来的元素当然追加到最终结果的文件里。
- 按照上面的操作，直到堆被取空了，此时最终结果文件里的全部数字就是有序的了。

思维拓展

类似的100亿个数字求和，求中位数，求平均数，套路就是一样的了。
求和：统计每个小文件的和，返回给master再求和就可以了。
求平均数：上面能求和了，再除以100亿就是平均数了
求中位数：在排序的基础上，遍历到中间的那个数就是中位数了。

更正，评论中网友提出了不用对数字进行哈希，直接平均分成1000份，进行内部排序后，直接进行 k 路归并排序，也是可以的。

哈希分治法 - 统计海量数据中出现次数最多的前10个IP

场景

这是一个 ip 地址 127.0.0.1
假设有100亿个这样的 ip 地址存在文件中
这个文件大小大约是 100GB
问题：要统计出100亿个 ip 中，重复出现次数最多的前10个

分析

100GB 几乎不可能一次加载进内存进行操作，所以必须要拆分
那么可以利用分治的思想，把规模大的问题化小，然后解决各个小的问题，最后得出结果。

实现思路

- ipv4 地址是一个 32 位的整数，可以用 uint 保存。
- 我先设计一个哈希函数，把100个G的文件分成10000份，每份大约是 10MB，可以加载进内存了。

> 例如：我设计一个简单的哈希函数是 f(ip) = ip % 10000，(ip 是个32位整数)
> 那么 5 % 10000 = 5，不管 5 在哪个地方 5 % 10000 的结果都是 5，这就保证了相同的 ip 会被放在同一个子文件中，方便统计，相同的元素经过同一个哈希函数，得出的哈希值是一样的。
> 那么我把100亿个 ip，都进行 ip % 10000 的操作，就完成了 100GB 文件分解成 10000个子文件的任务了。当然实际中哈希函数的选取很重要，尽量使得元素分布均匀，哈希冲突少的函数才是最好的。

记住，我把上面这个分解的过程叫做 **Map**，由一台叫 **master** 的计算机完成这个工作。

- 10MB 的小文件加进内存，统计出出现次数最多的那个ip

> 10MB 的小文件里面存着很多 ip，他们虽然是乱序的，但是相同的 ip 会映射到同一个文件中来！
> 那么可以用二叉树统计出现次数，二叉树节点保存（ip, count）的信息，把所有 ip 插入到二叉树中，如果这个 ip 不存在，那么新建一个节点, count 标记 1，如果有，那么把 count++，最终遍历一遍树，就能找出 count 最大的 ip 了。

我把这个过程叫做 **Reduce**，由很多台叫 **worker** 的计算机来完成。
每个 worker 至少要找出最大的前10个 ip 返回给 master，master 最后会收集到 10000 * 10 个 ip，大约 400KB，然后再找出最大的前 10 个 ip 就可以了。
最简单的遍历10遍，每次拿个最大值出来就可以了，或者用快速排序，堆排序，归并排序等等方法，找出最大前 k 个数也行。

MapReduce

我刚刚除了介绍了一种海量数据的哈希分治算法之外，还穿插了一个谷歌的 MapReduce 分布式并行编程模型，原理就是上面说的那些了，有兴趣的可以去详细了解。

哈希函数是什么？哈希函数是把大空间的元素映射到一个小空间里。

排序算法时间复杂度

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性	复杂性
直接插入排序	O(n2)	O(n2)	O(n)	O(1)	稳定	简单
希尔排序	O(nlog2n)	O(n2)	O(n)	O(1)	不稳定	较复杂
直接选择排序	O(n2)	O(n2)	O(n2)	O(1)	不稳定	简单
堆排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(1)	不稳定	较复杂
冒泡排序	O(n2)	O(n2)	O(n)	O(1)	稳定	简单
快速排序	O(nlog2n)	O(n2)	O(nlog2n)	O(nlog2n)	不稳定	较复杂
归并排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(n)	稳定	较复杂
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(n+r)	稳定	较复杂

判断链表中是否有环 ----- 有关单链表中环的问题

给定一个单链表，判断其中是否有环，已经是一个比较老同时也是比较经典的问题，在网上搜集了一些资料，

然后总结一下大概可以涉及到的问题，以及相应的解法。

首先，关于单链表中的环，一般涉及到一下问题：

1.给一个单链表，判断其中是否有环的存在；

2.如果存在环，找出环的入口点；

3.如果存在环，求出环上节点的个数；

4.如果存在环，求出链表的长度；

5.如果存在环，求出环上距离任意一个节点最远的点（对面节点）；

6.（扩展）如何判断两个无环链表是否相交；

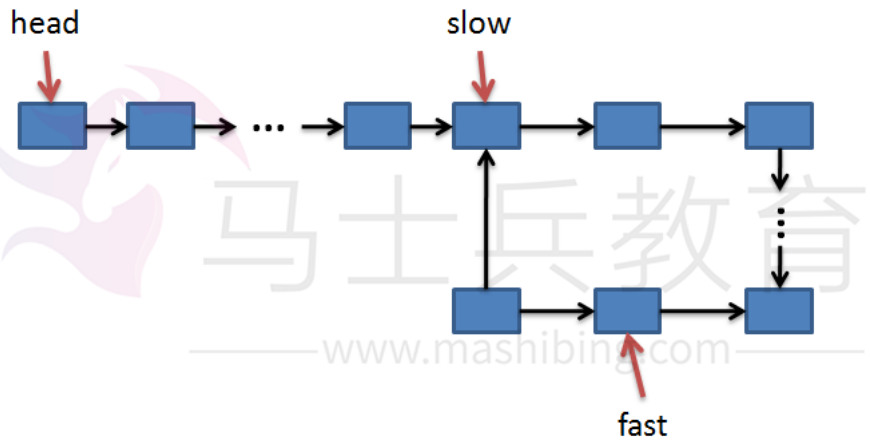
7.（扩展）如果相交，求出第一个相交的节点；

下面，我将针对上面这七个问题一一给出解释和相应的代码。

1.判断时候有环（链表头指针为head）

对于这个问题我们可以采用“快慢指针”的方法。就是有两个指针fast和slow，开始的时候两个指针都指向链表头head，然后在每一步操作中slow向前走一步即：slow = slow->next，而fast每一步向前两步即：fast = fast->next->next。

由于fast要比slow移动的快，如果有环，fast一定会先进入环，而slow后进入环。当两个指针都进入环之后，经过一定步的操作之后二者一定能够在环上相遇，并且此时slow还没有绕环一圈，也就是说一定是在slow走完第一圈之前相遇。证明可以看下图：



马士兵教育
www.mashibing.com

当slow刚进入环时每个指针可能处于上面的情况，接下来slow和fast分别向前走即：

```
if (slow != NULL && fast->next != NULL)
{
         slow = slow -> next ;
         fast = fast -> next -> next ;
}
```

也就是说，slow每次向前走一步，fast向前追了两步，因此每一步操作后fast到slow的距离缩短了1步，这样继续下去就会使得

两者之间的距离逐渐缩小：...、5、4、3、2、1、0 -> 相遇。又因为在同一个环中fast和slow之间的距离不会大于换的长度，因此到二者相遇的时候slow一定还没有走完一周（或者正好走完以后，这种情况出现在开始的时候fast和slow都在环的入口处）。

下面给出问题1的完整代码：

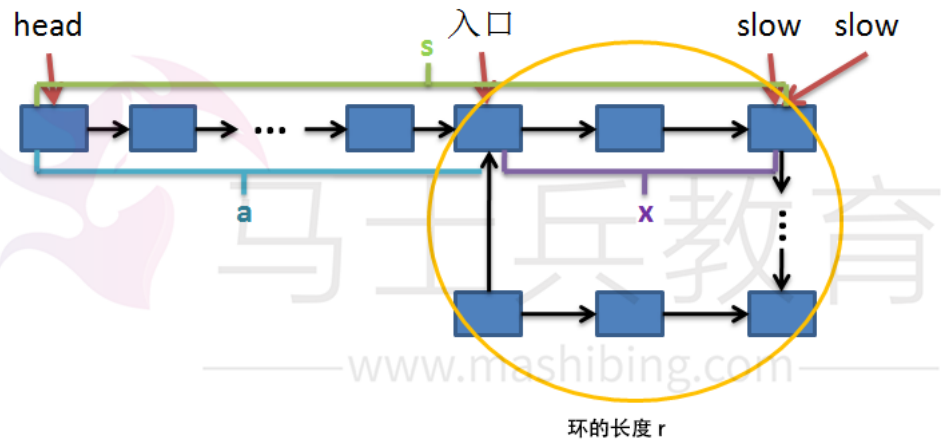
```
typedef struct node{
    char data ;
    node * next ;
}Node;
bool exitLoop(Node *head)
{
    Node *fast, *slow ;
    slow = fast = head ;

    while (slow != NULL && fast -> next != NULL)
    {
        slow = slow -> next ;
        fast = fast -> next -> next ;
        if (slow == fast)
```

```
            return true ;
    }
    return false ;
}
```

下面看问题2，找出环的入口点：

我结合着下图讲解一下：



从上面的分析知道，当fast和slow相遇时，slow还没有走完链表，假设fast已经在环内循环了n(1<= n)圈。假设slow走了s步，则fast走了2s步，又由于fast走过的步数 = s + n*r（s + 在环上多走的n圈），则有下面的等式：

2*s = s + n * r ; (1)

=> s = n*r (2)

如果假设整个链表的长度是L，入口和相遇点的距离是x（如上图所示），起点到入口点的距离是a(如上图所示)，则有：

a + x = s = n * r; (3) 由（2）推出

a + x = (n - 1) * r + r = (n - 1) * r + (L - a) (4) 由环的长度 = 链表总长度 - 起点到入口点的距离求出

a = (n - 1) * r + (L -a -x) (5)

集合式子（5）以及上图我们可以看出，从链表起点head开始到入口点的距离a,与从slow和fast的相遇点（如图）到入口点的距离相等。

因此我们就可以分别用一个指针（ptr1, prt2），同时从head与slow和fast的相遇点出发，每一次操作走一步，直到ptr1 == ptr2，此时的位置也就是入口点！

到此第二个问题也已经解决。

下面给出示意性的简单代码（没有测试但是应该没有问题）：

```
Node* findLoopStart(Node *head)
{
    Node *fast, *slow ;
    slow = fast = head ;

    while (slow != NULL && fast -> next != NULL)
    {
        slow = slow -> next ;
        fast = fast -> next -> next ;
        if (slow == fast) break ;
    }
    if (slow == NULL || fast -> next == NULL) return NULL ; //没有环，返回NULL值

    Node * ptr1 = head ; //链表开始点
    Node * ptr2 = slow ; //相遇点
    while (ptr1 != ptr2)
    {
        ptr1 = ptr1 -> next ;
        ptr2 = ptr2 -> next ;
    }
    return ptr1 ; //找到入口点
}
```

第3个问题，如果存在环，求环上节点的个数：

对于这个问题，我这里有两个思路（肯定还有其它跟好的办法）：

思路1：记录下相遇节点存入临时变量tempPtr，然后让slow(或者fast，都一样)继续向前走slow = slow -> next；一直到slow == tempPtr; 此时经过的步数就是环上节点的个数；

思路2： 从相遇点开始slow和fast继续按照原来的方式向前走slow = slow -> next; fast = fast -> next -> next；直到二者再次项目，此时经过的步数就是环上节点的个数 。

第一种思路很简单，其实就是一次遍历链表的环，从而统计出点的个数，没有什么可以详细解释的了。

对于第二种思路，我们可以这样想，结合上面的分析，fast和slow没一次操作都会使得两者之间的距离较少1。我们可以把两者相遇的时候看做两者之间的距离正好是整个环的长度r。因此，当再次相遇的时候所经过的步数正好是环上节点的数目。

由于这两种思路都比较简单，代码也很容易实现，这里就不给出了。

问题4是如果存在环，求出链表的长度：

到这里，问题已经简单的多了，因为我们在问题1、2、3中已经做得足够的"准备工作"。

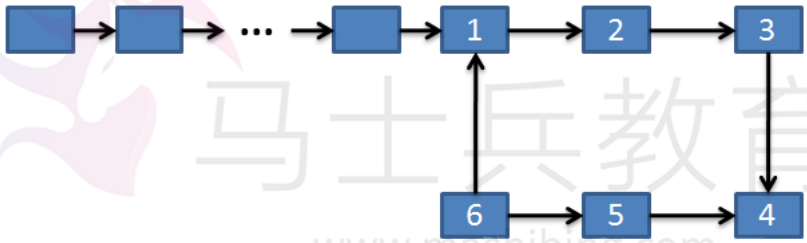
我们可以这样求出整个链表的长度：

链表长度L = 起点到入口点的距离 + 环的长度r;

已经知道了起点和入口点的位置，那么两者之间的距离很好求了吧！环的长度也已经知道了，因此该问题也就迎刃而解了！

问题5是，求出环上距离任意一个节点最远的点（对面节点）

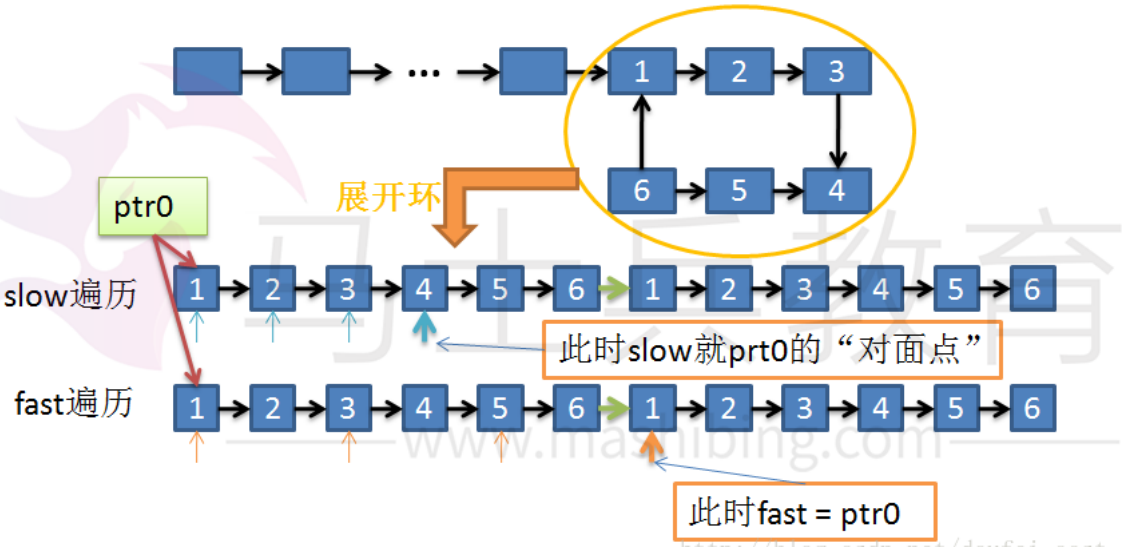
如下图所示，点1和4、点2和5、点3和6分别互为"对面节点"，也就是换上最远的点，我们的要求是怎么求出换上任意一个点的最远点。



对于换上任意的一个点ptr0, 我们要找到它的"对面点"，可以这样思考：同样使用上面的快慢指针的方法，让slow和fast都指向ptr0，每一步都执行与上面相同的操作（slow每次跳一步，fast每次跳两步），

当fast = ptr0或者fast = prt0->next的时候slow所指向的节点就是ptr0的"对面节点"。

为什么是这样呢？我们可以这样分析：



如上图，我们想像一下，把环从ptro处展开，展开后可以是无限长的（如上在6后重复前面的内容）如上图。

现在问题就简单了，由于slow移动的距离永远是fast的一般，因此当fast遍历玩整个环长度r个节点的时候slow正好遍历了r/2个节点，

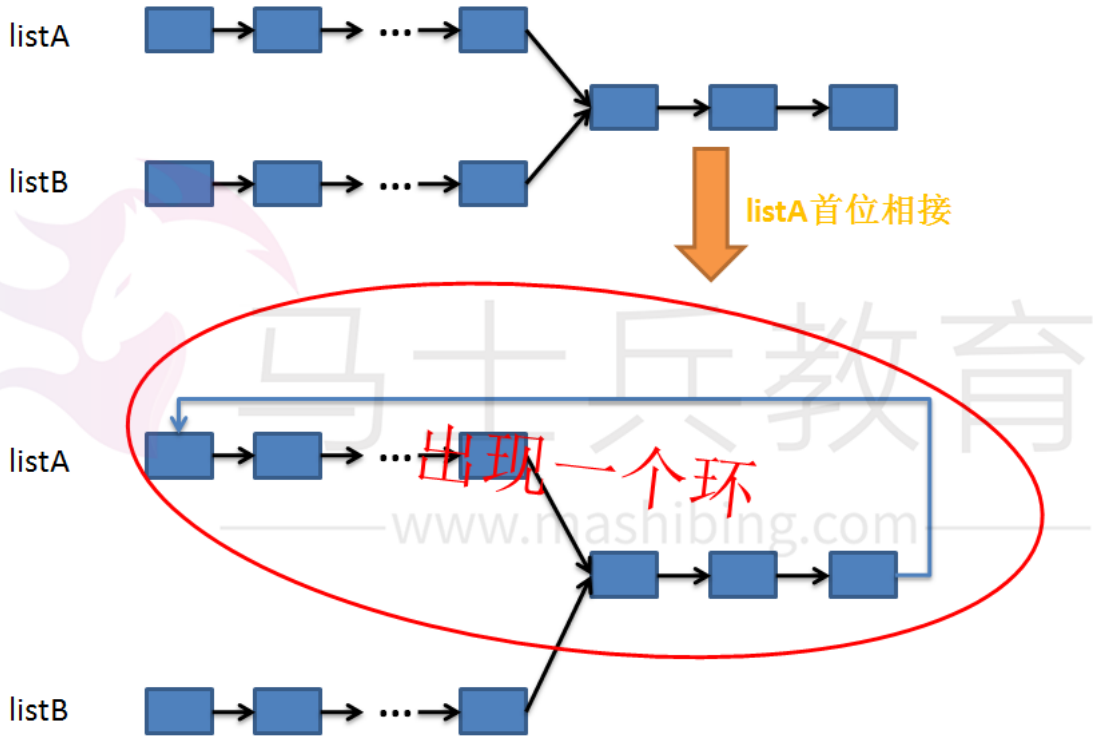
也就是说，此时正好指向距离ptr0最远的点。

对于问题6（扩展）如何判断两个无环链表是否相交，和7（扩展）如果相交，求出第一个相交的节点，其实就是做一个问题的转化：

假设有连个链表listA和listB，如果两个链表都无环，并且有交点，那么我们可以让其中一个链表（不妨设是listA）的为节点连接到其头部，这样在listB中就一定会出现一个环。

因此我们将问题6和7分别转化成了问题1和2.

看看下图就会明白了：



常见hash算法的原理

散列表,它是基于**快速存取**的角度设计的，也是一种典型的"**空间换时间**"的做法。顾名思义，该数据结构可以理解为一个线性表，但是其中的元素不是紧密排列的，而是可能存在空隙。

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

比如我们存储70个元素，但我们可能为这70个元素申请了100个元素的空间。70/100=0.7，这个数字称为负载因子。我们之所以这样做，也 是为了"快速存取"的目的。我们基于一种结果尽可能随机平均分布的固定函数H为每个元素安排存储位置，这样就可以避免遍历性质的线性搜索，以达到快速存 取。但是由于此随机性，也必然导致一个问题就是冲突。所谓冲突，即两个元素通过散列函数H得到的地址相同，那么这两个元素称为"同义词"。这类似于70个 人去一个有100个椅子的饭店吃饭。散列函数的计算结果是一个存储单位地址，每个存储单位称为"桶"。设一个散列表有m个桶，则散列函数的值域应为 [0,m-1]。

解决冲突是一个复杂问题。冲突主要取决于：

（1）散列函数，一个好的散列函数的值应尽可能平均分布。

（2）处理冲突方法。

（3）负载因子的大小。太大不一定就好，而且浪费空间严重，负载因子和散列函数是联动的。

解决冲突的办法：

（1）线性探查法：冲突后，线性向前试探，找到最近的一个空位置。缺点是会出现堆积现象。存取时，可能不是同义词的词也位于探查序列，影响效率。

（2）双散列函数法：在位置d冲突后，再次使用另一个散列函数产生一个与散列表桶容量m互质的数c，依次试探(d+n*c)%m，使探查序列跳跃式分布。

常用的构造散列函数的方法

散列函数能使对一个数据序列的访问过程更加迅速有效，通过散列函数，数据元素将被更快地定位：

1. 直接寻址法：取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a?key + b，其中a和b为常数（这种散列函数叫做自身函数）

2. 数字分析法：分析一组数据，比如一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体相 同，这样的话，出现冲突的几率就会很大，但是我们发现年月日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成散列地址，则冲突的几率会 明显降低。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址。

3. 平方取中法：取关键字平方后的中间几位作为散列地址。

4. 折叠法：将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。

5. 随机数法：选择一随机函数，取关键字的随机值作为散列地址，通常用于关键字长度不同的场合。

6. 除留余数法：取关键字被某个不大于散列表表长m的数p除后所得的余数为散列地址。即 H(key) = key MOD p, p<=m。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对p的选择很重要，一般取素数或m，若p选的不好，容易产生同义词。

查找的性能分析

散列表的查找过程基本上和造表过程相同。一些关键码可通过散列函数转换的地址直接找到，另一些关键码在散列函数得到的地址上产生了冲突，需要按 处理冲突的方法进行查找。在介绍的三种处理冲突的方法中，产生冲突后的查找仍然是给定值与关键码进行比较的过程。所以，对散列表查找效率的量度，依然用平 均查找长度来衡量。

查找过程中，关键码的比较次数，取决于产生冲突的多少，产生的冲突少，查找效率就高，产生的冲突多，查找效率就低。因此，影响产生冲突多少的因素，也就是影响查找效率的因素。影响产生冲突多少有以下三个因素：

```
1．散列函数是否均匀；

2．处理冲突的方法；

3．散列表的装填因子。
```

散列表的装填因子定义为：α= 填入表中的元素个数 / 散列表的长度

α是散列表装满程度的标志因子。由于表长是定值，α与“填入表中的元素个数”成正比，所以，α越大，填入表中的元素较多，产生冲突的可能性就越大；α越小，填入表中的元素较少，产生冲突的可能性就越小。

实际上，散列表的平均查找长度是装填因子α的函数，只是不同处理冲突的方法有不同的函数。

了解了hash基本定义，就不能不提到一些著名的hash算法，MD5 和 SHA-1 可以说是目前应用最广泛的Hash算法，而它们都是以 MD4 为基础设计的。那么他们都是什么意思呢?

这里简单说一下：

(1) MD4

MD4(RFC 1320)是 MIT 的 Ronald L. Rivest 在 1990 年设计的，MD 是 Message Digest 的缩写。它适用在32位字长的处理器上用高速软件实现--它是基于 32 位操作数的位操作来实现的。

(2) MD5

MD5(RFC 1321)是 Rivest 于1991年对MD4的改进版本。它对输入仍以512位分组，其输出是4个32位字的级联，与 MD4 相同。MD5比MD4来得复杂，并且速度较之要慢一点，但更安全，在抗分析和抗差分方面表现更好

(3) SHA-1 及其他

SHA1是由NIST NSA设计为同DSA一起使用的，它对长度小于264的输入，产生长度为160bit的散列值，因此抗穷举(brute-force)性更好。SHA-1 设计时基于和MD4相同原理，并且模仿了该算法。

哈希表不可避免冲突(collision)现象：**对不同的关键字可能得到同一哈希地址** 即key1≠key2，而hash(key1)=hash(key2)。因此，在建造哈希表时不仅要设定一个好的哈希函数，而且要设定一种处理冲突的方法。可如下描述**哈希表**：根据设定的哈希**函数**H(key)和所选中的**处理冲突的方法**，将一组关键字映象到一个**有限的**、**地址连续**的地址集(区间)上并以**关键字在地址集中的“象”作为相应记录在表中的存储位置**，这种表被称为哈希表。

对于动态查找表而言，1) 表长不确定；2)在设计查找表时，只知道关键字所属范围，而不知道确切的关键字。因此，一般情况需建立一个函数关系，以f(key)作为关键字为key的 录在表中的位置，通常称这个函数f(key)为哈希函数。(注意：这个函数并不一定是数学函数)

哈希函数是一个映象，即：将关键字的集合映射到某个地址集合上，它的设置很灵活，只要这个地址集合的大小不超出允许范围即可。

现实中哈希函数是需要构造的，并且构造的好才能使用的好。

那么这些Hash算法到底有什么用呢?

Hash算法在信息安全方面的应用主要体现在以下的3个方面：

(1) 文件校验

我们比较熟悉的校验算法有奇偶校验和CRC校验，这2种校验并没有抗数据篡改的能力，它们一定程度上能检测并纠正数据传输中的信道误码，但却不能防止对数据的恶意破坏。

MD5 Hash算法的"数字指纹"特性，使它成为目前应用最广泛的一种文件完整性校验和(Checksum)算法，不少Unix系统有提供计算md5 checksum的命令。

(2) 数字签名

Hash 算法也是现代密码体系中的一个重要组成部分。由于非对称算法的运算速度较慢，所以在数字签名协议中，单向散列函数扮演了一个重要的角色。 对 Hash 值，又称"数字摘要"进行数字签名，在统计上可以认为与对文件本身进行数字签名是等效的。而且这样的协议还有其他的优点。

(3) 鉴权协议

如下的鉴权协议又被称作挑战--认证模式：在传输信道是可被侦听，但不可被篡改的情况下，这是一种简单而安全的方法。

文件hash值

MD5-Hash-文件的数字文摘通过Hash函数计算得到。不管文件长度如何，它的Hash函数计算结果是一个固定长度的数字。与加密算法不 同，这一个Hash算法是一个不可逆的单向函数。采用安全性高的Hash算法，如MD5、SHA时，两个不同的文件几乎不可能得到相同的Hash结果。因 此，一旦文件被修改，就可检测出来。

Hash函数还有另外的含义。实际中的Hash函数是指把一个大范围映射到一个小范围。把大范围映射到一个小范围的目的往往是为了节省空间，使得数据容易保存。除此以外，Hash函数往往应用于查找上。所以，在考虑使用Hash函数之前，需要明白它的几个限制：

1. Hash的主要原理就是把大范围映射到小范围；所以，你输入的实际值的个数必须和小范围相当或者比它更小。不然冲突就会很多。
2. 由于Hash逼近单向函数；所以，你可以用它来对数据进行加密。
3. 不同的应用对Hash函数有着不同的要求；比如，用于加密的Hash函数主要考虑它和单项函数的差距，而用于查找的Hash函数主要考虑它映射到小范围的冲突率。

应用于加密的Hash函数已经探讨过太多了，在作者的博客里面有更详细的介绍。所以，本文只探讨用于查找的Hash函数。
Hash函数应用的主要对象是数组（比如，字符串），而其目标一般是一个int类型。以下我们都按照这种方式来说明。
一般的说，Hash函数可以简单的划分为如下几类：

1. 加法Hash；
2. 位运算Hash；
3. 乘法Hash；
3. 除法Hash；
5. 查表Hash；
4. 混合Hash；

下面详细的介绍以上各种方式在实际中的运用。

一 加法Hash

所谓的加法Hash就是把输入元素一个一个的加起来构成最后的结果。标准的加法Hash的构造如下：

```
static int additiveHash(String key, int prime)
{
 int hash, i;
 for (hash = key.length(), i = 0; i < key.length(); i++)
 hash += key.charAt(i);
 return (hash % prime);
}
```

这里的prime是任意的质数，看得出，结果的值域为[0,prime-1]。

二 位运算Hash

这类型Hash函数通过利用各种位运算（常见的是移位和异或）来充分的混合输入元素。比如，标准的旋转Hash的构造如下：

```
static int rotatingHash(String key, int prime)
{
 int hash, i;
 for (hash=key.length(), i=0; i
  hash = (hash<<4>>28)^key.charAt(i);
 return (hash % prime);
}
```

先移位，然后再进行各种位运算是这种类型Hash函数的主要特点。比如，以上的那段计算hash的代码还可以有如下几种变形：

```
hash = (hash<<5>>27)^key.charAt(i);
hash += key.charAt(i);
hash += (hash << 10);
hash ^= (hash >> 6);
if((i&1) == 0)
{
hash ^= (hash<<7>>3);
 }
else
 {
 hash ^= ~((hash<<11>>5));
 }
hash += (hash<<5>
hash = key.charAt(i) + (hash<<6>>16) ? hash;
hash ^= ((hash<<5>>2));
```

三 乘法Hash

这种类型的Hash函数利用了乘法的不相关性（乘法的这种性质，最有名的莫过于平方取头尾的随机数生成算法，虽然这种算法效果并不好）。比如，

```
static int bernstein(String key)
{
 int hash = 0;
 int i;
 for (i=0; i
 return hash;
}
```

jdk5.0里面的String类的hashCode()方法也使用乘法Hash。不过，它使用的乘数是31。推荐的乘数还有：131, 1313, 13131, 131313等等。
使用这种方式的著名Hash函数还有：

```
// 32位FNV算法
int M_SHIFT = 0;
 public int FNVHash(byte[] data)
 {
   int hash = (int)2166136261L;
   for(byte b : data)
     hash = (hash * 16777619) ^ b;
   if (M_SHIFT == 0)
     return hash;
   return (hash ^ (hash >> M_SHIFT)) & M_MASK;
}
```

以及改进的FNV算法：

```
public static int FNVHash1(String data)
{
   final int p = 16777619;
   int hash = (int)2166136261L;
   for(int i=0;i
     hash = (hash ^ data.charAt(i)) * p;
```

```
    hash += hash << 13;
    hash ^= hash >> 7;
    hash += hash << 3;
    hash ^= hash >> 17;
    hash += hash << 5;
    return hash;
}
```

除了乘以一个固定的数，常见的还有乘以一个不断改变的数，比如：

```
static int RSHash(String str)
{
    int b  = 378551;
    int a  = 63689;
    int hash = 0;

   for(int i = 0; i < str.length(); i++)
   {
     hash = hash * a + str.charAt(i);
     a  = a * b;
   }
   return (hash & 0x7FFFFFFF);
}
```

虽然Adler32算法的应用没有CRC32广泛，不过，它可能是乘法Hash里面最有名的一个了。关于它的介绍，大家可以去看RFC 1950规范。

四 除法Hash
除法和乘法一样，同样具有表面上看起来的不相关性。不过，因为除法太慢，这种方式几乎找不到真正的应用。需要注意的是，我们在前面看到的hash的 结果除以一个prime的目的只是为了保证结果的范围。如果你不需要它限制一个范围的话，可以使用如下的代码替代"hash%prime"： hash = hash ^ (hash>>10) ^ (hash>>20)。
**五 查表Hash

**查表Hash最有名的例子莫过于CRC系列算法。虽然CRC系列算法本身并不是查表，但是，查表是它的一种最快的实现方式。下面是CRC32的实现：

```
static int crctab[256] = {
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b, 0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef,
0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236, 0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92,
0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785,
0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8,
0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777, 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9, 0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};
int crc32(String key, int hash)
{
int i;
for (hash=key.length(), i=0; i
 hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k.charAt(i)];
return hash;
}
```

查表Hash中有名的例子有：Universal Hashing和Zobrist Hashing。他们的表格都是随机生成的。

六 混合Hash
混合Hash算法利用了以上各种方式。各种常见的Hash算法，比如MD5、Tiger都属于这个范围。它们一般很少在面向查找的Hash函数里面使用。

七大查找算法

查找是在大量的信息中寻找一个特定的信息元素，在计算机应用中，查找是常用的基本运算，例如编译程序中符号表的查找。本文简单概括性的介绍了常见的七种查找算法，说是七种，其实二分查找、插值查找以及斐波那契查找都可以归为一类——插值查找。插值查找和斐波那契查找是在二分查找的基础上的优化查找算法。树表查找和哈希查找会在后续的博文中进行详细介绍。

查找定义：根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）。

查找算法分类：

1）静态查找和动态查找；

注：静态或者动态都是针对查找表而言的。动态表指查找表中有删除和插入操作的表。

2）无序查找和有序查找。

无序查找：被查找数列有序无序均可；

有序查找：被查找数列必须为有序数列。

平均查找长度（Average Search Length，ASL）：需和指定key进行比较的关键字的个数的期望值，称为查找算法在查找成功时的平均查找长度。

对于含有n个数据元素的查找表，查找成功的平均查找长度为：ASL = Pi*Ci的和。
Pi：查找表中第i个数据元素的概率。
Ci：找到第i个数据元素时已经比较过的次数。

1. 顺序查找

说明：顺序查找适合于存储结构为顺序存储或链接存储的线性表。

基本思想：顺序查找也称为线形查找，属于无序查找算法。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值k相比较，若相等则表示查找成功；若扫描结束仍没有找到关键字等于k的结点，表示查找失败。

复杂度分析：

查找成功时的平均查找长度为：（假设每个数据元素的概率相等） ASL = 1/n(1+2+3+...+n) = (n+1)/2 ;
当查找不成功时，需要n+1次比较，时间复杂度为O(n);

所以，**顺序查找的时间复杂度为O(n**)。**

C++实现源码：

```
//顺序查找
int SequenceSearch(int a[], int value, int n)
{
    int i;
    for(i=0; i<n; i++)
        if(a[i]==value)
            return i;
    return -1;
}
```

2. 二分查找

说明：元素必须是有序的，如果是无序的则要先进行排序操作。

基本思想：也称为是折半查找，属于有序查找算法。用给定值k先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据k与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。

复杂度分析：最坏情况下，关键词比较次数为log2(n+1)，且**期望时间复杂度为O(log2n)**；

注：**折半查找的前提条件是需要有序表顺序存储，对于静态查找表，一次排序后不再变化，折半查找能得到不错的效率。但对于需要****频繁执行插入或删除操作的数据集来说，维护有序的排序会带来不小的工作量，那就不建议使用。——《大话数据结构》**

C++实现源码：

```
//二分查找（折半查找），版本1
int BinarySearch1(int a[], int value, int n)
{
    int low, high, mid;
    low = 0;
    high = n-1;
    while(low<=high)
    {
        mid = (low+high)/2;
        if(a[mid]==value)
            return mid;
        if(a[mid]>value)
            high = mid-1;
        if(a[mid]<value)
            low = mid+1;
    }
    return -1;
}

//二分查找，递归版本
int BinarySearch2(int a[], int value, int low, int high)
```

```
{
    int mid = low+(high-low)/2;
    if(a[mid]==value)
        return mid;
    if(a[mid]>value)
        return BinarySearch2(a, value, low, mid-1);
    if(a[mid]<value)
        return BinarySearch2(a, value, mid+1, high);
}
```

3. 插值查找

在介绍插值查找之前，首先考虑一个新问题，为什么上述算法一定要是折半，而不是折四分之一或者折更多呢?

打个比方，在英文字典里面查“apple”，你下意识翻开字典是翻前面的书页还是后面的书页呢？如果再让你查“zoo”，你又怎么查？很显然，这里你绝对不会是从中间开始查起，而是有一定目的的往前或往后翻。

同样的，比如要在取值范围1 ~ 10000 之间 100 个元素从小到大均匀分布的数组中查找5， 我们自然会考虑从数组下标较小的开始查找。

经过以上分析，折半查找这种查找方式，不是自适应的（也就是说是傻瓜式的）。二分查找中查找点计算如下：

mid=(low+high)/2, 即mid=low+1/2*(high-low);

通过类比，我们可以将查找的点改进为如下：

mid=low+(key-a[low])/(a[high]-a[low])*(high-low)，

也就是将上述的比例参数1/2改进为自适应的，根据关键字在整个有序表中所处的位置，让mid值的变化更靠近关键字key，这样也就间接地减少了比较次数。

基本思想：基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。当然，差值查找也属于有序查找。

注：**对于表长较大，而关键字分布又比较均匀的查找表来说，插值查找算法的平均性能比折半查找要好的多。反之，数组中如果分布非常不均匀，那么插值查找未必是很合适的选择。**

复杂度分析：查找成功或者失败的时间复杂度均为O(log2(log2n))。

C++实现源码：

```
//插值查找
int InsertionSearch(int a[], int value, int low, int high)
{
    int mid = low+(value-a[low])/(a[high]-a[low])*(high-low);
    if(a[mid]==value)
        return mid;
    if(a[mid]>value)
        return InsertionSearch(a, value, low, mid-1);
    if(a[mid]<value)
        return InsertionSearch(a, value, mid+1, high);
}
```

4. 斐波那契查找

在介绍斐波那契查找算法之前，我们先介绍一下很它紧密相连并且大家都熟知的一个概念——黄金分割。

黄金比例又称黄金分割，是指事物各部分间一定的数学比例关系，即将整体一分为二，较大部分与较小部分之比等于整体与较大部分之比，其比值约为1:0.618或1.618:1。

0.618被公认为最具有审美意义的比例数字，这个数值的作用不仅仅体现在诸如绘画、雕塑、音乐、建筑等艺术领域，而且在管理、工程设计等方面也有着不可忽视的作用。因此被称为黄金分割。

大家记不记得斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.....（从第三个数开始，后边每一个数都是前两个数的和）。然后我们会发现，随着斐波那契数列的递增，前后两个数的比值会越来越接近0.618，利用这个特性，我们就可以将黄金比例运用到查找技术中。

基本思想：也是二分查找的一种提升算法，通过运用黄金比例的概念在数列中选择查找点进行查找，提高查找效率。同样地，斐波那契查找也属于一种有序查找算法。

相对于折半查找，一般将待比较的key值与第mid=（low+high）/2位置的元素比较，比较结果分三种情况：

1）相等，mid位置的元素即为所求

2）>，low=mid+1;

3）<，high=mid-1。

斐波那契查找与折半查找很相似，他是根据斐波那契序列的特点对有序表进行分割的。他要求开始表中记录的个数为某个斐波那契数小1，及n=F(k)-1;

开始将k值与第F(k-1)位置的记录进行比较(及mid=low+F(k-1)-1),比较结果也分为三种

1）相等，mid位置的元素即为所求

2）>，low=mid+1,k-=2;

说明：low=mid+1说明待查找的元素在[mid+1,high]范围内，k-=2 说明范围[mid+1,high]内的元素个数为n-(F(k-1))= Fk-1-F(k-1)=Fk-F(k-1)-1=F(k-2)-1个，所以可以递归的应用斐波那契查找。

3）<，high=mid-1,k-=1。

说明：low=mid+1说明待查找的元素在[low,mid-1]范围内，k-=1 说明范围[low,mid-1]内的元素个数为F(k-1)-1个，所以可以递归 的应用斐波那契查找。

复杂度分析：最坏情况下，时间复杂度为O(log2n)，且其期望复杂度也为O(log2n)。**

C++实现源码：

```
// 斐波那契查找.cpp

#include "stdafx.h"
#include <memory>
#include  <iostream>
using namespace std;

const int max_size=20;//斐波那契数组的长度

/*构造一个斐波那契数组*/
void Fibonacci(int * F)
{
    F[0]=0;
    F[1]=1;
    for(int i=2;i<max_size;++i)
        F[i]=F[i-1]+F[i-2];
}

/*定义斐波那契查找法*/
int FibonacciSearch(int *a, int n, int key)  //a为要查找的数组,n为要查找的数组长度,key为要查找的关键字
{
  int low=0;
  int high=n-1;

  int F[max_size];
  Fibonacci(F);//构造一个斐波那契数组F

  int k=0;
  while(n>F[k]-1)//计算n位于斐波那契数列的位置
      ++k;

  int  * temp;//将数组a扩展到F[k]-1的长度
  temp=new int [F[k]-1];
  memcpy(temp,a,n*sizeof(int));

  for(int i=n;i<F[k]-1;++i)
     temp[i]=a[n-1];

  while(low<=high)
  {
    int mid=low+F[k-1]-1;
    if(key<temp[mid])
    {
      high=mid-1;
      k-=1;
    }
    else if(key>temp[mid])
    {
     low=mid+1;
     k-=2;
    }
    else
    {
       if(mid<n)
           return mid; //若相等则说明mid即为查找到的位置
       else
           return n-1; //若mid>=n则说明是扩展的数值,返回n-1
    }
  }
  delete [] temp;
  return -1;
}

int main()
{
    int a[] = {0,16,24,35,47,59,62,73,88,99};
    int key=100;
    int index=FibonacciSearch(a,sizeof(a)/sizeof(int),key);
    cout<<key<<" is located at:"<<index;
    return 0;
}
```

5. 树表查找

5.1 最简单的树表查找算法——二叉树查找算法。

基本思想：二叉查找树是先对待查找的数据进行生成树，确保树的左分支的值小于右分支的值，然后在就行和每个节点的父节点比较大小，查找最适合的范围。 这个算法的查找效率很高，但是如果使用这种查找方法要首先创建树。

二叉查找树（BinarySearch Tree，也叫二叉搜索树，或称二叉排序树Binary Sort Tree）或者是一棵空树，或者是具有下列性质的二叉树：

1）若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

2）若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

3）任意节点的左、右子树也分别为二叉查找树。

二叉查找树性质：对二叉查找树进行中序遍历，即可得到有序的数列。

不同形态的二叉查找树如下图所示：

有关二叉查找树的查找、插入、删除等操作的详细讲解，请移步[浅谈算法和数据结构: 七 二叉查找树]

复杂度分析：它和二分查找一样，插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度。原因在于插入和删除元素的时候，树没有保持平衡（比如，我们查找上图（b）中的“93”，我们需要进行n次查找操作）。我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是平衡查找树设计的初衷。

下图为二叉树查找和顺序查找以及二分查找性能的对比图：

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	√N	yes	`compareTo()`

other operations also become √N if deletions allowed

基于二叉查找树进行优化，进而可以得到其他的树表查找算法，如平衡树、红黑树等高效算法。

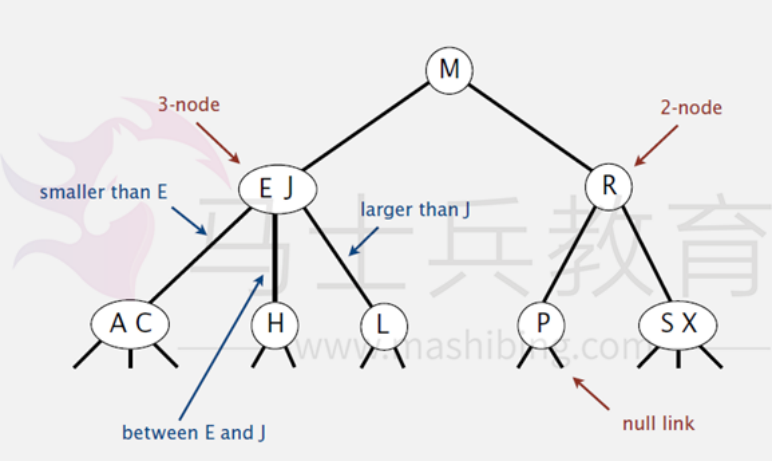
5.2 平衡查找树之2-3查找树（2-3 Tree）

2-3查找树定义：和二叉树不一样，2-3树运行每个节点保存1个或者两个的值。对于普通的2节点(2-node)，他保存1个key和左右两个自己点。对应3节点(3-node)，保存两个Key，2-3查找树的定义如下：

1）要么为空，要么：

2）对于2节点，该节点保存一个key及对应value，以及两个指向左右节点的节点，左节点也是一个2-3节点，所有的值都比key要小，右节点也是一个2-3节点，所有的值比key要大。

3）对于3节点，该节点保存两个key及对应value，以及三个指向左中右的节点。左节点也是一个2-3节点，所有的值均比两个key中的最小的key还要小；中间节点也是一个2-3节点，中间节点的key值在两个跟节点key值之间；右节点也是一个2-3节点，节点的所有key值比两个key中的最大的key还要大。



2-3查找树的性质：

1）如果中序遍历2-3查找树，就可以得到排好序的序列；

2）在一个完全平衡的2-3查找树中，根节点到每一个为空节点的距离都相同。（这也是平衡树中“平衡”一词的概念，根节点到叶节点的最长距离对应于查找算法的最坏情况，而平衡树中根节点到叶节点的距离都一样，最坏情况也具有对数复杂度。）

性质2）如下图所示：



复杂度分析：

2-3树的查找效率与树的高度是息息相关的。

- 在最坏的情况下，也就是所有的节点都是2-node节点，查找效率为lgN
- 在最好的情况下，所有的节点都是3-node节点，查找效率为log3N约等于0.631lgN

距离来说，对于1百万个节点的2-3树，树的高度为12-20之间，对于10亿个节点的2-3树，树的高度为18-30之间。

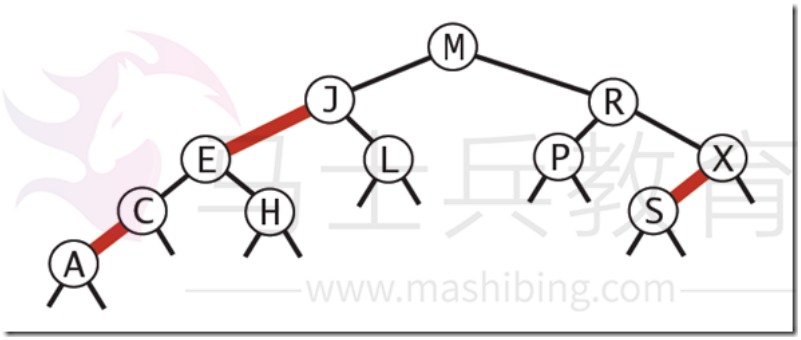
对于插入来说，只需要常数次操作即可完成，因为他只需要修改与该节点关联的节点即可，不需要检查其他节点，所以效率和查找类似。下面是2-3查找树的效率：

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`

5.3 平衡查找树之红黑树（Red-Black Tree）

2-3查找树能保证在插入元素之后能保持树的平衡状态，最坏情况下即所有的子节点都是2-node，树的高度为lgn，从而保证了最坏情况下的时间复杂度。但是2-3树实现起来比较复杂，于是就有了一种简单实现2-3树的数据结构，即红黑树（Red-Black Tree）。

基本思想：红黑树的思想就是对2-3查找树进行编码，尤其是对2-3查找树中的3-nodes节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个2-nodes节点来表示一个3-nodes节点。黑色链接用来链接普通的2-3节点。特别的，使用红色链接的两个2-nodes来表示一个3-nodes节点，并且向左倾斜，即一个2-node是另一个2-node的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。

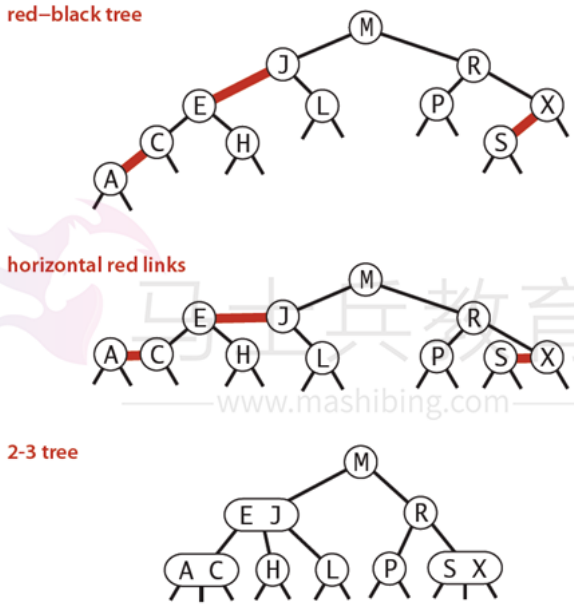


红黑树的定义：

红黑树是一种具有红色和黑色链接的平衡查找树，同时满足：

- 红色节点向左倾斜
- 一个节点不可能有两个红色链接
- 整个树完全黑色平衡，即从根节点到所以叶子结点的路径上，黑色链接的个数都相同。

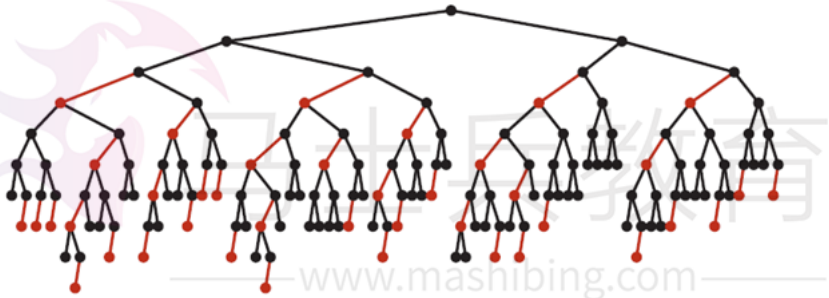
下图可以看到红黑树其实是2-3树的另外一种表现形式：如果我们将红色的连线水平绘制，那么他链接的两个2-node节点就是2-3树中的一个3-node节点了。



红黑树的性质：**整个树完全黑色平衡，即从根节点到所以叶子结点的路径上，黑色链接的个数都相同（2-3树的第2）性质，从根节点到叶子节点的距离都相等）。**

复杂度分析：**最坏的情况就是，红黑树中除了最左侧路径全部是由3-node节点组成，即红黑相间的路径长度是全黑路径长度的2倍。**

下图是一个典型的红黑树，从中可以看到最长的路径(红黑相间的路径)是最短路径的2倍：



红黑树的平均高度大约为logn。

下图是红黑树在各种情况下的时间复杂度，可以看出红黑树是2-3查找树的一种实现，它能保证最坏情况下仍然具有对数的时间复杂度。

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	`compareTo()`

红黑树这种数据结构应用十分广泛，在多种编程语言中被用作符号表的实现，如：

- Java中的java.util.TreeMap,java.util.TreeSet；
- C++ STL中的：map,multimap,multiset；
- .NET中的：SortedDictionary,SortedSet 等。

5.4 B树和B+树（B Tree/B+ Tree）

平衡查找树中的2-3树以及其实现红黑树。2-3树种，一个节点最多有2个key，而红黑树则使用染色的方式来标识这两个key。

维基百科对B树的定义为“在计算机科学中，B树（B-tree）是一种树状数据结构，它能够存储数据、对其进行排序并允许以O(log n)的时间复杂度运行进行查找、顺序读取、插入和删除的数据结构。B树，概括来说是一个节点可以拥有多于2个子节点的二叉查找树。与自平衡二叉查找树不同，B树为系统最优化**大块数据的读和写操作**。B-tree算法减少定位记录时所经历的中间过程，从而加快存取速度。普遍运用在**数据库**和**文件系统**。

B树定义：

B树可以看作是对2-3查找树的一种扩展，即他允许每个节点有M-1个子节点。

- 根节点至少有两个子节点

- 每个节点有M-1个key，并且以升序排列
- 位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间
- 其它节点至少有M/2个子节点

下图是一个M=4 阶的B树:



可以看到B树是2-3树的一种扩展，他允许一个节点有多于2个的元素。B树的插入及平衡化操作和2-3树很相似，这里就不介绍了。下面是往B树中依次插入

6 10 4 14 5 11 15 3 2 12 1 7 8 8 6 3 6 21 5 15 15 6 32 23 45 65 7 8 6 5 4

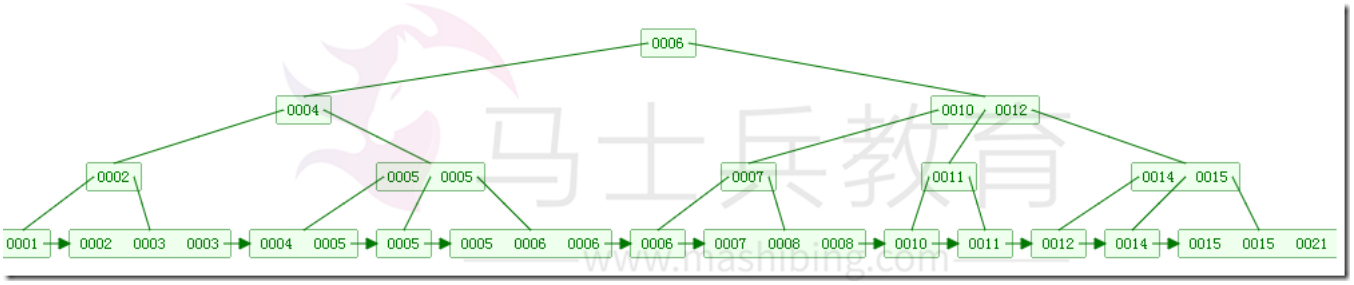
的演示动画:

B+树定义：

B+树是对B树的一种变形树，它与B树的差异在于:

- 有k个子结点的结点必然有k个关键码;
- 非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。
- 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

如下图，是一个B+树:



下图是B+树的插入动画:

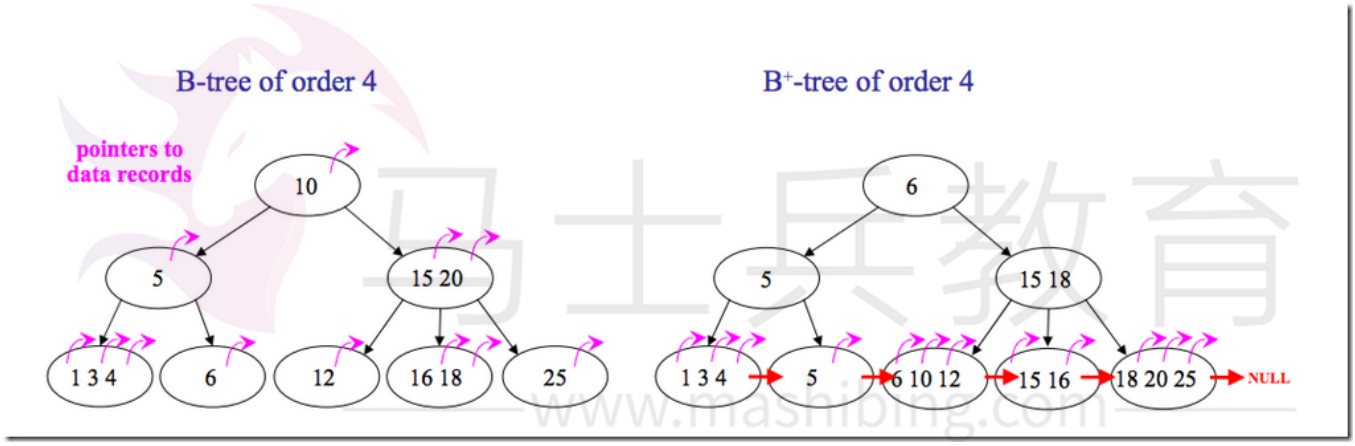
B和B+树的区别在于，B+树的非叶子结点只包含导航信息，不包含实际的值，所有的叶子结点和相连的节点使用链表相连，便于区间查找和遍历。

B+ 树的优点在于:

- 由于B+树在内部节点上不好含数据信息，因此在内存页中能够存放更多的key。 数据存放的更加紧密，具有更好的空间局部性。因此访问叶子几点上关联的数据也具有更好的缓存命中率。
- B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。

但是B树也有优点，其优点在于，由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。

下面是B 树和B+树的区别图:



B/B+树常用于文件系统和数据库系统中，它通过对每个节点存储个数的扩展，使得对连续的数据能够进行较快的定位和访问，能够有效减少查找时间，提高存储的空间局部性从而减少IO操作。它广泛用于文件系统及数据库中，如：

- Windows：HPFS文件系统；
- Mac：HFS，HFS+文件系统；
- Linux：ResiserFS，XFS，Ext3FS，JFS文件系统；
- 数据库：ORACLE，MYSQL，SQLSERVER等中。

有关B/B+树在数据库索引中的应用，请看张洋的[MySQL索引背后的数据结构及算法原理]这篇文章，这篇文章对MySQL中的如何使用B+树进行索引有比较详细的介绍，推荐阅读。

树表查找总结：

二叉查找树平均查找性能不错，为O(logn)，但是最坏情况会退化为O(n)。在二叉查找树的基础上进行优化，我们可以使用平衡查找树。平衡查找树中的2-3查找树，这种数据结构在插入之后能够进行自平衡操作，从而保证了树的高度在一定的范围内进而能够保证最坏情况下的时间复杂度。但是2-3查找树实现起来比较困难，红黑树是2-3树的一种简单高效的实现，他巧妙地使用颜色标记来替代2-3树中比较难处理的3-node节点问题。红黑树是一种比较高效的平衡查找树，应用非常广泛，很多编程语言的内部实现都或多或少的采用了红黑树。

除此之外，2-3查找树的另一个扩展——B/B+平衡树，在文件系统和数据库系统中有着广泛的应用。

6. 分块查找

分块查找又称索引顺序查找，它是顺序查找的一种改进方法。

算法思想：将n个数据元素"按块有序"划分为m块（m ≤ n）。每一块中的结点不必有序，但块与块之间必须"按块有序"；即第1块中任一元素的关键字都必须小于第2块中任一元素的关键字；而第2块中任一元素又都必须小于第3块中的任一元素，……

算法流程：

step1 先选取各块中的最大关键字构成一个索引表；

step2 查找分两个部分：先对索引表进行二分查找或顺序查找，以确定待查记录在哪一块中；然后，在已确定的块中用顺序法进行查找。

7. 哈希查找

什么是哈希表（Hash）？

我们使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数， 也叫做散列函数），使得每个元素的关键字都与一个函数值（即数组下标）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照关键字为每一个元素"分类"，然后将这个元素存储在相应"类"所对应的地方。但是，不能够保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了"冲突"，换句话说，就是把不同的元素分在了相同的"类"之中。后面我们将看到一种解决"冲突"的简便做法。

总的来说，"直接定址"与"解决冲突"是哈希表的两大特点。

什么是哈希函数?

哈希函数的规则是：通过某种转换关系，使关键字适度的分散到指定大小的的顺序结构中，越分散，则以后查找的时间复杂度越小，空间复杂度越高。

算法思想：哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。

算法流程：

1）用给定的哈希函数构造哈希表；

2）根据选择的冲突处理方法解决地址冲突；

3）在哈希表的基础上执行哈希查找。

哈希表是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为O(1)；如果没有时间限制，那么我们可以使用无序数组并进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个极端之间找到了平衡。只需要调整哈希函数算法即可在时间和空间上做出取舍。

复杂度分析：

单纯论查找复杂度：对于无冲突的Hash表而言，查找复杂度为O(1)（注意，在查找之前我们需要构建相应的Hash表）。

使用Hash，我们付出了什么?

我们在实际编程中存储一个大规模的数据，最先想到的存储结构可能就是map，也就是我们常说的KV pair，经常使用Python的博友可能更有这种体会。使用map的好处就是，我们在后续处理数据处理时，可以根据数据的key快速的查找到对应的value值。map的本质就是Hash表，那我们在获取了超高查找效率的基础上，我们付出了什么?

Hash是一种典型**以空间换时间**的算法，比如原来一个长度为100的数组，对其查找，只需要遍历且匹配相应记录即可，从空间复杂度上来看，假如数组存储的是byte类型数据，那么该数组占用100byte空间。现在我们采用Hash算法，我们前面说的Hash必须有一个规则，约束键与存储位置的关系，那么就需要一个固定长度的hash表，此时，仍然是100byte的数组，假设我们需要的100byte用来记录键与位置的关系，那么总的空间为200byte,而且用于记录规则的表大小会根据规则，大小可能是不定的。

Hash算法和其他查找算法的性能对比：

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`

马士兵教育 www.mashibing.com

一致性算法

Paxos

Paxos 算法解决的问题是一个分布式系统如何就某个值（决议）达成一致。一个典型的场景是，在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点执行相同的操作序列，那么他们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。zookeeper使用的 zab 算法是该算法的一个实现。 在 Paxos 算法中，有三种角色：Proposer，Acceptor，Learners

Paxos 三种角色：**Proposer**，**Acceptor**，**Learners**

Proposer：

只要 Proposer 发的提案被半数以上 Acceptor 接受，Proposer 就认为该提案里的 value 被选定了。

Acceptor：

只要 Acceptor 接受了某个提案，Acceptor 就认为该提案里的 value 被选定了。

Learner：

Acceptor 告诉 Learner 哪个 value 被选定，Learner 就认为那个 value 被选定。

Paxos 算法分为两个阶段。具体如下：

阶段一（准**leader**确定 ）：

(a) Proposer选择一个提案编号N，然后向半数以上的Acceptor发送编号为N的Prepare请求。 (b) 如果一个 Acceptor 收到一个编号为 N 的 Prepare 请求，且 N 大于该 Acceptor 已经响应过的所有 Prepare 请求的编号，那么它就会将它已经接受过的编号最大的提案（如果有的话）作为响应反馈给 Proposer，同时该 Acceptor 承诺不再接受任何编号小于 N 的提案。

阶段二（**leader**确认）：

(a) 如果 Proposer 收到半数以上 Acceptor 对其发出的编号为 N 的 Prepare 请求的响应，那么它就会发送一个针对[N,V]提案的 Accept 请求给半数以上的 Acceptor。注意：V 就是收到的响应中编号最大的提案的 value，如果响应中不包含任何提案，那么 V 就由 Proposer 自己决定。

(b) 如果 Acceptor 收到一个针对编号为 N 的提案的 Accept 请求，只要该 Acceptor 没有对编号大于 N 的 Prepare 请求做出过响应，它就接受该提案。

Zab

ZAB(ZooKeeper Atomic Broadcast , ZooKeeper 原子消息广播协议）协议包括两种基本的模式：崩溃恢复和消息广播

1. 当整个服务框架在启动过程中，或是当 Leader 服务器出现网络中断崩溃退出与重启等异常情况时，ZAB 就会进入恢复模式并选举产生新的 Leader 服务器。
2. 当选举产生了新的 Leader 服务器，同时集群中已经有过半的机器与该 Leader 服务器完成了状态同步之后，ZAB 协议就会退出崩溃恢复模式，进入消息广播模式。
3. 当有新的服务器加入到集群中去，如果此时集群中已经存在一个 Leader 服务器在负责进行消息广播，那么新加入的服务器会自动进入数据恢复模式，找到 Leader 服务器，并与其进行数据同步，然后一起参与到消息广播流程中去。

以上其实大致经历了三个步骤：

1.崩溃恢复：主要就是**Leader** 选举过程

2.数据同步：**Leader**服务器与其他服务器进行数据同步

3.消息广播：**Leader**服务器将数据发送给其他服务器说明：zookeeper 章节对该协议有详细描述。

Raft

与 Paxos 不同 Raft 强调的是易懂（Understandability），Raft 和 Paxos 一样只要保证 n/2+1 节点正常就能够提供服务；raft 把算法流程分为三个子问题：选举（Leader election）、日志复制

（Log replication）、安全性（Safety）三个子问题。

1. 角色

Raft 把集群中的节点分为三种状态：Leader、 Follower 、Candidate，理所当然每种状态负责的任务也是不一样的，Raft 运行时提供服务的时候只存在 Leader 与 Follower 两种状态；

Leader（领导者**-**日志管理）

负责日志的同步管理，处理来自客户端的请求，与 Follower 保持这 heartBeat 的联系；

Follower（追随者**-**日志同步）

刚启动时所有节点为Follower状态，响应Leader的日志同步请求，响应Candidate的请求，把请求到 Follower 的事务转发给 Leader；

Candidate（候选者**-**负责选票）

负责选举投票，Raft 刚启动时由一个节点从 Follower 转为 Candidate 发起选举，选举出

Leader 后从 Candidate 转为 Leader 状态；

2. Term（任期）

在 Raft 中使用了一个可以理解为周期（第几届、任期）的概念，用 Term 作为一个周期，每个 Term 都是一个连续递增的编号，每一轮选举都是一个 Term 周期，在一个 Term 中只能产生一个 Leader；当某节点收到的请求中 Term 比当前 Term 小时则拒绝该请求。

3. 选举（**Election**）

选举定时器

Raft 的选举由定时器来触发，每个节点的选举定时器时间都是不一样的，开始时状态都为 Follower 某个节点定时器触发选举后 Term 递增，状态由 Follower 转为 Candidate，向其他节点发起 RequestVote RPC 请求，这时候有三种可能的情况发生：

1：该RequestVote请求接收到n/2+1（过半数）个节点的投票，从Candidate 转为Leader，向其他节点发送 heartBeat 以保持 Leader 的正常运转。

2：在此期间如果收到其他节点发送过来的 AppendEntries RPC 请求，如该节点的 Term 大则当前节点转为 Follower，否则保持 Candidate 拒绝该请求。

3：Election timeout 发生则 Term 递增，重新发起选举

在一个 Term 期间每个节点只能投票一次，所以当有多个 Candidate 存在时就会出现每个 Candidate 发起的选举都存在接收到的投票数都不过半的问题，这时每个 Candidate 都将 Term 递增、重启定时器并重新发起选举，由于每个节点中定时器的时间都是随机的，所以就不会多次存在有多个 Candidate 同时发起投票的问题。

在 Raft 中当接收到客户端的日志（事务请求）后先把该日志追加到本地的 Log 中，然后通过 heartbeat 把该 Entry 同步给其他 Follower，Follower 接收到日志后记录日志然后向 Leader 发送 ACK，当 Leader 收到大多数（n/2+1）Follower 的 ACK 信息后将该日志设置为已提交并追加到本地磁盘中，通知客户端并在下个 heartbeat 中 Leader 将通知所有的 Follower 将该日志存储在自己的本地磁盘中。

4. 安全性（**Safety**）

安全性是用于保证每个节点都执行相同序列的安全机制如当某个 Follower 在当前 Leader commit Log 时变得不可用了，稍后可能该 Follower 又会倍选举为 Leader，这时新 Leader 可能会用新的 Log 覆盖先前已 committed 的 Log，这就是导致节点执行不同序列；Safety 就是用于保证选举出来的 Leader 一定包含先前 commited Log 的机制；

选举安全性（Election Safety）：每个 Term 只能选举出一个 Leader

Leader 完整性（Leader Completeness）：这里所说的完整性是指 Leader 日志的完整性， Raft 在选举阶段就使用 Term 的判断用于保证完整性：当请求投票的该 Candidate 的 Term 较大或 Term 相同 Index 更大则投票，该节点将容易变成 leader。

5. raft 协议和 **zab** 协议区别**相同点**

§ 采用 quorum 来确定整个系统的一致性,这个 quorum 一般实现是集群中半数以上的服务器,

§ zookeeper 里还提供了带权重的 quorum 实现.

§ 都由 leader 来发起写操作.

§ 都采用心跳检测存活性

§ leader election 都采用先到先得的投票方式**不同点**

§ zab 用的是 epoch 和 count 的组合来唯一表示一个值, 而 raft 用的是 term 和 index

§ zab 的 follower 在投票给一个 leader 之前必须和 leader 的日志达成一致,而 raft 的 follower 则简单地说是谁的 term 高就投票给谁

raft 协议的心跳是从 leader 到 follower, 而 zab 协议则相反

§ raft 协议数据只有单向地从 leader 到 follower(成为 leader 的条件之一就是拥有最新的 log),

而 zab 协议在 discovery 阶段, 一个 prospective leader 需要将自己的 log 更新为 quorum 里面最新的 log,然后才好在 synchronization 阶段将 quorum 里的其他机器的 log 都同步到一致.

NWR

N：在分布式存储系统中，有多少份备份数据

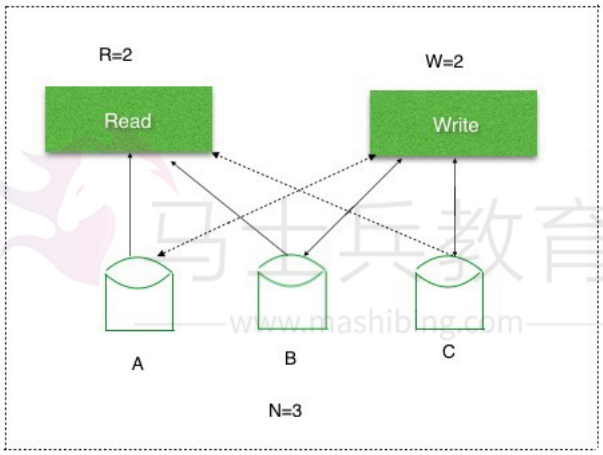
W：代表一次成功的更新操作要求至少有**w**份数据写入成功

R：代表一次成功的读数据操作要求至少有**R**份数据成功读取

NWR值的不同组合会产生不同的一致性效果，当W+R>N的时候，整个系统对于客户端来讲能保

证强一致性。而如果 R+W<=N，则无法保证数据的强一致性。以常见的 N=3、W=2、R=2 为例：

N=3，表示，任何一个对象都必须有三个副本（Replica），W=2 表示，对数据的修改操作（Write）只需要在 3 个 Replica 中的 2 个上面完成就返回，R=2 表示，从三个对象中要读取到 2 个数据对象，才能返回。



Gossip

Gossip 算法又被称为反熵（Anti-Entropy），熵是物理学上的一个概念，代表杂乱无章，而反熵就是在杂乱无章中寻求一致，这充分说明了 Gossip 的特点：在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致。每个节点可能知道所有其他节点，也可能仅知道几个邻居节点，只要这些节可以通过网络连通，最终他们的状态都是一致的，当然这也是疫情传播的特点。

一致性 Hash

一致性哈希算法(Consistent Hashing Algorithm)是一种分布式算法，常用于负载均衡。 Memcached client 也选择这种算法，解决将 key-value 均匀分配到众多 Memcached server 上的问题。它可以取代传统的取模操作，解决了取模操作无法应对增删 Memcached Server 的问题 (增删 server 会导致同一个 key,在 get 操作时分配不到数据真正存储的 server，命中率会急剧下

降)。

一致性 **Hash** 特性

§ 平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。

§ 单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。容易看到，上面的简单求余算法 hash(object)%N 难以满足单调性要求。

§ 平滑性(Smoothness)：平滑性是指缓存服务器的数目平滑改变和缓存对象的平滑改变是一致的。

一致性 **Hash** 原理

1.建构环形**hash** 空间：

1. 考虑通常的 hash 算法都是将 value 映射到一个 32 为的 key 值，也即是 0~2^32-1 次方的数值空间；我们可以将这个空间想象成一个首（0）尾（2^32-1）相接的圆环。

2.把需要缓存的内容**(**对象**)**映射到**hash** 空间

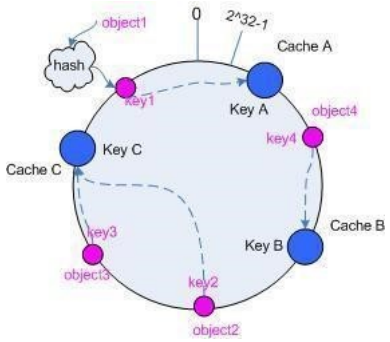
2. 接下来考虑 4 个对象 object1~object4，通过 hash 函数计算出的 hash 值 key 在环上的分布

3.把服务器**(**节点**)**映射到**hash** 空间

3. Consistent hashing 的基本思想就是将对象和 cache 都映射到同一个 hash 数值空间中，并且使用相同的 hash 算法。一般的方法可以使用 服务器(节点) 机器的 IP 地址或者机器名作为 hash 输入。

4.把对象映射到服务节点

4. 现在服务节点和对象都已经通过同一个 hash 算法映射到 hash 数值空间中了，首先确定对象 hash 值在环上的位置，从此位置沿环顺时针"行走"，第一台遇到的服务器就是其应该定位的服务器。



考察**cache** 的变动

5. 通过 hash 然后求余的方法带来的最大问题就在于不能满足单调性，当 cache 有所变动时， cache 会失效。

5.1 移除 cache：考虑假设 cache B 挂掉了，根据上面讲到的映射方法，这时受影响的将仅是那些沿 cache B 逆时针遍历直到下一个 cache （cache C）之间的对象。

5.2 添加 cache：再考虑添加一台新的 cache D 的情况，这时受影响的将仅是那些沿 cache D 逆时针遍历直到下一个 cache 之间的对象，将这些对象重新映射到 cache D 上即可。

虚拟节点

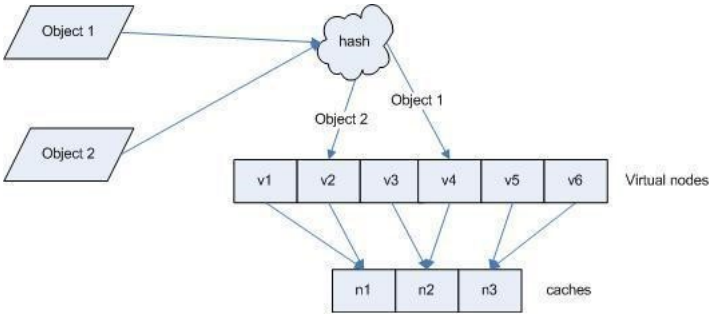
hash 算法并不是保证绝对的平衡，如果 cache 较少的话，对象并不能被均匀的映射到 cache 上，为了解决这种情况， consistent hashing 引入了“虚拟节点”的概念，它可以如下定义：虚拟节点（ virtual node ）是实际节点在 hash 空间的复制品（ replica ），一实际个节点对应了若干个“虚拟节点”，这个对应个数也成为“复制个数”，“虚拟节点”在 hash 空间中以 hash 值排列。

仍以仅部署 cache A 和 cache C 的情况为例。现在我们引入虚拟节点，并设置“复制个数”为 2 ，这就意味着一共会存在 4 个“虚拟节点”， cache A1, cache A2 代表了 cache A；cache C1, cache C2 代表了 cache C 。此时，对象到“虚拟节点”的映射关系为：

objec1->cache A2 ； objec2->cache A1 ； objec3->cache C1 ； objec4->cache C2 ； 因此对象 object1 和 object2 都被映射到了 cache A 上，而 object3 和 object4 映射到了 cache

C 上；平衡性有了很大提高。

引入“虚拟节点”后，映射关系就从 { 对象 -> 节点 } 转换到了 { 对象 -> 虚拟节点 } 。查询物体所在 cache 时的映射关系如下图 所示。



JAVA 算法

二分查找

又叫折半查找，要求待查找的序列有序。每次取中间位置的值与待查关键字比较，如果中间位置的值比待查关键字大，则在前半部分循环这个查找的过程，如果中间位置的值比待查关键字小，则在后半部分循环这个查找的过程。直到查找到了为止，否则序列中没有待查的关键字。

```
  public static int biSearch(int []array,int a){
  int lo=0;
  int hi=array.length-1;
  int mid;
  while(lo<=hi){
  mid=(lo+hi)/2;//中间位置
  if(array[mid]==a){
  return mid+1;
            }else if(array[mid]<a){ //向右查找
            lo=mid+1;
            }else{ //向左查找
            hi=mid-1;
            }
        }
        return -1;
    }
```

冒泡排序算法

（1）比较前后相邻的二个数据，如果前面数据大于后面的数据，就将这二个数据交换。

（2）这样对数组的第 0 个数据到 N-1 个数据进行一次遍历后，最大的一个数据就“沉”到数组第

N-1 个位置。

（3）N=N-1，如果 N 不为 0 就重复前面二步，否则排序完成。

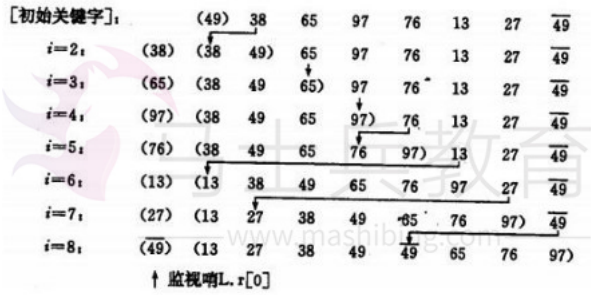
```
public static void bubbleSort1(int [] a, int n){
int i, j;
for(i=0; i<n; i++){//表示 n 次排序过程。
for(j=1; j<n-i; j++){
if(a[j-1] > a[j]){//前面的数字大于后面的数字就交换
//交换 a[j-1]和 a[j]
int temp; temp = a[j-1]; a[j-1] = a[j]; a[j]=temp;
}
}
}
}
```

插入排序算法

通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应的位置并插入。插入排序非常类似于整扑克牌。在开始摸牌时，左手是空的，牌面朝下放在桌上。接着，一次从桌上摸起一张牌，并将它插入到左手一把牌中的正确位置上。为了找到这张牌的正确位置，要将它与手中已有的牌从右到左地进行比较。无论什么时候，左手中的牌都是排好序的。

如果输入数组已经是排好序的话，插入排序出现最佳情况，其运行时间是输入规模的一个线性函数。如果输入数组是逆序排列的，将出现最坏情况。平均情况与最坏情况一样，其时间代价是(n2)。






```
public void sort(int arr[])
 {
           for(int i =1; i<arr.length;i++)
           {
               //插入的数
               int insertVal = arr[i];
               //被插入的位置(准备和前一个数比较)
               int index = i-1;
                       //如果插入的数比被插入的数小
                       while(index>=0&&insertVal<arr[index])
                       {
                           //将把 arr[index] 向后移动                                arr[index+1]=arr[index];
                           //让 index 向前移动
                           index--;
                       }
                       //把插入的数放入合适位置
               arr[index+1]=insertVal;
           }
 }
```

快速排序算法

快速排序的原理：选择一个关键值作为基准值。比基准值小的都在左边序列（一般是无序的），比基准值大的都在右边（一般是无序的）。一般选择序列的第一个元素。

一次循环：从后往前比较，用基准值和最后一个值比较，如果比基准值小的交换位置，如果没有继续比较下一个，直到找到第一个比基准值小的值才交换。找到这个值之后，又从前往后开始比较，如果有比基准值大的，交换位置，如果没有继续比较下一个，直到找到第一个比基准值大的值才交换。直到从前往后的比较索引>从后往前比较的索引，结束第一次循环，此时，对于基准值来说，左右两边就是有序的了。

```
   public void sort(int[] a,int low,int high){
   int start = low;
   int end = high;
          int key = a[low];
          while(end>start){
          //从后往前比较
          while(end>start&&a[end]>=key)
 //如果没有比关键值小的，比较下一个，直到有比关键值小的交换位置，然后又从前往后比较
               end--;
               if(a[end]<=key){
               int temp = a[end];
               a[end] = a[start];
               a[start] = temp;
              }
              //从前往后比较
              while(end>start&&a[start]<=key)
//如果没有比关键值大的，比较下一个，直到有比关键值大的交换位置
                 start++;
                 if(a[start]>=key){
                 int temp = a[start];
                 a[start] = a[end];
                 a[end] = temp;
               }
         //此时第一次循环比较结束，关键值的位置已经确定了。左边的值都比关键值小，右边的值都比关键值大，但是两边的顺序还有可能是不一样的，进行下面的递归调用
         }
         //递归
         if(start>low) sort(a,low,start-1);//左边序列。第一个索引位置到关键值索引-1          if(end<high) sort(a,end+1,high);//右边序列。从关键值索引
+1 到最后一个
      }
```

```
    }
```



上面的循环之后得到了基准值左右两边的序列了。注意：交换的是起始和结束位置的值，不是基准值

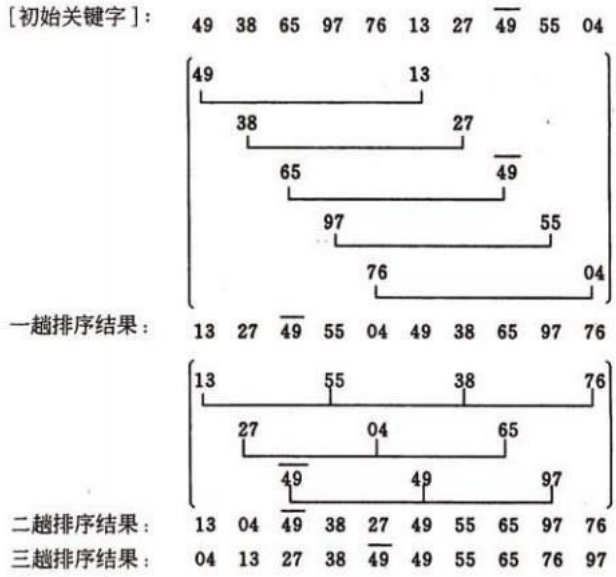
希尔排序算法

基本思想：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行依次直接插入排序。

1. 操作方法：

选择一个增量序列 t1，t2，…，tk，其中 ti>tj，tk=1；

2. 按增量序列个数 k，对序列进行 k 趟排序；
3. 每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

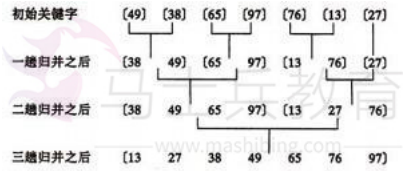



```
 private void shellSort(int[] a) {
        int dk = a.length/2;
        while( dk >= 1  ){
                  ShellInsertSort(a, dk);
                  dk = dk/2;
        }
   }
    private void ShellInsertSort(int[] a, int dk) {
//类似插入排序，只是插入排序增量是 1，这里增量是 dk,把 1 换成 dk 就可以了
       for(int i=dk;i<a.length;i++){
           if(a[i]<a[i-dk]){
               int j;
               int x=a[i];//x 为待插入元素
               a[i]=a[i-dk];
               for(j=i-dk;  j>=0 && x<a[j];j=j-dk){
//通过循环，逐个后移一位找到要插入的位置。
                   a[j+dk]=a[j];
```

```
                }
                a[j+dk]=x;//插入
            }
        }
    }
```

归并排序算法

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。




```
   *
*   @param data
*   数组对象
*   @param left
*   左数组的第一个元素的索引
*   @param center
*   左数组的最后一个元素的索引，center+1 是右数组第一个元素的索引
*   @param right
*   右数组最后一个元素的索引
     */
    public static void merge(int[] data, int left, int center, int right) {
        // 临时数组           int[] tmpArr = new int[data.length];
        // 右数组第一个元素索引
        int mid = center + 1;
        // third 记录临时数组的索引
        int third = left;
        // 缓存左数组第一个元素的索引
        int tmp = left;           while (left <= center && mid <= right) {
            // 从两个数组中取出最小的放入临时数组
            if (data[left] <= data[mid]) {                  tmpArr[third++] = data[left++];
            } else {
                tmpArr[third++] = data[mid++];
            }
        }
        // 剩余部分依次放入临时数组（实际上两个 while 只会执行其中一个）
        while (mid <= right) {               tmpArr[third++] = data[mid++];
public class MergeSortTest {
public static void main(String[] args) {
int[] data = new int[] { 5, 3, 6, 2, 1, 9, 4, 8, 7 };           print(data);
mergeSort(data);
        System.out.println("排序后的数组：");
        print(data);
    }
     public static void mergeSort(int[] data) {
     sort(data, 0, data.length - 1);
    }
      public static void sort(int[] data, int left, int right) {           if (left >= right)
      return;
        // 找出中间索引
        int center = (left + right) / 2;
        // 对左边数组进行递归
        sort(data, left, center);
        // 对右边数组进行递归
        sort(data, center + 1, right);
        // 合并
        merge(data, left, center, right);
        print(data);
    }
      /**
     * 将两个数组进行归并，归并前面 2 个数组已有序，归并后依然有序
   *
*   @param data
*   数组对象
*   @param left
*   左数组的第一个元素的索引
*   @param center
*   左数组的最后一个元素的索引，center+1 是右数组第一个元素的索引
*   @param right
*   右数组最后一个元素的索引
     */
    public static void merge(int[] data, int left, int center, int right) {
        // 临时数组           int[] tmpArr = new int[data.length];
        // 右数组第一个元素索引
        int mid = center + 1;
        // third 记录临时数组的索引
        int third = left;
```

```
        // 缓存左数组第一个元素的索引
        int tmp = left;
        while (left <= center && mid <= right) {
            // 从两个数组中取出最小的放入临时数组
            if (data[left] <= data[mid]) {                    tmpArr[third++] = data[left++];
            } else {
                tmpArr[third++] = data[mid++];
            }
        }
        // 剩余部分依次放入临时数组（实际上两个 while 只会执行其中一个）
        while (mid <= right) {
        tmpArr[third++] = data[mid++];
               }
        while (left <= center) {
        tmpArr[third++] = data[left++];
        }
        // 将临时数组中的内容拷贝回原数组中
        // （原 left-right 范围的内容被复制回原数组）
        while (tmp <= right) {
        data[tmp] = tmpArr[tmp++];
        }
    }
    public static void print(int[] data) {
    for (int i = 0; i < data.length; i++) {
            System.out.print(data[i] + "\t");
        }
        System.out.println();
    }
  }
```

桶排序算法

桶排序的基本思想是： 把数组 arr 划分为 n 个大小相同子区间（桶），每个子区间各自排序，最后合并 。计数排序是桶排序的一种特殊情况，可以把计数排序当成每个桶里只有一个元素的情况。

1.找出待排序数组中的最大值 max、最小值 min

2.我们使用 动态数组 ArrayList 作为桶，桶里放的元素也用 ArrayList 存储。桶的数量为(maxmin)/arr.length+1

3.遍历数组 arr，计算每个元素 arr[i] 放的桶

4.每个桶各自排序

```
  public static void bucketSort(int[] arr){

  int max = Integer.MIN_VALUE;
  int min = Integer.MAX_VALUE;
  for(int i = 0; i < arr.length; i++){
 max = Math.max(max, arr[i]);
 min = Math.min(min, arr[i]);
  }
   //创建桶
  int bucketNum = (max - min) / arr.length + 1;
  ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketNum);   for(int i = 0; i < bucketNum; i++){
  bucketArr.add(new ArrayList<Integer>());
  }
     //将每个元素放入桶
  for(int i = 0; i < arr.length; i++){
  int num = (arr[i] - min) / (arr.length);     bucketArr.get(num).add(arr[i]);
  }
     //对每个桶进行排序
  for(int i = 0; i < bucketArr.size(); i++){
    Collections.sort(bucketArr.get(i));
  }
 }
```

基数排序算法

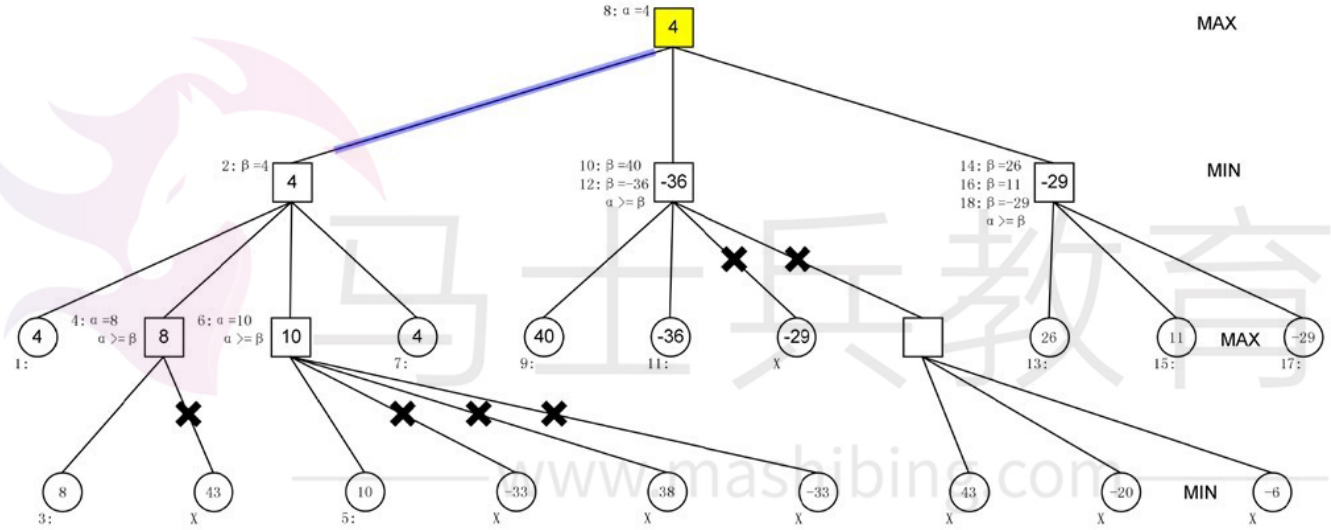
将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后,数列就变成一个有序序列。

```
 public class radixSort {
 inta[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,101,56,17,18,23,34,15,35,2
5,53,51};
public radixSort(){
sort(a);
       for(inti=0;i<a.length;i++){                 System.out.println(a[i]);
       }
    }
    public  void sort(int[] array){
       //首先确定排序的趟数;
       int max=array[0];
       for(inti=1;i<array.length;i++){
       if(array[i]>max){
             max=array[i];
```

```
            }
        }
        int time=0;             //判断位数;             while(max>0){                max/=10;                time++;
        }
         //建立 10 个队列;
        List<ArrayList> queue=newArrayList<ArrayList>();            for(int i=0;i<10;i++){
               ArrayList<Integer>queue1=new ArrayList<Integer>();              queue.add(queue1);
        }
           //进行 time 次分配和收集;
        for(int i=0;i<time;i++){                 //分配数组元素;               for(intj=0;j<array.length;j++){
                //得到数字的第 time+1 位数;
                  int x=array[j]%(int)Math.pow(10,i+1)/(int)Math.pow(10, i);                    ArrayList<Integer>queue2=queue.get(x);
              queue2.add(array[j]);                    queue.set(x, queue2);
           }
           int count=0;//元素计数器;
           //收集队列元素;               for(int k=0;k<10;k++){                 while(queue.get(k).size()>0){
                    ArrayList<Integer>queue3=queue.get(k);                     array[count]=queue3.get(0);
queue3.remove(0);                      count++;
                }
           }
        }
    }
}
```

剪枝算法

在搜索算法中优化中，剪枝，就是通过某种判断，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”，故称剪枝。应用剪枝优化的核心问题是设计剪枝判断方法，即确定哪些枝条应当舍弃，哪些枝条应当保留的方法。



回溯算法

回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

最短路径算法

从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径叫做最短路径。解决最短路的问题有以下算法，Dijkstra 算法，Bellman-Ford 算法，Floyd 算法和 SPFA 算法等。

最大子数组算法

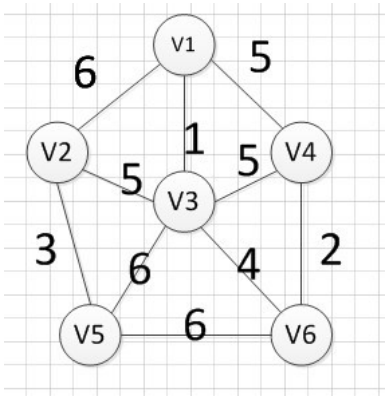
最长公共子序算法

最小生成树算法

现在假设有一个很实际的问题：我们要在 n 个城市中建立一个通信网络，则连通这 n 个城市需要布置 n-1 一条通信线路，这个时候我们需要考虑如何在成本最低的情况下建立这个通信网?

于是我们就可以引入连通图来解决我们遇到的问题，n 个城市就是图上的 n 个顶点，然后，边表示两个城市的通信线路，每条边上的权重就是我们搭建这条线路所需要的成本，所以现在我们有n个顶点的连通网可以建立不同的生成树，每一颗生成树都可以作为一个通信网，当我们构造这个连通网所花的成本最小时，搭建该连通网的生成树，就称为最小生成树。构造最小生成树有很多算法，但是他们都是利用了最小生成树的同一种性质：MST 性质（假设

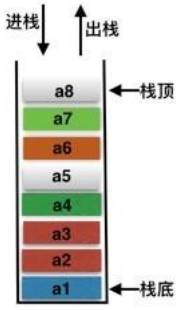
N=(V,{E})是一个连通网，U 是顶点集 V 的一个非空子集，如果（u，v）是一条具有最小权值的边，其中 u 属于U，v 属于V-U，则必定存在一颗包含边（u，v）的最小生成树），下面就介绍两种使用 MST 性质生成最小生成树的算法：普里姆算法和克鲁斯卡尔算法。



数据结构

栈（stack）

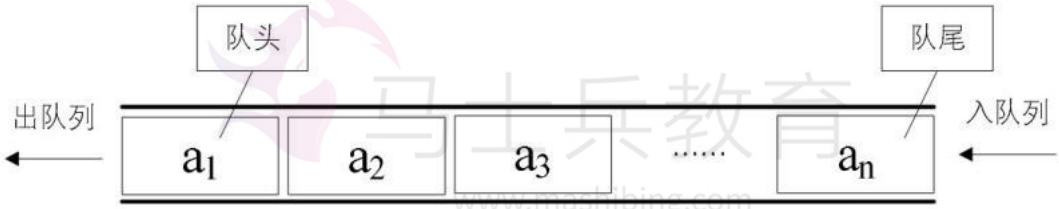
栈（stack）是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，叫做栈顶（top）。它是后进先出（LIFO）的。对栈的基本操作只有 push（进栈）和 pop（出栈）两种，前者相当于插入，后者相当于删除最后的元素。



图：栈的存储结构示意图

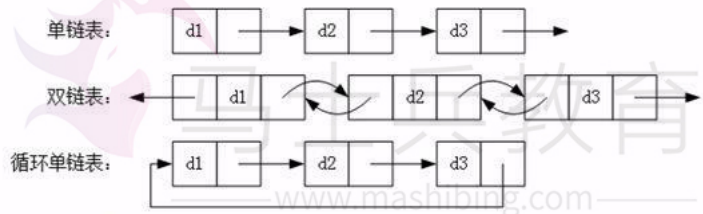
队列（queue）

队列是一种特殊的线性表,特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。



链表（Link）

链表是一种数据结构，和数组同级。比如，Java 中我们使用的 ArrayList，其实现原理是数组。而LinkedList 的实现原理就是链表了。链表在进行循环遍历时效率不高，但是插入和删除时优势明显。



散列表（Hash Table）

散列表（Hash table，也叫哈希表）是一种查找算法，与链表、树等算法不同的是，散列表算法在查找时不需要进行一系列和关键字（关键字是数据元素中某个数据项的值，用以标识一个数据元素）的比较操作。

散列表算法希望能尽量做到不经过任何比较，通过一次存取就能得到所查找的数据元素，因而必须要在数据元素的存储位置和它的关键字（可用key表示）之间建立一个确定的对应关系，使每个关键字和散列表中一个唯一的存储位置相对应。因此在查找时，只要根据这个对应关系找到给定关键字在散列表中的位置即可。这种对应关系被称为散列函数(可用 h(key)表示)。

用的构造散列函数的方法有：

（1）直接定址法： 取关键字或关键字的某个线性函数值为散列地址。

即：h(key) = key 或 h(key) = a * key + b，其中 a 和 b 为常数。

（2）数字分析法

（3）平方取值法： 取关键字平方后的中间几位为散列地址。

（4）折叠法：将关键字分割成位数相同的几部分，然后取这几部分的叠加和作为散列地址。

（5）除留余数法：取关键字被某个不大于散列表表长 m 的数 p 除后所得的余数为散列地址，即：h(key) = key MOD p p ≤ m

（6）随机数法：选择一个随机函数，取关键字的随机函数值为它的散列地址，即：h(key) = random(key)

排序二叉树

首先如果普通二叉树每个节点满足：左子树所有节点值小于它的根节点值，且右子树所有节点值大于它的根节点值，则这样的二叉树就是排序二叉树。

插入操作

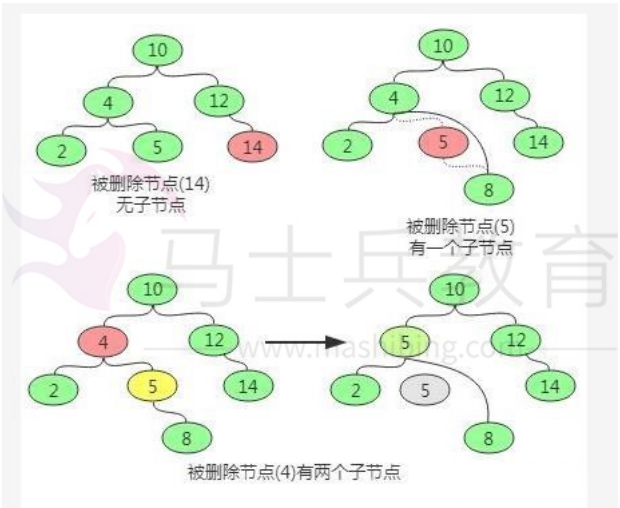
首先要从根节点开始往下找到自己要插入的位置（即新节点的父节点）；具体流程是：新节点与当前节点比较，如果相同则表示已经存在且不能再重复插入；如果小于当前节点，则到左子树中寻找，如果左子树为空则当前节点为要找的父节点，新节点插入到当前节点的左子树即可；如果大于当前节点，则到右子树中寻找，如果右子树为空则当前节点为要找的父节点，新节点插入到当前节点的右子树即可。



删除操作

删除操作主要分为三种情况，即要删除的节点无子节点，要删除的节点只有一个子节点，要删除的节点有两个子节点。

1. 对于要删除的节点无子节点可以直接删除，即让其父节点将该子节点置空即可。
2. 对于要删除的节点只有一个子节点，则替换要删除的节点为其子节点。
3. 对于要删除的节点有两个子节点，则首先找该节点的替换节点（即右子树中最小的节点），接着替换要删除的节点为替换节点，然后删除替换节点。



查询操作

查找操作的主要流程为：先和根节点比较，如果相同就返回，如果小于根节点则到左子树中递归查找，如果大于根节点则到右子树中递归查找。因此在排序二叉树中可以很容易获取最大（最右最深子节点）和最小（最左最深子节点）值。

红黑树

R-B Tree，全称是 Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树的特性

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

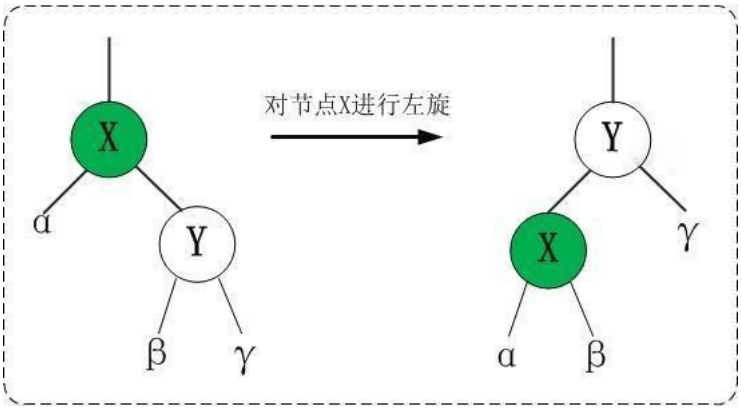
（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

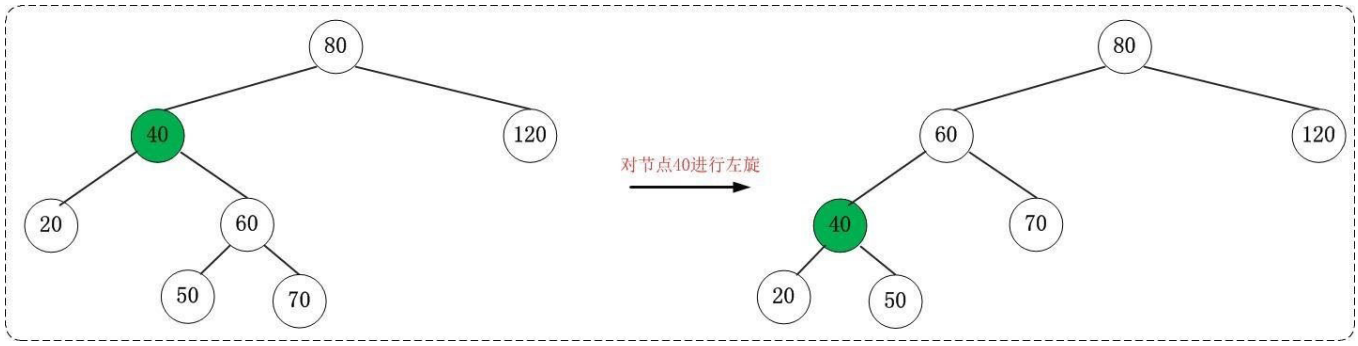
（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

左旋

对 x 进行左旋，意味着，将“x 的右孩子”设为“x 的父亲节点”；即，将 x 变成了一个左节点(x 成了为 z 的左孩子)。 因此，左旋中的“左”，意味着“被旋转的节点将变成一个左节点”。

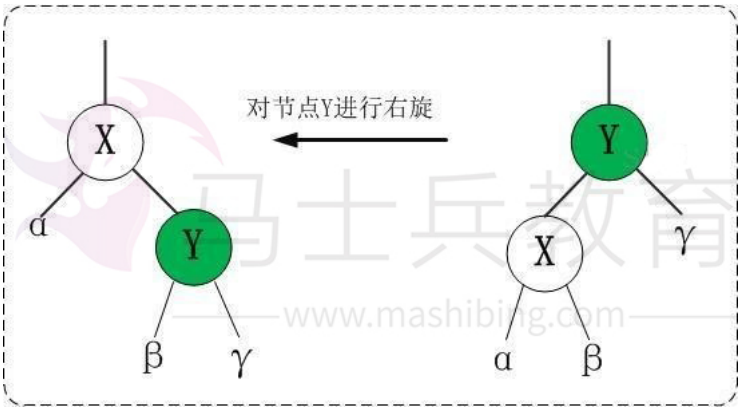



```
LEFT-ROTATE(T, x)
 y ← right[x]
 // 前提：这里假设 x 的右孩子为 y。下面开始正式操作   right[x] ← left[y]
 // 将 “y 的左孩子” 设为 “x 的右孩子”，即 将β设为 x 的右孩子  p[left[y]] ← x
 // 将 “x” 设为 “y 的左孩子的父亲”，即 将β的父亲设为 x  p[y] ← p[x]
 // 将 “x 的父亲” 设为 “y 的父亲”
 if p[x] = nil[T]
 then root[T] ← y
 // 情况 1：如果 “x 的父亲” 是空节点，则将 y 设为根节点
 else if x = left[p[x]]
           then left[p[x]] ← y
           // 情况 2：如果 x 是它父节点的左孩子，则将 y 设为“x 的父节点的左孩子”
           else right[p[x]] ← y
           // 情况 3：(x 是它父节点的右孩子) 将 y 设为“x 的父节点的右孩子”
 left[y] ← x
 // 将 “x” 设为 “y 的左孩子”  p[x] ← y
 // 将 “x 的父节点” 设为 “y”
```



右旋

对 x 进行右旋，意味着，将“x 的左孩子”设为“x 的父亲节点”；即，将 x 变成了一个右节点(x 成了为 y 的右孩子)！ 因此，右旋中的“右”，意味着“被旋转的节点将变成一个右节点”。



马士兵教育
www.mashibing.com

```
  RIGHT-ROTATE(T, y)
x ← left[y]
// 前提：这里假设 y 的左孩子为 x。下面开始正式操作   left[y] ← right[x]
```

```
// 将 “x 的右孩子” 设为 “y 的左孩子”，即 将β设为 y 的左孩子  p[right[x]] ← y
// 将 “y” 设为 “x 的右孩子的父亲”，即 将β的父亲设为 y  p[x] ← p[y]
// 将 “y 的父亲” 设为 “x 的父亲”
if p[y] = nil[T]
then root[T] ← x
// 情况 1：如果 “y 的父亲” 是空节点，则将 x 设为根节点
else if y = right[p[y]]
          then right[p[y]] ← x
          // 情况 2：如果 y 是它父节点的右孩子，则将 x 设为“y 的父节点的左孩子”
          else left[p[y]] ← x
          // 情况 3：(y 是它父节点的左孩子) 将 x 设为“y 的父节点的左孩子”
right[x] ← y
// 将 “y” 设为 “x 的右孩子”  p[y] ← x
// 将 “y 的父节点” 设为 “x”
```

添加

第一步: 将红黑树当作一颗二叉查找树，将节点插入。

第二步：将插入的节点着色为"红色"。

根据被插入节点的父节点的情况，可以将"当节点 z 被着色为红色节点，并插入二叉树"划分为三种情况来处理。

① 情况说明：被插入的节点是根节点。

处理方法：直接把此节点涂为黑色。

② 情况说明：被插入的节点的父节点是黑色。

处理方法：什么也不需要做。节点被插入后，仍然是红黑树。

③ 情况说明：被插入的节点的父节点是红色。这种情况下，被插入节点是一定存在非空祖父节点的；进一步的讲，被插入节点也一定存在叔叔节点(即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点)。理解这点之后，我们依据"叔叔节点的情况"，将这种情况进一步划分为 3 种情况(Case)。

	现象说明	处理策略
Case 1	当前节点的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。	(01) 将“父节点”设为黑色。 (02) 将“叔叔节点”设为黑色。 (03) 将“祖父节点”设为“红色”。 (04) 将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作。
Case 2	当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子	(01) 将“父节点”作为“新的当前节点”。 (02) 以“新的当前节点”为支点进行左旋。
Case 3	当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子	(01) 将“父节点”设为“黑色”。 (02) 将“祖父节点”设为“红色”。 (03) 以“祖父节点”为支点进行右旋。

第三步: 通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。

删除

第一步：将红黑树当作一颗二叉查找树，将节点删除。

这和"删除常规二叉查找树中删除节点的方法是一样的"。分 3 种情况：

① 被删除节点没有儿子，即为叶节点。那么，直接将该节点删除就 OK 了。

② 被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的位置。

③ 被删除节点有两个儿子。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制给

“该节点的内容”；之后，删除“它的后继节点”。

第二步：通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

因为"第一步"中删除节点之后，可能会违背红黑树的特性。所以需要通过"旋转和重新着色"来修正该树，使之重新成为一棵红黑树。

选择重着色 3 种情况。

① 情况说明：x 是“红+黑”节点。

处理方法：直接把 x 设为黑色，结束。此时红黑树性质全部恢复。

② 情况说明：x 是“黑+黑”节点，且 x 是根。

处理方法：什么都不做，结束。此时红黑树性质全部恢复。

③ 情况说明：x 是“黑+黑”节点，且 x 不是根。

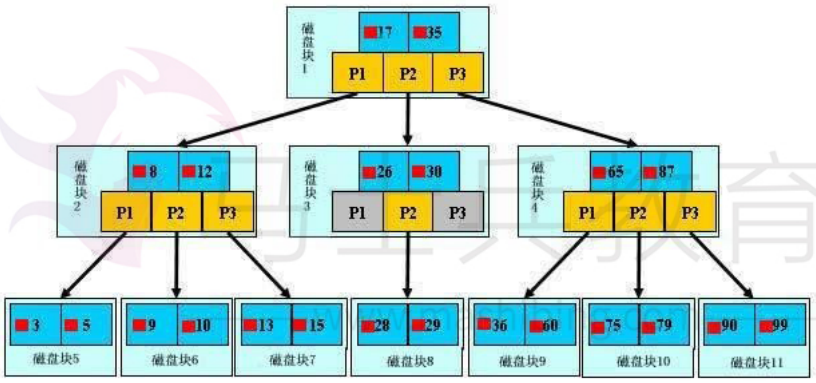
处理方法：这种情况又可以划分为 4 种子情况。这 4 种子情况如下表所示：

	现象说明	处理策略
Case 1	x是"黑+黑"节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)。	(01) 将x的兄弟节点设为"黑色"。 (02) 将x的父节点设为"红色"。 (03) 对x的父节点进行左旋。 (04) 左旋后，重新设置x的兄弟节点。
Case 2	x是"黑+黑"节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。	(01) 将x的兄弟节点设为"红色"。 (02) 设置"x的父节点"为"新的x节点"。
Case 3	x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。	(01) 将x兄弟节点的左孩子设为"黑色"。 (02) 将x兄弟节点设为"红色"。 (03) 对x的兄弟节点进行右旋。 (04) 右旋后，重新设置x的兄弟节点。
Case 4	x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节点的左孩子任意颜色。	(01) 将x父节点颜色 赋值给 x的兄弟节点。 (02) 将x父节点设为"黑色"。 (03) 将x兄弟节点的右子节设为"黑色"。 (04) 对x的父节点进行左旋。 (05) 设置"x"为"根节点"。

B-TREE

B-tree 又叫平衡多路查找树。一棵 m 阶的 B-tree (m 叉树)的特性如下（其中 ceil(x)是一个取上限的函数）：

1. 树中每个结点至多有 m 个孩子；
2. 除根结点和叶子结点外，其它每个结点至少有有 ceil(m / 2)个孩子；
3. 若根结点不是叶子结点，则至少有2个孩子（特殊情况：没有孩子的根结点，即根结点为叶子结点，整棵树只有一个根节点）；
4. 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息(可以看做是外部结点或查询失败的结点，实际上这些结点不存在，指向这些结点的指针都为 null)；
5. 每个非终端结点中包含有 n 个关键字信息：（n，P0，K1，P1，K2，P2，……，Kn，Pn)。其中：
   a) Ki (i=1...n)为关键字，且关键字按顺序排序 K(i-1)< Ki。
   b) Pi 为指向子树根的接点，且指针 P(i-1)指向子树种所有结点的关键字均小于 Ki，但都大于 K(i-1)。
   c) 关键字的个数 n 必须满足： ceil(m / 2)-1 <= n <= m-1。



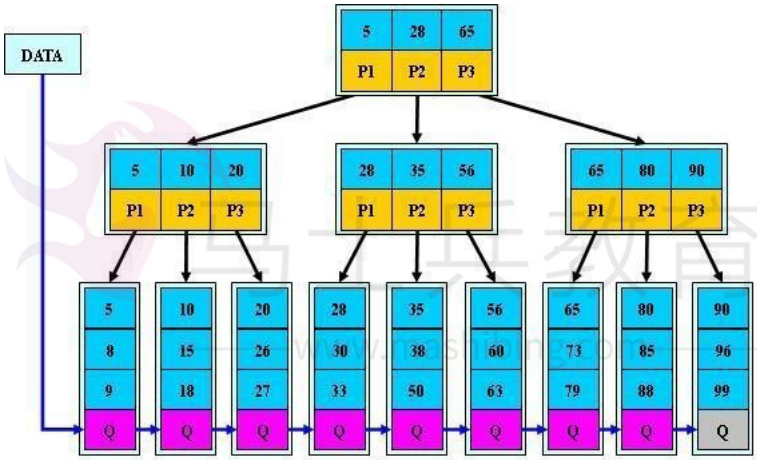
一棵 m 阶的 B+tree 和 m 阶的 B-tree 的差异在于：

1.有 n 棵子树的结点中含有 n 个关键字； (B-tree 是 n 棵子树有 n-1 个关键字)

2.所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。 (B-tree 的叶子节点并没有包括全部需要查找的信息)

3.所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。

(B-tree 的非终节点也包含需要查找的有效信息)



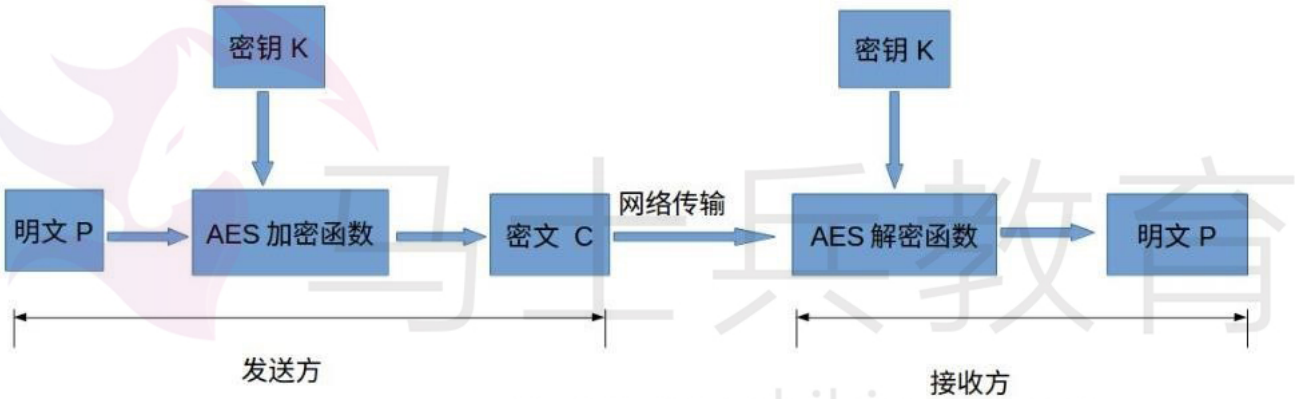
位图

位图的原理就是用一个 bit 来标识一个数字是否存在，采用一个 bit 来存储一个数据，所以这样可以大大的节省空间。 bitmap 是很常用的数据结构，比如用于 Bloom Filter 中；用于无重复整数的排序等等。bitmap 通常基于数组来实现，数组中每个元素可以看成是一系列二进制数，所有元素组成更大的二进制集合。

加密算法

AES

高级加密标准(AES,Advanced Encryption Standard)为最常见的对称加密算法(微信小程序加密传输就是用这个加密算法的)。对称加密算法也就是加密和解密用相同的密钥，具体的加密流程如下图：

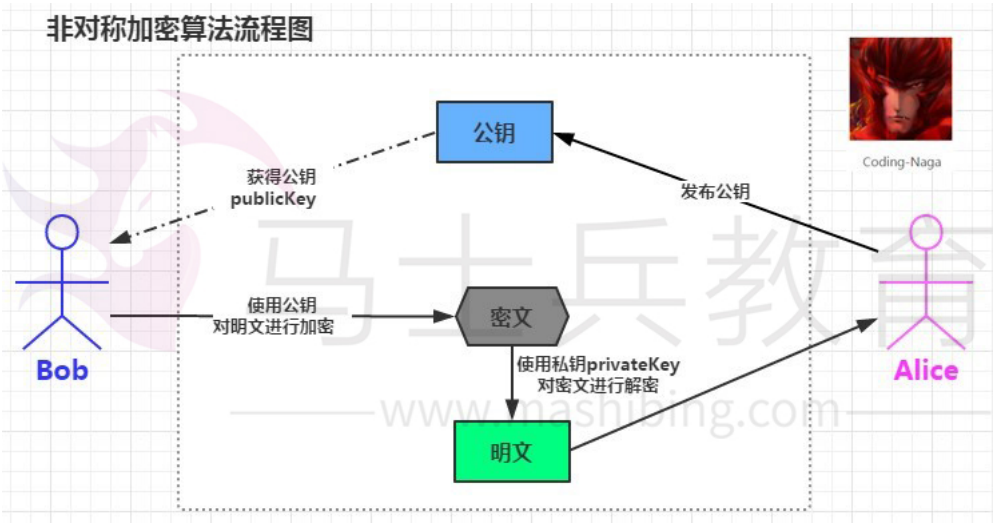


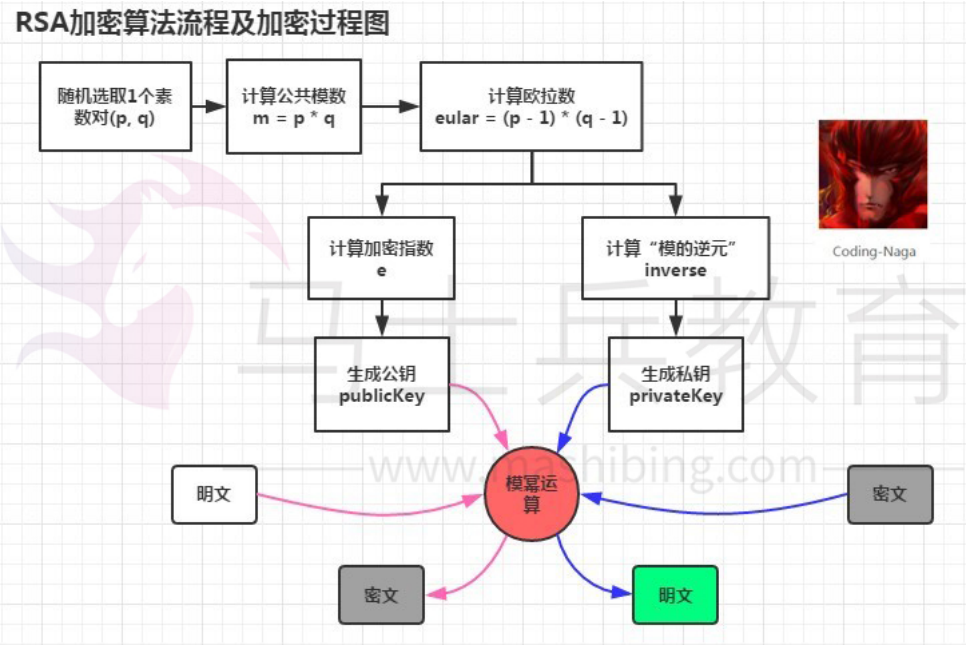
www.mashibing.com
http://blog.csdn.net/qq_28205153

RSA

RSA 加密算法是一种典型的非对称加密算法，它基于大数的因式分解数学难题，它也是应用最广泛的非对称加密算法。

非对称加密是通过两个密钥（公钥-私钥）来实现对数据的加密和解密的。公钥用于加密，私钥用于解密。





CRC

循环冗余校验(Cyclic Redundancy Check, CRC)是一种根据网络数据包或电脑文件等数据产生简短固定位数校验码的一种散列函数，主要用来检测或校验数据传输或者保存后可能出现的错误。

它是利用除法及余数的原理来作错误侦测的。

MD5

MD5 常常作为文件的签名出现，我们在下载文件的时候，常常会看到文件页面上附带一个扩展名为.MD5 的文本或者一行字符，这行字符就是就是把整个文件当作原数据通过 MD5 计算后的值，我们下载文件后，可以用检查文件 MD5 信息的软件对下载到的文件在进行一次计算。两次结果对比就可以确保下载到文件的准确性。 另一种常见用途就是网站敏感信息加密，比如用户名密码，支付签名等等。随着 https 技术的普及，现在的网站广泛采用前台明文传输到后台，MD5 加密

（使用偏移量）的方式保护敏感数据保护站点和数据安全。

项目方面（晁老师-主要讲经验）

日志

Slf4j

slf4j 的全称是 Simple Loging Facade For Java，即它仅仅是一个为 Java 程序提供日志输出的统一接口，并不是一个具体的日志实现方案，就比如 JDBC 一样，只是一种规则而已。所以单独的 slf4j 是不能工作的，必须搭配其他具体的日志实现方案，比如 apache 的 org.apache.log4j.Logger，jdk 自带的 java.util.logging.Logger 等。

Log4j

Log4j 是 Apache 的一个开源项目，通过使用 Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI 组件，甚至是套接口服务器、NT 的事件记录器、UNIX Syslog 守护进程等；我们也可以控

制每一条日志的输出格式；通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。

Log4j由三个重要的组成构成：日志记录器(Loggers)，输出端(Appenders)和日志格式化器(Layout)。

1.Logger：控制要启用或禁用哪些日志记录语句，并对日志信息进行级别限制

2.Appenders : 指定了日志将打印到控制台还是文件中

3.Layout : 控制日志信息的显示格式

Log4j 中将要输出的 Log 信息定义了 5 种级别，依次为 DEBUG、INFO、WARN、ERROR 和 FATAL，当输出时，只有级别高过配置中规定的 级别的信息才能真正的输出，这样就很方便的来配置不同情况下要输出的内容，而不需要更改代码。

LogBack

简单地说，Logback 是一个 Java 领域的日志框架。它被认为是 Log4j 的继承人。

Logback 主要由三个模块组成：logback-core，logback-classic。logback-access

logback-core 是其它模块的基础设施，其它模块基于它构建，显然，logback-core 提供了一些关键的通用机制。

logback-classic 的地位和作用等同于 Log4j，它也被认为是 Log4j 的一个改进版，并且它实现了简单日志门面 SLF4J；

logback-access 主要作为一个与 Servlet 容器交互的模块，比如说 tomcat 或者 jetty，提供一些与HTTP 访问相关的功能。

Logback 优点

§ 同样的代码路径，Logback 执行更快

§ 更充分的测试

§ 原生实现了 SLF4J API（Log4J 还需要有一个中间转换层）

§ 内容更丰富的文档

§ 支持 XML 或者 Groovy 方式配置

§ 配置文件自动热加载

§ 从 IO 错误中优雅恢复

§ 自动删除日志归档

§ 自动压缩日志成为归档文件

§ 支持 Prudent 模式，使多个 JVM 进程能记录同一个日志文件

§ 支持配置文件中加入条件判断来适应不同的环境

§ 更强大的过滤器

§ 支持 SiftingAppender（可筛选 Appender）

§ 异常栈信息带有包信息

设计模式（马老师）

单例模式

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这 个实例。

```
（1）懒汉式 public class Singleton {
    /* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载  */       private static Singleton instance = null;
    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }
    /* 1:懒汉式，静态工程方法，创建实例 */       public static Singleton getInstance() {           if (instance == null) {               instance
= new Singleton();
        }
        return instance;
        }
     }
    （2）饿汉式
    public class Singleton {
        /* 持有私有静态实例，防止被引用 */
        private static Singleton instance = new Singleton();
        /* 私有构造方法，防止被实例化 */
        private Singleton() {
        }
    /* 1:懒汉式，静态工程方法，创建实例 */       public static Singleton getInstance() {
        return instance;
    }
 }
```

使用场景：

- 要求生成唯一序列号的环境；
- 在整个项目中需要一个共享访问点或共享数据，例如一个Web页面上的计数 器，可以不用把每次刷新都记录到数据库中，使用单例模式保持计数器的值，并 确保是线程安全的；
- 创建一个对象需要消耗的资源过多，如要访问IO和数据库等资源；
- 需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式 （当然，也可以直接声明为static的方式）。

工厂模式

定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂 方法使一个类的实例化延迟到其子类。

```
接口

public interface Fruit {
public void print();   }
```

2个实现类

```
public class Apple implements Fruit{
    @Override
    public void print() {
       System.out.println("我是一个苹果");
    }
}
public class Orange implements Fruit{
    @Override
    public void print() {
       System.out.println("我是一个橘子");
    }
}
```

工厂类

```
public class FruitFactory {
    public Fruit produce(String type){
        if(type.equals("apple")){
            return new Apple();
        }else if(type.equals("orange")){
            return new Orange();
        }else{
            System.out.println("请输入正确的类型!");
        return null;
        }
    }
}
```

使用场景：jdbc连接数据库，硬件访问，降低对象的产生和销毁

抽象工厂模式

为创建一组相关或相互依赖的对象提供一个接口，而且无须指 定它们的具体类。

相对于工厂模式，我们可以新增产品类（只需要实现产品接口），只需要同时新 增一个工厂类，客户端就可以轻松调用新产品的代码。

```
interface food{}
class A implements food{}
class B implements food{}
interface produce{ food get();}
class FactoryForA implements produce{
    @Override
    public food get() {
        return new A();
    }
}
class FactoryForB implements produce{
    @Override
    public food get() {
    return new B();
    }
}
public class AbstractFactory {
    public void ClientCode(String name){
        food x= new FactoryForA().get();
        x = new FactoryForB().get();
    }
}
```

使用场景：一个对象族（或是一组没有任何关系的对象）都有相同的约束。 涉及不同操作系统的时候，都可以考虑使用抽象工厂模式

建造者模式

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可 以创建不同的表示。

```
public class Build {
    static class Student{
    String name = null ;
        int number = -1 ;
        String sex = null ;
    public Student(Builder builder)         {
    this.name=builder.name;
    this.number=builder.number;
    this.sex=builder.sex;
    }
        static class Builder{

        String name = null ;
        int number = -1 ;
        String sex = null ;
        public Builder setName(String name){
        this.name=name;
        return this;
        }
```

```
        public Builder setNumber(int number){
        this.number=number;
        return this;
        }
        public Builder setSex(String sex){
        this.sex=sex;
        return this;
        }
        public Student build(){
        return new Student(this);
        }
 }
}
public static void main(String[] args) {
Student A=new Student.Builder().setName("张 三").setNumber(1).build();     Student B=new Student.Builder().setSex("男").setName("李
四").build();
    System.out.println(A.name+" "+A.number+" "+A.sex);                      System.out.println(B.name+" "+B.number+" "+B.sex);
    }
}
```

使用场景：

1. 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模 式。
2. 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同 时，则可以使用该模式。
3. 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候 使用建造者模式非常合适。

原型模式

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的 对象。

```
public class Prototype implements Cloneable{
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
    this.name = name;
    }
    @Override
    protected Object clone()   {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }finally {
            return null;
        }
    }
    public static void main ( String[] args){
        Prototype pro = new Prototype();
        Prototype pro1 = (Prototype)pro.clone();
    }
}
```

原型模式实际上就是实现Cloneable接口，重写clone（）方法。

使用原型模式的优点：

1.性能优良

原型模式是在内存二进制流的拷贝，要比直接new一个对象性能好很多，特别是 要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。

2.逃避构造函数的约束

这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的（参见 13.4节）。

使用场景：

资源优化场景 类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。

性能和安全要求的场景 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模 式。

一个对象多个修改者的场景 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可 以考虑使用原型模式拷贝多个对象供调用者使用。

浅拷贝和深拷贝：

浅拷贝：Object类提供的方法clone只是拷贝本对象，其对象内部的数组、引用 对象等都不拷贝，还是指向原生对象的内部元素地址，这种拷贝就叫做浅拷贝， 其他的原始类型比如int、long、char、string（当做是原始类型）等都会被拷 贝。

注意： 使用原型模式时，引用的成员变量必须满足两个条件才不会被拷贝：一 是类的成员变量，而不是方法内变量；二是必须是一个可变的引用对象，而不是 一个原始类型或不可变对象。

深拷贝：对私有的类变量进行独立的拷贝

如：this.arrayList = (ArrayList)this.arrayList.clone();

适配器模式

将一个类的接口变换成客户端所期待的另一种接口，从而使原本 因接口不匹配而无法在一起工作的两个类能够在一起工作。

主要可分为3种：

1. 类适配：创建新类，继承源类，并实现新接口，例如 class adapter extends oldClass implements newFunc{}
2. 对象适配：创建新类持源类的实例，并实现新接口，例如 class adapter implements newFunc { private oldClass oldInstance ;}
3. 接口适配：创建新的抽象类实现旧接口方法。例如 abstract class adapter implements oldClassFunc { void newFunc();}

使用场景：

你有动机修改一个已经投产中的接口时，适配器模式可能是适合你的模式。比 如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接 口，怎么办？使用适配器模式，这也是我们例子中提到的。

装饰器模式

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰 器模式相比生成子类更为灵活 。

```
interface Source{ void method();}
public class Decorator implements Source{
    private Source source ;
    public void decotate1(){
        System.out.println("decorate");
    }
    @Override
    public void method() {
        decotate1();
        source.method();
    }
}
```

使用场景：

1. 需要扩展一个类的功能，或给一个类增加附加功能。
2. 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
3. 需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。

代理模式

为其他对象提供一种代理以控制对这个对象的访问。

```
interface Source{ void method();}
class OldClass implements Source{
    @Override
    public void method() {
    }
}
class Proxy implements Source{
    private Source source = new OldClass();
    void doSomething(){}
    @Override
    public void method() {
        new Class1().Func1();
        source.method();
        new Class2().Func2();
        doSomething();
    }
}
```

中介者模式

用一个中介对象封装一系列的对象交互，中介者使各对象不需要 显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

```
public abstract class Mediator {
    //定义同事类
    protected ConcreteColleague1 c1;
    protected ConcreteColleague2 c2;
    //通过getter/setter方法把同事类注入进来
    public ConcreteColleague1 getC1() {
        return c1;
    }
  public void setC1(ConcreteColleague1 c1) {
     this.c1 = c1;
  }
  public ConcreteColleague2 getC2() {
     return c2; }
  public void setC2(ConcreteColleague2 c2) {
      this.c2 = c2;
  }
  //中介者模式的业务逻辑
  public abstract void doSomething1();
  public abstract void doSomething2();
}
```

使用场景： 中介者模式适用于多个对象之间紧密耦合的情况，紧密耦合的标准是：在类图中 出现了蜘蛛网状结构，即每个类都与其他的类有直接的联系。

命令模式

将一个请求封装成一个对象，从而让你使用不同的请求把客户端 参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

Receiver接受者角色：该角色就是干活的角色，命令传递到这里是应该被执行的

Command命令角色：需要执行的所有命令都在这里声明

Invoker调用者角色：接收到命令，并执行命令

```
//通用Receiver类
public abstract class Receiver {
    public abstract void doSomething();
}
//具体Receiver类
public class ConcreteReciver1 extends Receiver{
    //每个接收者都必须处理一定的业务逻辑
    public void doSomething(){ }
}
public class ConcreteReciver2 extends Receiver{
    //每个接收者都必须处理一定的业务逻辑
    public void doSomething(){ }
}
//抽象Command类 public abstract class Command {
    public abstract void execute();
}
//具体的Command类
public class ConcreteCommand1 extends Command {
    //对哪个Receiver类进行命令处理
    private Receiver receiver;
    //构造函数传递接收者
    public ConcreteCommand1(Receiver _receiver){
        this.receiver = _receiver;
    }
    //必须实现一个命令
    public void execute() {
    //业务处理
        this.receiver.doSomething();
    }
}
public class ConcreteCommand2 extends Command {
    //哪个Receiver类进行命令处理
    private Receiver receiver;
    //构造函数传递接收者
    public ConcreteCommand2(Receiver _receiver){
        this.receiver = _receiver;
}
//必须实现一个命令
 public void execute() {
     //业务处理
     this.receiver.doSomething();
     }
 }
 //调用者Invoker类 public class Invoker {
 private Command command;
    public void setCommand(Command _command){
    this.command = _command;
 }
public void action() {
    this.command.execute();
    }
}
    //场景类
public class Client {
    public static void main(String[] args){
        Invoker invoker = new Invoker();
        Receiver receiver = new ConcreteReceiver1();
        Command command = new ConcreteCommand1(receiver);                    invoker.setCommand(command);
        invoker.action();
     }
}
```

使用场景： 认为是命令的地方就可以采用命令模式，例如，在GUI开发中，一个按钮的点击 是一个命令，可以采用命令模式；模拟DOS命令的时候，当然也要采用命令模式；触发 - 反馈机制的处理等。

责任链模式

使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

```
public abstract class Handler {
     private Handler nextHandler;
     //每个处理者都必须对请求做出处理
```

```
    public final Response handleMessage(Request request){
            Response response = null;
            //判断是否是自己的处理级别
            if(this.getHandlerLevel().equals(request.getRequestLevel())){
                   response = this.echo(request);
            }else{  //不属于自己的处理级别
                   //判断是否有下一个处理者
                   if(this.nextHandler != null){
                          response = this.nextHandler.handleMessage(request);
                   }else{
                          //没有适当的处理者，业务自行处理
                   }
            }
            return response;
     }
     //设置下一个处理者是谁
     public void setNext(Handler _handler){
            this.nextHandler = _handler;
     }
     //每个处理者都有一个处理级别
     protected abstract Level getHandlerLevel();
     //每个处理者都必须实现处理任务
     protected abstract Response echo(Request request);
}
```

策略模式

定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。

使用场景：

1. 多个类只有在算法或行为上稍有不同的场景。
2. 算法需要自由切换的场景。
3. 需要屏蔽算法规则的场景。

迭代器模式

它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。

迭代器模式已经被淘汰，java中已经把迭代器运用到各个聚集类（collection）中了，使用java自带的迭代器就已经满足我们的需求了。

组合模式

将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

```
public class Composite extends Component {
     //构件容器
     private ArrayList componentArrayList = new ArrayList();
     //增加一个叶子构件或树枝构件
     public void add(Component component){
            this.componentArrayList.add(component);
     }
     //删除一个叶子构件或树枝构件
     public void remove(Component component){
this.componentArrayList.remove(component);
     }
     //获得分支下的所有叶子构件和树枝构件
     public ArrayList getChildren(){
            return this.componentArrayList;
     }
}
```

使用场景：

1. 维护和展示部分-整体关系的场景，如树形菜单、文件和文件夹管理。
2. 从一个整体中能够独立出部分模块或功能的场景。

观察者模式

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

```
public abstract class Subject {
     //定义一个观察者数组
     private Vector obsVector = new Vector();
     //增加一个观察者
     public void addObserver(Observer o){
            this.obsVector.add(o);
     }
     //删除一个观察者
     public void delObserver(Observer o){
            this.obsVector.remove(o);
     }
     //通知所有观察者
     public void notifyObservers(){
            for(Observer o:this.obsVector){
```

```
                    o.update();
}
    }
}
```

使用场景：

1. 关联行为场景。需要注意的是，关联行为是可拆分的，而不是“组合”关系。
2. 事件多级触发场景。
3. 跨系统的消息交换场景，如消息队列的处理机制

门面模式

要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。

```
public class Facade {
    private subSystem1 subSystem1 = new subSystem1();
    private subSystem2 subSystem2 = new subSystem2();
    private subSystem3 subSystem3 = new subSystem3();
    public void startSystem(){
        subSystem1.start();
        subSystem2.start();
        subSystem3.start();
    }
    public void stopSystem(){
        subSystem1.stop();
        subSystem2.stop();
        subSystem3.stop();
    }
}
```

使用场景：

1. 为一个复杂的模块或子系统提供一个供外界访问的接口
2. 子系统相对独立——外界对子系统的访问只要黑箱操作即可
3. 预防低水平人员带来的风险扩散

备忘录模式

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

```
public class Originator {
    private String state;
    /**
*    工厂方法，返回一个新的备忘录对象
     */
    public Memento createMemento(){
        return new Memento(state);
    }
    /**
*    将发起人恢复到备忘录对象所记载的状态
     */
    public void restoreMemento(Memento memento){
        this.state = memento.getState();
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
        System.out.println("当前状态：" + this.state);
    }
}
```

使用场景：

1. 需要保存和恢复数据的相关状态场景。
2. 提供一个可回滚（rollback）的操作。
3. 需要监控的副本场景中。
4. 数据库连接的事务管理就是用的备忘录模式。

访问者模式

封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。

使用场景：

1. 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作，也就说是用迭代器模式已经不能胜任的情景。
2. 需要对一个对象结构中的对象进行很多不同并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。

状态模式

当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类。

使用场景：

1. 行为随状态改变而改变的场景这也是状态模式的根本出发点，例如权限设计，人员的状态不同即使执行相同的行为结果也会不同，在这种情况下需要考虑使用状态模式。
2. 条件、分支判断语句的替代者

解释器模式

给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

使用场景：

1. 重复发生的问题可以使用解释器模式
2. 一个简单语法需要解释的场景

享元模式

使用共享对象的方法，用来尽可能减少内存使用量以及分享资讯。

```
abstract class flywei{ }
public class Flyweight extends flywei{
    Object obj ;
    public Flyweight(Object obj){
        this.obj = obj;
    }
}
class  FlyweightFactory{     private HashMap data;
    public FlyweightFactory(){ data = new HashMap<>();}
    public Flyweight getFlyweight(Object object){
        if ( data.containsKey(object)){
            return data.get(object);
        }else {
            Flyweight flyweight = new Flyweight(object);
            data.put(object,flyweight);
            return flyweight;
        }
    }
}
```

使用场景：

1. 系统中存在大量的相似对象。
2. 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份。
3. 需要缓冲池的场景。

桥梁模式

将抽象和实现解耦，使得两者可以独立地变化。

Circle类将DrwaApi与Shape类进行了桥接，

```
interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}
class RedCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: red, radius: "
                + radius +", x: " +x+", "+ y +"]");
    }
}
class GreenCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: green, radius: "
                + radius +", x: " +x+", "+ y +"]");
    }
}
abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}
class Circle extends Shape {
    private int x, y, radius;
    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    public void draw() {
        drawAPI.drawCircle(radius,x,y);
    }
}
//客户端使用代码
```

```
Shape redCircle = new Circle(100,100, 10, new RedCircle()); Shape greenCircle = new Circle(100,100, 10, new GreenCircle());
redCircle.draw(); greenCircle.draw();
```

使用场景：

1. 不希望或不适用使用继承的场景
2. 接口或抽象类不稳定的场景
3. 重用性要求较高的场景

模板方法模式

定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

使用场景：

1. 多个子类有公有的方法，并且逻辑基本相同时。
2. 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。
3. 重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数（见"模板方法模式的扩展"）约束其行为。

一千万的用户实时排名如何实现；

海量积分数据实时排名处理

需求概述

积分排名在很多项目都会出现，大家都不会陌生，需求也很简单，积分排名主要满足以下需求：

1. 查询用户名次。
2. 查询TopN(即查询前N名的用户)
3. 实时排名（很多项目是可选的）

当排序的数据量不大的时候，这个需求很容易满足，但是如果数据量很大的时候比如百万级、千万级甚至上亿的时候，或者有实时排名需求；这个时候要满足性能、低成本等需求，在设计上就变得复杂起来了。

常规积分排名处理

这里列举下日常对于排名的常规做法和缺陷。

1. 数据库解决方案

这是最简单的做法，数据存储在数据库里面，然后利用数据库做排序处理。

这里分两种情况：

1.1单库/单表

参与排名的数据量小的时候的做法，所有数据存储在一张表上。

查询操作示例：

查询用户名次:

SELECT count(*) as rank FROM 积分表 WHERE 积分 > (SELECT 积分 FROM 积分表 WHERE uid='用户ID')

查询前N名：

SELECT uid, 积分 FROM 积分表 ORDER BY 积分 DESC LIMIT 0,N

1.2 分库/分表

对于这种情况数据不在一块，在查询操作上跟上面单表情况的区别就是，分库/分表需要做，查询任务切割和查询结果合并处理。

缺陷：

查询排名效率低，会造成扫描大量的记录，甚至全表扫描，性能低，在数据量大、高并发的情况下这种方案是不可用的。

2. 采用常规排序算法

思路上就是把积分排序处理从数据库转移出来，自己实现排序和查询处理。

实际排名业务的特点：

- 每次用户的积分更新都会在一个小的积分范围内波动。
- 已有的积分数据都是已排序的。

常见的几种排序算法大家都熟知这里就不列举了。

缺陷：

对于海量数据排序处理，简单的使用常规排序算法并不合适，要么就是排序造成大量的数据移动、要么就是对已排序的数据查询名次效率不高。

高效的排名算法

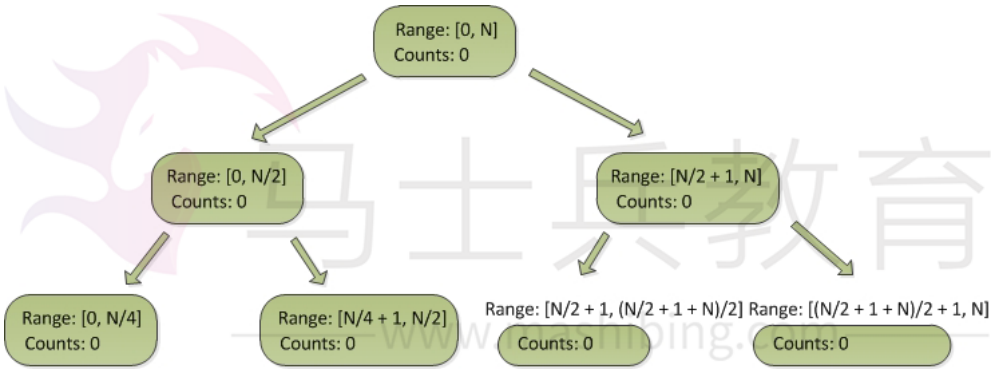
前面的排名算法都是针对积分进行排序，然后通过统计积分高于自己的人数获得排名。

要想知道某个用户的名次，只需要知道比这个用户高分的人数，不一定需要对积分做排序。

在这里换个思路不对积分进行排序，仅仅是统计每个积分区间的人数，用积分区间的形式去统计相应的人数，下面是算法描述。

1. 根据*积分范围*创建平衡二叉树。

设[0, N]为积分范围， 构造的平衡二叉树如下图。



每个节点包含两个数据字段（除了指针）：

Range: 表示积分范围。

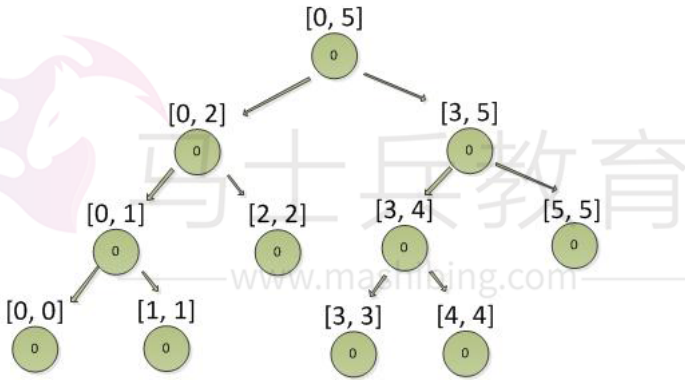
Counts： 表示当前积分区间包含多少人。

积分的区间的划分是根据平分的方式，把当前积分范围一分为二生成两个子节点，然后递归的重复该步骤，直到积分区间无法划分为止（即区间[x, y]， x == y）

例子：

假设积分范围为: [0, 5], 构造的平衡二叉树如下图：

节点内的数据表示当前积分区间的人数。



从上图可以看出来，所有积分都在叶子节点，叶子节点即最小粒度的积分区间。

2. 统计相应积分区间的人数。

这里主要有两种操作：

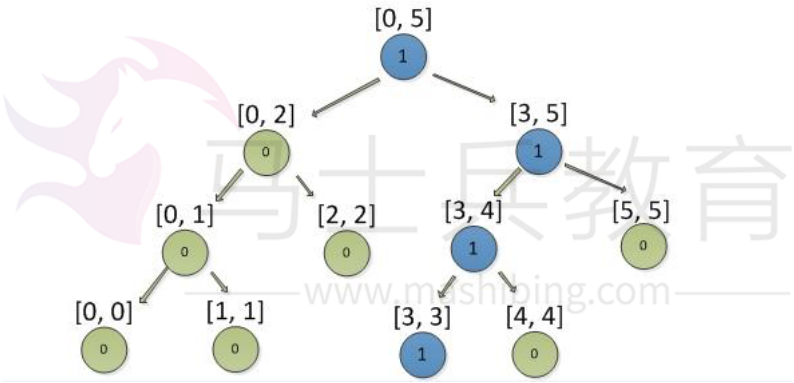
假设积分为i

- 添加积分

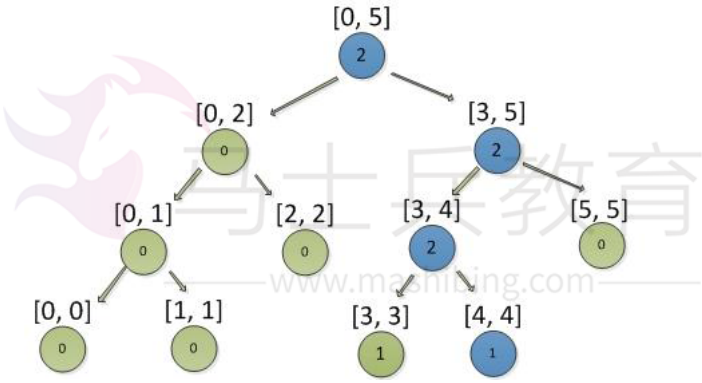
添加积分的过程就是查找积分i， 同时累加查找过程经过的节点计数。

下面给出操作例子，注意观察操作路径。

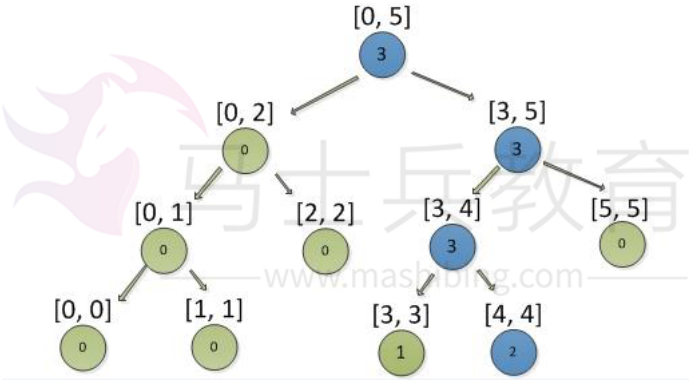
例： 需要添加积分3， 结果如下图



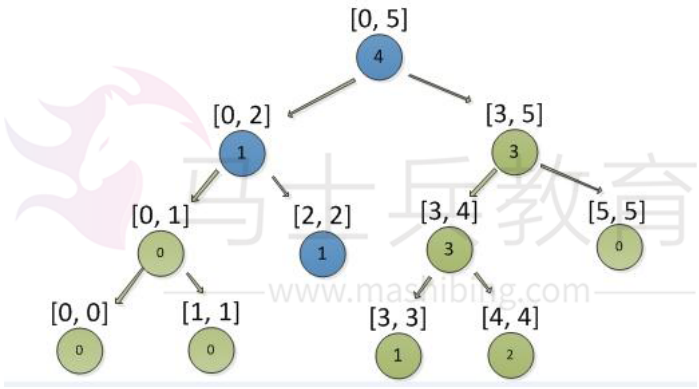
接着在添加积分4，结果如下图



接着再添加积分4，结果如下图



接着添加积分2，结果如下图

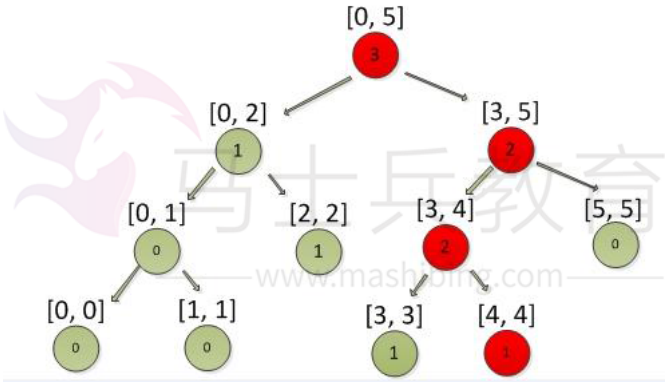


- 删除积分

删除积分的过程也是查找积分i， 区别是查找过程经过的节点计数全部减1。

Ps: 只有积分是存在的情况下，才能做删除操作，另外用一组标记，标识积分是否存在，这里就不列举了。

例子： 删除积分4， 结果如下图



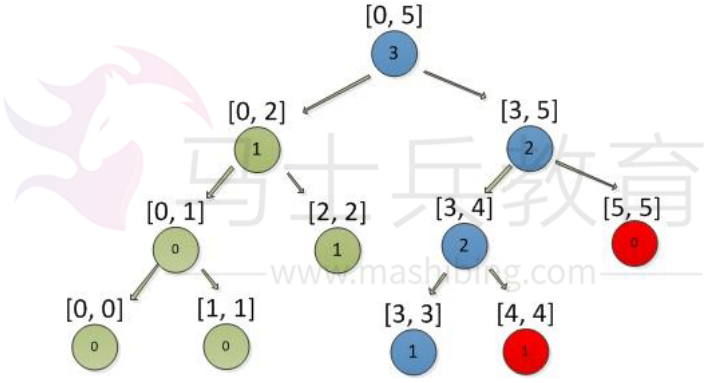
3. 查询名次操作。

查询某个积分的排名的过程也是查找积分i的过程，下面是查找过程统计节点计数的算法：

对于查找路径上的任意节点，如果积分在左节点区间，则累加右节点区间的计数。

最终累加计数的结果加1即是积分的名次

例子： 查找积分3的名次



蓝色节点是查找积分3经过的路径，红色节点是需要累加的计数值。

最终结果是：0 + 1 + 1， 积分3的名次是第2名

从上面的算法可以看出，对平衡二叉树的操作，算法复杂度是O(log N), N是最大积分。

在积分范围不变的情况下，算法复杂度是稳定的，跟用户量无关，因此可以实现海量用户积分排名、实时排名算法需要。

对于海量积分数据实时排名、这里给出的是核心算法，实际业务的时候还需要增加一些额外的处理，比如uid于积分的映射表用于记录用户历史积分、积分与uid的映射表用于TopN这种查询前N名的需求、数据持久化、高可用等需求。

如何高效地获得玩家排名?

这种题目一般很难答，因为具体得看需求，但是暂且抛开需求，我想了一种做法，并没有实践过。
由于排名是动态变化的，因此玩家不需要记录当前是第几名，玩家身上只保存有积分即可。然后可以构建一个以积分为下标的int数组，这个数组多长就看你积分最多会有多少了。（少说1000万也够用了吧？也就4G的内存，这个按具体的需求，如果每周清一次积分？）数组元素表示当前以此下标为积分数对应的玩家数，即：当前有多少个玩家拥有这么多积分。我们把这个数组叫作rank，rank[100]的值就是当前积分为100的玩家的数量。
这个数组在服务器启动的时候根据玩家数重建一份置于内存中（或者共享内存，对于启动速度有要求的话），后续的操作都是增量式操作这个数组，具体的步骤是：当A与B打架完后，取出两者老的排名，在对应的数组元素中减去1，再往新的排名下标元素加上1。这样，它的排序复杂度是O(1)，但是取玩家排名的时候就蛋疼了，因为我要从头到当前的位置扫一遍，把前面数量加一下才可以知道当前是第几名，针对这种情况我觉得可以优化一下，把数组元素扩展多一个变量(above_num)，保存前面的排名数量，这样当A跟B打架的时候，只需要把A跟B之间变化的above_num更新一下即可（一般来讲A跟B变化的区间总不会很大吧）。

如果你一定要立刻获得排名的话，你可以贡献一台服务器不断的做排序，一旦有分数update就立刻排序，这样延迟最多也就几秒钟。你战斗结束之后放个动画，就有准确排名了。

基于redis的zset实现排行榜功能

临近中秋，公司需要开发一款微信小游戏，里面有一个排行榜的功能

主要需求包括：

1. 用户可以上传每次游戏的分数，系统返回该用户的最高分和最高分排名（分数相同时，时间优先）；
2. 用户可以查询排行榜，返回top50，和自己所在的排名

最开始是想使用数据库来实现，保存每个用户最高分的记录，主要字段【name, score, createTime】

针对需求1，用户有新的高分产生的话，就更新用户的最高分，否则返回当前的最高分，获取排名时，需要查询两次数据库（1：查询分数大于自己的记录

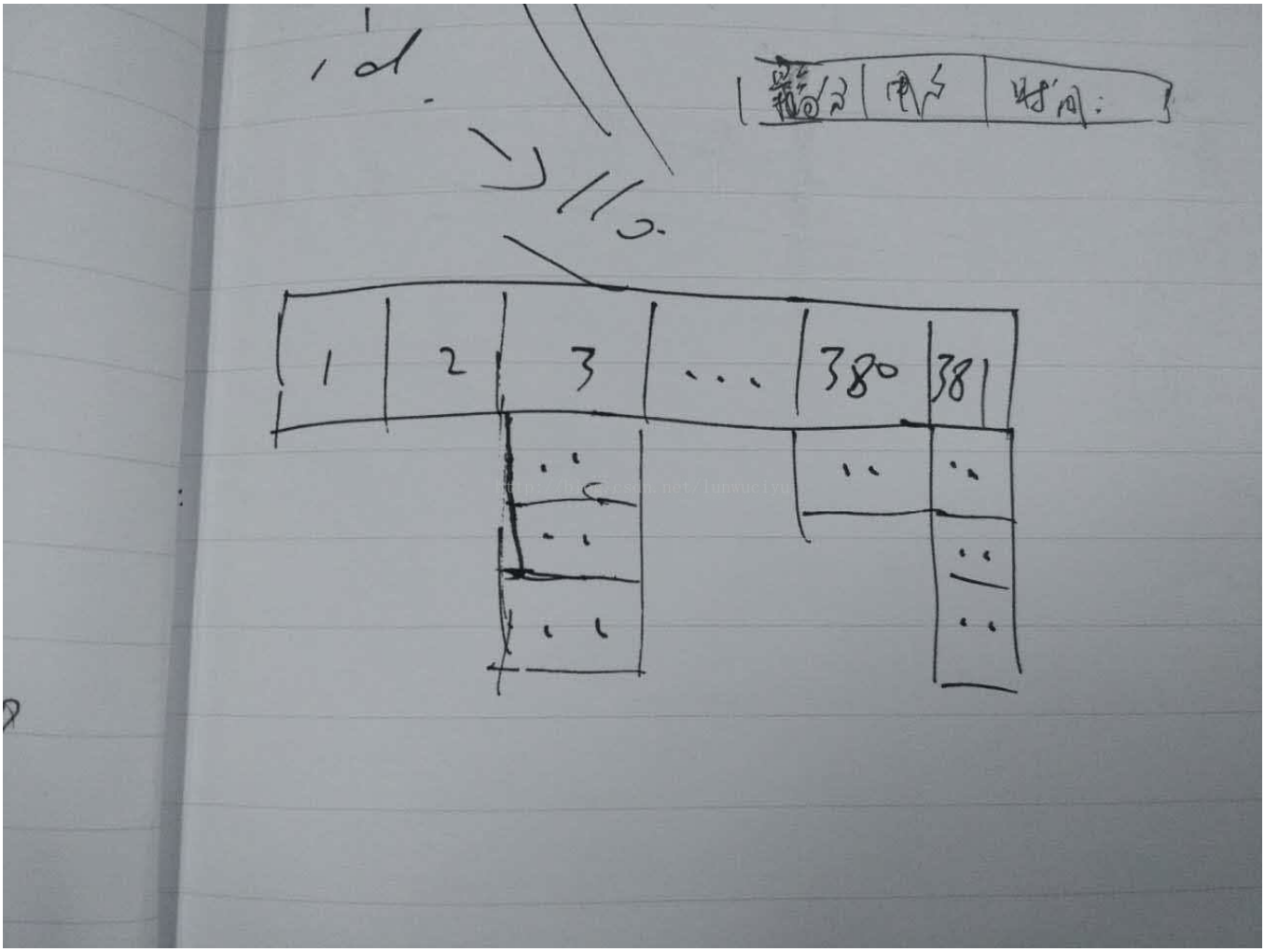
数；2：查询分数相同，时间小于自己的记录数）

针对需求2，按照分数倒序，时间正序查询top50，判断自己如果不是前50，则查询自己的记录，放到列表末尾

这样对于数据库的查询压力会比较大，而且只是一个临时活动，也没必要专门创建一张表来实现

然后和同事讨论可不可以参考HashMap的原理，使用数组+链表的形式的来实现，将分数作为数组下标，每个数组元素上是链表，按照达到该分数的先后

顺序保存用户



针对需求1: 用户上传新的分数时，在对应索引位置末尾增加用户，将原先的用户节点删除（需要另外维护用户原先的最大分数），从末尾循环每个节点的

链表，获取排名和最高分

针对需求2：类似于需求1，从末尾循环每个节点的链表，获取top50

这样每次都需要循环列表，如果用户分数很低的话，循环会比较消耗性能，而且系统重启也会丢失数据。

最后都懂得，搜索引擎，看到redis 的zset原来是解决排行榜的标配，天生就是来做排行榜的，参考了一些网上的文章

redis的zset可以给每个object标记一个分数，然后可以针对这个分数，为object排名，基于hashtable和skiplist执行insert和remove操作，可以通过range方法获取top50，通过rank方法获取排名，完美解决排行榜问题，直接上代码

获取top50的逻辑

```
public List<MidAutumnView> getRangeTop(Long userId) {
        // top
        Set<String> midAutumnStrs = stringRedisTemplate.opsForZSet().range(MID_AUTUMN, 0, TOP_NUM);
        List<MidAutumnView> midAutumnViews = Lists.newArrayList();
        Iterator<String> iterator = midAutumnStrs.iterator();
        int i = 1;
        while (iterator.hasNext()) {
            midAutumnViews.add(convStr2MidAutumnView(iterator.next(), i));
            i++;
        }
        // 判断是否在末尾追加自己
        String midAutumnStr = stringRedisTemplate.opsForValue().get(MID_AUTUMN_USER + userId);
        MidAutumnView midAutumnView = JSONObject.parseObject(midAutumnStr, MidAutumnView.class);
        Long rank = stringRedisTemplate.opsForZSet().rank(MID_AUTUMN,
            convView2ItemStr(midAutumnView));
        if (rank != null && rank > TOP_NUM) {
            midAutumnViews.add(midAutumnView);
        }
        return midAutumnViews;
    }
```

提交新分数的逻辑

```
public MidAutumnView putScore(RedisRankItem redisRankItem) {
        String midAutumnStr = stringRedisTemplate.opsForValue()
            .get(MID_AUTUMN_USER + redisRankItem.getUserId());
        // 首次提交分数
        if (StringUtils.isEmpty(midAutumnStr)) {
```

```
            stringRedisTemplate.opsForZSet().add(MID_AUTUMN, JSONObject.toJSONString(redisRankItem),
                redisRankItem.buildScore(redisRankItem.getScore()));
            Long rank = stringRedisTemplate.opsForZSet().rank(MID_AUTUMN,
                JSONObject.toJSONString(redisRankItem));
            MidAutumnView midAutumnView = MidAutumnView.builder().userId(redisRankItem.getUserId())
               .name(redisRankItem.getName()).portraitUrl(redisRankItem.getPortraitUrl())
                .score(redisRankItem.getScore()).createTime(redisRankItem.getCreateTime())
                .rank(rank.intValue()).maxScore(redisRankItem.getScore()).build();
            stringRedisTemplate.opsForValue().set(MID_AUTUMN_USER + redisRankItem.getUserId(),
                JSONObject.toJSONString(midAutumnView));
            return midAutumnView;
        } else {
            // 二次提交分数
            MidAutumnView midAutumnView = JSONObject.parseObject(midAutumnStr, MidAutumnView.class);
            midAutumnView.setScore(redisRankItem.getScore());
            midAutumnView.setCreateTime(redisRankItem.getCreateTime());
            // 更新最高分
            if (redisRankItem.getScore() > midAutumnView.getScore()) {
                stringRedisTemplate.opsForZSet().remove(MID_AUTUMN,
                    convView2ItemStr(midAutumnView));
                stringRedisTemplate.opsForZSet().add(MID_AUTUMN,
                    JSONObject.toJSONString(redisRankItem),
                   redisRankItem.buildScore(redisRankItem.getScore()));
                Long rank = stringRedisTemplate.opsForZSet().rank(MID_AUTUMN,
                    JSONObject.toJSONString(redisRankItem));
                midAutumnView.setMaxScore(redisRankItem.getScore());
                midAutumnView.setRank(rank.intValue());
                stringRedisTemplate.opsForValue().set(MID_AUTUMN_USER + redisRankItem.getUserId(),
                    JSONObject.toJSONString(midAutumnView));
            }
            return midAutumnView;
        }
    }
```

为了排错，最后用户的每次提交都会记录到mongo

五万人并发抢票怎么实现；

Web大规模高并发请求和抢购的解决方案

电商的秒杀和抢购，对我们来说，都不是一个陌生的东西。然而，从技术的角度来说，这对于Web系统是一个巨大的考验。当一个Web系统，在一秒钟内收到数以万计甚至更多请求时，系统的优化和稳定至关重要。这次我们会关注秒杀和抢购的技术实现和优化，同时，从技术层面揭开，为什么我们总是不容易抢到火车票的原因？

一、大规模并发带来的挑战

在过去的工作中，我曾经面对过5w每秒的高并发秒杀功能，在这个过程中，整个Web系统遇到了很多的问题和挑战。如果Web系统不做针对性的优化，会轻而易举地陷入到异常状态。我们现在一起来讨论下，优化的思路和方法哈。

1. 请求接口的合理设计

一个秒杀或者抢购页面，通常分为2个部分，一个是静态的HTML等内容，另一个就是参与秒杀的Web后台请求接口。

通常静态HTML等内容，是通过CDN的部署，一般压力不大，核心瓶颈实际上在后台请求接口上。这个后端接口，必须能够支持高并发请求，同时，非常重要的一点，必须尽可能“快”，在最短的时间里返回用户的请求结果。为了实现尽可能快这一点，接口的后端存储使用内存级别的操作会更好一点。仍然直接面向MySQL之类的存储是不合适的，如果有这种复杂业务的需求，都建议采用异步写入。

当然，也有一些秒杀和抢购采用“滞后反馈”，就是说秒杀当下不知道结果，一段时间后才可以从页面中看到用户是否秒杀成功。但是，这种属于“偷懒”行为，同时给用户的体验也不好，容易被用户认为是“暗箱操作”。

2. 高并发的挑战：一定要“快”

我们通常衡量一个Web系统的吞吐率的指标是QPS（Query Per Second，每秒处理请求数），解决每秒数万次的高并发场景，这个指标非常关键。举个例子，我们假设处理一个业务请求平均响应时间为100ms，同时，系统内有20台Apache的Web服务器，配置MaxClients为500个（表示Apache的最大连接数目）。

那么，我们的Web系统的理论峰值QPS为（理想化的计算方式）：

20*500/0.1 = 100000 （10万QPS）

咦？我们的系统似乎很强大，1秒钟可以处理完10万的请求，5w/s的秒杀似乎是“纸老虎”哈。实际情况，当然没有这么理想。在高并发的实际场景下，机器都处于高负载的状态，在这个时候平均响应时间会被大大增加。

就Web服务器而言，Apache打开了越多的连接进程，CPU需要处理的上下文切换也越多，额外增加了CPU的消耗，然后就直接导致平均响应时间增加。因此上述的MaxClient数目，要根据CPU、内存等硬件因素综合考虑，绝对不是越多越好。可以通过Apache自带的abench来测试一下，取一个合适的值。然后，我们选择内存操作级别的存储的Redis，在高并发的状态下，存储的响应时间至关重要。网络带宽虽然也是一个因素，不过，这种请求数据包一般比较小，一般很少成为请求的瓶颈。负载均衡成为系统瓶颈的情况比较少，在这里不做讨论哈。

那么问题来了，假设我们的系统，在5w/s的高并发状态下，平均响应时间从100ms变为250ms（实际情况，甚至更多）：

20*500/0.25 = 40000 （4万QPS）

于是，我们的系统剩下了4w的QPS，面对5w每秒的请求，中间相差了1w。

然后，这才是真正的恶梦开始。举个例子，高速路口，1秒钟来5部车，每秒通过5部车，高速路口运作正常。突然，这个路口1秒钟只能通过4部车，车流量仍然依旧，结果必定出现大塞车。（5条车道忽然变成4条车道的感觉）

同理，某一个秒内，20*500个可用连接进程都在满负荷工作中，却仍然有1万个新来请求，没有连接进程可用，系统陷入到异常状态也是预期之内。

其实在正常的非高并发的业务场景中，也有类似的情况出现，某个业务请求接口出现问题，响应时间极慢，将整个Web请求响应时间拉得很长，逐渐将Web服务器的可用连接数占满，其他正常的业务请求，无连接进程可用。

更可怕的问题是，是用户的行为特点，系统越是不可用，用户的点击越频繁，恶性循环最终导致“雪崩”（其中一台Web机器挂了，导致流量分散到其他正常工作的机器上，再导致正常的机器也挂，然后恶性循环），将整个Web系统拖垮。

3. 重启与过载保护

如果系统发生“雪崩”，贸然重启服务，是无法解决问题的。最常见的现象是，启动起来后，立刻挂掉。这个时候，最好在入口层将流量拒绝，然后再将重启。如果是redis/memcache这种服务也挂了，重启的时候需要注意“预热”，并且很可能需要比较长的时间。

秒杀和抢购的场景，流量往往是超乎我们系统的准备和想象的。这个时候，过载保护是必要的。如果检测到系统满负载状态，拒绝请求也是一种保护措施。在前端设置过滤是最简单的方式，但是，这种做法是被用户“千夫所指”的行为。更合适一点的是，将过载保护设置在CGI入口层，快速将客户的直接请求返回。

二、作弊的手段：进攻与防守

秒杀和抢购收到了“海量”的请求，实际上里面的水分是很大的。不少用户，为了“抢”到商品，会使用“刷票工具”等类型的辅助工具，帮助他们发送尽可能多的请求到服务器。还有一部分高级用户，制作强大的自动请求脚本。这种做法的理由也很简单，就是在参与秒杀和抢购的请求中，自己的请求数目占比越多，成功的概率越高。

这些都是属于“作弊的手段”，不过，有“进攻”就有“防守”，这是一场没有硝烟的战斗哈。

1. 同一个账号，一次性发出多个请求

部分用户通过浏览器的插件或者其他工具，在秒杀开始的时间里，以自己的账号，一次发送上百甚至更多的请求。实际上，这样的用户破坏了秒杀和抢购的公平性。

这种请求在某些没有做数据安全处理的系统里，也可能造成另外一种破坏，导致某些判断条件被绕过。例如一个简单的领取逻辑，先判断用户是否有参与记录，如果没有则领取成功，最后写入到参与记录中。这是个非常简单的逻辑，但是，在高并发的场景下，存在深深的漏洞。多个并发请求通过负载均衡服务器，分配到内网的多台Web服务器，它们首先向存储发送查询请求，然后，在某个请求成功写入参与记录的时间差内，其他的请求获查询到的结果都是“没有参与记录”。这里，就存在逻辑判断被绕过的风险。

应对方案：

在程序入口处，一个账号只允许接受1个请求，其他请求过滤。不仅解决了同一个账号，发送N个请求的问题，还保证了后续的逻辑流程的安全。实现方案，可以通过Redis这种内存缓存服务，写入一个标志位（只允许1个请求写成功，结合watch的乐观锁的特性），成功写入的则可以继续参加。

或者，自己实现一个服务，将同一个账号的请求放入一个队列中，处理完一个，再处理下一个。

2. 多个账号，一次性发送多个请求

很多公司的账号注册功能，在发展早期几乎是没有限制的，很容易就可以注册很多个账号。因此，也导致了出现了一些特殊的工作室，通过编写自动注册脚本，积累了一大批“僵尸账号”，数量庞大，几万甚至几十万的账号不等，专门做各种刷的行为（这就是微博中的“僵尸粉”的来源）。举个例子，例如微博中有转发抽奖的活动，如果我们使用几万个“僵尸号”去混进去转发，这样就可以大大提升我们中奖的概率。

这种账号，使用在秒杀和抢购里，也是同一个道理。例如，iPhone官网的抢购，火车票黄牛党。

应对方案：

这种场景，可以通过检测指定机器IP请求频率就可以解决，如果发现某个IP请求频率很高，可以给它弹出一个验证码或者直接禁止它的请求：

1. 弹出验证码，最核心的追求，就是分辨出真实用户。因此，大家可能经常发现，网站弹出的验证码，有些是“鬼神乱舞”的样子，有时让我们根本无法看清。他们这样做的原因，其实也是为了让验证码的图片不被轻易识别，因为强大的“自动脚本”可以通过图片识别里面的字符，然后让脚本自动填写验证码。实际上，有一些非常创新的验证码，效果会比较好，例如给你一个简单问题让你回答，或者让你完成某些简单操作（例如百度贴吧的验证码）。
2. 直接禁止IP，实际上是有些粗暴的，因为有些真实用户的网络场景恰好是同一出口IP的，可能会有“误伤“。但是这一个做法简单高效，根据实际场景使用可以获得很好的效果。

3. 多个账号，不同IP发送不同请求

所谓道高一尺，魔高一丈。有进攻，就会有防守，永不休止。这些“工作室”，发现你对单机IP请求频率有控制之后，他们也针对这种场景，想出了他们的“新进攻方案”，就是不断改变IP。

有同学会好奇，这些随机IP服务怎么来的。有一些是某些机构自己占据一批独立IP，然后做成一个随机代理IP的服务，有偿提供给这些“工作室”使用。还有一些更为黑暗一点的，就是通过木马黑掉普通用户的电脑，这个木马也不破坏用户电脑的正常运作，只做一件事情，就是转发IP包，普通用户的电脑被变成了IP代理出口。通过这种做法，黑客就拿到了大量的独立IP，然后搭建为随机IP服务，就是为了挣钱。

应对方案：

说实话，这种场景下的请求，和真实用户的行为，已经基本相同了，想做分辨很困难。再做进一步的限制很容易“误伤”真实用户，这个时候，通常只能通过设置业务门槛高来限制这种请求了，或者通过账号行为的”数据挖掘“来提前清理掉它们。

僵尸账号也还是有一些共同特征的，例如账号很可能属于同一个号码段甚至是连号的，活跃度不高，等级低，资料不全等等。根据这些特点，适当设置参与门槛，例如限制参与秒杀的账号等级。通过这些业务手段，也是可以过滤掉一些僵尸号。

4. 火车票的抢购

看到这里，同学们是否明白你为什么抢不到火车票？如果你只是老老实实地去抢票，真的很难。通过多账号的方式，火车票的黄牛将很多车票的名额占据，部分强大的黄牛，在处理验证码方面，更是“技高一筹“。

高级的黄牛刷票时，在识别验证码的时候使用真实的人，中间搭建一个展示验证码图片的中转软件服务，真人浏览图片并填写下真实验证码，返回给中转软件。对于这种方式，验证码的保护限制作用被废除了，目前也没有很好的解决方案。

因为火车票是根据身份证实名制的，这里还有一个火车票的转让操作方式。大致的操作方式，是先用买家的身份证开启一个抢票工具，持续发送请求，黄牛账号选择退票，然后黄牛买家成功通过自己的身份证购票成功。当一列车厢没有票了的时候，是没有很多人盯着看的，况且黄牛们的抢票工具也很强大，即使让我们看见有退票，我们也不一定能抢得过他们哈。

最终，黄牛顺利将火车票转移到买家的身份证下。

解决方案：

并没有很好的解决方案，唯一可以动心思的也许是对账号数据进行“数据挖掘”，这些黄牛账号也是有一些共同特征的，例如经常抢票和退票，节假日异常活跃等等。将它们分析出来，再做进一步处理和甄别。

三、高并发下的数据安全

我们知道在多线程写入同一个文件的时候，会存现“线程安全”的问题（多个线程同时运行同一段代码，如果每次运行结果和单线程运行的结果是一样的，结果和预期相同，就是线程安全的）。如果是MySQL数据库，可以使用它自带的锁机制很好的解决问题，但是，在大规模并发的场景中，是不推荐使用MySQL的。秒杀和抢购的场景中，还有另外一个问题，就是“超发”，如果在这方面控制不慎，会产生发送过多的情况。我们也曾经听说过，某些电商搞抢购活动，买家成功拍下后，商家却不承认订单有效，拒绝发货。这里的问题，也许并不一定是商家奸诈，而是系统技术层面存在超发风险导致的。

1. 超发的原因

假设某个抢购场景中，我们一共只有100个商品，在最后一刻，我们已经消耗了99个商品，仅剩最后一个。这个时候，系统发来多个并发请求，这批请求读取到的商品余量都是99个，然后都通过了这一个余量判断，最终导致超发。（同文章前面说的场景）

在上面的这个图中，就导致了并发用户B也“抢购成功”，多让一个人获得了商品。这种场景，在高并发的情况下非常容易出现。

2. 悲观锁思路

解决线程安全的思路很多，可以从“悲观锁”的方向开始讨论。

悲观锁，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待。

虽然上述的方案的确解决了线程安全的问题，但是，别忘记，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有机会抢到这个“锁”，这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗尽，系统陷入异常。

3. FIFO队列思路

那好，那么我们稍微修改一下上面的场景，我们直接将请求放入队列中的，采用FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程的感觉哈。

然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终Web系统平均响应时候还是会大幅下降，系统还是陷入异常。

4. 乐观锁思路

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。

有很多软件和服务都“乐观锁”功能的支持，例如Redis中的watch就是其中之一。通过这个实现，我们保证了数据的安全。

四、小结

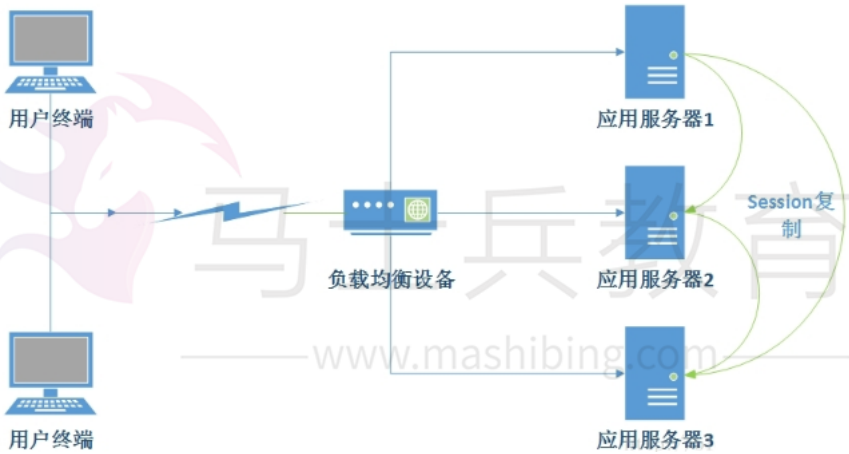
互联网正在高速发展，使用互联网服务的用户越多，高并发的场景也变得越来越多。电商秒杀和抢购，是两个比较典型的互联网高并发场景。虽然我们解决问题的具体技术方案可能千差万别，但是遇到的挑战却是相似的，因此解决问题的思路也异曲同工。

集群服务器Session同步

事实上，网站总是有状态的。每一个登录信息、用户信息常常被存储在session内部。而当一个网站被部署在不止一台服务器的时候，就会遇到session同步的问题。事实上即使一个很小的网站，也要至少有两台服务器互为备份，分单流量是必须得，更重要的是无缝切流量升级。为了保证服务的不间断又要进行网站的维护升级，切流量是最简单的。那么如何保证切流量的时候session也会跟着同步过去呢？在集群环境下，大致有以下几种手段：

一、Session复制

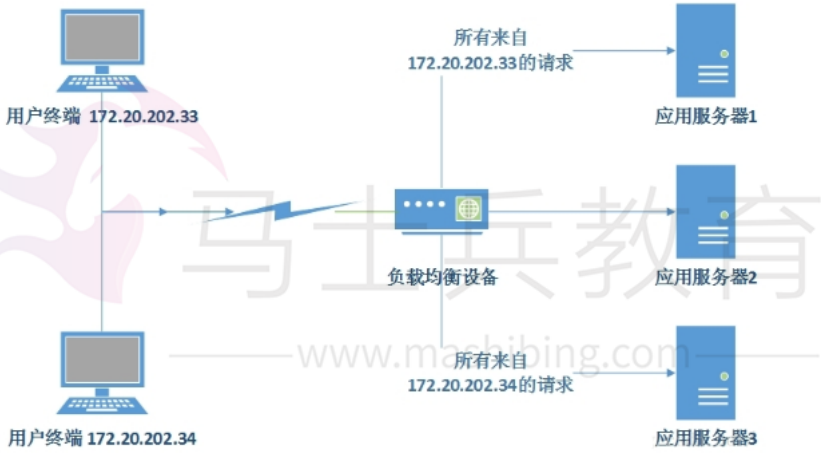
这是一种在早期应用系统中使用较多的服务器session管理方式。应用服务器开启Web容器的session的复制功能，在集群中的几台服务器之间同步session对象，这样一台服务器宕机不会导致session数据丢失。即每一台服务器都持有集群中所有的session，每次访问仅从本机获取就可以了。其工作形式如下所示：



从session复制的几条线就可以看出，这种方式仅适用用小型集群。当服务集群规模很大时，集群服务器间的复制就需要大量的通讯，占用大量网络资源，甚至会出现内存不够的情况。

二、Session绑定

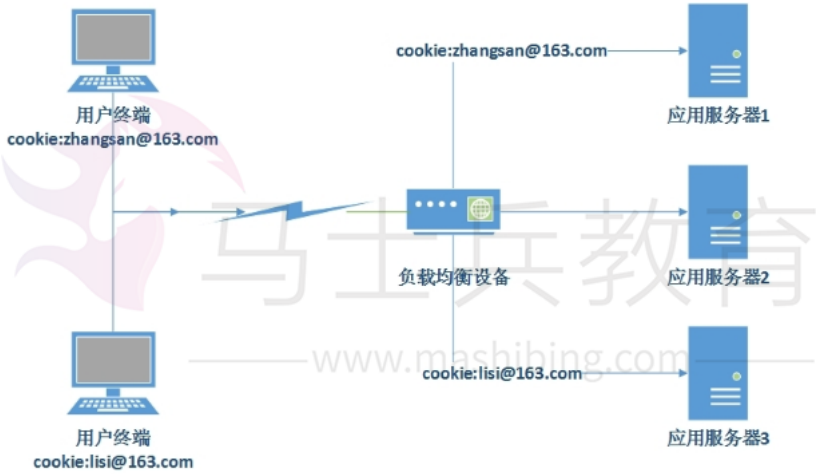
Session绑定可以利用负载均衡的源地址Hash算法实现，负载均衡服务器总是将来自同一个IP地址的访问分发到同一台服务器上。这样整个会话期间，用户所有的请求都来自一台服务器，保证了Session总是从这台服务器获取。其工作形式如下图所示：



但是这样的系统显然不符合我们对系统的需求。如果一台服务器宕机，那么其处理的所有请求Session会话全部丢失，用户因为切换服务器后没有Session而导致无法完成业务。

三、利用Cookie记录Session

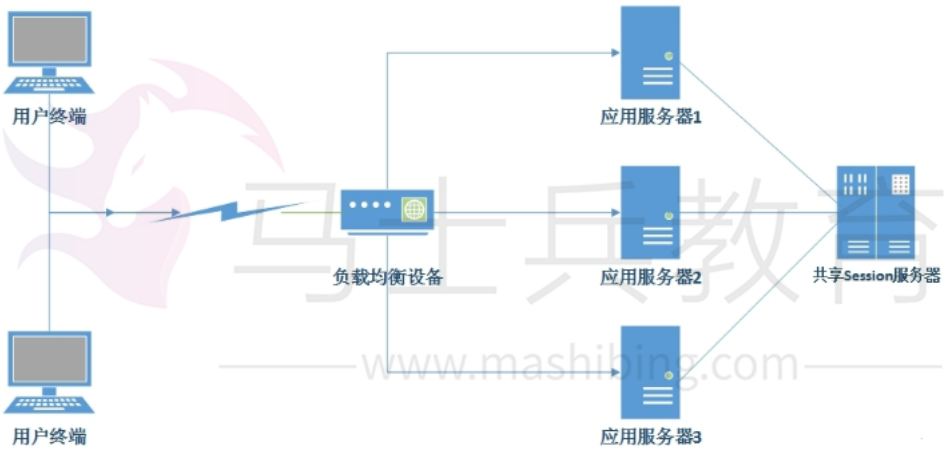
这种管理方式将Session记录在客户端，每次请求服务器的时候，将Session放在请求中发送给服务器，服务器处理完成后再将修改后的Session响应给客户端。



利用Cookie记录当然也有缺点，比如Cookie大小限制，能记录的信息也有限，因为很多时候我们在Session中储存的也并非String类型的记录。每次请求都需要传输Cookie，影响性能；另外如果用户关闭Cookie功能就不能用了。但是这种方式因此高可用性、支持服务器的线性伸缩，许多网站都在使用这种方式。我的学校网站也应用了这种技术。

四、Session服务器

如果有这样一个服务器，可用性高、伸缩性好、性能也不错，对信息大小又没有限制，那它就是Session服务器。利用独立部署的Session服务器统一管理Session，应用服务器每次读写Session时，都访问Session服务器。其工作形式如下所示：



这种方式实际上是将应用服务器的状态分离，分为无状态的应用服务器和有状态的Session服务器，然后针对这两种服务器的不同特性分别设计其架构。

对于有状态的Session服务器，一种比较简单的方式是利用分布式缓存、数据库等。

大数据量高并发的数据库优化

一、数据库结构的设计

如果不能设计一个合理的数据库模型，不仅会增加客户端和服务器段程序的编程和维护的难度，而且将会影响系统实际运行的性能。所以，在一个系统开始实施之前，完备的数据库模型的设计是必须的。

在一个系统分析、设计阶段，因为数据量较小，负荷较低。我们往往只注意到功能的实现，而很难注意到性能的薄弱之处，等到系统投入实际行一段时间后，才发现系统的性能在降低，这时再来考虑提高系统性能则要花费更多的人力物力，而整个系统也不可避免的形成了一个打补丁工程。

所以在考虑整个系统的流程的时候，我们必须要考虑，在高并发大数据量的访问情况下，我们的系统会不会出现极端的情况。（例如：对外统计系统在7月16日出现的数据异常的情况，并发大数据量的的访问造成，数据库的响应时间不能跟上数据刷新的速度造成。具体情况是：在日期临界时（00：00：00），判断数据库中是否有当前日期的记录，没有则插入一条当前日期的记录。在低并发访问的情况下，不会发生问题，但是当日期临界时的访问量相当大的时候，在做这一判断的时候，会出现多次条件成立，则数据库里会被插入多条当前日期的记录，从而造成数据错误。），数据库的模型确定下来之后，我们有必要做一个系统内数据流向图，分析可能出现的瓶颈。

为了保证数据库的一致性和完整性，在逻辑设计的时候往往会设计过多的表间关联，尽可能的降低数据的冗余。（例如用户表的地区，我们可以把地区另外存放到一个地区表中）如果数据冗余低，数据的完整性容易得到保证，提高了数据吞吐速度，保证了数据的完整性，清楚地表达数据元素之间的关系。而对于多表之间的关联查询（尤其是大数据表）时，其性能将会降低，同时也提高了客户端程序的编程难度，因此，物理设计需折衷考虑，根据业务规则，确定对关联表的数据量大小、数据项的访问频度，对此类数据表频繁的关联查询应适当提高数据冗余设计但增加了表间连接查询的操作，也使得程序的变得复杂，为了提高系统的响应时间，合理的数据冗余也是必要的。设计人员在设计阶段应根据系统操作的类型、频度加以均衡考虑。

另外，最好不要用自增属性字段作为主键与子表关联。不便于系统的迁移和数据恢复。对外统计系统映射关系丢失（******）。

原来的表格必须可以通过由它分离出去的表格重新构建。使用这个规定的好处是，你可以确保不会在分离的表格中引入多余的列，所有你创建的表格结构都与它们的实际需要一样大。应用这条规定是一个好习惯，不过除非你要处理一个非常大型的数据，否则你将不需要用到它。（例如一个通行证系统，我可以将USERID，USERNAME，USERPASSWORD，单独出来作个表，再把USERID作为其他表的外键）

表的设计具体注意的问题：

1、数据行的长度不要超过8020字节，如果超过这个长度的话在物理页中这条数据会占用两行从而造成存储碎片，低查询效率。

2、能够用数字类型的字段尽量选择数字类型而不用字符串类型的（电话号码），这会降低查询和连接的性能，并增加存储开销。这是因为引擎在处理查询和连接回逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

3、对于不可变字符类型char和可变字符类型varchar 都是8000字节,char查询快，但是耗存储空间，varchar查询相对慢一些但是节省存储空间。在设计字段的时候可以灵活选择，例如用户名、密码等长度变化不大的字段可以选择CHAR，对于评论等长度变化大的字段可以选择VARCHAR。

4、字段的长度在最大限度的满足可能的需要的前提下，应该尽可能的设得短一些，这样可以提高查询的效率，而且在建立索引的时候也可以减少资源的消耗。

二、查询的优化

保证在实现功能的基础上，尽量减少对数据库的访问次数；通过搜索参数，尽量减少对表的 访问行数,最小化结果集，从而减轻网络负担；能够分开的操作尽量分开处理，提高每次的 响应速度；在数据窗口使用 SQL 时，尽量把使用的索引放在选择的首列；算法的结构尽量 简单；在查询时，不要过多地使用通配符如 SELECT * FROM T1 语句，要用到几列就选择 几列如：SELECT COL1,COL2 FROM T1；在可能的情况下尽量限制尽量结果集行数如： SELECT TOP 300 COL1,COL2,COL3 FROM T1,因为某些情况下用户是不需要那么多的数据 的。

在没有建索引的情况下，数据库查找某一条数据，就必须进行全表扫描了，对所有数据进行 一次遍历，查找出符合条件的记录。在数据量比较小的情况下，也许看不出明显的差别，但 是当数据量大的情况下，这种情况就是极为糟糕的了。

SQL 语句在 SQL SERVER 中是如何执行的，他们担心自己所写的 SQL 语句会被 SQL SERVER 误解。

比如：

select * from table1 where name='zhangsan' and tID > 10000

和执行:

select * from table1 where tID > 10000 and name='zhangsan'

一些人不知道以上两条语句的执行效率是否一样，因为如果简单的从语句先后上看，这两个 语句的确是不一样，如果 tID 是一个聚合索引，那么后一句仅仅从表的 10000 条以后的记录 中查找就行了；而前一句则要先从全表中查找看有几个 name='zhangsan'的，而后再根据限 制条件条件 tID>10000 来提出查询结果。

事实上，这样的担心是不必要的。SQL SERVER 中有一个“查询分析优化器”，它可以计算 出 where 子句中的搜索条件并确定哪个索引能缩小表扫描的搜索空间，也就是说，它能实现 自动优化。虽然查询优化器可以根据 where 子句自动的进行查询优化，但有时查询优化器就 会不按照您的本意进行快速查询。

在查询分析阶段，查询优化器查看查询的每个阶段并决定限制需要扫描的数据量是否有用。 如果一个阶段可以被用作一个扫描参数（SARG），那么就称之为可优化的，并且可以利用 索引快速获得所需数据。

SARG 的定义：用于限制搜索的一个操作，因为它通常是指一个特定的匹配，一个值的范围 内的匹配或者两个以上条件的 AND 连接。形式如下：

列名 操作符 <常数 或 变量> 或 <常数 或 变量> 操作符 列名

列名可以出现在操作符的一边，而常数或变量出现在操作符的另一边。如：

Name='张三'

价格>5000

5000<价格

Name='张三' and 价格>5000

如果一个表达式不能满足 SARG 的形式，那它就无法限制搜索的范围了，也就是 SQL SERVER 必须对每一行都判断它是否满足 WHERE 子句中的所有条件。所以一个索引对于 不满足 SARG 形式的表达式来说是无用的。

> 所以，优化查询最重要的就是，尽量使语句符合查询优化器的规则避免全表扫描而使用 索引查询。

具体要注意的：

1.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进 行全表扫描，如：
select id from t where num is null
可以在 num上设置默认值 0，确保表中 num列没有 null 值，然后这样查询：
select id from t where num=0

2.应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫 描。优化器将无法通过索引来确定将要命中的行数,因此需要搜索该表的所有行。

3.应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全 表扫描，如：
select id from t where num=10 or num=20 可以这样查询：
select id from t where num=10
union all
select id from t where num=20

4.in 和 not in 也要慎用，因为 IN 会使系统无法使用索引,而只能直接搜索表中的数据。如：
select id from t where num in(1,2,3)
对于连续的数值，能用 between 就不要用 in 了：
select id from t where num between 1 and 3

5.尽量避免在索引过的字符数据中，使用非打头字母搜索。这也使得引擎无法利用索引。
见如下例子：
SELECT * FROM T1 WHERE NAME LIKE '%L%'
SELECT * FROM T1 WHERE SUBSTING(NAME,2,1)='L'
SELECT * FROM T1 WHERE NAME LIKE 'L%'
即使 NAME 字段建有索引，前两个查询依然无法利用索引完成加快操作，引擎不得不对全 表所有数据逐条操作来完成任务。而第三个查询能够使用索引来加快操作。

6.必要时强制查询优化器使用某个索引，如在 where 子句中使用参数，也会导致全表扫描。
因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行 时；
它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的， 因而无法作为索引选择的输入项。如下面语句将进行全表扫描：
select id from t where num=@num
可以改为强制查询使用索引：
select id from t with(index(索引名)) where num=@num

7.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行 全表扫描。
如： SELECT * FROM T1 WHERE F1/2=100 应改为: SELECT * FROM T1 WHERE F1=100*2

SELECT * FROM RECORD WHERE SUBSTRING(CARD_NO,1,4)='5378'
应改为: SELECT * FROM RECORD WHERE CARD_NO LIKE '5378%'

SELECT member_number, first_name, last_name FROM members
WHERE DATEDIFF(yy,datofbirth,GETDATE()) > 21
应改为:
SELECT member_number, first_name, last_name FROM members
WHERE dateofbirth < DATEADD(yy,-21,GETDATE())
即：任何对列的操作都将导致表扫描，它包括数据库函数、计算表达式等等，查询时要尽可 能将操作移至等号右边。

8.应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表 扫描。如：
select id from t where substring(name,1,3)='abc'--name 以 abc 开头的 id
select id from t where datediff(day,createdate,'2005-11-30')=0--'2005-11-30'
生成的 id 应改为:
select id from t where name like 'abc%'
select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'

9.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可 能无法正确使用索引

10.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一 个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让 字段顺序与索引顺序相一致

11.很多时候用 exists 是一个好的选择：
elect num from a where num in(select num from b)
用下面的语句替换：
select num from a where exists(select 1 from b where num=a.num)

SELECT SUM(T1.C1)FROM T1 WHERE(
(SELECT COUNT(*)FROM T2 WHERE T2.C2=T1.C2>0)
SELECT SUM(T1.C1) FROM T1WHERE EXISTS(
SELECT * FROM T2 WHERE T2.C2=T1.C2)
两者产生相同的结果，但是后者的效率显然要高于前者。因为后者不会产生大量锁定的表扫 描或是索引扫描。

如果你想校验表里是否存在某条纪录，不要用 count()*那样效率很低，而且浪费服务器资源。 可以用 EXISTS 代替。如:*
IF (SELECT COUNT() FROM table_name WHERE column_name = 'xxx')
可以写成:
IF EXISTS (SELECT * FROM table_name WHERE column_name = 'xxx')

经常需要写一个 T_SQL 语句比较一个父结果集和子结果集，从而找到是否存在在父结果集 中有而在子结果集中没有的记录，如:
SELECT a.hdr_key FROM hdr_tbl a---- tbl a 表示 tbl 用别名 a 代替
WHERE NOT EXISTS (SELECT * FROM dtl_tbl b WHERE a.hdr_key = b.hdr_key)
SELECT a.hdr_key FROM hdr_tbl a LEFT JOIN dtl_tbl b ON a.hdr_key = b.hdr_key
WHERE b.hdr_key IS NULL SELECT hdr_key FROM hdr_tbl WHERE hdr_key NOT IN (SELECT hdr_key FROM dtl_tbl)
三种写法都可以得到同样正确的结果，但是效率依次降低。

12.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有 主键索引）。

13.避免频繁创建和删除临时表，以减少系统表资源的消耗。

14.临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引 用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

15.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先 create table，然后 insert。

16.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

17.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。 无 需 在 执 行 存 储 过 程 和 触 发 器 的 每 个 语 句 后 向 客 户 端 发 送 DONE_IN_PROC 消息。

18.尽量避免大事务操作，提高系统并发能力。

19.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

20. 避免使用不兼容的数据类型。例如 float 和 int、char 和 varchar、binary和 varbinary是不 兼容的。数据类型的不兼容可能使优化器无法执行一些本来可以进行的优化操作。例如: SELECT name FROM employee WHERE salary > 60000
在这条语句中,如salary字段是money型的,则优化器很难对其进行优化,因为60000是个整型 数。我们应当在编程时将整型转化成为钱币型,而不要等到运行时转化。

21.充分利用连接条件，在某种情况下，两个表之间可能不只一个的连接条件，这时在 WHERE 子句中将连接条件完整的写上，有可能大大提高查询速度。
例:
SELECT SUM(A.AMOUNT) FROM ACCOUNT A,CARD B WHERE A.CARD_NO = B.CARD_NO
SELECT SUM(A.AMOUNT) FROM ACCOUNT A,CARD B WHERE A.CARD_NO = B.CARD_NO AND A.ACCOUNT_NO=B.ACCOUNT_NO
第二句将比第一句执行快得多。

22、使用视图加速查询
把表的一个子集进行排序并创建视图，有时能加速查询。它有助于避免多重排序 操作，而 且在其他方面还能简化优化器的工作。例如:
SELECT cust.name，rcvbles.balance，……other columns
FROM cust，rcvbles
WHERE cust.customer_id = rcvlbes.customer_id
AND rcvblls.balance>0
AND cust.postcode>“98000”
ORDER BY cust.name

如果这个查询要被执行多次而不止一次，可以把所有未付款的客户找出来放在一个视图中， 并按客户的名字进行排序:
CREATE VIEW DBO.V_CUST_RCVLBES
AS
SELECT cust.name，rcvbles.balance，……other columns
FROM cust，rcvbles
WHERE cust.customer_id = rcvlbes.customer_id
AND rcvblls.balance>0
ORDER BY cust.name
然后以下面的方式在视图中查询:
SELECT * FROM V_CUST_RCVLBES
WHERE postcode>“98000”
视图中的行要比主表中的行少，而且物理顺序就是所要求的顺序，减少了磁盘 I/O，所以查 询工作量可以得到大幅减少。

23、能用DISTINCT的就不用 GROUP BY
SELECT OrderID FROM Details WHERE UnitPrice > 10 GROUP BY OrderID
可改为:
SELECT DISTINCT OrderID FROM Details WHERE UnitPrice > 10

24.能用 UNION ALL就不要用 UNION
UNION ALL不执行 SELECT DISTINCT函数，这样就会减少很多不必要的资源

35.尽量不要用 SELECT INTO 语句。

SELECT INOT 语句会导致表锁定，阻止其他用户访问该表。

上面我们提到的是一些基本的提高查询速度的注意事项,但是在更多的情况下,往往需要 反复试验比较不同的语句以得到最佳方案。最好的方法当然是测试，看实现相同功能的 SQL 语句哪个执行时间最少，但是数据库中如果数据量很少，是比较不出来的，这时可以用查看 执行计划，即：把实现相同功能的多条 SQL语句考到查询分析器，按 CTRL+L 看查所利用 的索引，表扫描次数（这两个对性能影响最大），总体上看询成本百分比即可

三、算法的优化

尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过 1 万行，那么就应该考 虑改写。.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题， 基于集的方法通常更有效。与临时表一样，游标并不是不可使用。对小型数据集使AST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发 时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。
游标提供了对特定集合中逐行扫描的手段，一般使用游标逐行遍历数据，根据取出的数 据不同条件进行不同的操作。尤其对多表和大表定义的游标（大的数据集合）循环很容易使 程序进入一个漫长的等特甚至死机。
在有些场合，有时也非得使用游标，此时也可考虑将符合条件的数据行转入临时表中， 再对临时表定义游标进行操作，可时性能得到明显提高。 （例如：对内统计第一版） 封装存储过程

四、建立高效的索引

创建索引一般有以下两个目的：维护被索引列的唯一性和提供快速访问表中数据的策略。 大型数据库有两种索引即簇索引和非簇索引，一个没有簇索引的表是按堆结构存储数据，所 有的数据均添加在表的尾部，而建立了簇索引的表，其数据在物理上会按照簇索引键的顺序 存储，一个表只允许有一个簇索引，因此，根据 B 树结构，可以理解添加任何一种索引均 能提高按索引列查询的速度，但会降低插入、更新、删除操作的性能，尤其是当填充因子（Fill Factor）较大时。所以对索引较多的表进行频繁的插入、更新、删除操作，建表和索引时因 设置较小的填充因子，以便在各数据页中留下较多的自由空间，减少页分割及重新组织的工 作。

索引是从数据库中获取数据的最高效方式之一。95% 的数据库性能问题都可以采用索引技 术得到解决。作为一条规则，我通常对逻辑主键使用唯一的成组索引，对系统键（作为存储
过程）采用唯一的非成组索引，对任何外键列[字段]采用非成组索引。不过，索引就象是盐， 太多了菜就咸了。你得考虑数据库的空间有多大，表如何进行访问，还有这些访问是否主要 用作读写。

实际上，您可以把索引理解为一种特殊的目录。微软的 SQL SERVER 提供了两种索引：聚 集索引（clustered index，也称聚类索引、簇集索引）和非聚集索引（nonclustered index，也 称非聚类索引、非簇集索引）。下面，我们举例来说明一下聚集索引和非聚集索引的区别： 其实，我们的汉语字典的正文本身就是一个聚集索引。比如，我们要查“安”字，就会很自 然地翻开字典的前几页，因为“安”的拼音是“an”，而按照拼音排序汉字的字典是以英文 字母“a”开头并以“z”结尾的，那么“安”字就自然地排在字典的前部。如果您翻完了所 有以“a”开头的部分仍然找不到这个字，那么就说明您的字典中没有这个字；同样的，如 果查“张”字，那您也会将您的字典翻到最后部分，因为“张”的拼音是“zhang”。也就是 说，字典的正文部分本身就是一个目录，您不需要再去查其他目录来找到您需要找的内容。 我们把这种正文内容本身就是一种按照一定规则排列的目录称为“聚集索引”。

如果您认识某个字，您可以快速地从自动中查到这个字。但您也可能会遇到您不认识的字， 不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而需要去根据“偏旁 部首”查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。但您结 合“部首目录”和“检字表”而查到的字的排序并不是真正的正文的排序方法，比如您查“张” 字，我们可以看到在查部首之后的检字表中“张”的页码是 672页，检字表中“张”的上面 是“驰”字，但页码却是 63 页，“张”的下面是“弩”字，页面是 390 页。很显然，这些字并不是真正的分别位于“张”字的上下方，现在您看到的连续的“驰、张、弩”三字实际上 就是他们在非聚集索引中的排序，是字典正文中的字在非聚集索引中的映射。我们可以通过这种方式来找到您所需要的字，但它需要两个过程，先找到目录中的结果，然后再翻到您所 需要的页码。

我们把这种目录纯粹是目录，正文纯粹是正文的排序方式称为“非聚集索引”。 进一步引申一下，我们可以很容易的理解：每个表只能有一个聚集索引，因为目录只能按照 一种方法进行排序。

（一）何时使用聚集索引或非聚集索引

下面的表总结了何时使用聚集索引或非聚集索引（很重要）。

动作描述 使用聚集索引 使用非聚集索引

列经常被分组排序 应 应

返回某范围内的数据 应 不应

一个或极少不同值 不应 不应

小数目的不同值 应 不应

大数目的不同值 不应 应

频繁更新的列 不应 应

外键列 应 应

主键列 应 应

频繁修改索引列 不应 应

事实上，我们可以通过前面聚集索引和非聚集索引的定义的例子来理解上表。如：返回某范 围内的数据一项。比如您的某个表有一个时间列，恰好您把聚合索引建立在了该列，这时您 查询 2004 年 1 月 1 日至2004 年 10 月 1 日之间的全部数据时，这个速度就将是很快的，因
为您的这本字典正文是按日期进行排序的，聚类索引只需要找到要检索的所有数据中的开头 和结尾数据即可；而不像非聚集索引，必须先查到目录中查到每一项数据对应的页码，然后 再根据页码查到具体内容。

（二）结合实际，谈索引使用的误区

理论的目的是应用。虽然我们刚才列出了何时应使用聚集索引或非聚集索引，但在实践中以 上规则却很容易被忽视或不能根据实际情况进行综合分析。下面我们将根据在实践中遇到的 实际问题来谈一下索引使用的误区，以便于大家掌握索引建立的方法。

1、主键就是聚集索引

这种想法笔者认为是极端错误的，是对聚集索引的一种浪费。虽然 SQL SERVER 默认是在 主键上建立聚集索引的。 通常，我们会在每个表中都建立一个 ID 列，以区分每条数据，并且这个 ID 列是自动增大 的，步长一般为 1。我们的这个办公自动化的实例中的列 Gid就是如此。此时，如果我们将 这个列设为主键，SQL SERVER 会将此列默认为聚集索引。这样做有好处，就是可以让您 的数据在数据库中按照 ID 进行物理排序，但笔者认为这样做意义不大。

显而易见，聚集索引的优势是很明显的，而每个表中只能有一个聚集索引的规则，这使得聚 集索引变得更加珍贵。

从我们前面谈到的聚集索引的定义我们可以看出，使用聚集索引的最大好处就是能够根据查 询要求，迅速缩小查询范围，避免全表扫描。在实际应用中，因为 ID 号是自动生成的，我 们并不知道每条记录的 ID 号，所以我们很难在实践中用 ID 号来进行查询。这就使让 ID 号 这个主键作为聚集索引成为一种资源浪费。其次，让每个 ID 号都不同的字段作为聚集索引 也不符合“大数目的不同值情况下不应建立聚合索引”规则；当然，这种情况只是针对用户 经常修改记录内容，特别是索引项的时候会负作用，但对于查询速度并没有影响。

在办公自动化系统中，无论是系统首页显示的需要用户签收的文件、会议还是用户进行文件 查询等任何情况下进行数据查询都离不开字段的是“日期”还有用户本身的“用户名”。通常，办公自动化的首页会显示每个用户尚未签收的文件或会议。虽然我们的 where 语句可 以仅仅限制当前用户尚未签收的情况，但如果您的系统已建立了很长时间，并且数据量很大， 那么，每次每个用户打开首页的时候都进行一次全表扫描，这样做意义是不大的，绝大多数 的用户 1 个月前的文件都已经浏览过了，这样做只能徒增数据库的开销而已。事实上，我们 完全可以让用户打开系统首页时，数据库仅仅查询这个用户近 3个月来未阅览的文件，通过 “日期”这个字段来限制表扫描，提高查询速度。如果您的办公自动化系统已经建立的 2 年，那么您的首页显示速度理论上将是原来速度8 倍，甚至更快。

2、只要建立索引就能显著提高查询速度

事实上，我们可以发现上面的例子中，第2、3 条语句完全相同，且建立索引的字段也相同； 不同的仅是前者在 fariqi 字段上建立的是非聚合索引，后者在此字段上建立的是聚合索引， 但查询速度却有着天壤之别。所以，并非是在任何字段上简单地建立索引就能提高查询速度。
从建表的语句中，我们可以看到这个有着1000万数据的表中fariqi字段有5003个不同记录。 在此字段上建立聚合索引是再合适不过了。在现实中，我们每天都会发几个文件，这几个文 件的发文日期就相同，这完全符合建立聚集索引要求的：“既不能绝大多数都相同，又不能 只有极少数相同”的规则。由此看来，我们建立“适当”的聚合索引对于我们提高查询速度 是非常重要的。

3、把所有需要提高查询速度的字段都加进聚集索引，以提高查询速度

上面已经谈到：在进行数据查询时都离不开字段的是“日期”还有用户本身的“用户名”。 既然这两个字段都是如此的重要，我们可以把他们合并起来，建立一个复合索引（compound index）。

很多人认为只要把任何字段加进聚集索引，就能提高查询速度，也有人感到迷惑：如果把复 合的聚集索引字段分开查询，那么查询速度会减慢吗？带着这个问题，我们来看一下以下的 查询速度（结果集都是 25 万条数据）：（日期列 fariqi 首先排在复合聚集索引的起始列，用 户名 neibuyonghu排在后列）

我们可以看到如果仅用聚集索引的起始列作为查询条件和同时用到复合聚集索引的全部列 的查询速度是几乎一样的，甚至比用上全部的复合索引列还要略快（在查询结果集数目一样 的情况下）；而如果仅用复合聚集索引的非起始列作为查询条件的话，这个索引是不起任何 作用的。当然，语句 1、2 的查询速度一样是因为查询的条目数一样，如果复合索引的所有 列都用上，而且查询结果少的话，这样就会形成“索引覆盖”，因而性能可以达到最优。同 时，请记住：无论您是否经常使用聚合索引的其他列，但其前导列一定要是使用最频繁的列。

（三）其他注意事项 “水可载舟，亦可覆舟”，索引也一样。索引有助于提高检索性能，但过多或不当的索引也 会导致系统低效。因为用户在表中每加进一个索引，数据库就要做更多的工作。过多的索引 甚至会导致索引碎片。 所以说，我们要建立一个“适当”的索引体系，特别是对聚合索引的创建，更应精益求精， 以使您的数据库能得到高性能的发挥

如何提高服务器并发处理能力

说明

以下内容为入门级介绍，意在对老技术作较全的总结而不是较深的研究。主要参考《构建高性能Web站点》一书。

什么是服务器并发处理能力

一台服务器在单位时间里能处理的请求越多，服务器的能力越高，也就是服务器并发处理能力越强

有什么方法衡量服务器并发处理能力

1. 吞吐率

吞吐率，单位时间里服务器处理的最大请求数，单位req/s

从服务器角度，实际并发用户数的可以理解为服务器当前维护的代表不同用户的文件描述符总数，也就是并发连接数。服务器一般会限制同时服务的最多用户数，比如apache的MaxClents参数。

这里再深入一下，对于服务器来说，服务器希望支持高吞吐率，对于用户来说，用户只希望等待最少的时间，显然，双方不能满足，所以双方利益的平衡点，就是我们希望的最大并发用户数。

2. 压力测试

有一个原理一定要先搞清楚，假如100个用户同时向服务器分别进行10个请求，与1个用户向服务器连续进行1000次请求，对服务器的压力是一样吗？实际上是不一样的，因对每一个用户，连续发送请求实际上是指发送一个请求并接收到响应数据后再发送下一个请求。这样对于1个用户向服务器连续进行1000次请求, 任何时刻服务器的网卡接收缓冲区中只有1个请求，而对于100个用户同时向服务器分别进行10个请求，服务器的网卡接收缓冲区最多有100个等待处理的请求，显然这时的服务器压力更大。

压力测试前提考虑的条件

- 并发用户数: 指在某一时刻同时向服务器发送请求的用户总数(HttpWatch)
- 总请求数
- 请求资源描述
- 请求等待时间(用户等待时间)
- 用户平均请求的等待时间
- 服务器平均请求处理的时间
- 硬件环境

压力测试中关心的时间又细分以下2种:

1. 用户平均请求等待时间（这里暂不把数据在网络的传输时间，还有用户PC本地的计算时间计算入内）
2. 服务器平均请求处理时间

用户平均请求等待时间主要用于衡量服务器在一定并发用户数下，单个用户的服务质量；而服务器平均请求处理时间就是吞吐率的倒数，一般来说，用户平均请求等待时间 = 服务器平均请求处理时间 * 并发用户数

怎么提高服务器的并发处理能力

1. 提高CPU并发计算能力

服务器之所以可以同时处理多个请求，在于操作系统通过多执行流体系设计使得多个任务可以轮流使用系统资源，这些资源包括CPU，内存以及I/O. 这里的I/O主要指磁盘I/O, 和网络I/O。

多进程 & 多线程

多执行流的一般实现便是进程，多进程的好处可以对CPU时间的轮流使用，对CPU计算和IO操作重叠利用。这里的IO主要是指磁盘IO和网络IO，相对CPU而言，它们慢的可怜。

而实际上，大多数进程的时间主要消耗在I/O操作上。现代计算机的DMA技术可以让CPU不参与I/O操作的全过程，比如进程通过系统调用，使得CPU向网卡或者磁盘等I/O设备发出指令，然后进程被挂起，释放出CPU资源，等待I/O设备完成工作后通过中断来通知进程重新就绪。对于单任务而言，CPU大部分时间空闲，这时候多进程的作用尤为重要。

多进程不仅能够提高CPU的并发度。其优越性还体现在独立的内存地址空间和生命周期所带来的稳定性和健壮性，其中一个进程崩溃不会影响到另一个进程。

但是进程也有如下缺点：

1. fork()系统调用开销很大: prefork
2. 进程间调度和上下文切换成本: 减少进程数量
3. 庞大的内存重复：共享内存
4. IPC编程相对比较麻烦

减少进程切换

当硬件上下文频繁装入和移出时，所消耗的时间是非常可观的。可用Nmon工具监视服务器每秒的上下文切换次数。

为了尽量减少上下文切换次数，最简单的做法就是减少进程数，尽量使用线程并配合其它I/O模型来设计并发策略。

还可以考虑使用进程绑定CPU技术，增加CPU缓存的命中率。若进程不断在各CPU上切换，这样旧的CPU缓存就会失效。

减少使用不必要的锁

服务器处理大量并发请求时，多个请求处理任务时存在一些资源抢占竞争，这时一般采用"锁"机制来控制资源的占用，当一个任务占用资源时，我们锁住资源，这时其它任务都在等待锁的释放，这个现象称为锁竞争。

通过锁竞争的本质，我们要意识到尽量减少并发请求对于共享资源的竞争。比如在允许情况下关闭服务器访问日志，这可以大大减少在锁等待时的延迟时间。要最大程度减少无辜的等待时间。

这里说下无锁编程，就是由内核完成这个锁机制，主要是使用原子操作替代锁来实现对共享资源的访问保护 ,使用原子操作时，在进行实际的写操作时，使用了lock指令，这样就可以阻止其他任务写这块内存，避免出现数据竞争现象。原子操作速度比锁快，一般要快一倍以上。

例如fwrite(), fopen()，其是使用append方式写文件，其原理就是使用了无锁编程，无锁编程的复杂度高，但是效率快，而且发生死锁概率低。

考虑进程优先级

进程调度器会动态调整运行队列中进程的优先级，通过top观察进程的PR值

考虑系统负载

可在任何时刻查看/proc/loadavg, top中的load average也可看出

考虑CPU使用率

除了用户空间和内核空间的CPU使用率以外，还要关注I/O wait,它是指CPU空闲并且等待I/O操作完成的时间比例（top中查看wa的值）。

2. 考虑减少内存分配和释放

服务器的工作过程中，需要大量的内存，使得内存的分配和释放工作尤为重要。

可以通过改善数据结构和算法复制度来适当减少中间临时变量的内存分配及数据复制时间，而服务器本身也使用了各自的策略来提高效率。

例如Apache,在运行开始时一次申请大片的内存作为内存池，若随后需要时就在内存池中直接获取，不需要再次分配，避免了频繁的内存分配和释放引起的内存整理时间。

再如Nginx使用多线程来处理请求，使得多个线程之间可以共享内存资源，从而令它的内存总体使用量大大减少，另外，nginx分阶段的内存分配策略，按需分配，及时释放，使得内存使用量保持在很小的数量范围。

另外，还可以考虑共享内存。

共享内存指在多处理器的计算机系统中，可以被不同中央处理器（CPU）访问的大容量内存，也可以由不同进程共享，是非常快的进程通信方式。

但是使用共享内存也有不好的地方，就是对于多机器时数据不好统一。

shell命令ipcs可用来显示系统下共享内存的状态，函数shmget可以创建或打开一块共享内存区，函数shmat将一个存在的共享内存段连接到本进程空间, 函数shmctl可以对共享内存段进行多种操作，函数shmdt函数分离该共享内存。

3. 考虑使用持久连接

持久连接也为长连接，它本身是TCP通信的一种普通方式，即在一次TCP连接中持续发送多分数据而不断开连接，与它相反的方式称为短连接，也就是建立连接后发送一份数据就断开，然后再次建立连接发送下一份数据， 周而复始。是否采用持久连接，完全取决于应用特点。从性能角度看，建立TCP连接的操作本身是一项不小的开销，在允许的情况下，连接次数越少，越有利于性能的提升; 尤其对于密集型的图片或网页等小数据请求处理有明显的加速所用。

HTTP长连接需要浏览器和web服务器的共同协作，目前浏览器普遍支持长连接，表现在其发出的HTTP请求数据头中包含关于长连接的声明，如下： Connection: Keep-Alive

主流的web服务器都支持长连接，比如apache中，可以用KeepAlive off关闭长连接。

对于长连接的有效使用，还有关键一点在于长连接超时时间的设置，即长连接在什么时候关闭吗？ Apache的默认设置为5s, 若这个时间设置过长，则可能导致资源无效占有，维持大量空闲进程，影响服务器性能。

4. 改进I/O 模型

I/O操作根据设备的不同分为很多类型，比如内存I/O, 网络I/O, 磁盘I/O. 对于网络I/O和磁盘I/O, 它们的速度要慢很多，尽管使用RAID磁盘阵列可通过并行磁盘磁盘来加快磁盘I/O速度，购买大连独享网络带宽以及使用高带宽网络适配器可以提高网络i/O的速度。但这些I/O操作需要内核系统调用来完成，这些需要CPU来调度，这使得CPU不得不浪费宝贵的时间来等待慢速I/O操作。我们希望让CPU足够少的时间在i/O操作的调度上，如何让高速的CPU和慢速的I/O设备更好地协调工作，是现代计算机一直探讨的话题。各种I/O模型的本质区别在于CPU的参与方式。

1. DMA技术

I/O设备和内存之间的数据传输方式由DMA控制器完成。在DMA模式下，CPU只需向DMA下达命令，让DMA控制器来处理数据的传送，这样可以大大节省系统资源。

2. 异步I/O

异步I/O指主动请求数据后便可以继续处理其它任务，随后等待I/O操作的通知，这样进程在数据读写时不发生阻塞。

异步I/O是非阻塞的，当函数返回时，真正的I/O传输已经完成，这让CPU处理和I/O操作达到很好的重叠。

3. I/O多路复用

epoll服务器同时处理大量的文件描述符是必不可少的，若采用同步非阻塞I/O模型，若同时接收TCP连接的数据，就必须轮流对每个socket调用接收数据的方法，不管这些socket有没有可接收的数据，都要询问一次。假如大部分socket并没有数据可以接收，那么进程便会浪费很多CPU时间用于检查这些socket有没有可以接收的数据。多路I/O就绪通知的出现，提供了对大量文件描述符就绪检查的高性能方案，它允许进程通过一种方法同时监视所有文件描述符，并可以快速获得所有就绪的文件描述符，然后只针对这些文件描述符进行数据访问。

epoll可以同时支持水平触发和边缘触发，理论上边缘触发性能更高，但是代码实现复杂，因为任何意外的丢失事件都会造成请求处理错误。

epoll主要有2大改进：

1. epoll只告知就绪的文件描述符，而且当调用epoll_wait()获得文件描述符时，返回并不是实际的描述符，而是一个代表就绪描述符数量的值，然后只需去epoll指定的一个数组中依次取得相应数量的文件描述符即可，这里使用了内存映射(mmap)技术，这样彻底省掉了这些文件描述符在系统调用时复制的开销。
2. epoll采用基于事件的就绪通知方式。其事先通过epoll_ctrl()注册每一个文件描述符，一旦某个文件描述符就绪时，内核会采用类似callback的回调机制，当进程调用epoll_wait()时得到通知

4. Sendfile

大多数时候，我们都向服务器请求静态文件，比如图片，样式表等，在处理这些请求时，磁盘文件的数据先经过内核缓冲区，然后到用户内存空间，不需经过任何处理，其又被送到网卡对应的内核缓冲区，接着再被送入网卡进行发送。

Linux提供sendfile()系统调用，可以讲磁盘文件的特定部分直接传送到代表客户端的socket描述符，加快了静态文件的请求速度，同时减少CPU和内存的开销。

适用场景： 对于请求较小的静态文件，sendfile发挥的作用不那么明显，因发送数据的环节在整个过程中所占时间的比例相比于大文件请求时小很多。

5. 内存映射

Linux内核提供一种访问磁盘文件的特殊方式，它可以将内存中某块地址空间和我们指定的磁盘文件相关联，从而对这块内存的访问转换为对磁盘文件的访问。这种技术称为内存映射。

多数情况下，内存映射可以提高磁盘I/O的性能，无须使用read()或write()等系统调用来访问文件，而是通过mmap()系统调用来建立内存和磁盘文件的关联，然后像访问内存一样自由访问文件。

缺点：在处理较大文件时，内存映射会导致较大的内存开销，得不偿失。

6. 直接I/O

在linux 2.6中，内存映射和直接访问文件没有本质差异，因为数据需要经过2次复制，即在磁盘与内核缓冲区之间以及在内核缓冲区与用户态内存空间。

引入内核缓冲区的目的在于提高磁盘文件的访问性能，然而对于一些复杂的应用，比如数据库服务器，它们为了进一步提高性能，希望绕过内核缓冲区，由自己在用户态空间实现并管理I/O缓冲区，比如数据库可根据更加合理的策略来提高查询缓存命中率。另一方面，绕过内核缓冲区也可以减少系统内存的开销，因内核缓冲区本身就在使用系统内存。

Linux在open()系统调用中增加参数选项O_DIRECT,即可绕过内核缓冲区直接访问文件,实现直接I/O。

在Mysql中，对于Innodb存储引擎，自身进行数据和索引的缓存管理，可在my.cnf配置中分配raw分区跳过内核缓冲区，实现直接I/O。

5. 改进服务器并发策略

服务器并发策略的目的，是让I/O操作和CPU计算尽量重叠进行，一方面让CPU在I/O等待时不要空闲，另一方面让CPU在I/O调度上尽量花最少的时间。

一个进程处理一个连接，非阻塞I/O

这样会存在多个并发请求同时到达时，服务器必然要准备多个进程来处理请求。其进程的开销限制了它的并发连接数。但从稳定性和兼容性的角度，则其相对安全，任何一个子进程的崩溃不会影响服务器本身，父进程可以创建新的子进程；这种策略典型的例子就是Apache的fork和prefork模式。对于并发数不高（如150以内）的站点同时依赖Apache其它功能时的应用选择Apache还是可以的。

一个线程处理一个连接，非阻塞IO

这种方式允许在一个进程中通过多个线程来处理多个连接，一个线程处理一个连接。Apache的worker模式就是这种典型例子，使其可支持更多的并发连接。不过这种模式的总体性能还不如prefork，所以一般不选用worker模式。

一个进程处理多个连接，异步I/O

一个线程同时处理多个连接，潜在的前提条件就是使用IO多路复用就绪通知。

这种情况下，将处理多个连接的进程叫做worker进程或服务进程。worker的数量可以配置，如Nginx中的worker_processes 4。

一个线程处理多个连接，异步IO

即使有高性能的IO多路复用就绪通知，但磁盘IO的等待还是无法避免的。更加高效的方法是对磁盘文件使用异步IO，目前很少有Web服务器真正意义上支持这种异步IO。

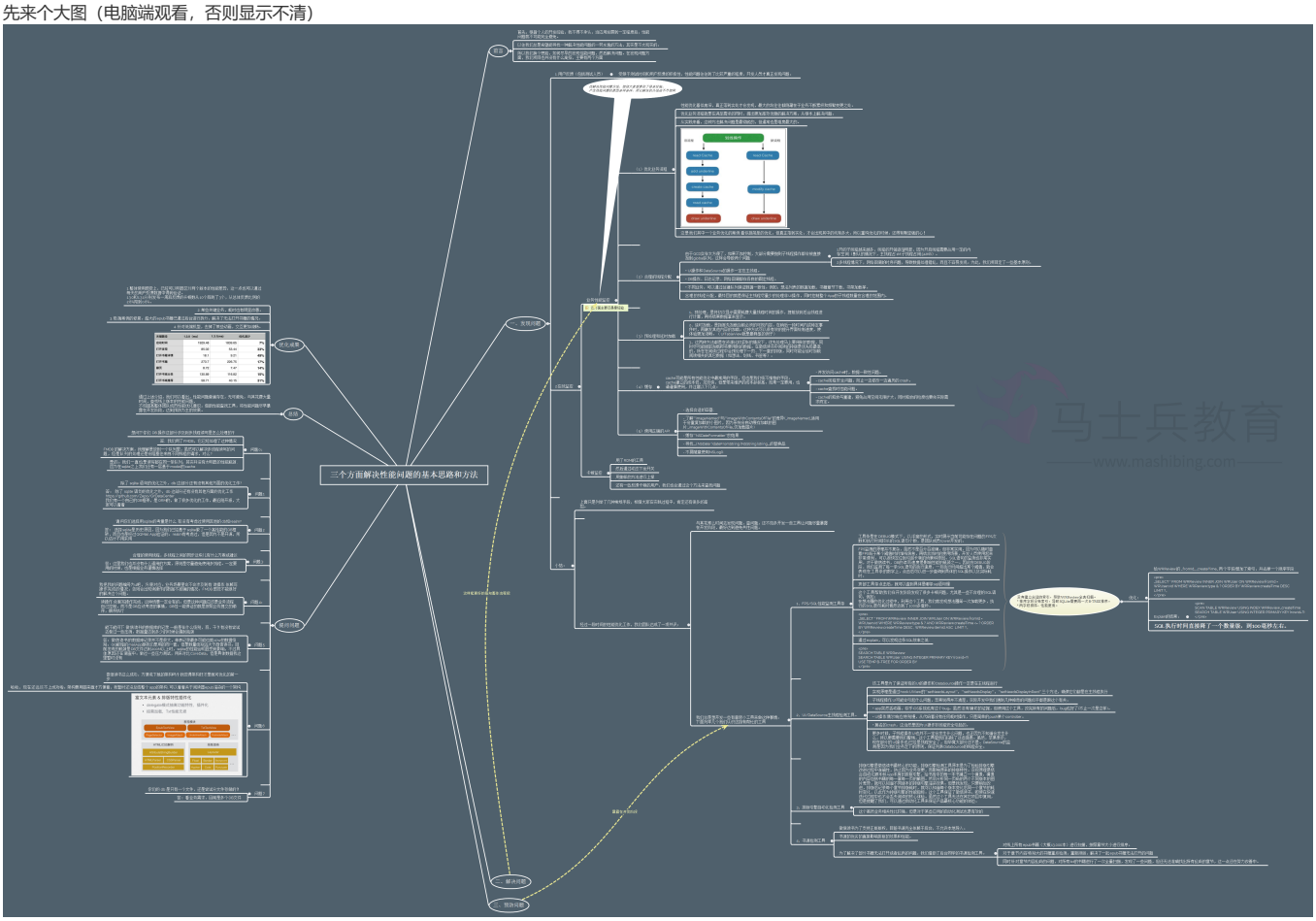
6. 改进硬件环境

还有一点要提及的是硬件环境，服务器的硬件配置对应用程序的性能提升往往是最直接，也是最简单的方式，这就是所谓的scale up。这里不做论述。

假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题。

三个方面解决性能问题的基本思路和方法

先来个大图（电脑端观看，否则显示不清）



发现问题、解决问题和预防问题三个方面

> 首先，根据个人的开发经验，我不得不承认，当应用发展到一定程度后，性能问题就不可能完全避免。
> 以往我们总是希望能寻找一种解决性能问题的一劳永逸的方法，其实是不太现实的。所以我们换个思路，如何尽早的发现性能问题，然后解决问题。
> 在发现问题方面，我们项目也并没有什么高招，主要有两个方面

1. 用户反馈（包括测试人员）

受限于测试时间和用户反馈的积极性，性能问题往往到了比较严重的程度，开发人员才真正发现问题。

2.在线监控

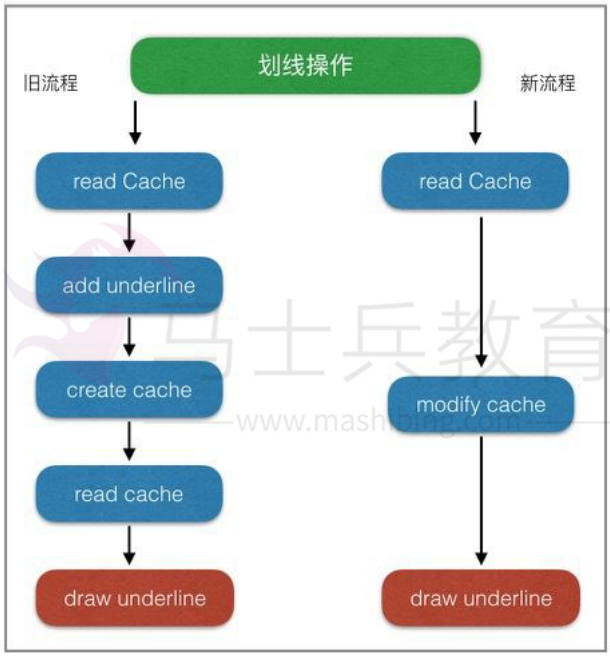
在线监控主要有业务性能监控和卡顿监控
卡顿监控，是用了RDM的工具，然后通过动态下发开关，用抽样的方法进行上报
还有一些反馈卡顿的用户，我们也会通过这个方法来查找问题

> 然后，在解决性能问题方法，相信大家都累积了很多经验。
> 产生性能问题的原因多种多样，所以解决的办法也不尽相同，各种奇技淫巧都有可能派上用场，这里我大概介绍一下我们项目中用到的一些方面：

（1）优化业务流程 （2）合理的线程分配 （3）预处理和延时加载 （4）缓存 （5）使用正确的API

(1). 优化业务流程

性能优化看似高深，真正落到实处才会发现，最大的坑往往都隐藏在于业务不断累积和频繁变更之处。优化业务流程就是在满足需求的同时，提出更加高效优雅的解决方案，从根本上解决问题。从实践来看，这种方法解决问题是最彻底的，但通常也是难度最大的。



这是我们其中一个业务优化的案例,看似挺简单的优化，但真正落到实处，才会出现其中的坑有多大，所以重构优化的时候，还得有颗坚强的心！

(2). 合理的线程分配

由于GCD实在太方便了，如果不加控制，大部分需要抛到子线程操作都会被直接加到global队列，这样会导致两个问题，1.开的子线程越来越多，线程的开销逐渐明显，因为开启线程需要占用一定的内存空间（默认的情况下，主线程占1M,子线程占用512KB）。2.多线程情况下，网络回调的时序问题，导致数据处理错乱，而且不容易发现。为此，我们项目定了一些基本原则。

UI操作和DataSource的操作一定在主线程。 DB操作、日志记录、网络回调都在各自的固定线程。 不同业务，可以通过创建队列保证数据一致性。例如，想法列表的数据加载、书籍章节下载、书架加载等。

合理的线程分配，最终目的就是保证主线程尽量少的处理非UI操作，同时控制整个App的子线程数量在合理的范围内。

(3). 预处理和延时加载。

预处理，是将初次显示需要耗费大量线程时间的操作，提前放到后台线程进行计算，再将结果数据拿来显示。

延时加载，是指首先加载当前必须的可视内容，在稍后一段时间内或特定事件时，再触发其他内容的加载。这种方式可以很有效的提升界面绘制速度，使体验更加流畅。（UITableView就是最典型的例子）

这两种方法都是在资源比较紧张的情况下，优先处理马上要用到的数据，同时尽可能提前加载即将要用到的数据。在微信读书中阅读的排版是优先级最高的，所在在阅读过程中会预处理下一页、下一章的排版，同时可能会延时加载阅读相关的其它数据（如想法、划线、书签等）。

(4). 缓存

cache可能是所有性能优化中最常用的手段，但也是我们极不推荐的手段。cache建立的成本低，见效快，但是带来维护的成本却很高。如果一定要用，也请谨慎使用，并注意以下几点：

并发访问cache时，数据一致性问题。 cache线程安全问题，防止一边修改一边遍历的crash。 cache查找时性能问题。 cache的释放与重建，避免占用空间无限扩大，同时释放的粒度也要依实际需求而定。

(5).使用正确的API

选择合适的容器; 了解**imageNamed**与**imageWithContentsOfFile**的差异(*imageNamed*适用于会重复加载的小图片，因为系统会自动缓存加载的图片;*imageWithContentsOfFile*仅加载图片) 缓存**NSDateFormatter**的结果。 寻找**(NSDate *)dateFromString:(NSString)string**的替换品 不要随意使用NSLog();

> 这方面主要还是靠经验的累积
> 上面只是列举了几种常规手段，相信大家在实践过程中，肯定还有很多的高招。
> 经过一段时间的性能优化工作，我们团队达成了一项共识，与其花那么时间去发现问题，查问题，还不如多开发一些工具，让问题尽量暴露在开发阶段，最好达到避免共性问题。
> 所以，我们总是想开发一些有意思小工具来做这种事情。
> 下面列举几个**我们认识还挺有帮忙的工具**

(1).FPS/SQL性能监测工具条

工具条是在DEBUG模式下，以浮窗的形式，实时展示当前可能存在问题的FPS次数和执行时间较长的SQL语句个数，是团队成员tower开发的。

FPS监测的原理并不复杂，虽然不是百分百准确，但非常实用，因为可以随时查看FPS低于某个阈值时的堆栈信息，再结合当时的使用场景，开发人员使用起来非常便利，可以很快定位到引起卡顿的场景和原因。SQL语句的监测也非常实用，对于微信读书，DB的读写速度是影响性能的瓶颈之一。因此在DEBUG阶段，我们监测了每一条SQL语句的执行速度，一旦执行时间超出某个阈值，就会表现在工具条的数字上，点击后可以进一步查询到具体的SQL操作以及实际耗时。

顶部工具条点击后，就可以查到具体是哪条sql语句慢 这个工具帮助我们在开发阶段发现了很多卡顿问题，尤其是一些不合理的SQL语句，例如：
在想法圈的优化过程中，利用这个工具，我们就发现想法圈第一次加载更多，执行的SQL语句耗时竟然达到了1000多毫秒。

```
_SELECT * FROM WRReview INNER JOIN WRUser ON WRReview.fromId = WRUser.vid WHERE WRReview.type & ? AND WRReview.createTime <= ? ORDER BY WRReview.createTime DESC , WRReview.itemId ASC LIMIT ?_
```

通过explain，可以发现这条SQL效率之低:

```
SEARCH TABLE WRReview SEARCH TABLE WRUser USING INTEGER PRIMARY KEY (rowid=?) USE TEMP B-TREE FOR ORDER BY
```

没有建立合适的索引，导致WRReview全表扫描。 排序字段没有索引，导致SQLite需要再一次B-TREE排序。 两字段排序，性能更低。

优化：给WRReview的 *fromId createTime* 两个字段增加了索引，并去掉一个排序字段:

```
_SELECT * FROM WRReview INNER JOIN WRUser ON WRReview.fromId = WRUser.vid WHERE WRReview.type & ? ORDER BY WRReview.createTime DESC LIMIT ?_
```

Explain的结果：

```
SCAN TABLE WRReview USING INDEX WRReview_createTime SEARCH TABLE WRUser USING INTEGER PRIMARY KEY (rowid=?)
```

SQL执行时间直接降了一个数量级，到100毫秒左右。

(2). UI/DataSource主线程检测工具。

该工具是为了保证所有的UI的操作和DataSource操作一定是在主线程进行。实现原理是通过hook UIView的**setNeedsLayout**，**setNeedsDisplay**，**setNeedsDisplayInRect**三个方法，确保它们都是在主线程执行。子线程操作UI可能会引起什么问题，苹果说得并不清楚，实际开发中我们遇到几种神奇的问题似乎都是跟这个有关。

app突然丢动画，似乎iOS系统也有这个bug。虽然没有确切的证据，但使用这个工具，改完所有的问题后，bug也好了(不止一次是这样)。

UI操作偶尔响应特别慢，从代码看没有任何耗时操作，只是简单的push某个controller。

莫名的crash，这当然是因为UI操作非线程安全引起的。

更多时候，子线程操作UI也并不一定会发生什么问题，也正因为不知道会发生什么，所以更需要我们警惕，这个工具替我们扫除了这些隐患。虽然，苹果表示，现在部分的UI操作也已经是线程安全了，但毕竟大部分还不是。DataSource的监测是因为我们业务定下的原则，保证列表DataSource的线程安全。

(3) 排版引擎自动化检测工具

排版引擎是微信读书最核心的功能，排版引擎检测工具原本是为了检验排版引擎改进过程中准确性，防止因为业务变更，而影响原来的排版特性。实现原理是结合自动化脚本和App本身的排版引擎，给书库中的每一本书建立一个镜像，镜像的内容包括书籍的每一章每一页的截图。然后分析同一页码的两个不同版本的图片差异，就可以知道不同版本的排版引擎渲染效果。但是我发现，只要稍加改进，排版后记录每个章节排版耗时，就可以知道每个版本变化后同一个章节的耗时变化，以此作为排版引擎的性能指标。这个工具保证了微信读书，即使在快速迭代过程中也不会丢失阅读的核心体验。虽然这个工具无法在其它项目中复用，但是提醒了我们，可以通过自动化工具来保证产品最核心功能的体验。

这个虽然业务相关性比较强，但是对于某些应用的自动化测试也是有效的

(4) 书源检测工具

微信读书为了支持正版版权，目前书源完全依赖于后台，不允许本地导入。书源的优劣的直接影响排版的效果和性能。为了解决了部分书籍无法打开或者乱码的问题，我们借助了后台同学的书源检测工具。对线上所有epub书籍（大概13,000本）进行扫描，按照章节大小进行排序。对于章节内容特别大的书籍重点检测，重新排版，解决了一批epub书籍无法打开的问题。同时针对章节内容乱码的问题，对所有txt的书籍进行了一次全量扫描，发现了一些问题，但还无法准确找出所有乱码的章节，这一点还在努力改善中。

优化成果

整体使用感受上，已经可以明显区分两个版本的性能差异，这一点也可以通过每天的用户反馈数据中得到验证。
1.3.0和1.3.1分别发布一周后反馈的卡顿数从10个降到了3个，从总体反馈比例的2.8%降到0.8%。 某些关键业务，耗时也有明显改善。 极端案例的修复。超大的epub书籍已通过后台进行拆分，解决了无法打开书籍的情况。 针对低端机型，去掉了某些动画，交互更加流畅。

关键路径	1.3.0（ms）	1.3.1(ms)	优化减少
启动时间	1929.48	1839.65	7%
打开发现	85.02	53.44	33%
打开书籍详情	18.1	9.21	49%
打开书籍	273.7	226.75	17%
翻页	8.72	7.47	14%
打开书城分类	130.89	116.82	15%
打开书城推荐	58.71	40.15	31%

总结

通过上述介绍，我们可以看出，性能问题普遍存在，无可避免，与其花费大量时间，查找线上版本的性能问题，
不如提高整体团队成员性能优化意识，借助性能查找工具，将性能问题尽早暴露在开发阶段，达到预防为主的效果。

问题

(0)想问下你们 DB 操作这部分涉及到多线程读写是怎么处理的?

我们用了FMDB，它已经处理了这种情况

MDB 的解决方案，我理解是放到一个队列里，虽然可以解决多线程读写的问题，但是队列的处理还是会阻塞住来自不同线程的请求，对么?

是的。我们一直也是读写都在同一条队列，其实并没有太明显的性能瓶颈，
因为在sqlite之上我们还有一层基于model的cache

(1)除了 sqlite 语句的优化之外，db 这部分还有没有其他方面的优化工作?

除了 sqlite 语句的优化之外，db 这部分还有没有其他方面的优化工作
我们有一个自己的DB框架，是ORM的，做了很多优化的工作，最近刚开源，大家可以看看

(2) 请问你们选择用sqlite的考量是什么, 有没有考虑过使用其他的db如realm?

选择sqlite是历史原因，因为我们已经基于sqlite做了一个高性能的DB框架，而且也是经过QQMail App验证的。realm有考虑过，但是因为不是开源，所以估计不用采用

(3)合理的使用线程，多线程之间的同步这块儿有什么方案或建议?

这里我们也并没有什么通用的方案，原则是尽量避免使用多线程。一定要用的时候，也是根据业务谨慎选择

(4) 我把我的问题编号为4吧，方便对应。业务场景里会不会涉及到有 读操作 依赖写操作完成的情况，否则会出现读操作的数据不准确的情况。FMDB 感觉不能很好的解决这个问题。

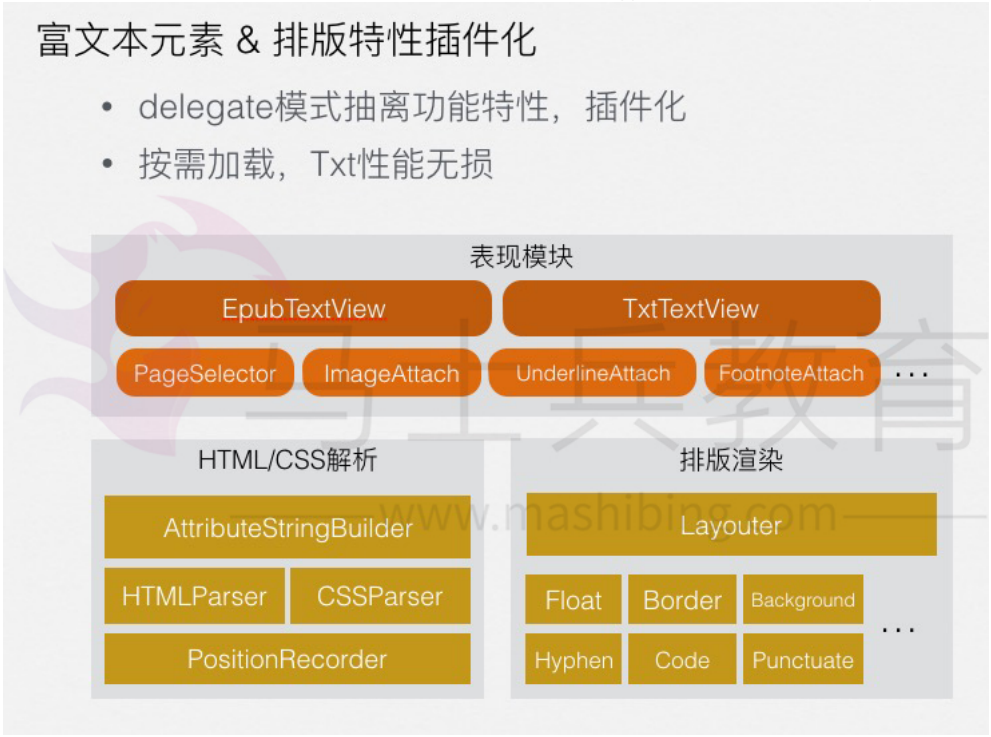
读操作 依赖写操作完成，这种场景一定会有的。但是这种问题应该是业务流程自己控制，而不是DB应该考虑的事情，DB性一能保证的就是按照业务提交的顺序，顺序执行

数据库的记录-一般是在什么级别百千有没有尝试去做过一些压测数据量达到多少的时候会遇到瓶颈">(5)能不能问下 微信读书的数据库的记录 一般是在什么级别，百、千？有没有尝试去做过一些压测，数据量达到多少的时候会遇到瓶颈?

微信读书的数据库记录并不是很大，单表记录最多可能也就10w的数据级别。QQ邮箱的mailApp跟我们是用的同一套，但是数量级别远大于微信读书。目前发现的瓶颈是DB文件达到200M以上时，sqlite的性能会明显受到影响，不过具体原因还在调查中。做过一些压力测试，用来对比CoreData，但是具体数据我这里暂时没有

(6) 微信读书这么成功，方便说下她的架构吗？我觉得架构好才是她可优化的第一步

哈哈，现在还远谈不上成功啦。架构要用图来画才方便看，我暂时还没总结整个app的架构, 可以看看关于阅读器epub渲染的一个架构



(7) 你们的 db 是只有一个文件，还是尝试分文件存储的?

看业务需求，目前是多个DB文件

造成性能瓶颈。

五步定位性能瓶颈

1、着手在*测试*前：理清数据流向，数据流程分解

通过绘制数据流向图，以便清晰的列出所有可能出现瓶颈的位置，避免在分析过程中遗漏可能的瓶颈点。

系统架构分解——水池模型

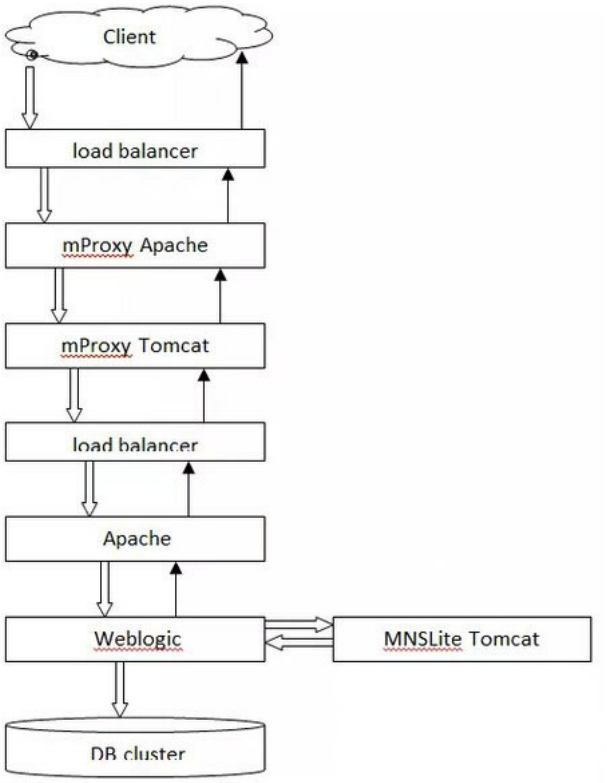
要查找瓶颈，首先要对系统的架构有详细的了解，清楚知道所有可能成为瓶颈的位置。只有这样才能在遇到问题是合理的设计**测试用例**，对流程的各个步骤进行逐一排查。

举个例子，家里厨房的水池下水堵了，我们要找原因，首先得知道水池的下水道都有哪些部分：

简单的看，可以把下水道分解为水漏、上连接管、回水弯、下连接管最后接入地漏。再查找堵塞位置时，我们就可以将水直接导入回水弯，排除水漏和上连接管道堵塞的可能。

应用在测试中，我们也可以利用直接向应用中间件发请求，来排除Web代理层的瓶颈。

通过绘制流向图，可以更清晰的展现系统中数据流向，帮助我们在定位瓶颈的过程始终能迅速的分析预计到下一个可能的瓶颈位置。



如上图，就是在play一个Mobile game时的数据流向图。

2、最直观的指征：检索日志中的异常

日志是系统异常的最直接反映，通过客户端(负载工具端)、服务器端的日志，可以迅速确定瓶颈可能存在的方向。一些在大用户量大并发情况下的功能问题，也会在错误日志中体现。

在**性能测试**过程中，一般情况是不把全部日志打开的，而是尽量保持与生产环境的设置相同，生产环境开启什么样的日志级别，在性能测试环境中也应该开启同样的级别。

但是往往在生产环境出于性能考虑，并不会把日志级别开的非常高，所以在发现系统存在性能问题时，我们可以适当调高日志级别，以便获得更多的信息。

在日志中，我们可以由一些关键字直接推断出系统的问题所在，比如：

· Too many open files

Linux下存在句柄数限制，系统的默认值较小，在测试前应该优化，另外还要怀疑是否程序存在打开句柄却在某些情况下没有关闭。

· OutOfMemoryError/Cannot allocate memory

Java环境的虚拟内存异常，往往需要关注是否有溢出。

· SQLException

数据库语句执行异常，一般日志中还会有数据库返回的信息。

· Connection closed/connection refused

连接被关闭被拒绝，一般是连接数限制不能承担当前的压力。

3、最底层的反映：分析硬件资源占用

硬件资源也是系统性能达到瓶颈点的重要指征，如果没有在日志中找到异常，那么通过监控硬件资源消耗，往往可以发现系统的资源瓶颈。

3.1 CPU占用率

CPU的高占用，并不一定表示有问题，因为实现最优性能的一方面就是充分发挥当前的硬件资源能力。

但是如果CPU长期出于满负荷，就很值得我们关注，至少说明在大多数情况下，系统已经是在耗用最大的计算能力进行计算，运算能力已经成为瓶颈。

另外还要注意CPU是消耗在User还是Sys还是Wait, 如果是Wait，还要观察其他硬件资源，查看CPU是在等待什么。

3.2 内存占用

内存在性能测试中是被重点关注的指标，因为它是反映重大缺陷——内存泄露的最直接指标，但是我们应该注意到，在JAVA框架中的内存泄漏是发生在虚拟内存中的。

观察内存/虚拟内存的占用情况，尤其是在压力消失后的内存占用恢复情况，是比较直接的判断内存泄漏的依据。

如果观察到内存的使用情况，在每次Full GC后，占用的内存都没能恢复到原来的水平，如果在压力撤除一段时间后，内存依旧不能恢复，那么十有八九当前系统存在内存泄漏。

3.3 磁盘I/O

通常情况下，磁盘是计算机中速度最慢的一个子系统，因此很多情况中，磁盘I/O会成为系统的瓶颈。实际上在设计高性能系统的时候，会把避免磁盘I/O作为一个首要准则。

虽然当前的技术发展让存储系统的读写速度不断提升，但高昂的成本使得大多数情况下，高速存储会使用在数据库或文件服务器上，而不会使用在应用服务器中。所以在我们进行性能测试时，要更多的注意应用服务器的磁盘使用情况。

3.4 网络I/O

很多时候大家都容易忽略网络对系统的影响，实际上网络带宽在一些情况下也会成为系统的瓶颈。一旦在业务的请求和响应中包含较大的数据传输时，往往会遇到网络瓶颈。因为更多的时候服务器采用的还是以太网卡，1000M网卡在全双工模式下传输速率也只有80M/s，如果响应中包含报表、图片之类的大尺寸数据，很有可能在性能测试中出现网络瓶颈。

还有一点就是不要忽略回环地址传输的影响，比如一些应用访问本地监听的其他服务，都会受到网卡的传输速率限制的影响。

4、软件性能软肋：数据库的监控分析

对于Web系统，超过七成的瓶颈都出现在数据库子系统，因此在进行之前几步不能明确瓶颈位置的时候，应优先进行数据库的监控分析。

Oracle数据库监控工具

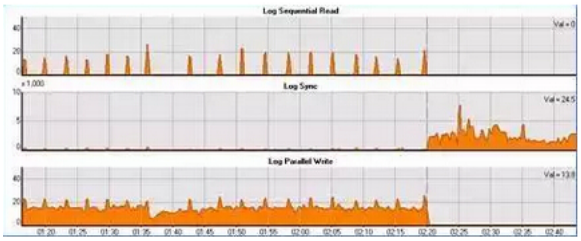
Oracle本身提供了ASH，AWR等Report来帮助进行性能分析，但是对于测试人员来说，掌握这些需要较深入的数据库知识学习，不是一朝一夕可以达成的。而一些第三方提供的工具，通过图形界面，可以更加直观的帮助我们进行Oracle数据库的监控和分析。

Lab128就是国内开发的一块很不错的共享软件，而且它还提供无限期的试用key，可以免费试用。

Oracle中的等待事件

判断Oracle中的瓶颈，了解Oracle中的等待事件Wait event，对于查找瓶颈有很大的帮助。在Oracle中，处理SQL的过程，会产生一系列的等待事件。

有等待事件并不代表数据库存在瓶颈，正常的处理也会有等待事件，但如果发现等待事件激增，或者SQL执行缓慢，这时候等待事件中排名靠前的事件将会直接反映出瓶颈所在。



上图是在测试中的某一时刻，log sync的等待事件突然增高，同时数据库的吞吐率大幅下降，原本正常的SQL执行速度也突然变长。

因为压力并没有突然改变，很有可能是写log的过程出现了问题，或者是在传输过程，或者是在存储子系统。后来经过排查，发现是存储集群的一个存储单元出现故障导致写入速度变慢致使出现大量等待。

5、最后的大杀器：应用服务器监控及代码分析

如果没能在其他位置发现瓶颈，那么软件程序所运行的平台——应用服务器很可能是最大的潜在瓶颈点，进行应用服务器的监控与分析将是我们最后的大杀器。

5.1 常见的软件资源种类

相对于硬件资源，软件资源往往容易被忽略，它不像CPU占用率那么让人更直观的和性能联系起来，但是实际上，软件资源同样限制着软件系统能达到什么样的性能。

软件资源不论是在Web层，应用层还是在数据库层，都可以按“入口”、“内部”、“出口”来划分。对于常见的原因中间件，“入口”就是如HTTP连接池之类，是数据来源方向的相关设置，比如连接数限制，超时时间，连接回收策略等等；“内部”就是处理请求的各项资源，不如线程数，线程调度策略，虚拟内存设置，GC策略等等；“出口”则是向后端交互的各项资源，如数据库连接池的配置。

5.2 应用中间件监控

要了解软件资源是否成为瓶颈，我们就需要监控这些软件资源指标。以JAVA环境为例，Weblogic 本身就有控制台，提供了各种计数器。

Active Execute Threads	Execute Thread Total Count	Execute Thread Idle Count	Queue Length	Pending User Request Count	Completed Request Count
200	200	111	0	0	5063

上图显示的是Execute Threads的计数器，对请求的处理就在这些Thread中进行。

Tomcat也有开源的控制台，常用的像PSI-Probe，提供了Tomcat服务器各项资源的图形化监控。

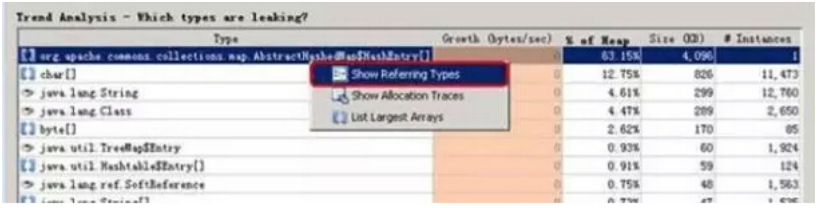
5.3 应用中间件剖析

仅仅监控只能初步判断问题的方向，例如发现ExecuteThreads持续的增加，我们虽然知道这个现象不正常，但是想要确定是程序中的哪个方法导致了当前问题，我还需要其他的工具进行深入剖析。

对于Java程序，最常用的工具有JProfiler，YourKit，他们的原理类似，都是要把一个小插件挂在到应用服务器上，以获取需要的程序运行信息。

而Sun在JDK1.7后版本整合了继承自JRockit的MissionControl，也提供了很强大的分析监控功能，而且开销较小，确实是个不错的选择。

它提供的Mem leak detector可以对对象的创建进行趋势分析，帮你找到最有可能出现泄漏的对象，



再通过展开剖析工具中的invoke tree，找出创建该对象的方法，可以更细致的定位问题的原因。

同时，Call tree 也可以依据CPU时间进行分析，找到在虚拟机中消耗最高的方法。

性能测试如何定位瓶颈

性能测试这种测试方式在发生过程中，其中一个过渡性的工作

1、网络瓶颈，如带宽，流量等形成的网络环境

2、应用服务瓶颈，如中间件的基本配置，CACHE等

3、系统瓶颈，这个比较常用：应用服务器，**数据库**服务器以及客户机的CPU，内存，硬盘等配置

4、数据库瓶颈，以ORACLE为例，SYS中默认的一些参数设置

5、应用程序本身瓶颈，这个是测试过程中最需要去关注的，需要测试人员和开发人员配合执行，然后定位

逐步细化分析，先可以监控一些常见衡量CPU，内存，磁盘的性能指标，进行综合分析，然后根据所测系统具体情况，进行初步问题定位，然后确定更详细的监控指标来分析。

怀疑内存不足时：

方法1：

【监控指标】：Memory Available MBytes ，Memory的Pages/sec， page read/sec， Page Faults/sec

【参考值】：

如果 Page Reads/Sec 比率持续保持为 5，表示可能内存不足。

Page/sec 推荐00-20（如果服务器没有足够的内存处理其工作负荷，此数值将一直很高。如果大于80，表示有问题）。

```
方法2：根据Physical Disk 值分析性能瓶颈
```

【监控指标】：Memory Available MBytes ，Pages read/sec，%Disk Time 和 Avg.Disk Queue Length

【参考值】：%Disk Time建议阈值90%

当内存不足时，有点进程会转移到硬盘上去运行，造成性能急剧下降，而且一个缺少内存的系统常常表现出很高的CPU利用率，因为它需要不断的扫描内存，将内存中的页面移到硬盘上。

怀疑内存泄漏时

【监控指标】：Memory Available MBytes ，Process\Private Bytes和Process\Working Set，PhysicalDisk/%Disk Time

【说明】：

Windows资源监控中，如果Process\Private Bytes计数器和Process\Working Set计数器的值在长时间内持续升高，同时Memory\Available bytes计数器的值持续降低，则很可能存在内存泄漏。内存泄漏应该通过一个长时间的，用来研究分析当所有内存都耗尽时，应用程序反应情况的测试来检验。

CPU分析

【监控指标】：

System %Processor Time CPU，Processor %Processor Time CPU

Processor%user time 和Processor%Privileged Time

system\Processor Queue Length

Context Switches/sec 和%Privileged Time

【参考值】：

System\%Total processor time不持续超过90%，如果服务器专用于SQL Server，可接受的最大上限是80-85% ，合理使用的范围在60%至70%。

Processor %Processor Time小于75%

system\Processor Queue Length值，小于CPU数量的总数+1

CPU瓶颈问题

1、System\%Total processor time如果该值持续超过90%，且伴随处理器阻塞，则说明整个系统面临着处理器方面的瓶颈.

注：在某些多CPU系统中，该数据虽然本身并不大，但CPU之间的负载状况极不均衡，此时也应该视作系统产生了处理器方面的瓶颈.

2、排除内存因素，如果Processor %Processor Time计数器的值比较大，而同时网卡和硬盘的值比较低，那么可以确定CPU 瓶颈。（内存不足时，有点进程会转移到硬盘上去运行，造成性能急剧下降，而且一个缺少内存的系统常常表现出很高的CPU利用率，因为它需要不断的扫描内存，将内存中的页面移到硬盘上。）

```
**造成高CPU使用率的原因：**
```

频繁执行程序，复杂运算操作，消耗CPU严重

数据库查询语句复杂，大量的 where 子句，order by， group by 排序等，CPU容易出现瓶颈

内存不足，IO磁盘问题使得CPU的开销增加

磁盘I/O分析

【监控指标】：PhysicalDisk/%Disk time，PhysicalDisk/%Idle Time，Physical Disk\ Avg.Disk Queue Length， Disk sec/Transfer

【参考值】：%Disk Time建议阈值90%

Windows资源监控中，如果% Disk Time和Avg.Disk Queue Length的值很高，而Page Reads/sec页面读取操作速率很低，则可能存在磁盘瓶径。

Processor%Privileged Time该参数值一直很高，且如果在 Physical Disk 计数器中，只有%Disk time 比较大，其他值都比较适中，硬盘可能会是瓶颈。若几个值都比较大，那么硬盘不是瓶颈。若数值持续超过80%，则可能是内存泄露。如果 Physical Disk 计数器的值很高时该计数器的值（Processor%Privileged Time）也一直很高， 则考虑使用速度更快或效率更高的磁盘子系统。

Disk sec/Transfer 一般来说，该数值小于15ms为最好，介于15-30ms之间为良好，30-60ms之间为可以接受，超过60ms则需要考虑更换硬盘或是硬盘的RAID方式了.

Average Transaciton Response Time（事务平均响应时间）随着测试时间的变化，系统处理事务的速度开始逐渐变慢，这说明应用系统随着投产时间的变化，整体性能将会有下降的趋势

Transactions per Second（每秒通过事务数/TPS）当压力加大时，点击率/TPS曲线如果变化缓慢或者有平坦的趋势，很有可能是服务器开始出现瓶颈

Hits per Second（每秒点击次数）通过对查看"每秒点击次数"，可以判断系统是否稳定。系统点击率下降通常表明服务器的响应速度在变慢，需进一步分析，发现系统瓶颈所在。

Throughput（吞吐率）可以依据服务器的吞吐量来评估虚拟用户产生的负载量，以及看出服务器在流量方面的处理能力以及是否存在瓶颈。

Connections（连接数）当连接数到达稳定状态而事务响应时间迅速增大时，添加连接可以使性能得到极大提高（事务响应时间将降低）

Time to First Buffer Breakdown（Over Time）（第一次缓冲时间细分（随时间变化））可以使用该图确定场景或会话步骤运行期间服务器或网络出现问题的时间。

```
**碰到过的性能问题：**

    1．在高并发的情况下，产生的处理失败（比如：数据库连接池过低，服务器连接数超过上限，数据库锁控制考虑不足等）

      2．内存泄露（比如：在长时间运行下，内存没有正常释放，发生宕机等）

                  3．CPU使用偏离（比如：高并发导致CPU使用率过高）

                    4．日志打印过多，服务器无硬盘空间
```

如何定位这些性能问题：

```
    1．查看系统日志，日志是定位问题的不二法宝，如果日志记录的全面，很容易通过日志发现问题。
```

比如，系统宕机时，系统日志打印了某方法执行时抛出out of memory的错误，我们就可以顺藤摸瓜，很快定位到导致内存溢出的问题在哪里。

```
    2．利用性能监控工具，比如：JAVA开发B/S结构的项目，可以通过JDK自带的Jconsole，或者JProfiler，来监控服务器性能，Jconsole可以远程监控服务器的CPU，内存，线程等状态，并绘制变化曲线图。
```

利用Spotlight可以监控数据库使用情况。

我们需要关注的性能点有：CPU负载，内存使用率，网络I/O等

```
    3．工具和日志只是手段，除此之外，还需要设计合理的性能测试场景
```

具体场景有：性能测试，负载测试，压力测试，稳定性测试，浪涌测试等

好的测试场景，能更加快速的发现瓶颈，定位瓶颈

```
    4．了解系统参数配置，可以进行后期的性能调优

最后要说的是：做性能测试的时候，我们一定要确保瓶颈不要发生在我们自己的测试脚本和测试工具上。
```

性能测试如何定位瓶颈

性能测试这种测试方式在发生过程中，其中一个过渡性的工作

1、网络瓶颈，如带宽，流量等形成的网络环境

2、应用服务瓶颈，如中间件的基本配置，CACHE等

3、系统瓶颈，这个比较常用：应用服务器，**数据库**服务器以及客户机的CPU，内存，硬盘等配置

4、数据库瓶颈，以ORACLE为例，SYS中默认的一些参数设置

5、应用程序本身瓶颈，这个是测试过程中最需要去关注的，需要测试人员和开发人员配合执行，然后定位

逐步细化分析，先可以监控一些常见衡量CPU，内存，磁盘的性能指标，进行综合分析，然后根据所测系统具体情况，进行初步问题定位，然后确定更详细的监控指标来分析。

怀疑内存不足时：

方法1：

【监控指标】：Memory Available MBytes ，Memory的Pages/sec， page read/sec， Page Faults/sec

【参考值】：

如果 Page Reads/Sec 比率持续保持为 5，表示可能内存不足。

Page/sec 推荐00-20（如果服务器没有足够的内存处理其工作负荷，此数值将一直很高。如果大于80，表示有问题）。

```
方法2：根据Physical Disk 值分析性能瓶颈
```

【监控指标】：Memory Available MBytes ，Pages read/sec，%Disk Time 和 Avg.Disk Queue Length

【参考值】：%Disk Time建议阈值90%

当内存不足时，有点进程会转移到硬盘上去运行，造成性能急剧下降，而且一个缺少内存的系统常常表现出很高的CPU利用率，因为它需要不断的扫描内存，将内存中的页面移到硬盘上。

怀疑内存泄漏时

【监控指标】：Memory Available MBytes ，Process\Private Bytes和Process\Working Set，PhysicalDisk/%Disk Time

【说明】：

Windows资源监控中，如果Process\Private Bytes计数器和Process\Working Set计数器的值在长时间内持续升高，同时Memory\Available bytes计数器的值持续降低，则很可能存在内存泄漏。内存泄漏应该通过一个长时间的，用来研究分析当所有内存都耗尽时，应用程序反应情况的测试来检验。

CPU分析

【监控指标】：

System %Processor Time CPU，Processor %Processor Time CPU

Processor%user time 和Processor%Privileged Time

system\Processor Queue Length

Context Switches/sec 和%Privileged Time

【参考值】：

System\%Total processor time不持续超过90%，如果服务器专用于SQL Server，可接受的最大上限是80-85% ，合理使用的范围在60%至70%。

Processor %Processor Time小于75%

system\Processor Queue Length值，小于CPU数量的总数+1

CPU瓶颈问题

1、System\%Total processor time如果该值持续超过90%，且伴随处理器阻塞，则说明整个系统面临着处理器方面的瓶颈.

注：在某些多CPU系统中，该数据虽然本身并不大，但CPU之间的负载状况极不均衡，此时也应该视作系统产生了处理器方面的瓶颈.

2、排除内存因素，如果Processor %Processor Time计数器的值比较大，而同时网卡和硬盘的值比较低，那么可以确定CPU 瓶颈。（内存不足时，有点进程会转移到硬盘上去运行，造成性能急剧下降，而且一个缺少内存的系统常常表现出很高的CPU利用率，因为它需要不断的扫描内存，将内存中的页面移到硬盘上。）

```
**造成高CPU使用率的原因：**
```

频繁执行程序，复杂运算操作，消耗CPU严重

数据库查询语句复杂，大量的 where 子句，order by， group by 排序等，CPU容易出现瓶颈

内存不足，IO磁盘问题使得CPU的开销增加

磁盘I/O分析

【监控指标】：PhysicalDisk/%Disk time，PhysicalDisk/%Idle Time，Physical Disk\ Avg.Disk Queue Length， Disk sec/Transfer

【参考值】：%Disk Time建议阈值90%

Windows资源监控中，如果% Disk Time和Avg.Disk Queue Length的值很高，而Page Reads/sec页面读取操作速率很低，则可能存在磁盘瓶径。

Processor%Privileged Time该参数值一直很高，且如果在 Physical Disk 计数器中，只有%Disk time 比较大，其他值都比较适中，硬盘可能会是瓶颈。若几个值都比较大，那么硬盘不是瓶颈。若数值持续超过80%，则可能是内存泄露。如果 Physical Disk 计数器的值很高时该计数器的值（Processor%Privileged Time）也一直很高， 则考虑使用速度更快或效率更高的磁盘子系统。

Disk sec/Transfer 一般来说，该数值小于15ms为最好，介于15-30ms之间为良好，30-60ms之间为可以接受，超过60ms则需要考虑更换硬盘或是硬盘的RAID方式了.

Average Transaciton Response Time（事务平均响应时间）随着测试时间的变化，系统处理事务的速度开始逐渐变慢，这说明应用系统随着投产时间的变化，整体性能将会有下降的趋势

Transactions per Second（每秒通过事务数/TPS）当压力加大时，点击率/TPS曲线如果变化缓慢或者有平坦的趋势，很有可能是服务器开始出现瓶颈

Hits per Second（每秒点击次数）通过对查看“每秒点击次数”，可以判断系统是否稳定。系统点击率下降通常表明服务器的响应速度在变慢，需进一步分析，发现系统瓶颈所在。

Throughput（吞吐率）可以依据服务器的吞吐量来评估虚拟用户产生的负载量，以及看出服务器在流量方面的处理能力以及是否存在瓶颈。

Connections（连接数）当连接数到达稳定状态而事务响应时间迅速增大时，添加连接可以使性能得到极大提高（事务响应时间将降低）

Time to First Buffer Breakdown（Over Time）（第一次缓冲时间细分（随时间变化））可以使用该图确定场景或会话步骤运行期间服务器或网络出现问题的时间。

```
**碰到过的性能问题：**

    1．在高并发的情况下，产生的处理失败（比如：数据库连接池过低，服务器连接数超过上限，数据库锁控制考虑不足等）

    2．内存泄露（比如：在长时间运行下，内存没有正常释放，发生宕机等）

            3．CPU使用偏离（比如：高并发导致CPU使用率过高）

              4．日志打印过多，服务器无硬盘空间
```

如何定位这些性能问题：

```
    1．查看系统日志，日志是定位问题的不二法宝，如果日志记录的全面，很容易通过日志发现问题。
```

比如，系统宕机时，系统日志打印了某方法执行时抛出out of memory的错误，我们就可以顺藤摸瓜，很快定位到导致内存溢出的问题在哪里。

2．利用性能监控工具，比如：JAVA开发B/S结构的项目，可以通过JDK自带的Jconsole，或者JProfiler，来监控服务器性能，Jconsole可以远程监控服务器的CPU，内存，线程等状态，并绘制变化曲线图。

利用Spotlight可以监控数据库使用情况。

我们需要关注的性能点有：CPU负载，内存使用率，网络I/O等

3．工具和日志只是手段，除此之外，还需要设计合理的性能测试场景

具体场景有：性能测试，负载测试，压力测试，稳定性测试，浪涌测试等

好的测试场景，能更加快速的发现瓶颈，定位瓶颈

4．了解系统参数配置，可以进行后期的性能调优

最后要说的是：做性能测试的时候，我们一定要确保瓶颈不要发生在我们自己的测试脚本和测试工具上。

你的项目中使用过缓存机制吗？有没用用户非本地缓存

使用缓存的9大误区（上）

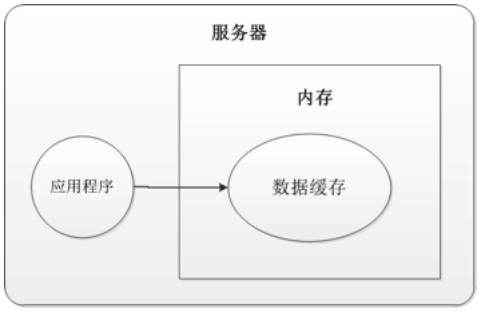
如果说要对一个站点或者应用程序经常优化，可以说缓存的使用是最快也是效果最明显的方式。一般而言，我们会把一些常用的，或者需要花费大量的资源或时间而产生的数据缓存起来，使得后续的使用更加快速。

如果真要细说缓存的好处，还真是不少，但是在实际的应用中，很多时候使用缓存的时候，总是那么的不尽人意。换句话说，假设本来采用缓存，可以使得性能提升为100（这里的数字只是一个计量符号而已，只是为了给大家一个“量”的体会），但是很多时候，提升的效果只有80,70，或者更少，甚至还会导致性能严重的下降，这个现象在使用分布式缓存的时候尤为突出。

在本篇文章中，我们将为大家讲述导致以上问题的9大症结，并且给出相对应的解决方案。文章以.NET为例子进行代码的演示，对于来及其他技术平台的朋友也是有参考价值的，只要替换相对应的代码就行了！

为了使得后文的阐述更加的方便，也使得文章更为的完整，我们首先来看看缓存的两种形式：本地内存缓存，分布式缓存。

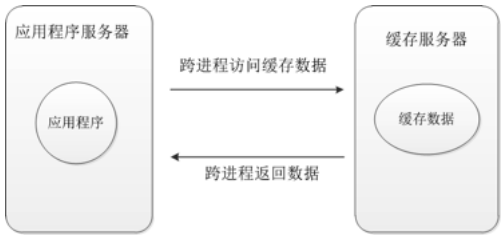
首先对于本地内存缓存，就是把数据缓存在本机的内存中，如下图1所示：



从上图中可以很清楚的看出：

- 应用程序把数据缓存在本机的内存，需要的时候直接去本机内存进行获取。
- 对于.NET的应用而言，在获取缓存中的数据的时候，是通过对象的引用去内存中查找数据对象的，也就说，如果我们通过引用获取了数据对象之后，我们直接修改这个对象，其实我们真正的是在修改处于内存中的那个缓存对象。

对于分布式的缓存，此时因为缓存的数据是放在缓存服务器中的，或者说，此时应用程序需要跨进程的去访问分布式缓存服务器，如图2：



不管缓存服务器在哪里，因为涉及到了跨进程，甚至是跨域访问缓存数据，那么缓存数据在发送到缓存服务器之前就要先被序列化，当要用缓存数据的时候，应用程序服务器接收到了序列化的数据之后，会将之反序列化。序列化与反序列化的过程是非常消耗CPU的操作，很多问题就出现在这上面。

另外，如果我们把获取到的数据，在应用程序中进行了修改，此时缓存服务器中的原先的数据是没有修改的，除非我们再次将数据保存到缓存服务器。请注意：这一点和之前的本地内存缓存是不一样的。

对于缓存中的每一份数据，为了后文的讲述方面，我们称之为“缓存项”。

普及完了这两个概念之后，我们就进入今天的主题：使用缓存常见的9大误区：

1. **太过于依赖.NET****默认的序列化机制****
2. **缓存大对象**
3. **使用缓存机制在线程间进行数据的共享**
4. **认为调用缓存API****之后，数据会被立刻缓存起来****

5. **缓存大量的数据集合，而读取其中一部分**
6. **缓存大量具有图结构的对象导致内存浪费**
7. **缓存应用程序的配置信息**
8. **使用很多不同的键指向相同的缓存项**
9. **没有及时的更新或者删除再缓存中已经过期或者失效的数据**

下面，我们就每一点来具体的看看！

太过于依赖.NET默认的序列化机制

当我们在应用中使用跨进程的缓存机制，例如分布式缓存memcached或者微软的AppFabric，此时数据被缓存在应用程序之外的进程中。每次，当我们要把一些数据缓存起来的时候，缓存的API就会把数据首先序列化为字节的形式，然后把这些字节发送给缓存服务器去保存。同理，当我们在应用中要再次使用缓存的数据的时候，缓存服务器就会将缓存的字节发送给应用程序，而缓存的客户端类库接受到这些字节之后就要进行反序列化的操作了，将之转换为我们需要的数据对象。

另外还有三点需要注意的就是：

- 这个序列化与反序列化的机制都是发生在应用程序服务器上的，而缓存服务器只是负责保存而已。
- .NET中的默认使用的序列化机制不是最优的，因为它要使用反射机制，而反射机制是是非常耗CPU的，特别是当我们缓存了比较复杂的数据对象的时候。

基于这个问题，我们要自己选择一个比较好的序列化方法来尽可能的减少对CPU的使用。常用的方法就是让对象自己来实现ISerializable接口。

首先我们来看看默认的序列化机制是怎么样的。如图3：

```
[Serializable]
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
}
```

然后，我们自己来实现ISerializable接口，如下图4所示：

```
public class Product:ISerializable
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }

    public Product(SerializationInfo info, StreamingContext context)
    {
        Name = info.GetString("Name");
        Price = info.GetDecimal("Price");
        Description = info.GetString("Description");
    }

    #region ISerializable Members

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("Name", Name);
        info.AddValue("Price", Price);
        info.AddValue("Description", Description);
    }

    #endregion

}
```

我们自己实现的方式与.NET默认的序列化机制的最大区别在于：没有使用反射。自己实现的这种方式速度可以是默认机制的上百倍。

可能有人认为没有什么，不就是一个小小的序列化而已，有必要小题大做么？

在开发一个高性能应用（例如网站）而言，从架构，到代码的编写，以及后面的部署，每一个地方都需要优化。一个小问题，例如这个序列化的问题，初看起来不是问题，如果我们站点应用的访问量是百万，千万，甚至更高级别的，而这些访问需要去获取一些公共的缓存的数据，这个之前所谓的小问题就不小了！

下面，我们来看第二个误区。

缓存大对象

有时候，我们想要把一些大对象缓存起来，因为产生一次大对象的代价很大，我们需要产生一次，尽可能的多次使用，从而提升响应。

提到大对象，这里就很有必要对其进行一个比较深入的介绍了。在.NET中，所谓的大对象，就是指的其占用的内存大于了85K的对象，下面通过一个比较将问题说清楚。

如果现在有一个Person类的集合，定义为List，每个Person对象占用1K的内存，如果这个Person集合中包含了100个Person对象实例，那么这个集合是否是大对象呢？

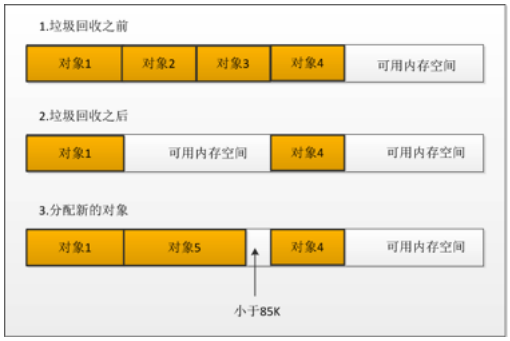
回答是：不是！

因为集合中只是包含的Person对象实例的引用而言，即，在.NET的托管堆上面，这个Person集合分配的内存大小也就是100个引用的大小而言。

然后，对于下面的这个对象，就是大对象了: byte[] data = new byte[87040]（85 * 1024 = 87040）。

说到了这里，那就谈谈，为什么说：产生一次大对象的代价很大。

因为在.NET中，大对象是分配在大对象托管堆上面的（我们简称为“大堆”，当然，还有一个对应的小堆），而这个大堆上面的对象的分配机制和小堆不一样：大堆在分配的时候，总是去需找合适的内存空间，结果就是导致出现内存碎片，导致内存不足！我们用一个图来描述一下，如图5所示：



上图非常明了，在图5中：

- 垃圾回收机制不会在回收对象之后压缩大堆（小堆是压缩的）。

- 分配对象的时候，需要去遍历大堆，去需找合适的空间，遍历是要花成本的。
- 如果某些空间小于85K，那么就不能分配了，只能白白浪费，也导致内存碎片。

讲完了这些之后，我们言归正传，来看看大对象的缓存。

正如之前讲过，将对象缓存和读取的时候是要进行序列化与反序列化的，缓存的对象越大（例如，有1M等），整个过程中就消耗更多的CPU。

对于这样的大对象，要看它使用的是否很频繁，是否是公用的数据对象，还是每个用户都要产生的。因为我们一旦缓存了（特别在分布式缓存中），就需要同时消耗缓存服务器的内存与应用程序服务器的CPU。如果使用的不频繁，建议每次生成！如果是公用的数据，那么建议多多的测试：将生产大对象的成本与缓存它的时候消耗的内存和CPU的成本进行比较，选择成本小的！如果是每个用户都要产生的，看看是否可以分解，如果实在不能分解，那么缓存，但是及时的释放！

使用缓存机制在线程间进行数据的共享

当数据放在缓存中的时候，我们程序的多个线程都可以访问这个公共的区域。多个线程在访问缓存数据的时候，会产生一些竞争，这也是多线程中常常发生的问题。

下面我们分别从本地内存缓存与分布式缓存两个方面介绍竞争的带来的问题。

看下面的一段代码：

```
public void Run()
{
    myCache["cacheItem"] = 0;

    var thread1 = new Thread(new ThreadStart(() =>
    {
        var item = myCache["cacheItem"]; //
        item++;
        myCache["cacheItem"] = item;
    }));

    var thread2 = new Thread(new ThreadStart(() =>
    {
        var item = myCache["cacheItem"];
        item++;
        myCache["cacheItem"] = item;
    }));

    var thread3 = new Thread(new ThreadStart(() =>
    {
        var item = myCache["cacheItem"];
        item++;
        myCache["SomeItem"] = item;
    }));

    thread1.Start();
    thread2.Start();
    thread3.Start();
}
```

对于本地内存缓存，对于上面的代码，当这个三个线程运行起来之后，在线程1中，item的值很多时候可能为1，线程2可能是2，线程3可能是3。当然，这不一定！只是大多数情况下的可能值！

如果是对于分布式缓存，就不好说了！因为数据的修改不是立刻发生在本机的内存中的，而是经过了一个跨进程的过程。

有一些缓存模块已经实现了加锁的方式来解决这个问题，例如AppFabric。大家在修改缓存数据的时候要特别注意这一点。

认为调用缓存API*之后，数据会被立刻缓存起来

有时候，当我们调用了缓存的API之后，我们就会认为：数据已经被换成了，之后就可以直接读取缓存中的数据。尽管情况很多时候如此，但是不是绝对的！很多的问题就是这样产生的！

我们通过一个例子来讲解。

例如，对于一个ASP.NET 应用而言，如果我们在一个按钮的Click事件中调用了缓存API，然后在页面呈现的时候，就去读取缓存，代码如下：

```
private void SomeButton_Clicked(object sender, EventArgs e)
{
    myCache["SomeItem"] = someItem;
}

private void OnPreRender()
{
    var someItem = myCache["SomeItem"];
    Render(someItem);
}
```

上面的代码照道理来说是对的，但是会发生问题。按钮点击之后回传页面，然后呈现页面的时候显示数据，流程没有问题。但是没有考虑到这样一个问题：如果服务器的内存紧张，而导致进行服务器内存的回收，那么很有可能缓存的数据就没有了！

这里有朋友就要说了：内存回收这么快？

这主要看我们的一些设置和处理。

一般而言，缓存机制都是会设置绝对过期时间与相对过期时间，二者的区别，大家应很清楚，我这里不多说。对于上面的代码而言，如果我们设置的是绝对过期时间，假设1分钟，如果页面处理的非常慢，时间超过了1分钟，那么等到呈现的时候，可能缓存中的数据已经没有了！

有时候，即使我们在第一行代码中缓存了数据，那么也许在第三行代码中，我们去缓存读取数据的时候，就已经没有了。这或许是因为在服务器内存压力很大的，缓存机制将最少访问的数据直接清掉。或者服务器CPU很忙，网络也不好，导致数据没有被即使的序列化保存到缓存服务器中。

另外，对于ASP.NET而言，如果使用了本地内存缓存，那么，还涉及到IIS的配置问题（对缓存内存的限制），我们有机会专门为大家分享这方面的知识。

所以，每次在使用缓存数据的时候，要判断是否存在，不然，会有很多的“找不到对象”的错误，产生一些我们认为的“奇怪而又合理的现象”。

使用缓存的9大误区（下）

本篇文章在上篇的基础上继续讨论了使用缓存的几个误区，包括：缓存大量的数据集合，而读取其中一部分；缓存大量具有图结构的对象导致内存浪费；缓存应用程序的配置信息；使用很多不同的键指向相同的缓存项；没有及时的更新或者删除再缓存中已经过期或者失效的数据。

缓存大量的数据集合，而读取其中一部分

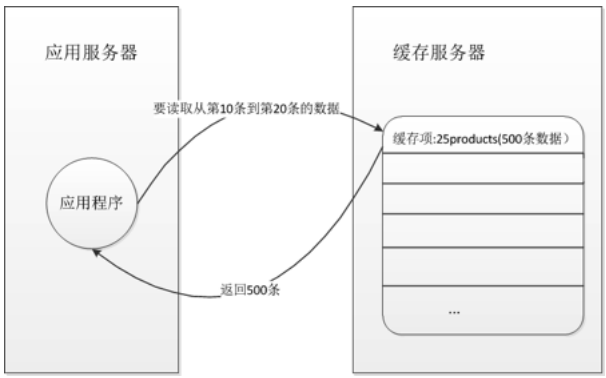
在很多时候，我们往往会缓存一个对象的集合，但是，我们在读取的时候，只是每次读取其中一部分。 我们举个例子来说明这个问题（例子可能不是很恰当，但是足以说明问题）。

在购物站点中，常见的操作就是查询一些产品的信息，这个时候，如果用户输入了“25寸电视机”，然后查找相关的产品。这个时候，在后台，我们可以查询数据库，找到几百条这样的数据，然后，我们将这几百条数据作为一个缓存项缓存起来，代码的代码如下：

```
myCache["25products"] = products;
```

同时，我们对找出的产品进行分页的显示，每次展示10条。其实在每次分页的时候，我们都是根据缓存的键去获取数据，然后选择下一个10条数据，然后显示。

如果是使用本地内存缓存，那么这可能不是什么问题，如果是采用分布式缓存，问题就来了。下图可以清楚的说明这个过程，如图所示：



相信大家看完这个图，然后结合之前的讲述应该很清楚了问题所在了：每次都按照缓存键获取全部数据，然后在应用服务器那里反序列化全部数据，但是只是取其中10条。

这里可以将数据集合再次拆分，分为例如25-0-10-products，25-11-20-products等的缓存项，如下图所示：

当然，查询和缓存的方式有很多，拆分的方式也有很多，这里这是给出一些常见的问题！

缓存大量具有图结构的对象导致内存浪费

为了更好的说明这个问题，我们首先看到下面的一个类结构图，如图：



如果我们要把一些Customer数据缓存起来，这里就可以可能出现两个问题：

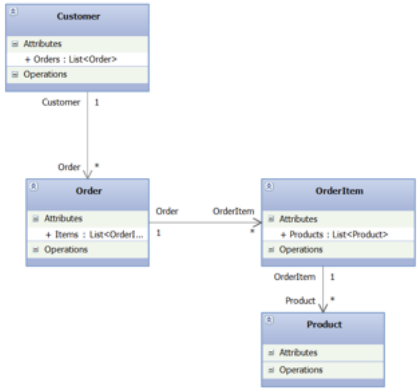
1. 由于使用.NET的默认序列化机制，或者没有适当的加入相应Attribute（属性），使得缓存了一些原本不需要缓存的数据。
2. 将Customer缓存的时候，同时，为了更快的获取Customer的Order信息，将Order信息缓存在了另外一个缓存项中，导致同一份数据被缓存两次。

下面，我们就分别来看看这两个问题。

首先看到第一个。如果我们使用分布式缓存来缓存一些Customer的信息的时候，如果我们没有自己重新Customer的序列化机制，而是采用的默认的，那么序列化机制在序列化Customer的时候，会将Customer所引用的对象也序列化，然后在序列化被序列化对象中的其他引用对象，最后的结果就是：Customer被序列化，Customer的Order信息被序列化，Order引用的OrderItem被序列化，最后OrderItem引用的Product也会序列化。

整个对象图全部被序列化了，如果这种情况是我们想要的，那么没有问题；如果不是的，那么，我们就浪费了很多的资源了，解决的方法有两个：第一，自己实现序列化，自己完全控制哪些对象需要序列化，我们前面已经讲过了；第二，如果使用默认的序列化机制，那么在不要需要序列化的对象上面加上[NonSerialized]标记。

下面，我们看到第二个问题。这个问题主要是由于第一个问题引起的：原本在缓存Customer的时候，已经将Customer的其他信息，例如Order，Product已经缓存了。但是很多的技术人员不清楚这一点，然后又把Customer的Order信息去缓存在其他的缓存项，使用的使用就根据Customer的标识，例如ID去缓存中获取Order信息，如下代码所示：



解决这个问题的方法也比较明显，参看第一个问题的解决方案就可以了！

缓存应用程序的配置信息

因为缓存是有一套数据失效检测周期的（之前说过，要么是固定时间失效，要么是相对时间失效），所以，很多的技术人员喜欢把一些动态变化的信息保存在缓存中，以充分利用缓存机制的这种特性，其中，缓存程序的配置信息就是其中一个例子。

因为在应用的中的一些配置，可能会发生变化，最简单的就是数据库连接字符串了，如下代码：

```
var customer = GetCustomerByName("agilesharp");
var customer.Orders = GetOrdersByCustomer(customer.ID);

cache.Add("Customer", customer);
cache.Add("Orders", orders);
```

当这样设置之后，每隔一段时间缓存失效之后，就去重新读取配置文件，这时候，可能此时的配置就和之前不一样了，并且其他的地方都可以读取缓存从而进行更新，特别是在多台服务器上面部署同一个站点的时候，有时候，我们没有及时的去修改每个服务器上面的站点的配置文件里面的信息，这个时候如何使用分布式缓存缓存配置信息，只要更新一个站点的配置文件，其他站点就全部修改了，技术人员皆大欢喜。OK，这确实看起来是个不错的方法（在必要的时候可以采用一下），但是，不是所有的配置信息都要保持一样的，而且还要考虑怎样一个情况：如果缓存服务器出了问题，宕机了，那么我们所有使用这个配置信息的站点可能都会出问题。

建议对于这些配置文件的信息，采用监控的机制，例如文件监控，每次文件发生变化，就重新加载配置信息。

使用很多不同的键指向相同的缓存项

我们有时候会遇到这样的一个情况：我们把一个对象缓存起来，用一个键作为缓存键来获取这个数据，之后，我们又通过一个索引作为缓存键来获取这个数据，如下代码所示：

```
var connectionString = cache.Get("ConnectionString");
```

我们之所以这样写，主要因为我们会以多种方式来从缓存中读取数据，例如在进行循环遍历的时候，需要通过索引来获取数据，例如index++等，而有些情况，我们可能需要通过其他的方式，例如，产品名来获取产品的信息。

如果遇到这样的情况，那么就建议将这些多个键组合起来，形成如下的形式：

```
var product = GetProduct();
cache["prod"] = product;

//...

cache[1] = product;
```

另外一个常见的问题就是：相同的数据被缓存在不同的缓存项中，例如，如果用户查询尺寸为36寸的彩电，那么可能有可能一个编号为100的电视产品就在结果中，此时，我们将结果缓存。另外，用户在查找一个生产厂家为TCL的电视，如果编号为100的电视产品又出现在结果中，我们把结果又缓存在另外一个缓存项中。这个时候，很显然，出现了内存的浪费。

对于这样的情况，之前笔者采用的方法就是，在缓存中创建了一个索引列表，如图所示：

```
var product = GetProduct();
cache["prod_1"] = product;

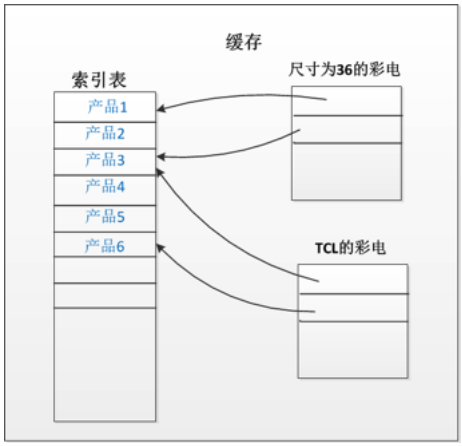
//...

var index = 1;
var cacheProduct = cache["prod" + index];
```

当然，这其中有很多的细节和问题需要解决，这里就不一一述说，要看各自的应用和情况而定！ 也非常欢迎大家提供更好的方法。

没有及时的更新或者删除再缓存中已经过期或者失效的数据

这种情况应该是使用缓存最常见的问题，例如，如果我们现在获取了一个Customer的所有没有处理的订单的信息，然后缓存起来，类似的代码如下：



之后，用户的一个订单被处理了，但是缓存还没有更新，那么这个时候，缓存中的数据就已经有问题！当然，我这里只是列举的最简单的场景，大家可以联想自己应用中的其他产品，很有可能会出现缓存中的数据和实际数据库中的不一样。

现在很多的时候，我们已经容忍了这种短时间的不一致的情况。其实对于这种情况，没有非常完美的解决方案，如果要做，倒是可以实现，例如每次修改或者删除一个数据，就去遍历缓存中的所有数据，然后进行操作，但是这样往往得不偿失。另外一个折中的方法就是，判断数据的变化周期，然后尽可能的将缓存的时间变短一点。

Tomcat 调优及 JVM 参数优化

Tomcat 的缺省配置是不能稳定长期运行的，也就是不适合生产环境，它会死机，让你不断重新启动，甚至在午夜时分唤醒你。对于操作系统优化来说，是尽可能的增大可使用的内存容量、提高CPU 的频率，保证文件系统的读写速率等。经过压力测试验证，在并发连接很多的情况下，CPU 的处理能力越强，系统运行速度越快。

Tomcat 的优化不像其它软件那样，简简单单的修改几个参数就可以了，它的优化主要有三方面，分为系统优化，Tomcat 本身的优化，Java 虚拟机（JVM）调优。系统优化就不在介绍了，接下来就详细的介绍一下 Tomcat 本身与 JVM 优化，以 Tomcat 7 为例。

一、Tomcat 本身优化

Tomcat 的自身参数的优化，这块很像 ApacheHttp Server。修改一下 xml 配置文件中的参数，调整最大连接数，超时等。此外，我们安装 Tomcat 是，优化就已经开始了。

1、工作方式选择

为了提升性能，首先就要对代码进行动静分离，让 Tomcat 只负责 jsp 文件的解析工作。如采用 Apache 和 Tomcat 的整合方式，他们之间的连接方案有三种选择，JK、http_proxy 和 ajp_proxy。相对于 JK 的连接方式，后两种在配置上比较简单的，灵活性方面也一点都不逊色。但就稳定性而言不像JK 这样久经考验，所以建议采用 JK 的连接方式。

2、Connector 连接器的配置

之前文件介绍过的 Tomcat 连接器的三种方式： bio、nio 和 apr，三种方式性能差别很大，apr 的性能最优， bio 的性能最差。而 Tomcat 7 使用的 Connector 默认就启用的 Apr 协议，但需要系统安装 Apr 库，否则就会使用 bio 方式。

3、配置文件优化

配置文件优化其实就是对 server.xml 优化，可以提大大提高 Tomcat 的处理请求的能力，下面我们来看 Tomcat 容器内的优化。

默认配置下，Tomcat 会为每个连接器创建一个绑定的线程池（最大线程数 200），服务启动时，默认创建了 5 个空闲线程随时等待用户请求。

首先，打开 ${TOMCAT_HOME}/conf/server.xml，搜索【<Executor name="tomcatThreadPool"】，开启并调整为

```
  ` `<` `Executor` `name` `=` `"tomcatThreadPool"` `namePrefix` `=` `"catalina-exec-"``    ` `maxThreads` `=` `"500"` `minSpareThreads` `=` `"20"` `maxSpareThreads` `=` `"50"` `maxIdleTime` `=` `"60000"` `/>
```

注意， Tomcat 7 在开启线程池前，一定要安装好 Apr 库，并可以启用，否则会有错误报出，shutdown.sh 脚本无法关闭进程。

然后，修改<Connector ...>节点，增加 executor 属性，搜索【port="8080"】，调整为

```
  ` `<` `Connector` `executor` `=` `"tomcatThreadPool"``        ` `port` `=` `"8080"` `protocol` `=` `"HTTP/1.1"``        ` `URIEncoding` `=` `"UTF-8"``        ` `connectionTimeout` `=` `"30000"``        ` `enableLookups` `=` `"false"``        ` `disableUploadTimeout` `=` `"false"``        ` `connectionUploadTimeout` `=` `"150000"``        ` `acceptCount` `=` `"300"``        ` `keepAliveTimeout` `=` `"120000"``        ` `maxKeepAliveRequests` `=` `"1"``        ` `compression` `=` `"on"``        ` `compressionMinSize` `=` `"2048"``        ` `compressableMimeType` `=` `"text/html,text/xml,text/javascript,text/css,text/plain,image/gif,image/jpg,image/png"` `        ` `redirectPort` `=` `"8443"` `/>
```

maxThreads :Tomcat 使用线程来处理接收的每个请求，这个值表示 Tomcat 可创建的最大的线程数，默认值是 200

minSpareThreads：最小空闲线程数，Tomcat 启动时的初始化的线程数，表示即使没有人使用也开这么多空线程等待，默认值是 10。

maxSpareThreads：最大备用线程数，一旦创建的线程超过这个值，Tomcat 就会关闭不再需要的 socket 线程。

上边配置的参数，最大线程 500（一般服务器足以），要根据自己的实际情况合理设置，设置越大会耗费内存和 CPU，因为 CPU 疲于线程上下文切换，没有精力提供请求服务了，最小空闲线程数 20，线程最大空闲时间 60 秒，当然允许的最大线程连接数还受制于操作系统的内核参数设置，设置多大要根据自己的需求与环境。当然线程可以配置在“tomcatThreadPool”中，也可以直接配置在“Connector”中，但不可以重复配置。

URIEncoding：指定 Tomcat 容器的 URL 编码格式，语言编码格式这块倒不如其它 WEB 服务器软件配置方便，需要分别指定。

connnectionTimeout： 网络连接超时，单位：毫秒，设置为 0 表示永不超时，这样设置有隐患的。通常可设置为 30000 毫秒，可根据检测实际情况，适当修改。

enableLookups： 是否反查域名，以返回远程主机的主机名，取值为：true 或 false，如果设置为false，则直接返回IP地址，为了提高处理能力，应设置为 false。

disableUploadTimeout：上传时是否使用超时机制。

connectionUploadTimeout：上传超时时间，毕竟文件上传可能需要消耗更多的时间，这个根据你自己的业务需要自己调，以使Servlet有较长的时间来完成它的执行，需要与上一个参数一起配合使用才会生效。

acceptCount：指定当所有可以使用的处理请求的线程数都被使用时，可传入连接请求的最大队列长度，超过这个数的请求将不予处理，默认为100个。

keepAliveTimeout：长连接最大保持时间（毫秒），表示在下次请求过来之前，Tomcat 保持该连接多久，默认是使用 connectionTimeout 时间，-1 为不限制超时。

maxKeepAliveRequests：表示在服务器关闭之前，该连接最大支持的请求数。超过该请求数的连接也将被关闭，1表示禁用，-1表示不限制个数，默认100个，一般设置在100~200之间。

compression：是否对响应的数据进行 GZIP 压缩，off：表示禁止压缩；on：表示允许压缩（文本将被压缩）、force：表示所有情况下都进行压缩，默认值为off，压缩数据后可以有效的减少页面的大小，一般可以减小1/3左右，节省带宽。

compressionMinSize：表示压缩响应的最小值，只有当响应报文大小大于这个值的时候才会对报文进行压缩，如果开启了压缩功能，默认值就是2048。

compressableMimeType：压缩类型，指定对哪些类型的文件进行数据压缩。

noCompressionUserAgents="gozilla, traviata"： 对于以下的浏览器，不启用压缩。

如果已经对代码进行了动静分离，静态页面和图片等数据就不需要 Tomcat 处理了，那么也就不需要配置在 Tomcat 中配置压缩了。

以上是一些常用的配置参数属性，当然还有好多其它的参数设置，还可以继续深入的优化，HTTP Connector 与 AJP Connector 的参数属性值，可以参考官方文档的详细说明：

https://tomcat.apache.org/tomcat-7.0-doc/config/http.html

https://tomcat.apache.org/tomcat-7.0-doc/config/ajp.html

二、JVM 优化

Tomcat 启动命令行中的优化参数，就是 JVM 的优化 。Tomcat 首先跑在 JVM 之上的，因为它的启动其实也只是一个 java 命令行，首先我们需要对这个 JAVA 的启动命令行进行调优。不管是 YGC 还是 Full GC，GC 过程中都会对导致程序运行中中断，正确的选择不同的 GC 策略，调整 JVM、GC 的参数，可以极大的减少由于 GC 工作，而导致的程序运行中断方面的问题，进而适当的提高 Java 程序的工作效率。但是调整 GC 是以个极为复杂的过程，由于各个程序具备不同的特点，如：web 和 GUI 程序就有很大区别（Web可以适当的停顿，但GUI停顿是客户无法接受的），而且由于跑在各个机器上的配置不同（主要 cup 个数，内存不同），所以使用的 GC 种类也会不同。

1、JVM 参数配置方法

Tomcat 的启动参数位于安装目录 ${JAVA_HOME}/bin目录下，Linux 操作系统就是 catalina.sh 文件。JAVA_OPTS，就是用来设置 JVM 相关运行参数的变量，还可以在 CATALINA_OPTS 变量中设置。关于这 2 个变量，还是多少有些区别的：

JAVA_OPTS：用于当 Java 运行时选项“start”、“stop”或“run”命令执行。

CATALINA_OPTS：用于当 Java 运行时选项“start”或“run”命令执行。

为什么有两个不同的变量？它们之间都有什么区别呢?

首先，在启动 Tomcat 时，任何指定变量的传递方式都是相同的，可以传递到执行“start”或“run”命令中，但只有设定在 JAVA_OPTS 变量里的参数被传递到“stop”命令中。对于 Tomcat 运行过程，可能没什么区别，影响的是结束程序，而不是启动程序。

第二个区别是更微妙，其他应用程序也可以使用 JAVA_OPTS 变量，但只有在 Tomcat 中使用 CATALINA_OPTS 变量。如果你设置环境变量为只使用 Tomcat，最好你会建议使用 CATALINA_OPTS 变量，而如果你设置环境变量使用其它的 Java 应用程序，例如 JBoss，你应该把你的设置放在JAVA_OPTS 变量中。

2、JVM 参数属性

32 位系统下 JVM 对内存的限制：不能突破 2GB ，那么这时你的 Tomcat 要优化，就要讲究点技巧了，而在 64 位操作系统上无论是系统内存还是 JVM 都没有受到 2GB 这样的限制。

针对于 JMX 远程监控也是在这里设置，以下为 64 位系统环境下的配置，内存加入的参数如下：

```
CATALINA_OPTS="``-server ``-Xms6000M ``-Xmx6000M ``-Xss512k ``-XX:NewSize=2250M ``-XX:MaxNewSize=2250M ``-XX:PermSize=128M``-
XX:MaxPermSize=256M ``-XX:+AggressiveOpts ``-XX:+UseBiasedLocking ``-XX:+DisableExplicitGC ``-XX:+UseParNewGC ``-XX:+UseConcMarkSweepGC ``-
XX:MaxTenuringThreshold=31 ``-XX:+CMSParallelRemarkEnabled ``-XX:+UseCMSCompactAtFullCollection ``-XX:LargePageSizeInBytes=128m ``-
XX:+UseFastAccessorMethods ``-XX:+UseCMSInitiatingOccupancyOnly``-Duser.timezone=Asia` `/Shanghai` `-Djava.awt.headless=` `true` `"
```

为了看着方便，将每个参数单独写一行。上面参数好多啊，可能有人写到现在都没见过一个在 Tomcat 的启动命令里加了这么多参数，当然，这些参数只是我机器上的，不一定适合你，尤其是参数后的 value（值）是需要根据你自己的实际情况来设置的。

上述这样的配置，基本上可以达到：

系统响应时间增快；

JVM回收速度增快同时又不影响系统的响应率；

JVM内存最大化利用；

线程阻塞情况最小化。

JVM 常用参数详解：

-server：一定要作为第一个参数，在多个 CPU 时性能佳，还有一种叫 -client 的模式，特点是启动速度比较快，但运行时性能和内存管理效率不高，通常用于客户端应用程序或开发调试，在 32 位环境下直接运行 Java 程序默认启用该模式。Server 模式的特点是启动速度比较慢，但运行时性能和内存管理效率很高，适用于生产环境，在具有 64 位能力的 JDK 环境下默认启用该模式，可以不配置该参数。

-Xms：表示 Java 初始化堆的大小，-Xms 与-Xmx 设成一样的值，避免 JVM 反复重新申请内存，导致性能大起大落，默认值为物理内存的 1/64，默认（MinHeapFreeRatio参数可以调整）空余堆内存小于 40% 时，JVM 就会增大堆直到 -Xmx 的最大限制。

-Xmx：表示最大 Java 堆大小，当应用程序需要的内存超出堆的最大值时虚拟机就会提示内存溢出，并且导致应用服务崩溃，因此一般建议堆的最大值设置为可用内存的最大值的 80%。如何知道我的 JVM 能够使用最大值，使用 java -Xmx512M -version 命令来进行测试，然后逐渐的增大 512 的值,如果执行正常就表示指定的内存大小可用，否则会打印错误信息，默认值为物理内存的 1/4，默认（MinHeapFreeRatio参数可以调整）空余堆内存大于 70% 时，JVM 会减少堆直到-Xms 的最小限制。

-Xss：表示每个 Java 线程堆栈大小，JDK 5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。根据应用的线程所需内存大小进行调整，在相同物理内存下，减小这个值能生成更多的线程，但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。一般小的应用， 如果栈不是很深， 应该是128k 够用的，大的应用建议使用 256k 或 512K，一般不易设置超过 1M，要不然容易出现out ofmemory。这个选项对性能影响比较大，需要严格的测试。

-XX:NewSize：设置新生代内存大小。

-XX:MaxNewSize：设置最大新生代新生代内存大小

-XX:PermSize：设置持久代内存大小

-XX:MaxPermSize：设置最大值持久代内存大小，永久代不属于堆内存，堆内存只包含新生代和老年代。

-XX:+AggressiveOpts：作用如其名（aggressive），启用这个参数，则每当 JDK 版本升级时，你的 JVM 都会使用最新加入的优化技术（如果有的话）。

-XX:+UseBiasedLocking：启用一个优化了的线程锁，我们知道在我们的appserver，每个http请求就是一个线程，有的请求短有的请求长，就会有请求排队的现象，甚至还会出现线程阻塞，这个优化了的线程锁使得你的appserver内对线程处理自动进行最优调配。

-XX:+DisableExplicitGC：在 程序代码中不允许有显示的调用“System.gc()”。每次在到操作结束时手动调用 System.gc() 一下，付出的代价就是系统响应时间严重降低，就和关于 Xms，Xmx 里的解释的原理一样，这样去调用 GC 导致系统的 JVM 大起大落。

-XX:+UseConcMarkSweepGC：设置年老代为并发收集，即 CMS gc，这一特性只有 jdk1.5
后续版本才具有的功能，它使用的是 gc 估算触发和 heap 占用触发。我们知道频频繁的 GC 会造面 JVM
的大起大落从而影响到系统的效率，因此使用了 CMS GC 后可以在 GC 次数增多的情况下，每次 GC 的响应时间却很短，比如说使用了 CMS
GC 后经过 jprofiler 的观察，GC 被触发次数非常多，而每次 GC 耗时仅为几毫秒。

-XX:+UseParNewGC：对新生代采用多线程并行回收，这样收得快，注意最新的 JVM 版本，当使用 -XX:+UseConcMarkSweepGC 时，-XX:UseParNewGC 会自动开启。因此，如果年轻代的并行 GC 不想开启，可以通过设置 -XX：-UseParNewGC 来关掉。

-XX:MaxTenuringThreshold：设置垃圾最大年龄。如果设置为0的话，则新生代对象不经过 Survivor 区，直接进入老年代。对于老年代比较多的应用（需要大量常驻内存的应用），可以提高效率。如果将此值设置为一 个较大值，则新生代对象会在 Survivor 区进行多次复制，这样可以增加对象在新生代的存活时间，增加在新生代即被回收的概率，减少Full GC的频率，这样做可以在某种程度上提高服务稳定性。该参数只有在串行 GC 时才有效，这个值的设置是根据本地的 jprofiler 监控后得到的一个理想的值，不能一概而论原搬照抄。

-XX:+CMSParallelRemarkEnabled：在使用 UseParNewGC 的情况下，尽量减少 mark 的时间。

-XX:+UseCMSCompactAtFullCollection：在使用 concurrent gc 的情况下，防止 memoryfragmention，对 live object 进行整理，使 memory 碎片减少。

-XX:LargePageSizeInBytes：指定 Java heap 的分页页面大小，内存页的大小不可设置过大， 会影响 Perm 的大小。

-XX:+UseFastAccessorMethods：使用 get，set 方法转成本地代码，原始类型的快速优化。

-XX:+UseCMSInitiatingOccupancyOnly：只有在 oldgeneration 在使用了初始化的比例后 concurrent collector 启动收集。

-Duser.timezone=Asia/Shanghai：设置用户所在时区。

-Djava.awt.headless=true：这个参数一般我们都是放在最后使用的，这全参数的作用是这样的，有时我们会在我们的 J2EE 工程中使用一些图表工具如：jfreechart，用于在 web 网页输出 GIF/JPG 等流，在 winodws 环境下，一般我们的 app server 在输出图形时不会碰到什么问题，但是在linux/unix 环境下经常会碰到一个 exception 导致你在 winodws 开发环境下图片显示的好好可是在 linux/unix 下却显示不出来，因此加上这个参数以免避这样的情况出现。

-Xmn：新生代的内存空间大小，注意：此处的大小是（eden+ 2 survivor space)。与 jmap -heap 中显示的 New gen 是不同的。整个堆大小 = 新生代大小 + 老生代大小 + 永久代大小。在保证堆大小不变的情况下，增大新生代后，将会减小老生代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的 3/8。

-XX:CMSInitiatingOccupancyFraction：当堆满之后，并行收集器便开始进行垃圾收集，例如，当没有足够的空间来容纳新分配或提升的对象。对于 CMS 收集器，长时间等待是不可取的，因为在并发垃圾收集期间应用持续在运行（并且分配对象）。因此，为了在应用程序使用完内存之前完成垃圾收集周期，CMS 收集器要比并行收集器更先启动。因为不同的应用会有不同对象分配模式，JVM 会收集实际的对象分配（和释放）的运行时数据，并且分析这些数据，来决定什么时候启动一次 CMS 垃圾收集周期。这个参数设置有很大技巧，基本上满足(Xmx-Xmn)*(100-CMSInitiatingOccupancyFraction)/100 >= Xmn 就不会出现 promotion failed。例如在应用中 Xmx 是6000，Xmn 是 512，那么 Xmx-Xmn 是 5488M，也就是老年代有 5488M，CMSInitiatingOccupancyFraction=90 说明老年代到 90% 满的时候开始执行对老年代的并发垃圾回收（CMS），这时还 剩 10% 的空间是 5488*10% = 548M，所以即使 Xmn（也就是新生代共512M）里所有对象都搬到老年代里，548M 的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的 promotion failed，因此这个参数的设置必须与 Xmn 关联在一起。

-XX:+CMSIncrementalMode：该标志将开启 CMS 收集器的增量模式。增量模式经常暂停 CMS 过程，以便对应用程序线程作出完全的让步。因此，收集器将花更长的时间完成整个收集周期。因此，只有通过测试后发现正常 CMS 周期对应用程序线程干扰太大时，才应该使用增量模式。由于现代服务器有足够的处理器来适应并发的垃圾收集，所以这种情况发生得很少，用于但 CPU情况。

-XX:NewRatio：年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代），-XX:NewRatio=4 表示年轻代与年老代所占比值为 1:4，年轻代占整个堆栈的 1/5，Xms=Xmx 并且设置了 Xmn 的情况下，该参数不需要进行设置。

-XX:SurvivorRatio：Eden 区与 Survivor 区的大小比值，设置为 8，表示 2 个 Survivor 区（JVM 堆内存年轻代中默认有 2 个大小相等的 Survivor 区）与 1 个 Eden 区的比值为 2:8，即 1 个 Survivor 区占整个年轻代大小的 1/10。

-XX:+UseSerialGC：设置串行收集器。

-XX:+UseParallelGC：设置为并行收集器。此配置仅对年轻代有效。即年轻代使用并行收集，而年老代仍使用串行收集。

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集，JDK6.0 开始支持对年老代并行收集。

-XX:ConcGCThreads：早期 JVM 版本也叫-XX:ParallelCMSThreads，定义并发 CMS 过程运行时的线程数。比如 value=4 意味着 CMS 周期的所有阶段都以 4 个线程来执行。尽管更多的线程会加快并发 CMS 过程，但其也会带来额外的同步开销。因此，对于特定的应用程序，应该通过测试来判断增加 CMS 线程数是否真的能够带来性能的提升。如果还标志未设置，JVM 会根据并行收集器中的 -XX:ParallelGCThreads 参数的值来计算出默认的并行 CMS 线程数。

-XX:ParallelGCThreads：配置并行收集器的线程数，即：同时有多少个线程一起进行垃圾回收，此值建议配置与 CPU 数目相等。

-XX:OldSize：设置 JVM 启动分配的老年代内存大小，类似于新生代内存的初始大小 -XX:NewSize。

以上就是一些常用的配置参数，有些参数是可以被替代的，配置思路需要考虑的是 Java 提供的垃圾回收机制。虚拟机的堆大小决定了虚拟机花费在收集垃圾上的时间和频度。收集垃圾能够接受的速度和应用有关，应该通过分析实际的垃圾收集的时间和频率来调整。假如堆的大小很大，那么完全垃圾收集就会很慢，但是频度会降低。假如您把堆的大小和内存的需要一致，完全收集就很快，但是会更加频繁。调整堆大小的的目的是最小化垃圾收集的时间，以在特定的时间内最大化处理客户的请求。在基准测试的时候，为确保最好的性能，要把堆的大小设大，确保垃圾收集不在整个基准测试的过程中出现。

假如系统花费很多的时间收集垃圾，请减小堆大小。一次完全的垃圾收集应该不超过 3-5 秒。假如垃圾收集成为瓶颈，那么需要指定代的大小，检查垃圾收集的周详输出，研究垃圾收集参数对性能的影响。当增加处理器时，记得增加内存，因为分配能够并行进行，而垃圾收集不是并行的。

3、设置系统属性

之前说过，Tomcat 的语言编码，配置起来很慢，要经过多次设置才可以了，否则中文很有可能出现乱码情况。譬如汉字"中"，以 UTF-8 编码后得到的是 3 字节的值 %E4%B8%AD，然后通过 GET 或者 POST 方式把这 3 个字节提交到 Tomcat 容器，如果你不告诉 Tomcat 我的参数是用 UTF-8编码的，那么 Tomcat 就认为你是用 ISO-8859-1 来编码的，而 ISO8859-1（兼容 URI 中的标准字符集 US-ASCII）是兼容 ASCII 的单字节编码并且使用了单字节内的所有空间，因此 Tomcat 就以为你传递的用 ISO-8859-1 字符集编码过的 3 个字符，然后它就用 ISO-8859-1 来解码。

设置起来不难使用" -D<名称>=<值> "来设置系统属性：

-Djavax.servlet.request.encoding=UTF-8

-Djavax.servlet.response.encoding=UTF-8

-Dfile.encoding=UTF-8

-Duser.country=CN

-Duser.language=zh

4、常见的 Java 内存溢出有以下三种

（1） java.lang.OutOfMemoryError: Java heap space —-JVM Heap（堆）溢出

JVM 在启动的时候会自动设置 JVM Heap 的值，其初始空间（即-Xms）是物理内存的1/64，最大空间（-Xmx）不可超过物理内存。可以利用 JVM提供的 -Xmn -Xms -Xmx 等选项可进行设置。Heap 的大小是 Young Generation 和 Tenured Generaion 之和。在 JVM 中如果 98% 的时间是用于 GC，且可用的 Heap size 不足 2% 的时候将抛出此异常信息。

解决方法：手动设置 JVM Heap（堆）的大小。
（2） java.lang.OutOfMemoryError: PermGen space —- PermGen space溢出。

PermGen space 的全称是 Permanent Generation space，是指内存的永久保存区域。为什么会内存溢出，这是由于这块内存主要是被 JVM 存放Class 和 Meta 信息的，Class 在被 Load 的时候被放入 PermGen space 区域，它和存放 Instance 的 Heap 区域不同，sun 的 GC 不会在主程序运行期对 PermGen space 进行清理，所以如果你的 APP 会载入很多 CLASS 的话，就很可能出现 PermGen space 溢出。

解决方法： 手动设置 MaxPermSize 大小

（3） java.lang.StackOverflowError —- 栈溢出

栈溢出了，JVM 依然是采用栈式的虚拟机，这个和 C 与 Pascal 都是一样的。函数的调用过程都体现在堆栈和退栈上了。调用构造函数的 "层"太多了，以致于把栈区溢出了。通常来讲，一般栈区远远小于堆区的，因为函数调用过程往往不会多于上千层，而即便每个函数调用需要 1K 的空间（这个大约相当于在一个 C 函数内声明了 256 个 int 类型的变量），那么栈区也不过是需要 1MB 的空间。通常栈的大小是 1 - 2MB 的。
通常递归也不要递归的层次过多，很容易溢出。

解决方法：修改程序。

设计模式

设计原则

工厂方法模式

抽象工厂模式

单例模式

建造者模式

原型模式

适配器模式

装饰器模式

代理模式

外观模式

桥接模式

组合模式

享元模式

策略模式

模板方法模式

观察者模式

迭代子模式

责任链模式

命令模式

备忘录模式

状态模式

访问者模式

中介者模式

解释器模式