目录

[Jvm 7](#_Toc5792831)

[内存划分 7](#_Toc5792832)

[GC机制和算法 11](#_Toc5792833)

[判断对象是否要回收的方法：可达性分析法 11](#_Toc5792834)

[二、 发现虚拟机频繁full GC时应该怎么办 11](#_Toc5792835)

[三、常见的垃圾回收算法 12](#_Toc5792836)

[为什么要分为新生代和老年代 13](#_Toc5792837)

[Jdk8垃圾回收新特性 13](#_Toc5792838)

[垃圾回收器 13](#_Toc5792839)

[JVM内存优化 14](#_Toc5792840)

[jvm中堆内存泄漏时怎么定位到导致内存泄漏的代码 16](#_Toc5792841)

[类加载机制 17](#_Toc5792842)

[反射机制 18](#_Toc5792843)

[JVM运行参数 19](#_Toc5792844)

[一、堆大小设置（堆 = 年轻代 + 年老代+ 持久代） 19](#_Toc5792845)

[二、垃圾回收器设置（串行收集器、并行收集器、并发收集器 ） 20](#_Toc5792846)

[三、辅助信息参数设置 22](#_Toc5792847)

[多线程 23](#_Toc5792848)

[线程状态以及转换 23](#_Toc5792849)

[说明 23](#_Toc5792850)

[wait(), notify(), notifyAll()等方法介绍 24](#_Toc5792851)

[为什么notify(), wait()等函数定义在Object中，而不是Thread中 24](#_Toc5792852)

[yield()介绍 25](#_Toc5792853)

[yield() 与 wait()的比较 25](#_Toc5792854)

[sleep()介绍 25](#_Toc5792855)

[sleep() 与 wait()的比较 25](#_Toc5792856)

[ThreadPoolExecutor创建线程池、以及如何计算大小、拒绝策略 25](#_Toc5792857)

[为什么要自己定义线程池 25](#_Toc5792858)

[ThreadPoolExecutor 26](#_Toc5792859)

[线程池的任务处理策略： 27](#_Toc5792860)

[线程池的关闭 28](#_Toc5792861)

[如何选择线程池数量 28](#_Toc5792862)

[线程池工厂 30](#_Toc5792863)

[扩展线程池 30](#_Toc5792864)

[手动创建线程池有几个注意点 30](#_Toc5792865)

[线程池中任务执行完后什么状态 31](#_Toc5792866)

[Synchronized和reentrantlock的区别 31](#_Toc5792867)

[Synchronized在JVM中的实现原理 31](#_Toc5792868)

[ThreadLocal的作用和弊端 31](#_Toc5792869)

[实现原理 32](#_Toc5792870)

[内存泄露 33](#_Toc5792871)

[应用场景 33](#_Toc5792872)

[InheritableThreadLocal 34](#_Toc5792873)

[Volatile 35](#_Toc5792874)

[公平锁和非公平锁 35](#_Toc5792875)

[线程安全的类 36](#_Toc5792876)

[并发包下工具类 36](#_Toc5792877)

[进程和线程的区别、通信机制 36](#_Toc5792878)

[线程和进程各自有什么区别和优劣呢？ 36](#_Toc5792879)

[进程通信 37](#_Toc5792880)

[线程通信 38](#_Toc5792881)

[多线程中的锁 38](#_Toc5792882)

[多线程运用实例 39](#_Toc5792883)

[多线程中runable和callable的区别 39](#_Toc5792884)

[两个方法均有sycrinised关键字 两个线程分别去执行这两个方法 是否有可能会造成死锁 39](#_Toc5792885)

[线程阻塞后唤醒 39](#_Toc5792886)

[Condition 40](#_Toc5792887)

[信号量Semaphore 40](#_Toc5792888)

[ReadWriteLock读写锁 40](#_Toc5792889)

[倒计时器CountDownLatch 40](#_Toc5792890)

[循环栅栏CyclicBarrier 40](#_Toc5792891)

[线程阻塞工具类LockSupport 40](#_Toc5792892)

[集合 40](#_Toc5792893)

[HashMap数据结构、扩容、构造 40](#_Toc5792894)

[HashMap resize后原来元素的位置如何变化 43](#_Toc5792895)

[Hashtable 43](#_Toc5792896)

[实现线程安全的HashMap 43](#_Toc5792897)

[ConcurrentHashMap结构 44](#_Toc5792898)

[如何对map中的元素根据key进行排序 44](#_Toc5792899)

[ArrayList和LinkedList 44](#_Toc5792900)

[List循环删除元素的问题 44](#_Toc5792901)

[CopyOnWriteArrayList 46](#_Toc5792902)

[ConcurrentLinkedQueue 46](#_Toc5792903)

[BlockingQueue 46](#_Toc5792904)

[ConcurrentSkipListMap 46](#_Toc5792905)

[遍历Map 46](#_Toc5792906)

[IO 47](#_Toc5792907)

[IO 字节流、字符流 47](#_Toc5792908)

[IO流的分类 48](#_Toc5792909)

[字符流和字节流的区别 48](#_Toc5792910)

[字节流和字符流的转换 49](#_Toc5792911)

[AIO、BIO、NIO 51](#_Toc5792912)

[Java中BIO、NIO、AIO的区别 51](#_Toc5792913)

[BIO、NIO、AIO适用场景分析 52](#_Toc5792914)

[NIO 53](#_Toc5792915)

[数据结构 54](#_Toc5792916)

[直接选择排序 54](#_Toc5792917)

[二分排序 55](#_Toc5792918)

[快速排序 57](#_Toc5792919)

[归并排序 57](#_Toc5792920)

[快速排序与归并排序比较 59](#_Toc5792921)

[堆排序 59](#_Toc5792922)

[希尔排序 59](#_Toc5792923)

[冒泡排序 59](#_Toc5792924)

[堆和栈 59](#_Toc5792925)

[队列和栈互相转化 59](#_Toc5792926)

[二叉树 63](#_Toc5792927)

[B树 63](#_Toc5792928)

[括号匹配问题 63](#_Toc5792929)

[循环报数取剩余 65](#_Toc5792930)

[大数据量排序问题 67](#_Toc5792931)

[求数组交集 67](#_Toc5792932)

[计算表达式 67](#_Toc5792933)

[一个数组里面有正数、负数 量1000万，找出子集之和最大的序列 67](#_Toc5792934)

[一亿个数据查出最大的前10个 68](#_Toc5792935)

[字符串数组找出里面出现次数最多的字符 68](#_Toc5792936)

[设计模式 69](#_Toc5792937)

[单例模式 69](#_Toc5792938)

[1、方式1 （饿汉式） 70](#_Toc5792939)

[2、方式1 （懒汉式） 70](#_Toc5792940)

[3、 用synchronized  加锁同步 72](#_Toc5792941)

[4、改进性能  双重校验 73](#_Toc5792942)

[5、volatile 关键字，解决双重校验带来的弊端 73](#_Toc5792943)

[6、静态内部类 74](#_Toc5792944)

[7、枚举类实现单例 75](#_Toc5792945)

[工厂模式 77](#_Toc5792946)

[代理模式 77](#_Toc5792947)

[观察者模式 77](#_Toc5792948)

[观察者模式的定义： 77](#_Toc5792949)

[大白话： 77](#_Toc5792950)

[可以看到，该模式包含四个角色 77](#_Toc5792951)

[使用场景例子 77](#_Toc5792952)

[适配器模式 78](#_Toc5792953)

[桥接模式 78](#_Toc5792954)

[策略模式 78](#_Toc5792955)

[Spring 79](#_Toc5792956)

[IOC原理、如何依赖注入 79](#_Toc5792957)

[互相依赖情况下spring能否启动成功 79](#_Toc5792958)

[@autowire和@resource的区别 80](#_Toc5792959)

[AOP原理 80](#_Toc5792960)

[AOP执行流程 80](#_Toc5792961)

[SpringMVC请求过程 80](#_Toc5792962)

[Spring事务控制 80](#_Toc5792963)

[Spring bean的生命周期 80](#_Toc5792964)

[Spring和SpringBoot的区别 80](#_Toc5792965)

[如何实现注解 80](#_Toc5792966)

[Hibernate、Mybatis 80](#_Toc5792967)

[两个ORM的区别 80](#_Toc5792968)

[Mybatis批量操作 82](#_Toc5792969)

[MyBatis中批量插入 82](#_Toc5792970)

[MyBatis中批量删除 83](#_Toc5792971)

[MyBatis中in子句 84](#_Toc5792972)

[Hibernate缓存机制 86](#_Toc5792973)

[Mybatis防止SQL注入 86](#_Toc5792974)

[Mybatis对JDBC进行的封装 87](#_Toc5792975)

[数据库 87](#_Toc5792976)

[悲观锁和乐观锁、如何实现 87](#_Toc5792977)

[事务特性 88](#_Toc5792978)

[事务隔离级别 88](#_Toc5792979)

[不考虑事务的隔离性，会产生的几种问题： 89](#_Toc5792980)

[事务传播机制 90](#_Toc5792981)

[一个慢sql能怎么优化 90](#_Toc5792982)

[SQL优化 90](#_Toc5792983)

[SQL执行计划 90](#_Toc5792984)

[myCat分库分表、以及如何查询 90](#_Toc5792985)

[分库分表策略 90](#_Toc5792986)

[索引 90](#_Toc5792987)

[聚合索引、单列索引 90](#_Toc5792988)

[聚集索引和非聚集索引 90](#_Toc5792989)

[走不走索引 90](#_Toc5792990)

[B+树的结构 90](#_Toc5792991)

[JDBC连接数据库 90](#_Toc5792992)

[设计数据库遵循的规范 92](#_Toc5792993)

[防止SQL注入 92](#_Toc5792994)

[MySql、Oracle 92](#_Toc5792995)

[两者区别 92](#_Toc5792996)

[默认隔离级别 92](#_Toc5792997)

[mysql优化 92](#_Toc5792998)

[MySQL事务传播级别 92](#_Toc5792999)

[MySQL分页 92](#_Toc5793000)

[分页比较 94](#_Toc5793001)

[Redis 94](#_Toc5793002)

[数据结构、以及特性、为何快速 94](#_Toc5793003)

[Redis为什么 那么快 94](#_Toc5793004)

[运用场景 95](#_Toc5793005)

[分布式锁用法 96](#_Toc5793006)

[Expire过期机制 96](#_Toc5793007)

[复制机制 96](#_Toc5793008)

[集群部署方式 96](#_Toc5793009)

[负载均衡 101](#_Toc5793010)

[批量操作数据 101](#_Toc5793011)

[持久化机制 101](#_Toc5793012)

[Redis挂了怎么办 101](#_Toc5793013)

[缓存更新策略（即如何让缓存和mysql保持一致性） 102](#_Toc5793014)

[热点key优化 103](#_Toc5793015)

[缓存击穿、缓存雪崩 103](#_Toc5793016)

[缓存更新策略 103](#_Toc5793017)

[Redis和memcached的区别 104](#_Toc5793018)

[经常更新的数据是否可以用缓存 105](#_Toc5793019)

[Dubbo 105](#_Toc5793020)

[架构图 105](#_Toc5793021)

[负载均衡 107](#_Toc5793022)

[服务熔断、降级 107](#_Toc5793023)

[服务升级 107](#_Toc5793024)

[服务网关 107](#_Toc5793025)

[分布式 107](#_Toc5793026)

[分布式优势、弊端、区别 107](#_Toc5793027)

[微服务和SOA 107](#_Toc5793028)

[微服务的好处 107](#_Toc5793029)

[分布式事务解决 107](#_Toc5793030)

[幂等性解决方案 107](#_Toc5793031)

[分布式Session 107](#_Toc5793032)

[限流算法 107](#_Toc5793033)

[集群有服务器挂了或者上线怎么处理 107](#_Toc5793034)

[缓存更新问题 107](#_Toc5793035)

[如何负载均衡 107](#_Toc5793036)

[分布式quartz 108](#_Toc5793037)

[MQ 108](#_Toc5793038)

[MQ的应用场景 108](#_Toc5793039)

[RabbitMQ架构 108](#_Toc5793040)

[RabbitMQ集群 108](#_Toc5793041)

[如何保证消息不丢失 109](#_Toc5793042)

[保证消息不被重复消费 110](#_Toc5793043)

[消息100%可靠性投递解决方案 112](#_Toc5793044)

[Linux 112](#_Toc5793045)

[常用命令 112](#_Toc5793046)

[实时查看Tomcat日志 112](#_Toc5793047)

[远程copy文件 112](#_Toc5793048)

[查看端口 112](#_Toc5793049)

[查看、杀死进程 112](#_Toc5793050)

[查看内存占用 112](#_Toc5793051)

[项目错误排查 112](#_Toc5793052)

[打开超大文件 112](#_Toc5793053)

[项目问题 112](#_Toc5793054)

[秒杀、超卖 112](#_Toc5793055)

[购物车实现 112](#_Toc5793056)

[订单退款操作 112](#_Toc5793057)

[物流 112](#_Toc5793058)

[订单模块调用的微服务 113](#_Toc5793059)

[电影下单模块调用的微服务 113](#_Toc5793060)

[消息可靠性 113](#_Toc5793061)

[项目架构图 113](#_Toc5793062)

[多实例令牌桶 114](#_Toc5793063)

[Java基础 114](#_Toc5793064)

[==、Equals、hashcode区别 114](#_Toc5793065)

[接口和抽象类 114](#_Toc5793066)

[深克隆和浅克隆 115](#_Toc5793067)

[强引用和弱引用 116](#_Toc5793068)

[把对象按某个属性排序 116](#_Toc5793069)

[Java web 116](#_Toc5793070)

[SSO实现 116](#_Toc5793071)

[session共享除了用redis还有什么办法 116](#_Toc5793072)

[Session的实现机制 116](#_Toc5793073)

[Cookie禁用 116](#_Toc5793074)

[Cookie与Session的区别 116](#_Toc5793075)

[RestfulAPI理解 116](#_Toc5793076)

[nginx底层原理 116](#_Toc5793077)

[http 116](#_Toc5793078)

[tcp和udp的区别 116](#_Toc5793079)

[tcp连接过程 116](#_Toc5793080)

[Java新特性 116](#_Toc5793081)

[JDK8新特性 116](#_Toc5793082)

[前端 117](#_Toc5793083)

[闭包理解、以及解决的问题 117](#_Toc5793084)

[最困难的事 117](#_Toc5793085)

[职业规划 117](#_Toc5793086)

[学习方式、看什么书 117](#_Toc5793087)

[自我评价 117](#_Toc5793088)

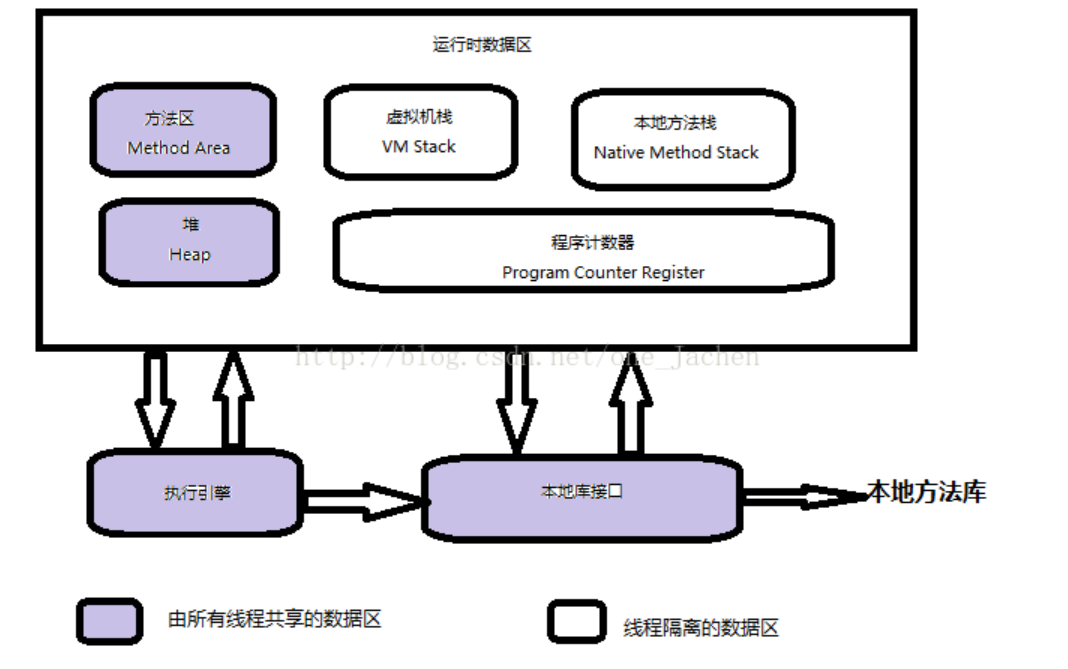
[有什么想问的 117](#_Toc5793089)

[谈薪资 117](#_Toc5793090)

[需要注意的问题 117](#_Toc5793091)

# Jvm

## 内存划分



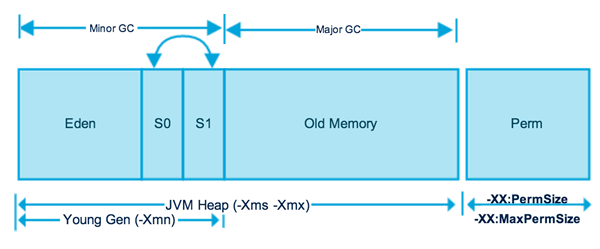
1、程序计数器：指向当前线程正在执行的字节码指令。线程私有的。  
2、虚拟机栈：虚拟机栈是Java执行方法的内存模型。每个方法被执行的时候，都会创建一个栈帧，把栈帧压人栈，当方法正常返回或者抛出未捕获的异常时，栈帧就会出栈。  
（1）栈帧：栈帧存储方法的相关信息，包含局部变量数表、返回值、操作数栈、动态链接  
a、局部变量表：包含了方法执行过程中的所有变量。局部变量数组所需要的空间在编译期间完成分配，在方法运行期间不会改变局部变量数组的大小。  
b、返回值：如果有返回值的话，压入调用者栈帧中的操作数栈中，并且把PC的值指向 方法调用指令 后面的一条指令地址。  
c、操作数栈：操作变量的内存模型。操作数栈的最大深度在编译的时候已经确定（写入方法区code属性的max\_stacks项中）。操作数栈的的元素可以是任意Java类型，包括long和double，32位数据占用栈空间为1，64位数据占用2。方法刚开始执行的时候，栈是空的，当方法执行过程中，各种字节码指令往栈中存取数据。  
d、动态链接：每个栈帧都持有在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链接。  
（2）线程私有  
3、本地方法栈：  
（1）调用本地native的内存模型  
（2）线程独享。  
4、方法区：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  
（1）线程共享的  
（2）运行时常量池：

A、是方法区的一部分

B、存放编译期生成的各种字面量和符号引用

C、**Class**文件中除了存有类的版本、字段、方法、接口等描述信息，还有一项是常量池，存有这个类的 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。

5、堆（Heap）：Java对象存储的地方  
（1）Java堆是虚拟机管理的内存中最大的一块  
（2）Java堆是所有线程共享的区域  
（3）在虚拟机启动时创建  
（4）此内存区域的唯一目的就是存放对象实例，几乎所有对象实例都在这里分配内存。存放new生成的对象和数组  
（5）Java堆是垃圾收集器管理的内存区域，因此很多时候称为“GC堆”



Java堆的内存划分如图所示，分别为年轻代、Old Memory（老年代）、Perm（永久代）。其中在Jdk1.8中，永久代被移除，使用MetaSpace代替。  
1、新生代：  
（1）使用复制清除算法（Copinng算法），原因是年轻代每次GC都要回收大部分对象。新生代里面分成一份较大的Eden空间和两份较小的Survivor空间。每次只使用Eden和其中一块Survivor空间，然后垃圾回收的时候，把存活对象放到未使用的Survivor（划分出from、to）空间中，清空Eden和刚才使用过的Survivor空间。  
（2）分为Eden、Survivor From、Survivor To，比例默认为8：1：1  
（3）内存不足时发生Minor GC  
2、老年代：  
（1）采用标记-整理算法（mark-compact），原因是老年代每次GC只会回收少部分对象。  
3、Perm：用来存储类的元数据，也就是方法区。  
（1）Perm的废除：在jdk1.8中，Perm被替换成MetaSpace，MetaSpace存放在本地内存中。原因是永久代进场内存不够用，或者发生内存泄漏。  
（2）MetaSpace（元空间）：元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过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。

**年轻代：**

         所有新生成的对象首先都是放在**年轻代**的。**年轻代**的目标就是尽可能快速的收集掉那些生命周期短的对象。**年轻代**分为三个区。一个Eden区，两个Survivor区（一般而言）。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活将被复制到另外一个Survivor区，当这个Survivor区也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制“年老区（Tenured）”。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来的对象和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor区过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在**年轻代**中的存在时间，减少被放到**年老代**的可能。

**年老代：**

         在**年轻代**中经历了N次垃圾回收后仍然存活的对象，就会被放到**年老代**中。因此，可以认为**年老代**中存放的都是一些生命周期较长的对象。

**持久代：**

         用于存放静态文件，如**java**类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class,例如Hibernate等，在这种时候需要设置一个比较大的持久空间来存放这些运行过程中新增的类。持久代大小通过 -XX:MaxPermSize = <N> 进行设置。

## GC机制和算法

### 判断对象是否要回收的方法：可达性分析法

1、 可达性分析法：通过一系列“GC Roots”对象作为起点进行搜索，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。不可达对象不一定会成为可回收对象。进入DEAD状态的线程还可以恢复，GC不会回收它的内存。（把一些对象当做root对象，JVM认为root对象是不可回收的，并且root对象引用的对象也是不可回收的）

2、 以下对象会被认为是root对象：

（1） 虚拟机栈（栈帧中本地变量表）中引用的对象

（2） 方法区中静态属性引用的对象

（3） 方法区中常量引用的对象

（4） 本地方法栈中Native方法引用的对象

3、 对象被判定可被回收，需要经历两个阶段：

（1） 第一个阶段是可达性分析，分析该对象是否可达

（2） 第二个阶段是当对象没有重写finalize()方法或者finalize()方法已经被调用过，虚拟机认为该对象不可以被救活，因此回收该对象。（finalize()方法在垃圾回收中的作用是，给该对象一次救活的机会）

4、 方法区中的垃圾回收：

（1） 常量池中一些常量、符号引用没有被引用，则会被清理出常量池

（2） 无用的类：被判定为无用的类，会被清理出方法区。判定方法如下：

A、 该类的所有实例被回收

B、 加载该类的ClassLoader被回收

C、 该类的Class对象没有被引用

5、 finalize():

（1） GC垃圾回收要回收一个对象的时候，调用该对象的finalize()方法。然后在下一次垃圾回收的时候，才去回收这个对象的内存。

（2） 可以在该方法里面，指定一些对象在释放前必须执行的操作。

### 二、 发现虚拟机频繁full GC时应该怎么办

（full GC指的是清理整个堆空间，包括年轻代和永久代）

（1） 首先用命令查看触发GC的原因是什么 jstat –gccause 进程id

（2） 如果是System.gc()，则看下代码哪里调用了这个方法

（3） 如果是heap inspection(内存检查)，可能是哪里执行jmap –histo[:live]命令

（4） 如果是GC locker，可能是程序依赖的JNI库的原因

### 三、常见的垃圾回收算法

1、Mark-Sweep（标记-清除算法）：

（1）思想：标记清除算法分为两个阶段，标记阶段和清除阶段。标记阶段任务是标记出所有需要回收的对象，清除阶段就是清除被标记对象的空间。

（2）优缺点：实现简单，容易产生内存碎片

2、Copying（复制清除算法）：

（1）思想：将可用内存划分为大小相等的两块，每次只使用其中的一块。当进行垃圾回收的时候了，把其中存活对象全部复制到另外一块中，然后把已使用的内存空间一次清空掉。

（2）优缺点：不容易产生内存碎片；可用内存空间少；存活对象多的话，效率低下。

3、Mark-Compact（标记-整理算法）：

（1）思想：先标记存活对象，然后把存活对象向一边移动，然后清理掉端边界以外的内存。

（2）优缺点：不容易产生内存碎片；内存利用率高；存活对象多并且分散的时候，移动次数多，效率低下

4、分代收集算法：（目前大部分JVM的垃圾收集器所采用的算法）：

思想：把堆分成新生代和老年代。（永久代指的是方法区）

（1） 因为新生代每次垃圾回收都要回收大部分对象，所以新生代采用Copying算法。新生代里面分成一份较大的Eden空间和两份较小的Survivor空间。每次只使用Eden和其中一块Survivor空间，然后垃圾回收的时候，把存活对象放到未使用的Survivor（划分出from、to）空间中，清空Eden和刚才使用过的Survivor空间。

（2） 由于老年代每次只回收少量的对象，因此采用mark-compact算法。

（3） 在堆区外有一个永久代。对永久代的回收主要是无效的类和常量

5、GC使用时对程序的影响？

垃圾回收会影响程序的性能，Java虚拟机必须要追踪运行程序中的有用对象，然后释放没用对象，这个过程消耗处理器时间

6、几种不同的垃圾回收类型：

（1）Minor GC：从年轻代（包括Eden、Survivor区）回收内存, 当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC

A、当JVM无法为一个新的对象分配内存的时候，越容易触发Minor GC。所以分配率越高，内存越来越少，越频繁执行Minor GC

B、执行Minor GC操作的时候，不会影响到永久代（Tenured）。从永久代到年轻代的引用，被当成GC Roots，从年轻代到老年代的引用在标记阶段直接被忽略掉。

（2）Major GC：清理整个老年代，当老年代中没有足够的内存空间来存放对象时。

（3）Full GC：清理整个堆空间，包括年轻代和老年代。当老年代内存不足时触发

如果对象在Eden出生并经过一次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，并将该对象的年龄设为1。对象每在Survivor中熬过一次Minor GC，年龄就增加1岁，当他的年龄增加到最大值15时，就将会被晋升到老年代中。虚拟机并不是永远地要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，如果在Survivor空间中所有相同年龄的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，无需等到MaxTenuringThreshold中要求的年龄。

补充概念，在JDK1.2之后引入了四个概念：强引用、软引用、弱引用、虚引用。

强引用：new出来的对象都是强引用，GC无论如何都不会回收，即使抛出OOM异常。

软引用：只有当JVM内存不足时才会被回收。

弱引用：只要GC,就会立马回收，不管内存是否充足。

虚引用：可以忽略不计，JVM完全不会在乎虚引用，你可以理解为它是来凑数的，凑够”四大天王”。它唯一的作用就是做一些跟踪记录，辅助finalize函数的使用。

## 为什么要分为新生代和老年代

年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象,在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象

## Jdk8垃圾回收新特性

元空间是java内存模型中一个极大的升级，从JDK8起，取消了持久代，使用元空间，从而引起了垃圾回收算法等的调整。

       为什么移除持久代

持久代的大小是在启动时固定好的——很难进行调优。-XX:MaxPermSize，设置成多少好呢？

HotSpot的内部类型也是Java对象：它可能会在Full GC中被移动，同时它对应用不透明，且是非强类型的，难以跟踪调试，还需要存储元数据的元数据信息（meta-metadata）。

        元空间的优点

简化Full GC：每一个回收器有专门的元数据迭代器。

可以在GC不进行暂停的情况下并发地释放类数据。

使得原来受限于持久代的一些改进未来有可能实现。

充分利用了Java语言规范中的好处：类及相关的元数据的生命周期与类加载器的一致。

每个加载器有专门的存储空间。

只进行线性分配。

不会单独回收某个类

省掉了GC扫描及压缩的时间

元空间里的对象的位置是固定的

如果GC发现某个类加载器不再存活了，会把相关的空间整个回收掉

  在JDK8上原有的JVM参数“-XX:MaxPermSize=128m”会无效，需要设置新的参数“-XX:MaxMetaspaceSize=128m”替换。

## 垃圾回收器

<https://blog.csdn.net/high2011/article/details/80177473>

JVM的垃圾回收器大致分为六种类型：

1、串行：垃圾回收器 (Serial Garbage Collector)

2、串行：ParNew收集器

3、并行：Parallel收集器

4、并行：Parallel Old 收集器

5、并发标记扫描CMS收集器

6、G1收集器

**小结：**

垃圾回收器目前分为6种类型, 串行，并行，并发标记，G1。

小数据量和小型应用，使用串行垃圾回收器即可。

对于对响应时间无特殊要求的，可以使用并行垃圾回收器和并发标记垃圾回收器。（中大型应用）

对于heap可以分配很大的中大型应用，使用G1垃圾回收器比较好，进一步优化和减少了GC暂停时间。

 没有银弹，针对不同的场景，选用不同的垃圾回收器

## JVM内存优化

1、一般来说，当survivor区不够大或者占用量达到50%，就会把一些对象放到老年区。通过设置合理的eden区，survivor区及使用率，可以将年轻对象保存在年轻代，从而避免full GC，使用-Xmn设置年轻代的大小

2、对于占用内存比较多的大对象，一般会选择在老年代分配内存。如果在年轻代给大对象分配内存，年轻代内存不够了，就要在eden区移动大量对象到老年代，然后这些移动的对象可能很快消亡，因此导致full GC。通过设置参数：-XX:PetenureSizeThreshold=1000000，单位为B，标明对象大小超过1M时，在老年代(tenured)分配内存空间。

3、一般情况下，年轻对象放在eden区，当第一次GC后，如果对象还存活，放到survivor区，此后，每GC一次，年龄增加1，当对象的年龄达到阈值，就被放到tenured老年区。这个阈值可以同构-XX:MaxTenuringThreshold设置。如果想让对象留在年轻代，可以设置比较大的阈值。

4、设置最小堆和最大堆：-Xmx和-Xms稳定的堆大小堆垃圾回收是有利的，获得一个稳定的堆大小的方法是设置-Xms和-Xmx的值一样，即最大堆和最小堆一样，如果这样子设置，系统在运行时堆大小理论上是恒定的，稳定的堆空间可以减少GC次数，因此，很多服务端都会将这两个参数设置为一样的数值。稳定的堆大小虽然减少GC次数，但是增加每次GC的时间，因为每次GC要把堆的大小维持在一个区间内。

5、一个不稳定的堆并非毫无用处。在系统不需要使用大内存的时候，压缩堆空间，使得GC每次应对一个较小的堆空间，加快单次GC次数。基于这种考虑，JVM提供两个参数，用于压缩和扩展堆空间。  
（1）-XX:MinHeapFreeRatio 参数用于设置堆空间的最小空闲比率。默认值是40，当堆空间的空闲内存比率小于40，JVM便会扩展堆空间  
（2）-XX:MaxHeapFreeRatio 参数用于设置堆空间的最大空闲比率。默认值是70， 当堆空间的空闲内存比率大于70，JVM便会压缩堆空间。  
（3）当-Xmx和-Xmx相等时，上面两个参数无效

6、通过增大吞吐量提高系统性能，可以通过设置并行垃圾回收收集器。  
（1）-XX:+UseParallelGC:年轻代使用并行垃圾回收收集器。这是一个关注吞吐量的收集器，可以尽可能的减少垃圾回收时间。  
（2）-XX:+UseParallelOldGC:设置老年代使用并行垃圾回收收集器。

7、尝试使用大的内存分页：使用大的内存分页增加CPU的内存寻址能力，从而系统的性能。-XX:+LargePageSizeInBytes设置内存页的大小

8、使用非占用的垃圾收集器。-XX:+UseConcMarkSweepGC老年代使用CMS收集器降低停顿。

9、-XXSurvivorRatio=3，表示年轻代中的分配比率：survivor:eden = 2:3

10、JVM性能调优的工具：  
（1）jps（Java Process Status）：输出JVM中运行的进程状态信息(现在一般使用jconsole)  
（2）jstack：查看java进程内线程的堆栈信息。  
（3）jmap：用于生成堆转存快照  
（4）jhat：用于分析jmap生成的堆转存快照（一般不推荐使用，而是使用Ecplise Memory Analyzer）  
（3）jstat是JVM统计监测工具。可以用来显示垃圾回收信息、类加载信息、新生代统计信息等。  
（4）VisualVM：故障处理工具

## jvm中堆内存泄漏时怎么定位到导致内存泄漏的代码

<https://blog.csdn.net/kingmax54212008/article/details/51858923>

<https://blog.csdn.net/moshenglv/article/details/82216240>

## 类加载机制

一、 概念：类加载器把class文件中的二进制数据读入到内存中，存放在方法区，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。类加载的步骤如下：  
1、加载：查找并加载类的二进制数据（把class文件里面的信息加载到内存里面）  
2、连接：把内存中类的二进制数据合并到虚拟机的运行时环境中  
（1）验证：确保被加载的类的正确性。包括：

A、类文件的结构检查：检查是否满足Java类文件的固定格式

B、语义检查：确保类本身符合Java的语法规范

C、字节码验证：确保字节码流可以被Java虚拟机安全的执行。字节码流是操作码组成的序列。每一个操作码后面都会跟着一个或者多个操作数。字节码检查这个步骤会检查每一个操作码是否合法。

D、二进制兼容性验证：确保相互引用的类之间是协调一致的。

（2）准备：为类的静态变量分配内存，并将其初始化为默认值  
（3）解析：把类中的符号引用转化为直接引用（比如说方法的符号引用，是有方法名和相关描述符组成，在解析阶段，JVM把符号引用替换成一个指针，这个指针就是直接引用，它指向该类的该方法在方法区中的内存位置）  
3、初始化：为类的静态变量赋予正确的初始值。当静态变量的等号右边的值是一个常量表达式时，不会调用static代码块进行初始化。只有等号右边的值是一个运行时运算出来的值，才会调用static初始化。

## 反射机制

Java反射就是在运行状态中，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方法和属性；并且能改变它的属性。

反射机制允许程序在运行时取得任何一个已知名称的class的内部信息，包括包括其modifiers(修饰符)，fields(属性)，methods(方法)等，并可于运行时改变fields内容或调用methods。那么我们便可以更灵活的编写代码，代码可以在运行时装配，无需在组件之间进行源代码链接，降低代码的耦合度；还有动态代理的实现等等；但是需要注意的是反射使用不当会造成很高的资源消耗！

**①、得到 Class 的三种方式**

[复制代码](javascript:void(0);)

1 //1、通过对象调用 getClass() 方法来获取,通常应用在：比如你传过来一个 Object

2 // 类型的对象，而我不知道你具体是什么类，用这种方法

3 　　Person p1 = new Person();

4 　　Class c1 = p1.getClass();

5

6 //2、直接通过 类名.class 的方式得到,该方法最为安全可靠，程序性能更高

7 // 这说明任何一个类都有一个隐含的静态成员变量 class

8 　　Class c2 = Person.class;

9

10 //3、通过 Class 对象的 forName() 静态方法来获取，用的最多，

11 // 但可能抛出 ClassNotFoundException 异常

12 　　Class c3 = Class.forName("com.ys.reflex.Person");

需要注意的是：**一个类在 JVM 中只会有一个 Class 实例,**即我们对上面获取的 c1,c2,c3进行 equals 比较，发现都是true

②、**通过 Class 类获取成员变量、成员方法、接口、超类、构造方法等**

查阅 API 可以看到 Class 有很多方法：

　　getName()：获得类的完整名字。  
　　getFields()：获得类的public类型的属性。  
　　getDeclaredFields()：获得类的所有属性。包括private 声明的和继承类  
　　getMethods()：获得类的public类型的方法。  
　　getDeclaredMethods()：获得类的所有方法。包括private 声明的和继承类  
　　getMethod(String name, Class[] parameterTypes)：获得类的特定方法，name参数指定方法的名字，parameterTypes 参数指定方法的参数类型。  
　　getConstructors()：获得类的public类型的构造方法。  
　　getConstructor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes 参数指定构造方法的参数类型。  
　　newInstance()：通过类的不带参数的构造方法创建这个类的一个对象。

## JVM运行参数

### ****一、堆大小设置****（堆 = 年轻代 + 年老代+ 持久代）

* **-Xmx3550m：**设置JVM最大可用内存为3550M。
* **-Xms3550m：**设置JVM初始内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

　　若设置初始内存过大，比如-Xms8000m，这都超过了我电脑内存，则会报错

Error occurred during initialization of VM

Unable to allocate 256000KB bitmaps for parallel garbage collection for the requested 8192000KB heap

Error: Could not create the Java Virtual Machine.

Error: A fatal exception has occurred. Program will exit.

　　若设置初始内存过小，比如-Xms10k，则会报错

Error occurred during initialization of VM

Too small initial heap

　　若设置初始内存过小得让gc无法触发，比如-Xms1200k，则会报错。估算gc需要2500K的内存。

Error occurred during initialization of VM

GC triggered before VM initialization completed. Try increasing NewSize, current value 1536K.

* **-Xss128k：**设置每个线程的堆栈大小。（JDK5.0以后每个线程堆栈大小为1M）
* **-Xmn1024m：**设置年轻代大小为1024m。等效于同时配置下面两个。
  + -XX:NewSize=1024m：设置年轻代初始值为1024M。
  + -XX:MaxNewSize=1024m：设置年轻代最大值为1024M。
* ~~-XX:PermSize=256m：设置持久代初始值为256M。~~
* ~~-XX:MaxPermSize=256m：设置持久代最大值为256M。~~

　　JVM8.0已经没有持久代，若设置则会警告

Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=10m; support was removed in 8.0

### ****二、垃圾回收器设置****（串行收集器、并行收集器、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智能判断。

**串行收集器**

* -XX:+UseSerialGC：设置串行收集器。

**并行收集器**（吞吐量优先）

* -XX:+UseParallelGC：设置年轻代为并行收集器。（此时年老代仍然为串行）
* -XX:+UseParallelOldGC：配置年老代为并行收集。
* -XX:ParallelGCThreads=20：配置并行收集器的线程数。
* -XX:MaxGCPauseMillis=100：设置每次年轻代垃圾回收的最长时间（单位毫秒）。如果无法满足此时间，JVM会自动调整年轻代大小，以满足此时间。
* -XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动调整年轻代Eden区大小和Survivor区大小的比例，以达成目标系统规定的最低响应时间或者收集频率等指标。此参数建议在使用并行收集器时，一直打开。

**并发收集器**（响应时间优先）

* **-XX:+UseConcMarkSweepGC：**即CMS收集，设置年老代为并发收集。
* -XX:+UseParNewGC：设置年轻代为并发收集。JDK5.0以上JVM会自行设置，无需再设。
* -XX:CMSFullGCsBeforeCompaction=0：每次Full GC后立刻开始压缩和整理内存。
* -XX:+UseCMSCompactAtFullCollection：打开内存空间的压缩和整理，在Full GC后执行。
* -XX:+CMSIncrementalMode：设置为增量收集模式。一般适用于单CPU情况。
* -XX:CMSInitiatingOccupancyFraction=70：表示年老代内存空间使用到70%时就开始执行CMS收集，以确保年老代有足够的空间接纳来自年轻代的对象，避免Full GC的发生。

**其它垃圾回收参数**

* -XX:+ScavengeBeforeFullGC：年轻代GC优于Full GC执行。
* **-XX:-DisableExplicitGC：**不响应 System.gc() 代码。
* -XX:+UseThreadPriorities：启用本地线程优先级API。即使 java.lang.Thread.setPriority() 生效，不启用则无效。
* -XX:SoftRefLRUPolicyMSPerMB=0：软引用对象在最后一次被访问后能存活0毫秒（JVM默认为1000毫秒）。
* -XX:TargetSurvivorRatio=90：允许90%的Survivor区被占用（JVM默认为50%）。提高对于Survivor区的使用率。

### ****三、辅助信息参数设置****

* -XX:-CITime：打印消耗在JIT编译的时间。
* -XX:ErrorFile=./hs\_err\_pid.log：保存错误日志或数据到指定文件中。
* -XX:HeapDumpPath=./java\_pid.hprof：指定Dump堆内存时的路径。
* -XX:-HeapDumpOnOutOfMemoryError：当首次遭遇内存溢出时Dump出此时的堆内存。
* -XX:OnError=";"：出现致命ERROR后运行自定义命令。
* -XX:OnOutOfMemoryError=";"：当首次遭遇内存溢出时执行自定义命令。
* -XX:-PrintClassHistogram：按下 Ctrl+Break 后打印堆内存中类实例的柱状信息，同JDK的 jmap -histo 命令。
* -XX:-PrintConcurrentLocks：按下 Ctrl+Break 后打印线程栈中并发锁的相关信息，同JDK的 jstack -l 命令。
* -XX:-PrintCompilation：当一个方法被编译时打印相关信息。
* -XX:-PrintGC：每次GC时打印相关信息。
* **-XX:-PrintGCDetails：**每次GC时打印详细信息。
* -XX:-PrintGCTimeStamps：打印每次GC的时间戳。
* -XX:-TraceClassLoading：跟踪类的加载信息。
* -XX:-TraceClassLoadingPreorder：跟踪被引用到的所有类的加载信息。
* -XX:-TraceClassResolution：跟踪常量池。
* -XX:-TraceClassUnloading：跟踪类的卸载信息。

# JMM

# 多线程

## 创建线程

java中创建线程的三种方法以及区别

Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。Java可以用三种方式来创建线程，如下所示：

1）继承Thread类创建线程

2）实现Runnable接口创建线程

3）使用Callable和Future创建线程

下面让我们分别来看看这三种创建线程的方法。

------------------------继承Thread类创建线程---------------------

通过继承Thread类来创建并启动多线程的一般步骤如下

1】d定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的任务，run()方法也称为线程执行体。

2】创建Thread子类的实例，也就是创建了线程对象

3】启动线程，即调用线程的start()方法

代码实例

public class MyThread extends Thread{//继承Thread类

　　public void run(){

　　//重写run方法

　　}

}

public class Main {

　　public static void main(String[] args){

　　　　new MyThread().start();//创建并启动线程

　　}

}

------------------------实现Runnable接口创建线程---------------------

通过实现Runnable接口创建并启动线程一般步骤如下：

1】定义Runnable接口的实现类，一样要重写run()方法，这个run（）方法和Thread中的run()方法一样是线程的执行体

2】创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象

3】第三部依然是通过调用线程对象的start()方法来启动线程

代码实例：

public class MyThread2 implements Runnable {//实现Runnable接口

　　public void run(){

　　//重写run方法

　　}

}

public class Main {

　　public static void main(String[] args){

　　　　//创建并启动线程

　　　　MyThread2 myThread=new MyThread2();

　　　　Thread thread=new Thread(myThread);

　　　　thread().start();

　　　　//或者 new Thread(new MyThread2()).start();

　　}

}

------------------------使用Callable和Future创建线程---------------------

和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()方法功能要强大。

》call()方法可以有返回值

》call()方法可以声明抛出异常

Java5提供了Future接口来代表Callable接口里call()方法的返回值，并且为Future接口提供了一个实现类FutureTask，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Thread类的target。在Future接口里定义了几个公共方法来控制它关联的Callable任务。

>boolean cancel(boolean mayInterruptIfRunning)：视图取消该Future里面关联的Callable任务

>V get()：返回Callable里call（）方法的返回值，调用这个方法会导致程序阻塞，必须等到子线程结束后才会得到返回值

>V get(long timeout,TimeUnit unit)：返回Callable里call（）方法的返回值，最多阻塞timeout时间，经过指定时间没有返回抛出TimeoutException

>boolean isDone()：若Callable任务完成，返回True

>boolean isCancelled()：如果在Callable任务正常完成前被取消，返回True

介绍了相关的概念之后，创建并启动有返回值的线程的步骤如下：

1】创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例（从java8开始可以直接使用Lambda表达式创建Callable对象）。

2】使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值

3】使用FutureTask对象作为Thread对象的target创建并启动线程（因为FutureTask实现了Runnable接口）

4】调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

代码实例：

public class Main {

　　public static void main(String[] args){

　　　MyThread3 th=new MyThread3();

　　　//使用Lambda表达式创建Callable对象

　　 //使用FutureTask类来包装Callable对象

　　　FutureTask<Integer> future=new FutureTask<Integer>(

　　　　(Callable<Integer>)()->{

　　　　　　return 5;

　　　　}

　　 );

　　　new Thread(task,"有返回值的线程").start();//实质上还是以Callable对象来创建并启动线程

　　 try{

　　　　System.out.println("子线程的返回值："+future.get());//get()方法会阻塞，直到子线程执行结束才返回

　　 }catch(Exception e){

　　　　ex.printStackTrace();

　　　}

　　}

}

--------------------------------------三种创建线程方法对比--------------------------------------

实现Runnable和实现Callable接口的方式基本相同，不过是后者执行call()方法有返回值，后者线程执行体run()方法无返回值，因此可以把这两种方式归为一种这种方式与继承Thread类的方法之间的差别如下：

1、线程只是实现Runnable或实现Callable接口，还可以继承其他类。

2、这种方式下，多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。

3、但是编程稍微复杂，如果需要访问当前线程，必须调用Thread.currentThread()方法。

4、继承Thread类的线程类不能再继承其他父类（Java单继承决定）。

注：一般推荐采用实现接口的方式来创建多线程

## 线程状态以及转换

<https://www.cnblogs.com/happy-coder/p/6587092.html>

### 说明

线程共包括以下5种状态。  
1. **新建状态(New)**: 线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。  
2. **就绪状态(Runnable)**: 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。  
3. **运行状态(Running)** : 线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。  
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：  
    (01) 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。  
    (02) 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。  
    (03) 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。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。

### ****wait(), notify(), notifyAll()等方法介绍****

在Object.java中，定义了wait(), notify()和notifyAll()等接口。wait()的作用是让当前线程进入等待状态，同时，wait()也会让当前线程释放它所持有的锁。而notify()和notifyAll()的作用，则是唤醒当前对象上的等待线程；notify()是唤醒单个线程，而notifyAll()是唤醒所有的线程。

Object类中关于等待/唤醒的API详细信息如下：  
**notify()**-- 唤醒在此对象监视器上等待的单个线程。  
**notifyAll()**  -- 唤醒在此对象监视器上等待的所有线程。  
**wait()**                               -- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法”，当前线程被唤醒(进入“就绪状态”)。  
**wait(long timeout)**                -- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量”，当前线程被唤醒(进入“就绪状态”)。  
**wait(long timeout, int nanos)**-- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量”，当前线程被唤醒(进入“就绪状态”)。

### ****为什么notify(), wait()等函数定义在Object中，而不是Thread中****

Object中的wait(), notify()等函数，和synchronized一样，会对“对象的同步锁”进行操作。

wait()会使“当前线程”等待，因为线程进入等待状态，所以线程应该释放它锁持有的“同步锁”，否则其它线程获取不到该“同步锁”而无法运行！  
OK，线程调用wait()之后，会释放它锁持有的“同步锁”；而且，根据前面的介绍，我们知道：等待线程可以被notify()或notifyAll()唤醒。现在，请思考一个问题：notify()是依据什么唤醒等待线程的？或者说，wait()等待线程和notify()之间是通过什么关联起来的？答案是：依据“对象的同步锁”。

负责唤醒等待线程的那个线程(我们称为“**唤醒线程**”)，它只有在获取“该对象的同步锁”(**这里的同步锁必须和等待线程的同步锁是同一个**)，并且调用notify()或notifyAll()方法之后，才能唤醒等待线程。虽然，等待线程被唤醒；但是，它不能立刻执行，因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后，等待线程才能获取到“对象的同步锁”进而继续运行。

总之，notify(), wait()依赖于“同步锁”，而“同步锁”是对象锁持有，并且每个对象有且仅有一个！这就是为什么notify(), wait()等函数定义在Object类，而不是Thread类中的原因。

### ****yield()介绍****

**yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后，其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！**

### ****yield() 与 wait()的比较****

**我们知道，**[**wait()**](http://www.cnblogs.com/skywang12345/p/3479224.html)**的作用是让当前线程由“运行状态”进入“等待(阻塞)状态”的同时，也会释放同步锁。而yield()的作用是让步，它也会让当前线程离开“运行状态”。它们的区别是：  
(01) wait()是让线程由“运行状态”进入到“等待(阻塞)状态”，而不yield()是让线程由“运行状态”进入到“就绪状态”。  
(02) wait()是会线程释放它所持有对象的同步锁，而yield()方法不会释放锁。**

### sleep()介绍

sleep() 定义在Thread.java中。  
sleep() 的作用是让当前线程休眠，即当前线程会从“[运行状态](http://www.cnblogs.com/skywang12345/p/3479024.html)”进入到“[休眠(阻塞)状态](http://www.cnblogs.com/skywang12345/p/3479024.html)”。sleep()会指定休眠时间，线程休眠的时间会大于/等于该休眠时间；在线程重新被唤醒时，它会由“[阻塞状态](http://www.cnblogs.com/skywang12345/p/3479024.html)”变成“[就绪状态](http://www.cnblogs.com/skywang12345/p/3479024.html)”，从而等待cpu的调度执行。

### ****sleep() 与 wait()的比较****

**我们知道，wait()的作用是让当前线程由“运行状态”进入“等待(阻塞)状态”的同时，也会释放同步锁。而sleep()的作用是也是让当前线程由“运行状态”进入到“休眠(阻塞)状态”。  
但是，wait()会释放对象的同步锁，而sleep()则不会释放锁。**

## ThreadPoolExecutor创建线程池、以及如何计算大小、拒绝策略

### 为什么要自己定义线程池

虽然jdk提供了几种常用特性的线程池给我们，但是很多时候，我还是需要自己去自定义自己需要特征的线程池。并且阿里巴巴规范手册里面，就是不建议使用jdk的线程池，而是建议程序员手动创建线程池。

为什么呢？原文如下

【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

说明：Executors 返回的线程池对象的弊端如下：

1、FixedThreadPool 和 SingleThreadPool:

允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。

2、CachedThreadPool 和 ScheduledThreadPool:

允许的创建线程数量为 Integer.MAX\_VALUE，可能会创建大量的线程，从而导致 OOM

但是除此之外，自己通过ThreadPoolExecutor定义线程池还有很多好处。

比如说，

1.自己根据需求定义自己的拒绝策略，如果线程过多，任务过多 如何处理。

2.补充完善的线程信息，比如线程名，这一点在将来如果出现线上bug的时候，你会感谢自己，因为你绝不想在线上看到什么threa-1 threa-2 等这种线程爆出的错误，而且是看到自己 “处理xxx线程 错误”，可以一眼看到

3.可以通过ThreadPoolExecutor的beforExecute(),

afterExecute()和terminated()方法去拓展对线程池运行前，运行后，结束后等不同阶段的控制。比如说通过拓展打印日志输出一些有用的调试信息。在故障诊断是非常有用的。

4.可以通过自定义线程创建，可以自定义线程名称，组，优先级等信息，包括设置为守护线程等，根据需求。

---------------------

### ThreadPoolExecutor

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

1.corePoolSize 指定了线程池里的线程数量

2.maximumPoolSize 指定了线程池里的最大线程数量

3.keepAliveTime 当线程池线程数量大于corePoolSize时候，多出来的空闲线程，多长时间会被销毁。

4.unit 时间单位

5.workQueue 任务队列，用于存放提交但是尚未被执行的任务。

6.threadFactory 线程工厂，用于创建线程，一般可以用默认的

7.handler 拒绝策略，当任务过多时候，如何拒绝任务。

主要是workQueue和handler的差异比较大

workQueue指被提交但未执行的任务队列，它是一个BlockingQueue接口的对象，仅用于存放Runnable对象。

ThreadPoolExecutor的构造函数中，可使用以下几种BlockingQueue

1.直接提交队列： 即SynchronousQueue ,这是一个比较特殊的BlockKingQueue， SynchronousQueue没有容量，每一个插入操作都要等待对应的删除操作，反之 一个删除操作都要等待对应的插入操作。 也就是如果使用SynchronousQueue，提交的任务不会被真实保存，而是将新任务交给空闲线程执行，如果没有空闲线程，则创建线程，如果线程数都已经大于最大线程数，则执行拒绝策略。使用这种队列，需要将maximumPoolSize设置的非常大，不然容易执行拒绝策略。

2.有界任务队列。 有界任务队列可以使用ArrayBlockingQueue实现。需要给一个容量参数表示该队列的最大值。当有新任务进来时，如果当前线程数小于corePoolSize，则会创建新线程执行任务。如果大于，则会将任务放到任务队列中，如果任务队列满了，在当前线程小于将maximumPoolSize的情况下，将会创建新线程，如果大于maximumPoolSize，则执行拒绝策略。

也就是，一阶段，当线程数小于coresize的时候，创建线程；二阶段，当线程任务数大于coresize的时候，放入到队列中；三阶段，队列满，但是还没大于maxsize的时候，创建新线程。 四阶段，队列满，线程数也大于了maxsize, 则执行拒绝策略。

可以发现，有界任务队列，会大概率将任务保持在coresize上，只有队列满了，也就是任务非常繁忙的时候，会到达maxsie。

3.无界任务队列。

使用linkedBlockingQueue实现，队列最大长度限制为integer.MAX。无界任务队列，不存在任务入队失败的情况， 当任务过来时候，如果线程数小于coresize ，则创建线程，如果大于，则放入到任务队列里面。也就是，线程数几乎会一直维持在coresize大小。FixedThreadPool和singleThreadPool即是如此。 风险在于，如果任务队列里面任务堆积过多，可能导致内存不足。

4.优先级任务队列。使用PrioriBlockingQueue ，特殊的无界队列，和普通的先进先出队列不同，它是优先级高的先出。

拒绝策略：

AbortPolicy:丢弃任务并抛出RejectedExecutionException

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。

DiscardOldestPolicy：丢弃队列中最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。

DiscardPolicy：丢弃任务，不做任何处理。

### 线程池的任务处理策略：

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

### 线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

### 如何选择线程池数量

**线程池的大小决定着系统的性能，过大或者过小的线程池数量都无法发挥最优的系统性能。**

当然线程池的大小也不需要做的太过于精确，只需要避免过大和过小的情况。一般来说，确定线程池的大小需要考虑CPU的数量，内存大小，任务是计算密集型还是IO密集型等因素

**NCPU = CPU的数量**

**UCPU = 期望对CPU的使用率 0 ≤ UCPU ≤ 1**

**W/C = 等待时间与计算时间的比率**

**如果希望处理器达到理想的使用率，那么线程池的最优大小为：**

**线程池大小=NCPU \*UCPU(1+W/C)**

在Java中使用

int ncpus = Runtime.getRuntime().availableProcessors();

获取CPU的数量。

要想合理的配置线程池的大小，首先得分析任务的特性，可以从以下几个角度分析：

1. 任务的性质：CPU密集型任务、IO密集型任务、混合型任务。
2. 任务的优先级：高、中、低。
3. 任务的执行时间：长、中、短。
4. 任务的依赖性：是否依赖其他系统资源，如数据库连接等。

性质不同的任务可以交给不同规模的线程池执行。

对于不同性质的任务来说，CPU密集型任务应配置尽可能小的线程，如配置CPU个数+1的线程数，IO密集型任务应配置尽可能多的线程，因为IO操作不占用CPU，不要让CPU闲下来，应加大线程数量，如配置两倍CPU个数+1，而对于混合型的任务，如果可以拆分，拆分成IO密集型和CPU密集型分别处理，前提是两者运行的时间是差不多的，如果处理时间相差很大，则没必要拆分了。

若任务对其他系统资源有依赖，如某个任务依赖数据库的连接返回的结果，这时候等待的时间越长，则CPU空闲的时间越长，那么线程数量应设置得越大，才能更好的利用CPU。  
当然具体合理线程池值大小，需要结合系统实际情况，在大量的尝试下比较才能得出，以上只是前人总结的规律。

在这篇[如何合理地估算线程池大小？](http://ifeve.com/how-to-calculate-threadpool-size/)文章中发现了一个估算合理值的公式

最佳线程数目 = （（线程等待时间+线程CPU时间）/线程CPU时间 ）\* CPU数目

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

可以得出一个结论：  
**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。**  
以上公式与之前的CPU和IO密集型任务设置线程数基本吻合。

### ****线程池工厂****

Executors的线程池如果不指定线程工厂会使用Executors中的DefaultThreadFactory,默认线程池工厂创建的线程都是非守护线程。

使用自定义的线程工厂可以做很多事情，比如可以跟踪线程池在何时创建了多少线程，也可以自定义线程名称和优先级。如果将

新建的线程都设置成守护线程，当主线程退出后，将会强制销毁线程池。

### 扩展线程池

ThreadPoolExecutor是可以拓展的，它提供了几个可以在子类中改写的方法：beforeExecute,afterExecute和terimated。

在执行任务的线程中将调用beforeExecute和afterExecute,这些方法中还可以添加日志，计时，监视或统计收集的功能，

还可以用来输出有用的调试信息，帮助系统诊断故障

### ****手动创建线程池有几个注意点****

1.**任务独立。**如何任务依赖于其他任务，那么可能产生死锁。例如某个任务等待另一个任务的返回值或执行结果，那么除非线程池足够大，否则将发生线程饥饿死锁。

2.**合理配置阻塞时间过长的任务。**如果任务阻塞时间过长，那么即使不出现死锁，线程池的性能也会变得很糟糕。在Java并发包里可阻塞方法都同时定义了限时方式和不限时方式。例如

Thread.join,BlockingQueue.put,CountDownLatch.await等，如果任务超时，则标识任务失败，然后中止任务或者将任务放回队列以便随后执行，这样，无论任务的最终结果是否成功，这种办法都能够保证任务总能继续执行下去。

3.**设置合理的线程池大小。**只需要避免过大或者过小的情况即可，上文的公式**线程池大小=NCPU \*UCPU(1+W/C)**。

4.**选择合适的阻塞队列。**newFixedThreadPool和newSingleThreadExecutor都使用了无界的阻塞队列，无界阻塞队列会有消耗很大的内存，如果使用了有界阻塞队列，它会规避内存占用过大的问题，但是当任务填满有界阻塞队列，新的任务该怎么办？在使用有界队列是，需要选择合适的拒绝策略，队列的大小和线程池的大小必须一起调节。对于非常大的或者无界的线程池，可以使用SynchronousQueue来避免任务排队，以直接将任务从生产者提交到工作者线程。

## 线程池中任务执行完后什么状态

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

## Synchronized和reentrantlock的区别

1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。reentrantlock提供能够中断等待锁的线程的机制，lock.lockInterruptibly(),Lock还提供了更强大的功能，它的tryLock()方法可以采用非阻塞的方式去获取锁。

2.公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

3、用法不一样。在需要同步的对象中加入synchronized控制，synchronized既可以加载方法的前面，也可以加在特定代码块中，括号中表示需要锁的对象。而Lock需要显示地指定起始位置和终止位置。synchronized是托管给Jvm执行的，自动解锁，而Lock的锁定是通过代码实现的

4、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的发生；而Lock在发生异常时，如果没有主动通过unlock()去释放锁，则很有可能造成死锁现象，因此使用Lock时需要在finally块中释放锁

5、通过Lock可以知道有没有成功获取锁，而synchronized却无法办到

## Synchronized在JVM中的实现原理

## ThreadLocal的作用和弊端

Threadlocal源码：

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

ThreadLocalMap getMap(Thread t) {

return t.threadLocals;

}

应用：

private static final ThreadLocal<String> threadLocalA = new ThreadLocal<>();

private static final ThreadLocal<String> threadLocalB = new ThreadLocal<>();

/\*\*

\* 在调用的线程的map中存入key为ThreadLocal本身，value为在该线程设置的值

\* @param value

\*/

public static void setValueA(String value){

threadLocalA.set(value);

}

public static String getValueA(){

return threadLocalA.get();

### 实现原理

首先 ThreadLocal 是一个泛型类，保证可以接受任何类型的对象。

因为一个线程内可以存在多个 ThreadLocal 对象，所以其实是 ThreadLocal 内部维护了一个 Map ，这个 Map 不是直接使用的 HashMap ，而是 ThreadLocal 实现的一个叫做 ThreadLocalMap 的静态内部类。而我们使用的 get()、set() 方法其实都是调用了这个 ThreadLocalMap 类对应的 get()、set() 方法。例如下面的 set 方法：

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

调用 ThreadLocal 的 set 方法时，首先获取到了当前线程，然后获取当前线程维护的 ThreadLocalMap 对象，最后在ThreadLocalMap 实例中添加上。如果 ThreadLocalMap 实例不存在则初始化并赋初始值。

这里看到 set 方法的第一个参数是 this ，this即指的是当前的 ThreadLocal 对象，会看上看的代码就是指的 mLocal 这个对象。而在 ThreadLocalMap 的 set 方法中会根据当前 ThreadLocal 对象实例，做一些操作和判断，最终实现赋值操作（具体参考源码）。

所以说，最终的变量是放在了当前线程的 ThreadLocalMap 中，并不是存在 ThreadLocal 上，ThreadLocal 可以理解为只是一个中间工具，传递了变量值。

### 内存泄露

实际上 ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用，弱引用的特点是，如果这个对象只存在弱引用，那么在下一次垃圾回收的时候必然会被清理掉。

所以如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候会被清理掉的，这样一来 ThreadLocalMap 中使用这个 ThreadLocal 的 key 也会被清理掉。但是，value 是强引用，不会被清理，这样一来就会出现 key 为 null 的 value。

ThreadLocalMap 实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。如果说会出现内存泄漏，那只有在出现了 key 为 null 的记录后，没有手动调用 remove() 方法，并且之后也不再调用 get()、set()、remove() 方法的情况下。

为了更好的避免这种情况的发生我们使用ThreadLocal时遵守以下两个小原则:

    ①ThreadLocal申明为private static final。

         Private与final 尽可能不让他人修改变更引用，

         Static 表示为类属性，只有在程序结束才会被回收。

    ②ThreadLocal使用后务必调用remove方法。

        最简单有效的方法是使用后将其移除。

### 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：

[复制代码](javascript:void(0);)

private static ThreadLocal<Connection> connectionHolder = new ThreadLocal<Connection>() {

public Connection initialValue() {

return DriverManager.getConnection(DB\_URL);

}

};

public static Connection getConnection() {

return connectionHolder.get();

}

[复制代码](javascript:void(0);)

Session管理：

[复制代码](javascript:void(0);)

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;

}

### InheritableThreadLocal

InheritableThreadLocal主要用于子线程创建时，需要自动继承父线程的ThreadLocal变量，方便必要信息的进一步传递。

InheritableThreadLocal类重写了ThreadLocal的3个函数：

/\*\*

\* 该函数在父线程创建子线程，向子线程复制InheritableThreadLocal变量时使用

\*/

protected T childValue(T parentValue) {

return parentValue;

}

/\*\*

\* 由于重写了getMap，操作InheritableThreadLocal时，

\* 将只影响Thread类中的inheritableThreadLocals变量，

\* 与threadLocals变量不再有关系

\*/

ThreadLocalMap getMap(Thread t) {

return t.inheritableThreadLocals;

}

/\*\*

\* 类似于getMap，操作InheritableThreadLocal时，

\* 将只影响Thread类中的inheritableThreadLocals变量，

\* 与threadLocals变量不再有关系

\*/

void createMap(Thread t, T firstValue) {

t.inheritableThreadLocals = new ThreadLocalMap(this, firstValue);

}

## Volatile

**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。

## 公平锁和非公平锁

ReentrantLock 的公平锁

ReentrantLock 默认采用非公平锁，除非在构造方法中传入参数 true 。

//默认

public ReentrantLock() {

sync = new NonfairSync();

}

//传入true or false

public ReentrantLock(boolean fair) {

sync = fair ? new FairSync() : new NonfairSync();

}

总结

非公平锁和公平锁的两处不同：

非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

## 线程安全的类

Threadlocal

Hashtable、ConcurrentHashMap、CopyOnWriteArrayList、ConcurrentLinkedList、BlockingQueue、vector、Stack、CopyOnWriteArraySet

SpringBuffer

## 并发包下工具类

## 进程和线程的区别、通信机制

进程是资源（CPU、内存等）分配的基本单位，它是程序执行时的一个实例。程序运行时系统就会创建一个进程，并为它分配资源，然后把该进程放入进程就绪队列，进程调度器选中它的时候就会为它分配CPU时间，程序开始真正运行。

线程是程序执行时的最小单位，它是进程的一个执行流，是CPU调度和分派的基本单位，一个进程可以由很多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。线程由CPU独立调度执行，在多CPU环境下就允许多个线程同时运行。同样多线程也可以实现并发操作，每个请求分配一个线程来处理。

一个程序至少一个进程，一个进程至少一个线程。

### 线程和进程各自有什么区别和优劣呢？

* 进程是资源分配的最小单位，线程是程序执行的最小单位。
* 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。
* 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。
* 但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

### 进程通信

在 linux 下进程间通信的几种主要手段简介：

1. **管道（Pipe）及有名管道（named pipe）**：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；
2. **信号（Signal）**：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）；
3. **消息队列（Message）**：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
4. \*\* 共享内存\*\*：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
5. **信号量（semaphore）**：主要作为进程间以及同一进程不同线程之间的同步手段。
6. **套接口（Socket）**：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

各种通信方式的比较和优缺点：

1. 管道：速度慢，容量有限，只有父子进程能通讯
2. 有名管道（named pipe）：任何进程间都能通讯，但速度慢
3. 消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题
4. 信号量：不能传递复杂消息，只能用来同步
5. 共享内存：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

### 线程通信

线程间的通信方式

# 锁机制：包括互斥锁、条件变量、读写锁  
\*互斥锁提供了以排他方式防止数据结构被并发修改的方法。  
\*读写锁允许多个线程同时读共享数据，而对写操作是互斥的。  
\*条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。  
# 信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量  
# 信号机制(Signal)：类似进程间的信号处理  
线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。

## 多线程中的锁

<https://blog.csdn.net/tyyj90/article/details/78236053>

* 公平锁/非公平锁
* 可重入锁
* 独享锁/共享锁
* 互斥锁/读写锁
* 乐观锁/悲观锁
* 分段锁
* 偏向锁/轻量级锁/重量级锁
* 自旋锁

## 多线程运用实例

## 多线程中runable和callable的区别

## 两个方法均有sycrinised关键字 两个线程分别去执行这两个方法 是否有可能会造成死锁

## 线程阻塞后唤醒

<https://www.cnblogs.com/heyq/p/9013783.html>

***方式1：***早期JAVA采用suspend()、resume()对线程进行阻塞与唤醒，但这种方式产生死锁的风险很大，因为线程被挂起以后不会释放锁，可能与其他线程、主线程产生死锁，如：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

***方式2：***wait、notify形式通过一个object作为信号，object的wait()方法是锁门的动作，notify()、notifyAll()是开门的动作，某一线程一旦关上门后其他线程都将阻塞，直到别的线程打开门。notify()准许阻塞的一个线程通过，notifyAll()允许所有线程通过。如下例子：主线程分别启动两个线程，随后通知子线程暂停等待，再逐个唤醒后线程抛异常退出。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

 wait、notify使用要点：

1、对象操作都需要加同步synchronized；

2、线程需要阻塞的地方调用对象的wait方法；

存在的不足：面向对象的阻塞是阻塞当前线程，而唤醒的是随机的一个线程或者所有线程，偏重线程间的通信；同时某一线程在被另一线程notify之前必须要保证此线程已经执行到wait等待点，错过notify则可能永远都在等待。

***方式3：***LockSupport提供的park和unpark方法，提供避免死锁和竞态条件，很好地代替suspend和resume组合。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

park与unpark方法控制的颗粒度更加细小，能准确决定线程在某个点停止，进而避免死锁的产生。

park与unpark引入了许可机制，许可逻辑为：

①park将许可在等于0的时候阻塞，等于1的时候返回并将许可减为0；

②unpark尝试唤醒线程，许可加1。根据这两个逻辑，对于同一条线程，park与unpark先后操作的顺序似乎并不影响程序正确地执行，假如先执行unpark操作，许可则为1，之后再执行park操作，此时因为许可等于1直接返回往下执行，并不执行阻塞操作。

park与unpark组合真正解耦了线程之间的同步，不再需要另外的对象变量存储状态，并且也不需要考虑同步锁，wait与notify要保证必须有锁才能执行，而且执行notify操作释放锁后还要将当前线程扔进该对象锁的等待队列，LockSupport则完全不用考虑对象、锁、等待队列等问题。

总结：suspend()、resume()已经被deprecated，不建议使用。wait、notify需要对对象加同步，性能有折扣。LockSupport则完全不用考虑对象、锁、等待队列。

## Condition

## 信号量Semaphore

## ReadWriteLock读写锁

## 倒计时器CountDownLatch

## 循环栅栏CyclicBarrier

## 线程阻塞工具类LockSupport

## ThreadPoolTaskExecutor

## Executors

## ExecutorService

## Executor

## Future

# 集合

## HashMap数据结构、扩容、构造

**数据结构**：HashMap的底层主要是基于数组和链表来实现的，

table就是HashMap的存储结构，显然这是一个数组，数组的每一个元素都是一个条目（Entry），Entry是HashMap中的一个内部类，它有如下4个属性：final K key;V value;Entry<K,V> next;int hash。分别为键、值、指向下一个链表结点的指针、散列（哈希）值，HashMap中主要是通过key的hashCode来计算hash值的，hash冲突时通过key的equals（）方法找到链表中的元素，DEFAULT\_INITIAL\_CAPACITY是HashMap默认的初始化桶数量16，对于HashMap中桶数量的值必须是2的N次幂，而且这个是HashMap强制规定的。这样做的原因就是因为计算机进行2次幂的运算是非常高效的，MAXIMUM\_CAPACITY是HashMap中散列桶数量的最大值，最大值为2的32次幂，即1073741824，默认的负载因子，如果在在创建HashMap的构造函数中没有指定负载因子，则指定该HashMap的默认负载因子为0.75，这意味着当HashMap中条目的数量达到了条目数量75%时，HashMap将进行resize操作以增加桶的数量，size表示HashMap中条目（即键-值对）的数量，threshold是HashMap的重构阈值，它的值为容量和负载因子的乘积。在HashMap中所有桶中条目的总数量达到了这个重构阈值之后，HashMap将进行resize操作以自动扩容，modCount表示HashMap被结构化更新的次数，比如插入、删除、清空等会更新HashMap结构的操作次数。

**扩容：**容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。负载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。其中加载因子是表示Hash表中元素的填满的程度.若:加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.反之,加载因子越小,填满的元素越少,  
好处是:冲突的机会减小了,但:空间浪费多了.冲突的机会越大,则查找的成本越高.反之,查找的成本越小.因而,查找时间就越小.因此,必须在 "冲突的机会"与"空间利用率"之间寻找一种平衡与折衷. 这种平衡与折衷本质上是数据结构中有名的"时-空"矛盾的平衡与折衷.

　　如果机器内存足够，并且想要提高查询速度的话可以将加载因子设置小一点；相反如果机器内存紧张，并且对查询速度没有什么要求的话可以将加载因子设置大一点。不过一般我们都不用去设置它，让它取默认值0.75就好了。

**构造：**HashMap提供了4个构造方法，按照它们在源码中的位置顺序从上至下列出：

HashMap(int initialCapacity, float loadFactor)

HashMap(int initialCapacity)

HashMap()

HashMap(Map<? extends K, ? extends V> m)

1、我们先来分析第一个同时传递初始化容量参数和负载因子参数的源码，因为其它的3个构造方法都会调用这个构造方法

public HashMap(int initialCapacity, float loadFactor) {

//部分构造参数容错处理的源码已省略...

    /\*\*

     \* 根据传入的初始化容量计算该HashMap的容量（即桶的数量）

     \* 算法为：将capacity进行不断的左移，直至capacity大于或等于初始化容量

    \*/

    int capacity = 1;

    while (capacity < initialCapacity)

        capacity <<= 1;

    //负载因子初始化

    this.loadFactor = loadFactor;

    /\*\*

     \* 条目阈值的计算

     \* 算法：超出条目最大容量前取容量与负载因子的乘积作为条目阈值

     \*/

    threshold = (int)Math.min(capacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

    //创建数组（散列桶）

    table = new Entry[capacity];

    //计算是否对字符串键的HashMap使用备选哈希函数

    useAltHashing = sun.misc.VM.isBooted() &&

        (capacity >= Holder.ALTERNATIVE\_HASHING\_THRESHOLD);

init();//调用初始化方法，默认情况下什么也没做

}

**4、**(4) 下面是根据已有Map构造新HashMap的构造方法：

public HashMap(Map<? extends K, ? extends V> m) {

    /\*\*

     \* 取下面两个值的较大的值作为当前要构造的HashMap的初始容量

     \* 第1个值：用传入的Map的条目数量除以默认加载因子再加上1

     \* 第2个值：默认的初始化容量

     \*/

     this(Math.max((int) (m.size() / DEFAULT\_LOAD\_FACTOR) + 1,

          DEFAULT\_INITIAL\_CAPACITY), DEFAULT\_LOAD\_FACTOR);

     /\*\*

      \* 把传入的map里的所有条目放入当前已构造的HashMap中

      \* 关于putAllForCreate方法后面会作分析

     \*/

     putAllForCreate(m);

}

## HashMap resize后原来元素的位置如何变化

HashMap 扩容后，原来的元素，要么在原位置，要么在原位置+原数组长度 那个位置上。+2的N次幂位置

## Hashtable

HashMap基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

<https://blog.csdn.net/yu849893679/article/details/81530298>

父类不同、null值问题、线程安全性、遍历方式不同、初始容量不同、计算哈希值的方法不同

## 实现线程安全的HashMap

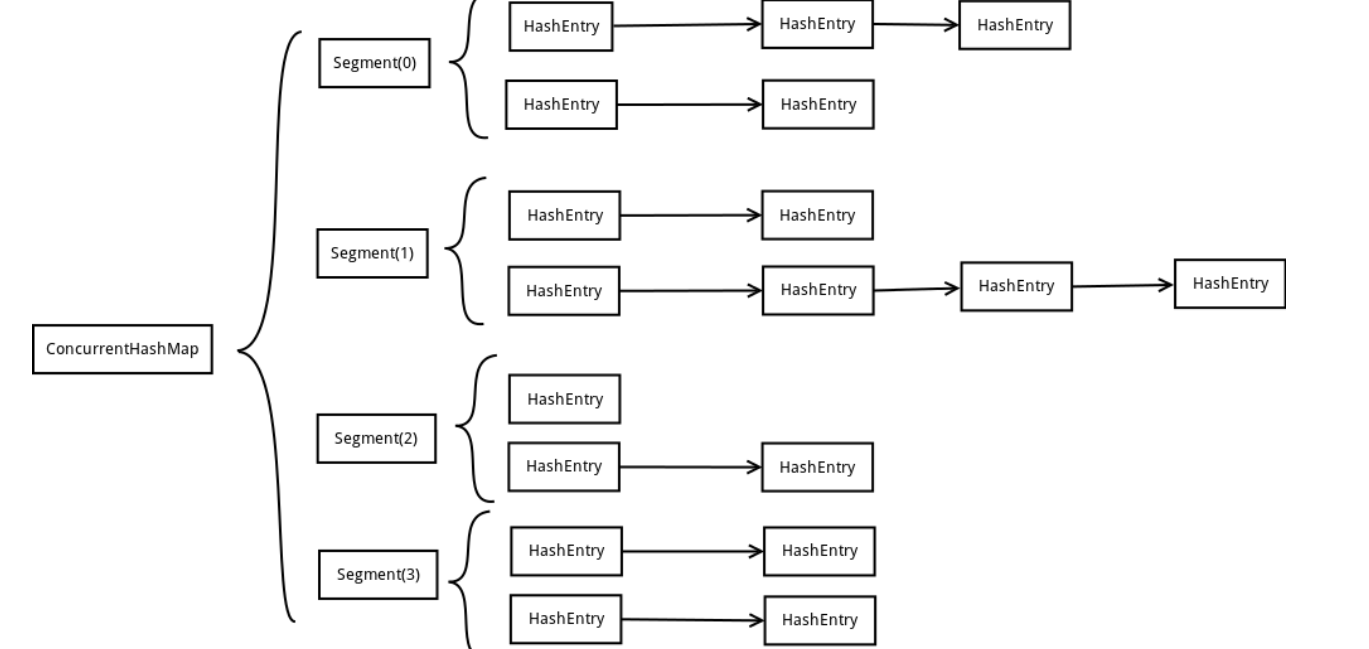
Hashtable

ConcurrentHashMap

Collections.synchronizedMap()

Hashtable和synchronizedMap源码中是使用synchronized来保证线程安全的, 所以当一个线程访问HashTable的同步方法时，其他线程如果也要访问同步方法，会被阻塞住。

## ConcurrentHashMap结构



ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作，第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部，因此，这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长，但是带来的好处是写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（

## 如何对map中的元素根据key进行排序

## ArrayList和LinkedList

ArrayList和LinkedList的大致区别：

1.ArrayList是实现了基于动态数组的数据结构，LinkedList是基于链表结构。

2.对于随机访问的get和set方法，ArrayList要优于LinkedList，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。

## List循环删除元素的问题

JAVA中循环遍历list有三种方式for循环、增强for循环（也就是常说的foreach循环）、iterator遍历。

 1、for循环遍历list

for(int i=0;i<list.size();i++){

if(list.get(i).equals("del"))

list.remove(i);

}

 　　这种方式的问题在于，删除某个元素后，list的大小发生了变化，而你的索引也在变化，所以会导致你在遍历的时候漏掉某些元素。比如当你删除第1个元素后，继续根据索引访问第2个元素时，因为删除的关系后面的元素都往前移动了一位，所以实际访问的是第3个元素。因此，这种方式可以用在删除特定的一个元素时使用，但不适合循环删除多个元素时使用。

2、增强for循环

for(String x:list){

if(x.equals("del"))

list.remove(x);

}

 　　这种方式的问题在于，删除元素后继续循环会报错误信息ConcurrentModificationException，因为元素在使用的时候发生了并发的修改，导致异常抛出。但是删除完毕马上使用break跳出，则不会触发报错。

3、iterator遍历

复制代码

Iterator<String> it = list.iterator();

while(it.hasNext()){

String x = it.next();

if(x.equals("del")){

it.remove();

}

}

复制代码

　　这种方式可以正常的循环及删除。但要注意的是，使用iterator的remove方法，如果用list的remove方法同样会报上面提到的ConcurrentModificationException错误。

　　总结：

　　（1）循环删除list中特定一个元素的，可以使用三种方式中的任意一种，但在使用中要注意上面分析的各个问题。

　　（2）循环删除list中多个元素的，应该使用迭代器iterator方式。

## CopyOnWriteArrayList

## ConcurrentLinkedQueue

## BlockingQueue

## ConcurrentSkipListMap

## 遍历Map

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

public class TestMap {

public static void main(String[] args) {

Map<Integer, String> map = new HashMap<Integer, String>();

map.put(1, "a");

map.put(2, "b");

map.put(3, "ab");

map.put(4, "ab");

map.put(4, "ab");// 和上面相同 ， 会自己筛选

System.out.println(map.size());

// 第一种：

/\*

\* Set<Integer> set = map.keySet(); //得到所有key的集合

\*

\* for (Integer in : set) { String str = map.get(in);

\* System.out.println(in + " " + str); }

\*/

System.out.println("第一种：通过Map.keySet遍历key和value：");

for (Integer in : map.keySet()) {

//map.keySet()返回的是所有key的值

String str = map.get(in);//得到每个key多对用value的值

System.out.println(in + " " + str);

}

// 第二种：

System.out.println("第二种：通过Map.entrySet使用iterator遍历key和value：");

Iterator<Map.Entry<Integer, String>> it = map.entrySet().iterator();

while (it.hasNext()) {

Map.Entry<Integer, String> entry = it.next();

System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue());

}

// 第三种：推荐，尤其是容量大时

System.out.println("第三种：通过Map.entrySet遍历key和value");

for (Map.Entry<Integer, String> entry : map.entrySet()) {

//Map.entry<Integer,String> 映射项（键-值对） 有几个方法：用上面的名字entry

//entry.getKey() ;entry.getValue(); entry.setValue();

//map.entrySet() 返回此映射中包含的映射关系的 Set视图。

System.out.println("key= " + entry.getKey() + " and value= "

+ entry.getValue());

}

// 第四种：

System.out.println("第四种：通过Map.values()遍历所有的value，但不能遍历key");

for (String v : map.values()) {

System.out.println("value= " + v);

}

}

}

# Java新特性

## JDK8新特性

### Stream流

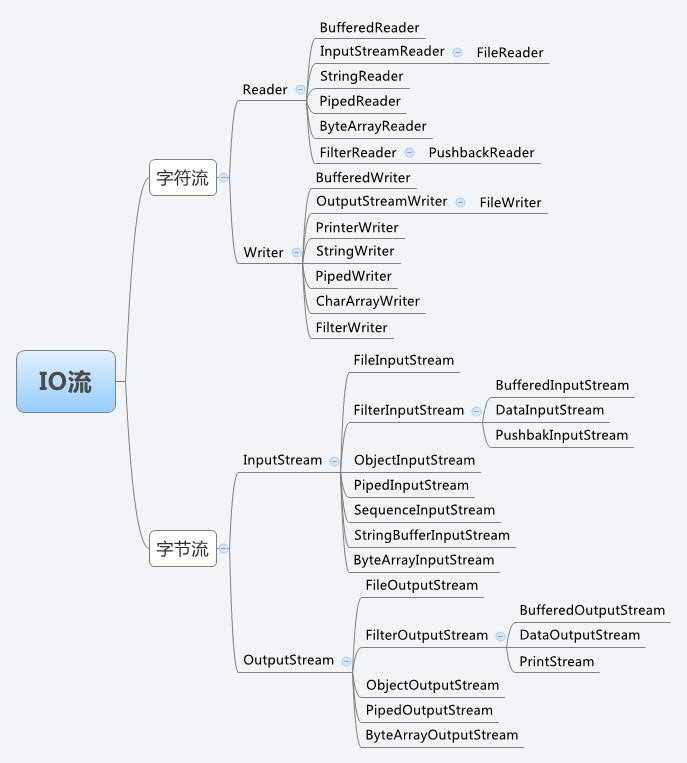
### 函数式接口

### HashMap变形

# IO

## IO 字节流、字符流

<https://www.cnblogs.com/QQ846300233/p/6046388.html>



### IO流的分类

根据处理数据类型的不同分为：字符流和字节流

根据数据流向不同分为：输入流和输出流

### 字符流和字节流的区别

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。

（3）字节流在操作的时候本身是不会用到缓冲区的，是文件本身的直接操作的；而字符流在操作的时候下后是会用到缓冲区的，是通过缓冲区来操作文件，我们将在下面验证这一点。

结论：优先选用字节流。首先因为硬盘上的所有文件都是以字节的形式进行传输或者保存的，包括图片等内容。但是字符只是在内存中才会形成的，所以在开发中，字节流使用广泛。

### 字节流和字符流的转换

<https://blog.csdn.net/qq_40856560/article/details/81808641>

1、InputStreamReader:字节到字符的桥梁

BufferedReader in= new BufferedReader(new InputStreamReader(System.in));

2、OutputStreamWriter:字符到字节的桥梁

Writer out= new BufferedWriter(new OutputStreamWriter(System.out));

1、字节流转字符流

package cn.yu.io;

import java.io.\*;

public class Demo1 {

/\*\*

\* 从控制台输入后写入文件

\*/

public static void main(String[] args) {

// 创建字节流对象 System.in 代表从控制台输入

InputStream in = System.in;

// 创建字符流对象

BufferedWriter bw = null;

BufferedReader br = null;

try {

// 实例化字符流对象 通过 InputStreamReader 将字节输入流转化成字符输入流

br = new BufferedReader(new InputStreamReader(in));

bw = new BufferedWriter(new FileWriter("a.txt"));

// 定义读取数据的行

String line = null;

// 读取数据

while ((line = br.readLine()) != null) {

// 如果输入的是"exit"就退出

if("exit".equals(line)){

break;

}

// 将数据写入文件

bw.write(line);

// 写入新的一行

bw.newLine();

// 刷新数据缓冲

bw.flush();

}

} catch (Exception e){

e.printStackTrace();

} finally {

  // 释放资源

try {

if(bw != null)

bw.close();

if (br != null)

br.close();

} catch (IOException e){

e.printStackTrace();

}

}

}

}

-2、字符流转字节流

package cn.yu.io;

import java.io.\*;

public class Demo2 {

/\*\*

\* 读取文件输出到控制台

\*/

public static void main(String[] args) {

// 定义字节输出流的对象

OutputStream out = System.out;

// 定义字节流的对象

BufferedReader br = null;

BufferedWriter bw = null;

try {

// 实例化字节输出流 使用

bw = new BufferedWriter(new OutputStreamWriter(out));

br = new BufferedReader(new FileReader("a.txt"));

// 定义读取行的字符串

String line = null;

// 读取数据

while ((line = br.readLine()) != null) {

// 输出到控制台

bw.write(line);

// 新的一行

bw.newLine();

// 刷新缓冲

bw.flush();

}

} catch (Exception e){

e.printStackTrace();

} finally {

// 释放资源

try {

if(bw != null)

bw.close();

if (br != null)

br.close();

}catch (Exception e){

e.printStackTrace();

}

}

}

}

## AIO、BIO、NIO

### Java中BIO、NIO、AIO的区别

Java中BIO、NIO、AIO的区别

* Java BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即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，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
* Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即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。
* Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为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、NIO、AIO适用场景分析

BIO、NIO、AIO适用场景分析:

* BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
* NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
* AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

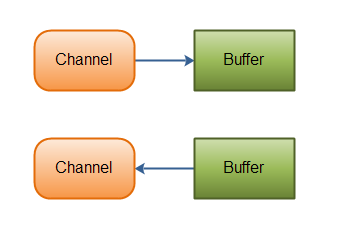
### NIO

<https://blog.csdn.net/charjay_lin/article/details/81810922>

<https://www.cnblogs.com/diegodu/p/6823855.html>

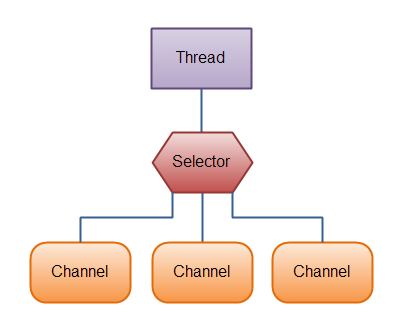
#### Java NIO的核心组成

##### 3.1、通道(Channel) 和 缓冲区(Buffer)

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。这里有个图示：  
[](http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/ba76486545ee41994605dd9102bc621e.png)

##### 3.2、多路复用器（Selector）

Selector允许单线程处理多个Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。

这是在一个单线程中使用一个Selector处理3个Channel的图示：  
[](http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/31c914f1dea54651d7af61ff9e3f279f.png)

要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

#### NIO给我们带来了些什么

最后总结一下到底NIO给我们带来了些什么：  
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，而是返回0  
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  
IO多路复用大大提高了Java网络应用的可伸缩性和实用性

# netty

# 数据结构

<https://www.cnblogs.com/zixinchen/archive/2017/08/30/7451316.html>

## 直接选择排序

直接选择排序(Straight Selection Sort)思想是：在第1趟中，从n个记录中找出关键字值最小的记录与第一个记录进行交换；再第2趟中，从第2个记录开始的n-1个记录中再选出关键字最小的记录与第二个记录交换1；以此类推，在第i趟中，从第i个记录开始的n-i+1个记录中选出关键字最小的记录与第i个记录交换，直到整个序列按关键字值有序为止。

import java.util.Arrays;

/\*\*

\* 直接选择排序算法

\*/

public class DirectSelectSort {

public static void sort(int[] array) {

int len = array.length;

for (int i = 0; i < len - 1; i++) {

int min = i;// 设当前第i条记录为最小值

// 找出(i+1)到(len-1)中最小值的下标

for (int j = i + 1; j < len; j++) {

if (array[j] < array[min]) {

min = j;// 记录最小值下标

}

}

// 将最小值记录与第i条记录交换

if (min != i) {

int temp = array[i];

array[i] = array[min];

array[min] = temp;

}

}

}

public static void main(String [] args){

int [] array = {45,89,699,52,14,236,58,7};

sort(array);

System.out.println(Arrays.toString(array));

}

}

## 二分排序

一.算法思想

　　1.二分插入排序的基本思想和插入排序一致；都是将某个元素插入到已经有序的序列的正确的位置；

　　2.和直接插入排序的最大区别是，元素A[i]的位置的方法不一样；直接插入排序是从A[i-1]往前一个个比较，从而找到正确的位置；而二分插入排序，利用前i-1个元素已经是有序的特点结合二分查找的特点，找到正确的位置，从而将A[i]插入，并保持新的序列依旧有序；

　　3.时间复杂度：

　　T(n) = O(n);

/\*\*

\* 二分法排序<br>

\* 根据排序原则，每次我们都是在一个有序序列中插入一个新的数字<br>

\* 那么我们可以将这个有序序列进行二分。<br>

\* 左游标left为0，右游标right为i-1(i是这个数字在原数组中的位置)<br>

\* middle初始为。<br>

\* 当left<=right时<br>

\* middle是left和right的中值。<br>

\* 我们作如下操作。如果array[i]的值比array[middle]值大。<br>

\* 那么我们就移动左游标令值为middle+1<br>

\* 负责就移动右游标为middle-1<br>

\* 移动完成后,我们需要将i-1到left之间的值进行依次向后移动给array[i]空出一个位置然后将array[i]插入

\* <p style="color:red">时间复杂度n</p>

\*/

public int[] binaryInsertSort(int[] array){

for(int i = 0;i<array.length;i++){

int temp = array[i];//待插入到前面有序序列的值

int left = 0;//有序序列的左侧

int right = i-1;//有序序列的右侧

int middle = 0;//有序序列的中间

while(left <= right){

middle = (left + right)/2;//赋值

if(temp<array[middle]){

right = middle-1;

}else{

left = middle + 1;

}

}

for(int j = i-1;j>=left;j--){

//从i-1到left依次向后移动一位,等待temp值插入

array[j+1] = array[j];

}

if(left != i ){

array[left] = temp;

}

}

return array;

}

## 快速排序

时间复杂度平均为O(nlogn)，最好为O(nlogn)，最差为O(logn2)

快速排序使用分治法策略来把一个序列分为两个子序列，基本步骤为：

1. 先从序列中取出一个数作为基准数；
2. 分区过程：将把这个数大的数全部放到它的右边，小于或者等于它的数全放到它的左边；
3. 递归地对左右子序列进行不走2，直到各区间只有一个数。

**代码：**

public static void quickSort(int[] array, int left, int right){

if(left < right){//如果只有1个元素就不执行，2个及以上元素才会执行

int pivot = array[left]; //设置为基准值

int low = left; //左指针

int high = right; //右指针

while(low < high){

while(low < high && array[high] >= pivot) //右指针指向的值大于等于基准值

high--; //右指针左移，直到指向的值小于基准值

array[low] = array[high]; //将左指针指向的值和该值交换

while(low < high && array[low] <= pivot) //左指针指向的值小于等于基准值

low++; //左指针右移，直到指向的值大于基准值

array[high] = array[low]; //将该值与右指针指向的值交换

}

array[low] = pivot; //将基准值填入正确的位置，这样得到基准值左边的数据都小于它，右边同理

System.out.println("pivot = " + pivot + "," + Arrays.toString(array));

quickSort(array, left, low - 1);

quickSort(array, low + 1, right);

}

}

## 归并排序

时间复杂度

　对长度为n的文件，需进行趟二路归并，每趟归并的时间为O(n)，故其时间复杂度无论是在最好情况下还是在最坏情况下均是O(nlgn)。

一、两路归并排序算法思路

分而治之(divide - conquer);每个递归过程涉及三个步骤  
第一, 分解: 把待排序的 n 个元素的序列分解成两个子序列, 每个子序列包括 n/2 个元素.  
第二, 治理: 对每个子序列分别调用归并排序MergeSort, 进行递归操作  
第三, 合并: 合并两个排好序的子序列,生成排序结果.

public class MergeSort\_归并排序 {

public static int[] sort(int[] a,int low,int high){

int mid = (low+high)/2;

if(low<high){

sort(a,low,mid);

sort(a,mid+1,high);

//左右归并

merge(a,low,mid,high);

}

return a;

}

public static void merge(int[] a, int low, int mid, int high) {

int[] temp = new int[high-low+1];

int i= low;

int j = mid+1;

int k=0;

// 把较小的数先移到新数组中

while(i<=mid && j<=high){

if(a[i]<a[j]){

temp[k++] = a[i++];

}else{

temp[k++] = a[j++];

}

}

// 把左边剩余的数移入数组

while(i<=mid){

temp[k++] = a[i++];

}

// 把右边边剩余的数移入数组

while(j<=high){

temp[k++] = a[j++];

}

// 把新数组中的数覆盖nums数组

for(int x=0;x<temp.length;x++){

a[x+low] = temp[x];

}

}

public static void main(String[] args){

int arr[]={2,1,4,3,2,9,5,6,8,7};

int[] newArr=sort(arr,0,arr.length-1);

for(int temp:newArr){

System.out.print(temp+",");

}

}

}

## 快速排序与归并排序比较

* 1. 在数组长度小于一千万的时候，如下图，快速排序的速度要略微快于归并排序，可能是因为归并需要额外的数组开销（比如声明临时local数组用来储存排序结果），这些操作让归并算法在小规模数据的并不占优势。
  2. （2）但是，当数据量达到亿级时，归并的速度开始超过快速排序了，如下图，因为归并排序比快排要稳定，所以在数据量大的时候，快排容易达到O(n^2)的时间复杂度，当然这里是指未改进的快排算法。

## 堆排序

## 希尔排序

## 冒泡排序

## 堆和栈

## 队列和栈互相转化

@author superYC

\* 两个队列实现一个栈

\*

\*/

public class TwoQueueImplStack {

Queue<Integer> queue1 = new ArrayDeque<Integer>();

Queue<Integer> queue2 = new ArrayDeque<Integer>();

/\*

\* 向栈中压入数据

\*/

public void push(Integer element){

//两个队列都为空时，优先考虑 queue1

if(queue1.isEmpty() && queue2.isEmpty()){

queue1.add(element);

return;

}

//如果queue1为空，queue2有数据，直接放入queue2

if(queue1.isEmpty()){

queue2.add(element);

return;

}

//如果queue2为空，queue1有数据，直接放入queue1中

if(queue2.isEmpty()){

queue1.add(element);

return;

}

}

/\*

\* 从栈中弹出一个数据

\*/

public Integer pop(){

//如果两个栈都为空，则没有元素可以弹出，异常

if(queue1.isEmpty() && queue2.isEmpty()){

try{

throw new Exception("satck is empty!");

}catch(Exception e){

e.printStackTrace();

}

}

//如果queue1中没有元素，queue2中有元素，将其queue2中的元素依次放入queue1中，直到最后一个元素，弹出即可

if(queue1.isEmpty()){

while(queue2.size() > 1){

queue1.add(queue2.poll());

}

return queue2.poll();

}

//如果queue2中没有元素，queue1中有元素，将其queue1中的元素依次放入queue2中，直到最后一个元素，弹出即可

if(queue2.isEmpty()){

while(queue1.size() > 1){

queue2.add(queue1.poll());

}

return queue1.poll();

}

return (Integer)null;

}

public static void main(String[] args) {

TwoQueueImplStack qs = new TwoQueueImplStack();

qs.push(2);

qs.push(4);

qs.push(7);

qs.push(5);

System.out.println(qs.pop());

System.out.println(qs.pop());

qs.push(1);

System.out.println(qs.pop());

}

}

@author superYC

\* 两个栈实现一个队列

\*

\*/

public class TwoStackImplQueue {

Stack<Integer> stack1 = new Stack<Integer>();

Stack<Integer> stack2 = new Stack<Integer>();

/\*

\* 队列的数据压入过程

\*/

public void push(Integer element){

stack1.push(element);

}

/\*

\* 队列的数据弹出过程

\*/

public Integer pop(){

if(stack2.size() <= 0){ //第二个栈为空

while(stack1.size() > 0){ //第一个栈不为空

stack2.push(stack1.pop()); //将其第一个栈的数据压入第二个栈中

}

}

if(stack2.isEmpty()){

try{

throw new Exception("queue is empty");

}catch(Exception e){

//e.printStackTrace();

}

}

Integer head = stack2.pop();

return head;

}

public static void main(String[] args) {

TwoStackImplQueue sq = new TwoStackImplQueue();

sq.push(1);

sq.push(3);

sq.push(5);

sq.push(4);

sq.push(2);

System.out.println(sq.pop());

System.out.println(sq.pop());

sq.push(7);

System.out.println(sq.pop());

}

}

## 二叉树

## B树

## 括号匹配问题

假设表达式中只允许两种括号：()、{}；  
正确表达顺序为：()或{}或({})或{({}{})}的形势；如{(}或(})或({)}的表达形势均不对。  
算法的设计思想：

　　出现左括弧则进栈；

　　出现右括弧则首先检测栈是否为空，

　　　　若栈空则表明此右括弧多余，表达式不匹配。

　　　　否则和栈顶数据比较，若匹配则栈顶出栈。

　　　　否则表明表达式不匹配；

　　最后若栈空，则表明匹配成功；否则表明不匹配。

package linklist;

import java.util.Stack;

/\*\*

\* 此题还可以引申至配对字符符匹配问题，如单引号，双引号匹配问题。

\*

\*/

public class ExpStackMatching {

public boolean matching(String expression)

{

if(expression==null||expression=="")

{

System.out.println( "输入表达式为空或没有输入表达式" ) ;

}

Stack<Character> stack = new Stack<Character>() ;

for(int index=0 ; index<expression.length();index++)

{

switch(expression.charAt(index))

{

case '(':

stack.push(expression.charAt(index)) ;

break ;

case '{':

stack.push(expression.charAt(index)) ;

break ;

case ')':

if(!stack.isEmpty()&&stack.peek()=='(')

{

stack.pop() ;

}

break ;

case '}':

if(!stack.isEmpty()&&stack.peek()=='{')

{

stack.pop();

}

}

}

if(stack.isEmpty())

return true ;

return false ;

}

public static void main(String[] args) {

String expression = "{((1+3)+2+4)+9\*7}" ;

ExpStackMatching oj = new ExpStackMatching() ;

boolean flag = oj.matching(expression) ;

if(flag)

{

System.out.println("匹配成功！") ;

}

else

{

System.out.println(" 匹配失败 ");

}

}

}

## 循环报数取剩余

问题描述：

十个猴子围成一圈选大王，依次1-3 循环报数，报到3 的猴子被淘汰，直到最后一只猴子成为大王。问，哪只猴子最后能成为大王？

方法一：Java链表

public class TestAll {

static Scanner scanner = new Scanner(System.in);  
static int num;  
static String str;  
static LinkedList<String> list = new LinkedList<String>();  
static LinkedList<String> result = new LinkedList<String>();

public static void main(String[] arg) {  
input();  
output();  
}

private static void output() {  
pushNum();  
Iterator it = result.iterator();  
while (it.hasNext()) {  
System.out.print(it.next() + " ");  
}  
}

private static void pushNum() {  
int i = 1;  
while (list.size() > 0) {  
// System.out.println(i+"!! ");  
Iterator it = list.iterator();  
while (it.hasNext()) {  
String node = (String) it.next();  
if (i == num) {  
result.add(node);  
it.remove();  
i = 0;  
}  
i++;  
}  
}

}

private static void input() {  
str = scanner.nextLine();  
String[] tmp = str.split(" ");  
num = Integer.parseInt(tmp[0]);  
for (int i = 1; i < tmp.length; i++) {  
list.add(tmp[i]);  
}

}  
}

方法二：数组

public class TimeTest {  
public static void main(String[] args) {  
int num = 10;  
boolean[] array = new boolean[num];  
for (int i = 0; i < num; i++) {  
array[i] = true;  
}  
int index = 0;  
int count = 0;  
int n = num;  
while (n > 1) {  
if (array[index] == true) {  
count++;  
if (count == 3)  
// 当count等于3时，就淘汰一个；  
{  
array[index] = false;  
n--; // 当有一个被淘汰时，n--；  
count = 0;  
}  
}  
index++;  
// 当从0循环到29时，重新置index为0；  
if (index == num-1) {  
index = 0;  
}  
}  
for (int i = 0; i < num; i++) {  
if (array[i] == true)  
System.out.println(i + 1);  
}  
}  
}

 其中方法一的时间复杂度为O（n^2）

方法二的时间复杂度为O(n)

## 大数据量排序问题

## 求数组交集

先排序，再利用两个指针

## 计算表达式

## 一个数组里面有正数、负数 量1000万，找出子集之和最大的序列

public static void maxSum(int[] nums) {  
 int start = 0;  
 int end = 0;  
 int max = Integer.*MIN\_VALUE*;  
  
 int sum = 0;  
 int ts = 0;  
 for(int i = 0; i < nums.length; i++) {  
 sum += nums[i];  
 if(sum > max) {  
 start = ts;  
 end = i;  
 max = sum;  
  
 }  
 if(sum < 0) {  
 ts = i + 1;  
 sum = 0;  
 }  
 }  
  
 System.*out*.println("maxSum = " + max + ", start : " + start + ", end = " + end);  
}

## 一亿个数据查出最大的前10个

## 字符串数组找出里面出现次数最多的字符

public class 字符串中出现最多的字符 {  
 public static void main(String[] args) throws Exception {  
 String Str = "AAbbcccaaaa";  
 char[] StrArr = Str.toCharArray();// 把字符串转为字符数组toCharArray  
  
 Map<Character, Integer> map = *MapFunction*(StrArr);  
 char ch = *FindMapMaxValue*(map);  
 }  
  
 */\*\*  
 \* MapFunction:实现将字符数组转存到Map中， 其中，Map中的key为出现的字符，value对应该字符出现的次数  
 \** ***@param*** *StrArr StrArr字符数组，输入前必须先将字符串转为字符数组  
 \** ***@return*** *map 集合中，key为出现的字符（Character），value对应该字符出现的次数（Integer）  
 \*/* public static Map<Character, Integer> MapFunction(char[] StrArr) {  
 Map<Character, Integer> map = new HashMap<Character, Integer>();  
 if (!(StrArr == null || StrArr.length == 0))// 先判断字符数组是否为空  
 for (int i = 0; i < StrArr.length; i++)  
 if (null != map.get(StrArr[i]))  
 // 若不为空，说明已经存在相同字符，则Value值在原来的基础上加1  
 map.put(StrArr[i], map.get(StrArr[i]) + 1);  
 else  
 map.put(StrArr[i], 1);  
  
 return map;  
 }  
  
 */\*\*  
 \* FindMapMaxValue 差找map中Value的最大值maxValue，类似于选择排序寻找最大值的过程：  
 \* 先任取一个Value值定义为最大值，然后与之比较  
 \** ***@param*** *map 输入Map集合，该集合key为出现的字符（Character），value对应该字符出现的次数（Integer）  
 \** ***@return*** *maxKey 返回出现次数最多的字符  
 \*/* public static Character FindMapMaxValue(Map<Character, Integer> map) {  
 Set<Character> keys = map.keySet();// 获得所有key值  
 Iterator keys\_Itera = keys.iterator();// 实例化Iterator  
 // keys\_Itera.next():依次获得key值  
 // map.get(key):获得对应的value值  
 Character maxKey = (Character) keys\_Itera.next();// 定义第一个为最大value和对应的key  
 int maxValue = map.get(maxKey);  
  
 while (keys\_Itera.hasNext()) {  
 Character temp = (Character) keys\_Itera.next();  
 if (maxValue < map.get(temp)) {  
 maxKey = temp;  
 maxValue = map.get(temp);  
 }  
 }  
 System.*out*.println("出现次数最多的字符：" + maxKey + " 出现次数：" + maxValue);  
 return maxKey;  
 }  
}

# 设计模式

## 单例模式

**为什么单例**

1、在内存中只有一个对象，节省内存空间。避免频繁的创建销毁对象，可以提高性能。避免对共享资源的多重占用。可以全局访问。

2、确保一个类只有一个实例，自行实例化并向系统提供这个实例

**单例需要注意的问题**

1、线程安全问题

2、资源使用问题

实际上本文就是在讨论这两个问题

### 1、方式1 （饿汉式）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package com;    public class Singleton {        private static Singleton instance = new Singleton() ;          private Singleton(){        }        public static Singleton getInstance() {          return  instance ;      }    } |

　　优点：在未调用getInstance() 之前，实例就已经创建了，天生线程安全

      缺点：如果一直没有调用getInstance() , 但是已经创建了实例，造成了资源浪费。

### 2、方式1 （懒汉式）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com;    public class Person {      private static Person person ;        private Person(){        }        public static Person get(){          if ( person == null ) {              person = new Person() ;          }          return person ;      }    } |

优点：get() 方法被调用的时候，才创建实例，节省资源。

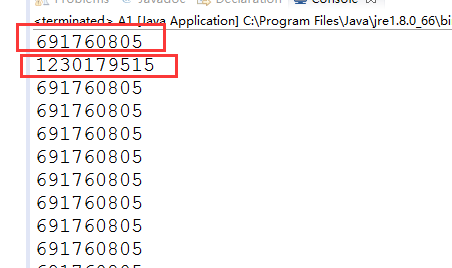
缺点：线程不安全。

　　这种模式，可以做到单例模式，但是只是在单线程中是单例的，如果在多线程中操作，可能出现多个实例。

     测试：启动20个线程，然后在线程中打印 Person 实例的内存地址

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com;    public class A1  {        public static void main(String[] args) {            for ( int i = 0 ;  i < 20 ; i ++ ) {              new Thread( new Runnable() {                    @Override                  public void run() {                      System.out.println( Person.get().hashCode() );                  }              }).start(); ;            }        }  } |

　　结果：可以看到出现了两个实例



    造成的原因：

线程A希望使用Person，调用get()方法。因为是第一次调用，A就发现 person 是null的，于是它开始创建实例，就在这个时候，CPU发生时间片切换，线程B开始执行，它要使用 Person ，调用get()方法，同样检测到 person 是null——注意，这是在A检测完之后切换的，也就是说A并没有来得及创建对象——因此B开始创建。B创建完成后，切换到A继续执行，因为它已经检测完了，所以A不会再检测一遍，它会直接创建对象。这样，线程A和B各自拥有一个 Person 的对象——单例失败！

总结：1、可以实现单线程单例

        2、多线单例无法保证

改进：1、加锁

### 3、 用synchronized  加锁同步

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com;    public class Person {      private static Person person ;        private Person(){        }        public synchronized static Person get(){          if ( person == null ) {              person = new Person() ;          }          return person ;      }        } |

　　经过测试，已经可以满足多线程的安全问题了，synchronized修饰的同步块可是要比一般的代码段慢上几倍的！如果存在很多次get()的调用，那性能问题就不得不考虑了！

优点：

1、满足单线程的单例

2、满足多线程的单例

缺点：

1、性能差

### 4、改进性能  双重校验

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package com;    public class Person {      private static Person person ;        private Person(){      }        public static Person get(){          if ( person == null ) {              synchronized ( Person.class ){                  if (person == null) {                      person = new Person();                  }              }          }          return person ;      }    } |

　　首先判断person 是不是为null，如果为null，加锁初始化；如果不为null，直接返回 person 。整个设计，进行了双重校验。

优点：

1、满足单线程单例

2、满足多线程单例

3、性能问题得以优化

缺点：第一次加载时反应不快，由于java内存模型一些原因偶尔失败

### 5、volatile 关键字，解决双重校验带来的弊端

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package com;    public class Person {        private static volatile Person person = null ;          private Person(){        }        public static Person getInstance(){          if ( person == null ) {              synchronized ( Person.class ){                  if ( person == null ) {                      person = new Person() ;                  }              }          }            return person ;      }    } |

　　假设没有关键字volatile的情况下，两个线程A、B，都是第一次调用该单例方法，线程A先执行 person = new Person()，该构造方法是一个非原子操作，编译后生成多条字节码指令，由于JAVA的指令重排序，可能会先执行 person 的赋值操作，该操作实际只是在内存中开辟一片存储对象的区域后直接返回内存的引用，之后 person 便不为空了，但是实际的初始化操作却还没有执行，如果就在此时线程B进入，就会看到一个不为空的但是不完整 （没有完成初始化）的 Person对象，所以需要加入volatile关键字，禁止指令重排序优化，从而安全的实现单例。

     补充：看了图片加载框架 Glide (3.7.0版) 源码，发现glide 也是使用volatile 关键字的双重校验实现的单例，可见这种方法是值得信赖的。

### 6、静态内部类

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package com;    public class Person {        private Person(){        }        private static class PersonHolder{          /\*\*           \* 静态初始化器，由JVM来保证线程安全           \*/          private static Person instance = new Person();      }          public static Person getInstance() {          return PersonHolder.instance;      }    } |

　　优点：

1、资源利用率高，不执行getInstance()不被实例，可以执行该类其他静态方法

### 7、枚举类实现单例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package com;    public enum Singleton {      INSTANCE ;        public void show(){          // Do you need to do things      }  } |

　　使用

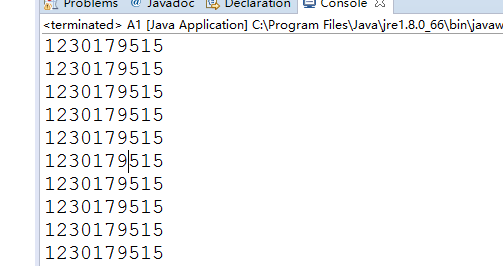
**获取实例对象：Singleton.INSTANCE**

**调用其他方法：Singleton.INSTANCE.show();**

**测试**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package com;    public class A1  {        public static void main(String[] args) {            for ( int i = 0 ;  i < 20 ; i ++ ) {              new Thread( new Runnable() {                    @Override                  public void run() {                      System.out.println( Singleton.INSTANCE.hashCode() ) ;                  }              }).start(); ;            }      }  } |

      结果：



   可以看到用20个线程去访问对象的内存地址， 内存地址都是一样的，保证了线程的安全性。

**总结：**

**1、上面的7中方法，都实现了某种程度的单例，各有利弊，根据使用的场景不同，需要满足的特性不同，选取合适的单例方法才是正道。**

**2、对线程要求严格，对资源要求不严格的推荐使用：1    饿汉式**

**3、对线程要求不严格，对资源要求严格的推荐使用：2    懒汉式**

**4、对线程要求稍微严格，对资源要求严格的推荐使用：4  双重加锁**

**5、同时对线程、资源要求非常严格的推荐使用：5 、 6**

## 工厂模式

工厂方法模式：工厂方法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

## 代理模式

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

## 观察者模式

### 观察者模式的定义：

　　在对象之间定义了一对多的依赖，这样一来，当一个对象改变状态，依赖它的对象会收到通知并自动更新。

### 大白话：

其实就是发布订阅模式，发布者发布信息，订阅者获取信息，订阅了就能收到信息，没订阅就收不到信息。

### 可以看到，该模式包含四个角色

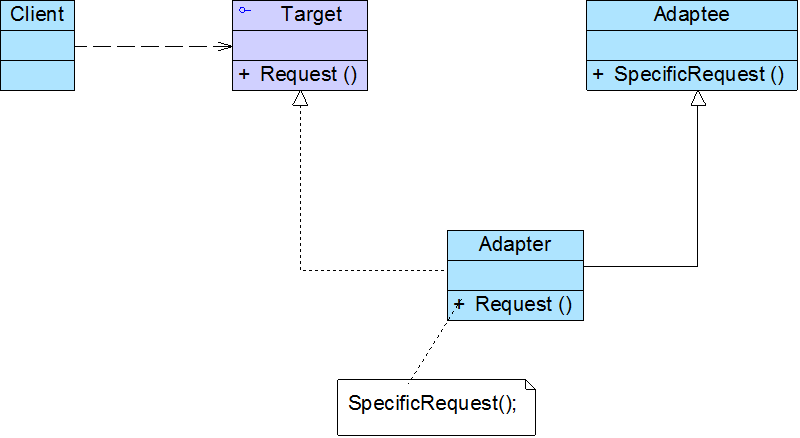
* **抽象被观察者角色**：也就是一个抽象主题，它把所有对观察者对象的引用保存在一个集合中，每个主题都可以有任意数量的观察者。抽象主题提供一个接口，可以增加和删除观察者角色。一般用一个抽象类和接口来实现。
* **抽象观察者角色**：为所有的具体观察者定义一个接口，在得到主题通知时更新自己。
* **具体被观察者角色**：也就是一个具体的主题，在集体主题的内部状态改变时，所有登记过的观察者发出通知。
* **具体观察者角色**：实现抽象观察者角色所需要的更新接口，一边使本身的状态与制图的状态相协调。

### 使用场景例子

　　有一个微信公众号服务，不定时发布一些消息，关注公众号就可以收到推送消息，取消关注就收不到推送消息。

## 适配器模式

适配器模式：将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作



由图可知适配器模式包含一下三个角色：

　　1：Target(目标抽象类)：目标抽象类定义客户所需的接口，可以是一个抽象类或接口，也可以是具体类。在类适配器中，由于C#语言不支持多重继承，所以它只能是接口。

　　2：Adapter(适配器类)：它可以调用另一个接口，作为一个转换器，对Adaptee和Target进行适配。它是适配器模式的核心。

　　3：Adaptee(适配者类)：适配者即被适配的角色，它定义了一个已经存在的接口，这个接口需要适配，适配者类包好了客户希望的业务方法。

## 桥接模式

将抽象部分与它的实现部分分离，使它们都可以独立地变化

## 策略模式

一、策略模式：  
 (1)解决场景：  
某个功能有多个方案可以实现，要达到某个目的，需要根据具体的实际情况，选择合适的方法，  
 (2)如何实现：  
分为两个层次  
　(a) 环境类   
　环境类负责接收用户的请求，并根据实际情况把相应的请求委托给一组策略类中的一个；

　(b) 一组策略类  
　　一组封装了具体的实现类的算法，并负责具体的计算过程

# Spring

## IOC原理、如何依赖注入

## 互相依赖情况下spring能否启动成功

形成一个死循环

## @autowire和@resource的区别

## AOP原理

## AOP执行流程

## SpringMVC请求过程

## Spring事务控制

## Spring bean的生命周期

## Spring和SpringBoot的区别

## 如何实现注解

# Hibernate、Mybatis

## 两个ORM的区别

JDBC：

   我们平时使用jdbc进行编程，大致需要下面几个步骤：

   1，使用jdbc编程需要连接[数据库](http://lib.csdn.net/base/mysql)，注册驱动和数据库信息

   2，操作Connection，打开Statement对象

   3，通过Statement对象执行SQL，返回结果到ResultSet对象

   4，使用ResultSet读取数据，然后通过代码转化为具体的POJO对象

   5，关闭数据库相关的资源

  jdbc的缺点：

  一：工作量比较大，需要连接，然后处理jdbc底层事务，处理数据类型，还需要操作Connection，Statement对象和ResultSet对象去拿数据并关闭他们。

   二：我们对jdbc编程可能产生的异常进行捕捉处理并正确关闭资源

Hibernate：

    Hibernate是建立在若干POJO通过xml映射文件（或注解）提供的规则映射到数据库表上的。我们可以通过POJO直接操作数据库的数据，他提供的是一种全表映射的模型。相对而言，Hibernate对JDBC的封装程度还是比较高的，我们已经不需要写SQL，只要使用HQL语言就可以了。

    使用Hibernate进行编程有以下好处：

    1，消除了代码的映射规则，它全部分离到了xml或者注解里面去配置。

    2，无需在管理数据库连接，它也配置到xml里面了。

    3，一个会话中不需要操作多个对象，只需要操作Session对象。

    4，关闭资源只需要关闭一个Session便可。

    这就是Hibernate的优势，在配置了映射文件和数据库连接文件后，Hibernate就可以通过Session操作，非常容易，消除了jdbc带来的大量代码，大大提高了编程的简易性和可读性。Hibernate还提供了级联，缓存，映射，一对多等功能。Hibernate是全表映射，通过HQL去操作pojo进而操作数据库的数据。

    Hibernate的缺点：

    1，全表映射带来的不便，比如更新时需要发送所有的字段。

    2，无法根据不同的条件组装不同的SQL。

    3，对多表关联和复杂的sql查询支持较差，需要自己写sql，返回后，需要自己将数据封装为pojo。

    4，不能有效的支持存储过程。

    5，虽然有HQL，但是性能较差，大型互联网系统往往需要优化sql，而hibernate做不到。

Mybatis可以配置动态sql，解决了hibernate表名根据时间变化，不同条件下列不一样的问题，同时你也可以对sql进行优化，通过配置决定你的sql映射规则，也能支持存储过程，所以对于一些复杂和需要优化性能的sql查询它就更加方便。

Hibernate具有自己的日志统计。Mybatis本身不带日志统计，使用Log4j进行日志记录。

Mybatis优势

    MyBatis可以进行更为细致的SQL优化，可以减少查询字段。

    MyBatis容易掌握，而Hibernate门槛较高。

Hibernate优势

    Hibernate的DAO层开发比MyBatis简单，Mybatis需要维护SQL和结果映射。

    Hibernate对对象的维护和缓存要比MyBatis好，对增删改查的对象的维护要方便。

    Hibernate数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。

    Hibernate有更好的二级缓存机制，可以使用第三方缓存。MyBatis本身提供的缓存机制不佳。

## Mybatis批量操作

### MyBatis中批量插入

方法一：

<insert id="insertbatch" parameterType="java.util.List">

  <selectKey keyProperty="fetchTime" order="BEFORE"

  resultType="java.lang.String">

  SELECT CURRENT\_TIMESTAMP()

  </selectKey>

  insert into kangaiduoyaodian ( depart1, depart2, product\_name,

  generic\_name, img, product\_specification, unit,

  approval\_certificate, manufacturer, marketPrice, vipPrice,

  website, fetch\_time, productdesc ) values

  <foreach collection="list" item="item" index="index"

  separator=",">

  ( #{item.depart1}, #{item.depart2}, #{item.productName},

  #{item.genericName}, #{item.img},

  #{item.productSpecification}, #{item.unit},

  #{item.approvalCertificate}, #{item.manufacturer},

  #{item.marketprice}, #{item.vipprice}, #{item.website},

  #{fetchTime}, #{item.productdesc} )

  </foreach>

  </insert>

方法二：

<insert id="batchInsertB2B" parameterType="ArrayList">  
insert into xxxxtable(hkgs,hkgsjsda,office,asdf,ddd,ffff,supfullName,classtype,agent\_type,remark)  
<foreach collection="list" item="item" index="index" separator="union all">  
select #{item.hkgs,jdbcType=VARCHAR},  
#{item.hkgsjsda,jdbcType=VARCHAR},  
#{item.office,jdbcType=VARCHAR},  
#{item.asdf,jdbcType=VARCHAR},  
#{item.ddd,jdbcType=VARCHAR},  
#{item.ffff,jdbcType=VARCHAR},  
#{item.supfullName,jdbcType=VARCHAR},0,0,  
#{item.remark,jdbcType=VARCHAR} from dual  
</foreach>  
</insert>

可以考虑用union all来实现批量插入。  
例如：  
insert into XX\_TABLE(XX,XX,XX)select 'xx','xx','xx' union all select 'xx','xx','xx' union all select 'xx','xx','xx' ...  
先拼装好语句再动态传入insert into XX\_TABLE(XX,XX,XX)后面部分

### MyBatis中批量删除

<!-- 通过主键集合批量删除记录 -->

<delete id="batchRemoveUserByPks" parameterType="java.util.List">

DELETE FROM LD\_USER WHERE ID in

<foreach item="item" index="index" collection="list" open="(" separator="," close=")">

#{item}

</foreach>

</delete>

### MyBatis中in子句

[mybatis in 参数 使用方法](http://blog.csdn.net/cumtcsguo054/article/details/7395676)

1.只有一个参数

参数的类型要声明为List或Array

Sql配置如下：

<select id="selectProduct" resultMap="Map">

SELECT \*

FROM PRODUCT

WHERE PRODUCTNO IN

     <foreach item="productNo" index="index" collection="参数的类型List或array">

            #{productNo}

    </foreach>

</select>

2.多个参数

首先要将多个参数写入同一个map，将map作为一个参数传入mapper

Sql配置如下：

<select id="selectProduct" resultMap="Map">

SELECT \*

FROM PRODUCT

WHERE PRODUCTNO IN

     <foreach item="productNo" index="index" collection="map中集合参数的名称">

            #{productNo}

    </foreach>

</select>

 MyBatis批量修改

 <update id="updateOrders" parameterType="java.util.List">  
 update orders set state = '0' where no in  
 <foreach collection="list" item="nos" open="(" separator="," close=")">  
   #{nos}  
 </foreach>  
 </update>

## Hibernate缓存机制

## Mybatis防止SQL注入

**mybatis中的#和$的区别：**

1、#将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。  
如：where username=#{username}，如果传入的值是111,那么解析成sql时的值为where username="111", 如果传入的值是id，则解析成的sql为where username="id".　  
2、$将传入的数据直接显示生成在sql中。  
如：where username=${username}，如果传入的值是111,那么解析成sql时的值为where username=111；  
如果传入的值是;drop table user;，则解析成的sql为：select id, username, password, role from user where username=;drop table user;  
3、#方式能够很大程度防止sql注入，$方式无法防止Sql注入。  
4、$方式一般用于传入数据库对象，例如传入表名.  
5、一般能用#的就别用$，若不得不使用“${xxx}”这样的参数，要手工地做好过滤工作，来防止sql注入攻击。  
6、在MyBatis中，“${xxx}”这样格式的参数会直接参与SQL编译，从而不能避免注入攻击。但涉及到动态表名和列名时，只能使用“${xxx}”这样的参数格式。所以，这样的参数需要我们在代码中手工进行处理来防止注入

**SQL注入**，大家都不陌生，是一种常见的攻击方式。**攻击者**在界面的表单信息或URL上输入一些奇怪的SQL片段（例如“or ‘1’=’1’”这样的语句），有可能入侵**参数检验不足**的应用程序。

传入参数后，打印出执行的SQL语句，会看到SQL是这样的：

select id, username, password, role from user where username=? and password=?

　　不管输入什么参数，打印出的SQL都是这样的。这是因为MyBatis启用了预编译功能，在SQL执行前，会先将上面的SQL发送给数据库进行编译；执行时，直接使用编译好的SQL，替换占位符“?”就可以了。因为SQL注入只能对编译过程起作用，所以这样的方式就很好地避免了SQL注入的问题。

## Mybatis对JDBC进行的封装

# DRUID

# 数据库

## 悲观锁和乐观锁、如何实现

<https://www.cnblogs.com/qjjazry/p/6581568.html>

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语synchronized关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，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**，**可以使用版本号等机制**。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

悲观锁机制存在以下问题：

　　　　　　1. 在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。

　　　　　　2. 一个线程持有锁会导致其它所有需要此锁的线程挂起。

　　　　　　3. 如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。

　　　　对比于悲观锁的这些问题，另一个更加有效的锁就是乐观锁。其实乐观锁就是：每次不加锁而是假设没有并发冲突而去完成某项操作，如果因为并发冲突失败就重试，直到成功为止。

　　乐观锁：

　　　　乐观锁（ Optimistic Locking ）在上文已经说过了，其实就是一种思想。相对悲观锁而言，乐观锁假设认为数据一般情况下不会产生并发冲突，所以在数据进行提交更新的时候，才会正式对数据是否产生并发冲突进行检测，如果发现并发冲突了，则让返回用户错误的信息，让用户决定如何去做。

　　　　上面提到的乐观锁的概念中其实已经阐述了它的具体实现细节：主要就是两个步骤：**冲突检测和数据更新。**其实现方式有一种比较典型的就是 Compare and Swap ( CAS )。

　　CAS：

　　　　CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

　　　　CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置V的值与预期原值A相匹配，那么处理器会自动将该位置值更新为新值B。否则处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“ 我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。 ”这其实和乐观锁的冲突检查+数据更新的原理是一样的。

## 事务特性

一．什么是事务   
事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤消。也就是事务具有原子性，一个事务中的一系列的操作要么全部成功，要么一个都不做。   
事务的结束有两种，当事务中的所以步骤全部成功执行时，事务提交。如果其中一个步骤失败，将发生回滚操作，撤消撤消之前到事务开始时的所有操作

数据库事务 transanction 正确执行的四个基本要素。ACID,原子性(Atomicity)、一致性(Correspondence)、隔离

性(Isolation)、持久性(Durability)。

      （1）原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

      （2）一致性：在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。

      （3）隔离性：一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

       （4）持久性：在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

## 事务隔离级别

### 不考虑事务的隔离性，会产生的几种问题：

01：脏读

是指一个事务处理过程里读取了另一个未提交的事务中的数据，然后使用了这个数据；

02：不可重复读

不可重复读是指在一个事务内，多次读取同一个数据，在这个事务还没有结束 ，另一个事务也访问该同一数据，但是由于第二个事务的修改，那么第一个事务两次读取的数据可能不一样，因此称为不可重复读；即同一个事务中原始数据读取不可重复。 注：不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

3，虚读(幻读)

　　幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

现在来看看MySQL数据库为我们提供的四种隔离级别：

　　① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

　　② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

　　③ Read committed (读已提交)：可避免脏读的发生。

　　④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

## 事务传播机制

## MySQL事务隔离级别

## 一个慢sql能怎么优化

## SQL优化

## SQL执行计划

## myCat分库分表、以及如何查询

## 分库分表策略

## 索引

## 聚合索引、单列索引

## 聚集索引和非聚集索引

## 走不走索引

## B+树的结构

## JDBC连接数据库

5、JDBC编程步骤

（1）加载驱动程序：Class.forName(driverClass)

加载mysql驱动：Class.forName("com.mysql.jdbc.Driver");

加载oracle驱动：Class.forName("oracle.jdbc.driver.OracleDriver");

（2）获得数据库连接

DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/imooc",user,password);

DriverManager.gerConnection(URL,user,password);

（3）创建Statement对象：conn.createStatement();

（4）向数据库发送SQL命令

（5）处理数据库的返回结果(ResultSet类)

public static void main(String[] args) throws ClassNotFoundException, SQLException {

13 String URL="jdbc:mysql://127.0.0.1:3306/imooc?useUnicode=true&amp;characterEncoding=utf-8";

14 String USER="root";

15 String PASSWORD="tiger";

16 //1.加载驱动程序

17 Class.forName("com.mysql.jdbc.Driver");

18 //2.获得数据库链接

19 Connection conn=DriverManager.getConnection(URL, USER, PASSWORD);

20 //3.通过数据库的连接操作数据库，实现增删改查（使用Statement类）

21 Statement st=conn.createStatement();

22 ResultSet rs=st.executeQuery("select \* from user");

23 //4.处理数据库的返回结果(使用ResultSet类)

24 while(rs.next()){

25 System.out.println(rs.getString("user\_name")+" "

26 +rs.getString("user\_password"));

27 }

28

29 //关闭资源

30 rs.close();

31 st.close();

32 conn.close();

33 }

## 设计数据库遵循的规范

## 防止SQL注入

# MySql、Oracle

## 两者区别

## 默认隔离级别

## mysql优化

## MySQL事务传播级别

## MySQL分页

Mysql的分页查询十分简单，但是当数据量大的时候一般的分页就吃不消了。

传统分页查询：SELECT c1,c2,cn… FROM table LIMIT n,m

MySQL的limit工作原理就是先读取前面n条记录，然后抛弃前n条，读后面m条想要的，所以n越大，偏移量越大，性能就越差。

推荐分页查询方法：

**1、尽量给出查询的大致范围**

1. SELECT c1,c2,cn... FROM table WHERE id>=20000 LIMIT 10;

**2、子查询法**

1. SELECT c1,c2,cn... FROM table WHERE id>=
2. (
3. SELECT id FROM table LIMIT 20000,1
4. )
5. LIMIT 10;

**3、高性能MySQL一书中提到的只读索引方法**

优化前SQL:

1. SELECT c1,c2,cn... FROM member ORDER BY last\_active LIMIT 50,5

优化后SQL:

1. SELECT c1, c2, cn .. .
2. FROM member
3. INNER JOIN (SELECT member\_id FROM member ORDER BY last\_active LIMIT 50, 5)
4. USING (member\_id)

分别在于，优化前的SQL需要更多I/O浪费，因为先读索引，再读数据，然后抛弃无需的行。而优化后的SQL(子查询那条)只读索引(Cover index)就可以了，然后通过member\_id读取需要的列。

**4、第一步用用程序读取出ID，然后再用IN方法读取所需记录**

程序读ID：

1. SELECT id FROM table LIMIT 20000, 10;
2. SELECT c1, c2, cn .. . FROM table WHERE id IN (id1, id2, idn.. .)

## 分页比较

**Mysql分页**

select \* from A where 1=1 limit 10,10

**Mssql分页**

select top 10 \* from A where A.id not in(sel ect top 20 id from A order by create\_time desc) order by create\_time desc

**Oracle分页**

select \* from( select rownum rn,\* from A where rn<21)where rn >10

# Elasticsearch

# ELK

# canal

# cat

# apollo

# nginx

# git\maven

# Sharding-jdbc

# Redis

<https://www.cnblogs.com/jasontec/p/9699242.html>

## 数据结构、以及特性、为何快速

### Redis为什么 那么快

Redis为什么这么快

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；

2、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用多路I/O复用模型，非阻塞IO；

5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

## 运用场景

Redis使用场景

Session 共享

1、缓存

    缓存机制几乎在所有的大型网站都有使用，合理地使用缓存不仅可以加快数据的访问速度，而且能够有效地降低后端数据源的压力。Redis提供了键值过期时间设置，并且也提供了灵活控制最大内存和内存溢出后的淘汰策略。

2、排行榜系统

    排行榜系统几乎存在于所有的网站，例如按照热度排名的排行榜，按照发布时间的排行榜，按照各种复杂维度计算出的排行榜，Redis提供了列表和有序集合数据结构，合理地使用这些数据结构可以很方便地构建各种排行榜系统。

3、计数器应用

    计数器在网站中的作用至关重要，例如视频网站有播放数、电商网站有浏览数，为了保证数据的实时性，每一次播放和浏览都要做加1的操作，如果并发量很大对于传统关系型数据的性能是一种挑战。Redis天然支持计数功能而且计数的性能也非常好，可以说是计数器系统的重要选择。

4、社交网络

    赞/踩、粉丝、共同好友/喜好、推送、下拉刷新等是社交网站的必备功能，由于社交网站访问量通常比较大，而且传统的关系型数据不太适合保存这种类型的数据，Redis提供的数据结构可以相对比较容易地实现这些功能。

5、消息队列系统

    消息队列系统可以说是一个大型网站的必备基础组件，因为其具有业务解耦、非实时业务削峰等特性。Redis提供了发布订阅功能和阻塞队列的功能，虽然和专业的消息队列比还不够足够强大，但是对于一般的消息队列功能基本可以满足。

Redis不适用场景

    站在数据规模的角度看，数据可以分为大规模数据和小规模数据。Redis的数据是存放在内存中的，如果数据量非常大，例如每天有几亿的用户行为数据，使用Redis来存储的话，经济成本相当的高。

## 分布式锁用法

先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

**如果在setnx之后执行expire之前进程意外crash或者要重启维护了，那会怎么样？**

set指令有非常复杂的参数，这个应该是可以同时把setnx和expire合成一条指令来用的

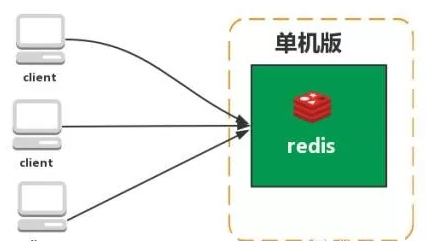
## Expire过期机制

## 复制机制

## 集群部署方式

**单机版、主从复制、哨兵、集群（proxy 型）、集群（直连型**

**单机版**

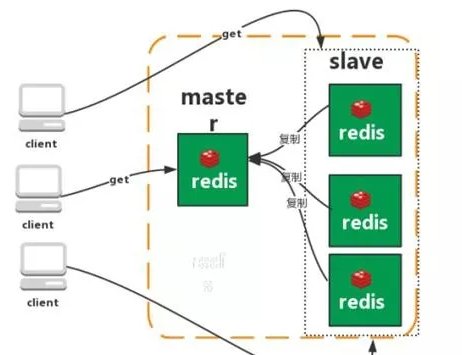


特点：简单

问题：

1、内存容量有限 2、处理能力有限 3、无法高可用。

**主从复制**

****

Redis 的复制（replication）功能允许用户根据一个 Redis 服务器来创建任意多个该服务器的复制品，其中被复制的服务器为主服务器（master），而通过复制创建出来的服务器复制品则为从服务器（slave）。 只要主从服务器之间的网络连接正常，主从服务器两者会具有相同的数据，主服务器就会一直将发生在自己身上的数据更新同步 给从服务器，从而一直保证主从服务器的数据相同。

特点：

1、master/slave 角色

2、master/slave 数据相同

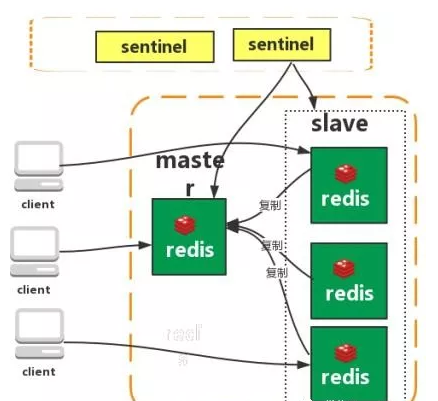
3、降低 master 读压力在转交从库

问题：

无法保证高可用

没有解决 master 写的压力

**哨兵**

****

Redis sentinel 是一个分布式系统中监控 redis 主从服务器，并在主服务器下线时自动进行故障转移。其中三个特性：

监控（Monitoring）：    Sentinel  会不断地检查你的主服务器和从服务器是否运作正常。

提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。

自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作。

特点：

1、保证高可用

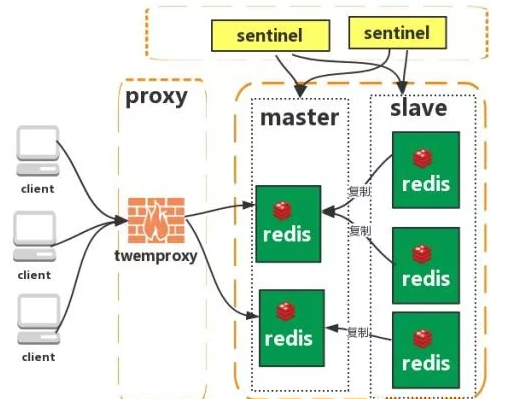
2、监控各个节点

3、自动故障迁移

缺点：主从模式，切换需要时间丢数据

没有解决 master 写的压力

**集群（proxy 型）：**

****

Twemproxy 是一个 Twitter 开源的一个 redis 和 memcache 快速/轻量级代理服务器； Twemproxy 是一个快速的单线程代理程序，支持 Memcached ASCII 协议和 redis 协议。

特点：1、多种 hash 算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins

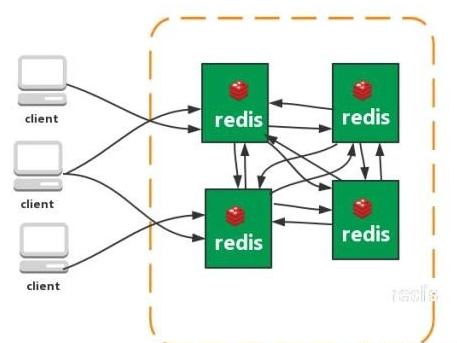
2、支持失败节点自动删除

3、后端 Sharding 分片逻辑对业务透明，业务方的读写方式和操作单个 Redis 一致

缺点：增加了新的 proxy，需要维护其高可用。

failover 逻辑需要自己实现，其本身不能支持故障的自动转移可扩展性差，进行扩缩容都需要手动干预

**集群（直连型）：**

****

从redis 3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。

特点：

1、无中心架构（不存在哪个节点影响性能瓶颈），少了 proxy 层。

2、数据按照 slot 存储分布在多个节点，节点间数据共享，可动态调整数据分布。

3、可扩展性，可线性扩展到 1000 个节点，节点可动态添加或删除。

4、高可用性，部分节点不可用时，集群仍可用。通过增加 Slave 做备份数据副本

5、实现故障自动 failover，节点之间通过 gossip 协议交换状态信息，用投票机制完成 Slave到 Master 的角色提升。

缺点：

1、资源隔离性较差，容易出现相互影响的情况。

2、数据通过异步复制,不保证数据的强一致性

## 负载均衡

## 批量操作数据

## 持久化机制

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

Redis 提供了两种持久化方式:RDB（默认） 和AOF

RDB持久化原理：通过bgsave命令触发，然后父进程执行fork操作创建子进程，子进程创建RDB文件，根据父进程内存生成临时快照文件，完成后对原有文件进行原子替换（定时一次性将所有数据进行快照生成一份副本存储在硬盘中）

优点：是一个紧凑压缩的二进制文件，Redis加载RDB恢复数据远远快于AOF的方式。

缺点：由于每次生成RDB开销较大，非实时持久化，

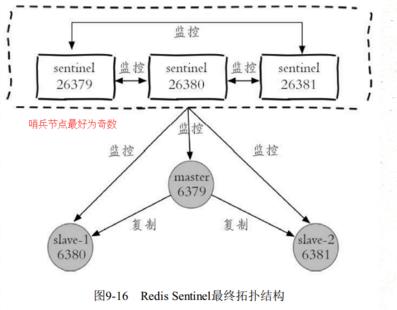
AOF持久化原理：开启后，Redis每执行一个修改数据的命令，都会把这个命令添加到AOF文件中。

优点：实时持久化。

缺点：所以AOF文件体积逐渐变大，需要定期执行重写操作来降低文件体积，加载慢

## Redis挂了怎么办

何谓哨兵模式？就是通过哨兵节点进行自主监控主从节点以及其他哨兵节点，发现主节点故障时自主进行故障转移。



## 缓存更新策略（即如何让缓存和mysql保持一致性）

* 1. **key过期清除（超时剔除）策略**
  2. **2、Redis的内存淘汰策略**

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。

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，移除最近最少使用的key。

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。

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，移除最近最少使用的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。

## 热点key优化



## 缓存击穿、缓存雪崩

缓存穿透

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。一些恶意的请求会故意查询不存在的key,请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

如何避免？

1：对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。

2：对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤。

缓存雪崩

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，会给后端系统带来很大压力。导致系统崩溃。

如何避免？

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期

3：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

## 缓存更新策略

1、缓存【失效】：客户端请求数据先从缓存中查询，如果没有再查询数据库，最后将数据放入缓存

2、缓存【命中】：客户端从缓存中直接取到数据，返回结果

3、缓存【更新】：客户端写入数据到数据库，成功之后，让缓存失效（下次请求时从缓存中拿不到，则查询数据库，再放入缓存）

4、为什么不采取更新后删除缓存策略？

      防止并发写操作导致脏数据

5、为什么不删除缓存后再更新数据库？

      两个并发请求，一个读操作，一个写操作，如果先删除缓存，读操作会将【旧数据】写入缓存，写操作【更新数据】后也不会更新缓存，导致脏数据一直存在

## Redis和memcached的区别

(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

## 经常更新的数据是否可以用缓存

## 事务

## 管道

## bitMap

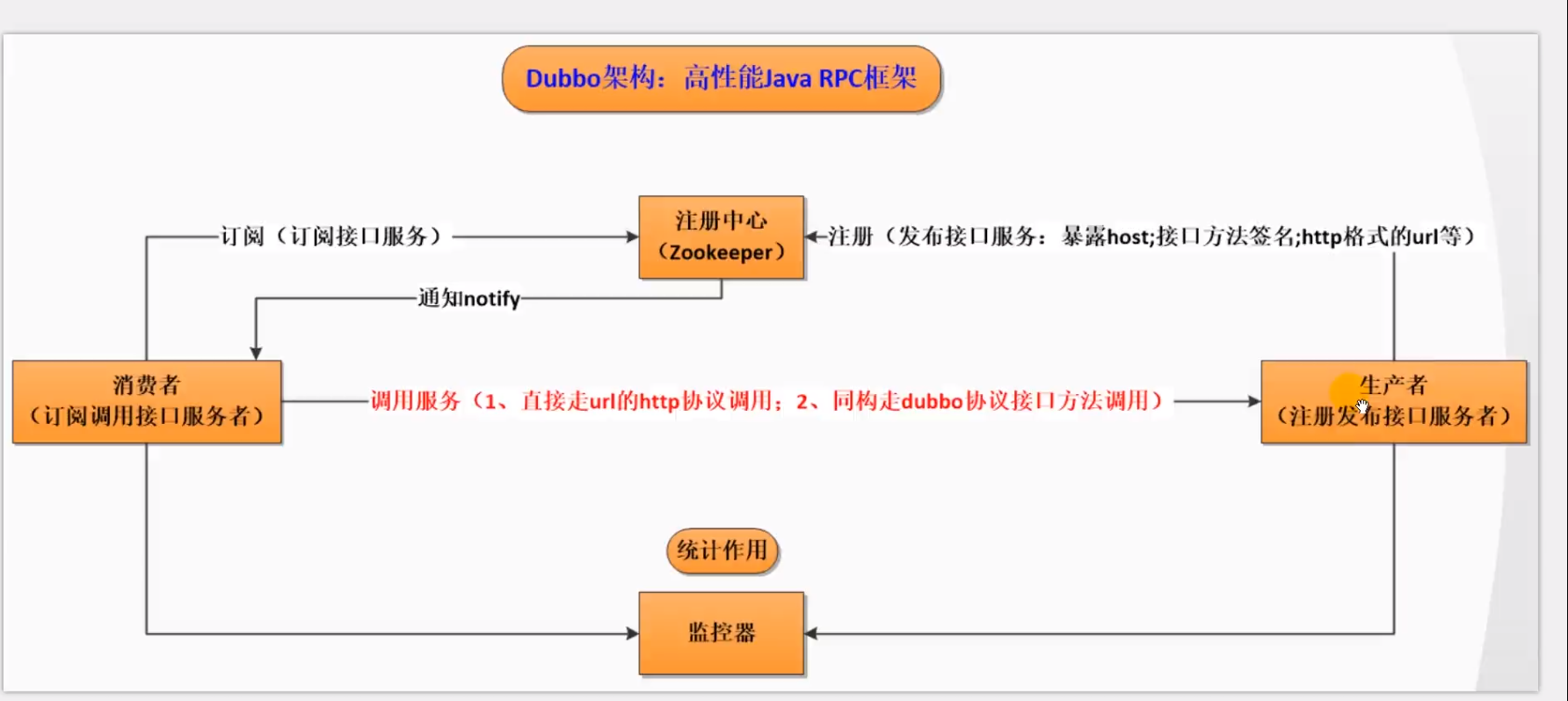
# Elastic-job

# zookeeper

# Dubbo

## 架构图





|  |  |
| --- | --- |
| 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 | 统计服务的调用次数和调用时间的监控中心 |
| Container | 服务运行容器 |

**调用关系说明**

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 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。
5. 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。
6. 服务消费者和提供者，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。

Dubbo 架构具有以下几个特点，分别是连通性、健壮性、伸缩性、以及向未来架构的升级性。

## 负载均衡

## 服务熔断、降级

## 服务升级

## 服务网关

# springcloud

# 分布式

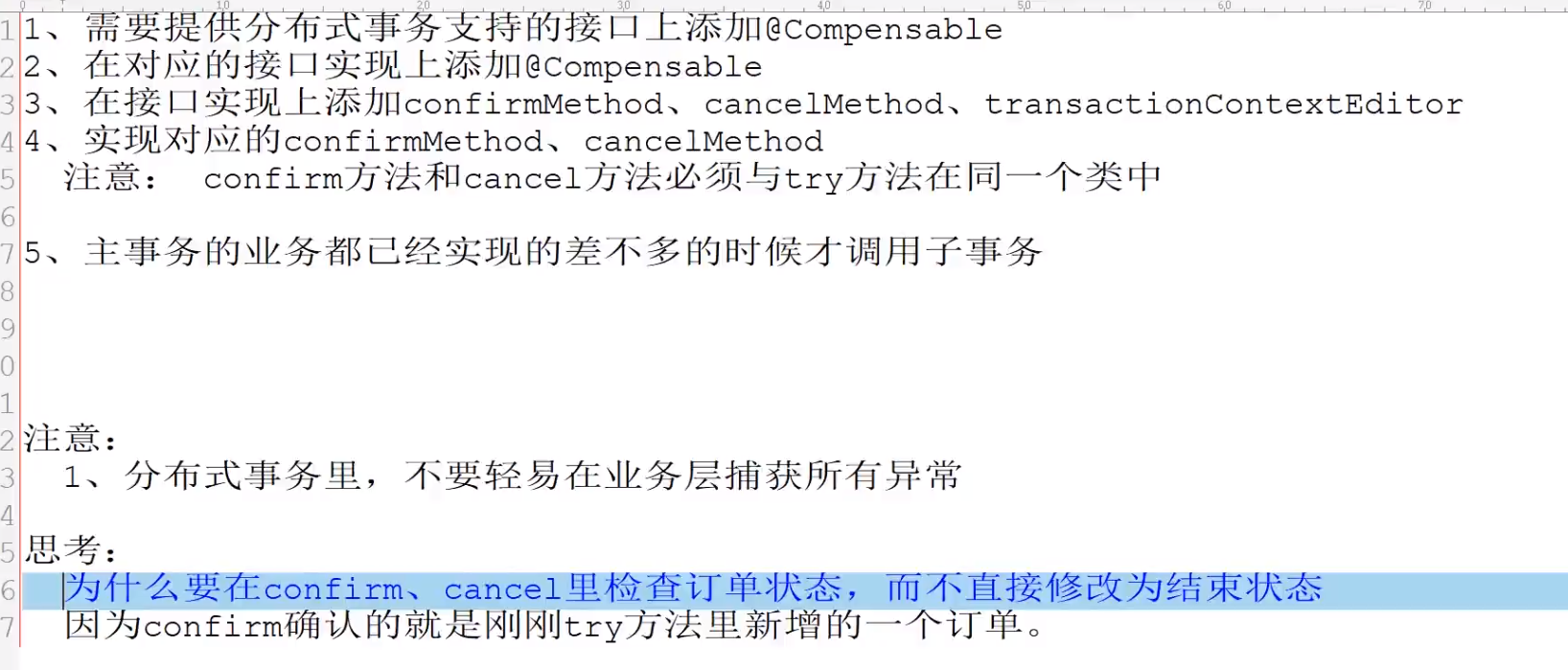
## 分布式优势、弊端、区别

## 微服务和SOA

## 微服务的好处

## 分布式事务解决

### TCC



## 幂等性解决方案

## 分布式Session

## 限流算法

## 集群有服务器挂了或者上线怎么处理

## 缓存更新问题

## 如何负载均衡

## CAP

## BASE

## 分布式quartz

# MQ

## MQ的应用场景

异步处理，应用解耦，流量削锋和消息通讯

## kafka

## artemis

## RabbitMQ

## RabbitMQ集群

rabbitmq有三种模式：单机模式，普通集群模式，镜像集群模式

1.1） 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

（1.2）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。但是你创建的queue，只会放在一个rabbtimq实例上，但是每个实例都同步queue的元数据。完了你消费的时候，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。

这种方式确实很麻烦，也不怎么好，没做到所谓的分布式，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个queue所在实例消费数据，前者有数据拉取的开销，后者导致单实例性能瓶颈。

而且如果那个放queue的实例宕机了，会导致接下来其他实例就无法从那个实例拉取，如果你开启了消息持久化，让rabbitmq落地存储消息的话，消息不一定会丢，得等这个实例恢复了，然后才可以继续从这个queue拉取数据。

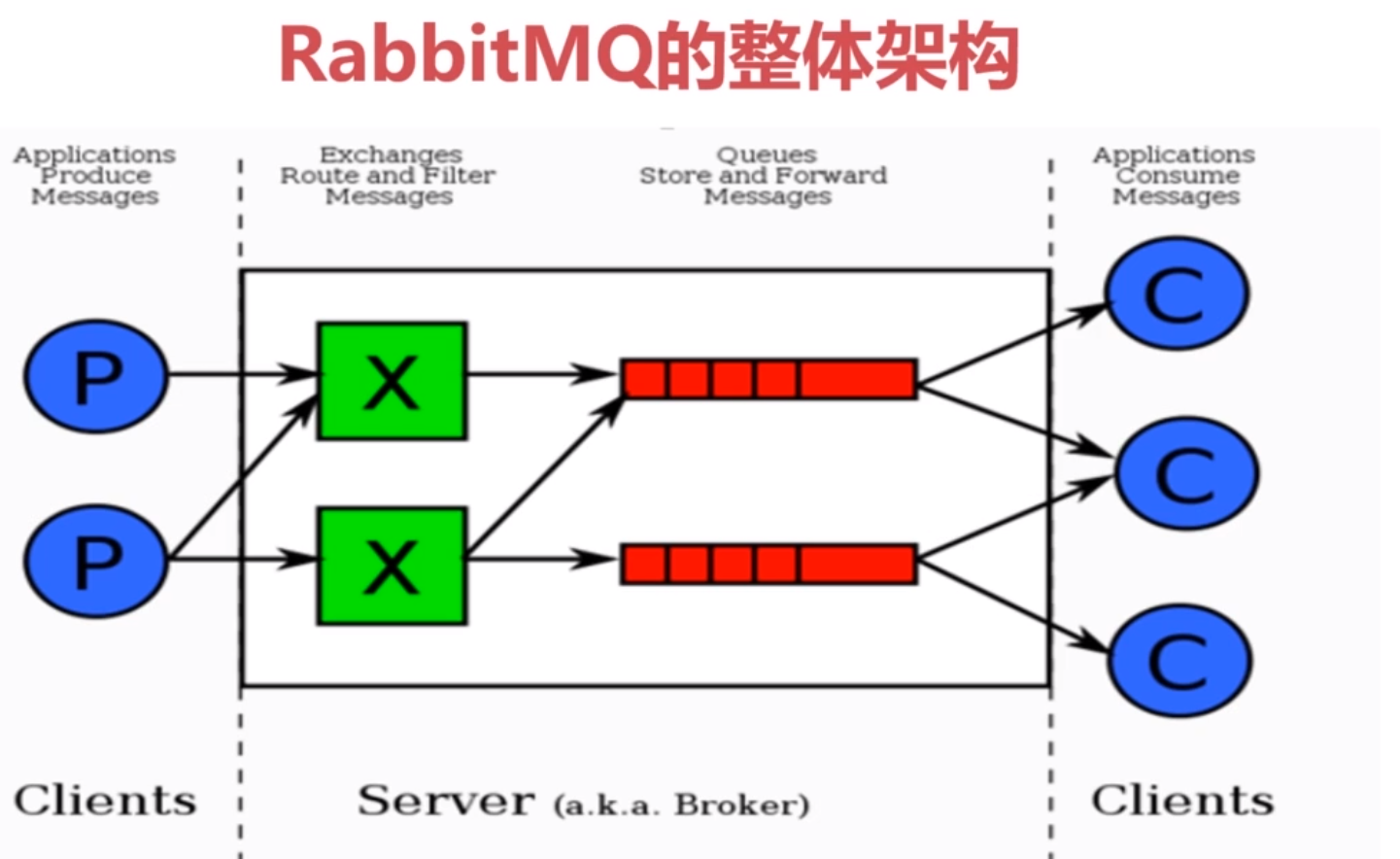
（1.3）镜像集群模式

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，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，然后每次你写消息到queue的时候，都会自动把消息到多个实例的queue里进行消息同步。

这样的话，好处在于，你任何一个机器宕机了，没事儿，别的机器都可以用。坏处在于，第一，这个性能开销也太大了吧，消息同步所有机器，导致网络带宽压力和消耗很重！第二，这么玩儿，就没有扩展性可言了，如果某个queue负载很重，你加机器，新增的机器也包含了这个queue的所有数据，并没有办法线性扩展你的queue

那么怎么开启这个镜像集群模式呢？我这里简单说一下，避免面试人家问你你不知道，其实很简单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，指定的时候可以要求数据同步到所有节点的，也可以要求就同步到指定数量的节点，然后你再次创建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

## RabbitMQ架构



## 如何保证消息不丢失

1.消息持久化

2.ACK确认机制

3.设置集群镜像模式

4.消息补偿机制

持久化：

1） Exchange 设置持久化

2）Queue 设置持久化

3）Message持久化发送：发送消息设置发送模式deliveryMode=2，代表持久化消息

补偿机制：

为什么还要消息补偿机制呢？难道消息还会丢失，没错，系统是在一个复杂的环境，不要想的太简单了，虽然以上的三种方案，基本可以保证消息的高可用不丢失的问题，

但是作为有追求的程序员来讲，要绝对保证我的系统的稳定性，有一种危机意识。

比如：持久化的消息，保存到硬盘过程中，当前队列节点挂了，存储节点硬盘又坏了，消息丢了，怎么办？

产线网络环境太复杂，所以不知数太多，消息补偿机制需要建立在消息要写入DB日志，发送日志，接受日志，两者的状态必须记录。

然后根据DB日志记录check 消息发送消费是否成功，不成功，进行消息补偿措施，重新发送消息处理。

## 消息100%可靠性投递解决方案

## 保证消息不被重复消费

1.当拿到这个消息做数据库的insert操作。那就容易了，给这个消息做一个唯一主键，

那么就算出现重复消费的情况，就会导致主键冲突，避免数据库出现脏数据。

3.如果上面两种情况还不行，准备一个第三方存储,来做消费记录。以redis为例，给消息分配一个全局id，

只要消费过该消息，将<id,message>以K-V形式写入redis。那消费者开始消费前，先去redis中查询有没消费记录即可。

# Linux

## 常用命令

## 实时查看Tomcat日志

## 远程copy文件

## 查看端口

## 查看、杀死进程

## 查看内存占用

## 项目错误排查

## 打开超大文件

# 网络

# Jekins\docker\k8s

# Python\shell\go

# Hive\hadoop\hbase\spark\hdfs

# 项目问题

## 秒杀、超卖

## 购物车实现

## 订单退款操作

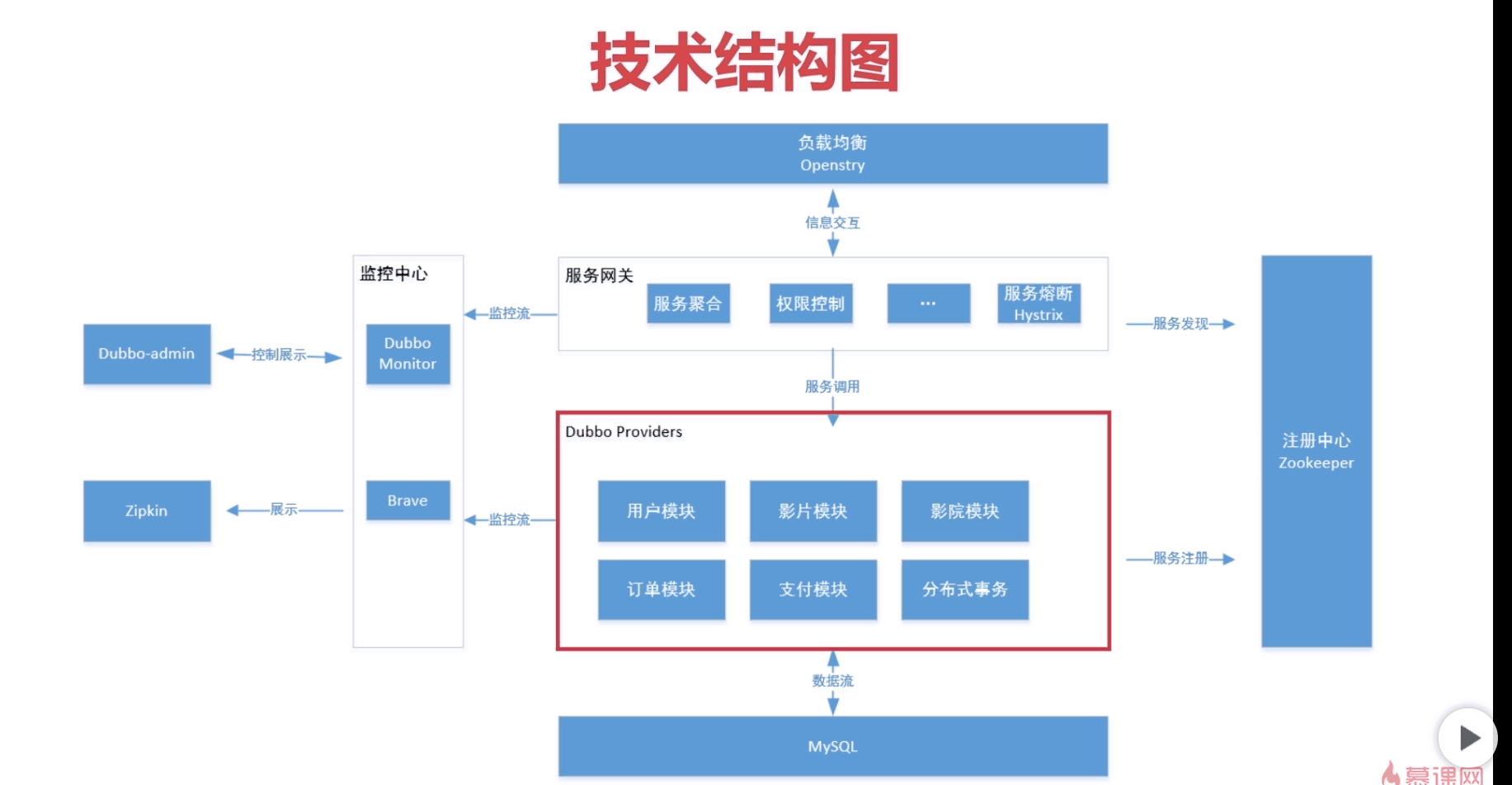
## 物流

## 订单模块调用的微服务

## 电影下单模块调用的微服务

## 消息可靠性

## 项目架构图



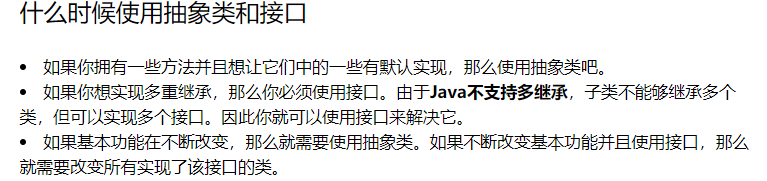
## 多实例令牌桶

# Java基础

## ==、Equals、hashcode区别

## 接口和抽象类





## 深克隆和浅克隆

**浅克隆**：创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。

**深克隆**：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址。

总之深浅克隆都会在堆中新分配一块区域，区别在于对象属性引用的对象是否需要进行克隆（递归性的）。

## 强引用和弱引用

## Error和Exception

## 把对象按某个属性排序

# Java web

## SSO实现

## session共享除了用redis还有什么办法

## Session的实现机制

## Cookie禁用

## Cookie与Session的区别

## RestfulAPI理解

## nginx底层原理

## http

## tcp和udp的区别

## tcp连接过程

# 前端

## 闭包理解、以及解决的问题

# 最困难的事

# 职业规划

# 学习方式、看什么书

# 自我评价

# 有什么想问的

# 谈薪资

# 需要注意的问题