目录

[自我介绍 1](#_Toc92381346)

[项目介绍 1](#_Toc92381347)

[面试经验 1](#_Toc92381348)

[1. JVM 2](#_Toc92381349)

[1.内存（运行时数据区） 2](#_Toc92381350)

[2.类加载 4](#_Toc92381351)

[1. 谈谈java类加载机制 4](#_Toc92381352)

[3.GC 5](#_Toc92381353)

[GC三个算法，分代收集讲下 5](#_Toc92381354)

[1. GC中如何判断对象需要被回收？ 6](#_Toc92381355)

[2.JVM中，哪些可作为ROOT对象？ 6](#_Toc92381356)

[4.minor gc运行很频繁的原因 6](#_Toc92381357)

[5.minor gc运行的很慢的原因 6](#_Toc92381358)

[7.java中内存泄露是啥，什么时候出现内存泄露？ 7](#_Toc92381359)

[8、GC是什么? 为什么要有GC? 9](#_Toc92381360)

[9.GC为什么要分为新生代和老年代（增加吞吐量） 10](#_Toc92381361)

[10垃圾收集器 10](#_Toc92381362)

[11. 、内存分配与回收策略 16](#_Toc92381363)

[12. JVM调优 18](#_Toc92381364)

[13.减少Full GC的方法 18](#_Toc92381365)

[JAVA集合框架 18](#_Toc92381366)

[2. ArrayList 18](#_Toc92381367)

[3.Iterator和ListIterator的区别是什么？ 19](#_Toc92381368)

[4.为什么集合类没有实现Cloneable和Serializable接口？ 20](#_Toc92381369)

[5. Java集合类框架的基本接口有哪些？ 20](#_Toc92381370)

[6. Hashmap 21](#_Toc92381371)

[7.、List、Map、Set三个接口存取元素时，各有什么特点？ 27](#_Toc92381372)

[JAVA基础 28](#_Toc92381373)

[1.数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用Array而不是ArrayList？ 28](#_Toc92381374)

[2、String、StringBuffer、StringBuilder的区别 29](#_Toc92381375)

[3、Object若不重写hashCode()的话，hashCode()如何计算出来的？ 29](#_Toc92381376)

[4、==比较的是什么？ 29](#_Toc92381377)

[5.若对一个类不重写，它的equals()方法是如何比较的？ 30](#_Toc92381378)

[8、Java支持的数据类型有哪些？什么是自动拆装箱？ 30](#_Toc92381379)

[9.什么是值传递和引用传递？ 30](#_Toc92381380)

[11、String是最基本的数据类型吗? 33](#_Toc92381381)

[12.int 和 Integer 有什么区别 33](#_Toc92381382)

[13.String 和StringBuffer和StringBuilder的区别 34](#_Toc92381383)

[15、&和&&的区别？ 34](#_Toc92381384)

[计算机基础 35](#_Toc92381385)

[计算机网络 35](#_Toc92381386)

[TCP UDP 35](#_Toc92381387)

[2.HTTP HTTPs 42](#_Toc92381388)

[3 cookie session 区别及联系？ 44](#_Toc92381389)

[数据结构 45](#_Toc92381390)

[1. 队列 45](#_Toc92381391)

[2. 堆 45](#_Toc92381392)

[3. 栈 54](#_Toc92381393)

[4. 树 55](#_Toc92381394)

[操作系统 59](#_Toc92381395)

[1.进程和线程以及区别 59](#_Toc92381396)

[2死锁(deadlock)？ 60](#_Toc92381397)

[3.虚拟内存？优缺点？ 60](#_Toc92381398)

[4.进程间通信有哪些方式？它们的区别？ 61](#_Toc92381399)

[5.进程的调度算法有哪些？ 62](#_Toc92381400)

[多线程 62](#_Toc92381401)

[1.多线程中的i++线程安全吗？为什么？ 62](#_Toc92381402)

[2.启动一个线程是用run()还是start()以及他们的区别? 62](#_Toc92381403)

[3.线程的sleep()方法和yield()方法有什么区别？ 和wait()有什么区别？ 63](#_Toc92381404)

[4.当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？ 63](#_Toc92381405)

[5. 锁 63](#_Toc92381406)

[9.你有哪些多线程开发良好的实践? 66](#_Toc92381407)

[10.线程池 66](#_Toc92381408)

[1.什么是线程池 66](#_Toc92381409)

[2.几种常见的线程池及使用场景 66](#_Toc92381410)

[3.为什么要用线程池？ 67](#_Toc92381411)

[4.使用线程池的好处： 67](#_Toc92381412)

[6.几种重要的参数 67](#_Toc92381413)

[7.线程池的拒绝策略 68](#_Toc92381414)

[10.线程池都有哪几种工作队列 68](#_Toc92381415)

[11.线程和进程各自有什么区别和优劣呢？ 69](#_Toc92381416)

[12.Callable 和 Runnable接口的区别 69](#_Toc92381417)

[13.ReentrantLock 70](#_Toc92381418)

[14单线程多线程区别 71](#_Toc92381419)

[15.保证十个线程的执行顺序 71](#_Toc92381420)

[16.sleep wait join怎么用的 71](#_Toc92381421)

[17 72](#_Toc92381422)

[关键字 72](#_Toc92381423)

[1. Synchronize 72](#_Toc92381424)

[2.Static 74](#_Toc92381425)

[1. static用法 74](#_Toc92381426)

[2.详解static 74](#_Toc92381427)

[一个例子学习静态变量与实例变量。 77](#_Toc92381428)

[静态变量的用法 78](#_Toc92381429)

[注意事项 79](#_Toc92381430)

[用途---单例模式 79](#_Toc92381431)

[其他 79](#_Toc92381432)

[静态方法的用场 80](#_Toc92381433)

[什么是代码块 80](#_Toc92381434)

[四种代码块 80](#_Toc92381435)

[静态内部类 84](#_Toc92381436)

[总结 84](#_Toc92381437)

[彩蛋 ------ 继承+代码块的执行顺序 84](#_Toc92381438)

[3.Violate 85](#_Toc92381439)

[1、介绍一下volatile 85](#_Toc92381440)

[4.Final 86](#_Toc92381441)

[5.Java中static，final,static final的区别 87](#_Toc92381442)

[6..synchronized volatile 区别 88](#_Toc92381443)

[数据库 88](#_Toc92381444)

[1. 数据库的隔离级别 88](#_Toc92381445)

[2. 存储引擎 90](#_Toc92381446)

[1.Mysql中的myisam与innodb的区别 90](#_Toc92381447)

[2.myisam与innodb select count(\*)哪个更快 91](#_Toc92381448)

[3.行级锁和表级锁 91](#_Toc92381449)

[3. 索引 91](#_Toc92381450)

[1. 什么是索引 91](#_Toc92381451)

[2. MySQL主要的索引类型 92](#_Toc92381452)

[3. 数据库索引有哪些，优缺点？ 92](#_Toc92381453)

[4. 为什么不用二叉查找树作为数据库索引？ 92](#_Toc92381454)

[5. B树 93](#_Toc92381455)

[6. B+树 93](#_Toc92381456)

[8. 索引的缺点 94](#_Toc92381457)

[12一个B+树的节点中到底存储多少个元素合适呢？ 94](#_Toc92381458)

[11.哪些数据结构能提高查询速度？MySQL为何选择使用B+树？ 94](#_Toc92381459)

[13.MySQL中B+树的一个节点大小为多大？ 95](#_Toc92381460)

[14一系列问题 95](#_Toc92381461)

[16为什么B+树比B树更适合做数据库索引？ 96](#_Toc92381462)

[18. 聚簇索引和非聚簇索引 96](#_Toc92381463)

[19. 聚集索引和非聚集索引 97](#_Toc92381464)

[20. 什么情况索引会失效： 97](#_Toc92381465)

[21 与红黑树的比较 98](#_Toc92381466)

[4.说一说你能想到的sql语句优化，至少五种 98](#_Toc92381467)

[设计模式 98](#_Toc92381468)

[1.单例模式 99](#_Toc92381469)

[介绍一下单例模式？懒汉式的单例模式如何实现单例？ 99](#_Toc92381470)

[ 饿汉式单例 99](#_Toc92381471)

[ 懒汉式单例（线程不安全） 99](#_Toc92381472)

[ 线程同步式（线程安全） 100](#_Toc92381473)

[ 双重检查锁式 101](#_Toc92381474)

[ 静态内部类式 105](#_Toc92381475)

[ 枚举单例 107](#_Toc92381476)

[2.适配器模式 107](#_Toc92381477)

[1.什么是适配器模式？ 107](#_Toc92381478)

[2.为什么要使用类适配器模式？优缺点？使用场景？ 107](#_Toc92381479)

[ 类适配器模式 108](#_Toc92381480)

[ 对象适配器模式 109](#_Toc92381481)

[ 双向适配器模式 109](#_Toc92381482)

[ 缺省适配器 110](#_Toc92381483)

[总结 111](#_Toc92381484)

[3.工厂模式 113](#_Toc92381485)

[4.代理模式 113](#_Toc92381486)

[算法 119](#_Toc92381487)

[1. 剑指offer算法 119](#_Toc92381488)

[2. 排序算法 158](#_Toc92381489)

[3.查找算法 178](#_Toc92381490)

[4.jdk源码 179](#_Toc92381491)

[SSM 179](#_Toc92381492)

[1. 用到Spring什么特性 结合项目说一下 依赖反转的好处 180](#_Toc92381493)

[2.Spring主要使用了什么模式？ 180](#_Toc92381494)

[3.IOC，AOP的实现原理？ 180](#_Toc92381495)

[4.讲述SpringMVC工作流程流程 180](#_Toc92381496)

[5.为什么使用spring 181](#_Toc92381497)

[6.Spring DI的几种方式 181](#_Toc92381498)

[7.Aop 181](#_Toc92381499)

[9. Spring中aop,ioc怎么实现的。 182](#_Toc92381500)

[中间件 183](#_Toc92381501)

[1.Redis 184](#_Toc92381502)

[1.redis的过期策略以及内存淘汰机制 redis采用的是定期删除+惰性删除策略。 为什么不用定时删除策略? 定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略. 定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redis岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。 184](#_Toc92381503)

[2.缓存穿透 击穿 雪崩 185](#_Toc92381504)

[3.Redis优势和特点 185](#_Toc92381505)

[4.知道Redis的持久化吗？都有什么缺点和优点？具体底层实现呢？ 186](#_Toc92381506)

[3. 消息队列 187](#_Toc92381507)

[1. 什么是消息队列 187](#_Toc92381508)

[2. 为什么使用消息队列 187](#_Toc92381509)

[3. 使用消息队列带来的一些问题 189](#_Toc92381510)

[面经 194](#_Toc92381511)

[猿辅导 194](#_Toc92381512)

[字节跳动 195](#_Toc92381513)

[1.Java AIO、NIO、BIO 195](#_Toc92381514)

[2、HashMap底层实现、链表转红黑树的阈值为什么是8？ 195](#_Toc92381515)

[3、String、StringBuffer、StringBuilder的区别 195](#_Toc92381516)

[4.CAS的实现原理 195](#_Toc92381517)

[5、Java线程状态 195](#_Toc92381518)

[6、Java类加载机制 195](#_Toc92381519)

[7、Redis分布式锁、底层实现原理 195](#_Toc92381520)

[8、Http状态码302/403 195](#_Toc92381521)

[9、HTTPs请求过程 195](#_Toc92381522)

[10、TCP/UDP区别 195](#_Toc92381523)

[11、分布式一致性算法 195](#_Toc92381524)

[12、进程间通信、线程间同步的方式分别 195](#_Toc92381525)

[13、虚拟内存的实现原理 195](#_Toc92381526)

[14、归并排序过程、时空复杂度 195](#_Toc92381527)

[15、用户态、内核态的区别。 195](#_Toc92381528)

[编程：字符串的解压。 195](#_Toc92381529)

[字节跳动成都研发中心秋招提前批123面面经 196](#_Toc92381530)

[2. java的static讲下，然后有6种用法，我说了4种，后来网上搜也只有五种呀 196](#_Toc92381531)

[3. 类方法，代码，静态变量，实例，实例变量，局部变量分别存到哪里 196](#_Toc92381532)

[4. JVM怎么表示一个对象 196](#_Toc92381533)

[6. https握手流程 197](#_Toc92381534)

[7. http格式 197](#_Toc92381535)

[8. http1.0 和 2.0 197](#_Toc92381536)

[9. 保持长连接的是那个字段（keep-alive） 197](#_Toc92381537)

[10. fork（）函数讲下，又结合项目来了一次 197](#_Toc92381538)

[11. 线程调度算法，linux采用哪种 197](#_Toc92381539)

[12. 进程线程区别，进程通信 197](#_Toc92381540)

[14. 内连接，外（左右）链接，全连接的区别 197](#_Toc92381541)

[15. join on 后and条件和where后加and条件区别 197](#_Toc92381542)

[17. 算法题：字符串转Int，如果越界就返回0 198](#_Toc92381543)

[4. 场景题目：设计一个短域名服务：短信存不了太长网站，需要弄成短域名，你该如何设计一个服务，可以为全国的网址服务。我说的将所有网址存数据库，用hashcode来当短域名，如果冲突就拼接随机数，表大就分表，当时忘记说再加缓存了，笨 198](#_Toc92381544)

[7. 索引，复合索引结构 199](#_Toc92381545)

[8. sql题，写了个连表查询外加模糊查询 199](#_Toc92381546)

[9. 算法：镜像二叉树 199](#_Toc92381547)

[10. 算法：将List转成tree：（当前节点id，节点名字，父节点id），（当前节点id，节点名字，父节点id），（当前节点id，节点名字，父节点id）=》转成一个树，说自己定义数据结构，设计用例，要能编译运行，我用hashmap存，然后写到差不多了他说不用写了 199](#_Toc92381548)

[字节面经 199](#_Toc92381549)

[操作系统 199](#_Toc92381550)

[计算机网络 199](#_Toc92381551)

[数据结构与算法 199](#_Toc92381552)

[美团：一面 201](#_Toc92381553)

[小米 202](#_Toc92381554)

[猿辅导 202](#_Toc92381555)

中国数值水池创新专项实验室

黑龙江省哈尔滨市南通大街145号

校二等奖学金（硕士） 2018 10

校二等奖学金（本科） 2014 10

大学英语四级 2014 2

y\*\*\*\*\*\_hrbeu@foxmail.com 已发送，感谢楼主~

算法整明白

# 自我介绍

我叫于楠，来自于哈尔滨工程大学，本科和硕士都是这个学校的，专业都是计算机。我在学校有一年半的项目开发经验，主要进行的是后台开发，前端和数据库也有涉及，发现自己对后台开发比较感兴趣，所有想应聘贵公司的后台开发工程师岗位。

# 项目介绍

第一个项目叫分表计电监管平台，此项目目的在改善天津北辰区的环境状况，对区内所有工厂的设备进行监管，报警，限产及大数据分析等功能。

1.作为核心开发人员，独立完成系统功能，后台数据分析处理，前台展示（ECharts，layui等）。

2.基于UDP协议的数据解析，并持久化数据。 (（网络编程，设置一个定时器每天向设备发送请求，解析得

到的数据）)

3.基于Redis显著提升数据查询时间。 （是因为我提到我们的业务量比较大，每天大概有几百万的新数据生成）

4.基于K-means，关联分析，层次聚类等方法进行数据分析。

. 使用redis提升了多少

项目中遇到的最大难点，是怎样解决的，用了什么方法，为什么使用这种方法可以解决当前难点。。。

# 面试经验

1.面试官：你还有什么问题？

他们平时怎么学习新技术？

我在这将来可能要使用的技术、负责的业务。

你个项目组能承担什么角色?可以学到什么呢？

这个岗位需要什么技术

这个岗位我去了之后 发展路线是什么样的

公司对于我们进去有没有技术的培养呢

提问环节:我可以进什么样的项目组 ，以后能有什么样的发展。但是在让他提问的环节，他提出要加我的微信，我就答应了。之后当天晚上，这哥们就把当天他没有回答好的问题，通过微信又回答了一遍发给了我。这让我挺惊喜，或许回去他也是查了资料才解答的，但是这至少证明了两点：记住哪些题目没有答好回去马上查，对自己负责；他对这个岗位很在意，有比较强烈的入职心态。最后在PK中就是这一点让我们选择了他。

1. 先去小公司试水，如果问题不大，可以尝试投大公司。另外记得刷一下大厂的面经，如果觉得差距比较大，慎投，至少等你把面经商遇到的问题都解决了再继续下一步吧
2. 你有什么行业背景/岗位经验（最有料的写在最前面，配合数据）你参与过什么重要的项目（没有的话就不写、有的话拿出数据或实例）针对应聘的岗位你有什么拿得出手的优势、技能职业动机/发展目标（你想在这个岗位做出什么样的成绩）介绍你的热情/与公司、产品的交集（XX重度用户，有XX年使用经验）

面试手机可以录音

1. 自我介绍  
   2.为什么应聘这个岗位？  
   3.别的岗位你考虑吗？

优先考虑这个岗位，其它的岗位也可以考虑  
4.你的职业规划  
5.你觉得你做这个岗位的优势？  
7.你有什么想问的？  
8.你有女朋友吗？

10.你做的最有成就感的事？

11. 工作中犯的最大的错误？

12. 工作中有什么挫败感的事情？是怎么解决的？

13.你还投递过什么别的公司吗？方便说一下吗？

# **JVM**

## 1.内存（运行时数据区）

Java 内存区域主要分为线程私有的和线程共享的。线程私有的包括程序计数器，虚拟机栈，本地方法栈。线程共享的包括堆，方法区，直接内存。

程序计数器：是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完。

另外，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

从上面的介绍中我们知道程序计数器主要有两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

**注意：**程序计数器是唯一不会出现 OutOfMemoryError 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

Java 虚拟机栈：与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型。

Java 内存可以粗糙的区分为堆内存（Heap）和栈内存（Stack）其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分。 （实际上，Java虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有局部变量表、操作数栈、动态链接、方法出口信息）

局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。

* StackOverFlowError： 若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出StackOverFlowError异常。
* OutOfMemoryError： 若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出OutOfMemoryError异常。

[堆](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=%e5%a0%86)

所有对象都在这里分配内存，是垃圾收集的主要区域（"GC 堆"）。

现代的垃圾收集器基本都是采用分代收集算法，其主要的思想是针对不同类型的对象采取不同的垃圾回收算法。可以将堆分成两块：

* 新生代（Young Generation）
* 老年代（Old Generation）

堆不需要连续内存，并且可以动态增加其内存，增加失败会抛OutOfMemoryError 异常。

[方法区](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=%e6%96%b9%e6%b3%95%e5%8c%ba)：

用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

和堆一样不需要连续的内存，并且可以动态扩展，动态扩展失败一样会抛出 OutOfMemoryError 异常。

对这块区域进行垃圾回收的主要目标是对常量池的回收和对类的卸载，但是一般比较难实现。

HotSpot 虚拟机把它当成永久代来进行垃圾回收。但很难确定永久代的大小，因为它受到很多因素影响，并且每次 Full GC 之后永久代的大小都会改变，所以经常会抛出 OutOfMemoryError 异常。为了更容易管理方法区，从 JDK 1.8 开始，移除永久代，并把方法区移至元空间，它位于本地内存中，而不是虚拟机内存中。

方法区是一个 JVM 规范，永久代与元空间都是其一种实现方式。在 JDK 1.8 之后，原来永久代的数据被分到了堆和元空间中。元空间存储类的元信息，静态变量和常量池等放入堆中。

## 2.类加载

### 谈谈java类加载机制

Java类加载主要分为三个步骤：加载、连接和初始化。

加载是将class文件读入内存(将其放入运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。即程序中使用任何类时，系统都会为之建立一个java.lang.Class对象，系统中所有的类都是java.lang.Class的实例)

连接负责将class文件合并到JRE中。其中连接又分三个步骤，分别为验证，准备和解析。验证确保确保加载的类信息符合JVM规范，无安全方面的问题。准备为类的静态变量分配内存，并初始化为默认值。解析将class文件中的符号引用替换成直接引用。

初始化主要是对静态变量赋予正确的初始值。

当程序主动使用某个类时，如果该类还未被加载到内存中，则JVM会通过**加载、连接、初始化**3个步骤来对该类进行初始化。如果没有意外，JVM将会连续完成3个步骤，所以有时也把这个3个步骤统称为类加载或类初始化。

类的加载由类加载器完成，JVM提供的类加载器叫做系统类加载器，此外还可以通过继承ClassLoader基类来自定义类加载器。

## 3.GC

### GC三个算法，分代收集讲下

(1)标记-清除算法:分为标记和清除两阶段,首先标记出所有需要回收的对象，然后统一回收所有被标记的对象。（之所以说他是最基础的收集算法，是因为后续的收集算法都是基于这种思路并且对其不足进行改进而得到的）

缺点:1.效率问题，标记和清除的效率都不高。

2.是空间问题，清除后产生了大量不连续的内存碎片，导致在程序运行过程中需要分配较大对象的时候，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。

(2)复制算法:将可用内存按容量划分为大小相等的两块，每次只用其中一块。当这块内存用完了，就将还存活的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

好处：效率高于标记-清除，但效率也不是很高。不会产生过多碎片，每次都对半块内存操作。

缺点：内存缩小为原来的一半

(3)标记-整理算法：标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

优点：自带整理功能，这样不会产生大量不连续的内存空间，适合老年代的大对象存储。

(4)分代收集算法  
 当前商业虚拟机的垃圾收集都采用分代收集。就是将上述三种算法 整合了一下。根据对象的存活周期的不同，将内存划分为几块。一般是把java堆分成新生代和老年代，这样就可以根绝各个年代的特点采取最适当的收集算法。

1、在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法。只需要付出少量存活对象的复制成本就可以完成收集。  
 2、老年代中因为对象存活率高、没有额外空间对他进行分配担保，就必须用标记-清除或者标记-整理。

### GC中如何判断对象需要被回收？

采用引用计数法和可达性分析法。引用计数法是给对象中添加一个引用计数器，每当一个地方引用这个对象时，计数器值+1；当引用失效时，计数器值-1。任何时刻计数值为0的对象就是不可能再被使用的。（这种算法使用场景很多，但是，Java中却没有使用这种算法，因为这种算法很难解决对象之间相互引用的情况）

可达性分析法通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链（即GC Roots到对象不可达）时，则证明此对象是不可用的。

### 2.JVM中，哪些可作为ROOT对象？

(1)虚拟机栈（栈帧中的局部变量区，也叫做局部变量表）中引用的对象。

(2)方法区中的类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JNI(Native方法)引用的对象。

### 4.minor gc运行很频繁的原因

产生了太多朝生夕灭的对象导致需要频繁minor gc ，新生代空间设置的比较小

### 5.minor gc运行的很慢的原因

(1)新生代空间设置过大。

(2)对象引用链较长，进行可达性分析时间较长。

(3) 新生代survivor区设置的比较小，清理后剩余的对象不能装进去需要移动到老年代，造成移动开销。

(4)内存分配担保失败，由minor gc转化为full gc

(5)采用的垃圾收集器效率较低，比如新生代使用serial收集器

6.简述GC算法

### 7.java中内存泄露是啥，什么时候出现内存泄露？

内存泄露是指一个不再被程序使用的对象或变量还在内存中占有存储空间。在C/C+ +语言中，内存的分配与释放是由开发人员来负责的，如果开发人员忘记释放已分配的内存就会造成内存泄露。而在Java语言中引进了垃圾回收机制，由垃圾回收器负责回收不再使用的对象, 既然有垃圾回收器来负责回收垃圾，那么是否还会存在内存泄露的问题呢？

其实，在Java语言中，判断一个内存空间是否符合垃圾回收的标准有两个：第一，给对象赋予了空值null,以后再没有被使用过；第二，给对象赋予了新值，重新分配了内存空间。 一般来讲，内存泄露主要有两种情况：一是在堆中申请的空间没有被释放；二是对象已不再被使用，但还仍然在内存中保留着。垃圾回收机制的引入可以有效地解决第一种情况；而对于第二种情况，垃圾回收机制则无法保证不再使用的对象会被释放。因此，Java语言中的内存泄露主要指的是第二种情况。

下面通过一个示例来介绍Java语言中的内存泄露：

Vector v = new Vector( 10)； for (int i = 1 ； i < 10 ； i++ ) {

Object o = new Object ()；

v. add( o)；

}

在上述例子的循环中，不断创建新的对象加到Vector对象中，当退出循环后，o的作用域将会结束，但是由于v在使用这些对象，因此垃圾回收器无法将其回收，此时就造成了内存泄露。只有将这些对象从Vector中删除才能释放创建的这些对象。

在Java语言中，容易引起内存泄露的原因很多，主要有以下几个方面的内容：

1. 静态集合类，例如HashMap和Vector。如果这些容器为静态的，由于它们的生命周期与程序一致，那么容器中的对象在程序结束之前将不能被释放，从而造成内存泄露，如上例所示。
2. 各种连接，例如数据库连接、网络联接以及I0连接等。在对数据库进行操作的过程 中，首先需要建立与数据库的连接，当不再使用时，需要调用close方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程 中，对ConnectionA Statement或ResultSet不显式地关闭，将会造成大量的对象无法被回收，从而引起内存泄露。
3. 监听器。在Java语言中，往往会使用到监听器。通常一个应用中会用到多个监听器, 但在释放对象的同时往往没有相应地删除监听器，这也可能导致内存泄露。
4. 变量不合理的作用域。一般而言，如果一个变量定义的作用范围大于其使用范围，很有可能会造成内存泄露，另一方面如果没有及时地把对象设置为null,很有可能会导致内存泄露的发生，示例如下：

class Server {

private String msg；

public void recieveMsg( ) {

readFromNet( ) ;//从网络接收数据保存到msg中

saveDB()； //把msg保存到数据库中

}

在上述伪代码中，通过readFromNet()方法接收的消息保存在变量msg中，然后调用 saveDB()方法把msg的内容保存到数据库中，此时msg已经没用了，但是由于msg的生命周期与对象的生命周期相同，此时msg还不能被回收，因此造成了内存泄露。对于这个问题，有 如下两种解决方法：第一种方法，由于msg的作用范围只在recieveMsg()方法内，因此可以把 msg定义为这个方法的局部变量，当方法结束后，msg的生命周期就会结束，此时垃圾回收器 就可以回收msg的内容了；第二种方法，在使用完msg后就把msg设置为null,这样垃圾回收器也会自动回收msg内容所占的内存空间。

1. 单例模式可能会造成内存泄露。单例模式的实现方法有很多种，下例中所使用的单例 模式就可能会造成内存泄露：

class BigClass {

//class body

}

class Singleton {

private BigClass be；

private static Singleton instance = new Singleton( new BigClass())；

private Singleton( BigClass be) | this, be = be； |

public Singleton getlnstance( ) j

return instance ；

在上述实现的单例模式中，Singleton存在一个对对象BigClass的引用，由于单例对象以静态变量的方式存储，因此它在JVM的整个生命周期中都存在，同时由于它有一个对对象Big­Class 的引用，这样会导致BigClass类的对象不能够被回收。

### 8、GC是什么? 为什么要有GC?

GC是垃圾回收，它的主要作用是回收程序中不再使用的内存。在使用C/C++语言进行程序开发时，开发人员必须非常仔细地管理好内存的分配与释放，如果忘记或者错误地释放内存往往会导致程序运行不正常甚至是程序崩溃。为了减轻开发人员的工作，同时增加系统的安全性与稳定性，Java语言提供了垃圾回收器来自动检测对象的作用域，可自动地把不再被使用的存储空间释放掉。具体而言，垃圾回收器要负责完成3项任务：分配内存、确保被引用对象的内存不被错误地回收以及回收不再被引用的对象的内存空间。

垃圾回收器的存在一方面把开发人员从释放内存的复杂工作中解脱出来，提高了开发人员 的生产效率；另一方面，对开发人员屏蔽了释放内存的方法，可以避免因开发人员错误地操作内存而导致应用程序的崩溃，保证了程序的稳定性。但是，垃圾回收也带来了问题，为了实现垃圾回收，垃圾回收器必须跟踪内存的使用情况，释放没用的对象，在完成内存的释放后还需要处理堆中的碎片，这些操作必定会增加JVM的负担，从而降低程序的执行效率。

对对象而言，如果没有任何变量去引用它，那么该对象将不可能被程序访问，因此可以认为它是垃圾信息，可以被回收。只要有一个以上的变量引用该对象，该对象就不会被垃圾回收。

对于垃圾回收器来说，它使用有向图来记录和管理堆内存中的所有对象，通过这个有向图 就可以识别哪些对象是“可达的”(有引用变量引用它就是“可达的”)，哪些对象是“不可达的”(没有引用变量引用它就是不可达的)，所有“不可达”对象都是可被垃圾回收的，示例如下：

public class Test {

public static void main( String[ ] a) |

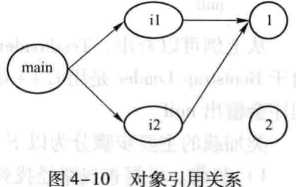
Integer il = new Integer( 1)；

Integer i2 = new Integer(2)；

i2 = il;

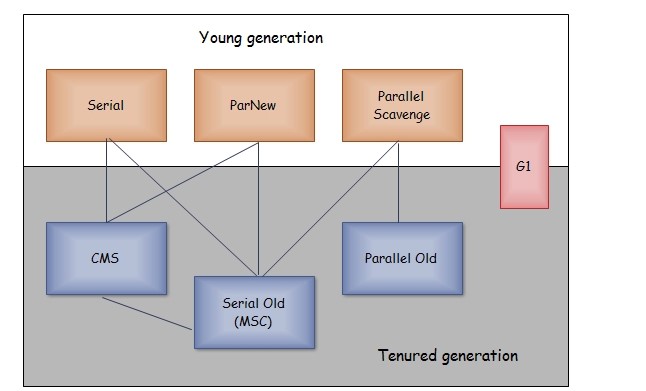
}

上述代码在执行到i2=il后，内存的引用关系如图4-10所示。

此时，如果垃圾回收器正在进行垃圾回收操作，在遍历上述有向图时，资源2所占的内存是不可达的，垃圾回收器就会认为这块内存已经不会再被使用了，因此就会回收该块 内存空间。

### 9.GC为什么要分为新生代和老年代（增加吞吐量）

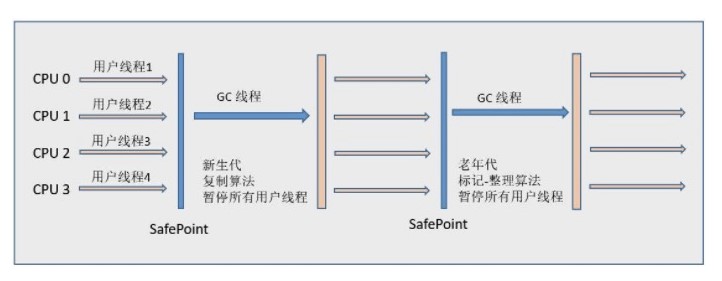
### 10[垃圾收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=%e5%9e%83%e5%9c%be%e6%94%b6%e9%9b%86%e5%99%a8)



以上是 HotSpot 虚拟机中的 7 个垃圾收集器，连线表示垃圾收集器可以配合使用。

* 单线程与多线程：单线程指的是垃圾收集器只使用一个线程，而多线程使用多个线程；
* 串行与并行：串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序；并行指的是垃圾收集器和用户程序同时执行。**除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。**

#### [1. Serial 收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_1-serial-%e6%94%b6%e9%9b%86%e5%99%a8)

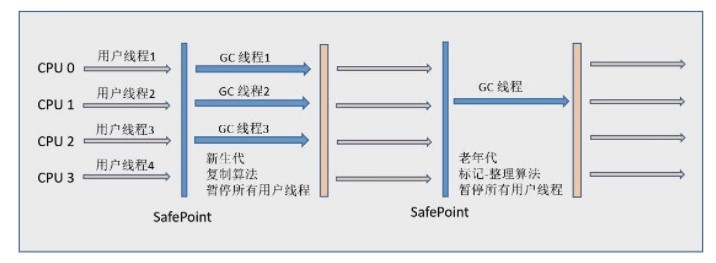


以串行的方式执行，是单线程的收集器，只会使用一个线程进行垃圾收集工作。

优点：简单高效，在单个 CPU 环境下，由于没有线程交互的开销，因此拥有最高的单线程收集效率。

它是 Client 场景下的默认新生代收集器，因为在该场景下内存一般来说不会很大。它收集一两百兆垃圾的停顿时间可以控制在一百多毫秒以内，只要不是太频繁，这点停顿时间是可以接受的。

#### [2. ParNew 收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_2-parnew-%e6%94%b6%e9%9b%86%e5%99%a8)



Serial 收集器的多线程版本。

它是 Server 场景下默认的新生代收集器，除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合使用。

#### [3. Parallel Scavenge 收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_3-parallel-scavenge-%e6%94%b6%e9%9b%86%e5%99%a8)

与 ParNew 一样是多线程收集器。

其它收集器目标是尽可能缩短垃圾收集时用户线程的停顿时间，而它的目标是达到一个可控制的吞吐量，因此它被称为“吞吐量优先”收集器。

吞吐量：CPU 用于运行用户程序的时间占总时间的比值。

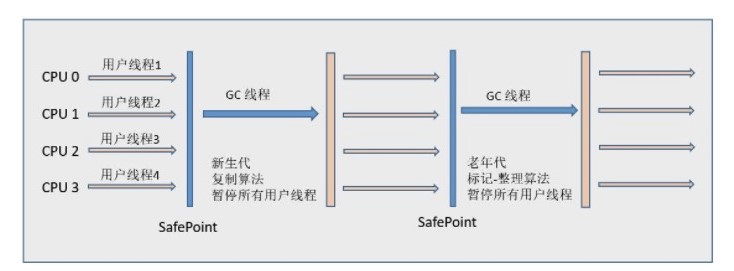
停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验。

而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，适合在后台运算而不需要太多交互的任务。

缩短停顿时间是以牺牲吞吐量和新生代空间来换取的：新生代空间变小，垃圾回收变得频繁，导致吞吐量下降。

可以通过一个开关参数打开 GC 自适应的调节策略（GC Ergonomics），就不需要手工指定新生代的大小（-Xmn）、Eden 和 Survivor 区的比例、晋升老年代对象年龄等细节参数了。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。

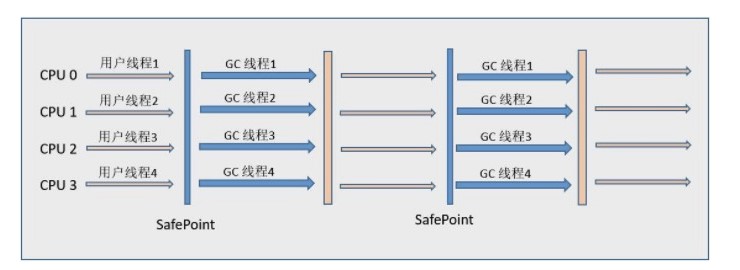
#### [4. Serial Old 收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_4-serial-old-%e6%94%b6%e9%9b%86%e5%99%a8)



Serial 收集器的老年代版本，也是给 Client 场景下的虚拟机使用。如果用在 Server 场景下，它有两大用途：

* 在 JDK 1.5 以及之前版本（Parallel Old 诞生以前）中与 Parallel Scavenge 收集器搭配使用。
* 作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

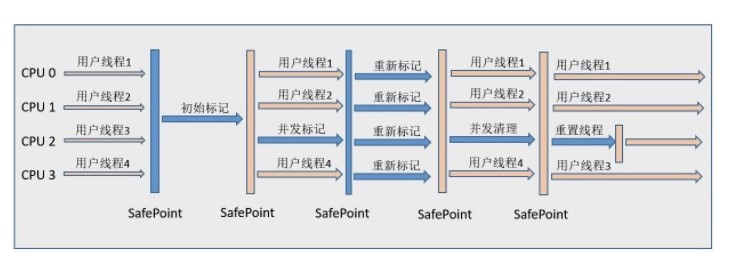
#### [5. Parallel Old 收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_5-parallel-old-%e6%94%b6%e9%9b%86%e5%99%a8)



是 Parallel Scavenge 收集器的老年代版本。

在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

#### [6. CMS 收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_6-cms-%e6%94%b6%e9%9b%86%e5%99%a8)



CMS（Concurrent Mark Sweep），Mark Sweep 指的是标记 - 清除算法。

分为以下四个流程：

* 初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。
* 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
* 重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿。
* 并发清除：不需要停顿。

在整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，不需要进行停顿。

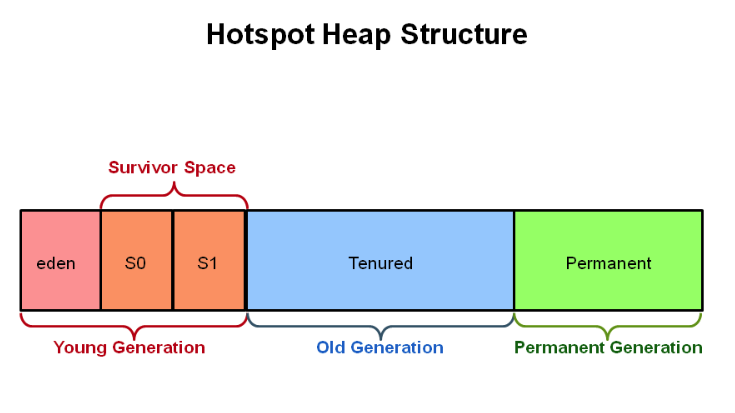
具有以下缺点：

* 吞吐量低：低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高。
* 无法处理浮动垃圾，可能出现 Concurrent Mode Failure。浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。
* 标记 - 清除算法导致的空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。

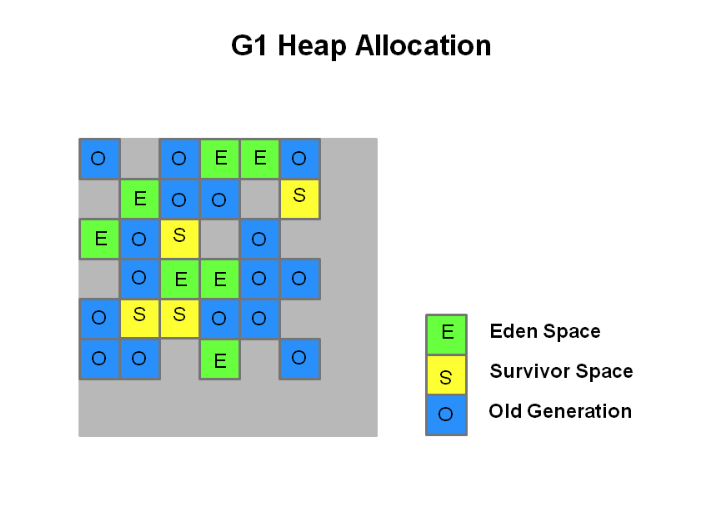
#### [7. G1 收集器](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_7-g1-%e6%94%b6%e9%9b%86%e5%99%a8)

G1（Garbage-First），它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能。HotSpot 开发团队赋予它的使命是未来可以替换掉 CMS 收集器。

堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，而 G1 可以直接对新生代和老年代一起回收。

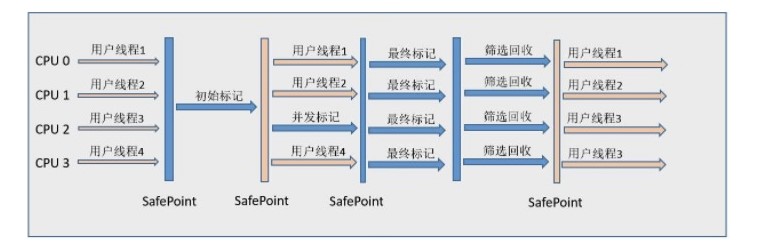


G1 把堆划分成多个大小相等的独立区域（Region），新生代和老年代不再物理隔离。



通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。这种划分方法带来了很大的灵活性，使得可预测的停顿时间模型成为可能。通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region。通过使用 Remembered Set，在做可达性分析的时候就可以避免全堆扫描。



如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：

* 初始标记
* 并发标记
* 最终标记：为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程的 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。这阶段需要停顿线程，但是可并行执行。
* 筛选回收：首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

具备如下特点：

* 空间整合：整体来看是基于“标记 - 整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。
* 可预测的停顿：能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在 GC 上的时间不得超过 N 毫秒。

### [、内存分配与回收策略](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=%e4%b8%89%e3%80%81%e5%86%85%e5%ad%98%e5%88%86%e9%85%8d%e4%b8%8e%e5%9b%9e%e6%94%b6%e7%ad%96%e7%95%a5)

#### 1.Minor GC Full GC

1.Minor GC：回收新生代，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。

Full GC：回收老年代和新生代，老年代对象其存活时间长，因此 Full GC 很少执行，执行速度会比 Minor GC 慢很多。

#### 2.[内存分配策略](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=%e5%86%85%e5%ad%98%e5%88%86%e9%85%8d%e7%ad%96%e7%95%a5)

[1. 对象优先在 Eden 分配](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_1-%e5%af%b9%e8%b1%a1%e4%bc%98%e5%85%88%e5%9c%a8-eden-%e5%88%86%e9%85%8d)

大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。

[2. 大对象直接进入老年代](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_2-%e5%a4%a7%e5%af%b9%e8%b1%a1%e7%9b%b4%e6%8e%a5%e8%bf%9b%e5%85%a5%e8%80%81%e5%b9%b4%e4%bb%a3)

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。

经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

-XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 和 Survivor 之间的大量内存复制。

[3. 长期存活的对象进入老年代](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_3-%e9%95%bf%e6%9c%9f%e5%ad%98%e6%b4%bb%e7%9a%84%e5%af%b9%e8%b1%a1%e8%bf%9b%e5%85%a5%e8%80%81%e5%b9%b4%e4%bb%a3)

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。

-XX:MaxTenuringThreshold 用来定义年龄的阈值。

[4. 动态对象年龄判定](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_4-%e5%8a%a8%e6%80%81%e5%af%b9%e8%b1%a1%e5%b9%b4%e9%be%84%e5%88%a4%e5%ae%9a)

虚拟机并不是永远要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

[5. 空间分配担保](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_5-%e7%a9%ba%e9%97%b4%e5%88%86%e9%85%8d%e6%8b%85%e4%bf%9d)

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的。

如果不成立的话虚拟机会查看 HandlePromotionFailure 的值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 的值不允许冒险，那么就要进行一次 Full GC。

#### 3.[Full GC 的触发条件](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=full-gc-%e7%9a%84%e8%a7%a6%e5%8f%91%e6%9d%a1%e4%bb%b6)

对于 Minor GC，其触发条件非常简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件：

[1. 调用 System.gc()](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_1-%e8%b0%83%e7%94%a8-systemgc)

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。

[2. 老年代空间不足](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_2-%e8%80%81%e5%b9%b4%e4%bb%a3%e7%a9%ba%e9%97%b4%e4%b8%8d%e8%b6%b3)

老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。

[3. 空间分配担保失败](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_3-%e7%a9%ba%e9%97%b4%e5%88%86%e9%85%8d%e6%8b%85%e4%bf%9d%e5%a4%b1%e8%b4%a5)

使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC。具体内容请参考上面的第 5 小节。

[4. JDK 1.7 及以前的永久代空间不足](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_4-jdk-17-%e5%8f%8a%e4%bb%a5%e5%89%8d%e7%9a%84%e6%b0%b8%e4%b9%85%e4%bb%a3%e7%a9%ba%e9%97%b4%e4%b8%8d%e8%b6%b3)

在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。

当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。

为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

[5. Concurrent Mode Failure](https://cyc2018.github.io/CS-Notes/#/notes/Java %E8%99%9A%E6%8B%9F%E6%9C%BA?id=_5-concurrent-mode-failure)

执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

### JVM调优

### 13.减少Full GC的方法

1. 方法区空间增大
2. 老年代空间增大
3. 新生代空间减小
4. 禁止使用System.gc()方法
5. 使用标记-整理算法，尽量让连续空间保持最大
6. 排查代码中的无用大对象(内存泄漏)

# JAVA集合框架

### ArrayList

#### 1.ArrayList是否会越界？

会，在多线程同时觉得不需要扩容，但是实际已经到了扩容的临界值的时候，就爆发了这个异常。

#### 2.Arraylist LinkedList 异同

**1. 是否保证线程安全：** ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

**2. 底层数据结构：** Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向循环链表；

**5. 内存空间占用：** LinkedList 比 ArrayList 需要更多的内存.(ArrayList的空间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。)

6.LinkedList 在插入和删除数据时效率更高，ArrayList 在查找某个 index 的数据时效率更高；

#### 3.在ArrayList和LinkedList尾部加元素,谁的效率高

当输入的数据一直是小于千万级别的时候，大部分是LinkedList效率高，而当数据量大于千万级别的时候，就会出现ArrayList的效率比较高了。为什么呢？

原来 LinkedList每次增加的时候，会new 一个Node对象来存新增加的元素，所以当数据量小的时候，这个时间并不明显，而ArrayList需要扩容，所以LinkedList的效率就会比较高，如果ArrayList出现不需要扩容的时候，那么ArrayList的效率应该是比LinkedList高的。

当数据量很大的时候，new对象的时间大于扩容的时间，那么就会出现ArrayList的效率比LinkedList高了

#### 4、ArrayList,Vector,LinkedList的存储性能和特性是什么？

ArrayList 和Vector他们底层的实现都是一样的，都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号查找元素，但是插入元素要涉及数组元素移动等内存操作，所以查找数据快而插入数据慢。  
  
Vector中的方法由于添加了synchronized修饰，因此Vector是线程安全的容器，但性能上较ArrayList差，因此已经是Java中的遗留容器。  
  
LinkedList使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比**，内存的利用率更高**，按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

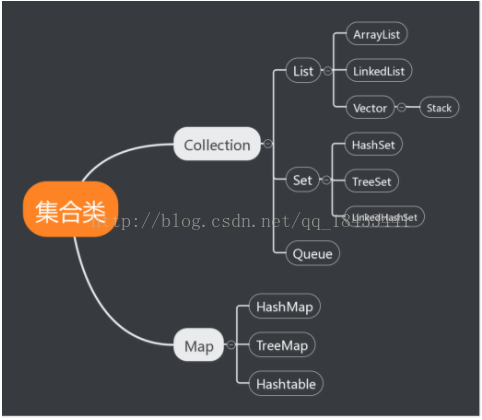
### 3.Iterator和ListIterator的区别是什么？

Iterator可用来遍历Set和List集合,但是ListIterator只能用来遍历List。Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。ListIterator实现了Iterator接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引等等。

### 4.为什么集合类没有实现Cloneable和Serializable接口？

克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。

### Java集合类框架的基本接口有哪些？



1.总共有两大接口：Collection和Map，一个是元素集合，一个是键值对集合。

2.其中List接口和Set接口继承了Collection接口，一个是有序元素集合，一个是无序元素集合

3.ArrayList类和LinkList类实现了List接口

4.HashSet（哈希表、散列表）实现了Set接口

5.TreeSet实现了SortedSet接口（图上没画出来）。无序，不可重复，但可按照元素大小自动排序，或者自定义排序方法

6.HashMap和HashTable实现了Map，其中HashTable是线程安全的，但是HashMap性能更好

7.TreeMap实现了SortedMap接口（图上没画出来）。无序，不可重复，但可按照元素大小自动排序或者自定义排序方法

### Hashmap

#### 说一下HashMap

HashMap是基于哈希表(散列表)，实现Map接口的双列集合，数据结构是“链表散列”，也就是数组+链表 ，key唯一的value可以重复，允许存储null 键null 值，元素无序。

1.HashMap可以接受null键值和值。

2.HashMap是非线程安全;

3.HashMap查询很快；

4.HashMap储存的是键值对

#### 2.HashMap的工作原理 HashMap的get()方法的工作原理

HashMap是基于hashing的原理，

我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。当我们给put()方法传递键和值时，我们先对key调用hashCode()方法，返回的hashCode用于找到bucket位置来储存Entry对象。”这里关键点在于指出，HashMap是在bucket中储存键对象和值对象，作为Map.Entry。这一点有助于理解获取对象的逻辑。如果你没有意识到这一点，或者错误的认为仅仅只在bucket中存储值的话，你将不会回答如何从HashMap中获取对象的逻辑。这个答案相当的正确，也显示出面试者确实知道hashing以及HashMap的工作原理。

#### 3.两个对象的hashcode相同会发生什么？

hashcode相同，所以它们的bucket位置相同，会发生碰撞。因为HashMap使用链表存储对象，这个Entry(包含有键值对的Map.Entry对象)会存储在链表中。”这个答案非常的合理，虽然有很多种处理碰撞的方法，这种方法是最简单的，也正是HashMap的处理方法。

在HashMap中采用链地址法来解决hash冲突，当发生碰撞是，将相同hash值的元素存储在链表的下一个节点处。除此之外还有开放地址法，再散列法。

#### 4.两个键的hashcode相同，你如何获取值对象？

当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，然后获取值对象。如果有两个值对象储存在同一个bucket，找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。

许多情况下，面试者会在这个环节中出错，因为他们混淆了hashCode()和equals()方法。因为在此之前hashCode()屡屡出现，而equals()方法仅仅在获取值对象的时候才出现。一些优秀的开发者会指出使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

#### 5.如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

HashMap中默认的负载因子为0.75，当HashMap中数组元素个数超过负载因子乘以数组总容量时会发生扩容，将数组扩展为原来的两倍，对原数组中的元素再进行rehashing运算，获得在新数组中的位置。扩容非常消耗性能。

#### HashMap的resize（rehash）

当 HashMap 中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的，所以为了提高查询的效率，就要对HashMap的数组进行扩容，在HashMap数组扩容之后，最消耗性能的点是：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是 resize（rehash）  
loadFactor指的是负载因子 HashMap能够承受住自身负载（大小或容量）的因子，loadFactor的默认值为 0.75认情况下，数组大小为 16，那么当HashMap中元素个数超过 12 的时候，就把数组的大小扩展为 32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知 HashMap 中元素的个数，那么预设元素的个数能够有效的提高HashMap 的性能  
负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是 O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费

#### 6.你了解重新调整HashMap大小存在什么问题吗？

你可能回答不上来，这时面试官会提醒你当多线程的情况下，可能产生条件竞争(race condition)。

当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。这个时候，你可以质问面试官，为什么这么奇怪，要在多线程的环境下使用HashMap呢？：）

#### 7.为什么String, Interger这样的wrapper类适合作为键？

HashMap推荐使用不可变变量作为键,String最为常用。因为String是final修饰，是不可变的，并且重写了equals()和hashCode()方法。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。

不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。

**我们可以使用自定义的对象作为键吗？**

可以，但是要确保这个对象重写了hashCode()和equals()方法，并且该对象插入Map后就不再改变，只要遵循了这些，它就已经可以作为键了。

#### 9.我们能否让HashMap同步？

通过Map m = Collections.synchronizeMap(hashMap)实现同步。在、synchronizeMap中给HashMap的所有操作增加synchronized竞争监视器锁来实现线程同步。我们可采用ConcurrentHashMap

#### 10.我们可以使用CocurrentHashMap来代替Hashtable吗？

可以，并且推荐使用CocurrentHashMap。HashTable采用synchronized实现同步，对其中所有方法添加监视器锁，问题在于，在一个线程进行put操作时，其他线程无法获得get或其他操作,效率不是很高.而CocurrentHashMap采用分段锁技术，比HashTable提供更强的线程安全。

#### 11.HashMap什么时候需要重写hashcode和equals方法？

当key值为对象时必须要求重写hashcode()和equals()方法。因为默认的hashcode()使用的是该值在内存中的地址作为该值的hash值，如果两个具有相同意义的对象进行比较时，由于其地址不同会导致计算出的hash值也不同。而重写equals()可以确保两个对象具有相同意义的属性。

#### 12.HashMap存储函数的实现put(K key, V value)

（1）计算key的hashCode()值

（2）然后对该哈希码值进行再哈希、

（3）然后把哈希值和(数组长度-1)进行按位与操作，得到存储的数组下标

（4）如果该位置处设有链表节点，那么就直接把包含<key,value>的节点放入该位置。如果该位置有结点，就对链表进行遍历，看是否有hash,key和要放入的节点相同的节点，如果有的话，就替换该节点的value值，如果没有相同的话，就创建节点放入值，并把该节点插入到链表表头(头插法)。

调用put 方法时候，如果key存在key不会覆盖，新的value会替代旧的value，返回旧的value;如果key不存在，该方法返回null

调用put时候，根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置 也就是下标，如果数组在该位置上已有其他元素，那么在这个位置上的元素将以链表的形式存放，新加入的在立案表头，先加入的在尾部

public V put(K key, V value) {

//其允许存放null的key和null的value，当其key为null时，调用putForNullKey方法，放入到table[0]的这个位置

if (key == null)

return putForNullKey(value);

//通过调用hash方法对key进行哈希，得到哈希之后的数值。该方法实现可以通过看源码，其目的是为了尽可能的让键值对可以分不到不同的桶中

int hash = hash(key);

//根据上一步骤中求出的hash得到在数组中是索引i

int i = indexFor(hash, table.length);

//如果i处的Entry不为null，则通过其next指针不断遍历e元素的下一个元素。

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;}

#### 13.为何数组的长度是2的n次方呢?

1.这个方法非常巧妙，它通过h & (table.length-1)来得到该对象的保存位，而HashMap底层数组的长度总是2的n次方，2^n -1得到的二进制数的每个位上的值都为1,那么与全部为1的一一个数进行与操作，速度会大大提升。

2.当length总是2的n次方时，h& (length-1)运 算等价王对length取模，也就是h%length,但是&比%县有更高的效率。

3.当数组长度为2的n次幂的时候，不同的key算得的index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰擁的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

#### 14.为什么阈值变为8变成红黑树？

当hashCode离散性很好的时候，树型bin用到的概率非常小，因为数据均匀分布在每个bin中，几乎不会有bin中链表长度会达到阈值。但是在随机hashCode下，离散性可能会变差，然而JDK又不能阻止用户实现这种不好的hash算法，因此就可能导致不均匀的数据分布。不过理想情况下随机hashCode算法下所有bin中节点的分布频率会遵循泊松分布，我们可以看到，一个bin中链表长度达到8个元素的概率为0.00000006，几乎是不可能事件。所以，之所以选择8，不是拍拍屁股决定的，而是根据概率统计决定的。

#### 15. HashMap中的hash碰撞问题，最坏的情况下，所有的key都被分配到了同一个桶，这时map的put和get时间复杂度都是O(n)。

#### 16.hashmap遍历方法，遍历过程中可以删除吗

public static void main(String[] args) {

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

map.put("a", 1);

map.put("b", 2);

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

while (entryIterator.hasNext()) {

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

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

}

运行结果没有显示，表明HashMap中的元素被正确删除了。

2. Iterator<String> iterator = map.keySet().iterator();

while (iterator.hasNext()) {

String key = iterator.next();

System.out.println("key=" + key + " value=" + map.get(key));

}

3. // 等同于第一种方式 foreach模式，适用于不需要修改HashMap内元素的遍历，只需要获取元素的键/值的情况。

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

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

}

}

运行上面的代码，Java抛出了 java.util.ConcurrentModificationException 的异常。　可以推测，由于我们在遍历HashMap的元素过程中删除了当前所在元素，下一个待访问的元素的指针也由此丢失了。

#### 17.hashmap为什么要用红黑树

红黑树相比avl树，在检索的时候效率其实差不多，都是通过平衡来二分查找。但对于插入删除等操作效率提高很多。红黑树不像avl树一样追求绝对的平衡，他允许局部很少的不完全平衡，这样对于效率影响不大，但省去了很多没有必要的调平衡操作，avl树调平衡有时候代价较大，所以效率不如红黑树，在现在很多地方都是底层都是红黑树的天下啦~

### 7.、List、Map、Set三个接口存取元素时，各有什么特点？

List与Set都是单列元素的集合，它们有一个功共同的父接口Collection。

(1)Set里面不允许有重复的元素，

存元素：add方法有一个boolean的返回值，当 中没有某个元素，此时add方法可成功加入该元素时，则返回true；当集合含有与某个元素equals相等的 元素时，此时add方法无法加入该元素，返回结果为false。

取元素：没法取第几个，只能以Iterator接口取得所有的元素，再逐一遍历各个元素。

(2)List表示有先后顺序的集合，

存元素：多次调用add(Object)方法时，每次加入的对象按先来后到的顺序排序，也可以插队，即调用add(int index,Object)方法，就可以指定当前对象在集合中的存放位置。

取元素：方法1：Iterator接口取得所有，逐一遍历各个元素

         方法2：调用get(index i)来明确说明取第几个。

(3)Map是双列的集合，存放用put方法:put(obj key,obj value)，每次存储时，要存储一对key/value，不能存储重复的key，这个重复的规则也是按equals比较相等。

取元素：用get(Object key)方法根据key获得相应的value。

         也可以获得所有的key的集合，还可以获得所有的value的集合，

         还可以获得key和value组合成的Map.Entry对象的集合。

List以特定次序来持有元素，可有重复元素。Set 无法拥有重复元素,内部排序。Map 保存key-value值，value可多值。

# JAVA基础

### 1.数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用Array而不是ArrayList？

(1)存储内容比较：Array 数组可以包含基本类型和对象类型，ArrayList 却只能包含对象类型。Array 数组在存放的时候一定是同种类型的元素。ArrayList 就不一定了，因为ArrayList可以存储Object。

(2)空间大小比较：Array 数组的空间大小是固定的,所以需要事前确定合适的空间大小。ArrayList 的空间是动态增长的,如果空间不够，它会创建一个空间比原空间多大约0.5倍的新数组，然后将所有元素复制到新数组中，接着抛弃旧数组。而且，每次添加新的元素的时候都会检查内部数组的空间是否足够。

Arraylist扩充机制：newCapacity=oldCapacity+(oldCapacity>>1)但由于源码里（不再分析，这里简要略过）传过来的minCapcatiy的值是size+1，能够实现grow方法调用就肯定是(size+1)>elementData.length的情况，所以size就是初始最大容量或上一次扩容后达到的最大容量，所以才会进行扩容。因此，扩容后的大小应该是原来的1.5倍+1

(3)方法上的比较：

ArrayList 方法上比 Array 更多样化，比如添加全部 addAll()、删除全部 removeAll()、返回迭代器 iterator() 等。对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

适用场景：

如果想要保存一些在整个程序运行期间都会存在而且不变的数据，我们可以将它们放进一个全局数组里， 但是如果我们单纯只是想要以数组的形式保存数据，而不对数据进行增加等操作，只是方便我们进行查找的话，那么，我们就选择 ArrayList。

如果我们需要对元素进行频繁的移动或删除，或者是处理的是超大量的数据，那么，使用 ArrayList 就真的不是一个好的选择，因为它的效率很低，使用数组进行这样的动作就很麻烦，那么，我们可以考虑选择 LinkedList。

### 2、String、StringBuffer、StringBuilder的区别

a.执行速度:StringBuilder > StringBuffer > String

b.线程安全:String StringBuffer线程安全. StringBuilder线程不安全

c.String适用与少量字符串操作

StringBuilder适用单线程下在字符缓冲区下进行大量操作的情况

StringBuffer使用多线程下在字符缓冲区进行大量操作的情况

### 3、Object若不重写hashCode()的话，hashCode()如何计算出来的？

Object的hashcode 方法是本地方法，也就是用 c 语言或 c++ 实现的，该方法直接返回对象的JVM 内存地址。

### 4、==比较的是什么？

1、对象引用类型,比较的是对象的内存地址。

public class ArrayTest {

public static void main(String[] args){

String a = new String("aw");

String b = new String("aw");

System.out.println(a==b);//false

}

}

显然，尽管 a 与 b 对象的值相同，但是在内存中的地址是不同的，即两个对象是不一样的。再看一个例子：

public static void main(String[] args){

String c= "aa";

String d= "aa";

System.out.println(c==d);//true

}

这是因为常量池的存在。而运行时常量池其实是属于方法区的一部分。通俗的说，c 和 d 其实都是都是指向 “aa”这个常量。

但是这里要注意，对于Integer对象来说，其能存储的范围为（-128~127），超过范围则存储到堆内存中。

2、基本类型数据，比较值。

### 5.若对一个类不重写，它的equals()方法是如何比较的？

如果没有对equals方法进行重写，会直接使用超类 Object 的 equals 方法，里面是用“==”比较，比较的是引用类型的变量所指向的对象的地址；诸如String、Date等类对equals方法进行了重写的话，比较的是所指向的对象的内容。

### 8、Java支持的数据类型有哪些？什么是自动拆装箱？

基本数据类型：

整数值型：byte,short,int,long

字符型：char

浮点类型：float,double

布尔型：boolean

整数默认int型，小数默认是double型。Float和long类型的必须加后缀。

首先知道String是引用类型不是基本类型，引用类型声明的变量是指该变量在内存中实际存储的是一个引用地址，实体在堆中。引用类型包括类、接口、数组等。String类还是final修饰的。

而包装类就属于引用类型，自动装箱和拆箱就是基本类型和引用类型之间的转换，至于为什么要转换，因为基本类型转换为引用类型后，就可以new对象，从而调用包装类中封装好的方法进行基本类型之间的转换或者toString（当然用类名直接调用也可以，便于一眼看出该方法是静态的），还有就是如果集合中想存放基本类型，泛型的限定类型只能是对应的包装类型。

### 9.什么是值传递和引用传递？

值传递:在方法的调用过程中，实参把它的实际值传递给形参，此传递过程就是将实参的值复制一份传递到函数中，这样如果在函数中对该值（形参的值）进行了操作将不会影响实参的值。因为是直接复制，所以这种方式在传递大量数据时，运行效率会特别低下。

引用传递:引用传递弥补了值传递的不足，如果传递的数据量很大，直接复过去的话，会占用大量的内存空间，而引用传递就是将对象的地址值传递过去，函数接收的是原始值的首地址值。在方法的执行过程中，形参和实参的内容相同，指向同一块内存地址，也就是说操作的其实都是源数据，所以方法的执行将会影响到实际对象。

public class Example {

String str = new String("hello");

char[] ch = {'a', 'b'};

public static void main(String[] args) {

Example ex = new Example();

ex.change(ex.str, ex.ch);

System.out.println(ex.str + " and");

System.out.println(ex.ch);

}

public void change(String str, char[] ch) {

str = "ok";

ch[0] = 'c';

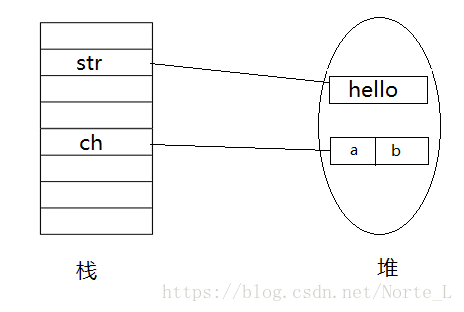
} }

输出是：

hello and

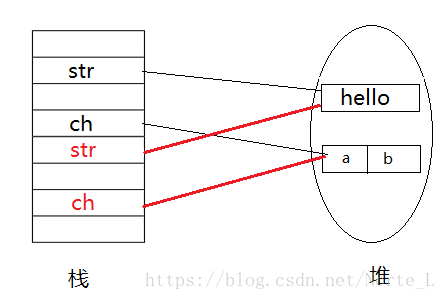
cb

过程分析：

1. 为对象分配空间
2. 

2、执行change()方法

执行前实参（黑色）和形参（红色）的指向如下：



因为String是不可变类且为值传递，而ch[]是引用传递，所以方法中的str = "ok",相当于重新创建一个对象并没有改变实参str的值，数组是引用传递，直接改变，所以执行完方法后，指向关系如下：

· 基本数据类型传值，对形参的修改不会影响实参；

· 引用类型传引用，形参和实参指向同一个内存地址（同一个对象），所以对参数的修改会影响到实际的对象。

· String, Integer, Double等immutable的类型特殊处理，可以理解为传值，最后的操作不会修改实参对象。

### 11、String是最基本的数据类型吗?

String不是基本的数据类型，是final修饰的java类。

### 12.int 和 Integer 有什么区别

(1)Integer是int的包装类，int则是java的一种基本数据类型   
(2)Integer变量必须实例化后才能使用，而int变量不需要   
(3)Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向此对象；而int则是直接存储数据值   
(4)Integer的默认值是null，int的默认值是0

**延伸：关于Integer和int的比较**  
(1)由于Integer变量实际上是对一个Integer对象的引用，所以两个通过new生成的Integer变量永远是不相等的（因为new生成的是两个对象，其内存地址不同）。

Integer i = new Integer(100);  
Integer j = new Integer(100);  
System.out.print(i == j); //false

(2)Integer变量和int变量比较时，只要两个变量的值是相等的，则结果为true（因为包装类Integer和基本数据类型int比较时，java会自动拆包装为int，然后进行比较，实际上就变为两个int变量的比较）

Integer i = new Integer(100);  
int j = 100；  
System.out.print(i == j); //true

(3)非new生成的Integer变量和new Integer()生成的变量比较时，结果为false。（因为非new生成的Integer变量指向的是java常量池中的对象，而new Integer()生成的变量指向堆中新建的对象，两者在内存中的地址不同）

Integer i = new Integer(100);  
Integer j = 100;  
System.out.print(i == j); //false

(4)两个非new生成的Integer对象

Integer i = 100;  
Integer j = 100;  
System.out.print(i == j); //true (-128-127)

Integer i = 128;  
Integer j = 128;  
System.out.print(i == j); //false

java在编译Integer i = 100 ;时，会翻译成为Integer i = Integer.valueOf(100)；，而java API中对Integer类型的valueOf的定义如下：

public static Integer valueOf(int i){  
 assert IntegerCache.high >= 127;  
 if (i >= IntegerCache.low && i <= IntegerCache.high){  
 return IntegerCache.cache[i + (-IntegerCache.low)];  
 }  
 return new Integer(i);  
}

java对于-128到127之间的数，会进行缓存，Integer i = 127时，会将127进行缓存，下次再写Integer j = 127时，就会直接从缓存中取，就不会new了（java常量池？）

### 13.String 和StringBuffer和StringBuilder的区别

(1)String含义为引用数据类型,是字符串常量.是不可变的对象,(显然线程安全)在每次对string类型进行改变的时候其实都等同与生成了一个新的String对象.然后指针指向新的String对象,所以经常改变内容的字符串最好不使用String,因为每次生成对象都会对系统性能产生影响,特别当内存中无引用对象多了之后.JVM的垃圾回收(GC)就会开始工作,对系统的性能会产生影响

(2)StringBuffer:线程安全的可变字符序列,对StringBuffer对象本身进行操作,而不是生成新的对象.所所以在改变对象引用条件下,一般推荐使用StringBuffer.同时主要是使用append和insert方法,

(3)StringBuilder:线程不安全的可变字符序列.提供一个与StringBuffer兼容的API,但不同步.设计作为StringBuffer的一个简易替换,用在字符缓冲区被单个线程使用的时候.效率比StringBuffer更快

### 15、&和&&的区别？

Java中&&和&都是表示与的逻辑运算符，都表示逻辑运输符and，当两边的表达式都为true的时候，整个运算结果才为true，否则为false。

&&的短路功能，当第一个表达式的值为false的时候，则不再计算第二个表达式；&则两个表达式都执行。

&可以用作位运算符，当&两边的表达式不是Boolean类型的时候，&表示按位操作。

17.

# 计算机基础

## 计算机网络

### TCP UDP

#### 1.TCP协议在哪一层？IP协议在那一层？HTTP在哪一层？

tcp在传输层，IP在网络层，http在应用层

#### 2.TCP/UDP区别及应用场景

1、TCP面向连接（三次握手）;UDP是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达。通过**确认和重传机制**来保证数据传输的可靠性;UDP尽最大努力交付，即不保证可靠交付

TCP还**提供了拥塞控制、滑动窗口等机制**来保证传输的质量，而UDP都没有

Tcp通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。

4.每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

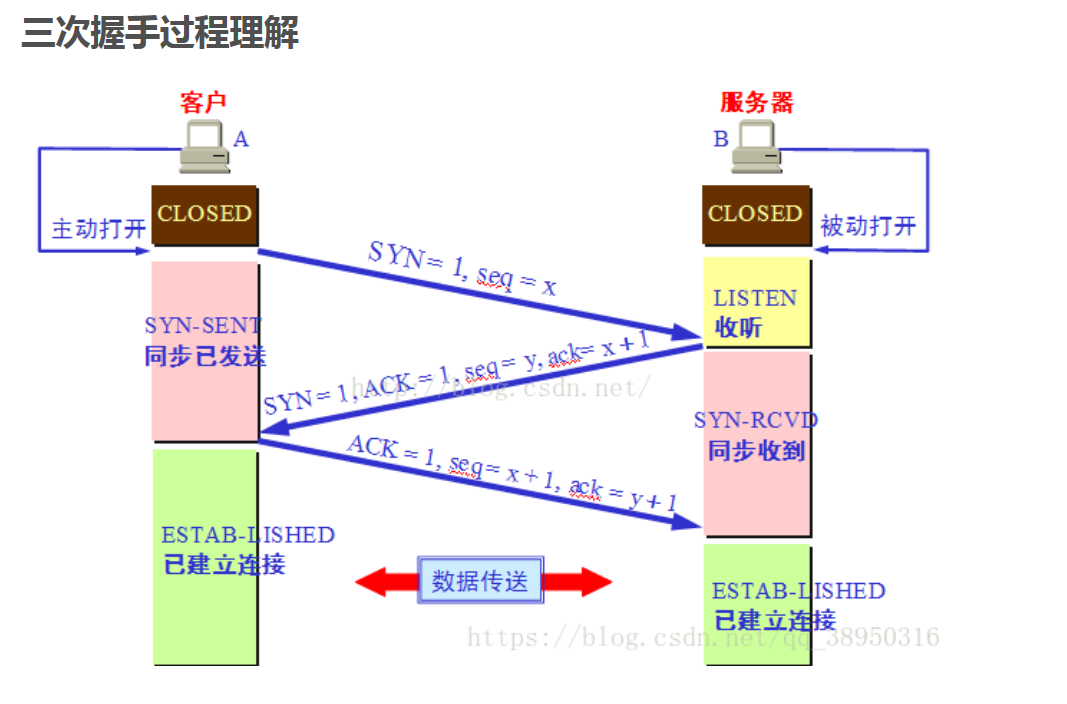
5、TCP对系统资源要求较多，UDP对系统资源要求较少。

6.TCP是基于**字节流**的，将数据看做无结构的字节流进行传输，当应用程序交给TCP的数据长度太长，超过MSS时，TCP就会对数据进行分段，因此TCP的数据是无边界的；而UDP是**面向报文**的，无论应用程序交给UDP层多长的报文，UDP都不会对数据报进行任何拆分等处理，因此UDP保留了应用层数据的边界

TCP：文件传输、重要状态的更新。通信数据完整性需比通信实时性重要，则应该选用 TCP

UDP：视频传输、实时通信。对实时性要求较高

#### 3.TCP 3次握手

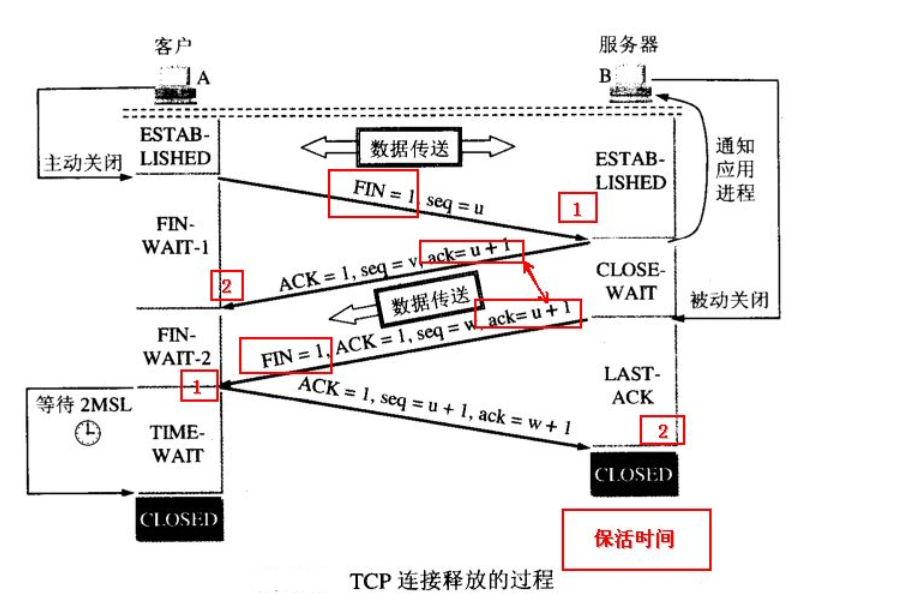


第一次握手：客户端主动打开，发送连接请求报文段，将SYN标识位置为1，序列号置为x（TCP规定SYN=1时不能携带数据，x为随机产生的一个值），然后进入SYN\_SEND状态,等待服务器确认.

第二次握手：服务器收到SYN报文段进行确认，将SYN标识位置为1，ACK置为1，序列号置为y，确认号置为x+1，然后进入SYN\_RECV状态，这个状态被称为半连接状态.

第三次握手：客户端再进行一次确认，将ACK置为1（此时不用SYN），序列号置为x+1，确认号置为y+1发向服务器，最后客户端与服务器都进入ESTABLISHED状态

#### 4次挥手



1. 客户端发送一个报文给服务端（没有数据），其中FIN设置为1，序列号置为u，客户端进入FIN\_WAIT\_1状态
2. 服务端收到来自客户端的请求，发送一个ACK给客户端，确认号置为u+1，同时发送序列号为v，服务端年进入CLOSE\_WAIT状态
3. 服务端发送一个FIN给客户端，ACK置为1，Sequence置为w，Acknowledge置为u+1，用来关闭服务端到客户端的数据传送，服务端进入LAST\_ACK状态
4. 客户端收到FIN后，进入TIME\_WAIT状态，接着发送一个ACK给服务端，Acknowledge置为w+1，Sequence Number置为u+1，最后客户端和服务端都进入CLOSED状态

#### 为什么要进行3次挥手

一句话，主要防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了，由于TCP的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

#### 为什么建立连接需要三次握手，而断开连接需要四次握手

**因为每个方向都需要一个FIN和ACK，当一端发送了FIN包之后，处于半关闭状态，此时仍然可以接收数据包。**   
　　在建立连接时，服务器**可以把SYN和ACK放在一个包中**发送。   
　　但是在断开连接时，如果一端收到FIN包，但此时仍有数据未发送完，此时就需要先向对端回复FIN包的ACK。等到将剩下的数据都发送完之后，再向对端发送FIN，断开这个方向的连接。   
　　因此很多时候**FIN和ACK需要在两个数据包中**发送，因此需要四次握手

#### TIME\_WAIT状态持续时间及原因

　　持续时间为2MSL，一个数据包在网络中的最长生存时间为MSL。   
　　**假设最后客户端回复的ACK丢失，服务器端会在超时时间到来时，重传最后一个FIN包。**   
　　ACK和FIN在网络中的最长生存时间就为2MSL，这样就可以可靠的断开TCP的双向连接。

#### 超时重传和快速重传

超时重传：当超时时间到达时，发送方还未收到对端的ACK确认，就重传该数据包

快速重传：当后面的序号先到达，如接收方接收到了1、 3、 4，而2没有收到，就会立即向发送方重复发送三次ACK=2的确认请求重传。如果发送方连续收到3个相同序号的ACK，就重传该数据包。而不用等待超时 。

#### TCP拥塞控制

TCP通过维护一个拥塞窗口来进行拥塞控制，拥塞控制的原则是，只要网络中没有出现拥塞，拥塞窗口的值就可以再增大一些，以便把更多的数据包发送出去，但只要网络出现拥塞，拥塞窗口的值就应该减小一些，以减少注入到网络中的数据包数。

1. 拥塞控制与流量控制的区别   
   拥塞控制是防止过多的数据注入到网络中，可以使网络中的路由器或链路不致过载，是一个全局性的过程。   
   流量控制是点对点通信量的控制，是一个端到端的问题，主要就是抑制发送端发送数据的速率，以便接收端来得及接收。
2. 拥塞的标志

1.重传计时器超时   
2.接收到三个重复确认

（3）拥塞控制机制：

**慢开始**

1.慢开始不是指cwnd的增长速度慢（指数增长），而是指TCP开始发送设置cwnd=1。

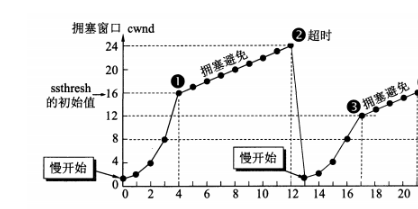
2.思路：不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。这里用报文段的个数的拥塞窗口大小举例说明慢开始算法，实时拥塞窗口大小是以字节为单位的。

**慢开始门限（ssthresh）**

为了防止cwnd增长过大引起网络拥塞，设置一个慢开始门限（ssthresh状态变量）   
当cnwd＜ssthresh，使用慢开始算法   
当cnwd=ssthresh，既可使用慢开始算法，也可以使用拥塞避免算法   
当cnwd＞ssthresh，使用拥塞避免算法

拥塞避免（按线性规律增长）

1. 拥塞避免并非完全能够避免拥塞，是说在拥塞避免阶段将拥塞窗口控制为按线性规律增长，使网络比较不容易出现拥塞。   
   2.思路：让拥塞窗口cwnd缓慢地增大，即每经过一个往返时间RTT就把发送方的拥塞控制窗口加一。

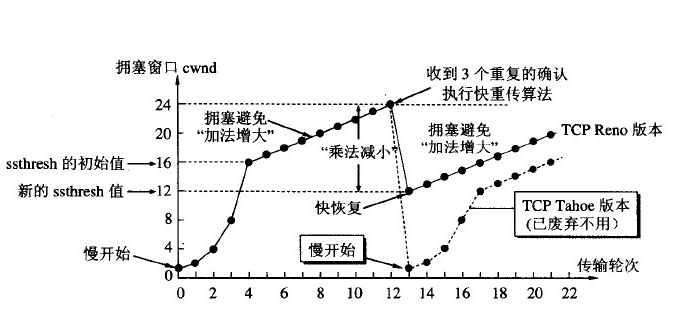
**无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。**   


乘法减小：无论是慢开始阶段还是拥塞避免，只要出现了网络拥塞（超时），就把慢开始门限值ssthresh减半   
加法增大：执行拥塞避免算法后，拥塞窗口线性缓慢增大，防止网络过早出现拥塞

增加快重传与快恢复

**快重传**

1.快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，**发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。**



2.由于不需要等待设置的重传计时器到期，能尽早重传未被确认的报文段，能提高整个网络的吞吐量。

快恢复（与快重传配合使用）

1.采用快恢复算法时，慢开始只在TCP连接建立时和网络出现超时时才使用。

2.当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把ssthresh门限减半。但是接下去并不执行慢开始算法。

3.考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。

注意

发送方窗口的上限值=Min（接受窗口rwnd，拥塞窗口cwnd）

rwnd＞cwnd 接收方的接收能力限制发送方窗口的最大值

rwnd＜cwnd 网络的拥塞限制发送方窗口的最大值

#### 10.TCP如何保证可靠性：

校验和

序列号

确认应答

超时重传：发送方没有介绍到响应的ACK报文原因可能有两点：

1. 数据在传输过程中由于网络原因等直接全体丢包，接收方根本没有接收到。

2. 接收方接收到了响应的数据，但是发送的ACK报文响应却由于网络原因丢包了。

发送方在发送完数据后等待一个时间，时间到达没有接收到ACK报文，那么对刚才发送的数据进行重新发送。如果是刚才第一个原因，接收方收到二次重发的数据后，便进行ACK应答。如果是第二个原因，接收方发现接收的数据已存在（判断存在的根据就是序列号，所以上面说序列号还有去除重复数据的作用），那么直接丢弃，仍旧发送ACK应答。

那么发送方发送完毕后等待的时间是多少呢？如果这个等待的时间过长，那么会影响TCP传输的整体效率，如果等待时间过短，又会导致频繁的发送重复的包。如何权衡？

由于TCP传输时保证能够在任何环境下都有一个高性能的通信，因此这个最大超时时间（也就是等待的时间）是动态计算的。

连接管理：三次握手 四次挥手

流量控制：发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。

TCP的流量控制是利用**滑动窗口机制**实现的，接收方在返回的ACK中会包含自己的接收窗口的大小，以控制发送方的数据发送。

但是当某个ACK报文丢失了，就会出现A等待B确认，并且B等待A发送数据的死锁状态。为了解决这种问题，TCP引入了持续计时器（Persistence timer），当A收到rwnd=0时，就启用该计时器，时间到了则发送一个1字节的探测报文，询问B是很忙还是上个ACK丢失了，然后B回应自身的接收窗口大小，返回仍为0（A重设持续计时器继续等待）或者会重发rwnd=x。

拥塞控制

### 2.HTTP HTTPs

#### 1.HTTPs请求过程

1.客户端发送请求https连接。

2、服务器返回加密公钥，通常是SSL证书。

3、客户端从这个SSL证书解析出公钥，并随机生成一个key，通过公钥加密这个key发送给服务器（这一步是安全的因为只有服务器才有私钥能读出这个key）。

4、服务器通过私钥解密出key。

5、客户端使用这个key来加密需要传输的数据。

6、服务器使用key来解析数据。

简单的来说SSL加密的方式是使用一个密钥来加密另一个密钥（key），在使用被加密的密钥来加密数据。这样的做法固然保证了安全性，但每次连接时都需要使用密钥加密，导致请求会需要额外的开销，同时服务器第一次返回的公钥的可靠性需要第三方来保证，通常是购买SSL证书。这也会造成额外的经济开销。

#### 2.Http和Https的区别

1.Http协议运行在TCP之上，明文传输，客户端与服务器端都无法验证对方的身份；

Https是身披SSL(Secure Socket Layer)外壳的Http，运行于SSL上，SSL运行于TCP之上，是添加了加密和认证机制的HTTP。

2.端口不同：Http与Https使用不同的连接方式，http是80，https是443；

3.资源消耗：和HTTP通信相比，Https通信会由于加减密处理消耗更多的CPU和内存资源；

4.开销：Https通信需要证书，而证书一般需要向认证机构购买；   
　 Https的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机制。

#### 3.Http状态码

200 - 服务器成功返回网页

400(错误请求)服务器不理解请求的语法。

403 forbidden，表示对请求资源的访问被服务器拒绝（服务器已经理解请求，但是拒绝执行它）

　404 - 请求的网页不存在

   500(服务器内部错误)服务器遇到错误，无法完成请求。

　503 - 服务器超时

#### 4.Https作用

1．地址栏绿色小锁（主要体现在浏览器地址栏，让网友客户更加信任，体现品牌形象，让客户更安全的访问）

2．防劫持（移动数据劫持、HTTP协议劫持，流量劫持等最为频繁，对于做百度推广或移动广告的，这一点必须要关注，不然广告都是给别人做的！全站Https是根治运营商、中间人流量劫持的解决方案，不仅可以杜绝网页中被插入的小广告，更可以保护用户隐私安全。）

3.认证网站服务器的真实身份，网站安装由第三方权威机构颁发的网站ssl证书，也就意味着，网站所在的服务器真实身份已经认证过了，网站的信息也是真实的，用户不用担心网站是不是安全的网站。访客发访问网站的时候，可以看到地址栏那有一个绿色是锁的标识，点击就可查看网站的身份信息，安装ssl证书的网站可以有效的避免网站钓鱼等不安全因素造成经济损失。

4.保护网站的数据安全。通过对数据传输层加密，确保数据在传输的过程中，数据不被窃取，保证数据的完整性，安全。防止用户信息，财务信息等一些重要的数据被篡改，以及被窃取。也就是说安装ssl证书的网站数据安全性比没有安装的网站安全性更高。

#### 5.get post 区别？

1.GET请求在URL中传送的参数是有长度限制的，而POST没有。

2.GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

3.GET参数通过URL传递，POST放在Request body中。

4.GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

5.GET请求只能进行url编码，而POST支持多种编码方式。

6.GET请求会被浏览器主动cache，而POST不会，除非手动设置。

7.GET产生的URL地址可以被Bookmark，而POST不可以。

8.GET在浏览器回退时是无害的，而POST会再次提交请求。

#### 6.网页从输入url到完整呈现出来都经历了哪几个过程

1输入网址；

2发送到DNS服务器，并获取域名对应的web服务器对应的ip地址；

3与web服务器建立TCP连接；

4浏览器向web服务器发送http请求；

5.web服务器响应请求，并返回指定url的数据（或错误信息，或重定向的新的url地址）；

6浏览器下载web服务器返回的数据及解析html源文件；

7生成DOM树，解析css和js，渲染页面，直至显示完成；

### 3 cookie session 区别及联系？

· **存放位置不同：**session 保存在服务器，cookie 保存在客户端；

· **存放的形式不同：**session 是以对象的形式保存，cookie 以字符串的形式保存；

· **用途不同：**session 适合做客户的身份验证，cookie 适合保存用户的个人设置，爱好等；

· **路径：**session 不能区分路径，同一用户在访问一个网站期间，所有的 session 在任何一个地方都可以访问到；cookie 中如果设置了参数路径，那么同一个网站下的 cookie 互相访问不到；

· **安全性：**cookie 不是很安全，别人可以分析存放在本地的 cookie 并进行 cookie 欺骗，session 较 cookie 更安全一些；

· **大小及数量限制：**单个 cookie 在客户端的限制是 3K，就是说一个站点在客户端存放的 COOKIE 不能 3K。不同浏览器所含 cookie 的最大个数不同，一般 30 到 50 个；一般认为 session 没有大小限制。

**联系：session 需要借助 cookie 才能正常工作**，如果客户端完全禁止 cookie，session 将失效，因为 session 是由应用服务器维持的一个服务端的存储空间，用户在连接服务器时，会由服务器生成唯一的 sesssion id，用该 session id 为标识来存取服务端的 session 空间。而 session id 存储在 cookie 中，用户提交页面时会将这个 session id 提交到服务端，来存取 session 数据.这一过程是不用开发人员干预的，所以一旦客户端禁用 cookie，那么 session 也会失效。

session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，如果主要考虑到减轻服务器性能方面，应当使用 COOKIE；

Sessionid 是服务器和客户端链接时候随机分配的。

## 数据结构

面试中链表和树比较高频

### 队列

队列与广度优先遍历算法紧密联系

### 堆

堆就是用**数组实现**的二叉树，所有它没有使用父指针或者子指针。堆根据“堆属性”来排序，“堆属性”决定了树中节点的位置。

堆的常用方法：

* 构建优先队列
* 支持堆排序
* 快速找出一个集合中的最小值（或者最大值）

堆分为两种：**最大堆和最小堆**，两者的差别在于节点的排序方式。

在最大堆中，父节点的值比每一个子节点的值都要大。在最小堆中，父节点的值比每一个子节点的值都要小。这就是所谓的“堆属性”，并且这个属性对堆中的每一个节点都成立。

根据这一属性，那么最大堆总是将其中的最大值存放在树的根节点。而对于最小堆，根节点中的元素总是树中的最小值。堆属性非常的有用，因为堆常常被当做优先队列使用，因为可以快速的访问到“最重要”的元素。

**注意：堆的根节点中存放的是最大或者最小元素，但是其他节点的排序顺序是未知的。例如，在一个最大堆中，最大的那一个元素总是位于 index 0 的位置，但是最小的元素则未必是最后一个元素。唯一能够保证的是最小的元素是一个叶节点，但是不确定是哪一个。**

#### 堆和普通树的区别

**堆并不能取代二叉搜索树（二叉查找树，二叉排序树Binary Search Tree）**，它们之间有相似之处也有一些不同。我们来看一下两者的主要差别：

**节点的顺序。**在二叉搜索树中，左子节点必须比父节点小，右子节点必须必比父节点大。但是在堆中并非如此。在最大堆中两个子节点都必须比父节点小，而在最小堆中，它们都必须比父节点大。

**内存占用。**普通树占用的内存空间比它们存储的数据要多。你必须为节点对象以及左/右子节点指针分配额外的内存。**堆仅仅使用一个数组，且不使用指针。**

**平衡。**二叉搜索树必须是“平衡”的情况下，其大部分操作的复杂度才能达到**O(log n)**。你可以按任意顺序位置插入/删除数据，或者使用 AVL 树或者红黑树，但是在堆中实际上不需要整棵树都是有序的。我们只需要满足对属性即可，所以在堆中平衡不是问题。因为堆中数据的组织方式可以保证**O(log n)** 的性能。

**搜索。在二叉树中搜索会很快，但是在堆中搜索会很慢**。**在堆中搜索不是第一优先级，因为使用堆的目的是将最大（或者最小）的节点放在最前面，从而快速的进行相关插入、删除操作。**

#### 来自数组的树

用数组来实现树相关的数据结构也许看起来有点古怪，但是它在时间和空间上都是很高效的。

我们准备将上面的例子中的树这样存储：[ 10, 7, 2, 5, 1 ]

就这多！我们除了一个简单的数组以外，不需要任何额外的空间。

如果我们不允许使用指针，那么我们怎么知道哪一个节点是父节点，哪一个节点是它的子节点呢？问得好！**节点在数组中的位置index 和它的父节点以及子节点的索引之间有一个映射关系。**

如果 i 是节点的索引，那么下面的公式就给出了它的父节点和子节点在数组中的位置：

parent(i) = floor((i - 1)/2)

left(i) = 2i + 1

right(i) = 2i + 2

注意 right(i) 就是简单的 left(i) + 1。左右节点总是处于相邻的位置。

我们将写公式放到前面的例子中验证一下。

| **Node** | **Array index (i)** | **Parent index** | **Left child** | **Right child** |
| --- | --- | --- | --- | --- |
| 10 | 0 | -1 | 1 | 2 |
| 7 | 1 | 0 | 3 | 4 |
| 2 | 2 | 0 | 5 | 6 |
| 5 | 3 | 1 | 7 | 8 |
| 1 | 4 | 1 | 9 | 10 |

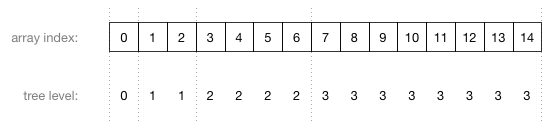
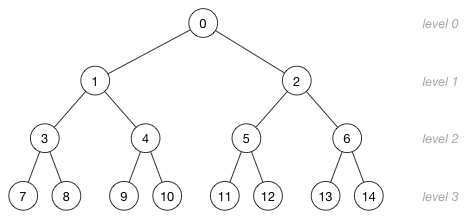
**注意：根节点(10)没有父节点，因为 -1 不是一个有效的数组索引。同样，节点 (2)，(5)和(1) 没有子节点，因为这些索引已经超过了数组的大小，所以我们在使用这些索引值的时候需要保证是有效的索引值。**

复习一下，在**最大堆**中，父节点的值总是要大于（或者等于）其子节点的值。这意味下面的公式对数组中任意一个索引 i都成立：

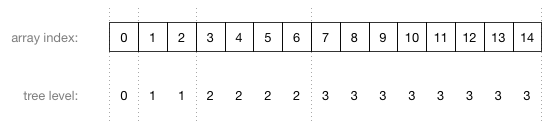
**array[parent(i)] >= array[i]**

如你所见，这些公式允许我们不使用指针就可以找到任何一个节点的父节点或者子节点。事情比简单的去掉指针要复杂，但这就是交易：我们**节约了空间，但是要进行更多计算。幸好这些计算很快并且只需要O(1)的时间。**

理解数组索引和节点位置之间的关系非常重要。这里有一个更大的堆，它有15个节点被分成了4层：

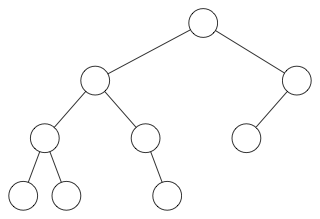


图片中的数字不是节点的值，而是存储这个节点的数组索引！这里是数组索引和树的层级之间的关系：

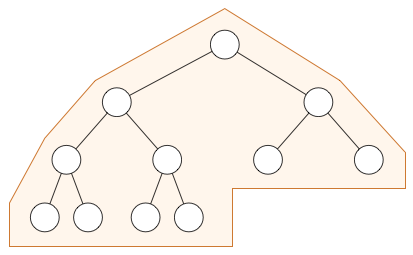


由上图可以看到，**数组中父节点总是在子节点的前面。**

注意这个方案与一些限制。你可以在普通二叉树中按照下面的方式组织数据，但是在堆中不可以：



在堆中，在当前层级所有的节点都已经填满之前不允许开是下一层的填充，所以堆总是有这样的形状：

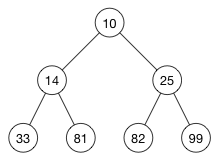


**注意：你可以使用普通树来模拟堆，但是那对空间是极大的浪费**。

小测验，假设我们有这样一个数组：

[ 10, 14, 25, 33, 81, 82, 99 ]

这是一个有效的堆吗？答案是 yes ！**一个从低到高有序排列的数组是以有效的最小堆**，我们可以将这个堆画出来：



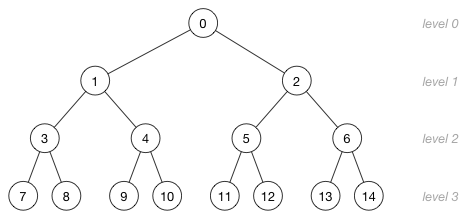
堆属性适用于每一个节点，因为父节点总是比它的字节点小。（你也可以验证一下：**一个从高到低有序排列的数组是一个有效的最大堆**）

**注意：并不是每一个最小堆都是一个有序数组！要将堆转换成有序数组，需要使用堆排序。**

#### 更多数学公式

树的高度是指从树的根节点到最低的叶节点所需要的步数，或者更正式的定义：高度是指节点之间的边的最大值。一个高度为 h 的堆有 h+1 层。

下面这个堆的高度是3，所以它有4层：



**如果一个堆有 n 个节点，那么它的高度是 h = floor(log2(n))**。这是因为我们总是要将这一层完全填满以后才会填充新的一层。上面的例子有 15 个节点，所以它的高度是 floor(log2(15)) = floor(3.91) = 3。

如果最下面的一层已经填满，那么那一层包含 2^h 个节点。树中这一层以上所有的节点数目为 2^h - 1。同样是上面这个例子，最下面的一层有8个节点，实际上就是 2^3 = 8。前面的三层一共包含7的节点，即：2^3 - 1 = 8 - 1 = 7。

所以整个堆中的节点数目为：\* 2^(h+1) - 1\*。上面的例子中，2^4 - 1 = 16 - 1 = 15

叶节点总是位于数组的 floor(n/2) 和 n-1 之间。

#### 可以用堆做什么？

**有两个原始操作用于保证插入或删除节点以后堆是一个有效的最大堆或者最小堆：**

* shiftUp(): 如果一个节点比它的父节点大（最大堆）或者小（最小堆），那么需要将它同父节点交换位置。这样是这个节点在数组的位置上升。
* shiftDown(): 如果一个节点比它的子节点小（最大堆）或者大（最小堆），那么需要将它向下移动。这个操作也称作“堆化（heapify）”。

**shiftUp 或者 shiftDown 是一个递归的过程，所以它的时间复杂度是 O(log n)。**

基于这两个原始操作还有一些其他的操作：

* insert(value): 在堆的尾部添加一个新的元素，然后使用 **shiftUp** 来修复堆。
* remove(): 移除并返回最大值（最大堆）或者最小值（最小堆）。为了将这个节点删除后的空位填补上，**需要将最后一个元素移到根节点的位置，然后使用 shiftDown 方法来修复堆**。
* removeAtIndex(index): 和 remove() 一样，差别在于可以移除堆中任意节点，而不仅仅是根节点。当它与子节点比较发现无序时使用 shiftDown()，如果与父节点比较发现无序则使用 shiftUp()。
* replace(index, value)：将一个更小的值（最小堆）或者更大的值（最大堆）赋值给一个节点。由于这个操作破坏了堆属性，所以需要使用 **shiftUp**() 来修复堆属性。

**上面所有的操作的时间复杂度都是 O(log n)，**因为 shiftUp 和 shiftDown 都很费时。还有少数一些操作需要更多的时间：

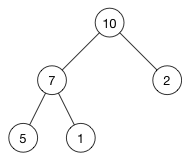
* search(value):**堆不是为快速搜索而建立的**，但是 replace() 和 removeAtIndex() 操作需要找到节点在数组中的index，所以你需要先找到这个index。时间复杂度：**O(n)**。
* buildHeap(array):通过反复调用 insert() 方法将一个（无序）数组转换成一个堆。如果你足够聪明，你可以在 **O(n)** 时间内完成。
* 堆排序：**由于堆就是一个数组**，**我们可以使用它独特的属性将数组从低到高排序。时间复杂度：O(n lg n)**。

堆还有一个 peek() 方法，不用删除节点就返回最大值（最大堆）或者最小值（最小堆）。时间复杂度 **O(1)** 。

**注意：**到目前为止，堆的常用操作还是使用 insert() 插入一个新的元素，和通过 remove()移除最大或者最小值。两者的时间复杂度都是**O(log n)**。其他的操作是用于支持更高级的应用，比如说建立一个优先队列。

##### 插入

我们通过一个插入例子来看看插入操作的细节。我们将数字 16 插入到这个堆中：

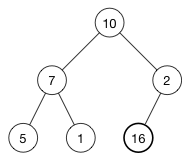
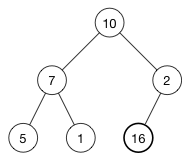
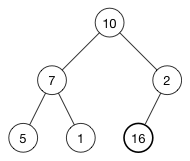


堆的数组是： [ 10, 7, 2, 5, 1 ]。

第一股是将新的元素插入到数组的尾部。数组变成：

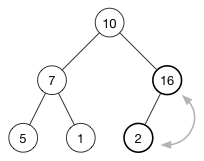
[ 10, 7, 2, 5, 1, 16 ]

相应的树变成了：



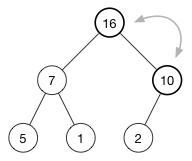
不行的是，现在堆属性不满足，因为 2 在 16 的上面，我们需要将大的数字在上面（这是一个最大堆）

为了恢复堆属性，我们需要交换 16 和 2。



现在还没有完成，因为 10 也比 16 小。我们继续交换我们的插入元素和它的父节点，**直到它的父节点比它大或者我们到达树的顶部**。这就是所谓的 **shift-up**，每一次插入操作后都需要进行。它将一个太大或者太小的数字“浮起”到树的顶部。

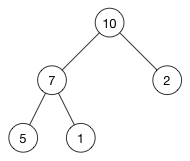
最后我们得到的堆：



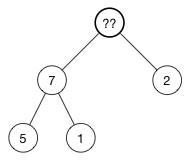
现在每一个父节点都比它的子节点大。

##### 删除根节点

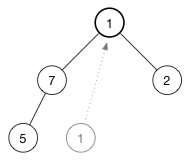
我们将这个树中的 (10) 删除：



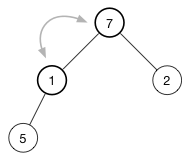
现在顶部有一个空的节点，怎么处理？



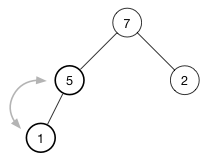
当插入节点的时候，我们将新的值返给数组的尾部。现在我们来做相反的事情：我们取出数组中的最后一个元素，将它放到树的顶部，然后再修复堆属性。



现在来看怎么 **shift-down** (1)。为了保持最大堆的堆属性，我们需要树的顶部是最大的数据。现在有两个数字可用于交换 7 和 2。我们选择这两者中的较大者称为最大值放在树的顶部，所以交换 7 和 1，现在树变成了：



继续堆化直到该节点没有任何子节点或者它比两个子节点都要大为止。对于我们的堆，我们只需要再有一次交换就恢复了堆属性：

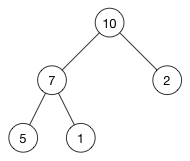


##### 删除任意节点

**绝大多数时候你需要删除的是堆的根节点，因为这就是堆的设计用途。**

但是，删除任意节点也很有用。这是 **remove() 的通用版本，它可能会使用到 shiftDown 和 shiftUp。**

我们还是用前面的例子，删除 (7):



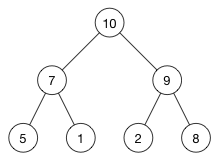
对应的数组是[ 10, 7, 2, 5, 1 ]

你知道，移除一个元素会破坏最大堆或者最小堆属性。我们需要将**删除的元素和最后一个元素交换：**

[ 10, 1, 2, 5, 7 ]

**最后一个元素就是我们需要返回的元素；然后调用 removeLast() 来将它删除。** 1比它的子节点小，所以需要 shiftDown() 来修复。

然而，shift down 不是我们要处理的唯一情况。也有可能我们需要 shift up。考虑一下从下面的堆中删除 (5) 会发生什么：



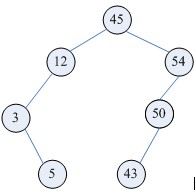
现在 (5) 和 (8) 交换了。因为 (8) 比它的父节点大，我们需要 shiftUp()。

### 栈

1. 后进先出
2. 与递归紧密联系

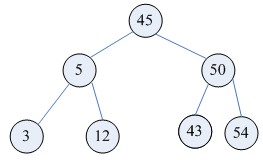
### 树

#### 1、二叉查找树（二叉排序树）BST



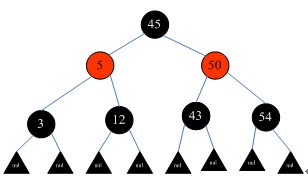
二叉查找树是一种动态查找表（图a），具有这些质：                                   
（1）若它的左子树不为空，则左子树上的所有节点的值都小于它的根节点的值；  
（2）若它的右子树不为空，则右子树上所有节点的值都大于它的根节点的值；  
（3）其他的左右子树也分别为二叉查找树；  
（4）二叉查找树是动态查找表，在查找的过程中可见添加和删除相应的元素，在这些操作中需要保持二叉查找树的以上性质。

#### 2、平衡二叉树（AVL树）



　　含有相同节点的二叉查找树可以有不同的形态，而二叉查找树的平均查找长度与树的深度有关，所以需要找出一个查找平均长度最小的一棵，那就是平衡二叉树（图b），具有以下性质：  
（1）要么是棵空树，要么其根节点左右子树的深度之差的绝对值不超过1；  
（2）其左右子树也都是平衡二叉树；  
（3）二叉树节点的平衡因子定义为该节点的左子树的深度减去右子树的深度。则平衡二叉树的所有节点的平衡因子只可能是-1,0,1。

#### 3、红黑树

（图c）

　　红黑树是一种自平衡二叉树，在平衡二叉树的基础上每个节点又增加了一个颜色的属性，节点的颜色只能是红色或黑色。具有以下性质：  
（1）根节点只能是黑色；  
（2）红黑树中所有的叶子节点后面再接上左右两个空节点，这样可以保持算法的一致性，而且所有的空节点都是黑色；  
（3）其他的节点要么是红色，要么是黑色，红色节点的父节点和左右孩子节点都是黑色，及黑红相间；  
（4）在任何一棵子树中，从根节点向下走到空节点的路径上所经过的黑节点的数目相同，从而保证了是一个平衡二叉树。

#### 4.B树

B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

系统从磁盘读取数据到内存时是以**磁盘块**（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb\_page\_size将页的大小设置为4K、8K、16K，在[MySQL](http://lib.csdn.net/base/mysql)中可通过如下命令查看页的大小：

mysql> show variables like 'innodb\_page\_size';

而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key, data] ，key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同。

一棵m阶的B-Tree有如下特性：   
1. 每个节点最多有m个孩子。   
2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。 （上界）  
3. 若根节点不是叶子节点，则至少有2个孩子   
4. 所有叶子节点都在同一层，且不包含其它关键字信息   
5. 每个非终端节点包含n个关键字信息（P0,P1,…Pn, k1,…kn）   
6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1   
7. ki(i=1,…n)为关键字，且关键字升序排序。   
8. Pi(i=1,…n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：   


每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

#### 5.B+Tree

B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 数据记录都存放在叶子节点中。

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：   


通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有22条数据记录，看不出B+Tree的优点，下面做一个推算：

InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为〖10〗^3）。也就是说一个深度为3的B+Tree索引可以维护10^3 \* 10^3 \* 10^3 = 10亿 条记录。

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree的高度一般都在2~4层。[mysql](http://lib.csdn.net/base/mysql)的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

数据库中的B+Tree索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。

## 操作系统

### 1.进程和线程以及区别

 线程：线程是进程的一个实体，是CPU调度和分派的基本单元。  
 进程：进程是具有一定独立功能的程序，它是系统进程资源分配和调度的一个独立单元。  
 区别：  
  （1）一个线程只属于一个进程，一个进程包含一个或者多个线程。  
  （2）进程拥有独立的内存单元，而多个线程共享内存。  
  （3）进程的创建调用fork或者vfork,而线程的创建调用pthead\_create，进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束。  
  （4）线程是轻量级的进程，它的创建和销毁所需要的时间比进程小很多，所有操作系统中的执行功能都是创建线程去完成的。  
  （5）线程中执行时一般都要进行同步和互斥，因为他们共享同一进程的资源。

**进程与线程的区别？**

（1）进程有自己的独立地址空间，线程没有

（2）进程是资源分配的最小单位，线程是CPU调度的最小单位

（3）进程和线程通信方式不同(线程之间的通信比较方便。同一进程下的线程共享数据（比如全局变量，静态变量），通过这些数据来通信不仅快捷而且方便，当然如何处理好这些访问的同步与互斥正是编写多线程程序的难点。而进程之间的通信只能通过[进程通信](http://baike.baidu.com/view/549640.htm)的方式进行。)

（4）进程上下文切换开销大，线程开销小

（5）一个进程挂掉了不会影响其他进程，而线程挂掉了会影响其他线程

（6）对进程操作一般开销都比较大，对线程开销就小了

### 2死锁(deadlock)？

**死锁** **:**两个或两个以上的进程在执行过程中,因争夺资源而造成的一种**互相等待**的现象,若无**外力**作用,它们都将无法推进下去

**产生原因：**（1） 系统资源不足。 （2） 进程运行推进顺序不合里。 （3） 资源分配不当。

**必要条件：**

（1） **互斥条件**：一个资源每次只能被一个进程使用。

（2） **请求与保持条件**：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） **不剥夺条件**:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） **循环等待条件**:若干进程之间形成一种头尾相接的循环等待资源关系。

**死锁的解除与预防：**

在系统设计、进程调度等方面注意不让四个必要条件成立，确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源。

### 3.虚拟内存？优缺点？

定义：具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充得一种存储器系统。其逻辑容量由内存之和和外存之和决定。

与传统存储器比较虚拟存储器有以下三个主要特征：

* 多次性，是指无需在作业运行时一次性地全部装入内存，而是允许被分成多次调入内存运行。
* 对换性，是指无需在作业运行时一直常驻内存，而是允许在作业的运行过程中，进行换进和换出。
* 虚拟性，是指从逻辑上扩充内存的容量，使用户所看到的内存容量，远大于实际的内存容量。

虚拟内存的实现有以下两种方式：

* 请求分页存储管理。
* 请求分段存储管理。

### 4.进程间通信有哪些方式？它们的区别？

1.管道（pipe）：管道是一种半双工的通信方式，数据只能单向流动，而且只能在有血缘关系的进程间使用，进程的血缘关系通常是指父子进程关系。

2.命名管道（named pipe）：也是半双工的通信方式，但是它允许无亲缘关系关系进程间通信。

3.信号（signal）：是一种比较复杂的通信方式，用于通知接收进程某一事件已经发生。

4.信号量（semophere）：信号量是一个计数器，可用来控制多个进程对共享资源的访问。它通常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

5.消息队列（message queue）:消息队列是由消息组成的链表，存放在内核中，并由消息队列标识符标识。消息队列克服了信号传递消息少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

6.共享内存（shared memory）:就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问，共享内存是最快的IPC方式，它是针对其他进程间的通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量等配合使用，来实现进程间的同步和通信。

7.套接字（socket）：套接口也是进程间的通信机制，与其他通信机制不同的是它可用于不同及其间的进程通信。

几种方式的比较：

管道：速度慢、容量有限

消息队列：容量收到系统限制，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。

信号量：不能传递复杂信息，只能用来同步。

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。

### 5.进程的调度算法有哪些？

　　1.先来先服务（FCFS）:此算法的原则是按照作业到达后备作业队列（或进程进入就绪队列）的先后次序选择作业（或进程）

　　2.短作业优先（SJF:Shortest Process First）：这种算法主要用于作业调度，它从作业后备序列中挑选所需运行时间最短的作业进入主存运行。

　　3.时间片轮转调度算法：当某个进程执行的时间片用完时，调度程序便终止该进程的执行，并将它送到就绪队列的末尾，等待分配下一时间片再执行。然后把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证队列中的所有进程，在已给定的时间内，均能获得一时间片处理机执行时间。

　　4.高响应比优先：按照高响应比（已等待时间+要求运行时间）/要求运行时间 优先的原则，在每次选择作业投入运行时，先计算此时后备作业队列中每个作业的响应比RP。选择最大的作业投入运行。

　　5.优先权调度算法：按照进程的优先权大小来调度。使高优先权进程得到优先处理的调度策略称为优先权调度算法。注意：优先数越多，优先权越小。

　　6.多级队列调度算法：多队列调度是根据作业的性质和类型的不同，将就绪队列再分为若干个队列，所有的作业（进程）按其性质排入相应的队列中，而不同的就绪队列采用不同的调度算法。

# 多线程

## 1.多线程中的i++线程安全吗？为什么？

i++和++i的线程安全分为两种情况：  
1、如果i是局部变量，那么是线程安全的。因为局部变量是线程私有的，别的线程访问不到，其实也可以说没有线程安不安全之说，因为别的线程对他造不成影响。  
2、如果i是全局变量，则同一进程的不同线程都可能访问到该变量，因而是线程不安全的，会产生脏读。

## 2.启动一个线程是用run()还是start()以及他们的区别?

通常，系统通过调用线程类的 start()方法来启动一个线程，此时该线程处于就绪状态，而非运行状态，也就意味着这个线程可以被 JVM 来调度执行。在调度过程中，JVM 通过调用线程类的 run()方法来完成实际的操作，当 run() 方法结束后，此线程就会终止。

如果直接调用线程类的 run ()方法，这会被当作一个普通的函数调用，程序中仍然只有主线程这一个线程，也就是说， start 方法()能够异步地调用 run()方法，但是直接调用 run( )方法却是同步的，因此也就无法达到多线程的目的。

由此可知，只有通过调用线程类的 start()方法才能真正达到多线程的目的。

## 3.线程的sleep()方法和yield()方法有什么区别？ 和wait()有什么区别？

1. sleep给其他线程运行机会时不考虑线程优先级，因此会给低优先级的线程运行的机会，而yield只会给相同优先级或更高优先级的线程以运行的机会。
2. 线程执行sleep()方法后会转入阻塞状态，所以，执行sleep()方法的线程在指定的时间内肯定不会被执行，而yield()方法只是使当前线程重新回到可执行状态，所以执行yield() 方法的线程有可能在进入到可执行状态后马上又被执行。
3. sleep()方法声明抛出InterruptedException,而yield()方法没有声明任何异常。
4. sleep()方法比yield()方法(跟操作系统相关)具有更好的可移植性。

1.sleep是Thread的静态方法，wait是Object类（为什么）

2.最主要sleep不释放锁，wait释放，使得其他线程可以使用同步控制块或者方法(锁代码块和方法锁)。

3sleep:任何地方使用, wait，notify和notifyAll:同步控制方法 同步控制块使用（JDK强制，调用前必须获得对象锁）

4sleep必须捕获异常，而wait，notify和notifyAll不需要

## 4.当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？

不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（不是等待池）中等待对象锁。

## 锁

#### 1.CAS的实现原理

CAS：无锁算法，3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B并返回true，否则什么都不做并返回false。CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。

#### **2.乐观锁和悲观锁**

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（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。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。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁：总是假设最好的情况，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，可以使用版本号机制和CAS算法实现。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。

两种锁的使用场景

像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

乐观锁常见的两种实现方式

乐观锁一般会使用版本号机制或CAS算法实现。

1. 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

2. CAS算法

即compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数

需要读写的内存值 V

进行比较的值 A

拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

乐观锁的缺点

ABA 问题是乐观锁一个常见的问题

1 ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2 循环时间长开销大

自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作。

CAS与synchronized的使用情景

简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）

对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。

对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充： Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 “重量级锁” 。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 Lock-Free 的队列，基本思路是 自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

## 9.你有哪些多线程开发良好的实践?

1.给线程命名

2.最小化同步范围

3.优先使用volatile

4.尽可能使用更高层次的并发工具而非wait和notify()来实现线程通信,如BlockingQueue,Semeaphore

5.优先使用并发容器而非同步容器.

6.考虑使用线程池

## 10.线程池

### 1.什么是线程池

线程池是一种多线程处理形式，处理过程中将任务提交到线程池，任务的执行交由线程池来管理。

如果每个请求都创建一个线程去处理，那么服务器的资源很快就会被耗尽，使用线程池可以减少创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

### ****2.几种常见的线程池及使用场景****

1、newSingleThreadExecutor

单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。（一个单线程的线程池，可以用于需要保证顺序执行的场景，并且只有一个线程在执行。）

2、newFixedThreadPool

定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。可以用于已知并发压力的情况下，对线程数做限制。

3、newCachedThreadPool

可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。比较适合处理执行时间比较小的任务。（一个可以无限扩大的线程池， ）

4、newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。可以延时启动，定时启动的线程池，适用于需要多个后台线程执行周期任务的场景。

5. newWorkStealingPool：一个拥有多个任务队列的线程池，可以减少连接数，创建当前可用cpu数量的线程来并行执行。

### 3.为什么要用线程池？

创建线程和销毁线程的花销是比较大的，这些时间有可能比处理业务的时间还要长。这样频繁的创建线程和销毁线程，再加上业务工作线程，消耗系统资源的时间，可能导致系统资源不足。使用线程池可以把创建和销毁的线程的过程去掉

### 4.使用线程池的好处：

**（1）降低资源消耗。** 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

**（2）提高响应速度。** 当任务到达时，任务可以不需要的等到线程创建就能立即执行。

**（3）提高线程的可管理性。** 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

### 6.几种重要的参数

corePoolSize：线程池中的核心线程数量，这几个核心线程，只是在没有用的时候，也不会被回收

maximumPoolSize：线程池中可以容纳的最大线程的数量

keepAliveTime：就是线程池中除了核心线程之外的其他的最长可以保留的时间，因为在线程池中，除了核心线程即使在无任务的情况下也不能被清 除，其余的都是有存活时间的，意思就是非核心线程可以保留的最长的空闲时间，

Util：就是计算这个时间的一个单位。

workQueue：就是等待队列，任务可以储存在任务队列中等待被执行，执行的是先进先出原则。

threadFactory：创建线程的线程工厂。

Handler：一种拒绝策略，我们可以在任务满了之后，拒绝执行某些任务。

### 7.线程池的拒绝策略

当请求任务不断的过来，而系统此时又处理不过来的时候，我们需要采取的策略是拒绝服务。RejectedExecutionHandler接口提供了拒绝任务处理的自定义方法的机会。在ThreadPoolExecutor中已经包含四种处理策略。

AbortPolicy策略： 当任务添加到线程池中被拒绝时，它将抛出 RejectedExecutionException 异常。(异常)

CallerRunsPolicy 策略：只要线程池未关闭，该策略直接在调用者线程中，运行当前的被丢弃的任务。

DiscardOleddestPolicy策略： 线程池会放弃等待队列中最旧的未处理任务，然后将被拒绝的任务添加到等待队列中。

DiscardPolicy策略：线程池将丢弃被拒绝的任务，不予任何处理。

除了JDK默认提供的四种拒绝策略，我们可以根据自己的业务需求去自定义拒绝策略，自定义的方式很简单，直接实现RejectedExecutionHandler接口即可。

### 10.线程池都有哪几种工作队列

1、ArrayBlockingQueue

是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

2、LinkedBlockingQueue

一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列

3、SynchronousQueue

一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。

4、PriorityBlockingQueue

一个具有优先级的无限阻塞队列。

## 11.线程和进程各自有什么区别和优劣呢？

1.进程是资源分配的最小单位，线程是程序执行的最小单位。

2.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。

3.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。

4.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。

## 12.Callable 和 Runnable接口的区别

1.Callable规定的方法是call()，而Runnable规定的方法是run().

2.Callable的任务执行后可返回值，而Runnable的任务是不能返回值的。

3.call()方法可抛出异常，而run()方法是不能抛出异常的。

4.运行Callable任务可拿到一个Future对象， Future表示异步计算的结果。

5.它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。

6.通过Future对象可了解任务执行情况，可取消任务的执行，还可获取任务执行的结果。

7.Callable是类似于Runnable的接口，实现Callable接口的类和实现Runnable的类都是可被其它线程执行的任务。

(【帮助理解】相同点：

两者都是接口；（废话）

两者都可用来编写多线程程序；

两者都需要调用Thread.start()启动线程；)

## 13.ReentrantLock

ReentrantLock重入锁，是实现Lock接口的一个类，支持重入性，表示能够对共享资源能够重复加锁，即当前线程获取该锁再次获取不会被阻塞。在java关键字synchronized隐式支持重入性synchronized通过获取自增，释放自减的方式实现重入。与此同时，ReentrantLock还支持公平锁和非公平锁两种方式。

1.重入性的实现原理

要想支持重入性，就要解决两个问题：1. 在线程获取锁的时候，如果已经获取锁的线程是当前线程的话则直接再次获取成功；2. 由于锁会被获取n次，那么只有锁在被释放同样的n次之后，该锁才算是完全释放成功。通过[这篇文章](https://link.juejin.im?target=https://juejin.im/post/5aeb07ab6fb9a07ac36350c8)，我们知道，同步组件主要是通过重写AQS的几个protected方法来表达自己的同步语义。针对第一个问题，我们来看看ReentrantLock是怎样实现的，以非公平锁为例，判断当前线程能否获得锁为例，核心方法为nonfairTryAcquire：

为了支持重入性，增加了处理逻辑，如果该锁已经被线程所占有了，会继续检查占有线程是否为当前线程，如果是的话，同步状态加1返回true，表示可以再次获取成功。每次重新获取都会对同步状态进行加一的操作，那么释放的时候处理思路是怎样的了？（依然还是以非公平锁为例）核心方法为tryRelease：

需要注意的是，重入锁的释放必须得等到同步状态为0时锁才算成功释放，否则锁仍未释放。如果锁被获取n次，释放了n-1次，该锁未完全释放返回false，只有被释放n次才算成功释放，返回true。到现在我们可以理清ReentrantLock重入性的实现了，也就是理解了同步语义的第一条。

1. ReentrantLock支持两种锁：**公平锁**和**非公平锁**。**何谓公平性，是针对获取锁而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求上的绝对时间顺序，满足FIFO**。ReentrantLock的构造方法无参时是构造**非公平锁**

另外还提供了另外一种方式，可传入一个boolean值，true时为公平锁，false时为非公平锁.

在上面非公平锁获取时（nonfairTryAcquire方法）只是简单的获取了一下当前状态做了一些逻辑处理，并没有考虑到当前同步队列中线程等待的情况。我们来看看公平锁的处理逻辑是怎样的，核心方法为tryAcquire：

这段代码的逻辑与nonfairTryAcquire基本上一直，唯一的不同在于增加了hasQueuedPredecessors的逻辑判断，方法名就可知道该方法用来判断当前节点在同步队列中是否有前驱节点的判断，如果有前驱节点说明有线程比当前线程更早的请求资源，根据公平性，当前线程请求资源失败。如果当前节点没有前驱节点的话，再才有做后面的逻辑判断的必要性。**公平锁每次都是从同步队列中的第一个节点获取到锁，而非公平性锁则不一定，有可能刚释放锁的线程能再次获取到锁**。

**公平锁 VS 非公平锁**

公平锁每次获取到锁为同步队列中的第一个节点，**保证请求资源时间上的绝对顺序**，而非公平锁有可能刚释放锁的线程下次继续获取该锁，则有可能导致其他线程永远无法获取到锁，**造成“饥饿”现象**。

公平锁为了保证时间上的绝对顺序，需要频繁的上下文切换，而非公平锁会降低一定的上下文切换，降低性能开销。因此，ReentrantLock默认选择的是非公平锁，则是为了减少一部分上下文切换，**保证了系统更大的吞吐量**。

## 14单线程多线程区别

1. 多线程的产生并不是因为发明了多核CPU甚至现在有多个CPU+多核的硬件，也不是因为多线程CPU运行效率比单线程高。单从CPU的运行效率上考虑，单任务进程及单线程效率是最高的，因为CPU没有任何进程及线程的切换开销，  
   （2）实际上，多线程的出现主要为了解决IO设备的读写速度往往比CPU的处理速度慢造成的单线程程序运行阻塞问题，一个极端的例子就是如果你需要用户在键盘上输入一个数据，当用户没有输入前，单线程程序就阻塞了，多线程程序就可以放个音乐或继续干一些程序中除了键盘输入外的工作，因此，多线程能提高因程序由于等待某个资源阻塞时其他资源的利用率（是利用率不是效率）。  
   （3）因此多线程与单线程的最大区别，多线程程序能在等待某个IO操作时，继续完成非这个IO的其他工作，有利于提高完成整个任务的效果和速度。此外，多线程程序与单线程程序对程序设计也有不同的流程和结构，多线程需要考虑对静态变量等资源的操作互锁及程序执行的同步问题。

## 15.保证十个线程的执行顺序

用join方法

## 16.sleep wait join怎么用的

sleep:,强制当前线程休眠.当休眠时间到期后恢复到可运行状态.不一定会立即执行,具体取决于线程调度器.(不释放锁)

wait:使当前线程,进入到一个和对象相关的等待池中,同时失去对象的锁,其他线程可以访问.

通过notify(),notifyAll(),或者指定超时时间来唤醒当前等待池中线程.进入到锁池

用在synchronized代码块中,否则会抛出异常.

join：将并行运行的线程改为串行.前线程等待指定线程终止后在执行,将两个交替执行的线程合并为顺序执行的线程.比如在B线程中调用A线程的join()方法,直到A线程执行完毕,B线程才会继续执行.

## 17

# 关键字

## Synchronize

#### synchronized可以修饰什么(可以写在哪些地方)？

可以修饰普通方法、静态方法、代码块，锁定的对象分别是：当前实例，当前Class实例，synchronized括号里包起来的实例。

#### 2.synchronized修饰方法、代码块、静态方法时有何不同？锁定的对象分别是什么？

#### 3.synchronized性能如何？jdk是否有对synchronized做过优化？是在哪个版本做了优化？优化的内容是？

JAVA6之前synchronized是重量级锁。JAVA6引入了偏向锁和轻量级锁并默认启用，JVM会动态地根据锁的竞争激烈程度适当对锁进行升级。如果一个synchronized方法一开始没有竞争，这时它的锁可能是偏向锁或轻量级锁，但随着时间的变化这个锁竞争越来越激烈，竞争激烈到一定程度JVM会将该锁升级为重量级锁)。锁升级只能由轻量级锁升级为重量级锁，不会由重量级锁降级为轻量级锁。

java6引入了偏向锁和轻量级锁，这使得锁竞争不激烈或几乎没有竞争时大大提高了获取锁和释放锁的性能。

#### 4.Synchronized加在static 方法上和不加在static方法上不会产生互斥

#### 5.Lock和synchronized的区别，二者孰优孰劣？各自的使用场景？

（1）Lock是一个接口，而synchronized是关键字。

（2）Lock必须手动释放锁，synchronized会自动释放锁。

（3）Lock可以让等待锁的线程响应中断，而synchronized不会，线程会一直等待下去。

（4）通过Lock可以知道线程有没有拿到锁，而synchronized不能。

（5）Lock能提高多个线程读操作的效率。

（6）而Lock是块范围内的，synchronized能锁住类、方法和代码块。

#### 6.synchronized与Lock的区别及底层实现

1、synchronized 是Java内置的关键字，使用后会自动释放锁，

Lock是java.util.concurrent.Locks 包下的一个接口，必须要手动释放。特别是在发生异常时，需要在 finally 块中进行手动释放，否则会发生死锁行为

2、Lock可响应中断，而synchronized 不能响应中断，并且Lock提供了更丰富的方法实现；例如

Lock() ; //获取锁

tryLock(); //获取锁

tryLock(long time, TimeUnit unit); //在一定时间单位内等待

lockInterruptibly(); //获取锁，可响应中断(AB线程同时获取锁，A得到后，B进行等待，则B会被Tread.interrupt()方法中端并可去执行其他的代码逻辑，而synchronized无法被中端)

unlock(); //释放锁

3、synchronized 是非公平锁，即不能保证等待锁线程的顺序，

Lock的实现 ReentrantLock 可通过实例化true or false 的构造参数实现公平锁和非公平锁，默认为非公平锁

ReentrantLock的构造参数

4、ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法

lck.isFair();

lck.isLocked();

lck.getHoldCount();

lck.getQueueLength();

lck.wait();

5、synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；

6、Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

7、都是可重入锁：在执行对象中所有同步方法不用再次获得锁

8、synchronized是一个悲观锁，Lock是一个乐观锁（底层基于volatile和cas实现）

二、底层实现

1、synchronznized映射成字节码指令就是增加两个指令：monitorenter、monitorexit，

当一条线程执行时遇到monitorenter指令时，它会尝试去获得锁，如果获得锁，那么所计数器+1（为什么要加1，因为它是可重入锁，可根据这个琐计数器判断锁状态），如果没有获得锁，那么阻塞，

当它遇到一个monitoerexit时，琐计数器会-1，当计数器为0时，就释放锁

（tips：节码中出现的两个monitoerexit指令的原因是：一个正常执行-1，令一个异常时执行，这两个用goto的方式只执行一个）

2、Lock底层则基于volatile和cas实现

## 2.Static

### static用法

1. 静态导入，就是把一个静态变量或者静态方法一次性导入，导入后可以直接使用该方法或者变量，而不再需要写类名。
2. 静态变量：静态变量属于类，在内存中只有一个实例。当静态变量所在的类被加载的时候，就会为该静态变量分配内存空间，该变量就可以被使用。静态变量有两种被使用方式：【类名.变量名】和【对象.变量名】。
3. 静态方法：static方法是类的方法，不需要创建对象就可以使用，比如Math类里面的方法。使用方法【对象.方法名】或者【类名.方法名】
4. 静态代码块：主要目的就是对静态属性进行初始化。静态代码块可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果static代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。
5. 静态内部类：被static修饰的内部类，它可以不依赖于外部类实例对象而被实例化，而通常的内部类需要在外部类实例化后才能实例化。静态内部类不能与外部类有相同的名字，不能访问外部类的普通成员变量，只能访问内部类中的静态成员和静态方法（包括私有类型）。

### 2.详解static

#### 静态导入

public class OldImport {

public static void main(String[] args) {

double a = Math.cos(Math.PI / 2);

double b = Math.pow(2.4,1.2);

double r = Math.max(a,b);

System.out.println(r);

}

}

Math出现的次数太多了

**import static java.lang.Math.\*;**

public class StaticImport {

public static void main(String[] args) {

double a = cos(PI / 2);

double b = pow(2.4,1.2);

double r = max(a,b);

System.out.println(r);

}

}

这就是静态导入。

我们平时使用一个静态方法的时候，都是【类名.方法名】，使用静态变量的时候都是【类名.变量名】，如果一段代码中频繁的用到了这个类的方法或者变量，我们就要写好多次类名，比如上面的Math，这显然不是喜欢偷懒的程序员所希望做的，所以出现了静态导入的功能。

静态导入，就是把一个静态变量或者静态方法一次性导入，导入后可以直接使用该方法或者变量，而不再需要写对象名。

import static java.lang.Double.\*;import static java.lang.Integer.\*;import static java.lang.Math.\*;import static java.text.NumberFormat.\*;

public class ErrorStaticImport {

// 输入半径和精度要求，计算面积

public static void main(String[] args) {

double s = PI \* parseDouble(args[0]);

NumberFormat nf = getInstance();

nf.setMaximumFractionDigits(parseInt(args[1]));

formatMessage(nf.format(s));

}

// 格式化消息输出

public static void formatMessage(String s){

System.out.println(" 圆面积是："+s);

}

}

就这么一段程序，看着就让人火大：常量PI，这知道，是圆周率；parseDouble 方法可能是Double 类的一个转换方法，这看名称也能猜测到。那紧接着的getInstance 方法是哪个类的？是ErrorStaticImport本地类的方法？不对呀，没有这个方法，哦，原来是NumberFormate 类的方法，这和formateMessage 本地方法没有任何区别了。这代码也太难阅读了，这才几行？要是你以后接别人的代码，看到成千上万行这种代码大概你想死的心都有了吧？

所以，不要滥用静态导入！！！不要滥用静态导入！！！不要滥用静态导入！！！

正确使用静态导入的姿势是什么样子的呢？

import java.text.NumberFormat;

import static java.lang.Double.parseDouble;import static java.lang.Integer.parseInt;import static java.lang.Math.PI;import static java.text.NumberFormat.getInstance;

public class ErrorStaticImport {

// 输入半径和精度要求，计算面积

public static void main(String[] args) {

double s = PI \* parseDouble(args[0]);

NumberFormat nf = getInstance();

nf.setMaximumFractionDigits(parseInt(args[1]));

formatMessage(nf.format(s));

}

// 格式化消息输出

public static void formatMessage(String s){

System.out.println(" 圆面积是："+s);

}

}

没错，这才是正确的姿势，你使用哪个方法或者哪个变量，就把他导入进来，而不要使用通配符（\*）！

并且，由于不用写类名了，所以难免会和本地方法混淆。所以，本地方法在起名字的时候，一定要起得有意义，让人一看这个方法名大概就能知道你这个方法是干什么的，而不是什么method1(),method2()，鬼知道你写的是什么。。

**总结：**

* 不使用\*通配符，除非是导入静态常量类（只包含常量的类或接口）。
* 方法名是具有明确、清晰表象意义的工具类。

#### 静态变量

java类提供了两种类型的变量：用static修饰的静态变量和不用static修饰的成员变量。

**静态变量**属于类，在内存中只有一个实例。当静态变量所在的类被加载的时候，就会为该静态变量分配内存空间，该变量就可以被使用。静态变量有两种被使用方式：【类名.变量名】和【对象.变量名】。

**实例变量**属于对象，只有对象被创建后，实例对象才会被分配空间，才能被使用。他在内存中存在多个实例，只能通过【对象.变量名】来使用。

第一篇文章《万物皆对象》中讲过，java的内存大体上有四块：堆，栈，静态区，常量区。

其中的静态区，就是用来放置静态变量的。当静态变量的类被加载时，虚拟机就会在静态区为该变量开辟一块空间。所有使用该静态变量的对象都访问这一个空间。

### ****一个例子学习静态变量与实例变量****。

public class StaticAttribute {

public static int staticInt = 10;

public static int staticIntNo ;

public int nonStatic = 5;

public static void main(String[] args) {

StaticAttribute s = new StaticAttribute();

System.out.println("s.staticInt= " + s.staticInt);

System.out.println("StaticAttribute.staticInt= " + StaticAttribute.staticInt);

System.out.println("s.staticIntNo= " + s.staticIntNo);

System.out.println("StaticAttribute.staticIntNo= " + StaticAttribute.staticIntNo);

System.out.println("s.nonStatic= " + s.nonStatic);

System.out.println("使用s,让三个变量都+1");

s.staticInt ++;

s.staticIntNo ++;

s.nonStatic ++;

StaticAttribute s2 = new StaticAttribute();

System.out.println("s2.staticInt= " + s2.staticInt);

System.out.println("StaticAttribute.staticInt= " + StaticAttribute.staticInt);

System.out.println("s2.staticIntNo= " + s2.staticIntNo);

System.out.println("StaticAttribute.staticIntNo= " + StaticAttribute.staticIntNo);

System.out.println("s2.nonStatic= " + s2.nonStatic);

}

}// 结果：// s.staticInt= 10// StaticAttribute.staticInt= 10// s.staticIntNo= 0// StaticAttribute.staticIntNo= 0// s.nonStatic= 5// 使用s,让三个变量都+1// s2.staticInt= 11// StaticAttribute.staticInt= 11// s2.staticIntNo= 1// StaticAttribute.staticIntNo= 1// s2.nonStatic= 5

从上例可以看出，静态变量只有一个，被类拥有，所有对象都共享这个静态变量，而实例对象是与具体对象相关的。

与c++不同的是，在java中，不能在方法体中定义static变量，我们之前所说的变量，都是类变量，不包括方法内部的变量。

那么，静态变量有什么用途呢？

### ****静态变量的用法****

最开始的代码中有一个静态变量 --- PI，也就是圆周率。为什么要把它设计为静态的呢？因为我们可能在程序的任何地方使用到这个变量，如果不是静态的，那么我们每次使用这个变量的时候都要创建一个Math对象，不仅代码臃肿而且浪费了内存空间。

所以，当你的某一个变量会经常被外部代码访问的时候，可以考虑设计为静态的。

#### 静态方法

同样，静态方法大家应该也比较熟悉了。就是在定义类的时候加一个static修饰符。

与静态变量一样，java类也同时提供了static方法和非static方法。

* static方法是类的方法，不需要创建对象就可以使用，比如Math类里面的方法。使用方法【对象.方法名】或者【类名.方法名】
* 非static方法是对象的方法，只有对象呗创建出来以后才可以被使用。使用方法【对象.方法名】

static怎么用代码写我想大家都知道，这里我就不举例了，你们看着烦，我写着也烦。

### ****注意事项****

static方法中不能使用this和super关键字，不能调用非static方法，只能访问所属类的静态变量和静态方法。因为当static方法被调用的时候，这个类的对象可能还没有创建，即使已经被创建了，也无法确认调用那个对象的方法。不能访问非静态方法同理。

### ****用途---单例模式****

static的一个很常见的用途是实现单例模式。单例模式的特点是一个类只能有一个实例，为了实现这一功能，必须隐藏该类的构造函数，即把构造函数声明为private，并提供一个创建对象的方法。我们来看一下怎么实现：

public class Singleton {

private static Singleton singleton;

public static Singleton getInstance() {

if (singleton == null) {

singleton = new Singleton();

}

return singleton;

}

private Singleton() {

}

}

这个类，只会有一个对象。

### ****其他****

用public修饰的static成员变量和成员方法本质是全局变量和全局方法，当声明它类的对象时，不生成static变量的副本，而是类的所有实例共享同一个static变量。

static 变量前可以有private修饰，表示这个变量可以在类的静态代码块中，或者类的其他静态成员方法中使用（当然也可以在非静态成员方法中使用--废话），但是不能在其他类中通过类名来直接引用，这一点很重要。

实际上你需要搞明白，private是访问权限限定，static表示不要实例化就可以使用，这样就容易理解多了。static前面加上其它访问权限关键字的效果也以此类推。

### ****静态方法的用场****

静态变量可以被非静态方法调用，也可以被静态方法调用。但是静态方法只能被静态方法调用。

一般工具方法会设计为静态方法，比如Math类中的所有方法都是惊天的，因为我们不需要Math类的实例，我们只是想要用一下里面的方法。所以，你可以写一个通用的 工具类，然后里面的方法都写成静态的。

#### 静态代码块

在讲静态代码块之前，我们先来看一下，什么是代码块。

### ****什么是代码块****

所谓代码块就是用大括号将多行代码封装在一起，形成一个独立的数据体，用于实现特定的算法。一般来说代码块是不能单独运行的，它必须要有运行主体。在Java中代码块主要分为四种：普通代码块，静态代码块，同步代码块和构造代码块。

### ****四种代码块****

普通代码块

普通代码块是我们用得最多的也是最普遍的，它就是在方法名后面用{}括起来的代码段。普通代码块是不能够单独存在的，它必须要紧跟在方法名后面。同时也必须要使用方法名调用它。

public void common(){

System.out.println("普通代码块执行");

}

* 静态代码块

静态代码块就是用static修饰的用{}括起来的代码段，它的主要目的就是对静态属性进行初始化。

静态代码块可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果static代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。

看一段代码：

public class Person{

private Date birthDate;

public Person(Date birthDate) {

this.birthDate = birthDate;

}

boolean isBornBoomer() {

Date startDate = Date.valueOf("1990");

Date endDate = Date.valueOf("1999");

return birthDate.compareTo(startDate)>=0 && birthDate.compareTo(endDate) < 0;

}

}

isBornBoomer是用来这个人是否是1990-1999年出生的，而每次isBornBoomer被调用的时候，都会生成startDate和birthDate两个对象，造成了空间浪费，如果改成这样效率会更好：

public class Person{

private Date birthDate;

private static Date startDate,endDate;

static{

startDate = Date.valueOf("1990");

endDate = Date.valueOf("1999");

}

public Person(Date birthDate) {

this.birthDate = birthDate;

}

boolean isBornBoomer() {

return birthDate.compareTo(startDate)>=0 && birthDate.compareTo(endDate) < 0;

}

}

因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。

* 同步代码块

使用 synchronized 关键字修饰，并使用“{}”括起来的代码片段，它表示同一时间只能有一个线程进入到该方法块中，是一种多线程保护机制。

等讲多线程的时候，在详细讲解这种代码块~

* 构造代码块

在类中直接定义没有任何修饰符、前缀、后缀的代码块即为构造代码块。我们明白一个类必须至少有一个构造函数，构造函数在生成对象时被调用。构造代码块和构造函数一样同样是在生成一个对象时被调用，那么构造代码在什么时候被调用？如何调用的呢？

看一段代码：

public class CodeBlock {

private int a = 1;

private int b ;

private int c ;

//静态代码块

static {

int a = 4;

System.out.println("我是静态代码块1");

}

//构造代码块

{

int a = 0;

b = 2;

System.out.println("构造代码块1");

}

public CodeBlock(){

this.c = 3;

System.out.println("构造函数");

}

public int add(){

System.out.println("count a + b + c");

return a + b + c;

}

//静态代码块

static {

System.out.println("我是静态代码块2，我什么也不做");

}

//构造代码块

{

System.out.println("构造代码块2");

}

public static void main(String[] args) {

CodeBlock c = new CodeBlock();

System.out.println(c.add());

System.out.println();

System.out.println("\*\*\*\*\*\*\*再来一次\*\*\*\*\*\*\*\*\*");

System.out.println();

CodeBlock c1 = new CodeBlock();

System.out.println(c1.add());

}

}//结果：//我是静态代码块1//我是静态代码块2，我什么也不做//构造代码块1//构造代码块2//构造函数//count a + b + c//6////\*\*\*\*\*\*\*再来一次\*\*\*\*\*\*\*\*\*////构造代码块1//构造代码块2//构造函数//count a + b + c//6

这段代码综合了构造代码块，普通代码块和静态代码块。我们来总结一下：

* 静态代码块只会执行一次。有多个静态代码块时按顺序依次执行。
* 构造代码块每次创建新对象时都会执行。有多个时依次执行。
* 执行顺序：静态代码块 > 构造代码块 > 构造函数。
* 构造代码块和静态代码块有自己的作用域，作用域内部的变量不影响作用域外部。

构造代码块的应用场景：

1、 初始化实例变量  
如果一个类中存在若干个构造函数，这些构造函数都需要对实例变量进行初始化，如果我们直接在构造函数中实例化，必定会产生很多重复代码，繁琐和可读性差。这里我们可以充分利用构造代码块来实现。这是利用编译器会将构造代码块添加到每个构造函数中的特性。

2、 初始化实例环境  
一个对象必须在适当的场景下才能存在，如果没有适当的场景，则就需要在创建对象时创建此场景。我们可以利用构造代码块来创建此场景，尤其是该场景的创建过程较为复杂。构造代码会在构造函数之前执行。

## ****静态内部类****

被static修饰的内部类，它可以不依赖于外部类实例对象而被实例化，而通常的内部类需要在外部类实例化后才能实例化。静态内部类不能与外部类有相同的名字，不能访问外部类的普通成员变量，只能访问内部类中的静态成员和静态方法（包括私有类型）。

由于还没有详细讲解过内部类，这里先一笔带过，在讲解内部类的时候会详细分析静态内部类。

只有内部类才能被static修饰，普通的类不可以。

## ****总结****

本文内容就先到这里，我们再来回顾一下学了什么：

static关键字的五种用法：

* + 静态导入
  + 静态变量
  + 静态方法
  + 静态代码块

代码块

* 普通代码块
* 静态代码块
* 构造代码块
* 同步代码块

回忆一下这些知识点的内容，如果想不起来，记得翻上去再看一遍~

### ****彩蛋 ------ 继承+代码块的执行顺序****

如果既有继承，又有代码块，执行的顺序是怎样呢？

public class Parent {

static {

System.out.println("父类静态代码块");

}

{

System.out.println("父类构造代码块");

}

public Parent(){

System.out.println("父类构造函数");

}

}

class Children extends Parent {

static {

System.out.println("子类静态代码块");

}

{

System.out.println("子类构造代码块");

}

public Children(){

System.out.println("子类构造函数");

}

public static void main(String[] args) {

new Children();

}

}

//结果：//父类静态代码块//子类静态代码块//父类构造代码块//父类构造函数//子类构造代码块//子类构造函数

结果你也知道了：

先执行静态内容(先父类后子类)，然后执行父类非静态，最后执行子类非静态。（非静态包括构造代码块和构造函数，构造代码块先执行）

## 3.Violate

### 1、介绍一下volatile

（1）JVM提供的最轻量级的同步机制

（2）特性：

1）保证可见性，不保证原子性

    a.当写一个volatile变量时，JVM会把该线程本地内存中的变量强制刷新到主内存中去；

    b.这个写会操作会导致其他线程中的缓存无效。

  2）禁止指令重排

    重排序是指编译器和处理器为了优化程序性能而对指令序列进行排序的一种手段。重排序需要遵守一定规则：

1. 重排序操作不会对存在数据依赖关系的操作进行重排序。

比如：a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。

    b.**重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变**

　    比如：a=1;b=2;c=a+b这三个操作，第一步（a=1)和第二步(b=2)由于不存在数据依赖关系， 所以可能会发生重排序，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。

    重排序在单线程下一定能保证结果的正确性，但是在多线程环境下，发生重排序，可能会影响结果，下例中的1和2由于不存在数据依赖关系，则有可能会被重排序，先执行status=true再执行a=2。而此时线程B会顺利到达4处，而线程A中a=2这个操作还未被执行，所以b=a+1的结果也有可能依然等于2。

volatile不适合复合操作 例如，in c++不是一个原子性操作，可以由读取、加、赋值3步组成，所以结果并不能达到30000。

volatile 实现原理：

（3）实现原理

如果对声明了volatile关键字进行写操作，JVM就会向CPU发送一个Lock前缀指令，然后变量所在缓存写回到系统内存。又由于缓存一致性原理，其他处理器通过嗅探发现自己缓存被修改，就会标记这块缓存无效，当处理器要对这数据操作时，会从系统内存重新读取。

## 4.Final

1.**final关键字怎么用？**

（1）修饰类时，表明这个类不能被继承。final类中的所有成员方法都会被隐式地指定为final方法。

（2）修饰方法

1）把方法锁定，以防任何继承类修改它的含义；

2）效率问题。在早期的Java实现版本中，会将final方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升。在最近的Java版本中，不需要使用final方法进行这些优化了。“

（3）修饰变量

对于一个final变量，如果是基本数据类型，其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

## 5.Java中static，final,static final的区别

1. final修饰属性，方法，类，局部变量。static修饰属性，方法，代码段，内部类（静态内部类或嵌套内部类）

(2)final修饰的属性的初始化可以在编译期，也可以在运行期，初始化后不能被改变。static修饰的属性的初始化在编译期（类加载的时候），初始化后能改变。

(3)final修饰的属性跟具体对象有关，在运行期初始化的final属性，不同对象可以有不同的值。static修饰的属性所有对象都只有一个值。

(4)final修饰的属性表明是一个常数（创建后不能被修改）。static修饰的属性强调它们只有一个。

(5)final修饰的方法表示该方法在子类中不能被重写，final修饰的类表示该类不能被继承。static修饰的属性、方法、代码段跟该类的具体对象无关，不创建对象也能调用static修饰的属性、方法等

对于基本类型数据，final会将值变为一个常数（创建后不能被修改）；但是对于对象句柄（亦可称作引用或者指针），final会将句柄变为一个常数（进行声明时，必须将句柄初始化到一个具体的对象。而且不能再将句柄指向另一个对象。但是，对象的本身是可以修改的。这一限制也适用于数组，数组也属于对象，数组本身也是可以修改的。方法参数中的final句柄，意味着在该方法内部，我们不能改变参数句柄指向的实际东西，也就是说在方法内部不能给形参句柄再另外赋值）。

static和“this、super”势不两立，static跟具体对象无关，而this、super正好跟具体对象有关。

static不可以修饰局部变量。

static final和final static没什么区别，一般static写在前面。

static修饰的属性强调它们只有一个，final修饰的属性表明是一个常数（创建后不能被修改）。

static final修饰的属性表示一旦给值，就不可修改，并且可以通过类名访问。也可以修饰方法，表示该方法不能重写，可以在不new对象的情况下调用。

## 6..synchronized volatile 区别

**1**.volatile是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。

2.volatile只能用于变量,synchronized可以修饰方法以及代码块

3.synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用 synchronized 关键字的场景还是更多一些。

4.多线程访问volatile不会发生阻塞，而synchronized可能会发生阻塞

4.volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。

5.volatile主要用于解决变量在多个线程之间的可见性，synchronized解决的是多个线程之间访问资源的同步性。

# 数据库

## 数据库的隔离级别

首先用通俗的语言介绍以下事务（是[数据库管理系统](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%AE%A1%E7%90%86%E7%B3%BB%E7%BB%9F)执行过程中的一个逻辑单位，由一个有限的[数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93)操作序列构成）的特性（ACID）：

原子性（Atomicity）：一个事务中的操作，要么全部成功，要么全部失败，如果失败，就回滚到事务开始前的状态。

一致性（Consistency）：事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。那转账举栗子，A账户和B账户之间相互转账，无论如何操作，A、B账户的总金额都必须是不变的。

隔离性（Isolation）：当多个用户并发的访问数据库时，如果操作同一张表，数据库则为每一个用户都开启一个事务，且事务之间互不干扰，也就是说事务之间的并发是隔离的。再举个栗子，现有两个并发的事务T1和T2，T1要么在T2开始前执行，要么在T2结束后执行，如果T1先执行，那T2就在T1结束后在执行。关于数据的隔离性级别，将在后文讲到。

持久性（Durability）：如果事务一旦被提交，数据库中数据的改变就是永久性的，即使断电或者宕机的情况下，也不会丢失提交的事务操作。

事务的隔离性：多个用户的并发事务访问同一个数据库时，一个用户的事务不应该被其他用户的事务干扰，多个并发事务之间要相互隔离。

如果不考虑隔离性，会发生什么事呢？

1.脏读：一个事务在处理数据的过程中，读取到另一个未提交事务的数据。

2.不可重复读：对于数据库中的某个数据，一个事务范围内的多次查询却返回了不 同的结果，这是由于在查询过程中，数据被另外一个事务修改并提交了。

不可重复读和脏读的区别：脏读读取到的是一个未提交的数据，不可重复读读取到的是前一个事务提交的数据。

而不可重复读在一些情况也并不影响数据的正确性，比如需要多次查询的数据也是要以最后一次查询到的数据为主。

3.幻读：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）。

不可重复读和幻读是初学者不易分清的概念，我也是看了详细的解读才明白的，总的来说，**解决不可重复读的方法是锁行，解决幻读的方式是锁表。**

四种隔离级别解决了上述问题

1.读未提交（Read uncommitted）：这种事务隔离级别下，select语句不加锁。此时，可能读取到不一致的数据，即“读脏 ”。这是并发最高，一致性最差的隔离级别。

2.读已提交（Read committed）：可避免脏读的发生。这是一致性最好的，但并发性最差的隔离级别。在互联网大数据量，高并发量的场景下，几乎 不会使用 上述两种隔离级别。

3.可重复读（Repeatable read）：MySql默认隔离级别。可避免脏读 、不可重复读 的发生。

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

以上四种隔离级别最高的是 Serializable 级别，最低的是 Read uncommitted 级别，当然级别越高，执行效率就越低。像 Serializable 这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。在MySQL数据库中默认的隔离级别为Repeatable read (可重复读) 。

在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读) ；而在 Oracle数据库 中，只支持Serializable (串行化) 级别和 Read committed (读已提交) 这两种级别，其中默认的为 Read committed（读已提交） 级别。

## 存储引擎

不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以获得特定的功能。现在许多不同的数据库管理系统都支持多种不同的数据引擎。

因为在关系数据库中数据的存储是以表的形式存储的，所以存储引擎也可以称为表类型(Table Type，即存储和操作此表的类型)。

### 1.Mysql中的myisam与innodb的区别

1. InooDB支持事务，而MyISAM不支持；
2. InnoDB支持行级锁，而MyISAM支持表级锁；
3. InnoDB支持MVCC，而MyISAM不支持；
4. InnoDB支持外键，而MyISAM不支持；
5. //InnoDB不支持全文索引，而MyISAM支持；
6. InnoDB不能通过直接拷贝表文件的方法拷贝表到另外一台机器， myisam 支持；
7. InnoDB表支持多种行格式，myisam 不支持；
8. InnoDB是索引组织表，myisam 是堆表；

### 2.myisam与innodb select count(\*)哪个更快

myisam更快，因为myisam内部维护了一个计数器，可以直接调取。

### 3.行级锁和表级锁

相对其他数据库而言，MySQL的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。比如，MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；InnoDB存储引擎既支持行级锁（ row-level locking），也支持表级锁，但默认情况下是采用行级锁。

表级锁： 开销小，加锁快；不会出现死锁(因为MyISAM会一次性获得SQL所需的全部锁)；锁定粒度大，发生锁冲突的概率最高,并发度最低。  
行级锁： 开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

考虑上述特点，表级锁使用与并发性不高，以查询为主，少量更新的应用，比如小型的web应用；而行级锁适用于高并发环境下，对事务完整性要求较高的系统，如在线事务处理系统。

## 索引

### 什么是索引

索引(Index)是帮助MySQL高效获取数据的数据结构。我们可以简单理解为：**快速查找排好序的一种数据结构。（**索引其实是一种数据结构，能够帮助我们快速的检索数据库中的数据**）**

**索引具体采用哪种数据结构：**

Mysql索引主要有两种结构：B+Tree索引和Hash索引。我们平常所说的索引，如果没有特别指明，一般都是指B树结构组织的索引(B+Tree索引)。

为什么加索引能优化慢查询？

因为索引其实就是一种优化查询的数据结构，比如Mysql中的索引是用B+树实现的，而B+树就是一种数据结构，可以优化查询速度，可以利用索引快速查找数据，所以能优化查询。

### MySQL主要的索引类型

普通索引：是最基本的索引，它没有任何限制；

唯一索引：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一；

主键索引：是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值；

组合索引：指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合；多列值组成一个索引，专门用于组合搜索，其效率大于索引合并

全文索引：主要用来查找文本中的关键字，而不是直接与索引中的值相比较，mysql中MyISAM支持全文索引而InnoDB不支持；

### 数据库索引有哪些，优缺点？

hash索引和B+树索引  
B+树索引：数据有序，能够进行范围查询

哈希表适合等值查询（因为哈希索引底层是哈希表，哈希表是一种key-value存储数据的结构）

哈希索引没办法利用索引完成排序 ,因此不能进行范围查询

哈希索引不支持多列联合索引的最左匹配规则

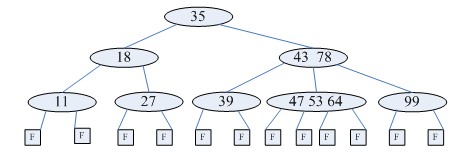
如果有大量重复键值的情况下，哈希索引的效率会很低，因为存在哈希碰撞问题

### 为什么不用二叉查找树作为数据库索引？

二叉查找树，查找到指定数据，效率其实很高logn。查询速度和比较次数都是较小的。

实际上，查询索引操作最耗资源的不在内存中，而是磁盘IO。数据库索引文件有可能很大，关系型数据存储了上亿条数据，索引文件大则上G，不可能全部放入内存中，而是需要的时候换入内存，方式是磁盘页。一般来说树的一个节点就是一个磁盘页。如果使用二叉查找树，那么每个节点存储一个元素，查找到指定元素，需要进行大量的磁盘IO，效率很低。而B树解决了这个问题，通过单一节点包含多个data，大大降低了树的高度，大大减少了磁盘IO次数。

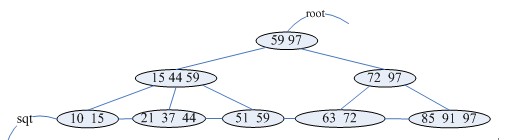
### B树



B 树是为了磁盘或其它存储设备而设计的一种**多叉平衡查找树**。

　　B-树是一种平衡多路查找树，它在文件系统中很有用。一棵m阶B-树（图为4阶B-树），具有下列性质：  
（1）树中每个节点至多有m棵子树；  
（2）若根节点不是叶子节点，则至少有2棵子树；  
（3）除根节点之外的所有非终端节点至少有IMG_260棵子树；  
（4）每个节点中的信息结构为（A0,K1,A1,K2......Kn,An），其中n表示关键字个数，Ki为关键字，Ai为指针；  
（5）所有的叶子节点都出现在同一层次上，且不带任何信息，也是为了保持算法的一致性。

### B+树

（图e）

B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

B+数是B-树的一种变形，它与B-树的差别在于（图e为3阶B+树）：  
（1）有n棵子树的节点含有n个关键字；  
（2）所有的叶子节点包含了全部关键字的信息，及指向这些关键字记录的指针，且叶子节点本身按关键字大小自小到大顺序链接；  
（3）所有非终端节点可以看成是索引部分，节点中仅含有其子树（根节点）中最大（或最小）关键字，所有B+树更像一个索引顺序表；  
（4）对B+树进行查找运算，一是从最小关键字起进行顺序查找，二是从根节点开始，进行随机查找。

1. 那么，如果在firstname、lastname、age这三个列上分别创建单列索引，效果是否和创建一个firstname、lastname、age的多列索引一样呢？答案是否定的，两者完全不同。当我们执行查询的时候，MySQL只能使用一个索引。如果你有三个单列的索引，MySQL会试图选择一个限制最严格的索引。但是，即使是限制最严格的单列索引，它的限制能力也肯定远远低于firstname、lastname、age这三个列上的多列索引。

### 索引的缺点

1. 索引要占用磁盘空间。通常情况下，这个问题不是很突出。但是，如果你创建每一种可能列组合的索引，索引文件体积的增长速度将远远超过数据文件。如果你有一个很大的表，索引文件的大小可能达到操作系统允许的最大文件限制。   
     
   2.对于需要写入数据的操作，比如DELETE、UPDATE以及INSERT操作，索引会降低它们的速度。这是因为MySQL不仅要把改动数据写入数据文件，而且它还要把这些改动写入索引文件。

**9.哪种情况需要创建索引**  
　　1.主键自动建立唯一索引  
　　2.频繁作为查询条件的字段应该创建索引  
　　3.查询中与其他表关联的字段,外键关系建立索引  
　  
　　5.Where条件里用得到的字段适合创建索引  
　　6.单键/组合索引的选择问题,在高并发下倾向创建组合索引  
　　7.查询中排序的字段,排序字段若通过索引去访问将大大提高排序速度  
　　8.查询中统计或者分组字段  
10哪种情况不需要建索引  
　　1.表记录太少(一般生产环境下,三百万条记录性能就可能开始下降,官方说的是五百万到八百万)  
　　2.经常增删改的表  
　　3.某个数据列的值包含许多重复的内容

### 12一个B+树的节点中到底存储多少个元素合适呢？

　　B+树中一个节点为一页或页的倍数最为合适。因为如果一个节点的大小小于1页，那么读取这个节点的时候其实也会读出1页，会造成资源的浪费；如果一个节点的大小大于1页，比如1.2页，那么读取这个节点的时候会读出2页，也会造成资源的浪费；所以为了不造成资源的浪费，最后把一个节点的大小控制在1页、2页、3页、4页等倍数页大小最为合适！

### 11.哪些数据结构能提高查询速度？MySQL为何选择使用B+树？

哈希表、完全平衡二叉搜索树、B树、B+树等等；

哈希表的特点就是可以快速的精确查询，但是不支持范围查询

完全平衡二叉搜索树是有序的，简单的说就是 "左边的小于右边的"，假如我们现在来查找 '周瑜' ，需要查找2次（第一次操作，第二次周瑜），**比哈希表要多一次。而且由于完全平衡二叉搜索树是有序的，所以支持范围查找**。

B树表示的要比完全平衡二叉搜索树要 "矮",原因在于B树中的一个节点可以存储多个元素！

我们可以发现同样的元素，B+树的表示要比B树要 "胖"，原因在于B+树中的非叶子节点会冗余一份在叶子节点中，并且叶子节点之间用指针相连！

### 13.MySQL中B+树的一个节点大小为多大？

　　一页，这里说的 "页" 是MySQL自定义的单位（其实和操作系统类似），MySQL的Innodb引擎中一页的默认大小是16K（如果操作系统中一页大小是4K，那么MySQL中1页 = 操作系统中的4页），这样存取数据的时候都是一页一页的获取索引文件中节点数据的！

### 14一系列问题

1. 系统中某个sql查询比较慢，怎么办？

看条件字段是否加索引，若没加可以加索引

1. mysql存储索引用什么数据结构？

B+树

1. B+树查询时间大约是多少？

和树的高度有关，log(n)

1. 如果用hash存储索引，查询时间？

O(1)

1. hash比B+快，为啥不用hash

如果只选一条 hash快，但实际数据库经常选多条。时候由于B+树索引有序，并且又有链表相连，它的查询效率比hash就快很多了。且数据库中的索引一般是在磁盘上，数据量大的情况可能无法一次装入内存，B+树的设计可以允许数据分批加载，同时树的高度较低，提高查找效率。

1. 树你知道哪些？
2. 前中序遍历，二叉树，二叉搜索树，平衡二叉树，更高级一点红黑树，B tree，B+树
3. 二叉搜索树极端退化成链表，所以用AVL树，红黑树就是平衡树的一种，复杂的定义和规则最后都是为了保证树的平衡性。
4. 为什么要保证树的平衡性

树的查找性能取决于树的高度，让树尽可能平衡，就是为了降低树的高度。

10 .B tree

是一种多路搜索树，每个节点可以拥有多于两个孩子节点，M路的B树最多能拥有M个孩子节点

11.为什么设计成多路

为了进一步降低树的高度

1. 可以设计成无限多路吗

退化成有序数组了

1. B树一般用在文件系统的索引
2. 为什么文件系统的索引用B树而不用红黑树或有序数组

文件系统和数据库的索引存在硬盘上，并且如果数据量大的话，不一定能一次性加载到内存中。

1. 如果一棵树无法一次性加载进内存，怎么查找？

每次加载树的一个节点，然后一步步向下找

如果在内存中，红黑树比B树效率更高，但涉及到硬盘操作，B树更优。

1. B+树在B树基础上进行改进，数据都在叶子结点，同时叶子结点之间还加了指针形成链表。
2. B+树为什么这么设计

Select不一定只选一条，很多时候会选多条。这时B树做局部的中序遍历，可能要跨层访问。而B+树由于所有数据都在叶子结点，不用跨层，同时由于有链表结构，只需要找到首尾，通过链表就能把所有数据取出来了。

### 16为什么B+树比B树更适合做数据库索引？

B+树的磁盘读写代价更低。由于B+树内部的结点并没有指向关键字的指针，所以其内部结点相对B树更小，如果把同一内部结点的关键字存放在同一块磁盘中，那么盘块所能容纳的关键字数量也就较多，一次性读入内存需要查找的关键字也越多，相对IO读写次数也就越低。

B+树的查询效率更加稳定。由于非叶子结点并不是指向文件内容的结点，而只是叶子结点中关键字的索引，所以任何关键字的查找必须走一条从根结点到叶子结点的路，所有关键字查询的路径长度相同，导致每一个数据的查询效率一样。

B+树在查询时，只需要扫一遍叶子结点就可以了，因为分支结点都为索引，没有存文件内容。而B树分支结点中还存在数据，我们需要找到具体的数据，需要进行一次中序遍历按序扫描，所以B+树更适合在区间查询，因此，B+树更适合用于数据库索引。

### 聚簇索引和非聚簇索引

1.首先，我们要认识到聚簇索引和非聚簇索引的划分依据是什么 ？

答案就是 InnoDB 会使用聚簇索索引来保存数据，而非聚簇索引的目的仅仅是加快查询速度

在第一点认知基础上，我们就可以知道

聚簇索引是唯一的，一个 InnoDB 表只有一个聚簇索引，而且一定会有一个聚簇索引，如果不存在，Innodb 存储引擎会自动添加一个

非聚簇所以可以有多个，而且只能由用户自己添加，InnoDB 默认并不会创建任何非聚簇索引。

非聚簇索引中一定包含了聚簇索引的列值，但反过来却不存在。

因此，使用非聚簇索引查询数据一定会用到聚簇索引，但反过来却不存在。

### 聚集索引和非聚集索引

聚集索引和非聚集索引的根本区别是**表记录的排列顺序和与索引的排列顺序是否一致**。

**聚集索引表记录的排列顺序与索引的排列顺序一致**

* 优点是查询速度快，因为一旦具有第一个索引值的纪录被找到，具有连续索引值的记录也一定物理的紧跟其后。
* 缺点是对表进行修改速度较慢，这是为了保持表中的记录的物理顺序与索引的顺序一致，而把记录插入到数据页的相应位置，必须在数据页中进行数据重排， 降低了执行速度。建议使用聚集索引的场合为：  
  a. 此列包含有限数目的不同值；  
  b. 查询的结果返回一个区间的值；  
  c. 查询的结果返回某值相同的大量结果集。

**非聚集索引指定了表中记录的逻辑顺序，但记录的物理顺序和索引的顺序不一致**，聚集索引和非聚集索引都采用了B+树的结构，但非聚集索引的叶子层并不与实际的数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针的方式。  
非聚集索引比聚集索引层次多，添加记录不会引起数据顺序的重组。  
建议使用非聚集索引的场合为：  
a. 此列包含了大量数目不同的值；  
b. 查询的结束返回的是少量的结果集；  
c. order by 子句中使用了该列。

### 什么情况索引会失效：

1. 条件中有or

**注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引**

　　2.对于多列索引，不是使用的第一部分，则不会使用索引

　　3.like查询是以%开头

　　4.如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引

　　5.如果mysql估计使用全表扫描要比使用索引快,则不使用索引

此外，查看索引的使用情况

show status like ‘Handler\_read%’;  
大家可以注意：  
handler\_read\_key:这个值越高越好，越高表示使用索引查询到的次数

handler\_read\_rnd\_next:这个值越高，说明查询低效

### 21 [与红黑树的比较](https://cyc2018.github.io/CS-Notes/#/notes/MySQL?id=_3-%e4%b8%8e%e7%ba%a2%e9%bb%91%e6%a0%91%e7%9a%84%e6%af%94%e8%be%83)

## 4.说一说你能想到的sql语句优化，至少五种

1）避免select \*，将需要查找的字段列出来；

2）使用连接（join）来代替子查询；

3）拆分大的delete或insert语句；

4）使用limit对查询结果的记录进行限定；

5）用 exists 代替 in 是一个好的选择；

6）用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤；

7）不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引尽量避免在where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描；

8）尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描；

9）尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描；

# 设计模式

## 1.单例模式

### 介绍一下单例模式？懒汉式的单例模式如何实现单例？

单例模式作用：保证整个程序在一次运行的过程中，被单例模式声明的类的对象要有且只有一个。

实现单例模式：

1. **构造方法私有，保证无法从外部通过 new 的方式创建对象。**
2. **对外提供获取该类实例的静态方法**
3. **类的内部创建该类的对象，通过第 2 步的静态方法返回**

**特点：**  
1、单例类只能有一个实例。  
2、单例类必须自己创建自己的唯一实例。  
3、单例类必须给所有其他对象提供这一实例。

### 饿汉式单例

//饿汉式单例类.在类初始化时，已经自行实例化

public class Singleton1 {

private Singleton1() {}

private static final Singleton1 single = new Singleton1();

//静态工厂方法

public Singleton1 getInstance() {

return single;

}

}

饿汉式在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以天生是线程安全的。

以空间换时间，故不存在线程安全问题。

单例模式的应用场景可以是网站的计数器，windows的任务管理器。

### 懒汉式单例（线程不安全）

不着急实例化，需要用的时候才初始化

public class Singleton {

private Singleton() {}

private static Singleton singleton = null;

public static Singleton getInstance() {

if (singleton== null) {

singleton= new Singleton();

}

return singleton;

}

}

懒汉模式在方法被调用后才创建对象，以时间换空间，在多线程环境下存在风险。

缺点：

1. 在反射面前没什么用，因为java反射机制是能够实例化构造方法为private的类的；
2. 线程不安全，因为线程访问具有随机性，并发环境可能出现多个singleton实例

### 线程同步式（线程安全）

public class Singleton {

private Singleton() {}

private static Singleton singleton = null;

//为了保证多线程环境下正确访问，给方法加上同步锁synchronized

public static synchronized Singleton getInstance() {

if (singleton == null) {

singleton = new Singleton();

}

return singleton;

}

}

缺点：在方法调用上加了同步，虽然线程安全了，但是每次都有同步，会影响性能，毕竟99%是不需要同步的

### 双重检查锁式

public class Singleton {

private Singleton() {}

private static Singleton singleton = null;

//双重检查锁

public volatile static Singleton getInstance() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

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

优点：线程安全，且确保了只有第一次调用单例时才会做同步，避免了每次都同步的性能损耗；  
缺点：双重锁降低了程序响应速度和性能

#### ①懒汉式单例→②线程同步式→③双重检查锁式

##### ①

在实现单例模式时，如果未考虑多线程的情况，就容易写出下面的错误代码：

【懒汉式单例】

public class Singleton {

private static Singleton singleton;

private Singleton() {

}

public Singleton getInstance() {

if (null == singleton) {

singleton= new Singleton();

}

return singleton;

}

}

在多线程的情况下，这样写可能会导致singleton有多个实例。比如下面这种情况，考虑有两个线程同时调用getInstance()：

| **Time** | **Thread A** | **Thread B** |
| --- | --- | --- |
| T1 | 检查到singleton为空 |  |
| T2 |  | 检查到singleton为空 |
| T3 |  | 初始化对象A |
| T4 |  | 返回对象A |
| T5 | 初始化对象B |  |
| T6 | 返回对象B |  |

可以看到，singleton被实例化了两次并且被不同对象持有。完全违背了单例的初衷。

##### ②synchronized

加锁

出现这种情况，第一反应就是加锁，如下：

public class Singleton {

private static Singleton singleton;

private Singleton() {

}

public synchronized Singleton getInstance() {

if (null == singleton) {

singleton= new Singleton();

}

return singleton;

}

}

这样虽然解决了问题，但是因为用到了synchronized，会导致很大的性能开销，并且加锁其实只需要在第一次初始化的时候用到，之后的调用都没必要再进行加锁。

##### ③synchronized +volatile

**双重检查锁**

双重检查锁（double checked locking）是对上述问题的一种优化。**先判断对象是否已经被初始化，再决定要不要加锁。**

错误的双重检查锁

public class Singleton {

private static Singleton singleton;

private Singleton() {

}

public Singleton getInstance() {

if (null == singleton) {

synchronized (Singleton.class) {

if (null == singleton) {

singleton= new Singleton(); // error

}

}

}

return singleton;

}

}

如果这样写，运行顺序就成了：

1. 检查变量是否被初始化(不去获得锁)，如果已被初始化则立即返回。
2. 获取锁。
3. 再次检查变量是否已经被初始化，如果还没被初始化就初始化一个对象。

执行双重检查是因为，如果**多个线程同时了通过了第一次检查，并且其中一个线程首先通过了第二次检查并实例化了对象，那么剩余通过了第一次检查的线程就不会再去实例化对象。**

这样，除了初始化的时候会出现加锁的情况，后续的所有调用都会避免加锁而直接返回，**解决了性能消耗的问题**。

**隐患**

上述写法看似解决了问题，但是有个很大的隐患。实例化对象的那行代码（标记为error的那行），实际上可以分解成以下三个步骤：

1. 分配内存空间
2. 初始化对象
3. 将对象指向刚分配的内存空间

但是有些编译器为了性能的原因，可能会将第二步和第三步进行**重排序**，顺序就成了：

1. 分配内存空间
2. 将对象指向刚分配的内存空间
3. 初始化对象

现在考虑重排序后，两个线程发生了以下调用：

| **Time** | **Thread A** | **Thread B** |
| --- | --- | --- |
| T1 | 检查到singleton为空 |  |
| T2 | 获取锁 |  |
| T3 | 再次检查到singleton为空 |  |
| T4 | 为singleton分配内存空间 |  |
| T5 | 将singleton指向内存空间 |  |
| T6 |  | 检查到singleton不为空 |
| T7 |  | 访问singleton（此时对象还未完成初始化） |
| T8 | 初始化singleton |  |

在这种情况下，T7时刻线程B对singleton的访问，访问的是一个**初始化未完成**的对象。

**正确的双重检查锁**

public class Singleton {

private volatile static Singleton singleton;

private Singleton() {

}

public Singleton getInstance() {

if (null == singleton) {

synchronized (Singleton.class) {

if (null == singleton) {

singleton= new Singleton();

}

}

}

return singleton;

}

}

为了解决上述问题，需要在singleton前加入关键字volatile。使用了volatile关键字后，重排序被禁止，所有的写（write）操作都将发生在读（read）操作之前。

至此，双重检查锁就可以完美工作了。

### 静态内部类式

public class Singleton {

private Singleton() {}

//内部类的初始化需要依赖主类，需要先等主类实例化之后，内部类才能开始实例化

private static class LazyHolder {

//这里加final是为了防止内部将这个属性覆盖掉

private static final Singleton INSTANCE = new Singleton();

}

//这里加final是为了防止子类重写父类

public static final Singleton getInstance() {

return LazyHolder.INSTANCE;

}

}

优点：利用了classloader机制来保证初始化instance时只有一个线程，线程安全且没有性能损耗。

外部类加载时并不需要立即加载内部类，内部类不被加载则不去初始化INSTANCE，故而不占内存。即当SingleTon第一次被加载时，并不需要去加载LazyHolder ，只有当getInstance()方法第一次被调用时，才会去初始化INSTANCE,第一次调用getInstance()方法会导致虚拟机加载LazyHolder 类，这种方法不仅能确保线程安全，也能保证单例的唯一性，同时也延迟了单例的实例化。

**性能方面：**

getInstance()方法，调用的是LazyHolder .INSTANCE，取的是LazyHolder 里的INSTANCE对象，跟上面那个DCL方法不同的是，getInstance()方法并没有多次去new对象，故不管多少个线程去调用getInstance()方法，取的都是同一个INSTANCE对象，而不用去重新创建。当getInstance()方法被调用时，LazyHolder 才在SingleTon的运行时常量池里，把符号引用替换为直接引用，这时静态对象INSTANCE也真正被创建，然后再被getInstance()方法返回出去，这点同饿汉模式。**线程安全方面：**

虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果在一个类的<clinit>()方法中有耗时很长的操作，就可能造成多个进程阻塞(需要注意的是，其他线程虽然会被阻塞，但如果执行<clinit>()方法后，其他线程唤醒之后不会再次进入<clinit>()方法。同一个加载器下，一个类型只会初始化一次。)，在实际应用中，这种阻塞往往是很隐蔽的。

缺点：传参的问题。由于是静态内部类的形式去创建单例的，故外部无法传递参数进去，例如Context这种参数。

所以，我们创建单例时，可以在静态内部类与双重检查锁式里自己斟酌。

### 枚举单例

public enum SingleTon{

INSTANCE;

public void method(){

//TODO

}

}

枚举在java中与普通类一样，都能拥有字段与方法，而且枚举实例创建是线程安全的，在任何情况下，它都是一个单例。我们可直接以

SingleTon.INSTANCE

的方式调用。

<https://blog.csdn.net/mnb65482/article/details/80458571>

## 2.适配器模式

### 1.什么是适配器模式？

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。什么是适配器？我们手机充电是5V，而插座是220V怎么办，用充电插头转换呗，充电插头就是适配器，那么现在2个接口分别是代入手机充电口和插座口，适配器模式就是代表插头的中间件。

### 2.为什么要使用类适配器模式？优缺点？使用场景？

将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的，使用适配器模式让他满足。

**优点：(1)**可以让任何两个没有关联的类一起运行 (2)提高了类的复用 (3)增加了类的透明度(4)灵活性好。

**缺点：**(1)过多地使用适配器，会让系统非常零乱

(2)由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

**使用场景：**(1)JAVA JDK 1.1 提供了 Enumeration 接口，而在 1.2 中提供了 Iterator 接口，想要使用 1.2 的 JDK，则要将以前系统的 Enumeration 接口转化为 Iterator 接口，这时就需要适配器模式

1. 在 LINUX 上运行 WINDOWS 程序
2. JAVA 中的 jdbc。

### 类适配器模式

定义一个需要被适配的Adaptee类：

public class Adaptee {  
public void specificRequest(){  
System.out.println("将需要被适配的方法");  
}  
}

定义一个目标接口Target：

public interface Target {  
public void request();  
}

定义适配器Adapter类， 因为是类适配器模式，所以我们需要继承Adaptee类：

public class Adapter extends Adaptee implements Target {  
@Override  
public void request() {  
specificRequest();  
}  
@Override  
public void specificRequest() {  
//这里可以加下其他操作  
super.specificRequest();  
//这里可以加下其他操作  
}  
}

客户端测试：

public static void main(String args []){  
Target adapter=new Adapter();  
adapter.request();  
}

输出结果 ：

将需要被适配的方法

这里我们可以看到适配器（Adapter）继承了适配者（Adaptee），然后实现了目标接口（Target）。这样使目标接口跟适配者的接口就关联起来了，客户端通过调用适配器的方法，从而达到调用了适配者所被适配的方法。

### 对象适配器模式

对象适配器模式跟类适配器模式所不同的就是适配器（Adapter）类是基于继承的，而对象适配器是基于组合的。其他的Target、Adaptee不变。

定义适配器类。

public class Adapter implements Target {  
private Adaptee adaptee =new Adaptee();  
@Override  
public void request() {  
adaptee.specificRequest();  
}  
}

客户端测试：

public static void main(String args []){  
Target adapter=new Adapter();  
adapter.request();  
}

输出结果 ：

将需要被适配的方法

一般常用的就是对象适配器模式，很少用类适配器。

### 双向适配器模式

还有一种双向适配器模式，就是（Adapter）适配器包含对目标类（Target）和适配者类（Adaptee）两个引用。目标类可以通过适配器（Adapter）调用适配者（Adaptee中的方法， 适配者类也可以通过适配器调用目标类的方法。

适配器（Adapter）类一般写法如下：

public class Adapter implements Target , Adaptee {

//同时对适配者和目标类的引用

private Target target;  
private Adaptee adaptee;  
public Adapter(Target target) {  
this.target = target;  
}  
public Adapter(Adaptee adaptee) {  
this.adaptee = adaptee;  
}  
@Override  
public void request() {  
adaptee.specificRequest();  
}  
@Override  
public void specificRequest() {  
target.request();  
}  
}

在实际的开发中很少使用双向适配器。

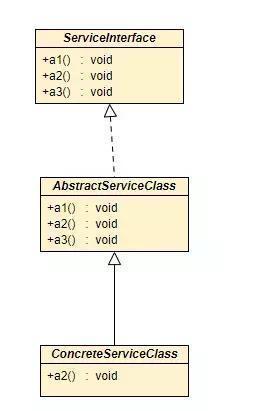
### 缺省适配器

缺省适配器模式是适配器模式一种变体。

定义：当不需要实现一个接口所提供的所有方法时，可以设计一个抽象类实现该接口，并为每个接口提供一个默认实现，那么该抽象类的子类可以选择性覆盖父类中的方法，它适用于不想使用一个接口中的所有方法的情况，又称为单接口适配器模式。

简单的理解就是当适配者（Adaptee）有大量的方法时，那么每个适配器（Adapter）都要去实现接口中的这些方法，这样就感觉太费劲。所以这种情况下，可以考虑实现个默认的适配器，然后根据实际目标角色接口的类集成这个默认适配器，然后选择性的实现默认适配器中的一些方法。

缺省适配器模式结构如下图：



\*\*ServiceInterface(适配者接口) \*\*：是一个接口，里面包含大量的方法。

\*\*AbstractServiceClass(缺省适配器类) \*\*：实现了ServiceInterface中声明的方法， 通常定义为抽象类。

\*\*ConcreteServiceClass(具体业务类) \*\*：它是缺省适配器的子类，在没有引用适配器之前，它需要实现适配者（ServiceInterface）所有的方法， 有了缺省适配器类后，就可以有选择性的覆盖适配器类中的方法。

### 总结

适配器模式总共包括四种：类适配器模式、对象适配器模式、双向适配器模式、缺省适配器模式。

* 类适配器模式 ：适配器（Adapter）跟适配者（Adaptee）是一种继承关系。
* 对象适配器模式：适配器（Adapter）跟适配者（Adaptee）是一种组合关系。
* 双向适配器模式 ：双向适配器是对象适配器模式的一种变体， 该模式的适配器（Adapter）不仅引用了适配者（Adaptee）,还对目标类（Target）引用。所以这种模式目标类和适配者之间互相适配。
* 缺省适配器模式：当适配者接口中出现大量的方法时，就可以考虑这种模式。

#### 1、优点

类适配器模式和对象适配器模式都具有的优点：

* 将目标类（Target）和 适配者类（Adaptee）解耦， 引用一个适配器类（Adapter）就可以对适配者进行重用。
* 增加了类的透明性和复用性，将具体的业务实现过程封装在适配者类中，对客户端来而言是透明的，而且提高了适配者的复用性，同一个适配者类可以在多个不同的系统中复用。

对象适配器模式优点：

* 一个对象适配器可以把多个适配者Adaptee适配到同一目标。
* 可以适配一个适配者的子类，由于适配器和适配者之间是组合关系，根据“里氏代换原则”，适配者的子类也可以通过该适配器进行适配。

#### 2、缺点

类适配器模式缺点：

* 因为Java不支持多重继承，一次最多适配一个适配者类，不能同时适配多个适配者。
* 适配者类不能为最终类，即适配者类不能用final关键字修饰。
* 类适配器模式中的目标类target只能为接口，不能为类。

对象适配器模式缺点：

与类适配器模式相比，要在适配器中置换适配者Adaptee类的某些方法比较麻烦（因为类适配器模式是基于继承的，可以重写适配者类的方法，对象适配器模式则不能），如果一定要置换掉适配者类的方法，可以先做一个适配者的子类，将适配者类的方法置换掉，然后再把适配者的子类当作真正的适配者进行适配，实现过程较为复杂。

#### 3、适用场景

* 想创建一个可以重用的类，用于与一些彼此之间没有太大的关联的一些类，包括一些可能再将来引进的类一起工作。

## 3.工厂模式

简单工厂：把对象的创建放到一个工厂类中，通过参数来创建不同的对象。在创建一个对象时不向客户暴露内部细节，并提供一个创建对象的通用接口。 简单工厂把实例化的操作单独放到一个类中，这个类就成为简单工厂类，让简单工厂类来决定应该用哪个具体子类来实例化。

这样做能把客户类和具体子类的实现解耦，客户类不再需要知道有哪些子类以及应当实例化哪个子类。客户类往往有多个，如果不使用简单工厂，那么所有的客户类都要知道所有子类的细节。而且一旦子类发生改变，例如增加子类，那么所有的客户类都要进行修改。

工厂方法：每种产品由一种工厂来创建。（不这样会有什么问题？）   
抽象工厂：感觉只是工厂方法的复杂化，产品系列复杂化的工厂方法。

## 4.代理模式

提供了间接对目标对象进行访问的方式;即通过代理对象访问目标对象.

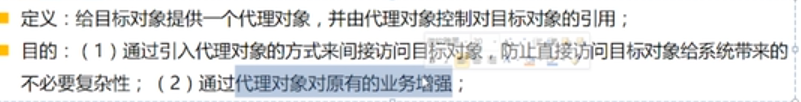
好处：可以在目标对象实现的功能上,增加额外的功能补充,即扩展目标对象的功能.这就符合了设计模式的开闭原则，即在对既有代码不改动的情况下进行功能的扩展。

#### 1.静态代理

优点：可以做到在不修改目标对象的功能前提下,对目标功能扩展.  
缺点:因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类,类太多.同时,一旦接口增加方法,目标对象与代理对象都要维护.而动态代理方式可以解决上面的问题

#### 动态代理

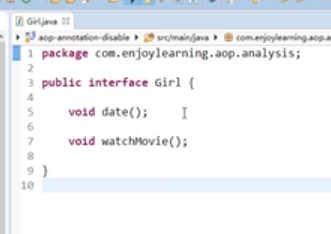
##### 2.1整体代码框架



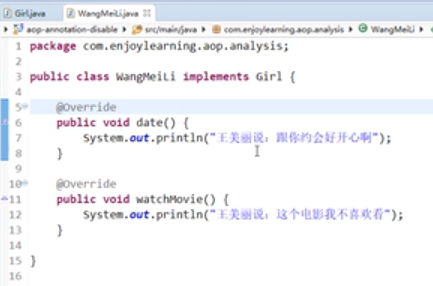
Proxy:调度器，根据情况去选择不同的连接

invocationHandler:增强 invoke

①

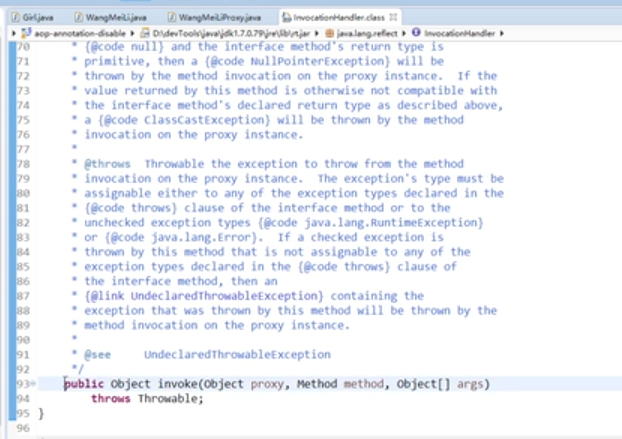
接口

②



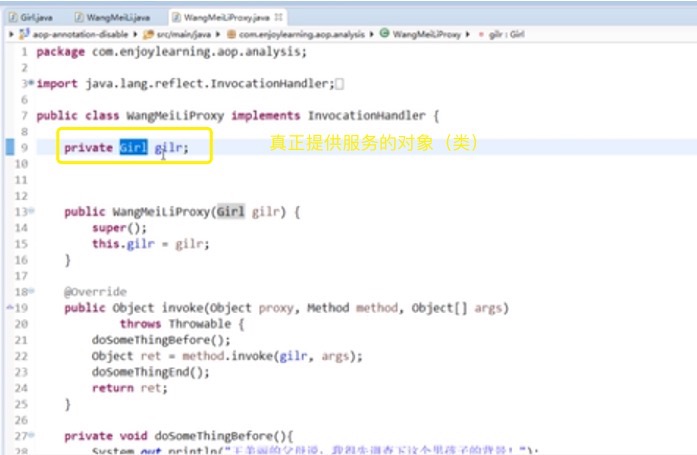
接口girl的实现类

③



invocationhandler只有invoke一个方法

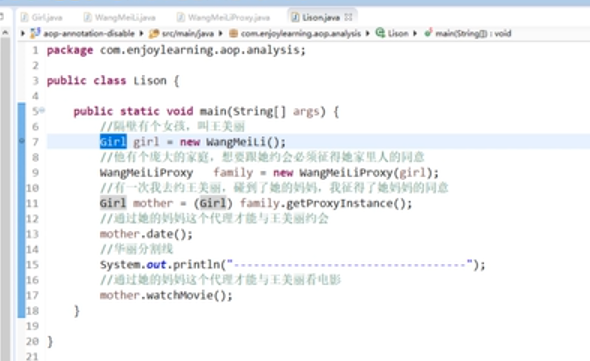
④



增强部分的代码：



newProxyInstance(类加载器，指定接口，invocationhandler方法本身）

⑤

调用

##### 2.2动态代理

动态代理的主要特点就是能够在程序运行时JVM才为被代理对象生成代理对象。

常说的动态代理也叫做JDK代理也是一种接口代理，JDK中生成代理对象的代理类就是Proxy，所在包是java.lang.reflect

##### 2.3JDK动态代理

每一个动态代理类都必须要实现InvocationHandler这个接口，并且每个代理类的实例都关联到了一个handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的 invoke 方法来进行调用。我们来看看InvocationHandler这个接口的唯一一个方法 invoke 方法：

Object invoke(Object proxy, Method method, Object[] args) throws Throwable

proxy: 指代我们所代理的那个真实对象

method: 指代的是我们所要调用真实对象的某个方法的Method对象

args: 指代的是调用真实对象某个方法时接受的参数

<https://blog.csdn.net/chushoufengli/article/details/88583897>

<https://www.bilibili.com/video/BV1Dt41187wj?from=search&seid=4206312863759547324>

##### 2.4动态代理和cglib的区别（删掉）

java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP   
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

如何强制使用CGLIB实现AOP？  
 （1）添加CGLIB库，SPRING\_HOME/cglib/\*.jar  
 （2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>  
  
JDK动态代理和CGLIB字节码生成的区别？  
 （1）JDK动态代理只能对实现了接口的类生成代理，而不能针对类  
 （2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法  
   因为是继承，所以该类或方法最好不要声明成final

# 算法

## 剑指offer算法

可以先说出暴力解法，然后和面试官交流不断优化，是否可以修改原始数据？是否 使用递归?是否可以使用辅助内存？

3.public class Solution {

public boolean duplicate(int numbers[],int length,int [] duplication) {

if(length <= 0|| numbers.length!=length||numbers==null) return false;

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

if(numbers[i]<0||numbers[i]>length-1)

return false;

}

int i = 0;

while(i < length){

if(numbers[i] != i){

if(numbers[i] == numbers[numbers[i]]){

duplication[0] = numbers[i];

return true;

}

int temp;

temp = numbers[i];

numbers[i] = numbers[temp];

numbers[temp] = temp;

}

else i++;

}

return false;

}

}

4.public class Solution {

public boolean Find(int target, int [][] array) {

if(array == null || array.length == 0 || (array.length == 1&&array[0].length == 0))

return false;

int row = array.length;

int col = array[0].length;

int i = 0;

while(i!=row && col!=0){

if(array[i][col-1] == target) return true;

else if(array[i][col-1] > target) col--;

else i++;

}

return false;

}

}

5.public class Solution {

public String replaceSpace(StringBuffer str) {

if(str == null) return null;

int length = str.length();

int spaceNum = 0;

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

if(str.charAt(i)==' ') spaceNum++;

}

int newLength = length + 2 \* spaceNum;

str.setLength(newLength);

for(int i=length-1; i >= 0;i--){

if(str.charAt(i)!=' '){

str.setCharAt(--newLength,str.charAt(i));

}else{

str.setCharAt(--newLength,'0');

str.setCharAt(--newLength,'2');

str.setCharAt(--newLength,'%');

}

}

return str.toString();

}

}

6.

import java.util.ArrayList;

import java.util.Stack;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

ArrayList<Integer> res = new ArrayList<>();

if(listNode == null) return res;

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

ListNode cur = listNode;

while(cur != null){

stack.push(cur.val);

cur = cur.next;

}

while(!stack.isEmpty()){

res.add(stack.pop());

}

return res;

}

}

7. public TreeNode reConstructBinaryTree(int [] pre,int [] in) {

TreeNode root=reConstructBinaryTree(pre,0,pre.length-1,in,0,in.length-1);

return root;

}

//前序遍历{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}

private TreeNode reConstructBinaryTree(int [] pre,int startPre,int endPre,int [] in,int startIn,int endIn) {

if(startPre>endPre||startIn>endIn)

return null;

TreeNode root=new TreeNode(pre[startPre]);

for(int i=startIn;i<=endIn;i++)

if(in[i]==pre[startPre]){

root.left=reConstructBinaryTree(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);

root.right=reConstructBinaryTree(pre,i-startIn+startPre+1,endPre,in,i+1,endIn);

break;

}

return root;

}

8.

public class Solution {

public TreeLinkNode GetNext(TreeLinkNode pNode)

{

if(pNode == null) return null;

if(pNode.right != null){

pNode = pNode.right;

while(pNode.left != null) pNode = pNode.left;

return pNode;

}

while(pNode.next!=null)

{

if(pNode.next.left==pNode)return pNode.next;

pNode=pNode.next;

}

return null;

}

}

9.import java.util.Stack;

public class Solution {

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

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

public void push(int node) {

stack1.push(node);

}

public int pop() {

if(stack1.empty()&&stack2.empty()){

throw new RuntimeException("Queue is empty!");

}

if(stack2.isEmpty()){

while(!stack1.isEmpty()){

stack2.push(stack1.pop());

}

}

return stack2.pop();

}

}

10.

public class Solution {

public int Fibonacci(int n) {

if( n == 0 || n == 1) return n;

int first = 0;

int two = 1;

int res = 0;

for(int i = 2;i <=n;i++){

res = first + two;

first = two;

two = res;

}

return res;

}

}

11.public class Solution {

     public int minNumberInRotateArray(int[] array) {

         if (array == null || array.length <= 0)  return 0;

        int low = 0;

        int high = array.length - 1;

         int mid = (low + high) / 2;

         //升序数组

         if (array[low] < array[high]) return array[low];

         //中间数字与首尾数字相等

         if (array[mid] == array[high] && array[mid] == array[low]) {

             for (int i = 1; i <= high; i++) {

                 if (array[i] < array[i - 1] return array[i];

             }

             return array[low];

         }

         //正常情况

         while (low < high) {

             if (high - low == 1)  break;

             mid = (low + high) / 2;

             if (array[mid] <= array[high]) high = mid;

                          if (array[mid] > array[low]) low = mid;

                   }

         return array[high]; // 别错写成了return high; !!

    }

1 2.public class Solution {

    public boolean hasPath(char[] matrix, int rows, int cols, char[] str) {

        if(matrix==null || matrix.length==0 || str==null || str.length==0 || matrix.length!=rows\*cols || rows<=0 || cols<=0 || rows\*cols < str.length)  return false ;

        boolean[] visited = new boolean[rows\*cols] ;

        int[] pathLength = {0} ;

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

            for(int j=0 ; j<=cols-1 ; j++) {

                if(hasPathCore(matrix, rows, cols, str, i, j, visited, pathLength)) return true ;

            }

        }

        return false ;

    }

    public boolean hasPathCore(char[] matrix, int rows, int cols, char[] str, int row, int col, boolean[] visited, int[] pathLength) {

        boolean flag = false ;

        if(row>=0 && row<rows && col>=0 && col<cols && !visited[row\*cols+col] && matrix[row\*cols+col]==str[pathLength[0]]) {

            pathLength[0]++ ;

            visited[row\*cols+col] = true ;

            if(pathLength[0]==str.length) { return true ; }

            flag = hasPathCore(matrix, rows, cols, str, row, col+1, visited, pathLength)  ||

                   hasPathCore(matrix, rows, cols, str, row+1, col, visited, pathLength)  ||

                   hasPathCore(matrix, rows, cols, str, row, col-1, visited, pathLength)  ||

                   hasPathCore(matrix, rows, cols, str, row-1, col, visited, pathLength) ;

            if(!flag) {

                pathLength[0]-- ;

                visited[row\*cols+col] = false ;

            }

        }

        return flag ;

    }

}

13.public class Solution {

public int movingCount(int threshold, int rows, int cols)

{

boolean[] visited=new boolean[rows\*cols];

return movingCountCore(threshold, rows, cols, 0,0,visited);

}

private int movingCountCore(int threshold, int rows, int cols,

int row,int col,boolean[] visited) {

if(row<0||row>=rows||col<0||col>=cols) return 0;

int i=row\*cols+col;

if(visited[i]||!checkSum(threshold,row,col)) return 0;

visited[i]=true;

return 1+movingCountCore(threshold, rows, cols,row,col+1,visited)

+movingCountCore(threshold, rows, cols,row,col-1,visited)

+movingCountCore(threshold, rows, cols,row+1,col,visited)

+movingCountCore(threshold, rows, cols,row-1,col,visited);

}

private boolean checkSum(int threshold, int row, int col) {

int sum=0;

while(row!=0){

sum+=row%10;

row=row/10;

}

while(col!=0){

sum+=col%10;

col=col/10;

}

if(sum>threshold) return false;

return true;

}

}

14.public int maxProductAfterCutting(int len){

if(len<2) return 0;

if(len==2) return 1;

if(len==3) return 2;

//存储长度从 0-len 的最大结果

int[] result=new int[len+1];

result[0]=0;

result[1]=1;

result[2]=2;

result[3]=3;

//自底向上开始求解

int max=0;

for(int i=4;i<=len;i++){

max=0;

for(int j=1;j<=i/2;j++){

int tempResult=result[j]\*result[i-j];

if(max<tempResult) max=tempResult;

}

result[i]=max;

}

max=result[len];

return max;

}

15.public class Solution {

public int NumberOf1(int n) {

int count = 0;

while(n != 0){

count++;

n = n & ( n - 1);

}

return count;

}

}

16.public class Solution {

    public double Power(double base, int exponent) {

        if(exponent==0 && base != 0)

            return 1;

        if(exponent==1)

            return base;

        if(base == 0 && exponent <= 0){

            throw new RuntimeException();

        }

        if(base ==0 && exponent>0){

            return 0;

        }

        int n= exponent;

        if(exponent<0){

            n = -exponent;

        }

        double  result=Power(base,n>>1);

        result\*=result;

        if((n&1)==1)

            result\*=base;

        if(exponent<0)

            result=1/result;

        return result;

  }

}

17. public static void Print1ToMaxOfNDigits\_2(int n){

if(n <= 0) return;

StringBuffer number = new StringBuffer();

for(int i = 0; i < n; i++) number.append('0');

while(!Increment(number)) PrintNumber(number);

}

public static boolean Increment(StringBuffer s){

boolean isOverflow = false;

int nTakeOver = 0;

int nLength = s.length();

for(int i = nLength - 1; i >= 0; i--){

int nSum = s.charAt(i) - '0' + nTakeOver;

if( i == nLength - 1) nSum++;

if(nSum >= 10){

if(i == 0) isOverflow = true;

else{

nSum -= 10;

nTakeOver = 1;

s.setCharAt(i, (char) ('0' + nSum));

}

}

else{

s.setCharAt(i, (char) ('0' + nSum));

break;

}

}

return isOverflow;

}

public static void PrintNumber(StringBuffer s){

boolean isBeginning0 = true;

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

if(isBeginning0 && s.charAt(i) != '0') isBeginning0 = false;

if(!isBeginning0) System.out.print(s.charAt(i));

}

System.out.println();

}

18.1public class Demo1 {

public void deleteNode(Node head, Node toBeDeleted) {

// 1.如果链表为空，或者要删除的节点为空，返回

if (head == null || toBeDeleted == null)

return;

// 如果要删除的节点不是尾节点

if (toBeDeleted.next != null) {

// 要建一个临时节点保存toBeDeleted的next节点

Node nextNode = toBeDeleted.next;

toBeDeleted = nextNode;

nextNode = null;

}

// 删除只有一个节点的链表的头结点

else if (head == toBeDeleted) {

head = null;

}

// 删除含有多个节点的尾节点

else {

Node preNode = head;

while (preNode.next != toBeDeleted)

preNode = preNode.next;

preNode.next = null;

}

}

}

18.2 public ListNode deleteDeplication(ListNode pHead){

if (pHead == null)

return null;

//注意备用头结点，头结点可能被删除

ListNode first = new ListNode(-1);

first.next = pHead;

ListNode p = pHead;

//前节点

ListNode preNode = first;

while (p != null && p.next != null){

if (p.val == p.next.val){ //两节点相等

int val = p.val; //记录val用于判断后面节点是否重复

while(p != null && p.val == val){ //这一步用于跳过相等的节点，用于删除

p = p.next;

}

preNode.next = p; //删除操作，前节点的next直接等于现在的节点，把中间的节点直接跨过

}else {

preNode = p;

p = p.next;

}

}

return first.next;

}

}

19.public class Solution {

    public boolean match(char[] str, char[] pattern) {

    if (str == null || pattern == null) {

        return false;

    }

    int strIndex = 0;

    int patternIndex = 0;

    return matchCore(str, strIndex, pattern, patternIndex);

}

public boolean matchCore(char[] str, int strIndex, char[] pattern, int patternIndex) {

    //有效性检验：str到尾，pattern到尾，匹配成功

    if (strIndex == str.length && patternIndex == pattern.length) {

        return true;

    }

    //pattern先到尾，匹配失败

    if (strIndex != str.length && patternIndex == pattern.length) {

        return false;

    }

    //模式第2个是\*，且字符串第1个跟模式第1个匹配,分3种匹配模式；如不匹配，模式后移2位

    if (patternIndex + 1 < pattern.length && pattern[patternIndex + 1] == '\*') {

        if ((strIndex != str.length && pattern[patternIndex] == str[strIndex]) || (pattern[patternIndex] == '.' && strIndex != str.length)) {

            return matchCore(str, strIndex, pattern, patternIndex + 2)//模式后移2，视为x\*匹配0个字符

                    || matchCore(str, strIndex + 1, pattern, patternIndex + 2)//视为模式匹配1个字符

                    || matchCore(str, strIndex + 1, pattern, patternIndex);//\*匹配1个，再匹配str中的下一个

        } else {

            return matchCore(str, strIndex, pattern, patternIndex + 2);

        }

    }

    //模式第2个不是\*，且字符串第1个跟模式第1个匹配，则都后移1位，否则直接返回false

    if ((strIndex != str.length && pattern[patternIndex] == str[strIndex]) || (pattern[patternIndex] == '.' && strIndex != str.length)) {

        return matchCore(str, strIndex + 1, pattern, patternIndex + 1);

    }

    return false;

    }

}



//剑指offer第一版题 同于牛客

//已AC

public class Solution {

public void reOrderArray(int []array){

int[] a=new int[array.length];

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

a[z]=array[z];

}

int j=0;

for(int i=0;i<array.length;i++) {//放入奇数

if(array[i]%2==1) {

a[j]=array[i];

j++;

}

}

for(int k=0;k<array.length;k++) {//放入偶数

if(array[k]%2==0) {

a[j]=array[k];

j++;

}

}

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

array[z]=a[z];

}

}

}

//剑指offer第二版题，不同于牛客

public static void reorderOddEven1(int []number){

int len = number.length;

if(number == null || len == 0){

return;

}

int begin = 0;

int end = len -1;

while(begin < end){

while((begin < end) && isOdd(number[begin])){

++begin;

}

while((begin < end) && !isOdd(number[end])){

--end;

}

if(begin < end){

int temp = number[begin];

number[begin] = number[end];

number[end] = temp;

}

}

}

public static boolean isOdd(int num) {

return (num & 1) == 1;

}

22.

public class Solution {

public ListNode FindKthToTail(ListNode head,int k) {

if(head == null || k < 1) return null;

ListNode pFront = head;

ListNode pBehind = head;

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

if(pFront.next !=null){

pFront = pFront.next;

}else return null;

}

while(pFront.next!=null){

pFront = pFront.next;

pBehind = pBehind.next;

}

return pBehind;

}

}

23.

public class Solution {

ListNode EntryNodeOfLoop(ListNode pHead){

if(pHead == null || pHead.next == null)

return null;

ListNode p1 = pHead;

ListNode p2 = pHead;

while(p2 != null && p2.next != null ){

p1 = p1.next;

p2 = p2.next.next;

if(p1 == p2){

p2 = pHead;

while(p1 != p2){

p1 = p1.next;

p2 = p2.next;

}

return p1;

}

}

return null;

}

}

24.

public class Solution {

public ListNode ReverseList(ListNode head) {

if(head==null) return null;

ListNode pre = null;

ListNode next = null;

while(head!=null){

next = head.next;

head.next = pre;

pre = head;

head = next;

}

return pre;

}

}

25.

public class Solution {

public ListNode Merge(ListNode list1,ListNode list2) {

ListNode dummy = new ListNode(0);

ListNode cur = dummy;

ListNode p1 = list1;

ListNode p2 = list2;

while((p1 != null) && (p2 != null)){

if(p1.val <= p2.val ){

cur.next = p1;

p1 = p1.next;

}else{

cur.next = p2;

p2 = p2.next;

}

cur = cur.next;

}

if(p1 == null) cur.next = p2;

else cur.next = p1;

return dummy.next;

}

}

26.

public class Solution {

public boolean HasSubtree(TreeNode root1,TreeNode root2) {

if(root1 == null || root2 == null){return false;}

return isSame(root1,root2) || HasSubtree(root1.left,root2) || HasSubtree(root1.right,root2);

}

public boolean isSame(TreeNode root1,TreeNode root2) {

if(root2 == null){return true;}

if(root1 == null){return false;}

if(root1.val == root2.val){

return isSame(root1.left,root2.left) && isSame(root1.right,root2.right);

}else{

return false;

}

}

}

27.

public class Solution {

public void Mirror(TreeNode root) {

TreeNode temp;

if(root!=null){

temp = root.left;

root.left = root.right;

root.right = temp;

Mirror(root.left);

Mirror(root.right);

}

}

}

28.

public class Solution {

boolean isSymmetrical(TreeNode pRoot)

{

return isSymmetrical(pRoot,pRoot);

}

boolean isSymmetrical(TreeNode pRoot1,TreeNode pRoot2){

if(pRoot1 == null && pRoot2 == null)

return true;

if(pRoot1 == null || pRoot2 == null)

return false;

if(pRoot1.val!=pRoot2.val)

return false;

return isSymmetrical(pRoot1.left,pRoot2.right)&& isSymmetrical(pRoot1.right,pRoot2.left);

}

}

29.import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> printMatrix(int [][] matrix) {

ArrayList<Integer> result = new ArrayList<Integer>();

if(matrix == null && matrix.length==0) return result;

int start = 0;

int rows = matrix.length;

int cols = matrix[0].length;

while(rows > start \* 2 && cols > start \* 2){

printMatrixInCircle(matrix,rows,cols,start,result);

start++;

}

return result;

}

private void printMatrixInCircle(int [][] matrix , int rows, int cols, int start,ArrayList<Integer> result ){

int endX = cols - start - 1;

int endY = rows - start - 1;

for(int i = start ; i <= endX ; i++){

result.add(matrix[start][i]);

}

if(endY > start){

for(int i = start+1 ; i <= endY ; i++){

result.add(matrix[i][endX]);

}

}

if(endX > start && endY > start){

for(int i = endX-1 ; i >=start ; i--){

result.add(matrix[endY][i]);

}

}

if(endX > start && endY > start + 1){

for(int i = endY-1 ; i >=start+1 ; i--){

result.add(matrix[i][start]);

}

}

}

}

30.import java.util.Stack;

public class Solution {

Stack<Integer> data\_stack = new Stack<Integer>();

Stack<Integer> min\_stack = new Stack<Integer>();

public void push(int node) {

data\_stack.push(node);

if(min\_stack.empty()||node<min\_stack.peek()){

min\_stack.push(node);

}else{

min\_stack.push(min\_stack.peek());

}

}

public void pop() {

data\_stack.pop();

min\_stack.pop();

}

public int top() {

return data\_stack.peek();

}

public int min() {

if(!min\_stack.empty()){

return min\_stack.peek();

}else

throw new RuntimeException("Stack is empty!");

}

}

31.import java.util.ArrayList;

import java.util.Stack;

public class Solution {

public boolean IsPopOrder(int [] pushA,int [] popA) {

if(pushA==null || popA == null) return false;

if(pushA.length != popA.length) return false;

if(pushA.length == 0 || popA.length == 0) return true;

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

int i = 0;

int j = 0;

stack.push(pushA[i++]);

while(j < popA.length){

if(popA[j]!= stack.peek()){

if(i< pushA.length){

stack.push(pushA[i++]);

}else return false;

}else{

j++;

stack.pop();

}

}

return true;

}

}

32.import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

public class Solution {

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {

ArrayList<Integer> result = new ArrayList<Integer>();

if (root==null){

return result;

}

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

while (!queue.isEmpty()){

TreeNode node= queue.poll();

result.add(node.val);

if (node.left!=null){

queue.add(node.left);

}

if (node.right!=null){

queue.add(node.right);

}

}

return result;

}

}

33.public class Solution {

public boolean VerifySquenceOfBST(int [] sequence) {

if(sequence == null || sequence.length ==0) return false;

return VerifySquenceOfBST(sequence,0,sequence.length-1);

}

boolean VerifySquenceOfBST(int [] sequence , int start , int end){

if(start >= end) return true;

int i = start;

for(;i < end; i++){

if (sequence[i] > sequence[end]) break;

}

for (int j = i; j < end; j++) {

if(sequence[j] < sequence[end]) return false;

}

return VerifySquenceOfBST(sequence,start,i-1)&&VerifySquenceOfBST(sequence,i,end - 1);

}

}

36.

import java.util.LinkedList;

public class Solution {

public TreeNode Convert(TreeNode root) {

        TreeNode head = null;

        TreeNode pre = null;

        LinkedList<TreeNode> stack = new LinkedList<>();

        while (root != null || !stack.isEmpty()) {

            // 把root当作指针使用

            while (root != null) {

                stack.push(root);

                root = root.left;

            }

            TreeNode node = stack.pop();

            if (head == null) {

                head = node;

                pre = node;

            } else {

                node.left = pre;

                pre.right = node;

                pre = node; // 别漏写了

            }

            root = node.right;

        }

        return head;

    }

}

37.

public class Solution {

String Serialize(TreeNode root) {

if(root == null)

return "";

StringBuilder sb = new StringBuilder();

Serialize2(root, sb);

return sb.toString();

}

void Serialize2(TreeNode root, StringBuilder sb) {

if(root == null) {

sb.append("#,");

return;

}

sb.append(root.val);

sb.append(',');

Serialize2(root.left, sb);

Serialize2(root.right, sb);

}

int index = -1;

TreeNode Deserialize(String str) {

if(str.length() == 0)

return null;

String[] strs = str.split(",");

return Deserialize2(strs);

}

TreeNode Deserialize2(String[] strs) {

index++;

if(!strs[index].equals("#")) {

TreeNode root = new TreeNode(0);

root.val = Integer.parseInt(strs[index]);

root.left = Deserialize2(strs);

root.right = Deserialize2(strs);

return root;

}

return null;

}

}

39.public class Solution {

public int MoreThanHalfNum\_Solution(int [] array) {

if(array.length==0){

return 0;

}

int res = array[0], cnt = 1;

for (int i = 1;i < array.length ; i++) {

if (cnt == 0) { res = array[i]; cnt++;}

else if (array[i] == res) cnt++;

else cnt--;

}

int realCount = 0;

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

if(array[i] == res) realCount++;

}

if(realCount > array.length/2) return res;

return 0;

}

}

41.

import java.util.Comparator;

import java.util.PriorityQueue;

public class Solution {

//记录数据容器中元素个数

private int count = 0;

//利用优先级队列实现小顶堆，PriorityQueue默认会对入队的元素进行排序，所以在队列顶端的总是最小的元素

private PriorityQueue<Integer> minHeap = new PriorityQueue<>();

//利用优先级队列实现大顶堆，根据指定的比较器comparator来排序其元素，所以在队列顶端的总是最大的元素

private PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(15, new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o2 - o1;

}

});

public void Insert(Integer num) {

if (count % 2 == 0) {// 当数据总数为偶数时，新加入的元素，应当进入小根堆

// （注意不是直接进入小根堆，而是经大根堆筛选后取大根堆中最大元素进入小根堆,

//用于保证小顶堆中的最小的元素都比大顶堆中最大的元素还要大）

// 1.新加入的元素先入到大根堆，由大根堆筛选出堆中最大的元素

maxHeap.offer(num);

//筛选出大顶堆中最大的元素

int filteredMaxNum = maxHeap.poll();

// 2.筛选后的【大根堆中的最大元素】进入小根堆

minHeap.offer(filteredMaxNum);

} else {// 当数据总数为奇数时，新加入的元素，应当进入大根堆

// （注意不是直接进入大根堆，而是经小根堆筛选后取小根堆中最小元素进入大根堆）

// 用于保证小顶堆中的最小的元素都比大顶堆中最大的元素还要大

// 1.新加入的元素先入到小根堆，由小根堆筛选出堆中最小的元素

minHeap.offer(num);

int filteredMinNum = minHeap.poll();

// 2.筛选后的【小根堆中的最小元素】进入大根堆

maxHeap.offer(filteredMinNum);

}

count++;

}

public Double GetMedian() {

if (count % 2 == 0) {

return new Double((minHeap.peek() + maxHeap.peek())) / 2;

} else {

return new Double(minHeap.peek());

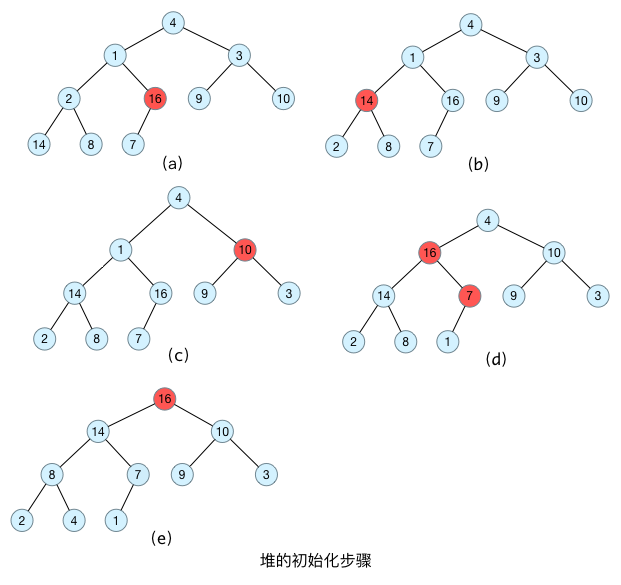
}

}

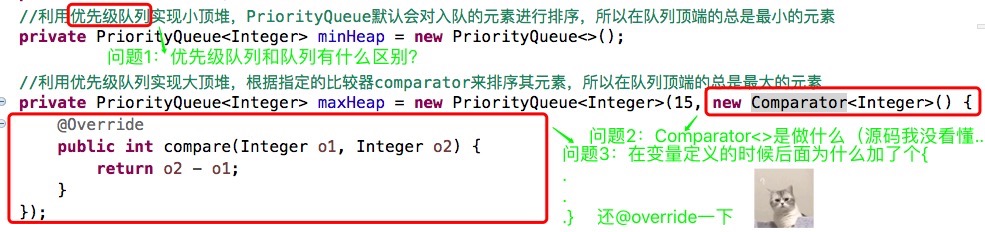
}

PART1

Pasted Graphic

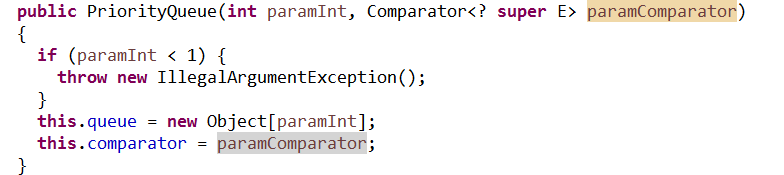


PART2



1.Priority 每次添加元素会加队列中元素进行排序，保证队列第一个是最小的元素。底层基于最小堆的原理实现，就是你一开始问我的java中有没有实现堆的类。

2.PriorityQueue底层代码的构造方法如下



重点就是第二个参数要传Comparator类型的对象，所以用new生成对象。

Compartor是一个接口，如下



里面有这种方法，如下。如果你想new对象。必须得加Override注解并实现下面的方法（就是规定，没有为什么）



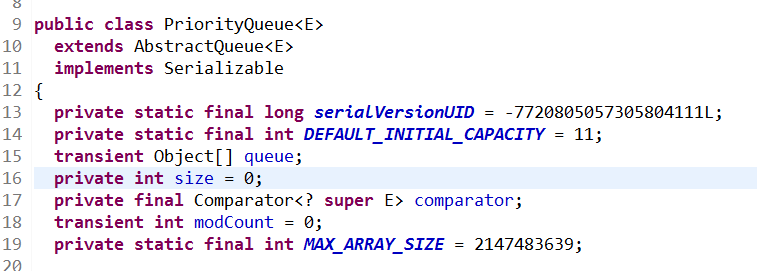
1. 加大括号这种写法叫匿名内部类

<https://www.jianshu.com/p/f53661f3cc8a>

如果再不懂，就记住吧。 如果再想就和“为什么new 可以生成一个对象”是差不多的问题了，估计永远想不明白

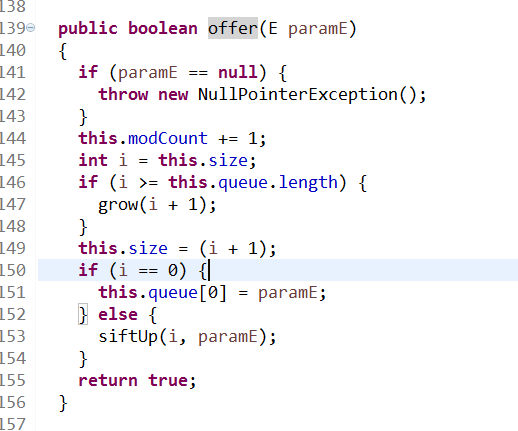
PART3

PriorQueue的源代码如下

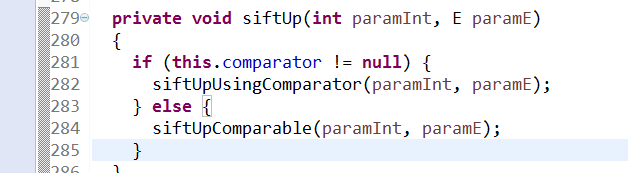


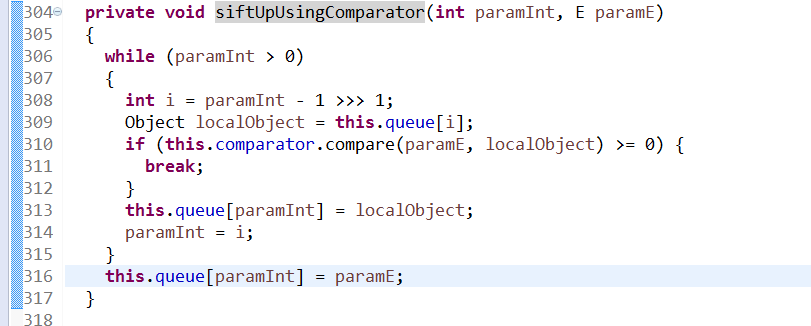
17行 有个比较器的变量 咱么new PriorQueue已经给它了

然后我想往优先级队列加元素 调用offer 方法，源码如下，看150行，如果是第一个元素，直接添加，否则调用siftUp调整队列元素顺序



siftUp源码如下 执行282行





最终310行就是调用compare方法

你想要使用PriorQueue这个类，就得需要按照他的规则，传比较器，重写compare方法。

底层比大小就是用的compare比的 所以你必须在这个接口写

42.public class Solution {

public int FindGreatestSumOfSubArray(int[] array) {

if(array == null || array.length == 0) return 0;

int maxSum = Integer.MIN\_VALUE;

int curSum = 0;

for(int digit:array){

curSum = Math.max(curSum+digit, digit);

maxSum = Math.max(maxSum, curSum);

}

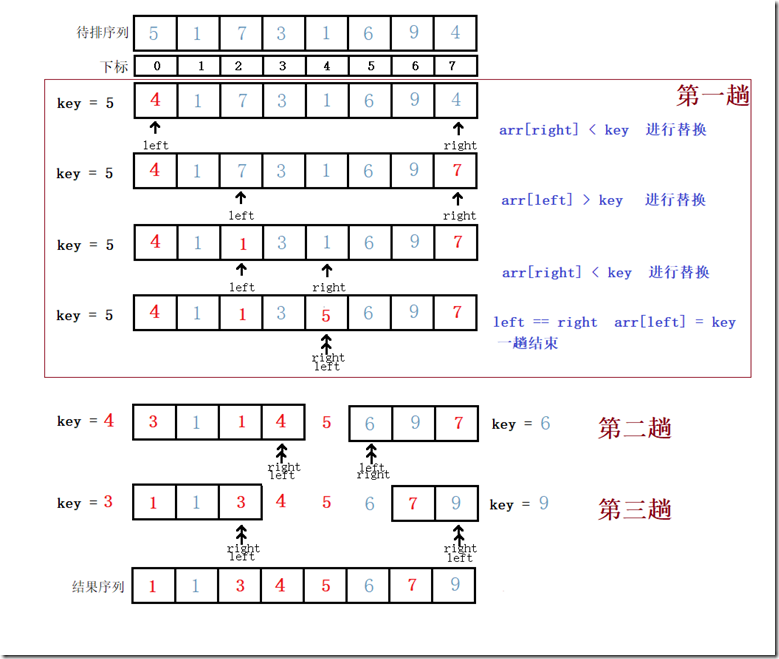
return maxSum;

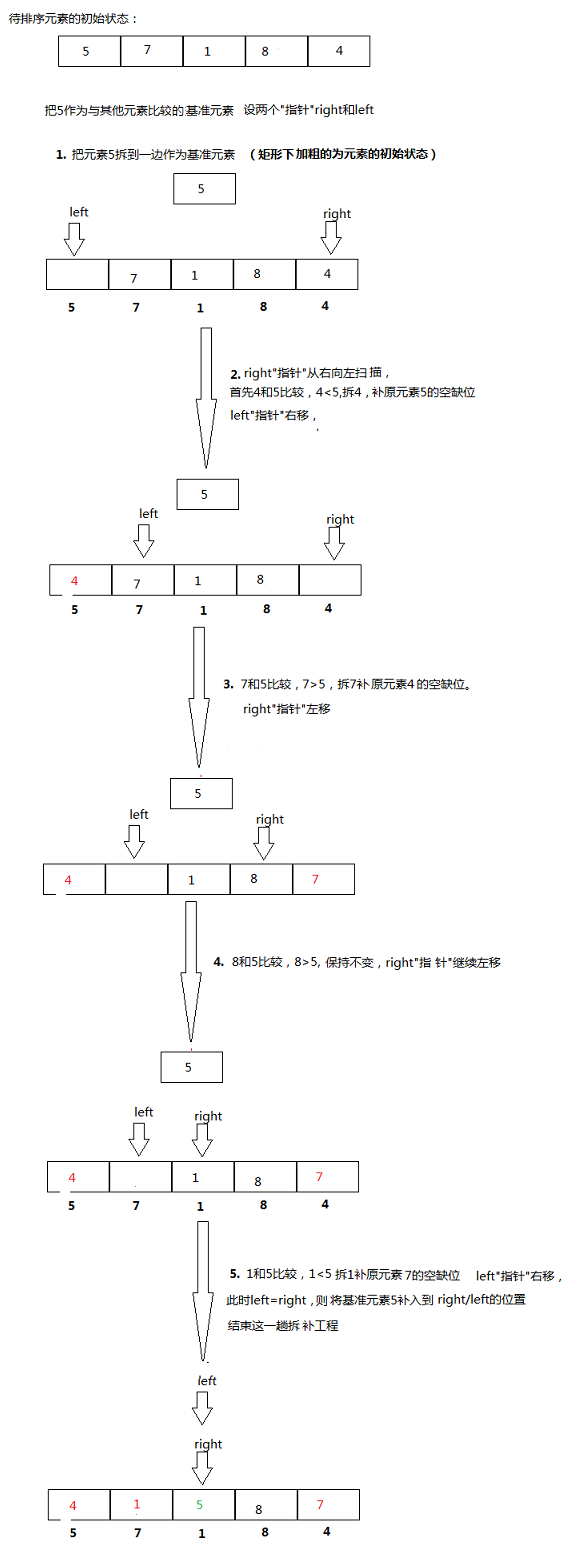
}

}

45.

快速排序算法图解：





46.public static int getTranslationCount(int number) {

if(number<0) return 0;

String sNumber=String.valueOf(number);

int len=sNumber.length();

int[] counts=new int[len];

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

if(i==len-1) {

counts[i]=1;

}else {

counts[i]=counts[i+1];

if(canBeTrans(sNumber,i)) {

if(i==len-2)

counts[i]+=1;

else

counts[i]+=counts[i+2];

}

}

}

return counts[0];

}

private static boolean canBeTrans(String sNumber, int i) {

int a=sNumber.charAt(i)-'0';

int b=sNumber.charAt(i+1)-'0';

int convert=a\*10+b;

if(convert>=10 && convert<=25)

return true;

return false;

}

1. public class GreatestGift2 {

public int getMaxValue(int[][] arr) {

if (arr == null || arr.length == 0) return 0;

int rows = arr.length;

int cols = arr[0].length;

int[] maxValue = new int[cols];

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

int left = 0;

int up = 0;

if(i > 0){

up = maxValue[j];

}

if(j > 0){

left = maxValue[j-1];

}

maxValue[j] = Math.max(up, left) + arr[i][j];

}

}

return maxValue[cols-1];

}

}

49.public class Solution {

public int GetUglyNumber\_Solution(int index) {

if(index <=0) return 0;

int[] uglys = new int[index];

uglys[0] = 1;

int multiply2 = 0;

int multiply3= 0;

int multiply5 = 0;

for(int i = 1;i < index; i++){

int min = min(uglys[multiply2]\*2, uglys[multiply3]\*3, uglys[multiply5]\*5);

uglys[i] = min;

if(uglys[multiply2]\*2 == min) multiply2++;

if(uglys[multiply3]\*3== min) multiply3++;

if(uglys[multiply5]\*5 == min) multiply5++;

}

return uglys[index-1];

}

public int min(int a, int b, int c){

int min = (a < b) ? a : b;

return (min < c) ? min : c;

}

}

50.import java.util.HashMap;

public class Solution {

public int FirstNotRepeatingChar(String str) {

if(str == null || str.length() == 0) return -1;

HashMap<Character,Integer> hashMap = new HashMap<>();

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

char c = str.charAt(i);

if(hashMap.containsKey(c)) {

int count = hashMap.get(c);

hashMap.put(c,++count);

}

else hashMap.put(c,1);

}

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

char c = str.charAt(i);

if(hashMap.get(c) == 1) return i;

}

return -1;

}

}

51.

52.

public class Solution {

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {

if(pHead1 == null || pHead2 == null ) return null;

ListNode p = pHead1;

ListNode q = pHead2;

int length1 = getLength(p);

int length2 = getLength(q);

if(length1 > length2){

for(int i = 0;i < length1 - length2; i++) p = p.next;

}else{

for(int i = 0;i < length2 - length1; i++) q = q.next;

}

while(p != null && q !=null && p!=q){

p = p.next;

q = q.next;

}

return p;

}

public int getLength(ListNode node){

if(node == null) return 0;

int length = 0;

ListNode current = node;

while(current != null){

length++;

current = current.next;

}

return length;

}

}

53.class Solution {

public int[] searchRange(int[] nums, int target) {

int[] res = new int[2];

res[0] = getStartPos(nums,target,0,nums.length - 1);

res[1] = getEndPos(nums,target,0,nums.length - 1);

return res;

}

public int getStartPos(int[] nums, int target, int start, int end){

while(start <= end){

int mid = (start + end) >> 1;

if(nums[mid] == target ){

if((mid > 0 && nums[mid - 1] != target) || mid == 0) return mid;

else end = mid - 1;

}

else if(nums[mid] > target) end = mid - 1;

else start = mid + 1;

}

return -1;

}

public int getEndPos(int[] nums, int target, int start, int end){

while(start <= end){

int mid = (start + end) >> 1;

if(nums[mid] == target ){

if((mid < nums.length - 1 && nums[mid + 1] != target) || mid == nums.length - 1) return mid;

else start = mid + 1;

}

else if(nums[mid] > target) end = mid - 1;

else start = mid + 1;

}

return -1;

}

}

55.

public class Solution {

public int TreeDepth(TreeNode root) {

if(root == null) return 0;

return 1 + Math.max(TreeDepth(root.left),TreeDepth(root.right));

}

}

56.//num1,num2分别为长度为1的数组。传出参数

//将num1[0],num2[0]设置为返回结果

public class Solution {

public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {

if(array.length < 2) return ;

        int myxor = 0;

        int flag = 1;

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

            myxor ^= array[i];

        while((myxor & flag) == 0) flag <<= 1;//flag 左移1位(二进制)

       // num1[0] = myxor;

        //num2[0] = myxor;

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

            if((flag & array[i]) == 0) num2[0]^= array[i];

            else num1[0]^= array[i];

        }

}

}

57.1.import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) {

ArrayList<Integer> res = new ArrayList<>();

if( array == null || array.length == 1) return res;

int minProduct = Integer.MAX\_VALUE;

int left = 0;

int right = array.length - 1;

while(left < right){

if(array[left] + array[right] == sum ){

if(array[left]\*array[right] < minProduct){

minProduct = array[left]\*array[right];

if(res.isEmpty()){

res.add(0,array[left]);

res.add(1,array[right]);

}else{

res.set(0,array[left]);

res.set(1,array[right]);

}

}

left++;

right--;

}else if(array[left] + array[right] < sum) left++;

else right--;

}

return res;

}

}

57.2public static void findContinusSequence(int sum){

int small = 1;

int big = 2;

int curSum = small + big;

while(small <= (sum/2)){

if(curSum == sum) {

printContinusSequence(small,big);

big++;

curSum += big;

}

else if(curSum < sum){

big++;

curSum += big;

}

else{

curSum -= small;

small++;

}

}

}

public static void printContinusSequence(int small, int big){

for(int i = small; i <= big; i++){

System.out.print(i + " ");

}

System.out.println();

}

58.public class Solution {

public String ReverseSentence(String str) {

int n = str.length();

char[] chars = str.toCharArray();

int i = 0, j = 0;

while (j <= n) {

if (j == n || chars[j] == ' ') {

reverse(chars, i, j - 1);

i = j + 1;

}

j++;

}

reverse(chars, 0, n - 1);

return new String(chars);

}

private void reverse(char[] c, int i, int j) {

while (i < j)

swap(c, i++, j--);

}

private void swap(char[] c, int i, int j) {

char t = c[i];

c[i] = c[j];

c[j] = t;

}

}

1. private static int g\_maxValue = 6;

public static void PrintProbability\_1(int number){

if(number<1) return;

int[][] pProbabilities = new int[2][g\_maxValue\*number +1];

int flag = 0;

for(int i=1;i<=g\_maxValue;i++){//当第一次抛掷骰子时，有6种可能，每种可能出现一次

pProbabilities[flag][i] = 1;

}

//从第二次开始掷骰子，假设第一个数组中的第n个数字表示骰子和为n出现的次数，

//在下一循环中，我们加上一个新骰子，此时和为n的骰子出现次数应该等于上一次循环中骰子点数和为n-1,n-2,n-3,n-4,n-5，

//n-6的次数总和，所以我们把另一个数组的第n个数字设为前一个数组对应的n-1,n-2,n-3,n-4,n-5，n-6之和

for(int k =2;k<=number;k++){

for(int i=0;i<k;i++){//第k次掷骰子，和最小为k，小于k的情况是不可能发生的！所以另不可能发生的次数设置为0！

pProbabilities[1-flag][i] = 0;

}

for(int i=k;i<=g\_maxValue\*k;i++){//第k次掷骰子，和最小为k，最大为g\_maxValue\*k

pProbabilities[1-flag][i] = 0;//初始化，因为这个数组要重复使用，上一次的值要清0

for(int j=1;j<=i&&j<=g\_maxValue;j++){

pProbabilities[1-flag][i] += pProbabilities[flag][i-j];

}

}

flag = 1-flag;

}

//求概率

double total = Math.pow(g\_maxValue, number);

for(int i=number;i<=g\_maxValue\*number;i++){

double ratio = pProbabilities[flag][i]/total;

System.out.println("sum: "+i+" ratio: "+ratio);

}

}

61.import java.util.Arrays;

public class Solution {

public boolean isContinuous(int [] numbers) {

if(numbers == null || numbers.length != 5) return false;

Arrays.sort(numbers);

int zeroNum = 0;

int i = 0;

for(;i < numbers.length && numbers[i] == 0; i++) zeroNum++;

for(;i < numbers.length - 1 && numbers[i] >= 0; i++){

int numOfVacancies = numbers[i + 1] - numbers[i] ;

if(numOfVacancies == 0) return false;

if(numOfVacancies -1 <= zeroNum){

zeroNum -= numOfVacancies - 1;

}else return false;

}

return true;

}

}

63.class Solution {

public int maxProfit(int[] prices) {

if(prices == null || prices.length<2)

return 0;

int res = 0;

int min = Integer.MAX\_VALUE;

for(int price : prices){

min = Math.min(min,price);

res = Math.max(res,price - min);

}

return res;

}

}

64.public static int Sum\_Solution(int n) {

int res = n;

boolean flag = (n>0)&&((res+=Sum\_Solution(n-1))>0);

return res;

}

66.import java.util.ArrayList;

public class Solution {

public int[] multiply(int[] A) {

/\*这道题的中心思想是将返回的数据列成一个矩阵，每一个矩阵的行向量都是一个从a(0)到a(n-1)

,然后这个对角线都是1.那么此时B0的值就是A0为一，剩下的行向量相乘。

\*/

int length = A.length;

int[] B = new int[length];

B[0] = 1;

for(int i = 1; i < length; i ++){

B[i] = B[i - 1] \* A[i - 1];

}

int temp = 1;

for(int j = length - 2; j >= 0; j --){

temp \*= A[j + 1];//temp始终会记录值，每次只需要加上这行上三角没有乘进来的数就可以了。

B[j] \*= temp;

}

return B;

}

}

## 排序算法

数字的范围？是否可以使用辅助空间？

一、算法的分类

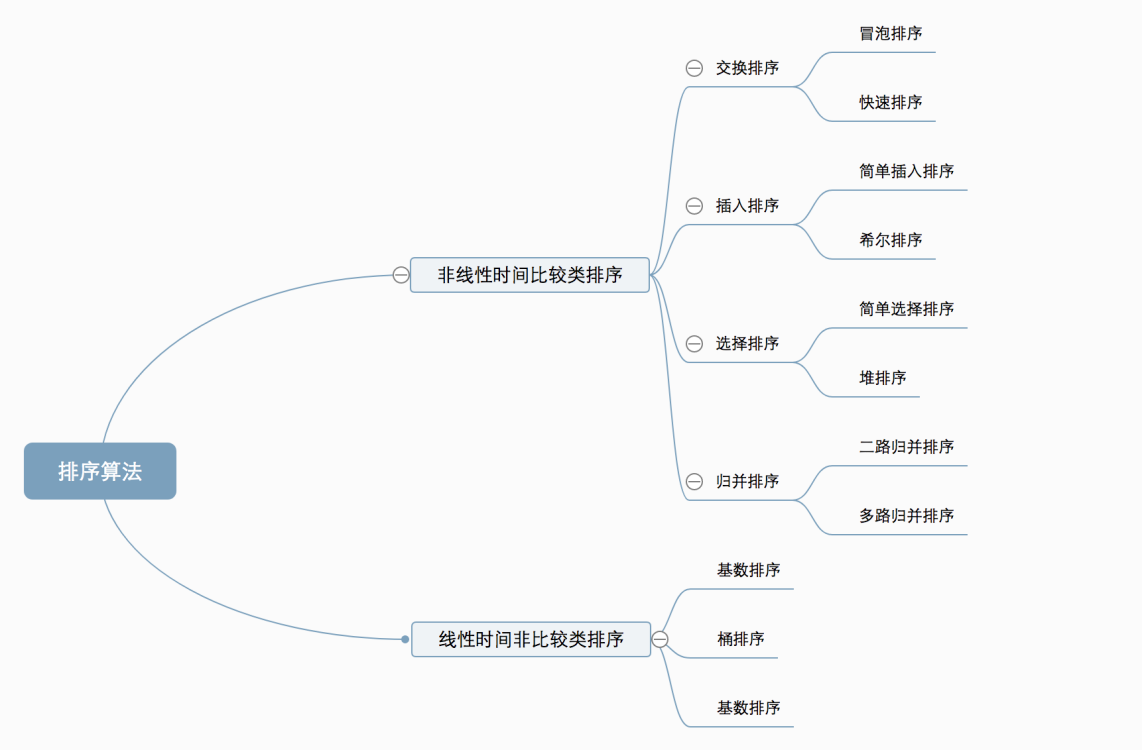
1、概念

将杂乱无章的数据元素，通过一定的方法按关键字顺序排列的过程叫做排序。

2、分类

非线性时间比较类排序：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破O(nlogn)，因此称为非线性时间比较类排序。

线性时间非比较类排序：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此称为线性时间非比较类排序。



3、比较



说明：

稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；

不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；

内排序：所有排序操作都在内存中完成；

外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；

二、各算法原理及实现

1、冒泡排序（Bubble Sort）

①、基本思想：两个数比较大小，较大的数下沉，较小的数冒起来。

②、算法描述：

比较相邻的元素。如果第一个比第二个大，就交换它们两个；

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

针对所有的元素重复以上的步骤，除了最后一个；

重复步骤1~3，直到排序完成。

public static int[] bubbleSort(int[] array) {

if (array.length == 0)

return array;

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

for (int j = 0; j < array.length - 1 - i; j++)

if (array[j + 1] < array[j]) {

int temp = array[j + 1];

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

array[j] = temp;

}

return array;

}

2、选择排序（Selection Sort）

①、基本思想：选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

②、算法描述：（n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。）

初始状态：无序区为R[1..n]，有序区为空；

第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；

n-1趟结束，数组有序化了。

③、动图演示

④、代码实现

public static int[] selectionSort(int[] array) {

if (array.length == 0)

return array;

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

int minIndex = i;

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

if (array[j] < array[minIndex]) //找到最小的数

minIndex = j; //将最小数的索引保存

}

int temp = array[minIndex];

array[minIndex] = array[i];

array[i] = temp;

}

return array;

}

3、插入排序（Insertion Sort）

①、基本思想：在要排序的一组数中，假定前n-1个数已经排好序，现在将第n个数插到前面的有序数列中，使得这n个数也是排好顺序的。如此反复循环，直到全部排好顺序。

②、算法描述：

从第一个元素开始，该元素可以认为已经被排序；

取出下一个元素，在已经排序的元素序列中从后向前扫描；

如果该元素（已排序）大于新元素，将该元素移到下一位置；

重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

将新元素插入到该位置后；

重复步骤2~5。

③、动图演示

④、代码实现

public static int[] insertionSort(int[] array) {

if (array.length == 0)

return array;

int current;

for (int i = 0; i < array.length - 1; i++) {

current = array[i + 1];

int preIndex = i;

while (preIndex >= 0 && current < array[preIndex]) {

array[preIndex + 1] = array[preIndex];

preIndex--;

}

array[preIndex + 1] = current;

}

return array;

}

4、希尔排序（Shell Sort）

①、基本思想：希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破O(n2）的第一批算法之一。它与插入排序的不同之处在于，它会优先比较距离较远的元素。

②、算法描述：

选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

按增量序列个数k，对序列进行k 趟排序；

每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

③、动图演示：

④、代码实现

public static int[] ShellSort(int[] array) {

int len = array.length;

int temp, gap = len / 2;

while (gap > 0) {

for (int i = gap; i < len; i++) {

temp = array[i];

int preIndex = i - gap;

while (preIndex >= 0 && array[preIndex] > temp) {

array[preIndex + gap] = array[preIndex];

preIndex -= gap;

}

array[preIndex + gap] = temp;

}

gap /= 2;

}

return array;

}

⑤、算法分析

希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。

while (gap < len / 3) { // 动态定义间隔序列

gap = gap \* 3 + 1;

}

5、归并排序（Merge Sort）（会写）

①、基本思想：

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

②、算法描述：

把长度为n的输入序列分成两个长度为n/2的子序列；

对这两个子序列分别采用归并排序；

将两个排序好的子序列合并成一个最终的排序序列。

public static int[] MergeSort(int[] array) {

if (array.length < 2) return array;

int mid = array.length / 2;

int[] left = Arrays.copyOfRange(array, 0, mid);

int[] right = Arrays.copyOfRange(array, mid, array.length);

return merge(MergeSort(left), MergeSort(right));

}

/\*\*

\* 归并排序——将两段排序好的数组结合成一个排序数组

public static int[] merge(int[] left, int[] right) {

int[] result = new int[left.length + right.length];

for (int index = 0, i = 0, j = 0; index < result.length; index++) {

if (i >= left.length) result[index] = right[j++];

else if (j >= right.length) result[index] = left[i++];

else if (left[i] > right[j]) result[index] = right[j++];

else result[index] = left[i++];

}

return result;

}

6、快速排序（Quick Sort）（会写）

①、基本思想（分治）：

通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

②、算法描述：快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

从数列中挑出一个元素，称为 “基准”（pivot）；

重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

/\*\*

\* 快速排序方法

\* @param array

\* @param start

\* @param end

\* @return

\*/

public static int[] QuickSort(int[] array, int start, int end) {

if (array.length < 1 || start < 0 || end >= array.length || start > end) return null;

int smallIndex = partition(array, start, end);

if (smallIndex > start)

QuickSort(array, start, smallIndex - 1);

if (smallIndex < end)

QuickSort(array, smallIndex + 1, end);

return array;

}

/\*\*

\* 快速排序算法——partition

\* @param array

\* @param start

\* @param end

\* @return

\*/

public static int partition(int[] array, int start, int end) {

int pivot = (int) (start + Math.random() \* (end - start + 1));

int smallIndex = start - 1;

swap(array, pivot, end);

for (int i = start; i <= end; i++)

if (array[i] <= array[end]) {

smallIndex++;

if (i > smallIndex)

swap(array, i, smallIndex);

}

return smallIndex;

}

/\*\*

\* 交换数组内两个元素

\* @param array

\* @param i

\* @param j

\*/

public static void swap(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

7、堆排序（Heap Sort）

①、基本思想：堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个**近似完全二叉树**的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

②、算法描述：

将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；

将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；

由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

//声明全局变量，用于记录数组array的长度；

static int len;

public static int[] HeapSort(int[] array) {

len = array.length;

if (len < 1) return array;

//1.构建一个最大堆

buildMaxHeap(array);

//2.循环将堆首位（最大值）与末位交换，然后在重新调整最大堆

while (len > 0) {

swap(array, 0, len - 1);

len--;

adjustHeap(array, 0);

}

return array;

}

/\*\*

\* 建立最大堆

\*/

public static void buildMaxHeap(int[] array) {

//从最后一个非叶子节点开始向上构造最大堆

for (int i = (len/2 - 1); i >= 0; i--) { //感谢 @让我发会呆 网友的提醒，此处应该为 i = (len/2 - 1)

adjustHeap(array, i);

}

}

/\*\*

\* 调整使之成为最大堆

\*/

public static void adjustHeap(int[] array, int i) {

int maxIndex = i;

//如果有左子树，且左子树大于父节点，则将最大指针指向左子树

if (i \* 2 < len && array[i \* 2] > array[maxIndex])

maxIndex = i \* 2;

//如果有右子树，且右子树大于父节点，则将最大指针指向右子树

if (i \* 2 + 1 < len && array[i \* 2 + 1] > array[maxIndex])

maxIndex = i \* 2 + 1;

//如果父节点不是最大值，则将父节点与最大值交换，并且递归调整与父节点交换的位置。

if (maxIndex != i) {

swap(array, maxIndex, i);

adjustHeap(array, maxIndex);

}

}

8、计数排序（Counting Sort）

①、基本思想：计数排序不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

②、算法描述：

找出待排序的数组中最大和最小的元素；

统计数组中每个值为i的元素出现的次数，存入数组C的第i项；

对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；

反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

③、动图演示：

④、代码实现：

/\*\*

\* 计数排序

\*

\* @param array

\* @return

\*/

public static int[] CountingSort(int[] array) {

if (array.length == 0) return array;

int bias, min = array[0], max = array[0];

for (int i = 1; i < array.length; i++) {

if (array[i] > max)

max = array[i];

if (array[i] < min)

min = array[i];

}

bias = 0 - min;

int[] bucket = new int[max - min + 1];

Arrays.fill(bucket, 0);

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

bucket[array[i] + bias]++;

}

int index = 0, i = 0;

while (index < array.length) {

if (bucket[i] != 0) {

array[index] = i - bias;

bucket[i]--;

index++;

} else

i++;

}

return array;

}

9、桶排序（Bucket Sort）

①、基本思想：

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排）。

②、算法描述：

设置一个定量的数组当作空桶；

遍历输入数据，并且把数据一个一个放到对应的桶里去；

对每个不是空的桶进行排序；

从不是空的桶里把排好序的数据拼接起来。

③、图片演示：

④、代码实现：

/\*\*

\* 桶排序

\*

\* @param array

\* @param bucketSize

\* @return

\*/

public static ArrayList<Integer> BucketSort(ArrayList<Integer> array, int bucketSize) {

if (array == null || array.size() < 2)

return array;

int max = array.get(0), min = array.get(0);

// 找到最大值最小值

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

if (array.get(i) > max)

max = array.get(i);

if (array.get(i) < min)

min = array.get(i);

}

int bucketCount = (max - min) / bucketSize + 1;

ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketCount);

ArrayList<Integer> resultArr = new ArrayList<>();

for (int i = 0; i < bucketCount; i++) {

bucketArr.add(new ArrayList<Integer>());

}

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

bucketArr.get((array.get(i) - min) / bucketSize).add(array.get(i));

}

for (int i = 0; i < bucketCount; i++) {

if (bucketSize == 1) { // 如果带排序数组中有重复数字时 感谢 @见风任然是风 朋友指出错误

for (int j = 0; j < bucketArr.get(i).size(); j++)

resultArr.add(bucketArr.get(i).get(j));

} else {

if (bucketCount == 1)

bucketSize--;

ArrayList<Integer> temp = BucketSort(bucketArr.get(i), bucketSize);

for (int j = 0; j < temp.size(); j++)

resultArr.add(temp.get(j));

}

}

return resultArr;

}

10、基数排序（Radix Sort）

①、基本思想：

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。

②、算法描述：

取得数组中的最大数，并取得位数；

arr为原始数组，从最低位开始取每个位组成radix数组；

对radix进行计数排序（利用计数排序适用于小范围数的特点）；

③、动图演示：

④、代码实现：

/\*\*

\* 基数排序

\* @param array

\* @return

\*/

public static int[] RadixSort(int[] array) {

if (array == null || array.length < 2)

return array;

// 1.先算出最大数的位数；

int max = array[0];

for (int i = 1; i < array.length; i++) {

max = Math.max(max, array[i]);

}

int maxDigit = 0;

while (max != 0) {

max /= 10;

maxDigit++;

}

int mod = 10, div = 1;

ArrayList<ArrayList<Integer>> bucketList = new ArrayList<ArrayList<Integer>>();

for (int i = 0; i < 10; i++)

bucketList.add(new ArrayList<Integer>());

for (int i = 0; i < maxDigit; i++, mod \*= 10, div \*= 10) {

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

int num = (array[j] % mod) / div;

bucketList.get(num).add(array[j]);

}

int index = 0;

for (int j = 0; j < bucketList.size(); j++) {

for (int k = 0; k < bucketList.get(j).size(); k++)

array[index++] = bucketList.get(j).get(k);

bucketList.get(j).clear();

}

}

return array;

}

## 3.查找算法

二分查找(jdk 中Arrays类中的)

**private** **static** **int** binarySearch0(**int**[] a, **int** fromIndex, **int** toIndex,

**int** key) {

**int** low = fromIndex;

**int** high = toIndex - 1;

**while** (low <= high) {

**int** mid = (low + high) >>> 1;

**int** midVal = a[mid];

**if** (midVal < key)

low = mid + 1;

**else** **if** (midVal > key)

high = mid - 1;

**else**

**return** mid; // key found

}

**return** -(low + 1); // key not found.

}

## 4.jdk源码

二分查找

public static int binarySearch(int[] a, int key) {

return binarySearch0(a, 0, a.length, key);

}

// Like public version, but without range checks.

**private** **static** **int** binarySearch0(**int**[] a, **int** fromIndex, **int** toIndex,**int** key) {

**int** low = fromIndex;

**int** high = toIndex - 1;

**while** (low <= high) {

**int** mid = (low + high) >>> 1; //无符号右移 除2

**int** midVal = a[mid];

**if** (midVal < key)

low = mid + 1;

**else** **if** (midVal > key)

high = mid - 1;

**else**

**return** mid; // key found

}

**return** -(low + 1); // key not found.

}

# SSM

## 用到Spring什么特性 结合项目说一下 依赖反转的好处

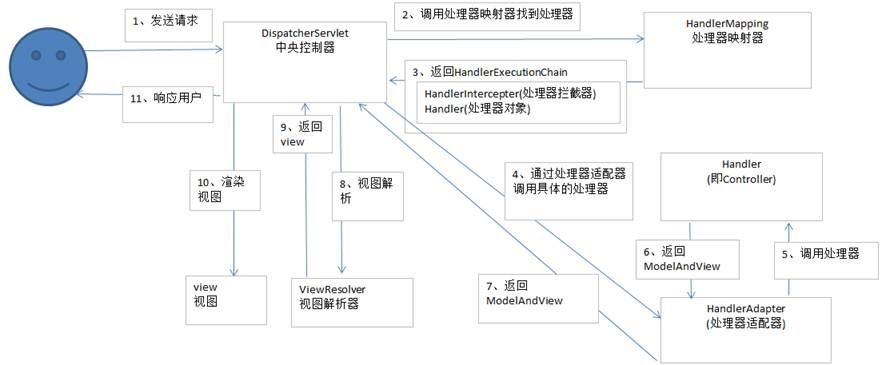
## 2.Spring主要使用了什么模式？

* 工厂模式：每个Bean的创建通过方法
* 单例模式：默认的每个Bean的作用域都是单例
* 代理模式：关于Aop的实现通过代理模式(JDK动态代理）
* 3.适配器模式（HandlerAdater）
* 4.装饰者模式
* 6.观察者模式（listener的实现，例如ApplicationListener）
* 7.策略模式（定义一系列的算法，把它们一个个的封装起来，并且使它们可以相互替换。在实例化对象时用到）
* 8.模板模式（jdbcTemplate）

## 3.IOC，AOP的实现原理？

* IOC：通过反射机制生成对象注入
* AOP：动态代理

## 4.讲述SpringMVC工作流程流程



1、用户发送请求至前端控制器DispatcherServlet

2、DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、处理器映射器找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、DispatcherServlet调用HandlerAdapter处理器适配器

5、HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、Controller执行完成返回ModelAndView

7、HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

8、DispatcherServlet将ModelAndView传给ViewReslover视图解析器

9、ViewReslover解析后返回具体View

10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

11、DispatcherServlet响应用户

## 5.为什么使用spring

Spring是一个库，它的功能是提供了一个软件框架，这个框架目的是使软件之间的逻辑更加清晰，配置更灵活。降低对象耦合度,让代码更加清晰,提供一些常见的模版

## 6.Spring DI的几种方式

(1)构造器注入：通过构造方法初始化

<constructor-arg name="dao"</constructor-arg>

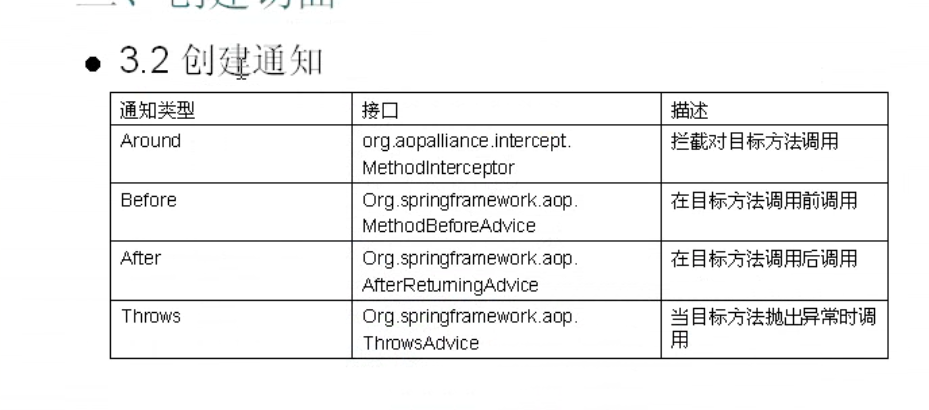
(2)setter注入：通过setter方法初始化注入

<property name="dao" ref="dao2"></property>

注意：在实际开发中常用setter注入。

## 7.Aop

1.通知：



## Spring中aop,ioc怎么实现的。

IOC：

java程序中的每个业务逻辑至少需要两个或以上的对象来协作完成，通常，在使用其他的合作对象时，均要使用像new object（） 这样的语法来完成合作对象的申请工作。你会发现：对象间的耦合度高了。而IOC的思想是：Spring容器来实现这些相互依赖对象的创建、协调工作。对象只需要关心业务逻辑本身就可以了。从这方面来说，对象如何得到他的协作对象的责任被反转了（IOC、DI）。

IOC是基于java的反射机制以及工厂模式实现的。具体可以看下这两篇文章，

工厂：各类都实现了某一接口，并提供一个工场类，根据传入参数的不同来创建不同的对象

反射：可以根据类名获取该类实例， Class.forName(ClassName).newInstance()

在使用spring的时候会将每一个service定义为一个bean，这些bean的名字肯定是不同的，spring会维护一个xml文件，文件中各条记录的id是bean的名字，value是对应的完整类名(包括路径)，在项目启动时将该文件中的id和value获取，通过反射的方式将value即类名所对应的实例创建，然后将id即bean的名字与实例以类似键值对的方式存储在spring容器中，当需要使用时直接利用getBean(String beanName)从容器中获取对应的实例。

AOP主要是使用动态代理实现

AOP(面向切面编程)技术利用一种称为“横切”的技术，解剖封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，这样就能减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。

动态代理(被代理的类需要实现一个接口)：中使用InvocationHandler接口和Proxy类来实现，在InvocationHandler中的invoke(Proxy proxy,Method method, object arg[])方法，中处理日志，认证，事务等操作

步骤：（1）构造InvocationHandler处理器，定义invoke()方法（2）获取Proxy代理类，并生成代理类对象，调用方法完成相关操作

Proxy.newProxyInstance()

public static Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

throws IllegalArgumentException

newProxyInstance，方法有三个参数：

loader: 用哪个类加载器去加载代理对象

interfaces:动态代理类需要实现的接口

h:动态代理方法在执行时，会调用h里面的invoke方法去执行，invoke(Proxy proxy,Method method, object arg[])

静态代理：实现了被代理类接口的类，其中保存了一个被代理类的实例。在接口的实现方法中，调用被代理对象对应的方法，同时添加需要的其他操作。缺点：代理类和委托类都实现了同样的接口，代码重复；当需要对对不同的委托类实现相同的处理时，需要定义多个代理类

无论是静态代理还是动态代理都需要委托类实现接口，如果要对不实现接口的类进行代理，就需要使用cglib库,使用Enhancer来创建代理类，使用MethodIntercepter接口，重写intercept()方法，实现具体操作

# 中间件

## 1.Redis

### 1.redis的过期策略以及内存淘汰机制redis采用的是定期删除+惰性删除策略。 为什么不用定时删除策略? 定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略. 定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redis岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。

1. 于是，惰性删除派上用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除。  
   采用定期删除+惰性删除就没其他问题了么?  
   不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用内存淘汰机制。  
   在redis.conf中有一行配置

# maxmemory-policy allkeys-lru

该配置就是配内存淘汰策略:当内存不足以容纳新写入数据时  
1）noeviction：新写入操作会报错。应该没人用吧。  
2）allkeys-lru：在键空间中，移除最近最少使用的key。推荐使用。  
3）allkeys-random：在键空间中，随机移除某个key。应该也没人用吧，你不删最少使用Key,去随机删。  
4）volatile-lru：在设置了过期时间的键空间中，移除最近最少使用的key。这种情况一般是把redis既当缓存，又做持久化存储的时候才用。不推荐  
5）volatile-random：在设置了过期时间的键空间中，随机移除某个key。依然不推荐  
6）volatile-ttl：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。不推荐  
ps:如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

### 2.缓存穿透 击穿 雪崩

这几种情况都是从缓存没有获取到数据，大量的并发请求到了数据源，给数据源造成很大压力，从而可能引发问题

1.缓存穿透：key对应的数据在数据源并不存在，每次针对此key的请求从缓存获取不到，请求都会到数据源，从而可能压垮数据源。比如用一个不存在的用户id获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。

采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热(项目启动前，先加载缓存)操作。

提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的key。迅速判断出，请求所携带的Key是否合法有效。如果不合法，则直接返回。

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。

1. 缓存击穿：key对应的数据存在，但在redis中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试。

3.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，也会给后端系统(比如DB)带来很大压力。

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。

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。

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.Redis优势和特点

特点1.内存数据库，速度快，也支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

2.Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

3.Redis支持数据的备份，即master-slave模式的数据备份。

4支持事务

优势：

1.性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。

2.丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

3.原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行。（事务）

4.丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同？

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

### 4.知道Redis的持久化吗？都有什么缺点和优点？具体底层实现呢？

RDB:

优点：最大化发挥redis的性能，主进程不需要进行任何I/O操作，只需要派生一个子进程来处理，文件非常紧凑，恢复数据的效率高。

缺点：数据恢复时一致性和完整性较差，因为也许最后一次备份前就宕机了，那么在最后一次备份到宕机中间的数据是不会存在的

AOF:

优点：能保持数据的一致性和完整性

缺点：AOF文件比RDB文件大，在读取过程中，会比RDB更慢一些。

底层实现

RDB:是在达到指定的时间或者操作次数后，自动将在内存中的数据写入磁盘

AOF:是日志形式，当数据写入内存中的时候，在日志文件下记录下所有写操作。

## 消息队列

### 什么是消息队列

我们可以把消息队列比作是一个存放消息的容器，当我们需要使用消息的时候可以取出消息供自己使用。消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提高系统性能和削峰、降低系统耦合性。目前使用较多的消息队列有ActiveMQ，RabbitMQ，Kafka，RocketMQ。

另外，我们知道队列 Queue 是一种先进先出的数据结构，所以消费消息时也是按照顺序来消费的。比如生产者发送消息1,2,3...对于消费者就会按照1,2,3...的顺序来消费。但是偶尔也会出现消息被消费的顺序不对的情况，比如某个消息消费失败又或者一个 queue 多个consumer 也会导致消息被消费的顺序不对，我们一定要保证消息被消费的顺序正确。

除了上面说的消息消费顺序的问题，使用消息队列，我们还要考虑如何保证消息不被重复消费？如何保证消息的可靠性传输（如何处理消息丢失的问题）？......等等问题。所以说使用消息队列也不是十全十美的，使用它也会让系统可用性降低、复杂度提高，另外需要我们保障一致性等问题。

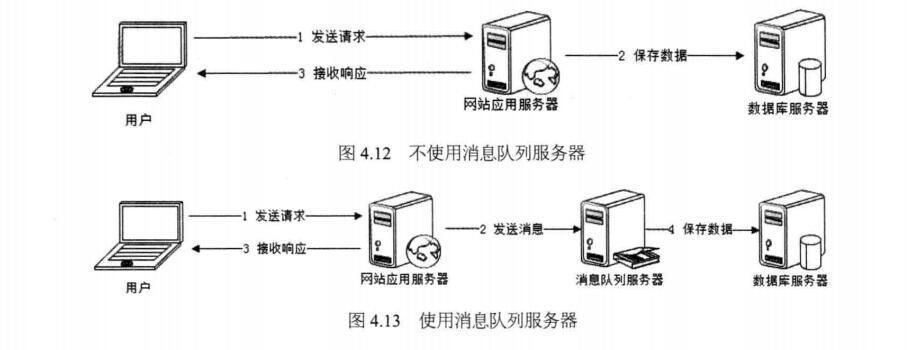
 把数据放到消息队列叫做生产者  从消息队列里边取数据叫做消费者

### 为什么使用消息队列

1. 通过异步处理提高系统性能（削峰、减少响应所需时间）;

2.降低系统耦合性。如果在面试的时候你被面试官问到这个问题的话，一般情况是你在你的简历上涉及到消息队列这方面的内容，这个时候推荐你结合你自己的项目来回答。

**(1) 通过异步处理提高系统性能（削峰、减少响应所需时间）**



通过异步处理提高系统性能

如上图，**在不使用消息队列服务器的时候，用户的请求数据直接写入数据库，在高并发的情况下数据库压力剧增，使得响应速度变慢。但是在使用消息队列之后，用户的请求数据发送给消息队列之后立即 返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。由于消息队列服务器处理速度快于数据库（消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。**

通过以上分析我们可以得出**消息队列具有很好的削峰作用的功能**——即**通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。**举例：在电子商务一些秒杀、促销活动中，合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击。如下图所示：

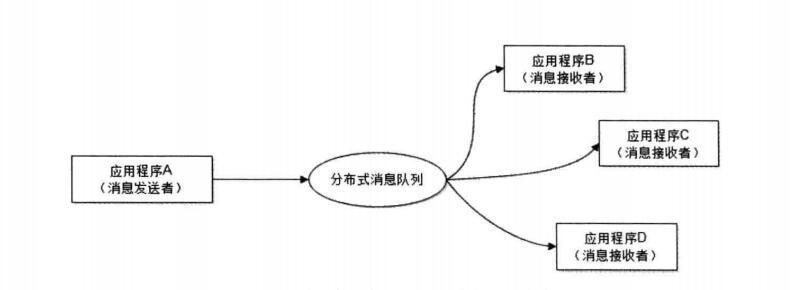
合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击

因为**用户请求数据写入消息队列之后就立即返回给用户了，但是请求数据在后续的业务校验、写数据库等操作中可能失败**。因此使用消息队列进行异步处理之后，需要**适当修改业务流程进行配合**，比如**用户在提交订单之后，订单数据写入消息队列，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单之后，甚至出库后，再通过电子邮件或短信通知用户订单成功**，以免交易纠纷。这就类似我们平时手机订火车票和电影票。

**(2) 降低系统耦合性**

我们知道如果模块之间不存在直接调用，那么新增模块或者修改模块就对其他模块影响较小，这样系统的可扩展性无疑更好一些。

我们最常见的**事件驱动架构**类似生产者消费者模式，在大型网站中通常用利用消息队列实现事件驱动结构。如下图所示：



利用消息队列实现事件驱动结构

**消息队列使利用发布-订阅模式工作，消息发送者（生产者）发布消息，一个或多个消息接受者（消费者）订阅消息。** 从上图可以看到**消息发送者（生产者）和消息接受者（消费者）之间没有直接耦合**，消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接受者从分布式消息队列获取该消息后进行后续处理，并不需要知道该消息从何而来。**对新增业务，只要对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展性设计**。

消息接受者对消息进行过滤、处理、包装后，构造成一个新的消息类型，将消息继续发送出去，等待其他消息接受者订阅该消息。因此基于事件（消息对象）驱动的业务架构可以是一系列流程。

**另外为了避免消息队列服务器宕机造成消息丢失，会将成功发送到消息队列的消息存储在消息生产者服务器上，等消息真正被消费者服务器处理后才删除消息。在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中的其他服务器发布消息。**

**备注：** 不要认为消息队列只能利用发布-订阅模式工作，只不过在解耦这个特定业务环境下是使用发布-订阅模式的。**除了发布-订阅模式，还有点对点订阅模式（一个消息只有一个消费者），我们比较常用的是发布-订阅模式。** 另外，这两种消息模型是 JMS 提供的，AMQP 协议还提供了 5 种消息模型。

### 3. 使用消息队列带来的一些问题

**1.系统可用性降低：** 系统可用性在某种程度上降低，为什么这样说呢？在加入MQ之前，你不用考虑消息丢失或者说MQ挂掉等等的情况，但是，引入MQ之后你就需要去考虑了！

**2.系统复杂性提高：** 加入MQ之后，你需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等等问题！

**3.一致性问题：** 我上面讲了消息队列可以实现异步，消息队列带来的异步确实可以提高系统响应速度。但是，万一消息的真正消费者并没有正确消费消息怎么办？这样就会导致数据不一致的情况了!

#### 四 JMS VS AMQP

**4.1 JMS**

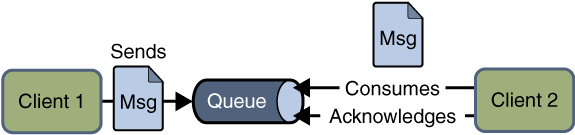
**4.1.1 JMS 简介**

JMS（JAVA Message Service,java消息服务）是java的消息服务，JMS的客户端之间可以通过JMS服务进行异步的消息传输。**JMS（JAVA Message Service,Java消息服务）API是一个消息服务的标准或者说是规范**，允许应用程序组件基于JavaEE平台创建、发送、接收和读取消息。它使分布式通信耦合度更低，消息服务更加可靠以及异步性。

**ActiveMQ 就是基于 JMS 规范实现的。**

**4.1.2 JMS两种消息模型**

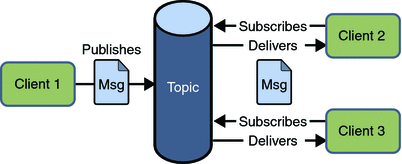
①点到点（P2P）模型



点到点（P2P）模型

使用**队列（Queue）**作为消息通信载体；满足**生产者与消费者模式**，一条消息只能被一个消费者使用，未被消费的消息在队列中保留直到被消费或超时。比如：我们生产者发送100条消息的话，两个消费者来消费一般情况下两个消费者会按照消息发送的顺序各自消费一半（也就是你一个我一个的消费。）

② 发布/订阅（Pub/Sub）模型



发布/订阅（Pub/Sub）模型

发布订阅模型（Pub/Sub） 使用**主题（Topic）**作为消息通信载体，类似于**广播模式**；发布者发布一条消息，该消息通过主题传递给所有的订阅者，**在一条消息广播之后才订阅的用户则是收不到该条消息的**。

**4.1.3 JMS 五种不同的消息正文格式**

JMS定义了五种不同的消息正文格式，以及调用的消息类型，允许你发送并接收以一些不同形式的数据，提供现有消息格式的一些级别的兼容性。

* StreamMessage -- Java原始值的数据流
* MapMessage--一套名称-值对
* TextMessage--一个字符串对象
* ObjectMessage--一个序列化的 Java对象
* BytesMessage--一个字节的数据流

**4.2 AMQP**

​ AMQP，即Advanced Message Queuing Protocol,一个提供统一消息服务的应用层标准 **高级消息队列协议**（二进制应用层协议），是应用层协议的一个开放标准,为面向消息的中间件设计，兼容 JMS。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件同产品，不同的开发语言等条件的限制。

**RabbitMQ 就是基于 AMQP 协议实现的。**

**4.3 JMS vs AMQP**

| **对比方向** | **JMS** | **AMQP** |
| --- | --- | --- |
| 定义 | Java API | 协议 |
| 跨语言 | 否 | 是 |
| 跨平台 | 否 | 是 |
| 支持消息类型 | 提供两种消息模型：①Peer-2-Peer;②Pub/sub | 提供了五种消息模型：①direct exchange；②fanout exchange；③topic change；④headers exchange；⑤system exchange。本质来讲，后四种和JMS的pub/sub模型没有太大差别，仅是在路由机制上做了更详细的划分； |
| 支持消息类型 | 支持多种消息类型 ，我们在上面提到过 | byte[]（二进制） |

**总结：**

* AMQP 为消息定义了线路层（wire-level protocol）的协议，而JMS所定义的是API规范。在 Java 体系中，多个client均可以通过JMS进行交互，不需要应用修改代码，但是其对跨平台的支持较差。而AMQP天然具有跨平台、跨语言特性。
* JMS 支持TextMessage、MapMessage 等复杂的消息类型；而 AMQP 仅支持 byte[] 消息类型（复杂的类型可序列化后发送）。
* 由于Exchange 提供的路由算法，AMQP可以提供多样化的路由方式来传递消息到消息队列，而 JMS 仅支持 队列 和 主题/订阅 方式两种。

#### 五 常见的消息队列对比

| **对比方向** | **概要** |
| --- | --- |
| 吞吐量 | 万级的 ActiveMQ 和 RabbitMQ 的吞吐量（ActiveMQ 的性能最差）要比 十万级甚至是百万级的 RocketMQ 和 Kafka 低一个数量级。 |
| 可用性 | 都可以实现高可用。ActiveMQ 和 RabbitMQ 都是基于主从架构实现高可用性。RocketMQ 基于分布式架构。 kafka 也是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用 |
| 时效性 | RabbitMQ 基于erlang开发，所以并发能力很强，性能极其好，延时很低，达到微秒级。其他三个都是 ms 级。 |
| 功能支持 | 除了 Kafka，其他三个功能都较为完备。 Kafka 功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用，是事实上的标准 |
| 消息丢失 | ActiveMQ 和 RabbitMQ 丢失的可能性非常低， RocketMQ 和 Kafka 理论上不会丢失。 |

**总结：**

* ActiveMQ 的社区算是比较成熟，但是较目前来说，ActiveMQ 的性能比较差，而且版本迭代很慢，不推荐使用。
* RabbitMQ 在吞吐量方面虽然稍逊于 Kafka 和 RocketMQ ，但是由于它基于 erlang 开发，所以并发能力很强，性能极其好，延时很低，达到微秒级。但是也因为 RabbitMQ 基于 erlang 开发，所以国内很少有公司有实力做erlang源码级别的研究和定制。如果业务场景对并发量要求不是太高（十万级、百万级），那这四种消息队列中，RabbitMQ 一定是你的首选。如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。
* RocketMQ 阿里出品，Java 系开源项目，源代码我们可以直接阅读，然后可以定制自己公司的MQ，并且 RocketMQ 有阿里巴巴的实际业务场景的实战考验。RocketMQ 社区活跃度相对较为一般，不过也还可以，文档相对来说简单一些，然后接口这块不是按照标准 JMS 规范走的有些系统要迁移需要修改大量代码。还有就是阿里出台的技术，你得做好这个技术万一被抛弃，社区黄掉的风险，那如果你们公司有技术实力我觉得用RocketMQ 挺好的
* kafka 的特点其实很明显，就是仅仅提供较少的核心功能，但是提供超高的吞吐量，ms 级的延迟，极高的可用性以及可靠性，而且分布式可以任意扩展。同时 kafka 最好是支撑较少的 topic 数量即可，保证其超高吞吐量。kafka 唯一的一点劣势是有可能消息重复消费，那么对数据准确性会造成极其轻微的影响，在大数据领域中以及日志采集中，这点轻微影响可以忽略这个特性天然适合大数据实时计算以及日志收集。

# 面经

## 猿辅导

项目

三次握手四次挥手

list set

hashmap

set和map区别

cookie session

jvm内存结构

方法区

static加载一次

泛型实现

注解实现

单例模式

死锁条件

动态代理

volatile

mcv

序列化

synchroized

项目

Spring AOP

线程池

tcp和udp

事务

B树和B+树

求一个二叉树的宽度

项目

为什么要用SpringBoot

扫描配置的过程

索引的数据结构

组合索引的规则

分时操作系统和实时操作系统

进程和线程的区别

中断和陷入

协程

Java的GC

返回子节点的值等于其孩子节点值之差的父节点

## 字节跳动

### 1.Java AIO、NIO、BIO

### 2、HashMap底层实现、链表转红黑树的阈值为什么是8？

### 3、String、StringBuffer、StringBuilder的区别

### 4.CAS的实现原理

### 5、Java线程状态

### 6、Java类加载机制

### 7、Redis分布式锁、底层实现原理

### 8、Http状态码302/403

### 9、HTTPs请求过程

### 10、TCP/UDP区别

### 11、分布式一致性算法

### 12、进程间通信、线程间同步的方式分别

### 13、虚拟内存的实现原理

### 14、归并排序过程、时空复杂度

将待排序序列从中间一分为二，对左右两边再进行递归分割操作，得到n个相互独立的子序列；

对n个独立的子序列递归的执行合并操作，最终得到有序的序列。

### 15、用户态、内核态的区别。

## 编程：字符串的解压。

Input：3%abc#

Output：abcabcabc

其中%和#可以内部嵌套

### 字节跳动成都研发中心秋招提前批123面面经

1. https 讲下，为什么要双方确定一个随机文本当密钥，我说怕中间有人抓包，拿到客户端发的参数，然后问为什么不都用公钥加密呢？我想也是哈，中间反正也解不开，但是没想到客户端解不开服务端的信息，脑子卡克了。
2. fork（）函数讲下，以前在做OJ判题的时候涉及到过，结合项目讲了下。
3. 线程池种类，cache线程池讲下，线程无限制增长吗？
4. 数据库索引，B+树和HASH中为什么用B+树，好处是啥（这个只把两个结构说了，范围查询，排序快的好处居然没想到）
5. 复合索引的结构，最左匹配原理（abc查询问题）
6. 算法01背包（无限物品）： 有商品（200 150 350），你只有N块钱，但是不能找钱，你怎么买最不亏，也就是买的多。三层for循环A了，然后问我优化（心想这个是最简模板，假装思考了下说没了）。然后目瞪口呆的看他秀了一下：优化一：商品200+150=350，完全可以抛开一个物品，复杂度减到2/3 优化二：都是50的差距，没必要1块1块的减少，又可以减很多复杂度。我去这太坑爹了吧，这个算法题，真的是服

## java的static讲下，然后有6种用法，我说了4种，后来网上搜也只有五种呀

## 类方法，代码，静态变量，实例，实例变量，局部变量分别存到哪里

类方法

代码

静态变量：方法区

实例

实例变量

局部变量：虚拟机栈中的栈桢中

## JVM怎么表示一个对象

1. GC三个算法，分代收集讲下

## https握手流程

## http格式

## http1.0 和 2.0

## 保持长连接的是那个字段（keep-alive）

## fork（）函数讲下，又结合项目来了一次

## 线程调度算法，linux采用哪种

## 进程线程区别，进程通信

1进程是资源分配的最小单位，线程是程序执行的最小单位（资源调度的最小单位）  
2、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。  
3、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。  
4、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。

1. 数据库聚簇索引，和非聚簇索引的区别

## 内连接，外（左右）链接，全连接的区别

join(inner join):也叫等值连接：返回两张表都满足条件的部分

left join (left outer join):取左边的表的全部，右边的表按条件，符合的显示，不符合则显示null

right join(right outer join) : 取右边的表的全部，左边的表按条件，符合的显示，不符合则显示null

全连接: 完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

## join on 后and条件和where后加and条件区别

1. 索引结构（B+树）讲下，这次没问HASH的区别，但是复习过，当然得细细讲下啦

## 算法题：字符串转Int，如果越界就返回0

## 场景题目：设计一个短域名服务：短信存不了太长网站，需要弄成短域名，你该如何设计一个服务，可以为全国的网址服务。我说的将所有网址存数据库，用hashcode来当短域名，如果冲突就拼接随机数，表大就分表，当时忘记说再加缓存了，笨

1. 3次握手四次挥手

首先 服务器端 处于 LISTEN（监听）状态，等待客户的连接请求。

客户端 向 服务器端 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。

服务器端收到连接请求报文，如果同意建立连接，则向客户端发送连接确认报文，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 y。

客户端收到服务器端的连接确认报文后，还要向服务器端发出确认，确认号为 y+1，序号为 x+1。

服务器端收到客户端的确认后，连接建立。

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。

客户端发送的连接请求如果在网络中滞留，那么就会隔很长一段时间才能收到服务器端发回的连接确认。客户端等待一个超时重传时间之后，就会重新请求连接。但是这个滞留的连接请求最后还是会到达服务器，如果不进行三次握手，那么服务器就会打开两个连接。如果有第三次握手，客户端会忽略服务器之后发送的对滞留连接请求的连接确认，不进行第三次握手，因此就不会再次打开连接。

1. 数据库的隔离级别

## 索引，复合索引结构

## sql题，写了个连表查询外加模糊查询

## 算法：镜像二叉树

## 算法：将List转成tree：（当前节点id，节点名字，父节点id），（当前节点id，节点名字，父节点id），（当前节点id，节点名字，父节点id）=》转成一个树，说自己定义数据结构，设计用例，要能编译运行，我用hashmap存，然后写到差不多了他说不用写了

### 字节面经

你会什么，擅长什么

按熟悉程度排序，计算机基础课你哪些学得不错

（一顿闲聊）

## 操作系统

1.什么是操作系统

2.你觉得作为一个软件开发人员，需要知道哪些关于操作系统的知识

3.如何实现系统调用

4.内核态与用户态的区别

5.物理地址和虚拟地址是如何转化的，其中涉及了哪些硬件？

## 计算机网络

1.在一个子网内，你不知道外网的IP，如何访问到外网？（大概是这样，具体记不太得了。。这题我乱说的）

2.假如你要传一个很大的文件，怎么传？

3.TCP如何实现可靠传输，具体解释一下每一点

4.UDP不面向连接是什么意思？不连接怎么传递？UDP传输不可靠怎么办？

## 数据结构与算法

1. 丢了一个链接，让写快速排序

（大约5分钟写完）

2.快速排序稳定吗？为什么

3.哪个常见排序是稳定的，为什么

4.算法复杂度了解吗

5.大O复杂度具体指的是什么？

6.有哪些常见的时间复杂度，各举一个例子

7.多举几个O(1)时间复杂度的例子

8.动态规划了解吗，主要是用来解决什么问题？适用于什么场景

9.来写个背包吧

问了最简单的0/1背包，一些东西的weights，一个capacity，问最多可以装几个物品

（3分钟写完了？）

讲一下你这么写为什么是对的  
  
方向是后台。  
大概感受:主要考察计算机基础（数据结构，数据库，os，网络）以及算法能力，算法难度在leetcode medium这样子。  
  
一面:  
1.开场自我介绍  
2.介绍项目  
3.tcp/udp区别应用场景  
4.timewaitclosewait  
5.cpu负载命令（linux）  
6.进程查看命令。  
7.jvm调优思路。  
8.垃圾回收机制。  
9.算法题，找出一串字符串的最长子串，要求最多两个不同的字母。  
  
二面:  
1.自我介绍  
2.为什么选择字节跳动。  
3.怎么自学的。  
4.jvm垃圾回收机制和内存分配。  
5.cpu调度算法（时间片轮转）  
6.业务场景模拟题（记不清了）  
7.算法（搜索旋转排序数组，leetcode的）  
  
三面:  
1.自我介绍。  
2.单点登录系统实现（项目）  
3.自我评价（长处短处）  
4.职业规划  
5.github你用\*\*\*什么  
6.tcp全家桶不展开了。  
7.多路复用io。  
8.文件分块断点续传模拟题。  
9.算法:给一个n，要求顺时针打印（剑指的一道）  
1 2 3  
8 9 4  
7 6 5

## 美团：一面

1. 1分钟自我介绍

2．项目在哪里做的

3. 项目具体做了什么

4. 使用redis提升了多少

5. java是自学的

Java内存模型

6. String 类型常量应该存在

7. hashMap

8.用过Spring吗 用到Spring什么特性 结合项目说一下 依赖反转的好处

9.线程用过 怎么定义一个线程

尊敬的面试官，您好！我是于楠，下午3点您面试的我。面试结束后我回忆了一下今天面试答的不好的地方，去网上查了一下。再次感谢您今天对我的面试！

1.在多线程中可以用Join()方法控制3个线程顺序执行

2.Synchronized加在static 方法上和不加在static方法上不会产生互斥

3.单例模式的应用场景可以是网站的计数器，windows的任务管理器。

4.用抽象类而不用接口的场景是定义了一组接口，但又不想强迫每个实现类都必须实现所有的接口。可以用abstract class定义一组方法体，甚至可以是空方法体，然后由子类选择自己所感兴趣的方法来覆盖。

5. HashMap中的hash碰撞问题，最坏的情况下，所有的key都被分配到了同一个桶，这时map的put和get时间复杂度都是O(n)。

## 小米

1. HashMap Hashtable concurrenthashmap区别和联系
2. 数组，链表，hashmap 区别和联系
3. Hashmap resize() 方法怎么实现的
4. Sleep 为什么在Thread中?wait为什么在Object中?
5. Lock()源码看过吗

object类有哪些方法

终止线程的方法

读文件的时候会不会释放cpu

## 猿辅导

1、进程与线程的区别

2、堆和栈的不同

3、TCP与UDP的区别

4、TCP怎么保持可靠性

5、判断链表是否是回文链表

public boolean isPalindrome(ListNode head) {

if (head == null || head.next == null) {

return true;

}

ListNode result = reverseList(head);

while (head != null ) {

if (head.val != result.val) {

return false;

}

head = head.next;

result = result.next;

}

return true;

}

public ListNode reverseList(ListNode head) {

if (head == null || head.next == null) {

return head;

}

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

while (head != null) {

stack.push(head);

head = head.next;

}

ListNode res = new ListNode(-1);

ListNode temp = res;

while (!stack.isEmpty()) {

temp.next = new ListNode(stack.pop().val);

temp = temp.next;

}

return res.next;

}

1. 判断二叉树是否是二叉搜索树

**int** last = Integer.***MIN\_VALUE***;

**public** **boolean** isTree(TreeNode root){

**if**(root == **null**)

**return** **true**;

**if**(!isTree(root.left))

**return** **false**;

**if**(root.val <= last)

**return** **false**;

last=root.val;

**if**(!isTree(root.right))

**return** **false**;

**return** **true**;

}

7、栈的时间复杂度、空间复杂度

8、python是什么语言

9、is 和 == 的区别

10、蓄水池算法

int[] ls = {3,8,2,1,9,5,3};

int l = ls.length;

int i = 0;

int j = l - 1;

int max = 0;

int s;

while(i<j) {

s = (j-i)\*Math.min(ls[i], ls[j]);

if(ls[i]<ls[j])

i++;

else

j--;

if(s>max) {

max = s;

}

}

System.out.println("最大蓄水量为"+max);

**二面：**

1、HTTP的GET与POST的区别

2、TCP的keep alive字段

3、HTTP的keep alive字段

4、二叉树的宽度

5、python多线程、GIL锁

完全二叉树的最大深度与节点个数

public int getLeftHeight(TreeNode root) {

int count = 0;

while (root != null) {

count++;

root = root.left;

}

return count;

}

// 获取右子树的高度（其实是最右侧分支的高度）

public int getRightHeight(TreeNode root) {

int count = 0;

while (root != null) {

count++;

root = root.right;

}

return count;

}

public int countNodes(TreeNode root) {

if (root == null) {

return 0;

}

int leftHeight = getLeftHeight(root);

int rightHeight = getRightHeight(root);

if (leftHeight == rightHeight) {

// 表示是满二叉树，二叉树的节点数直接由公式2^n-1得到

// leftHeight即为层数， 1 << leftHeight使用位运算计算2^leftHeight，效率更高

// 注意(1 << leftHeight) - 1 的括号必须有！！

return (1 << leftHeight) - 1;

} else {

// 若该二叉树不是满二叉树，递归的调用该方法，计算左子树和右子树的节点数

return countNodes(root.left) + countNodes(root.right) + 1;

}

}