

2018 秋招 java 后台妹子面经

作者：倾听潮汐

1、关于我.....	2
1.1 自我介绍.....	2
1.2 复习过程.....	2
2、面经--排名按面试时间.....	4
3、 知识点.....	9
3.1. 问到过的问题（包含答案）.....	9
3.2 没问到过的问题（包含答案）.....	64
4、 后记.....	100

1、关于我

1.1 自我介绍

本人妹子，985 硕士，211 本科，专业都是软件工程，一直投的是 java 后台开发，只投过一次网易的测试，技术不是大牛，但是比较努力。实验室没有项目，so 项目经验是 0，在去年这个时候看到实验室师兄找工作的艰难，因此开始复习的时间比较早。最开始先看的 java 基础，看的毕向东的视频，后面就看框架视频，后来也看过尚硅谷的视频，都是在网上找的免费的。《剑指 offer》刷了一遍，有些重点的题一定要滚瓜烂熟。《大话设计模式》《mysql 必知必会》《程序员面试金典》《java 并发编程实战》《计算机操作系统第三版》《计算机网络》都看了一遍，《深入理解 java 虚拟机》看了两遍。《java 编程思想》《Java 数据结构和算法中文第二版》《Spring 技术内幕：深入解析 Spring 架构与设计原理(第 2 版)》《Struts2 in action 中文版》《tcp 详解卷一》挑的重点看。现在觉得最后悔的是 leetcode 没有刷，有时间一定要刷，有时间一定要刷，有时间一定要刷，对解题思路很有帮助。因为没有项目经验，就在网上找了一个项目，对于项目中的问题一定要很熟，用到的技术不仅要会用，还要理解底层，反正每次都会被问，感觉面试官对什么高并发的东西挺感兴趣的。我主要是想找成都的公司，但是最开始海投的时候，不管公司在哪里，我都投了，反正投的公司有几十家，我认为多面几家公司可以多增加一点面经。多关注自己学校的 bbs，有很多内推信息（很多公司内推可以免笔试，这是很爽的！！），我的前两个 offer 都是内推的（美团和携程），因为美团是成都的，自己也比较满意，所有后面很多公司的面试都没有去，但是建议还投的时候还是都投，先拿一个心里有底。

1.2 复习过程

说明：全部自学，没有项目经验，没有实习经历。

研二的时候看见师兄他们找工作很困难，在研二的时候，差不多 5、6 月份就把 java 基础视频看了一遍，在牛客网上刷了一点题，后来要写论文，java 复习就不了了之了。正式复习是从今年 4、5 月份开始的，还是先看 java 的基础视频，把视频的所有东西都挨着敲了一遍，当然看到后面的时候就会发现前面的也忘得差不多了，这是肯定的，解决办法就是再看。看了基础视频之后就开始刷《剑指 offer》，他是 c++ 写的，然后在牛客网上也可以刷，可以用 java 写了，检查编译什么的，我就在上面把所有题都刷了一遍。在刷这个的时候，也开始看 spring 和 springMVC 的视频，我都是在尚硅谷上下的免费的。然后看了《深入理解 Java 虚拟机：JVM 高级特性与最佳实践（最新第二版）》，每一章都要认真看，每一章都是重点。Jvm 调优的那部分，我只是看了，没有自己实践，关于 OOM，我自己实验了一次，面试的时候被问到过 2、3 次。然后看了《java 程序员面试宝典》。在看这些的过

程中，也一直在牛客网上刷题，我刷的主要都是和 **java** 相关的题目，目的是加深记忆。有时间一定要刷刷 **LeetCode** 的题，感觉刷了和没刷差别真的很大，刷了之后笔试通过率会高很多。然后上篇面经中提到的其他书都挨着挨着看。

关于 **项目**，我是在慕课网上找的，我跟着视频都敲了一遍。面试的时候会问到很多自己从来没有考虑过的问题，每次问到不会的问题之后，就记下来，下来查好了，可能下次还会问。慕课网上其他视频也还不错，有时间都可以看看。

关于 **投递简历**，我是从 7 月份开始关注河畔的就业信息栏的，里面有很多的信息，几乎每天早上来实验室第一件事就是刷河畔，看有没有招聘信息，我是不论公司大小，不论公司在哪里，我都投。投内推一定要抓紧，我投了成都美团内推之后的 2、3 天，我的室友想投，然后就被告知美团成都已经满了，只有其他城市的，所以时机还是很重要。给别人邮箱发邮件的时候，一定要写好你的基本信息，包括姓名、电话、意向岗位、性别、意向城市等。校招的时候填简历真的很恶心，东西很多，几乎每个公司都要填。我建议可以拿一个本子或者自己做一个 **Excel** 来记录投的公司和流程，这样心里有数一些。

关于 **面试**，一般内推都是电话面试，只遇到过一次视频面试。开始还是很紧张，后来面多了就好了。我内推的时候电话面试没有被要求过写代码的，但是快排说了很多次，都是问的基础，问的很深。有些面试官，比如阿里的就是不把你问到你说不出话来，他是不会停的。有的面试官感觉有一个问问题的单子，然后他就照着那个单子在问。每一次面试都是一个查漏的机会，都会出现很多自己不会的问题，包括一些你认为你会的问题，面试官问深了，你还是不会。每次面试完，我都把问题记下来，把不会的都查一遍。

关于 **其他计算机基础**，网络，3 次握手，4 次分手的图一定要会。**https** 的过程，**tcp/ip**，**http** 相关问题被问的也很多。操作系统，问过计算机的缓存，其他的没有什么映像了。数据结构，排序是基础，要滚瓜烂熟，推荐一个网址 <http://blog.csdn.net/hguisu/article/details/7776068/>，链表相关（判断有没有环之类的问题），二叉树（**B+**，**B***，**B-**，平衡二叉树），图（深度遍历，广度遍历）都要会。算法里面贪心和动态规划要会，感觉很多代码题都涉及这些，这个只有多刷题，多写代码，木有其他捷径。推荐一个左神的书《程序员代码面试指南—IT 名企算法与数据结构题目最优解》，我没有买这本书，主要是我都是很晚才知道左神的，听了他的几次课，感觉受益匪浅，如果我很早就知道，我应该会买他的书看的。关于 **java** 底层，多看 **jdk 源码**。

复习是一个很乏味的过程，贵在坚持！！

关于找工作，自己的实力当然很重要，但我觉得心态和运气也很重要。当你被某个公司拒了之后，也不用灰心，很多面试都是玄学，我觉得题目都回答上了，然而还是被刷了。保持好的心态，在加上好好复习，再来点运气，完美！！好的 **offer** 会有的！！

分割线-----

2、面经--排名按面试时间

只记得这些公司了

蚂蚁金服-内推的，面了 3 面技术，1 面 hr（在池子中泡了 3 个多月，最后还是 gg 了）

一面

这是我秋招的第一次面试，别提多紧张了，声音都在发抖，自我介绍都不知道说什么

1. 自我介绍（说的很乱）
2. 序列化的底层怎么实现的
3. synchronized 的底层怎么实现的
4. tomcat 集群怎么保证同步
5. 了解 nosql 吗？redis？
6. 怎么解决项目中超卖的问题
7. int 的范围
8. 有什么想问的？我问对于没有实现经验和实际项目经验的，阿里会考虑吗？面试官说对于应届生，阿里还是最看重基础。我：你们平时怎么学习技术？面试官：从实际项目中学习。。。

面完之后，面试官说我基础可以，给我过，我能走多远就不知道了，反正很谢谢这位面试官，给我的秋招增加了很多信心，毕竟是阿里的面试官说我基础可以。
偷笑.jpg

二面

清楚的记得二面是在晚上 7 点左右，刚走到教研室楼下，看见是杭州的电话，顿时激动了，然后一接，果然是阿里，因为下面很吵，我就一口气跑到 5 楼，然后喘着粗气进行自我介绍。。尴尬

1. 问了项目中超卖的问题怎么解决
2. 你熟悉什么数据结构
3. 说说快排，我说了快排实现的过程，相当于口述代码，然后问了复杂度
4. int 的范围，我说的是 2 的⁻³¹次方到 2 的³¹次方-1，面试官说具体点，我就不知道了，后来查到是-2147483648~2147483647
5. 乐观锁 vs 悲观锁
6. gc
7. concurrenthashmap 分段锁的细节
8. 设计模式怎么分类，每一类都有哪些
9. 并发包里了解哪些
10. b 树，b+树，b*树
11. 字节与字符的区别

三面

一二面只隔了 2,3 天，三面和二面隔了 2 周，当时一直以为自己挂掉了，接到电话也很突然

1. 自我介绍
2. 项目
3. 知道哪些服务器？答：JBoss，Apache，weblogic。问：区别？
4. java 有什么后端技术
5. springIOC 优点
6. jdk 动态代理 vs cglib 动态代理，他们底层分别怎么实现的
7. synchronizedMap 知道吗？他和 concurrentHashMap 分别使用于什么场景？
8. https 过程？公钥能用公钥解吗？在客户端抓包，看到的是加密的还是没加密的？当时乱猜的加密，面试官说不不对，是没加密的
9. 描述一下 java 线程池。
10. 怎么保证 redis 和 db 中的数据一致
11. 设计模式怎么用到项目中？
12. 类加载

hr 面

不记得 3 面和 hr 面之间间隔多久了，也是晚上 6,7 点收到的电话

1. 自我介绍
2. 有什么优点？
3. 有什么缺点？
4. 项目中遇到的困难？怎么解决
5. 意向城市

时间已经过去太久了，只记得这么多，可能有些问题不是阿里面试官问的，但都是在面试过程中遇到的，希望能帮到更多的人。

携程（offer）内推-只有一次视频面

1. 链表的定义
 2. 怎么实现链表翻转
 3. 怎么判断链表是否有环
 4. 二叉平衡树，怎么用一维数组存储
 5. 讲讲 jvm 分区
 6. 讲讲 jvm gc
 7. 怎么求数组的最大子序列和
 8. final 关键字 4 种用法
 9. sleep 用法
 10. await 关键字
 11. 线程池
 12. spring ioc, aop 以及其优点
 13. 栈和堆的定义
- 现在记得的只有这么多了。

美团成都提前批（收到 offer）

在河畔上看到的内推消息，然后就发了邮件，3、4 天后，收到邮件让去公司面试，在天府三街附近。

一面

先给了一个 A4 纸，上面写满了题，然后就做题。做完题就自我介绍，说项目，问项目里的东西（自己一定要对项目里用的东西很熟，还要明白底层原理，我用了 redis，面试官就问 redis 怎么实现），然后就问 java 基础，面了 40 分钟吧，然后就让我在外面等，过了一会，二面面试官来了。

试卷题目，只记得部分

一、简答题

1. 浏览器访问一个网址的时候都有哪些过程（还要了解 DNS 查找的过程）
2. tcp 三次握手，四次分手
3. 线程池
4. 你了解的设计模式（面试官说什么单例模式就不用说了）

二、编程题

1. 二分查找
2. 树的中序遍历

三、智力题

给你一个 5L 和 3L 桶，水无限多，怎么倒出 4L

二面

面试官问我觉得一面面的怎么样，我。。。然后就看我做的试卷，我竟然连二分查找都写错，面试官说不行，然后我改了改就好了，然后面试官就问还有什么可以优化的地方吗？我只说出来一个。。然后又是自我介绍，说项目，问项目，问基础。差不多也是 40 分钟。

三面

过了几天，收到 3 面邮，应该是部门 boss，也是自我介绍，然后出了两个题 1) 一个 $n \times n$ 的矩阵，按副对角线打印 2) 4 个瓶盖换 1 瓶酒，要和 150 瓶酒，他自己最少多少瓶？然后问了职业规划。感觉有 1 个小时。

Hr 面

北京的电面，就随便聊聊。

现在只记得这么多了，祝大家都拿到心仪的 offer！

58 现场面试 (offer)

两面技术，一面 hr，然后就让我回来了

技术一面

1. 自我介绍
2. 根据 58 笔试题，看着问的
服务器之间怎么通信，写了一个链表反转，项目中 nginx 怎么配置的，什么硬件条件可以实现多线程，什么情况下多线程才能发挥作用，mysql 索引底层是什么，b 树和 hash 应用场景

技术二面

讲项目，什么是乐观锁，以后的规划

hr 面

1. 项目流程
2. 项目中遇到的问题，怎么解决
3. 自己最大的缺点
4. 喜欢和什么样的人合作，不喜欢和什么样的人合作
5. 了解 58 吗

- 6.找工作有区域限制吗？家人支持吗
- 7.讲了讲学生活动，一个活动怎么组织的
- 8.有其他公司 offer 吗？

中兴（一面挂）

所有人在一个大厅里面，很多面试官，每个人两个面试官。

- 1.自我介绍
- 2.手撕代码 1) 怎么写多线程 2) 1, 2, 3, 4 组成的不重复三位数，都用代码写出来
- 3.问数据库会不会，我说可以，他问会什么，我说索引，然后让写 sql，让我建一个表，我说这种我都是百度的，然后让我写了两个查询
- 4.问项目
- 5.会 linux 吗？答不怎么用，会常用命令
- 6.愿不愿意转测试，答不愿意，问看不起测试吗？答不是，只是自己喜欢研发
- 7.意向城市，答成都，只有成都吗？答，嗯

滴滴（二面挂，现场面）

一面

- 1.一来就手撕代码，给你三个 string s1, s2 和 s3, 判断 s3 能不能由 s1 和 s2 组成，s1 和 s2 内部元素相对位置不能变，比如 a 和 bc, 可以组成的有 abc bac bca, 我写的暴力的，在面试官的指导下改为了动态规划，不过最后还有不对的地方
- 2.自我介绍
- 3.hashCode 和 equals 区别
- 4.实现一个类，要求要放在 hashset 里
- 5.你了解哪些设计模式？我说了一些，然后让写了一个单例，我用枚举写了一个，感觉面试官不懂，说再写一个，然后写了一个双重检查的
- 6.你还有什么要问的？

二面

- 1.说项目，被批的很惨
- 2.写了四次分手的图，然后问若客户端和服务端之间，1s 会发生 5000 到 6000 次短链接，会发生什么问题
- 3.手撕代码，不断优化，最后应该是对了
- 4.你有什么问的吗？

招银网络（offer，现场面）

一面

自我介绍

问项目

用过 log4j 吗？把他输出到数据库用过吗？

一致性 hash，怎么解决 hash 冲突

除了写代码，有什么爱好

怎么保证写的代码出错少

写自己项目中类的函数声明

有没有遇到什么有印象的 bug

二面

全是项目，没什么好说的，反正问的很深，被怼死了

hr 面

自我介绍

得过哪些奖

家在哪里

选公司的标准

抗压的能力

父母是做什么的，具体一点

父母对于你找工作的意见

对开发工作的要求，我本来说的不想做 android，后来我问他们公司业务时，听 hr 的意思就是，如果一个项目用 c 写，你就用 c，如果用 c#，你就用 c#，如果是 ios，你又用 iso，具体用什么语言都不确定的，然后我就说我什么都愿意做。。。。就酱

华为（offer，现场面）

一面

1. 自我介绍
2. 自己实现一个链表，写了以后也不说对不对
3. 说项目，问了很多，还让画框架图
4. 写生产者消费者代码，我用的阻塞队列
5. 说我不适合研发，让转测试或资料
6. 说发的论文的算法

二面

1. 自我介绍
2. 平时怎么提高自己的编程能力，我说刷题，他问频率，我说不忙的时候一天一次，然后他说既然刷那么多次，怎么笔试成绩还这么低。。。感觉自己给自己挖了一个坑
3. 意向城市
4. 自己的缺点，怎么改进
5. 在项目中遇到的最大的困难
6. 在课题中遇到的最大困难
7. 你们 4 人合作项目是怎么分工的
8. 你有什么想问的

贝贝网

一面

1. 自我介绍
2. java 同步机制有哪些
3. equals 和 hashCode 区别和联系

4. 进程和线程讲讲
5. equals 和 == 区别
5. 代码题，没让写，只说了思路，有 n 个长方形，每个的长宽为 x, y ，从下往上堆，要求上面的长宽小于下面的长宽，求最多可以堆几层
6. 在浏览器输入一个网址到得到页面的过程，越详细越好

二面

1. 自我介绍
2. 问项目，问的很深，还有什么可以改进的
3. 同 1 面的 6 题
4. 进程 vs 线程
5. 平时怎么学习
6. 写 sql

hr 面

hr 和二面在一起，先二面然后 hr 面

1. 你本来不是杭州人，愿意来杭州工作吗？为什么
2. 有关注杭州其他企业吗
3. 周末平时干什么
4. 还有什么问题

大疆（只记得 3 面）

1. 自我介绍
2. 说项目，问项目
3. 讲讲你的家庭
4. 为什么来大疆
5. 面过其他公司吗
6. 如果主管没给你安排事情，又没有项目，你会干些什么
7. 遇到的挫折，我说了一个，然后他说这不算挫折。。
8. 了解大疆吗
9. 有什么想问的

3、知识点

这里包含我遇到过的问题和没有遇到过的问题，遇到的问题你一定要看，因为你很可能也会遇到，没遇到过的问题你也要看，因为这都是我学习时记得笔记，都是和 java 相关的基础，都是别人面试时遇到的。虽然很多，但是一定要有耐心，找工作本来就是一个比耐力的过程，当自己最后拿到满意的 offer 时，就会发现以前的所有努力都是值得的。

3.1. 问到过的问题（包含答案）

JVM 有哪些分区?答案：程序计数器，Java 虚拟机栈，本地方法栈，堆，方法区（Java

栈中存放的是一个一个的栈帧，每一个栈帧对应一个被调用的方法。栈帧包括局部变量表，操作数栈，方法的返回地址，指向当前方法所属的类的运行时常量池的引用，附加信息）。JVM 中只有一个堆。方法区中重要的是运行时常量池。

mysql 使用的引擎？

- A:1)MyIsam 不支持事务，适用于选择密集型，插入密集型，mysql 默认的引擎
- 2)innodb 适用于更新密集型，支持事务，自动灾难恢复，行级锁，外键约束
- 3) memory 出发点是速度 采用的逻辑存储介质是内存
- 4) merge 一组 myisam 表的组合

linux 查看文件内容的命令

Cat: 从第一行开始显示内容，并将所有内容输出
Tac: 从最后一行开始显示内容，并将所有内容输出
Head: 只显示前几行
Tail: 只显示后几行
nl: 和 cat 一样，只是 nl 要显示行号

线程池 ThreadPoolExecutor

corepoolsize: 核心池的大小，默认情况下，在创建了线程池之后，线程池中线程数为 0，当有任务来之后，就会创建一个线程去执行任务，当线程池中线程数达到 corepoolsize 后，就把任务放在任务缓存队列中。

Maximumpoolsize:线程池中最多创建多少个线程。

Keeplivetime: 线程没有任务执行时，最多保存多久的时间会终止，默认情况下，当线程池中线程数>corepoolsize 时，Keeplivetime 才起作用，直到线程数不大于 corepoolsize。

workQueue:阻塞队列，用来存放等待被执行的任務

threadFactory:线程工厂，用来创建线程。

继承关系



线程池的状态

- 1.当线程池创建后，初始为 running 状态
- 2.调用 shutdown 方法后，处 shutdown 状态，此时不再接受新的任务，等待已有的任务执行完毕
- 3.调用 shutdownnow 方法后，进入 stop 状态，不再接受新的任务，并且会尝试终止正在执行的任务。
- 4.当处于 shutdown 或 stop 状态，并且所有工作线程已经销毁，任务缓存队列已清空，线程池被设为 terminated 状态。

当有任务提交到线程池之后的一些操作：

1. 若当前线程池中线程数 < corepoolsize，则每来一个任务就创建一个线程去执行。
2. 若当前线程池中线程数 ≥ corepoolsize，会尝试将任务添加到任务缓存队列中去，若添加成功，则任务会等待空闲线程将其取出执行，若添加失败，则尝试创建线程去执行这个任务。
3. 若当前线程池中线程数 ≥ Maximumpoolsize，则采取拒绝策略（有 4 种，1）abortpolicy 丢弃任务，抛出 RejectedExecutionException 2）discardpolicy 拒绝执行，不抛异常 3）discardoldestpolicy 丢弃任务缓存队列中最老的任务，并且尝试重新提交新的任务 4）callerrunspolicy 有反馈机制，使任务提交的速度变慢）。

生产者-消费者代码-用 blockingqueue 实现

生产者消费者的示例代码：

生产者：

Java代码



```
1. import java.util.concurrent.BlockingQueue;
2.
3. public class Producer implements Runnable {
4.     BlockingQueue<String> queue;
5.
6.     public Producer(BlockingQueue<String> queue) {
7.         this.queue = queue;
8.     }
9.
10.    @Override
11.    public void run() {
12.        try {
13.            String temp = "A Product, 生产线程: "
14.                + Thread.currentThread().getName();
15.            System.out.println("I have made a product:"
16.                + Thread.currentThread().getName());
17.            queue.put(temp); //如果队列是满的话，会阻塞当前线程
18.        } catch (InterruptedException e) {
19.            e.printStackTrace();
20.        }
21.    }
22.
23. }
```

消费者：

Java代码

```
1. import java.util.concurrent.BlockingQueue;
2.
3. public class Consumer implements Runnable {
4.     BlockingQueue<String> queue;
5.
6.     public Consumer(BlockingQueue<String> queue) {
7.         this.queue = queue;
8.     }
9.
10.    @Override
11.    public void run() {
12.        try {
13.            String temp = queue.take(); //如果队列为空，会阻塞当前线程
14.            System.out.println(temp);
15.        } catch (InterruptedException e) {
16.            e.printStackTrace();
17.        }
18.    }
19. }
```

测试类:

Java代码  

```
1. import java.util.concurrent.ArrayBlockingQueue;
2. import java.util.concurrent.BlockingQueue;
3. import java.util.concurrent.LinkedBlockingQueue;
4.
5. public class Test3 {
6.
7.     public static void main(String[] args) {
8.         BlockingQueue<String> queue = new LinkedBlockingQueue<String>(2);
9.         // BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
10.        //不设置的话, LinkedBlockingQueue默认大小为Integer.MAX_VALUE
11.
12.        // BlockingQueue<String> queue = new ArrayBlockingQueue<String>(2);
13.
14.        Consumer consumer = new Consumer(queue);
15.        Producer producer = new Producer(queue);
16.        for (int i = 0; i < 5; i++) {
17.            new Thread(producer, "Producer" + (i + 1)).start();
18.
19.            new Thread(consumer, "Consumer" + (i + 1)).start();
20.        }
21.    }
22. }
```

打印结果:

Text代码  

```
1. I have made a product:Producer1
2. I have made a product:Producer2
3. A Product, 生产线程: Producer1
4. A Product, 生产线程: Producer2
5. I have made a product:Producer3
6. A Product, 生产线程: Producer3
7. I have made a product:Producer5
8. I have made a product:Producer4
9. A Product, 生产线程: Producer5
10. A Product, 生产线程: Producer4
```

GC

对象是否存活

- 1) 引用计数法 缺点: 很难解决对象之间循环引用的问题。
- 2) 可达性分析法 基本思想: 通过一系列的称为“GC roots”的对象作为起始点, 从这些节点, 开始向下搜索, 搜索所走过的路径称为引用链, 当一个对象到 GC root 没有任何引用链相连(用图论的话来说, 就是从 GC roots 到这个对象不可达), 则证明此对象是不可用的。

可作为 GC roots 的对象

- 1) java 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 2) 方法区中类的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 引用的对象

引用强度 强引用>软引用>弱引用>虚引用

任何一个对象的 `finalize()` 方法都只会被系统调用一次。

若对象在进行可达性分析后发现没有与 GC roots 相连接的引用链,那么他将会被第一次标记并进行一次筛选,筛选的条件是该对象是否有必要执行 `finalize()`方法,当对象没有重写 `finalize()`方法或者 `finalize()`方法已经被虚拟机调用过,虚拟机将这两种情况都视为没必要执行。

若该对象被判定为有必要执行 `finalize` 方法,则这个对象会被放在一个 F-Queue 队列, `finalize` 方法是对象逃脱死亡命运的最后一次机会,稍后 GC 将对 F-queue 中的对象进行第二次小规模标记,若对象要在 `finalize` 中成功拯救自己—只要重新与引用链上的任何一个对象建立关联即可,那么在第二次标记时他们将会被移出“即将回收”集合。

`Finalize` 方法不是 c 或 c++的析构函数。

停止-复制算法: 它将可用内存按照容量划分为大小相等的两块,每次只使用其中一块。当这一块的内存用完了,则就将还存活的对象复制到另一块上面,然后再把已经使用过的内存空间一次清理掉。**商业虚拟机:** 将内存分为一块较大的 `eden` 空间和两块较小的 `survivor` 空间,默认比例是 8:1:1,即每次新生代中可用内存空间为整个新生代容量的 90%,每次使用 `eden` 和其中一个 `survivor`。当回收时,将 `eden` 和 `survivor` 中还存活的对象一次性复制到另外一块 `survivor` 上,最后清理掉 `eden` 和刚才用过的 `survivor`,若另外一块 `survivor` 空间没有足够内存空间存放上次新生代收集下来的存活对象时,这些对象将直接通过分配担保机制进入老年代。

标记-清除算法: 缺点 1) 产生大量不连续的内存碎片 2) 标记和清除效率都不高

标记-清理算法: 标记过程和“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除,而是让 all 存活对象都向一端移动,然后直接清理掉端边界以外的内存。

分代收集: 新生代 停止-复制算法 老年代 标记-清理或标记-清除

垃圾收集器,前 3 个是新生代,后 3 个是老年代

1) **serial 收集器:** 单线程(单线程的意义不仅仅说明它会使用一个 cpu or 一条垃圾收集线程去完成垃圾收集工作,更重要的是在它进行垃圾收集的时候,必须暂停其他 all 工作线程,直到他收集结束)。对于运行在 `client` 模式下的虚拟机来说是个很好的选择。停止-复制

2) **parNew 搜集器:** `serial` 收集器的单线程版本,是许多运行在 `server` 模式下的虚拟机首选的新生代收集器。停止-复制

3) **parallel scavenge:** 目标 达到一个可控制的吞吐量,适合在后台运算,没有太多的交互。停止-复制。

4) **serial old:** `serial` 的老年代版本,单线程,标记-清理

5) **parallel old:** `parallel scavenge` 老年代的版本,多线程 标记-清理

6) **cms 收集器:** 一种以获取最短回收停顿时间为目标的收集器“标记-清除”,有 4 个过程初始标记(查找直接与 gc roots 链接的对象) 并发标记(`tracing` 过程) 重新标记(因为并发标记时有用户线程在执行,标记结果可能有变化) 并发清除 其中初始标记和重新标记阶段,要“stop the world”(停止工作线程)。优点: 并发收集,低停顿 缺点: 1) 不能处理浮动垃圾 2) 对 cpu 资源敏感 3) 产生大量内存碎片 {每一天具体的详解请看《深入理解 Java 虚拟机: JVM 高级特性与最佳实践》}

对象的分配: 1) 大多数情况下,对象在新生代 `eden` 区中分配,当 `Eden` 区中没有足够的

内存空间进行分配时，虚拟机将发起一次 minor GC {minor gc: 发生在新生代的垃圾收集动作，非常频繁，一般回收速度也比较快 full gc: 发生在老年代的 gc} 2) 大对象直接进入老年代 3) 长期存活的对象将进入老年代 4) 若在 survivor 空间中相同年龄 all 对象大小的总和 > survivor 空间的一半，则年龄 >= 改年龄的对象直接进入老年代，无须等到 MaxTenuringThreshold（默认为 15）中的要求。

空间分配担保

在发生 minor gc 前，虚拟机会检测老年代最大可用的连续空间是否 > 新生代 all 对象总空间，若这个条件成立，那么 minor gc 可以确保是安全的。若不成立，则虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败。若允许，那么会继续检测老年代最大可用的连续空间是否 > 历次晋升到老年代对象的平均大小。若大于，则将尝试进行一次 minor gc，尽管这次 minor gc 是有风险的。若小于或 HandlePromotionFailure 设置不允许冒险，则这时要改为进行一次 full gc。

单例模式

1) 双重检测

2) 枚举实现

```
public class EnumSingleton {
    public static void main(String[] args) {
        danli one=danli.INSTANCE;
        danli two=danli.INSTANCE;
        System.out.println(one.getString());
        one.test();
        one.setString("wo shi one");
        //System.out.println(one.getString());
        System.out.println(one==two);//true
        System.out.println(two.getString());
    }
}
//用enum实现的单例设计模式
enum danli{
    INSTANCE;

    private String string=name();
    // private String string; 上面这句也可以这样写，这样string就没有默认的值

    public String getString() {
        return string;
    }
    public void setString(String string) {
        this.string = string;
    }
    public void test(){
        System.out.println("测试开始");
    }
}
```

3) 静态内部类的

Out of Memory

1. 程序计数器是唯一一个在 Java 虚拟机规范中没有规定任何 oom 情况的区域。

2. 在 java 虚拟机规范中，对于 java 虚拟机栈，规定了 2 中异常，1) 若线程请求的栈深度 > 虚拟机所允许的栈深度，则抛出 Stack OverflowError 异常 2) 若虚拟机可以动态扩展，若扩展时无法申请到足够的内存空间，则抛出 oom 异常。

3. java 虚拟机栈为执行 java 方法，本地方法栈为虚拟机使用 native 方法服务，本地方法栈也会抛出 Stack OverflowError 和 oom。

4. Java 堆可以处于物理上连续的内存空间，只要逻辑上是连续的即可。可固定，可扩展。

若堆中没有内存完成实例分配，并且堆也无法再扩展，则会抛出 oom。

5.直接内存不是运行时数据区的一部分。

堆上的 OOM 测试

```
import java.util.ArrayList;

// -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails
// -Xms20M -Xmx20M 堆最大最小容量都是20M，则限制了堆不可以扩容
// 当堆上没有足够的空间分配对象，又不能扩容时，就抛出OutOfMemoryError

public class OutOfMemoryTest {
    static class OOMObject{}

    public static void main(String[] args){
        List<OOMObject> list=new ArrayList<OOMObject>();

        while(true){
            list.add(new OOMObject());
        }
    }
}
```

Java 内存模型和线程

每个线程都有个工作内存，线程只可以修改自己工作内存中的数据，然后再同步回主内存，主内存由多个内存共享。

下面 8 个操作都是原子的，不可再分的：

- 1) lock: 作用于主内存的变量，它把一个变量标识为一个线程独占的状态。
- 2) unlock: 作用于主内存的变量，他把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- 3) read: 作用于主内存变量，他把一个变量的值从主内存传输到线程的工作内存，以便随后的 load 操作使用。
- 4) load: 作用于工作内存的变量，他把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。
- 5) use: 作用于工作内存的变量，他把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值得字节码指令时将会执行这个操作。
- 6) assign: 作用于工作内存的变量，他把一个从执行引擎接收到的值付给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- 7) store: 作用于工作内存的变量，他把工作内存中一个变量的值传送到主内存中，以便随后 write 使用。
- 8) write: 作用于主内存的变量，他把 store 操作从工作内存中得到的变量的值放入主内存的变量中。

关键字 volatile 是轻量级的同步机制。

Volatile 变量对于 all 线程的可见性，指当一条线程修改了这个变量的值，新值对于其他线程来说是可见的、立即得知的。

Volatile 变量在多线程下不一定安全，因为他只有可见性、有序性，但是没有原子性。

线程的状态

在任意一个时间点，一个线程只能有且只有其中的一种状态

1) 新建：创建后尚未启动的线程处于这种状态。

2) 运行：包括了 OS 中 **Running** 和 **Ready** 状态，也就是处于次状态的线程可能正在运行，也可能正在等待 **cpu** 为他分配执行时间。

3) 无限期等待：处于这种状态的线程不会被分配 **cpu** 执行时间，要等待其他线程显示唤醒。以下方法会让线程进入无限期等待：1. 没有设置 **timeout** 的 **object.wait()** 方法 2. 没有设置 **timeout** 参数的 **Thread.join()** 方法 3. **LockSupport.park()**;

4) 有限期的等待：处于这种状态的线程也不会被分配 **cpu** 执行时间，不过无需等待被其他线程显示唤醒，而是在一定时间后，他们会由 **os** 自动唤醒 1. 设置了 **timeout** 的 **object.wait()** 方法 2. 设置了 **timeout** 参数的 **Thread.join()** 3. **LockSupport.parkNanos()** 4. **LockSupport.parkUnit()**

5) 阻塞：线程被阻塞了与“等待状态”的区别是：阻塞状态在等待获取一个排它锁，这个事件将在另外一个线程放弃这个锁的时候发生。等待状态在等待一段时间或者唤醒动作。

6) 结束：已终止线程的线程状态，线程已经结束执行。

synchronized (S) VS lock (L)

1) **L** 是接口，**S** 是关键字

2) **S** 在发生异常时，会自动释放线程占有的锁，不会发生死锁。**L** 在发生异常时，若没有主动通过 **unlock()** 释放锁，则很有可能造成死锁。所以用 **lock** 时要在 **finally** 中释放锁。

3) **L** 可以当等待锁的线程响应中断，而 **S** 不行，使用 **S** 时，等待的线程将会一直等下去，不能响应中断。

4) 通过 **L** 可以知道是否成功获得锁，**S** 不可以。

5) **L** 可以提高多个线程进行读写操作的效率。

mysql 索引使用的是 B+的数据结构

索引：用于提高数据访问速度的数据库对象。

优点：1) 索引可以避免全表扫描；2) 对于非聚集索引，有些查询甚至可以不访问数据项；3) 聚集索引可以避免数据插入操作集中于表的最后一个数据页；4) 一些情况下，索引还可以避免排序。

虽然索引可以提高查询速度，但是他们也会导致数据库更新数据的性能下降，因为大部分数据更新时需要同时更新索引。

聚集索引：数据按索引顺序存储，叶子节点存储真实的数据行，不再有另外单独的数据页。在一张表上只能创建一个聚集索引，因为真实数据的物理顺序只能有 1 种，若一张表没有聚集索引，则他被称为堆集，这样表的数据行无特定的顺序，所有新行将被添加到表的末尾。

非聚集索引与聚集索引的区别：

1) 叶子节点并非数据节点

2) 叶子节点为每一个真正的数据行存储一个“键-指针”对

3) 叶子节点中还存储了一个指针偏移量，根据页指针及指针偏移可以定位到具体的数据行。

4) 在除叶节点外的其他索引节点, 存储的是类似内容, 只不过是指向下一级索引页。

类加载

类从加载到虚拟机内存中开始, 到卸载出内存为止, 它的整个生命周期包括:

加载-验证-准备-解析-初始化-使用-卸载, 其中验证-准备-解析称为链接。

在遇到下列情况时, 若没有初始化, 则需要先触发其初始化(加载-验证-准备自然需要在此之前):

1) 1.使用 `new` 关键字实例化对象 2.读取或设置一个类的静态字段 3.调用一个类的静态方法。

2) 使用 `java.lang.reflect` 包的方法对类进行反射调用时, 若类没有进行初始化, 则需要触发其初始化

3) 当初始化一个类时, 若发现其父类还没有进行初始化, 则要先触发其父类的初始化。

4) 当虚拟机启动时, 用户需要制定一个要执行的主类(有 `main` 方法的那个类), 虚拟机会先初始化这个类。

在加载阶段, 虚拟机需要完成下面 3 件事:

1) 通过一个类的全限定名获取定义此类的二进制字节流;

2) 将这个字节流所表示的静态存储结构转化为方法区运行时数据结构

3) 在内存中生成一个代表这个类的 `class` 对象, 作为方法区的各种数据的访问入口。

验证的目的是为了确保 `class` 文件的字节流中包含的信息符合当前虚拟机的要求, 且不会危害虚拟机自身的安全。验证阶段大致会完成下面 4 个阶段的检验动作: 1) 文件格式验证 2) 元数据验证 3) 字节码验证 4) 符号引用验证{字节码验证将对类的方法进行校验分析, 保证被校验的方法在运行时不会做出危害虚拟机的事, 一个类方法体的字节码没有通过字节码验证, 那一定有问题, 但若一个方法通过了验证, 也不能说明它一定安全}。

准备阶段是正式为类变量分配内存并设置变量的初始化值得阶段, 这些变量所使用的内存都将在方法区中进行分配。(不是实例变量, 且是初始值, 若 `public static int a=123`;准备阶段后 `a` 的值为 0, 而不是 123, 要在初始化之后才变为 123, 但若被 `final` 修饰, `public static final int a=123`;在准备阶段后就变为了 123)

解析阶段是虚拟机将常量池中的符号引用变为直接引用的过程。

静态代码块只能访问在静态代码块之前的变量, 在它之后的变量, 在前面的静态代码块中可以复制, 但是不可以使用。

通过一个类的全限定名来获取定义此类的二进制字节流, 实现这个动作的代码就是“类加载器”。

比较两个类是否相同, 只有这两个类是由同一个类加载器加载的前提下才有意义, 否则即使这两个类来源于同一个 `class` 文件, 被同一个虚拟机加载, 只要加载他们的加载器不同, 他们就是不同的类。

从 `Java` 虚拟机的角度来说, 只存在两种不同的类加载器: 一种是启动类加载器, 这个类加载器使用 `c++` 实现, 是虚拟机自身的一部分。另一种就是所有其他的类加载器, 这些类加载器都由 `Java` 实现, 且全部继承自 `java.lang.ClassLoader`。

从 JAVA 开发人员角度，类加载器分为：

1) 启动类加载器，这个加载器负责把<JAVA_HOME>\lib 目录中或者 -Xbootclasspath 下的类库加载到虚拟机内存中，启动类加载器无法被 Java 程序直接引用。

2) 扩展类加载器：负责加载<JAVA_HOME>\lib\ext 下或者 java.ext.dirs 系统变量指定路径下 all 类库，开发者可以直接使用扩展类加载器。

3) 应用程序类加载器，负责加载用户路径 classpath 上指定的类库，开发者可以直接使用这个类加载器，若应用程序中没有定义过自己的类加载器，一般情况下，这个就是程序中默认类加载器。

双亲委派模型：若一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把所这个请求委派给父类加载器去完成，每一层的加载器都是如此，因此 all 加载请求最终都应该传送到顶级的启动类加载器。只有当父类加载器反馈自己无法加载时（他的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

双亲委派模型好处：eg, object 类。它存放在 rt.jar 中，无论哪个类加载器要加载这个类，最终都是委派给处于模型顶端的启动类加载器加载，因此 object 类在程序的各种加载环境中都是同一个类。

线程安全与锁优化 东西太多了，直接见《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》最后一章。

同步：多个线程并发访问共享数据时，保证共享数据在同一个时刻，只被一个（or 一些，使用信号量）线程使用，而互斥是实现同步的一种手段，临界区，互斥量，信号量都是主要的互斥实现方式。互斥是因，同步是果；互斥是方法，同步是目的。

公平锁：（先申请先得到）多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获取，而非公平锁则不保证这一点。Synchronized 不是公平锁，reentrantlock 默认下也是非公平的，但是在构造函数中，可以设置为公平的。

互斥同步对性能最大的影响是阻塞的实现，挂起线程和恢复线程的需要转入内核态中完成。若物理机器上有一个以上的处理器，能让 2 个 or 2 个以上的线程同时并行执行，我们就可以让后面请求锁的那个线程“稍等一下”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就释放锁，为了让线程等待，我们只需要让他执行一个忙循环（自旋），这项技术就是所谓的自旋锁。

自适应自旋锁的自旋时间不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态决定。

tcp 如何保证传输的可靠性？tcp 是面向连接，可靠的字节流服务。

面向连接意味着两个使用 tcp 的应用（通常是一个客户端和一个服务器）在彼此交换数据之前必须先建立一个 tcp 连接。在一个 tcp 连接中，仅有两方进行彼此通信，广播和多播不能用于 tcp。

Tcp 通过下列方式提供可靠性：

1) 将应用数据分割为 tcp 认为最合适发送的数据块；

2) 超时重传：当 tcp 发出一个段后，他启动一个定时器，等待目的端确认收到这个报文段。若不能及时收到一个确认，将重发这个报文段。

3) 当 tcp 收到发自 tcp 链接另一端的数据时，它将发送一个确认（对于收到的请求，给

出确认响应)。这个确认不是立即发送,通常将推迟几分之一秒(之所以推迟,可能是要对包做完校验);

4) 若 tcp 收到包,校验出包有错,丢弃报文段,不给出响应, tcp 发送端会超时重传;

5) 对于失序数据进行重新排序,然后交给应用层(tcp 报文段作为 ip 数据报进行传输,而 ip 数据报的到达会失序,因此 tcp 报文段的到达也可能失序。若必要, tcp 将对收到的数据进行重新排列,以正确的顺序交给应用层)。

6) 对于重复数据,直接丢弃。

7) tcp 可以进行流量控制,防止较快主机致使较慢主机的缓冲区溢出。

字节流服务:两个应用程序通过 tcp 连接, tcp 不在字节中插入记录标识符,我们将这种为字节流服务。

Tcp 对字节流的内容不做任何解释, tcp 不知道传输的字节流数据是二进制数据还是 ascii 字符或其他类型数据,对字节流的解释由 tcp 连接双方的应用层。

https 其实是由两部分组成: http+ssl/tls,也就是在 http 上又加了一层处理加密信息的模块,服务端和客户端的信息传输都会通过 tls 加密,传输的数据都是加密后的数据。加解密过程:

1) 客户端发起 https 请求(就是用户在浏览器里输入一个 https 网址,然后连接到 server 的 443 端口)

2) 服务端的配置(采用 https 协议的服务器必须要有一对数字证书,可以自己制作,也可以向组织申请,这套证书就是一对公钥和私钥)。

3) 传输证书(这个证书就是公钥,只是包含了很多信息)

4) 客户端解析证书(由客户端 tls 完成,首先验证公钥是否有效,若发现异常,则弹出一个警示框,提示证书存在问题,若无问题,则生成一个随机值,然后用证书对随机值进行加密)

5) 传输加密信息(这里传输的是加密后的随机值,目的是让服务端得到这个随机值,以后客户端和服务端的通信就可以通过这个随机值来进行加密了)

6) 服务端解密信息(服务端用私钥解密后得到了客户端传来的随机值, then 把内容通过该值进行对称加密。所谓对称加密就是,将信息和私钥通过某种算法混在一起,这样除非知道私钥,不然无法获取内容,而正好客户端和服务端都知道这个私钥,所以只要加密算法够彪悍,私钥够复杂,数据就够安全)

7) 传输加密的信息

8) 客户端解密信息,用随机数来解。

问:在客户端抓包抓到的是加密的还是没加密的?不知道是哪个面试官问的,我就乱说是加密的,然后面试官说错了,是没有加密的

java 集合类:太多了,自己网上查。

ArrayList VS Vector (都实现了 list 接口,都是数组实现)

不安全 安全

扩容 50% 扩容 100%

ArrayList VS LinkedList

数组 链表

适合检索和在末尾插入 or 删除 适合在中间插入 or 删除
LinkedList 还实现了 queue 接口，他还提供 peek(),pool(),offer()等方法。

hashmap 和 hashtable 底层原理（必问，每一个点都要清楚）

给几个网址：<http://zhangshixi.iteye.com/blog/672697>
<http://www.admin10000.com/document/3322.html>
<http://www.cnblogs.com/skywang12345/p/3310887.html>
<http://blog.csdn.net/chdj/article/details/38581035>

使用线程池。我们只需要运行 Executors 类给我们提供的静态方法，就可以创建相应的线程池。

```
Public static ExecutorService newSingleThreadExecutor()
```

```
Public static ExecutorService newFixedThreadPool()
```

```
Public static ExecutorService newCachedThreadPool()
```

newSingleThreadExecutor 返回包含一个单线程的 Executor，将多个任务交给这个 Executor 时，这个线程处理完一个任务之后接着处理下一个任务，若该线程出现异常，将会有一个新线程来代替。

newFixedThreadPool 返回一个包含指定数目线程的线程池，若任务数多于线程数目，则没有执行的任务必须等待，直到有任务完成为止。

newCachedThreadPool 根据用户的任务数创建相应的线程数来处理，该线程池不会对线程数加以限制，完全依赖于 JVM 能创建线程的数量，可能引起内存不足。

我们只需要将待执行的方法放入 run 方法中，将 Runnable 接口的实现类交给线程池的 execute 方法，作为他的一个参数，比如：

```
Executor e=Executors.newSingleThreadExecutor();
e.execute(new Runnable(){ //匿名内部类
    public void run(){
        //需要执行的任务
    }
});
```

排序，感觉这个写的很好（<http://blog.csdn.net/hguisu/article/details/7776068/>），每一个点都要懂，直接插入，选择，冒泡，归并，快排要会写，面试问的最多的是快排（时间复杂度，最好和平均都是 $O(n\lg n)$ ，最差是 $O(n^2)$ ，当数据几乎有序时是最差的，这是退为冒泡，空间复杂度 $O(n\lg n)$ ）。

当运行一个程序时，JVM 启动，运行 bootstrap classloader，该 classloader 加载核心 API（Ext classloader 和 app classloader 也在此时被加载），then 调用 ext classloader 加载扩展 API，最后 app classloader 加载 classpath 目录下定义的 class，这就是一个程序最基本的加载流程。

通过 classloader 加载类实际上就是加载的时候并不对该类进行解析，因此也不会初始化，而 class 类的 forName 方法则相反，使用 forName 方法加载的时候会将 class 进行解析与初始化。

Error 和 **exception** 的区别？**Error** 类一般指与虚拟机相关的问题，比如系统崩溃，虚拟机错误，内存空间不足，对于这种错误导致的应用程序中断，仅靠程序本身无法恢复和预防，遇到这样的错误，建议让程序终止。**Exception** 表示程序可以处理的异常，遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

1) **final** 定义的常量，一旦初始化，不能被修改，对基本类型来说是其值不可变，而对引用来说是其引用不可变。其初始化只能在两个地方 1.其定义处 2.构造函数中，二者只能选其一，不能在定义时给了值，又在构造函数中赋值。2) 不管有无异常发生，**finally** 总会执行，若 **catch** 中有 **return** 语句，也执行，在 **return** 语句之前执行。3) 一个类不能既被声明为 **abstract**，也被声明为 **final** 4) **finalize** 方法定义在 **object** 中。

在 **Java** 中，内存泄漏就是存在一些被分配的对象，这些对象存在以下一些特点：1) 对象是可达的，即在有向图中，存在通路可以与其相连 2) 对象是无用的，即程序以后不会再使用这些对象。这些对象不会被 **gc** 所回收，然而他们却占用内存。

发生内存泄漏的第一个迹象通常是：在应用程序中出现了 **OutOfMemoryError** (**OOM**)

ArrayList 和 **LinkedList** 的 **remove** 和 **contains** 方法都依赖 **equals** 方法。

HashSet: 底层 **hash** 表，若两个元素的 **hash** 值不同，则一定不是同一个元素，若 **hash** 值相同，则判断 **euqals**，若 **equals** 相同，则是同一个对象，若 **equals** 不同，则不是一个对象 (**hashset** 的 **contains** 和 **remove** 依赖于和顺从的和 **equals**)。

TreeSet 集合的特点是可以对其中的元素进行排序，换句话说，一种情况是放进 **treeset** 中的元素必须具有比较性，集合的底层结构是二叉树，保证元素唯一性的方法是 **compareTo** 方法返回 0。另一种情况是元素没有比较性，但是集合有比较性，定义一个类，实现 **conparator** 接口，实现一个比较器。

IOC 控制反转 容器控制程序对象之间的关系，而不是传统实现中，有程序代码之间控制，又名依赖注入。**All** 类的创建，销毁都由 **Spring** 来控制，也就是说控制对象生命周期的不是引用他的对象，而是 **Spring**。对于某个对象而言，以前是他控制其他对象，现在是 **all** 对象都被 **spring** 控制，这就叫控制反转。

依赖注入的思想是通过反射机制实现的。在实例化一个类时，它通过反射调用类中的 **set** 方法将事先保存在 **hashmap** 中的类属性注入到类中。

Spring 实现 **aop**: **JDK** 动态代理，**cglib** 动态代理。

JDK 动态代理：其代理对象必须是某个接口的实现，他是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。核心类有：**InnovactionHandler**，**Proxy**。

Cgib 动态代理：实现原理类似于 **jdk** 动态代理，只是他在运行期间生成的代理对象是针对目标类扩展的子类。**MethodInterceptor**。

多态：父类引用指向子类对象，对于 **static**，编译运行都看左边。

二叉平衡树：他是一颗空树 or 他左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一颗平衡二叉树。

图的深度优先遍历和广度优先遍历。深：设 **x** 是当前被访问的顶点，在对 **x** 做过访问标

记后，选择一条从 x 出发的未检测过的边 (x, y) ，若发现顶点 y 已经访问过，则重新选择另一条从 x 出发的未检测的边，否则沿边 (x, y) 到达未曾访问过的 y ，对 y 访问并将其标记为已经访问；然后从 y 开始搜索，直到搜索完从 y 出发的 all 路径，即访问完 all 从 y 出发可达的顶点后，才回溯回 x ，并且再选择一条从 x 出发的未检测的边。上述过程直至从 x 出发的 all 边都已经检测过为止。遍历结果不唯一。广：设 x 和 y 是两个相继要被访问的未访问过的节点，他们的邻节点分别是 $x_1, x_2, x_3, \dots, x_n$ 和 y_1, y_2, \dots, y_n 。为确保先访问的节点其邻节点先被访问，在搜索过程中使用 FIFO 队列来保存已经访问过的顶点。当访问 x 和 y 时，这两个顶点相继入队。此后，当 x 和 y 相继出队时，我们分别从 x 和 y 出发搜索其邻节点 $x_1, x_2, x_3, \dots, x_n$ 和 y_1, y_2, \dots, y_n 。对其中未访问者进行访问并将其入队列。这种方法是将每个已经访问的顶点入队，故保证了每个顶点至多只有一次入队。广度优先遍历结果不唯一。

排序的代码

// 冒泡排序 从小到大排

```
public static void maopao(int[] arr) {
    int len = arr.length;
    ;
    for (int i = 0; i < len - 1; i++)
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr, j, j + 1);
        }
}
```

// 选择排序 从小到大排

```
public static void xuanze(int[] arr) {
    int len = arr.length;
    for (int i = 0; i < len - 1; i++)
        for (int j = i + 1; j < len; j++) {
            if (arr[j] < arr[i])
                swap(arr, j, i);
        }
}
```



```

// 直接插入排序 从小到大排
public static void zhijiecharu(int[] arr) {
    int len = arr.length;
    for (int i = 1; i < len; i++)
        for (int j = i; j > 0 && arr[j] < arr[j - 1]; j--) {
            swap(arr, j, j - 1);
        }
}

// 归并排序 从小到大排
public static void guibing(int[] arr) {
    int len = arr.length;
    mergesort(arr, 0, len - 1);
}

private static void mergesort(int[] arr, int start, int end) {
    // TODO Auto-generated method stub
    if (start < end) {
        int mid = (start + end) / 2;
        mergesort(arr, start, mid);
        mergesort(arr, mid + 1, end);
        mergepaihaoxu(arr, start, mid, mid + 1, end);
    }
}

private static void mergepaihaoxu(int[] arr, int start1, int end1, int start2, int end2)
    // TODO Auto-generated method stub
    int i = start1;
    int j = start2;
    int[] t = new int[end2 - start1 + 1];
    int k = 0;
    while (i <= end1 && j <= end2) {
        if (arr[i] < arr[j]) {
            t[k] = arr[i];
            i++;
            k++;
        }
        else {
            t[k] = arr[j];
            j++;
            k++;
        }
    }

    while (i <= end1) {
        t[k++] = arr[i++];
    }
    while (j <= end2) {
        t[k++] = arr[j++];
    }
    for (int m = 0; m < t.length; m++)
        arr[m + start1] = t[m];
}

```

```

// 快速排序 从小到大排
public static void kuaisu(int[] arr) {
    int len = arr.length;
    int low = 0;
    int high = len - 1;
    qsort(arr, low, high);
}

private static void qsort(int[] arr, int low, int high) {
    int pos;
    if(low<high){
        pos=partition(arr,low,high);
        qsort(arr,low,pos-1);
        qsort(arr,pos+1,high);
    }
}

private static int partition(int[] arr, int low, int high) {
    // 选择第一个元素作为基准
    int key=arr[low];
    while(low<high){
        while(arr[high]>=key&&high>low)
            high--;
        swap(arr,low,high);

        while(arr[low]<=key&&high>low)
            low++;
        swap(arr,low,high);
    }
    return high;
}

```

泛型，即参数化类型，泛型擦除：Java 编译器生成的字节码文件不包含有泛型信息，泛型信息将在编译时被擦除，这个过程称为泛型擦除。其主要过程为 1) 将所有泛型参数用其最左边界（最顶级的父类型）类型替换；2) 移除 all 的类型参数。

内存溢出：程序在申请内存时，没有足够的内存空间供其使用。

内存泄漏：分配出去的内存不再使用，但是无法回收。

String s="a"+"b"+"c"+"d"+"e";改语句只创建了一个对象"abcde";

String s1="aaa111"; String s2="aaa111"; 对于值相同的常量，在常量池中只会创建一个，所以在编译好的 class 文件中，我们只能找到一个“aaa111”的表示。另外，对于“ ”内容为 null 的常量字符串，会创建一个长度为 0，内容为空的字符串放入常量池中。常量池在运行期间可以动态扩展。

String s1="abcde"; String s2="abc"+"de"; s1==s2 是 true

常量池中只会维护一个值相同的 String 对象。

当调用 String 类的 intern（）方法时，若常量池中已经包含一个等于此 String 对象的字符串（用 Object 的 equals 方法确定），则返回池中的字符串，否则将此 String 对象添加到池中，并返回此 String 对象在常量池中的引用。比如,String s1=new String("asd"); s1=s1.intern(); String s2="asd"; s1==s2;是 true 的。

几个 String 的例子如下：

String s1=new String("777");

String s2="aaa777";

```
String s4="aaa"+s1;  
S2==s4 ;是 false 的
```

```
String s1="abcd";  
Char[] ch={'a','b','c','d'};  
S.O.P(s1.equals(ch));是 false 的，因为他们是不同的类型。
```

```
Char[] c= {'a','b','c','d'};  
String s="abcd";  
Char[] cc=s.toCharArray();  
Cc==c;是 false 的  
Cc.equals(c);是 false 的
```

StringBuffer 内部实现是 char 数组，默认初始化长度为 16，每当字符串长度>char 数组长度时，JVM 会构造更大的 char 数组，并将原先的数组内容复制到新数组。

String s="abc"+"de"; 在编译期间就能确定自字符串值的情况下，使用+效率最高。

String s="a"+"b"; 只创建了一个对象。

String s1="a"; s1+="b"; 创建了两个对象，第一个是"a",当执行第二句话时，因为 String 类不可变性，所以又创建了一个对象。

```
1.String s1=new String("abc");
```

```
2.String s2=new String("abc");
```

执行 1 时，"abc"本来就是池中的对象，而在运行时执行 new String("abc")时，将池中的对象复制一份放入堆中，并且把堆中这个对象的引用交给 s1 持有，这条语句就创建了 2 个对象。执行 2 后，一共创建了三个对象。

MVC M:model entity 业务逻辑 V: view 视图,用户看到并与之交互的界面 C: controller 根据用户的输入,控制用户界面数据显示和更新 model 对象状态 功能模块和显示模块分离,提高可维护性,可复用性,可扩展性。

数据库事务 ACID。

聚集索引的叶节点是数据节点，而非聚集索引的叶节点仍然是索引节点，只不过有一个指针指向对应数据块。

有索引就一定快？NO

tcp 与 udp 的差别在适用性上，并非在与速度上，泛泛地说谁比谁快都是不恰当的。

tcp: http、ftp、pop3、telnet、smtp udp: dns、rip、tftp

tcp 的三次握手，4 次分手的状态图一定要会画，一定要滚瓜烂熟。

使用端口号来区分应用进程。

依赖注入：在运行期间由容器将依赖关系注入到组件中，就是在运行期，由 spring 根据配置文件将其他对象的引用通过组将提供的 setter 方法进行设定。控制反转：对组件对象控制权的转移，从程序代码本身转移到了外部容器，通过容器来实现对象组件的装配和管理。

Hibernate 可以理解为一个中间件，他负责把 java 程序的 sql 语句接受过来并发送到数据库，而数据库返回来的信息由 hibernate 接收后直接生成一个对象传给 java。Hibernate 有 1) 一个配置文件 cfg.xml，包括基本配置信息，比如数据库的操作，username，password，url，driver 和 format sql 和方言；2) 还有一个映射文件 hbm.xml 对数据表中表的映射。

Hibernate 一级缓存是必要的，位于 session 部分，二级缓存则不是必须的，是由 sessionFactory 控制的进程级缓存，由开发人员自行指定。二级缓存可指定使用何种开源的 cache 工具。Hibernate3 以后的默认版本为 Ehcache。查询时间缓存的过程 1) 查询一级缓存中是否有需要的数据 2) 若没有，则查二级缓存 3) 若二级缓存中也没有，此时再执行查询数据库的操作。速度：一级>二级>数据库。

Hibernate 也会自行维护缓存中的数据，以保证缓存中的数据和数据库中的真实数据的一致性。无论如何，当你调用方法 or 获取一个对象时，该对象都将被加入到 session 的内部缓存中，当 flush()方法最后被调用时，对象的状态会和数据库同步。也就是说。删除、更新、增加数据的时候，同时更新缓存。

Session 接口：复制执行被持久化对象的 CRUD 操作（CRUD 的任务是完成与数据库的交流，包含很多的 sql 语句）session 对象是非线程安全的。

SessionFactory 接口：复制初始化 hibernate，复制创建 session 对象，他并不是轻量级的。通常一个项目，一个 sessionFactory 对象。当需要操作多个数据库时，一个数据库，一个 sessionFactory。

Transaction 接口：与事务有关。

Query，criteria 接口：复制执行各种数据库查询，可使用 sql 和 hql。

Hibernate 的主键 1) assign：由程序生成主键，且要在 save()之前，否则抛出异常，特点是主键的生成值，完全由用户决定，与底层数据库无关。用户需要维护主键值，在调用 session.save()之前指定主键值。2) hilo：使用高低算法生成主键值，该方法需要额外的数据表和字段提供高位值来源，默认，使用表 hibernate-unique-key。默认字段为 next-hilo。特点：能保证同一个数据库主键的唯一性，但不能保证多个数据库中主键的唯一性。Hilo 与底层数据库无关，由 hibernate 维护。3) increment：由 hibernate 从数据库中取出主键的最大值，以该值为基础，每次增量为 1，在内存中生成主键，不依赖于 DB。4) sequence：采用数据库提供的 sequence 机制生成主键，需要数据库支 sequence，比如 oracle，db2。Mysql 不支持。5) identity：由底层数据库生成标识符，identity 是有数据库自己生成的，但这个主键必须设置为自增长。使用 identity 前提是数据库支持自动类型增长字段类型。比如：db2，mysql，sql server。Oracle 不支持。6) native：由 hibernate 根据使用的数据库，自行判断采用 hilo，sequence，identity，其中一种作为主键生成方式。比如 mysql 是 identity，oracle 是 sequence。

Hibernate 查询数据的方式：sql；query、criteria，以对象的方式添加查询条件；sql，直接使用 sql 语句操作数据库。

JDBC 中，链接操作是由 Driver Manager 管理，JDBC 建立和关闭时及其耗资源的，connection 接口实现链接数据库。JDBC 链接数据库分为 4 部：1) 加载驱动 Class.forName("com.mysql.jdbc.Driver");2) 创建连接 Connection

con=DriverManager.getConnection(url,username,password); 3) 创建 statement staticment
s=con.createStatement() 4) 执行 sql s.executeQuery();

prepareStatement 是预先编译的语句, statement 不是 防止 sql 注入 性能提高 ,
callableStatement: 存储过程调用 DatabaseMetaDate 类的很多方法都可以知道数据库中存储
过程的详情。

servlet: 是用于 java 编写的服务器端程序, 其使用 java servlet API, 当客户机发送请求
到服务器时, 服务器可以将请求信息发送给 servlet, 并让 servlet 建立起服务器返回给客户
机的响应。当启动 web 服务器 or 客户机第一次请求服务时, 可以自动装入 servlet, 装入后,
servlet 继续运行直到其他客户机发出请求。

servlet 生命周期? Servlet 生命周期分为 3 个阶段 1) 初始化阶段: 调用 init()方法 2) 响
应客户请求: 调用 service()3) 终止:调用 destory().1) 初始化阶段: 在下列时刻 servlet 容器
装载 servlet 1.servlet 容器启动时, 自动装载某些 servlet2. 在 servlet 容器启动后, 客户首
次向 servlet 发送请求 3.servlet 类文件被更新以后, 重新装载 servlet。Servlet 被装载后, servlet
容器创建一个 servlet 对象并调用 servlet 的 init 方法。在 servlet 生命周期内, init()方法只被
调用过一次。Servlet 工作原理: 客户端发起一个请求, servlet 调用 service()方法时请求进行
响应, service 对请求的方式进行了匹配, 选择调用 dopost 或者 doget 等这些方法, 然后进
入对应方法中调用逻辑层上的方法, 实现对客户的响应。2) 响应客户请求: 对于用户到达
servlet 的请求, servlet 容器会创建特定于该请求的 servletRequest 和 servletresponse 对象,
然后调用 servlet 的 service 方法, service 方法从 servletrequest 对象中获得客户请求的信息,
处理该请求, 并通过 servletresponse 对象向客户返回响应消息。3) 终止: 当 web 应用终止
或者 servlet 容器终止或 servlet 容器重新装载 servlet 新实例时, servlet 容器会调用 servlet
对象的第 destory 方法, 在 destory 方法中可释放 servlet 占用的资源。

hashset 不是线程安全的。

若两个对象的 equals 为 true, 则 hashcode 为 true。

tomcat 就是 servlet 容器, servlet 容器为 javaweb 应用提供运行时环境, 它复制管理 servlet
和 jsp 的生命周期, 以及管理他们的共享数据。

Servlet 生命周期以下方法: 以下方法都由 servlet 容器调用。1) 构造函数: 只有第一次
请求 servlet 时, 创建 servlet 实例, 调用构造器, 这说明 servlet 是单例的, 所以又线程安全
问题, 只被调用一次。2) init: 只被调用一次, 在创建好实例后, 立即被调用, 用来初始化
servlet3) service: 被多次调用, 每次请求都会调用 service, 实际响应请求。4) destory: 只
被调用一次, 在当前 servlet 所在的 web 应用被卸载前调用, 用于释放当前 servlet 所占用的
资源。

MVC1) m: model。Dao。与数据库打交道 2) V: view, jsp 在页面上填写 java 代码,
实现显示 3) C, controller, servlet 受理请求, 获取请求参数, 调用 dao 方法, 转发 or 重
定向页面。

Cookie 和 session http 都是无状态的协议，web 服务器不能识别出哪些请求是同一个浏览器发的，浏览器的每一次请求都是完全独立的。

Cookie 放在浏览器端，浏览器第一次范围服务器时，没有 cookie，then 服务器给浏览器一个 cookie，以后每次浏览器访问服务器都要带上这个 cookie。Jsp 是服务端

<% 1.创建一个 cookie 对象 Cookie cookie =new Cookie("name","jhb");

2.调用 response 的一个方法把 cookie 传输给客户端

Response.addCookie (cookie) ;

%>

默认情况下 cookie 是会话级别，存储在浏览器内存中，用户退出浏览器之后被删除，若希望浏览器将 cookie 存在磁盘上，则使用 maxage，并给一个以秒为单位的时间，表示 cookie 的存活时间。

Cookie 作用范围：可以作用当前目录和当前目录的子目录，但不能作用与当前目录的上一级。

Cookie:在客户端保持 http 状态信息的方案，会话跟踪。

Session: 在服务器端保持 http 状态信息。

当浏览器第一次范围服务器时，没有 cookie，服务器也就得不到 JSession_id，then 服务器创建一个 session 对象，并返回一个 JSession_id 给浏览器，通过 cookie 返回，下次浏览器再访问服务器时就带上 cookie（其中有 JSession_id），服务器就能找到对应的 session 对象。JSession_id 可以通过 cookie 传输，也可以通过 url 传送。

一个 session，对应一个 sessionID。

Session: 浏览器和服务器的会话，若浏览器不关，至始至终都是一个 session，因为用 cookie 传送。

URL 重写: response 的 encodeURL(String URL) 方法和 encodedirectURL(String URL)两个都一样。

当程序需要为某给客户端的请求创建会话 session 时，服务器首先检查这个客户端的请求是否包含一个 session 标识，即 JSession_id，如果包含，则说明以前已经为此客户创建过 JSession_id，服务器就按照这个 JSession_id 检索出来（若检索不到，可能会新建一个，这种情况下可能会出现在服务器已经删除了该用户对应的 session 对象。但用户人为的在请求的 URL 上附加会话 JSession_id 的参数。）如不包含则创建一个 session，并生成与这个 session 有关的 JSession_id，这个 JSession_id 将在本次响应中返回给客户端。

默认 session 用 cookie。

若第一次访问某 web 应用程序的一个 jsp 页面，且该 jsp 页面的 page 指定 session=true，服务器会创建一个 httpsession 对象。

注：关闭浏览器只是使存储在浏览器内存中的 session cookie 失效，不会使服务器端的

session 对象失效。

多线程安全隐患的原因：当多条语句在操作同一个线程共享语句时，一个线程对多条语句只执行了一部分，还没执行完，另一个线程参与执行，导致共享数据错误。解决办法：对多条操作共享数据的语句，只能让一个线程执行完，在执行过程中，其他行程不能执行。

```
Synchronized(对象){  
    需要被同步的代码  
}
```

Sleep 只释放 cpu 执行权，不释放锁。

找多个线程的共享数据及操作共享数据的语句。

同步函数使用的锁：this

静态同步函数使用的锁：函数所在类的 class 对象

SpringMVC

使用 DispatcherServlet 进行分发，到达合适的 controller。每个请求来到 dispatcherServlet，dispatcherServlet 来决定到哪个 controller，通过 handlermapping。先要在 web.xml 中配置 dispatcherServlet 的<Servlet>和<ServletMapping>,在 Servlet 中需要一个 xml 文件，自己创建，里面来配置 SpringMVC 的属性。

代理设计模式

代理类和委托类有相同的接口，一个代理类的对象与一个委托类的对象关联。代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法来提供特定的服务。JDK 动态代理（掌握程度是自己会写，知道每个函数的每个参数的作用）反射 1) proxy 类：类的静态方法用来生成动态代理的实例 2) invocationhandler 接口有一个 invoke 方法，用来集中处理在动态代理类对象的方法调用，通常在该方法中实现对委托类的代理访问，每次生成动态代理对象时都要指定一个对应的调用处理器。CGlib 动态代理：jdk 代理机制只能代理实现了接口的类，而没有实现接口的类不能用 jdk 动态代理。CGlib 是针对类来实现代理，他的原理是对指定的目标类生成一个子类，并且覆盖其中方法实现增强，因为采用的是继承，所以不能对 final 修饰的类进行代理，methodinterceptor {我写的远远不够，网上很多例子，自己多看几遍就回了，一定要滚瓜烂熟，被问了很多次}

第一个被加载的类是所谓的“初始化类”，也就是有 main 方法的类，这个类由 jvm 本身加载。

一个抽象类可以没有抽象方法。

一个接口可以继承任意数量的接口，一个抽象类只能继承一个抽象类。

集合里放的是对象。

Finally 除了 try 块调用了 System.exit(0)，finally 都会执行，有 return，也会先执行 finally

中的内容，再执行 `return`。

向上转型：子类转为父类

向下转型：父类转为子类

`ClassCastException`：当试图将对象强制转换为不是实例的子类时，抛出该异常。

内部类一共有 4 种。1) 成员内部类：外部类的成员。2) 局部内部类。3) 静态内部类：类似于静态成员。4) 匿名内部类：实现一个接口 or 继承一个抽象类。

外部类不能任意访问内部类的成员，要做到这一点，外部类必须创建其内部类的对象（静态内部类除外）。内部类可以访问 all 外部类的成员，因为内部类就像是一个类的成员。

Java 本地方法：该方法的实现是由非 java 写的，可以用 java 调用 c 或 c++ 的代码，这是出于性能的考虑 or 访问底层操作系统。

Java 反射：可以在运行时获取类 or 对象的相关信息。

一个类可以有一个 or 多个静态代码块。静态代码块在类被加载时执行，优于构造函数的执行，并且按照各静态代码块放在类中的顺序执行。

java 用 `Observer` 接口和 `Observable` 类实现观察者模式。1) 创建被观察者类 `observable` 类 2) 创建观察者类 `Observer` 接口 3) 对于被观察者，添加他的观察者 `void add Observer(Observer o)`：该方法把观察者对象添加到观察者对象列表中，当被观察事件发生时，调用 `1.setchange()` 用来设置一个内部标志，注明数据发生了变换；`2.notifyObservers()`：调用观察者对象列表中 all `Observer` 的 `update` 方法，通知他们数据发生变换，只有在 `setchange` 被调用以后，`notifyObservers` 才会调用 `update`。4) 对于观察者类，实现 `Observer` 接口的唯一方法 `update`。例子：员工看老板来了，就不玩手机了。员工是观察者，老板是被观察者。

突然想起来，面试的时候被问道代理模式和装饰模式的区别？我自己理解的是：代理是在内部生成一个代理对象，构造函数参数为空。装饰的构造函数参数有一个对象，就是对这个对象进行装饰。

堆排序空间复杂度 $O(1)$ 建堆的时间复杂度 $O(n)$ 调整堆的时间复杂度 $O(\log n)$

B-tree 索引可以用于使用 `<=` `=` `>=` 或者 `between` 运算符的列比较。如果 `Like` 的参数是一个没有以通配符起始的常量字符串也可以使用这种索引。**Hash 索引**：1) 只能够用于使用 `=` 或 `<=>` 运算符的相比较，不能用于有范围的运算符，因为键经过 `hash` 以后的值没有意义 2) 优化器不能使用 `hash` 索引来加速 `order by` 操作。这种类型的索引不能够用于按照顺序查找下一个条目 3) `mysql` 无法使用 `hash` 索引来估计两个值之间有多少行 4) 查找某行记录必须全键匹配，而且 `B-tree` 索引，任何该键的左前缀都可以用以查找记录 5) 若将一张 `myisam` 或 `innodb` 表换为一个 `hash` 索引的内存表，一些查询可能受影响。

输出保留两位小数 `S.O.printf("%.2f\n",cc);` 其中 `cc` 是需要输出的数。

Java 接口的修饰符只可以是 `abstract` 和 `public`。

Iterator 和 collection、map 无关。

JDBC: 桥接模式

局部变量没有默认值。

-Xmx: 最大堆大小

-Xms: 初始堆大小

-Xmn: 年轻代大小

-XXSurvivorRatio: 年轻代中 eden 和 survivor 区的大小比值。

ThreadLocal 不继承 Thread, 也不实现 Runnable 接口, ThreadLocal 类为每一个线程都维护了自己独有的变量拷贝。每个线程都拥有自己独立的变量, 其作用在于数据独立。ThreadLocal 采用 hash 表的方式来为每个线程提供一个变量的副本。

多个静态变量或静态代码块, 按顺序加载。

Object 的 wait () 方法要使用 try, 不然就抛出 InterruptedException。

Hibernate 持久化对象的生命周期 1) 临时状态 (transient), 也称为自由状态, 他只存在于内存中, 且在数据库中无相应的数据。2) 持久化状态 (Persist), 与 session 关联, 且在数据库中有对应数据。3) 游离状态 (detached), 已经持久化, 但不在 session 缓存中。

	持久化	游离	暂时
是否处于 session 缓存中	有	无	无
数据库中是否有记录	有	有	无

最小生成树不唯一。

在单链表中, 增加一个头结点是方便运算。

哈夫曼树有左右子树之分。

稀疏矩阵, 三分组的方法: 非零元素所在的行, 列以及他的值构成一个三元组 (x, y, v), 然后按照某种规律存放三元组, 这样可以节约空间。还要 3 个成员来记录矩阵的行数、列数和总的元素数。

BFS: 广度优先遍历, 使用队列, 先进先出, 正拓扑排序。

DFS: 深度优先遍历, 使用栈, 先进后出, 逆拓扑排序。

若 try, finally 都有 return, 则忽略 try 的 return 语句。

求字符串的 next 数组 自己百度吧

快速排序在无序时效率最高，有序时效率低。

堆排序、基数排序、归并排序、选择排序的排序次数和初始化状态无关，即最好情况和最坏情况一样

知道中序，且知道前序或后序任何一个就可以确定一颗二叉树。

`Integer.parseInt(s,x)`; `s` 是需要转换的字符串，`x` 为按什么进制 2, 10, 16, 8, 默认是 10。16 进制的字符串前面有 0x。

哈希表查找的时间复杂度与原始数量无关，hash 表在查找元素时是通过计算 hash 值来定位元素的位置的，从而直接访问元素。所以 hash 表的插入、删除、查找都是 $O(1)$ 。

空间复杂度 归并 $O(n)$ 快速 $O(\log n)$ 其余 5 种 $O(1)$ 。

观察者模式 1) 推模型：传递的信息多，更具体 2) 拉模型：目标对象在通知观察者时，只传递少量信息，由观察者主动到目标对象中获取，相当于是观察者从目标对象中拉数据，`update` 方法中传目标对象的引用为拉模型 `Observer` 接口 `Observable` 类。

用代理模式，代理类可以对他的用户隐藏一个对象的具体信息。因此，使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。当我们使用装饰模式时，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

Synchronized 1) 同步代码块 使用 `monitorenter` 和 `moniterexit` 指令实现，`monitorenter` 指令插入到同步代码块的开始位置，`moniterexit` 指令插入到同步代码块的结束位置，jvm 需要保证每一个 `monitorenter` 都有一个 `moniterexit` 与之对应。任何对象都有一个 `monitor` 与之相关联，当且一个 `monitor` 被持有之后，他将处于锁定状态。线程执行到 `monitor` 指令前，将会尝试获取对象所对应的 `monitor` 所有权，即尝试获取对象的锁。2) 同步方法。依靠的是方法修饰符上的 `ACC_SYNCHRONIZED` 实现。**Synchronized** 方法则会被翻译为普通的方法调用和返回指令，比如 `invokevirtual` 指令，在 jvm 字节码层面并没有任何特别的指令来实现 `synchronized` 修饰的方法，而是在 `class` 文件的方法表中将该方法的 `access_flags` 字段中的 `synchronized` 标志位置为 1，表示该方法为 `synchronized` 方法，且使用调用该方法的对象 or 该方法所属的 `class` 在 jvm 内部对象表示作为锁对象。

在 java 设计中，每一个对象自打娘胎里出来就带了一把看不见的锁，即 `monitor` 锁。**Monitor** 是线程私有的数据结构，每一个线程都有一个可用 `monitor record` 列表，同时还有一个全局可用列表。每一个被锁住对象都会和一个 `monitor` 关联。**Monitor** 中有一个 `owner` 字段存放拥有该对象的线程的唯一标识，表示该锁这个线程占有。**Owner**：初始时为 `null`，表示当前没有任何线程拥有该 `monitor record`，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时，又设为 `null`。**Entry Q**：关联一个系统互斥锁，阻塞 all 试图锁住 `monitor entry` 失败的线程。**Next**：用来实现重入锁的计数。

锁主要有 4 中状态：无锁状态、偏向状态、轻量级状态、重量级状态。他们会随着竞争的激烈而逐渐升级，锁可以升级但不可以降级。

自旋锁、自适应自旋锁、锁消除。锁消除的依据是逃逸分析的数据支持。锁粗化：将多个连续加锁解锁操作链接起来扩展成一个范围更大的锁。

轻量级锁：传统的锁是重量级锁，他是用系统的互斥量来实现。轻量级锁的本意是在没

有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。

偏向锁：目的是消除数据在无竞争情况下的同步原语。如果说轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那么偏向锁就是在无竞争的情况下把整个同步消除掉。

Synchronized 用的锁时存在 java 对象头里。对象头包含标记字段和类型指针。类型指针是对象指向它类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。标记字段用于存储对象自身运行时数据，他是实现轻量级锁和偏向锁的关键。

JVM 可以通过对象的元数据信息确定对象的大小，但是无法从数组的元数据来确定数组的大小。

Redis 是 **Nosql** 数据库系统，高并发，高性能 面向 key/value。

Java 的序列化算：1) 当前类的描述 2) 当前类属性的描述 3) 父类描述 4) 父类属性描述 5) 父类属性值描述 6) 子类属性值描述 类描述是从下到上，类属性描述是从上到下。

Unicode 码是 2 个字节

B 树、B-树、B+树、B*树

1.B 树就是 **B-树**，是一种多路索引树（并不是二叉的）1) 任意非叶子节点最多只有 m 个儿子，且 $m > 2$ 。2) 根节点的儿子数最多为 $[2, m]$ 。3) 除了跟节点以外的非叶子节点的儿子数为 $[m/2, m]$ 。4) 每个节点至少存放 $m/2 - 1$ 和最多 $m - 1$ 个关键字。5) 非叶子节点的关键字个数 = 儿子数 - 1。6) 非叶子节点的关键字， $k[1], k[2], k[3], \dots, k[m-1]$ ，且 $k[i] < k[i+1]$ 。7) 非叶子节点的指针 $p[1], p[2], p[3], \dots, p[m]$ ，其中 $p[i]$ 指向关键字 $< k[i]$ 的子树， $p[m]$ 指向关键字 $> p[m-1]$ 的子树，其他 $p[i]$ 指向关键字属于 $(k[i-1], k[i])$ 的子树。8) 所有叶子节点位于同一层。

2.B+数：适合文件系统索引，1) 其基本定义和 **B-树** 相同。2) 非叶子节点的儿子数 = 关键字个数 3) 非叶子节点的子树指针 $p[i]$ 指向关键字属于 $[k[i], k[i+1])$ 的子树 (**B-树** 为开区间)。4) **all** 叶子节点增加一个键指针。5) **all** 关键字都在叶子节点出现，且叶子节点本身依赖关键字的大小顺序排序。

查找某一个关键字时，**B-树** 可能在非叶子节点中命中，**B+树** 只有到达叶子节点才命中。

3.B* 树 是 **B+树** 的变体，**B*树** 中非跟和非叶子节点增加指向兄弟的指针。**B*树** 中非叶子节点关键字个数 $\geq 2m/3$ ，即块的最低使用率为 $2/3$ (**B+树** 为 $1/2$)。

B+树 比 **b-树** 根适合 **OS** 的文件索引和数据库索引。

B-树：有序数组 + 平衡多叉树。

B+树：有序数组链表 + 平衡多叉树

B*树：一个丰满的 **B+树**。

乐观锁 VS 悲观锁 1) **悲观锁：**就是很悲观，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 **block** 直到它拿到锁。传统的关系型数据库就用到了很多这种机制，比如行锁，写锁等，都是在操作之前上锁。2) **乐观锁：**就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据。适用于多读，比如 **write_condition**。两种锁各有优缺点，不能认为一种比一种好。

JVM 中，一个字节以下的整形数据 **byte**，-128 到 127 会在 jvm 启动时加载入内存，除非用 **new Integer()** 显示的创建对象，否则都是同一个对象。**Integer** 的 **value** 方法返回一个对

象，先判断传入的参数是否在-128 到 127 之间，若已经存在引用，则直接返回引用，否则返回 new Integer(n).

```
Integer i1=59;
```

```
Int i2=59;
```

```
Integer i3=Integer.valueOf(59);
```

Integer i4=new Integer(59); i1==i2 是 true，因为只有一份；i1==i3 是 true，因为 59 在-128 到 127 之间，因为内存中已经有了 59 的引用，则直接返回；i3==i4 是 false；i4==i2 是 true，因为 i2 为 int 类型，在和 i4 比较时，i4 会自动拆箱。

Integer i=100；相当于编译器为我们做了 Integer i=Integer.value（100）。

HashMap 的 resize 在多线程的情况下可能产生条件竞争。因为如果两个线程都发现 hashmap 需要进行 resize 了，他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。因为移动到新的位置时，hashmap 并不会将元素放在链表的尾部，而是放在头部在，这是为了避免尾部遍历。（否则，针对 key 的 hashCode 相同的 entry 每次添加还要定位到尾节点）。如果条件竞争发生了，可能出现环形列表。之后，当我们调用 get(key)操作时就可能发生死循环。

ConcurrentHashMap：初始化时除了 initialCapacity，loadfactor 参数，还有一个重要的参数 concurrency level，它决定了 segment 数组的长度，默认为 16（长度需要为 2 的 n 次方，与采用的 hash 算法有关）。每次 get/put 操作都会通过 hash 算法定位到一个 segment，然后再通过 hash 算法定位到某个具体的 entry。

Get 操作时不需要加锁的。因为 get 方法里将要使用的共享变量都定义为 volatile。定义为 volatile 的变量，能够在线程之间保存可见性，能够被多个线程同时读，并且保证不会读到过期的值。但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值，像直接 set 值就可以，而 i++这种就非线程安全。）

Put 必须加锁。

ReentrantLock 1) 可重入，单线程可以重复进入，但要重复退出。synchronized 的也是可重入锁。2) 可中断，可响应中断。3) 可限时：超时不能获得锁，就返回 false，不会永久等待构成死锁。4) 公平锁：线程先来，先得到锁，可在构造函数中设置。

Condition 的 await()与 single()和 object 的 wait()和 notify()类似。

对于锁来说，它是互斥的排它的，意思就是只要我获得了锁，没人再能获得，而对于 semaphore 来说，它允许多个线程同时进入临界区，可认为它是一个共享锁，但是共享的额度是有限的，额度用完了，其他没有拿到额度的线程还是要阻塞在临界区外，当额度为 1 时，相当于 lock。

ReadWriteLock：读-读不互斥，读-写互斥，写-写互斥，并发性提高。

CountDownLatch：倒计时器，一种典型的场景就是火箭发射，在火箭发射前，往往还要进行各项设备仪器的检查，只能等所有检查完成后，引擎才能点火，这种场景就非常适合使用 CountDownLatch，它可以使点火线程使得所有检查线程全部完工后，再执行。

cyclicBarrier: 模拟高并发。

Select * from tablename orderby liename limit m,n (从 m+1 条开始, 取 n 条数据)。

a 表

Id	Name
1	张三
2	李四
3	王五

b 表

Id	Age	parent_id
1	23	1
2	34	2
3	34	4

A 的 id 和 b 的 parent_id 关联

1) 内连接

Select a.*,b.*from a inner join b on a.id=b. parent_id;

结果为: 1 张三 1 23 1

2 李四 2 34 2

2) 左外链接 (左表 all 元素都有)

Select a.*,b.*from a left join b on a.id=b. parent_id;

结果为: 1 张三 1 23 1

2 李四 2 34 2

3 王五 null

3) 右外链接 (右表 all 元素都有)

Select a.*,b.*from a right join b on a.id=b. parent_id;

结果为: 1 张三 1 23 1

2 李四 2 34 2

Null 3 34 2

4) 完全链接 (返回左边和右表中所有行, 若某行在另外一个表中无匹配, 则另一表的选择列表列包含空值)

Select a.*,b.*from a full join b on a.id=b. parent_id;

结果为: 1 张三 1 23 1

2 李四 2 34 2

Null 3 34 2

3 王五 null

Not in 的例子:

Select emp_no from employees where emp_no not in (select emp_no from dept_manager).

Sql 的不等为<>

设计模式(大类为 5 种)1) 创建型模式: 工厂方法, 抽象工厂, 单例, 建造者, 原型 2)

结构型模式：适配器，桥接，装饰，代理，外观，组合，享元 3) 行为型模式：策略，模板方法，责任链，观察者，命令，备忘录，访问者，中介者，解析器，状态，迭代子模式。

其实还有两种，并发模式 and 线程池模式。

工厂方法模式适合，凡是出现了大量的产品需要创建，并且具有公共的接口，可以通过工厂方法模式进行创建，一个工厂里，不同方法创建不同的类。

抽象工厂：工厂方法模式有一个问题就是，类的创建依赖于工厂，也就是说想要扩展程序，必须对工厂类进行修改，用抽象工厂，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不用改以前的代码，一个工厂生产一个具体对象。

建造者模式，工厂模式提供的是创建单个对象的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象。

原型模式，将一个对象作为原型，对其进行复制克隆，产生一个和原对象类似的新对象。

适配器模式：将某个类的接口转换为客户端期望的另一个接口表示。1) 接口的适配器模式：有时我们写的接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，但并不是所有方法都是我们必须的，有时只需要某一些，此时，我们可以借助于一个抽象类，该抽象类实现该接口，实现所有的方法，我们不和原始接口打交道，只和抽象类联系，所以我们写一个类继承抽象类，重写我们需要的方法就行。2) 类的适配：当希望将一个类转换为满足另一个接口的类时，创建一个新类，继承原有的类，实现新的接口。3) 对象的适配：当希望将一个对象转换为满足另一个新接口的对象时，创建一个 wrapper 类，持有原类实例，在 wrapper 类的方法中，调用实例的方法

外观模式 **façade**：为了解决类与类之间的依赖关系，将他们的关系放在 **façade** 类中。

组合模式，在处理类似树形结构的问题时很方便。

ConcurrentHashMap

采用了二次哈希的方式，第一次哈希将 key 映射到对应的 segment 中，而第二次哈希则是映射到 segment 的不同桶中。

主要实现类为 ConcurrentHashMap (整个 hash 表)、segment (桶)、HashEntry (每个哈希链中的一个节点)

HashEntry 中，除了 value，其余 3 个 key、hash、next 都是 final 的。这意味着不能从中间或者尾部添加或删除节点，因为这需要改变 next 的引用值，所有节点修改只能从头部开始，对于 put 操作，可以一律添加到哈希链头部，对于 remove 操作，可能需要从中间删一个节点，这就需要将要删除节点的前面所有节点整个复制一遍，最后一个节点指向要删除节点的下一个节点

Segment 类中，count 用来统计该段数据的个数，每次修改操作做了结构上的改变，比如增加或删除 (修改节点的值不算结构上的改变)，都要写 count 的值，每次读取操作开始都要读 count 的值。

put 操作也是委托给段的 put 方法；remove 操作也是委托给段的 remove 方法；get 操作也是委托给段的 get 方法。都是在持有段锁的情况下进行，lock 和 unlock。

由于所有修改操作，在进行结构修改时，都会在最后一步写 count 变量，通过这种机制保证 get 操作，能够得到几乎最新的结构更新。count 是 volatile 的，对 hash 链进行遍历不需要加锁的原因在于 next 指针是 final 的。读的时候，若 value 不为空，直接返回；若为空，则，加锁重读

Segment 是一种可重入锁

Segments 数组的长度 size 通过 ConcurrencyLevel 计算得出 (ConcurrencyLevel 为 14、15、

16 时，size 都为 16。Concurrencylevel 默认为 16），Segments 数组长度必须为 2^n （一个 Concurrenthashmap 有一个 Segments 数组，每个 Segment 相当于一个 hashtable）。

Concurrenthashmap 的 get 操作时如何做到不加锁的呢？因为 get 方法里将需要使用的共享变量都定义为 volatile（定义为 volatile 的变量，可以在线程之间保证可见性，可以被多线程同时读，但只能被单线程写）。Get 中只需要读，不需要写，所以不用加锁。之所以不会读到过期值，因为 java 内存模型的 happen-before 原则，对 volatile 字段的写入操作先于读操作，即使两个线程同时修改 volatile 变量，get 也能拿到最新的值。

Resize 时，只对 Segment 扩容，不对整个 Concurrenthashmap 扩容，先判断是否需要扩容，再 put，传统 hashmap 是先 put 再判断。

Size()，求大小：最安全的做法是统计 size() 时把 all Segment 的 put、remove、clean 方法全部锁住，效率低。所以 Concurrenthashmap 的做法是：先尝试两次不锁住 Segment 的方法来统计各个 Segment 大小。若统计的过程中，容器的 count 发生了变化，则采用加锁的方式来统计。使用 modcount 变量来判断统计过程中，count 是否变化。在 put、remove、clean 方法操作元素前 modcount 要加 1，那么在统计 size 前后比较 modcount 是否发生变化，从而得知容器大小是否变化。

事务隔离级别

	脏读	不可重复读	幻读
1. 读未提交	✓	✓	✓
2. 读已提交	☒	✓	✓
3. 重复读	☒	☒	✓
4. 序列化	☒	☒	☒

悲观锁大多数情况下，依靠数据库的锁机制。乐观锁大多基于数据版本，记录机制实现。数据版本。为数据增加一个版本标识，比如增加一个 version 字段。读数据时，将版本号一块儿读出，之后更新时，版本号加 1，将提交数据的版本号与数据库表对应记录的当前版本号进行对比。若提交的大于数据库里面的，则可以更新，否则认为是过期数据。将乐观锁策略在存储过程中实现，对外只开放基于此存储过程的数据更新途径，而不是将数据表直接对外公开。

解决超卖的方法 1) 乐观锁 2) 队列，建立一条先进先出的队列，每个请求加入到队列中，然后异步获取队列数据进行处理，把多线程变为单线程，处理完一个就从队列中移出一个，因为高并发，可能一下将内存撑爆，然后系统又陷入异常状态。或者设计一个极大的队列，但是系统处理速度和涌入速度根本没办法比，平均响应时间小。

建堆 $O(n)$ ，调整堆 $O(\log n)$ 。

索引（B-树，B+树），一般来说，应该在这些列上加索引：1) 经常需要搜索的列，加快搜索速度 2) 第作为主键的列，强制该列的唯一性 3) 在经常用在连接的列上，这些列主要是外键，可以加快连接速度 4) 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，指定的范围是连续的 5) 在经常需要排序的列上。

唯一索引：不允许其中任何两行有相同的值。

主键索引：为表定义主键自动创建主键索引，是唯一索引的特定类型：

聚集索引：b-树中一次检索最多需要 $h-1$ 次 io（根节点常驻内存，h 为树的高度），一般用磁盘 io 来评价索引的优劣，b 树中一个节点对应一页，所以一个 node 只要一次 io。

下列事件索引会失效：1) 条件中有 **or**，即使其中有条件带索引也不会使用（要想使用 **or** 又想索引生效，只能将 **or** 条件中每个列加上索引）2) **like** 查询，以%开头 3) 若列类型为字符串，则一定要在条件中将数据用引号引起来，否则不使用索引 4) 若 **mysql** 估计使用全表扫描要比索引快，则不使用索引 5) 对索引进行运算导致索引列失效 6) 使用内部函数导致索引失效，这样应当创建基于函数的索引 7) **b-树**，**is null** 不会用，**is not null** 会用。

同步阻塞，用户空间的应用程序执行一个系统调用，这意味着应用程序会一直阻塞，直到系统调用完成为止（数据传输完成或者发生错误）。

同步非阻塞，设备以非阻塞形式打开，这意味着 **io** 操作不会立刻完成，需要应用程序调用多次来等待完成。

同步和异步 1) 同步：发出一个调用时，在没有得到结果前，该调用就不返回，一旦返回就有结果。2) 异步：调用在发出之后就返回，所以没有返回结果，换句话说，当一个异步调用发生后，调用者不会立即得到结果，而是在调用发生后，被调用者通过状态通知来通知调用者，或者通过回调函数来处理这个调用。

阻塞和非阻塞 1) 阻塞：调用结果返回之前，当前线程会被挂起，调用线程只有在得到结果之后才会返回。2) 非阻塞：不能立刻得到结果之前，该调用不会阻塞当前线程。

BIO：同步并阻塞，一个连接一个线程，适用于链接数量小且固定的架构。

NIO：同步非阻塞：一个请求一个线程，客户端发送的链接请求都会注册到多路复用器上，多路复用器轮训到链接有 **io** 请求时才启动一个线程进行处理，适用于链接比较多，比较短。

AIO：异步非阻塞，一个有效请求一个线程，适用于链接数目多且长。

装饰模式：新增行为。

代理模式：控制访问。

线程池中线程任务就是不断检测任务队列是否有任务并不断执行队列中的任务。

Treemap 的底层就是红黑树。

<%@include file=.... %> 静态 include 伪指令 使用高速缓存
<jsp:include > 动态

Hibernate.cfg.xml 和 **Hibernate.hbm.xml**

Hibernate 一级缓存（**session** 缓存）是事务级别的，每个事务（**session**）都有单独的一级缓存。这一级别的缓存是由 **hibernate** 进行管理。一般情况下无须进行干预。每个事务都有单独的一级缓存，不会出现并发问题，因此无须提供并发访问策略。

一级或二级缓存，若查询的是对象的属性，则不会把对象加入缓存中。

双亲委派模型是通过 **loadclass** 方法实现：先检查类是否被加载过，若没有，则调用父类加载器的 **loadclass** 方法，若父类加载器为空，则使用启动类加载器为父类加载器。若父类加载器加载失败，先抛出 **ClassNotFoundException**，然后调用自己 **findclass** 方法进行加载。

要实现自定义类加载器：只需要继承 `java.lang.classLoader`。

`HashSet` 基于 `hashmap` 实现，`hashset` 的元素都放在 `hashmap` 的 `key` 上面，而 `value` 统一为 `private static final Object PERSENT=new Object();` 允许 `null` 值，不允许重。`Hashset` 的 `add` 调用的是 `hashmap` 的 `put()`。

`springMVC` 运行原理 1) 客户端请求提交到 `DispatcherServlet`。2) 由 `DispatcherServlet` 控制器查查询一个 or 多个 `handlermapping`，找到处理请求的 `controller`。3) `DispatcherServlet` 将请求提交给 `controller`。4) `controller` 调用逻辑处理完后，返回 `methodAndView`。5) `DispatcherServlet` 查询一个 or 多个 `ViewResolver` 视图解析器，找到 `modleAndView` 指定的视图。6) 视图负责将结果显示到客户端。

排序：1) 内部排序：在内存中进行排序。2) 外部排序：因排序的数据很大，一次不能容纳 `all` 的排序记录，在排序工程中需要访问外存。

堆对应一颗完全二叉树。

选择排序：1) 从 `n` 个记录中找出关键字最小的记录与第一个记录交换。2) 从第二个记录开始的 `n-1` 个记录中再选出关键码最小的记录与第二个交换。以此类推。

对冒泡排序常见的改进方式是加入一个标志变量 `exchange`，用于标志某一趟排序过程中是否有数据交换，若进行某一趟排序时并没有数据交换，则说明数据已经按要求排好，可立即结束排序，避免不必要的比较过程。

不受初始元素影响的排序：选择，堆，归并。

应用服务器：1) `Tomcat`：应用十分广泛的 `web` 服务器，支持部分 `j2ee`，免费，支持 `servlet`、`jsp`。2) `JBoss`：开源应用服务器。3) `Weblogic`、`webSphere`：业界第一的 `appserver`。4) `Apache`：全球应用最广泛的 `http` 服务器，但只支持静态网页，比如 `asp`、`PHP` 等动态网页不支持。

`Memcached`：是高性能分布式内存缓存服务器，本质是一个内存 `key-value` 数据库，但不支持数据持久化，服务器关闭后，数据全丢失。只支持 `key-value` 结构。

`Redis`：将大部分数据放在内存中，支持的数据类型有：字符串、`hash` 表、链表、集合、有序集合以及基于这些数据类型的相关操作。`Redis` 内部使用一个 `redisobject` 对象来表示 `all key` 和 `value`。

他们的区别：1) `redis` 中并不是 `all` 数据都一直存储在内存中，这是和 `memcached` 相比一个最大的区别。2) `redis` 不仅仅支持简单的 `key-value` 类型的数据，同时还支持字符串、`hash` 表、链表、集合、有序集合。3) `redis` 支持数据备份，即 `master-slave` 模式的备份。4) `redis` 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内存使用。`Memcached` 服务器关闭后，数据丢失。5) `memcached` 挂掉后，数据不可以恢复，`redis` 数据丢失后可通过 `AOF` 恢复（灾难恢复）。

`Mysql` 使用 `B+树`。

索引：1) 可以避免全表扫描。2) 若一个表没有聚集索引，则这样表中的数据没有特定的顺序，`all` 新行将被添加到表的末尾。3) 聚集索引是一种稀疏索引，数据页上一页的索引

页存储的是页指针，不是行指针。对于非聚集索引，则是密集索引，数据页的上一级索引为为每一个每一个数据行存储一条索引记录。

定义一个对象的过程：加载类，分配对象大小，默认初始化（<init>方法），把对象按程序员的意愿进行初始化。

CAS: Compare and swap

Java.util.concurrent 包完全建立在 cas 上，没有 cas 就不会有此包。Cas 有 3 种操作数：内存值 V，旧的预期值 A 和要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 改为 B，返回 V。

CAS 通过调用 JNI 的代码实现。

Java 虚拟机里的对象由 3 部分组成：1）对象头：标记字段（在 32 位和 64 位 jvm 中分别为 32bits 和 64bits）+类型指针 2）实例数据 3）对齐填充。

若对象是一个 java 数组，则对象头中还必须有一块用于记录数组长度的数据。因为虚拟机可以通过普通 java 对象的元数据信息确定 java 对象的大小，但是从数组的元数据中无法确定数组的大小。

对象大小必须是 8 字节的整数倍。

Xms: 堆初始大小。Xmx: 堆最大大小。

内存溢出：程序在申请内存时，没有足够的空间供其使用。内存泄漏：分配出的内存不再使用，但无法回收。

Xss: 设置栈的大小。

标记-清除：首先标记出 all 需要回收的对象，在标记完成后统一回收 all 被标记的对象。

Cms: 并发收集、低停顿。

Gc 进行时必须停顿 all java 执行线程（stop-the-world），即使在号称几乎不会发生停顿的 cms 收集器中，枚举根节点时必须停顿。（因为不可以发生在分析过程中对象引用关系还在不断变化）。

在大多数情况下，对象在新生代 eden 区中分配，当 eden 区域没有足够的内存空间时，虚拟机发起一次 minor gc。

Xmn: 分给新生代的大小。

虚拟机给每一个对象定义一个对象年龄计数器，若对象在 eden 出生并经过第一次 minor gc 后仍然存活，并且能被 survivor 容纳的话，将被移到 survivor 空间中，并且对象年龄设为 1。对象在 survivor 中每熬过一次 minor gc，年龄就+1，当他年龄达到一定程度（默认为 15），就会晋升到老年代。

虚拟机可以从方法表中的 acc_synchronized 访问标志得知一个方法是否为同步方法。当方法调用时，调用指令将会检查方法的 acc_synchronized 访问标志是否被设置了。若被设置了，执行线程就要求先成功持有 monitor，然后才能执行方法，最后当方法完成（无论是正

常完成还是非正常完成)时释放 **monitor**。在方法执行期间, 执行线程持有了 **monitor**, 其他任何线程都无法再获取到同一个 **monitor**, 若一个同步方法执行期间抛出了异常, 并且在方法内部无法处理此异常, 那么这个方法所持有的 **monitor** 将在异常执行到同步方法之外时自动释放。

加载、验证、准备、初始化卸载这 5 个阶段的顺序是确定的。

到初始化的阶段才真正开始执行类中定义的 **Java** 程序代码 (或者说是字节码)。初始化阶段是执行类的构造器 (**<init>()**) 方法的过程。

<init>() 方法是由编译器自动收集类的 **all** 类变量的赋值动作和静态语句块 (**static { }** 块) 中的语句合并产生的。编译器收集的顺序是由语义在源文件中出现的顺序决定的。

<clinit>() 方法与类的构造函数不同, 他不需要显示的调用父类的构造器, 虚拟机会保证在子类的 **<clinit>()** 方法执行之前, 父类的 **<clinit>()** 方法已经执行完毕。因此在虚拟机中第一个被执行的 **<clinit>()** 方法的类肯定是 **java.lang.Object**。

由于父类的 **<clinit>()** 方法先执行, 也就意味着父类中的静态语句块要优于子类的变量赋值。

接口中, 不能有静态代码块。

虚拟机会保证一个类的 **<clinit>()** 方法在多线程环境中被正确的加锁同步。若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, 其他线程都要阻塞等待, 直到活动线程执行 **<clinit>()** 完毕。

若自己编写一个类 **java.lang.Object** 并放在 **classpath** 下, 可以正常编译, 但永远无法被加载。

若有一个类加载请求, 先检查是否已经被加载, 若没有被加载, 则调用父类加载器的 **loadclass** 方法, 若父类加载器为空, 则默认使用启动类加载器作为父类加载器, 若父类加载失败, 抛出 **ClassNotFoundException**, 再调用自己的 **findclass** 方法进行加载。

语法糖, 在计算机语言中添加的某种语法。这种语法对语言的功能并没有影响, 但是更方便程序员使用。**java** 中最常用的语法糖有: 泛型、变长参数, 自动拆箱/装箱。虚拟机运行时不支持这些语法, 他们在编译阶段还原回简单的基础语法结构, 这个过程为解语法糖。

包装类的 “==” 运算, 在不遇到算术运算的情况下, 不会自动拆箱, 遇到了就自动拆箱。

java 中的运算并非原子操作, 导致 **volatile** 变量的运算在并发下并不一定安全。

只有使用 **invokespecial** 指令调用的私有方法, 实例构造器, 父类方法, 以及 **invokestatic** 指令调用的静态方法, 才是在编译期进行解析的, 除了上述四种方法, 其他 **java** 方法调用都需要在运行期进行方法接收者的多态选择, 并且很有可能存在多于一个版本的方法接收者 (最多再除去被 **final** 修饰的方法, 尽管它用 **invokevirtual** 调用, 但也是非虚方法), **java** 中默认的实例方法是虚方法。

每次使用 **use volatile** 变量之前, 都必须先从主内存刷新最新的值, 用于保证能看见其他线程对该变量所做的修改后的值。在工作内存中, 每次修改 **volatile** 变量后, 都必须立刻同步回主内存中, 用于保证其他线程可以看到自己对该变量所做的修改。**volatile** 修饰的变

量不会被指令重排序优化，保证代码的执行顺序与程序顺序相同。

原子性、可见性、有序性。

Happen-before: 对于一个 **volatile** 变量的写操作，先行发生于后面对这个变量的读操作。

各个线程可以共享进程资源（内存地址、文件 io），也可以独立调度。线程是 CPU 调度的基本单位。

synchronized 关键字经过编译后，会在同步块的前后分别形成 **monitorenter** 和 **monitorexit** 这两个字节码指令，这两个字节码指令都需要一个 **reference** 类型的参数来指明要锁定和解锁的对象。若 java 程序中的 **synchronized** 明确指定了对象参数，那就是这个对象的 **reference**；若没有明确指定，那就根据 **synchronized** 修饰的是实例方法或类方法去取对应的对象实例或 **class** 对象来作为搜索对象。

在执行 **monitorenter** 指令时，首先要尝试获取对象的锁，若这个对象没被锁定，或当前线程已经拥有那个对象的锁，把锁的计数器加一。相应的，在执行 **monitorexit** 指令时，会将计数器-1，当计数器为零时，锁被释放。若获取对象锁失败，则当前线程就要阻塞等待，直到对象锁被另外一个线程释放。

对于线程的阻塞或唤醒，需要用户态和核心态切换，状态切换需要很多处理器时间。

类只可以是 **public** 或包访问权限的。

组合：只需要将对象引用置入新类中。

字符串与任意数据类型相连接都用“+”，最终都变为字符串。**S.O.P(a)**调用的是 **a** 的 **toString** 方法。“**source**”+**aa**；由于只能将一个 **String** 对象与另一个 **String** 对象相加，所以编译器会调用 **aa** 的 **toString** 方法，将 **aa** 转为 **string** 对象。对于空指针 **Person p=null**；**S.O.P(p)**；不会报 **null** 指针异常，而是直接打印 **null**；

Arrays.sort 对于基本数据类型用快速排序，对象数组用改进的归并排序。

通常加载发生于创建的第一个对象之时，但是当访问 **static** 变量或 **static** 方法时也会发生加载。

初次使用之处也是 **static** 初始化发生之处，所有 **static** 对象和 **static** 代码都会在加载时按程序中的顺序（定义类时的书写顺序）而依次初始化。当然，定义为 **static** 的东西只会被初始化一次。

private 不能被继承。

将一个方法调用同一个方法主体关联起来叫做绑定。**java** 中除了 **static** 方法和 **final** 方法，其他所有方法都是动态绑定。

private 方法默认为 **final** 的，**private** 方法不能被覆盖。

初始化顺序：1）成员变量默认初始化 2）调用基类的构造器，一层一层调用 3）按声明顺序调用成员的初始化方法 4）调用子类构造器主体。

Collection.sort(list<T> list) 按自然顺序排序，元素必须有可比较性，实现 **comparable** 接口。

Collection.sort(list<T> list,comparator<q super T>)

Listiterator 是一个更强大的 iterator 子类型，只能用于各种 list 访问。

Tree-set 将元素放在红黑树。

LinkedList 是 queue 的实现。

PriorityQueue: 优先级越高，越在头，越早出来。

0-1 背包问题 自己网上去看，一定要懂。

对于一个给定的问题，若具有以下两条性质，可用动态规划 1) 最优子结构：若一个问题的最优解包含了其子问题的最优解，就说明该问题具有最优子结构。2) 重叠子问题。

一致性 hash: 在移除 or 添加一个 cache 时，他能够尽可能小的改变已经存在 key 映射关系。

Hash 冲突解决办法: 1) 链地址法 2) 开放地址法。

字节是一种计量单位，表示数据量多少，他是计算机信息技术用于计量存储容量的一种单位。字符: 计算机中使用的文字和符号。不用编码里，字符和字节对应关系不同。1) ASCII 中，一个英文字母（不论大小写）占一个字节，一个汉字占 2 个字节。2) UTF-8，一个英文 1 个字节，一个中文，3 个字节。3) unicode，中文、英文都是两个字节。

验证证书是否有效。1) 验证证书是否在有效期内: 证书中会包含证书的有效期的起始时间和结束时间。2) 验证证书是否被吊销。被吊销的证书是无效的，验证吊销有两种: CRL 和 OCSP。CRL: 吊销证书列表，证书被吊销后会被记录在 CRL 中，CA 定期发布 CRL，应用程序可以依靠 CRL 来验证证书是否被吊销。有两个缺点: 1) 可能会很大，下载很麻烦。2) 有滞后性，就算证书被吊销了，应用也只能等到发布最新的 CRL 才能知道。OCSP: 在线证书状态检查协议，应用按照标准发送一个请求，对某张证书进行查询，之后服务器返回证书状态。OCSP 可认为是即时的（实际实现中可能有延迟）。3) 验证证书上是否有上级 CA 签名: 每一级证书都是由上一级 CA 证书签发的，上级 CA 证书还可能有上级，最后会找到根证书，根证书即自签证书，自己签自己。以上三点只要有一个没通过，这张证书就是无效的，不该信任。

Java 的包有: java.io; java.net; java.lang; java.util; java.awt; java.sql; javax.swing; java.math (javax 开头的都是扩展包)。

Mysql 分页查询: 客户端通过传递 start (页码)，limit (每页显示的条数) 两个参数去分页查询数据库中的数据。Limit m, n 从 m+1 条开始，取 n 条。1) 查询第一条到第十条的是: select * from table limit 0,10;对应的就是第一页的数据。2) 查询第 10 条到第 20 条的是: select * from table limit 10,10;对应的就是第二页的数据。3) 查询第 20 条到第 30 条的是: select * from table limit 20,10;对应的就是第三页的数据。总结: select * from table limit (页数-1) * 每页条数, 每页条数;

动态代理类的字节码在程序运行时由 java 反射机制形成。cglib 动态代理原理是生成被代理类的子类，并覆盖其中方法进行增强。

系统架构师是一个既需要控制整体，又需要洞悉局部瓶颈，并依据具体的应用场景给出解决方案的人。确定和评估需求，给出开发规范。在需求阶段，负责理解和管理非功能性需求（可维护性、可复用性），还需要确定客户及市场人员所提出的需求，确定开发团队所提出的设计。在软件设计阶段，负责对软件体系结构关键构件、接口。在编码阶段，详细设计者和编码者的顾问，并且经常举行技术研讨会。随着集成和测试，集成和测试支持将成为软件架构师的工作重点。

二分查找返回的是：-（插入点）-1

Myisam：不支持事务行级锁和外键约束。所以当执行 insert 和 update 时，执行写操作时，要锁定整个表，所以效率低。但是它保存了表的行数，执行 select count(*) from table 时，不需要全表扫描，而是直接读取保存的值。所以若读操作远远多于写操作，并且不需要事务，myisam 是首选。

Innodb：支持事务、行级锁和外键，mysql 运行时，Innodb 会在内存中建立缓冲池，用于缓冲数据和索引。不保存表的行数，执行 select count(*) from table 时要全表扫描。写不锁定全表，高效并发时效率高。

短链接一般只会在 client/server 间传递一次读写操作。

Tcp 保活功能，主要为服务器应用程序提供，服务器应用程序需要知道客户主机是否崩溃，从而可以代表客户使用资源。

如果一个给定的连接，在两个小时内没有任何的动作，则服务器就向客户发送一个探测报文段，客户主机必须处于以下四个状态之一：1）客户主机依然正常运行，并从服务器可达，客户的 tcp 响应正常，而服务器也知道对方是正常的，服务器在两个小时后将保活定时器复位。2）客户主机已经崩溃，并且关闭或正在重新启动。在任何一种情况下，客户的 tcp 都没有响应，服务器不能收到对探测的响应，并在 75s 后超时，服务器总共发送 10 个这样的探测，每个间隔 75s。若服务器没有收到一个响应，他就认为客户主机已经关闭，并终止连接。3）客户主机崩溃并已重新启动，服务器将收到一个对其保活探测的响应，这个效应是一个复位，使得服务器终止这个链接。4）客户机正常运行，但服务器不可达，和 2）类似。

tcp 保护功能是探测长连接存活状况。

Tcp 的 keep-alive 是检查当前 tcp 是否还活着；http 的 keep-alive 是让一个 tcp 连接活多久，他们是不同层次的概念。

满二叉树：除叶子结点外的所有节点均有两个子节点，叶子节点数达到最大，所有叶子节点都在同一层；完全二叉树：满是完全的特例；哈夫曼树：每个节点要么没子节点，要么有两个子节点。

链表倒数第 k 个节点，见《剑指 offer》108 页。

建造者模式，对于一个复杂产品，建造的过程一样，但过程可以有不同的表示。

可重入锁：同一线程，外层函数获得锁所之后，内层递归函数仍有获得该锁的机会但不受影响。`Synchronized` 和 `reentrantlock` 都是可重入锁。

如何快速查找链表的中间节点？只遍历一次。答案：建立两个指针，一个指针一次遍历两个节点，一个指针一次遍历一个节点，当快指针遍历到控节点时，慢指针指向的位置为链表中间位置。

`HashMap` 的实现用 `hash` 表，`tree-map` 采用红黑树。

`Collections` 类中提供了一个方法，返回一个同步版本的 `hashmap`，该方法返回的是一个 `synchronized Map` 的实例，`synchronized Map` 类是定义在 `Collections` 中的一个静态内部类。它实现 `map` 接口，并对其中的每一个方法通过 `synchronized` 关键字进行了同步控制。但 `synchronized Map` 在使用过程中不一定线程安全。

一、`Runnable` 的 `run` 方法没有返回值，并且不可以抛出异常；`callable` 的 `call` 方法有返回值，并且可以抛出异常，且返回值可以被 `future` 接收。

```
Callable c=new Callable<Integer>(){
    Public Integer call() throws Exception{
        Return new Random().nextInt(100);
    }
}
```

`Future` 是一个接口。`FutureTask` 是他的一个实现类，可通过 `futureTask` 的 `get` 方法得到返回值。

```
FutureTask<Integer> future =new FutureTask< Integer >( callable);
New Thread(future).start();
```

二、使用 `ExecutorService` 的 `submit` 方法执行 `callable`;

```
ExecutorService threadpoll=Executors.new SingleThreadExecutor();
FutureTask<Integer> future= threadpoll.submit(new callable(){
    Public Integer call() throws Exception{
        Return new Random().nextInt(100);
    }
})
```

`Future` 对象表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成，并获取计算的结果。计算完成后只能使用 `get` 方法来获取结果，若没有执行完，`get` 方法可能会阻塞，直到线程执行完。取消由 `cancel` 方法执行。`isDone` 确定任务是正常执行还是被取消。一旦计算完成，就不能再取消了。

共享锁 VS 排他锁：1) 共享锁，又称读锁，若事务 `T` 对数据对象 `A` 加了 `S` 锁，则是事务 `T` 可以读 `A` 但不能修改 `A`，其它事务只能再对他加 `S` 锁，而不能加 `X` 锁，直到 `T` 释放 `A` 上的 `S` 锁。这保证了其他事务可以读 `A`，但在事务 `T` 释放 `S` 锁之前，不能对 `A` 做任何操作。2) 排他锁，又称写锁，若事务 `T` 对数据对象加 `X` 锁，事务 `T` 可以读 `A` 也可以修改 `A`，其他事务不能对 `A` 加任何锁，直到 `T` 释放 `A` 上的锁。这保证了，其他事务在 `T` 释放 `A` 上的锁之前不能再读取和修改 `A`。

DNS 即使用 tcp，又使用 udp。

a 发送给 b 一个信息，但是 a 不承认他发送了，防止这个可用数字签名。

Spring1) ioc 容器——BeanFactory 是最原始的 ioc 容器，有以下方法 1.getBean2.判断是否有 Bean，containsBean3.判断是否单例 isSingleton。BeanFactory 只是对 ioc 容器最基本行为作了定义，而不关心 Bean 是怎样定义和加载的。如果我们想要知道一个工厂具体产生对象的过程，则要看这个接口的实现类。在 spring 中实现这个接口有很多类，其中一个 xmlBeanFactory。xmlBeanFactory 的功能是建立在 DefaultListablexmlBeanFactory 这个基本容器的基础上的，并在这个基本容器的基础上实行了其他诸如 xml 读取的附加功能。xmlBeanFactory (Resource resource) 构造函数，resource 是 spring 中对与外部资源的抽象，最常见的是文件的抽象，特别是 xml 文件，而且 resource 里面通常是保存了 spring 使用者的 Bean 定义，eg.applicationContext.xml 在被加载时，就会被抽象为 resource 处理[我自己理解，resource 就是定义 Bean 的 xml 文件]。

ioc 容器建立过程: 1) 创建 ioc 配置文件的抽象资源，这个抽象资源包含了 BeanDefinition 的定义信息。2) 创建一个 BeanFactory，这里使用的是 DefaultListablexmlBeanFactory。3) 创建一个载入 BeanDefinition 的读取器，这里使用 xmlBeanDefinitionReader 来载入 xml 文件形式的 BeanDefinition。4) 然后将上面定义好的 resource 通过一个回调配置给 BeanFactory。5) 从资源里读入配置信息，具体解析过程由 xmlBeanDefinitionReader 完成。6) ioc 容器建立起来。

BeanDefinition 类似于 resource 接口的功能，起到的作用就是对所有的 Bean 进行一层抽象的统一，把形式多样的对象统一封装为一个便于 spring 内部进行协调管理和调度的数据结构。BeanDefinition 屏蔽了不同对象对于 spring 框架的差异。

Resource 里有 inputStream。

解析 xml，获得 document 对象，接下来只要再对 document 结构进行分析便可知道 Bean 在 xml 中是怎么定义的，也就可以将其转化为 BeanDefinition 对象。我们配置的 Bean 的信息经过解析，在 spring 内部已经转换为 BeanDefinition 这种统一的结构，但这些数据还不能供 ioc 容器直接使用，需要在 ioc 容器中对这些 BeanDefinition 数据进行注册，注册完成的 BeanDefinition，都会以 BeanName 为 Key，BeanDefinition 为 value，交由 map 管理。注册完之后，一个 ioc 容器就可以用了。

自己理解的，xml 文件抽象为 resource 对象，Bean 抽象为 BeanDefinition 对象。

2) 依赖注入——依赖注入发生在 getBean 方法中，getBean 又调用 doGetBean 方法。getBean 是依赖注入的起点，之后调用 createBean 方法，创建过程又委托给了 docreateBean 方法。在 docreateBean 中有两个方法: 1) createBeanInstance，生成 Bean 包含的 java 对象 2) populateBean 完成注入。在创建 Bean 的实例中，getInstantiationStrategy 方法挺重要，该方法作用是获得实例化策略对象，也就是指通过哪种方案进行实例化的过程。spring 当中提供两种方案进行实例化: BeanUtils 和 cglib。BeanUtils 实现机制是 java 反射，cglib 是一个第三方类库，采用的是一种字节码加强方式。Spring 中默认实例化策略为 cglib。populateBean 进行依赖注入，获得 BeanDefinition 中设置的 property 信息，简单理解依赖注入的过程就是对这些 property 进行赋值的过程，在配置 Bean 的属性时，属性可能有多种类型，我们在进行注入的时候，不同类型的属性我们不能一概而论地进行处理。集合类型属性和非集合类型属性差别很大，对不同的类型应该有不同的处理过程。所以要先判断 value 类型，再调用具体方法。

3) aop——将那些与业务无关，却为业务模块所公共调用的逻辑或责任封装起来，称其

为 aspect，便于减少系统的重复代码。使用模块技术，aop 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。实现 aop 的两大技术：1）采用动态代理，利用截取消息的方式，对该消息进行装饰，以获取原有对象行为的执行。2）采用静态织入，引入特定的语法创建切面，从而可以使编译器可在编译期间织入有关切面的代码。

spring 提供两种方式生成代理对象，jdkProxy 和 cglib。默认的策略是，若目标类是接口则用 jdk 动态代理技术，否则使用 cglib 生成代理。在 jdk 动态代理中使用 Proxy.newProxyInstance() 生成代理对象（JdkDynamicAopProxy 类的 getProxy 方法），JdkDynamicAopProxy 也实现了 invocationhandler 接口，有 invoke 方法，就是在该方法中实现了切片织入。主流程可以简述为：获取可应用到此方法上的通知链（Interceptor chain），若有，则应用通知，并执行 joinpoint，若没有，则直接反射执行 joinpoint。

Introduction 是指给一个已有类添加方法或字段属性，Introduction 还可以在不改变现有类代码的情况下，让现有 java 类实现新的接口，或为其指定一个父类实现多继承，相对于 advice 可以动态改变程序的功能或流程来说，Introduction 用来改变类的静态结构。

拦截器，是对连接点进行拦截，从而在连接点前后加入自定义的切面模块功能。作用于同一个连接点的多个拦截器组成一个拦截器链，拦截器链上的每一个拦截器，通常会调用下一个拦截器。

连接点，程序执行过程中的行为，比如方法调用或特定异常被抛出。

切入点，指定一个 advice 将被引发的一系列的连接点的集合。aop 框架必须允许开发者指定切入点。

通知（advice）：在特定的连接点，aop 框架执行的动作。Spring 以拦截器作通知模型，维护一个围绕连接点的拦截器链。

拦截器（advisor），自己理解，在 invoke 前后加入的方法就是通知。使用 spring 的 PointCutAdvisor，只拦截特定的方法，一个 advisor 定义订一个 PointCut 和一个 advice，满足 PointCut（指定哪些方面需要拦截），则执行相应的 advice（定义了增强的功能）。PointCutAdvisor 有两个常用实现类：NameMatchMethodPointCutAdvisor 和 regexMethodPointCutAdvisor。前者需要注入 mappedname 和 advice 属性，后者需要注入 pattern 和 advice 属性。mappedname 指明要拦截的方法名，pattern 按照正则表达式的方法指明了要拦截的方法名，advice 定义一个增强，即要加入的操作（需要自己实现 MethodBeforeAdvice、MethodafterAdvice、throwAdvice、Methodinterceptor 接口之一），然后在 ProxyBeanFactory 的拦截器中注入这个 PointCutAdvisor。注：一个 ProxyFactoryBean 只能指定一个代理目标。

在 spring 中配置 aop 很麻烦，首先需要编写 xxxadvice 类（需要自己实现 MethodBeforeAdvice、MethodafterAdvice、throwAdvice、Methodinterceptor 接口之一），然后在 xml 配置 advisor。还要在 advisor 中注入 advice，然后将 advisor 加入 ProxyFactoryBean 中。而在 spring2.x 以后引入了 aspect J 注解，只需要定义一个 aspect 类，在 aspect 中声明 advice 类(可同时声明多个)，然后在 xml 配置这个 aspect 类，最后添加一行<aop: aspect j-auto proxy>就可以搞定。

通知类型	接口	描述
前置通知	MethodBeforeAdvice	在目标方法调用前调用
后置通知	MethodafterAdvice	在目标方法调用后调用
异常通知	throwAdvice	在目标方法抛出异常时调用
环绕通知	Methodinterceptor	拦截对目标方法调用
还有一类是引入通知，用来定义切入点的。		

HashMap 不是线程安全的，所以如果在使用迭代器的过程中有其他线程修改了 map，则会抛出 ConcurrentModificationException，这就是所谓的 fast-fail 策略，是通过 modcount 实现，对 HashMap 内容的修改都将增加这个值，那么在迭代器初始化时会把这个值赋给迭代器的 expectedmodcount。在迭代过程中，判断 modcount 和 expectedmodcount 是否相等，若不相等，则说明其他线程修改了 map。Modcount 是 volatile 的。

Hashtable 继承 Dictionary 类，实现 map、cloneable、serializable 接口。

Hashtable 键为 null，则抛出 NullPointerException。他的 clear 方法就是把数组元素都设为 null。

事务隔离级别就是对事物并发控制的等级。串行化：所有事务一个接一个的串行执行。可重复读：所有被 select 获取的数据都不能被修改，这样就可以避免一个事务前后读取数据不一致的情况。读已提交：被读取的数据可以被其他事务修改。读未提交：允许其他事务看到没提交的数据。脏读：事务没提交，提前读取，就是当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没提交到数据库，这时，另外一个事务也访问这个数据，然后使用了这个数据。

https 若在浏览器端抓包，是可以看到数据的，并没有加密，抓到的是加密之前的。

Executors.newCachedThreadPool：创建一个可缓存线程池，若线程池中线程数量超过处理需要，可灵活回收空闲线程（终止并从缓存中移除那些已有 60 秒钟未被使用的线程，因此长时间保持空闲的线程是不会使用任何资源）。对于需要执行很多短期异步任务的程序来说，这个可以提高程序性能，因为长时间保持空闲的这种类型线程池，不会占用任何资源。调用缓存的线程池对象，将重用以前构造的线程（线程可用状态），若没有线程可用，则创建一个新线程添加到池中，缓存线程池将终止并移除 60 秒未被使用的线程。

当线程池的任务缓存队列已满，并且线程池中的线程数目达到 maximumPoolSize 时，若还有任务到来，就会采取任务拒绝策略。通常有以下四种策略：1) ThreadPoolExecutor.AbortPolicy-丢弃任务并抛出 RejectedExecutionException 异常。2) ThreadPoolExecutor.DiscardPolicy-丢弃任务，但不抛出异常。3) ThreadPoolExecutor.DiscardOldestPolicy-丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）。4) ThreadPoolExecutor.CallersRunsPolicy-重试添加当前任务，他会自动重复调用 execute 方法，直到成功。此策略提供简单的反馈机制，能够减缓新任务的提交速度。

怎么判断一个 mysql 中 select 语句是否使用了索引，可以在 select 语句前加上 explain，比如 explain select * from tablename;返回的一列中，若列名为 key 的那列为 null，则没有使用索引，若不为 null，则返回实际使用的索引名。让 select 强制使用索引的语法：select * from tablename from index(index_name);

HashMap，两次哈希，第一次直接调用 key 的 hashCode 方法，第二次再调用一个函数即 hash(key.hashCode())，此方法加入高位计算，防止低位不变高位变化时造成的冲突。

@RestController 注解相当于 @RequestBody+@controller 合在一起的作用。1) 若只使用 @RestController 注解，则 controller 中的方法无法返回 jsp，页面配置的视图解析器 viewResolver 不起作用，返回的内容就是 return 里的内容。比如：本来应该到 success.jsp 页面的，但其显示为 success。2) 若需要返回指定页面，则要使用 @controller 和 viewResolver

结合。3) 若要返回 Json, 则要在方法上加@RequestBody。

都用过哪些注解: @controller、@service、@RequestMapping、@transactional、@RequestBody、@component、@Autowired、@CookieValue、@Repository、@RestController。其中@Controller、@service 负责注册一个 bean 到 spring 上下文中, bean 的 id 默认为类名称开头字母小写。@Autowired 根据 bean 的类型从 spring 上下文进行查找, 注册类型必须唯一。@PathVariable 一个函数里有一个输入参数, 就是{seckillId}的值就用 pathvariable。

http 请求过程——当我们在浏览器输入 www.baidu.com, 然后回车之后的详解。1) 域名解析(域名 www.baidu.com 变为 ip 地址)。2) 发起 tcp 的三次握手, 建立 tcp 连接。浏览器会以一个随机端口(1024-65535)向服务端的 web 程序 80 端口发起 tcp 的连接。这个请求(原始的 http 请求, 经过原始的 tcp/ip 四层模型层层封装), 到达服务器端后, 进入网卡, 然后进入内核的协议栈(一层一层拨开), 然后到达 web 应用程序, 最终建立了 tcp/ip 链接。3) 建立 tcp 连接后发起 http 请求。4) 服务器响应 http 请求, 客户端得到 html 代码。服务器 web 应用程序收到 http 请求后, 就开始处理请求, 处理之后就返回给浏览器 html 文件。5) 浏览器解析 html 代码, 并请求 html 中的资源。6) 浏览器对页面进行渲染, 并呈现给用户。

乐观锁适用于写较少的情况, 即冲突真的很少发生, 这样可以省去了锁的开销, 加大了系统的吞吐量。

正向代理: 我是一个用户, 我访问不了某网站, 但是我能访问一个代理服务器, 这个代理服务器呢, 他能访问那个我不能访问的网站, 于是我先连上代理服务器, 告诉他我需要哪个无法访问的网站的内容, 代理服务器取回来给我。server 不知道 client。

反向代理: client 不知道 server, 并不是 URL 中请求的那个资源, 而是不知道从什么地方来的。以代理服务器来接收 internet 上的请求, 然后将请求转发给内部网络的服务器, 并将从服务器上得到的结果返回给 internet 上请求的客户, 此时代理服务器对外就表现为一个服务器。1) 保证内网安全、2) 负载均衡, nginx 通过 proxy-pass-http 配置代理站点, upstream 实现负载均衡。

正向代理作用: 1) 访问原来无法访问的资源。2) 可以做缓存, 加速访问资源。3) 但代理可以记录用户访问记录(上网行为管理), 对外隐藏用户信息。4) 客户端访问权限, 上网进行验证。

域名解析过程: 1) 在浏览器中输入 www.baidu.com, 操作系统会先检查自己本地的 hosts 文件是否有这个网址映射关系, 若有就调用这个 ip 地址映射, 完成解析。2) 若 hosts 没有, 则找本地 dns 缓存。3) 若 hosts 与本地 dns 缓存都没有, 则找 tcp/ip 参数中设置的首选 dns 服务器, 在此我们叫它本地 dns 服务器, 此服务器收到查询时, 若要查询的域包含在本地配置资源中, 则返回。4) 若要查询的域名不是本地 dns 解析, 但该服务器已经缓存了此网址映射关系, 则调用这个 ip 地址映射。5) 若本地资源和缓存里都没有, 则根据本地 dns 服务器的设置(是否设置转发器)进行查询--1) 未用转发模式, 本地 dns 把请求发给根 dns 服务器, 根 dns 收到请求后会判断这个域名是谁来授权管理, 则会返回一个负责的顶级域名服务器的 ip, 本地 dns 服务器收到 ip 后, 将到联系负责 .com 的这个服务器, 若这台负责 .com 的服务器无法解析, 则找下一级 dns 服务器的 ip 给主机, 依次下去。2) 若用转发方式, dns 服务器就把请求转发给上一级 dns 服务器, 由上一级服务器进行解析, 上一级服务器若不能解析,

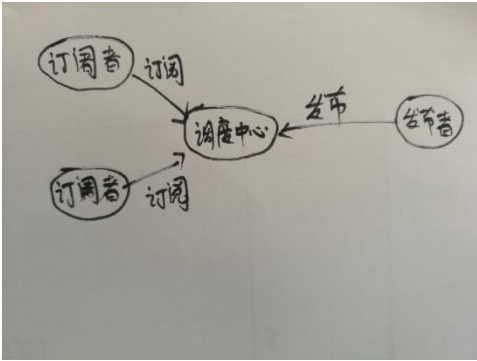
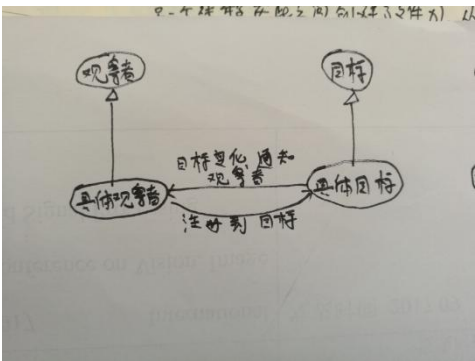
再上一级，以此循环。不管转发还是不转发，都把结果给本地 dns 服务器，再由本地 dns 服务器给客户机。

主机向本地域名服务器查询一般是递归查询，递归查询，就是若主机所询问的本地域名服务器不知道查询域名的 ip 地址，则本地域名服务器就以客户的身份向其他根域名服务器发出查询请求，而不是让主机自己进行下一步查询，所以递归查询返回的结果，要么是 ip 地址，要么报错。

本地域名服务器向根 dns 服务器的查询是迭代查询，当根域名服务器收到本地 dns 服务器的请求时，要么返回一个 ip，要么告诉本地 dns 服务器，你下一步需要向哪一个 dns 服务器查询，然后让本地服务器自己查询。

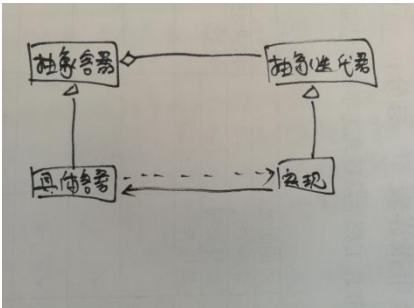
快排改进：1）和插入排序组合，由于快速排序在处理小规模数据时表现不好，因此在数据规模小到一定程度时，改用插入排序，具体小到何种程度，一般文章说 5-10.（SCISTL 硅谷标准模块采用 10）2）中轴元素（左边比他小，右边比他大），可以取最左边、最右边、中间这三个位置的元素中的中间值。3）分成三堆，一方面避免相同元素，另一方面降低了子问题规模（一堆小于一堆等于一堆大于）。

观察者 VS 发布-订阅



虽然两种模式都存在订阅者和发布者（具体观察者可认为是订阅者，具体目标可认为是发布者），但是观察者模式是由具体目标调度，发布-订阅是由调度中心调度，所以观察者模式的订阅者和发布者之间存在依赖，而发布-订阅就不会。

迭代器模式：1）定义：提供一种方法访问一个容器对象中各个元素，而又不暴露该对象的内部细节。2）适用场景：迭代器和集合共生死。3）优点：简化遍历。4）类图



如果当多个线程访问同一个可变的变量时，没有使用合适的同步，那么程序就会出问题。有三种方法可以修复这个问题：1）不在线程之间共享该变量。2）变量改为不可变。3）使用同步。

竞态条件：由于不恰当的执行顺序而不正确的结果。本质——基于一种可能失效的观察结果来作出判断。先检查后执行：首先观察到某个条件为真（例如某个文件 **x** 不存在），然后根据这个观察结果采取相应的动作（创建文件 **x**），但事实上，在你观察到这个结果以及开始创建文件之间，观察结果可能变得无效（另一个线程在此期间创建了文件 **x**），从而导致各种问题。

要保持状态的一致性，就需要在单个原子操作中更新所有相关的状态变量。

可重入锁：若某个线程试图获取一个已经由他自己持有的锁，那么这个请求会成功。实现方式：为每一个锁关联一个获取计数器和一个所有者线程。当计数器为零时，这个锁就被认为是没有被任何线程持有，当线程请求一个未被持有的锁时，将计数器置为 1。若同一个线程再次获取这个锁，计数器递增，而当前线程退出同步代码块时，计数器会相应递减，为 0 时，会释放锁。

锁保证原子性和可见性，**volatile** 只有可见性。

当且仅当满足以下所有条件，才使用 **volatile** 变量：1）写操作不依赖当前值。2）该变量不会与其他变量一起纳入不变性条件。3）访问变量时不需要加锁。

线程封闭：仅在单线程内访问数据，比如 **threadlocal**。

Threadlocal——当使用 **Threadlocal** 维护变量时，**Threadlocal** 为每个使用该变量的线程提供独立的变量副本，所以每个线程可以独立改变自己的变量副本，而不影响其他线程的副本。

从线程的角度看，目标变量就是线程本地变量。

在 **Threadlocal** 类中有一个 **map**，用于存储每一个线程的变量副本，**map** 中 **key** 为 **Threadlocal** 对象，**value** 为该线程的变量副本。

线程隔离的秘密就在 **ThreadlocalMap** 类中。**ThreadlocalMap** 类是 **Threadlocal** 类的静态内部类。它实现了键值对的设置和获取。每个线程对应一个 **ThreadlocalMap**，从而实现了变量访问在不同线程中的隔离。因为每个线程的变量都是自己特有的，完全不会有并发错误。

Threadlocal 的 **set** 方法

```
Public void set(T value){
    Thread T=Thread.currentThread();
    ThreadLocalMap map=getMap(t);
    If(map!=null)
        Map.set(this.value);
    Else
        createMap(t,value);
}
```

当前线程的 **ThreadlocalMap** 是在第一次调用 **set** 方法时创建，并设置上对应的值。每个线程的数据还是在自己线程内部，只是用 **Threadlocal** 引用。**ThreadlocalMap(key,value).key** 为当前 **Threadlocal** 对象，**value** 为变量值。一个线程的 **ThreadlocalMap** 中有很多变量，通过 **Threadlocal** 对象判断选哪个变量。

wait() 抛出 **InterruptedException**

数据库要复习的地方：1) 隔离级别；2) 索引；3) 范式[1NF：属性不可分；2NF：非主键属性完全依赖于主键属性；3NF：非主键属性之间无传递依赖；4NF：主键属性之间无传递依赖]；4) 引擎。

并发包——1) Executor、ExecutorService、AbstractExecutorService、ThreadPoolExecutor。2) copyOnWriteArrayList、copyOnWriteSet。3) BlockingQueue 4) CycleBarrier、CountDownLatch、Semaphore 5) Future、Callable 6) lock

NIO：nio 是 new io，主要用到的是块，所以 nio 效率比 io 高。Java api 中有 2 套 nio：1) 针对标准输入输出 nio；2) 网络编程 nio。io 以流的方式处理数据；nio 以块的方式处理数据。面向流的 io 一次处理一个字节，一个输入流产生一个字节，一个输出流消费一个字节。面向块的 io，每一个操作都在一步中产生或消费一个数据块。

channel 是对原 io 中流的模拟，任何来源和目的数据都必须通过一个 channel 对象。一个 Buffer 是一个容器对象，发给 channel 的所有对象都必须先放到 Buffer 中，同样的，从 channel 中读取的任何数据都要读到 Buffer。

在 nio 中，数据是放入 buffer 对象的。在 io 中，数据是直接写入或读到 stream 对象。应用程序不能直接对 channel 进行读写操作，而必须通过 buffer 来进行。

使用 buffer 读写数据，一般经过以下四步：1) 写入数据到 buffer；2) 调用 flip 方法；3) 从 buffer 中读取数据；4) 调用 clear 方法和 compact 方法。

当向 buffer 写入数据时，buffer 会记录下写了多少数据，一旦要读取数据，通过调用 flip 方法将 buffer 从写模式切换到读模式。在读模式下，可以读取之前写入到 buffer 中的所有数据。一旦读完所有的数据，就需要清空缓冲区，让他可以再次被写入。有两种方式能清空缓冲区：1) clear 清空整个缓冲区 2) compact 只会清除已经读过的数据，未读的数据被移到缓冲区的起始处，新写的数字将放到缓冲区未读数据之后。

Buffer 是父类，它的子类有 ByteBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer、CharBuffer。

buffer 是一个对象，它包含要写入或读取的数据；channel 是一个对象，可以通过他读取和写入数据。

可以把 channel 看作 io 流，但它和 io 流相比还有一些不同：1) channel 是双向的，既可以读又可以写，io 流是单向的；2) channel 可以进行异步读写；3) 对 channel 的读写必须经过 Buffer 对象。

FileChannel：从文件读取数据的。DatagramChannel：读写 udp 的网络协议数据。SocketChannel：读写 tcp 网络协议数据。ServerSocketChannel：可以监听 tcp 连接。

从文件读取数据分三步：1) 从 FileInputStream 中获取 channel；2) 创建 Buffer；3) 从 channel 中读数据到 Buffer。通过 nio 进行文件复制的代码：


```

import java.io.FileInputStream;

//通过nio进行文件的复制
public class NioTest {

    public static void main(String[] args) throws IOException {
        //1.把数据读出来
        //通过inputstream获取Channel
        FileInputStream fileInputStream=new FileInputStream("C:\\Users\\jhb\\Desktop\\Cl
        FileChannel inChannel=fileInputStream.getChannel();
        //创建Buffer
        ByteBuffer byteBuffer=ByteBuffer.allocate(1024);

        //把channel中的数据写入buffer中,若读到末尾,则返回-1
        //channel.read(byteBuffer);

        //2.把数据写入文件
        //获得一个通道
        FileOutputStream fileOutputStream=new FileOutputStream("C:\\Users\\jhb\\Desktop'
        FileChannel outChannel=fileOutputStream.getChannel();
        //创建buffer缓冲区,就用上面的那个缓冲区
        while(true){
            int eof=inChannel.read(byteBuffer);
            //若为-1,则表示已经读到末尾
            if(eof==-1){
                break;
            }
            else {
                //把读模式改为写模式

                byteBuffer.flip();
                outChannel.write(byteBuffer);
                //清空缓冲区
                byteBuffer.clear();
            }

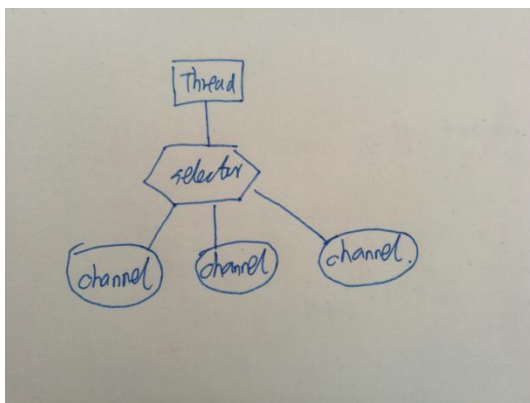
        }
        inChannel.close();
        outChannel.close();
        fileInputStream.close();
        fileOutputStream.close();

    }
}

```

网络编程 nio（异步 io）异步 io 是一种没有阻塞的读写数据的方法。selector 是一个对象，可以注册到很多 channel，监听各个 channel 上发生的事情，并且能够根据事件情况决定 channel 读写，这样通过一个线程管理多个 channel，就可以处理大量网络连接。

有了 selector，可以用一个线程处理所有 channel。线程之间的切换对操作系统来说，代价是很高的，并且每个线程也会占用一定的系统资源，所以对系统来说，线程越少越好（但若 CPU 有多个内核，不使用多任务是在浪费 CPU 能力）。



1) `Selector selector=Selector.open();`

2) 注册 Channel 到 selector 上

`channel.configureBlocking(false)`

`SelectionKey key=channel.register(selector, SelectionKey.OP_READ);`

注：注册到 server 上的 channel 必须设置为异步模式，否则异步 io 无法工作，这就意味着我们不可以把一个 Filechannel 注册到 selector 上。因为 Filechannel 没有异步模式，但 socketchannel 有异步模式。

register 方法的第二个参数，它是一个“interest set”，意思是注册的 selector 对 channel 中的哪些事件感兴趣。事件类型有四种：read、write、connect、accept。通道触发一个事件指该事件已经 Ready，所有某个 channel 成功连接到另一个服务器称为 connect ready。一个 Serversocketchannel 准备好接收新连接称为 connect ready。一个数据可读的通道可以说 read ready。等待写数据的通道 write ready。

Write: `SelectionKey.OP_WRITE`

Read: `SelectionKey.OP_READ`

Accept: `SelectionKey.OP_ACCEPT`

Connect: `SelectionKey.OP_CONNECT`

若对多个事件感兴趣，可写为（用 or）：

`Int interest=SelectionKey.OP_READ|SelectionKey.OP_ACCEPT`

SelectionKey 表示通道在 selector 上的这个注册，通过 SelectionKey 可以得到 selector 和注册的 channel.selector 感兴趣的事情。

一旦向 selector 上注册了一个或多个通道，就可以调用重载的 select 方法返回你所感兴趣的事件（比如连接、接受、读、写）已经准备就绪的通道。比如若你对 Read Ready 感兴趣，select 方法读事件已经就绪的通道，select 方法返回的 int 值，表示有多少通道已经就绪。

`Int select()`：阻塞到至少有一个通道在你注册的事件上就绪；`Int select(long timeout)`：与 `select()` 一样，只是最长只会阻塞 timeout ms；`Int selectnow()`，不阻塞，不管什么通道就绪都立即返回，若自从前一次选择操作后，没有通道变为可选的，则直接返回 0。

Buffer 的三个属性：1) capacity: buffer 有一个固定大小，也就是 capacity，你只能往里面写 capacity 个 byte、long、char 等类型；2) position: 当你写数据时，position 表示当前位置，即下一个可以开始写的位置。初始时 position 为 0，当写入一个数据时，position 会向前移动到下一个可插入数据的 buffer 中，position 最大为 capacity-1；当读数据时，也就是从某个特定位置开始读，应当将 position 从写模式切换到读模式，position 被置为 0。3) limit: 在写模式下，limit 表示你最多能写多少数据，此时 limit=capacity；在读模式中，limit 表示你最多能读到多少数据。所以当切换到读模式时，limit 被设置为写模式下 position。

Socketchannel: 可以通过以下两种方式创建 Socketchannel: 1) 打开一个 Socketchannel,

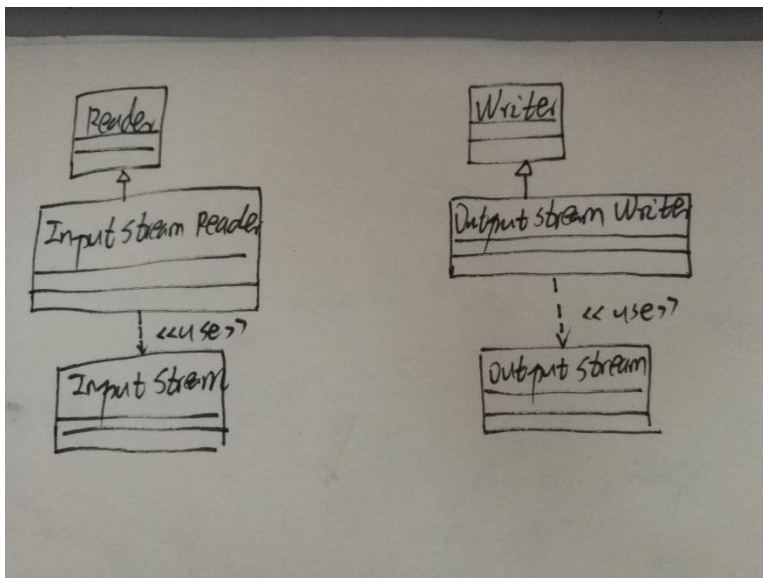
并连到互联网上一台服务器；2) 一个新连接到达 `serverSocketChannel` 时，会创建一个 `SocketChannel`。

打开 `SocketChannel` 方式：1) `SocketChannel socketChannel=SocketChannel.open();`
`socketChannel.connect(new InetSocketAddress("www.baidu.com",80));` 关闭 `SocketChannel`
`SocketChannel.close();int eof=SocketChannel.read(ByteBuffer);` eof: 表示读了多少字节进 buffer 中，若为-1，则表示已经读到流的末尾。

`ServerSocketChannel serverSocketChannel=ServerSocketChannel.open();`
`serverSocketChannel.socket().bind(new InetSocketAddress (900));`
io: 面向流、阻塞、无选择器；nio: 面向缓冲，非阻塞 io，有选择器；

3 个时间复杂度都是一样的：选择、堆、归并。

jdk 中的设计模式：1) 单例，比如 `Runtime` 类；2) 静态工厂 `Integer a=Integer.valueOf(int or String);` 3) 迭代器模式 `Collection.iterator();` 4) 原型设计模式，clone 方法；5) 适配器 `InputStreamReader` 和 `OutputStreamWriter`；



6) 桥接模式，jdbc，抽象部分与实现相分离；7) 装饰模式 `Reader` 和 `bufferedReader`；8) 代理，jdk 动态代理；9) 观察者 `observable` 和 `observer`；10) 责任链，`classloader` 的双亲委派模型；11) 组合，某个类型的方法同时也接收自身类型作为参数 `java.util.list.addall(collection);` 12) 抽象工厂，一个创建新对象的方法，返回的是接口或抽象类 `Connection c=DriverManager.getConnection();` 13) 工厂方法，返回一个具体对象的方法 `Proxy.newProxyInstance;` 14) 解释器模式，该模式通常定义了一个语言的语法，`java.util.pattern`。

CMS 收集器——目的：获取最短的回收停顿时间；基于标记-清除。初始标记，并发标记，重新标记，并发清除。初始标记和重新标记这两步需要“Stop the world”，初始标记只是标记 GC root 能直接关联到的对象，速度快。并发标记就是进行 GC root tracing 的过程，而重新标记阶段是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。

并发标记和并发清除中，垃圾收集线程可以和用户线程同时工作。

并发收集，低停顿。

缺点：1) 对 CPU 资源敏感，cms 默认启动的回收线程数是 (cpu 数量+3)/4; 2) 无法处

理浮动垃圾, 由于 cms 并发清除阶段, 用户线程还在继续执行, 伴随程序进行, 还有新的垃圾产生, 这一部分垃圾发生在标记之后, cms 无法在当次收集时处理他们, 只能留到下一次 gc。3) 它基于标记-清除, 产生大量内存碎片, 大对象分配困难。

Tcp/ip:两军问题,信道不可信.

并发编程中的三个概念:原子性、可见性、有序性。

cpu 为了提高程序运行顺序, 可能会对输入代码进行优化, 它不保证程序中各个语句的执行顺序同代码中的顺序一致, 但是它会保证程序最终执行结果和代码顺序执行结果一致。cpu 在进行重排序时, 会考虑指令之间的依赖性, 若一个指令 instruction 2 必须用到 instruction 1 的结果, 则 cpu 保证 1 在 2 之前执行。

指令重排序不影响单个线程, 但是影响线程并发执行的正确性。

线程对变量的所有操作都必须在工作内存中进行, 而不能直接对主内存进行操作, 且每个线程不能访问其他线程的工作内存。

```
X=10;
```

```
X=y;
```

```
X++;
```

X=x+1; 只有第一个是原子的, 后 3 个是非原子的。

volatile 禁止指令重排序有两层意思: 1) 当程序执行到 volatile 变量的读操作或者写操作时, 在其前面操作的更改肯定全部已经进行, 且结果已经对后面的操作可见; 在其后面的操作肯定还没执行。2) 在进行指令优化时, 不能将在对 volatile 变量访问的语句放在其后面执行, 也不能把 volatile 变量后面的语句放到其前面执行。

```
x=0; //语句 1 x, y 为非 volatile 变量, flag 是 volatile 的;
```

```
y=2; //语句 2
```

```
flag=true; //语句 3
```

```
x=4; //语句 4
```

```
y=-1; //语句 5
```

由于 flag 为 volatile 的, 那么在进行指令重排序时, 不会将语句 3 放在语句 1、2 前, 也不会将语句 3 放到语句 4、5 后, 但是 1、2 的顺序和 4、5 的顺序不可保证. 并且 volatile 关键字能保证执行到语句 3 时, 语句 1 和 2 已经执行完毕, 且语句 1 和语句 2 的执行结果对语句 3、4、5 是可见的。

Volatile 原理: 观察加入 volatile 关键字和没有加入 volatile 时所产生的汇编代码, 发现加入 volatile 时, 会多出一个 lock 前缀指令, lock 前缀指令相当于内存屏障, 内存屏障提供三个功能: 1) 它确保指令重排序时不会把其后面的指令排到内存屏障之前, 也不会把前面的放在内存屏障之后, 即在执行到内存屏障这句指令时, 在他前面的操作已经全部完成。2) 它会强制将对缓存的修改操作立即写入主存。3) 若是写操作, 它会导致其他 cpu 中对应的缓存行无效。

使用场景: 1) 对变量的写操作不依赖当前值。2) 该变量没有包含在具有其他变量的不变式中。比如: 1 标记状态、2) 双重检查。

mysql 默认可重复读。

Spring 用事务管理器来管理事务, 有一套统一的 api, 对于不同持久化方式有不同实现,

比如 jdbc、hibernate。

TransactionDefinition 类用来定义事务的属性。

1) 传播行为, 有 7 种。propagation_required: 支持当前事务, 若当前没有事务, 则创建一个事务; propagation_supports: 支持当前事务, 若当前没有事务, 则以非事务方式执行; propagation_mandatory: 支持当前事务, 若当前没有事务, 则抛异常; propagation_required_new: 新建事务, 若存在当前事务, 把当前事务挂起; propagation_not_supported: 以非事务的方式执行, 若存在当前事务, 则把当前事务挂起; propagation_never: 以非事务的方式运行, 若存在当前事务, 则抛异常; propagation_nexted: 若当前事物存在, 则嵌套事务执行。

2) 隔离级别

3) 它是否为只读事务。

4) 事务超时: 为了使应用程序很好的运行, 事务不能运行太长时间, 因为事务可能涉及对后端数据库的锁定, 所以长时间的事务会不必要的占用数据库资源, 事务超时就是事务的一个定时器, 在特定时间内事务没有完成, 那么就会自动回滚, 而不是一直等待其结束。

5) 回滚规则: 默认情况下, 事务只会遇到 RuntimeException 才回滚。

声明式事务基于 aop, 配置事务: @transaction+事务管理器。

在 jdk1.8 之后, hashmap 中, 若链表长度太长 (默认为 8) 时, 链表转为红黑树。

java 内存模型: 工作内存, 主内存。

Concurrenthashmap 和 hashtable 不允许空键空值。

-xx:newRadio: 设置 young 和 old 的比例; -xx:survivorRadio: 设置 eden 和 survivor 的比例: survivor 大了, 会浪费空间, 空间利用率低。若 survivor 太小, 会使一些大对象在 minor gc 时直接从 eden 区到 old 区, 让 old 区的 gc 频繁。

Xmx: 堆到最大值;

Xms: 堆到初始值;

Xmn: 年轻代的大小;

Xss: 栈的大小。我在测试 OOM 时, Xmx 和 Xms 都设为 20M。。

物理-数据链路-网络-传输-会话-表示-应用。

TCP 和 UDP——tcp: 面向连接, 提供可靠的服务, 无重复、无丢失、无差错, 面向字节流, 只能是点对点, 首部 20 字节, 全双工。UDP: 无连接, 尽最大努力交付, 面向报文, 支持一对一、一对多、多对多, 首部 8 字节。

怎么用 udp 实现 tcp: 由于在传输层 udp 已经是不可靠的, 那就要在应用层自己实现一些保证可靠传输的机制, 简单来说, 要使用 udp 来构建可靠的面向连接的数据传输, 就要实现类似于 tcp 的超时重传 (定时器), 拥塞控制 (滑动窗口), 有序接收 (添加包序号), 应答确认 (ack 和 seq)。目前已经有了实现 udp 可靠运输的机制——udt: 主要目的高速广域网海量数据传输, 他是应用层协议。

进程 VS 程序——程序, 一段代码, 一组指令的有序集合。进程: 程序的一次动态运行, 通过进程控制块唯一的标识这个进程。进程: 动态, 有资源, 有唯一标识, 有并发性; 程序:

静态，无资源，无唯一标识，无并发性。

并行：两个或多个事件，在同一时刻发生。并发：两个或多个事件，在同一时间间隔发生。

进程基本状态：县城：比进程更小的独立运行单位，同一进程中可有多个线程并发执行。
线程：cpu 调度基本单位。进程：cpu 分配资源基本单位。

Java8 新特性:1) 接口的默认方法，java 8 允许我们给接口添加一个非抽象方法，只需使用 default 关键字。2) lambda 表达式，在 java8 之前，若想将行为传入函数，仅有的选择是匿名类，而定义行为最重要的那行代码，却混在中间不够突出。lambda 表达式取代了匿名类，编码更清晰。3) 函数式接口：指仅仅只有一个抽象方法的接口，每一个该类型的 lambda 表达式都会被匹配到这个抽象方法。每一个 lambda 表达式都对应一个类型，通常是接口类型，我们可以把 lambda 表达式当作任意只包含一个抽象方法的接口类型，为了确保接口一定达到这个要求（即有一个抽象方法），你只需要给你的接口加上 @FunctionalInterface 注释（编译器若发现标注了这个注释的接口有多于一个抽象方法，则报错）。4) lambda 作用域，在 lambda 表达式中访问外层作用域和老版本的匿名对象中的方法很相似，你可以直接访问标记了 final 的外层局部变量或实例的字段以及静态变量。lambda 表达式对外层局部变量只可读不可写，对类实例变量可读也可写。5) date API: java8 在 java.time 包中包含一组全新日期 API。6) annotation 注释，java8 支持可重复注解，相同的注解可以在同一地方使用多次。

常见的 RuntimeException：空指针、数组越界、类型转换，除零、并发修改异常，RejectedExecutionException。

Hash 冲突：1) 开放地址法：线性探测、二次探测。2) 拉链法。

Web service：是一种跨编程语言和跨操作系统的远程调用技术，就是说服务器采用 java 编写，客户端程序可以采用其他编程语言编写，且客户端和服务端程序可以在不同操作系统上运行。

所谓远程调用，就是一台计算机 a 上的一个程序可以调用另一台计算机上的一个对象的方法，比如天气预报系统把服务以 web service 服务的形式暴露出来，让第三方的程序可以调用这些服务功能。

从表面上看，web service 就是一个应用程序向外界暴露出一个能通过 web 进行调用的 api；从深层次看，他定义了应用程序如何在 web 上实现互操作性，是一套标准。

web service 技术：1) XML+XSD: web service 采用 http 协议传输数据，采用 xml 格式封装数据(xml 中说明调用远程服务对象的哪个方法，传递的参数是什么,以及服务对象的返回结果是什么).xml 易建立、易分析、与平台无关。Xsd: sml schema: 定义了一套标准的数据类型，当你用某种语言（java、c++等）来构造一个 web service 时，为了符合 web service 标准，所有你使用的数据类型都必须被转换为 xsd 类型（你用的功能可能已经帮你自动完成了这个转换）；2) soap, soap 协议=http 协议+xml 数据格式；3) wsdl: 比如我们去商店买东西，首先要知道商店里有什么东西可买，然后再买，商店的做法是贴张海报。web service 也一样，web service 客户端要调用一个 web service 服务，首先要知道这个服务的地址在哪里，以及服务里有什么方法可调用，所以 web service 服务器首先要通过一个 wsdl 文件说明自己家里有什么服务可以对外调用，服务是什么（服务中有哪些方法、参数）。wsdl 就是一个基于 xml 语言，用于描述 web service 及其函数、参数和返回值。

联合索引：两个或多个列上的索引被称为联合索引。复合索引的结构与电话簿类似，人名由姓和名构成，电话簿首先按姓氏进行排序，对于同一个姓氏按名字来排序，若您知道姓，电话簿将非常有用，若您知道姓和名，则电话簿更为有用，但若你不知道姓，只知道名字，电话簿将没有用处。所以创建复合索引时，要仔细考虑列的顺序。对索引中所有列执行搜索，**or** 仅对前几列执行搜索，复合索引十分有效；但若仅对后面的列进行搜索时，复合索引则没用。

mysql 每次查询，只能用一个索引。

若我们创建了 (A, B, C) 的复合索引，那么其实相当于创建了 (A, B, C)、(A, B)、(A) 三个索引，这被称为最佳左前缀特性。因此，我们在创建复合索引时，应将最常用的放在最左边。

Tcp：1) 流量控制：防止较快主机使较慢主机缓冲区溢出，是点对点；2) 拥塞控制：全局性，防止过多的数据注入网络。

tcp 采用滑动窗口进行流量控制，滑动窗口大小可变，窗口大小的单位是字节。

发送窗口在连接建立时由双方确定，但在通信过程中，接收端可以根据自己的资源情况，随时动态的调整对方的发送窗口上限制。

拥塞控制：慢开始、拥塞避免、快重传、快恢复。

接收端窗口：这是接收端根据其目前的接收缓存大小所许诺的最新窗口值，是来自接收端的流量控制。接收端将此窗口值放在 **tcp** 报文的首部中的窗口字段，传送给发送端，是来自接收端的流量控制。

拥塞窗口：这是发送端根据自己估计的网络拥塞程度而设置的窗口值，是来自发送端的流量控制。

当网络发生拥塞时，路由器就会丢掉分组，因此，只要发送端没有按时收到应当到达的确认报文 **ack**，就可认为网络出现了拥塞。

发送窗口上限= $\min\{\text{接收窗口}, \text{拥塞窗口}\}$ 。

慢开始：由小到大逐渐增大发送端的拥塞窗口数值。

为了防止拥塞，窗口 **cwnd** 的增长引起网络拥塞，还需要慢开始门限 **ssthresh**。

当 $\text{cwnd} < \text{ssthresh}$ ，使用慢开始算法；当 $\text{cwnd} > \text{ssthresh}$ ，使用拥塞避免算法；当 $\text{cwnd} = \text{ssthresh}$ ，既可用慢开始算法也可用拥塞避免算法。

无论是慢开始还是拥塞避免，只要发送端发现网络阻塞，就将慢开始门限设为出现拥塞时的发送窗口值的一半，然后拥塞窗口为一，并执行慢开始算法。这样做的目的是迅速减少主机发送到网络中的分组数，使得发生拥塞的路由器有足够时间把队列中积压的分组处理完毕。

快重传算法规定：发送端只要一连收到三个重复 **ack**，即可断定有分组丢失，就应该立即重传丢失的报文，而不需要等待为该报文设置的重传计时器超时。

与慢开始不同，拥塞窗口不设为 1，而设为慢开始门限+3*mss (mss: 最大报文段)。

linux 常用命令：1) **cd** 打开文件夹；2) **cd..** 后退；3) **mkdir** 创建目录；4) **rm** 删除文件；5) **head tail** 显示文件头尾；6) **hostname** 显示主机名；7) **ipconfig** 查看网络情况；8) **ping** 测试网络连通；9) **netstat** 显示网络状态信息；10) **clear** 清屏。

数组 VS 链表——数组在内存中连续，链表不连续。数组可以随机访问，链表顺序访问。数组查询方面，链表插入和删除方便。

Hash:把任意长度的输入，通过散列算法，变换成固定长度的输出。不同的输入可能导致相同的输出。作用：文件校验，数字签名，hash 表查找 $O(1)$ 。

Int 和 integer: 1) 只要 int 和 integer 比较, integer 就自动拆箱; 2) integer 和 integer
1.integer i=new integer(2); i==i2 是 false;
integer i1=new integer(2);
2.integer i2=new integer(2); i3==i2 是 false;
integer i3=2;
3. integer i4=a;
integer i5=a; 若 a 属于[-128,127],则 i4==i5 是 true; 若 a 不属于[-128,127],则 i4==i5 是 false;

1) 判断一个链表有无环, 一个快指针(走 2 步), 一个慢指针(走 1 步), 都从头开始, 若有环, 则它们相撞(设碰撞点为 x); 若无环, 则快指针遇到空就跳出循环。2) 有环, 则求环的长度: 从碰撞点 x 开始, 又走(一个走一步, 一个两步), 当再次碰撞时, 他们走的次数为环的长度。3) 求连接点, 记住结论: 碰撞点到连接点的距离等于头接点到连接点的距离。两个指针(都一次一步), 一个从头节点走, 一个从碰撞点走, 第一次碰撞的节点就是连接点。

一致性算法: 1) 平衡性: 哈希的结果能够尽可能分布到所有缓存中去。2) 单调性: 如果已经有一些内容通过 hash 分派到了相应的缓冲中, 又有新的缓冲区加入到系统中, 那么 hash 的结果应该能保证原有已分配的内容可以被映射到新的缓冲区中, 或原来的缓冲区中。3) 分散性: 在分布式环境中, 终端有可能看不到所有缓冲区, 而只能看到其中一部分, 当终端希望通过哈希过程将内容映射到缓冲区上时, 由于不同终端所见的缓冲范围有可能不同, 可能导致相同的内容被不同的终端映射到不同的缓冲区上, 这种情况应该避免。4) 负载: 既然不同的终端可能将相同的内容映射到不同缓冲区中, 那么对于一个特定的缓冲区而言, 也可能被不同的用户映射为不同的内容。

在分布式集群中, 对机器的添加和删除或机器故障后自动脱离集群, 这些操作是分布式集群管理最基本的功能, 若采用常用的 $\text{hash}(\text{object})\%N$ 算法, 那么在有机添加或删除以后, 很多原有的数据就无法找到, 所以出现一致性哈希算法——1) 环形 hash 空间: 按照常用的 hash 算法来将对应的 key 哈希到一个具有 2^{32} 个桶的空间, 即 $(0-2^{32}-1)$ 的数字空间中, 现在我们将这些数字头尾相连, 想象成一个闭合的环形。2) 把数据通过一定的 hash 算法映射到环上。3) 将机器通过一定的 hash 算法映射到环上。4) 节点按顺时针转动, 遇到的第一个机器, 就把数据放在该机器上。

Nginx: 可以根据客户端的 ip 进行负载均衡, 在 upstream 里设置 ip_path, 负载均衡五种配置: 1) 轮询(默认), 每个请求按时间顺序逐一分配到不同的后端服务器, 如果后端服务器 down 掉, 能自动剔除; 2) 指定权重, 指定轮询几率。权重越大, 轮询几率越大, 用于后端服务器性能不均的情况。3) ip 绑定 ip_path, 每个请求按访问 ip 的哈希结果分配, 这样每个客户固定访问一个服务器, 可以解决 session 问题。4) fair(第三方)按后端服务器的响应时间来分配请求, 响应时间短的优先分配。5) url_hash 按访问的 url 结果来分配请求, 使每个 url 定位到同一个后端服务器。后端服务器为缓存时比较有效。具体的配置自己网上百度。

Treemap: 树总是平衡的，保证插入、删除、查询的性能 $O(\log n)$ 。

Offer、peek、poll 不抛异常。

start()和 run()——1) **start** 方法启动线程，真正实现多线程运行，通过调用 **Thread** 类的 **start** 方法来启动一个线程，这时此线程是处于就绪状态，并没有运行，若 **cpu** 调度该线程，则该线程就执行 **run** 方法；2) **run** 方法当做普通方法的方式调用，程序要顺序执行，要等 **run** 方法执行完毕，才可以执行下面的代码，程序中只有主线程这一个线程（除了 **gc** 线程）。

抽象类和接口应用场景——1) 标记接口：什么抽象方法都没有。2) 需要实现特定的多项功能，而这些功能之间完全没有联系。3) 定义了一个接口，但又不想强迫每个实现类都必须实现所有抽象方法，所以可以用抽象类来实现一部分方法体，比如 **adapter**。抽象类是简化接口的实现，它不仅提供了公共方法的实现，让我们可以快速开发，又允许你的类完全可以实现所有的方法，不会出现紧耦合的情况。

Jdk1.5 泛型、自动拆箱/装箱，foreach、枚举、线程池；jdk1.7 switch 中用 string、菱形语法。

索引的缺点：1) 创建和维护索引要耗费时间。2) 索引需要占用物理空间，除了数据表占数据空间以外，每一个索引还要占物理空间。3) 当对表中数据进行增加、删除和修改时，索引也要动态维护，这样就降低了数据的维护速度。

常量池——**byte、short、int、long、char、boolean** 包装类实现了的常量池（**float、double** 没有）；

```
Integer i1=123;
Integer i2=123;
i1==i2; 是 true 的;
Boolean b1=true;
Boolean b2=true;
b1==b2; 是 true 的;
```

常量池主要用于存放两大类常量：1) 字面量、符号引用。字面量相当于 **java** 语言层面常量的概念，符号引用包括类和接口的全限定名，字段名称和描述名称，方法名称和描述符。

运行时常量池有动态性，**java** 语言并不要求常量一定只有在编译时产生，也就是并非预置入 **class** 文件中常量池的内容才能放入常量池，运行期间有新的常量也可放入池中，比如 **String** 的 **intern** 方法。优：对象共享，节省内存空间，节省运行时间。

左连接 on 和 where——**on and** 是在生成临时表时使用的条件，不管 **on** 中条件是否为真，都会返回左表中的记录。2) **where** 是临时表生成好后，再对临时表进行过滤的条件，这时已经没有了 **left join** 的含义（必须返回左表记录），条件不为真的就全部过滤掉。**on** 后的条件用来生成临时表，**where** 条件用来对临时表关联。

Inner join: 不管对左表或右表进行筛选，**on and** 和 **on where** 都会对生成的临时表进行过滤。

若一个 **treemap** 集合中放的 **student**，按 **student** 的 **age** 排序，把 10 个 **student** 放入 **treemap**

中，他们已经排好序了，若修改一个 student 的 age，他们顺序没有变。

如下图：

```
import java.util.Comparator;

public class TreeMapTest {

    public static void main(String[] args) {
        TreeMap<Student1,Integer> treemap=new TreeMap<>(new Comparator<Student1>() {

            @Override
            public int compare(Student1 o1, Student1 o2) {
                // TODO Auto-generated method stub
                return o1.age-o2.age;
            }

        });

        Student1 student1=new Student1(1);
        Student1 student2=new Student1(2);
        Student1 student3=new Student1(3);
        Student1 student4=new Student1(4);
        Student1 student5=new Student1(5);
        treemap.put(student4, 1);
        treemap.put(student5, 2);
        treemap.put(student3, 6);
        treemap.put(student1, 2);
        treemap.put(student2, 7);
        System.out.println(treemap);

        System.out.println(treemap);
        student1.age=100;
        System.out.println("-----");
        System.out.println(treemap);

    }

}

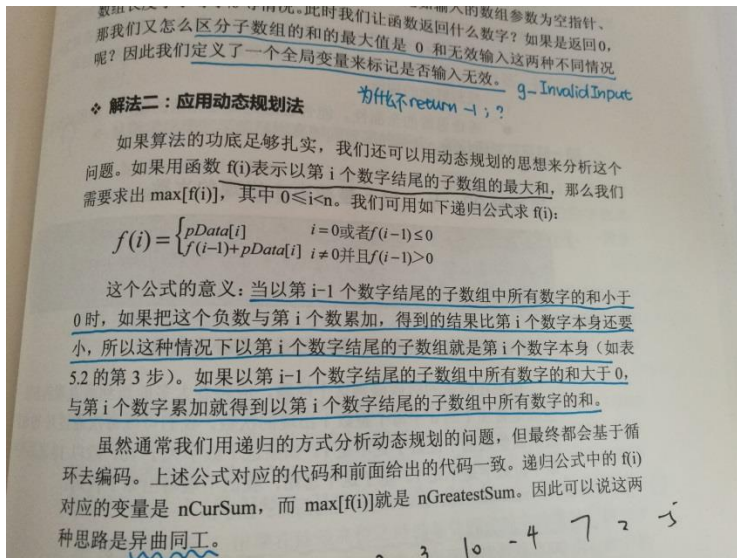
class Student1{
    int age;
    public Student1(int age) {
        this.age=age;
    }
    @Override
    public String toString() {
        return "Student [age=" + age + "]";
    }
}
```

运行结果：

```
{Student [age=1]=2, Student [age=2]=7, Student [age=3]=6, Student [age=4]=1, Student [age=5]=2}
-----
{Student [age=100]=2, Student [age=2]=7, Student [age=3]=6, Student [age=4]=1, Student [age=5]=2}
```

连续子数组的最大和，用 $f(i)$ 表示以 i 结尾的子数组的最大和。《剑指 offer》173 页。

如下图：



我会把手机定义为一个抽象类，它是各种不同手机类的抽象，然而有些手机有指纹解锁的功能，有些手机有防水防尘的功能，对于附加的功能，可以分别定义为接口，当需要这些功能时，再去实现。

堆——完全二叉树——优先级队列。

空间复杂度：快速 $O(n \log n)$ ，归并 $O(n)$ 、堆 $O(1)$ 。

Hibernate——1) 一级缓存 session: hibernate 内置，不能卸除。缓存范围：缓存只能被当前 session 对象访问，缓存的生命周期依赖于当前 session 的生命周期，session 关闭后，缓存也就没有了。2) 二级缓存 sessionFactory: 使用第三方插件。缓存范围：被应用内所有 session 共享，缓存生命周期依赖于应用的生命周期。

单调栈——单调栈的维护是 $O(n)$ 的时间复杂度。所有元素只会进入栈一次，并且出栈后再也不会入栈。性质：1) 单调栈里元素有单调性；2) 元素加入栈前，会在栈顶端把破坏栈单调性的元素都删除；3) 使用单调栈可以找到元素向左遍历第一个比它小的元素，也可以找到元素向左边遍历第一个比它大的元素。

CMS: 由于在垃圾收集阶段用户线程还在运行，也就是还需要预留有足够的内存空间给用户线程使用，因此 cms 收集器不能像其他收集器一样，等到老年代几乎完全被填满了再进行收集，需要预留一部分空间，提供并发收集时的线程使用。Jdk1.5 中，默认老年使用 68% 后就会被激活 cms，jdk1.6 变为 92%，要是 cms 运行期间预留的内存不够，就会出现一次 “concurrent mode failure”，这时 JVM 会启动后备预案，临时启用 serial 收集器来重新进行老年代的垃圾收集，这样停顿的时间就长了。

loc 好处：降低组件之间的耦合性。

聚集索引：返回适用于某范围内的数据。

接口：定义与实现分离。

面向接口编程优点：1）接口经过合理设计之后，有助于程序设计的规范化，可以并行开发，提高工作效率；2）实现了程序的可插拔性，降低耦合度。

Hash 函数里不能有随机函数。

开放地址法：1）线性探测，若当前冲突，则往后一个，若再冲突，则再往后，直到找到一个合适的；2）二次探测；3）随机探测。

Get、post、head、delete、put、options、trace、connect。

基本数据类型 double 在判断相等时，为什么不可以用==？答案：存储精度问题，一般使用阈值，a-b 小于这个阈值，一般就认为相等。

锁有一个专门的名字：对象监视器。当多个线程同时请求某个锁时，则锁会设置几种状态来区分请求的线程；1）connection list：所以所有请求锁的线程将被首先放置到该竞争队列中；2）entry list：那些有资格成为候选人的线程被移到 entry list；3）wait set：那些调用 wait 方法被阻塞的线程放置到 wait set 中。

3.2 没问到过的问题（包含答案）

GetVSPost——get 提交的信息显示在地址栏 不安全 对大数据不行（因为地址栏存储体积有限） 获取/查询资源；Post 提交的信息不显示在地址栏 安全 对大数据可以 更改信息

URL 中可以存在中文吗？

A:可以，先将中文进行编码，tomcat 默认解码为 iso8859-1，这样就会出现乱码，我们可以再用 iso8859-1 进行编码，再用指定码表解码（post 和 get 都可以）。对于 post，可以使用 request 的 setCharacterEncoding 方法设置指定的解码表。

现在需要测试系统的高并发性能，如何模拟高并发？

使用 cyclicBarrier，cyclicBarrier 初始化时规定一个数目，then 计算调用了 cyclicBarrier.await() 进入等待的线程数，当线程数达到这个数目时，所有进入等待状态的线程将被唤醒并继续。Cyclic 就像他的名字一样，可看成是一个屏障，所有线程必须到达后才可以一起通过这个屏障。

jdk1.8bin 目录下的东西

Java.exe javac.exe Javadoc.exe jar.exe jstack.exe (打印所有 java 线程堆栈跟踪信息)

若发现 sql 语句执行很慢，怎么找出这条 sql 语句

查日志

JVM 如何知道一个对象要分配多大的内存

当类被加载如内存后，jvm 就知道。JVM 可以通过普通 java 对象的元数据信息确定 java

对象的大小。

类需要同时满足下面 3 个条件才是“无用的类”：

1. 该类的 all 实例都已经被回收
2. 加载该类的 classloader 已经被回收
3. 该对象的 java.lang.class 对象没有在任何地方被引用, 无法再任何地方通过反射访问该类的方法。

永久代的垃圾回收主要回收废弃常量和无用的类。

对象所需的内存大小在类加载完成就可以确定, 为对象分配内存 1) 指针碰撞 (内存是绝对规则的, 用过的放一边, 没用过的放一边) 2) 空闲列表

在虚拟机中, 对象在内存中的存储的布局可以分为 3 个区域: 对象头、实例数据、对齐填充。对象头由 2 部分组成: 标记字段 (对象自身运行时的数据) 和类型指针 (对象指向它的元数据的指针, 虚拟机通过这个指针来确定这个对象是哪个类的实例)。对象的大小必须是 8 字节的整数倍, 而对象头已经是 8 字节的整数倍 (1 倍 or 2 倍), 所以当实例数据没有对齐时, 就要通过对齐填充来补全。

对象的访问方式

- 1) 句柄访问, 堆中有一个句柄池, reference 中存放的是对象的句柄地址, 句柄中包含了对象实例数据与类型数据各自的具体地址信息。
- 2) 直接指针访问, reference 放的就是对象的地址

假如堆内存最大为 2G, 现在发现电脑内存占用了 3G, 原因可能是 直接内存。

like "%aaa%" 不会使用索引, like "aaa%" 可以使用索引。

被动引用: 1) 通过子类引用父类的静态字段, 不会导致子类的初始化, 即父类静态代码块执行, 子类静态代码块不执行。2) 通过数组定义来引用类, 不会触发此类的初始化, eg, SuperClass [] ss=new SuperClass[10]。3) 常量在编译阶段会存入调用的类的常量池, 本质上没有直接引用到定义常量的类, 因此不会触发定义常量的类的初始化。

synchronized 是重量级的同步机制。

all 的可重入代码都是线程安全的, 但是并非 all 线程安全的代码都是可重入的。

锁消除是指虚拟机的编译器在运行时, 对一些代码上要求同步, 但是被检测到不可能存在共享数据竞争的锁进行消除。

jdk 和 jre 区别? jdk 的 bin 下有 javac, jre 的 bin 下没有 javac。

面向对象和面向过程的区别?

(过程) 优点: 性能比面向对象高, 因为类调用时需要实例化, 开销比较大, 比

较消耗源;比如嵌入式开发、Linux/Unix 等一般采用面向过程开发,性能是最重要的因素。缺点:没有面向对象易维护、易复用、易扩展。

(对象)优点:易维护、易复用、易扩展,由于面向对象有封装、继承、多态性的特性,可以设计出低耦合的系统。缺点:性能比面向过程低。

一个数的集合,里面只有一个数不同,其他都成对出现,怎么找出这个不同的数?

见《剑指 offer》P211。

两个栈实现队列? 见《剑指 offer》P59,插入的元素都放在 stack1 中, pop 时,若 stack2 不为空,则直接弹出 stack2 的栈顶,若 stack2 为空,则把 stack1 的元素弹入 stack2,再 pop 出 stack2 的栈顶。

现在有很多 xml 布局文件里面要使用相同的布局,要怎么实现复用? 用 include (这道题不知道在哪里看到的,从来没被问到过)。

在 JAVA 中,字符只以一种形式存在,那就是 Unicode (不选择任何特定的编码,直接使用他们在字符集中的编号,这是统一的唯一的方法)。在 java 中指:在 JVM 中,在内存中,在你的代码里声明的每一个 char, string 类型的变量中。

Reader 和 Writer 类使用的是 GBK 编码。

```
int i=0;
```

S.o.p(i+'o'); 打印结果为 48; 因为'o'是 char 类型,要转为 int 'o'对应 48

```
int j=0;    j=j++; //此时 j 还是 0
```

```
j=++j; //此时 j 是 1
```

类型由低到高为 (byte, short, char) --int--long--float--double

byte, short, char 相同级别,不能相互转换,但可以使用强制类型转换

六个包装类: Float, Double, Byte, Short, Int, Long, Character, Boolean

断言是软件开发中一种常用的调试方式,在实现中,assertion 就是在程序中的一条语句,他对一个布尔表达式进行检查,一个正确的程序必须保证这个 boolean 表达式的值为 true,若该值为 false,则说明程序已经处于不正确的状态下,系统将给出警告并退出。

什么时候使用断言? 可以在预计正常情况下不会到达的任何位置上放置断言,断言可以用于验证传递给私有方法的参数。不过,断言不应应用于验证传递给公有方法的参数。因为不管是否启用了断言,公有方法都必须检查参数,不过既可以在公有方法中,也可以在非公有方法中利用断言测试后置条件。另外,断言不应该以任何方式改变程序的状态。

main 方法必须是 public 的

public static void main() 是对的

static void main(String [] args) 是错的,因为默认为 protected 的、

&& || 短路

```
int num=32;
```

S.o.p(num>>32) 结果为 32; 因为移位操作符右边的参数要先与 32 进行模运算 所以 num>>32 等价于 32>>0 num>>33 等价于 num>>1

不论 java 的参数类型是什么, 一律传递参数额副本, 若 java 是传值, 那么传递的是值的副本, 若传递的是引用, 则传递的是引用的副本。

boolean 的默认值是 false

读有很多字节数额文本文件, 用 bufferedReader

写出一段代码, 描述字符串写入文件

```
1 import java.io.*;
2
3 public class TestWriter {
4
5     public static void main(String[] args) throws IOException {
6         FileWriter fw=new FileWriter("demo.txt");
7         fw.write("abjjd");
8         fw.flush();
9         fw.close();
10
11     }
12
13 }
14
```

若 d 是 2 的次方, 则((d-1)&d)==0 比如 8-1=7 7&8==0

序列化一个对象? 只要让他实现 serializable 接口(该接口没有方法, 是一个标记接口)。静态不能被序列化, 因为他不在堆里, 用 transient 修饰的也不能序列化(即使在堆中), 使用 ObjectOutputStream 和 ObjectInputStream。

```
import java.io.*;

public class TestObjectOutputStream {

    public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {
        writeObj();
        readObj();
    }

    public static void writeObj() throws FileNotFoundException, IOException{
        ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("obj.txt"));
        oos.writeObject(new Person("lisi",38));
        oos.close();
    }

    public static void readObj() throws FileNotFoundException, IOException, ClassNotFoundException{
        ObjectInputStream ois=new ObjectInputStream(new FileInputStream("obj.txt"));
        Object o=ois.readObject();
        Person p=(Person)o;
        System.out.println(p);
    }
}
```

```

class Person implements Serializable{
    /**
     */
    private static final long serialVersionUID = 1L;
    String name;
    int age;

    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}

```

斐波拉契数列

1) 采用递归的方法

```

//采用递归的方式
private static int test(int n) {
    if(n==1)
        return 1;
    if(n==2)
        return 1;
    else return test(n-1)+test(n-2);
}

```

2) 采用非递归

```

//采用非递归的方式
private static int test1(int n) {
    if(n==1)
        return 1;
    if(n==2)
        return 1;
    int n1=1;
    int n2=1;
    int t=0;
    for(int x=3;x<=n;x++){
        t=n1+n2;
        n1=n2;
        n2=t;
    }
    return t;
}

```

在 main 方法中

```

For(int x=0;x<=10;x++)
    Integer k=new Integer(i);
    S.o.p("hello");

```

//编译出错，for 循环可以不使用{}，但是仅限于执行语句（其中不包括变量声明语句）

加上{}后:

```
For(int x=0;x<=10;x++){
    Integer k=new Integer(i); //编译通过
    S.o.p("hello");
}
```

若语句执行次数为常数, 则时间复杂度为 $O(1)$

$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

For(int x=0;x<=n;x++) $O(n)$

For(int j=0;j<=n;j+=n/2) $O(1)$

For(int k=0;k<=n;2*n) $O(\log_2 n)$

3 个相乘, 整体时间复杂度为 $O(n \log_2 n)$

用筛选法求 100 以内的素数? 见《面试宝典》P81, 筛选法就是从小到大筛取一个已知素数的所有倍数, 例如, 根据 2, 我们可以筛去 4,8,16。。。根据 3, 我们可以筛去 9,15, 由于 4 被筛去了, 下一个用于筛选的素数为 5, 以此类推最后剩余的就是 100 以内的素数。

垃圾收集器是自动运行的, 一般情况下, 无须显示的请求垃圾收集器, 调用 System 类的 gc 方法可以运行垃圾收集器, 但这样并不能保证立即回收指定对象。

- 1) 垃圾收集器并不是一个独立的平台, 它具有平台依赖
- 2) 一段程序可以建议垃圾回收执行, 但是不能强迫他执行
- 3) 当一个对象的 all 引用都被置为 null, 这个对象就可以变为能被垃圾回收

调用 System.gc(), 这是一个不确定的方法。Java 中并不保证每次调用该方法就一定能够启动垃圾收集, 他不过是会向 JVM 发送这样一个申请, 到底是否真正执行垃圾收集, 一切都是未知数。

一个 url 对象, 其构造函数 new URL("String s"), 比如 new URL("http://www.baidu.com"), 该字符串对于的 ip 为 202.108.33.94, 判断 URL 对象是否相等: 若两个主机名可以解析为同一个 ip 地址, 则认为两个主机相同, 即 url 对象相同; 若有一个主机名无法解析, 但两个主机名相等 (不区分大小写), or 两个主机名都为 null, 则也认为这两个主机相等。

例子:

```
String[] s={"http://www.baidu.com","www.sina.com.cn","www.uestc.edu.cn"}, 他们对象的 ip 分别为 203.109.00.93 202.108.33.94 203.109.00.93
```

```
U1=new URL(s[0]) U2=new URL(s[1]) U3=new URL(s[2])
```

则 u1.equals(u2)为 false 因为 ip 不同 u1.equals(u3)为 true 因为 ip 相同

Object 的 clone 方法 public Object clone() throws CloneNotSupportedException 创建并返回此对象的一个副本 对于任意对象 x, 则 x.clone()!=x 是 true x.clone().getClass()==x.getClass()是 true, 这些并非必须满足的条件

首先, 若此对象的类不能实现 cloneable 接口, 则会抛出 CloneNotSupportedException (注: all 数组都被视为实现了 cloneable 接口) 否则, 此方法会创建此对象的类的新实例, 严格使用此对象相应字段的内容初始化该对象的所有字段; 这些字段的内容没有被自我复制, 所以此方法的执行时该对象的“浅表复制”, 而不是“深层复制”操作。

Object 类没有实现 cloneable 接口，所以在类为 Object 的对象上调用 clone 方法时会抛出异常。

Cloneable 是一个标记接口。

浅拷贝是指拷贝对象时仅仅拷贝对象本身（包括对象中的基本变量），而不拷贝休息包含的引用指向的对象。深拷贝不仅拷贝对象，而且拷贝对象包含的引用所指向的对象。比如，对象 a1 中包含对象 b1 的引用，对象 b1 中包含对象 c1 的引用，浅拷贝 a1 得到 a2，则 a2 中仍然包含对 b1 的引用，b1 中任然包含对 c1 的引用。深拷贝是对浅拷贝的递归，若是深拷贝 a1 得到 a2，则 a2 中包含 b2（b2 是 b1 的 copy），b2 中包含 c2（c2 是 c1 的 copy）。

在 Java 中，有事会遇到子类中的成员变量 or 方法和父类同名的情况。因为子类中的成员变量 or 成员方法的优先级高，所以子类中的同名成员变量 or 方法就隐藏了父类的成员变量 or 方法，但父类同名的成员变量 or 方法仍然存在，可用 super 显示调用。

存储结构分为 4 种。1）随机存取：可以随意直接存取任何一个元素，可以通过下标直接存取任意一个元素，如数组等，有如内存，可以通过地址直接访问任意一个空间。2）顺序存取：只能从前往后逐个访问，如链表。3）索引存取：为某个关键字建立索引，从索引表中得到地址，再直接访问。4）散列存取：建立散列表。

哈夫曼编码。哈夫曼树不好画就自己百度吧，反正也不难。

基数排序 稳定 $O(n)$ 1) 当 n 较小时，比如 $n < 50$ ，可以采用直接插入排序 or 直接选择排序。2) 若文件初始状态基本有序（指正序），则应该使用直接插入，冒泡，or 随机的快速排序。3) 堆排序所需要的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。

	有序队列数据的时间复杂度	无序队列数据的时间复杂度
寻找最小值	$O(1)$	$O(n)$
估算平均值	$O(n)$	$O(n)$
找出中间值	$O(1)$	$O(n)$
找出最大出现的可能性值	$O(n)$	$O(n \log n)$

一个包含 n 个节点的四叉树，每个节点都有 4 个指向孩子的指针，这个 4 叉树有多少个控制在？答案： $4*n - (n-1) = 3n+1$ ，解释： n 个节点，一共有 $4n$ 个指针，除了 root 根节点外，all 节点都用了一个指针，其父节点用来指向他的，用了 $n-1$ 个，则剩下的 null 指针为 $4*n - (n-1) = 3n+1$ 个。

百度时，一输入“北京”，搜索框下面就会出现“北京爱情故事”，“北京公交”，“北京语言大学”等，实现这类技术用的数据结构是什么？答案：trie 树，又称为单词查找树，字典树，是一种树形结构，是一种哈希树的变种，是一种用于快速检索的多叉树结构。其核心思想是：空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。Trie 树的性质：1) 根节点不包含字符，除了根节点以外的每个节点包含一个字符 2) 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串 3) 每个节点的所有子节点包含的字符串不同。

哈夫曼编码。哈夫曼树不好画就自己百度吧，反正也不难。

辗转相除法：用来求两个自然数的最大公约数（已知 a, b, c 为正数，若 a 除以 b 余数是 c ，则 $[a, b] = [b, c]$ ，其中 $[a, b]$ 表示 a 和 b 的最大公约数）代码见 p14 反 时间复杂度 $O(n \log n)$
 $a * b = \text{最大公约数} * \text{最小公倍数}$ 。

```
private static int gongyueshu(int a, int b) {
    int c=a%b;
    while(c!=0){
        a=b;
        b=c;
        c=a%b;
    }
    return b;
}
```

数组中有一个数字出现的次数超过了数组长度的一半，找出他，没有就返回-1？先排序，找中间那个，遍历一次数组，统计中间那个元素出现的次数，若超过一半则返回，否则返回-1.

Java 中的大数类。BigInteger 1) 该类在 java.math 包中 2) 他的其中一个构造器为 new BigInteger(String s), 将 BigInteger 的十进制字符串表示转换为 BigInteger, 若要将 int 型的转为 BigInteger, 则 BigInteger two=new BigInteger(“2”); 双引号不能省略。 BigInteger oneo=new BigInteger(“1”); one.add(two) 其中 two 必须为 BigInteger 类型，并且 add() 方法返回一个 BigInteger 对象，其值为 3，one 中的值不变。

约瑟夫环 《Java 面试宝典》P193

```
public class YeSeFuHuanXiaHai {
    //java编程思想193页的题
    public static void main(String[] args) {
        boolean [] b=new boolean[30];
        for(int i=0;i<b.length;i++){
            b[i]=true;
        }

        int len=b.length;
        int index=0;
        int count=0;
        int he=0;

        while(he!=15){
            count++;
            if(count==9){
                count=0;
                while(b[index]==false){
                    //若已经被标记为false, 则往下移一个
                    index++;
                }
                b[index]=false;
                he++;
            }
            index++;
            if(index==len)
                index=0;
        }
    }
}
```

```

        int zonghe=0;
        for(int x=0;x<len;x++){
            System.out.println(x+"的值为: "+b[x]);
            if(b[x]==false)
                zonghe++;
        }

        System.out.println("总共有"+zonghe+"个FALSE");
    }
}

```

a 对应的 ASCII 码为 97 A: 65 。

queue 的方法中，抛出异常：add remove element 返回特殊值：offer poll peek。

“我”.getBytes().length()=2; “我”.toCharArray().length()=2;

优先级队列（PriorityQueue）的头是按指定排序方式确定的最小元素。

只有具有执行权限（execute）才允许用户进入一个文件系统的目录。

linux 32 位 os 下，一个应用程序最多分配和访问的内存大小？ 答案：3G，32 位可以映射为 4G，有 2G 的用户模式虚拟地址空间位于 4G 地址空间的一半，而与之相应的另一半 2g 地址空间由 os 内容使用。具体解释见《面试宝典》p211

若系统有 5 台绘图仪，有多个进程均需要使用 2 台，规定每个进程一次仅允许申请 1 台，则至多有多少个进程参与竞争而不发生死锁？ 答案：4 个，哲学家就餐问题，最多允许 4 个哲学家同时进餐，以保证至少有一个哲学家能够进餐，最总总会释放出他们所使用过的 2 只筷子，从而可以使更多的哲学家进餐。

win32 环境中，线程有 3 种线程模式，单线程（一个线程完成 all 工作，一个人搬房子），单元线程（all 线程都在主应用程序内存中各自的子段范围内运行，此模式允许多个代码实例同时单独运行，找朋友搬房子，每个朋友在单独的房间工作，并且不能帮助在其他房间工作的人），自由线程（多个线程可以同时调用相同的方法和组件，与单元线程不同，自由线程不会被限制在独立的内存空间，找朋友搬房子，all 朋友可以随时在任何一个房间工作）

同步：一个进程在执行某个请求时，若该请求需要一段时间才能返回消息，那么这个进程将会一直等下去，直到收到返回信息才继续执行下去。异步：进程不需要一直等下去，而是继续执行下面的操作，不管其他进程的状态，当有消息返回时，系统会通知进程进行处理，这样可以提高效率。

范式 1NF：第一范式：强调列的原子性，列不能再分为其他几列，属性不可分。 2NF：首先是 1NF，另外包含两个部分，1）一个表必须有主键 2）没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分 3NF:首先是 2NF，非主键列必须直接依赖于主键，不能存在传递依赖，即不能存在非主键 A 依赖于非主键 b,非主键 b 依赖于主键的情况，消除传递依赖。

存储过程 VS 函数 都是为了重复执行 sql 语句的集合 他们的区别：1) 写法上：存储过程的参数列表可以有输入参数，输出参数，输入输出参数，函数的参数列表只有输入参数。2) 返回值：存储过程的返回值，可以有多个值，函数只能有 1 个。3) 调用方式：存储过程 1.exec<过程名> 2.execute<过程名> 3.在 sql 语句中直接调用 函数：在 sql 语句中直接调用。

游标用于定位结果集的行，通过判断全局变量 @@FETCH_STATUS，可以判断游标是否到达了最后，通常此变量不等于 0 表示出错 or 到了最后。

事前触发器运行于出发事件之前，事后触发器运行于出发事件之后。语句级触发器：触发器只会在相应的语句执行前 or 执行后执行一次，行级触发器：该语句影响了几行，就要执行几次触发器。

执行查询时，若要查询的数据很多，假设要查询 1000 万条，用什么方法提升效率？答案：1) 从数据库方面：建立索引、分区、尽量使用固定长度的字段、限制字段长度 2) 从数据库 IO 方面：增加缓冲区 3) 在 sql 语句方面：优化 sql 语句，减少比较次数、限制返回的条目数 4) 在 java 方面，如果是反复使用的查询，使用 preparedStatement。

NAT：网络地址转换，常用于私有地址与公有地址的转换，以解决 ip 地址匮乏的问题。

A: 1-127 B: 128-191 C: 192-223

138.96.0.0/16 其中 16 表示子网掩码 1 的个数，则其子网掩码为 225.225.0.0, 1 为网络号，0 为主机号。

在一个 ip 数据包到达目的之前，他不可能重组，但是可以分散为碎片。

win2000 os 中 1) 配置 ip 地址的命令是 ipconfig 2) 用 ping 来测试本机是否安装了 tcp/ip 协议 3) 列出本机当前可建立的链接 netstat -a

ping 基于什么协议？答案：icmp，是 tcp/ip 协议族的额自协议，在网络层，用于在 ip 主机、路由器之间传输控制消息。控制消息是指网络通不通，主机是否可达、路由是否可用等网络本身的消息。这些控制信息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。Ping, tracert 都是基于 icmp。可以利用 os 规定的 icmp 数据包的最大尺寸不超过 64k，向主句发起 ping of death 攻击 该攻击原理：若 icmp 数据包尺寸超过 64k 上限，主机就会出现内存分配错误导致 tcp/ip 堆栈奔溃，致使主机死机。防范方法：1) 在路由器上对 icmp 数据包进行带宽限制，将 icmp 占用的带宽控制在一定的范围，这样即使有 icmp 攻击，他所占用的带宽也非常有限。2) 在主机上限制，设置 icmp 数据包的处理规则，最好是设定拒绝 all 的 icmp 数据包。

每经过一个路由器，ttl 减 1，到 0 时丢弃。

应用网关 1) 可以是一个代理服务器 2) 可以被用来链接公司内部网络应用和公司外部网络应用 3) 可以是防火器构造的一部分。

0 和 127 不作为 A 类地址。子网掩码只有一个作用，就是将某个 ip 地址划分为网络地址和主机地址两部分。两台计算机各自的 ip 地址和子网掩码进行与操作运算后，若得出相同结果，则说明这两台计算机处于同一个子网中。

1) 使用 usb2.0 闪存盘，往 usb 闪存盘上拷贝文件的数据传输效率。2) 使用 100Mb/s 以太网，在局域网内拷贝大文件时网络的传输效率。3) 使用一辆卡车拉 1000 块单块 1TB 装满数据的硬盘，以 100km/h 的速度从北京到天津（100km）一趟所等价的数据传输带宽。4) 使用电脑播放 MP3，电脑的 pci 总线到声卡的数据传输速率 传输速率排行：4<1<2<3 普通 U 盘写数据约为 6MB/s=48Mb/s;100Mb 以太网的速率为 100Mb/s; 卡车拉硬盘 $1000 \times 1000 \times B / 3600 = 2222 \text{ Mb/s}$; MP3 在 256kb/s 码率下也有平均只有 1min2MB，所以约 0.3Mb/s。

Class.forName(String s)方法返回与带有给定字符串名的类 or 接口相关联的 class 对象，这将导致命名为 s 的类被加载。

数据库连接池的工作机制？答案：J2ee 服务器启动时会建立一定数量的链接，并一直维护不少于此数目的池连接。客户端程序需要连接时，池驱动程序就会返回一个未使用的池连接并将其标记为忙。若当前没有空闲连接，池驱动程序就建立一定数量的连接，新建连接的数量由配置参数决定。当使用的池连接调用完成后，池驱动程序将此连接标记为空闲，其他调用就可以使用这个连接。

forward VS redirect 区别 1) forward 是服务器请求资源，服务器直接访问目标地址的 url，把 url 的响应内容读取出来，然后把这些内容发送给浏览器，浏览器根本不知道服务器发送的内容是从哪取出来，所以他的地址栏中还是原来的地址。Redirect 服务器根据逻辑发送一个状态码，告诉浏览器重新去请求事先访问过的那个地址，一般来说，浏览器会用刚才请求的 all 参数重新请求，所以 session，request 参数都可以获取。地址栏显示的是新的 url。用 redirect 等于客户端向服务器端发两次 request，同时也接受两次 response。2) forward: 转发页面和转发到的页面可以共享 request 的数据；redirect 不能共享数据 3) forward: 只能在同一个 web 应用程序之间转发请求；redirect: 不仅可以重定向到当前应用程序的其他资源，还可以重定向到同一个站点上的其他应用程序中的资源，甚至使用绝对 url 重定向到其他站点的资源。4) forward: /代表当前 web 应用根目录 redirect: /代表当前 web 站点的根目录。

JSP 内置对象。1) request: 用户端的请求，此请求会包含来着 get or post 请求的参数。2) response: 网页传回用户端的响应。3) pageContext: 网页的属性 4) session: 与请求有关的会话 5) application: servlet 正在执行的内容 6) out: 传送回应的输出 7) config: servlet 的部件 8) page: jsp 网页本身 9) exception: 网页错误

jsp 有哪些动作？6 个 1) jsp:include 在页面被请求时，引入一个文件 2) jsp:useBean 寻找 or 实例化一个 JavaBean 3) jsp:setProperty 这只 javaBean 的属性 4) jsp:getProperty 输出某个 javaBean 的属性 5) jsp:forward 把请求转到一个新页面 6) jsp:plugin 根据浏览器类型为 java 插件生成 object or embed 标配。

jsp 中动态 include 与静态 include 区别？答案：动态 include 用 jsp:include 动作实现，它总是会检查所包含文件中的变化，适用于包含动态页面；静态 include 用 include 伪码实现，不会检查所含文件的变化，适用于包含静态页面。

```
<jsp:include page="include.jsp" flash="true">
```

```
<jsp:forward page="next.jsp">
```

前者页面不会转向 include 所指向的页面，只显示该页的结果，主页面还是原来的页面，指向完以后还会回来，相当于函数调用，并且可以带参数。后者完全转向新页面，不会再回来，相当于 goto 语句。

servlet VS CGI 区别 1) servlet 可移植 跨平台 CGI 不行 2) 在传统 CGI 中每个请求都要启动一个新进程，若 CGI 程序的本身执行时间较短，启动进程所需要的进行所需要的开销很可能反而超过实际执行的时间；servlet，每个请求由轻量级的 java 线程处理 3) 在传统 CGI 中，若有 N 个并发的对同一个 CGI 的请求，该 CGI 的程序代码在内存中重载了 N 次，对于 servlet，处理请求的是 N 个线程，只要一份 servlet 类代码。

如何实现 servlet 单线程模式？答案：要实现单线程模式，可以在配置文件中修改 isThreadSafe 属性，比如，

```
<%@page isThreadSafe="false"%>
```

servlet 页面间对象传递的方法有几种？答案：用 request, session, application。Cookie 等方法实现页面间的对象传递。

jsp VS servlet: jsp 是 servlet 技术的扩展，本质上是 servlet 的简单方式，jsp 编译后是“类 servlet”。他们最主要不同在于：servlet 的应用逻辑在 java 文件中，并且完全从表示层中的 html 分离出来。Jsp 是 java 和 html 可以组合为一个扩展名为.jsp 的文件。Jsp 侧重视图，servlet 侧重控制逻辑。

xml 解析技术？答案：1) dom：必须在解析之前把整个文档装入内存，处理大型文件时，性能低，适合 xml 随机访问。2) sax：事件驱动，它顺序读取 xml 文件，不需要一次全部载入整个文件。当遇到文件开头，文档结束 or 标签开头 or 标签结束时，他会触发一个事件，用户通过在其回调事件中写入处理代码来处理 xml 文件，适合顺序访问。

J2ee 号称多层结构，为什么多层比两层好？答案：多层结构解耦性好，使得维护与扩展方便，灵活。

典型的 j2ee 至少划分为 3 层：表现层、业务逻辑层、持久层。

测试 1) 语句覆盖：至少每个语句应该执行一次，最弱的逻辑覆盖标准。2) 判定覆盖：每个判定的每种可能结果都要执行一次，建立判定表之后，要保证每种判定的结果中都包含了 T 和 F。3) 条件覆盖：不但每个语句要执行一次，而且判定表达式中的每个条件都要取到可能的结果，建立判定表以后，要保证每种条件的结果中包含了 T 和 F。4) 判定-条件覆盖：每个判定及每个判定中的每个条件都取到可能的结果，建立判定表后，要保证每个判定结果包含 T 和 F，而且每个条件的结果包含 T 和 F，也就是综合了上面的判定覆盖和条件覆盖。5) 条件组合覆盖：每个判定中的条件的各种组合至少出现一次，也就是说，先把程序中的条件列出来，排列组合写出 all 可能性，看有没有哪些值同时满足这些排列组合。6) 路

径覆盖：每条可能的路径至少执行一次。

功能测试：也称为黑盒测试，只考虑各个功能，不考虑整个软件的内部结构及代码。可用性测试：用户在和系统交互时对用户体验质量的度量，由用户测试。

边界测试就是找到边界，then 在边界附近（两边）选点。

3,16,37,?,289 应该填 100，因为 $16-3=7$, $37-16=21$, $100-37=63$, $289-100=189$

2012!末尾有几个 0？答案：乘机会产生 0 的就是 2 的倍数与 5 的倍数相乘产生的。 $2012/5=402$, $402/5=80$, $80/5=16$, $16/5=3$, $402+80+16+3=501$, 所以末尾公有 501 个 0。

tomcat 服务器默认端口 8080，启动 tomcat 用 bin 目录下的 startup.bat--用的是 catalina.bat.

jsp 中 java 代码写在<% %>中。

Load-on-startup: 配置在 servlet 节点中，用来指定 servlet 实例被创建的时机。若为负数，则在第一次被请求时创建，若为 0 或正数，则在当前 web 应用被 servlet 容器加载时创建实例，且数值越小则越早被创建。

一个 servlet 可以有多个 servlet-mapping 对其进行映射。

Servletconfig: init 函数的参数，他封装了 servlet 的配置信息，并且可以获得 servletContext 对象。

Servlet 引擎为每个 web 应用创建一个 servletcontext 对象，一个 web 应用程序中 all servlet 都共享一个 servletcontext 对象，他对应一个 web 应用。

Servletrequest 里封装了 all 有关请求的信息，但要查看是 get 方式还是 post 方式，必须用 httpServletRequest 的 getMethod()方法。

Jsp 可以放在 web 应用程序中除了 web-inf 中。每个 jsp 页面在第一次被访问时，jsp 引擎将它翻译为一个 servlet 源程序，接着再把这个 servlet 源程序编译成 servlet 的 class 文件，然后再由 web 容器（servlet 引擎）像调用普通 servlet 程序一样来装载和解释这个由 jsp 页面译成的 servlet 程序。

Jsp 九个内置对象。1) request: httpServletRequest 2) response: httpServletResponse, 在 jsp 页面中几乎不用 response 任何方法 3) pagecontext: 页面的上下文，是 pagecontext 的一个对象，可以从该对象获得其他 8 个对象及页面的其他信息。4) session: 代表浏览器和服务器的会话，是 HttpSession 对象。5) application: 代表当前 web 应用，是 servletContext 对象。6) config: 当前 jsp 对应的 servlet 的 ServletConfig 对象，开发时几乎不用 7) out: JspWriter 对象，调用 out.println()可以直接把字符串打印到浏览器。8) page: 只能调用 Object 方法，开发时几乎不用，指向当前 jsp 对应的 Servlet 对象的引用。9) exception: 在声明了 page 指

令的 iserrorpage= “true” 时才可以使用。

Jsp 注释<%-- --%>

和属性相关的方法 1) object `getAttribute(String name)`:获取指定的属性 2)Enumeration `getAttributeNames()`: 获取 all 的属性的名字组成一个 enumeration 对象 3)`removeAttribute(String name)`: 移除指定的属性 4)`setAttribute(String name,Object 0)`: 设置属性。

`pageContext`, `request`, `session`, `application` 对象都有这些方法, 这 4 个对象为域对象。
Pagecontext: 属性的作用范围仅限于当前 jsp 页面。**Request**: 属性的作用范围仅限于同一个请求。**Request**: 属性的作用范围限于一次会话。浏览器打开直到关闭称之为一次会话(在此期间会话不失效)。**Application**: 属性的作用范围限于当前 web 应用, 是范围最大的属性作用范围。只要在一处设置属性, 在其他 jsp 和 servlet 中都可以获取到。

请求重定向 VS 请求转发 1)请求 `HttpServletRequest` 的 `getRequestDispatcher()`方法, 获取 `requestDispatcher` 对象, 调用该方法时, 传入需要转发的地址 2)调用 `requestDispatcher` 的 `forward(request, response)`进行请求的转发。

请求重定向, 两次请求, 直接调用 `response` 的 `sendRedirect(path)` 方法。
`Resp.sendRedirect(location)`。

Jsp 指令<%@ %> <%@include %>静态引入嵌入整个源码, 生成一个 servlet 源文件。Jsp 指令是为 jsp 引擎设计的, 只是告诉 jsp 引擎如何处理 jsp 页面中的相关部分。
`pageEncoding`: 指定当前页面的字符编码 `charset`: 指定返回页面的字符编码。

<jsp:include>动态引入 生成两个 servlet 源文件。

Java 中中文乱码问题 1) 在 jsp 页面上出入中文, 提交页面后不出现乱码 `pageEncoding,charset` 的编码一致, 且都支持中文, 建议为 utf-8, 还需要保证浏览器显示的字符编码也和请求的 jsp 编码一样。2)获取参数中的中文, 默认参数在传输过程中使用 iso-8859-1 1. 对于 post 请求: 值需要在获取请求信息之前调用 `request.setCharacterEncoding("utf-8")` 2. 对于 get 请求, `String username =new String(val.getBytes(iso-8859-1),"utf-8")`,先用 iso-8859-1 解码, 再用 utf-8 编码。

多个请求映射到同一个 jsp 页面? 答案: 1) 这几个请求的名字都为 xxx.do,all 以.do 结尾的都映射到同一个 servlet。2) 用 `request` 获得 `path`, 即/xxx.do。3) 去掉后面的.do 和前面的"/", 得到方法名,即 xxx。4) 利用反射调用相关方法。

Httpsession 生命周期: 1) 什么时候创建 `httpsession`? 答案: 1.是否浏览器访问服务器的任何一个 jsp 或者 servlet, 服务器都会创建一个 `httpsession`? 不一定, 设置 `session=false`, 若当前 jsp 是客户端访问的当前 web 应用的第一个资源, 且 `page` 指定的 `session=false`, 则服务器不会为 jsp 创建 `session` 对象。若当前 jsp 不是客户端访问的当前 web 应用的第一个资源, 且其他页面已经创建了其他 `httpsession` 对象, 则返回一个和当前会话相关的 `httpsession` 对

象，不创建一个新对象。2.session=false 是什么意思？当前 jsp 页面禁用 session 隐含变量，但可以使用其他的显示的 HttpSession 的对象。3.对于 servlet，若 servlet 是客户端访问的第一个 web 应用资源，只有调用 request.getSession() 或除了 1 和 2 以外，只要访问 jsp，则要创建 session 对象。2) 什么时候销毁 session 对象？答案：1.直接调用 HttpSession 的 invalidate 方法，该方法使 HttpSession 失效。2.当前 web 应用被卸载。3.超出 HttpSession 的过期时间。

建议使用绝对路径，在 java 中什么是绝对路径？相对于当前 web 应用根路径的路径。

JavaBean：就是一个特殊的 java 类，用作 JavaBean 的类必须具有一个公共的无参数的构造函数，Setter 和 getter。JavaBean 的属性名是根据 setter 和 getter 来生成，且首字母必须小写。

EL 表达式 Expression Language 表达式语言 all 的 EL 都以 \${ } 开始。

EL 隐含对象 11 个 1) 和范围有关 application scope, session scope, request scope, pagescope 2) 和输入有关 param, paramValues 3) 其他隐含对象 cookie, header, headerValues, initparam, pageContext。

自定义标签：1) 先写一个标签处理器类，实现 SimpleTag 接口，或者继承 SimpleTagSupport 类。2) 建一个 tld 文件，一个描述文件。3) 在 jsp 中使用自定义标签。4) 在 jsp 页面中使用 jsp 标签 <%@ taglib %> 引入。

jspFragment 封装 jsp 标签体。jspFragment.invoke(writer out)。

若配置的标签有标签体，则 jsp 引擎会调用 setjspBody() 方法把 jspFragment 传递给标签处理器，我们可以使用 SimpleTagSupport 的 getjspbody 方法来返回 jspFragment。



子标签作为父标签的标签体存在，父标签无法获取子标签引用，但子标签可以获取父标签引用。

EL 自定义函数 1) 写一个静态函数实现功能 2) tld 文件中配置

JSTL (jsp 标准标签函数库)：使页面上没有 java 代码，用 el 或 jstl 或其他自定义标签。

<c:choose> 必须是 <c:when> 和 <c:otherwise> 的父标签且在同一个 <c:choose>，<c:when> 必须在 <c:otherwise> 之前。

<c:forEach> 为循环控制，它可以将集合 collection 中的成员循环浏览一遍，运行方式为当前条件符合时，就会持续重复执行 <c:forEach> 的本体内容。

Filter 过滤器 filter 的基本功能是对 servlet 容器调用 servlet 的过程进行拦截，从而在 servlet 进行相应处理的前后实现一些特殊功能。Filter 程序是一个实现了 Filter 接口的 Java 类，他由 servlet 容器进行调用和执行。Servlet 容器  filter  servlet。

Filter 1) init() 方法类似于 servlet 的 init 方法，创建 filter 对象后，立即被调用，且只被

调用一次。Filter 对象在 servlet 容器加载当前 web 应用程序时就被创建。Filter 实例是单例的。2) doFilter()方法，逻辑代码写在这里面，每次拦截器都会调用的方法。他有 3 个参数，servletrequest, servletresponse, filterchain（指 filter 链，多个 filter 可构成一个 filter 链，filterchain 只有一个方法 doFilter(request, response),把请求传给 filter 链的下一个 filter，若当前 filter 是 filter 链的最后一个 filter，将把请求传给目标 servlet（or jsp））。若有多个 filter 满足要求，则拦截顺序与 web.xml 中的 filter-mapping 的配置顺序有关。先配置先调用。3) destroy: 释放当前 filter 所占用的资源，filter 销毁之前被调用，且只被调用一次。

与开发 servlet 不同，Filter 接口并没有相应的实现类可以继承，要开发过滤器，只能直接实现 filter 接口。

多个 filter 代码的执行顺序

HelloFilter 的 doFilter 方法 有 3 个方法

S.O.P(before hello);

doFilter();//调用的是 secondFilter 的 doFilter 方法

S.O.P(after hello);

secondFilter 的 doFilter 方法 有 3 个方法

S.O.P(before second);

doFilter();//调用的是 jsp 的 doFilter 方法

S.O.P(after second);

jsp 的 doFilter 方法

<% S.O.P(test jsp);%>

这 5 个输出语句的执行顺序是：

S.O.P(before hello);

S.O.P(before second);

S.O.P(test jsp);

S.O.P(after second);

S.O.P(after hello);

Filter 的 dispatcher 节点（表明拦截哪种方式的页面）默认为 request，还可以的取值有 forward, error, include。Request: 直接发 get 或 post 请求，当用户直接访问页面时，web 容器将会调用过滤器。Include: 若目标资源是通过 requestdispatcher 的 include 方法访问，则该过滤器将被调用。Forward: 1) <jsp:forward page=...> 2) requestdispatcher.forward 3) 通过 page 指令的 errorpage 转发页面。Error: 若目标资源是通过声明式异常处理机制调用时，则该过滤器将被调用。Dispatcher 节点可以有多个。

Filter 的应用: 1) 禁用浏览器的缓存。2) 字符编码过滤器。若没有过滤器则在每一个请求页面中都要写 request.setCharacterEncoding，有了 filter，对于 all 请求都经过 filter，只需要写一次。3) 检测用户是否登录的 filter。

Filter 是 javaweb 的一个重要组件，可以对发送到 servlet 的请求进行拦截，并对相应也进行拦截。

URL: http://localhost:8080/day_01/login/a.jsp（其中 day_01 为工程名字）

URI: day_01/login/a.jsp
Servletpath: login/a.jsp

JavaWeb-servlet 监听器

监听器：专门用与对其他对象身上发生的事件或状态的改变进行监听 or 相应处理的对象。
当被监视的对象发生情况时，立即采取相应的行动。

Servlet 监听器：一种特殊的类，用于监听 ServletContext, HttpSession, ServletRequest 等域对象的创建与销毁事情，以及监听这些对象中的属性发生修改的事件。

一、监听域对象的创建和销毁

	创建时间	销毁时间
ServletContext (application)	web 应用被加载	web 应用被销毁
HttpSession (session)		
ServletRequest (request)	每次请求开始时创建	每次访问结束后销毁

ServletContextListener 最为常用，监听 ServletContext 对象创建 or 销毁的 Servlet 监听器，可以在当前 web 应用被加载时对当前 web 应用的相关资源进行初始化，比如：创建数据库连接池。HttpSessionListener: sessionCreated 在 session 被创建后调用, sessionDestroyed 在 session 被销毁前调用。

生命周期：1) request: 是一个请求，当一个相应返回时被销毁，发一个请求时被创建
1.forward 只有一个请求，只有一个 request 2.response 有两个 request。2) session: 当第一次访问 web 应用的一个 jsp 或 Servlet 时，且该 jsp 和 Servlet 中还需要创建 session 对象，此时服务器会创建一个 session 对象。3) application: ServletContext 对象。当前 web 应用被加载时创建，web 应用被卸载时销毁。

二、域对象中属性的变更的监听器

变更：添加，置换，删除。

要使用监听器：写一个类，实现相对应的接口，在 web.xml 中配置。

三、感知 session 绑定的事件监听器，不用再 web.xml 中配置

1) 实现了 HttpSessionBindingListener 的 javabean 可以感知自己是否放入 session 属性中 or 被移出，有两个方法

1. public void valueBound(): 当前对象被绑定到 session 时调用。
2. public void valueUnbound(): 当前对象从 session 中解除绑定时被调用。

2) 活化：从硬盘里读出

钝化：把自己写入硬盘

实现了 HttpSessionActivationListener 可感知自己被活化 or 钝化。

关了 web 服务器，session 本应该销毁，但有一个缓存机制，把 session 存放在 tomcat 服务器的 work/catalane/localhost/contextpath 目录下 session.ser。

项目管理工具，我看的是慕课网上 maven 的视频。

RandomaccessFile: 用来访问哪些保存数据记录的文件，可以用 **seek** 方法进行访问，并进行读写。这些记录的大小不必相同，但是其大小和位置必须可知，该类仅限于操作文件。**RandomaccessFile** 不属于 **InputStream** 和 **OutputStream** 类系。随机访问文件的行为类似存储在文件系统中的一个大 **byte** 数组，存在一个光标。

任务优先级是一个整数型，任务调度器会根据这个值来确定合适执行这个任务。

ResultSetMetaData: 可用于获取关于 **ResultSet** 对象中列的类型和属性信息的对象。

创建类实例的方法: 1) **new** 2) **Class.forName(类全名).newInstance()** 3) **clone()**.

Java 5 个特性: 面向对象，跨平台，多线程，健壮性，解释性。

Java 中 **float** 默认 **0.0f**，**double** 默认 **0.0d**

String 对象一旦被创建，就不可以改变

java.util.regex 使用正则表达式来实现模式匹配。

CheckedException VS **UncheckedException**

将派生于 **Error** 或者 **RuntimeException** 的异常称为 **UncheckedException**，所有其他异常称为 **CheckedException**。若出现了 **RuntimeException**，则一定是程序员自己的问题。

Exception 一般分为 **IOException** 和 **RuntimeException**。

Java 实现封装用的是 **private**。

守护线程 (**daemon thread**)，必须声明在 **start** 方法之前。

Thread 的 **yield()**: 暂停当前正在执行的线程，并执行其他线程。

Swing 组件里用到 **MVC**。

DeflaterOutputStream 和 **InflaterInputStream** 在 **java.util.zip** 包下

Java 中所有带缓存机制的类的默认缓冲区大小为 **512bytes**。

算法的基本概念

输出、输入，有穷性，确切性，可行性

History 对象是包含用户访问过的 **URL**，是 **Window** 对象的一部分，他的几个方法 1) **length**: 返回浏览器历史列表中的 **URL** 数量。2) **back**: 加载 **history** 列表中的前一个 **URL**。3) **forward**: 加载 **history** 列表中的下一个 **URL**。4) **go**: 加载 **history** 列表中的某个具体页面。

超链接的下划线默认是有的，去掉的方法是 `a{text-decoration:none}`

选择排序：每扫描一遍数组，只交换一次。

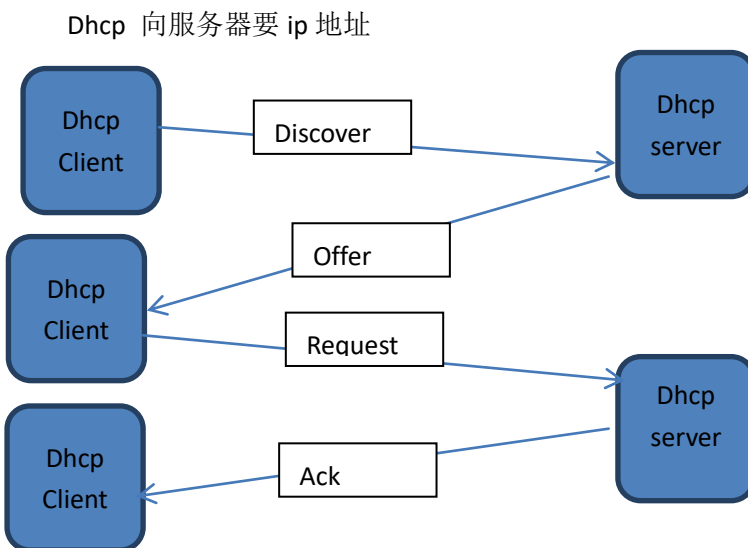
基于比较的排序算法：插入、希尔、选择、冒泡、归并、快速

链地址法的平均查找长度

$H(k) = k \bmod 7$ 有一个数组 32 24 15 27 20 13
下标 0 1 2 3 4 5 6
15 (1) 24(1) 32(1) 27(1)
20(2)
13(3)

所以平均查找长度为 $(1+1+1+1+2+3)/6=1.5$

简单网络管理协议 SNMP 协议的组成部分：SNMP 本身+管理信息结构 SMI+管理信息库 MIB，SNMP 报文组成部分：版本+首部+安全参数+SNMP 报文的数据部分。



发现阶段：dhcp Client 寻找 dhcp 服务器的过程，收到 discovery 的 server 都会回复。Offer：表示该服务器可以给他提供 ip 地址。Request：客户端选择一个 server 来要 ip。Ack：把 ip 给客户端。

读屏障用于保证读操作有序，屏障之前的读操作一定会优先于屏障之后的读操作完成，写操作不受影响。

优化屏障用于限制编译器的指令重排。

通用屏障则对读写操作都有用。

Blob 对象中的 type 属性表示文件的 MIME 类型。

Linux 用户磁盘配额配置文件 aquota.user 的默认访问权限是 600.

Location 和 history 对象和浏览器列表有关。

Padding 透明且可以显示背景。

DNS 区域配置文件，默认有 localhost.zone 和 named.local

二叉排序树又名二叉搜索树，左<根<右

FTP 端口 20：传输数据 21：传输控制信息。

中断方式一般用于处理随机出现的服务请求。DMA 方式数据传送不需要 cpu 控制。DMA 和 cpu 必须同时使用总线。

一个流程图称为可归约的等价于流程图中去除回边外，其余的边构成一个无环路流图。

耦合程度从高到低 内容耦合>标记耦合>数据耦合

PAD 图：问题分析图，描述软件的详细设计，只能描述结构化程序运行使用的几种基本结果。

计数排序：基于统计 时间复杂度 $O(n)$ 空间复杂度 $O(n)$ ，采用计数排序法的辅助数组长度为 $\max - \min + 1$ ，（ \max 和 \min 分别为待排序数组的最大值和最小值），比如数组为 1,0,3,1,0,1,1， $\max - \min + 1 = 3 - 0 + 1 = 4$ 。

线性窥孔优化：1）既可以是中间代码，又可以是目标级代码 2）每次处理的只是一组相邻指令，相当于将一组相邻指令暴露在一个优化窗口中（正如窥孔的含义）。3）对优化对象进行线性扫描 4）优化后产生的结果可能会给后面的代码提供进一步优化的机会 5）窥孔优化程序通常很小，只需很少的额内存，执行速度很快。

DNS 服务中资源记录类型 PTR A CNAME

直接插入排序是从当前位置往前插

N-S 图是用于取代传统流程图的一种方式，去掉了流程线，可在总体设计时使用。

当用 n 比特进行分组编号时，若接受窗口为 1（即只能按序接受分组），那么若要求连续 arq 协议能正常运行时，发送窗口大小不超过 $2^n - 1$ 。

Java 中的线程由一个虚拟处理机、cpu 执行的代码以及代码操作的数据三部分组成。

几个类通过编译后就产生几个字节码文件。

String 传的也是副本。

Math.ceil(a) 返回 $\geq a$ 的最小
Math.floor(a) 返回 $\leq a$ 的最大
Math.round(a) 四舍五入

Byte, short, Int, Long, Float, Double, Character, Boolean 都在 java.lang 包中。

Float a=1.0f; 是对的
Float a=1.0; 是错的, 因为 1.0 默认为是 double 类型, 必须强转为 float。

Long 和 float 正常定义要加上 L 和 F byte 的范围是-128 到 127

所有 byte, short, char 类型的值将被提高为 int 进行计算。被 final 修饰的变量不会自动改变类型, 当 2 个 final 修饰的变量操作时, 结果根据左边类型而转化。

Java 中数组是对象, 不是原生类。

持久化 java 堆溢出: 使用 cglib 技术直接操作字节码运行, 产生大量的动态类; 年老代溢出: 上万次字符串处理, 创建上万个对象或在一段代码内申请上百 M 甚至上 G 的内存。

GetDeclaredMethods(): 返回 method 对象的一个数组, 这些对象反应 class 对象表示的类 or 接口声明的 all 方法, 包括公有、私有、默认和保护方法, 但是不包括继承的方法。
GetMethods(): 反应此 class 对象表示的类 or 接口的公共 method 方法。

一个 AOV 网的拓扑排序可能不唯一。对 aov 网进行排序的基本思想: 1) 从 aov 网中选择一个没有前驱的顶点输出他 2) 从 aov 网中删去该节点, 并且删去 all 以该节点为尾的弧 3) 重复上述步骤, 直到 all 点输出 or aov 网中不存在没有前驱的点。

曼切斯特编码。从低到高表示 1, 从高到低表示 0, 使用他的目的是实现对通道过程中收发双方的数据同步。

SMTP: 邮件服务器之间传递报文。

Mac 48bit ipv4:32 ipv6:128

BGP 是在自治系统之间的路由协议, 当前英特网路由选择协议分为内部网关协议和外部网关协议。

线性结构分为顺序结构和链接存储结构。

Hash 表的做法其实很简单, 就是把 key 通过一个固定的算法函数, 即所谓的 hash 函数转换为一个整形数字, then 将该数字对数组长度进行取余, 取余结果就当做数组下标, 将 value 存在在以该数字为下标的数组空间里。

要找最大的 k 个数，建小顶堆。

AOV 网：用顶点表示活动，用弧表示活动间的优先关系，在网中，若从顶点 i 到 j 有一条有向路径，则 i 为 j 前驱。对于一个 aov 网，从源点到终点的路径称为关键路径（错的），因为最早完成时间=最晚完成时间的点构成的路径为关键路径。

`int a[]=new int[]; a++;`（这句话是错的，因为 a 是常量，是地址）。

外部排序常用的是归并排序。

在 `hashmap` 中，若 `key` 为 `null`，则 `put` 时调用 `putForNullKey(value)` 方法，该方法是在 `table[0]` 中查找 `key` 为 `null` 的元素，若找到，则将 `value` 重新赋值给这个元素的 `value`，并且返回原来的 `value`，若没有，则把（`null`，`value`）添加到 `table[0]` 中。

InetAddress：表示互联网 or 局域网一台主机的地址。

```
InputStream 和 outputStream 之间传数据需要一个 byte[], byte[] b=new byte[1024];
While(len=in.read(b)!=-1)
    Out.write(b,0,len);
```

`Tcp serverSocket` 和 `socket` 类 `UDP datagramSocket` 和 `datagramPacket` 类

Where `liename` is `null`

Where `liename` is not `null`

`Select e.last_name,e.first_name ,d.dept_no from employees e,dept_emp d where e.emp_no=d.emp_no and dept_no is not null.`（给列取别名）。

getDeclaredMethod：返回 `method` 数组，表示 `class` 对象表示的类 or 接口声明的 `all` 方法，包括公共、保护、私有，但不包括继承。

GetMethod：返回 `method` 数组，表示 `class` 对象所表示的类 or 接口的公有方法，包括从父类继承的。

Bit-map:用一个 `bit` 位来表示某个元素对应的 `value`，而 `key` 为该元素，最多有多少个元素就要多少个 `bit` 位。

集群 1) 伸缩性: 系统适应不断增长的用户数的能力。2) 高可用性: 避免单一服务器的单点失效。3) 负载均衡: 把请求发给不同服务器。4) 失效转移: 当一个节点失效后，通过选择集群中的另一个节点，处理将会继续而不会终止。**httpsession** 失效转移，用来保证当某台服务器失效以后，会话状态不会丢失，为了支持会话转移，**web** 服务器将在一定的时候把会话对象备份到其他地方，以防丢失。这个其他地方有 `jdbc` 数据库、其他所有服务器、任意选择其他一台 `web server`、中央 `server`。

Fast-fail: `java` 集合中一种错误检查机制。当多个线程对集合进行结构上的改变的操作时，

有可能会 Fast-fail。注意：是有可能，而不是一定。比如：假设存在两个线程 x，y。x 通过 iterator 遍历集合 A 中的元素。在某个时刻线程 y 修改了集合 A 的结构（是结构上的修改，而不是简单的修改集合元素的内容）。则这时程序就会抛出 ConcurrentModificationException，从而发生 Fast-fail。若单线程违反该规则，也会抛出异常。因为 modcount 与 ExceptedModcount 改变不同步。当 iterator 执行 next()、remove() 时，都要判断 modcount 与 ExceptedModcount 是否相等，若不相等，则抛异常。

CopyOnWriteArrayList 的核心：对于任何 Array 在结构上有所改变的操作（add，remove 等），CopyOnWriteArrayList 都会 copy 现有数组，再在 copy 的数组上修改。这样就不影响 iterator 中的数据，修改完成后改变原有数据的引用即可。

若不设置过期时间，则 cookie 的生命周期为浏览器。关闭浏览器，cookie 就消失。这是会话 cookie，会话 cookie 一般不存储在硬盘里，还是放在内存。若设置了过期时间，则 cookie 会被放到硬盘，关闭浏览器后再打开，cookie 仍有效，直到超过设定的过期时间。存在硬盘上的 cookie 可在不同浏览器进程之间共享，比如两个 ie 窗口。

Session 比 cookie 安全。

红黑树—使二叉搜索树更加平衡：他是一种二叉搜索树，但在每个节点上增加一个存储位表示节点的颜色，可是 red 或 black，红黑树的查找、插入、删除的时间复杂度最坏为 $O(\lg n)$ 。因为由 n 个节点随机生成的二叉搜索树的高度为 $\lg n$ ，所以二叉搜索树的一般操作的执行时间为 $O(\lg n)$ 。如何保证一颗 n 个节点的红黑树的高度始终为 $\lg n$ ，因为红黑树的 5 个性质：1）

每个节点，要么红的，要么黑的 2）根节点是黑的 3）叶节点都是黑的，指的是 NIL 指针 4）若一个节点是红的，则它的两个儿子都是黑的。5）对于任意节点而言，其到叶节点数尾端 NIL 指针的每条路径都包含相同数目的叶节点。

Hibernate 1+N 问题：1）一对多（<set><list>）：在 1 的这方，通过一条 sql 查到 1 个对象，由于关联的存在，那么又需要将这 n 个对象关联的集合取出。因为集合数量为 n，还要发 n 条 sql，于是本来的一条 sql 变为 n+1 条。2）多对一（many-to-one），在多的这方通过一条 sql 查到了 n 个对象，由于关联的存在，也会将这 n 个对象对应的一方的对象取出，所以本来一条 sql 变为 n+1 条。3）iterator 查询时，一定先去缓存中找（1 条 sql 查集合，只查 id），在没命中时，会再按 id 到库中逐一查找，产生 1+n 条 sql。解决办法：1）lazy=true：懒加载，不立即查询关联对象，只有当需要关联对象（访问其属性，非 id 字段）时才会发生查询动作。2）使用二级缓存：二级缓存的应用不怕 1+n 问题，因为即使第一次查询很慢（未命中），以后查询直接缓存命中也是很快的，刚好利用了 1+n。

Load() VS get() 1）若找不到符合条件的记录，get 返回 null，而 load 抛 ObjectNotFoundException。2）load 返回实体的代理类，get 返回实体类。3）load 可利用二级缓存和内部缓存的现有数据。Get 只用内部缓存，若没有发现对应数据，将跳过二级缓存，直接调用 sql 进行查找。4）当我们使用 session.load 方法来加载一个对象时，此时不会发 sql 语句，当前得到的只是一个代理对象。这个代理对象只保存了实体对象的 id 值。当我们要使用这个对象，得到其他属性时，这时才发 sql 语句，从数据库中去查我们的对象。对于 get：当我们使用 session.get 方法来得到一个对象时，不管我们使不使用这个对象，此时都会发 sql 语句去数据库里查。5）Load 可能抛出 LazyInitializationException，在还没有发出 sql 语句去数据库查实体对象时，当前对象为代理对象（只有 id），但关了 session，再使用该对象就报异常。

许可协议：许可的目的是向使用你产品的人提供一定的权限，常见协议有 BSD、Apache license 、GPL、LGPL。

Xml 的 3 中解析方式：1) sax：不需要读入整个文件就可以对解析出的内容进行处理，是一种逐步解析的方法，程序也可以随时终止解析。适合大规模的解析，事件驱动，不能修改，只能顺序访问。2) 读入整个文档，建立 dom 树，支持应用程序对 xml 数据的内容和结构进行修改，可随机访问。3) digester：满足将 xml 转为 javabean。

Spring 实例中的成员注入：1) 若改变量有 set 方法，用 property。2) 若改变量没有 set 方法，用 constructor-args。

Java 虚拟机栈，每个方法对应一个栈帧，存放局部变量表、操作数栈、方法出口等信息。

堆分为新生代和老年代。Java 堆物理上可不连续，只要逻辑上连续。

堆归整（用了的在一边，没用的在另一边）。

在使用 serial、parnew 等带 compact 过程的收集器时，系统采用的分配方法是指针碰撞。而使用 cms 这种基于 mark-sweep 算法的收集器时，通常采用空闲列表。

《深入理解 java 虚拟机》68 页：1) 对分配内存空间的动作进行同步处理——采用 cas 配上失败重试的方式保证更新操作的原子性。2) 把内存分配的动作按照线程划分在不同的空间上进行，每个线程在堆中预先分配 1 小块内存，称为本地线程分配缓冲（TLAB），哪个线程要分配内存，就在哪个线程的 TLAB 上分配，只有 TLAB 用完，要分配新的 TLAB 时，才需要加锁。

强引用：类似 Object o=new Object(),只要强引用还在，则垃圾收集器永远不会收集被引用的对象。软引用：有但并非必须的对象。Softreference 类实现软引用。弱引用：用 weakreference 类实现。虚引用：用 phantomreference 类实现。

程序执行时并非在 all 地方都能停顿下来开始 gc，只能在到达安全点才可以。

使用 wmap 来得知哪些地方存放着对象引用。

Jps：列出正在运行的虚拟机进程。

Jstat：虚拟机的运行状态信息。类加载、垃圾收集、运行期编译状况。

Jinfo：虚拟机参数。

Jmap：堆转储快照。Heapdump/dump 文件。

Jhat：分析 Heapdump 文件。

Jstack：用于生成虚拟机当时时刻的线程快照。

Jconsole

Visual VM

Memory analysis tool

Support assistant

源文件通过编译变为字节码文件。

任何一个 class 文件都对应着唯一的一个类 or 接口的定义信息。

Class 文件为二进制字节流，以 8 位字节为基础。1) 每个 class 文件的头 4 个字节成为魔数，他的唯一作用是确定这个文件是否为一个能被虚拟机接受的 class 文件。2) 紧接着魔数的 4 个字节是 class 文件的版本号（5、6 字节为次版本号，7、8 字节为主版本号）。3) 紧接着主次版本号的是常量池入口。4) 紧接着两个字节为访问标志。5) 类索引、父类索引、接口索引集合。6) 字段表。7) 方法表。8) 属性表（java 程序方法体中的代码经过 java 编译后，最终变为字节码指令存在 code 属性中）。

Java 虚拟机的操作码只有一个字节。

子类引用父类静态字段，子类不会初始化。通过数据定义引用类，不会触发此类的初始化，比如 `Person[] p=new Person[10];`

若一个类方法体的字节码没有通过验证，那肯定是有问题的；但若一个方法体通过了字节码验证，也不能说明其一定安全。

符号引用的目标不一定已经加载入内存；直接引用的目标一定已经加载入内存。

在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的 code 属性中，因此，一个栈帧需要分配多少内存，不会受到运行期变量数据的影响，而仅仅取决于具体虚拟机的实现。

局部变量表以 slot 为单位，long 和 double 占 2 个 slot，第 0 位索引的 slot 默认为用于传递方法所属对象实例调用，this。

若一个局部变量定义了，但没有赋初始值是不能使用的。

操作数栈中元素的类型必须与字节码指令的序列严格匹配。

每个栈帧都包含一个指向运行时常量池中该栈所属方法的引用，持有这个引用是为了支持方法调用过程中的动态链接。

只要能被 `invokestatic` 和 `invokespecial` 指令调用的方法，都可以在解析阶段中确定唯一的调用版本，符合这个条件的有静态方法，私有方法，实例构造器，父类方法，他们在类加载的时候就会把符号引用，改为该方法的直接引用，他们称为非虚方法（包括 `final` 修饰的方法）。

`Animal a=new Dog();`Animal 是静态类型，dog 是实例类型。虚拟机（准确的说是编译器），在重载时是通过参数的静态类型，而不是实例类型作为判断依据。

所有依赖静态类型来定位方法执行版本的分派动作称为静态分派，静态分派的典型应用是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作，实际上不是由虚拟机执行。

多态，重写，动态分派。

java 语言是一门静态多分派，动态单分派。

Java、c++都是静态类型语言，即在编译期进行类型检查。在运行期进行类型检查的是

动态类型语言。

运行时异常，就是只要代码不运行到这一行，就不会有问题。

JIT 编译，在虚拟机内部，把字节码转换为机器码。

Java 编译器是由 java 语言写的程序。编译的三个过程：1) 解析与填充符号表过程。2) 插入或注解处理器的注解处理过程 3) 分析与字节码生成过程。

热点代码需要即时编译，JIT 编译，判断一段代码是否是热点代码叫热点探测。1) 基于采样的热点探测：采用这种方法的虚拟机会周期性的检查各个线程的栈顶，若发现某个或某些方法经常出现在栈顶，则该方法为热点方法 2) 基于计数器的热点探测，采用这种方法的虚拟机会为每个方法建立计数器，统计方法的执行次数，若执行次数超过一次阈值，则认为是热点方法。

即时编译优化技术 1) 公共子表达式消除：如果一个表达式 E 已经计算过了，且从先前的计算到现在，E 中所有变量的值都没有发生变化，则 E 的这次出现就成了公共子表达式。对于这种表达式，没有必要花时间再对他进行计算，只需要直接用前面计算过的表达式结果替代 E 即可 2) 数组边界检查消除 3) 方法内联 4) 逃逸分析。

不可变类型 String、long、double 等数值包装类型，BigInteger 和 BigDecimal 等大数据类型。

Java 中各种操作共享的数据分为五类：不可变，绝对线程安全，相对线程安全，线程兼容，线程对立。

绑定多个条件：一个 reentrantlock 对象可以同时绑定多个 condition。

处理器的指令：比较并交换，cas 指令。cas 指令需要三个操作数，分别为内存位置（在 java 中，可以简单理解为内存地址，用 v 表示）、旧的预期值 A 和新值 B。cas 指令执行时，当且仅当 V 符合旧的预期值 A 时，处理器用新值 B 更新 V 值，否则他就不更新。但无论是否更新了 V 值，都会返回旧的 V 值。上述过程是一个原子过程。

cas 操作定义在 unsafe 类中，但用户程序不能直接调用 unsafe 类，要通过其他 java api。比如整型原子类 compareandset() 和 Incrementandget() 等方法都使用了 unsafe 类的 cas 操作。

Incrementandget(): 以原子的方式将当前值加一，该方法在一个无限循环中，不断尝试把一个比当前值大一的值赋给自己。若失败了，则表明在 compareandset 操作时已经有了修改，于是再次循环进行下一次操作，直到设置成功为止。

Cas 的逻辑漏洞，若一个变量 V 初次读取的时候是 A 值，并且准备赋值的时候，检查到他仍然为 A 值，那我们就能说它的值没被其他线程改变过吗？若在这段期间他的值曾被改变为 B 值，后来又被改变回 A，则 cas 操作会误认为它从来没有被改变过，这个漏洞被称为 ABA 问题。解决办法：控制变量值的版本。

轻量级锁，在没有多线程竞争的条件下，减少传统的重量级锁使用操作系统互斥量带来的性能消耗（使用 cas 操作）。如果说轻量级锁是在无竞争的情况下，用 cas 操作消除同步使用的互斥量，那么偏向锁就是在无竞争的情况下把整个同步去掉，连 cas 操作都不做了。

用遥控板（句柄）控操纵电视（对象），即使没有电视机，遥控板也可以单独存在，即拥有一个句柄，并不表示必须有一个对象同他联系。

```
{
  Int    x=100;
  {
    Int    x=1;  //不合法，因为编译器认为变量已经被定义
  }
}
```

默认值——Boolean-false; char-null; byte- (byte) 0; short- (short) 0; int-0; long-0L; float-0.0f; double-0.0d。

System.out 返回 PrintStream 对象，println()是该对象中的一个方法。

Javadoc 生成注释文档，只能为 public 和 protected 成员生成注释。

赋值 A=4; 是对的 4=a 是错的，只能为变量赋值。

编译器通常把指数默认为 double，若要定义为 float，则加 f。float a=1.43e-13; 编译失败。float a=1.43e-13f; 编译成功。

Boolean - exp? value 0: value 1; 若 Boolean - exp 为 true，则计算 value 0，而且这个计算结果也就是操作符最终产生的；若 Boolean - exp 为 false，则计算 value 1，而且这个计算结果也就是操作符最终产生的值。

java 不允许我们使用一个数字作为布尔值。

For(1;;) 1 的位置可以有任意数量的同一类型的定义。

构造器没有返回值，与返回空不一样。

只有构造函数才可以调用构造器。

Dog dog=new Dog();这句话的执行过程：1) 加载 dog 的 class 文件。2) 初始 all 静态（包括 static 变量和静态代码块），并且要按照顺序。3) 默认初始化。4) 显示初始化。5) 构造代码块。6) 构造函数。

Collection 有 iterator 方法，所以 list 和 set 有。Map 没有，要遍历 map，要使用 entryset 或 keyset。

Treemap 根据键的自然顺序排序。

```
1)数组转为 list Arrays.asList 方法，
String[] s={"str1","str2"};
ArrayList<String> al=Arrays. Aslist(s);
2)list 转为数组 list 的 toArray 方法。
```

可变类型参数 (Object...args) 仍是 object 的数组，可用 foreach 遍历。

Package 语句必须是文件中的第一行（除了注释）。

即使一个类只有包访问权限，其 public main 仍然可以访问。

Has-a 组合；is-a 继承。

既是 **static**，又是 **final** 是编译器常量。

Final 初始化只有两个地方：定义处和构造函数。

覆盖，只有在某方法是基类的接口的一部分时才会出现，即必须能将一个对象向上转型为它的基本类型，并调用相同的方法。若一个方法为 **private**，他就不是接口的一部分，他仅是一些隐藏于类中的程序代码，只不过具有相同的名称而已。但是若在子类中以相同的名称生成一个 **public**、**protected** 或包访问权限方法的话，此时并没有覆盖该方法，只是生成了一个新的方法。由于 **private** 方法无法触及，而且能有效隐藏，所以除了把它看成是因为他所归属的类的组织结构的原因而存在外，其他任何事物都不需要考虑到他。

适配器中的代码接受你所拥有的接口，并且产生你需要的接口。

一个类可以实现多个接口，但只能继承一个类，且继承类要写在前面，后面才跟接口，否则编译报错。接口与接口也是继承，接口可以多继承。

策略模式：所有策略都来自同一个接口，一个策略一个类，接口为参数，传入具体策略（具体类对象），利用多态。

实现某个接口时，并不需要实现嵌套在其内部的任何接口，且 **private** 接口不能在定义它的类之外被发现。

`StringBuffer.append("123").append(26).append(true).reverse()`后变为 `eurt62321`。

内部类拥有其外围类的所有元素的访问权，包括 **private**，包括方法和字段。构造内部类对象时，需要一个指向其外围类对象的引用（**static** 内部类除外），这个引用编译器已经帮你做了。非静态内部类，用外部类对象创建内部类对象：`new outer().new Inner();`

接口的所有成员（方法+变量），自动被设为 **public**。

可以在一个方法里或任意作用域来定义内部类。

每个类都会生成一个 **class** 文件，内部类的 **class** 文件命名为：外部类名\$内部类名。若是匿名的，则编译器会简单生成一个数字作为标识符。

你可以将任意数量的对象放置到容器中，并且不需要担心容器应该设置为多大。

使用泛型可以在编译期防止将错误类型的对象放置到容器中。

`Collection.reverseOrder (comparator<T> cmp)` 返回一个比较器，顺序和传入的比较器相反。

高级 **for** 循环可用于数组和任何 **collection** 对象。如果我们创建了任何实现 **iterator** 的类，就可以用高级 **for** 循环。但数组不是一个 **iterator**。

Linkedhashset 和 **linkedhashmap** 保持元素插入顺序。

任何 **collection** 都可以生成 **iterator**，**list** 还可以生成 **listiterator**。

若在一个方法内部抛出了异常，这个方法将在抛出异常的过程中结束。

在编译时被强制检查的异常称为被检查异常。

一个出现在基类方法得当异常说明中的异常，不一定会出现在派类类方法的异常说明里。

`String s1=" abc" + " def" +47+ " mm"` ;编译器使用了 `StringBuilder` 的 `append()`方法，最后调用 `toString()`;

`String` 的 `format` 方法，用于格式化输出。`S.O.P(String.format(“格式”,变量..));`

正则表达式 `\\ \\+ \\d \\`

`X?` 0 次或 1 次

`X+` 1 次或多次

`X*` 0 次或多次

`X{n}` 恰好 n 次

`X{n, }` 至少 n 次

`X{n,m}` 至少 n 次，至多 m 次

`Pattern p=Pattern.compile(正则表达式);`

`Macher m=p.matcher(要匹配的 String);`

组是用括号划分的正则表达式，可用组的编号来引用组，组号为 0 为整个组，组号为 1 表示第一对括号。

`Getclass()`和 `class.forName()`会初始化对象，即 `static` 将被执行；`class` 属性不会初始化。

`Instance of` 时要看他实例类型，而不是引用类型，子类也是父类实例。

可以有泛型方法，但该方法所属类可是泛型类，也可不是泛型类，泛型写在返回类型之前。

方法会把一个，数组变成一个字符串，对于数组中每个元素调用方式，用隔开，整体用括起来。

`Array.toString()`方法会把一个数组变为一个字符串，对于数组中每个元素调用 `toString` 方法，用“，”隔开，整体用“[]”括起来。

`System.arraycopy(object[] src,int l, object[] desc,int j,int len)`: 从 `src` 数组复制到 `desc` 数组，从 `src` 的 `i` 位置复制 `len` 个元素，从 `desc` 的 `j` 位置开始放。基本数据类型就放值，引用数据类型就复制地址，不支持自动拆箱或装箱，两个类型必须一致。

`Arrays.equals()`比较两个数组是否相等，必须元素个数相同，并且对应位置元素相同（用 `equals()`来比），`int` 变为 `integer`。

`Collection` 的 `reverseOrder()`方法返回一个比较器。

进程调度算法：1）先来先服务 2）短作业优先 3）时间片轮转 4）最高优先权-抢占式和 非抢占式。

除了 8 种基本类型以及它们的封装类型，另外还有 String 类，会发生深拷贝，其余都是浅拷贝。

当 hashmap 的 get()方法返回 null 时，既可以表示 hashmap 中没有该键，也可以表示该键对应的值为 null，所以 hashmap 中不能用 get()方法来判断是否存在某个键，而应该用 containskey()。

深度为 k 的完全二叉树，最多有 2^k-1 个节点，第 k 层最多有 2^{k-1} 个节点。

UML 类图，泛华、实现、依赖、聚合、组合这种基础的要会，有时面试官问设计模式的时候，会让你写出这个设计模式的类图，如果你都会，那当然好，否则就自己准备几个拿手的。

0: 48 a: 97 A: 65

String 的 contains 和 indexOf 都可以判断是否有某词。

锁隔离级别实现原理：1) 读未提交，事务在读数据时并不加锁，事务在写数据时加行级共享锁。2) 读已提交，事务对当前被读取的数据加行级共享锁（当读到时才加锁），一旦读完，立即释放该行级共享锁；事务在更新某数据的瞬间，必须先加行级排他锁，直到事务结束才释放。3) 可重复读，事务在读取某数据瞬间（开始读取的瞬间），必须先对其加行级共享锁，直到事务结束才释放；事务在更新某数据瞬间（发生更新的瞬间），必须先对其加行级排他锁，直到事务结束才释放。4) 序列化，事务在读取数据时，必须先对其加表级共享锁，直到事务结束才释放；事务在更新数据时，必须先对其加表级排他锁，直到事务结束才释放。

Class 的 getConstructor()返回指定的公有的构造函数；Class 的 getDeclaredConstructor()返回指定的构造函数，私有也可以得到。

单例模式 1) 序列化，又反序列化之后，不是同一个对象，但是可以加上 readResolve()函数，并在方法体内写上 return instance;这样就是一个对象了。2) 用反射获得私有构造函数，然后用 constructor.newInstance 弄的两个对象也不是同一个对象，为了避免这个漏洞，可以在私有的构造函数中加上 if(null!=instance) return new RuntimeException();

可以被序列化：String、数组、Enum、实现了序列化接口（）从 WriteObject 方法看出来，这四个可以被序列化。

若只是让某个类实现 serializable 接口，而没有其他处理的话，就是使用默认序列化机制，即在序列化对象时，不仅会序列化当前对象本身，还会对该对象引用的其他对象也进行序列化。同样的，这些其他对象引用的另外对象也将被序列化。

若一个成员变量被 transient 修饰后，不可被序列化。要想被序列化，则要在类中加上两个方法：1.WriteObject(ObjectOutputStream out) out.defaultWriteObject(); out.writeInt(age);

和 2.ReadObject(ObjectInputStream in) in.defaultReadObject(); age=in.ReadInt();都是 private 的，通过反射调用。

无状态的对象一定是线程安全的。无状态指：不包含任何域，也不包含任何对其他类中域的引用。我理解的状态就是域。（i++不是原子的）。

发布一个对象：使对象能在当前作用域之外的代码中使用。某个不该发布的对象被发布称为逸出。

如果一个类是由多个独立且线程安全的状态变量组成，并且在所有的操作中都不包含无效状态转换，那么可以将线程安全性委托给底层的状态变量。

同步封装类是由 `Collections.synchronizedXxx` 等工厂方法创建的，这些类实现安全的方式是，将他们的状态封装起来，并且每个公有方法进行同步，使得每次只有一个线程能访问容器的状态。

同步容器类都是线程安全的，但是在某些情况下可能需要额外的客户端加锁来保护复合操作。比如：“若没有，则添加”。

`copyOnWriteArrayList` 和 `copyOnWriteSet` 从一开始大家都在共享同一个内容，当某个人想要修改这个内容时，才会真正的把内容 `copy` 出去，形成一个新的内容后再改。比如：当我们往一个容器添加元素时，不直接往当前容器添加，而是先将容器进行 `copy`，复制出一个新容器，再往新容器里加元素。添加完之后，再将原容器引用指向新容器。好处：对 `copyOnWrite` 容器进行并发读时，不需要加锁，因为当前容器不会增加新元素，读写分离。

`copyOnWriteArrayList` 的 `add` 方法要加锁，否则多线程的时候会 `copy` N 个副本。`copyOnWrite` 适合于读多写少的场景，但他只能保证数据最终一致性，不能保证实时一致性。若你希望写入马上被读到，不要用 `copyOnWrite` 容器。

`Queue` 上的操作不会阻塞，如果队列为空，那么获取元素的操作返回空。`BlockingQueue` 扩展了 `Queue`，增强了可阻塞的插入和获取等。若队列对 `null`，那么获取元素的操作将一直阻塞，直到队列中出现一个可用的元素；若队列已满（对于有界队列来说），那么插入元素的操作将一直阻塞，直到队列中有可用空间。

`ConcurrentHashMap` 允许一边迭代，一边更新，不会出现并发修改异常。也就是说，在用 `iterator` 遍历时，`ConcurrentHashMap` 也可以进行 `remove`、`put` 操作，且遍历的数据会随着 `remove`、`put` 发生变化，（弱一致性）。

`Concurrentmap` 中以下功能：若没有则添加，若相等则移除，若相等则替换，且都为原子性。

`copyOnWrite` 容器不会抛出并发修改异常，并且返回的元素与迭代器创建时的元素完全一致，而不必考虑之后修改操作带来的影响，当迭代操作远大于修改操作时适用。

`Deque` 和 `BlockingDeque` 为双端队列，可以在队列头和尾插入删除，在生产者-消费者中，所有消费者有一个共同的工作队列。而在工作窃取设计中，每个消费者都有自己的双端队列。若一个消费者完成了自己双端队列中的全部工作容，那么他可以从其他消费者双端队列末尾秘密的获取工作。大多数情况下，消费者只访问自己的双端队列，从而极大减少竞争，而且访问别人双端队列时是从末尾开始取，进一步减小竞争。

`CountDownLatch` 是一种灵活的闭锁实现，可以使一个或多个线程等待一组事件发生，

事件全发生后阻塞的才能继续执行。闭锁状态包括一个计数器，该计数器被初始化为一个正数，表示要等待的事件发生。**CountDown** 方法递减计数器，表示有一个事件已经发生，而 **await** 方法等待计数器变为零，表示所有需要等待的事情都发生。当计数器为零时，**await** 上等待的线程可继续执行。

CountDownLatch 构造函数传入的值就是计数器的初始值，且只可被设置一次。

Semaphore：用来控制同时访问某个特定资源的操作数量。用 **acquire** 获取一个许可，若没有则等待。用 **release** 释放一个许可。单个信号量 **Semaphore** 可实现互斥。

CountDownLatch 计数只能用 1 次。一个或多个线程等待其他线程完成后再执行；**cyclicBarrier**：计数可用多次。所有线程互相等待完成。

Executor：基于生产者-消费者模式。

对中断操作（调用 **interrupt**）的正确理解是，他并不会真正的中断一个正在运行的线程，而只是发出中断请求，然后由线程在下一个合适的时候中断自己。比如，**wait**、**sleep**、**join** 等方法，当他们收到中断请求或开始执行时，发现某个已经被设置好的中断状态，则抛出异常 **InterruptedException**。

每个线程都有一个 **boolean** 中断状态，当中断线程时，这个中断状态将被设为 **true**。**interrupt** 方法：中断目标线程。**isInterrupted**：返回目标线程的中断状态。静态的 **interrupted** 方法：清除当前线程的中断状态，并返回它之前的值。大多数可中断的阻塞方法会在入口处检查中断状态。

JVM 关闭：1）正常关闭：当最后一个非守护线程结束或调用了 **System.exit** 或通过其他特定于平台的方式，比如 **ctrl+c**。2）强制关闭：调用 **Runtime.halt** 方法，或在操作系统中直接 **kill**（发送 **single** 信号）掉 JVM 进程。3）异常关闭：运行中遇到 **RuntimeException** 异常等。

在某些情况下，我们需要在 JVM 关闭时做一些扫尾的工作，比如删除临时文件、停止日志服务。为此 JVM 提供了关闭钩子（**shutdown hooks**）来做这些事件。

Runtime 类封装 java 应用运行时的环境，每个 java 应用程序都有一个 **Runtime** 类实例，使用程序能与其运行环境相连。

关闭钩子本质上是一个线程（也称为 **hook** 线程），可以通过 **Runtime** 的 **addShutdownHook**（**Thread hook**）向主 jvm 注册一个关闭钩子。**hook** 线程在 jvm 正常关闭时执行，强制关闭不执行。

对于在 jvm 中注册的多个关闭钩子，他们会并发执行，jvm 并不能保证他们的执行顺序。

线程可以分为两种：普通线程和守护线程。**t.setDaemon(true)** 将 **t** 设为守护线程。

有一项技术可以缓解执行时间较长任务造成的影响，即限定任务等待资源的时间，而不要无限的等待。

newFixedThreadPool 中线程数固定且线程不超时，**newCachedThreadPool** 最大线程数 **Integer.Max_value**，并且超时设为 60 秒。

AbortPolicy 是默认拒绝策略。

http：一个 **http** 的请求包含三个方面，请求方法-uri-协议/版本，请求头，实体；**http**

响应也包含三个方面，协议-状态码-描述，响应头，响应实体。

用 memoryanalyzer 看 OOM。

http 请求报文由请求行请求头部空行和请求数据四部分。请求行：请求方法-url-http 协议版本。请求头：通知服务器有关客户端请求的信息，每行一个键值，键值用“：”分开。

This.interrpted：检查当前线程中断状态，且将中断状态标志清除；**this.isInterrupted**：检查 **this** 的中断状态，且不改变线程的状态标识。在 **Thread** 对象上调用 **isInterrupted** 方法可以检查任何线程的中断状态，但是线程一旦被中断，**isInterrupted** 方法变会返回 **true**，而一旦 **sleep().wait** 方法抛出异常，他将清除中断标志，下一次调用 **isInterrupted** 返回 **false**。

interrpted 是静态方法，不能在特定的线程中使用，只能报告它的线程的中断状态，若当前线程被中断，返回 **true**，但会自动重置中断状态为 **false**，第二次调用时总是返回 **false**。

默认空余堆内存小于 40%时，JVM 就会增大堆到-Xmx 的最大值；空余堆内存大于 70%时，JVM 会减小堆到-Xms 的最小值。

RandomAccessFile：用来访问那些保存数据记录的文件，可用 **seek** 方法访问记录，并进行读写，这些记录的大小不必相同，但其大小和位置必须可知。该类仅限操作文件。

RandomAccessFile 不属于 **InputStream** 和 **OutputStream** 系列，直接继承自 **Object**。

RandomAccessFile 构造函数，两种模式：1) “r”，只读，不会创建文件，会去读一个已存在的文件，若文件不存在则抛异常 2) “w”读写模式，若操作的文件不存在，则自动创建，若存在也不会覆盖。

Java web1) 前端 (html、css、js、bootstrap、jquery)；2) java web 核心 (servlet、tomcat、cookie)；3) redis。

get 请求在 url 中发送，post 请求在 http 消息主体中发送。

DNS 劫持：通过某些手段获得某域名的解析控制权，修改此域名的解析结果，导致对该域名的访问由原 ip 地址转入到修改后的指定 ip。其结果是对特定的网址不能访问或访问的是假网址。

当对网络通讯质量有要求的时候，tcp，QQ 文件；当对网络通讯质量要求不高，udp，qq 消息。

Group by：使用 **Group by** 的两个要素。1) 出现在 **select** 后面的字段，要么是聚集函数中的，要么是 **Group by** 中的；2) 要筛选结果，可以先用 **where**，再用 **Group by** 或先用 **Group by** 再用 **where**。

Session：可以放在服务器内存、文件或数据库中。如何实现 **session** 共享：1) 通过组播的方式进行集群间共享，当一台机器上的 **session** 变更后会将变更的数据以组播的方式分发给集群中所有其他节点。2) 利用 **NFS** 等共享存储来共享 **session**，用一台 **nfs** 服务器或数据库来放 **session**，缺点：**nfs** down 掉。3) 利用 **memcached** 或 **redis** 来共享，所有 web 服务器

都把 session 写入 memcached 或 redis，也都从 memcached 或 redis 来取。优点：网络开销小，几乎没有 io；缺点：受限于 memcached 容量。memcached 本来就是分布式。

一个绳子烧完要一个小时，怎么计出 1 小时 15 分钟？绳子无限多。答案：要三条绳子 1、2、3。1) 1 点一头，2 点两头，同时点。2 烧完用了 30 分钟 2) 2 一烧完，马上点 1 的另一头，则 1 烧完用了 15 分钟。3) 1 一烧完，马上点 3，两头点，3 烧完，用了 30 分钟。所以：30+15+30=1 小时 15 分钟。

最短路径算法，迪杰斯特拉算法。

sql 优化：1)对查询进行优化，要尽量避免全表扫描。在 where 或 order by 的列上加索引。2) 尽量避免在 where 子语句中有 where num is null，这样不用索引，要全表扫描，可用 0 代替 null。3) 避免在 where 中用 <>or!=,因为要全表扫描。4) 尽量避免在 where 中用 or，因为若一个字段有索引，一个没有，则要全表扫描。5) like"%abc%"，全表扫描。6) 避免在 where 子语句中对字段进行函数操作，因为要全表扫描。7) 使用复合索引时，必须用到该索引的第一个字段，否则索引不被使用。

在 hashmap 的构造器中不会初始化 entry 数组，而是在执行 put 操作时，才真正创建 table 数组。Put: 1) 先判断 entry 数组是否为空，为空则创建 entry 数组。2) 判断是否为空。。。4) codcount++; 5) 判断是否要扩容，当发生 hash 冲突并且 size>阈值时，扩容。

快速排序若以左边元素为基准，则先 where(arr[high]>=key&&low<high) high--，若以左边元素为基准，则先 where(arr[low]<=key&&low<high) low++.

桶排序：把数组分组放在一个个的桶中，然后对每个桶里的再进行排序。当 n=m 时，可以实现 O(n)，n 个数字 m 个桶，缺点：1) 空间复杂度高；2) 待排序的元素要在某一个范围。

每定义一个 servlet，都必须实现 servlet 接口。genericservlet 实现了该接口，不限于任何协议，httpServlet 受限于 http 协议。

Filter——1) 目的：调用目标资源前让一段代码执行，是否允许用户调用目标资源，调用目标资源之后让一段代码执行。2) 编写 java 类，实现 filter 接口，配置 filter-mapping，说明对哪些资源进行拦截。3) 每次拦截时，都调用 filter 的 dofilter 方法。4) filter 创建和销毁由 servlet 容器，在 web 应用启动时，创建实例，并调用 init 方法，filter 对象只创建一次，init 只执行一次。Destory，容器卸载 filter 对象前被调用。5) 用户在配置 filter 时，可以使用<init-param>为 filter 配置一些初始化参数，当 web 容器实例化 filter 对象时，调用其 init 方法，会把封装了 filter 初始化参数的 filter config 对象初始化，所以开发人员在写 filter 时，可以用 filter config 对象的方法来获得初始化对象。6) 应用：统一全站字符编码；禁止浏览器缓存；实现 url 级别的权限认证。

jsp 引擎就是 servlet 程序。

Jsp 转译为 servlet 源文件，然后编译为 servlet 的 class 文件。

动态：先编译后包含；静态：先包含后编译。

数据链路层协议：ppp、cdma/cd。

邮件，——1) 当用户需要发送电子邮件时，首先利用客户端的电子邮件应用程序，编辑一封邮件，利用 **smtp** 将邮件送往发送端的邮件服务器；2) 发送端的邮件服务器，接收到用户送来的邮件后，通过 **smtp** 将邮件送到接收端的邮件服务器，接收端的邮件服务器根据地址中的账号将邮件投递到对应的邮箱；3) 利用 **pop3** 或 **imap**，接收端的用户可以在任何时间、地址，利用电子邮件应用程序从自己邮件中读取邮件。

BlockingQueue: 1) **ArrayBlockingQueue** 一个由数组实现的有界阻塞队列，其构造函数必须带一个 **int** 参数，来指明其大小。其所含的对象是由 **FIFO** 顺序排列的。2) **linkedBlockingQueue**: 大小不一定，若构造函数传入 **int**，则 **BlockingQueue** 有大小限制，否则大小为 **Integer.MAX_VALUE**，其所含的对象是由 **FIFO** 顺序。3) **PriorityBlockingQueue**: 类似于 **linkedBlockingQueue**，但不是 **fifo**，而是自然顺序或比较器的顺序。4) **synchronizeQueue** 对其的操作必须是放和取交替完成的，见下图：



生产者消费者的示例代码：

生产者：

Java代码  

```
1. import java.util.concurrent.BlockingQueue;
2.
3. public class Producer implements Runnable {
4.     BlockingQueue<String> queue;
5.
6.     public Producer(BlockingQueue<String> queue) {
7.         this.queue = queue;
8.     }
9.
10.    @Override
11.    public void run() {
12.        try {
13.            String temp = "A Product, 生产线程: "
14.                + Thread.currentThread().getName();
15.            System.out.println("I have made a product:"
16.                + Thread.currentThread().getName());
17.            queue.put(temp); //如果队列是满的话，会阻塞当前线程
18.        } catch (InterruptedException e) {
19.            e.printStackTrace();
20.        }
21.    }
22.
23. }
```

消费者:

Java代码  

```
1. import java.util.concurrent.BlockingQueue;
2.
3. public class Consumer implements Runnable{
4.     BlockingQueue<String> queue;
5.
6.     public Consumer(BlockingQueue<String> queue){
7.         this.queue = queue;
8.     }
9.
10.    @Override
11.    public void run() {
12.        try {
13.            String temp = queue.take();//如果队列为空，会阻塞当前线程
14.            System.out.println(temp);
15.        } catch (InterruptedException e) {
16.            e.printStackTrace();
17.        }
18.    }
19. }
```

测试类:

Java代码  

```
1. import java.util.concurrent.ArrayBlockingQueue;
2. import java.util.concurrent.BlockingQueue;
3. import java.util.concurrent.LinkedBlockingQueue;
4.
5. public class Test3 {
6.
7.     public static void main(String[] args) {
8.         BlockingQueue<String> queue = new LinkedBlockingQueue<String>(2);
9.         // BlockingQueue<String> queue = new LinkedBlockingQueue<String>();
10.        //不设置的话，LinkedBlockingQueue默认大小为Integer.MAX_VALUE
11.
12.        // BlockingQueue<String> queue = new ArrayBlockingQueue<String>(2);
13.
14.        Consumer consumer = new Consumer(queue);
15.        Producer producer = new Producer(queue);
16.        for (int i = 0; i < 5; i++) {
17.            new Thread(producer, "Producer" + (i + 1)).start();
18.
19.            new Thread(consumer, "Consumer" + (i + 1)).start();
20.        }
21.    }
22. }
```

打印结果:

Text代码  

```
1. I have made a product:Producer1
2. I have made a product:Producer2
3. A Product, 生产线程: Producer1
4. A Product, 生产线程: Producer2
5. I have made a product:Producer3
6. A Product, 生产线程: Producer3
7. I have made a product:Producer5
8. I have made a product:Producer4
9. A Product, 生产线程: Producer5
10. A Product, 生产线程: Producer4
```

4、后记

花了半个月终于把全部面经写完了，这意味着我的秋招真的全部结束了。这个面经只是给大家一个参考，我的方法不一定大家都适用，还希望学弟学妹们结合自己实际，好好复习。作为一个没有实习经历、没有项目经验的女生，在今年也能拿几个 offer，虽然和 bat 大佬不能比，但我希望用自己的经历告诉大家，天道酬勤是真的。最后祝大家都能找一个好工作，就酱！