序言

J2SE 基础

1. 九种基本数据类型的大小，以及他们的封装类。
2. Switch 能否用 string 做参数？
3. equals 与==的区别。
4. Object 有哪些公用方法？
5. Java 的四种引用，强弱软虚，用到的场景。
6. Hashcode 的作用。
7. ArrayList、LinkedList、Vector 的区别。
8. String、StringBuffer 与 StringBuilder 的区别。
9. Map、Set、List、Queue、Stack 的特点与用法。
10. HashMap 和 HashTable 的区别。
11. HashMap 和 ConcurrentHashMap 的区别，HashMap 的底层源码。
12. TreeMap、HashMap、LindedHashMap 的区别。
13. Collection 包结构，与 Collections 的区别。
14. try catch finally，try 里有 return，finally 还执行么？
15. Excption 与 Error 包结构。OOM 你遇到过哪些情况，SOF 你遇到过哪些情况。
16. Java 面向对象的三个特征与含义。
17. Override 和 Overload 的含义去区别。
18. Interface 与 abstract 类的区别。
19. Static class 与 non static class 的区别。
20. java 多态的实现原理。
21. 实现多线程的两种方法：Thread 与 Runable。
22. 线程同步的方法：sychronized、lock、reentrantLock 等。
23. 锁的等级：方法锁、对象锁、类锁。
24. 写出生产者消费者模式。
25. ThreadLocal 的设计理念与作用。
26. ThreadPool 用法与优势。
27. Concurrent 包里的其他东西：ArrayBlockingQueue、CountDownLatch 等等。
28. wait()和 sleep()的区别。
29. foreach 与正常 for 循环效率对比。
30. Java IO 与 NIO。
31. 反射的作用于原理。
32. 泛型常用特点，List<String>能否转为 List<Object>。
33. 解析 XML 的几种方式的原理与特点：DOM、SAX、PULL。
34. Java 与 C++对比。
35. Java1.7 与 1.8 新特性。
36. 设计模式：单例、工厂、适配器、责任链、观察者等等。
37. JNI 的使用。

Java 里有很多很杂的东西，有时候需要你阅读源码，大多数可能书里面讲的不是太清楚，需要你在网上寻找答案。

推荐书籍：《java 核心技术卷 I》《Thinking in java》《java 并发编程》《effictive java》《大话设计模式》

JVM

1. 内存模型以及分区，需要详细到每个区放什么。
2. 堆里面的分区：Eden，survival from to，老年代，各自的特点。
3. 对象创建方法，对象的内存分配，对象的访问定位。
4. GC 的两种判定方法：引用计数与引用链。
5. GC 的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？
6. GC 收集器有哪些？CMS 收集器与 G1 收集器的特点。
7. Minor GC 与 Full GC 分别在什么时候发生？
8. 几种常用的内存调试工具：jmap、jstack、jconsole。
9. 类加载的五个过程：加载、验证、准备、解析、初始化。
10. 双亲委派模型：Bootstrap ClassLoader、Extension ClassLoader、ApplicationClassLoader。
11. 分派：静态分派与动态分派。

JVM 过去过来就问了这么些问题，没怎么变，内存模型和 GC 算法这块问得比较多，可以在网上多找几篇博客来看看。

推荐书籍：《深入理解 java 虚拟机》

操作系统

1. 进程和线程的区别。
2. 死锁的必要条件，怎么处理死锁。
3. Window 内存管理方式：段存储，页存储，段页存储。
4. 进程的几种状态。
5. IPC 几种通信方式。
6. 什么是虚拟内存。
7. 虚拟地址、逻辑地址、线性地址、物理地址的区别。

因为是做 android 的这一块问得比较少一点，还有可能上我简历上没有写操作系统的原因。

推荐书籍：《深入理解现代操作系统》

TCP/IP

1. OSI 与 TCP/IP 各层的结构与功能，都有哪些协议。
2. TCP 与 UDP 的区别。
3. TCP 报文结构。
4. TCP 的三次握手与四次挥手过程，各个状态名称与含义，TIMEWAIT 的作用。
5. TCP 拥塞控制。
6. TCP 滑动窗口与回退 N 针协议。
7. Http 的报文结构。
8. Http 的状态码含义。
9. Http request 的几种类型。
10. Http1.1 和 Http1.0 的区别
11. Http 怎么处理长连接。
12. Cookie 与 Session 的作用于原理。
13. 电脑上访问一个网页，整个过程是怎么样的：DNS、HTTP、TCP、OSPF、IP、ARP。
14. Ping 的整个过程。ICMP 报文是什么。
15. C/S 模式下使用 socket 通信，几个关键函数。
16. IP 地址分类。
17. 路由器与交换机区别。

网络其实大体分为两块，一个 TCP 协议，一个 HTTP 协议，只要把这两块以及相关协议搞清楚，一般问题不大。

推荐书籍：《TCP/IP 协议族》

数据结构与算法

1. 链表与数组。
2. 队列和栈，出栈与入栈。
3. 链表的删除、插入、反向。
4. 字符串操作。
5. Hash 表的 hash 函数，冲突解决方法有哪些。
6. 各种排序：冒泡、选择、插入、希尔、归并、快排、堆排、桶排、基数的原理、平均时间复杂度、最坏时间复杂度、空间复杂度、是否稳定。
7. 快排的 partition 函数与归并的 Merge 函数。
8. 对冒泡与快排的改进。
9. 二分查找，与变种二分查找。
10. 二叉树、B+树、AVL 树、红黑树、哈夫曼树。
11. 二叉树的前中后续遍历：递归与非递归写法，层序遍历算法。
12. 图的 BFS 与 DFS 算法，最小生成树 prim 算法与最短路径 Dijkstra 算法。
13. KMP 算法。
14. 排列组合问题。
15. 动态规划、贪心算法、分治算法。（一般不会问到）
16. 大数据处理：类似 10 亿条数据找出最大的 1000 个数.....等等

算法的话其实是个重点，因为最后都是要你写代码，所以算法还是需要花不少时间准备，这里有太多算法题，写不全，我的建议是没事多在 OJ 上刷刷题（牛客网、leetcode 等），剑指 offer 上的算法要能理解并自己写出来，编程之美也推荐看一看。

推荐书籍：《大话数据结构》《剑指 offer》《编程之美》

J2SE 基础

1.九种基本数据类型的大小，以及他们的封装类。

java 提供了一组基本数据类型，包括
boolean, byte, char, short, int, long, float, double, void.
同时，java 也提供了这些类型的封装类，分别为
Boolean, Byte, Character, Short, Integer, Long, Float, Double, Void

类型	字节	表示范围	包装类
byte(字节型)	1	-128~127	Byte
short(短整型)	2	-32768～32767	Short
int(整型)	4	-2147483648～2147483647	Integer
long(长整型)	8	-9223372036854775808 ~ 9223372036854775807	Long
float(浮点型)	4	-3.4E38～3.4E38	Float
double(双精度型)	8	-1.7E308～1.7E308	Double
char(字符型)	2	从字符型对应的整型数来划分，其表示范围是 0～65535	Charater
booealn(布尔型)	1	true 或 false	Boolean

2.Switch 能否用 string 做参数？

在 Java 7 之前，switch 只能支持 byte、short、char、int 或者其对应的封装类以及 Enum 类型。在 Java 7 中，String 支持被加上了。

3.equals 与==的区别。

“==”比较的是值【变量(栈)内存中存放的对象的(堆)内存地址】
equal 用于比较两个对象的值是否相同【不是比地址】
【特别注意】Object 类中的 equals 方法和“==”是一样的，没有区别，而 String 类，Integer 类等等一些类，是重写了 equals 方法，

才使得 equals 和“==不同”，所以，当自己创建类时，自动继承了 Object 的 equals 方法，要想实现不同的等于比较，必须重写 equals 方法。
"=="比"equal"运行速度快,因为"=="只是比较引用.

4. Object 有哪些公用方法？

Object 是所有类的父类，任何类都默认继承 Object。Object 类到底实现了哪些方法？
(1)clone 方法
保护方法，实现对象的浅复制，只有实现了 Cloneable 接口才可以调用该方法，否则抛出 CloneNotSupportedException 异常。
主要是 JAVA 里除了 8 种基本类型传参数是值传递，其他的类对象传参数都是引用传递，我们有时候不希望在方法里讲参数改变，这是就需要在类中复写 clone 方法。
(2)getClass 方法
final 方法，获得运行时类型。
(3)toString 方法
该方法用得比较多，一般子类都有覆盖。
(4)finalize 方法
该方法用于释放资源。因为无法确定该方法什么时候被调用，很少使用。
(5)equals 方法
该方法是非常重要的一个方法。一般 equals 和==是不一样的，但是在 Object 中两者是一样的。子类一般都要重写这个方法。
(6)hashCode 方法
该方法用于哈希查找，可以减少在查找中使用 equals 的次数，重写了 equals 方法一般都要重写 hashCode 方法。这个方法在一些具有哈希功能的 Collection 中用到。
一般必须满足 obj1.equals(obj2)==true。可以推出 obj1.hash- Code()==obj2.hashCode()，但是 hashCode 相等不一定就满足 equals。不过为了提高效率，应该尽量使上面两个条件接近等价。
如果不重写 hashcode(),在 HashSet 中添加两个 equals 的对象，会将两个对象都加入进去。
(7)wait 方法
wait 方法就是使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。wait()方法一直等待，直到获得锁或者被中断。wait(long timeout)设定一个超时间隔，如果在规定时间内没有获得锁就返回。
调用该方法后当前线程进入睡眠状态，直到以下事件发生。
（7.1）其他线程调用了该对象的 notify 方法。
（7.2）其他线程调用了该对象的 notifyAll 方法。
（7.3）其他线程调用了 interrupt 中断该线程。
（7.4）时间间隔到了。
此时该线程就可以被调度了，如果是被中断的话就抛出一个 InterruptedException 异常。
(8)notify 方法
该方法唤醒在该对象上等待的某个线程。
(9)notifyAll 方法
该方法唤醒在该对象上等待的所有线程。

5. Java 的四种引用，强弱软虚，用到的场景。

(1)强引用
以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。
(2)软引用（SoftReference）
如果一个对象只具有软引用，那就类似于可有可无的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。
软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。
(3)弱引用（WeakReference）
如果一个对象只具有弱引用,那就类似于可有可物的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。
弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。
(4)虚引用（PhantomReference）
"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。
虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃 圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是 否已经加入了虚引用，来了解

6. Hashcode 的作用。

Hash 是散列的意思，就是把任意长度的输入，通过散列算法变换成固定长度的输出，该输出就是散列值。关于散列值，有以下几个关键结论：
1、如果散列表中存在和散列原始输入 K 相等的记录，那么 K 必定在 f(K)的存储位置上
2、不同关键字经过散列算法变换后可能得到同一个散列地址，这种现象称为碰撞
3、如果两个 Hash 值不同（前提是同一 Hash 算法），那么这两个 Hash 值对应的原始输入必定不同
HashCode
然后讲下什么是 HashCode，总结几个关键点：
1、HashCode 的存在主要是为了查找的快捷性，HashCode 是用来在散列存储结构中确定对象的存储地址的

2、如果两个对象 equals 相等，那么这两个对象的 HashCode 一定也相同
3、如果对象的 equals 方法被重写，那么对象的 HashCode 方法也尽量重写
4、如果两个对象的 HashCode 相同，不代表两个对象就相同，只能说明这两个对象在散列存储结构中，存放于同一个位置
HashCode 有什么用
回到最关键的问题，HashCode 有什么用？不妨举个例子：
1、假设内存中有 0 1 2 3 4 5 6 7 8 这 8 个位置，如果我有个字段叫做 ID，那么我要把这个字段存放在以上 8 个位置之一，如果不用 HashCode 而任意存放，那么当查找时就需要到 8 个位置中去挨个查找
2、使用 HashCode 则效率会快很多，把 ID 的 HashCode%8，然后把 ID 存放在取得余数的那个位置,然后每次查找该类的时候都可以通过 ID 的 HashCode%8 求余数直接找到存放的位置了
3、如果 ID 的 HashCode%8 算出来的位置上本身已经有数据了怎么办？这就取决于算法的实现了，比如 ThreadLocal 中的做法就是从算出来的位置向后查找第 一个为空的位置，放置数据；HashMap 的做法就是通过链式结构连起来。反正，只要保证放的时候和取的时候的算法一致就行了。
4、如果 ID 的 HashCode%8 相等怎么办（这种对应的是第三点说的链式结构的场景）？这时候就需要定义 equals 了。先通过 HashCode%8 来判断类在哪一 个位置，再通过 equals 来在这个位置上寻找需要的类。对比两个类的时候也差不多，先通过 HashCode 比较，假如 HashCode 相等再判断 equals。如果两个类的 HashCode 都不相同，那么这两个类必定是不同的。
举个实际的例子 Set。我们知道 Set 里面的元素是不可以重复的，那么如何做到？Set 是根据 equals()方法来判断两个元素是否相等的。比方 说 Set 里面已经有 1000 个元素了，那么第 1001 个元素进来的时候，最多可能调用 1000 次 equals 方法，如果 equals 方法写得复杂,对比的 东西特别多,那么效率会大大降低。使用 HashCode 就不一样了,比方说 HashSet，底层是基于 HashMap 实现的,先通过 HashCode 取一 个模,这样一下子就固定到某个位置了，如果这个位置上没有元素，那么就可以肯定 HashSet 中必定没有和新添加的元素 equals 的元素，就可以直接存 放了，都不需要比较；如果这个位置上有元素了，逐一比较，比较的时候先比较 HashCode，HashCode 都不同接下去都不用比了，肯定不一 样，HashCode 相等，再 equals 比较，没有相同的元素就存，有相同的元素就不存。如果原来的 Set 里面有相同的元素，只要 HashCode 的生 成方式定义得好（不重复），不管 Set 里面原来有多少元素，只需要执行一次的 equals 就可以了。这样一来，实际调用 equals 方法的次数大大降低， 提高了效率。

7. ArrayList、LinkedList、Vector 的区别。

ArrayList,Vector 底层是由数组实现，LinkedList 底层是由双线链表实现，从底层的实现可以得出它们的性能问题，ArrayList,Vector 插入速度相对较慢，查询速度相对较快，而 LinkedList 插入速度较快,而查询速度较慢。再者由于 Vevtor 使用了线程安全锁，所以 ArrayList 的运行效率高于 Vector。

8. String、StringBuffer 与 StringBuilder 的区别。

String 类型和 StringBuffer 的主要性能区别：String 是不可变的对象，因此在每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后， JVM 的 GC 就会开始工作，性能就会降低。

使用 StringBuffer 类时，每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。所以多数情况下推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。

StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的

StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的

StringBuilder 与 StringBuffer 有公共父类 AbstractStringBuilder(抽象类)。

9. Map、Set、List、Queue、Stack 的特点与用法。

```
Collection          接口的接口    对象的集合
├ List              子接口        按进入先后有序保存    可重复
│ ├ LinkedList      接口实现类    链表    插入删除    没有同步    线程不安全
│ ├ ArrayList       接口实现类    数组    随机访问    没有同步    线程不安全
│ └ Vector          接口实现类    数组                同步        线程安全
│     └ Stack
└ Set                子接口        仅接收一次，并做内部排序
  ├ HashSet
  │     └ LinkedHashSet
  └ TreeSet
```

对于 List ，关心的是顺序， 它保证维护元素特定的顺序（允许有相同元素），使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素。

对于 Set ，只关心某元素是否属于 Set （不 允许有相同元素 ），而不关心它的顺序。

```
Map                 接口        键值对的集合
├ Hashtable         接口实现类                同步            线程安全
├ HashMap           接口实现类                没有同步        线程不安全
│ ├ LinkedHashMap
│ └ WeakHashMap
├ TreeMap
└ IdentifyHashMap
```

对于 Map ，最大的特点是键值映射，且为一一映射，键不能重复，值可以，所以是用键来索引值。 方法 put(Object key, Object value) 添加一个“值”（ 想要得东西 ） 和与“值”相关联的“键” (key) (使用它来查找) 。方法 get(Object key) 返回与给定“键”相关联的“值”。

Map 同样对每个元素保存一份，但这是基于 ″ 键 ″ 的， Map 也有内置的排序，因而不关

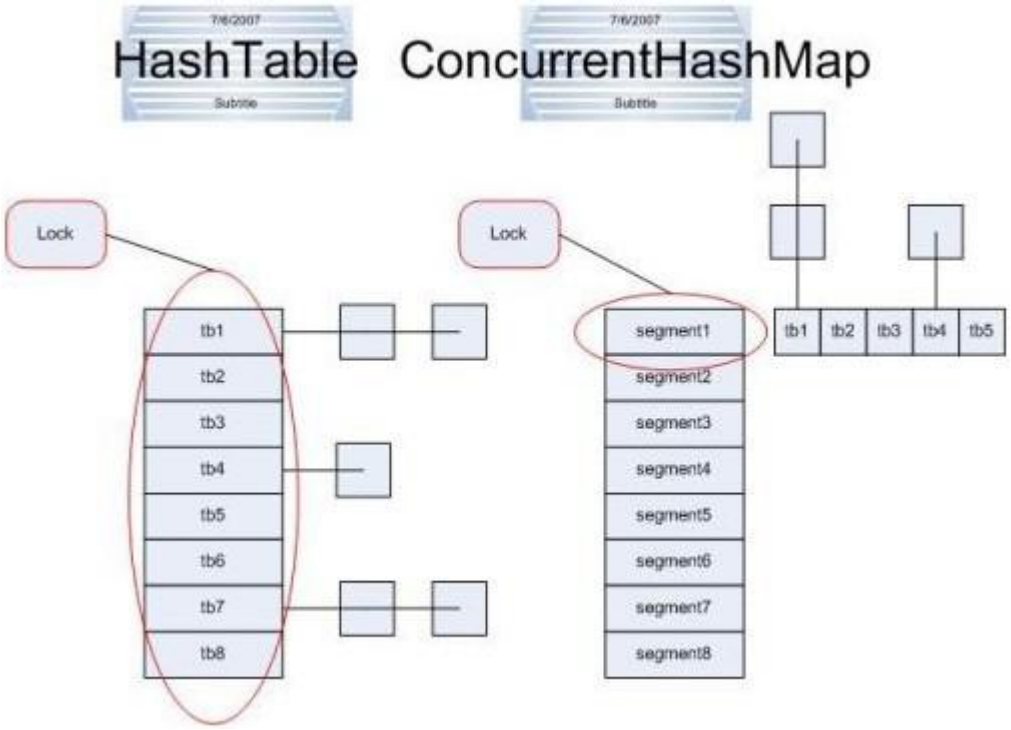
心元素添加的顺序。如果添加元素的顺序对你很重要，应该使用 LinkedHashSet 或者 LinkedHashMap.
对于效率， Map 由于采用了哈希散列，查找元素时明显比 ArrayList 快。
更为精炼的总结：
Collection 是对象集合， Collection 有两个子接口 List 和 Set
List 可以通过下标 (1,2..) 来取得值，值可以重复
而 Set 只能通过游标来取值，并且值是不能重复的
ArrayList ， Vector ， LinkedList 是 List 的实现类
ArrayList 是线程不安全的， Vector 是线程安全的，这两个类底层都是由数组实现的
LinkedList 是线程不安全的，底层是由链表实现的
Map 是键值对集合
HashTable 和 HashMap 是 Map 的实现类
HashTable 是线程安全的，不能存储 null 值
HashMap 不是线程安全的，可以存储 null 值
Stack 类：继承自 Vector，实现一个后进先出的栈。提供了几个基本方法，push、pop、peak、empty、search 等。
Queue 接口：提供了几个基本方法，offer、poll、peek 等。已知实现类有 LinkedList、PriorityQueue 等。

10. HashMap 和 HashTable 的区别。

HashMap 和 Hashtable 都实现了 Map 接口,但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。
HashMap 几乎可以等价于 Hashtable，除了 HashMap 是非 synchronized 的，并可以接受 null(HashMap 可以接受为 null 的键值(key)和值(value)，而 Hashtable 则不行)。
HashMap 是非 synchronized，而 Hashtable 是 synchronized，这意味着 Hashtable 是线程安全的，多个线程可以共享一个 Hashtable；而如果没有正确的同步的话，多个线程是不能共享 HashMap 的。Java 5 提供了 ConcurrentHashMap，它是 HashTable 的替代，比 HashTable 的扩展性更好。
另一个区别是 HashMap 的迭代器(Iterator)是 fail-fast 迭代器，而 Hashtable 的 enumerator 迭代器不是 fail-fast 的。所以当有其它线程改变了 HashMap 的结构（增加或者移除元素)，将会抛出 ConcurrentModificationException，但迭代器本身的 remove()方法移除元素则不会抛出 ConcurrentModificationException 异常。但这并不是一个一定发生的行为，要看 JVM。这条同样也是 Enumeration 和 Iterator 的区别。
由于 Hashtable 是线程安全的也是 synchronized，所以在单线程环境下它比 HashMap 要慢。如果你不需要同步，只需要单一线程，那么使用 HashMap 性能要好过 Hashtable。
HashMap 不能保证随着时间的推移 Map 中的元素次序是不变的。

11. HashMap 和 ConcurrentHashMap 的区别，HashMap 的底层源码。

ConcurrentHashMap 融合了 hashtable 和 hashmap 二者的优势。
hashtable 是做了同步的，hashmap 未考虑同步。所以 hashmap 在单线程情况下效率较高。
hashtable 在的多线程情况下，同步操作能保证程序执行的正确性。
但是 hashtable 每次同步执行的时候都要锁住整个结构。看下图：



图左侧清晰的标注出来，lock 每次都要锁住整个结构。
oncurrentHashMap 正是为了解决这个问题而诞生的。
ConcurrentHashMap 锁的方式是稍微细粒度的(分段锁)。 ConcurrentHashMap 将 hash 表分为 16 个桶（默认值），诸如 get,put,remove 等常用操作只锁当前需要用到的桶。
从 ConcurrentHashMap 代码中可以看出，它引入了一个“分段锁”的概念，具体可以理解为把一个大的 Map 拆分成 N 个小的 HashTable，根据 key.hashCode()来决定把 key 放到哪个 HashTable 中。
在 ConcurrentHashMap 中，就是把 Map 分成了 N 个 Segment，put 和 get 的时候，都是现根据 key.hashCode()算出放到哪个 Segment 中：
试想，原来 只能一个线程进入，现在却能同时 16 个写线程进入（写线程才需要锁定，而读线程几乎不受限制，之后会提到），并发性的提升是显而易见的。
更令人惊讶的是 ConcurrentHashMap 的读取并发，因为在读取的大多数时候都没有用到锁定，所以读取操作几乎是完全的并发操作，而写操作锁定的粒度又非常细，比起之前又更加快速（这一点在桶更多时表现得更明显些）。只有在求 size 等操作时才需要锁定整个表。
而在迭代时，ConcurrentHashMap 使用了不同于传统集合的快速失败迭代器的另一种迭代方式，我们称为弱一致迭代器。在这种迭代方式中，当 iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在改变时 new 新的数据从而

不影响原有的数 据，iterator 完成后再将头指针替换为新的数据，这样 iterator 线程可以使用原来老的数据，而写线程也可以并发的完成改变，更重要的，这保证了多个线程并发执行的连续性和扩展性，是性能提升的关键。

12. TreeMap、HashMap、LindedHashMap 的区别。

Map 主要用于存储健值对，根据键得到值，因此不允许键重复(重复了覆盖了),但允许值重复。

Hashmap 是一个最常用的 Map,它根据键的 HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度，遍历时，取得数据的顺序是完全随机的。HashMap 最多只允许一条记录的键为 Null;允许多条记录的值为 Null;HashMap 不支持线程的同步，即任一时刻可以有多个线程同时写 HashMap;可能会导致数据的不一致。如果需要同步，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有同步的能力，或者使用 ConcurrentHashMap。

Hashtable 与 HashMap 类似,它继承自 Dictionary 类，不同的是:它不允许记录的键或者值为空;它支持线程的同步，即任一时刻只有一个线程能写 Hashtable,因此也导致了 Hashtable 在写入时会比较慢。

LinkedHashMap 保存了记录的插入顺序，在用 Iterator 遍历 LinkedHashMap 时，先得到的记录肯定是先插入的.也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比 HashMap 慢，不过有种情况例外，当 HashMap 容量很大，实际数据较少时，遍历起来可能会比 LinkedHashMap 慢，因为 LinkedHashMap 的遍历速度只和实际数据有关，和容量无关，而 HashMap 的遍历速度和他的容量有关。

TreeMap 实现 SortMap 接口，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 TreeMap 时，得到的记录是排过序的。

一般情况下,我们用的最多的是 HashMap,HashMap 里面存入的键值对在取出的时候是随机的,它根据键的 HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度。在 Map 中插入、删除和定位元素，HashMap 是最好的选择。

TreeMap 取出来的是排序后的键值对。但如果您要按自然顺序或自定义顺序遍历键，那么 TreeMap 会更好。

LinkedHashMap 是 HashMap 的一个子类，如果需要输出的顺序和输入的相同,那么用 LinkedHashMap 可以实现,它还可以按读取顺序来排列，像连接池中可以应用。

13. Collection 包结构，与 Collections 的区别。

(1)java.util.Collection 是一个集合接口。它提供了对集合对象进行基本操作的通用接口方法。Collection 接口在 Java 类库中有很多具体的实现。Collection 接口的意义是为各种具体的集合提供了最大化的统一操作方式。

Collection
├List
│ ├LinkedList
│ ├ArrayList

| └Vector
| └Stack
└Set
(2)java.util.Collections 是一个包装类。它包含有各种有关集合操作的静态多态方法，用于实现对各种集合的搜索、排序、线程安全化等操作。此类不能实例化，就像一个工具类，服务于 Java 的 Collection 框架。

14. try catch finally，try 里有 return，finally 还执行么？

(1)、不管有木有出现异常，finally 块中代码都会执行；
(2)、当 try 和 catch 中有 return 时，finally 仍然会执行；
(3)、在 try 语句中，try 要把返回的结果放置到不同的局部变量当中，执行 finaly 之后，从中取出返回结果，因此，即使 finaly 中对变量进行了改变，但是不会影响返回结果，因为使用栈保存返回值，即使在 finaly 当中进行数值操作，但是影响不到之前保存下来的具体的值，所以 return 影响不了基本类型的值，这里使用的栈保存返回值。而如果修改 list，map，自定义类等引用类型时，在进入了 finaly 之前保存了引用的地址， 所以在 finaly 中引用地址指向的内容改变了，影响了返回值。
总结：
1.影响返回结果的前提是在 非 finally 语句块中有 return 且非基本类型
2.不影响返回结果 的前提是 非 finally 块中有 return 且为基本类型
究其本质 基本类型在栈中存储,返回的是真实的值,而引用类型返回的是其浅拷贝堆地址.所以才会改变。
return 的若是对象，则先把对象的副本保存起来，也就是说保存的是指向对象的地址。若对原来的对象进行修改。对象的地址仍然不变，return 的副本仍然是指向这个对象，所用 finally 中对对象的修改仍然有作用。而基本数据类型保存的是原原本本的数据，return 保存副本后，在 finally 中修改都是修改原来的数据。副本中的数据还是不变，所以 finally 中修改对 return 无影响。

(4)、finally 中最好不要包含 return，否则程序会提前退出，返回值不是 try 或 catch 中保存的返回值。

15. Excption 与 Error 包结构。OOM 你遇到过哪些情况，SOF 你遇到过哪些情况。

(1). Throwable
Throwable 是 Java 语言中所有错误或异常的超类。
Throwable 包含两个子类: Error 和 Exception 。它们通常用于指示发生了异常情况。
Throwable 包含了其线程创建时线程执行堆栈的快照，它提供了 printStackTrace()等接口用于获取堆栈跟踪数据等信息。

```
Throwable
        -Error
        -Exception
                -ClassNotFoundException
                -CloneNotSupportedException
                -IOException
                        -EOFException
                        -FileNotFoundException
                        -MalformedURLException
                        -UnknownHostException
                -RuntimeException
                         -ArithmeticException
                         -ClassCastException
                         -IllegalArgumentException
                         -IllegalStateException
                         -IndexOutOfBoundsException
                         -NoSuchElementException
                         -NullPointerException
```

(2). Exception

Exception 及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。

(3). RuntimeException

RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。

编译器不会检查 RuntimeException 异常。 例如，除数为零时，抛出 ArithmeticException 异常。RuntimeException 是 ArithmeticException 的超类。当代码发生除数为零的情况时，倘若既"没有通过 throws 声明抛出 ArithmeticException 异常"，也"没有通过 try…catch…处理该异常"，也能通过编译。这就是我们所说的"编译器不会检查 RuntimeException 异常"！

如果代码会产生 RuntimeException 异常，则需要通过修改代码进行避免。 例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

(4). Error

和 Exception 一样， Error 也是 Throwable 的子类。 它用于指示合理的应用程序不应该试图捕获的严重问题，大多数这样的错误都是异常条件。

和 RuntimeException 一样， 编译器也不会检查 Error。

Java 将可抛出(Throwable)的结构分为三种类型： 被检查的异常(Checked Exception)，运行时异常(RuntimeException)和错误(Error)。

(01) 运行时异常

定义 ： RuntimeException 及其子类都被称为运行时异常。

特点 ： Java 编译器不会检查它。 也就是说，当程序中可能出现这类异常时，倘若既"没有通过 throws 声明抛出它"，也"没有用 try-catch 语句捕获它"，还是会编译通过。例如，除数为零时产生的 ArithmeticException 异常，数组越界时产生的 IndexOutOfBoundsException 异常，fail-fail 机制产生的 ConcurrentModificationException 异常等，都属于运行时异常。

虽然 Java 编译器不会检查运行时异常，但是我们也可以通过 throws 进行声明抛出，也可以通过 try-catch 对它进行捕获处理。

如果产生运行时异常，则需要通过修改代码来进行避免。 例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

(02) 被检查的异常

定义 ： Exception 类本身，以及 Exception 的子类中除了”运行时异常”之外的其它子类都属于被检查异常。
特点 ： Java 编译器会检查它。 此类异常，要么通过 throws 进行声明抛出，要么通过 try-catch 进行捕获处理，否则不能通过编译。例如，CloneNotSupportedException 就属于被检查异常。当通过 clone()接口去克隆一个对象，而该对象对应的类没有实现 Cloneable 接口，就会抛出 CloneNotSupportedException 异常。
被检查异常通常都是可以恢复的。
(03) 错误
定义 ： Error 类及其子类。
特点 ： 和运行时异常一样，编译器也不会对错误进行检查。
当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。例如，VirtualMachineError 就属于错误。
按照 Java 惯例，我们是不应该是实现任何新的 Error 子类的！

学习《深入理解 Java 虚拟机 JVM 高级特性域最佳实践》,学习到了 JVM 中常见的 OutOfMemory 和 StackOverFlow 产生的机理，感觉非常有用。
1.平时代码运行时遇到这两种错误后就可以根据具体情况去适时地调整 JVM 参数来处理问题
2.平时写代码的时候也会多加注意，不要让代码产生这两种异常
下面就记录下，当作学习笔记。

首先必须了解 JVM 运行时数据区域
方法区
用于存储已被 JVM 加载的类信息，常量，静态变量，即时编译器编译后的代码，线程共享。
运行时常量池
方法区一部分。存放编译期生成的各种字面量和符号引用。
虚拟机栈
内部创建栈帧，来存放局部变量表，操作数栈，动态链接，方法出口等，线程私有。
本地方法栈（HotSpot 不区分虚拟机栈和本地方法栈）
类似虚拟机栈，但是只为 Native 方法服务。
堆
存放实例对象和数组，线程共享。
程序计数器
存放当前线程执行的字节码的行号。

1.产生堆溢出
堆是存放实例对象和数组的地方，当对象多过设置的堆大小，同时避免 GC 回收即可。最大内存块 Xmx 和最小内存块 Xms 一样，堆就不可扩展了。将 new 出的对象放到 List 中可防止 GC 回收。
[java] view plain copy 在 CODE 上查看代码片派生到我的代码片

```
/*
 * VM args: -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
 * Xms equals Xmx lead to head value can't extend
 */
```

```
import java.util.ArrayList;
import java.util.List;
public class HeapOOM {
    static class OOMObject {

    }

    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<OOMObject>();
        while (true) {
            list.add(new OOMObject());
        }
    }
}
```

2.产生虚拟机栈或本地方法栈 StackOverFlow
当请求的栈深度超过 JVM 允许最大深度即可，用 Xss 设置
[java] view plain copy 在 CODE 上查看代码片派生到我的代码片

```
/*
 *VM args: -Xss128K
 */
public class JavaVMStackSOF {
    private int stackLength = 1;
    public void stackLeak() {
        stackLength++;
        stackLeak();
    }
    public static void main(String[] args) throws Throwable {
        JavaVMStackSOF oom = new JavaVMStackSOF();
        try {
            oom.stackLeak();
        } catch(Throwable e) {
            System.out.println("stack length:" + oom.stackLength);
            throw e;
        }
    }
}
```

3.产生虚拟机栈或方法栈 OutOfMemory
不断创建线程
[java] view plain copy 在 CODE 上查看代码片派生到我的代码片

```
/*
 * VM args: -Xss2M
 */
```

```
public class JavaVMStackOOM {
    private void dontStop() {
        while (true) {

        }
    }

    public void stackLeakByThread() {
        while (true) {
            Thread thread = new Thread(new Runnable() {
                @Override
                public void run() {
                    dontStop();
                }
            });
            thread.start();
        }
    }

    public static void main(String[] args) {
        JavaVMStackOOM oom = new JavaVMStackOOM();
        oom.stackLeakByThread();
    }
}
```

4.运行时常量池异常

使用 String.interm()填充常量池。intern 的左右是如果该常量不再常量池中，则添加到常量池，否则返回该常量引用。常量池是方法区一部分，运行时可限制方法区 PermSize 和最大方法区 MaxPermSize 大小

[java] view plain copy 在 CODE 上查看代码片派生到我的代码片

```
/*
 * VM args: -XX:PermSize=10M -XX:MaxPermSize=10M
 */

import java.util.List;
import java.util.ArrayList;

public class RuntimeConstantPoolOOM {
    public static void main(String[] args) {
        //keep reference,avoid GC collect
        List<String> list = new ArrayList<String>();
        //10M PermSize in integer range enough to lead to OOM
        int i = 0;
        while (true) {
```

```
            list.add(String.valueOf(i++).intern());
        }
    }
}
```

5.方法区溢出
通过 CGLib 将大量信息放到方法区
[java] view plain copy 在 CODE 上查看代码片派生到我的代码片

```
/*
 * VM args: -XX:PermSize=10M -XX:MaxPermSize=10M
 */
import java.lang.reflect.Method;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class JavaMethodAreaOOM {
    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create();
        }
    }

    static class OOMObject() {

    }
}
```

16. Java 面向对象的三个特征与含义。

三大特征是：封装、继承和多态。
封装是指将某事物的属性和行为包装到对象中，这个对象只对外公布需要公开的属性和行为，而这个公布也是可以有选择性的公布给其它对象。在 Java 中能使用 private、protected、public 三种修饰符或不用(即默认 defalut)对外部对象访问该对象的属性和行为进行限制。
继承是子对象可以继承父对象的属性和行为，亦即父对象拥有的属性和行为，其子对象也就

拥有了这些属性和行为。这非常类似大自然中的物种遗传。
多态不是很好解释：更倾向于使用 java 中的固定用法，即 overriding（覆盖）和 overload（过载）。多态则是体现在 overriding（覆盖）上，而 overload（过载）则不属于面向对象中多态的范畴，因为 overload（过载）概念在非面向对象中也存在。overriding（覆盖）是面向对象中的多态，因为 overriding（覆盖）是与继承紧密联系，是面向对象所特有的。多态是指父对象中的同一个行为能在其多个子对象中有不同的表现。也就是说子对象可以使用重写父对象中的行为，使其拥有不同于父对象和其它子对象的表现，这就是 overriding（覆盖）。

17. Override 和 Overload 的含义和区别。

(1). Override 特点
(01)、覆盖的方法的标志必须要和被覆盖的方法的标志完全匹配，才能达到覆盖的效果；
(02)、覆盖的方法的返回值必须和被覆盖的方法的返回一致；
(03)、覆盖的方法所抛出的异常必须和被覆盖方法的所抛出的异常一致，或者是其子类；
(04)、方法被定义为 final 不能被重写。
(05)、对于继承来说，如果某一方法在父类中是访问权限是 private，那么就不能在子类对其进行重写覆盖，如果定义的话，也只是定义了一个新方法，而不会达到重写覆盖的效果。（通常存在于父类和子类之间。）
(2).Overload 特点
(01)、在使用重载时只能通过不同的参数样式。例如，不同的参数类型，不同的参数个数，不同的参数顺序（当然，同一方法内的几个参数类型必须不一样，例如可以是 fun(int, float)， 但是不能为 fun(int, int)）；
(02)、不能通过访问权限、返回类型、抛出的异常进行重载；
(03)、方法的异常类型和数目不会对重载造成影响；
(04)、重载事件通常发生在同一个类中，不同方法之间的现象。
(05)、存在于同一类中，但是只有虚方法和抽象方法才能被覆写。
其具体实现机制：
overload 是重载，重载是一种参数多态机制，即代码通过参数的类型或个数不同而实现的多态机制。 是一种静态的绑定机制（在编译时已经知道具体执行的是哪个代码段）。
override 是覆盖。覆盖是一种动态绑定的多态机制。即在父类和子类中同名元素（如成员函数）有不同 的实现代码。执行的是哪个代码是根据运行时实际情况而定的。

18. Interface 与 abstract 类的区别。

(01).abstract class 在 Java 中表示的是一种继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个 interface。
(02).在 abstract class 中可以有自己的数据成员，也可以有非 abstarct 的方法，而在 interface 中，只能够有静态的不能被修改的数据成员（也就是必须是 static final 的，不过在 interface 中一般不定义数据成员），所有的方法都是 public abstract 的。
(03).抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。

接口中定义的变量默认是 public static final 型，且必须给其赋初值，所以实现类中不能重新定义，也不能改变其值。
(04).abstract class 和 interface 所反映出的设计理念不同。其实 abstract class 表示的是"is-a"关系，interface 表示的是"like-a"关系。
(05).实现抽象类和接口的类必须实现其中的所有方法。抽象类中可以有非抽象方法。接口中则不能有实现方法。
abstract class 和 interface 是 Java 语言中的两种定义抽象类的方式，它们之间有很大的相似性。但是对于它们的选择却又往往反映出对于问题领域中的概念本质的理解、对于设计意图的反映是否正确、合理，因为它们表现了概念间的不同的关系。

19. Static class 与 non static class 的区别。

内部静态类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。非静态内部类能够访问外部类的静态和非静态成员。静态类不能访问外部类的非静态成员。他只能访问外部类的静态成员。一个非静态内部类不能脱离外部类实体被创建，一个非静态内部类可以访问外部类的数据和方法，因为他就在外部类里面。

20. java 多态的实现原理。

Sad

21. 实现多线程的两种方法：Thread 与 Runable。

实现多线程有两种方式：(自 JDK1.5 之后有三种，最后一种并不常用)
(1).继承 Thread 类
(2).实现 Runnable 接口（Callable 接口）
一个类如果实现了 Runnable 接口或者继承了 Thread 类，那么它就是一个多线程类，如果是要实现多线程，还需要重写 run()方法，所以 run() 方法是多线程的入口。
但是在启动多线程的时候，不是从 run()方法开始的，而是从 start()开始的 理由是：当执行多线程的时候，每一个线程会抢占资源，而操作系统会为其分配资源，在 start()方法中不仅执行了多线程的代码，除此还调用了一个 start0()方法，该方法的声明是 native,在 Java 语言中用一种技术叫做 JNI，即 JavaNativeInterface，该技术特点是使用 Java 调用本机操作系统提供的函数,但是有一个缺点是不能离开特定的操作系统,如果线程需要执行，必须有操作系统去分配资源，所以此操作主要是 JVM 根据不同的操作系统来实现的
如果多线程是通过实现 Runnable 接口来实现的,那么与通过继承 Thread 来实现有一个区别，那就是多线程的启动方式——必须是通过 start()来启动，但是 Runnable 接口只有一个方法，并没有 start()方法，所以在启动多线程的时候必须调用 Thread 类的一个构造方法——Thread(Runnable target),该构造方法得到了 Runnable 接口的一个实现，于是就可以调用 Thread 类的 start()方法了。

多线程的两种实现方式的区别：
(1).Thread 是 Runnable 接口的子类，实现 Runnable 接口的方式解决了 Java 单继承的局限
(2).Runnable 接口实现多线程比继承 Thread 类更加能描述数据共享的概念

```
public class Thread_Runnable implements Runnable {
 int ticket=10;
 @Override  public void run()
{
for (int i = 0; i < 100; i++) {
      if(ticket>0){
       System.out.println(ticket--);
       }
  }
}
public static void main(String[] args) {
     Thread_Runnable Thread_Runnable=new Thread_Runnable();
      new Thread(Thread_Runnable).start();
     new Thread(Thread_Runnable).start();
     new Thread(Thread_Runnable).start();
    MyThread myThread1=new MyThread();
      MyThread myThread2=new MyThread();
      MyThread myThread3=new MyThread();
      myThread1.start();
       myThread2.start();
      myThread3.start();
 }
} class MyThread extends Thread{
 int ticket=10;
 @Override  public void run() {
  for (int i = 0; i < 100; i++) {
       if(ticket>0){
                 System.out.println(ticket--);
       }
    }
  }
}
```

实现 Runnable 的多线程指挥执行 10 次，继承 Thread 类的多线程会执行 30 次，每个线程 10 次。

22. 线程同步的方法：sychronized、lock、reentrantLock 等。

如果你向一个变量写值，而这个变量接下来可能会被另一个线程所读取，或者你从一个变量读值，而它的值可能是前面由另一个线程写入的，此时你就必须使用同步。
sychronizedJava 语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证

在同一时刻最多只有一个线程执行该段代码，它是在 软件层面依赖 JVM 实现同步。
synchronized 方法或语句的使用提供了对与每个对象相关的隐式监视器锁的访问，但却强制所有锁获取和释放均要出现在一个块结构中：当获取了多个锁时，它们必须以相反的顺序释放，且必须在与所有锁被获取时相同的词法范围内释放所有锁。
通过在方法声明中加入 synchronized 关键字来声明 synchronized 方法。
synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能
执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行
状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）
synchronized 方法的缺陷：若将一个大的方法声明为 synchronized 将会大大影响效率，典型地，若将线程类的方法 run() 声明为 synchronized ，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 synchronized 方法的调用都永远不会成功。
解决 synchronized 方法的缺陷
通过 synchronized 关键字来声明 synchronized 块。
[code]synchronized(lock) {
// 访问或修改被锁保护的共享状态
}
其中的代码必须获得对象 syncObject （类实例或类）的锁方能执行。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。
当两个并发线程访问同一个对象中的这个 synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。
当一个线程访问对象的一个 synchronized(this)同步代码块时，另一个线程仍然可以访问该对象中的非 synchronized(this)同步代码块。其他线程对对象中所有其它 synchronized(this)同步代码块的访问将被阻塞。
如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续） synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。
在修饰代码块的时候需要一个 reference 对象作为锁的对象.
在修饰方法的时候默认是当前对象作为锁的对象.
在修饰类时候默认是当前类的 Class 对象作为锁的对象.
lockLock 接口实现提供了比使用 synchronized 方法和语句可获得的更广泛的锁定操作。此实现允许更灵活的结构，可以具有差别很大的属性，可以支持多个相关的 Condition 对象。在硬件层面依赖特殊的 CPU 指令实现同步更加灵活。
什么是 Condition ？
Condition 接口将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set（wait-set）。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。
虽然 synchronized 方法和语句的范围机制使得使用监视器锁编程方便了很多，而且还帮助

避免了很多涉及到锁的常见编程错误，但有时也需要以更为灵活的方式使用锁。例如，某些遍历并发访问的数据结果的算法要求使用 “hand-over-hand” 或 “chain locking”：获取节点 A 的锁，然后再获取节点 B 的锁，然后释放 A 并获取 C，然后释放 B 并获取 D，依此类推。Lock 接口的实现允许锁在不同的作用范围内获取和释放，并允许以任何顺序获取和释放多个锁，从而支持使用这种技术。
随着灵活性的增加，也带来了更多的责任。不使用块结构锁就失去了使用 synchronized 方法和语句时会出现的锁自动释放功能。在大多数情况下，应该使用以下语句：
[code]
```
Lock l = ...; //lock 接口的实现类对象
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```
在 java.util.concurrent.locks 包中有很多 Lock 的实现类，常用的有 ReentrantLock、ReadWriteLock（实现类 ReentrantReadWriteLock）.它们是具体实现类，不是 java 语言关键字。
ReentrantLock 一个可重入的互斥锁 Lock，它具有与使用 synchronized 方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大。
最典型的代码如下：
[code]
```
class X {
  private final ReentrantLock lock = new ReentrantLock();
  // ...

  public void m() {
    lock.lock();  // block until condition holds
    try {
      // ... method body
    } finally {
      lock.unlock()
    }
  }
}
```
重入性：指的是同一个线程多次试图获取它所占有的锁，请求会成功。当释放锁的时候，直到重入次数清零，锁才释放完毕。
ReentrantLock 的 lock 机制有 2 种，忽略中断锁和响应中断锁，这给我们带来了很大的灵活性。比如：如果 A、B 2 个线程去竞争锁，A 线程得到了锁，B 线程等待，但是 A 线程这个时候实在有太多事情要处理，就是 一直不返回，B 线程可能就会等不及了，想中断自己，不再等待这个锁了，转而处理其他事情。这个时候 ReentrantLock 就提供了 2 种机制，第一，B 线程中断自己（或者别的线程中断它），但是 ReentrantLock 不去响应，继续让 B 线程等待，你再怎么中断，我全当耳边风（synchronized 原语就是如此）；第二，B 线程中断自己（或者别的线程中断它），ReentrantLock 处理了这个中断，并且不再等待这个锁的到来，完全放弃。 ReentrantLock 相对于 synchronized 多了三个高级功能：
①等待可中断

在持有锁的线程长时间不释放锁的时候,等待的线程可以选择放弃等待.
[code]tryLock(long timeout, TimeUnit unit)
②公平锁
按照申请锁的顺序来一次获得锁称为公平锁.synchronized 的是非公平锁,ReentrantLock 可以通过构造函数实现公平锁.
[code]new RenentrantLock(boolean fair)
公平锁和非公平锁。这 2 种机制的意思从字面上也能了解个大概：即对于多线程来说，公平锁会依赖线程进来的顺序，后进来的线程后获得锁。而非公平锁的意思就是后进来的锁也可以和前边等待锁的线程同时竞争锁资源。对于效率来讲，当然是非公平锁效率更高，因为公平锁还要判断是不是线程队列的第一个才会让线程获得锁。
③绑定多个 Condition
通过多次 newCondition 可以获得多个 Condition 对象,可以简单的实现比较复杂的线程同步的功能.通过 await(),signal();
synchronized 和 lock 的用法与区别 synchronized 是托管给 JVM 执行的，而 lock 是 java 写的控制锁的代码。
synchronized 原始采用的是 CPU 悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在 CPU 转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起 CPU 频繁的上下文切换导致效率很低。
Lock 用的是乐观锁方式。每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。
ReentrantLock 必须在 finally 中释放锁,否则后果很严重,编码角度来说使用 synchronized 更加简单，不容易遗漏或者出错。
ReentrantLock 提供了可轮询的锁请求,他可以尝试的去取得锁,如果取得成功则继续处理,取得不成功，可以等下次运行的时候处理，所以不容易产生死锁，而 synchronized 则一旦进入锁请求要么成功，要么一直阻塞，所以更容易产生死锁。
synchronized 的话，锁的范围是整个方法或 synchronized 块部分；而 Lock 因为是方法调用，可以跨方法，灵活性更大
一般情况下都是用 synchronized 原语实现同步，除非下列情况使用 ReentrantLock
①某个线程在等待一个锁的控制权的这段时间需要中断
②需要分开处理一些 wait-notify,ReentrantLock 里面的 Condition 应用,能够控制 notify 哪个线程
③具有公平锁功能，每个到来的线程都将排队等候
了解 Lock 接口的实现类 ReentrantReadWriteLock 读写锁
前面提到的 ReentrantLock 是排他锁，该锁在同一时刻只允许一个线程来访问，而读写锁在同一时刻允许可以有多个线程来访问，但在写线程访问时，所有的读线程和其他写线程被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过读写锁分离，使得并发性相比一般的排他锁有了很大的提升。
读写锁除了使用在写操作 happends-before 与读操作以及并发性的提升之外,读写锁也能够简化读写交互场景的编程方式。假设在程序中定义一个共享的用作缓存数据结构，它的大部分时间提供读服务(查询，搜索等)而写操作较少，但写操作之后需要立即对后续的读操作可见。在没有读写锁之前，实现这个功能需要使用等待通知机制(http://blog.csdn.net/canot/article/details/50879963)。无论使用那种方式，目的都是为了写操作立即可见于读操作而避免脏读。但使用读写锁却比等待通知简单明了多了。
一般情况下，读写锁性能优于排他锁。它能提供更好的并发性和吞吐量。

ReentrantReadWriteLock 读写锁的几个特性:
公平选择性
重进入
锁降级
读写锁的示例:缓存

```
public class Cache{
  static Map<String,Object> map = new HashMap<String,Object>();
  static ReentrantReadWriteLock  rwl = new ReentrantReadWriteLock();
  static Lock rLock = rwl.readLock();
  static Lock wLock = rwl.writeLock();
  //获取一个 key 对应的 value
  public static final Object get(String key){
  r.lock();
  try{
   return map.get(key);
   }finally{
    r.unlock();
    }
  }
  //设置 key 对应的 value 并返回旧的 value
  public static fianl Object put(String key,Object value){
  w.lock();
  try{
   return map.put(key,value);
   }final{
   w.unlock();
    }
  }
  //清空缓存
  public static fianl void clear(){
  w.lock();
  try{
     map.clear();
   } finally{
    w.unlock();
    }
  }
}
```

上述缓存示例中,我们使用了一个非线程安全的 HashMap 作为缓存的时候然后使用读写锁来保证线程安全。Cache 使用读写锁提升读操作的并发性，也保证每次写操作对读操作的及时可见性，同时简化了编程方式。
读写锁的锁降级
锁降级是指写锁降级成为读锁。如果当前线程持有写锁，然后将其释放再获取读锁的过程不能称为锁降级。锁降级指的在持有写锁的时候再获取读锁,获取到读锁后释放之前写锁的过

程称为锁释放。
锁降级在某些情况下是非常必要的，主要是为了保证数据的可见性。如果当前线程不获取读锁而直接释放写锁，假设此时另外一个线程获取了写锁并修改了数据。那么当前线程无法感知该线程的数据更新。

23. 锁的等级：方法锁、对象锁、类锁。

对象锁(方法锁)是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步

```
// 类锁：形式 1
public static synchronized void Method1()
// 类锁：形式 2
public void Method 2 ()
{
    synchronized (Test.class)
    {
        System.out.println(" 我是类锁二号 ");
        try
        {
            Thread.sleep(500);
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

24. 写出生产者消费者模式。

```
public class Product {

public int product = 0;

    public final static int MAX = 999;

    public final static int MIN = 0;

    public synchronized void produce(){

         if(this.product >= MAX){
             try{
```

```
                wait();
                System.out.println("仓库已满，请等会再生产");
            }catch(Exception e){
                System.out.println("wait方法异常");
            }
            return;
        }
        this.product ++;
        System.out.println("++++正在生产第" + this.product +"个产品");
        notifyAll();
    }
    public synchronized void consume(){

        if(this.product <= MIN){
            try{
                wait();
                System.out.println("仓库缺货，请等会再消费");
            }catch(Exception e){
                System.out.println("wait方法异常");
            }
            return;
        }

        System.out.println("----正在消费第" + this.product +"个产品");
        this.product --;
        notifyAll();
    }

}
```

25. ThreadLocal 的设计理念与作用。

在 JDK 的早期版本中,提供了一种解决多线程并发问题的方案： java.lang.ThreadLocal 类.ThreadLocal 类在维护变量时,实际使用了当前线程（Thread）中的一个叫做 ThreadLocalMap 的独立副本,每个线程可以独立修改属于自己的副本而不会互相影响,从而隔离了线程和线程,避免了线程访问实例变量发生冲突的问题.
ThreadLocal 本身并不是一个线程,而是通过操作当前线程（Thread）中的一个内部变量来达到与其他线程隔离的目的.之所以取名为 ThreadLocal,所期望表达的含义是其操作的对象是线程（Thread）的一个本地变量.
ThreadLocal 类的大致结构和进行 ThreadLocalMap 的操作.我们可以从中得出以下的结论:1. ThreadLocalMap 变量属于线程（Thread）的内部属性,不同的线程（Thread）拥有完全不同的 ThreadLocalMap 变量.2. 线程(Thread)中的 ThreadLocalMap 变量的值是在 ThreadLocal 对象进行 set 或者 get 操作时创建的.3. 在创建 ThreadLocalMap 之前,会首先检查当前线程

（Thread)中的 ThreadLocalMap 变量是否已经存在,如果不存在则创建一个;如果已经存在,则使用当前线程（Thread）已创建的 ThreadLocalMap.4. 使用当前线程（Thread）的 ThreadLocalMap 的关键在于使用当前的 ThreadLocal 的实例作为 key 进行存储 ThreadLocal 模式,至少从两个方面完成了数据访问隔离,有了横向和纵向的两种不同的隔离方式,ThreadLocal 模式就能真正地做到线程安全：纵向隔离 —— 线程（Thread）与线程（Thread）之间的数据访问隔离.这一点由线程（Thread）的数据结构保证.因为每个线程（Thread）在进行对象访问时,访问的都是各自线程自己的 ThreadLocalMap.横向隔离 —— 同一个线程中,不同的 ThreadLocal 实例操作的对象之间的相互隔离.这一点由 ThreadLocalMap 在存储时,采用当前 ThreadLocal 的实例作为 key 来保证.
结论 ： 使用 ThreadLocal 模式,可以使得数据在不同的编程层次得到有效地共享,
这一点,是由 ThreadLocal 模式的实现机理决定的.因为实现 ThreadLocal 模式的一个重要步骤,就是构建一个静态的共享存储空间.从而使得任何对象在任何时刻都可以安全地对数据进行访问.
结论 使用 ThreadLocal 模式,可以对执行逻辑与执行数据进行有效解耦
这一点是 ThreadLocal 模式给我们带来的最为核心的一个影响,因为在一般情况下,Java 对象之间的协作关系，主要通过参数和返回值来进行消息传递,这也是对象协作之间的一个重要依赖,而 ThreadLocal 模式彻底打破了这种依赖关系,通过线程安全的共享对象来进行数据共享,可以有效避免在编程层次之间形成数据依赖,这也成为了 XWork 事件处理体系设计的核心.

26. ThreadPool 用法与优势。

(1). 引言
合理利用线程池能够带来三个好处。第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。
(2). 线程池的使用
线程池的创建
我们可以通过 ThreadPoolExecutor 来创建一个线程池。
new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, milliseconds,runnableTaskQueue, handler);
创建一个线程池需要输入几个参数：
corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的 prestartAllCoreThreads 方法，线程池会提前创建并启动所有基本线程。
runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列。
ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按 FIFO （先进先出） 排序

元素，吞吐量通常要高于 ArrayBlockingQueue。静态工厂方法 Executors.newFixedThreadPool()使用了这个队列。
SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlockingQueue，静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。
PriorityBlockingQueue：一个具有优先级的无限阻塞队列。
maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。
RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是 AbortPolicy，表示无法处理新任务时抛出异常。以下是 JDK1.5 提供的四种策略。
AbortPolicy：直接抛出异常。
CallerRunsPolicy：只用调用者所在线程来运行任务。
DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
DiscardPolicy：不处理，丢弃掉。
当然也可以根据应用场景需要来实现 RejectedExecutionHandler 接口自定义策略。如记录日志或持久化不能处理的任务。
keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
TimeUnit(线程活动保持时间的单位):可选的单位有天(DAYS),小时(HOURS),分钟(MINUTES),毫秒(MILLISECONDS),微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。
向线程池提交任务
我们可以使用 execute 提交的任务，但是 execute 方法没有返回值，所以无法判断任务是否被线程池执行成功。通过以下代码可知 execute 方法输入的任务是一个 Runnable 类的实例。

```
threadsPool.execute(new Runnable() {
            @Override
            public void run() {
                // TODO Auto-generated method stub
            }
        });
```

我们也可以使用 submit 方法来提交任务，它会返回一个 future,那么我们可以通过这个 future 来判断任务是否执行成功，通过 future 的 get 方法来获取返回值，get 方法会阻塞住直到任务完成，而使用 get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

```
Future<Object> future = executor.submit(harReturnValuetask);
try {
     Object s = future.get();
} catch (InterruptedException e) {
    // 处理中断异常
} catch (ExecutionException e) {
```

```
    // 处理无法执行任务异常
} finally {
    // 关闭线程池
    executor.shutdown();
}
```

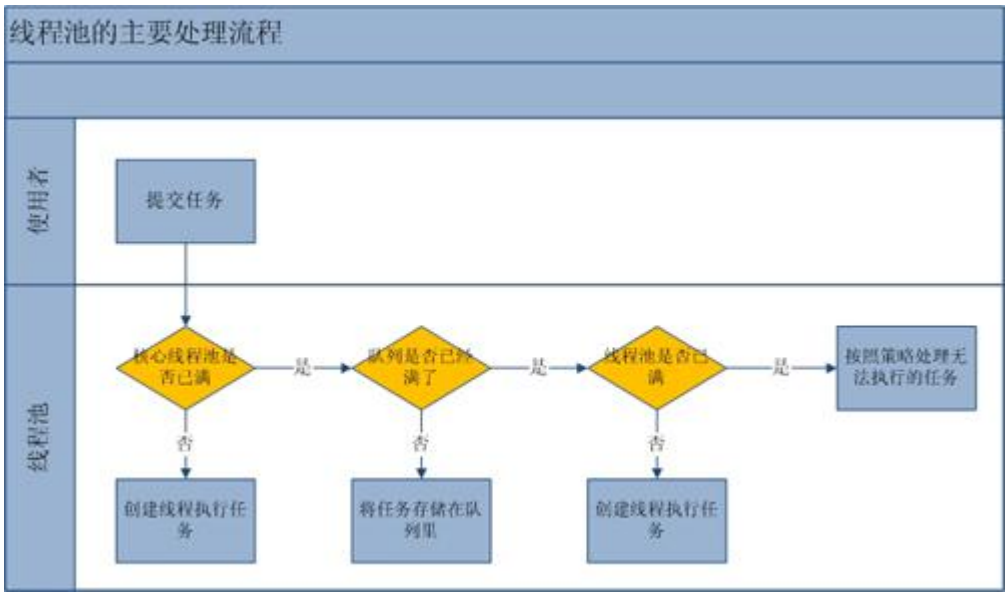
线程池的关闭

我们可以通过调用线程池的 shutdown 或 shutdownNow 方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的 interrupt 方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow 首先将线程池的状态设置成 STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而 shutdown 只是将线程池的状态设置成 SHUTDOWN 状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，isShutdown 方法就会返回 true。当所有的任务都已关闭后,才表示线程池关闭成功，这时调用 isTerminaed 方法会返回 true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用 shutdown 来关闭线程池，如果任务不一定要执行完，则可以调用 shutdownNow。

(3). 线程池的分析

流程分析：线程池的主要工作流程如下图：



从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

首先线程池判断基本线程池是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。

其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。

最后线程池判断整个线程池是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

源码分析。上面的流程分析让我们很直观的了解了线程池的工作原理，让我们再通过源代码来看看是如何实现的。线程池执行任务的方法如下：

```
public void execute(Runnable command) {
    if (command == null)
```

```
        throw new NullPointerException();
    //如果线程数小于基本线程数，则创建线程并执行当前任务
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
    //如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。
        if (runState == RUNNING && workQueue.offer(command)) {
            if (runState != RUNNING || poolSize == 0)
                    ensureQueuedTaskHandled(command);
        }
    //如果线程池不处于运行中或任务无法放入队列,并且当前线程数量小于最大允许的线
程数量,
则创建一个线程执行任务。
        else if (!addIfUnderMaximumPoolSize(command))
        //抛出 RejectedExecutionException 异常
            reject(command); // is shutdown or saturated
    }
}
```

工作线程。线程池创建线程时，会将线程封装成工作线程 Worker，Worker 在执行完任务后，还会无限循环获取工作队列里的任务来执行。我们可以从 Worker 的 run 方法里看到这点：

```
public void run() {
     try {
           Runnable task = firstTask;
           firstTask = null;
            while (task != null || (task = getTask()) != null) {
                    runTask(task);
                    task = null;
            }
      } finally {
             workerDone(this);
      }
}
```

(4). 合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

任务的性质：CPU 密集型任务，IO 密集型任务和混合型任务。

任务的优先级：高，中和低。

任务的执行时间：长，中和短。

任务的依赖性：是否依赖其他系统资源，如数据库连接。

任务性质不同的任务可以用不同规模的线程池分开处理。CPU 密集型任务配置尽可能小的线程，如配置 Ncpu+1 个线程的线程池。IO 密集型任务则由于线程并不是一直在执行任务，则配置尽可能多的线程，如 2*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个 CPU 密集型任务和一个 IO 密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过 Runtime.getRuntime().availableProcessors()方法获得当前设备的 CPU 个数。

优先级不同的任务可以使用优先级队列 PriorityBlockingQueue 来处理。它可以让优先级高

的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。
执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。
依赖数据库连接池的任务，因为线程提交 SQL 后需要等待数据库返回结果，如果等待的时间越长 CPU 空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用 CPU。
建议使用有界队列，有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断的抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行 SQL 变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞住，任务积压在线程池里。如果当时我们设置成无界队列，线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然我们的系统所有的任务是用的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。
(5). 线程池的监控
通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的时候可以使用
taskCount：线程池需要执行的任务数量。
completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于 taskCount。
largestPoolSize：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
getPoolSize:线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不+ getActiveCount：获取活动的线程数。
通过扩展线程池进行监控。通过继承线程池并重写线程池的 beforeExecute，afterExecute 和 terminated 方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。
如：
```
protected void beforeExecute(Thread t, Runnable r) { }
```

27. Concurrent 包里的其他东西： ArrayBlockingQueue、CountDownLatch 等等。

java.util.concurrent 包分成了三个部分，分别是 java.util.concurrent 、java.util.concurrent.atomic 和 java.util.concurrent.lock。内容涵盖了并发集合类、线程池机制、同步互斥机制、线程安全的变量更新工具类、锁等等常用工具。
java.util.concurrent 包中提供了几个并发结合类，例如 ConcurrentHashMap 、ConcurrentLinkedQueue 和 CopyOnWriteArrayList 等等
对变量的读写操作都是原子操作（除了 long 或者 double 的变量），但像数值类型的++ --操作不是原子操作，像 i++中包含了获得 i 的原始值、加 1、写回 i、返回原始值，在进行类似 i++这样的操作时如果不进行同步问题就大了。好在 java.util.concurrent.atomic 为我们提供了很多工具类，可以以原子方式更新变量。
以 AtomicInteger 为例，提供了代替++ --的 getAndIncrement()、incrementAndGet()、getAndDecrement()和 decrementAndGet()方法，还有加减给定值的方法、当前值等于预期

值时更新的 compareAndSet()方法。

28. wait()和 sleep()的区别。

① 这两个方法来自不同的类分别是，sleep 来自 Thread 类，和 wait 来自 Object 类。
sleep 是 Thread 的静态类方法，谁调用的谁去睡觉，即使在 a 线程里调用 b 的 sleep 方法，实际上还是 a 去睡觉，要让 b 线程睡觉要在 b 的代码中调用 sleep。
② 锁：最主要是 sleep 方法没有释放锁，而 wait 方法释放了锁，使得其他线程可以使用同步控制块或者方法。
sleep 不出让系统资源；wait 是进入线程等待池等待，出让系统资源，其他线程可以占用 CPU。一般 wait 不会加时间限制，因为如果 wait 线程的运行资源不够，再出来也没用，要等待其他线程调用 notify/notifyAll 唤醒等待池中的所有线程，才会进入就绪队列等待 OS 分配系统资源。sleep(milliseconds)可以用时间指定使它自动唤醒过来，如果时间不到只能调用 interrupt()强行打断。
Thread.sleep(0)的作用是“触发操作系统立刻重新进行一次 CPU 竞争”。
③ 使用范围：wait，notify 和 notifyAll 只能在同步控制方法或者同步控制块里面使用，而 sleep 可以在任何地方使用。

```
synchronized(x){
   x.notify()
  //或者 wait()
}
```

29. foreach 与正常 for 循环效率对比。

需要**循环数组结构**的数据时，建议**使用普通 for 循环**，因为 for 循环采用下标访问，对于数组结构的数据来说，采用下标访问比较好。

需要**循环链表结构**的数据时，**一定不要使用普通 for 循环**，这种做法很糟糕，数据量大的时候有可能会导致系统崩溃。

30. Java IO 与 NIO。

31. 反射的作用于原理。

一、 原理

简单的来说，反射机制其实就是指程序在运行的时候能够获取自身的信息。如果知道一个类的名称/或者它的一个实例对象， 就能把这个类的所有方法和变量的信息(方法名，变量名，方法，修饰符，类型，方法参数等等所有信息)找出来。如果明确知道这个类里的某个方法名+参数个数 类型，还能通过传递参数来运行那个类里的那个方法，这就是反射。

尽管 Java 不是一种动态语言，但它却有一个非常突出的动态机制：Reflection。它使我们可以于运行时加载、探知、使用编译期间完全未知的 classes。换句话说，Java 程序可以加载一个运行时才得知名称的 class，获悉其完整构造（但不包括 methods 定义），并生成其对象实体、 或对其 fields 设值、或唤起其 methods。既一种“看透 class”的能力。

当然，在平时的编程中，反射基本用不到，但是在编写框架的时候，反射用的就多了，比如你要使用某一个类进行操作，但是这个类是用户通过配置文件配置进来 的，你需要先读配置文件，然后拿到这个类的全类名：比如 test.Person,然后在利用反射 API 来完成相应的操作。

二、 优缺点

反射的优点当然是体现在它的动态性上面，能运行时确定类型，绑定对象。动态编译最大限度发挥了 java 的灵活性，体现了多态的应用，降低类之间的藕合性。 一句话，反射机制的优点就是可以实现动态创建对象和编译，特别是在 J2EE 的开发中，它的灵活性就表现的十分明显。比如，一个大型的软件，不可能一次就把 它设计的很完美，当这个程序编译后，发布了，当发现需要更新某些功能时，我们不可能要用户把以前的卸载，再重新安装新的版本，假如这样的话，这个软件肯定 是没有多少人用的。采用静态的话，需要把整个程序重新编译一次才可以实现功能的更新，而采用反射机制的话，它就可以不用卸载，只需要在运行时才动态的创建 和编译，就可以实现该功能。

它的缺点是对性能有影响。使用反射基本上是一种解释操作，我们可以告诉 JVM，我们希望做什么并且它满足我们的要求。这类操作总是慢于只直接执行相同的操作

32. 泛型常用特点，List<String>能否转为 List<Object>。

不能

(1)、性能（最主要的优点）：

因为使用非泛型类来存储值类型或把引用类型转换为值类型时需要装箱和拆箱的操作。频繁的进行装箱和拆箱的操作会使系统性能下降，耗费资源。例如：

使用 System.Collection 命名空间的 ArrayList 类存储对象时，Add 方法会将各个对象保存到该类中，这时若是值类型作为对象，会把值类型转换为引用类型，即会出现装箱操作，当要使用该类中的值时就要把该引用类型转换为值类型，即会出现拆箱操作。如下例子，把 int 型转为 ArrayList 类的一个对象，进行了装箱操作，然后把 Arraylist 的一个对象转换为 int i1 时会需要进行强制转换为 int 类型，进行了拆箱操作，当使用 foreach（）循环遍历 Arraylist 时也会自动的进行拆箱操作，

因为泛型不是类，不能直接创建对象，而是在使用时定义类型后再创建对象

(2)、类型安全

与使用 Arraylist 类一样，如果对象，就可以在这个集合中添加任意类型的对象，例如：

```
var list=new ArrayList();
list.Add(44);
list.Add("string");
list.Add("MyNewClass()");
```

这时如果使用 foreach 遍历时就会出现类型转换的异常。而如果使用泛型 List<T>,泛型类型 T 需要程序员自己指定，这样也就避免了隐式类型转换的不可预料的异常。因为泛型 List<T>的类型被定义后就只能把该类型的数据添加到集合中，不然编译器会因为 Add（）方法的参数无效而报错。
(3)、二进制代码的重用。一个泛型可供多个类型使用，因为泛型是在运行时工作的，所以.net 平台中，泛型可以在一种语言定义，在任何其他.net 语言中也可以使用。
(4)、代码的扩展
因为泛型类的定义会放在程序集中，所以用特定类型实例化泛型类不会在 IL 代码中复制这些类。但是，在 JiT 编译器把泛型类编译为本地代码时，会给每个值类型创建一个新类，引用类型共享同一个本地类的所有相同的实现代码。

33. 解析 XML 的几种方式的原理与特点：DOM、SAX、PULL。

XML 的解析方式有很多，光开源的就有十多种：如 Xerces、JDOM、DOM4J、XOM、JiBX、KXML、XMLBeans、jConfig、XStream、XJR 等。
但是最常用的还是 sax、dom、pull、dom4j
而 android 中用的比较多的是 sax（Simple APIs for XML）、dom（Document Object Model）、pull，其中 pull 在这三个中又最为适用。（大部分用于 java 的解析器在 android 上都可以用，对于有人说 dom4j 最好，这个没试验过，暂时不好说，但是大部分人都说 pull 好）
SAX
sax 是一个用于处理 xml 事件驱动的“推”模型；
优点：解析速度快，占用内存少，它需要哪些数据再加载和解析哪些内容。
缺点：它不会记录标签的关系，而是需要应用程序自己处理，这样就会增加程序的负担。
DOM
dom 是一种文档对象模型；
优点：dom 可以以一种独立于平台和语言的方式访问和修改一个文档的内容和结构，dom 技术使得用户页面可以动态的变化，如动态显示隐藏一个元素，改变它的属性，增加一个元素等，dom 可以使页面的交互性大大增强。
缺点：dom 解析 xml 文件时会将 xml 文件的所有内容以文档树方式存放在内存中。
PULL
pull 和 sax 很相似，区别在于：pull 读取 xml 文件后触发相应的事件调用方法返回的是数字，且 pull 可以在程序中控制，想解析到哪里就可以停止解析。 （SAX 解析器的工作方式是自动将事件推入事件处理器进行处理，因此你不能控制事件的处理主动结束；而 Pull 解析器的工作方式为允许你的应用程序代码主动从解析器中获取事件，正因为是主动获取事件，因此可以在满足了需要的条件后不再获取事件，结束解析。pull 是一个 while 循环，随时可以跳出，而 sax 不是，sax 是只要解析了，就必须解析完成。）
1）DOM4J 性能最好，连 Sun 的 JAXM 也在用 DOM4J.目前许多开源项目中大量采用 DOM4J，例如大名鼎鼎的 Hibernate 也用 DOM4J 来读取 XML 配置文件。如果不考虑可移植性，那就采用 DOM4J.
2）JDOM 和 DOM 在性能测试时表现不佳，在测试 10M 文档时内存溢出。在小文档情况下还值得考虑使用 DOM 和 JDOM.虽然 JDOM 的开发者已经说明他们期望在正式发行版前专注性能问题，但是从性能观点来看，它确实没有值得推荐之处。另外，DOM 仍是一个非常好的选择。DOM 实现广泛应用于多种编程语言。它还是许多其它与 XML 相关的标准的基础，因为它正式

获得 W3C 推荐(与基于非标准的 Java 模型相对),所以在某些类型的项目中可能也需要它(如在 JavaScript 中使用 DOM)。
3）SAX 表现较好，这要依赖于它特定的解析方式－事件驱动。一个 SAX 检测即将到来的 XML 流，但并没有载入到内存（当然当 XML 流被读入时，会有部分文档暂时隐藏在内存中)。

34. Java 与 C++对比。

35. Java1.7 与 1.8 新特性。

36. 设计模式：单例、工厂、适配器、责任链、观察者等等。

（http://www.cnblogs.com/lyl6796910/p/4337362.html）
单列
Singleton 模式要求一个类有且仅有一个实例，并且提供了一个全局的访问点。这就提出了一个问题：如何绕过常规的构造器，提供一种机制来保证一个类只有一个实例？客户程序在调用某一个类时，它是不会考虑这个类是否只能有一个实例等问题的，所以，这应该是类设计者的责任，而不是类使用者的责任。
从另一个角度来说，Singleton 模式其实也是一种职责型模式。因为我们创建了一个对象，这个对象扮演了独一无二的角色,在这个单独的对象实例中,它集中了它所属类的所有权力,同时它也肩负了行使这种权力的职责！

```
public sealed class Singleton
{
    static Singleton instance=null;
    static readonly object padlock = new object();

    Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            if (instance==null)
            {
                lock (padlock)
                {
                    if (instance==null)
                    {
                        instance = new Singleton();
                    }
```

```
                }
            }
            return instance;
        }
    }
}
```

抽象工厂模式（Abstract Factory）

在软件系统中，经常面临着“一系列相互依赖的对象”的创建工作；同时由于需求的变化，往往存在着更多系列对象的创建工作。如何应对这种变化？如何绕过常规的对象的创建方法（new），提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？这就是我们要说的抽象工厂模式。

```
/// <summary>
/// Factory 类
/// </summary>
public class Factory
{
    public Tax CreateTax()
    {
        return new ChineseTax();
    }

    public Bonus CreateBonus()
    {
        return new ChineseBonus();
    }
}
```

观察者模式（Observer Pattern）

在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系” ——一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。使用面向对象技术，可以将这种依赖关系弱化，并形成一种稳定的依赖关系。从而实现软件体系结构的松耦合。

```
public abstract class Stock
{
    private List<IObserver> observers = new List<IObserver>();
    private String _symbol;
    private double _price;
    public Stock(String symbol, double price)
    {
        this._symbol = symbol;

        this._price = price;
    }
    public void Update()
    {
```

```
            foreach (IObserver ob in observers)
            {
                ob.SendData(this);
            }
        }
        public void AddObserver(IObserver observer)
        {
            observers.Add(observer);
        }
        public void RemoveObserver(IObserver observer)
        {
            observers.Remove(observer);
        }
        public String Symbol
        {
            get { return _symbol; }
        }
        public double Price
        {
            get { return _price; }
        }
    }
    public class Microsoft : Stock
    {
        public Microsoft(String symbol, double price)
            : base(symbol, price)
        { }
    }
    public interface IObserver
    {
        void SendData(Stock stock);
    }
    public class Investor : IObserver
    {
        private string _name;
        public Investor(string name)
        {
            this._name = name;
        }
        public void SendData(Stock stock)
        {
            Console.WriteLine("Notified {0} of {1}'s " + "change to {2:C}", _name,
    stock.Symbol,stock.Price);
        }
```

```
}
```
客户端程序代码如下：
```
 class Program
{
    static void Main(string[] args)
    {
        Stock ms = new Microsoft("Microsoft",120.00);

        ms.AddObserver(new Investor("Jom"));

        ms.AddObserver(new Investor("TerryLee"));

        ms.Update();

        Console.ReadLine();
    }
}
```
适配器模式（Adapter Pattern）
在软件系统中，由于应用环境的变化，常常需要将“一些现存的对象”放在新的环境中应用，但是新环境要求的接口是这些现存对象所不满足的。那么如何应对这种“迁移的变化”？如何既能利用现有对象的良好实现，同时又能满足新的应用环境所要求的接口？这就是本文要说的 Adapter 模式。

37. JNI 的使用。

Java 里有很多很杂的东西，有时候需要你阅读源码，大多数可能书里面讲的不是太清楚，需要你在网上寻找答案。
推荐书籍：《java 核心技术卷 I》《Thinking in java》《java 并发编程》《effictive java》《大话设计模式》

数据结构与算法

1. 链表与数组。
2. 队列和栈，出栈与入栈。
3. 链表的删除、插入、反向。
4. 字符串操作。

5. Hash 表的 hash 函数，冲突解决方法有哪些。

6. 各种排序：冒泡、选择、插入、希尔、归并、快排、堆排、桶排、基数的原理、平均时间复杂度、最坏时间复杂度、空间复杂度、是否稳定。

1. 选择排序

这个排序方法最简单，废话不多说，直接上代码：

```
public class SelectSort {

    /**

     * 选择排序

     * 思路:每次循环得到最小值的下标，然后交换数据。

     * 如果交换的位置不等于原来的位置，则不交换。

     */

    public static void main(String[] args) {

        selectSort(Datas.data);

        Datas.prints("选择排序");

    }

    public static void selectSort(int[] data){

        int index=0;

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

            index = i;

            for (int j = i; j < data.length; j++) {

                if (data[index]>data[j]) {

                    index = j;
```

```
                }
            }
            if (index != i) {
                swap(data,index,i);
            }
        }
    }
    public static void swap(int[] data,int i,int j){
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
}
```

选择排序两层循环，第一个层循环遍历数组，第二层循环找到剩余元素中最小值的索引，内层循环结束，交换数据。内层循环每结束一次，排好一位数据。两层循环结束，数据排好有序。

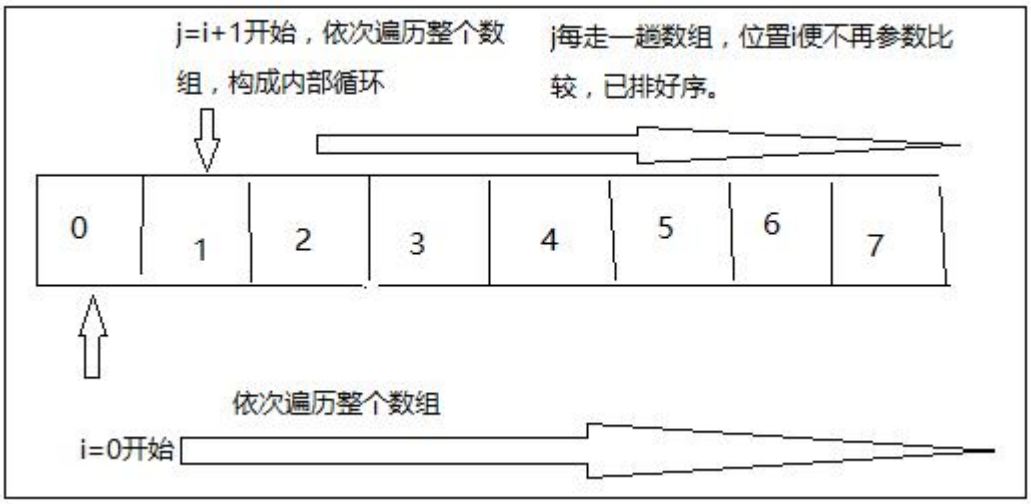
2 冒泡排序

冒泡排序也简单，上代码先：

```
public class BubbleSort {
    /**
     * 冒泡排序
     * 思路：内部循环每走一趟排好一位，依次向后排序
```

```
     */
    public static void main(String[] args) {
        bubbleSort(Datas.data);
    }

    private static void bubbleSort(int[] data) {
        int temp;
        for (int i = 0; i < data.length; i++) {
            for (int j = i+1; j < data.length; j++) {
                if (data[i]>data[j]) {
                    temp =data[i];
                    data[i]=data[j];
                    data[j] = temp;
                }
            }
        }
        Datas.prints("冒泡排序");
    }
}
```

冒泡排序和选择排序有点像，两层循环，内层循环每结束一次，排好一位数据。不同的是，数据像冒泡一样，不断的移动位置，内层循环结束，刚好移动到排序的位置。



该图对应上面的代码进行的说明，没有用专门的画图工具，使用的是 window 的 maspint，

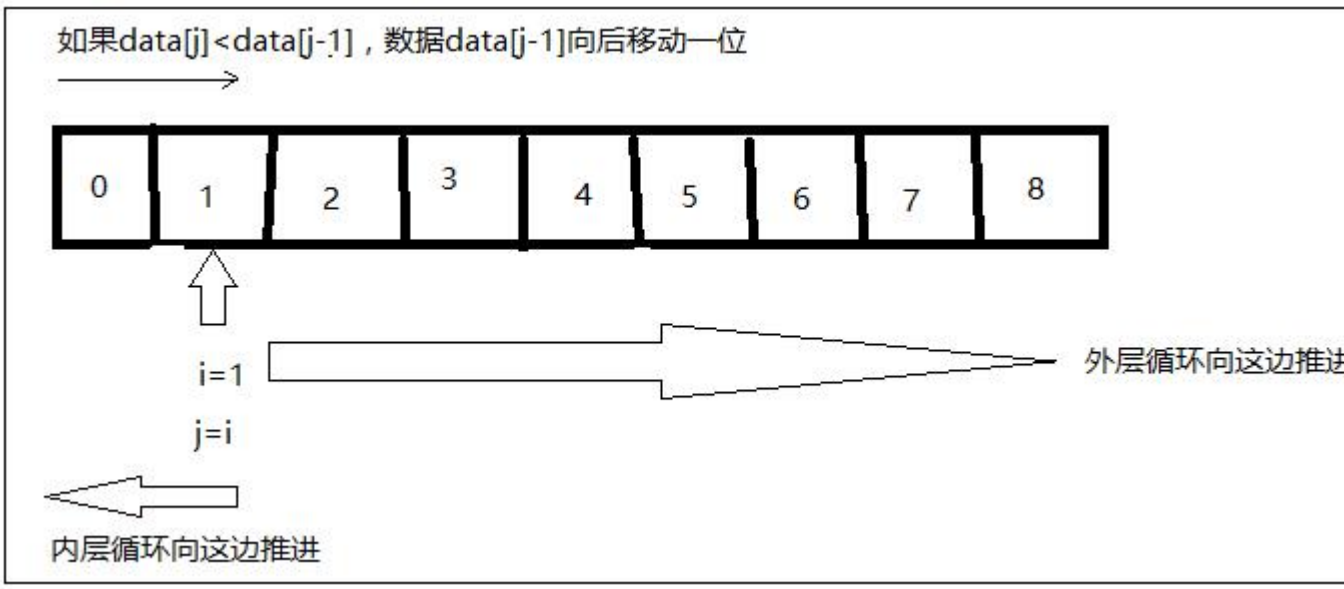
大家凑合着看哈^_^明白意思就成！

3 插入排序

插入排序也是简单的排序方法，代码量不多，先看代码：

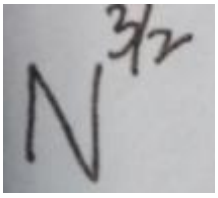
```
public class InsertSort {
    /**
     * 插入排序

     * 思路：将数据插入到已排序的数组中。

     */
    public static void main(String[] args) {
        int[] data = Datas.data;
        int temp;
        for (int i = 1; i < data.length; i++) {

            temp = data[i];//保存待插入的数值
```

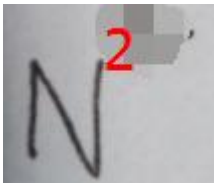
```
            int j = i;
            for (; j>0 && temp<data[j-1]; j--) {
                data[j] = data[j-1];
                //如果带插入的数值前面的元素比该值大，就向后移动一位
            }
            //内部循环结束，找到插入的位置赋值即可。
            data[j]=temp;
        }
        Datas.prints("插入排序");
    }
}
```



该图是上面插入排序的说明图，插入排序，其过程就是其名字说明的一样，将待排序的数据插入到已排序的数据当中。两层循环，内层循环结束一次，插入排序排好一位数据。

4 希尔排序

希尔排序，也叫缩减增量排序，其中增量的设置影响着程序的性能。最好的增量的设置为 1，3，5，7，11，。。。这样一组素数，并且各个元素之间没有公因子。这样的一组增量 叫做 Hibbard 增量。使用这种增量的希尔排序的最坏清醒运行时间为θ（  ）

当不使用这种增量时，希尔排序的最坏情形运行时间为θ（  ）

步骤 1：比如现在有数组{82 ,31 ,29 ,71, 72, 42, 64, 5,110}　第一次取增量设置为 array.length/2 = 4　先从 82 开始以 4 为增量遍历直到末尾，得到（82,42） 排序得到{42 ,31 ,29 ,71, 72, 82, 64, 5,110}。 然后从第二个数 31 开始重复上一个步骤，得到（31,64） 排序得到{42 ,31 ,29 ,71, 72, 82, 64, 5,110}.....　以 4 为增量的遍历完数组之后，得到的结果是{42 ,31,5,71,72,82,64,29,110}

然后重新区增量,这儿设定为incrementNum/2 = 2，对{42 ,31,5,71,72,82,64,29,110}重复步骤 1。　完事之后，在取新的增量，重复步骤 1。 直到取到的增量小于 1，退出循环。

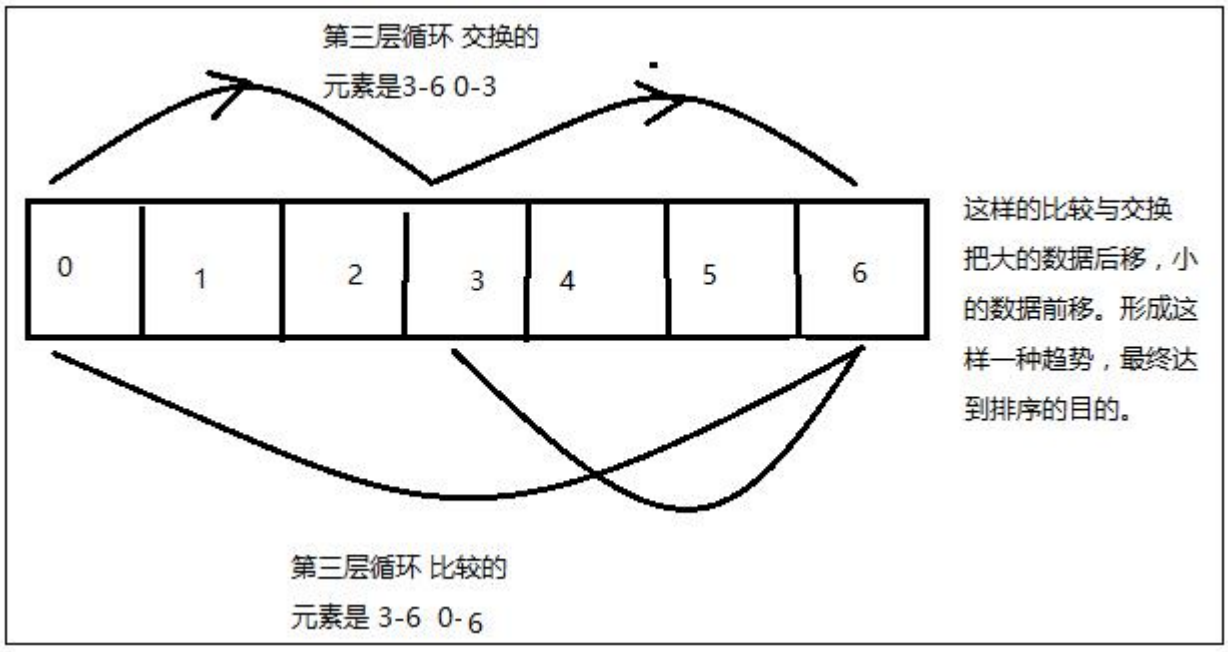
这里电脑打印这些太麻烦，干脆手写拍照啦哈哈哈哈。。。。。

好了，废话不多说，上代码；

```
public class ShellSort {


    /**

     * 希尔排序（缩减增量排序）

     * 想想也不难。
```

```
     * 思路：三层循环
     * 第一层循环：控制增量-增量随着程序的进行依次递减一半
     * 第二层循环：遍历数组
     * 第三层循环：比较元素，交换元素。
     * 这里需要注意的是：比较的两个元素和交换的两个元素是不同的。
     */
    public static void main(String[] args) {
        int[] data = Datas.data;
        int k;
        for (int div = data.length/2; div>0; div/=2) {
            for (int j = div; j < data.length; j++) {
                int temp = data[j];
                for (k=j; k>=div && temp<data[k-div] ; k-=div)
{
                    data[k] = data[k-div];
                }
                data[k] = temp;
            }
        }
        Datas.prints("希尔排序");
    }
```

}



程序中，需要注意的是第三层循环，第三层循环的代码中，if 语句的比较和内部的交换是分别不同的两个数据。原因是：把大的数据后移，小的数据前移，形成这样一种趋势，才能实现排序。

当然可以试试，if 语句比较的两个数据和内部移动的数据一致的话，会出现什么问题？出现的问题就是移动的数据打破了之前形成的大的数据在后，小的数据在前的趋势。无法排序。

5 堆排序

堆排序，要知道什么是堆？说白了，堆就是完全二叉树，堆是优先队列。要求父元素比两个子元素要大。这就好办了。数组元素构建堆，根节点最大，删除根节点得到最大值，剩下的元素再次构建堆，接着再删除根节点，得到第二大元素，剩下的元素再次构建堆，依次类推，得到一组排好序的数据。为了更好地利用空间，我们把删除的元素不使用新的空间，而是使用堆的最后一位保存删除的数据。

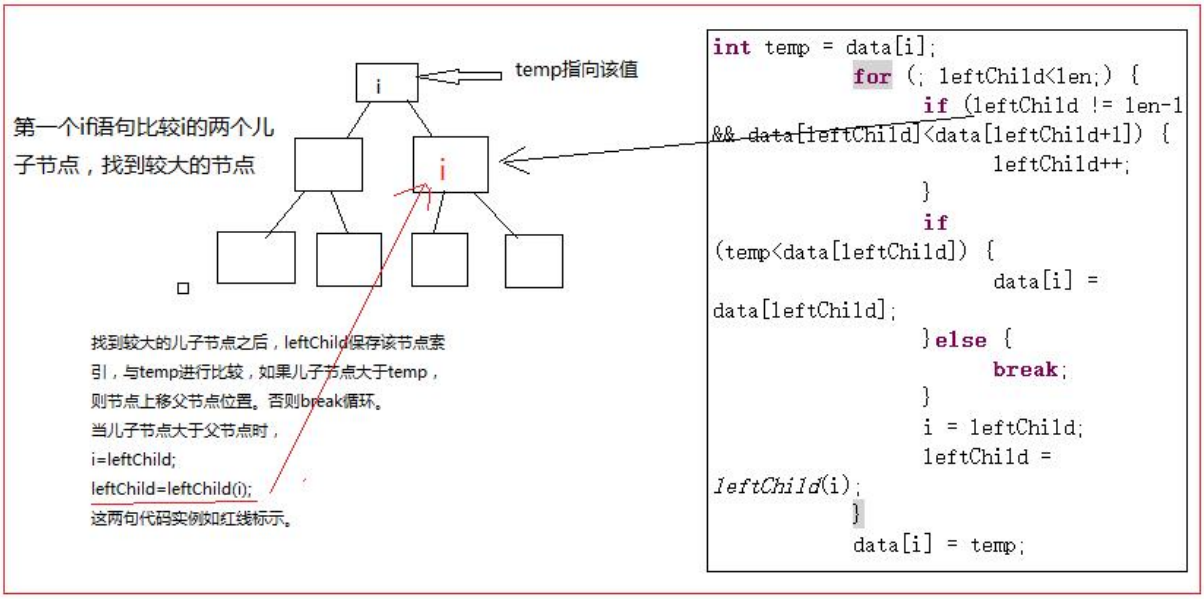
代码上来：

```
public class HeapSort {

    /**

     * 堆排序(就是优先队列)
```

```
     * 也就是完全二叉树
     * 第一步：建堆.其实就是讲数组中的元素进行下虑操作，
     *       使得数组中的元素满足堆的特性。
     * 第二步：通过将最大的元素转移至堆的末尾，
     *       然后将剩下的元素在构建堆。
     *       完成排序。
     * 最重要的过程就是构建堆的过程。
     * 里面的比较思路和希尔排序中的比较思路一致。
     * 将大的元素上浮，小的元素下浮。始终和 temp 比较。
     * temp 除了第一次比较可能改变外，其他次数的比较不改变该值。
     * 这样的处理就是让较大的元素趋于上浮，较小的元素下浮。
     */
    public static void main(String[] args) {
        int[] data = Datas.data;
        for (int i = data.length/2; i >=0; i--) {
            buildHeap(data,i,data.length);
        }
        Datas.prints("堆排序-构建树");
```

```
        System.out.println("============================");
        for (int i = data.length-1; i>0; i--) {
            swap(data, 0, i);
            buildHeap(data, 0, i);
        }
        Datas.prints("堆排序-排序后");
    }
    static void swap(int[] data,int i,int j){
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
    static void buildHeap(int[] data,int i,int len){
        int leftChild = leftChild(i);
        int temp = data[i];
        for (; leftChild<len;) {
            if (leftChild != len-1 && data[leftChild]<data[l
eftChild+1]) {
                leftChild++;
            }
            if (temp<data[leftChild]) {
                data[i] = data[leftChild];
            }else {
                /**braek 说明两个儿子都比父节点小，
```

```
                * 父节点大于两个儿子

                * 所以直接停止比较，减小比较的次数。

                */

                break;

            }

            i = leftChild;

            leftChild = leftChild(i);

        }

        data[i] = temp;

    }

    //返回节点 i 的左儿子的 index

    static int leftChild(int i){

        return 2*i+1;

    }

}
```



堆排序，最重要的就是构建堆，构建堆是核心！我们代码中使用的是数组形式的二叉树，也就是优先队列。要真正看懂这部分的代码，需要知道优先队列部分的知识，不难，看看就懂啦。说白了就是二叉树。

6 归并排序

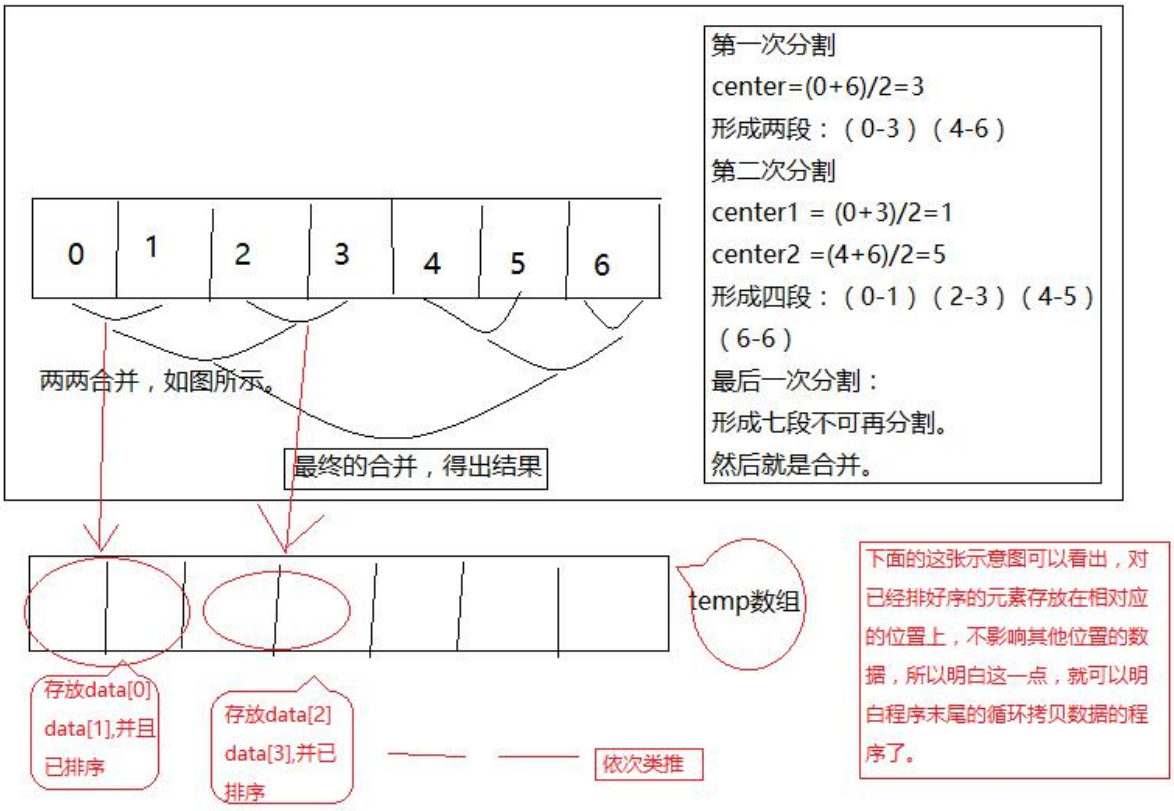
归并排序思路就是将两个已经排好序的数组插入到第三个数组当中。核心就是将原有数组分割两部分，排好序，插入到第三个与原有数组大小一致的数组中。代码上来：

```
public class Merge Sort {

    /**

     * 归并排序

     * 思路：如果是两个已排序的数组，进行合并非常简单。

     * 所以就对原有数组进行分割，分割成各个排序的数组，

     * 然后递归合并。

     */

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

```
        int[] data = Datas.data;
        merge(data);

        Datas.prints("归并排序");

    }
    public static void merge(int[] data){
        int[] temp = new int[data.length];
        merge0(data, temp, 0, data.length-1);
    }
    public static void merge0(int[] data,int[] temp,int lef
t,int rigth){
        if (left<rigth) {
            int center = (left+rigth)/2;
            merge0(data, temp, left, center);
            merge0(data, temp, center+1, rigth);
            mergeSort(data,temp,left,center,rigth);
        }
    }
    public static void mergeSort(int[] data,int[] temp,int
left,int center,int right){
        int leftEnd = center;
        int rightStar = center+1;
        int len = right-left+1;
        int tempPos = left;
        /**
```

```
 * 这里的三个循环很容易理解。

 * 其实现实两个已经排序的数组进行比较，

 * 将元素添加到 temp 数组中保存。

 */
while (left<=leftEnd&&rightStar<=right) {
    if (data[left]<=data[rightStar]) {
        temp[tempPos++] = data[left++];
    }else {
        temp[tempPos++] = data[rightStar++];
    }
}
while (left<=leftEnd) {
    temp[tempPos++]=data[left++];
}
while (rightStar<=right) {
    temp[tempPos++]=data[rightStar++];
}
/**

 * 关键的一步是下面的拷贝工作。

 * 为什么数组中的拷贝是从 right--开始？？？

 * 原因是：通过说明图中，我们知道，元素比较之后，
```

```
            * 会将元素赋值给 temp 数组相对应的位置上，并不会影响其他位置的数据。
            * 并且下面的循环中也没有使用其他位置上面的数据，仅仅拷贝
            * 本次已经排序的元素。
            * 下面的拷贝是从 right 开始，right 位置是本次排序最右边的元素
            * 其实也可以从 left 开始，只不过 left 在上面的排序中值已经改变，
            * 可以定义一个 int leftFlag = left;保存初始最左边的位置，
            * 此时下面的循环可以改为：
            * for (int i = 0; i < len; i++,leftFlag++) {
            *  data[leftFlag]=temp[leftFlag];
            * }
            * 运行程序，你会发现，正确输出结果。
            */
           for (int i = 0; i < len; i++,right--) {
               data[right]=temp[right];
           }
       }
   }
```



说明图中已经说明了关于**数组分割，排序、归并的步骤**。归并排序其实就是分割，排序，归并，最后得到排序的结果。

排序要等到分割完成之后进行，归并要等盗排序之后进行。

分割通过递归调用进行，排序通过程序中的三个 while 循环进行，即完成了归并。最后将数据拷贝要原来的数组中去。

7 快速排序

快速排序有点类似于归并排序，其实也是分割，不同的是，快速排序的分割是按照中值进行分割的，所以中值的好坏影响着程序的性能。最常见的情形是**三数中值分割法**！！

该方法的思路是选取左，右、中三个数进行交换，把三个数中最小值放在左边，中间值放在中间，最大值放在右边，这样以中值为界形成了两部分，要注意这时候还没有排序，只是进行了枢纽元的选取。中间值就是枢纽元！

然后以枢纽元为中心，分别交换左右两侧的数据，把大的数据放在枢纽元的右侧，把小的数据放在枢纽元的左侧，最终形成大致排序的两组数据，枢纽元排好序，然后递归调用快速排

序。

同时，为了移动数据方便，我们把枢纽元的位置放在 right-1 的位置。

大家可能要问了:为什么把枢纽元放在 right-1 的位置呢？

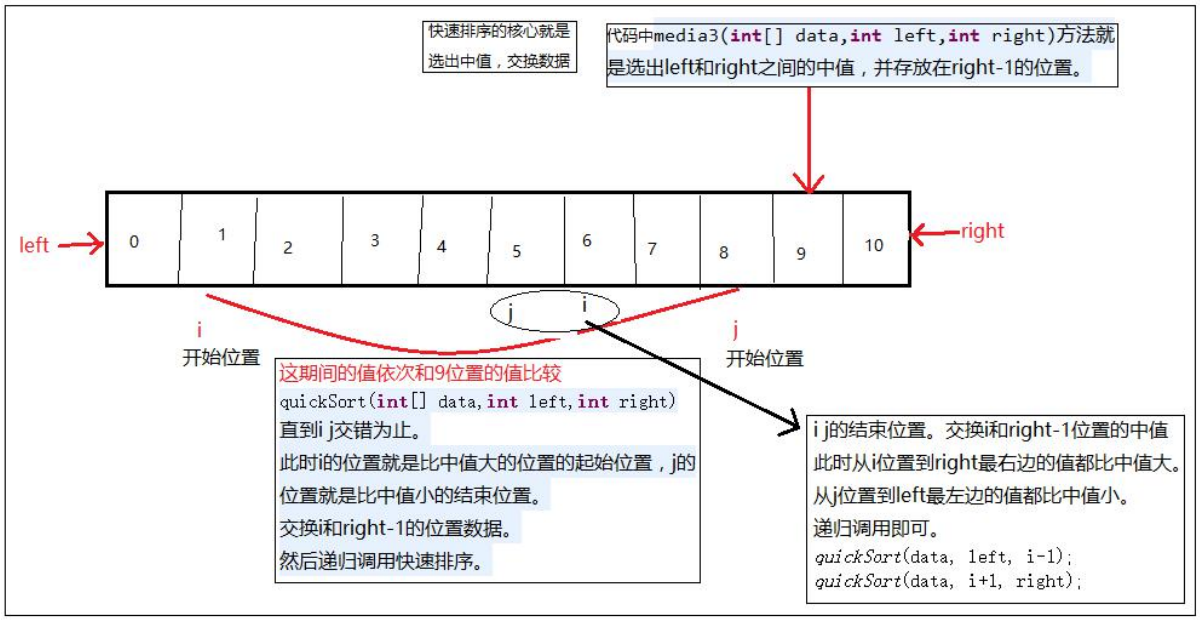
原因是：right 的位置放置的是比枢纽元大的数，在选取枢纽元的时候，我们把大的数放在 right 的位置，最小的数放在 left 的位置，把枢纽元放在 right-1 的位置，这样当完成数据交换之后枢纽元只需一次交换。如果把枢纽元放在中间的位置，要知道的是，中间位置并不一定就是枢纽元要排序的位置。这个地方要搞清楚，还得看代码：

```
public class QuickSort {
    /**
     * 快速排序
     * 首先找到三数中值，然后分别移动左右两边的数据，
     * 以中值数分割成两组，一组比中值数大，一组比中值数小。
     * 然后递归快排两组数组。
     * 当待排序的数组小于 CUTOFF 时，使用插入排序。
     */
    public static void main(String[] args) {
        quickSort(Datas.data,0,Datas.data.length-1);
        Datas.prints("快速排序");
    }
    public static void quickSort(int[] data,int left,int ri
ght){
```

```
int CUTOFF = 1;
if (left+CUTOFF<right) {
    //找到中值数
    int media = media3(data, left, right);
    //保存左右界，left,right 值不变
    int i =left;
    int j = right-1;
    //循环移动左右两边的元素
    while (true) {
        while(data[++i]<media);
        while (data[--j]>media);
        if (i>j) {
            break;
        }
        swap(data, i, j);
    }
    //将中值数移动到 i 处。中值数即排在 i 处。
    swap(data, i, right-1);
    //递归排序中值数两边的数据
    quickSort(data, left, i-1);
    quickSort(data, i+1, right);
}else {
```

```
            /**
             * 插入排序
             * 当待排序的元素少于 20 个时候，
             * 快速排序性能不如直接插入排序好。
             * 所以 else 语句里面，在待排序基本有序的情况下
             * 可以使用直接插入排序更好。
             */
            InsertSort.main(null);
        }
    }
    //找到中值数
    public static int media3(int[] data,int left,int right)
{
        int center = (left+right)/2;
        /**
         * 前两个 if 语句的比较，
         * 使得最小值放在最左边。
         */
        if (data[center]<data[left]) {
            swap(data, center, left);
```

```
        }
        if (data[right]<data[left]) {
            swap(data, right, left);
        }
        /**
         * 第三个 if 语句使得最大值放在最右边。
         * 中间值，放在中间位置。
         */
        if (data[right]<data[center]) {
            swap(data, right, center);
        }
        //把中间的位置放在 right-1 的位置。
        swap(data, center, right-1);
        return data[right-1];
    }
    //交换数据
    public static void swap(int[] data,int i,int j){
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
}
```



快速排序说明图，示意了 i、j 游标的移动位置。结合程序应该能看懂。

不过值得大家注意的是，程序中不仅仅使用快速排序的思路，而在最后，当 left 与 right 的差值在 CUTOFF 的时候，直接使用直接插入排序，不再使用快速排序。原因我在注释中已经给出。

如果不使用 CUTOFF 时候的插入排序，最终的结果并不是我们想要的。如果仅仅使用快速排序得到最终结果，则代码是不正确的。

上面的代码必须在最后使用一次插入排序才能得到最终的结果。

8 基数排序

桶排序之前不了解，我看的《数据结构与算法分析》一书中并没有给出大量的讲解，反而代码是通过例题的形式给出的。桶排序其实就是形成大的容器，通过比较数据各个位上的数进行排序。

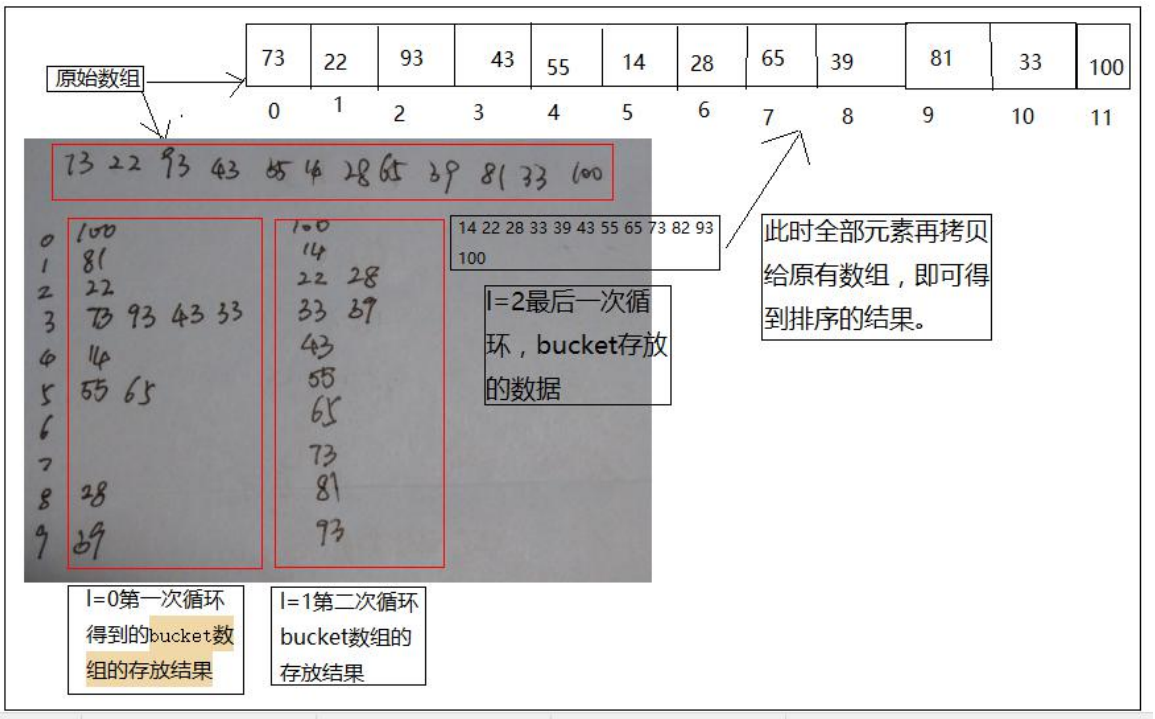
别的不多说了，直接上代码：

```
public class RadixSort {

    /**

     * 基数排序
```

```
     * 二维数组构成桶

     * 一维数组记录每个位存放的个数。

     * 每次构建桶完成，拷贝数据到原来的数组中去。

     * 继续下一轮桶的构建。

     * 分别个位，十位，百位。。。

     * 程序必须知道最大值的位数。

     */
    public static void main(String[] args) {
        radixSort(Datas.data,3);

        Datas.prints("基数排序");

    }
    public static void radixSort(int[] data,int maxLen){

        //maxLen 表示最大值的长度

        //LSD 最低位优先排序  MSD 最高位优先排序    1 从 0 开始 循环三
次

        int k = 0;

        int n = 1;

        int[][] bucket = new int[10][data.length];//桶

        /**
```

```
 * 表示桶的每一行也就是每一位存放的个数
 */
int[] orders = new int[10];
int temp = 0;
for (int l = 0; l < maxLen; l++) {
    for (int i = 0; i < data.length; i++) {
        temp = (data[i]/n)%10;
        bucket[temp][orders[temp]] = data[i];
        orders[temp]++;
    }
    //将桶中的数值保存会原来的数组中
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < orders[i]; j++) {
            if (orders[i]>0) {
                data[k]=bucket[i][j];
                k++;
            }
        }
        //拷贝完成清除记录的个数，设为 0
        orders[i]=0;
    }
    //n 乘以 10 取十位  百位的数值
```

```
                n*=10;
                k=0;
                //k 值记录拷贝数据到原有数组中的位置，拷贝完成恢复 0
            }
        }
}
```



这个是实例，程序打印结果的话，不好看，只好手写大家看效果。

bucket 二维数组存放原始数据。orders 数组存放每一位数存放的原始数据的个数。外层循环每执行一次，就把数据拷贝给原来的数组。然后进行下一轮循环。分别进行个位、十位、百位、、、、的循环。这是从最低位开始排序。也有最高位开始进行的排序。

9 计数排序

这个直接上代码：

```java
public class CuntingSort {

    /**
     * 计数排序
     * 思路：构建一个与待排序中最大值相同大小的数组，
     * 该数组存放待排序数组中每个数字出现的个数。
     */
    public static void main(String[] args) {
        cunting(Datas.data, 333);
        Datas.prints("计数排序");
    }

    public static void cunting(int[] data,int max){
        int[] temp = new int[max+1];
        int[] result = new int[data.length];
        /**
         * 该循环设置初始值为 0
         */
        for (int i = 0; i < temp.length; i++) {
            temp[i]=0;
        }
        /**
```

```
         * 该 for 语句循环遍历原数组，将数组中元素出现的个数存放在

         * temp 数组中相对应的位置上。

         * temp 数组长度与最大值的长度一致。保证每个元素都有一个对应的位置。

         */
        for (int i = 0; i < data.length; i++) {
            temp[data[i]]+=1;
        }
        /**
         * 累计每个元素出现的个数。

         * 通过该循环，temp 中存放原数组中数据小于等于它的个数。

         * 也就是说此时 temp 中存放的就是对应的元素排序后，在数组中存放的位置+1。

         */
        for (int i = 1; i < temp.length; i++) {
            temp[i]=temp[i]+temp[i-1];
        }
        /**
         * 这里从小到大遍历也可以输出正确的结果，但是不是稳定的。

         * 只有从大到小输出，结果才是稳定的。
```

```
         * result 中存放排序会的结果。
         */
        for (int i = data.length-1; i>=0; i--) {
            int index = temp[data[i]];
            result[index-1]= data[i];
            temp[data[i]]--;
        }
        Datas.data = result;
    }
}
```

这个排序还真没法画图，其实这个排序相当容易理解。找出待排序数组中最大的元素，构建一个与最大元素数+1 长度的数组 temp，这样保证待排序数组中的每一个元素都能在 temp 数组中找到自己的位置，但是 temp 不是用来存放元素的，而是存放每个元素在待排序数组中出现的个数。这一步通过第二个 for 循环得到。

接着第三个 for 循环，循环遍历 temp，目的就是得到每个元素排序后存放在原有数组中的位置。大家想一想，每个位置存放自己出现的个数，那么小于自己的元素出现的个数加到一起，即可得到自己排序后数组中的存放位置。是不是真的很巧妙！！！！！
到此，基本就是该排序算法的核心啦！！！

10 桶排序

桶排序是另外一种以 O(n)或者接近 O(n)的复杂度排序的算法. 它假设输入的待排序元素是等可能的落在等间隔的值区间内.一个长度为 N 的数组使用桶排序，需要长度为 N 的辅助数组. 等间隔的区间称为桶，每个桶内落在该区间的元素. 桶排序是基数排序的一种归纳结

果。

算法的主要思想: 待排序数组 A[1...n]内的元素是随机分布在[0,1)区间内的的浮点数.辅助排序数组 B[0....n-1]的每一个元素都连接一个链表. 将 A 内每个元素乘以 N(数组规模)取底,并以此为索引插入(插入排序)数组 B 的对应位置的连表中. 最后将所有的链表依次连接起来就是排序结果.

这个过程可以简单的分步如下:

1、设置一个定量的数组当作空桶子。

2、寻访序列，并且把项目一个一个放到对应的桶子去。

3、对每个不是空的桶子进行排序。

4、从不是空的桶子里把项目再放回原来的序列中。

例如要对大小为[1..1000]范围内的 n 个整数 A[1..n]排序，可以把桶设为大小为 10 的范围，具体而言，设集合 B[1]存储[1..10]的整数，集合 B[2]存储(10..20]的整数，……集合 B[i]存储((i-1)*10, i*10]的整数，i = 1,2,..100。总共有 100 个桶。然后对 A[1..n]从头到尾扫描一遍，把每个 A[i]放入对应的桶 B[j]中。 然后再对这 100 个桶中每个桶里的数字排序，这时可用冒泡，选择，乃至快排，一般来说任何排序法都可以。最后依次输出每个桶里面的数字，且每个桶中的数字从小到大输出，这样就得到所有数字排好序的一个序列了。

```
/**
     * 桶排序算法，对 arr 进行桶排序，排序结果仍放在 arr 中
     * @param arr
     */
    public static void main(String[] args){
        bucketSort(Datas.datad);
```

```
        for (int i = 0; i < Datas.datad.length; i++) {
            System.out.println(Datas.datad[i]+",");
        }
    }
    public static void bucketSort(double arr[]){

        int n = arr.length;

        ArrayList<Double> arrList[] = new ArrayList[n];
        //把 arr 中的数均匀的的分布到[0,1)上，每个桶是一个 list，存放落在此桶上的元素
        for(int i =0;i<n;i++){
            int temp = (int) Math.floor(n*arr[i]);
            if(null==arrList[temp])
                arrList[temp] = new ArrayList<>();
            arrList[temp].add(arr[i]);
        }

        //对每个桶中的数进行插入排序
        for(int i = 0;i<n;i++){
            if(null!=arrList[i])
                insert(arrList[i]);
        }
```

```
        //把各个桶的排序结果合并
        int count = 0;

        for(int i = 0;i<n;i++){
            if(null!=arrList[i]){
                Iterator<Double> iter = arrList[i].iterator
();
                while(iter.hasNext()){
                    Double d = (Double)iter.next();
                    arr[count] = d;
                    count++;
                }
            }
        }
    }

    /**
     * 用插入排序对每个桶进行排序
     * @param list
     */
    public static void insert(ArrayList<Double> list){
        if(list.size()>1){
            for(int i =1;i<list.size();i++){
```

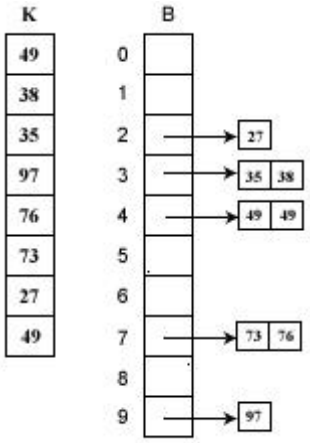
```
                if((Double)list.get(i)<(Double)list.get(i-1))
{
                    double temp = (Double) list.get(i);
                    int j = i-1;
                    for(;j>=0&&((Double)list.get(j)>(Double)l
ist.get(j+1));j--)
                        list.set(j+1, list.get(j));
                    list.set(j+1, temp);
                }
            }
        }
    }
}
```

原文地址：http://www.tuicool.com/articles/3emMVz

这个是我看到的关于桶排序说的最好的一篇文章。

举例说明：

假如待排序列 K={49、38 、35、97 、76、73 、27、49}。这些数据全部在 1—100 之间。因此我们定制 10 个桶，然后确定映射函数 f(k)=k/10。则第一个关键字 49 将定位到第 4 个桶中(49/10=4)。依次将所有关键字全部堆入桶中，并在每个非空的桶中进行快速排序后得到如下图所示：



对上图只要顺序输出每个 B[i]中的数据就可以得到有序序列了。

这个例子有点类似于基数排序。因此，可以说，桶排序是基数排序的一种。

7. 快排的 partition 函数与归并的 Merge 函数。
8. 对冒泡与快排的改进。
9. 二分查找，与变种二分查找。
10. 二叉树、B+树、AVL 树、红黑树、哈夫曼树。
11. 二叉树的前中后续遍历：递归与非递归写法，层序遍历算法。
12. 图的 BFS 与 DFS 算法，最小生成树 prim 算法与最短路径 Dijkstra 算法。
13. KMP 算法。
14. 排列组合问题。
15. 动态规划、贪心算法、分治算法。（一般不会问到）
16. 大数据处理：类似 10 亿条数据找出最大的 1000 个数.....等等