Java创建对象的5种方式

<https://www.cnblogs.com/wxd0108/p/5685817.html>

【数】1. 数组跟链表有什么区别？

1）存储空间

数组存栈，链表放堆。

2）开辟模式

数组需要预先开辟一块空间。如果使用不满会有浪费。对动态数组而言，存放数据超出空间会另外申请开辟。

链表属于动态存储，内存不一定连续。

3）增删改查

查询而言数组更快，下标索引。增删来说链表效率高些，因为只需要操作一下指针。

【算】2. 你知道的常用排序算法里面，有哪些是比较快的？复杂度大概是多少？

最近正好有看排序，就答的快排和归并。效率都趋近于N\*logN。我记得面试官补充了一下，问的是快排的最好情况和最坏情况，当时有点慌，因为没有仔细想过，模棱两可说的是最好应该会趋近于N最坏也不会差过N方。

其实对快排而言，最好的情况就是每次partition操作的元素正好插在整个子序列的中间，这样是最好的，因此最好效率是N\*logN。最坏情况当然就是往两极插了，N方。

还是得多思考各种排序的复杂度情况，以及其是否稳定，为什么？这些应该都是面试官喜欢问的。

【数】3. Hash表的实现方式是怎么样的？查询和插入复杂度都是多少？附加：如果插入的时候出现冲突，解决方案？

Hash表应该也算数据结构里面一个比较基础的类型了。

大致的实现为，Hash表会根据Hash函数为每个value计算一个hash值，然后用这个hash值/表空间大小，余数就是当前对象放的位置了。这个时候问题就来了，满了咋办？位置上有别的东西了咋办呢…这就需要冲突解决方案了。一般来说有：线性探测（比如0上有元素了，我往后挪1个看看有没有，往后挪2个有没有之类的），跳跃型探测（左右闪躲啊有木有，比如+11，-11，+21，-21……），拉链法（等于变成了个二维结构）。

查询插入复杂度都是1。一步到位。

【网】4. TCP/IP 5层结构都知道吗？每一层的职能呢？

1）应用层

负责应用间通讯，包含了很多协议，比如HTTP、HTTPS、SIP之类的。

2）传输层

传输层包含协议常考的有两个，TCP和UDP。为上一层提供的是端到端的数据传输（因为要端口号了哦亲）。TCP连接时的3次握手和断开时的4次挥手需要理解。UDP的话属于不太负责任的那种，不保证完整收货的。

3）网络层

这层主要就是做路由的。什么概念呢，我这个包从我这来要到哪去，怎么找路都是这层负责的。不同传输速率的路由器使用的算法不一致。这层可以理解是主机到主机之间的通信。

4）数据链路层

数据链路层主要管的是包在信道上发送时的一些问题，比如物理地址寻址、数据的成帧、流量控制、数据的检错、重发等等。

5）物理层

这层主要是做信号转换的。怎么把01字节转为电信号。算法有很多，速率不同算法也不一样。

【网】5. HTTP和HTTPS的基本概念？具体工作流程？如何构造安全的HTTP？（因为4.2我就没答上来，面试官小哥哥很委婉的表示，想问你这个你也不懂啊…）

根据上文所说，HTTP是工作在应用层的协议。全名超文本传输协议，通过TCP建立双端的连接，包括了客户端的请求和服务端的响应。HTTP是不安全的，它把用户请求封装成包，但是包里面的东西都是可见的（大神亲自演示抓包之后，看到了我校校园网明文传输的校园网账户登录信息…）。HTTPS多出来的那个S，多的是一个认证和加密的过程。认证，指的是“这个数据包是从正确的人手里出发的”，加密，指的是“这个数据包没有被修改过”。基于SSL进行的，现在演化到TLS啦。

【基】6. +load 跟 +initialize的区别呢？附加：+load是否需要调用super？为什么？

这个偏基础了。+load是在程序启动时加载所有类的时候执行的。 +initialize可以理解成一种懒加载，实例化某个类的时候进行。

+load不需要调用super，因为它是迭代进行的，先调用父类，再调用子类。

【基】7. category的用处？实例题：如果有个NSObject(Dog)并实现了一个run方法，直接调用[NSObject run]是否有效？为什么？如果有效的话，方法的查询/调用流程是怎么样的？

category可以修改现有类的犯法，不能添加实例变量。（protocol可以实现，但是没有使用runtime实现来的实际。）

第二个呢是有效的。从runtime的角度来说，在执行这个方法的时候，会从NSObject的分类中优先寻找名为run的方法，如果有就会执行，没有的话找本类，最后向父类进行寻找。

【基】8. mateclass…（这个就有点丢人了。因为7的缘故发散出来的题。以后细说）

这个是因为当时我回答面试官的时候只说了会先找分类（如果分类有重名方法，执行最后编译的是哪个分类的），再在本类找。因为分类中有run所以能执行（因为脑子里，NSObject上面没东西了呀，不会找父类了吧）。

面试官指正说，其实每个类都会有个metaclass，这个metaclass会指向这个类的父类。顶类的metaclass指向自己。虽然NSObject处于顶端了，但还是可以“寻找”父类的，尽管是自己。

【基】9. \_\_block的用处？

block基础。普通变量在block里面是不能改变值的，加了\_\_block前缀之后在block里面能进行值改变。原因是将变量从栈区转到堆区（这个我不是很确定），在\_\_block后，变量的内存地址是有发生改变的，童鞋们可以亲测一下哦。

【基】10. 是否能给编译后的类添加属性？运行时的呢？（runtime基础知识）

这个就不细说了。runtime基本概念。

【基】11. 如果要使用runtime添加实例属性，往里传入的参数都有哪些？

这个只要思考我们@property的时候都有啥就明白了。名称，类型，参数权限（readwrite啊，nonatomic啊，strong啊之类的）和持有这个变量的类的类名。

面试官提醒：如果要更严谨一点，加个identifier。

这个可以详细看runtime的associate那块。

【基】12. runloop的基础知识。经典题目：timer准不准？mode都有几种？scrollView滑动的时候timer失效吗？怎么处理？

这个也是偏基础的。timer不是一定准的，因为受到runloop的影响。

Mode一共5种，API提供给用户的两种，DefaultMode和CommonMode。UIInitializationRunLoopMode这个mode在第一次启动后就不用了。

滑动的时候是失效的，解决方法就是讲timer加入CommonMode。

衍生：为什么加入CommonMode就能解决？说实话之前这个我没思考过，回答的是CommonMode和TrackingMode优先级一样？面试官说不对…回来查了查，CommonMode其实是一个占位Mode，把timer加入CommonMode后能同时支持DefaultMode和TrackingMode，这样就能避免scrollView对timer的影响了。

然后因为我昨天跟面试官沟通的时候问了是否需要带电脑，所以今天有手写代码的题目。

一共三道，第一道忘记了，因为面试官小哥哥说不用写了。

【基】13. 写个OC的结构体吧…（丢人丢大发了…快1年没碰OC了，之前也是CV大法的多…就写了一半…写到一半的时候，面试官小哥哥嘀咕了一句，这个题是考你的基础，我：…T . T）

这个不写了吧。其实蛮简单的，当时也大致写对了，就是类型和名称写反了。有点紧张。

【算】14. 给出两个有序数组，如何归并到一个？（这个也就是归并的主函数实现，挺简单的，不用IDE也是一下能写出来。倒是被问了如何能增强健壮性，各位可以考虑一下）

代码不上了，就是归并的排序那块。健壮性，童鞋们需要考虑的不仅仅是list1和list2是否为空（都为空，一个为空和都不为空）的情况，还需要考虑，list1和list2是否真的是有序数组？面试官小哥哥的原话是，这个是经验问题，因为你作为API的提供者，需要考虑别人会不会骗你。往往给你的都不是一个有序数组！这个时候你需要做一下排序或者返还一个异常。其实我当时想…那我还得判断有没有英文字符啥的诶…毕竟OC数组里你给我个啥我还真不知道…（面试官小哥哥会问我这个，大概是因为我简历写了我有1年多的实习经历，虽然当时公司的移动组没有，我们是第一批入驻的- - 等于自学… 其实平时开发的时候对字段的处理还是满全面的，特别是对json回来的，判空，判类型，判规范都有，自己工具类里面也有。还是因为1年多没开发了，生疏了。）

String 和 new String()的区别：

**==是比较地址，equals()比较对象内容。**

记住一点：栈区存引用和基本类型，不能存对象，而堆区存对象。==是比较地址，equals()比较对象内容。

String str1 = "abcd"的实现过程：首先栈区创建str引用，然后在String池（独立于栈和堆而存在，存储不可变量）中寻找其指向的内容为"abcd"的对象，如果String池中没有，则创建一个，然后str指向String池中的对象，如果有，则直接将str1指向"abcd""；如果后来又定义了字符串变量 str2 = "abcd",则直接将str2引用指向String池中已经存在的“abcd”，不再重新创建对象；当str1进行了赋值（str1=“abc”），则str1将不再指向"abcd"，而是重新指String池中的"abc"，此时如果定义String str3 = "abc",进行str1 == str3操作，返回值为true，因为他们的值一样，地址一样，但是如果内容为"abc"的str1进行了字符串的+连接str1 = str1+"d"；此时str1指向的是在堆中新建的内容为"abcd"的对象，即此时进行str1==str2，返回值false，因为地址不一样。

String str3 = new String("abcd")的实现过程：直接在堆中创建对象。如果后来又有String str4 = new String("abcd")，str4不会指向之前的对象，而是重新创建一个对象并指向它，所以如果此时进行str3==str4返回值是false，因为两个对象的地址不一样，如果是str3.equals(str4)，返回true,因为内容相同。

1. **String和StringBuffer的区别**

String最慢的原因：

　　String为字符串常量，而StringBuilder和StringBuffer均为字符串变量，即String对象一旦创建之后该对象是不可更改的，但后两者的对象是变量，是可以更改的。以下面一段代码为例：

String str="abc";

System.out.println(str);

str=str+"de";

System.out.println(str);

如果运行这段代码会发现先输出“abc”，然后又输出“abcde”，好像是str这个对象被更改了，其实，这只是一种假象罢了，JVM对于这几行代码是这样处理的，首先创建一个String对象str，并把“abc”赋值给str，然后在第三行中，其实JVM又创建了一个新的对象也名为str，然后再把原来的str的值和“de”加起来再赋值给新的str，而原来的str就会被JVM的垃圾回收机制（GC）给回收掉了，所以，str实际上并没有被更改，也就是前面说的String对象一旦创建之后就不可更改了。所以，Java中对String对象进行的操作实际上是一个不断创建新的对象并且将旧的对象回收的一个过程，所以执行速度很慢。

　　而StringBuilder和StringBuffer的对象是变量，对变量进行操作就是直接对该对象进行更改，而不进行创建和回收的操作，所以速度要比String快很多。

　　另外，有时候我们会这样对字符串进行赋值

String str="abc"+"de";

StringBuilder stringBuilder=new StringBuilder().append("abc").append("de");

System.out.println(str);

System.out.println(stringBuilder.toString());

　　这样输出结果也是“abcde”和“abcde”，但是String的速度却比StringBuilder的反应速度要快很多，这是因为第1行中的操作和

　String str="abcde";

　是完全一样的，所以会很快，而如果写成下面这种形式

String str1="abc";

String str2="de";

String str=str1+str2;

　那么JVM就会像上面说的那样，不断的创建、回收对象来进行这个操作了。速度就会很慢。

　2. 再来说线程安全

　在线程安全上，StringBuilder是线程不安全的，而StringBuffer是线程安全的

　如果一个StringBuffer对象在字符串缓冲区被多个线程使用时，StringBuffer中很多方法可以带有synchronized关键字，所以可以保证线程是安全的，但StringBuilder的方法则没有该关键字，所以不能保证线程安全，有可能会出现一些错误的操作。所以如果要进行的操作是多线程的，那么就要使用StringBuffer，但是在单线程的情况下，还是建议使用速度比较快的StringBuilder。

　3. 总结一下

　String：适用于少量的字符串操作的情况

　StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

　StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

在java中有3个类来负责字符的操作。

1.Character 是进行单个字符操作的，

2.String 对一串字符进行操作。不可变类。

3.StringBuffer 也是对一串字符进行操作，但是可变类。

String:

是对象不是原始类型.

为不可变对象,一旦被创建,就不能修改它的值.

对于已经存在的String对象的修改都是重新创建一个新的对象,然后把新的值保存进去.

String 是final类,即不能被继承.

StringBuffer:

是一个可变对象,当对他进行修改的时候不会像String那样重新建立对象

它只能通过构造函数来建立,

StringBuffer sb = new StringBuffer();

note:不能通过付值符号对他进行付值.

sb = "welcome to here!";//error

对象被建立以后,在内存中就会分配内存空间,并初始保存一个null.向StringBuffer

中付值的时候可以通过它的append方法.

sb.append("hello");

字符串连接操作中StringBuffer的效率要比String高:

String str = new String("welcome to ");

str += "here";

的处理步骤实际上是通过建立一个StringBuffer,让侯调用append(),最后

再将StringBuffer toSting();

这样的话String的连接操作就比StringBuffer多出了一些附加操作,当然效率上要打折扣.

并且由于String 对象是不可变对象,每次操作Sting 都会重新建立新的对象来保存新的值.

这样原来的对象就没用了,就要被垃圾回收.这也是要影响性能的.

**Ps**：效率：StringBuilder>StringBuffer>String，所以，如果这是考虑单线程程序的话，用StringBuilder，如果涉及到多线程的，那只能是StringBuffer，具体业务具体选择，要具体了解请查看API或者查看源码！

1. **gc的概念，如果A和B对象循环引用，是否可以被GC？**

**<https://blog.csdn.net/oopsoom/article/details/40348125>**

1. **Java中的内存溢出是如何造成的**

在解决java内存溢出问题之前，需要对jvm（java虚拟机）的内存管理有一定的认识。jvm管理的内存大致包括三种不同类型的内存区 域：Permanent Generation space（永久保存区域）、Heap space(堆区域)、Java Stacks(Java栈）。其中永久保存区域主要存放Class（类）和Meta的信息，Class第一次被Load的时候被放入PermGen space区域，Class需要存储的内容主要包括方法和静态属性。堆区域用来存放Class的实例（即对象），对象需要存储的内容主要是非静态属性。每 次用new创建一个对象实例后，对象实例存储在堆区域中，这部分空间也被jvm的垃圾回收机制管理。而Java栈跟大多数编程语言包括汇编语言的栈功能相 似，主要基本类型变量以及方法的输入输出参数。Java程序的每个线程中都有一个独立的堆栈。容易发生内存溢出问题的内存空间包括：Permanent Generation space和Heap space。

## **第一种OutOfMemoryError： PermGen space**

发生这种问题的原意是程序中使用了大量的jar或class，使java虚拟机装载类的空间不够，与Permanent Generation space有关。解决这类问题有以下两种办法：

1. 增加java虚拟机中的XX:PermSize和XX:MaxPermSize参数的大小，其中XX:PermSize是初始永久保存区域大 小，XX:MaxPermSize是最大永久保存区域大小。如针对tomcat6.0，在catalina.sh 或catalina.bat文件中一系列环境变量名说明结束处（大约在70行左右） 增加一行： JAVA\_OPTS=" -XX:PermSize=64M -XX:MaxPermSize=128m" 如果是windows服务器还可以在系统环境变量中设置。感觉用tomcat发布sprint+struts+hibernate架构的程序时很容易发生这种内存溢出错误。使用上述方法，我成功解决了部署ssh项目的tomcat服务器经常宕机的问题。
2. 清理应用程序中web-inf/lib下的jar，如果tomcat部署了多个应用，很多应用都使用了相同的jar，可以将共同的jar移到 tomcat共同的lib下，减少类的重复加载。这种方法是网上部分人推荐的，我没试过，但感觉减少不了太大的空间，最靠谱的还是第一种方法。

## **第二种OutOfMemoryError：  Java heap space**

发生这种问题的原因是java虚拟机创建的对象太多，在进行垃圾回收之间，虚拟机分配的到堆内存空间已经用满了，与Heap space有关。解决这类问题有两种思路：

1. 检查程序，看是否有死循环或不必要地重复创建大量对象。找到原因后，修改程序和算法。 我以前写一个使用K-Means文本聚类算法对几万条文本记录（每条记录的特征向量大约10来个）进行文本聚类时，由于程序细节上有问题，就导致了 Java heap space的内存溢出问题，后来通过修改程序得到了解决。
2. 增加Java虚拟机中Xms（初始堆大小）和Xmx（最大堆大小）参数的大小。如：set JAVA\_OPTS= -Xms256m -Xmx1024m

## **第三种OutOfMemoryError：unable to create new native thread**

在java应用中，有时候会出现这样的错误：OutOfMemoryError: unable to create new native thread.这种怪事是因为JVM已经被系统分配了大量的内存(比如1.5G)，并且它至少要占用可用内存的一半。有人发现，在线程个数很多的情况下， 你分配给JVM的内存越多，那么，上述错误发生的可能性就越大。

那么是什么原因造成这种问题呢？

每一个32位的进程最多可以使用2G的可用内存，因为另外2G被操作系统保留。这里假设使用1.5G给JVM，那么还余下500M可用内存。这 500M内存中的一部分必须用于系统dll的加载，那么真正剩下的也许只有400M，现在关键的地方出现了：当你使用Java创建一个线程，在JVM的内 存里也会创建一个Thread对象，但是同时也会在操作系统里创建一个真正的物理线程(参考JVM规范)，操作系统会在余下的400兆内存里创建这个物理 线程，而不是在JVM的1500M的内存堆里创建。在jdk1.4里头，默认的栈大小是256KB，但是在jdk1.5里头，默认的栈大小为1M每线程， 因此，在余下400M的可用内存里边我们最多也只能创建400个可用线程。

这样结论就出来了，要想创建更多的线程，你必须减少分配给JVM的最大内存。还有一种做法是让JVM宿主在你的JNI代码里边。

给出一个有关能够创建线程的最大个数的估算公式：

(MaxProcessMemory - JVMMemory - ReservedOsMemory) / (ThreadStackSize) = Number of threads

对于jdk1.5而言，假设操作系统保留120M内存：

1.5GB JVM: (2GB-1.5Gb-120MB)/(1MB) = ~380 threads1.0GB JVM: (2GB-1.0Gb-120MB)/(1MB) = ~880 threads

对于栈大小为256KB的jdk1.4而言，

1.5GB allocated to JVM: ~1520 threads1.0GB allocated to JVM: ~3520 threads

对于这个异常我们首先需要判断下，发生内存溢出时进程中到底都有什么样的线程，这些线程是否是应该存在的，是否可以通过优化来降低线程数； 另外一方面默认情况下java为每个线程分配的栈内存大小是1M，通常情况下，这1M的栈内存空间是足足够用了，因为在通常在栈上存放的只是基础类型的数 据或者对象的引用，这些东西都不会占据太大的内存， 我们可以通过调整jvm参数，降低为每个线程分配的栈内存大小来解决问题，例如在jvm参数中添加-Xss128k将线程栈内存大小设置为128k。

1. **String s = “123”;这个语句有几个对象产生**

String str=new String("123"); 紧接着这段代码之后的往往是这个问题，那就是这行代码究竟创建了几个String对象呢？

相信大家对这道题并不陌生，答案也是众所周知的，2个。

接下来我们就从这道题展开，一起回顾一下与创建String对象相关的一些JAVA知识。

我们可以把上面这行代码分成String str、=、"123"和new String()四部分来看待。String str只是定义了一个名为str的String类型的变量，因此它并没有创建对象；=是对变量str进行初始化，将某个对象的引用（或者叫句柄）赋值给它，显然也没有创建对象；现在只剩下new String("123")了。那么，new String("123")为什么又能被看成"123"和new String()呢？

我们来看一下被我们调用了的String的构造器：

public String(String original) { //other code ... } 大家都知道，我们常用的创建一个类的实例（对象）的方法有以下两种:

一、使用new创建对象。

二、调用Class类的newInstance方法，利用反射机制创建对象。

我们正是使用new调用了String类的上面那个构造器方法创建了一个对象，并将它的引用赋值给了str变量。同时我们注意到，被调用的构造器方法接受的参数也是一个String对象，这个对象正是"123"。由此我们又要引入另外一种创建String对象的方式的讨论——引号内包含文本。

这种方式是String特有的，并且它与new的方式存在很大区别。

String str="123";

毫无疑问，这行代码创建了一个String对象。

String a="123"; String b="123"; 那这里呢？

答案还是一个。

String a="123"+"456"; 再看看这里呢？

答案是三个。

说到这里，我们就需要引入对字符串池相关知识的回顾了。

在JAVA虚拟机（JVM）中存在着一个字符串池，其中保存着很多String对象，并且可以被共享使用，因此它提高了效率。由于String类是final的，它的值一经创建就不可改变，因此我们不用担心String对象共享而带来程序的混乱。字符串池由String类维护，我们可以调用intern()方法来访问字符串池。

我们再回头看看String a="123";，这行代码被执行的时候，JAVA虚拟机首先在字符串池中查找是否已经存在了值为"123"的这么一个对象，它的判断依据是String类equals(Object obj)方法的返回值。如果有，则不再创建新的对象，直接返回已存在对象的引用；如果没有，则先创建这个对象，然后把它加入到字符串池中，再将它的引用返回。因此，我们不难理解前面三个例子中头两个例子为什么是这个答案了。

只有使用引号包含文本的方式创建的String对象之间使用“+”连接产生的新对象才会被加入字符串池中。对于所有包含new方式新建对象（包括null）的“+”连接表达式，它所产生的新对象都不会被加入字符串池中，对此我们不再赘述。因此我们提倡大家用引号包含文本的方式来创建String对象以提高效率，实际上这也是我们在编程中常采用的。

栈（stack）：主要保存基本类型（或者叫内置类型）（char、byte、short、int、long、float、double、boolean）和对象的引用，数据可以共享，速度仅次于寄存器（register），快于堆。

堆（heap）：用于存储对象

1. **Error、Exception和RuntimeException的区别，作用又是什么？**

Error（错误）表示系统级的错误和程序不必处理的异常，是java运行环境中的内部错误或者硬件问题。比如：内存资源不足等。对于这种错误，程序基本无能为力，除了退出运行外别无选择，它是由Java虚拟机抛出的。

Exception（违例）表示需要捕捉或者需要程序进行处理的异常，它处理的是因为程序设计的瑕疵而引起的问题或者在外的输入等引起的一般性问题，是程序必须处理的。

Exception又分为运行时异常，受检查异常。

运行时异常，表示无法让程序恢复的异常，导致的原因通常是因为执行了错误的操作，建议终止程序，因此，编译器不检查这些异常。

受检查异常，是表示程序可以处理的异常，也即表示程序可以修复（由程序自己接受异常并且做出处理）， 所以称之为受检查异常。

1. 异常机制

异常机制是指当程序出现错误后，程序如何处理。具体来说，异常机制提供了程序退出的安全通道。当出现错误后，程序执行的流程发生改变，程序的控制权转移到异常处理器。

传统的处理异常的办法是，函数返回一个特殊的结果来表示出现异常（通常这个特殊结果是大家约定俗称的），调用该函数的程序负责检查并分析函数返回的结果。这样做有如下的弊端：例如函数返回-1代表出现异常，但是如果函数确实要返回-1这个正确的值时就会出现混淆；可读性降低，将程序代码与处理异常的代码混爹在一起；由调用函数的程序来分析错误，这就要求客户程序员对库函数有很深的了解。

异常处理的流程：

① 遇到错误，方法立即结束，并不返回一个值；同时，抛出一个异常对象 。

② 调用该方法的程序也不会继续执行下去，而是搜索一个可以处理该异常的异常处理器，并执行其中的代码 。

2 异常的分类

异常的分类：

① 异常的继承结构：基类为Throwable，Error和Exception继承Throwable，RuntimeException和IOException等继承Exception，具体的RuntimeException继承RuntimeException。

② Error和RuntimeException及其子类成为未检查异常（unchecked），其它异常成为已检查异常（checked）。

每个类型的异常的特点

Error体系 ：

Error类体系描述了Java运行系统中的内部错误以及资源耗尽的情形。应用程序不应该抛出这种类型的对象（一般是由虚拟机抛出）。如果出现这种错误，除了尽力使程序安全退出外，在其他方面是无能为力的。所以，在进行程序设计时，应该更关注Exception体系。

Exception体系包括RuntimeException体系和其他非RuntimeException的体系：

① RuntimeException：RuntimeException体系包括错误的类型转换、数组越界访问和试图访问空指针等等。处理RuntimeException的原则是：如果出现RuntimeException，那么一定是程序员的错误。例如，可以通过检查数组下标和数组边界来避免数组越界访问异常。

②其他非RuntimeException（IOException等等）：这类异常一般是外部错误，例如试图从文件尾后读取数据等，这并不是程序本身的错误，而是在应用环境中出现的外部错误。

1. **列举3个以上的RuntimeException**

总结了一下JAVA中常见的几种RuntimeException,大约有如下几种：

NullPointerException - 空指针引用异常

ClassCastException - 类型强制转换异常。

IllegalArgumentException - 传递非法参数异常。

ArithmeticException - 算术运算异常

ArrayStoreException - 向数组中存放与声明类型不兼容对象异常

IndexOutOfBoundsException - 下标越界异常

NegativeArraySizeException - 创建一个大小为负数的数组错误异常

NumberFormatException - 数字格式异常

SecurityException - 安全异常

UnsupportedOperationException - 不支持的操作异常

常见的RuntimeException- -

RuntimeException是开发中最容易遇到的，下面列举一下常见的RuntimeException：

1、NullPointerException：见的最多了，其实很简单，一般都是在null对象上调用方法了。

String s=null;

boolean eq=s.equals(""); // NullPointerException

这里你看的非常明白了，为什么一到程序中就晕呢？

public int getNumber(String str){

　　if(str.equals("A")) return 1;

　　 else if(str.equals("B")) return 2;

}

这个方法就有可能抛出NullPointerException,我建议你主动抛出异常，因为代码一多，你可能又晕了。

public int getNumber(String str){

　　if(str==null) throw new NullPointerException("参数不能为空");

//你是否觉得明白多了

　　if(str.equals("A")) return 1;

　　 else if(str.equals("B")) return 2;

}

2、NumberFormatException：继承IllegalArgumentException，字符串转换为数字时出现。比如int i= Integer.parseInt("ab3");

3、ArrayIndexOutOfBoundsException:数组越界。比如 int[] a=new int[3]; int b=a[3];

4、StringIndexOutOfBoundsException：字符串越界。比如 String s="hello"; char c=s.chatAt(6);

5、ClassCastException:类型转换错误。比如 Object obj=new Object(); String s=(String)obj;

6、UnsupportedOperationException:该操作不被支持。如果我们希望不支持这个方法，可以抛出这个异常。既然不支持还要这个干吗？有可能子类中不想支持父类中有的方法，可以直接抛出这个异常。

7、ArithmeticException：算术错误，典型的就是0作为除数的时候。

8、IllegalArgumentException：非法参数，在把字符串转换成数字的时候经常出现的一个异常，我们可以在自己的程序中好好利用这个异常。

我们可创建一个控制器，令其捕获所有类型的违例。具体的做法是捕获基础类违例类型Exception（也存在其他类型的基础违例，但Exception是适用于几乎所有编程活动的基础）。如下所示：

　　catch(Exception e) {

　　System.out.println("caught an exception");

　　}

　　这段代码能捕获任何违例，所以在实际使用时最好将其置于控制器列表的末尾，防止跟随在后面的任何特殊违例控制器失效。

　　对于程序员常用的所有违例类来说，由于Exception类是它们的基础，所以我们不会获得关于违例太多的信息，但可调用来自它的基础类Throwable的方法：

　　String getMessage()

　　获得详细的消息。

　　String toString()

　　返回对Throwable的一段简要说明，其中包括详细的消息（如果有的话）。

　　void printStackTrace()

　　void printStackTrace(PrintStream)

　　打印出Throwable和Throwable的调用堆栈路径。调用堆栈显示出将我们带到违例发生地点的方法调用的顺序。

　　第一个版本会打印出标准错误，第二个则打印出我们的选择流程。若在Windows下工作，就不能重定向标准错误。因此，我们一般愿意使用第二个版本，并将结果送给System.out；这样一来，输出就可重定向到我们希望的任何路径。

　　除此以外，我们还可从Throwable的基础类Object（所有对象的基础类型）获得另外一些方法。对于违例控制来说，其中一个可能有用的是getClass()，它的作用是返回一个对象，用它代表这个对象的类。我们可依次用getName()或toString()查询这个Class类的名字。亦可对Class对象进行一些复杂的操作，尽管那些操作在违例控制中是不必要的。本章稍后还会详细讲述Class对象。

　　下面是一个特殊的例子，它展示了Exception方法的使用（若执行该程序遇到困难，请参考第3章3.1.2小节“赋值”）：

　　//: ExceptionMethods.Java

　　// Demonstrating the Exception Methods

　　package c09;

　　public class ExceptionMethods {

　　　public static void main(String[] args) {

　　　　try {

　　　　　throw new Exception("Here's my Exception");

　　　　} catch(Exception e) {

　　　　　System.out.println("Caught Exception");

　　　　　System.out.println(

　　　　　　"e.getMessage(): " + e.getMessage());

　　　　　System.out.println(

　　　　　　"e.toString(): " + e.toString());

　　　　　System.out.println("e.printStackTrace():");

　　　　　e.printStackTrace();

　　　　}

　　　}

　　} ///:~

　　该程序输出如下：

　　Caught Exception

　　e.getMessage(): Here's my Exception

　　e.toString(): java.lang.Exception: Here's my Exception

　　e.printStackTrace():

　　java.lang.Exception: Here's my Exception

　　　　　　at ExceptionMethods.main

　　可以看到，该方法连续提供了大量信息——每类信息都是前一类信息的一个子集。

本章的第一个例子是：

　　if(t == null)

　　throw new NullPointerException();

　　看起来似乎在传递进入一个方法的每个句柄中都必须检查null（因为不知道调用者是否已传递了一个有效的句柄），这无疑是相当可怕的。但幸运的是，我们根本不必这样做——它属于Java进行的标准运行期检查的一部分。若对一个空句柄发出了调用，Java会自动产生一个NullPointerException违例。所以上述代码在任何情况下都是多余的。

　　这个类别里含有一系列违例类型。它们全部由Java自动生成，毋需我们亲自动手把它们包含到自己的违例规范里。最方便的是，通过将它们置入单独一个名为RuntimeException的基础类下面，它们全部组合到一起。这是一个很好的继承例子：它建立了一系列具有某种共通性的类型，都具有某些共通的特征与行为。此外，我们没必要专门写一个违例规范，指出一个方法可能会“掷”出一个RuntimeException，因为已经假定可能出现那种情况。由于它们用于指出编程中的错误，所以几乎永远不必专门捕获一个“运行期违例”——RuntimeException——它在默认情况下会自动得到处理。若必须检查RuntimeException，我们的代码就会变得相当繁复。在我们自己的包里，可选择“掷”出一部分RuntimeException。

　　如果不捕获这些违例，又会出现什么情况呢？由于编译器并不强制违例规范捕获它们，所以假如不捕获的话，一个RuntimeException可能过滤掉我们到达main()方法的所有途径。为体会此时发生的事情，请试试下面这个例子：

　　//: NeverCaught.java

　　// Ignoring RuntimeExceptions

　　public class NeverCaught {

　　　static void f() {

　　　　throw new RuntimeException("From f()");

　　　}

　　　static void g() {

　　　　f();

　　　}

　　　public static void main(String[] args) {

　　　　g();

　　　}

　　} ///:~

　　大家已经看到，一个RuntimeException（或者从它继承的任何东西）属于一种特殊情况，因为编译器不要求为这些类型指定违例规范。

　　输出如下：

　　java.lang.RuntimeException: From f()

　　at NeverCaught.f(NeverCaught.java:9)

　　at NeverCaught.g(NeverCaught.java:12)

　　at NeverCaught.main(NeverCaught.java:15)

　　所以答案就是：假若一个RuntimeException获得到达main()的所有途径，同时不被捕获，那么当程序退出时，会为那个违例调用printStackTrace()。

　　注意也许能在自己的代码中仅忽略RuntimeException，因为编译器已正确实行了其他所有控制。因为RuntimeException在此时代表一个编程错误：

　　(1) 一个我们不能捕获的错误（例如，由客户程序员接收传递给自己方法的一个空句柄）。

　　(2) 作为一名程序员，一个应在自己的代码中检查的错误（如ArrayIndexOutOfBoundException，此时应注意数组的大小）。

　　可以看出，最好的做法是在这种情况下违例，因为它们有助于程序的调试。

　　另外一个有趣的地方是，我们不可将Java违例划分为单一用途的工具。的确，它们设计用于控制那些讨厌的运行期错误——由代码控制范围之外的其他力量产生。但是，它也特别有助于调试某些特殊类型的编程错误，那些是编译器侦测不到的。

覆盖一个方法时，只能产生已在方法的基础类版本中定义的违例。这是一个重要的限制，因为它意味着与基础类协同工作的代码也会自动应用于从基础类衍生的任何对象（当然，这属于基本的OOP概念），其中包括违例。

　　下面这个例子演示了强加在违例身上的限制类型（在编译期）：

　　//: StormyInning.Java

　　// Overridden methods may throw only the

　　// exceptions specified in their base-class

　　// versions, or exceptions derived from the

　　// base-class exceptions.

　　class BaseballException extends Exception {}

　　class Foul extends BaseballException {}

　　class Strike extends BaseballException {}

　　abstract class Inning {

　　　Inning() throws BaseballException {}

　　　void event () throws BaseballException {

　　　 // Doesn't actually have to throw anything

　　　}

　　　abstract void atBat() throws Strike, Foul;

　　　void walk() {} // Throws nothing

　　}

　　class StormException extends Exception {}

　　class RainedOut extends StormException {}

　　class PopFoul extends Foul {}

　　interface Storm {

　　　void event() throws RainedOut;

　　　void rainHard() throws RainedOut;

　　}

　　public class StormyInning extends Inning

　　　　implements Storm {

　　　// OK to add new exceptions for constrUCtors,

　　　// but you must deal with the base constructor

　　　// exceptions:

　　　StormyInning() throws RainedOut,

　　　　BaseballException {}

　　　StormyInning(String s) throws Foul,

　　　　BaseballException {}

　　　// Regular methods must conform to base class:

　　//! void walk() throws PopFoul {} //Compile error

　　　// Interface CANNOT add exceptions to existing

　　　// methods from the base class:

　　//! public void event() throws RainedOut {}

　　　// If the method doesn't already exist in the

　　　// base class, the exception is OK:

　　　public void rainHard() throws RainedOut {}

　　　// You can choose to not throw any exceptions,

　　　// even if base version does:

　　　public void event() {}

　　　// Overridden methods can throw

　　　// inherited exceptions:

　　　void atBat() throws PopFoul {}

　　　public static void main(String[] args) {

　　　　try {

　　　　　StormyInning si = new StormyInning();

　　　　　si.atBat();

　　　　} catch(PopFoul e) {

　　　　} catch(RainedOut e) {

　　　　} catch(BaseballException e) {}

　　　　// Strike not thrown in derived version.

　　　　try {

　　　　　// What happens if you upcast?

　　　　　Inning i = new StormyInning();

　　　　　i.atBat();

　　　　　// You must catch the exceptions from the

　　　　　// base-class version of the method:

　　　　} catch(Strike e) {

　　　　} catch(Foul e) {

　　　　} catch(RainedOut e) {

　　　　} catch(BaseballException e) {}

　　　}

　　} ///:~

　　在Inning中，可以看到无论构建器还是event()方法都指出自己会“掷”出一个违例，但它们实际上没有那样做。这是合法的，因为它允许我们强迫用户捕获可能在覆盖过的event()版本里添加的任何违例。同样的道理也适用于abstract方法，就象在atBat()里展示的那样。

　　“interface Storm”非常有趣，因为它包含了在Incoming中定义的一个方法——event()，以及不是在其中定义的一个方法。这两个方法都会“掷”出一个新的违例类型：RainedOut。当执行到“StormyInning extends”和“implements Storm”的时候，可以看到Storm中的event()方法不能改变Inning中的event()的违例接口。同样地，这种设计是十分合理的；否则的话，当我们操作基础类时，便根本无法知道自己捕获的是否正确的东西。当然，假如interface中定义的一个方法不在基础类里，比如rainHard()，它产生违例时就没什么问题。

　　对违例的限制并不适用于构建器。在StormyInning中，我们可看到一个构建器能够“掷”出它希望的任何东西，无论基础类构建器“掷”出什么。然而，由于必须坚持按某种方式调用基础类构建器（在这里，会自动调用默认构建器），所以衍生类构建器必须在自己的违例规范中声明所有基础类构建器违例。

　　StormyInning.walk()不会编译的原因是它“掷”出了一个违例，而Inning.walk()却不会“掷”出。若允许这种情况发生，就可让自己的代码调用Inning.walk()，而且它不必控制任何违例。但在以后替换从Inning衍生的一个类的对象时，违例就会“掷”出，造成代码执行的中断。通过强迫衍生类方法遵守基础类方法的违例规范，对象的替换可保持连贯性。

　　覆盖过的event()方法向我们显示出一个方法的衍生类版本可以不产生任何违例——即便基础类版本要产生违例。同样地，这样做是必要的，因为它不会中断那些已假定基础类版本会产生违例的代码。差不多的道理亦适用于atBat()，它会“掷”出PopFoul——从Foul衍生出来的一个违例，而Foul违例是由atBat()的基础类版本产生的。这样一来，假如有人在自己的代码里操作Inning，同时调用了atBat()，就必须捕获Foul违例。由于PopFoul是从Foul衍生的，所以违例控制器（模块）也会捕获PopFoul。

　　最后一个有趣的地方在main()内部。在这个地方，假如我们明确操作一个StormyInning对象，编译器就会强迫我们只捕获特定于那个类的违例。但假如我们上溯造型到基础类型，编译器就会强迫我们捕获针对基础类的违例。通过所有这些限制，违例控制代码的“健壮”程度获得了大幅度改善（注释③）。

　　③：ANSI/ISO C++施加了类似的限制，要求衍生方法违例与基础类方法掷出的违例相同，或者从后者衍生。在这种情况下，C++实际上能够在编译期间检查违例规范。

　　我们必须认识到这一点：尽管违例规范是由编译器在继承期间强行遵守的，但违例规范并不属于方法类型的一部分，后者仅包括了方法名以及自变量类型。因此，我们不可在违例规范的基础上覆盖方法。除此以外，尽管违例规范存在于一个方法的基础类版本中，但并不表示它必须在方法的衍生类版本中存在。这与方法的“继承”颇有不同（进行继承时，基础类中的方法也必须在衍生类中存在）。换言之，用于一个特定方法的“违例规范接口”可能在继承和覆盖时变得更“窄”，但它不会变得更“宽”——这与继承时的类接口规则是正好相反的。

RuntimeException可以由系统自动抛出，可以不进行try...catch

但如果有try,则必须有finally,可以没有catch

1. **reader和inputstream区别**

java.io下面有两个抽象类：InputStream和Reader  
InputStream是表示字节输入流的所有类的超类  
Reader是用于读取字符流的抽象类  
InputStream提供的是字节流的读取，而非文本读取，这是和Reader类的根本区别。  
即用Reader读取出来的是char数组或者String ，使用InputStream读取出来的是byte数组。  
弄清了两个超类的根本区别，再来看他们底下子类的使用，这里只对最常用的几个说明  
  
InputStream   
   | \_\_FileInputStream   
  
FileInputStream 从文件系统中的某个文件中获得输入字节。  
构造方法摘要    
FileInputStream (File  file)   
          通过打开一个到实际文件的连接来创建一个 FileInputStream ，该文件通过文件系统中的 File 对象 file 指定。   
FileInputStream (FileDescriptor  fdObj)   
          通过使用文件描述符 fdObj 创建一个 FileInputStream ，该文件描述符表示到文件系统中某个实际文件的现有连接。   
FileInputStream (String  name)   
          通过打开一个到实际文件的连接来创建一个 FileInputStream ，该文件通过文件系统中的路径名 name 指定。   
   
   
Reader  
  
   |——BufferedReader   
   |\_\_\_InputStreamReader   
         |\_\_FileReader   
  
  
BufferedReader : 从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。  
   
构造方法摘要    
BufferedReader (Reader  in)   
          创建一个使用默认大小输入缓冲区的缓冲字符输入流。   
BufferedReader (Reader  in, int sz)   
          创建一个使用指定大小输入缓冲区的缓冲字符输入流。   
BufferedReader (Java Platform SE 6)   
BufferedReader的最大特点就是缓冲区的设置。通常Reader 所作的每个读取请求都会导致对底层字符或字节流进行相应的读取请求，如果没有缓冲，则每次调用  read() 或 readLine() 都会导致从文件中读取字节，并将其转换为字符后返回，而这是极其低效的。   
使用BufferedReader可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。   
因此，建议用 BufferedReader 包装所有其 read() 操作可能开销很高的 Reader（如 FileReader 和InputStreamReader）。例如，   
 BufferedReader in  
   = new BufferedReader(new FileReader("foo.in"));  
 将缓冲指定文件的输入。   
InputStreamReader (Java Platform SE 6)   
InputStreamReader 是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。   
   
构造方法摘要    
InputStreamReader (InputStream  in)   
          创建一个使用默认字符集的 InputStreamReader。   
InputStreamReader (InputStream  in, Charset  cs)   
          创建使用给定字符集的 InputStreamReader。   
InputStreamReader (InputStream  in, CharsetDecoder  dec)   
          创建使用给定字符集解码器的 InputStreamReader。   
InputStreamReader (InputStream  in, String  charsetName)   
          创建使用指定字符集的 InputStreamReader。   
   
每次调用 InputStreamReader 中的一个 read() 方法都会导致从底层输入流读取一个或多个字节。要启用从字节到字符的有效转换，可以提前从底层流读取更多的字节，使其超过满足当前读取操作所需的字节。   
为了达到最高效率，可要考虑在 BufferedReader 内包装 InputStreamReader。例如：   
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));  
InputStreamReader最大的特点是可以指转换的定编码格式  
，这是其他类所不能的，从构造方法就可看出，  
这一点在读取中文字符时非常有用  
  
FileReader  
1）FileReader类介绍：  
InputStreamReader类的子类，所有方法（read（）等）都从父类InputStreamReader中继承而来；  
2）与InputStreamReader类的区别：  
构造方法摘要    
FileReader (File  file)   
          在给定从中读取数据的 File 的情况下创建一个新 FileReader 。   
FileReader (FileDescriptor  fd)   
          在给定从中读取数据的 FileDescriptor 的情况下创建一个新 FileReader 。   
FileReader (String  fileName)   
          在给定从中读取数据的文件名的情况下创建一个新 FileReader    
  
该类与它的父类InputStreamReader的主要不同在于构造函数，主要区别也就在于构造函数！  
从InputStreamReader的构造函数中看到，参数为InputStream和编码方式，可以看出，  
当要指定编码方式时，必须使用InputStreamReader  
类；而FileReader构造函数的参数与FileInputStream同，为File对象或表示path的String，可以看出，当要根据File对象或者String读取一个文件时，用FileReader；  
我想FileReader子类的作用也就在于这个小分工吧。该类与它的父类InputStreamReader  
的主要不同在于构造函数，主要区别也就在于构造函数！  
从InputStreamReader  
的构造函数中看到，参数为InputStream和编码方式，可以看出，  
当要指定编码方式时，必须使用InputStreamReader  
类；而FileReader构造函数的参数与FileInputStream  
同，为File对象或表示path的String，可以看出，当要根据File对象或者String读取一个文件时，用FileReader；  
我想FileReader子类的作用也就在于这个小分工吧。  
二 联系与区别   
（1）字符与字节：   
FileInputStream 类以二进制输入/输出，I/O速度快且效率搞，但是它的read（）方法读到的是一个字节（二进制数据），很不利于人们阅读，而且无法直接对文件中的字符进行操作，比如替换，查找（必须以字节形式操作）；  
而Reader类弥补了这个缺陷，可以以文本格式输入/输出，非常方便；比如可以使用while((ch = filereader.read())!=-1 )循环来读取文件；可以使用BufferedReader的readLine()方法一行一行的读取文本。  
（2）编码  
InputStreamReader ，它是字节转换为字符的桥梁。 你可以在构造器重指定编码的方式，如果不指定的话将采用底层操作系统的默认编码方式，例如GBK等。   
FileReader与InputStreamReader 涉及编码转换(指定编码方式或者采用os默认编码)，可能在不同的平台上出现乱码现象！而FileInputStream 以二进制方式处理，不会出现乱码现象.   
因此要指定编码方式时，必须使用InputStreamReader 类，所以说它是字节转换为字符的桥梁；  
(3) 缓存区  
    BufferReader类用来包装所有其 read() 操作可能开销很高的 Reader（如 FileReader 和InputStreamReader）。  
（4）规范用法  
总结以上内容，得出比较好的规范用法：   
1） File file = new File ("hello.txt");   
FileInputStream in=new FileInputStream (file);   
2） File file = new File ("hello.txt");   
FileInputStream in=new FileInputStream (file);   
InputStreamReader inReader=new InputStreamReader (in,"UTF-8");   
BufferedReader bufReader=new BufferedReader(inReader);   
3） File file = new File ("hello.txt");   
FileReader fileReader=new FileReader(file);   
BufferedReader bufReader=new BufferedReader(fileReader);

1. **hashCode的作用**

**<https://blog.csdn.net/zhangyuan19880606/article/details/51240372>**

1. **Object类中有哪些方法，列举3个以上（可以引导）**

#### 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。

#### 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，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。

#### 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。

（1）其他线程调用了该对象的notify方法。

（2）其他线程调用了该对象的notifyAll方法。

（3）其他线程调用了interrupt中断该线程。

（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。

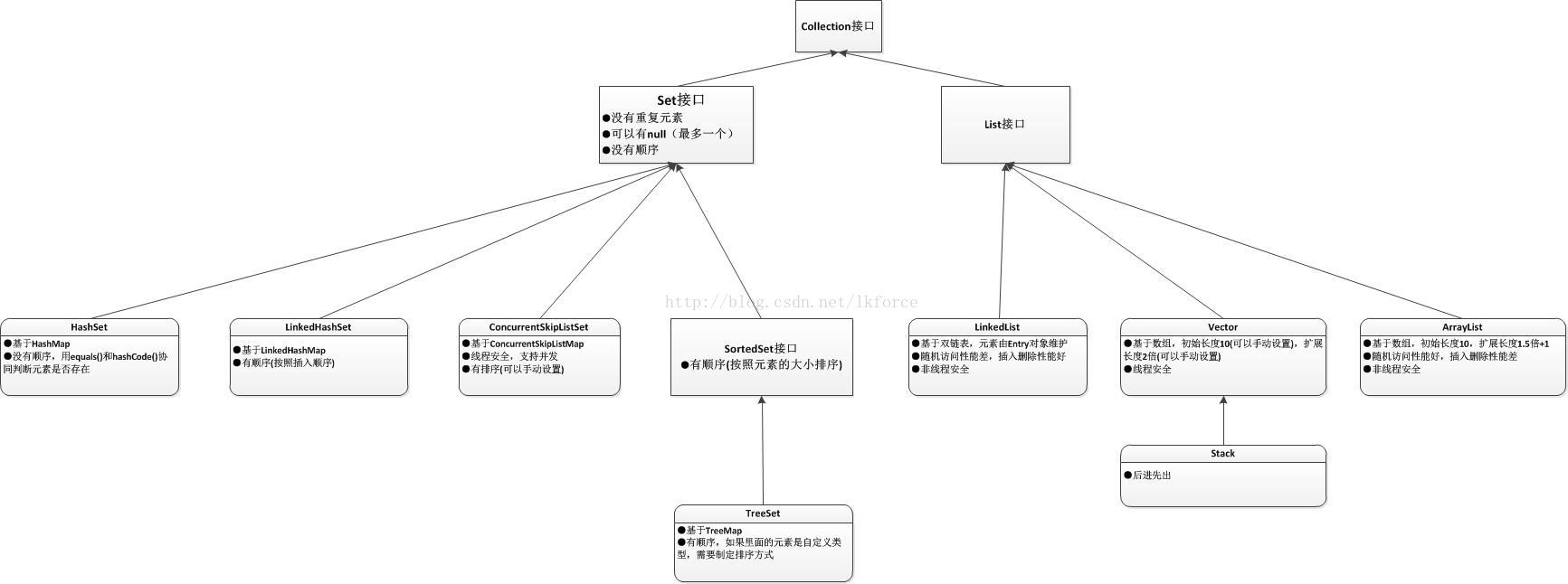
#### 9．notifyAll方法

该方法唤醒在该对象上等待的所有线程。

1. **char型变量中能不能存贮一个中文汉字?为什么?**

 \* char型变量是用来存储Unicode编码的字符的，unicode编码字符集中包含了汉字，  
 \* 所以，char型变量中当然可以存储汉字啦。不过，如果某个特殊的汉字没有被包含在  
 \* unicode编码字符集中，那么，这个char型变量中就不能存储这个特殊汉字。补充  
 \* 说明：unicode编码占用两个字节，所以，char类型的变量也是占用两个字节

1. **列举几个Java Collection类库中的常用类**



Set接口

特点：没有重复元素，可以有null（最多一个），没有顺序

HashSet

实现：基于HashMap

特点：没有顺序，用equals()和hashCode()协同判断元素是否存在

LinkedHashSet

实现：基于LinkedHashMap

特点：有顺序(按照插入顺序)

SortedSet接口

特点：有顺序(按照元素的大小排序)

TreeSet

实现：基于TreeMap

特点：排过序的，如果里面的元素是自定义类型，需要指定排序方式

ConcurrentSkipListSet

实现：基于ConcurrentSkipListMap

特点：线程安全，支持并发，有排序(可以手动设置)

List接口

LinkedList

实现：基于双链表，元素由Entry对象维护

特点：随机访问性能差，插入删除性能好，非线程安全

ArrayList

实现：基于数组，初始长度10，扩展长度1.5倍+1

特点：随机访问性能好，插入删除性能差，非线程安全

Vector

实现：基于数组，初始长度10(可以手动设置)，扩展长度2倍(可以手动设置)

特点：线程安全

Stack

特点：后进先出

**12、List、Set、Map是否都继承自Collection接口？**

**13、HashMap和Hashtable的区别**

**14、HashMap中是否任何对象都可以做为key,用户自定义对象做为key有没有什么要求？**

**15、interface 和 abstrat class的区别 是否可以继承多个接口，是否可以继承多个抽象类**

**16、 启动一个线程是用run()还是start()? 多线程有几种实现 同步和并发是如何解决的 什么叫守护线程，用什么方法实现守护线程（Thread.setDeamon()的含义） 如何停止一个线程？ 解释是一下什么是线程安全？举例说明一个线程不安全的例子。解释Synchronized关键字的作用。 当一个线程进入一个对象的一个synchronized方法后，其它线程是否可进入此对象的其它方法?**

**17、了解过哪些JDK8的新特性，举例描述下相应的特性？**

**18、对sql进行优化的原则有哪些？**

**19、servlet生命周期是生命与cgi的区别？**

**20、StringBuffer有什么优势？为什么快？**

**other**

**谈谈你对HashMap的理解，底层原理的基本实现，HashMap怎么解决碰撞问题的？**

**开发中用了那些数据库？回答mysql，储存引擎有哪些？然后问了我悲观锁和乐观锁问题使用场景、分布式集群实现的原理。**

**springmvc和mybatis的工作原理，有没有看过底层源码？**

**熟悉IO么？与NIO的区别，阻塞与非阻塞的区别**

**微信红包怎么实现。**

**海量数据分析。**

**线程安全和非线程安全。**

**HTTP2.0、thrift。**

**java反射应用**

**分布式事务一致性。**

**nio的底层实现。**

**jvm基础是必问的，jvm GC原理，JVM怎么回收内存。**

**API接口与SDI接口的区别**

**dubbo如何一条链接并发多个调用。Dubbo的原理，序列化相关问题。**

**用过哪些中间件。**

**做过工作流引擎没有。**

**以前的工作经历，自己觉得出彩的地方**

**线程池的一些原理，锁的机制升降级**

**从系统层面考虑，分布式从哪些纬度考虑**

**Hadoop底层怎么实现**

**threadLocal，线程池，hashMap/hashTable/coccurentHashMap等**

**秒杀系统的设计**

**虚拟机，IO相关知识点**

**Linux的命令**

**一个整形数组，给定一个数，在数组中找出两个数的和等于这个数，并打印出来，我写的时间复杂度高，要求O(n)。**

**n个整数，找出连续的m个数加和是最大。**

**更重视开源技术**

**数据库锁隐原理**

**1000个线程同时运行，怎么防止不卡**

**并列的并发消费问题）**

**高并发量大的话怎么处理热点，数据等**

**如何获取一个本地服务器上可用的端口**

**流量控制相关问题**

**数据库TPS是多少，是否进行测试过**

**缓存击穿有哪些方案解决**

**Java怎么挖取回收器相关原理**

**Java的集合都有哪些，都有什么特点**

**分布式锁，redis缓存，spring aop，系统架构图，MySQL的特性**

**场景，同时给10万个人发工资，怎么样设计并发方案，能确保在1分钟内全部发完 打个比方会提出类似的场景**

## **引言**

据说有人归纳了计算机的五大常用算法，它们是贪婪算法，动态规划算法，分治算法，回溯算法以及分支限界算法。虽然不知道为何要将这五个算法归为最常用的算法，但是毫无疑问，这五个算法是有很多应用场景的，最优化问题大多可以利用这些算法解决。算法的本质就是解决问题。当数据量比较小时，其实根本就不需要什么算法，写一些for循环完全就可以很快速的搞定了，但是当数据量比较大，场景比较复杂的时候，编写for循环就是一个很不明智的方式了。一是耗时，二是写出的代码绝对是天书。当然还有第三点，这点也是最重要的，写代码是一种艺术，而不是搬砖。前面的文章里对这五种算法都已经做了详细的讲解和归纳，本文主要是一个总结，将这五种算法整理到一起来对比，分析一下。

## **0） 穷举法**

穷举法简单粗暴，没有什么问题是搞不定的，只要你肯花时间。同时对于小数据量，穷举法就是最优秀的算法。就像太祖长拳，简单，人人都能会，能解决问题，但是与真正的高手过招，就颓了。

## **1） 贪婪算法**

贪婪算法可以获取到问题的局部最优解，不一定能获取到全局最优解，同时获取最优解的好坏要看贪婪策略的选择。特点就是简单，能获取到局部最优解。就像打狗棍法，同一套棍法，洪七公和鲁有脚的水平就差太多了，因此同样是贪婪算法，不同的贪婪策略会导致得到差异非常大的结果。   
具体的详细解析请参见下面的文章：   
[http://blog.csdn.net/changyuanchn/article/details/51417211](http://blog.csdn.net/changyuanchn/article/details/51417211" \t "https://blog.csdn.net/changyuanchn/article/details/_blank)

## **2） 动态规划算法**

当最优化问题具有重复子问题和最优子结构的时候，就是动态规划出场的时候了。动态规划算法的核心就是提供了一个memory来缓存重复子问题的结果，避免了递归的过程中的大量的重复计算。动态规划算法的难点在于怎么将问题转化为能够利用动态规划算法来解决。当重复子问题的数目比较小时，动态规划的效果也会很差。如果问题存在大量的重复子问题的话，那么动态规划对于效率的提高是非常恐怖的。就像斗转星移武功，对手强它也会比较强，对手若，他也会比较弱。   
具体的详细解析请参见下面的文章：   
[http://blog.csdn.net/changyuanchn/article/details/51420028](http://blog.csdn.net/changyuanchn/article/details/51420028" \t "https://blog.csdn.net/changyuanchn/article/details/_blank)   
[http://blog.csdn.net/changyuanchn/article/details/51429979](http://blog.csdn.net/changyuanchn/article/details/51429979" \t "https://blog.csdn.net/changyuanchn/article/details/_blank)

## **3）分治算法**

分治算法的逻辑更简单了，就是一个词，分而治之。分治算法就是把一个大的问题分为若干个子问题，然后在子问题继续向下分，一直到base cases，通过base cases的解决，一步步向上，最终解决最初的大问题。分治算法是递归的典型应用。   
具体的详细解析请参见下面的文章：   
[http://blog.csdn.net/changyuanchn/article/details/17150109](http://blog.csdn.net/changyuanchn/article/details/17150109" \t "https://blog.csdn.net/changyuanchn/article/details/_blank)   
[http://blog.csdn.net/changyuanchn/article/details/51465175](http://blog.csdn.net/changyuanchn/article/details/51465175" \t "https://blog.csdn.net/changyuanchn/article/details/_blank)

## **4） 回溯算法**

回溯算法是深度优先策略的典型应用，回溯算法就是沿着一条路向下走，如果此路不同了，则回溯到上一个   
分岔路，在选一条路走，一直这样递归下去，直到遍历万所有的路径。八皇后问题是回溯算法的一个经典问题，还有一个经典的应用场景就是迷宫问题。   
具体的详细解析请参见下面的文章：   
[http://blog.csdn.net/changyuanchn/article/details/17354461](http://blog.csdn.net/changyuanchn/article/details/17354461" \t "https://blog.csdn.net/changyuanchn/article/details/_blank)

## **5） 分支限界算法**

回溯算法是深度优先，那么分支限界法就是广度优先的一个经典的例子。回溯法一般来说是遍历整个解空间，获取问题的所有解，而分支限界法则是获取一个解（一般来说要获取最优解）。   
具体的详细解析请参见下面的文章：   
[http://blog.csdn.net/changyuanchn/article/details/17102037](http://blog.csdn.net/changyuanchn/article/details/17102037" \t "https://blog.csdn.net/changyuanchn/article/details/_blank)