目录

[1、JVM面试题 3](#_Toc482113662)

[9. 类加载的五个过程：加载、验证、准备、解析、初始化。 3](#_Toc482113663)

[在JVM中什么算是大对象？ 9](#_Toc482113664)

[9.Math.round(11.5) 等于多少? Math.round(-11.5)等于多少? 10](#_Toc482113665)

[10. String, StringBuffer StringBuilder的区别。 10](#_Toc482113666)

[2、java中的关键字作用 10](#_Toc482113667)

[3、Object类里面有哪几种方法，作用 11](#_Toc482113668)

[类加载器 11](#_Toc482113669)

[4、equals 和 hashCode方法，重写equals的原则() 13](#_Toc482113670)

[6、Java引用类型(强引用，软引用，弱引用，虚引用) 13](#_Toc482113671)

[如何利用软引用和弱引用解决OOM问题？ 16](#_Toc482113672)

[8、线程和进程的区别 17](#_Toc482113673)

[9、Exception和Error 18](#_Toc482113674)

[10、反射的用途 21](#_Toc482113675)

[11、HashMap实现原理 21](#_Toc482113676)

[12、有顺序的map 23](#_Toc482113677)

[Java集合类面试题 23](#_Toc482113678)

[List和set是怎样排序的 23](#_Toc482113679)

[TreeMap的实现原理 26](#_Toc482113680)

[Map集合 27](#_Toc482113681)

[12、为什么要使用迭代器 28](#_Toc482113682)

[12、List有哪些子类，各有什么区别 28](#_Toc482113683)

[13、NIO相关，缓冲区、通道、selector。。。 28](#_Toc482113684)

[14、内存泄露，举个例子 31](#_Toc482113685)

[15、OOM是怎么出现的，有哪几块JVM区域会产生OOM，如何解决 31](#_Toc482113686)

[16、Java里面的观察者模式实现 31](#_Toc482113687)

[17、单例实现(我一般用enum写，不容易被挑毛病) 31](#_Toc482113688)

[19、Java泛型机制，泛型机制的优点，以及类型变量 33](#_Toc482113689)

[20、Java集合框架源码 34](#_Toc482113690)

[21、synchronized关键字原理 34](#_Toc482113691)

[ReentrantLock的实现原理，AbstractQueuedSynchronizer的实现原理 35](#_Toc482113692)

[ReentrantReadWriteLock的原理 36](#_Toc482113693)

[22、concurrent包 36](#_Toc482113694)

[23、preparedstatement底层如何实现？ 37](#_Toc482113695)

[24、类之间有哪些关系？ 37](#_Toc482113696)

[26、GC太频繁如何处理？ 38](#_Toc482113697)

[27、线程池 38](#_Toc482113698)

[Executor、ExecutorService和Executors 40](#_Toc482113699)

[Future与FutureTask 42](#_Toc482113700)

[28、多线程和多任务 43](#_Toc482113701)

[29、设计模式 43](#_Toc482113702)

[30、如何实现接口分离？ 43](#_Toc482113703)

[29、什么是虚拟内存？ 44](#_Toc482113704)

[30、动态代理的源码要看 44](#_Toc482113705)

[31、进程间通信机制 46](#_Toc482113706)

[竞态条件 47](#_Toc482113707)

[33、三次握手和四次挥手 48](#_Toc482113708)

[四次挥手 49](#_Toc482113709)

[34、滑动窗口 50](#_Toc482113710)

[35、最安全的单例模式 50](#_Toc482113711)

[连接池的原理是什么 50](#_Toc482113712)

[连接池中的连接是长连接还是短连接？为什么？ 50](#_Toc482113713)

[重载重写 56](#_Toc482113714)

[wait、notify为什么不能用在Thread类里面 56](#_Toc482113715)

[使用new和newInstance创建对象的区别 56](#_Toc482113716)

[使用forName和loadClass加载类的区别 56](#_Toc482113717)

[注解 56](#_Toc482113718)

[interrupt与isInterrupt 57](#_Toc482113719)

[String为什么要设计为不可变 57](#_Toc482113720)

[不开辟新的变量空间，原地交换两个变量 57](#_Toc482113721)

[Java回调机制 57](#_Toc482113722)

[为什么要使用内部类？ 58](#_Toc482113723)

[Java中采用的是大端序 59](#_Toc482113724)

[Java中Comparator与Comparable有什么不同？ 60](#_Toc482113725)

[静态导包 60](#_Toc482113726)

[用原子操作可以实现无锁编程 60](#_Toc482113727)

[Java中指令重排的意义 60](#_Toc482113728)

[共享锁(S锁)与排他锁(X锁)是怎样实现的 60](#_Toc482113729)

[什么是饥饿？ 60](#_Toc482113730)

[公平锁与非公平锁 60](#_Toc482113731)

[阻塞与非阻塞的区别 61](#_Toc482113732)

[Java序列化的过程 61](#_Toc482113733)

[什么是事务？ 62](#_Toc482113734)

[事务隔离 62](#_Toc482113735)

[HTTP面试题 63](#_Toc482113736)

[Java IO和NIO面试题 64](#_Toc482113737)

[在Java程序中，我有三个socket，我需要多少个线程来处理。 64](#_Toc482113738)

[Java中怎么创建ByteBuffer？ 64](#_Toc482113739)

[Java中怎么读写ByteBuffer？ 64](#_Toc482113740)

[Java采用的是大端序还是小端序？ 64](#_Toc482113741)

[ByteBuffer中的字节序是什么？ 64](#_Toc482113742)

[Java中直接缓冲区与非直接缓冲区有什么区别？ 64](#_Toc482113743)

[Java中的内存映射缓冲区是什么？ 64](#_Toc482113744)

[Socket选项TCP NO DELAY是指什么？ 64](#_Toc482113745)

[TCP协议与UDP协议有什么区别？ 64](#_Toc482113746)

[Java中ByteBuffer与StringBuffer有什么区别？ 64](#_Toc482113747)

[编程和代码相关的面试题 64](#_Toc482113748)

[93）怎么检查一个字符串只包含数字？ 64](#_Toc482113749)

[94）Java 中如何利用泛型写一个 LRU 缓存？ 64](#_Toc482113750)

[95）写一段 Java 程序将 byte 转换为 long？ 64](#_Toc482113751)

[95）在不使用 StringBuffer 的前提下，怎么反转一个字符串？ 64](#_Toc482113752)

[97）Java 中，怎么获取一个文件中单词出现的最高频率？ 64](#_Toc482113753)

[98）如何检查出两个给定的字符串是反序的？ 65](#_Toc482113754)

[99）Java 中，怎么打印出一个字符串的所有排列？ 65](#_Toc482113755)

[100）Java 中，怎样才能打印出数组中的重复元素？ 65](#_Toc482113756)

[101）Java 中如何将字符串转换为整数？ 65](#_Toc482113757)

[102）在没有使用临时变量的情况如何交换两个整数变量的值？ 65](#_Toc482113758)

[设计模式面试题 65](#_Toc482113759)

[129）Java 中，Serializable 与 Externalizable 的区别？ 68](#_Toc482113760)

[设计模式六大原则 68](#_Toc482113761)

[设计模式 69](#_Toc482113762)

[设计模式的分类 69](#_Toc482113763)

# 1、JVM面试题

方法区和永久带的关系：方法区是JVM规范中定义的一个概念，用于存储类信息、常量池、静态变量、JIT编译后的代码等信息，具体放在哪里，不同的实现可以放在不同的地方。而永久代是HotSpot虚拟机特有的概念，是方法区的一种实现，别的JVM都没有这个这个东西。

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**](http://lib.csdn.net/base/datastructure)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？

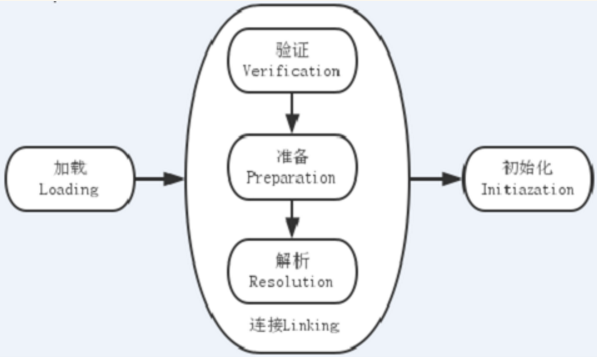
6. GC收集器有哪些？CMS收集器与G1收集器的特点。

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。

类加载器的任务就是根据一个类的权限定名来读取此类的二进制字节流到JVM中，然后转化为一个与目标类对应的java.lang.Class对象实例。



#### 加载

根据权限定名读取二进制字节流到JVM内部，并存储在运行时内存区的方法区(永久带)，然后将其转化为一个与目标类对应的java.lang.Class对象实例，这个Class对象在日后就会作为方法区中该类的各种数据的访问入口。

#### 链接

链接分为验证、准备和解析三个阶段

1. 验证：验证类信息是否符合JVM规范，是否是一个有效的字节码文件，**验证内容覆盖了类数据信息的格式验证、语义分析、操作验证等**。格式验证：验证是否符合class文件规范。语义验证：检查一个被标记为final的类型是否包含子类；检查一个类中的final方法是否被进行重写；确保子类和父类之间没有不兼容的一些方法声明(比如，重写中，方法签名相同但是返回类型不同)。操作验证：在操作数栈中的数据必须进行正确的操作，
2. 准备：**将类中的所有静态变量分配内存空间，并为其设置一个初始值。**被final修饰的静态变量会直接赋予原值：类字段的字段属性表中存在ConstantValue属性，则在准备阶段，其值就是ConstantValue的值
3. 解析：将常量池中的符号引用转换为直接引用(得到类或者字段、方法在内存中的指针、偏移量，以便直接调用该方法)，这个可以在初始化之后再执行。可以认为是一些静态绑定的会被解析，动态绑定则只会在运行时进行解析；静态绑定包含一些final方法(不可以重写)，static方法，构造器

#### 初始化

将一个类中所有被static标识的代码统一执行一遍，如果执行的是静态变量，那么就会使用用户指定的值覆盖之前在准备阶段设置的初始值；如果执行的是static代码块，那么在初始化阶段，JVM就会执行static代码块中定义的所有操作。

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。

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

- [**Java**](http://lib.csdn.net/base/javase)字节码文件的格式  
- 内部类的存储方式  
- 垃圾回收器的分类及优缺点  
- 类在虚拟机中的加载过程  
- 即时编译器的前后端优化方法  
- CMS垃圾回收器的工作过程  
- CAS指令以及其他线程安全的方法  
- 各种内存溢出的情况，包括JNI调

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

1）JVM内存模型与垃圾回收机制：

jvm的内存：

**java堆（JavaHeap）**

1.用来存放对象的，几乎所有对象都放在这里，被线程共享的，或者说是被栈共享的

2.堆又可以分为**新生代和老年代**，实际还有一个区域叫永久代，但是jdk1.7已经去永久代了，所以可以当作没有，永久代是当jvm启动时就存放的JDK自身的类和接口数据，关闭则释放。  
      3.StackOverflowError：如果在线程执行的过程中，栈空间不够用，那么JVM就会抛出此异常，这种情况一般是死递归造成的。  
    4. OutOfMemoryError:如果JVM内存大小是可扩展的，当然一般都是可以扩展的，当自动扩展到计算机本身内存大小时会抛出OutOfMemoryError。比如内存为2G，当JVM超过配置大小自动扩展至2G时会抛出OutOfMemoryError。

例如：

public class StackOverflowTest {

public static void main(String[] args) {

method();

}

public static void method(){

for(;;)

method();

}

}

import java.util.ArrayList;

import java.util.List;

public class OutOfMemoryTest {

public static void main(String[] args){

List list=new ArrayList();

for(;;){

int[] tmp=new int[1000000];

list.add(tmp);

}

}

}

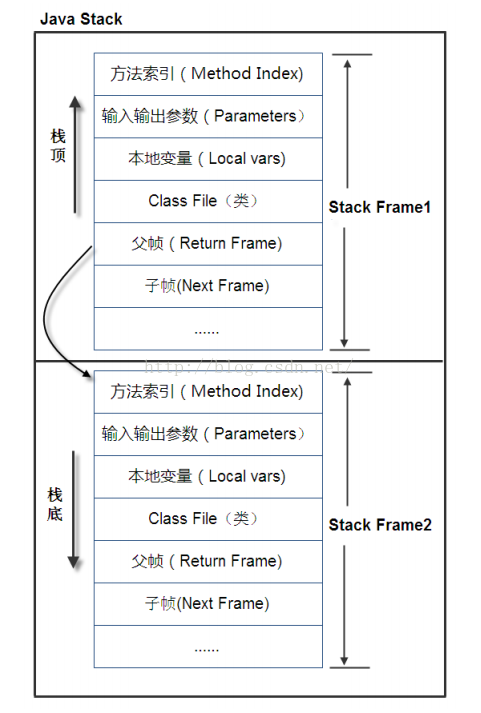
JVM堆一般分为三个部分：  
Young：年轻代  
Young区被划分为三部分，Eden(伊甸)区和两个大小严格相同的Survivor(幸存)区，其中Survivor区间中，某一时刻只有其中一个是被使用的，另外一个留做垃圾收集时复制对象用，在Young区间变满的时候，minor GC就会将存活的对象移到空闲的Survivor区间中，**根据JVM的策略，在经过几次垃圾收集后，任然存活于Survivor的对象将被移动到Tenured区间。**

**新建对象就在Eden区，Eden就是伊甸，顾名思义。但是并不是对象最活跃的区域，对象最活跃的区域是老年代，因为经过各种垃圾回收之后对象都跑到这里来了。**

Tenured：年老代  
Tenured区主要保存生命周期长的对象，一般是一些老的对象，当一些对象在Young复制转移一定的次数以后，对象就会被转移到Tenured区，一般如果系统中用了application级别的缓存，缓存中的对象往往会被转移到这一区间。  
Perm：持久代  
Perm代主要保存class,method,filed等对象，这部门的空间一般不会溢出。

2.2堆的大小  
-Xmx：指定JVM堆得最大内存，在JVM启动以后，会分配-Xmx参数指定大小的内存给JVM，但是不一定全部使用，JVM会根据-Xms参数来调节真正用于JVM的内存  
-Xms ：指定了JVM初始启动以后初始化内存  
-Xmx -Xms之差就是三个Virtual空间的总大小（年轻代、年老代、持久代）

jvm栈  
1.要说栈是用来存什么的，其实我感觉不严谨，**栈是运行时创建的，是跟随线程的**，它不是用来存什么的，那它用来干什么的，它是用来存栈帧的，没有图不太好说呢，等下我去截个图。每当一个方法被执行时都会在虚拟机中新创建一个栈帧，方法调用结束后即被销毁。每一个栈帧都有自己的局部变量表、操作数栈和指向当前方法所属的类的运行时常量池的引用。



2.栈的好处就是不需要垃圾回收，随着线程结束内存就释放。

方法区和运行时常量池  
**1.方法区是**堆的一个逻辑区域，但是又叫非堆。运行时常量池又是方法区的一部分，真正的一部分。方法区并不是存方法的，存方法的应该是栈或者栈帧。**方法区存的是类信息、常量、静态变量等，也是被线程共享的区域。运行时常量池存放的是编译期生产的各种字面量和符号引用。**  
2.这块内存区域的回收没啥好说的，因为我也不太清楚，我只知道HotSpot的设计团队选择把GC分代扩展至方法区了，或者是使用永久代实现方法区。  
3.内存是肯定会溢出的，不断创建类会导致方法区内存溢出，而不断将常量放入常量池（String.intern()），常量池也会内存溢出。

3、栈与堆的比较：  
栈是**运行**时的单位，而堆是**存储**的单位。  
栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。在Java中**一个线程**就会相应有**一个线程栈**与之对应，这点很容易理解，**因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。  
对象存放在堆中，对象引用和基本类型存放在栈中；对象的属性存放在堆中，对象的方法存在栈中。**  
程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。  
**在JVM中，静态属性保存在Stack指令内存区，动态属性保存在Heap数据内存区**

**垃圾回收机分代垃圾回收**

**分代垃圾回收**不同的对象生命周期不同。与业务信息有关的对象生命周期较长，如Http请求中的session对象、线程、socket连接。程序运行过程中的临时变量生命周期较短，如String对象。把不同生命周期的对象放在不同代上，**不同代上采用最合适它的垃圾回收方式进行回收**。JVM中共划分为三个代：年轻代、年老代和持久代，其中持久代主要存放Java类的类信息，与垃圾收集要收集的Java对象关系不大，年轻代和年老代的划分是对垃圾收集影响较大的。  
年轻代：存放所有新生成的对象；  
年老代：在年轻代中经历了N次垃圾回收仍然存活的对象，将被放到年老代中，故都是一些生命周期较长的对象；  
持久代：用于存放静态文件，如Java类、方法等。（持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如[**hibernate**](http://lib.csdn.net/base/javaee)等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。）  
新生代的垃圾收集器命名为“minor gc”，Full gc是清理整个heap。其中用System.gc()强制执行的是Full Gc.

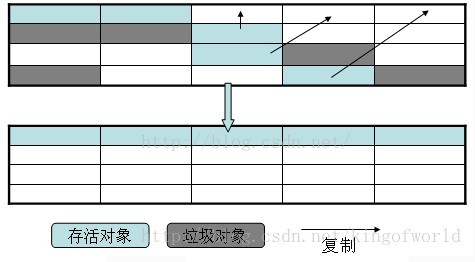
**判断对象是否需要回收的方法有两种：**

**1.引用计数  
　　引用计数存储对特定对象的所有引用数**，也就是说，当应用程序创建引用以及引用超出范围时，jvm必须适当增减引用数。**当某对象的引用数为0时，便可以进行垃圾收集。  
2.对象引用遍历**  
　　早期的jvm使用引用计数，现在大多数jvm采用**对象引用遍历**。***对象引用遍历从一组对象开始，沿着整个对象图上的每条链接，递归确定可到达（reachable）的对象。***如果某对象不能从这些根对象的一个（至少一个）到达，则将它作为垃圾收集。在对象遍历阶段，gc必须记住哪些对象可以到达，以便删除不可到达的对象，这称为**标记（marking）对象**。  
　　下一步，gc要删除不可到达的对象。删除时，有些gc只是简单的扫描堆栈，删除未标记的对象，并释放它们的内存以生成新的对象，这叫做清除（sweeping）。**这种方法的问题在于内存会分成好多小段，而它们不足以用于新的对象，但是组合起来却很大**。因此，**许多gc可以重新组织内存中的对象，**并进行压缩（compact），形成可利用的空间。  
　　为此，gc需要停止其他的活动。这种方法意味着所有与应用程序相关的工作停止，只有gc运行。结果，在响应期间增减了许多混杂请求。另外，更复杂的gc不断增加或同时运行以减少或者清除应用程序的中断。有的gc使用单线程完成这项工作，有的则采用多线程以增加效率。

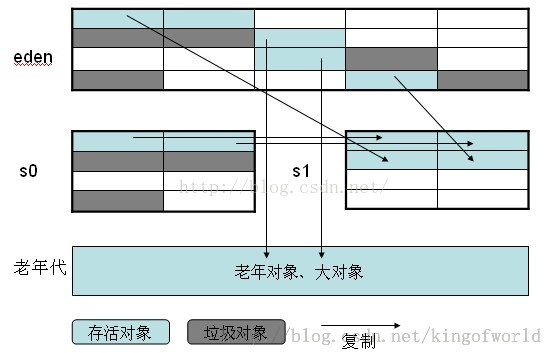
**几种垃圾回收机制(算法)**

**标记-清除算法(Mark-Sweep)**  
从根节点开始标记所有可达对象，其余没标记的即为垃圾对象，执行清除。但回收后的空间是不连续的。

**2.标记－压缩收集器**（适合用于老年代的算法（存活对象多于垃圾对象）。）  
　　有时也叫标记－清除－压缩收集器，与标记－清除收集器有相同的标记阶段。在第二阶段，则把标记对象复制到堆栈的新域中以便压缩堆栈。这种收集器也停止其他操作。



**3.复制收集器**  
　　复制算法(copying)  
将内存分成两块，每次只使用其中一块，垃圾回收时，将标记的对象拷贝到另外一块中，然后完全清除原来使用的那块内存。复制后的空间是连续的。复制算法适用于新生代，因为垃圾对象多于存活对象，复制算法更高效。在新生代串行垃圾回收算法中，将eden中标记存活的对象拷贝未使用的s1中，s0中的年轻对象也进入s1，如果s1空间已满，则进入老年代；这样交替使用s0和s1。这种改进的复制算法，既保证了空间的连续性，有避免了大量的内存空间浪费。



4.增量收集器  
　　增量收集器把堆栈分为多个域，每次仅从一个域收集垃圾。这会造成较小的应用程序中断。  
　　5.分代收集器  
　　这种收集器把堆栈分为两个或多个域，用以存放不同寿命的对象。jvm生成的新对象一般放在其中的某个域中。过一段时间，继续存在的对象将获得使用期并转入更长寿命的域中。分代收集器对不同的域使用不同的算法以优化性能。  
　　6.并发收集器  
　　并发收集器与应用程序同时运行。这些收集器在某点上（比如压缩时）一般都不得不停止其他操作以完成特定的任务，但是因为其他应用程序可进行其他的后台操作，所以中断其他处理的实际时间大大降低。  
　　7.并行收集器  
　　并行收集器使用某种传统的算法并使用多线程并行的执行它们的工作。在多cpu机器上使用多线程技术可以显著的提高java应用程序的可扩展性。

## 在JVM中什么算是大对象？

JVM内存模型，每个区域存储什么数据，每个区域的作用，问题入口从String a,new String(“”)开始

这里再说一下String类型的知识点：

String是不可变的，体现在：String类内部通过char数组来保存字符串，而这个char数组被声明为final。之所以这样做是为了考虑线程安全的同时降低消耗(与普通的并发机制相比)，这在设计模式中叫做“不变模式”。

请看这两个语句：String x = "abc"; String y = new String("abcd");

内存分配情况：

Java栈 常量池

x ------> "abc"

"abcd"

Java堆

y ------> "abcd"

String a = "a1";

String b = "a" + 1; //"a"是字符串字面量

System.out.println(a == b); //true 两个字符串字面量相连，得到的新的字符串仍然是字符串字面量

String a = "ab";

String bb = "b";

String b = "a" + bb;

System.out.println(a == b); //false 当字符串字面量与String类型变量连接时，得到的新字符串不再保留在常量池中，而是在堆中新建一个对象来存放

String a = "ab";

final String bb = "b";

String b = "a" + bb;

System.out.println(a == b); //true 字符串字面量与String类型常量相连得到的新字符串依然保留在常量池中。

通过函数return返回字符串字面量与String a = new String("ab")相同;

public String getBB(){

return "b";

}

当调用intern方法时，如果常量池中已经包含一个等于此String对象的字符串(用equals()方法确定)，则返回常量池中的字符串，否则将此字符串添加到池中，并返回此字符串对象的引用。

### 9.Math.round(11.5) 等于多少? Math.round(-11.5)等于多少?

Math.round(11.5)==12 Math.round(-11.5)==-11 round 方法返回与参数 最接近的长整数，参数加 1/2 后求其 floor.

### 10. String, StringBuffer StringBuilder的区别。

String 的长度是不可变的；

StringBuffer的长度是可变的，如果你对字符串中的内容经常进行操作，特别是内容要修改时，那么使用 StringBuffer，如果最后需要 转换为String，那么使用 StringBuffer 的 toString() 方法；线程安全；

StringBuilder 是从 JDK 5 开始，为StringBuffer该类补充了一个单个线程使用的等价类；通常应该优先使用 StringBuilder 类，因为它支持所有相同的操作，但由于它不执行同步，所以速度更快。   
使用字符串的时候要特别小心，如果对一个字符串要经常改变的话，就一定不要用String,否则会创建许多无用的对象出来.   
来看一下比较

String a = “abcd”;

a = a + 1;

JVM会这样解析这段代码，首先创建一个对象a，赋值为abcd，然后再创建一个新的对象s，用来执行第二段代码，也就是我们之前对象并没有变化。

**下面是一个特殊的例子：**

String s = "hello"+"world"+"i love you";

StringBuffer Sb = new StringBuilder("hello").append("world").append("i love you");

这个时候s有多个字符串进行拼接，按理来说会有多个对象产生，但是jvm会对此进行一个优化，也就是说只创建了一个对象，此时它的执行速度要比StringBuffer拼接快.再看下面这个:

String s2 = "hello";

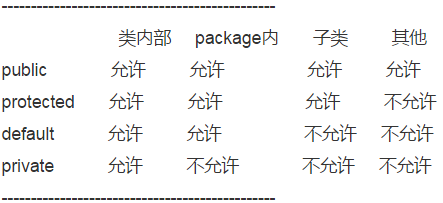
String s3 = "world";

String s4 = "i love you";

String s1 = s2 + s3 + s4;

## 2、java中的关键字作用

private protected public static final 等



## 3、Object类里面有哪几种方法，作用

protected Object clone()创建并返回此对象的一个副本(clone()方法是浅拷贝，需要重写clone()方法来实现深拷贝，需要重写的类要实现克隆Cloneable接口)

boolean equals()

protected void finalize()

#### Class<?> getClass()

获得一个对象的类型类，例如A a = new A(); a.getClass()的值为A.class

int hashCode()返回此对象的hash码值

void notify()唤醒在此对象监视器上等待的单个线程

void notifyAll()唤醒在此对象监视器上等待的所有线程

toString()

void wait()在其他线程调用此对象的notify()或者notifyAll()方法前，导致当前线程等待

void wait(long timeout)

void wait(long timeout, int nanos)

## 类加载器

#### 类加载器的基本概念

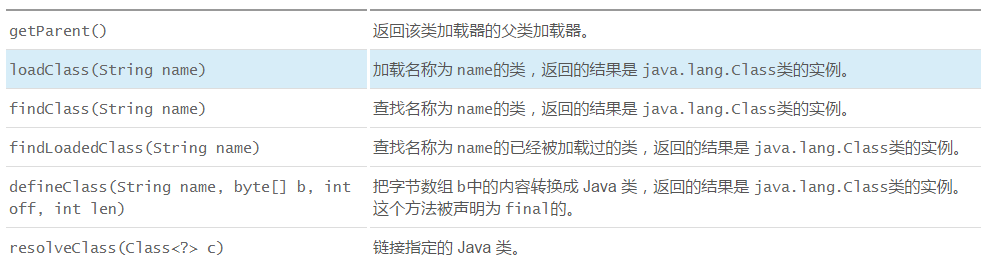
类加载器负责加载Java类的字节代码到虚拟机中。一般来说java虚拟机使用java类的方式如下：java源程序在经过java编译器编译之后就被转换为java字节代码(.class文件)。类加载器负责读取Java字节代码，并转换成java.lang.Class的一个实例。每个这样的实例表示一个Java类。通过此实例的newInstance()方法就可以创建出该类的一个对象。

基本上所有的类加载器都是java.lang.ClassLoader类的一个实例。下面详细介绍这个java类。

#### java.lang.ClassLoader类介绍

java.lang.ClassLoader类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个Java类，即java.lang.Class类的一个实例。除此之外，ClassLoader还负责加载java应用所需的资源，如图像文件和配置文件等

ClassLoader中与加载类相关的方法



#### 类加载器的树状组织结构

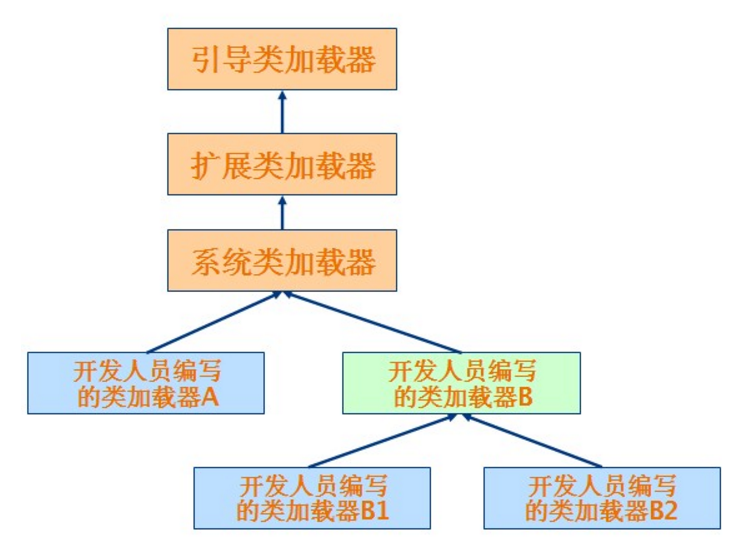
java中的类加载器大致可以分成两类：一类是系统提供的，另外一类是由java开发人员编写的。系统提供的类加载器主要有下面3个：

* 引导类加载器(bootstrap class loader)：它用来加载java的核心库(rt.jar)，是用原生代码来实现的，并不继承自java.lang.ClassLoader
* 扩展类加载器(extensions class loader)：它用来加载java的扩展库(jre/lib/ext)。java虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载java类。
* 系统类加载器(system class loader)：它根据Java应用的类路径(CLASSPATH)来加载Java类。一般来说，java应用的类都是由它来加载完成的。可以通过ClassLoader.getSystemClassLoader()来获取它。

除了系统提供的类加载器之外，开发人员可以通过继承java.lang.ClassLoader类的方式实现自己的类加载器，以满足一些特殊需求。

除了引导类加载器之外，所有类加载器都有一个父类加载器。通过上面表中给出的getParent()方法可以获得。

类加载器树状组织结构示意图：



演示类加载器的树状组织结构：

public class ClassLoaderTree {  
 public static void main (String[] args) {  
 ClassLoader loader = ClassLoaderTree.class.getClassLoader();  
 while (loader != null) {  
 System.*out*.println(loader.toString());  
 loader = loader.getParent();  
 }  
 }  
}

运行结果：



第一个输出的是ClassLoaderTree的类加载器，即系统类加载器。它是sun.misc.Launcher$AppClassLoader类的实例；第二个输出是扩展类加载器，是sun.misc.Launcher$ExtClassLoader类的实例。注意，这里并没有输出引导类加载器，这是由于有些JDK的实现对于父类加载器是引导类加载器的情况，getParent()返回null。

#### 类加载器的代理模式(双亲委派模型)

类加载器在尝试自己去查找某个类的字节代码并定义它时，会先代理给其父类加载器，由父类加载器先去尝试加载这个类，以此类推。

首先需要说明下java虚拟机是如何判断两个java类是相同的：java虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。如果是同样的字节代码，被不同的类加载器加载之后所得到的类也是不同的。

这样就可以了解代理模式的设计动机了。代理模式是为了保证Java核心库的类型安全。我们都知道所有的java类都是Object类的子类，在运行的时候，Object类需要被加载到java虚拟机中，如果这个过程由java类自己的类加载器来完成的话，很可能就存在多个Object类，而且这些类之间是不兼容的。通过代理模式，对于java核心库的类的加载工作由引导类加载器来统一完成，保证了所使用的都是同一个版本的java核心库的类，是相互兼容的。

使用双亲委托模型来组织类加载器之间的关系，有一个显而易见的好处就是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类java.lang.Object，它存放在rt.jar之中，无论哪一个类加载器要加载这个类，最终都是委托给处于模型最顶端的启动类加载器进行加载，因此Object类在程序的各种加载器环境中都是同一个类。相反，如果没有使用双亲委托模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，那系统中将会出现多个不同的Object类，Java类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。如果自己去编写一个与rt.jar类库中已有类重名的Java类，将会发现可以正常编译，但永远无法被加载运行。

双亲委托模型对于保证Java程序的稳定运作很重要，但它的实现却非常简单，实现双亲委托的代码都集中在java.lang.ClassLoader的loadClass()方法中，逻辑清晰易懂：先检查是否已经被加载过，若没有加载则调用父类加载器的loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载器加载失败，抛出ClassNotFoundException异常后，再调用自己的findClass方法进行加载。

## 4、equals 和 hashCode方法，重写equals的原则()

原始的equals()方法用来比较两个对象的地址值

hashCode()用来获得对象地址的hash码值

重写equals必须重写hashCode()，因为有规定，如果两个对象equals的结果相同，那么hashCode()的结果也要相同。两个对象的hashCode()相同，但是两个对象不一定相等。

自反性：对于任何非空引用x，x.equals(x)应该返回true

对称性：如果x.equals(y)，那么y.equals(x)

传递性：对于任何引用，如果x.equals(y)，y.equals(z)那么x.equal(z)

一致性：如果x和y引用的对象没有发生变化，那么反复调用x.equals(y)应该返回同样的结果

## 6、Java引用类型(强引用，软引用，弱引用，虚引用)

#### Java中提供这四种引用的目的：

第一是可以让程序员通过代码的方式决定某些对象的生命周期。

第二是有利于JVM进行垃圾回收。

强引用：通常我们使用new操作符创建一个对象时所返回的引用即为强引用

软引用：若一个对象只能通过软引用到达，那么这个对象在内存不足时会被回收，可用于图片缓存中，内存不足时系统会自动回收不在使用的BitMap

弱引用：若一个引用只能通过弱引用到达，那么它就会被回收(即使内存充足)，同样可用于图片缓存中，这时候只要BitMap不再使用就会被回收。

虚引用：虚引用是Java中最“弱”的引用，通过它甚至无法获得被引用的对象，它存在的唯一作用就是当它指向的对象回收时，它本身会被加入到引用队列中，这样我们可以知道它指向的对象何时被销毁。

#### 强引用

是指创建一个对象并把这个对象赋给一个引用变量，比如：

Object object = new Object();

String str = “hello”;

强引用有引用变量指向时永远不会被垃圾回收，JVM宁愿抛出OOM错误也不会回收这些对象。如果想中断强引用与某个对象之间的关联，可以显示的将引用赋值为null，这样一来，JVM在合适的时间就会回收该对象。

#### 软引用

如果一个对象具有软引用，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。

软引用可用来实现内存敏感的高速缓存，比如网页缓存、图片缓存等。使用软引用可以防止内存泄漏，增强程序的健壮性。

java.lang.ref.SoftReference的特点是它的一个实例保存对一个Java对象的软引用，该软引用的存在不妨碍垃圾收集器对该Java对象的回收。一旦SoftReference保存了对一个Java对象的软引用后，在垃圾线程对这个对象回收前，SoftReference类所提供的get()方法返回Java对象的强引用，另外一旦线程回收该Java对象之后，get()方法将返回null。

举个例子：

Object obj = **new** Object();

SoftReference softRef = **new** SoftReference(obj);

上面代码中obj有两个引用路径，一个是来自SoftReference对象的软引用，一个来自变量obj的强引用。随即我们可以结束对obj的强引用，通过代码obj = null；此后这个对象obj就成为了软引用对象(注意只有将obj设为null，它才能成为软引用)。如果垃圾收集器进行内存垃圾收集，并不会因为有一个SoftReference对该对象的引用而始终保留该对象。垃圾收集器会在虚拟机抛出OOM之前回收软引用对象，而且虚拟机会尽可能回收长时间闲置不用的软引用对象，对那些新的软引用对象会尽可能保留。在回收这些对象之前，我们可以通过

Object object = (Object)softRef.get();重新获得对该实例的强引用。而回收之后，调用get()方法就只能得到null了。

SoftReference对象除了具有保存软引用的特殊性之外，也具有java对象的一般性。所以，当软引用对象被回收之后，虽然这个SoftReference对象的get()方法返回null，但这个SoftReference对象已经不具有存在的价值了，所以需要一种清除机制来回收这些SoftReference对象，避免内存泄漏。在java.lang.ref包里面还提供了ReferenceQueue。如果在创建SoftReference对象的时候，使用了一个ReferenceQueue对象作为参数提供给SoftReference的构造方法，如：

ReferenceQueue queue = new ReferenceQueue();

SoftReference ref = new SoftReference(obj,queue);

当SoftReference所引用的obj被垃圾回收器回收的同时，ref所强引用的SoftReference的对象被列入ReferenceQueue。也就是说，ReferenceQueue中保存的对象是Reference对象，而且是已经失去了它所软引用的对象的Reference对象。对于ReferenceQueue这个队列，当我们调用它的poll()方法的时候，如果这个队列中不是空队列，那么将返回队列前面的那个Reference对象。利用这个方法，我们可以检查哪个SoftReference所软引用的对象已经被回收，于是我们可以把这些对象清除掉。常用方式如下：

SoftReference ref = null;

while((ref=(SoftReference)q.poll())!=null){

//清除ref

}

#### 弱引用

当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。在Java中，用java.lang.ref.WeakReference类来表示。下面是实例代码：

**public** **class** RefTest {

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

People people = **new** People(15,"Lizan");

WeakReference weakRef = **new** WeakReference(people);

people = **null**;

System.*out*.println(weakRef.get());

System.*gc*();

System.*out*.println(weakRef.get());

}

}

结果如下：



注意：只有将people设为null，调用System.gc()才能绝对回收弱引用。也就是说，只要调用垃圾回收，被弱引用关联的对象必定会被回收掉。这里说的被弱引用关联的对象是指只有弱引用与之关联，如果同时存在强引用，则进行垃圾回收时也不会回收该对象(软引用也是一样)，所以将对象设为null来释放强引用。

#### 虚引用

虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在java中用java.lang.ref.PhantomReference类来表示。如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。要注意的是，虚引用必须和引用队列关联使用，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，就会把这个虚引用加入到与之关联的引用队列中。

被软引用关联的对象只有在内存不足的时候才会被回收，而被弱引用关联的对象在JVM进行垃圾回收时总被回收。

可以使用System.gc()来通知JVM进行垃圾回收。但是要注意，虽然发出了通知，JVM不一定会立刻执行，也就是说这句是无法确保此时JVM一定会进行垃圾回收的。

#### 对象可及性判断

在很多时候，一个对象并不是从根集直接引用的，而是一个对象被其他对象引用，甚至同时被几个对象引用，从而构成一个以根集为顶的树状结构。



由此带来了一个问题，那就是对象的可及性判断：

* 单条引用路径可及性判断：在这条路径中，最弱的一个引用决定对象的可及性。
* 多条引用路径可及性判断：几条路径中，最强的一条引用决定对象的可及性。

比如，我们假设引用①和③为强引用，⑤为软引用，⑦为弱引用，对于对象5按照这个两个规则判断，路径①-⑤取最弱的引用⑤，因此该路径对对象5的引用为软引用。同样，③-⑦为弱引用。在这条路径之间取最强的引用，于是对象5是一个软可及对象。

## 如何利用软引用和弱引用解决OOM问题？

下面举个例子，加入某个应用需要读取大量本地图片，如果每次读取图片都从硬盘读取，则会严重影响性能，但是如果全部加载到内存当中，又有可能造成内存溢出，此时使用软引用可以解决这个问题。

设计思路：用一个HashMap来保存图片路径和与相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM会自动回收这些缓存图片对象所占用的空间。在Android开发中经常用到。

#### 通过软可及对象的重获方法实现Java对象的高速缓存

下面来说明如何构建一种高速缓存器来避免重复构建同一个对象所带来的性能损失。

其核心思想是：先去SoftReference中调用get()方法来获取对象引用，如果返回null，则重新创建，这样可以节省性能开销。

#### 使用弱引用构建非敏感数据缓存

有些对象具有中等生命周期，比分配它的那个方法调用的生命周期长，但是比应用程序的生命周期短，如客户机的Socket连接。需要将一些元数据与这个套接字关联，如生成连接的用户的标识，但是这些数据不能添加到Socket对象上。这是典型的就是使用一个全局Map来存储这些信息。但是这种方法的问题就是元数据的生命周期需要与套接字的生命周期挂钩，但是需要知道什么时候不再需要这些套接字，并记住从Map中删除对应的值。但是找到这些Socket什么时候不再被使用是不容易的。

#### 使用WeakHashMap

该WeakHashMap存放了键对象的弱引用，当一个键对象被垃圾回收器回收时，对应的值也会被从Map中删除。

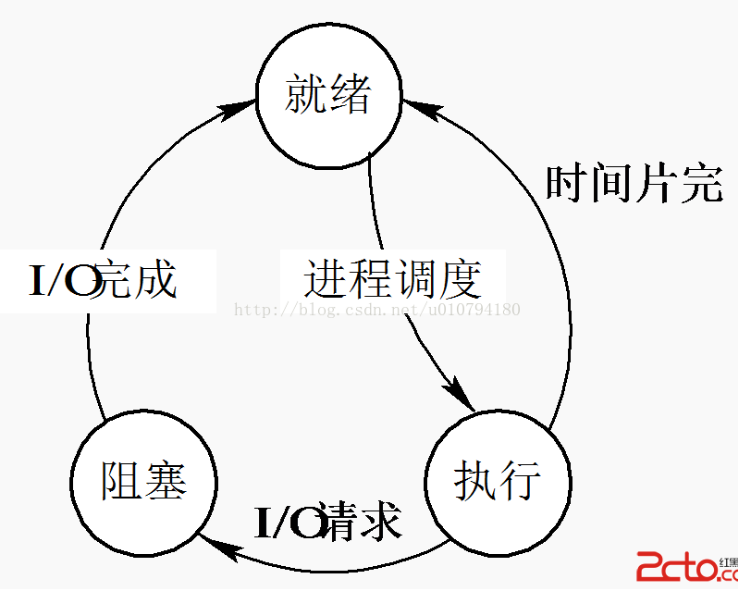
## 8、线程和进程的区别

#### 进程：

是并发执行的程序在执行过程中分配和管理资源的基本单位，是一个，动态，，概念，竞争，计，。算，，，。机系统资源的基本单位。每一个进程都有自己的地址空间，空间大小与处理机的位数有关。

进程的状态：

* 就绪状态：当进程分配到除CPU以外所有的必要资源以后，只要再获得CPU便可立即执行，进程这时的状态称为就绪状态。在一个系统中处于就绪状态的进程可能有多个，通常将他们排成一个队列，称为就绪队列。
* 执行状态：进程已获得CPU，其程序正在执行。在单处理机系统中，只有一个进程处于执行状态；在多处理机进程中，则有多个进程处于执行状态。
* 阻塞状态：正在执行的进程由于发生某事件而暂时无法继续执行时，便放弃处理机而处于暂停状态，即进程的执行受到阻塞，这种状态称为阻塞状态，有时也称为等待状态或封锁状态。致使进程阻塞的典型事件有：请求I/O，申请缓冲空间等，**进程sleep**，等待解锁。三者转换图如下：



阻塞的情况分三种：

1. 等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待队列中。
2. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。
3. 其他阻塞：运行的线程执行sleep()或join()方法，或者发出I/O请求时，JVM会把该线程设为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行状态。

#### 线程：

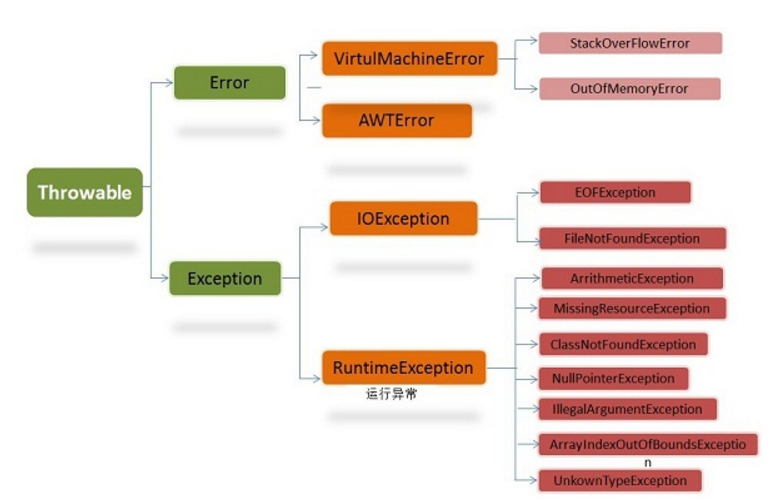
服务器通常接收大量请求，为每个请求都创建一个进程显然是行不通的。因此出现了线程。线程，是进程的一部分，一个没有线程的进程可以被看做是单线程的。线程有时也被称为轻量级进程，也是CPU调度的一个基本单位。

#### 二者的区别

一旦发生进程上下文切换，这些资源都是要被保护起来的。线程的改变只代表了CPU执行过程的改变，而没有发生进程所拥有的资源变化。①除了CPU之外，计算机内的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源。②进程拥有完整的虚拟地址空间，不依赖于线程而独立存在；反之，线程是进程的一部分，没有自己的地址空间，与进程内的其他线程一起共享分配给进程的所有资源。

线程并不是在所有计算机系统中都是适用的，如某些很少做进程调度和切换的实时系统。适用线程的好处是有多个任务需要处理机处理时，减少处理机的处理时间。最适合适用线程的系统是多处理机系统和网络系统或分布式系统。

## 9、Exception和Error



Error是程序无法处理的错误，表示代码运行时JVM出现的问题，例如OOM。这种异常发生时虚拟机一般选择线程终止。

Exception是程序本身可以处理的异常。Exception类有一个重要的子类RuntimeException。例如空指针异常，数组下标越界，除数为零等。

不可查异常包括运行时异常和错误。其他都是可查异常。这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch捕获，要么用throws抛出，否则编译不通过。

#### Java中什么是异常

简单来说，异常是Java传达给你的系统或程序错误的方式。在Java中，异常功能是通过实现比如Throwable，Exception，RuntimeException之类的类，然后还有一些处理异常时候的关键字，比如，throw，throws，try-catch，finally之类的。所有的异常都是通过Throwable衍生出来的。

#### Java中的检查异常和非检查异常有什么区别？

检查异常和非检查异常的区别在于其处理方式。检查型异常需要使用try-catch或throws在编译期进行处理，否则会出现编译器报错。对于非检查异常则不需要这麽做。Java中除了RuntimeException和Error之外都是检查型异常

#### 在Java异常处理过程中，你遵循的那些最好的实践是什么？

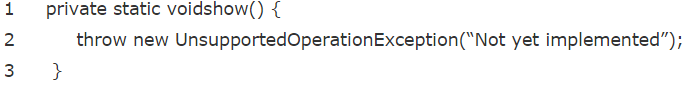
1. catch块里要写代码。通常起码要打印出错信息，当然最好根据需求对异常信息进行处理
2. 在Java中一定要在数据库连接、数据库查询、流处理后，在finally块中调用close()方法

#### 既然我们可以用RuntimeException来处理错误，那么你认为为什么Java中还会存在检查型异常

我认为体现了Java的一种完善机制。因为有些地方是很容易出现异常的，比如将文件读入到流中，这个文件很可能不存在，这个非常明显的，这恰恰体现了Java语言本身的完善机制。

#### throw和throws这两个关键在Java有什么不同

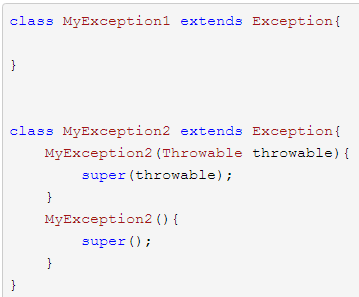
throws总是出现在一个函数头中，用来标明该成员函数可能出现的各种异常，你也可以申明未检查的异常，但这不是编译器强制的。如果方法抛出了异常那么调用这个方法的时候就要将这个异常进行处理。另一个关键字throw是用来抛出任意异常的，按照语法你可以抛出任意Throwable，throw可以中断程序执行，因此可以代替return。最常用的例子是在一个空方法中需要return的地方抛出：



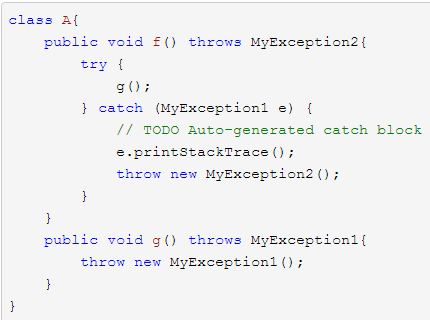
#### 什么是异常链？

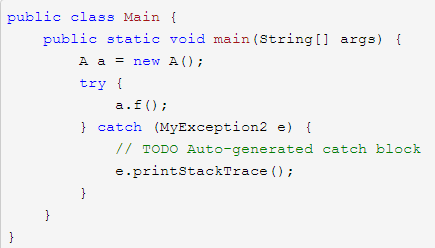
是指在进行一个异常处理时抛出了另外一个异常，由此产生了一个异常链条。顺便说下，如果因为处理一个异常你决定抛出一个新的异常，你一定要包含原有的异常，这样处理程序才可以通过getCause()和initCause()方法来访问异常最终的根源。

先自定义两个异常：

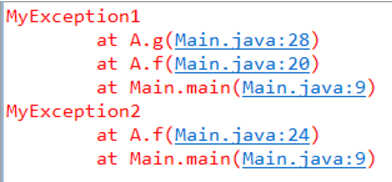


接下来：



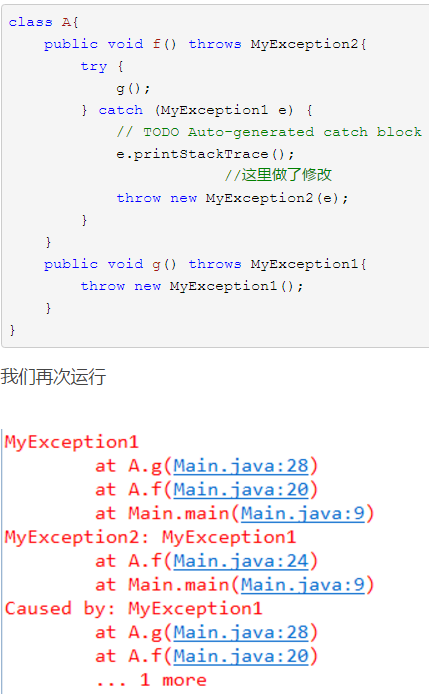


输出异常信息：



发现信息明显变少。这是因为MyException2对MyException1的环境一无所知。如果2能够持有1的环境信息，这样做起来就十分方便了，并且也形成了一条链，我们也成为异常链。

唯一要修改的就是调用MyException2的有参构造函数，把MyException1当做cause传递进去，这样的话我们就能获取到1的信息了。



## 10、反射的用途

#### 1、反射机制是什么？

反射机制是在运行状态中，对于任何一个类，都能够知道这个类的属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

反射就是增加程序的灵活性，避免将程序写死到代码中，(例如，class.forName("com.lizan.Person").newInstance())一般常用在框架中。

## 11、HashMap实现原理

(最常用的：数组+链表)，查找数据的时间复杂度

HashSet实现Set接口，HashMap实现Map接口，他们底层的Hash存储机制完全一样。甚至HashSet本身就采用HashMap来实现，其中将key作为set来使用。系统会采用hash算法来计算元素的存储位置。

集合当中存储的是对象的引用。

#### HashMap的存储实现：

当程序执行map.put(“语文”,80)时，系统将调用语文的hashCode()方法得到其hashCode值------每个java对象都有hashCode()方法，都可通过该方法获得它的hashCode值。系统会根据该hashCode值来决定该元素的存储位置。

public V put(K key, V value) {  
 if (key == null)  
 return putForNullKey(value);  
 int hash = *hash*(key.hashCode());

//搜索指定hash值在对应table表中的索引  
 int i = *indexFor*(hash, table.length);  
 for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
 Object k;  
 if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  
 V oldValue = e.value;  
 e.value = value;  
 e.recordAccess(this);  
 return oldValue;  
 }  
 }  
  
 modCount++;  
 addEntry(hash, key, value, i);  
 return null;  
}

indexFor(int h, int length)方法的代码如下：

static int indexFor(int h, int length) {  
 return h & (length-1);  
}

这种方法非常巧妙，HashMap底层数组的长度总是2的n次方，这样一来，假设h=5,length=16, 那么 h & length - 1 将得到 5；如果 h=6,length=16, 那么 h & length - 1 将得到 6 ……如果 h=15,length=16, 那么 h & length - 1 将得到 15；但是当 h=16 时 , length=16 时，那么 h & length - 1 将得到 0 了；当 h=17 时 , length=16 时，那么 h & length - 1 将得到 1 了……这样保证计算得到的索引值总是位于 table 数组的索引之内。

所以在调用put方法时，程序首先根据该key的hashCode()返回值决定该Entry的存储位置。当两个Entry对象的key的hashCode()返回值相同时，将由key通过equals()比较值决定是采用覆盖行为还是产生Entry链。

addEntry代码如下：

void addEntry(int hash, K key, V value, int bucketIndex) {

Entry<K,V> e = table[bucketIndex];  
 table[bucketIndex] = new Entry<K,V>(hash, key, value, e);  
 if (size++ >= threshold)  
 resize(2 \* table.length);  
 }

如果索引处有了一个对象，那新添加的Entry对象指向原有Entry对象(产生一个Entry链)，如果索引处没有Entry对象，那么就没有产生Entry链。

Hash算法的性能选项：

上面程序中有两个重要变量：

size：保存了HashMap中键值对的数量

threshold：该变量包含了HashMap能容纳的键值对的极限，它的值等于HashMap的容量乘以负载因子。每扩充一次HashMap的数组的长度扩大一倍。

当创建HashMap时，系统会自动创建一个table数组来保存HashMap中的Entry

HashMap的实际容量总是2的n次方，假如通过构造方法HashMap(int initialCapacity，float loadFactor)输入容量为10，那该方法总是指定实际容量为大于该数的最小的2的倍数，代码如下：

while (capacity < initialCapacity)//capacity从1开始  
 capacity <<= 1;

系统开始初始化HashMap时，系统会创建一个容量为capacity的Entry数组，这个数组里可以存储元素的位置称为“桶(bucket)”

HashMap存储示意图：



如果HashMap的每个bucket只有一个Entry，那性能是最好的；在发生冲突时，系统必须遍历链中的每个Entry

## 12、有顺序的map

TreeMap和LinkedHashMap

## Java集合类面试题

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

## ****List和set是怎样排序的****

List中每插入一个元素就会用数组elementData[size++] = newElement;这样保证顺序。

Java SDK不提供直接继承自Collection的类，Java SDK提供的类都是继承自Collection的“子接口”如List和Set。

#### ****21.HashMap的工作原理是什么?**** HashMap之所以在每个数组元素存储的是一个链表，是为了解决hash冲突问题，当两个对象的hash值相等时，那么一个位置肯定是放不下两个值的，于是hashmap采用链表来解决这种冲突，hash值相等的两个元素会形成一个链表。

#### ****22.HashMap与HashTable的区别是什么?****

1.HashTable基于Dictionary类，而HashMap是基于AbstractMap。  
2. HashMap的key和value都允许为null，而Hashtable的key和value都不允许为null。HashMap遇到key为null的时候，调用putForNullKey方法进行处理，而对value没有处理；Hashtable遇到null，直接返回NullPointerException。  
3. Hashtable是同步的，而HashMap是非同步的，但是我们也可以通过Collections.synchronizedMap(hashMap),使其实现同步。

#### 23.CorrentHashMap的工作原理?

**jdk 1.6版:** ConcurrenHashMap可以说是HashMap的升级版，ConcurrentHashMap是线程安全的，但是与Hashtable相比，实现线程安全的方式不同。Hashtable是通过对hash表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁。ConcurrentHashMap是采用分离锁的方式，它并没有对整个hash表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对hash表其他地方的访问。  
具体实现: ConcurrentHashMap内部有一个Segment数组, 该Segment对象可以充当锁。Segment对象内部有一个HashEntry数组，于是每个Segment可以守护若干个桶(HashEntry),每个桶又有可能是一个HashEntry连接起来的链表，存储发生碰撞的元素。  
每个ConcurrentHashMap在默认并发级下会创建包含16个Segment对象的数组，每个数组有若干个桶，当我们进行put方法时，通过hash方法对key进行计算，得到hash值，找到对应的segment，然后对该segment进行加锁，然后调用segment的put方法进行存储操作，此时其他线程就不能访问当前的segment，但可以访问其他的segment对象，不会发生阻塞等待。  
**jdk 1.8版** 在jdk 8中，ConcurrentHashMap不再使用Segment分离锁，而是采用一种乐观锁CAS[**算法**](http://lib.csdn.net/base/datastructure)来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

#### 24.遍历一个List有哪些不同的方式？

List<String> strList = new ArrayList<>();

//for-each

for(String str:strList) {

System.out.print(str);

}

//use iterator 尽量使用这种 更安全(fail-fast)

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

while(it.hasNext) {

System.out.printf(it.next());

}

#### 25.fail-fast与fail-safe有什么区别？

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util包中的所有集合类都被设计为fail->fast的，而java.util.concurrent中的集合类都为fail-safe的。当检测到正在遍历的集合的结构被改变时，Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛出ConcurrentModificationException。

#### 26.Array和ArrayList有何区别？什么时候更适合用Array？

1. Array可以容纳基本类型和对象，而ArrayList只能容纳对象。
2. Array是指定大小的，而ArrayList大小是固定的

#### 27.哪些集合类提供对元素的随机访问？

ArrayList、HashMap、TreeMap和HashTable类提供对元素的随机访问。

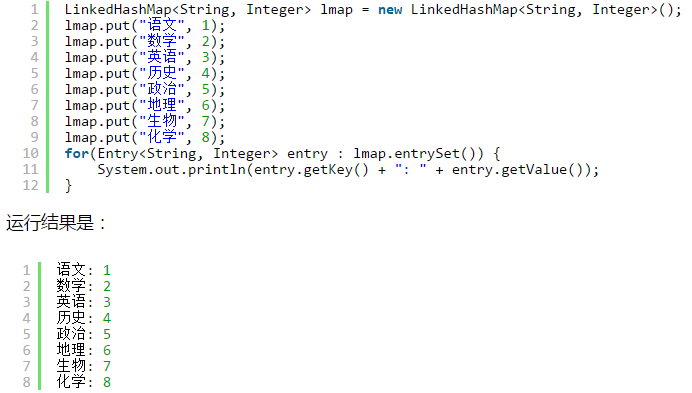
#### 28.HashSet的底层实现是什么?

通过看源码知道HashSet的实现是依赖于HashMap的，HashSet的值都是存储在HashMap中的。在HashSet的构造法中会初始化一个HashMap对象，HashSet不允许值重复，因此，HashSet的值是作为HashMap的key存储在HashMap中的，当存储的值已经存在时返回false。

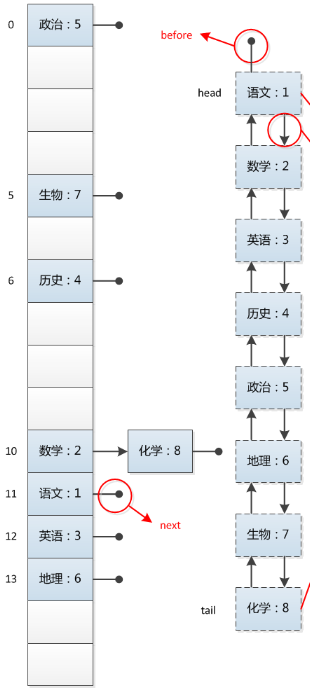
#### 29.LinkedHashMap的实现原理?

LinkedHashMap也是基于HashMap实现的，不同的是它定义了一个Entry header，这个header不是放在Table里，它是额外独立出来的。LinkedHashMap通过继承hashMap中的Entry,并添加两个属性Entry before,after,和header结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。LinkedHashMap定义了排序模式accessOrder，该属性为boolean型变量，对于访问顺序，为true；对于插入顺序，则为false。一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。

如下代码进行验证：



LinkedHashMap的内部结构图：

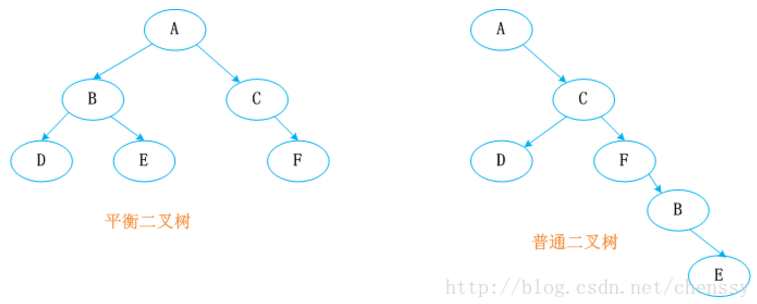


## TreeMap的实现原理

TreeMap的实现是红黑树算法的实现。

一棵基本的二叉查找树他们都需要满足一个基本性质：即树中任何节点的值大于它的左子节点，且小于它的右子节点。按照这个基本性质使得树的检索效率大大提高。但是在生成二叉树的过程中是非常容易失衡的，最坏的情况就是一边倒(只有左子树或右子树)，这样势必会导致二叉树的检索效率大大降低(O(n))，所以为了维持二叉树的平衡，想出各种方法。

平衡二叉树具备如下性质：它是一个空树或它的左右两个子树的高度差的绝对值不超过1，而且左右两个子树都是平衡二叉树。

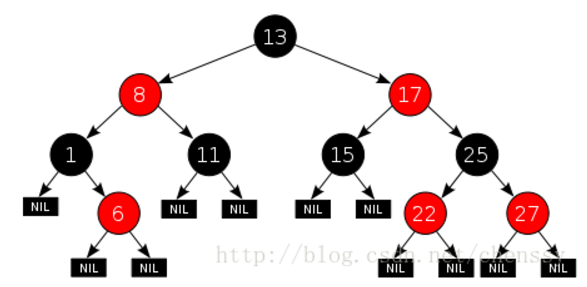


红黑树就是节点是红色或者是黑色的平衡二叉树，它通过颜色的约束来维持着二叉树的平衡，对于一棵有效的红黑二叉树而言我们必须制定如下规则：

1. 每个节点都只能是红色或黑色
2. 根节点是黑色
3. 每个叶节点是黑色
4. 如果一个节点是红的，则它两个子节点都是黑的。也就是在一条路径上不能出现相邻的两个红色节点。
5. 从任意节点到其每个叶子的所有路径都包含相同数目的黑色节点。

检索效率是O(logN)

下图为典型的红黑二叉树：



#### 30.LinkedList和ArrayList的区别是什么?

1. ArrayList是基于数组实现，LinkedList是基于链表实现
2. ArrayList在查找时速度快，LinkedList在插入与删除时更具优势

## Map集合

**java为数据结构中的映射定义了一个接口java.util.Map;它有四个实现类,分别是HashMap Hashtable LinkedHashMap 和TreeMap.**

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 是HashMap的一个子类，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,在Map 中插入、删除和定位元素，HashMap 是最好的选择。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。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.

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。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性。如果需要同步，可以用Collections的synchronizedMap方法使HashMap具有同步的能力。

Hashtable与HashMap类似，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。

在遍历的时候会比HashMap慢TreeMap能够把它保存的记录根据键排序，默认是按升序排序，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。

## 12、为什么要使用迭代器

迭代模式是访问集合类的通用方法，只要集合类实现了Iterator接口，就可以用迭代的方式来访问集合类内部的数据，Iterator访问方式把对不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

例如，如果不使用Iterator，遍历一个数组的方法是使用索引。这种方法的确定就是事先必须知道集合的数据结构，而且当我换了一种集合的话代码不可重用，要修改，比如我用set，就不能通过索引来遍历了。访问代码和集合是紧耦合，无法将访问逻辑从集合类和客户端代码中剥离出来，每一种集合类对应一种访问方式，代码不可重用。未解决以上问题，Iterator模式总是用同一种模式来遍历集合。

## 12、List有哪些子类，各有什么区别

List元素是有序的，元素可以重复。因为该集合体系有重复。

ArrayList底层的数据结构使用的是数组结构(数组长度是可变的50%延长)，查询很快，增删较慢，线程不同步。

LinkedList底层的数据结构是链表结构，查询慢增删快

Vector 底层是数组结构，线程同步(数组长度是可变的100%延长)，查询快增删慢，被ArrayList替代。

13、NIO相关，缓冲区、通道、selector。。。

Java NIO(New IO从Java 1.4开始)是一个可以替代标准Java IO API的IO API，Java NIO提供了与标准IO不同的IO工作方式。

Java NIO 由以下几个核心部分组成：

* Channels
* Buffers
* Selectors

**通道和缓冲区**：标准的IO基于字节流和字符流进行操作，而NIO是基于通道和缓冲区进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

**Non-blocking IO (非阻塞IO)**

Java NIO可以让你非阻塞的使用IO，例如，当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

**选择器**：选择器用于监听多个通道的事件(比如链接打开，数据到达)。因此，单个的线程可以监听多个数据通道。

Channel对应之前的流，Buffer不是新东西，Selector是因为NIO可以使用异步的非阻塞模式才加入的东西。

以前的流总是阻塞的，一个线程只要对它进行操作，其它操作就会被阻塞，也就相当于水管没有阀门，你伸手接水的时候，不管水到了没有，你就都只能耗在接水(流)上。

NIO的Channel的加入相当于接上了水龙头，虽然一个时刻也只能接一个水管的水，但依赖轮换策略，在水量不大的时候，各个水管流出来的水都可以得到妥善接纳，这个关键之处就是增加了一个接水工，也就是Selector，他负责协调，也就是看哪根水管有水了的话，在当前水管的水接到一定程度的时候，就切换一下：临时关上当前水龙头，试着打开另一个水龙头。

当其他人需要用水的时候，不用直接去接水，而是事前提了一个水桶给接水工，这个水桶就是Buffer。也就是其他人虽然也可能要等，但是不会在现场等，而是回家等，可以做其他事去，水接满了，接水工会通知他们。

在之前的编程中，I/O是用流的方式读取文件，所有的I/O都将视为单个的字节的移动。通过一个称为Stream的对象一次移动一个字节。NIO弥补了原来I/O的不足，它在标准Java代码中提供了高速、面向块的I/O。通过定义包含数据的块，以及通过以块的形式来处理这些数据，NIO不用使用本机代码就可以使用低级优化，这是原来的I/O包所无法做到的。

#### 流与块的比较

原来的I/O库和NIO最重要的区别就是数据打包和传输的方式，原来的I/O以流的方式处理数据，而NIO以块的方式处理数据。

面向流的I/O系统一次一个字节的处理数据，一个输入流产生一个字节的数据，一个输出流产生一个字节的数据。

一个面向块的I/O系统以块的形式处理数据。每一个操作都在一步中产生或者消费一个数据块。按块处理数据比按字节处理数据要快得多，即便它没有面向流的I/O那样的简单性。

#### 通道和缓冲区

通道和缓冲区是NIO中的核心对象，几乎在每一个I/O操作中都要使用它们。

通道是对原I/O包中的流的模拟。到任何目的地或来自任何地方的所有数据都必须通过一个Channel对象。一个Buffer实质上是一个容器对象。发送给一个通道的所有对象都必须首先存放到缓冲区中；同样的，从通道中读取任何的数据都必须首先读取到缓冲区里。

#### 什么是缓冲区？

Buffer是一个对象，它包含一些要写入或者刚读出的数据。 在 NIO 中加入Buffer对象，体现了新库与原 I/O 的一个重要区别。在面向流的  I/O 中，您将数据直接写入或者将数据直接读到Stream对象中。在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的。在写入数据时，它是写入到缓冲区中的。任何时候访问 NIO 中的数据，您都是将它放到缓冲区中。缓冲区实质上是一个数组。通常它是一个字节数组，但是也可以使用其他种类的数组。但是一个缓冲区不仅仅是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。

#### 缓冲区类型

最常用的缓冲区类型是ByteBuffer。一个ByteBuffer可以在其底层字节数组上进行 get/set 操作(即字节的获取和设置)。ByteBuffer不是 NIO  中唯一的缓冲区类型。事实上，对于每一种基本 Java 类型都有一种缓冲区类型：

ByteBuffer

CharBuffer

ShortBuffer

IntBuffer

LongBuffer

FloatBuffer

DoubleBuffer

每一个Buffer类都是Buffer接口的一个实例。 除了ByteBuffer，每一个 Buffer 类都有完全一样的操作，只是它们所处理的数据类型不一样。因为大多数标准 I/O 操作都使用ByteBuffer，所以它具有所有共享的缓冲区操作以及一些特有的操作。

#### 什么是通道？

Channel是一个对象，可以通过它读取和写入数据。拿 NIO 与原来的 I/O 做个比较，通道就像是流。正如前面提到的，所有数据都通过 Buffer 对象来处理。您永远不会将字节直接写入通道中，相反，您是将数据写入包含一个或者多个字节的缓冲区。同样，您不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。

#### 通道类型

通道与流的不同之处在于通道是双向的。而流只是在一个方向上移动(一个流必须是InputStream或者OutputStream的子类)，而通道可以用于读、写或者同时用于读写。

#### NIO中的读和写

读和写是 I/O 的基本过程。从一个通道中读取很简单：只需创建一个缓冲区，然后让通道将数据读到这个缓冲区中；写入也相当简单：创建一个缓冲区，用数据填充它，然后让通道用这些数据来执行写入操作。

**public** **class** NioTest {

**public** **static** **void** main(String[] args) **throws** IOException {

String inFile = "D:\\北京欢迎你.txt";

String outFile = "D:\\out.txt";

FileInputStream inStreamFile = **new** FileInputStream(inFile);

FileOutputStream outStreamFile = **new** FileOutputStream(outFile);

FileChannel inChannel = inStreamFile.getChannel();

FileChannel outChannel = outStreamFile.getChannel();

ByteBuffer buffer = ByteBuffer.*allocate*(4018);

**int** a = 0;

**while**(**true**) {

buffer.clear();

**int** i = inChannel.read(buffer);

**if** (i == -1) {

**break**;

}

buffer.flip();

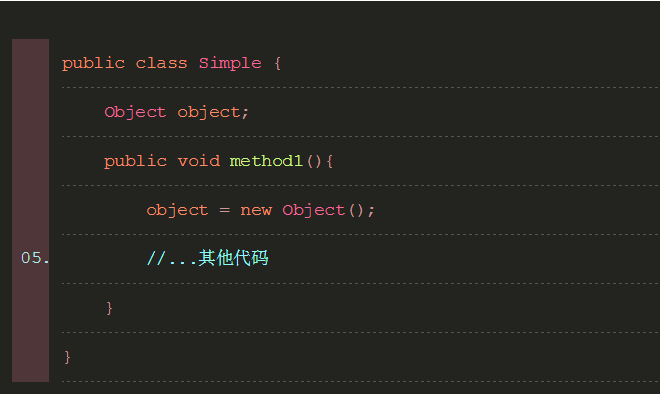
outChannel.write(buffer);

}

}

}

## 14、内存泄露，举个例子



这里的object实例，其实我们只希望它作用于method1()方法中，且其他地方不会再用到它，但是当method1()方法执行完成之后，object的内存并不会马上回收，只有在Simple类创建的对象被释放后才会被释放，严格的说，这就是一种内存泄漏。解决方法可以将object变量作为method1()方法的局部变量，或者在方法最后加上object = null;

## 15、OOM是怎么出现的，有哪几块JVM区域会产生OOM，如何解决

## 16、Java里面的观察者模式实现

转到设计模式面试题

## 17、单例实现(我一般用enum写，不容易被挑毛病)

最安全的写法是用单例模式实现的：

先来说下Java枚举的用法：

首先，枚举类似类，一个枚举可以拥有成员变量、成员方法、构造方法。先来看枚举最基本的用法：

public enum EnumEg {  
 *A*,*B*,*C*,*D*;  
}

创建enum时，编译器会自动为我们生成一个继承自java.lang.Enum的类，我们上面的enum可以简单看作：

class EnumEg extends Enum {  
 public static final EnumEg *A*;  
 public static final EnumEg *B*;

…  
}

对于上面的例子，我们可以把EnumEg看作一个类，而把A,B,C,D看作类EnumEg的实例。

当然这个构建的实例的过程不是我们做的，一个enum的构造方法限制是private的，也就是不允许我们在外部调用。

在EnumEg中，我们可以定义变量以及方法，看下面的代码：

public enum EnumEg {  
 *A*,*B*,*C*,*D*;  
 static int *value*;  
 public static int getValue () {  
 return *value*;  
 }  
 String type;  
 public String getType () {  
 return type;  
 }  
}

上面提到A,B,C,D是枚举类EnumEg的实例，那么我们调用方法getType()就可以通过这样调用EnumEg.A.getType()。

另外，每个实例可以实现自己的实例方法来覆盖原本方法，实现属于自己的定制。

public enum EnumEg {  
 *A*{  
 public String getType () {  
 return type;  
 }  
 },*B*,*C*,*D*;  
 static int *value*;  
 public static int getValue () {  
 return *value*;  
 }  
 String type;  
 public String getType () {  
 return type;  
 }  
}

除此之外，我们还可以添加抽象方法来强制各个实例来实现自己的逻辑：

public enum EnumEg {  
 *A*{  
 public String getType () {  
 return type;  
 }  
 },*B*{  
 public String getType () {  
 return type;  
 }  
 },*C*{  
 public String getType () {  
 return type;  
 }  
 },*D*{  
 public String getType () {  
 return type;  
 }  
 };  
 static int *value*;  
 public static int getValue () {  
 return *value*;  
 }  
 String type;  
 public abstract String getType ();  
}

有了上面的基础，我们来看下用枚举来实现单例

public enum SomeThing {  
 *INSTANCE*;  
 private Resource instance;  
 SomeThing () {  
 instance = new Resource();  
 }  
 public Resource getInstance () {  
 return instance;  
 }  
}

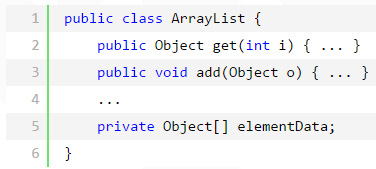
SomeThing.INSTANCE.getInstance()来获取实例。

下面来看下，为什么这样可以保证单例：

## 19、Java泛型机制，泛型机制的优点，以及类型变量

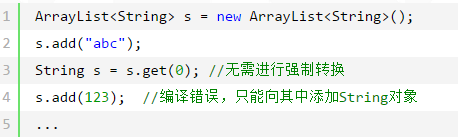
**1、引入泛型机制的原因**

我们都知道ArrayList这个集合可以保存各种类型的对象，在jdk5之前其源码设计如下：



这种也可以操作各种类型的对象，都将它们转换为Object类型。但是基于继承的泛型实现会带来问题：调用get()方法都要进行强制类型转换操作，比较麻烦。

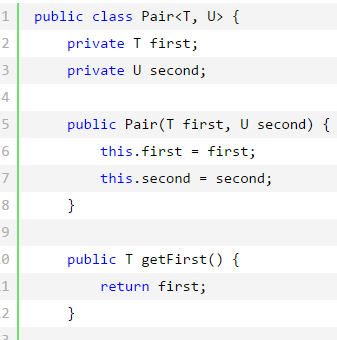
所以从jdk5开始，ArrayList在使用时可以加上一个类型参数，这个类型参数用来指明ArrayList中的元素类型。



这样虚拟机将会帮我们完成强制类型转换。

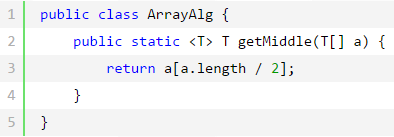
**2、泛型类**

所谓泛型类就是一个有一个或多个类型参数的类，比如，



**3、泛型方法**

所谓泛型方法，就是带有类型参数的方法，既可以定义在泛型类中，也可以定义在普通类中，比如，



**4、类型变量的限定**

比如，限定某个类型参数只能为某个类的子类或者为只能实现了某个接口的类。相关语法如下：



**5、深入理解泛型类的实现**

实际上，从虚拟机的角度看，不存在“泛型”概念，它其实还是用Object来实现的，只不过虚拟机为我们完成了强制类型转换。

**7、类型通配符**

Box<? extends Integer>类型通配符上限，类型通配符下限Box<? super Number>

## 20、Java集合框架源码

LinkedHashSet ，TreeSet(底层是用TreeMap)， TreeMap

## 21、synchronized关键字原理

* 修饰普通方法
* 修饰静态方法
* 修饰代码块

synchronized是JVM实现的一种锁，其中锁的获取和释放分别是monitorenter和monitorexit指令，该锁在实现上分为了偏向锁、轻量级锁和重量级锁，其中偏向锁在jdk1.6是默认开启的，轻量级锁在多线程竞争的情况下会膨胀成重量级锁，有关锁的数据都保存在对象头中

对同步代码块进行反编译：



**synchronized修饰代码块**：每个对象都有一个监视器锁(monitor)，当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

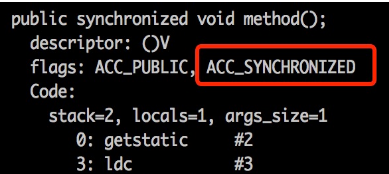
1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

指令执行时monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者，其他被这个monitor阻塞的线程可以尝试去获取这个monitor的所有权。

其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的方法或代码块中才能调用wait/notify方法。

**同步方法的反编译结果**：



方法的同步并没有通过monitorenter和monitorexit来完成，不过相对于普通方法其常量池中多了ACC\_SYNCHRONIZED标识符：当方法调用时，调用指令将会检查方法的ACC\_SYNCHRONIZED访问标志是否被设置，如果被设置了，执行线程将先获取monitor，获取成功后才能执行方法体，方法执行完后再释放monitor。

## ReentrantLock的实现原理，AbstractQueuedSynchronizer的实现原理

ReentrantLock是基于AQS实现的，AQS的基础又是CAS。

ReentrantLock的锁的非公平获取是通过判断状态是否为0来实现的，当然表示状态的变量是通过volatile修饰的，来保证在各个线程之间的可见性。这样每个线程都判断自己的状态是不是为0，如果为0，就将当前线程获得锁，如果当前状态已经获得锁那么就在状态上继续加上给定的值(一般加1).

非公平获取的代码实现：

final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0) // overflow  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}

从else if()上也可以判断ReentrantLock的锁也是可重入的。

ReentrantLock中的公平锁(获取一个锁是按照请求的顺序得到的)的实现：首先调用tryAcquire(int acquires)方法来尝试获取锁，如果当前线程是第一个节点的线程那么让当前线程获得锁或者当前线程已经获得锁，那么该方法返回true；否则返回false(表示没有获得锁)。当返回false之后，接着调用acquireQueued(addWaiter(Node.EXCLUSIVE),arg)，其中执行addWaiter()方法的作用就是新创建持有当前线程的节点，并加入到线程队列尾部。然后调用acquireQueued()方法来让要获取锁的线程一直尝试判断线程所在节点是不是头结点。

非公平锁的不同在于，刚开始的时候不管是不是头结点，先检测state是不是0，如果不是0那么将线程加入队列中，然后一直判断state是不是0.

所以公平锁与非公平锁在实现的区别就是：公平锁会一直检测当前线程所在节点是不是头结点；而非公平锁会一直检测state是不是0，如果是0，就争抢将状态变为1.

公平锁与非公平锁的unLock()方法的实现是一样的，都是将当前状态减去一个值，如果状态变为0，那么就放弃当前线程。

## ReentrantReadWriteLock的原理

多个线程可以同时读，只有一个线程能够同时写

#### ConcurrentHashMap的实现原理

基于分段锁，对每一个Segment也就是链加锁

jdk1.8中对该类增加了红黑树。

size()方法是如何实现的：每put一个元素size就加1，每remove一个元素size就减1.

## 22、concurrent包

这个包中自带了内置的线程池

## 23、preparedstatement底层如何实现？

1、代码可读性和可维护性

2、尽可能提高性能

preparestatement为预编译语句。

每一种数据库都会对预编译语句进行最好的性能优化，因为预编译语句有可能被重复调用，所以语句在被DB的编译器编译后的执行代码缓存下来，那么下次调用时只要是相同的预编译语句就不需要编译，只要将参数传入编译过的语句

执行代码中就会得到执行。对于整个DB中，只要预编译的语句语法和缓存中匹配，那么在任何时候不需要再次编译就能直接运行。而statement语句中，即使是相同的操作，由于每次操作的数据不同所以使整个语句相匹配的机会很小，几乎不太可能匹配。

3、最重要的一点是极大的提高了安全性

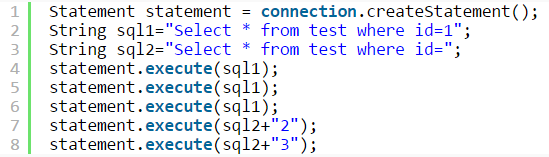
#### 原理分析

先来看下statement和preparedstatement的执行过程，一个sql语句执行过程中，将经历如下过程：

1. 传输sql给数据库
2. 数据库验证并解析sql
3. 计算Access Plan。数据库会检测index、statistics来给出最优的访问计划。
4. 根据访问计划进行检索，返回数据

在上面步骤中，第3步是非常耗时的。因此，为了提高性能，数据库会缓存执行语句以及其Access Plan。这被称为statement cache。在statement cache中，sql语句本身为key，access plan为value。当相同的sql语句被发送过来时，数据库会使用缓存中的access plan来节省CPU时间。

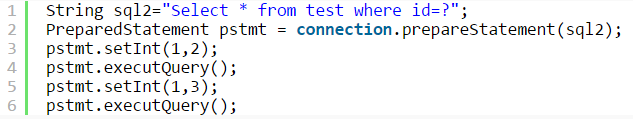
下面是statement执行代码：



sql1在第一次执行的时候，需要计算执行计划。但在之后的执行过程中，会使用缓存中的执行计划。因此效率会提高。

sql2却每次都在变化，在cache中，key为整个sql语句，所以每次sql2都无法命中cache，即使参数不同也必须重新检查语法和计算执行计划，效率降低。

PreparedStatement的存在是为了避免sql2的劣势，代码如下：



preparedstatement在创建的时候，会将参数化的语句发送给数据库，进行语法检测和执行计划计算。cache中的key将是参数化语句。preparedstatement的生命周期是在一个数据库中的connection中有效。

## 24、类之间有哪些关系？

继承关系、实现关系、依赖关系、关联关系、聚合关系、组合关系

## 26、GC太频繁如何处理？

## 27、线程池

大体思路就是，假如有15个线程，一个一个添加到线程池(线程池的大小假设为5)中去执行，如果超过5，就将多出的放到缓存队列中等待执行。

MyTask类：

public class MyTask implements Runnable {  
 private int taskNum;  
  
 public MyTask (int num) {  
 taskNum = num;  
 }  
  
 @SuppressWarnings("static-access")  
 public void run() {  
 // *TODO Auto-generated method stub* System.*out*.println("正在执行task" + taskNum);  
 try {  
 Thread.*currentThread*().*sleep*(4000);  
 } catch (InterruptedException e) {  
 // *TODO Auto-generated catch block* e.printStackTrace();  
 }  
 System.*out*.println("task" + taskNum + "执行完毕");  
 }  
}

Test类：

public class Test {  
 public static void main(String[] args) {  
  
 ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 200, TimeUnit.MILLISECONDS, new ArrayBlockingQueue<Runnable>(5));  
 for (int i = 0; i < 15; i++) {  
 MyTask myTask = new MyTask(i+1);  
 executor.execute(myTask);  
 System.*out*.println("线程池中的线程数目：" + executor.getPoolSize() + ", 队列中等待执行的任务数目：" + executor.getQueue().size() + "，已经执行完的任务数目：" + executor.getCompletedTaskCount());  
 executor.shutdown();  
 }  
 }  
}

ThreadPoolExecutor是线程池的最核心的类，它的构造方法中的各个参数的含义如下：

* corePoolSize：核心池的大小。在创建线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从名字可以看出，是预创建线程的意思，即在任务没有到来之前就创建corePoolSize个线程或者一个线程。当线程池中的线程数量达到corePoolSize之后，就会把到达的任务放到缓存队列当中。
* maximumPoolSize：线程池最大线程数，它表示在线程池中，最多能创建多少个线程。
* keepAliveTime：表示线程没有任务执行时最多保持多久会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程数不大于corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，即使线程数不大于corePoolSize，keepAliveTime也会超时。
* unit：参数keepAliveTime的时间单位，有7种取值，天，小时，分，秒，毫秒，微妙，纳秒



* workQueue：阻塞队列，用于存储等待执行的任务，一般使用LinkedBlockingQueue和SynchronousQueue。线程池的排队策略和BlockingQueue有关。
* threadFactory：线程工厂，主要用来创建线程
* handler：表示当拒绝任务时的策略，有四种取值：



executor()是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务

submit()是用来向线程池提交任务的，它实际上还是调用的executor()方法，只不过它通过Future来获取任务执行结果。

shutdown()和shutdownNow()是用来关闭线程池的。

#### 线程池状态



* 当创建线程池后，初始时，线程池处于running状态
* 如果调用了shutdown()方法，则线程池处于shutdown状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕
* 如果调用shutdownNow()方法，则线程池处于stop状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务
* 当线程池处于shutdown或者stop状态，并且所有线程都已销毁，任务缓存队列清空或执行结束后，线程池被设置为terminated状态

#### 任务的执行

举个例子：

假设有一个工厂，工厂里有10个工人，每个工人同时只能做一个任务。

因此当10个工人中有工人是空闲的，来了任务就分配给空闲的人做。

当10个工人都有任务时，如果还来了任务，就把任务进行排队等待；

如果新任务增长的速度远远大于工人处理任务的速度，那么就要采取措施，比如再招4个工人来

然后将任务也分配给这4个人做。

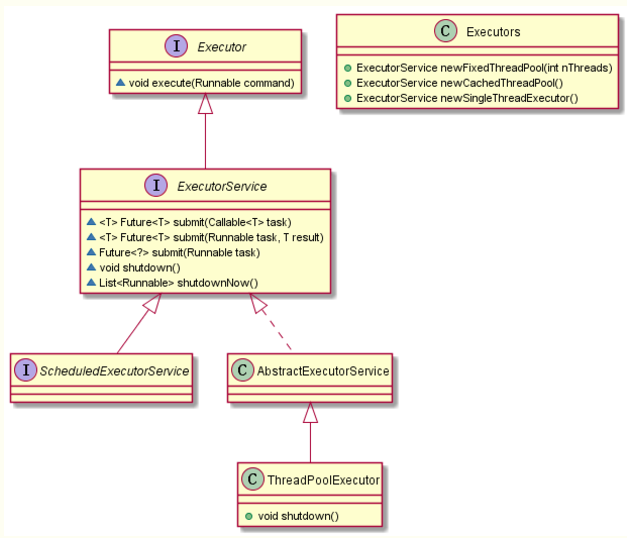
如果说14个工人做任务的速度还是不够，那么就要考虑不再接受新的任务或者抛弃之前的任务。

如果14个人当中有空闲，或者新增任务的速度比较缓慢，那么可能就考虑辞掉4个工人。

在这个例子中corePoolSize就是10，maximumPoolSize就是14

## Executor、ExecutorService和Executors

UML图：



* ExecutorService接口继承了Executor接口
* Executor接口定义了execute()方法用来接收一个Runnable接口的对象，而ExecutorService接口中的submit()可以接收Runnable和Callable接口的对象。
* Executor中的execute()方法不返回任何结果，而ExecutorService中的submit()可以通过一个Future对象返回运算结果。
* ExecutorService还提供用来控制线程池的方法。比如：调用shutDown()方法来终止线程池。
* Executors类提供工厂方法用来创建不同类型的线程池。

1. newFixedThreadPool (int nThreads)

创建一个可重用固定数量线程的线程池，以共享的无界队列方式来运行这些线程。在任何点，大多数nThreads个线程会处于处理任务的活动状态。在所有线程都处于活动状态时如果有任务被提交了，他们将一直等待直到有线程可用了。如果有任何线程在关闭之前执行失败了，那么一个新线程将代替它执行后面的任务(如果需要的话)。线程池中的线程会一直存在直到线程池关闭。

示例程序如下：

public class ThreadTemp implements Runnable {  
 private int num;  
 public ThreadTemp(int num){  
 this.num = num;  
 }  
 @Override  
 public void run() {  
 System.*out*.println("Thread " + num + " started...");  
 try {  
 Thread.*sleep*(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("Thread " + num + " ended...");  
 }  
}

调用线程池执行：

public class Exercise {  
 public static void main(String[] args){  
 Executor executor = Executors.*newFixedThreadPool*(3);  
 for (int i = 0; i < 10; i++) {  
 ThreadTemp threadTemp = new ThreadTemp(i);  
 Thread thread = new Thread(threadTemp);  
 executor.execute(thread);  
 }  
 }  
}

输出结果：

Thread 0 started...

Thread 2 started...

Thread 1 started...

Thread 0 ended...

Thread 1 ended...

Thread 2 ended...

Thread 4 started...

Thread 3 started...

Thread 5 started...

Thread 5 ended...

Thread 6 started...

Thread 3 ended...

Thread 7 started...

Thread 4 ended...

Thread 8 started...

Thread 6 ended...

Thread 8 ended...

Thread 7 ended...

Thread 9 started...

Thread 9 ended...

从结果看出，开始共有3个线程同时执行，之后每结束一个就会新建一个，总是维持着3个的数量。

1. newSingleThreadExecutor ()

创建有单个线程的线程池

1. newCacheThreadPool()
2. newScheduledThreadPool(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

**使用Futurn、Callable以及线程池并行计算数组内各个整数的和**

基本思想就是将数组按照处理器的核心数平均分成几组，每组分给一个线程进行计算，计算完成之后将结果进行汇总得出总的和。

ConcurrentCalculater.java代码：

public class ConcurrentCalculator {  
 private ExecutorService exec = Executors.*newFixedThreadPool*(getCPUCoreNumber());  
 private int[] num;  
 public void setNum(int[] num){  
 this.num = num;  
 }  
 public int getCPUCoreNumber(){  
 return Runtime.*getRuntime*().availableProcessors();  
 }  
  
 public int getAllSum () throws ExecutionException, InterruptedException {  
 int group = num.length/getCPUCoreNumber() + 1; //按照CPU核心数将数组平均分组  
 int sum = 0;  
 for (int i = 0; i < num.length; i += group) {  
 int start = i;  
 int end = start + group - 1;  
 if (end > num.length - 1) {  
 end = num.length - 1;  
 }  
 ComputeArray computeArray = new ComputeArray(num, start, end);  
 Future future = exec.submit(computeArray);  
 sum += (Integer) future.get();  
 }  
 exec.shutdown();  
 return sum;  
 }  
}

主方法：

public class Main {  
 public static void main (String[] args) throws ExecutionException, InterruptedException {  
 ConcurrentCalculator calculator = new ConcurrentCalculator();  
 int[] num = {1,2,3,4,5,6,7,8,9,10};  
 calculator.setNum(num);  
 System.*out*.println(calculator.getAllSum());  
 }  
}

## Future与FutureTask

Executor就是Runnable和Callable的调度容器，Future就是对于具体的调度任务的执行结果进行查看，最为关键的是Future可以检查对应的任务是否已经完成，也可以阻塞在get()方法上一直等待任务返回结果。

FutureTask则是一个RunnableFuture<V>，即实现了Runnable又实现了Future这两个接口，另外它还可以包装Runnable和Callable，他可以通过Thread包装来直接执行，也可以提交给ExecuteService来执行，而且还可以通过get()返回执行结果，在线程体还没有完成的时候，主线程一直阻塞等待，执行完则直接返回结果。

FutureTask futureTask = new FutureTask(new ComputeArray(num, 1,3));  
Thread thread = new Thread(futureTask);  
thread.start();

ComputeArray.java

public class ComputeArray implements Callable {  
 private int start;  
 private int end;  
 private int[] num;  
  
 public ComputeArray(int[] num, int start, int end) {  
 this.num = num;  
 this.start = start;  
 this.end = end;  
 }  
 @Override  
 public Integer call() throws Exception {  
 System.*out*.println("start...");  
 int sum = 0;  
 for (int i = start; i <= end; i++) {  
 sum += num[i];  
 }  
 System.*out*.println("end...");  
 return sum;  
 }  
}

## 28、多线程和多任务

多任务是相对于操作系统而言的，指的是同一时间执行多个程序的能力。但是在只有一个核的CPU的条件下，要分片。

多线程是能同时执行同一个程序的不同部分。

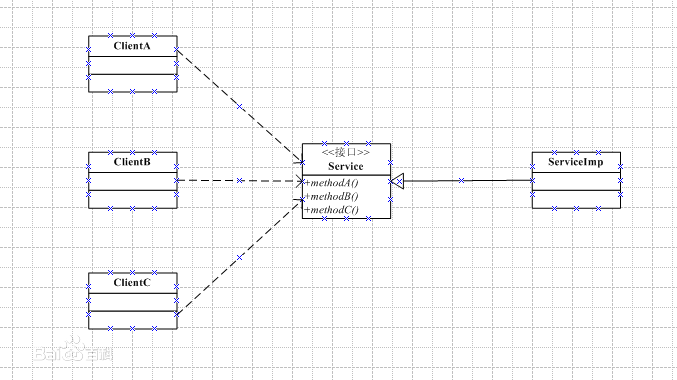
**在一个进程里线程的调度有抢占和非抢占的模式**。在抢占模式下，操作系统负责分配CPU时间给各个进程，一旦当前的线程使用完分配给自己的CPU时间，那么操作系统将中断当前正在执行的线程，将CPU分配给在等待队列的下一个线程。所以任何一个线程都不能独占CPU。在非抢占模式下，每个线程可以需要CPU多少就占用CPU多少时间。占用CPU的线程只有它主动释放CPU时，其他的线程才可以使用CPU。在有些操作系统里面这两种调度策略都会用到。非抢占策略在线程运行优先级一般时用到，而对于高优先级的线程多采用抢占式。

## 29、设计模式

## 30、如何实现接口分离？

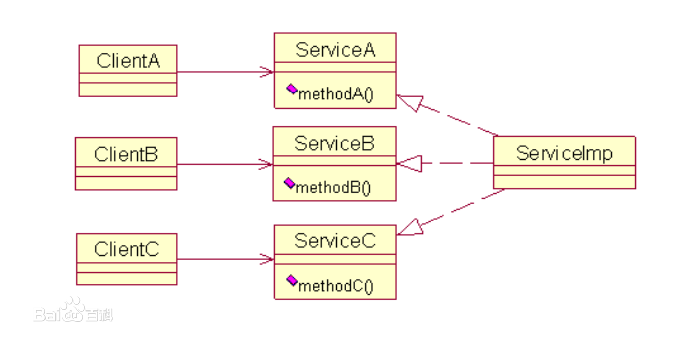
接口分离原则指在设计时采用多个与特定客户类有关的接口比采用一个通用的接口要好。即，一个类要给多个接口使用，那么可以为每个客户创建一个接口，然后这个类实现所有的接口；而不是只创建一个接口，其中包含所有客户类需要的方法，然后这个类实现这个接口。

没有使用接口分离原则：



优点：对每个客户类都有一个专用的接口，这个接口中只声明了与这个客户类相关的方法，而ServiceImpl类实现了所有的接口。如果ClientA要改变它所使用的接口中的方法，只需改动ServiceA接口和ServiceImpl类即可，ClientB和ClientC不受影响。

使用接口分离原则：



## 29、什么是虚拟内存？

## 30、动态代理的源码要看

代理的优点：可以隐藏委托类的实现、可以实现客户与委托类间的解耦，在不修改委托类代码的情况下能够做一些额外的处理。

先说静态代理：实现代码如下

先定义一个接口Sell，委托类Vendor和代理类BusinessAgent都实现这个接口

Sell接口：

public interface Sell {  
 void sell();  
 void ad();  
}

委托类Vendor：

public class Vendor implements Sell {  
 @Override  
 public void sell() {  
 System.*out*.println("In sell method");  
 }  
 @Override  
 public void ad() {  
 System.*out*.println("ad method");  
 }  
}

代理类BusinessAgent，内部有一个委托类的引用：

public class BusinessAgent implements Sell {  
 private Vendor mVendor;  
 public BusinessAgent (Vendor vendor) {  
 this.mVendor = vendor;  
 }  
 @Override  
 public void sell() {  
 mVendor.sell();  
 }  
 @Override  
 public void ad() {  
 mVendor.ad();  
 }  
}

上面就是一个简单的静态代理。

下面要增加一个需求：委托类的代码不能动(这里是模拟使用别人的已经编好的代码)，添加一个过滤功能，实现只卖货给特定的人比如大学生。通过静态代理可以在代理类BusinessAgent中添加过滤代码：

public void sell() {  
 if (isColleage == true) {  
 mVendor.sell();  
 }  
}

#### 什么是动态代理

代理类在程序运行时创建的代理方式称为动态代理。也就是说代理类是动态生成的，不像静态代理，代理类在编译期就生成了。也就是说代理类并不是在java代码中定义的，而是在运行时根据我们在java代码中的“提示”动态生成的。**动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类的函数。**

现在有一个新需求：在执行委托类中的方法之前输出“before”，执行之后输出“after”。使用静态代理实现这一需求：

@Override  
public void sell() {  
 System.*out*.println("before");  
 mVendor.sell();  
 System.*out*.println("after");  
}  
@Override  
public void ad() {  
 System.*out*.println("before");  
 mVendor.ad();  
 System.*out*.println("after");  
}

当方法比较少时，这样修改还可以接受，但是当类很多时，这种修改方式就显得很繁琐很不明智了。

比较明智的方法是在委托类和代理类中间添加一个中介类。这个中介类被要求实现一个接口InvocationHandler接口，这个接口的定义如下：

public interface InvocationHandler {  
 Object invoke (Object proxy, Method method, Object[] args);  
}

当我们在调用代理类中的方法时，这个“调用”会转送到这个invoke方法中，proxy是代理类对象实例，method是被调用的代理类的方法，args为方法的参数。这样我们对代理类中所有方法的调用都会转为对invoke方法的调用，这样我们可以在invoke方法中添加统一的处理逻辑

public class DynamicProxy implements InvocationHandler {  
 private Object obj;  
 public DynamicProxy (Object obj) {  
 this.obj = obj;  
 }  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Exception {  
 System.*out*.println("before");  
 Object result = method.invoke(obj, args);  
 System.*out*.println("after");  
 return result;  
 }  
}

#### 动态生成代理类

public class Main {  
 public static void main (String[] args) {  
 //创建中介类实例  
 DynamicProxy inter = new DynamicProxy(new Vendor());  
 //加上这句将会产生一个$Proxy0.class文件，这个文件即为动态生成的代理类文件  
 System.*getProperties*().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");  
 //获取代理类实例sell  
 Sell sell = (Sell) (Proxy.*newProxyInstance*(Sell.class.getClassLoader(), new Class[]{Sell.class},inter));  
 sell.sell();  
 sell.ad();  
 }  
}

## 31、进程间通信机制

#### 管道

管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子关系。

#### 有名管道

也是半双工通信方式，但是它允许无亲缘关系间通信

#### 信号量

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

#### 信号

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

#### 消息队列

消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

#### 共享内存

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

#### 套接字

套接字也是一种进程间通信方式，与其他方式不同的是，它可用于不同主机间的进程通信。

## 竞态条件

竞态条件，通俗地说，就是线程A需要判断一个变量的状态，然后根据这个变量的状态来执行某个操作。在执行这个操作之前，这个变量的状态可能会被其他线程修改。

如下代码：

public class Exercise {  
 private Object instance = null;  
 public Object getInstance () {  
 if (instance == null) {  
 instance = new Object();  
 }  
 return instance;  
 }  
}

假定线程A和线程B同时执行判断条件。A看到instance为空会创建一个实例，B同样需要判断instance是否为空。这是的instance是否为空，要取决于不可预测的时序，包括线程的调度方式，以及A需要花多长时间来初始化并设置instance，比如：



使用同步可以解决这种问题。

## 33、三次握手和四次挥手

#### 三次握手

首先要熟悉TCP的报文格式：



SYN(synchronous)建立连接标志位，1代表请求连接

ACK(acknowledgement确认 )确认标志位，1代表确认号有效

FIN(finish结束)

RST(reset重置)

URG(urgent紧急指针)

PSH(push传送)

Sequence number(seq序列号)

Acknowledge number(ack确认号)

三次握手过程图示：



过程描述：

服务端处于监听状态，客户端发起连接

第一次握手：A向B发送一个数据包，其中的数据为SYN=1表示请求建立连接，初始化序列号isn=200(这个数是随机产生的)

第二次握手：如果B同意建立连接，那B向A发送一个数据包，其中SYN=1，初始化序列号=500(也是随机产生的)，同时确认标志ACK=1，确认号ack=200+1

第三次握手：C收到数据以后检验ACK==1，ack==201，如果正确，发送一个数据包，其中ACK=1，ack=501.B收到以后检验ACK==1，ack==501，如果正确，连接建立成功。

在这里讨论下为什么不能是两次握手？：首先第二次握手表示B告诉A，“你发送的信息我已经收到”。但是B给A的连接B收到没收到并没说，所以这时候还需要A给B发送信息，“你发送的信息我也收到了”。

不采用更多次是因为3次就足够了，避免浪费资源。或者也可以理解成将中间的一次分成2次，三次握手将其进行了合并。

## 四次挥手

任意一方都可以执行close来触发

过程图示：



由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上不再接收数据了，但是仍然可以发送数据，直到另一方也发送FIN。首先进行关闭的一方将进行主动关闭，而另一方将进行被动关闭。流程描述如下：

第一次挥手：客户端发送一个FIN，用来关闭客户端到服务器的数据传送。

第二次挥手：服务器收到一个FIN后，发送一个ACK给客户端，确认序号为收到序号+1

第三次挥手：服务器发送一个FIN，用来关闭与客户端的传送。

第四次挥手：客户端收到FIN，接着发送ACK给服务器，确认序列号为收到序列号+1，服务器进入closed状态，完成四次挥手。

上面是一个主动关闭一个被动关闭。实际上两边可能同时发起主动关闭。

现在来讨论为什么四次挥手中间的两次不能合并：关闭连接时当收到对方的FIN报文时，仅仅表示对方不在发送数据了，但是还能接收数据，乙方也未必全部数据都发送给对方了，所以乙方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，乙方ACK和FIN都会分开发送。三次握手中，ACK和SYN是同时发送的。

## 34、滑动窗口

## 

## 35、最安全的单例模式

利用枚举实现单例模式

电脑上访问一个网页的过程是什么样的？

拥塞避免/慢启动/快重传/快速恢复(所有公司都要问)

回退N帧协议

数据库范式

什么样的数据库设计是符合要求的(从范式的角度)

理论上说达到第三范式是符合要求的但是一般生产环境下为了数据查询方便，数据会有一定的冗余，也就是说一般达到第二范式即可。

第一范式：字段不可分

第二范式：非主属性必须完全函数依赖于码，即如果码的子集能唯一确定一个非主属性的话，那该表就不符合第二范式。

第三范式：消除了第二范式中的传递函数依赖

MySQL中的索引

## 连接池的原理是什么

## 连接池中的连接是长连接还是短连接？为什么？

长连接就是长时间保持客户端与服务器之间的连接状态：连接🡪数据传输🡪保持连接🡪数据传输🡪保持连接……🡪关闭连接

这就要求长连接在没有数据通信时，定时发送数据包，以维持连接状态，短连接在没有数据传输时直接关闭就行了。

通常的短连接操作步骤是：连接🡪数据传输🡪关闭连接

#### 什么时候用长连接、短连接？

长连接主要用在少数客户端与服务器的频繁通信，因为这时候如果用短连接频繁通信常会发生Socket请求错误，并且频繁创建Socket也是对资源的浪费。

连接池中的连接是基于什么协议的连接？为什么？

怎么建立TCP连接，怎么断开TCP连接？

为什么要三次握手，为什么要四次挥手？

剑指offer上的算法要理解并能自己写出来

项目中遇到的最大的困难是什么？

反转链表、冒泡排序、生产者消费者

========================J2SE基础=======================

作者：Xoper.ducky

链接：https://www.nowcoder.com/discuss/3043?type=2&order=0&pos=25&page=7

来源：牛客网

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。

============================JVM=============================

作者：Xoper.ducky

链接：https://www.nowcoder.com/discuss/3043?type=2&order=0&pos=25&page=7

来源：牛客网

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。

=========================操作系统=============================

作者：Xoper.ducky

链接：https://www.nowcoder.com/discuss/3043?type=2&order=0&pos=25&page=7

来源：牛客网

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。

=========================TCP/IP===============================

作者：Xoper.ducky

链接：https://www.nowcoder.com/discuss/3043?type=2&order=0&pos=25&page=7

来源：牛客网

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。

=====================================数据结构与算法===================================

作者：Xoper.ducky

链接：https://www.nowcoder.com/discuss/3043?type=2&order=0&pos=25&page=7

来源：牛客网

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

算法面试80题

=========================有什么要问的？==============================

作者：Xoper.ducky

链接：https://www.nowcoder.com/discuss/3043?type=2&order=0&pos=25&page=7

来源：牛客网

1.

贵公司一向以XXX著称，能不能说明一下公司这方面的特点？

2.

贵公司XXX业务发展很好，这是公司发展的重点么？

3.

对技术和业务怎么看？

4.

贵公司一般的团队是多大，几个人负责一个产品或者业务？

5.

贵公司的开发中是否会使用到一些最新技术？

6.

对新人有没有什么培训，会不会安排导师？

7.

对Full?Stack怎么看？您觉得我们这些程序员有必要向这方面发展吗？

8.

你觉得我有哪些需要提高的地方？

9.

是否看好人工智能呢？

10.

您现在在研究什么技术？您比较喜欢哪一方面？

## 重载重写

重载：方法名相同，参数列表不同

重写：返回类型相同，方法名相同，参数列表相同，也就是方法签名要完全相同

## wait、notify为什么不能用在Thread类里面

## 使用new和newInstance创建对象的区别

* newInstance()只能调用无参构造函数，new没有限制
* newInstance()在使用之前要求类首先要被加载并且连接了，恰好Class.forName()正好完成了上面两步，这个方法调用了启动类加载器(Bootstrap)；而new可以没有被加载

## 使用forName和loadClass加载类的区别

Class.forName(“XX.XX.XX”)这个调用的是Class.forName(“”,true,ClassLoader)。第二个参数表示装载类的时候是否初始化类，为true表示初始化类，所以调用Class.forName(“XX.XX.XX”)就会加载被static修饰的属性、代码块和方法。

loadClass(“XX.XX.XX”)表示加载类的时候不初始化类，所以也就不会执行static修饰的属性、代码块和方法。

## 注解

#### 什么是注解？

注解就是元数据，即一种描述数据的数据。所以，可以说注解就是源代码的元数据。

#### 注解的好处

例如用@Override注解标记代码，即使不使用，程序也能正确执行。@Override告诉编译器这个方法是一个重写方法，如果父类中不存在该方法，编译器便会报错，提示该方法没有重写父类中的方法。如果不小心拼写错误，例如将toString()写成了toStrring()，而且也没有使用@Override注解，那程序依然能编译运行，但运行结果会和期望的不相同。

#### 为什么要引入注解？

## interrupt与isInterrupt

yelid()、join()

## String为什么要设计为不可变

 只有当字符串是不可变的，字符串池才有可能实现。字符串池的实现可以在运行时节约很多heap空间，因为不同的字符串变量都指向池中的同一个字符串。但如果字符串是可变的，那么String interning将不能实现([String interning](https://link.zhihu.com/?target=http%3A//en.wikipedia.org/wiki/String_interning)是指对不同的字符串仅仅只保存一个，即不会保存多个相同的字符串)，因为这样的话，如果变量改变了它的值，那么其它指向这个值的变量的值也会一起改变。

 如果字符串是可变的，那么会引起很严重的安全问题。譬如，数据库的用户名、密码都是以字符串的形式传入来获得数据库的连接，或者在socket编程中，主机名和端口都是以字符串的形式传入。因为字符串是不可变的，所以它的值是不可改变的，否则黑客们可以钻到空子，**改变字符串指向的对象的值，造成安全漏洞。**

 因为字符串是不可变的，所以是多线程安全的，同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。

 类加载器要用到字符串，不可变性提供了安全性，以便正确的类被加载。譬如你想加载java.sql.Connection类，而这个值被改成了myhacked.Connection，那么会对你的数据库造成不可知的破坏。

 因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。

## 不开辟新的变量空间，原地交换两个变量

int a = 5, b = 10;

a = a+b;

b = a-b;

a = a-b;

## Java回调机制

先来上一段代码：

先定义一个接口，

public interface MyInterface {  
 void sayYourName();  
}

接着，定义一个类，其中一个方法的参数用接口来声明，

public class MyClass {  
 public void sayYourName (MyInterface myInterface) {  
 System.*out*.println("start");  
 myInterface.sayYourName();  
 System.*out*.println("finished");  
 }  
}

接着，我们在MainActivity中声明这个类，并调用其中的方法，

public class MainActivity {  
 public void onCreate () {  
 MyClass myClass = new MyClass();  
 myClass.sayYourName(new MyInterface() { //这里直接可以new接口，后面直接跟实现方法.其实这是内部类的用法  
 @Override  
 public void sayYourName() {  
 System.*out*.println("CallBack");  
 }  
 });  
 }  
}

这其实就是回调函数的用法。

## 为什么要使用内部类？

#### 内部类的作用

1. **内部类可以很好的实现隐藏**：一般的非内部类，对于类的权限修饰符是不允许有private和protected，但内部类可以。
2. **内部类拥有外围类所有元素的访问权限**
3. **可以实现多重继承**
4. **可以避免修改接口而实现同一类中两种同名方法的调用**

例子1、实现隐藏

一般的非内部类的权限访问修饰符不能为private和protected，但内部类可以，所以我们能通过内部类来隐藏一些方法的实现。代码如下：

**public** **class** ClassExample {

**private** **class** InnerClass **implements** MyInterface {

**public** **void** increment() {

// **TODO** Auto-generated method stub

System.*out*.println("This is a test.");

}

}

**public** MyInterface getIn () {

**return** **new** InnerClass();

}

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

**new** ClassExample().getIn().increment();

}

}

getIn()返回一个MyInterface实例，这样我们将内部类声明为private权限，调用该实例的increment()方法，但是我们甚至不知道MyInterface实例的名字，所以实现了很好的隐藏。

例子2、内部类可以访问外围类的所有属性和方法

这一点很好理解。内部类也可以有自己的构造方法。

例子3、可以实现多继承

Java中没有多继承，但是可以通过内部类变相实现多继承。实现方法是，定义多个内部类，让每个内部类实现一个其他类。

例子4、可以避免修改接口而实现同一类中两种同名方法的调用

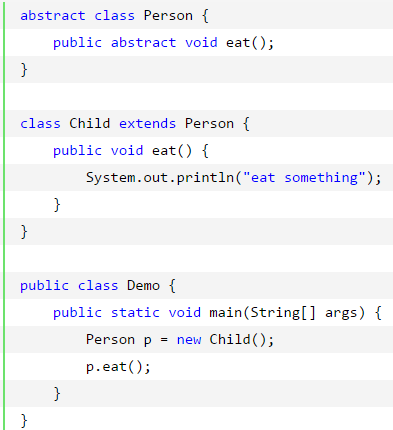
如果你的类要继承一个类还要实现一个接口，但是父类和接口有同名的方法怎么办？怎样区分它们？

可以让类继承父类，然后让内部类实现接口；或者反过来。这样就可以区分了。

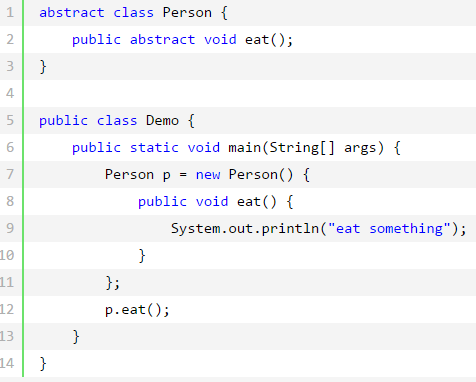
#### 什么是匿名内部类？

匿名内部类就是没有名字的内部类。正因为没有名字所以匿名内部类只能使用一次，它通常用来简化代码编写。匿名内部类的使用有一个前提条件：**必须继承一个父类或实现一个接口**。

有下述代码：



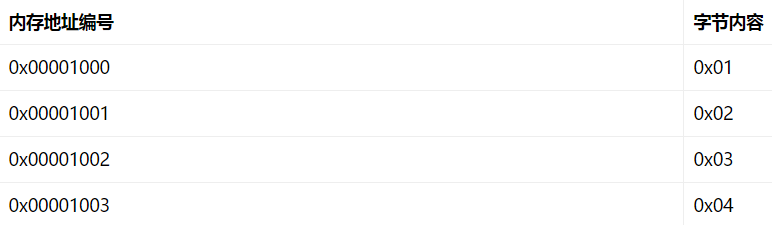
此处的Child只使用了一次，那么将其编写为独立的类岂不是很麻烦？这个时候就引入了匿名内部类。



也可以在接口上使用匿名内部类，与抽象类的用法一样。

## Java中采用的是大端序

高字节数据存放在低地址处，低字节数据存放在高地址处。例如，一个int类型数据占据4个字节，int value = 0x01020304。占据情况如下：



## Java中Comparator与Comparable有什么不同？

Comparable接口用于定义对象的自然顺序，而Comparator接口用于定义用户定制的顺序。Comparable总是只有一个，但是可以有多个comparator来定义对象的顺序。

## 静态导包

import static com…ClassName.\* 这里多了个static，还有类名后面多了.\*，意思是导入这个类类里面的静态变量和方法，在使用的时候不必加类名就能直接使用。

## 用原子操作可以实现无锁编程

## Java中指令重排的意义

JVM能根据处理器的特性，充分利用多级缓存，多核等进行适当的指令重排序，使程序在保证业务运行的同时，充分利用CPU的执行特点，最大的发挥CPU的性能。

HashMap中key可以为null，但是Hashtable中不能为null

## 共享锁(S锁)与排他锁(X锁)是怎样实现的

共享锁就是读锁，可以有多个线程同时控制state的值；排他锁就是写锁，只能有一个线程控制state的值。

## 什么是饥饿？

饥饿就是进程无法得到资源，所以无法进行下去，称为饿死，比较常见的就是在优先级调度中，不停的有高优先级的进程创建，导致其他进程无法获得CPU，从而饿死。

## 公平锁与非公平锁

首先要清楚一点，线程执行了但是不一定获得锁。

公平锁就是按照线程的执行的顺序依次获得锁。

非公平锁就是不按照线程执行的顺序，随机获得锁。

## 阻塞与非阻塞的区别

阻塞就是干不完不准返回，要一直等在那里。

非阻塞就是先把事情通知给你，我先去干别的事情，如果你干完了通知我就行了。

## Java序列化的过程

transit和static属性不能参与序列化，并且任何方法也不参与序列化。

父类能被序列化那么子类也能被序列化。

#### 序列化是干什么的？

简单说就是保存内存中各种对象的状态，并且可以把保存的对象的状态再读出来。Java本身提供了一种保存对象的机制，这就是序列化。

#### 什么情况下需要序列化？

1. 当你想把内存中的对象状态保存到一个文件中或者数据库中的时候
2. 当你想用套接字在网络上传输对象的时候
3. 当你想通过RMI传输对象的时候

#### 当对一个对象实现序列化时，究竟发生了什么？

#### java.io.Externalizable接口

该接口继承了java.io.Serializable接口，作用就是控制对象的序列化和反序列化。控制序列化就是有选择的序列化对象，而不是把对象的所有内容都序列化，并且可以在序列化的过程中储存其他非this对象包含的数据。

该接口定义了两个方法，writeExternal()在序列化时被调用，可以在该方法中控制序列化内容，readExternal方法在反序列化是被调用，可以在该方法中控制反序列化的内容。

示例代码如下：

public class UserInfo implements Externalizable {  
 public String name;  
 public String password;  
 public int age;  
  
 public UserInfo (String name, String password, int age) {  
 this.name = name;  
 this.password = password;  
 this.age = age;  
 }  
   
 @Override  
 //当序列化对象时，该方法自动调用  
 public void writeExternal(ObjectOutput out) throws IOException {  
 System.*out*.println("现在执行序列化方法");  
 Date date = new Date();  
 out.writeObject(date);  
 //只序列化name和password两个变量  
 out.writeObject(name);  
 out.writeObject(password);  
 }  
  
 @Override  
 //当反序列化对象时，该方法自动调用  
 public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  
 System.*out*.println("现在执行反序列化方法");  
 Date date = (Date) in.readObject();  
 System.*out*.println(date);  
 this.name = (String) in.readObject();  
 this.age = (Integer) in.readObject();  
 }  
}

## 什么是事务？

事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在某个操作中所做的所有更改都会被撤销。也就是事务具有原子性，一个事务中的一系列操作要么全部成功，要么一个都不做。

事务的结束有两种，当事务中的所有步骤全部成功执行时，事务提交。如果其中一个步骤失败，将发生回滚操作，撤销之前到事务开始时的全部操作。

#### 事务的ACID

事务具有四个特征：原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)和持续性(Durability)。

**原子性：**事务是数据库的逻辑工作单位，事务中包含的各操作要么都做要么都不做

**一致性**：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库发生中断，有一些事务尚未完成就被迫中断，这些未完成事务已经将一些数据写入物理数据库，这时数据库就处于一种不正确状态，或者说是不一致状态。

**隔离性**：一个事务的执行不能被其他事务所干扰。即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

**持续性**：也成永久性。指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其执行结果有任何影响。

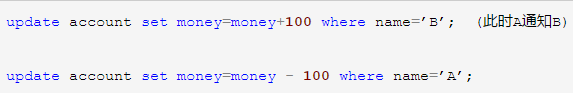
## 事务隔离

如果不考虑事务隔离会出现如下情况：

#### 脏读

是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

当一个事务正在多次修改某个数据，而在这个事务中这多次的修改还都未提交(事务未提交就相当于什么都没做)，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如，用户A向用户B转账100元，对应SQL语句如下：



当只执行第一条SQL时，B查看账户确实钱到账了，但是之后无论第二条SQL是不是执行，只要该事务不提交，则所有操作都将回滚，那么当B再次查看账户时发现钱其实没有到账。

#### 不可重复读

在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

不可重复度和脏读的区别：脏读是某一事务读取了另一事务未提交的脏数据，不可重复读是读取了前一事务提交的数据。

在某些情况下，不可重复读并不是问题，比如我们以最后查询到的结果为主。但是在一些情况下就可能发生问题。

#### 虚读(幻读)

幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据行，而这个数据项的数值还是为“1”并且提交给数据项。所以对于T1来说会发现还有一行没有修改。

幻读和不可重复读不同的是，不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体(比如统计修改的个数)。

#### MySQL中提供了4中隔离级别

1. Serializable(串行化)：可避免脏读、不可重复读、幻读的发生
2. Repeatable read(可重复读)：可避免脏读、不可重复读的发生
3. Read committed(读已提交)：可避免脏读的发生
4. Read uncommitted(读未提交)：最低级别，任何情况都无法保证

级别越高，执行效率越低。像Serializable这样的级别，就是以锁表的方式使得其他线程只能在锁外等待。所以应该根据实际情况来选择隔离级别。在MySQL数据库中支持以上4种隔离级别，默认的隔离级别是Repeatable read(可重复读)；而在Oracle数据库中，只支持Serializable(串行化)和Read committed(读已提交)这两种级别，其中默认的为Read committed(读已提交)

## Session与Cookie

# HTTP面试题

# Java IO和NIO面试题

## 在Java程序中，我有三个socket，我需要多少个线程来处理。

## Java中怎么创建ByteBuffer？

## Java中怎么读写ByteBuffer？

## Java采用的是大端序还是小端序？

## ByteBuffer中的字节序是什么？

## Java中直接缓冲区与非直接缓冲区有什么区别？

## Java中的内存映射缓冲区是什么？

## Socket选项TCP NO DELAY是指什么？

## TCP协议与UDP协议有什么区别？

## Java中ByteBuffer与StringBuffer有什么区别？

# 编程和代码相关的面试题

## 93）怎么检查一个字符串只包含数字？

## 94）Java 中如何利用泛型写一个 LRU 缓存？

## 95）写一段 Java 程序将 byte 转换为 long？

## 95）在不使用 StringBuffer 的前提下，怎么反转一个字符串？

## 97）Java 中，怎么获取一个文件中单词出现的最高频率？

## 98）如何检查出两个给定的字符串是反序的？

## 99）Java 中，怎么打印出一个字符串的所有排列？

## 100）Java 中，怎样才能打印出数组中的重复元素？

## 101）Java 中如何将字符串转换为整数？

## 102）在没有使用临时变量的情况如何交换两个整数变量的值？

# 设计模式面试题

103）接口是什么？为什么要使用接口而不是直接使用具体类？  
接口用于定义 API。它定义了类必须得遵循的规则。同时，它提供了一种抽象，因为客户端只使用接口，这样可以有多重实现，如 List 接口，你可以使用可随机访问的 ArrayList，也可以使用方便插入和删除的 LinkedList。接口中不允许写代码，以此来保证抽象，但是 Java 8 中你可以在接口声明静态的默认方法，这种方法是具体的。

104）Java 中，抽象类与接口之间有什么不同？([答案](http://javarevisited.blogspot.sg/2013/05/difference-between-abstract-class-vs-interface-java-when-prefer-over-design-oops.html))  
Java 中，抽象类和接口有很多不同之处，但是最重要的一个是 Java 中限制一个类只能继承一个类，但是可以实现多个接口。抽象类可以很好的定义一个家族类的默认行为，而接口能更好的定义类型，有助于后面实现多态机制。关于这个问题的讨论请查看答案。

105）除了单例模式，你在生产环境中还用过什么设计模式？  
这需要根据你的经验来回答。一般情况下，你可以说依赖注入，工厂模式，装饰模式或者观察者模式，随意选择你使用过的一种即可。不过你要准备回答接下的基于你选择的模式的问题。

106）你能解释一下里氏替换原则吗?([答案](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html))

107) 什么情况下会违反迪米特法则？为什么会有这个问题？([答案](http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html))  
迪米特法则建议“只和朋友说话，不要陌生人说话”，以此来减少类之间的耦合。

108）适配器模式是什么？什么时候使用？  
适配器模式提供对接口的转换。如果你的客户端使用某些接口，但是你有另外一些接口，你就可以写一个适配去来连接这些接口。

109）什么是“依赖注入”和“控制反转”？为什么有人使用？([答案](http://javarevisited.blogspot.sg/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html))

110）抽象类是什么？它与接口有什么区别？你为什么要使用过抽象类？([答案](http://java67.blogspot.sg/2014/06/why-abstract-class-is-important-in-java.html))

111）构造器注入和 setter 依赖注入，那种方式更好？([答案](http://javarevisited.blogspot.sg/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html))  
每种方式都有它的缺点和优点。构造器注入保证所有的注入都被初始化，但是 setter 注入提供更好的灵活性来设置可选依赖。如果使用 XML 来描述依赖，Setter 注入的可读写会更强。经验法则是强制依赖使用构造器注入，可选依赖使用 setter 注入。

112）依赖注入和工程模式之间有什么不同？([答案](http://javarevisited.blogspot.sg/2015/06/difference-between-dependency-injection.html))  
虽然两种模式都是将对象的创建从应用的逻辑中分离，但是依赖注入比工程模式更清晰。通过依赖注入，你的类就是 POJO，它只知道依赖而不关心它们怎么获取。使用工厂模式，你的类需要通过工厂来获取依赖。因此，使用 DI 会比使用工厂模式更容易测试。关于这个话题的更详细讨论请参见答案。

113）适配器模式和装饰器模式有什么区别？([答案](http://javarevisited.blogspot.sg/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html))  
虽然适配器模式和装饰器模式的结构类似，但是每种模式的出现意图不同。适配器模式被用于桥接两个接口，而装饰模式的目的是在不修改类的情况下给类增加新的功能。

114）适配器模式和代理模式之前有什么不同？([答案](http://javarevisited.blogspot.sg/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html))  
这个问题与前面的类似，适配器模式和代理模式的区别在于他们的意图不同。由于适配器模式和代理模式都是封装真正执行动作的类，因此结构是一致的，但是适配器模式用于接口之间的转换，而代理模式则是增加一个额外的中间层，以便支持分配、控制或智能访问。

115）什么是模板方法模式？(答案)  
模板方法提供算法的框架，你可以自己去配置或定义步骤。例如，你可以将排序算法看做是一个模板。它定义了排序的步骤，但是具体的比较，可以使用 Comparable 或者其语言中类似东西，具体策略由你去配置。列出算法概要的方法就是众所周知的模板方法。

116）什么时候使用访问者模式？(答案)  
访问者模式用于解决在类的继承层次上增加操作，但是不直接与之关联。这种模式采用双派发的形式来增加中间层。

117）什么时候使用组合模式？(答案)  
组合模式使用树结构来展示部分与整体继承关系。它允许客户端采用统一的形式来对待单个对象和对象容器。当你想要展示对象这种部分与整体的继承关系时采用组合模式。

118）继承和组合之间有什么不同？([答案](http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html))  
虽然两种都可以实现代码复用，但是组合比继承共灵活，因为组合允许你在运行时选择不同的实现。用组合实现的代码也比继承测试起来更加简单。

119）描述 Java 中的重载和重写？([答案](http://java67.blogspot.sg/2012/09/difference-between-overloading-vs-overriding-in-java.html))  
重载和重写都允许你用相同的名称来实现不同的功能，但是重载是编译时活动，而重写是运行时活动。你可以在同一个类中重载方法，但是只能在子类中重写方法。重写必须要有继承。

120）Java 中，嵌套公共静态类与顶级类有什么不同？([答案](http://javarevisited.blogspot.sg/2012/12/inner-class-and-nested-static-class-in-java-difference.html))  
类的内部可以有多个嵌套公共静态类，但是一个 Java 源文件只能有一个顶级公共类，并且顶级公共类的名称与源文件名称必须一致。

121) OOP 中的 组合、聚合和关联有什么区别？([答案](http://javarevisited.blogspot.sg/2014/02/ifference-between-association-vs-composition-vs-aggregation.html))  
如果两个对象彼此有关系，就说他们是彼此相关联的。组合和聚合是面向对象中的两种形式的关联。组合是一种比聚合更强力的关联。组合中，一个对象是另一个的拥有者，而聚合则是指一个对象使用另一个对象。如果对象 A 是由对象 B 组合的，则 A 不存在的话，B一定不存在，但是如果 A 对象聚合了一个对象 B，则即使 A 不存在了，B 也可以单独存在。

122）给我一个符合开闭原则的设计模式的例子？([答案](http://javarevisited.blogspot.sg/2011/11/great-example-of-open-closed-design.html))  
开闭原则要求你的代码对扩展开放，对修改关闭。这个意思就是说，如果你想增加一个新的功能，你可以很容易的在不改变已测试过的代码的前提下增加新的代码。有好几个设计模式是基于开闭原则的，如策略模式，如果你需要一个新的策略，只需要实现接口，增加配置，不需要改变核心逻辑。一个正在工作的例子是 Collections.sort() 方法，这就是基于策略模式，遵循开闭原则的，你不需为新的对象修改 sort() 方法，你需要做的仅仅是实现你自己的 Comparator 接口。

123）抽象工厂模式和原型模式之间的区别？(答案)

124）什么时候使用享元模式？(答案)  
享元模式通过共享对象来避免创建太多的对象。为了使用享元模式，你需要确保你的对象是不可变的，这样你才能安全的共享。JDK 中 String 池、Integer 池以及 Long 池都是很好的使用了享元模式的例子

## 129）Java 中，Serializable 与 Externalizable 的区别？

10 大 Spring 框架面试题及答案([参见](http://javarevisited.blogspot.sg/2011/09/spring-interview-questions-answers-j2ee.html))   
20 个非常好的设计模式面试问题([参见](http://java67.blogspot.com/2012/09/top-10-java-design-pattern-interview-question-answer.html))  
10个最流行的 Struts 面试题（Java 开发者）([参见](http://javarevisited.blogspot.sg/2011/11/struts-interview-questions-answer-j2ee.html))   
10 个最频繁被问到的 Servlet 面试问题及答案([参见](http://javarevisited.blogspot.sg/2011/09/servlet-interview-questions-answers.html))   
10 个非常好的 Oracle 面试问题（Java 开发者）([参见](http://javarevisited.blogspot.sg/2012/12/top-10-oracle-interview-questions-and-answers-database-sql.html))  
10 大 来自 J2EE 面试中的 JSP 问题([更多](http://javarevisited.blogspot.sg/2011/10/jsp-interview-questions-answers-for.html))   
10 个非常好 Hibernate 面试问题（Java EE 开发者）([参见](http://javarevisited.blogspot.sg/2013/05/10-hibernate-interview-questions-answers-java-j2ee-senior.html))  
10 个非常好的 JDBC 面试题（Java 开发者）([参见](http://javarevisited.blogspot.sg/2012/12/top-10-jdbc-interview-questions-answers.html))  
15 个 Java NIO 和网络面试题及答案

# 设计模式六大原则

#### 单一职责原则

一个类只负责一项原则

#### 里氏替换原则

所有引用基类的地方必须能透明的使用其子类的对象。

#### 依赖倒转原则

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。

本质思想就是要面向接口编程。

#### 接口隔离原则

客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

#### 迪米特法则

一个对象应该对其他对象保持最少的了解。如果两个类不必彼此通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三方转发这个调用。比如需要修电脑，不必把电脑亲自送到修理厂而死给售后进行处理。

#### 开闭原则

# 设计模式

## 设计模式的分类

**创建型模式5**：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式

**结构型模式7**：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式

**行为型模式11**：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

还有两种：并发型模式和线程池模式

#### 观察者模式

定义了对象之间的一对多的依赖，这样一来，当一个对象改变时，它的所有的依赖者都会收到通知并自动更新。

#### 工厂方法模式

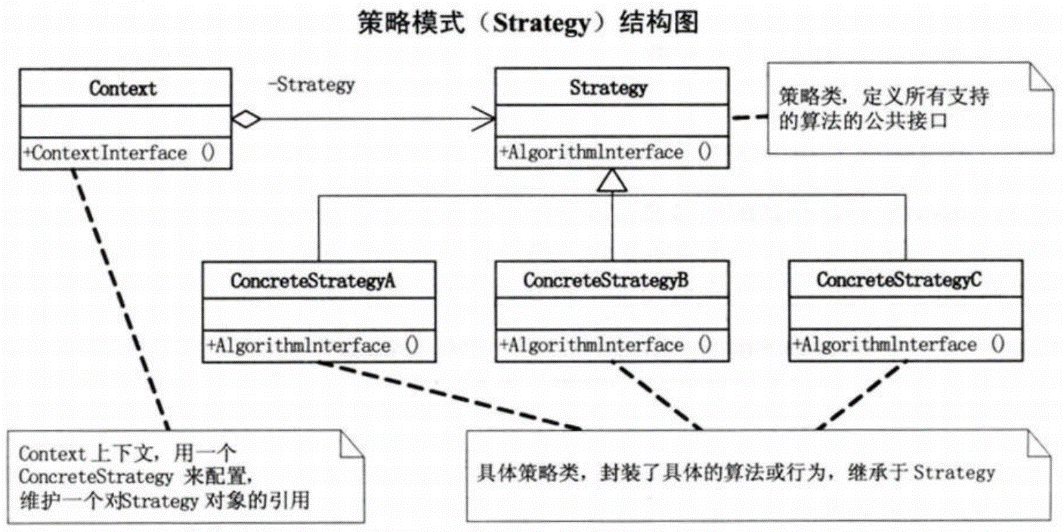
用具体的工厂来创建具体的产品而不是只用一个工厂创建所有产品。

#### 单例模式

最完美的单例模式就是用枚举类写。

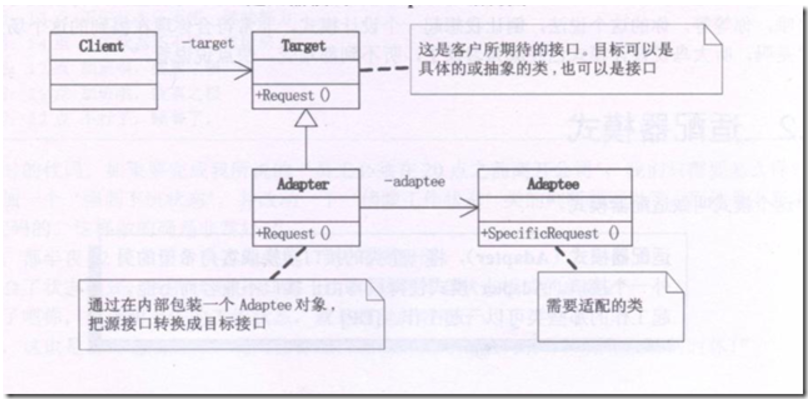
#### 策略模式

定义一个抽象的算法类，其中有一个算法方法。定义具体的算法类来继承抽象的算法类，具体的算法类里面重写算法方法，实现具体的算法。



#### 适配器模式

将一个类的接口转换为客户希望的另一个接口，使得原本接口不兼容而不能一起工作的那些类可以一起工作。



#### 命令模式

本来用户想直接调用这些命令的，如菜单上打开文档，就直接指向打开文档的代码，使用Command命令，就是在这两者之间增加一个中间者，将这种直接关系拗断，同时两者之间都隔离，基本没有关系了。

#### 装饰者模式

Java中的I/O API就是使用装饰者模式实现的，如BufferWriter装饰Writer等。

在不必改变原类文件和使用继承的情况下，动态的扩展一个对象的功能，它是通过一个包装对象，也就是装饰来包裹真实对象。

#### 外观模式

为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。A想要去a地旅游需要买票乘车，去b第旅游需要买票乘车，去c地旅游需要买票乘车。但是现在我们可以把这一系列的手续交给旅行社，让旅行社来实现。这时旅行社可以帮助我们买票、预定车和预定房间等，我们只需要交付相应的钱，这时旅行社要封装a,b,c的属性和方法。

#### 模板方法模式

准备一个抽象类，将部分逻辑以具体方法以及具体的构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对不同的逻辑有不同的实现。代码如下：

AbstractPerson类：

public abstract class AbstractPerson {  
 public abstract void dressUp();  
 public abstract void eatBreakFirst();  
 public abstract void takeThings();  
  
 public void prepareGoToSchool(){  
 dressUp();  
 eatBreakFirst();  
 takeThings();  
 }  
}

Student类：

public class Student extends AbstractPerson {  
 @Override  
 public void dressUp() {  
 System.*out*.println("穿衣服");  
 }  
 @Override  
 public void eatBreakFirst() {  
 System.*out*.println("吃饭");  
 }  
 @Override  
 public void takeThings() {  
 System.*out*.println("拿东西");  
 }  
}

Client类：

public class Client {  
 public static void main(String[] args) {  
 Student student = new Student();  
 student.prepareGoToSchool();  
 }  
}

#### 状态模式

假如，一个类需要根据不同的状态来做不同的行为，通常情况下，会对状态进行大量的判断，根据状态的不同来执行不同的操作，但是这样会使代码显得很长很繁琐而且不容易维护。

状态模式的思路就是将每一个状态写成一个具体的类，然后在每一个状态类里面判断状态，如果满足这个类的状态采取什么操作，如果不满足就转到其他状态类里面进行处理。