JAVA基础

如何减少gc出现的次数（java内存管理）

（1）对不用时最好显示设置为NULL。（方便提高GC收集器判定垃圾，提高GC效率）

（2）尽量少使用System.gc()（很多情况下会触发主GC，增加了间歇性停顿的次数）

（3）尽量少使用静态变量（静态变量属于全局变量，不会被GC回收，他们会一直占用内存）

（4）尽量使用StringBuffer，而不是用String来累加字符串（String是不可变长度的字符串，会在拼接过程中产生垃圾）

（5）分散对象创建或删除的时间

（6）尽量少使用finalize函数，因为他会加大GC的工作量，尽量少用finalize方式回收资源。

（7）如果需要使用经常用到的图片，可以使用软引用类型，他可以尽可能将图片保存在内存中，而不引起OutOfMemory。

（8）能使用基本类型,就不要使用包装类型。

（9）增大-Xmx的值

数组多大放在jvm老年代?

虚拟机提供了一个-XX:PretenureSizeThreshold参数（通常是3MB）,令大于这个设置值的对象直接在老年代分配，这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制。

jvm常见的启动参数

-Xms:设置堆的最小值

-Xmx:设置堆的最大值

-Xmn:设置新生代的大小

-Xss:设置每个线程的栈大小

-XX:NewSize:设置新生代的初始值

-XX:MaxNewSize:设置新生代的最大值

-XX:PermSize:设置永久代的初始值

-XX:MaxPermSize:设置永久代的最大值

-XX:SurviorRatio:年轻代中Eden区与Survivor大小比值。

-XX:PretenureSizeThreshold:令大于这个设置值的对象直接在老年代分配

虚拟机类加载机制

类加载的时机

类从被加载到虚拟机内存中开始，到卸载出内存为止，整个生命周期包括：加载、验证、准备、解析、初始化、使用、和卸载七个阶段。其中验证、准备、解析三个部分统称为连接（Linking）

五种必须初始换的情况：

1.遇到new、getstatic、putstatic或invokestatic者4条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化、生成这四条指令最常见的java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译期吧结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

2.使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

3.当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

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

5.当时用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF_getStatic,REF_putStatic、REF_invokeStatic的方法句柄，并且这个方法句柄对应的类没有进行初始化，则需要先触发其初始化。

上述行为被称为主动引用，只有主动引用才会触发初始化。被动引用不会触发初始化。

被动引用：

1.通过子类引用父类的静态字段不会导致子类初始化

2.通过数组定义来引用类，不会触发此类的初始化。（此时会初始化一个由虚拟机生成的、直接继承于java.lang.Object的子类，创建动作由字节码指令newarray触发）

3.常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的初始化。

类加载的过程

加载：获取Class文件的字节流

在加载阶段，虚拟机需要完成以下三件事情：

1. 通过一个类的全限定名来获取定义此类的二进制字节流（没有规定从哪个地方获取二进制数据流 ）
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
3. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

验证：为了确保Class文件的字节流中包含信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

主要四个检验动作：

文件格式验证（验证是否符合Class文件格式的标准）

元数据验证（保证其描述信息符合java语言规范的要求）

字节码验证（通过数据流和控制流的分析，确定语义是合法的、符合逻辑的）

符号引用验证（符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验）

准备：正式为类标量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都讲在方法区中进行分配。类变量（被static修饰的变量）

解析：虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用：以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义的定位到目标即可。

直接引用：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。

初始化：根据程序员通过程序制定的主观计划去初始化变量和其他资源，或者从另一个角度讲初始化阶段是执行类构造器()方法的过程。

双亲委派模型

从java虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器，这个类加载器使用C++语言实现，是虚拟机自身的一部分；另一种就是其他类加载器，这些类加载器都由java语言实现，独立于虚拟机外部，并且都继承自抽象类java.lang.ClassLoader.

从java开发人员的角度来看，类加载器还可以划分的更细致一些，绝大部分java程序都会使用到一下3种系统提供的类加载器：

启动类加载器：加载<JAVA_HOME>/lib目录中的，或者被-Xbootclasspath参数所指定的路径中的。

扩展类加载器：这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载<JAVA_HOME>/lib/ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器:这个类加载器由sun.misc.Launcher$AppClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般也可以趁他为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

工作过程：如果一个类加载器收到了类加载的请求，他首先不会自己去尝试加载这个类，而不是这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己午饭完成这个加载请求时，自加载器才会尝试自己去加载。

双亲委派模型中有哪些方法：

findLoadedClass(),LoadClass(),findBootStrapClassOrNull(),findClass(),defineClass(),resolveClass()

```
protected synchronized Class<?> loadClass(String name,boolean resolve)
throws ClassNotFoundException
{
  //首先，检查请求的类是否已经被加载过了
  Class c = findLoadedClass(name);
  if(c == null){
    try{
      if(parent != null){
        c = parent.loadClass(name,false);
      }else{
        c = findBootstrapClassOrNull(name);
      }
      }catch(ClassNotFoundException e){
        //如果父类加载器抛出ClassNotFoundException
        //说明父类无法完成加载请求
      }
```

```
            }
        if(c == null){
            //在父类加载器无法加载的时候
            //再调用本身的findClass方法来进行类加载
            c = findClass(name);
        }
    }
    if(resolve){
        resolveClass(c);
    }
    return c;
}
```

用户如何自定义类加载器

继承ClassLoader类，重写findClass()方法.

怎么打破双亲委派模型?

1. 继承ClassLoader然后重写lodaClass()方法。

描述java类加载器的工作原理及其组织结构

java类加载器的作用就是在运行时加载类。

java类加载器基于三个机制：委托性、可见性和单一性。

1. 委托机制是指双亲委派模型。优点：能够提高软件系统的安全性。因为在此机制下，用户自定义的类加载器不可能加载本应该由父类加载器加载的可靠类，从而防止不可靠的恶意代码代替父类加载器加载可靠代码。
2. 可见性原理是子类的加载器可以看见所有的父类加载器加载的类，而父类加载器看不到子类加载器加载的类。
3. 单一性原理是指仅加载一个类一次，这是由委托机制确保子类加载器不会再次加载父类加载器加载过的类。

静态分派和动态分派

静态分派：静态类型（重载）

动态分派：动态类型（重写）

javac编译器的编译过程:

1. 解析与填充符号表过程

解析步骤包含了词法分析和语法分析两个过程。

1. 插入式注解处理器的注解处理过程

插入式注解处理器可以看做一组编译器插件，在这些插件里面，可以读取、修改、添加抽象语法树中的任意元素。

1. 语义分析与字节码生成过程

语法分析之后，编译器获得了程序代码的抽象语法树表示，语法树能够表示结构正确的源程序的抽象，但是无法保证原程序是否符合逻辑，而语义分析主要是对结构上正确的源程序进行上下文有关性质的检查。

编译阶段对程序做了哪些优化?

编译期其实是个"不确定"的操作过程，他极可能指的是前端编译器的编译过程，也可能值得是后端运行期编译器的编译过程。

new的对象如何不分配在堆而分配在栈上?

方法逃逸

在一般应用中，不会逃逸的局部对象占比很大，如果使用栈上分配，那大量对象就会随着方法的结束而自动销毁，垃圾回收系统压力就会小很多。

JAVA中的集合类

ArrayLiat、LikedList、Vector（线程安全）的区别和实现原理

ArrayList和Vctor只能按照顺序存储元素（从下标为0的位置开始），删除元素的时候，需要移位被置空，默认初始容量都是10

ArrayList和Vector基于数组实现的，LinkedList基于双向循环链表实现的（含有头结点）

HashMap、HashTable、LinkedHashMap、ConcurrentHashMap、WeakHashMap的区别和实现原理

一、HashMap VS HashTable

HashMap中为什么数组的长度是2的n次方呢?

这个方法是非常巧妙，他通过h&(table.length-1)来得到对象的保存位，而HashMap底层数组的长度是2的n次方，2n-1得到二进制的每个位上的值为1，那么与全部为1的一个数进行与操作，速度会大大提升。

HashMap和HashTable的区别

1. HashTable是线程安全的，HashMap不是线程安全的。

2. HashMap的key和value都可以是null，HashTable的key和value都不允许有NULL。
3. HashMap的数组默认大小是16，而且一定是2的n次幂，扩容后的数组长度是之前数组长度的两倍。HashTable中数组的默认大小是11，扩容后数组长度是之前数组长度的2n+1
4. 哈希值的使用不同
5. 判断是否含含有某个键

ConcurrentHashMap的原理

分段锁

ConcurrentHashMap的高并发性主要来自于三个方面：

1. 用分离锁实现多个线程间的更深层次的共享访问
2. 用HashEntry对象的不变性来降低执行读操作的线程在遍历链表间对加锁的需求
3. 通过对同一个Volatile变量的写/读访问，协调不同线程间读写操作的内存可见性

   通过HashEntry对象的不变形对同一个Volatile变量的读/写来协调内存可见性，是的读操作大多数时候不需要加锁就能成功获取到需要的值。

ConcurrentHashMap存在的问题？

弱一致性

WeakHashMap VS HashMap

WeakHash中的key采用的是“弱引用”的方式，只要WeakHashMap中的key不再被外部引用，所对应的键值对就可以被垃圾回收器回收。

HashMap中的key采用的是“强引用”的方式，当key不再被外部引用时，只有当这个key从hashMap中删除后，才可以被垃圾回收器回收。

HashMap和TreeMap区别

1.实现方式

HashMap:基于哈希表实现。TreeMap：基于红黑树实现

TreeMap:能够把它保存的记录根据键排序

HashMap:适用于在Map中插入、删除、查找数据

TreeMap：适用于按自然顺序或自定义顺序遍历键

HashMap通常比TreeMap快一点

对于HashSet而言，他是基于HashMap实现的，Hashset底层使用HashMap来保存所有元素，因此HashSet的实现比较简单，相关HashSet的操作，基本上都是直接调用HashMap的相关方法来完成的。HashSet中的元素都存放在HashMap的key上面，而value中的值都是统一的一个private static final Object PRESENT = new Object();

讲一下集合中的fail-fast机制

例如：假设存在两个线程（线程1，线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出ConcurrentModificationException异常，从而产生fail-fast机制。

产生的原因：

当调用iterator（）方法返回Iterater对象时，把容器中包含对象的个数赋值给一个变量expectedModCount，在调用next（）方法时，会比较expectedModCount与容器中实际对象的个数是否相等，若两者不相等，则会抛出一个concurrentModeficationException异常

如果在遍历集合的同时，需要删除元素的话，可以用iterator里面的remove()方法。

Collection和Collections的区别

Collection是集合类的上级接口，子接口主要有Set和List

Collections是针对集合类的一个帮助类，提供了操作集合的工具方法：一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

设计模式相关

享元模式?

String采用了享元模式

原型模式

原型模式主要用于对象复制，实现了一个接口（实现Cloneable接口），重写一个方法（重写Object类中的clone方法），即完成了原型模式

浅拷贝：对值类型的成员进行值得复制，对引用类型的成员变量只复制不引用，不复制引用的对象

深拷贝：对值类型的成员变量进行值的赋值，对引用类型的成员变量也进行引用对象的复制。

原型模式的优点：

1.如果创建新的对象比较复杂时，可以利用原型模式简化对象的创建过程。

2.使用原型模式创建对象，比直接new一个对象在性能上要好的多，因为Object类的Clone方法是一个本地方法，他直接操作内存中的二进制流，特别是复制大对象时，性能的差别比较明显。

JAVA语言相关

---

instanceof关键字的作用

用法：对象A instanceof 类B

instanceof通过返回一个布尔值来之处，这个对象是否是这个特定类或者是他的子类的一个实例。如果A为NULL则返回false

strictfp的作用

strictfp可以用来修饰一个类、接口或者方法，在所声明的范围内，所有浮点数的计算都是精确的。当一个类被strictfp修饰时，所有方法默认也被strictfp修饰。

什么是不可变类?

不可变类：当创建了一个类的实例之后，就不允许修改他的值了，特别注意：String和包装类（Integer,Long）都是不可变类。String采用了享元模式

String s = "abc"

String = "ab"+"c"

System.out.println(s ==ss)

输出为true

解析："ab"+"c"在编译时期就已经转换为abc

在java中基本数据类型占据几个字节?

在java中：

占一个字节：byte,boolean

占两个字节：char,short

占四个字节：int,float

占八个字节:long,double

他们对应的封装类型是：Integer，
Double,Long,Float,Short,Bute,Charater,Boolean

Integer的缓存策略

在类加载时就将-128到127的Integer对象创建了，并保存到cache数组中（Integer cache[]），一旦程序调用Integer.valueOf(i)方法，如果i的值是-128到127之间，就直接在cache缓存数组中去取Integer对象，不再的话，就创建新的包装类对象。

强制转换类型的规则

当对小于int'的数据类型（byte，char，short）进行运算时，首先会把这些类型的变量值强制转为int类型，对int类型进行运算，最后得到的值也是int类型的。因此，如果把两个short类型的值相加，最后得到的结果是int类型，如果需要得到short类型的结果，就必须显式的将运算结果转为short类型。

数组初始化时需要注意的问题

数组被创建之后会根据数组存放的数据类型默认初始化对应的初始值，例如，int类型会初始化为0，对象类型会初始化为null

二维数组中，每行元素个数可以不同。

如何在main方法执行前，输出“Hello world”

用静态代码块。静态代码块在类加载的初始化阶段就会被调用。

Java程序初始化的顺序（对象实例化的过程）

1.父类的静态变量、父类的静态代码块（谁在前，谁先初始化）

2.子类的静态变量、子类的静态代码块（谁在前，谁先初始化）

3.父类的非静态变量、父类的非静态代码块（谁在前，谁先初始化）、父类构造函数

4.子类的非静态变量、子类的非静态代码块（谁在前，谁先初始化），子类构造函数

构造函数的特点

1. 构造函数必须和类名一样，（但和类名一样的不一定是构造方法，普通方法也可以和类名同名），并且不能有返回值，返回值也不能为void。
2. 构造函数总是伴随着new一起调用，并且不能由程序的代码调用，只能由系统调用
3. 构造函数不能被继承
4. 子类可以通过super（）来显示调用父类的构造函数

序列化和反序列化

定义：

把java对象转换为字节序列的过程称为对象的序列化

把字节序列恢复为java对象的过程称为对象的反序列化

实现方式：

实现Serializable接口，一个标记接口。当序列化的时候需要用到ObjectOutputStream里面的writeObject();当反序列化的时候，需要用到ObjectInputStream里面的readObjec方法。

特点：

1. 序列化是，只对对象的状态进行保存，而不管对象的方法。
2. 当一个父类实现序列化时，子类自动实现序列化，不需要显式的实现Serializable接口
3. 当一个对象的实例变量引用了其他对象时，序列化该对象时，也把引用对象进行序列化
4. 对象中被static或者transient修饰的变量，在序列化时器变量值是不被保存的。

好处：

一：实现了数据的持久化，通过序列化可以把数据永久地保存到硬盘上。
二：利用序列化实现远程通信，即在网络上传送对象的字节序列。

ArrayList和LikedList能否序列化

都可以序列化。ArrayList里面的数组elementData是声明为transient的，表示ArrayList在序列化的时候，默认不会序列化这些数组元素。因为ArrayList实际上是动态数组，每次在放满以后会扩容，如果数组扩容之后，实际上只存放一个元素，那就会序列化很多null元素，浪费空间，所以ArrayList吧数组设置为transient，仅仅序列化已经保存的数据。

Switch能否用String做参数

在java7之前，switch只能支持byte,short,char,int或者对应的包装类以及Enum类型。在java7中，String支持被加上了。

Hashcode的作用

hashCode()方法是从Object类继承过来的，Object类中的hashCode()方法返回的是对象在内存中的地址转换成int值，如果对象没有重写hashCode()方法，任何对象的hashCode()方法返回的值是不想等。

Object有哪些公用的方法?

clone():创建并返回此对象的一个副本

equals(Object obj):只是其他某个对象是否与此对象的地址“相等”

hashCode()返回该对象的hash值

wait():

notify():唤醒在此对象监视器上等待的单个线程

notifyAll():唤醒在此对象监视器上等待的所有线程

toString():返回该对象的字符串表示

finalize():当垃圾回收器确定不存在该对象的更多引用时，由对象的垃圾回收器调用此方法。

getClass():返回此Object的运行时类。

String、StringBuffer与StringBuilder的区别

1.可变与不可变

String对象是不可变的；StringBuilder与StringBuffer对象是可变的。

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

修改String对象的原理：首先创建一个StringBUffer对象，然后调用append()方法，最后调用toString()方法。

2.是否线程安全

String和StringBuffer是线程安全的。StringBuilder并没有对方法进行同步加锁，所以是非线程安全的。

3.初始化方式不同

StringBuffer和StringBuilder只能用构造函数的形式来初始化。String除了用构造函数初始化外，还可以直接赋值。

说一下异常的原理

异常是指程序运行时所发生的错误。

Throwable是所有异常的父类，他有两个子类：Error和Exception

1.Error表示程序在运行期间发生了非常严重的错误，并且该错误是不可恢复的。Error不需要捕捉。如OutOfMemoryError。

2.Exception是可恢复异常。他包含了两种类型：检查异常和运行时异常、。

检查异常

比如IOException、SQLException和FileNotFoundException都是检查异常。他发生在编译阶段，编译器会强制程序去捕捉此类异常，需要在编码时用try-catch捕捉。

运行时异常

他发生在运行阶段，编译器不会检查运行时异常。比如空指针异常，算术运算异常，数组越界异常等。如果代码产生RuntimeException异常，则需要通过修改代码避免。例如，若发生除数为0的情况，则需要通过代码避免该情况发生。

JAVA面向对象的三个特征与含义

封装：属性的封装和方法的封装。把属性定义为私有的get（）,set()方法。

好处是信息隐藏和模块化，提高安全性。封装的主要作用在于对外隐藏内部实现细节，增强程序的安全性。

继承：子类可以继承父类的成员变量和成员方法，继承可以提高代码的复用性。

继承的特性：

1. 单一继承
2. 子类只能继承父类的非私有成员变量和方法。
3. 成员变量的隐藏和方法的覆盖。

多态：当同一个操作作用在不同对象时，会产生不同的结果。

java多态的实现原理

有两种方式实现多态，一种是编译时多态，另外一种是运行时多态；编译时多态是通过方法的重载来实现的，运行时多态是通过方法的重写来实现的。

方法重载：指的是同一个类中有多个同名的方法，但这些方法有着不同的参数。在编译时可以确定调用到那个方法。

方法重写，子类重写父类中的方法。父类的引用变量不仅可以指向父类的实例对象，还可以指向子类的对象时，只有在运行时才可以确定调用那个方法。

特别注意：只有类中的方法才有多态的概念，类成员变量没有多态的概念。

静态内部类和非静态内部类主要的不同：

（1）静态内部类不依赖与外部类实例而被实例化，而非静态内部类需要在外部类实例化后才可以被实例化。

（2）静态内部类不需要持有外部类引用。但非静态内部类需要持有对外部类的引用。

（3）静态内部类不能访问外部类的非静态成员和非静态方法。他只能访问外部类的静态成员和静态方法。非静态内部类能访问外部类的静态和非静态成员和方法。

static的使用方式

static有4中使用方式：修饰类（静态内部类），修饰成员变量（静态变量），修饰成员方法（静态成员方法），静态代码块

1.修饰类（静态内部类）

2.修饰成员变量（静态变量）

3.修饰成员方法（静态成员方法）

4.静态代码块

反射的作用与原理

定义：反射机制是在运行时，对于任意一个类，都能够知道这个类所有属性和方法；对于任意一个对象，都能够调用他的任意一个方法。在java中，只要给定类的名字，那么就可以通过反射机制来获得类的所有信息。

2.反射机制主要提供了一下功能：在运行时判定任意一个对象所属的类。在运行时创建对象；在运行时判定任意一个类所具有的成员变量和方法；在运行时可以调用任意一个对象的方法；生成动态代理。

反射的实现方式

有四种方式可以得到Class对象

1. Class.forname("类的路径")
2. 类名.class
3. 对象名.getClass()
4. 4.如果是基本包装类，则可以通过调用包装类的Type属性来获得该包装类的Class对象。

反射机制的优缺点：

优点：

1. 能够运行时动态的获取类的实例，大大提高程序的灵活性
2. 与java动态编译相结合，可以实现强大的功能

缺点：

1. 使用反射的性能较低 java反射是要解析字节码，将内存中的对象进行解析。

解决方法：

由于jdk的安全检查耗时较多，所以通过SetAccessible（true）的方式关闭安全检查来提升反射速度

需要多次动态创建一个类的实例的时候，有缓存的写法会比没有缓存要快很多。

继承和组合的区别

组合和继承是代码复用的两种方式

1. 组合是在新类里面创建原有类的对象，重复利用已有类的功能。
2. 组合关系在运行期决定，而继承关系在编译期就已经决定了
3. 使用继承关系是，可以实现类型的回溯，即用父类变量引用子类对象，这样便可以实现多态。二组合没有这个特性
4. 从逻辑上看，组合最重要的体现是一种整体和部分的思想，例如在电脑类是由内存类，CPU类，硬盘类等等组成的，而继承则体现的是一种可以回溯的父子关系，子类也是父类的一个对象。

创建虚引用的时候，构造方法传入一个ReferenceQueue，作用是什么?

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，如果发现他还有虚引用，就会把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入了引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

java中的NIO，BIO分别是什么。NIO主要用来解决什么问题?

NIO的主要作用就是来解决速度差异的。

java中NIO和IO之间的区别

IO面向流 NIO面向缓冲区

阻塞与非阻塞IO

Java的各种流是阻塞的。

NIO是非阻塞模式的。

选择器

java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经又可以处理输入，或者选择一准备写入的通道。为了将Channel和Selector配合使用，必须将channel注册到selector上，通过SelectableChannel.register()方法来实现。

只要Channel向Selector注册了某种特定的时间，Selector就会监听这些事件是否会发生，一旦发生某个时间，便会通知对象的Channel。使用选择器，借助单一线程，就可对数量庞大的活动I/O通道实施监控和维护。

NIO原理

在NIO中有几个核心对象：缓冲区（Buffer）、通道（Channel）、选择器（Selector）

缓冲区实际上是一个容器对象，其实就是一个数组。

任何时候访问NIO中的数据都是将他放到缓冲区中的。

通道是一个对象，通过他可以读取和写入数据，所有数据都通过Buffer对象来处理。我们永远不会将字节写入通道中，相反是将数据写入包含一个或者多个字节的缓冲区。读取也是从缓冲区中读取。

选择器

NIO有一个主要的类Selector，这个类似一个观察者，只要我们把需要探知的socketchannel告诉Selector,我们接着作别的事情，当有事件发生时，他会通知我们，传回一组SelectionKey，我们读取这些key，就会获取我们刚刚注册过的SocketChannel，然后我们从这个Channel中读取数据。

面向对象的6个基本原则(设计模式的6个基本原则)

单一职责原则

开放封闭原则

里氏替换原则

接口隔离原则

依赖倒置原则

单一职责：是指一个类的功能要单一，一个类只负责一个职责。一个类只做他该做的事情（高内聚）。在面向对象中，如果只让一个类完成它该做的是，而不涉及与他无关的领域就是践行了高内聚的原则。

开放封闭：软件实体应对扩展开放，对修改关闭。对扩展开放，意味着有新的需求或变化时，可以对现有的代码进行扩展，以适应新的情况。对修改封闭，意味着类一旦涉及完成，就可以独立其工作，而不要对类尽任何修改。

里氏替换：任何使用基类的地方，都能够使用子类替换，而且在替换子类后，系统能够正常工作。子类一定是增加父类能力，而不是减少父类能力。

接口隔离：即应该将接口的粒度最小化，将功能划分到每一个不能再分的角色，为每一个子角色创建接口，通过这样，才不会让接口的实现类实现一些不必要的功能。

依赖倒置：即我们的类要依赖于抽象，而不是依赖于具体的实体，也就是我们经常听到的“面向接口编程。”

JDK源码中用到的设计模式

单例模式：Runtime

享元模式：String常量池和Integer等包装类的缓存策略：Integer.ValueOf(int i)等。

原型模式：Object.clone;Cloneable

装饰器模式：IO流中

迭代器模式：Iterator

执行Student s = new Student();在内存中做了哪些事情?

1. 类加载过程
2. 对象初始化的顺序（对象实例化的过程）
3. 在栈内存为S开辟空间，把对象地址复制给S变量。

怎样让一个线程放弃锁

OBject.wait()

condition.await();

动态代理

动态代理

---

jdk1.3加入了动态代理相关的API，从上面静态代理的例子我们知道，静态代理，需要为被代理对象和方法实现撰写特定的代理对象，显然这样做并不灵活，我们希望可以有一个公用的代理，可以动态的实现对不同对象的代理，这就需要利用到反射机制和动态代理机制。 在动态代理中，一个handler可以代理服务各种对象，首先，每一个handler都必须继承实现java.lang.reflect.InvocationHandler接口，下面具体实例说明，依然是上面那个记录日志的例子

- LogHandler.java

```
package Reflection;

import java.util.logging.*;
import java.lang.reflect.*;

public class LogHandler implements InvocationHandler {
  private Logger logger =
      Logger.getLogger(this.getClass().getName());

  private Object delegate;

  public Object bind(Object delegate) {
    this.delegate = delegate;
    return Proxy.newProxyInstance(
                delegate.getClass().getClassLoader(),
                delegate.getClass().getInterfaces(),
                this);
  }

  public Object invoke(Object proxy, Method method,
                Object[] args) throws Throwable {
    Object result = null;

    try {
      log("method starts..." + method);

      result = method.invoke(delegate, args);


      logger.log(Level.INFO, "method ends..." + method);
```

```
            logger.log(Level.INFO, "method ends..." + method);
        } catch (Exception e){
            log(e.toString());
        }

        return result;
    }

    private void log(String message) {
        logger.log(Level.INFO, message);
    }
}
```

具体来说就是使用Proxy.newProxyInstance()静态方法new一个代理对象出来，底层会使用反射机制，建立代理对象的时候，需要传入被代理对象的class，以及被代理对象的所实现的接口，以及代理方法调用的调用程序InvocationHandler，即实现 InvocationHandler接口的对象。这个对象会返回一个指定类指定接口，指定 InvocationHandler的代理类实例，这个实例执行方法时，每次都会调用 InvocationHandler的invoke方法，invoke方法会传入被代理对象的方法与方法参数，实际方法的执行会交给method.invoke().所以我们就可以在其前后加上日志记录的工作。

接下来我们就来测试一下，使用logHandler的bind方法来绑定代理对象：

```
package Reflection;

import java.lang.reflect.Proxy;

public class ProxyDemo {

    public static void main(String[] args) {


        LogHandler logHandler  = new LogHandler();

        IHello helloProxy =
                (IHello) logHandler.bind(new HelloSpeaker());
        helloProxy.hello("baba");

    }

}
```