# Java基础面试题整理

## 面向对象的三个特征

封装,继承,多态.这个应该是人人皆知.有时候也会加上抽象.

## 多态的好处

允许不同类对象对同一消息做出响应,即同一消息可以根据发送对象的不同而采用多种不同的行为方式(发送消息就是函数调用).主要有以下优点:

可替换性:多态对已存在代码具有可替换性.

可扩充性:增加新的子类不影响已经存在的类结构.

接口性:多态是超类通过方法签名,向子类提供一个公共接口,由子类来完善或者重写它来实现的.

灵活性:

简化性:

## JAVA虚拟机是如何实现多态的

动态绑定技术(dynamic binding),执行期间判断所引用对象的实际类型,根据实际类型调用对应的方法.如果你知道Hotspot中oop-klass模型的实现,对这个问题就了解比较深了.

## 接口的意义

接口的意义用三个词就可以概括:规范,扩展,回调.

## 抽象类的意义

抽象类的意义可以用三句话来概括:

为其他子类提供一个公共的类型

封装子类中重复定义的内容

定义抽象方法,子类虽然有不同的实现,但是定义时一致的

### 接口和抽象类的区别

比较点 抽象类 接口

默认方法 抽象类可以有默认的方法实现 java 8之前,接口中不存在方法的实现

实现方式 子类使用extends关键字来继承抽象类.如果子类不是抽象类,子类需要提供抽象类中所声明方法的实现 子类使用implements来实现接口,需要提供接口中所有声明的实现.

构造器 抽象类中可以有构造器 接口中不能

和正常类区别 抽象类不能被实例化 接口则是完全不同的类型

访问修饰符 抽象方法可以有public,protected和default等修饰 接口默认是public,不能使用其他修饰符

多继承 一个子类只能存在一个父类 一个子类可以存在多个接口

添加新方法 抽象类中添加新方法,可以提供默认的实现,因此可以不修改子类现有的代码 如果往接口中添加新方法,则子类中需要实现该方法

## 父类的静态方法能否被子类重写?

不能.重写只适用于实例方法,不能用于静态方法,而子类当中含有和父类相同签名的静态方法,我们一般称之为隐藏.

## 什么是不可变对象?好处是什么?

不可变对象指对象一旦被创建,状态就不能再改变,任何修改都会创建一个新的对象,如 String、Integer及其它包装类.不可变对象最大的好处是线程安全.

## 静态变量和实例变量的区别?

静态变量存储在方法区,属于类所有.实例变量存储在堆当中,其引用存在当前线程栈.需要注意的是从JDK1.8开始用于实现方法区的PermSpace被MetaSpace取代了.

## 能否创建一个包含可变对象的不可变对象?

当然可以,比如final Person[] persons = new Persion[]{}.persons是不可变对象的引用,但其数组中的Person实例却是可变的.这种情况下需要特别谨慎,不要共享可变对象的引用.这种情况下,如果数据需要变化时,就返回原对象的一个拷贝.

## java 创建对象的几种方式

java中提供了以下四种创建对象的方式:

new创建新对象

通过反射机制

采用clone机制

通过序列化机制

前两者都需要显式地调用构造方法. 对于clone机制,需要注意浅拷贝和深拷贝的区别,对于序列化机制需要明确其实现原理,在java中序列化可以通过实现Externalizable或者Serializable来实现.

## switch中能否使用string做参数?

在JDK 1.7之前,switch只能支持byte,short,char,int或者其对应的包装类以及Enum类型.从JDK 1.7之后switch开始支持String类型.但到目前为止,switch都不支持long类型.

## Object中有哪些公共方法?

equals(),clone(),getClass(),notify(),notifyAll(),wait(),toString

## java中==和eqauls()的区别?

==是运算符,用于比较两个变量是否相等,对于基本类型而言比较的是变量的值,对于对象类型而言比较的是对象的地址.

equals()是Object类的方法,用于比较两个对象内容是否相等.默认Object类的equals()实现如下:

public class Object {

......

public boolean equals(Object obj) {

return (this == obj);

}

......

}

不难看出此时equals()是比较两个对象的地址,此时直接==比较的的结果一样.对于可能用于集合存储中的对象元素而言,通常需要重写其equals()方法.

## a==b与a.equals(b)有什么区别

如果a 和b 都是对象,则 a==b 是比较两个对象内存地址,只有当 a 和 b 指向的是堆中的同一个对象才会返回 true.而 a.equals(b) 是进行内容比较,其比较结果取决于equals()具体实现.多数情况下,我们需要重写该方法,如String 类重写 equals()用于两个不同对象，但是包含的字母相同的比较:

public boolean equals(Object anObject) {

if (this == anObject) { // 同一个对象直接返回true

return true;

}

if (anObject instanceof String) {

String anotherString = (String)anObject;

int n = value.length;

if (n == anotherString.value.length) {

char v1[] = value;

char v2[] = anotherString.value;

int i = 0;

while (n-- != 0) { // 按字符依次比较

if (v1[i] != v2[i])

return false;

i++;

}

return true;

}

}

return false;

}

## Object中的equals()和hashcode()的联系

hashCode()是Object类的一个方法,返回一个哈希值.如果两个对象根据equal()方法比较相等,那么调用这两个对象中任意一个对象的hashCode()方法必须产生相同的哈希值;如果两个对象根据eqaul()方法比较不相等,那么产生的哈希值不一定相等(碰撞的情况下还是会相等的.)

## a.hashCode()有什么用?与a.equals(b)有什么关系

hashCode()方法是为对象产生整型的 hash 值,用作对象的唯一标识.它常用于基于 hash 的集合类,如 Hashtable,HashMap等等.根据 Java 规范,使用 equal()方法来判断两个相等的对象,必须具有相同的 hashcode.

将对象放入到集合中时,首先判断要放入对象的hashcode是否已经在集合中存在,不存在则直接放入集合.如果hashcode相等,然后通过equal()方法判断要放入对象与集合中的任意对象是否相等:如果equal()判断不相等,直接将该元素放入集合中,否则不放入.

## 有没有可能两个不相等的对象有相同的hashcode

有可能.在产生hash冲突时,两个不相等的对象就会有相同的 hashcode 值.当hash冲突产生时,一般有以下几种方式来处理:

拉链法:每个哈希表节点都有一个next指针,多个哈希表节点可以用next指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向链表进行存储.

开放定址法:一旦发生了冲突,就去寻找下一个空的散列地址,只要散列表足够大,空的散列地址总能找到,并将记录存入

再哈希:又叫双哈希法,有多个不同的Hash函数.当发生冲突时,使用第二个,第三个….等哈希函数计算地址,直到无冲突.

可以在hashcode中使用随机数字吗?

不行,因为同一对象的 hashcode 值必须是相同的.

## & 和 &&的区别

基础的概念不能弄混:&是位操作,&&是逻辑运算符.需要记住逻辑运算符具有短路特性,而&不具备短路特性.来看看一下代码执行结果?

public class Test{

static String name;

public static void main(String[] args){

if(name!=null&userName.equals("")){

System.out.println("ok");

}else{

System.out.println("erro");

}

}

}

上述代码将会抛出空指针异常.原因你懂得.

在.java文件内部可以有多少类(非内部类)?

在一个java文件中只能有一个public公共类,但是可以有多个default修饰的类.

## 如何正确的退出多层嵌套循环?

使用标号和break;

通过在外层循环中添加标识符

## 内部类有什么作用?

内部类可以有多个实例,每个实例都有自己的状态信息,并且与其他外围对象的信息相互独立.在单个外围类当中,可以让多个内部类以不同的方式实现同一接口,或者继承同一个类.创建内部类对象的时刻不依赖于外部类对象的创建.内部类并没有令人疑惑的”is-a”关系,它就像是一个独立的实体.此外,内部类提供了更好的封装,除了该外围类,其他类都不能访问.

## final,finalize()和finally{}的不同之处

三者没有任何相关性,遇到有问着问题的面试官就拖出去砍了吧.final是一个修饰符,用于修饰变量,方法和类.如果 final 修饰变量,意味着该变量的值在初始化后不能被改变.finalize()方法是在对象被回收之前调用的方法,给对象自己最后一个复活的机会.但是该方法由Finalizer线程调用,但调用时机无法保证.finally是一个关键字,与 try和catch一起用于异常的处理,finally{}一定会被执行,在此处我们通常用于资源关闭操作.

## clone()是哪个类的方法?

java.lang.Cloneable 是一个标示性接口,不包含任何方法.clone ()方法在 Object 类中定义的一个Native方法:

protected native Object clone() throws CloneNotSupportedException;

## 深拷贝和浅拷贝的区别是什么?

浅拷贝:被复制对象的所有变量都含有与原来的对象相同的值,而所有的对其他对象的引用仍然指向原来的对象.换言之,浅拷贝仅仅复制所考虑的对象,而不复制它所引用的对象.

深拷贝:被复制对象的所有变量都含有与原来的对象相同的值.而那些引用其他对象的变量将指向被复制过的新对象.而不再是原有的那些被引用的对象.换言之.深拷贝把要复制的对象所引用的对象都复制了一遍.

static都有哪些用法?

所有的人都知道static关键字这两个基本的用法:静态变量和静态方法.也就是被static所修饰的变量/方法都属于类的静态资源,类实例所共享.

除了静态变量和静态方法之外,static也用于静态块,多用于初始化操作:

public calss PreCache{

static{

//执行相关操作

}

}

此外static也多用于修饰内部类,此时称之为静态内部类.

最后一种用法就是静态导包,即import static.import static是在JDK 1.5之后引入的新特性,可以用来指定导入某个类中的静态资源,并且不需要使用类名,可以直接使用资源名,比如:

import static java.lang.Math.\*;

public class Test{

public static void main(String[] args){

//System.out.println(Math.sin(20));传统做法

System.out.println(sin(20));

}

}

## final有哪些用法?

final也是很多面试喜欢问的地方,但我觉得这个问题很无聊,通常能回答下以下5点就不错了:

被final修饰的类不可以被继承

被final修饰的方法不可以被重写

被final修饰的变量不可以被改变.如果修饰引用,那么表示引用不可变,引用指向的内容可变.

被final修饰的方法,JVM会尝试将其内联,以提高运行效率

被final修饰的常量,在编译阶段会存入常量池中.

除此之外,编译器对final域要遵守的两个重排序规则更好:

在构造函数内对一个final域的写入,与随后把这个被构造对象的引用赋值给一个引用变量,这两个操作之间不能重排序

初次读一个包含final域的对象的引用,与随后初次读这个final域,这两个操作之间不能重排序.

数据类型相关

java中int char,long各占多少字节?

这个问题其实很无聊.应该去问java中的各种数据类型在不同的平台运行时期所占位数一样么?

类型 字节

short 2

int 4

long 8

float 4

double 8

char

## 64位的JVM当中,int的长度是占几个字节?

Java中数据类型所占用的位数和平台无关,在 32 位和64位 的Java 虚拟机中,int 类型的长度都是占4字节.

## int和Integer的区别?

Integer是int的包装类型,在拆箱和装箱中,二者自动转换.int是基本类型,直接存数值;而integer是对象;用一个引用指向这个对象.由于Integer是一个对象,在JVM中对象需要一定的数据结构进行描述,相比int而言,其占用的内存更大一些.

## String s = new String("abc")创建了几个String对象?

2个.一个是字符串字面常数,在字符串常量池中;另一个是new出来的字符串对象,在堆中.

请问s1==s3是true还是false，s1==s4是false还是true?s1==s5呢？

String s1 = "abc";

String s2 = "a";

String s3 = s2 + "bc";

String s4 = "a" + "bc";

String s5 = s3.intern();

s1==s3返回false,s1==s4返回true,s1==s5返回true.

“abc"这个字符串常量值会直接方法字符串常量池中,s1是对其的引用.由于s2是个变量,编译器在编译期间无法确定该变量后续会不会改,因此无法直接将s3的值在编译器计算出来,因此s3是堆中"abc"的引用.因此s1!=s3.对于s4而言,其赋值号右边是常量表达式,因此可以在编译阶段直接被优化为"abc”,由于"abc"已经在字符串常量池中存在,因此s4是对其的引用,此时也就意味s1和s4引用了常量池中的同一个"abc".所以s1==s4.String中的intern()会首先从字符串常量池中检索是否已经存在字面值为"abc"的对象,如果不存在则先将其添加到字符串常量池中,否则直接返回已存在字符串常量的引用.此处由于"abc"已经存在字符串常量池中了,因此s5和s1引用的是同一个字符串常量.

## 以下代码中,s5==s2返回值是什么?

String s1="ab";

String s2="a"+"b";

String s3="a";

String s4="b";

String s5=s3+s4;

返回false.在编译过程中,编译器会将s2直接优化为"ab",将其放置在常量池当中;而s5则是被创建在堆区,相当于s5=new String(“ab”);

## 你对String对象的intern()熟悉么?

Stirng中的intern()是个Native方法,它会首先从常量池中查找是否存在该常量值的字符串,若不存在则先在常量池中创建,否则直接返回常量池已经存在的字符串的引用. 比如

String s1="aa";

String s2=s1.intern();

System.out.print(s1==s2);

上述代码将返回true.因为在"aa"会在编译阶段确定下来,并放置字符串常量池中,因此最终s1和s2引用的是同一个字符串常量对象.

## String,StringBuffer和StringBuilder区别?

String是字符串常量,final修饰;StringBuffer字符串变量(线程安全);StringBuilder 字符串变量(线程不安全).此外StringBuilder和StringBuffer实现原理一样,都是基于数组扩容来实现的.

## String和StringBuffer的区别?

String和StringBuffer主要区别是性能:String是不可变对象,每次对String类型进行操作都等同于产生了一个新的String对象,然后指向新的String对象.所以尽量不要对String进行大量的拼接操作,否则会产生很多临时对象,导致GC开始工作,影响系统性能.

StringBuffer是对象本身操作,而不是产生新的对象,因此在有大量拼接的情况下,我们建议使用StringBuffer(线程安全).

需要注意现在JVM会对String拼接做一定的优化,比如

String s="This is only "+ "simple" +"test";

以上代码在编译阶段会直接被优化成会`String s=“This is only simple test”.

## StringBuffer和StringBuilder

StringBuffer和StringBuilder的实现原理一样,其父类都是AbstractStringBuilder.StringBuffer是线程安全的,StringBuilder是JDK 1.5新增的,其功能和StringBuffer类似,但是非线程安全.因此,在没有多线程问题的前提下,使用StringBuilder会取得更好的性能.

## 什么是编译器常量?使用它有什么风险?

公共静态不可变,即public static final修饰的变量就是我们所说的编译期常量.这里的public可选的.实际上这些变量在编译时会被替换掉,因为编译器明确的能推断出这些变量的值(如果你熟悉C++,那么这里就相当于宏替换).

编译器常量虽然能够提升性能,但是也存在一定问题:你使用了一个内部的或第三方库中的公有编译时常量,但是这个值后面被其他人改变了,但是你的客户端没有重新编译,这意味着你仍然在使用被修改之前的常量值.

3\*0.1==0.3返回值是什么

false,因为有些浮点数不能完全精确的表示出来.

## java当中使用什么类型表示价格比较好?

如果不是特别关心内存和性能的话,使用BigDecimal.否则使用预定义精度的 double 类型.

## 如何将byte转为String

可以使用String接收 byte[] 参数的构造器来进行转换,注意要使用的正确的编码,否则会使用平台默认编码.这个编码可能跟原来的编码相同.也可能不同.

## 可以将int强转为byte类型么?会产生什么问题?

可以做强制转换,但是Java中int是32位的而byte是8 位的.如果强制转化int类型的高24位将会被丢弃,byte 类型的范围是从-128到128.

## a=a+b与a+=b有什么区别吗?

+=操作符会进行隐式自动类型转换,此处a+=b隐式的将加操作的结果类型强制转换为持有结果的类型,而a=a+b则不会自动进行类型转换.如：

byte a = 127;

byte b = 127;

b = a + b; // 报编译错误:cannot convert from int to byte

b += a;

## 以下代码是否有错,有的话怎么改？

short s1= 1;

s1 = s1 + 1;

## 以下代码是否有错,有的话怎么改？

short s1= 1;

s1 = s1 + 1;

有错误.short类型在进行运算时会自动提升为int类型,也就是说s1+1的运算结果是int类型,而s1是short类型,此时编译器会报错.

## 以下代码是否有错,有的话怎么改？

short s1= 1;

s1 += 1;

+=操作符会对右边的表达式结果强转匹配左边的数据类型,所以没错.

## 了解泛型么?简述泛型的上界和下界?

有时候希望传入的类型有一个指定的范围，从而可以进行一些特定的操作,这时候就需要通配符了?在Java中常见的通配符主要有以下几种:

<?>: 无限制通配符

<? extends E>: extends 关键字声明了类型的上界,表示参数化的类型可能是所指定的类型,或者是此类型的子类

<? super E>: super关键字声明了类型的下界,表示参数化的类型可能是指定的类型,或者是此类型的父类

它们的目的都是为了使方法接口更为灵活,可以接受更为广泛的类型.

< ? extends E>: 用于灵活读取，使得方法可以读取 E 或 E 的任意子类型的容器对象。

< ? super E>: 用于灵活写入或比较,使得对象可以写入父类型的容器,使得父类型的比较方法可以应用于子类对象。

用简单的一句话来概括就是为了获得最大限度的灵活性,要在表示生产者或者消费者的输入参数上使用通配符,使用的规则就是:生产者有上限(读操作使用extends),消费者有下限(写操作使用super).

垃圾回收

## 简单的解释一下垃圾回收?

JVM中垃圾回收机制最基本的做法是分代回收.内存中的区域被划分成不同的世代,对象根据其存活的时间被保存在对应世代的区域中.一般的实现是划分成3个世代:年轻,年老和永久代.所有新生成的对象优先放在年轻代的(大对象可能被直接分配在老年代,作为一种分配担保机制),年轻代按照统计规律被分为三个区:一个Eden区，两个 Survivor区.在年轻代中经历了N次垃圾回收后仍然存活的对象,就会被放到年老代中.因此可以认为年老代中存放的都是一些生命周期较长的对象.

方法区也被称为永久代,用于存储每一个java类的结构信息:比如运行时常量池,字段和方法数据,构造函数和普通方法的字节码内容以及类,实例,接口初始化时需要使用到的特殊方法等数据,根据虚拟机实现不同,GC可以选择对方法区进行回收也可以不回收.

对于不同的世代可以使用不同的垃圾回收算法。比如对由于年轻代存放的对象多是朝生夕死,因此可以采用标记-复制,而对于老年代则可以采用标记-整理/清除.

Minor GC

发生在新生代的GC为Minor GC .在Minor GC时会将新生代中还存活着的对象复制进一个Survivor中,然后对Eden和另一个Survivor进行清理.所以,平常可用的新生代大小为Eden的大小+一个Survivor的大小.

Major GC

在老年代中的GC则为Major GC.

Full GC

通常是和Major GC等价的,针对整个新生代,老年代,元空间metaspace(java8以上版本取代perm gen)的全局范围的GC.

关于GC的类型,其实依赖于不同的垃圾回收器.可以具体查看相关垃圾回收器的实现.

新生代进入老年代

分配担保机制:当Minor GC时,新生代存活的对象大于Survivor的大小时,这时一个Survivor装不下它们,那么它们就会进入老年代.

如果设置了-XX：PretenureSizeThreshold5M 那么大于5M的对象就会直接就进入老年代.

在新生代的每一次Minor GC 都会给在新生代中的对象+1岁,默认到15岁时就会从新生代进入老年代,可以通过-XX：MaxTenuringThreshold来设置这个临界点

常见的垃圾回收算法有哪些?简述其原理.

垃圾回收从理论上非常容易理解,具体的方法有以下几种:

标记-清除

标记-复制

标记-整理

分代回收

更详细的内容参见深入理解垃圾回收算法

如何判断一个对象是否应该被回收?

这就是所谓的对象存活性判断,常用的方法有两种:

引用计数法

对象可达性分析

由于引用计数法存在互相引用导致无法进行GC的问题,所以目前JVM虚拟机多使用对象可达性分析算法.

哪些对象可以做GC Root?

主要由以下四种:

JVM方法栈中引用的对象

本地方法栈中引用的对象

方法区常量引用的对象

方法区类属性引用的对象

调用System.gc()会发生什么?

通知GC开始工作,但是GC真正开始的时间不确定.

## 了解java当中的四种引用类型?他们之间的区别是什么?

在java中主要有以下四种引用类型:强引用,软引用,弱引用,虚引用.不同的引用类型主要体现在GC上:

强引用:如果一个对象具有强引用,它就不会被垃圾回收器回收.即使当前内存空间不足,JVM也不会回收它.而是抛出 OutOfMemoryError 错误.使程序异常终止.如果想中断强引用和某个对象之间的关联.可以显式地将引用赋值为null,这样一来的话.JVM在合适的时间就会回收该对象.

软引用:在使用软引用时,如果内存的空间足够,软引用就能继续被使用而不会被垃圾回收器回收.只有在内存不足时,软引用才会被垃圾回收器回收.

弱引用:具有弱引用的对象拥有的生命周期更短暂.因为当 JVM 进行垃圾回收,一旦发现弱引用对象,无论当前内存空间是否充足,都会将弱引用回收.不过由于垃圾回收器是一个优先级较低的线程,所以并不一定能迅速发现弱引用对象.

虚引用:如果一个对象仅持有虚引用,那么它相当于没有引用,在任何时候都可能被垃圾回收器回收.

更多了解参见深入对象引用

WeakReference与SoftReference的区别?

这点在四种引用类型中已经做了解释,这里在重复一下.虽然WeakReference与SoftReference都有利于提高 GC和内存的效率,但是 WeakReference ,一旦失去最后一个强引用,就会被 GC 回收,而软引用虽然不能阻止被回收,但是可以延迟到 JVM 内存不足的时候.

## 为什么要有不同的引用类型

不像C语言,我们可以控制内存的申请和释放,在Java中有时候我们需要适当的控制对象被回收的时机,因此就诞生了不同的引用类型,可以说不同的引用类型实则是对GC回收时机不可控的妥协.

进程,线程相关

说说进程,线程之间的区别?

简而言之,进程是程序运行和资源分配的基本单位,一个程序至少有一个进程,一个进程至少有一个线程.进程在执行过程中拥有独立的内存单元,而多个线程共享内存资源,减少切换次数,从而效率更高.线程是进程的一个实体,是cpu调度和分派的基本单位,是比程序更小的能独立运行的基本单位.同一进程中的多个线程之间可以并发执行.在Linux中,进程也称为Task.

## 守护线程是什么?它和非守护线程有什么区别

程序运行完毕,jvm会等待非守护线程完成后关闭,但是jvm不会等待守护线程.守护线程最典型的例子就是GC线程.

## 什么是多线程上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程.

## 创建两种线程的方式?他们有什么区别?

通过实现java.lang.Runnable或者通过扩展java.lang.Thread类.相比扩展Thread,实现Runnable接口可能更优.原因有二:

Java不支持多继承.因此扩展Thread类就代表这个子类不能扩展其他类.而实现Runnable接口的类还可能扩展另一个类.

类可能只要求可执行即可,因此继承整个Thread类的开销过大.

Callable和Runnable的区别是什么?

两者都能用来编写多线程,但实现Callable接口的任务线程能返回执行结果,而实现Runnable接口的任务线程不能返回结果.Callable通常需要和Future/FutureTask结合使用,用于获取异步计算结果.

## Thread类中的start()和run()方法有什么区别?

在start()方法中最终要的是调用了Native方法start0()用来启动新创建的线程线程启动后会自动调用run()方法.如果我们直接调用其run()方法就和我们调用其他方法一样,不会在新的线程中执行.

## ##怎么检测一个线程是否持有对象锁

Thread类提供了一个Native方法holdsLock(Object obj)方法用于检测是否持有某个对象锁:当且仅当对象obj的锁被某线程持有的时候才会返回true.

public static native boolean holdsLock(Object obj);

## 线程阻塞有哪些原因?

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。

方法 说明

sleep() sleep() 允许 指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。 典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止

suspend() 和 resume() 两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。

yield() yield() 使当前线程放弃当前已经分得的CPU 时间，但不使当前线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程

wait() 和 notify() 两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许 指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用.

wait(),notify()和suspend(),resume()之间的区别

初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的notify()方法则导致从调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和synchronized关键字一起使用，将它们和操作系统进程间通信机制作一个比较就会发现它们的相似性：synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点：

第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

## 产生死锁的条件

1.互斥条件：一个资源每次只能被一个进程使用。

2.请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3.不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

4.循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

## wait()与sleep()的区别

关于这两者已经在上面进行详细的说明,这里就做个概括好了:

sleep()来自Thread类，和wait()来自Object类.调用sleep()方法的过程中，线程不会释放对象锁。而 调用 wait 方法线程会释放对象锁

sleep()睡眠后不出让系统资源，wait让其他线程可以占用CPU

sleep(milliseconds)需要指定一个睡眠时间，时间一到会自动唤醒.而wait()需要配合notify()或者notifyAll()使用

为什么wait,nofity和nofityAll这些方法不放在Thread类当中

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

## ##怎么唤醒一个阻塞的线程

如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统。

## ##什么是多线程的上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

synchronized和ReentrantLock的区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的:ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word.

## FutureTask是什么

这个其实前面有提到过，FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

## 一个线程如果出现了运行时异常怎么办?

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放

## Java虚拟机对synchronized的优化

锁的状态总共有四种，无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁，但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级.

偏向锁: 偏向锁是JDK 1.6之后加入的新锁，它是一种针对加锁操作的优化手段，经过研究发现，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁

轻量级锁:倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁

自旋锁: 轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

除此之外,锁消除也是一项非常重要的优化手段.Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间.

## 线程中断相关方法

当一个线程处于被阻塞状态或者试图执行一个阻塞操作时，使用Thread.interrupt()方式中断该线程，此时将会抛出一个InterruptedException的异常，同时中断状态将会被复位(由中断状态改为非中断状态).在Java中提供了以下三个与中断相关的方法:

//中断线程（实例方法）

public void Thread.interrupt();

//判断线程是否被中断（实例方法）

public boolean Thread.isInterrupted();

//判断是否被中断并清除当前中断状态

public static boolean Thread.interrupted();

## 如何在两个线程间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

## 如何正确的使用wait()?使用if还是while?

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

synchronized (obj) {

while (condition does not hold)

obj.wait(); // (Releases lock, and reacquires on wakeup)

... // Perform action appropriate to condition

}

## 什么是线程局部变量ThreadLocal

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

ThreadLoal的作用是什么?

简单说ThreadLocal就是一种以空间换时间的做法在每个Thread里面维护了一个ThreadLocal.ThreadLocalMap把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了.

生产者消费者模型的作用是什么?

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

写一个生产者-消费者队列

可以通过阻塞队列实现,也可以通过wait-notify来实现.

使用阻塞队列来实现

//消费者

public class Producer implements Runnable{

private final BlockingQueue<Integer> queue;

public Producer(BlockingQueue q){

this.queue=q;

}

@Override

public void run() {

try {

while (true){

Thread.sleep(1000);//模拟耗时

queue.put(produce());

}

}catch (InterruptedException e){

}

}

private int produce() {

int n=new Random().nextInt(10000);

System.out.println("Thread:" + Thread.currentThread().getId() + " produce:" + n);

return n;

}

}

//消费者

public class Consumer implements Runnable {

private final BlockingQueue<Integer> queue;

public Consumer(BlockingQueue q){

this.queue=q;

}

@Override

public void run() {

while (true){

try {

Thread.sleep(2000);//模拟耗时

consume(queue.take());

}catch (InterruptedException e){

}

}

}

private void consume(Integer n) {

System.out.println("Thread:" + Thread.currentThread().getId() + " consume:" + n);

}

}

//测试

public class Main {

public static void main(String[] args) {

BlockingQueue<Integer> queue=new ArrayBlockingQueue<Integer>(100);

Producer p=new Producer(queue);

Consumer c1=new Consumer(queue);

Consumer c2=new Consumer(queue);

new Thread(p).start();

new Thread(c1).start();

new Thread(c2).start();

}

}

使用wait-notify来实现

该种方式应该最经典,这里就不做说明了

##如果你提交任务时,线程池队列已满,这时会发生什么

如果使用的LinkedBlockingQueue,也就是无界队列的话,继续添加任务到阻塞队列中等待执行.因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列,可以无限存放任务;如果你使用的是有界队列比方说ArrayBlockingQueue的话,任务首先会被添加到ArrayBlockingQueue中,ArrayBlockingQueue满了,则会使用拒绝策略RejectedExecutionHandler处理满了的任务,默认是AbortPolicy.