史上最全 Java 面试题，带全部答案！

相关概念

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

代码中如何实现多态

实现多态主要有以下三种方式：

1. 接口实现
2. 继承父类重写方法
3. 同一类中进行方法重载

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。

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	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	想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

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。

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。

能否创建一个包含可变对象的不可变对象?

当然可以创建一个包含可变对象的不可变对象的，你只需要谨慎一点，不要共享可变对象的引用就可以了，如果需要变化时，就返回原对象的一个拷贝。最常见的例子就是对象中包含一个日期对象的引用。

java 创建对象的几种方式

采用 new

通过反射

采用 clone

通过序列化机制

前 2 者都需要显式地调用构造方法。造成耦合性最高的恰好是第一种，因此你发现无论什么框架，只要涉及到解耦必先减少 new 的使用。

switch 中能否使用 string 做参数

在 idk 1.7 之前， switch 只能支持 byte, short, char, int 或者其对应的封装类以及 Enum 类型。从 idk 1.7 之后 switch 开始支持 String。

switch 能否作用在 byte, long 上?

可以用在 byte 上，但是不能用在 long 上。

String s1=”ab”，String s2=”a”+”b”，String s3=”a”，String s4=”b”， s5=s3+s4 请问 s5==s2 返回什么？

返回 false。在编译过程中，编译器会将 s2 直接优化为”ab”，会将其放置在常量池当中，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?

intern()方法会首先从常量池中查找是否存在该常量值，如果常量池中不存在则现在常量池中创建，如果已经存在则直接返回。

比如

String s1=”aa”;

String s2=s1.intern();

System.out.print(s1==s2);//返回 true

Object 中有哪些公共方法?

equals()

clone()

getClass()

notify(),notifyAll(),wait()

toString

java 当中的四种引用

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：顾名思义，就是形同虚设，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。

WeakReference 与 SoftReference 的区别?

这点在四种引用类型中已经做了解释,这里简单说明一下即可:

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。有以下几个使用场景可以充分的说明：

> 利用软引用和弱引用解决 OOM 问题：用一个 HashMap 来保存图片的路径和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，从而有效地避免了 OOM 的问题.
>
> 通过软引用实现 Java 对象的高速缓存:比如我们创建了一 Person 的类，如果每次需要查询一个人的信息,哪怕是几秒中之前刚刚查询过的，都要重新构建一个实例，这将引起大量 Person 对象的消耗，并且由于这些对象的生命周期相对较短，会引起多次 GC 影响性能。此时，通过软引用和 HashMap 的结合可以构建高速缓存，提供性能。

java 中==和 eqauls()的区别,equals()和`hashcode 的区别

==是运算符，用于比较两个变量是否相等，而 equals 是 Object 类的方法，用于比较两个对象是否相等。默认 Object 类的 equals 方法是比较两个对象的地址，此时和==的结果一样。换句话说：基本类型比较用==，比较的是他们的值。默认下，对象用==比较时，比较的是内存地址，如果需要比较对象内容，需要重写 equal 方法。

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() 方法是相应对象整型的 hash 值。它常用于基于 hash 的集合类，如 Hashtable、HashMap、LinkedHashMap 等等。它与 equals() 方法关系特别紧密。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，两个不相等的对象可能会有相同的 hashcode 值，这就是为什么在 hashmap 中会有冲突。如果两个对象相等，必须有相同的 hashcode 值，反之不成立。

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

a==b 与 a.equals(b)有什么区别

如果 a 和 b 都是对象，则 a==b 是比较两个对象的引用，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) 是进行逻辑比较，所以通常需要重写该方法来提供逻辑一致性的比较。例如，String 类重写 equals() 方法，所以可以用于两个不同对象，但是包含的字母相同的比较。

`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。

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; // error : cannot convert from int to byte
b += a; // ok
```

（译者注：这个地方应该表述的有误，其实无论 a+b 的值为多少，编译器都会报错，因为 a+b 操作会将 a、b 提升为 int 类型，所以将 int 类型赋值给 byte 就会编译出错）

short s1= 1; s1 = s1 + 1; 该段代码是否有错，有的话怎么改？

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。

short s1= 1; s1 += 1; 该段代码是否有错，有的话怎么改？

+=操作符会自动对右边的表达式结果强转匹配左边的数据类型，所以没错。

& 和 &&的区别

首先记住&是位操作，而&&是逻辑运算符。另外需要记住逻辑运算符具有短路特性，而&不具备短路特性。

```
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");
        }
    }
}
```

以上代码将会抛出空指针异常。

一个 java 文件内部可以有类？（非内部类）

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

内部类的作用

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”管系，它就像是一个独立的实体。

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

final 是一个修饰符，可以修饰变量、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，但是什么时候调用 finalize 没有保证。finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

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。并且需要知道 clone() 方法是一个本地方法，这意味着它是由 c 或 c++ 或 其他本地语言实现的。

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?

几乎所有的人都知道 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。资源名，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，能回答下以下三点就不错了：

1.被 final 修饰的类不可以被继承

2.被 final 修饰的方法不可以被重写

3.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。

4.被 final 修饰的方法，JVM 会尝试将其内联，以提高运行效率

5.被 final 修饰的常量，在编译阶段会存入常量池中。

回答出编译器对 final 域要遵守的两个重排序规则更好：

1.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。

2.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?

类型	位数	字节数
short	2	16
int	4	32
long	8	64
float	4	32
double	8	64
char	2	16

64 位的 JVM 当中,int 的长度是多少?

Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位。意思就是说，在 32 位 和 64 位 的 Java 虚拟机中，int 类型的长度是相同的。

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。

int 和 Integer 谁占用的内存更多?

Integer 对象会占用更多的内存。Integer 是一个对象，需要存储对象的元数据。但是 int 是一个原始类型的数据，所以占用的空间更少。

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)。

String 和 StringBuffer

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。所以尽量不在对 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。

但是需要注意现在 JVM 会对 String 拼接做一定的优化：`String s="This is only "+"simple"+"test"`会被虚拟机直接优化成 `String s="This is only simple test"`，此时就不存在拼接过程。

StringBuffer 和 StringBuilder

StringBuffer 是线程安全的可变字符串，其内部实现是可变数组。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（public static final ）变量也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，因为编译器知道这些变量的值，并且知道这些变量在运行时不能改变。这种方式存在的一个问题是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，但是你的客户端仍然在使用老的值，甚至你已经部署了一个新的 jar。为了避免这种情况，当你在更新依赖 JAR 文件时，确保重新编译你的程序。

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，需要注意的点是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?

我们可以做强制转换，但是 Java 中 int 是 32 位的而 byte 是 8 位的，所以,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

关于垃圾回收

你知道哪些垃圾回收算法?

垃圾回收从理论上非常容易理解,具体的方法有以下几种:

1. 标记-清除
2. 标记-复制
3. 标记-整理
4. 分代回收

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：1.引用计数法; 2.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。

简单的解释一下垃圾回收

Java 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。内存的分配是发生在年轻世代中的。当一个对象存活时间足够长的时候，它就会被复制到年老世代中。对于不同的世代可以使用不同的垃圾回收算法。进行世代划分的出发点是对应用中对象存活时间进行研究之后得出的统计规律。一般来说，一个应用中的大部分对象的存活时间都很短。比如局部变量的存活时间就只在方法的执行过程中。基于这一点，对于年轻世代的垃圾回收算法就可以很有针对性。

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。

进程,线程相关

说说进程，线程，协程之间的区别

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。

你了解守护线程吗？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。

Thread 类中的 start()和 run()方法有什么区别?

start()方法被用来启动新创建的线程，而且 start()内部调用了 run()方法，这和直接调用 run()方法的效果不一样。当你调用 run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

怎么检测一个线程是否持有对象监视器

Thread 类提供了一个 holdsLock(Object obj)方法，当且仅当对象 obj 的监视器被某条线程持有的时候才会返回 true，注意这是一个 static 方法，这意味着"某条线程"指的是当前线程。

Runnable 和 Callable 的区别

Runnable 接口中的 run()方法的返回值是 void，它做的事情只是纯粹地去执行 run()方法中的代码而已；Callable 接口中的 call()方法是有返回值的，是一个泛型，和 Future、FutureTask 配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而 Callable+Future/FutureTask 却可以方便获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务。

什么导致线程阻塞

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 当中有哪几种锁

自旋锁: 自旋锁在 JDK1.6 之后就默认开启了。基于之前的观察，共享数据的锁定状态只会持续很短的时间，为了这一小段时间而去挂起和恢复线程有点浪费，所以这里就做了一个处理，让后面请求锁的那个线程在稍等一会，但是不放弃处理器的执行时间，看看持有锁的线程能否快速释放。为了让线程等待，所以需要让线程执行一个忙循环也就是自旋操作。在 jdk6 之后，引入了自适应的自旋锁，也就是等待的时间不再固定了，而是由上一次在同一个锁上的自旋时间及锁的拥有者状态来决定。

偏向锁: 在 JDK1.之后引入的一项锁优化，目的是消除数据在无竞争情况下的同步原语。进一步提升程序的运行性能。 偏向锁就是偏心的偏，意思是这个锁会偏向第一个获得他的线程，如果接下来的执行过程中，改锁没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。偏向锁可以提高带有同步但无竞争的程序性能，也就是说他并不一定总是对程序运行有利，如果程序中大多数的锁都是被多个不同的线程访问，那偏向模式就是多余的，在具体问题具体分析的前提下，可以考虑是否使用偏向锁。

轻量级锁: 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在 Java SE1.6 里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。

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

如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，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。

为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

Thread.sleep(0)的作用是什么

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true，否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的值 A，只要某次 CAS 操作失败，永远都不可能成功。

什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 的并发度就是 segment 的大小，默认为 16，这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap，这也是 ConcurrentHashMap 对 Hashtable 的最大优势，任何情况下，Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

ConcurrentHashMap 的工作原理

ConcurrentHashMap 在 jdk 1.6 和 jdk 1.8 实现原理是不同的。

jdk 1.6:

ConcurrentHashMap 是线程安全的，但是与 Hashtablea 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.

jdk 1.8

在 jdk 8 中，ConcurrentHashMap 不再使用 Segment 分离锁，而是采用一种乐观锁 CAS 算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

CyclicBarrier 和 CountDownLatch 区别

这两个类非常类似，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二者的区别在于：

> CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某个点上之后，只是给某个数值-1 而已，该线程继续运行。
>
> CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务
>
> CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就不可再用了。

java 中的++操作符线程安全么?

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

你有哪些多线程开发良好的实践?

> 给线程命名
>
> 最小化同步范围

优先使用 volatile

尽可能使用更高层次的并发工具而非 wait 和 notify()来实现线程通信,如 BlockingQueue,Semeaphore

优先使用并发容器而非同步容器.

考虑使用线程池

关于 volatile 关键字

可以创建 Volatile 数组吗?

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

volatile 能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是 64 位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障(write barrier)，读一个 volatile 变量之前，会插入一个读屏障(read barrier)。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

volatile 类型变量提供什么保证?

volatile 主要有两方面的作用:1.避免指令重排 2.可见性保证.例如，JVM 或者 JIT 为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低 32 位和高 32 位)，但 volatile 类型的 double 和 long 就是原子的。

关于集合

Java 中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的 List,Map 的原理更该如此.这里我们看这张图即可:

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

如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，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。

为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

Thread.sleep(0)的作用是什么

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true，否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的值 A，只要某次 CAS 操作失败，永远都不可能成功。

什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 的并发度就是 segment 的大小，默认为 16，这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap，这也是 ConcurrentHashMap 对 Hashtable 的最大优势，任何情况下，Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

ConcurrentHashMap 的工作原理

ConcurrentHashMap 在 jdk 1.6 和 jdk 1.8 实现原理是不同的。

jdk 1.6:

ConcurrentHashMap 是线程安全的，但是与 Hashtablea 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.

jdk 1.8

在 jdk 8 中，ConcurrentHashMap 不再使用 Segment 分离锁，而是采用一种乐观锁 CAS 算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

CyclicBarrier 和 CountDownLatch 区别

这两个类非常类似，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二者的区别在于：

- CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某个点上之后，只是给某个数值-1 而已，该线程继续运行。
- CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务
- CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就不可再用了。

java 中的++操作符线程安全么?

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

你有哪些多线程开发良好的实践?

1.

给线程命名

2.
3.

最小化同步范围

4.
5.

优先使用 volatile

6.
7.

尽可能使用更高层次的并发工具而非 wait 和 notify()来实现线程通信,如 BlockingQueue,Semeaphore

8.
9.

优先使用并发容器而非同步容器.

10.
11.

考虑使用线程池

12.

关于 volatile 关键字

可以创建 Volatile 数组吗?

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

volatile 能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是 64 位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障(write barrier)，读一个 volatile 变量之前，会插入一个读屏障(read barrier)。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

volatile 类型变量提供什么保证?

volatile 主要有两方面的作用:1.避免指令重排 2.可见性保证.例如，JVM 或者 JIT 为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低 32 位和高 32 位)，但 volatile 类型的 double 和 long 就是原子的。

关于集合

Java 中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的 List,Map 的原理更该如此.这里我们看这张图即可:

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

如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，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。

为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

Thread.sleep(0)的作用是什么

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true，否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的

变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的值 A，只要某次 CAS 操作失败，永远都不可能成功。

什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 的并发度就是 segment 的大小，默认为 16，这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap，这也是 ConcurrentHashMap 对 Hashtable 的最大优势，任何情况下，Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

ConcurrentHashMap 的工作原理

ConcurrentHashMap 在 jdk 1.6 和 jdk 1.8 实现原理是不同的。

jdk 1.6:

ConcurrentHashMap 是线程安全的，但是与 Hashtablea 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.

jdk 1.8

在 jdk 8 中，ConcurrentHashMap 不再使用 Segment 分离锁，而是采用一种乐观锁 CAS 算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

CyclicBarrier 和 CountDownLatch 区别

这两个类非常类似，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二者的区别在于：

- CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某个点上之后，只是给某个数值-1 而已，该线程继续运行。
- CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务
- CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就不可再用了。

java 中的++操作符线程安全么?

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

你有哪些多线程开发良好的实践?

1. 给线程命名
2.
3. 最小化同步范围
4.
5.

优先使用 volatile

6.
7.

尽可能使用更高层次的并发工具而非 wait 和 notify()来实现线程通信,如 BlockingQueue,Semeaphore

8.
9.

优先使用并发容器而非同步容器.

10.
11.

考虑使用线程池

12.

关于 volatile 关键字

可以创建 Volatile 数组吗?

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

volatile 能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是 64 位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然

后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

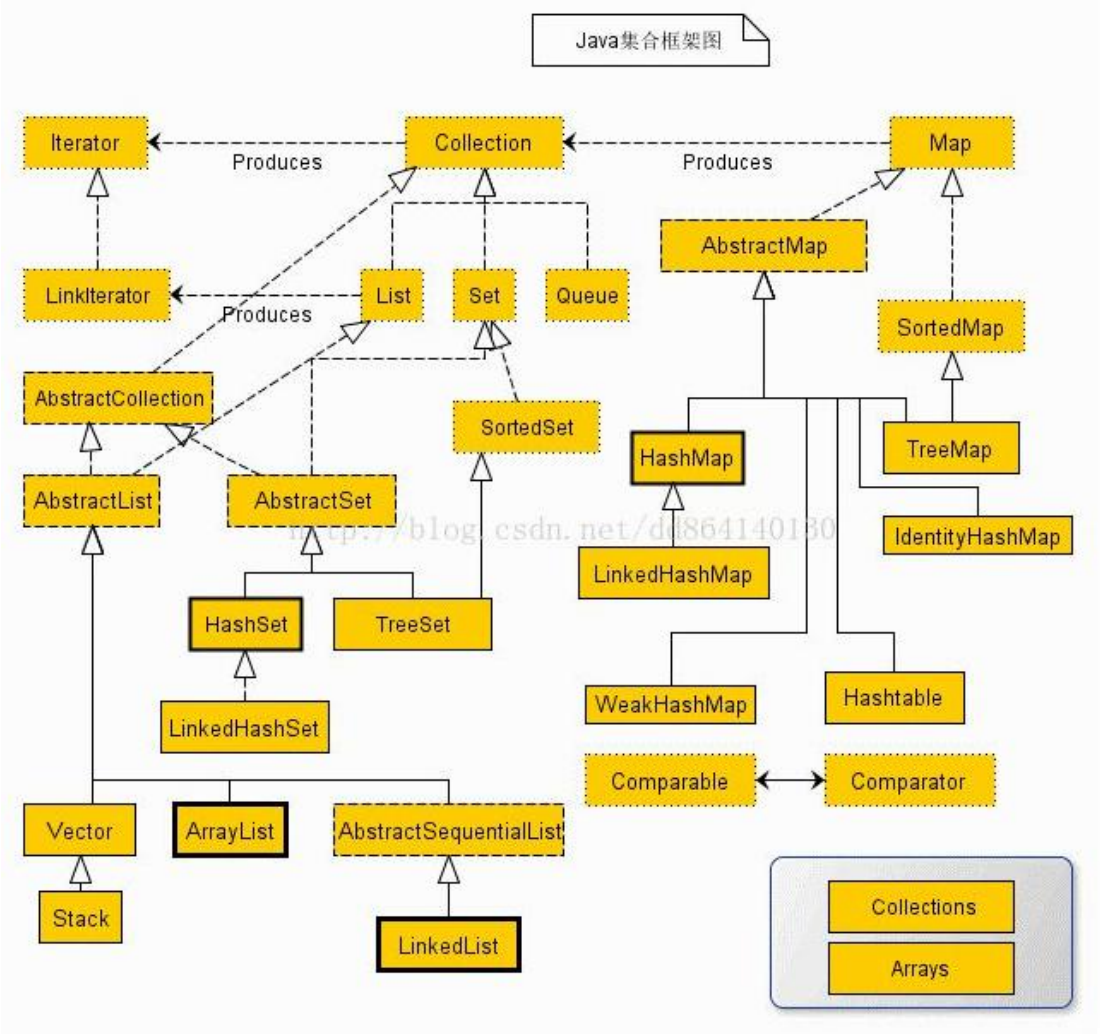
volatile 类型变量提供什么保证?

volatile 主要有两方面的作用:1.避免指令重排 2.可见性保证.例如，JVM 或者 JIT 为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低 32 位和高 32 位)，但 volatile 类型的 double 和 long 就是原子的。

关于集合

Java 中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的 List,Map 的原理更该如此.这里我们看这张图即可:



poll()方法和 remove()方法区别？

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

LinkedHashMap 和 PriorityQueue 的区别

PriorityQueue 是一个优先级队列,保证最高或者最低优先级的的元素总是在队列头部，但是 LinkedHashMap 维持的顺序是元素插入的顺序。当遍历一个 PriorityQueue 时，没有任何顺序保证，但是 LinkedHashMap 课保证遍历顺序是元素插入的顺序。

WeakHashMap 与 HashMap 的区别是什么？

WeakHashMap 的工作与正常的 HashMap 类似，但是使用弱引用作为 key，意思就是当 key 对象没有任何引用时，key/value 将会被回收。

ArrayList 和 LinkedList 的区别?

最明显的区别是 ArrrayList 底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O(1)，而 LinkedList 是 O(n)。

ArrayList 和 Array 有什么区别?

Array 可以容纳基本类型和对象，而 ArrayList 只能容纳对象。

Array 是指定大小的，而 ArrayList 大小是固定的

ArrayList 和 HashMap 默认大小?

在 Java 7 中，ArrayList 的默认大小是 10 个元素，HashMap 的默认大小是 16 个元素（必须是 2 的幂）。这就是 Java 7 中 ArrayList 和 HashMap 类的代码片段。

```
private static final int DEFAULT_CAPACITY = 10;

//from HashMap.java JDK 7
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

Comparator 和 Comparable 的区别?

Comparable 接口用于定义对象的自然顺序，而 comparator 通常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。

如何实现集合排序?

你可以使用有序集合，如 TreeSet 或 TreeMap，你也可以使用有顺序的的集合，如 list，然后通过 Collections.sort() 来排序。

如何打印数组内容

你可以使用 Arrays.toString() 和 Arrays.deepToString() 方法来打印数组。由于数组没有实现 toString() 方法，所以如果将数组传递给 System.out.println() 方法，将无法打印出数组的内容，但是 Arrays.toString() 可以打印每个元素。

LinkedList 的是单向链表还是双向?

双向循环列表，具体实现自行查阅源码。

TreeMap 是实现原理

采用红黑树实现，具体实现自行查阅源码。

遍历 ArrayList 时如何正确移除一个元素

该问题的关键在于面试者使用的是 ArrayList 的 remove() 还是 Iterator 的 remove()方法。这有一段示例代码，是使用正确的方式来实现在遍历的过程中移除元素，而不会出现 ConcurrentModificationException 异常的示例代码。

什么是 ArrayMap?它和 HashMap 有什么区别?

ArrayMap 是 Android SDK 中提供的，非 Android 开发者可以略过。
ArrayMap 是用两个数组来模拟 map，更少的内存占用空间,更高的效率。

HashMap 的实现原理

1. HashMap 概述： HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
2. HashMap 的数据结构： 在 java 编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap 也不例外。HashMap 实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

当我们往 Hashmap 中 put 元素时,首先根据 key 的 hashcode 重新计算 hash 值,根绝 hash 值得到这个元素在数组中的位置(下标),如果该数组在该位置上已经存放了其他元素,那么在这个位置上的元素将以链表的形式存放,新加入的放在链头,最先加入的放入链尾.如果数组中该位置没有元素,就直接将该元素放到数组的该位置上.

需要注意 Jdk 1.8 中对 HashMap 的实现做了优化,当链表中的节点数据超过八个之后,该链表会转为红黑树来提高查询效率,从原来的 O(n)到 O(logn)

你了解 Fail-Fast 机制吗？

Fail-Fast 即我们常说的快速失败，

Fail-fast 和 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。

关于日期

SimpleDateFormat 是线程安全的吗?

非常不幸，DateFormat 的所有实现，包括 SimpleDateFormat 都不是线程安全的，因此你不应该在多线程序中使用，除非是在对外线程安全的环境中使用，如 将 SimpleDateFormat 限制在 ThreadLocal 中。如果你不这么做，在解析或者格式化日期的时候，可能会获取到一个不正确的结果。因此，从日期、时间处理的所有实践来说，我强力推荐 joda-time 库。

如何格式化日期?

Java 中，可以使用 SimpleDateFormat 类或者 joda-time 库来格式日期。DateFormat 类允许你使用多种流行的格式来格式化日期。参见答案中的示例代码，代码中演示了将日期格式化成不同的格式，如 dd-MM-yyyy 或 ddMMyyyy。

关于异常

简单描述 java 异常体系

相比没有人不了解异常体系,关于异常体系的更多信息可以见

什么是异常链

详情直接参见上面的白话异常机制，不做解释了。

throw 和 throws 的区别

throw 用于主动抛出 java.lang.Throwable 类的一个实例化对象，意思是说你可以通过关键字 throw 抛出一个 Error 或者一个 Exception，如：`throw new IllegalArgumentException("size must be multiple of 2")`,
而 throws 的作用是作为方法声明和签名的一部分，方法被抛出相应的异常以便调用者能处理。Java 中，任何未处理的受检查异常强制在 throws 子句中声明。

关于序列化

Java 中，Serializable 与 Externalizable 的区别

Serializable 接口是一个序列化 Java 类的接口，以便于它们可以在网络上传输或者可以将它们的状态保存在磁盘上，是 JVM 内嵌的默认序列化方式，成本高、脆弱而且不安全。
Externalizable 允许你控制整个序列化过程，指定特定的二进制格式，增加安全机制。

关于 JVM

JVM 特性

平台无关性.
Java 语言的一个非常重要的特点就是与平台的无关性。而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而

引入 Java 语言虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用模式 Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。

简单解释一下类加载器

有关类加载器一般会问你四种类加载器的应用场景以及双亲委派模型,

简述堆和栈的区别

VM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。

简述 JVM 内存分配

基本数据类型比变量和对象的引用都是在栈分配的。

堆内存用来存放由 new 创建的对象和数组。

类变量（static 修饰的变量），程序在一加载的时候就在堆中为类变量分配内存，堆中的内存地址存放在栈中。

实例变量：当你使用 java 关键字 new 的时候，系统在堆中开辟并不一定是连续的空间分配给变量，是根据零散的堆内存地址，通过哈希算法换算为一长串数字以表征这个变量在堆中的"物理位置",实例变量的生命周期–当实例变量的引用丢失后，将被 GC（垃圾回收器）列入可回收"名单"中，但并不是马上就释放堆中内存。

局部变量: 由声明在某方法，或某代码段里（比如 for 循环），执行到它的时候在栈中开辟内存，当局部变量一但脱离作用域，内存立即释放。

其他

java 当中采用的是大端还是小端?

XML 解析的几种方式和特点

DOM, SAX, PULL 三种解析方式：

DOM:消耗内存：先把 xml 文档都读到内存中，然后再用 DOM API 来访问树形结构，并获取数据。这个写起来很简单，但是很消耗内存。要是数据过大，手机不够牛逼，可能手机直接死机

SAX:解析效率高，占用内存少，基于事件驱动的：更加简单地说就是对文档进行顺序扫描，当扫描到文档(document)开始与结束、元素(element)开始与结束、文档(document)结束等地方时通知事件处理函数，由事件处理函数做相应动作，然后继续同样的扫描，直至文档结束。

PULL:与 SAX 类似，也是基于事件驱动，我们可以调用它的 next（）方法，来获取下一个解析事件（就是开始文档，结束文档，开始标签，结束标签），当处于某个元素时可以调用 XmlPullParser 的 getAttributte()方法来获取属性的值，也可调用它的 nextText()获取本节点的值。

JDK 1.7 特性

然 JDK 1.7 不像 JDK 5 和 8 一样的大版本，但是，还是有很多新的特性，如 try-with-resource 语句，这样你在使用流或者资源的时候，就不需要手动关闭，Java 会自动关闭。Fork-Join 池某种程度上实现 Java 版的 Map-reduce。允许 Switch 中有 String 变量和文本。菱形操作符(<>)用于类型推断，不再需要在变量声明的右边申明泛型，因此可以写出可读写更强、更简洁的代码。

JDK 1.8 特性

java 8 在 Java 历史上是一个开创新的版本，下面 JDK 8 中 5 个主要的特性：
Lambda 表达式，允许像对象一样传递匿名函数
Stream API，充分利用现代多核 CPU，可以写出很简洁的代码
Date 与 Time API，最终，有一个稳定、简单的日期和时间库可供你使用
扩展方法，现在，接口中可以有静态、默认方法。
重复注解，现在你可以将相同的注解在同一类型上使用多次。

Maven 和 ANT 有什么区别?

虽然两者都是构建工具，都用于创建 Java 应用，但是 Maven 做的事情更多，在基于“约定优于配置”的概念下，提供标准的 Java 项目结构，同时能为应用自动管理依赖（应用中所依赖的 JAR 文件。

JDBC 最佳实践

优先使用批量操作来插入和更新数据

使用 PreparedStatement 来避免 SQL 漏洞

使用数据连接池

通过列名来获取结果集

IO 操作最佳实践

使用有缓冲的 IO 类,不要单独读取字节或字符

使用 NIO 和 NIO 2 或者 AIO,而非 BIO

在 finally 中关闭流

使用内存映射文件获取更快的 IO