## 1、什么是线程？

线程是操作系统中能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位

## 2、线程和进程有什么区别？

线程是进程的子集。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。每个线程都有单独的栈内存来存储本地数据。

## 3、如何在Java中实现线程

编写多线程一般有三种方法：继承Thread、实现Runnable、实现Callable

* Callable规定的方法是call()，Runnable规定的方法是run()
* Callable的任务执行后可返回值，而Runnable的任务是不能返回值
* call方法可以抛出异常，但是run()方法不行
* 运行Callable任务可以拿到一个Future对象，Future表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并获得计算的结果。计算完成之后只能使用get方法来获得结果，如果线程没有执行完Future.get()可能会阻塞当前线程的执行；如果线程出现异常，Future.get()可能会抛出InterruptedException或者ExecutionException；如果线程已经取消，会抛出CancellationException。取消由cancel()方法来执行。isDone确定任务是正常执行还是被取消了。一旦计算完成，就不能再取消计算。

Thread类的实例就是一个线程但是它需要调用java.lang.Runnable接口来执行，由于线程类本身就是调用的Runnable接口所以可以继承Thread类或者实现Runnable接口。

## 4、用Runnable还是Thread

如果是要继承其他类，那么调用Runnable接口就好了

## 5、Thread类中的start()和run()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()才会启动新的线程。

## **6、java中Runnable和Callable有什么区别？**

Callable的call()可以返回值和抛出异常，而Runnable的run()没有这些功能。Callable可以返回装载有计算结果的Future对象

## 8 Java中CyclicBarrier 和 CountDownLatch有什么不同？

两者都可以让一组线程等待其他线程。CountDownLatch强调一个线程等多个线程，CyclicBarrier强调多个线程相互等待。

CountDownLatch，一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

主要方法

public CountDownLatch(int count); //参数指定了计数的次数

public void countDown(); //当线程调用此方法计数减一

public void await throws InterruptedException; //调用此方法会一直阻塞当前线程，直到计数器的值为0

CountDownLatch类使用如下：

public class CountDownLatchDemo {  
 final static SimpleDateFormat *sdf* = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
  
 public static void main (String[] args) throws InterruptedException {  
 CountDownLatch count = new CountDownLatch(2);  
 Worker w1 = new Worker("zhang san", 5000, count);  
 Worker w2 = new Worker("li si", 8000, count);  
 w1.start();  
 w2.start();  
 count.await(); //表示在这里等待(判断哪个线程等待就看count在哪里定义的，这里是主线程)

System.*out*.println("all work completed at " + *sdf*.format(new Date()));  
 }  
  
 static class Worker extends Thread {  
 String workName;  
 int workTime;  
 CountDownLatch latch;  
 public Worker (String workName, int workTime,CountDownLatch latch) {  
 this.workName = workName;  
 this.workTime = workTime;  
 this.latch = latch;  
 }  
 public void run () {  
 System.*out*.println("Worker " + workName + " do work begin at " + *sdf*.format(new Date()));  
 doWork();  
 System.*out*.println("Worker " + workName + " do work end at " + *sdf*.format(new Date()));  
 latch.countDown();  
 }  
  
 private void doWork () {  
 try{  
 Thread.*sleep*(workTime);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }

输出结果：

Worker zhang san do work begin at 2017-04-07 13:51:00

Worker li si do work begin at 2017-04-07 13:51:00

Worker zhang san do work end at 2017-04-07 13:51:05

Worker li si do work end at 2017-04-07 13:51:08

all work completed at 2017-04-07 13:51:08

如果将main方法中的count.await();注释掉，结果就会变成：

all work completed at 2017-04-07 13:53:56

Worker zhang san do work begin at 2017-04-07 13:53:56

Worker li si do work begin at 2017-04-07 13:53:56

Worker zhang san do work end at 2017-04-07 13:54:01

Worker li si do work end at 2017-04-07 13:54:04

可见count.await()方法的作用就是阻塞当前线程(当前线程的意思就是count.await()在哪个线程调用那个线程就是当前线程，在本例中就是main方法的主线程)直到latch.countDown()的计数变为0，其他线程完成各自的任务，然后恢复主线程。

该类通常用在如下场景：主线程开始执行前要等待n个线程完成各自的任务。

CyclicBarrier类使用实例如下：

public class TestCyclicBarrier {  
 private static final int *THREAD\_NUM* = 5;  
 static class WorkThread implements Runnable {  
 CyclicBarrier barrier;  
 public WorkThread (CyclicBarrier barrier) {  
 this.barrier = barrier;  
 }  
 public void run () {  
 try {  
 System.*out*.println("is waiting...");  
 barrier.await(); //线程在这里等待，直到所有线程到达barrier  
 System.*out*.println("ID " + Thread.*currentThread*().getId() + " is working...");  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
 }  
 public static void main (String[] args) {  
 CyclicBarrier barrier = new CyclicBarrier(*THREAD\_NUM*, new Runnable() {  
 @Override //当所有线程到达barrier时执行  
 public void run() {  
 System.*out*.println("Completed...");  
 }  
 });  
 for (int i = 0; i < *THREAD\_NUM*; i++) {  
 new Thread(new WorkThread(barrier)).start();  
 }  
 }  
}

输出结果：

is waiting...

is waiting...

is waiting...

is waiting...

is waiting...

Completed...

ID 13 is working...

ID 12 is working...

ID 15 is working...

ID 16 is working...

ID 14 is working...

## 9) Java内存模型是什么？

《Java并发编程实战》第十六章

## 10) Java中的volatile变量是什么？

使变量在各个线程之间透明

## 11) 什么是线程安全？Vector是一个线程安全类吗？

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码，如果每次运行的结果和单线程运行的结果是一样的，而且其他变量的值也和预期是一样的，就是线程安全的。Vector是使用同步方法实现线程安全的，而和它相似的ArrayList不是线程安全的。

12) Java中什么是竞态条件？ 举个例子说明

## 13) Java中如何停止一个线程？

在JDK1.1的时候，Java中有stop()、suspend()方法等来终止线程，但是考虑到这些方法可能会引起死锁，所以后来就废除了这些方法，后面反而没有明确的方法来终止线程。当run()方法或者main()方法执行完成，或者中间发生异常而退出run()方法，则该线程结束生命周期。

## 14) 一个线程运行时发生异常会怎样？

这是我在一次面试中遇到的一个[很刁钻的Java面试题](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html), 简单的说，如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候JVM会使用Thread.getUncaughtExceptionHandler()来查询线程的UncaughtExceptionHandler并将线程和异常作为参数传递给handler的uncaughtException()方法进行处理。

## 15） 如何在两个线程间共享数据？

## 16) Java中notify 和 notifyAll有什么区别？

## 17) 为什么wait, notify 和 notifyAll这些方法不在thread类里面？

这是个设计相关的问题，它考察的是面试者对现有系统和一些普遍存在但看起来不合理的事物的看法。回答这些问题的时候，你要说明为什么把这些方法放在Object类里是有意义的，还有不把它放在Thread类里的原因。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。

## 18) 什么是ThreadLocal变量？

## 19) 什么是FutureTask？

## 20) Java中interrupted() 和 isInterrupted()方法的区别？

前者会将中断状态清楚而后者不会。Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其他线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有可能被其他线程调用中断来改变。

## 21) 为什么wait和notify方法要在同步块中调用？

## 22) 为什么你应该在循环中检查等待条件?

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在notify()方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用wait()方法效果更好的原因，你可以在[Eclipse](http://res.importnew.com/eclipse)中创建模板调用wait和notify试一试。如果你想了解更多关于这个问题的内容，我推荐你阅读《[Effective Java](http://www.amazon.com/gp/product/B000WJOUPA/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B000WJOUPA&linkCode=as2&tag=job0ae-20)》这本书中的线程和同步章节。

## 23) Java中的同步集合与并发集合有什么区别？

同步集合和并发集合都支持线程安全，他们之间主要的区别体现在性能和可扩展性，还有它们如何实现的线程安全。同步HashMap，HashTable，HashSet,Vector,ArrayList相比他们并发的实现(比如，ConcurrentHashMap，CopyOnWriteArrayList，CopyOnWriteHashSet)会慢得多。造成如此慢的原因是**锁**，同步集合会把整个Map或List锁起来，而并发集合不会。并发集合实现线程安全是通过使用先进的和成熟的技术像锁剥离。**比如ConcurrentHashMap会把整个Map划分成几个片段，只对相关的几个片段上锁，同时允许多个线程访问其他未上锁的片段**，同样，CopyOnWriteArrayList允许多个线程以非同步的方式读，当有线程写的时候它会将整个List复制一个副本给它。

同步集合类(Hashtable和Vector)还有同步集合包装类(Collections.synchronizedMap()和Collections.synchronizedList)提供了一个基本的有条件的线程安全的Map和List的实现。所以Hashtale和ConcurrentHashMap的区别是什么，它们都可以在多线程环境中使用，但一旦Hashtable的大小变得相当大的时候其性能会降低，以为迭代的时间会被长时间锁定。

## 24） Java中堆和栈有什么不同？

## 25） 什么是线程池？ 为什么要使用它？

## 26） 如何写代码来解决生产者消费者问题？

## 27） 如何避免死锁？

## 28) Java中活锁和死锁有什么区别？

## 29） 怎么检测一个线程是否拥有锁？

在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁。

## 30) 你如何在Java中获取线程堆栈？

对于不同的操作系统，有多种方法来获得Java进程的线程堆栈。当你获取线程堆栈时，JVM会把所有线程的状态存到日志文件或者输出到控制台。在Windows你可以使用Ctrl + Break组合键来获取线程堆栈，[**Linux**](http://lib.csdn.net/base/linux)下用kill -3命令。你也可以用jstack这个工具来获取，它对线程id进行操作，你可以用jps这个工具找到id。

## 31) JVM中哪个参数是用来控制线程的栈堆栈小的

这个问题很简单， -Xss参数用来控制线程的堆栈大小。

## 32） Java中synchronized 和 java.util.concurrent.locks.Lock 有什么不同？

**相同点**：lock能完成synchronized的所有功能

**不同点**：Lock有比synchronized更精确的线程语义和更好的性能。它的优势有：

* 可以使锁更公平
* 可以使线程在等待锁的时候响应中断
* 可以让线程尝试获取锁，并在无法获取锁的时候立刻返回或者等待一段时间
* 可以在不同的范围，以不同的顺序获取和释放锁

synchronized会自动释放锁，而Lock一定要求程序员手动释放，并且必须在finally从句中释放。

## 33） 有三个线程T1，T2，T3，怎么确保它们按顺序执行？

## 34) Thread类中的yield方法有什么作用？

## 35） Java中ConcurrentHashMap的并发度是什么？

ConcurrentHashMap把实际map划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16，这样在多线程情况下就能避免争用。

## 36） Java中Semaphore是什么？

Java中的Semaphore是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore只对可用许可的号码进行计数，并采取相应的行动。信号量常常用于多线程的代码中，比如[**数据库**](http://lib.csdn.net/base/mysql)连接池。

## 37）如果你提交任务时，线程池队列已满。会时发会生什么？

这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果一个任务不能被调度执行那么ThreadPoolExecutor’s submit()方法将会抛出一个RejectedExecutionException异常。

## 38) Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

## 39) 什么是阻塞式方法？

## 41） Java中invokeAndWait 和 invokeLater有什么区别？

## 43) 如何在Java中创建Immutable对象？

这个问题看起来和多线程没什么关系， 但不变性有助于简化已经很复杂的并发程序。Immutable对象可以在没有同步的情况下共享，降低了对该对象进行并发访问时的同步化开销。可是Java没有@Immutable这个注解符，要创建不可变类，要实现下面几个步骤：通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝。

## 44） Java中的ReadWriteLock是什么？

一般而言，读写锁是用来提升并发程序性能的锁分离技术的成果。Java中的ReadWriteLock是Java 5 中新增的一个接口，一个ReadWriteLock维护一对关联的锁，一个用于只读操作一个用于写。在没有写线程的情况下一个读锁可能会同时被多个读线程持有。写锁是独占的，你可以使用JDK中的ReentrantReadWriteLock来实现这个规则，它最多支持65535个写锁和65535个读锁。

## 45) 多线程中的忙循环是什么?

忙循环就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了

## 46）volatile 变量和 atomic 变量有什么不同？

这是个有趣的问题。首先，volatile 变量和 atomic 变量看起来很像，但功能却不一样。Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

## 47) 如果同步块内的线程抛出异常会发生什么？

这个问题坑了很多Java程序员，若你能想到锁是否释放这条线索来回答还有点希望答对。无论你的同步块是正常还是异常退出的，里面的线程都会释放锁，所以对比锁接口我更喜欢同步块，因为它不用我花费精力去释放锁，该功能可以在[finally block](http://javarevisited.blogspot.com/2012/11/difference-between-final-finally-and-finalize-java.html)里释放锁实现。

## 48） 单例模式的双检锁是什么？

## 49） 如何在Java中创建线程安全的Singleton？

## 50) 写出3条你遵循的多线程最佳实践

**给你的线程起个有意义的名字。**  
这样可以方便找bug或追踪。OrderProcessor, QuoteProcessor or TradeProcessor 这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。

**避免锁定和缩小同步的范围**  
锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。

**多用同步类少用wait 和 notify**  
首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。

**多用并发集合少用同步集合**  
这是另外一个容易遵循且受益巨大的最佳实践，并发集合比同步集合的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap。

## 51) 如何强制启动一个线程？

使用方法join()，该方法让一个线程强制运行，线程强制运行期间，只有该线程拥有CPU的执行权，其他线程无法运行，必须等待此线程完成之后才可以继续执行。

**public** **class** JoinDemo **implements** Runnable {

/\* (non-Javadoc)

\* @see java.lang.Runnable#run()

\*/

**public** **void** run() {

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

**int** i = 0;

**while** (i < 5) {

i++;

System.*out*.println("JoinDemo");

}

}

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

JoinDemo joinDemo = **new** JoinDemo();

Thread thread = **new** Thread(joinDemo);

thread.start();

thread.join();

**int** i = 0;

**while** (i < 5) {

i++;

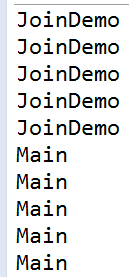
System.*out*.println("Main");

}

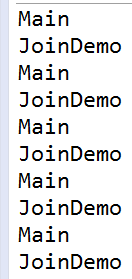
}

}

上述代码的执行结果：



如果将thread.join()注释掉结果为：



## 52) Java中的fork join框架是什么？

没用过

## 53） Java多线程中调用wait() 和 sleep()方法有什么不同？

1. sleep属于Thread，wait属于Object
2. 最主要的是sleep没有释放锁，而wait释放了锁，使得其他线程可以使用同步控制块或者方法
3. wait、notify和notifyAll只能在同步控制方法或者同步控制块里使用，而sleep可以在任何地方使用。

## 54) 用户线程和守护线程有什么区别

当我们在Java中创建一个线程，它就称为用户线程。一个守护线程是在后台执行并且不会阻止JVM终止的线程。当没有用户线程在运行的时候，JVM关闭程序并且退出。一个守护线程创建的子线程依然是守护线程。

## 55) 什么是线程调度器和时间分片

线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间片。一旦我们创建一个线程并启动它，它的执行更依赖于线程调度器的实现。时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。线程调度并不受到JVM控制，所以由应用程序控制它是更好的选择(也就是说不要让你的线程依赖于线程的优先级)。

## 56) 在多线程中什么是上下文切换

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

## 57) 什么是Java线程转储，如何得到它？

线程转储是一个JVM活动线程的列表，它对于分析系统瓶颈和死锁非常有用。有很多方法可以获得线程转储------使用Profiler，Kill -3命令，jstack工具等等。jstack更容易使用而且是JDK自带的。由于它是一个基于终端的工具，所以我们可以编写一些脚本去定时的产生线程转储以待分析。

## 58) 什么是Callable和Future

Callable和Runnable接口很相似，但是它可以返回一个对象或者抛出一个异常

Callable使用泛型去定义它的返回类型

## 59) 什么时候抛出InvalidMonitorStateException异常，为什么？

调用wait()/notify()/notifyAll()中的任何一个方法时，如果当前线程没有获得该对象的锁，那么就会抛出IllegalMonitorStateException的异常(也就是说程序在没有执行对象的任何同步块或者同步方法时，仍然尝试调用wait()/notify()/notifyAll()时)。由于该异常时RuntimeException的子类，所以该异常不一定要捕获(或者也可以捕获)。作为RuntimeException，此类异常不会在wait()、notify()、notifyAll的方法签名提及。

## 60) 什么是死锁

死锁就是两个或两个以上的线程无限的阻塞，线程之间相互等待所需资源。这种情况可能发生在当两个尝试获取其他资源的锁，而每个线程又陷入无限等待其他资源锁的释放，除非一个用户进程被终止。就Java API而言，线程死锁可能发生在以下情况：

1. 当两个线程相互调用Thread.join()
2. 当两个线程使用嵌套的同步块，一个线程占用了另外一个线程必须的锁，相互等待时被阻塞就有可能出现死锁。

## 61) 什么是线程饿死，什么是活锁

## 怎样检测一个线程是否拥有锁？

使用Thread.holdsLock(锁)方法来检测当前线程是否拥有某个对象的锁。

public class Main {  
 static Object *o* = new Object();  
 public static void main (String[] args){  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 synchronized (*o*){  
 System.*out*.println("Current thread has a lock? " + Thread.*holdsLock*(*o*));  
 }  
 }  
 }).start();  
 System.*out*.println("Main thread has a lock? " + Thread.*holdsLock*(*o*));  
 }  
}