Java线程与线程池面试问题

目录

[1.1. 在java中wait和sleep方法的不同？ 1](#_Toc2958)

[1.2. 为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？ 2](#_Toc32585)

[1.3. Java中你怎样唤醒一个阻塞的线程？ 2](#_Toc5485)

[1.4. 在Java中CycliBarriar和CountdownLatch有什么区别？ 3](#_Toc181)

[1.5. 你在多线程环境中遇到的常见的问题是什么？你是怎么解决它的？ 4](#_Toc28856)

[2. 线程池问题 4](#_Toc6737)

[2.1. 什么是线程池： 4](#_Toc17168)

[2.2.      一个线程池包括以下四个基本组成部分：线程池管理器（ThreadPool）工作线程（PoolWorker）任务接口（Task）任务队列（taskQueue）： 5](#_Toc24803)

[2.3. 2.常见线程池newFixedThreadExecutor(n) ③newCacheThreadExecutor（推荐使用） 5](#_Toc25894)

[2.4. 二：线程池线程池的作用： 5](#_Toc3414)

[2.5. 为什么要用线程池: 6](#_Toc19282)

[2.6. 排队有三种通用策略：直接提交。无界队列。有界队列。 6](#_Toc11625)

[3. 线程池常见参数 6](#_Toc13298)

[3.1. 为什么要使用线程池？ 8](#_Toc22302)

[3.2. 线程池有什么作用？ 8](#_Toc20915)

[3.3. 说说几种常见的线程池及使用场景 8](#_Toc23640)

[3.4. 线程池都有哪几种工作队列 8](#_Toc6450)

[3.5. 怎么理解无界队列和有界队列 9](#_Toc10360)

## 在java中wait和sleep方法的不同？

通常会在电话面试中经常被问到的Java线程面试问题。   
最大的不同是在等待时wait会释放锁，而sleep一直持有锁。Wait通常被用于线程间交互，sleep通常被用于暂停执行

## 为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？

这是另一个非常经典的java多线程面试问题。这也是我刚开始写线程程序时候的困惑。现在这个问题通常在电话面试或者是在初中级Java面试的第一轮被问到。这个问题的回答应该是这样的，当你调用start()方法时你将创建新的线程，并且执行在run()方法里的代码。但是如果你直接调用run()方法，它不会创建新的线程也不会执行调用线程的代码。

简单点来说：   
new一个Thread，线程进入了新建状态;调用start()方法，线程进入了就绪状态，当分配到时间片后就可以开始运行了。   
start()会执行线程的相应准备工作，然后自动执行run()方法的内容。是真正的多线程工作。   
而直接执行run()方法，会把run方法当成一个mian线程下的普通方法去执行，并不会在某个线程中执行它，这并不是多线程工作。

## Java中你怎样唤醒一个阻塞的线程？

这是个关于线程和阻塞的棘手的问题，它有很多解决方法。如果线程遇到了IO阻塞，我并且不认为有一种方法可以中止线程。如果线程因为调用wait()、sleep()、或者join()方法而导致的阻塞，你可以中断线程，并且通过抛出InterruptedException来唤醒它。我之前写的《How to deal with blocking methods in java》有很多关于处理线程阻塞的信息。

这个我们先简单粗暴地对某些阻塞方法进行分类：   
- 会抛出InterruptedException的方法：wait、sleep、join、Lock.lockInterruptibly等，针对这类方法，我们在线程内部处理好异常(要不完全内部处理，要不把这个异常抛出去)，然后就可以实现唤醒。   
- 不会抛InterruptedException的方法：Socket的I/O,同步I/O，Lock.lock等。对于I/O类型，我们可以关闭他们底层的通道，比如Socket的I/O，关闭底层套接字，然后抛出异常处理就好了;比如同步I/O，关闭底层Channel然后处理异常。对于Lock.lock方法，我们可以改造成Lock.lockInterruptibly方法去实现。

## 在Java中CycliBarriar和CountdownLatch有什么区别？

同步屏障CyclicBarrier

ountDownLatch是一个非常实用的多线程控制工具类,称之为“倒计时器”,它允许一个或多个线程一直等待,直到其他线程的操作执行完后再执行

这个线程问题主要用来检测你是否熟悉JDK5中的并发包。这两个的区别是CyclicBarrier可以重复使用已经通过的障碍，而CountdownLatch不能重复使用。

还要注意一点的区别：   
CountDownLatch : 一个线程(或者多个)， 等待另外N个线程完成某个事情之后才能执行。   
CyclicBarrier : N个线程相互等待，任何一个线程完成之前，所有的线程都必须等待

## 你在多线程环境中遇到的常见的问题是什么？你是怎么解决它的？

多线程和并发程序中常遇到的有Memory-interface、竞争条件、死锁、活锁和饥饿。问题是没有止境的，如果你弄错了，将很难发现和调试。这是大多数基于面试的，而不是基于实际应用的Java线程问题。

此类问题请大家面试的时候提前准备，方便交流，如果实在找不出来，可以想想自己平时解决问题的思路，总结下来告诉考官。

# 线程池问题

## 什么是线程池：

1. java.util.concurrent.Executors提供了一个 java.util.concurrent.Executor接口的实现用于创建线程池

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。      
    假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。

   如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。

## 一个线程池包括以下四个基本组成部分：线程池管理器（ThreadPool）工作线程（PoolWorker）任务接口（Task）任务队列（taskQueue）：

                1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；  
                2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；  
                3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；  
                4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

   线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。

## 2.常见线程池newFixedThreadExecutor(n) ③newCacheThreadExecutor（推荐使用）

①newSingleThreadExecutor

单个线程的线程池，即线程池中每次只有一个线程工作，单线程串行执行任务

②newFixedThreadExecutor(n)

固定数量的线程池，没提交一个任务就是一个线程，直到达到线程池的最大数量，然后后面进入等待队列直到前面的任务完成才继续执行

③newCacheThreadExecutor（推荐使用）

可缓存线程池，**当线程池大小超过了处理任务所需的线程，那么就会回收部分空闲（一般是60秒无执行）的线程**，当有任务来时，又智能的添加新线程来执行。

④newScheduleThreadExecutor

大小无限制的线程池，支持定时和周期性的执行线

## **二：线程池**线程池的作用：****

线程池作用就是限制系统中执行线程的数量。  
     根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程排队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池中有等待的工作线程，就可以开始运行了；否则进入等待队列。

## ****为什么要用线程池:****

1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。

## 排队有三种通用策略：****直接提交。无界队列。有界队列。****

**直接提交。**工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

**无界队列。**使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

**有界队列。**当使用有限的 maximumPoolSizes时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量

# 线程池常见参数

**keepAliveTime**

jdk中的解释是：当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。

有点拗口，其实这个不难理解，在使用了“池”的应用中，大多都有类似的参数需要配置。比如[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://www.cnblogs.com/aspirant/p/_blank)连接池，DBCP中的maxIdle，minIdle参数。

什么意思？接着上面的解释，后来向老板派来的工人始终是“借来的”，俗话说“有借就有还”，但这里的问题就是什么时候还了，如果借来的工人刚完成一个任务就还回去，后来发现任务还有，那岂不是又要去借？这一来一往，老板肯定头也大死了。

合理的策略：既然借了，那就多借一会儿。直到“某一段”时间后，发现再也用不到这些工人时，便可以还回去了。这里的某一段时间便是keepAliveTime的含义，TimeUnit为keepAliveTime值的度量。

**RejectedExecutionHandler**

另一种情况便是，即使向老板借了工人，但是任务还是继续过来，还是忙不过来，这时整个队伍只好拒绝接受了。

RejectedExecutionHandler接口提供了对于拒绝任务的处理的自定方法的机会。在ThreadPoolExecutor中已经默认包含了4中策略，因为源码非常简单，这里直接贴出来。

**CallerRunsPolicy**：线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。

这个策略显然不想放弃执行任务。但是由于池中已经没有任何资源了，那么就直接使用调用该execute的线程本身来执行。

**AbortPolicy：**处理程序遭到拒绝将抛出运行时RejectedExecutionException

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

throw new RejectedExecutionException();

}

 这种策略直接抛出异常，丢弃任务。

**DiscardPolicy：**不能执行的任务将被删除

 这种策略和AbortPolicy几乎一样，也是丢弃任务，只不过他不抛出异常。

**DiscardOldestPolicy：**如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）

## 为什么要使用线程池？

创建线程和销毁线程的花销是比较大的，这些时间有可能比处理业务的时间还要长。这样频繁的创建线程和销毁线程，再加上业务工作线程，消耗系统资源的时间，可能导致系统资源不足。（我们可以把创建和销毁的线程的过程去掉）

## 线程池有什么作用？

1、提高效率 创建好一定数量的线程放在池中，等需要使用的时候就从池中拿一个，这要比需要的时候创建一个线程对象要快的多。  
2、方便管理 可以编写线程池管理代码对池中的线程同一进行管理，比如说启动时有该程序创建100个线程，每当有请求的时候，就分配一个线程去工作，如果刚好并发有101个请求，那多出的这一个请求可以排队等候，避免因无休止的创建线程导致系统崩溃。

## 说说几种常见的线程池及使用场景

1、newSingleThreadExecutor  
创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

## 线程池都有哪几种工作队列

1、ArrayBlockingQueue  
是一个基于数组结构的**有界阻塞队列**，此队列按 FIFO（先进先出）原则对元素进行排序。  
2、LinkedBlockingQueue  
一个基于链表结构的**阻塞队列**，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列  
3、SynchronousQueue  
一个不存储元素的**阻塞队列**。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。  
4、PriorityBlockingQueue  
一个具有优先级的**无限阻塞队列**。

## 怎么理解无界队列和有界队列

**有界队列**  
1.初始的poolSize < corePoolSize，提交的runnable任务，会直接做为new一个Thread的参数，立马执行 。  
2.当提交的任务数超过了corePoolSize，会将当前的runable提交到一个block queue中。  
3.有界队列满了之后，如果poolSize < maximumPoolsize时，会尝试new 一个Thread的进行救急处理，立马执行对应的runnable任务。  
4.如果3中也无法处理了，就会走到第四步执行reject操作。  
**无界队列**  
与有界队列相比，除非系统资源耗尽，否则无界的任务队列不存在任务入队失败的情况。当有新的任务到来，系统的线程数小于corePoolSize时，则新建线程执行任务。当达到corePoolSize后，就不会继续增加，若后续仍有新的任务加入，而没有空闲的线程资源，则任务直接进入队列等待。若任务创建和处理的速度差异很大，无界队列会保持快速增长，直到耗尽系统内存。

当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略。