多线程最终版 ---自我总结

9 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，而且在面试中会经常被问到。很简单，但是很多人都会答不上来！

new一个Thread，线程进入了新建状态;调用start()方法，会启动一个线程并使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，这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，所以这并不是多线程工作。

总结： 调用start方法方可启动线程并使线程进入就绪状态，而run方法只是thread的一个普通方法调用，还是在主线程里执行。

**1.简述线程，程序、进程的基本概念。以及他们之间关系是什么？**

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

**2. 何为多线程？**

多线程就是多个线程同时运行或交替运行。单核CPU的话是顺序执行，也就是交替运行。多核CPU的话，因为每个CPU有自己的运算器，所以在多个CPU中可以同时运行。

**3. sleep()方法和wait()方法简单对比**

- 两者最主要的区别在于：sleep方法没有释放锁，而wait方法释放了锁 。

- 两者都可以暂停线程的执行。

- Wait通常被用于线程间交互/通信，sleep通常被用于暂停执行。

- wait()方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的notify()或者notifyAll()方法。sleep()方法执行完成后，线程会自动苏醒。

多线程：指的是这个程序（一个进程）运行时产生了不止一个线程

并行与并发：

并行：多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。

并发：通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力。

java.util.concurrent.Executor : 负责线程的使用与调度的根接口

|–ExecutorService：Executor的子接口，线程池的主要接口

|–ThreadPoolExecutor：ExecutorService的实现类

|–ScheduledExecutorService：ExecutorService的子接口，负责线程的调度

|–ScheduledThreadPoolExecutor：既继承了ThreadPoolExecutor，

同时实现了ScheduledExecutorService

在实际使用过程中，我们不需要自己手动new出线程池，juc的工具类Executors为我们提供了方法用以创建线程池： （注意：最近阿里发布的 Java开发手册中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。下面转载部分有较为详细的介绍）

**ExecutorService** newFixedThreadPool() : 创建固定大小的线程池

**ExecutorService** newCachedThreadPool() : 缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量。

**ExecutorService** newSingleThreadExecutor() : 创建单个线程池。线程池中只有一个线程

**ScheduledExecutorService** newScheduledThreadPool() : 创建固定大小的线程池，可以延迟或定时的执行任务。

**ThreadPoolExecutor**

先看看如何使用ThreadPoolExecutor创建线程池：

corePoolSize - 线程池核心池的大小。

maximumPoolSize - 线程池的最大线程数。

keepAliveTime - 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。

unit - keepAliveTime 的时间单位。

workQueue - 用来储存等待执行任务的队列。

threadFactory - 线程工厂。

handler - 拒绝策略。

**关注点1 线程池大小**

线程池有两个线程数的设置，一个为核心池线程数，一个为最大线程数。

在创建了线程池后，默认情况下，线程池中并没有任何线程，等到有任务来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法

当创建的线程数等于 corePoolSize 时，会加入设置的阻塞队列。当队列满时，会创建线程执行任务直到线程池中的数量等于maximumPoolSize。

**关注点2 适当的阻塞队列**

java.lang.IllegalStateException: Queue full

方法 抛出异常 返回特殊值 一直阻塞 超时退出

插入方法 add(e) offer(e) put(e) offer(e,time,unit)

移除方法 remove() poll() take() poll(time,unit)

检查方法 element() peek() 不可用 不可用

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

DelayQueue： 一个使用优先级队列实现的无界阻塞队列。

SynchronousQueue： 一个不存储元素的阻塞队列。

LinkedTransferQueue： 一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque： 一个由链表结构组成的双向阻塞队列。

ThreadPoolExecutor.AbortPolicy: 丢弃任务并抛出RejectedExecutionException异常。 (默认)

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

说明：Executors 各个方法的弊端：

(1）newFixedThreadPool 和 newSingleThreadExecutor:

主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至 OOM。

(2）newCachedThreadPool 和 newScheduledThreadPool:

主要问题是线程数最大数是 Integer.MAX\_VALUE，可能会创建数量非常多的线程，甚至 OOM。

**newSingleThreadExecutor**

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。

此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

new ThreadPoolExecutor(1, 1,0L,TimeUnit.MILLISECONDS,new LinkedBlockingQueue());

**newFixedThreadPool**

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。

线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue());

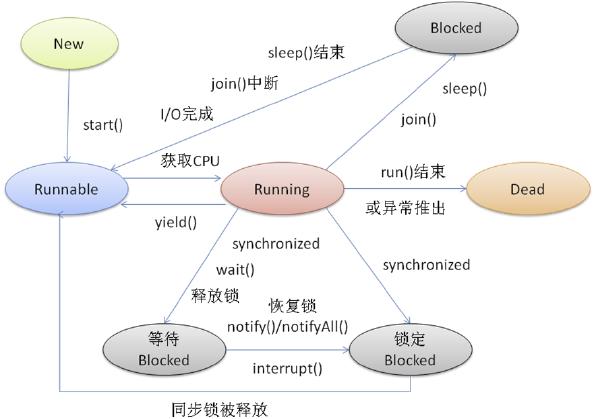
**newCachedThreadPool**

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。

此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit.SECONDS,new SynchronousQueue());



线程在Running的过程中可能会遇到阻塞(Blocked)情况

1. 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
2. 调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）
3. 对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

此外，在runnable状态的线程是处于被调度的线程，此时的调度顺序是不一定的。Thread类中的yield方法可以让一个running状态的线程转入runnable。

基本线程类指的是Thread类，Runnable接口，Callable接口

Thread 类实现了Runnable接口，启动一个线程的方法：

　MyThread my = new MyThread();　　my.start();

**Thread类相关方法：**

//当前线程可转让cpu控制权，让别的就绪状态线程运行（切换）publicstatic Thread.yield() //暂停一段时间publicstatic Thread.sleep() //在一个线程中调用other.join(),将等待other执行完后才继续本线程。　　　　public join()//后两个函数皆可以被打断public interrupte()

**关于中断**：它并不像stop方法那样会中断一个正在运行的线程。线程会不时地检测中断标识位，以判断线程是否应该被中断（中断标识值是否为true）。终端只会影响到wait状态、sleep状态和join状态。被打断的线程会抛出InterruptedException。

Thread.interrupted()检查当前线程是否发生中断，返回boolean

synchronized在获锁的过程中是不能被中断的。

中断是一个状态！interrupt()方法只是将这个状态置为true而已。所以说正常运行的程序不去检测状态，就不会终止，而wait等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加while(!Thread.interrupted()) ，则同样可以在中断后离开代码体

**Thread类最佳实践**：

写的时候最好要设置线程名称 Thread.name，并设置线程组 ThreadGroup，目的是方便管理。在出现问题的时候，打印线程栈 (jstack -pid) 一眼就可以看出是哪个线程出的问题，这个线程是干什么的。

**如何获取线程中的异常**

不能用try,catch来获取线程中的异常

**Runnable**

与Thread类似

**Callable**

future模式：并发模式的一种，可以有两种形式，即无阻塞和阻塞，分别是isDone和get。其中Future对象用来存放该线程的返回值以及状态

ExecutorService e = Executors.newFixedThreadPool(3); //submit方法有多重参数版本，及支持callable也能够支持runnable接口类型.Future future = e.submit(new myCallable());future.isDone() //return true,false 无阻塞future.get() // return 返回值，阻塞直到该线程运行结束