多线程新得感悟：

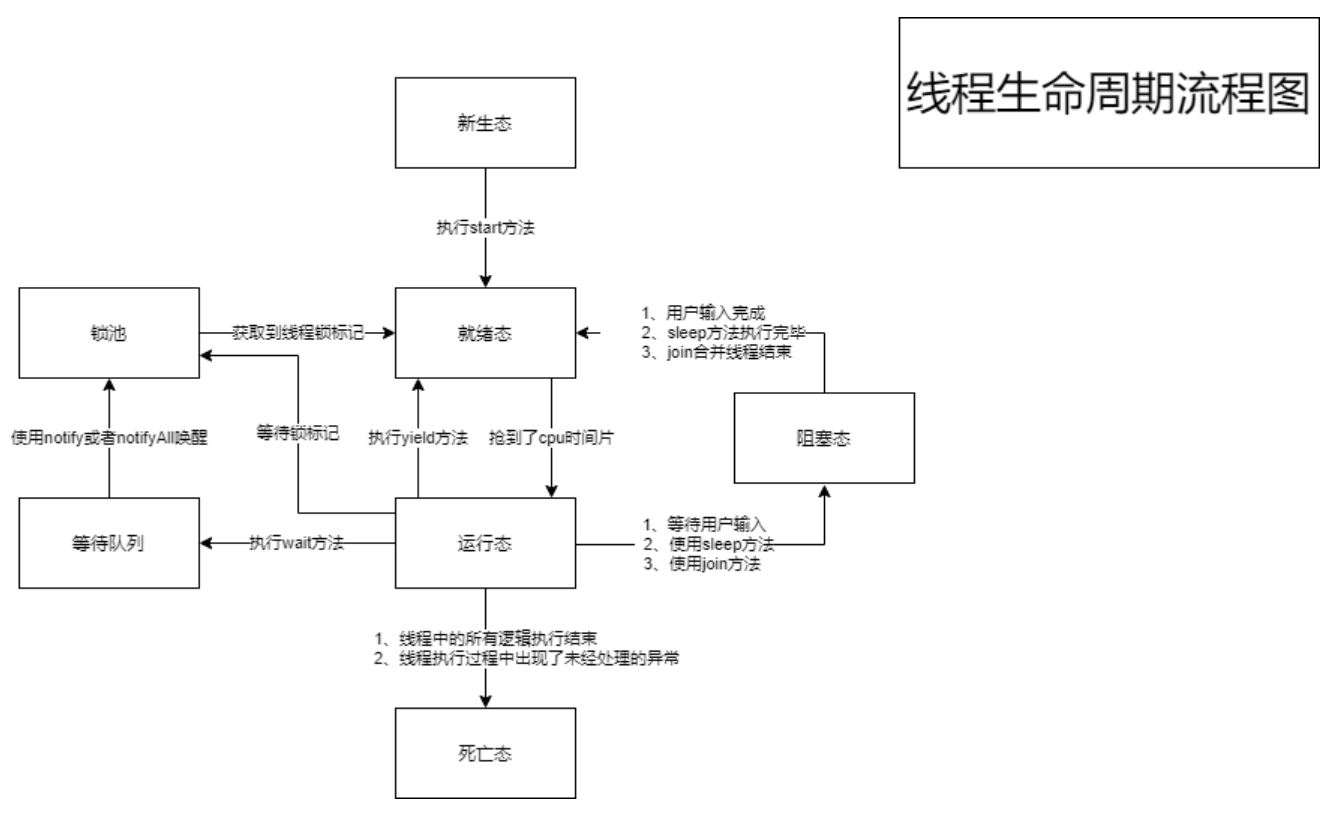
多线程存在的目的是提高CPU利用率，加快接口响应速度

当用户请求后台时，每个请求就是一个线程，当进入方法时，每个线程相当于一个主线程，如果要进行请求比较长的数据，那么可以用线程池去创建线程进行多线程的操作，保证方法响应速度提升。

但是多线程会存在线程安全，数据一致性的问题，所以必要的时候使用线程安全的集合或者加锁synchrnized或者AQS（抽象队列同步器），比如reentrantLock 自旋+park()+队列

在方法里要使用多线程一定要new Thread()，然后注意线程安全，尽量用线程安全的集合 concurrentHashMap等

### 线程生命周期



#### 第一点、java线程新生态的生成

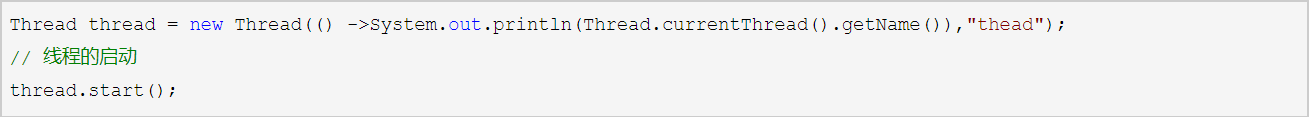




#### 第二点、java由新生态转化为就绪态

实际就是start方法，start方法是开启一个新的线程，如果只调用run方法，还是在主线程执行run方法，不会开启新的线程，线程不会进入到就绪状态，所以start方法才是开启线程的方法，

run方法中存入的只是任务内容。代码如下：



#### 第三点、就绪态到运行态

这一步的话，没有办法使用代码实现，因为呢，当线程执行start方法后，许多线程会去抢占cpu时间片，哪个线程抢占到了cpu时间片，哪个线程就进入到运行状态。

#### 第四点、由运行态回到就绪态

回到就绪态，就需要线程主动放弃抢到的cpu时间片，也就是大家常说的线程的礼让

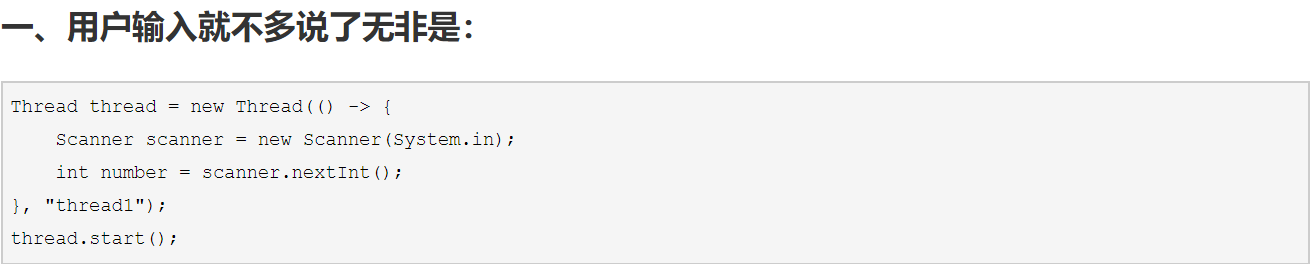
但是需要注意的是，礼让不代表将cpu时间片彻底让给其他线程，还可能在次抢占时又获取到cpu时间片（没办法，有时候都不知道自己折磨厉害）

代码如下：



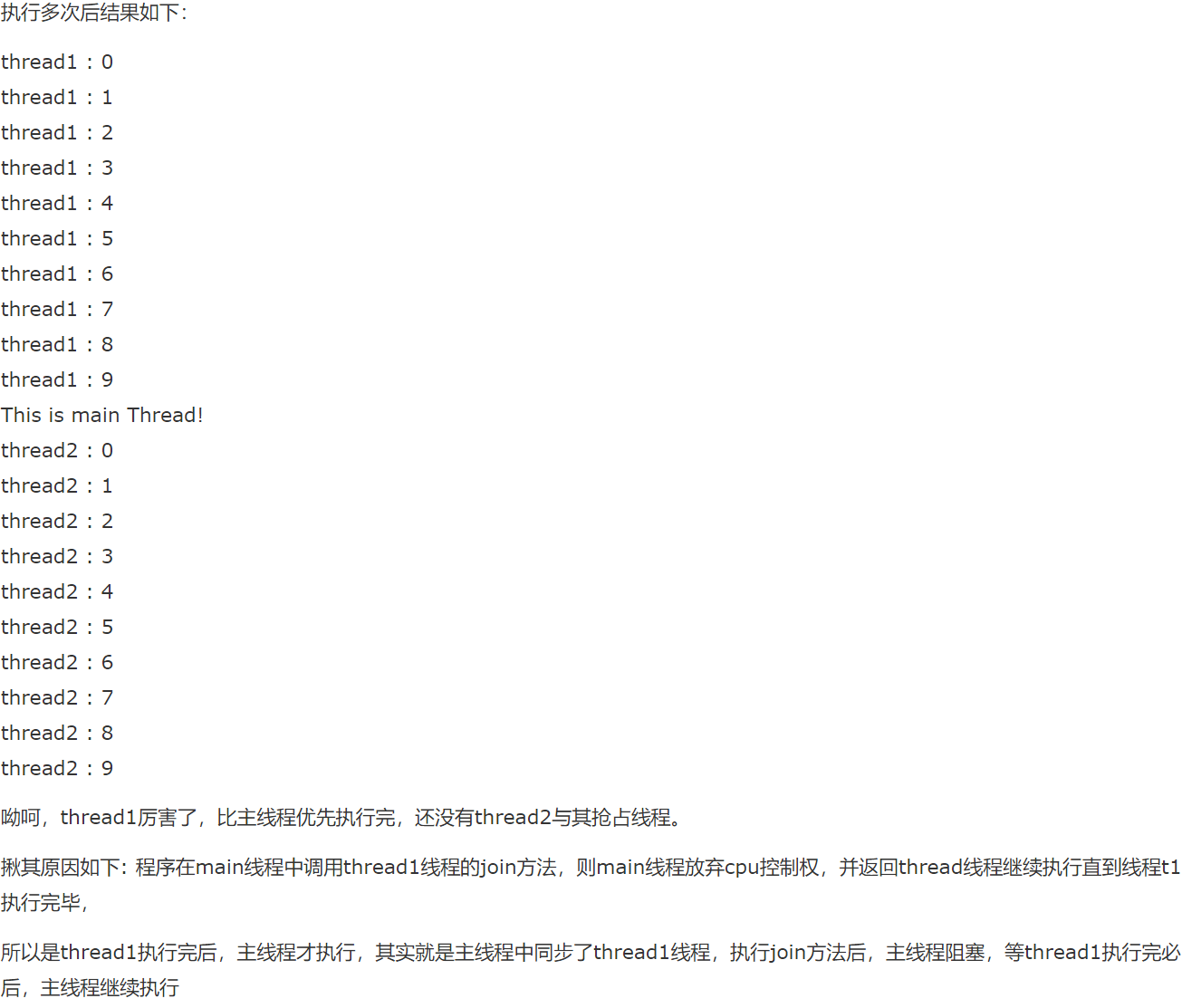
#### 第五点、由运行态到阻塞态

故名思意就是阻塞了，线程暂停了，主要导致线程暂停的方法由：等待用户输入、使用sleep方法、使用jion方法









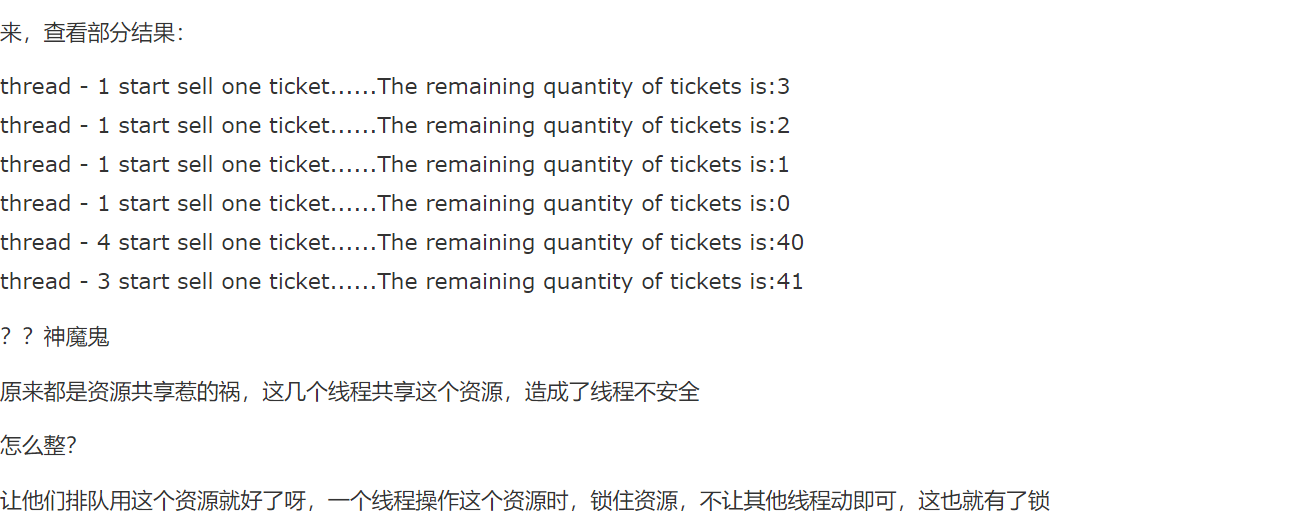
#### 第六点、阻塞态回到就绪态

话不多说了，就是用户输入完必，sleep方法执行完毕、join方法执行完毕

#### 第七点、由运行态到锁池

首先，涉及到的内容为锁，那么什么条件可以用到锁呢，都有哪些锁呢











#### 三、所以由运行态怎样到锁池呢

当某一代码段或者方法被锁住后，其他线程等待锁标记，无法访问就会进入锁池



#### 第八点、使用wait方法进入等待对列，之后通过唤醒进入锁池





#### 第九点、锁池到就绪态

锁池中的线程，拿到锁标记回到就绪态，也就是占有锁标记的线程将锁标记释放

#### 第十点、死亡态

一、线程中的所有逻辑执行结束

二、线程执行过程中出现了未经处理的异常

### 2.Thread的sleep()、join()和wait()

#### 1 sleep

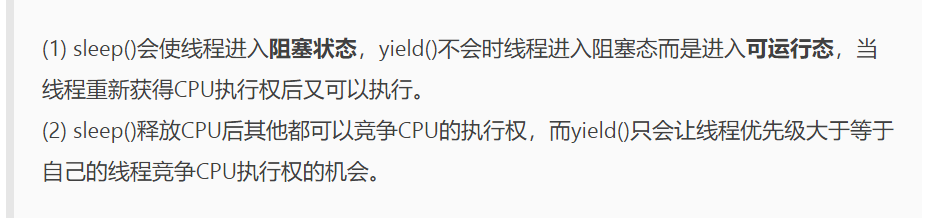
sleep()会让线程交出CPU的执行权，但是不会释放锁。





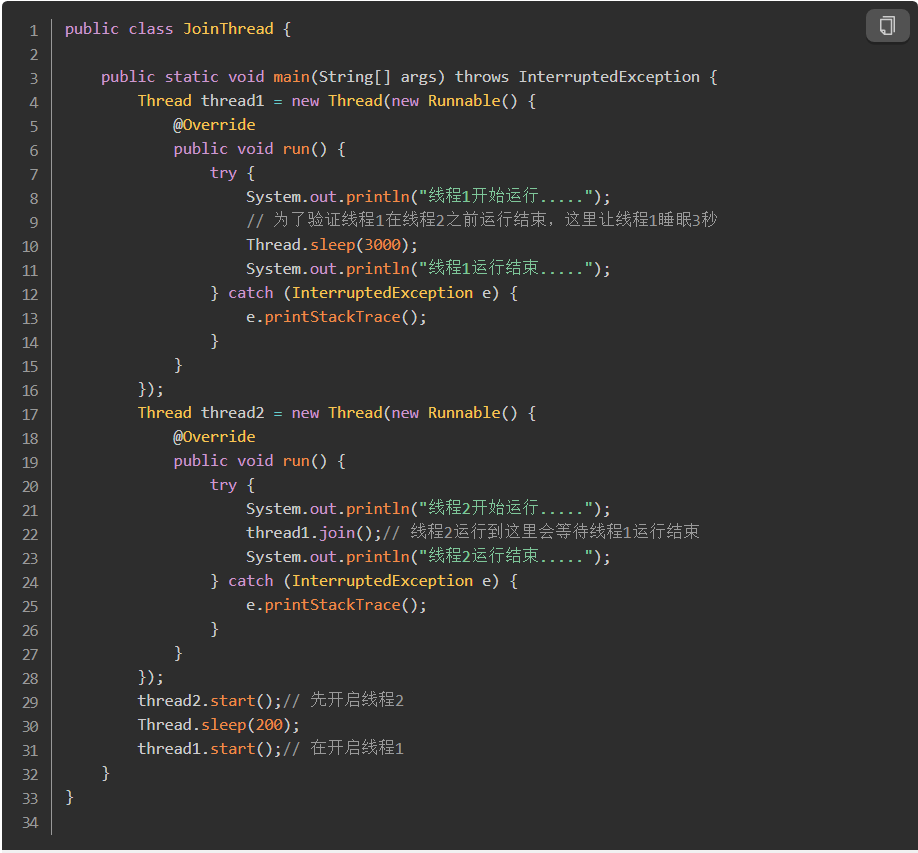
从运行的结果可以看到，线程1先开始执行，在线程1拿到锁后，线程2也开始执行，但是此时是线程1持有锁，所以线程2阻塞，当线程1开始睡眠时，线程2还是没有拿到锁，直到线程1运行结束释放了锁后，线程2才继续运行。  
  所以，**sleep()不释放锁。**

yield()和sleep方法相似，也会交出CPU的执行权，也不会释放锁，两者之间的区别有



#### 2 join

 join()可以保证让一个线程在另一个线程之前执行结束。如何保证一个工作在另一个工作结束之前完成，就可以使用join()方法。





线程2先启动，当运行到thread1.join()时，线程2停止运行，等待线程1执行结束，虽然线程1启动比线程2迟，但是只有当线程1运行结束后，线程2才能继续运行。

#### 3.wait

wait()可以让线程从运行态转换为阻塞态，同时还会释放线程的同步锁。



运行结果：线程2运行结束后，由于线程1没有被唤醒，所以一直阻塞在那里......可以使用带有参数的wait(long mills)，当等待的时间结束后，线程会被自动唤醒。



### 3.CountDownLatch与thread.join()的区别

join，在当前线程中，如果调用某个thread的join方法，那么当前线程就会被阻塞，直到thread线程执行完毕，当前线程才能继续执行。join的原理是，不断的检查thread是否存活，如果存活，那么让当前线程一直wait，直到thread线程终止，线程的this.notifyAll 就会被调用。

我们来看一下这个应用场景：假设现在公司有三个员工A,B,C，他们要开会。但是A需要等B,C准备好之后再才能开始，B,C需要同时准备。我们先用join模拟上面的场景。







可以看到,A总是在B,C准备完成之后才开始执行的。

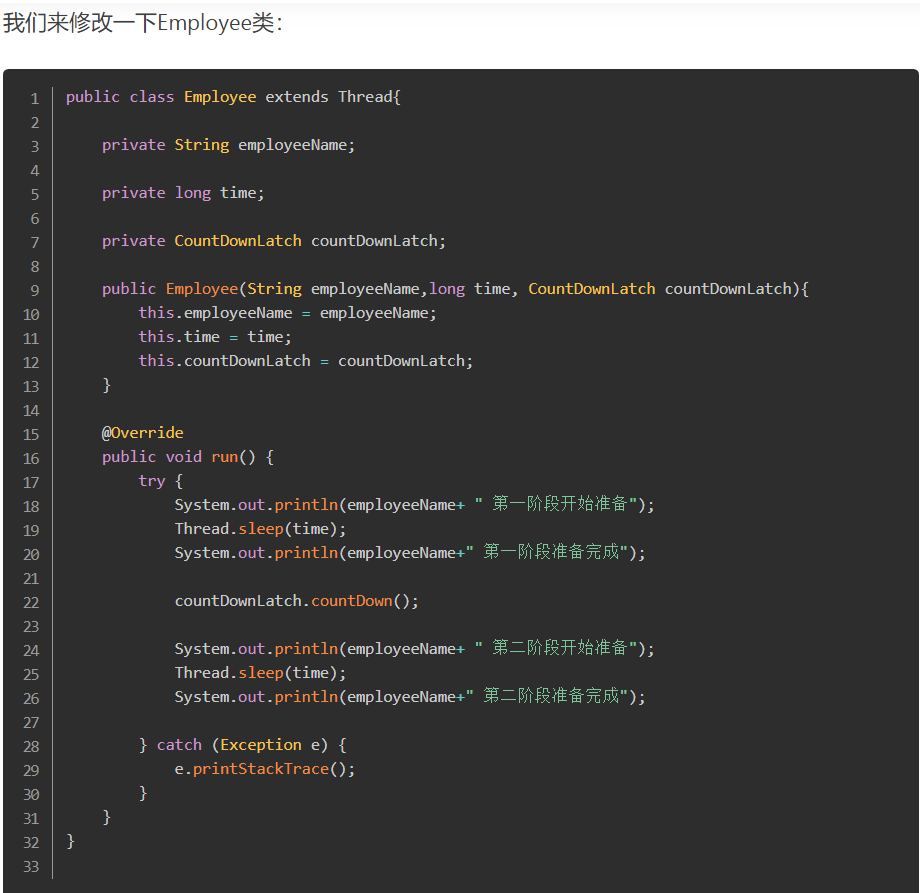
CountDownLatch中我们主要用到两个方法一个是await()方法，调用这个方法的线程会被阻塞，另外一个是countDown()方法，调用这个方法会使计数器减一，当计数器的值为0时，因调用await()方法被阻塞的线程会被唤醒，继续执行。





上面可以看到，CountDownLatch与join都能够模拟上述的场景，那么他们有什么不同呢?这时候我们试想另外一个场景就能看到他们的区别了。

假设A，B，C的工作都分为两个阶段，A只需要等待B，C各自完成他们工作的第一个阶段就可以执行了。





从结果可以看出，A在B，C第一阶段准备完成的时候就开始执行了，不需要等到第二阶段准备完成。这种场景下，用join是没法实现的。

总结：调用join方法需要等待thread执行完毕才能继续向下执行,而CountDownLatch只需要检查计数器的值为零就可以继续向下执行，相比之下，CountDownLatch更加灵活一些，可以实现一些更加复杂的业务场景。

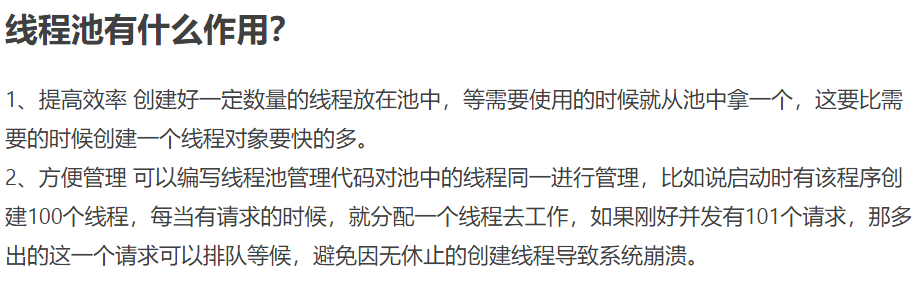
### 4.为什么要使用线程池？

线程是稀缺资源，它的创建与销毁是一个相对偏重且耗资源的一个过程，而java线程依赖于内核进程，创建线程需要进行操作系统状态切换，为避免资源过度消耗需要设法重用线程执行多个任务。线程池就是一个线程缓存，负责对线程进行统一分配、调优与监控。

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）

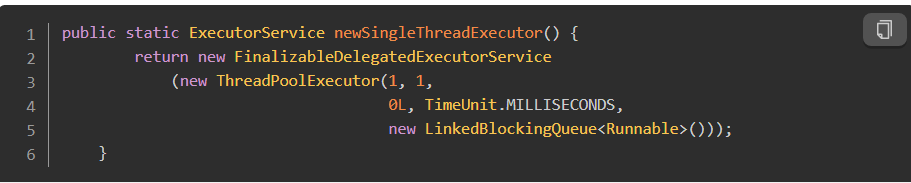
什么时候使用线程池？

1. 单个任务处理时间比较短
2. 需要处理的任务数量很大

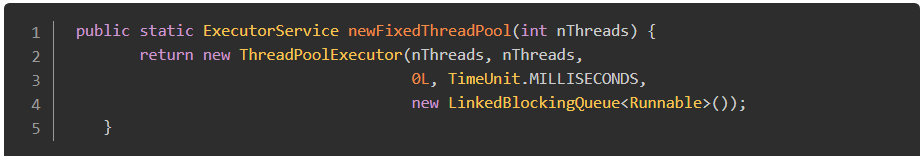


#### 1.说说几种常见的线程池及使用场景

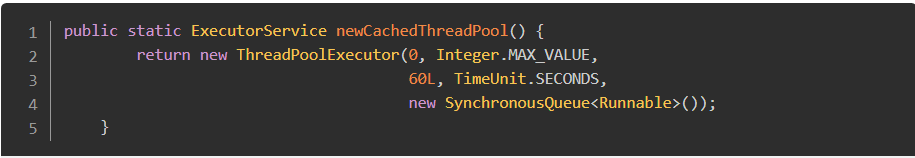
1、newSingleThreadExecutor  
 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。



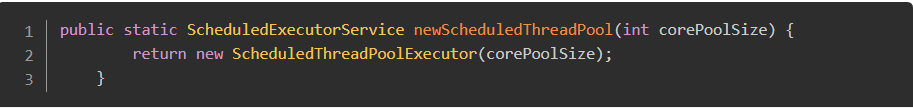
1. newFixedThreadPool  
   创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。



1. newCachedThreadPool  
   创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。



1. newScheduledThreadPool  
   创建一个定长线程池，支持定时及周期性任务执行。



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。 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。

### 5.一问一答

<https://blog.csdn.net/weixin_43848065/article/details/107204950?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-5.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-5.nonecase>

### 什么时候用多线程

****1.高并发：**** 系统接受实现多用户多请求的高并发时，通过多线程来实现。

****2.线程后台处理大任务：**** 一个程序是线性执行的。如果程序执行到要花大量时间处理的任务时，那主程序就得等待其执行完才能继续执行下面的。那用户就不得不等待它执行完。这时候可以开线程把花大量时间处理的任务放在线程处理，这样线程在后台处理时，主程序也可以继续执行下去，用户就不需要等待。线程执行完后执行回调函数。

****3.大任务：**** 大任务处理起来比较耗时，这时候可以起到多个线程并行加快处理（例如：分片上传）。比如处理一个for循环时要花费大量时间，就可以考虑多线程了

<https://blog.csdn.net/wrs120/article/details/90634863?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase>

### 多线程的使用经验

是否使用多线程是看实际场景，跟架构没多大关系。比如你用ssh做电商网站  
，肯定要考虑多线程问题，如果用ssh做一般的管理应用系统，并发量不是很大，就不需要多线程。 并不是说采用ssh的项目，框架自动就帮你弄好多线程了，那是不可能的。

场景一:

一个业务逻辑有很多次的循环，每次循环之间没有影响，比如验证1万条url路径是否存在，正常情况要循环1万次，逐个去验证每一条URL，这样效率会很低，假设验证一条需要1分钟，总共就需要1万分钟，有点恐怖。这时可以用多线程，将1万条URL分成50等份，开50个线程，没个线程只需验证200条，这样所有的线程执行完是远小于1万分钟的。

场景二:

需要知道一个任务的执行进度，比如我们常看到的****进度条****，实现方式可以是在****任务中加入一个整型属性变量(这样不同方法可以共享)，任务执行一定程度就给变量值加1，另外开一个线程按时间间隔不断去访问这个变量，并反馈给用户****。总之使用多线程就是为了充分利用cpu的资源，提高程序执行效率，当你发现一个业务逻辑执行效率特别低，耗时特别长，就可以考虑使用多线程。  
问题：  
不过CPU执行哪个线程的时间和顺序是不确定的，即使设置了线程的优先级，因此使用多线程的风险也是比较大的，会出现很多预料不到的问题，一定要多熟悉概念，多构造不同的场景去测试才能够掌握!

