[https://mp.weixin.qq.com/s?\_\_biz=MzA5MTkxMDQ4MQ==&mid=2648933151&idx=1&sn=2020066b974b5f4c0823abd419e8adae&chksm=88621b21bf159237bdacfb47bd1a344f7123aabc25e3607e78d936dd554412edce5dd825003d&token=995072421&lang=zh\_CN&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933151&idx=1&sn=2020066b974b5f4c0823abd419e8adae&chksm=88621b21bf159237bdacfb47bd1a344f7123aabc25e3607e78d936dd554412edce5dd825003d&token=995072421&lang=zh_CN&scene=21" \l "wechat_redirect)

# Java线程池

## 本文主要内容

* 什么是线程池
* 线程池实现原理
* 线程池中常见的各种队列
* 自定义线程创建的工厂
* 常见的饱和策略
* 自定义饱和策略
* 线程池中两种关闭方法有何不同
* 扩展线程池
* 合理地配置线程池
* 线程池中线程数量的配置

## 什么是线程池

2大家用jdbc操作过数据库应该知道，操作数据库需要和数据库建立连接，拿到连接之后才能操作数据库，用完之后销毁。数据库连接的创建和销毁其实是比较耗时的，真正和业务相关的操作耗时是比较短的。每个数据库操作之前都需要创建连接，为了提升系统性能，后来出现了数据库连接池，系统启动的时候，先创建很多连接放在池子里面，使用的时候，直接从连接池中获取一个，使用完毕之后返回到池子里面，继续给其他需要者使用，这其中就省去创建连接的时间，从而提升了系统整体的性能。

线程池和数据库连接池的原理也差不多，创建线程去处理业务，可能创建线程的时间比处理业务的时间还长一些，如果系统能够提前为我们创建好线程，我们需要的时候直接拿来使用，用完之后不是直接将其关闭，而是将其返回到线程中中，给其他需要这使用，这样直接节省了创建和销毁的时间，提升了系统的性能。

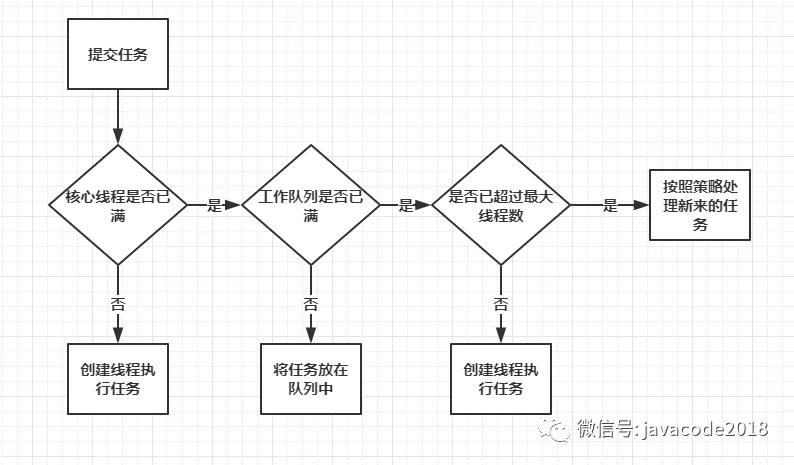
简单的说，在使用了线程池之后，创建线程变成了从线程池中获取一个空闲的线程，然后使用，关闭线程变成了将线程归还到线程池。

## 线程池实现原理

当向线程池提交一个任务之后，线程池的处理流程如下：

1. 判断是否达到核心线程数，若未达到，则直接创建新的线程处理当前传入的任务，否则进入下个流程
2. 线程池中的工作队列是否已满，若未满，则将任务丢入工作队列中先存着等待处理，否则进入下个流程
3. 是否达到最大线程数，若未达到，则创建新的线程处理当前传入的任务，否则交给线程池中的饱和策略进行处理。

流程如下图：



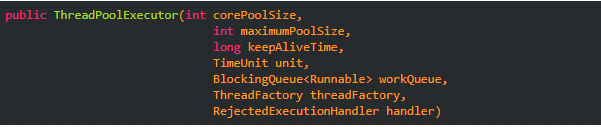
举个例子，加深理解：

咱们作为开发者，上面都有开发主管，主管下面带领几个小弟干活，CTO给主管授权说，你可以招聘5个小弟干活，新来任务，如果小弟还不到五个，立即去招聘一个来干这个新来的任务，当5个小弟都招来了，再来任务之后，将任务记录到一个表格中，表格中最多记录100个，小弟们会主动去表格中获取任务执行，如果5个小弟都在干活，并且表格中也记录满了，那你可以将小弟扩充到20个，如果20个小弟都在干活，并且存放任务的表也满了，产品经理再来任务后，是直接拒绝，还是让产品自己干，这个由你自己决定，小弟们都尽心尽力在干活，任务都被处理完了，突然公司业绩下滑，几个员工没事干，打酱油，为了节约成本，CTO主管把小弟控制到5人，其他15个人直接被干掉了。所以作为小弟们，别让自己闲着，多干活。

原理：先找几个人干活，大家都忙于干活，任务太多可以排期，排期的任务太多了，再招一些人来干活，最后干活的和排期都达到上层领导要求的上限了，那需要采取一些其他策略进行处理了。对于长时间不干活的人，考虑将其开掉，节约资源和成本。

## java中的线程池

jdk中提供了线程池的具体实现，实现类是：**java.util.concurrent.ThreadPoolExecutor**，主要构造方法：



* corePoolSize：核心线程大小，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使有其他空闲线程可以处理任务也会创新线程，等到工作的线程数大于核心线程数时就不会在创建了。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前把核心线程都创造好，并启动；
* maximumPoolSize：线程池允许创建的最大线程数。如果队列满了，并且以创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。如果我们使用了无界队列，那么所有的任务会加入队列，这个参数就没有什么效果了
* keepAliveTime：线程池的工作线程空闲后，保持存活的时间。如果没有任务处理了，有些线程会空闲，空闲的时间超过了这个值，会被回收掉。如果任务很多，并且每个任务的执行时间比较短，避免线程重复创建和回收，可以调大这个时间，提高线程的利用率
* unit：keepAliveTIme的时间单位，可以选择的单位有天、小时、分钟、毫秒、微妙、千分之一毫秒和纳秒。类型是一个枚举java.util.concurrent.TimeUnit，这个枚举也经常使用，有兴趣的可以看一下其源码
* workQueue：工作队列，用于缓存待处理任务的阻塞队列，常见的有4种，本文后面有介绍
* threadFactory：线程池中创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字
* handler：饱和策略，当线程池无法处理新来的任务了，那么需要提供一种策略处理提交的新任务，默认有4种策略，文章后面会提到

### 调用线程池的execute方法处理任务，执行execute方法的过程

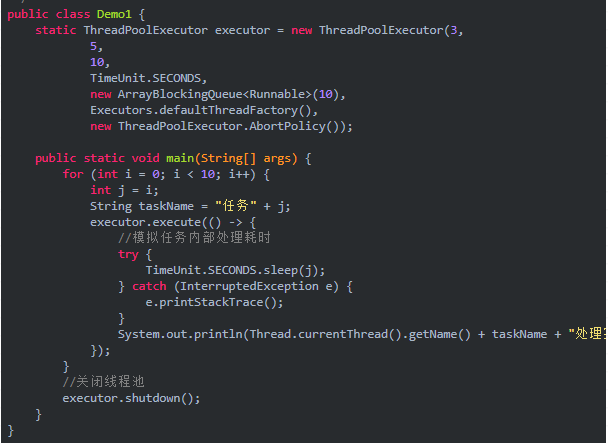
1. 判断线程池中运行的线程数是否小于corepoolsize，是：则创建新的线程来处理任务，否：执行下一步；
2. 试图将任务添加到workQueue指定的队列中，如果无法添加到队列，进入下一步；
3. 判断线程池中运行的线程数是否小于maximumPoolSize，是：则新增线程处理当前传入的任务，否：将任务传递给handler对象rejectedExecution方法处理；

### 线程池的使用步骤

1. 调用构造方法创建线程池
2. 调用线程池的方法处理任务
3. 关闭线程池

## 线程池使用的简单示例

上一个简单的示例，如下：



输出：



## 线程池中常见5种工作队列

任务太多的时候，工作队列用于暂时缓存待处理的任务，jdk中常见的5种阻塞队列：

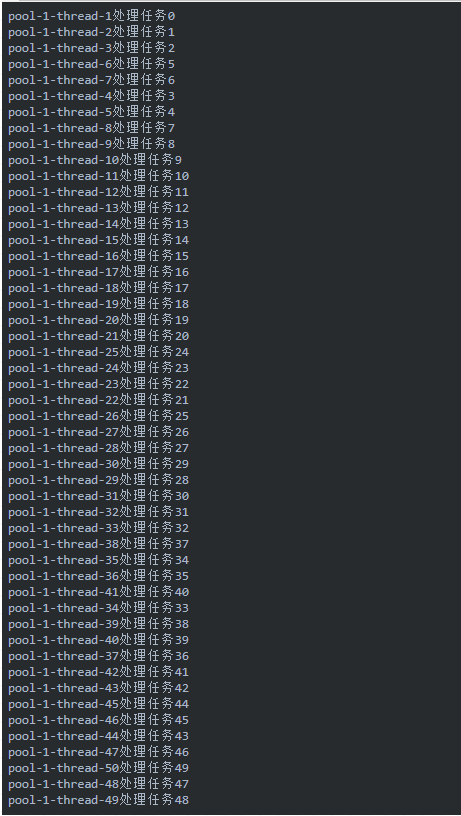
* ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按照先进先出原则对元素进行排序
* LinkedBlockingQueue：是一个基于链表结构的阻塞队列，此队列按照先进先出排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool使用了这个队列。
* SynchronousQueue ：一个不存储元素的阻塞队列，每个插入操作必须等到另外一个线程调用移除操作，否则插入操作一直处理阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用这个队列
* PriorityBlockingQueue：优先级队列，进入队列的元素按照优先级会进行排序

前2种队列相关示例就不说了，主要说一下后面2种队列的使用示例。

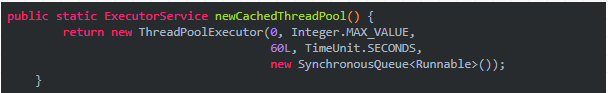
### SynchronousQueue队列的线程池



输出：



代码中使用Executors.newCachedThreadPool()创建线程池，看一下的源码：

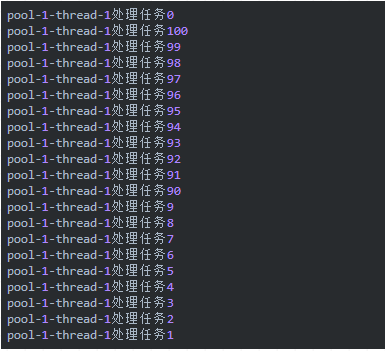


从输出中可以看出，系统创建了50个线程处理任务，代码中使用了SynchronousQueue同步队列，这种队列比较特殊，放入元素必须要有另外一个线程去获取这个元素，否则放入元素会失败或者一直阻塞在那里直到有线程取走，示例中任务处理休眠了指定的时间，导致已创建的工作线程都忙于处理任务，所以新来任务之后，将任务丢入同步队列会失败，丢入队列失败之后，会尝试新建线程处理任务。使用上面的方式创建线程池需要注意，如果需要处理的任务比较耗时，会导致新来的任务都会创建新的线程进行处理，可能会导致创建非常多的线程，最终耗尽系统资源，触发OOM。

### PriorityBlockingQueue优先级队列的线程池



输出：

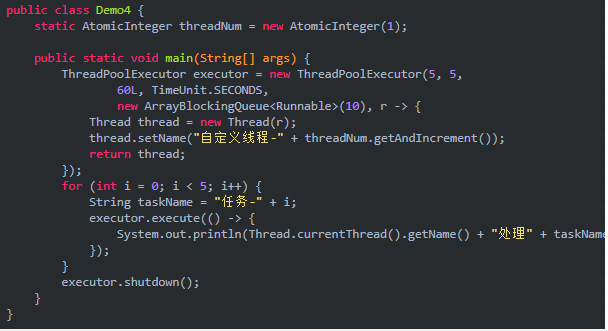


输出中，除了第一个任务，其他任务按照优先级高低按顺序处理。原因在于：创建线程池的时候使用了优先级队列，进入队列中的任务会进行排序，任务的先后顺序由Task中的i变量决定。向PriorityBlockingQueue加入元素的时候，内部会调用代码中Task的compareTo方法决定元素的先后顺序。

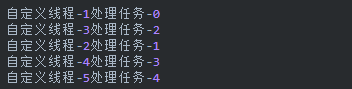
## 自定义创建线程的工厂

给线程池中线程起一个有意义的名字，在系统出现问题的时候，通过线程堆栈信息可以更容易发现系统中问题所在。自定义创建工厂需要实现java.util.concurrent.ThreadFactory接口中的Thread newThread(Runnable r)方法，参数为传入的任务，需要返回一个工作线程。

示例代码：



输出：



代码中在任务中输出了当前线程的名称，可以看到是我们自定义的名称。

通过jstack查看线程的堆栈信息，也可以看到我们自定义的名称，我们可以将代码中executor.shutdown();先给注释掉让程序先不退出，然后通过jstack查看，如下：



## 饱和策略

### 什么是饱和策略

当线程池中队列已满，并且线程池已达到最大线程数，线程池会将任务传递给饱和策略进行处理。这些策略都实现了RejectedExecutionHandler接口。接口中有个方法：

void rejectedExecution(Runnable r, ThreadPoolExecutor executor)

**参数说明：**

r：需要执行的任务

executor：当前线程池对象

### 常见饱和策略

JDK中提供了4种常见的饱和策略:

**AbortPolicy**：直接抛出异常

**CallerRunsPolicy**：在当前调用者的线程中运行任务，即随丢来的任务，由他自己去处理

**DiscardOldestPolicy**：丢弃队列中最老的一个任务，即丢弃队列头部的一个任务，然后执行当前传入的任务

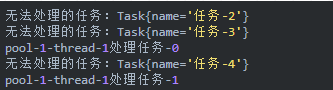
**DiscardPolicy**：不处理，直接丢弃掉，方法内部为空

### 自定义饱和策略

需要实现RejectedExecutionHandler接口。任务无法处理的时候，我们想记录一下日志，我们需要自定义一个饱和策略，示例代码：



输出:



输出结果中可以看到有3个任务进入了饱和策略中，记录了任务的日志，对于无法处理多任务，我们最好能够记录一下，让开发人员能够知道。任务进入了饱和策略，说明线程池的配置可能不是太合理，或者机器的性能有限，需要做一些优化调整。

## 线程池中的2个关闭方法

线程池提供了2个关闭方法：shutdown和shutdownNow，当调用者两个方法之后，线程池会遍历内部的工作线程，然后调用每个工作线程的interrrupt方法给线程发送中断信号，内部如果无法响应中断信号的可能永远无法终止，所以如果内部有无线循环的，最好在循环内部检测一下线程的中断信号，合理的退出。调用这两个方法中任意一个，线程池的isShutdown方法就会返回true，当所有的任务线程都关闭之后，才表示线程池关闭成功，这时调用isTerminaed方法会返回true。

调用shutdown方法之后，线程池将不再接口新任务，内部会将所有已提交的任务处理完毕，处理完毕之后，工作线程自动退出。

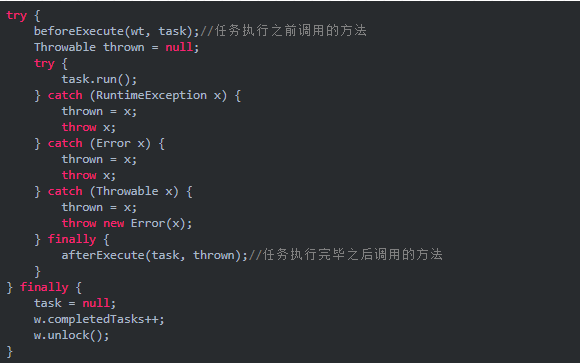
而调用shutdownNow方法后，线程池会将还未处理的（在队里等待处理的任务）任务移除，将正在处理中的处理完毕之后，工作线程自动退出。

至于调用哪个方法来关闭线程，应该由提交到线程池的任务特性决定，多数情况下调用shutdown方法来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow方法。

## 扩展线程池

虽然jdk提供了ThreadPoolExecutor这个高性能线程池，但是如果我们自己想在这个线程池上面做一些扩展，比如，监控每个任务执行的开始时间，结束时间，或者一些其他自定义的功能，我们应该怎么办？

这个jdk已经帮我们想到了，ThreadPoolExecutor内部提供了几个方法beforeExecute、afterExecute、terminated，可以由开发人员自己去这些方法。看一下线程池内部的源码：



**beforeExecute：任务执行之前调用的方法，有2个参数，第1个参数是执行任务的线程，第2个参数是任务**

protected void beforeExecute(Thread t, Runnable r) { }

**afterExecute：任务执行完成之后调用的方法，2个参数，第1个参数表示任务，第2个参数表示任务执行时的异常信息，如果无异常，第二个参数为null**

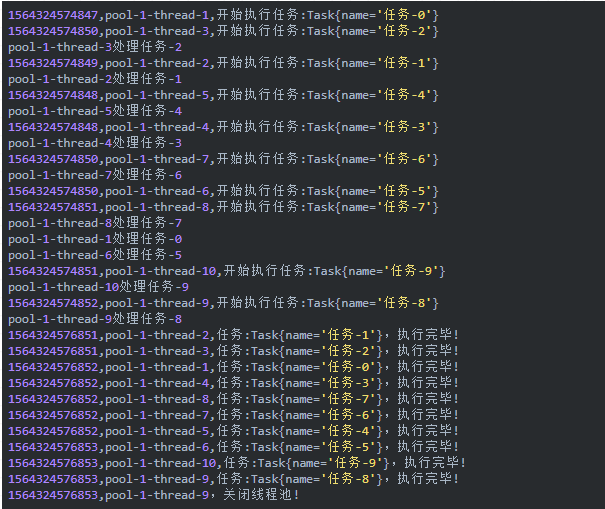
protected void afterExecute(Runnable r, Throwable t) { }

**terminated：线程池最终关闭之后调用的方法。所有的工作线程都退出了，最终线程池会退出，退出时调用该方法**

示例代码：



输出：



从输出结果中可以看到，每个需要执行的任务打印了3行日志，执行前由线程池的beforeExecute打印，执行时会调用任务的run方法，任务执行完毕之后，会调用线程池的afterExecute方法，从每个任务的首尾2条日志中可以看到每个任务耗时2秒左右。线程池最终关闭之后调用了terminated方法。

## 合理地配置线程池

要想合理的配置线程池，需要先分析任务的特性，可以冲一下几个角度分析：

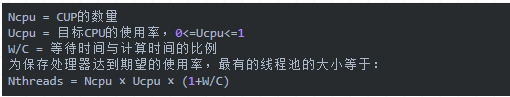
* 任务的性质：CPU密集型任务、IO密集型任务和混合型任务
* 任务的优先级：高、中、低
* 任务的执行时间：长、中、短
* 任务的依赖性：是否依赖其他的系统资源，如数据库连接。

性质不同任务可以用不同规模的线程池分开处理。CPU密集型任务应该尽可能小的线程池，如配置cpu数量+1个线程的线程池。由于IO密集型任务并不是一直在执行任务，不能让cpu闲着，则应配置尽可能多的线程，如：cpu数量\*2。混合型的任务，如果可以拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这2个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。**可以通过Runtime.getRuntime().availableProcessors()方法获取cpu数量**。优先级不同任务可以对线程池采用优先级队列来处理，让优先级高的先执行。

使用队列的时候建议使用有界队列，有界队列增加了系统的稳定性，如果采用无界队列，任务机太多的时候可能导致系统OOM，直接让系统宕。

## 线程池中线程数量的配置

线程池中总线程大小对系统的性能有一定的影响，我们的目标是希望系统能够发挥最好的性能，过多或者过小的线程数量无法有效的使用机器的性能。在Java Concurrency in Practice书中给出了估算线程池大小的公式：



## 一些使用建议

在《阿里巴巴java开发手册》中指出了线程资源必须通过线程池提供，不允许在应用中自行显示的创建线程，这样一方面是线程的创建更加规范，可以合理控制开辟线程的数量；另一方面线程的细节管理交给线程池处理，优化了资源的开销。而线程池不允许使用Executors去创建，而要通过ThreadPoolExecutor方式，这一方面是由于jdk中Executor框架虽然提供了如newFixedThreadPool()、newSingleThreadExecutor()、newCachedThreadPool()等创建线程池的方法，但都有其局限性，不够灵活；另外由于前面几种方法内部也是通过ThreadPoolExecutor方式实现，使用ThreadPoolExecutor有助于大家明确线程池的运行规则，创建符合自己的业务场景需要的线程池，避免资源耗尽的风险。