

发件人: 方堃 fangkun119@icloud.com

主题: CH09 Thread Pool

日期: 2017年5月2日 下午12:01

收件人:

方

CH09 Thread Pool

2017年4月29日 星期六
下午2:11

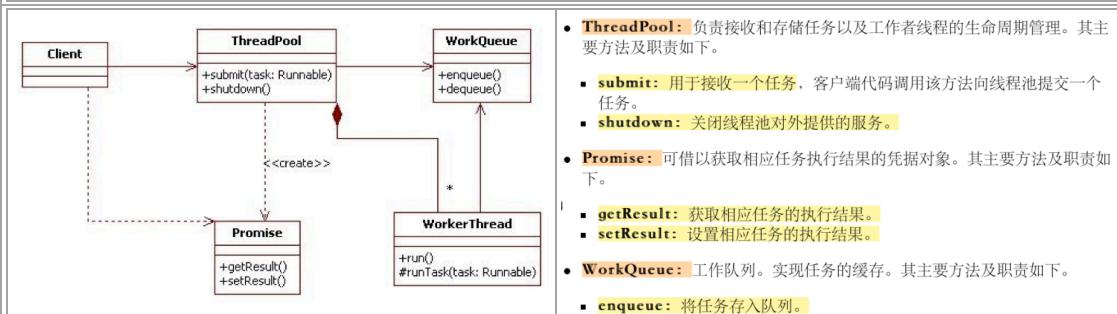
功能 一个系统中的线程相对于其所要处理的任务而言,总是一种非常有限的资源。线程不仅在其执行任务时需要消耗CPU时间和内存等资源,线程对象(Thread实例)本身以及线程所需的调用栈(Call Stack)也占用内存,并且Java中创建一个线程往往意味着JVM会创建相应的依赖于宿主机操作系统的本地线程(Native Thread)。因此,为每个或者每一批任务创建一个线程以对其进行执行,通常是一种奢侈而不现实的事情。比较常见的一种做法是复用一定数量的线程,由这些线程去执行不断产生的任务。绝大多数的Web服务器就是采用这种方法。例如,Tomcat服务器复用一定数量的线程用于处理其接收到的请求。

Thread Pool模式的核心思想是使用队列对待处理的任务进行缓存,并复用一定数量的工作者线程去取队列中的任务进行执行。

Thread Pool模式的本质是使用极其有限的资源去处理相对无限的任务。这好比一个生意兴隆的饭店,虽然每天顾客不断,但饭店却不可能因为来一批客人就增加一个服务员。相反,服务员的人数还是那么多,只不过饭店生意好的时候,服务员们比较忙碌,生意不好时,服务员们比较空闲。

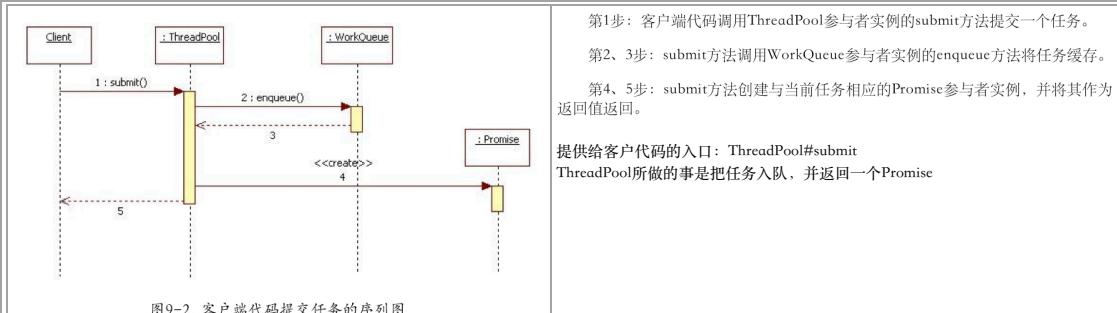
JDK 1.5引入的标准库类java.util.concurrent.ThreadPoolExecutor就是Thread Pool模式的一个实现。读者如果对ThreadPoolExecutor有所了解,也不妨继续阅读本章。毕竟,使用Thread Pool模式不仅仅是会使用ThreadPoolExecutor提供的相关API那么简单,其背后还有不少风险和问题需要注意。

类图

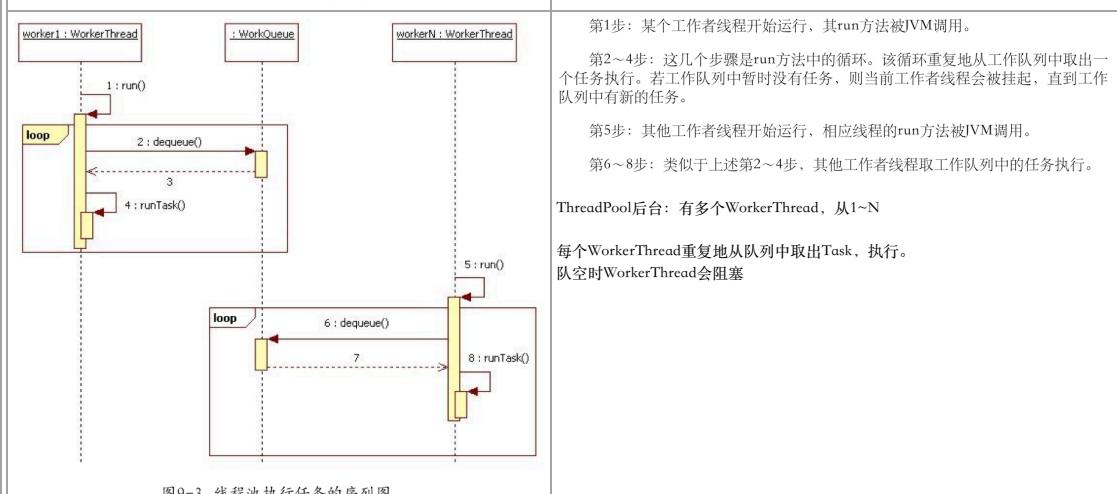


- ThreadPool:** 负责接收和存储任务以及工作者线程的生命周期管理。其主要方法及职责如下。
 - **submit:** 用于接收一个任务,客户端代码调用该方法向线程池提交一个任务。
 - **shutdown:** 关闭线程池对外提供的服务。
- Promise:** 可借以获取相应任务执行结果的凭据对象。其主要方法及职责如下。
 - **getResult:** 获取相应任务的执行结果。
 - **setResult:** 设置相应任务的执行结果。
- WorkQueue:** 工作队列。实现任务的缓存。其主要方法及职责如下。
 - **enqueue:** 将任务存入队列。
 - **dequeue:** 从队列中取出一个任务。
- WorkerThread:** 负责任务执行的工作者线程。其主要方法及职责如下。
 - **run:** 逐一从工作队列中取出任务执行。
 - **runTask:** 执行指定的任务。

时序



第1步: 客户端代码调用ThreadPool参与者实例的submit方法提交一个任务。
第2、3步: submit方法调用WorkQueue参与者实例的enqueue方法将任务缓存。
第4、5步: submit方法创建与当前任务相应的Promise参与者实例,并将其作为返回值返回。
提供给客户代码的入口: ThreadPool#submit
ThreadPool所做的事是把任务入队,并返回一个Promise



第1步: 某个工作者线程开始运行,其run方法被JVM调用。
第2~4步: 这几个步骤是run方法中的循环。该循环重复地从工作队列中取出一个任务执行。若工作队列中暂时没有任务,则当前工作者线程会被挂起,直到工作队列中有新的任务。
第5步: 其他工作者线程开始运行,相应线程的run方法被JVM调用。
第6~8步: 类似于上述第2~4步,其他工作者线程取工作队列中的任务执行。
ThreadPool后台: 有多个WorkerThread,从1~N
每个WorkerThread重复地从队列中取出Task,执行。
队空时WorkerThread会阻塞

好处

Thread Pool模式通过复用一定数量的工作者线程去执行不断被提交的任务,节约了线程这种有限而昂贵的资源。Thread Pool模式还可以带来以下好处。

- 抵消线程创建的开销,提高响应性。**创建线程的消耗不仅包括线程对象本身以及其调用栈所需的内存空间,还包括创建依赖于JVM宿主机操作系统的本地线程。因此,线程创建是一个昂贵的操作。Thread Pool模式通过事先创建一部分线程使得被提交的任务可以立即被执行而无须等待工作者线程的创建,从而提高了响应性。另外,由于一个工作者线程可以为多个任务服务,因此创建工作者线程的开销被平摊到其执行的所有任务中。一个工作者线程执行的任务越多,那么其创建的开销就越明显地被抵消。
- 封装了工作者线程生命周期管理。**线程池本身负责了其工作者线程的生命周期的管理,包括何时创建工作线程、创建多少工作者线程以及何时销毁工作者线程。这使得Thread Pool模式的客户端代码往往只需要关心任务的提交及其处理结果,而无须关心工作者线程的生命周期。如果我们不再需要某个线程池,那么只需要客户端代码调用线程池实例的shutdown方法即可。
- 减少销毁线程的开销。**JVM销毁一个已停止的线程也有其时间上的开销。Thread Pool模式使我们避免了频繁地创建工作线程,因此避免了频繁地销毁已停止的线程。

队列类型选择	无界队列	以无界队列（如 <code>LinkedBlockingQueue</code> ）作为工作队列，虽然工作队列本身并不限制线程池中等待运行的任务的数量，但工作队列中实际可容纳的任务数取决于任务本身对资源的使用情况。例如，线程池的客户端代码在创建向线程池提交的任务对象（ <code>Runnable</code> ）的时候同时创建了该任务对象所需引用的其他对象，而这些被引用的对象占用的内存空间比较大。这样以来，随着工作队列中这样的任务对象越来越多，这些任务对象所导致的内存占用也越来越多（这些任务还没有被执行，因此其占用的内存空间不能被垃圾回收）。极端的情况下，这种情形还可能导致VM内存溢出，从而影响了这个Java应用程序，而不仅仅是使用该线程池的对象。因此，无界工作队列可能导致系统的不稳定，适合在任务占用的内存空间以及其他稀缺资源比较少的情况下使用。										
	Synchronous	如果应用程序确实需要比较大的工作队列容量，而又想避免无界工作队列可能导致的问题，不妨考虑 <code>SynchronousQueue</code> 。 <code>SynchronousQueue</code> 实现上并不使用缓存空间。由于 <code>ThreadPoolExecutor</code> 内部实现任务提交的时候调用的是工作队列（ <code>BlockingQueue</code> 接口的实现类）的非阻塞式入队列方法（ <code>offer</code> 方法），因此，在使用 <code>SynchronousQueue</code> 作为工作队列的前提下，客户端代码向线程池提交任务时，而线程池中又没有空闲的线程能够从 <code>SynchronousQueue</code> 队列实例中取一个任务，那么相应的 <code>offer</code> 方法调用就会失败（即任务没有被存入工作队列）。此时， <code>ThreadPoolExecutor</code> 会新建一个新的工作者线程用于对这个入队列失败的任务进行处理（假设此时线程池的大小还未达到其最大线程池大小）。所以，使用 <code>SynchronousQueue</code> 作为工作队列，工作队列本身并不限制待执行的任务的数量。但此时需要限定线程池的最大大小为一个合理的有限值，而不是 <code>Integer.MAX_VALUE</code> ，否则可能导致线程池中的工作者线程的数量一直增加到系统资源所无法承受为止。本案例就是采用了这种方法来配置其所用的线程池。										
	有界队列	以有界队列（如 <code>ArrayBlockingQueue</code> 、有界的 <code>LinkedBlockingQueue</code> ）作为工作队列则可以限定线程池中待执行任务的数量，这在一定程度上可以限制资源的消耗。通常，使用有界队列作为工作队列需要指定线程池的最大大小为一个合理有限值，而不是 <code>Integer.MAX_VALUE</code> 。其理由类似上面使用 <code>SynchronousQueue</code> 作为工作队列的情形。而有界工作队列加上有限数量的工作者线程则可能导致死锁。有关线程池的死锁问题，下文会提到。有界队列适合在提交给线程池执行的各个任务之间是相互独立（而非有依赖关系）的情况下使用。										
线程池大小	问题	线程池大小指线程池中的工作者线程的数量。线程池大小太大会消耗过多的资源，并增加上下文切换。线程池大小太小，又可能导致无法充分利用CPU资源，使任务处理的吞吐率过低。因此，实战中使用线程池时需要寻找一个合理（或者所谓最佳）的线程池大小。合理的线程池大小取决于该线程池所要处理的任务的特性、系统资源状况以及任务所使用的稀缺资源状况等因素。										
	系统资源考虑	系统资源状况主要考虑系统CPU个数以及JVM堆内存的大小。通常，线程池的大小不是硬编码（Hard-coded）在代码中，而是可配置的（如通过配置文件配置）或者动态计算出来的。动态计算出来的线程池大小通常是基于CPU个数计算的。在Java中我们可以调用 <code>java.lang.Runtime</code> 类的 <code>availableProcessors</code> 方法获取JVM宿主机CPU个数。下面为了讨论方便，我们用 N_{cpu} 表示系统的CPU个数。										
	任务特性考虑	任务的特性主要考虑任务是CPU密集型、I/O密集型，还是混合型（同时包含较多计算和I/O操作）。对于CPU密集型任务，相应的线程池的大小可以考虑设置为 $N_{cpu}+1$ 。这里之所以线程池的大小比CPU的个数还多1个，是因为考虑到即便是CPU密集型的任务其执行线程也可能在某一时刻由于某种原因，如缺页中断（Page Fault）而出现等待。此时，一个额外的线程可以继续使用CPU时间片。对于I/O密集型任务，相应的线程池大小可以考虑设置得相对大一点。这是因为I/O密集型任务执行过程中等待I/O的时间相对于其使用CPU的时间长，而处于I/O等待状态的线程并不会消耗CPU资源，因此相应的线程池的大小应大于 N_{cpu} 的数字，比如 $2 \times N_{cpu}$ 。另外，对于I/O密集型任务我们需要注意I/O操作会引起上下文切换 ⁴¹ 。这就意味着进行I/O操作的线程越多由I/O操作引起的上下文切换也越多。因此，对于I/O密集型任务不妨将相应的线程池的核心线程池大小（Core Pool Size）设置为1，并将其最大线程池大小（Maximum Pool Size）设置为 $2 \times N_{cpu}$ 这样。如果线程池只需要一个工作者线程就可以轻松处理提交给它的任务，那么此时由I/O操作导致的上下文切换是最少的；如果一个工作者线程无法满足任务处理的需要，那么 <code>ThreadPoolExecutor</code> 会逐渐增加工作者线程的数量，直到其达到 $2 \times N_{cpu}$ 。清单9-2展示了这样一个设置I/O密集型任务相应的线程池大小的例子。对于混合型任务，可以将任务进行相应的分解，将其分解为CPU密集型和I/O密集型两种任务，这些子任务采用各自的线程池执行。										
	尽管所谓合理或者最佳的线程池大小本身也是不精确的，但是，上述方法还是有些理性的。不过其好处是容易实施。实际上，商用软件往往会规定某个软件在其运行过程中对CPU的使用率不能超过某个限定值（如75%）。因此，如果要进一步“精确”地设置线程池的大小，我们可能需要考虑目标CPU使用率，即我们期望软件运行过程中会保持多少平均CPU使用率。	另外，如果任务本身是混合型而又不太好将其拆分成CPU密集型和I/O密集型的子任务的话，我们也可以考虑不拆分，只是在设置线程池大小的时候要将该任务涉及的等待时间，如等待外部服务器的HTTP响应，以及该任务真正执行计算（使用了CPU时间片）的时间，考虑进来。										
	代码参考	下面给出一个计算线程合理大小的公式，如下所示。										
		$S = N_{cpu} \times U_{cpu} \times (1 + \frac{WT}{ST})$										
线程泄漏	产生原因	线程泄漏（Thread Leak）指线程池中的工作者线程意外中止，使得线程池中实际可用的工作者线程变少。如果线程泄漏持续存在，那么线程池中的工作者线程会越来越少，最终使得线程池无法处理提交给它的任务。线程泄漏通常是由线程对象的 <code>run</code> 方法中异常处理没有捕获 <code>RuntimeException</code> 和 <code>Error</code> 导致 <code>run</code> 方法意外返回，使得相应线程提前中止。尽管Java中的 <code>ThreadPoolExecutor</code> 已经对线程泄漏进行了预防，但是，实战中我们需要注意另外一种可能会事实上造成线程泄漏的场景：如果线程池中的某个工作者线程执行的任务涉及外部资源等待，如等待网络I/O，而该任务又没有对这种等待指定时间限制。那么，外部资源如果一直没有返回该任务所等待的结果，就会导致执行该任务的工作者线程一直处于等待状态而无法执行其他任务，这就形成了事实上的线程泄漏。										
	问题	如果我们在创建 <code>ThreadPoolExecutor</code> 实例的时候指定了有界队列作为工作队列，那么当线程池中的工作队列满，并且工作者线程数量已达到最大工作者线程数（线程池的最大大小）时，线程池就处于饱和状态。此时，新提交给线程池的任务就会被拒绝（无法提交成功）。但是，从线程池的客户端代码的角度来看，其为了提高计算的可靠性，必需考虑如何应对这种任务提交被拒绝的情形，即线程池饱和处理策略。 <code>ThreadPoolExecutor</code> 已经提供了线程池饱和处理策略的接口和一些预定义的实现类。										
	接口	接口 <code>java.util.concurrent.RejectedExecutionHandler</code> 对线程池饱和处理策略进行了抽象，其声明如清单9-2所示。当一个提交给 <code>ThreadPoolExecutor</code> 实例的任务被拒绝时，相应的 <code>ThreadPoolExecutor</code> 实例会调用其 <code>RejectedExecutionHandler</code> 实例的 <code>rejectedExecution</code> 方法以执行线程池饱和处理策略。JDK提供了该接口的几个实现类，如清单9-3所示。										
JDK预实现	代码	清单9-3. <code>RejectedExecutionHandler</code> 接口声明										
		<pre>public interface RejectedExecutionHandler {</pre>										
		<pre> /** * 提交失败的任务 * @param r 试图执行任务的线程池 * @throws RejectedExecutionException */ void rejectedExecution(Runnable r, ThreadPoolExecutor executor);</pre>										
	从表9-2可以看出，JDK中预置的线程池饱和处理策略中只有 <code>ThreadPoolExecutor.CallerRunsPolicy</code> 能够实现补救，即提交失败的任务可以获得再次（或者更多）执行的机会。其他的饱和处理策略都是放弃提交失败的任务，而 <code>ThreadPoolExecutor.AbortPolicy</code> 则是 <code>ThreadPoolExecutor</code> 默认的线程池饱和处理策略。如果使用 <code>ThreadPoolExecutor.CallerRunsPolicy</code> 作为线程池饱和处理策略，需要注意它可能会引起线程安全问题。假设某些任务其执行过程中使用了非线程安全对象，并且不需要与其他线程共享任何对象，此时我们完全可以考虑使用最大工作者线程数为1的 <code>ThreadPoolExecutor</code> 实例来执行这些任务。此时，这些任务的相关代码可以采用单线程的方式去编写，而无须考虑数据同步和线程安全。这种情况下，如果我们采用 <code>ThreadPoolExecutor.CallerRunsPolicy</code> 作为该 <code>ThreadPoolExecutor</code> 实例的线程饱和处理策略则可能引起线程安全问题。这是因为情形下提交失败的任务会通过 <code>ThreadPoolExecutor.CallerRunsPolicy</code> 实例被客户端线程重新执行，而客户端线程与该 <code>ThreadPoolExecutor</code> 实例中的唯一的工作者线程可能形成两个并发的线程，从而引发线程安全。	1										
		<table border="1"><thead><tr><th>实现类</th><th>所实现的处理策略</th></tr></thead><tbody><tr><td><code>ThreadPoolExecutor.AbortPolicy</code></td><td>直接抛出异常</td></tr><tr><td><code>ThreadPoolExecutor.DiscardPolicy</code></td><td>丢弃当前被拒绝的任务（而不抛出任何异常）</td></tr><tr><td><code>ThreadPoolExecutor.DiscardOldestPolicy</code></td><td>将缓冲区中最老的任务丢弃，然后重新尝试接纳被拒绝的任务</td></tr><tr><td><code>ThreadPoolExecutor.CallerRunsPolicy</code></td><td>在客户端线程中执行被拒绝的任务</td></tr></tbody></table>	实现类	所实现的处理策略	<code>ThreadPoolExecutor.AbortPolicy</code>	直接抛出异常	<code>ThreadPoolExecutor.DiscardPolicy</code>	丢弃当前被拒绝的任务（而不抛出任何异常）	<code>ThreadPoolExecutor.DiscardOldestPolicy</code>	将缓冲区中最老的任务丢弃，然后重新尝试接纳被拒绝的任务	<code>ThreadPoolExecutor.CallerRunsPolicy</code>	在客户端线程中执行被拒绝的任务
实现类	所实现的处理策略											
<code>ThreadPoolExecutor.AbortPolicy</code>	直接抛出异常											
<code>ThreadPoolExecutor.DiscardPolicy</code>	丢弃当前被拒绝的任务（而不抛出任何异常）											
<code>ThreadPoolExecutor.DiscardOldestPolicy</code>	将缓冲区中最老的任务丢弃，然后重新尝试接纳被拒绝的任务											
<code>ThreadPoolExecutor.CallerRunsPolicy</code>	在客户端线程中执行被拒绝的任务											
自定义实现	线程池自动重试的处理策略。清单9-4展示了一个自定义的线程池饱和处理策略，它支持将提交失败的任务重新放入线程池的工作队列等待处理。当然，由于线程池饱和时其工作队列是满的，对工作队列的阻塞式入队列方法（ <code>put</code> 方法）的调用需要等到相应队列非满时才能返回，因此该处理策略可能导致线程池的客户端线程因等待线程池工作队列非空而阻塞。如果提交给线程池的任务执行过程中又会提交新的任务给同一个线程池，而这两个任务之间又有依赖关系，则该处理策略可能导致线程池死锁。	1										
	代码	<pre>public class ReEnqueueRejectedExecutionHandler implements RejectedExecutionHandler {</pre>	客户端线程：已经因为队满而被阻塞									
			线程池后台线程：执行用户自定义的饱和处理策略									

		<pre> @Override public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) { if (executor.isShutdown()) return; try { executor.getQueue().put(r); } catch (InterruptedException e) { ; } } </pre>	该策略要求将Runnable追加到任务队列中而任务队列本身已经满了，因此后台线程也阻塞了 (假定线程池使用了有界阻塞队列)																				
	无界队列	虽然没有队满阻塞问题，但物理上（计算资源）仍然存在上限，仍然存在饱和问题																					
死锁	<p>问题 如果线程池中执行的任务在其执行过程中又会向同一个线程池提交另外一个任务，而前一个任务的执行结束又依赖于后一个任务的执行结果，那么当线程池中所有的线程都处于这种等待其他任务的处理结果，而这些线程所等待的任务仍然还在工作队列中的时候，由于线程池已经没有可以对工作队列中的任务进行处理的工作者线程，这种等待就会一直持续下去而形成死锁（Deadlock）。</p> <p>解决 因此，适合提交给同一线程池实例执行的任务是相互独立的任务，而不是彼此有依赖关系的任务。 要执行彼此有依赖关系的任务可以考虑将不同类型的任务交给不同的线程池实例执行，或者对负责任务执行的线程池实例进行如下配置。</p> <p>配置1：设置线程池的最大大小为一个有限值，而不是默认值Integer.MAX_VALUE。</p> <p>配置2：使用SynchronousQueue作为工作队列。</p> <p>配置3：使用ThreadPoolExecutor.CallerRunsPolicy作为线程池饱和处理策略。</p> <p>原理 采用上述配置的线程池之所以能够避免死锁，是因为：当线程池中的一个工作者线程（ThreadA）执行某个任务（TaskA）时，该任务向同一个线程池提交了另外一个任务（TaskB），而TaskB的执行结束依赖于TaskA的处理结果。若此时线程池已满（这有赖于上述配置1），则TaskB从工作队列会失败（这有赖于上述配置2）。这时，TaskB会由于线程池饱和（工作者队列“满”并且线程池也“满”）而被拒绝。但巧妙的一点是，此时TaskB可以由当前线程池中提交该任务的工作者线程（即ThreadA）自身去执行（这有赖于上述配置3），即存在依赖关系的两个任务TaskA和TaskB此时由线程池中的同一个工作者线程执行，因此避免了死锁，如清单9-5所示。</p>	代码 电子书2152																					
空闲线程	线程池中长期处于空闲状态（即没有在执行任务）的工作者线程会浪费宝贵的线程资源。因此，清理一部分这样的线程可以节约有限的资源。ThreadPoolExecutor支持将其核心工作者线程以外的空闲线程进行清理。创建ThreadPoolExecutor实例时，我们可以在其构造器的第3、4个参数中指定一个空闲持续时间。核心工作者线程以外的工作者线程空闲了指定时间以后，ThreadPoolExecutor就可以将其清理掉。																						
监控	<p>用途 线程池的大小、线程空闲时间限制这些线程池的属性虽然我们可通过配置的方式进行指定（而不是在代码中硬编码），但是所指定的值是否恰当就需要通过监控来判断。例如，如果我们选择有界队列作为工作队列，那么这个队列的容量以多少为宜呢，这需要在软件测试过程中对线程池进行监控来确定。另外，考虑到测试环境和软件实际运行环境总是存在差别的，出于软件运维的考虑我们可能需要对线程池进行监控。ThreadPoolExecutor类提供了对线程池进行监控的相关方法，如表9-1所示。</p> <table border="1"> <thead> <tr> <th>方法</th><th>方 法</th><th>用 途</th></tr> </thead> <tbody> <tr> <td></td><td>getPoolSize()</td><td>获取当前线程池大小</td></tr> <tr> <td></td><td>getQueue()</td><td>返回工作队列实例，通过该实例可获取工作队列的当前大小</td></tr> <tr> <td></td><td>getLargestPoolSize()</td><td>获取工作者线程数曾经达到的最大数，该数值有助于确认线程池的最大大小设置是否合理</td></tr> <tr> <td></td><td>getActiveCount()</td><td>获取线程池中当前正在执行任务的工作者线程数（近似值）</td></tr> <tr> <td></td><td>getTaskCount()</td><td>获取线程池到目前为止所接收到的任务数（近似值）</td></tr> <tr> <td></td><td>getCompletedTaskCount()</td><td>获取线程池到目前为止已经处理完毕的任务数（近似值）</td></tr> </tbody> </table>	方法	方 法	用 途		getPoolSize()	获取当前线程池大小		getQueue()	返回工作队列实例，通过该实例可获取工作队列的当前大小		getLargestPoolSize()	获取工作者线程数曾经达到的最大数，该数值有助于确认线程池的最大大小设置是否合理		getActiveCount()	获取线程池中当前正在执行任务的工作者线程数（近似值）		getTaskCount()	获取线程池到目前为止所接收到的任务数（近似值）		getCompletedTaskCount()	获取线程池到目前为止已经处理完毕的任务数（近似值）	
方法	方 法	用 途																					
	getPoolSize()	获取当前线程池大小																					
	getQueue()	返回工作队列实例，通过该实例可获取工作队列的当前大小																					
	getLargestPoolSize()	获取工作者线程数曾经达到的最大数，该数值有助于确认线程池的最大大小设置是否合理																					
	getActiveCount()	获取线程池中当前正在执行任务的工作者线程数（近似值）																					
	getTaskCount()	获取线程池到目前为止所接收到的任务数（近似值）																					
	getCompletedTaskCount()	获取线程池到目前为止已经处理完毕的任务数（近似值）																					
可复用	Java标准库类java.util.concurrent.ThreadPoolExecutor就是Thread Pool模式的一个可复用的实现。利用ThreadPoolExecutor实现Thread Pool模式，应用代码只需要完成以下几件事情。	<p>1. 【必需】创建一个ThreadPoolExecutor实例。根据应用程序的需要，创建ThreadPoolExecutor实例时指定一个合适的线程池饱和处理策略。</p> <p>2. 【必需】创建Runnable实例用于表示待执行的任务，并调用ThreadPoolExecutor实例的submit方法提交任务。</p> <p>3. 【可选】使用ThreadPoolExecutor实例的submit方法返回值获取相应任务的执行结果。</p> <p>Thread Pool模式在实现上需要考虑比较多的问题与风险，因此我们尽量不要自己去实现该模式，而是要使用JDK中的现成类ThreadPoolExecutor。</p>																					
JDK例	Java Swing中类javax.swing.SwingWorker可用于执行耗时较长的任务。该类使用了Thread Pool模式。SwingWorker内部维护了一个线程池(ThreadPoolExecutor实例)，该线程池包含了若干个工作者线程用于执行提交给SwingWorker的任务。																						
例子	<p>背景 某系统在用户执行一些关键的操作前要求其输入一个验证码。验证码是一串随机数字，由该系统的服务器端代码生成并通过短信发送给用户。</p> <p>服务端代码实现短信发送功能需要调用其他系统提供的Web服务（Web Service）。从设计上看，我们希望一个名为SmsVerificationCodeSender的类负责验证码的生成和相应短信的下发。这样，系统在发送验证码短信时只需要调用SmsVerificationCodeSender实例的相应方法即可。考虑到SmsVerificationCodeSender的客户端代码（即该系统需要发送验证码短信的代码）是运行在服务器的请求处理线程中的，我们不希望发送验证码短信时所涉及的网络I/O这种相对慢的操作影响服务器请求处理线程执行其他操作（如继续处理其他请求）。因此，SmsVerificationCodeSender需要采用专门的工作者线程负责验证码短信的发送。另外，考虑到系统的并发量，每次发送验证码短信都启动一个专门的工作者线程显然是过于昂贵的。</p> <p>这里，Thread Pool模式就可以派上用场。我们可以创建一个包含若干个工作者线程的线程池，系统需要下发验证码短信时就创建一个相应的任务，并将其提交给这个线程池执行。由于向线程池提交任务这个操作可即刻返回，因此验证码短信发送的快慢并不会影响需要发送验证码的线程继续处理。代码如清单9-1所示。</p>	<pre> public class SmsVerificationCodeSender { private static final ExecutorService EXECUTOR = new ThreadPoolExecutor(1, Runtime.getRuntime().availableProcessors(), 60, TimeUnit.SECONDS, new SynchronousQueue<Runnable>(), new ThreadFactory() { @Override public Thread newThread(Runnable r) { Thread t = new Thread(r, "VerfCodeSender"); t.setDaemon(true); return t; } }); } </pre>																					

```
        return t;
    }

}, new ThreadPoolExecutor.DiscardPolicy());

/**
 * 生成并下发验证码短信到指定的手机号码。
 *
 * @param msisdn 短信接收方号码。
 */
public void sendVerificationSms(final String msisdn) {
    Runnable task = new Runnable() {
        @Override
        public void run() {
            //生成强随机数验证码
            int verificationCode = ThreadSpecificSecureRandom.INSTANCE
                .nextInt(999999);
            DecimalFormat df = new DecimalFormat("000000");
            String txtVerCode = df.format(verificationCode);

            //发送验证码短信
            sendSms(msisdn, txtVerCode);
        }
    };
    EXECUTOR.submit(task);
}

private void sendSms(String msisdn, String verificationCode) {
    System.out.println("Sending verification code " + verificationCode + " to "
        + msisdn);
    // 省略其他代码
}
}
```

使用 Microsoft OneNote for Mac 创建。