Java线程池使用说明

**一简介**

线程的使用在java中占有极其重要的地位，在jdk1.4极其之前的jdk版本中，关于线程池的使用是极其简陋的。在jdk1.5之后这一情况有了很大的改观。Jdk1.5之后加入了java.util.concurrent包，这个包中主要介绍java中线程以及线程池的使用。为我们在开发中处理线程的问题提供了非常大的帮助。

**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。

比较重要的几个类：

|  |  |
| --- | --- |
| ExecutorService | 真正的线程池接口。 |
| ScheduledExecutorService | 能和Timer/TimerTask类似，解决那些需要任务重复执行的问题。 |
| ThreadPoolExecutor | ExecutorService的默认实现。 |
| ScheduledThreadPoolExecutor | 继承ThreadPoolExecutor的ScheduledExecutorService接口实现，周期性任务调度的类实现。 |

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

1. 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。

2.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。

3. 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。

4.newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

**实例**

**1：newSingleThreadExecutor**

MyThread.java

|  |
| --- |
| **publicclass**MyThread **extends** Thread {      @Override  **publicvoid** run() {          System.*out*.println(Thread.*currentThread*().getName() + "正在执行。。。");      }  } |

TestSingleThreadExecutor.java

|  |
| --- |
| **publicclass**TestSingleThreadExecutor {  **publicstaticvoid** main(String[] args) {          //创建一个可重用固定线程数的线程池          ExecutorService pool = Executors.*newSingleThreadExecutor*();          //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口          Thread t1 = **new** MyThread();          Thread t2 = **new** MyThread();          Thread t3 = **new** MyThread();          Thread t4 = **new** MyThread();          Thread t5 = **new** MyThread();          //将线程放入池中进行执行          pool.execute(t1);          pool.execute(t2);          pool.execute(t3);          pool.execute(t4);          pool.execute(t5);          //关闭线程池          pool.shutdown();      }  } |

**输出结果**

|  |
| --- |
| pool-1-thread-1正在执行。。。  pool-1-thread-1正在执行。。。  pool-1-thread-1正在执行。。。  pool-1-thread-1正在执行。。。  pool-1-thread-1正在执行。。。 |

**2newFixedThreadPool**

TestFixedThreadPool.Java

|  |
| --- |
| **publicclass** TestFixedThreadPool {  **publicstaticvoid** main(String[] args) {          //创建一个可重用固定线程数的线程池          ExecutorService pool = Executors.*newFixedThreadPool*(2);          //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口          Thread t1 = **new** MyThread();          Thread t2 = **new** MyThread();          Thread t3 = **new** MyThread();          Thread t4 = **new** MyThread();          Thread t5 = **new** MyThread();          //将线程放入池中进行执行          pool.execute(t1);          pool.execute(t2);          pool.execute(t3);          pool.execute(t4);          pool.execute(t5);          //关闭线程池          pool.shutdown();      }  } |

**输出结果**

|  |
| --- |
| pool-1-thread-1正在执行。。。  pool-1-thread-2正在执行。。。  pool-1-thread-1正在执行。。。  pool-1-thread-2正在执行。。。  pool-1-thread-1正在执行。。。 |

3**newCachedThreadPool**

TestCachedThreadPool.java

|  |
| --- |
| **publicclass** TestCachedThreadPool {  **publicstaticvoid** main(String[] args) {          //创建一个可重用固定线程数的线程池          ExecutorService pool = Executors.*newCachedThreadPool*();          //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口          Thread t1 = **new** MyThread();          Thread t2 = **new** MyThread();          Thread t3 = **new** MyThread();          Thread t4 = **new** MyThread();          Thread t5 = **new** MyThread();          //将线程放入池中进行执行          pool.execute(t1);          pool.execute(t2);          pool.execute(t3);          pool.execute(t4);          pool.execute(t5);          //关闭线程池          pool.shutdown();      }  } |

输出结果：

|  |
| --- |
| pool-1-thread-2正在执行。。。  pool-1-thread-4正在执行。。。  pool-1-thread-3正在执行。。。  pool-1-thread-1正在执行。。。  pool-1-thread-5正在执行。。。 |

4**newScheduledThreadPool**

TestScheduledThreadPoolExecutor.java

|  |
| --- |
| **publicclass** TestScheduledThreadPoolExecutor {  **publicstaticvoid** main(String[] args) {          ScheduledThreadPoolExecutor exec = **new** ScheduledThreadPoolExecutor(1);          exec.scheduleAtFixedRate(**new** Runnable() {//每隔一段时间就触发异常                        @Override  **publicvoid** run() {                             //throw new RuntimeException();                             System.*out*.println("================");                        }                    }, 1000, 5000, TimeUnit.*MILLISECONDS*);          exec.scheduleAtFixedRate(**new** Runnable() {//每隔一段时间打印系统时间，证明两者是互不影响的                        @Override  **publicvoid** run() {                             System.*out*.println(System.*nanoTime*());                        }                    }, 1000, 2000, TimeUnit.*MILLISECONDS*);      }  } |

输出结果

|  |
| --- |
| ================  8384644549516  8386643829034  8388643830710  ================  8390643851383  8392643879319  8400643939383 |

**三：ThreadPoolExecutor详解**

ThreadPoolExecutor的完整构造方法的签名是：**ThreadPoolExecutor**(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler) .

**corePoolSize** - 池中所保存的线程数，包括空闲线程。

**maximumPoolSize**-池中允许的最大线程数。

**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 参数的时间单位。

**workQueue** - 执行前用于保持任务的队列。此队列仅保持由 execute方法提交的 Runnable任务。

**threadFactory** - 执行程序创建新线程时使用的工厂。

**handler** - 由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序。

**ThreadPoolExecutor是Executors类的底层实现。**

在JDK帮助文档中，有如此一段话：

“强烈建议程序员使用较为方便的Executors工厂方法Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）、Executors.newFixedThreadPool(int)（固定大小线程池）Executors.newSingleThreadExecutor()（单个后台线程）

它们均为大多数使用场景预定义了设置。”

下面介绍一下几个类的源码：

**ExecutorService  newFixedThreadPool (int nThreads):固定大小线程池。**

可以看到，corePoolSize和maximumPoolSize的大小是一样的（实际上，后面会介绍，如果使用无界queue的话maximumPoolSize参数是没有意义的），keepAliveTime和unit的设值表名什么？-就是该实现不想keep alive！最后的BlockingQueue选择了LinkedBlockingQueue，该queue有一个特点，他是无界的。

|  |
| --- |
| 1.     public static ExecutorService newFixedThreadPool(int nThreads) {  2.             return new ThreadPoolExecutor(nThreads, nThreads,  3.                                           0L, TimeUnit.MILLISECONDS,  4.                                           new LinkedBlockingQueue<Runnable>());  5.         } |

**ExecutorService  newSingleThreadExecutor()：单线程**

|  |
| --- |
| 1.     public static ExecutorService newSingleThreadExecutor() {  2.             return new FinalizableDelegatedExecutorService  3.                 (new ThreadPoolExecutor(1, 1,  4.                                         0L, TimeUnit.MILLISECONDS,  5.                                         new LinkedBlockingQueue<Runnable>()));  6.         } |

**ExecutorService newCachedThreadPool()：无界线程池，可以进行自动线程回收**

这个实现就有意思了。首先是无界的线程池，所以我们可以发现maximumPoolSize为big big。其次BlockingQueue的选择上使用SynchronousQueue。可能对于该BlockingQueue有些陌生，简单说：该QUEUE中，每个插入操作必须等待另一个线程的对应移除操作。

|  |
| --- |
| 1.     public static ExecutorService newCachedThreadPool() {  2.             return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,  3.                                           60L, TimeUnit.SECONDS,  4.                                           new SynchronousQueue<Runnable>());   1. } |

先从[BlockingQueue](http://dongxuan.iteye.com/admin/blogs/901659/)<[Runnable](http://dongxuan.iteye.com/admin/blogs/901659/)> workQueue这个入参开始说起。在JDK中，其实已经说得很清楚了，一共有三种类型的queue。

所有BlockingQueue 都可用于传输和保持提交的任务。可以使用此队列与池大小进行交互：

如果运行的线程少于 corePoolSize，则 Executor始终首选添加新的线程，而不进行排队。（如果当前运行的线程小于corePoolSize，则任务根本不会存放，添加到queue中，而是直接抄家伙（thread）开始运行）

如果运行的线程等于或多于 corePoolSize，则 Executor始终首选将请求加入队列，**而不添加新的线程**。

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。

**queue上的三种类型。**

排队有三种通用策略：

**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。

**BlockingQueue的选择。**

**例子一：使用直接提交策略，也即SynchronousQueue。**

首先SynchronousQueue是无界的，也就是说他存数任务的能力是没有限制的，但是由于该Queue本身的特性，**在某次添加元素后必须等待其他线程取走后才能继续添加**。在这里不是核心线程便是新创建的线程，但是我们试想一样下，下面的场景。

我们使用一下参数构造ThreadPoolExecutor：

1.     new ThreadPoolExecutor(

2.                 2, 3, 30, TimeUnit.SECONDS,

3.                 new  SynchronousQueue<Runnable>(),

4.                 new RecorderThreadFactory("CookieRecorderPool"),

1. new ThreadPoolExecutor.CallerRunsPolicy());

new ThreadPoolExecutor(

  2, 3, 30, TimeUnit.SECONDS,

  new SynchronousQueue<Runnable>(),

  new RecorderThreadFactory("CookieRecorderPool"),

  new ThreadPoolExecutor.CallerRunsPolicy());

 当核心线程已经有2个正在运行.

1. 此时继续来了一个任务（A），根据前面介绍的“如果运行的线程等于或多于 corePoolSize，则Executor始终首选将请求加入队列，**而不添加新的线程**。”,所以A被添加到queue中。
2. 又来了一个任务（B），且核心2个线程还没有忙完，OK，接下来首先尝试1中描述，但是由于使用的SynchronousQueue，所以一定无法加入进去。
3. 此时便满足了上面提到的“如果无法将请求加入队列，则创建新的线程，除非创建此线程超出maximumPoolSize，在这种情况下，任务将被拒绝。”，所以必然会新建一个线程来运行这个任务。
4. 暂时还可以，但是如果这三个任务都还没完成，连续来了两个任务，第一个添加入queue中，后一个呢？queue中无法插入，而线程数达到了maximumPoolSize，所以只好执行异常策略了。

所以在使用SynchronousQueue通常要求maximumPoolSize是无界的，这样就可以避免上述情况发生（如果希望限制就直接使用有界队列）。对于使用SynchronousQueue的作用jdk中写的很清楚：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。

什么意思？如果你的任务A1，A2有内部关联，A1需要先运行，那么先提交A1，再提交A2，当使用SynchronousQueue我们可以保证，A1必定先被执行，在A1么有被执行前，A2不可能添加入queue中。

**例子二：使用无界队列策略，即LinkedBlockingQueue**

这个就拿**newFixedThreadPool**来说，根据前文提到的规则：

如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。那么当任务继续增加，会发生什么呢？

如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。OK，此时任务变加入队列之中了，那什么时候才会添加新线程呢？

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。这里就很有意思了，可能会出现无法加入队列吗？不像SynchronousQueue那样有其自身的特点，对于无界队列来说，总是可以加入的（资源耗尽，当然另当别论）。换句说，永远也不会触发产生新的线程！corePoolSize大小的线程数会一直运行，忙完当前的，就从队列中拿任务开始运行。所以要防止任务疯长，比如任务运行的实行比较长，而添加任务的速度远远超过处理任务的时间，而且还不断增加，不一会儿就爆了。

**例子三：有界队列，使用ArrayBlockingQueue。**

这个是最为复杂的使用，所以JDK不推荐使用也有些道理。与上面的相比，最大的特点便是可以防止资源耗尽的情况发生。

举例来说，请看如下构造方法：

1.     new ThreadPoolExecutor(

2.                 2, 4, 30, TimeUnit.SECONDS,

3.                 new ArrayBlockingQueue<Runnable>(2),

4.                 new RecorderThreadFactory("CookieRecorderPool"),

5.                 new ThreadPoolExecutor.CallerRunsPolicy());

new ThreadPoolExecutor(

    2, 4, 30, TimeUnit.SECONDS,

    new ArrayBlockingQueue<Runnable>(2),

    new RecorderThreadFactory("CookieRecorderPool"),

    new ThreadPoolExecutor.CallerRunsPolicy());

假设，所有的任务都永远无法执行完。

对于首先来的A,B来说直接运行，接下来，如果来了C,D，他们会被放到queue中，如果接下来再来E,F，则增加线程运行E，F。但是如果再来任务，队列无法再接受了，线程数也到达最大的限制了，所以就会使用拒绝策略来处理。

**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。

1.     public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

2.                 if (!e.isShutdown()) {

3.                     r.run();

4.                 }

5.             }

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

           if (!e.isShutdown()) {

               r.run();

           }

       }

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

1.     public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

2.                 throw new RejectedExecutionException();

3.             }

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

1.     public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

2.             }

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

       }

 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）

1.     public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

2.                 if (!e.isShutdown()) {

3.                     e.getQueue().poll();

4.                     e.execute(r);

5.                 }

1. }

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

           if (!e.isShutdown()) {

               e.getQueue().poll();

               e.execute(r);

           }

       }

该策略就稍微复杂一些，在pool没有关闭的前提下首先丢掉缓存在队列中的最早的任务，然后重新尝试运行该任务。这个策略需要适当小心。

设想:如果其他线程都还在运行，那么新来任务踢掉旧任务，缓存在queue中，再来一个任务又会踢掉queue中最老任务。

总结：

keepAliveTime和maximumPoolSize及BlockingQueue的类型均有关系。如果BlockingQueue是无界的，那么永远不会触发maximumPoolSize，自然keepAliveTime也就没有了意义。

反之，如果核心数较小，有界BlockingQueue数值又较小，同时keepAliveTime又设的很小，如果任务频繁，那么系统就会频繁的申请回收线程。

|  |
| --- |
|  |

public static ExecutorService newFixedThreadPool(int nThreads) {

       return new ThreadPoolExecutor(nThreads, nThreads,

                                     0L, TimeUnit.MILLISECONDS,

                                     new LinkedBlockingQueue<Runnable>());

   }