Java线程池架构原理和源码解析(ThreadPoolExecutor)

2016/04/08 | 分类： [基础技术](http://www.importnew.com/cat/basic) | [0 条评论](http://www.importnew.com/19067.html#respond) | 标签： [线程池](http://www.importnew.com/tag/%e7%ba%bf%e7%a8%8b%e6%b1%a0)

分享到：**0**

原文出处： [xieyu\_zy](http://blog.csdn.net/xieyuooo/article/details/8718741)

在前面介绍JUC的文章中，提到了关于线程池**Execotors**的创建介绍，在文章：《[java之JUC系列-外部Tools](http://blog.csdn.net/xieyuooo/article/details/8572543)》中第一部分有详细的说明，请参阅；

文章中其实说明了外部的使用方式，但是没有说内部是如何实现的，为了加深对实现的理解，在使用中可以放心，我们这里将做源码解析以及反馈到原理上，Executors工具可以创建普通的线程池以及schedule调度任务的调度池，其实两者实现上还是有一些区别，但是理解了ThreadPoolExecutor，在看ScheduledThreadPoolExecutor就非常轻松了，后面的文章中也会专门介绍这块，但是需要先看这篇文章。

使用Executors最常用的莫过于是使用：Executors.newFixedThreadPool(int)这个方法，因为它既可以限制数量，而且线程用完后不会一直被cache住；那么就通过它来看看源码，回过头来再看其他构造方法的区别：

在《[java之JUC系列-外部Tools](http://blog.csdn.net/xieyuooo/article/details/8572543)》文章中提到了构造方法，为了和本文对接，再贴下代码。

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

其实你可以自己new一个**ThreadPoolExecutor**，来达到自己的参数可控的程度，例如，可以将LinkedBlockingQueue换成其它的（**如：SynchronousQueue**），只是可读性会降低，这里只是使用了一种[设计模式](http://www.amazon.cn/gp/product/B001130JN8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B001130JN8)。

我们现在来看看**ThreadPoolExecutor**的源码是怎么样的，也许你刚开始看他的源码会很痛苦，因为你不知道作者为什么是这样设计的，所以本文就我看到的思想会给你做一个介绍，此时也许你通过知道了一些作者的思想，你也许就知道应该该如何去操作了。

这里来看下构造方法中对那些属性做了赋值：

**源码段1：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public ThreadPoolExecutor(int corePoolSize,                             int maximumPoolSize,                             long keepAliveTime,                             TimeUnit unit,                             BlockingQueue<Runnable> workQueue,                             ThreadFactory threadFactory,                             RejectedExecutionHandler handler) {       if (corePoolSize < 0 ||           maximumPoolSize <= 0 ||           maximumPoolSize < corePoolSize ||           keepAliveTime < 0)           throw new IllegalArgumentException();       if (workQueue == null || threadFactory == null || handler == null)           throw new NullPointerException();       this.corePoolSize = corePoolSize;       this.maximumPoolSize = maximumPoolSize;       this.workQueue = workQueue;       this.keepAliveTime = unit.toNanos(keepAliveTime);       this.threadFactory = threadFactory;       this.handler = handler;   } |

这里你可以看到最终赋值的过程，可以先大概知道下参数的意思：

**corePoolSize**：核心运行的poolSize，也就是当超过这个范围的时候，就需要将新的Thread放入到等待队列中了；

**maximumPoolSize**：一般你用不到，当大于了这个值就会将Thread由一个丢弃处理机制来处理，但是当你发生：newFixedThreadPool的时候，corePoolSize和maximumPoolSize是一样的，而corePoolSize是先执行的，所以他会先被放入等待队列，而不会执行到下面的丢弃处理中，看了后面的代码你就知道了。

**workQueue**：等待队列，当达到corePoolSize的时候，就向该等待队列放入线程信息（默认为一个**LinkedBlockingQueue**），运行中的队列属性为：workers，为一个HashSet；内部被包装了一层，后面会看到这部分代码。

**keepAliveTime**：默认都是0，当线程没有任务处理后，保持多长时间，cachedPoolSize是默认60s，不推荐使用。

**threadFactory**：是构造Thread的方法，你可以自己去包装和传递，主要实现newThread方法即可；

**handler**：也就是参数**maximumPoolSize**达到后丢弃处理的方法，java提供了5种丢弃处理的方法，当然你也可以自己弄，主要是要实现接口：RejectedExecutionHandler中的方法：

public void **rejectedExecution**(**Runnable**r, **ThreadPoolExecutor** e)

java默认的是使用：AbortPolicy，他的作用是当出现这中情况的时候会抛出一个异常；其余的还包含：

1、**CallerRunsPolicy**：如果发现线程池还在运行，就直接运行这个线程

2、**DiscardOldestPolicy**：在线程池的等待队列中，将头取出一个抛弃，然后将当前线程放进去。

3、**DiscardPolicy**：什么也不做

4、**AbortPolicy**：java默认，抛出一个异常：RejectedExecutionException。

通常你得到线程池后，会调用其中的：**submit**方法或**execute**方法去操作；其实你会发现，submit方法最终会调用execute方法来进行操作，只是他提供了一个Future来托管返回值的处理而已，当你调用需要有返回值的信息时，你用它来处理是比较好的；这个Future会包装对Callable信息，并定义一个Sync对象（），当你发生读取返回值的操作的时候，会通过Sync对象进入锁，直到有返回值的数据通知，具体细节先不要看太多，继续向下：

来看看execute最为核心的方法吧:

**源码段2：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public void execute(Runnable command) {      if (command == null)          throw new NullPointerException();      if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {          if (runState == RUNNING && workQueue.offer(command)) {              if (runState != RUNNING || poolSize == 0)                  ensureQueuedTaskHandled(command);          }          else if (!addIfUnderMaximumPoolSize(command))              reject(command); // is shutdown or saturated      }  } |

这段代码看似简单，其实有点难懂，很多人也是这里没看懂，没事，我一个if一个if说：

首先第一个判定空操作就不用说了，下面判定的poolSize >= corePoolSize成立时候会进入if的区域，当然它不成立也有可能会进入，他会判定addIfUnderCorePoolSize是否返回false，如果返回false就会进去；

我们先来看下addIfUnderCorePoolSize方法的源码是什么：

**源码段3：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | private boolean addIfUnderCorePoolSize(Runnable firstTask) {      Thread t = null;      final ReentrantLock mainLock = this.mainLock;      mainLock.lock();      try {          if (poolSize < corePoolSize && runState == RUNNING)              t = addThread(firstTask);      } finally {          mainLock.unlock();      }      if (t == null)          return false;      t.start();      return true;  } |

可以发现，这段源码是如果发现小雨corePoolSize就会创建一个新的线程，并且调用线程的**start()**方法将线程运行起来:这个**addThread()**方法，我们先不考虑细节，因为我们还要先看到前面是怎么进去的，这里可以发信啊，只有没有创建成功Thread才会返回false，也就是当当前的poolSize > corePoolSize的时候，或线程池已经不是在running状态的时候才会出现；

**注意：**这里在外部判定一次poolSize和corePoolSize只是初步判定，内部是加锁后判定的，以得到更为准确的结果，而外部初步判定如果是大于了，就没有必要进入这段有锁的代码了。

此时我们知道了，当前线程数量大于corePoolSize的时候，就会进入【**代码段2**】的第一个if语句中，回到【**源码段2**】，继续看if语句中的内容:

这里标记为

**源码段4**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if (runState == RUNNING && workQueue.offer(command)) {      if (runState != RUNNING || poolSize == 0)          ensureQueuedTaskHandled(command);  }  else if (!addIfUnderMaximumPoolSize(command))      reject(command); // is shutdown or saturated |

第一个if，也就是当当前状态为running的时候，就会去执行workQueue.offer(command)，这个workQueue其实就是一个BlockingQueue，offer()操作就是在队列的尾部写入一个对象，此时写入的对象为线程的对象而已；所以你可以认为只有线程池在RUNNING状态，才会在队列尾部插入数据，否则就执行else if，其实else if可以看出是要做一个是否大于MaximumPoolSize的判定，如果大于这个值，就会做reject的操作，关于reject的说明，我们在【**源码段1**】的解释中已经非常明确的说明，这里可以简单看下源码，以应征结果：

**源码段5：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | private boolean addIfUnderMaximumPoolSize(Runnable firstTask) {          Thread t = null;          final ReentrantLock mainLock = this.mainLock;          mainLock.lock();          try {              if (poolSize < maximumPoolSize && runState == RUNNING)                  //在corePoolSize = maximumPoolSize下，该代码几乎不可能运行                  t = addThread(firstTask);          } finally {              mainLock.unlock();          }          if (t == null)              return false;          t.start();          return true;  }  void reject(Runnable command) {          handler.rejectedExecution(command, this);      } |

也就是如果线程池满了，而且线程池调用了shutdown后，还在调用execute方法时，就会抛出上面说明的异常：RejectedExecutionException

再回头来看下【**代码段4**】中进入到等待队列后的操作：

**if** (runState != RUNNING || poolSize == 0)

                   ensureQueuedTaskHandled(command);

这段代码是要在线程池运行状态不是RUNNING或poolSize == 0才会调用，他是干啥呢？

他为什么会不等于RUNNING呢？外面那一层不是判定了他== RUNNING了么，其实有时间差就是了，如果是poolSize == 0也会执行这段代码，但是里面的判定条件是如果不是RUNNING，就做reject操作，在第一个线程进去的时候，会将第一个线程直接启动起来；很多人也是看这段代码很绕，因为不断的循环判定类似的判定条件，你主要记住他们之间有时间差，要取最新的就好了。

**此时貌似代码看完了？咦，此时有问题了：**

1、  等待中的线程在后来是如何跑起来的呢？线程池是不是有类似Timer一样的守护进程不断扫描线程队列和等待队列？还是利用某种锁机制，实现类似wait和notify实现的？

2、  线程池的运行队列和等待队列是如何管理的呢？这里还没看出影子呢！

NO，NO，NO！

Java在实现这部分的时候，使用了怪异的手段，神马手段呢，还要再看一部分代码才晓得。

在前面【**源码段3**】中，我们看到了一个方法叫：**addThread()**，也许很少有人会想到关键在这里，其实关键就是在这里：

我们看看**addThread()**方法到底做了什么。

**源码段6：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | private Thread addThread(Runnable firstTask) {          Worker w = new Worker(firstTask);          Thread t = threadFactory.newThread(w);          if (t != null) {              w.thread = t;              workers.add(w);              int nt = ++poolSize;              if (nt > largestPoolSize)                  largestPoolSize = nt;          }          return t;      } |

这里创建了一个Work，其余的操作，就是讲poolSize叠加，然后将将其放入workers的运行队列等操作；

我们主要关心Worker是干什么的，因为这个threadFactory对我们用途不大，只是做了Thread的命名处理；而Worker你会发现它的定义也是一个Runnable，外部开始在代码段中发现了调用哪个这个Worker的start()方法，也就是线程的启动方法，其实也就是调用了Worker的run()方法，那么我们重点要关心run方法是如何处理的

**源码段7：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public void run() {            try {                Runnable task = firstTask;                firstTask = null;                while (task != null || (task = getTask()) != null) {                    runTask(task);                    task = null;                }            } finally {                workerDone(this);            }        } |

FirstTask其实就是开始在创建work的时候，由外部传入的Runnable对象，也就是你自己的Thread，你会发现它如果发现task为空，就会调用getTask()方法再判定，直到两者为空，并且是一个while循环体。

那么看看getTask()方法的实现为：

**源码段8：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | Runnable getTask() {      for (;;) {          try {              int state = runState;              if (state > SHUTDOWN)                  return null;              Runnable r;              if (state == SHUTDOWN)  // Help drain queue                  r = workQueue.poll();              else if (poolSize > corePoolSize || allowCoreThreadTimeOut)                  r = workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS);              else                  r = workQueue.take();              if (r != null)                  return r;              if (workerCanExit()) {                  if (runState >= SHUTDOWN) // Wake up others                      interruptIdleWorkers();                  return null;              }              // Else retry          } catch (InterruptedException ie) {              // On interruption, re-check runState          }      }  } |

你会发现它是从workQueue队列中，也就是等待队列中获取一个元素出来并返回！

回过头来根据代码段6理解下：

当前线程运行完后，在到workQueue中去获取一个task出来，继续运行，这样就保证了线程池中有一定的线程一直在运行；此时若跳出了while循环，只有workQueue队列为空才会出现或出现了类似于shutdown的操作，自然运行队列会减少1，当再有新的线程进来的时候，就又开始向worker里面放数据了，这样以此类推，实现了线程池的功能。

这里可以看下run方法的finally中调用的workerDone方法为：

**源码段9：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | void workerDone(Worker w) {          final ReentrantLock mainLock = this.mainLock;          mainLock.lock();          try {              completedTaskCount += w.completedTasks;              workers.remove(w);              if (--poolSize == 0)                  tryTerminate();          } finally {              mainLock.unlock();          }      } |

注意这里将workers.remove(w)掉，并且调用了—poolSize来做操作。

至于tryTerminate是做了更多关于回收方面的操作。

最后我们还要看一段代码就是在【**源码段6**】中出现的代码调用为：runTask(task);这个方法也是运行的关键。

**源码段10：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | private void runTask(Runnable task) {         final ReentrantLock runLock = this.runLock;         runLock.lock();         try {             if (runState < STOP &&                 Thread.interrupted() &&                 runState >= STOP)                 thread.interrupt();               boolean ran = false;             beforeExecute(thread, task);             try {                 task.run();                 ran = true;                 afterExecute(task, null);                 ++completedTasks;             } catch (RuntimeException ex) {                 if (!ran)                     afterExecute(task, ex);                 throw ex;             }         } finally {             runLock.unlock();         }     } |

你可以看到，这里面的task为传入的task信息，调用的不是start方法，而是run方法，因为run方法直接调用不会启动新的线程，也是因为这样，导致了你无法获取到你自己的线程的状态，因为线程池是直接调用的run方法，而不是start方法来运行。

这里有个**beforeExecute**和**afterExecute**方法，分别代表在执行前和执行后，你可以做一段操作，在这个类中，这两个方法都是【空body】的，因为普通线程池无需做更多的操作。

如果你要实现类似暂停等待通知的或其他的操作，可以自己extends后进行重写构造；

本文没有介绍关于**ScheduledThreadPoolExecutor**调用的细节，下一篇文章会详细说明，因为大部分代码和本文一致，区别在于一些细节，在介绍：**ScheduledThreadPoolExecutor**的时候，会明确的介绍它与**Timer**和**TimerTask**的巨大区别，区别不在于使用，而是在于本身内在的处理细节。