

CSDN

首页

博客

学院

下载

图文课

论坛

APP

问答

商城

VIP会员

活动

招聘

ITeye

GitChat

搜博主文章

Q

写博客

赚零钱

原

Java 线程池 ThreadPoo

lExecutor 源码分析

2016年02月18日 18:51:33

clevergump

阅读数：7239

标签：

java

线程池

并发

更多

个人分类：

Java

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/cleverGump/article/details/50688008>

转载请注明本文出自 clevergump 的博客：<http://blog.csdn.net/clevergump/article/details/50688008>, 谢谢!

线程池能够对线程进行有效的管理, 复用和数量上限的限制, 如果你需要创建多个线程来执行多个异步任务, 那么使用线程池显然要比频繁地 new Thread 的方式要好.

Java 中的线程池是用 **ThreadPoolExecutor** 类来表示的. 我们今天就结合该类的源码来分析一下这个类内部对于线程的创建, 管理以及后台任务的调度等方面原理. 我这里分析的是 Oracle JDK 1.8 的源码.

1. ctl

ThreadPoolExecutor 类中有个非常重要的字段 ctl, ctl 其实可以理解为单词 control 的简写, 翻译过来就是“控制”, 具体来说就是对线程池的运行状态和池子中线程的数量进行控制的一个字段. 我们看下该字段在源码中的定义:

```
1  /**
2   * The main pool control state, ctl, is an atomic integer packing
3   * two conceptual fields
4   *   workerCount, indicating the effective number of threads
5   *   runState,   indicating whether running, shutting down etc
6   *
7   * In order to pack them into one int, we limit workerCount to
8   * (2^29)-1 (about 500 million) threads rather than (2^31)-1 (2
9   * billion) otherwise representable. If this is ever an issue in
10  * the future, the variable can be changed to be an AtomicLong,
11  * and the shift/mask constants below adjusted. But until the need
12  * arises, this code is a bit faster and simpler using an int.
13  *
14  * The workerCount is the number of workers that have been
15  * permitted to start and not permitted to stop. The value may be
16  * transiently different from the actual number of live threads,
17  * for example when a ThreadFactory fails to create a thread when
18  * asked, and when exiting threads are still performing
19  * bookkeeping before terminating. The user-visible pool size is
20  * reported as the current size of the workers set.
21  *
22  * The runState provides the main lifecycle control, taking on values:
23  *
24  *   RUNNING:  Accept new tasks and process queued tasks
25  *   SHUTDOWN: Don't accept new tasks, but process queued tasks
26  *   STOP:     Don't accept new tasks, don't process queued tasks,
27  *             and interrupt in-progress tasks
28  *   TIDYING:  All tasks have terminated, workerCount is zero,
29  *             the thread transitioning to state TIDYING
30  *             will run the terminated() hook method
31  *   TERMINATED: terminated() has completed
32  *
33  * The numerical order among these values matters, to allow
34  * ordered comparisons. The runState monotonically increases over
35  * time, but need not hit each state. The transitions are:
36  *
37  *   RUNNING -> SHUTDOWN
38  *   On invocation of shutdown(), perhaps implicitly in finalize()
39  *   (RUNNING or SHUTDOWN) -> STOP
40  *   On invocation of shutdownNow()
41  *   SHUTDOWN -> TIDYING
42  *   When both queue and pool are empty
43  *   STOP -> TIDYING
44  *   When pool is empty
```

6

19

```
45 * TIDYING -> TERMINATED
46 *   When the terminated() hook method has completed
47 *
48 * Threads waiting in awaitTermination() will return when the
49 * state reaches TERMINATED.
50 *
51 * Detecting the transition from SHUTDOWN to TIDYING is less
52 * straightforward than you'd like because the queue may become
53 * empty after non-empty and vice versa during SHUTDOWN state, but
54 * we can only terminate if, after seeing that it is empty, we see
55 * that workerCount is 0 (which sometimes entails a recheck -- see
56 * below).
```

ctl 是一个 AtomicInteger 对象, 也就是一个特殊的 int 型变量, 特殊之处在于所有需要修改其数值的操作都是原子化的. 如果你不熟悉原子化 (atomic) 这个概念, 你可以将它简单理解为 synchronized, 即: 所有修改其数值的操作都需要在加了同步锁的情况下来进行.

一个 ctl 变量可以包含两部分信息: 线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount). 由于 int 型的变量是由32位二进制的数构成, 所以用高3位来表示线程池的运行状态, 用低29位来表示线程池内有效线程的数量. 由于这两部分信息在该类中很多地方都会使用到, 所以我们也经常会涉及到要获取一个信息或操作, 通常来说, 代表这两个信息的变量的名称直接用他们各自英文单词首字母的组合来表示, 所以, 表示线程池运行状态的变量通常命名为 rs, 表示线程池内有效线程数量的变量通常命名为 wc, 另外, ctl 也通常会简写作 c, 你一定要对这里提到的几个变量名稍微留个印象哦. 如果你在该类源码的某个地方遇到了见不着的变量名时, 你在抱怨这糟糕的命名的时候, 要试着去核实一下, 那些变量是不是正是这里提到的几个信息哦.

由于 ctl 变量是由线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)这两个信息组合而成, 所以, 如果知道了这两部分信息各自的数值, 就可以用下面的 ctlOf() 方法来计算出 ctl 的数值:

```
1 // rs: 表示线程池的运行状态 (rs 是 runState中各单词首字母的简写组合)
2 // wc: 表示线程池内有效线程的数量 (wc 是 workerCount中各单词首字母的简写组合)
3 private static int ctlOf(int rs, int wc) { return rs | wc; }
```

反过来, 如果知道了 ctl 的值, 那么也可以通过如下的 runStateOf() 和 workerCountOf() 两个方法来分别获取线程池的运行状态和线程池内有效线程的数量.

```
1 private static int runStateOf(int c)      { return c & ~CAPACITY; }
2 private static int workerCountOf(int c)   { return c & CAPACITY; }
```

其中, CAPACITY 等于 (2^29)-1, 也就是高3位是0, 低29位是1的一个int型的数,

```
1 private static final int COUNT_BITS = Integer.SIZE - 3;      // 29
2 private static final int CAPACITY = (1 << COUNT_BITS) - 1;   // COUNT_BITS == 29
```

所以上边两个方法的计算过程也就不难理解了吧 (ps: 如果此时你还是不理解这两个方法的计算过程, 请先学习二进制位运算的相关知识, 然后再来看这两个方法, 你会发现他们很容易理解的). 另外, CAPACITY 这个常量从名字上可以知道, 该常量表示某个容量值, 那么表示的是什么容量值呢? 其实, 我们前面介绍过, ctl 只用低29位来表示线程池内的有效线程数, 也就是说, 线程池内有效线程的数量上限就是29个二进制1所表示的数值 (约为5亿), 而线程池就是用 CAPACITY 这个常量来表示这个上限数值的.

下面再介绍下线程池的运行状态. 线程池一共有五种状态, 分别是:

- ① **RUNNING (运行状态):** 能接受新提交的任务, 并且也能处理阻塞队列中的任务.
- ② **SHUTDOWN (关闭状态):** 不再接受新提交的任务, 但却可以继续处理阻塞队列中已保存的任务. 在线程池处于 RUNNING 状态时, 调用 shutdown()方法会使线程池进入到该状态. 当然, finalize() 方法在执行过程中或许也会隐式地进入该状态.
- ③ **STOP :** 不能接受新提交的任务, 也不能处理阻塞队列中已保存的任务, 并且会中断正在处理中的任务. 在线程池处于 RUNNING 或 SHUTDOWN 状态时, 调用 shutdownNow()方法会使线程池进入到该状态.
- ④ **TIDYING (清理状态):** 所有的任务都已终止了, workerCount (有效线程数) 为0, 线程池进入该状态后会调用 terminated() 方法以让该线程池进入TERMINATED 状态. 当线程池处于 SHUTDOWN 状态时, 如果此后线程池内没有线程了并且阻塞队列内也没有待执行的任务了 (即: 二者都为空), 线程池就会进入到该状态. 当线程池处于 STOP 状态时, 如果此后线程池内没有线程了, 线程池就会进入到该状态.
- ⑤ **TERMINATED :** terminated() 方法执行完后就进入该状态.

中文翻译可能不太准确, 也不能充分表达源码所表示的所有含义, 还可能造成歧义, 例如: STOP 和 TERMINATED 似乎翻译过来的意思没太大区别啊. 所以我们在线程池的运行状态时, 建议直接使用上面的5个英文单词来表示. 这五种状态的具体数值如下:

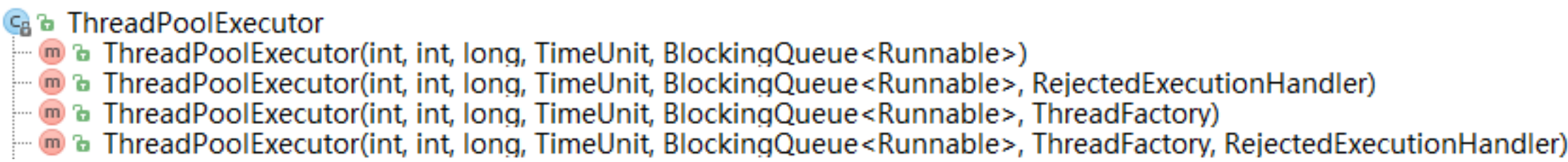
```
1 // runState is stored in the high-order bits
2 private static final int RUNNING      = -1 << COUNT_BITS;
3 private static final int SHUTDOWN    = 0 << COUNT_BITS;
4 private static final int STOP        = 1 << COUNT_BITS;
5 private static final int TIDYING     = 2 << COUNT_BITS;
6 private static final int TERMINATED  = 3 << COUNT_BITS;
```



前边提到过 COUNT\_BITS == 29. 其实我们只需要知道, 上边这5个常量是按照从小到大的顺序列出的即可. 如果你在源码中看到 rs < SHUTDOWN (假如用 rs 代表线程池的运行状态), 那么你要知道, 这表示线程池处于 RUNNING 状态.

## 2. 几个重要的参数

ThreadPoolExecutor 类的构造方法中提供了几个非常重要的参数, 这几个参数也对应着该类中的几个同名的字段. 理解这几个重要参数/字段的含义, 将有助于析线程池对线程调度的原理. 下面我们就来看看该类的构造方法吧, 如下图所示:



前三个方法最终都会去调用第四个方法, 也就是参数数量最多的那个方法, 所以我们来看看这第四个方法的源码, 如下:

```
1  /**
2   * Creates a new {@code ThreadPoolExecutor} with the given initial
3   * parameters.
4   *
5   * @param corePoolSize the number of threads to keep in the pool, even
6   *     if they are idle, unless {@code allowCoreThreadTimeOut} is set
7   * @param maximumPoolSize the maximum number of threads to allow in the
8   *     pool
9   * @param keepAliveTime when the number of threads is greater than
10  *     the core, this is the maximum time that excess idle threads
11  *     will wait for new tasks before terminating.
12  * @param unit the time unit for the {@code keepAliveTime} argument
13  * @param workQueue the queue to use for holding tasks before they are
14  *     executed. This queue will hold only the {@code Runnable}
15  *     tasks submitted by the {@code execute} method.
16  * @param threadFactory the factory to use when the executor
17  *     creates a new thread
18  * @param handler the handler to use when execution is blocked
19  *     because the thread bounds and queue capacities are reached
20  * @throws IllegalArgumentException if one of the following holds:<br>
21  *     {@code corePoolSize < 0}<br>
22  *     {@code keepAliveTime < 0}<br>
23  *     {@code maximumPoolSize <= 0}<br>
24  *     {@code maximumPoolSize < corePoolSize}
25  * @throws NullPointerException if {@code workQueue}
26  *     or {@code threadFactory} or {@code handler} is null
27  */
28 public ThreadPoolExecutor(int corePoolSize,
29                           int maximumPoolSize,
30                           long keepAliveTime,
31                           TimeUnit unit,
32                           BlockingQueue<Runnable> workQueue,
33                           ThreadFactory threadFactory,
34                           RejectedExecutionHandler handler) {
35     if (corePoolSize < 0 ||
36         maximumPoolSize <= 0 ||
37         maximumPoolSize < corePoolSize ||
38         keepAliveTime < 0)
39         throw new IllegalArgumentException();
40     if (workQueue == null || threadFactory == null || handler == null)
41         throw new NullPointerException();
42     this.corePoolSize = corePoolSize;
43     this.maximumPoolSize = maximumPoolSize;
44     this.workQueue = workQueue;
45     this.keepAliveTime = unit.toNanos(keepAliveTime);
46     this.threadFactory = threadFactory;
47     this.handler = handler;
```

从上述源码可知, 传递的参数必须符合如下要求:

如果传递的所有参数都符合上述要求, 那么就会执行后边的6个赋值语句, 将6个参数分别赋值给该类内部的6个成员字段. 接下来我们就分别分析一下这6个参数的含义. 而要想正确理解这些参数以及对应字段的含义, 需要同时结合该构造方法的注释以及对应字段的注释, 对应字段的getter(), setter()方法的注释才能基本理解透彻, 有时甚至还需要结合源码才能真正理解. 下面我们来逐一分析这几个参数(其实也就是分析6个成员字段的含义).

- corePoolSize

将该类最前边关于 Core and maximum pool sizes 和 On-demand construction 的注释, 构造方法对该参数 corePoolSize 的注释, 以及该类对同名字段 corePoolSize 的注释汇总如下:

```
1  /**
2   * <dt>Core and maximum pool sizes</dt>
3   *
4   * <dd>A {@code ThreadPoolExecutor} will automatically adjust the
5   * pool size (see {@link #getPoolSize})
6   * according to the bounds set by
7   * corePoolSize (see {@link #getCorePoolSize}) and
8   * maximumPoolSize (see {@link #getMaximumPoolSize}).
9   *
10  * When a new task is submitted in method {@link #execute(Runnable)},
11  * and fewer than corePoolSize threads are running, a new thread is
12  * created to handle the request, even if other worker threads are
13  * idle. If there are more than corePoolSize but less than
14  * maximumPoolSize threads running, a new thread will be created only
15  * if the queue is full. By setting corePoolSize and maximumPoolSize
16  * the same, you create a fixed-size thread pool. By setting
17  * maximumPoolSize to an essentially unbounded value such as {@code
18  * Integer.MAX_VALUE}, you allow the pool to accommodate an arbitrary
19  * number of concurrent tasks. Most typically, core and maximum pool
20  * sizes are set only upon construction, but they may also be changed
21  * dynamically using {@link #setCorePoolSize} and {@link
22  * #setMaximumPoolSize}. </dd>
23  *
24  * <dt>On-demand construction</dt>
25  *
26  * <dd>By default, even core threads are initially created and
27  * started only when new tasks arrive, but this can be overridden
28  * dynamically using method {@link #prestartCoreThread} or {@link
29  * #prestartAllCoreThreads}. You probably want to prestart threads if
30  * you construct the pool with a non-empty queue. </dd>
31  */
32
33 /**
34  * Core pool size is the minimum number of workers to keep alive
35  * (and not allow to time out etc) unless allowCoreThreadTimeOut
36  * is set, in which case the minimum is zero.
37  */
38 private volatile int corePoolSize;
39
40 /**
41  * Creates a new {@code ThreadPoolExecutor} with the given initial
42  * parameters.
43  *
44  * @param corePoolSize the number of threads to keep in the pool, even
45  *       if they are idle, unless {@code allowCoreThreadTimeOut} is set
46  * ...
47  */
48 public ThreadPoolExecutor(int corePoolSize,
49                           int maximumPoolSize,
50                           long keepAliveTime,
51                           TimeUnit unit,
52                           BlockingQueue<Runnable> workQueue,
53                           ThreadFactory threadFactory,
54                           RejectedExecutionHandler handler) {
55     // ...
56     this.corePoolSize = corePoolSize;
57     "
```

结合上述注释可知, `corePoolSize` 字段表示的是线程池中一直存活着的线程的最小数量, 这些一直存活着的线程又被称为核心线程. 默认情况下, 核心线程的数量都是正数, 除非调用了`allowCoreThreadTimeOut()`方法并传递参数为`true`, 设置允许核心线程因超时而停止(`terminated`), 在那种情况下, 一旦所有的核心线程后因超时而停止了, 将使得线程池中的核心线程数量最终变为0, 也就是一直存活着的线程数为0, 这将是那种情况下, 线程池中核心线程数量的最小值. 默认情况下, 核心线程是按需创建并启动的, 也就是说, 只有当线程池接收到我们提交给他的任务后, 他才会去创建并启动一定数量的核心线程来执行这些任务. 如果他没有接受到任务, 他就不会主动去创建核心线程. 这种默认的核心线程的创建启动机制, 有助于降低系统资源的消耗. 变主动为被动, 类似于常见的观察者模式. 当然这只是默认的方式, 如果有特殊需求的话, 我们也可以通过调用 `prestartCoreThread()` 或 `prestartAllCoreThreads()` 方法来改变这一机制, 使得在新任务还未提交到线程池的时候, 线程池就已经创建并启动了一个或所有核心线程, 并让这些核心线程在池子里等待着新任务的到来.

- **maximumPoolSize**

将该类最前边对 `Core` and `maximum pool sizes` 的注释, 构造方法对该参数 `maximumPoolSize` 的注释, 以及该类中的同名字段 `maximumPoolSize` 的注释进行如下:

```
1  /**
2   * <dt>Core and maximum pool sizes</dt>
3   *
4   * <dd>A {@code ThreadPoolExecutor} will automatically adjust the
5   * pool size (see {@link #getPoolSize})
6   * according to the bounds set by
7   * corePoolSize (see {@link #getCorePoolSize}) and
8   * maximumPoolSize (see {@link #getMaximumPoolSize}).
9   *
10  * When a new task is submitted in method {@link #execute(Runnable)},
11  * and fewer than corePoolSize threads are running, a new thread is
12  * created to handle the request, even if other worker threads are
13  * idle. If there are more than corePoolSize but less than
14  * maximumPoolSize threads running, a new thread will be created only
15  * if the queue is full. By setting corePoolSize and maximumPoolSize
16  * the same, you create a fixed-size thread pool. By setting
17  * maximumPoolSize to an essentially unbounded value such as {@code
18  * Integer.MAX_VALUE}, you allow the pool to accommodate an arbitrary
19  * number of concurrent tasks. Most typically, core and maximum pool
20  * sizes are set only upon construction, but they may also be changed
21  * dynamically using {@link #setCorePoolSize} and {@link
22  * #setMaximumPoolSize}. </dd>
23  */
24
25 /**
26  * Maximum pool size. Note that the actual maximum is internally
27  * bounded by CAPACITY.
28  */
29 private volatile int maximumPoolSize;
30
31 /**
32  * Creates a new {@code ThreadPoolExecutor} with the given initial
33  * parameters.
34  *
35  * @param maximumPoolSize the maximum number of threads to allow in the
36  *       pool
37  */
38 public ThreadPoolExecutor(int corePoolSize,
39                           int maximumPoolSize,
40                           long keepAliveTime,
41                           TimeUnit unit,
42                           BlockingQueue<Runnable> workQueue,
43                           ThreadFactory threadFactory,
44                           RejectedExecutionHandler handler) {
45     // ...
46     this.maximumPoolSize = maximumPoolSize;
47     // ...
```

`maximumPoolSize` 表示线程池内能够容纳线程数量的最大值. 当然, 线程数量的最大值还不能超过常量 `CAPACITY` 的数值大小, 根据如下的源码, 我们很容易知道 `CAPACITY` 的数值等于  $(1 \ll 29 - 1)$ , 也就是 1先左移29位然后再减1以后的数值, 在介绍 `ctl` 的时候曾经提到过, 这个数值约等于5亿.

```
1 private static final int COUNT_BITS = Integer.SIZE - 3;
2 private static final int CAPACITY = (1<< COUNT_BITS) - 1;
```

所以, 我们设定的参数 `maximumPoolSize` 和 `CAPACITY` 二者中较小的那个数值才是线程池中线程数量的最大值. 这里多说一句, 如果我们提供的阻塞队列 (也



数 `workQueue`) 是一个无界的队列, 那么这里提供的 `maximumPoolSize` 的数值将毫无意义, 这一点我们会在后面解释. 当我们通过方法 `execute(Runnable)` 个任务到线程池时, 如果处于运行状态(RUNNING)的线程数量少于核心线程数(`corePoolSize`), 那么即使有一些非核心线程处于空闲等待状态, 系统也会倾向于个新的线程来处理这个任务. 如果此时处于运行状态(RUNNING)的线程数量大于核心线程数(`corePoolSize`), 但又小于最大线程数(`maximumPoolSize`), 那么系去判断线程池内部的阻塞队列 `workQueue` 中是否还有空位子. 如果发现有空位子, 系统就会将该任务先存入该阻塞队列; 如果发现队列中已没有空位子(即: 队系统就会新创建一个线程来执行该任务.

如果将线程池的核心线程数 `corePoolSize` 和 最大线程数 `maximumPoolSize` 设置为相同的数值(也就是说, 线程池中的所有线程都是核心线程), 那么该线程池个容量固定的线程池. 如果将最大线程数 `maximumPoolSize` 设置为一个非常大的数值(例如: `Integer.MAX_VALUE`), 那么就相当于允许线程池自己在不同时段调整参与并发的任务总数. 通常情况下, 核心线程数 `corePoolSize` 和 最大线程数 `maximumPoolSize` 仅在创建线程池的时候才去进行设定, 但是, 如果在线程池成以后, 你又想去修改这两个字段的值, 你就可以调用 `setCorePoolSize()` 和 `setMaximumPoolSize()` 方法来分别重新设定核心线程数 `corePoolSize` 和 最大线程数 `maximumPoolSize` 的数值.

- `keepAliveTime`

将该类最前面关于 `Keep-alive times` 的注释, 构造方法对参数`keepAliveTime` 的注释, 以及字段 `keepAliveTime` 的注释汇总如下:

```
1  /**
2   * <dt>Keep-alive times</dt>
3   *
4   * <dd>If the pool currently has more than corePoolSize threads,
5   * excess threads will be terminated if they have been idle for more
6   * than the keepAliveTime (see {@link #getKeepAliveTime(TimeUnit)}).
7   * This provides a means of reducing resource consumption when the
8   * pool is not being actively used. If the pool becomes more active
9   * later, new threads will be constructed. This parameter can also be
10  * changed dynamically using method {@link #setKeepAliveTime(long,
11  * TimeUnit)}. Using a value of {@code Long.MAX_VALUE} {@link
12  * TimeUnit#NANOSECONDS} effectively disables idle threads from ever
13  * terminating prior to shut down. By default, the keep-alive policy
14  * applies only when there are more than corePoolSize threads. But
15  * method {@link #allowCoreThreadTimeOut(boolean)} can be used to
16  * apply this time-out policy to core threads as well, so long as the
17  * keepAliveTime value is non-zero. </dd>
18  */
19
20 /**
21  * Timeout in nanoseconds for idle threads waiting for work.
22  * Threads use this timeout when there are more than corePoolSize
23  * present or if allowCoreThreadTimeOut. Otherwise they wait
24  * forever for new work.
25  */
26 private volatile long keepAliveTime;
27
28 /**
29  * Creates a new {@code ThreadPoolExecutor} with the given initial
30  * parameters.
31  *
32  * @param keepAliveTime when the number of threads is greater than
33  *       the core, this is the maximum time that excess idle threads
34  *       will wait for new tasks before terminating.
35  */
36 public ThreadPoolExecutor(int corePoolSize,
37                           int maximumPoolSize,
38                           long keepAliveTime,
39                           TimeUnit unit,
40                           BlockingQueue<Runnable> workQueue,
41                           ThreadFactory threadFactory,
42                           RejectedExecutionHandler handler) {
43     // ...
44     this.keepAliveTime = unit.toNanos(keepAliveTime);
45     // ...
```

从注释可知, `keepAliveTime` 表示空闲线程处于等待状态的超时时间(也即, 等待时间的上限值, 超过该时间后该线程会停止工作). 当总线程数大于 `corePoolSize` (核心线程数) 并且 `allowCoreThreadTimeOut` 为 `false` 时, 这些多出来的非核心线程一旦进入到空闲等待状态, 就开始计算各自的等待时间, 并用这里设定的 `keepAliveTime` 的数值作为他们的超时时间, 一旦某个非核心线程的等待时间达到了超时时间, 该线程就会停止工作(terminated), 而核心线程在这种情况下却不会受超时机制的核心线程即使等待的时间超出了这里设定的 `keepAliveTime`, 也依然可以继续处于空闲等待状态而不会停止工作. 但是, 如果 `allowCoreThreadTimeOut` 被设置并且设置的 `keepAliveTime > 0`, 那么不论是非核心线程还是核心线程, 都将受超时机制的制约. 所以, 如果要执行的任务相对较多, 并且每个任务执行的时间比那么可以为该参数设置一个相对较大的数值, 以提高线程的利用率. 如果执行的任务相对较少, 线程池使用率相对较低, 那么可以先将该参数设置为一个较小值.

通过超时停止的机制来降低系统线程资源的开销, 后续如果发现线程池的使用率逐渐增高以后, 线程池会根据当前提交的任务数自动创建新的线程, 当然, 我们

自己手动调用 `setKeepAliveTime(long, TimeUnit)`方法来重新设定 `keepAliveTime` 字段的值. 如果将 `keepAliveTime` 和 `unit` 这两个参数分别设置为 `Long.MAX_VALUE` 和 `TimeUnit.NANOSECONDS`(纳秒), 这可以让空闲线程基本上一直处于存活状态. 因为这两个数值的组合表示设置超时时间约为292年, 是一个非常非常长的时间了, 所以我们可以认为这种设置基本等效于让空闲线程一直处于存活状态. 另外值得注意的是, 构造方法中的参数 `keepAliveTime` 的数值和字段的 `keepAliveTime` 的数值很可能不同. 因为参数 `keepAliveTime` 对应的时间单位可以是任意的, 这取决于另一个参数 `unit` 赋的是什么值, 可选的值有纳秒(`NANOSECONDS`), 微秒(`MICROSECONDS`), 毫秒(`MILLISECONDS`), 秒(`SECONDS`), 分钟(`MINUTES`), 小时(`HOURS`), 天(`DAYS`). 而与其同名的字段 `keepAliveTime` 对应的时间单位则被强制要求是纳秒. 所以, 构造方法会将参数 `keepAliveTime` 和 `unit` 二者组合起来的时间值换算成以纳秒为单位的数值, 并将得到的数值作为字段 `keepAliveTime` 的值. 注意这里的参数 `keepAliveTime` 是 `long` 型而不是 `int` 型的, 因为如果 `keepAliveTime` 为 `int` 型, 并且为 `unit` 赋的值是 `NANOSECONDS`(纳秒), 那么即使取 `int` 型的最大值 `Integer.MAX_VALUE` 作为 `keepAliveTime` 的数值, 设置的超时时间就是 `0x7FFFFFFF` 纳秒, 换算成秒就是 0.7 秒. 即: 这种情况下能够设置的最大超时时间是2秒, 时间太短, 所以将参数 `keepAliveTime` 定义为 `long` 型就保证了能够设置的超时时间不至于太短.

- `workQueue`

从源码可知, 这里的 `workQueue` 是一个 `BlockingQueue`(阻塞队列) 的实例, 传入的泛型类型是 `Runnable`. 也就是说, `workQueue` 是一个内部元素为 `Runnable` (任务, 通常是异步的任务) 的阻塞队列. 阻塞队列是一种类似于 “生产者 - 消费者”模型的队列. 当队列已满时如果继续向队列中插入元素, 该插入操作将被阻塞一直等待状态, 直到队列中有元素被移除产生空位子后, 才有可能执行这次插入操作; 当队列为空时如果继续执行元素的删除或获取操作, 该操作同样会被阻塞而进入等待状态, 直到队列中又有了该元素后, 才有可能执行该操作.

下面将该类最前面的注释中关于 `Queuing` 的那部分注释, 字段 `workQueue` 的注释, 以及构造方法对参数 `workQueue` 的注释三者汇总如下:

```
1  /**
2   * <dt>Queuing</dt>
3   *
4   * Any {@link BlockingQueue} may be used to transfer and hold
5   * submitted tasks. The use of this queue interacts with pool sizing:
6   *
7   * If fewer than corePoolSize threads are running, the Executor
8   * always prefers adding a new thread
9   * rather than queuing.
10  *
11  * If corePoolSize or more threads are running, the Executor
12  * always prefers queuing a request rather than adding a new
13  * thread.
14  *
15  * If a request cannot be queued, a new thread is created unless
16  * this would exceed maximumPoolSize, in which case, the task will be
17  * rejected.
18  *
19  *
20  * There are three general strategies for queuing:
21  *
22  * <em> Direct handoffs.</em> A good default choice for a work
23  * queue is a {@link SynchronousQueue} that hands off tasks to threads
24  * without otherwise holding them. Here, an attempt to queue a task
25  * will fail if no threads are immediately available to run it, so a
26  * new thread will be constructed. This policy avoids lockups when
27  * handling sets of requests that might have internal dependencies.
28  * Direct handoffs generally require unbounded maximumPoolSizes to
29  * avoid rejection of new submitted tasks. This in turn admits the
30  * possibility of unbounded thread growth when commands continue to
31  * arrive on average faster than they can be processed.
32  *
33  * <em> Unbounded queues.</em> Using an unbounded queue (for
34  * example a {@link LinkedBlockingQueue} without a predefined
35  * capacity) will cause new tasks to wait in the queue when all
36  * corePoolSize threads are busy. Thus, no more than corePoolSize
37  * threads will ever be created. (And the value of the maximumPoolSize
38  * therefore doesn't have any effect.) This may be appropriate when
39  * each task is completely independent of others, so tasks cannot
40  * affect each others execution; for example, in a web page server.
41  * While this style of queuing can be useful in smoothing out
42  * transient bursts of requests, it admits the possibility of
43  * unbounded work queue growth when commands continue to arrive on
44  * average faster than they can be processed.
45  *
46  * <em> Bounded queues.</em> A bounded queue (for example, an
47  * {@link ArrayBlockingQueue}) helps prevent resource exhaustion when
48  * used with finite maximumPoolSizes, but can be more difficult to
```



```
49  * tune and control. Queue sizes and maximum pool sizes may be traded
50  * off for each other: Using large queues and small pools minimizes
51  * CPU usage, OS resources, and context-switching overhead, but can
52  * lead to artificially low throughput. If tasks frequently block (for
53  * example if they are I/O bound), a system may be able to schedule
54  * time for more threads than you otherwise allow. Use of small queues
55  * generally requires larger pool sizes, which keeps CPUs busier but
56  * may encounter unacceptable scheduling overhead, which also
57  * decreases throughput.
58  */
59
60 /**
61  * The queue used for holding tasks and handing off to worker
62  * threads. We do not require that workQueue.poll() returning
63  * null necessarily means that workQueue.isEmpty(), so rely
64  * solely on isEmpty to see if the queue is empty (which we must
65  * do for example when deciding whether to transition from
66  * SHUTDOWN to TIDYING). This accommodates special-purpose
67  * queues such as DelayQueues for which poll() is allowed to
68  * return null even if it may later return non-null when delays
69  * expire.
70  */
71 private final BlockingQueue<Runnable> workQueue;
72
73 /**
74  * Creates a new {@code ThreadPoolExecutor} with the given initial
75  * parameters.
76  *
77  * @param workQueue the queue to use for holding tasks before they are
78  *     executed. This queue will hold only the {@code Runnable}
79  *     tasks submitted by the {@code execute} method.
80  */
81 public ThreadPoolExecutor(int corePoolSize,
82                           int maximumPoolSize,
83                           long keepAliveTime,
84                           TimeUnit unit,
85                           BlockingQueue<Runnable> workQueue,
86                           ThreadFactory threadFactory,
87                           RejectedExecutionHandler handler) {
88     //
```

workQueue 是一个用于保存等待执行的任务的阻塞队列. 当提交一个新的任务到线程池以后, 线程池会根据当前池中正在运行着的线程的数量, 指定出对该任务的处理方式, 主要有以下几种处理方式:

- (1) 如果线程池中正在运行的线程数少于核心线程数, 那么线程池总是倾向于创建一个新线程来执行该任务, 而不是将该任务提交到该队列 workQueue 中进行等待.
- (2) 如果线程池中正在运行的线程数不少于核心线程数, 那么线程池总是倾向于将该任务先提交到队列 workQueue 中先让其等待, 而不是创建一个新线程来执行该任务.
- (3) 如果线程池中正在运行的线程数不少于核心线程数, 并且线程池中的阻塞队列也满了使得该任务入队失败, 那么线程池会去判断当前池中运行的线程数是否等于了该线程池允许运行的最大线程数 maximumPoolSize. 如果发现已经等于了, 说明池子已满, 无法再继续创建新的线程了, 那么就会拒绝执行该任务. 如果运行的线程数小于池子允许的最大线程数, 那么就会创建一个线程(这里创建的线程是非核心线程)来执行该任务.

一般来说, 队列对新提交的任务有三种常见的处理策略:

- (1) 直接切换.** 常用的队列是 SynchronousQueue (同步队列). 这种队列内部不会存储元素. 每一次插入操作都会先进入阻塞状态, 一直等到另一个线程执行了删除操作, 然后该插入操作才会执行. 同样地, 每一次删除操作也都会先进入阻塞状态, 一直等到另一个线程执行了插入操作, 然后该删除操作才会执行. 当提交一个任务到包含这种 SynchronousQueue 队列的线程池以后, 线程池会去检测是否有可用的空闲线程来执行该任务, 如果没有就直接新建一个线程来执行该任务而不是将该任务暂存在队列中. “直接切换”的意思就是, 处理方式由”将任务暂时存入队列”直接切换为”新建一个线程来处理该任务”. 这种策略适合用来处理多个有相互依赖关系的任务, 因为该策略可以避免这些任务因一个没有及时处理而导致依赖于该任务的其他任务也不能及时处理而造成的锁定效果. 因为这种策略的目的是要让几乎每一个任务都能得到立即处理, 所以这种策略通常要求最大线程数 maximumPoolSizes 是无界的(即: Integer.MAX\_VALUE). 静态工厂方法 Executors.newCachedThreadPool() 使用了这个队列.
- (2) 使用无界队列** (也就是不预设队列的容量, 队列将使用 Integer.MAX\_VALUE 作为其默认容量, 例如: 基于链表的阻塞队列 LinkedBlockingQueue). 使用无界队列使得线程池中能够创建的最大线程数就等于核心线程数 corePoolSize, 这样线程池的 maximumPoolSize 的数值起不到任何作用. 如果向这种线程池中提交一个任务时发现所有核心线程都处于运行状态, 那么该任务将被放入无界队列中等待处理. 当要处理的多个任务之间没有任何相互依赖关系时, 就适合使用这种队列策略来处理这些任务. 静态工厂方法 Executors.newFixedThreadPool() 使用了这个队列.
- (3) 使用有界队列** (例如: 基于数组的阻塞队列 ArrayBlockingQueue). 当要求线程池的最大线程数 maximumPoolSizes 要限定在某个值以内时, 线程池使用有界队列可以降低资源的消耗, 但这也使得线程池对线程的调控变得更加困难. 因为队列容量和线程池容量都是有限的值, 要想使线程处理任务的吞吐量能够在一个相对合理的范围内, 同时又能使线程调配的难度相对较低, 并且又尽可能节省系统资源的消耗, 那么就需要合理地调配这两个数值. 通常来说, 设置较大的队列容量和较小的线程数, 能够降低系统资源的消耗(包括CPU的使用率, 操作系统资源的消耗, 上下文环境切换的开销等), 但却会降低线程处理任务的吞吐量. 如果发现提交的任务经常性地发生阻塞的情况, 那么你就可以考虑增大线程池的容量, 可以通过调用 setMaximumPoolSize() 方法来重新设定线程池的容量. 而设置较小的队列容量时, 通



将线程池的容量设置大一点, 这种情况下, CPU的使用率会相对较高, 当然如果线程池的容量设置过大的话, 可能会有非常非常多的线程来同时处理提交来的多并发数过大时, 线程之间的调度将会是个非常严峻的问题, 这反而有可能降低任务处理的吞吐量, 出现过犹不及的局面.

- **threadFactory**

将该类前边关于该参数的那部分注释 “Creating new threads”, 构造方法对该参数的注释, 以及该类对字段 threadFactory 的注释汇总如下:

```
1  /** <dt>Creating new threads</dt>
2  *
3  * <dd>New threads are created using a {@link ThreadFactory}. If not
4  * otherwise specified, a {@link Executors#defaultThreadFactory} is
5  * used, that creates threads to all be in the same {@link
6  * ThreadGroup} and with the same {@code NORM_PRIORITY} priority and
7  * non-daemon status. By supplying a different ThreadFactory, you can
8  * alter the thread's name, thread group, priority, daemon status,
9  * etc. If a {@code ThreadFactory} fails to create a thread when asked
10 * by returning null from {@code newThread}, the executor will
11 * continue, but might not be able to execute any tasks.</dd>
12 */
13
14 /**
15 * Factory for new threads. All threads are created using this
16 * factory (via method addWorker). All callers must be prepared
17 * for addWorker to fail, which may reflect a system or user's
18 * policy limiting the number of threads. Even though it is not
19 * treated as an error, failure to create threads may result in
20 * new tasks being rejected or existing ones remaining stuck in
21 * the queue.
22 *
23 * We go further and preserve pool invariants even in the face of
24 * errors such as OutOfMemoryError, that might be thrown while
25 * trying to create threads. Such errors are rather common due to
26 * the need to allocate a native stack in Thread.start, and users
27 * will want to perform clean pool shutdown to clean up. There
28 * will likely be enough memory available for the cleanup code to
29 * complete without encountering yet another OutOfMemoryError.
30 */
31 private volatile ThreadFactory threadFactory;
32
33 /**
34 * Creates a new {@code ThreadPoolExecutor} with the given initial
35 * parameters.
36 *
37 * @param threadFactory the factory to use when the executor
38 *     creates a new thread
39 */
40 public ThreadPoolExecutor(int corePoolSize,
41                           int maximumPoolSize,
42                           long keepAliveTime,
43                           TimeUnit unit,
44                           BlockingQueue<Runnable> workQueue,
45                           ThreadFactory threadFactory,
46                           RejectedExecutionHandler handler) {
47     // ...
48     this.threadFactory = threadFactory;
49     // ...
```

threadFactory 线程工厂, 用于创建线程. 如果我们在创建线程池的时候未指定该 threadFactory 参数, 线程池则会使用 Executors.defaultThreadFactory() 方法认的线程工厂. 如果我们想要为线程工厂创建的线程设置一些特殊的属性, 例如: 设置见名知意的名字, 设置特定的优先级等等, 那么我们就需要自己去实现 ThreadFactory 接口, 并在实现其抽象方法 newThread()的时候, 使用Thread类包含 threadName (线程名字)的那个构造方法就可以指定线程的名字(通常可以见名知意的名字), 还可以用 setPriority() 方法为线程设置特定的优先级等. 然后在创建线程池的时候, 将我们自己实现的 ThreadFactory 接口的实现类对象作为 threadFactory 参数的值传递给线程池的构造方法即可.

- **handler**

将该类前边关于该参数的那部分注释 “Rejected tasks”, 构造方法对该参数的注释, 以及该类对字段 handler 的注释汇总如下:

```
1  /**
2  * <dt>Rejected tasks</dt>
```

```
3  *
4  * <dd>New tasks submitted in method {@link #execute(Runnable)} will be
5  * <em>rejected</em> when the Executor has been shut down, and also when
6  * the Executor uses finite bounds for both maximum threads and work queue
7  * capacity, and is saturated. In either case, the {@code execute} method
8  * invokes the {@link
9  * RejectedExecutionHandler#rejectedExecution(Runnable, ThreadPoolExecutor)}
10 * method of its {@link RejectedExecutionHandler}. Four predefined handler
11 * policies are provided:
12 *
13 * <ol>
14 *
15 * <li> In the default {@link ThreadPoolExecutor.AbortPolicy}, the
16 * handler throws a runtime {@link RejectedExecutionException} upon
17 * rejection. </li>
18 *
19 * <li> In {@link ThreadPoolExecutor.CallerRunsPolicy}, the thread
20 * that invokes {@code execute} itself runs the task. This provides a
21 * simple feedback control mechanism that will slow down the rate that
22 * new tasks are submitted. </li>
23 *
24 * <li> In {@link ThreadPoolExecutor.DiscardPolicy}, a task that
25 * cannot be executed is simply dropped. </li>
26 *
27 * <li>In {@link ThreadPoolExecutor.DiscardOldestPolicy}, if the
28 * executor is not shut down, the task at the head of the work queue
29 * is dropped, and then execution is retried (which can fail again,
30 * causing this to be repeated.) </li>
31 *
32 * </ol>
33 *
34 * It is possible to define and use other kinds of {@link
35 * RejectedExecutionHandler} classes. Doing so requires some care
36 * especially when policies are designed to work only under particular
37 * capacity or queuing policies. </dd>
38 */
39
40 /**
41  * Handler called when saturated or shutdown in execute.
42  */
43 private volatile RejectedExecutionHandler handler;
44
45 /**
46  * Creates a new {@code ThreadPoolExecutor} with the given initial
47  * parameters.
48  *
49  * @param handler the handler to use when execution is blocked
50  *        because the thread bounds and queue capacities are reached
51  */
52 public ThreadPoolExecutor(int corePoolSize,
53                          int maximumPoolSize,
54                          long keepAliveTime,
55                          TimeUnit unit,
56                          BlockingQueue<Runnable> workQueue,
57                          ThreadFactory threadFactory,
58                          RejectedExecutionHandler handler) {
59     // ...
60     this.handler = handler;
```

根据注释可知, 以下两个条件满足其中任意一个的时候, 如果继续向该线程池中提交新的任务, 那么线程池将会调用他内部的 RejectedExecutionHandler 对象(handler)的 rejectedExecution()方法, 表示拒绝执行这些新提交的任务:

- ① 当线程池处于 SHUTDOWN (关闭) 状态时 (不论线程池和阻塞队列是否都已满)

② 当线程池中的所有线程都处于运行状态并且线程池中的阻塞队列已满时

可能只是用文字描述不太容易理解, 并且印象也不深刻, 下面我们写个简单的 demo来给大家展示一下这两种情况到底是什么意思:

```
1 package com.example.thread_pool_executor_test;
2
```



```
3 import java.util.concurrent.LinkedBlockingQueue;
4 import java.util.concurrent.ThreadFactory;
5 import java.util.concurrent.ThreadPoolExecutor;
6 import java.util.concurrent.TimeUnit;
7
8 /**
9  * 线程池触发 RejectedExecutionHandler拒绝新提交任务的场景模拟 demo
10  *
11  * @author zhangzhiyi
12  * @version 1.0
13  * @createTime 2016/2/25 13:53
14  * @projectName ThreadPoolExecutorTest
15  */
16 public class ThreadPoolExecutorRejectNewTaskDemo {
17
18     // 线程池的最大容量
19     private static final int MAX_POOL_SIZE = 3;
20     // 阻塞队列的容量
21     private static final int QUEUE_CAPACITY = 2;
22     // 非核心线程处于空闲状态的最长时间
23     private static final int KEEP_ALIVE_TIME_VALUE = 1;
24     // 线程池对象
25     private static final ThreadPoolExecutor THREAD_POOL_EXECUTOR = new ThreadPoolExecutor(
26         MAX_POOL_SIZE, MAX_POOL_SIZE,
27         KEEP_ALIVE_TIME_VALUE, TimeUnit.SECONDS,
28         new LinkedBlockingQueue<Runnable>(QUEUE_CAPACITY),
29         new MyThreadFactory());
30
31     public static void main(String[] args) {
32         // 模拟线程池及其内部的队列都已满后, 继续向其提交新任务将会被拒绝的场景
33         // threadPoolFullToRejectNewTask();
34
35         // 模拟线程池被关闭(shutdown)后, 继续向其提交新任务将会被拒绝的场景
36         shutdownThreadPoolToRejectNewTask();
37     }
38
39     /**
40     * 模拟线程池被关闭(shutdown)后, 继续向其提交新任务将会被拒绝的场景
41     */
42     private static void shutdownThreadPoolToRejectNewTask() {
43         MyRunnable r = new MyRunnable();
44
45         int cycleCount = Math.max(MAX_POOL_SIZE - 1, 0);
46
47         // 先提交(MAX_POOL_SIZE - 1)个任务. 显然, 线程池此时还未满
48         for (int i = 0; i < cycleCount; i++) {
49             System.out.println("提交任务" + i);
50             THREAD_POOL_EXECUTOR.execute(r);
51         }
52         // 在线程池未满的情况下关闭线程池.
53         THREAD_POOL_EXECUTOR.shutdown();
54
55         // 在线程池已处于关闭(SHUTDOWN)的状态下
56         if (THREAD_POOL_EXECUTOR.isShutdown()) {
57             try {
58                 System.out.println("提交任务" + cycleCount);
59                 Thread.sleep(10);
60                 // 在线程池未满但却已经关闭了的情况下, 继续向该线程池中提交任务.
61                 THREAD_POOL_EXECUTOR.execute(r);
62             } catch (InterruptedException e) {
63                 e.printStackTrace();
64             }
65         }
66     }
67
68     /**
69     * 模拟线程池及其内部的队列都已满后, 继续向其提交新任务将会被拒绝的场景
70     */
71     private static void threadPoolFullToRejectNewTask() {
72         MyRunnable r = new MyRunnable();
73         // 循环提交任务的总次数. 该总次数等于"线程池的最大线程容量和阻塞队列的容量之和", 在执行完
```

```
74 // 该循环后, 线程池和阻塞队列都已满.
75 int cycleCount = MAX_POOL_SIZE + QUEUE_CAPACITY;
76
77 for (int i = 0; i < cycleCount; i++) {
78     System.out.println("提交任务" + i);
79     THREAD_POOL_EXECUTOR.execute(r);
80 }
81 // 当前已提交的任务数
82 int tasksCount = cycleCount;
83
84 // 在线程池和阻塞队列都已满的情况下, 继续提交任务.
85 try {
86     System.out.println("提交任务" + (tasksCount));
87     Thread.sleep(10);
88     THREAD_POOL_EXECUTOR.execute(r);
89 } catch (InterruptedException e) {
90     e.printStackTrace();
91 }
92 }
93
94 /**
95 * 自定义的线程工厂类, 用于为线程池创建线程对象.
96 */
97 private static class MyThreadFactory implements ThreadFactory {
98     static int threadNumber = 0;
99
100     @Override
101     public Thread newThread(Runnable r) {
102         String threadName = "thread-" + (threadNumber++);
103         System.out.println("创建线程 " + threadName);
104         return new Thread(r, threadName);
105     }
106 }
107
108 /**
109 * 表示向线程池提交的任务的 Runnable实现类
110 */
111 private static class MyRunnable implements Runnable {
112     @Override
113     public void run() {
114         try {
115             Thread.sleep(10);
116         } catch (InterruptedException e) {
117             e.printStackTrace();
```

在上边的demo中, 我们定义了一个最大线程数为3, 阻塞队列容量为2的线程池, 并且我们使用的是不带 RejectedExecutionHandler 参数的那个构造方法. 我们两个条件结合到我们这个demo中来说就是:

- ① 当线程池的 isShutdown() 方法返回 true 的时候 (即使此时线程池中处于运行状态的线程数少于3个, 或者阻塞队列中的元素个数少于2个, 也会拒绝新提交的任务)
- ② 当线程池中处于运行状态的线程数为3个, 并且阻塞队列中包含的元素个数等于2个的时候, 继续提交的任务就会被拒绝.

我们定义了两个方法 shutdownThreadPoolToRejectNewTask() 和 threadPoolFullToRejectNewTask(), 来模拟在分别满足上边两个条件的情况下, 向线程池中提交任务的情景.

在方法 shutdownThreadPoolToRejectNewTask() 中, 我们先提交了 (MAX\_POOL\_SIZE - 1) 个任务, 也就是2个任务, 此时线程池还未满, 阻塞队列还是空的, 我们调用线程池的 shutdown() 方法 (也可调用 shutdownNow()方法) 将线程池关闭, 并且为了确保线程池已经关闭, 我们又调用了线程池的 isShutdown()方法来判断线程池是否已经关闭了, 在该方法返回 true的时候, 也就是说, 在确认线程池已经处于关闭(SHUTDOWN) 状态时, 我们向该线程池中又提交了一个任务 (见代码第61行), 也就是第3个任务, 我们来看看此时会发生什么情况呢? 下面是执行过程所打印的 log信息:



```
提交任务0
创建线程 thread-0
提交任务1
创建线程 thread-1
提交任务2
Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task com.example.thread_pool_executor_test
.ThreadPoolExecutorRejectNewTaskDemo$MyRunnable@6d6f6e28 rejected from java.util.concurrent.ThreadPoolExecutor@135fbaa4[Termi
pool size = 0, active threads = 0, queued tasks = 0, completed tasks = 2]
    at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2047)
    at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:823)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1369)
    at com.example.thread_pool_executor_test.ThreadPoolExecutorRejectNewTaskDemo.shutdownThreadPoolToRejectNewTask
(ThreadPoolExecutorRejectNewTaskDemo.java:61)
    at com.example.thread_pool_executor_test.ThreadPoolExecutorRejectNewTaskDemo.main(ThreadPoolExecutorRejectNewTaskDemo.java:88)
```

从log信息可以看出, 当我们提交第3个任务 (即: 任务2) 的时候, 线程池直接抛出了异常, 并且导致异常发生的代码就是61行的  
THREAD\_POOL\_EXECUTOR.execute(r); 这句代码. 所以, 线程池拒绝接受新任务的处理方式就是直接抛异常 (其实, 在看完后边的介绍后, 你就会知道, 抛异常  
其中的一种处理方式, 也是线程池默认的处理方式, 线程池还为我们提供了其他几种处理方式, 当然我们自己也可以提供自定义的处理方式).

我们再来看看另一个方法 threadPoolFullToRejectNewTask() 所模拟的场景. 在该方法中, 我们一共提交了 (MAX\_POOL\_SIZE + QUEUE\_CAPACITY + 1) 个任务  
就是6个任务, 而线程池的最大线程数和核心线程数都为3, 所以在提交前3个任务 (即: 任务0, 任务1, 任务2) 后, 线程池会分别创建3个核心线程 (此时线程池已  
理这3个任务, 接下来我们再提交的任务就会被暂时存入阻塞队列中, 而我们为阻塞队列设定的容量为2, 所以我们接下来提交的第4个和第5个任务 (即: 任务3和  
都将存入阻塞队列中 (此时阻塞队列也满了), 这时已经满足了前边提到的条件②, 而我们一共要提交6个任务, 所以当我们继续提交第6个任务 (也就是任务5, 见  
88行) 时, 同样也会被拒绝, 而拒绝的方式也是抛出异常, 见如下的 log 信息:

```
提交任务0
创建线程 thread-0
提交任务1
创建线程 thread-1
提交任务2
创建线程 thread-2
提交任务3
提交任务4
提交任务5
Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task com.example.thread_pool_executor_test
.ThreadPoolExecutorRejectNewTaskDemo$MyRunnable@135fbaa4 rejected from java.util.concurrent.ThreadPoolExecutor@45ee12a7[Run
pool size = 3, active threads = 3, queued tasks = 2, completed tasks = 0]
    at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2047)
    at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:823)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1369)
    at com.example.thread_pool_executor_test.ThreadPoolExecutorRejectNewTaskDemo.threadPoolFullToRejectNewTask?
(ThreadPoolExecutorRejectNewTaskDemo.java:88)
    at com.example.thread_pool_executor_test.ThreadPoolExecutorRejectNewTaskDemo.main(ThreadPoolExecutorRejectNewTaskDemo.java:88)
```

从 log 也可以看出, 正是第88行提交的第6个任务, 导致了异常的抛出. 当然, 抛异常也同样是这种情况下线程池默认的处理方式, 我们也可以改为使用线程池提  
他处理方式, 或者我们自己提供自定义的处理方式.

在上面的demo例子中, 我们对拒绝新任务的处理方式有了一个较为直观的认识, 我们知道了默认情况下线程池是使用抛异常的方式来拒绝新提交的任务的, 这  
常的方式在线程池中被称为 AbortPolicy. 当然, 除了这种 AbortPolicy 方式外, 线程池还为我们提供了 CallerRunsPolicy, DiscardPolicy和 DiscardOldestPolicy  
下面我们就来分别简要介绍下这几种方式:

① AbortPolicy:

前面已经介绍过, 这是一种直接抛异常的处理方式, 抛出 RejectedExecutionException 异常. 如果在 ThreadPoolExecutor 的构造方法中未指定 RejectedExecutionHandler 参数,  
线程池将使用他内部预定义的 defaultHandler 这个字段作为该参数的值, 而这个 defaultHandler 就是采用的 AbortPolicy 抛异常的方式 . 这也就解释了为什么在前边的demo例子  
程池在满足前边提到的两个条件中的任意一个时, 都会采取抛异常的方式来拒绝新提交的任务. 另外, 在Android开发中, 我们常用的 AsyncTask 类中也有个已定义好的线程池对象  
THREAD\_POOL\_EXECUTOR, 而这个线程池同样采用的是 AbortPolicy 抛异常的方式来拒绝新任务, 抛异常会导致我们的 Android APP 直接 crash 掉, 这是一种非常糟糕的用法.  
所以我们在Android开发中要使用 AsyncTask 结合线程池来并发处理异步任务, 如果并发执行的任务数较多的话, 建议不要直接使用 AsyncTask 内部自带的那个线程池, 而应该自  
一个线程池对象, 并为 RejectedExecutionHandler 参数赋予一个能够提供更友好处理方式的实现类对象.

② CallerRunsPolicy:

将新提交的任务放在 ThreadPoolExecutor.execute()方法所在的那个线程中执行.

③ DiscardPolicy:

直接不执行新提交的任务.

④ DiscardOldestPolicy:

为避免片面理解, 我们有必要看下他的源码:

```
1  /**
2   * A handler for rejected tasks that discards the oldest unhandled
3   * request and then retries {@code execute}, unless the executor
4   * is shut down, in which case the task is discarded.
5   */
6  public static class DiscardOldestPolicy implements RejectedExecutionHandler {
7      /**
8       * Creates a {@code DiscardOldestPolicy} for the given executor.
9       */
10     public DiscardOldestPolicy() { }
11
12     /**
13      * Obtains and ignores the next task that the executor
14      * would otherwise execute, if one is immediately available,
15      * and then retries execution of task r, unless the executor
16      * is shut down, in which case task r is instead discarded.
17      *
18      * @param r the runnable task requested to be executed
19      * @param e the executor attempting to execute this task
20      */
21     public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
22         if (!e.isShutdown()) {
23             e.getQueue().poll();
24             e.execute(r);
25         }
26     }
27 }
```

结合注释和源码可知, 这种处理方式分为两种情况:

- ① 当线程池已经关闭 (SHUTDOWN) 时, 就不执行这个任务了, 这也是 DiscardPolicy 的处理方式.
- ② 当线程池未关闭时, 会将阻塞队列中处于队首 (head) 的那个任务从队列中移除, 然后再将这个新提交的任务加入到该阻塞队列的队尾 (tail) 等待执行.

以上就是对这四种处理方式的简要介绍. 但是, 你可千万不要认为就只能在这四种方式中挑选一种, 因为这四种方式只是线程池预定义的四种常用方式, 而 RejectedExecutionHandler 这个参数其实是个接口, 线程池所提供的这四种方式其实都是该接口的实现类, 所以, 只要我们自定义一个类来实现该接口, 并在重写的 rejectedExecution() 方法时提供我们自己的处理逻辑, 那么我们就可以将我们自定义的这个类的对象作为参数传递给线程池的构造方法. 当线程池满足拒绝任务的条件时, 如果我们继续向其提交新任务, 那么线程池就会采用我们自己提供的那套逻辑来处理这些新提交的任务了.

### 3. 线程池进行任务调度的原理

我们只能向线程池提交任务, 而被提交的任务最终能否执行以及能否立即执行, 则都由线程池自己来控制, 至于怎么控制就涉及到线程池对任务调度的原理了. 向线程池提交一个任务, 可以通过调用 execute() 或 submit()方法来实现, 而二者的区别是, execute()方法只能进行任务的提交而不能获取该任务执行的结果, 而 submit()方法则既能进行任务的提交, 又能获取该任务执行的结果. 所以, 如果你需要获取一个任务执行的结果或者需要对一个任务执行的结果进行某种操作, 那么就要使用 submit()方法来提交任务. 其实 submit()方法就是对 execute()方法的一种封装, 它内部也是调用 execute()方法来实现任务提交的, 只是因为 submit()方法返回值是一个 Future 对象, 通过返回的 Future对象就能获取该任务最终执行的结果. 由于我们这里介绍的主题是线程池对任务调度的原理, 而任务调度, 用较为通俗的话来说, 就是一个任务被提交到线程池后能否被执行, 如果能被执行, 那么是立即执行, 还是在未来的某个时刻去执行, 用哪个线程执行; 如果不能被执行, 那么又该如何处理这个任务等等. 而这些既与前面介绍过的线程池中那[几个重要的参数](#)的设置有关, 还与任务被提交的时刻有关(准确来说, 就是与该任务被提交时, 线程池内已有的任务情况有关). 所以, 我们有必要从线程的提交开始分析, 由于 submit()方法内部也是调用 execute()方法, 所以我们就直接分析 execute()方法, 其源码如下:

```
1  /**
2   * Executes the given task sometime in the future.  The task
3   * may execute in a new thread or in an existing pooled thread.
4   *
5   * If the task cannot be submitted for execution, either because this
6   * executor has been shutdown or because its capacity has been reached,
7   * the task is handled by the current {@code RejectedExecutionHandler}.
8   *
9   * @param command the task to execute
10     * @throws RejectedExecutionException at discretion of
11     *         {@code RejectedExecutionHandler}, if the task
12     *         cannot be accepted for execution
13     * @throws NullPointerException if {@code command} is null
14     */
```



```
15 public void execute(Runnable command) {
16     if (command == null)
17         throw new NullPointerException();
18     /*
19     * Proceed in 3 steps:
20     *
21     * 1. If fewer than corePoolSize threads are running, try to
22     * start a new thread with the given command as its first
23     * task. The call to addWorker atomically checks runState and
24     * workerCount, and so prevents false alarms that would add
25     * threads when it shouldn't, by returning false.
26     *
27     * 2. If a task can be successfully queued, then we still need
28     * to double-check whether we should have added a thread
29     * (because existing ones died since last checking) or that
30     * the pool shut down since entry into this method. So we
31     * recheck state and if necessary roll back the enqueueing if
32     * stopped, or start a new thread if there are none.
33     *
34     * 3. If we cannot queue task, then we try to add a new
35     * thread. If it fails, we know we are shut down or saturated
36     * and so reject the task.
37     */
38
39     // 获取ctl的值.
40     int c = ctl.get();
41
42
43     /***** 情况1 *****/
44
45     // 根据ctl的值, 获取线程池中的有效线程数 workerCount, 如果 workerCount
46     // 小于核心线程数 corePoolSize
47     if (workerCountOf(c) < corePoolSize) {
48
49         // 调用addWorker()方法, 将核心线程数corePoolSize设置为线程池中线程
50         // 数的上限值, 将此次提交的任务command作为参数传递进去, 然后再次获取
51         // 线程池中的有效线程数 workerCount, 如果 workerCount依然小于核心
52         // 线程数 corePoolSize, 就创建并启动一个线程, 然后返回 true结束整个
53         // execute()方法. 如果此时的线程池已经关闭, 或者此时再次获取到的有
54         // 效线程数 workerCount已经 >= 核心线程数 corePoolSize, 就再继续执
55         // 行后边的内容.
56         if (addWorker(command, true))
57             return;
58
59         // 再次获取 ctl的值
60         c = ctl.get();
61     }
62
63     /**** 分析1 ****/
64     // 如果情况1的判断条件不满足, 则直接进入情况2. 如果情况1的判断条件满足,
65     // 但情况1中的 addWorker()方法返回 false, 也同样会进入情况2.
66     // 总之, 进入情况2时, 线程池要么已经不处于RUNNING(运行)状态, 要么仍处于RUNNING
67     // (运行)状态但线程池内的有效线程数 workerCount >= 核心线程数 corePoolSize
68
69
70     /***** 情况2 *****/
71
72     /**** 分析2 ****/
73     // 经过上一段分析可知, 进入这个情况时, 线程池要么已经不处于RUNNING(运行)
74     // 状态, 要么仍处于RUNNING(运行)状态但线程池内的有效线程数 workerCount
75     // 已经 >= 核心线程数 corePoolSize
76
77     // 如果线程池未处于RUNNING(运行)状态, 或者虽然处于RUNNING(运行)状态但线程池
78     // 内的阻塞队列 workQueue已满, 则跳过此情况直接进入情况3.
79     // 如果线程池处于RUNNING(运行)状态并且线程池内的阻塞队列 workQueue未滿,
80     // 则将提交的任务 command 添加到阻塞队列 workQueue中.
81     if (isRunning(c) && workQueue.offer(command)) {
82         // 再次获取 ctl的值.
83         int recheck = ctl.get();
84
85         // 再次判断线程池此时的运行状态. 如果发现线程池未处于 RUNNING(运行)
```

```
86 // 状态, 由于先前已将任务 command 加入到阻塞队列 workQueue 中了, 所以需
87 // 要将该任务从 workQueue 中移除. 一般来说, 该移除操作都能顺利进行.
88 // 所以一旦移除成功, 就再调用 handler 的 rejectedExecution() 方法, 根据
89 // 该 handler 定义的拒绝策略, 对该任务进行处理. 当然, 默认的拒绝策略是
90 // AbortPolicy, 也就是直接抛出 RejectedExecutionException 异常, 同时也
91 // 结束了整个 execute() 方法的执行.
92 if (! isRunning(recheck) && remove(command))
93     reject(command);
94
95 // 再次计算线程池内的有效线程数 workerCount, 一旦发现该数量变为0,
96 // 就将线程池内的线程数上限值设置为最大线程数 maximumPoolSize, 然后
97 // 只是创建一个线程而不去启动它, 并结束整个 execute() 方法的执行.
98 else if (workerCountOf(recheck) == 0)
99     addWorker(null, false);
100
101 // 如果线程池处于 RUNNING(运行) 状态并且线程池内的有效线程数大于0, 那么就结束该
102 // execute() 方法, 被添加到阻塞队列中的该任务将会在未来的某个时刻被执行.
103 }
104
105
106 /***** 情况3 *****/
107
108 /**** 分析3 ****/
109 // 如果该方法能够执行到这里, 那么结合分析1和分析2可知, 线程池此时必定是
110 // 下面两种情况中的一种:
111 // ① 已经不处于 RUNNING(运行) 状态
112 // ② 处于 RUNNING(运行) 状态, 并且线程池内的有效线程数 workerCount 已经
113 // >= 核心线程数 corePoolSize, 并且线程池内的阻塞队列 workQueue 已满
114
115 // 再次执行 addWorker() 方法, 将线程池内的线程数上限值设置为最大线程数
116 // maximumPoolSize, 并将提交的任务 command 作为被执行的对象, 尝试创建并
117 // 启动一个线程来执行该任务. 如果此时线程池的状态为如下两种中的一种,
118 // 就会触发 handler 的 rejectedExecution() 方法来拒绝该任务的执行:
119 // ① 未处于 RUNNING(运行) 状态.
120 // ② 处于 RUNNING(运行) 状态, 但线程池内的有效线程数已达到本次设定的最大
121 // 线程数 (另外根据分析3可知, 此时线程池内的阻塞队列 workQueue 已满).
122 //
123 // 如果线程池处于 RUNNING(运行) 状态, 但有效线程数还未达到本次设定的最大
124 // 线程数, 那么就会尝试创建并启动一个线程来执行任务 command. 如果线程的
125 // 创建和启动都很顺利, 那么就结束掉该 execute() 方法; 如果线程的创建或
```

代码中的注释非常详细, 这里再简要概括一下. execute()方法主要分为以下四种情况:

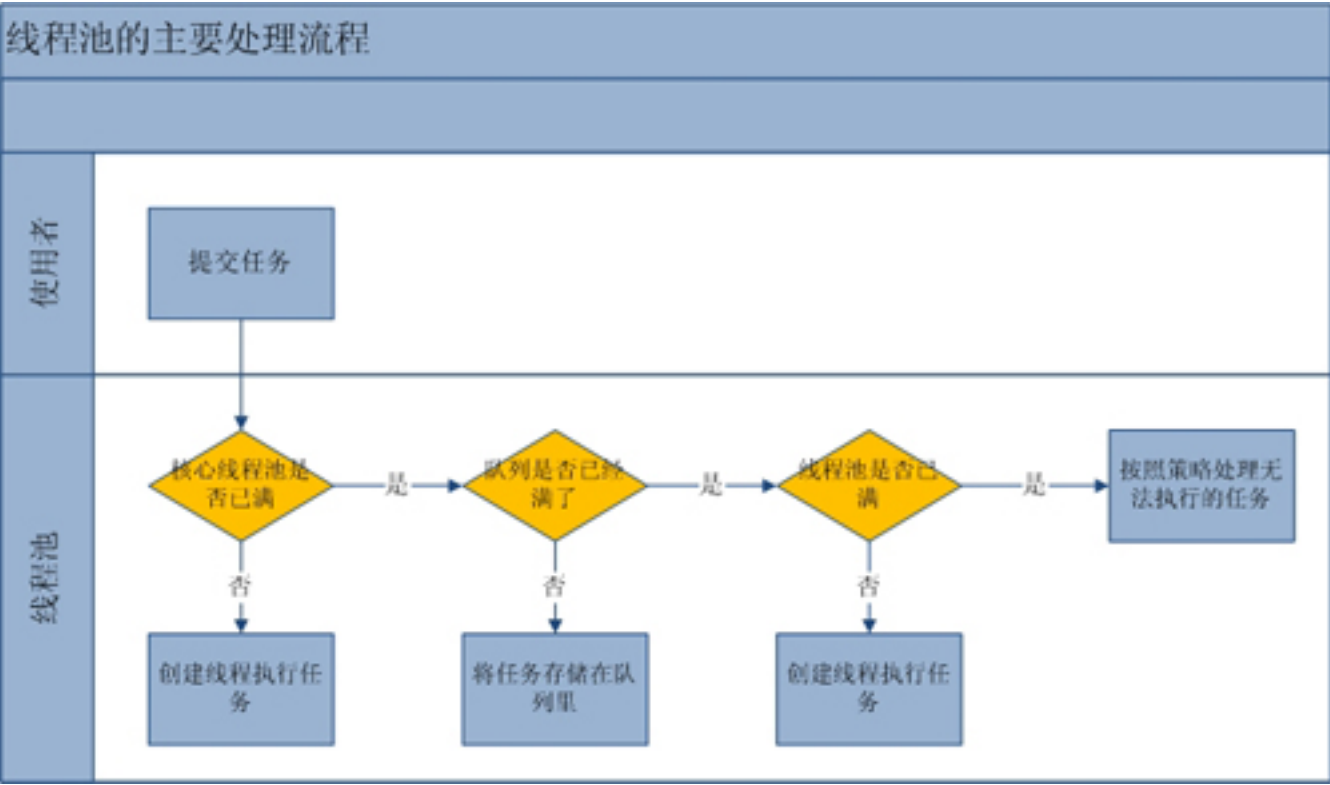
**情况1:** 如果线程池内的有效线程数少于核心线程数 corePoolSize, 那么就创建并启动一个线程来执行新提交的任务.

**情况2:** 如果线程池内的有效线程数达到了核心线程数 corePoolSize, 并且线程池内的阻塞队列未满, 那么就将新提交的任务加入到该阻塞队列中.

**情况3:** 如果线程池内的有效线程数达到了核心线程数 corePoolSize 但却小于最大线程数 maximumPoolSize, 并且线程池内的阻塞队列已满, 那么就创建并启动线程来执行新提交的任务.

**情况4:** 如果线程池内的有效线程数达到了最大线程数 maximumPoolSize, 并且线程池内的阻塞队列已满, 那么就让 RejectedExecutionHandler 根据它的拒绝处理该任务, 默认的处理方式是直接抛异常.

上述四种情况可以使用下面的流程图来描述 (这里就直接借用方腾飞老师在 [聊聊并发（三）——JAVA线程池的分析和使用](#) 这篇文章里的流程图. 若侵权):



上述 execute()源码在三种情况下分别调用了 addWorker()方法, 并且三次传递的参数都不同 (见第 56, 99, 128行):

```
1 addWorker(command, true);
```



```
2 addWorker(null, false);
3 addWorker(command, false)
```

所以, 要想彻底看懂 execute()方法的逻辑, 就必须要先大致了解 addWorker()方法的逻辑以及该方法分别在上述三组赋值情况下各自到底做了什么事情. 下面我来分析一下 addWorker()方法的源码吧:

```
1 /**
2  * Checks if a new worker can be added with respect to current
3  * pool state and the given bound (either core or maximum). If so,
4  * the worker count is adjusted accordingly, and, if possible, a
5  * new worker is created and started, running firstTask as its
6  * first task. This method returns false if the pool is stopped or
7  * eligible to shut down. It also returns false if the thread
8  * factory fails to create a thread when asked. If the thread
9  * creation fails, either due to the thread factory returning
10 * null, or due to an exception (typically OutOfMemoryError in
11 * Thread.start()), we roll back cleanly.
12 *
13 * @param firstTask the task the new thread should run first (or
14 * null if none). Workers are created with an initial first task
15 * (in method execute()) to bypass queuing when there are fewer
16 * than corePoolSize threads (in which case we always start one),
17 * or when the queue is full (in which case we must bypass queue).
18 * Initially idle threads are usually created via
19 * prestartCoreThread or to replace other dying workers.
20 *
21 * @param core if true use corePoolSize as bound, else
22 * maximumPoolSize. (A boolean indicator is used here rather than a
23 * value to ensure reads of fresh values after checking other pool
24 * state).
25 * @return true if successful
26 */
27 private boolean addWorker(Runnable firstTask, boolean core) {
28
29     // retry 是个无限循环. 当线程池处于 RUNNING (运行)状态时, 只有在线程池中
30     // 的有效线程数被成功加一以后, 才会退出该循环而去执行后边的代码. 也就是说,
31     // 当线程池在 RUNNING (运行)状态下退出该 retry 循环时, 线程池中的有效线程数
32     // 一定少于此次设定的最大线程数(可能是 corePoolSize 或 maximumPoolSize).
33     retry:
34     for (;;) {
35         int c = ctl.get();
36         int rs = runStateOf(c);
37
38         // 线程池满足如下条件中的任意一种时, 就会直接结束该方法, 并且返回 false
39         // 表示没有创建新线程, 新提交的任务也没有被执行.
40         // ① 处于 STOP, TYDING 或 TERMINATD 状态
41         // ② 处于 SHUTDOWN 状态, 并且参数 firstTask != null
42         // ③ 处于 SHUTDOWN 状态, 并且阻塞队列 workQueue为空
43
44         // Check if queue empty only if necessary.
45         if (rs >= SHUTDOWN &&
46             ! (rs == SHUTDOWN &&
47                 firstTask == null &&
48                 ! workQueue.isEmpty()))
49             return false;
50
51         for (;;) {
52             int wc = workerCountOf(c);
53
54             // 如果线程池内的有效线程数大于或等于了理论上的最大容量 CAPACITY 或者实际
55             // 设定的最大容量, 就返回 false直接结束该方法. 这样同样没有创建新线程,
56             // 新提交的任务也同样未被执行.
57             // (core ? corePoolSize : maximumPoolSize) 表示如果 core为 true,
58             // 那么实际设定的最大容量为 corePoolSize, 反之则为 maximumPoolSize.
59             if (wc >= CAPACITY ||
60                 wc >= (core ? corePoolSize : maximumPoolSize))
61                 return false;
62
63             // 有效线程数加一
64             if (compareAndIncrementWorkerCount(c))
```

```
65         break retry;
66         c = ctl.get();    // Re-read ctl
67         if (runStateOf(c) != rs)
68             continue retry;
69         // else CAS failed due to workerCount change; retry inner loop
70     }
71 }
72
73 boolean workerStarted = false;
74 boolean workerAdded = false;
75 Worker w = null;
76 try {
77     // 根据参数 firstTask来创建 Worker对象 w.
78     w = new Worker(firstTask);
79     // 用 w创建线程对象 t.
80     final Thread t = w.thread;
81     if (t != null) {
82         final ReentrantLock mainLock = this.mainLock;
83         mainLock.lock();
84         try {
85             // Recheck while holding lock.
86             // Back out on ThreadFactory failure or if
87             // shut down before lock acquired.
88             int rs = runStateOf(ctl.get());
89
90             if (rs < SHUTDOWN ||
91                 (rs == SHUTDOWN && firstTask == null)) {
92                 if (t.isAlive()) // precheck that t is startable
93                     throw new IllegalStateException();
94                 workers.add(w);
95                 int s = workers.size();
96                 if (s > largestPoolSize)
97                     largestPoolSize = s;
98                 workerAdded = true;
99             }
100         } finally {
101             mainLock.unlock();
102         }
103         if (workerAdded) {
104
105             // 启动线程 t. 由于 t指向 w.thread所引用的对象, 所以相当于启动的是 w.thread所引用的线程对象.
106             // 而 w是 Runnable 的实现类, w.thread 是以 w作为 Runnable参数所创建的一个线程对象, 所以启动
107             // w.thread所引用的线程对象, 也就是要执行 w 的 run()方法.
108             t.start();
109             workerStarted = true;
110         }
111     }
112 } finally {
113     if (! workerStarted)
```

这里我们只分析线程池处于 RUNNING(运行)状态时的情况, 并且只分析上述方法中核心的重要的代码, 至于其他情况以及其他行的代码, 就不分析了.

先看第78和80行, 根据参数 firstTask来创建 Worker对象 w, 并用 w再创建线程对象 t, t 指向 w.thread所指向的线程对象. 既然这里提到了 Worker 类, 那么我们要介绍一下这个类. Worker是 ThreadPoolExecutor类的一个内部类, 同时也是 Runnable接口的实现类, 其源码如下:

```
1  /**
2   * Class Worker mainly maintains interrupt control state for
3   * threads running tasks, along with other minor bookkeeping.
4   * This class opportunistically extends AbstractQueuedSynchronizer
5   * to simplify acquiring and releasing a lock surrounding each
6   * task execution. This protects against interrupts that are
7   * intended to wake up a worker thread waiting for a task from
8   * instead interrupting a task being run. We implement a simple
9   * non-reentrant mutual exclusion lock rather than use
10  * ReentrantLock because we do not want worker tasks to be able to
11  * reacquire the lock when they invoke pool control methods like
12  * setCorePoolSize. Additionally, to suppress interrupts until
13  * the thread actually starts running tasks, we initialize lock
14  * state to a negative value, and clear it upon start (in
15  * runWorker).
```



```
16 */
17 private final class Worker
18     extends AbstractQueuedSynchronizer
19     implements Runnable
20 {
21     // ...
22
23     /** Thread this worker is running in. Null if factory fails. */
24     final Thread thread;
25     /** Initial task to run. Possibly null. */
26     Runnable firstTask;
27
28     /**
29     * Creates with given first task and thread from ThreadFactory.
30     * @param firstTask the first task (null if none)
31     */
32     Worker(Runnable firstTask) {
33         setState(-1); // inhibit interrupts until runWorker
34         this.firstTask = firstTask;
35         this.thread = getThreadFactory().newThread(this);
36     }
37
38     /** Delegates main run loop to outer runWorker. */
39     public void run() {
40         runWorker(this);
41     }
42
43     "
```

这里我只贴出了 Worker类中和本次分析有关的代码, 其他代码就省略掉了. Worker类内部还包含一个线程对象 thread 和一个 Runnable对象 firstTask. 而这个对象的创建过程见35行, 将Worker对象自身作为这个线程对象 thread的 Runnable参数传递给 thread的构造方法. 那么我们就可以知道, 如果要运行该线程, 也就是thread.start(), 那么实际上就是要执行 thread所在的 Worker类中的 run()方法, 而从第40行又可以知道, Worker类中的 run()方法又是调用 runWorker(this); 这个的, 并将 thread所在的 Worker对象作为这个 runWorker()方法的参数. 简单来说就是:

启动一个 Worker对象中包含的线程 thread, 就相当于要执行 runWorker()方法, 并将该 Worker对象作为该方法的参数.

我们对 Worker类的分析就到此为止吧, 我们再回到 addWorker()方法继续来分析. 我们看 addWorker() 方法的第108行, 启动了线程 t, 而从第80行我们可以知道 t 就是 w.thread 所指向的对象, 所以第108行就相当于启动了 w.thread 这个线程, 也就是启动了 Worker对象 w 中的线程对象 thread. 而我们在分析 Worker类时出过这样的结论: 启动一个 Worker对象中包含的线程 thread, 就相当于要执行 runWorker()方法, 并将该 Worker对象作为该方法的参数. (结论见[这里](#)) 所以, 启动线程 w.thread, 也就相当于要执行 runWorker(w)方法. 我们再来看一下 runWorker()方法的源码吧:

```
1 /**
2  * Main worker run loop. Repeatedly gets tasks from queue and
3  * executes them, while coping with a number of issues:
4  *
5  * 1. We may start out with an initial task, in which case we
6  * don't need to get the first one. Otherwise, as long as pool is
7  * running, we get tasks from getTask. If it returns null then the
8  * worker exits due to changed pool state or configuration
9  * parameters. Other exits result from exception throws in
10 * external code, in which case completedAbruptly holds, which
11 * usually leads processWorkerExit to replace this thread.
12 *
13 * 2. Before running any task, the lock is acquired to prevent
14 * other pool interrupts while the task is executing, and then we
15 * ensure that unless pool is stopping, this thread does not have
16 * its interrupt set.
17 *
18 * 3. Each task run is preceded by a call to beforeExecute, which
19 * might throw an exception, in which case we cause thread to die
20 * (breaking loop with completedAbruptly true) without processing
21 * the task.
22 *
23 * 4. Assuming beforeExecute completes normally, we run the task,
24 * gathering any of its thrown exceptions to send to afterExecute.
25 * We separately handle RuntimeException, Error (both of which the
26 * specs guarantee that we trap) and arbitrary Throwables.
27 * Because we cannot rethrow Throwables within Runnable.run, we
```

```
28 * wrap them within Errors on the way out (to the thread's
29 * UncaughtExceptionHandler). Any thrown exception also
30 * conservatively causes thread to die.
31 *
32 * 5. After task.run completes, we call afterExecute, which may
33 * also throw an exception, which will also cause thread to
34 * die. According to JLS Sec 14.20, this exception is the one that
35 * will be in effect even if task.run throws.
36 *
37 * The net effect of the exception mechanics is that afterExecute
38 * and the thread's UncaughtExceptionHandler have as accurate
39 * information as we can provide about any problems encountered by
40 * user code.
41 *
42 * @param w the worker
43 */
44 final void runWorker(Worker w) {
45     Thread wt = Thread.currentThread();
46     Runnable task = w.firstTask;
47     w.firstTask = null;
48     w.unlock(); // allow interrupts
49     boolean completedAbruptly = true;
50     try {
51         // 由前边可知, task 就是 w.firstTask
52         // 如果 task 为 null, 那么就不进入该 while 循环, 也就不运行该 task. 如果
53         // task 不为 null, 那么就执行 getTask() 方法. 而 getTask() 方法是个无限
54         // 循环, 会从阻塞队列 workQueue 中不断取出任务来执行. 当阻塞队列 workQueue
55         // 中所有的任务都被取完之后, 就结束下面的 while 循环.
56         while (task != null || (task = getTask()) != null) {
57             w.lock();
58             // If pool is stopping, ensure thread is interrupted;
59             // if not, ensure thread is not interrupted. This
60             // requires a recheck in second case to deal with
61             // shutdownNow race while clearing interrupt
62             if ((runStateAtLeast(ctl.get(), STOP) ||
63                 (Thread.interrupted() &&
64                  runStateAtLeast(ctl.get(), STOP))) &&
65                 !wt.isInterrupted())
66                 wt.interrupt();
67             try {
68                 beforeExecute(wt, task);
69                 Throwable thrown = null;
70                 try {
71                     // 执行从阻塞队列 workQueue 中取出的任务.
72                     task.run();
73                 } catch (RuntimeException x) {
74                     thrown = x; throw x;
75                 } catch (Error x) {
76                     thrown = x; throw x;
77                 } catch (Throwable x) {
78                     thrown = x; throw new Error(x);
79                 } finally {
80                     afterExecute(task, thrown);
81                 }
82             } finally {
83                 // 将 task 置为 null, 这样使得 while 循环是否继续执行的判断, 就只能依赖于判断
84                 // 第二个条件, 也就是 (task = getTask()) != null 这个条件, 是否满足.
85                 task = null;
86                 w.completedTasks++;
87                 w.unlock();
88             }
89         }
90         completedAbruptly = false;
```

第46行, 将 w.firstTask 赋值给局部变量 task.

第56行, 这一行分为下面两部分内容来介绍:

(1)

如果 task 为 null (也就是 w.firstTask 为 null), 那么就不进入 while 循环, 也就不运行 task.

还记得在分析 [execute\(\)方法的源码](#) 时, 我们提到过, 该方法会在三种情况下分别调用 addWorker()方法吗? 只是这三次调用时, 分别传入的参数都不同, 而该方



99行代码以及我们对该代码的注释如下:

```
1 // 再次计算线程池内的有效线程数 workerCount, 一旦发现该数量变为0,
2 // 就将线程池内的线程数上限值设置为最大线程数 maximumPoolSize, 然后
3 // 只是创建一个线程而不去启动它, 并结束整个 execute()方法的执行.
4 else if (workerCountOf(recheck) == 0)
5     addWorker(null, false);
```

我们在注释中写到”一旦发现该数量变为0, 就只是创建一个线程而不去启动它”, 而在分析过 [Worker类](#) 以及 [addWorker\(\)](#) 和 [runWorker\(\)](#) 方法的源码后, 这句注释就容易理解了. 下面我们就来分析一下 addWorker(null, false); 这句代码的含义吧. 这句代码为参数 firstTask 赋值为 null, 为参数 core 赋值为 false. 回到 [addWorker方法的源码](#), 第78行有如下代码:

```
1 w = new Worker(firstTask);
```

我们传递的 firstTask 为 null, 再看 [Worker类的源码](#), 第32~36行是其构造方法, 在构造方法中, 将值为 null 的参数 firstTask 传给该Worker类内部的 Runnable字 firstTask (也就是设置 w.firstTask = null), 而在 [addWorker\(\)方法](#) 的第108行又去运行线程t, 也就是运行线程 w.thread, 即: 启动 Worker 对象 w 中的线程 thread. 们又在前边总结过一个结论: **启动一个 Worker对象中包含的线程 thread, 就相当于要执行 runWorker()方法, 并将该 Worker对象作为该方法的参数.** (见[这里](#)) 启动线程 w.thread, 就相当于执行 runWorker(w)方法, 而我们还需要记住, 此时的 w.firstTask 是 null 的, 所以再回到 [runWorker\(\)方法的源码](#), 见第46行, 我们将 null 的 w.firstTask 传递给局部变量 task, 这样 task 也为 null. 这样第56行 while循环的 task != null 判断条件就为 false, 从而直接跳过该 while 循环, 也就跳过了 while 循环中第72行 task.run(); 这句启动线程的代码, 所以说, addWorker(null, false); 这句代码的含义是”只是创建一个线程而不去启动它”.

(2)  
我们继续回到 runWorker() 方法的第56行. 我们在 (1) 中分析了 w.firstTask 为 null 导致直接 while()循环直接不执行的情况. 而如果 w.firstTask 不为 null (也就是我们提交的任务不为 null). 那么在执行到 runWorker() 方法的第56行时, 由于满足了第一个条件 task != null, 所以会进入到 while 循环中, 在72行会去执行我们提交的任务. 然后在第85行, 将 task 置为 null, 使得下次循环开始时, 也就是再次来到该方法的第56行时, 由于第一个条件 task != null 为 false, 使得我们必须去判断第二个条件 (task = getTask()) != null 是否成立. 如果 getTask() 方法的返回值不为 null, 那么该方法会将其返回值赋值给 task, 然后再次进入 while 循环, 然后还是在第72行去执行 task.run(). 如果 getTask() 方法返回 null, 那么该方法会将其返回值赋值给 task, 然后再次进入 while 循环, 然后还是在第72行去执行 task.run(). 这样一直循环下去, 直到 getTask() 方法返回 null, 才会结束 while循环, 然后结束掉 runWorker() 方法. 那么, getTask() 方法到底是用来做什么的呢? 其实, 根据该方法的名字, 想必你应该已经猜到了吧, “获取任务”, 这个猜测对不对呢? 我们还是来看看它的源码吧, 如下所示:

```
1 /**
2  * Performs blocking or timed wait for a task, depending on
3  * current configuration settings, or returns null if this worker
4  * must exit because of any of:
5  * 1. There are more than maximumPoolSize workers (due to
6  *    a call to setMaximumPoolSize).
7  * 2. The pool is stopped.
8  * 3. The pool is shutdown and the queue is empty.
9  * 4. This worker timed out waiting for a task, and timed-out
10 *    workers are subject to termination (that is,
11 *    {@code allowCoreThreadTimeOut || workerCount > corePoolSize})
12 *    both before and after the timed wait, and if the queue is
13 *    non-empty, this worker is not the last thread in the pool.
14 *
15 * @return task, or null if the worker must exit, in which case
16 *         workerCount is decremented
17 */
18 private Runnable getTask() {
19     boolean timedOut = false; // Did the last poll() time out?
20
21     // 无限循环.
22     for (;;) {
23         int c = ctl.get();
24         int rs = runStateOf(c);
25
26         // 如果线程池已停止, 或者线程池被关闭并且线程池内的阻塞队列为空, 则结束该方法并返回 null.
27         // Check if queue empty only if necessary.
28         if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
29             decrementWorkerCount();
30             return null;
31         }
32
33         int wc = workerCountOf(c);
34
35         // 如果 allowCoreThreadTimeOut 这个字段设置为 true(也就是允许核心线程受超时机制的控制), 则
```

```
37 // 直接设置 timed 为 true. 反之, 则再看当前线程池中的有效线程数是否已经超过了核心线程数, 也
38 // 就是是否存在非核心线程. 如果存在非核心线程, 那么也会设置 timed 为true.
39 // 如果 wc <= corePoolSize (线程池中的有效线程数少于核心线程数, 即: 线程池内运行着的都是核心线程),
40 // 并且 allowCoreThreadTimeOut 为 false(即: 核心线程即使空闲, 也不会受超时机制的限制),
41 // 那么就设置 timed 为 false.
42 // Are workers subject to culling?
43 boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
44
45 // 当线程池处于 RUNNING (运行)状态但阻塞队列内已经没有任务(为空)时, 将导致有线程接下来会一直
46 // 处于空闲状态. 如果空闲的是核心线程并且设置核心线程不受超时机制的影响(默认情况下就是这个设置),
47 // 那么这些核心线程将一直在线程池中处于空闲状态, 等待着新任务的到来, 只要线程池处于 RUNNING
48 // (运行)状态, 那么, 这些空闲的核心线程将一直在池中而不会被销毁. 如果空闲的是非核心线程, 或者
49 // 虽然是核心线程但是设置了核心线程受超时机制的限制, 那么当空闲达到超时时间时, 这就满足了这里的
50 // if条件而去执行 if内部的代码, 通过返回 null 结束掉该 getTask()方法, 也最终结束掉 runWorker()方法.
51 if ((wc > maximumPoolSize || (timed && timedOut))
52     && (wc > 1 || workQueue.isEmpty())) {
53     if (compareAndDecrementWorkerCount(c))
54         return null;
55     continue;
56 }
57
58 try {
59     // 从阻塞队列中取出队首的那个任务, 设置给 r. 如果空闲线程等待超时或者该队列已经为空, 则 r为 null.
60     Runnable r = timed ?
61         workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
62         workQueue.take();
63
64     // 如果阻塞队列不为空并且未发生超时的情况, 那么取出的任务就不为 null, 就直接返回该任务对象.
65     if (r != null)
66         return r;
67     timedOut = true;
68 } catch (InterruptedException retry) {
69     timedOut = false;
```

getTask()方法内有个无限循环, 要想结束一个无限循环, 要么用 break 语句, 要么用 return 语句. 而观察上述源码可以发现, 也只有第30, 54, 66行满足结束循环. 三个return语句, 我们接下来会分别进行分析.


先看第30行的 return null; 语句, 这是当线程池处于关闭或者停止时, 如果满足其他条件, 就会结束该循环, 并返回 null, 表示线程池已关闭且阻塞队列已空, 或者已停止, 这两种情况下都没有或无法获取要处理的任务了. 再看第60~62行, 尝试从阻塞队列中取出位于队首的那个任务, 如果发生了线程等待超时, 或者执行取之前, 阻塞队列已经空了, 那么任务 r 就为 null; 如果队列中还有任务并且未发生超时, 那么 r 就不为 null. 第65~66行, 如果 r 不为 null, 就表示此次取出的确实是实实在在的任务对象, 于是就将该任务对象返回并结束该无限循环和该方法. 当然, 我们终究是会遇到执行取出动作的时候阻塞队列已经为空的情况, 那么取出的 null, 这样就不满足第65行的条件, 也就不会执行第66行的 return 语句, 这样又会来到第22行, 再次执行这个无限循环. 如果检测到超时条件满足了, 由于此时阻塞队列为空, 那么这就满足了第51~52行的条件, 就会执行第54行的 return null; 语句, 表示已经没有可以取出的任务了, 所以返回 null, 并结束该无限循环也结束该方法.

总之, 如果有任务可以取出的话, getTask()方法就会返回一个具体的任务对象; 如果线程池被关闭或停止, 或者阻塞队列在相当长的一段时间 (超过超时时间) 内没有任务可供取出, 那么就会返回 null, 表示没有任务了或者无法再获取任务了, 那么 [runWorker\(\) 方法](#) 第56行的判断条件 (task = getTask()) != null 就为 false, 早退出 while 循环, 最终结束该方法.

以上就是线程池对任务调度的大致原理. 至此, 这篇文章也就结束了. 感谢各位有耐心的读者能够读完这篇相对较长也较为枯燥的文章, 也希望大家能够多提宝贵建议或意见, 互相交流学习!

参考资料:

- 1. Oracle JDK 1.8 源码
- 2. [知乎—Android中的Thread与AsyncTask的区别? —肥肥鱼的回答](#)
- 3. [聊聊并发（三）——JAVA线程池的分析和使用](#)
- 4. 《Android开发艺术探索》第11章 Android的线程和线程池



### 码农不会英语怎么行？英语文档都看不懂！

不背单词和语法，一个公式教你读懂天下英文→



想对作者说点什么



volvoxc：“这样第56行 while循环的 task != null 判断条件就为 false, 从而直接跳过该 while 循环” task != null 判断条件就为 false 还需要再判断 (task = getTask()) != null 的吧？

个条件之间用的是“||”，不是“&&”，这里逻辑是不是弄错了？

（2个月前

#16楼）

[查看回复\(1\)](#)



- 

中华丛迅： 写的真是好，服了服了 (7个月前 #15楼)
- 

大多多： 牛逼，真牛逼，，， (8个月前 #14楼)
- 

search\_forever： 很有收获，感谢博主的分享。 (9个月前 #13楼)
- 

开心1002： 自己看完源码有些东西还一知半解，看了博主的博客，如同打通了任督二脉，感激不尽 (1年前 #12楼)
- 

yangpeng686： 想问博主，workqueue定义第三条最后一句，如果发现运行的线程数小于池子允许的最大线程数, 那么就会创建一个线程(这里创建的线程是非核心线程)来执...  
务. 此时这个新创建的线程搁哪执行，队列已经满了呢 (1年前 #11楼)
- 

yangpeng686： 想问博主，workqueue定义第三条最后一句，如果发现运行的线程数小于池子允许的最大线程数, 那么就会创建一个线程(这里创建的线程是非核心线程)来执...  
务. 此时这个新创建的线程搁哪执行，队列已经满了呢 (1年前 #10楼)
- 

yangpeng686： 想问博主，workqueue定义第三条最后一句，如果发现运行的线程数小于池子允许的最大线程数, 那么就会创建一个线程(这里创建的线程是非核心线程)来执...  
务. 此时这个新创建的线程搁哪执行，队列已经满了呢 (1年前 #9楼)
- 

ss3ss： 衷心感谢楼主的分享！期待作者更好的文章！ (1年前 #8楼)
- 

dengzhuopeng8023： 为什么 getTask 获取不到任务的时候，会去 remove 当前的work对象呢 (1年前 #7楼)
- 

dengzhuopeng8023： 111 (1年前 #6楼)
- 

Aron2001\_： // 由前边可知, task 就是 w.firstTask // 如果 task为 null, 那么就不进入该 while循环, 也就不运行该 task. 如果 // task不为 null, 那么就执行 getTask()方法. 而getTa...  
法是个无限 // 循环, 会从阻塞队列 workQueue中不断取出任务来执行. 当阻塞队列 workQueue // 中所有的任务都被取完之后, 就结束下面的while循环. while (task != null || (tas...  
getTask()) != null) { 哈哈，错了吧 (1年前 #5楼) [查看回复\(1\)](#)
- 

rudy\_yuan： 您好，请问为啥在addThread中要通过works.add(w)将新建的work对象加入到缓存队列中呢？这个新建的work不是马上就会在thread.start时被执行吗？放到队列...  
目的是什么？谢谢。 (2年前 #4楼)
- 

skxy： 注释写得很详细，源码阅读能力很给力！ (2年前 #3楼)
- 

-非子墨-： 写的很详细~受教了~ (2年前 #2楼)

Java线程池源码分析（一）

阅读数 801

Java线程池源码分析 使用线程池场景，好处，不在本文范围内，我们分析的是源码... 来自：[mayongzhan\\_c...](#)

Java线程池实现原理与源码解析(jdk1.8)


阅读数 4136

为什么需要线程池？ 线程池能够对线程进行统一分配，调优和监控： - 降低资源消... 来自：[programmer\\_at...](#)

JAVA 线程池源码分析

阅读数 336

java5之后为我们提供了线程池，只需要使用API，不用去考虑线程池里特殊的处理机... 来自：[小程故事多的博客](#)



茅台镇26岁美女大学生，曝光白酒行业“丑闻”，常喝酒的看一看

钥翔 · 鸛鸛

温州泳恒科技有限公司

泳恒科技 · 顶新

Java线程池源码解析

阅读数 108

Java线程池ThreadPoolExecutor继承了AbstractExecutorService，间接实现了Execu... 来自：[juvenfan的专栏](#)

java源码分析系列— 线程池Executors

阅读数 2348

用了线程池已经有一段时间了,以前只是偶尔看看源码,了解了其中调度策略,没有深入... 来自：[常见的专栏](#)

Java线程池及其底层源码实现分析

阅读数 1046

相关类： Executors ExecutorService Callable ThreadPool Future 接口： Executor ... 来自：[两只猴子的博客](#)

Java多线程-线程池ThreadPoolExecutor构造方法和规则

阅读数 8.6万

为什么用线程池博客地址 [http://blog.csdn.net/qq\\_25806863](http://blog.csdn.net/qq_25806863)原文地址 [http://blog.csdn.net/qq\\_25806863](http://blog.csdn.net/qq_25806863) 来自：[喵了个鸣的博客](#)

ThreadPoolExecutor源码分析

阅读数 1998

一、线程和任务的概念首先区分概念，任务和线程。可以简单理解为任务为Runnabl... 来自：[xingfeng\\_coder...](#)



茅台镇26岁美女大学生，曝光白酒行业“丑闻”，常喝酒的看一看

广告 白酒 白酒



年轻人都说好！酸化缓蚀剂驭腾实业专业供应咨询热线：一...

陕西驭腾实业 · 顶新

## Java四种线程池介绍

阅读数 1.2万

Java四种线程池

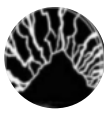
来自：[yunan0808的专栏](#)

## java4种线程池的使用

阅读数 5038

Java通过Executors提供四种线程池，分别为： newCachedThreadPool创建一个可...

来自：[hanshangzhi的...](#)



chichenzhe

1篇文章

排名:千里之外

关注



挤不上公交车的路人甲

33篇文章

排名:千里之外

关注



bice1222

0篇文章

排名:千里之外

关注



jjwkl2000

1篇文章

排名:千里之外

关注

## ThreadPoolExecutor解析-主要源码研究

阅读数 8983

ThreadPoolExecutor解析-主要源码研究

来自：[智公博客的专栏](#)

## 深入源码分析Java线程池的实现原理

阅读数 40

转载自：<https://mp.weixin.qq.com/s/-89-CcDnSLBYy3THmcLEdQ> 程序的运行，其...

来自：[johnhuster的专栏](#)

## 超详细的java线程池源码解读

阅读数 2.1万

线程池的继承关系是这样的ThreadPoolExecutor继承了AbstractExecutorService，A...

来自：[IT农场](#)



这变态传奇你卸载算我输！爆率9.8，有充值入口我跪键盘！

贪玩游戏 · 顶新



本周台中市本地地区好消息：昌盛专业候车亭广告灯箱制作商

昌盛候车亭 · 顶新

## Java线程池之ThreadPoolExecutor源码分析

阅读数 86

一、引言 Java并发工具包自带了很多常用的线程池，程序可以将定义的Runnable、...

来自：[APlus](#)

## Java线程池源码分析(基于JDK1.8)

阅读数 239

前言线程是稀缺资源，如果被无限制的创建，不仅会消耗系统资源，还会降低系统的...

来自：[ting说你跳?](#)

## java线程池执行有返回值线程源码详解

阅读数 119

java线程池提供了几种执行线程的方式，这里主要讨论关于执行有返回值的线程的实...

来自：[诺浅的专栏](#)

## ThreadPoolExecutor源码剖析

阅读数 527

线程池是Doug lea大神写的concurrent包中应用最广泛的一个框架，通常搭配阻塞队...

来自：[Niulx](#)

## 线程池为什么能维持线程不释放，随时运行各种任务？

阅读数 400

版权声明：本文为博主原创文章，未经博主允许不得转载。技术交流可邮:cjh94520...

来自：[varyall的专栏](#)



50万码农评论：英语对于程序员有多重要！

不背单词和语法，老司机教你一个数学公式秒懂天下英语

## 线程池中的线程为什么不会释放而是循环等待任务呢

阅读数 514

线程池 之前一直有这个疑问：我们平时使用线程都是各种new Thread(),然后直接在r...

来自：[nwpu\\_geeker的...](#)

## ThreadPoolExecutor 是如何做到线程重用的

阅读数 201

前言： 看关于ThreadPoolExecutor参数时，看到了keepaliveTime这个参数，这个参...

来自：[大树叶 技术专栏](#)

## Java线程池ThreadPoolExecutor使用和分析(二) - execute()原理

阅读数 630

相关文章目录：[Java线程池ThreadPoolExecutor使用和分析\(一\)](#) [Java线程池Th...](#)

来自：[lkj41110的博客](#)

## spring线程池ThreadPoolExecutor配置以及FutureTask的使用

12-14

最代码，<http://www.zuidaima.com/share/1724478138158080.htm> 的代码及例子



<div><div><div></div><div>下载</div></div><div>线程池源码分析</div></div>	08-06
线程池ThreadPoolExecutor的源码分析，含中文注释，深入了解线程池的构造	
<div><div><div></div><div>码农不会英语怎么行？英语文档都看不懂！ 不背单词和语法，一个公式教你读懂天下英文→</div></div></div>	
<div><div><div><div></div><div>下载</div></div><div>ThreadPoolExecutor的使用和Android常见的4种线程池使用介绍</div></div></div> <div>ThreadPoolExecutor的使用和Android常见的4种线程池使用介绍</div>	10-11
<div><div><div><div></div><div>下载</div></div><div>Java线程池使用说明</div></div></div> <div>Java线程池使用说明：一 简介 二：线程池 三：ThreadPoolExecutor详解</div>	07-30
<div><div><div><div></div><div>下载</div></div><div>java 线程池例子ThreadPoolExecutor</div></div></div> <div>一个关于java 线程池的例子，也适合android</div>	04-29
<div><div><div></div><div>java六种线程池</div></div></div> <div>六大线程池 本文讲述之前我们提到的Executors类(注意加了s的)中的六个静态方法，... 来自： qq_26963495的...</div>	阅读数 7558
<div><div><div></div><div>Java ThreadPoolExecutor线程池原理及源码分析</div></div></div> <div>一、源码分析（基于JDK1.6） ThreadExecutorPool是使用最多的线程池组件，了解... 来自： Schelor的专栏</div>	阅读数 2125
<div><div><div></div><div>50万码农评论：英语对于程序员有多重要！ 不背单词和语法，老司机教你一个数学公式秒懂天下英语</div></div></div>	
<div><div><div></div><div>线程池ThreadPoolExecutor源码解析</div></div></div> <div>最近将ThreadPoolExecutor源码又读了一遍，将以前没有弄的太懂的地方给弄懂了... 来自： 屌丝程序员的奋...</div>	阅读数 812
<div><div><div></div><div>java中的线程池实现以及代码分析</div></div></div> <div>为什么要使用线程池: 我的理解是线程池可以使线程复用，避免了每次线程都new一... 来自： XXXAndroidXXX</div>	阅读数 1214
<div><div><div></div><div>Java线程池源码解析及高质量代码案例</div></div></div> <div>ThreadPoolExecutor是一个 ExecutorService，它使用可能的几个池线程之一执行每... 来自： XingLiu's Blog</div>	阅读数 5578
<div><div><div><div></div><div>下载</div></div><div>ControllableThreadPoolExecutor</div></div></div> <div>ThreadPoolExecutor线程池源码</div>	05-14
<div><div><div></div><div>ThreadPoolExecutor线程池</div></div></div> <div>ThreadPoolExecutor.execute(Runnable) 之后 这个 Runnable 会自己释放吗?rnrn我所实现的 Runnable 的run方...</div>	
<div><div><div></div><div>50万码农评论：英语对于程序员有多重要！ 不背单词和语法，老司机教你一个数学公式秒懂天下英语</div></div></div>	
<div><div><div><div></div><div>下载</div></div><div>JDK1.5中的线程池</div></div></div> <div>JDK1.5中的线程池(ThreadPoolExecutor)使用简介</div>	08-16
<div><div><div><div></div><div>下载</div></div><div>JDK1[1].5中的线程池(ThreadPoolExecutor)使用简介</div></div></div> <div>JDK1[1].5中的线程池(ThreadPoolExecutor)使用简介</div>	08-30
<div><div><div></div><div>java程序员必精--从源码讲解java线程池ThreadPoolExecuter的实现原...</div></div></div> <div>类结构图 示例 自带线程池的各种坑 基础参数 源码分析java.util.concurrent.ThreadP... 来自： zqz_zqz的博客</div>	阅读数 6872
<div><div><div></div><div>Android中的线程和线程池及其源码分析：</div></div></div> <div>一.基本的知识点: #线程: 什么是线程: 线程的几种状态 实现方式和区别: ##什么... 来自： kunkun5love的...</div>	阅读数 1386
<div><div><div></div><div>JAVA线程池原理以及几种线程池类型介绍</div></div></div> <div>在什么情况下使用线程池？ 1.单个任务处理的时间比较短 2.将需处理的任务的... 来自： GarfieldEr007的...</div>	阅读数 1618
<div><div><div></div><div>50万码农评论：英语对于程序员有多重要！ 不背单词和语法，老司机教你一个数学公式秒懂天下英语</div></div></div>	

	• <b>Java线程池带图详解</b>		阅读数 814
	线程池作为Java中一个重要的知识点，看了很多文章，在此以Java自带的线程池为...		来自： <a href="#">刘成</a>
	• <b>线程池代码</b>		阅读数 398
	#ifndef __THREADPOOL_H_ #define __THREADPOOL_H_ typedef struct threadp...		来自： <a href="#">HicerWu的博客</a>
	• <div>下载</div> <b>Android线程池管理的代码例子</b>		05-30
	Android线程池管理的代码例子。用于演示普通线程池ThreadPoolExecutor、定时器线程池ScheduledExecutorService等功能。		
	• <div>下载</div> <b>Java线程池文档</b>		03-17
	Java 线程池学习 Reference: 《创建Java线程池》[1], 《Java线程：新特征-线程池》[2], 《Java线程池学习》[3], 《线程池ThreadPoolExecutor使用简介》[4], 《Java5中的线程...		
	• <b>线程池ThreadPoolExecutor的先进先出问题</b>		
	[code=Java]rn//1rn threadPool = new ThreadPoolExecutor(poolMinSize, poolMaxSize, keepAliveTime,rn Time...		
	• <div>下载</div> <b>线程池： java_ThreadPoolExecutor.mht</b>		11-20
	(转)线程池： java_util_ThreadPoolExecutor 比较详细的介绍了ThreadPoolExecutor用法与属性		
	• <div>下载</div> <b>线程池并发测试例子</b>		12-19
	java代码 ThreadPoolExecutor线程池并发测试例子如有误欢迎指正		
	• <b>Java多线程池的使用</b>		阅读数 4051
	Java多线程池的使用	Java通过Executors提供四种线程池，分别为： newCachedThr...	来自： <a href="#">永远年轻、永远...</a>
	• <b>Java线程池架构原理和源码解析(ThreadPoolExecutor)</b>		阅读数 1129
	在前面的文章中，我们使用线程的时候就去创建一个线程，这样实现起来非常简便， ...		来自： <a href="#">Aaron的学习历程</a>
	• <b>ThreadPoolExecutor源码解析（基于Java1.8）</b>		阅读数 3424
	第一部分：ThreadPoolExecutor的继承结构	根据上图可以知道，ThreadPoolExecut...	来自： <a href="#">Rebirth_Love的...</a>
	• <b>Java并发包： ExecutorService和ThreadPoolExecutor</b>		阅读数 6679
	文章译自：	http://tutorials.jenkov.com/java-util-concurrent/index.html 抽空翻译了一...	来自： <a href="#">charming的专栏</a>
	Java	Java教程	Java培训
	android 蓝牙源码分析	c++ string源码分析	android mms源码分析
	c++ pdf stl源码分析	c#mvc线程池	
	java程序员学习python	java区块链教程	
	• <b>线程池基础类_ThreadPoolExecutor（JDK1.8）</b>		阅读数 1759
	ThreadPoolExecutor简介	ThreadPoolExecutor使用线程池执行提交的任务，还...	来自： <a href="#">技术流水</a>
	• <b>Qt 之等待提示框（QMovie）</b>		阅读数 8619
	简述关于gif的使用在实际项目中我用的并不多，因为我感觉瑕疵挺多的，很多时候锯...		来自： <a href="#">青春不老，奋斗...</a>
	• <b>PHP jgraph库的配置及生成统计图表:折线图、柱状图、饼状图等</b>		阅读数 19398
	JpGraph简介	JpGraph是开源的PHP统计图表生成库，基于PHP的GD2图形库构...	来自： <a href="#">郎涯工作室</a>
	• <b>tensorflow在linux系统上的安装</b>		阅读数 12584
	tensorflow在ubuntu系统上按照官方文档安装起来相对容易，在centos上由于没有apt...	来自： <a href="#">zhangweijiqn的...</a>	
	• <b>java中的“回车”与“换行”</b>		阅读数 11376
	不同平台下的回车与换行		来自： <a href="#">buger的专栏</a>
	• <b>C#实现开发windows服务实现自动从FTP服务器下载文件（自行设置分...</b>		阅读数 6532
	最近在做一个每天定点从FTP自动下载节目.xml并更新到数据库的功能。首先想到用 ...	来自： <a href="#">kongwei521的...</a>	





Android Messenger

移动端跨平台开发框架 Cordova 学习笔记

(一) 环境搭建及创建第一个 Cordova

Android APP

Android 热修复技术浅析

个人分类

Android	11 篇
Java	2 篇
git	1 篇
cordova	1 篇
tool	1 篇

归档

2019年1月	1 篇
2017年7月	2 篇
2017年2月	1 篇
2017年1月	2 篇
2016年7月	2 篇

展开

热门文章

Android 如何判断当前线程是否是主线程

阅读数 18592

Java 线程池 ThreadPoolExecutor 源码分析

阅读数 7233

Android 图解向 Android Studio 中导入

Eclipse 工程的步骤

阅读数 5675

Java 中 List.subList() 方法的使用陷阱

阅读数 5671

Android 自定义控件源码分析----谈Android

自定义控件中 onMeasure()方法处理

阅读数 5510

最新评论

Java 线程池 ThreadPo...

u014473112: [reply]volvoxc[reply] 这个是有问题

Java 线程池 ThreadPo...

volvoxc: “这样第56行 while循环的 task != null 判断条件就为 false, 从而直接跳...

Java 中 List.subLi...

qq\_37625188: 这是稀粥啊,不过汤喝了,还是干货,有意思

Java 中 List.subLi...

qq\_42743987: 大佬戏好多哈哈哈哈

Java 中 List.subLi...

yxm234786: 有趣又有料，赞！



文字识别技术 免费调用50,000次/天

百度OCR可识别银行卡/身份证/票据/ 车牌/表格/网络图片等.准确度高!

香港服务器、CN2中港专线

CN2专线直联大陆，ping≤5ms，限量促销！ 智能电源控制+实时流量监控！

6元虚拟主机

直播系统源码

夏季运动紧身裤

游戏平台源码


联系我们




微信客服




QQ客服

 QQ客服


 kefu@csdn.net

 客服论坛

 400-660-0108

工作时间 8:30-22:00

关于我们 | 招聘 | 广告服务 | 网站地图

 百度提供站内搜索 京ICP证09002463号

©1999-2019 江苏乐知网络技术有限公司

江苏知之为计算机有限公司 北京创新乐知信息技术有限公司版权所有

网络110报警服务    经营性网站备案信息

北京互联网违法和不良信息举报中心

中国互联网举报中心

联系我们

 QQ客服


 kefu@csdn.net

 客服论坛

 400-660-0108

工作时间 8:30-22:00

关于我们 | 招聘 | 广告服务 | 网站地图

 百度提供站内搜索 京ICP证09002463号

©1999-2019 江苏乐知网络技术有限公司

江苏知之为计算机有限公司 北京创新乐知信息技术有限公司版权所有

网络110报警服务    经营性网站备案信息

北京互联网违法和不良信息举报中心

中国互联网举报中心