# Idea Buffer

## 深入理解Java线程池：ThreadPoolExecutor

2017-04-04 | 开发手册 , J.U.C |

### 线程池介绍

在web开发中，服务器需要接受并处理请求，所以会为一个请求来分配一个线程来进行处理。如果每次请求都新创建一个线程的话实现起来非常简便，但是存在一个问题：

**如果并发的请求数量非常多，但每个线程执行的时间很短，这样就会频繁的创建和销毁线程，如此一来会大大降低系统的效率。可能出现服务器在为每个请求创建新线程和销毁线程上花费的时间和消耗的系统资源要比处理实际的用户请求的时间和资源更多。**

那么有没有一种办法使执行完一个任务，并不被销毁，而是可以继续执行其他的任务呢?

这就是线程池的目的了。线程池为线程生命周期的开销和资源不足问题提供了解决方案。通过对多个任务重用线程，线程创建的开销被分摊到了多个任务上。

**什么时候使用线程池?**

- 单个任务处理时间比较短
- 需要处理的任务数量很大

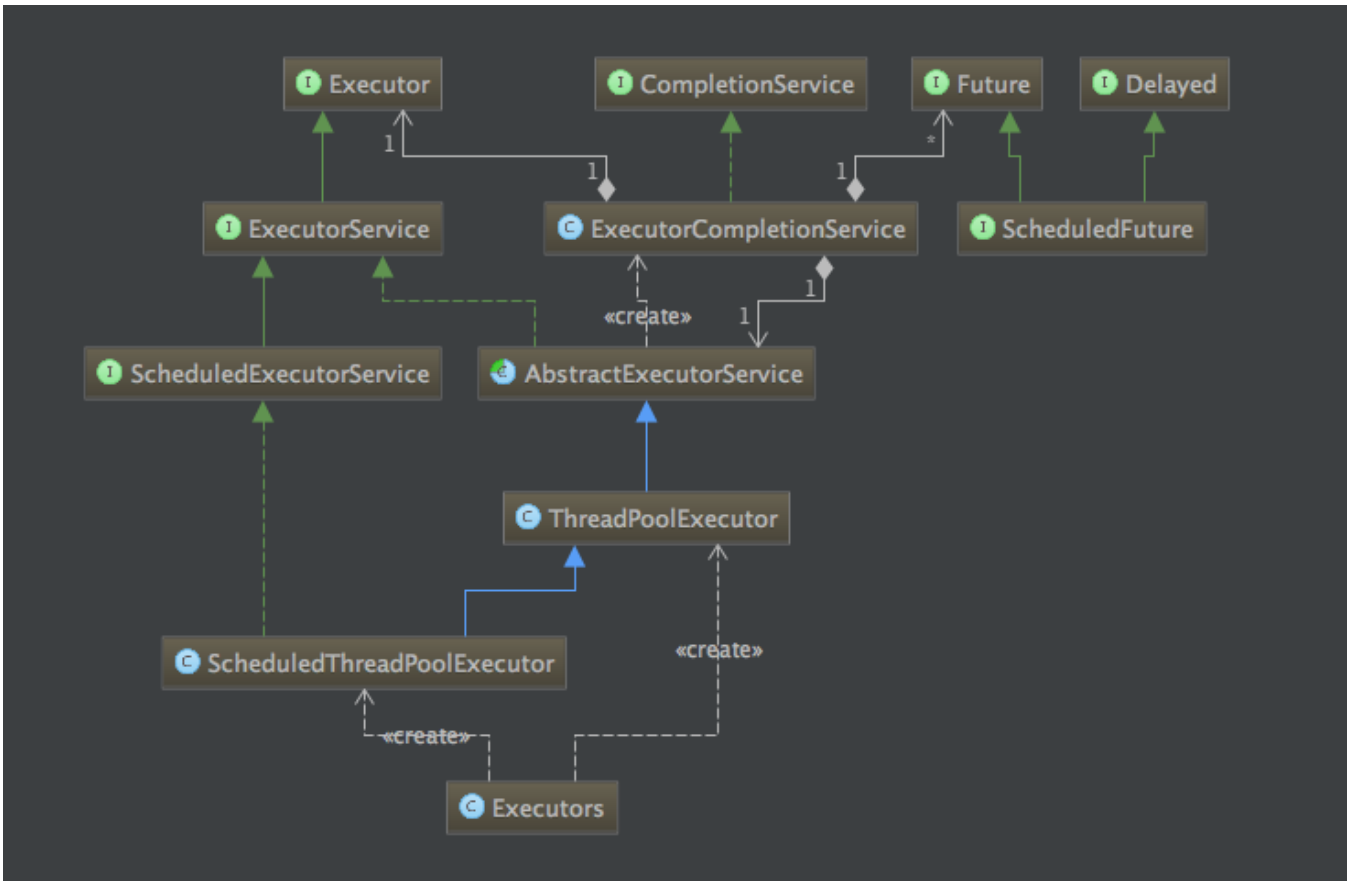
**使用线程池的好处**

引用自 http://ifeve.com/java-threadpool/ 的说明：

- 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

Java中的线程池是用ThreadPoolExecutor类来实现的. 本文就结合JDK 1.8对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的执行原理。

先看一下线程池的类图：




## Executor框架接口

Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架，目的是提供一种将”任务提交”与”任务如何运行”分离开来的机制。

J.U.C中有三个Executor接口：

- **Executor**：一个运行新任务的简单接口；
- **ExecutorService**：扩展了Executor接口。添加了一些用来管理执行器生命周期和任务生命周期的方法；
- **ScheduledExecutorService**：扩展了ExecutorService。支持Future和定期执行任务。

### Executor接口

```
1  public interface Executor {
2      void execute(Runnable command);
3  }
```

Executor接口只有一个execute方法，用来替代通常创建或启动线程的方法。例如，使用Thread来创建并启动线程的代码如下：

```
1   Thread t = new Thread();
2   t.start();
```

使用Executor来启动线程执行任务的代码如下：

```
1   Thread t = new Thread();
2   executor.execute(t);
```

对于不同的Executor实现，execute()方法可能是创建一个新线程并立即启动，也有可能是使用已有的工作线程来运行传入的任务，也可能是根据设置线程池的容量或者阻塞队列的容量来决定是否要将传入的线程放入阻塞队列中或者拒绝接收传入的线程。

### ExecutorService接口

ExecutorService接口继承自Executor接口，提供了管理终止的方法，以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。增加了shutDown()，shutDownNow()，invokeAll()，invokeAny()和submit()等方法。如果需要支持即时关闭，也就是shutDownNow()方法，则任务需要正确处理中断。

### ScheduledExecutorService接口

ScheduledExecutorService扩展ExecutorService接口并增加了schedule方法。调用schedule方法可以在指定的延时后执行一个Runnable或者Callable任务。ScheduledExecutorService接口还定义了按照指定时间间隔定期执行任务的scheduleAtFixedRate()方法和scheduleWithFixedDelay()方法。

## ThreadPoolExecutor分析

ThreadPoolExecutor继承自AbstractExecutorService，也是实现了ExecutorService接口。

### 几个重要的字段

```
1   private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
2   private static final int COUNT_BITS = Integer.SIZE - 3;
3   private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
4
5   // runState is stored in the high-order bits
6   private static final int RUNNING    = -1 << COUNT_BITS;
```

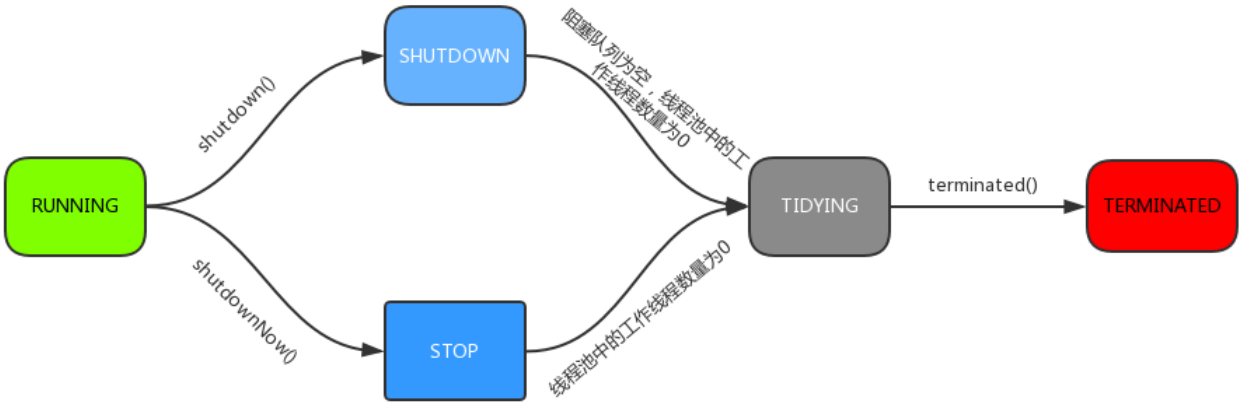
```
7   private static final int SHUTDOWN   =  0 << COUNT_BITS;
8   private static final int STOP       =  1 << COUNT_BITS;
9   private static final int TIDYING    =  2 << COUNT_BITS;
10  private static final int TERMINATED =  3 << COUNT_BITS;
```

`ctl` 是对线程池的运行状态和线程池中有效线程的数量进行控制的一个字段， 它包含两部分的信息: 线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)，这里可以看到，使用了Integer类型来保存，高3位保存runState，低29位保存workerCount。COUNT_BITS 就是29，CAPACITY就是1左移29位减1（29个1），这个常量表示workerCount的上限值，大约是5亿。

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**：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。（finalize() 方法在执行过程中也会调用shutdown()方法进入该状态）；
3. **STOP**：不能接受新任务，也不处理队列中的任务，会中断正在处理任务的线程。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**：如果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() 方法进入TERMINATED 状态。
5. **TERMINATED**：在terminated() 方法执行完后进入该状态，默认terminated()方法中什么也没有做。进入TERMINATED的条件如下：
    - 线程池不是RUNNING状态；
    - 线程池状态不是TIDYING状态或TERMINATED状态；
    - 如果线程池状态是SHUTDOWN并且workerQueue为空；
    - workerCount为0；
    - 设置TIDYING状态成功。

下图为线程池的状态转换过程：




## ctl相关方法

这里还有几个对ctl进行计算的方法：

```
1   private static int runStateOf(int c)     { return c & ~CAPACITY; }
2   private static int workerCountOf(int c)  { return c & CAPACITY; }
3   private static int ctlOf(int rs, int wc) { return rs | wc; }
```

- **runStateOf**：获取运行状态；
- **workerCountOf**：获取活动线程数；
- **ctlOf**：获取运行状态和活动线程数的值。

## ThreadPoolExecutor构造方法

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

```
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

构造方法中的字段含义如下：

- **corePoolSize**：核心线程数量，当有新任务在execute()方法提交时，会执行以下判断：
    - 如果运行的线程少于 corePoolSize，则创建新线程来处理任务，即使线程池中的其他线程是空闲的；
    - 如果线程池中的线程数量大于等于 corePoolSize 且小于 maximumPoolSize，则只有当workQueue满时才创建新的线程去处理任务；
    - 如果设置的corePoolSize 和 maximumPoolSize相同，则创建的线程池的大小是固定的，这时如果有新任务提交，若workQueue未满，则将请求放入workQueue中，等待有空闲的线程去从workQueue中取任务并处理；
    - 如果运行的线程数量大于等于maximumPoolSize，这时如果workQueue已经满了，则通过handler所指定的策略来处理任务；

  所以，任务提交时，判断的顺序为 corePoolSize –> workQueue –> maximumPoolSize。

- **maximumPoolSize**：最大线程数量；
- **workQueue**：等待队列，当任务提交时，如果线程池中的线程数量大于等于corePoolSize的时候，把该任务封装成一个Worker对象放入等待队列；
- **workQueue**：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, 线程池会根据当前线程池中正在运行着的线程的数量来决定对该任务的处理方式，主要有以下几种处理方式:
    - **直接切换**：这种方式常用的队列是SynchronousQueue，但现在还没有研究过该队列，这里暂时还没法介绍；
    - **使用无界队列**：一般使用基于链表的阻塞队列LinkedBlockingQueue。如果使用这种方式，那么线程池中能够创建的最大线程数就是corePoolSize，而maximumPoolSize就不会起作用了（后面也会说到）。当线程池中所有的核心线程都是RUNNING状态时，这时一个新的任务提交就会放入等待队列中。
    - **使用有界队列**：一般使用ArrayBlockingQueue。使用该方式可以将线程池的最大线程数量限制为maximumPoolSize，这样能够降低资源的消耗，但同时这种方式也使得线程池对线程的调度变得更困难，因为线程池和队列的容量都是有限的值，所以要想使线程池处理任务的吞吐率达到一个相对合理的范围，又想使线程调度相对简单，并且还要尽可能的降低线程池对资源的消耗，就需要合理的设置这两个数量。
        - 如果要想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），可以设置较大的队列容量和较小的线程池容量, 但这样也会降低线程处理任务的吞吐量。

    - 如果提交的任务经常发生阻塞，那么可以考虑通过调用 setMaximumPoolSize() 方法来重新设定线程池的容量。
    - 如果队列的容量设置的较小，通常需要将线程池的容量设置大一点，这样CPU的使用率会相对的高一些。但如果线程池的容量设置的过大，则在提交的任务数量太多的情况下，并发量会增加，那么线程之间的调度就是一个要考虑的问题，因为这样反而有可能降低处理任务的吞吐量。
- **keepAliveTime**：线程池维护线程所允许的空闲时间。当线程池中的线程数量大于corePoolSize的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了keepAliveTime；
- **threadFactory**：它是ThreadFactory类型的变量，用来创建新线程。默认使用Executors.defaultThreadFactory() 来创建线程。使用默认的ThreadFactory来创建线程时，会使新创建的线程具有相同的NORM_PRIORITY优先级并且是非守护线程，同时也设置了线程的名称。
- **handler**：它是RejectedExecutionHandler类型的变量，表示线程池的饱和策略。如果阻塞队列满了并且没有空闲的线程，这时如果继续提交任务，就需要采取一种策略处理该任务。线程池提供了4种策略：
    - AbortPolicy：直接抛出异常，这是默认策略；
    - CallerRunsPolicy：用调用者所在的线程来执行任务；
    - DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
    - DiscardPolicy：直接丢弃任务；

## execute方法

execute()方法用来提交任务，代码如下：

```
1  public void execute(Runnable command) {
2      if (command == null)
3          throw new NullPointerException();
4      /*
5       * clt记录着runState和workerCount
6       */
7      int c = ctl.get();
8      /*
9       * workerCountOf方法取出低29位的值，表示当前活动的线程数；
10      * 如果当前活动线程数小于corePoolSize，则新建一个线程放入线程池中；
11      * 并把任务添加到该线程中。
12      */
13     if (workerCountOf(c) < corePoolSize) {
14         /*
15          * addWorker中的第二个参数表示限制添加线程的数量是根据corePoolSize来判断还是maximumP
16          * 如果为true，根据corePoolSize来判断；
17          * 如果为false，则根据maximumPoolSize来判断
18          */
```

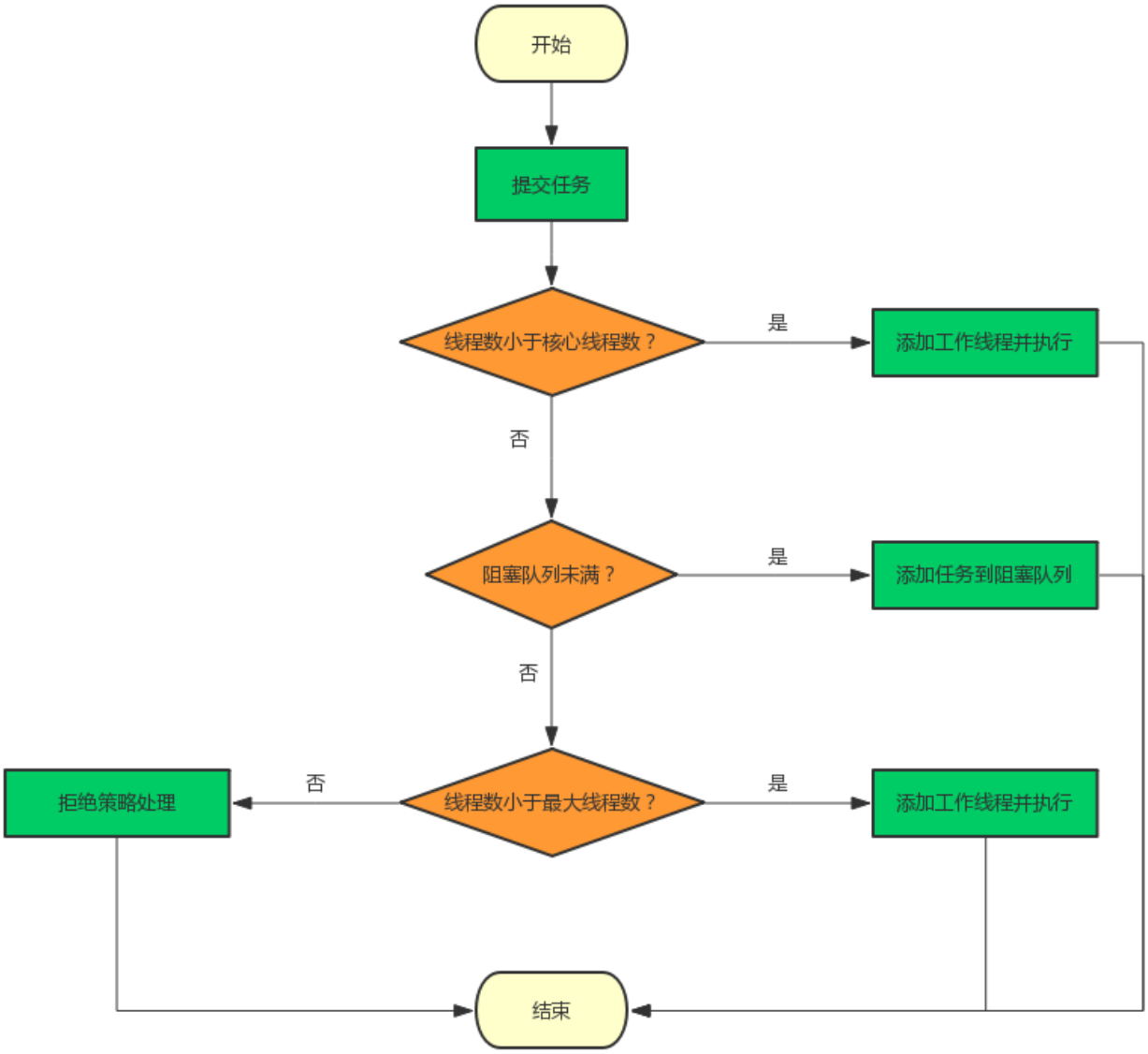
```
            if (addWorker(command, true))
                return;
            /*
             * 如果添加失败，则重新获取ctl值
             */
            c = ctl.get();
        }
        /*
         * 如果当前线程池是运行状态并且任务添加到队列成功
         */
        if (isRunning(c) && workQueue.offer(command)) {
            // 重新获取ctl值
            int recheck = ctl.get();
            // 再次判断线程池的运行状态，如果不是运行状态，由于之前已经把command添加到workQueue中
            // 这时需要移除该command
            // 执行过后通过handler使用拒绝策略对该任务进行处理，整个方法返回
            if (! isRunning(recheck) && remove(command))
                reject(command);
            /*
             * 获取线程池中的有效线程数，如果数量是0，则执行addWorker方法
             * 这里传入的参数表示：
             * 1. 第一个参数为null，表示在线程池中创建一个线程，但不去启动；
             * 2. 第二个参数为false，将线程池的有限线程数量的上限设置为maximumPoolSize，添加线程
             * 如果判断workerCount大于0，则直接返回，在workQueue中新增的command会在将来的某个时
             */
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        /*
         * 如果执行到这里，有两种情况：
         * 1. 线程池已经不是RUNNING状态；
         * 2. 线程池是RUNNING状态，但workerCount >= corePoolSize并且workQueue已满。
         * 这时，再次调用addWorker方法，但第二个参数传入为false，将线程池的有限线程数量的上限设置
         * 如果失败则拒绝该任务
         */
        else if (!addWorker(command, false))
            reject(command);
    }
```

简单来说，在执行execute()方法时如果状态一直是RUNNING时，的执行过程如下：

1. 如果 `workerCount < corePoolSize` ，则创建并启动一个线程来执行新提交的任务；
2. 如果 `workerCount >= corePoolSize` ，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中；
3. 如果 `workerCount >= corePoolSize && workerCount < maximumPoolSize` ，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务；
4. 如果 `workerCount >= maximumPoolSize` ，并且线程池内的阻塞队列已满, 则根据拒绝策略来处理该任务, 默认的处理方式是直接抛异常。

这里要注意一下 `addWorker(null, false);`，也就是创建一个线程，但并没有传入任务，因为任务已经被添加到workQueue中了，所以worker在执行的时候，会直接从workQueue中获取任务。所以，在 `workerCountOf(recheck) == 0` 时执行 `addWorker(null, false);` 也是为了保证线程池在RUNNING状态下必须要有一个线程来执行任务。

execute方法执行流程如下：




## addWorker方法

addWorker方法的主要工作是在线程池中创建一个新的线程并执行，firstTask参数 用于指定新增的线程执行的第一个任务，core参数为true表示在新增线程时会判断当前活动线程数是否少于corePoolSize，false表示新增线程前需要判断当前活动线程数是否少于maximumPoolSize，代码如下：

```
1   private boolean addWorker(Runnable firstTask, boolean core) {
```

```
    retry:
    for (;;) {
        int c = ctl.get();
        // 获取运行状态
        int rs = runStateOf(c);

        /*
         * 这个if判断
         * 如果rs >= SHUTDOWN，则表示此时不再接收新任务；
         * 接着判断以下3个条件，只要有1个不满足，则返回false：
         * 1. rs == SHUTDOWN，这时表示关闭状态，不再接受新提交的任务，但却可以继续处理阻塞队列
         * 2. firsTask为空
         * 3. 阻塞队列不为空
         *
         * 首先考虑rs == SHUTDOWN的情况
         * 这种情况下不会接受新提交的任务，所以在firstTask不为空的时候会返回false；
         * 然后，如果firstTask为空，并且workQueue也为空，则返回false，
         * 因为队列中已经没有任务了，不需要再添加线程了
         */
        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) {
            // 获取线程数
            int wc = workerCountOf(c);
            // 如果wc超过CAPACITY，也就是ctl的低29位的最大值（二进制是29个1），返回false；
            // 这里的core是addWorker方法的第二个参数，如果为true表示根据corePoolSize来比较，
            // 如果为false则根据maximumPoolSize来比较。
            //
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            // 尝试增加workerCount，如果成功，则跳出第一个for循环
            if (compareAndIncrementWorkerCount(c))
                break retry;
            // 如果增加workerCount失败，则重新获取ctl的值
            c = ctl.get();  // Re-read ctl
            // 如果当前的运行状态不等于rs，说明状态已被改变，返回第一个for循环继续执行
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }

    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
```

```
            // 根据firstTask来创建Worker对象
            w = new Worker(firstTask);
            // 每一个Worker对象都会创建一个线程
            final Thread t = w.thread;
            if (t != null) {
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                    // Recheck while holding lock.
                    // Back out on ThreadFactory failure or if
                    // shut down before lock acquired.
                    int rs = runStateOf(ctl.get());
                    // rs < SHUTDOWN表示是RUNNING状态;
                    // 如果rs是RUNNING状态或者rs是SHUTDOWN状态并且firstTask为null，向线程池中添
                    // 因为在SHUTDOWN时不会在添加新的任务，但还是会执行workQueue中的任务
                    if (rs < SHUTDOWN ||
                        (rs == SHUTDOWN && firstTask == null)) {
                        if (t.isAlive()) // precheck that t is startable
                            throw new IllegalThreadStateException();
                        // workers是一个HashSet
                        workers.add(w);
                        int s = workers.size();
                        // largestPoolSize记录着线程池中出现过的最大线程数量
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                        workerAdded = true;
                    }
                } finally {
                    mainLock.unlock();
                }
                if (workerAdded) {
                    // 启动线程
                    t.start();
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                addWorkerFailed(w);
        }
        return workerStarted;
    }
```

注意一下这里的 `t.start()` 这个语句，启动时会调用Worker类中的run方法，Worker本身实现了Runnable接口，所以一个Worker类型的对象也是一个线程。

## Worker类

线程池中的每一个线程被封装成一个Worker对象，ThreadPool维护的其实就是一组Worker对象，看一下Worker的定义：

```
1  private final class Worker
2          extends AbstractQueuedSynchronizer
3          implements Runnable
4  {
5      /**
6       * This class will never be serialized, but we provide a
7       * serialVersionUID to suppress a javac warning.
8       */
9      private static final long serialVersionUID = 6138294804551838833L;
10
11     /** Thread this worker is running in.  Null if factory fails. */
12     final Thread thread;
13     /** Initial task to run.  Possibly null. */
14     Runnable firstTask;
15     /** Per-thread task counter */
16     volatile long completedTasks;
17
18     /**
19      * Creates with given first task and thread from ThreadFactory.
20      * @param firstTask the first task (null if none)
21      */
22     Worker(Runnable firstTask) {
23         setState(-1); // inhibit interrupts until runWorker
24         this.firstTask = firstTask;
25         this.thread = getThreadFactory().newThread(this);
26     }
27
28     /** Delegates main run loop to outer runWorker  */
29     public void run() {
30         runWorker(this);
31     }
32
33     // Lock methods
34     //
35     // The value 0 represents the unlocked state.
36     // The value 1 represents the locked state.
37
38     protected boolean isHeldExclusively() {
39         return getState() != 0;
40     }
41
42     protected boolean tryAcquire(int unused) {
43         if (compareAndSetState(0, 1)) {
44             setExclusiveOwnerThread(Thread.currentThread());
45             return true;
46         }
47         return false;
```

```
48        }
49
50        protected boolean tryRelease(int unused) {
51            setExclusiveOwnerThread(null);
52            setState(0);
53            return true;
54        }
55
56        public void lock()        { acquire(1); }
57        public boolean tryLock()  { return tryAcquire(1); }
58        public void unlock()      { release(1); }
59        public boolean isLocked() { return isHeldExclusively(); }
60
61        void interruptIfStarted() {
62            Thread t;
63            if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
64                try {
65                    t.interrupt();
66                } catch (SecurityException ignore) {
67                }
68            }
69        }
70    }
```

Worker类继承了AQS，并实现了Runnable接口，注意其中的firstTask和thread属性：firstTask用它来保存传入的任务；thread是在调用构造方法时通过ThreadFactory来创建的线程，是用来处理任务的线程。

在调用构造方法时，需要把任务传入，这里通过 `getThreadFactory().newThread(this);` 来新建一个线程，newThread方法传入的参数是this，因为Worker本身继承了Runnable接口，也就是一个线程，所以一个Worker对象在启动的时候会调用Worker类中的run方法。

Worker继承了AQS，使用AQS来实现独占锁的功能。为什么不使用ReentrantLock来实现呢？可以看到tryAcquire方法，它是不允许重入的，而ReentrantLock是允许重入的：

1. lock方法一旦获取了独占锁，表示当前线程正在执行任务中；
2. 如果正在执行任务，则不应该中断线程；
3. 如果该线程现在不是独占锁的状态，也就是空闲的状态，说明它没有在处理任务，这时可以对该线程进行中断；
4. 线程池在执行shutdown方法或tryTerminate方法时会调用interruptIdleWorkers方法来中断空闲的线程，interruptIdleWorkers方法会使用tryLock方法来判断线程池中的线程是否是空闲状态；
5. 之所以设置为不可重入，是因为我们不希望任务在调用像setCorePoolSize这样的线程池控制方法时重新获取锁。如果使用ReentrantLock，它是可重入的，这样如果在任务中调用了如setCorePoolSize这类线程池控制的方法，会中断正在运行的线程。

所以，Worker继承自AQS，用于判断线程是否空闲以及是否可以被中断。

此外，在构造方法中执行了 `setState(-1);`，把state变量设置为-1，为什么这么做呢？是因为AQS中默认的state是0，如果刚创建了一个Worker对象，还没有执行任务时，这时就不应该被中断，看一下tryAquire方法：

```
protected boolean tryAcquire(int unused) {
    if (compareAndSetState(0, 1)) {
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    }
    return false;
}
```

tryAcquire方法是根据state是否是0来判断的，所以， `setState(-1);`将state设置为-1是为了禁止在执行任务前对线程进行中断。

正因为如此，在runWorker方法中会先调用Worker对象的unlock方法将state设置为0.

### runWorker方法

在Worker类中的run方法调用了runWorker方法来执行任务，runWorker方法的代码如下：

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    // 获取第一个任务
    Runnable task = w.firstTask;
    w.firstTask = null;
    // 允许中断
    w.unlock(); // allow interrupts
    // 是否因为异常退出循环
    boolean completedAbruptly = true;
    try {
        // 如果task为空，则通过getTask来获取任务
        while (task != null || (task = getTask()) != null) {
            w.lock();
            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted.  This
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
```

```
                    wt.interrupt();
                try {
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
                        task.run();
                    } catch (RuntimeException x) {
                        thrown = x; throw x;
                    } catch (Error x) {
                        thrown = x; throw x;
                    } catch (Throwable x) {
                        thrown = x; throw new Error(x);
                    } finally {
                        afterExecute(task, thrown);
                    }
                } finally {
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            processWorkerExit(w, completedAbruptly);
        }
    }
```

这里说明一下第一个if判断，目的是：

- 如果线程池正在停止，那么要保证当前线程是中断状态；
- 如果不是的话，则要保证当前线程不是中断状态；

这里要考虑在执行该if语句期间可能也执行了shutdownNow方法，shutdownNow方法会把状态设置为STOP，回顾一下STOP状态：

> 不能接受新任务，也不处理队列中的任务，会中断正在处理任务的线程。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。

STOP状态要中断线程池中的所有线程，而这里使用 `Thread.interrupted()` 来判断是否中断是为了确保在RUNNING或者SHUTDOWN状态时线程是非中断状态的，因为Thread.interrupted()方法会复位中断的状态。

总结一下runWorker方法的执行过程：

1. while循环不断地通过getTask()方法获取任务；

2. getTask()方法从阻塞队列中取任务；
3. 如果线程池正在停止，那么要保证当前线程是中断状态，否则要保证当前线程不是中断状态；
4. 调用 task.run() 执行任务；
5. 如果task为null则跳出循环，执行processWorkerExit()方法；
6. runWorker方法执行完毕，也代表着Worker中的run方法执行完毕，销毁线程。

这里的beforeExecute方法和afterExecute方法在ThreadPoolExecutor类中是空的，留给子类来实现。

completedAbruptly变量来表示在执行任务过程中是否出现了异常，在processWorkerExit方法中会对该变量的值进行判断。

## getTask方法

getTask方法用来从阻塞队列中取任务，代码如下：

```
1  private Runnable getTask() {
2      // timeOut变量的值表示上次从阻塞队列中取任务时是否超时
3      boolean timedOut = false; // Did the last poll() time out?
4
5      for (;;) {
6          int c = ctl.get();
7          int rs = runStateOf(c);
8
9          // Check if queue empty only if necessary.
10         /*
11          * 如果线程池状态rs >= SHUTDOWN，也就是非RUNNING状态，再进行以下判断：
12          * 1. rs >= STOP，线程池是否正在stop；
13          * 2. 阻塞队列是否为空。
14          * 如果以上条件满足，则将workerCount减1并返回null。
15          * 因为如果当前线程池状态的值是SHUTDOWN或以上时，不允许再向阻塞队列中添加任务。
16          */
17         if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
18             decrementWorkerCount();
19             return null;
20         }
21
22         int wc = workerCountOf(c);
23
24         // Are workers subject to culling?
25         // timed变量用于判断是否需要进行超时控制。
26         // allowCoreThreadTimeOut默认是false，也就是核心线程不允许进行超时；
27         // wc > corePoolSize，表示当前线程池中的线程数量大于核心线程数量；
28         // 对于超过核心线程数量的这些线程，需要进行超时控制
29         boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
30
31         /*
32          * wc > maximumPoolSize的情况是因为可能在此方法执行阶段同时执行了setMaximumPoolSize
```

```
33              * timed && timedOut 如果为true，表示当前操作需要进行超时控制，并且上次从阻塞队列中获
34              * 接下来判断，如果有效线程数量大于1，或者阻塞队列是空的，那么尝试将workerCount减1；
35              * 如果减1失败，则返回重试。
36              * 如果wc == 1时，也就说明当前线程是线程池中唯一的一个线程了。
37              */
38             if ((wc > maximumPoolSize || (timed && timedOut))
39                 && (wc > 1 || workQueue.isEmpty())) {
40                 if (compareAndDecrementWorkerCount(c))
41                     return null;
42                 continue;
43             }
44
45             try {
46                 /*
47                  * 根据timed来判断，如果为true，则通过阻塞队列的poll方法进行超时控制，如果在keepA
48                  * 否则通过take方法，如果这时队列为空，则take方法会阻塞直到队列不为空。
49                  *
50                  */
51                 Runnable r = timed ?
52                     workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
53                     workQueue.take();
54                 if (r != null)
55                     return r;
56                 // 如果 r == null，说明已经超时，timedOut设置为true
57                 timedOut = true;
58             } catch (InterruptedException retry) {
59                 // 如果获取任务时当前线程发生了中断，则设置timedOut为false并返回循环重试
60                 timedOut = false;
61             }
62         }
63     }
```

这里重要的地方是第二个if判断，目的是控制线程池的有效线程数量。由上文中的分析可以知道，在执行execute方法时，如果当前线程池的线程数量超过了corePoolSize且小于maximumPoolSize，并且workQueue已满时，则可以增加工作线程，但这时如果超时没有获取到任务，也就是timedOut为true的情况，说明workQueue已经为空了，也就说明了当前线程池中不需要那么多线程来执行任务了，可以把多于corePoolSize数量的线程销毁掉，保持线程数量在corePoolSize即可。

什么时候会销毁？当然是runWorker方法执行完之后，也就是Worker中的run方法执行完，由JVM自动回收。

getTask方法返回null时，在runWorker方法中会跳出while循环，然后会执行processWorkerExit方法。

## processWorkerExit方法

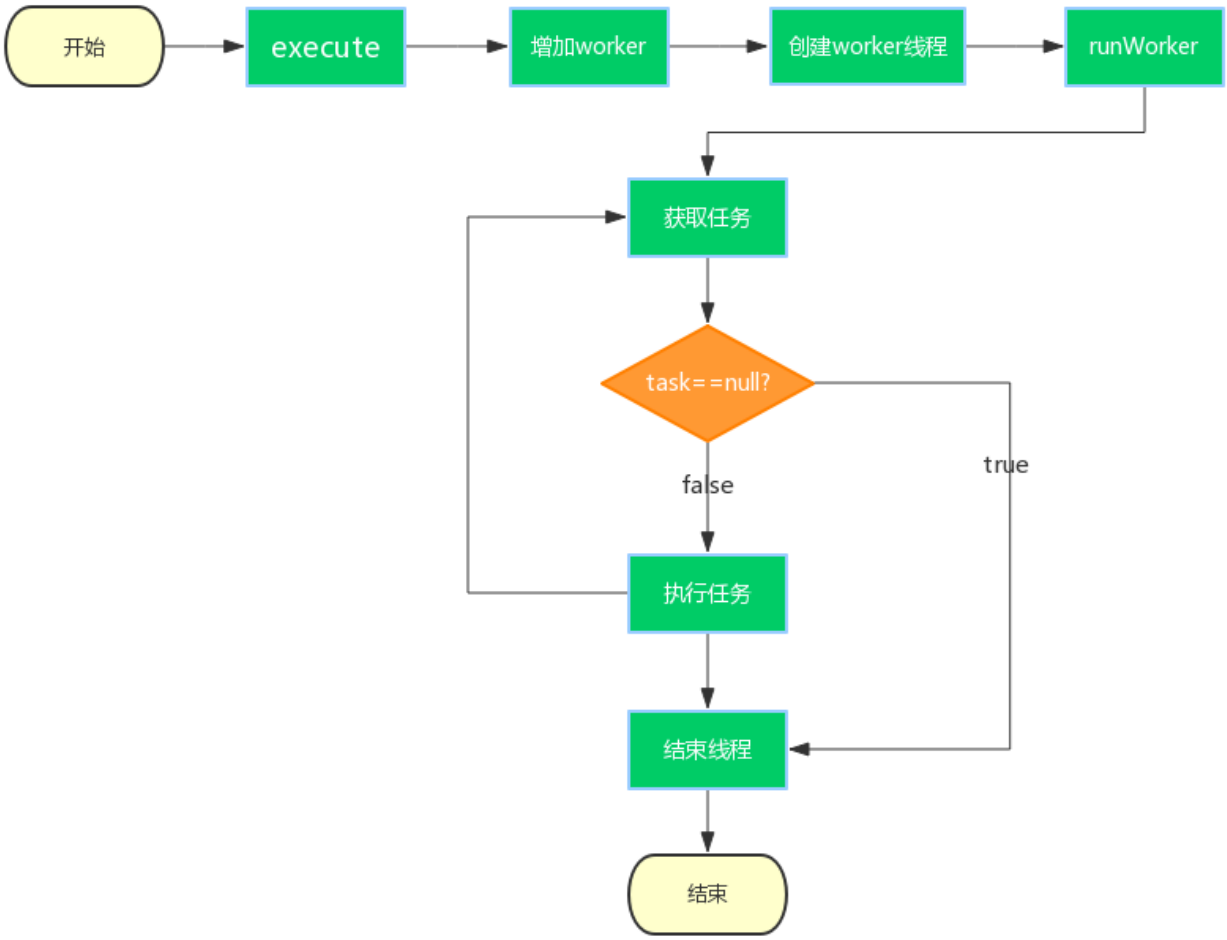
```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    // 如果completedAbruptly值为true，则说明线程执行时出现了异常，需要将workerCount减1；
    // 如果线程执行时没有出现异常，说明在getTask()方法中已经已经对workerCount进行了减1操作，这
    if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
        decrementWorkerCount();

    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //统计完成的任务数
        completedTaskCount += w.completedTasks;
        // 从workers中移除，也就表示着从线程池中移除了一个工作线程
        workers.remove(w);
    } finally {
        mainLock.unlock();
    }

    // 根据线程池状态进行判断是否结束线程池
    tryTerminate();

    int c = ctl.get();
    /*
     * 当线程池是RUNNING或SHUTDOWN状态时，如果worker是异常结束，那么会直接addWorker；
     * 如果allowCoreThreadTimeOut=true，并且等待队列有任务，至少保留一个worker；
     * 如果allowCoreThreadTimeOut=false，workerCount不少于corePoolSize。
     */
    if (runStateLessThan(c, STOP)) {
        if (!completedAbruptly) {
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
            if (min == 0 && ! workQueue.isEmpty())
                min = 1;
            if (workerCountOf(c) >= min)
                return; // replacement not needed
        }
        addWorker(null, false);
    }
}
```

至此，processWorkerExit执行完之后，工作线程被销毁，以上就是整个工作线程的生命周期，从execute方法开始，Worker使用ThreadFactory创建新的工作线程，runWorker通过getTask获取任务，然后执行任务，如果getTask返回null，进入processWorkerExit方法，整个线程结束，如图所示：




## tryTerminate方法

tryTerminate方法根据线程池状态进行判断是否结束线程池，代码如下：

```
1   final void tryTerminate() {
2       for (;;) {
3           int c = ctl.get();
4           /*
5            * 当前线程池的状态为以下几种情况时，直接返回：
6            * 1. RUNNING，因为还在运行中，不能停止；
7            * 2. TIDYING或TERMINATED，因为线程池中已经没有正在运行的线程了；
8            * 3. SHUTDOWN并且等待队列非空，这时要执行完workQueue中的task；
9            */
10          if (isRunning(c) ||
11              runStateAtLeast(c, TIDYING) ||
12              (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
13              return;
14          // 如果线程数量不为0，则中断一个空闲的工作线程，并返回
15          if (workerCountOf(c) != 0) { // Eligible to terminate
16              interruptIdleWorkers(ONLY_ONE);
17              return;
18          }
19
```

```
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // 这里尝试设置状态为TIDYING，如果设置成功，则调用terminated方法
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                    try {
                        // terminated方法默认什么都不做，留给子类实现
                        terminated();
                    } finally {
                        // 设置状态为TERMINATED
                        ctl.set(ctlOf(TERMINATED, 0));
                        termination.signalAll();
                    }
                    return;
                }
            } finally {
                mainLock.unlock();
            }
            // else retry on failed CAS
        }
    }
```

`interruptIdleWorkers(ONLY_ONE);`的作用是因为在getTask方法中执行`workQueue.take()`时，如果不执行中断会一直阻塞。在下面介绍的shutdown方法中，会中断所有空闲的工作线程，如果在执行shutdown时工作线程没有空闲，然后又去调用了getTask方法，这时如果workQueue中没有任务了，调用`workQueue.take()`时就会一直阻塞。所以每次在工作线程结束时调用tryTerminate方法来尝试中断一个空闲工作线程，避免在队列为空时取任务一直阻塞的情况。

## shutdown方法

shutdown方法要将线程池切换到SHUTDOWN状态，并调用interruptIdleWorkers方法请求中断所有空闲的worker，最后调用tryTerminate尝试结束线程池。

```
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 安全策略判断
            checkShutdownAccess();
            // 切换状态为SHUTDOWN
            advanceRunState(SHUTDOWN);
            // 中断空闲线程
            interruptIdleWorkers();
            onShutdown(); // hook for ScheduledThreadPoolExecutor
        } finally {
            mainLock.unlock();
        }
```

```
        // 尝试结束线程池
        tryTerminate();
    }
```

这里思考一个问题：在runWorker方法中，执行任务时对Worker对象w进行了lock操作，为什么要在执行任务的时候对每个工作线程都加锁呢?

下面仔细分析一下：

- 在getTask方法中，如果这时线程池的状态是SHUTDOWN并且workQueue为空，那么就应该返回null来结束这个工作线程，而使线程池进入SHUTDOWN状态需要调用shutdown方法；
- shutdown方法会调用interruptIdleWorkers来中断空闲的线程，interruptIdleWorkers持有mainLock，会遍历workers来逐个判断工作线程是否空闲。但getTask方法中没有mainLock；
- 在getTask中，如果判断当前线程池状态是RUNNING，并且阻塞队列为空，那么会调用 `workQueue.take()` 进行阻塞；
- 如果在判断当前线程池状态是RUNNING后，这时调用了shutdown方法把状态改为了SHUTDOWN，这时如果不进行中断，那么当前的工作线程在调用了 `workQueue.take()` 后会一直阻塞而不会被销毁，因为在SHUTDOWN状态下不允许再有新的任务添加到workQueue中，这样一来线程池永远都关闭不了了；
- 由上可知，shutdown方法与getTask方法（从队列中获取任务时）存在竞态条件；
- 解决这一问题就需要用到线程的中断，也就是为什么要用interruptIdleWorkers方法。在调用 `workQueue.take()` 时，如果发现当前线程在执行之前或者执行期间是中断状态，则会抛出InterruptedException，解除阻塞的状态；
- 但是要中断工作线程，还要判断工作线程是否是空闲的，如果工作线程正在处理任务，就不应该发生中断；
- 所以Worker继承自AQS，在工作线程处理任务时会进行lock，interruptIdleWorkers在进行中断时会使用tryLock来判断该工作线程是否正在处理任务，如果tryLock返回true，说明该工作线程当前未执行任务，这时才可以被中断。

下面就来分析一下interruptIdleWorkers方法。

## interruptIdleWorkers方法

```
private void interruptIdleWorkers() {
    interruptIdleWorkers(false);
}

private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
```

```
        mainLock.lock();
        try {
            for (Worker w : workers) {
                Thread t = w.thread;
                if (!t.isInterrupted() && w.tryLock()) {
                    try {
                        t.interrupt();
                    } catch (SecurityException ignore) {
                    } finally {
                        w.unlock();
                    }
                }
                if (onlyOne)
                    break;
            }
        } finally {
            mainLock.unlock();
        }
    }
```

interruptIdleWorkers遍历workers中所有的工作线程，若线程没有被中断tryLock成功，就中断该线程。

为什么需要持有mainLock？因为workers是HashSet类型的，不能保证线程安全。

### shutdownNow方法

```
    public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            advanceRunState(STOP);

            // 中断所有工作线程，无论是否空闲
            interruptWorkers();
            // 取出队列中没有被执行的任务
            tasks = drainQueue();
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
        return tasks;
    }
```

shutdownNow方法与shutdown方法类似，不同的地方在于：

1. 设置状态为STOP；
2. 中断所有工作线程，无论是否是空闲的；
3. 取出阻塞队列中没有被执行的任务并返回。

shutdownNow方法执行完之后调用tryTerminate方法，该方法在上文已经分析过了，目的就是使线程池的状态设置为TERMINATED。

## 线程池的监控

通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的时候可以使用

- **getTaskCount**：线程池已经执行的和未执行的任务总数；
- **getCompletedTaskCount**：线程池已完成的任务数量，该值小于等于taskCount；
- **getLargestPoolSize**：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过，也就是达到了maximumPoolSize；
- **getPoolSize**：线程池当前的线程数量；
- **getActiveCount**：当前线程池中正在执行任务的线程数量。

通过这些方法，可以对线程池进行监控，在ThreadPoolExecutor类中提供了几个空方法，如beforeExecute方法，afterExecute方法和terminated方法，可以扩展这些方法在执行前或执行后增加一些新的操作，例如统计线程池的执行任务的时间等，可以继承自ThreadPoolExecutor来进行扩展。

## 总结

本文比较详细的分析了线程池的工作流程，总体来说有如下几个内容：

- 分析了线程的创建，任务的提交，状态的转换以及线程池的关闭；
- 这里通过execute方法来展开线程池的工作流程，execute方法通过corePoolSize，maximumPoolSize以及阻塞队列的大小来判断决定传入的任务应该被立即执行，还是应该添加到阻塞队列中，还是应该拒绝任务。
- 介绍了线程池关闭时的过程，也分析了shutdown方法与getTask方法存在竞态条件；
- 在获取任务时，要通过线程池的状态来判断应该结束工作线程还是阻塞线程等待新的任务，也解释了为什么关闭线程池时要中断工作线程以及为什么每一个worker都需要lock。

在向线程池提交任务时，除了execute方法，还有一个submit方法，submit方法会返回一个Future对象用于获取返回值，有关Future和Callable请自行了解一下相关的文章，这里就不介绍了。

#Java #Concurrent #线程池 #ThreadPool

深入理解AbstractQueuedSynchronizer（三）

FutureTask源码解析