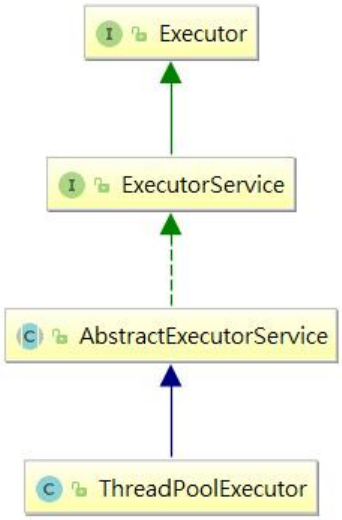
ThreadPoolExecutor(线程池) 源码学习

作者：zouxiaochaun
原文链接：https://ld246.com/article/1568601232030
来源网站：链滴

1. 概述

1.1 ThreadPoolExecutor 继承关系：

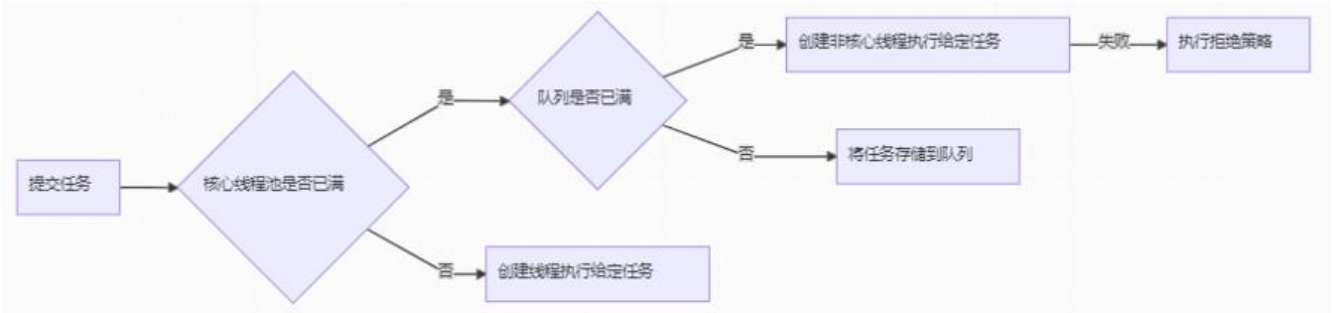


1.2 线程池主要方法：

```
// 由AbstractExecutorService实现，在ThreadPoolExecutor中没有重载实现
// 提交一个Runnable任务，并返回任务执行结果
<T> Future<T> submit(Runnable task, T result);

// 由AbstractExecutorService实现，在ThreadPoolExecutor中没有重载实现
// 向线程池中批量提交任务，并返回任务执行结果
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;

// 由AbstractExecutorService实现，在ThreadPoolExecutor中没有重载实现
// 批量提交任务，返回其中一个已成功完成的任务的结果，其他任务将被取消。
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

// 由ThreadPoolExecutor实现，继承自 Executor接口
// 在将来的某个时间执行给定的任务
void execute(Runnable command);
```

2. ThreadPoolExecutor 基本结构

2.1 线程池处理流程



原文链接：ThreadPoolExecutor(线程池) 源码学习

2.2 存储结构

```
// 存储Worker线程
private final HashSet<Worker> workers = new HashSet<Worker>();
// 阻塞队列，当添加任务时线程池中工作线程已满，添加的任务将被存入workQueue队列
private final BlockingQueue<Runnable> workQueue;
```

2.3 基本参数

2.3.1 构造方法参数：

- corePoolSize: 核心线程池大小
- maximumPoolSize: 线程池中最大线程数
- keepAliveTime & unit: 线程等待任务超时时间
- workQueue: 任务队列，当线程池中没有空闲线程来接受新任务时，新任务将被存储在此队列中
- threadFactory: 线程工厂，默认为 Executors.defaultThreadFactory()
- handler: 拒绝任务处理器，当新任务无法被添加到线程池时，新任务将交由 handler 来处理，默认理器为 AbortPolicy，该处理器的处理方式是抛出一个 RejectedExecutionException 异常

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    /* 只做了参数校验和参数保存，没有其他任何操作 */
}
```

2.3.2 其他参数

- ctl 参数是一个 AtomicInteger 类型变量，其中保存了线程池两个重要概念

a. 高 3 位表示线程池的状态值(runState)

b. 低 29 位表示线程池内线程数量(workerCount)

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

与此表示方法相似的还有 ReentrantReadWriteLock 中的 state 以及 ConcurrentHashMap 中的 siz Ctl，都是用一个变量存储两部分内容

- 线程池的 5 种状态，在 ctl 的高 3 位中 111 表示 RUNNING 状态，000 表示 SHUTDOWN 状态，01 表示 STOP 状态，010 表示 TIDYING 状态，011 表示 TERMINATED 状态

```
private static final int COUNT_BITS = Integer.SIZE - 3;
// runState is stored in the high-order bits
// 11100000000000000000000000000000
private static final int RUNNING    = -1 << COUNT_BITS;
// 00000000000000000000000000000000
private static final int SHUTDOWN   =  0 << COUNT_BITS;
```

```
// 00100000000000000000000000000000
private static final int STOP       =  1 << COUNT_BITS;
// 01000000000000000000000000000000
private static final int TIDYING    =  2 << COUNT_BITS;
// 01100000000000000000000000000000
private static final int TERMINATED =  3 << COUNT_BITS;
```

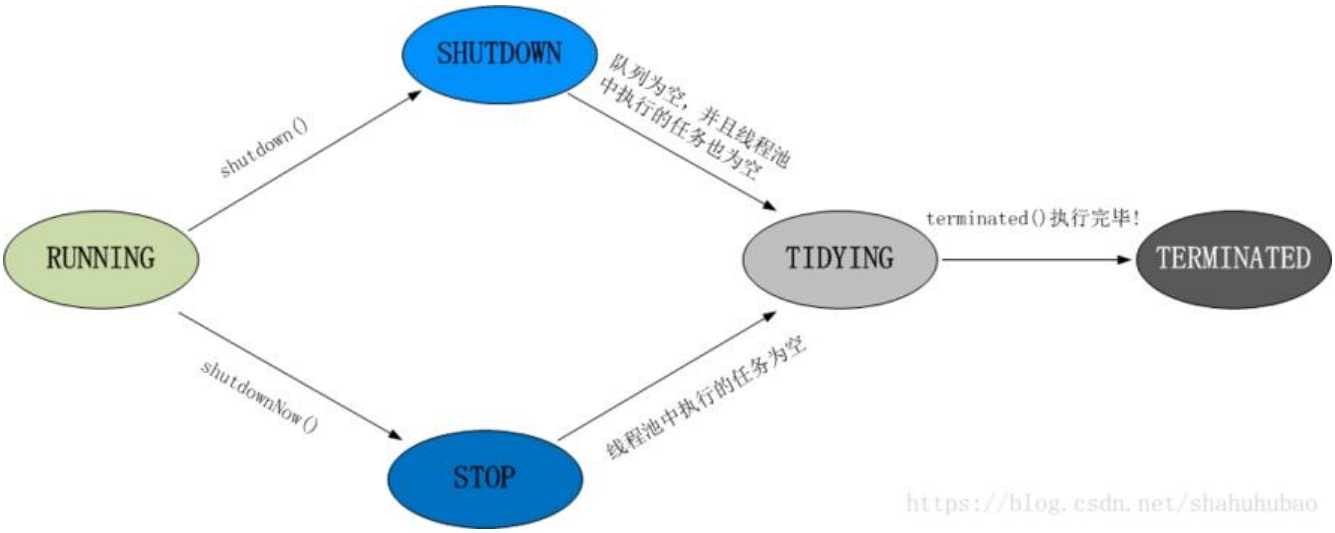
- ctl 参数的解析、封装方法

```
// Packing and unpacking ctl
// 解析线程池状态码
private static int runStateOf(int c)     { return c & ~CAPACITY; }
// 解析工作线程数
private static int workerCountOf(int c)  { return c & CAPACITY; }
// 将 线程池状态码 和 工作线程数 打包
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

- 线程池允许的最大线程数(约 500 万)

```
// 00011111111111111111111111111111
// 536,870,911
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
```

2.4 线程池状态以及状态切换



1. RUNNING(111)

(a) 状态说明：线程池处在 RUNNING 状态时，能够接收新任务，以及对已添加的任务进行处理。

(b) 状态切换：线程池的初始化状态是 RUNNING。换句话说，线程池被一旦被创建，就处于 RUNNI G 状态，并且线程池中的任务数为 0!

2. SHUTDOWN(000)

(a) 状态说明：线程池处在 SHUTDOWN 状态时，不接收新任务，但能处理已添加的任务。

(b) 状态切换：调用线程池的 shutdown()接口时，线程池由 RUNNIN -> SHUTDOWN。

3. STOP(001)

(a) 状态说明：线程池处在 STOP 状态时，不接收新任务，不处理已添加的任务，并且会中断正在处

的任务。

(b) 状态切换：调用线程池的 shutdownNow()接口时，线程池由 RUNNING or SHUTDOWN) -> STOP。

4. TIDYING(010)

(a) 状态说明：当所有的任务已终止，ctl 记录的”任务数量”为 0，线程池会变为 TIDYING 状态。线程池变为 TIDYING 状态时，会执行钩子函数 terminated()。terminated()在 ThreadPoolExecutor 类中是空的，若用户想在线程池变为 TIDYING 时，进行相应的处理；可以通过重载 terminated()函来实现。

(b) 状态切换：当线程池在 SHUTDOWN 状态下，阻塞队列为空并且线程池中执行的任务也为空时，会由 SHUTDOWN -> TIDYING。

当线程池在 STOP 状态下，线程池中执行的任务为空时，就会由 STOP -> TIDYING。

5. TERMINATED(011)

(a) 状态说明：线程池彻底终止，就变成 TERMINATED 状态。

(b) 状态切换：线程池处在 TIDYING 状态时，执行完 terminated()之后，就会由 TIDYING -> TERMINATED。

3 主要代码

3.1 ThreadPoolExecutor#execute

提交一个 Runnable 任务到线程池，任务执行的时间由当前线程池中工作线程的数量决定，如果线程量小于 corePoolSize，线程池会直接创建一个新的线程来执行任务，否则会将其存入一个队列中等线程获取并执行

1. 如果线程池中线程数量少于 corePoolSize，创建新的线程来执行新添加的任务，详见 addWorker()

2. 如果线程池中线程数量大于等于 corePoolSize，但队列 workQueue 未满，则将新添加的任务放到 workQueue 中

3. 如果线程池中线程数量大于等于 corePoolSize 小于 maximumPoolSize，且队列 workQueue 已，则会在 maximumPoolSize 容量下创建新的线程来处理被添加的任务

4. 如果线程池中线程数量等于 maximumPoolSize，就用 RejectedExecutionHandler 来执行拒绝策略

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    /*
     * Proceed in 3 steps:
     *
     * 1. If fewer than corePoolSize threads are running, try to start a new thread with the given command as its first
     * task.  The call to addWorker atomically checks runState and workerCount, and so prevent false alarms that would add
     * threads when it shouldn't, by returning false.
     * 1、如果线程池中worker线程数量小于corePoolSize，开一个新的线程并将command作为该线的第一个任务。
```

```
     *
     * 2. If a task can be successfully queued, then we still need to double-check whether we sh
uld have added a thread
     * (because existing ones died since last checking) or that the pool shut down since entry in
o this method. So we
     * recheck state and if necessary roll back the enqueuing if stopped, or start a new thread if
there are none.
     * 2、如果任务成功放入队列，我们仍需要重新校验去确认是否应该新建一个线程（因为可能存在
些线程在我们上次检查后死了） 或者 从我们进入这个方法后，线程池被关闭了
     * 所以我们需要再次检查线程池状态，如果线程池停止了需要回滚入队列，如果池中没有线程了，
开启一个线程
     *
     * 3. If we cannot queue task, then we try to add a new thread.  If it fails, we know we are s
ut down or saturated
     * and so reject the task.
     * 3、如果不能将任务存入workQueue，那么需要添加一个新的worker线程，如果添加线程失败
明线程池shutdown或者饱和了，需要将这个任务交给拒绝任务处理器来处理
     */
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        // 如果核心线程池未满，则向核心线程池中添加一个worker线程，并将command作为该线程
第一个任务
        // 如果操作成功，直接返回
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 此时，核心线程池已满 或 核心线程池未满但是添加worker线程失败
    // 如果此时线程池处于RUNNING状态，则将提交的任务添加到workQueue队列 (线程池只有在R
NNING状态才会接收新任务)
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        // 再次检查线程池是否处于运行状态，如果不是则将入队成功的任务移除，并执行拒绝策略，
认抛出RejectedExceptionException异常
        if (! isRunning(recheck) && remove(command))
            reject(command);
        // else 即线程池当前为运行状态 或者 从workQueue中移除任务失败
        // 如果此时工作线程计数为0，则在maxPoolSize容量下开启空的工作线程用于处理workQueu
队列中的任务
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // else 即线程池状态不为RUNNING 或者 RUNNING状态但是存储新任务失败
    // 则在maxPoolSize容量下开启新工作线程，开启失败则执行拒绝策略
    else if (!addWorker(command, false))
        reject(command);
}
```

3.2 ThreadPoolExecutor#addWorker

addWorker 在线程池中添加并启动新的 worker 线程，可以为该线程指定第一个任务 firstTask ，也以不指定(firstTask = null)，如果不指定 firstTask 那么 worker 线程启动后将直接从 workQueue 中取任务，详见runWorker()

1. 如果成功添加并启动 worker 线程，返回 true

2. addWorker 返回 false 的几种情况：

a. 线程池处于 STOP、TIDYING、TERMINATED 三种状态之一，线程池将不再接收新的任务也不处已添加的任务，不需要再添加新的 worker 线程

b. 线程池处于 SHUTDOWN 状态时只处理已添加任务不接受新任务，即如果 firstTask 不为空返回 false，新任务添加失败

c. 线程池处于 SHUTDOWN 状态时，如果 workQueue 队列为空，即没有存量任务需要执行，返回 false

d. worker 线程计数超出 corePoolSize or maximumPoolSize 限制(由参数 core 决定)

e. 启动 worker 线程失败

3. 如果启动 worker 线程失败，将会执行移除失败的线程、更新 worker 线程计数器 并 检查线程池状态是否需要更新的流程，详见 tryTerminate()

```
/**
    向线程池中添加一个新的工作线程
    firstTask作为该线程的firstTask，如果 firstTask 为 null 表示添加一个空的工作线程
    core决定是向核心线程池添加还是向最大线程池添加线程
*/
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // 当线程池处于TERMINATED、TIDYING、STOP三种状态之一线程池将不再接收新的任务也处理已添加的任务
        // 当线程池处于SHUTDOWN状态时，只处理已添加任务，不接受新任务，即如果firstTask不空返回false新任务添加失败
        // 当线程池处于SHUTDOWN状态时，workQueue队列为空，即没有存量任务需要执行，返回false
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;
        // 首先更新worker线程计数器
        for (;;) {
            int wc = workerCountOf(c);
            // 如果worker线程数量超出限制，添加失败
            // 核心线程数限制corePoolSize or 最大线程数限制maximumPoolSize 由 core参数决定
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                // 更新计数器成功，跳出外循环
                break retry;
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs)
                // 线程池的状态已被其他线程改变，重试外循环
                continue retry;
```

```
                // else CAS failed due to workerCount change; retry inner loop
            }
        }
        // worker线程是否启动
        boolean workerStarted = false;
        // worker线程是否被添加进线程池
        boolean workerAdded = false;
        Worker w = null;
        try {
            // 实例化worker线程
            w = new Worker(firstTask);
            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());
                    // 当线程池处于RUNNING 状态时允许添加新的worker线程
                    // 当线程池处于SHUTDOWN 状态时允许添加空的worker线程
                    if (rs < SHUTDOWN ||
                        (rs == SHUTDOWN && firstTask == null)) {
                        // 已启动的线程不允许加入到线程池
                        if (t.isAlive()) // precheck that t is startable
                            throw new IllegalThreadStateException();
                        workers.add(w);
                        int s = workers.size();
                        // 记录历史最大工作线程数
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                        workerAdded = true;
                    }
                } finally {
                    mainLock.unlock();
                }
                // 如果成功添加worker线程，启动该线程
                if (workerAdded) {
            // worker线程启动实际上是执行runWorker()方法
                    t.start();
                    workerStarted = true;
                }
            }
        } finally {
            // 如果添加 或 启动worker线程失败，调用addWorkerFailed方法删除失败的线程并转换线程
状态
            if (! workerStarted)
                addWorkerFailed(w);
        }
        return workerStarted;
    }
```

```
private void addWorkerFailed(Worker w) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        if (w != null)
            // 将worker线程从线程池中移除
            workers.remove(w);
        // 计数器减1
        decrementWorkerCount();
        // 调用tryTerminate()方法转换线程池状态
        tryTerminate();
    } finally {
        mainLock.unlock();
    }
}
```

3.3 ThreadPoolExecutor#runWorker

Worker 类 run() 方法实际执行的方法，传入的 Worker 参数是 this

1. 首先执行 firstTask，firstTask 执行完后再从 workQueue 中 getTask() 获取任务，如果 getTask 获取任务结果为 null，那么 Worker 线程将会退出 run 方法死亡

2. 调用的是 task 的 run() 方法，而不是 start() 方法，因此没有线程的切换

3. task 执行前后预设了两个钩子函数 beforeExecute(wt, task) 和 afterExecute(task, thrown)，如有特殊业务需求可继承 ThreadPoolExecutor 实现对应钩子函数

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        // 首先执行firstTask，firstTask执行完后再从workQueue 中获取任务
        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();
            try {
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
            // 调用的是task的run()方法，而不是start()方法，没有线程的切换
                    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);
    }
}
```

3.4 ThreadPoolExecutor#getTask

从 workQueue 中获取任务，根据 allowCoreThreadTimeOut 和线程池中线程数量决定线程等待任是否会超时，如果超时那么 getTask 将返回 null

1. 如果线程池状态为 STOP、TIDYING、TERMINATED 将不再处理已有任务，直接返回 null

2. 如果线程池状态为 SHUTDOWN 但是 workQueue 为空也没有任务需要执行，直接返回 null

3. 如果 worker 线程允许超时，调用 workQueue 的 poll(long timeout, TimeUnit unit)方法，如果列为空会一直阻塞直到队列不为空 或者 超出指定时间 或者 线程被中断，如果没有获取到元素会返回ull；如果 worker 线程不允许超时，调用 take()方法，如果队列为空该方法会一直阻塞，直到队列不空或者线程被中断

4. 如果 getTask() 方法返回 null，worker 线程将退出 run() 方法线程死亡，getTask会提前更新线程数器

```
private Runnable getTask() {
    // 上一次从workQueue中获取任务是否超时
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        // 如果线程池状态为STOP、TIDYING、TERMINATED将不再处理已有任务
        // 如果线程池状态为SHUTDOWN 但是workQueue为空也没有任务需要执行
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            // 返回null当前线程会死亡，线程计数-1
            decrementWorkerCount();
            return null;
        }
```

```
        int wc = workerCountOf(c);

        // Are workers subject to culling?
        // worker线程是否会超时
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        //wc > maximumPoolSize  worker线程总数超出 maximumPoolSize
        //timed && timedOut     worker线程可以超时死亡并且上一次从 workQueue 中获取任务超时
        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            // 返回null当前线程会死亡，线程计数-1
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            // 如果worker线程允许超时，调用workQueue的poll(long timeout, TimeUnit unit)方法，
            //     如果队列为空会一直阻塞直到超出指定时间或者队列不为空或者线程被中断，如果没有
取到元素会返回null
            // 如果worker线程不允许超时，调用take()方法，如果队列为空该方法会一直阻塞，直到队
不为空或者线程被中断
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}
```

3.5 ThreadPoolExecutor#tryTerminate

这个方法在任何可能导致线程池终止的动作后执行：比如减少 wokerCount 或 SHUTDOWN 状态下队列中移除任务。

1. 允许 terminate 的两种状态

a. 线程池当前处于 STOP 状态

b. 线程池当前处于 SHUTDOWN 状态并且 workQueue 为空

2. 如果线程池中还有线程正在运行，中断其中一个线程

3. 先将线程池状态更新到 TIDYING 状态，执行 terminated()钩子函数完毕之后再将线程池状态更新到 TERMINATED 状态

```
final void tryTerminate() {
    for (;;) {
        int c = ctl.get();
        // 如果线程池的状态为RUNNING、TIDYING、TERMINATED之一不允许直接terminate
        // 如果线程池的状态为SHUTDOWN 但是workQueue不为空，也不允许直接terminate
```

```
        // 即允许terminate的状态为STOP 以及workQueue为空并且状态为SHUTDOWN
        if (isRunning(c) ||
            runStateAtLeast(c, TIDYING) ||
            (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
            return;
        // 如果线程池中还有线程正在运行，中断其中一个线程
        if (workerCountOf(c) != 0) { // Eligible to terminate
            interruptIdleWorkers(ONLY_ONE);
            return;
        }

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        // 此时线程池全部线程都已死亡，并且线程池处于STOP状态 或者 处于SHUTDOWN并且work
ueue为空的状态
        try {
            // 更新线程池状态为TIDYING
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {
                    // 调用钩子函数，默认实现为空
                    terminated();
                } finally {
                    // 调用terminated()钩子函数完毕后更新线程池状态为TERMINATED
                    ctl.set(ctlOf(TERMINATED, 0));
                    termination.signalAll();
                }
                return;
            }
        } finally {
            mainLock.unlock();
        }
        // else retry on failed CAS
    }
}
```

3.6 ThreadPoolExecutor#shutdown

```
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 检查是否有停止线程池权限
        checkShutdownAccess();
        // 将线程池状态更改成 SHUTDOWN
        advanceRunState(SHUTDOWN);
        // 中断空闲线程
        interruptIdleWorkers();
        // 执行钩子函数
        onShutdown(); // hook for ScheduledThreadPoolExecutor
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
```

```
}
```

3.7 ThreadPoolExecutor#hutdownNow

```
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 检查是否有停止线程池权限
        checkShutdownAccess();
        // 将线程池状态更改成 STOP
        advanceRunState(STOP);
        // 中断空闲线程
        interruptWorkers();
        // 提取所有没有开始执行的任务
        tasks = drainQueue();
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
    return tasks;
}
```

4. submit & invoke 方法

父类 AbstractExecutorService 中的 submit 和 invoke 两类方法都是在 execute 方法的基础上封装一个 Future 类的返回值，其作用就是方便获取任务提交到线程池中之后的执行结果，Future 的具体现原理可参考 java.util.concurrent.Future

4.1 AbstractExecutorService#submit(Callable<T>)

```
    public <T> Future<T> submit(Callable<T> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task);
        execute(ftask);
        return ftask;
    }
```