线程池ThreadPoolExecutor代码分析

ThreadPoolExecutor初始化

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    //核心线程数
    this.corePoolSize = corePoolSize;
    //最大线程数
    this.maximumPoolSize = maximumPoolSize;
    //请求等待队列
    this.workQueue = workQueue;
    //线程存活时间
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    //线程创建工厂类
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

可以看出ThreadPoolExecutor主要包括corePoolSize 、maximumPoolSize 、workQueue 、keepAliveTime 、threadFactory 属性。以及一个很重要的属性ctl，AtomicInteger类型，原子数，保证原子操作，下面是对该属性注释的部分解释。

```java
/**
     * 整个线程池的控制状态，包含了两个属性：有效线程的数量、线程池的状态（runState）。
     *   workerCount,有效线程的数量
     *   runState,   线程池的状态
     *
     * ctl 包含32位数据，低29位存线程数，高3位存runState,这样runState有5个值：
     *
     *   RUNNING:  接受新任务，处理任务队列中的任务
     *   SHUTDOWN: 不接受新任务，处理任务队列中的任务
     *   STOP:     不接受新任务，不处理任务队列中的任务
     *   TIDYING:  所有任务完成，线程数为0，然后执行terminated()
     *   TERMINATED: terminated() 已经完成
     * 具体值：
     * RUNNING:-536870912
     * SHUTDOWN:0
     * STOP:536870912
     * TIDYING:1073741824
     * TERMINATED:1610612736
     */
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
  // Packing and unpacking ctl
   //获取runState值，线程池的运行状态
    private static int runStateOf(int c)     { return c & ~CAPACITY; }
    //获取workerCount值，有效线程的数量
```

```java
private static int workerCountOf(int c)  { return c & CAPACITY; }

//将运行状态和线程池数组合成新的ctl值。

private static int ctlOf(int rs, int wc) { return rs | wc; }

//是否运行中

private static boolean isRunning(int c) {

    return c < SHUTDOWN;

}
```

- corePoolSize :
  这些线程一直存活，就是只要当前线程数小于corePoolSize ，那么就会添加，而且就算当前没有任务，只要线程数不大于当前corePoolSize ，那么这些线程就会一直存活，当然如果调用allowCoreThreadTimeOut（true）方法，那么这些线程在没有任务的时候也会释放掉。
- maximumPoolSize ：
  最大线程数，顾名思义就算当前线程池所持有的最多线程，如果超出这个数就会报异常。
- workQueue ：
  请求等待队列，当当前线程数不小于corePoolSize 时，而workQueue 队列没有满，那么这时就会把请求放到workQueue 队列中，等待执行。
- keepAliveTime ：
  线程等待存活时间，也就是当线程闲置下来时等待下次任务最长时间，默认情况下，这时对核心线程之外的线程的处理，也就是大于corePoolSize 的线程等待时间，当调用allowCoreThreadTimeOut（true）方法，如果当前没有任务，核心线程也会有存活时间。
- threadFactory ：
  线程创建工厂类，创建线程，一般就是调用new Thread()创造线程，当然也可以自己继承Thread，添加自己需要的属性以及操作。

```java
public class ThreadPoolExecutor extends AbstractExecutorService {
    // 线程池的控制状态（用来表示线程池的运行状态（整形的高3位）和运行的worker数量（低29位））
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    // 29位的偏移量
    private static final int COUNT_BITS = Integer.SIZE - 3;
    // 最大容量（2^29 - 1）
    private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

    // runState is stored in the high-order bits
    // 线程运行状态，总共有5个状态，需要3位来表示（所以偏移量的29 = 32 - 3）
    private static final int RUNNING    = -1 << COUNT_BITS;
    private static final int SHUTDOWN   =  0 << COUNT_BITS;
    private static final int STOP       =  1 << COUNT_BITS;
    private static final int TIDYING    =  2 << COUNT_BITS;
    private static final int TERMINATED =  3 << COUNT_BITS;
    // 阻塞队列
    private final BlockingQueue<Runnable> workQueue;
    // 可重入锁
    private final ReentrantLock mainLock = new ReentrantLock();
    // 存放工作线程集合
    private final HashSet<Worker> workers = new HashSet<Worker>();
    // 终止条件
    private final Condition termination = mainLock.newCondition();
    // 最大线程池容量
    private int largestPoolSize;
    // 已完成任务数量
    private long completedTaskCount;
    // 线程工厂
    private volatile ThreadFactory threadFactory;
    // 拒绝执行处理器
    private volatile RejectedExecutionHandler handler;
    // 线程等待运行时间
    private volatile long keepAliveTime;
    // 是否运行核心线程超时
    private volatile boolean allowCoreThreadTimeOut;
    // 核心池的大小
    private volatile int corePoolSize;
    // 最大线程池大小
    private volatile int maximumPoolSize;
    // 默认拒绝执行处理器
    private static final RejectedExecutionHandler defaultHandler =
        new AbortPolicy();
    //
    private static final RuntimePermission shutdownPerm =
        new RuntimePermission("modifyThread");

}
```

添加任务以及执行

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        /*
    * 进行下面三步
    *
    * 1. 如果运行的线程小于corePoolSize,则尝试使用用户定义的Runnalbe对象创建一个新的线程
    *      调用addWorker函数会原子性的检查runState和workCount，通过返回false来防止在不应
    *      该添加线程时添加了线程
    * 2. 如果一个任务能够成功入队列，在添加一个线城时仍需要进行双重检查（因为在前一次检查后
    *      该线程死亡了），或者当进入到此方法时，线程池已经shutdown了，所以需要再次检查状态，
    *     若有必要，当停止时还需要回滚入队列操作，或者当线程池没有线程时需要创建一个新线程
    * 3. 如果无法入队列，那么需要增加一个新线程，如果此操作失败，那么就意味着线程池已经shut
    *      down或者已经饱和了，所以拒绝任务
    */
    // 获取线程池控制状态
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) { // worker数量小于corePoolSize
        if (addWorker(command, true)) // 添加worker
            // 成功则返回
            return;
        // 不成功则再次获取线程池控制状态
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) { // 线程池处于RUNNING状态，将命令（用户自定义的Runnable对象）添加进workQueue队列
        // 再次检查，获取线程池控制状态
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command)) // 线程池不处于RUNNING状态，将命令从workQueue队列中移除
            // 拒绝执行命令
            reject(command);
        else if (workerCountOf(recheck) == 0) // worker数量等于0
            // 添加worker
            addWorker(null, false);
    }
    else if (!addWorker(command, false)) // 添加worker失败
        // 拒绝执行命令
        reject(command);
    }
```

在这里一直调用了addWorker方法，可见该方法的重要性。

```java
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) { // 外层无限循环
        // 获取线程池控制状态
        int c = ctl.get();
        // 获取状态
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&            // 状态大于等于SHUTDOWN，初始的ctl为RUNNING，小于SHUTDOWN
            ! (rs == SHUTDOWN &&         // 状态为SHUTDOWN
               firstTask == null &&         // 第一个任务为null
               ! workQueue.isEmpty()))      // worker队列不为空
            // 返回
            return false;

        for (;;) {
            // worker数量
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||                                // worker数量大于等于最大容量
                wc >= (core ? corePoolSize : maximumPoolSize))    // worker数量大于等于核心线程池大小或者最大线程池大小
                return false;
            if (compareAndIncrementWorkerCount(c))                 // 比较并增加worker的数量
                // 跳出外层循环
                break retry;
            // 获取线程池控制状态
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs) // 此次的状态与上次获取的状态不相同
                // 跳过剩余部分，继续循环
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }

    // worker开始标识
    boolean workerStarted = false;
    // worker被添加标识
    boolean workerAdded = false;
    //
    Worker w = null;
    try {
        // 初始化worker
        w = new Worker(firstTask);
        // 获取worker对应的线程
        final Thread t = w.thread;
        if (t != null) { // 线程不为null
```

```
            // 线程池锁
            final ReentrantLock mainLock = this.mainLock;
            // 获取锁
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                // 线程池的运行状态
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
// 小于SHUTDOWN
                    (rs == SHUTDOWN && firstTask == null)) {
// 等于SHUTDOWN并且firstTask为null
                    if (t.isAlive()) // precheck that t is startable
 // 线程刚添加进来，还未启动就存活
                        // 抛出线程状态异常
                        throw new IllegalThreadStateException();
                    // 将worker添加到worker集合
                    workers.add(w);
                    // 获取worker集合的大小
                    int s = workers.size();
                    if (s > largestPoolSize) // 队列大小大于largestPoolSize
                        // 重新设置largestPoolSize
                        largestPoolSize = s;
                    // 设置worker已被添加标识
                    workerAdded = true;
                }
            } finally {
                // 释放锁
                mainLock.unlock();
            }
            if (workerAdded) { // worker被添加
                // 开始执行worker的run方法
                t.start();
                // 设置worker已开始标识
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted) // worker没有开始
            // 添加worker失败
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

可以看出addWorker方法主要是生成新的线程，而线程的重用则在Worker类中实现。

```java
private final class Worker
        extends AbstractQueuedSynchronizer
        implements Runnable
    {
        /**
         * This class will never be serialized, but we provide a
         * serialVersionUID to suppress a javac warning.
         */
        private static final long serialVersionUID = 6138294804551838833
L;

        /** worker持有的线程 */
        final Thread thread;
        /** worker正在执行的任务 ，可能为null. */
        Runnable firstTask;
        /** Per-thread task counter */
        volatile long completedTasks;

        /**
         * 创建Worker时会同时创建一个新线程.
         * @param firstTask the first task (null if none)
         */
        Worker(Runnable firstTask) {
            setState(-1); // inhibit interrupts until runWorker
            this.firstTask = firstTask;
          //把Worker传递给新建的线程，当线程执行是会调用Worker的run方法。
            this.thread = getThreadFactory().newThread(this);
        }

        /** 线程执行时会调用该方法 */
        public void run() {
            runWorker(this);
        }

        protected boolean isHeldExclusively() {
            return getState() != 0;
        }

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

        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }
```

```
        public void lock()        { acquire(1); }
        public boolean tryLock()  { return tryAcquire(1); }
        public void unlock()      { release(1); }
        public boolean isLocked() { return isHeldExclusively(); }

        void interruptIfStarted() {
            Thread t;
            if (getState() >= 0 && (t = thread) != null && !t.isInterrupt
ed()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                }
            }
        }
    }
```

最后执行runWorker方法：

```
final void runWorker(Worker w) {
    // 获取当前线程
    Thread wt = Thread.currentThread();
    // 获取w的firstTask
    Runnable task = w.firstTask;
    // 设置w的firstTask为null
    w.firstTask = null;
    // 释放锁（设置state为0，允许中断）
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        while (task != null || (task = getTask()) != null) { // 任务不为nu
ll或者阻塞队列还存在任务
            // 获取锁
            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) ||    // 线程池的运行状态
至少应该高于STOP
                 (Thread.interrupted() &&                // 线程被中断
                  runStateAtLeast(ctl.get(), STOP))) &&    // 再次检查，线程
池的运行状态至少应该高于STOP
                !wt.isInterrupted())                     // wt线程（当前线
程）没有被中断
                wt.interrupt();                            // 中断wt线程
（当前线程）
            try {
                // 在执行之前调用钩子函数
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    // 运行给定的任务
                    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 {
                    // 执行完后调用钩子函数
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                // 增加给worker完成的任务数量
                w.completedTasks++;
                // 释放锁
                w.unlock();
```

```
            }
        }
        completedAbruptly = false;
    } finally {
        // 处理完成后，调用钩子函数
        processWorkerExit(w, completedAbruptly);
    }
}
```

线程获取待执行任务方法，从上面的分析可以看出，这里才是线程重用的关键，所以下面分析getTask方法。

```java
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) { // 无限循环，确保操作成功
        // 获取线程池控制状态
        int c = ctl.get();
        // 运行的状态
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) { // 大于等于SHUTDOWN（表示调用了shutDown）并且（大于等于STOP（调用了shutDownNow）或者worker阻塞队列为空）
            // 减少worker的数量
            decrementWorkerCount();
            // 返回null，不执行任务
            return null;
        }
        // 获取worker数量
        int wc = workerCountOf(c);

        // Are workers subject to culling?
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize; // 是否允许coreThread超时或者workerCount大于核心大小

        if ((wc > maximumPoolSize || (timed && timedOut))     // worker数量大于maximumPoolSize
            && (wc > 1 || workQueue.isEmpty())) {            // workerCount大于1或者worker阻塞队列为空（在阻塞队列不为空时，需要保证至少有一个wc）
            if (compareAndDecrementWorkerCount(c))            // 比较并减少workerCount
                // 返回null，不执行任务，该worker会退出
                return null;
            // 跳过剩余部分，继续循环
            continue;
        }

        try {
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) : // 等待指定时间
                workQueue.take();                                     // 一直等待，直到有元素
            if (r != null)
                return r;
            // 等待指定时间后，没有获取元素，则超时
            timedOut = true;
        } catch (InterruptedException retry) {
            // 抛出了被中断异常，重试，没有超时
            timedOut = false;
        }
```

```
    }
}
```

结束线程池

ThreadPoolExecutor有两个结束的方法shutdown、shutdownNow。shutdown是把线程池状态转为SHUTDOWN，这时等待队列中的任务可以继续执行；
shutdownNow方法是把线程池状态转为SHUTDOWN，这时等待队列中的任务不可以继续执行，只能执行已经执行的任务；

```
public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            //把线程池状态改为SHUTDOWN
            advanceRunState(SHUTDOWN);
           // 中断所有空闲线程
            interruptIdleWorkers();
            onShutdown(); // hook for ScheduledThreadPoolExecutor
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
    }
    public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
           //把线程池状态改为STOP
            advanceRunState(STOP);
            // 中断所有空闲线程
            interruptWorkers();
            // 返回队列中还没有被执行的任务
            tasks = drainQueue();
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
        return tasks;
    }
```

advanceRunState 改变线程池的状态

```
//把线程池状态改为目标状态targetState
 private void advanceRunState(int targetState) {
        for (;;) {
            int c = ctl.get();
            if (runStateAtLeast(c, targetState) ||
                ctl.compareAndSet(c, ctlOf(targetState, workerCountOf
(c))))
                break;
        }
    }
```

interruptIdleWorkers中断线程

```
private void interruptIdleWorkers() {
        interruptIdleWorkers(false);
    }
 private void interruptIdleWorkers(boolean onlyOne) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers) {
                Thread t = w.thread;
                //线程没有被中断并且Worker 正在获取任务中，就是空闲中。线程中断
                if (!t.isInterrupted() && w.tryLock()) {
                    try {
                        t.interrupt();
                    } catch (SecurityException ignore) {
                    } finally {
                        w.unlock();
                    }
                }
                if (onlyOne)
                    break;
            }
        } finally {
            mainLock.unlock();
        }
    }
```

getTask方法中可以看出如果线程池处于STOP已经以上状态时不会继续获取任务，而是尝试中断线程，这也就是shutdown、shutdownNow的区别。我查找资料发现这些内容：
1、ReentrantLock.lockInterruptibly允许在等待时由其它线程调用等待线程的Thread.interrupt方法来中断等待线程的等待而直接返回，这时不用获取锁，而会抛出一个InterruptedException。

然后我们进入Executors的生成方法，发现使用的是LinkedBlockingQueue类，而LinkedBlockingQueue的take()方法如下

```java
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
   //调用了lockInterruptibly方法
    takeLock.lockInterruptibly();
    try {
        while (count.get() == 0) {
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

所以当线程在获取任务阻塞时，如果该线程被调用了interupt方法，则该线程释放，所以说释放空闲线程。