一、线程池执行任务

execute方法

```java
//原子整型由（线程池运行状态、工作线程数）组成 高3位代表线程池的状态，低29位代表工作线程的数量
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
//COUNT_BITS = 29
private static final int COUNT_BITS = Integer.SIZE - 3;
//工作线程的最大数量为2^29 - 1
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

// runState is stored in the high-order bits
//线程池的状态表示，3bits表示状态
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;

// Packing and unpacking ctl
//获取线程池的状态
private static int runStateOf(int c)     { return c & ~CAPACITY; }
//工作线程数量
private static int workerCountOf(int c)  { return c & CAPACITY; }
//将线程池状态和工作线程数量表示为一个int型数
private static int ctlOf(int rs, int wc) { return rs | wc; }

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 prevents false alarms that would add
     * threads when it shouldn't, by returning false.
     *
     * 2. If a task can be successfully queued, then we still need
     * to double-check whether we should have added a thread
     * (because existing ones died since last checking) or that
     * the pool shut down since entry into this method. So we
     * recheck state and if necessary roll back the enqueuing if
     * stopped, or start a new thread if there are none.
     *
     * 3. If we cannot queue task, then we try to add a new
     * thread.  If it fails, we know we are shut down or saturated
     * and so reject the task.
     */
    //通过ctl获取当前由线程池状态和当前工作现线程数量表示的int数
    int c = ctl.get();
    //如果当前的工作线程小于核心线程，则为当前的任务创建一个核心线程进行任务处理
    if (workerCountOf(c) < corePoolSize) {
        //为当前的任务创建一个核心线程，创建成功则返回
        if (addWorker(command, true))
            return;
        //创建失败需要重新获取当前的状态，为什么会创建失败呢
        //1.由于并发操作，可能在创建核心线程时，另一个并发线程已经创建了一个核心线程导致，核心
线程数已经够了
        //2.可能是线程池已经被关闭，导致了核心线程创建失败。
        c = ctl.get();
    }
    //1.判断线程池状态是否是RUNNING,如果是RUNNING状态则将任务添加到任务队列当中
    if (isRunning(c) && workQueue.offer(command)) {
        //提供双检，如果当前是RUNNING,说明状态没有改变，如果不是RUNNING,那么可能状态已经被改
变了，c保留了旧的状态值，
        //在前面的c=ctl.get()到isRUNNING(c)这个过程状态被修改了，
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //线程池关闭，或者线程池未关闭但是队列已满
    else if (!addWorker(command, false))
        reject(command);
}
```

addWorker方法

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        //如果当前状态时RUNNING或者是SHUTDOWN,firstTask==null,但是任务队列中不为空，则不
会进入if代码块
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) {
            //判断工作线程是否超出最大工作线容量，或者如果是核心线程，判断是否超出核心线程数，
非核心时判断是否超过                    //maximumPoolSize
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            //如果修改成功，退出循环
            if (compareAndIncrementWorkerCount(c))
                break retry;
            //1.可能状态被修改
            //2.可能线程数量被修改
            c = ctl.get();  // Re-read ctl
            //如果线程状态被修改，则需要跳到retry,重新执行外层循环
            //如果是线程数量被修改了，则在内层循环中进行
            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 {
        //Worker即使AQS也是Runnable
        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());
                //线程池状态是RUNNING或者SHUTDOWN并且firstTask==null，检查一下新创建的线程是否启动，如果启动就抛异常
                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN &&  == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    workers.add(w);//将当前新建的Worker加入集合中（执行单元）
                    int s = workers.size();
                    //统计一下当前线程数量最大值
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            if (workerAdded) {
                //线程执行过程出现异常不会影响后续执行，线程执行任务过程中，内部捕捉或者忽略
                t.start();//创建的线程开始执行任务，调用run()->runWork()
                workerStarted = true;
            }
        }
    } finally {
        //线程创建失败或者线程创建了但是提前启动了抛出了异常
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

addWorkerFailed方法

```
private void addWorkerFailed(Worker w) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //工作线程启动失败，那么就需要把该线程从集合中移除，并减少线程数量
        if (w != null)
            workers.remove(w);
        decrementWorkerCount();
        tryTerminate();
    } finally {
        mainLock.unlock();
    }
}
```

runWorker方法

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        //循环获取任务
        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
            //如果当前线程池状态是STOP，并且当前线程未被中断;或者是当前线程池已经STOP，并且
当前线程已被中断
            //则中断当前线程，打了个标记
            //如果当前的状态<STOP，并且如果线程被中断过，则清楚中断标记
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
            try {
                //钩子函数，任务执行前可以做一些统计、日志等工作
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    //执行任务，如果任务执行过程中出现了异常，则会跳过
completedAbruptly=false;表示
                    //任务已完成（出现异常也算是完成），
                    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 {
                    //钩子函数，任务完成后可以做一些统计、日志等工作
                    afterExecute(task, thrown);

                }
            } finally {
                //正常完成任务，或者执行任务出现异常也算完成任务
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        //如果获取任务为null,则会执行到这里
        //如果任务在执行过程中出现了异常，则不会执行到这里，也就是completedAbruptly最后是
true
        completedAbruptly = false;
    } finally {
        //处理线程退出的操作
        processWorkerExit(w, completedAbruptly);
    }
}
```

getTask方法

```
private Runnable getTask() {
    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.
        //如果线程池状态是SHUTDOWN并且任务队列为空，则减少线程数量，并返回null,表示没有获取
到任务
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }
        //走到这里说明线程池状态是RUNNING或者是SHUTDOWN并且队列非空
        int wc = workerCountOf(c);

        // Are workers subject to culling?
        //如果allowCoreThreadTimeOut==true,或者wc>corePoolSize,则timed=true
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
        //wc>maximumPoolSize,则可能是修改了maximumPoolSize
        if ((wc > maximumPoolSize || (timed && timedOut))
            //走到这里，那么wc>1时需要减少线程或者wc==1,队列空了需要减少线程
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            //1.如果是timed=true，那么线程以最大keepAliveTime的等待时间从队列中获取任务，
如果获取到任务就返回，如果没有获             取到任务就返回null
            //2.如果timed==false，那么就阻塞式从队列中获取任务，
```

```
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }
```

processWorkerExit方法

processWorkerExit方法是任务被执行完成后，线程继续获取任务过程中，没有得到任务，则会进行线程退出操作的处理。

在处理线程退出过程中，可能是线程没有获取到任务而退出，也可能是任务在执行过程中出现了异常而退出。

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    //如果线程在执行任务过程中出现了异常，则completedAbruptly是true
    //那么线程数量没有减少，在这里减少一个线程；正常退出的线程由于getTask方法会处理减少线程的
操作，这里不会执行
    if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
        decrementWorkerCount();

    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //统计当前线程完成的任务数量，并添加到线程池完成的任务数量
        completedTaskCount += w.completedTasks;
        //从workers中移除当前工作线程，移除操作需要获取锁，进行多线程操作的同步
        workers.remove(w);
    } finally {
        mainLock.unlock();
    }
    //当前线程退出的时候会尝试关闭线程池，因为可能之前调用过shutdown方法，而当前线程当时还在执
行任务，所以当前线程在退出的
    //时候会尝试关闭线程池
    tryTerminate();

    int c = ctl.get();
    //如果关闭失败，也就是线程池的状态还处于STOP状态
    if (runStateLessThan(c, STOP)) {
        //如果当前线程是正常退出进入
        if (!completedAbruptly) {
            //获取线程池应该存在的线程最小数量
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
            if (min == 0 && ! workQueue.isEmpty())
                min = 1;
            //如果线程数量已经满足了最小线程数量，直接返回了
            //如果线程数量小于最小线程数量，则会执行下面的addWorker方法，添加一个线程，
            if (workerCountOf(c) >= min)
                return; // replacement not needed
        }
        //当前线程异常退出的话，尝试补偿一个线程
```

```
            //当前补偿的线程，可能是非核心线程，也可能是核心线程
            //但是要重点强调，所谓核心线程和非核心线程并不是说某个线程被打上了核心和非核心的标签，而是线程池中应该有一个
            //corePoolSize的线程的数量
            //打个比喻，我有20个充电宝，但现在就先提供5个固定的充电宝（线程），和10把椅子（队列），如果来了一个顾客（任务）要使用充电宝，先从5个中拿走一个使用，后续来的客户也是如此，如果5个充电宝都在被使用，后续顾客需要到椅子中去排队了，如果有顾客已经使用完毕，那么他把充电宝归还，唤醒等待排序中的顾客使用充电宝，如果椅子已经坐满了顾客，那后续如果在由顾客到来的话，那么就尝试添加一个充电宝给他使用，如果在使用完毕后，那个多余的充电宝如果在一定时间没有使用就回收了。
            //直接尝试添加一个线程，可能是当前线程异常退出（不管是不是核心线程）直接按照非核心（false）添加一个
            //也可能是
            addWorker(null, false);
        }
    }
```

二、关闭线程池

shutdown方法

shutdown方法会将当前线程池的状态修改为SHUTDOWN,当然如果线程池的状态已经>SHUTDOWN了，就没必要设置了

shutdown方法并不会中断正在执行任务的，使用tryLock保证正在执行任务的线程不会被打上中断标记，因为Worker即是AQS又是Runnable,所以在执行任务的时候，使用w.lock获取锁，故shutdown方法在使用tryLock的时候会失败

shutdown方法将线程池状态设置SHUTDOWN后便不在接受新的任务，但是会将队列中的任务执行完后再关闭线程池。

```java
public void shutdown() {
    //获取锁，需要操作workerset集合
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //shutdown访问权限检查，不是很了解
        checkShutdownAccess();
        //将当前线程池的状态修改为SHUTDOWN状态
        advanceRunState(SHUTDOWN);
        //打断空闲的线程，其实就是添加了一个中断的标记
        interruptIdleWorkers();
        onShutdown(); // hook for ScheduledThreadPoolExecutor
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
}
```

shutdownNow方法

shutdownNow方法会将当前线程池的状态修改为STOP,当然如果线程池的状态已经>STOP了，就没必要设置了

shutdownNow方法不再接受新的任务，会尝试打断正在执行任务的工作线程（也可能没有，如果线程不响应中断）和空闲线程，并且返回任务队列中所有未执行的任务（和shutdown的区别）

shutdownNow方法在尝试打断正在执行的工作线程，可能不会中断其运行操作（线程可能不响应中断）

```java
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //同shutdown方法
        checkShutdownAccess();
        advanceRunState(STOP);
        //打断正在执行任务的工作线程和空闲线程
        interruptWorkers();
        //返回队列中所有未执行的任务
        tasks = drainQueue();
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
    return tasks;
}
```

interruptWorkers方法

```java
private void interruptWorkers() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //直接就来打断，不使用w.tryLock()
        for (Worker w : workers)
            w.interruptIfStarted();
    } finally {
        mainLock.unlock();
    }
}
```

interruptIfStarted方法

```java
void interruptIfStarted() {
    Thread t;
    //Worker的state>=0时才允许打断，创建Worker的时候state==-1,不允许打断
    if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
        try {
            //中断当前线程，打上中断标记
            t.interrupt();
        } catch (SecurityException ignore) {
        }
    }
}
}
```

advanceRunState方法

```
private void advanceRunState(int targetState) {
    for (;;) {
        //获取当前状态，如果当前状态大于targetState状态，则退出，如果当前状态
        //小于targetState状态，那么将当前状态修改为targetState状态
        int c = ctl.get();
        if (runStateAtLeast(c, targetState) ||
            ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c))))
            break;
    }
}
```

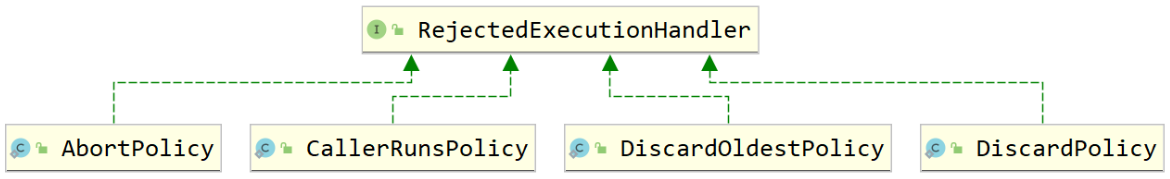
interruptIdleWorkers方法

```
private void interruptIdleWorkers() {
    interruptIdleWorkers(false);
}
private void interruptIdleWorkers(boolean onlyOne) {
    //打断所有的空闲线程
    //对workers集合要先加锁才能访问，防止并发操作造成错误
    final ReentrantLock mainLock = this.mainLock;//又加锁，不懂
    mainLock.lock();
    try {
        for (Worker w : workers) {
            Thread t = w.thread;
            //如果当前worker未被打断，那么需要使用tryLock获取锁，以判断当前线程是否在处理任
务

            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    //当前线程是空闲线程，应该被打断，打上中断标记
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            //如果只要求被打断一个，打断后退出，打断一个是辅助线程池状态是STOP的时候，或者是
SHUTDOWN并且队列是空
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}
```

tryTerminate方法

```
final void tryTerminate() {
    for (;;) {
        int c = ctl.get();
        //如果线程池状态是RUNNING、TIDYING、TERMINATED、或者是SHUTDOWN但是workQueue非空
        //直接return
        if (isRunning(c) ||
            runStateAtLeast(c, TIDYING) ||
            (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
            return;
        //到这里说明线程池状态是SHUTDOWN,并且队列空；或者是STOP
        //如果工作线程数量不是0，那么中断一个线程，打上标记
        if (workerCountOf(c) != 0) { // Eligible to terminate
            interruptIdleWorkers(ONLY_ONE);
            return;
        }

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //CAS修改线程池的状态和工作线程数量
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {
                    //钩子函数
                    terminated();
                } finally {
                    //线程池关闭
                    ctl.set(ctlOf(TERMINATED, 0));
                    //唤醒可能存在的在等待线程池关闭而阻塞在条件队列中的线程
                    termination.signalAll();
                }
                return;
            }
        } finally {
            mainLock.unlock();
        }
        // 使用CAS修改失败，重试，并发可能其他线程已经修改了
    }
}
```

三、拒绝策略



DiscardPolicy拒绝策略

该策略采用直接丢弃的方式进行

```
public static class DiscardPolicy implements RejectedExecutionHandler {
    /**
     * Creates a {@code DiscardPolicy}.
     */
    public DiscardPolicy() { }

    /**
     * Does nothing, which has the effect of discarding task r.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    }
}
```

DiscardOldestPolicy拒绝策略

该策略丢弃进入队列中最久的任务，然后执行当前任务

```
public static class DiscardOldestPolicy implements RejectedExecutionHandler {
    /**
     * Creates a {@code DiscardOldestPolicy} for the given executor.
     */
    public DiscardOldestPolicy() { }

    /**
     * Obtains and ignores the next task that the executor
     * would otherwise execute, if one is immediately available,
     * and then retries execution of task r, unless the executor
     * is shut down, in which case task r is instead discarded.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            //丢弃进入队列最久的那个任务，真惨啊
            e.getQueue().poll();
            //执行当前任务，
            e.execute(r);
        }
    }
}
```

AbortPolicy拒绝策略

该策略直接抛出异常

```
public static class AbortPolicy implements RejectedExecutionHandler {
    /**
     * Creates an {@code AbortPolicy}.
     */
    public AbortPolicy() { }

    /**
     * Always throws RejectedExecutionException.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     * @throws RejectedExecutionException always
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        //抛出异常
        throw new RejectedExecutionException("Task " + r.toString() +
                                             " rejected from " +
                                             e.toString());
    }
}
```

CallerRunsPolicy拒绝策略

该策略由调用者自己执行任务

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {
    /**
     * Creates a {@code CallerRunsPolicy}.
     */
    public CallerRunsPolicy() { }

    /**
     * Executes task r in the caller's thread, unless the executor
     * has been shut down, in which case the task is discarded.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            //如果线程池未关闭，那么由调用者执行
            r.run();
        }
    }
}
```