线程池源码解析：

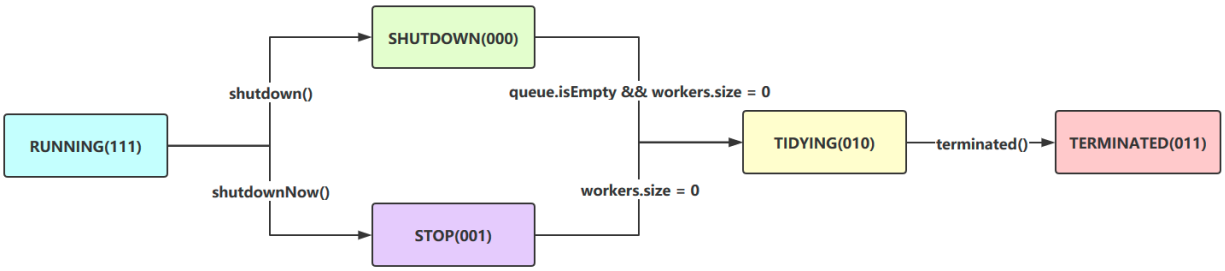
- 线程池内部的线程池状态有几种含义：

```
RUNNING(111)：        表示运行状态，可接受新的任务，也可以处理队列中的任务；
SHUTDOWN(000)：       待关闭状态，不再接受新的任务，但是可以继续处理阻塞队列中的任务；
STOP(001)：           停止状态，不接收新任务也不处理阻塞队列中的任务，并且会尝试结束执行中的任务，当工作线程数为0时，进入TIDYING状态；
TIDYING(010)：        整理状态，此时任务都已经执行完毕，并且也没有工作线程，执行terminated方法后进入TERMINATED状态；
TERMINATED(011)：     终止状态，此时线程池完全终止了，并完成了所有资源的释放；
```

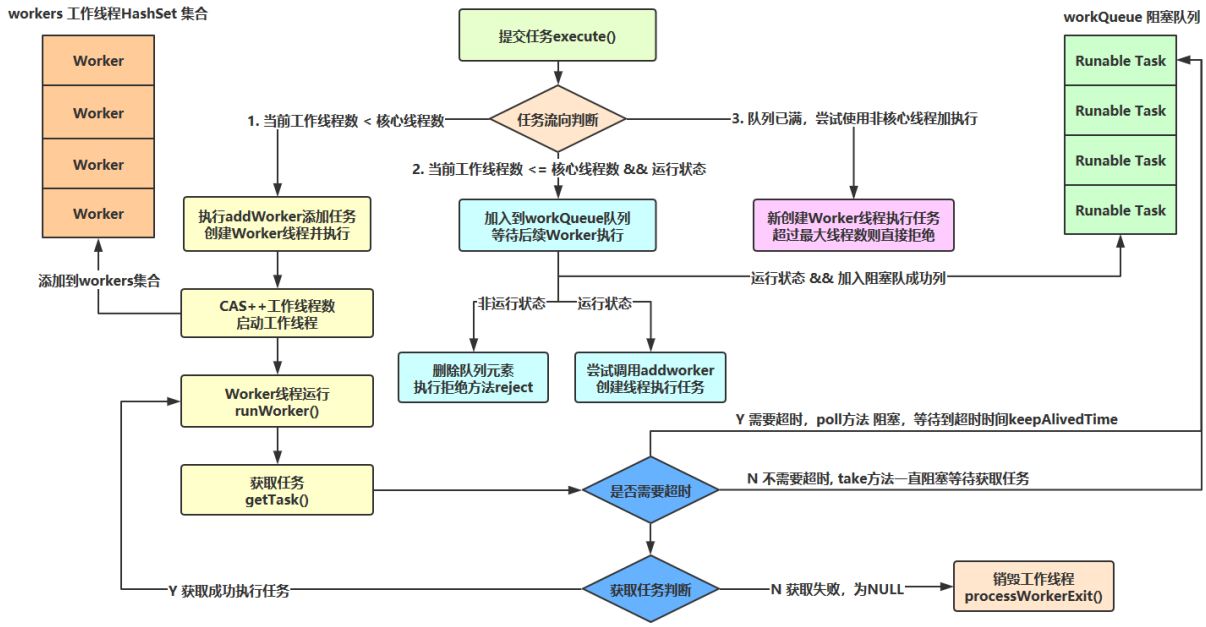
- 线程池核心数据结构

```
(1) BlockingQueue<Runnable> workQueue，任务队列，负责保存任务并将任务交给工作线程处理；
(2) ReentrantLock mainLock，在更新内部数据时要使用的锁，如：线程数量，运行状态，工作线程添加减少等；
(3) Condition termination，用于支持awaitTermination的等待条件；
(4) Worker类：（Worker类是可以重复使用的，一般情况下只有线程池超过空闲时间或异常时才会中断线程）
    class Worker extends AbstractQueuedSynchronizer implements Runnable;
    任务内部包装类，包含当前执行任务和当前执行线程的引用，最终执行任务的是run方法内部的runWorker()方法；
(5) HashSet<Worker> workers，包含所有工作类的集合，Worker只能在持有mainLock的情况下使用(所有的Worker对象都会进入这个集合)；
```

- 线程池状态解析：



- 线程池工作流程解析：



- 线程池源码解析：

```
package com.bfxy.thread.design.concurrent;
import java.util.*;

import com.bfxy.thread.design.concurrent.atomic.AtomicInteger;
import com.bfxy.thread.design.concurrent.locks.AbstractQueuedSynchronizer;
import com.bfxy.thread.design.concurrent.locks.Condition;
import com.bfxy.thread.design.concurrent.locks.ReentrantLock;

/**
 *  $ThreadPoolExecutor
 * @author hezhuo.bai-JiFeng
 * @since 2020年2月12日 下午9:06:49
 */
public class ThreadPoolExecutor extends AbstractExecutorService {

    //  ctl 这个变量用于保存当前容器的运行状态和容器大小，并将存在于ThreadPoolExecutor的整个生命周
期
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    //  Integer.SIZE = 32 bit, COUNT_BITS = 29
    private static final int COUNT_BITS = Integer.SIZE - 3;
    //  高三位表示运行状态，低29位表示容器的当前大小(也就是工作线程数的大小)；
    //  从而实现了使用一个int同时保存两种信息的功能；
    //  CAPACITY相当于掩码
    private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

    // runState is stored in the high-order bits

    /** 运行状态，该状态下线程池可以接受新的任务，也可以处理阻塞队列中的任务     */
    private static final int RUNNING    = -1 << COUNT_BITS;     //  111

    /** 待关闭状态，不再接受新的任务，继续处理阻塞队列中的任务；  */
```

```
    private static final int SHUTDOWN    =  0 << COUNT_BITS;     //  000

    /**
     * 停止状态，不接收新任务也不处理阻塞队列中的任务，并且会尝试结束执行中的任务；
     * 当工作线程数为0时，进入 TIDYING 状态；
     */
    private static final int STOP        =  1 << COUNT_BITS;     //  001

    /**
     * 整理状态，此时任务都已经执行完毕，并且也没有工作线程；
     * 执行terminated方法后进入 TERMINATED 状态
     */
    private static final int TIDYING     =  2 << COUNT_BITS;     //  010

    /** 终止状态，此时线程池完全终止了，并完成了所有资源的释放；  */
    private static final int TERMINATED =  3 << COUNT_BITS;     //  011

    // Packing and unpacking ctl
    //  ~CAPACITY表示取反，获得c的高三位，得到运行状态
    private static int runStateOf(int c)     { return c & ~CAPACITY; }

    //  用来获得c的低29位，获得当前工作线程数量的大小
    private static int workerCountOf(int c)  { return c & CAPACITY; }

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

    /**
     * <B>方法名称：</B>runStateLessThan<BR>
     * <B>概要说明：</B>根据运行状态值进行比对，前者小于后者<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午8:00:59
     * @param c 状态c
     * @param s 状态s
     * @return c < s
     */
    private static boolean runStateLessThan(int c, int s) {
        return c < s;
    }

    /**
     * <B>方法名称：</B>runStateAtLeast<BR>
     * <B>概要说明：</B>根据运行状态值进行比对，前者大于等于后者<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午7:59:33
     * @param c 状态c
     * @param s 状态s
     * @return c >= s
     */
    private static boolean runStateAtLeast(int c, int s) {
        return c >= s;
    }

    private static boolean isRunning(int c) {
```

```
        //  RUNNING = 0, c < 0 ;
        return c < SHUTDOWN;
    }

    /**
     * <B>方法名称：</B>compareAndIncrementWorkerCount<BR>
     * <B>概要说明：</B>操作工作线程数CAS递增<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午8:11:31
     * @param expect 期望的原始值
     * @return CAS操作结果
     */
    private boolean compareAndIncrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect + 1);
    }

    /**
     * <B>方法名称：</B>compareAndDecrementWorkerCount<BR>
     * <B>概要说明：</B>操作工作线程数CAS递减<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午8:11:56
     * @param expect 期望的原始值
     * @return CAS操作结果
     */
    private boolean compareAndDecrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect - 1);
    }

    /**
     * <B>方法名称：</B>decrementWorkerCount<BR>
     * <B>概要说明：</B>do while 方式递减，也就是一定会执行递减成功，否则一直做CAS操作<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午8:13:42
     */
    private void decrementWorkerCount() {
        do {} while (! compareAndDecrementWorkerCount(ctl.get()));
    }

    /** workQueue 任务队列，负责保存任务并将任务交给工作线程处理  */
    private final BlockingQueue<Runnable> workQueue;

    /** mainLock 在更新内部数据时要使用的锁(如：线程数量，运行状态，工作线程集等)    */
    private final ReentrantLock mainLock = new ReentrantLock();

    /**
     *  用于支持awaitTermination的等待条件
     */
    private final Condition termination = mainLock.newCondition();

    /**
     *  包含所有工作类的集合，只能在持有mainLock的情况下使用
     */
    private final HashSet<Worker> workers = new HashSet<Worker>();
```

```
    /**
     *  记录曾经达到的最大的线程数量
     */
    private int largestPoolSize;

    /**
     *  统计任务完成数量的计数器，在工作线程终止的时候才会更新
     */
    private long completedTaskCount;

    /** 线程工厂，用于创建新线程    */
    private volatile ThreadFactory threadFactory;

    /**
     *  当线程池饱和或者关闭时，负责处理新来任务的处理器，称为拒绝任务处理器
     */
    private volatile RejectedExecutionHandler handler;

    /**
     *  空闲回收线程池中线程时间
     */
    private volatile long keepAliveTime;

    /**
     *  如果为false(默认情况下)，核心线程就算空闲也会一直存活；
     *  如果为true，等待任务的核心线程会使用keepAliveTime作为超时时间；如果超时，线程被回收；
     */
    private volatile boolean allowCoreThreadTimeOut;

    /**
     *  核心线程数量，只能在持有mainLock的情况下修改，volatile可以保证可见性；
     */
    private volatile int corePoolSize;

    /**
     *  最大线程数量，只能在持有mainLock的情况下修改，volatile可以保证可见性；
     */
    private volatile int maximumPoolSize;

    /**
     *  默认拒绝策略defaultHandler = AbortPolicy
     */
    private static final RejectedExecutionHandler defaultHandler = new AbortPolicy();

    private static final RuntimePermission shutdownPerm = new
RuntimePermission("modifyThread");

    /**
     *  $Worker
     *  任务内部包装类，包含当前执行任务和执行线程，最终执行任务的是run方法内部的runWorker()方法
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:20:34
```

```
     */
    private final class Worker extends AbstractQueuedSynchronizer implements Runnable {

        private static final long serialVersionUID = 6138294804551838833L;

        /** 执行task任务的线程，此线程是由ThreadFactury创建的    */
        final Thread thread;
        /** task任务引用     */
        Runnable firstTask;
        /** 完成的任务数，用于线程池统计      */
        volatile long completedTasks;

        /**
         * <B>构造方法</B>Worker<BR>
         * 每个worker其实就是一个线程(thread)，同时包含了一个任务(firstTask)，即初始化时要被首先执行的任务<BR>
         * 这个Worker对象是以单线程的方式重复执行不同的任务的<BR>
         * @param firstTask
         */
        Worker(Runnable firstTask) {
            //  初始状态-1，防止在调用runWorker()，也就是真正执行task前中断thread
            setState(-1); // inhibit interrupts until runWorker
            this.firstTask = firstTask;
            this.thread = getThreadFactory().newThread(this);
        }

        /** Delegates main run loop to outer runWorker  */
        public void run() {
            //
            runWorker(this);
        }

        // Lock methods
        // The value 0 represents the unlocked state.
        // The value 1 represents the locked state.
        protected boolean isHeldExclusively() {
            return getState() != 0;
        }

        //  获取许可
        protected boolean tryAcquire(int unused) {
            //  获取许可方法，compareAndSetState设置为1
            if (compareAndSetState(0, 1)) {
                //  设置为独占线程
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }

        //  释放许可
        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
```

```
            setState(0);
            return true;
        }

        //  AQS加锁
        public void lock()        { acquire(1); }
        //  AQS尝试加锁
        public boolean tryLock()  { return tryAcquire(1); }
        //  AQS释放锁
        public void unlock()      { release(1); }
        //  是否正处于加锁状态(独占状态)
        public boolean isLocked() { return isHeldExclusively(); }
        //  打断当前线程方法
        void interruptIfStarted() {
            Thread t;
            //  如果当前线程处于空闲状态，并且当前线程不为空，并且不是打断状态；则进行打断当前线程；
            if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                }
            }
        }
    }

    /*
     * Methods for setting control state
     */

    private void advanceRunState(int targetState) {
        for (;;) {
            int c = ctl.get();
            //  将当前线程池运行状态和targetState状态进行比对；
            //  如果c >= s，也就是当前状态大于等于targetState状态，则break
            //  否则执行后面的逻辑比对，CAS操作改变状态
            if (
                    runStateAtLeast(c, targetState) ||
                    ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c)))
                )
                break;
        }
    }

    //  尝试停止线程池
    final void tryTerminate() {
        for (;;) {
            int c = ctl.get();
            if (isRunning(c) ||
                runStateAtLeast(c, TIDYING) ||
                (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
                return;
            if (workerCountOf(c) != 0) { // Eligible to terminate
                interruptIdleWorkers(ONLY_ONE);
```

```
                return;
            }

            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                    try {
                        terminated();
                    } finally {
                        ctl.set(ctlOf(TERMINATED, 0));
                        termination.signalAll();
                    }
                    return;
                }
            } finally {
                mainLock.unlock();
            }
            // else retry on failed CAS
        }
    }

    /*
     * Methods for controlling interrupts to worker threads.
     */

    //  检查是否允许进行Shutdown
    private void checkShutdownAccess() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkPermission(shutdownPerm);
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                for (Worker w : workers)
                    security.checkAccess(w.thread);
            } finally {
                mainLock.unlock();
            }
        }
    }

    //  打断所有工作workers
    private void interruptWorkers() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers)
                w.interruptIfStarted();
        } finally {
            mainLock.unlock();
        }
    }
```

```
//  打断工作workers
private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers) {
            Thread t = w.thread;
            //  如果当前线程不是中断状态并且尝试加锁成功，则进行打断
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            //  仅仅作用于一个worker之后就退出
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}

/**
 * Common form of interruptIdleWorkers, to avoid having to
 * remember what the boolean argument means.
 */
private void interruptIdleWorkers() {
    interruptIdleWorkers(false);
}

private static final boolean ONLY_ONE = true;

/*
 * Misc utilities, most of which are also exported to
 * ScheduledThreadPoolExecutor
 */

// 内部方法执行拒绝
final void reject(Runnable command) {
    handler.rejectedExecution(command, this);
}

//  子类需重新方法
void onShutdown() {
}

// 判断是运行或者将要停止状态
final boolean isRunningOrShutdown(boolean shutdownOK) {
    int rs = runStateOf(ctl.get());
```

```
        return rs == RUNNING || (rs == SHUTDOWN && shutdownOK);
    }

    //  把workQueue里面的元素取出来
    private List<Runnable> drainQueue() {
        BlockingQueue<Runnable> q = workQueue;
        ArrayList<Runnable> taskList = new ArrayList<Runnable>();
        q.drainTo(taskList);
        if (!q.isEmpty()) {
            for (Runnable r : q.toArray(new Runnable[0])) {
                if (q.remove(r))
                    taskList.add(r);
            }
        }
        return taskList;
    }

    /*
     * Methods for creating, running and cleaning up after workers
     */

    /**
     * <B>方法名称：</B>addWorker<BR>
     * <B>概要说明：</B>把任务添加到队列<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:46:11
     * @param firstTask 执行的任务
     * @param core      线程是否是一个core线程
     * @return
     */
    private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        for (;;) {
            int c = ctl.get();
            //  获取运行状态
            int rs = runStateOf(c);

            // Check if queue empty only if necessary.
            /**
             *  1.  线程池已经shutdown后，还要添加新的任务则直接拒绝；
             *
             *  2.  SHUTDOWN状态虽然不接受新任务，但仍然会执行已经加入任务队列的任务；
             *      所以当进入SHUTDOWN状态，而传进来的任务为空，并且任务队列不为空的时候，是允许添
加新线程的；
             *      如果把这个条件取反，就表示不允许添加 worker；
             */
            if (rs >= SHUTDOWN &&
                ! (rs == SHUTDOWN &&
                   firstTask == null &&
                   ! workQueue.isEmpty()))
                return false;

            for (;;) {
```

```
                //  获得 Worker工作线程数
                int wc = workerCountOf(c);
                //  如果工作线程数大于默认容量大小或者大于动态线程数大小，则直接返回false；表示不能
再添加worker;
                //  动态线程数大小取决于core参数:
                //  core = true 则为corePoolSize
                //  core = false 则为 maximumPoolSize
                if (wc >= CAPACITY ||
                    wc >= (core ? corePoolSize : maximumPoolSize))
                    return false;
                //  通过 cas来增加工作线程数，如果 cas失败，则直接重试
                if (compareAndIncrementWorkerCount(c))
                    break retry;
                //  再次获取ctl的值，通过新的ctl值获取运行状态
                c = ctl.get();
                //  如果新的运行状态和旧的运行状态不相等，说明线程的状态发生了变化，继续重试
                if (runStateOf(c) != rs)
                    continue retry;
                // else CAS failed due to workerCount change; retry inner loop
            }
        }

        //  上面这段代码主要是对worker数量做CAS+1操作，接下来才是构建worker对象

        boolean workerStarted = false;  //  工作线程是否启动的标识
        boolean workerAdded = false;    //  工作线程是否已经添加成功的标识
        Worker w = null;                //  worker对象
        try {
            //  构建一个 Worker，传入task对象
            w = new Worker(firstTask);
            //  获取当前worker的线程引用
            final Thread t = w.thread;
            if (t != null) {
                final ReentrantLock mainLock = this.mainLock;
                //  加锁阻塞，避免并发
                mainLock.lock();
                try {
                    // Recheck while holding lock.
                    // Back out on ThreadFactory failure or if
                    // shut down before lock acquired.

                    //  获取运行状态
                    int rs = runStateOf(ctl.get());
                    /**
                     * 只有当前线程池是正在运行状态，
                     * 或是SHUTDOWN且firstTask为空，才能添加到workers集合中;
                     */
                    if (rs < SHUTDOWN ||(rs == SHUTDOWN && firstTask == null)) {
                        //  任务刚封装到 work里面，还没 start，如果是alive状态则直接抛出异常
                        if (t.isAlive()) // precheck that t is startable
                            throw new IllegalThreadStateException();
                        //  将新创建的worker添加到workers集合中;
                        workers.add(w);
```

```
                        //  如果集合中的工作线程数大于最大线程数，这个最大线程数表示线程池曾经出现过的最大线程数
                        int s = workers.size();
                        if (s > largestPoolSize)
                            //  更新线程池出现过的最大线程数
                            largestPoolSize = s;
                        //  工作线程是否已经添加成功的标识，设置为成功
                        workerAdded = true;
                    }
                } finally {
                    //  释放锁
                    mainLock.unlock();
                }
                //  如果workerAdded=true 标记成功，则启动工作线程
                if (workerAdded) {
                    t.start();
                    //  线程启动成功标记，设置为成功
                    workerStarted = true;
                }
            }
        } finally {
            //  线程启动成功标记如果为失败，则调用addWorkerFailed方法
            //  如果添加失败，就需要做一件事；就是递减实际工作线程数(还原之前的CAS+1操作)
            if (! workerStarted)
                addWorkerFailed(w);
        }
        //  最终返回是否启动成功标记，也就是添加成功标记，表示addWorker方法成功添加了任务并且任务已经执行
        return workerStarted;
    }

    /**
     * <B>方法名称：</B>addWorkerFailed<BR>
     * <B>概要说明：</B>如果添加Worker并且启动线程失败，则会做失败后的处理，CAS-1操作<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午5:33:50
     * @param w
     */
    private void addWorkerFailed(Worker w) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //  如果worker已经构造好了，则从workers集合中移除这个worker
            if (w != null)
                workers.remove(w);
            //  原子递减容器工作线程数
            decrementWorkerCount();
            //  尝试结束线程池
            tryTerminate();
        } finally {
            mainLock.unlock();
        }
    }
```

```
    /**
     * <B>方法名称：</B>processWorkerExit<BR>
     * <B>概要说明：</B>Worker退出的后续处理工作，比如销毁工作线程<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午6:06:13
     * @param w
     * @param completedAbruptly
     */
    private void processWorkerExit(Worker w, boolean completedAbruptly) {
        //  如果出现了异常情况，强制打断worker运行，workerCount(线程工作数量执行CAS--操作)
        //  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，这里就不必再减了
        if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
            decrementWorkerCount();

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //  非强制打断worker运行，也就是工作线程正常运行完成退出时，统计任务完成数量
            completedTaskCount += w.completedTasks;
            //  删除工作线程;
            workers.remove(w);
        } finally {
            mainLock.unlock();
        }
        //  尝试中断操作
        tryTerminate();

        //  再次获取ctl，并确定当前容器状态是否小于STOP状态(RUNNING或SHUTDOWN状态：处于可以执行任务的状态)
        int c = ctl.get();

        if (runStateLessThan(c, STOP)) {
            //  如果worker是正常结束运行
            if (!completedAbruptly) {
                //  allowCoreThreadTimeOut：默认为false，workerCount空闲也要至少要保持核心线程数量，反之则核心线程就算空闲也会存活;
                //  如果allowCoreThreadTimeOut=false，那么workerCount不少于corePoolSize
                //  如果allowCoreThreadTimeOut=true，并且等待队列有任务，至少保留一个worker用于执行任务
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                //  如果队列里还有剩余任务排队，则分配一个线程
                if (min == 0 && ! workQueue.isEmpty())
                    min = 1;
                //  如果当前工作线程数大于等于最小线程数，说明任务肯定会被剩余的工作线程执行完，直接return即可
                if (workerCountOf(c) >= min)
                    return; // replacement not needed
            }
            //  如果worker是异常退出：则直接addWorker进行处理，可能是新建一个线程去继续处理队列中的任务;
            //  如果worker是正常退出：也是一样等待后续处理任务;
```

```
                addWorker(null, false);
            }
        }

        /**
         * <B>方法名称：</B>getTask<BR>
         * <B>概要说明：</B>获取任务方法<BR>
         * @author hezhuo.bai-JiFeng
         * @since 2020年2月13日 下午6:44:29
         * @return Runnable返回任务
         */
        private Runnable getTask() {
            boolean timedOut = false; // Did the last poll() time out?
            //  自旋操作
            for (;;) {
                //  获取当前线程池运行状态
                int c = ctl.get();
                int rs = runStateOf(c);

                /**
                 *  对线程池状态的判断，两种情况会workerCount-1并且返回null：
                 *
                 *  1.  线程池状态为SHUTDOWN，且workQueue为空；
                 *      SHUTDOWN状态的线程池还是要执行workQueue中剩余的任务；
                 *
                 *  2.  线程池状态为STOP，shutdownNow()会导致变成 STOP；
                 *      此时不用考虑 workQueue的情况；
                 */
                if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                    decrementWorkerCount();
                    //  这里返回空，则上层的runWorker方法中自旋获取任务为null，
                    //  就会执行runWorker的finally中的processWorkerExit方法，对Worker进行彻底回
收处理工作
                    return null;
                }
                //  获取当前工作线程数量
                int wc = workerCountOf(c);

                //  Are workers subject to culling?
                //  timed变量用于判断是否需要进行超时控制：
                //  对于allowCoreThreadTimeOut = true，或者超过核心线程数量的这些线程需要进行超时控
制；
                //  allowCoreThreadTimeOut，默认是false，也就是核心线程不允许进行超时；
                //  wc > corePoolSize，表示当前线程池中的线程数量大于核心线程数量；
                boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

                /**
                 *  1.  线程数量超过maximumPoolSize的情况：
                 *      可能是线程池在运行时被调用了setMaximumPoolSize()被改变了大小，
                 *      否则不会超过maximumPoolSize；
                 *
                 *  2.  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取任务发生了超时，其实就是体现了空闲线程的存活时间；
```

```
             *
             *  如果发生1,2条件命中，并且workQueue为null，则首先对WorkerCount做CAS--操作;
             *  然后则退出当前for循环，return null;
             *  如果没有成功对WorkerCount做CAS--操作；则执行continue，继续for循环;
             */
            if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }

            try {
                /**
                 *  timed=true，则通过阻塞队列 poll方法进行超时控制;
                 *  如果在keepaliveTime时间内没有获取到任务，则返回 null;
                 *  否则通过take方法阻塞式获取队列中的任务;
                 */
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                //  返回任务数据
                if (r != null) {
                    return r;
                }
                //  如果r==null则说明已经超时了，设置timedOut=true并在下次自旋的时候进行回收;
                //  注意: 下次自旋操作本方法只是对计数器进行CAS--操作，而返回空后由runWorker方进
行回收worker处理
                timedOut = true;
            } catch (InterruptedException retry) {
                /**
                 *  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;
                 *  注意虽然poll和take方法均支持InterruptedException打断当前worker内部的线程对
象，
                 *  但是这里catch到了这个打断异常，所以不会出现跳出自旋的情况;
                 */
                timedOut = false;
            }
        }
    }

    /**
     * <B>方法名称: </B>runWorker => Main worker run loop<BR>
     * <B>概要说明: </B>执行任务方法<BR>
     *  ThreadPoolExecutor的核心方法addWorker主要作用是增加工作线程，
     *  而Worker则其实就是一个线程里面重写了run方法，由runWorker方法真正执行任务对象(Worker);
     *  runWorker这个方法主要做几件事:
     *
     *  1. 如果task不为空,则开始执行task;
     *
     *  2. 如果task为空，则通过getTask()再去取任务并赋值给task，如果取到的Runnable不为空则执行该
任务;
     *
```

```
     *  3. 执行完毕后，通过while循环继续 getTask()取任务;
     *
     *  4. 如果getTask()取到的任务依然是空，那么整个runWorker()方法执行完毕;
     *
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午5:40:49
     * @param w
     */
    final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        /**
         *  unlock表示当前worker线程允许中断，因为 new Worker默认的 state=-1;
         *  此处是调用Worker类的tryRelease()方法，将state置为0;
         *  而interruptIfStarted()中只有state>=0才允许调用中断;
         */
        w.unlock(); // allow interrupts
        //  标记变量，表示在执行任务过程中是否出现了异常
        boolean completedAbruptly = true;
        try {
            //  while循环在这里实现了线程复用，如果task为空则通过getTask来获取任务;
            while (task != null || (task = getTask()) != null) {
                /**
                 *  上锁的目的不是为了防止并发执行任务，因为Worker本身就是单线程执行的;
                 *  而是为了在shutdown()时不终止正在运行的worker任务;
                 *
                 *  线程池为stop状态时不接受新任务，不执行已经加入任务队列的任务并且还会尝试中断正
在执行的任务
                 *  所以对于stop状态以上的状态是要中断线程的，比如TIDYING、TERMINATED状态;
                 */
                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

                //  1.  如果是STOP或以上运行状态
                //  或者情况:
                //  2.  首先中断线程，然后判断是否是STOP或以上运行状态;
                //      如果不是(!wt.isInterrupted())

                //  1和2的条件成立，最终还要则还需要再次中断
                if ((runStateAtLeast(ctl.get(), STOP) ||
                     (Thread.interrupted() &&
                      runStateAtLeast(ctl.get(), STOP))) &&
                    !wt.isInterrupted())
                    wt.interrupt();
                try {
                    //  任务运行前的前置处理器
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
```

```
                        //  执行任务中的run方法
                        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 {
                        //  任务运行后的后置处理器
                        afterExecute(task, thrown);
                    }
                } finally {
                    //  运行完成后，置空任务；这样下次循环开始时，task依然为 null，需要再通过
getTask()获取任务
                    task = null;
                    //  记录该Worker完成任务数量
                    w.completedTasks++;
                    //  解锁操作
                    w.unlock();
                }
            }
            //  表示执行过了while循环之后的completedAbruptly状态，没有出现异常情况，正常执行完毕
任务
            completedAbruptly = false;
        } finally {
            //  1.将入参 worker从数组 workers里删除掉;
            //  2.根据布尔值allowCoreThreadTimeOut来决定是否补充新的Worker进数组worker
            processWorkerExit(w, completedAbruptly);
        }
    }

    // Public constructors and methods

    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), defaultHandler);
    }

    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             threadFactory, defaultHandler);
    }
```

```
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              RejectedExecutionHandler handler) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), handler);
    }

    /**
     * <B>构造方法</B>ThreadPoolExecutor<BR>
     * @param corePoolSize      核心线程数
     * @param maximumPoolSize   最大线程数
     * @param keepAliveTime     空闲超时时间
     * @param unit              时间单位
     * @param workQueue         工作队列
     * @param threadFactory     线程工厂
     * @param handler           拒绝策略
     */
    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();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }

    /**
     * <B>方法名称：</B>execute<BR>
     * <B>概要说明：</B>执行execute方法<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:52:55
     * @see com.bfxy.thread.design.concurrent.Executor#execute(java.lang.Runnable)
     */
    public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
```

```
        int c = ctl.get();
        /**
         *  1.  如果当前线程池（或者说容器）中的线程数少于corePoolSize，
         *      则调用addWorker(command, true)去创建一个核心线程，并执行任务；
         */
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        /**
         *  2.  如果不满足条件1，则说明核心线程已满，则调用workQueue.offer(command)方法，
         *      把当前要执行的任务加入到队列中，加入成功返回true；
         *      如果一个任务可以成功排队(加入成功)，我们仍然需要再次检查我们是否应该添加一个线程；
         *      因为上次检查过后可能现在线程池已经关闭了，所以要再次检查；
         */
        //  如果是运行状态 && 加入到队列成功
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            //  如果不是运行状态，进行删除队列元素并且执行拒绝策略
            if (! isRunning(recheck) && remove(command))
                reject(command);
            /**
             *  workerCountOf(recheck) == 0 表示当前工作线程数已经没有了，
             *  但是由上一个条件得知任务已经添加队列成功，workQueue.offer(command) = true;
             *
             *  接下来则调用addWorker方法，尝试执行任务：
             *  addWorker第一个参数是null，表示没有新任务加入；
             *  addWorker第二个参数是false，说明当前线程池（或者说容器）中的工作线程数大于
corePoolSize,
             *   这时如果是运行状态或SHUTDOWN状态，会新创建线程来执行加入的任务；
             *  否则会执行内部addWorkerFailed方法添加失败记录；
             */
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        /**
         *  3.  如果加入队列失败，说明这时候 核心池已满，队列已经满，试着创建一个新线程；
         *      如果创建新线程失败了，说明线程池被关闭或者线程池完全满了，拒绝任务
         */
        else if (!addWorker(command, false))
            reject(command);
    }

    /**
     * <B>方法名称：</B>shutdown<BR>
     * <B>概要说明：</B>停止<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:24:32
     * @see com.bfxy.thread.design.concurrent.ExecutorService#shutdown()
     */
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
```

```
        mainLock.lock();
        try {
            checkShutdownAccess();
            advanceRunState(SHUTDOWN);
            interruptIdleWorkers();
            onShutdown(); // hook for ScheduledThreadPoolExecutor
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
    }

    /**
     * <B>方法名称：</B>shutdownNow<BR>
     * <B>概要说明：</B>立即停止，并且返回在队列里等待的元素集合<BR>
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:24:42
     * @see com.bfxy.thread.design.concurrent.ExecutorService#shutdownNow()
     */
    public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            advanceRunState(STOP);
            interruptWorkers();
            //  返回取出来的元素，也就是正在等待的队列
            tasks = drainQueue();
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
        return tasks;
    }

    public boolean isShutdown() {
        return ! isRunning(ctl.get());
    }

    public boolean isTerminating() {
        int c = ctl.get();
        return ! isRunning(c) && runStateLessThan(c, TERMINATED);
    }

    public boolean isTerminated() {
        return runStateAtLeast(ctl.get(), TERMINATED);
    }

    public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        final ReentrantLock mainLock = this.mainLock;
```

```
        mainLock.lock();
        try {
            for (;;) {
                if (runStateAtLeast(ctl.get(), TERMINATED))
                    return true;
                if (nanos <= 0)
                    return false;
                nanos = termination.awaitNanos(nanos);
            }
        } finally {
            mainLock.unlock();
        }
    }

    //  最终关闭线程
    protected void finalize() {
        shutdown();
    }

    //  设置线程工厂
    public void setThreadFactory(ThreadFactory threadFactory) {
        if (threadFactory == null)
            throw new NullPointerException();
        this.threadFactory = threadFactory;
    }

    //  获取线程工厂
    public ThreadFactory getThreadFactory() {
        return threadFactory;
    }

    //  设置拒绝策略对象
    public void setRejectedExecutionHandler(RejectedExecutionHandler handler) {
        if (handler == null)
            throw new NullPointerException();
        this.handler = handler;
    }

    //  返回拒绝策略对象
    public RejectedExecutionHandler getRejectedExecutionHandler() {
        return handler;
    }

    //  设置corePoolSize
    public void setCorePoolSize(int corePoolSize) {
        if (corePoolSize < 0)
            throw new IllegalArgumentException();
        int delta = corePoolSize - this.corePoolSize;
        //  设置新的corePoolSize
        this.corePoolSize = corePoolSize;
        //  如果新的corePoolSize小于旧的corePoolSize, 则进行打断空闲的workers
        if (workerCountOf(ctl.get()) > corePoolSize)
            interruptIdleWorkers();
```

```
        //  新的corePoolSize - 旧的corePoolSize > 0，则添加新的worker对象
        else if (delta > 0) {
            // We don't really know how many new threads are "needed".
            // As a heuristic, prestart enough new workers (up to new
            // core size) to handle the current number of tasks in
            // queue, but stop if queue becomes empty while doing so.
            int k = Math.min(delta, workQueue.size());
            while (k-- > 0 && addWorker(null, true)) {
                if (workQueue.isEmpty())
                    break;
            }
        }
    }

    //  获取核心coreSize
    public int getCorePoolSize() {
        return corePoolSize;
    }

    //  如果所有核心线程已经启动则返回true，否则返回未启动所有线程并启动一个worker
    public boolean prestartCoreThread() {
        return workerCountOf(ctl.get()) < corePoolSize &&
            addWorker(null, true);
    }

    //  与prestartAllCoreThreads方法相同，    但是只会启动一个worker对象，即使coreSize=0;
    void ensurePrestart() {
        int wc = workerCountOf(ctl.get());
        if (wc < corePoolSize)
            addWorker(null, true);
        else if (wc == 0)
            addWorker(null, false);
    }

    //  启动所有核心线程，导致它们无所事事地等待工作；预启动策略
    public int prestartAllCoreThreads() {
        int n = 0;
        while (addWorker(null, true))
            ++n;
        return n;
    }

    //  获取是否可以回收空闲的核心线程，默认为false，不可回收核心线程
    public boolean allowsCoreThreadTimeOut() {
        return allowCoreThreadTimeOut;
    }

    //  设置是是否可以回收空闲的核心线程，true则可以回收；
    public void allowCoreThreadTimeOut(boolean value) {
        if (value && keepAliveTime <= 0)
            throw new IllegalArgumentException("Core threads must have nonzero keep
alive times");
        if (value != allowCoreThreadTimeOut) {
```

```
            allowCoreThreadTimeOut = value;
            if (value)
                interruptIdleWorkers();
        }
    }

    //  设置最大线程池数量
    public void setMaximumPoolSize(int maximumPoolSize) {
        if (maximumPoolSize <= 0 || maximumPoolSize < corePoolSize)
            throw new IllegalArgumentException();
        this.maximumPoolSize = maximumPoolSize;
        if (workerCountOf(ctl.get()) > maximumPoolSize)
            interruptIdleWorkers();
    }

    //  获取最大线程池数量
    public int getMaximumPoolSize() {
        return maximumPoolSize;
    }

    //  设置keepAliveTime
    public void setKeepAliveTime(long time, TimeUnit unit) {
        if (time < 0)
            throw new IllegalArgumentException();
        if (time == 0 && allowsCoreThreadTimeOut())
            throw new IllegalArgumentException("Core threads must have nonzero keep
alive times");
        long keepAliveTime = unit.toNanos(time);
        long delta = keepAliveTime - this.keepAliveTime;
        this.keepAliveTime = keepAliveTime;
        if (delta < 0)
            interruptIdleWorkers();
    }

    //  获取空闲时间keepAliveTime
    public long getKeepAliveTime(TimeUnit unit) {
        return unit.convert(keepAliveTime, TimeUnit.NANOSECONDS);
    }

    /* User-level queue utilities */

    //  获取队列workQueue
    public BlockingQueue<Runnable> getQueue() {
        return workQueue;
    }

    //  remove 移除一个任务
    public boolean remove(Runnable task) {
        boolean removed = workQueue.remove(task);
        tryTerminate(); // In case SHUTDOWN and now empty
        return removed;
    }
```

```
    //  purge 清空队列元素，通常需先做取消操之后才可以删除
    public void purge() {
        final BlockingQueue<Runnable> q = workQueue;
        try {
            Iterator<Runnable> it = q.iterator();
            while (it.hasNext()) {
                Runnable r = it.next();
                if (r instanceof Future<?> && ((Future<?>)r).isCancelled())
                    it.remove();
            }
        } catch (ConcurrentModificationException fallThrough) {
            // Take slow path if we encounter interference during traversal.
            // Make copy for traversal and call remove for cancelled entries.
            // The slow path is more likely to be O(N*N).
            for (Object r : q.toArray())
                if (r instanceof Future<?> && ((Future<?>)r).isCancelled())
                    q.remove(r);
        }

        tryTerminate(); // In case SHUTDOWN and now empty
    }

    /* Statistics */

    //  加锁获取线程池大小
    public int getPoolSize() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // Remove rare and surprising possibility of
            // isTerminated() && getPoolSize() > 0
            return runStateAtLeast(ctl.get(), TIDYING) ? 0
                : workers.size();
        } finally {
            mainLock.unlock();
        }
    }

    //  加锁获取正在执行的任务数量
    public int getActiveCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            int n = 0;
            for (Worker w : workers)
                if (w.isLocked())
                    ++n;
            return n;
        } finally {
            mainLock.unlock();
        }
    }
```

```
    //  加锁获取线程池中出现过的最大线程数值: largestPoolSize
    public int getLargestPoolSize() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            return largestPoolSize;
        } finally {
            mainLock.unlock();
        }
    }

    //  加锁方式获取 已经执行完成 + 正在执行 + 已经入队列的三者任务总数
    public long getTaskCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            long n = completedTaskCount;
            for (Worker w : workers) {
                n += w.completedTasks;
                if (w.isLocked())
                    ++n;
            }
            return n + workQueue.size();
        } finally {
            mainLock.unlock();
        }
    }

    //  加锁方式获取线程池中所有worker对象已经完成的任务数量之和
    public long getCompletedTaskCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            long n = completedTaskCount;
            for (Worker w : workers)
                n += w.completedTasks;
            return n;
        } finally {
            mainLock.unlock();
        }
    }

    public String toString() {
        long ncompleted;
        int nworkers, nactive;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            ncompleted = completedTaskCount;
            nactive = 0;
            nworkers = workers.size();
            for (Worker w : workers) {
                ncompleted += w.completedTasks;
```

```
                if (w.isLocked())
                    ++nactive;
            }
        } finally {
            mainLock.unlock();
        }
        int c = ctl.get();
        String rs = (runStateLessThan(c, SHUTDOWN) ? "Running" :
                     (runStateAtLeast(c, TERMINATED) ? "Terminated" :
                      "Shutting down"));
        return super.toString() +
            "[" + rs +
            ", pool size = " + nworkers +
            ", active threads = " + nactive +
            ", queued tasks = " + workQueue.size() +
            ", completed tasks = " + ncompleted +
            "]";
    }

    /* Extension hooks */

    /** 任务执行之前的前置处理器，用于子类实现        */
    protected void beforeExecute(Thread t, Runnable r) { }

    /** 任务执行之后的后置处理器，用于子类实现        */
    protected void afterExecute(Runnable r, Throwable t) { }

    /** 后置方法待子类实现    */
    protected void terminated() { }

    /* Predefined RejectedExecutionHandlers */

    /**
     *  $CallerRunsPolicy 拒绝策略，暂时不执行，等待后续执行
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:28:51
     */
    public static class CallerRunsPolicy implements RejectedExecutionHandler {

        public CallerRunsPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                r.run();
            }
        }
    }

    /**
     *  $AbortPolicy 拒绝策略，直接抛出系统异常RejectedExecutionException
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:28:09
     */
```

```
    public static class AbortPolicy implements RejectedExecutionHandler {

        public AbortPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            throw new RejectedExecutionException("Task " + r.toString() +
                                                 " rejected from " +
                                                 e.toString());
        }
    }

    /**
     *  $DiscardPolicy 拒绝策略，不做任何处理直接丢弃
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:27:32
     */
    public static class DiscardPolicy implements RejectedExecutionHandler {

        public DiscardPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        }
    }

    /**
     *  $DiscardOldestPolicy 拒绝策略，丢弃最老的任务
     * @author hezhuo.bai-JiFeng
     * @since 2020年2月13日 下午3:26:40
     */
    public static class DiscardOldestPolicy implements RejectedExecutionHandler {

        public DiscardOldestPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                e.getQueue().poll();
                e.execute(r);
            }
        }
    }
}
```