线程池ThreadPoolExecutor源码剖析

一、Java构建线程的方式

- 继承Thread类

Thread类

Thread.java

```
139     */
140    public
141    class Thread implements Runnable {
```

```
162        /* What will be run. */
163        private Runnable target;
```

```
463    public Thread(Runnable target) {
464        init( g: null, target, name: "Thread-" + nextThreadNum(), stackSize: 0);
465    }
```

```
745    @Override
746    public void run() {
747        if (target != null) {
748            target.run();
749        }
750    }
```

可以看出Thread类本身就实现了Runnable接口，并且在创建Thread类时，通过有参构造传入自己写好的Thread类，会将Thread类中的target属性赋值。并且在调用线程的start方法后自然会执行你传入的Thread类重新好的run方法执行。

- 实现Runnable接口

  Java毕竟是单继承，覆写Runnable接口实现多线程可以避免单继承局限

Runnable接口

```
55     @FunctionalInterface
56     public interface Runnable {
57         /**
58          * When an object implementing interface <code>Runnable</code> is used
59          * to create a thread, starting the thread causes the object's
60          * <code>run</code> method to be called in that separately executing
61          * thread.
62          * <p>
63          * The general contract of the method <code>run</code> is that it may
64          * take any action whatsoever.
65          *
66          * @see     java.lang.Thread#run()
67          */
68         public abstract void run();
69     }
```

- 实现Callable接口

  这种方式跟上述的两种方式不同，上述两种方式在run方法结束线程销毁后无法获取返回结果，最多就是通过共享变量的方式获取，但是Callable提供的重新call方法是可以有返回结果的并且可以抛出异常信息。

  并且获取Callable结果的方式需要配置FutureTask 的get方法才可以获取到

Callable接口

```java
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

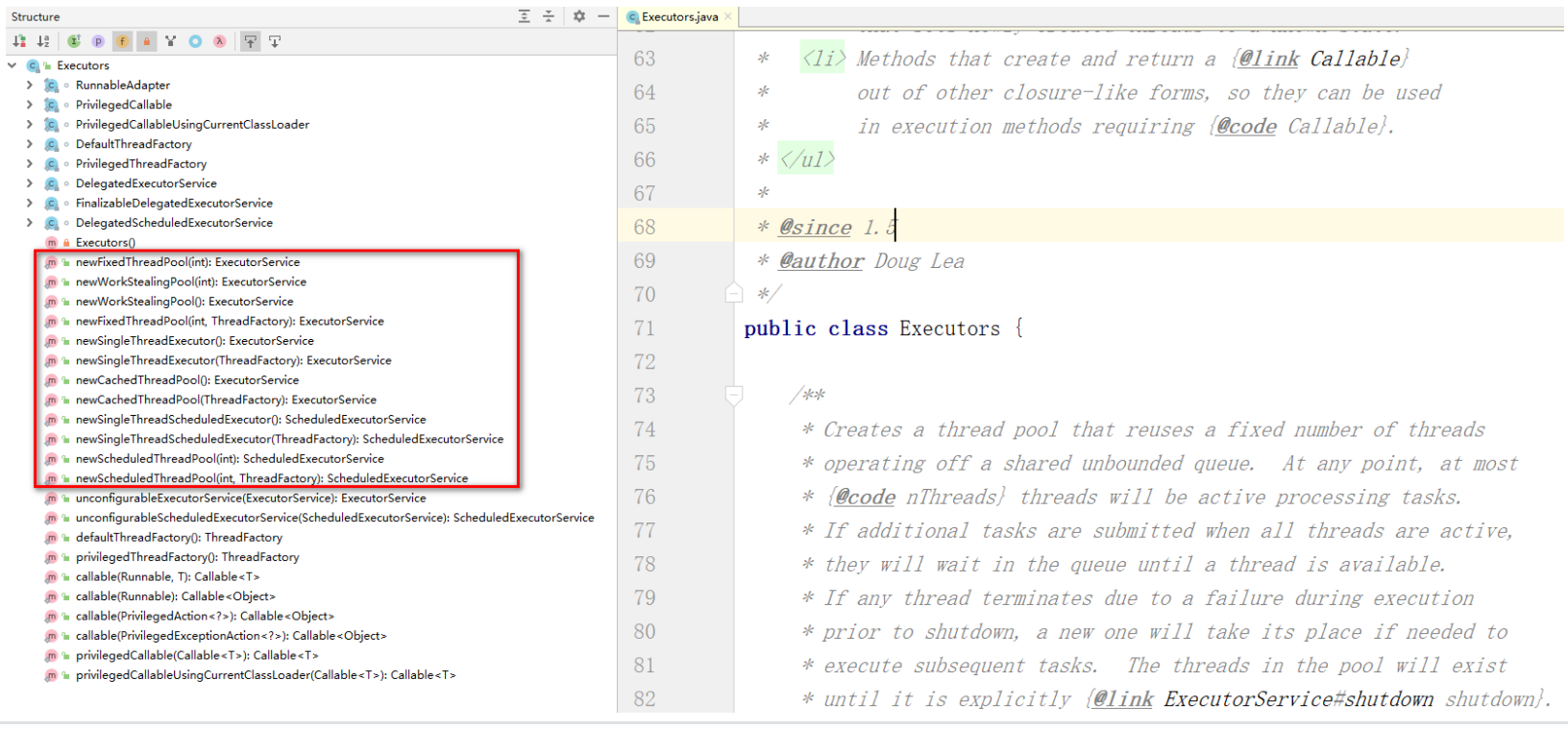
```java
public class CallableTest implements Callable<Integer> {
      private int num = 0;

      @Override
      public Integer call() throws Exception {
            for (int i = 0; i < 10; i++) {
                  num++;
            }
            return num;
      }

       public static void main(String[] args) throws Exception {
            Callable<Integer> callable = new CallableTest();
            FutureTask<Integer> futureTask = new FutureTask<>(callable);
            Thread t = new Thread(futureTask);
            t.start();
            System.out.println(futureTask.get());
      }
}
```

- 线程池方式

  线程池方式其实和上面没有区别，只不过上面需要频繁的创建和销毁线程，会造成一些不必要的额外资源消耗，所以在实际开发中肯定会采用线程池的方式

  Java中的Executors自带了一些创建线程池的方式

Executors自带线程池



在构建好线程池ThreadPoolExecutor后就可以直接调用execute、submit方法执行线程任务。

- execute适用于Runnable没有返回结果
- submit更适合Callable具有返回结果

二、线程池的7个参数

虽然提供了这么多种线程池的构建，但是在规范中我们更推荐手动构建线程池并设置线程池自带的七个参数，第一个目的是为了更了解线程池的运行原理，其次也是为了更好的根据业务情况去制定线程池的参数而提升任务的执行性能，避免不必要的资源消耗。

所以我们需要对线程池的参数有所掌握

```java
/**
      * Creates a new {@code ThreadPoolExecutor} with the given initial
      * parameters.
```

```
        *
        * @param corePoolSize the number of threads to keep in the pool, even
        *          if they are idle, unless {@code allowCoreThreadTimeOut} is set
        * @param maximumPoolSize the maximum number of threads to allow in the
        *          pool
        * @param keepAliveTime when the number of threads is greater than
        *          the core, this is the maximum time that excess idle threads
        *          will wait for new tasks before terminating.
        * @param unit the time unit for the {@code keepAliveTime} argument
        * @param workQueue the queue to use for holding tasks before they are
        *          executed.  This queue will hold only the {@code Runnable}
        *          tasks submitted by the {@code execute} method.
        * @param threadFactory the factory to use when the executor
        *          creates a new thread
        * @param handler the handler to use when execution is blocked
        *          because the thread bounds and queue capacities are reached
        * @throws IllegalArgumentException if one of the following holds:<br>
        *             {@code corePoolSize < 0}<br>
        *             {@code keepAliveTime < 0}<br>
        *             {@code maximumPoolSize <= 0}<br>
        *             {@code maximumPoolSize < corePoolSize}
        * @throws NullPointerException if {@code workQueue}
        *             or {@code threadFactory} or {@code handler} is null
        */
public ThreadPoolExecutor(int corePoolSize,
                                  int maximumPoolSize,
                                  long keepAliveTime,
                                  TimeUnit unit,
                                  BlockingQueue<Runnable> workQueue,
                                  ThreadFactory threadFactory,
                                  RejectedExecutionHandler handler) {   /* 阿巴阿巴 */}
```

- corePoolSize 线程池核心线程大小

  线程池中会维护一个最小的线程数量，即使这些线程处理空闲状态，他们也不会被销毁，除非设置了allowCoreThreadTimeOut。这里的最小线程数量即是corePoolSize。

- maximumPoolSize 线程池最大线程数量

  一个任务被提交到线程池以后，首先会找有没有空闲存活线程，如果有则直接将任务交给这个空闲线程来执行，如果没有则会缓存到工作队列（后面会介绍）中，如果工作队列满了，才会创建一个新线程，然后从工作队列的头部取出一个任务交由新线程来处理，而将刚提交的任务放入工作队列尾部。线程池不会无限制的去创建新线程，它会有一个最大线程数量的限制，这个数量即由maximunPoolSize指定。

- keepAliveTime 空闲线程存活时间。

  一个线程如果处于空闲状态，并且当前的线程数量大于corePoolSize，那么在指定时间后，这个空闲线程会被销毁，这里的指定时间由keepAliveTime来设定。

- unit 空闲线程存活时间单位

  keepAliveTime的计量单位

- workQueue 工作队列

  新任务被提交后，会先进入到此工作队列中，任务调度时再从队列中取出任务。jdk中提供了四种工作队列：

  - ArrayBlockingQueue

    基于数组的有界阻塞队列，按FIFO排序。新任务进来后，会放到该队列的队尾，有界的数组可以防止资源耗尽问题。当线程池中线程数量达到corePoolSize后，再有新任务进来，则会将任务放入该队列的队尾，等待被调度。如果队列已经是满的，则创建一个新线程，如果线程数量已经达到maxPoolSize，则会执行拒绝策略。

  - LinkedBlockingQueue

    基于链表的无界阻塞队列（其实最大容量为Interger.MAX），按照FIFO排序。由于该队列的近似无界性，当线程池中线程数量达到corePoolSize后，再有新任务进来，会一直存入该队列，而不会去创建新线程直到maxPoolSize，因此使用该工作队列时，参数maxPoolSize其实是不起作用的。

  - SynchronousQueue

    一个不缓存任务的阻塞队列，生产者放入一个任务必须等到消费者取出这个任务。也就是说新任务进来时，不会缓存，而是直接被调度执行该任务，如果没有可用线程，则创建新线程，如果线程数量达到maxPoolSize，则执行拒绝策略。

  - PriorityBlockingQueue

    具有优先级的无界阻塞队列，优先级通过参数Comparator实现。

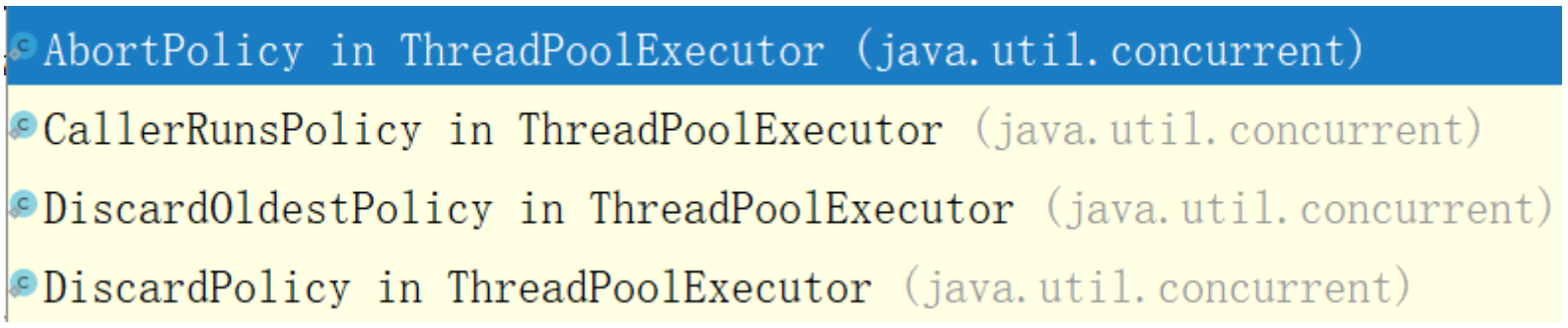
- threadFactory 线程工厂

  创建一个新线程时使用的工厂，可以用来设定线程名、是否为daemon线程等等。

- handler 拒绝策略

  当工作队列中的任务已到达最大限制，并且线程池中的线程数量也达到最大限制，这时如果有新任务提交进来，该如何处理呢。这里的拒绝策略，就是解决这个问题的，jdk中提供了4中拒绝策略：

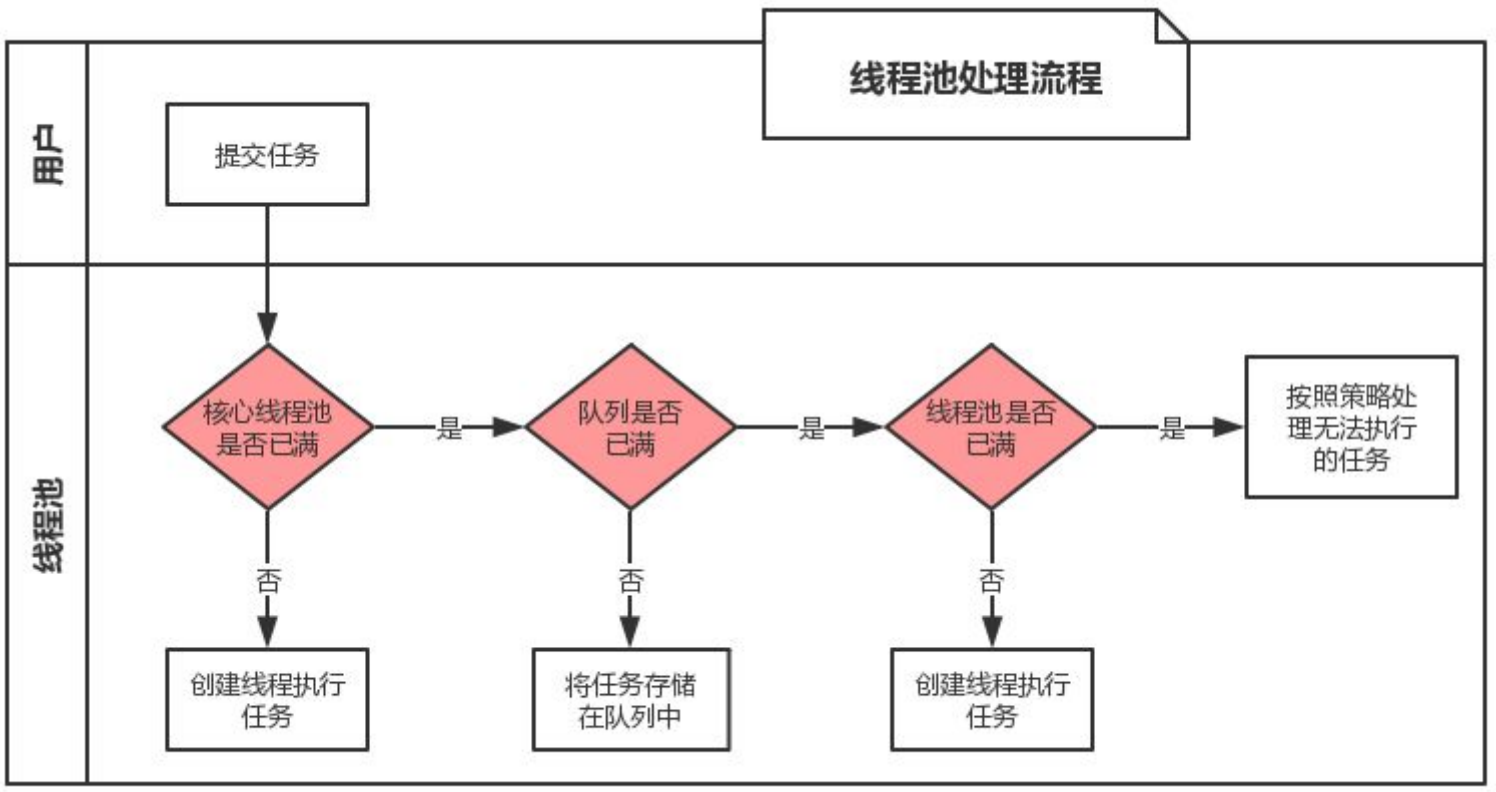
四种拒绝策略，也可以自己实现



三、线程池的执行流程

在了解了线程池的七个参数后，我们还需要掌握在线程池中提交任务后线程池的具体走向

线程池执行流程



四、线程池源码初体验

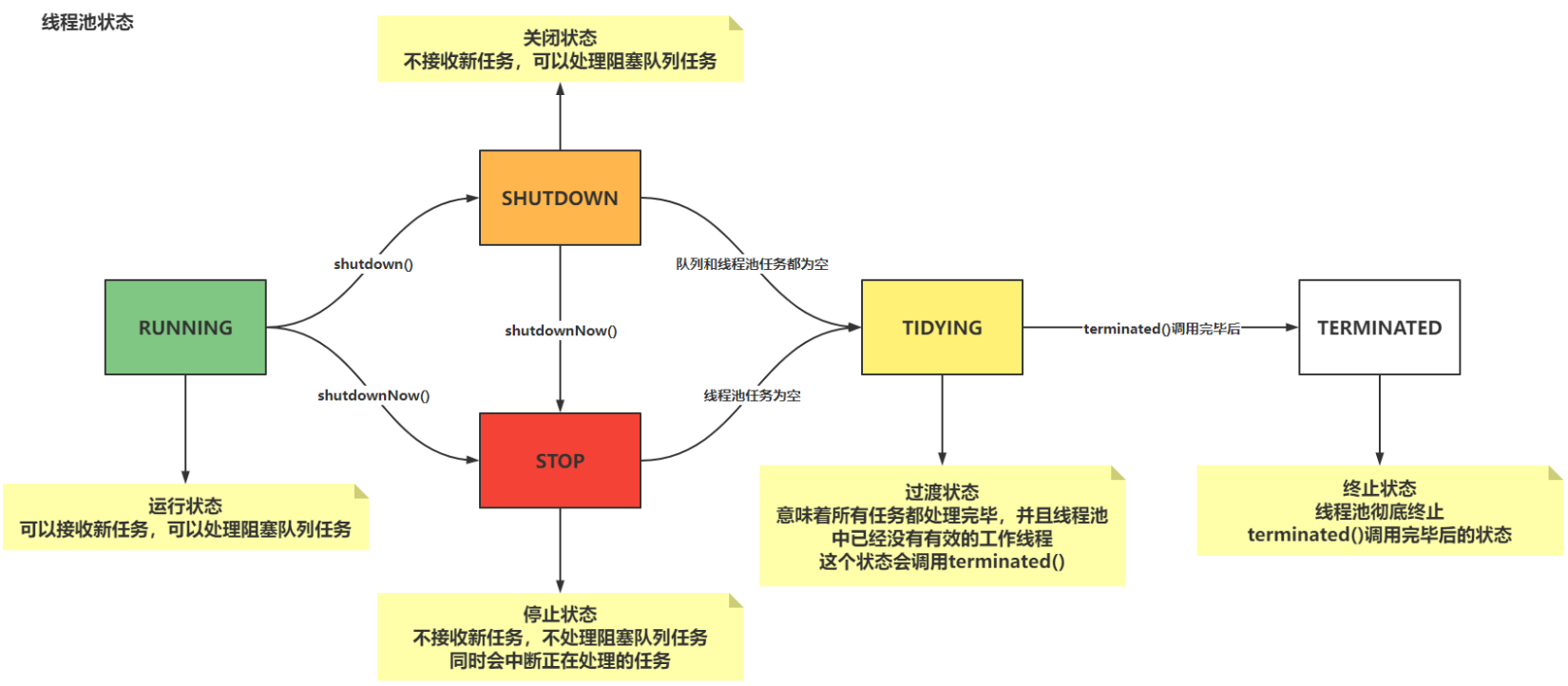
4.1 线程池中的主要成员变量

```
public class ThreadPoolExecutor extends AbstractExecutorService {

    // ctl初始化了线程的状态和线程数量，初始状态为RUNNING并且线程数量为0
    // 这里一个Integer既包含了状态也包含了数量，其中int类型一共32位，高3位标识状态，低29位标识数量
  private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    // 这里指定了Integer.SIZE - 3，也就是32 - 3 = 29，表示线程数量最大取值长度
    private static final int COUNT_BITS = Integer.SIZE - 3;
    // 这里标识线程池容量，也就是将1向左位移上面的29长度，并且-1代表最大取值，二进制就是 000111..111
    private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

    // 这里是高三位的状态表示
    private static final int RUNNING    = -1 << COUNT_BITS;   // 111
    private static final int SHUTDOWN   =  0 << COUNT_BITS;   // 000
    private static final int STOP       =  1 << COUNT_BITS;   // 001
    private static final int TIDYING    =  2 << COUNT_BITS;   // 010
    private static final int TERMINATED =  3 << COUNT_BITS;   // 011

    // 获取线程数量、状态等方式
    // 通过传入的c，获取最高三位的值，拿到线程状态吗，最终就是拿 1110 000……和c做&运算得到高3位结果
    private static int runStateOf(int c)     { return c & ~CAPACITY; }
    // 获取当前线程数量，最终得到现在线程数量，就是拿c 和 0001 111……做&运算，得到低29位结果
    private static int workerCountOf(int c)  { return c & CAPACITY; }
    // 这里是用来更新线程状态和数量的方式。


    private static int ctlOf(int rs, int wc) { return rs | wc; }
        private static boolean runStateLessThan(int c, int s) {
        return c < s;
    }
```

```
        private static boolean runStateAtLeast(int c, int s) {
            return c >= s;
        }

        // 判断当前线程池状态是否是Running。
        private static boolean isRunning(int c) {
            return c < SHUTDOWN;
        }

}
```

4.2 线程池状态

线程池状态



五、线程池的execute方法执行

首先执行execute方法

```
public class ThreadPoolExecutorTest {

    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                2,
                4,
                10,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<>(2),
                new ThreadPoolExecutor.AbortPolicy());

        executor.execute(() -> {
            System.out.println("Hello!");
        });
    }
}
```

进到源码，可以看到execute的执行内容

```
public void execute(Runnable command) {
    // 首先是健壮性判断
    if (command == null)
        throw new NullPointerException();
    // 这里是获取核心线程数
    int c = ctl.get();
    // 判断工作线程是否少于核心线程数
    if (workerCountOf(c) < corePoolSize) {
        // 如果少于，需要创建新的任务线程，如果addWorker返回false，证明核心线程初始化过了，返回true就直接结束
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 如果线程池属于RUNNING状态，那就添加到任务队列，如果添加成功，进入if
    if (isRunning(c) && workQueue.offer(command)) {
        // 重新获取ctl属性
        int recheck = ctl.get();
        // 再次判断是否是RUNNING状态，如果不是了，就执行remove删掉添加到阻塞队列的任务
        if (!isRunning(recheck) && remove(command))
            // 拒绝任务
            reject(command);
```

```
            // 如果线程数为0
            else if (workerCountOf(recheck) == 0)
                  // 添加一个线程，避免出现队列任务没有线程执行的情况
                  addWorker(null, false);
      }
       // 如果不能入队列，就尝试创建最大线程数
       else if (!addWorker(command, false))
            // 如果创建最大线程数失败，直接拒绝任务
            reject(command);
}
```

这里可以继续看正常流程，如果任务需要执行，就会执行addWorker方法

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

            // 查看是否可以不去构建新的工作线程.
            if (rs >= SHUTDOWN      // 如果线程状态不是RUNNING
                        &&
                  !(rs == SHUTDOWN && firstTask == null && !workQueue.isEmpty()))
                  // 线程不是SHUTDOWN状态（那也就是STOP，TIDYING，TERMINATED三者之一）
                  // 任务不为空 -> 这里对应上述的addWorker(null,false)
                  // 工作队列为null
                  // 这时直接return false，不去构建新的工作线程
                  return false;

            for (;;) {
                  // 获取工作线程数
                  int wc = workerCountOf(c);
                  if (wc >= CAPACITY ||    // 如果工作线程大于最大线程容量
                        wc >= (core ? corePoolSize : maximumPoolSize))   // 或者当前线程数达到核心/最大线程数的要求
                        return false;   // 直接结束，添加Worker失败。
                  if (compareAndIncrementWorkerCount(c))   // CAS的方式
                        break retry;    // 如果成功，跳出最外层循环
                  c = ctl.get();   // 再次读取ctl的值
                  if (runStateOf(c) != rs)   // 如果运行状态不等于最开始查询到的rs，那就从头循环一波。
                        continue retry;
                  // else CAS failed due to workerCount change; retry inner loop
            }
      }

       // 开始添加工作线程
       boolean workerStarted = false;
       boolean workerAdded = false;
       Worker w = null;
       try {
            w = new Worker(firstTask);   // 构建Worker对象传入任务
            final Thread t = w.thread;   // 将worker线程取出
            if (t != null) {
                  // 拿到全局锁，避免我在添加任务时，其他线程干掉线程池，因为干掉线程池需要获取到这个锁
                  final ReentrantLock mainLock = this.mainLock;
                  // 加锁
                  mainLock.lock();
                  try {
                        // 获取线程池状态
                        int rs = runStateOf(ctl.get());

                        // 两种情况允许添加工作线程
                        if (rs < SHUTDOWN ||    // 判断线程池是否是RUNNING状态
                              (rs == SHUTDOWN && firstTask == null)) {   // 如果线程池状态为SHUTDOWN并且任务为空
                              if (t.isAlive()) // 如果线程正在运行，直接抛出异常
                                    throw new IllegalThreadStateException();
                              // 添加任务线程到workers中
                              workers.add(w);
                              // 获取任务线程个数
                              int s = workers.size();
                              // 如果任务线程大于记录的当前出现过的最大线程数，替换一下。
                              if (s > largestPoolSize)
                                    largestPoolSize = s;
                              workerAdded = true;    // 任务添加的标识设置为true
                        }
                  } finally {
                        mainLock.unlock(); //任务添加成功，退出锁资源
                  }
                  if (workerAdded) {   //  如果添加成功
                        t.start();   // 启动线程
                        workerStarted = true;   // 标识启动标识为true
                  }
            }
      } finally {
```

```
            if (!workerStarted)   // 如果线程启动失败
                  addWorkerFailed(w);   // 移除掉刚刚添加的任务
      }
       return workerStarted;
}
```

六、Worker的封装

Work是线程池中具体工作线程，他还继承了AQS，详细分析下Worker工作线程

```
private final class Worker
      extends AbstractQueuedSynchronizer
            implements Runnable{    // 实现了Runnable，意味着是一个线程

            // 线程对象
            final Thread thread;
           // 具体任务
            Runnable firstTask;

            // 有参构造
            Worker(Runnable firstTask) {
                  setState(-1);   // 添加标识，worker运行前，禁止中断（AQS）
                  this.firstTask = firstTask;
                  this.thread = getThreadFactory().newThread(this);    // 创建了一个新的线程，并且线程指向了当前对象，引入就是Worker，所以在调用start方法时，调用的是Worker中的run方法
            }

        // run方法，线程的工作内容
            public void run() {
                  runWorker(this);   // 核心内容
            }
            // …… 省略部分代码
    }
```

了解了Worker之后，需要再次查看runWorker中的操作，线程启动后，调用的就是runWorker

```
final void runWorker(Worker w) {
      // 获取当前线程
      Thread wt = Thread.currentThread();
      // 拿到任务
      Runnable task = w.firstTask;
      w.firstTask = null;
      w.unlock(); // allow interrupts
      boolean completedAbruptly = true;
      try {
            // 任务不为null，就一致循环，否则调用getTask尝试从阻塞队列获取任务
            while (task != null || (task = getTask()) != null) {
                  // 加锁的目的是表示当前任务正在执行，你shutdown任务也不会中断
                  w.lock();
                  if (
                        (runStateAtLeast(ctl.get(), STOP) ||   // 判断线程池是否处于STOP状态
                          (Thread.interrupted() && runStateAtLeast(ctl.get(), STOP)))
                            &&
                          !wt.isInterrupted()   // 判断线程池是否中断
                      )
                        // 只要线程池STOP了，工作线程没有被中断，就中断线程
                        wt.interrupt();
                  try {
                        // 任务执行的前置处理
                        beforeExecute(wt, task);
                        Throwable thrown = null;
                        try {
                              // 任务走起
                              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 {
                              // 任务执行的后置处理
                              afterExecute(task, thrown);
                        }
                  } finally {
                        task = null;
                        w.completedTasks++;
                        w.unlock();
                  }
            }
            completedAbruptly = false;
      } finally {
            processWorkerExit(w, completedAbruptly);
```

```
    }
}
```

这里知道了最终调用task.run()方法让任务启动，前面还有一个getTask方法从阻塞队列获取任务，有就是addWorker中添加到阻塞队列中的任务。这里也是线程池的核心之一

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

      for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);   获取线程池运行状态

            // 如果状态大于0，代表线程池凉凉，
            if (rs >= SHUTDOWN &&
                 (rs >= STOP || workQueue.isEmpty())   // 如果是STOP状态，或者阻塞队列为空
                ) {
                  decrementWorkerCount();
                  // 返回null
                  return null;
            }
          // 获取工作线程数
            int wc = workerCountOf(c);

            // 查看是否允许
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

            if ((wc > maximumPoolSize || (timed && timedOut))   //当前工作线程数大于最大线程数 ，后面判断表示是否是允许核心线程超时并且真的超时
                  &&
                 (wc > 1 || workQueue.isEmpty())) {   //工作线程 > 1或者 阻塞队列为空
                  if (compareAndDecrementWorkerCount(c))   // 干掉当前工作线程并返回null，CAS的方式，如果失败，重新从头走一遍
                        return null;
                  continue;
            }

            try {
                  Runnable r = timed ?
                        // 这里是可能出现超时情况并且允许回收线程，那就阻塞这么久拿阻塞队列的任务
                        workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                     // 这里是线程阻塞在这，等待任务，不参与回收的情况，直到触发signal方法被唤醒，走catch继续下次循环
                     workQueue.take();
                  if (r != null)
                        return r;
                  timedOut = true;
            } catch (InterruptedException retry) {
                  timedOut = false;
            }
      }
}
```

如果上述getTask最终返回了false，那就代表while循环结束，要执行processWorkerExit方法了

七、线程执行的后续处理

processWorkerExit是对线程的后续处理了。

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
 // 判断是否是认为停止，如果是要减掉一个工作线程数
 if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
       decrementWorkerCount();

 // 加锁移除工作线程
 final ReentrantLock mainLock = this.mainLock;
 mainLock.lock();
 try {
       // 记录当前线程处理的任务数
       completedTaskCount += w.completedTasks;
       // 将worker从Set中移除
       workers.remove(w);
} finally {
       mainLock.unlock();
}

 // 尝试干掉线程池
 tryTerminate();

 int c = ctl.get();
 // 判断线程状态是否小于STOP。
 if (runStateLessThan(c, STOP)) {
       // 如果不是认为停止，需要判断线程是否需要追加一个线程处理任务
       if (!completedAbruptly) {
             int min = allowCoreThreadTimeOut ? 0 : corePoolSize;   // 查看核心线程是否允许超时
             if (min == 0 && ! workQueue.isEmpty())   // 如果允许超时，并且工作队列不是空，就将min设置为1
```

```
                    min = 1;
                if (workerCountOf(c) >= min)   // 如果工作线程数量大于核心线程数，就直接结束
                    return; // replacement not needed
            }
             addWorker(null, false);
        }
    }
```