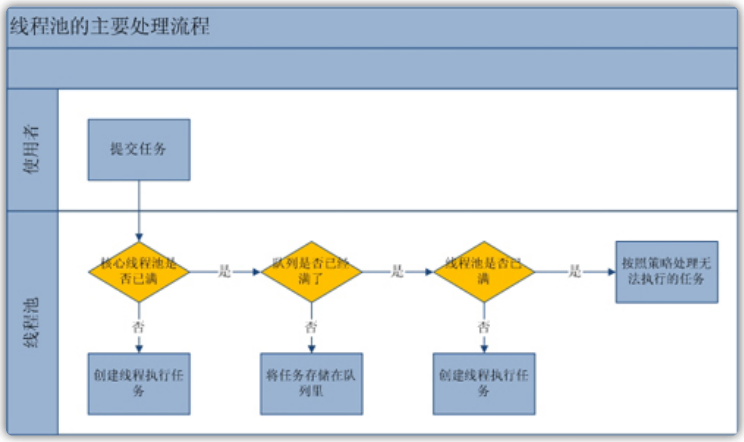
jdk1.8.0_144ThreadPoolExecutor下execute方法执行流程源码分析

> 当我第一次接触线程池，就非常好奇整个线程池执行策略是如何实现的，下面根据JDK源码讲述线程池执行流程，瞻仰大师笔法，在分析过程中我也会加入一下我自己的理解和问题，本篇文章只介绍execute()执行流程，关于线程池其他方面不做介绍，整个过程如有地方不对，不吝赐教，感谢

ThreadPoolExecutor执行策略



> 如上图（若侵删）所示，线程池执行策略主要分以下4步骤
>
> 1. 任务提交到线程池中，首先会检测线程池中核心线程数是否已经到达到达上限，若未到达则执行步骤2，反之执行步骤3
> 2. 初始化一条线程执行任务
> 3. 尝试将任务放到线程池下的工作队列中，若入队成功则等待其他线程执行完成从对列中取出执行，反之执行步骤4
> 4. 检查线程池内线程数量是否超出最大线程数量，若超出则拒绝执行任务，反之初始化一条线程执行此任务
>
> 总结来说就是任务提交到线程池，如果池内无核心线程则初始化一条线程执行任务，如果核心线程空闲则直接执行任务，如果核心线程都被占用则将任务放入到工作队列中，等待核心线程空闲，在从队列中获取任务，如果核心线程全部被占用而且工作队列也放满了则
> 初始化线程执行任务，线程数以最大线程数为限，线程数量达到最大线程数，还有任务进来，那么线程池将执行拒绝执行策略。
>
> 注意：这只是线程池在运行状态下的执行策略，只是一种理想状态下，整个执行策略并未受到线程池状态变化的影响，这部分会在源码分析中讨论

ThreadPoolExecutor源码分析

> 源码分析过程中对某一点的理解我会写在注释中，这样结合代码会更直观
>
> ThreadPoolExecutor的构造方法：

```
/**
 * Creates a new ThreadPoolExecutor with the given initial parameters.
 *
 * @param corePoolSize
 *          the number of threads to keep in the pool, even if they are
 *          idle, unless allowCoreThreadTimeOut is set
 *          核心线程数，线程池内常驻线程数量，即使这些线程是空闲状态。
 *            注意以下三点：
 *            1. 核心线程数只表示线程数量，没有必须指定哪一条线程为核心线程
 *            2. 如果将线程池allowCoreThreadTimeOut属性设置为true，此时核心
 *            线程数量将受keepAliveTime参数控制
 *            3. 此类线程初始化可能在任务提交之后逐一初始化或者执行prestartAllCoreThreads()
 *            方法预先初始化
 * @param maximumPoolSize
 *          the maximum number of threads to allow in the pool
```

```
     *            最大线程数量，线程池所能允许的最大线程数量，在系统提交的任务过多
     *            核心线程和工作队列都已经处理不过来时会初始化此类线程
     * @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.
     *            线程存活时间，对于超出核心线程数量的线程，如果它们暂无任务执行的
     *            情况下允许它们存活的时长
     * @param unit
     *          the time unit for the keepAliveTime argument
     *            线程存活时间的单位，具体参考java.util.concurrent.TimeUnit
     * @param workQueue
     *          the queue to use for holding tasks before they are executed.
     *          This queue will hold only the Runnable tasks submitted
     *          by the execute method.
     *            当核心线程处理不过来时，会将任务放在此队列中保存，等待线程处理
     * @param threadFactory
     *          the factory to use when the executor creates a new thread
     *            线程池中线程的均由此线程工厂创建
     * @param handler
     *          the handler to use when execution is blocked because the
     *          thread bounds and queue capacities are reached
     *            当线程池队列和最大线程数均达到上限或者线程池被终止，会拒绝执行
     *            任务，对于被拒绝执行的任务如何处理由此参数指定
     */
    public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
        long keepAliveTime, TimeUnit unit,
        BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
        RejectedExecutionHandler handler) {
      /**
       * 校验初始化参数合法性，此处需要注意的是corePoolSize可以等于0，
       * maximumPoolSize不可以等于0，否则线程池将无法使用。
       */
      if (corePoolSize < 0 || maximumPoolSize <= 0
          || maximumPoolSize < corePoolSize || keepAliveTime < 0)
        throw new IllegalArgumentException();
      if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
      this.acc = System.getSecurityManager() == null ? null
          : AccessController.getContext();
      this.corePoolSize = corePoolSize;
      this.maximumPoolSize = maximumPoolSize;
      this.workQueue = workQueue;
      // 统一将时间单位转换为纳秒，用它实现计时更加准确，不受系统的实时时钟的调整所影响
      this.keepAliveTime = unit.toNanos(keepAliveTime);
      this.threadFactory = threadFactory;
      this.handler = handler;
    }
```

在分析execute方法之前首先熟悉一下线程池下常用方法和属性，这些方法和属性在execute方法中会被多次调用

```
    /**
     * ctl用来保存高3位用来保存线程池状态，低29位用来保存线程池内线程数量
     * 二进制初始值：111 00000000000000000000000000000000
     */
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    /**
     * 初始值：29
     */
    private static final int COUNT_BITS = Integer.SIZE - 3;
    /**
     * CAPACITY表示线程池可容纳线程数量
     * 二进制初始值：000 11111111111111111111111111111
     * 十进制值：536870911
     */
    private static final int CAPACITY = (1 << COUNT_BITS) - 1;

    /**
     * 以下五个常量表示线程池五种状态，二进制值分别为
     * RUNNING:    111 00000000000000000000000000000000
     * SHUTDOWN:   000 00000000000000000000000000000000
     * STOP:       001 00000000000000000000000000000000
     * TIDYING:    010 00000000000000000000000000000000
```

```
 * TERMINATED: 011 00000000000000000000000000000000
 *
 * 线程池状态介绍：当线程池处于RUNING状态下则可以接收提交的任务和处理
 * 工作队列中的任务，处于SHUTTDOWN状态下则不再接收新任务，只处理工作队
 * 列中的任务，其它三种状态则既不接收新任务，也不会再处理工作队列中的
 * 任务，需要注意的是线程池状态值从RUNNING -> SHUTDOWN -> STOP ->
 * TIDYING -> TERMINATED 逐渐变化的，并且转换过程是不可逆的
 */
// runState is stored in the high-order bits
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;

/**
 * 通过传入ctl值与CAPACITY按位非之后的值~CAPACITY进行按位与操作获取线程池状态
 * ~CAPACITY: 111 00000000000000000000000000000000
 */
// Packing and unpacking ctl
private static int runStateOf(int c) {
  return c & ~CAPACITY;
}

/**
 * 通过传入ctl值与CAPACITY常量进行按位与操作，计算线程池中线程数量
 * CAPACITY: 000 11111111111111111111111111111
 */
private static int workerCountOf(int c) {
  return c & CAPACITY;
}

/**
 * 通过传入线程状态值rs和线程数量wc按位或获取ctl的值
 */
private static int ctlOf(int rs, int wc) {
  return rs | wc;
}
```

程序中会经常调用workerCountof和runStateOf方法来获取线程池内线程数量和线程池当前状态，所以要熟悉。

ThreadPoolExecutor的execute方法如下：

```
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.
   */
  int c = ctl.get();
  /**
   * 根据ctl值获取线程池内线程数，判断线程数是否小于核心线程数
   */
  if (workerCountOf(c) < corePoolSize) {
    /**
     * 上面判断成立则执行addWorker()，初始化一条线程执行提交的任务command
     */
    if (addWorker(command, true))
      /**
       * 初始化线程成功并返回
       */
```

```
            return;
        /**
         * 执行到此处有两个原因：
         * 1.线程池状态发生变化，已不再是RUNNING状态
         * 2.线程池内线程数已经达到或者超出核心线程数
         */
        c = ctl.get();
      }
      /**
       * 将任务放入队列，等待线程去取
       */
      if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (!isRunning(recheck) && remove(command))
          /**
           * 若状态有变且工作对列移除任务成功，则调用拒绝执行策略
           */
          reject(command);
        else if (workerCountOf(recheck) == 0)
          /**
           * 若线程池已无线程，则初始化一个不携带任务的线程，让它自动去工作队列中取任务执行
           */
          addWorker(null, false);
      /**
       * 执行到此处两个原因：
       * 1.线程池状态有变
       * 2.线程池核心线程数已满，工作队列已满
       * 将继续增加线程数，以maximumPoolSize为上限
       */
      } else if (!addWorker(command, false))
        /**
         * 线程数超出最大上限或者线程池已关闭，拒绝执行任务
         */
        reject(command);
    }
```

根据以上代码分析，线程池是如何初始化和管理线程是封装在addWorker方法内，addWorker方法如下：

```
    private boolean addWorker(Runnable firstTask, boolean core) {
      /**
       * 由于防止多个线程向同一个线程池内提交任务，会在此处发生竞争修改ctl的值，
       * 所以此处引入死循环，每个线程都会循环执行CAS操作修改ctl的值，若修改成功，
       * 则break跳出循环，否则继续在循环内竞争修改。总体来说是通过死循环和CAS操
       * 作来实现类似与锁的功能两层死循环，外层循环判断线程池状态变化，内层循环
       * 判断线程数量
       */
      retry: for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        /**
         * 判断两种情况：
         * 1.若线程是状态大于SHUTDOWN，（即STOP, TIDYING, TERMINATED）,则立即返回false，
         * 也就是说此三种状态，addWorker会返回false，初始化线程失败
         * 2.若线程是状态等于SHUTDOWN，firstTask != null或者workQueue.isEmpty()，则
         * 立即返回false，也就说此状态下，线程池不接受新任务(firstTask == null)，但是会处理
         * 已在队列中的任务(!workQueue.isEmpty())。
         * 3.若线程池状态为RUNNING，即可以处理新任务，也可以处理队列中任务。
         */
        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && !(rs == SHUTDOWN && firstTask == null
            && !workQueue.isEmpty()))
          return false;

        for (;;) {
          /**
           * 到此处说明线程池的状态校验已经通过，下面校验线程池内线程数量
           */
          int wc = workerCountOf(c);
          /**
           * 若线程数量超出总容量或者大于等于核心线程数或者最大线程数（具体大
           * 于哪个是根据addWorker方法的第二个参数core来判断的），若线程数
           * 量过多则返回。
```

```
     */
    if (wc >= CAPACITY || wc >= (core ? corePoolSize
        : maximumPoolSize))
      return false;
    /**
     * 线程数量校验若通过，则进行compareAndIncrementWorkerCount()下的
     * compareAndSwapInt来将ctl自增1，也就是将线程数量加1，compareAndSwapInt(CAS)
     * 是原子操作，也就是说自增的过程不会被打断，但是CAS有可能会失败，是因为有其他线程已经修
     * 改过ctl的值。
     */
    if (compareAndIncrementWorkerCount(c))
      /**
       * CAS操作成功则跳出最外层循环
       * 注意：此时虽然ctl值已经加1，但是还并未初始化一条线程。
       */
      break retry;
    /**
     * CAS操作失败，说明ctl已经被修改，重新读取值，再次循环
     */
    c = ctl.get(); // Re-read ctl
    /**
     * 判断此时线程池状态与进入内存循环之前相比是否发生变化，若不一致
     * 则需要跳到外层循环再次校验一下线程池状态是否满足标准，若一致则
     * 只需要在内层循环
     */
    if (runStateOf(c) != rs)
      continue retry;
    // else CAS failed due to workerCount change; retry inner loop
  }
}

/**
 * 到达此处说明ctl安全赋值完成，与之对应是立即初始化一个线程，否则ctl的低29
 * 位值将无法正确表示线程数
 */
boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
  /**
   * 初始化一个Worker对象，此对象内包含一个线程的引用
   */
  w = new Worker(firstTask);
  final Thread t = w.thread;
  /**
   * 此处有必要校验Worker下的线程是否为空，在Worker类的构造函数下会发现，
   * thread是由ThreadPoolExecutor初始化是指定的ThreadFactory创建的，
   * 虽然初始化ThreadPoolExecutor时已经校验ThreadFactory不为空，但是
   * 无法保证ThreadFactory生产出来的线程不为空。
   */
  if (t != null) {
    final ReentrantLock mainLock = this.mainLock;
    /**
     * 获取全局锁对象
     */
    mainLock.lock();
    try {
      // Recheck while holding lock.
      // Back out on ThreadFactory failure or if
      // shut down before lock acquired.
      int rs = runStateOf(ctl.get());

      /**
       * 由于修改线程池状态的方法执行都需要获取mainLock，再次已经获取mainLock
       * 其它修改同步方法无法执行，此时获取线程池状态是准确的，校验也是有必要的
       */
      if (rs < SHUTDOWN || (rs == SHUTDOWN
          && firstTask == null)) {
        // precheck that t is startable
        if (t.isAlive())
          throw new IllegalThreadStateException();
        /**
         * workers全篇只在锁内使用，无需用volatile关键字修饰
         * 将worker装入workers容器中的原因是为了在后续终止worker线程使用
         */
```

```
          workers.add(w);
          int s = workers.size();
          if (s > largestPoolSize)
            largestPoolSize = s;
          workerAdded = true;
        }
      } finally {
        mainLock.unlock();
      }
      if (workerAdded) {
        /**
         * 启动执行工作任务的线程，实际上是执行的Worker类下的run方法
         */
        t.start();
        workerStarted = true;
      }
    }
  } finally {
    if (!workerStarted)
      /**
       * 添加工作线程失败，执行后续收尾工作，将当前worker从workers容器中清除
       * 并将ctl值自减一
       */
      addWorkerFailed(w);
  }
  return workerStarted;
}
```

从addWorker方法中可以看出具体的线程执行是首先初始化一个Worker类，此类中包含一个Thread对象的引用，最终线程的启动是启动Worker下的thread，Worker类是ThreadPoolExecutor下的内部类，主要代码（去除无关代码）如下：

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

  final Thread thread;
  Runnable firstTask;

  Worker(Runnable firstTask) {
    // inhibit interrupts until runWorker
    setState(-1);
    // 将线程池提交的任务传给Worker下的firstTask
    this.firstTask = firstTask;
    // getThreadFactory()会返回ThreadPoolExecutor初始化时的传入的线程工厂，此工厂接口
    // 下有newThread方法有来初始化线程
    // 让线程工厂以当前Worker对象为参数生产线程，并返回给此Worker对象下的thread属性
    // 也就是说如果Worker.thread如果启动，将执行Worker类型的run方法
    this.thread = getThreadFactory().newThread(this);
  }

  public void run() {
    // 此处就是线程如何执行提交的任务
    runWorker(this);
  }
}
```

在Worker类中的run方法直接调用runWorker，代码如下：

```
final void runWorker(Worker w) {
  Thread wt = Thread.currentThread();
  Runnable task = w.firstTask;
  w.firstTask = null;
  // allow interrupts
  w.unlock();
  boolean completedAbruptly = true;
  try {
    /**
     * while循环获取task，如果返回为空，线程就会死亡，线程池之所以能够重用现有线程的原因也就是
     * 在这里，它会循环获取task，可能是线程池刚提交的task，也可能是从线程池下工作队列workQueue
     * 中取得
     */
```

```
    while (task != null || (task = getTask()) != null) {
      /**
       * 注意此处获取的是worker的锁，并非是mainLock锁，如果是获取的是mainLock，
       * 那么程序到此处就变成串行执行，线程池也就失去意义了，此处获取worker锁的原因
       * 在于与线程池中断当前worker线程是互斥，也就是说在执行任务时，无法中断线程，
       * 具体实现请参考shutdown();
       */
      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
      // 再次判断线程池状态
      if ((runStateAtLeast(ctl.get(), STOP) || (Thread.interrupted()
          && runStateAtLeast(ctl.get(), STOP))) && !wt
              .isInterrupted())
        wt.interrupt();
      try {
        // 钩子函数，由子类去实现，可以在方法内记录时间等信息
        beforeExecute(wt, task);
        Throwable thrown = null;
        try {
          /**
           * 注意此处执行的是run方法，并未将提交的task封装成线程启动，而是方法调用，
           * 这可能会误认为是串行执行，其实不然，task.run()是放在worker的run方法内
           * 执行，而worker是被封装成线程启动的，从始至终线程池从未将提交的Runnable
           * 任务封装成Thread执行，而是将它交给内部类Worker，由Worker启动执行
           * 其简单模型如下：
           * new Thread(new Worker(task) {
           *     Runnable task = task;
           *     @Override
           *     public void run() {
           *            task.run();
           *     }
           * }).start();
           * 这么做的好处是在task.run()执行先后可以切入一些操作，例如beforeExecute(wt, task)
           * 和afterExecute(task, thrown)方法，这些操作与task下业务代码天生隔离，例如计算
           * 线程池执行每个任务的时间，就可以在beforeExecute和afterExecute中记录起止时间并
           * 计算，否则你可能需要在task.run()下记录起止时间，这样代码耦合太严重。
           */
          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 {
          // 与beforeExecute一样，钩子函数，由子类去实现，可以在方法内记录时间等信息
          afterExecute(task, thrown);
        }
      } finally {
        // 任务执行完成显式置空，防止误导下次while循环
        task = null;
        w.completedTasks++;
        w.unlock();
      }
    }
    completedAbruptly = false;
  } finally {
    /**
     * 执行完成任务，进行一些收尾工作，这个方法执行完成，当前线程就会死亡
     */
    processWorkerExit(w, completedAbruptly);
  }
}
```

> 上文提到线程池能够复用线程，是因为runWorker方法下的while循环让他保持活性，如果跳出while循环则线程执行完成runWorker方法将会死亡，要想线程复用关键在于while循环条件是如何判断的，引申出getTask方法，这个方法很巧妙，代码如下：

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

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

    // Check if queue empty only if necessary.
    /**
     * 1.线程池状态为SHUTDOWN且workQueue.isEmpty();
     * 2.线程池状态不小于STOP;
     * 此两种情况会执行decrementWorkerCount将ctl值自减1，然后返回null，
     * 返回null会跳出上层while()循环，意味着此线程快要死亡了
     */
    if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
      // 设置ctl值自减一
      decrementWorkerCount();
      return null;
    }

    int wc = workerCountOf(c);

    /**
     * 根据allowCoreThreadTimeOut和wc > corePoolSize判断线程是否存在生命周期
     * allowCoreThreadTimeOut表示是否允许核心线程超时死亡，默认是false
     * 若allowCoreThreadTimeOut为false，则判断线程数是否超出核心线程数
     */
    // Are workers subject to culling?
    boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

    if ((wc > maximumPoolSize || (timed && timedOut)) && (wc > 1
        || workQueue.isEmpty())) {
      // 设置ctl值自减一
      if (compareAndDecrementWorkerCount(c))
        return null;
      continue;
    }

    try {
      /**
       * 线程池初始化时传入的keepAliveTime将在此应用，线程在keepAliveTime规定
       * 的时间内从阻塞队列中获取任务，若获取不到，则线程会死亡，以此来表示线
       * 程生命周期
       */
      Runnable r = timed ? workQueue.poll(keepAliveTime,
          TimeUnit.NANOSECONDS) : workQueue.take();
      if (r != null)
        return r;
      timedOut = true;
    } catch (InterruptedException retry) {
      timedOut = false;
    }
  }
}
```

以上就是线程池如何执行任务的整个流程，整个过程讲述比较笼统，具体详细信息请阅读jdk下ThreadPoolExecutor源码，建议在工作中尽量用线程池来代替传统的new Thread().start()启动线程，线程池有主要优点如下：

1. 它将线程的初始化和线程的执行分开，线程初始化由线程池维护
2. 线程池的执行策略可以使你的程序具有更好的可伸缩性，何时初始化核心线程，何时提交任务到阻塞队列，何时初始化非核心线程及线程的生命周期均由线程池操作，你只需要设置执行策略并提交任务就可以了
3. 线程池可以达到线程的复用，有时会省去线程的初始化时间，整体提升程序性能

我在阅读源码时，曾感到疑惑的几个问题如下：

1. 线程池为什么要封装一个内部类Worker，既然execute方法参数是Runnable，为什么不直接调用线程工厂初始化线程执行？
2. 线程池是如何维护池内线程存活时间的？
3. 线程池没有任务时，核心线程将处于空闲状态，但是又不会死亡，他是如何维护这种状态的？

> 4. 线程池本身是线程安全的类吗，也就是说线程池在多条线程操作的时候会保证执行结果的正确性吗，在execute方法中，
>
>   Doug Lea大神几乎很少使用锁，而且锁的范围非常小