链滴

从源码的角度解析线程池运行原理

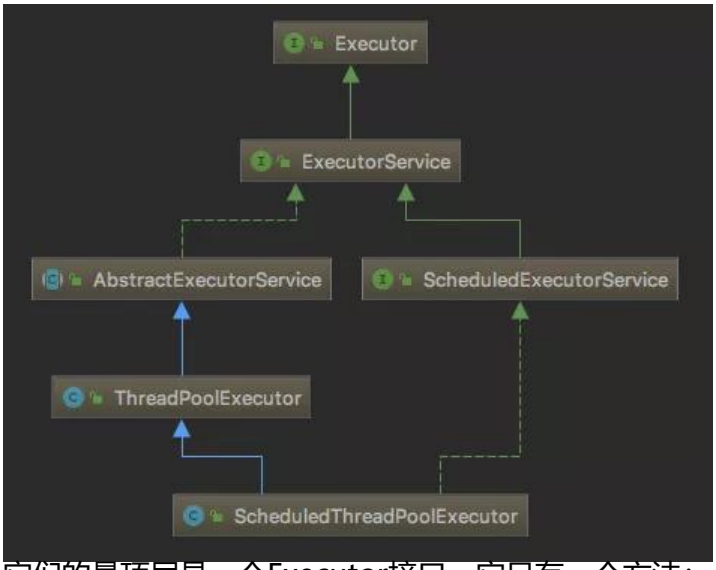
作者：XieWeiZM

原文链接：https://ld246.com/article/1568703564420

来源网站：链滴

ThreadPoolExecutor

在深入源码之前先来看看J.U.C包中的线程池类图：



它们的最顶层是一个Executor接口，它只有一个方法：

```
public interface Executor {
    void execute(Runnable command);
}
```

它提供了一个运行新任务的简单方法，Java线程池也称之为Executor框架。

ExecutorService扩展了Executor，添加了操控线程池生命周期的方法，如shutDown()，shutDownow()等，以及扩展了可异步跟踪执行任务生成返回值Future的方法，如submit()等方法。

ThreadPoolExecutor继承自AbstractExecutorService，同时实现了ExecutorService接口，也是Executor框架默认的线程池实现类，也是这篇文章重点分析的对象，一般我们使用线程池，如没有特殊要，直接创建ThreadPoolExecutor，初始化一个线程池，如果需要特殊的线程池，则直接继承ThreadPolExecutor，并实现特定的功能，如ScheduledThreadPoolExecutor，它是一个具有定时执行任务线程池。

下面我们开始ThreadPoolExecutor的源码分析了（以下源码为JDK8版本）：

ctl变量

ctl是一个Integer值，它是对线程池运行状态和线程池中有效线程数量进行控制的字段，Integer值一有32位，其中高3位表示"线程池状态"，低29位表示"线程池中的任务数量"。我们看看Doug Lea大神如何实现的：

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
private static final int COUNT_BITS = Integer.SIZE - 3;
private static final int CAPACITY = (1 << COUNT_BITS) - 1;

// 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;

// Packing and unpacking ctl
// 通过位运算获取线程池运行状态
private static int runStateOf(int c)     { return c & ~CAPACITY; }
// 通过位运算获取线程池中有效的工作线程数
private static int workerCountOf(int c)  { return c & CAPACITY; }
// 初始化ctl变量值
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

线程池一共有状态5种状态，分别是：

1. Running：线程池初始化时默认的状态，表示线程正处于运行状态，能够接受新提交的任务，同时能够处理阻塞队列中的任务；

2. SHUTDOWN：调用shutdown()方法会使线程池进入到该状态，该状态下不再继续接受新提交的务，但是还会处理阻塞队列中的任务；

3. STOP：调用shutdownNow()方法会使线程池进入到该状态，该状态下不再继续接受新提交的任务同时不再处理阻塞队列中的任务；

4. TIDYING：如果线程池中workerCount=0，即有效线程数量为0时，会进入该状态；

5. TERMINATED：在terminated()方法执行完后进入该状态，只不过terminated()方法需要我们自实现。

我们再来看看位运算：

COUNT_BITS表示ctl变量中表示有效线程数量的位数，这里COUNT_BITS=29；

CAPACITY表示最大有效线程数，根据位运算得出COUNT_MASK=11111111111111111111111111111，这算成十进制大约是5亿，在设计之初就已经想到不会开启超过5亿条线程，所以完全够用了；

线程池状态的位运算得到以下值：

1. RUNNING：高三位值111

2. SHUTDOWN：高三位值000

3. STOP：高三位值001

4. TIDYING：高三位值010

5. TERMINATED：高三位值011

这里简单解释一下Doug Lea大神为什么使用一个Integer变量表示两个值：

很多人会想，一个变量表示两个值，就节省了存储空间，但是这里很显然不是为了节省空间而设计的即使将这辆个值拆分成两个Integer值，一个线程池也就多了4个字节而已，为了这4个字节而去大费章地设计一通，显然不是Doug Lea大神的初衷。

在多线程的环境下，运行状态和有效线程数量往往需要保证统一，不能出现一个改而另一个没有改的况，如果将他们放在同一个AtomicInteger中，利用AtomicInteger的原子操作，就可以保证这两个始终是统一的。

Doug Lea大神牛逼！

Worker

Worker类继承了AQS，并实现了Runnable接口，它有两个重要的成员变量：firstTask和thread。firsTask用于保存第一次新建的任务；thread是在调用构造方法时通过ThreadFactory来创建的线程，是来处理任务的线程。

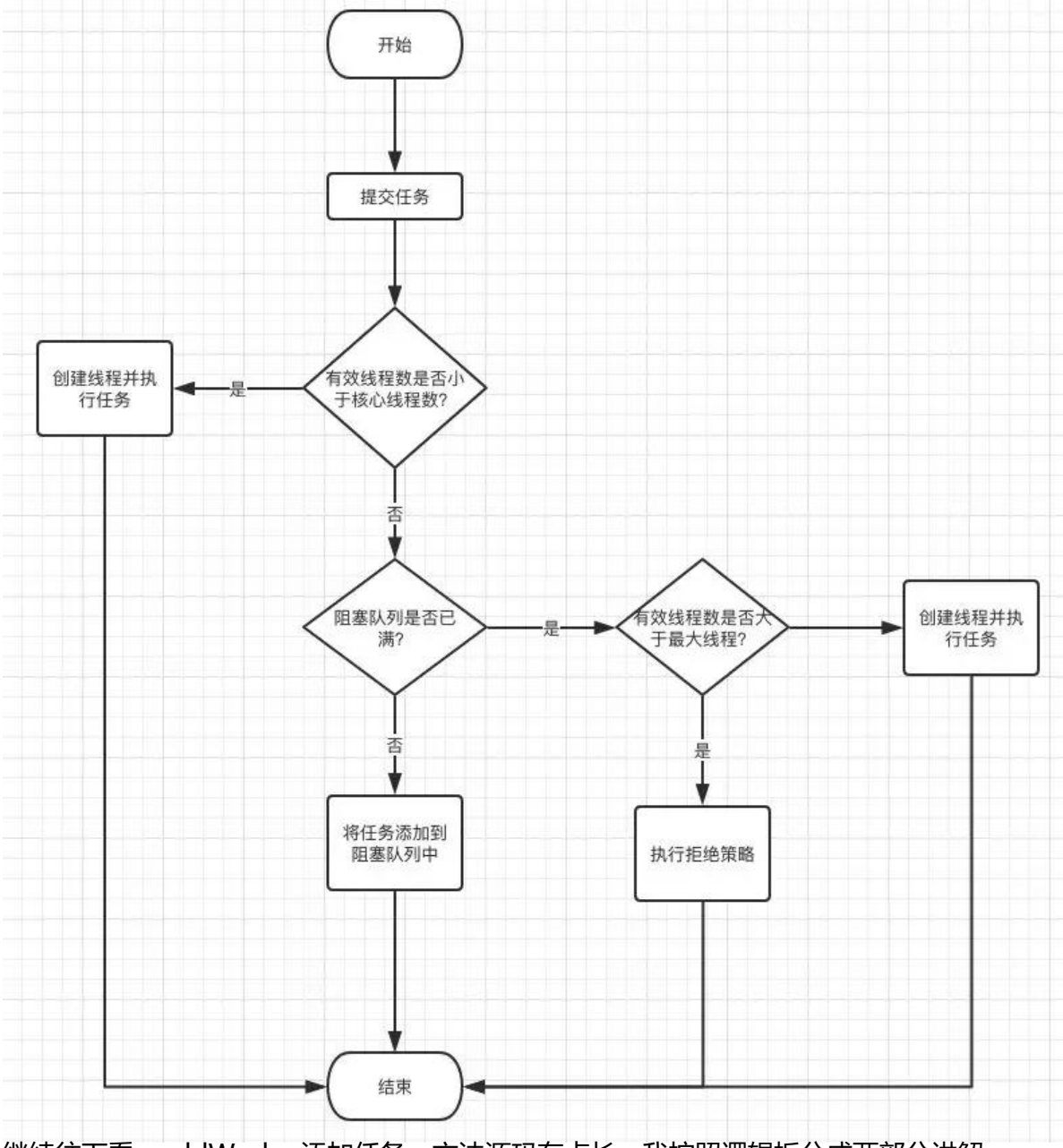
如何在线程池中添加任务？

线程池要执行任务，那么必须先添加任务，execute()虽说是执行任务的意思，但里面也包含了添加任的步骤在里面，下面源码：

java.util.concurrent.ThreadPoolExecutor#execute：

```
public void execute(Runnable command) {
  // 如果添加订单任务为空，则空指针异常
  if (command == null)
    throw new NullPointerException();
  // 获取ctl值
  int c = ctl.get();
  // 1.如果当前有效线程数小于核心线程数，调用addWorker执行任务（即创建一条线程执行该任务）
  if (workerCountOf(c) < corePoolSize) {
    if (addWorker(command, true))
      return;
    c = ctl.get();
  }
  // 2.如果当前有效线程大于等于核心线程数，并且当前线程池状态为运行状态，则将任务添加到阻
队列中，等待空闲线程取出队列执行
  if (isRunning(c) && workQueue.offer(command)) {
    int recheck = ctl.get();
    if (! isRunning(recheck) && remove(command))
      reject(command);
    else if (workerCountOf(recheck) == 0)
      addWorker(null, false);
  }
  // 3.如果阻塞队列已满，则调用addWorker执行任务（即创建一条线程执行该任务）
  else if (!addWorker(command, false))
    // 如果创建线程失败，则调用线程拒绝策略
    reject(command);
}
```

我在这里画一下execute执行任务的流程图：



继续往下看，addWorker添加任务，方法源码有点长，我按照逻辑拆分成两部分讲解：

java.util.concurrent.ThreadPoolExecutor#addWorker：

```
retry:
for (;;) {
  int c = ctl.get();
  // 获取线程池当前运行状态
  int rs = runStateOf(c);

  // 如果rs大于SHUTDOWN，则说明此时线程池不在接受新任务了
  // 如果rs等于SHUTDOWN，同时满足firstTask为空，且阻塞队列如果有任务，则继续执行任务
  // 也就说明了如果线程池处于SHUTDOWN状态时，可以继续执行阻塞队列中的任务，但不能继续线程池中添加任务了
  if (rs >= SHUTDOWN &&
      ! (rs == SHUTDOWN &&
         firstTask == null &&
         ! workQueue.isEmpty()))
```

```
    return false;

  for (;;) {
    // 获取有效线程数量
    int wc = workerCountOf(c);
    // 如果有效线程数大于等于线程池所容纳的最大线程数（基本不可能发生），不能添加任务
    // 或者有效线程数大于等于当前限制的线程数，也不能添加任务
    // 限制线程数量有任务是否要核心线程执行决定，core=true使用核心线程执行任务
    if (wc >= CAPACITY ||
        wc >= (core ? corePoolSize : maximumPoolSize))
      return false;
    // 使用AQS增加有效线程数量
    if (compareAndIncrementWorkerCount(c))
      break retry;
    // 如果再次获取ctl变量值
    c = ctl.get();  // Re-read ctl
    // 再次对比运行状态，如果不一致，再次循环执行
    if (runStateOf(c) != rs)
      continue retry;
    // else CAS failed due to workerCount change; retry inner loop
  }
}
```

这里特别强调，firstTask是开启线程执行的首个任务，之后常驻在线程池中的线程执行的任务都是从塞队列中取出的，需要注意。

以上for循环代码主要作用是判断ctl变量当前的状态是否可以添加任务，特别说明了如果线程池处于SUTDOWN状态时，可以继续执行阻塞队列中的任务，但不能继续往线程池中添加任务了；同时增加作线程数量使用了AQS作同步，如果同步失败，则继续循环执行。

```
// 任务是否已执行
boolean workerStarted = false;
// 任务是否已添加
boolean workerAdded = false;
// 任务包装类，我们的任务都需要添加到Worker中
Worker w = null;
try {
  // 创建一个Worker
  w = new Worker(firstTask);
  // 获取Worker中的Thread值
  final Thread t = w.thread;
  if (t != null) {
    // 操作workers HashSet 数据结构需要同步加锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
      // Recheck while holding lock.
      // Back out on ThreadFactory failure or if
      // shut down before lock acquired.
      // 获取当前线程池的运行状态
      int rs = runStateOf(ctl.get());
      // rs < SHUTDOWN表示是RUNNING状态；
      // 如果rs是RUNNING状态或者rs是SHUTDOWN状态并且firstTask为null，向线程池中添加线程

      // 因为在SHUTDOWN时不会在添加新的任务，但还是会执行workQueue中的任务
```

```
      // rs是RUNNING状态时，直接创建线程执行任务
      // 当rs等于SHUTDOWN时，并且firstTask为空，也可以创建线程执行任务，也说说明了SHUTD
WN状态时不再接受新任务
      if (rs < SHUTDOWN ||
          (rs == SHUTDOWN && firstTask == null)) {
        if (t.isAlive()) // precheck that t is startable
          throw new IllegalThreadStateException();
        workers.add(w);
        int s = workers.size();
        if (s > largestPoolSize)
          largestPoolSize = s;
        workerAdded = true;
      }
    } finally {
      mainLock.unlock();
    }
    // 启动线程执行任务
    if (workerAdded) {
      t.start();
      workerStarted = true;
    }
  }
} finally {
  if (! workerStarted)
    addWorkerFailed(w);
}
return workerStarted;
}
```

以上源码主要的作用是创建一个Worker对象，并将新的任务装进Worker中，开启同步将Worker添进workers中，这里需要注意workers的数据结构为HashSet，非线程安全，所以操作workers需要加步锁。添加步骤做完后就启动线程来执行任务了，继续往下看。

如何执行任务?

我们注意到上面的代码中：

```
// 启动线程执行任务
if (workerAdded) {
  t.start();
  workerStarted = true;
}
```

这里的t是w.thread得到的，即是Worker中用于执行任务的线程，该线程由ThreadFactory创建，我再看看生成Worker的构造方法：

```
Worker(Runnable firstTask) {
  setState(-1); // inhibit interrupts until runWorker
  this.firstTask = firstTask;
  this.thread = getThreadFactory().newThread(this);
}
```

newThread传的参数是Worker本身，而Worker实现了Runnable接口，所以当我们执行t.start()时，行的是Worker的run()方法**，找到入口了：

java.util.concurrent.ThreadPoolExecutor.Worker#run：

```
public void run() {
  runWorker(this);
}
```

java.util.concurrent.ThreadPoolExecutor#runWorker：

```
final void runWorker(Worker w) {
  Thread wt = Thread.currentThread();
  Runnable task = w.firstTask;
  w.firstTask = null;
  w.unlock(); // allow interrupts
  boolean completedAbruptly = true;
  try {
    // 循环从workQueue阻塞队列中获取任务并执行
    while (task != null || (task = getTask()) != null) {
      // 加同步锁的目的是为了防止同一个任务出现多个线程执行的问题
      w.lock();
      // 如果线程池正在关闭，须确保中断当前线程
      if ((runStateAtLeast(ctl.get(), STOP) ||
           (Thread.interrupted() &&
            runStateAtLeast(ctl.get(), STOP))) &&
          !wt.isInterrupted())
        wt.interrupt();
      try {
        // 执行任务前可以做一些操作
        beforeExecute(wt, task);
        Throwable thrown = null;
        try {
          // 执行任务
          task.run();
        } catch (RuntimeException x) {
          thrown = x; throw x;
        } catch (Error x) {
          thrown = x; throw x;
        } catch (Throwable x) {
          thrown = x; throw new Error(x);
        } finally {
          // 执行任务后可以做一些操作
          afterExecute(task, thrown);
        }
      } finally {
        // 将task置为空，让线程自行调用getTask()方法从workQueue阻塞队列中获取任务
        task = null;
        // 记录Worker执行了多少次任务
        w.completedTasks++;
        w.unlock();
      }
    }
    completedAbruptly = false;
  } finally {
    // 线程回收过程
    processWorkerExit(w, completedAbruptly);
```

原文链接：从源码的角度解析线程池运行原理

```
  }
}
```

这一步是执行任务的核心方法，首次执行不为空的firstTask任务，之后便一直从workQueue阻塞队中获取任务并执行，如果你想在任务执行前后做点啥不可告人的小动作，你可以实现ThreadPoolExector以下两个方法：

```
protected void beforeExecute(Thread t, Runnable r) { }
protected void afterExecute(Runnable r, Throwable t) { }
```

这样一来，我们就可以对任务的执行进行实时监控了。

这里还需要注意，在finally块中，将task置为空，目的是为了让线程自行调用getTask()方法从workQueue阻塞队列中获取任务。

如何保证核心线程不被销毁?

我们之前已经知道线程池中可维持corePoolSize数量的常驻核心线程，那么它们是如何保证执行完任而不被线程池回收的呢？**在前面的章节中你可能已经到从workQueue队列中会阻塞式地获取任务，果没有获取任务，那么就会一直阻塞下去**，很聪明，你已经知道答案了，现在我们来看Doug Lea大是如何实现的。

java.util.concurrent.ThreadPoolExecutor#getTask：

```
private Runnable getTask() {
  // 超时标记，默认为false，如果调用workQueue.poll()方法超时了，会标记为true
  // 这个标记非常之重要，下面会说到
  boolean timedOut = false;

  for (;;) {
    // 获取ctl变量值
    int c = ctl.get();
    int rs = runStateOf(c);

    // 如果当前状态大于等于SHUTDOWN，并且workQueue中的任务为空或者状态大于等于STOP
    // 则操作AQS减少工作线程数量，并且返回null，线程被回收
    // 也说明假设状态为SHUTDOWN的情况下，如果workQueue不为空，那么线程池还是可以继续行剩下的任务
    if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
      // 操作AQS将线程池中的线程数量减一
      decrementWorkerCount();
      return null;
    }

    // 获取线程池中的有效线程数量
    int wc = workerCountOf(c);

    // 如果开发者主动开启allowCoreThreadTimeOut并且获取当前工作线程大于corePoolSize，那该线程是可以被超时回收的
    // allowCoreThreadTimeOut默认为false，即默认不允许核心线程超时回收
    // 这里也说明了在核心线程以外的线程都为“临时”线程，随时会被线程池回收
    boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

    // 这里说明了两点销毁线程的条件：
```

```
    // 1.原则上线程池数量不可能大于maximumPoolSize，但可能会出现并发时操作了setMaximum
oolSize方法，如果此时将最大线程数量调少了，很可能会出现当前工作线程大于最大线程的情况，
时就需要线程超时回收，以维持线程池最大线程小于maximumPoolSize，
    // 2.timed && timedOut 如果为true，表示当前操作需要进行超时控制，这里的timedOut为tru
，说明该线程已经从workQueue.poll()方法超时了
    // 以上两点满足其一，都可以触发线程超时回收
    if ((wc > maximumPoolSize || (timed && timedOut))
        && (wc > 1 || workQueue.isEmpty())) {
      // 尝试用AQS将线程池线程数量减一
      if (compareAndDecrementWorkerCount(c))
        // 减一成功后返回null，线程被回收
        return null;
      // 否则循环重试
      continue;
    }

    try {
      // 如果timed为true，阻塞超时获取任务，否则阻塞获取任务
      Runnable r = timed ?
        workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
      workQueue.take();
      if (r != null)
        return r;
      // 如果poll超时获取任务超时了, 将timeOut设置为true
      // 继续循环执行，如果碰巧开发者开启了allowCoreThreadTimeOut，那么该线程就满足超时回
了
      timedOut = true;
    } catch (InterruptedException retry) {
      timedOut = false;
    }
  }
}
```

我把我对getTask()方法源码的深度解析写在源码对应的地方了，该方法就是实现默认的情况下核心程不被销毁的核心实现，其实现思路大致是：

1. 将timedOut超时标记默认设置为false；

2. 计算timed的值，该值决定了线程的生死大权，(timed && timedOut) 即是线程超时回收的条件一，需要注意的是第一次(timed && timedOut) 为false，因为timedOut默认值为false，此时还没到oll超时获取的操作；

3. 根据timed值来决定是用阻塞超时获取任务还是阻塞获取任务，如果用阻塞超时获取任务，超时后tmedOut会被设置为true，接着继续循环，若 (timed && timedOut) 为true，满足线程超时回收。

呕心沥血的一篇源码解读到此结束，希望能助同学们彻底吃透线程池的底层原理，以后遇到面试官问线程池的问题，你就说看过科代表的线程池源码解读，面试官这时就会夸你：

这同学基础真扎实！