线程池源码解析+设计思想+线程池监控框架设计

1. 线程池理论基础

ctl

```
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;


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;


// 获得线程池的状态
private static int runStateOf(int c)     { return c & ~CAPACITY; }
// 获得线程池里面线程的数量
private static int workerCountOf(int c)  { return c & CAPACITY; }
// 根据 给定的线程状态 + 线程的数量设置 ctl
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

`ctl`是对线程池的运行状态和线程池中有效线程的数量进行控制的一个字段，它包含两部分的信息:
线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)，这里可以看到，使用了Integer类型来保存，高3位保存runState，低29位保存workerCount。COUNT_BITS 就是29，CAPACITY就是1左移29位减1（29个1），这个常量表示workerCount的上限值，`大约是5亿`

线程池状态

RUNNING

能接受新提交的任务，并且也能处理阻塞队列中的任务；

SHUTDOWN

关闭状态，**不再接受新提交的任务**，**但却可以继续处理阻塞队列中已保存的任务 ， 会中断处于空闲状态的线程**。在线程池处于 RUNNING 状态时 ，调用 shutdown()方法会使线程池进入到该状态。

STOP

不能接受新任务，**也不处理队列中的任务**，**会中断正在处理任务的线程**。在线程池处于 RUNNING 或 SHUTDOWN 状态时，调用 shutdownNow() 方法会使线程池进入到该状态；

TIDYING

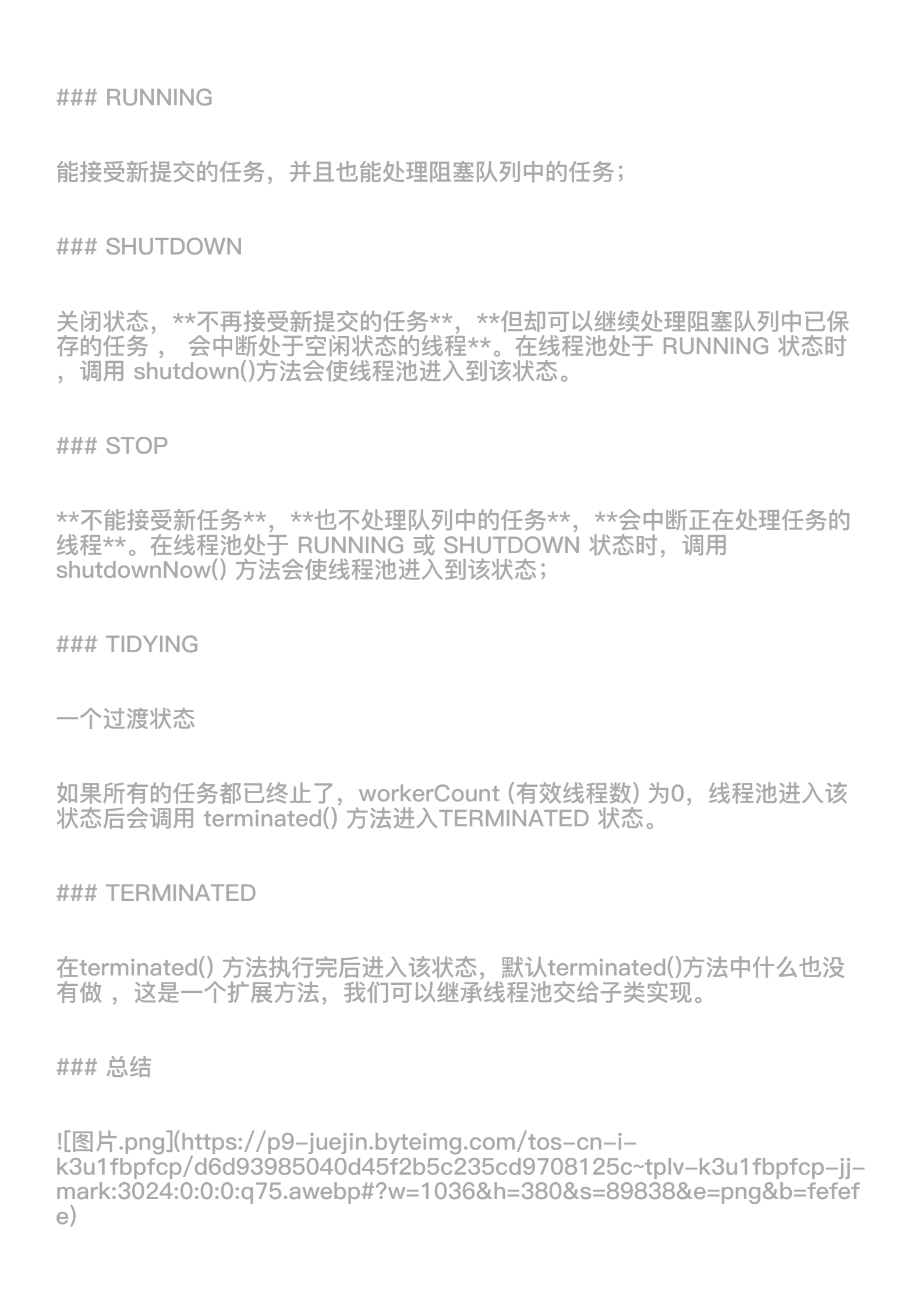
一个过渡状态

如果所有的任务都已终止了，workerCount (有效线程数) 为0，线程池进入该状态后会调用 terminated() 方法进入TERMINATED 状态。

TERMINATED

在terminated() 方法执行完后进入该状态，默认terminated()方法中什么也没有做 ，这是一个扩展方法，我们可以继承线程池交给子类实现。

总结



SHUTDOWN 状态下 需要阻塞队列为空（因为 **SHUTDOWN** 状态下需要处理阻塞队列里面已经存在的任务），线程池的工作线程数量为0， 才能转化为 TIDYING状态

STOP 状态下 不需要阻塞队列为空（因为 **STOP** 状态下不会接受任何任务）只需要线程池的工作线程数量为0， 就能转化为 TIDYING状态

我们还可以看到状态之间的一个大小关系。

**RUNNING < SHUTDOWN < STOP < TIDYING < TERMINATED ，而且线程池的一个状态转换过程也是
RUNNING -> SHUTDOWN -> STOP -> TIDYING -> TERMINATED。所以这么设计会在代码里面的状态判断部分十分方便。**

ThreadPoolExecutor构造方法

```
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;
}
```

构造方法中的字段含义如下：

* **corePoolSize**：核心线程数量，当有新任务在execute()方法提交时，会执行以下判断：

+ 如果运行的线程少于 corePoolSize，则创建新线程来处理任务，即使线程池中的其他线程是空闲的；
+ 如果线程池中的线程数量大于等于 corePoolSize 且小于 maximumPoolSize，则只有当**workQueue满** 时才创建新的线程去处理任务；
+ 如果设置的corePoolSize 和 maximumPoolSize相同，则创建的线程池的大小是固定的，这时如果有新任务提交，若workQueue未满，则将请求放入workQueue中，等待有空闲的线程去从workQueue中取任务并处理；
+ 如果运行的线程数量大于等于maximumPoolSize，这时如果workQueue已经满了，则通过handler所指定的策略来处理任务；
* 所以，任务提交时，判断的顺序为 corePoolSize —> workQueue —> maximumPoolSize。
* **maximumPoolSize**：最大线程数量；
* **workQueue**：保存等待执行的任务的阻塞队列，当提交一个新的任务到线程池以后, 线程池会根据当前线程池中正在运行着的线程的数量来决定对该任务的处理方式，主要有以下几种处理方式:

+ **使用无界队列**：一般使用基于链表的阻塞队列LinkedBlockingQueue。如果使用这种方式，那么线程池中能够创建的**最大线程数就是corePoolSize**，而maximumPoolSize就不会起作用了。当线程池中所有的核心线程都是RUNNING状态时，这时一个新的任务提交就会放入等待队列中。
+ **使用有界队列**：一般使用ArrayBlockingQueue。使用该方式可以将线程池的最大线程数量限制为maximumPoolSize，这样能够降低资源的消耗，但同时这种方式也使得线程池对线程的调度变得更困难，因为线程池和队列的容量都是有限的值，所以要想使线程池处理任务的吞吐率达到一个相对合理的范围，又想使线程调度相对简单，并且还要尽可能的降低线程池对资源的消耗，就需要合理的设置这两个数量。

1. 如果要想降低系统资源的消耗（包括CPU的使用率，操作系统资源的消耗，上下文环境切换的开销等），可以设置**较大的队列容量**和**较小的线程池容量**, 但这样也会降低线程处理任务的吞吐量。
2. 如果提交的任务经常发生阻塞，那么可以考虑通过调用 setMaximumPoolSize() 方法来重新设定线程池的容量。
3. 如果队列的容量设置的较小，通常需要将线程池的容量设置大一点，这样CPU的使用率会相对的高一些。但如果线程池的容量设置的过大，则在提交的任务数量太多的情况下，并发量会增加，那么线程之间的调度就是一个要考虑的问题，因为这样反而有可能降低处理任务的吞吐量。
* **keepAliveTime**：线程池维护线程所允许的空闲时间。当线程池中的线程

数量大于corePoolSize的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了keepAliveTime；
* **threadFactory**：它是ThreadFactory类型的变量，用来创建新线程。默认使用Executors.defaultThreadFactory() 来创建线程。使用默认的ThreadFactory来创建线程时，会使新创建的线程具有相同的NORM_PRIORITY优先级并且是非守护线程，同时也设置了线程的名称。
* **handler**：它是RejectedExecutionHandler类型的变量，表示线程池的饱和策略。如果阻塞队列满了并且没有空闲的线程，这时如果继续提交任务，就需要采取一种策略处理该任务。线程池提供了4种策略：

+ AbortPolicy：直接抛出异常，这是默认策略；
+ CallerRunsPolicy：用调用者所在的线程来执行任务；
+ DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
+ DiscardPolicy：直接丢弃任务；

线程池拒绝策略

拒绝策略体现的设计模式

拒绝策略是策略模式的体现。将方法的实现交给外部去决定。

JDK自带的拒绝策略

```
public static class AbortPolicy implements RejectedExecutionHandler {
    public AbortPolicy() { }
    //直接抛出异常
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        throw new RejectedExecutionException("Task " + r.toString() +
                                             " rejected from " +
                                             e.toString());
    }
}

public static class CallerRunsPolicy implements
RejectedExecutionHandler {

    public CallerRunsPolicy() { }
```

```
    //由调用excute方法 的线程自己执行
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
}



public static class DiscardPolicy implements RejectedExecutionHandler {
    public DiscardPolicy() { }
    //什么都不做
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    }
}



public static class DiscardOldestPolicy implements
RejectedExecutionHandler {

    public DiscardOldestPolicy() { }

    //把阻塞队列头部的任务淘汰 ， 然后塞入最新的任务
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            e.getQueue().poll();
            e.execute(r);
        }
    }
}

```

第三方框架对拒绝策略的扩展

1. **Dubbo的实现，在抛出 RejectedExecutionException异常之前会记录日志，并dump线程栈信息，方便定位问题。**

2. **Netty 的实现，是创建一个新线程来执行任务**

2.源码解析
======

execute

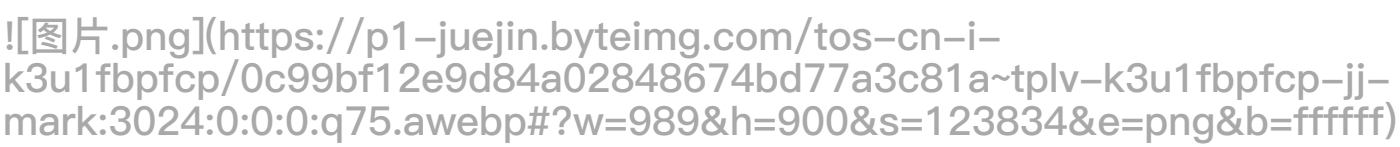
```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

    // 线程池状态
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        // 现在的线程总数 < 核心线程数 直接添加
        if (addWorker(command, true))
            return;
        // 添加失败，重新获得ctl
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        // 线程池是 Running 状态 && 任务添加到工作队列里面成功

        // 重新获得 ctl
        int recheck = ctl.get();

        // 状态不是 Running 那么从阻塞队列删除这个任务 执行拒绝策略 因为不管是 SHUTDOWN 还是 STOP 状态都不会接受新的任务
        if (! isRunning(recheck) && remove(command))
            reject(command);
        // 状态是Running 线程池里面没有线程 所以要创建一个线程去完成工作队列里面的任务
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // 任务添加到工作队列里面失败，添加非核心线程执行任务
    else if (!addWorker(command, false))
        reject(command);
}
```

简单来说，在执行execute()方法时如果状态一直是RUNNING时，的执行过程如下：

1. 如果`workerCount < corePoolSize`，则创建并启动一个线程来执行新提交的任务；

2. 如果`workerCount >= corePoolSize`，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中；
3. 如果`workerCount >= corePoolSize && workerCount < maximumPoolSize`，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务；
4. 如果`workerCount >= maximumPoolSize`，并且线程池内的阻塞队列已满，则根据拒绝策略来处理该任务，默认的处理方式是直接抛异常。

这里要注意一下`addWorker(null, false);`，也就是创建一个线程，但并没有传入任务，因为任务已经被添加到workQueue中了，所以worker在执行的时候，会直接从workQueue中获取任务。所以，在`workerCountOf(recheck) == 0`时执行`addWorker(null, false);`也是为了保证线程池在RUNNING状态下必须要有一个线程来执行任务。



addWorker

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:

    // 通过自旋 + CAS 的方式来创建线程，防止多创建线程
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);


        // 1. 线程池是 STOP 状态 直接返回
        // 2. 线程池是 SHUTDOWN状态 && firstTask != null，说明这是新提交的任务，SHUTDOWN状态下
        // 不会处理新提交的任务
        // 3. 线程池是 SHUTDOWN状态 && firstTask == null && 工作队列为空，说明已经没有任务需要处理了
        // 那么更加不需要创建线程
        if (rs >= SHUTDOWN &&! (rs == SHUTDOWN && firstTask == null && ! workQueue.isEmpty()))
            return false;

        for (;;) {
```

```
            int wc = workerCountOf(c);
            // 1. 线程池线程数直接超过了 2^29 次方，不能创建
            // 2. 线程池线程数超过了corePoolSize (core == true) ，说明添加的是核心线程，不能创建                    // 创建
            // 3. 线程池数量超过了 maximumPoolSize (core == false), 说明添加的是非核心线程，不能创建
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;

            // CAS 防止多创建线程
            if (compareAndIncrementWorkerCount(c))
                // 如果成功，直接跳出最外层循环，然后去创建线程
                break retry;

            // 走到这里 说明有并发创建线程
            c = ctl.get();
            // 检查一下线程池状态是否发生改变
            // 如果改变那么继续外层循环（外层循环会重新判断状态）
            // 如果没有发生改变，那么继续内层循环，内存循环继续尝试创建线程
            if (runStateOf(c) != rs)
                continue retry;

        }
    }

    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            final ReentrantLock mainLock = this.mainLock;
            // 创建线程的时候 使用 mainLock 加锁
            mainLock.lock();
            try {

                // 线程池状态
                int rs = runStateOf(ctl.get());

                // 1. 线程池是 RUNNING 状态
                // 2. 线程池是 SHUTDOWN状态，并且 firstTask 为 null -> 这个时候是需要创建一个线程去处理工作队列里面的任务
                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)
                        // 指标信息 统计一下线程池里面最大的线程数 可以用来做
指标分析
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            if (workerAdded) {
                // 开启线程
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

Worker类

线程池中的每一个线程被封装成一个Worker对象，ThreadPool维护的其实就是一组Worker对象，看一下Worker的定义：

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

    private static final long serialVersionUID = 6138294804551838833L;
    final Thread thread;
    Runnable firstTask;
    volatile long completedTasks;
```

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

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

    protected boolean isHeldExclusively() {
        return getState() != 0;
    }
    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }
    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }
    public void lock()        { acquire(1); }
    public boolean tryLock()  { return tryAcquire(1); }
    public void unlock()      { release(1); }
    public boolean isLocked() { return isHeldExclusively(); }
    void interruptIfStarted() {
        Thread t;
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
    }
}

```

Worker类继承了AQS，并实现了Runnable接口，注意其中的firstTask和thread属性：firstTask用它来保存传入的任务；thread是在调用构造方法时通过ThreadFactory来创建的线程，是用来处理任务的线程。

在调用构造方法时，需要把任务传入，这里通过

`getThreadFactory().newThread(this);`来新建一个线程，newThread方法传入的参数是this，因为Worker本身继承了Runnable接口，也就是一个线程，所以一个Worker对象在启动的时候会调用Worker类中的run方法。

Worker继承了AQS，使用AQS来实现独占锁的功能。为什么不使用ReentrantLock来实现呢？可以看到tryAcquire方法，它是 **不允许重入** 的，而 **ReentrantLock是允许重入**的：

1. lock方法一旦获取了 **独占锁**，表示当前线程 **正在执行任务中**；
2. 如果 **正在执行任务**，则 **不应该中断线程**；
3. 如果该线程现在不是独占锁的状态，也就是 **空闲** 的状态，说明它没有在处理任务，这时 **可以对该线程进行中断**；
4. 线程池在执行 **shutdown方法** **或 **tryTerminate方法** 时会调用 **interruptIdleWorkers方法** 来中断空闲的线程，**interruptIdleWorkers方法会使用tryLock方法来判断线程池中的线程是否是空闲状态**；
5. 之所以设置为不可重入，是因为我们不希望任务在调用像setCorePoolSize这样的线程池控制方法时重新获取锁。如果使用ReentrantLock，它是可重入的，这样如果在任务中调用了如setCorePoolSize这类线程池控制的方法，会中断正在运行的线程。

所以，Worker继承自AQS，用于判断线程是否空闲以及是否可以被中断。

此外，在构造方法中执行了`setState(-1);`，把state变量设置为-1，为什么这么做呢？是因为AQS中默认的state是0，如果刚创建了一个Worker对象，还没有执行任务时，这时就不应该被中断，看一下tryAquire方法：

```
protected boolean tryAcquire(int unused) {
    if (compareAndSetState(0, 1)) {
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    }
    return false;
}
```

tryAcquire方法是根据state是否是0来判断的，所以，`setState(-1);`将state设置为-1是为了禁止在执行任务前对线程进行中断。

正因为如此，在runWorker方法中会先调用Worker对象的unlock方法将state设

置为0.

runWorker

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    // 获取第一个任务
    Runnable task = w.firstTask;
    w.firstTask = null;
    // 允许中断 因为worker刚创建的时候 不允许被中断 所以这里需要解除限制
    w.unlock();
    // 是否因为异常退出循环
    boolean completedAbruptly = true;
    try {
        // 如果task为空，则通过getTask来获取任务
        while (task != null || (task = getTask()) != null) {
            w.lock();

            if ((runStateAtLeast(ctl.get(), STOP) || (Thread.interrupted() && runStateAtLeast(ctl.get(), STOP))) && !wt.isInterrupted())

                /*
                 * { [线程池是STOP状态]
                 *   ||
                 *   ([线程处于打断状态] && [线程池状态是STOP])
                 * }
                 * &&
                 * [线程没有被打断] (这里是因为前面的 Thread.interrupted() 清除了打断标记)
                 *
                 * 解释一下上面的if判断
                 * 如果线程池是STOP状态 那么该线程应该被打断
                 * 如果if执行的过程中 其他线程把线程池状态修改为了 STOP 并且当前线程修改为了打断                    * 状态
                 * 那么该线程也应该被打断。
                 * ps : Thread.interrupted() 方法会判断线程是否被打断，并且重置打断标记为false
                 */

                //打断线程
                wt.interrupt();
            try {
```

```
                //执行任务前置钩子
                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 {
                    //执行任务后置钩子
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        //如果执行任务的中出现了异常，那么不会走到这里
        completedAbruptly = false;
    } finally {
        //处理线程的回收工作
        processWorkerExit(w, completedAbruptly);
    }
}

```

总结一下runWorker方法的执行过程：

1. while循环不断地通过getTask()方法获取任务；
2. getTask()方法从阻塞队列中取任务；
3. 如果线程池正在停止，那么要保证当前线程是中断状态，否则要保证当前线程不是中断状态；
4. 调用`task.run()`执行任务；
5. 如果task为null则跳出循环，执行processWorkerExit()方法；
6. runWorker方法执行完毕，也代表着Worker中的run方法执行完毕，销毁线程。

这里的beforeExecute方法和afterExecute方法在ThreadPoolExecutor类中是空的，留给子类来实现。

completedAbruptly变量来表示在执行任务过程中是否出现了异常，在processWorkerExit方法中会对该变量的值进行判断。

getTask

```
private Runnable getTask() {
    // timeOut变量的值表示非核心线程上次从阻塞队列中取任务时是否超时
    boolean timedOut = false;
    for (; ; ) {
        int c = ctl.get();
        int rs = runStateOf(c);
        /*
         * 如果线程池状态rs >= SHUTDOWN，也就是非RUNNING状态，再进行以下判断：
         * 1. rs >= STOP，线程池是否正在stop；
         * 2. 阻塞队列是否为空。
         * 如果以上条件满足，则将workerCount减1并返回null。
         * 因为如果当前线程池状态的值是SHUTDOWN或以上时，不允许再向阻塞队列中添加任务。
         */
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);
        // timed变量用于判断是否需要进行超时控制。
        // 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；
        // 对于超过核心线程数量的这些线程，需要进行超时控制 ，这些就叫做非核心线程！
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        /*
         * wc > maximumPoolSize 线程数超过了 设置的最大线程数；
         * 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
```

```
         * 接下来判断，如果有效线程数量大于1，或者阻塞队列是空的，那么尝试将workerCount减1；
         * 如果减1失败，则返回重试。
         * 如果wc == 1时，也就说明当前线程是线程池中唯一的一个线程了。
         */
        if ((wc > maximumPoolSize || (timed && timedOut)) && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }
        try {
            /*
             * 根据timed来判断，如果为true，则通过阻塞队列的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，如果这时队列为空，则take方法会阻塞直到队列不为空。
             */
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
            workQueue.take();
            if (r != null)
                return r;
            // 如果 r == null，说明已经超时，timedOut设置为true
            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
            timedOut = false;
        }
    }
}
```

该方法会告诉我们 非工作线程是怎么淘汰的 ， 构造函数里面的 maximumPoolSize, keepAliveTime究竟是怎么使用的

processWorkerExit

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    // 如果completedAbruptly值为true，则说明线程执行时出现了异常，需要
```

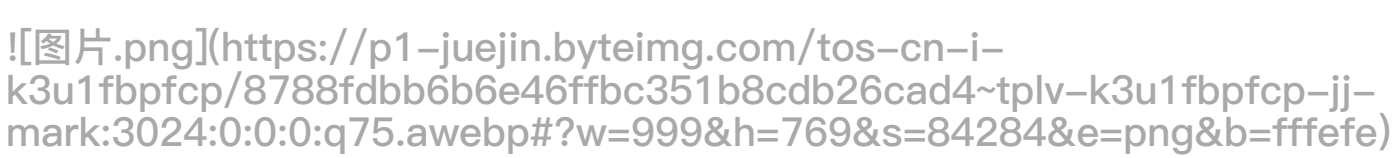
```
将workerCount减1；
    // 如果线程执行时没有出现异常，说明在getTask()方法中已经已经对
workerCount进行了减1操作，这里就不必再减了。
    if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
        decrementWorkerCount();

    final ReentrantLock mainLock = this.mainLock;
    // 上锁 因为workerSet不是线程安全的集合 对其操作需要加锁
    mainLock.lock();
    try {
        //统计完成的任务数
        completedTaskCount += w.completedTasks;
        // 从workers中移除，也就表示着从线程池中移除了一个工作线程
        workers.remove(w);
    } finally {
        mainLock.unlock();
    }
    // 根据线程池状态进行判断是否结束线程池
    tryTerminate();
    int c = ctl.get();

    /*
     * 当线程池是RUNNING或SHUTDOWN状态时，如果worker是异常结束
，那么会直接addWorker；
     *
     * 如果allowCoreThreadTimeOut=true，并且等待队列有任务，至少保留
一个worker；
     * 如果allowCoreThreadTimeOut=false，workerCount不少于
corePoolSize。
     */
    if (runStateLessThan(c, STOP)) {

        //这里是因为非核心线程超时了
        if (!completedAbruptly) {

            // 如果允许核心线程超时 那么最小worker数就是 0
            // 因为允许核心线程超时 就说明全都是非核心线程，
            // 那么就是来一个任务就创建一个线程来处理，没有核心线程这一说
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;

            // 最小线程数为0 ， 并且任务队列里面还有任务没有处理
            // 那么我们需要把 最小线程数设置为 1
            if (min == 0 && ! workQueue.isEmpty())
                min = 1;

            // 判断现在的worker数是否已经 min，
            // 如果已经超过了 那么就不需要创建新的worker了
            if (workerCountOf(c) >= min)
```

```
                return; // replacement not needed
        }

        //该 worker是因为执行任务的过程中出现了异常，所以直接添加一个worker补充上去
        addWorker(null, false);
    }
}

```

该方法会做回收线程的工作，他回收的线程可能是执行任务过程中出现异常的线程，也可能是没有任务可以获取的非工作线程。

这里我们也看到，对于出现异常的线程，他是直接销毁掉，然后再创建一个新的线程。所以我们建议最好自己去处理线程里面的异常，避免重新创建线程浪费系统资源

至此，processWorkerExit执行完之后，工作线程被销毁，以上就是整个工作线程的生命周期，从execute方法开始，Worker使用ThreadFactory创建新的工作线程，runWorker通过getTask获取任务，然后执行任务，如果getTask返回null，进入processWorkerExit方法，整个线程结束，如图所示：



tryTerminate

```
final void tryTerminate() {
    for (; ; ) {
        int c = ctl.get();
        /*
         * 当前线程池的状态为以下几种情况时，直接返回：
         * 1. RUNNING，因为还在运行中，不能停止；
         * 2. TIDYING或TERMINATED，这个时候线程池已经结束了 不需要重复结束了
         * 3. SHUTDOWN并且等待队列非空，这时要执行完workQueue中的task；
```

```
         */

        if (isRunning(c) ||
            runStateAtLeast(c, TIDYING) ||
            (runStateOf(c) == SHUTDOWN && !workQueue.isEmpty()))
            return;

        // 如果线程数量不为0，则中断一个空闲的工作线程，并返回
        if (workerCountOf(c) != 0) {
            interruptIdleWorkers(ONLY_ONE);
            return;
        }

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 这里尝试设置状态为TIDYING，如果设置成功，则调用
terminated方法
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {

                    // terminated方法默认什么都不做，留给子类实现
                    terminated();

                } finally {
                    // 设置状态为TERMINATED
                    ctl.set(ctlOf(TERMINATED, 0));
                    termination.signalAll();
                }
                return;
            }
        } finally {
            mainLock.unlock();
        }
        // else retry on failed CAS
    }
}

```

`interruptIdleWorkers(ONLY_ONE);` 的作用是因为在 **getTask** 方法中执行`workQueue.take()`*时，如果不执行中断会一直阻塞。在下面介绍的 **shutdown** 方法中，会中断所有空闲的工作线程，如果在执行shutdown时工作线程没有空闲，然后又去调用了getTask方法，这时如果workQueue中没有任务了，调用`workQueue.take()` 时就会一直阻塞。所以每次在工作线程结束时调用 **tryTerminate** 方法来尝试中断一个空闲工作线程，避免在队列为空时取任务一直阻塞的情况。

会调用 tryTerminate() 方法的方法汇总：

1. addWorkerFailed（）添加worker失败
2. processWorkerExit ( ) 回收worker
3. shutdown ()
4. shutdownNow ()
5. remove（）将任务从任务队列里面移除
6. purge () 官方注释 ： Tries to remove from the work queue all Future tasks that have been cancelled，就是去 尝试移除 future 类型 并且 已经被取消的任务。

简而言之 就是当 worker 或者 任务的数量 减少的时候 ， 都会尝试去 调用 tryTerminate() 方法 把线程池终结。

shutDown

```
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 安全策略判断(不重要)
        checkShutdownAccess();
        // 切换状态为SHUTDOWN
        advanceRunState(SHUTDOWN);
        // 中断空闲线程
        interruptIdleWorkers();
        onShutdown(); // hook for ScheduledThreadPoolExecutor
    } finally {
        mainLock.unlock();
    }
    // 尝试结束线程池
    tryTerminate();
}
```

这里思考一个问题：在 **runWorker方法** 中，**执行任务时对Worker对象w进行了lock操作**，为什么要在执行任务的时候对每个工作线程都加锁呢?

下面仔细分析一下：

* 在getTask方法中，如果这时线程池的状态是SHUTDOWN并且workQueue为空，那么就应该返回null来结束这个工作线程，而使线程池进入SHUTDOWN状态需要调用shutdown方法；
* shutdown方法会调用interruptIdleWorkers来中断 **空闲** 的线程，**interruptIdleWorkers持有mainLock**，会遍历**workers**来逐个判断工作线程是否空闲。但getTask方法中没有mainLock；
* 在getTask中，如果判断当前线程池状态是 **RUNNING**，**并且阻塞队列为空**，那么会调用`workQueue.take()` 进行阻塞；
* 如果在判断当前线程池状态是 **RUNNING** 后，这时调用了 **shutdown** 方法把状态改为了 **SHUTDOWN**，这时如果不进行中断，那么当前的工作线程在调用了 **`workQueue.take()`** 后会一直阻塞而不会被销毁，因为在**SHUTDOWN状态下不允许再有新的任务添加到workQueue中**，**这样一来线程池永远都关闭不了了**；
* 由上可知，**shutdown 方法**与 **getTask 方法**（**从队列中获取任务时**）存在 **竞态条件**；
* 解决这一问题就需要用到**线程的中断**，也就是为什么要用**interruptIdleWorkers方法**。在调用`workQueue.take()` 时，**如果发现当前线程在执行之前或者执行期间是中断状态，则会抛出InterruptedException，解除阻塞的状态**；
* 但是要中断工作线程，**还要判断工作线程是否是空闲的**，**如果工作线程正在处理任务，就不应该发生中断**；
* 所以Worker继承自AQS，在工作线程处理任务时会进行lock，interruptIdleWorkers在进行中断时会使用tryLock来判断该工作线程是否正在处理任务，**如果tryLock返回true，说明该工作线程当前未执行任务，这时才可以被中断。也就是说，通过继承AQS，**

然后就相当于给**worker添加了一个状态信息**，**如果没上锁，那么处于空闲状态**，**是可以被中断的**。**如果上锁了，那么 处于 正在执行任务的状态 ， 此时不能中断该线程**

interruptIdleWorkers

```
private void interruptIdleWorkers() {
    interruptIdleWorkers(false);
}
private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
```

```
            for (Worker w : workers) {
                Thread t = w.thread;
                if (!t.isInterrupted() && w.tryLock()) {
                    try {
                        t.interrupt();
                    } catch (SecurityException ignore) {
                    } finally {
                        w.unlock();
                    }
                }
                if (onlyOne)
                    break;
            }
        } finally {
            mainLock.unlock();
        }
    }
```

interruptIdleWorkers遍历workers中所有的工作线程，若**线程没有被中断**并且**tryLock成功**，就中断该线程。

为什么要使用 mainLock加锁？**假设添加线程和 interruptIdleWorkers 同时执行，那么就可能会存在 有些线程永远无法中断。**

shutdownNow

```
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();
        advanceRunState(STOP);
        // 中断所有工作线程，无论是否空闲
        interruptWorkers();
        // 取出队列中没有被执行的任务
        tasks = drainQueue();
    } finally {
        mainLock.unlock();
    }
```

```
    tryTerminate();
    return tasks;
}
```

shutdownNow方法与shutdown方法类似，不同的地方在于：

1. 设置状态为STOP；
2. 中断所有工作线程，无论是否是空闲的；
3. 取出阻塞队列中没有被执行的任务并返回。

shutdownNow方法执行完之后调用tryTerminate方法，该方法在上文已经分析过了，目的就是使线程池的状态设置为TERMINATED。

interruptWorkers

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

我们可以看到该方法会打断所有的线程。不管线程是否在执行任务还是空闲

3. 个人对线程池设计思想的一些感悟
==========

原文链接: https://juejin.cn/post/7385752495534817295