线程池之ThreadPoolExecutor执行原理

分析ThreadPoolExecutor的执行原理，直接从execute方法开始

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

int c = ctl.get();

// 1、工作线程 < 核心线程

if (workerCountOf(c) < corePoolSize) {

if (addWorker(command, true))

return;

c = ctl.get();

}

// 2、运行态，并尝试将任务加入队列

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);

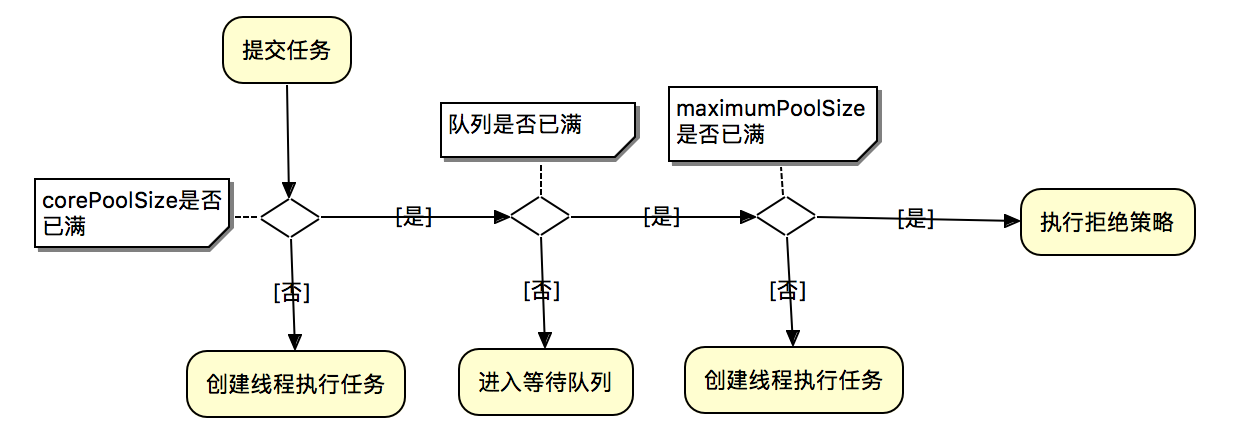
} // 3、使用尝试使用最大线程运行

else if (!addWorker(command, false))

reject(command);

}

这三处if判断，还是比较泛的，整体大框框上的流程，可用下图表示：



在execute方法中，用到了double-check的思想，我们看到上述代码中并没有同步控制，都是基于乐观的check，如果任务可以创建则进入addWorker（Runnable firstTask, boolean core）方法，注意上述代码中的三种传参方式：

 addWorker(command, true)： 创建核心线程执行任务；

 addWorker(command, false)：创建非核心线程执行任务；

 addWorker(null, false)： 创建非核心线程，当前任务为空；

addWorker的返回值是boolean，不保证操作成功。下面详看addWorker方法（代码稍微有点长）：

private boolean addWorker(Runnable firstTask, boolean core) {

// 第一部分：自旋、CAS、重读ctl 等结合，直到确定是否可以创建worker，可以则跳出循环继续操作，否则返回false

retry:

for (;;) {

int c = ctl.get(),rs = runStateOf(c);

// Check if queue empty only if necessary.

if (rs >= SHUTDOWN &&! (rs == SHUTDOWN &&firstTask == null &&! workQueue.isEmpty()))

return false;

for (;;) {

int wc = workerCountOf(c);

if (wc >= CAPACITY ||wc >= (core ? corePoolSize : maximumPoolSize)) return false;

if (compareAndIncrementWorkerCount(c)) break retry; // CAS增长workerCount，成功则跳出循环

c = ctl.get(); // Re-read ctl 重新获取ctl

if (runStateOf(c) != rs) continue retry; // 状态改变则继续外层循环，否则在内层循环

// else CAS failed due to workerCount change; retry inner loop

}

}

// 第二部分：创建worker，这部分使用ReentrantLock锁

boolean workerStarted = false, workerAdded = false; // 线程启动标志位、线程是否加入workers 标志位

Worker w = null;

try {

w = new Worker(firstTask); //创建worker

final Thread t = w.thread;

if (t != null) {

final ReentrantLock mainLock = this.mainLock;

mainLock.lock();

try {

// 获取到锁以后仍需检查ctl，可能在上一个获取到锁处理的线程可能会改变runState

// 如 ThreadFactory 创建失败 或线程池被 shut down等

int rs = runStateOf(ctl.get());

if (rs < SHUTDOWN || (rs == SHUTDOWN && firstTask == null)) {

if (t.isAlive()) throw new IllegalThreadStateException();

workers.add(w);

int s = workers.size();

if (s > largestPoolSize) largestPoolSize = s;

workerAdded = true;

}

} finally {mainLock.unlock();}

if (workerAdded) {

t.start(); // 启动线程

workerStarted = true;

}

}

} finally {if (! workerStarted) addWorkerFailed(w); // 失败操作}

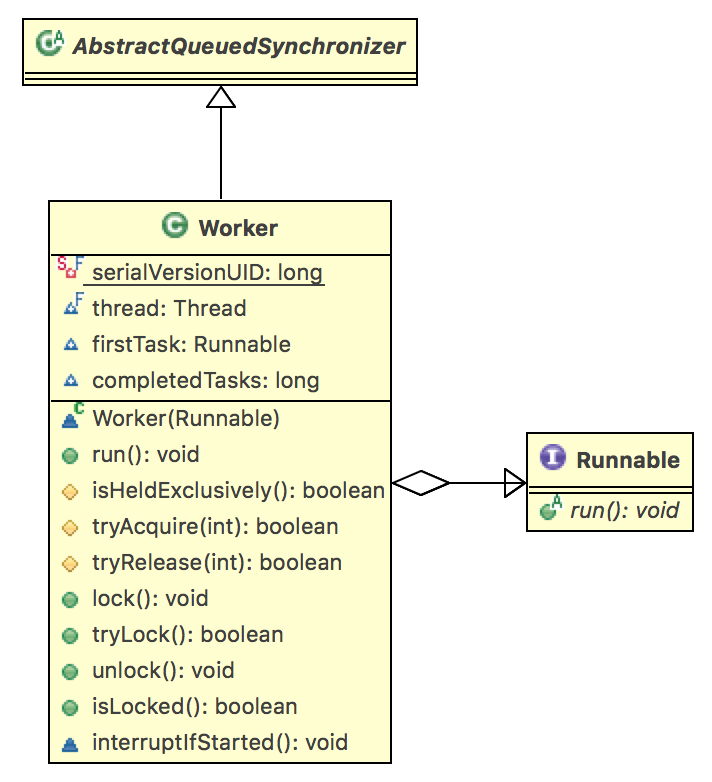
return workerStarted;

}

addWorker的工作可分为两个部分：

* 第一部分：原子操作，判断是否可以创建worker。通过自旋、CAS、ctl等操作，判断继续创建还是返回false，自旋周期一般很短。
* 第二部分：同步创建worker，并启动线程。

第一部分只要思路清楚就可以理解了。下面详解第二部分的worker:



woker是ThreadPoolExecutor的内部类，实现了AbstractQueuedSynchronizer并集成了Runnable。

private final class Worker extends AbstractQueuedSynchronizer implements Runnable{

private static final long serialVersionUID = 6138294804551838833L;

final Thread thread; //每个worker有自己的内部线程，ThreadFactory创建失败时是null

Runnable firstTask; //初始化任务，可能是null

volatile long completedTasks; //每个worker的完成任务数

Worker(Runnable firstTask) {

setState(-1); // 禁止线程在启动前被打断

this.firstTask = firstTask;

this.thread = getThreadFactory().newThread(this);

}

/\*\* 重要的执行方法 \*/

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

protected boolean isHeldExclusively() {

return getState() != 0; // state = 0 代表未锁；state = 1 代表已锁

}

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(); }

// interrupt已启动线程

void interruptIfStarted() {

Thread t;

// 初始化是 state = -1，不会被interrupt

if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {

try {

t.interrupt();

} catch (SecurityException ignore) {

}

}

}

}

worker实现了简单的非重入互斥锁。互斥容易理解，非重入是为了避免线程池的一些控制方法获得重入锁，比如setCorePoolSize操作。注意worker实现锁的目的与传统锁的意义不太一样。其主要是为了控制线程是否可interrupt，以及其它的监控，如线程是否active（正在执行任务）。

线程池里线程是否处于运行状态与普通线程不一样，普通线程可以调用 Thread.currentThread().isAlive() 方法来判断，而线程池，在run方法中可能在等待获取新任务，这期间线程线程是 alive 但是却不是 active。

runWorker代码如下：

final void runWorker(Worker w) {

Thread wt = Thread.currentThread();

Runnable task = w.firstTask;

w.firstTask = null;

w.unlock(); // 允许被 interrupt

boolean completedAbruptly = true;

try {

// loop 直至 task = null （线程池关闭、超时等），注意这里的getTask()方法，我们配置的阻塞队列会在这里起作用

while (task != null || (task = getTask()) != null) {

w.lock(); // 执行任务前上锁

// 如果线程池停止，确保线程中断; 如果没有，确保线程不中断。这需要在第二种情况下进行重新获取ctl，以便在清除中断时处理shutdownNow竞争

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

wt.interrupt();

try {

beforeExecute(wt, task); // 扩展点

Throwable thrown = null;

try {

task.run(); // 真正执行run方法

} catch (RuntimeException x) { thrown = x; throw x;

} catch (Error x) { thrown = x; throw x;

} catch (Throwable x) { thrown = x; throw new Error(x);

} finally { afterExecute(task, thrown); // 扩展点}

} finally {

task = null;

w.completedTasks++;

w.unlock();

}

}

completedAbruptly = false;

} finally { processWorkerExit(w, completedAbruptly); // 线程退出工作 }

}

runWorker的主要任务就是一直loop循环，来一个任务处理一个任务，没有任务就去getTask()，getTask()可能会阻塞，代码如下：

private Runnable getTask() {

boolean timedOut = false; // 上一次 poll() 是否超时

for (;;) {

int c = ctl.get();

int rs = runStateOf(c);

// 是否继续处理任务 可以参见上一篇的状态控制

if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {

decrementWorkerCount();

return null;

}

int wc = workerCountOf(c);

// 是否允许超时

boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

if ((wc > maximumPoolSize || (timed && timedOut))

&& (wc > 1 || workQueue.isEmpty())) {

if (compareAndDecrementWorkerCount(c)) return null;

continue;

}

try {

Runnable r = timed ?

workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :

workQueue.take();

if (r != null) return r;

timedOut = true;

} catch (InterruptedException retry) {

timedOut = false;

}

}

}

getTask()方法里面主要用我们配置的workQueue来工作，其阻塞原理与超时原理基于阻塞队列实现，这里不做详解。

总结，ThreadPoolExecutor的执行主要围绕worker，worker实现了AbstractQueuedSynchronizer并继承了Runnable，其对锁的巧妙运用，值得思考。