这边文章主要介绍线程池

相关包与类

包：

java.util.concurrent

类：

Executor -> ExecutorService -> ScheduledExecutorService / AbstractExecutorService

和AbstractExecutorService相关的线程池：

ThreadPoolExecutor -> [ScheduledThreadPoolExecutor]]

ForkJoinPool相关：

ForkJoinWorkerThread

ForkJoinPool

ForkJoinTask->RecursiveAction / RecursiveTask

线程结合Queue：

CompletableFuture

CompletionService

ExecutorCompletionService

和ScheduledExecutorService相关的线程池

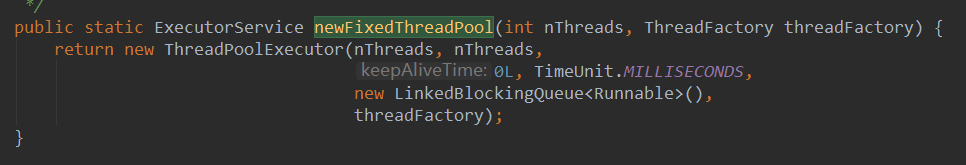
Executors#DelegatedExecutorService

线程池的使用和创建

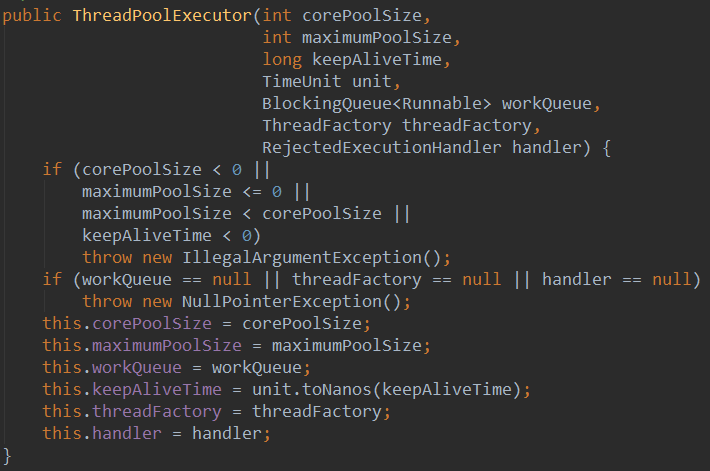
创建固定线程个数的线程池：

ThreadPoolExecutor：Executors.newFixedThreadPool

使用构造函数创建ThreadPoolExecutor：



下面开始分析ThreadPoolExecutor构造函数：



1. 入参检查

2. 各参数意义

|  |  |
| --- | --- |
| 变量名 | 变量解释 |
| corePoolSize | 活跃的线程个数（用于优先创建的线程） |
| maximumPoolSize | 最大支持的线程个数（如果corePoolSize不足，尝试创建一个Worker线程，但是这个线程有等待时间keepAliveTime，如果没有等待时间，Queue将一直等待获取runnable，那么这个Worker也不会结束） |
| workQueue | 工作队列，任务阻塞队列 |
| keepAliveTime | 后续等待线程的超时时间 |
| threadFactory | 创建线程的工厂，默认Executors#DefaultThreadFactory |
| handler | RejectedExecutionHandler拒接任务的处理器[如果任务超时] |

根据入参我们大概可以猜到它实现方式，就是将runnable任务加入workQueue中，然后开启固定线程数量去take阻塞获取，添加过程不是阻塞的，但是没有对应线程去运行。至于handler应该就是拒接任务的异常处理，表示线程池拒接任务的回调

下面我们开始追踪源码开始运行任务。

ThreadPoolExecutor线程池源码解析

#execute

提交任务到线程池，下面开始分析线程池如何分配线程运行

public void execute(Runnable command) {

// 1 传进来的线程为null，则抛出空指针异常  
 if (command == null)  
 throw new NullPointerException();

// 2 获取当前线程池的状态+线程个数变量  
 int c = ctl.get();

// 3 判断当前线程池线程个数是否小于corePoolSize,小于则调用addWorker方法创建新线程运行,且传进来的Runnable当做第一个任务执行

//如果调用addWorker方法返回false，则直接返回  
 if (*workerCountOf*(c) < corePoolSize) {  
 if (addWorker(command, true))  
 return;  
 c = ctl.get();  
 }

// 4如果线程池处于RUNNING状态，则添加任务到阻塞队列  
 if (*isRunning*(c) && workQueue.offer(command)) {  
 int recheck = ctl.get();

// 5如果当前线程池状态不是RUNNING则从队列删除任务，并执行拒绝策略  
 if (! *isRunning*(recheck) && remove(command)) //否者如果当前线程池线程空，则添加一个线程  
 reject(command);  
 else if (*workerCountOf*(recheck) == 0)  
 addWorker(null, false);  
 }

// 6新增线程，新增失败则执行拒绝策略  
 else if (!addWorker(command, false))  
 reject(command);  
}

1. 如果runnable 是null 抛出异常

2. ctl.get() ： 获取线程池状态，使用位来标记，这里有点类似于 Android里面的MeasureSpec#

void onMeasure(int widthMeasureSpec, int heightMeasureSpec)

解析如下：

**//用来标记线程池状态（高3位），线程个数（低29位）**

**//默认是RUNNING状态，线程个数为0**

private final AtomicInteger ctl = new AtomicInteger(*ctlOf*(*RUNNING*, 0));

**//线程个数掩码位数**

**private static final int COUNT\_BITS = Integer.SIZE - 3;**

**//线程最大个数(低29位)00011111111111111111111111111111**

**private static final int CAPACITY = (1 << COUNT\_BITS) - 1;**

**//（高3位）：11100000000000000000000000000000**

**private static final int RUNNING = -1 << COUNT\_BITS;**

**//（高3位）：00000000000000000000000000000000**

**private static final int SHUTDOWN = 0 << COUNT\_BITS;**

**//（高3位）：00100000000000000000000000000000**

**private static final int STOP = 1 << COUNT\_BITS;**

**//（高3位）：01000000000000000000000000000000**

**private static final int TIDYING = 2 << COUNT\_BITS;**

**//（高3位）：01100000000000000000000000000000**

**private static final int TERMINATED = 3 << COUNT\_BITS;**

**// 获取高三位 运行状态**

**private static int runStateOf(int c) { return c & ~CAPACITY; }**

**//获取低29位 线程个数**

**private static int workerCountOf(int c) { return c & CAPACITY; }**

**//计算ctl新值，线程状态 与 线程个数**

**private static int ctlOf(int rs, int wc) { return rs | wc; }**

简单来说，就是Integer类的

前3位 作为线程池状态， 后3位最为线程池里面线程个数

比如数字0：计算机里面表示位：

000 00000000000000000000000000000

那么取前面3位 红色作为计算状态，000~ 111（二进制表示）

后面N位 绿色作为线程个数 0 ~ 11111111111111111111111111（二进制表示）

3.

if (*workerCountOf*(c) < corePoolSize) {  
 if (addWorker(command, true))  
 return;  
 c = ctl.get();  
}

如果线程个数，小于corePoolSize，构造函数里面的线程个数

那么执行addWorker(command, true)方法去创建线程，并且将传入的runnable作为线程的第一个runnable。如果执行成功那么成功返回，否则重新刷新2中状态值

下面开始分析addWroker的方法

private boolean addWorker(Runnable firstTask, boolean core) {

// 第一部分分析  
 retry:  
 for (;;) {

// 3.0  
 int c = ctl.get();  
 int rs = *runStateOf*(c);  
 // 3.1   
 // Check if queue empty only if necessary.  
 if (rs >= *SHUTDOWN* &&  
 ! (rs == *SHUTDOWN* &&  
 firstTask == null &&  
 ! workQueue.isEmpty()))  
 return false;  
 // 3.2  
 for (;;) {  
 int wc = *workerCountOf*(c);  
 if (wc >= *CAPACITY* ||  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 return false;  
 if (compareAndIncrementWorkerCount(c))  
 break retry;  
 c = ctl.get(); // Re-read ctl  
 if (*runStateOf*(c) != rs)  
 continue retry;  
 // else CAS failed due to workerCount change; retry inner loop  
 }  
}

第二部分。。。  
}

第一部分分析：

3.0 自旋锁（线程安全），获取ctl的int值，并获取其高三位的状态

3.1 检查当前线程池状态是否是SHUTDOWN、STOP、TIDYING或者TERMINATED 。当前状态为SHUTDOWN、且传入的任务为null，且队列不为null， 条件都成立则返回false

3.2 自旋锁

如果当前线程数量大于最大容量，或者大于（是否是核心线程）线程数量[核心线程数/最大线程数]，返回false

增加C的数值，CAS操作，线程之前的C数值比较并自增，如果操作成功，那么退出该循环，否则重新获取C的值

重新获取C的值，判断此时ctl的状态，如果与开始3.1状态不一致，那么就继续自旋操作

第一部分自旋锁主要是保证线程安全的操作ctl的数字，将其线程数+1，并且在线程池活跃状态下才确认操作成功

第二部分分析：

在第一部分线程数量+1之后，开始第二部分代码

boolean workerStarted = false;  
boolean workerAdded = false;  
Worker w = null;  
try {

// 3.1 生成Worker包装类，firstTask是传入的runnable任务  
 w = new Worker(firstTask);  
 final Thread t = w.thread;  
 if (t != null) {  
 final ReentrantLock mainLock = this.mainLock;

// 3.2 开始锁，保证操作线程的安全性  
 mainLock.lock();  
 try {  
 // Recheck while holding lock.  
 // Back out on ThreadFactory failure or if  
 // shut down before lock acquired.  
 int rs = *runStateOf*(ctl.get());  
 // 3.3 判断如果线程池当前处于活跃状态  
 if (rs < *SHUTDOWN* ||  
 (rs == *SHUTDOWN* && firstTask == null)) {  
 if (t.isAlive()) // 如果新线程已经活跃状态，那么不会再次start，抛出异常  
 throw new IllegalThreadStateException();

// 3.4 向workers[map]中添加包装的runnable类  
 workers.add(w);  
 int s = workers.size();

// 3.5 这里记录workers曾经达到过的最大长度  
 if (s > largestPoolSize)  
 largestPoolSize = s;

// 3.6 任务添加成功  
 workerAdded = true;  
 }  
 } finally {  
 mainLock.unlock();  
 }

// 3.7 如果已经添加任务成功，t代表新建的线程，那么启动该线程，workerStarted标志true  
 if (workerAdded) {  
 t.start();  
 workerStarted = true;  
 }  
 }  
} finally {  
 if (! workerStarted)

// 3.8 如果任务启动失败， 调用该failed方法，减少线程数，移除该wroker任务，判断是否需要尝试关闭线程池  
 addWorkerFailed(w);  
}  
return workerStarted;

第二部分分析如下：

这部分主要就是向workers[Set]中添加Worker并开启Thread线程，这里线程数量在第一部分判断了，包装线程个数不会超过core ? corePoolSize : maximumPoolSize 个数

添加完成，将开启线程，这里Thread由ThreadFactory构造函数里面的创建，并且Thread将一直Run[下面将分析Worker线程]

private final class Worker  
 extends AbstractQueuedSynchronizer  
 implements Runnable

Worker是AQS的实现类，我们看下其实现方式：

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;  
}

我们可以看出，该类的锁只能保证一个锁，也就是同时只允许一个线程获取该Worker，其他线程必须等待，且是独占锁

前面ThreadFactory创建的Thread。启动之后将启动该Worker的run方法：

*/\*\* Delegates main run loop to outer runWorker. \*/*public void run() {  
 runWorker(this);  
}

下面我们分析其run方法：

final void runWorker(Worker w) {  
 Thread wt = Thread.*currentThread*();

// p1 worker任务的第一个runnable,构造函数加入  
 Runnable task = w.firstTask;  
 w.firstTask = null;

// p2 解锁一次，用于重置状态  
 w.unlock(); // allow interrupts  
 boolean completedAbruptly = true;  
 try {

// p3 重要的一点，task第一个是worker构造传入，后面使用getTask获取  
 while (task != null || (task = getTask()) != null) {

// p4 当前线程锁，这里worker只会存在一个线程中，这里用于判断线程是否处于活跃状态可以这样做，使用AQS独占锁判断是否独占  
 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  
 if ((*runStateAtLeast*(ctl.get(), *STOP*) ||  
 (Thread.*interrupted*() &&  
 *runStateAtLeast*(ctl.get(), *STOP*))) &&  
 !wt.isInterrupted())  
 wt.interrupt();  
 try {  
 beforeExecute(wt, task);  
 Throwable thrown = null;  
 try {

// p5 真正传入的runnable执行  
 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 {  
 afterExecute(task, thrown);  
 }  
 } finally {

// p6 一些其他记录增加，worker解锁  
 task = null;  
 w.completedTasks++;  
 w.unlock();  
 }  
 }  
 completedAbruptly = false;  
 } finally {

// p7 最后删除set中存储的worker，worker退出，线程执行结果  
 processWorkerExit(w, completedAbruptly);  
 }  
}

上面代码看上去比较多，其实我们挑重点：

1. worker#run 使用while循环首先运行构造函数传入的runnable，然后后续从阻塞队列LinkedBlockingQueue里面阻塞获取runnable，等待获取成功执行runnable

2.while一直循环等待，这里有点类似于Handler的实现原理，使用MessageQueue来获取与休眠

3.Queue在没有消息的时候会阻塞run的线程，添加数据的时候主要在下面4中的4.0

4.新技能：使用AQS独占锁可以在while里面判断是否线程处于独占状态

4. 判断线程池是否是运行状态并且尝试offer添加runnable，如果成功

// 4.0 如果线程池处于RUNNING状态，则添加任务到阻塞队列

if (*isRunning*(c) && workQueue.offer(command)) {

//二次检查  
 int recheck = ctl.get();

// 4.1如果当前线程池状态不是RUNNING则从队列删除任务，并执行拒绝策略  
 if (! *isRunning*(recheck) && remove(command))  
 reject(command); // 4.2否者如果当前线程池线程空，则添加一个线程  
 else if (*workerCountOf*(recheck) == 0)  
 addWorker(null, false);  
 }   
else if (!addWorker(command, false)) // 添加runnable到线程任务中，如果失败，执行拒绝策略  
 reject(command);

4.0 这里判断线程池的运行状态，并且调用重要代码：workQueue.offer(command)，向队列里面添加任务，这里就起到了之前coreThread#run阻塞获取任务，后面我们分析Worker#run方法

4.1 再次刷新ctl值，判断是否处于运行状态，如果此时多线程关闭线程池，那么拒绝运行该任务，否则如果当前线程池线程空，则添加一个线程

4.2 这里添加一个空任务来执行[目前还不确定是否一定需要]

关于关闭线程池：

public void shutdown() {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();  
 advanceRunState(*SHUTDOWN*);  
 interruptIdleWorkers();  
 onShutdown(); // hook for ScheduledThreadPoolExecutor  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();  
}

interruptIdleWorkers：

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

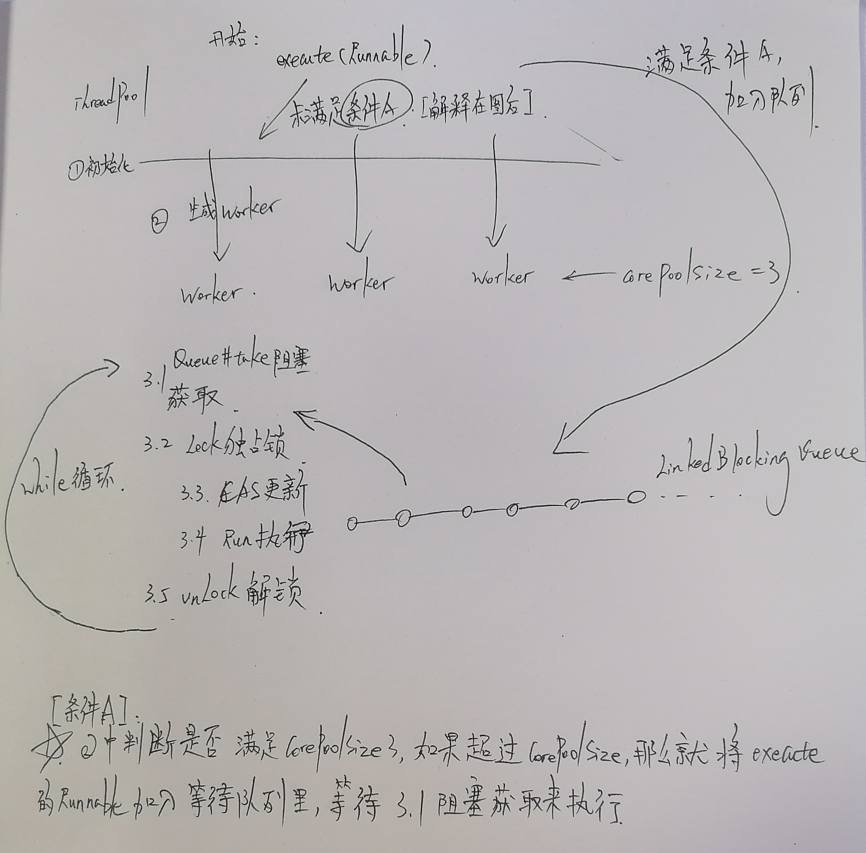
}

主要代码使用打断线程

1. worker尝试独占锁，执行线程打断，那么Worker#Runnable线程执行Finally

2.如果work获取锁失败，并且没有被打断，那么执行advanceRunState(SHUTDOWN)将状态至为shutDown，[Worker里面退出](#shutDown标志), Worker#Runnable线程执行Finally

下面是手画的流程图：



ScheduledThreadPoolExecutor 支持周期调度任务的线程池

ScheduledThreadPoolExecutor是在ThreadPoolExector线程池之上的扩展，简单来讲就是任务支持周期性调度了

扩展：

ScheduledThreadPoolExecutor 使用DelayedWorkQueue任务队列作为任务数据集合

static class DelayedWorkQueue extends AbstractQueue<Runnable>  
 implements BlockingQueue<Runnable>

在添加任务的时候通过：

ScheduledFutureTask包装Runnable，如下：

public <V> ScheduledFuture<V> schedule(Callable<V> callable,  
 long delay,  
 TimeUnit unit) {  
 if (callable == null || unit == null)  
 throw new NullPointerException();  
 RunnableScheduledFuture<V> t = decorateTask(callable,  
 new ScheduledFutureTask<V>(callable,  
 triggerTime(delay, unit),  
 *sequencer*.getAndIncrement()));  
 delayedExecute(t);  
 return t;  
}

看下ScheduledFutureTask本身的构造函数：

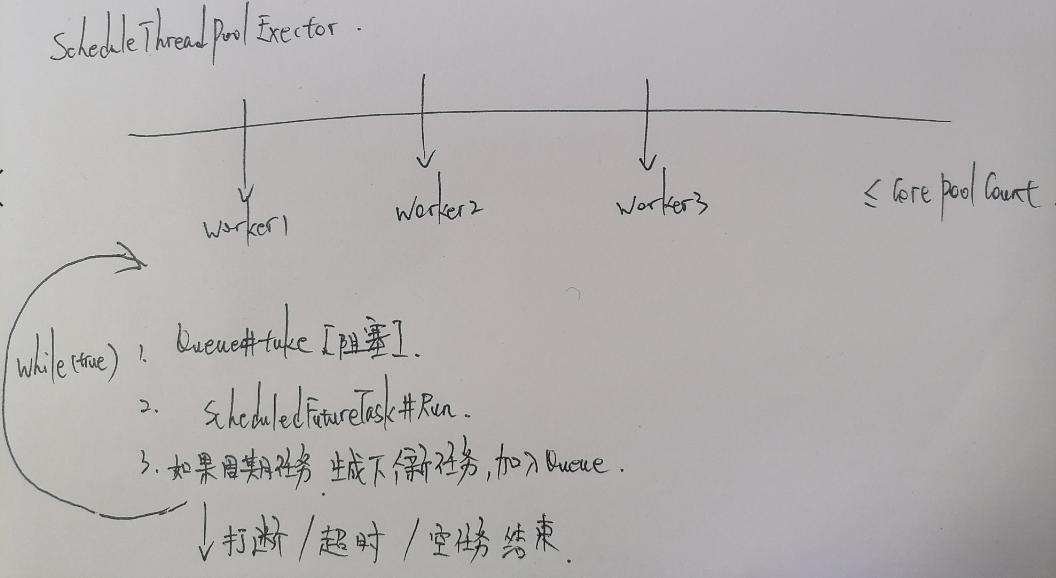
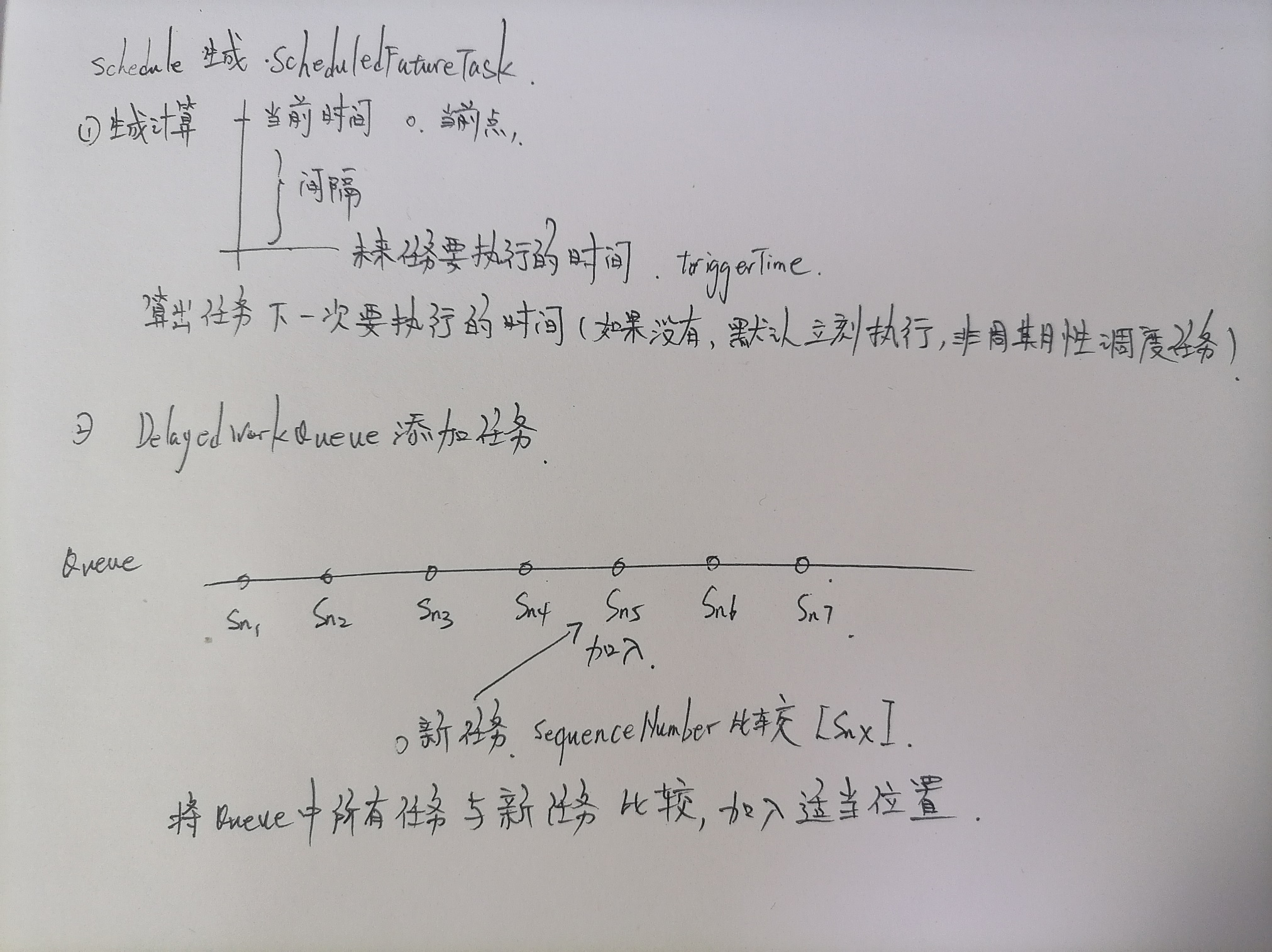
*/\*\*  
 \* Creates a periodic action with given nanoTime-based initial  
 \* trigger time and period.  
 \*/*ScheduledFutureTask(Runnable r, V result, long triggerTime,  
 long period, long sequenceNumber) {  
 super(r, result);  
 this.time = triggerTime;  
 this.period = period;  
 this.sequenceNumber = sequenceNumber;  
}

|  |  |
| --- | --- |
| 字段 | 说明 |
| time | 需要触发的时间 |
| period | 任务需要重复的次数 |
| sequenceNumber | 序列号，理解为当前任务的序号，用于比较加入队列排序，小的序号排前，这里理解就是，当前队列是一个有序队列，根据sequenceNumber排序的 |

比较方法：

public int compareTo(Delayed other) {  
 if (other == this) // compare zero if same object  
 return 0;  
 if (other instanceof ScheduledFutureTask) {  
 ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;  
 long diff = time - x.time;  
 if (diff < 0)  
 return -1;  
 else if (diff > 0)  
 return 1;  
 else if (sequenceNumber < x.sequenceNumber)  
 return -1;  
 else  
 return 1;  
 }  
 long diff = getDelay(*NANOSECONDS*) - other.getDelay(*NANOSECONDS*);  
 return (diff < 0) ? -1 : (diff > 0) ? 1 : 0;  
}

流程图：



public void run() {  
 boolean periodic = isPeriodic();

// 如果任务无法执行，那么执行cancel  
 if (!canRunInCurrentRunState(periodic))  
 cancel(false);  
 else if (!periodic) // 非周期性任务，那么执行run一次  
 super.run();  
 else if (super.runAndReset()) { // 执行run并且重置  
 setNextRunTime(); //更新下一次执行时间  
 reExecutePeriodic(outerTask); //重新加入Queue队列  
 }  
}

生成下一个任务：

void reExecutePeriodic(RunnableScheduledFuture<?> task) {  
 if (canRunInCurrentRunState(true)) {  
 super.getQueue().add(task);  
 if (!canRunInCurrentRunState(true) && remove(task))  
 task.cancel(false);  
 else  
 ensurePrestart();  
 }  
}

下面看下DelayedWorkQueue如何实现阻塞：

public RunnableScheduledFuture<?> take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 for (;;) {

// 获取第一个任务  
 RunnableScheduledFuture<?> first = queue[0];

// 如果任务是空，那么condition等待，直到有任务加入，重新自旋  
 if (first == null)  
 available.await();  
 else {  
 long delay = first.getDelay(*NANOSECONDS*); //获取延迟时间，是否当前时间已经超过未来执行的时间  
 if (delay <= 0L) // 需要返回任务执行  
 return finishPoll(first);  
 first = null; // don't retain ref while waiting  
 if (leader != null) //leader是头线程，如果是其他线程，那么等待（1）  
 available.await();  
 else {

// 当前线程作为leader  
 Thread thisThread = Thread.*currentThread*();  
 leader = thisThread;  
 try {

// 等待超时时间，此时系统时间等待完成就会自旋返回  
 available.awaitNanos(delay);  
 } finally {

// 确认leader是同一个线程，方式外部修改指向，否则其他线程进入（1）判断等待  
 if (leader == thisThread)  
 leader = null;  
 }  
 }  
 }  
 }  
 } finally {

// 这里finally用于通知其他（1）的等待线程  
 if (leader == null && queue[0] != null)  
 available.signal();  
 lock.unlock();  
 }  
}

这里take就是一个重入锁，加上条件等待调度时间，通过for自旋返回任务来进行阻塞

That’s All。Thank You