# **一行一行源码分析清楚 AbstractQueuedSynchronizer (三)**

https://javadoop.com/post/AbstractQueuedSynchronizer-3

创建时间: 2022-12-04 19:17:56

这篇文章是 AQS 系列的最后一篇，第一篇，我们通过 ReentrantLock 公平锁分析了 AQS 的核心，第二篇的重点是把 Condition 说明白，同时也说清楚了对于线程中断的使用。

这篇，我们的关注点是 AQS 最后的部分，****AQS 共享模式****的使用。有前两篇文章的铺垫，剩下的源码分析将会简单很多。

本文先用 CountDownLatch 将共享模式说清楚，然后顺着把其他 AQS 相关的类 CyclicBarrier、Semaphore 的源码一起过一下。

相对来说，如果读者有前面两篇文章的基础，这篇文章是简单很多，不过对于初学者来说，1 小时估计也是免不了的。

## **CountDownLatch**

CountDownLatch 这个类是比较典型的 AQS 的共享模式的使用，这是一个高频使用的类。latch 的中文意思是****门栓、栅栏****，具体怎么解释我就不废话了，大家随意，看两个例子就知道在哪里用、怎么用了。

### **使用例子**

我们看下 Doug Lea 在 java doc 中给出的例子，这个例子非常实用，我经常会写到这个代码。

假设我们有 N ( N > 0 ) 个任务，那么我们会用 N 来初始化一个 CountDownLatch，然后将这个 latch 的引用传递到各个线程中，在每个线程完成了任务后，调用 latch.countDown() 代表完成了一个任务。

调用 latch.await() 的方法的线程会阻塞，直到所有的任务完成。

class Driver2 { // ...

void main() throws InterruptedException {

CountDownLatch doneSignal = new CountDownLatch(N);

Executor e = Executors.newFixedThreadPool(8);

// 创建 N 个任务，提交给线程池来执行

for (int i = 0; i < N; ++i) // create and start threads

e.execute(new WorkerRunnable(doneSignal, i));

// 等待所有的任务完成，这个方法才会返回

doneSignal.await(); // wait for all to finish

}

}

class WorkerRunnable implements Runnable {

private final CountDownLatch doneSignal;

private final int i;

WorkerRunnable(CountDownLatch doneSignal, int i) {

this.doneSignal = doneSignal;

this.i = i;

}

public void run() {

try {

doWork(i);

// 这个线程的任务完成了，调用 countDown 方法

doneSignal.countDown();

} catch (InterruptedException ex) {

} // return;

}

void doWork() { ...}

}

所以说 CountDownLatch 非常实用，我们常常会将一个比较大的任务进行拆分，然后开启多个线程来执行，等所有线程都执行完了以后，再往下执行其他操作。这里例子中，****只有 main 线程调用了 await 方法****。

我们再来看另一个例子，这个例子很典型，用了两个 CountDownLatch：

class Driver { // ...

void main() throws InterruptedException {

CountDownLatch startSignal = new CountDownLatch(1);

CountDownLatch doneSignal = new CountDownLatch(N);

for (int i = 0; i < N; ++i) // create and start threads

new Thread(new Worker(startSignal, doneSignal)).start();

// 这边插入一些代码，确保上面的每个线程先启动起来，才执行下面的代码。

doSomethingElse(); // don't let run yet

// 因为这里 N == 1，所以，只要调用一次，那么所有的 await 方法都可以通过

startSignal.countDown(); // let all threads proceed

doSomethingElse();

// 等待所有任务结束

doneSignal.await(); // wait for all to finish

}

}

class Worker implements Runnable {

private final CountDownLatch startSignal;

private final CountDownLatch doneSignal;

Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {

this.startSignal = startSignal;

this.doneSignal = doneSignal;

}

public void run() {

try {

// 为了让所有线程同时开始任务，我们让所有线程先阻塞在这里

// 等大家都准备好了，再打开这个门栓

startSignal.await();

doWork();

doneSignal.countDown();

} catch (InterruptedException ex) {

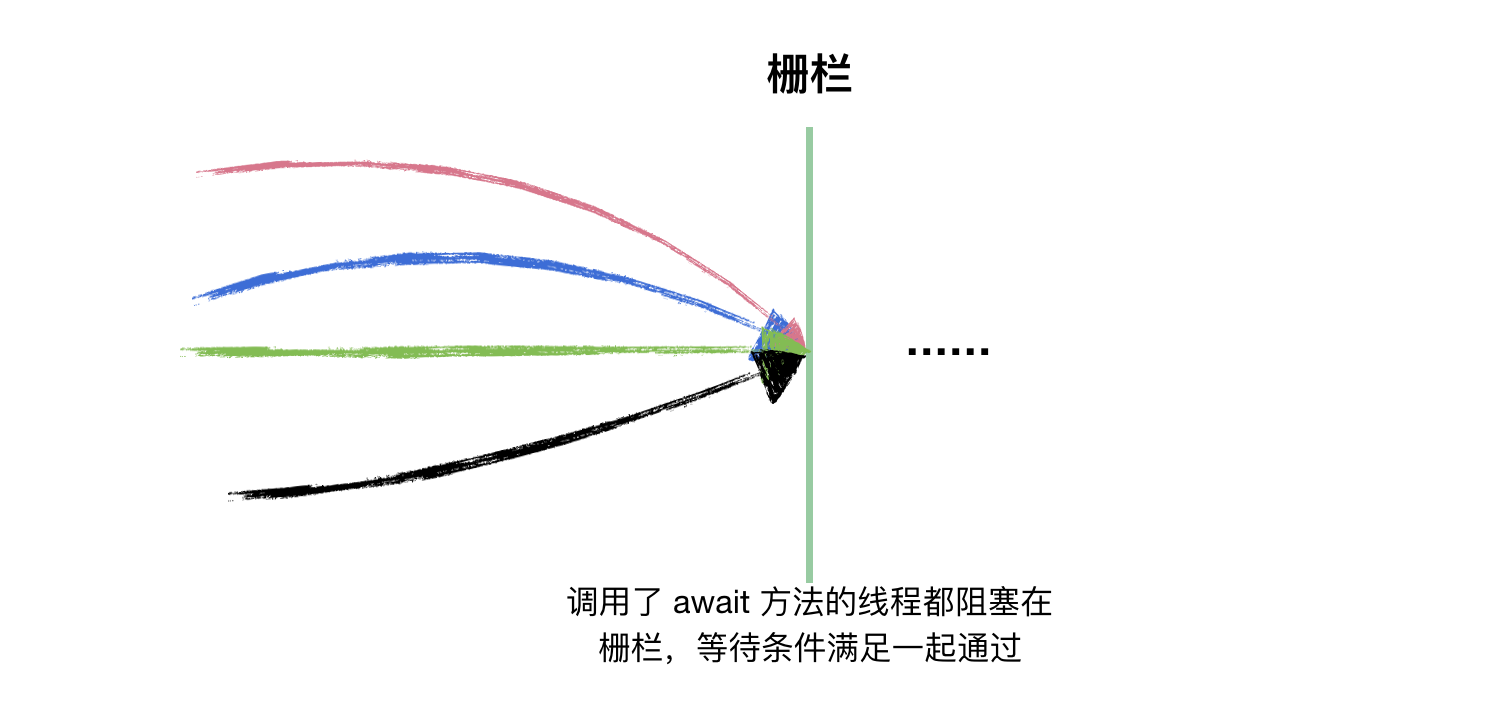
} // return;

}

void doWork() { ...}

}

这个例子中，doneSignal 同第一个例子的使用，我们说说这里的 startSignal。N 个新开启的线程都调用了startSignal.await() 进行阻塞等待，它们阻塞在****栅栏****上，只有当条件满足的时候（startSignal.countDown()），它们才能同时通过这个栅栏，目的是让所有的线程站在一个起跑线上。



如果始终只有一个线程调用 await 方法等待任务完成，那么 CountDownLatch 就会简单很多，所以之后的源码分析读者一定要在脑海中构建出这么一个场景：有 m 个线程是做任务的，有 n 个线程在某个栅栏上等待这 m 个线程做完任务，直到所有 m 个任务完成后，n 个线程同时通过栅栏。

### **源码分析**

Talk is cheap, show me the code.

构造方法，需要传入一个不小于 0 的整数：

public CountDownLatch(int count) {

if (count < 0) throw new IllegalArgumentException("count < 0");

this.sync = new Sync(count);

}// 老套路了，内部封装一个 Sync 类继承自 AQSprivate static final class Sync extends AbstractQueuedSynchronizer {

Sync(int count) {

// 这样就 state == count 了

setState(count);

}

...

}

代码都是套路，先分析套路：AQS 里面的 state 是一个整数值，这边用一个 int count 参数其实初始化就是设置了这个值，所有调用了 await 方法的等待线程会挂起，然后有其他一些线程会做 state = state - 1 操作，当 state 减到 0 的同时，那个将 state 减为 0 的线程会负责唤醒 所有调用了 await 方法的线程。都是套路啊，只是 Doug Lea 的套路很深，代码很巧妙，不然我们也没有要分析源码的必要。

对于 CountDownLatch，我们仅仅需要关心两个方法，一个是 countDown() 方法，另一个是 await() 方法。

countDown() 方法每次调用都会将 state 减 1，直到 state 的值为 0；而 await 是一个阻塞方法，当 state 减为 0 的时候，await 方法才会返回。await 可以被多个线程调用，读者这个时候脑子里要有个图：所有调用了 await 方法的线程阻塞在 AQS 的阻塞队列中，等待条件满足（state == 0），将线程从队列中一个个唤醒过来。

我们用以下程序来分析源码，t1 和 t2 负责调用 countDown() 方法，t3 和 t4 调用 await 方法阻塞：

public class CountDownLatchDemo {

public static void main(String[] args) {

CountDownLatch latch = new CountDownLatch(2);

Thread t1 = new Thread(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(5000);

} catch (InterruptedException ignore) {

}

// 休息 5 秒后(模拟线程工作了 5 秒)，调用 countDown()

latch.countDown();

}

}, "t1");

Thread t2 = new Thread(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(10000);

} catch (InterruptedException ignore) {

}

// 休息 10 秒后(模拟线程工作了 10 秒)，调用 countDown()

latch.countDown();

}

}, "t2");

t1.start();

t2.start();

Thread t3 = new Thread(new Runnable() {

@Override

public void run() {

try {

// 阻塞，等待 state 减为 0

latch.await();

System.out.println("线程 t3 从 await 中返回了");

} catch (InterruptedException e) {

System.out.println("线程 t3 await 被中断");

Thread.currentThread().interrupt();

}

}

}, "t3");

Thread t4 = new Thread(new Runnable() {

@Override

public void run() {

try {

// 阻塞，等待 state 减为 0

latch.await();

System.out.println("线程 t4 从 await 中返回了");

} catch (InterruptedException e) {

System.out.println("线程 t4 await 被中断");

Thread.currentThread().interrupt();

}

}

}, "t4");

t3.start();

t4.start();

}

}

上述程序，大概在过了 10 秒左右的时候，会输出：

线程 t3 从 await 中返回了

线程 t4 从 await 中返回了

这两条输出，顺序不是绝对的

后面的分析，我们假设 t3 先进入阻塞队列

接下来，我们按照流程一步一步走：先 await 等待，然后被唤醒，await 方法返回。

首先，我们来看 await() 方法，它代表线程阻塞，等待 state 的值减为 0。

public void await() throws InterruptedException {

sync.acquireSharedInterruptibly(1);

}public final void acquireSharedInterruptibly(int arg)

throws InterruptedException {

// 这也是老套路了，我在第二篇的中断那一节说过了

if (Thread.interrupted())

throw new InterruptedException();

// t3 和 t4 调用 await 的时候，state 都大于 0（state 此时为 2）。

// 也就是说，这个 if 返回 true，然后往里看

if (tryAcquireShared(arg) < 0)

doAcquireSharedInterruptibly(arg);

}// 只有当 state == 0 的时候，这个方法才会返回 1protected int tryAcquireShared(int acquires) {

return (getState() == 0) ? 1 : -1;

}

从方法名我们就可以看出，这个方法是获取共享锁，并且此方法是可中断的（中断的时候抛出 InterruptedException 退出这个方法）。

private void doAcquireSharedInterruptibly(int arg)

throws InterruptedException {

// 1. 入队

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();

if (p == head) {

// 同上，只要 state 不等于 0，那么这个方法返回 -1

int r = tryAcquireShared(arg);

if (r >= 0) {

setHeadAndPropagate(node, r);

p.next = null; // help GC

failed = false;

return;

}

}

// 2

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

throw new InterruptedException();

}

} finally {

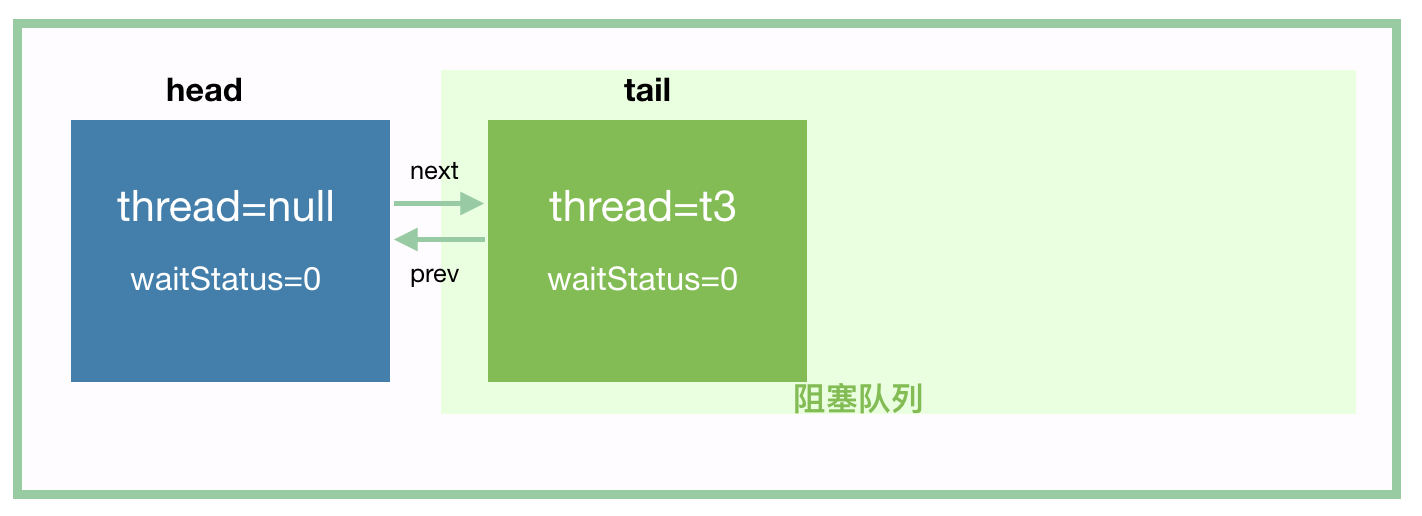
if (failed)

cancelAcquire(node);

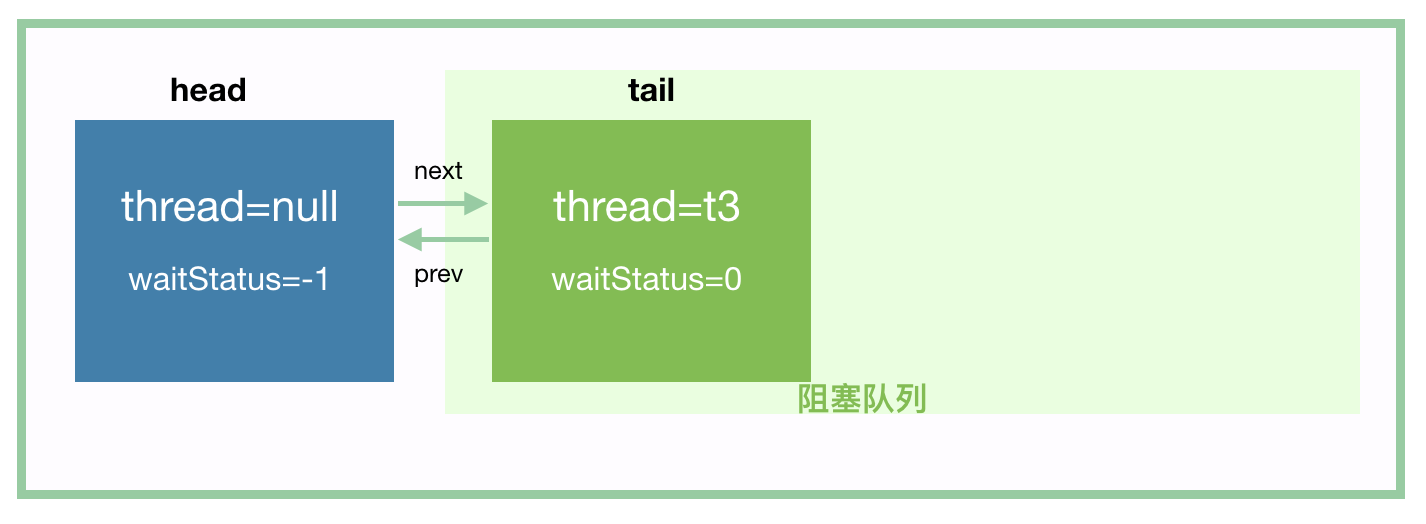
}

}

我们来仔细分析这个方法，线程 t3 经过第 1 步 addWaiter 入队以后，我们应该可以得到这个：

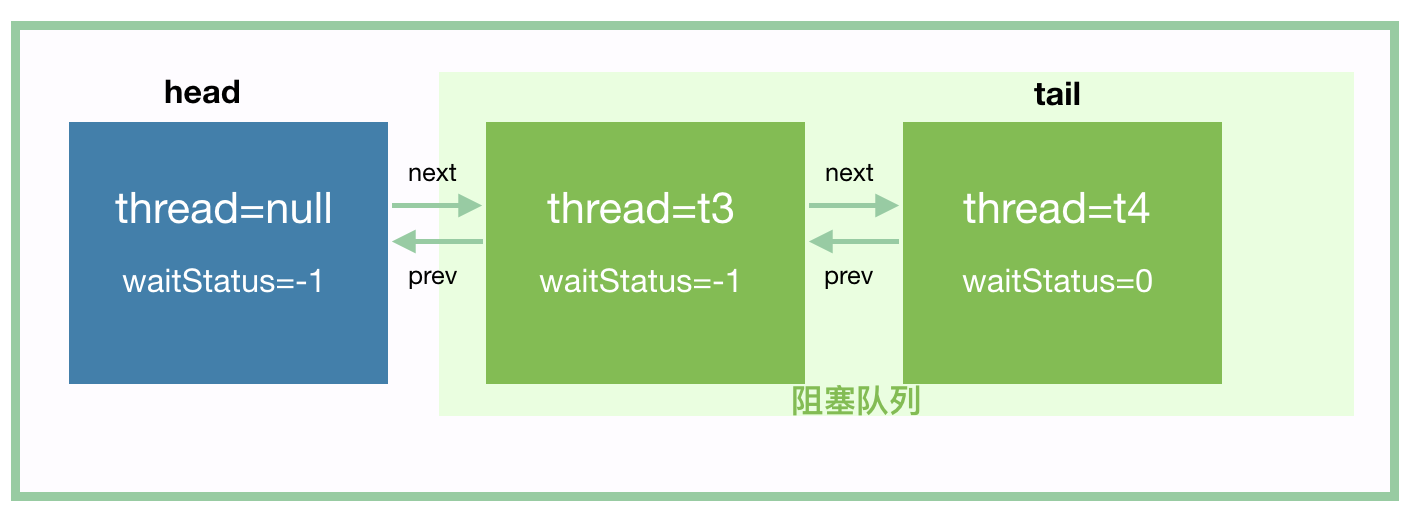


由于 tryAcquireShared 这个方法会返回 -1，所以 if (r >= 0) 这个分支不会进去。到 shouldParkAfterFailedAcquire 的时候，t3 将 head 的 waitStatus 值设置为 -1，如下：



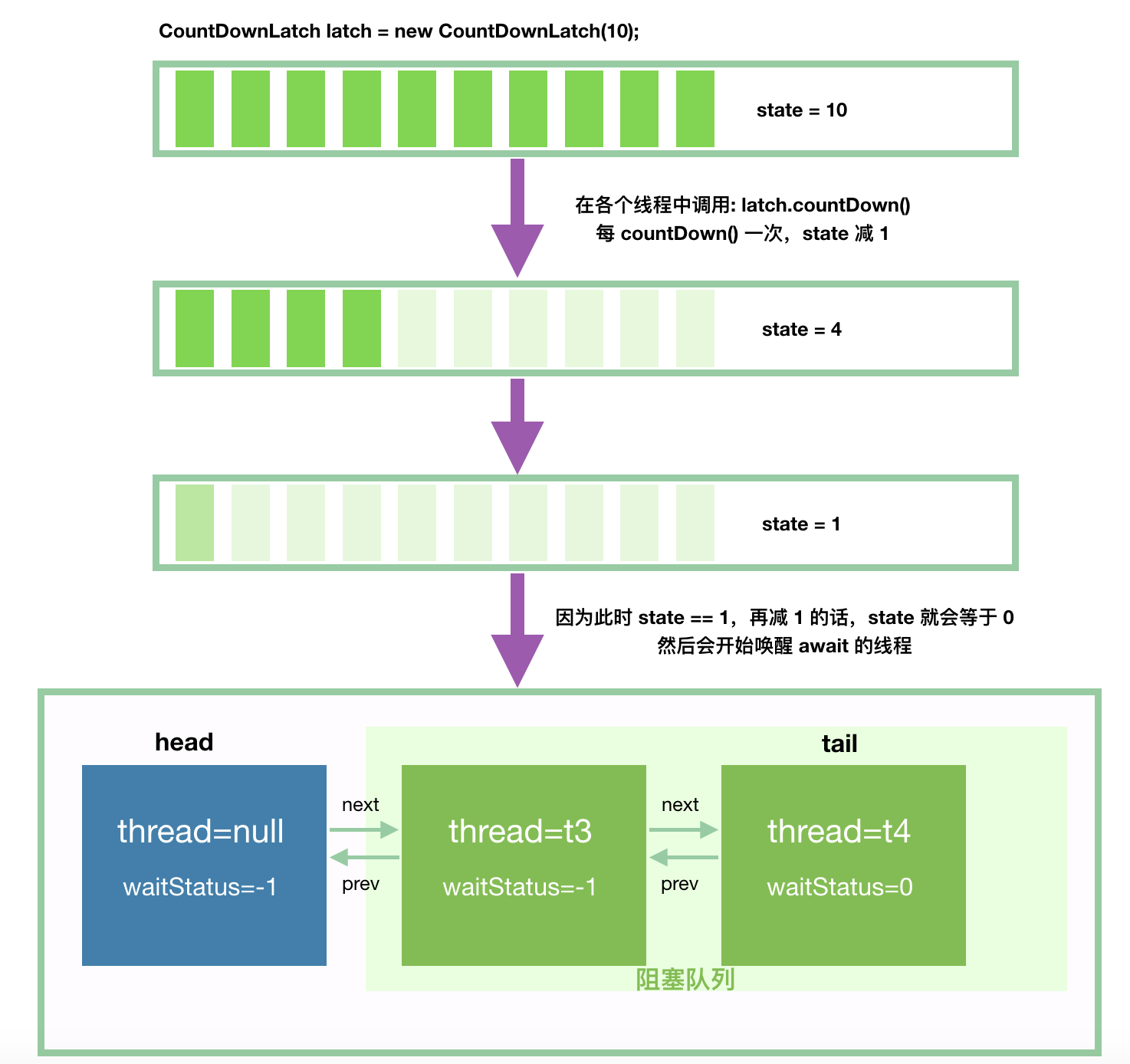
然后进入到 parkAndCheckInterrupt 的时候，t3 挂起。

我们再分析 t4 入队，t4 会将前驱节点 t3 所在节点的 waitStatus 设置为 -1，t4 入队后，应该是这样的：



然后，t4 也挂起。接下来，t3 和 t4 就等待唤醒了。

接下来，我们来看唤醒的流程。为了让下面的示意图更丰富些，我们假设用 10 初始化 CountDownLatch。



当然，我们的例子中，其实没有 10 个线程，只有 2 个线程 t1 和 t2，只是为了让图好看些罢了。

我们再一步步看具体的流程。首先，我们看 countDown() 方法:

public void countDown() {

sync.releaseShared(1);

}public final boolean releaseShared(int arg) {

// 只有当 state 减为 0 的时候，tryReleaseShared 才返回 true

// 否则只是简单的 state = state - 1 那么 countDown() 方法就结束了

// 将 state 减到 0 的那个操作才是最复杂的，继续往下吧

if (tryReleaseShared(arg)) {

// 唤醒 await 的线程

doReleaseShared();

return true;

}

return false;

}// 这个方法很简单，用自旋的方法实现 state 减 1protected boolean tryReleaseShared(int releases) {

for (;;) {

int c = getState();

if (c == 0)

return false;

int nextc = c-1;

if (compareAndSetState(c, nextc))

return nextc == 0;

}

}

countDown 方法就是每次调用都将 state 值减 1，如果 state 减到 0 了，那么就调用下面的方法进行唤醒阻塞队列中的线程：

// 调用这个方法的时候，state == 0// 这个方法先不要看所有的代码，按照思路往下到我写注释的地方，我们先跑通一个流程，其他的之后还会仔细分析private void doReleaseShared() {

for (;;) {

Node h = head;

if (h != null && h != tail) {

int ws = h.waitStatus;

// t3 入队的时候，已经将头节点的 waitStatus 设置为 Node.SIGNAL（-1） 了

if (ws == Node.SIGNAL) {

// 将 head 的 waitStatue 设置为 0

if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))

continue; // loop to recheck cases

// 就是这里，唤醒 head 的后继节点，也就是阻塞队列中的第一个节点

// 在这里，也就是唤醒 t3

unparkSuccessor(h);

}

else if (ws == 0 &&

!compareAndSetWaitStatus(h, 0, Node.PROPAGATE)) // todo

continue; // loop on failed CAS

}

if (h == head) // loop if head changed

break;

}

}

一旦 t3 被唤醒后，我们继续回到 await 的这段代码，parkAndCheckInterrupt 返回，我们先不考虑中断的情况：

private void doAcquireSharedInterruptibly(int arg)

throws InterruptedException {

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();

if (p == head) {

int r = tryAcquireShared(arg);

if (r >= 0) {

setHeadAndPropagate(node, r); // 2. 这里是下一步

p.next = null; // help GC

failed = false;

return;

}

}

if (shouldParkAfterFailedAcquire(p, node) &&

// 1. 唤醒后这个方法返回

parkAndCheckInterrupt())

throw new InterruptedException();

}

} finally {

if (failed)

cancelAcquire(node);

}

}

接下来，t3 会进到 setHeadAndPropagate(node, r) 这个方法，先把 head 给占了，然后唤醒队列中其他的线程：

private void setHeadAndPropagate(Node node, int propagate) {

Node h = head; // Record old head for check below

setHead(node);

// 下面说的是，唤醒当前 node 之后的节点，即 t3 已经醒了，马上唤醒 t4

// 类似的，如果 t4 后面还有 t5，那么 t4 醒了以后，马上将 t5 给唤醒了

if (propagate > 0 || h == null || h.waitStatus < 0 ||

(h = head) == null || h.waitStatus < 0) {

Node s = node.next;

if (s == null || s.isShared())

// 又是这个方法，只是现在的 head 已经不是原来的空节点了，是 t3 的节点了

doReleaseShared();

}

}

又回到这个方法了，那么接下来，我们好好分析 doReleaseShared 这个方法，我们根据流程，头节点 head 此时是 t3 节点了：

// 调用这个方法的时候，state == 0private void doReleaseShared() {

for (;;) {

Node h = head;

// 1. h == null: 说明阻塞队列为空

// 2. h == tail: 说明头结点可能是刚刚初始化的头节点，

// 或者是普通线程节点，但是此节点既然是头节点了，那么代表已经被唤醒了，阻塞队列没有其他节点了

// 所以这两种情况不需要进行唤醒后继节点

if (h != null && h != tail) {

int ws = h.waitStatus;

// t4 将头节点(此时是 t3)的 waitStatus 设置为 Node.SIGNAL（-1） 了

if (ws == Node.SIGNAL) {

// 这里 CAS 失败的场景请看下面的解读

if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))

continue; // loop to recheck cases

// 就是这里，唤醒 head 的后继节点，也就是阻塞队列中的第一个节点

// 在这里，也就是唤醒 t4

unparkSuccessor(h);

}

else if (ws == 0 &&

// 这个 CAS 失败的场景是：执行到这里的时候，刚好有一个节点入队，入队会将这个 ws 设置为 -1

!compareAndSetWaitStatus(h, 0, Node.PROPAGATE))

continue; // loop on failed CAS

}

// 如果到这里的时候，前面唤醒的线程已经占领了 head，那么再循环

// 否则，就是 head 没变，那么退出循环，

// 退出循环是不是意味着阻塞队列中的其他节点就不唤醒了？当然不是，唤醒的线程之后还是会调用这个方法的

if (h == head) // loop if head changed

break;

}

}

我们分析下最后一个 if 语句，然后才能解释第一个 CAS 为什么可能会失败：

1. h == head：说明头节点还没有被刚刚用 unparkSuccessor 唤醒的线程（这里可以理解为 t4）占有，此时 break 退出循环。
2. h != head：头节点被刚刚唤醒的线程（这里可以理解为 t4）占有，那么这里重新进入下一轮循环，唤醒下一个节点（这里是 t4 ）。我们知道，等到 t4 被唤醒后，其实是会主动唤醒 t5、t6、t7...，那为什么这里要进行下一个循环来唤醒 t5 呢？我觉得是出于吞吐量的考虑。

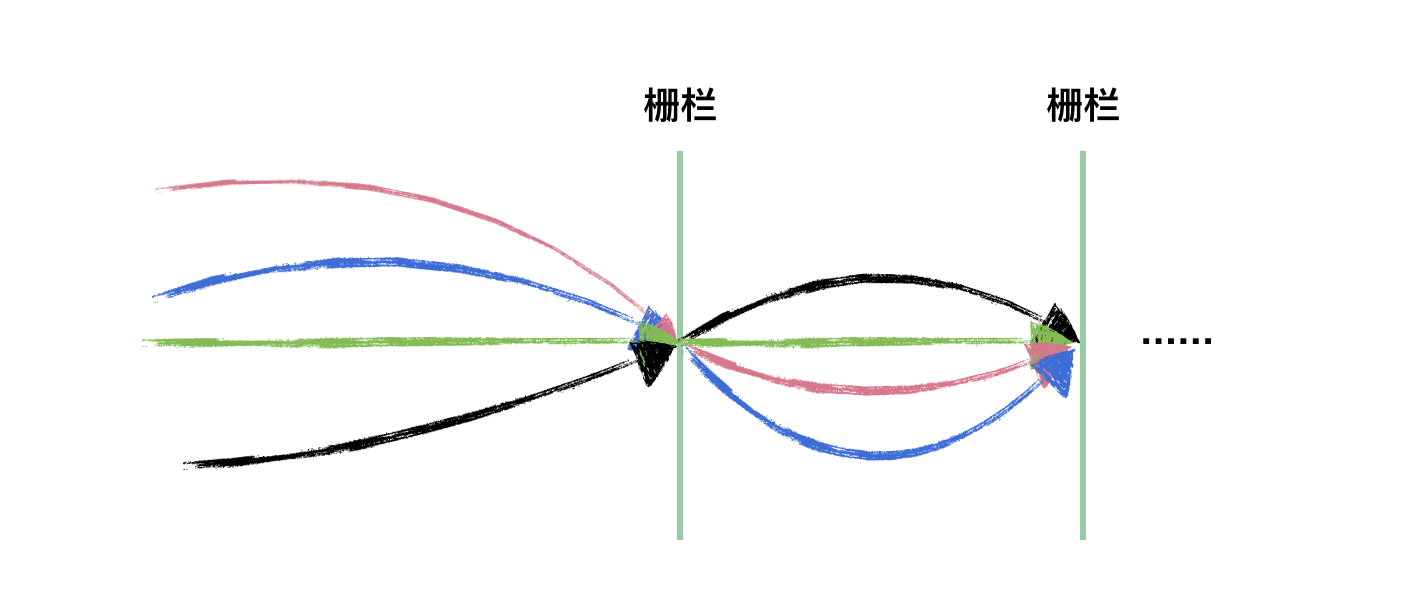
满足上面的 2 的场景，那么我们就能知道为什么上面的 CAS 操作 compareAndSetWaitStatus(h, Node.SIGNAL, 0) 会失败了？

因为当前进行 for 循环的线程到这里的时候，可能刚刚唤醒的线程 t4 也刚刚好到这里了，那么就有可能 CAS 失败了。

for 循环第一轮的时候会唤醒 t4，t4 醒后会将自己设置为头节点，如果在 t4 设置头节点后，for 循环才跑到 if (h == head)，那么此时会返回 false，for 循环会进入下一轮。t4 唤醒后也会进入到这个方法里面，那么 for 循环第二轮和 t4 就有可能在这个 CAS 相遇，那么就只会有一个成功了。

## **CyclicBarrier**

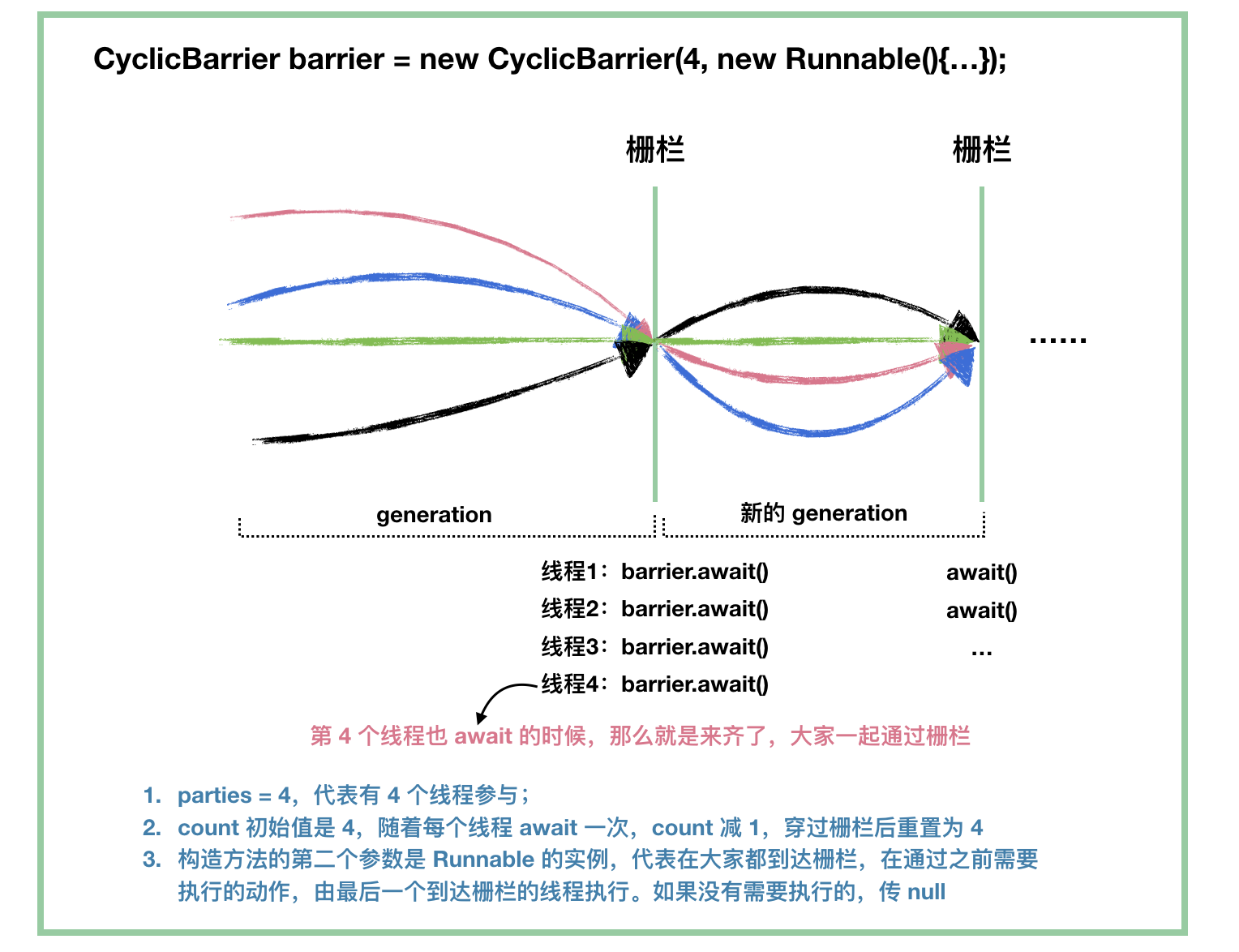
字面意思是“可重复使用的栅栏”或“周期性的栅栏”，总之不是用了一次就没用了的，CyclicBarrier 相比 CountDownLatch 来说，要简单很多，其源码没有什么高深的地方，它是 ReentrantLock 和 Condition 的组合使用。看如下示意图，CyclicBarrier 和 CountDownLatch 是不是很像，只是 CyclicBarrier 可以有不止一个栅栏，因为它的栅栏（Barrier）可以重复使用（Cyclic）。



首先，CyclicBarrier 的源码实现和 CountDownLatch 大相径庭，CountDownLatch 基于 AQS 的共享模式的使用，而 CyclicBarrier 基于 Condition 来实现。

因为 CyclicBarrier 的源码相对来说简单许多，读者只要熟悉了前面关于 Condition 的分析，那么这里的源码是毫无压力的，就是几个特殊概念罢了。

先用一张图来描绘下 CyclicBarrier 里面的一些概念，和它的基本使用流程：



看图我们也知道了，CyclicBarrier 的源码最重要的就是 await() 方法了。

大家先把图看完，然后我们开始源码分析：

public class CyclicBarrier {

// 我们说了，CyclicBarrier 是可以重复使用的，我们把每次从开始使用到穿过栅栏当做"一代"，或者"一个周期"

private static class Generation {

boolean broken = false;

}

/\*\* The lock for guarding barrier entry \*/

private final ReentrantLock lock = new ReentrantLock();

// CyclicBarrier 是基于 Condition 的

// Condition 是“条件”的意思，CyclicBarrier 的等待线程通过 barrier 的“条件”是大家都到了栅栏上

private final Condition trip = lock.newCondition();

// 参与的线程数

private final int parties;

// 如果设置了这个，代表越过栅栏之前，要执行相应的操作

private final Runnable barrierCommand;

// 当前所处的“代”

private Generation generation = new Generation();

// 还没有到栅栏的线程数，这个值初始为 parties，然后递减

// 还没有到栅栏的线程数 = parties - 已经到栅栏的数量

private int count;

public CyclicBarrier(int parties, Runnable barrierAction) {

if (parties <= 0) throw new IllegalArgumentException();

this.parties = parties;

this.count = parties;

this.barrierCommand = barrierAction;

}

public CyclicBarrier(int parties) {

this(parties, null);

}

首先，先看怎么开启新的一代：

// 开启新的一代，当最后一个线程到达栅栏上的时候，调用这个方法来唤醒其他线程，同时初始化“下一代”private void nextGeneration() {

// 首先，需要唤醒所有的在栅栏上等待的线程

trip.signalAll();

// 更新 count 的值

count = parties;

// 重新生成“新一代”

generation = new Generation();

}

开启新的一代，类似于重新实例化一个 CyclicBarrier 实例

看看怎么打破一个栅栏：

private void breakBarrier() {

// 设置状态 broken 为 true

generation.broken = true;

// 重置 count 为初始值 parties

count = parties;

// 唤醒所有已经在等待的线程

trip.signalAll();

}

这两个方法之后用得到，现在开始分析最重要的等待通过栅栏方法 await 方法：

// 不带超时机制public int await() throws InterruptedException, BrokenBarrierException {

try {

return dowait(false, 0L);

} catch (TimeoutException toe) {

throw new Error(toe); // cannot happen

}

}// 带超时机制，如果超时抛出 TimeoutException 异常public int await(long timeout, TimeUnit unit)

throws InterruptedException,

BrokenBarrierException,

TimeoutException {

return dowait(true, unit.toNanos(timeout));

}

继续往里看：

private int dowait(boolean timed, long nanos)

throws InterruptedException, BrokenBarrierException,

TimeoutException {

final ReentrantLock lock = this.lock;

// 先要获取到锁，然后在 finally 中要记得释放锁

// 如果记得 Condition 部分的话，我们知道 condition 的 await() 会释放锁，被 signal() 唤醒的时候需要重新获取锁

lock.lock();

try {

final Generation g = generation;

// 检查栅栏是否被打破，如果被打破，抛出 BrokenBarrierException 异常

if (g.broken)

throw new BrokenBarrierException();

// 检查中断状态，如果中断了，抛出 InterruptedException 异常

if (Thread.interrupted()) {

breakBarrier();

throw new InterruptedException();

}

// index 是这个 await 方法的返回值

// 注意到这里，这个是从 count 递减后得到的值

int index = --count;

// 如果等于 0，说明所有的线程都到栅栏上了，准备通过

if (index == 0) { // tripped

boolean ranAction = false;

try {

// 如果在初始化的时候，指定了通过栅栏前需要执行的操作，在这里会得到执行

final Runnable command = barrierCommand;

if (command != null)

command.run();

// 如果 ranAction 为 true，说明执行 command.run() 的时候，没有发生异常退出的情况

ranAction = true;

// 唤醒等待的线程，然后开启新的一代

nextGeneration();

return 0;

} finally {

if (!ranAction)

// 进到这里，说明执行指定操作的时候，发生了异常，那么需要打破栅栏

// 之前我们说了，打破栅栏意味着唤醒所有等待的线程，设置 broken 为 true，重置 count 为 parties

breakBarrier();

}

}

// loop until tripped, broken, interrupted, or timed out

// 如果是最后一个线程调用 await，那么上面就返回了

// 下面的操作是给那些不是最后一个到达栅栏的线程执行的

for (;;) {

try {

// 如果带有超时机制，调用带超时的 Condition 的 await 方法等待，直到最后一个线程调用 await

if (!timed)

trip.await();

else if (nanos > 0L)

nanos = trip.awaitNanos(nanos);

} catch (InterruptedException ie) {

// 如果到这里，说明等待的线程在 await（是 Condition 的 await）的时候被中断

if (g == generation && ! g.broken) {

// 打破栅栏

breakBarrier();

// 打破栅栏后，重新抛出这个 InterruptedException 异常给外层调用的方法

throw ie;

} else {

// 到这里，说明 g != generation, 说明新的一代已经产生，即最后一个线程 await 执行完成，

// 那么此时没有必要再抛出 InterruptedException 异常，记录下来这个中断信息即可

// 或者是栅栏已经被打破了，那么也不应该抛出 InterruptedException 异常，

// 而是之后抛出 BrokenBarrierException 异常

Thread.currentThread().interrupt();

}

}

// 唤醒后，检查栅栏是否是“破的”

if (g.broken)

throw new BrokenBarrierException();

// 这个 for 循环除了异常，就是要从这里退出了

// 我们要清楚，最后一个线程在执行完指定任务(如果有的话)，会调用 nextGeneration 来开启一个新的代

// 然后释放掉锁，其他线程从 Condition 的 await 方法中得到锁并返回，然后到这里的时候，其实就会满足 g != generation 的

// 那什么时候不满足呢？barrierCommand 执行过程中抛出了异常，那么会执行打破栅栏操作，

// 设置 broken 为true，然后唤醒这些线程。这些线程会从上面的 if (g.broken) 这个分支抛 BrokenBarrierException 异常返回

// 当然，还有最后一种可能，那就是 await 超时，此种情况不会从上面的 if 分支异常返回，也不会从这里返回，会执行后面的代码

if (g != generation)

return index;

// 如果醒来发现超时了，打破栅栏，抛出异常

if (timed && nanos <= 0L) {

breakBarrier();

throw new TimeoutException();

}

}

} finally {

lock.unlock();

}

}

好了，我想我应该讲清楚了吧，我好像几乎没有漏掉任何一行代码吧？

下面开始收尾工作。

首先，我们看看怎么得到有多少个线程到了栅栏上，处于等待状态：

public int getNumberWaiting() {

final ReentrantLock lock = this.lock;

lock.lock();

try {

return parties - count;

} finally {

lock.unlock();

}

}

判断一个栅栏是否被打破了，这个很简单，直接看 broken 的值即可：

public boolean isBroken() {

final ReentrantLock lock = this.lock;

lock.lock();

try {

return generation.broken;

} finally {

lock.unlock();

}

}

前面我们在说 await 的时候也几乎说清楚了，什么时候栅栏会被打破，总结如下：

1. 中断，我们说了，如果某个等待的线程发生了中断，那么会打破栅栏，同时抛出 InterruptedException 异常；
2. 超时，打破栅栏，同时抛出 TimeoutException 异常；
3. 指定执行的操作抛出了异常，这个我们前面也说过。

最后，我们来看看怎么重置一个栅栏：

public void reset() {

final ReentrantLock lock = this.lock;

lock.lock();

try {

breakBarrier(); // break the current generation

nextGeneration(); // start a new generation

} finally {

lock.unlock();

}

}

我们设想一下，如果初始化时，指定了线程 parties = 4，前面有 3 个线程调用了 await 等待，在第 4 个线程调用 await 之前，我们调用 reset 方法，那么会发生什么？

首先，打破栅栏，那意味着所有等待的线程（3个等待的线程）会唤醒，await 方法会通过抛出 BrokenBarrierException 异常返回。然后开启新的一代，重置了 count 和 generation，相当于一切归零了。

怎么样，CyclicBarrier 源码很简单吧。

## **Semaphore**

有了 CountDownLatch 的基础后，分析 Semaphore 会简单很多。Semaphore 是什么呢？它类似一个资源池（读者可以类比线程池），每个线程需要调用 acquire() 方法获取资源，然后才能执行，执行完后，需要 release 资源，让给其他的线程用。

大概大家也可以猜到，Semaphore 其实也是 AQS 中共享锁的使用，因为每个线程共享一个池嘛。

套路解读：创建 Semaphore 实例的时候，需要一个参数 permits，这个基本上可以确定是设置给 AQS 的 state 的，然后每个线程调用 acquire 的时候，执行 state = state - 1，release 的时候执行 state = state + 1，当然，acquire 的时候，如果 state = 0，说明没有资源了，需要等待其他线程 release。

构造方法：

public Semaphore(int permits) {

sync = new NonfairSync(permits);

}

public Semaphore(int permits, boolean fair) {

sync = fair ? new FairSync(permits) : new NonfairSync(permits);

}

这里和 ReentrantLock 类似，用了公平策略和非公平策略。

看 acquire 方法：

public void acquire() throws InterruptedException {

sync.acquireSharedInterruptibly(1);

}public void acquireUninterruptibly() {

sync.acquireShared(1);

}public void acquire(int permits) throws InterruptedException {

if (permits < 0) throw new IllegalArgumentException();

sync.acquireSharedInterruptibly(permits);

}public void acquireUninterruptibly(int permits) {

if (permits < 0) throw new IllegalArgumentException();

sync.acquireShared(permits);

}

这几个方法也是老套路了，大家基本都懂了吧，这边多了两个可以传参的 acquire 方法，不过大家也都懂的吧，如果我们需要一次获取超过一个的资源，会用得着这个的。

我们接下来看不抛出 InterruptedException 异常的 acquireUninterruptibly() 方法吧：

public void acquireUninterruptibly() {

sync.acquireShared(1);

}public final void acquireShared(int arg) {

if (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

前面说了，Semaphore 分公平策略和非公平策略，我们对比一下两个 tryAcquireShared 方法：

// 公平策略：protected int tryAcquireShared(int acquires) {

for (;;) {

// 区别就在于是不是会先判断是否有线程在排队，然后才进行 CAS 减操作

if (hasQueuedPredecessors())

return -1;

int available = getState();

int remaining = available - acquires;

if (remaining < 0 ||

compareAndSetState(available, remaining))

return remaining;

}

}// 非公平策略：protected int tryAcquireShared(int acquires) {

return nonfairTryAcquireShared(acquires);

}final int nonfairTryAcquireShared(int acquires) {

for (;;) {

int available = getState();

int remaining = available - acquires;

if (remaining < 0 ||

compareAndSetState(available, remaining))

return remaining;

}

}

也是老套路了，所以从源码分析角度的话，我们其实不太需要关心是不是公平策略还是非公平策略，它们的区别往往就那么一两行。

我们再回到 acquireShared 方法，

public final void acquireShared(int arg) {

if (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

由于 tryAcquireShared(arg) 返回小于 0 的时候，说明 state 已经小于 0 了（没资源了），此时 acquire 不能立马拿到资源，需要进入到阻塞队列等待，虽然贴了很多代码，不在乎多这点了：

private void doAcquireShared(int arg) {

final Node node = addWaiter(Node.SHARED);

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

if (p == head) {

int r = tryAcquireShared(arg);

if (r >= 0) {

setHeadAndPropagate(node, r);

p.next = null; // help GC

if (interrupted)

selfInterrupt();

failed = false;

return;

}

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

这个方法我就不介绍了，线程挂起后等待有资源被 release 出来。接下来，我们就要看 release 的方法了：

// 任务介绍，释放一个资源public void release() {

sync.releaseShared(1);

}public final boolean releaseShared(int arg) {

if (tryReleaseShared(arg)) {

doReleaseShared();

return true;

}

return false;

}

protected final boolean tryReleaseShared(int releases) {

for (;;) {

int current = getState();

int next = current + releases;

// 溢出，当然，我们一般也不会用这么大的数

if (next < current) // overflow

throw new Error("Maximum permit count exceeded");

if (compareAndSetState(current, next))

return true;

}

}

tryReleaseShared 方法总是会返回 true，然后是 doReleaseShared，这个也是我们熟悉的方法了，我就贴下代码，不分析了，这个方法用于唤醒所有的等待线程：

private void doReleaseShared() {

for (;;) {

Node h = head;

if (h != null && h != tail) {

int ws = h.waitStatus;

if (ws == Node.SIGNAL) {

if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))

continue; // loop to recheck cases

unparkSuccessor(h);

}

else if (ws == 0 &&

!compareAndSetWaitStatus(h, 0, Node.PROPAGATE))

continue; // loop on failed CAS

}

if (h == head) // loop if head changed

break;

}

}

Semphore 的源码确实很简单，基本上都是分析过的老代码的组合使用了。

## **总结**

写到这里，终于把 AbstractQueuedSynchronizer 基本上说完了，对于 Java 并发，Doug Lea 真的是神一样的存在。日后我们还会接触到很多 Doug Lea 的代码，希望我们大家都可以朝着大神的方向不断打磨自己的技术，少一些高大上的架构，多一些实实在在的优秀代码吧。

（全文完）