# **一行一行源码分析清楚 AbstractQueuedSynchronizer (二)**

https://javadoop.com/post/AbstractQueuedSynchronizer-2

创建时间: 2022-12-04 19:17:09

文章比较长，信息量比较大，建议在 pc 上阅读。文章标题是为了呼应前文，其实可以单独成文的，主要是希望读者看文章能系统看。

本文关注以下几点内容：

1. 深入理解 ReentrantLock 公平锁和非公平锁的区别
2. 深入分析 AbstractQueuedSynchronizer 中的 ConditionObject
3. 深入理解 Java 线程中断和 InterruptedException 异常

基本上本文把以上几点都说清楚了，我假设读者看过[上一篇文章中对 AbstractQueuedSynchronizer 的介绍](http://hongjiev.github.io/2017/06/16/AbstractQueuedSynchronizer/) ，当然如果你已经熟悉 AQS 中的独占锁了，那也可以直接看这篇。各小节之间基本上没什么关系，大家可以只关注自己感兴趣的部分。

其实这篇文章的信息量很大，初学者估计****至少要 1 小时****才能看完，希望本文对得起大家的时间。

## **公平锁和非公平锁**

ReentrantLock 默认采用非公平锁，除非你在构造方法中传入参数 true 。

public ReentrantLock() {

// 默认非公平锁

sync = new NonfairSync();

}public ReentrantLock(boolean fair) {

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

}

公平锁的 lock 方法：

static final class FairSync extends Sync {

final void lock() {

acquire(1);

}

// AbstractQueuedSynchronizer.acquire(int arg)

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

protected final boolean tryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

// 1. 和非公平锁相比，这里多了一个判断：是否有线程在等待

if (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0)

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

setState(nextc);

return true;

}

return false;

}

}

非公平锁的 lock 方法：

static final class NonfairSync extends Sync {

final void lock() {

// 2. 和公平锁相比，这里会直接先进行一次CAS，成功就返回了

if (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());

else

acquire(1);

}

// AbstractQueuedSynchronizer.acquire(int arg)

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

protected final boolean tryAcquire(int acquires) {

return nonfairTryAcquire(acquires);

}

}/\*\*

\* Performs non-fair tryLock. tryAcquire is implemented in

\* subclasses, but both need nonfair try for trylock method.

\*/final boolean nonfairTryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

// 这里没有对阻塞队列进行判断

if (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0) // overflow

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

setState(nextc);

return true;

}

return false;

}

总结：公平锁和非公平锁只有两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

## **Condition**

Tips: 这里重申一下，要看懂这个，必须要先看懂上一篇关于 [AbstractQueuedSynchronizer](http://hongjiev.github.io/2017/06/16/AbstractQueuedSynchronizer/) 的介绍，或者你已经有相关的知识了，否则这节肯定是看不懂的。

我们先来看看 Condition 的使用场景，Condition 经常可以用在****生产者-消费者****的场景中，请看 Doug Lea 给出的这个例子：

import java.util.concurrent.locks.Condition;import java.util.concurrent.locks.Lock;import java.util.concurrent.locks.ReentrantLock;

class BoundedBuffer {

final Lock lock = new ReentrantLock();

// condition 依赖于 lock 来产生

final Condition notFull = lock.newCondition();

final Condition notEmpty = lock.newCondition();

final Object[] items = new Object[100];

int putptr, takeptr, count;

// 生产

public void put(Object x) throws InterruptedException {

lock.lock();

try {

while (count == items.length)

notFull.await(); // 队列已满，等待，直到 not full 才能继续生产

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

notEmpty.signal(); // 生产成功，队列已经 not empty 了，发个通知出去

} finally {

lock.unlock();

}

}

// 消费

public Object take() throws InterruptedException {

lock.lock();

try {

while (count == 0)

notEmpty.await(); // 队列为空，等待，直到队列 not empty，才能继续消费

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

notFull.signal(); // 被我消费掉一个，队列 not full 了，发个通知出去

return x;

} finally {

lock.unlock();

}

}

}

1、我们可以看到，在使用 condition 时，必须先持有相应的锁。这个和 Object 类中的方法有相似的语义，需要先持有某个对象的监视器锁才可以执行 wait(), notify() 或 notifyAll() 方法。

2、ArrayBlockingQueue 采用这种方式实现了生产者-消费者，所以请只把这个例子当做学习例子，实际生产中可以直接使用 ArrayBlockingQueue

我们常用 obj.wait()，obj.notify() 或 obj.notifyAll() 来实现相似的功能，但是，它们是基于对象的监视器锁的。需要深入了解这几个方法的读者，可以参考我的另一篇文章《[深入分析 java 8 编程语言规范：Threads and Locks](http://hongjiev.github.io/2017/07/05/Threads-And-Locks-md/)》。而这里说的 Condition 是基于 ReentrantLock 实现的，而 ReentrantLock 是依赖于 AbstractQueuedSynchronizer 实现的。

在往下看之前，读者心里要有一个整体的概念。condition 是依赖于 ReentrantLock 的，不管是调用 await 进入等待还是 signal 唤醒，****都必须获取到锁才能进行操作****。

每个 ReentrantLock 实例可以通过调用多次 newCondition 产生多个 ConditionObject 的实例：

final ConditionObject newCondition() {

// 实例化一个 ConditionObject

return new ConditionObject();

}

我们首先来看下我们关注的 Condition 的实现类 AbstractQueuedSynchronizer 类中的 ConditionObject。

public class ConditionObject implements Condition, java.io.Serializable {

private static final long serialVersionUID = 1173984872572414699L;

// 条件队列的第一个节点

// 不要管这里的关键字 transient，是不参与序列化的意思

private transient Node firstWaiter;

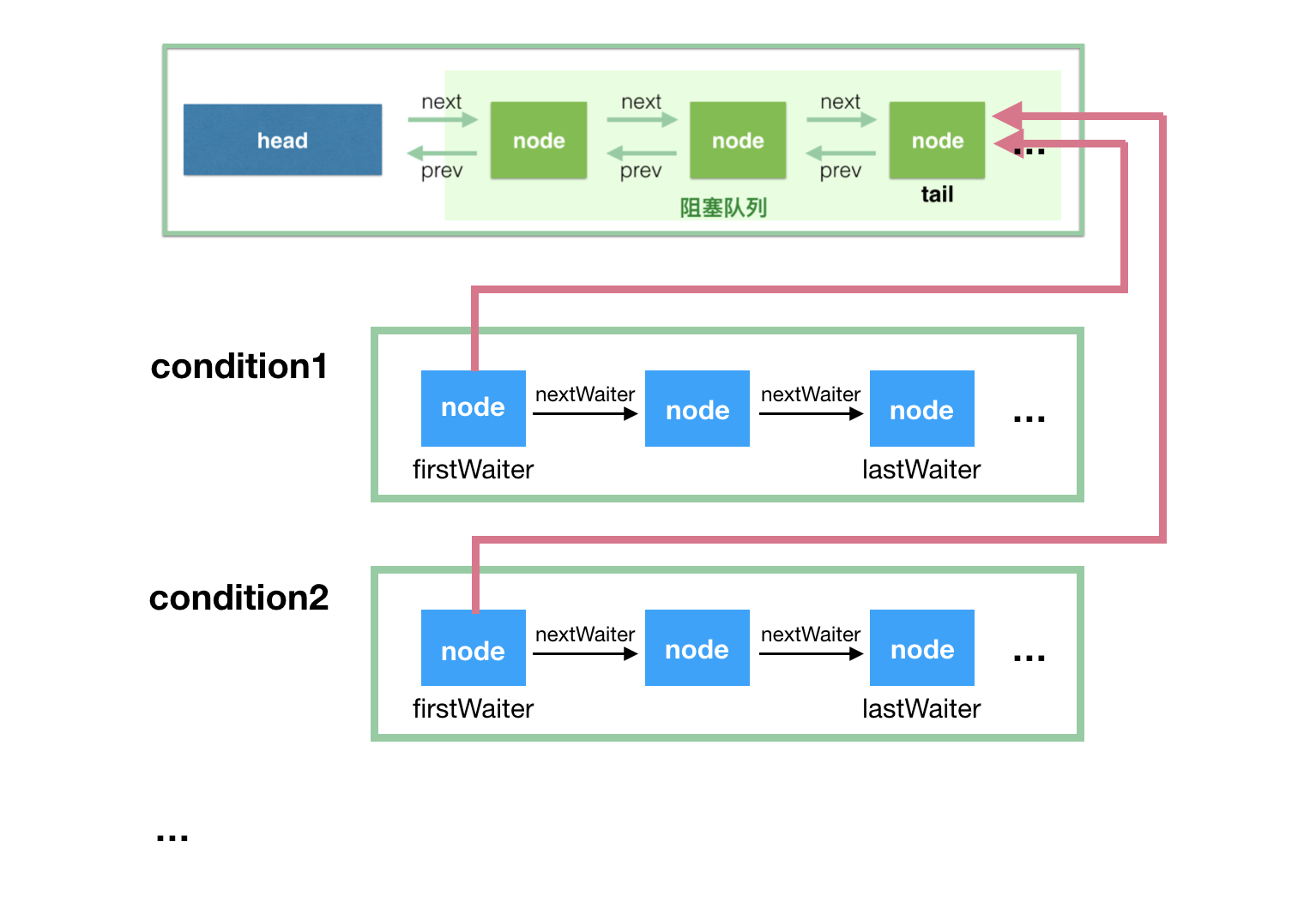
// 条件队列的最后一个节点

private transient Node lastWaiter;

......

在上一篇介绍 AQS 的时候，我们有一个****阻塞队列****，用于保存等待获取锁的线程的队列。这里我们引入另一个概念，叫****条件队列****（condition queue），我画了一张简单的图用来说明这个。

这里的阻塞队列如果叫做同步队列（sync queue）其实比较贴切，不过为了和前篇呼应，我就继续使用阻塞队列了。记住这里的两个概念，****阻塞队列****和****条件队列****。



这里，我们简单回顾下 Node 的属性：

volatile int waitStatus; // 可取值 0、CANCELLED(1)、SIGNAL(-1)、CONDITION(-2)、PROPAGATE(-3)volatile Node prev;volatile Node next;volatile Thread thread;

Node nextWaiter;

prev 和 next 用于实现阻塞队列的双向链表，这里的 nextWaiter 用于实现条件队列的单向链表

基本上，把这张图看懂，你也就知道 condition 的处理流程了。所以，我先简单解释下这图，然后再具体地解释代码实现。

1. 条件队列和阻塞队列的节点，都是 Node 的实例，因为条件队列的节点是需要转移到阻塞队列中去的；
2. 我们知道一个 ReentrantLock 实例可以通过多次调用 newCondition() 来产生多个 Condition 实例，这里对应 condition1 和 condition2。注意，ConditionObject 只有两个属性 firstWaiter 和 lastWaiter；
3. 每个 condition 有一个关联的****条件队列****，如线程 1 调用 condition1.await() 方法即可将当前线程 1 包装成 Node 后加入到条件队列中，然后阻塞在这里，不继续往下执行，条件队列是一个单向链表；
4. 调用condition1.signal() 触发一次唤醒，此时唤醒的是队头，会将condition1 对应的****条件队列****的 firstWaiter（队头） 移到****阻塞队列的队尾****，等待获取锁，获取锁后 await 方法才能返回，继续往下执行。

上面的 2->3->4 描述了一个最简单的流程，没有考虑中断、signalAll、还有带有超时参数的 await 方法等，不过把这里弄懂是这节的主要目的。

同时，从图中也可以很直观地看出，哪些操作是线程安全的，哪些操作是线程不安全的。

这个图看懂后，下面的代码分析就简单了。

接下来，我们一步步按照流程来走代码分析，我们先来看看 wait 方法：

// 首先，这个方法是可被中断的，不可被中断的是另一个方法 awaitUninterruptibly()// 这个方法会阻塞，直到调用 signal 方法（指 signal() 和 signalAll()，下同），或被中断public final void await() throws InterruptedException {

// 老规矩，既然该方法要响应中断，那么在最开始就判断中断状态

if (Thread.interrupted())

throw new InterruptedException();

// 添加到 condition 的条件队列中

Node node = addConditionWaiter();

// 释放锁，返回值是释放锁之前的 state 值

// await() 之前，当前线程是必须持有锁的，这里肯定要释放掉

int savedState = fullyRelease(node);

int interruptMode = 0;

// 这里退出循环有两种情况，之后再仔细分析

// 1. isOnSyncQueue(node) 返回 true，即当前 node 已经转移到阻塞队列了

// 2. checkInterruptWhileWaiting(node) != 0 会到 break，然后退出循环，代表的是线程中断

while (!isOnSyncQueue(node)) {

LockSupport.park(this);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

// 被唤醒后，将进入阻塞队列，等待获取锁

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null) // clean up if cancelled

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

}

其实，我大体上也把整个 await 过程说得十之八九了，下面我们分步把上面的几个点用源码说清楚。

### **1. 将节点加入到条件队列**

addConditionWaiter() 是将当前节点加入到条件队列，看图我们知道，这种条件队列内的操作是线程安全的。

// 将当前线程对应的节点入队，插入队尾private Node addConditionWaiter() {

Node t = lastWaiter;

// 如果条件队列的最后一个节点取消了，将其清除出去

// 为什么这里把 waitStatus 不等于 Node.CONDITION，就判定为该节点发生了取消排队？

if (t != null && t.waitStatus != Node.CONDITION) {

// 这个方法会遍历整个条件队列，然后会将已取消的所有节点清除出队列

unlinkCancelledWaiters();

t = lastWaiter;

}

// node 在初始化的时候，指定 waitStatus 为 Node.CONDITION

Node node = new Node(Thread.currentThread(), Node.CONDITION);

// t 此时是 lastWaiter，队尾

// 如果队列为空

if (t == null)

firstWaiter = node;

else

t.nextWaiter = node;

lastWaiter = node;

return node;

}

上面的这块代码很简单，就是将当前线程进入到条件队列的队尾。

在addWaiter 方法中，有一个 unlinkCancelledWaiters() 方法，该方法用于清除队列中已经取消等待的节点。

当 await 的时候如果发生了取消操作（这点之后会说），或者是在节点入队的时候，发现最后一个节点是被取消的，会调用一次这个方法。

// 等待队列是一个单向链表，遍历链表将已经取消等待的节点清除出去// 纯属链表操作，很好理解，看不懂多看几遍就可以了private void unlinkCancelledWaiters() {

Node t = firstWaiter;

Node trail = null;

while (t != null) {

Node next = t.nextWaiter;

// 如果节点的状态不是 Node.CONDITION 的话，这个节点就是被取消的

if (t.waitStatus != Node.CONDITION) {

t.nextWaiter = null;

if (trail == null)

firstWaiter = next;

else

trail.nextWaiter = next;

if (next == null)

lastWaiter = trail;

}

else

trail = t;

t = next;

}

}

### **2. 完全释放独占锁**

回到 wait 方法，节点入队了以后，会调用 int savedState = fullyRelease(node); 方法释放锁，注意，这里是完全释放独占锁（fully release），因为 ReentrantLock 是可以重入的。

考虑一下这里的 savedState。如果在 condition1.await() 之前，假设线程先执行了 2 次 lock() 操作，那么 state 为 2，我们理解为该线程持有 2 把锁，这里 await() 方法必须将 state 设置为 0，然后再进入挂起状态，这样其他线程才能持有锁。当它被唤醒的时候，它需要重新持有 2 把锁，才能继续下去。

// 首先，我们要先观察到返回值 savedState 代表 release 之前的 state 值// 对于最简单的操作：先 lock.lock()，然后 condition1.await()。// 那么 state 经过这个方法由 1 变为 0，锁释放，此方法返回 1// 相应的，如果 lock 重入了 n 次，savedState == n// 如果这个方法失败，会将节点设置为"取消"状态，并抛出异常 IllegalMonitorStateExceptionfinal int fullyRelease(Node node) {

boolean failed = true;

try {

int savedState = getState();

// 这里使用了当前的 state 作为 release 的参数，也就是完全释放掉锁，将 state 置为 0

if (release(savedState)) {

failed = false;

return savedState;

} else {

throw new IllegalMonitorStateException();

}

} finally {

if (failed)

node.waitStatus = Node.CANCELLED;

}

}

考虑一下，如果一个线程在不持有 lock 的基础上，就去调用 condition1.await() 方法，它能进入条件队列，但是在上面的这个方法中，由于它不持有锁，release(savedState) 这个方法肯定要返回 false，进入到异常分支，然后进入 finally 块设置 node.waitStatus = Node.CANCELLED，这个已经入队的节点之后会被后继的节点”请出去“。

### **3. 等待进入阻塞队列**

释放掉锁以后，接下来是这段，这边会自旋，如果发现自己还没到阻塞队列，那么挂起，等待被转移到阻塞队列。

int interruptMode = 0;// 如果不在阻塞队列中，注意了，是阻塞队列while (!isOnSyncQueue(node)) {

// 线程挂起

LockSupport.park(this);

// 这里可以先不用看了，等看到它什么时候被 unpark 再说

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

isOnSyncQueue(Node node) 用于判断节点是否已经转移到阻塞队列了：

// 在节点入条件队列的时候，初始化时设置了 waitStatus = Node.CONDITION// 前面我提到，signal 的时候需要将节点从条件队列移到阻塞队列，// 这个方法就是判断 node 是否已经移动到阻塞队列了final boolean isOnSyncQueue(Node node) {

// 移动过去的时候，node 的 waitStatus 会置为 0，这个之后在说 signal 方法的时候会说到

// 如果 waitStatus 还是 Node.CONDITION，也就是 -2，那肯定就是还在条件队列中

// 如果 node 的前驱 prev 指向还是 null，说明肯定没有在 阻塞队列(prev是阻塞队列链表中使用的)

if (node.waitStatus == Node.CONDITION || node.prev == null)

return false;

// 如果 node 已经有后继节点 next 的时候，那肯定是在阻塞队列了

if (node.next != null)

return true;

// 下面这个方法从阻塞队列的队尾开始从后往前遍历找，如果找到相等的，说明在阻塞队列，否则就是不在阻塞队列

// 可以通过判断 node.prev() != null 来推断出 node 在阻塞队列吗？答案是：不能。

// 这个可以看上篇 AQS 的入队方法，首先设置的是 node.prev 指向 tail，

// 然后是 CAS 操作将自己设置为新的 tail，可是这次的 CAS 是可能失败的。

return findNodeFromTail(node);

}

// 从阻塞队列的队尾往前遍历，如果找到，返回 trueprivate boolean findNodeFromTail(Node node) {

Node t = tail;

for (;;) {

if (t == node)

return true;

if (t == null)

return false;

t = t.prev;

}

}

回到前面的循环，isOnSyncQueue(node) 返回 false 的话，那么进到 LockSupport.park(this); 这里线程挂起。

### **4. signal 唤醒线程，转移到阻塞队列**

为了大家理解，这里我们先看唤醒操作，因为刚刚到 LockSupport.park(this); 把线程挂起了，等待唤醒。

唤醒操作通常由另一个线程来操作，就像生产者-消费者模式中，如果线程因为等待消费而挂起，那么当生产者生产了一个东西后，会调用 signal 唤醒正在等待的线程来消费。

// 唤醒等待了最久的线程// 其实就是，将这个线程对应的 node 从条件队列转移到阻塞队列public final void signal() {

// 调用 signal 方法的线程必须持有当前的独占锁

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node first = firstWaiter;

if (first != null)

doSignal(first);

}

// 从条件队列队头往后遍历，找出第一个需要转移的 node// 因为前面我们说过，有些线程会取消排队，但是可能还在队列中private void doSignal(Node first) {

do {

// 将 firstWaiter 指向 first 节点后面的第一个，因为 first 节点马上要离开了

// 如果将 first 移除后，后面没有节点在等待了，那么需要将 lastWaiter 置为 null

if ( (firstWaiter = first.nextWaiter) == null)

lastWaiter = null;

// 因为 first 马上要被移到阻塞队列了，和条件队列的链接关系在这里断掉

first.nextWaiter = null;

} while (!transferForSignal(first) &&

(first = firstWaiter) != null);

// 这里 while 循环，如果 first 转移不成功，那么选择 first 后面的第一个节点进行转移，依此类推

}

// 将节点从条件队列转移到阻塞队列// true 代表成功转移// false 代表在 signal 之前，节点已经取消了final boolean transferForSignal(Node node) {

// CAS 如果失败，说明此 node 的 waitStatus 已不是 Node.CONDITION，说明节点已经取消，

// 既然已经取消，也就不需要转移了，方法返回，转移后面一个节点

// 否则，将 waitStatus 置为 0

if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))

return false;

// enq(node): 自旋进入阻塞队列的队尾

// 注意，这里的返回值 p 是 node 在阻塞队列的前驱节点

Node p = enq(node);

int ws = p.waitStatus;

// ws > 0 说明 node 在阻塞队列中的前驱节点取消了等待锁，直接唤醒 node 对应的线程。唤醒之后会怎么样，后面再解释

// 如果 ws <= 0, 那么 compareAndSetWaitStatus 将会被调用，上篇介绍的时候说过，节点入队后，需要把前驱节点的状态设为 Node.SIGNAL(-1)

if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))

// 如果前驱节点取消或者 CAS 失败，会进到这里唤醒线程，之后的操作看下一节

LockSupport.unpark(node.thread);

return true;

}

正常情况下，ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL) 这句中，ws <= 0，而且 compareAndSetWaitStatus(p, ws, Node.SIGNAL) 会返回 true，所以一般也不会进去 if 语句块中唤醒 node 对应的线程。然后这个方法返回 true，也就意味着 signal 方法结束了，节点进入了阻塞队列。

假设发生了阻塞队列中的前驱节点取消等待，或者 CAS 失败，只要唤醒线程，让其进到下一步即可。

### **5. 唤醒后检查中断状态**

上一步 signal 之后，我们的线程由条件队列转移到了阻塞队列，之后就准备获取锁了。只要重新获取到锁了以后，继续往下执行。

等线程从挂起中恢复过来，继续往下看

int interruptMode = 0;while (!isOnSyncQueue(node)) {

// 线程挂起

LockSupport.park(this);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

先解释下 interruptMode。interruptMode 可以取值为 REINTERRUPT（1），THROW\_IE（-1），0

* REINTERRUPT： 代表 await 返回的时候，需要重新设置中断状态
* THROW\_IE： 代表 await 返回的时候，需要抛出 InterruptedException 异常
* 0 ：说明在 await 期间，没有发生中断

有以下三种情况会让 LockSupport.park(this); 这句返回继续往下执行：

1. 常规路径。signal -> 转移节点到阻塞队列 -> 获取了锁（unpark）
2. 线程中断。在 park 的时候，另外一个线程对这个线程进行了中断
3. signal 的时候我们说过，转移以后的前驱节点取消了，或者对前驱节点的CAS操作失败了
4. 假唤醒。这个也是存在的，和 Object.wait() 类似，都有这个问题

线程唤醒后第一步是调用 checkInterruptWhileWaiting(node) 这个方法，此方法用于判断是否在线程挂起期间发生了中断，如果发生了中断，是 signal 调用之前中断的，还是 signal 之后发生的中断。

// 1. 如果在 signal 之前已经中断，返回 THROW\_IE// 2. 如果是 signal 之后中断，返回 REINTERRUPT// 3. 没有发生中断，返回 0private int checkInterruptWhileWaiting(Node node) {

return Thread.interrupted() ?

(transferAfterCancelledWait(node) ? THROW\_IE : REINTERRUPT) :

0;

}

Thread.interrupted()：如果当前线程已经处于中断状态，那么该方法返回 true，同时将中断状态重置为 false，所以，才有后续的 重新中断（REINTERRUPT） 的使用。

看看怎么判断是 signal 之前还是之后发生的中断：

// 只有线程处于中断状态，才会调用此方法// 如果需要的话，将这个已经取消等待的节点转移到阻塞队列// 返回 true：如果此线程在 signal 之前被取消，final boolean transferAfterCancelledWait(Node node) {

// 用 CAS 将节点状态设置为 0

// 如果这步 CAS 成功，说明是 signal 方法之前发生的中断，因为如果 signal 先发生的话，signal 中会将 waitStatus 设置为 0

if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {

// 将节点放入阻塞队列

// 这里我们看到，即使中断了，依然会转移到阻塞队列

enq(node);

return true;

}

// 到这里是因为 CAS 失败，肯定是因为 signal 方法已经将 waitStatus 设置为了 0

// signal 方法会将节点转移到阻塞队列，但是可能还没完成，这边自旋等待其完成

// 当然，这种事情还是比较少的吧：signal 调用之后，没完成转移之前，发生了中断

while (!isOnSyncQueue(node))

Thread.yield();

return false;

}

这里再说一遍，即使发生了中断，节点依然会转移到阻塞队列。

到这里，大家应该都知道这个 while 循环怎么退出了吧。要么中断，要么转移成功。

这里描绘了一个场景，本来有个线程，它是排在条件队列的后面的，但是因为它被中断了，那么它会被唤醒，然后它发现自己不是被 signal 的那个，但是它会自己主动去进入到阻塞队列。

### **6. 获取独占锁**

while 循环出来以后，下面是这段代码：

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

由于 while 出来后，我们确定节点已经进入了阻塞队列，准备获取锁。

这里的 acquireQueued(node, savedState) 的第一个参数 node 之前已经经过 enq(node) 进入了队列，参数 savedState 是之前释放锁前的 state，这个方法返回的时候，代表当前线程获取了锁，而且 state == savedState了。

注意，前面我们说过，不管有没有发生中断，都会进入到阻塞队列，而 acquireQueued(node, savedState) 的返回值就是代表线程是否被中断。如果返回 true，说明被中断了，而且 interruptMode != THROW\_IE，说明在 signal 之前就发生中断了，这里将 interruptMode 设置为 REINTERRUPT，用于待会重新中断。

继续往下：

if (node.nextWaiter != null) // clean up if cancelled

unlinkCancelledWaiters();if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

本着一丝不苟的精神，这边说说 node.nextWaiter != null 怎么满足。我前面也说了 signal 的时候会将节点转移到阻塞队列，有一步是 node.nextWaiter = null，将断开节点和条件队列的联系。

可是，在判断发生中断的情况下，是 signal 之前还是之后发生的？ 这部分的时候，我也介绍了，如果 signal 之前就中断了，也需要将节点进行转移到阻塞队列，这部分转移的时候，是没有设置 node.nextWaiter = null 的。

之前我们说过，如果有节点取消，也会调用 unlinkCancelledWaiters 这个方法，就是这里了。

### **7. 处理中断状态**

到这里，我们终于可以好好说下这个 interruptMode 干嘛用了。

* 0：什么都不做，没有被中断过；
* THROW\_IE：await 方法抛出 InterruptedException 异常，因为它代表在 await() 期间发生了中断；
* REINTERRUPT：重新中断当前线程，因为它代表 await() 期间没有被中断，而是 signal() 以后发生的中断

private void reportInterruptAfterWait(int interruptMode)

throws InterruptedException {

if (interruptMode == THROW\_IE)

throw new InterruptedException();

else if (interruptMode == REINTERRUPT)

selfInterrupt();

}

这个中断状态这部分内容，大家应该都理解了吧，不理解的话，多看几遍就是了。

### **\* 带超时机制的 await**

经过前面的 7 步，整个 ConditionObject 类基本上都分析完了，接下来简单分析下带超时机制的 await 方法。

public final long awaitNanos(long nanosTimeout)

throws InterruptedExceptionpublic final boolean awaitUntil(Date deadline)

throws InterruptedExceptionpublic final boolean await(long time, TimeUnit unit)

throws InterruptedException

这三个方法都差不多，我们就挑一个出来看看吧：

public final boolean await(long time, TimeUnit unit)

throws InterruptedException {

// 等待这么多纳秒

long nanosTimeout = unit.toNanos(time);

if (Thread.interrupted())

throw new InterruptedException();

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

// 当前时间 + 等待时长 = 过期时间

final long deadline = System.nanoTime() + nanosTimeout;

// 用于返回 await 是否超时

boolean timedout = false;

int interruptMode = 0;

while (!isOnSyncQueue(node)) {

// 时间到啦

if (nanosTimeout <= 0L) {

// 这里因为要 break 取消等待了。取消等待的话一定要调用 transferAfterCancelledWait(node) 这个方法

// 如果这个方法返回 true，在这个方法内，将节点转移到阻塞队列成功

// 返回 false 的话，说明 signal 已经发生，signal 方法将节点转移了。也就是说没有超时嘛

timedout = transferAfterCancelledWait(node);

break;

}

// spinForTimeoutThreshold 的值是 1000 纳秒，也就是 1 毫秒

// 也就是说，如果不到 1 毫秒了，那就不要选择 parkNanos 了，自旋的性能反而更好

if (nanosTimeout >= spinForTimeoutThreshold)

LockSupport.parkNanos(this, nanosTimeout);

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

// 得到剩余时间

nanosTimeout = deadline - System.nanoTime();

}

if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

if (node.nextWaiter != null)

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

return !timedout;

}

超时的思路还是很简单的，不带超时参数的 await 是 park，然后等待别人唤醒。而现在就是调用 parkNanos 方法来休眠指定的时间，醒来后判断是否 signal 调用了，调用了就是没有超时，否则就是超时了。超时的话，自己来进行转移到阻塞队列，然后抢锁。

### **\* 不抛出 InterruptedException 的 await**

关于 Condition 最后一小节了。

public final void awaitUninterruptibly() {

Node node = addConditionWaiter();

int savedState = fullyRelease(node);

boolean interrupted = false;

while (!isOnSyncQueue(node)) {

LockSupport.park(this);

if (Thread.interrupted())

interrupted = true;

}

if (acquireQueued(node, savedState) || interrupted)

selfInterrupt();

}

很简单，贴一下代码大家就都懂了，我就不废话了。

## **AbstractQueuedSynchronizer 独占锁的取消排队**

这篇文章说的是 AbstractQueuedSynchronizer，只不过好像 Condition 说太多了，赶紧把思路拉回来。

接下来，我想说说怎么取消对锁的竞争？

上篇文章提到过，最重要的方法是这个，我们要在这里面找答案：

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

if (failed)

cancelAcquire(node);

}

}

首先，到这个方法的时候，节点一定是入队成功的。

我把 parkAndCheckInterrupt() 代码贴过来：

private final boolean parkAndCheckInterrupt() {

LockSupport.park(this);

return Thread.interrupted();

}

这两段代码联系起来看，是不是就清楚了。

如果我们要取消一个线程的排队，我们需要在另外一个线程中对其进行中断。比如某线程调用 lock() 老久不返回，我想中断它。一旦对其进行中断，此线程会从 LockSupport.park(this); 中唤醒，然后 Thread.interrupted(); 返回 true。

我们发现一个问题，即使是中断唤醒了这个线程，也就只是设置了 interrupted = true 然后继续下一次循环。而且，由于 Thread.interrupted(); 会清除中断状态，第二次进 parkAndCheckInterrupt 的时候，返回会是 false。

所以，我们要看到，在这个方法中，interrupted 只是用来记录是否发生了中断，然后用于方法返回值，其他没有做任何相关事情。

所以，我们看外层方法怎么处理 acquireQueued 返回 false 的情况。

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}static void selfInterrupt() {

Thread.currentThread().interrupt();

}

所以说，lock() 方法处理中断的方法就是，你中断归中断，我抢锁还是照样抢锁，几乎没关系，只是我抢到锁了以后，设置线程的中断状态而已，也不抛出任何异常出来。调用者获取锁后，可以去检查是否发生过中断，也可以不理会。

来条分割线。有没有被骗的感觉，我说了一大堆，可是和取消没有任何关系啊。

我们来看 ReentrantLock 的另一个 lock 方法：

public void lockInterruptibly() throws InterruptedException {

sync.acquireInterruptibly(1);

}

方法上多了个 throws InterruptedException ，经过前面那么多知识的铺垫，这里我就不再啰里啰嗦了。

public final void acquireInterruptibly(int arg)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

if (!tryAcquire(arg))

doAcquireInterruptibly(arg);

}

继续往里：

private void doAcquireInterruptibly(int arg) throws InterruptedException {

final Node node = addWaiter(Node.EXCLUSIVE);

boolean failed = true;

try {

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

failed = false;

return;

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

// 就是这里了，一旦异常，马上结束这个方法，抛出异常。

// 这里不再只是标记这个方法的返回值代表中断状态

// 而是直接抛出异常，而且外层也不捕获，一直往外抛到 lockInterruptibly

throw new InterruptedException();

}

} finally {

// 如果通过 InterruptedException 异常出去，那么 failed 就是 true 了

if (failed)

cancelAcquire(node);

}

}

既然到这里了，顺便说说 cancelAcquire 这个方法吧：

private void cancelAcquire(Node node) {

// Ignore if node doesn't exist

if (node == null)

return;

node.thread = null;

// Skip cancelled predecessors

// 找一个合适的前驱。其实就是将它前面的队列中已经取消的节点都”请出去“

Node pred = node.prev;

while (pred.waitStatus > 0)

node.prev = pred = pred.prev;

// predNext is the apparent node to unsplice. CASes below will

// fail if not, in which case, we lost race vs another cancel

// or signal, so no further action is necessary.

Node predNext = pred.next;

// Can use unconditional write instead of CAS here.

// After this atomic step, other Nodes can skip past us.

// Before, we are free of interference from other threads.

node.waitStatus = Node.CANCELLED;

// If we are the tail, remove ourselves.

if (node == tail && compareAndSetTail(node, pred)) {

compareAndSetNext(pred, predNext, null);

} else {

// If successor needs signal, try to set pred's next-link

// so it will get one. Otherwise wake it up to propagate.

int ws;

if (pred != head &&

((ws = pred.waitStatus) == Node.SIGNAL ||

(ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &&

pred.thread != null) {

Node next = node.next;

if (next != null && next.waitStatus <= 0)

compareAndSetNext(pred, predNext, next);

} else {

unparkSuccessor(node);

}

node.next = node; // help GC

}

}

其实这个方法没什么好说的，一行行看下去就是了，节点取消，只要把 waitStatus 设置为 Node.CANCELLED，会有非常多的情况被从阻塞队列中请出去，主动或被动。

## **再说 java 线程中断和 InterruptedException 异常**

在之前的文章中，我们接触了大量的中断，这边算是个总结吧。如果你完全熟悉中断了，没有必要再看这节，本节为新手而写。

### **线程中断**

首先，我们要明白，中断不是类似 linux 里面的命令 kill -9 pid，不是说我们中断某个线程，这个线程就停止运行了。中断代表线程状态，每个线程都关联了一个中断状态，是一个 true 或 false 的 boolean 值，初始值为 false。

Java 中的中断和操作系统的中断还不一样，这里就按照****状态****来理解吧，不要和操作系统的中断联系在一起

关于中断状态，我们需要重点关注 Thread 类中的以下几个方法：

// Thread 类中的实例方法，持有线程实例引用即可检测线程中断状态public boolean isInterrupted() {}

// Thread 中的静态方法，检测调用这个方法的线程是否已经中断// 注意：这个方法返回中断状态的同时，会将此线程的中断状态重置为 false// 所以，如果我们连续调用两次这个方法的话，第二次的返回值肯定就是 false 了public static boolean interrupted() {}

// Thread 类中的实例方法，用于设置一个线程的中断状态为 truepublic void interrupt() {}

我们说中断一个线程，其实就是设置了线程的 interrupted status 为 true，至于说被中断的线程怎么处理这个状态，那是那个线程自己的事。如以下代码：

while (!Thread.interrupted()) {

doWork();

System.out.println("我做完一件事了，准备做下一件，如果没有其他线程中断我的话");

}

这种代码就是会响应中断的，它会在干活的时候先判断下中断状态，不过，除了 JDK 源码外，其他用中断的场景还是比较少的，毕竟 JDK 源码非常讲究。

当然，中断除了是线程状态外，还有其他含义，否则也不需要专门搞一个这个概念出来了。

如果线程处于以下三种情况，那么当线程被中断的时候，能自动感知到：

来自 Object 类的 wait()、wait(long)、wait(long, int)，

来自 Thread 类的 join()、join(long)、join(long, int)、sleep(long)、sleep(long, int)

这几个方法的相同之处是，方法上都有: throws InterruptedException

如果线程阻塞在这些方法上（我们知道，这些方法会让当前线程阻塞），这个时候如果其他线程对这个线程进行了中断，那么这个线程会从这些方法中立即返回，抛出 InterruptedException 异常，同时重置中断状态为 false。

实现了 InterruptibleChannel 接口的类中的一些 I/O 阻塞操作，如 DatagramChannel 中的 connect 方法和 receive 方法等

如果线程阻塞在这里，中断线程会导致这些方法抛出 ClosedByInterruptException 并重置中断状态。

Selector 中的 select 方法，参考下我写的 NIO 的文章

一旦中断，方法立即返回

对于以上 3 种情况是最特殊的，因为他们能自动感知到中断（这里说自动，当然也是基于底层实现），****并且在做出相应的操作后都会重置中断状态为 false****。

那是不是只有以上 3 种方法能自动感知到中断呢？不是的，如果线程阻塞在 LockSupport.park(Object obj) 方法，也叫挂起，这个时候的中断也会导致线程唤醒，但是唤醒后不会重置中断状态，所以唤醒后去检测中断状态将是 true。

### **InterruptedException 概述**

它是一个特殊的异常，不是说 JVM 对其有特殊的处理，而是它的使用场景比较特殊。通常，我们可以看到，像 Object 中的 wait() 方法，ReentrantLock 中的 lockInterruptibly() 方法，Thread 中的 sleep() 方法等等，这些方法都带有 throws InterruptedException，我们通常称这些方法为阻塞方法（blocking method）。

阻塞方法一个很明显的特征是，它们需要花费比较长的时间（不是绝对的，只是说明时间不可控），还有它们的方法结束返回往往依赖于外部条件，如 wait 方法依赖于其他线程的 notify，lock 方法依赖于其他线程的 unlock等等。

当我们看到方法上带有 throws InterruptedException 时，我们就要知道，这个方法应该是阻塞方法，我们如果希望它能早点返回的话，我们往往可以通过中断来实现。

除了几个特殊类（如 Object，Thread等）外，感知中断并提前返回是通过轮询中断状态来实现的。我们自己需要写可中断的方法的时候，就是通过在合适的时机（通常在循环的开始处）去判断线程的中断状态，然后做相应的操作（通常是方法直接返回或者抛出异常）。当然，我们也要看到，如果我们一次循环花的时间比较长的话，那么就需要比较长的时间才能****感知****到线程中断了。

### **处理中断**

一旦中断发生，我们接收到了这个信息，然后怎么去处理中断呢？本小节将简单分析这个问题。

我们经常会这么写代码：

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

// ignore

}// go on

当 sleep 结束继续往下执行的时候，我们往往都不知道这块代码是真的 sleep 了 10 秒，还是只休眠了 1 秒就被中断了。这个代码的问题在于，我们将这个异常信息吞掉了。（对于 sleep 方法，我相信大部分情况下，我们都不在意是否是中断了，这里是举例）

AQS 的做法很值得我们借鉴，我们知道 ReentrantLock 有两种 lock 方法：

public void lock() {

sync.lock();

}

public void lockInterruptibly() throws InterruptedException {

sync.acquireInterruptibly(1);

}

前面我们提到过，lock() 方法不响应中断。如果 thread1 调用了 lock() 方法，过了很久还没抢到锁，这个时候 thread2 对其进行了中断，thread1 是不响应这个请求的，它会继续抢锁，当然它不会把“被中断”这个信息扔掉。我们可以看以下代码：

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

// 我们看到，这里也没做任何特殊处理，就是记录下来中断状态。

// 这样，如果外层方法需要去检测的时候，至少我们没有把这个信息丢了

selfInterrupt();// Thread.currentThread().interrupt();

}

而对于 lockInterruptibly() 方法，因为其方法上面有 throws InterruptedException ，这个信号告诉我们，如果我们要取消线程抢锁，直接中断这个线程即可，它会立即返回，抛出 InterruptedException 异常。

在并发包中，有非常多的这种处理中断的例子，提供两个方法，分别为响应中断和不响应中断，对于不响应中断的方法，记录中断而不是丢失这个信息。如 Condition 中的两个方法就是这样的：

void await() throws InterruptedException;void awaitUninterruptibly();

通常，如果方法会抛出 InterruptedException 异常，往往方法体的第一句就是：

public final void await() throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

......

}

熟练使用中断，对于我们写出优雅的代码是有帮助的，也有助于我们分析别人的源码。

## **总结**

这篇文章的信息量真的很大，如果你花了时间，还是没有看懂，那是我的错了。

欢迎大家向我提问，我不一定能每次都及时出现，我出现也不一定能解决大家的问题，欢迎探讨。