目录

31 AbstractQueuedSynchronizer 源码解析（下）

更新时间：2019-11-08 11:04:36

低头要有勇气，抬头要有底气。

——韩寒

引导语

AQS 的内容太多，所以我们分成了两个章节，没有看过 AQS 上半章节的同学可以回首看一下哈，上半章节里面说了很多锁的基本概念，基本属性，如何获得锁等等，本章我们主要聊下如何释放锁和同步队列两大部分。

1 释放锁

释放锁的触发时机就是我们常用的 Lock.unLock () 方法，目的就是让线程释放对资源的访问权（流程见整体架构图紫色路线）。

释放锁也是分为两类，一类是排它锁的释放，一类是共享锁的释放，我们分别来看下。

1.1 释放排它锁 release

排它锁的释放就比较简单了，从队头开始，找它的下一个节点，如果下一个节点是空的，就会从尾开始，一直找到状态不是取消的节点，然后释放该节点，源码如下：

```
// unlock 的基础方法
public final boolean release(int arg) {
    // tryRelease 交给实现类去实现，一般就是用当前同步器状态减去 arg，如果返回 true 说明成功释
    if (tryRelease(arg)) {
        Node h = head;
        // 头节点不为空，并且非初始化状态
        if (h != null && h.waitStatus != 0)
            // 从头开始唤醒等待锁的节点
```

```
    return false;
}

// 很有意思的方法，当线程释放锁成功后，从 node 开始唤醒同步队列中的节点
// 通过唤醒机制,保证线程不会一直在同步队列中阻塞等待
private void unparkSuccessor(Node node) {
    // node 节点是当前释放锁的节点，也是同步队列的头节点
    int ws = node.waitStatus;
    // 如果节点已经被取消了，把节点的状态置为初始化
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);

    // 拿出 node 节点的后面一个节点
    Node s = node.next;
    // s 为空，表示 node 的后一个节点为空
    // s.waitStatus 大于0，代表 s 节点已经被取消了
    // 遇到以上这两种情况，就从队尾开始，向前遍历，找到第一个 waitStatus 字段不是被取消的
    if (s == null || s.waitStatus > 0) {
        s = null;
        // 这里从尾迭代，而不是从头开始迭代是有原因的。
        // 主要是因为节点被阻塞的时候，是在 acquireQueued 方法里面被阻塞的，唤醒时也一定会在
        for (Node t = tail; t != null && t != node; t = t.prev)
            // t.waitStatus <= 0 说明 t 没有被取消，肯定还在等待被唤醒
            if (t.waitStatus <= 0)
                s = t;
    }
    // 唤醒以上代码找到的线程
    if (s != null)
        LockSupport.unpark(s.thread);
}
```


1.2 释放共享锁 releaseShared

释放共享锁的方法是 releaseShared，主要分成两步：

1. tryReleaseShared 尝试释放当前共享锁，失败返回 false，成功走 2；
2. 唤醒当前节点的后续阻塞节点，这个方法我们之前看过了，线程在获得共享锁的时候，就会去唤醒其后面的节点，方法名称为：doReleaseShared。

我们一起来看下 releaseShared 的源码：

```
// 共享模式下，释放当前线程的共享锁
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        // 这个方法就是线程在获得锁时，唤醒后续节点时调用的方法
        doReleaseShared();
        return true;
    }
    return false;
}
```

2 条件队列的重要方法

在看条件队列的方法之前，我们先得弄明白为什么有了同步队列，还需要条件队列？

目录

步队列里面排队阻塞；获得锁的多个线程在碰到队列满或者空的时候，可以使用 Condition 来管理这些线程，让这些线程阻塞等待，然后在合适的时机后，被正常唤醒。

同步队列 + 条件队列联手使用的场景，最多被使用到锁 + 队列的场景中。

所以说条件队列也是不可或缺的一环。

接下来我们来看一下条件队列一些比较重要的方法，以下方法都在 ConditionObject 内部类中。

2.1 入队列等待 await

获得锁的线程，如果在碰到队列满或空的时候，就会阻塞住，这个阻塞就是用条件队列实现的，这个动作我们叫做入条件队列，方法名称为 await，流程见整体架构图中深绿色箭头流向，我们一起来看下 await 的源码：

```
// 线程入条件队列
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    // 加入到条件队列的队尾
    Node node = addConditionWaiter();
    // 标记位置 A
    // 加入条件队列后，会释放 lock 时申请的资源，唤醒同步队列队列头的节点
    // 自己马上就要阻塞了，必须马上释放之前 lock 的资源，不然自己不被唤醒的话，别的线程永远得
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    // 确认node不在同步队列上，再阻塞，如果 node 在同步队列上，是不能够上锁的
    // 目前想到的只有两种可能：
    // 1:node 刚被加入到条件队列中，立马就被其他线程 signal 转移到同步队列中去了
    // 2:线程之前在条件队列中沉睡，被唤醒后加入到同步队列中去
    while (!isOnSyncQueue(node)) {
        // this = AbstractQueuedSynchronizer$ConditionObject
        // 阻塞在条件队列上
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    // 标记位置 B
    // 其他线程通过 signal 已经把 node 从条件队列中转移到同步队列中的数据结构中去了
    // 所以这里节点苏醒了，直接尝试 acquireQueued
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        // 如果状态不是CONDITION，就会自动删除
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```


await 方法有几点需要特别注意：

1. 上述代码标记位置 A 处，节点在准备进入条件队列之前，一定会先释放当前持有的锁，不然自己进去条件队列了，其余的线程都无法获得锁了；

acquireQueued 方法；

3. Node 在条件队列中的命名，源码喜欢用 Waiter 来命名，所以我们在条件队列中看到 Waiter，其实就是 Node。

await 方法中有两个重要方法：addConditionWaiter 和 unlinkCancelledWaiters，我们一一看下。

2.1.1 addConditionWaiter

addConditionWaiter 方法主要是把节点放到条件队列中，方法源码如下：

```
// 增加新的 waiter 到队列中，返回新添加的 waiter
// 如果尾节点状态不是 CONDITION 状态，删除条件队列中所有状态不是 CONDITION 的节点
// 如果队列为空，新增节点作为队列头节点，否则追加到尾节点上
private Node addConditionWaiter() {
    Node t = lastWaiter;
    // If lastWaiter is cancelled, clean out.
    // 如果尾部的 waiter 不是 CONDITION 状态了，删除
    if (t != null && t.waitStatus != Node.CONDITION) {
        unlinkCancelledWaiters();
        t = lastWaiter;
    }
    // 新建条件队列 node
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    // 队列是空的，直接放到队列头
    if (t == null)
        firstWaiter = node;
    // 队列不为空，直接到队列尾部
    else
        t.nextWaiter = node;
    lastWaiter = node;
    return node;
}
```


整体过程比较简单，就是追加到队列的尾部，其中有个重要方法叫做 unlinkCancelledWaiters，这个方法会删除掉条件队列中状态不是 CONDITION 的所有节点，我们来看下 unlinkCancelledWaiters 方法的源码，如下：

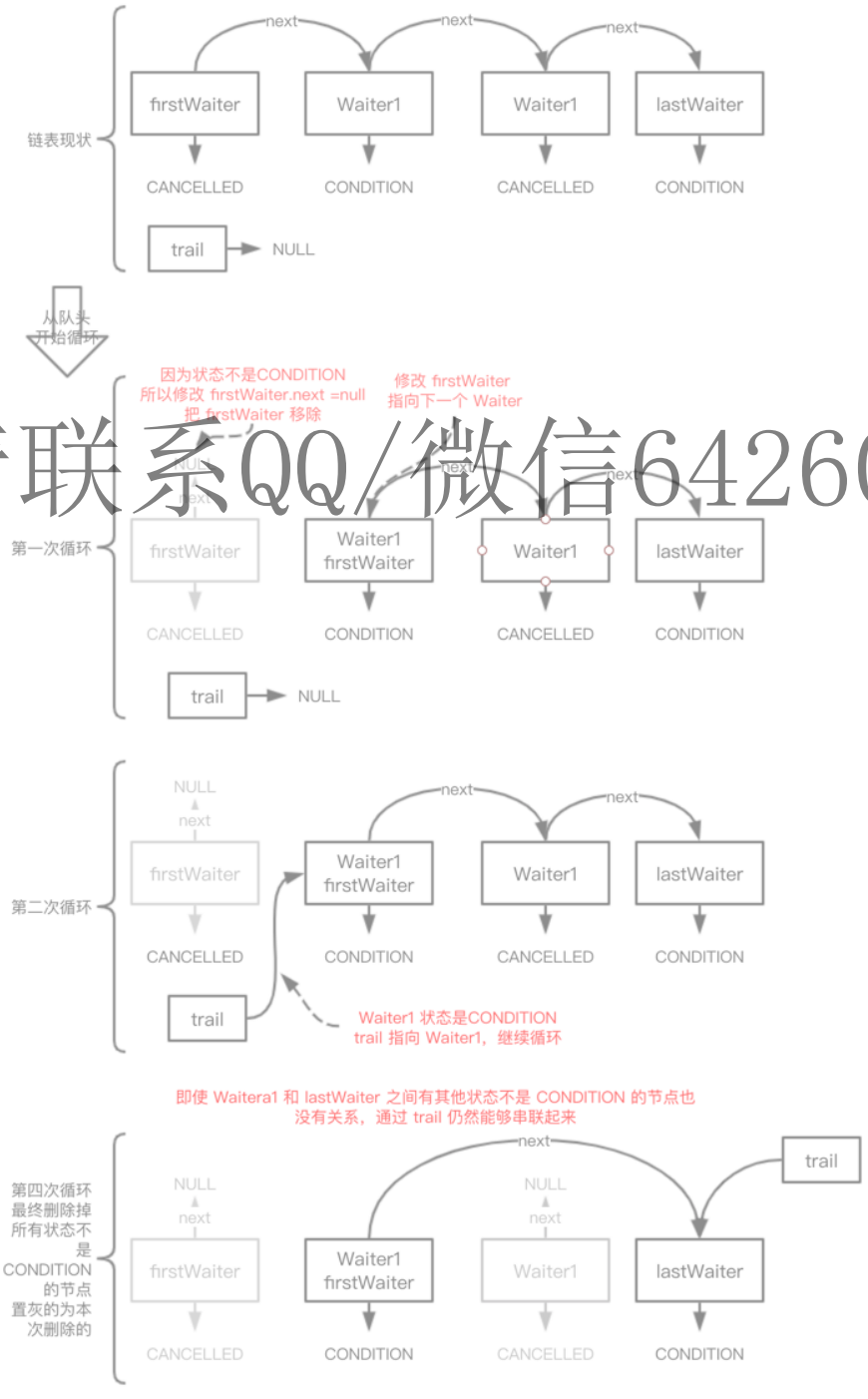
2.1.2 unlinkCancelledWaiters

```
// 会检查尾部的 waiter 是不是已经不是CONDITION状态了
// 如果不是，删除这些 waiter
private void unlinkCancelledWaiters() {
    Node t = firstWaiter;
    // trail 表示上一个状态,这个字段作用非常大，可以把状态都是 CONDITION 的 node 串联起来，
    Node trail = null;
    while (t != null) {
        Node next = t.nextWaiter;
        // 当前node的状态不是CONDITION，删除自己
        if (t.waitStatus != Node.CONDITION) {
            //删除当前node
            t.nextWaiter = null;
            // 如果 trail 是空的，咱们循环又是从头开始的，说明从头到当前节点的状态都不是 CONDIT
            // 都已经被删除了，所以移动队列头节点到当前节点的下一个节点
            if (trail == null)
                firstWaiter = next;
            // 如果找到上次状态是CONDITION的节点的话，先把当前节点删掉，然后把自己挂到上一个
```

目录

```
            if (next == null)
                lastWaiter = trail;
        }
        // 状态是 CONDITION 的 Node
        else
            trail = t;
        // 继续循环，循环顺序从头到尾
        t = next;
    }
}
```

为了方便大家理解这个方法，画了一个释义图，如下：



2.2 单个唤醒 signal

程，会从条件队列的头节点开始唤醒（流程见整体架构图中蓝色部分），源码如下：

```
// 唤醒阻塞在条件队列中的节点
public final void signal() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    // 从头节点开始唤醒
    Node first = firstWaiter;
    if (first != null)
        // doSignal 方法会把条件队列中的节点转移到同步队列中去
        doSignal(first);
}
// 把条件队列头节点转移到同步队列去
private void doSignal(Node first) {
    do {
        // nextWaiter为空，说明到队尾了
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        // 从队列头部开始唤醒，所以直接把头节点.next 置为 null，这种操作其实就是把 node 从条件
        // 这里有个重要的点是，每次唤醒都是从队列头部开始唤醒，所以把 next 置为 null 没有关系，
        first.nextWaiter = null;
        // transferForSignal 方法会把节点转移到同步队列中去
        // 通过 while 保证 transferForSignal 能成功
        // 等待队列的 node 不用管他，在 await 的时候，会自动清除状态不是 Condition 的节点(通过
        // (first = firstWaiter) != null  = true 的话，表示还可以继续循环， = false 说明队列中的元素
    } while (!transferForSignal(first) &&
             (first = firstWaiter) != null);
}
```


我们来看下最关键的方法：transferForSignal。

```
// 返回 true 表示转移成功， false 失败
// 大概思路：
// 1. node 追加到同步队列的队尾
// 2. 将 node 的前一个节点状态置为 SIGNAL，成功直接返回，失败直接唤醒
// 可以看出来 node 的状态此时是 0 了
final boolean transferForSignal(Node node) {
    /*
     * If cannot change waitStatus, the node has been cancelled.
     */
    // 将 node 的状态从 CONDITION 修改成初始化，失败返回 false
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        return false;

    // 当前队列加入到同步队列，返回的 p 是 node 在同步队列中的前一个节点
    // 看命名是 p，实际是 pre 单词的缩写
    Node p = enq(node);
    int ws = p.waitStatus;
    // 状态修改成 SIGNAL，如果成功直接返回
    // 把当前节点的前一个节点修改成 SIGNAL 的原因，是因为 SIGNAL 本身就表示当前节点后面的节
    if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
        // 如果 p 节点被取消，或者状态不能修改成SIGNAL，直接唤醒
        LockSupport.unpark(node.thread);
    return true;
}
```

整个源码下来，我们可以看到，唤醒条件队列中的节点，实际上就是把条件队列中的节点转移到同步队列中，并把其前置节点状态置为 SIGNAL。

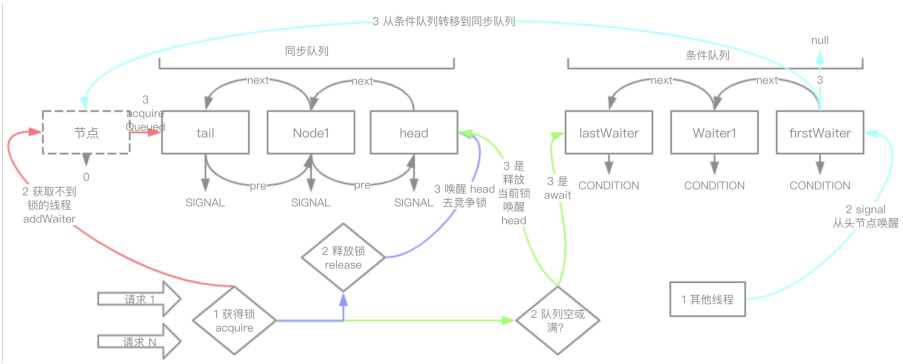
signalAll 的作用是唤醒条件队列中的全部节点，源码如下：

```
public final void signalAll() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    // 拿到头节点
    Node first = firstWaiter;
    if (first != null)
        // 从头节点开始唤醒条件队列中所有的节点
        doSignalAll(first);
}
// 把条件队列所有节点依次转移到同步队列去
private void doSignalAll(Node first) {
    lastWaiter = firstWaiter = null;
    do {
        // 拿出条件队列队列头节点的下一个节点
        Node next = first.nextWaiter;
        // 把头节点从条件队列中删除
        first.nextWaiter = null;
        // 头节点转移到同步队列中去
        transferForSignal(first);
        // 开始循环头节点的下一个节点
        first = next;
    } while (first != null);
}
```

从源码中可以看出，其本质就是 for 循环调用 transferForSignal 方法，将条件队列中的节点循环转移到同步队列中去。

3 总结

AQS 源码终于说完了，你都懂了么，可以在默默回忆一下 AQS 架构图，看看这张图现在能不能看懂了。



30 AbstractQueuedSynchronizer 源码解析（上）

32 ReentrantLock 源码解析

精选留言 2

欢迎在这里发表留言，作者筛选后可公开显示

目录

同步协作。

0 回复 4天前

敲木鱼的小和尚

// 如果节点已经被取消了，把节点的状态置为初始化 if (ws < 0) compareAndSetWaitStatus(node, ws, 0);这个解释应该是如果节点处于SIGNAL状态，将节点设置为初始状态吧，但是感觉有说不通，希望老师解答一下

0 回复 2019-12-03

胖子胖 回复 **敲木鱼的小和尚**

对啊，大于0才是CANCELLED，注释上写着【If status is negative (i.e., possibly needing signal) try to clear in anticipation of signalling.】感觉这块就是因为SIGNAL的后续节点需要被唤醒，先把signal节点状态清空变成0

回复 3天前

千学不如一看，千看不如一练