AQS(AbstractQueuedSynchronizer) 源码分析

说明

1. 本文基于 jdk 8 写作。
2. @author JellyfishMIX - github / blog.jellyfishmix.com
3. LICENSE GPL-2.0

AQS 需要关注的点有哪些?

- AQS 全称 AbstractQueuedSynchronizer，是 juc 包(java.util.concurrent)中一个同步器开发框架，用于支持上层的锁。
- 关键的属性：state 同步状态，CHL 同步队列。
- 两种模式：独占模式，共享模式。

AQS 的关键属性

1. sate 同步状态，是锁的表述，表示有多少线程获取了锁。
2. CHL 队列（同步队列），由链表实现，以 CAS + 自旋的方式获取资源，是可阻塞的先进先出的双向队列。通过 CAS + 自旋保证节点插入和移除的原子性。当有线程获取锁失败，就被添加到队列末尾。当有线程释放了锁，会从 CHL 队头出队一个线程。

state 同步状态

```
/**
 * The synchronization state.
 */
private volatile int state;
```

AQS 锁支持的子类，可以借助操作同步状态 state 来实现尝试获取同步状态(tryAcquire)、尝试释放同步状态(tryRelease)的逻辑。

也就是说，AQS 的子类(各种锁)判断何时能获取同步状态，何时能释放同步状态，需要借助于判断 state 的值。然后能把获取同步状态进一步封装成获取锁(lock)，把释放同步状态封装成释放锁(unlock)。至于 state 为何值时可以获取同步状态，state 为何值时可以释放同步状态，这些都是交由子类判断的。

AQS 只是当子类判断为未获取到同步状态时，通过 CHL 入队操作，把线程加入到同步队列中，然后让线程休眠。当子类判断为可以释放同步状态时，从 CHL 同步队列中唤醒一个线程。

CHL 队列

在分析前，可能会有如下疑惑：

1. 节点的数据结构是什么样的?
2. 是单向还是双向?
3. 有无头节点和尾节点?

Node

CHL 是链表，链表是由节点 node 组成的。

```
    static final class Node {
        /**
         * 标记一个节点正在共享模式下等待
         */
        static final Node SHARED = new Node();
        /**
         * 标记一个节点正在独占模式下等待
         */
        static final Node EXCLUSIVE = null;

        /**
         * waitStatus value, 标记当前节点的线程被 cancel 取消
         */
        static final int CANCELLED =  1;
        /**
         * waitStatus value, 标记后继节点的线程需要被唤醒
         */
        static final int SIGNAL    = -1;
        /**
         * waitStatus value, 标记当前节点的线程进入等待队列中
         */
        static final int CONDITION = -2;
        /**
         * waitStatus value, 表示下一次共享式的同步状态获取将会无条件传播下去
         */
        static final int PROPAGATE = -3;

        /**
         * 节点的状态
         */
        volatile int waitStatus;

        /**
         * 当前节点的前驱节点
         */
        volatile Node prev;
```

```
        /**
         * 当前节点的后继节点
         */
        volatile Node next;

        /**
         * 当前节点引用的线程
         */
        volatile Thread thread;

        /**
         * 等待队列中的下一个节点
         */
        Node nextWaiter;
    }
```

注意到属性 prev, next, 说明 CHL 是一个双向队列。

AQS 中有两个属性，用作头尾指针，通过头尾指针来管理同步队列。头尾指针使用场景包括获取锁失败的线程进行入队，释放锁时对同步队列中的线程进行通知等核心方法。

```
    /**
     * 指向 CHL 队列的头节点
     */
    private transient volatile Node head;

    /**
     * 指向 CHL 队列的尾节点
     */
    private transient volatile Node tail;
```

CHL 队列中 node 的出队和入队操作，实际上对应着同步状态的获取和释放两个操作：获取同步状态失败 -> 进行入队操作，获取同步状态成功 -> 进行出队操作。

独占模式与共享模式

1. AQS 的工作模式分为独占模式(EXCLUSIVE)和共享模式(SHARED)，记录在 node 的信息中。
2. 独占模式：同一时间只有一个线程能拿到锁执行，锁的 state 只有0和1两种情况（ReentrantLock 的可重入锁是个例外）。典型的实现如 ReentrantLock。
3. 共享模式：同一时间有多个线程可以拿到锁执行，锁的 state 大于或等于0。典型的实现如 CountDownLunch。
4. 它的实现类(子类)中，要么实现并使用了它独占功能的 API，要么使用了共享功能的 API，而不会同时使用两套 API。即便是它有名的子类 ReentrantReadWriteLock，也是通过两个内部类：读锁和写锁，分别实现的两套 API 来实现的。
5. AQS 使用了模板方法设计模式，定义一套操作逻辑，相当于一个算法的骨架，而将一些步骤的实现延迟到子类中，比如获取、释放 state 同步状态。

独占模式

以下是独占模式的方法

```
/* 独占式获取同步状态，如果获取失败则插入同步队列进行等待 */
void acquire(int arg);
/* 与 acquire 方法相同，但在同步队列中进行等待的时候可以检测中断 */
void acquireInterruptibly(int arg);
/* 在 acquireInterruptibly 基础上增加了超时等待功能，在超时时间内没有获得同步状态返回 false */
boolean tryAcquireNanos(int arg, long nanosTimeout);
/* 释放同步状态，该方法会唤醒在同步队列中的下一个节点 */
boolean release(int arg);
```

acquire 方法（独占模式获取同步状态）

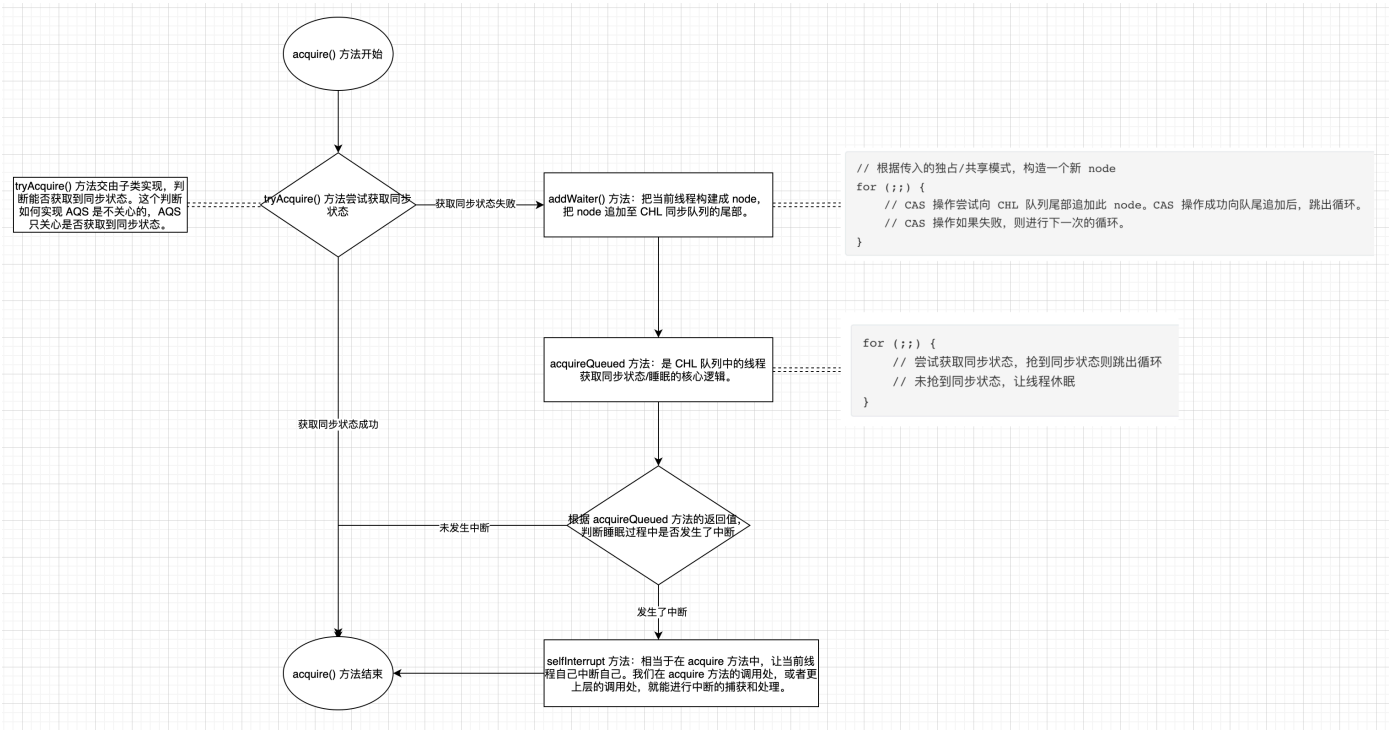
```
    /**
     * Acquires in exclusive mode, ignoring interrupts.  Implemented
     * by invoking at least once {@link #tryAcquire},
     * returning on success.  Otherwise the thread is queued, possibly
     * repeatedly blocking and unblocking, invoking {@link
     * #tryAcquire} until success.  This method can be used
     * to implement method {@link Lock#lock}.
     *
     * 模板方法，封装了线程获取资源失败后，进入同步队列并阻塞的逻辑
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     */
    public final void acquire(int arg) {
        // tryAcquire 方法由子类实现，尝试获取同步状态
        if (!tryAcquire(arg) &&
            // 未获取到同步状态，则进入同步队列
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            // 如果在同步队列中发生中断，则中断标记为 true，执行 selfInterrupt() 方法
            selfInterrupt();
    }
```

我们把 acquire 方法概括一下：

```
// 交由子类判断是否能成功获取到同步状态
if (!tryAcquire(arg) &&
            // 未获取到同步状态，则调用 addWaiter 方法把线程构建成 node，追加到 CHL 同步队列尾部。
            // 追加至 CHL 同步队列后，调用 acquireQueued 方法让线程休眠，此方法也包含了线程苏醒后重新抢同步状态的逻辑。
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            // 如果在休眠过程中发生中断，则中断标记为 true，执行 selfInterrupt() 方法
            selfInterrupt();
```

1. 先调用 tryAcquire 方法（由子类实现，获取同步状态），判断是否获取成功。
2. 如果获取成功，acquire 方法返回结束。如果获取失败，调用 addWaiter 方法，把当前线程构建成 node，把 node 追加至 CHL 同步队列的尾部。
3. node 追加至 CHL 队列尾部后，调用 acquireQueued 方法，把当前线程休眠。同时 acquireQueued 里面也包括了线程休眠被唤醒后尝试抢锁的逻辑。

acquire 方法流程图：



addWaiter 方法

```
    /**
     * Creates and enqueues node for current thread and given mode.
     *
     * 根据传入的独占/共享模式，在 CHL 队列中添加一个 node
     *
     * @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared
     * @return the new node
     */
    private Node addWaiter(Node mode) {
        // 根据传入的独占/共享模式，构造一个新 node
        Node node = new Node(Thread.currentThread(), mode);
```

```
        // Try the fast path of enq; backup to full enq on failure
        /*
         * 下面这个 if 分支里在做的事情，和 enq 方法中在做的事情相同。可以理解为，把 enq 中的部分代码放在了外面。
         * 这种做法在并发编程中还挺常见，把最有可能成功执行的代码直接写在最常用的调用处。
         * 在线程数不多的情况下，CAS 还是很难失败的，因此这种写法可以节省多条指令。比如节省了方法调用、enq 方法中进入循环的开销。
         * 是 jdk 中一种极致的优化，业务代码可以不用做到这么极致。
         */
        Node pred = tail;
        // 如果队尾不为空（即当前队列中有元素），则尝试在队尾追加 node
        if (pred != null) {
            /*
             * node.prev = pred 这里其实是有线程安全问题的，并发情况下可能有多个刚构造出来的 node.prev 被设置为了 pred。
             * 但是没关系，后面 CAS 设置队尾的时候会失败，然后进入 enq 方法中循环去做。当某次 CAS 成功时，说明 CAS 前的那步 node.prev 赋值也正确了。
             * 所以，这里只是局部存在线程安全问题，最终结果是安全的。
             * 为什么设置尾节点时都要先将之前的尾节点设置为 node.prev 的值呢，而不是在 CAS 之后再设置?
             * 因为如果不提前设置 node.prev 的话，在 CAS 设置完 tail 后会存在一瞬间的 tail.pre 为 null 的情况，而 Doug Lea 正是考虑到这种情况，不论何时获取 tail.pre 都不会为 null
             */
            node.prev = pred;
            // CAS 尝试设置尾节点为当前节点
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        // 尝试循环在队尾追加 node
        enq(node);
        return node;
    }
```

enq 方法

```
    /**
     * Inserts node into queue, initializing if necessary. See picture above.
     *
     * 尝试循环在队尾追加 node
     *
     * @param node the node to insert
     * @return node's predecessor
     */
    private Node enq(final Node node) {
        for (;;) {
            Node t = tail;
            // 如果 tail 为空，说明队列中还没有元素，需要初始化一个头节点
```

```
            if (t == null) { // Must initialize
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else {
                /*
                 * node.prev = t 这里其实是有线程安全问题的，并发情况下可能有多个刚构造出来的
node.prev 被设置为了 t。
                 * 但是没关系，下一步 CAS 设置队尾的时候会失败，然后进入下一次循环。当某次 CAS 成功
时，说明 CAS 前的那步 node.prev 赋值也正确了。
                 * 所以，这里只是局部存在线程安全问题，最终结果是安全的。
                 * 为什么设置尾节点时都要先将之前的尾节点设置为 node.prev 的值呢，而不是在 CAS 之
后再设置?
                 * 因为如果不提前设置 node.prev 的话，在 CAS 设置完 tail 后会存在一瞬间的
tail.pre 为 null 的情况，而 Doug Lea 正是考虑到这种情况，不论何时获取 tail.pre 都不会为 null
                 */
                node.prev = t;
                // CAS 尝试设置尾节点为当前节点
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
```

addWaiter 方法的逻辑：把当前线程构建成 node，把 node 追加至 CHL 同步队列的尾部。粗略地理解，我们可以把 enq 方法合并至 addWaiter 方法中，enq 方法在承担 addWaiter 方法的一部分逻辑。我们把主要逻辑概括一下：

```
// 根据传入的独占/共享模式，构造一个新 node
for (;;) {
    // CAS 操作尝试向 CHL 队列尾部追加此 node。CAS 操作成功向队尾追加后，跳出循环。
    // CAS 操作如果失败，则进行下一次的循环。
}
```

我们注意到，addWaiter 方法 和 enq 方法中，有部分代码做的事情相同。可以理解为，把 enq 中的部分代码放在了外面。这种做法在并发编程中还挺常见，把最有可能成功执行的代码直接写在最常用的调用处。在线程数不多的情况下，CAS 还是很难失败的，因此这种写法可以节省多条指令。比如节省了方法调用、enq 方法中进入循环的开销。这种写法是 jdk 中一种极致的优化，业务代码可以不用做到这么极致。

我们还可以注意到 node.prev = t; 放在了 if CAS 之前，这里其实是有线程安全问题的，并发情况下可能有多个刚构造出来的 node.prev 被设置为了 t。但是没关系，下一步 CAS 设置队尾的时候会失败，然后进入下一次循环。当某次 CAS 成功时，说明 CAS 前的那步 node.prev 赋值也正确了。所以，这里只是局部存在线程安全问题，最终结果是安全的。

为什么设置尾节点时都要先将之前的尾节点设置为 node.prev 的值呢，而不是在 CAS 之后再设置？因为如果不提前设置 node.prev 的话，在 CAS 设置完 tail 后会存在一瞬间的 tail.pre 为 null 的情况，而 Doug Lea 正是考虑到这种情况，不论何时获取 tail.pre 都不会为 null

acquireQueued 方法

我们回到 acquire 方法中，去看 acquireQueued 方法。

```
    /**
     * Acquires in exclusive uninterruptible mode for thread already in
     * queue. Used by condition wait methods as well as acquire.
     *
     * 在 CHL 队列中的线程获取同步状态的逻辑，是 CHL 队列中的线程获取同步状态/睡眠的核心逻辑
     *
     * @param node the node
     * @param arg the acquire argument
     * @return {@code true} if interrupted while waiting
     */
    final boolean acquireQueued(final Node node, int arg) {
        // 获取同步状态是否成功
        boolean failed = true;
        try {
            // 中断标记
            boolean interrupted = false;
            // 循环尝试获取同步状态
            for (;;) {
                // 当前节点的前驱节点
                final Node p = node.predecessor();
                // 判断当前节点的前驱节点是不是头节点，如果是则说明应该出队了，调用 tryAcquire 方法
(子类实现)，获取同步状态
                if (p == head && tryAcquire(arg)) {
                    // 同步状态获取成功，把当前节点设置为头节点
                    setHead(node);
                    // 把老的头节点释放掉（通过去除老头节点的引用关系，最终让 GC 去回收此对象）
                    p.next = null; // help GC
                    failed = false;
                    // 返回中断标记
                    return interrupted;
                }
                // 如果走到这一步，说明当前节点不是头节点，或未获取到同步状态，说明获取同步状态失败
(这种情况存在于其他线程已经抢先获取了同步状态）。判断是否需要休眠。
                if (shouldParkAfterFailedAcquire(p, node) &&
                    // 进入睡眠状态，判断在睡眠状态中是否被中断
                    parkAndCheckInterrupt())
                    // 如果在睡眠状态中被中断，则设置中断标记为 true
                    interrupted = true;
            }
        } finally {
            // 比如当前节点因为超时或响应了中断，需要取消自己
            if (failed)
                // 取消获取同步状态
                cancelAcquire(node);
        }
```

```
    }
```

acquireQueued 方法是 CHL 队列中的线程获取同步状态/睡眠的核心逻辑。我们把主要逻辑概括一下：

```
for (;;) {
    // 尝试获取同步状态，抢到同步状态则跳出循环
    // 未抢到同步状态，让线程休眠
}
```

在进入休眠前，尝试再获取一次同步状态，如果仍然失败，则会调用 shouldParkAfterFailedAcquire 方法，判断是否需要休眠。

shouldParkAfterFailedAcquire 方法

```
    /**
     * Checks and updates status for a node that failed to acquire.
     * Returns true if thread should block. This is the main signal
     * control in all acquire loops.  Requires that pred == node.prev.
     *
     * 检查未获取到同步状态的 node 的状态，并更新 status。
     * 如果线程应该阻塞，返回 true。这是在所有循环获取同步状态中，主要的信号控制。要求 pred ==
node.prev
     *
     * @param pred node's predecessor holding status
     * @param node the node
     * @return {@code true} if thread should block
     */
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        // 获取当前节点的前驱节点的状态
        int ws = pred.waitStatus;
        // 如果前驱节点的状态为 SIGNAL，表示前驱节点的后继节点（即当前节点）的线程需要被唤醒
        if (ws == Node.SIGNAL)
            /*
             * This node has already set status asking a release
             * to signal it, so it can safely park.
             *
             * 当前节点已经把前驱节点设置为了 SIGNAL，表示当前节点需要被唤醒，所以它可以安全地阻塞
（休眠）
             */
            return true;
        // 如果前驱节点的状态 > 0 （即 CANCELLED 状态）
        if (ws > 0) {
            /*
             * Predecessor was cancelled. Skip over predecessors and
             * indicate retry.
             *
             * 前驱节点被取消（比如前驱节点已经因为超时或响应了中断，而取消了自己），跳过前驱节点至下
一个前驱节点。
             */
```

```
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            // 新的前驱节点把后继节点修改为当前节点
            pred.next = node;
        } else {
            /*
             * waitStatus must be 0 or PROPAGATE.  Indicate that we
             * need a signal, but don't park yet.  Caller will need to
             * retry to make sure it cannot acquire before parking.
             *
             * 在休眠前需要给前驱节点设置一个标识，将 waitStatus 设置为 Node.SIGNAL(-1)
             */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

在 shouldParkAfterFailedAcquire 方法中，需要过滤无效的前驱节点（比如已经被取消的前驱节点），然后把有效的前驱节点设置为 SIGNAL，表示前驱节点的下一个节点（即当前节点）需要被唤醒，然后它可以安全地阻塞（休眠）。

在之前的 acquireQueued 方法中，调用 shouldParkAfterFailedAcquire 方法判断是否需要休眠。如果需要休眠，则会调用 parkAndCheckInterrupt 方法，进行休眠。

parkAndCheckInterrupt 方法

```
    /**
     * Convenience method to park and then check if interrupted
     *
     * 当前线程进入休眠状态，返回值表示休眠过程中是否发生了中断
     *
     * @return {@code true} if interrupted
     */
    private final boolean parkAndCheckInterrupt() {
        // 当前线程休眠
        LockSupport.park(this);
        // 重置线程的中断标志为 false，并返回一个 boolean 值表示线程的中断标记被重置前的值
(true/false)
        return Thread.interrupted();
    }
```

如果在休眠过程中发生中断，中断标记会被设置为 true，然后继续在 acquireQueued 的自旋 `for(;;)` 中尝试获取同步状态，获取同步状态后，acquireQueued 方法返回。这也说明了，即使线程在休眠过程中发生了中断，也只有在线程获取到同步状态后，才能响应中断。如果在休眠过程中没有发生中断，则中断标记依然是 false。

acquireQueued 的返回值是一个中断标记，表示在休眠过程中是否发生中断。我们再重新看一下 acquire 方法。

```
    /**
```

```
     * Acquires in exclusive mode, ignoring interrupts.  Implemented
     * by invoking at least once {@link #tryAcquire},
     * returning on success.  Otherwise the thread is queued, possibly
     * repeatedly blocking and unblocking, invoking {@link
     * #tryAcquire} until success.  This method can be used
     * to implement method {@link Lock#lock}.
     *
     * 模板方法，封装了线程获取资源失败后，进入同步队列并阻塞的逻辑
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     */
    public final void acquire(int arg) {
        // tryAcquire 方法由子类实现，尝试获取同步状态
        if (!tryAcquire(arg) &&
            // 未获取到同步状态，则进入同步队列
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            // 如果在同步队列中发生中断，则中断标记为 true，执行 selfInterrupt() 方法
            selfInterrupt();
    }
```

如果在同步队列中发生中断，则中断标记为 true，执行 selfInterrupt() 方法。

selfInterrupt() 方法

```
    /**
     * Convenience method to interrupt current thread.
     *
     * 当前线程中断自己
     */
    static void selfInterrupt() {
        Thread.currentThread().interrupt();
    }
```

相当于在 acquire 方法中，让当前线程自己中断自己。我们在 acquire 方法的调用处，或者更上层的调用处，就能进行中断的捕获和处理。

acquireQueued 方法的追问 -- 为什么 acquire 方法要靠 acquireQueued 方法的返回值判断，来调用 .interrupt() 方法

这里的追问探究的比较细致，如果不感兴趣直接跳过即可，对理解主要逻辑影响不大。

acquireQueued 方法会返回一个中断标记(acquireQueued 方法中的局部变量 interrupted，以下简称为中断标记 interrupted)，表示是否发生中断。acquireQueued 方法返回后，回到了 acquire 方法：

```
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

如果中断标记 interrupted 为 true，则会执行 selfInterrupt() 方法，也就是执行：

```
Thread.currentThread().interrupt();
```

为什么要做的这么麻烦呢，还要等 acquireQueued 方法返回之后，根据中断标记来做中断。我们直接在 acquireQueued 方法中去执行 `Thread.currentThread().interrupt();` 不行吗?

我们来回顾一下 acquireQueued 的源码，尝试在 1 号位置 和 2 号位置加 `Thread.currentThread().interrupt();`，看看能不能避免使用 interrupted 这个返回值。

```
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;
                    // 2 号位置
                    return interrupted;
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    // 1 号位置
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

1 号位置加 `Thread.currentThread().interrupt();` 不可以。如果 .interrupt() 方法在 1 号位置被调用了，当前 Thread 的中断标志会被被置为 true。如果下一次循环中，线程没有获取到同步状态，会走让线程休眠的逻辑。而让线程休眠的方法 parkAndCheckInterrupt 调用的是 LockSupport.park

```
    private final boolean parkAndCheckInterrupt() {
        // 当前线程休眠
        LockSupport.park(this);
        // 重置线程的中断标志为 false，并返回一个 boolean 值表示线程的中断标记被重置前的值
(true/false)
        return Thread.interrupted();
    }
```

LockSupport.park 方法会让线程进入休眠状态，但是当前 Thread 的中断标记为 true 时，LockSupport.park 会立刻返回，不会发生阻塞。这就很危险了，如果线程一直无法获取到同步状态，acquireQueued 方法中

```
for (;;) {
    // 尝试获取同步状态，抢到同步状态则跳出循环（方法返回）
    // 未抢到同步状态，让线程休眠
}
```

`for (;;)` 循环无法休眠，会一直空转，对 cpu 性能消耗会十分严重。

这也是为什么在让未获取到同步状态的线程休眠时，调用的方法 parkAndCheckInterrupt 中，在 LockSupport.park 方法结束后，会调用 Thread.interrupted() 方法重置线程的中断标志。

这里插入一下关于线程的中断标志(interrupt flag)的介绍：

> 什么是 Thread 的中断标志(interrupt flag)?
>
> 中断标志(interrupt flag) 是线程发生中断时，设置的内部属性。在 Thread.java 中是看不到这个 interrupt flag 属性的，因为它写在了 c++ 的代码中，底层设置中断标志的方法 private native void interrupt0() 是一个 native 方法。我们只需要知道，中断标志(interrupt flag) 用来表示线程是否发生了中断。
>
> 怎么设置中断标志?
>
> 要设置一个线程的中断标志，只需要简单的在线程对象上调用 thread.interrupt() 方法。
>
> 怎么清除中断标志?
>
> Thread.interrupted() 方法可以重置中断标志为 false，并返回一个 boolean 值表示中断标记被重置前的值(true/false)。

2 号位置加 `Thread.currentThread().interrupt();` 也不可以。我们看一下 AQS 中 acquireQueued 方法的调用处，可以看到有好几处地方用到了 acquireQueued 方法：

```
849     /**
850      * Acquires in exclusive uninterruptible mode for thread already in
851      * queue. Used by condition wait methods as well as acquire.
852      *
853      * @param node the node
854      * @param arg the acquire argument
855      * @return {@code true} if interrupted while waiting
856      */
857 @   final boolean acquireQueued(final Node node, int arg) {
858         boolean failed = tru
859         try {
860             boolean interrup
861             for (;;) {
862                 final Node p
863                 if (p == hea
864                     setHead(
865                     p.next =
866                     failed = false;
867                     return interrupted;
868                 }
869                 if (shouldParkAfterFailedAcquire(p, node) &&
870                     parkAndCheckInterrupt())
871                     interrupted = true;
872             }
873         } finally {
874             if (failed)
875                 cancelAcquire(node);
876         }
877     }
878
```

Usages of acquireQueued(Node, int) in All Places (6 usages found)

- AbstractQueuedSynchronizer.java 1199 **acquireQueued**(addWaiter(Node.EXCLUSIVE), arg))
- AbstractQueuedSynchronizer.java 1980 if (**acquireQueued**(node, savedState) || interrupted)
- AbstractQueuedSynchronizer.java 2043 if (**acquireQueued**(node, savedState) && interruptMode != THROW_IE)
- AbstractQueuedSynchronizer.java 2083 if (**acquireQueued**(node, savedState) && interruptMode != THROW_IE)
- AbstractQueuedSynchronizer.java 2124 if (**acquireQueued**(node, savedState) && interruptMode != THROW_IE)
- AbstractQueuedSynchronizer.java 2168 if (**acquireQueued**(node, savedState) && interruptMode != THROW_IE)

Press ⌥⌘F7 again to search in Project and Libraries

我们选一个调用处来看看：

```
2019         /**
2020          * Implements interruptible condition wait.
2021          * <ol>
2022          * <li> If current thread is interrupted, throw InterruptedException.
2023          * <li> Save lock state returned by {@link #getState}.
2024          * <li> Invoke {@link #release} with saved state as argument,
2025          *      throwing IllegalMonitorStateException if it fails.
2026          * <li> Block until signalled or interrupted.
2027          * <li> Reacquire by invoking specialized version of
2028          *      {@link #acquire} with saved state as argument.
2029          * <li> If interrupted while blocked in step 4, throw InterruptedException.
2030          * </ol>
2031          */
2032 I↑      public final void await() throws InterruptedException {
2033             if (Thread.interrupted())
2034                 throw new InterruptedException();
2035             Node node = addConditionWaiter();
2036             int savedState = fullyRelease(node);
2037             int interruptMode = 0;
2038             while (!isOnSyncQueue(node)) {
2039                 LockSupport.park( blocker: this);
2040                 if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
2041                     break;
2042             }
2043             if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
2044                 interruptMode = REINTERRUPT;
2045             if (node.nextWaiter != null) // clean up if cancelled
2046                 unlinkCancelledWaiters();
2047             if (interruptMode != 0)
2048                 reportInterruptAfterWait(interruptMode);
2049         }
2050
```

可以看到，在这里，根据 acquireQueued 方法的返回值（中断标记 interrupted），进行了别的一些处理，并没有直接调用`Thread.currentThread().interrupt()`。所以我们在 acquireQueued 方法内部 2 号位置加`Thread.currentThread().interrupt()`去控制当前线程的中断标记 interrupt flag 也是不合适的。

至此，我们解释了问题：acquireQueued 方法的追问 -- 为什么 acquire 方法要靠 acquireQueued 方法的返回值判断，来调用 .interrupt() 方法。

release 方法（独占模式释放同步状态）

release 是 AQS 独占模式释放同步状态的方法。

```
    /**
     * Releases in exclusive mode.  Implemented by unblocking one or
     * more threads if {@link #tryRelease} returns true.
     * This method can be used to implement method {@link Lock#unlock}.
     *
     * 模版方法，其中 tryRelease 方法由子类实现。当前锁释放指定的 state 数量，并返回锁是否释放成功。
     *
     * @param arg the release argument.  This value is conveyed to
     *        {@link #tryRelease} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @return the value returned from {@link #tryRelease}
     */
    public final boolean release(int arg) {
        // tryRelease 方法由子类实现，尝试释放同步状态
        if (tryRelease(arg)) {
            Node h = head;
            // 判断边界条件，然后把 CHL 同步队列中头节点的后继节点唤醒
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
        return false;
    }
```

这是一个模板方法，主要做了两件事情。一个是调用子类实现的方法去尝试释放同步状态。一个是把队列中头节点的后继节点唤醒。

AQS 唤醒 CHL 同步队列中的后续节点，调用了 unparkSuccessor 方法。

unparkSuccessor 方法

```
    /**
     * Wakes up node's successor, if one exists.
     *
     * 如果当前节点的后继节点存在，则唤醒它
     *
     * @param node the node
     */
    private void unparkSuccessor(Node node) {
```

```
        /*
         * If status is negative (i.e., possibly needing signal) try
         * to clear in anticipation of signalling.  It is OK if this
         * fails or if status is changed by waiting thread.
         */
        // node 的状态
        int ws = node.waitStatus;
        // 如果当前节点的状态不是被取消，则把当前节点的状态置为 0，可以理解为重置当前节点的状态（之前的状态可能是用于指示后继节点的，比如 SIGNAL）。
        if (ws < 0)
            compareAndSetWaitStatus(node, ws, 0);

        /*
         * Thread to unpark is held in successor, which is normally
         * just the next node.  But if cancelled or apparently null,
         * traverse backwards from tail to find the actual
         * non-cancelled successor.
         */
        // 当前节点的后继节点
        Node s = node.next;
        // 如果当前节点的后继节点为空，或者后继节点被取消
        if (s == null || s.waitStatus > 0) {
            s = null;
            // 则从尾节点开始从后向前遍历，找到最靠近当前节点的正常节点
            for (Node t = tail; t != null && t != node; t = t.prev)
                if (t.waitStatus <= 0)
                    s = t;
        }
        // 当前节点的后继节点不为 null 时，唤醒当前节点的后继节点
        if (s != null)
            LockSupport.unpark(s.thread);
    }
```

unparkSuccessor 方法在做的事情是，如果当前节点的后继节点存在，则唤醒它。大部分代码在做边界条件的处理，真正唤醒线程使用的 LockSupport.unpark(thread) 方法。

LockSupport.unpark(thread) 这个方法我们可以暂时不用关注。它里面调用了 UNSAFE.unpark(thread)，UNSAFE 类是一个很底层的类，并且是闭源的，不推荐普通开发人员在业务代码中直接调用的，一般是框架底层一些的实现会去使用。我们只需要知道，LockSupport.unpark(thread) 可以唤醒一个线程即可。

未完待续

本文目前仅分析了 AQS 中，独占模式获取同步状态的逻辑，以下内容待完善：

1. AQS 中共享模式获取同步状态的逻辑。
2. AQS 中共享模式释放同步状态的逻辑。
3. CHL 队列为什么是双向的？（分析中已经体现了，计划在文章中概括回答）

时间有限，未完待续...