```
ure, block.
         *   CANCELLED:  This node is cancelled due to timeout or
interrupt.
         *               Nodes never leave this state. In particular,
         *               a thread with cancelled node never again blocks.
         *   CONDITION:  This node is currently on a condition queue.
         *               It will not be used as a sync queue node
         *               until transferred, at which time the status
         *               will be set to 0. (Use of this value here has
         *               nothing to do with the other uses of the
         *               field, but simplifies mechanics.)
         *   PROPAGATE:  A releaseShared should be propagated to other
         *               nodes. This is set (for head node only) in
         *               doReleaseShared to ensure propagation
         *               continues, even if other operations have
         *               since intervened.
         *   0:          None of the above
         *
         * The values are arranged numerically to simplify use.
         * Non-negative values mean that a node doesn't need to
         * signal. So, most code doesn't need to check for particular
         * values, just for sign.
         *
         * The field is initialized to 0 for normal sync nodes, and
         * CONDITION for condition nodes.  It is modified using CAS
         * (or when possible, unconditional volatile writes).
         */
        volatile int waitStatus;

        /**
         * Link to predecessor node that current node/thread relies on
         * for checking waitStatus. Assigned during enqueuing, and nulled
         * out (for sake of GC) only upon dequeuing.  Also, upon
         * cancellation of a predecessor, we short-circuit while
         * finding a non-cancelled one, which will always exist
         * because the head node is never cancelled: A node becomes
         * head only as a result of successful acquire. A
         * cancelled thread never succeeds in acquiring, and a thread only
         * cancels itself, not any other node.
         */
        volatile Node prev;

        /**
         * Link to the successor node that the current node/thread
         * unparks upon release. Assigned during enqueuing, adjusted
```

```
     * when bypassing cancelled predecessors, and nulled out (for
     * sake of GC) when dequeued.  The enq operation does not
     * assign next field of a predecessor until after attachment,
     * so seeing a null next field does not necessarily mean that
     * node is at end of queue. However, if a next field appears
     * to be null, we can scan prev's from the tail to
     * double-check.  The next field of cancelled nodes is set to
     * point to the node itself instead of null, to make life
     * easier for isOnSyncQueue.
     */
    volatile Node next;

    /**
     * The thread that enqueued this node.  Initialized on
     * construction and nulled out after use.
     */
    volatile Thread thread;

    /**
     * Link to next node waiting on condition, or the special
     * value SHARED.  Because condition queues are accessed only
     * when holding in exclusive mode, we just need a simple
     * linked queue to hold nodes while they are waiting on
     * conditions. They are then transferred to the queue to
     * re-acquire. And because conditions can only be exclusive,
     * we save a field by using special value to indicate shared
     * mode.
     */
    Node nextWaiter;

    /**
     * Returns true if node is waiting in shared mode.
     */
    final boolean isShared() {
        return nextWaiter == SHARED;
    }

    /**
     * Returns previous node, or throws NullPointerException if null.
     * Use when predecessor cannot be null.  The null check could
     * be elided, but is present to help the VM.
     *
     * @return the predecessor of this node
     */
    final Node predecessor() throws NullPointerException {
        Node p = prev;
        if (p == null)
            throw new NullPointerException();
```

```
            else
                return p;
        }

        Node() {    // Used to establish initial head or SHARED marker
        }

        Node(Thread thread, Node mode) {     // Used by addWaiter
            this.nextWaiter = mode;
            this.thread = thread;
        }

        Node(Thread thread, int waitStatus) { // Used by Condition
            this.waitStatus = waitStatus;
            this.thread = thread;
        }
    }
```

```

看看一些重要属性：

* `EXCLUSIVE`：表示一个**独占节点**，即只有一个线程可以获取锁资源，如`ReentrantLock`。当一个线程成功获取锁时，会创建一个`EXCLUSIVE`节点对象并将其设置为当前线程的节点状态。当其他线程获取锁时，发现已经有线程持有了锁，则将自身封装成一个`EXCLUSIVE`节点并加入等待队列中。
* `SHARED`：表示一个**共享节点**，即多个线程可以同时获取锁资源，如`ReentrantReadWriteLock`。与`EXCLUSIVE`不同，`SHARED`允许多个线程同时持有锁，但仍需循序公平性。当一个线程请求共享锁时，如果锁是可用的，则线程可以直接获取锁；否则，线程会被封装成一个`SHARED`节点并加入等待队列中。
* `waitStatus`：当前节点在等待队列中的状态。
+ 0：当一个Node被初始化时默认的状态
+ `CANCELLED`：当节点在等待过程中被中断或超时，它将被标记为取消状态，此后该节点将不再参与竞争，其线程也不会再阻塞。
+ `CONDITION`：这表示节点当前在条件队列中等待。线程执行了`await()`方法后，释放了锁并进入等待状态，直到其他线程调用`signal()`方法。在条件队列中的节点可以被移动到一个特殊的条件等待队列，直到条件得到满足。有关**条件队列**的内容我将在之后的文章中讲解。
+ `SIGNAL`：线程需要被唤醒
+ `PROPAGATE`：这个状态通常用于共享模式，当一个线程释放锁或者资源时，如果头节点是PROPAGATE状态，它会将释放操作传播到后续的节点，以便这些节点也能尝试获取共享资源。

AQS源码深入分析
```

以最常用的`ReentrantLock`作为突破口进行解读，分析AQS源码。

`ReentrantLock`实现了`Lock`接口，`Lock`通过聚合一个AQS的子类`Sync`实现线程访问的控制。`Sync`又延申出公平锁和非公平锁。

构造方法

我们创建`ReentrantLock`，默认的构造方法会创建出非公平锁。

```
public ReentrantLock() {
    sync = new NonfairSync();
}
```

如果创建公平锁，创建的时候传入`true`。

```
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

lock()方法

如果调用`lock()`，可以看见底层调用的是`Sync`类中的`lock()`方法。

```
public void lock() {
    sync.lock();
}
```

`Sync`类中的`lock()`为抽象方法，有公平和非公平两种实现方式，默认为非公平。在非公平锁中，`lock()`方法中通过`compareAndSetState(0, 1)`来设置锁的状态，如果`state`在设置之前的值就是0，那就可以成功修改成1，如果设置之前不是0，则修改失败，其实就是CAS算法。第一个线程抢占锁，`compareAndSetState(0,1)`设置成功，当前线程抢到锁。第二个线程调用`compareAndSetState(0,1)`就会设置失败，进而调用`acquire(1)`。

```
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = 7316153563782823691L;

    /**
     * `null`并且`waitStatus`为-1，调用`unparkSuccessor(h)`。
```

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
protected final void setExclusiveOwnerThread(Thread thread) {
    exclusiveOwnerThread = thread;
}
```

进入`unparkSuccessor(Node node)`，传入参数为head虚拟头节点，其`waitStatus`为-1，所以要调用`compareAndSetWaitStatus(node, ws, 0)`，将head的`waitStatus`设置为0。然后获取node的后置节点，也就是线程B节点。s不为null并且s的`waitStatus`也不大于0，就不会进入`if`代码块里。

由于s不为null，所以会调用`unpark()`唤醒B线程。

```
Wakes up node's successor, if one exists.
Params:
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.
     */
    int ws = node.waitStatus;
    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.
     */
    Node s = node.next;
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

如果锁的可重入次数不为1，也就是state不为1，tryRelease()返回false，release()方法也返回false，释放锁失败。

唤醒B线程之后，回到之前的`acquireQueued()`方法，B线程会在`for`循环中继续执行，重新尝试调用`tryAcquire()`抢锁，这次在`if`判断中执行`tryAcquire()`可以成功，再次把锁的`state`设置为1。

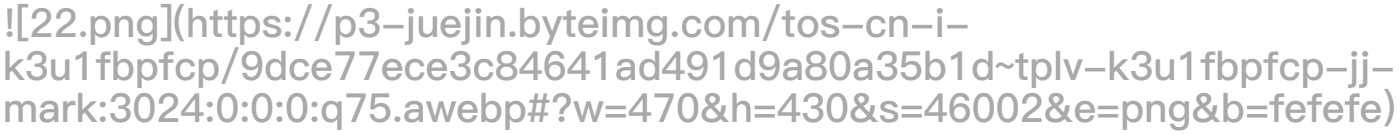
正常情况线程A释放锁之后就该线程B抢到锁了，但是有极端情况，突然来个线

程D把锁抢走了，出现这种情况就是因为`ReentrantLock`是非公平锁，会出现插队的情况。

线程B抢到锁之后就会调用`setHead()`离开队列里，去窗口办理业务了。在`setHead(Node node)`中，将线程B设置为头节点，并且将原来B节点的`thread`属性设置为null，再将B节点的`prev`属性设置为null。通过`setHead()`方法的操作，就可以将原来的B节点移除队列，设置新的虚拟头节点。接着会将p节点也就是原来的虚拟头节点的`next`设置为null，这样队列中就不存在原来的虚拟头节点了，而原来的线程B去窗口办理业务了，他所在的node节点变成了虚拟头节点。上述就是完整的解锁过程。

```
/**
 * Sets head of queue to be node, thus dequeuing. Called only by
 * acquire methods.  Also nulls out unused fields for sake of GC
 * and to suppress unnecessary signals and traversals.
 *将队列的头部设置为node，从而脱离队列。只能由acquire方法调用。为了GC和抑制不必要的信号和遍历，还将未使用的字段清空。
 * @param node the node
 */
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```

抢锁入队和释放锁出队的正常流程都已经走完了，大家好好沉淀沉淀，把这部分捋顺了，也这是一个学习阅读源码的好机会。我们还差一个方法没有看，就是在`acquireQueued()`方法中`failed`为true，就会调用`cancelAcquire(Node node)`方法，来研究一下`cancelAcquire(Node node)`方法。



`cancelAcquire(Node node)`

如果现在有三个线程节点在排队，线程A、线程B以及线程C。线程B不想排队，那它退出之后A就得指向C了，这个过程是有一些麻烦的，所以这个取消流程也比较重要。

```
  private void cancelAcquire(Node node) {
        // Ignore if node doesn't exist
        if (node == null)
            return;

        node.thread = null;

        // Skip cancelled predecessors
        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 {
```

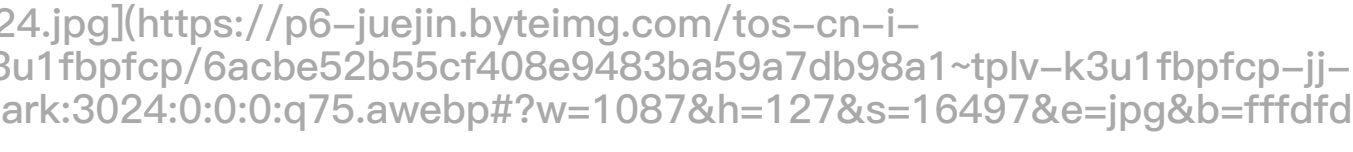
```
                unparkSuccessor(node);
            }

            node.next = node; // help GC
        }
    }
```

下面我们分情况讨论

* **队尾5号节点退出**

5号节点肯定不为null，将`thread`属性设置为null。5号节点的前置节点是4号节点，4号节点的`waitStatus`不大于0，不会进入循环。4号节点的next为5号节点，将`predNext`设置为5号节点。将5号节点的`waitStatus`设置为`Node.CANCELLED`。进行`if`判断，node为tail节点然后用`caompareAndSetTail(node,pred)`将尾节点设置为4号节点。再通过`compareAndSetNext(pred,predNext,null)`将4号节点的next设置为null。至此，5号节点完成退出。



* **4号节点出队**

4号节点的前一个节点是3号节点，正常情况下是不会进入`while`循环中的，但是有不正常的情况。4号节点退出的时候，**3号节点也要取消**，3号节点的`waitStatus`设置为`Node.CANCELLED`，所以可以进入`while`循环中，会把4号节点的`prev`设置为2号节点，就是要找到前面`waitStatus`不大于0的节点，也就是没有取消的节点。

还是假设3号节点没取消，`pred`为3号节点，`predNext`为4号节点。将4号节点的`waitStatus`设置为`Node.CANCELLED`。4号节点不是尾节点，所以进入`else`代码块。可以通过if判断条件，`next`赋值为5号节点。if判断也可以进入，就通过`compareAndSetNext(pred, predNext, next)`方法将3号节点（pred）的下一个节点（predNext）设置为next，也就是5号节点。

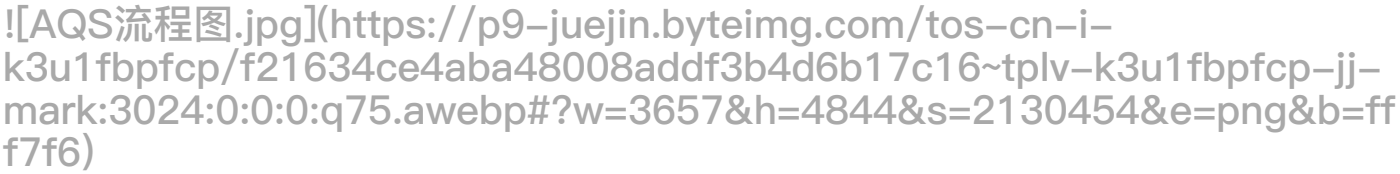
最后将4号节点的`next`设置为node，也就是指向了自己，方便垃圾回收。

总结
--

整个`ReentrantLock`的加锁过程，可以分为三阶段：

1. 尝试加锁
2. 加锁失败，线程入队列
3. 线程入队列之后，进入阻塞状态

我在这里给大家放一张加锁和释放锁的流程图，绝对有助于理解整个流程。大家也可以在学完这部分内容之后画一张流程图，梳理一下脉络。



大多数开发者可能永远不会直接使用AQS，但是知道AQS原理对于架构设计非常有帮助，学习之后我都觉得我长脑子了。

原文链接: https://juejin.cn/post/7346486084735483945