AQS源码:

获取锁的过程:本质就是获取和state的交配权,哈哈哈,如果获取不到怎么进行后续处理,AQS就是这个.

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

3.tryAcquire

```
AQS:
 protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
    }
所以要看子类的实现
java.util.concurrent.locks.ReentrantLock.FairSync

protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            //getState是AQS里的
            int c = getState();
            //如果没有其他线程获取到锁,state
            if (c == 0) {
                //公平锁和非公平锁的区别,先判断有没有前序节点,如果没有cas一次
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    //成功之后设置专属线程,然后返回获取成功的
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            //锁重入
            else if (current == getExclusiveOwnerThread()) {
                //
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
}
```

3.1getState

```
protected final int getState() {
        return state;
    }
```

3.2hasQueuedPredecessors

```
public final boolean hasQueuedPredecessors() {
        // The correctness of this depends on head being initialized
        // before tail and on head.next being accurate if the current
        // thread is first in queue.
        Node t = tail; // Read fields in reverse initialization order
        Node h = head;
        Node s;
        return h != t &&
            ((s = h.next) == null || s.thread != Thread.currentThread());
    }
```

3.3setExclusiveOwnerThread

```
protected final void setExclusiveOwnerThread(Thread thread) {
        exclusiveOwnerThread = thread;
    }
```

4.acquireQueued

该方法是最复杂的一个方法, 也是最难啃的骨头, 看代码之前首先简单的说明几点:

(1) 能执行到该方法, 说明`addWaiter`方法已经成功将包装了当前Thread的节点添加到了等待队列的队尾 (2) 该方法中将再次尝试去获取锁 (3) 在再次尝试获取锁失败后, 判断是否需要把当前线程挂起

```
final boolean acquireQueued(final Node node, int arg) {
        //失败的标识
        boolean failed = true;
        try {
            //打断标识
            boolean interrupted = false;
            //自旋
            for (;;) {
                final Node p = node.predecessor();
                //如果节点的头节点是head,再获取一次锁
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
```

```
                }
                //
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    //被打断
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

4.1.0setHead

```
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```

4.1shouldParkAfterFailedAcquire

获取失败后park线程

Node的waitstatus

```
/** waitStatus value to indicate thread has cancelled */
static final int CANCELLED =  1;
/** waitStatus value to indicate successor's thread needs unparking */
static final int SIGNAL    = -1;
/** waitStatus value to indicate thread is waiting on condition */
static final int CONDITION = -2;
/**
 * waitStatus value to indicate the next acquireShared should
 * unconditionally propagate
 */
static final int PROPAGATE = -3;
```

一共有四种状态，但是我们在开篇的时候就说过，在独占锁锁的获取操作中，我们只用到了其中的两个——`CANCELLED`和`SIGNAL`。当然，前面我们在创建节点的时候并没有给waitStatus赋值，因此每一个节点最开始的时候waitStatus的值都被初始化为0，即不属于上面任何一种状态。

那么`CANCELLED`和`SIGNAL`代表什么意思呢?

`CANCELLED`状态很好理解，它表示Node所代表的当前线程已经取消了排队，即放弃获取锁了。

`SIGNAL`这个状态就有点意思了，它不是表征当前节点的状态，而是当前节点的下一个节点的状态。 当一个节点的waitStatus被置为`SIGNAL`，就说明它的下一个节点（即它的后继节点）已经被挂起了（或者马上就要被挂起了），因此在当前节点释放了锁或者放弃获取锁时，如果它的waitStatus属性为`SIGNAL`，它还要完成一个额外的操作——唤醒它的后继节点。

有意思的是，`SIGNAL`这个状态的设置常常不是节点自己给自己设的，而是前序节点设置的，这里给大家打个比方：

比如说出去吃饭，在人多的时候经常要排队取号，你取到了8号，前面还有7个人在等着进去，你就和排在你前面的7号讲"哥们，我现在排在你后面，队伍这么长，估计一时半会儿也轮不到我，我去那边打个盹，一会轮到你进去了(release)或者你不想等了(cancel), 麻烦你都叫醒我"，说完，你就把他的waitStatus值设成了`SIGNAL`。

换个角度讲，当我们决定要将一个线程挂起之前，首先要确保自己的前驱节点的waitStatus为`SIGNAL`，这就相当于给自己设一个闹钟再去睡，这个闹钟会在恰当的时候叫醒自己，否则，如果一直没有人来叫醒自己，自己可能就一直睡到天荒地老了。

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        //确认下pred是前一个节点吗,应该是,因为waitstatus保存是下一个节点的状态
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
            /*
             * This node has already set status asking a release
             * to signal it, so it can safely park.
             */
            return true;
        //被取消的节点要跳过,继续
        if (ws > 0) {
            /*
             * Predecessor was cancelled. Skip over predecessors and
             * indicate retry.
             */
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            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.
             */
            //cas设置
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        //不需要park,只有等于signal的才需要park
        return false;
    }
```

4.2parkAndCheckInterrupt

```
  private final boolean parkAndCheckInterrupt() {
        //park线程
        LockSupport.park(this);
        //打断
        return Thread.interrupted();
    }
```

4.2.1LockSupport.park(this)

```
public static void park(Object blocker) {
        Thread t = Thread.currentThread();
        setBlocker(t, blocker);
        //UNSAFE要单独搞一个笔记
        UNSAFE.park(false, 0L);
        setBlocker(t, null);
    }
```

4.2.2Thread.interrupted();

检测当前的线程有没有被打断.这个线程会清空线程的中断状态.

```
public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}
```

4.2.2.1setBlocker

```
private static void setBlocker(Thread t, Object arg) {
    // Even though volatile, hotspot doesn't need a write barrier here.
    UNSAFE.putObject(t, parkBlockerOffset, arg);
}
```

5.addWaiter

在获取锁失败后调用, 将当前请求锁的线程包装成Node扔到sync queue中去，并返回这个Node。

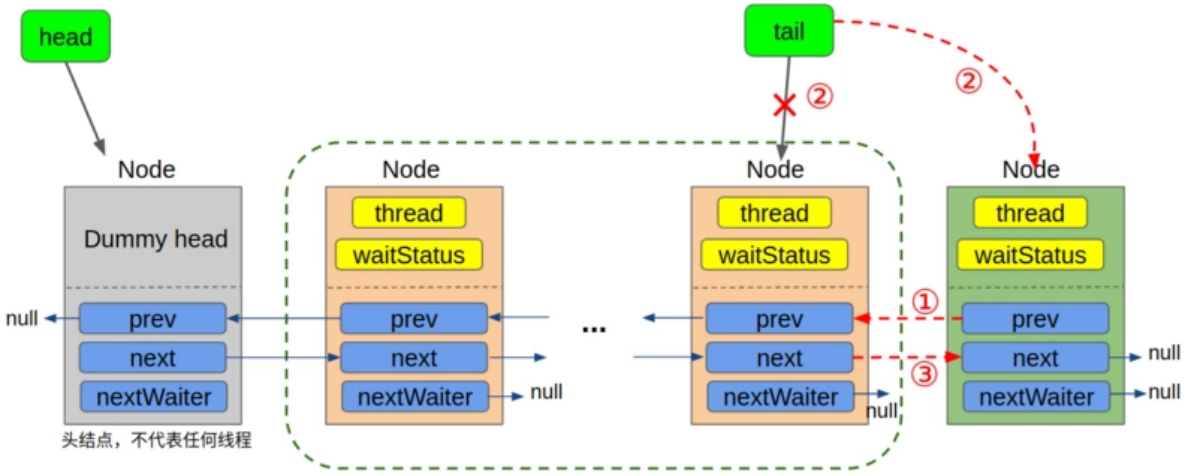
5.addWaiter

```
private Node addWaiter(Node mode) {
        //包装当前的线程
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
```

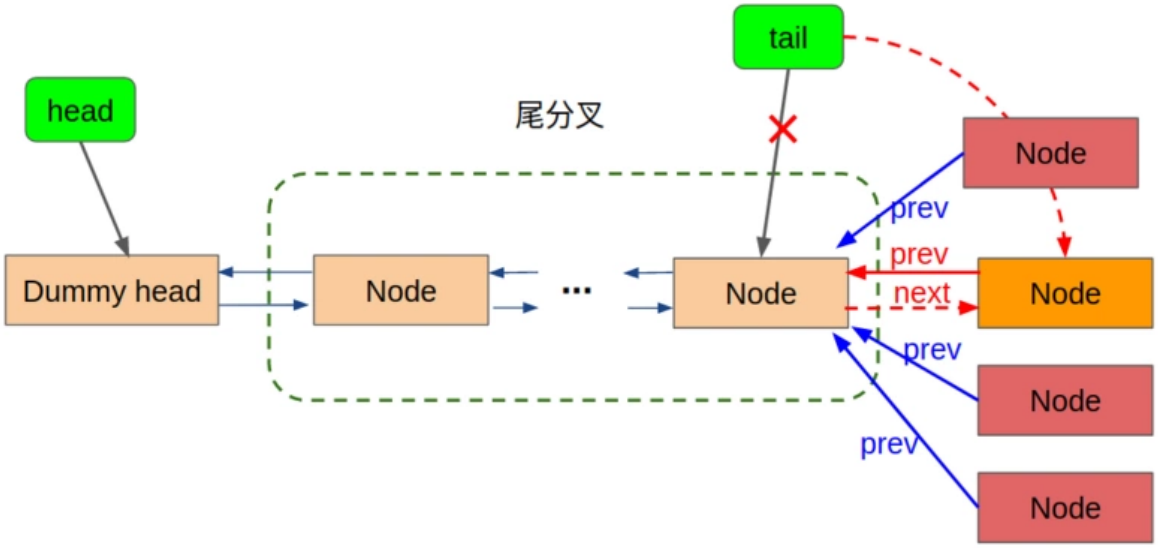
```
        Node pred = tail;
        //如果尾结点不等于null
        if (pred != null) {
            node.prev = pred;
            //cas设置,添加尾结点
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                //成功返回
                return node;
            }
        }
        //1.pred==null
        //2.cas失败
        //两种情况都有可能进入enq,如果能成功就立马成功,应该是这种设计思想吧
        enq(node);
        return node;
    }
```

5.1enq

尾分叉:将一个节点node添加到`sync queue`的末尾也需要三步：

1. 设置node的前驱节点为当前的尾节点：`node.prev = t`
2. 修改`tail`属性，使它指向当前节点
3. 修改原来的尾节点，使它的next指向当前节点
4. 

5.



6. 注意，这里第三步是在第二步执行成功后才执行的，这就意味着，有可能即使我们已经完成了第二步，将新的节点设置成了尾节点，**此时原来旧的尾节点的next值可能还是`null`**(因为还没有来的及执行第三步)，所以如果此时有线程恰巧从头节点开始向后遍历整个链表，则它是遍历不到新加进来的尾节点的，但是这显然是不合理的，因为现在的tail已经指向了新的尾节点。 另一方面，当我们完成了第二步之后，第一步一定是完成了的，所以如果我们从尾节点开始向前遍历，已经可以遍历到所有的节点。这也就是为什么我们在AQS相关的源码中，有时候常常会出现从尾节点开始逆向遍历链表——因为一个节点要能入队，则它的prev属性一定是有值的，但是它的next属性可能暂时还没有值。

   至于那些“分叉”的入队失败的其他节点，在下一轮的循环中，它们的prev属性会重新指向新的尾节点，继续尝试新的CAS操作，最终，所有节点都会通过自旋不断的尝试入队，直到成功为止。

```
private Node enq(final Node node) {
  //自旋,直到添加到尾部
  for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            //新建一个头节点,头尾相等
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            //队列不为空
            //第一步,node.prev指向尾结点.同时会有多个线程成功
            node.prev = t;
            //第二步cas设置尾结点,只有一个线程能成功
            if (compareAndSetTail(t, node)) {
                //第三步设置返回
                t.next = node;
                return t;
            }
        }
    }
}
```

总结:

6.selfInterrupt

```
static void selfInterrupt() {
    Thread.currentThread().interrupt();
}
```

注意interrupt和interrupted的区别

```
public void interrupt() {
    if (this != Thread.currentThread())
        checkAccess();

    synchronized (blockerLock) {
        Interruptible b = blocker;
        if (b != null) {
            interrupt0();           // Just to set the interrupt flag
            b.interrupt(this);
            return;
        }
    }
    interrupt0();
}
```

```
public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}
```

一目了然吗

```
private native void interrupt0();
```

引用:https://segmentfault.com/a/1190000015739343,有很多解释都是从这里面copy的,但是还是要回归源码的,只有自己的思考才是自己的.