Java多线程进阶（七）—— J.U.C之locks框架：AQS独占功能剖析(2)

 Ressmix 发布于 2018-07-29

本文首发于一世流云的专栏：https://segmentfault.com/blog...

一、本章概述

本章以ReentrantLock的调用为例，说明AbstractQueuedSynchronizer提供的独占功能。
本章结构如下：

1. 以ReentrantLock的公平策略为例，分析AbstractQueuedSynchronizer的独占功能
2. 以ReentrantLock的非公平策略为例，分析AbstractQueuedSynchronizer的独占功能
3. 分析AbstractQueuedSynchronizer的锁中断、限时等待等功能

二、ReentrantLock的公平策略原理

本节对ReentrantLock公平策略的分析基于以下示例：

> 假设现在有3个线程：ThreadA、ThreadB、ThreadC，一个公平的独占锁，3个线程会依次尝试去获取锁：`ReentrantLock lock=new ReentrantLock(true);`

线程的操作时序如下：

```
//ThreadA    lock

//ThreadB    lock

//ThreadC    lock

//ThreadA    release

//ThreadB    release

//ThreadC    release
```

2.1 ThreadA首先获取到锁

ThreadA首先调用ReentrantLock的**lock**方法，我们看下该方法的内部：

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

最终其实调用了FairSync的lock方法：

```
static final class FairSync extends Sync {
    private static final long serialVersionUID = -3000897897090466540L;

    final void lock() {
        acquire( arg: 1);
    }
```

acquire方法来自AQS：

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

其中**tryAcquire**方法需要AQS的子类自己去实现，我们来看下ReentrantLock中的实现：

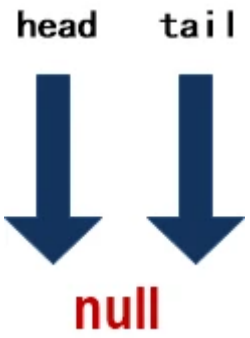
```
/**
 * 尝试获取锁.
 *
 * @return 获取成功返回true，否则返回false
 */
protected final boolean tryAcquire(int acquires) {  // 入参acquires=1
    final Thread current = Thread.currentThread();  // 当前线程
    int c = getState();                             // 获取同步状态
    if (c == 0) {                                   // 表示锁未被占用
        // 如果等待队列中，当前线程前没有其它线程，则以CAS方式更新同步状态
        if (!hasQueuedPredecessors() &&
            compareAndSetState( expect: 0, acquires)) {
            // 更新成功，设置锁的占有线程为当前线程
            setExclusiveOwnerThread(current);
            return true;
        }
    } else if (current == getExclusiveOwnerThread()) {  // 判断是否属于重入的情况
        int nextc = c + acquires;                       // 重入时，同步状态累加1
        if (nextc < 0)                                  // 重入次数过大，溢出了
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

可以看到，在ReentrantLock中，同步状态State的含义如下：

State	资源的定义
0	表示锁可用

State	资源的定义
1	表示锁被占用
大于1	表示锁被占用，且值表示同一线程的重入次数

ThreadA是首个获取锁的线程，所以上述方法会返回true，第一阶段结束。（ThreadA一直保持占有锁的状态）
此时，AQS中的等待队列还是空：



2.2 ThreadB开始获取锁

终于，ThreadB要登场了，一样，ThreadB先去调用lock方法，最终调用AQS的acquire方法：

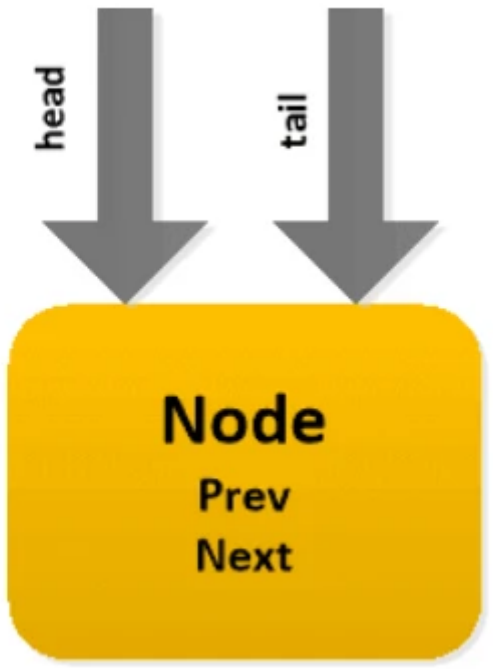
```
public final void acquire(int arg) {
    // 尝试获取锁
    // 获取失败，则将当前线程包装成结点后加入等待队列
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

tryAcquire方法肯定是返回false（因为此时ThreadA占有着锁）。
接下来看下**addWaiter**方法，这个方法其实就是将当前调用线程包装成一个【独占结点】，添加到等待队列尾部。

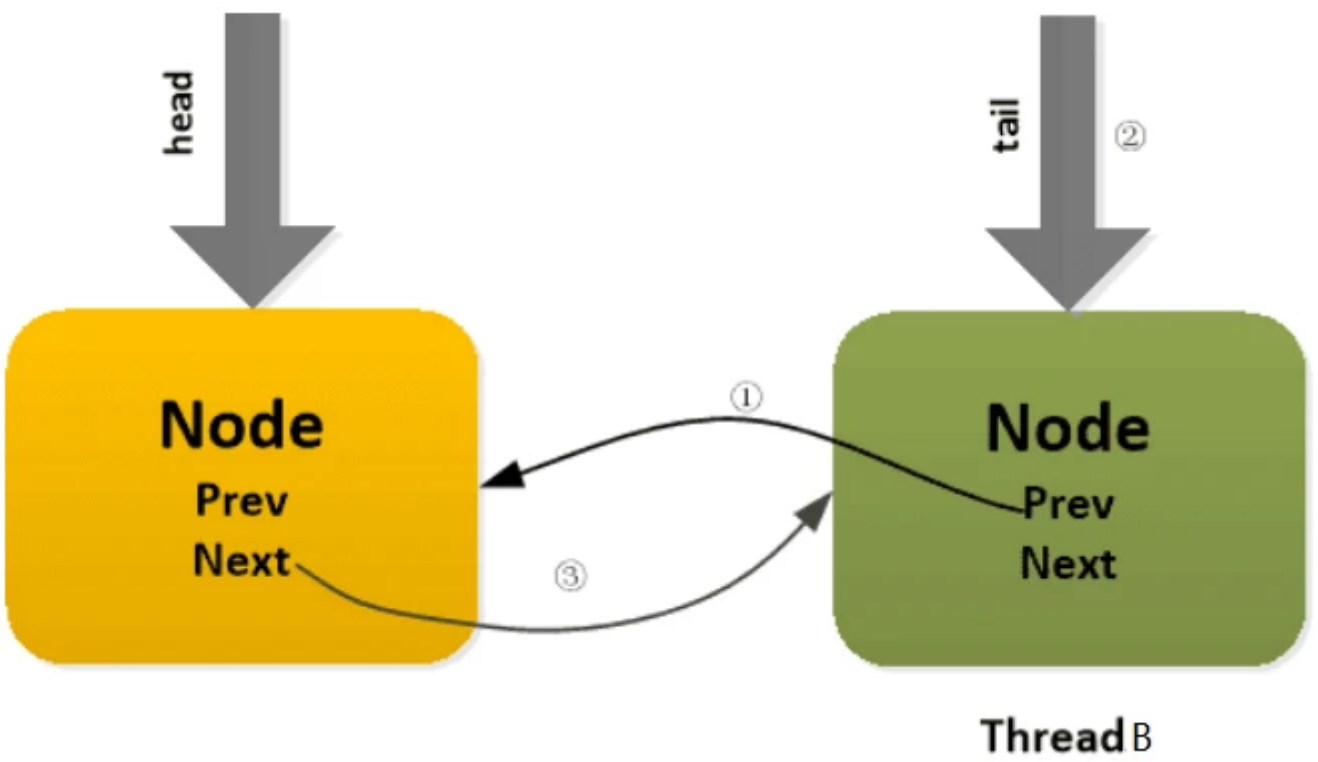
```
private Node addWaiter(Node mode) {
    // 将线程包装成结点
    Node node = new Node(Thread.currentThread(), mode);

    // 先尝试一次添加到尾部，如果添加成功，就不用走下面的enq方法了（这部分算是优化，可以忽略）
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }

    // 将结点插入到队尾
    enq(node);
    return node;
}
```

```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

这里关键是enq方法，因为并发插入的情况存在，所以该方法设计成了自旋操作，保证结点能成功插入，具体步骤如下：
①当队列为空的时候，先创建一个dummy头结点；



②进入下一次循环，插入队尾结点。



好了，ThreadB已经被包装成结点插入队尾了，接下来会调用**acquireQueued**方法，这也是AQS中最重要的方法之一：

```
/**
 * 从等待队列中选取队首线程，并尝试获取锁。如果获取不到，就要确保在前驱能唤醒自己的情况下（将前驱状态置为SIGNAL）进入阻塞状态。
 *
 * 注意，正常情况下，该方法会一直阻塞当前线程，除非获取到锁才返回；但是如果执行过程中，抛出异常（tryAcquire方法），那么会将当前结点移除，继续上抛异常
 *
 * @return 如果线程阻塞过程中被中断，则返回true
 */
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);
    }
}
```

> 在AQS中，等待队列中的线程都是阻塞的，当某个线程被唤醒时，只有该线程是首结点（线程）时，才有权去尝试获取锁。

上述方法中，将ThreadB包装成结点插入队尾后，先判断ThreadB是否是首结点（注意不是头结点，头结点是个dummy结点），发现确实是首结点（node.predecessor==head），于是调用tryAcquire尝试获取锁，但是获取失败了（此时ThreadA占有着锁），就要判断是否需要阻塞当前线程。

判断是否需要阻塞线程：

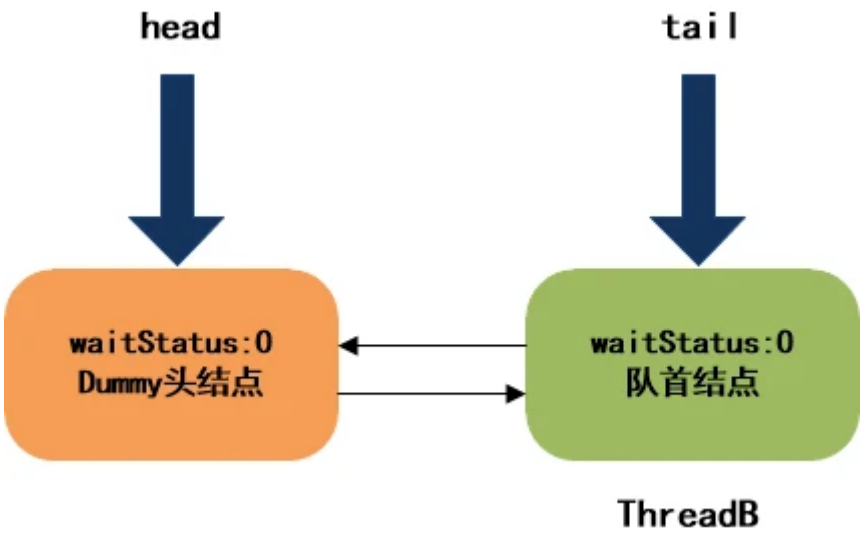
```java
/**
 * 判断是否需要阻塞当前线程
 *
 * 注意：CLH队列的一个特点就是：将当前结点的状态保存在它的前驱中。
 * 前驱状态是【-1:等待唤醒】时，才会阻塞当前线程
 *
 * @return true表示需要阻塞当前调用线程
 */
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;        // 前驱结点的状态
    if (ws == Node.SIGNAL)           // SIGNAL:后续结点需要被唤醒（这个状态说明当前结点的前驱将来会来唤醒我，我可以安心的被阻塞）
        return true;
    if (ws > 0) {                    // CANCELED:取消（说明当前结点（线程）因意外被中断/取消，需要将其从等待队列移除）
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 对于独占功能来说，这里表示结点的初始状态0
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

注意，对于独占功能，只使用了3种结点状态：

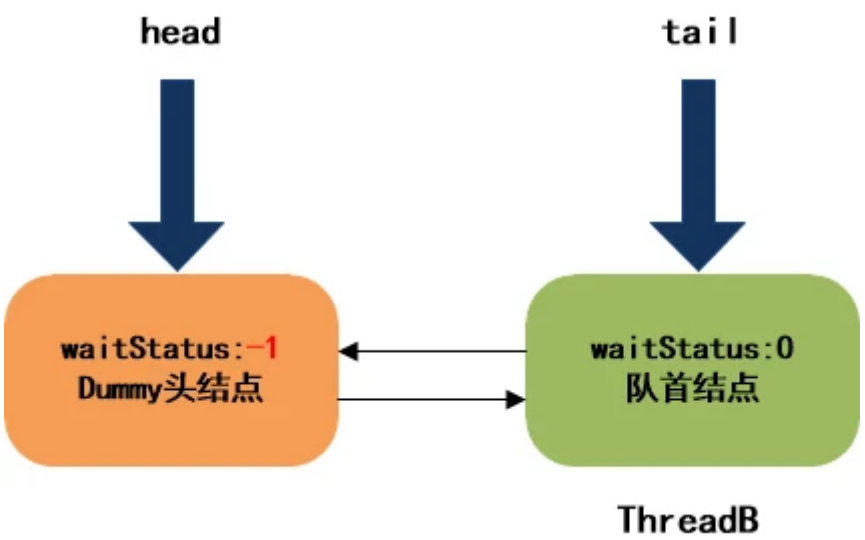
结点状态	值	描述
CANCELLED	1	取消。表示后驱结点被中断或超时，需要移出队列
SIGNAL	-1	发信号。表示后驱结点被阻塞了（当前结点在入队后、阻塞前，应确保将其prev结点类型改为SIGNAL，以便prev结点取消或释放时将当前结点唤醒。）
CONDITION	-2	Condition专用。表示当前结点在Condition队列中，因为等待某个条件而被阻塞了

对于在等待队列中的线程，如果要阻塞它，需要确保将来有线程可以唤醒它，AQS中通过将前驱结点的状态置为SIGNAL:-1来表示将来会唤醒当前线程，当前线程可以安心的阻塞。

看下图或许比较好理解：
①插入完ThreadB后，队列的初始状态如下：



②虽然ThreadB是队首结点，但是它拿不到锁（被ThreadA占有着），所以ThreadB会阻塞，但在阻塞前需要设置下前驱的状态，以便将来可以唤醒我：



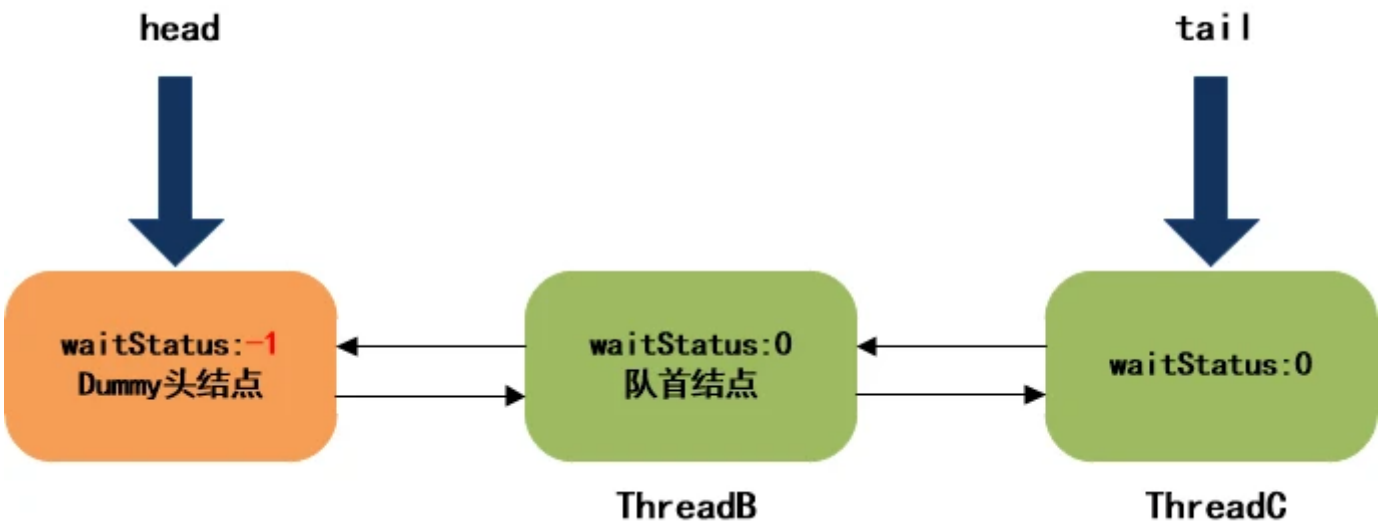
至此，ThreadB的执行也暂告一段落了（安心得在等待队列中睡觉）。

注意：补充一点，如果ThreadB在阻塞过程中被中断，其实是不会抛出异常的，只会在acquireQueued方法返回时，告诉调用者在阻塞器件有没被中断过，具体如果处理，要不要抛出异常，取决于调用者，这其实是一种延时中断机制。

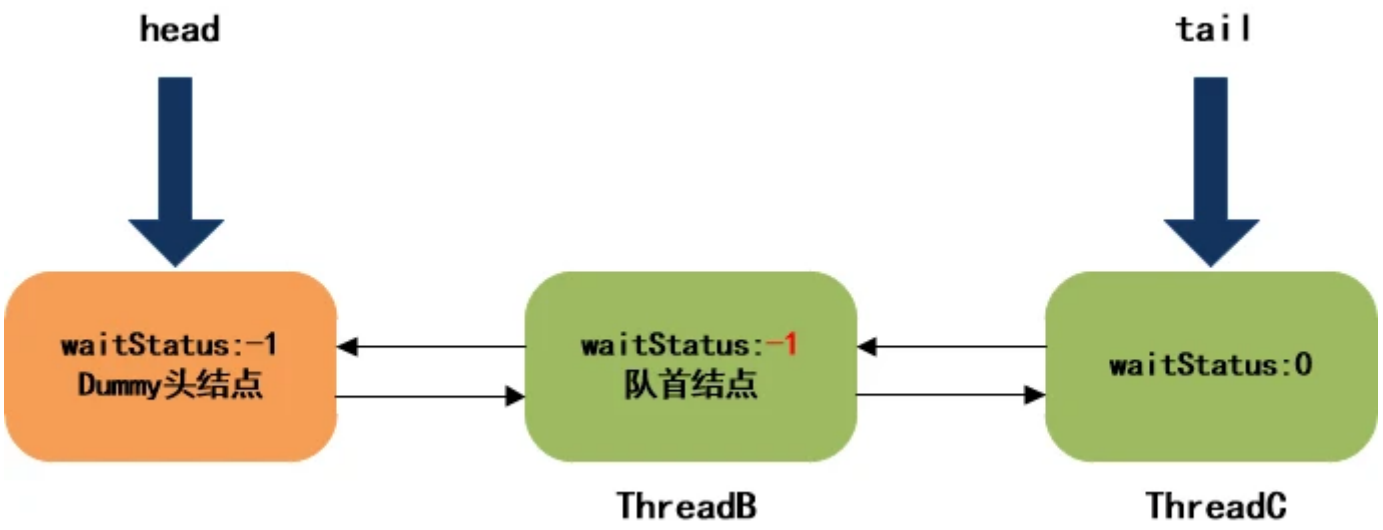
```
if (shouldParkAfterFailedAcquire(p, node) &&
    parkAndCheckInterrupt())
    interrupted = true;

/**
 * Convenience method to park and then check if interrup
 *
 * @return {@code true} if interrupted
 */
private final boolean parkAndCheckInterrupt() {
    LockSupport.park( blocker: this);
    return Thread.interrupted();
}
```

2.3 ThreadC开始获取锁

终于轮到ThreadC出场了，ThreadC的调用过程和ThreadB完全一样，同样拿不到锁，然后加入到等待队列队尾：



然后，ThreadC在阻塞前需要把前驱结点的状态置为SIGNAL：-1，以确保将来可以被唤醒：



至此，ThreadC的执行也暂告一段落了（安心得在等待队列中睡觉）。

2.4 ThreadA释放锁

ThreadA终于使用完了临界资源，要释放锁了，来看下ReentrantLock的**unlock**方法：

```
public void unlock() {
    sync.release( arg: 1);
}
```

unlock内部调用了AQS的release方法，传参1：

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {  //尝试释放锁
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);      //释放成功，则唤醒首结点
        return true;
    }
    return false;
}
```

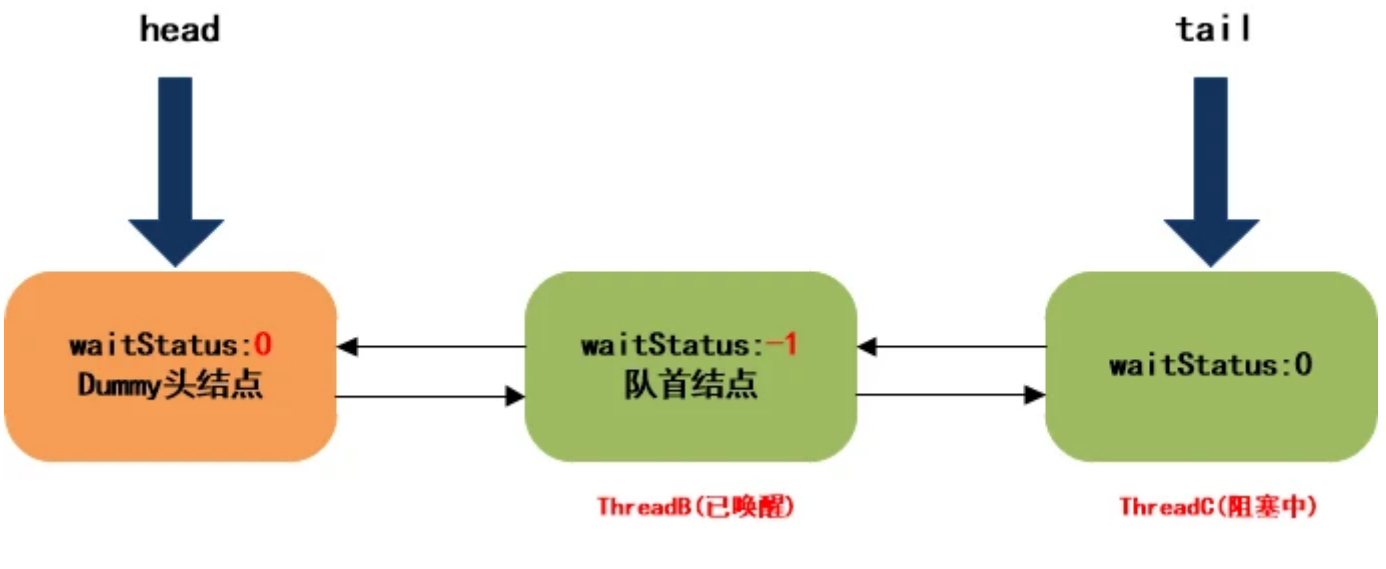
尝试释放锁的操作**tryRelease**：

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;      // 同步状态值减去1
    if (Thread.currentThread() != getExclusiveOwnerThread())    // 持有锁和释放锁的线程必须是同一个，否则会抛错
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {                       // 如果状态值为0，说明没有线程占用锁
        free = true;
        setExclusiveOwnerThread(null);  //清除占有线程
    }
    setState(c);                        // 更新状态值
    return free;
}
```

释放成功后，调用**unparkSuccessor**方法，唤醒队列中的首结点：

```
/**
 * 唤醒当前结点的后继结点（线程）
 *
 * @param node 当前结点
 */
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    if (ws < 0)     // SIGNAL:-1
        compareAndSetWaitStatus(node, ws, update: 0);    // 预置当前结点的状态为0，表示后续结点即将被唤醒

    Node s = node.next;  // 后继结点

    // 正常情况下，会直接唤醒后继结点
    // 但是如果后继结点处于1:CANCELLED状态时（说明被取消了），会从队尾开始，向前找到第一个未被CANCELLED的结点
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)     // 从tail开始向前查找是为了考虑并发入队（enq）的情况
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)      //唤醒结点
        LockSupport.unpark(s.thread);
}
```

此时，队列状态为：



2.5 ThreadB唤醒后继续执行

好了，队首结点（ThreadB）被唤醒了。
ThreadB会继续从以下位置开始执行，先返回一个中断标识，用于表示ThreadB在阻塞期间有没被中断过：

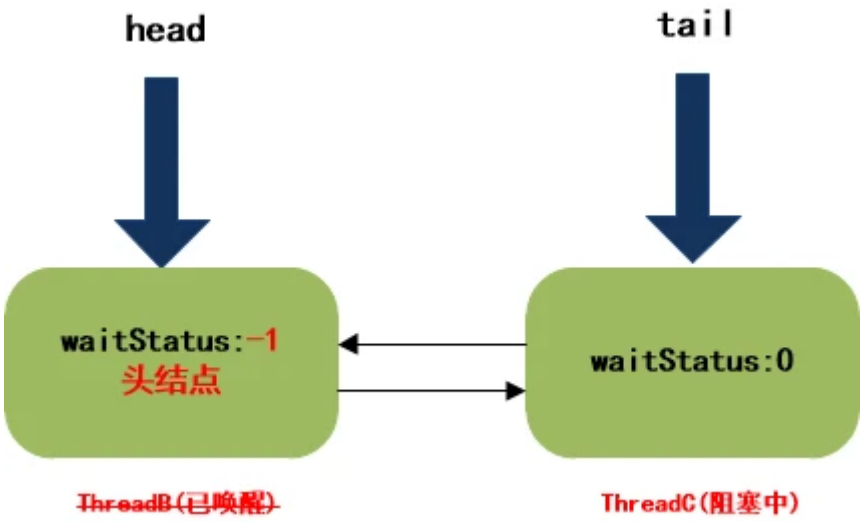
```
/**
 * Convenience method to park and then check if interrup
 *
 * @return {@code true} if interrupted
 */
private final boolean parkAndCheckInterrupt() {
    LockSupport.park( blocker: this);
    return Thread.interrupted();
}
```

然后ThreadB又开始了自旋操作，被唤醒的是队首结点，所以可以尝试tryAcquire获取锁，此时获取成功（ThreadA已经释放了锁）。
获取成功后会调用setHead方法，将头结点置为当前结点，并清除线程信息：

```
/**
 * 从等待队列中选取队首线程，并尝试获取锁。如果获取不到，就要确保在前驱能唤醒自己的情况下（将前驱状态置为SIGNAL）进入阻塞状态。
 *
 * 注意，正常情况下，该方法会一直阻塞当前线程，除非获取到锁才返回；但是如果执行过程中，抛出异常（tryAcquire方法），那么会将当前结点移除，继续上抛异常
 *
 * @return 如果线程阻塞过程中被中断，则返回true
 */
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);
    }
}
```

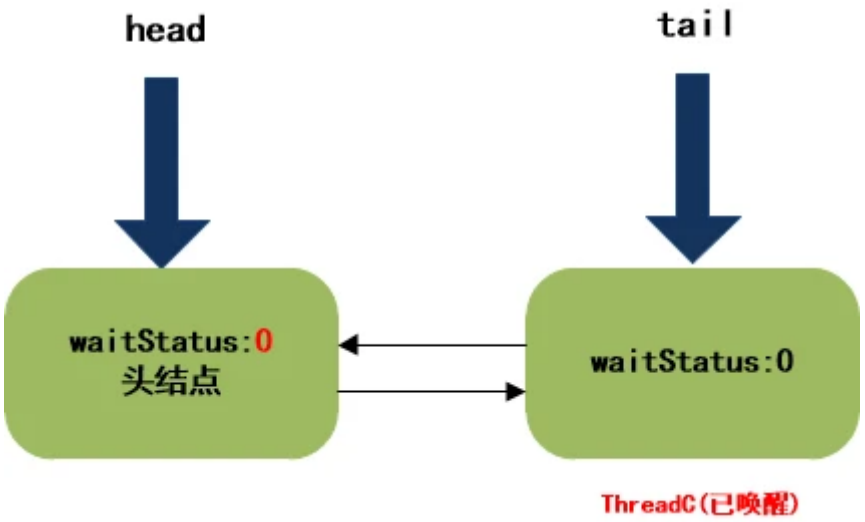
```
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```

最终的队列状态如下：

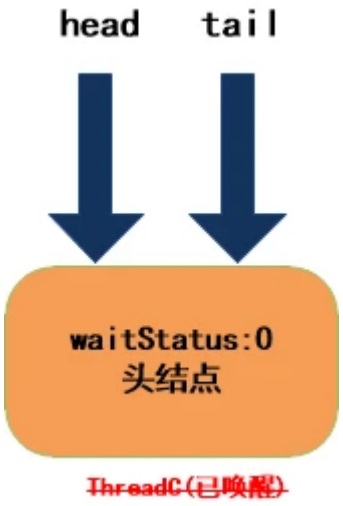


2.6 ThreadB释放锁

ThreadB也终于使用完了临界资源，要释放锁了，过程和ThreadA释放时一样，释放成功后，会调用**unparkSuccessor**方法，唤醒队列中的首结点：



队首结点（ThreadC）被唤醒后，继续从原来的阻塞处向下执行，并尝试获取锁，获取成功，最终队列状态如下：



2.7 ThreadC释放锁

ThreadC也终于使用完了临界资源，要释放锁了。释放成功后，调用unparkSuccessor方法，唤醒队列中的首结点：
此时队列中只剩下一个头结点（dummy），所以这个方法其实什么都不做。最终队列的状态就是只有一个dummy头结点。

```
/**
 * 唤醒当前结点的后继结点（线程）
 *
 * @param node 当前结点
 */
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    if (ws < 0)     // SIGNAL:-1
        compareAndSetWaitStatus(node, ws, update: 0);    // 预置当前结点的状态为0，表示后续结点即将被唤醒

    Node s = node.next;  // 后继结点

    // 正常情况下，会直接唤醒后继结点
    // 但是如果后继结点处于1:CANCELLED状态时（说明被取消了），会从队尾开始，向前找到第一个未被CANCELLED的结点
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)     // 从tail开始向前查找是为了考虑并发入队（enq）的情况
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)      //唤醒结点
        LockSupport.unpark(s.thread);
}
```

至此，AQS的独占功能已经差不多分析完了，剩下还有几个内容没分析：

1. 锁中断功能
2. 限时等待功能
3. Conditon等待功能

这些功能将在后续章节陆续分析。

三、ReentrantLock的非公平策略原理

ReenrantLock非公平策略的内部实现和公平策略没啥太大区别：
非公平策略和公平策略的最主要区别在于：

1. 公平锁获取锁时，会判断等待队列中是否有线程排在当前线程前面。只有没有情况下，才去获取锁，这是公平的含义。

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
                compareAndSetState( expect: 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;
}
```

2. 非公平锁获取锁时，会立即尝试修改同步状态，失败后再调用AQS的acquire方法。

```
/**
 * 非公平锁的lock方法.
 * 先立即尝试以CAS的方法修改同步状态，以占用锁。失败后再去排队
 */
final void lock() {
    if (compareAndSetState( expect: 0,  update: 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire( arg: 1);
}
```

acquire方法会转调非公平锁自身的tryAcquire方法，其实最终是调了nofairTryAcquire方法，而该方法相对于公平锁，只是少了“队列中是否有其它线程排在当前线程前”这一判断：

```
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) {
        if (compareAndSetState( expect: 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;
}
```

四、AQS对中断的支持

还是以ReentrantLock为例，来看下AQS是如何实现锁中断和超时的。
我们知道ReentrantLock的**lockInterruptibly**方法是会响应中断的。（线程如果在阻塞过程中被中断，会抛出InterruptedException异常）

该方法调用了AQS的**acquireInterruptibly**方法：

```
public void lockInterruptibly() throws InterruptedException {
    sync.acquireInterruptibly( arg: 1);
}
public final void acquireInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())       // 如果线程的中断标志为True,则抛出异常
        throw new InterruptedException();
    if (!tryAcquire(arg))           // 尝试获取锁
        doAcquireInterruptibly(arg);
}
```

上述代码会先去尝试获取锁，如果失败，则调用doAcquireInterruptibly方法，如下：

```
/**
 * Acquires in exclusive interruptible mode.
 *
 * @param arg the acquire argument
 */
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())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

很眼熟有木有？看下和**acquireQueued**方法的对比，唯一的区别就是：
当调用线程获取锁失败，进入阻塞后，如果中途被中断，**acquireQueued**只是用一个标识记录线程被中断过，而**doAcquireInterruptibly**则是直接抛出异常。

```
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);
    }
}
```

五、AQS对限时等待的支持

Lock接口中有一个方法：**tryLock**，用于在指定的时间内尝试获取锁，获取不到就返回。
ReentrantLock实现了该方法，可以看到，该方法内部调用了AQS的**tryAcquireNanos**方法：

```
public boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException {
        return sync.tryAcquireNanos( arg: 1, unit.toNanos(timeout));
    }
```

tryAcquireNanos方法是响应中断的，先尝试获取一次锁，失败则调用**doAcquireNanos**方法进行超时等待：

```
public final boolean tryAcquireNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        return tryAcquire(arg) ||
            doAcquireNanos(arg, nanosTimeout);
    }
```

关键是**doAcquireNano**方法，和**acquireQuqued**方法类似，又是一个自旋操作，在超时前不断尝试获取锁，获取不到则阻塞（加上了等待时间的判断）。该方法内部，调用了`LockSupport.parkNanos`来超时阻塞线程：

```
/**
 * Acquires in exclusive timed mode.
 * @param arg          the acquire argument
 * @param nanosTimeout max wait time
 * @return {@code true} if acquired
 */
private boolean doAcquireNanos(int arg, long nanosTimeout)
    throws InterruptedException {
    if (nanosTimeout <= 0L) return false;
    final long deadline = System.nanoTime() + nanosTimeout; // 当前时间+限时=截止等待时间点
    final Node node = addWaiter(Node.EXCLUSIVE);            // 将线程加入等待队列
    boolean failed = true;                                  // 标识在等待时间内是否获取到锁
    try {
        for (; ; ) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {             // 如果当前结点是首结点，可以立即尝试获取锁
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return true;
            }
            nanosTimeout = deadline - System.nanoTime();
            if (nanosTimeout <= 0L)                         // 超过了截止等待时间点
                return false;
            if (shouldParkAfterFailedAcquire(p, node) &&
                nanosTimeout > spinForTimeoutThreshold)     // 剩余等待时间超过某一阈值（spinForTimeoutThreshold：1000纳秒）
                LockSupport.parkNanos(this, nanosTimeout);  // 阻塞线程
            if (Thread.interrupted())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);                            //等待时间内，依旧没有获取到锁，则取消获取
    }
}
```

`LockSupport.parkNanos`内部其实通过Unsafe这个类来操作线程的阻塞，底层是一个native方法：

```
public static void parkNanos(long nanos) {
    if (nanos > 0)
        UNSAFE.park( b: false, nanos);
}
```

如果当前线程在指定时间内获取不到锁，除了返回false外，最终还会执行**cancelAcquire**方法：

```
/**
 * 取消一个正在尝试获取锁的线程操作.
 *
 * @param 线程的包装结点
 */
private void cancelAcquire(Node node) {
    if (node == null)
        return;
    node.thread = null;

    // 跳过当前结点之前，所有已经取消的结点
    Node pred = node.prev;
    while (pred.waitStatus > 0)          //CANCELLED:1
        node.prev = pred = pred.prev;
    Node predNext = pred.next;           //preNext此时指向第一个CANCELLED结点

    node.waitStatus = Node.CANCELLED;    //将当前结点标记为CANCELLED:1
    // 当前结点是尾结点，则尝试直接移除
    if (node == tail && compareAndSetTail(node, pred)) {
        compareAndSetNext(pred, predNext,  update: null);
    } else {     //当前结点不是尾结点或尝试移除失败（存在尾部的并发操作）
        int ws;
        if (pred != head &&
        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
    }
}
```

示例

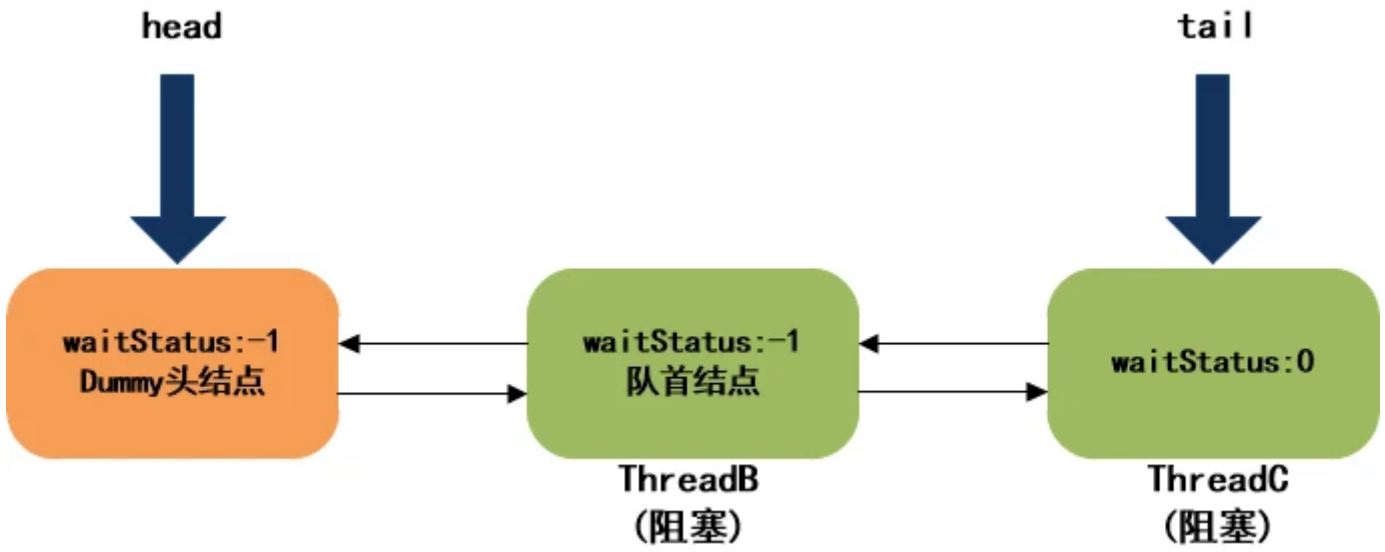
为了便于理解还是以3个线程为例：

> 假设现在有3个线程：ThreadA、ThreadB、ThreadC，一个公平的独占锁，3个线程会依次尝试去获取锁，不过此时加上了限时等待：ThreadB等待10s，ThreadA等待20s。

```
ReentrantLock lock=new ReentrantLock(true);

//ThreadA    tryLock

//ThreadB    tryLock, 10s

//ThreadC    tryLock, 20s

//ThreadA    release

//ThreadB    release

//ThreadC    release
```

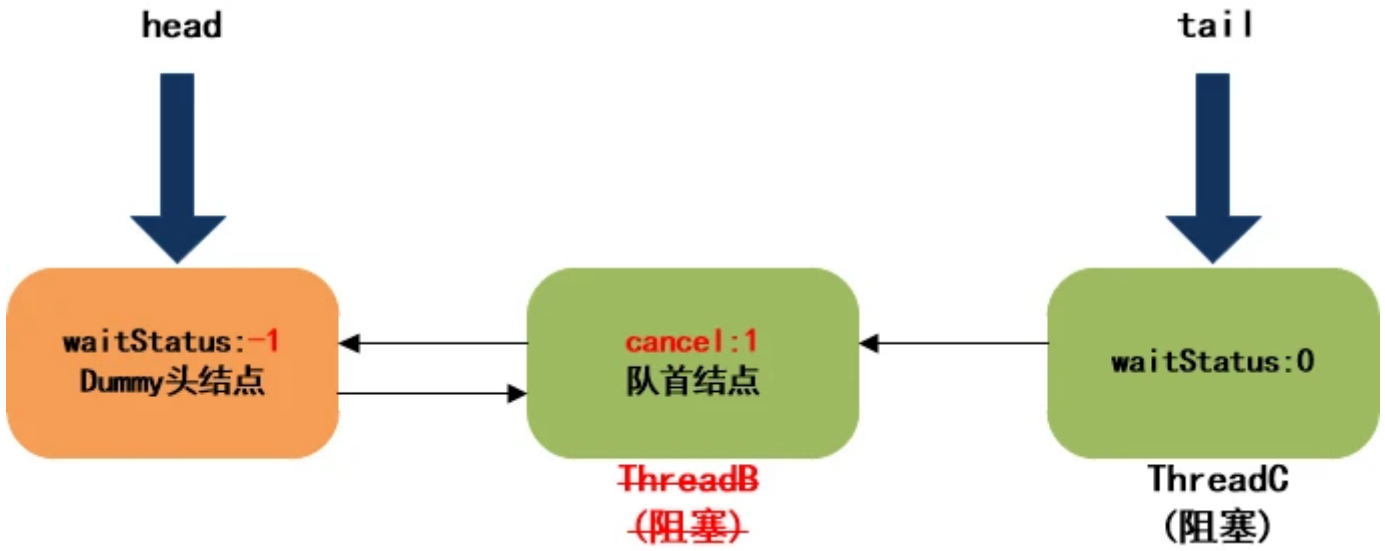
***1. ThreadA*首先获取到锁，*ThreadB*和*ThreadC*依次尝试去获取锁**

ThreadB和ThreadC经过两轮自旋操作后，等待队列的情况如下：



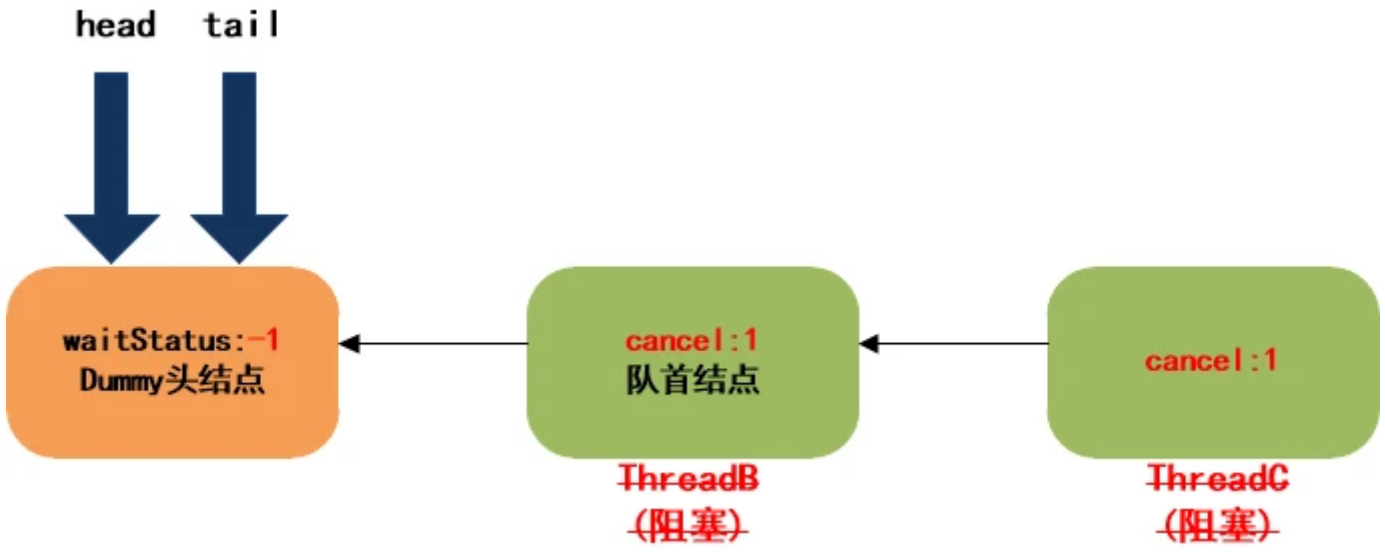
2. ThreadB先到超时时间

调用了**cancelAcquire**方法取消操作，队列状态变成：



3. ThreadC到达超时时间

调用了**cancelAcquire**方法取消操作，队列状态变成：



在退出**cancelAcquire**后，原来ThreadB和ThreadC对应的结点会被JVM垃圾回收器回收。

六、总结

本章从ReentrantLock入手，分析AQS的独占功能的内部实现细节。下一章，从CountDownLatch入手，看下AQS的共享功能如何实现。

java　多线程