线程这块的一些工具类，基本都会以原理为主，希望大家能有一个这样的意识，通过分析别人代码的设计和实现，给自己提供积累一些方法和工具。

Condition

在前面学习 synchronized 的时候，有讲到 wait/notify 的基本使用，结合 synchronized 可以实现对线程的通信。那么这个时候我就在思考了，既然 J.U.C 里面提供了锁的实现机制，那 J.U.C 里面有没有提供类似的线程通信的工具呢？ 于是找阿找，发现了一个 Condition 工具类。

Condition 是一个多线程协调通信的工具类，可以让某些线程一起等待某个条件（condition），只有满足条件时，线程才会被唤醒

Condition 的基本使用

ConditionWait

```
public class ConditionDemoWait implements  Runnable{

    private Lock lock;
    private Condition condition;
    public ConditionDemoWait(Lock lock, Condition condition){
        this.lock=lock;
        this.condition=condition;
    }

    @Override
    public void run() {
        System.out.println("begin -ConditionDemoWait");
        try {
            lock.lock();
            condition.await();
            System.out.println("end - ConditionDemoWait");
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }finally {
            lock.unlock();
        }
    }
}
```

ConditionSignal

```
public class ConditionDemoSignal implements  Runnable{

    private Lock lock;
    private Condition condition;
    public ConditionDemoSignal(Lock lock, Condition condition){
        this.lock=lock;
        this.condition=condition;
    }

    @Override
    public void run() {
        System.out.println("begin -ConditionDemoSignal");
        try {
            lock.lock();
            condition.signal();
            System.out.println("end - ConditionDemoSignal");
        }finally {
            lock.unlock();
        }
    }
}
```

通过这个案例简单实现了 wait 和 notify 的功能，当调用 await 方法后，当前线程会释放锁并等待，而其他线程调用 condition 对象的 signal 或者 signalall 方法通知并被阻塞的线程，然后自己执行 unlock 释放锁，被唤醒的线程获得之前的锁继续执行，最后释放锁。

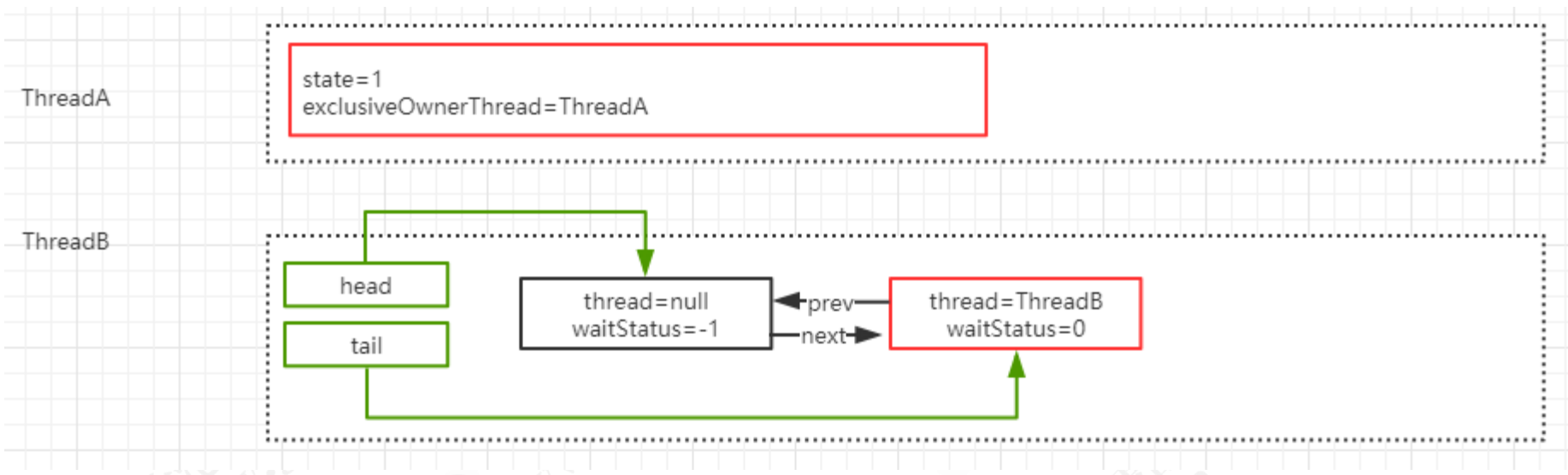
所以，condition 中两个最重要的方法，一个是 await，一个是 signal 方法

await:把当前线程阻塞挂起

signal:唤醒阻塞的线程

Condition 源码分析

调用 Condition，需要获得 Lock 锁，所以意味着会存在一个 AQS 同步队列，在上面那个案例中，假如两个线程同时运行的话，那么 AQS 的队列可能是下面这种情况



那么这个时候 ThreadA 调用了 condition.await 方法，它做了什么事情呢?

condition.await

调用 Condition 的 await()方法（或者以 await 开头的方法），会使当前线程进入等待队列并释放锁，同时线程状态变为等待状态。当从 await()方法返回时，当前线程一定获取了 Condition 相关联的锁

```
public final void await() throws InterruptedException {
    if (Thread.interrupted()) //表示 await 允许被中断
        throw new InterruptedException();
    Node node = addConditionWaiter(); //创建一个新的节点，节点状态为 condition，采用的数据结
    int savedState = fullyRelease(node); //释放当前的锁，得到锁的状态，并唤醒 AQS 队列中的一
    int interruptMode = 0;
    //如果当前节点没有在同步队列上，即还没有被 signal，则将当前线程阻塞
    while (!isOnSyncQueue(node)) {//判断这个节点是否在 AQS 队列上，第一次判断的是 false，因为
        LockSupport.park(this); //通过 park 挂起当前线程
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    // 当这个线程醒来,会尝试拿锁, 当 acquireQueued 返回 false 就是拿到锁了.
    // interruptMode != THROW_IE -> 表示这个线程没有成功将 node 入队,但 signal 执行了 enq
```

```
// 将这个变量设置成 REINTERRUPT.
if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
    interruptMode = REINTERRUPT;
// 如果 node 的下一个等待者不是 null, 则进行清理,清理 Condition 队列上的节点.
// 如果是 null ,就没有什么好清理的了.
if (node.nextWaiter != null) // clean up if cancelled
    unlinkCancelledWaiters();
// 如果线程被中断了,需要抛出异常.或者什么都不做
if (interruptMode != 0)
    reportInterruptAfterWait(interruptMode);
}
```

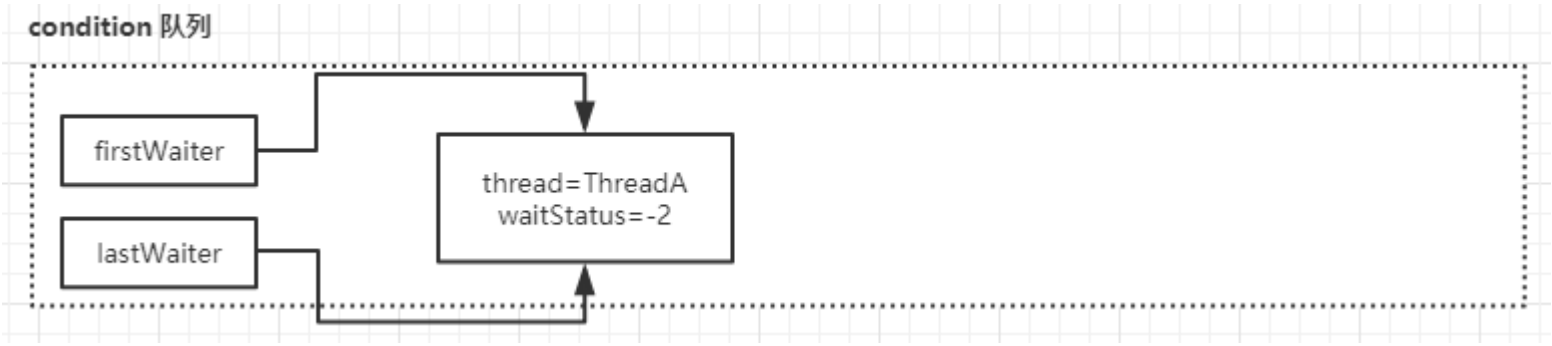
addConditionWaiter

这个方法的主要作用是把当前线程封装成 Node，添加到等待队列。这里的队列不再是双向链表，而是单向链表

```
private Node addConditionWaiter() {
    Node t = lastWaiter;
    // 如果 lastWaiter 不等于空并且 waitStatus 不等于 CONDITION 时，把冲处
    if (t != null && t.waitStatus != Node.CONDITION) {
        unlinkCancelledWaiters();
        t = lastWaiter;
    }
    //构建一个 Node，waitStatus=CONDITION。 这里的链表是一个单向的，所以
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    if (t == null)
        firstWaiter = node;
    else
        t.nextWaiter = node;
    lastWaiter = node;
    return node;
}
```

图解分析

执行完 addConditionWaiter 这个方法之后，就会产生一个这样的 condition 队列



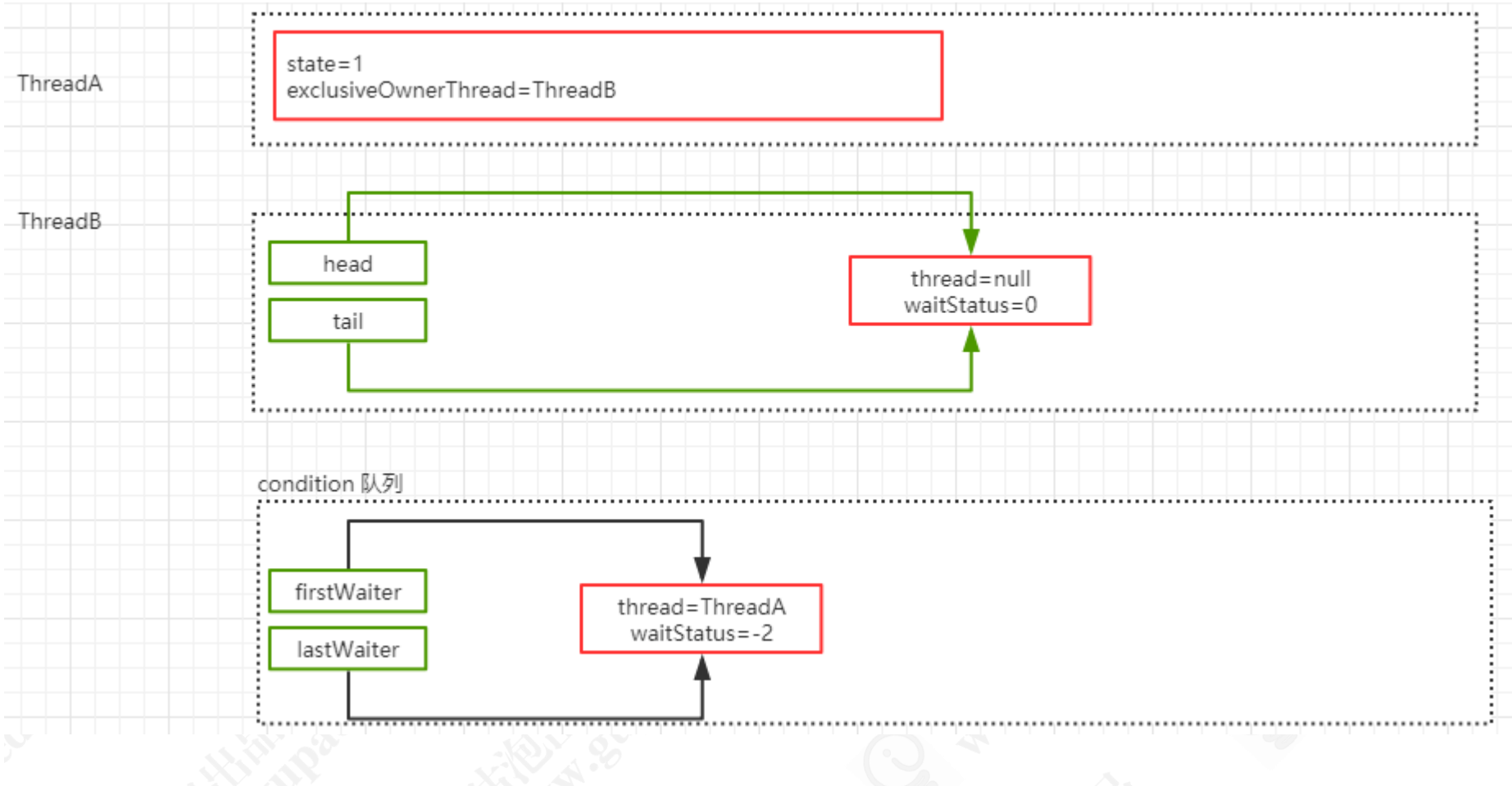
fullyRelease

fullRelease，就是彻底的释放锁，什么叫彻底呢，就是如果当前锁存在多次重入，那么在这个方法中只需要释放一次就会把所有的重入次数归零。

```
final int fullyRelease(Node node) {
    boolean failed = true;
    try {
        int savedState = getState(); //获得重入的次数
        if (release(savedState)) {//释放锁并且唤醒下一个同步队列中的线程
            failed = false;
            return savedState;
        } else {
            throw new IllegalMonitorStateException();
        }
    } finally {
        if (failed)
            node.waitStatus = Node.CANCELLED;
    }
}
```

图解分析

此时，同步队列会触发锁的释放和重新竞争。ThreadB 获得了锁。



isOnSyncQueue

判断当前节点是否在同步队列中，返回 false 表示不在，返回 true 表示在

如果不在 AQS 同步队列，说明当前节点没有唤醒去争抢同步锁，所以需要把当前线程阻塞起来，直到其他的线程调用 signal 唤醒

如果在 AQS 同步队列，意味着它需要去竞争同步锁去获得执行程序执行权限

为什么要做这个判断呢？原因是在 condition 队列中的节点会重新加入到 AQS 队列去竞争锁。也就是当调用 signal 的时候，会把当前节点从 condition 队列转移到 AQS 队列

- 大家思考一下，基于现在的逻辑结构。如何去判断 ThreadA 这个节点是否存在于 AQS 队列中呢?

1. 如果 ThreadA 的 waitStatus 的状态为 CONDITION，说明它存在于 condition 队列中，不在 AQS 队列。因为 AQS 队列的状态一定不可能有 CONDITION
2. 如果 node.prev 为空，说明也不存在于 AQS 队列，原因是 prev=null 在 AQS 队列中只有一种可能性，就是它是 head 节点，head 节点意味着它是获得锁的节点。
3. 如果 node.next 不等于空，说明一定存在于 AQS 队列中，因为只有 AQS 队列才会存在 next 和 prev 的关系
4. findNodeFromTail，表示从 tail 节点往前扫描 AQS 队列，一旦发现 AQS 队列的节点和当前节点相等，说明节点一定存在于 AQS 队列中

```
final boolean isOnSyncQueue(Node node) {
    if (node.waitStatus == Node.CONDITION || node.prev
== null)
        return false;
```

```
    if (node.next != null) // If has successor, it must
be on queue
        return true;

    return findNodeFromTail(node);
}
```

Condition.signal

await 方法会阻塞 ThreadA，然后 ThreadB 抢占到了锁获得了执行权限，这个时候在 ThreadB 中调用了 Condition 的 signal()方法，将会唤醒在等待队列中节点

```
public final void signal() {
    if (!isHeldExclusively()) //先判断当前线程是否获得了锁，这个判断比较简单，直接用获得锁的线程和当前线程相比即可
        throw new IllegalMonitorStateException();
    Node first = firstWaiter; // 拿到 Condition 队列上第一个节点
    if (first != null)
        doSignal(first);
}
```

Condition.doSignal

对 condition 队列中从首部开始的第一个 condition 状态的节点，执行 transferForSignal 操作，将 node 从 condition 队列中转换到 AQS 队列中，同时修改 AQS 队列中原先尾节点的状态

```
private void doSignal(Node first) {
    do {
        //从 Condition 队列中删除 first 节点
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null; // 将 next 节点设置成 null
        first.nextWaiter = null;
    } while (!transferForSignal(first) &&
             (first = firstWaiter) != null);
}
```

AQS.transferForSignal

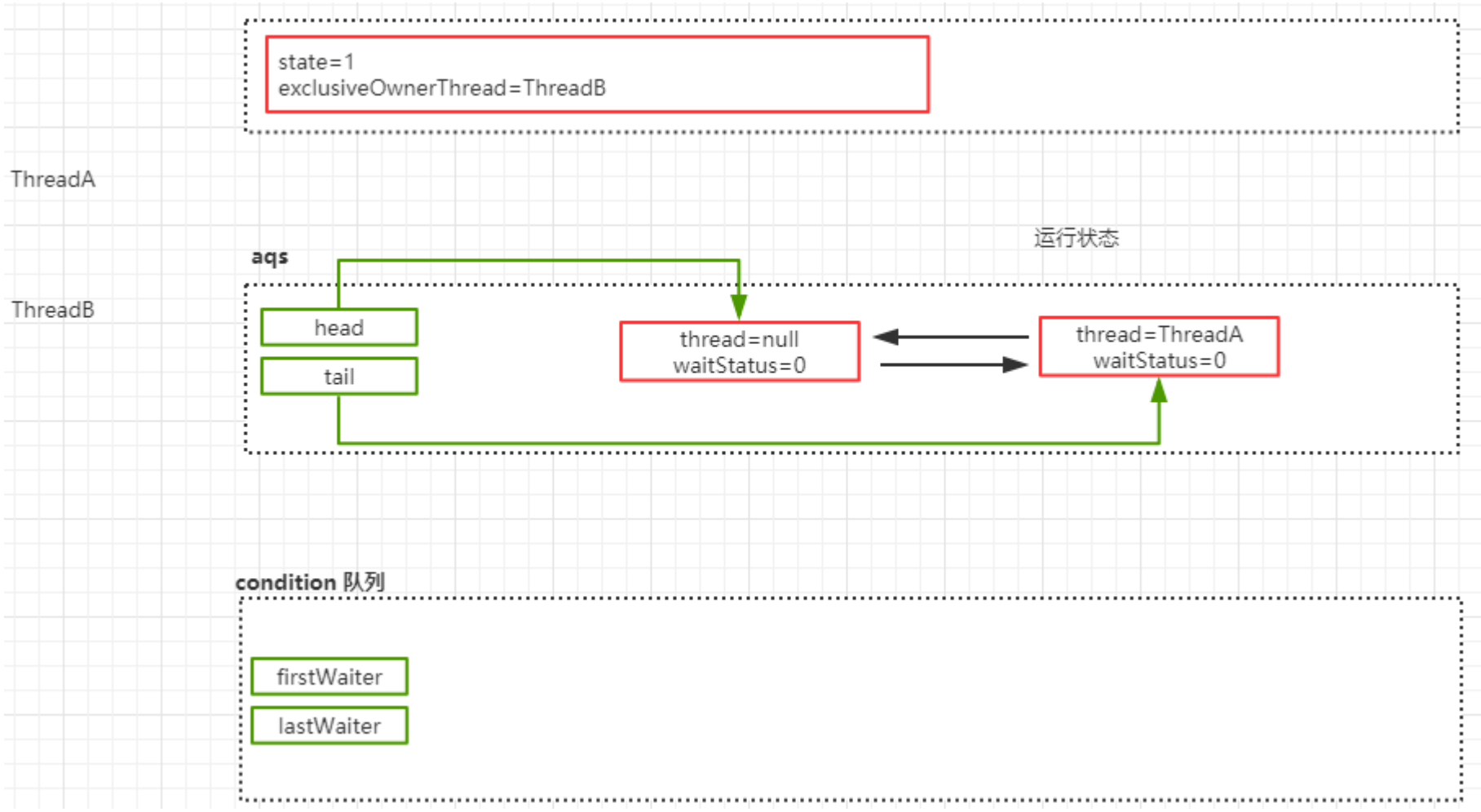
该方法先是 CAS 修改了节点状态，如果成功，就将这个节点放到 AQS 队列中，然后唤醒这个节点上的线程。此时，那个节点就会在 await 方法中苏醒

```
    final boolean transferForSignal(Node node) {
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))//更新节点的状态为0，如果更新失败，只有一种可能就是节点被 CANCELLED 了
        return false;
    Node p = enq(node);//调用 enq，把当前节点添加到 AQS 队列。并且返回返回按当前节点的上一个节点，也就是原 tail 节点
    int ws = p.waitStatus;
    // 如果上一个节点的状态被取消了，或者尝试设置上一个节点的状态为 SIGNAL 失败了(SIGNAL 表示: 他的 next 节点需要停止阻塞),
        if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
        LockSupport.unpark(node.thread); // 唤醒节点上的线程.
    return true; //如果 node 的 prev 节点已经是 signal 状态，那么被阻塞的 ThreadA 的唤醒工作由 AQS 队列来完成
}
```

图解分析

执行完 doSignal 以后，会把 condition 队列中的节点转移到 aqs 队列上，逻辑结构图如下

这个时候会判断 ThreadA 的 prev 节点也就是 head 节点的 waitStatus，如果大于 0 或者设置 SIGNAL 失败，表示节点被设置成了 CANCELLED 状态。这个时候会唤醒 ThreadA 这个线程。否则就基于 AQS 队列的机制来唤醒，也就是等到 ThreadB 释放锁之后来唤醒 ThreadA



被阻塞的线程唤醒后的逻辑

前面在分析 await 方法时，线程会被阻塞。而通过 signal 被唤醒之后又继续回到上次执行的逻辑中标注为红色部分的代码

checkInterruptWhileWaiting 这个方法是干嘛呢？其实从名字就可以看出来，就是 ThreadA 在 condition 队列被阻塞的过程中，有没有被其他线程触发过中断请求

```
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    Node node = addConditionWaiter();
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
        LockSupport.park(this);
        if ((interruptMode =
checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    if (acquireQueued(node, savedState) &&
interruptMode != THROW_IE)
```

```
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if
cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```

checkInterruptWhileWaiting

如果当前线程被中断，则调用 transferAfterCancelledWait 方法判断后续的处理应该是抛出 InterruptedException 还是重新中断。

这里需要注意的地方是，如果第一次 CAS 失败了，则不能判断当前线程是先进行了中断还是先进行了 signal 方法的调用，可能是先执行了 signal 然后中断，也可能是先执行了中断，后执行了 signal，当然，这两个操作肯定是发生在 CAS 之前。这时需要做的就是等待当前线程的 node 被添加到 AQS 队列后，也就是 enq 方法返回后，返回 false 告诉 checkInterruptWhileWaiting 方法返回 REINTERRUPT(1)，后续进行重新中断。

简单来说，该方法的返回值代表当前线程是否在 park 的时候被中断唤醒，如果为 true 表示中断在 signal 调用之前，signal 还未执行，那么这个时候会根据 await 的语义，在 await 时遇到中断需要抛出 interruptedException，返回 true 就是告诉 checkInterruptWhileWaiting 返回 THROW_IE(-1)。

如果返回 false，否则表示 signal 已经执行过了，只需要重新响应中断即可

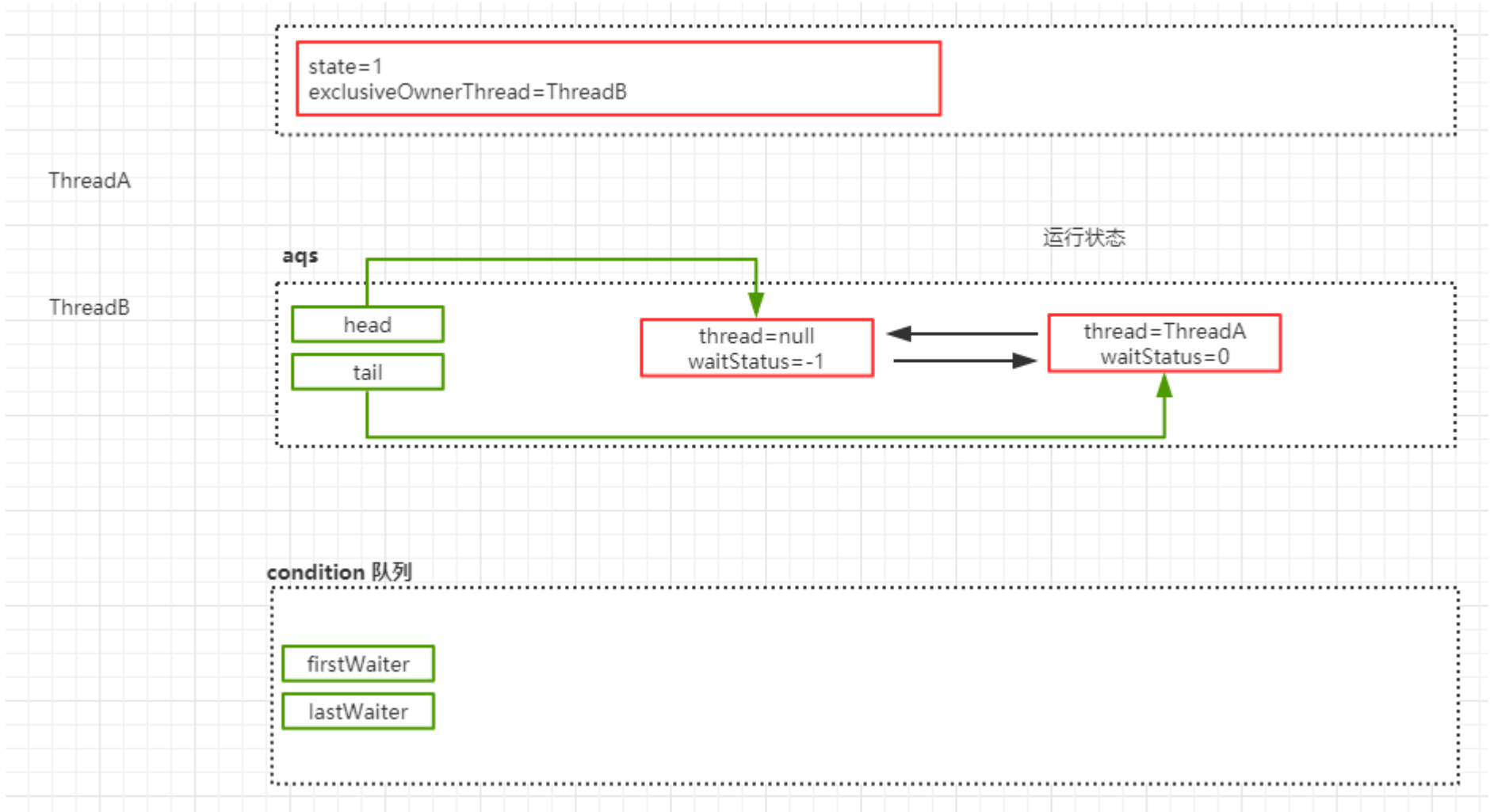
```
private int checkInterruptWhileWaiting(Node node) {
    return Thread.interrupted() ?
        (transferAfterCancelledWait(node) ? THROW_IE :
REINTERRUPT) :0;
}
final boolean transferAfterCancelledWait(Node node) {
//使用 cas 修改节点状态，如果还能修改成功，说明线程被中断时，
signal 还没有被调用。
```

```
// 这里有一个知识点，就是线程被唤醒，并不一定是在 java 层面执行了 locksupport.unpark，也可能是调用了线程的 interrupt()方法，这个方法会更新一个中断标识，并且会唤醒处于阻塞状态下的线程。
    if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {
        enq(node); //如果 cas 成功，则把 node 添加到 AQS 队列
        return true;
    }
  //如果 cas 失败，则判断当前 node 是否已经在 AQS 队列上，如果不在，则让给其他线程执行
    //当 node 被触发了 signal 方法时，node 就会被加到 aqs 队列上
    while (!isOnSyncQueue(node))//循环检测 node 是否已经成功添加到 AQS 队列中。如果没有，则通过 yield，
        Thread.yield();
    return false;
}
```

acquireQueued

这个方法在讲 aqs 的时候说过，是的当前被唤醒的节点 ThreadA 去抢占同步锁。并且要恢复到原本的重入次数状态。调用完这个方法之后，AQS 队列的状态如下

将 head 节点的 waitStatus 设置为-1，Signal 状态。



reportInterruptAfterWait

根据 checkInterruptWhileWaiting 方法返回的中断标识来进行中断上报。

如果是 THROW_IE，则抛出中断异常

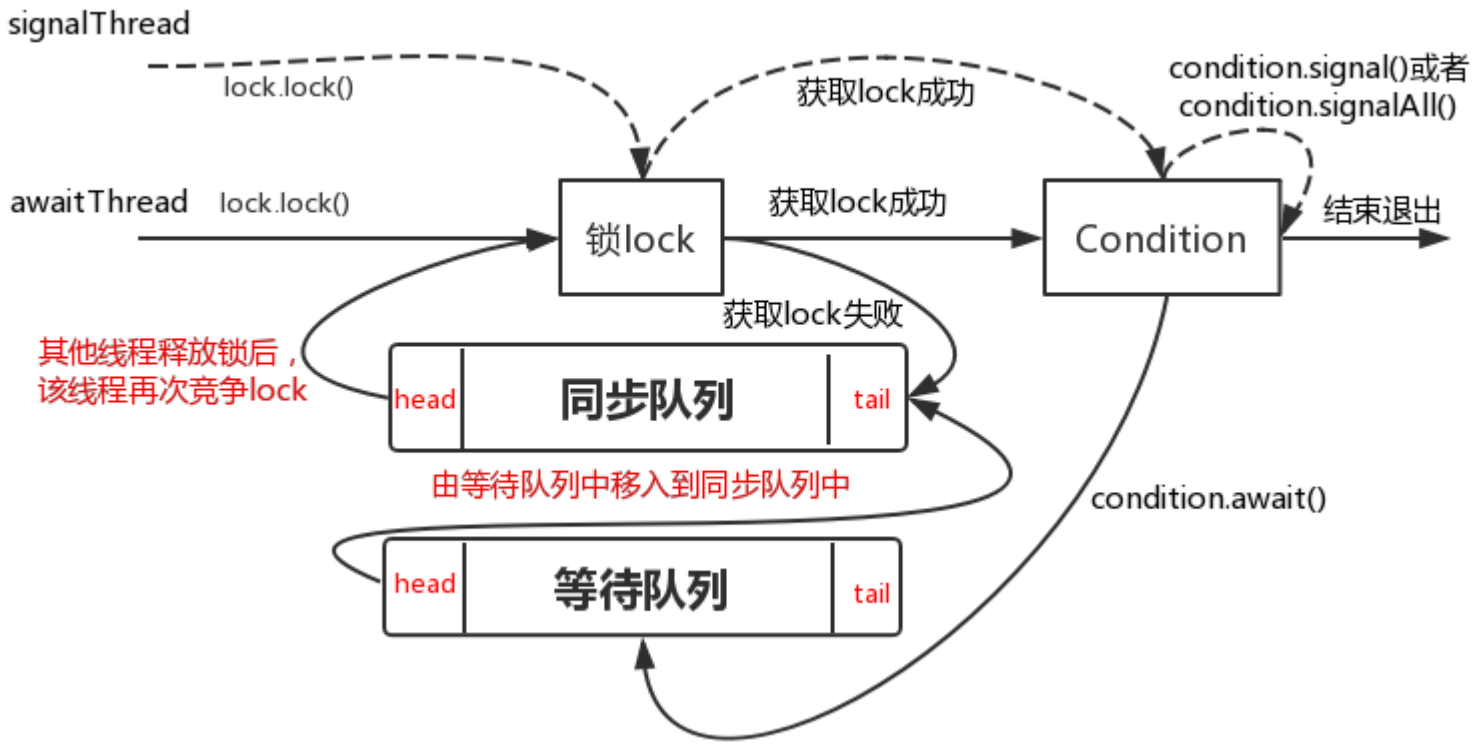
如果是 REINTERRUPT，则重新响应中断

```
private void reportInterruptAfterWait(int
interruptMode)
    throws InterruptedException {
    if (interruptMode == THROW_IE)
        throw new InterruptedException();
    else if (interruptMode == REINTERRUPT)
        selfInterrupt();
}
```

Condition 总结

await 和 signal 的总结

我把前面的整个分解的图再通过一张整体的结构图来表述，线程 awaitThread 先通过 lock.lock()方法获取锁成功后调用了 condition.await 方法进入等待队列，而另一个线程 signalThread 通过 lock.lock()方法获取锁成功后调用了 condition.signal 或者 signalAll 方法，使得线程 awaitThread 能够有机会移入到同步队列中，当其他线程释放 lock 后使得线程 awaitThread 能够有机会获取 lock，从而使得线程 awaitThread 能够从 await 方法中退出执行后续操作。如果 awaitThread 获取 lock 失败会直接进入到同步队列。



阻塞：await()方法中，在线程释放锁资源之后，如果节点不在 AQS 等待队列，则阻塞当前线程，如果在等待队列，则自旋等待尝试获取锁

释放：signal()后，节点会从 condition 队列移动到 AQS 等待队列，则进入正常锁的获取流程

- 了解完 Lock 以及 Condition 之后，意味着我们对于 J.U.C 里面的锁机制以及线程通信机制有了一个全面和深入的了解，接下来我们来看看其他比较常用的一些工具

限制

JUC 中提供了几个比较常用的并发工具类，比如 CountDownLatch、CyclicBarrier、Semaphore。其实在以前我们课堂的演示代码中，或多或少都有用到过这样一些 api，接下来我们会带大家去深入研究一些常用的 api。

CountDownLatch

countdownlatch 是一个同步工具类，它允许一个或多个线程一直等待，直到其他线程的操作执行完毕再执行。从命名可以解读到 countdown 是倒数的意思，类似于我们倒计时的概念。countdownlatch 提供了两个方法，一个是 countDown，一个是 await， countdownlatch 初始化的时候需要传入一个整数，在这个整数倒数到 0 之前，调用了 await 方法的程序都必须要等待，然后通过 countDown 来倒数。

使用案例

```
public static void main(String[] args) throws InterruptedException {
    CountDownLatch countDownLatch=new CountDownLatch(3);
```

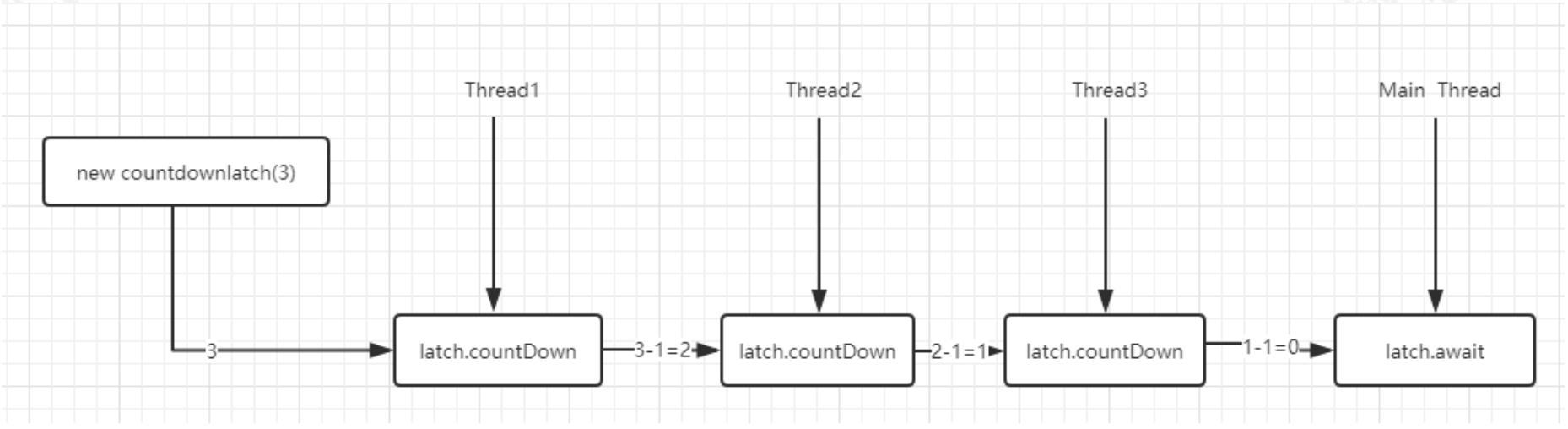
```
        new Thread(()->{
            System.out.println(""+Thread.currentThread().getName()+"-执行中");
            countDownLatch.countDown();
            System.out.println(""+Thread.currentThread().getName()+"-执行完毕");
        },"t1").start();

        new Thread(()->{
            System.out.println(""+Thread.currentThread().getName()+"-执行中");
            countDownLatch.countDown();
            System.out.println(""+Thread.currentThread().getName()+"-执行完毕");
        },"t2").start();

        new Thread(()->{
            System.out.println(""+Thread.currentThread().getName()+"-执行中");
            countDownLatch.countDown();
            System.out.println(""+Thread.currentThread().getName()+"-执行完毕");
        },"t3").start();
        countDownLatch.await();
        System.out.println("所有线程执行完毕");
    }
```

从代码的实现来看，有点类似 join 的功能，但是比 join 更加灵活。CountDownLatch 构造函数会接收一个 int 类型的参数作为计数器的初始值，当调用 CountDownLatch 的 countDown 方法时，这个计数器就会减一。

通过 await 方法去阻塞去阻塞主流程



模拟高并发场景

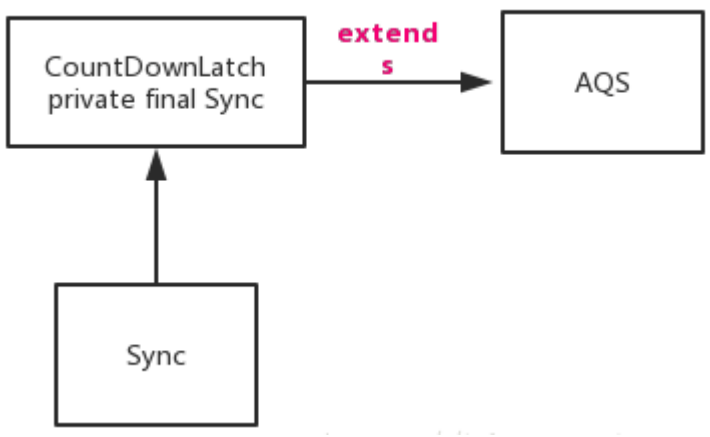
```
static CountDownLatch countDownLatch=new
CountDownLatch(1);
@Override
public void run() {
    try {
        countDownLatch.await();
        //TODO
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

System.out.println("ThreadName:"+Thread.currentThread().
getName());
}

public static void main(String[] args) throws
InterruptedException {
    for(int i=0;i<1000;i++){
        new Demo().start();
    }
    countDownLatch.countDown();
}
```

总的来说，凡事涉及到需要指定某个人物在执行之前，要等到前置人物执行完毕之后才执行的场景，都可以使用 CountDownLatch

CountDownLatch 源码分析



对于 CountDownLatch，我们仅仅需要关心两个方法，一个是 countDown() 方法，另一个是 await() 方法。

countDown() 方法每次调用都会将 state 减 1，直到 state 的值为 0；而 await 是一个阻塞方法，当 state 减为 0 的时候，await 方法才会返回。await 可以被多个线程调用，大家在这个时候脑子里要有个图：所有调用了 await 方法的线程阻塞在 AQS 的阻塞队列中，等待条件满足（state == 0），将线程从队列中一个个唤醒过来。

acquireSharedInterruptibly

countdownlatch 也用到了 AQS，在 CountDownLatch 内部写了一个 Sync 并且继承了 AQS 这个抽象类重写了 AQS 中的共享锁方法。首先看到下面这个代码，这块代码主要是判断当前线程是否获取到了共享锁；（在 CountDownLatch 中，使用的是共享锁机制，因为 CountDownLatch 并不需要实现互斥的特性）

```
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0) //state 如果不等于 0，说明当前线程需要加入到共享锁队列中
        doAcquireSharedInterruptibly(arg);
}
```

doAcquireSharedInterruptibly

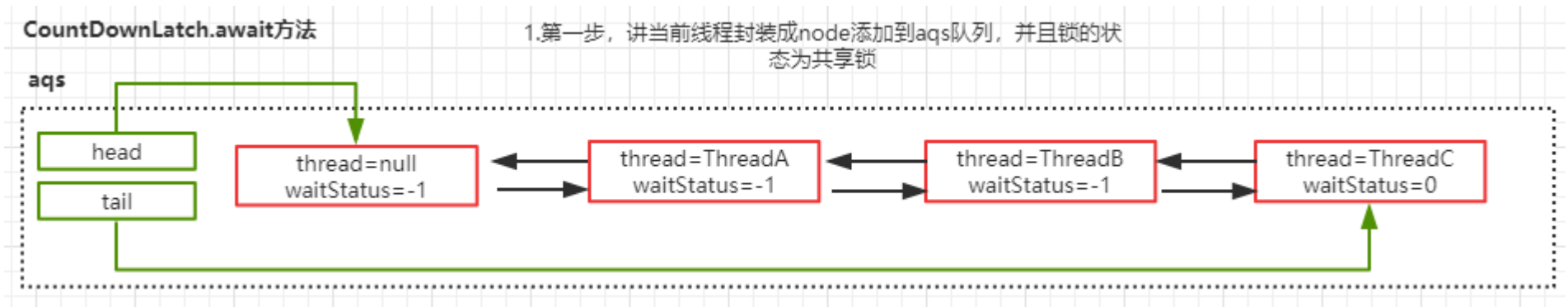
1. addWaiter 设置为 shared 模式。
2. tryAcquire 和 tryAcquireShared 的返回值不同，因此会多出一个判断过程
3. 在判断前驱节点是头节点后，调用了 setHeadAndPropagate 方法，而不是简单的更新一下头节点。

```
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED); //创建一个共享模式的节点添加到队列中
    boolean failed = true;
    try {
```

```
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);//就判断尝试获取锁
                if (r >= 0) {//r>=0 表示获取到了执行权限，这个时候因为 state!=0，所
以不会执行这段代码
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
            //阻塞线程
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

图解分析

加入这个时候有 3 个线程调用了 await 方法，由于这个时候 state 的值还不为 0，所以这三个线程都会加入到 AQS 队列中。并且三个线程都处于阻塞状态



CountDownLatch.countDown

由于线程被 await 方法阻塞了，所以只有等到 countdown 方法使得 state=0 的时候才会被唤醒，我们来看看 countdown 做了什么

1. 只有当 state 减为 0 的时候，tryReleaseShared 才返回 true, 否则只是简单的 state = state - 1
2. 如果 state=0，则调用 doReleaseShared 唤醒处于 await 状态下的线程

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

用自旋的方法实现 state 减 1

```
protected boolean tryReleaseShared(int releases) {
    // Decrement count; signal when transition to zero
    for (;;) {
        int c = getState();
        if (c == 0)
            return false;
        int nextc = c-1;
        if (compareAndSetState(c, nextc))
            return nextc == 0;
    }
}
```

AQS. doReleaseShared

共享锁的释放和独占锁的释放有一定的差别

前面唤醒锁的逻辑和独占锁是一样，先判断头结点是不是 SIGNAL 状态，如果是，则修改为 0，并且唤醒头结点的下一个节点

PROPAGATE： 标识为 PROPAGATE 状态的节点，是共享锁模式下的节点状态，处于这个状态下的节点，会对线程的唤醒进行传播

```
private void doReleaseShared() {
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {
                if (!compareAndSetWaitStatus(h,
Node.SIGNAL, 0))
                    continue;            // loop to recheck
cases
                unparkSuccessor(h);
            }
            // 这个 CAS 失败的场景是：执行到这里的时候，刚好有一个
节点入队，入队会将这个 ws 设置为 -1
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0,
Node.PROPAGATE))
                continue;                // loop on failed
CAS
        }
        // 如果到这里的时候，前面唤醒的线程已经占领了 head，那么再
循环
        // 通过检查头节点是否改变了，如果改变了就继续循环
        if (h == head)                   // loop if head
changed
            break;
    }
}
```

h == head：说明头节点还没有被刚刚用 unparkSuccessor 唤醒的线程（这里可以理解为 ThreadB）占有，此时 break 退出循环。

h != head：头节点被刚刚唤醒的线程（这里可以理解为 ThreadB）占有，那么这里重新进入下一轮循环，唤醒下一个节点（这里是 ThreadB ）。我们知道，等到 ThreadB 被唤醒后，其实是会主动唤醒 ThreadC...

doAcquireSharedInterruptibly

一旦 ThreadA 被唤醒，代码又会继续回到 doAcquireSharedInterruptibly 中来执行。如果当前 state 满足=0 的条件，则会执行 setHeadAndPropagate 方法

```
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        for (;;) {//被唤醒的线程进入下一次循环继续判断
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // 把当前节点移除 aqs 队列
                    failed = false;
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

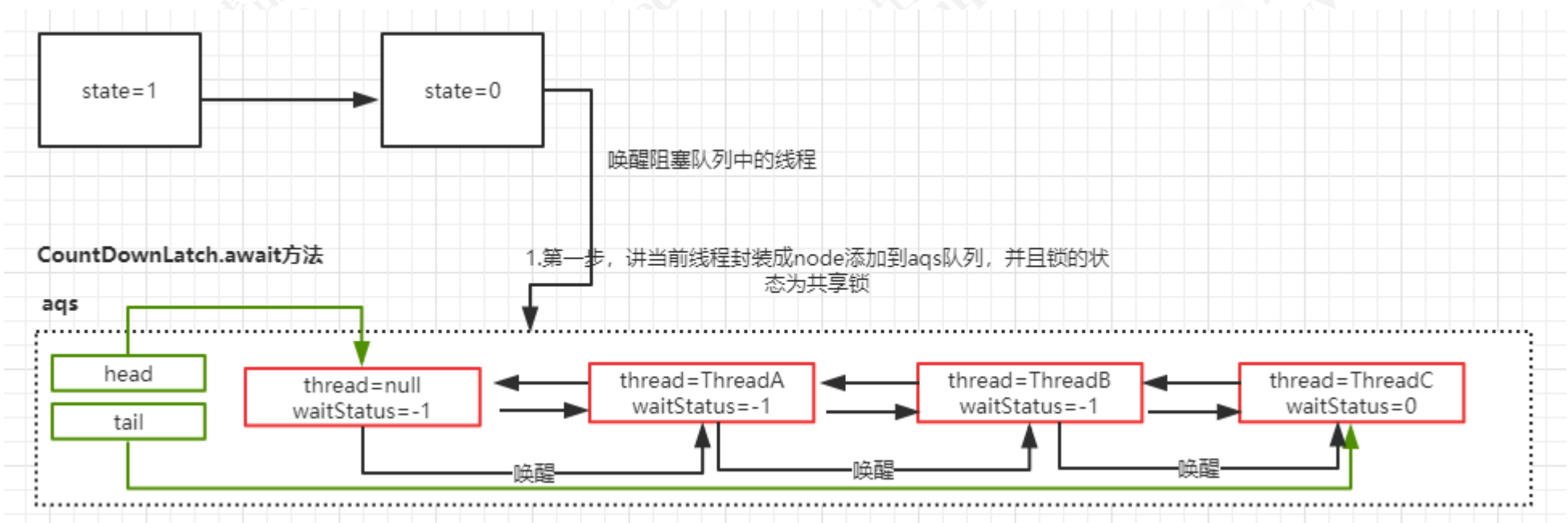
setHeadAndPropagate

这个方法的主要作用是把被唤醒的节点，设置成 head 节点。 然后继续唤醒队列中的其他线程。

由于现在队列中有 3 个线程处于阻塞状态，一旦 ThreadA 被唤醒，并且设置为 head 之后，会继续唤醒后续的 ThreadB

```
private void setHeadAndPropagate(Node node, int
propagate) {
    Node h = head; // Record old head for check below
    setHead(node);
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next;
        if (s == null || s.isShared())
            doReleaseShared();
    }
}
```

图解分析



Semaphore

semaphore 也就是我们常说的信号灯，semaphore 可以控制同时访问的线程个数，通过 acquire 获取一个许可，如果没有就等待，通过 release 释放一个许可。有点类似限流的作用。叫信号灯的原因也和他的用处有关，比如某商场就 5 个停车位，每个停车位只能停一辆车，如果这个时候来了 10 辆车，必须要等前面有空的车位才能进入。

使用案例

```
public class Test {
    public static void main(String[] args) {
        Semaphore semaphore=new Semaphore(5);
        for(int i=0;i<10;i++){
            new Car(i,semaphore).start();
        }
    }
    static class Car extends Thread{
        private int num;
        private Semaphore semaphore;

        public Car(int num, Semaphore semaphore) {
            this.num = num;
            this.semaphore = semaphore;
        }
        public void run(){
            try {
                semaphore.acquire();//获取一个许可
                System.out.println("第"+num+"占用一个停车位");
                TimeUnit.SECONDS.sleep(2);
                System.out.println("第"+num+"俩车走喽");
                semaphore.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

使用场景

Semaphore 比较常见的就是用来做限流操作了。

Semaphore 源码分析

从 Semaphore 的功能来看，我们基本能猜测到它的底层实现一定是基于 AQS 的共享所，因为需要实现多个线程共享一个领排池

创建 Semaphore 实例的时候，需要一个参数 permits，这个基本上可以确定是设置给 AQS 的 state 的，然后每个线程调用 acquire 的时候，执行 state = state - 1，release 的时候执行 state = state + 1，当然，acquire 的时候，如果 state = 0，说明没有资源了，需要等待其他线程 release。

Semaphore 分公平策略和非公平策略

FairSync

```
static final class FairSync extends Sync {
    private static final long serialVersionUID =
2014338818796000944L;

    FairSync(int permits) {
        super(permits);
    }

    protected int tryAcquireShared(int acquires) {
        for (;;) {
// 区别就在于是不是会先判断是否有线程在排队，然后才进行 CAS 减操
作
            if (hasQueuedPredecessors())
                return -1;
            int available = getState();
            int remaining = available - acquires;
            if (remaining < 0 ||
                compareAndSetState(available, remaining))
```

```
                return remaining;
            }
        }
}
```

NofairSync

通过对比发现公平和非公平的区别就在于是否多了一个 hasQueuedPredecessors 的判断

```
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = -
2694183684443567898L;

    NonfairSync(int permits) {
        super(permits);
    }

    protected int tryAcquireShared(int acquires) {
        return nonfairTryAcquireShared(acquires);
    }
}

final int nonfairTryAcquireShared(int acquires) {
    for (;;) {
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

由于后面的代码和 CountDownLatch 的是完全一样，都是基于共享锁的实现，所以也就没必要再花时间来分析了。

CyclicBarrier

CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续工作。CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 当前线程已经到达了屏障，然后当前线程被阻塞

使用场景

当存在需要所有的子任务都完成时，才执行主任务，这个时候就可以选择使用 CyclicBarrier

使用案例

DataImportThread

```
public class DataImportThread extends Thread{

    private CyclicBarrier cyclicBarrier;

    private String path;

    public DataImportThread(CyclicBarrier cyclicBarrier,
String path) {
        this.cyclicBarrier = cyclicBarrier;
        this.path = path;
    }

    @Override
    public void run() {
        System.out.println("开始导入: "+path+"位置的数据");
        try {
            cyclicBarrier.await();//阻塞
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
}
```

CycliBarrierDemo

```
public class CycliBarrierDemo extends Thread{

    @Override
    public void run() {
        System.out.println("开始进行数据分析");
    }

    public static void main(String[] args) {
        CyclicBarrier cycliBarrier=new
                CyclicBarrier(3,new CycliBarrierDemo());
        new Thread(new
DataImportThread(cycliBarrier,"file1")).start();
        new Thread(new
DataImportThread(cycliBarrier,"file2")).start();
        new Thread(new
DataImportThread(cycliBarrier,"file3")).start();
    }
}
```

注意点

1）对于指定计数值 parties，若由于某种原因，没有足够的线程调用 CyclicBarrier 的 await，则所有调用 await 的线程都会被阻塞；

2）同样的 CyclicBarrier 也可以调用 await(timeout, unit)，设置超时时间，在设定时间内，如果没有足够线程到达，则解除阻塞状态，继续工作；

3）通过 reset 重置计数，会使得进入 await 的线程出现 BrokenBarrierException；

4）如果采用是 CyclicBarrier(int parties, Runnable barrierAction) 构造方法，执行

barrierAction 操作的是最后一个到达的线程

实现原理

CyclicBarrier 相比 CountDownLatch 来说，要简单很多，源码实现是基于 ReentrantLock 和 Condition 的组合使用。看如下示意图，CyclicBarrier 和 CountDownLatch 是不是很像，只是 CyclicBarrier 可以有不止一个栅栏，因为它的栅栏（Barrier）可以重复使用（Cyclic）