AQS 自定义同步锁，挺难的！

原创 小霸戈 架构文摘 今天

`AQS` 是 `AbstractQueuedSynchronizer` 的简称。

AbstractQueuedSynchronizer 同步状态

`AbstractQueuedSynchronizer` 内部有一个 `state` 属性，用于指示同步的状态：

```
private volatile int state;
```

`state` 的字段是个 `int` 型的，它的值在 `AbstractQueuedSynchronizer` 中是没有具体的定义的，只有子类继承 `AbstractQueuedSynchronizer` 那么 `state` 才有意义，如在 `ReentrantLock` 中，`state=0` 表示资源未被锁住，而 `state>=1` 的时候，表示此资源已经被另外一个线程锁住。

`AbstractQueuedSynchronizer` 中虽然没有具体获取、修改 `state` 的值，但是它为子类提供一些操作 `state` 的模板方法：

获取状态

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

更新状态

```
    protected final void setState(int newState) {
        state = newState;
    }
```

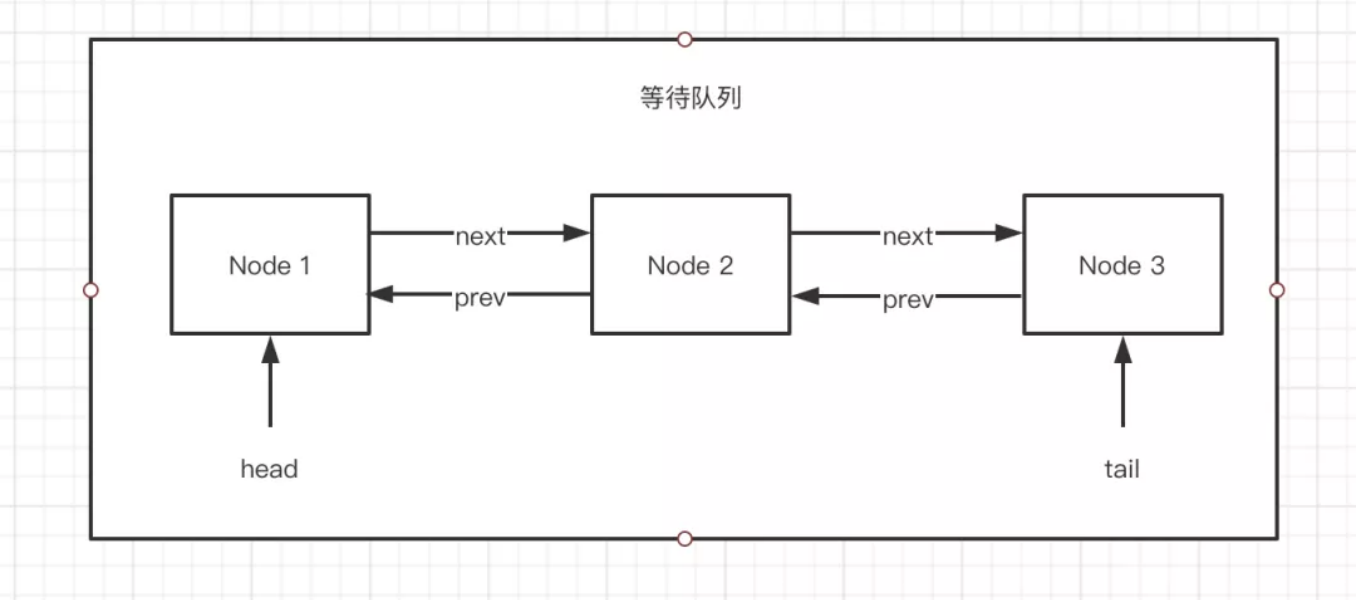
CAS更新状态

```
    protected final boolean compareAndSetState(int expect, int update) {
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
```

```
    }
```

AQS 等待队列

AQS 等待列队是一个双向队列，队列中的成员都有一个 `prev` 和 `next` 成员，分别指向它前面的节点和后面的节点。



队列节点

在 `AbstractQueuedSynchronizer` 内部，等待队列节点由内部静态类 `Node` 表示：

```
static final class Node {
    ...
}
```

节点模式

队列中的节点有两种模式：

- **独占节点**：同一时刻只能有一个线程访问资源，如 `ReentrantLock`
- **共享节点**：同一时刻允许多个线程访问资源，如 `Semaphore`

节点的状态

等待队列中的节点有五种状态：

- **CANCELLED**：此节点对应的线程，已经被取消
- **SIGNAL**：此节点的下一个节点需要一个唤醒信号

- **CONDITION**：当前节点正在条件等待
- **PROPAGATE**：共享模式下会传播唤醒信号，就是说当一个线程使用共享模式访问资源时，如果成功访问到资源，就会继续唤醒等待队列中的线程。

自定义同步锁

为了便于理解，使用AQS自己实现一个简单的同步锁，感受一下使用AQS实现同步锁是多么的轻松。

下面的代码自定了一个 `CustomLock` 类，继承了 `AbstractQueuedSynchronizer` ，并且还实现了 `Lock` 接口。
`CustomLock` 类是一个简单的可重入锁，类中只需要重写 `AbstractQueuedSynchronizer` 中的 `tryAcquire` 与 `tryRelease` 方法，然后在修改少量的调用就可以实现一个最基本的同步锁。

```
public class CustomLock extends AbstractQueuedSynchronizer implements Lock {

    @Override
    protected boolean tryAcquire(int arg) {

        int state = getState();
        if(state == 0){
            if( compareAndSetState(state, arg)){
                setExclusiveOwnerThread(Thread.currentThread());
                System.out.println("Thread: " + Thread.currentThread().getName() + "拿到
                return true;
            }
        }else if(getExclusiveOwnerThread() == Thread.currentThread()){
            int nextState = state + arg;
            setState(nextState);
            System.out.println("Thread: " + Thread.currentThread().getName() + "重入");
            return true;
        }

        return false;
    }

    @Override
    protected boolean tryRelease(int arg) {

        int state = getState() - arg;

        if(getExclusiveOwnerThread() != Thread.currentThread()){
            throw new IllegalMonitorStateException();
        }

        boolean free = false;
        if(state == 0){
            free = true;
            setExclusiveOwnerThread(null);
            System.out.println("Thread: " + Thread.currentThread().getName() + "释放了锁
        }
```

```
        setState(state);

        return free;
    }


    @Override
    public void lock() {
        acquire(1);
    }



    @Override
    public void unlock() {
        release(1);
    }
    ...
}
```

`CustomLock`是实现了`Lock`接口，所以要重写`lock`和`unlock`方法，不过方法的代码很少只需要调用AQS中的`acquire`和`release`。

然后为了演示AQS的功能写了一个小演示程序，启动两根线程，分别命名为`线程A`和`线程B`，然后同时启动，调用`runInLock`方法，模拟两条线程同时访问资源的场景：

```
public class CustomLockSample {

    public static void main(String[] args) throws InterruptedException {

        Lock lock = new CustomLock();
        new Thread(()->runInLock(lock), "线程A").start();
        new Thread(()->runInLock(lock), "线程B").start();
    }

    private static void runInLock(Lock lock){

        try {
            lock.lock();
            System.out.println("Hello: " + Thread.currentThread().getName());
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            lock.unlock();
        }

    }
}
```

访问资源（acquire）

在CustomLock的lock方法中，调用了 `acquire(1)` ， `acquire` 的代码如下：

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

- **CustomLock.tryAcquire(…)**： `CustomLock.tryAcquire` 判断当前线程是否能够访问同步资源
- **addWaiter(…)**：将当前线程添加到等待队列的队尾，当前节点为独占模型（Node.EXCLUSIVE）
- **acquireQueued(…)**：如果当前线程能够访问资源，那么就会放行，如果不能那当前线程就需要阻塞。
- **selfInterrupt**：设置线程的中断标记

注意： 在acquire方法中，如果tryAcquire(arg)返回true, 就直接执行完了，线程被放行了。所以的后面的方法调用acquireQueued、addWaiter都是tryAcquire(arg)返回false时才会被调用。

tryAcquire 的作用

`tryAcquire` 在AQS类中是一个直接抛出异常的实现：

```
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
```

而在我们自定义的 CustomLock 中，重写了此方法：

```
  @Override
    protected boolean tryAcquire(int arg) {

        int state = getState();
        if(state == 0){
            if( compareAndSetState(state, arg)){
                setExclusiveOwnerThread(Thread.currentThread());
                System.out.println("Thread: " + Thread.currentThread().getName() + "拿到
                return true;
            }
        }else if(getExclusiveOwnerThread() == Thread.currentThread()){
            int nextState = state + arg;
            setState(nextState);
            System.out.println("Thread: " + Thread.currentThread().getName() + "重入");
            return true;
        }

        return false;
    }
```

`tryAcquire`方法返回一个布而值，`true`表示当前线程能够访问资源，`false`当前线程不能访问资源，所以`tryAcquire`的作用：**决定线程是否能够访问受保护的资源。**`tryAcquire`里面的逻辑在子类可以自由发挥，AQS不关心这些，只需要知道能不能访问受保护的资源，然后来决定线程是放行还是进行等待队列（阻塞）。

因为是在多线程环境下执行，所以不同的线程执行`tryAcquire`时会返回不同的值，假设线程A比线程B要快一步，先到达`compareAndSetState`设置state的值成员并成功，那线程A就会返回true，而 B 由于state的值不为0或者`compareAndSetState`执行失败，而返回false。

线程B 抢占锁流程

上面访问到线程A成功获得了锁，那线程B就会抢占失败，接着执行后面的方法。

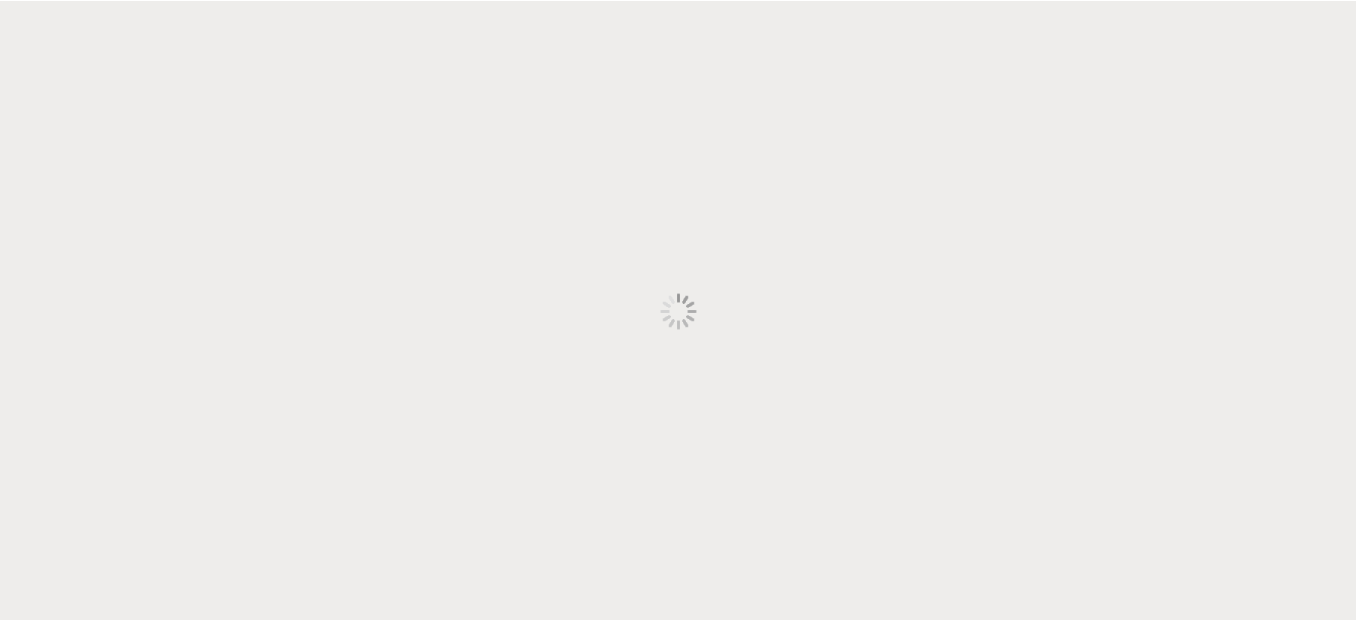
线程的入队

线程的入队是逻辑是在`addWaiter`方法中，addWaiter方法的具体逻辑也不需要说太多，如果你知道`链表`的话，就非常容易理解了，最终的结果就是将新线程添加到队尾。AQS的中有两个属性`head`、`tail`分别指定等待队列的队首和队尾。

```
private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
        Node pred = tail;
        if (pred != null) {
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        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;
                }
            }
        }
    }
```

需要注意的是在 `enq` 方法中，**初始化队列的时候，会新建一个 `Node` 做为 `head` 和 `tail`**，然后在之后的循环中将参数 `node` 添加到队尾，**队列初始化完后，里面会有两个节点**，一个是空的结点 `new Node()` 另外一个就是对应当前线程的结点。

由于线程A在 `tryAcquire` 时返回了 `true`，所以它会被直接放行，那么只有B线程会进入 `addWaiter` 方法，此时的等待队列如下：



注意： 等待队列内的节点都是正在等待资源的线程，如果一个线程直接能够访问资源，那它压根就不需要进入等待队列，会被放行。

线程B 的阻塞

线程B被添加到等待队列的尾部后，会继续执行 `acquireQueued` 方法，这个方法就是AQS阻塞线程的地方， `acquireQueued` 方法代码的一些解释：

- 外面是一个 `for (;;)` 无限循环，这个很重要
- 会重新调用一次 `tryAcquire(arg)` 判断线程是否能够访问资源了
- `node.predecessor()` 获取参数 `node` 的前一个节点
- `shouldParkAfterFailedAcquire` 判断当前线程获取锁失败后，需不需要阻塞
- `parkAndCheckInterrupt()` 使用 `LockSupport` 阻塞当前线程，

```
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);
        }
    }
```

shouldParkAfterFailedAcquire 判断是否要阻塞

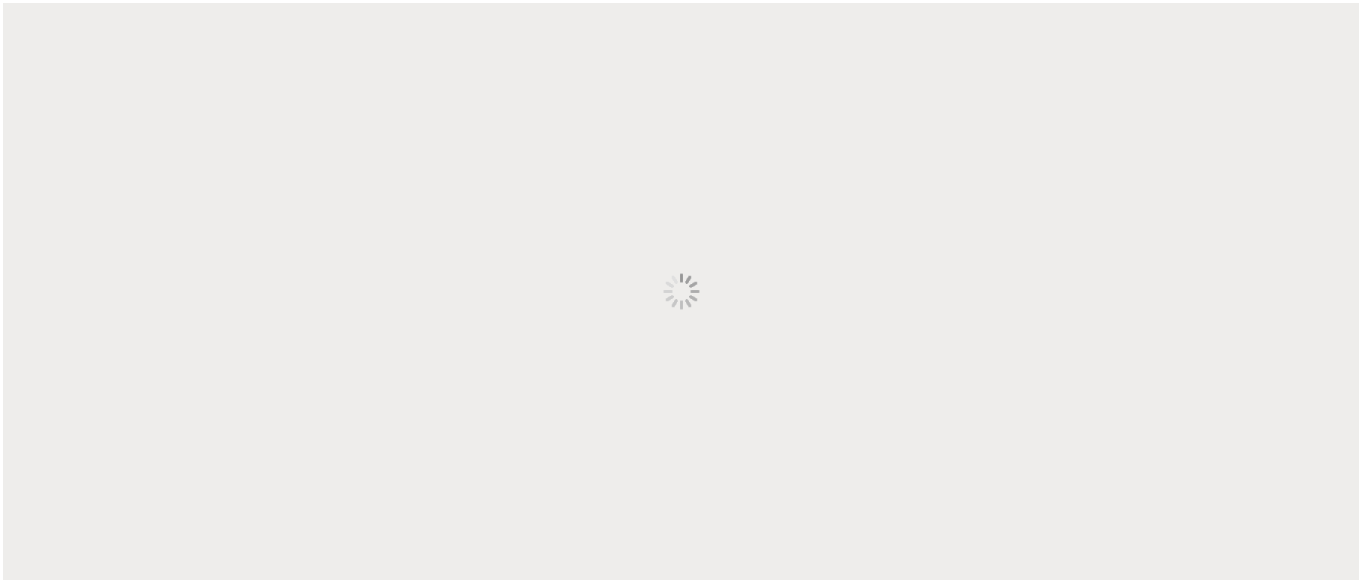
`shouldParkAfterFailedAcquire` 接收两个参数：前一个节点、当前节点，它会判断**前一个节点的** `waitStatus` **属性**，如果前一个节点的 `waitStatus=Node.SIGNAL` 就会返回true：

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
            return true;
        if (ws > 0) {
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

`acquireQueued` 方法在循环中会多次调用 `shouldParkAfterFailedAcquire` ，在等待队列中节点的 `waitStatus` 的属性默认为0，所以第一次执行 `shouldParkAfterFailedAcquire` 会执行：

```
compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
```

更新完 `pred.waitStatus` 后，节点的状态如下：



然后 `shouldParkAfterFailedAcquire` 返回false，回到 `acquireQueued` 的循环体中，又去抢锁还是失败了，又会执行 `shouldParkAfterFailedAcquire` ，第二次循环时此时的 `pred.waitStatus` 等于 `Node.SIGNAL` 那么就会返回true。

parkAndCheckInterrupt 阻塞线程

这个方法就比较直观了， 就是将线程的阻塞住：

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

为什么是一个`for (;;)`无限循环呢

先看一个 `for (;;)` 的退出条件，**只有 `node` 的前一个节点是 `head` 并且tryAcquire返回true时才会退出循环**，否则的话线程就会被 `parkAndCheckInterrupt` 阻塞。

线程被 `parkAndCheckInterrupt` 阻塞后就不会向下面执行了，但是等到它被唤醒后，它还在 `for (;;)` 体中，然后又会继续先去抢占锁，然后如果还是失败，那又会处于等待状态，所以一直循环下去，就只有两个结果：

1. 抢到锁退出循环
2. 抢占锁失败，等待下一次唤醒再次抢占锁

线程 A 释放锁

线程A的业务代码执行完成后，会调用 `CustomLock.unlock` 方法，释放锁。unlock方法内部调用的 `release(1)` ：

```
    public void unlock() {
        release(1);
    }
```

`release` 是AQS类的方法，它跟 `acquire` 相反是释放的意思：

```
    public final boolean release(int arg) {
        if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
        return false;
    }
```

方法体中的 `tryRelease` 是不是有点眼熟，没错，它也是在实现 `CustomLock` 类时重写的方法，首先在 `tryRelease` 中会判断当前线程是不是已经获得了锁，如果没有就直接抛出异常，否则的话计算state的值，如果state为0的话就可以释放锁了。

```
protected boolean tryRelease(int arg) {

        int state = getState() - arg;

        if(getExclusiveOwnerThread() != Thread.currentThread()){
            throw new IllegalMonitorStateException();
        }

        boolean free = false;
        if(state == 0){
            free = true;
            setExclusiveOwnerThread(null);
            System.out.println("Thread: " + Thread.currentThread().getName() + "释放了锁
        }

        setState(state);

        return free;
    }
```

`release` 方法只做了两件事：

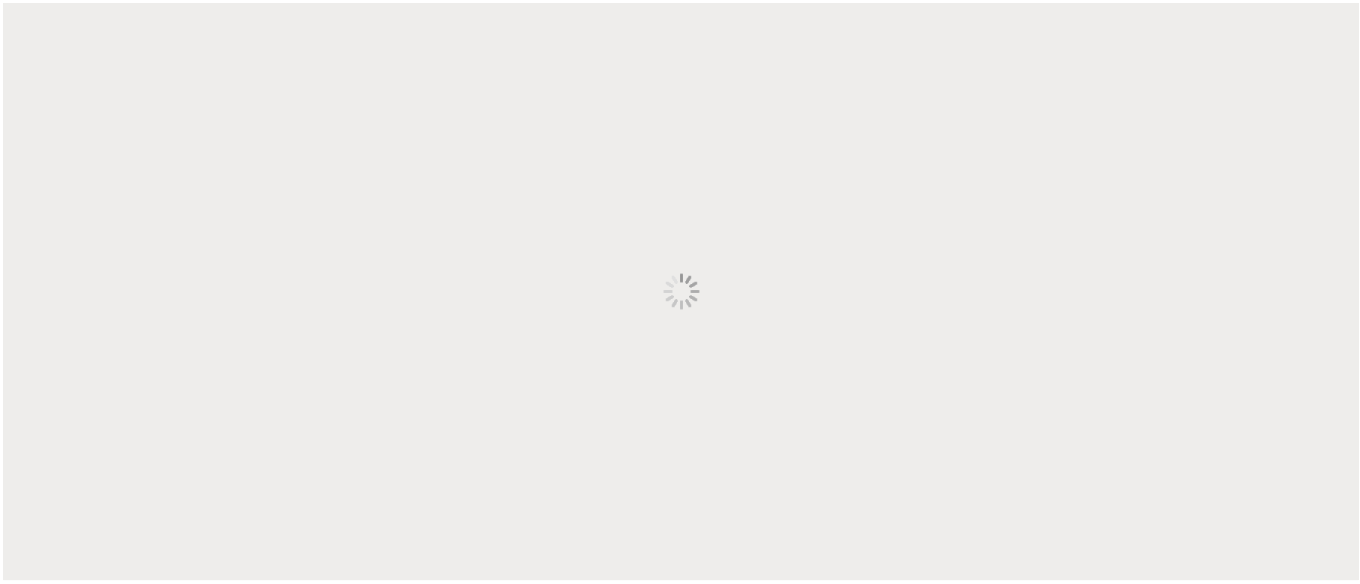
1. 调用 `tryRelease` 判断当前线程释放锁是否成功
2. 如果当前线程锁释放锁成功，唤醒其他线程（也就是正在等待中的B线程）

`tryRelease` 返回true后，会执行if里面的代码块：

```
if (tryRelease(arg)) {
            Node h = head;
```

```
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
```

先回顾一下现在的等待队列的样子：



根据上面的图，来走下流程：

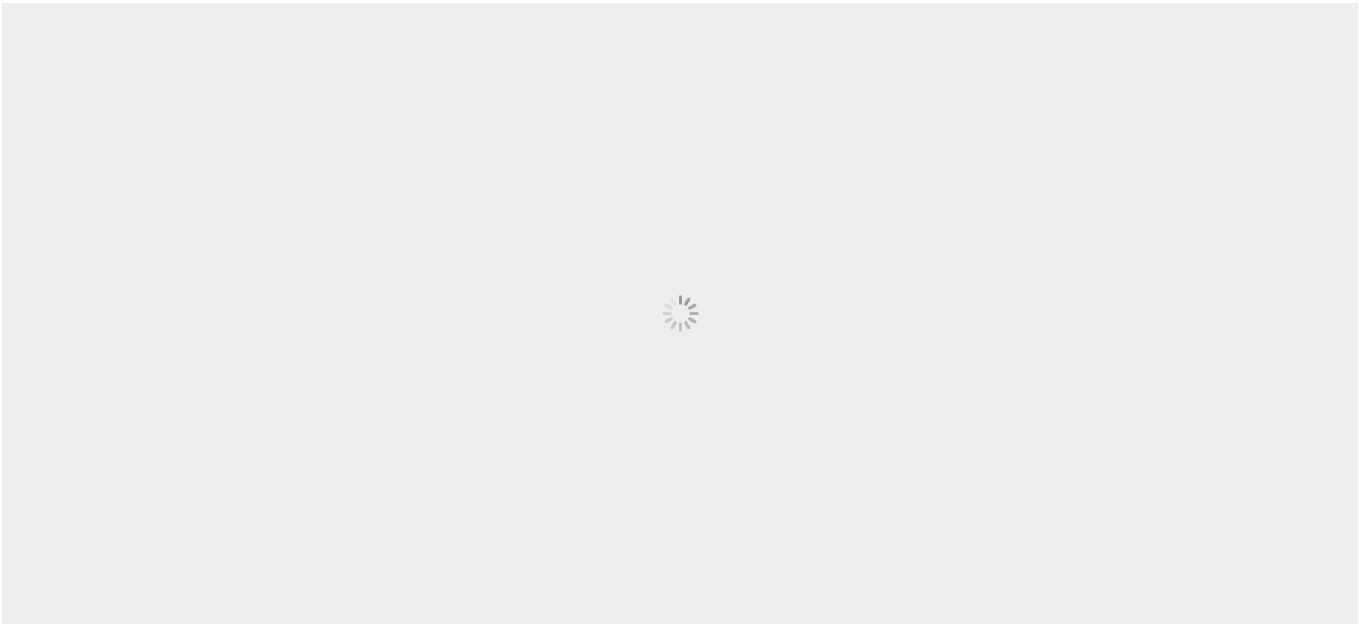
- 首先拿到 `head` 属性的对象，也就是队列的第一个对象
- 判断 `head` 不等于空，并且waitStatus!=0，很明显现在的waitStatus是等于 `Node. SIGNAL` 的，它的值是-1

所以 `if (h != null && h.waitStatus != 0)` 这个if肯定是满足条件的，接着执行 `unparkSuccessor(h)`：

```
    private void unparkSuccessor(Node node) {
        int ws = node.waitStatus;
        if (ws < 0)
            compareAndSetWaitStatus(node, ws, 0);

        Node s = node.next;

        ...

        if (s != null)
            LockSupport.unpark(s.thread);
    }
```

`unparkSuccessor` 首先将 `node.waitStatus` 设置为0，然后**获取node的下一个节点**，最后调用 `LockSupport.unpark(s.thread)` 唤醒线程，至此我们的B线程就被唤醒了。

此时的队列又回到了，线程B刚刚入队的样子：



线程B 唤醒之后

线程A释放锁后，会唤醒线程B，回到线程B的阻塞点， `acquireQueued` 的for循环中：

```
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);
      }
  }
```

线程唤醒后的第一件事就是，拿到它的上一个节点（当前是head结点），然后使用if判断

```
if (p == head && tryAcquire(arg))
```

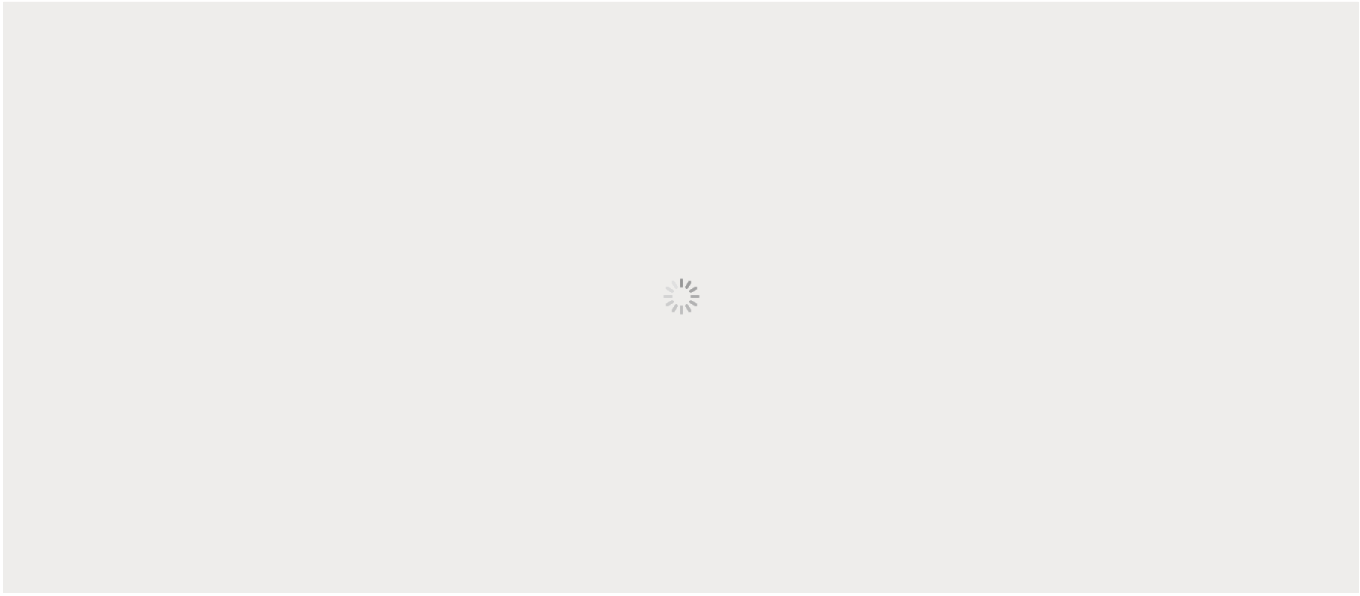
根据现在等待队列中的节点状态， `p == head` 是返回true的，然后就是 `tryAcquire(arg)` 了，由于线程A已经释放了锁，那现在的线程B自然就能获取到锁了，所以tryAcquire(arg)也会返回true。

设置队列头

线路B拿到锁后，会调用`setHead(node)`自己设置为队列的头：

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

调用`setHead(node)`后队列会发生些变化：

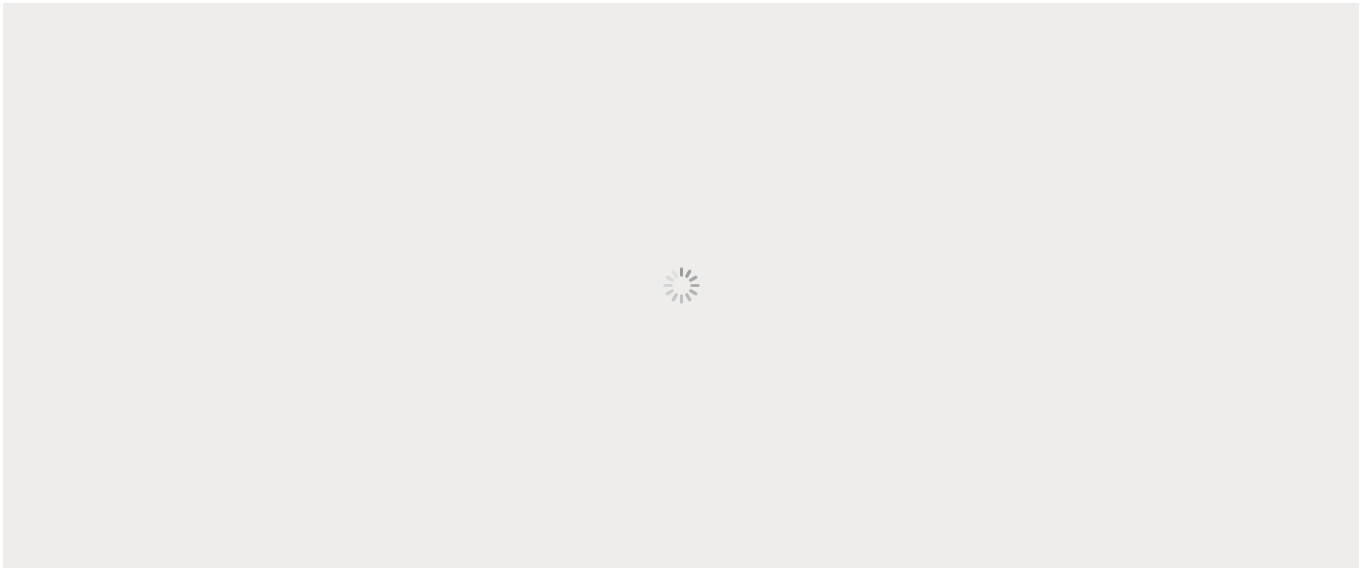


移除上一个节点

`setHead(node)`执行完后，接着按上一个节点完全移除：

```
p.next = null;
```

此时的队列：



线程B 释放锁

线程B 释放锁的流程与线程A基本一致，只是当前队列中已经没有需要唤醒的线程，所以不需要执行代码去唤醒其他线程：

```
if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0)s
                unparkSuccessor(h);
            return true;
        }
```

`h != null && h.waitStatus != 0` 这里的 `h.waitStatus` 已经是0了，不满足条件，不会去唤醒其他线程。

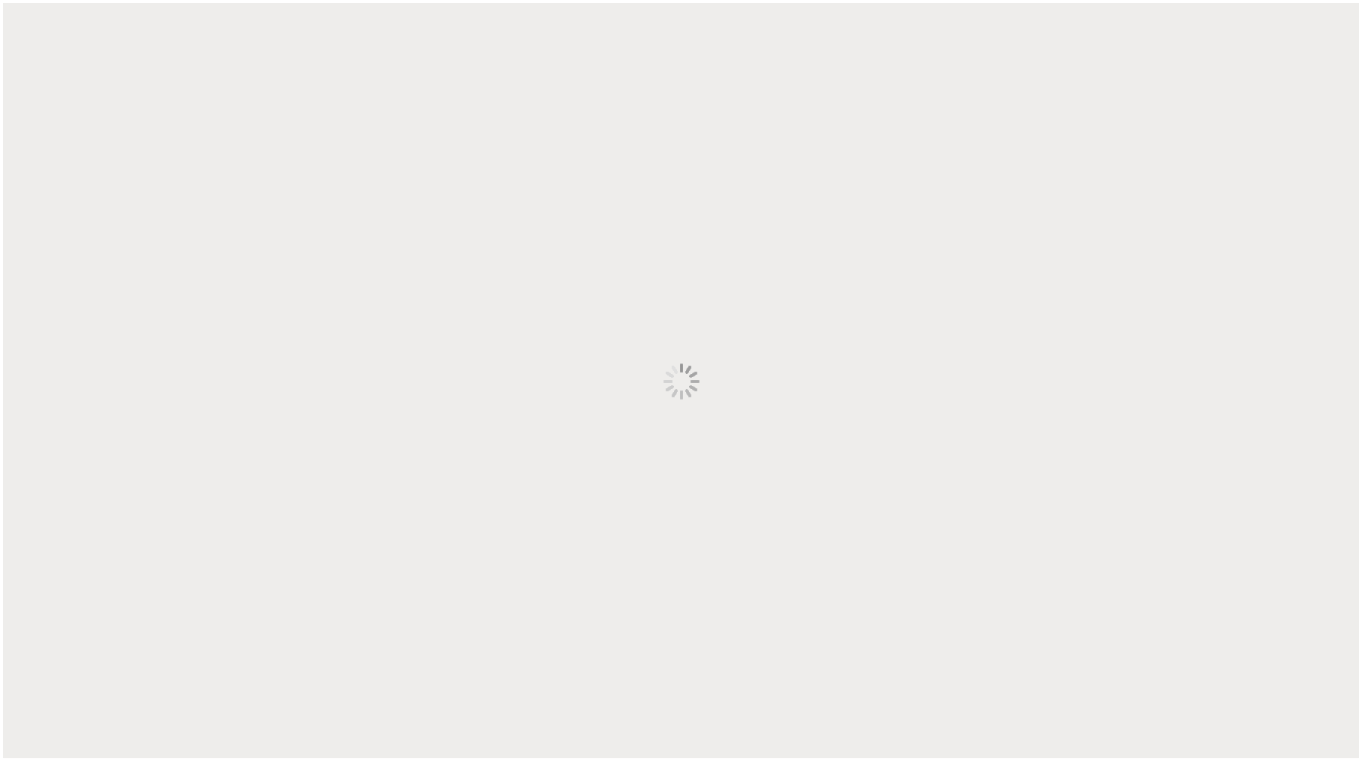
总结

文中通过自定义一个 `CustomLock` 类，然后通过查看AQS源码来学习AQS的部分原理。通过完整的走完锁的获取、释放两个流程，加深对AQS的理解，希望对大家有所帮助。

end

推荐阅读：

- TCP 三次握手、四手挥手，这样说你能明白吧！
- 拜托，不要再问我线程池啦！
- 为什么 Redis 单线程还这么快？
- Spring Cloud架构的各个组件的原理分析
- 手把手教你使用 OpenResty 搭建高性能服务端！



如有收获，点个在看，诚挚感谢