面试难点:深度解析ReentrantLock的实现原理

程序员面试 6月28日

以下文章来源于后端开发技术，作者阿笠在健身

什么是Reentrant

Jdk中独占锁的实现除了使用关键字synchronized外,还可以使用ReentrantLock。虽然在性能上ReentrantLock和synchronized没有什么区别，但ReentrantLock相比synchronized而言功能更加丰富，使用起来更为灵活，也更适合复杂的并发场景。

Synchronized和ReentrantLock的相同点:

1.ReentrantLock和synchronized都是独占锁,只允许线程互斥的访问临界区。但是实现上两者不同:synchronized加锁解锁的过程是隐式的,用户不用手动操作,优点是操作简单，但显得不够灵活。一般并发场景使用synchronized的就够了；ReentrantLock需要手动加锁和解锁,且解锁的操作尽量要放在finally代码块中,保证线程正确释放锁。ReentrantLock操作较为复杂，但是因为可以手动控制加锁和解锁过程,在复杂的并发场景中能派上用场。

2.ReentrantLock和synchronized都是可重入的。synchronized因为可重入因此可以放在被递归执行的方法上,且不用担心线程最后能否正确释放锁；而ReentrantLock在重入时要却确保重复获取锁的次数必须和重复释放锁的次数一样，否则可能导致其他线程无法获得该锁。

Synchronized和ReentrantLock的不同点:

1.ReentrantLock是Java层面的实现，synchronized是JVM层面的实现。
2.ReentrantLock可以实现公平和非公平锁。
3.ReentantLock获取锁时，限时等待，配合重试机制更好的解决死锁
4.ReentrantLock可响应中断
5.使用synchronized结合Object上的wait和notify方法可以实现线程间的等待通知机制。ReentrantLock结合Condition接口同样可以实现这个功能。而且相比前者使用起来更清晰也更简单。

源码阅读

简要总结

简要总结

由于篇幅较长，本人写这篇文章用了6个小时，总结为三个词。
实现原理：volatile 变量 + CAS设置值 + AQS

ReentrantLock锁的实现步骤总结为三点
1、未竞争到锁的线程将会被CAS为一个链表结构并且被挂起。
2、竞争到锁的线程执行完后释放锁并且将唤醒链表中的下一个节点。
3、被唤醒的节点将从被挂起的地方继续执行逻辑。

基础用法

先写一个demo，方便下面的源码解读。核心方法有两个 lock() 和 unlock()。

```java
public class Demo {
    private int num;
    private static final Lock LOCK = new ReentrantLock();
    public void add() {
        try {
            LOCK.lock();
            num++;
        } finally {
            LOCK.unlock();
         }
     }
}

```

lock()方法

基础用法

lock是用来加锁的方法，代码如下。

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

public class ReentrantLock implements Lock, java.io.Serializable {
        private final Sync sync;
        static final class FairSync extends Sync {
    }
    static final class NonfairSync extends Sync {
    }
    abstract static class Sync extends AbstractQueuedSynchronizer {
    }
}
```

由上图源码可以看出很多信息，ReentrantLock继承了Lock接口， lock方法实际上是调用了Sync的子类NonfairSync(非公平锁)的lock方法。ReentrantLock的真正实现在他的两个内部类NonfairSync 和 FairSync中，并且内部类都继承于内部类Sync，而Sync根本的实现则是大名鼎鼎的AbstractQueuedSynchronizer同步器（AQS）。

```java
final void lock() {
    //CAS操作尝试获取锁
    if (compareAndSetState(0, 1))
    //设置当前线程为持有锁线程
        setExclusiveOwnerThread(Thread.currentThread());
    else
    //获取锁失败
        acquire(1);
}
protected final void setExclusiveOwnerThread(Thread thread) {
    exclusiveOwnerThread = thread;
}
```

lock方法首先执行compareAndSetState，而该方法实际上就是AQS中的一个方法，这个方法最终会调用unsafe的一个CAS操作，线程安全的改变state为1，独占锁。（实际就是类似于乐观锁的操作，比较版本号是否与预期版本号相同，如果相同设置给定值并返回成功标识，如果不同则返回失败标识，关于CAS的详解在之前文章中有）。

compareAndSetState方法则是判断AbstractQueuedSynchronizer中的state值是否为0，如果为0，就修改为1，并返回true。state初始值为0，修改成功调用AQS父类AbstractOwnableSynchronizer的方法setExclusiveOwnerThread(Thread.currentThread())方法将当前独占锁线程设置为当前线程。线程抢锁成功。

如果此时其他线程也调用了lock方法，执行compareAndSetState方法失败，因为此时的state不为0，于是执行acquire方法。

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

protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```

```java
final boolean nonfairTryAcquire(int acquires) {
    //获得当前线程饮用
    final Thread current = Thread.currentThread();
    int c = getState();
    //获得state状态 再次尝试获取锁
    if (c == 0) {
        //获取锁成功
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //当前线程即为持有锁线程
    else if (current == getExclusiveOwnerThread()) {
        //重新计算state 这段代码表明ReentranLock是可重入锁
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
            //更新state
        setState(nextc);
        return true;
    }
    //获取锁失败
    return false;
    }
```

进入acquire 后执行 tryAcquire方法，该方法是AQS中的一个抽象方法，具体实现由子类NonfairSync实现，最终调用的是父类Sync的nonfairTryAcquire方法。

如代码中注释，这段代码说明ReentrantLock是重入锁。这个方法的意思就是在取尝试重试获取一次锁，因为有可能其他占有锁的线程很快就释放锁了，这里在次尝试了一次，如果获取到了，就直接返回true。如果失败继续会判断当前独占锁的线程和当前线程是否为同一个线程（重入锁的实现），如果是，将state设置为state+1，并且反回true，而这里他们不相等，当前独占锁的线程为线程A,当前线程为B,所以结构会返回false。!tryAcquire(arg)则为true,所以会继续执行同步器的addWaiter(Node.EXCLUSIVE)方法。

Node的代码如下：

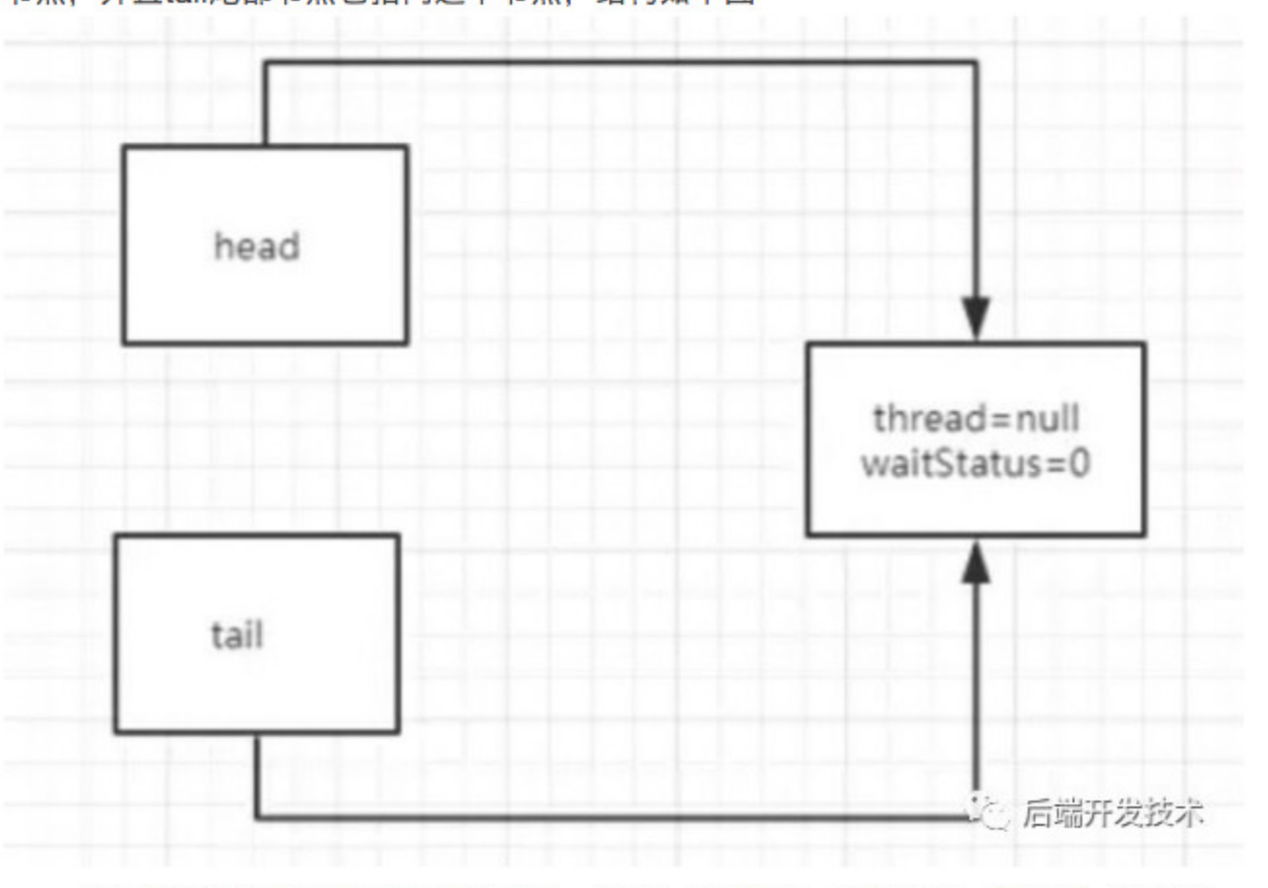
```java
static final class Node {
    /** Marker to indicate a node is waiting in shared mode */
    表明是共享锁
    static final Node SHARED = new Node();
    //表明是独占锁
    /** Marker to indicate a node is waiting in exclusive mode */
    static final Node EXCLUSIVE = null;//独占
    /** waitStatus value to indicate thread has cancelled */
    static final int CANCELLED =  1;//取消
     /** waitStatus value to indicate successor's thread needs unparking */
     static final int SIGNAL    = -1;//表面后即线程需要unpark
     /** waitStatus value to indicate thread is waiting on condition */
     static final int CONDITION = -2;//表明线程在等待condition条件
     static final int PROPAGATE = -3;//表明会向后面节点广播共享状态
     volatile int waitStatus;
     volatile Node prev;
     volatile Node next;
     volatile Thread thread;
     Node nextWaiter;

    }
```

```java
private Node addWaiter(Node mode) {
    //保存当前线程到node
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    //尾节点不为空 将当前节点放入链表尾部
    if (pred != null) {
        node.prev = pred;
        //通过CAS更新尾节点
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    //尾节点为空
    enq(node);
    return node;
}
```

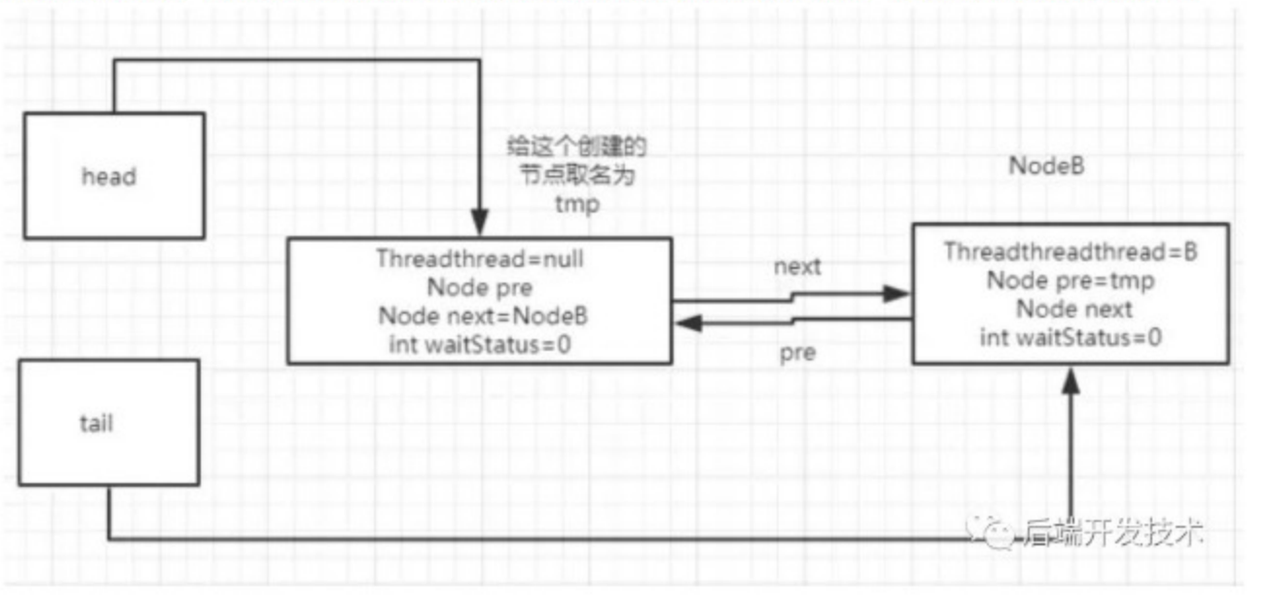
首先是创建一个EXCLUSIVE独占模式的node节点，并且将当前线程保存在node中，如果尾节点不为空，追加当前节点为尾节点并返回当前节点。如果尾节点tail是null,执行enq(node)，建立新的链表。

```java
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        //第一次进入循环 尾节点为空 建立新节点为头节点 尾等于头节点
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            //第二次设置当前节点为尾节点
             node.prev = t;
             if (compareAndSetTail(t, node)) {
                 t.next = node;
                 return t;
             }
         }
    }
}
```

具体看代码注释，第一次循环，通过CAS将一个刚创建出来的Node节点设置为头节点，并且tail尾部节点也指向这个节点，结构如下图



第二次循环执行死循环里面的代码，此时tail不等于null,执行else代码块，将传进来的Node结点（thread=B的节点）的pre前节点指向tail节点，将传进来的node节点设置为tail节点，并且将头节点的next节点设置为当前node节点，此时结构图变成如下。



addWaiter(Node.EXCLUSIVE)方法执行完成，并返回了当前节点node，继续执行acquireQueued(addWaiter(Node.EXCLUSIVE), arg)

```java
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            //说明前一节点为头节点 并且持有锁，加锁成功
            if (p == head && tryAcquire(arg)) {
                //设置当前节点为头节点
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            //前一节点非头节点 需要继续等待
            //或者 是头节点 ，但是已有其他线程持有锁
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

继续分析acquireQueued方法的实现，同样又是个死循环，首先执行node.predecessor()方法返回传入的node节点的pre节点。如果前一节点是头节点，则继续执行tryAcquire方法，如果其他线程占据着锁，代码会执行shouldParkAfterFailedAcquire(p, node)。如果头节点持有锁，设置当前新节点为头节点。p.next = null标记GC。

```java
private static boolean shouldParkAfterFailedAcquire(Node pred, Node n
    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.
         */
         //表明线程已经准备好被阻塞并等待之后被唤醒
        return true;
    if (ws > 0) {
        /*
         * Predecessor was cancelled. Skip over predecessors and
         * indicate retry.
         */
         //若pred.waitStatus状态位大于0，说明这个节点已经取消了获取锁的操作
         //doWhile循环会递归删除掉这些放弃获取锁的节点
        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.
         */
         //若状态位不为Node.SIGNAL,且没有取消操作，则会尝试将状态位修改为No
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

shouldParkAfterFailedAcquire方法有三种执行条件：

1、若pred.waitStatus状态位大于0，说明这个节点已经取消了获取锁的操作，doWhile循环会递归删除掉这些放弃获取锁的节点。

2、若状态位不为Node.SIGNAL,且没有取消操作，则会尝试将状态位修改为Node.SIGNAL。

3、状态位是Node.SIGNAL，表明线程是否已经准备好被阻塞并等待唤醒。

正常状况会执行compareAndSetWaitStatus()方法，将head节点的waitStatus设置为-1 Node.SIGNAL。shouldParkAfterFailedAcquire返回的false，继续会循环执行到这个方法，而此时的waitStatus=-1，所有直接返回true，继续执行parkAndCheckInterrupt方法。

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

该方法阻塞当前线程并且下次被唤醒的时候返回是否被中断标识。等到被唤醒，会再次执行acquireQueued中的循环，可能直接获得锁成功，或者再次被阻塞。

到这里 lock主要逻辑结束。

unlock()方法

lock是用来加锁的方法，代码如下

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

```java
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
        //释放锁成功 并且头节点不为空，状态为<0
            unparkSuccessor(h);
         return true;
    }
    return false;
}
 protected final boolean tryRelease(int releases) {
    //计算state
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        //state=0, 表示没有线程持有锁
        free = true;
        setExclusiveOwnerThread(null);
    }
    //如果c! =0 ，表明 同一线程可重入 更新state
    setState(c);
    return free;
}
```

unlock方法很简单，尝试释放锁，最后执行AQS中的release方法，并且通过调用ReentrantLock类中的tryRelease方法尝试释放锁。

如果释放的线程跟独占线程不是同一个，抛异常；如果当前state值减1等于0的话表示锁释放完成，把当前独占线程设置为null并把state值设置为0，返回true。state可能不为0，因为这是重入锁，同一个线程可以lock多次，所以必须得释放多次才是可以完全释放锁。

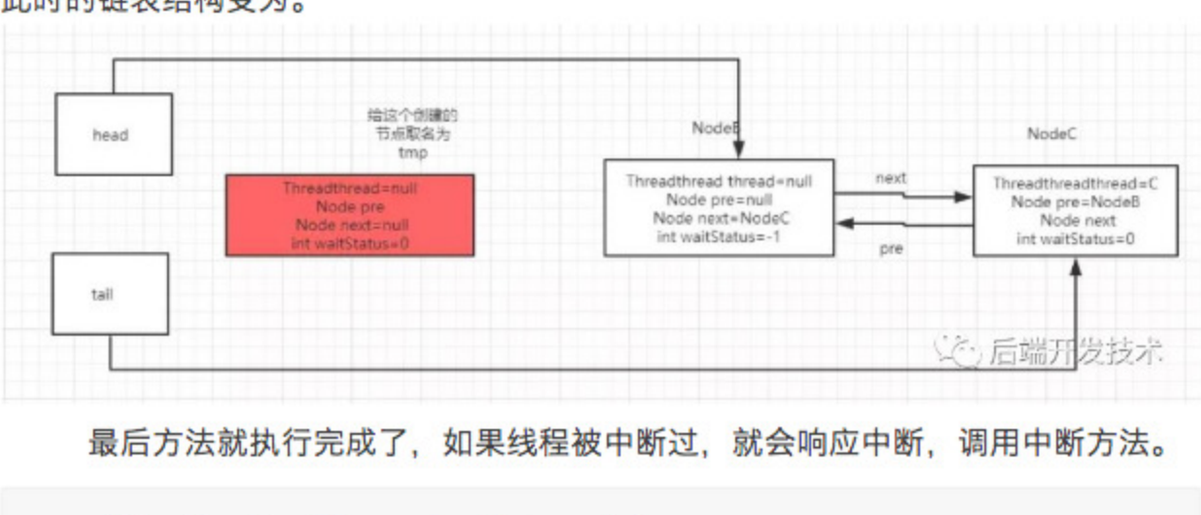
```java
private void unparkSuccessor(Node node) {
    /*
     * If status is negative (i.e., possibly needing signal) try
     * to clear in anticipation of signalling.  It is OK if this
     * fails or if status is changed by waiting thread.
     */
    int ws = node.waitStatus;
    if (ws < 0)
        //CAS 头节点状态waitStatus小于0 设为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.
     */
     //找到下一个节点
    Node s = node.next;
    //从尾节点开始倒着找 找到非空waitStatus<0的节点 设置头节点下一个节点为它
    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;
    }
    //释放下一个节点的锁
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

释放锁成功，执行unparkSuccessor方法，如果waitStatus <0,会继续执行compareAndSetWaitStatus方法将tmp的waitStatus改为0。

然后找到头节点的下一个节点，继续执行LockSupport.unpark(s.thread)，唤醒下一节点之前被阻塞的线程。

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

线程唤醒后将继续从上次挂起的地方继续执行，也就是执行Thread.interrupted()方法，如果线程被中断过，将interruped标识设为true，继续for循环逻辑。而此时如果获取锁成功，执行setHead方法，并且返回interrupted标识。此时的链表结构变为。



最后方法就执行完成了，如果线程被中断过，就会响应中断，调用中断方法。

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

公平锁和非公平锁

也许大家会觉得都是需要进入队列，为什么不公平。实际上进入队列中挂起的线程确实是公平的，最开始进入直接调用tryAcquire方法，如果获取到锁了就不会进入链表中，也不会被挂起。而公平和非公平锁的tryAcquire就一个地方不同，公平锁多了hasQueuedPredecessors方法，公平锁会判断链表中是否有其他线程在等待，如果有，就会把自己也加入到链表末尾，而非公平锁没有这个判断，直接是尝试获取锁，而正当锁被释放，有一个新的线程调用了lock方法这就会与链表中被唤醒的线程形成竞争关系，所以就成了非公平。代码如下。

```java
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //不同之处在这里！！！
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
     //...... 一样省略
    return false;
}
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
   //...... 一样省略
    return false;
}
```

有什么疑问或者发现错误，欢迎交流。
写文章不易感谢支持！