第五章 ReentrantLock源码解析1--获得非公平锁与公平锁lock()

最常用的方式：

```
        int a = 12;
        //注意：通常情况下，这个会设置成一个类变量，比如说Segement中的段锁与copyOnWriteArrayList中的全局锁
        final ReentrantLock lock = new ReentrantLock();

        lock.lock();//获取锁
        try {
            a++;//业务逻辑
        } catch (Exception e) {
        }finally{
            lock.unlock();//释放锁
        }
```

1、对于ReentrantLock需要掌握以下几点

- ReentrantLock的创建（公平锁/非公平锁）
- 上锁：lock()
- 解锁：unlock()

首先说一下类结构：

- ReentrantLock-->Lock
- NonfairSync/FairSync-->Sync-->AbstractQueuedSynchronizer-->AbstractOwnableSynchronizer
- NonfairSync/FairSync-->Sync是ReentrantLock的三个内部类
- Node是AbstractQueuedSynchronizer的内部类

注意：上边这四条线，对应关系："子类"-->"父类"

2、ReentrantLock的创建

- 支持公平锁（先进来的线程先执行）
- 支持非公平锁（后进来的线程也可能先执行）

非公平锁与非公平锁的创建

- 非公平锁：ReentrantLock()或ReentrantLock(false)

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

- 公平锁：ReentrantLock(true)

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

默认情况下使用非公平锁。

源代码如下：

ReentrantLock：

```
    /** 同步器：内部类Sync的一个引用 */
    private final Sync sync;

    /**
     * 创建一个非公平锁
     */
    public ReentrantLock() {
        sync = new NonfairSync();
    }

    /**
     * 创建一个锁
     * @param fair true-->公平锁  false-->非公平锁
     */
    public ReentrantLock(boolean fair) {
        sync = (fair)? new FairSync() : new NonfairSync();
    }
```

上述源代码中出现了三个内部类Sync/NonfairSync/FairSync，这里只列出类的定义，至于这三个类中的具体的方法会在后续的第一次引用的时候介绍。

Sync/NonfairSync/FairSync类定义：

```
    /**
     * 该锁同步控制的一个基类.下边有两个子类：非公平机制和公平机制.使用了AbstractQueuedSynchronizer类的
     */
    static abstract class Sync extends AbstractQueuedSynchronizer

    /**
     * 非公平锁同步器
     */
    final static class NonfairSync extends Sync

    /**
     * 公平锁同步器
     */
    final static class FairSync extends Sync
```

3、非公平锁的lock()

具体使用方法：

```
        lock.lock();
```

下面先介绍一下这个总体步骤的简化版，然后会给出详细的源代码，并在源代码的lock()方法部分给出详细版的步骤。

简化版的步骤：（非公平锁的核心）

基于CAS尝试将state（锁数量）从0设置为1

A、如果设置成功，设置当前线程为独占锁的线程；

B、如果设置失败，还会再获取一次锁数量，

B1、如果锁数量为0，再基于CAS尝试将state（锁数量）从0设置为1一次，如果设置成功，设置当前线程为独占锁的线程；

B2、如果锁数量不为0或者上边的尝试又失败了，查看当前线程是不是已经是独占锁的线程了，如果是，则将当前的锁数量+1；如果不是，则将该线程封装在一个Node内，并加入到等待队列中去。等待被其前一个线程节点唤醒。

源代码：（再介绍源代码之前，心里有一个获取锁的步骤的总的一个印象，就是上边这个"简化版的步骤"）

3.1、ReentrantLock：lock()

```
    /**
     *获取一个锁
     *三种情况：
     *1、如果当下这个锁没有被任何线程（包括当前线程）持有，则立即获取锁，锁数量==1，之后再执行相应的业务逻辑
     *2、如果当前线程正在持有这个锁，那么锁数量+1，之后再执行相应的业务逻辑
     *3、如果当下锁被另一个线程所持有，则当前线程处于休眠状态，直到获得锁之后，当前线程被唤醒，锁数量==1，再执行相应的业务逻辑
     */
    public void lock() {
        sync.lock();//调用NonfairSync（非公平锁）或FairSync（公平锁）的lock()方法
    }
```

3.2、NonfairSync：lock()

```
        /**
         * 1）首先基于CAS将state（锁数量）从0设置为1，如果设置成功，设置当前线程为独占锁的线程；-->请求成功-->第一次插队
         * 2）如果设置失败(即当前的锁数量可能已经为1了，即在尝试的过程中，已经被其他线程先一步占有了锁)，这个时候当前线程执行acquire(1)方法
         * 2.1）acquire(1)方法首先调用下边的tryAcquire(1)方法，在该方法中，首先获取锁数量状态，
         * 2.1.1）如果为0(证明该独占锁已被释放，当下没有线程在使用)，这个时候我们继续使用CAS将state（锁数量）从0设置为1，如果设置成功，当前线程独占锁；-->请求成功-->第二次插队；当然，如果设置不成功，直接返回false
         * 2.2.2）如果不为0，就去判断当前的线程是不是就是当下独占锁的线程，如果是，就将当前的锁数量状态值+1（这也就是可重入锁的名称的来源）-->请求成功
         *
         * 下边的流程一句话：请求失败后，将当前线程链入队尾并挂起，之后等待被唤醒。
         *
```

```
        * 2.2.3）如果最后在tryAcquire(1)方法中上述的执行都没成功，即请求没有成功，则返回false，继续执行acquireQueued(addWaiter(Node.EXCLUSIVE), arg)方法
        * 2.2）在上述方法中，首先会使用addWaiter(Node.EXCLUSIVE)将当前线程封装进Node节点node，然后将该节点加入等待队列（先快速入队，如果快速入队不成功，其使用正常入队方法无限循环一直到Node节点入队为止）
        * 2.2.1）快速入队：如果同步等待队列存在尾节点，将使用CAS尝试将尾节点设置为node，并将之前的尾节点插入到node之前
        * 2.2.2）正常入队：如果同步等待队列不存在尾节点或者上述CAS尝试不成功的话，就执行正常入队（该方法是一个无限循环的过程，即直到入队为止）-->第一次阻塞
        * 2.2.2.1）如果尾节点为空（初始化同步等待队列），创建一个dummy节点，并将该节点通过CAS尝试设置到头节点上去，设置成功的话，将尾节点也指向该dummy节点（即头节点和尾节点都指向该dummy节点）
        * 2.2.2.1）如果尾节点不为空，执行与快速入队相同的逻辑，即使用CAS尝试将尾节点设置为node，并将之前的尾节点插入到node之前
        * 最后，如果顺利入队的话，就返回入队的节点node，如果不顺利的话，无限循环去执行2.2)下边的流程，直到入队为止
        * 2.3）node节点入队之后，就去执行acquireQueued(final Node node, int arg)（这又是一个无限循环的过程，这里需要注意的是，无限循环等于阻塞，多个线程可以同时无限循环--每个线程都可以执行自己的循环，这样才能使在后边排队的节点不断前进）
        * 2.3.1）获取node的前驱节点p，如果p是头节点，就继续使用tryAcquire(1)方法去尝试请求成功，-->第三次插队（当然，这次插队不一定不会使其获得执行权，请看下边一条），
        * 2.3.1.1）如果第一次请求就成功，不用中断自己的线程，如果是之后的循环中将线程挂起之后又请求成功了，使用selfInterrupt()中断自己
        * （注意p==head&&tryAcquire(1)成功是唯一跳出循环的方法，在这之前会一直阻塞在这里，直到其他线程在执行的过程中，不断的将p的前边的节点减少，直到p成为了head且node请求成功了--即node被唤醒了，才退出循环）
        * 2.3.1.2）如果p不是头节点，或者tryAcquire(1)请求不成功，就去执行shouldParkAfterFailedAcquire(Node pred, Node node)来检测当前节点是不是可以安全的被挂起，
        * 2.3.1.2.1）如果node的前驱节点pred的等待状态是SIGNAL（即可以唤醒下一个节点的线程），则node节点的线程可以安全挂起，执行2.3.1.3）
        * 2.3.1.2.2）如果node的前驱节点pred的等待状态是CANCELLED，则pred的线程被取消了，我们会将pred之前的连续几个被取消的前驱节点从队列中剔除，返回false（即不能挂起），之后继续执行2.3）中上述的代码
        * 2.3.1.2.3）如果node的前驱节点pred的等待状态是除了上述两种的其他状态，则使用CAS尝试将前驱节点的等待状态设为SIGNAL，并返回false（因为CAS可能会失败，这里不管失败与否，都返回false，下一次执行该方法的之后，pred的等待状态就是SIGNAL了），之后继续执行2.3）中上述的代码
        * 2.3.1.3）如果可以安全挂起，就执行parkAndCheckInterrupt()挂起当前线程，之后，继续执行2.3）中之前的代码
        * 最后，直到该节点的前驱节点p之前的所有节点都执行完毕为止，我们的p成为了头节点，并且tryAcquire(1)请求成功，跳出循环，去执行。
        * （在p变为头节点之前的整个过程中，我们发现这个过程是不会被中断的）
        * 2.3.2）当然在2.3.1）中产生了异常，我们就会执行cancelAcquire(Node node)取消node的获取锁的意图。
        */
       final void lock() {
           if (compareAndSetState(0, 1))//如果CAS尝试成功
               setExclusiveOwnerThread(Thread.currentThread());//设置当前线程为独占锁的线程
           else
               acquire(1);
       }
```

注意：在这个方法中，我列出了一个线程获取锁的详细的过程，自己看注释。

下面列出NonfairSync：lock()中调用的几个方法与相关属性。

3.2.1、AbstractQueuedSynchronizer：锁数量state属性+相关方法：

```
    /**
     * 锁数量
     */
    private volatile int state;

    /**
     * 获取锁数量
     */
    protected final int getState() {
        return state;
    }

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

注意：state是volatile型的

3.2.2、AbstractOwnableSynchronizer：属性+setExclusiveOwnerThread(Thread t)

```
    /**
     * 当前拥有独占锁的线程
     */
```

```
    private transient Thread exclusiveOwnerThread;

    /**
     * 设置独占锁的线程为线程t
     */
    protected final void setExclusiveOwnerThread(Thread t) {
        exclusiveOwnerThread = t;
    }
```

3.2.3、AbstractQueuedSynchronizer：属性+acquire(int arg)

```
    /**
     * 获取锁的方法
     * @param arg
     */
    public final void acquire(int arg) {
        if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();//中断自己
    }
```

在介绍上边这个方法之前，先要说一下AbstractQueuedSynchronizer的一个内部类Node的整体构造，源代码如下：

```
    /**
     * 同步等待队列（双向链表）中的节点
     */
    static final class Node {
        /** 线程被取消了 */
        static final int CANCELLED = 1;
        /**
         * 如果前驱节点的等待状态是SIGNAL，表示当前节点将来可以被唤醒，那么当前节点就可以安全的挂起了
         * 否则，当前节点不能挂起
         */
        static final int SIGNAL = -1;
        /**线程正在等待条件*/
        static final int CONDITION = -2;
        /**
         * waitStatus value to indicate the next acquireShared should
         * unconditionally propagate
         */
        static final int PROPAGATE = -3;
        /** Marker to indicate a node is waiting in shared mode */
        static final Node SHARED = new Node();
        /** 一个标记：用于表明该节点正在独占锁模式下进行等待 */
        static final Node EXCLUSIVE = null;
        //值就是前四个int（CANCELLED/SIGNAL/CONDITION/PROPAGATE），再加一个0

        volatile int waitStatus;
        /**前驱节点*/
        volatile Node prev;

        /**后继节点*/
        volatile Node next;

        /**节点中的线程*/
        volatile Thread thread;

        /**
         * Link to next node waiting on condition, or the special value SHARED.
         * Because condition queues are accessed only when holding in exclusive
         * mode, we just need a simple linked queue to hold nodes while they are
         * waiting on conditions. They are then transferred to the queue to
         * re-acquire. And because conditions can only be exclusive, we save a
         * field by using special value to indicate shared mode.
         */
        Node nextWaiter;

        /**
```

```
         * Returns true if node is waiting in shared mode
         */
        final boolean isShared() {
            return nextWaiter == SHARED;
        }

        /**
         * 返回该节点前一个节点
         */
        final Node predecessor() throws NullPointerException {
            Node p = prev;
            if (p == null)
                throw new NullPointerException();
            else
                return p;
        }

        Node() { // Used to establish initial head or SHARED marker
        }

        Node(Thread thread, Node mode) { // 用于addWaiter中
            this.nextWaiter = mode;
            this.thread = thread;
        }

        Node(Thread thread, int waitStatus) { // Used by Condition
            this.waitStatus = waitStatus;
            this.thread = thread;
        }
    }
```

注意：这里我给出了Node类的完整版，其中部分属性与方法是在共享锁的模式下使用的，而我们这里的ReentrantLock是一个独占锁，只需关注其中的与独占锁相关的部分就好（具体有注释）

3.3、AbstractQueuedSynchronizer：acquire(int arg)方法中使用到的两个方法

3.3.1、NonfairSync：tryAcquire(int acquires)

```
        /**
         * 试着请求成功
         */
        protected final boolean tryAcquire(int acquires) {
            return nonfairTryAcquire(acquires);
        }
```

Syn：

```
        /**
         * 非公平锁中被tryAcquire调用
         */
        final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();//获取当前线程
            int c = getState();//获取锁数量
            if (c == 0) {//如果锁数量为0，证明该独占锁已被释放，当下没有线程在使用
                if (compareAndSetState(0, acquires)) {//继续通过CAS将state由0变为1，注意这里传入的acquires为1
                    setExclusiveOwnerThread(current);//将当前线程设置为独占锁的线程
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {//查看当前线程是不是就是独占锁的线程
                int nextc = c + acquires;//如果是，锁状态的数量为当前的锁数量+1
                if (nextc < 0) // overflow
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);//设置当前的锁数量
                return true;
            }
            return false;
        }
```

注意：这个方法就完成了"简化版的步骤"中的"A/B/B1"三步，如果上述的请求不能成功，就要执行下边的代码了，

下边的代码，用一句话介绍：**请求失败后，将当前线程链入队尾并挂起，之后等待被唤醒。**在你看下边的代码的时候心里默记着这句话。

3.3.2、AbstractQueuedSynchronizer：addWaiter(Node mode)

```
    /**
     * 将Node节点加入等待队列
     * 1）快速入队，入队成功的话，返回node
     * 2）入队失败的话，使用正常入队
     * 注意：快速入队与正常入队相比，可以发现，正常入队仅仅比快速入队多而一个判断队列是否为空且为空之后的过程
     * @return 返回当前要插入的这个节点，注意不是前一个节点
     */
    private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);//创建节点
        /*
         * 快速入队
         */
        Node pred = tail;//将尾节点赋给pred
        if (pred != null) {//尾节点不为空
            node.prev = pred;//将尾节点作为创造出来的节点的前一个节点，即将node链接到为节点后
            /**
             * 基于CAS将node设置为尾节点，如果设置失败，说明在当前线程获取尾节点到现在这段过程中已经有其他线程将尾节点给替换过了
             * 注意：假设有链表node1-->node2-->pred（当然是双链表，这里画成双链表才合适），
             * 通过CAS将pred替换成了node节点，即当下的链表为node1-->node2-->node,
             * 然后根据上边的"node.prev = pred"与下边的"pred.next = node"将pred插入到双链表中去，组成最终的链表如下：
             * node1-->node2-->pred-->node
             * 这样的话，实际上我们发现没有指定node2.next=pred与pred.prev=node2，这是为什么呢？
             * 因为在之前这两句就早就执行好了，即node2.next和pred.prev这连个属性之前就设置好了
             */
            if (compareAndSetTail(pred, node)) {
                pred.next = node;//将node放在尾节点上
                return node;
            }
        }
        enq(node);//正常入队
        return node;
    }
```

AbstractQueuedSynchronizer：enq(final Node node)

```
    /**
     * 正常入队
```

```
     * @param node
     * @return 之前的尾节点
     */
    private Node enq(final Node node) {
        for (;;) {//无限循环，一定要阻塞到入队成功为止
            Node t = tail;//获取尾节点
            if (t == null) { //如果尾节点为null，说明当前等待队列为空
                /*Node h = new Node(); // Dummy header
                h.next = node;
                node.prev = h;
                if (compareAndSetHead(h)) {//根据代码实际上是：compareAndSetHead(null,h)
                    tail = node;
                    return h;
                }*/
                /*
                 * 注意：上边注释掉的这一段代码是jdk1.6.45中的，在后来的版本中，这一段改成了如下这段
                 * 基于CAS将新节点（一个dummy节点）设置到头上head去，如果发现内存中的当前值不是null，则说明，在这个过程中，已经有其他线程设置过了。
                 * 当成功的将这个dummy节点设置到head节点上去时，我们又将这个head节点设置给了tail节点，即head与tail都是当前这个dummy节点，
                 * 之后有新节点入队的话，就插入到该dummy之后
                 */
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else {//这一块儿的逻辑与快速入队完全相同
                node.prev = t;
                if (compareAndSetTail(t, node)) {//尝试将node节点设为尾节点
                    t.next = node;//将node节点设为尾节点
                    return t;
                }
            }
        }
    }
```

注意：这里就是一个完整的入队方法，具体逻辑看注释和ReentrantLock：lock()的注释部分的相关部分。

3.3.3、AbstractQueuedSynchronizer：acquireQueued(final Node node, int arg)

```
    final boolean acquireQueued(final Node node, int arg) {
        try {
            boolean interrupted = false;
            /*
             * 无限循环（一直阻塞），直到node的前驱节点p之前的所有节点都执行完毕，p成为了head且node请求成功了
             */
            for (;;) {
                final Node p = node.predecessor();//获取插入节点的前一个节点p
                /*
                 * 注意：
                 * 1、这个是跳出循环的唯一条件，除非抛异常
                 * 2、如果p == head && tryAcquire(arg)第一次循环就成功了，interrupted为false，不需要中断自己
                 *          如果p == head && tryAcquire(arg)第一次以后的循环中如果执行了挂起操作后才成功了，interrupted为true，就要中断自己了
                 */
                if (p == head && tryAcquire(arg)) {
                    setHead(node);//当前节点设置为头节点
                    p.next = null;
                    return interrupted;//跳出循环
                }
                if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())
                    interrupted = true;//被中断了
            }
        } catch (RuntimeException ex) {
            cancelAcquire(node);
            throw ex;
        }
    }
```

AbstractQueuedSynchronizer：shouldParkAfterFailedAcquire(Node pred, Node node)

```
    /**
     * 检测当前节点是否可以被安全的挂起（阻塞）
     * @param pred    当前节点的前驱节点
     * @param node    当前节点
     */
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;//获取前驱节点（即当前线程的前一个节点）的等待状态
        if (ws == Node.SIGNAL)//如果前驱节点的等待状态是SIGNAL，表示当前节点将来可以被唤醒，那么当前节点就可以安全的挂起了
            return true;
        /*
         * 1）当ws>0(即CANCELLED==1），前驱节点的线程被取消了，我们会将该节点之前的连续几个被取消的前驱节点从队列中剔除，返回false（即不能挂起）
         * 2）如果ws<=0&&!=SIGNAL,将当前节点的前驱节点的等待状态设为SIGNAL
         */
        if (ws > 0) {
            do {
                /*
                 * node.prev = pred = pred.prev;
                 * 上边这句代码相当于下边这两句
                 * pred = pred.prev;
                 * node.prev = pred;
                 */
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            /*
             * 尝试将当前节点的前驱节点的等待状态设为SIGNAL
             * 1/这为什么用CAS，现在已经入队成功了，前驱节点就是pred，除了node外应该没有别的线程在操作这个节点了，那为什么还要用CAS？而不直接赋值呢？
             * （解释：因为pred可以自己将自己的状态改为cancel，也就是pred的状态可能同时会有两条线程（pred和node）去操作）
             * 2/既然前驱节点已经设为SIGNAL了，为什么最后还要返回false
             * （因为CAS可能会失败，这里不管失败与否，都返回false，下一次执行该方法的之后，pred的等待状态就是SIGNAL了）
             */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

AbstractQueuedSynchronizer：

```
    private final boolean parkAndCheckInterrupt() {
        LockSupport.park(this);//挂起当前的线程
        return Thread.interrupted();//如果当前线程已经被中断了，返回true
    }
```

以上就是一个线程获取非公平锁的整个过程（lock()）。

4、公平锁的lock()

具体用法与非公平锁一样

如果掌握了非公平锁的流程，那么掌握公平锁的流程会非常简单，只有两点不同（最后会讲）。

简化版的步骤：（公平锁的核心）

获取一次锁数量，

B1、如果锁数量为0，如果当前线程是等待队列中的头节点，基于CAS尝试将state（锁数量）从0设置为1一次，如果设置成功，设置当前线程为独占锁的线程；

B2、如果锁数量不为0或者当前线程不是等待队列中的头节点或者上边的尝试又失败了，查看当前线程是不是已经是独占锁的线程了，如果是，则将当前的锁数量+1；如果不是，则将该线程封装在一个Node内，并加入到等待队列中去。等待被其前一个线程节点唤醒。

源代码：

4.1、ReentrantLock：lock()

```
    /**
     *获取一个锁
     *三种情况：
     *1、如果当下这个锁没有被任何线程（包括当前线程）持有，则立即获取锁，锁数量==1，之后被唤醒再执行相应的业务逻辑
     *2、如果当前线程正在持有这个锁，那么锁数量+1，之后被唤醒再执行相应的业务逻辑
     *3、如果当下锁被另一个线程所持有，则当前线程处于休眠状态，直到获得锁之后，当前线程被唤醒，锁数量==1，再执行相应的业务逻辑
     */
    public void lock() {
        sync.lock();//调用FairSync的lock()方法
    }
```

4.2、FairSync：lock()

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

4.3、AbstractQueuedSynchronizer：acquire(int arg)就是非公平锁使用的那个方法

4.3.1、FairSync：tryAcquire(int acquires)

```
        /**
         * 获取公平锁的方法
         * 1）获取锁数量c
         * 1.1)如果c==0，如果当前线程是等待队列中的头节点，使用CAS将state（锁数量）从0设置为1，如果设置成功，当前线程独占锁-->请求成功
         * 1.2)如果c!=0，判断当前的线程是不是就是当下独占锁的线程，如果是，就将当前的锁数量状态值+1（这也就是可重入锁的名称的来源）-->请求成功
         * 最后，请求失败后，将当前线程链入队尾并挂起，之后等待被唤醒。
         */
        protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                if (isFirst(current) && compareAndSetState(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;
        }
```

最后，如果请求失败后，将当前线程链入队尾并挂起，之后等待被唤醒，下边的代码与非公平锁一样。

总结：公平锁与非公平锁对比

- FairSync：lock()少了插队部分（即少了CAS尝试将state从0设为1，进而获得锁的过程）
- FairSync：tryAcquire(int acquires)多了需要判断当前线程是否在等待队列首部的逻辑（实际上就是少了再次插队的过程，但是CAS获取还是有的）。

最后说一句，

- ReentrantLock是基于AbstractQueuedSynchronizer实现的，AbstractQueuedSynchronizer可以实现独占锁也可以实现共享锁，ReentrantLock只是使用了其中的独占锁模式
- 这一块儿代码比较多，逻辑比较复杂，最好在阅读的过程中，可以拿一根笔画画入队等与数据结构相关的图
- 一定要记住"简化版的步骤"，这是整个非公平锁与公平锁的核心