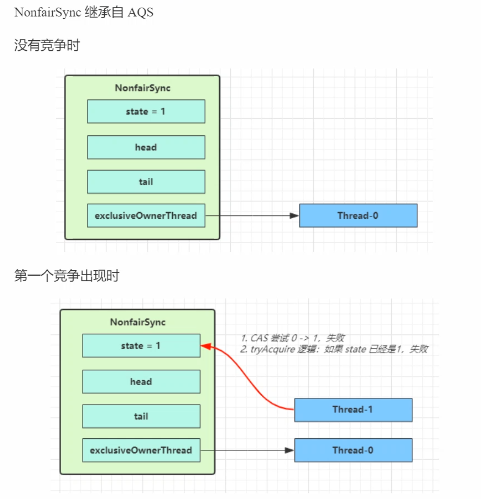
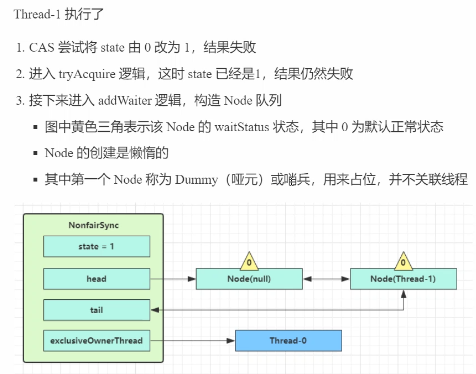


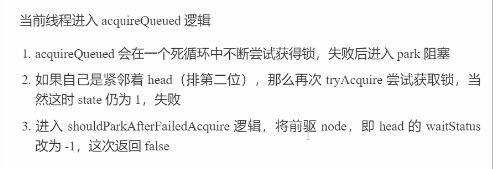
public ReentrantLock() {  
 sync = new NonfairSync();  
}

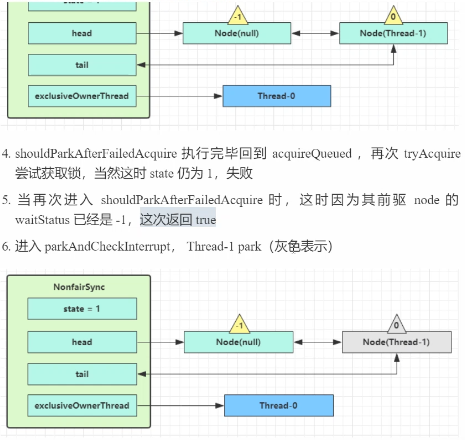
public void lock() {  
 sync.lock();  
}

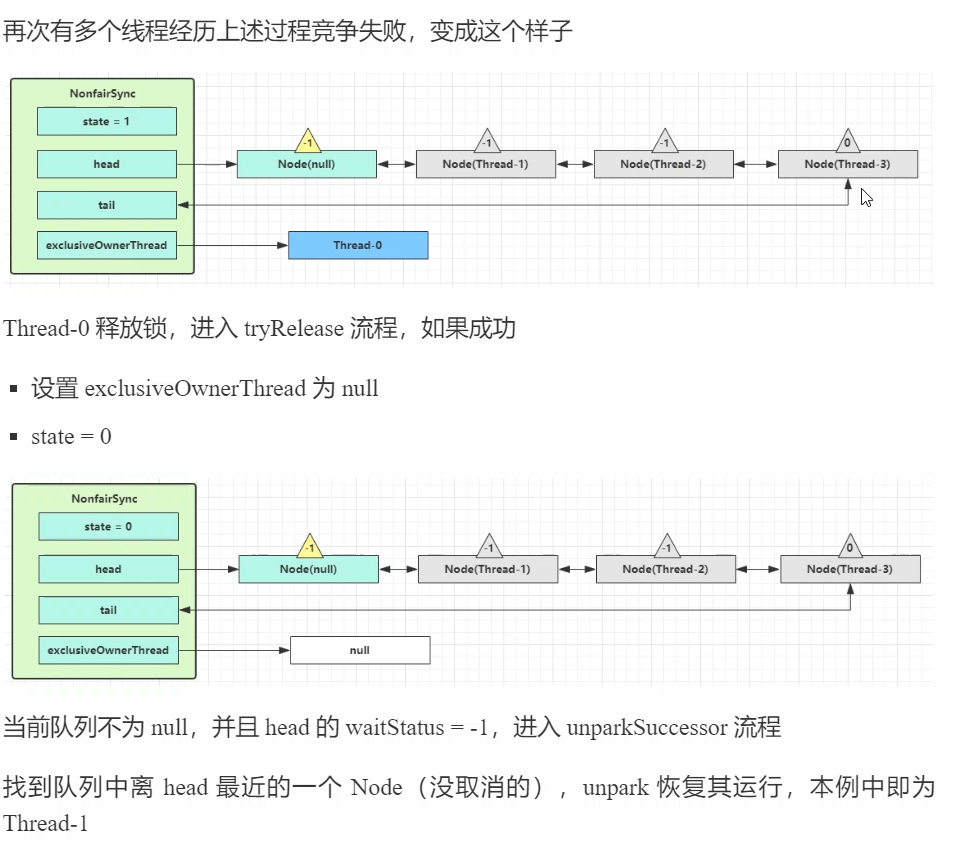
final void lock() {  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 acquire(1);  
}

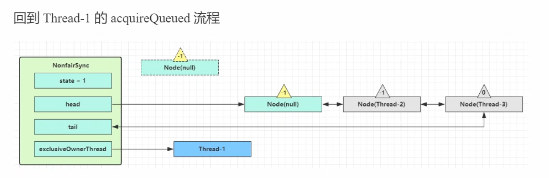


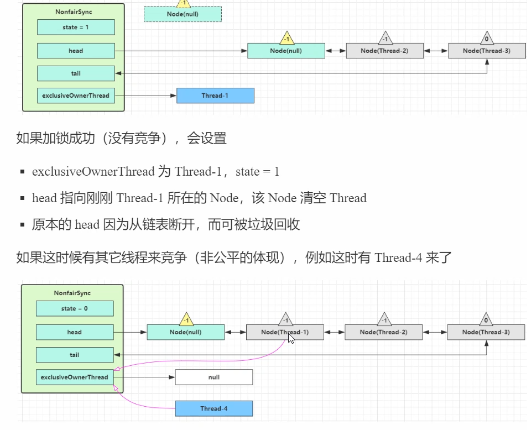


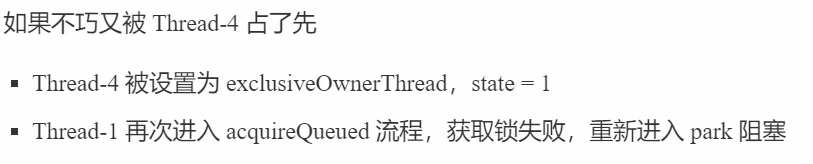










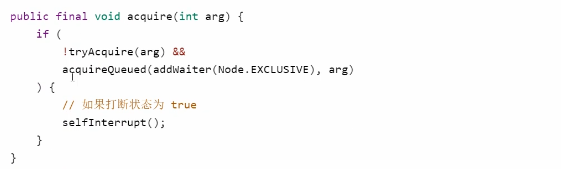




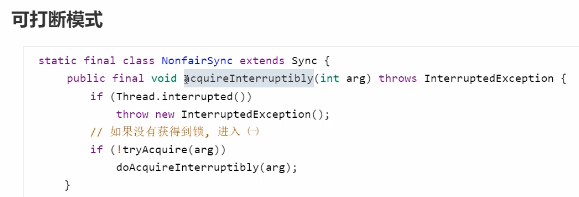






\





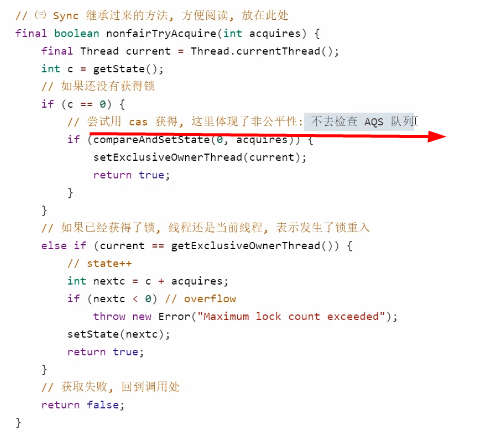


可打断和不可打断的区别：

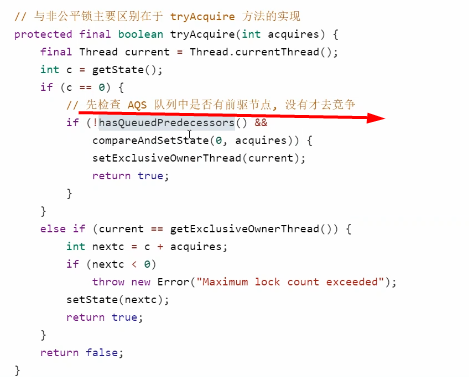
不可打断： 即使被打断，也只是将打断表示置为true，还是要去尝试获取锁，没有获取到锁还是会被挂起

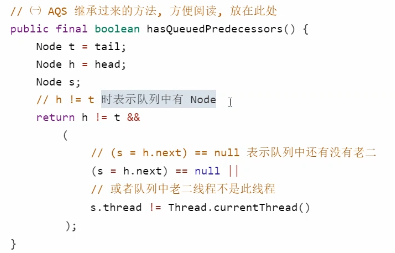
可打断： 被打断后会抛出异常

非公平锁原理



公平锁原理

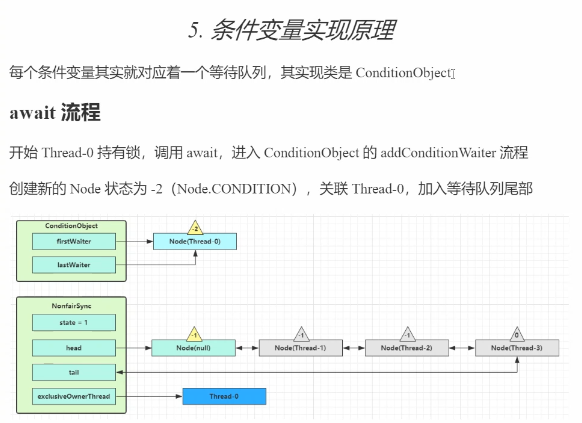




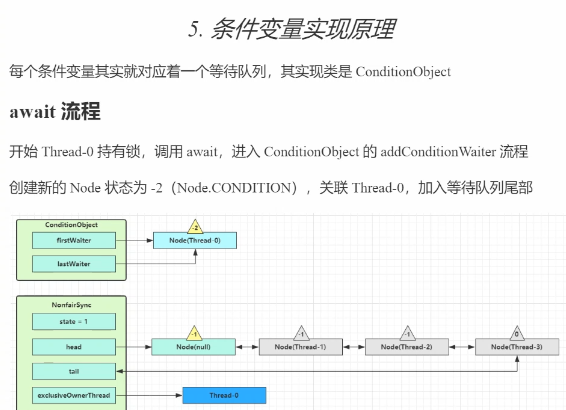
公平锁和非公平锁的区别：

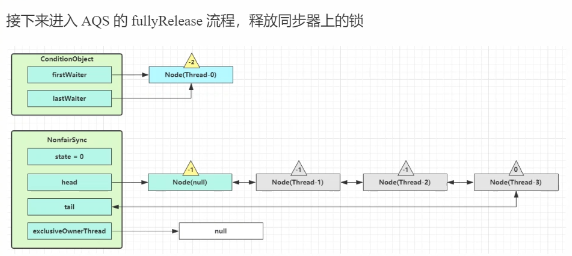
新加入的线程直接去尝试获取锁，不需要判断 阻塞队列是否有其它线程在等待

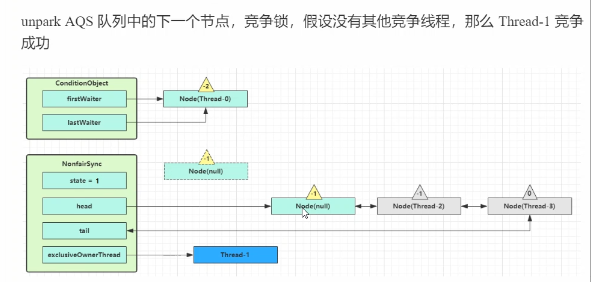
； 非公平锁：新加入的线程 需要先去阻塞队列中判断是否还有其它线程在等待，如果没有，则再尝试获取锁

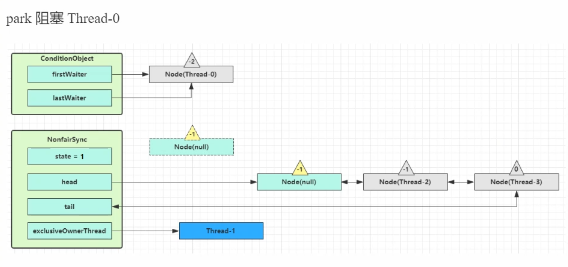


Await 和 signal的原理：









public final void await() throws InterruptedException {  
 if (Thread.*interrupted*())  
 throw new InterruptedException();

// 当前线程加入waitSet  
 Node node = addConditionWaiter();

// 清除当前线程的可重入锁计数，并释放锁  
 long savedState = fullyRelease(node);  
 int interruptMode = 0;  
 while (!isOnSyncQueue(node)) {

// 挂起当前线程  
 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);  
}

Signal原理：

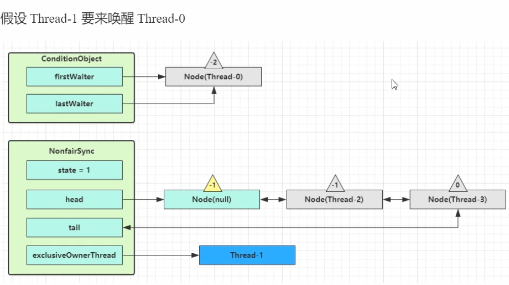
public final void signal() {

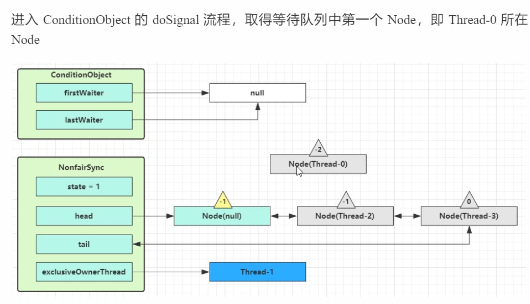
// 是不是锁的持有者  
 if (!isHeldExclusively())  
 throw new IllegalMonitorStateException();  
 // 获取waitSet中的第一个等待线程

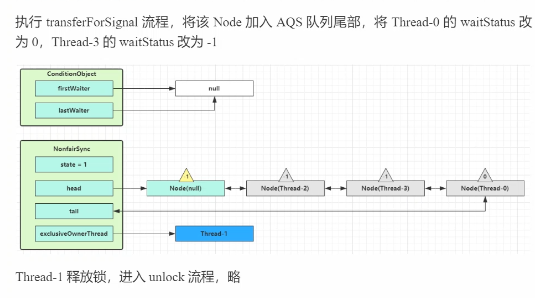
Node first = firstWaiter;  
 if (first != null)  
 doSignal(first);  
}

private void doSignal(Node first) {  
 do {  
 if ( (firstWaiter = first.nextWaiter) == null)  
 lastWaiter = null;  
 first.nextWaiter = null;

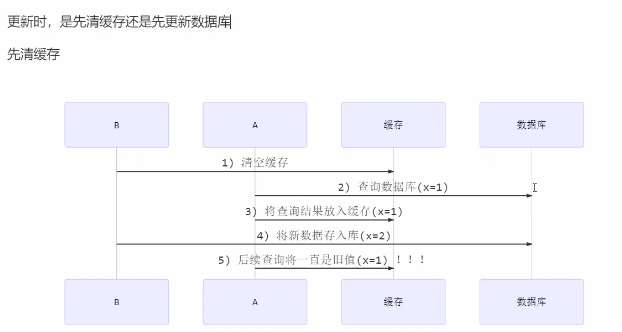
// 将waiset第一个转移到 阻塞队列  
 } while (!transferForSignal(first) &&  
 (first = firstWaiter) != null);  
}

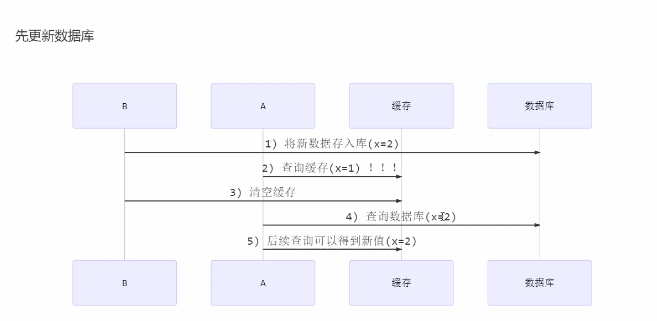


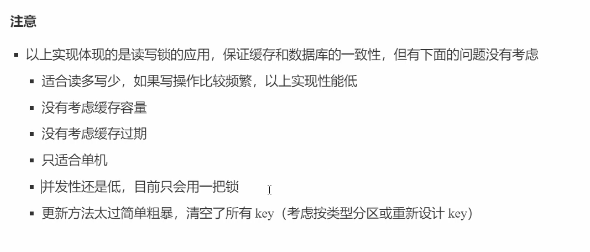




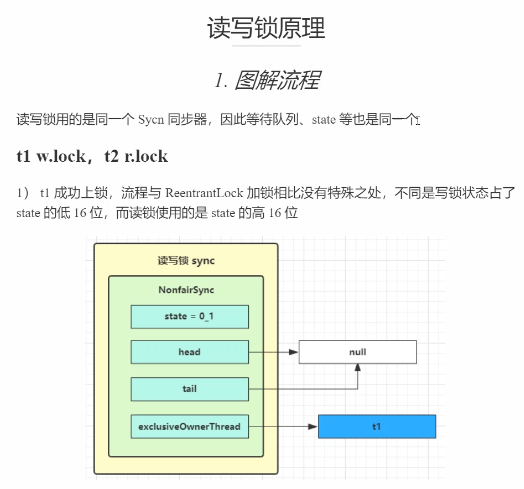
缓存更新策略







读写锁原理



// 写锁加锁过程

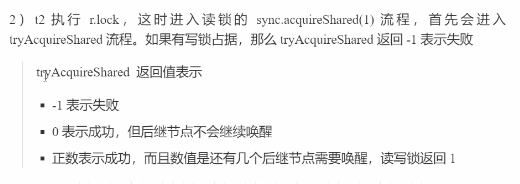
protected final boolean tryAcquire(int acquires) {  
 /\*  
 \* Walkthrough:  
 \* 1. If read count nonzero or write count nonzero  
 \* and owner is a different thread, fail.  
 \* 2. If count would saturate, fail. (This can only  
 \* happen if count is already nonzero.)  
 \* 3. Otherwise, this thread is eligible for lock if  
 \* it is either a reentrant acquire or  
 \* queue policy allows it. If so, update state  
 \* and set owner.  
 \*/  
 Thread current = Thread.*currentThread*();  
 int c = getState();  
 int w = *exclusiveCount*(c);  
 if (c != 0) {  
 // (Note: if c != 0 and w == 0 then shared count != 0)

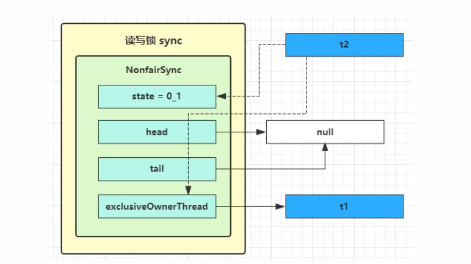
// 是否是读锁；如果不是读锁，是不是当前线程持有写锁  
 if (w == 0 || current != getExclusiveOwnerThread())  
 return false;

// 重入次数是否超过最大值  
 if (w + *exclusiveCount*(acquires) > *MAX\_COUNT*)  
 throw new Error("Maximum lock count exceeded");  
 // Reentrant acquire

// 重入成功  
 setState(c + acquires);  
 return true;  
 }

// 非公平锁返回false，公平锁检查 等待队列  
 if (writerShouldBlock() ||  
 !compareAndSetState(c, c + acquires))  
 return false;  
 setExclusiveOwnerThread(current);  
 return true;



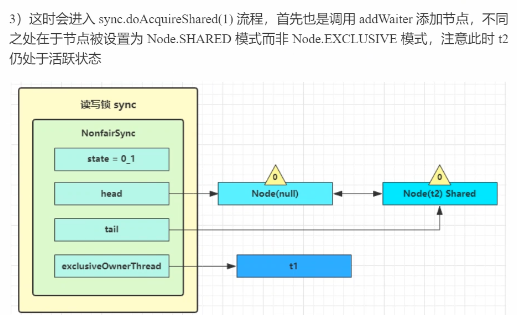


public final void acquireShared(int arg) {  
 if (tryAcquireShared(arg) < 0)  
 doAcquireShared(arg);  
}

// 读锁加锁过程

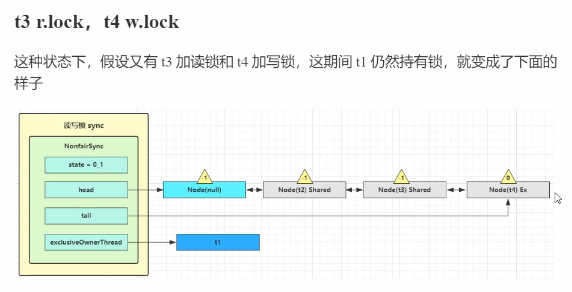
protected final int tryAcquireShared(int unused) {  
 /\*  
 \* Walkthrough:  
 \* 1. If write lock held by another thread, fail.  
 \* 2. Otherwise, this thread is eligible for  
 \* lock wrt state, so ask if it should block  
 \* because of queue policy. If not, try  
 \* to grant by CASing state and updating count.  
 \* Note that step does not check for reentrant  
 \* acquires, which is postponed to full version  
 \* to avoid having to check hold count in  
 \* the more typical non-reentrant case.  
 \* 3. If step 2 fails either because thread  
 \* apparently not eligible or CAS fails or count  
 \* saturated, chain to version with full retry loop.  
 \*/  
 Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (*exclusiveCount*(c) != 0 &&

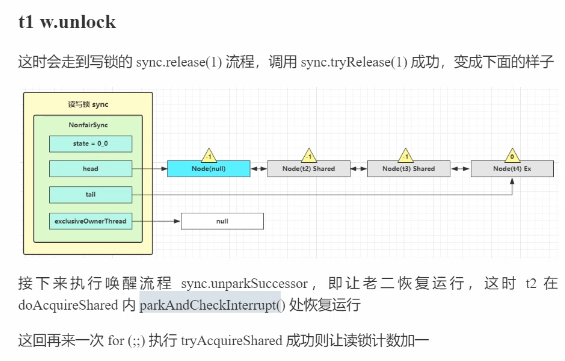
//   
 getExclusiveOwnerThread() != current)  
 return -1;  
 int r = *sharedCount*(c);  
 if (!readerShouldBlock() &&  
 r < *MAX\_COUNT* &&  
 compareAndSetState(c, c + *SHARED\_UNIT*)) {  
 if (r == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != *getThreadId*(current))  
 cachedHoldCounter = rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 rh.count++;  
 }  
 return 1;  
 }  
 return fullTryAcquireShared(current);  
}



*/\*\*  
 \* Acquires in shared uninterruptible mode.  
 \** ***@param*** *arg the acquire argument  
 \*/*private void doAcquireShared(int arg) {  
 final Node node = addWaiter(Node.*SHARED*);  
 boolean failed = true;  
 try {  
 boolean interrupted = false;  
 for (;;) {  
 final Node p = node.predecessor();  
 if (p == head) {  
 int r = tryAcquireShared(arg);  
 if (r >= 0) {  
 setHeadAndPropagate(node, r);  
 p.next = null; // help GC  
 if (interrupted)  
 *selfInterrupt*();  
 failed = false;  
 return;  
 }  
 }

// 当前线程进入阻塞之前，在这个for会尝试两次的获取读锁  
 if (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())  
 interrupted = true;  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

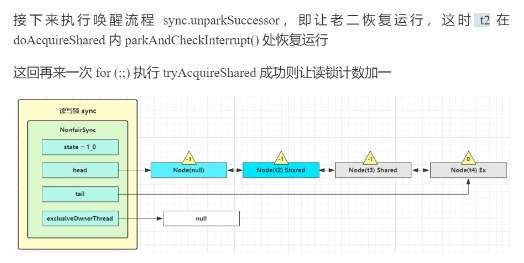




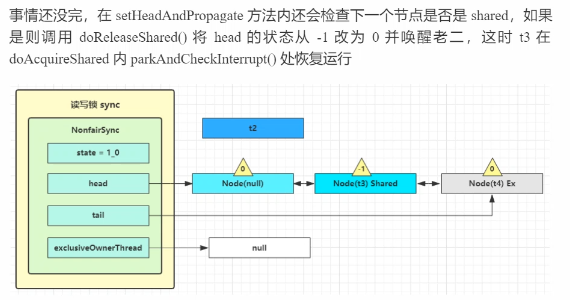
// 写锁释放

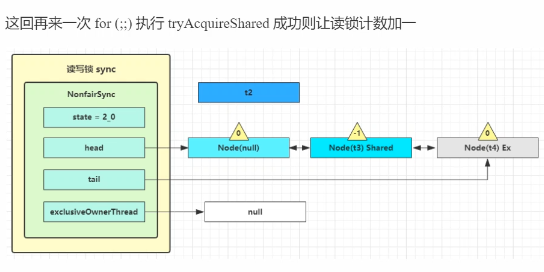
public final boolean release(int arg) {  
 if (tryRelease(arg)) {  
 Node h = head;  
 if (h != null && h.waitStatus != 0)  
 unparkSuccessor(h);  
 return true;  
 }  
 return false;  
}

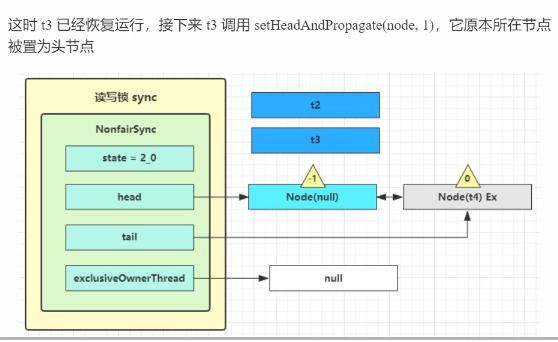
protected final boolean tryRelease(int releases) {  
 if (!isHeldExclusively())  
 throw new IllegalMonitorStateException();  
 int nextc = getState() - releases;  
 boolean free = *exclusiveCount*(nextc) == 0;  
 if (free)  
 setExclusiveOwnerThread(null);  
 setState(nextc);  
 return free;  
}

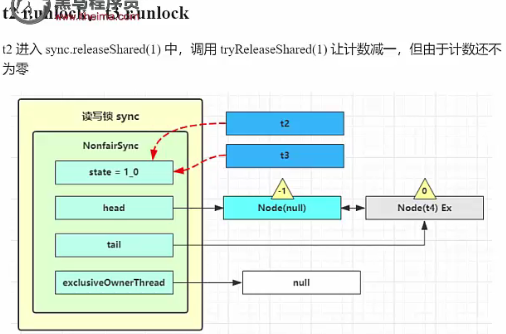


private void doReleaseShared() {  
 /\*  
 \* Ensure that a release propagates, even if there are other  
 \* in-progress acquires/releases. This proceeds in the usual  
 \* way of trying to unparkSuccessor of head if it needs  
 \* signal. But if it does not, status is set to PROPAGATE to  
 \* ensure that upon release, propagation continues.  
 \* Additionally, we must loop in case a new node is added  
 \* while we are doing this. Also, unlike other uses of  
 \* unparkSuccessor, we need to know if CAS to reset status  
 \* fails, if so rechecking.  
 \*/  
 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);  
 }  
 else if (ws == 0 &&  
 !*compareAndSetWaitStatus*(h, 0, Node.*PROPAGATE*))  
 continue; // loop on failed CAS  
 }  
 if (h == head) // loop if head changed  
 break;  
 }  
}









// 读锁释放

protected final boolean tryReleaseShared(int unused) {

for (;;) {  
 int c = getState();  
 int nextc = c - *SHARED\_UNIT*;  
 if (compareAndSetState(c, nextc))  
 // Releasing the read lock has no effect on readers,  
 // but it may allow waiting writers to proceed if  
 // both read and write locks are now free.  
 return nextc == 0;  
 }

}

