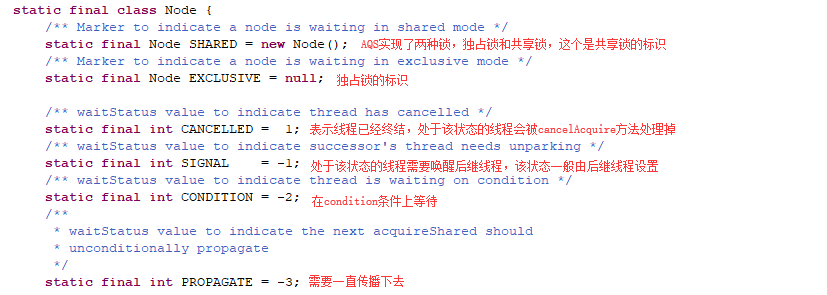
# AbstractQueuedSynchronizer(jdl1.7.0\_67)

AQS（AbstractQueuedSynchronizer）是一个抽象类，其中有很多的抽象方法和模板方法，抽象方法主要是用来设置同步状态的，模板方法大多数是用来维护同步队列和阻塞唤醒线程的，还有一部分模板方法是提供给锁的实现者直接调用的。

AQS作为实现锁和相关同步器（ReentrantLock，Semaphores...）的基础框架，使实现者只需关注同步状态，而无需关注同步队列和线程的阻塞与唤醒。AQS内部实现了一个CLH的同步队列，保存了所有请求锁的线程的相关信息。

先看一下AQS中的Node属性，封装了同步队列节点的相关信息。



同步队列节点的状态，以上状态的一种或者0，0表示无状态。



前驱节点。



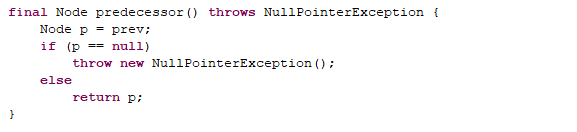
后继节点。



当前线程。

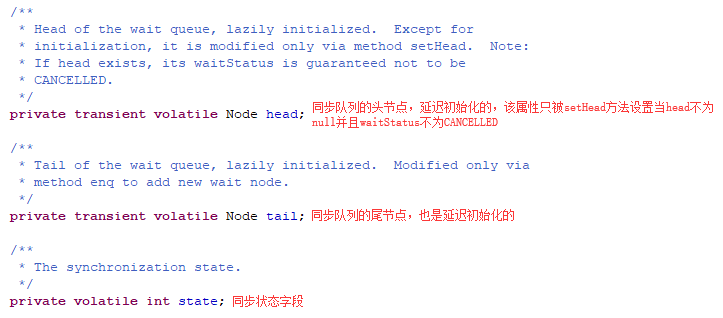


是否是共享锁。



获取前驱节点。

AQS中几个重要的属性。



# 同步状态管理

AQS通过这几个方法设置同步状态。

|  |  |  |  |
| --- | --- | --- | --- |
| 方法名 | 参数 | 返回值 | 说明 |
| getState |  | int | 获取同步状态。 |
| setState | int newState |  | 设置同步状态。 |
| compareAndSetState | int expect,  int update | boolean | 原子的设置同步状态。 |

# 抽象方法

下面的方法是需要子类来实现的。

|  |  |  |  |
| --- | --- | --- | --- |
| 方法名 | 参数 | 返回值 | 说明 |
| tryAcquire | int arg | boolean | 在独占模式下获取锁，该方法必须检查当前线程是否已经持有了锁，如果已经持有了锁，则可以获取，这也是可重入锁的特性。 |
| tryRelease | int arg | boolean | 在独占模式下释放锁，只有在完全释放的条件下才会返回true。 |
| tryAcquireShared | int args | int | 在共享模式下获取锁，该方法必须检查当前线程是否已经持有了锁，如果持有了锁，则可以获取。该方法返回负数表示失败，0表示获取成功但不会在被共享模式下获取成功，返回正数n表示获取成功，之后仍可以被共享模式下获取n次。 |
| tryReleaseShared | int arg | boolean | 在共享模式下释放锁。 |
| isHeldExclusively |  | boolean | 检查当前线程是否独占该锁。 |

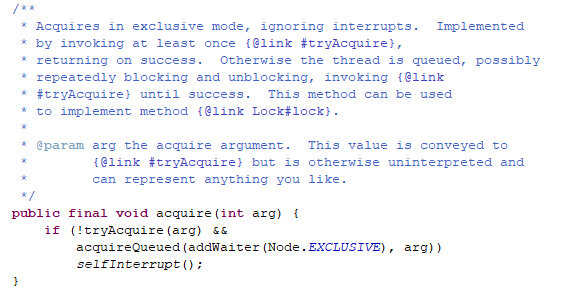
# 模板方法

下面简单汇总一下AQS中的模板方法，在第四节会结合源码详细简述方法实现的具体细节。

|  |  |
| --- | --- |
| 方法签名 | 说明 |
| void acquire(int) | 在独占模式下获取锁，不会响应中断。 |
| void acquireInterruptibly(int arg) | 在独占或是下获取锁，会响应中断。 |
| void acquireShared(int arg) | 在共享模式下获取锁，不会响应中断。 |
| void acquireSharedInterruptibly(int arg) | 在共享模式下获取锁，会响应中断。 |
| boolean release(int arg) | 在独占模式下释放锁。 |
| boolean releaseShared(int arg) | 在共享模式下释放锁。 |
| boolean tryAcquireNanos(int arg, long nanosTimeout) | 在独占模式下超时获取锁，会响应中断。 |
| boolean tryAcquireShreadNanos(int arg, long nanosTimeout) | 在共享模式下超时获取锁，会响应中断。 |

# 源码

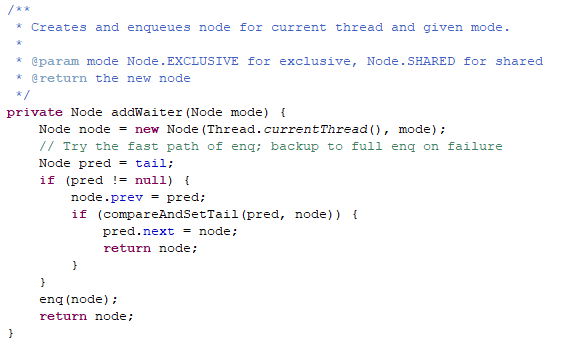
## 4.1acquire



在独占模式下获取锁，不会响应中断。acquire方法至少会调用一次tryAcquire方法尝试获取锁，tryAcquire返回true表示获取成功，否则将线程添加到CLH队列中，并不断重复的挂起线程直到tryAcquire返回true为止。acquire方法可用来实现Lock.lock方法。

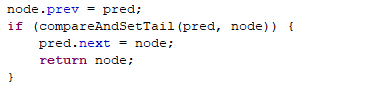
如果第一个线程调用并获取到了锁，那么此时head与tail为null。

## 4.2addWaiter



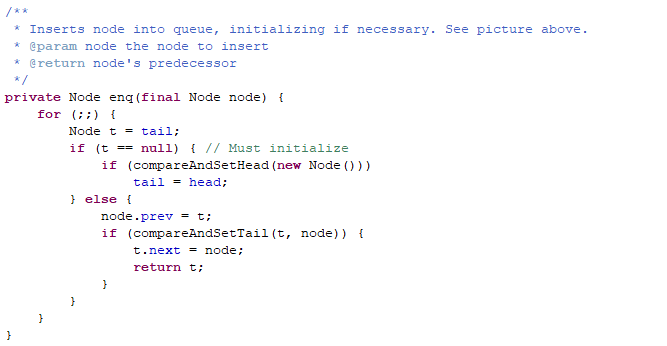
创建Node并将新节点添加到等待队列中。

（1）如果tail不为null并且CAS更新tail成功，会首先调用一次CAS操作更新tail属性。注意在更新tail之前会先将新节点的prev属性指向tail，然后更新tail，最后设置pred的next，如果先更新tail，会出现从head遍历队列到达不了tail的情况。



1. 如果tail为null（只有一种情况，tail还没有被初始化）或者CAS更新tail失败，都会调用enq方法。

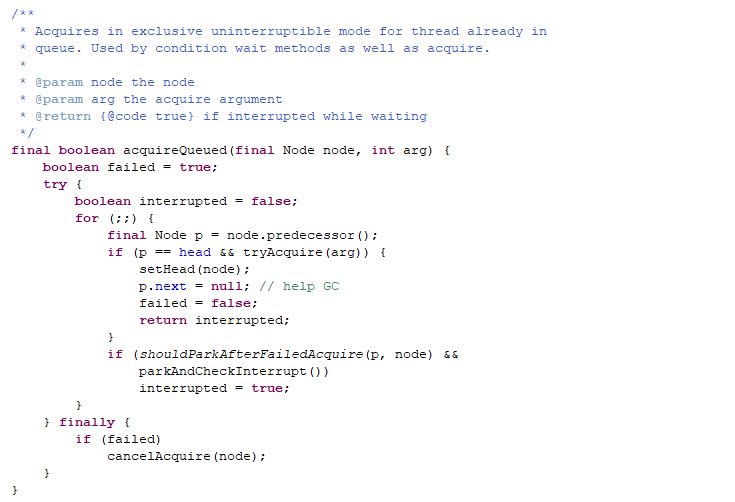
## 4.3enq



enq方法完成了tail与head属性的初始化以及节点的入队。

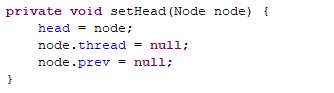
在入队之前首先确保了tail与head已经初始化，然后不停的尝试更新tail属性直到更新成功为止。

## 4.4acquireQueued



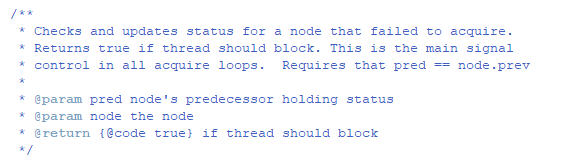
acquireQueued不停的使线程挂起直到tryAcquire返回true为止。只有前驱节点是头节点的线程才会尝试获取锁，其他都即将被挂起。

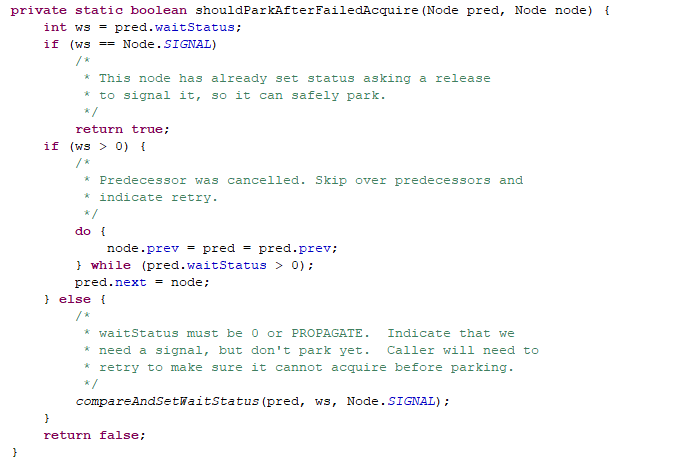
1. 前驱节点是头节点并获取锁成功（tryAcquire返回true），会设置head为当前节点并将原head的next设置null（GC回收）。然后返回。



1. 前驱节点不是头节点或者前驱节点为头节点但是获取锁失败（tryAcquire返回false，在非公平锁的实现中会出现），这种情况会检查挂起线程的条件并挂起线程。

## 4.5shouldParkAfterFailedAcquire

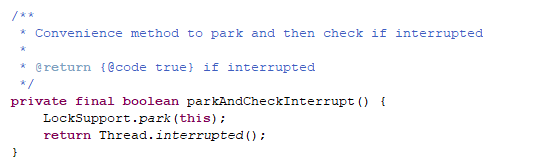




该方法用来设置前驱节点的状态，只有前驱节点在SINGNAL状态下当前节点线程才会被挂起。如果返回false，会重新进入acquireQueued的循环。

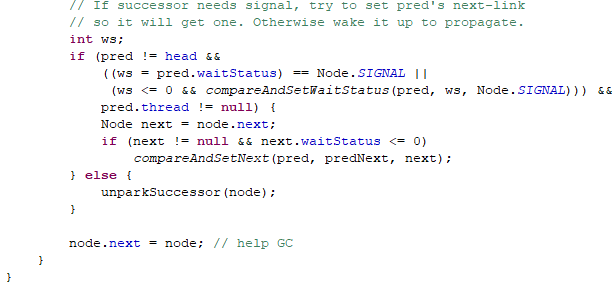
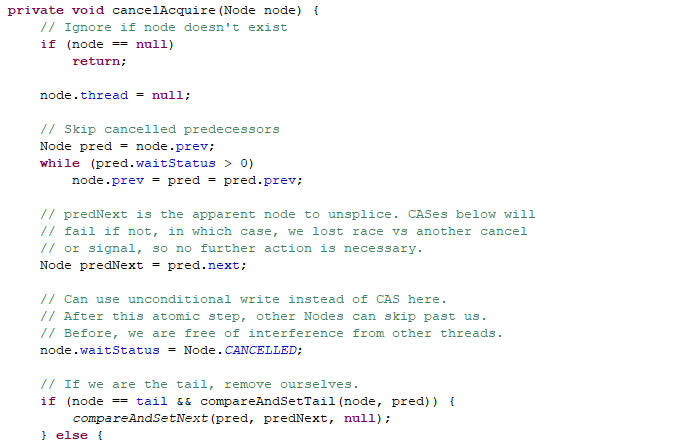
1. ws（前驱节点的状态）==Node.SINGNAL直接返回true。
2. ws>0（前驱节点为CANCELLED），会向前遍历找到第一个不为CANCELLED的节点，将CANCELLED的节点从链表中删除。返回false。
3. ws为0或者PROPAGATE，CAS更新前驱节点的状态为SINGNAL。

## 4.6parkAndCheckInterrupt



挂起线程并返回线程中断状态。

## 4.7cancelAcquire



当acquireQueued方法出现异常时，将会调用该方法处理关闭的节点。

1. 将当前节点的thread置null。



1. 获取第一个状态不为CANCELLED的前驱节点pred。



1. 记录pred的next，方便下文CAS更新。



1. 将当前节点状态设置为CANCELLED。



1. 当前节点为tail，并且CAS更新tail为pred成功，然后更新pred的next，这里更新失败也没有关系，说明有其他线程更新了next。



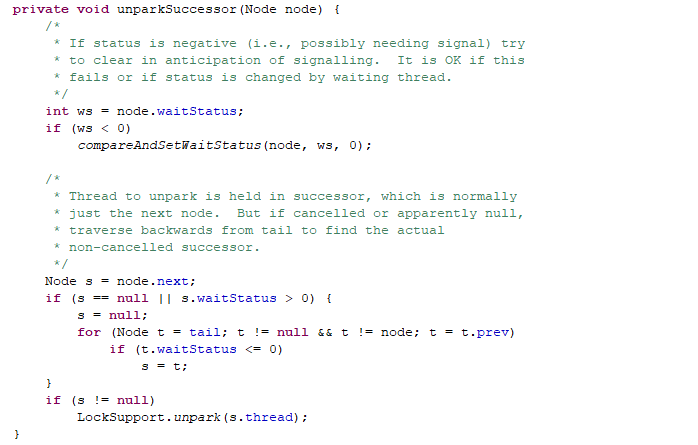
1. 当前节点不为tail或者CAS更新tail失败，这有可能有其他线程更新了tail。
2. 如果pred不为head，并且pred没有被设置为CANCELLED，并且pred的thread属性不为null，那么当当前节点存在后继节点并且后继节点状态不为CANCELLED时，CAS更新pred的next。

b.否则会唤醒线程。假设pred为头节点，tail指向新添加的线程A并挂起，这时可能没有线程正在执行，那么A将一直挂起无法被唤醒。所以这里调用unparkSuccessor唤醒线程。

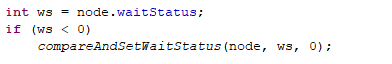
1. 将node的next指向自己。GC回收。



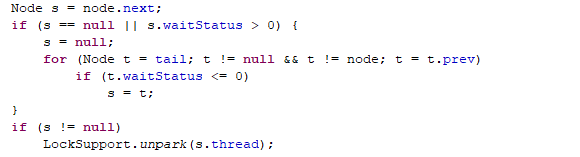
## 4.8unparkSuccessor



1. 将node的状态设置为0。这里可以让获取锁的线程在重新走一遍acquireQueued循环。



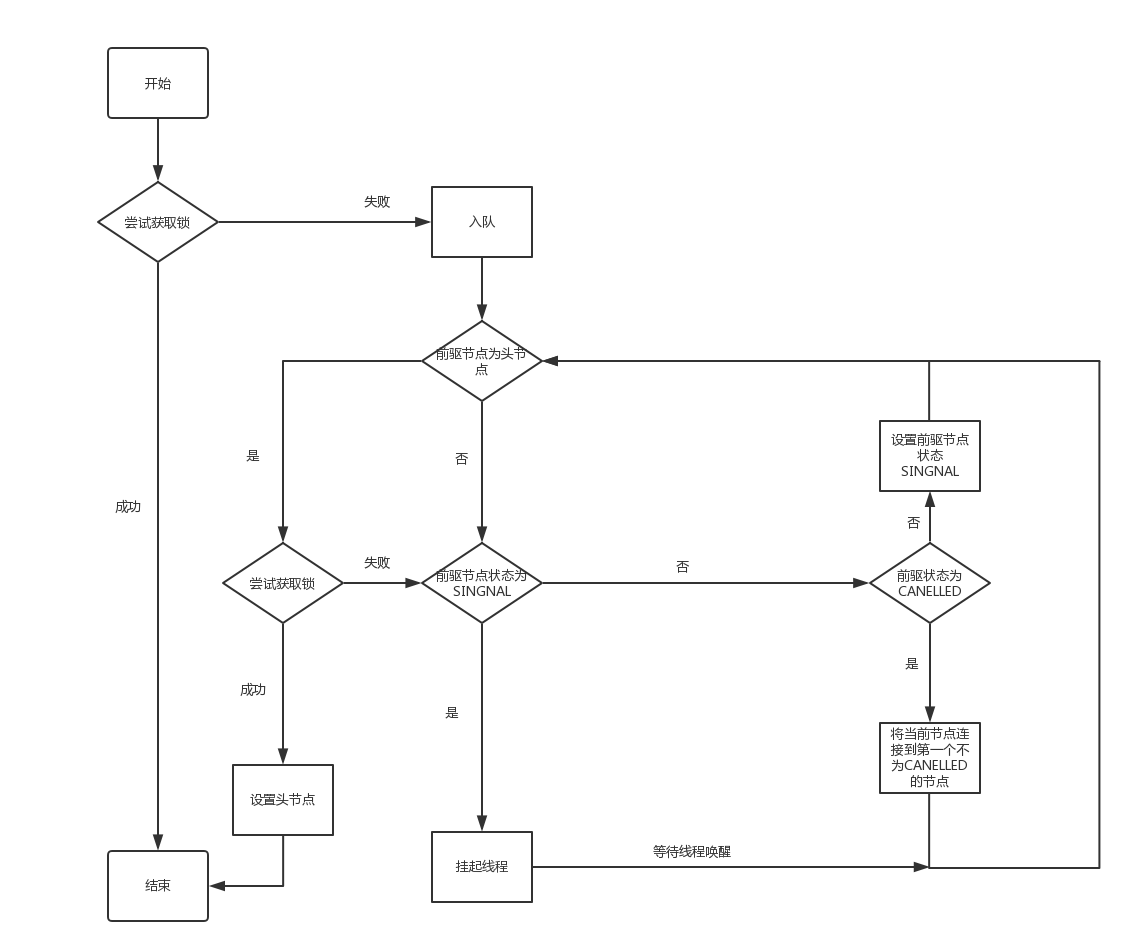
1. 唤醒后继节点线程。如果后继节点为空或者后继节点状态为CANCELLED，那么从tail往前遍历获取第一个不为null且状态不为CANCELLED的节点。如果该节点不为空，唤醒该节点线程。



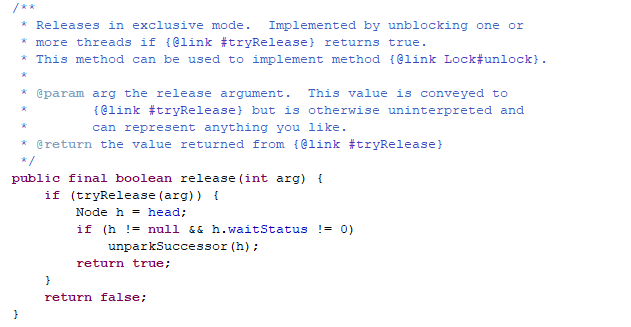
这里之所以从tail开始遍历，在前文addWaiter方法中已经说明过了。

## 4.9总结

下图是acquire调用的流程图。

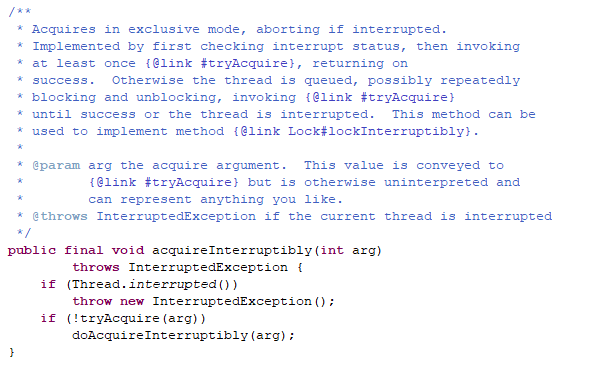


## 4.10release



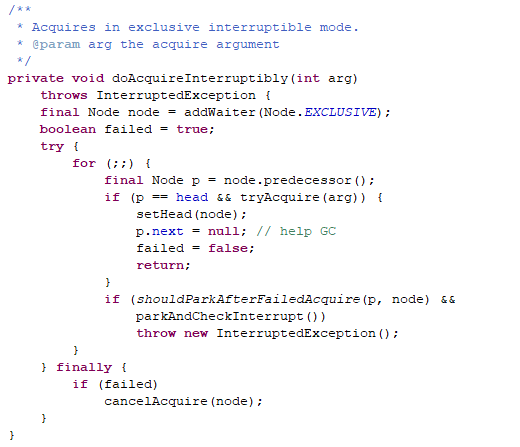
在独占模式下释放锁，可以用来实现Lock.unlock方法。该方法会唤醒挂起的线程。

## 4.11acquireInterruptibly

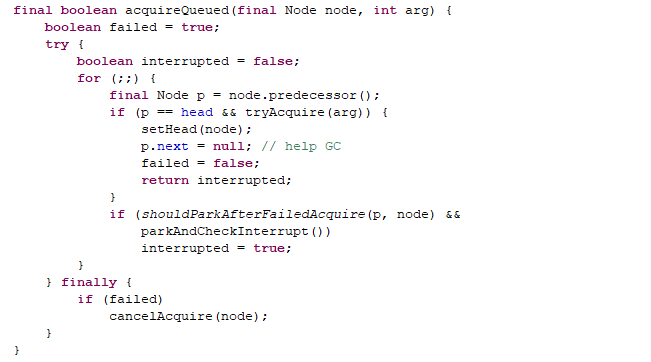


在独占模式下获取锁，会响应中断。首先检查线程的中断状态，如果已经中断，抛出中断异常。该方法可以用来实现Lock.lockInterruptibly。

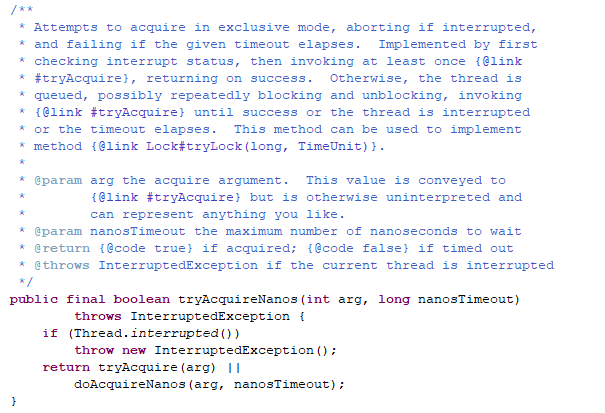
## 4.12doAcquireInterruptibly



可以看到这个方法跟acquireQueued方法的区别是在，这里获取到中断状态为true时，抛出了中断异常。而acquireQueued方法只是将interrupted变量设置为true，以下是acquireQueued的源码。

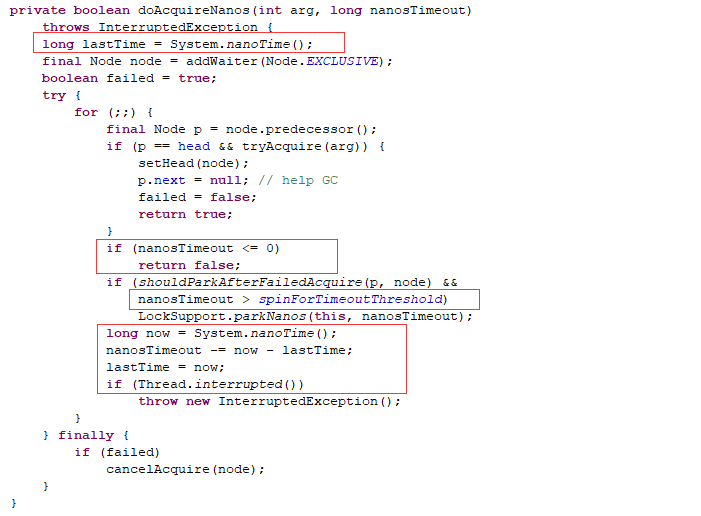


## 4.13tryAcquireNanos



不仅响应中断而且设置了超时时间，如果在指定时间内仍然没有获取到锁，返回false。

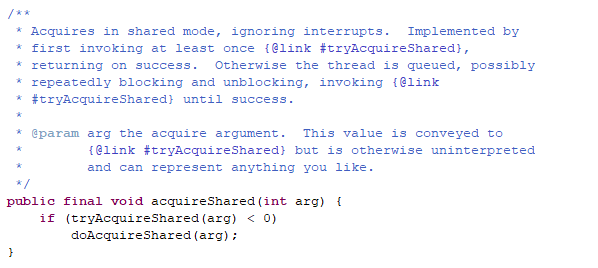
## 4.14doAcquireNanos



注意红框圈起来的部分。doAcquireNanos在尝试获取锁之前记录了执行时间lastTime，每次唤醒线程后会更新这个时间，并更新超时时间nanosTimeout。获取锁失败后，会判断超时时间，如果已经超时直接返回false，否则判断nanosTimeout>spinForTimeoutThreadhold，大于就挂起线程。

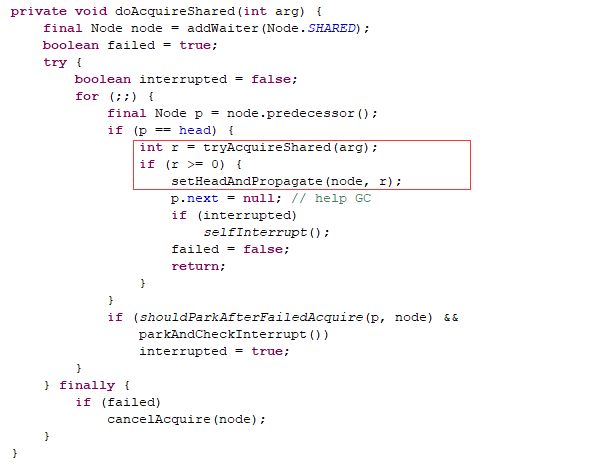


## 4.15acquireShared



在共享模式下获取锁不会响应中断。tryAcquireShared<0表示获取锁失败，该线程会进入等待队列。

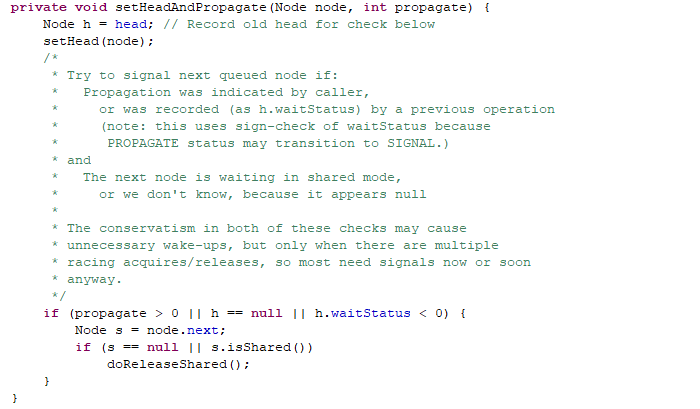
## 4.16doAcquireShared



注意圈红的部分，这是跟独占模式的主要区别。tryAcquireShared方法返回负数表示获取锁失败，返回0获取锁成功且后继节点的线程不会获取到锁，返回正数获取锁成功且后继线程可能会获取到锁。

当获取锁成功时，调用setHeadAndPropagate方法设置头节点并传播唤醒后继线程。

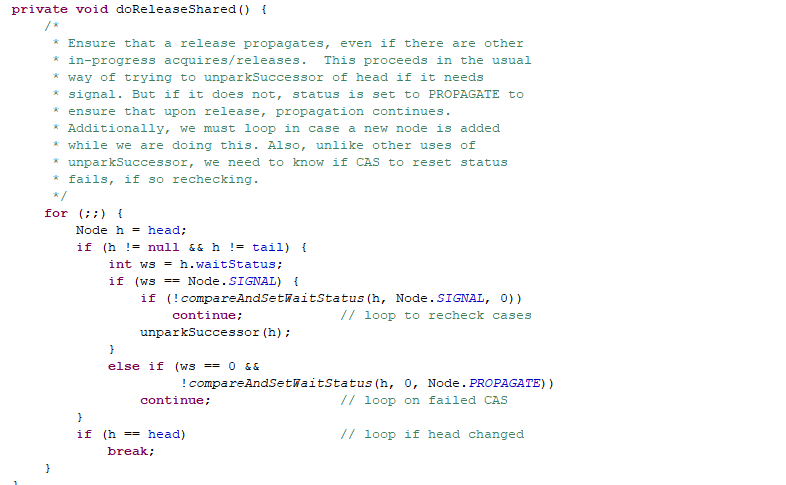
## 4.17setHeadAndPropagate



setHeadAndPropagate做了两件事，第一设置头节点，第二根据tryAcquireShared的返回值和当前节点状态以及后继节点的的模式判断是否要唤醒后继节点的线程。

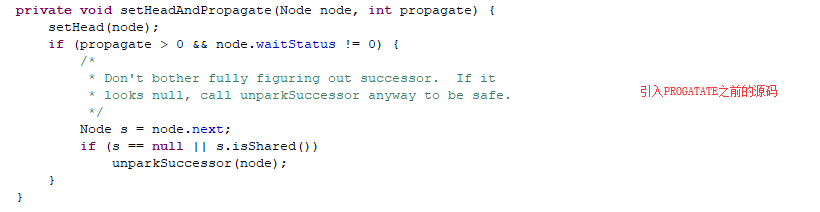
这里可能会有疑问，判断是否唤醒后继线程时，使用了progatate>0还有h.waitStatus<0这两个条件，下文会讲述不仅仅使用progatate的原因。

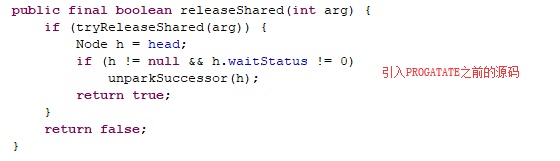
## 4.18doReleaseShared

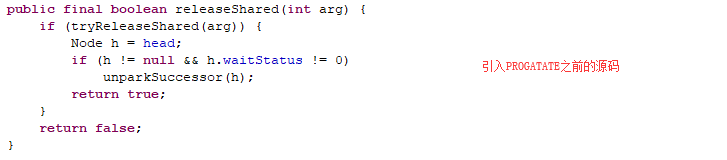


设置了线程的状态并唤醒了后继线程，当头节点状态为SINGNAL时，会设置为0并唤醒后继线程，当头节点状态为0时，会设置为PROGATATE。这里设置PROGATATE是防止在并发acquire与release时，出现线程hang情况，而引入PROGATATE这个状态字段就是为了解决这个bug的。

来看一下引入PROGATATE之前的源码。







来看下面一种情况state初始为2，有四个线程请求获取锁，t1，t2等待获取锁，t3，t4获取到了锁，内存布局为为head->t1->t2。假设t3先释放锁，t4后释放锁。

引入PROGATATE之前执行流程：  
①t3释放锁，调用upartSuccessor将head状态设置为0并唤醒后继线程t1。

②t1被唤醒，tryAcquireShared返回0。

③t4释放锁，t4看到的head的状态与①时刻一致，此时状态为0，不会调用uparkSuccessor。

④t1继续执行，调用setHeadAndPropagate，不满足propagate>0，不会唤醒后继线程。所以t2将一直挂起。

在看引入PROPAGATE之后的执行流程：

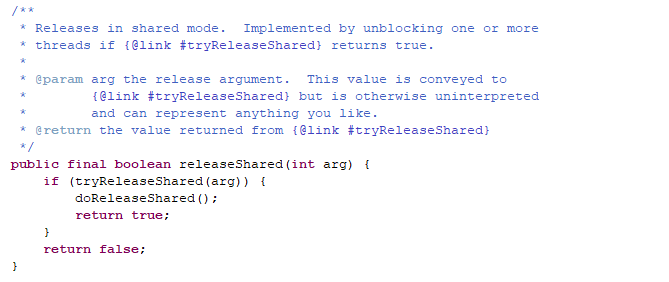
①t3释放锁，调用upartSuccessor将head状态设置为0并唤醒后继线程t1。

②t1被唤醒，tryAcquireShared返回0。

③t4释放锁，t4看到的head的状态与①时刻一致，状态为0，这是调用doReleaseShared，状态为0时设置为PROPAGATE。

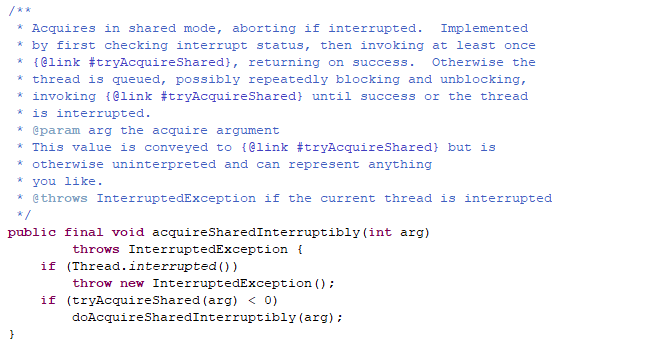
④t1继续执行，调用setHeadAndPropagate，满足h.waitStatus<0，将唤醒后继线程。

## 4.19releaseShared



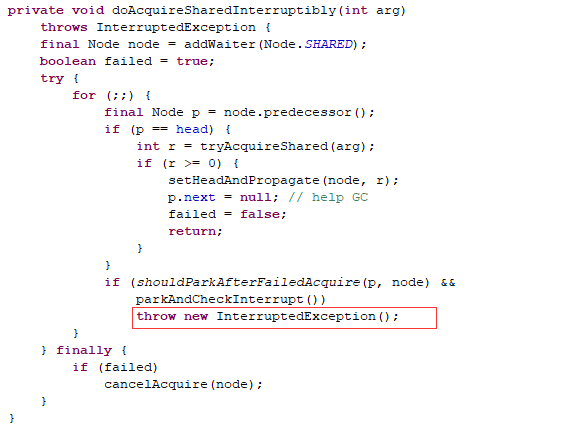
在共享模式下释放锁。调用了doReleaseShared核心方法。

## 4.20acquireSharedInterruptibly



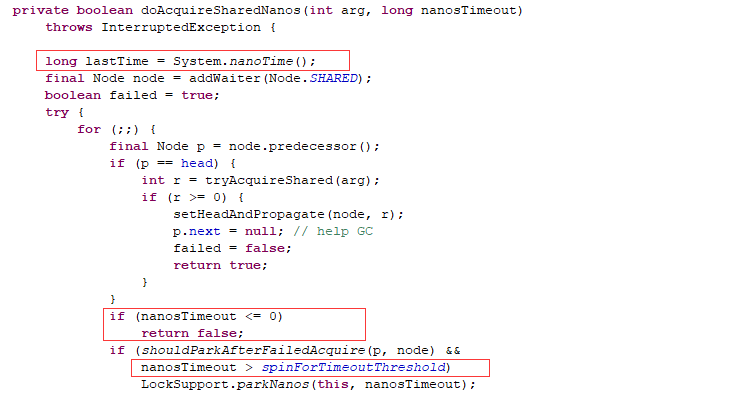
在共享模式下获取锁响应中断。

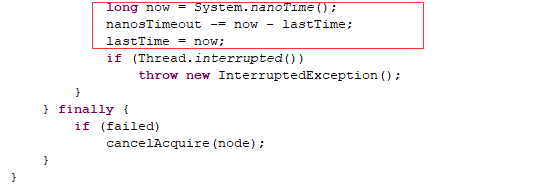
## 4.21doAcquireSharedInterruptibly



这里检查到中断时会抛出中断异常。

## 4.22doAcquireSharedNanos





在共享模式下超时获取锁响应中断，跟独占模式超时获取锁实现方式相同。