# **一行一行源码分析清楚AbstractQueuedSynchronizer**

https://javadoop.com/post/AbstractQueuedSynchronizer

创建时间: 2022-12-04 19:11:15

在分析 Java 并发包 java.util.concurrent 源码的时候，少不了需要了解 AbstractQueuedSynchronizer（以下简写AQS）这个抽象类，因为它是 Java 并发包的基础工具类，是实现 ReentrantLock、CountDownLatch、Semaphore、FutureTask 等类的基础。

Google 一下 AbstractQueuedSynchronizer，我们可以找到很多关于 AQS 的介绍，但是很多都没有介绍清楚，因为大部分文章没有把其中的一些关键的细节说清楚。

本文将从 ReentrantLock 的公平锁源码出发，分析下 AbstractQueuedSynchronizer 这个类是怎么工作的，希望能给大家提供一些简单的帮助。

申明以下几点：

1. 本文有点长，但还是挺简单，主要面向读者对象为并发编程的初学者，或者想要阅读 Java 并发包源码的开发者。对于新手来说，可能需要花好几个小时才能完全看懂，但是这时间肯定是值得的。
2. 源码环境 JDK1.7（1.8没啥变化），看到不懂或有疑惑的部分，最好能自己打开源码看看。Doug Lea 大神的代码写得真心不错。
3. 本文不分析共享模式，这样可以给读者减少很多负担，[第三篇文章](https://javadoop.com/post/AbstractQueuedSynchronizer-3)对共享模式进行了分析。而且也不分析 condition 部分，所以应该说很容易就可以看懂了。
4. 本文大量使用我们平时用得最多的 ReentrantLock 的概念，本质上来说是不正确的，读者应该清楚，AQS 不仅仅用来实现可重入锁，只是希望读者可以用锁来联想 AQS 的使用场景，降低阅读压力。
5. ReentrantLock 的公平锁和非公平锁只有一点点区别，[第二篇文章](https://javadoop.com/post/AbstractQueuedSynchronizer-2)做了介绍。
6. 评论区有读者反馈本文直接用代码说不友好，应该多配点流程图，这篇文章确实有这个问题。但是作为过来人，我想告诉大家，对于 AQS 来说，形式真的不重要，重要的是把细节说清楚。

## **AQS 结构**

先来看看 AQS 有哪些属性，搞清楚这些基本就知道 AQS 是什么套路了，毕竟可以猜嘛！

// 头结点，你直接把它当做 当前持有锁的线程 可能是最好理解的private transient volatile Node head;

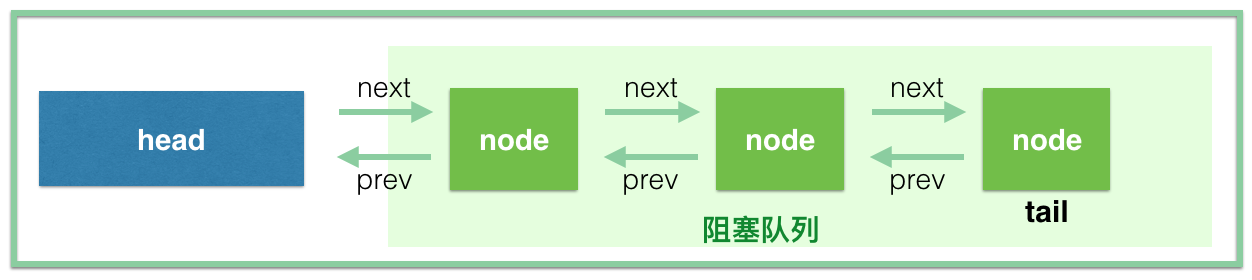
// 阻塞的尾节点，每个新的节点进来，都插入到最后，也就形成了一个链表private transient volatile Node tail;

// 这个是最重要的，代表当前锁的状态，0代表没有被占用，大于 0 代表有线程持有当前锁// 这个值可以大于 1，是因为锁可以重入，每次重入都加上 1private volatile int state;

// 代表当前持有独占锁的线程，举个最重要的使用例子，因为锁可以重入// reentrantLock.lock()可以嵌套调用多次，所以每次用这个来判断当前线程是否已经拥有了锁// if (currentThread == getExclusiveOwnerThread()) {state++}private transient Thread exclusiveOwnerThread; //继承自AbstractOwnableSynchronizer

怎么样，看样子应该是很简单的吧，毕竟也就四个属性啊。

AbstractQueuedSynchronizer 的等待队列示意如下所示，注意了，之后分析过程中所说的 queue，也就是阻塞队列****不包含 head，不包含 head，不包含 head****。



等待队列中每个线程被包装成一个 Node 实例，数据结构是链表，一起看看源码吧：

static final class Node {

// 标识节点当前在共享模式下

static final Node SHARED = new Node();

// 标识节点当前在独占模式下

static final Node EXCLUSIVE = null;

// ======== 下面的几个int常量是给waitStatus用的 ===========

/\*\* waitStatus value to indicate thread has cancelled \*/

// 代码此线程取消了争抢这个锁

static final int CANCELLED = 1;

/\*\* waitStatus value to indicate successor's thread needs unparking \*/

// 官方的描述是，其表示当前node的后继节点对应的线程需要被唤醒

static final int SIGNAL = -1;

/\*\* waitStatus value to indicate thread is waiting on condition \*/

// 本文不分析condition，所以略过吧，下一篇文章会介绍这个

static final int CONDITION = -2;

/\*\*

\* waitStatus value to indicate the next acquireShared should

\* unconditionally propagate

\*/

// 同样的不分析，略过吧

static final int PROPAGATE = -3;

// =====================================================

// 取值为上面的1、-1、-2、-3，或者0(以后会讲到)

// 这么理解，暂时只需要知道如果这个值 大于0 代表此线程取消了等待，

// ps: 半天抢不到锁，不抢了，ReentrantLock是可以指定timeouot的。。。

volatile int waitStatus;

// 前驱节点的引用

volatile Node prev;

// 后继节点的引用

volatile Node next;

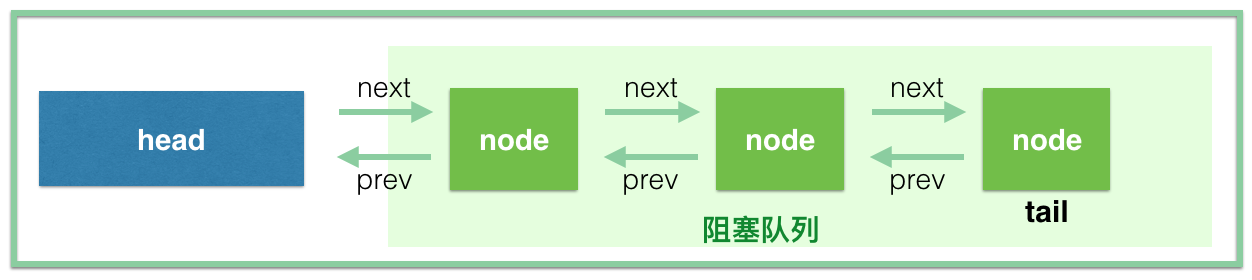
// 这个就是线程本尊

volatile Thread thread;

}

Node 的数据结构其实也挺简单的，就是 thread + waitStatus + pre + next 四个属性而已，大家先要有这个概念在心里。

上面的是基础知识，后面会多次用到，心里要时刻记着它们，心里想着这个结构图就可以了。下面，我们开始说 ReentrantLock 的公平锁。再次强调，我说的阻塞队列不包含 head 节点。



首先，我们先看下 ReentrantLock 的使用方式。

// 我用个web开发中的service概念吧public class OrderService {

// 使用static，这样每个线程拿到的是同一把锁，当然，spring mvc中service默认就是单例，别纠结这个

private static ReentrantLock reentrantLock = new ReentrantLock(true);

public void createOrder() {

// 比如我们同一时间，只允许一个线程创建订单

reentrantLock.lock();

// 通常，lock 之后紧跟着 try 语句

try {

// 这块代码同一时间只能有一个线程进来(获取到锁的线程)，

// 其他的线程在lock()方法上阻塞，等待获取到锁，再进来

// 执行代码...

// 执行代码...

// 执行代码...

} finally {

// 释放锁

reentrantLock.unlock();

}

}

}

ReentrantLock 在内部用了内部类 Sync 来管理锁，所以真正的获取锁和释放锁是由 Sync 的实现类来控制的。

abstract static class Sync extends AbstractQueuedSynchronizer {

}

Sync 有两个实现，分别为 NonfairSync（非公平锁）和 FairSync（公平锁），我们看 FairSync 部分。

public ReentrantLock(boolean fair) {

sync = fair ? new FairSync() : new NonfairSync();

}

## **线程抢锁**

很多人肯定开始嫌弃上面废话太多了，下面跟着代码走，我就不废话了。

static final class FairSync extends Sync {

private static final long serialVersionUID = -3000897897090466540L;

// 争锁

final void lock() {

acquire(1);

}

// 来自父类AQS，我直接贴过来这边，下面分析的时候同样会这样做，不会给读者带来阅读压力

// 我们看到，这个方法，如果tryAcquire(arg) 返回true, 也就结束了。

// 否则，acquireQueued方法会将线程压到队列中

public final void acquire(int arg) { // 此时 arg == 1

// 首先调用tryAcquire(1)一下，名字上就知道，这个只是试一试

// 因为有可能直接就成功了呢，也就不需要进队列排队了，

// 对于公平锁的语义就是：本来就没人持有锁，根本没必要进队列等待(又是挂起，又是等待被唤醒的)

if (!tryAcquire(arg) &&

// tryAcquire(arg)没有成功，这个时候需要把当前线程挂起，放到阻塞队列中。

acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) {

selfInterrupt();

}

}

/\*\*

\* Fair version of tryAcquire. Don't grant access unless

\* recursive call or no waiters or is first.

\*/

// 尝试直接获取锁，返回值是boolean，代表是否获取到锁

// 返回true：1.没有线程在等待锁；2.重入锁，线程本来就持有锁，也就可以理所当然可以直接获取

protected final boolean tryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

// state == 0 此时此刻没有线程持有锁

if (c == 0) {

// 虽然此时此刻锁是可以用的，但是这是公平锁，既然是公平，就得讲究先来后到，

// 看看有没有别人在队列中等了半天了

if (!hasQueuedPredecessors() &&

// 如果没有线程在等待，那就用CAS尝试一下，成功了就获取到锁了，

// 不成功的话，只能说明一个问题，就在刚刚几乎同一时刻有个线程抢先了 =\_=

// 因为刚刚还没人的，我判断过了

compareAndSetState(0, acquires)) {

// 到这里就是获取到锁了，标记一下，告诉大家，现在是我占用了锁

setExclusiveOwnerThread(current);

return true;

}

}

// 会进入这个else if分支，说明是重入了，需要操作：state=state+1

// 这里不存在并发问题

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0)

throw new Error("Maximum lock count exceeded");

setState(nextc);

return true;

}

// 如果到这里，说明前面的if和else if都没有返回true，说明没有获取到锁

// 回到上面一个外层调用方法继续看:

// if (!tryAcquire(arg)

// && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

// selfInterrupt();

return false;

}

// 假设tryAcquire(arg) 返回false，那么代码将执行：

// acquireQueued(addWaiter(Node.EXCLUSIVE), arg)，

// 这个方法，首先需要执行：addWaiter(Node.EXCLUSIVE)

/\*\*

\* Creates and enqueues node for current thread and given mode.

\*

\* @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared

\* @return the new node

\*/

// 此方法的作用是把线程包装成node，同时进入到队列中

// 参数mode此时是Node.EXCLUSIVE，代表独占模式

private Node addWaiter(Node mode) {

Node node = new Node(Thread.currentThread(), mode);

// Try the fast path of enq; backup to full enq on failure

// 以下几行代码想把当前node加到链表的最后面去，也就是进到阻塞队列的最后

Node pred = tail;

// tail!=null => 队列不为空(tail==head的时候，其实队列是空的，不过不管这个吧)

if (pred != null) {

// 将当前的队尾节点，设置为自己的前驱

node.prev = pred;

// 用CAS把自己设置为队尾, 如果成功后，tail == node 了，这个节点成为阻塞队列新的尾巴

if (compareAndSetTail(pred, node)) {

// 进到这里说明设置成功，当前node==tail, 将自己与之前的队尾相连，

// 上面已经有 node.prev = pred，加上下面这句，也就实现了和之前的尾节点双向连接了

pred.next = node;

// 线程入队了，可以返回了

return node;

}

}

// 仔细看看上面的代码，如果会到这里，

// 说明 pred==null(队列是空的) 或者 CAS失败(有线程在竞争入队)

// 读者一定要跟上思路，如果没有跟上，建议先不要往下读了，往回仔细看，否则会浪费时间的

enq(node);

return node;

}

/\*\*

\* Inserts node into queue, initializing if necessary. See picture above.

\* @param node the node to insert

\* @return node's predecessor

\*/

// 采用自旋的方式入队

// 之前说过，到这个方法只有两种可能：等待队列为空，或者有线程竞争入队，

// 自旋在这边的语义是：CAS设置tail过程中，竞争一次竞争不到，我就多次竞争，总会排到的

private Node enq(final Node node) {

for (;;) {

Node t = tail;

// 之前说过，队列为空也会进来这里

if (t == null) { // Must initialize

// 初始化head节点

// 细心的读者会知道原来 head 和 tail 初始化的时候都是 null 的

// 还是一步CAS，你懂的，现在可能是很多线程同时进来呢

if (compareAndSetHead(new Node()))

// 给后面用：这个时候head节点的waitStatus==0, 看new Node()构造方法就知道了

// 这个时候有了head，但是tail还是null，设置一下，

// 把tail指向head，放心，马上就有线程要来了，到时候tail就要被抢了

// 注意：这里只是设置了tail=head，这里可没return哦，没有return，没有return

// 所以，设置完了以后，继续for循环，下次就到下面的else分支了

tail = head;

} else {

// 下面几行，和上一个方法 addWaiter 是一样的，

// 只是这个套在无限循环里，反正就是将当前线程排到队尾，有线程竞争的话排不上重复排

node.prev = t;

if (compareAndSetTail(t, node)) {

t.next = node;

return t;

}

}

}

}

// 现在，又回到这段代码了

// if (!tryAcquire(arg)

// && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

// selfInterrupt();

// 下面这个方法，参数node，经过addWaiter(Node.EXCLUSIVE)，此时已经进入阻塞队列

// 注意一下：如果acquireQueued(addWaiter(Node.EXCLUSIVE), arg))返回true的话，

// 意味着上面这段代码将进入selfInterrupt()，所以正常情况下，下面应该返回false

// 这个方法非常重要，应该说真正的线程挂起，然后被唤醒后去获取锁，都在这个方法里了

final boolean acquireQueued(final Node node, int arg) {

boolean failed = true;

try {

boolean interrupted = false;

for (;;) {

final Node p = node.predecessor();

// p == head 说明当前节点虽然进到了阻塞队列，但是是阻塞队列的第一个，因为它的前驱是head

// 注意，阻塞队列不包含head节点，head一般指的是占有锁的线程，head后面的才称为阻塞队列

// 所以当前节点可以去试抢一下锁

// 这里我们说一下，为什么可以去试试：

// 首先，它是队头，这个是第一个条件，其次，当前的head有可能是刚刚初始化的node，

// enq(node) 方法里面有提到，head是延时初始化的，而且new Node()的时候没有设置任何线程

// 也就是说，当前的head不属于任何一个线程，所以作为队头，可以去试一试，

// tryAcquire已经分析过了, 忘记了请往前看一下，就是简单用CAS试操作一下state

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

setHead(node);

p.next = null; // help GC

failed = false;

return interrupted;

}

// 到这里，说明上面的if分支没有成功，要么当前node本来就不是队头，

// 要么就是tryAcquire(arg)没有抢赢别人，继续往下看

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

interrupted = true;

}

} finally {

// 什么时候 failed 会为 true???

// tryAcquire() 方法抛异常的情况

if (failed)

cancelAcquire(node);

}

}

/\*\*

\* Checks and updates status for a node that failed to acquire.

\* Returns true if thread should block. This is the main signal

\* control in all acquire loops. Requires that pred == node.prev

\*

\* @param pred node's predecessor holding status

\* @param node the node

\* @return {@code true} if thread should block

\*/

// 刚刚说过，会到这里就是没有抢到锁呗，这个方法说的是："当前线程没有抢到锁，是否需要挂起当前线程？"

// 第一个参数是前驱节点，第二个参数才是代表当前线程的节点

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

int ws = pred.waitStatus;

// 前驱节点的 waitStatus == -1 ，说明前驱节点状态正常，当前线程需要挂起，直接可以返回true

if (ws == Node.SIGNAL)

/\*

\* This node has already set status asking a release

\* to signal it, so it can safely park.

\*/

return true;

// 前驱节点 waitStatus大于0 ，之前说过，大于0 说明前驱节点取消了排队。

// 这里需要知道这点：进入阻塞队列排队的线程会被挂起，而唤醒的操作是由前驱节点完成的。

// 所以下面这块代码说的是将当前节点的prev指向waitStatus<=0的节点，

// 简单说，就是为了找个好爹，因为你还得依赖它来唤醒呢，如果前驱节点取消了排队，

// 找前驱节点的前驱节点做爹，往前遍历总能找到一个好爹的

if (ws > 0) {

/\*

\* Predecessor was cancelled. Skip over predecessors and

\* indicate retry.

\*/

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.

\*/

// 仔细想想，如果进入到这个分支意味着什么

// 前驱节点的waitStatus不等于-1和1，那也就是只可能是0，-2，-3

// 在我们前面的源码中，都没有看到有设置waitStatus的，所以每个新的node入队时，waitStatu都是0

// 正常情况下，前驱节点是之前的 tail，那么它的 waitStatus 应该是 0

// 用CAS将前驱节点的waitStatus设置为Node.SIGNAL(也就是-1)

compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

}

// 这个方法返回 false，那么会再走一次 for 循序，

// 然后再次进来此方法，此时会从第一个分支返回 true

return false;

}

// private static boolean shouldParkAfterFailedAcquire(Node pred, Node node)

// 这个方法结束根据返回值我们简单分析下：

// 如果返回true, 说明前驱节点的waitStatus==-1，是正常情况，那么当前线程需要被挂起，等待以后被唤醒

// 我们也说过，以后是被前驱节点唤醒，就等着前驱节点拿到锁，然后释放锁的时候叫你好了

// 如果返回false, 说明当前不需要被挂起，为什么呢？往后看

// 跳回到前面是这个方法

// if (shouldParkAfterFailedAcquire(p, node) &&

// parkAndCheckInterrupt())

// interrupted = true;

// 1. 如果shouldParkAfterFailedAcquire(p, node)返回true，

// 那么需要执行parkAndCheckInterrupt():

// 这个方法很简单，因为前面返回true，所以需要挂起线程，这个方法就是负责挂起线程的

// 这里用了LockSupport.park(this)来挂起线程，然后就停在这里了，等待被唤醒=======

private final boolean parkAndCheckInterrupt() {

LockSupport.park(this);

return Thread.interrupted();

}

// 2. 接下来说说如果shouldParkAfterFailedAcquire(p, node)返回false的情况

// 仔细看shouldParkAfterFailedAcquire(p, node)，我们可以发现，其实第一次进来的时候，一般都不会返回true的，原因很简单，前驱节点的waitStatus=-1是依赖于后继节点设置的。也就是说，我都还没给前驱设置-1呢，怎么可能是true呢，但是要看到，这个方法是套在循环里的，所以第二次进来的时候状态就是-1了。

// 解释下为什么shouldParkAfterFailedAcquire(p, node)返回false的时候不直接挂起线程：

// => 是为了应对在经过这个方法后，node已经是head的直接后继节点了。剩下的读者自己想想吧。

}

说到这里，也就明白了，多看几遍 final boolean acquireQueued(final Node node, int arg) 这个方法吧。自己推演下各个分支怎么走，哪种情况下会发生什么，走到哪里。

## **解锁操作**

最后，就是还需要介绍下唤醒的动作了。我们知道，正常情况下，如果线程没获取到锁，线程会被 LockSupport.park(this); 挂起停止，等待被唤醒。

// 唤醒的代码还是比较简单的，你如果上面加锁的都看懂了，下面都不需要看就知道怎么回事了public void unlock() {

sync.release(1);

}

public final boolean release(int arg) {

// 往后看吧

if (tryRelease(arg)) {

Node h = head;

if (h != null && h.waitStatus != 0)

unparkSuccessor(h);

return true;

}

return false;

}

// 回到ReentrantLock看tryRelease方法protected final boolean tryRelease(int releases) {

int c = getState() - releases;

if (Thread.currentThread() != getExclusiveOwnerThread())

throw new IllegalMonitorStateException();

// 是否完全释放锁

boolean free = false;

// 其实就是重入的问题，如果c==0，也就是说没有嵌套锁了，可以释放了，否则还不能释放掉

if (c == 0) {

free = true;

setExclusiveOwnerThread(null);

}

setState(c);

return free;

}

/\*\*

\* Wakes up node's successor, if one exists.

\*

\* @param node the node

\*/// 唤醒后继节点// 从上面调用处知道，参数node是head头结点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;

// 如果head节点当前waitStatus<0, 将其修改为0

if (ws < 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.

\*/

// 下面的代码就是唤醒后继节点，但是有可能后继节点取消了等待（waitStatus==1）

// 从队尾往前找，找到waitStatus<=0的所有节点中排在最前面的

Node s = node.next;

if (s == null || s.waitStatus > 0) {

s = null;

// 从后往前找，仔细看代码，不必担心中间有节点取消(waitStatus==1)的情况

for (Node t = tail; t != null && t != node; t = t.prev)

if (t.waitStatus <= 0)

s = t;

}

if (s != null)

// 唤醒线程

LockSupport.unpark(s.thread);

}

唤醒线程以后，被唤醒的线程将从以下代码中继续往前走：

private final boolean parkAndCheckInterrupt() {

LockSupport.park(this); // 刚刚线程被挂起在这里了

return Thread.interrupted();

}// 又回到这个方法了：acquireQueued(final Node node, int arg)，这个时候，node的前驱是head了

好了，后面就不分析源码了，剩下的还有问题自己去仔细看看代码吧。

## **总结**

总结一下吧。

在并发环境下，加锁和解锁需要以下三个部件的协调：

1. 锁状态。我们要知道锁是不是被别的线程占有了，这个就是 state 的作用，它为 0 的时候代表没有线程占有锁，可以去争抢这个锁，用 CAS 将 state 设为 1，如果 CAS 成功，说明抢到了锁，这样其他线程就抢不到了，如果锁重入的话，state进行 +1 就可以，解锁就是减 1，直到 state 又变为 0，代表释放锁，所以 lock() 和 unlock() 必须要配对啊。然后唤醒等待队列中的第一个线程，让其来占有锁。
2. 线程的阻塞和解除阻塞。AQS 中采用了 LockSupport.park(thread) 来挂起线程，用 unpark 来唤醒线程。
3. 阻塞队列。因为争抢锁的线程可能很多，但是只能有一个线程拿到锁，其他的线程都必须等待，这个时候就需要一个 queue 来管理这些线程，AQS 用的是一个 FIFO 的队列，就是一个链表，每个 node 都持有后继节点的引用。AQS 采用了 CLH 锁的变体来实现，感兴趣的读者可以参考这篇文章[关于CLH的介绍](http://coderbee.net/index.php/concurrent/20131115/577)，写得简单明了。

## **示例图解析**

下面属于回顾环节，用简单的示例来说一遍，如果上面的有些东西没看懂，这里还有一次帮助你理解的机会。

首先，第一个线程调用 reentrantLock.lock()，翻到最前面可以发现，tryAcquire(1) 直接就返回 true 了，结束。只是设置了 state=1，连 head 都没有初始化，更谈不上什么阻塞队列了。要是线程 1 调用 unlock() 了，才有线程 2 来，那世界就太太太平了，完全没有交集嘛，那我还要 AQS 干嘛。

如果线程 1 没有调用 unlock() 之前，线程 2 调用了 lock(), 想想会发生什么？

线程 2 会初始化 head【new Node()】，同时线程 2 也会插入到阻塞队列并挂起 (注意看这里是一个 for 循环，而且设置 head 和 tail 的部分是不 return 的，只有入队成功才会跳出循环)

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;

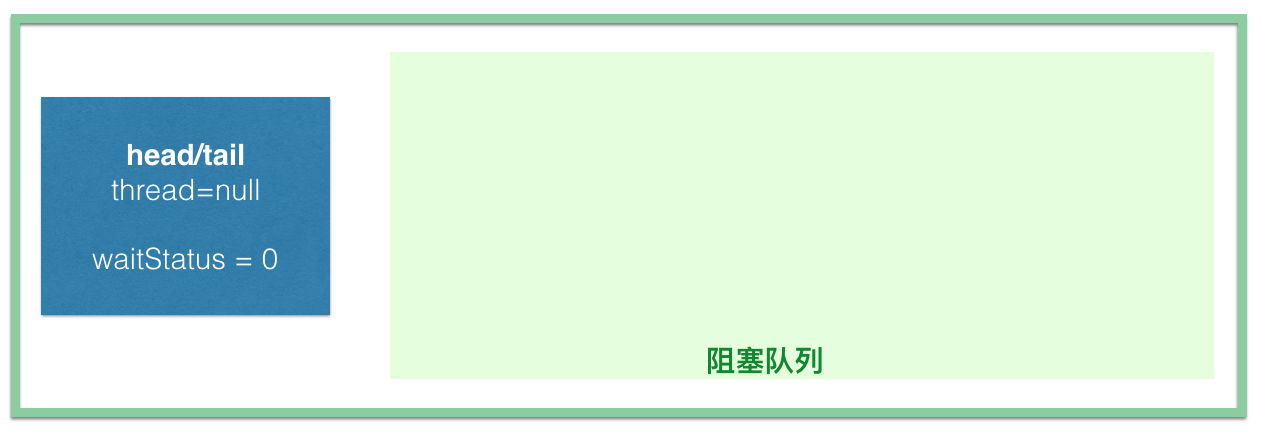
}

}

}

}

首先，是线程 2 初始化 head 节点，此时 head==tail, waitStatus==0



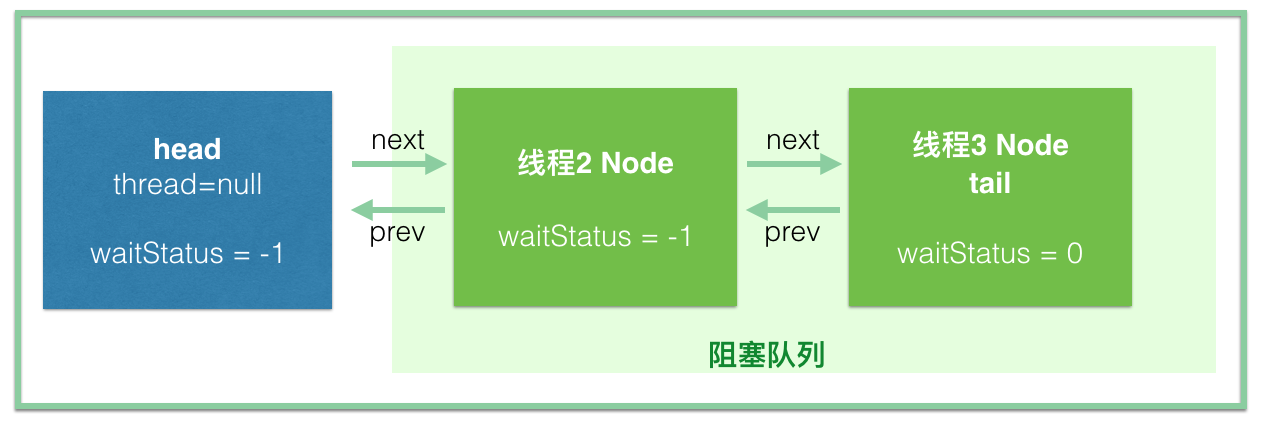
然后线程 2 入队：



同时我们也要看此时节点的 waitStatus，我们知道 head 节点是线程 2 初始化的，此时的 waitStatus 没有设置， java 默认会设置为 0，但是到 shouldParkAfterFailedAcquire 这个方法的时候，线程 2 会把前驱节点，也就是 head 的waitStatus设置为 -1。

那线程 2 节点此时的 waitStatus 是多少呢，由于没有设置，所以是 0；

如果线程 3 此时再进来，直接插到线程 2 的后面就可以了，此时线程 3 的 waitStatus 是 0，到 shouldParkAfterFailedAcquire 方法的时候把前驱节点线程 2 的 waitStatus 设置为 -1。



这里可以简单说下 waitStatus 中 SIGNAL(-1) 状态的意思，Doug Lea 注释的是：代表后继节点需要被唤醒。也就是说这个 waitStatus 其实代表的不是自己的状态，而是后继节点的状态，我们知道，每个 node 在入队的时候，都会把前驱节点的状态改为 SIGNAL，然后阻塞，等待被前驱唤醒。这里涉及的是两个问题：有线程取消了排队、唤醒操作。其实本质是一样的，读者也可以顺着 “waitStatus代表后继节点的状态” 这种思路去看一遍源码。

（全文完）