# 前言

在java.util.concurrent.locks包中有很多Lock的实现类，常用的有ReentrantLock、ReadWriteLock（实现类ReentrantReadWriteLock），内部实现都依赖AbstractQueuedSynchronizer类，接下去让我们看看Doug Lea大神是如何使用一个普通类就完成了代码块的并发访问控制。为了方便，本文中使用AQS代替AbstractQueuedSynchronizer。

# 定义

public abstract class AbstractQueuedSynchronizer extends

AbstractOwnableSynchronizer implements java.io.Serializable {

//等待队列的头节点

private transient volatile Node head;

//等待队列的尾节点

private transient volatile Node tail;

//同步状态

private volatile int state;

protected final int getState() { return state;}

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

...

}

队列同步器AQS是用来构建锁或其他同步组件的基础框架，内部使用一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作，其中内部状态state，等待队列的头节点head和尾节点tail，都是通过volatile修饰，保证了多线程之间的可见。

在深入实现原理之前，我们先看看内部的FIFO队列是如何实现的。

static final class Node {

static final Node SHARED = new Node();

static final Node EXCLUSIVE = null;

static final int CANCELLED = 1;

static final int SIGNAL = -1;

static final int CONDITION = -2;

static final int PROPAGATE = -3;

volatile int waitStatus;

volatile Node prev;

volatile Node next;

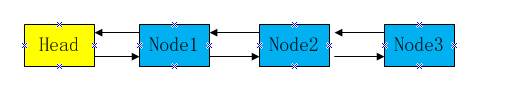
volatile Thread thread;

Node nextWaiter;

...

}

先来一张形象的图（该图其实是网上找的）



FIFO.png

黄色节点是默认head节点，其实是一个空节点，我觉得可以理解成代表当前持有锁的线程，每当有线程竞争失败，都是插入到队列的尾节点，tail节点始终指向队列中的最后一个元素。

每个节点中， 除了存储了当前线程，前后节点的引用以外，还有一个waitStatus变量，用于描述节点当前的状态。多线程并发执行时，队列中会有多个节点存在，这个waitStatus其实代表对应线程的状态：有的线程可能获取锁因为某些原因放弃竞争；有的线程在等待满足条件，满足之后才能执行等等。一共有4中状态：

CANCELLED 取消状态

SIGNAL 等待触发状态

CONDITION 等待条件状态

PROPAGATE 状态需要向后传播

等待队列是FIFO先进先出，只有前一个节点的状态为SIGNAL时，当前节点的线程才能被挂起。

# 实现原理

子类重写tryAcquire和tryRelease方法通过CAS指令修改状态变量state。

public final void acquire(int arg) {

if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

## 线程获取锁过程

下列步骤中线程A和B进行竞争。

1. 线程A执行CAS指令成功，state值被修改并返回true，线程A继续执行。
2. 线程A执行CAS指令失败，说明线程B也在执行CAS指令且成功，这种情况下线程A会执行步骤3。
3. 生成新Node节点node，并通过CAS指令插入到等待队列的队尾（同一时刻可能会有多个Node节点插入到等待队列中），如果tail节点为空，则将head节点指向一个空节点（代表线程B），具体实现如下：

private Node addWaiter(Node mode) {

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

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

Node pred = tail;

if (pred != null) {

node.prev = pred;

if (compareAndSetTail(pred, node)) {

pred.next = node;

return node;

}

}

enq(node);

return node;

}

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;

}

}

}

}

1. node插入到队尾后，该线程不会立马挂起，会进行自旋操作。因为在node的插入过程，线程B（即之前没有阻塞的线程）可能已经执行完成，所以要判断该node的前一个节点pred是否为head节点（代表线程B），如果pred == head，表明当前节点是队列中第一个“有效的”节点，因此再次尝试tryAcquire获取锁，

1、如果成功获取到锁，表明线程B已经执行完成，线程A不需要挂起。

2、如果获取失败，表示线程B还未完成，至少还未修改state值。进行步骤5。

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);

}

}

1. 前面我们已经说过只有前一个节点pred的线程状态为SIGNAL时，当前节点的线程才能被挂起。

1、如果pred的waitStatus == 0，则通过CAS指令修改waitStatus为Node.SIGNAL。

2、如果pred的waitStatus > 0，表明pred的线程状态CANCELLED，需从队列中删除。

3、如果pred的waitStatus为Node.SIGNAL，则通过LockSupport.park()方法把线程A挂起，并等待被唤醒，被唤醒后进入步骤6。

具体实现如下：

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

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.

\*/

return true;

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.

\*/

compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

}

return false;

}

1. 线程每次被唤醒时，都要进行中断检测，如果发现当前线程被中断，那么抛出InterruptedException并退出循环。从无限循环的代码可以看出，并不是被唤醒的线程一定能获得锁，必须调用tryAccquire重新竞争，因为锁是非公平的，有可能被新加入的线程获得，从而导致刚被唤醒的线程再次被阻塞，这个细节充分体现了“非公平”的精髓。

## 线程释放锁过程

1. 如果头结点head的waitStatus值为-1，则用CAS指令重置为0；
2. 找到waitStatus值小于0的节点s，通过LockSupport.unpark(s.thread)唤醒线程。

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)

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.

\*/

Node s = node.next;

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;

}

if (s != null)

LockSupport.unpark(s.thread);

}

# 总结

Doug Lea大神的思路跳跃的太快，把CAS指令玩的出神入化，以至于有些逻辑反反复复debug很多次才明白。