ReentrantLock的源码分析

<https://www.jianshu.com/p/af1876625ecf>

概念

ReentrandLock是基于**AbstractQueuedSynchronizer**实现的，所以先来简单介绍下AQS。

我们最熟悉的同步锁应该是**synchronized**，熟悉java的应该都用过这个，它是通过底层的 **monitorenter** 和 **monitorexit** 这两个**字节码指令**来实现锁的获取和释放。

这里介绍的AbstractQueuedSynchronizer同步器(以下简称AQS)，是基于**FIFO队列**来实现的，**通过state的状态, 来实现acquire和release**。

**state为0，表示当前没有线程执行，可以获取锁，state为1表示已有线程正在执行。**

代码分析：

# 源码分析

AQS是基于FIFO队列来实现的，所以队列必然由**一个个节点**组成，我们先从节点入手。

Node是AQS中的一个内部静态类，主要包含以下几个成员变量：

## AQS中的Node类：

### Node 有3个构造方法：

**Node(Thread thread, int waitStatus)**

**Node(Thread thread, Node mode)**

**Node()**

### Node重要的成员变量：

volatile int **waitStatus**;

volatile Node **prev**;

volatile Node **next**;

static final class Node {  
 */\*\* Marker to indicate a node is waiting in shared mode \*/* static final Node *SHARED* = new Node();  
 */\*\* Marker to indicate a node is waiting in exclusive mode \*/* static final Node *EXCLUSIVE* = null;  
 */\*\* waitStatus value to indicate thread has cancelled \*/* static final int *CANCELLED* = 1;  
 */\*\* waitStatus value to indicate successor's thread needs unparking \*/* static final int *SIGNAL* = -1;  
 */\*\* waitStatus value to indicate thread is waiting on condition \*/* static final int *CONDITION* = -2;  
 */\*\*  
 \* waitStatus value to indicate the next acquireShared should  
 \* unconditionally propagate  
 \*/* static final int *PROPAGATE* = -3;

volatile int waitStatus;

volatile Node prev;

volatile Node next;

*/\*\*  
 \* The thread that enqueued this node. Initialized on  
 \* construction and nulled out after use.  
 \*/* volatile Thread thread;

Node nextWaiter;

*/\*\*  
 \* Returns true if node is waiting in shared mode.  
 \*/*final boolean isShared() {  
 return nextWaiter == *SHARED*;  
}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) { // Used by addWaiter  
 this.nextWaiter = mode;  
 this.thread = thread;  
}  
  
Node(Thread thread, int waitStatus) { // Used by Condition  
 this.waitStatus = waitStatus;  
 this.thread = thread;  
}

}

//waitStatus有以下几个状态

/\*\*

\*\* CANCELLED,值为1，表示当前线程被取消，被终端的Node不回去竞争锁，最终会被踢出队列

\*\* SIGNAL，值为-1，表示当前节点的后继节点被阻塞了，需要被唤醒

\*\* CONDITION，值为-2，表示当前节点在等待condition，因为某个条件被阻塞

\*\* PROPAGATE，值为-3，表示锁的下一次获取可以无条件传播

\*\*/

volatile int waitStatus;

// 前驱节点

volatile Node prev;

// 后继节点

volatile Node next;

// 入队列时的当前线程

volatile Thread thread;

// 表示下一个等待condition的Node

Node nextWaiter;

## AQS重要的成员变量：

// 头节点

private transient volatile **Node head**;

// 尾节点

private transient volatile **Node tail**;

// 同步状态

private volatile int **state**;

## ReentrantLock中的Sync：NonfairSync和FairSync

ReentrantLock中有一个**抽象类Sync**，它继承了AQS，所以ReentrantLock的实现大多就是基于**Sync**来完成实现，**ReentrantLock又分为公平锁和非公平锁之分，**所以，ReentrantLock内部有两个sync的实现：

static final class **NonfairSync** extends Sync{...}

static final class **FairSync** extends Sync{...}

公平锁：获取锁的顺序为调用lock的顺序  
非公平锁：每个线程抢占锁的顺序是不定的

**面试问题：ReentrantLock中如何实现公平锁和非公平锁？**

## lock方法：

ReentrantLock的lock方法：基于sync实现的，两种sync。

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

由于ReentrantLock我们用的比较多的是非公平锁，所以先来看下非公平锁lock的实现

**NonfairSync**的lock方法：

*/\*\*  
 \* Performs lock. Try immediate barge, backing up to normal  
 \* acquire on failure.  
 \*/*final void lock() {  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 acquire(1);  
}

启动了两个线程，假设线程1进来，调用compareAndSetState方法，此时没有任何线程占有锁，所以通过原子操作CAS将state的状态改为1，这个是肯定能成功的，之后将线程1设置到AQS的**exclusiveOwnerThread**个变量中，表示线程1拿走了这个锁。这个时候线程2进入了lock方法，此时通过CAS操作，发现预计值0此时已经变成了1，所以设置失败，进入else分支，调用**acquire方法**。

FairSync的lock方法：直接调用acquire方法。

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

重点来了：

**FairSync**与**NonfairSync的区别：**

**公平锁FairSync直接调用acquire方法，而NonfairSync是先通过CAS抢夺一次，如果成功则直接获取锁，如果抢夺失败，再调用acquire方法。**

**也就是说非公平锁在线程第一次竞争失败后，还是会调用acquire方法，可能进入队列中，而公平锁是直接调用acquire方法。**

## acquire方法的作用呢？

acquire方法是调用父类AQS的acquire方法，代码如下：

*/\*\*  
 \* Acquires in exclusive mode, ignoring interrupts. Implemented  
 \* by invoking at least once {****@link*** *#tryAcquire},  
 \* returning on success. Otherwise the thread is queued, possibly  
 \* repeatedly blocking and unblocking, invoking {****@link*** *\* #tryAcquire} until success. This method can be used  
 \* to implement method {****@link*** *Lock#lock}.  
 \*  
 \** ***@param*** *arg the acquire argument. This value is conveyed to  
 \* {****@link*** *#tryAcquire} but is otherwise uninterpreted and  
 \* can represent anything you like.  
 \*/*public final void acquire(int arg) {  
 if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))   
 *selfInterrupt*();  
}

以上代码可以看出大概的逻辑，**线程1尝试去获取锁，如果失败，则加入到队列**，挂起。

通过一个&&实现了。

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

我们来看下tryAcquire方法，非公平锁最终调用的是nonfairTryAcquire方法，代码如下

final boolean nonfairTryAcquire(int acquires) { // 获取当前线程 final Thread current = Thread.currentThread(); // 获取AQS中锁的标志位state 0 表示锁没有被占有 1 表示锁已经被拿走了 int c = getState(); if (c == 0) { // cas修改state状态 if (compareAndSetState(0, acquires)) { setExclusiveOwnerThread(current); return true; } } else if (current == getExclusiveOwnerThread()) { int nextc = c + acquires; if (nextc < 0) // overflow throw new Error("Maximum lock count exceeded"); setState(nextc); return true; } return false; }

中部了