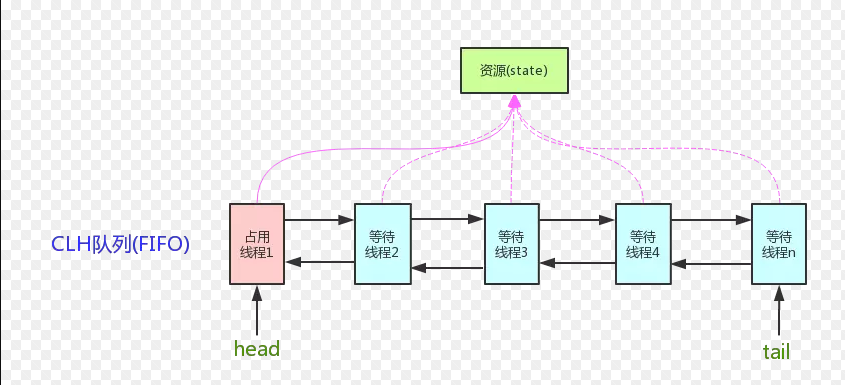
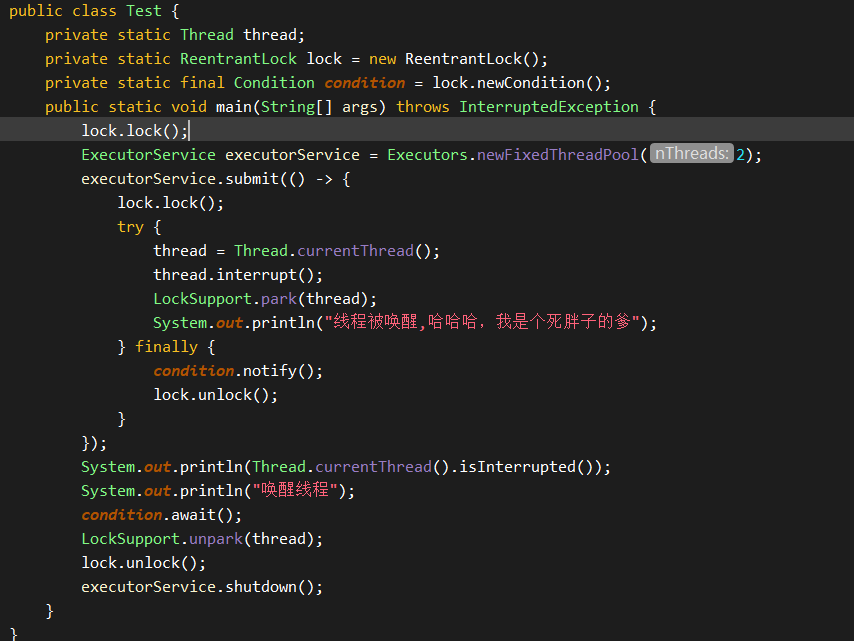
ReentrantLock源码分析

针对ReentrantLock源码，主要有两个概念比较重要，第一个是同步队列（AbstractQueueSynchronizer）简称AQS，另一个就是阻塞队列，两者的本质区别，其一是在同步队列是竞争锁关系，而阻塞队列则为线程状态量（condition）竞争关系。



1.1

以下图1.2作为demo示例：



1.2

* 首先进入到lock.lock()方法去看一下，具体做了哪些操作：

final void lock()

{

if ( compareAndSetState( 0, 1 ) ) /\*看AQS队列state状态，是否有锁资源 \*/

setExclusiveOwnerThread( Thread.currentThread() ); /\* 存在资源同步队列监视器，塞入当前线程 \*/

else

acquire( 1 );

}

详细解读上面的部分

\* compareAndSetState( 0, 1 )，通过CAS操作判断同步队列状态，0：存在锁，>0: 不存在锁。

\* setExclusiveOwnerThread( )：同步队列见识器，设入当前线程

\* acquire( 1 )：

public final void acquire( int arg )

{

if ( !tryAcquire( arg ) &&

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

/\* 中断队列中的线程，并且不可阻塞 \*/

selfInterrupt();

}

\* tryAcquire( arg )：顾名思义这部分应该是尝试去看监视器状态来获取锁

protected boolean tryAcquire( int arg )

{

throw new UnsupportedOperationException();

}

？？？？AQS只是一个抽象的同步队列器！

protected final boolean tryAcquire( int acquires ) /\* 尝试到监视器获取锁资源 \*/

{

return(nonfairTryAcquire( acquires ) );

}  
咱们的重点来了，看如何去尝试获取锁的：

\* nonfairTryAcquire：

final Boolean nonfairTryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

//同步状态为0，个人理解为有锁状态

if (compareAndSetState(0, acquires)) {

//改变同步队列状态

setExclusiveOwnerThread(current);

//监视器监视当前线程

return true;

}

} else if (current == getExclusiveOwnerThread()) {

//如果当前线程状态已经获取到锁

int nextc = c + acquires;

//c=1

if (nextc < 0) // overflow

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

setState(nextc);

//设置同步队列新状态，可重复拿锁

return true;

}

return false;

}

这一部分，做了必要的解释，首先还是看监视器状态，这里说要做一个解释，解释线程是可以重复拿锁的，监视器会记录该线程一次获取锁成功的总次数。

\* addWaiter():没有获取到锁资源的线程会新建一个线程Node，第一次新建Node的时候会构建一个空个队列头。

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); //插入同步队列队尾

return node;

}

源码上写的很清楚，新建一个线程节点，查看队列是否存在尾节点，存在尾结点，在尾结点之后添加当前节点，如果不存在enq

\*enq():通过自旋和CAS避免线程并发的问题，没有尾结点就新建一个空的Node作为头结点，之后将当前节点挂在头结点之后。

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;

}

}

}

}

同步队列形成之后，同步队列的唤醒按照节点顺序，但是如果有节点出现问题，就会导致后续节点没法正常唤醒,线程节点进入到同步队列还需要找到一个有效的前缀节点。

\*acquireQueued(addWaiter(Node.EXCLUSIVE), arg)

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

Boolean failed = true;

try {

Boolean interrupted = false;

for (;;) {

//自旋

//拿到前驱节点

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) &&//找到当前的可用的前驱节点，并且附上节点信号量为signal

parkAndCheckInterrupt())

interrupted = true;

}

}

finally {

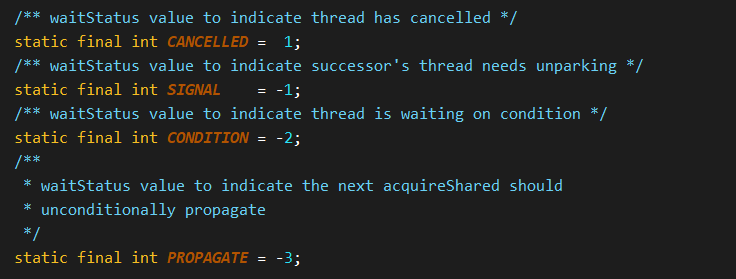
if (failed)

cancelAcquire(node);

}

}

\* shouldParkAfterFailedAcquire:判断前驱节点状态



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

int ws = pred.waitStatus;

if (ws == Node.SIGNAL)

return true;

if (ws > 0) {

do {

node.prev = pred = pred.prev;

}

while (pred.waitStatus > 0);

pred.next = node;

} else {

compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

}

return false;

}

如果当前节点的前缀节点是头结点，当前节点尝试获取锁，如果没有获取到则阻塞当前节点：

\* parkAndCheckInterrupt

private final Boolean parkAndCheckInterrupt() {

LockSupport.park(this); //暂停该节点,this是当前同步队列

return Thread.interrupted();

}

自此，lock方法完结不知道你是不是一头雾水呢，那就多看几遍就ok了！！！

总结：核心每次线程节点都会放在同步队列尾结点，并找到在用的前缀节点

* Next->unLock()

咱们先不说源码,其实大致也知道会发什么什么事情了,首先清空监视器持有的对象,更新监视器状态,唤醒头结点后缀节点的,这个时候还得做一件事顾全大局,重新整体同步节点,去除无效的线程节点.上代码:

\* unlock()

public void unlock() {

sync.release(1);

}

\* release()

public final Boolean release(int arg) {

if (tryRelease(arg)) {

//释放锁资源

Node h = head;

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

unparkSuccessor(h);

return true;

}

return false;

}

\* tryRelease()

protected final Boolean tryRelease(int releases) {

int c = getState() - releases;

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

throw new IllegalMonitorStateException();

Boolean free = false;

if (c == 0) {

free = true;

setExclusiveOwnerThread(null);

}

setState(c);

return free;

}

上面源码都讲的很详细了，主要是对监视器做了一些列操作，next：

Node h = head;  
if (h != null && h.waitStatus != 0)  
 unparkSuccessor(h);

接下来主要就是看unparkSuccessor

\*unparkSuccessor()

private void unparkSuccessor(Node node) {

int ws = node.waitStatus;

if (ws < 0)

compareAndSetWaitStatus(node, ws, 0);

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);

}

检查当前节点的状态情况，如果不是废弃状态，状态更新为废弃状态，取头结点的后缀节点，不是废弃状态，则更新为废弃状态（0），同步队列倒序遍历，重新整理同步队列，去除不再用节点（注意gc），最后唤醒后缀节点，over。

* Condition信号量阻塞队列