AQS

AbstractQueuedSynchronizer 类（简称 AQS）。它是一个**抽象类**，定义一套多线程访问共享资源的同步器框架，是抽象的队列式的同步器。

同步器是基于**模板方法模式**的，同步器的子类可以通过继承同步器并实现它的抽象方法来管理同步状态，子类推荐被定义为自定义同步组件的**静态内部类**。

两个核心

volatile int state

同步的核心其实就一个用volatile修饰的int成员变量，**锁的状态就是这个值的更改。**0就是当前没有线程获取锁，1是有。可重入锁可以多次加锁，即把state值加一，当然也需要同样次数的解锁，因为0才代表当前没有线程获取锁。

```
    private volatile int state;
```

FIFO队列

一个先进先出的双向链表。*这个队列的操作有一点复杂，我建议您可以先跳下去看重要方法后，再回头看Node类。*

```
static final class Node {
```

```
    static final Node SHARED = new Node();//标识等待节点处于共享模式
    static final Node EXCLUSIVE = null;//标识等待节点处于独占模式

    static final int CANCELLED =  1;//由于超时或中断，节点已被取消
    static final int SIGNAL    = -1;//表示下一个节点是通过park堵塞的，需要通过unpark唤
醒
    static final int CONDITION = -2;//表示线程在等待条件变量（先获取锁，加入到条件等待队
列，然后释放锁，等待条件变量满足条件；只有重新获取锁之后才能返回）
    static final int PROPAGATE = -3;//表示后续结点会传播唤醒的操作，共享模式下起作用

    //等待状态：对于condition节点，初始化为CONDITION；其它情况，默认为0，通过CAS操作原子更
新
    volatile int waitStatus;
    //前节点
    volatile Node prev;
    //后节点
    volatile Node next;
    //线程对象
    volatile Thread thread;
    //对于Condtion表示下一个等待条件变量的节点；其它情况下用于区分共享模式和独占模式；
    Node nextWaiter;

    final boolean isShared() {
        return nextWaiter == SHARED;//判断是否共享模式
    }
    //获取前节点，如果为null，抛出异常
    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) {     //addWaiter方法使用
        this.nextWaiter = mode;
        this.thread = thread;
    }
    Node(Thread thread, int waitStatus) { //Condition使用
        this.waitStatus = waitStatus;
        this.thread = thread;
    }
}
```

重要方法

isHeldExclusively() 该线程是否正在独占资源。只有用到 condition 才需要去实现它。
tryAcquire(int)/tryRelease(int)独占方式，尝试获取/释放资源。
tryAcquireShared(int)/tryReleaseShared(int)共享方式，尝试获取/释放资源。

acquire()获取同步状态

不要小看这一行判断，这一句代码其实就是获取许可的核心操作了。

- tryAcquire**尝试获取同步状态**，成功就没必要加入队列。
- 如果获取同步状态失败，把线程构造成结点（Node.EXCLUSIVE，独占式）addWaiter把**结点加入队列尾部。**
- 加入之后acquireQueued()死循环去**轮询前一个结点看是否轮到自己了。**
- 如果轮到自己了，把自己的线程状态设置为**打断等待**

```
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

tryAcquire()尝试获取同步状态

注意：这里的tryAcquire我是把**可重入锁的公平锁对tryAcquire()的实现**贴过来了！因为AQS抽象类并没有实现这个方法，而是留给子类去实现。

```
protected final boolean tryAcquire(int acquires) {
    //获取当前线程
            final Thread current = Thread.currentThread();
            int c = getState();
    //c等于0，说明锁没有被线程占有，可以试图获取锁
            if (c == 0) {
                //如果前面没有线程排队，就用CAS把state从0更新为1
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    //获取到锁了，标记一下，告诉大家，现在是本线程占用了锁
                    setExclusiveOwnerThread(current);
                    //锁获取成功，直接返回
                    return true;
                }
            }
     //c不等于0或者上面CAS操作失败了，说明锁被某线程占有
    //重入就在下面这段代码实现的
    //由于ReentrantLock是可重入，如果获取锁的线程是当前线程，那还是可以再操作一波的
            else if (current == getExclusiveOwnerThread()) {
                //重入了就对state再加1，别忘了acquires是写死为1的
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                //注意：可重入锁加了几次就要释放几次
                setState(nextc);
                return true;
            }
            return false;
        }
    }
```

addWaiter()线程构造成结点尾部入队和enq()自旋入队

用CAS结点加入队列

> 不过当结点被**并发**地被添加到 LinkedList时, LinkedList将难以保证Node的正确添加,最终的结果可能是节点的数量有偏差,而且顺序也是混乱的。
> 所以在enq(final Node node)方法中,同步器通过"死循环"来保证节点的正确添加,在"死循环"中只有通过CAS将节点设置成为尾节点之后,当前线程才能从该方法返回,否则当前线程不断地尝试设置。可以看出,enq(final Node node)方法将并发添加节点的请求通过CAS变得"**串行化**"了。
>
> ——以上出自《java并发编程的艺术》

```
    private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        Node pred = tail;
        if (pred != null) {
             // 将自己的前驱指向尾节点
            node.prev = pred;
            //用CAS把自己加到尾部
            if (compareAndSetTail(pred, node)) {
           // 设置尾结点的后继为自己，双向链表嘛
                pred.next = node;
                //线程成功添加到尾部，可以返回了
                return node;
            }
        }
        //但是添加尾结点的操作在并发的情况下可能失败，于是有了enq方法
        //如果到这里，说明 pred==null队列是空的，或者 CAS把自己探究到尾结点失败(有线程在竞争入队)
        enq(node);
        return node;
    }
```

enq()因为addWaiter中首次添加到队列尾部失败了，自旋加入队列尾部

```
 private Node enq(final Node node) {
     //死循环添加，我就不信加不进去了哼
        for (;;) {
            Node t = tail;
            if (t == null) { // Must initialize
                // 初始化head节点
                if (compareAndSetHead(new Node()))
                    // 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 {
                 //还是加入队列尾部，不过是写在循环里的，加进去了才跳出循环。
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
```

acquireQueued()自旋询问是否到我了

进入一个自旋的过程，不断轮询前面结点的状态，看啥时候到我了。

```
final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            //一直空轮询判断自己是不是
            for (;;) {
                final Node p = node.predecessor();
                //当前节点如果前驱是头结点，就tryAcquire用CAS尝试操作一下state
                if (p == head && tryAcquire(arg)) {
                //获取许可了，就把自己设置为头结点
                    setHead(node);
                    //把前驱的后继指针设置为null，帮助GC回收
                    p.next = null; // help GC
                    //标记设置成功
                    failed = false;
                    //没有被其他线程打断
                    return interrupted;
                }
                // 前驱不是头结点，或者上面的if分支没有成功，tryAcquire(arg)没有抢赢别人
                //
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

shouldParkAfterFailedAcquire抢锁失败，判断是否需要挂起当前线程

在获取同步状态失败后，线程并不是立马进行阻塞，需要检查该线程的状态。该方法主要靠前驱节点判断当前线程是否应该被阻塞。

1. 如果当前线程的前驱节点状态为SINNAL（ws=-1），则表明当前线程需要被阻塞，调用unpark()方法唤醒，直接返回true，当前线程阻塞.
2. 如果当前线程的前驱节点状态为CANCELLED（ws > 0），则表明该线程的前驱节点已经等待超时或者被中断了，则需要从CLH队列中将该前驱节点删除掉，直到回溯到前驱节点状态 <= 0 ，返回false
3. 如果前驱节点非SINNAL，非CANCELLED，则通过CAS的方式将其前驱节点设置为SINNAL，返回false

```
    // 第一个参数是前驱节点，第二个参数是当前线程的节点
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
```

```
            //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
            return true;
        // 前驱节点 waitStatus大于0 ，说明前驱节点取消了排队。往前遍历找一个前驱
        if (ws > 0) {
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            // 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，
        // 直到ws=-1，进入第一个if分支
        return false;
    }
```

如果 shouldParkAfterFailedAcquire(Node pred, Node node) 方法返回true，则调用 parkAndCheckInterrupt()方法阻塞当前线程：

```
    private final boolean parkAndCheckInterrupt() {
        LockSupport.park(this);
        return Thread.interrupted();
    }
```

接下来以ReentrantLock的源码来分析AQS的具体实现。

ReentrantLock

最核心的成员变量final Sync sync，这是一个继承AQS的抽象类，它有两个实现，一个是公平锁一个是非公平锁。

获取当前占用锁的线程，如果State为0表示当前没有线程获取锁返回null，如果有就 getExclusiveOwnerThread获取线程。

```
final Thread getOwner() {
return getState() == 0 ? null : getExclusiveOwnerThread();}
```

公平锁

lock

这里直接**把值写死了**，每次加锁，acquire方法传入1.

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

AQS的acquire

*点进acquire方法发现它跳到AQS的acquire方法里去了，而点进AQS的tryAcquire发现它只抛出一个不支持操作的异常。也就是说在公平锁里，加锁这个操作用的AQS的**acquire**（和非公平锁**共用**），而**tryAcquire**是公平锁和非公平锁**各自**实现的。*

```
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
                acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

tryAcquire

这个方法在上文的AQS中已经分析过了，为了方便顺着看，再贴过来。

```
protected final boolean tryAcquire(int acquires) {
   //获取当前线程
           final Thread current = Thread.currentThread();
           int c = getState();
   //c等于0，说明锁没有被线程占有，可以试图获取锁
           if (c == 0) {
               //如果前面没有线程排队，就用CAS把state从0更新为1
               if (!hasQueuedPredecessors() &&
                   compareAndSetState(0, acquires)) {
                   //获取到锁了，标记一下，告诉大家，现在是本线程占用了锁
                   setExclusiveOwnerThread(current);
                   //锁获取成功，直接返回
                   return true;
               }
           }
    //c不等于0或者上面CAS操作失败了，说明锁被某线程占有
   //重入就在下面这段代码实现的
   //由于ReentrantLock是可重入，如果获取锁的线程是当前线程，那还是可以再操作一波的
           else if (current == getExclusiveOwnerThread()) {
               //重入了就对state再加1，别忘了acquires是写死为1的
               int nextc = c + acquires;
               if (nextc < 0)
                   throw new Error("Maximum lock count exceeded");
               //注意：可重入锁加了几次就要释放几次
               setState(nextc);
               return true;
           }
           return false;
       }
   }
```

非公平锁

非公平锁就两方法lock和tryAcquire

lock

第一次不加队列，直接先CAS试图获取锁。没有成功在走AQS的acquire方法

```
    final void lock() {
            if (compareAndSetState(0, 1))
                setExclusiveOwnerThread(Thread.currentThread());
            else
                //没有成功在走AQS的acquire方法,去排队。
                acquire(1);
        }
```

AQS的acquire

*点进acquire方法发现它跳到AQS的acquire方法里去了，而点进AQS的tryAcquire发现它只抛出一个不支持操作的异常。也就是说在公平锁里，加锁这个操作用的AQS的**acquire**（和非公平锁**共用**），而**tryAcquire**是公平锁和非公平锁**各自**实现的。*

```
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
                acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

tryAcquire

```
    protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
```

tryRelease()

可重入方式的释放锁

```
    protected final boolean tryRelease(int releases) {
    //只是State减1，还未更新
```

```
            int c = getState() - releases;
        //如果获取锁的线程不是当前线程，释放失败
            if (Thread.currentThread() != getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();

            boolean free = false;
        //State等于0，锁被释放了
        //也有可能减1后不等于0，因为可重入锁的State的可以一直加的
            if (c == 0) {
                free = true;
                setExclusiveOwnerThread(null);
            }//更新
            setState(c);
            return free;
        }
```