> *本文由 简悦 SimpRead 转码， 原文地址 juejin.cn*

> *设计 java 的大叔们为了我们方便的自定义各种同步工具，为我们提供了大杀器 AbstractQueuedSynchronizer 类，这是一个抽象类，以下我们会简称 AQS，翻译成中文就是抽象队列同步器。*

标签：「我们都是小青蛙」公众号文章

设计 java 的大叔们为了我们方便的自定义各种同步工具，为我们提供了大杀器`AbstractQueuedSynchronizer`类，这是一个抽象类，以下我们会简称`AQS`，翻译成中文就是抽象队列同步器。这家伙老有用了，封装了各种底层的同步细节，我们程序员想自定义自己的同步工具的时候，只需要定义这个类的子类并覆盖它提供的一些方法就好了。我们前边用到的显式锁`ReentrantLock`就是借助了`AQS`的神力实现的，现在马上来看看这个类的实现原理以及如何使用它自定义同步工具。

同步状态

在`AQS`中维护了一个名叫`state`的字段，是由`volatile`修饰的，它就是所谓的同步状态：

```
private volatile int state;
复制代码
```

并且提供了几个访问这个字段的方法：

方法名	描述
`protected final int getState()`	获取`state`的值
`protected final void setState(int newState)`	设置`state`的值
`protected final boolean compareAndSetState(int expect, int update)`	使用`CAS`方式更新`state`的值

可以看到这几个方法都是`final`修饰的，说明子类中无法重写它们。另外它们都是`protected`修饰的，说明只能在子类中使用这些方法。

在一些线程协调的场景中，一个线程在进行某些操作的时候其他的线程都不能执行该操作，比如持有锁时的操作，在同一时刻只能有一个线程持有锁，我们把这种情景称为独占模式；在另一些线程协调的场景中，可以同时允许多个线程同时进行某种操作，我们把这种情景称为共享模式。

我们可以通过修改`state`字段代表的同步状态来实现多线程的独占模式或者共享模式。

比如在独占模式下，我们可以把state的初始值设置成0，每当某个线程要进行某项独占操作前，都需要判断state的值是不是0，如果不是0的话意味着别的线程已经进入该操作，则本线程需要阻塞等待；如果是0的话就把state的值设置成1，自己进入该操作。这个先判断再设置的过程我们可以通过CAS操作保证原子性，我们把这个过程称为尝试获取同步状态。如果一个线程获取同步状态成功了，那么在另一个线程尝试获取同步状态的时候发现state的值已经是1了就一直阻塞等待，直到获取同步状态成功的线程执行完了需要同步的操作后释放同步状态，也就是把state的值设置为0，并通知后续等待的线程。

在共享模式下的道理也差不多，比如说某项操作我们允许10个线程同时进行，超过这个数量的线程就需要阻塞等待。那么我们就可以把state的初始值设置为10，一个线程尝试获取同步状态的意思就是先判断state的值是否大于0，如果不大于0的话意味着当前已经有 10 个线程在同时执行该操作，本线程需要阻塞等待；如果state的值大于0，那么可以把state的值减1后进入该操作，每当一个线程完成操作的时候需要释放同步状态，也就是把state的值加1，并通知后续等待的线程。

所以对于我们自定义的同步工具来说，需要自定义获取同步状态与释放同步状态的方式，而AQS中的几个方法正是用来做这个事儿的：

方法名	描述
`protected boolean tryAcquire(int arg)`	独占式的获取同步状态，获取成功返回 true，否则 false
`protected boolean tryRelease(int arg)`	独占式的释放同步状态，释放成功返回 true，否则 false
`protected int tryAcquireShared(int arg)`	共享式的获取同步状态，获取成功返回 true，否则 false
`protected boolean tryReleaseShared(int arg)`	共享式的释放同步状态，释放成功返回 true，否则 false
`protected boolean isHeldExclusively()`	在独占模式下，如果当前线程已经获取到同步状态，则返回 true；其他情况则返回 false

我们说AQS是一个抽象类，我们以tryAcquire为例看看它在AQS中的实现：

```
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
复制代码
```

喔😯我的天，竟然只是抛出个异常，这不科学。是的，在AQS中的确没有实现这个方法，不同的同步工具针对的具体并发场景不同，所以如何获取同步状态和如何释放同步状态是需要我们在自定义的AQS子类中实现的，如果我们自定义的同步工具需要在独占模式下工作，那么我们就重写tryAcquire、tryRelease和isHeldExclusively方法，如果是在共享模式下工

作，那么我们就重写tryAcquireShared和tryReleaseShared方法。比如在独占模式下我们可以这样定义一个AQS子类：

```
public class Sync extends AbstractQueuedSynchronizer {

    @Override
    protected boolean tryAcquire(int arg) {
        return compareAndSetState(0, 1);
    }

    @Override
    protected boolean tryRelease(int arg) {
        setState(0);
        return true;
    }

    @Override
    protected boolean isHeldExclusively() {
        return getState() == 1;
    }
}
复制代码
```

tryAcquire表示尝试获取同步状态，我们这里定义了一种极其简单的获取方式，就是使用CAS的方式把state的值设置成1，如果成功则返回true，失败则返回false，tryRelease表示尝试释放同步状态，这里同样采用了一种极其简单的释放算法，直接把state的值设置成0就好了。isHeldExclusively就表示当前是否有线程已经获取到了同步状态。如果你有更复杂的场景，可以使用更复杂的获取和释放算法来重写这些方法。

通过上边的唠叨，我们只是了解了啥是个同步状态，学会了如何通过继承AQS来自定义独占模式和共享模式下获取和释放同步状态的各种方法，但是你会惊讶的发现会了这些仍然没有什么卵用。我们期望的效果是一个线程获取同步状态成功会立即返回true，并继续执行某些需要同步的操作，在操作完成后释放同步状态，如果获取同步状态失败的话会立即返回false，并且进入阻塞等待状态，那线程是怎么进入等待状态的呢？不要走开，下节更精彩。

同步队列

AQS中还维护了一个所谓的同步队列，这个队列的节点类被定义成了一个静态内部类，它的主要字段如下：

```
static final class Node {
    volatile int waitStatus;
    volatile Node prev;
    volatile Node next;
    volatile Thread thread;
    Node nextWaiter;

    static final int CANCELLED =  1;
    static final int SIGNAL    = -1;
    static final int CONDITION = -2;
    static final int PROPAGATE = -3;
}
复制代码
```

AQS中定义一个头节点引用，一个尾节点引用：

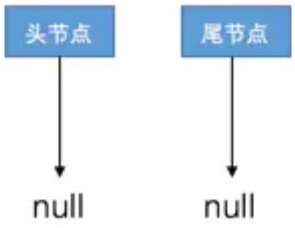
```
private transient volatile Node head;
private transient volatile Node tail;
复制代码
```

通过这两个节点就可以控制到这个队列，也就是说可以在队列上进行诸如插入和移除操作。可以看到Node类中有一个Thread类型的字段，这表明每一个节点都代表一个线程。我们期望的效果是当一个线程获取同步状态失败之后，就把这个线程阻塞并包装成Node节点插入到这个同步队列中，当获取同步状态成功的线程释放同步状态的时候，同时通知在队列中下一个未获取到同步状态的节点，让该节点的线程再次去获取同步状态。

这个节点类的其他字段的意思我们之后遇到会详细唠叨，我们先看一下独占模式和共享模式下在什么情况下会往这个同步队列里添加节点，什么情况下会从它里边移除节点，以及线程阻塞和恢复的实现细节。

独占式同步状态获取与释放

在独占模式下，同一个时刻只能有一个线程获取到同步状态，其他同时去获取同步状态的线程会被包装成一个Node节点放到同步队列中，直到获取到同步状态的线程释放掉同步状态才能继续执行。初始状态的同步队列是一个空队列，里边一个节点也没有，就长这样：



接下来我们就要详细看一下获取同步状态失败的线程是如何被包装成Node节点插入到队列中同时阻塞等待的。

前边说过，获取和释放同步状态的方式是由我们自定义的，在独占模式需要我们定义AQS的子类并且重写下边这些方法：

```
protected boolean tryAcquire(int arg)
protected boolean tryRelease(int arg)
protected boolean isHeldExclusively()
复制代码
```

在定义了这些方法后，谁去调用它们呢？AQS里定义了一些调用它们的方法，这些方法都是由public final修饰的：

方法名	描述
void acquire(int arg)	独占式获取同步状态，如果获取成功则返回，如果失败则将当前线程包装成Node节点插入同步队列中。
void acquireInterruptibly(int arg)	与上个方法意思相同，只不过一个线程在执行本方法过程中被别的线程中断，则抛出InterruptedException异常。
boolean tryAcquireNanos(int arg, long nanos)	在上个方法的基础上加了超时限制，如果在给定时间内没有获取到同步状态，则返回false，否则返回true。
boolean release(int arg)	独占式的释放同步状态。

忽然摆了这么多方法可能有点突兀哈，我们先看一下acquire方法的源代码：

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

代码显示acquire方法实际上是通过tryAcquire方法来获取同步状态的，如果tryAcquire方法返回true则结束，如果返回false则继续执行。这个tryAcquire方法就是我们自己规定的获取同步状态的方式。假设现在有一个线程已经获取到了同步状态，而线程t1同时调用tryAcquire方法尝试获取同步状态，结果就是获取失败，会先执行addWaiter方法，我们一起来看一下这个方法：

```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);  //构造一个新节点
    Node pred = tail;
    if (pred != null) { //尾节点不为空，插入到队列最后
        node.prev = pred;
```

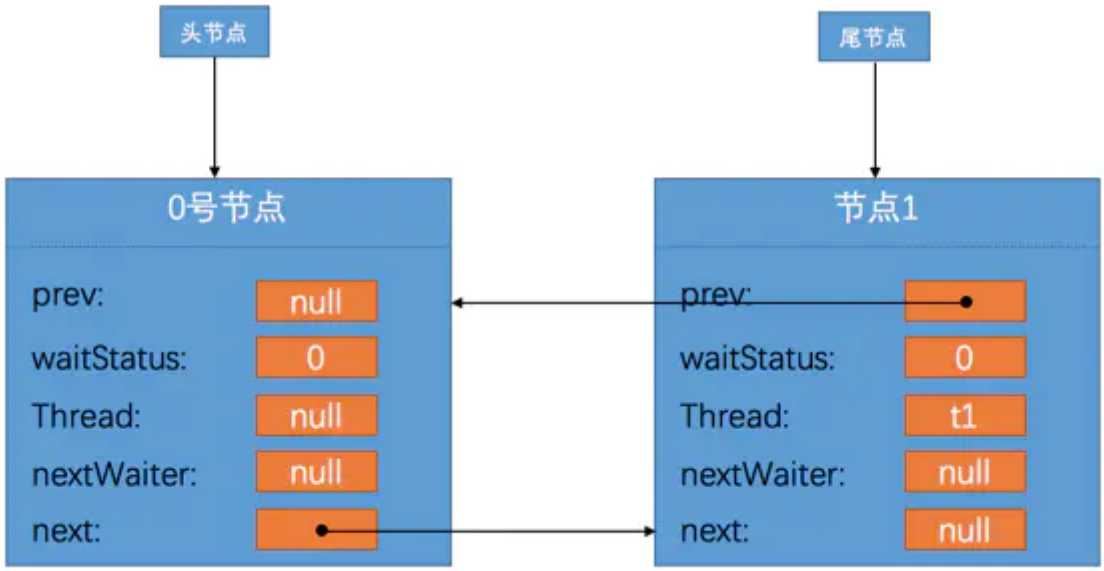
```
            if (compareAndSetTail(pred, node)) {        //更新tail，并且把新节点
插入到列表最后
                pred.next = node;
                return node;
            }
        }
        enq(node);
        return node;
    }

    private Node enq(final Node node) {
        for (;;) {
            Node t = tail;
            if (t == null) {     //tail节点为空，初始化队列
                if (compareAndSetHead(new Node()))   //设置head节点
                    tail = head;
            } else {     //tail节点不为空，开始真正插入节点
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
}
复制代码
```

可以看到，这个addWaiter方法就是向队列中插入节点的方法。首先会构造一个Node节点，假设这个节点为节点1，它的thread字段就是当前线程t2，这个节点被刚刚创建出来的样子就是这样：

然后我们再分析一下具体的插入过程。如果tail节点不为空，直接把新节点插入到队列后边就返回了，如果tail节点为空，调用enq方法先初始化一下head和tail节点之后再把新节点插入到队列后边。enq方法的这几行初始化队列的代码需要特别注意：

```
if (t == null) {     //tail节点为空，初始化队列
    if (compareAndSetHead(new Node()))   //设置head节点
        tail = head;
} else {
    //真正插入节点的过程
}
复制代码
```

也就是说在队列为空的时候会先让head和tail引用指向同一个节点后再进行插入操作，而这个节点竟然就是简简单单的new Node()，真是没有任何添加剂呀～ 我们先把这个节点称为0号节点吧，这个节点的任何一个字段都没有被赋值，所以在第一次节点插入后，队列其实长这样：



其中的节点1才是我们真正插入的节点，代表获取同步状态失败的线程，0号节点是在初始化过程中创建的，我们之后再看它有什么用。

addWaiter方法调用完会返回新插入的那个节点，也就是节点1，acquire方法会接着调用acquireQueued方法：

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();  //获取前一个节点
            if (p == head && tryAcquire(arg)) { 前一个节点是头节点再次尝试获
取同步状态
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
```

```
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```
复制代码

可以看到，如果新插入的节点的前一个节点是头节点的话，会再次调用tryAcquire尝试获取同步状态，这个主要是怕获取同步状态的线程很快就把同步状态给释放了，所以在当前线程阻塞之前抱着侥幸的心理再试试能不能成功获取到同步状态，如果侥幸可以获取，那就调用setHead方法把头节点换成自己：

```
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```
复制代码

同时把本Node节点的thread字段设置为null，意味着自己成为了0号节点。

如果当前Node节点不是头节点或者已经获取到同步状态的线程并没有释放同步状态，那就乖乖的往下执行shouldParkAfterFailedAcquire方法：

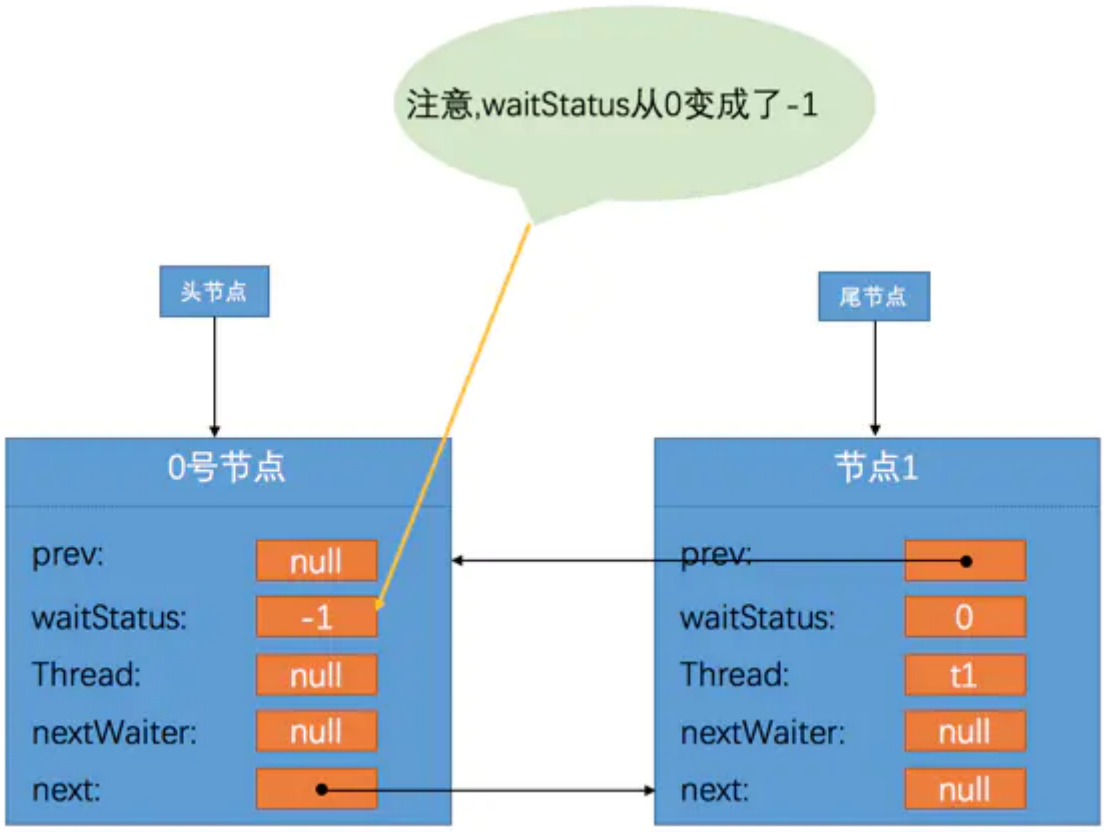
```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node
node) {
    int ws = pred.waitStatus;    //前一个节点的状态
    if (ws == Node.SIGNAL)  //Node.SIGNAL的值是-1
        return true;
    if (ws > 0) {    //当前线程已被取消操作，把处于取消状态的节点都移除掉
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {    //设置前一个节点的状态为-1
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```
复制代码

这个方法是对Node节点中的waitStatus的各种操作。如果当前节点的前一个节点的waitStatus是Node.SIGNAL，也就是 - 1，那么意味着当前节点可以被阻塞，如果前一个节点的waitStatus大于0，意味着该节点代表的线程已经被取消操作了，需要把所有waitStatus大于0的节点都移除掉，如果前一个节点的waitStatus既不是-1，也不大于0，

就把如果前一个节点的waitStatus设置成Node.SIGNAL。我们知道Node类里定义了一些代表waitStatus的静态变量，我们来看看waitStatus的各个值都是什么意思吧：

静态变量	值	描述
Node.CANCELLED	1	节点对应的线程已经被取消了（我们后边详细会说线程如何被取消）
Node.SIGNAL	-1	表示后边的节点对应的线程处于等待状态
Node.CONDITION	-2	表示节点在等待队列中（稍后会详细说什么是等待队列）
Node.PROPAGATE	-3	表示下一次共享式同步状态获取将被无条件的传播下去（稍后再说共享式同步状态的获取与释放时详细唠叨）
无	0	初始状态

现在我们重点关注waitStauts为0或者-1的情况。目前我们的当前节点是节点1，它对应着当前线程，当前节点的前一个节点是0号节点。在一开始，所有的Node节点的waitStatus都是0，所以在第一次调用shouldParkAfterFailedAcquire方法时，当前节点的前一个节点，也就是0号节点的waitStatus会被设置成Node.SIGNAL立即返回false，这个状态的意思就是说0号节点后边的节点都处于等待状态，现在的队列已经变成了这个样子：



由于acquireQueued方法是一个循环，在第二次执行到shouldParkAfterFailedAcquire方法时，由于0号节点的waitStatus已经为Node.SIGNAL了，所以shouldParkAfterFailedAcquire方法会返回true，然后继续执行parkAndCheckInterrupt方法：

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

LockSupport.park(this)方法：

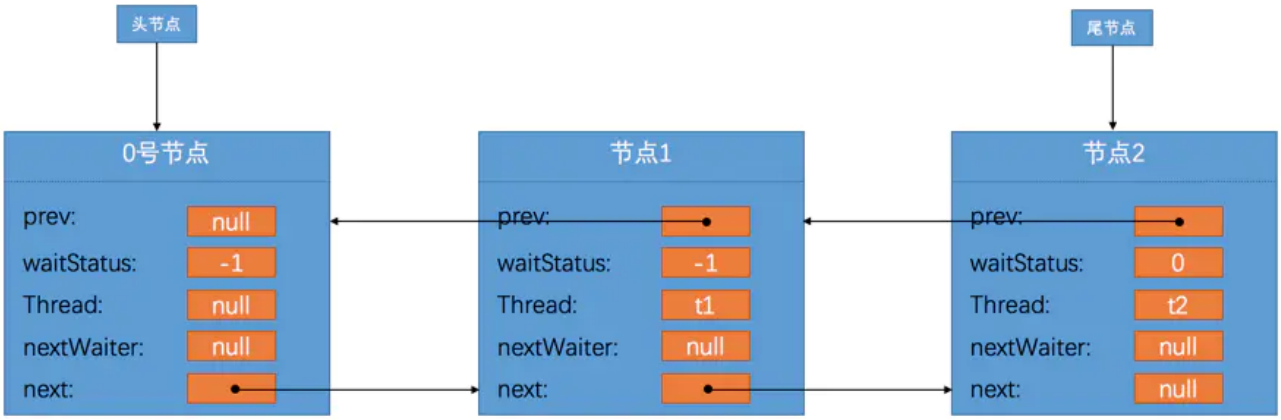
```
public static void park(Object blocker) {
    Thread t = Thread.currentThread();
    setBlocker(t, blocker);
    UNSAFE.park(false, 0L);     //调用底层方法阻塞线程
    setBlocker(t, null);
}
复制代码
```

其中的UNSAFE.park(false, 0L)方法如下：

```
public native void park(boolean var1, long var2);
复制代码
```

这个就表示立即阻塞线程，这是一个底层方法，我们程序员就不用关心操作系统时如何阻塞线程的了。呼～至此，我们在独占模式下跑完了一个获取不到同步状态的线程是怎么被插入到同步队列以及被阻塞的过程。这个过程需要大家多看几遍，毕竟比较麻烦哈～

如果此时再新来一个线程t2调用acquire方法要求获取同步状态的话，它同样会被包装成Node插入同步队列的，效果就像下图一样：



大家注意一下节点1的waitStauts已经变成-1了，别忘了waitStauts值为-1的时候，也就是Node.SIGNAL意味着它的下一个节点处于等待状态，因为0号节点和节点1的waitStauts值都为-1，也就意味着它们两个的后继节点，也就是节点1和节点2都处于等待状态。

以上就是线程t1和t2在某个线程已经获取了同步状态的情况下调用acquire方法时所产生的后果，acquireInterruptibly和acquire方法基本一致，只不过它是可中断的，也就是说在一个线程调用acquireInterruptibly由于没有获取到同步状态而发生阻塞之后，如果有别的线程中断了这个线程，则acquireInterruptibly方法会抛出InterruptedException异常并返回。tryAcquireNanos也是支持中断的，只不过还带有一个超时时间，如果超出了该时间tryAcquireNanos还没有返回，则返回false。

如果一个线程在各种acquire方法中获取同步状态失败的话，会被包装成Node节点放到同步队列，这个可以看作是一个插入过程。有进就有出，如果一个线程完成了独占操作，就需要释放同步状态，同时把同步队列第一个(非0号节点)节点代表的线程叫醒，在我们上边的例子中就是节点1，让它继续执行，这个释放同步状态的过程就需要调用release方法了:

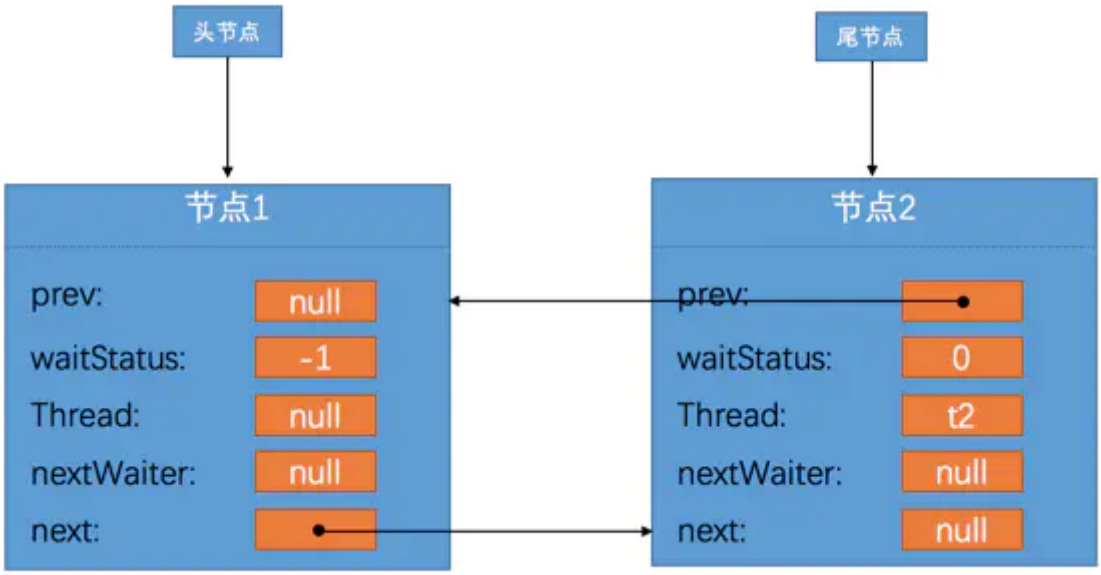
```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
复制代码
```

可以看到这个方法会用到我们在AQS子类里重写的tryRelease方法，如果成功的释放了同步状态，那么就继续往下执行，如果头节点head不为null并且head的waitStatus不为0，就执行unparkSuccessor方法:

```
private void unparkSuccessor(Node node) {
        int ws = node.waitStatus;   //节点的等待状态
        if (ws < 0)
            compareAndSetWaitStatus(node, ws, 0);
        Node s = node.next;
        if (s == null || s.waitStatus > 0) {    //如果node为最后一个节点或者
node的后继节点被取消了
            s = null;
            for (Node t = tail; t != null && t != node; t = t.prev)
                if (t.waitStatus <= 0)  //找到离头节点最近的waitStatus为负数
的节点
                    s = t;
        }
        if (s != null)
            LockSupport.unpark(s.thread);    //唤醒该节点对应的线程
    }
```

复制代码

我们现在的头节点head指向的是0号节点，它的状态为-1，所以它的waitStatus首先会被设置成0，接着它的后继节点，也就是节点1代表的线程会被这样调用LockSupport.unpark(s.thread)，这个方法的意思就是唤醒节点1对应的线程t2，把节点1的thread设置为null并把它设置为头节点，修改后的队列就长下边这样：



所以现在等待队列里只有一个t2线程是阻塞的。这就是释放同步状态的过程。

独占式同步工具举例

看完了独占式同步状态获取与释放的原理，我们可以尝试自定义一个简单的独占式同步工具，我们常用的锁就是一个独占式同步工具，我们下边来定义一个简单的锁：

```
import java.util.concurrent.locks.AbstractQueuedSynchronizer;

public class PlainLock {

    private static class Sync extends AbstractQueuedSynchronizer {

        @Override
        protected boolean tryAcquire(int arg) {
            return compareAndSetState(0, 1);
        }

        @Override
        protected boolean tryRelease(int arg) {
            setState(0);
            return true;
        }

        @Override
```

```
        protected boolean isHeldExclusively() {
            return getState() == 1;
        }
    }

    private Sync sync = new Sync();


    public void lock() {
        sync.acquire(1);
    }

    public void unlock() {
        sync.release(1);
    }
}
复制代码
```

我们在PlainLock中定义了一个AQS子类Sync，重写了一些方法来自定义了在独占模式下获取和释放同步状态的方式，静态内部类就是AQS子类在我们自定义同步工具中最常见的定义方式。然后在PlainLock里定义了lock方法代表加锁，unlock方法代码解锁，具体的方法调用我们上边都快说吐了，这里就不想让你再吐一次了，看一下这个锁的应用：

```
public class Increment {

    private int i;

    private PlainLock lock = new PlainLock();

    public void increase() {
        lock.lock();
        i++;
        lock.unlock();
    }

    public int getI() {
        return i;
    }

    public static void test(int threadNum, int loopTimes) {
        Increment increment = new Increment();

        Thread[] threads = new Thread[threadNum];

        for (int i = 0; i < threads.length; i++) {
            Thread t = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < loopTimes; i++) {
```

```
                        increment.increase();
                    }
                }
            });
            threads[i] = t;
            t.start();
        }

        for (Thread t : threads) {  //main线程等待其他线程都执行完成
            try {
                t.join();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        System.out.println(threadNum + "个线程，循环" + loopTimes + "次结果：" + increment.getI());
    }

    public static void main(String[] args) {
        test(20, 1);
        test(20, 10);
        test(20, 100);
        test(20, 1000);
        test(20, 10000);
        test(20, 100000);
        test(20, 1000000);
    }
}
复制代码
```

执行结果：

```
20个线程，循环1次结果：20
20个线程，循环10次结果：200
20个线程，循环100次结果：2000
20个线程，循环1000次结果：20000
20个线程，循环10000次结果：200000
20个线程，循环100000次结果：2000000
20个线程，循环1000000次结果：20000000
复制代码
```

很显然这个我们只写了几行代码的锁已经起了作用，这就是AQS的强大之处，我们只需要写很少的东西就可以构建一个同步工具，而且不用考虑底层复杂的同步状态管理、线程的排队、等待与唤醒等等机制。

共享式同步状态获取与释放

共享式获取与独占式获取的最大不同就是在同一时刻是否有多个线程可以同时获取到同步状态。获取不到同步状态的线程也需要被包装成Node节点后阻塞的，而可以访问同步队列的方法就是下边这些：

|void acquireShared(int arg)| 共享式获取同步状态，如果失败则将当前线程包装成Node节点插入同步队列中。。||void acquireSharedInterruptibly(int arg)| 与上个方法意思相同，只不过一个线程在执行本方法过程中被别的线程中断，则抛出InterruptedException异常。||boolean tryAcquireSharedNanos(int arg, long nanos)| 在上个方法的基础上加了超时限制，如果在给定时间内没有获取到同步状态，则返回false，否则返回true。||boolean releaseShared(int arg)| 共享式的释放同步状态。|

哈，和独占模式下的方法长得非常像嘛，只是每个方法中都加了一个Shared单词。它们的功能也是一样一样的，以acquireShared方法为例：

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
复制代码
```

这个方法会调用我们自定义的AQS子类中的tryAcquireShared方法去获取同步状态，只不过tryAcquireShared的返回值是一个int值，该值不小于 0 的时候表示获取同步状态成功，则acquireShared方法直接返回，什么都不做；如果该返回值大于 0 的时候，表示获取同步状态失败，则会把该线程包装成Node节点插入同步队列，插入过程和独占模式下的过程差不多，我们这就不多废话了。

另外两个acquire方法也不多废话了，只不过一个是可中断的，一个是支持超时的～

释放同步状态的方法也和独占模式的差不多：

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
复制代码
```

这个方法会调用我们自定义的AQS子类中的tryReleaseShared方法去释放同步状态，如果释放成功的话会移除同步队列中的一个阻塞节点。与独占模式不同的一点是，可能同时会有多个线程释释放同步状态，也就是可能多个线程会同时移除同步队列中的阻塞节点，哈哈，如何保证移除过程的安全性？这个问题就不看源码了，大家自己尝试着写写。

共享式同步工具举例

假设某个操作只能同时有两个线程操作，其他的线程需要处于等待状态，我们可以这么定义这个锁：

```
import java.util.concurrent.locks.AbstractQueuedSynchronizer;

public class DoubleLock {


    private static class Sync extends AbstractQueuedSynchronizer {

        public Sync() {
            super();
            setState(2);    //设置同步状态的值
        }

        @Override
        protected int tryAcquireShared(int arg) {
            while (true) {
                int cur = getState();
                int next = getState() - arg;
                if (compareAndSetState(cur, next)) {
                    return next;
                }
            }
        }

        @Override
        protected boolean tryReleaseShared(int arg) {
            while (true) {
                int cur = getState();
                int next = cur + arg;
                if (compareAndSetState(cur, next)) {
                    return true;
                }
            }
        }
    }

    private Sync sync = new Sync();

    public void lock() {
        sync.acquireShared(1);
    }

    public void unlock() {
        sync.releaseShared(1);
    }
```

```
}
复制代码
```

state的初始值释2，每当一个线程调用tryAcquireShared获取到同步状态时，state的值都会减1，当state的值为0时，其他线程就无法获取到同步状态从而被包装成Node节点进入同步队列等待。

AQS中其他针对同步队列的重要方法

除了一系列acquire和release方法，AQS还提供了许多直接访问这个队列的方法，它们由都是public final修饰的：

方法名	描述
boolean hasQueuedThreads()	是否有正在等待获取同步状态的线程。
boolean hasContended()	是否某个线程曾经因为获取不到同步状态而阻塞
Thread getFirstQueuedThread()	返回队列中第一个（等待时间最长的）线程，如果目前没有将任何线程加入队列，则返回 null
boolean isQueued(Thread thread)	如果给定线程的当前已加入同步队列，则返回 true。
int getQueueLength()	返回等待获取同步状态的线程数估计值，因为在构造该结果时，多线程环境下实际线程集合可能发生大的变化
Collection<Thread> getQueuedThreads()	返回包含可能正在等待获取的线程 collection，因为在构造该结果时，多线程环境下实际线程集合可能发生大的变化

如果有需要的话，可以在我们自定义的同步工具中使用它们。

题外话

写文章挺累的，有时候你觉得阅读挺流畅的，那其实是背后无数次修改的结果。如果你觉得不错请帮忙转发一下，万分感谢～ 这里是我的公众号「我们都是小青蛙」，里边有更多技术干货，时不时扯一下犊子，欢迎关注：

小册

另外，作者还写了一本 MySQL 小册：《MySQL 是怎样运行的：从根儿上理解 MySQL》的链接。小册的内容主要是从小白的角度出发，用比较通俗的语言讲解关于 MySQL 进阶的一些核心概念，比如记录、索引、页面、表空间、查询优化、事务和锁等，总共的字数大约是三四十万字，配有上百幅原创插图。主要是想降低普通程序员学习 MySQL 进阶的难度，让学习曲线更平滑一点～ 有在 MySQL 进阶方面有疑惑的同学可以看一下：

小孩子4919

邀您一起读小册

掘金小册

MySQL 是怎样运行的：从根儿上理解 MySQL

小孩子

前·跟谁学后端工程师

你将得到

12 小时 · 26 小节

从根儿上理解 MySQL，让 MySQL 不再是一个黑盒

- MySQL 入门：认识 MySQL 的本质及入门基础知识讲解
- 基础原理：理解记录、页面、索引、表空间是什么
- 查询优化：理解 MySQL 是如何进行优化查询
- 事务与锁：理解事务的起源以及实现原理

名人推荐

周彦伟

Oracle MySQL ACE Director

《MySQL运维内参》作者

罗斌

跟谁学用户增长VP

前百度凤巢高级技术经理

老钱

掌阅服务端技术专家

梁桂钊

公众号「服务端思维」作者

原价 ~~29.9~~ 元 / 26 小节


优惠购买25.92元