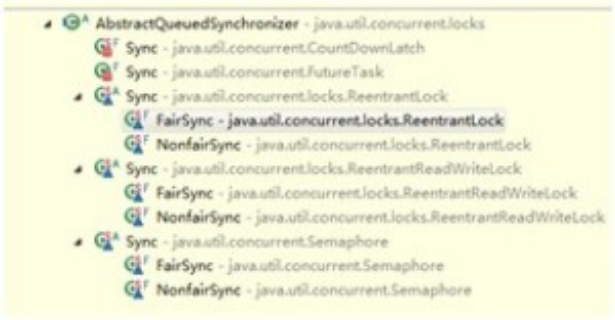
AQS的原理浅析

- AQS的原理浅析

本文是《Java特种兵》的样章，本书即将由**工业出版社**出版

AQS的全称为（AbstractQueuedSynchronizer），这个类也是在java.util.concurrent.locks下面。这个类似乎很不容易看懂，因为它仅仅是提供了一系列公共的方法，让子类来调用。那么要理解意思，就得从子类下手，反过来看才容易看懂。如下图所示：

图 5-15 AQS的子类实现



这么多类，我们看那一个？刚刚提到过锁（Lock），我们就从锁开始吧。这里就先以ReentrantLock排它锁为例开始展开讲解如何利用AQS的，然后再简单介绍读写锁的要点（读写锁本身的实现十分复杂，要完全说清楚需要大量的篇幅来说明）。

首先来看看ReentrantLock的构造方法，它的构造方法有两个，如下图所示：

图 5-16 排它锁的构造方法

```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = (fair)? new FairSync() : new NonfairSync();
}
```

很显然，对象中有一个属性叫sync，有两种不同的实现类，默认是“NonfairSync”来实现，而另一个“FairSync”它们都是排它锁的内部类，不论用那一个都能实现排它锁，只是内部可能有点原理上的区别。先以“NonfairSync”类为例，它的lock()方法是如何实现的呢？

图 5-17 排它锁的lock方法

```
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
protected final void setExclusiveOwnerThread(Thread t) {
    exclusiveOwnerThread = t;
}
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

lock()方法先通过CAS尝试将状态从0修改为1。若直接修改成功，前提条件自然是锁的状态为0，则直接将线程的OWNER修改为当前线程，这是一种理想情况，如果并发粒度设置适当也是一种乐观情况。

若上一个动作未成功，则会间接调用了acquire(1)来继续操作，这个acquire(int)方法就是在AbstractQueuedSynchronizer当中了。这个方法表面上看起来简单，但真实情况比较难以看懂，因为第一次看这段代码可能不知道它要做什么！不急，一步一步来分解。

首先看tryAcquire(arg)这里的调用（当然传入的参数是1），在默认的“NonfairSync”实现类中，会这样来实现：

妈呀，这代码好费劲，胖哥第一回看也是觉得这样，细心看看也不是想象当中那么难：

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        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;
}
```

○ 首先获取这个锁的状态，如果状态为0，则尝试设置状态为传入的参数（这里就是1），若设置成功就代表自己获取到了锁，返回true了。状态为0设置1的动作在外部就有做过一次，内部再一次做只是提升概率，而且这样的操作相对锁来讲不占开销。

○ 如果状态不是0，则判定当前线程是否为排它锁的Owner，如果是Owner则尝试将状态增加acquires（也就是增加1），如果这个状态值越界，则会抛出异常提示，若没有越界，将状态设置进去后返回true（实现了类似于偏向的功能，可重入，但是无需进一步征用）。

○ 如果状态不是0，且自身不是owner，则返回false。

回到图 5-17中对tryAcquire()的调用判定中是通过if(!tryAcquire())作为第1个条件的，如果返回true，则判定就不会成立了，自然后面的acquireQueued动作就不会再执行了，如果发生这样的情况是最理想的。

无论多么乐观，征用是必然存在的，如果征用存在则owner自然不会是自己，tryAcquire()方法会返回false，接着就会再调用方法：acquireQueued(addWaiter(Node.EXCLUSIVE), arg)做相关的操作。

这个方法的调用的代码更不好懂，需要从里往外看，这里的Node.EXCLUSIVE是节点的类型，看名称应该清楚是排它类型的意思。接着调用addWaiter()来增加一个排它锁类型的节点，这个addWaiter()的代码是这样写的：

```
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;
}
```

图 5-19 addWaiter的代码

这里创建了一个Node的对象，将当前线程和传入的Node.EXCLUSIVE传入，也就是说Node节点理论上包含了这两项信息。代码中的tail是AQS的一个属性，刚开始的时候肯定是为null，也就是不会进入第一层if判定的区域，而直接会进入enq(node)的代码，那么直接来看看enq(node)的代码。

看到了tail就应该猜到了AQS是链表吧，没错，而且它还应该有一个head引用来指向链表的头节点，AQS在初始化的时候head、tail都是null，在运行时来回移动。此时，我们最少至少知道AQS是一个基于状态（state）的链表管理方式。

图 5-20 enq(Node)的源码

```
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            Node h = new Node(); // Dummy header
            h.next = node;
            node.prev = h;
            if (compareAndSetHead(h)) {
                tail = node;
                return h;
            }
        }
        else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

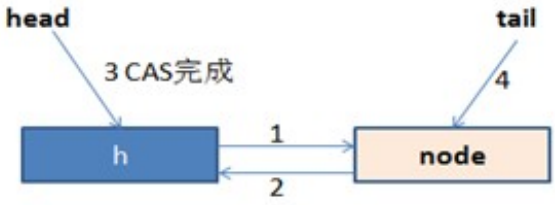
这段代码就是链表的操作，某些同学可能很牛，一下就看懂了，某些同学一扫而过觉得知道大概就可以了，某些同学可能会莫不着头脑。胖哥为了给第三类同学来“开开荤”，简单讲解下这个代码。

首先这个是一个死循环，而且本身没有锁，因此可以有多个线程进来，假如某个线程进入方法，此时head、tail都是null，自然会进入if(t == null)所在的代码区域，这部分代码会创建一个Node出来名字叫h，这个Node没有像开始那样给予类型和线程，很明显是一个空的Node对象，而传入的Node对象首先被它的next引用所指向，此时传入的node和某一个线程创建的h对象如下图所示。

图 5-21 临时的h对象创建后的与传入的Node指向关系

刚才我们很理想的认为只有一个线程会出现这种情况，如果有多个线程并发进入这个if判定区域，可能就会同时存在多个这样的数据结构，在各自形成数据结构后，多个线程都会去做compareAndSetHead(h)的动作，也就是尝试将这个临时h节点设置为head，显然并发时只有一个线程会成功，因此成功的那个线程会执行tail = node的操作，整个AQS的链表就成为：

图 5-22 AQS被第一个请求成功的线程初始化后

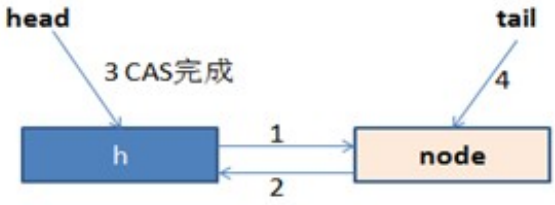


有一个线程会成功修改head和tail的值，其它的线程会继续循环，再次循环就不会进入if (t == null)的逻辑了，而会进入else语句的逻辑中。

在else语句所在的逻辑中，第一步是node.prev = t，这个t就是tail的临时值，也就是首先让尝试写入的node节点的prev指针指向原来的结束节点，然后尝试通过CAS替换掉AQS中的tail的内容为当前线程的Node，无论有多少个线程并发到这里，依然只会有一个能成功，成功者执行t.next = node，也就是让原先的tail节点的next引用指向现在的node，现在的node已经成为了最新的结束节点，不成功者则会继续循环。

简单使用图解的方式来说明，3个步骤如下所示，如下图所示：

图 5-23 插入一个节点步骤前后动作



插入多个节点的时候，就以此类推了哦，总之节点都是在链表尾部写入的，而且是线程安全的。

知道了AQS大致的写入是一种双向链表的插入操作，但插入链表节点对锁有何用途呢，我们还得退

回到前面图 5-19的代码中addWaiter方法最终返回了要写入的node节点， 再回退到图5-17中所在的代码中需要将这个返回的node节点作为acquireQueued方法入口参数，并传入另一个参数（依然是1），看看它里面到底做了些什么？请看下图：

图 5-24 acquireQueued的方法内容

```
final boolean acquireQueued(final Node node, int arg) {
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } catch (RuntimeException ex) {
        cancelAcquire(node);
        throw ex;
    }
}
```

这里也是一个死循环，除非进入if(p == head && tryAcquire(arg))这个判定条件，而p为node.predcessor()得到，这个方法返回node节点的前一个节点，也就是说只有当前一个节点是head的时候，进一步尝试通过tryAcquire(arg)来征用才有机会成功。tryAcquire(arg)这个方法我们前面介绍过，成立的条件为：锁的状态为0，且通过CAS尝试设置状态成功或线程的持有者本身是当前线程才会返回true，我们现在来详细拆分这部分代码。

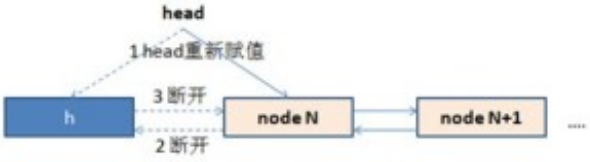
○ 如果这个条件成功后，发生的几个动作包含：

（1） 首先调用setHead(Node)的操作，这个操作内部会将传入的node节点作为AQS的head所指向的节点。线程属性设置为空（因为现在已经获取到锁，不再需要记录下这个节点所对应的线程了），再将这个节点的perv引用赋值为null。

（2） 进一步将的前一个节点的next引用赋值为null。

在进行了这样的修改后，队列的结构就变成了以下这种情况了，通过这样的方式，就可以让执行完的节点释放掉内存区域，而不是无限制增长队列，也就真正形成FIFO了：

图 5-25 CAS成功获取锁后，队列的变化



○ 如果这个判定条件失败

会首先判定：“shouldParkAfterFailedAcquire(p , node)”，这个方法内部会判定前一个节点的状态是否为：“Node.SIGNAL”，若是则返回true，若不是都会返回false，不过会再做一些操作：判定节点的状态是否大于0，若大于0则认为被“CANCELLED”掉了（我们没有说明几个状态的值，不过大于0的只可能被CANCELLED的状态），因此会从前一个节点开始逐步循环找到一个没有被“CANCELLED”节点，然后与这个节点的next、prev的引用相互指向；如果前一个节点的状态不是大于0的，则通过CAS尝试将状态修改为“Node.SIGNAL”，自然的如果下一轮循环的时候会返回值应该会返回true。

如果这个方法返回了true，则会执行：“parkAndCheckInterrupt()”方法，它是通过LockSupport.park(this)将当前线程挂起到WATING状态，它需要等待一个中断、unpark方法来唤醒它，通过这样一种FIFO的机制的等待，来实现了Lock的操作。

相应的，可以自己看看FairSync实现类的lock方法，其实区别不大，有些细节上的区别可能会决定某些特定场景的需求，你也可以自己按照这样的思路去实现一个自定义的锁。

接下来简单看看unlock()解除锁的方式，如果获取到了锁不释放，那自然就成了死锁，所以必须要释放，来看看它内部是如何释放的。同样从排它锁（ReentrantLock）中的unlock()方法开始，请先看下面的代码截图：

图 5-26 unlock方法间接调用AQS的release(1)来完成

```
public void unlock() {
    sync.release(1);
}
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

通过tryRelease(int)方法进行了某种判定，若它成立则会将head传入到unparkSuccessor(Node)方法中并返回true，否则返回false。首先来看看tryRelease(int)方法，如下图所示：

图 5-27 tryRelease(1)方法

```
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;
}
```

这个动作可以认为就是一个设置锁状态的操作，而且是将状态减掉传入的参数值（参数是1），如果结果状态为0，就将排它锁的Owner设置为null，以使得其它的线程有机会进行执行。

在排它锁中，加锁的时候状态会增加1（当然可以自己修改这个值），在解锁的时候减掉1，同一个锁，在可以重入后，可能会被叠加为2、3、4这些值，只有unlock()的次数与lock()的次数对应才会将Owner线程设置为空，而且也只有这种情况下才会返回true。

这一点大家写代码要注意了哦，如果是在循环体中lock()或故意使用两次以上的lock(),而最终只有一次unlock()，最终可能无法释放锁。在本书的src/chapter05/locks/目录下有相应的代码，大家可以自行测试的哦。

在方法unparkSuccessor(Node)中，就意味着真正要释放锁了，它传入的是head节点（head节点是已经执行完的节点，在后面阐述这个方法的body的时候都叫head节点），内部首先会发生的动作是获取head节点的next节点，如果获取到的节点不为空，则直接通过：“LockSupport.unpark()”方法来释放对应的被挂起的线程，这样一来将会有一个节点唤醒后继续进入图 5-24中的循环进一步尝试tryAcquire()方法来获取锁，但是也未必能完全获取到哦，因为此时也可能有一些外部的请求正好与之征用，而且还奇迹般的成功了，那这个线程的运气就有点悲剧了，不过通常乐观认为不会每一次都那么悲剧。

再看看共享锁，从前面的排它锁可以看得出来是用一个状态来标志锁的，而共享锁也不例外，但是Java不希望去定义两个状态，所以它与排它锁的第一个区别就是在锁的状态上，它用int来标志锁的状态，int有4个字节，它用高16位标志读锁（共享锁），低16位标志写锁（排它锁），高16位每次增加1相当于增加65536（通过1 << 16得到），自然的在这种读写锁中，读锁和写锁的个数都不能超过65535个（条件是每次增加1的，如果递增是跳跃的将会更少）。在计算读锁数量的时候将状态左移16位，而计算排它锁会与65535“按位求与”操作，如下图所示。

图 5-28 读写锁中的数量计算及限制

```
static final int SHARED_SHIFT   = 16;
static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;
static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;

/** Returns the number of shared holds represented in count  */
static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
/** Returns the number of exclusive holds represented in count  */
static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
```

写锁的功能与“ReentrantLock”基本一致，区域在于它会在tryAcquire操作的时候，判定状态的时候会更加复杂一些（因此有些时候它的性能未必好）。

读锁也会写入队列，Node的类型被改为：“Node.SHARED”这种类型，lock()时候调用的是AQS的acquireShared(int)方法，进一步调用tryAcquireShared()操作里面只需要检测是否有排它锁，如果没有则可以尝试通过CAS修改锁的状态，如果没有修改成功，则会自旋这个动作（可能会有很多线程在这自旋开销CPU）。如果这个自旋的过程中检测到排它锁竞争成功，那么tryAcquireShared()会返回-1，从而会走如排它锁的Node类似的流程，可能也会被park住，等待排它锁相应的线程最终调用unpark()动作来唤醒。

这就是Java提供的这种读写锁，不过这并不是共享锁的诠释，在共享锁里面也有多种机制 ，或许这种读写锁只是其中一种而已。在这种锁下面，读和写的操作本身是互斥的，但是读可以多个一起发生。这样的锁理论上是非常适合应用在“读多写少”的环境下（当然我们所讲的读多写少是读的比例远远大于写，而不是多一点点），理论上讲这样锁征用的粒度会大大降低，同时系统的瓶颈会减少，效率得到总体提升。

在本节中我们除了学习到AQS的内在，还应看到Java通过一个AQS队列解决了许多问题，这个是Java层面的队列模型，其实我们也可以利用许多队列模型来解决自己的问题，甚至于可以改写模型模型来满足自己的需求，在本章的5.6.1节中将会详细介绍。

关于Lock及AQS的一些补充：

1、 Lock的操作不仅仅局限于lock()/unlock()，因为这样线程可能进入WAITING状态，这个时候如果没有unpark()就没法唤醒它，可能会一直“睡”下去，可以尝试用tryLock()、tryLock(long , TimeUnit)来做一些尝试加锁或超时来满足某些特定场景的需要。例如有些时候发现尝试加锁无法加上，先释放已经成功对其它对象添加的锁，过一小会再来尝试，这样在某些场合下可以避免“死锁”哦。

2、 lockInterruptibly() 它允许抛出InterruptException异常，也就是当外部发起了中断操作，程序内部有可能会抛出这种异常，但是并不是绝对会抛出异常的，大家仔细看看代码便清楚了。

3、 newCondition()操作，是返回一个Condition的对象，Condition只是一个接口，它要求实现await()、awaitUninterruptibly()、awaitNanos(long)、await(long , TimeUnit)、awaitUntil(Date)、signal()、signalAll()方法，AbstractQueuedSynchronizer中有一个内部类叫做ConditionObject实现了这个接口，它也是一个类似于队列的实现，具体可以参考源码。大多数情况下可以直接使用，当然觉得自己比较牛逼的话也可以参考源码自己来实现。

4、 在AQS的Node中有每个Node自己的状态（waitStatus），我们这里归纳一下，分别包含:

SIGNAL 从前面的代码状态转换可以看得出是前面有线程在运行，需要前面线程结束后，调用

unpark()方法才能激活自己，值为：-1

CANCELLED 当AQS发起取消或fullyRelease()时，会是这个状态。值为1，也是几个状态中唯一一个大于0的状态，所以前面判定状态大于0就基本等价于是CANCELLED的意思。

CONDITION 线程基于Condition对象发生了等待，进入了相应的队列，自然也需要Condition对象来激活，值为-2。

PROPAGATE 读写锁中，当读锁最开始没有获取到操作权限，得到后会发起一个doReleaseShared()动作，内部也是一个循环，当判定后续的节点状态为0时，尝试通过CAS自旋方式将状态修改为这个状态，表示节点可以运行。

状态0 初始化状态，也代表正在尝试去获取临界资源的线程所对应的Node的状态。