深入浅出Java并发包—锁机制(二) - 人生设计师 - 博客园

深入浅出Java并发包—锁机制(二)

接上文《深入浅出Java并发包—锁机制(一)》

2、Sync.FairSync.TryAcquire（公平锁）

我们直接来看代码

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

和明细我们可以看出，公平锁就比不公平锁多了一个判断头结点的方法，就是采用此方法来保证锁的公平性。

3、AbstractQueuedSynchronizer.addWaiter

tryAcquire**失败就意味着入队列了。此时**AQS**的队列中节点**Node**就开始发挥作用了。一般情况下**AQS**支持独占锁和共享锁，而独占锁在**Node**中就意味着条件（**Condition**）队列为空。在**java.util.concurrent.locks.AbstractQueuedSynchronizer.Node**中有两个常量**

```
static final Node EXCLUSIVE = null; //独占节点模式

static final Node SHARED = new Node(); //共享节点模式
```

addWaiter(mode)**中的**mode**就是节点模式，也就是共享锁还是独占锁模式。添加的节点是当前线程。(注：**ReentrantLock**是独占锁模式)，我们来看下对应的实现代码：**

```
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;
  }
```

这块代码并不复杂，如果当前队尾存在元素（tail!=null），则通过CAS添加当前线程到队尾，如果队尾为空或者CAS失败，则通过enq方法设置tail。我们来看下enq的代码

```
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;
        }
      }
    }
  }
```

该方法就是循环调用CAS，即使有高并发的场景，无限循环将会最终成功把当前线程追加到队尾（或设置队头）。总而言之，addWaiter的目的就是通过CAS把当前现在追加到队尾，并返回包装后的Node实例。

把线程要包装为Node对象的主要原因，除了用Node构造供虚拟队列外，还用Node包装了各种线程状态，这些状态被精心设计为一些数字值：

1)、 SIGNAL(-1)：线程的后继线程正/已被阻塞，当该线程release或cancel时要重新这个后继线程(unpark)

2)、 CANCELLED(1)：因为超时或中断，该线程已经被取消

3)、 CONDITION(-2)：表明该线程被处于条件队列，就是因为调用了Condition.await而被阻塞

4)、 PROPAGATE(-3)：传播共享锁

5)、 0：0代表无状态

3、AbstractQueuedSynchronizer.acquireQueued(进行阻塞)

acquireQueued的主要作用是把已经追加到队列的线程节点（addWaiter方法返回值）进行阻塞，但阻塞前又通过tryAccquire重试是否能获得锁，如果重试成功能则无需阻塞，直接返回。下面我们来看以下它对应的源码信息

```
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;
        }
    }
```

仔细看看这个方法是个无限循环，感觉如果p == head && tryAcquire(arg)条件不满足循环将永远无法结束，当然不会出现死循环，奥秘在于parkAndCheckInterrupt会把当前线程挂起，从而阻塞住线程的调用栈。我们来看下他的实现方法：

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

如前面所述，LockSupport.park最终把线程交给系统（Linux）内核进行阻塞。当然也不是马上把请求不到锁的线程进行阻塞，还要检查该线程的状态，比如如果该线程处于Cancel状态则没有必要，具体的检查在shouldParkAfterFailedAcquire中：

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
            /*
             * This node has already set status asking a release
             * to signal it, so it can safely park
             */
            return true;
        if (ws > 0) {
            /*
             * Predecessor was cancelled. Skip over predecessors and
             * indicate retry.
             */
        do {
       node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
        } else {
            /*
             * waitStatus must be 0 or PROPAGATE. Indicate that we
             * need a signal, but don't park yet. Caller will need to
             * retry to make sure it cannot acquire before parking.
             */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

检查原则如下：

1、如果前继节点的waitStatus为signal，则说明前面的节点都还灭有获取到锁，此时当前线程需要阻塞，直接返回true

2、如果前继节点waitStatus>0，说明前继节点已经被取消，则重新设置当前节点的前继节点，返回false，之后无限循环直到第一步状态返回true，导致线程阻塞

3、如果前继节点waitStatus小于0而且不等于-1(signal)，则通过CAS设置前继节点额外isignal，并返回false，之后无限循环直到步骤1返回true，线程阻塞。

请求锁不成功的线程会被挂起在acquireQueued方法的第12行，12行以后的代码必须等线程被解锁锁才能执行，假如被阻塞的线程得到解锁，则执行第13行，即设置interrupted = true，之后又进入无限循环。

从无限循环的代码可以看出，并不是得到解锁的线程一定能获得锁，必须在第6行中调用tryAccquire重新竞争，非公平锁中有可能被新加入的线程获取到，从而导致刚刚被唤醒的线程再次阻塞；公平锁通过判断当前节点是否是头结点来保证锁的公平性。上面的代码我们还可以看到，因为每次第一个被解锁的是头结点，因此一般p==head的判断都会成功。解锁相对比较简单，主要体现在AbstractQueuedSynchronizer.release和Sync.tryRelease方法中：

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

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;
    }
```

这个逻辑也比较简单：

1.判断持有锁的线程是否是当前线程，如果不是就抛出IllegalMonitorStateExeception()，因为一个线程是不能释放另一个线程持有的锁（否则锁就失去了意义）。否则进行2。

2.将AQS状态位减少要释放的次数（对于独占锁而言总是1），如果剩余的状态位0（也就是没有线程持有锁），那么当前线程就是最后一个持有锁的线程，清空AQS持有锁的独占线程。进行3。

3.将剩余的状态位写回AQS，如果没有线程持有锁就返回true，否则就是false。

从上面我们可以知道，这里c==0决定了是否完全释放了锁。由于ReentrantLock是可重入锁，因此同一个线程可能多重持有锁，那么当且仅当最后一个持有锁的线程释放锁是才能将AQS中持有锁的独占线程清空，这样接下来的操作才需要唤醒下一个需要锁的AQS节点（Node），否则就只是减少锁持有的计数器，并不能改变其他操作。

当tryRelease操作成功后（也就是完全释放了锁），release操作才能检查是否需要唤醒下一个继任节点。这里的前提是AQS队列的头结点需要锁(waitStatus!=0)，如果头结点需要锁，就开始检测下一个继任节点是否需要锁操作。

上文说道acquireQueued操作完成后（拿到了锁），会将当前持有锁的节点设为头结点，所以一旦头结点释放锁，那么就需要寻找头结点的下一个需要锁的继任节点，并唤醒它。我们来看下对应的实现代码：

```
private void unparkSuccessor(Node node) {
        /*
         * If status is negative (i.e., possibly needing signal) try
         * to clear in anticipation of signalling. It is OK if this
         * fails or if status is changed by waiting thread.
         */
        int ws = node.waitStatus;
        if (ws < 0)
            compareAndSetWaitStatus(node, ws, 0);

        /*
         * Thread to unpark is held in successor, which is normally
         * just the next node.  But if cancelled or apparently null,
         * traverse backwards from tail to find the actual
         * non-cancelled successor.
         */
        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);
    }
```

对比对应的代码我们可以看出，一旦头结点的后继结点被唤醒，那么后继结点就尝试去获取锁，如果获取成功就将头结点设置为自身，并将头结点的前任节点清空。

```
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;
        }
    }
```

对比lock，unlock是相当比较简单的，主要是释放需要响应的资源，并唤醒AQS队列中有效的后继结点，这样就试图以请求的顺序获取锁资源了。

对比公平锁和不公平锁，其实就是在获取锁的时候有区别，释放锁的时候都是一样的。非公平锁总

是尝试看当前有没有线程持有锁，如果没有则使用现有的线程去抢占锁资源，但是一旦抢占失败，也就和公平锁一样，进入阻塞队列老老实实排队去了，也就是说公平锁和非公平锁只有在进入AQS的CLH队列之前有区别，后面都是按照队列的顺序请求锁资源的。

怀有希望!!

posted @ 2016-02-24 17:44 人生设计师 阅读(2943) 评论() 编辑 收藏