根据ReentrantLock分析AQS在java中的应用

AQS

AQS 全程AbstractQueuedSynchronizer,类如其名，抽象队列同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch，AQS就是一个半成品框架，方便程序员实现锁

结合ReentrantLock源码来分析AQS以及ReentrantLock

ReentrantLock提供了2种构造器实例化对象
非公平锁机制

```
1 public ReentrantLock() {
2     sync = new NonfairSync();
3 }
```

公平锁机制

```
1 public ReentrantLock(boolean fair) {
2 sync = fair ? new FairSync() : new NonfairSync();
3 }
```

公平锁和非公平锁的概念

非公平锁实现源码分析

```
1 static final class NonfairSync extends Sync {
2     //.......
3     /**
4      * Performs lock.  Try immediate barge, backing up to normal
5      * acquire on failure.
6      */
7     final void lock() {
8         if (compareAndSetState(0, 1))
9             setExclusiveOwnerThread(Thread.currentThread());
10        else
11            acquire(1);
12    }
13    protected final boolean tryAcquire(int acquires) {
14        return nonfairTryAcquire(acquires);
15    }
16 }
```

NonfairSync是ReentrantLock的一个内部类继承自Sync，我们在来看Sync的代码

```
1 abstract static class Sync extends AbstractQueuedSynchronizer {
2     private static final long serialVersionUID = -5179523762034025860L;
3     /**
4      * Performs {@link Lock#lock}. The main reason for subclassing
```

```
     * is to allow fast path for nonfair version.
     */
    abstract void lock();
    /**
     * Performs non-fair tryLock.  tryAcquire is implemented in
     * subclasses, but both need nonfair try for trylock method.
     */
    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;
    }

    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;
    }

    protected final boolean isHeldExclusively() {
        // While we must in general read state before owner,
        // we don't need to do so to check if current thread is owner
        return getExclusiveOwnerThread() == Thread.currentThread();
    }

    final ConditionObject newCondition() {
        return new ConditionObject();
    }

    // Methods relayed from outer class

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

    final int getHoldCount() {
```

```
61          return isHeldExclusively() ? getState() : 0;
62      }
63
64      final boolean isLocked() {
65          return getState() != 0;
66      }
67
68      /**
69       * Reconstitutes the instance from a stream (that is, deserializes it).
70       */
71      private void readObject(java.io.ObjectInputStream s)
72          throws java.io.IOException, ClassNotFoundException {
73          s.defaultReadObject();
74          setState(0); // reset to unlocked state
75      }
76  }
```

Sync是ReentrantLock的一个抽象内部类，继承自AbstractQueuedSynchronizer，到此类与类之间的关系基本清楚，我们分析加锁和释放锁的过程
ReentrantLock中的lock方法如下所示

```
1 public void lock() {
2     sync.lock();
3 }
```

Sync类中的lock是一个抽象方法，我们看他实现类NonfairSync中的方法，如下所示

```
1 final void lock() {
2     if (compareAndSetState(0, 1))
3         setExclusiveOwnerThread(Thread.currentThread());
4     else
5         acquire(1);
6 }
```

先来看看上边的方法compareAndSetState(0, 1)，该方法是通过CAS算法，把锁状态设置为1
我们接着看acquire(1);方法
在java.util.concurrent.locks.AbstractQueuedSynchronizer类中
代码段1

```
1 public final void acquire(int arg) {
2     if (!tryAcquire(arg) &&
3         acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
4         selfInterrupt();
5 }
```

先看看第一个条件判断tryAcquire(arg)
在java.util.concurrent.locks.AbstractQueuedSynchronizer类中

```
1 protected boolean tryAcquire(int arg) {
2     throw new UnsupportedOperationException();
3 }
```

发现只是抛出了异常没有其他逻辑代码，这时候我们可以想想NonfairSync继承Sync,Sync继承自AbstractQueuedSynchronizer，NonfairSync类中提供了tryAcquire(int arg)方法，
相当于对AbstractQueuedSynchronizer类中的方法进行了重写，那我们只需要查看NonfairSync类中的tryAcquire(int arg)方法，代码如下所示

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```

nonfairTryAcquire(acquires)方法调用了父类Sync中的方法代码如下所示

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //不判断是否有等待队列,直接进行占用,如果占用失败也进到等待队列尾
        //这个也是公平锁和非公平锁的区别所在
        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;
}
```

终于到核心代码了，我们一行一行来看
final Thread current = Thread.currentThread();
获取当前的线程，目的是实现重入锁
int c = getState();
获取AbstractQueuedSynchronizer中的当前线程的状态
接着看if....if else条件判断
如果当前线程的状态为0，表示该线程第一次获取到锁，compareAndSetState(0, acquires)
通过CAS把锁的状态改变为1，返回true，获取锁成功
else if 表示当前线程之前以及获取过锁，而且没有释放锁，此中情况下把锁的状态值加1，再次设置给state，返回true，表示没有释放锁的线程获取锁成功
如果以上情况都不成立返回false，表示获取锁失败

我们接着回到代码段1
!tryAcquire(arg) 经过上边的分析如果tryAcquire(arg)返会true表示获取锁成功
如果返回false表示获取锁失败，!tryAcquire(arg)之后变为了true，我么看获取不到锁之后的处理逻辑
执行了acquireQueued(addWaiter(Node.EXCLUSIVE), arg)代码
先来看addWaiter(Node.EXCLUSIVE)代码

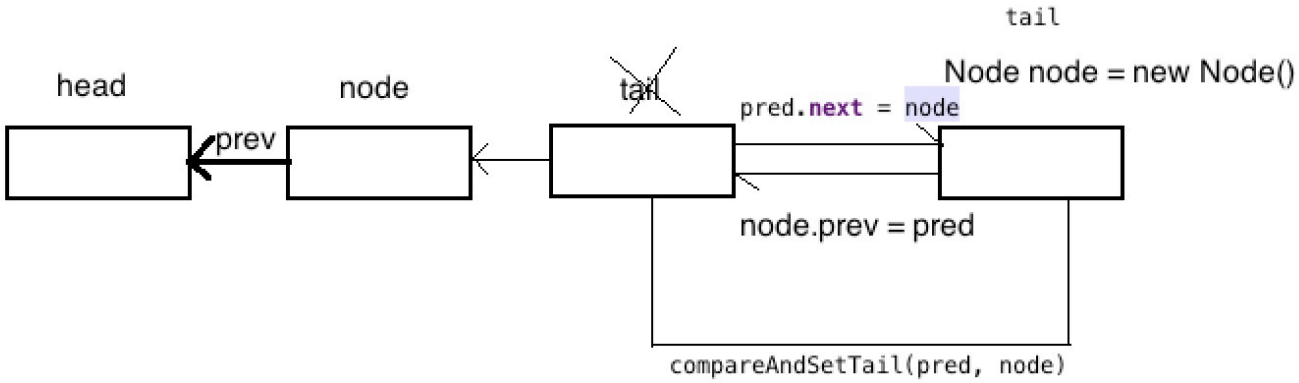
```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
```

```
11      }
12      enq(node);
13      return node;
14  }
```

这段代码的主要作用是把没有抢到锁的线程放入到一个队列中，我么来分析一下这个过程
Node node = new Node(Thread.currentThread(), mode);
说明当前的一个节点就代表一个线程，mode默认是独享模式
Node pred = tail;
定义一个node节点把尾节点先保存起来

```
1  if (pred != null) {
2     node.prev = pred;
3     if (compareAndSetTail(pred, node)) {
4         pred.next = node;
5         return node;
6     }
7  }
```

如果尾节点不为空，我们来分析一下插入数据的过程



如果尾节点为空，说明还没有初始化该队列，执行enq()方法，我们来看过程

```
1  private Node enq(final Node node) {
2      for (;;) {
3          Node t = tail;
4          if (t == null) { // Must initialize
5              if (compareAndSetHead(new Node()))
6                  tail = head;
7          } else {
8              node.prev = t;
9              if (compareAndSetTail(t, node)) {
10                 t.next = node;
11                 return t;
12             }
13         }
14     }
15 }
```

有了上边的分析看这个应该比较简单，在这里就不在做多余的描述
到此获取锁的整个过程完结，我们接下来分析释放所得过程
java.util.concurrent.locks.ReentrantLock中释放锁的代码如下

```
1  public void unlock() {
2      sync.release(1);
3  }
```

在java.util.concurrent.locks.AbstractQueuedSynchronizer中找到了release(1)方法

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

java.util.concurrent.locks.ReentrantLock.Sync中重写了tryRelease(arg)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;
}
```

这段代码没有什么其他的主要是设置线程的状态
unparkSuccessor(h);//作用是唤醒后续的节点，来看看具体是怎么唤醒的

```
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.
     */
    //当前节点线程的状态,如果小于 0 ，设置为0
    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.
     */
    //当前节点的下一个节点
    Node s = node.next;
    //后继节点为null或者其状态 > 0 (超时或者被中断了)
    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);
}
```

重要看这段代码

```
for (Node t = tail; t != null && t != node; t = t.prev)
    if (t.waitStatus <= 0)
        s = t;
```

这段代码是采用了回溯法获取到需要唤醒的线程节点

> 回溯法
> 是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为"回溯点"

接着看怎么唤醒线程
LockSupport.unpark(s.thread);
java.util.concurrent.locks.LockSupport类中的方法如下

```
public static void unpark(Thread thread) {
    if (thread != null)
        UNSAFE.unpark(thread);
}
```

```
public native void unpark(Object var1);
```

看到了native方法
我们是有去jvm虚拟机的代码中查看
查看网友找到的代码如下所示

```
void Parker::unpark() {
//定义两个变量，staus用于判断是否获取锁
  int s, status ;
  //获取锁
  status = os::Solaris::mutex_lock (_mutex) ;
  //判断是否成功
  assert (status == 0, "invariant") ;
  //存储原先变量_counter
  s = _counter;
  //把_counter设为1
  _counter = 1;
  //释放锁
  status = os::Solaris::mutex_unlock (_mutex) ;
  assert (status == 0, "invariant") ;
  if (s < 1) {
  //如果原先_counter信号量小于1，即为0，则进行signal操作，唤醒操作
    status = os::Solaris::cond_signal (_cond) ;
    assert (status == 0, "invariant") ;
  }
}
```

到此使用AQS实现非公平锁的加锁，解锁过程完毕

公平锁的实现和非公平锁基本相同，只是判断不判断有等待队列存在不相同
调用加锁方法之后非公平锁的实现代码如下所示

```
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;
}
```

公平锁的实现如下所示

```
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;
}
```

我们看到多了!hasQueuedPredecessors()判断代码

重入锁的本质原理是使用CAS算法改变锁的状态

没有完成的部分

1. 获取锁失败，做中断，超时判断没有分析

请读者结合源码自己分析