一: ReentrantLock公平锁原理

这里前面介绍了sychronized这个关键字，这里除了这个关键字外，其实还有另外一个ReentrantLock方法，这个相对于前者来说，其实功能更加丰富，而且特点之一就是可以实现公平性。其实这里的公平性，本质就是在竞争的时候，如果非公平锁则会在lock的时候，先去竞争锁，如果竞争失败了则放进到同步队列当中。而公平锁则是在lock的时候，就直接放进同步队列里面，不去竞争，按照进来的先后顺序去获取锁了。

```
    package ReentrantLock;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class FairAndUnfairTest {
       /*CountDownLatch一个同步辅助类，在jdk5中引入，
        * 它允许一个或多个线程等待其他线程操作完成之后才执行。
    能够使一个或多个线程等待其他线程完成各自的工作后再执行*/
    private static CountDownLatch start;

    private static class MyReentrantLock extends ReentrantLock {
        public MyReentrantLock(boolean fair) {
            super(fair); //是否开启公平锁
        }

        public Collection<Thread> getQueuedThreads() {
            List<Thread> arrayList = new ArrayList<Thread>(super.getQueuedThreads());
            Collections.reverse(arrayList);
            return arrayList;
        }
    }

    private static class Worker extends Thread {
        private Lock lock;

        public Worker(Lock lock) { this.lock = lock;}

        @Override
        public void run() {
            try {
                start.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 连续两次打印当前的Thread和等待队列中的Thread
            //for (int i = 0; i < 2; i++) {
                lock.lock(); //加锁
                try {
                    System.out.println("Lock by [" + getName() + "], Waiting by "
                     + ((MyReentrantLock) lock).getQueuedThreads());
                } finally {
                    System.out.println(getName()+" is unlock ");
                    lock.unlock();
                    //slowly();
                }

            //}
        }

        public String toString() {
            return getName();
        }
    }

    public static void main(String[] args) {
        Lock fairLock = new MyReentrantLock(true);
        Lock unfairLock = new MyReentrantLock(false);

        //testLock(fairLock);   //开启公平性
        testLock(unfairLock);  //
    }

    private static void testLock(Lock lock) {
        start = new CountDownLatch(1); //计数器加1
        for (int i = 0; i < 10; i++) { //启动五个线程
```

```
            Thread thread = new Worker(lock);
            thread.setName("" + i);
            thread.start();
        }
        start.countDown();
    }

    private static void slowly() {
        try {
            TimeUnit.MICROSECONDS.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

这里的结果如下所示:

```
Lock by [7], Waiting by [2, 1, 4, 0, 3, 5]
7 is unlock
Lock by [2], Waiting by [1, 4, 0, 3, 5, 6, 9, 8]
2 is unlock
Lock by [1], Waiting by [4, 0, 3, 5, 6, 9, 8]
1 is unlock
Lock by [4], Waiting by [0, 3, 5, 6, 9, 8]
4 is unlock
Lock by [0], Waiting by [3, 5, 6, 9, 8]
0 is unlock
Lock by [3], Waiting by [5, 6, 9, 8]
3 is unlock
Lock by [5], Waiting by [6, 9, 8]
5 is unlock
Lock by [6], Waiting by [9, 8]
6 is unlock
Lock by [9], Waiting by [8]
9 is unlock
Lock by [8], Waiting by []
8 is unlock
```

下面测试一下公平锁的情况，注意这里测试公平锁就是放开上面的 testLock(fairLock); 这一句的代码，然后注释公平性锁的代码，执行情况如下所示:

```
  Lock by [1], Waiting by [2, 0, 4, 3, 5, 6]
  1 is unlock
  Lock by [2], Waiting by [0, 4, 3, 5, 6, 7, 9, 8]
  2 is unlock
  Lock by [0], Waiting by [4, 3, 5, 6, 7, 9, 8]
  0 is unlock
  Lock by [4], Waiting by [3, 5, 6, 7, 9, 8]
  4 is unlock
  Lock by [3], Waiting by [5, 6, 7, 9, 8]
  3 is unlock
  Lock by [5], Waiting by [6, 7, 9, 8]
  5 is unlock
  Lock by [6], Waiting by [7, 9, 8]
  6 is unlock
  Lock by [7], Waiting by [9, 8]
  7 is unlock
  Lock by [9], Waiting by [8]
  9 is unlock
  Lock by [8], Waiting by []
  8 is unlock
```

那么这里先简单了解一下关于公平锁与非公平锁的区别，为了查看两者区别，这里先确定看哪里的源代码入手。因为这里MyReentrantLock继承ReentrantLock，那么因为使用了super(fair)这个代码，因此这里可以从super入手，因为这里的值就会关联到ReentrantLock的构造方法。

```
  /**
 * Creates an instance of {@code ReentrantLock} with the
 * given fairness policy.
 *
 * @param fair {@code true} if this lock should use a fair ordering policy
 */
public ReentrantLock(boolean fair) {
    //这里根据传入的公平是否来调用不同的代码，也就是公平锁与非公平锁的内容.
    sync = fair ? new FairSync() : new NonfairSync();
}

/**
 * Acquires the lock.
 *
```

```
 * <p>Acquires the lock if it is not held by another thread and returns
 * immediately, setting the lock hold count to one.
 *
 * <p>If the current thread already holds the lock then the hold
 * count is incremented by one and the method returns immediately.
 *
 * <p>If the lock is held by another thread then the
 * current thread becomes disabled for thread scheduling
 * purposes and lies dormant until the lock has been acquired,
 * at which time the lock hold count is set to one.
 */
public void lock() {
    sync.lock();
}
```

通过super这个方法可以找到上面的内容，这里可以发现传入的boolean值或进入到不同的方法里面，那么这里就先看一下关于NonfairSync的内容。

```
  /**
     * Sync object for non-fair locks
     */
    static final class NonfairSync extends Sync {
        private static final long serialVersionUID = 7316153563782823691L;

        /**
         * Performs lock.  Try immediate barge, backing up to normal
         * acquire on failure.
         */
        final void lock() {
            //这里是先尝试加锁修改结构体的内容，
            if (compareAndSetState(0, 1))
                //如果成功了则通过setExclusiveOwnerThread记录当前线程
                setExclusiveOwnerThread(Thread.currentThread());
            else
                //这里不成功则调用该方法，把线程放入到同步队列里面
                acquire(1);
        }

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

这里可以看到，使用lock方法加锁的时候，这里通过代码其实可以明显看到，非公平锁在开始的时候会竞争锁的，然后这里关于acquire(1)这个方法后面再看一下，接着先来看看如果是公平锁的时候，上面的new FairSync()的内容，这里的源码见如下所示:

```
  /**
 * Sync object for fair locks
 */
static final class FairSync extends Sync {
    private static final long serialVersionUID = -300897897090466540L;

    final void lock() {
        //公平锁直接调用这个方法，不去竞争
        acquire(1);
    }

    /**
     * Fair version of tryAcquire.  Don't grant access unless
     * recursive call or no waiters or is first.
     */
    //这个方法是acquire里面判断第一个需要执行的方法，这里后面会看到代码，同时与非公平锁的对比
    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;
```

```
    }
}
```

那么这里可以看到不管公平锁还是非公平锁，这里都会调用到这个acquire(1)方法，那么这个方法到底是处理什么的呢？这里再具体看下代码。

```
 /**
 * Acquires in exclusive mode, ignoring interrupts.  Implemented
 * by invoking at least once {@link #tryAcquire},
 * returning on success.  Otherwise the thread is queued, possibly
 * repeatedly blocking and unblocking, invoking {@link
 * #tryAcquire} until success.  This method can be used
 * to implement method {@link Lock#lock}.
 *
 * @param arg the acquire argument.  This value is conveyed to
 *        {@link #tryAcquire} but is otherwise uninterpreted and
 *        can represent anything you like.
 */
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

首先这里都可以看到acquire的第一个调用方法是tryAcquire,那么这里又是什么呢？根据上面的代码可以知道
上面非公平锁的tryAcquire是调用nonfairTryAcquire,那么下面就看下具体的代码了。

```
  /**
     * 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;
    }
```

然后这里就可以对比一下发现，其实tryAcquire这个方法，第一步不管是公平与否，都是先执行getState这个方法的，然后查看这个方法的说明是

```
Returns the current value of synchronization state. This operation has memory semantics of a volatile read.
```

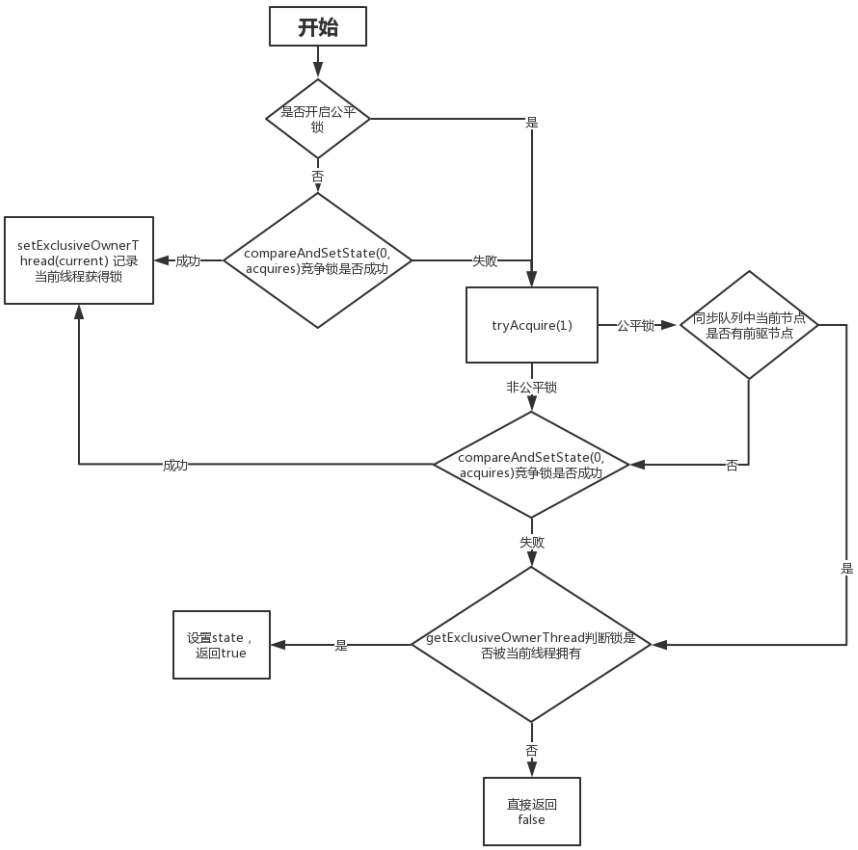
通过这里的说明就可以知道这个方法其实是就是获取当前的锁的state,也就是说如果当前锁没有被获取到的话，那么这里就会进入到c==0的方法里面，那么这里也是再一次的竞争了，但是可以看到公平锁是多了一句!hasQueuedPredecessors()，这句如果进入方法里面看，如下所示：

```
    *  <pre> {@code
 * protected boolean tryAcquire(int arg) {
 *   if (isHeldExclusively()) {
 *     // A reentrant acquire; increment hold count
 *     return true;
 *   } else if (hasQueuedPredecessors()) {
 *     return false;
 *   } else {
 *     // try to acquire normally
 *   }
 * }}</pre>
 *
 * @return {@code true} if there is a queued thread preceding the
 *         current thread, and {@code false} if the current thread
 *         is at the head of the queue or the queue is empty
 * @since 1.7
 */
public final boolean hasQueuedPredecessors() {
    // The correctness of this depends on head being initialized
    // before tail and on head.next being accurate if the current
    // thread is first in queue.
    Node t = tail; // Read fields in reverse initialization order
```

```
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

通过这里的说明其实就可以知道，如果当前的同步队列当前节点有前置节点的话，那么公平锁就不会再去竞争了，但是非公平锁是会再次竞争的。那么这里什么时候会把线程放入到同步队列里面呢？再看回上面的tryAcquire这个方法，这里也就是走到了current == getExclusiveOwnerThread()这一句了，那么这句的代码又是什么意思呢？这里关于这个方法的解释是The current owner of exclusive mode synchronization. 也就是用来记录当前获取锁的线程的。也就是说，这里如果当前的锁再次获取到锁的话，那么这里就会走这个方法了，会把state累加。

那么下面就总结一下这个流程:



那么这里说明了非公平锁其实是有两次竞争机会的，而公平锁在第一次的时候有一次获取锁的机会，但是后面就只能够根据同步队列来获取锁了。但是这里同步队列到底是什么时候处理线程的，这个有待研究。另外改造一下最开始的程序，这里修改一下Run方法如下所示:

```
@Override
    public void run() {
        try {
            start.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 连续两次打印当前的Thread和等待队列中的Thread
        for (int i = 0; i < 2; i++) {
            //这里添加了slowly等待时间是为了防止释放锁之后，当前线程又立马获得锁了，给其他线程时间来竞争
            slowly();
            lock.lock(); //加锁
            try {
                System.out.println("Lock by [" + getName() + "], Waiting by "
                 + ((MyReentrantLock) lock).getQueuedThreads());
            } finally {
                System.out.println(getName()+" is unlock ");
                lock.unlock();
                //slowly();
            }
```

```
        }
    }
```

那么这里的如果使用非公平锁的结果如下所示:

```
Lock by [5], Waiting by [8]
5 is unlock
Lock by [8], Waiting by []
8 is unlock
Lock by [7], Waiting by []
7 is unlock
Lock by [6], Waiting by [9, 4, 3]
6 is unlock
Lock by [9], Waiting by [4, 3, 0, 2, 1, 8, 5]
9 is unlock
Lock by [4], Waiting by [3, 0, 2, 1, 8, 5]
4 is unlock
Lock by [3], Waiting by [0, 2, 1, 8, 5]
3 is unlock
Lock by [0], Waiting by [2, 1, 8, 5]
0 is unlock
Lock by [2], Waiting by [1, 8, 5]
2 is unlock
Lock by [1], Waiting by [8, 5]
1 is unlock
Lock by [7], Waiting by [8, 5, 6]
7 is unlock
Lock by [8], Waiting by [5, 6, 9, 4, 0, 2, 3]
8 is unlock
Lock by [5], Waiting by [6, 9, 4, 0, 2, 3]
5 is unlock
Lock by [6], Waiting by [9, 4, 0, 2, 3]
6 is unlock
Lock by [9], Waiting by [4, 0, 2, 3]
9 is unlock
Lock by [4], Waiting by [0, 2, 3]
4 is unlock
Lock by [0], Waiting by [2, 3]
0 is unlock
Lock by [2], Waiting by [3, 1]
2 is unlock
Lock by [3], Waiting by [1]
3 is unlock
Lock by [1], Waiting by []
1 is unlock
```

可以看到这里是乱序的，也出现了多次获得锁的情况。说明这里是有竞争的。那么下面测试公平锁的结果

```
Lock by [0], Waiting by [7, 9, 6, 8, 5, 3, 2, 1, 4]
0 is unlock
Lock by [7], Waiting by [9, 6, 8, 5, 3, 2, 1, 4]
7 is unlock
Lock by [9], Waiting by [6, 8, 5, 3, 2, 1, 4]
9 is unlock
Lock by [6], Waiting by [8, 5, 3, 2, 1, 4, 0, 7]
6 is unlock
Lock by [8], Waiting by [5, 3, 2, 1, 4, 0, 7]
8 is unlock
Lock by [5], Waiting by [3, 2, 1, 4, 0, 7]
5 is unlock
Lock by [3], Waiting by [2, 1, 4, 0, 7]
3 is unlock
Lock by [2], Waiting by [1, 4, 0, 7]
2 is unlock
Lock by [1], Waiting by [4, 0, 7]
1 is unlock
Lock by [4], Waiting by [0, 7]
4 is unlock
Lock by [0], Waiting by [7, 9, 6, 8, 5, 3, 2, 1]
0 is unlock
Lock by [7], Waiting by [9, 6, 8, 5, 3, 2, 1]
7 is unlock
Lock by [9], Waiting by [6, 8, 5, 3, 2, 1]
9 is unlock
Lock by [6], Waiting by [8, 5, 3, 2, 1]
6 is unlock
Lock by [8], Waiting by [5, 3, 2, 1]
8 is unlock
Lock by [5], Waiting by [3, 2, 1, 4]
```

```
5 is unlock
Lock by [3], Waiting by [2, 1, 4]
3 is unlock
Lock by [2], Waiting by [1, 4]
2 is unlock
Lock by [1], Waiting by [4]
1 is unlock
Lock by [4], Waiting by []
4 is unlock
```

可以看到这里线程是按照队列里面的FIFO的，证明这里是有序的。