21 更高级的锁—深入解析Lock

更新时间：2019-11-07 10:15:23

> 没有智慧的头脑，就象没有腊烛的灯笼。
>
> ——列夫·托尔斯泰

前面的章节我们刚刚学习了 Java 的内置锁，也就是 synchronized 关键字的使用。在 Java 5.0 之前只有 synchronized 和 volatile 可以用来进行同步。在 Java 5.0 之后，出现了新的同步机制，也就是使用 ReentrantLock 显式的加锁。而 ReentrantLock 的诞生并不是用来取代 synchroinized。而是应该在 synchroinized 无法满足我们需求的时候才使用 ReentrantLock。

我们生活中也是一样的，不要过分追求名牌、追求功能齐全。其实绝大多数情况下，我们选择一般的产品已经足够用了。一个产品 80% 的功能其实在你淘汰它之前都不会用到。当然，如果你确实有需求，那么还是应该选择更为高级的产品。

1、ReentrantLock 的使用

简单应用

ReentrantLock 的使用相比较 synchronized 会稍微繁琐一点，所谓显示锁，也就是你在代码中需要主动的去进行 lock 操作。一般来讲我们可以按照下面的方式使用 ReentrantLock。

```
Lock lock = new ReentrantLock();
lock.lock();
try {
 doSomething();
}finally {
 lock.unlock();
}
```

lock.lock () 就是在显式的上锁。上锁后，下面的代码块一定要放到 try 中，并且要结合 finally 代码块调用 lock.unlock () 来释放锁，否则一定 doSomething 方法中出现任何异常，这个锁将永远不会被释放掉。

公平锁和非公平锁

synchronized 是非公平锁，也就是说每当锁匙放的时候，所有等待锁的线程并不会按照排队顺去依次获得锁，而是会再次去争抢锁。ReentrantLock 相比较而言更为灵活，它能够支持公平和非公平锁两种形式。只需要在声明的时候传入 true。

```
Lock lock = new ReentrantLock(true);
```

而默认的无参构造方法则会创建非公平锁。

tryLock

前面我们通过 lock.lock (); 来完成加锁，此时加锁操作是阻塞的，直到获取锁才会继续向下进行。ReentrantLock 其实还有更为灵活的枷锁方式 tryLock。tryLock 方法有两个重载，第一个是无参数的 tryLock 方法，被调用后，该方法会立即返回获取锁的情况。获取为 true，未能获取为 false。我们的代码中可以通过返回的结果进行进一步的处理。第二个是有参数的 tryLock 方法，通过传入时间和单位，来控制等待获取锁的时长。如果超过时间未能获取锁则放回 false，反之返回 true。使用方法如下：

```
if(lock.tryLock(2, TimeUnit.SECONDS)){
  try {
    doSomething();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }finally {
    lock.unlock();
  }
}else{
 doSomethingElse();
}
```

我们如果不希望无法获取锁时一直等待，而是希望能够去做一些其它事情时，可以选择此方式。

2、lock 方法源码分析

我们先从 lock 方法看起。lock 方法的代码如下：

```
public void lock() {
      sync.lock();
}
```

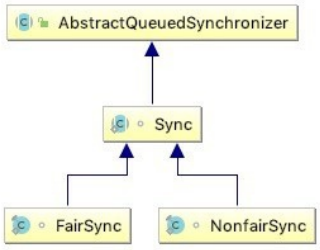
通过内置的 sync 对象加锁，那么 sync 对象是什么呢？我们来看 ReentrantLock 的无参构造函数：

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

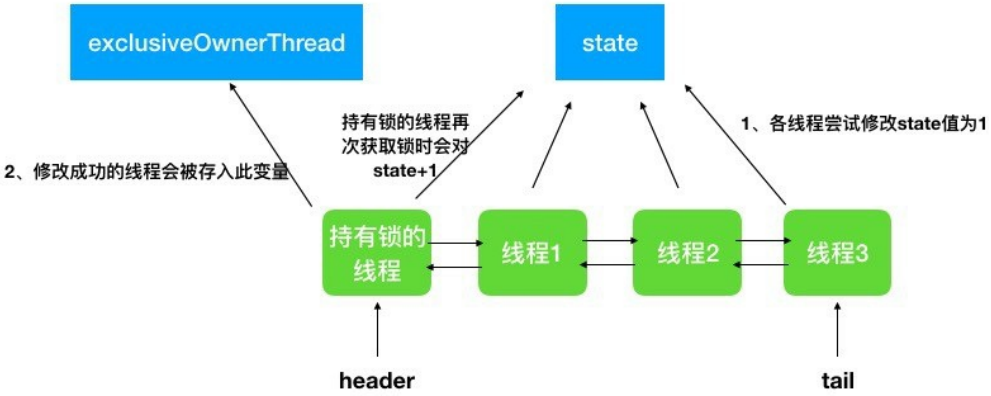
有参的构造函数：

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

FairSync 和 NonFairSync 都继承自 Sync。它们的继承关系如下图：



都是最终继承自 AbstractQueuedSynchronizer。这就是 Java 中著名的 AQS。通过查看 AQS 的注释我们了解到，AQS 依赖先进先出队列实现了阻塞锁和相关的同步器（信号量、事件等）。AQS 内部有一个 volatile 类型的 state 属性，实际上多线程对锁的竞争体现在对 state 值写入的竞争。一旦 state 从 0 变为 1，代表有线程已经竞争到锁，那么其它线程则进入等待队列。等待队列是一个链表结构的 FIFO 队列，这能够确保公平锁的实现。同一线程多次获取锁时，如果之前该线程已经持有锁，那么对 state 再次加 1。释放锁时，则会对 state-1。直到减为 0，才意味着此线程真正释放了锁。



我们回过头来，继续跟进 sync.lock (); 的源代码。我们对代码的分析选择公平锁这条线。FairSync 实现的 lock 代码很简单：

```
final void lock() {
    acquire(1);
}
```

在 FairSync 并没有重写 acquire 方法代码。调用的为 AbstractQueuedSynchronizer 的代码，如下：

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

首先调用一次 tryAcquire 方法。如果 tryAcquire 方法返回 true，那么 acquire 就会立即返回。但如果 tryAcquire 返回了 false，那么则会先调用 addWaiter，把当前线程包装成一个等待的 node，加入到等待队列。然后调用 acquireQueued 尝试排队获取锁，如果成功后发现自己被中断过，那么返回 true，导致 selfInterrupt 被触发，这个方里只是调用 Thread.currentThread ().interrupt (); 进行 interrupt。

acquireQueued 代码如下：

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

在此方法中进入自旋，不断查看自己排队的情况。如果轮到自己（ header 是已经获取锁的线程，而 header 后面的线程是排队到要去获取锁的线程），那么调用 tryAcquire 方法去获取锁，然后把自己设置为队列的 header。在自旋中，如果没有排队到自己，还会检查是否应该应该被中断。

整个获取锁的过程我们可以总结下：

1. 直接通过 tryAcquire 尝试获取锁，成功直接返回；
2. 如果没能获取成功，那么把自己加入等待队列；
3. 自旋查看自己的排队情况；
4. 如果排队轮到自己，那么尝试通过 tryAcquire 获取锁；
5. 如果没轮到自己，那么回到第三步查看自己的排队情况。

从以上过程我们可以看到锁的获取是通过 tryAcquire 方法。而这个方法在 FairSync 和 NonfairSync 有不同实现，我们来分析在 FairSync 中的实现。

```
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;
}
```

实际上它的实现和 NonfairSync 的实现，值是在 c==0 时，多了对 hasQueuedPredecessors 方法的调用。故名思义，这个方法做的事情就是判断当前线程是否前面还有排队的线程。当它前面没有排队线程，说明已经排队到自己了，这是才会通过 CAS 的的方式去改变 state 值为 1，如果成功，那么说明当前线程获取锁成功。接下来就是调用 setExclusiveOwnerThread 把自己设置成为锁的拥有者。else if 中逻辑则是在处理重入逻辑，如果当前线程就是锁的拥有者，那么会把 state 加 1 更新回去。

通过以上分析，我们可以看出 AbstractQueuedSynchronizer 提供 acquire 方法的模板逻辑，但其中真正对锁的获取方法 tryAcquire，是在不同子类中实现的，这是很好的设计思想。

3、unlock 方法源码分析

下面我们来分析 unlock 的源码：

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

和 lock 很像，实际调用的是 sync 实现类的 release 方法。和 lock 方法一样，这个 release 方法在 AbstractQueuedSynchronizer 中，

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

这个方法中会先执行 tryRelease，它的实现也在 AbstractQueuedSynchronizer 的子类 Sync 中，如果释放锁成功，那么则会通过 unparkSuccessor 方法找到队列中第一个 waitStatus<0 的线程进行唤醒。我们下面看一下 tryRelease 方法代码：

```
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;
}
```

还是比较简单，释放的时候会把 state 减 1，如果减到 0，那么说明没有线程持有锁，则会设置 free=true 并且清空锁的持有者。如果 state 值还是大于 0，这说明可重入锁还有其它线程持有，那么锁并没有被真正释放，仅仅是减少了持有的数量，所以返回 false。

总结

本节学习了 ReentrantLock 的使用及其核心源代码，其实 Lock 相关的代码还有很多。我们可以尝试自己去阅读。ReentrantLock 的设计思想是通过 FIFO 的队列保存等待锁的线程。通过 volatile 类型的 state 保存锁的持有数量，从而实现了锁的可重入性。而公平锁则是通过判断自己是否排队成功，来决定是否去争抢锁。学习完本节相信你一定会有疑问，为什么在内置锁之外又设计了 Lock 显式锁呢？下一节，我们将对这两种锁进行对比，看看各自适合的场景。

}

20 其实不用造轮子—Executor框架详解

22 到底哪把锁更适合你？—synchronized与ReentrantLock对比