ReentrantLock 实现原理

使用 `synchronized` 来做同步处理时，锁的获取和释放都是隐式的，实现的原理是通过编译后加上不同的机器指令来实现。

而 `ReentrantLock` 就是一个普通的类，它是基于 `AQS(AbstractQueuedSynchronizer)` 来实现的。

是一个**重入锁**：一个线程获得了锁之后仍然可以**反复**的加锁，不会出现自己阻塞自己的情况。

> *`AQS` 是 `Java` 并发包里实现锁、同步的一个重要的基础框架。*

锁类型

ReentrantLock 分为**公平锁**和**非公平锁**，可以通过构造方法来指定具体类型：

```
//默认非公平锁
public ReentrantLock() {
    sync = new NonfairSync();
}

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

默认一般使用**非公平锁**，它的效率和吞吐量都比公平锁高的多(后面会分析具体原因)。

获取锁

通常的使用方式如下：

```
private ReentrantLock lock = new ReentrantLock();
public void run() {
    lock.lock();
    try {
        //do bussiness
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

公平锁获取锁

首先看下获取锁的过程：

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

可以看到是使用 `sync` 的方法，而这个方法是一个抽象方法，具体是由其子类(`FairSync`)来实现的，以下是公平锁的实现:

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

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

第一步是尝试获取锁(`tryAcquire(arg)`),这个也是由其子类实现：

```
        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;
        }
    }
```

首先会判断 `AQS` 中的 `state` 是否等于 0，0 表示目前没有其他线程获得锁，当前线程就可以尝试获取锁。

注意:尝试之前会利用 `hasQueuedPredecessors()` 方法来判断 AQS 的队列中中是否有其他线程，如果有则不会尝试获取锁(**这是公平锁特有的情况**)。

如果队列中没有线程就利用 CAS 来将 AQS 中的 state 修改为1，也就是获取锁，获取成功则将当前线程置为获得锁的独占线程(`setExclusiveOwnerThread(current)`)。

如果 `state` 大于 0 时，说明锁已经被获取了，则需要判断获取锁的线程是否为当前线程(`ReentrantLock` 支持重入)，是则需要将 `state + 1`，并将值更新。

写入队列

如果 `tryAcquire(arg)` 获取锁失败，则需要用 `addWaiter(Node.EXCLUSIVE)` 将当前线程写入队列中。

写入之前需要将当前线程包装为一个 `Node` 对象(`addWaiter(Node.EXCLUSIVE)`)。

> *AQS 中的队列是由 Node 节点组成的双向链表实现的。*

包装代码:

```
    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;
    }
```

首先判断队列是否为空，不为空时则将封装好的 `Node` 利用 `CAS` 写入队尾，如果出现并发写入失败就需要调用 `enq(node);` 来写入了。

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

这个处理逻辑就相当于`自旋`加上 `CAS` 保证一定能写入队列。

挂起等待线程

写入队列之后需要将当前线程挂起(利用`acquireQueued(addWaiter(Node.EXCLUSIVE), arg)`):

```
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);
    }
}
```

首先会根据 `node.predecessor()` 获取到上一个节点是否为头节点，如果是则尝试获取一次锁，获取成功就万事大吉了。

如果不是头节点，或者获取锁失败，则会根据上一个节点的 `waitStatus` 状态来处理 (`shouldParkAfterFailedAcquire(p, node)`)。

`waitStatus` 用于记录当前节点的状态，如节点取消、节点等待等。

`shouldParkAfterFailedAcquire(p, node)` 返回当前线程是否需要挂起，如果需要则调用 `parkAndCheckInterrupt()`：

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

他是利用 `LockSupport` 的 `part` 方法来挂起当前线程的，直到被唤醒。

非公平锁获取锁

公平锁与非公平锁的差异主要在获取锁：

公平锁就相当于买票，后来的人需要排到队尾依次买票，**不能插队**。

而非公平锁则没有这些规则，是**抢占模式**，每来一个人不会去管队列如何，直接尝试获取锁。

非公平锁：

```
        final void lock() {
            //直接尝试获取锁
            if (compareAndSetState(0, 1))
                setExclusiveOwnerThread(Thread.currentThread());
            else
                acquire(1);
        }
```

公平锁:

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

还要一个重要的区别是在尝试获取锁时`tryAcquire(arg)`，非公平锁是不需要判断队列中是否还有其他线程，也是直接尝试获取锁:

```
        final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                //没有 !hasQueuedPredecessors() 判断
                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;
        }
```

释放锁

公平锁和非公平锁的释放流程都是一样的:

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

public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
               //唤醒被挂起的线程
            unparkSuccessor(h);
        return true;
    }
    return false;
}

//尝试释放锁
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;
}
```

首先会判断当前线程是否为获得锁的线程，由于是重入锁所以需要将 `state` 减到 0 才认为完全释放锁。

释放之后需要调用 `unparkSuccessor(h)` 来唤醒被挂起的线程。

总结

由于公平锁需要关心队列的情况，得按照队列里的先后顺序来获取锁(会造成大量的线程上下文切换)，而非公平锁则没有这个限制。

所以也就能解释非公平锁的效率会被公平锁更高。