ReentrantLock的原理

ReentrantLock实现了Lock接口，提供了lock、trylock、unlock等方法。这些方法通过AQS同步器来管理锁状态，实现加锁和解锁。ReentrantLock包含几个特性：**公平锁、可重入、非阻塞获取锁、可中断**等，下面来看看这些特性是如何实现的。

先简单介绍下AQS，它是并发包里锁管理的核心，包含几个重要属性：
1. state字段，锁计数器来记录锁的状态
2. thread字段，拥有锁的线程
3. Node，Node包含pre和next指针，实现了CLH等待队列

公平性

ReentrantLock中通过继承Sync实现了公平(FairSync)和非公平(NonfairSync)两种同步器。主要区别体现在lock方法上，lock方法会调用tryAcquire方法，去获取锁。公平和非公平同步器获取锁的方式就相差hasQueuedPredecessors()这一个方法。

```
//公平锁tryAcquire方法片段
if (c == 0) {
    if (!hasQueuedPredecessors() && //非公平锁没有!hasQueuedPredecessors()这个
条件
        compareAndSetState(0, acquires)) {
        setExclusiveOwnerThread(current);
        return true;
    }
}

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());
}
```

hasQueuedPredecessors()方法是判断，等待队列上是否有非当前线程符合锁获取条件。
所以公平锁实现是先判断锁没有被获取，且等待队列没有其他 **符合锁获取条件**的线程在等待，那

么当前线程才尝试去获取锁。而非公平锁是不管有没有线程在等待，都直接去尝试获取锁。

> **符合锁获取条件**：等待线程处于等待队列的第二个，第一个是代表当前获取锁的线程，第二个代表下一个可以获取锁的线程

可重入

可重入性也是体现在加锁的时候。加锁时，如果锁已经被占用，当前线程会判断占有锁的线程是不是自己。如果是那么再次进入锁，锁计数器(state)加n。

```
if (c == 0) {
    if (compareAndSetState(0, acquires)) {
        setExclusiveOwnerThread(current);
        return true;
    }
}
//可重入性
else if (current == getExclusiveOwnerThread()) {
    int nextc = c + acquires;
    if (nextc < 0) // overflow
        throw new Error("Maximum lock count exceeded");
    setState(nextc);
    return true;
}
```

非阻塞获取锁

通过tryLock方法，可以实现非阻塞获取锁。获取锁时，如果锁没被占用，则通过compareAndSetState更新锁状态为1代表被占用，更新失败再判断是否可重入，都失败那么直接返回，并不会阻塞等待。
如果是设置了超时的tryLock，在获取锁失败后加入等待队列，并通过LockSupport.parkNanos方法使线程进入有限时间的阻塞，线程没被唤醒或者阻塞时间到，则获取锁失败。

可中断

一般情况ReentrantLock在获取锁的时候，如果线程被中断，中断信息并不会被抛出来，如果通过tryInterruptibly方法获取锁，就可以捕获到中断异常，终止锁获取行为。

```
//tryInterruptibly实际调用该方法
```

```
public final void acquireInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    //获取锁失败
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}

private void doAcquireInterruptibly(int arg)
        throws InterruptedException {
    final Node node = addWaiter(Node.EXCLUSIVE);
    boolean failed = true;
    try {
        for (;;) {
            //判断当前线程是否符合所获取条件，尝试获取锁
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                //抛出线程中断异常
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

doAcquireInterruptibly方法先尝试获取锁，如果获取失败，则进入等待队列，线程等待被唤醒或者被中断。如果是被中断信息，会重新抛出一个InterruptedException异常，导致锁获取行为中断。

关于锁的释放

ReentrantLock是可重入的独占锁，对于锁的释放有点特殊。每次重入锁时，都会在同步器的计数器上加1，而调用unlock释放锁时，是对计数器减1，所以重入多次数和释放次数要一样，计数器为0后锁才能被其他线程使用。

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

public final boolean release(int arg) {
    //如果锁释放成功，就更新head节点状态，并唤醒下一个node
    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;
    //计数器为0时，返回值free才成功
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```