你知道ReentrantLock吗，谈一谈对它的理解

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：ReentrantLock

题目描述

深入理解ReentrantLock原理

题目解决

ReentrantLock是什么?

ReentrantLock是个典型的独占模式AQS，同步状态为0时表示空闲。当有线程获取到空闲的同步状态时，它会将同步状态加1，将同步状态改为非空闲，于是其他线程挂起等待。在修改同步状态的同时，并记录下自己的线程，作为后续重入的依据，即一个线程持有某个对象的锁时，再次去获取这个对象的锁是可以成功的。如果是不可重入的锁的话，就会造成死锁。

ReentrantLock会涉及到公平锁和非公平锁，实现关键在于成员变量`sync`的实现不同,这是锁实现互斥同步的核心。

```
//公平锁和非公平锁的变量
private final Sync sync;
//父类
abstract static class Sync extends AbstractQueuedSynchronizer {}
//公平锁子类
static final class FairSync extends Sync {}
//非公平锁子类
static final class NonfairSync extends Sync {}
```

那公平锁和非公平锁是什么？有什么区别?

那公平锁和非公平锁是什么？有什么区别?

公平锁是指当锁可用时,在锁上等待时间最长的线程将获得锁的使用权，即先进先出。而非公平锁则随机分配这种使用权，是一种抢占机制，是随机获得锁，并不是先来的一定能先得到锁。

ReentrantLock提供了一个构造方法，可以实现公平锁或非公平锁:

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

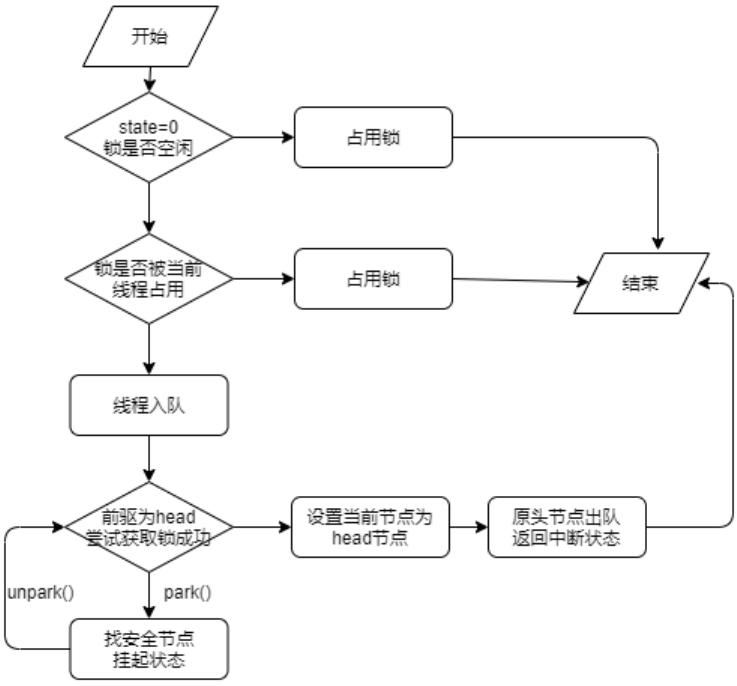
虽然公平锁在公平性得以保障，但因为公平的获取锁没有考虑到操作系统对线程的调度因素以及其他因素，会影响性能。

虽然非公平模式效率比较高，但是非公平模式在申请获取锁的线程足够多,那么可能会造成某些线程长时间得不到锁，这就是非公平锁的“饥饿”问题。

但大部分情况下我们使用非公平锁，因为其性能比公平锁好很多。但是公平锁能够避免线程饥饿，某些情况下也很有用。

接下来看看ReentrantLock公平锁的实现：

ReentrantLock::lock公平锁模式实现



首先需要在构建函数中传入`true`创建好公平锁

```
  ReentrantLock reentrantLock = new ReentrantLock(true);
```

调用`lock()`进行上锁，直接`acquire(1)`上锁

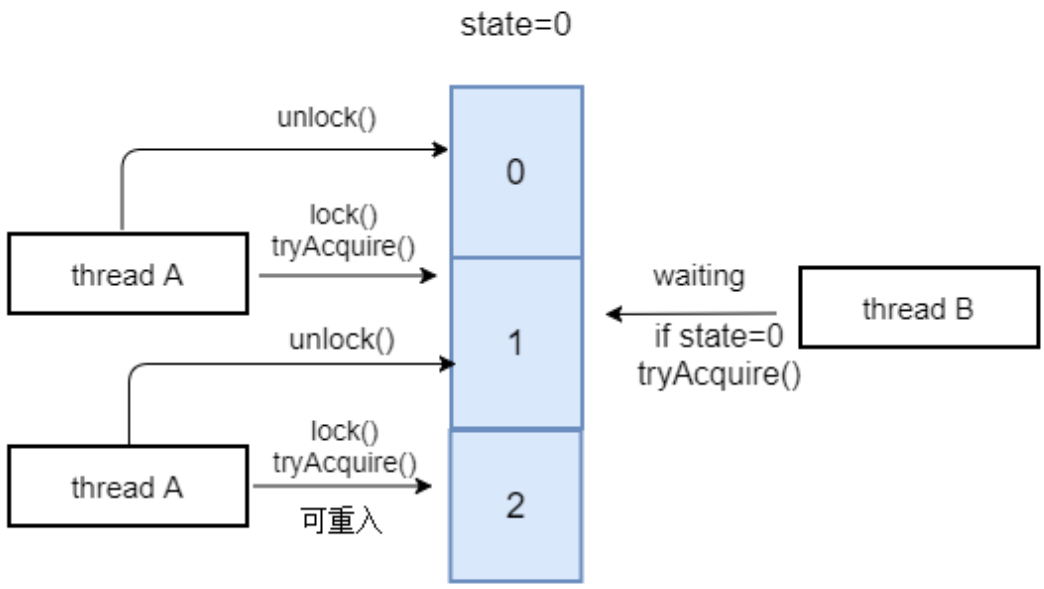
```
public void lock() {
    // 调用的sync的子类FairSync的lock()方法：ReentrantLock.FairSync.lock()
    sync.lock();
}
final void lock() {
    // 调用AQS的acquire()方法获取锁，传的值为1
    acquire(1);
}
```

直接尝试获取锁，

```
// AbstractQueuedSynchronizer.acquire()
public final void acquire(int arg) {
    // 尝试获取锁
    // 如果失败了，就排队
    if (!tryAcquire(arg) &&
        // 注意addWaiter()这里传入的节点模式为独占模式
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

具体获取锁流程

- `getState()` **获取同步状态** `state` **值，进行判断是否为0**:
  - **如果状态变量的值为0，说明暂时还没有人占有锁， 使用hasQueuedPredecessors()保证了不论是新的线程还是已经排队的线程都顺序使用锁，如果没有其它线程在排队，那么当前线程尝试更新state的值为1，并自己设置到exclusiveOwnerThread变量中，供后续自己可重入获取锁作准备。**
  - **如果exclusiveOwnerThread中为当前线程说明本身就占有着锁，现在又尝试获取锁，需要将状态变量的值** `state+1`




```
// ReentrantLock.FairSync.tryAcquire()
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // 状态变量的值为0，说明暂时还没有线程占有锁
    if (c == 0) {
        // hasQueuedPredecessors()保证了不论是新的线程还是已经排队的线程都顺序使用锁
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            // 当前线程获取了锁，并将本线程设置到exclusiveOwnerThread变量中，
            //供后续自己可重入获取锁作准备
            setExclusiveOwnerThread(current);
            return true;
        }
    }
```

```
    // 之所以说是重入锁，就是因为在获取锁失败的情况下，还会再次判断是否当前线程已经持有锁了
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        // 设置到state中
        // 因为当前线程占有着锁，其它线程只会CAS把state从0更新成1，是不会成功的
        // 所以不存在竞争，自然不需要使用CAS来更新
        setState(nextc);
        return true;
    }
    return false;
}
```

如果获取失败加入队列里，那具体怎么处理呢？通过自旋的方式，队列中线程不断进行尝试获取锁操作，中间是可以通过中断的方式打断，

- 如果当前节点的前一个节点为head节点，则说明轮到自己获取锁了，调用`tryAcquire()`方法再次尝试获取锁

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        // 自旋
        for (;;) {
            // 当前节点的前一个节点，
            final Node p = node.predecessor();
            // 如果当前节点的前一个节点为head节点，则说明轮到自己获取锁了
            // 调用ReentrantLock.FairSync.tryAcquire()方法再次尝试获取锁
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                // 未失败
                failed = false;
                return interrupted;
            }
            // 是否需要阻塞
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {

        if (failed)
              // 如果失败了，取消获取锁
            cancelAcquire(node);
    }
}
```

- 当前的Node的上一个节点不是Head,是需要判断是否需要阻塞，以及寻找安全点挂起。

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    // 上一个节点的等待状态
    int ws = pred.waitStatus;
```

```
    // 等待状态为SIGNAL(等待唤醒)，直接返回true
    if (ws == Node.SIGNAL)
        return true;
    // 前一个节点的状态大于0，已取消状态
    if (ws > 0) {
        // 把前面所有取消状态的节点都从链表中删除
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 前一个Node的状态小于等于0，则把其状态设置为等待唤醒
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

在看完获取锁的流程，那么你知道ReentrantLock如何实现公平锁了吗？其实就是在`tryAcquire()`的实现中。

ReentrantLock如何实现公平锁?

在`tryAcquire()`的实现中使用了`hasQueuedPredecessors()`保证了线程先进先出FIFO的使用锁，不会产生"饥饿"问题，

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // 状态变量的值为0，说明暂时还没有线程占有锁
    if (c == 0) {
        // hasQueuedPredecessors()保证了不论是新的线程还是已经排队的线程都顺序使用锁
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
          ....
        }
        ...
    }
}
public final boolean hasQueuedPredecessors() {
        Node t = tail;
        Node h = head;
        Node s;
        return h != t &&
            ((s = h.next) == null || s.thread != Thread.currentThread());
    }
```

tryAcquire都会检查CLH队列中是否仍有前驱的元素，如果仍然有那么继续等待，通过这种方式来保证先来先服务的原则。

那这样ReentrantLock如何实现可重入？是怎么重入的?

ReentrantLock如何实现可重入?

其实也很简单，在获取锁后，设置一个标识变量为当前线程`exclusiveOwnerThread`，当线程再次进入判断`exclusiveOwnerThread`变量是否等于本线程来判断.

```
protected final boolean tryAcquire(int acquires) {

    // 状态变量的值为0，说明暂时还没有线程占有锁
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            // 当前线程获取了锁，并将本线程设置到exclusiveOwnerThread变量中，
            //供后续自己可重入获取锁作准备
            setExclusiveOwnerThread(current);
            return true;
        }
    } //之所以说是重入锁，就是因为在获取锁失败的情况下，还会再次判断是否当前线程已经持有锁了
    else if (current == getExclusiveOwnerThread()) {
        ...
    }

}
```

当看完公平锁获取锁的流程，那其实我们也了解非公平锁获取锁，那我们来看看。

ReentrantLock公平锁模式与非公平锁获取锁的区别?

其实非公平锁获取锁获取区别主要在于:

- 构建函数中传入`false`或者为null，为创建非公平锁`NonfairSync`,`true`创建公平锁,
- 非公平锁在获取锁的时候，先去检查`state`状态，再直接执行`aqcuire(1)`，这样可以提高效率,

```
final void lock() {
            if (compareAndSetState(0, 1))
                //修改同步状态的值成功的话，设置当前线程为独占的线程
                setExclusiveOwnerThread(Thread.currentThread());
            else
                //获取锁
                acquire(1);
        }
```

- 在`tryAcquire()`中没有`hasQueuedPredecessors()`保证了不论是新的线程还是已经排队的线程都顺序使用锁。

其他功能都类似。在理解了获取锁下，我们更好理解ReentrantLock::unlock()锁的释放，也比较简单。

ReentrantLock::unlock()释放锁，如何唤醒等待队列中的线程?

- 释放当前线程占用的锁

```
protected final boolean tryRelease(int releases) {
    // 计算释放后state值
    int c = getState() - releases;
    // 如果不是当前线程占用锁,那么抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
```

```
            // 锁被重入次数为0,表示释放成功
            free = true;
            // 清空独占线程
            setExclusiveOwnerThread(null);
        }
        // 更新state值
        setState(c);
        return free;
    }
```

- **若释放成功，就需要唤醒等待队列中的线程**，先查看头结点的状态是否为SIGNAL，如果是则唤醒头结点的下个节点关联的线程，如果释放失败那么返回false表示解锁失败。
    - 设置waitStatus为0，
    - 当头结点下一个节点不为空的时候，会直接唤醒该节点，如果该节点为空，则会队尾开始向前遍历，找到
    - - 历，找到最后一个不为空的节点，然后唤醒。

```
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    if (ws < 0)
    compareAndSetWaitStatus(node, ws, 0);
    Node s = node.next;//这里的s是头节点（现在是头节点持有锁）的下一个节点，也就是期望唤醒的节点
    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); //唤醒s代表的线程
}
```

综合上面的ReentrantLock的可重入，可实现公平\非公平锁的特性外，还具有哪些特性?

ReentrantLock除了可重入还有哪些特性?

- 支持线程中断，只是在线程上增加一个中断标志`interrupted`，并不会对运行中的线程有什么影响，具体需要根据这个中断标志干些什么，用户自己去决定。比如，实现了等待锁的时候，5秒没有获取到锁，中断等待，线程继续做其它事情。
- 超时机制，在`ReetrantLock::tryLock(long timeout, TimeUnit unit)`提供了超时获取锁的功能。它的语义是在指定的时间内如果获取到锁就返回true，获取不到则返回false。这种机制避免了线程无限期的等待锁释放。

ReentrantLock与Synchrionized的区别

- ReentrantLock支持等待可中断，可以中断等待中的线程
- ReentrantLock可实现公平锁
- ReentrantLock可实现选择性通知，即可以有多个Condition队列

ReentrantLock使用场景

- 场景1：如果已加锁，则不再重复加锁，多用于进行非重要任务防止重复执行，如，清除无用临时文件，检查某些资源的可用性，数据备份操作等
- 场景2：如果发现该操作已经在执行，则尝试等待一段时间，等待超时则不执行，防止由于资源处理不当长时间占用导致死锁情况
- 场景3：如果发现该操作已经加锁，则等待一个一个加锁，主要用于对资源的争抢（如：文件操作，同步消息发送，有状态的操作等）
- 场景4：可中断锁，取消正在同步运行的操作，来防止不正常操作长时间占用造成的阻塞