注：下文中会多次出现同步队列这个关键词，这里的同步队列指的是没有获取到锁而处于阻塞状态的线程形成的队列。

1. ReentrantLock使用

上篇文章介绍的synchronized关键字是一种隐式锁，即它的加锁与释放是自动的，无需我们关心。而ReentrantLock是一种显式锁，需要我们手动编写加锁和释放锁的代码。下面我们来看下ReentrantLock的使用方法。

public class ReentrantLockDemo {

// 实例化一个非公平锁，构造方法的参数为true表示公平锁，false为非公平锁。

private final ReentrantLock lock = new ReentrantLock(false);

private int i;

public void testLock() {

// 拿锁，如果拿不到会一直等待

lock.lock();

try {

// 再次尝试拿锁(可重入)，拿锁最多等待100毫秒

if (lock.tryLock(100, TimeUnit.MILLISECONDS))

i++;

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

// 释放锁

lock.unlock();

lock.unlock();

}

}

}

上述代码中lock.lock()会进行拿锁操作，如果拿不到锁则会一直等待。如果拿到锁之后则会执行try代码块中的代码。接下来在try代码块中又通过tryLock(100, TimeUnit.MILLISECONDS)方法尝试再次拿锁，此时，拿锁最多会等待100毫秒，如果在100毫秒内能获得锁，则tryLock方法返回true，拿锁成功，执行i++操作，如果返回false，获取锁失败，i++不会被执行。（因为第一次线程已经拿到锁了，由于ReentrantLock是可重入，因此，第二次必定能拿到锁。此处仅仅为了演示ReetranLock的使用，不必纠结）。

另外，要注意被ReentrantLock加锁区域必须用try代码块包裹，且释放锁需要在finally中来避免死锁。执行几次加锁，就需要几次释放锁。

### 2.公平锁与非公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入同步队列中排队，队列中最先到的线程先获得锁。非公平锁是多个线程加锁时每个线程都会先去尝试获取锁，如果刚好获取到锁，那么线程无需等待，直接执行，如果获取不到锁才会被加入同步队列的队尾等待执行。

当然，公平锁和非公平锁各有优缺点，适用于不同的场景。公平锁的优点在于各个线程公平平等，每个线程等待一段时间后，都有执行的机会，而它的缺点相较于于非公平锁整体执行速度更慢，吞吐量更低。同步队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

而非公平锁非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程。它的缺点呢也比较明显，即队列中等待的线程可能一直或者长时间获取不到锁。

3.可重入锁与非可重入锁

在本章的第1小节同时也提到了可重入锁的概念：

可重入锁又名递归锁，是指同一个线程在获取外层同步方法锁的时候，再进入该线程的内层同步方法会自动获取锁（前提锁对象得是同一个对象或者class），不会因为之前已经获取过还没释放而阻塞。非可重入锁与可重入锁是对立的关系，即一个线程在获取到外层同步方法锁后，再进入该方法的内层同步方法无法获取到锁，即使锁是同一个对象。

上篇文章讲到的synchronized与本篇讲的ReentrantLock都属于可重入锁。可重入锁可以有效避免死锁的产生。

4.排他锁与共享锁

由于后文中还会涉及到排它锁与共享锁的概念，因此在这里一并解释了。

排他锁也叫独占锁，是指该锁一次只能被一个线程所持有。如果线程T对数据A加上排它锁后，则其他线程不能再对A加任何类型的锁。获得排它锁的线程即能读数据又能修改数据。共享锁是指该锁可被多个线程所持有。如果线程T对数据A加上共享锁后，则其他线程只能对A再加共享锁，不能加排它锁。获得共享锁的线程只能读数据，不能修改数据。

接下来我们来看一下ReentrantLock类的代码结构：

public class ReentrantLock implements Lock, java.io.Serializable {

private final Sync sync;

public ReentrantLock() {

sync = new NonfairSync();

}

public ReentrantLock(boolean fair) {

sync = fair ? new FairSync() : new NonfairSync();

}

// ...省略其它代码

}

可以看到，ReentrantLock的代码结构非常简单。它实现了Lock和Serializable两个接口，同时有两个构造方法，在无参构造方法中初始化了一个非公平锁，在有参构造方法中根据参数决定是初始化公平锁还是非公平锁。

接下来，我们看一下Lock接口的代码：

public interface Lock {

// 获取锁

void lock();

// 获取可中断锁，即在拿锁过程中可中断，synchronized是不可中断锁。

void lockInterruptibly() throws InterruptedException;

// 尝试获取锁，成功返回true，失败返回false

boolean tryLock();

// 在给定时间内尝试获取锁，成功返回true，失败返回false

boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

// 释放锁

void unlock();

// 等待与唤醒机制

Condition newCondition();

}

可以看到在Lock中定义了多个获取锁的方法，以及释放锁的方法。同时还有一个与等待与唤醒机制有关系的newCondition方法。本篇文章我们暂且不讨论Condition。

那么既然ReentrantLock实现Lock接口，它一定也实现了这些方法，看下ReentrantLock中的实现：

public class ReentrantLock implements Lock, java.io.Serializable {

private final Sync sync;

public void lock() {

sync.acquire(1);

}

public void lockInterruptibly() throws InterruptedException {

sync.acquireInterruptibly(1);

}

public boolean tryLock() {

return sync.nonfairTryAcquire(1);

}

public boolean tryLock(long timeout, TimeUnit unit)

throws InterruptedException {

return sync.tryAcquireNanos(1, unit.toNanos(timeout));

}

public void unlock() {

sync.release(1);

}

// ...省略其它代码

}

可以看到ReentrantLock中对这几个方法的实现非常简单。都是调用了Sync中的相关方法。可见ReentrantLock所有拿锁和释放锁的操作都是通过Sync这个成员变量来实现的。Sync是ReentrantLock中的一个抽象内部类，它的源码如下：

abstract static class Sync extends AbstractQueuedSynchronizer {

// 尝试获取非公平锁

final boolean nonfairTryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

// 未上锁状态

if (c == 0) {

// 通过CAS尝试拿锁

if (compareAndSetState(0, acquires)) {

// 设置持有排他锁的线程

setExclusiveOwnerThread(current);

return true;

}

}

// 如果是已上锁状态，判断持有锁的线程是不是自己，这里即可重入锁的实现

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

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;

}

protected final boolean isHeldExclusively() {

return getExclusiveOwnerThread() == Thread.currentThread();

}

// 获取持有锁的线程

final Thread getOwner() {

return getState() == 0 ? null : getExclusiveOwnerThread();

}

// 获取持有锁线程重入的次数

final int getHoldCount() {

return isHeldExclusively() ? getState() : 0;

}

// 是否上锁状态

final boolean isLocked() {

return getState() != 0;

}

// ...省略其他代码

}

可以看到Sync中的代码逻辑也并不复杂。通过nonfairTryAcquire方法实现非公平锁的拿锁操作，tryRelease则实现了释放锁的操作。其它还有几个与锁状态相关的方法。细心的同学会发现Sync对锁状态的判断都是通过state来实现的，state为0表示未加锁状态，state大于0表示加锁状态。关于state我们在后文中还会提及。

另外，由于Sync是一个抽象类，那必然有继承它的类。在ReentrantLock中有两个Sync的实现，分别为NonfairSync与FairSync。从名字上可以看到一个是非公平锁，一个是非公平锁。

首先来看下NonfairSync非公平锁的实现：

static final class NonfairSync extends Sync {

private static final long serialVersionUID = 7316153563782823691L;

protected final boolean tryAcquire(int acquires) {

return nonfairTryAcquire(acquires);

}

}

上边我们也提到了Sync中已经实现了非公平锁的逻辑了，所以NonfairSync的代码非常简单，仅仅在tryAcquire中直接调用了nonfairTryAcquire。

FairSync公平锁的代码如下：

static final class FairSync extends Sync {

protected final boolean tryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

// 未上锁状态

if (c == 0) {

// 首先判断没有等待节点时才会开启CAS去拿锁

if (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

// 设置持有排他锁的线程

setExclusiveOwnerThread(current);

return true;

}

}

// 与非公平锁一样，持有锁线程是自己，则可重入

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0)

throw new Error("Maximum lock count exceeded");

setState(nextc);

return true;

}

return false;

}

}

可以看到在tryAcquire中实现了公平锁的操作，这段代码与与非公平锁的实现其实只有一句之差。即公平锁先去判断了同步队列中是否有在等待的线程，如果没有才会去进行拿锁操作。而非公平锁不会管是否有同步队列，先去拿了再说。

到这里，关于ReentrantLock的源码基本已经分析完了，但是我们并没有看到拿锁和释放锁的底层逻辑。而这些逻辑都在Sync的父类AbstractQueuedSynchronizer中实现。

AbstractQueuedSynchronizer可以翻译为队列同步器，通常简称为AQS。国际惯例，还是先来看一下AbstractQueuedSynchronizer类的内部结构：

public abstract class AbstractQueuedSynchronizer

extends AbstractOwnableSynchronizer

implements java.io.Serializable {

protected AbstractQueuedSynchronizer() { }

// 同步队列的头结点

private transient volatile Node head;

// 同步队列的尾结点

private transient volatile Node tail;

// 同步状态

private volatile int state;

}

AbstractQueuedSynchronizer类继承了AbstractOwnableSynchronizer，AbstractOwnableSynchronizer中的代码如下：

public abstract class AbstractOwnableSynchronizer

implements java.io.Serializable {

private transient Thread exclusiveOwnerThread;

protected final void setExclusiveOwnerThread(Thread thread) {

exclusiveOwnerThread = thread;

}

protected final Thread getExclusiveOwnerThread() {

return exclusiveOwnerThread;

}

}

可以看到AbstractOwnableSynchronizer与AbstractQueuedSynchronizer中维护了四个成员，分别为：Thread类型的exclusiveOwnerThread，Node类型的head和tail以及一个int类型的state。

exclusiveOwnerThread 表示独占当前锁的线程

head与tail 分别表示了等待线程队列的头结点和尾结点；

state 表示同步的状态，为0时表示未加锁状态，而大于0时表示加锁状态。

看到这里，不禁想起上篇文章讲到的synchronized锁中的monitor对象。在monitor对象中同样维护了一个\_ower字段表示持有锁的线程，维护了一个\_EntryList和\_WaitSet的集合用来存放阻塞和等待的线程，以及一个计数器count表示锁的状态，为0时即为未加锁状态，大于0时则为加锁状态。

1.ReentrantLock的lock方法

以ReentrantLock的lock方法为例继续分析。我们知道lock方法调用了Sync的acquire：

// ReentrantLock

public void lock() {

sync.acquire(1);

}

但Sync中并没有实现acquire方法，而是在Sync的父类AbstractQueuedSynchronizer中实现的。那么我们就来看下AbstractQueuedSynchronizer的acquire方法：

// AbstractQueuedSynchronizer

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

在AQS的acquire方法中首先调用了tryAcquire，而AQS中没有实现tryAcquire,而是抛出了一个异常：

protected boolean tryAcquire(int arg) {

throw new UnsupportedOperationException();

}

tryAcquire的实现在上一章的NonfairSync和FairSync类中已经分析过了，这个方法会通过CAS去尝试拿锁，返回值表示是否成功获取锁。最理想的情况是通过tryAcquire方法直接拿到了锁。但是如果没有拿到锁该怎么办呢？可以看到在tryAcquire返回false的时候接着又调用了addWaiter方法将其加入到了同步队列。这意味着线程进入到了阻塞状态，排队并且等待持有锁的线程释放锁，这一机制主要是依赖一个变形的CLH队列来实现的，同时唤醒线程就是在acquireQueued方法中，后边详细分析。

2.AQS与双向CLH队列

CLH队列是Craig、Landin and Hagersten队列的简称（Craig、Landin and Hagersten是三个人的名字），它是单向链表。而AQS中的队列是CLH变体的虚拟双向队列。在AQS中将所有请求锁失败的线程或者调用了await方法的线程封装成一个Node节点来实现锁的分配。关于Node节点，在上文中已经有所提及，来看一下Node中的代码：

static final class Node {

// 共享模式

static final Node SHARED = new Node();

// 独占模式

static final Node EXCLUSIVE = null;

// 线程已处于结束状态的标记

static final int CANCELLED = 1;

// 线程在等待被唤醒的标记

static final int SIGNAL = -1;

// 条件状态

static final int CONDITION = -2;

// 共享模式下的状态

static final int PROPAGATE = -3;

// 表示等待状态，有以上四种

volatile int waitStatus;

// 同步队列的前驱节点

volatile Node prev;

// 同步队列的后继节点

volatile Node next;

// 阻塞状态或者等待状态的线程

volatile Thread thread;

Node nextWaiter;

Node() {}

Node(Node nextWaiter) {

this.nextWaiter = nextWaiter;

THREAD.set(this, Thread.currentThread());

}

Node(int waitStatus) {

WAITSTATUS.set(this, waitStatus);

THREAD.set(this, Thread.currentThread());

}

// ...省略其他代码

}

可以看到，在Node中封装了等待的线程和线程当前的状态，其中线程的状态有四种，分别为CANCELLED、SIGNAL、CONDITION和PROPAGATE，它们分别表示的含义如下：

CANCELLED 表示线程被取消的状态。同步队列中的线程等待超时或者被中断后会将waitStatus改为CANCELLED。

SIGNAL 表示节点处于被唤醒状态，当其前驱结点释放了同步锁或者被取消后就会通知处于SIGNAL状态的后继节点的线程执行。

CONDITION 调用了await方法后处于等待状态的线程节点会被标记为此种状态，当调用了Condition的singal方法后，CONDITION状态会变为SIGNAL状态，并且会在适当的时机从等待队列转移到同步队列中。

PROPAGATE 这种状态与共享模式有关，在共享模式下，表示节点处于可运行状态。

除此之外，Node中还维护了一个前驱节点和一个后继节点。接下来我们来看一下addWaiter方法是怎么将线程封装成Node并插入到同步队列的队尾的。

// 将当前线程封装成Node，并插入到队尾

private Node addWaiter(Node mode) {

// 实例化包含当前线程的Node节点

Node node = new Node(mode);

// 执行死循环

for (;;) {

Node oldTail = tail;

if (oldTail != null) {

// 设置前驱节点为旧的尾结点

node.setPrevRelaxed(oldTail);

// 用CAS执行尾部结点替换

if (compareAndSetTail(oldTail, node)) {

// oldTail的next节点指向node

oldTail.next = node;

return node;

}

} else {

// 同步队列为空，初始化tail和head，初始化成功后会继续执行死循环，此时oldTail就不为null了

initializeSyncQueue();

}

}

}

// 初始化头结点和尾结点

private final void initializeSyncQueue() {

Node h;

// 注意这里实例化了一个空的Node节点作为头结点

if (HEAD.compareAndSet(this, null, (h = new Node())))

// 将尾结点指向头结点

tail = h;

}

注意，在aquire方法中调用addWaiter方法时传入的参数是Node.EXCLUSIVE，表示独占模式。通过new Node(mode)实例化Node时会设置当前线程。

接下来开启一个死循环进行node插入队尾的操作。如果队列不为空的话，那么通过CAS将node节点插入队尾，如果队列为空，则会去初始化队列，在初始化队列中又实例化了一个空的Node节点作为head，并将tail也指向这个头结点。初始化完成后会继续执行死循环进行node插入操作。从这里也可以看出同步队列的头结点是一个不存储任何数据的节点。

在将节点加入到同步队列后，节点就会开启自旋操作，并观察前驱节点的状态，等待满足执行的条件。这一操作是在acquire方法中的acquireQueued()方法中进行的。

final boolean acquireQueued(final Node node, int arg) {

boolean interrupted = false;

try {

// 开启自旋

for (;;) {

// 获取前驱节点

final Node p = node.predecessor();

// 如果前驱节点就是head节点了则执行tryAcquire尝试获取锁

if (p == head && tryAcquire(arg)) {

// 获取到同步状态后，将当前node设置为头结点

setHead(node);

// 置空后继节点

p.next = null; // help GC

return interrupted;

}

// p不是头结点，则判断是否要挂起线程

if (shouldParkAfterFailedAcquire(p, node))

interrupted |= parkAndCheckInterrupt();

}

} catch (Throwable t) {

cancelAcquire(node);

if (interrupted)

selfInterrupt();

throw t;

}

}

在acquireQueued方法中开启自旋操作，并查看node的前驱节点是不是头结点

如果node前驱节点是头结点，则尝试去获取同步状态，成功之后则可执行同步代码，此时的node节点其实已经无用，调用setHead方法将其设置为head节点（head节点是没有数据的空节点），并置空它的后继节点，以方便垃圾回收。

private void setHead(Node node) {

head = node;

// 置空当前线程

node.thread = null;

// 置空前驱节点

node.prev = null;

}

如果node的前驱节点不是头结点，那么则调用shouldParkAfterFailedAcquire方法判断是否要将线程挂起。如果是则调用parkAndCheckInterrupt将线程挂起。

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

// 获取node的等待状态

int ws = pred.waitStatus;

// 如果状态是等待唤醒则返回true

if (ws == Node.SIGNAL)

return true;

// 状态大于0说明线程处于结束状态

if (ws > 0) {

// 遍历前驱节点，知道找到线程不是结束状态的node

do {

node.prev = pred = pred.prev;

} while (pred.waitStatus > 0);

pred.next = node;

} else {

// 如果等待状态小于0却又不是SIGNAL状态,则CAS将其改为SIGNAL等待唤醒

pred.compareAndSetWaitStatus(ws, Node.SIGNAL);

}

return false;

}

private final boolean parkAndCheckInterrupt() {

// 挂起线程

LockSupport.park(this);

// 返回线程的状态

return Thread.interrupted();

}

通过lock拿锁的流程到此结束。

3.可中断锁lockInterruptibly

在ReentrantLock中还支持可中断锁的获取，是通过lockInterruptibly()和tryLock()方法来实现的。我们以lockInterruptibly方法为例来看它与lock方法的区别。

// ReentrantLock

public void lockInterruptibly() throws InterruptedException {

sync.acquireInterruptibly(1);

}

接着调用AQS的acquireInterruptibly方法:

// AbstractQueuedSynchronizer

public final void acquireInterruptibly(int arg)

throws InterruptedException {

// 如果线程中断，则直接抛出异常

if (Thread.interrupted())

throw new InterruptedException();

// 尝试拿锁

if (!tryAcquire(arg))

// 拿锁失败

doAcquireInterruptibly(arg);

}

如果尝试拿锁失败后调用doAcquireInterruptibly方法(tryLock方法最终也是调用到了doAcquireInterruptibly方法)：

private void doAcquireInterruptibly(int arg)

throws InterruptedException {

final Node node = addWaiter(Node.EXCLUSIVE);

try {

for (;;) {

final Node p = node.predecessor();

if (p == head && tryAcquire(arg)) {

setHead(node);

p.next = null; // help GC

return;

}

if (shouldParkAfterFailedAcquire(p, node) &&

parkAndCheckInterrupt())

// 线程中断抛出异常

throw new InterruptedException();

}

} catch (Throwable t) {

cancelAcquire(node);

throw t;

}

}

可以看到这个方法与acquireQueued方法逻辑几乎一样，而差别在于检测到线程中断后直接抛出异常。

4.锁的释放

ReentrantLock释放锁是通过它自身的unlock方法，而在unlock方法中同样调用了AQS的release方法:

public void unlock() {

sync.release(1);

}

AQS中的release方法的代码如下：

// AbstractQueuedSynchronizer

public final boolean release(int arg) {

// 尝试释放锁

if (tryRelease(arg)) {

Node h = head;

if (h != null && h.waitStatus != 0)

// 唤醒后继节点

unparkSuccessor(h);

return true;

}

return false;

}

protected boolean tryRelease(int arg) {

throw new UnsupportedOperationException();

}

tryRelease是在AbstractQueuedSynchronizer的子类Sync中实现的，上文中我们已经有提及，即操控state，对state减去releases，如果state为0那么久释放锁，并且将排他线程设置为null,最后更新state。代码如下:

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;

}

释放锁成功之后则会调用unparkSuccessor来唤起后继节点。代码如下：

// AbstractQueuedSynchronizer

private void unparkSuccessor(Node node) {

int ws = node.waitStatus;

// 将节点状态修改为0

if (ws < 0)

node.compareAndSetWaitStatus(ws, 0);

// 拿到后继节点

Node s = node.next;

if (s == null || s.waitStatus > 0) { // 后继节点为null或者线程为取消状态

s = null;

// 从尾结点开始遍历

for (Node p = tail; p != node && p != null; p = p.prev)

if (p.waitStatus <= 0) // 说明该节点处于有效状态

s = p;

}

// 唤醒线程

if (s != null)

LockSupport.unpark(s.thread);

}

ReentrantLock独占锁拿锁和排队的流程：ReentrantLock内部通过FairSync和NonfairSync来实现公平锁和非公平锁。它们都是继承自AQS实现，在AQS内部通过state来标记同步状态，如果state为0，线程可以直接获取锁，如果state大于0，则线程会被封装成Node节点进入CLH队列并阻塞线程。AQS的CLH队列是一个双向的链表结构，头结点是一个空的Node节点。新来的node节点会被插入队尾并开启自旋去判断它的前驱节点是不是头结点。如果是头结点则尝试获取锁，如果不是头结点，则根据条件进行挂起操作。

画一个流程图大家可做参考：

