目录

32 ReentrantLock 源码解析

更新时间：2019-11-11 13:38:59

> 才能一旦让懒惰支配，它就一无可为。
>
> ——克雷洛夫

引导语

上两小节我们学习了 AQS，本章我们就要来学习一下第一个 AQS 的实现类：ReentrantLock，看看其底层是如何组合 AQS ，实现了自己的那些功能。

本章的描述思路是先描述清楚 ReentrantLock 的构成组件，然后使用加锁和释放锁的方法把这些组件串起来。

1 类注释

ReentrantLock 中文我们习惯叫做可重入互斥锁，可重入的意思是同一个线程可以对同一个共享资源重复的加锁或释放锁，互斥就是 AQS 中的排它锁的意思，只允许一个线程获得锁。

我们来一起来看下类注释上都有哪些重要信息：

1. 可重入互斥锁，和 synchronized 锁具有同样的功能语义，但更有扩展性；
2. 构造器接受 fairness 的参数，fairness 是 ture 时，保证获得锁时的顺序，false 不保证；
3. 公平锁的吞吐量较低，获得锁的公平性不能代表线程调度的公平性；
4. tryLock() 无参方法没有遵循公平性，是非公平的（lock 和 unlock 都有公平和非公平，而 tryLock 只有公平锁，所以单独拿出来说一说）。

我们补充一下第二点，ReentrantLock 的公平和非公平，是针对获得锁来说的，如果是公平的，可以保证同步队列中的线程从头到尾的顺序依次获得锁，非公平的就无法保证，在释放锁的过程中，我们是没有公平和非公平的说法的。

目录

ReentrantLock 类本身是不继承 AQS 的，实现了 Lock 接口，如下：

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

Lock 接口定义了各种加锁，释放锁的方法，接口有如下几个：

```
// 获得锁方法，获取不到锁的线程会到同步队列中阻塞排队
void lock();
// 获取可中断的锁
void lockInterruptibly() throws InterruptedException;
// 尝试获得锁，如果锁空闲，立马返回 true，否则返回 false
boolean tryLock();
// 带有超时等待时间的锁，如果超时时间到了，仍然没有获得锁，返回 false
boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
// 释放锁
void unlock();
// 得到新的 Condition
Condition newCondition();
```

ReentrantLock 就负责实现这些接口，我们使用时，直接面对的也是这些方法，这些方法的底层实现都是交给 Sync 内部类去实现的，Sync 类的定义如下：

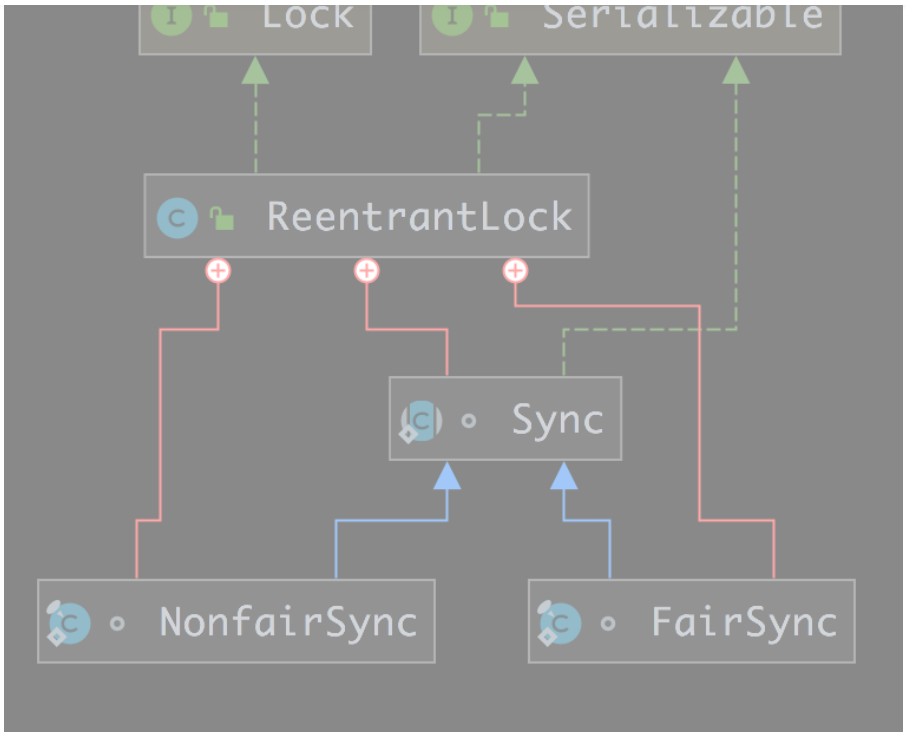
```
abstract static class Sync extends AbstractQueuedSynchronizer {}
```


Sync 继承了 AbstractQueuedSynchronizer，所以 Sync 就具有了锁的框架，根据 AQS 的框架，Sync 只需要实现 AQS 预留的几个方法即可，但 Sync 也只是实现了部分方法，还有一些交给子类 NonfairSync 和 FairSync 去实现了，NonfairSync 是非公平锁，FairSync 是公平锁，定义如下：

```
// 同步器 Sync 的两个子类锁
static final class FairSync extends Sync {}
static final class NonfairSync extends Sync {}
```

几个类整体的结构如下：

目录



图中 Sync、NonfairSync、FairSync 都是静态内部类的方式实现的，这个也符合 AQS 框架定义的实现标准。

3 构造器

ReentrantLock 构造器有两种，代码如下：

```
// 无参数构造器，相当于 ReentrantLock(false)，默认是非公平的
public ReentrantLock() {
    sync = new NonfairSync();
}

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

无参构造器默认构造是非公平的锁，有参构造器可以选择。

从构造器中可以看出，公平锁是依靠 FairSync 实现的，非公平锁是依靠 NonfairSync 实现的。

4 Sync 同步器

Sync 表示同步器，继承了 AQS，UML 图如下：

目录

从 UML 图中可以看出，lock 方法是个抽象方法，留给 FairSync 和 NonfairSync 两个子类去实现，我们一起来看下剩余重要的几个方法。

4.1 nonfairTryAcquire

```
// 尝试获得非公平锁
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // 同步器的状态是 0，表示同步器的锁没有人持有
    if (c == 0) {
        // 当前线程持有锁
        if (compareAndSetState(0, acquires)) {
            // 标记当前持有锁的线程是谁
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // 如果当前线程已经持有锁了，同一个线程可以对同一个资源重复加锁，代码实现的是可重入锁
    else if (current == getExclusiveOwnerThread()) {
        // 当前线程持有锁的数量 + acquires
        int nextc = c + acquires;
        // int 是有最大值的，<0 表示持有锁的数量超过了 int 的最大值
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    //否则线程进入同步队列
    return false;
}
```


以上代码有三点需要注意：

1. 通过判断 AQS 的 state 的状态来决定是否可以获得锁，0 表示锁是空闲的；
2. else if 的代码体现了可重入加锁，同一个线程对共享资源重入加锁，底层实现就是把 state + 1，并且可重入的次数是有限制的，为 Integer 的最大值；
3. 这个方法是非公平的，所以只有非公平锁才会用到，公平锁是另外的实现。

无参的 tryLock 方法调用的就是此方法，tryLock 的方法源码如下：

```
public boolean tryLock() {
    // 入参数是 1 表示尝试获得一次锁
    return sync.nonfairTryAcquire(1);
}
```

目录

```
// 释放锁方法，非公平和公平锁都使用
protected final boolean tryRelease(int releases) {
    // 当前同步器的状态减去释放的个数，releases 一般为 1
    int c = getState() - releases;
    // 当前线程根本都不持有锁，报错
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    // 如果 c 为 0，表示当前线程持有的锁都释放了
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    // 如果 c 不为 0，那么就是可重入锁，并且锁没有释放完，用 state 减去 releases 即可，无需做其
    setState(c);
    return free;
}
```

tryRelease 方法是公平锁和非公平锁都公用的，在锁释放的时候，是没有公平和非公平的说法的。

从代码中可以看到，锁最终被释放的标椎是 state 的状态为 0，在重入加锁的情况下，需要重入解锁相应的次数后，才能最终把锁释放，比如线程 A 对共享资源 B 重入加锁 5 次，那么释放锁的话，也需要释放 5 次之后，才算真正的释放该共享资源了。

5. FairSync 公平锁

FairSync 公平锁只实现了 lock 和 tryAcquire 两个方法，lock 方法非常简单，如下：

```
// acquire 是 AQS 的方法，表示先尝试获得锁，失败之后进入同步队列阻塞等待
final void lock() {
    acquire(1);
}
```

tryAcquire 方法是 AQS 在 acquire 方法中留给子类实现的抽象方法，FairSync 中实现的源码如下：

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        // hasQueuedPredecessors 是实现公平的关键
        // 会判断当前线程是不是属于同步队列的头节点的下一个节点(头节点是释放锁的节点)
        // 如果是(返回false)，符合先进先出的原则，可以获得锁
        // 如果不是(返回true)，则继续等待
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    // 可重入锁
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
```

```
    return false;
}
```

代码和 Sync 的 nonfairTryAcquire 方法实现类似，唯一不同的是在获得锁时使用 hasQueuedPredecessors 方法体现了其公平性。

6 NonfairSync 非公平锁

NonfairSync 底层实现了 lock 和 tryAcquire 两个方法，如下:

```
// 加锁
final void lock() {
    // cas 给 state 赋值
    if (compareAndSetState(0, 1))
        // cas 赋值成功，代表拿到当前锁，记录拿到锁的线程
        setExclusiveOwnerThread(Thread.currentThread());
    else
        // acquire 是抽象类AQS的方法,
        // 会再次尝试获得锁，失败会进入到同步队列中
        acquire(1);
}
// 直接使用的是 Sync.nonfairTryAcquire 方法
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```


7 如何串起来

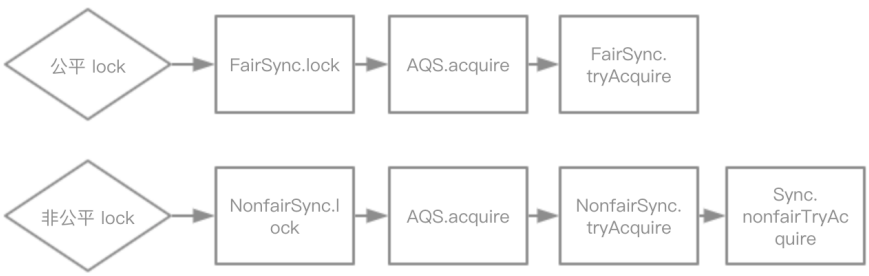
以上内容主要说了 ReentrantLock 的基本结构，比较零散，那么这些零散的结构如何串联起来呢？我们是通过 lock、tryLock、unlock 这三个 API 将以上几个类串联起来，我们来一一看下。

7.1 lock 加锁

lock 的代码实现：

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

其底层的调用关系(只是简单表明调用关系，并不是完整分支图)如下：

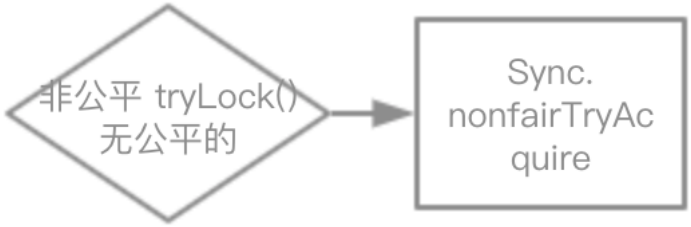


7.2 tryLock 尝试加锁

目录

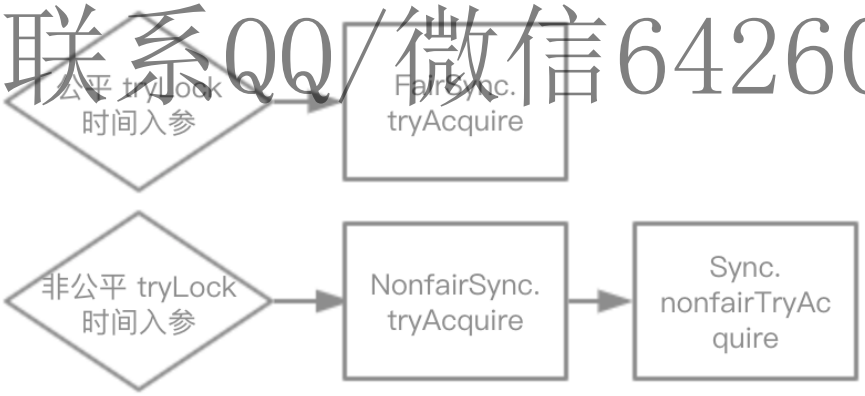
```
// 无参构造器
public boolean tryLock() {
    return sync.nonfairTryAcquire(1);
}
// timeout 为超时的时间，在时间内，仍没有得到锁，会返回 false
public boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException {
    return sync.tryAcquireNanos(1, unit.toNanos(timeout));
}
```

接着我们一起看下两个 tryLock 的调用关系图，下图显示的是无参 tryLock 的调用关系图，如下：



我们需要注意的是 tryLock 无参方法底层走的就是非公平锁实现，没有公平锁的实现。

下图展示的是带有超时时间的有参 tryLock 的调用实现图：



7.3 unlock 释放锁

unlock 释放锁的方法，底层调用的是 Sync 同步器的 release 方法，release 是 AQS 的方法，分成两步：

1. 尝试释放锁，如果释放失败，直接返回 false；
2. 释放成功，从同步队列的头节点的下一个节点开始唤醒，让其去竞争锁。

第一步就是我们上文中 Sync 的 tryRelease 方法（4.1），第二步 AQS 已经实现了。

unLock 的源码如下：

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

ReentrantLock 对 Condition 并没有改造，直接使用 AQS 的 ConditionObject 即可。

8 总结

这就是我们在研究完 AQS 源码之后，碰到的第一个锁，是不是感觉很简单，AQS 搭建了整个锁架构，子类锁只需要根据场景，实现 AQS 对应的方法即可，不仅仅是 ReentrantLock 是这样，JUC 中的其它锁也都是这样，只要对 AQS 了如指掌，锁其实非常简单。

31 AbstractQueuedSynchronizer 源码解析（下）

33 CountDownLatch、Atomic 等其它源码解析

精选留言 2

欢迎在这里发表留言，作者筛选后可公开显示

为了angular耻辱上线

emmm，前面寫的tryLock的無參方法沒有公平性，然後括號裏又說衹有公平鎖。沒有矛盾嗎?

0 回复 2019-12-05

文贤 回复 **为了angular耻辱上线**

同学你好，trylock 有两个方法，一个无参，一个有参，无参方法是非公平的，有参方法是公平的。

回复 7天前

敲木鱼的小和尚

每天跟着笔记，自己理解写博客，终于看懂了AQS，很感谢老师，但是有个点，应该很重要的，同步队列是有空的头节点的，因为这个后面流程一直在处理，加了好多逻辑

0 回复 2019-12-03

千学不如一看，千看不如一练