ReentrantLock实现原理及源码分析

ReentrantLock是Java并发包中提供的一个**可重入的互斥锁**。**ReentrantLock**和**synchronized**在基本用法，行为语义上都是类似的，同样都具有可重入性。只不过相比原生的Synchronized，ReentrantLock增加了一些高级的扩展功能，比如它可以实现**公平锁**，同时也可以绑定**多个Conditon**。

可重入性/公平锁/非公平锁

可重入性

所谓的可重入性，就是**可以支持一个线程对锁的重复获取**，原生的synchronized就具有可重入性，一个用synchronized修饰的递归方法，当线程在执行期间，它是可以反复获取到锁的，而不会出现自己把自己锁死的情况。ReentrantLock也是如此，在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不被阻塞。那么有可重入锁，就有不可重入锁，我们在之前文章中自定义的一个Mutex锁就是个不可重入锁，不过使用场景极少而已。

公平锁/非公平锁

所谓公平锁,顾名思义，意指锁的获取策略相对公平，当多个线程在获取同一个锁时，必须按照锁的申请时间来依次获得锁，排排队，不能插队；非公平锁则不同，当锁被释放时，等待中的线程均有机会获得锁。synchronized是非公平锁，ReentrantLock默认也是非公平的，但是可以通过带boolean参数的构造方法指定使用公平锁，但**非公平锁的性能一般要优于公平锁。**

synchronized是Java原生的互斥同步锁，使用方便，对于synchronized修饰的方法或同步块，无需再显式释放锁。synchronized底层是通过monitorenter和monitorexit两个字节码指令来实现加锁解锁操作的。而ReentrantLock做为API层面的互斥锁，需要显式地去加锁解锁。

```
class X {
    private final ReentrantLock lock = new ReentrantLock();
    // ...

    public void m() {
      lock.lock();  // 加锁
      try {
        // ... 函数主题
      } finally {
        lock.unlock() //解锁
      }
    }
  }
```

源码分析

接下来我们从源码角度来看看ReentrantLock的实现原理，它是如何保证可重入性，又是如何实现公平锁的。

ReentrantLock是基于AQS的，AQS是Java并发包中众多同步组件的构建基础，它通过一个int类型的状态变量state和一个FIFO队列来完成共享资源的获取，线程的排队等待等。AQS是个底层框架，采用模板方法模式，它定义了通用的较为复杂的逻辑

骨架，比如线程的排队，阻塞，唤醒等，将这些复杂但实质通用的部分抽取出来，这些都是需要构建同步组件的使用者无需关心的，使用者仅需重写一些简单的指定的方法即可（其实就是对于共享变量state的一些简单的获取释放的操作）。

上面简单介绍了下AQS，详细内容可参考本人的另一篇文章《Java并发包基石-AQS详解》，此处就不再赘述了。先来看常用的几个方法，我们从上往下推。

无参构造器（默认为公平锁）

```
public ReentrantLock() {
        sync = new NonfairSync();//默认是非公平的
    }
```

sync是ReentrantLock内部实现的一个同步组件，它是Reentrantlock的一个静态内部类，继承于AQS，后面我们再分析。

带布尔值的构造器（是否公平）

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

看到了吧，此处可以指定是否采用公平锁，**FailSync和NonFailSync亦为Reentrantlock的静态内部类，都继承于Sync。**

再来看看几个我们常用到的方法

lock()

```
public void lock() {
        sync.lock();//代理到Sync的lock方法上
    }
```

Sync的lock方法是抽象的，实际的lock会代理到FairSync或是NonFairSync上（根据用户的选择来决定，公平锁还是非公平锁）

lockInterruptibly()

```
public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1);//代理到sync的相应方法上，同lock方法的区别是此方法响应中断
    }
```

此方法响应中断，当线程在阻塞中的时候，若被中断，会抛出InterruptedException异常

tryLock()

```
public boolean tryLock() {
        return sync.nonfairTryAcquire(1);//代理到sync的相应方法上
    }
```

tryLock，尝试获取锁，成功则直接返回true，不成功也不耽搁时间，立即返回false。

unlock()

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

释放锁，调用sync的release方法，其实是AQS的release逻辑。

newCondition()

获取一个conditon，ReentrantLock支持多个Condition

```
public Condition newCondition() {
        return sync.newCondition();
    }
```

其他方法就不再赘述了，若想继续了解可去API中查看。

小结

其实从上面这写方法的介绍，我们都能大概梳理出ReentrantLock的处理逻辑，其内部定义了三个重要的静态内部类，Sync，NonFairSync，FairSync。Sync作为ReentrantLock中公用的同步组件，继承了AQS（要利用AQS复杂的顶层逻辑嘛，线程排队，阻塞，唤醒等等）；NonFairSync和FairSync则都继承Sync，调用Sync的公用逻辑，然后再在各自内部完成自己特定的逻辑（公平或非公平）。

接下来，关于如何实现重入性，如何实现公平性，就得去看这几个静态内部类了

NonFairSync（非公平可重入锁）

Oracle(4)
并发编程(8)
分布式系统
数据结构(2)
算法(5)

随笔档案(20)
2017年7月 (2)
2017年6月 (1)
2017年5月 (2)
2017年4月 (1)
2017年3月 (1)
2017年2月 (1)
2017年1月 (1)
2016年12月 (3)
2016年11月 (4)
2016年10月 (2)
2016年9月 (2)

积分与排名
积分 - 45231
排名 - 8350

最新评论
1. Re:HashMap实现原
@大脸喵感谢支持哈...
--dr
2. Re:HashMap实现原
博主，你好，在分析tab
是2的N次方的时，个人
-->hashcode------>h

```
static final class NonfairSync extends Sync {//继承Sync
        private static final long serialVersionUID = 7316153563782823691L;
        /** 获取锁 */
        final void lock() {
            if (compareAndSetState(0, 1))//CAS设置state状态，若原值是0，将其置为1
                setExclusiveOwnerThread(Thread.currentThread());//将当前线程标记为已持有锁
            else
                acquire(1);//若设置失败，调用AQS的acquire方法，acquire又会调用我们下面重写的tryAcquire方法。这里说的调用失败有两种情况：1当前没有线程获取到资源，state为0，但是将state由0设置为1的时候，其他线程抢占资源，将state修改了，导致了CAS失败；2 state原本就不为0，也就是已经有线程获取到资源了，有可能是别的线程获取到资源，也有可能是当前线程获取的，这时线程又重复去获取，所以去tryAcquire中的nonfairTryAcquire我们应该就能看到可重入的实现逻辑了。
        }
        protected final boolean tryAcquire(int acquires) {
            return nonfairTryAcquire(acquires);//调用Sync中的方法
        }
    }
```

nonfairTryAcquire()

```
final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();//获取当前线程
            int c = getState();//获取当前state值
            if (c == 0) {//若state为0，意味着没有线程获取到资源，CAS将state设置为1，并将当前线程标记我获取到排他锁的线程，返回true
                if (compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {//若state不为0，但是持有锁的线程是当前线程
                int nextc = c + acquires;//state累加1
                if (nextc < 0) // int类型溢出了
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);//设置state，此时state大于1，代表着一个线程多次获锁，state的值即是线程重入的次数
                return true;//返回true，获取锁成功
            }
            return false;//获取锁失败了
        }
```

简单总结下流程：

1.先获取state值，若为0，意味着此时没有线程获取到资源，CAS将其设置为1，设置成功则代表获取到排他锁了；

2.若state大于0，肯定有线程已经抢占到资源了，此时再去判断是否就是自己抢占的，是的话，state累加，返回true，重入成功，state的值即是线程重入的次数；

3.其他情况，则获取锁失败。

来看看可重入公平锁的处理逻辑

FairSync

```
static final class FairSync extends Sync {
        private static final long serialVersionUID = -3000897897090466540L;

        final void lock() {
            acquire(1);//直接调用AQS的模板方法acquire，acquire会调用下面我们重写的这个tryAcquire
        }

        protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();//获取当前线程
            int c = getState();//获取state值
            if (c == 0) {//若state为0，意味着当前没有线程获取到资源，那就可以直接获取资源了吗？NO!这不就跟之前的非公平锁的逻辑一样了
```


x，整体上这是一个多对
----->h.....

3. Re:图解排序算法(一
(选择，冒泡，直接插入)

@Wendy.Jacky是这样

--dr

4. Re:图解排序算法(一
(选择，冒泡，直接插入)

交换数组元素的方法，我
下。正常我们交换元素都
变量，比如：/** * 交换
aram arr * @param a
*/public s.....

5. Re:HashMap实现原

终于看到有个图的了，网
p的文章，全是将使用方
图，给作者打call

阅读排行榜

1. 图解排序算法(一)之3
择，冒泡，直接插入)(7

2. HashMap实现原理及
8)

3. 图解排序算法(三)之堆

4. 图解排序算法(二)之希

5. 图解排序算法(四)之归

评论排行榜

1. HashMap实现原理及

2. 图解排序算法(一)之3
择，冒泡，直接插入)(1

3. 图解排序算法(四)之归

4. 图解排序算法(三)之堆

5. 图解排序算法(二)之希

嘛。看下面的逻辑

```
                if (!hasQueuedPredecessors() &&//判断在时间顺序上，是否有申请锁排在自己之前的线程，若没有，才能去获取，CAS设置state，并标记当前线程为持有排他锁的线程；反之，不能获取！这即是公平的处理方式。
                    compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {//重入的处理逻辑，与上文一致，不再赘述
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
    }
```

可以看到，公平锁的大致逻辑与非公平锁是一致的，不同的地方在于有了!hasQueuedPredecessors()这个判断逻辑，即便state为0，也不能贸然直接去获取，要先去看有没有还在排队的线程，若没有，才能尝试去获取，做后面的处理。反之，返回false，获取失败。

看看这个判断是否有排队中线程的逻辑

hasQueuedPredecessors()

```
    public final boolean hasQueuedPredecessors() {
        Node t = tail; // 尾结点
        Node h = head;//头结点
        Node s;
        return h != t &&
            ((s = h.next) == null || s.thread != Thread.currentThread());//判断是否有排在自己之前的线程
    }
```

需要注意的是，这个判断是否有排在自己之前的线程的逻辑稍微有些绕，我们来梳理下，由代码得知，有两种情况会返回true，我们将此逻辑分解一下（注意：返回true意味着有其他线程申请锁比自己早，需要放弃抢占）

1. **h !=t && (s = h.next) == null**，这个逻辑成立的一种可能是head指向头结点，tail此时还为null。考虑这种情况：当其他某个线程去获取锁失败，需构造一个结点加入同步队列中（假设此时同步队列为空），在添加的时候，需要先创建一个无意义傀儡头结点（在AQS的enq方法中，这是个自旋CAS操作），有可能在将head指向此傀儡结点完毕之后，还未将tail指向此结点。很明显，此线程时间上优于当前线程，所以，返回true，表示有等待中的线程且比自己来的还早。

2.**h != t && (s = h.next) != null && s.thread != Thread.currentThread()**。同步队列中已经有若干排队线程且当前线程不是队列的老二结点，此种情况会返回true。假如没有s.thread !=Thread.currentThread()这个判断的话，会怎么样呢？若当前线程已经在同步队列中是老二结点（头结点此时是个无意义的傀儡结点),此时持有锁的线程释放了资源，唤醒老二结点线程，老二结点线程重新tryAcquire（此逻辑在AQS中的acquireQueued方法中），又会调用到hasQueuedPredecessors，不加s.thread !=Thread.currentThread()这个判断的话，返回值就为true，导致tryAcquire失败。

最后，来看看ReentrantLock的tryRelease，定义在Sync中

```
 protected final boolean tryRelease(int releases) {
            int c = getState() - releases;//减去1个资源
            if (Thread.currentThread() != getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();
            boolean free = false;
            //若state值为0，表示当前线程已完全释放干净，返回true，上层的AQS会意识到资源已空出。若不为0，则表示线程还占有资源，只不过将此次重入的资源的释放了而已，返回false。
            if (c == 0) {
                free = true;//
                setExclusiveOwnerThread(null);
            }
            setState(c);
```

```
            return free;
        }
```

总结

ReentrantLock是一种可重入的，可实现公平性的互斥锁，它的设计基于AQS框架，可重入和公平性的实现逻辑都不难理解，每重入一次，state就加1，当然在释放的时候，也得一层一层释放。至于公平性，在尝试获取锁的时候多了一个判断：是否有比自己申请早的线程在同步队列中等待，若有，去等待；若没有，才允许去抢占。

分类： 并发编程

« 上一篇：Java并发包基石-AQS详解

posted @ 2017-07-29 23:59 dreamcatcher-cx 阅读(538) 评论(0) 编辑 收藏