~ Java多线程系列——深入重入锁ReentrantLock

阅读目录

简述

ReentrantLock 是一个**可重入的互斥（/独占）锁**，又称为“独占锁”。

ReentrantLock通过自定义队列同步器（AQS-AbstractQueuedSychronized，是实现锁的关键）来实现锁的获取与释放。

其可以完全替代 synchronized 关键字。JDK 5.0 早期版本，其性能远好于 synchronized，但 JDK 6.0 开始，JDK 对 synchronized 做了大量的优化，使得两者差距并不大。

“独占”，就是在同一时刻只能有一个线程获取到锁，而其它获取锁的线程只能处于同步队列中等待，只有获取锁的线程释放了锁，后继的线程才能够获取锁。

“可重入”，就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。

该锁还支持获取锁时的**公平和非公平性**选择。“公平”是指“不同的线程获取锁的机制是公平的”，而“不公平”是指“不同的线程获取锁的机制是非公平的”。

简单实例

```
+ View code
import java.util.concurrent.locks.ReentrantLock;
/**
 * Created by zhengbinMac on 2017/3/2.
 */
public class ReenterLock implements Runnable{
    public static ReentrantLock lock = new ReentrantLock();
    public static int i = 0;
    public void run() {
        for (int j = 0;j<100000;j++) {
            lock.lock();
//            lock.lock();
            try {
                i++;
            }finally {
                lock.unlock();
//                lock.unlock();
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        ReenterLock reenterLock = new ReenterLock();
        Thread t1 = new Thread(reenterLock);
        Thread t2 = new Thread(reenterLock);
        t1.start();t2.start();
        t1.join();t2.join();
        System.out.println(i);
    }
}
```

与 synchronized 相比，重入锁有着显示的操作过程，何时加锁，何时释放，都在程序员的控制中。

为什么称作是“重入”？这是因为这种锁是可以反复进入的。将上面代码中注释部分去除注释，也就是连续两次获得同一把锁，两次释放同一把锁，这是允许的。

注意，获得锁次数与释放锁次数要相同，如果释放锁次数多了，会抛出 java.lang.IllegalMonitorStateException 异常；如果释放次数少了，相当于线程还持有这个锁，其他线程就无法进入临界区。

引出第一个问题：**为什么 ReentrantLock 锁能够支持一个线程对资源的重复加锁？**

除了简单的加锁、解锁操作，重入锁还提供了一些更高级的功能，下面结合实例进行简单介绍：

中断响应（lockInterruptibly）

对于 synchronized 来说，如果一个线程在等待锁，那么结果只有两种情况，获得这把锁继续执行，或者线程就保持等待。

而使用重入锁，提供了另一种可能，这就是**线程可以被中断**。也就是在等待锁的过程中，程序可以根据需要取消对锁的需求。

下面的例子中，产生了死锁，但得益于锁中断，最终解决了这个死锁：

+ View code

```
import java.util.concurrent.locks.ReentrantLock;
/**
 * Created by zhengbinMac on 2017/3/2.
 */
public class IntLock implements Runnable{
    public static ReentrantLock lock1 = new ReentrantLock();
    public static ReentrantLock lock2 = new ReentrantLock();
    int lock;
    /**
     * 控制加锁顺序，产生死锁
     */
    public IntLock(int lock) {
        this.lock = lock;
    }
    public void run() {
        try {
            if (lock == 1) {
                lock1.lockInterruptibly(); // 如果当前线程未被 中断，则获取锁。
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                lock2.lockInterruptibly();
                System.out.println(Thread.currentThread().getName()+"，执行完毕！");
            } else {
                lock2.lockInterruptibly();
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                lock1.lockInterruptibly();
                System.out.println(Thread.currentThread().getName()+"，执行完毕！'
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 查询当前线程是否保持此锁。
            if (lock1.isHeldByCurrentThread()) {
                lock1.unlock();
            }
            if (lock2.isHeldByCurrentThread()) {
                lock2.unlock();
            }
            System.out.println(Thread.currentThread().getName() + "，退出。");
        }
    }
```

```
49      public static void main(String[] args) throws InterruptedException {
50          IntLock intLock1 = new IntLock(1);
51          IntLock intLock2 = new IntLock(2);
52          Thread thread1 = new Thread(intLock1, "线程1");
53          Thread thread2 = new Thread(intLock2, "线程2");
54          thread1.start();
55          thread2.start();
56          Thread.sleep(1000);
57          thread2.interrupt(); // 中断线程2
58      }
59  }
```

上述例子中，线程 thread1 和 thread2 启动后，thread1 先占用 lock1，再占用 lock2；thread2 反之，先占 lock2，后占 lock1。这便形成 thread1 和 thread2 之间的相互等待。

代码 56 行，main 线程处于休眠（sleep）状态，两线程此时处于死锁的状态，代码 57 行 thread2 被中断（interrupt），故 thread2 会放弃对 lock1 的申请，同时释放已获得的 lock2。这个操作导致 thread1 顺利获得 lock2，从而继续执行下去。

执行代码，输出如下：

```
线程2, 退出。
线程1, 执行完毕!
线程1, 退出。
java.lang.InterruptedException
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireInterruptibly(AbstractQueuedSynchronizer.java:898)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInterruptibly(AbstractQueuedSynchronizer.java:1222)
    at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.java:335)
    at IntLock.run(IntLock.java:33)
    at java.lang.Thread.run(Thread.java:745)
```

锁申请等待限时（tryLock）

除了等待外部通知（中断操作 interrupt ）之外，限时等待也可以做到避免死锁。

通常，无法判断为什么一个线程迟迟拿不到锁。也许是因为产生了死锁，也许是产生了饥饿。但如果给定一个等待时间，让线程自动放弃，那么对系统来说是有意义的。可以使用 tryLock() 方法进行一次限时的等待。

```
+ View code
 1  import java.util.concurrent.TimeUnit;
 2  import java.util.concurrent.locks.ReentrantLock;
 3  /**
 4   * Created by zhengbinMac on 2017/3/2.
 5   */
 6  public class TimeLock implements Runnable{
 7      public static ReentrantLock lock = new ReentrantLock();
 8      public void run() {
 9          try {
10              if (lock.tryLock(5, TimeUnit.SECONDS)) {
11                  Thread.sleep(6 * 1000);
12              }else {
13                  System.out.println(Thread.currentThread().getName()+" get Lock Failed");
14              }
15          } catch (InterruptedException e) {
16              e.printStackTrace();
17          }finally {
18              // 查询当前线程是否保持此锁。
19              if (lock.isHeldByCurrentThread()) {
20                  System.out.println(Thread.currentThread().getName()+" release lo
21                  lock.unlock();
22              }
23          }
24      }
25      /**
26       * 在本例中，由于占用锁的线程会持有锁长达6秒，故另一个线程无法再5秒的等待时间内获得锁，因此
27       */
28      public static void main(String[] args) {
29          TimeLock timeLock = new TimeLock();
30          Thread t1 = new Thread(timeLock, "线程1");
31          Thread t2 = new Thread(timeLock, "线程2");
32          t1.start();
33          t2.start();
```

```
34      }
35  }
```

上述例子中，由于占用锁的线程会持有锁长达 6 秒，故另一个线程无法在 5 秒的等待时间内获得锁，因此，请求锁失败。

ReentrantLock.tryLock()方法也可以不带参数直接运行。这种情况下，当前线程会尝试获得锁，如果锁并未被其他线程占用，则申请锁成功，立即返回 true。否则，申请失败，立即返回 false，当前线程不会进行等待。这种模式不会引起线程等待，因此也不会产生死锁。

公平锁

默认情况下，锁的申请都是非公平的。也就是说，如果线程 1 与线程 2，都申请获得锁 A，那么谁获得锁不是一定的，是由系统在等待队列中随机挑选的。这就好比，买票的人不排队，售票姐姐只能随机挑一个人卖给他，这显然是不公平的。而公平锁，它会按照时间的先后顺序，保证先到先得。公平锁的特点是：不会产生饥饿现象。

重入锁允许对其公平性进行设置。构造函数如下：

```
- Hide code
```

下面举例来说明，公平锁与非公平锁的不同：

```
+ View code
1  import java.util.concurrent.locks.ReentrantLock;
2  /**
3   * Created by zhengbinMac on 2017/3/2.
4   */
5  public class FairLock implements Runnable{
6      public static ReentrantLock fairLock = new ReentrantLock(true);
7
8      public void run() {
9          while (true) {
10             try {
11                 fairLock.lock();
12                 System.out.println(Thread.currentThread().getName()+"，获得锁!");
13             }finally {
14                 fairLock.unlock();
15             }
16         }
17     }
18     public static void main(String[] args) {
19         FairLock fairLock = new FairLock();
20         Thread t1 = new Thread(fairLock, "线程1");
21         Thread t2 = new Thread(fairLock, "线程2");
22         t1.start();t2.start();
23     }
24 }
```

修改重入锁是否公平，观察输出结果，如果公平，输出结果始终为两个线程交替的获得锁，如果是非公平，输出结果为一个线程占用锁很长时间，然后才会释放锁，另个线程才能执行。

引出第二个问题：**为什么公平锁例子中出现，公平锁线程是不断切换的，而非公平锁出现同一线程**

结合源码再看“重入”

何为重进入（重入）？

重进入是指任意线程在获取到锁之后能够再次获取该锁而不会被锁阻塞，该特性的实现需要解

- 线程再次获取锁：锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再
- 锁的最终释放。线程重复 n 次获取了锁，随后在第 n 次释放该锁后，其它线程能够获取 数自增，

计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于 0 时表示锁

以非公平锁源码分析：

获取：

```
+ View code
```

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    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;
}
```

acquireQueued 方法增加了再次获取同步状态的处理逻辑：通过判断当前线程是否为获取锁的线程，来决定获取操作是否成功，如果获取锁的线程再次请求，则将同步状态值进行增加并返回 true，表示获取同步状态成功。

成功获取锁的线程再次获取锁，只是增加了同步状态值，也就是要求 ReentrantLock 在释放同步状态时减少同步状态值，释放锁源码如下：

```
+ View code
public void unlock() {
    sync.release(1);
}
public final boolean release(int arg) {
    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;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

如果锁被获取 n 次，那么前 (n-1) 次 tryRelease(int releases) 方法必须返回 false，只有同步状态完全释 0 作为最终释放的条件，当同步状态为 0 时，将占有线程设置为 null，并返回 true，表示释放成功。

通过对获取与释放的分析，就可以解释，以上两个例子中出现的两个问题：为什么 ReentrantLock 公平锁例子中出现，公平锁线程是不断切换的，而非公平锁出现同一线程连续获取锁的情况？

- 为什么支持重复加锁？因为源码中用变量 c 来保存当前锁被获取了多少次，故在释放时 算锁的最终释放。所以可以 lock 多次，同时 unlock 也必须与 lock 同样的次数。
- 为什么非公平锁出现同一线程连续获取锁的情况？tryAcquire 方法中增加了再次获取同步

小结

对上面ReentrantLock的几个重要方法整理如下：

- lock()：获得锁，如果锁被占用，进入等待。
- lockInterruptibly()：获得锁，但优先响应中断。
- tryLock()：尝试获得锁，如果成功，立即放回 true，反之失败返回 false。该方法不会进行等待，立即返回。
- tryLock(long time, TimeUnit unit)：在给定的时间内尝试获得锁。
- unLock()：释放锁。

对于其实现原理，下篇博文将详细分析，其主要包含三个要素：

- 原子状态：原子状态有 CAS（compareAndSetState）操作来存储当前锁的状态，判断锁是否有其他线程持有。
- 等待队列：所有没有请求到锁的线程，会进入等待队列进行等待。待有线程释放锁后，系统才能够从等待队列中唤醒一个线程，继续工作。详见：队列同步器——AQS（待更新）
- 阻塞原语 park() 和 unpark()，用来挂起和恢复线程。没有得到锁的线程将会被挂起。关于阻塞原语，详见：线程阻塞工具类——LockSupport（待更新）。

参考资料

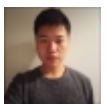
[1] Java并发编程的艺术, 5.3 - 重入锁

[2] 实战Java高并发程序设计, 3.1.1 - synchronized的功能扩展：重入锁

梦想要一步步来！

分类: 备战阿里 , 并发

好文要顶 关注我 收藏该文



郑州的文武
关注 - 36
粉丝 - 176
+加关注

« 上一篇： JVM——深入分析对象的内存布局
» 下一篇： Java多线程系列——过期的suspend()挂起、resume()继续执行线程

posted @ 2017-03-05 09:08 郑州的文武 阅读(1434) 评论(0) 收藏

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。