[https://mp.weixin.qq.com/s?\_\_biz=MzA5MTkxMDQ4MQ==&mid=2648933116&idx=1&sn=83ae2d1381e3b8a425e65a9fa7888d38&chksm=88621ac2bf1593d4de1c5f6905c31c7d88ac4b53c0c5c071022ba2e25803fc734078c1de589c&token=2041017112&lang=zh\_CN&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933116&idx=1&sn=83ae2d1381e3b8a425e65a9fa7888d38&chksm=88621ac2bf1593d4de1c5f6905c31c7d88ac4b53c0c5c071022ba2e25803fc734078c1de589c&token=2041017112&lang=zh_CN&scene=21" \l "wechat_redirect)

# JUC:ReentrantLock重入锁

本篇文章开始将juc中常用的一些类，估计会有十来篇。

## synchronized的局限性

synchronized是java内置的关键字，它提供了一种独占的加锁方式。synchronized的获取和释放锁由jvm实现，用户不需要显示的释放锁，非常方便，然而synchronized也有一定的局限性，例如：

1、当线程尝试获取锁的时候，如果获取不到锁会一直阻塞，这个阻塞的过程，用户无法控制

2、如果获取锁的线程进入休眠或者阻塞，除非当前线程异常，否则其他线程尝试获取锁必须一直等待

JDK1.5之后发布，加入了Doug Lea实现的java.util.concurrent包。包内提供了Lock类，用来提供更多扩展的加锁功能。Lock弥补了synchronized的局限，提供了更加细粒度的加锁功能。

## ReentrantLock简介

ReentrantLock是Lock的默认实现，在聊ReentranLock之前，我们需要先弄清楚一些概念：

1、可重入锁：可重入锁是指同一个线程可以多次获得同一把锁；ReentrantLock和关键字Synchronized都是可重入锁

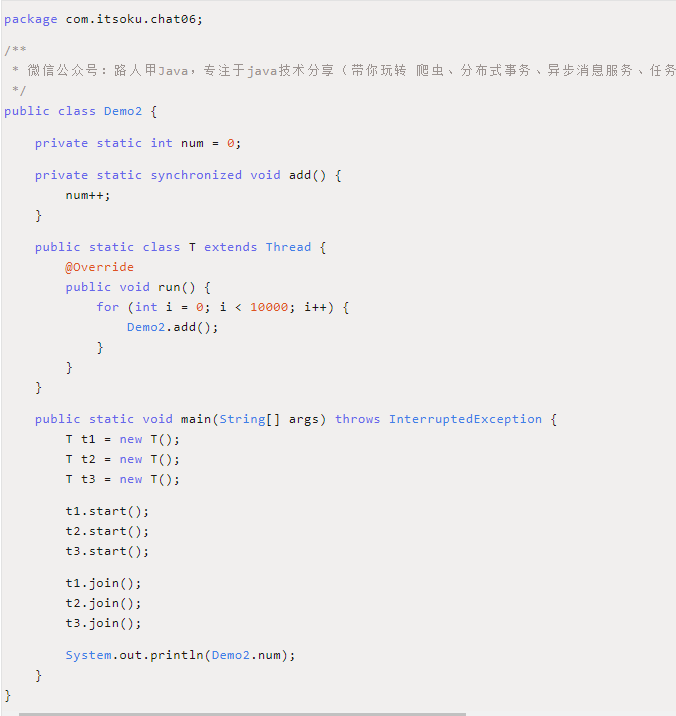
2、可中断锁：可中断锁时只线程在获取锁的过程中，是否可以相应线程中断操作。**synchronized是不可中断的，ReentrantLock是可中断的；**

3、公平锁和非公平锁：公平锁是指多个线程尝试获取同一把锁的时候，获取锁的顺序按照线程到达的先后顺序获取，而不是随机插队的方式获取。synchronized是非公平锁，而ReentrantLock是两种都可以实现，不过默认是非公平锁。

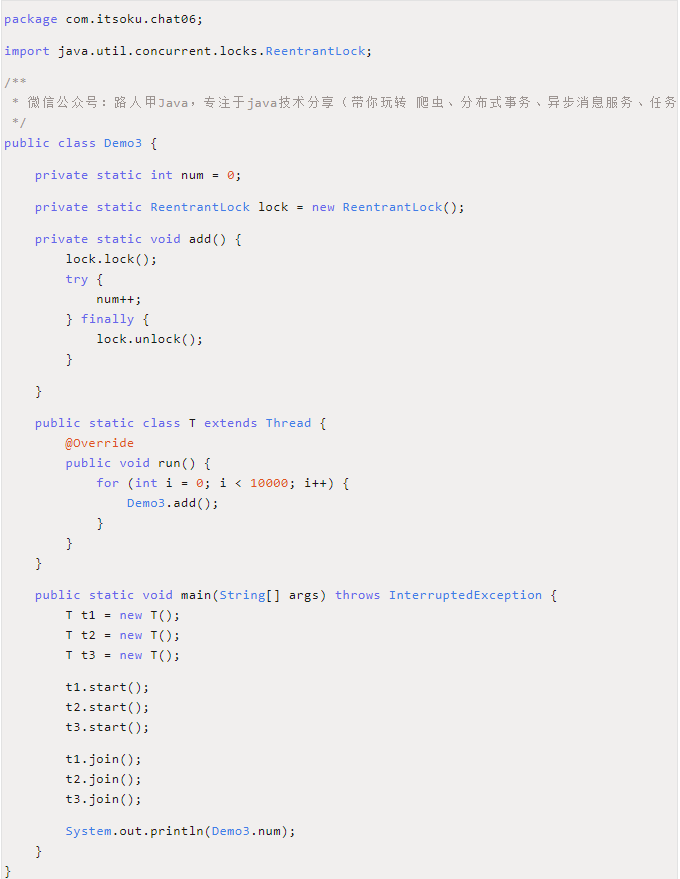
## ReentrantLock基本使用

我们使用3个线程来对一个共享变量++操作，先使用synchronized实现，然后使用ReentrantLock实现。

**synchronized方式：**



**ReentrantLock方式：**



ReentrantLock的使用过程：

1. 创建锁：ReentrantLock lock = new ReentrantLock();
2. 获取锁：lock.lock()
3. 释放锁：lock.unlock();

对比上面的代码，与关键字synchronized相比，ReentrantLock锁有明显的操作过程，开发人员必须手动的指定何时加锁，何时释放锁，正是因为这样手动控制，ReentrantLock对逻辑控制的灵活度要远远胜于关键字synchronized，**上面代码需要注意lock.unlock()一定要放在finally中**，否则，若程序出现了异常，锁没有释放，那么其他线程就再也没有机会获取这个锁了。

## ReentrantLock是可重入锁

来验证一下ReentrantLock是可重入锁，实例代码：



上面代码中add()方法中，当一个线程进入的时候，会执行2次获取锁的操作，运行程序可以正常结束，并输出和期望值一样的30000，假如ReentrantLock是不可重入的锁，那么同一个线程第2次获取锁的时候由于前面的锁还未释放而导致死锁，程序是无法正常结束的。ReentrantLock命名也挺好的Re entrant Lock，和其名字一样，可重入锁。

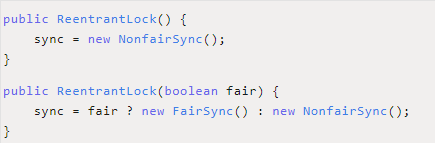
代码中还有几点需要注意：

* lock()方法**和unlock()方法需要成对出现，锁了几次，也要释放几次，否则后面的线程无法获取锁了；可以将add中的unloc**k**删除一个事实，上面代码运行将无法结束**
* **unlock()方法放在finally中执行，保证不管程序是否有异常，锁必定会释放**

## ReentrantLock实现公平锁

在大多数情况下，锁的申请都是非公平的，也就是说，线程1首先请求锁A，接着线程2也请求了锁A。那么当锁A可用时，是线程1可获得锁还是线程2可获得锁呢？这是不一定的，系统只是会从这个锁的等待队列中随机挑选一个，因此不能保证其公平性。这就好比买票不排队，大家都围在售票窗口前，售票员忙的焦头烂额，也顾及不上谁先谁后，随便找个人出票就完事了，最终导致的结果是，有些人可能一直买不到票。而公平锁，则不是这样，它会按照到达的先后顺序获得资源。公平锁的一大特点是：它不会产生饥饿现象，只要你排队，最终还是可以等到资源的；synchronized关键字默认是有jvm内部实现控制的，是非公平锁。而ReentrantLock运行开发者自己设置锁的公平性。

看一下jdk中ReentrantLock的源码，2个构造方法：



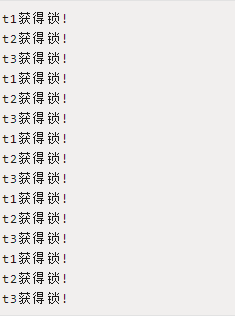
默认构造方法创建的是非公平锁。

第2个构造方法，有个fair参数，当fair为true的时候创建的是公平锁，公平锁看起来很不错，不过要实现公平锁，系统内部肯定需要维护一个有序队列，因此公平锁的实现成本比较高，性能相对于非公平锁来说相对低一些。因此，在默认情况下，锁是非公平的，如果没有特别要求，则不建议使用公平锁。

公平锁和非公平锁在程序调度上是很不一样，来一个公平锁示例看一下：



运行结果：

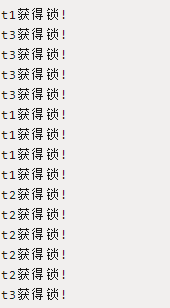


看一下输出的结果，锁时按照先后顺序获得的。

修改一下上面代码，改为非公平锁试试，如下：

ReentrantLock fairLock = new ReentrantLock(false);

运行结果如下：



可以看到t3可能会连续获得锁，结果是比较随机的，不公平的。

## ReentrantLock获取锁的过程是可中断的

对于synchronized关键字，如果一个线程在等待获取锁，最终只有2种结果：

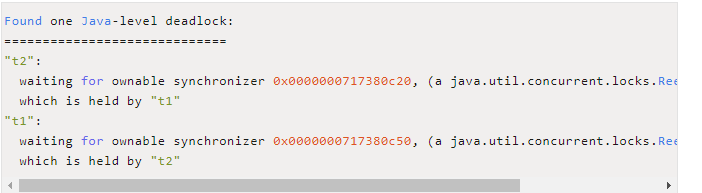
* + 要么获取到锁然后继续后面的操作
  + 要么一直等待，直到其他线程释放锁为止

而ReentrantLock提供了另外一种可能，就是在等的获取锁的过程中（发起获取锁请求到还未获取到锁这段时间内）是可以被中断的，也就是说在等待锁的过程中，程序可以根据需要取消获取锁的请求。有些使用这个操作是非常有必要的。比如：你和好朋友越好一起去打球，如果你等了半小时朋友还没到，突然你接到一个电话，朋友由于突发状况，不能来了，那么你一定打道回府。中断操作正是提供了一套类似的机制，如果一个线程正在等待获取锁，那么它依然可以收到一个通知，被告知无需等待，可以停止工作了。

示例代码：

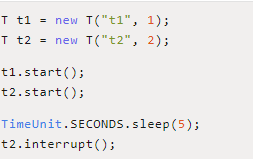


先运行一下上面代码，发现程序无法结束，使用jstack查看线程堆栈信息，发现2个线程死锁了。

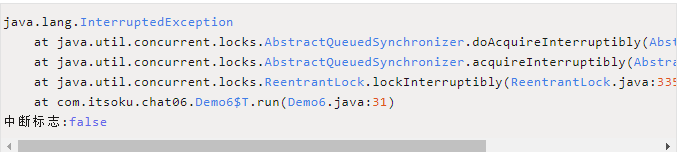


lock1被线程t1占用，lock2倍线程t2占用，线程t1在等待获取lock2，线程t2在等待获取lock1，都在相互等待获取对方持有的锁，最终产生了死锁，如果是在synchronized关键字情况下发生了死锁现象，程序是无法结束的。

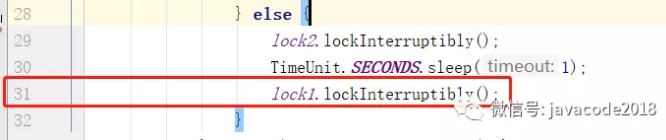
我们队上面代码改造一下，线程t2一直无法获取到lock1，那么等待5秒之后，我们中断获取锁的操作。主要修改一下main方法，如下：



新增了2行代码 TimeUnit.SECONDS.sleep(5);t2.interrupt();，程序可以结束了，运行结果：



从上面信息中可以看出，代码的31行触发了异常，中断标志输出：false



t2在31行一直获取不到lock1的锁，主线程中等待了5秒之后，t2线程调用了 interrupt()方法，将线程的中断标志置为true，此时31行会触发 InterruptedException异常，然后线程t2可以继续向下执行，释放了lock2的锁，然后线程t1可以正常获取锁，程序得以继续进行。**线程发送中断信号触发InterruptedException异常之后，中断标志将被清空。**

关于获取锁的过程中被中断，注意几点:

* ReentrankLock中必须使用实例方法 lockInterruptibly()获取锁时，在线程调用interrupt()方法之后，才会引发 InterruptedException异常
* 线程调用interrupt()之后，线程的中断标志会被置为true
* 触发InterruptedException异常之后，线程的中断标志有会被清空，即置为false
* 所以当线程调用interrupt()引发InterruptedException异常，中断标志的变化是:false->true->false

## ReentrantLock锁申请等待限时

申请锁等待限时是什么意思？一般情况下，获取锁的时间我们是不知道的，synchronized关键字获取锁的过程中，只能等待其他线程把锁释放之后才能够有机会获取到所。所以获取锁的时间有长有短。如果获取锁的时间能够设置超时时间，那就非常好了。

ReentrantLock刚好提供了这样功能，给我们提供了获取锁限时等待的方法 tryLock()，可以选择传入时间参数，表示等待指定的时间，无参则表示立即返回锁申请的结果：true表示获取锁成功，false表示获取锁失败。

### tryLock无参方法

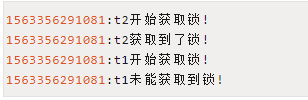
看一下源码中tryLock方法：



返回boolean类型的值，此方法会立即返回，结果表示获取锁是否成功，示例：



代码中获取锁成功之后，休眠5秒，会导致另外一个线程获取锁失败，运行代码，输出：



可以看到t2获取成功，t1获取失败了，tryLock()是立即响应的，中间不会有阻塞。

### tryLock有参方法

可以明确设置获取锁的超时时间，该方法签名：

public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException

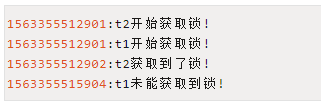
该方法在指定的时间内不管是否可以获取锁，都会返回结果，返回true，表示获取锁成功，返回false表示获取失败。此方法由2个参数，第一个参数是时间类型，是一个枚举，可以表示时、分、秒、毫秒等待，使用比较方便，第1个参数表示在时间类型上的时间长短。此方法在执行的过程中，如果调用了线程的中断interrupt()方法，会触发InterruptedException异常。

示例：



程序中调用了ReentrantLock的实例方法 tryLock(3,TimeUnit.SECONDS)，表示获取锁的超时时间是3秒，3秒后不管是否能否获取锁，该方法都会有返回值，获取到锁之后，内部休眠了5秒，会导致另外一个线程获取锁失败。

运行程序，输出：



输出结果中分析，t2获取到锁了，然后休眠了5秒，t1获取锁失败，t1打印了2条信息，时间相差3秒左右。

**关于tryLock()方法和tryLock(long timeout, TimeUnit unit)方法，说明一下：**

* + 都会返回boolean值，结果表示获取锁是否成功
  + tryLock()方法，不管是否获取成功，都会立即返回；而有参的tryLock方法会尝试在指定的时间内去获取锁，中间会阻塞的现象，在指定的时间之后会不管是否能够获取锁都会返回结果
  + tryLock()方法不会响应线程的中断方法；而有参的tryLock方法会响应线程的中断方法，而出发 InterruptedException异常，这个从2个方法的声明上可以可以看出来

## ReentrantLock其他常用的方法

isHeldByCurrentThread：实例方法，判断当前线程是否持有ReentrantLock的锁，上面代码中有使用过。

## 获取锁的4种方法对比

| **获取锁的方法** | **是否立即响应(不会阻塞)** | **是否响应中断** |
| --- | --- | --- |
| lock() | × | × |
| lockInterruptibly() | × | √ |
| tryLock() | √ | × |
| tryLock(long timeout, TimeUnit unit) | × | √ |

## 总结

* ReentrantLock可以实现公平锁和非公平锁
* ReentrantLock默认实现的是非公平锁
* ReentrantLock的获取锁和释放锁必须成对出现，锁了几次，也要释放几次
* 释放锁的操作必须放在finally中执行
* lockInterruptibly()实例方法可以相应线程的中断方法，调用线程的interrupt()方法时，lockInterruptibly()方法会触发 InterruptedException异常
* 关于 InterruptedException异常说一下，看到方法声明上带有 throwsInterruptedException，表示该方法可以相应线程中断，调用线程的interrupt()方法时，这些方法会触发 InterruptedException异常，触发InterruptedException时，线程的中断中断状态会被清除。所以如果程序由于调用 interrupt()方法而触发 InterruptedException异常，线程的标志由默认的false变为ture，然后又变为false
* 实例方法tryLock()获会尝试获取锁，会立即返回，返回值表示是否获取成功
* 实例方法tryLock(long timeout, TimeUnit unit)会在指定的时间内尝试获取锁，指定的时间内是否能够获取锁，都会返回，返回值表示是否获取锁成功，该方法会响应线程的中断