

Java中的锁

[原文链接](#) 作者: Jakob Jenkov 译者: [申章](#) 校对: 丁一

锁像synchronized同步块一样，是一种线程同步机制，但比Java中的synchronized同步块更复杂。因为锁（以及其它更高级的线程同步机制）是由synchronized同步块的方式实现的，所以我们还不能完全摆脱synchronized关键字（译者注：这说的是Java 5之前的情况）。

自Java 5开始，java.util.concurrent.locks包中包含了一些锁的实现，因此你不用去实现自己的锁了。但是你仍然需要去了解怎样使用这些锁，且了解这些实现背后的理论也是很有用处的。可以参考我对[java.util.concurrent.locks.Lock](#)的介绍，以了解更多关于锁的信息。

以下是本文所涵盖的主题：

1. [一个简单的锁](#)
2. [锁的可重入性](#)
3. [锁的公平性](#)
4. [在finally语句中调用unlock\(\)](#)

一个简单的锁

让我们从java中的一个同步块开始：

1	<code>public class Counter{</code>
2	<code> private int count = 0;</code>
3	
4	<code> public int inc(){</code>
5	<code> synchronized(this){</code>
6	<code> return ++count;</code>
7	<code> }</code>
8	<code> }</code>
9	<code>}</code>

可以看到在inc()方法中有一个synchronized(this)代码块。该代码块可以保证在同一时间只有一个线程可以执行return ++count。虽然在synchronized的同步块中的代码可以更加复杂，但是++count这种简单的操作已经足以表达出线程同步的意思。

以下的Counter类用Lock代替synchronized达到了同样的目的：

01	<code>public class Counter{</code>
02	<code> private Lock lock = new Lock();</code>
03	<code> private int count = 0;</code>
04	
05	<code> public int inc(){</code>
06	<code> lock.lock();</code>
07	<code> int newCount = ++count;</code>
08	<code> lock.unlock();</code>
09	<code> return newCount;</code>
10	<code> }</code>
11	<code>}</code>

lock()方法会对Lock实例对象进行加锁，因此所有对该对象调用lock()方法的线程都会被阻塞，直到该Lock对象的unlock()方法被调用。

这里有一个Lock类的简单实现：

01	<code>public class Counter{</code>
----	------------------------------------

02	<code>public class Lock{</code>
03	<code>private boolean isLocked = false;</code>
04	
05	<code>public synchronized void lock()</code>
06	<code>throws InterruptedException{</code>
07	<code>while(isLocked){</code>
08	<code>wait();</code>
09	<code>}</code>
10	<code>isLocked = true;</code>
11	<code>}</code>
12	
13	<code>public synchronized void unlock(){</code>
14	<code>isLocked = false;</code>
15	<code>notify();</code>
16	<code>}</code>
17	<code>}</code>

注意其中的while(isLocked)循环，它又被叫做“自旋锁”。自旋锁以及wait()和notify()方法在[线程通信](#)这篇文章中有更加详细的介绍。当isLocked为true时，调用lock()的线程在wait()调用上阻塞等待。为防止该线程没有收到notify()调用也从wait()中返回（也称作[虚假唤醒](#)），这个线程会重新去检查isLocked条件以决定当前是否可以安全地继续执行还是需要重新保持等待，而不是认为线程被唤醒了就可以安全地继续执行了。如果isLocked为false，当前线程会退出while(isLocked)循环，并将isLocked设回true，让其它正在调用lock()方法的线程能够在Lock实例上加锁。

当线程完成了[临界区](#)（位于lock()和unlock()之间）中的代码，就会调用unlock()。执行unlock()会重新将isLocked设置为false，并且通知（唤醒）其中一个（若有的话）在lock()方法中调用了wait()函数而处于等待状态的线程。

锁的可重入性

Java中的synchronized同步块是可重入的。这意味着如果一个java线程进入了代码中的synchronized同步块，并因此获得了该同步块使用的同步对象对应的管程上的锁，那么这个线程可以进入由同一个管程对象所同步的另一个java代码块。下面是一个例子：

1	<code>public class Reentrant{</code>
2	<code>public synchronized outer(){</code>
3	<code>inner();</code>
4	<code>}</code>
5	
6	<code>public synchronized inner(){</code>
7	<code>//do something</code>
8	<code>}</code>
9	<code>}</code>

注意outer()和inner()都被声明为synchronized，这在Java中和synchronized(this)块等效。如果一个线程调用了outer()，在outer()里调用inner()就没有什么问题，因为这两个方法（代码块）都由同一个管程对象（“this”）所同步。如果一个线程已经拥有了一个管程对象上的锁，那么它就有权访问被这个管程对象同步的所有代码块。这就是可重入。线程可以进入任何一个它已经拥有的锁所同步着的代码块。

前面给出的锁实现不是可重入的。如果我们像下面这样重写Reentrant类，当线程调用outer()时，会在inner()方法的lock.lock()处阻塞住。

01	<code>public class Reentrant2{</code>
02	<code>Lock lock = new Lock();</code>

03	
04	public outer(){
05	lock.lock();
06	inner();
07	lock.unlock();
08	}
09	
10	public synchronized inner(){
11	lock.lock();
12	//do something
13	lock.unlock();
14	}
15	}

调用outer()的线程首先会锁住Lock实例，然后继续调用inner()。inner()方法中该线程将再一次尝试锁住Lock实例，结果该动作会失败（也就是说该线程会被阻塞），因为这个Lock实例已经在outer()方法中被锁住了。

两次lock()之间没有调用unlock()，第二次调用lock就会阻塞，看过lock()实现后，会发现原因很明显：

01	public class Lock{
02	boolean isLocked = false ;
03	
04	public synchronized void lock()
05	throws InterruptedException{
06	while (isLocked){
07	wait();
08	}
09	isLocked = true ;
10	}
11	
12	...
13	}

一个线程是否被允许退出lock()方法是由while循环（自旋锁）中的条件决定的。当前的判断条件是只有当isLocked为false时lock操作才被允许，而没有考虑是哪个线程锁住了它。

为了让这个Lock类具有可重入性，我们需要对它做一点小的改动：

01	public class Lock{
02	boolean isLocked = false ;
03	Thread lockedBy = null ;
04	int lockedCount = 0;
05	

06	<code>public synchronized void lock()</code>
07	<code>throws InterruptedException{</code>
08	<code>Thread callingThread =</code>
09	<code>Thread.currentThread();</code>
10	<code>while(isLocked && lockedBy != callingThread){</code>
11	<code>wait();</code>
12	<code>}</code>
13	<code>isLocked = true;</code>
14	<code>lockedCount++;</code>
15	<code>lockedBy = callingThread;</code>
16	<code>}</code>
17	
18	<code>public synchronized void unlock(){</code>
19	<code>if(Thread.curentThread() ==</code>
20	<code>this.lockedBy){</code>
21	<code>lockedCount--;</code>
22	
23	<code>if(lockedCount == 0){</code>
24	<code>isLocked = false;</code>
25	<code>notify();</code>
26	<code>}</code>
27	<code>}</code>
28	<code>}</code>
29	
30	<code>...</code>
31	<code>}</code>

注意到现在的while循环（自旋锁）也考虑到了已锁住该Lock实例的线程。如果当前的锁对象没有被加锁(isLocked = false)，或者当前调用线程已经对该Lock实例加了锁，那么while循环就不会被执行，调用lock()的线程就可以退出该方法（译者注：“被允许退出该方法”在当前语义下就是指不会调用wait()而导致阻塞）。

除此之外，我们需要记录同一个线程重复对一个锁对象加锁的次数。否则，一次unlock()调用就会解除整个锁，即使当前锁已经被加锁过多次。在unlock()调用没有达到对应lock()调用的次数之前，我们不希望锁被解除。

现在这个Lock类就是可重入的了。

锁的公平性

Java的synchronized块并不保证尝试进入它们的线程的顺序。因此，如果多个线程不断竞争访问相同的synchronized同步块，就存在一种风险，其中一个或多个线程永远也得不到访问权——也就是说访问权总是分配给了其它线程。这种情况被称作线程饥饿。为了避免这种问题，锁需要实现公平性。本文所展现的锁在内部是用synchronized同步块实现的，因此它们也不保证公平性。[饥饿和公平](#)中有更多关于该内容的讨论。

在finally语句中调用unlock()

如果用Lock来保护临界区，并且临界区有可能会抛出异常，那么在finally语句中调用unlock()就显得非常重要了。这样可以保证这个锁对象可以被解锁以便其它线程能继续对其加锁。以下是一个示例：

1	<code>lock.lock();</code>
2	<code>try{</code>
3	<code>//do critical section code,</code>
4	<code>//which may throw exception</code>
5	<code>} finally {</code>
6	<code>lock.unlock();</code>
7	<code>}</code>

这个简单的结构可以保证当临界区抛出异常时Lock对象可以被解锁。如果不是在finally语句中调用的unlock(), 当临界区抛出异常时, Lock对象将永远停留在被锁住的状态, 这会导致其它所有在该Lock对象上调用lock()的线程一直阻塞。

原创文章, 转载请注明: 转载自[并发编程网 – ifeve.com](http://ifeve.com)

本文链接地址: [Java中的锁](#)