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*)。

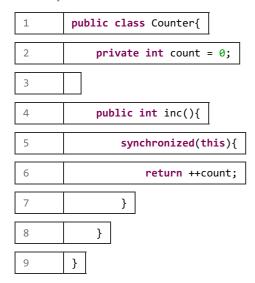
自Java 5开始,java.util.concurrent.locks包中包含了一些锁的实现,因此你不用去实现自己的锁了。但是你仍然需要去了解怎样使用这些锁,且了解这些实现背后的理论也是很有用处的。可以参考我对java.util.concurrent.locks.Lock的介绍,以了解更多关于锁的信息。

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:



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

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 **public class** Counter{

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

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<u>临界区</u>(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

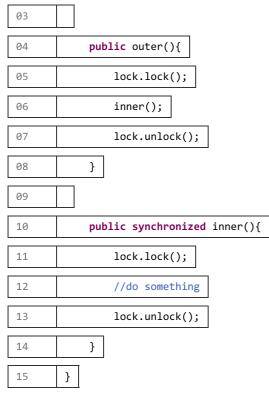
Java中的synchronized同步块是可重入的。这意味着如果一个java线程进入了代码中的synchronized同步块,并因此获得了该同步块使用的同步对象对应的管程上的锁,那么这个线程可以进入由同一个管程对象所同步的另一个java代码块。下面是一个例子:

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

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



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;
            }
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	<pre>public class Lock{</pre>
02	<pre>boolean isLocked = false;</pre>
03	Thread lockedBy = null;
04	<pre>int lockedCount = 0;</pre>
05	

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

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unblock()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。<u>饥饿和公平</u>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	lock.lock();
2	try{
3	//do critical section code,
4	//which may throw exception
5	} finally {
6	lock.unlock();
7	}

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<u>并发编程网 – ifeve.com</u>

本文链接地址: Java中的锁