**Locks in Java**

* [A Simple Lock](http://tutorials.jenkov.com/java-concurrency/locks.html#simple-lock)
* [Lock Reentrance](http://tutorials.jenkov.com/java-concurrency/locks.html#reentrance)
* [Lock Fairness](http://tutorials.jenkov.com/java-concurrency/locks.html#fairness)
* [Calling unlock() From a finally-clause](http://tutorials.jenkov.com/java-concurrency/locks.html#finally)

锁是一种线程同步机制就像同步块，除了锁比Java的同步块更加精妙。锁(和其他的更高级的同步机制)被使用同步块创建，因此我们不大可能能完全摆脱synchronized关键字。

自Java5起java.util.concurrent.locks包包含若干锁实现，因此你可能不必实现自己的锁。但你仍将需要了解如何使用它们，以及了解它们的实现背后的理论仍旧是有用的。关于更多细节，参见关于[**java.util.concurrent.locks.Lock**](http://tutorials.jenkov.com/java-util-concurrent/lock.html)接口的教程。

**A Simple Lock**

Let's start out by looking at a synchronized block of Java code:

public class Counter{

private int count = 0;

public int inc(){

synchronized(this){

return ++count;

}

}

}

记住inc()方法中的synchronized(this)块。这个块儿确保同一时间只有一个线程可以执行return ++count。同步块中的代码可以更加高级，但是简单的++count足以得到点跨越。

Counter类也可以写成这样，使用Lock而不是同步块：

public class Counter{

private Lock lock = new Lock();

private int count = 0;

public int inc(){

lock.lock();

int newCount = ++count;

lock.unlock();

return newCount;

}

}

lock()方法锁定Lock实例以便全部调用lock()的线程被阻塞直到unlock()执行。

这是一个简单的Lock实现：

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

public synchronized void unlock(){

isLocked = false;

notify();

}

}

注意while(isLocked)循环，也被称为自旋锁。自旋锁和方法wait()和notify()的更多细节在[**Thread Signaling**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html)文中讲到。当isLocked为true时，调用lock()的线程停靠在wait()调用中等待。以防线程应当出人意料地从wait()调用返回而未接收到notify()调用(也称为假唤醒[**Spurious Wakeup**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spuriouswakeups))，线程重新检查isLocked条件，看看是否可以安全地继续执行，而不只是假设被唤醒意味着可以安全地继续执行。如果isLocked为false，线程退出while(isLocked)循环，设置isLocked为true，以锁定Lock实例对其他调用lock()的线程。

当线程完成在[**critical section**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html)内部的代码(在lock()和unlock()之间的代码)，线程调用unlock()。执行unlock()设置isLocked为false，唤醒在lock()方法内的wait()调用中等待的线程中的一个，如果存在的话。

**Lock Reentrance**

Java中的同步块是可重入的。这意味着，如果一个Java线程进入一个同步代码块，因此获取在被同步的指示器对象上的锁，线程可以进入其他的在同一个指示器对象上同步的Java代码块。这有个例子：

public class Reentrant{

public synchronized outer(){

inner();

}

public synchronized inner(){

//do something

}

}

注意outer()和inner()都被声明为同步的，在Java中等价于synchronized(this)块。如果一个线程调用outer()，在outer()内部调用inner()方法没有问题，因为两个方法(或代码块)在相同的指示器对象(“this”)上同步。如果一个线程已经在一个指示器对象上持有锁，它可以访问在相同指示器对象上同步的全部块。这被称作可重入。该线程可以重进入任何已经锁定该锁的代码块。

先前展示的锁是不可重入的。如果我们重写Reentrant类如下，调用outer()的线程将被阻塞在inner()方法的lock.lock()内部。

public class Reentrant2{

Lock lock = new Lock();

public outer(){

lock.lock();

inner();

lock.unlock();

}

public synchronized inner(){

lock.lock();

//do something

lock.unlock();

}

}

调用outer()的线程将会首先锁定Lock实例。然后它将调用inner()。在inner()方法内部线程将会再次尝试锁定Lock实例。这将会失败(意味着线程将被锁定)，因为Lock实例已经在outer()方法中被锁定。

线程第二次调用lock()而未在两次中间调用unlock()将被阻塞，原因在我们查看lock()实现时是显而易见的：

public class Lock{

boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

...

}

在while循环(自旋锁)内部的条件决定了是否一个线程允许退出lock()方法。现在条件是isLocked必须为false，以便允许它，无论什么线程锁了它。

为了使Lock类可重入，我们需要做一点小改变：

public class Lock{

boolean isLocked = false;

Thread lockedBy = null;

int lockedCount = 0;

public synchronized void lock()

throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(isLocked && lockedBy != callingThread){

wait();

}

isLocked = true;

lockedCount++;

lockedBy = callingThread;

}

public synchronized void unlock(){

if(Thread.curentThread() == this.lockedBy){

lockedCount--;

if(lockedCount == 0){

isLocked = false;

notify();

}

}

}

...

}

注意while循环(自旋锁)现在也将锁定Lock实例的线程考虑在内。如果或者锁被解开(isLocked = false)或者调用线程是锁定Lock实例的线程，while循环将不会执行，调用lock()的线程将被允许退出方法。

另外，我们需要计数锁被相同线程锁定的次数。否则，单独的unlock()调用将会解锁，即使锁已经被锁定了多次。我们不想要锁被解开直到锁定它的线程已经执行了对比lock()调用相同数量的unlock()调用。

Lock类现在是可重入的。

**Lock Fairness**

Java同步块不保证序列中尝试进入它们的线程被授权访问。因此，如果许多线程频繁竞争访问同一段同步块，有风险一个或多个线程从未被授权访问 - 访问总是授予其他线程。这被称作饥饿。为了避免这个Lock应当是公平的。因为本文中展示的Lock实现内部使用同步块，它们不保证公平。饥饿和公平在[**Starvation and Fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html)文中被讨论更多细节。

**Calling unlock() From a finally-clause**

当使用Lock保障一个临界区间并且临界区间可能抛出异常时，从finally-从句内部调用unlock()方法很重要。这么做确保Lock被解锁，以便其他的线程可以锁定它。这是个例子：

lock.lock();

try{

//do critical section code, which may throw exception

} finally {

lock.unlock();

}

这个小构造确保Lock 被解锁，以防从临界区中的代码中抛出异常。如果未从finally子句内部调用unlock()，并且从临界区抛出异常，则Lock将永远保持锁定，导致调用该Lock实例上的lock()的所有线程无限期地停止。