**Anatomy of a Synchronizer**

* [State](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#state)
* [Access Condition](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#accesscondition)
* [State Changes](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#statechanges)
* [Notification Strategy](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#notificationstrategy)
* [Test and Set Method](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#testandset)
* [Set Method](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#set)

即使许多同步器(锁、信号量、阻塞队列等)在功能上不同，它们通常在内部设计上没那么不同。换言之，它们内部由相同(或者近似)的基本部分组成。了解这些基本部分在设计同步器时可以有很大帮助。本文着眼于这些部分。

**注意**：本文内容是IT University of Copenhagen 2004年春M.Sc.学生项目由Jakob Jenkov、Toke Johansen和Lars Bjørn完成的一部分结果。在项目期间我们问了Doug Lea他是否知道相似的工作。有趣的是他在开发Java 5并发工具时已经有了独立于这个项目的近似的结论。Doug Lea的工作，我相信，描述在[**"Java Concurrency in Practice"**](http://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601/ref=pd_bbs_sr_1?ie=UTF8&s=books&qid=1215418711&sr=8-1)书中。这本书也包含一章"Anatomy of a Synchronizer"，内容与本文近似，不过不完全一样。

大部分(如果不是全部)的同步器的目的是从线程的并发访问保卫一些代码区域(临界区间)。为了达成这个，以下的部分在一个同步器中经常是必要的：

1. [**State**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#state)
2. [**Access Condition**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#accesscondition)
3. [**State Changes**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#statechanges)
4. [**Notification Strategy**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#notificationstrategy)
5. [**Test and Set Method**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#testandset)
6. [**Set Method**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#set)

并不是所有的同步器都具有这些部件，而那些具有这些部件的同步器可能拥有的部分没有与这里所描述的完全一样。不过，通常你可以找到其中的一个或多个部分。

**State**

一个同步器的状态被访问条件用于确定是否一个线程能被授权访问。在一个[**Lock**](http://tutorials.jenkov.com/java-concurrency/locks.html)中状态保存在一个boolean中说明Lock是否被锁上。在一个[**Bounded Semaphore**](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded)中内部状态保存在一个计数器(整型)和一个上界(整型)中,说明了“takes”的当前数量和“takes”的最大数量。在一个[**Blocking Queue**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html)中，状态保存在队列中的元素List和最大队列大小（int）成员（如果有的话）中。

这里有两个代码段，分别来自Lock和BoundedSemaphore。状态代码用粗体标记。

public class Lock{

**//state is kept here**

**private boolean isLocked = false;**

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

...

}

public class BoundedSemaphore {

**//state is kept here**

**private int signals = 0;**

**private int bound = 0;**

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

while(this.signals == bound) wait();

this.signal++;

this.notify();

}

...

}

**Access Condition**

访问条件决定了调用“测试和设置状态”方法的线程是否可以设置状态。访问条件通常基于同步器的状态([**state**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#state))。通常在一个while循环中检查访问条件，以防止虚假唤醒([**Spurious Wakeups**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spuriouswakeups))。当访问条件被评估时，它是true或false。

在一个[**Lock**](http://tutorials.jenkov.com/java-concurrency/locks.html)中访问条件只检查isLocked成员变量的值。在一个[**Bounded Semaphore**](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded)中，实际有两个访问条件取决于是否你正试着“take”或“release”信号量。如果一个线程试着获取信号量，signals变量被再次与上界核对。如果一个线程试着释放信号量signals变量被与0核对。

这有两个代码片段关于一个Lock和一个BoundedSemaphore，访问条件被加粗标记。注意条件总是在while循环中被检查。

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

**//access condition**

while(**isLocked**){

wait();

}

isLocked = true;

}

...

}

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

**//access condition**

while(**this.signals == bound**) wait();

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

**//access condition**

while(**this.signals == 0**) wait();

this.signals--;

this.notify();

}

}

**State Changes**

一旦一个线程获取访问临界区间不得不改变同步器状态来(有可能)阻塞其他的线程进入它。换句话说，状态需要反应此事实，一个线程现在正在临界区间内部执行。这应当影响其他线程尝试获取访问的的访问条件。

在一个[**Lock**](http://tutorials.jenkov.com/java-concurrency/locks.html)中状态改变是设置isLocked = true的代码。在一个信号量中，它是或者signal--或者signal++代码；

这是两个代码片段，状态改变加粗标记：

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

**//state change**

**isLocked = true;**

}

public synchronized void unlock(){

**//state change**

**isLocked = false;**

notify();

}

}

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

while(this.signals == bound) wait();

**//state change**

**this.signals++;**

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

**//state change**

**this.signals--;**

this.notify();

}

}

**Notification Strategy**

一旦一个线程改变了一个同步器的状态它可能有时需要提醒其他的等待线程关于状态改变。也许这个状态改变对于其他线程可能转变访问条件为true。

通知策略通常分为三个策略。

1. 通知全部等待线程
2. 通知N个等待线程中的随机一个
3. 通知N个等待线程中的特定一个

通知全部等待线程相当容易。全部等待线程调用wait()在相同对象上。一旦一个线程想要通知等待线程，它在等待线程调用wait()的对象上调用notifyAll()。

通知一个单独的随机等待线程也是相当容易的。只需要让通知线程调用notify()在等待线程已经调用了wait()的对象上。调用notify不保证等待线程中的哪一个将被通知。正如术语“随机等待线程”。

有时候你需要通知一个特定的而不是随机的线程。举例来说，如果你需要保证等待线程被通知以一种特定顺序，比如他们调用同步器的顺序，或者某种其他优先级顺序。为了获取这种效果，要实现这一点，每个等待线程必须自己在单独的对象上调用wait()。当通知的线程想要通知一个特定的等待线程，它将会在这个特定线程已经调用wait()的对象上调用notify()。一个例子可以在[**Starvation and Fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html)文中找到。

下面是一个代码片段，通知策略(通知一个随机等待线程)标粗：

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

**//wait strategy - related to notification strategy**

**wait();**

}

isLocked = true;

}

public synchronized void unlock(){

isLocked = false;

**notify(); //notification strategy**

}

}

**Test and Set Method**

同步器最通常有两种方法，test-and-set是第一种([**set**](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html#set)是另一种)。“测试和设置”意味着调用这个方法的线程**测试**同步器访问条件下的内部状态。如果条件满足，线程**设置**同步器内部状态来反映线程已获取访问。

状态转换通常导致访问条件转为false对于其他尝试获取访问的线程，但可能不总是如此。举例来说，在一个[**Read - Write Lock**](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html)中，一个线程获取读访问将会更新读写锁的状态来反映这个，但其他的请求读访问的线程也将被授权访问，只要没有线程请求了写访问。

测试和设置操作必须原子执行，这意味着不允许在测试和设置方法中测试和设置状态之间执行其他线程。

测试和设置方法的程序流程通常是这样的：

1. 如有必要在测试前设置状态
2. 测试访问条件下的状态
3. 如果访问条件不符合，等待
4. 如果访问条件符合，设置状态，并通知等待线程如有必要

展示在下面的[**ReadWriteLock**](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html)类的lockWrite()方法是一个测试和设置方法的例子。线程调用lockWrite()首先设置状态，在测试(writeRequests++)之前。然后它测试canGrantWriteAccess()方法中的访问条件下内部状态。如果测试成功，内部状态被再次设置在方法退出之前。注意这个方法不通知等待线程。

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

...

**public synchronized void lockWrite() throws InterruptedException{**

**writeRequests++;**

**Thread callingThread = Thread.currentThread();**

**while(! canGrantWriteAccess(callingThread)){**

**wait();**

**}**

**writeRequests--;**

**writeAccesses++;**

**writingThread = callingThread;**

**}**

...

}

下面展示的BoundedSemaphore类有两个测试和设置方法：take()和release()。两个方法都测试和设置内部状态。

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

**public synchronized void take() throws InterruptedException{**

**while(this.signals == bound) wait();**

**this.signals++;**

**this.notify();**

**}**

**public synchronized void release() throws InterruptedException{**

**while(this.signals == 0) wait();**

**this.signals--;**

**this.notify();**

**}**

}

**Set Method**

设置方法是第二种同步器通常包含的方法。设置方法只设置同步器内部状态而不先测试它。设置方法的一个典型的例子是Lock类的unlock()方法。一个线程持有锁总能解锁它而无需测试是否锁是被解锁了的。

设置方法的程序流程如下：

1. 设置内部状态
2. 通知等待线程

这是一个例子关于unlock()方法：

public class Lock{

private boolean isLocked = false;

**public synchronized void unlock(){**

**isLocked = false;**

**notify();**

**}**

}