**Semaphores**

* [Simple Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#simple)
* [Using Semaphores for Signaling](http://tutorials.jenkov.com/java-concurrency/semaphores.html#as-signal)
* [Counting Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#counting)
* [Bounded Semaphore](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded)
* [Using Semaphores as Locks](http://tutorials.jenkov.com/java-concurrency/semaphores.html#aslock)

信号量是一个线程同步结构，可以被用于在线程之间发送信号以避免丢失信号([**missed signals**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missedsignals))或者保卫一个临界区间([**critical section**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html))就如你会用一个锁([**lock**](http://tutorials.jenkov.com/java-concurrency/locks.html))做的。Java 5带来java.util.concurrent包中信号量实现以便你不需要实现自己的信号量。仍旧，了解它们的实现和使用背后的理论很有用。

Java 5带来一个内建信号量，以便你不需要实现你自己的。你可以在我的java.util.concurrent教程的[**java.util.concurrent.Semaphore**](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html)文中读到关于它更多内容。

**Simple Semaphore**

这是一个简单的Semaphore实现：

public class Semaphore {

private boolean signal = false;

public synchronized void take() {

this.signal = true;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(!this.signal) wait();

this.signal = false;

}

}

take()方法发送一个存储在信号量内部的信号。release()方法等待信号。当收到信号时标识再次被清空，release()方法退出。

使用像这样的信号量你可以避免丢失信号。你将会调用take()而不是notify()，release()而不是wait()。如果对take()的调用发生在对release()的调用之前，调用release()的线程仍将会直到take()被调用，因为信号被存储在signal变量内部。wait()和notify()不是这种情况。

名称take()和release()可能看上去有点奇怪，当使用一个信号量为了信令。名称来源于信号量作为锁的用途，正如在这篇文章稍后解释的。那种情况下名称更有意义。

**Using Semaphores for Signaling**

这是一个简化的例子关于两个线程使用Semaphore互相发信号：

Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();

sender.start();

public class SendingThread {

Semaphore semaphore = null;

public SendingThread(Semaphore semaphore){

this.semaphore = semaphore;

}

public void run(){

while(true){

//do something, then signal

this.semaphore.take();

}

}

}

public class RecevingThread {

Semaphore semaphore = null;

public ReceivingThread(Semaphore semaphore){

this.semaphore = semaphore;

}

public void run(){

while(true){

this.semaphore.release();

//receive signal, then do something...

}

}

}

**Counting Semaphore**

之前章节的Semaphore实现不会数通过take()方法调用发送给它的信号数量。我们能修改Semaphore来这样做。这被称作计数信号量。这是一个技术信号量的简单的实现：

public class CountingSemaphore {

private int signals = 0;

public synchronized void take() {

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

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

this.signals--;

}

}

**Bounded Semaphore**

CoutingSemaphore没有上限关于它可以存储多少信号。我们可以改变信号量的实现，使其具有上界，例如：

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();

}

}

注意take()方法现在阻塞如果信号数量等于上界。不到一个线程调用release()，调用take()的线程不会被允许传递它的信号，如果BoundedSemaphore达到了它的信号上限。

**Using Semaphores as Locks**

使用有界信号量作为锁是可能的。为了这么做，设置上界为1，让对take()和release()的调用守护临界区间。这是例子：

BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{

//critical section

} finally {

semaphore.release();

}

与信令用例相反，方法take()和release()现在由同一线程调用。因为只有一个线程被允许获取信号量，全部其他的调用take()的线程将会被阻塞直到release()被调用。对release()的调用将不会阻塞因为总是首先有一个对take()的调用。

你也可以使用有界信号量限制被允许进入一段代码区间的线程数。举例来说，在上面的例子里，如果你设置BoundedSemaphore的极限是5会发生什么？5个线程将会被允许同一时刻进入临界区间。不过你将得确保，线程操作不会对这5个线程发生冲突。否则你的应用将会失败。

release()方法在finally块内部被调用以确保它被调用，即使一个异常从临界区间抛出。