**Thread Signaling**

* [Signaling via Shared Objects](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#shared-objects)
* [Busy Wait](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#busy-wait)
* [wait(), notify() and notifyAll()](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#wait-notify)
* [Missed Signals](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missed-signals)
* [Spurious Wakeups](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spurious-wakeups)
* [Multiple Threads Waiting for the Same Signals](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#multiple-threads)
* [Don't call wait() on constant String's or global objects](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#strings)

线程信令的目的是使线程能够彼此发送信号。此外，线程信令使线程能够等待来自其他线程的信号。例如，线程B可能等待线程A的信号，指示数据已准备好被处理。

**Signaling via Shared Objects**

一种简单的线程发送信号的方法是在一些共享对象变量中设置信号值。线程A可以在同步块内部将布尔成员变量hasDataToProcess设置为true，线程B也可以在同步块内部读取hasDataToProcess成员变量。下面是一个对象的简单例子，它可以保存这样的信号，并提供设置和检查它的方法：

public class MySignal{

protected boolean hasDataToProcess = false;

public synchronized boolean hasDataToProcess(){

return this.hasDataToProcess;

}

public synchronized void setHasDataToProcess(boolean hasData){

this.hasDataToProcess = hasData;

}

}

线程A和B必须有一个共享的MySignal实例的引用，以便信令工作。如果线程A和B引用了不同的MySignal实例，它们将不会检测到彼此的信号。要处理的数据可以位于一个与MySignal实例分开的共享缓冲区中。

**Busy Wait**

处理数据的线程B正在等待数据变得可用以进行处理。换句话说，它正在等待线程A发出的信号，致使hasDataToProcess()返回true。这里是线程B正在运行的循环，同时等待这个信号：

protected MySignal sharedSignal = ...

...

while(!sharedSignal.hasDataToProcess()){

//do nothing... busy waiting

}

注意while循环一直执行直到hasDataToProcess()返回true。这叫做忙等待。线程在等待时占线。

**wait(), notify() and notifyAll()**

忙等待不是运行等待线程的计算机中CPU的非常有效的利用，除非平均等待时间非常短。否则，如果等待线程可以以某种方式睡眠或变得不活动，直到它接收到正在等待的信号，则会更加智能。

Java有一个内建的等待机制，能使得线程在等待信号时变得不活跃。java.lang.Object类定义了三个方法，wait()、notify()和notifyAll()，来促进这个机制。

一个在任一个对象上调用wait()方法的线程变得不活跃，直到另一个线程在那个对象上调用notify()方法。为了调用wait()或者notify()方法，调用线程必须首先在那个对象上占有锁。换句话说，调用线程必须在一个同步代码块内部调用wait()或者notify()。以下是MySignal的一个修改版本，称为MyWaitNotify，使用wait()和notify()。

public class MonitorObject{

}

public class MyWaitNotify{

MonitorObject myMonitorObject = new MonitorObject();

public void doWait(){

synchronized(myMonitorObject){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

}

public void doNotify(){

synchronized(myMonitorObject){

myMonitorObject.notify();

}

}

}

等待线程可以调用doWait()，唤醒线程可以调用doNotify()。当一个线程在一个对象上调用notify()时，在对象上等待的线程之一被唤醒并允许执行。还有一个notifyAll()方法，它将唤醒在给定对象上等待的所有线程。

如你所见，等待和唤醒线程都从同步块中调用wait()和notify()。这是强制性的！线程不能调用wait()、notify()或notifyAll()而不在方法调用的对象上持有锁。如果发生了，会抛出IllegalMonitorStateException异常。

但是，这怎么可能呢？等待线程不会保持对监视器对象(myMonitorObject)的锁定，如果它在同步块中执行吗？等待线程不会阻塞唤醒线程进入doNotify()中的同步块吗？答案是不。一旦线程调用wait()，它就释放它在监视器对象上保存的锁。这允许其他线程调用wait()或notify()，因为这些方法必须从同步块内部调用。

一旦线程被唤醒，它就不能退出wait()调用，直到调用notify()的线程离开它的同步块。换句话说：被唤醒的线程在退出wait()调用之前必须重新获得监视器对象上的锁，因为等待调用嵌套在同步块中。如果使用notifyAll()唤醒多个线程，那么一次只有一个唤醒的线程可以退出wait()方法，因为每个线程必须在退出wait()之前依次获得监视器对象上的锁。

**Missed Signals**

方法notify()和notifyAll()不保存方法调用，以防在调用它们时没有线程在等待。然后通知信号就丢失了。因此，如果在线程调用wait()之前线程调用notify()，则等待线程将错过信号。这个可能是或不是个问题，但是某些情况下这可能导致等待线程永久等待，从未唤醒，因为唤醒信号错过了。

为了避免丢失信号，它们应该存储在信号类内。在MyWaitNotify示例中，通知信号应该存储在MyWaitNotify实例内部的成员变量中。这里是MyWaitNotify的一个修改版本，它是这样做的：

public class MyWaitNotify2{

MonitorObject myMonitorObject = new MonitorObject();

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

if(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

注意，doNotify()方法现在在调用notify()之前将wasSignalled变量设置为true。另外，注意doWait()方法在调用wait()之前检查wasSignalled变量。事实上，它只调用wait()，如果在上一次的doWait()调用和这个中间没有接收到信号。

**Spurious Wakeups**

由于无法解释的原因，即使没有调用notify()和notifyAll()，也有可能唤醒线程。这被称为虚假唤醒。无故醒来。

如果在MyWaitNofity2类的doWait()方法中发生虚假唤醒，则等待线程可以在没有接收到正确信号的情况下继续处理！这可能会导致应用程序出现严重问题。

为了防止虚假唤醒，在while循环中检查信号成员变量，而不是在if语句内部。这样的while循环也被称为自旋锁。线程唤醒自旋，直到自旋锁(while循环)中的条件变为假。下面是MyWaitNotify2的修改版本，它显示：

public class MyWaitNotify3{

MonitorObject myMonitorObject = new MonitorObject();

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

while(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

注意wait()调用是嵌套在while循环中而不是if语句中嵌套的。如果等待线程在没有收到信号的情况下醒来，wasSignalled成员仍然为false，while循环将再次执行，导致唤醒的线程返回等待。

**Multiple Threads Waiting for the Same Signals**

如果您有多个线程在等待，while循环也是一个很好的解决方案，这些线程都使用notifyAll()唤醒，但是应该只允许其中一个线程继续。一次只能有一个线程能够获得监视器对象上的锁，这意味着只有一个线程可以退出wait()调用并清除wasSignalled标志。一旦这个线程退出doWait()方法中的同步块，其他线程就可以退出wait()调用，并在while循环中检查wasSignalled成员变量。但是，第一个线程唤醒清除了这个标志，所以其他被唤醒的线程返回等待，直到下一个信号到达。

**Don't call wait() on constant String's or global objects**

本文较早的版本有一个MyWaitNotify示例类的版本，它使用一个常量字符串( "" )作为监视对象。下面是这个例子的例子：

public class MyWaitNotify{

**String myMonitorObject = "";**

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

while(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

调用空字符串或任何其他常量字符串上的wait()和notify()的问题在于，JVM/Compiler在内部将常量字符串转换为相同的对象。这意味着，即使您有两个不同的MyWaitNotify实例，它们都引用相同的空字符串实例。这也意味着在第一个MyWaitNotify实例上调用doWait()的线程面临被第二个MyWaitNotify实例上的doNotify()调用唤醒的风险。

情势在下面的图表中勾勒出来：

|  |
| --- |
| Calling wait()/notify() on string constants |

记住，即使4个线程在同一个共享字符串实例上调用wait()和notify()，来自doWait()和doNotify()调用的信号也分别存储在两个MyWaitNotify实例中。在MyWaitNotify1上的doNotify()调用可以唤醒在MyWaitNotify2中等待的线程，但是信号将仅存储在MyWaitNotify1中。

起初，这似乎不是一个大问题。毕竟，如果在第二个MyWaitNotify实例上调用doNotify()，那么真正可能发生的事情就是线程A和线程B被错误地唤醒。这个被唤醒的线程（A或B）将在while循环中检查其信号，然后返回等待，因为没有在第一个MyWaitNotify实例上调用doNotify()，而在第一个MyWaitNotify实例中它们正在等待。这种情况等于激起了虚假的觉醒。线程A或B在没有被信号通知的情况下醒来。但是代码可以处理这一点，因此线程返回等待。

问题是，因为doNotify()调用只调用notify()而不调用notifyAll()，所以即使4个线程正在等待相同的字符串实例（空字符串），也只有一个线程被唤醒。因此，如果线程A或B之一在信号为C或D时被唤醒，则被唤醒的线程（A或B）将检查其信号，查看没有收到信号，然后返回等待。C或D都没有醒来检查他们实际接收到的信号，所以信号丢失了。这种情况等于前面描述的丢失信号问题。C和D发出一个信号，但没有响应。

如果doNotify()方法调用notifyAll()而不是notify()，则所有等待的线程都被唤醒，并依次检查信号。线程A和B本来可以返回等待，但是C或D中的任何一个都会注意到该信号并留下doWait()方法调用。C和D的另一个将返回到等待，因为发现信号的线程在doWait()的方式上清除它。

你可能会被诱惑总是调用notifyAll()而不是notify()，但是这是一个坏主意。没有理由唤醒所有线程，当它们中只有一个能够响应信号时。

所以：不要使用全局对象、字符串常量等来处理wait()/notify()机制。使用对于使用它的结构体唯一的对象。例如，每个MyWaitNotify3(来自前面章节的示例)实例都有自己的MonitorObject实例，而不是对wait()/notify()调用使用空字符串。