线程通信

原文链接 **作者：**Jakob Jenkov

译者：杜建雄 **校对：**方腾飞

线程通信的目标是使线程间能够互相发送信号。另一方面，线程通信使线程能够等待其他线程的信号。

例如，线程B可以等待线程A的一个信号，这个信号会通知线程B数据已经准备好了。本文将讲解以下几个JAVA线程间通信的主题：

1、通过共享对象通信

线程间发送信号的一个简单方式是在共享对象的变量里设置信号值。线程A在一个同步块里设置boolean型成员变量hasDataToProcess为true，线程B也在同步块里读取hasDataToProcess这个成员变量。这个简单的例子使用了一个持有信号的对象，并提供了set和check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必须获得指向一个MySignal共享实例的引用，以便进行通信。如果它们持有的引用指向不同的MySingal实例，那么彼此将不能检测到对方的信号。需要处理的数据可以存放在一个共享缓存区里，它和MySignal实例是分开存放的。

2、忙等待(Busy Wait)

准备处理数据的线程B正在等待数据变为可用。换句话说，它在等待线程A的一个信号，这个信号使hasDataToProcess()返回true。线程B运行在一个循环里，以等待这个信号：

```
protected MySignal sharedSignal = ...

...

while(!sharedSignal.hasDataToProcess()){
  //do nothing... busy waiting
}
```

3、wait(),notify()和notifyAll()

忙等待没有对运行等待线程的CPU进行有效的利用，除非平均等待时间非常短。否则，让等待线程进入睡眠或者非运行状态更为明智，直到它接收到它等待的信号。

Java有一个内建的等待机制来允许线程在等待信号的时候变为非运行状态。java.lang.Object 类定义了三个方法，wait()、notify()和notifyAll()来实现这个等待机制。

一个线程一旦调用了任意对象的wait()方法，就会变为非运行状态，直到另一个线程调用了同一个对象的notify()方法。为了调用wait()或者notify()，线程必须先获得那个对象的锁。也就是说，线程必须在同步块里

调用wait()或者notify()。以下是MySingal的修改版本——使用了wait()和notify()的MyWaitNotify：

```
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();
    }
  }
}
```

等待线程将调用doWait()，而唤醒线程将调用doNotify()。当一个线程调用一个对象的notify()方法，正在等待该对象的所有线程中将有一个线程被唤醒并允许执行（校注：这个将被唤醒的线程是随机的，不可以指定唤醒哪个线程）。同时也提供了一个notifyAll()方法来唤醒正在等待一个给定对象的所有线程。

如你所见，不管是等待线程还是唤醒线程都在同步块里调用wait()和notify()。这是强制性的！一个线程如果没有持有对象锁，将不能调用wait()，notify()或者notifyAll()。否则，会抛出IllegalMonitorStateException异常。

（校注：JVM是这么实现的，当你调用wait时候它首先要检查下当前线程是否是锁的拥有者，不是则抛出IllegalMonitorStateExcept，参考JVM源码的1422行。）

但是，这怎么可能？等待线程在同步块里面执行的时候，不是一直持有监视器对象（myMonitor对象）的锁吗？等待线程不能阻塞唤醒线程进入doNotify()的同步块吗？答案是：的确不能。一旦线程调用了wait()方法，它就释放了所持有的监视器对象上的锁。这将允许其他线程也可以调用wait()或者notify()。

一旦一个线程被唤醒，不能立刻就退出wait()的方法调用，直到调用notify()的线程退出了它自己的同步块。换句话说：被唤醒的线程必须重新获得监视器对象的锁，才可以退出wait()的方法调用，因为wait方法调用运行在同步块里面。如果多个线程被notifyAll()唤醒，那么在同一时刻将只有一个线程可以退出wait()方法，因为每个线程在退出wait()前必须获得监视器对象的锁。

4、丢失的信号（Missed Signals）

notify()和notifyAll()方法不会保存调用它们的方法，因为当这两个方法被调用时，有可能没有线程处于等待状态。通知信号过后便丢弃了。因此，如果一个线程先于被通知线程调用wait()前调用了notify()，等待的线程将错过这个信号。这可能是也可能不是个问题。不过，在某些情况下，这可能使等待线程永远在等待，不再醒来，因为线程错过了唤醒信号。

为了避免丢失信号，必须把它们保存在信号类里。在MyWaitNotify的例子中，通知信号应被存储在MyWaitNotify实例的一个成员变量里。以下是MyWaitNotify的修改版本：

```
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();
    }
  }
}
```

留意doNotify()方法在调用notify()前把wasSignalled变量设为true。同时，留意doWait()方法在调用wait()前会检查wasSignalled变量。事实上，如果没有信号在前一次doWait()调用和这次doWait()调用之间的时间段里被接收到，它将只调用wait()。

（校注：为了避免信号丢失，用一个变量来保存是否被通知过。在notify前，设置自己已经被通知过。在wait后，设置自己没有被通知过，需要等待通知。）

5、假唤醒

由于莫名其妙的原因，线程有可能在没有调用过notify()和notifyAll()的情况下醒来。这就是所谓的假唤醒（spurious wakeups）。无端端地醒过来了。

如果在MyWaitNotify2的doWait()方法里发生了假唤醒，等待线程即使没有收到正确的信号，也能够执行后续的操作。这可能导致你的应用程序出现严重问题。

为了防止假唤醒，保存信号的成员变量将在一个while循环里接受检查，而不是在if表达式里。这样的一个while循环叫做自旋锁*（校注：这种做法要慎重，目前的JVM实现自旋会消耗CPU，如果长时间不调用doNotify方法，doWait方法会一直自旋，CPU会消耗太大）*。被唤醒的线程会自旋直到自旋锁(while循环)里的条件变为false。以下MyWaitNotify2的修改版本展示了这点：

```
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();
    }
  }
}
```

留意wait()方法是在while循环里，而不在if表达式里。如果等待线程没有收到信号就唤醒，wasSignalled变量将变为false,while循环会再执行一次，促使醒来的线程回到等待状态。

6、多个线程等待相同信号

如果你有多个线程在等待，被notifyAll()唤醒，但只有一个被允许继续执行，使用while循环也是个好方法。每次只有一个线程可以获得监视器对象锁，意味着只有一个线程可以退出wait()调用并清除wasSignalled标志（设为false）。一旦这个线程退出doWait()的同步块，其他线程退出wait()调用，并在while循环里检查wasSignalled变量值。但是，这个标志已经被第一个唤醒的线程清除了，所以其余醒来的线程将回到等待状态，直到下次信号到来。

7、不要在字符串常量或全局对象中调用wait()

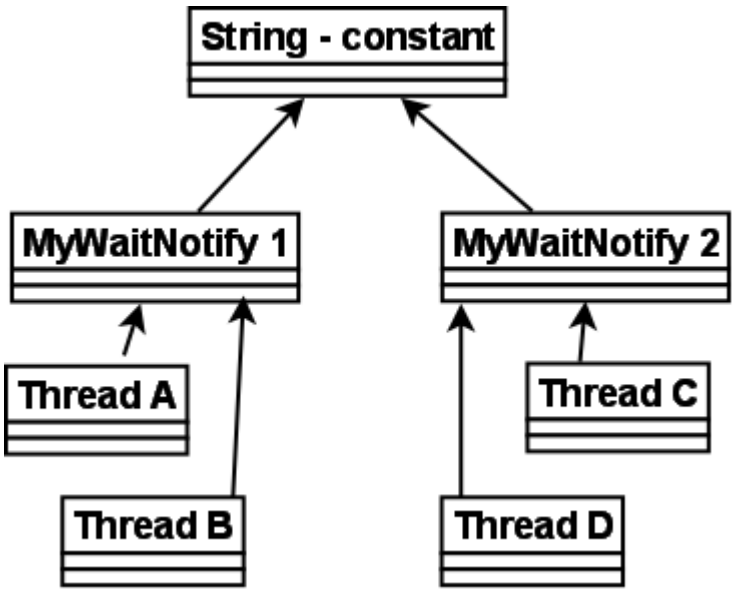
（校注：本章说的字符串常量指的是值为常量的变量）

本文早期的一个版本在MyWaitNotify例子里使用字符串常量（""）作为管程对象。以下是那个例子：

```
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();
    }
  }
}
```

在空字符串作为锁的同步块(或者其他常量字符串)里调用wait()和notify()产生的问题是，JVM/编译器内部会把常量字符串转换成同一个对象。这意味着，即使你有2个不同的MyWaitNotify实例，它们都引用了相同的空字符串实例。同时也意味着存在这样的风险：在第一个MyWaitNotify实例上调用doWait()的线程会被在第二个MyWaitNotify实例上调用doNotify()的线程唤醒。这种情况可以画成以下这张图：



起初这可能不像个大问题。毕竟，如果doNotify()在第二个MyWaitNotify实例上被调用，真正发生的事不外乎线程A和B被错误的唤醒了 。这个被唤醒的线程（A或者B）将在while循环里检查信号值，然后回到等待状态，因为doNotify()并没有在第一个MyWaitNotify实例上调用，而这个正是它要等待的实例。这种情况相当于引发了一次假唤醒。线程A或者B在信号值没有更新的情况下唤醒。但是代码处理了这种情况，所以线程回到了等待状态。记住，即使4个线程在相同的共享字符串实例上调用wait()和notify()，doWait()和doNotify()里的信号还会被2个MyWaitNotify实例分别保存。在MyWaitNotify1上的一次doNotify()调用可能唤醒MyWaitNotify2的线程，但是信号值只会保存在MyWaitNotify1里。

问题在于，由于doNotify()仅调用了notify()而不是notifyAll()，即使有4个线程在相同的字符串（空字符串）实例上等待，只能有一个线程被唤醒。所以，如果线程A或B被发给C或D的信号唤醒，它会检查自己的信号值，看看有没有信号被接收到，然后回到等待状态。而C和D都没被唤醒来检查它们实际上接收到的信号值，这样信号便丢失了。这种情况相当于前面所说的丢失信号的问题。C和D被发送过信号，只是都不能对信号作出回应。

如果doNotify()方法调用notifyAll()，而非notify()，所有等待线程都会被唤醒并依次检查信号值。线程A和B将回到等待状态，但是C或D只有一个线程注意到信号，并退出doWait()方法调用。C或D中的另一个将回到等待状态，因为获得信号的线程在退出doWait()的过程中清除了信号值(置为false)。

看过上面这段后，你可能会设法使用notifyAll()来代替notify()，但是这在性能上是个坏主意。在只有一个线程能对信号进行响应的情况下，没有理由每次都去唤醒所有线程。

所以：**在wait()/notify()机制中，不要使用全局对象，字符串常量等。应该使用对应唯一的对象。**例如，每一个MyWaitNotify3的实例（前一节的例子）拥有一个属于自己的监视器对象，而不是在空字符串上调用

wait()/notify()。

校注：

管程 (英语：Monitors，也称为监视器) 是对多个工作线程实现互斥访问共享资源的对象或模块。这些共享资源一般是硬件设备或一群变量。管程实现了在一个时间点，最多只有一个线程在执行它的某个子程序。与那些通过修改数据结构实现互斥访问的并发程序设计相比，管程很大程度上简化了程序设计。

Snway
2013/03/14 11:23上午

弥补了平常容易忽视的知识点，感谢作者！

jTux
2013/03/14 4:21下午

有个typo, 倒数第5行:

而不是在空字符串上调用wiat()/notify()

wiat应为wait

丁一
2013/03/14 6:20下午

多谢提醒，已更正。

方腾飞
2013/03/14 6:21下午

已修正，谢谢指正。

夕水溪下
2013/03/14 5:28下午

以一个例子来串烧下知识点：

上学的时候我们经常会去图书馆借书，这个我印象比较深刻(本来想举买火车票的例子)。图书馆里，有一本书叫《JAVA并发编程实战》，小A早上的时候把这本书借走了，小B中午的时候去图书馆找这本书，这里小A和小B分别是两个线程，他们都要看的书

是共享资源。

1.通过共享资源通信

小B去了图书馆，发现这本书被借走了(执行了例子中的hasDataToProcess)。他回到家，等了几天，再去图书馆找这本书，发现这本书已经被还回，他顺利借走了书。

2.忙等待

其实小B在小A走后一会就把书还回去了，小A却在几天后才去找的书。为了早点借到书(减少延迟)，小A可能就在图书馆等着，每隔几分钟(while循环)他就去检查这本书有没有被还回，这样只要小B一还回书，小A马上就会知道。

3.wait(),notify()和notifyAll()

很多次后，小A发现自己这样做太累了，身体有点吃不消。不过很快，学校图书馆系统改进，加入了短信通知功能(notify())，只要小B一还回书，立马会短信通知小A，这样小A就可以在家睡觉等短信了。

4.丢失的信号

学校图书馆系统是这么设计的：当一本书被还回来的时候，会给等待者发送短信，并且只会发一次，如果没有等待者，他也会发(只不过没有接收者)。问题出现了，因为短信只会发一次，当书被还回来的时候，没有人等待借书，他会发一条空短信，但是之后有等待借此本书的同学永远也不会再收到短信，导致这些同学会无休止的等待。为了避免这个问题，我们在等待的时候先打个电话问问图书馆管理员是否继续等待(if(!wasSignalled))。

5.假唤醒

图书馆系统还有一个BUG：系统会偶尔给你发条错误短信，说书可以借了(其实书不可以借)。我们之前已经给图书馆管理员打过电话了，他说让我们等短信。我们很听话，一等到短信(其实是bug引起的错误短信)，就去借书了，到了图书馆后发现这书根本就没还回来！我们很郁闷，但也没办法啊，学校不修复BUG，我们得聪明点：每次在收到短信后，再打电话问问书到底能不能借(while(!wasSignalled))。

方 腾飞
2013/03/14 6:31下午

很生动的例子，这种学习风格会让知识记得更牢固。

Yole
2013/04/30 12:47上午

关于信号丢失，我好像没理解清楚。我的理解是，对于某一个资源，从时间线上来看，程序先notify(),然后才有线程wait()，但是收不到之前notify()的消息了。

如果我对这个概念理解是正确的，那我觉得信号丢失是不是一种逻辑上的错误呢。拿图书馆的这个例子来说，当A同学还书，并且发出空白短信后，那么下一个B同学，就可以直接借到书了；如果继续有C同学想借，那么C等待，当B还书后，再次发一个短信不就不会存在这个问题了吗？

从程序上来看，notify（ ）这个操作应该满足一定条件然后被调用。出现先notify，再wait，后面不再notify，这种情况，

是不是在逻辑上就应该避免这种情况的发生？在需要的时候，可以尝试再次的notify()来保证在wait的线程能收到消息呢？

不知道是不是理解有偏差，望解答。

watermellon
2016/09/29 5:10下午

很生动的例子,其中第2点的忙等待小A小B是不是顺序反了?

还有就是第5点假唤醒,能不能这样理解:图书管其实这本书可以借了,图书管发短信通知可以借书的这个时间段,书刚好又被借走了,所以造成了收到短信来借书的同学发现书还没有回来.

wh
2018/08/04 5:44下午

示例生动形象，理解起来十分容易。也更容易记在脑中。

BlueIceQ
2013/03/14 10:02下午

好文一定要顶！

bazhen.csy
2013/04/21 9:23下午

为了防止假唤醒，保存信号的成员变量将在一个while循环里接受检查，而不是在if表达式里。这样的一个while循环叫做自旋锁（校注：这种做法要慎重，目前的JVM实现自旋会消耗CPU，如果长时间不调用doNotify方法，doWait方法会一直自旋，CPU会消耗太大）。

我的理解:

自旋的核心是cpu空转循环check，但是这里其实没有所谓的空转check机制，仅仅是假唤醒后为了保险落在一个循环中保证多次假唤醒也能够正确检测到。这里的自旋是不是存在误用。

fangin
2014/06/18 2:52下午

我也觉得这不叫自旋锁，自旋锁应该是由硬件支持的一个cpu不断check高速缓存的操作，目的是为了减少线程切换到内核态。

单纯用java写不出来吧。。。。

方 腾飞
2014/06/21 1:42下午

你这么理解也对。从Java这个角度理解Java的自旋锁也对。

音无麻里亚
2016/11/25 11:06上午

同意，这个我一开始看的时候也是懵逼，然后自己测试了一下。就算在while循环里面，wait之后也不会反复检测（菜鸟用system.out简单打印只输出一次），所以这里称为自旋锁容易让人误会

匿名
2013/04/23 6:03下午

“如果在MyWaitNotify2的doWait()方法里发生了假唤醒，等待线程即使没有收到正确的信号，也能够执行后续的操作。这可能导致你的应用程序出现严重问题”

这个表述有些歧义吧——即使等待的线程在没有收到信号的情况下自己醒来，它也需要得到锁之后才能继续执行。那么持有锁的线程此时如果没有释放，醒来的线程还会继续等待；如果刚好锁被释放，这个醒来的线程才有可能得到锁（也许还有其它线程在此锁上等待）并继续执行。

丁一
2013/04/28 10:07下午

表述是没有问题的。MyWaitNotify2中对条件变量的判断并不在一个循环中，发生诸如虚假唤醒之后，虽然可能当时不一定能获得锁，但最终一定会获得锁而继续执行下去，这才是问题根本所在。而将对条件的判断放到循环中，获得锁之后还会对条件在持有锁的情形下做一次测试，这就能保证接下来继续执行时，条件是一直满足的。

快下班了
2013/05/10 5:22下午

没用过JAVA，菜鸟发问：

1.为什么wait前要获得锁？而且还是要一个对象锁，不能是同步方法吗？

2. public void doNotify(){

```
synchronized(myMonitorObject){
20
wasSignalled = true;
21
myMonitorObject.notify();
22
}
23
}
24
}
```

留意doNotify()方法在调用notify()前把wasSignalled变量设为true。这里有什么深意？

丁一
2013/05/14 7:01上午

在什么对象上调用wait/notify，就要获取该对象的锁，否则会抛出IllegalMonitorStateException。

后面那句没有什么深意，就是要注意下wasSignalled在doWait和doNotify中的访问顺序。

劳希
2013/07/06 12:01上午

可能有个单词打错了,第四点的倒数第二行,notify写成了nodify

方 腾飞
2013/07/07 4:10下午

感谢校对，我已经修正了。

小浣熊
2013/07/08 10:59下午

假唤醒是在老版本的java中才有的情况还是在所有版本中都会有这种情况？

xiaoli
2014/01/03 5:54下午

关于第五点假唤醒，个人感觉有问题

“（校注：这种做法要慎重，目前的JVM实现自旋会消耗CPU，如果长时间不调用doNotify方法，doWait方法会一直自旋，CPU会消耗太大）。被唤醒的线程会自旋直到自旋锁(while循环)里的条件变为false”

线程被异常唤醒，如果

1. wasSignalled = true，则会继续执行可能会产生文中提到的严重bug

2. wasSignalled = false, 则立即进入wait等待

这是两个比较清晰的分支，不明白为什么会有 “CPU会消耗太大”的说法。

大家能否帮忙解释下，或者是我理解的context中有缺失，导致我理解错误。请提醒

谢了

宿城
2014/03/16 9:15下午

我也不是很明白，求解释

study
2014/05/19 3:00下午

1.wasSignalled =true 的时候不会继续执行,这时候while条件不满足，代码不会执行到后面。

会在while这一句无限循环。

java_coder
2014/07/11 11:41上午

while条件不满足的时候怎么不会执行到后面呢

AnyStretch
2014/08/15 3:33下午

被假唤醒的时候，wasSignalled并不会置为true，因此不会退出while循环。

dongzh
2014/09/28 11:57上午

wasSignalled为false，那直接又会执行wait，线程进入等待，为什么会有 “CPU会消耗太大”的说法？假唤醒这条表述的有问题吧

luoyuyou
2014/05/19 9:32下午

一：

假唤醒（spurious wakeups）：

由于莫名其妙的原因，线程有可能在没有调用过notify()和notifyAll()的情况下醒来（For inexplicable reasons it is possible for threads to wake up even if notify() and notifyAll() has not been called）。

我认为这个假唤醒是可以接受的，但是下面的：

如果在MyWaitNotify2的doWait()方法里发生了假唤醒，等待线程即使没有收到正确的信号，也能够执行后续的操作。

（If a spurious wakeup occurs in the MyWaitNofity2 class's doWait() method the waiting thread may continue processing without having received a proper signal to do so!）。

我个人认为是错误的，因为已经调用wait的线程事实上进入了“等待”状态，这与得不到锁的“阻塞”状态是有区别的。处于“等待”状态的线程即使假唤醒也不会尝试去获取锁，自然也不可能执行后面的操作（事实上这里的假唤醒并不是while循环出现的原因），除非接受了notify信号。

显式锁中是这么处理的，内置锁不可能存在差别吧？

二：

此处添加while循环的真正原因是为了应对等待线程重新获取锁之后，发现诱使自己醒来的那个条件（wasSignalled=true），已经被之前的某个获得锁的线程消费掉了（变回false）。

xuxx09
2016/08/24 2:26下午

赞同这个观点

zxpbenson
2014/08/12 1:27下午

第五部分 5、假唤醒

```
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();
}
}
}
```

我怎么觉得这个一旦有哪个线程调用doWait()方法以后，这个线程就把myMonitorObject这个对象锁锁住，然后在自旋里永远也出不来了吧，别的线程调用doNotify()也永远拿不到myMonitorObject，永远等待，死锁了吧。。。

zxpbenson
2014/08/12 7:36下午

哦 懵了 wait操作会释放掉myMonitorObject上的锁

AnyStretch
2014/08/15 3:36下午

很好的文章，让人茅塞顿开。

AllenZhou
2015/04/08 3:30下午

请教个问题,这个假唤醒是什么原因导致的?跟操作系统 API 有关?

麦芽糖

2016/04/12 11:20上午

好文章 顶起！！！！！！！！！！！！！！

黄 文海
2016/04/12 4:45下午

“Spurious wakeup”翻译成“欺骗性唤醒”可能跟恰当一些。这种“唤醒”使得等待线程被唤醒是事实，问题在于等待线程所需要满足的条件根本就没有变化（因此还是不成立）。所以说这种唤醒是一种“欺骗”。就像人们撒谎一样，撒谎的人按照双方事先的协议（这个协议就是等待——通知）在特定情况下他需要为你（等待线程）做一件事情（更新共享状态变量）并通知你，然而他没有为你做这件事却通知你他完成了，其实他根本没有做，这就是在欺骗你。欺骗性唤醒这个问题的根源在不在JVM自身，而在底层平台。这点Condition接口的javadoc有说明：

When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics.

可见，文档中用了“妥协”（concession）一词。也就是说，JVM允许欺骗性唤醒的存在是其与底层平台妥协的结果。当然，我们只要保证在始终将wait方法调用放在一个循环之中，这个问题就不会对我们产生影响。

tian251yu
2016/08/23 11:46上午

你好，丢失的信号代码第八行，if(!wasSignalled)是否应该是if(wasSignalled)

walter
2017/01/13 12:04下午

最后一个全局对象是什么意思

zhili
2018/06/08 2:04下午

原文 : the wasSignalled member will still be false.

译文 : wasSignalled变量将变为false

应改为 : wasSignalled变量仍然为false