java线程中含有waiting与blocked两种状态，两种状态有着明显的区别，本文从jdk源码中两种状态的设计角度和状态进入角度来分析两者的区别。

**源码分析**

/\*

\* Thread state for a thread blocked waiting for a monitor lock.

\* A thread in the blocked state is waiting for a monitor lock

\* to enter a synchronized block/method or

\* reenter a synchronized block/method after calling

\* {@link Object#wait() Object.wait}.

\*/

BLOCKED,

处于 blocked状态的线程**等待获取监视器锁以期进入同步代码块/方法中**，文档中的在调用Object.wait()方法后重入同步代码块/方法不是很理解。

/\*\*

\* Thread state for a waiting thread.

\* A thread is in the waiting state due to calling one of the

\* following methods:

\* <ul>

\* <li>{@link Object#wait() Object.wait} with no timeout</li>

\* <li>{@link #join() Thread.join} with no timeout</li>

\* <li>{@link LockSupport#park() LockSupport.park}</li>

\* </ul>

\*

\* <p>A thread in the waiting state is waiting for another thread to

\* perform a particular action.

\*

\* For example, a thread that has called <tt>Object.wait()</tt>

\* on an object is waiting for another thread to call

\* <tt>Object.notify()</tt> or <tt>Object.notifyAll()</tt> on

\* that object. A thread that has called <tt>Thread.join()</tt>

\* is waiting for a specified thread to terminate.

\*/

WAITING,

文档介绍了导致线程进入Wating状态的原因以及如何退出Wating状态。

**区别**

线程可以通过notify,join,LockSupport.park方式进入wating状态，**进入wating状态的线程等待唤醒(notify或notifyAll)才有机会获取cpu的时间片段来继续执行**。

**线程的 blocked状态往往是无法进入同步方法/代码块来完成的**。这是因为无法获取到与同步方法/代码块相关联的锁。

与wating状态相关联的是**等待队列**，与blocked状态相关的是**同步队列**，一个线程由等待队列迁移到同步队列时，线程状态将会由wating转化为blocked。**可以这样说，blocked状态是处于wating状态的线程重新焕发生命力的必由之路。**

作者：high\_m  
链接：https://www.jianshu.com/p/da90ab7bc79b  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# ava线程状态中BLOCKED和WAITING有什么区别？

原创 2015年08月23日 17:48:40

* 标签：
* [Java](http://so.csdn.net/so/search/s.do?q=Java&t=blog) /
* [Thread State](http://so.csdn.net/so/search/s.do?q=Thread%20State&t=blog)
* 8100

刚才在看CSDN的问答时，发现这个问题。原问题的作者是在观察jstack的输出时提出的疑问，那么BLOCKED和WAITING有什么区别呢？  
答复在JDK源码中可以找到，如下是java.lang.Thread.State类的一部分注释。

/\*\*  
\* Thread state for a thread blocked waiting for a monitor lock.  
\* A thread in the blocked state is waiting for a monitor lock  
\* to enter a synchronized block/method or  
\* reenter a synchronized block/method after calling  
\* {@link Object#wait() Object.wait}.  
\*/  
BLOCKED,

/\*\*  
\* Thread state for a waiting thread.  
\* A thread is in the waiting state due to calling one of the  
\* following methods:  
\* {@link Object#wait() Object.wait} with no timeout  
\* {@link #join() Thread.join} with no timeout  
\* {@link LockSupport#park() LockSupport.park}  
\*  
\*  
\* A thread in the waiting state is waiting for another thread to  
\* perform a particular action.  
\*  
\* For example, a thread that has called Object.wait()  
\* on an object is waiting for another thread to call  
\* Object.notify() or Object.notifyAll() on  
\* that object. A thread that has called Thread.join()  
\* is waiting for a specified thread to terminate.  
\*/  
WAITING,

从中可以清晰的得到线程处于BLOCKED和WAITING状态的场景。

## BLOCKED状态

线程处于BLOCKED状态的场景。

* 当前线程在等待一个monitor lock，比如等待执行synchronized代码块或者使用synchronized标记的方法。
* 在synchronized块中循环调用Object类型的wait方法，如下是样例  
  synchronized(this)  
  {  
  while (flag)  
  {  
  obj.wait();  
  }  
  // some other code  
  }

## WAITING状态

线程处于WAITING状态的场景。

* 调用Object对象的wait方法，但没有指定超时值。
* 调用Thread对象的join方法，但没有指定超时值。
* 调用LockSupport对象的park方法。

提到WAITING状态，顺便提一下TIMED\_WAITING状态的场景。

## TIMED\_WAITING状态

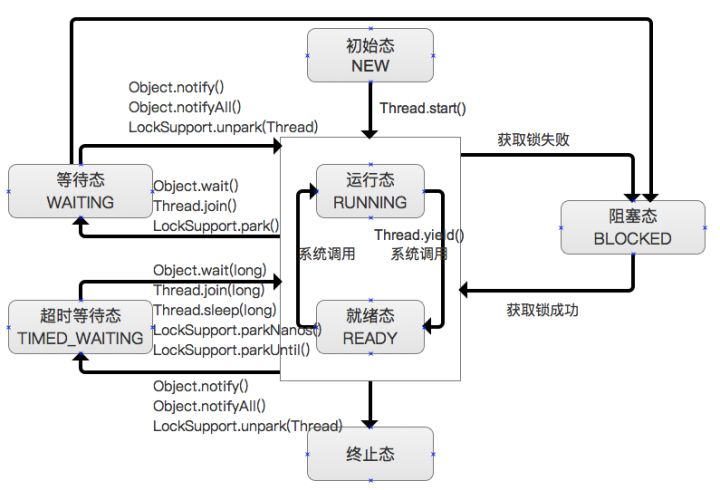
线程处于TIMED\_WAITING状态的场景。

* 调用Thread.sleep方法。
* 调用Object对象的wait方法，指定超时值。
* 调用Thread对象的join方法，指定超时值。
* 调用LockSupport对象的parkNanos方法。
* 调用LockSupport对象的parkUntil方法。

欢迎访问Jackie的家，http://jackieathome.sinaapp.com/，如需转载文章，请注明出处。

作者：大闲人柴毛毛  
链接：https://www.zhihu.com/question/27654579/answer/252912242  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

先来一张线程状态转化图，然后我再慢慢解释：

<img src="https://pic1.zhimg.com/50/v2-6fdc939e35ec801c31ea515e80a88367\_hd.jpg" data-caption="" data-rawwidth="795" data-rawheight="541" class="origin\_image zh-lightbox-thumb" width="795" data-original="https://pic1.zhimg.com/v2-6fdc939e35ec801c31ea515e80a88367\_r.jpg">

在Java中线程的状态一共被分成6种：

**初始态：NEW**

创建一个Thread对象，但还未调用start()启动线程时，线程处于初始态。

**运行态：RUNNABLE**

在Java中，运行态包括就绪态 和 运行态。

* 就绪态
  + 该状态下的线程已经获得执行所需的所有资源，只要CPU分配执行权就能运行。
  + 所有就绪态的线程存放在就绪队列中。
* 运行态
  + 获得CPU执行权，正在执行的线程。
  + 由于一个CPU同一时刻只能执行一条线程，因此每个CPU每个时刻只有一条运行态的线程。

**阻塞态**

* 当一条正在执行的线程请求某一资源失败时，就会进入阻塞态。
* 而在Java中，阻塞态专指请求锁失败时进入的状态。
* 由一个阻塞队列存放所有阻塞态的线程。
* 处于阻塞态的线程会不断请求资源，一旦请求成功，就会进入就绪队列，等待执行。

PS：锁、IO、Socket等都资源。

**等待态**

* 当前线程中调用wait、join、park函数时，当前线程就会进入等待态。
* 也有一个等待队列存放所有等待态的线程。
* 线程处于等待态表示它需要等待其他线程的指示才能继续运行。
* 进入等待态的线程会释放CPU执行权，并释放资源（如：锁）

**超时等待态**

* 当运行中的线程调用sleep(time)、wait、join、parkNanos、parkUntil时，就会进入该状态；
* 它和等待态一样，并不是因为请求不到资源，而是主动进入，并且进入后需要其他线程唤醒；
* 进入该状态后释放CPU执行权 和 占有的资源。
* **与等待态的区别：**到了超时时间后自动进入阻塞队列，开始竞争锁。

**终止态**

线程执行结束后的状态。

**注意：**

* wait()方法会释放CPU执行权 和 占有的锁。
* sleep(long)方法仅释放CPU使用权，锁仍然占用；线程被放入超时等待队列，与yield相比，它会使线程较长时间得不到运行。
* yield()方法仅释放CPU执行权，锁仍然占用，线程会被放入就绪队列，会在短时间内再次执行。
* wait和notify必须配套使用，即必须使用同一把锁调用；
* wait和notify必须放在一个同步块中
* 调用wait和notify的对象必须是他们所处同步块的锁对象。

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](http://ifeve.com/thread-signaling/#shared_object)

2、[忙等待](http://ifeve.com/thread-signaling/#busy_wait)

3、[wait(),notify()和notifyAll()](http://ifeve.com/thread-signaling/#wait_notify)

4、[丢失的信号](http://ifeve.com/thread-signaling/#missed_signal)

5、[假唤醒](http://ifeve.com/thread-signaling/#spurious_wakeups)

6、[多线程等待相同信号](http://ifeve.com/thread-signaling/#same_signals)

7、[不要对常量字符串或全局对象调用wait()](http://ifeve.com/thread-signaling/#global_wait)

## 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:

|  |  |  |
| --- | --- | --- |
| 01 | public class MySignal{ | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | protected boolean hasDataToProcess = false; | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | public synchronized boolean hasDataToProcess(){ | |
| 06 | return this.hasDataToProcess; |

|  |  |  |
| --- | --- | --- |
| 07 | } | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | public synchronized void setHasDataToProcess(boolean hasData){ | |
| 10 | this.hasDataToProcess = hasData; |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 |  |

|  |  |
| --- | --- |
| 13 | } |

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：

|  |  |  |
| --- | --- | --- |
| 1 | protected MySignal sharedSignal = ... | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | ... | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | while(!sharedSignal.hasDataToProcess()){ | |
| 6 | //do nothing... busy waiting |

|  |  |
| --- | --- |
| 7 | } |

## 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：

|  |  |  |
| --- | --- | --- |
| 01 | public class MonitorObject{ | |
| 02 | } |

|  |  |
| --- | --- |
| 03 |  |
| 04 | public class MyWaitNotify{ | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | MonitorObject myMonitorObject = new MonitorObject(); | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | public void doWait(){ | |

|  |  |  |
| --- | --- | --- |
| 09 | synchronized(myMonitorObject){ | |
| 10 | try{ |

|  |  |
| --- | --- |
| 11 | myMonitorObject.wait(); |
| 12 | } catch(InterruptedException e){...} | |

|  |  |  |
| --- | --- | --- |
| 13 | } | |
| 14 | } |

|  |  |
| --- | --- |
| 15 |  |
| 16 | public void doNotify(){ | |

|  |  |  |
| --- | --- | --- |
| 17 | synchronized(myMonitorObject){ | |
| 18 | myMonitorObject.notify(); |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 | } |

|  |  |
| --- | --- |
| 21 | } |

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*](http://hg.openjdk.java.net/hsx/hotspot-main/hotspot/file/61b82be3b1ff/src/share/vm/runtime/objectMonitor.cpp)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：

|  |  |  |
| --- | --- | --- |
| 01 | public class MyWaitNotify2{ | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | MonitorObject myMonitorObject = new MonitorObject(); | |
| 04 | boolean wasSignalled = false; |

|  |  |
| --- | --- |
| 05 |  |
| 06 | public void doWait(){ | |

|  |  |  |
| --- | --- | --- |
| 07 | synchronized(myMonitorObject){ | |
| 08 | if(!wasSignalled){ |

|  |  |
| --- | --- |
| 09 | try{ |
| 10 | myMonitorObject.wait(); | |

|  |  |  |
| --- | --- | --- |
| 11 | } catch(InterruptedException e){...} | |
| 12 | } |

|  |  |  |
| --- | --- | --- |
| 13 | //clear signal and continue running. | |
| 14 | wasSignalled = false; |

|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |
| 18 | public void doNotify(){ | |

|  |  |  |
| --- | --- | --- |
| 19 | 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。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：

|  |  |  |
| --- | --- | --- |
| 01 | public class MyWaitNotify3{ | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | MonitorObject myMonitorObject = new MonitorObject(); | |
| 04 | boolean wasSignalled = false; |

|  |  |
| --- | --- |
| 05 |  |
| 06 | public void doWait(){ | |

|  |  |  |
| --- | --- | --- |
| 07 | synchronized(myMonitorObject){ | |
| 08 | while(!wasSignalled){ |

|  |  |
| --- | --- |
| 09 | try{ |
| 10 | myMonitorObject.wait(); | |

|  |  |  |
| --- | --- | --- |
| 11 | } catch(InterruptedException e){...} | |
| 12 | } |

|  |  |  |
| --- | --- | --- |
| 13 | //clear signal and continue running. | |
| 14 | wasSignalled = false; |

|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |
| 18 | public void doNotify(){ | |

|  |  |  |
| --- | --- | --- |
| 19 | synchronized(myMonitorObject){ | |
| 20 | wasSignalled = true; |

|  |  |  |
| --- | --- | --- |
| 21 | myMonitorObject.notify(); | |
| 22 | } |

|  |  |  |
| --- | --- | --- |
| 23 | } | |
| 24 | } |

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：

|  |  |  |
| --- | --- | --- |
| 01 | public class MyWaitNotify{ | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | String myMonitorObject = ""; |
| 04 | boolean wasSignalled = false; | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | public void doWait(){ | |

|  |  |  |
| --- | --- | --- |
| 07 | synchronized(myMonitorObject){ | |
| 08 | while(!wasSignalled){ |

|  |  |
| --- | --- |
| 09 | try{ |
| 10 | myMonitorObject.wait(); | |

|  |  |  |
| --- | --- | --- |
| 11 | } catch(InterruptedException e){...} | |
| 12 | } |

|  |  |  |
| --- | --- | --- |
| 13 | //clear signal and continue running. | |
| 14 | wasSignalled = false; |

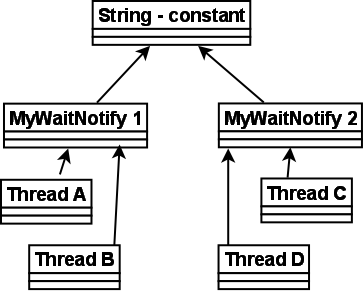
|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |
| 18 | public void doNotify(){ | |

|  |  |  |
| --- | --- | --- |
| 19 | synchronized(myMonitorObject){ | |
| 20 | wasSignalled = true; |

|  |  |  |
| --- | --- | --- |
| 21 | myMonitorObject.notify(); | |
| 22 | } |

|  |  |  |
| --- | --- | --- |
| 23 | } | |
| 24 | } |

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：  
[](http://ifeve.com/thread-signaling/strings-wait-notify/)  
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](http://ifeve.com/thread-signaling/#MyWaitNotify3)）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。