Slipped Conditions

原文链接 **作者：**Jakob Jenkov **译者：**余绍亮 **校对：**丁一

所谓Slipped conditions，就是说， 从一个线程检查某一特定条件到该线程操作此条件期间，这个条件已经被其它线程改变，导致第一个线程在该条件上执行了错误的操作。这里有一个简单的例子：

```
01 public class Lock {
02     private boolean isLocked = true;
03
04     public void lock(){
05       synchronized(this){
06         while(isLocked){
07           try{
08             this.wait();
09           } catch(InterruptedException e){
10             //do nothing, keep waiting
11           }
12         }
13       }
14
15       synchronized(this){
16         isLocked = true;
17       }
18     }
19
20     public synchronized void unlock(){
21       isLocked = false;
22       this.notify();
23     }
24 }
```

我们可以看到，lock()方法包含了两个同步块。第一个同步块执行wait操作直到isLocked变为false才退出，第二个同步块将isLocked置为true，以此来锁住这个Lock实例避免其它线程通过lock()方法。

我们可以设想一下，假如在某个时刻isLocked为false， 这个时候，有两个线程同时访问lock方法。如果第一个线程先进入第一个同步块，这个时候它会发现isLocked为false，若此时允许第二个线程执行，它也进入第一个同步块，同样发现isLocked是false。现在两个线程都检查了这个条件为false，然后它们都会继续进入第二个同步块中并设置isLocked为true。

这个场景就是slipped conditions的例子，两个线程检查同一个条件， 然后退出同步块，因此在这两个线程改变条件之前，就允许其它线程来检查这个条件。换句话说，条件被某个线程检查到该条件被此线程改变期间，这个条件已经被其它线程改变过了。

为避免slipped conditions，条件的检查与设置必须是原子的，也就是说，在第一个线程检查和设置条件期间，不会有其它线程检查这个条件。

解决上面问题的方法很简单，只是简单的把isLocked = true这行代码移到第一个同步块中，放在while循环后面即可：

```
01 public class Lock {
02     private boolean isLocked = true;
03
04     public void lock(){
05       synchronized(this){
06         while(isLocked){
07           try{
08             this.wait();
09           } catch(InterruptedException e){
10             //do nothing, keep waiting
11           }
12         }
13         isLocked = true;
14       }
15     }
16
17     public synchronized void unlock(){
18       isLocked = false;
19       this.notify();
20     }
21 }
```

现在检查和设置isLocked条件是在同一个同步块中原子地执行了。

一个更现实的例子

也许你会说，我才不可能写这么挫的代码，还觉得slipped conditions是个相当理论的问题。但是第一个简单的例子只是用来更好的展示slipped conditions。

饥饿和公平中实现的公平锁也许是个更现实的例子。再看下嵌套管程锁死中那个幼稚的实现，如果我们试图解决其中的嵌套管程锁死问题，很容易产生slipped conditions问题。 首先让我们看下嵌套管程锁死中的例子：

```
01 //Fair Lock implementation with nested monitor lockout problem
02 public class FairLock {
03   private boolean isLocked = false;
04   private Thread lockingThread = null;
05   private List waitingThreads =
06             new ArrayList();
07
08   public void lock() throws InterruptedException{
09     QueueObject queueObject = new QueueObject();
10
11     synchronized(this){
12       waitingThreads.add(queueObject);
13
14       while(isLocked || waitingThreads.get(0) != queueObject){
15
16         synchronized(queueObject){
17           try{
18             queueObject.wait();
19           }catch(InterruptedException e){
20             waitingThreads.remove(queueObject);
21             throw e;
22           }
23         }
```

```
      }
      waitingThreads.remove(queueObject);
      isLocked = true;
      lockingThread = Thread.currentThread();
    }
  }

  public synchronized void unlock(){
    if(this.lockingThread != Thread.currentThread()){
      throw new IllegalMonitorStateException(
        "Calling thread has not locked this lock");
    }
    isLocked      = false;
    lockingThread = null;
    if(waitingThreads.size() > 0){
      QueueObject queueObject = waitingThread.get(0);
      synchronized(queueObject){
        queueObject.notify();
      }
    }
  }
}
```

```
public class QueueObject {}
```

我们可以看到synchronized(queueObject)及其中的queueObject.wait()调用是嵌在synchronized(this)块里面的，这会导致嵌套管程锁死问题。为避免这个问题，我们必须将synchronized(queueObject)块移出synchronized(this)块。移出来之后的代码可能是这样的：

```
//Fair Lock implementation with slipped conditions problem
public class FairLock {
  private boolean isLocked = false;
  private Thread lockingThread  = null;
  private List waitingThreads =
            new ArrayList();

  public void lock() throws InterruptedException{
    QueueObject queueObject = new QueueObject();

    synchronized(this){
      waitingThreads.add(queueObject);
    }

    boolean mustWait = true;
    while(mustWait){

      synchronized(this){
        mustWait = isLocked || waitingThreads.get(0) != queueObject;
      }

      synchronized(queueObject){
        if(mustWait){
          try{
            queueObject.wait();
          }catch(InterruptedException e){
            waitingThreads.remove(queueObject);
            throw e;
          }
        }
      }
    }

    synchronized(this){
      waitingThreads.remove(queueObject);
      isLocked = true;
```

```
      lockingThread = Thread.currentThread();
    }
  }
}
```

注意：因为我只改动了lock()方法，这里只展现了lock方法。

现在lock()方法包含了3个同步块。

第一个，synchronized(this)块通过mustWait = isLocked || waitingThreads.get(0) != queueObject检查内部变量的值。

第二个，synchronized(queueObject)块检查线程是否需要等待。也有可能其它线程在这个时候已经解锁了，但我们暂时不考虑这个问题。我们就假设这个锁处在解锁状态，所以线程会立马退出synchronized(queueObject)块。

第三个，synchronized(this)块只会在mustWait为false的时候执行。它将isLocked重新设回true，然后离开lock()方法。

设想一下，在锁处于解锁状态时，如果有两个线程同时调用lock()方法会发生什么。首先，线程1会检查到isLocked为false，然后线程2同样检查到isLocked为false。接着，它们都不会等待，都会去设置isLocked为true。这就是slipped conditions的一个最好的例子。

解决Slipped Conditions问题

要解决上面例子中的slipped conditions问题，最后一个synchronized(this)块中的代码必须向上移到第一个同步块中。为适应这种变动，代码需要做点小改动。下面是改动过的代码：

```
//Fair Lock implementation without nested monitor lockout problem,
//but with missed signals problem.
public class FairLock {
  private boolean isLocked = false;
  private Thread lockingThread  = null;
  private List waitingThreads =
            new ArrayList();

  public void lock() throws InterruptedException{
    QueueObject queueObject = new QueueObject();

    synchronized(this){
      waitingThreads.add(queueObject);
    }

    boolean mustWait = true;
    while(mustWait){
      synchronized(this){
        mustWait = isLocked || waitingThreads.get(0) != queueObject;
        if(!mustWait){
          waitingThreads.remove(queueObject);
          isLocked = true;
          lockingThread = Thread.currentThread();
```

```
            return;
          }
        }

        synchronized(queueObject){
          if(mustWait){
            try{
              queueObject.wait();
            }catch(InterruptedException e){
              waitingThreads.remove(queueObject);
              throw e;
            }
          }
        }
      }
    }
```

我们可以看到对局部变量mustWait的检查与赋值是在同一个同步块中完成的。还可以看到，即使在synchronized(this)块外面检查了mustWait，在while(mustWait)子句中，mustWait变量从来没有在synchronized(this)同步块外被赋值。当一个线程检查到mustWait是false的时候，它将自动设置内部的条件（isLocked），所以其它线程再来检查这个条件的时候，它们就会发现这个条件的值现在为true了。

synchronized(this)块中的`return;`语句不是必须的。这只是个小小的优化。如果一个线程肯定不会等待（即mustWait为false），那么就没必要让它进入到synchronized(queueObject)同步块中和执行if(mustWait)子句了。

细心的读者可能会注意到上面的公平锁实现仍然有可能丢失信号。设想一下，当该FairLock实例处于锁定状态时，有个线程来调用lock()方法。执行完第一个 synchronized(this)块后，mustWait变量的值为true。再设想一下调用lock()的线程是通过抢占式的，拥有锁的那个线程那个线程此时调用了unlock()方法，但是看下之前的unlock()的实现你会发现，它调用了queueObject.notify()。但是，因为lock()中的线程还没有来得及调用queueObject.wait()，所以queueObject.notify()调用也就没有作用了，信号就丢失掉了。如果调用lock()的线程在另一个线程调用queueObject.notify()之后调用queueObject.wait()，这个线程会一直阻塞到其它线程调用unlock方法为止，但这永远也不会发生。

公平锁实现的信号丢失问题在饥饿和公平一文中我们已有过讨论，把QueueObject转变成一个信号量，并提供两个方法：doWait()和doNotify()。这些方法会在QueueObject内部对信号进行存储和响应。用这种方式，即使doNotify()在doWait()之前调用，信号也不会丢失。

fair_jm
2013/03/30 10:19上午

简单点说 就是对一个变量的读写要在同一个同步块中来避免这个错误是吧??

PH
2013/05/17 7:44下午

对于第二个例子，有一点疑问：两个线程可能同时看到 isLocked 都是 false，但是 waitingThreads.get(0) != queueObject 判断必然有一个为 true 另一个为 false（如果 List 那边采用线程安全的实现的话），那么必然会有一个线程的 mustWait 会变成 true。

tongxin0421
2014/06/14 3:19下午

个人理解应该是因为，lock方法里面是有3个同步块，进程1执行完了第一个和第二个同步块，在执行第三个同步块之前，进程2执行第一个和第二个同步块，这时候两个同步块在这里的判断就是一样的了，都是false

景区的小小强
2014/10/10 5:49下午

“首先，线程1会检查到isLocked为false，然后线程2同样检查到isLocked为false。接着，它们都不会等待，都会去设置isLocked为true。这就是slipped conditions的一个最好的例子。” 作者好像没有考虑“ waitingThreads.get(0) != queueObject”这个条件，从而“都会去设置isLocked为true”的结论是不正确的。

无为
2014/11/13 10:20下午

是的 作者这里没有说正确

音无麻里亚
2016/11/26 10:49上午

同意，而且我反复看了好几遍，都没看出这个公平锁哪里会发生slipped conditions

chenqimiao1994@126.com
2017/08/10 4:39下午

反复看了好几遍，确实有问题

zen_me_yi

2014/01/11 11:30上午

Slipped Conditions 这个词该怎么翻译呢？

匿名
2014/06/20 11:17上午

不稳定的条件

无为
2014/11/13 10:27下午

作为一个伪前端,每次请求后刷新页面相当的蛋疼啊.

ps:确实学到不少并发相关的知识,nice job!

无为
2014/11/13 10:28下午

“请求” => 应该是”评论”

smart1988
2017/08/10 5:04下午

最近看了夜行侠老师讲的免费的netty视频

大数据统计分析架构-netty部分

数据统计分析架构介绍

netty服务器编写

Netty服务器与socket通信时编码解码

Netty客户端编写

与spring整合

实现ES或者Dubbo RPC还需要解决那些问题