**JavaScript同步**

译自：<http://onjava.com/pub/a/onjava/2006/04/05/ajax-mutual-exclusion.html?page=3>

随着AJAX范例得到越来越广泛的应用，浏览器页面可以在向后台服务器请求数据的同时保持前端用户界面的活跃性（因此在AJAX中称为异步）。然而，当这两个活动同时访问共用的JavaScript和DOM数据结构时就会引发问题。JavaScript没有提供针对该并发程序问题的经典解决方案。本文描述了作者在互斥机制方面的新见解，该经过验证的互斥机制在JavaScript中能发挥良好的作用。

**为什么需要互斥？**

当多个程序逻辑线程同时访问相同数据的时候，问题便产生了。程序通常假定与其交互的数据在交互过程中不发生改变。访问这些共享数据结构的代码称为临界区，一次只允许一个程序访问的机制被称为互斥。在AJAX应用程序中，当对来自XMLHttpRequest的应答进行异步处理的代码同时操纵正在被用户界面使用的数据时，便会发生这种情况。这个共用的数据可能是用于实现MVC数据模型的JavaScript和/或web页面自身的DOM。如果二者中的任一个对共享数据做了不协调的更改，那么二者的逻辑都将中断。

也许您会说“等等，为什么我没有遇到过这种问题？”。遗憾的是，这种问题是同步依赖的（也叫做竞态条件），因此它们并不总是发生，或者也许从不发生。它们的或然性基于许多因素。基于健壮性考虑，富internet应用程序应该通过确保这些问题不会发生来阻止出现这种情况。

因此，需要一种互斥机制来确保同时只能打开一个临界区，并且在它结束之后才能打开另一个。在大多数主流计算机语言和执行框架中，都提供互斥机制（经常是几种），但是应用于浏览器端的JavaScript却没有提供这种互斥机制。虽然存在一些无需专门的语言或环境支持的经典互斥实现算法，但是即使这样还是需要一些JavaScript和浏览器（如Internet Explorer）所缺少的要素。接下来介绍的经典算法在这些浏览器和语言中能发挥良好的作用。

**面包店算法**

在计算机科学文献中的几种互斥算法中，所谓的Lamport面包店算法可以有效地用于多个相互竞争的控制线程，该算法中线程之间的通信只能在共享内存中进行（即，不需要诸如信号量、原子性的set-and-test之类的专门机制）。该算法的基本思想源于面包店，因为面包店需要先取号然后等候叫号。清单1给出了该算法的框架（引自Wikipedia），该算法可以使各线程进出临界区而不产生冲突。

清单1. Lamport面包店算法伪代码

// declaration & initial values of global variables

Enter, Number: array [1..N] of integer = {0};

// logic used by each thread...

// where "(a, b) < (c, d)"

// means "(a < c) or ((a == c) and (b < d))"

Thread(i) {

while (true) {

Enter [i] = 1;

Number[i] = 1 + max(Number[1],...,Number[N]);

Enter [i] = 0;

for (j=1; j<=N; ++j) {

while (Enter[j] != 0) {

// wait until thread j receives its number

}

while ((Number[j]!=0)

&& ((Number[j],j) < (Number[i],i))) {

// wait until threads with smaller numbers

// or with the same number, but with higher

// priority, finish their work

}

}

// critical section...

Number[i] = 0;

// non-critical section...

}

}

如上所示，该算法假定各线程清楚自己的线程编号（常量i）和当前正在活动的线程总数（常量N）。此外，还假定存在一种等待或休眠方式，例如：暂时将CPU释放给其他线程。遗憾的是，Internet Explorer中的JavaScript没有这种能力。虽然如此，如果实际运行在同一线程上的多个代码部分表现为各自运行在独立的虚拟线程上，那么该面包店算法不会中断。同样，JavaScript具有一种在指定延迟后调度函数的机制，所以，可以使用下面的这些方法来优化面包店算法。

**Wallace变体**

在JavaScript中实现Lamport面包店算法的主要障碍在于缺少线程API。无法确定当前正在哪个线程上运行以及当前正在活动的线程数目，也无法将CPU释放给其他的线程，无法创建新的线程来管理其他线程。因此，无法查证如何将特定的浏览器事件（例如：单击按纽、可用的XML应答等）分配到线程。

克服这些障碍的一种方法是使用Command设计模式。通过将所有应该进入临界区的逻辑以及所有启动该逻辑所需的数据一起放入到command 对象中，可以在负责管理command的类中重写面包店算法。该互斥类仅在没有其他临界区（封装为独立的command对象方法）在执行时调用临界区，就像它们各自运行在不同的虚拟线程中一样。JavaScript的setTimeout()机制用于将CPU释放给其他正在等待的command。

为command对象假定一个简单的基类（见清单2中的Command），可以定义一个类（见清单3中的Mutex）来实现面包店算法的Wallace变体。注意，虽然可以通过很多方式在JavaScript中实现基类对象（为了简洁起见，这里使用一种简单的方式），但是只要各个command对象拥有某个惟一的id，而且整个临界区被封装在单独的方法中，那么任何对象模式都可以使用这种方法。

清单2. 用于 Command 对象的简单基类

1 function Command() {

2 if (!Command.NextID) Command.NextID = 0;

3 this.id = ++Command.NextID;

4 // unsynchronized API

5 this.doit = function(){ alert("DOIT called"); }

6 this.undo = function(){ alert("UNDO called"); }

7 this.redo = function(){ this.doit(); }

8 // synchronized API

9 this.sDoIt = function(){ new Mutex(this,"doit"); }

10 this.sUnDo = function(){ new Mutex(this,"undo"); }

11 this.sReDo = function(){ new Mutex(this,"redo"); }

12 }

Command类演示了三个临界区方法（见5-7行），但是只要预先将对该方法的调用封装在Mutex中（见9-11行），那么就可以使用任何方法。有必要认识到，常规方法调用（例如非同步的方法调用）与同步方法调用之间存在着重要的区别：具有讽刺意味的是，必须保证同步方法不同步运行。换句话说，当调用sDoIt()方法时，必须确保方法doit()还未运行，即使方法sDoIt()已经返回。doit()方法可能已结束，或者直到将来的某一时间才开始执行。也就是说，将对Mutex的实例化视为启动一个新的线程。

清单3.作为类 Mutex实现的 Wallace 变体

1 function Mutex( cmdObject, methodName ) {

2 　// define static field and method

3 　if (!Mutex.Wait) Mutex.Wait = new Map();

4 　　Mutex.SLICE = function( cmdID, startID ) {

5 　　　Mutex.Wait.get(cmdID).attempt( Mutex.Wait.get(startID) );

6 　　}

7 　　// define instance method

8 　　this.attempt = function( start ) {

9 　　　for (var j=start; j; j=Mutex.Wait.next(j.c.id)) {

10　　　　 if (j.enter

11　　　　　 || (j.number && (j.number ＜ this.number ||

12　　　　　 (j.number == this.number

13　　　　　 && j.c.id ＜ this.c.id))))

14　　　　　 return setTimeout

15　　　　　 ("Mutex.SLICE("+this.c.id+","+j.c.id+")",10);

16　　　　 }

17　　　　 //run with exclusive access

18　　　　 this.c[ this.methodID ]();

19　　　　 //release exclusive access

20　　　　 this.number = 0;

21　　　　 Mutex.Wait.remove( this.c.id );

22　　　 }

23　　　 // constructor logic

24　　　 this.c = cmdObject;

25　　　 this.methodID = methodName;

26　　　 //(enter and number are "false" here)

27　　　 Mutex.Wait.add( this.c.id, this );

28　　　 this.enter = true;

29　　　 this.number = (new Date()).getTime();

30　　　 this.enter = false;

31　　 this.attempt( Mutex.Wait.first() );

32　 }

Mutex类的基本逻辑是将每个新的Mutex实例放入主等待清单，然后将其在等待队列中启动。因为每次到达“队首”的尝试都需要等待（除了最后一次），所以使用setTimeout来调度每次在当前尝试停止的位置启动的新尝试。到达队首时（见17行），便实现了互斥性访问；因此，可以调用临界区方法。执行完临界区后，释放互斥性访问并从等待清单中移除Mutex实例（见20-21行）。

Mutex构造函数（见23-31行）记录其Command对象和方法名参数，然后寄存在一个运行中临界区的稀疏数组中（Mutex.Wait），这通过清单4中所示的Map类来实现。然后构造函数获得下一个编号，并在队尾开始排队。由于等待编号中的间隔或副本不存在问题，所以实际上使用当前的时间戳作为下一个编号。

attempt()方法将初始伪代码中的两个wait循环组合成一个单独的循环，该循环直到队首时才对临界区失效。该循环是一种忙碌-等待循环检测方式，可以通过在setTimeout()调用中指定延迟量来终止该循环。由于setTimeout需要调用“无格式函数”，所以在第4-6行定义了静态帮助器方法（Mutex.SLICE）。SLICE在主等待清单中查找指定的Mutex对象，然后调用其attempt()方法，用start参数指定到目前为止其所获得的等待清单的长度。每次SLICE()调用都像获得了“一块CPU”。这种（通过setTimeout）适时释放CPU的协作方式令人想到协同程序。

清单4. 作为 Map数据结构实现的稀疏数组

function Map() {

this.map = new Object();

// Map API

this.add = function( k,o ){

this.map[k] = o;

}

this.remove = function( k ){

delete this.map[k];

}

this.get = function( k ){

return k==null ? null : this.map[k];

}

this.first = function(){

return this.get( this.nextKey() );

}

this.next = function( k ){

return this.get( this.nextKey(k) );

}

this.nextKey = function( k ){

for (i in this.map) {

if ( !k ) return i;

if (k==i) k=null; /\*tricky\*/

}

return null;

}

}

**富Internet应用程序集成**

由于Mutex所处理的线程（虚拟的或者非虚拟的）数量是动态变化的，所以可以确定一个基本事实：无法通过像浏览器为各个浏览器事件分配单独的线程那样的方式来获得线程标识符。这里做了一个类似的假定，那就是每个完整的事件处理程序组成一个完整的临界区。基于这些假定，每个事件处理函数都可以转变成一个command对象，并使用Mutex对其进行管理。当然，如果未将代码明确组织成事件处理函数，那么将需要重构。换句话说，不是直接在HTML事件属性中进行逻辑编码（例如：onclick='++var'），而是调用事件处理函数（例如：onclick='FOO()'和function FOO(){++var;}）。

清单5. 使用了非同步事件处理程序的示例web页面

<html>

<script language="JavaScript">

function newState(){

if (XMLreq.readyState==4) processReply();

}

function requestData(){

...set up asynchronous XML request...

XMLreq.onreadystatechange = newState;

...launch XML request...

}

function processReply(){

var transformedData = ...process data to HTML...

OutputArea.innerHTML = transformedData + "<br>";

}

function clearArea(){

OutputArea.innerHTML = "cleared<br>";

}

</script>

<body onload="requestData();">

<input type="button" value="clear" onclick="clearArea()">

<div id="OutputArea"/>

</body>

</html>

例如，假设有三个事件处理程序函数，它们操纵清单5所示的共用数据。它们处理页面加载事件、单击按钮事件和来自XML请求的应答事件。页面加载事件发出某个异步请求来要求获取数据并指定请求-应答事件处理程序，该处理程序处理接收到的数据，并将其加载到共用数据结构。单击按钮事件处理程序也影响共用数据结构。为了避免这些事件处理程序发生冲突，可以通过清单6所示的Mutex将它们转变成command并加以调用（假设JavaScript include文件mutex.js中包含Map和Mutex）。注意，虽然可以使用优美的类继承机制来实现Command子类，但是该代码说明了最简单的方法，该方法仅需要全局变量NEXT\_CMD\_ID。

清单6. 转化为同步事件处理程序的web页面

<html>

<script src="mutex.js"></script>

<script language="JavaScript">

function requestData (){

new Mutex(new RequestDataCmd(),"go"); }

function processReply(){

new Mutex(new ProcessReplyCmd(),"go"); }

function clearArea (){

new Mutex(new ClearAreaCmd(),"go"); }

function newState (){

if (XMLreq.readyState==4) processReply(); }

var NEXT\_CMD\_ID = 0;

function RequestDataCmd(){

this.id = ++NEXT\_CMD\_ID;

this.go = function(){

...set up asynchronous XML request...

XMLreq.onreadystatechange = NewState;

...launch XML request...

}

}

function ProcessReplyCmd(){

this.id = ++NEXT\_CMD\_ID;

this.go = function(){

var transformedData = ...process data to HTML...

OutputArea.innerHTML = transformedData + "<br>";

}

}

function ClearAreaCmd(){

this.id = ++NEXT\_CMD\_ID;

this.go = function(){

OutputArea.innerHTML = "cleared<br>"; }

}

</script>

<body onload="requestData();">

<input type="button" value="clear" onclick="clearArea()">

<div id="OutputArea"/>

</body>

</html>

已经通过Mutex将这三个事件处理程序函数转变为调用它们的初始逻辑（当前都被预包装于command类中）。各个command类定义一个独特的标识符和一个包含临界区逻辑的方法，从而满足了command接口的要求。

**结束语**

借助于AJAX和RIA，构建复杂的动态用户界面的推动力正在促使开发人员使用先前与胖GUI客户端紧密联系的设计模式（例如：模型-视图-控制器）。随着视图和控制器的定义模块化，且每一个都带有自己的事件和事件处理程序（除了共用数据模型），发生冲突的机率成倍提高。通过把事件处理逻辑封装到Command类中，不仅可以使用Wallace变体，而且为提供丰富的撤消/重做功能、脚本编写界面和单元测试工具创造了条件。

**Resources**

1. The [sample code](http://www.polyglotinc.com/Mutex/examples) in this article is available for viewing or download. It includes the omitted details such that the web pages can execute directly in the browser without a server connection.
2. An example JavaScript framework (Gravy) that uses the techniques in this article is available for viewing or downloading, complete with [JsDoc](http://jsdoc.sourceforge.net/) documentation. Gravy supports placing all application functionality in the browser in JavaScript. Applications need only access the server for database CRUD operations.