**Concurrent.Thread使用说明**

虽然有越来越多的网站在应用AJAX技术进行开发，但是构建一个复杂的AJAX应用仍然是一个难题。造成这些困难的主要原因是什么呢？是与服务器的异步通信问题？还是GUI程序设计问题呢？通常这两项工作都是由桌面程序来完成的，那究竟为何开发一个可以实现同样功能的AJAX应用就这么困难呢？

**AJAX 开发中的难题**

让我们通过一个简单的例子来认识这个问题。假设你要建立一个树形结构的公告栏系统(BBS)，它可以根据用户请求与服务器进行交互，动态加载每篇文章的信息，而不是一次性从服务器载入所有文章信息。每篇文章有四个相关属性：系统中可以作为唯一标识的ID、发贴人姓名NAME、文章内容CONTENT以及包含其所有子文章 ID的数组信息CHILDREN。首先假定有一个名为getArticle(ID)的函数可以加载一篇文章信息。该函数接收的参数是要加载文章的ID，通过它可从服务器获取文章信息。它返回的对象包含与文章相关的四条属性：ID，NAME，CONTENT和CHILDREN。例程如下：

function ( id ) {

var a = getArticle(id);

document.writeln(a.name + " " + a.content);

}

或许你会注意到，重复用同一个文章ID调用此函数，需要与服务器之间进行反复且无益的通信。想要解决这个问题，可以考虑使用函数 getArticleWithCache(id)，它相当于一个带有缓存能力的getArticle(id)。在这个例子中，getArticle(id)返回的数据只是作为一个全局变量被保存下来：

var cache = {};

function getArticleWithCache ( id ) {

if ( !cache[id] ) {

cache[id] = getArticle(id);

}

return cache[id];

}

现在已将读入的文章缓存起来，让我们再来考虑一下函数backgroundLoad()，它应用我们上面提到的缓存机制加载所有文章信息。其用途是，当读者在阅读某篇文章时，从后台预加载它所有子文章。因为文章数据是树状结构的，所以很容易写一个递归的算法来遍历树并且加载所有的文章：

function backgroundLoad ( ids ) {

for ( var i=0; i < ids.length; i++ ) {

var a = getArticleWithCache(ids[i]);

backgroundLoad(a.children);

}

}

backgroundLoad ()函数接收一个ID数组作为参数，然后通过每个ID调用前面定义过的getArticldWithCache()方法，这样就把每个ID对应的文章缓存起来。之后再通过已加载文章的子文章ID数组递归调用backgroundLoad()方法，如此整个文章树就被缓存起来。

到目前为止，一切似乎看起来都很完美。然而，只要你有过开发AJAX应用的经验，你就应该知晓这种幼稚的实现方法根本不会成功，这个例子成立的基础是默认 getArticle()用的是同步通信。可是，作为一条基本原则，JavaScript要求在与服务器进行交互时要用异步通信，因为它是单线程的。就简单性而言，把每一件事情（包括GUI事件和渲染）都放在一个线程里来处理是一个很好的程序模型，因为这样就无需再考虑线程同步这些复杂问题。另一方面，他也暴露了应用开发中的一个严重问题，单线程环境看起来对用户请求响应迅速，但是当线程忙于处理其它事情时(比如说调用getArticle())，就不能对用户的鼠标点击和键盘操作做出响应。

如果在这个单线程环境里进行同步通信会发生什么事情呢？同步通信会中断浏览器的执行直至获得通信结果。在等待通信结果的过程中，由于服务器的调用还没有完成，线程会停止响应用户并保持锁定状态直到调用返回。因为这个原因，当浏览器在等待服务器响应时它不能对用户行为作出响应，所以看起来像是冻结了。当执行 getArticleWithCache()和backgroundLoad()会有同样的问题，因为它们都是基于getArticle()函数的。由于下载所有的文章可能会耗费很可观的一段时间，因此对于backgroundLoad()函数来说，浏览器在此段时间内的冻结就是一个很严重的问题——既然浏览器都已经冻结，当用户正在阅读文章时就不可能首先去执行后台预加载数据，如果这样做连当前的文章都没办法读。

如上所述，既然同步通信在使用中会造成如此严重的问题，JavaScript就把异步通信作为一条基本原则。因此，我们可以基于异步通信改写上面的程序。 JavaScript要求以一种事件驱动的程序设计方式来写异步通信程序。在很多场合中，你都必须指定一个回调程序，一旦收到通信响应，这个函数就会被调用。例如，上面定义的getArticleWithCache()可以写成这样：

var cache = {};

function getArticleWithCache ( id, callback ) {

if ( !cache[id] ) {

callback(cache[id]);

} else {

getArticle(id, function( a ){

cache[id] = a;

callback(a);

});

}

}

这个程序也在内部调用了getArticle()函数。然而需要注意的是，为异步通信设计的这版getArticle()函数要接收一个函数作为第二个参数。当调用这个getArticle()函数时，与从前一样要给服务器发送一个请求，不同的是，现在函数会迅速返回而非等待服务器的响应。这意味着，当执行权交回给调用程序时，还没有得到服务器的响应。如此一来，线程就可以去执行其它任务直至获得服务器响应，并在此时调用回调函数。一旦得到服务器响应， getArticle()的第二个参数作为预先定义的回调函数就要被调用，服务器的返回值即为其参数。同样的，getArticleWithCache ()也要做些改变，定义一个回调参数作为其第二个参数。这个回调函数将在被传给getArticle()的回调函数中调用，因而它可以在服务器通信结束后被执行。

单是上面这些改动你可能已经认为相当复杂了，但是对backgroundLoad()函数做得改动将会更复杂，它也要被改写成可以处理回调函数的形式：

function backgroundLoad ( ids, callback ) {

var i = 0;

function l ( ) {

if ( i < ids.length ) {

getArticleWithCache(ids[i++], function( a ){

backgroundLoad(a.children, l);

});

} else {

callback();

}

}

l();

}

改动后的backgroundLoad()函数看上去和我们以前的那个函数已经相去甚远，不过他们所实现的功能并无二致。这意味着这两个函数都接受 ID数组作为参数，对于数组里的每个元素都要调用getArticleWithCache()，再应用已经获得子文章ID递归调用 backgroundLoad ()。不过同样是对数组的循环访问，新函数中的就不太好辨认了，以前的程序中是用一个for循环语句完成的。为什么实现同样功能的两套函数是如此的大相径庭呢？

这个差异源于一个事实：任何函数在遇到有需要同服务器进行通信情况后，都必须立刻返回，例如getArticleWithCache()。除非原来的函数不在执行当中，否则应当接受服务器响应的回调函数都不能被调用。对于JavaScript，在循环过程中中断程序并在稍后从这个断点继续开始执行程序是不可能的，例如一个for语句。因此，本例利用递归传递回调函数实现循环结构而非一个传统循环语句。对那些熟悉连续传送风格(CPS)的人来说，这就是一个 CPS的手动实现，因为不能使用循环语法，所以即便如前面提到的遍历树那么简单的程序也得写得很复杂。与事件驱动程序设计相关的问题是控制流问题：循环和其它控制流表达式可能比较难理解。

这里还有另外一个问题：如果你把一个没有应用异步通信的函数转换为一个使用异步通信的函数，那么重写的函数将需要一个回调函数作为新增参数，这为已经存在的APIs造成了很大问题，因为内在的改变没有把影响限于内部，而是导致整体混乱的APIs以及API的其它使用者的改变。

造成这些问题目的根本原因是什么呢？没错，正是JavaScript单线程机制导致了这些问题。在单线程里执行异步通信需要事件驱动程序设计和复杂的语句。如果当程序在等待服务器的响应时，有另外一个线程可以来处理用户请求，那么上述复杂技术就不需要了。

**试试多线程编程**

让我来介绍一下Concurrent.Thread，它是一个允许JavaScript进行多线程编程的库，应用它可以大大缓解上文提及的在 AJAX开发中与异步通信相关的困难。这是一个用JavaScript写成的免费的软件库，使用它的前提是遵守Mozilla Public License和GNU General Public License这两个协议。你可以从他们的网站 下载源代码。

马上来下载和使用源码吧！假定你已经将下载的源码保存到一个名为Concurrent.Thread.js的文件夹里，在进行任何操作之前，先运行如下程序，这是一个很简单的功能实现：

<script type="text/javascript" src="Concurrent.Thread.js"> </script>

<script type="text/javascript">

Concurrent.Thread.create(function(){

var i = 0;

while ( 1 ) {

document.body.innerHTML += i++ + " <br>";

}

});

</script>

执行这个程序将会顺序显示从0开始的数字，它们一个接一个出现，你可以滚屏来看它。现在让我们来仔细研究一下代码，他应用while(1)条件制造了一个不会中止的循环，通常情况下，象这样不断使用一个并且是唯一一个线程的JavaScript程序会导致浏览器看起来象冻结了一样，自然也就不会允许你滚屏。那么为什么上面的这段程序允许你这么做呢？关键之处在于while(1)上面的那条Concurrent.Thread.create()语句，这是这个库提供的一个方法，它可以创建一个新线程。被当做参数传入的函数在这个新线程里执行，让我们对程序做如下微调：

<script type="text/javascript" src="Concurrent.Thread.js"> </script>

<script type="text/javascript">

function f ( i ){

while ( 1 ) {

document.body.innerHTML += i++ + " <br>";

}

}

Concurrent.Thread.create(f, 0);

Concurrent.Thread.create(f, 100000);

</script>

在这个程序里有个新函数f()可以重复显示数字，它是在程序段起始定义的，接着以f()为参数调用了两次create()方法，传给create()方法的第二个参数将会不加修改地传给f()。执行这个程序，先会看到一些从0开始的小数，接着是一些从100,000开始的大数，然后又是接着前面小数顺序的数字。你可以观察到程序在交替显示小数和大数，这说明两个线程在同时运行。

让我来展示Concurrent.Thread的另外一个用法。上面的例子调用create()方法来创建新线程。不调用库里的任何APIs也有可能实现这个目的。例如，前面那个例子可以这样写：

<script type="text/javascript" src="Concurrent.Thread.js"> </script>

<script type="text/x-script.multithreaded-js">

var i = 1;

while ( 1 ) {

document.body.innerHTML += i++ + " <br>";

}

</script>

在script 标签内，很简单地用JavaScript写了一个无穷循环。你应该注意到标签内的type属性，那里是一个很陌生的值(text/x- script.multithreaded-js)，如果这个属性被放在script标签内，那么Concurrent.Thread就会在一个新的线程内执行标签之间的程序。你应当记住一点，在本例一样，必须将Concurrent.Thread库包含进来。

有了Concurrent.Thread，就有可能自如的将执行环境在线程之间进行切换，即使你的程序很长、连续性很强。我们可以简要地讨论下如何执行这种操作。简言之，需要进行代码转换。粗略地讲，首先要把传递给create()的函数转换成一个字符串，接着改写直至它可以被分批分次执行。然后这些程序可以依照调度程序逐步执行。调度程序负责协调多线程，换句话说，它可以在适当的时候做出调整以便每一个修改后的函数都会得到同等机会运行。 Concurrent.Thread实际上并没有创建新的线程，仅仅是在原本单线程的基础上模拟了一个多线程环境。

虽然转换后的函数看起来是运行在不同的线程内，但是实际上只有一个线程在做这所有的事情。在转换后的函数内执行同步通信仍然会造成浏览器冻结，你也许会认为以前的那些问题根本就没有解决。不过你不必担心，Concurrent.Thread提供了一个应用JavaScript 的异步通信方式实现的定制通信库，它被设计成当一个线程在等待服务器的响应时允许其它线程运行。这个通信库存于 Concurrent.Thread.Http下。它的用法如下所示：

<script type="text/javascript" src="Concurrent.Thread.js"> </script>

<script type="text/x-script.multithreaded-js">

var req = Concurrent.Thread.Http.get(url, ["Accept", "\*"]);

if (req.status == 200) {

alert(req.responseText);

} else {

alert(req.statusText);

}

</script>

get()方法，就像它的名字暗示的那样，可以通过HTTP的GET方法获得指定URL的内容，它将目标URL作为第一个参数，将一个代表HTTP 请求头的数组作为可选的第二个参数。get()方法与服务器交互，当得到服务器的响应后就返回一个XMLHttpRequest对象作为返回值。当 get()方法返回时，已经收到了服务器响应，所以就没必要再用回调函数接收结果。自然，也不必再耽心当程序等待服务器的响应时浏览器冻结的情况了。另外，还有一个 post()方法可以用来发送数据到服务器：

<script type="text/javascript" src="Concurrent.Thread.js"> </script>

<script type="text/x-script.multithreaded-js">

var req = Concurrent.Thread.Http.post(url, "key1=val1&key2=val2");

alert(req.statusText);

</script>

post()方法将目的URL作为第一个参数，要发送的内容作为第二个参数。像get()方法那样，你也可以将请求头作为可选的第三个参数。

如果你用这个通信库实现了第一个例子当中的getArticle()方法，那么你很快就能应用文章开头示例的那种简单的方法写出 getArticleWithCache(),backgroundLoad ()以及其它调用了getArticle()方法的函数了。即使是那版backgroundLoad()正在读文章数据，照例还有另外一个线程可以对用户请求做出响应，浏览器因此也不会冻结。现在，你能理解在JavaScript中应用多线程有多实用了？

**想了解更多**

我向你介绍了一个可以在JavaScript中应用多线程的库：Concurrent.Thread。这篇文章的内容只是很初级的东西，如果你想更深入的了解，我推荐您去看the tutorial。它提供有关Concurrent.Thread用法的更多内容，并列出了可供高级用户使用的文档，是最适合起步的材料。访问他们的网站也不错，那里提供更多信息。

**有关作者**

Daisuke Maki：从 International Christian大学文科学院自然科学分部毕业后（取得文学学士学位），又在Electro-Communications大学的研究生院信息专业攻读硕士学位。擅长Web开发和应用JavaScript的AJAX。他开发了Concurrent.Thread。2006财政年度在日本信息技术促进机构（IPA）指导的项目Explatory Software Project中应用了这个设计。

目前已经拥有一个工学硕士学位的他正在Electro-Communications大学的研究生院注册攻读博士学位。

作者 Daisuke Maki译者 张凯峰