[**非阻塞式JavaScript脚本介绍**](http://www.cnblogs.com/jenry/archive/2011/02/13/1953211.html)

JavaScript 倾向于阻塞浏览器某些处理过程，如HTTP 请求和界面刷新，这是开发者面临的最显著的性能问题。保持JavaScript文件短小，并限制HTTP请求的数量，只是创建反应迅速的网页应用的第一步。一个应用程序所包含的功能越多，所需要的JavaScript 代码就越大，保持源码短小并不总是一种选择。尽管下载一个大JavaScript 文件只产生一次HTTP 请求，却会锁定浏览器一大段时间。为避开这种情况，你需要向页面中逐步添加JavaScript，某种程度上说不会阻塞浏览器。非阻塞脚本的秘密在于，等页面完成加载之后，再加载JavaScript 源码。从技术角度讲，这意味着在window 的load 事件发出之后开始下载代码。有三种方法可以实现这种效果。

**延期脚本**

HTML 4 为<script>标签定义了一个扩展属性：defer。这个defer 属性指明元素中所包含的脚本不打算修改DOM，因此代码可以稍后执行。defer 属性只被Internet Explorer 4 和Firefox 3.5 更高版本的浏览器所支持，它不是一个理想的跨浏览器解决方案。在其他浏览器上，defer 属性被忽略，<script>标签按照默认方式被处理（造成阻塞）。如果浏览器支持的话，这种方法仍是一种有用的解决方案。示例如下：

<script type="text/javascript" src="file1.js" defer></script>

一个带有defer 属性的<script>标签可以放置在文档的任何位置。对应的JavaScript 文件将在<script>被解析时启动下载，但代码不会被执行，直到DOM 加载完成。（在onload 事件句柄被调用之前）。当一个defer的JavaScript 文件被下载时，它不会阻塞浏览器的其他处理过程，所以这些文件可以与页面的其他资源一起并行下载。任何带有defer 属性的<script>元素在DOM 加载完成之前不会被执行，不论是内联脚本还是外部脚本文件，都是这样。下面的例子展示了defer 属性如何影响脚本行为：

<html>  
 <head>  
 <title>Script Defer Example</title>  
 </head>  
 <body>  
 <script defer>  
 alert("defer");  
 </script>  
 <script>  
 alert("script");  
 </script>  
 <script>  
 window.onload = function(){  
 alert("load");  
 };  
 </script>  
 </body>  
</html>

这些代码在页面处理过程中弹出三个对话框。如果浏览器不支持defer 属性，那么弹出对话框的顺序是"defer"，"script"和"load"。如果浏览器支持defer 属性，那么弹出对话框的顺序是"script"，"defer"和"load"。注意，标记为defer 的<script>元素不是跟在第二个后面运行，而是在onload 事件句柄处理之前被调用。如果你的目标浏览器只包括Internet Explorer 和Firefox 3.5，那么defer 脚本确实有用。如果你需要支持跨领域的多种浏览器，那么还有更一致的实现方式。

**动态脚本元素**

文档对象模型（DOM）允许你使用JavaScript 动态创建HTML 的几乎全部文档内容。其根本在于，<script>元素与页面其他元素没有什么不同：引用变量可以通过DOM 进行检索，可以从文档中移动、删除，也可以被创建。一个新的<script>元素可以非常容易地通过标准DOM 函数创建：

var script = document.createElement ("script");  
 script.type = "text/javascript";  
 script.src = "file1.js";  
 document.getElementsByTagName\_r("head")[0].appendChild(script);

新的<script>元素加载file1.js 源文件。此文件当元素添加到页面之后立刻开始下载。此技术的重点在于：无论在何处启动下载，文件的下载和运行都不会阻塞其他页面处理过程。你甚至可以将这些代码放在<head>部分而不会对其余部分的页面代码造成影响（除了用于下载文件的HTTP 连接）。

当文件使用动态脚本节点下载时，返回的代码通常立即执行（除了Firefox 和Opera，他们将等待此前的所有动态脚本节点执行完毕）。当脚本是"自运行"类型时这一机制运行正常，但是如果脚本只包含供页面其他脚本调用调用的接口，则会带来问题。这种情况下，你需要跟踪脚本下载完成并准备妥善的情况。可以使用动态<script>节点发出事件得到相关信息。

Firefox, Opera, Chorme 和Safari 3+会在<script>节点接收完成之后发出一个load 事件。你可以监听这一事件，以得到脚本准备好的通知：

var script = document.createElement ("script")  
 script.type = "text/javascript";  
 //Firefox, Opera, Chrome, Safari 3+  
 script.onload = function(){  
 alert("Script loaded!");  
 };  
 script.src = "file1.js";  
 document.getElementsByTagName("head")[0].appendChild(script);

Internet Explorer 支持另一种实现方式，它发出一个readystatechange 事件。<script>元素有一个readyState 属性，它的值随着下载外部文件的过程而改变。readyState 有五种取值：

* "uninitialized"默认状态
* "loading"下载开始
* "loaded"下载完成
* "interactive"下载完成但尚不可用
* "complete"所有数据已经准备好

微软文档上说，在<script>元素的生命周期中，readyState 的这些取值不一定全部出现，但并没有指出哪些取值总会被用到。实践中，我们最感兴趣的是"loaded"和"complete"状态。Internet Explorer 对这两个readyState 值所表示的最终状态并不一致，有时<script>元素会得到"loader"却从不出现"complete"，但另外一些情况下出现"complete"而用不到"loaded"。最安全的办法就是在readystatechange 事件中检查这两种状态，并且当其中一种状态出现时，删除readystatechange 事件句柄（保证事件不会被处理两次）：

var script = document.createElement("script")  
 script.type = "text/javascript";  
 //Internet Explorer  
 script.onreadystatechange = function(){  
 if (script.readyState == "loaded" || script.readyState == "complete"){  
 script.onreadystatechange = null;  
 alert("Script loaded.");  
 }  
 };  
 script.src = "file1.js";  
 document.getElementsByTagName("head")[0].appendChild(script);

大多数情况下，你希望调用一个函数就可以实现JavaScript 文件的动态加载。下面的函数封装了标准实现和IE 实现所需的功能：

function loadScript(url, callback){  
 var script = document.createElement ("script")  
 script.type = "text/javascript";  
 if (script.readyState){ //IE  
 script.onreadystatechange = function(){  
 if (script.readyState == "loaded" || script.readyState == "complete"){  
 script.onreadystatechange = null;  
 callback();  
 }  
 };  
 } else { //Others  
 script.onload = function(){  
 callback();  
 };  
 }  
 script.src = url;  
 document.getElementsByTagName("head")[0].appendChild(script);  
 }

此函数接收两个参数：JavaScript 文件的URL，和一个当JavaScript 接收完成时触发的回调函数。属性检查用于决定监视哪种事件。最后一步，设置src 属性，并将<script>元素添加至页面。此loadScript()函数使用方法如下：

loadScript("file1.js", function(){  
 alert("File is loaded!");  
 });

你可以在页面中动态加载很多JavaScript 文件，但要注意，浏览器不保证文件加载的顺序。所有主流浏览器之中，只有Firefox 和Opera 保证脚本按照你指定的顺序执行。其他浏览器将按照服务器返回它们的次序下载并运行不同的代码文件。你可以将下载操作串联在一起以保证他们的次序，如下：

loadScript("file1.js", function(){  
 loadScript("file2.js", function(){  
 loadScript("file3.js", function(){  
 alert("All files are loaded!");  
 });  
 });  
 });

此代码等待file1.js 可用之后才开始加载file2.js，等file2.js 可用之后才开始加载file3.js。虽然此方法可行，但如果要下载和执行的文件很多，还是有些麻烦。如果多个文件的次序十分重要，更好的办法是将这些文件按照正确的次序连接成一个文件。独立文件可以一次性下载所有代码（由于这是异步进行的，使用一个大文件并没有什么损失）。

动态脚本加载是非阻塞JavaScript 下载中最常用的模式，因为它可以跨浏览器，而且简单易用。

**使用XMLHttpRequest(XHR)对象**

此技术首先创建一个XHR 对象，然后下载JavaScript 文件，接着用一个动态<script>元素将JavaScript 代码注入页面。下面是一个简单的例子：

var xhr = new XMLHttpRequest();  
 xhr.open("get", "file1.js", true);  
 xhr.onreadystatechange = function(){  
 if (xhr.readyState == 4){  
 if (xhr.status >= 200 && xhr.status < 300 || xhr.status == 304){  
 var script = document.createElement ("script");  
 script.type = "text/javascript";  
 script.text = xhr.responseText;  
 document.body.appendChild(script);  
 }  
 }  
 };  
 xhr.send(null);

此代码向服务器发送一个获取file1.js 文件的GET 请求。onreadystatechange 事件处理函数检查readyState是不是4，然后检查HTTP 状态码是不是有效（2XX 表示有效的回应，304 表示一个缓存响应）。如果收到了一个有效的响应，那么就创建一个新的<script>元素，将它的文本属性设置为从服务器接收到的responseText 字符串。这样做实际上会创建一个带有内联代码的<script>元素。一旦新<script>元素被添加到文档，代码将被执行，并准备使用。这种方法的主要优点是，你可以下载不立即执行的JavaScript 代码。由于代码返回在<script>标签之外（换句话说不受<script>标签约束），它下载后不会自动执行，这使得你可以推迟执行，直到一切都准备好了。另一个优点是，同样的代码在所有现代浏览器中都不会引发异常。此方法最主要的限制是：JavaScript 文件必须与页面放置在同一个域内，不能从CDNs 下载（CDN 指"内容投递网络（Content Delivery Network）"，大型网页通常不采用XHR 脚本注入技术。

推荐的向页面加载大量JavaScript 的方法分为两个步骤：第一步，包含动态加载JavaScript 所需的代码，然后加载页面初始化所需的除JavaScript 之外的部分。这部分代码尽量小，可能只包含loadScript()函数，它下载和运行非常迅速，不会对页面造成很大干扰。当初始代码准备好之后，用它来加载其余的JavaScript。例如：

<script type="text/javascript" src="loader.js"></script>  
 <script type="text/javascript">  
 loadScript("the-rest.js", function(){  
 Application.init();  
 });  
 </script>

将此代码放置在body 的关闭标签</body>之前。这样做有几点好处：首先，像前面讨论过的那样，这样做确保JavaScript 运行不会影响页面其他部分显示。其次，当第二部分JavaScript 文件完成下载，所有应用程序所必须的DOM 已经创建好了，并做好被访问的准备，避免使用额外的事件处理（例如window.onload）来得知页面是否已经准备好了。另一个选择是直接将loadScript()函数嵌入在页面中，这可以避免另一次HTTP 请求。例如：

<script type="text/javascript">  
 function loadScript(url, callback){  
 var script = document.createElement ("script")  
 script.type = "text/javascript";  
 if (script.readyState){ //IE  
 script.onreadystatechange = function(){  
 if (script.readyState == "loaded" ||  
 script.readyState == "complete"){  
 script.onreadystatechange = null;  
 callback();  
 }  
 };  
 } else { //Others  
 script.onload = function(){  
 callback();  
 };  
 }  
 script.src = url;  
 document.getElementsByTagName\_r("head")[0].appendChild(script);  
 }  
 loadScript("the-rest.js", function(){  
 Application.init();  
 });  
 </script>

如果你决定使用这种方法，建议你使用''YUI Compressor''或者类似的工具将初始化脚本缩小到最小字节尺寸。一旦页面初始化代码下载完成，你还可以使用loadScript()函数加载页面所需的额外功能函数。