Loading and Execution 加载和运行

JavaScript 在浏览器中的性能，可认为是开发者所要面对的最重要的可用性问题。此问题因JavaScript的阻塞特征而复杂，也就是说，当JavaScript 运行时其他的事情不能被浏览器处理。事实上，大多数浏览器使用单进程处理UI 更新和JavaScript 运行等多个任务，而同一时间只能有一个任务被执行。JavaScript

运行了多长时间，那么在浏览器空闲下来响应用户输入之前的等待时间就有多长。

从基本层面说，这意味着<script>标签的出现使整个页面因脚本解析、运行而出现等待。不论实际的JavaScript 代码是内联的还是包含在一个不相干的外部文件中，页面下载和解析过程必须停下，等待脚本完成这些处理，然后才能继续。这是页面生命周期必不可少的部分，因为脚本可能在运行过程中修改页面

内容。典型的例子是document.write()函数，例如：

<html>

<head>

<title>Script Example</title>

</head>

<body>

<p>

<script type="text/javascript">

document.write("The date is " + (new Date()).toDateString());

</script>

</p>

</body>

</html>

当浏览器遇到一个<script>标签时，正如上面HTML 页面中那样，无法预知JavaScript 是否在<p>标签中添加内容。因此，浏览器停下来，运行此JavaScript 代码，然后再继续解析、翻译页面。同样的事情发生在使用src 属性加载JavaScript 的过程中。浏览器必须首先下载外部文件的代码，这要占用一些时间，然后

解析并运行此代码。此过程中，页面解析和用户交互是被完全阻塞的。

HTML 4 文档指出，一个<script>标签可以放在HTML 文档的<head>或<body>标签中，可以在其中多次出现。传统上，<script>标签用于加载外部JavaScript 文件。<head>部分除此类代码外，还包含<link>标签用于加载外部CSS 文件和其他页面中间件。也就是说，最好把风格和行为所依赖的部分放在一起，首先加

载他们，使得页面可以得到正确的外观和行为。例如：

<html>

<head>

<title>Script Example</title>

<-- Example of inefficient script positioning -->

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

<script type="text/javascript" src="file2.js"></script>

<script type="text/javascript" src="file3.js"></script>

<link rel="stylesheet" type="text/css" href="styles.css">

</head>

<body>

<p>Hello world!</p>

</body>

</html>

虽然这些代码看起来是无害的，但它们确实存在性能问题：在<head>部分加载了三个JavaScript 文件。因为每个<script>标签阻塞了页面的解析过程，直到它完整地下载并运行了外部JavaScript 代码之后，页面处理才能继续进行。用户必须忍受这种可以察觉的延迟。请记住，浏览器在遇到<body>标签之前，不会渲

染页面的任何部分。用这种方法把脚本放在页面的顶端，将导致一个可以察觉的延迟，通常表现为：页面打开时，首先显示为一幅空白的页面，而此时用户即不能阅读，也不能与页面进行交互操作

Internet Explorer 8, Firefox 3.5, Safari 4, 和Chrome 2 允许并行下载JavaScript 文件。这个好消息表明，当一个<script>标签正在下载外部资源时，不必阻塞其他<script>标签。不幸的是，JavaScript 的下载仍然要阻塞其他资源的下载过程，即使脚本之间的下载过程互不阻塞，页面仍旧要等待所有JavaScript代码下载并执行完成之后才能继续。所以，当浏览器通过允许并行下载提高性能之后，该问题并没有完全解决。脚本阻塞仍旧是一个问题。

因为脚本阻塞其他页面资源的下载过程，所以推荐的办法是：将所有<script>标签放在尽可能接近<body>标签底部的位置，尽量减少对整个页面下载的影响。例如：

<html>

<head>

<title>Script Example</title>

<link rel="stylesheet" type="text/css" href="styles.css">

</head>

<body>

<p>Hello world!</p>

<-- Example of recommended script positioning -->

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

<script type="text/javascript" src="file2.js"></script>

<script type="text/javascript" src="file3.js"></script>

</body>

</html>

此代码展示了所推荐的<script>标签在HTML 文件中的位置。尽管脚本下载之间互相阻塞，但页面已经下载完成并且显示在用户面前了，进入页面的速度不会显得太慢。这正是“Yahoo! 优越性能小组”关于JavaScript 的第一条定律：将脚本放在底部。

**Grouping Scripts 成组脚本**

由于每个<script>标签下载时阻塞页面解析过程，所以限制页面的<script>总数也可以改善性能。这个规则对内联脚本和外部脚本同样适用。每当页面解析碰到一个<script>标签时，紧接着有一段时间用于代码执行。最小化这些延迟时间可以改善页面的整体性能。

这个问题与外部JavaScript 文件处理过程略有不同。每个HTTP 请求都会产生额外的性能负担，下载一个100KB 的文件比下载四个25KB 的文件要快。总之，减少引用外部脚本文件的数量。典型的，一个大型网站或网页应用需要多次请求JavaScript 文件。你可以将这些文件整合成一个文件，只需要一个<script>标

签引用，就可以减少性能损失

Yahoo! 为他的“Yahoo! 用户接口（Yahoo! User Interface，YUI）”库创建一个“联合句柄”，这是通过他们的“内容投递网络（Content Delivery Network，CDN）”实现的。任何一个网站可以使用一个“联合句柄”URL指出包含YUI 文件包中的哪些文件。例如，下面的URL 包含两个文件

http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js

此URL 调用2.7.0 版本的yahoo-min.js 和event-min.js 文件。这些文件在服务器上是两个分离的文件，但是当服务器收到此URL 请求时，两个文件将被合并在一起返回给客户端。通过这种方法，就不再需要两个<script>标签（每个标签加载一个文件），一个<script>标签就可以加载他们：

<html>

<head>

<title>Script Example</title>

<link rel="stylesheet" type="text/css" href="styles.css">

</head>

<body>

<p>Hello world!</p>

<-- Example of recommended script positioning -->

<script type="text/javascript"

src="http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js"></s

cript>

</body>

</html>

此代码只有一个<script>标签，位于页面的底部，加载多个JavaScript 文件。这是在HTML 页面中包含多个外部JavaScript 的最佳方法。

**Nonblocking Scripts 非阻塞脚本**

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。有几种方法可以实现这种效果。

**Deferred Scripts 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。

**Dynamic Script Elements 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\_r("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" The default state

“uninitialized”默认状态

"loading" Download has begun

“loading”下载开始

"loaded" Download has completed

“loaded”下载完成

"interactive" Data is completely downloaded but isn't fully available

“interactive”下载完成但尚不可用

"complete" All data is ready to be used

“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\_r("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\_r("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。

**XMLHttpRequest Script Injection XHR 脚本注入**

另一个以非阻塞方式获得脚本的方法是使XMLHttpRequest(XHR)对象将脚本注入到页面中。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）”，。正因为这个原因，大型网页通常不采用XHR 脚本注入技术。

**Recommended Nonblocking Pattern 推荐的非阻塞模式**

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：

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。

The YUI 3 approach

YUI 3 的核心设计理念为：用一个很小的初始代码，下载其余的功能代码。要在页面上使用YUI 3，首先包含YUI 的种子文件：

<script type="text/javascript"

src=http://yui.yahooapis.com/combo?3.0.0/build/yui/yui-min.js></script>

此种子文件大约10KB（gzipped 压缩后6KB）包含从Yahoo! CDN 下载YUI 组件所需的足够功能。举例来说，如果你想使用DOM 功能，你可以指出它的名字（"dom"），传递给YUI 的use()函数，再提供一个回调函数，当代码准备好时这个回调函数将被调用：

YUI().use("dom", function(Y){

Y.DOM.addClass(docment.body, "loaded");

});

这个例子创建了一个新的YUI 实例，然后调用use()函数。种子文件拥有关于文件名和依赖关系的所有

信息，所以指定“dom”实际上建立了一个由正确的依赖文件所组成的“联合句柄”URL，并创建一个动态脚本元素下载并执行这些文件。当所有代码可用时，回调函数被调用，YUI 实例将作为参数传入，使你可以立即使用新下载的功能。

The LazyLoad library

作为一个更通用的工具，Yahoo! Search 的Ryan Grove 创建了LazyLoad 库（参见

http://github.com/rgrove/lazyload/）。LazyLoad 是一个更强大的loadScript()函数。LazyLoad 精缩之后只有大约1.5KB（精缩，而不是用gzip 压缩的）。用法举例如下：

<script type="text/javascript" src="lazyload-min.js"></script>

<script type="text/javascript">

LazyLoad.js("the-rest.js", function(){

Application.init();

});

</script>

LazyLoad 还可以下载多个JavaScript 文件，并保证它们在所有浏览器上都能够按照正确的顺序执行。要加载多个JavaScript 文件，只要调用LazyLoad.js()函数并传递一个URL 队列给它：

<script type="text/javascript" src="lazyload-min.js"></script>

<script type="text/javascript">

LazyLoad.js(["first-file.js", "the-rest.js"], function(){

Application.init();

});

</script>

即使这些文件是在一个非阻塞的方式下使用动态脚本加载，它建议应尽可能减少文件数量。每次下载仍然是一个单独的HTTP 请求，回调函数直到所有文件下载并执行完之后才会运行。

The LABjs library

另一个非阻塞JavaScript 加载库是LABjs（http://labjs.com/），Kyle Simpson 写的一个开源库，由SteveSouders 赞助。此库对加载过程进行更精细的控制，并尝试并行下载尽可能多的代码。LABjs 也相当小，只有4.50KB（精缩，而不是用gzip 压缩的），所以具有最小的页面代码尺寸。用法举例：

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

<script type="text/javascript">

$LAB.script("the-rest.js")

.wait(function(){

Application.init();

});

</script>

$LAB.script()函数用于下载一个JavaScript 文件，$LAB.wait()函数用于指出一个函数，该函数等待文件下载完成并运行之后才会被调用。LABjs 鼓励链操作，每个函数返回一个指向$LAB 对象的引用。要下载多个JavaScript 文件，那么就链入另一个$LAB.script()调用，如下：

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

<script type="text/javascript">

$LAB.script("first-file.js")

.script("the-rest.js")

.wait(function(){

Application.init();

});

</script>

LABjs 的独特之处在于它能够管理依赖关系。一般来说<script>标签意味着每个文件下载（或按顺序，或并行，如前所述），然后按顺序执行。在某些情况下这非常必要，但有时未必。

LABjs 通过wait()函数允许你指定哪些文件应该等待其他文件。在前面的例子中，first-file.js 的代码不保证在the-rest.js 之前运行。为保证这一点，你必须在第一个script()函数之后添加一个wait()调用：

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

<script type="text/javascript">

$LAB.script("first-file.js").wait()

.script("the-rest.js")

.wait(function(){

Application.init();

});

</script>

**第二部分 Data Access 数据访问**

经典计算机科学的一个问题是确定数据应当存放在什么地方，以实现最佳的读写效率。数据存储在哪里，关系到代码运行期间数据被检索到的速度。在JavaScript 中，此问题相对简单，因为数据存储只有少量方式可供选择。正如其他语言那样，数据存储位置关系到访问速度。在JavaScript 中有四种基本的数据访问位置：

Literal values 直接量:

直接量仅仅代表自己，而不存储于特定位置。 JavaScript 的直接量包括：字符串，数字，布尔值，对象，数组，函数，正则表达式，具有特殊意义的空值，以及未定义。

Variables 变量:

开发人员使用var 关键字创建用于存储数据值。

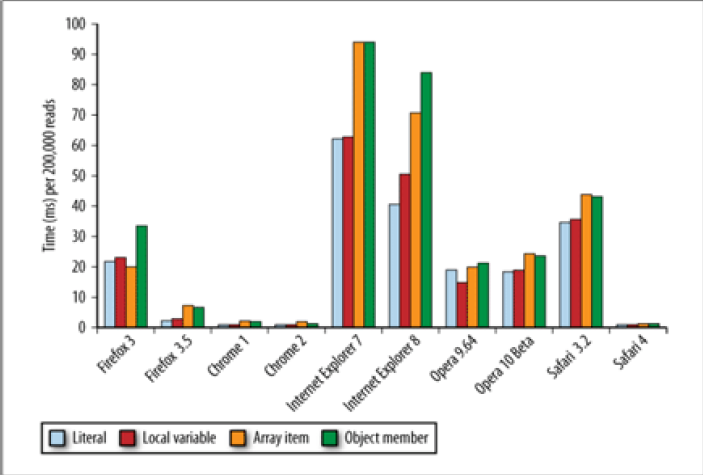
Array items 数组项:

具有数字索引，存储一个JavaScript 数组对象。

Object members 对象成员:

具有字符串索引，存储一个JavaScript 对象。

每一种数据存储位置都具有特定的读写操作负担。大多数情况下，对一个直接量和一个局部变量数据访问的性能差异是微不足道的。访问数组项和对象成员的代价要高一些，具体高多少，很大程度上依赖于浏览器。图显示了不同浏览器中，分别对这四种数据类型进行200'000 次操作所用的时间。



老一些的浏览器使用传统的JavaScript 引擎，如Firefox 3，Internet Explorer 和Safari 3.2，它们比优化后的JavaScript 引擎耗费太多时间。总的来说，直接量和局部变量的访问速度要快于数组项和对象成员的访问速度。只有一个例外，Firefox 3，优化过数组项访问所以非常快。即使如此，一般的建议是，如果关心运行速度，那么尽量使用直接量和局部变量，限制数组项和对象成员的使用。为此，有几种模式来查看、避免并优化你的代码。

**Managing Scope 管理作用域**

作用域概念是理解JavaScript 的关键，不仅从性能的角度，而且从功能的角度。作用域对JavaScript 有许多影响，从确定哪些变量可以被函数访问，到确定this 的值。JavaScript 作用域也关系到性能，但是要理解速度与作用域的关系，首先要理解作用域的工作原理。

Scope Chains and Identifier Resolution 作用域链和标识符解析

每一个JavaScript 函数都被表示为对象。进一步说，它是一个函数实例。函数对象正如其他对象那样，拥有你可以编程访问的属性，和一系列不能被程序访问，仅供JavaScript 引擎使用的内部属性。其中一个内部属性是[[Scope]]，由ECMA-262 标准第三版定义

内部[[Scope]]属性包含一个函数被创建的作用域中对象的集合。此集合被称为函数的作用域链，它决定哪些数据可由函数访问。此函数作用域链中的每个对象被称为一个可变对象，每个可变对象都以“键值对”的形式存在。当一个函数创建后，它的作用域链被填充以对象，这些对象代表创建此函数的环境中可访问

的数据。例如下面这个全局函数：

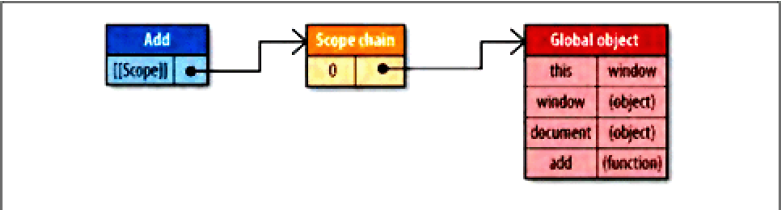
function add(num1, num2){

var sum = num1 + num2;

return sum;

}

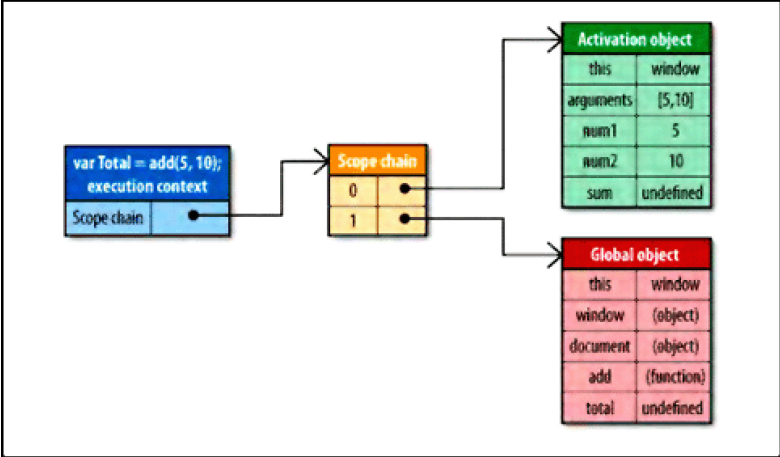
当add()函数创建后，它的作用域链中填入一个单独的可变对象，此全局对象代表了所有全局范围定义的变量。此全局对象包含诸如窗口、浏览器和文档之类的访问接口。图指出它们之间的关系（注意：此图中只画出全局变量中很少的一部分，其他部分还很多）。



add 函数的作用域链将会在运行时用到。假设运行下面的代码：

var total = add(5, 10);运行此add 函数时建立一个内部对象，称作“运行期上下文”。一个运行期上下文定义了一个函数运行时

的环境。对函数的每次运行而言，每个运行期上下文都是独一的，所以多次调用同一个函数就会导致多次创建运行期上下文。当函数执行完毕，运行期上下文就被销毁。一个运行期上下文有它自己的作用域链，用于标识符解析。当运行期上下文被创建时，它的作用域链被初始化，连同运行函数的[[Scope]]属性中所包含的对象。这些值按照它们出现在函数中的顺序，被复制到运行期上下文的作用域链中。这项工作一旦完成，一个被称作“激活对象”的新对象就为运行期上下文创建好了。此激活对象作为函数执行期的一个可变对象，包含访问所有局部变量，命名参数，参数集合，和this的接口。然后，此对象被推入作用域链的前端。当作用域链被销毁时，激活对象也一同销毁。图 显示了前面实例代码所对应的运行期上下文和它的作用域链。



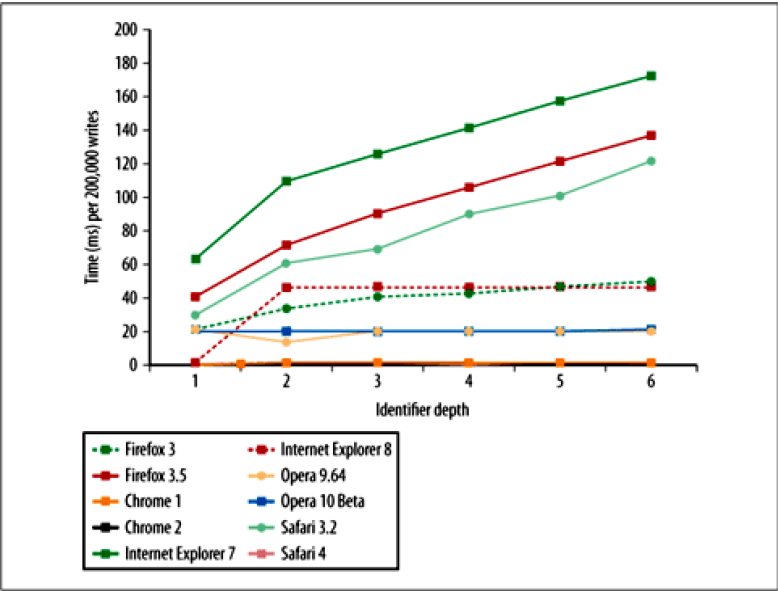
在函数运行过程中，每遇到一个变量，标识符识别过程要决定从哪里获得或者存储数据。此过程搜索运行期上下文的作用域链，查找同名的标识符。搜索工作从运行函数的激活目标之作用域链的前端开始。如果找到了，那么就使用这个具有指定标识符的变量；如果没找到，搜索工作将进入作用域链的下一个对象。

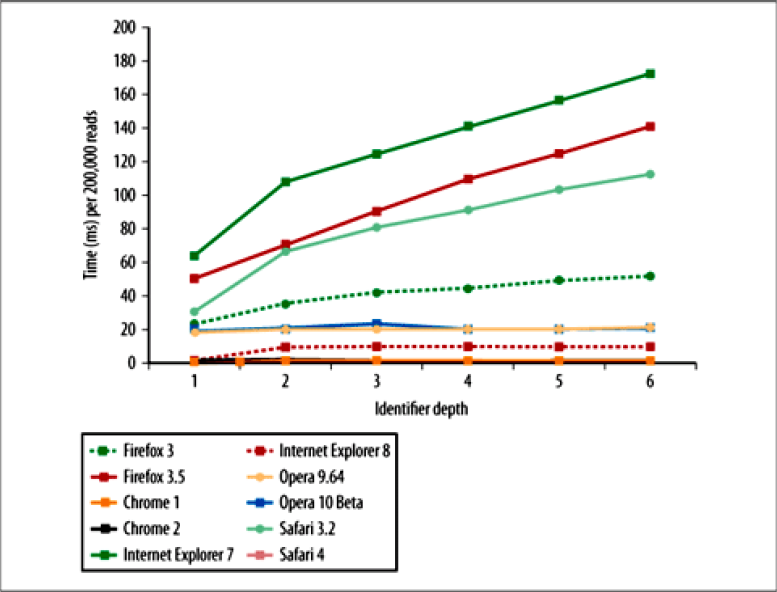
此过程持续运行，直到标识符被找到，或者没有更多对象可用于搜索，这种情况下标识符将被认为是未定义的。函数运行时每个标识符都要经过这样的搜索过程，例如前面的例子中，函数访问sum，num1，num2时都会产生这样的搜索过程。正是这种搜索过程影响了性能。

**Identifier Resolution Performance 标识符识别性能**

标识符识别不是免费的，事实上没有哪种电脑操作可以不产生性能开销。在运行期上下文的作用域链中，一个标识符所处的位置越深，它的读写速度就越慢。所以，函数中局部变量的访问速度总是最快的，而全局变量通常是最慢的（优化的JavaScript 引擎在某些情况下可以改变这种状况）。请记住，全局变量总是

处于运行期上下文作用域链的最后一个位置，所以总是最远才能触及的。图显示了作用域链上不同深度标识符的识别速度，深度为1 表示一个局部变量。





总的趋势是，对所有浏览器来说，一个标识符所处的位置越深，读写它的速度就越慢。采用优化的JavaScript 引擎的浏览器，如Safari 4，访问域外标识符时没有这种性能损失，而Internet Explorer，Safari 3.2，和其他浏览器则有较大幅度的影响。值得注意的是，早期浏览器如Internet Explorer 6 和Firefox 2，有令人

难以置信的陡峭斜坡，如果此图包含它们的数据，曲线高点将超出图表边界。

通过以上信息，在没有优化JavaScript 引擎的浏览器中，最好尽可能使用局部变量。一个好的经验法则是：用局部变量存储本地范围之外的变量值，如果它们在函数中的使用多于一次。考虑下面的例子：

function initUI(){

var bd = document.body,

links = document.getElementsByTagName\_r("a"),

i = 0,

len = links.length;

while(i < len){

update(links[i++]);

}

document.getElementById("go-btn").onclick = function(){

start();

};

bd.className = "active";

}

此函数包含三个对document 的引用，document 是一个全局对象。搜索此变量，必须遍历整个作用域链，直到最后在全局变量对象中找到它。你可以通过这种方法减轻重复的全局变量访问对性能的影响：首先将全局变量的引用存储在一个局部变量中，然后使用这个局部变量代替全局变量。例如，上面的代码可以重

写如下：

function initUI(){

var doc = document,

bd = doc.body,

links = doc.getElementsByTagName\_r("a"),

i = 0,

len = links.length;

while(i < len){

update(links[i++]);

}

doc.getElementById("go-btn").onclick = function(){

start();

};

bd.className = "active";

}

initUI()的新版本首先将document 的引用存入局部变量doc 中。现在访问全局变量的次数是1 次，而不是3 次。用doc 替代document 更快，因为它是一个局部变量。当然，这个简单的函数不会显示出巨大的性能改进，因为数量的原因，不过可以想象一下，如果几十个全局变量被反复访问，那么性能改进将显得多

么出色。

**Scope Chain Augmentation 改变作用域链**

一般来说，一个运行期上下文的作用域链不会被改变。但是，有两种表达式可以在运行时临时改变运行期上下文作用域链。第一个是with 表达式。with 表达式为所有对象属性创建一个默认操作变量。在其他语言中，类似的功能通常用来避免书写一些

重复的代码。initUI()函数可以重写成如下样式：

function initUI(){

with (document){ //avoid!

var bd = body,

links = getElementsByTagName\_r("a"),

i = 0,

len = links.length;

while(i < len){

update(links[i++]);

}

getElementById("go-btn").onclick = function(){

start();

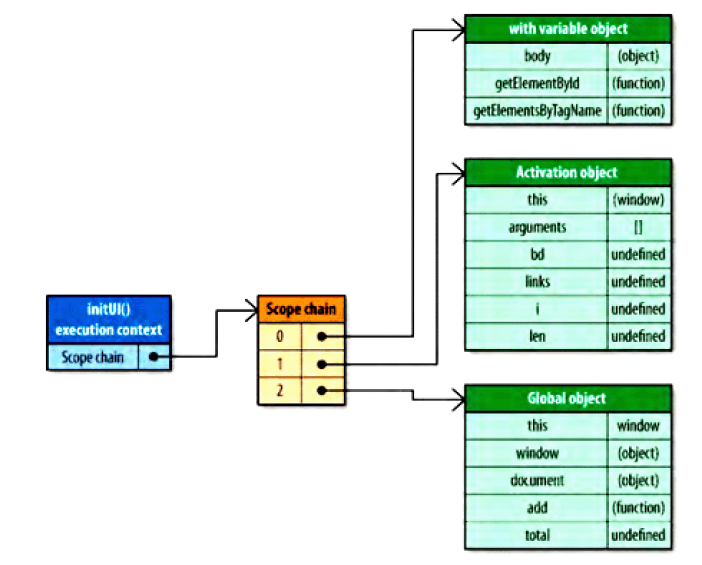
};

bd.className = "active";

}

此重写的initUI()版本使用了一个with 表达式，避免多次书写“document”。这看起来似乎更有效率，而实际上却产生了一个性能问题。

当代码流执行到一个with 表达式时，运行期上下文的作用域链被临时改变了。一个新的可变对象将被创建，它包含指定对象的所有属性。此对象被插入到作用域链的前端，意味着现在函数的所有局部变量都被推入第二个作用域链对象中，所以访问代价更高了



通过将document 对象传递给with 表达式，一个新的可变对象容纳了document 对象的所有属性，被插入到作用域链的前端。这使得访问document 的属性非常快，但是访问局部变量的速度却变慢了，例如bd 变量。正因为这个原因，最好不要使用with 表达式。正如前面提到的，只要简单地将document 存储在一个

局部变量中，就可以获得性能上的提升。

在JavaScript 中不只是with 表达式人为地改变运行期上下文的作用域链，try-catch 表达式的catch 子句具有相同效果。当try 块发生错误时，程序流程自动转入catch 块，并将异常对象推入作用域链前端的一个可变对象中。在catch 块中，函数的所有局部变量现在被放在第二个作用域链对象中。例如：

try {

methodThatMightCauseAnError();

} catch (ex){

alert(ex.message); //scope chain is augmented here

}

请注意，只要catch 子句执行完毕，作用域链就会返回到原来的状态。如果使用得当，try-catch 表达式是非常有用的语句，所以不建议完全避免。如果你计划使用一个try-catch语句，请确保你了解可能发生的错误。一个try-catch 语句不应该作为JavaScript 错误的解决办法。如果你知道一个错误会经常发生，那说明应当修正代码本身的问题。

你可以通过精缩代码的办法最小化catch 子句对性能的影响。一个很好的模式是将错误交给一个专用函数来处理。例子如下：

try {

methodThatMightCauseAnError();

} catch (ex){

handleError(ex); //delegate to handler method

}

handleError()函数是catch 子句中运行的唯一代码。此函数以适当方法自由地处理错误，并接收由错误产生的异常对象。由于只有一条语句，没有局部变量访问，作用域链临时改变就不会影响代码的性能。

**Dynamic Scopes 动态作用域**

无论是with 表达式还是try-catch 表达式的catch 子句，以及包含()的函数，都被认为是动态作用域。一个动态作用域只因代码运行而存在，因此无法通过静态分析（察看代码结构）来确定（是否存在动态作用域）。例如：

function execute(code) {

(code);

function subroutine(){

return window;

}

var w = subroutine();

//what value is w?

};

execute()函数看上去像一个动态作用域，因为它使用了()。w 变量的值与code 有关。大多数情况下，w将等价于全局的window 对象，但是请考虑如下情况：

execute("var window = {};")

这种情况下，()在execute()函数中创建了一个局部window 变量。所以w 将等价于这个局部window 变量而不是全局的那个。所以说，不运行这段代码是没有办法了解具体情况的，标识符window 的确切含义不能预先确定。

优化的JavaScript 引擎，例如Safari 的Nitro 引擎，企图通过分析代码来确定哪些变量应该在任意时刻被访问，来加快标识符识别过程。这些引擎企图避开传统作用域链查找，取代以标识符索引的方式进行快速查找。当涉及一个动态作用域后，此优化方法就不起作用了。引擎需要切回慢速的基于哈希表的标识符

识别方法，更像传统的作用域链搜索。

正因为这个原因，只在绝对必要时才推荐使用动态作用域。

**Closures, Scope, and Memory 闭包，作用域，和内存**

当assignEvents()被执行时，一个激活对象被创建，并包含了一些应有的内容，其中包括id 变量。它将成为运行期上下文作用域链上的第一个对象，全局对象是第二个。当闭包创建时，[[Scope]]属性与这些对象一起被初始化。闭包是JavaScript 最强大的一个方面，它允许函数访问局部范围之外的数据。闭包的使用通过DouglasCrockford 的著作流行起来，当今在最复杂的网页应用中无处不在。不过，有一种性能影响与闭包有关。为了解与闭包有关的性能问题，考虑下面的例子：

function assignEvents(){

var id = "xdi9592";

document.getElementById("save-btn").onclick = function(event){

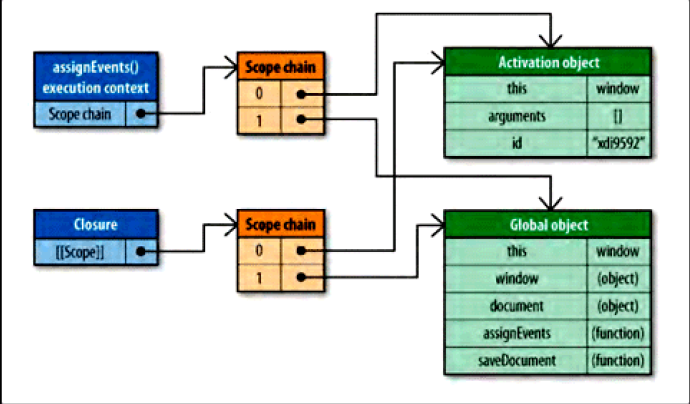
saveDocument(id);

};

}

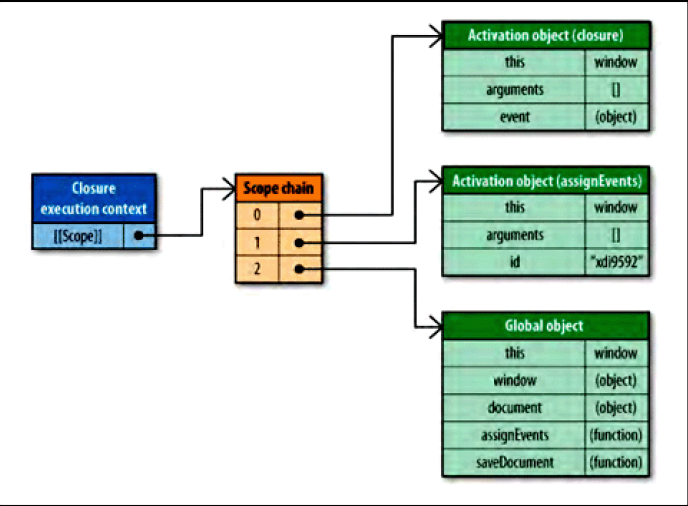
assignEvents()函数为一个DOM 元素指定了一个事件处理句柄。此事件处理句柄是一个闭包，当assignEvents()执行时创建，可以访问其范围内部的id 变量。用这种方法封闭对id 变量的访问，必须创建一个特定的作用域链。

当assignEvents()被执行时，一个激活对象被创建，并包含了一些应有的内容，其中包括id 变量。它将成为运行期上下文作用域链上的第一个对象，全局对象是第二个。当闭包创建时，[[Scope]]属性与这些对象一起被初始化



由于闭包的[[Scope]]属性包含与运行期上下文作用域链相同的对象引用，会产生副作用。通常，一个函数的激活对象与运行期上下文一同销毁。当涉及闭包时，激活对象就无法销毁了，因为引用仍然存在于闭包的[[Scope]]属性中。这意味着脚本中的闭包与非闭包函数相比，需要更多内存开销。在大型网页应用中，这可能是个问题，尤其在Internet Explorer 中更被关注。IE 使用非本地JavaScript 对象实现DOM 对象，闭包可能导致内存泄露

当闭包被执行时，一个运行期上下文将被创建，它的作用域链与[[Scope]]中引用的两个相同的作用域链同时被初始化，然后一个新的激活对象为闭包自身被创建



注意闭包中使用的两个标识符，id 和saveDocument，存在于作用域链第一个对象之后的位置上。这是闭包最主要的性能关注点：你经常访问一些范围之外的标识符，每次访问都导致一些性能损失。在脚本中最好是小心地使用闭包，内存和运行速度都值得被关注。但是，你可以通过早先讨论过的关于域外变量的处理建议，减轻对运行速度的影响：将常用的域外变量存入局部变量中，然后直接访问局部变量。

**Object Members 对象成员；**

大多数JavaScript 代码以面向对象的形式编写。无论通过创建自定义对象还是使用内置的对象，诸如文档对象模型（DOM）和浏览器对象模型（BOM）之中的对象。因此，存在很多对象成员访问。对象成员包括属性和方法，在JavaScript 中，二者差别甚微。对象的一个命名成员可以包含任何数据类型。既然函数也是一种对象，那么对象成员除传统数据类型外，也可以包含一个函数。当一个命名成员引用了一个函数时，它被称作一个“方法”，而一个非函数类型的数据则被称作“属性”。对象成员比直接量或局部变量访问速度慢，在某些浏览器上比访问数组项还

要慢。要理解此中的原因，首先要理解JavaScript 中对象的性质。

Prototypes 原形:

JavaScript 中的对象是基于原形的。原形是其他对象的基础，定义并实现了一个新对象所必须具有的成员。这一概念完全不同于传统面向对象编程中“类”的概念，它定义了创建新对象的过程。原形对象为所有给定类型的对象实例所共享，因此所有实例共享原形对象的成员。

一个对象通过一个内部属性绑定到它的原形。Firefox，Safari，和Chrome 向开发人员开放这一属性，称作\_\_proto\_\_；其他浏览器不允许脚本访问这一属性。任何时候你创建一个内置类型的实例，如Object 或Array，这些实例自动拥有一个Object 作为它们的原形。因此，对象可以有两种类型的成员：实例成员（也称作“own”成员）和原形成员。实例成员直接存在于实例自身，而原形成员则从对象原形继承。考虑下面的例子：

var book = {

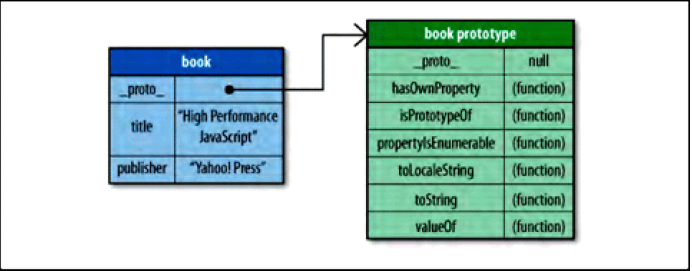
title: "High Performance JavaScript",

publisher: "Yahoo! Press"

};

alert(book.toString()); //"[object Object]"

此代码中，book 对象有两个实例成员：title 和publisher。注意它并没有定义toString()接口，但是这个接口却被调用了，也没有抛出错误。toString()函数就是一个book 对象继承的原形成员。图显示出它们之间的关系。



处理对象成员的过程与变量处理十分相似。当book.toString()被调用时，对成员进行名为“toString”的搜索，首先从对象实例开始，如果book 没有名为toString 的成员，那么就转向搜索原形对象，在那里发现了toString()方法并执行它。通过这种方法，booke 可以访问它的原形所拥有的每个属性或方法。

你可以使用hasOwnProperty()函数确定一个对象是否具有特定名称的实例成员，（它的参数就是成员名称）。要确定对象是否具有某个名称的属性，你可以使用操作符in。例如：

var book = {

title: "High Performance JavaScript",

publisher: "Yahoo! Press"

};

alert(book.hasOwnProperty("title")); //true

alert(book.hasOwnProperty("toString")); //false

alert("title" in book); //true

alert("toString" in book); //true

此代码中，hasOwnProperty()传入“title”时返回true，因为title 是一个实例成员。传入“toString”时返回false，因为toString 不在实例之中。如果使用in 操作符检测这两个属性，那么返回都是true，因为它既搜索实例又搜索原形。

**Prototype Chains 原形链**

对象的原形决定了一个实例的类型。默认情况下，所有对象都是Object 的实例，并继承了所有基本方法，如toString()。你可以用“构造器”创建另外一种类型的原形。例如：

function Book(title, publisher){

this.title = title;

this.publisher = publisher;

}

Book.prototype.sayTitle = function(){

alert(this.title);

};

var book1 = new Book("High Performance JavaScript", "Yahoo! Press");

var book2 = new Book("JavaScript: The Good Parts", "Yahoo! Press");

alert(book1 instanceof Book); //true

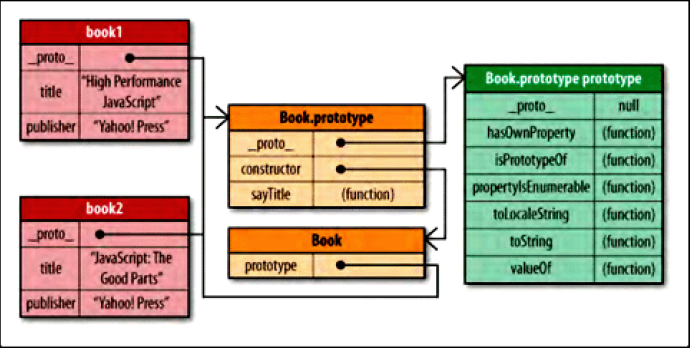
alert(book1 instanceof Object); //true

book1.sayTitle(); //"High Performance JavaScript"

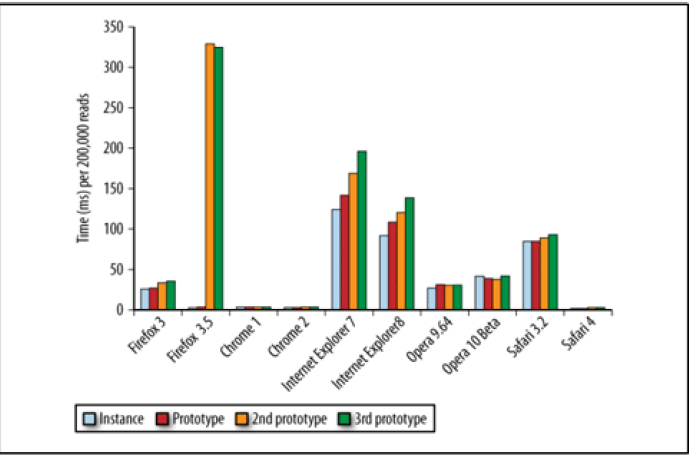
alert(book1.toString()); //"[object Object]"

Book 构造器用于创建一个新的Book 实例。book1 的原形（\_\_proto\_\_）是Book.prototype，Book.prototype

的原形是Object。这就创建了一个原形链，book1 和book2 继承了它们的成员



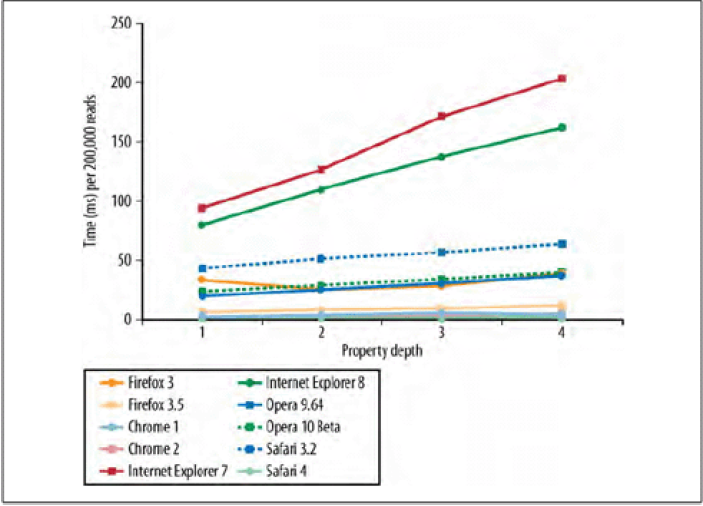
注意，两个Book 实例共享同一个原形链。每个实例拥有自己的title 和publisher 属性，但其他成员均继承自原形。当book1.toString()被调用时，搜索工作必须深入原形链才能找到对象成员“toString”。正如你所怀疑的那样，深入原形链越深，搜索的速度就会越慢。图 显示出成员在原形链中所处的深度与访问时间的关系。



虽然使用优化JavaScript 引擎的新式浏览器在此任务中表现良好，但是老的浏览器，特别是InternetExplorer 和Firefox 3.5，每深入原形链一层都会增加性能损失。记住，搜索实例成员的过程比访问直接量或者局部变量负担更重，所以增加遍历原形链的开销正好放大了这种效果。

**Nested Members 嵌套成员**

由于对象成员可能包含其它成员，例如不太常见的写法window.location.href 这种模式。每遇到一个点号，JavaScript 引擎就要在对象成员上执行一次解析过程。图显示出对象成员深度与访问时间的关系。



结果并不奇怪，成员嵌套越深，访问速度越慢。location.href 总是快于window.location.href，而后者也要

比window.location.href.toString()更快。如果这些属性不是对象的实例属性，那么成员解析还要在每个点上搜索原形链，这将需要更长时间

**Caching Object Member Values 缓存对象成员的值**

由于所有这些性能问题与对象成员有关，所以如果可能的话请避免使用它们。更确切地说，你应当小心地，只在必要情况下使用对象成员。例如，没有理由在一个函数中多次读取同一个对象成员的值：

function hasEitherClass(element, className1, className2){

return element.className == className1 || element.className == className2;

}

在此代码中，element.className 被访问了两次。很明显，在这个函数过程中它的值是不会改变的，但仍然引起两次对象成员搜索过程。你可以将它的值存入一个局部变量，消除一次搜索过程。修改如下：

function hasEitherClass(element, className1, className2){

var currentClassName = element.className;

return currentClassName == className1 || currentClassName == className2;

}

此重写后的版本中成员搜索只进行了一次。既然两次对象搜索都在读属性值，所以有理由只读一次并将值存入局部变量中。局部变量的访问速度要快得多。

一般来说，如果在同一个函数中你要多次读取同一个对象属性，最好将它存入一个局部变量。以局部变量替代属性，避免多余的属性查找带来性能开销。在处理嵌套对象成员时这点特别重要，它们会对运行速度产生难以置信的影响。

JavaScript 的命名空间，如YUI 所使用的技术，是经常访问嵌套属性的来源之一。例如：

function toggle(element){

if (YAHOO.util.Dom.hasClass(element, "selected")){

YAHOO.util.Dom.removeClass(element, "selected");

return false;

} else {

YAHOO.util.Dom.addClass(element, "selected");

return true;

}

}

此代码重复YAHOO.util.Dom 三次以获得三种不同的方法。每个方法都产生三次成员搜索过程，总共九次，导致此代码相当低效。一个更好的方法是将YAHOO.util.Dom 存储在局部变量中，然后访问局部变量：

function toggle(element){

var Dom = YAHOO.util.Dom;

if (Dom.hasClass(element, "selected")){

Dom.removeClass(element, "selected");

return false;

} else {

Dom.addClass(element, "selected");

return true;

}

}

总的成员搜索次数从九次减少到五次。在一个函数中，你绝不应该对一个对象成员进行超过一次搜索，除非该值可能改变。

**总结：**

**在JavaScript 中，数据存储位置可以对代码整体性能产生重要影响。有四种数据访问类型：直接量，变量，数组项，对象成员。它们有不同的性能考虑。**

**直接量和局部变量访问速度非常快，数组项和对象成员需要更长时间。**

**局部变量比域外变量快，因为它位于作用域链的第一个对象中。变量在作用域链中的位置越深，访问所需**

**的时间就越长。全局变量总是最慢的，因为它们总是位于作用域链的最后一环。**

**避免使用with 表达式，因为它改变了运行期上下文的作用域链。而且应当小心对待try-catch 表达式的catch子句，因为它具有同样效果。**

**嵌套对象成员会造成重大性能影响，尽量少用。**

**一个属性或方法在原形链中的位置越深，访问它的速度就越慢。**

**一般来说，你可以通过这种方法提高JavaScript 代码的性能：将经常使用的对象成员，数组项，和域外变量存入局部变量中。然后，访问局部变量的速度会快于那些原始变量。**

**通过使用这些策略，你可以极大地提高那些需要大量JavaScript 代码的网页应用的实际性能。**