*High Performance JavaScript*
*Build Faster Web Application Interfaces*

# 高性能 JavaScript

*Nicholas C. Zakas* 著
丁琛 译 赵泽欣 审校

O'REILLY® | YAHOO! PRESS
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

# 目录

# 第一章　Loading and Execution　加载和运行

JavaScript performance in the browser is arguably the most important usability issue facing developers. The problem is complex because of the blocking nature of JavaScript, which is to say that nothing else can happen while JavaScript code is being executed. In fact, most browsers use a single process for both user interface (UI) updates and JavaScript execution, so only one can happen at any given moment in time. The longer JavaScript takes to execute, the longer it takes before the browser is free to respond to user input.

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。

On a basic level, this means that the very presence of a <script> tag is enough to make the page wait for the script to be parsed and executed. Whether the actual JavaScript code is inline with the tag or included in an external file is irrelevant; the page download and rendering must stop and wait for the script to complete before proceeding. This is a necessary part of the page's life cycle because the script may cause changes to the page while executing. The typical example is using document.write() in the middle of a page (as often used by advertisements). For example:

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>
```

When the browser encounters a <script> tag, as in this HTML page, there is no way of knowing whether the JavaScript will insert content into the <p>, introduce additional elements, or perhaps even close the tag. Therefore, the browser stops processing the page as it comes in, executes the JavaScript code, then continues parsing and rendering the page. The same takes place for JavaScript loaded using the src attribute; the browser must first download the code from the external file, which takes time, and then parse and execute the code. Page rendering and user interaction are completely blocked during this time.

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。

**Script Positioning 脚本位置**

The HTML 4 specification indicates that a <script> tag may be placed inside of a <head> or <body> tag in an HTML document and may appear any number of times within each. Traditionally, script> tags that are used to load external JavaScript files have appeared in the <head>, along with <link> tags to load external CSS files and other metainformation about the page. The theory was that it's best to keep as many style and behavior dependencies together, loading them first so that the page will come in looking and behaving correctly. For example:

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>
```

Though this code seems innocuous, it actually has a severe performance issue: there are three JavaScript files being loaded in the <head>. Since each <script> tag blocks the page from continuing to render until it has fully downloaded and executed the JavaScript code, the perceived performance of this page will suffer. Keep in mind that browsers don't start rendering anything on the page until the opening <body> tag is encountered. Putting scripts at the top of the page in this way typically leads to a noticeable delay, often in the form of a blank white page, before the user can even begin reading or otherwise interacting with the page. To get a good understanding of how this occurs, it's useful to look at a waterfall diagram showing when each resource is downloaded. Figure 1-1 shows when each script and the stylesheet file get downloaded as the page is loading.

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。为了更好地理解此过程，我们使用瀑布图来描绘每个资源的下载过程。图 1-1 显示出页面加载过程中，每个脚本文件和样式表文件下载的过程。

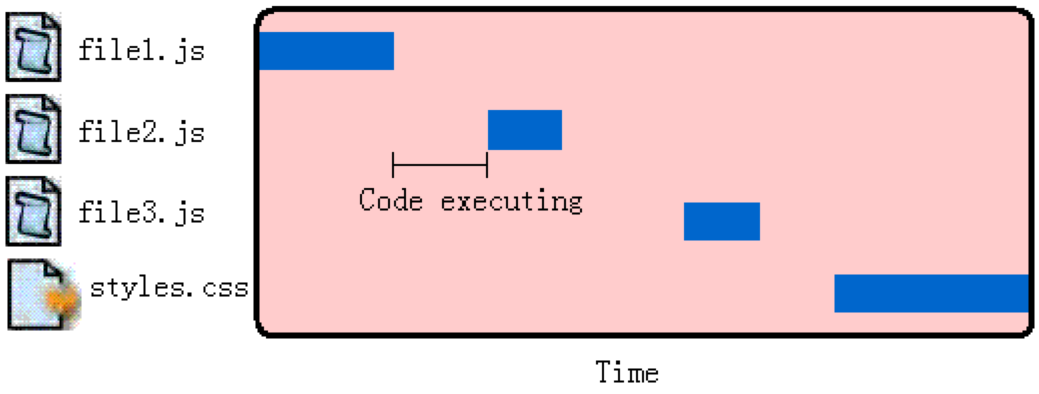


Figure 1-1. JavaScript code execution blocks other file downloads

图 1-1，JavaScript 代码运行过程阻塞其他文件下载

Figure 1-1 shows an interesting pattern. The first JavaScript file begins to download and blocks any of the other files from downloading in the meantime. Further, there is a delay between the time at which file1.js is completely downloaded and the time at which file2.js begins to download. That space is the time it takes for the code contained in file1.js to fully execute. Each file must wait until the previous one has been downloaded and executed before the next download can begin. In the meantime, the user is met with a blank screen as the files are being downloaded one at a time. This is the behavior of most major browsers today.

图 1-1 是一个令人感兴趣的模板。第一个 JavaScript 文件开始下载，并阻塞了其他文件的下载过程。进一步，在 file1.js 下载完之后和 file2.js 开始下载之前有一个延时，这是 file1.js 完全运行所需的时间。每个文件必须等待前一个文件下载完成并运行完之后，才能开始自己的下载过程。当这些文件下载时，用户面对一个空白的屏幕。这就是今天大多数浏览器的行为模式。

Internet Explorer 8, Firefox 3.5, Safari 4, and Chrome 2 all allow parallel downloads of JavaScript files. This is good news because the <script> tags don't necessarily block other <script> tags from downloading external resources. Unfortunately, JavaScript downloads still block downloading of other resources, such as images. And even though downloading a script doesn't block other scripts from downloading, the page must still wait for the JavaScript code to be downloaded and executed before continuing. So while the latest browsers have improved performance by allowing parallel downloads, the problem hasn't been completely solved. Script blocking still remains a problem.

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，例如图片。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。

Because scripts block downloading of all resource types on the page, it's recommended to place all <script> tags as close to the bottom of the <body> tag as possible so as not to affect the download of the entire page. For example:

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>
```

This code represents the recommended position for <script> tags in an HTML file. Even though the script downloads will block one another, the rest of the page has already been downloaded and displayed to the user so that the entire page isn't perceived as slow. This is the Yahoo! Exceptional Performance team's first rule about JavaScript: put scripts at the bottom.

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**

Since each <script> tag blocks the page from rendering during initial download, it's helpful to limit the total number of <script> tags contained in the page. This applies to both inline scripts as well as those in external files. Every time a <script> tag is encountered during the parsing of an HTML page, there is going to be a delay while the code is executed; minimizing these delays improves the overall performance of the page.

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。

The problem is slightly different when dealing with external JavaScript files. Each HTTP request brings with it additional performance overhead, so downloading one single 100 KB file will be faster than downloading four 25 KB files. To that end, it's helpful to limit the number of external script files that your page references. Typically, a large website or web application will have several required JavaScript files. You can minimize the performance impact by concatenating these files together into a single file and then calling that single file with a single <script> tag. The concatenation can happen offline using a build tool (discussed in Chapter 9) or in real-time using a tool such as the Yahoo! combo handler.

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。这一系列的工作可通过一个打包工具实现（我们在第 9 章讨论），或者一个实时工具，诸如“Yahoo! combo handler”。

Yahoo! created the combo handler for use in distributing the Yahoo! User Interface (YUI) library files through their Content Delivery Network (CDN). Any website can pull in any number of YUI files by using a combo-handled URL and specifying the files to include. For example, this URL includes two files:

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

This URL loads the 2.7.0 versions of the yahoo-min.js and event-min.js files. These files exist separately on the server but are combined when this URL is requested. Instead of using two <script> tags (one to load each file), a single <script> tag can be used to load both:

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"></script>
  </body>
</html>
```

This code has a single <script> tag at the bottom of the page that loads multiple JavaScript files, showing the best practice for including external JavaScript on an HTML page.

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's tendency to block browser processes, both HTTP requests and UI updates, is the most notable performance issue facing developers. Keeping JavaScript files small and limiting the number of HTTP requests are only the first steps in creating a responsive web application. The richer the functionality an application requires, the more JavaScript code is required, and so keeping source code small isn't always an option. Limiting yourself to downloading a single large JavaScript file will only result in locking the browser out for a long period of time, despite it being just one HTTP request. To get around this situation, you need to incrementally add more JavaScript to the page in a way that doesn't block the browser.

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。

The secret to nonblocking scripts is to load the JavaScript source code after the page has finished loading. In technical terms, this means downloading the code after the window's load event has been fired. There are a few techniques for achieving this result.

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 defines an additional attribute for the <script> tag called defer. The defer attribute indicates that the script contained within the element is not going to modify the DOM and therefore execution can be safely deferred until a later point in time. The defer attribute is supported only in Internet Explorer 4+ and Firefox 3.5+, making it less than ideal for a generic cross-browser solution. In other browsers, the defer attribute is simply ignored and so the <script> tag is treated in the default (blocking) manner. Still, this solution is useful if your target browsers support it. The following is an example usage:

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

A <script> tag with defer may be placed anywhere in the document. The JavaScript file will begin downloading at the point that the <script> tag is parsed, but the code will not be executed until the DOM has been completely loaded (before the onload event handler is called). When a deferred JavaScript file is downloaded, it doesn't block the browser's other processes, and so these files can be downloaded in parallel with others on the page.

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。

Any <script> element marked with defer will not execute until after the DOM has been completely loaded; this holds true for inline scripts as well as for external script files. The following simple page demonstrates how the defer attribute alters the behavior of scripts:

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>
```

This code displays three alerts as the page is being processed. In browsers that don't support defer, the order of the alerts is “defer”, “script”, and “load”. In browsers that support defer, the order of the alerts is “script”, “defer”, and “load”. Note that the deferred <script> element isn't executed until after the second but is executed before the onload event handler is called.

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。

If your target browsers include only Internet Explorer and Firefox 3.5, then deferring scripts in this manner can be helpful. If you have a larger cross-section of browsers to support, there are other solutions that work in a more consistent manner.

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

The Document Object Model (DOM) allows you to dynamically create almost any part of an HTML document using JavaScript. At its root, the <script> element isn't any different than any other element on a page: references can be retrieved through the DOM, and they can be moved, removed from the document, and even created. A new <script> element can be created very easily using standard DOM methods:

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);
```

This new <script> element loads the source file file1.js. The file begins downloading as soon as the element is added to the page. The important thing about this technique is that the file is downloaded and executed without blocking other page processes, regardless of where the download is initiated. You can even place this code in the <head> of a document without affecting the rest of the page (aside from the one HTTP connection that is used to download the file).

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）。

When a file is downloaded using a dynamic script node, the retrieved code is typically executed immediately (except in Firefox and Opera, which will wait until any previous dynamic script nodes have executed). This works well when the script is self-executing but can be problematic if the code contains only interfaces to be used by other scripts on the page. In that case, you need to track when the code has been fully downloaded and is ready for use. This is accomplished using events that are fired by the dynamic <script> node.

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, Chrome, and Safari 3+ all fire a load event when the src of a <script> element has been retrieved. You can therefore be notified when the script is ready by listening for this event:

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 supports an alternate implementation that fires a readystatechange event. There is a readyState property on the <script> element that is changed at various times during the download of an external file. There are five possible values for readyState:

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

Microsoft's documentation for readyState and each of the possible values seems to indicate that not all states will be used during the lifetime of the <script> element, but there is no indication as to which will always be used. In practice, the two states of most interest are "loaded" and "complete". Internet Explorer is inconsistent with which of these two readyState values indicates the final state, as sometimes the <script> element will reach the "loaded" state but never reach "complete" whereas other times "complete" will be reached without "loaded" ever having been used. The safest way to use the readystatechange event is to check for both of these states and remove the event handler when either one occurs (to ensure the event isn't handled twice):

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);
```

In most cases, you'll want to use a single approach to dynamically load JavaScript files. The following function encapsulates both the standard and IE-specific functionality:

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);
}
```

This function accepts two arguments: the URL of the JavaScript file to retrieve and a callback function to execute when the JavaScript has been fully loaded. Feature detection is used to determine which event handler should monitor the script's progress. The last step is to assign the src property and add the <script> element to the page. The loadScript() function is used as follows:

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!");
});
```

You can dynamically load as many JavaScript files as necessary on a page, but make sure you consider the order in which files must be loaded. Of all the major browsers, only Firefox and Opera guarantee that the order of script execution will remain the same as you specify. Other browsers will download and execute the various code files in the order in which they are returned from the server. You can guarantee the order by chaining the downloads together, such as:

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!");
    });
  });
});
```

This code waits to begin loading file2.js until file1.js is available and also waits to download file3.js until file2.js is available. Though possible, this approach can get a little bit difficult to manage if there are multiple files to download and execute.

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。

If the order of multiple files is important, the preferred approach is to concatenate the files into a single file where each part is in the correct order. That single file can then be downloaded to retrieve all of the code at once (since this is happening asynchronously, there's no penalty for having a larger file).

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）。

Dynamic script loading is the most frequently used pattern for nonblocking JavaScript downloads due to its cross-browser compatibility and ease of use.

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

Another approach to nonblocking scripts is to retrieve the JavaScript code using an XMLHttpRequest (XHR) object and then inject the script into the page. This technique involves creating an XHR object, downloading the JavaScript file, then injecting the JavaScript code into the page using a dynamic <script> element. Here's a simple example:

另一个以非阻塞方式获得脚本的方法是使用 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);
```

This code sends a GET request for the file file1.js. The onreadystatechange event handler checks for a readyState of 4 and then verifies that the HTTP status code is valid (anything in the 200 range means a valid response, and 304 means a cached response). If a valid response has been received, then a new <script> element is created and its text property is assigned to the responseText received from the server. Doing so essentially creates a <script> element with inline code. Once the new <script> element is added to the document, the code is executed and is ready to use.

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。

The primary advantage of this approach is that you can download the JavaScript code without executing it immediately. Since the code is being returned outside of a <script> tag, it won't automatically be executed upon download, allowing you to defer its execution until you're ready. Another advantage is that the same code works in all modern browsers without exception cases.

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。

The primary limitation of this approach is that the JavaScript file must be located on the same domain as the page requesting it, which makes downloading from CDNs impossible. For this reason, XHR script injection typically isn't used on large-scale web applications.

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）”，前面 002 篇《成组脚本》一节提到）。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

The recommend approach to loading a significant amount of JavaScript onto a page is a two-step process: first, include the code necessary to dynamically load JavaScript, and then load the rest of the JavaScript code needed for page initialization. Since the first part of the code is as small as possible, potentially containing just the loadScript() function, it downloads and executes quickly, and so shouldn't cause much interference with the page. Once the initial code is in place, use it to load the remaining JavaScript. For example:

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>
```

Place this loading code just before the closing </body> tag. Doing so has several benefits. First, as discussed earlier, this ensures that JavaScript execution won't prevent the rest of the page from being displayed. Second, when the second JavaScript file has finished downloading, all of the DOM necessary for the application has been created and is ready to be interacted with, avoiding the need to check for another event (such as window.onload) to know when the page is ready for initialization.

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。

Another option is to embed the loadScript() function directly into the page, thus avoiding another HTTP request. For example:

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>
```

If you decide to take the latter approach, it's recommended to minify the initial script using a tool such as YUI Compressor (see Chapter 9) for the smallest byte-size impact on your page.

如果你决定使用这种方法，建议你使用“YUI Compressor”(参见第 9 章)或者类似的工具将初始化脚本缩小到最小字节尺寸。

Once the code for page initialization has been completely downloaded, you are free to continue using loadScript() to load additional functionality onto the page as needed.

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

The concept of a small initial amount of code on the page followed by downloading additional functionality is at the core of the YUI 3 design. To use YUI 3 on your page, begin by including the YUI seed file:

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>
```

The seed file is around 10 KB (6 KB gzipped) and includes enough functionality to download any other YUI components from the Yahoo! CDN. For example, if you'd like to use the DOM utility, you specify its name ("dom") with the YUI use() method and then provide a callback that will be executed when the code is ready:

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");
});
```

This example creates a new instance of the YUI object and then calls the use() method. The seed file has all of the information about filenames and dependencies, so specifying "dom" actually builds up a combo-handler URL with all of the correct dependency files and creates a dynamic script element to download and execute those files. When all of the code is available, the callback method is called and the YUI instance is passed in as the argument, allowing you to immediately start using the newly downloaded functionality.

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

For a more general-purpose tool, Ryan Grove of Yahoo! Search created the LazyLoad library (available at http://github.com/rgrove/lazyload/). LazyLoad is a more powerful version of the loadScript() function. When minified, the LazyLoad file is around 1.5 KB (minified, not gzipped). Example usage:

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 is also capable of downloading multiple JavaScript files and ensuring that they are executed in the correct order in all browsers. To load multiple JavaScript files, just pass an array of URLs to the LazyLoad.js() method:

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>
```

Even though the files are downloaded in a nonblocking fashion using dynamic script loading, it's recommended to have as few files as possible. Each download is still a separate HTTP request, and the callback function won't execute until all of the files have been downloaded and executed.

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**

Another take on nonblocking JavaScript loading is LABjs (http://labjs.com/), an open source library written by Kyle Simpson with input from Steve Souders. This library provides more fine-grained control over the loading process and tries to download as much code in parallel as possible. LABjs is also quite small, 4.5 KB (minified, not gzipped), and so has a minimal page footprint. Example usage:

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 Steve Souders 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>
```

The $LAB.script() method is used to define a JavaScript file to download, whereas $LAB.wait() is used to indicate that execution should wait until the file is downloaded and executed before running the given function.

LABjs encourages chaining, so every method returns a reference to the $LAB object. To download multiple JavaScript files, just chain another $LAB.script() call:

$LAB.script()函数用于下载一个 JavaScript 文件，$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>
```

What sets LABjs apart is its ability to manage dependencies. Normal inclusion with <script> tags means that each file is downloaded (either sequentially or in parallel, as mentioned previously) and then executed sequentially. In some cases this is truly necessary, but in others it is not.

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 allows you to specify which files should wait for others by using wait(). In the previous example, the code in first-file.js is not guaranteed to execute before the code in the-rest.js. To guarantee this, you must add a wait() call after the first script():

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>
```

Now the code in first-file.js is guaranteed to execute before the code in the-rest.js, although the contents of the files are downloaded in parallel.

现在，first-file.js 的代码保证会在 the-rest.js 之前执行，虽然两个文件的内容是并行下载的。

## Summary 总结

Managing JavaScript in the browser is tricky because code execution blocks other browser processes such as UI painting. Every time a <script> tag is encountered, the page must stop and wait for the code to download (if external) and execute before continuing to process the rest of the page. There are, however, several ways to minimize the performance impact of JavaScript:

管理浏览器中的 JavaScript 代码是个棘手的问题，因为代码执行阻塞了其他浏览器处理过程，诸如用户界面绘制。每次遇到<script>标签，页面必须停下来等待代码下载（如果是外部的）并执行，然后再继续处理页面其他部分。但是，有几种方法可以减少 JavaScript 对性能的影响：

• Put all <script> tags at the bottom of the page, just inside of the closing </body> tag. This ensures that the page can be almost completely rendered before script execution begins.

将所有<script>标签放置在页面的底部，紧靠 body 关闭标签</body>的上方。此法可以保证页面在脚本运行之前完成解析。

• Group scripts together. The fewer <script> tags on the page, the faster the page can be loaded and become interactive. This holds true both for <script> tags loading external JavaScript files and those with inline code.

将脚本成组打包。页面的<script>标签越少，页面的加载速度就越快，响应也更加迅速。不论外部脚本文件还是内联代码都是如此。

• There are several ways to download JavaScript in a nonblocking fashion:

— Use the defer attribute of the <script> tag (Internet Explorer and Firefox 3.5+ only)

— Dynamically create <script> elements to download and execute the code

— Download the JavaScript code using an XHR object, and then inject the code into the page

有几种方法可以使用非阻塞方式下载 JavaScript：

——为<script>标签添加 defer 属性（只适用于 Internet Explorer 和 Firefox 3.5 以上版本）

——动态创建<script>元素，用它下载并执行代码

——用 XHR 对象下载代码，并注入到页面中

By using these strategies, you can greatly improve the perceived performance of a web application that requires a large amount of JavaScript code.

通过使用上述策略，你可以极大提高那些大量使用 JavaScript 代码的网页应用的实际性能。

# 第二章 Data Access 数据访问

One of the classic computer science problems is determining where data should be stored for optimal reading and writing. Where data is stored is related to how quickly it can be retrieved during code execution. This problem in JavaScript is somewhat simplified because of the small number of options for data storage. Similar to other languages, though, where data is stored can greatly affect how quickly it can be accessed later. There are four basic places from which data can be accessed in JavaScript:

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

Any value that represents just itself and isn't stored in a particular location. JavaScript can represent strings, numbers, Booleans, objects, arrays, functions, regular expressions, and the special values null and undefined as literals.

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

Any developer-defined location for storing data created by using the var keyword.

开发人员使用 var 关键字创建用于存储数据值。

Array items 数组项

A numerically indexed location within a JavaScript Array object.

具有数字索引，存储一个 JavaScript 数组对象。

Object members 对象成员

A string-indexed location within a JavaScript object.

具有字符串索引，存储一个 JavaScript 对象。

Each of these data storage locations has a particular cost associated with reading and writing operations involving the data. In most cases, the performance difference between accessing information from a literal value versus a local variable is trivial. Accessing information from array items and object members is more expensive, though exactly which is more expensive depends heavily on the browser. Figure 2-1 shows the relative speed of accessing 200,000 values from each of these four locations in various browsers.

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 2-1 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。

Older browsers using more traditional JavaScript engines, such as Firefox 3, Internet Explorer, and Safari 3.2, show a much larger amount of time taken to access values versus browsers that use optimizing JavaScript engines. The general trends, however, remain the same across all browsers: literal value and local variable access tend to be faster than array item and object member access. The one exception, Firefox 3, optimized array item access to be much faster. Even so, the general advice is to use literal values and local variables whenever possible and limit use of array items and object members where speed of execution is a concern. To that end, there are several patterns to look for, avoid, and optimize in your code.

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。

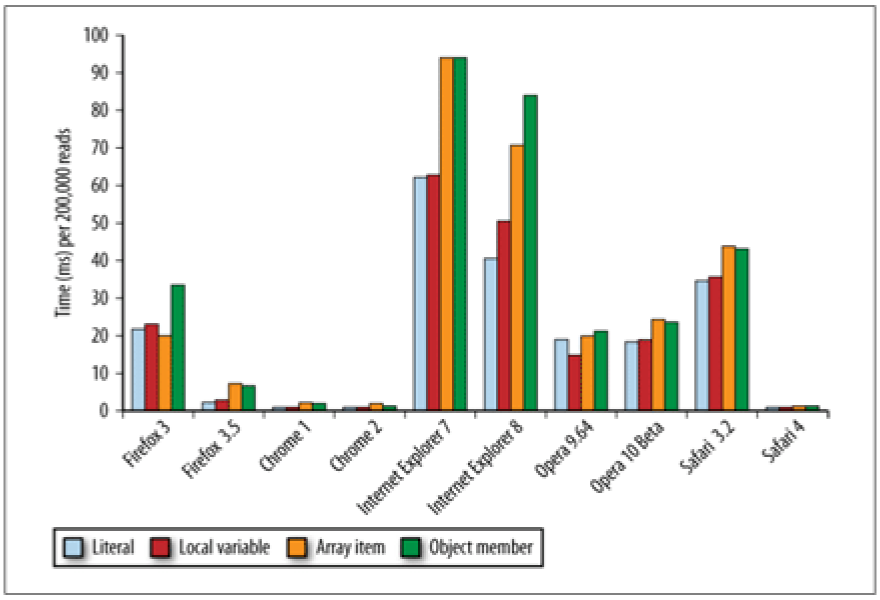


Figure 2-1. Time per 200,000 reads from various data locations

图 2-1 对不同数据类型进行 200'000 次读操作所用的时间

## Managing Scope 管理作用域

The concept of scope is key to understanding JavaScript not just from a performance perspective, but also from a functional perspective. Scope has many effects in JavaScript, from determining what variables a function can access to assigning the value of this. There are also performance considerations when dealing with JavaScript scopes, but to understand how speed relates to scope, it's necessary to understand exactly how scope works.

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

Every function in JavaScript is represented as an object—more specifically, as an instance of Function. Function objects have properties just like any other object, and these include both the properties that you can access programmatically and a series of internal properties that are used by the JavaScript engine but are not accessible through code. One of these properties is [[Scope]], as defined by ECMA-262, Third Edition.

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。

The internal [[Scope]] property contains a collection of objects representing the scope in which the function was created. This collection is called the function's scope chain and it determines the data that a function can access. Each object in the function's scope chain is called a variable object, and each of these contains entries for variables in the form of key-value pairs. When a function is created, its scope chain is populated with objects representing the data that is accessible in the scope in which the function was created. For example, consider the following global function:

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;
}
```

When the add() function is created, its scope chain is populated with a single variable object: the global object representing all of the variables that are globally defined. This global object contains entries for window, navigator, and document, to name a few. Figure 2-2 shows this relationship (note the global object in this figure shows only a few of the global variables as an example; there are many others).

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 2-2 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）。

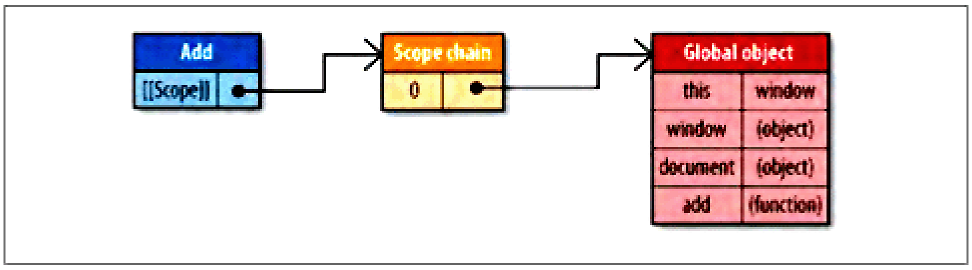


Figure 2-2. Scope chain for the add() function

图 2-2 add()函数的作用域链

The add function's scope chain is later used when the function is executed. Suppose that the following code is executed:

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);
```

Executing the add function triggers the creation of an internal object called an execution context. An execution context defines the environment in which a function is being executed. Each execution context is unique to one particular execution of the function, and so multiple calls to the same function result in multiple execution contexts being created. The execution context is destroyed once the function has been completely executed.

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。

An execution context has its own scope chain that is used for identifier resolution. When the execution context is created, its scope chain is initialized with the objects contained in the executing function's [[Scope]] property. These values are copied over into the execution context scope chain in the order in which they appear in the function. Once this is complete, a new object called the activation object is created for the execution context. The activation object acts as the variable object for this execution and contains entries for all local variables, named arguments, the arguments collection, and this. This object is then pushed to the front of the scope chain. When the execution context is destroyed, so is the activation object. Figure 2-3 shows the execution context and its scope chain for the previous example code.

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 2-3 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。

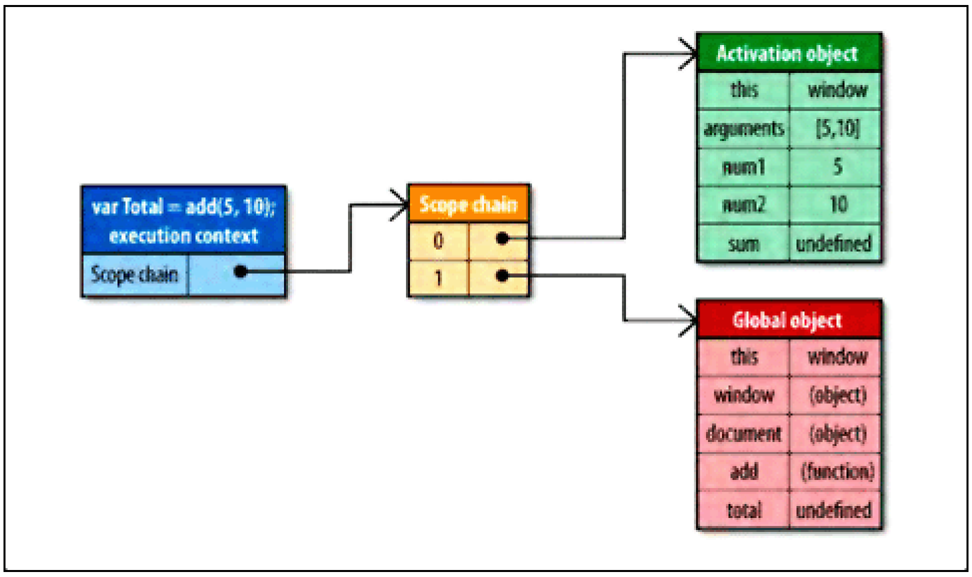


Figure 2-3. Scope chain while executing add()

图 2-3 运行 add()时的作用域链

Each time a variable is encountered during the function's execution, the process of identifier resolution takes place to determine where to retrieve or store the data. During this process, the execution context's scope chain is searched for an identifier with the same name. The search begins at the front of the scope chain, in the execution function's activation object. If found, the variable with the specified identifier is used; if not, the search continues on to the next object in the scope chain. This process continues until either the identifier is found or there are no more variable objects to search, in which case the identifier is deemed to be undefined. The same approach is taken for each identifier found during the function execution, so in the previous example, this would happen for sum, num1, and num2. It is this search process that affects performance.

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**

Identifier resolution isn't free, as in fact no computer operation really is without some sort of performance overhead. The deeper into the execution context's scope chain an identifier exists, the slower it is to access for both reads and writes. Consequently, local variables are always the fastest to access inside of a function, whereas global variables will generally be the slowest (optimizing JavaScript engines are capable of tuning this in certain situations). Keep in mind that global variables always exist in the last variable object of the execution context's scope chain, so they are always the furthest away to resolve. Figures 2-4 and 2-5 show the speed of identifier resolution based on their depth in the scope chain. A depth of 1 indicates a local variable.

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 2-4 和 2-5 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。

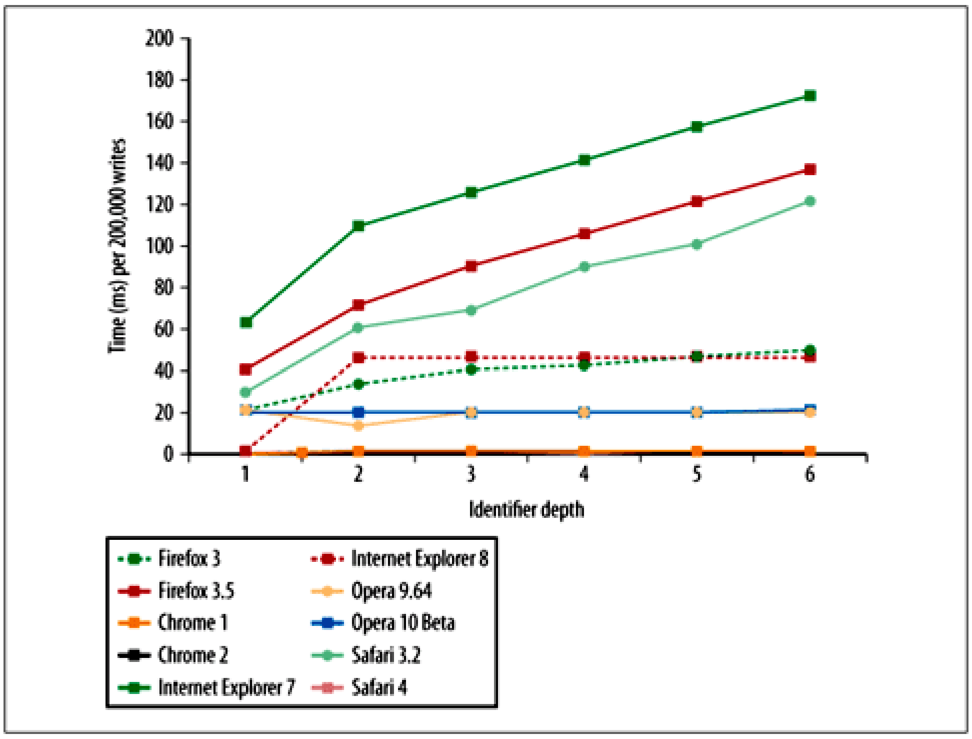


Figure 2-4. Identifier resolution for write operations

图 2-4 写操作的标识符识别速度

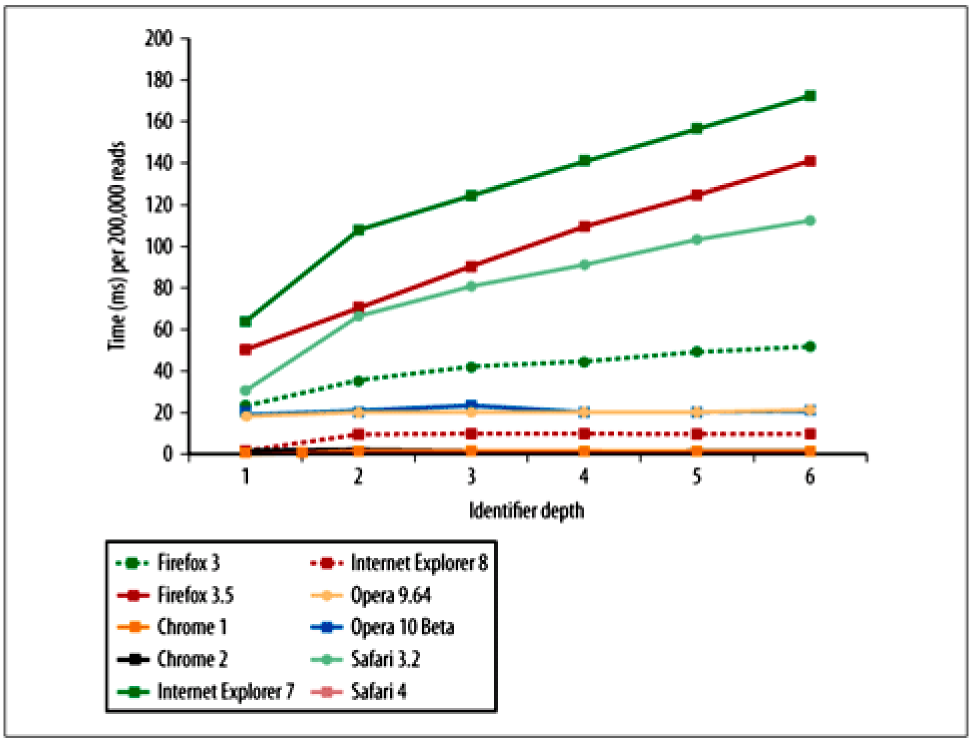


Figure 2-5. Identifier resolution for read operations

图 2-5 读操作的标识符识别速度

The general trend across all browsers is that the deeper into the scope chain an identifier exists, the slower it will be read from or written to. Browsers with optimizing JavaScript engines, such as Chrome and Safari 4, don't have this sort of performance penalty for accessing out-of-scope identifiers, whereas Internet Explorer, Safari 3.2, and others show a more drastic effect. It's worth noting that earlier browsers, such as Internet Explorer 6 and Firefox 2, had incredibly steep slopes and would not even appear within the bounds of this graph at the high point if their data had been included.

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。

Given this information, it's advisable to use local variables whenever possible to improve performance in browsers without optimizing JavaScript engines. A good rule of thumb is to always store out-of-scope values in local variables if they are used more than once within a function. Consider the following example:

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";
}
```

This function contains three references to document, which is a global object. The search for this variable must go all the way through the scope chain before finally being resolved in the global variable object. You can mitigate the performance impact of repeated global variable access by first storing the reference in a local variable and then using the local variable instead of the global. For example, the previous code can be rewritten as follows:

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";
}
```

The updated version of initUI() first stores a reference to document in the local doc variable. Instead of accessing a global variables three times, that number is cut down to one. Accessing doc instead of document is faster because it's a local variable. Of course, this simplistic function won't show a huge performance improvement, because it's not doing that much, but imagine larger functions with dozens of global variables being accessed repeatedly; that is where the more impressive performance improvements will be found.

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**

Generally speaking, an execution context's scope chain doesn't change. There are, however, two statements that temporarily augment the execution context's scope chain while it is being executed. The first of these is with.

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。

The with statement is used to create variables for all of an object's properties. This mimics other languages with similar features and is usually seen as a convenience to avoid writing the same code repeatedly. The initUI() function can be written as the following:

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";
  }
}
```

This rewritten version of initUI() uses a with statement to avoid writing document elsewhere. Though this may seem more efficient, it actually creates a performance problem.

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。

When code execution flows into a with statement, the execution context's scope chain is temporarily augmented. A new variable object is created containing all of the properties of the specified object. That object is then pushed to the front of the scope chain, meaning that all of the function's local variables are now in the second scope chain object and are therefore more expensive to access (see Figure 2-6).

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（参见图 2-6）。

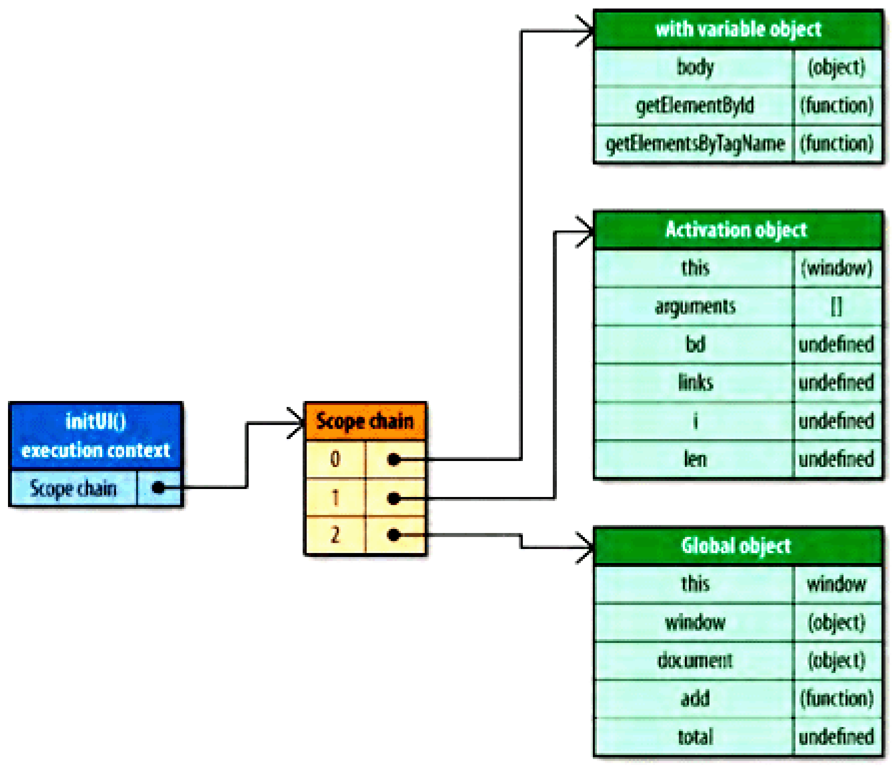


Figure 2-6. Augmented scope chain in a with statement

图 2-6 with 表达式改变作用域链

By passing the document object into the with statement, a new variable object containing all of the document object's properties is pushed to the front of the scope chain. This makes it very fast to access document properties but slower to access the local variables such as bd. For this reason, it's best to avoid using the with statement. As shown previously, it's just as easy to store document in a local variable and get the performance improvement that way.

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。

The with statement isn't the only part of JavaScript that artificially augments the execution context's scope chain; the catch clause of the try-catch statement has the same effect. When an error occurs in the try block, execution automatically flows to the catch and the exception object is pushed into a variable object that is then placed at the front of the scope chain. Inside of the catch block, all variables local to the function are now in the second scope chain object. For example:

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
}
```

Note that as soon as the catch clause is finished executing, the scope chain returns to its previous state.

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。

The try-catch statement is very useful when applied appropriately, and so it doesn't make sense to suggest complete avoidance. If you do plan on using a try-catch, make sure that you understand the likelihood of error. A try-catch should never be used as the solution to a JavaScript error. If you know an error will occur frequently, then that indicates a problem with the code itself that should be fixed.

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。

You can minimize the performance impact of the catch clause by executing as little code as necessary within it. A good pattern is to have a method for handling errors that the catch clause can delegate to, as in this example:

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
}
```

Here a handleError() method is the only code that is executed in the catch clause. This method is free to handle the error in an appropriate way and is passed the exception object generated from the error. Since there is just one statement executed and no local variables accessed, the temporary scope chain augmentation does not affect the performance of the code.

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**

Both the with statement and the catch clause of a try-catch statement, as well as a function containing (), are all considered to be dynamic scopes. A dynamic scope is one that exists only through execution of code and therefore cannot be determined simply by static analysis (looking at the code structure). For example:

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?
};
```

The execute() function represents a dynamic scope due to the use of (). The value of w can change based on the value of code. In most cases, w will be equal to the global window object, but consider the following:

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 = {};")
```

In this case, () creates a local window variable in execute(), so w ends up equal to the local window instead of the global. There is no way to know if this is the case until the code is executed, which means the value of the window identifier cannot be predetermined.

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。

Optimizing JavaScript engines such as Safari's Nitro try to speed up identifier resolution by analyzing the code to determine which variables should be accessible at any given time. These engines try to avoid the traditional scope chain lookup by indexing identifiers for faster resolution. When a dynamic scope is involved, however, this optimization is no longer valid. The engines need to switch back to a slower hash-based approach for identifier resolution that more closely mirrors traditional scope chain lookup.

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。

For this reason, it's recommended to use dynamic scopes only when absolutely necessary.

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**

Closures are one of the most powerful aspects of JavaScript, allowing a function to access data that is outside of its local scope. The use of closures has been popularized through the writings of Douglas Crockford and is now ubiquitous in most complex web applications. There is, however, a performance impact associated with using closures.

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 Douglas Crockford 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。

To understand the performance issues with closures, consider the following:

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);
  };
}
```

The assignEvents() function assigns an event handler to a single DOM element. This event handler is a closure, as it is created when the assignEvents() is executed and can access the id variable from the containing scope. In order for this closure to access id, a specific scope chain must be created.

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。

When assignEvents() is executed, an activation object is created that contains, among other things, the id variable. This becomes the first object in the execution context's scope chain, with the global object coming second. When the closure is created, its [[Scope]] property is initialized with both of these objects (see Figure 2-7).

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（见图 2-7）。

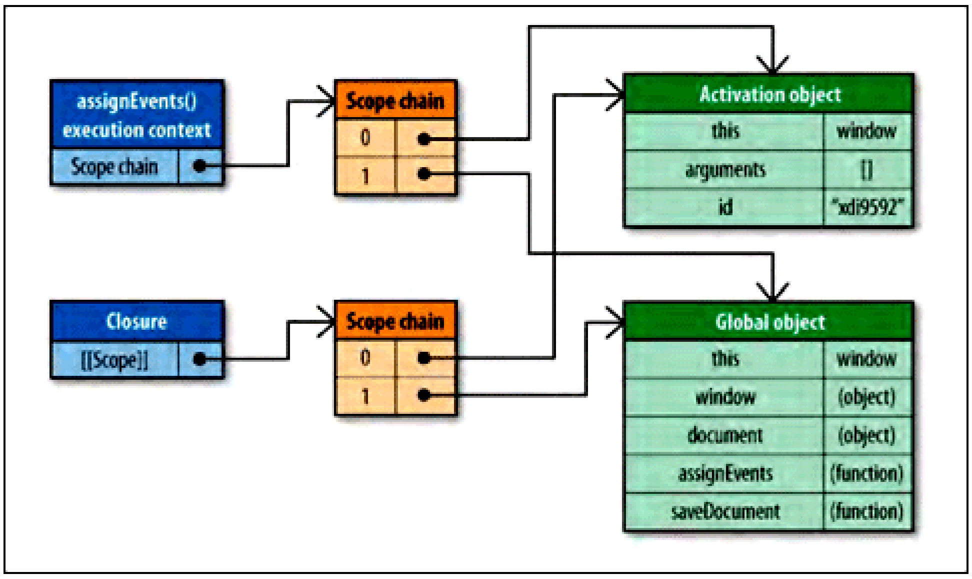


Figure 2-7. Scope chains of the assignEvents() execution context and closure

图 2-7 assignEvents()运行期上下文的作用域链和闭包

Since the closure's [[Scope]] property contains references to the same objects as the execution context's scope chain, there is a side effect. Typically, a function's activation object is destroyed when the execution context is destroyed. When there's a closure involved, though, the activation object isn't destroyed, because a reference still exists in the closure's [[Scope]] property. This means that closures require more memory overhead in a script than a nonclosure function. In large web applications, this might become a problem, especially where Internet Explorer is concerned. IE implements DOM objects as nonnative JavaScript objects, and as such, closures can cause memory leaks (see Chapter 3 for more information).

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（更多信息参见第 3 章）。

When the closure is executed, an execution context is created whose scope chain is initialized with the same two scope chain objects referenced in [[Scope]], and then a new activation object is created for the closure itself (see Figure 2-8).

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（参见图 2-8）。

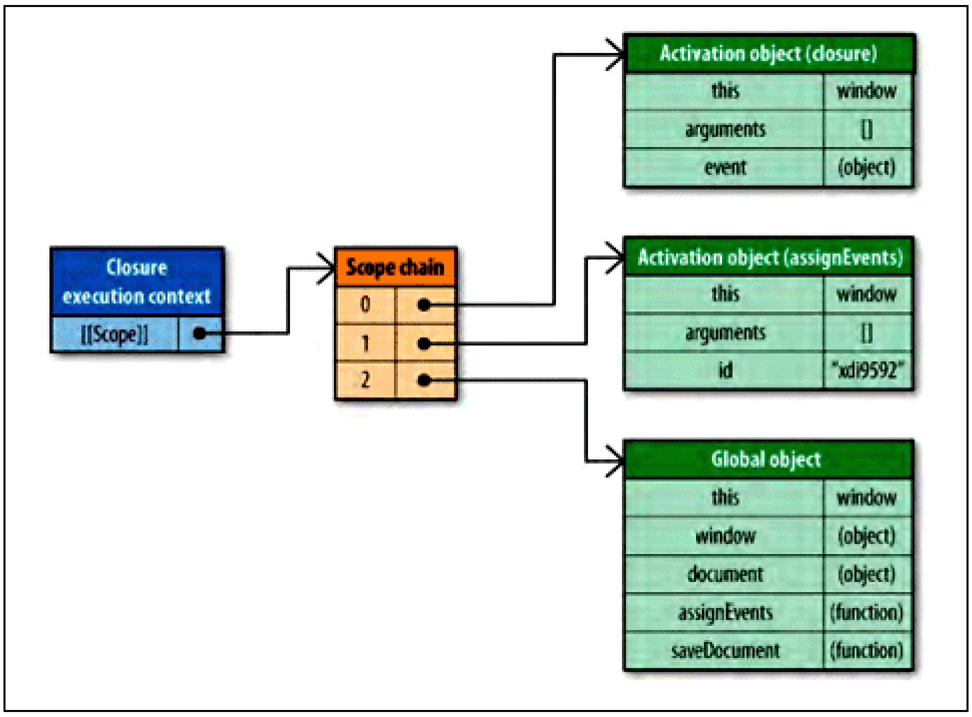


Figure 2-8. Executing the closure

图 2-8 闭包运行

Note that both identifiers used in the closure, id and saveDocument, exist past the first object in the scope chain. This is the primary performance concern with closures: you're often accessing a lot of out-of-scope identifiers and therefore are incurring a performance penalty with each access.

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。

It's best to exercise caution when using closures in your scripts, as they have both memory and execution speed concerns. However, you can mitigate the execution speed impact by following the advice from earlier in this chapter regarding out-of-scope variables: store any frequently used out-of-scope variables in local variables, and then access the local variables directly.

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，你可以通过本章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

Most JavaScript is written in an object-oriented manner, either through the creation of custom objects or the use of built-in objects such as those in the Document Object Model (DOM) and Browser Object Model (BOM). As such, there tends to be a lot of object member access.

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。

Object members are both properties and methods, and there is little difference between the two in JavaScript. A named member of an object may contain any data type. Since functions are represented as objects, a member may contain a function in addition to the more traditional data types. When a named member references a function, it's considered a method, whereas a member referencing a nonfunction data type is considered a property.

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”。

As discussed earlier in this chapter, object member access tends to be slower than accessing data in literals or variables, and in some browsers slower than accessing array items. To understand why this is the case, it's necessary to understand the nature of objects in JavaScript.

正如本章前面所讨论过的，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**

Objects in JavaScript are based on prototypes. A prototype is an object that serves as the base of another object, defining and implementing members that a new object must have. This is a completely different concept than the traditional object-oriented programming concept of classes, which define the process for creating a new object. Prototype objects are shared amongst all instances of a given object type, and so all instances also share the prototype object's members.

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。

An object is tied to its prototype by an internal property. Firefox, Safari, and Chrome expose this property to developers as __proto__; other browsers do not allow script access to this property. Any time you create a new instance of a built-in type, such as Object or Array, these instances automatically have an instance of Object as their prototype.

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。

Consequently, objects can have two types of members: instance members (also called "own" members) and prototype members. Instance members exist directly on the object instance itself, whereas prototype members are inherited from the object prototype. Consider the following example:

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]"
```

In this code, the book object has two instance members: title and publisher. Note that there is no definition for the method toString() but that the method is called and behaves appropriately without throwing an error. The toString() method is a prototype member that the book object is inheriting. Figure 2-9 shows this relationship.

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 2-9 显示出它们之间的关系。

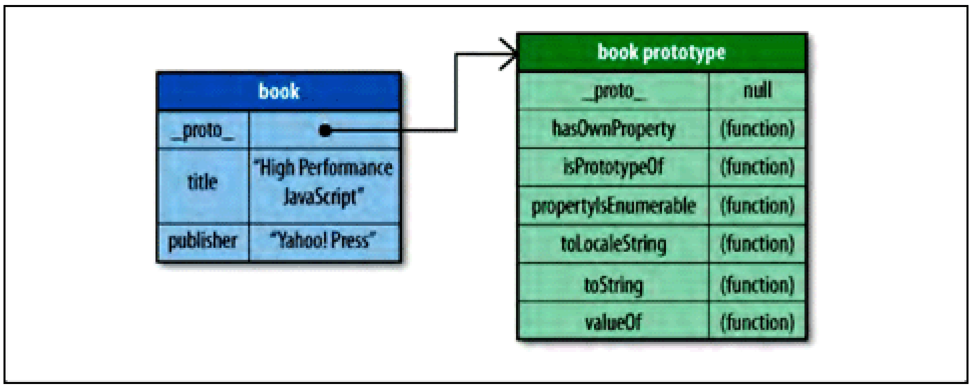


Figure 2-9. Relationship between an instance and prototype

图 2-9 实例与原形的关系

The process of resolving an object member is very similar to resolving a variable. When book.toString() is called, the search for a member named "toString" begins on the object instance. Since book doesn't have a member named toString, the search then flows to the prototype object, where the toString() method is found and executed. In this way, book has access to every property or method on its prototype.

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。

You can determine whether an object has an instance member with a given name by using the hasOwnProperty() method and passing in the name of the member. To determine whether an object has access to a property with a given name, you can use the **in** operator. For example:

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
```

In this code, hasOwnProperty() returns true when "title" is passed in because title is an object instance; the method returns false when "toString" is passed in because it doesn't exist on the instance. When each property name is used with the **in** operator, the result is true both times because it searches the instance and prototype.

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**

The prototype of an object determines the type or types of which it is an instance. By default, all objects are instances of Object and inherit all of the basic methods, such as toString(). You can create a prototype of another type by defining and using a constructor. For example:

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]"
```

The Book constructor is used to create a new instance of Book. The book1 instance's prototype (__proto__) is Book.prototype, and Book.prototype's prototype is Object. This creates a prototype chain from which both book1 and book2 inherit their members. Figure 2-10 shows this relationship.

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。图 2-10 显示出这种关系。

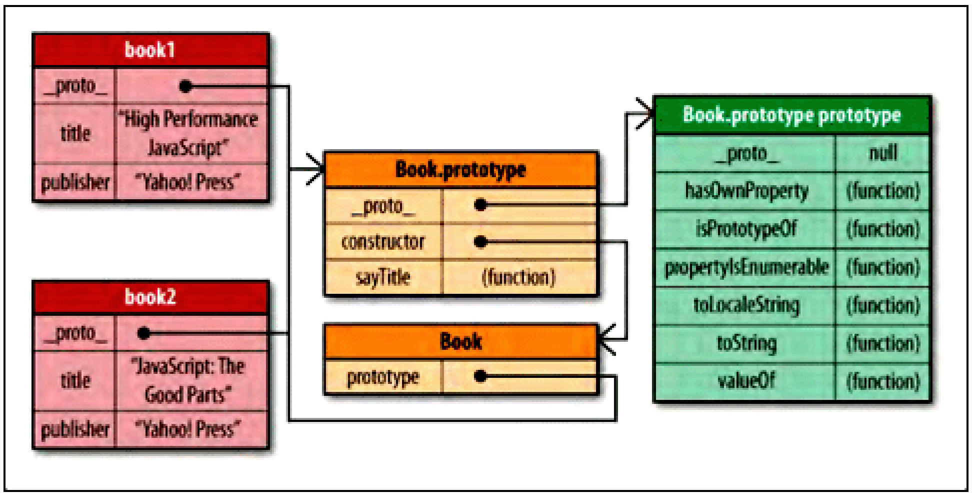


Figure 2-10. Prototype chains

图 2-10　原形链

Note that both instances of Book share the same prototype chain. Each instance has its own title and publisher properties, but everything else is inherited through prototypes. Now when book1.toString() is called, the search must go deeper into the prototype chain to resolve the object member "toString". As you might suspect, the deeper into the prototype chain that a member exists, the slower it is to retrieve. Figure 2-11 shows the relationship between member depth in the prototype and time to access the member.

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 2-11 显示出成员在原形链中所处的深度与访问时间的关系。

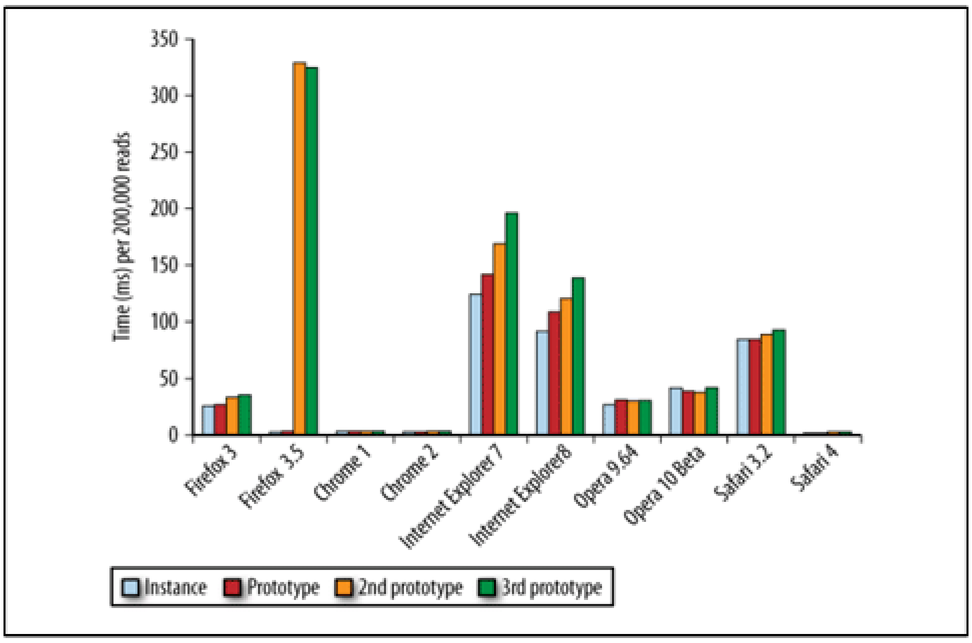


Figure 2-11. Data access going deeper into the prototype chain

图 2-11　数据访问深入原形链

Although newer browsers with optimizing JavaScript engines perform this task well, older browsers—especially Internet Explorer and Firefox 3.5—incur a performance penalty with each additional step into the prototype chain. Keep in mind that the process of looking up an instance member is still more expensive than accessing data from a literal or a local variable, so adding more overhead to traverse the prototype chain just amplifies this effect.

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 Internet Explorer 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**

Since object members may contain other members, it's not uncommon to see patterns such as window.location.href in JavaScript code. These nested members cause the JavaScript engine to go through the object member resolution process each time a dot is encountered. Figure 2-12 shows the relationship between object member depth and time to access.

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 2-12 显示出对象成员深度与访问时间的关系。

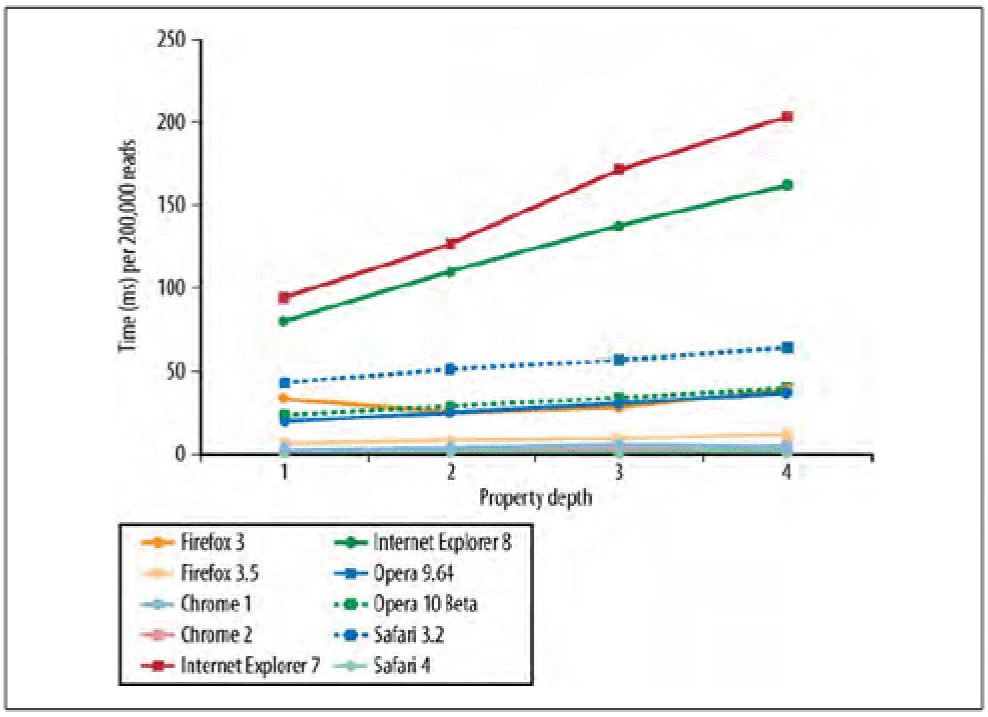


Figure 2-12. Access time related to property depth

图 2-12　访问时间与属性深度的关系

It should come as no surprise, then, that the deeper the nested member, the slower the data is accessed. Evaluating location.href is always faster than window.location.href, which is faster than window.location.href.toString(). If these properties aren't on the object instances, then member resolution will take longer as the prototype chain is searched at each point.

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

With all of the performance issues related to object members, it's easy to believe that they should be avoided whenever possible. To be more accurate, you should be careful to use object member only when necessary. For instance, there's no reason to read the value of an object member more than once in a single function:

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;
}
```

In this code, element.className is accessed twice. Clearly this value isn't going to change during the course of the function, yet there are still two object member lookups performed. You can eliminate one property lookup by storing the value in a local variable and using that instead:

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;
}
```

This rewritten version of the function limits the number of member lookups to one. Since both member lookups were reading the property's value, it makes sense to read the value once and store it in a local variable. That local variable then is much faster to access.

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。

Generally speaking, if you're going to read an object property more than one time in a function, it's best to store that property value in a local variable. The local variable can then be used in place of the property to avoid the performance overhead of another property lookup. This is especially important when dealing with nested object members that have a more dramatic effect on execution speed.

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 namespacing, such as the technique used in **YUI**, is a source of frequently accessed nested properties. For example:

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;
  }
}
```

This code repeats **YAHOO.util.Dom** three times to access three different methods. For each method there are three member lookups, for a total of nine, making this code quite inefficient. A better approach is to store **YAHOO.util.Dom** in a local variable and then access that local variable:

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;
  }
}
```

The total number of member lookups in this code has been reduced from nine to five. You should never look up an object member more than once within a single function, unless the value may have changed.

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。

## Summary 总结

Where you store and access data in JavaScript can have a measurable impact on the overall performance of your code. There are four places to access data from: literal values, variables, array items, and object members. These locations all have different performance considerations.

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。

• Literal values and local variables can be accessed very quickly, whereas array items and object members take longer.

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。

• Local variables are faster to access than out-of-scope variables because they exist in the first variable object of the scope chain. The further into the scope chain a variable is, the longer it takes to access. Global variables are always the slowest to access because they are always last in the scope chain.

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。

• Avoid the with statement because it augments the execution context scope chain. Also, be careful with the catch clause of a try-catch statement because it has the same effect.

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。

• Nested object members incur significant performance impact and should be minimized.

嵌套对象成员会造成重大性能影响，尽量少用。

• The deeper into the prototype chain that a property or method exists, the slower it is to access.

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。

• Generally speaking, you can improve the performance of JavaScript code by storing frequently used object members, array items, and out-of-scope variables in local variables. You can then access the local variables faster than the originals.

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。

By using these strategies, you can greatly improve the perceived performance of a web application that requires a large amount of JavaScript code.

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。

# 第三章 **DOM Scripting DOM** 编程

DOM scripting is expensive, and it's a common performance bottleneck in rich web applications. This chapter discusses the areas of DOM scripting that can have a negative effect on an application's responsiveness and gives recommendations on how to improve response time. The three categories of problems discussed in the chapter include:

对 DOM 操作代价昂贵，在富网页应用中通常是一个性能瓶颈。本章讨论可能对程序响应造成负面影响的 DOM 编程，并给出提高响应速度的建议。本章讨论三类问题：

• Accessing and modifying DOM elements

访问和修改 DOM 元素

• Modifying the styles of DOM elements and causing repaints and reflows

修改 DOM 元素的样式，造成重绘和重新排版

• Handling user interaction through DOM events

通过 DOM 事件处理用户响应

But first—what is DOM and why is it slow?

但首先——什么是 DOM？他为什么慢？

## DOM in the Browser World 浏览器世界中的 DOM

The Document Object Model (DOM) is a language-independent application interface (API) for working with XML and HTML documents. In the browser, you mostly work with HTML documents, although it's not uncommon for web applications to retrieve XML documents and use the DOM APIs to access data from those documents.

文档对象模型（DOM）是一个独立于语言的，使用 XML 和 HTML 文档操作的应用程序接口（API）。在浏览器中，主要与 HTML 文档打交道，在网页应用中检索 XML 文档也很常见。DOM APIs 主要用于访问这些文档中的数据。

Even though the DOM is a language-independent API, in the browser the interface is implemented in JavaScript. Since most of the work in client-side scripting has to do with the underlying document, DOM is an important part of everyday JavaScript coding.

尽管 DOM 是与语言无关的 API，在浏览器中的接口却是以 JavaScript 实现的。客户端大多数脚本程序与文档打交道，DOM 就成为 JavaScript 代码日常行为中重要的组成部分。

It's common across browsers to keep DOM and JavaScript implementations independent of each other. In Internet Explorer, for example, the JavaScript implementation is called JScript and lives in a library file called jscript.dll, while the DOM implementation lives in another library, mshtml.dll (internally called Trident). This separation allows other technologies and languages, such as VBScript, to benefit from the DOM and the rendering functionality Trident has to offer. Safari uses WebKit's WebCore for DOM and rendering and has a separate JavaScriptCore engine (dubbed SquirrelFish in its latest version). Google Chrome also uses WebCore libraries from WebKit for rendering pages but implements its own JavaScript engine called V8. In Firefox, Spider-Monkey (the latest version is called TraceMonkey) is the JavaScript implementation, a separate part of the Gecko rendering engine.

浏览器通常要求 DOM 实现和 JavaScript 实现保持相互独立。例如，在 Internet Explorer 中，被称为 JScript 的 JavaScript 实现位于库文件 jscript.dll 中，而 DOM 实现位于另一个库 mshtml.dll（内部代号 Trident）。这种分离技术允许其他技术和语言，如 VBScript，受益于 Trident 所提供的 DOM 功能和渲染功能。Safari 使用 WebKit 的 WebCore 处理 DOM 和渲染，具有一个分离的 JavaScriptCore 引擎（最新版本中的绰号是 SquirrelFish）。Google Chrome 也使用 WebKit 的 WebCore 库渲染页面，但实现了自己的 JavaScript 引擎 V8。在 Firefox 中，JavaScript 实现采用 Spider-Monkey（最新版中称作 TraceMonkey），与其 Gecko 渲染引擎相分离。

**Inherently Slow** 天生就慢

What does that mean for performance? Simply having two separate pieces of functionality interfacing with each other will always come at a cost. An excellent analogy is to think of DOM as a piece of land and JavaScript (meaning ECMAScript) as another piece of land, both connected with a toll bridge (see John Hrvatin, Microsoft, MIX09, http://videos.visitmix.com/MIX09/T53F). Every time your ECMAScript needs access to the DOM, you have to cross this bridge and pay the performance toll fee. The more you work with the DOM, the more you pay. So the general recommendation is to cross that bridge as few times as possible and strive to stay in ECMAScript land. The rest of the chapter focuses on what this means exactly and where to look in order to make user interactions faster.

这对性能意味着什么呢？简单说来，两个独立的部分以功能接口连接就会带来性能损耗。一个很形象的比喻是把 DOM 看成一个岛屿，把 JavaScript（ECMAScript）看成另一个岛屿，两者之间以一座收费桥连接（参见 John Hrvatin，微软，MIX09，http://videos.visitmix.com/MIX09/T53F）。每次 ECMAScript 需要访问 DOM 时，你需要过桥，交一次"过桥费"。你操作 DOM 次数越多，费用就越高。一般的建议是尽量减少过桥次数，努力停留在 ECMAScript 岛上。本章将对此问题给出详细解答，告诉你应该关注什么地方，以提高用户交互速度。

## DOM Access and Modification DOM 访问和修改

Simply accessing a DOM element comes at a price—the "toll fee" discussed earlier. Modifying elements is even more expensive because it often causes the browser to recalculate changes in the page geometry.

简单来说，正如前面所讨论的那样，访问一个 DOM 元素的代价就是交一次"过桥费"。修改元素的费用可能更贵，因为它经常导致浏览器重新计算页面的几何变化。

Naturally, the worst case of accessing or modifying elements is when you do it in loops, and especially in loops over HTML collections.

当然，访问或修改元素最坏的情况是使用循环执行此操作，特别是在 HTML 集合中使用循环。

Just to give you an idea of the scale of the problems with DOM scripting, consider this simple example:

为了给你一个关于 DOM 操作问题的量化印象，考虑下面的例子：

```
function innerHTMLLoop() {
  for (var count = 0; count < 15000; count++) {
    document.getElementById('here').innerHTML += 'a';
  }
}
```

This is a function that updates the contents of a page element in a loop. The problem with this code is that for every loop iteration, the element is accessed twice: once to read the value of the innerHTML property and once to write it.

此函数在循环中更新页面内容。这段代码的问题是，在每次循环单元中都对 DOM 元素访问两次：一次读取 innerHTML 属性能容，另一次写入它。

A more efficient version of this function would use a local variable to store the updated contents and then write the value only once at the end of the loop:

一个更有效率的版本将使用局部变量存储更新后的内容，在循环结束时一次性写入：

```
function innerHTMLLoop2() {
  var content = '';
  for (var count = 0; count < 15000; count++) {
    content += 'a';
  }
  document.getElementById('here').innerHTML += content;
}
```

This new version of the function will run much faster across all browsers. Figure 3-1 shows the results of measuring the time improvement in different browsers. The y-axis in the figure (as with all the figures in this chapter) shows execution time improvement, i.e., how much faster it is to use one approach versus another. In this case, for example, using innerHTMLLoop2() is 155 times faster than innerHTMLLoop() in IE6.

在所有浏览器中，新版本运行速度都要快得多。图 3-1 显示了在不同浏览器上测量到的速度提升。Y 轴的数字显示出速度提升，比方说，一个比另一个快了多少倍。例如在 IE6 中，innerHTMLLoop2()比 innerHTMLLoop()快了 155 倍。

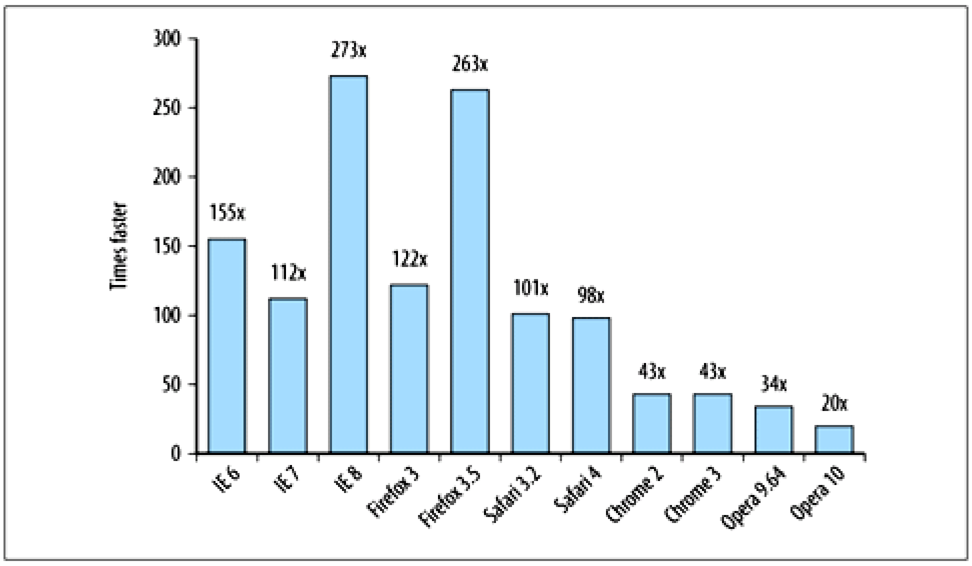


Figure 3-1. One benefit of staying within ECMAScript: innerHTMLLoop2() is hundreds of times faster than innerHTMLLoop()

图 3-1 innerHTMLLoop2()比 innerHTMLLoop()快上百倍

As these results clearly show, the more you access the DOM, the slower your code executes. Therefore, the general rule of thumb is this: touch the DOM lightly, and stay within ECMAScript as much as possible.

这些结果清楚地表明，你访问 DOM 越多，代码的执行速度就越慢。因此，一般经验法则是：轻轻地触摸 DOM，并尽量保持在 ECMAScript 范围内。

**innerHTML Versus DOM methods innerHTML 与 DOM 方法比较**

Over the years, there have been many discussions in the web development community over this question: is it better to use the nonstandard but well-supported innerHTML property to update a section of a page, or is it best to use only the pure DOM methods, such as document.createElement ()? Leaving the web standards discussion aside, does it matter for performance? The answer is: it matters increasingly less, but still, innerHTML is faster in all browsers except the latest WebKit-based ones (Chrome and Safari).

多年来，在 web 开发者社区已经对此问题进行了许多讨论：更新页面时，使用虽不标准却被良好支持的 innerHTML 属性更好呢，还是使用纯 DOM 方法，如 document.createElement ()更好呢？如果不考虑标准

问题，它们的性能如何？答案是：性能差别不大，但是，在所有浏览器中，innerHTML 速度更快一些，除了最新的基于 WebKit 的浏览器（Chrome 和 Safari）。

Let's examine a sample task of creating a table of 1000 rows in two ways:

让我们检验一个例子，用两种方法来创建一个 1000 行的表：

• By concatenating an HTML string and updating the DOM with innerHTML

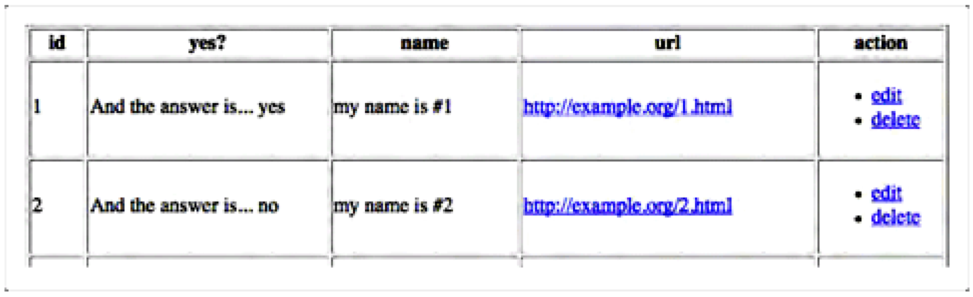
通过构造一个 HTML 字符串，然后更新 DOM 的 innerHTML 属性

• By using only standard DOM methods such as document.createElement() and document.createTextNode()

通过标准 DOM 方法 document.createElement ()和 document.createTextNode()

Our example table has content similar to content that would have come from a Content Management System (CMS). The end result is shown in Figure 3-2.

我们例子中的表内容从一个内容管理系统（CMS）中获得，其显示结果如图 3-2。



| id | yes? | name | url | action |
|---|---|---|---|---|
| 1 | And the answer is... yes | my name is #1 | http://example.org/1.html | • edit<br>• delete |
| 2 | And the answer is... no | my name is #2 | http://example.org/2.html | • edit<br>• delete |

Figure 3-2. End result of generating an HTML table with 1,000 rows and 5 columns

图 3-2　创建一个 1000 行 5 列的 HTML 表

The code to generate the table with innerHTML is as follows:

使用 innerHTML 创建表的代码如下：

```
function tableInnerHTML() {
  var i, h = ['<table border="1" width="100%">'];
  h.push('<thead>');
  h.push('<tr><th>id<\/th><th>yes?<\/th><th>name<\/th><th>url<\/th><th>action<\/th><\/tr>');
  h.push('<\/thead>');
  h.push('<tbody>');
  for (i = 1; i <= 1000; i++) {
    h.push('<tr><td>');
    h.push(i);
    h.push('<\/td><td>');
    h.push('And the answer is... ' + (i % 2 ? 'yes' : 'no'));
    h.push('<\/td><td>');
    h.push('my name is #' + i);
    h.push('<\/td><td>');
    h.push('<a href="http://example.org/' + i + '.html">http://example.org/' + i + '.html<\/a>');
    h.push('<\/td><td>');
    h.push('<ul>');
    h.push(' <li><a href="edit.php?id=' + i + '">edit<\/a><\/li>');
    h.push(' <li><a href="delete.php?id="' + i + '-id001">delete<\/a><\/li>');
    h.push('<\/ul>');
    h.push('<\/td>');
    h.push('<\/tr>');
  }
  h.push('<\/tbody>');
  h.push('<\/table>');
  document.getElementById('here').innerHTML = h.join('');
};
```

In order to generate the same table with DOM methods alone, the code is a little more verbose:

如果使用 DOM 方法创建同样的表，代码有些冗长。

```
function tableDOM() {
  var i, table, thead, tbody, tr, th, td, a, ul, li;
  tbody = document.createElement ('tbody');
  for (i = 1; i <= 1000; i++) {
    tr = document.createElement ('tr');
    td = document.createElement ('td');
    td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
    tr.appendChild(td);
    td = document.createElement ('td');
    td.appendChild(document.createTextNode(i));
    tr.appendChild(td);
    td = document.createElement ('td');
    td.appendChild(document.createTextNode('my name is #' + i));
    tr.appendChild(td);
    a = document.createElement ('a');
    a.setAttribute('href', 'http://example.org/' + i + '.html');
    a.appendChild(document.createTextNode('http://example.org/' + i + '.html'));
    td = document.createElement ('td');
    td.appendChild(a);
    tr.appendChild(td);
    ul = document.createElement ('ul');
    a = document.createElement ('a');
    a.setAttribute('href', 'edit.php?id=' + i);
    a.appendChild(document.createTextNode('edit'));
    li = document.createElement ('li');
    li.appendChild(a);
    ul.appendChild(li);
    a = document.createElement ('a');
    a.setAttribute('href', 'delete.php?id=' + i);
    a.appendChild(document.createTextNode('delete'));
    li = document.createElement ('li');
    li.appendChild(a);
    ul.appendChild(li);
    td = document.createElement ('td');
    td.appendChild(ul);
    tr.appendChild(td);
    tbody.appendChild(tr);
  }
  tr = document.createElement ('tr');
  th = document.createElement ('th');
  th.appendChild(document.createTextNode('yes?'));
  tr.appendChild(th);
  th = document.createElement ('th');
  th.appendChild(document.createTextNode('id'));
  tr.appendChild(th);
  th = document.createElement ('th');
  th.appendChild(document.createTextNode('name'));
  tr.appendChild(th);
  th = document.createElement('th');
  th.appendChild(document.createTextNode('url'));
  tr.appendChild(th);
  th = document.createElement('th');
  th.appendChild(document.createTextNode('action'));
  tr.appendChild(th);
  thead = document.createElement('thead');
  thead.appendChild(tr);
  table = document.createElement('table');
  table.setAttribute('border', 1);
```

```
  table.setAttribute('width', '100%');
  table.appendChild(thead);
  table.appendChild(tbody);
  document.getElementById('here').appendChild(table);
};
```

The results of generating the HTML table using innerHTML as compared to using pure DOM methods are shown in Figure 3-3. The benefits of innerHTML are more obvious in older browser versions (innerHTML is 3.6 times faster in IE6), but the benefits are less pronounced in newer versions. And in newer WebKit-based browsers it's the opposite: using DOM methods is slightly faster. So the decision about which approach to take will depend on the browsers your users are commonly using, as well as your coding preferences.

使用 innerHTML 和纯 DOM 方法创建 HTML 表的比较结果参见图 3-3。innerHTML 的好处在老式浏览器上显而易见（在 IE6 中 innerHTML 比对手快 3.6 倍），但在新版本浏览器上就不那么明显了。而在最新的基于 WebKit 的浏览器上其结果正好相反：使用 DOM 方法更快。因此，决定采用哪种方法将取决于用户经常使用的浏览器，以及你的编码偏好。

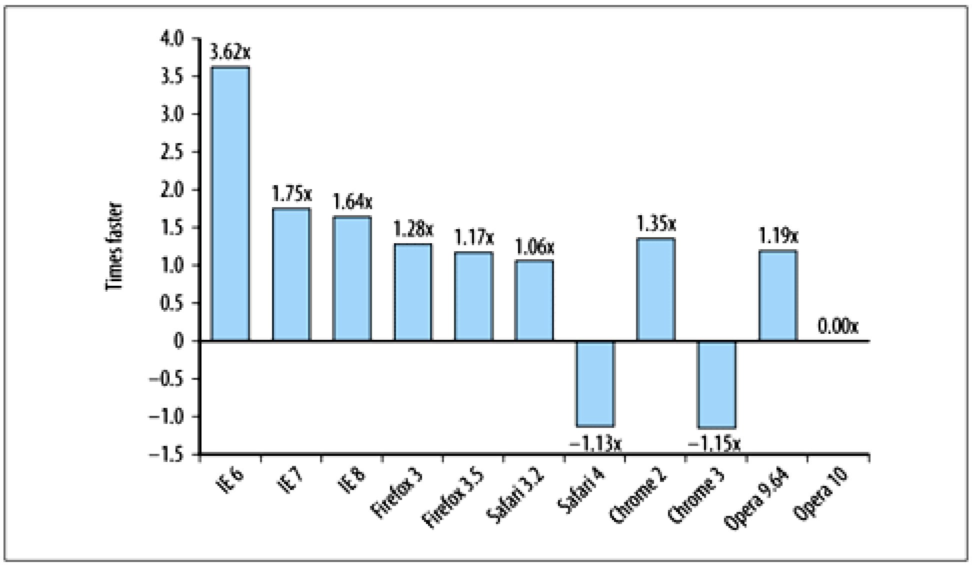


Figure 3-3. The benefit of using innerHTML over DOM methods to create a 1,000-row table; innerHTML is more than three times faster in IE6 and slightly slower in the latest WebKit browsers

图 3-3 使用 innerHTML 和 DOM 方法创建一个 1000 行的表

在 IE6 中，innerHTML 比对手快三倍，但在最新的基于 WebKit 的浏览器中慢于对手

Using innerHTML will give you faster execution in most browsers in performance-critical operations that require updating a large part of the HTML page. But for most everyday cases there isn't a big difference, and so you should consider readability, maintenance, team preferences, and coding conventions when deciding on your approach.

如果在一个性能苛刻的操作中更新一大块 HTML 页面，innerHTML 在大多数浏览器中执行更快。但对于大多数日常操作而言，其差异并不大，所以你应当根据代码可读性，可维护性，团队习惯，代码风格来综合决定采用哪种方法。

## Cloning Nodes 节点克隆

Another way of updating page contents using DOM methods is to clone existing DOM elements instead of creating new ones—in other words, using element.cloneNode() (where element is an existing node) instead of document.createElement().

使用 DOM 方法更新页面内容的另一个途径是克隆已有 DOM 元素，而不是创建新的——即使用 element.cloneNode()（element 是一个已存在的节点）代替 document.createElement();

Cloning nodes is more efficient in most browsers, but not by a big margin. Regenerating the table from the previous example by creating the repeating elements only once and then copying them results in slightly faster execution times:

在大多数浏览器上，克隆节点更有效率，但提高不太多。用克隆节点的办法重新生成前面例子中的表，单元只创建一次，然后重复执行复制操作，这样做只是稍微快了一点：

• 2% in IE8, but no change in IE6 and IE7

在 IE8 中快 2%，但在 IE6 和 IE7 中无变化

• Up to 5.5% in Firefox 3.5 and Safari 4

在 Firefox 3.5 和 Safari 4 中快了 5.5%

• 6% in Opera (but no savings in Opera 10)

在 Opera 中快了 6%（但是在 Opera 10 中无变化）

• 10% in Chrome 2 and 3% in Chrome 3

在 Chrome 2 中快了 10%，在 Chrome 3 中快了 3%

As an illustration, here's a partial code listing for generating the table using element.cloneNode():

一个示例，这里是使用 element.cloneNode()创建表的部分代码：

```
function tableClonedDOM() {
  var i, table, thead, tbody, tr, th, td, a, ul, li,
  oth = document.createElement('th'),
  otd = document.createElement('td'),
  otr = document.createElement('tr'),
  oa = document.createElement('a'),
  oli = document.createElement('li'),
  oul = document.createElement('ul');
  tbody = document.createElement('tbody');
  for (i = 1; i <= 1000; i++) {
    tr = otr.cloneNode(false);
    td = otd.cloneNode(false);
    td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
    tr.appendChild(td);
    td = otd.cloneNode(false);
    td.appendChild(document.createTextNode(i));
    tr.appendChild(td);
    td = otd.cloneNode(false);
```

```
      td.appendChild(document.createTextNode('my name is #' + i));
      tr.appendChild(td);
      // ... the rest of the loop ...
    }
    // ... the rest of the table generation ...
}
```

## HTML Collections HTML 集合

HTML collections are array-like objects containing DOM node references. Examples of collections are the values returned by the following methods:

HTML 集合是用于存放 DOM 节点引用的类数组对象。下列函数的返回值就是一个集合：

- document.getElementsByName()
- document.getElementsByClassName()
- document.getElementsByTagName_r()

The following properties also return HTML collections:

下列属性也属于 HTML 集合：

**document.images**

All **img** elements on the page

页面中所有的<img>元素

**document.links**

All **a** elements

所有的<a>元素

**document.forms**

All forms

所有表单

**document.forms[0].elements**

All fields in the first form on the page

页面中第一个表单的所有字段

These methods and properties return **HTMLCollection** objects, which are array-like lists. They are not arrays (because they don't have methods such as **push**() or **slice**()), but provide a **length** property just like arrays and allow indexed access to the elements in the list. For example, **document.images[1]** returns the second element in the collection. As defined in the DOM standard, HTML collections are "assumed to be live, meaning that they are automatically updated when the underlying document is updated" (seehttp://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-75708506).

这些方法和属性返回 HTMLCollection 对象，是一种类似数组的列表。它们不是数组（因为它们没有诸如 push()或 slice()之类的方法），但是提供了一个 length 属性，和数组一样你可以使用索引访问列表中的元素。例如，document.images[1]返回集合中的第二个元素。正如 DOM 标准中所定义的那样，HTML 集合是一个"虚拟存在，意味着当底层文档更新时，它们将自动更新"（参见 http://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-75708506）。

The HTML collections are in fact queries against the document, and these queries are being reexecuted every time you need up-to-date information, such as the number of elements in the collection (i.e., the collection's length). This could be a source of inefficiencies.

HTML 集合实际上在查询文档，当你更新信息时，每次都要重复执行这种查询操作。例如读取集合中元素的数目（也就是集合的 length）。这正是低效率的来源。

**Expensive collections** 昂贵的集合

To demonstrate that the collections are live, consider the following snippet:

为演示集合的存在性，考虑下列代码段：

```
// an accidentally infinite loop
var alldivs = document.getElementsByTagName_r('div');
for (var i = 0; i < alldivs.length; i++) {
  document.body.appendChild(document.createElement('div'))
}
```

This code looks like it simply doubles the number of **div** elements on the page. It loops through the existing **div**s and creates a new **div** every time, appending it to the **body**. But this is in fact an infinite loop because the loop's exit condition, **alldivs.length**, increases by one with every iteration, reflecting the current state of the underlying document.

这段代码看上去只是简单地倍增了页面中 div 元素的数量。它遍历现有 div，每次创建一个新的 div 并附加到 body 上面。但实际上这是个死循环，因为循环终止条件 alldivs.length 在每次迭代中都会增加，它反映出底层文档的当前状态。

Looping through HTML collections like this may lead to logic mistakes, but it's also slower, due to the fact that the query needs to run on every iteration (see Figure 3-4).

像这样遍历 HTML 集合会导致逻辑错误，而且也很慢，因为每次迭代都进行查询（如图 3-4）。

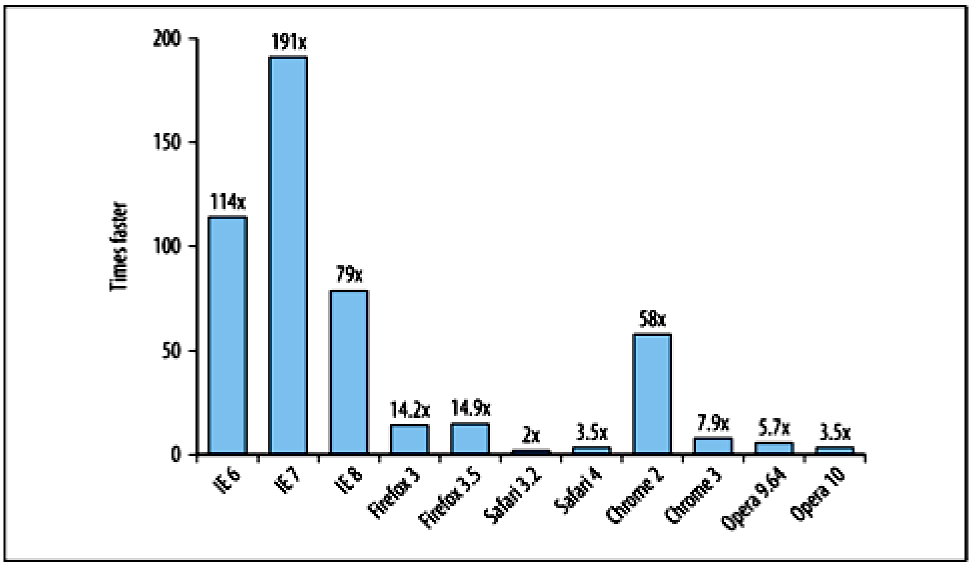


Figure 3-4. Looping over an array is significantly faster than looping through an HTML collection of the same size and content

图 3-4　遍历数组明显快于同样大小和内容的 HTML 集合

As discussed in Chapter 4, accessing an array's **length** property in loop control conditions is not recommended. Accessing a collection's **length** is even slower than accessing a regular array's **length** because it means rerunning the query every time. This is demonstrated by the following example, which takes a collection **coll**, copies it into an array **arr**, and then compares how much time it takes to iterate through each.

正如在第四章中将要讨论的，不建议用数组的 length 属性做循环判断条件。访问集合的 length 比数组的 length 还要慢，因为它意味着每次都要重新运行查询过程。在下面的例子中，将一个集合 coll 拷贝到数组 arr 中，然后比较每次迭代所用的时间。

Consider a function that copies an HTML collection into a regular array:

考虑这个函数，它将一个 HTML 集合拷贝给一个常规数组：

```
function toArray(coll) {
  for (var i = 0, a = [], len = coll.length; i < len; i++) {
    a[i] = coll[i];
  }
  return a;
}
```

And setting up a collection and a copy of it into an array:

设置一个集合，并把它拷贝到一个数组：

```
var coll = document.getElementsByTagName_r('div');
var ar = toArray(coll);
```

The two functions to compare would be:

比较下列两个函数：

```
//slower
function loopCollection() {
  for (var count = 0; count < coll.length; count++) {

  }
}
// faster
function loopCopiedArray() {
  for (var count = 0; count < arr.length; count++) {

  }
}
```

When the **length** of the collection is accessed on every iteration, it causes the collection to be updated and has a significant performance penalty across all browsers. The way to optimize this is to simply cache the **length** of the collection into a variable and use this variable to compare in the loop's exit condition:

当每次迭代过程访问集合的 length 属性时，它导致集合器更新，在所有浏览器上都会产生明显的性能损失。优化的办法很简单，只要将集合的 length 属性缓存到一个变量中，然后在循环判断条件中使用这个变量：

```
function loopCacheLengthCollection() {
  var coll = document.getElementsByTagName_r('div'),
  len = coll.length;
  for (var count = 0; count < len; count++) {

  }
}
```

This function will run about as fast as loopCopiedArray().

此函数运行得与 loopCopiedArray()一样快。

For many use cases that require a single loop over a relatively small collection, just caching the **length** of the collection is good enough. But looping over an array is faster that looping over a collection, so if the elements of the collection are copied into an array first, accessing their properties is faster. Keep in mind that this comes at the price of an extra step and another loop over the collection, so it's important to profile and decide whether using an array copy will be beneficial in your specific case.

许多用例需要对一个相关的小集合进行遍历，只要将 length 缓存一下就足够好了。但是遍历数组比遍历集合快，如果先将集合元素拷贝到数组，访问它们的属性将更快。请记住这需要一个额外的步骤，要遍历集合，所以应当评估在特定条件下使用这样一个数组副本是否有益。

Consult the function toArray() shown earlier for an example of a generic collection-to-array function.

前面提到的 toArray()函数可认为是一个通用的集合转数组函数。

**Local variables when accessing collection elements 访问集合元素时使用局部变量**

The previous example used just an empty loop, but what happens when the elements of the collection are accessed within the loop?

前面的例子使用了一个空循环，如果在循环中访问集合元素，会发生什么？

In general, for any type of DOM access it's best to use a local variable when the same DOM property or method is accessed more than once. When looping over a collection, the first optimization is to store the collection in a local variable and cache the **length** outside the loop, and then use a local variable inside the loop for elements that are accessed more than once.

一般来说，对于任何类型的 DOM 访问，如果同一个 DOM 属性或方法被访问一次以上，最好使用一个局部变量缓存此 DOM 成员。当遍历一个集合时，第一个优化是将集合引用存储于局部变量，并在循环之外缓存 length 属性。然后，如果在循环体中多次访问同一个集合元素，那么使用局部变量缓存它。

In the next example, three properties of each element are accessed within the loop. The slowest version accesses the global document every time, an optimized version caches a reference to the collection, and the fastest version also stores the current element of the collection into a variable. All three versions cache the **length** of the collection.

在下面的例子中，在循环中访问每个元素的三个属性。最慢的版本每次都要访问全局的 document，优化后的版本缓存了一个指向集合的引用，最快的版本将集合的当前元素存入局部变量。所有三个版本都缓存了集合的 length 属性。

```
// slow
function collectionGlobal() {
  var coll = document.getElementsByTagName_r('div'),
  len = coll.length,
  name = '';
  for (var count = 0; count < len; count++) {
    name = document.getElementsByTagName_r('div')[count].nodeName;
    name = document.getElementsByTagName_r('div')[count].nodeType;
    name = document.getElementsByTagName_r('div')[count].tagName;
  }
  return name;
};
// faster
function collectionLocal() {
  var coll = document.getElementsByTagName_r('div'),
  len = coll.length,
  name = '';
  for (var count = 0; count < len; count++) {
    name = coll[count].nodeName;
    name = coll[count].nodeType;
    name = coll[count].tagName;
  }
  return name;
};
// fastest
function collectionNodesLocal() {
  var coll = document.getElementsByTagName_r('div'),
  len = coll.length,
  name = '',
  el = null;
  for (var count = 0; count < len; count++) {
    el = coll[count];
    name = el.nodeName;
    name = el.nodeType;
    name = el.tagName;
  }
  return name;
};
```

Figure 3-5 shows the benefits of optimizing collection loops. The first bar plots how many times faster it is to access the collection through a local reference, and the second bar shows that there's additional benefit to caching collection items when they are accessed multiple times.

图 3-5 显示了优化集合循环的好处。第一条柱形图标出通过局部引用访问集合带来的速度提升，第二条柱形图显示出多次访问时缓冲集合项带来的速度提升。

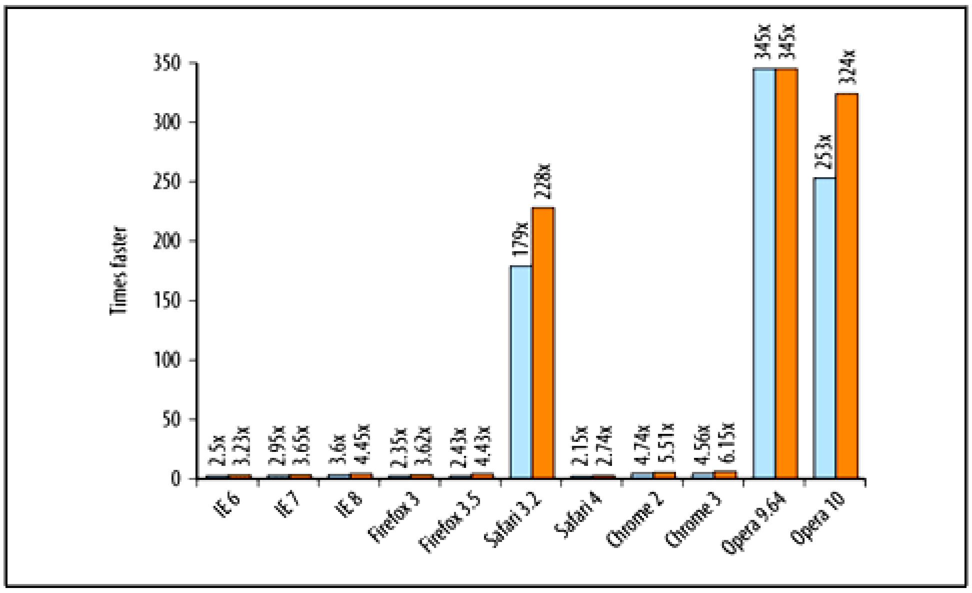


Figure 3-5. Benefit of using local variables to store references to a collection and its elements during loops

图 3-5　在循环中使用局部变量缓存集合引用和集合元素带来的速度提升

## Walking the DOM　DOM 漫谈

The DOM API provides multiple avenues to access specific parts of the overall document structure. In cases when you can choose between approaches, it's beneficial to use the most efficient API for a specific job.

DOM API 提供了多种途径访问整个文档结构的特定部分。当你在多种可行方法之间进行选择时，最好针对特定操作选择最有效的 API。

### Crawling the DOM　抓取 DOM

Often you need to start from a DOM element and work with the surrounding elements, maybe recursively iterating over all children. You can do so by using the **childNodes** collection or by getting each element's sibling using **nextSibling**.

你经常需要从一个 DOM 元素开始，操作周围的元素，或者递归迭代所有的子节点。你可以使用 childNode 集合或者使用 nextSibling 获得每个元素的兄弟节点。

Consider these two equivalent approaches to a nonrecursive visit of an element's children:

考虑这两个同样功能的例子，采用非递归方式遍历一个元素的子节点：

```
 function testNextSibling() {
  var el = document.getElementById('mydiv'),
  ch = el.firstChild,
  name = '';
  do {
    name = ch.nodeName;
  } while (ch = ch.nextSibling);
  return name;
};
function testChildNodes() {
 var el = document.getElementById('mydiv'),
  ch = el.childNodes,
  len = ch.length,
  name = '';
  for (var count = 0; count < len; count++) {
    name = ch[count].nodeName;
  }
  return name;
};
```

Bear in mind that **childNodes** is a collection and should be approached carefully, caching the **length** in loops so it's not updated on every iteration.

记住，childNodes 是一个集合，要小心处理，在循环中缓存 length 属性所以不会在每次迭代中更新。

The two approaches are mostly equal in terms of execution time across browsers. But in IE, **nextSibling** performs much better than **childNodes**. In IE6, **nextSibling** is 16 times faster, and in IE7 it's 105 times faster. Given these results, using **nextSibling** is the preferred method of crawling the DOM in older IE versions in performance-critical cases. In all other cases, it's mostly a question of personal and team preference.

在不同浏览器上，这两种方法的运行时间基本相等。但是在 IE 中，nextSibling 表现得比 childNode 好。在 IE6 中，nextSibling 比对手快 16 倍，而在 IE7 中快乐 105 倍。鉴于这些结果，在老的 IE 中性能严苛的使用条件下，用 nextSibling 抓取 DOM 是首选方法。在其他情况下，主要看个人和团队偏好。

**Element nodes 元素节点**

DOM properties such as **childNodes**, **firstChild**, and **nextSibling** don't distinguish between element nodes and other node types, such as comments and text nodes (which are often just spaces between two tags). In many cases, only the element nodes need to be accessed, so in a loop it's likely that the code needs to check the type of node returned and filter out nonelement nodes. This type checking and filtering is unnecessary DOM work.

DOM 属性诸如 childNode，firstChild，和 nextSibling 不区分元素节点和其他类型节点，如注释节点和文本节点（这两个标签之间往往只是一些空格）。在许多情况下，只有元素节点需要被访问，所以在循环中，似乎应当对节点返回类型进行检查，过滤出非元素节点。这些检查和过滤都是不必要的 DOM 操作。

Many modern browsers offer APIs that only return element nodes. It's better to use those when available, because they'll be faster than if you do the filtering yourself in JavaScript. Table 3-1 lists those convenient DOM properties.

许多现代浏览器提供了 API 函数只返回元素节点。如果可用最好利用起来，因为它们比你自己在 JavaScript 中写的过滤方法要快。表 3-1 列出这些便利的 DOM 属性。

Table 3-1. DOM properties that distinguish element nodes (HTML tags) versus all nodes

表 3-1 只表示元素节点的 DOM 属性（HTML 标签）和表示所有节点的属性

| Property | Use as a replacement for |
|---|---|
| `children` | `childNodes` |
| `childElementCount` | `childNodes.length` |
| `firstElementChild` | `firstChild` |
| `lastElementChild` | `lastChild` |
| `nextElementSibling` | `nextSibling` |
| `previousElementSibling` | `previousSibling` |

All of the properties listed in Table 3-1 are supported as of Firefox 3.5, Safari 4, Chrome 2, and Opera 9.62. Of these properties, IE versions 6, 7, and 8 only support **children**.

表 3-1 中列举的所有属性能够被 Firefox 3.5，Safari 4，Chrome 2，和 Opera 9.62 支持。所有这些属性中，IE6，7，8 只支持 children。

Looping over **children** instead of **childNodes** is faster because there are usually less items to loop over. Whitespaces in the HTML source code are actually text nodes, and they are not included in the **children** collection. **children** is faster than **childNodes** across all browsers, although usually not by a big margin—1.5 to 3 times faster. One notable exception is IE, where iterating over the **children** collection is significantly faster than iterating over **childNodes**—24 times faster in IE6 and 124 times faster in IE7.

遍历 children 比 childNodes 更快，因为集合项更少。HTML 源码中的空格实际上是文本节点，它们不包括在 children 集合中。在所有浏览器中 children 比 childNodes 更快，虽然差别不是太大，通常快 1.5 到 3 倍。特别值得注意的是 IE，遍历 children 明显快于遍历 childNodes——在 IE6 中快 24 倍，在 IE7 中快 124 倍。

**The Selectors API 选择器 API**

When identifying the elements in the DOM to work with, developers often need finer control than methods such as **getElementById**() and **getElementsByTagName**() can provide. Sometimes you combine these calls and iterate over the returned nodes in order to get to the list of elements you need, but this refinement process can become inefficient.

识别 DOM 中的元素时，开发者经常需要更精细的控制，而不仅是 getElementById()和 getElementsByTagName_r()之类的函数。有时你结合这些函数调用并迭代操作它们返回的节点，以获取所需要的元素，这一精细的过程可能造成效率低下。

On the other hand, using CSS selectors is a convenient way to identify nodes because developers are already familiar with CSS. Many JavaScript libraries have provided APIs for that purpose, and now recent browser versions provide a method called **querySelectorAll**() as a native browser DOM method. Naturally this approach is faster than using JavaScript and DOM to iterate and narrow down a list of elements.

另一方面，使用 CSS 选择器是一个便捷的确定节点的方法，因为开发者已经对 CSS 很熟悉了。许多 JavaScript 库为此提供了 API，而且最新的浏览器提供了一个名为 querySelectorAll()的原生浏览器 DOM 函数。显然这种方法比使用 JavaScript 和 DOM 迭代并缩小元素列表的方法要快。

Consider the following:

考虑下列代码：

```
var elements = document.querySelectorAll('#menu a');
```

The value of **elements** will contain a list of references to all a elements found inside an element with **id="menu"**. The method **querySelectorAll**() takes a CSS selector string as an argument and returns a **NodeList**—an array-like object containing matching nodes. The method doesn't return an HTML collection, so the returned nodes do not represent the live structure of the document. This avoids the performance (and potentially logic) issues with HTML collection discussed previously in this chapter.

elements 的值将包含一个引用列表，指向那些具有 id="menu"属性的元素。函数 querySelectorAll()接收一个 CSS 选择器字符串参数并返回一个 NodeList——由符合条件的节点构成的类数组对象。此函数不返回 HTML 集合，所以返回的节点不呈现文档的"存在性结构"。这就避免了本章前面提到的 HTML 集合所固有的性能问题（以及潜在的逻辑问题）。

To achieve the same goal as the preceding code without using **querySelectorAll**(), you will need the more verbose:

如果不使用 querySelectorAll()，达到同样的目标的代码会冗长一些：

```
var elements = document.getElementById('menu').getElementsByTagName_r('a');
```

In this case **elements** will be an HTML collection, so you'll also need to copy it into an array if you want the exact same type of static list as returned by querySelectorAll().

这种情况下 elements 将是一个 HTML 集合，所以你还需要将它拷贝到一个数组中，如果你想得到与 querySelectorAll()同样的返回值类型的话。

Using **querySelectorAll**() is even more convenient when you need to work with a union of several queries. For example, if the page has some div elements with a class name of "warning" and some with a class of "notice", to get a list of all of them you can use **querySelectorAll**():

当你需要联合查询时，使用 querySelectorAll()更加便利。例如，如果页面中有些 div 元素的 class 名称是"warning"，另一些 class 名是"notice"，你可以用 querySelectorAll()一次性获得这两类节点。

```
var errs = document.querySelectorAll('div.warning, div.notice');
```

Getting the same list without **querySelectorAll**() is considerably more work. One way is to select all div elements and iterate through them to filter out the ones you don't need.

如果不使用 querySelectorAll()，获得同样列表需要更多工作。一个办法是选择所有的 div 元素，然后通过迭代操作过滤出那些不需要的单元。

```
var errs = [],
divs = document.getElementsByTagName_r('div'),
classname = '';
for (var i = 0, len = divs.length; i < len; i++) {
  classname = divs[i].className;
  if (classname === 'notice' || classname === 'warning') {
    errs.push(divs[i]);
  }
}
```

Comparing the two pieces of code shows that using the Selectors API is 2 to 6 times faster across browsers (Figure 3-6).

比较这两段代码，使用选择器 API 比对手快了 2~6 倍（图 3-6）。

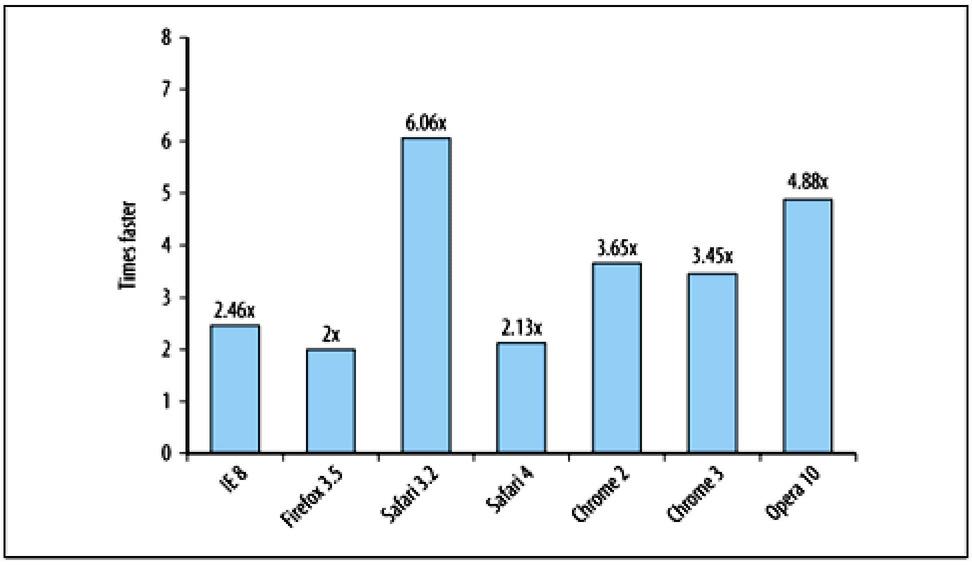


Figure 3-6. The benefit of using the Selectors API over iterating instead of the results of getElementsByTagName_r()

图 3-6　使用选择器 API 和使用 getElementsByTagName_r()的性能对比

The Selectors API is supported natively in browsers as of these versions: Internet Explorer 8, Firefox 3.5, Safari 3.1, Chrome 1, and Opera 10.

下列浏览器支持选择器 API：Internet Explorer 8，Firefox 3.5，Safari 3.1，Chrome 1，Opera 10。

As the results in the figure show, it's a good idea to check for support for **document.querySelectorAll()** and use it when available. Also, if you're using a selector API provided by a JavaScript library, make sure the library uses the native API under the hood. If not, you probably just need to upgrade the library version.

正如图中显示的那样，如果浏览器支持 document.querySelectorAll()，那么最好使用它。如果你使用 JavaScript 库所提供的选择器 API，确认一下该库是否确实使用了原生方法。如果不是，你大概需要将库升级到新版本。

You can also take advantage of another method called **querySelector**(), a convenient method that returns only the first node matched by the query.

你还可以从另一个函数 querySelector()获益，这个便利的函数只返回符合查询条件的第一个节点。

These two methods are properties of the DOM nodes, so you can use **document.querySelector('.myclass')** to query nodes in the whole document, or you can query a subtree using **elref.querySelector('.myclass')**, where elref is a reference to a DOM element.

这两个函数都是 DOM 节点的属性，所以你可以使用 document.querySelector('.myclass')来查询整个文档中的节点，或者使用 elref.querySelector('.myclass')在子树中进行查询，其中 elref 是一个 DOM 元素的引用。

## Repaints and Reflows 重绘和重排版

Once the browser has downloaded all the components of a page—HTML markup, JavaScript, CSS, images—it parses through the files and creates two internal data structures:

当浏览器下载完所有页面 HTML 标记，JavaScript，CSS，图片之后，它解析文件并创建两个内部数据结构：

A DOM tree

A representation of the page structure

一棵 DOM 树

表示页面结构

A render tree

A representation of how the DOM nodes will be displayed

一棵渲染树

表示 DOM 节点如何显示

The render tree has at least one node for every node of the DOM tree that needs to be displayed (hidden DOM elements don't have a corresponding node in the render tree). Nodes in the render tree are called frames or boxes in accordance with the CSS model that treats page elements as boxes with padding, margins, borders, and position. Once the DOM and the render trees are constructed, the browser can display ("paint") the elements on the page.

渲染树中为每个需要显示的 DOM 树节点存放至少一个节点（隐藏 DOM 元素在渲染树中没有对应节点）。渲染树上的节点称为“框”或者“盒”，符合 CSS 模型的定义，将页面元素看作一个具有填充、边距、边框和位置的盒。一旦 DOM 树和渲染树构造完毕，浏览器就可以显示（绘制）页面上的元素了。

When a DOM change affects the geometry of an element (width and height)—such as a change in the thickness of the border or adding more text to a paragraph, resulting in an additional line—the browser needs to recalculate the geometry of the element as well as the geometry and position of other elements that could have been affected by the change. The browser invalidates the part of the render tree that was affected by the change and reconstructs the render tree. This process is known as a reflow. Once the reflow is complete, the browser redraws the affected parts of the screen in a process called repaint.

当 DOM 改变影响到元素的几何属性（宽和高）——例如改变了边框宽度或在段落中添加文字，将发生一系列后续动作——浏览器需要重新计算元素的几何属性，而且其他元素的几何属性和位置也会因此改变受到影响。浏览器使渲染树上受到影响的部分失效，然后重构渲染树。这个过程被称作重排版。重排版完成时，浏览器在一个重绘进程中重新绘制屏幕上受影响的部分。

Not all DOM changes affect the geometry. For example, changing the background color of an element won't change its width or height. In this case, there is a repaint only (no reflow), because the layout of the element hasn't changed.

不是所有的 DOM 改变都会影响几何属性。例如，改变一个元素的背景颜色不会影响它的宽度或高度。在这种情况下，只需要重绘（不需要重排版），因为元素的布局没有改变。

Repaints and reflows are expensive operations and can make the UI of a web application less responsive. As such, it's important to reduce their occurrences whenever possible.

重绘和重排版是负担很重的操作，可能导致网页应用的用户界面失去相应。所以，十分有必要尽可能减少这类事情的发生。

## When Does a Reflow Happen?　重排版时会发生什么？

As mentioned earlier, a reflow is needed whenever layout and geometry change. This happens when:

正如前面所提到的，当布局和几何改变时需要重排版。在下述情况中会发生重排版：

• Visible DOM elements are added or removed

添加或删除可见的 DOM 元素

• Elements change position

元素位置改变

• Elements change size (because of a change in margin, padding, border thickness, width, height, etc.)

元素尺寸改变（因为边距，填充，边框宽度，宽度，高度等属性改变）

• Content is changed, e.g., text changes or an image is replaced with one of a different size

内容改变，例如，文本改变或图片被另一个不同尺寸的所替代

• Page renders initially

最初的页面渲染

• Browser window is resized

浏览器窗口改变尺寸

Depending on the nature of the change, a smaller or bigger part of the render tree needs to be recalculated. Some changes may cause a reflow of the whole page: for example, when a scroll bar appears.

根据改变的性质，渲染树上或大或小的一部分需要重新计算。某些改变可导致重排版整个页面：例如，当一个滚动条出现时。

## Queuing and Flushing Render Tree Changes 查询并刷新渲染树改变

Because of the computation costs associated with each reflow, most browsers optimize the reflow process by queuing changes and performing them in batches. However, you may (often involuntarily) force the queue to be flushed and require that all scheduled changes be applied right away. Flushing the queue happens when you want to retrieve layout information, which means using any of the following:

因为计算量与每次重排版有关，大多数浏览器通过队列化修改和批量显示优化重排版过程。然而，你可能（经常不由自主地）强迫队列刷新并要求所有计划改变的部分立刻应用。获取布局信息的操作将导致刷新队列动作，这意味着使用了下面这些方法：

- **offsetTop, offsetLeft, offsetWidth, offsetHeight**
- **scrollTop, scrollLeft, scrollWidth, scrollHeight**
- **clientTop, clientLeft, clientWidth, clientHeight**
- **getComputedStyle()** (**currentStyle** in IE)（在 IE 中此函数称为 **currentStyle**）

The layout information returned by these properties and methods needs to be up to date, and so the browser has to execute the pending changes in the rendering queue and reflow in order to return the correct values.

布局信息由这些属性和方法返回最新的数据，所以浏览器不得不运行渲染队列中待改变的项目并重新排版以返回正确的值。

During the process of changing styles, it's best not to use any of the properties shown in the preceding list. All of these will flush the render queue, even in cases where you're retrieving layout information that wasn't recently changed or isn't even relevant to the latest changes.

在改变风格的过程中，最好不要使用前面列出的那些属性。任何一个访问都将刷新渲染队列，即使你正在获取那些最近未发生改变的或者与最新的改变无关的布局信息。

Consider the following example of changing the same style property three times (this is probably not something you'll see in real code, but is an isolated illustration of an important topic):

考虑下面这个例子，它改变同一个风格属性三次（这也许不是你在真正的代码中所见到的，不过它孤立地展示出一个重要话题）：

```
// setting and retrieving styles in succession
var computed,
    tmp = '',
    bodystyle = document.body.style;
if (document.body.currentStyle) { // IE, Opera
  computed = document.body.currentStyle;
} else { // W3C
  computed = document.defaultView.getComputedStyle(document.body, '');
}
// inefficient way of modifying the same property
// and retrieving style information right after
bodystyle.color = 'red';
tmp = computed.backgroundColor;
bodystyle.color = 'white';
tmp = computed.backgroundImage;
bodystyle.color = 'green';
tmp = computed.backgroundAttachment;
```

In this example, the foreground color of the **body** element is being changed three times, and after every change, a computed style property is retrieved. The retrieved properties—**backgroundColor**, **backgroundImage**, and **backgroundAttachment**—are unrelated to the color being changed. Yet the browser needs to flush the render queue and reflow due to the fact that a computed style property was requested.

在这个例子中，body 元素的前景色被改变了三次，每次改变之后，都导入 computed 的风格。导入的属性 backgroundColor, backgroundImage, 和 backgroundAttachment 与颜色改变无关。然而，浏览器需要刷新渲染队列并重排版，因为 computed 的风格被查询而引发。

A better approach than this inefficient example is to never request layout information while it's being changed. If the computed style retrieval is moved to the end, the code looks like this:

比这个不讲效率的例子更好的方法是不要在布局信息改变时查询它。如果将查询 computed 风格的代码搬到末尾，代码看起来将是这个样子：

```
 bodystyle.color = 'red';
bodystyle.color = 'white';
bodystyle.color = 'green';
tmp = computed.backgroundColor;
tmp = computed.backgroundImage;
tmp = computed.backgroundAttachment;
```

The second example will be faster across all browsers, as seen in Figure 3-7.

在所有浏览器上，第二个例子将更快，如图 3-7 所示。

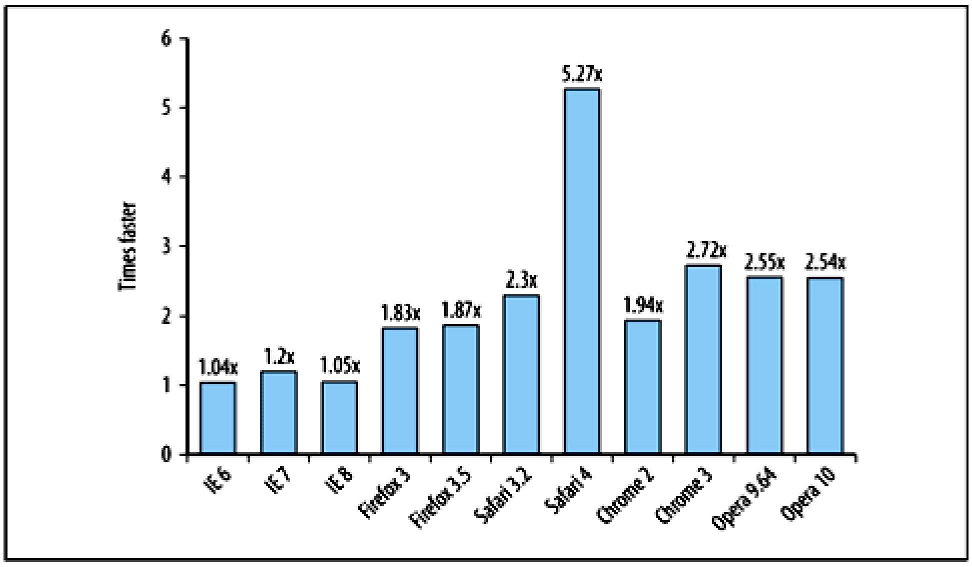


Figure 3-7. Benefit of preventing reflows by delaying access to layout information

图 3-7 通过延迟访问布局信息避免重排版而带来的性能提升

## Minimizing Repaints and Reflows 最小化重绘和重排版

Reflows and repaints can be expensive, and therefore a good strategy for responsive applications is to reduce their number. In order to minimize this number, you should combine multiple DOM and style changes into a batch and apply them once.

重排版和重绘代价昂贵，所以，提高程序响应速度一个好策略是减少此类操作发生的机会。为减少发生次数，你应该将多个 DOM 和风格改变合并到一个批次中一次性执行。

**Style changes 改变风格**

Consider this example:

考虑这个例子：

```
var el = document.getElementById('mydiv');
el.style.borderLeft = '1px';
el.style.borderRight = '2px';
el.style.padding = '5px';
```

Here there are three style properties being changed, each of them affecting the geometry of the element. In the worst case, this will cause the browser to reflow three times. Most modern browsers optimize for such cases and reflow only once, but it can still be inefficient in older browsers or if there's a separate asynchronous process happening at the same time (i.e., using a timer). If other code is requesting layout information while this code is running, it could cause up to three reflows. Also, the code is touching the DOM four times and can be optimized.

这里改变了三个风格属性，每次改变都影响到元素的几何属性。在这个糟糕的例子中，它导致浏览器重排版了三次。大多数现代浏览器优化了这种情况只进行一次重排版，但是在老式浏览器中，或者同时有一个分离的同步进程（例如使用了一个定时器），效率将十分低下。如果其他代码在这段代码运行时查询布局信息，将导致三次重布局发生。而且，此代码访问 DOM 四次，可以被优化。

A more efficient way to achieve the same result is to combine all the changes and apply them at once, modifying the DOM only once. This can be done using the cssText property:

一个达到同样效果而效率更高的方法是：将所有改变合并在一起执行，只修改 DOM 一次。可通过使用 cssText 属性实现：

```
var el = document.getElementById('mydiv');
el.style.cssText = 'border-left: 1px; border-right: 2px; padding: 5px;';
```

Modifying the cssText property as shown in the example overwrites existing style information, so if you want to keep the existing styles, you can append this to the cssText string:

这个例子中的代码修改 cssText 属性，覆盖已存在的风格信息。如果你打算保持当前的风格，你可以将它附加在 cssText 字符串的后面。

```
el.style.cssText += '; border-left: 1px;';
```

Another way to apply style changes only once is to change the CSS class name instead of changing the inline styles. This approach is applicable in cases when the styles do not depend on runtime logic and calculations. Changing the CSS class name is cleaner and more maintainable; it helps keep your scripts free of presentation code, although it might come with a slight performance hit because the cascade needs to be checked when changing classes.

另一个一次性改变风格的办法是修改 CSS 的类名称，而不是修改内联风格代码。这种方法适用于那些风格不依赖于运行逻辑，不需要计算的情况。改变 CSS 类名称更清晰，更易于维护；它有助于保持脚本免除显示代码，虽然它可能带来轻微的性能冲击，因为改变类时需要检查级联表。

```
var el = document.getElementById('mydiv');
el.className = 'active';
```

**Batching DOM changes 批量修改 DOM**

When you have a number of changes to apply to a DOM element, you can reduce the number of repaints and reflows by following these steps:

当你需要对 DOM 元素进行多次修改时，你可以通过以下步骤减少重绘和重排版的次数：

1. Take the element off of the document flow.

从文档流中摘除该元素

2. Apply multiple changes.

对其应用多重改变

3. Bring the element back to the document.

将元素带回文档中

This process causes two reflows—one at step 1 and one at step 3. If you omit those steps, every change you make in step 2 could cause its own reflows.

此过程引发两次重排版——第一步引发一次，第三步引发一次。如果你忽略了这两个步骤，那么第二步中每次改变都将引发一次重排版。

There are three basic ways to modify the DOM off the document:

有三种基本方法可以将 DOM 从文档中摘除：

• Hide the element, apply changes, and show it again.

隐藏元素，进行修改，然后再显示它。

• Use a document fragment to build a subtree outside of the live DOM and then copy it to the document.

使用一个文档片断在已存 DOM 之外创建一个子树，然后将它拷贝到文档中。

• Copy the original element into an off-document node, modify the copy, and then replace the original element once you're done.

将原始元素拷贝到一个脱离文档的节点中，修改副本，然后覆盖原始元素。

To illustrate the off-document manipulations, consider a list of links that must be updated with more information:

为演示脱离文档操作，考虑这样一个链接列表，它必须被更多的信息所更新：

```
<ul id="mylist">
  <li><a href="http://phpied.com">Stoyan</a></li>
  <li><a href="http://julienlecomte.com">Julien</a></li>
</ul>
```

Suppose additional data, already contained in an object, needs to be inserted into this list. The data is defined as:

假设附加数据已经存储在一个对象中了，需要插入到这个列表中。这些数据定义如下：

```
var data = [
  {
    "name": "Nicholas",
    "url":   "http://nczonline.net"
  },
  {
    "name": "Ross",
    "url":   "http://techfoolery.com"
  }
];
```

The following is a generic function to update a given node with new data:

下面是一个通用的函数，用于将新数据更新到指定节点中：

```
function appendDataToElement(appendToElement, data) {
  var a, li;
  for (var i = 0, max = data.length; i < max; i++) {
    a = document.createElement('a');
    a.href = data[i].url;
    a.appendChild(document.createTextNode(data[i].name));
    li = document.createElement('li');
    li.appendChild(a);
    appendToElement.appendChild(li);
  }
};
```

The most obvious way to update the list with the data without worrying about reflows would be the following:

将数据更新到列表而不管重排版问题，最明显的方法如下：

```
 var ul = document.getElementById('mylist');
appendDataToElement(ul, data);
```

Using this approach, however, every new entry from the **data** array will be appended to the live DOM tree and cause a reflow. As discussed previously, one way to reduce reflows is to temporarily remove the **<ul>** element from the document flow by changing the **display** property and then revert it:

使用这个方法，然而，data 队列上的每个新条目追加到 DOM 树都会导致重排版。如前面所讨论过的，减少重排版的一个方法是通过改变 display 属性，临时从文档上移除<ul>元素然后再恢复它。

```
 var ul = document.getElementById('mylist');
ul.style.display = 'none';
appendDataToElement(ul, data);
ul.style.display = 'block';
```

Another way to minimize the number of reflows is to create and update a document fragment, completely off the document, and then append it to the original list. A document fragment is a lightweight version of the **document** object, and it's designed to help with exactly this type of task—updating and moving nodes around. One syntactically convenient feature of the document fragments is that when you append a fragment to a node, the fragment's children actually get appended, not the fragment itself. The following solution takes one less line of code, causes only one reflow, and touches the live DOM only once:

另一种减少重排版次数的方法是：在文档之外创建并更新一个文档片断，然后将它附加在原始列表上。文档片断是一个轻量级的 document 对象，它被设计专用于更新、移动节点之类的任务。文档片断一个便利的语法特性是当你向节点附加一个片断时，实际添加的是文档片断的子节点群，而不是片断自己。下面的例子减少一行代码，只引发一次重排版，只触发“存在 DOM”一次。

```
 var fragment = document.createDocumentFragment();
appendDataToElement(fragment, data);
document.getElementById('mylist').appendChild(fragment);
```

A third solution would be to create a copy of the node you want to update, work on the copy, and then, once you're done, replace the old node with the newly updated copy:

第三种解决方法首先创建要更新节点的副本，然后在副本上操作，最后用新节点覆盖老节点：

```
 var old = document.getElementById('mylist');
var clone = old.cloneNode(true);
appendDataToElement(clone, data);
old.parentNode.replaceChild(clone, old);
```

The recommendation is to use document fragments (the second solution) whenever possible because they involve the least amount of DOM manipulations and reflows. The only potential drawback is that the practice of using document fragments is currently underused and some team members may not be familiar with the technique.

推荐尽可能使用文档片断（第二种解决方案）因为它涉及最少数量的 DOM 操作和重排版。唯一潜在的缺点是，当前文档片断还没有得到充分利用，开发者可能不熟悉此技术。

## Caching Layout Information 缓冲布局信息

As already mentioned, browsers try to minimize the number of reflows by queuing changes and executing them in batches. But when you request layout information such as offsets, scroll values, or computed style values, the browser flushes the queue and applies all the changes in order to return the updated value. It is best to minimize the number of requests for layout information, and when you do request it, assign it to local variables and work with the local values.

浏览器通过队列化修改和批量运行的方法，尽量减少重排版次数。当你查询布局信息如偏移量、滚动条位置，或风格属性时，浏览器刷队列并执行所有修改操作，以返回最新的数值。最好是尽量减少对布局信息的查询次数，查询时将它赋给局部变量，并用局部变量参与计算。

Consider an example of moving an element **myElement** diagonally, one pixel at a time, starting from position 100 × 100px and ending at 500 × 500px. In the body of a timeout loop you could use:

考虑一个例子，将元素 myElement 向右下方向平移，每次一个像素，起始于 100x100 位置，结束于 500x500 位置，在 timeout 循环体中你可以使用：

```
// inefficient
myElement.style.left = 1 + myElement.offsetLeft + 'px';
myElement.style.top = 1 + myElement.offsetTop + 'px';
if (myElement.offsetLeft >= 500) {
  stopAnimation();
}
```

This is not efficient, though, because every time the element moves, the code requests the offset values, causing the browser to flush the rendering queue and not benefit from its optimizations. A better way to do the same thing is to take the start value position once and assign it to a variable such as **var current = myElement.offsetLeft;**. Then, inside of the animation loop, work with the **current** variable and don't request offsets:

这样做很没效率，因为每次元素移动，代码查询偏移量，导致浏览器刷新渲染队列，并没有从优化中获益。另一个办法只需要获得起始位置值一次，将它存入局部变量中 var current = myElement.offsetLeft;。然后，在动画循环中，使用 current 变量而不再查询偏移量：

```
current++
myElement.style.left = current + 'px';
myElement.style.top = current + 'px';
if (current >= 500) {
  stopAnimation();
}
```

## Take Elements Out of the Flow for Animations 将元素提出动画流

Showing and hiding parts of a page in an expand/collapse manner is a common interaction pattern. It often includes geometry animation of the area being expanded, which pushes down the rest of the content on the page.

显示和隐藏部分页面构成展开/折叠动画是一种常见的交互模式。它通常包括区域扩大的几何动画，将页面其他部分推向下方。

Reflows sometimes affect only a small part of the render tree, but they can affect a larger portion, or even the whole tree. The less the browser needs to reflow, the more responsive your application will be. So when an animation at the top of the page pushes down almost the whole page, this will cause a big reflow and can be expensive, appearing choppy to the user. The more nodes in the render tree that need recalculation, the worse it becomes.

重排版有时只影响渲染树的一小部分，但也可以影响很大的一部分，甚至整个渲染树。浏览器需要重排版的部分越小，应用程序的响应速度就越快。所以当一个页面顶部的动画推移了差不多整个页面时，将引发巨大的重排版动作，使用户感到动画卡顿。渲染树的大多数节点需要被重新计算，它变得更糟糕。

A technique to avoid a reflow of a big part of the page is to use the following steps:

使用以下步骤可以避免对大部分页面进行重排版：

1. Use absolute positioning for the element you want to animate on the page, taking it out of the layout flow of the page.

使用绝对坐标定位页面动画的元素，使它位于页面布局流之外。

2. Animate the element. When it expands, it will temporarily cover part of the page. This is a repaint, but only of a small part of the page instead of a reflow and repaint of a big page chunk.

启动元素动画。当它扩大时，它临时覆盖部分页面。这是一个重绘过程，但只影响页面的一小部分，避免重排版并重绘一大块页面。

3. When the animation is done, restore the positioning, thereby pushing down the rest of the document only once.

当动画结束时，重新定位，从而只一次下移文档其他元素的位置。

译者注：文字描述比较简单概要，我对这三步的理解如下：

1、页面顶部可以"折叠/展开"的元素称作"动画元素"，用绝对坐标对它进行定位，当它的尺寸改变时，就不会推移页面中其他元素的位置，而只是覆盖其他元素。

2、展开动作只在"动画元素"上进行。这时其他元素的坐标并没有改变，换句话说，其他元素并没有因为"动画元素"的扩大而随之下移，而是任由动画元素覆盖。

3、"动画元素"的动画结束时，将其他元素的位置下移到动画元素下方，界面"跳"了一下。

**IE and :hover  IE 和:hover**

Since version 7, IE can apply the **:hover** CSS pseudo-selector on any element (in strict mode). However, if you have a significant number of elements with a **:hover**, the responsiveness degrades. The problem is even more visible in IE 8.

自从版本 7 之后，IE 可以在任何元素（严格模式）上使用:hover 这个 CSS 伪选择器。然而，如果大量的元素使用了:hover 那么会降低反应速度。此问题在 IE8 中更显著。

For example, if you create a table with 500–1000 rows and 5 columns and use **tr:hover** to change the background color and highlight the row the user is on, the performance degrades as the user moves over the table. The highlight is slow to apply, and the CPU usage increases to 80%–90%. So avoid this effect when you work with a large number of elements, such as big tables or long item lists.

例如，如果你创建了一个由 500-1000 行 5 列构成的表，并使用 tr:hover 改变背景颜色，高亮显示鼠标光标所在的行，当鼠标光标在表上移动时，性能会降低。使用高亮是个慢速过程，CPU 使用率会提高到 80%-90%。所以当元素数量很多时避免使用这种效果，诸如很大的表或很长的列表。

## Event Delegation　事件托管

When there are a large number of elements on a page and each of them has one or more event handlers attached (such as **onclick**), this may affect performance. Attaching every handler comes at a price—either in the form of heavier pages (more markup or JavaScript code) or in the form of runtime execution time. The more DOM nodes you need to touch and modify, the slower your application, especially because the event attaching phase usually happens at the **onload** (or **DOMContentReady**) event, which is a busy time for every interaction-rich web page. Attaching events takes processing time, and, in addition, the browser needs to keep track of each handler, which takes up memory. And at the end of it, a great number of these event handlers might never be needed(because the user clicked one button or link, not all 100 of them, for example), so a lot of the work might not be necessary.

当页面中存在大量元素，而且每个元素有一个或多个事件句柄与之挂接（例如 onclick）时，可能会影响性能。连接每个句柄都是有代价的，无论其形式是加重了页面负担（更多的页面标记和 JavaScript 代码）还是表现在运行期的运行时间上。你需要访问和修改更多的 DOM 节点，程序就会更慢，特别是因为事件挂接过程都发生在 onload（或 DOMContentReady）事件中，对任何一个富交互网页来说那都是一个繁忙的时间段。挂接事件占用了处理时间，另外，浏览器需要保存每个句柄的记录，占用更多内存。当这些工作结束时，这些事件句柄中的相当一部分根本不需要（因为并不是 100%的按钮或者链接都会被用户点到），所以很多工作都是不必要的。

A simple and elegant technique for handling DOM events is event delegation. It's based on the fact that events bubble up and can be handled by a parent element. With event delegation, you attach only one handler on a wrapper element to handle all events that happen to the children descendant of that parent wrapper.

一个简单而优雅的处理 DOM 事件的技术是事件托管。它基于这样一个事实：事件逐层冒泡总能被父元素捕获。采用事件托管技术之后，你只需要在一个包装元素上挂接一个句柄，用于处理子元素发生的所有事件。

According to the DOM standard, each event has three phases:

根据 DOM 标准，每个事件有三个阶段：

• Capturing

捕获

• At target

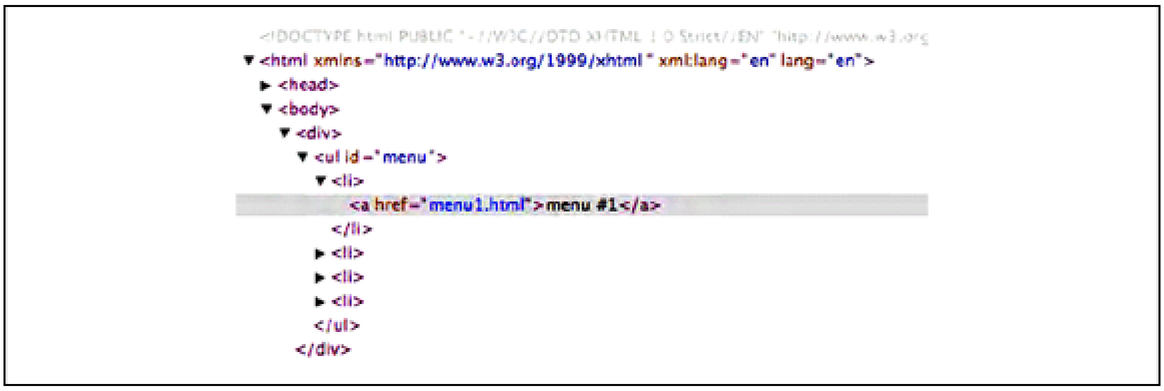
到达目标

• Bubbling

冒泡

Capturing is not supported by IE, but bubbling is good enough for the purposes of delegation. Consider a page with the structure shown in Figure 3-8.

IE 不支持捕获，但实现托管技术使用冒泡就足够了。考虑图 3-8 所示的页面结构。



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org
▼ <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  ▶ <head>
  ▼ <body>
    ▼ <div>
      ▼ <ul id="menu">
        ▼ <li>
            <a href="menu1.html">menu #1</a>
          </li>
        ▶ <li>
        ▶ <li>
        ▶ <li>
        </ul>
      </div>
```

Figure 3-8. An example DOM tree

图 3-8　一个 DOM 树的例子

When the user clicks the "menu #1" link, the click event is first received by the **<a>** element. Then it bubbles up the DOM tree and is received by the **<li>** element, then the **<ul>**, then the **<div>**, and so on, all the way to the top of the document and even the **window**. This allows you to attach only one event handler to a parent element and receive notifications for all events that happen to the children.

当用户点击了“menu #1”链接，点击事件首先被<a>元素收到。然后它沿着 DOM 树冒泡，被<li>元素收到，然后是<ul>，接着是<div>，等等，一直到达文档的顶层，甚至 window。这使得你可以只在父元素上挂接一个事件句柄，来接收所有子元素产生的事件通知。

Suppose that you want to provide a progressively enhanced Ajax experience for the document shown in the figure. If the user has JavaScript turned off, then the links in the menu work normally and reload the page. But if JavaScript is on and the user agent is capable enough, you want to intercept all clicks, prevent the default behavior (which is to follow the link), send an Ajax request to get the content, and update a portion of the page without a refresh. To do this using event delegation, you can attach a click listener to the UL "menu" element that wraps all links and inspect all clicks to see whether they come from a link.

假设你要为图中所显示的文档提供一个逐步增强的 Ajax 体验。如果用户关闭了 JavaScript，菜单中的链接仍然可以正常地重载页面。但是如果 JavaScript 打开而且用户代理有足够能力，你希望截获所有点击，阻止默认行为（转入链接），发送一个 Ajax 请求获取内容，然后不刷新页面就能够更新部分页面。使用事件托管实现此功能，你可以在 UL"menu"单元挂接一个点击监听器，它封装所有链接并监听所有 click 事件，看看他们是否发自一个链接。

```
document.getElementById('menu').onclick = function(e) {
  // x-browser target
  e = e || window.event;
  var target = e.target || e.srcElement;
  var pageid, hrefparts;
  // only interesed in hrefs
  // exit the function on non-link clicks
  if (target.nodeName !== 'A') {
    return;
  }
  // figure out page ID from the link
  hrefparts = target.href.split('/');
  pageid = hrefparts[hrefparts.length - 1];
  pageid = pageid.replace('.html', '');
  // update the page
  ajaxRequest('xhr.php?page=' + id, updatePageContents);
  // x-browser prevent default action and cancel bubbling
  if (typeof e.preventDefault === 'function') {
    e.preventDefault();
    e.stopPropagation();
  } else {
    e.returnValue = false;
    e.cancelBubble = true;
  }
};
```

As you can see, the event delegation technique is not complicated; you only need to inspect events to see whether they come from elements you're interested in. There's a little bit of verbose cross-browser code, but if you move this part to a reusable library, the code becomes pretty clean. The cross-browser parts are:

正如你所看到的那样，事件托管技术并不复杂；你只需要监听事件，看看他们是不是从你感兴趣的元素中发出的。这里有一些冗余的跨浏览器代码，如果你将它们移入一个可重用的库中，代码就变得相当干净。跨浏览器部分包括：

• Access to the event object and identifying the source (target) of the event

访问事件对象，并判断事件源（目标）

• Cancel the bubbling up the document tree (optional)

结束文档树上的冒泡（可选）

• Prevent the default action (optional, but needed in this case because the task was to trap the links and not follow them)

阻止默认动作（可选，但此例中是必须的，因为任务是捕获链接而不转入这些链接）

## Summary 总结

DOM access and manipulation are an important part of modern web applications. But every time you cross the bridge from ECMAScript to DOM-land, it comes at a cost. To reduce the performance costs related to DOM scripting, keep the following in mind:

DOM 访问和操作是现代网页应用中很重要的一部分。但每次你通过桥梁从 ECMAScript 岛到达 DOM 岛时，都会被收取"过桥费"。为减少 DOM 编程中的性能损失，请牢记以下几点：

• Minimize DOM access, and try to work as much as possible in JavaScript.

最小化 DOM 访问，在 JavaScript 端做尽可能多的事情。

• Use local variables to store DOM references you'll access repeatedly.

在反复访问的地方使用局部变量存放 DOM 引用.

• Be careful when dealing with HTML collections because they represent the live, underlying document. Cache the collection **length** into a variable and use it when iterating, and make a copy of the collection into an array for heavy work on collections.

小心地处理 HTML 集合，因为他们表现出"存在性"，总是对底层文档重新查询。将集合的 length 属性缓存到一个变量中，在迭代中使用这个变量。如果经常操作这个集合，可以将集合拷贝到数组中。

• Use faster APIs when available, such as **querySelectorAll**() and **firstElementChild**.

如果可能的话，使用速度更快的 API，诸如 querySelectorAll()和 firstElementChild。

• Be mindful of repaints and reflows; batch style changes, manipulate the DOM tree "offline," and cache and minimize access to layout information.

注意重绘和重排版；批量修改风格，离线操作 DOM 树，缓存并减少对布局信息的访问。

• Position absolutely during animations, and use drag and drop proxies.

动画中使用绝对坐标，使用拖放代理。

• Use event delegation to minimize the number of event handlers.

使用事件托管技术最小化事件句柄数量。

# 第四章 Algorithms and Flow Control 算法和流程控制

The overall structure of your code is one of the main determinants as to how fast it will execute. Having a very small amount of code doesn't necessarily mean that it will run quickly, and having a large amount of code doesn't necessarily mean that it will run slowly. A lot of the performance impact is directly related to how the code has been organized and how you're attempting to solve a given problem.

代码整体结构是执行速度的决定因素之一。代码量少不一定运行速度快，代码量多也不一定运行速度慢。性能损失与代码组织方式和具体问题解决办法直接相关。

The techniques in this chapter aren't necessarily unique to JavaScript and are often taught as performance optimizations for other languages. There are some deviations from advice given for other languages, though, as there are many more JavaScript engines to deal with and their quirks need to be considered, but all of the techniques are based on prevailing computer science knowledge.

本章技术不仅适用于 JavaScript 也适用于其他语言的性能优化。还有一些为其他语言提供的建议，还要处理多种 JavaScript 引擎并考虑它们的差异，但这些技术都以当前计算机科学知识为基础。

## Loops 循环

In most programming languages, the majority of code execution time is spent within loops. Looping over a series of values is one of the most frequently used patterns in programming and as such is also one of the areas where efforts to improve performance must be focused. Understanding the performance impact of loops in JavaScript is especially important, as infinite or long-running loops severely impact the overall user experience.

在大多数编程语言中，代码执行时间多数在循环中度过。在一系列编程模式中，循环是最常用的模式之一，因此也是提高性能必须关注的地区之一。理解 JavaScript 中循环对性能的影响至关重要，因为死循环或者长时间运行的循环会严重影响用户体验。

## Types of Loops 循环的类型

ECMA-262, 3rd Edition, the specification that defines JavaScript's basic syntax and behavior, defines four types of loops. The first is the standard **for** loop, which shares its syntax with other C-like languages:

ECMA-263 标准第三版规定了 JavaScript 的基本语法和行为，定义了四种类型的循环。第一个是标准的 for 循环，与类 C 语言使用同样的语法：

```
 for (var i=0; i < 10; i++){
   //loop body
}
```

The for loop tends to be the most commonly used JavaScript looping construct. There are four parts to the **for** loop: initialization, pretest condition, post-execute, and the loop body. When a **for** loop is encountered, the initialization code is executed first, followed by the pretest condition. If the pretest condition evaluates to **true**, then the body of the loop is executed. After the body is executed, the post-execute code is run. The perceived encapsulation of the **for** loop makes it a favorite of developers.

for 循环大概是最常用的 JavaScript 循环结构。它由四部分组成：初始化体，前测条件，后执行体，循环体。当遇到一个 for 循环时，初始化体首先执行，然后进入前测条件。如果前测条件的计算结果为 true，则执行循环体。然后运行后执行体。for 循环封装上的直接性是开发者喜欢的原因。

The second type of loop is the **while** loop. A **while** loop is a simple pretest loop comprised of a pretest condition and a loop body:

第二种循环是 while 循环。while 循环是一个简单的预测试循环，由一个预测试条件和一个循环体构成：

```
 var i = 0;
while(i < 10){
  //loop body
  i++;
}
```

Before the loop body is executed, the pretest condition is evaluated. If the condition evaluates to **true**, then the loop body is executed; otherwise, the loop body is skipped. Any **for** loop can also be written as a **while** loop and vice versa.

在循环体执行之前，首先对前测条件进行计算。如果计算结果为 true，那么就执行循环体；否则循环体将被跳过。任何 for 循环都可以写成 while 循环，反之亦然。

The third type of loop is the **do-while** loop. A **do-while** loop is the only post-test loop available in JavaScript and is made up of two parts, the loop body and the post-test condition:

第三种循环类型是 do-while 循环。do-while 循环是 JavaScript 中唯一一种后测试的循环，它包括两部分：循环体和后测试条件体：

```
var i = 0;
do {
  //loop body
} while (i++ < 10);
```

In a **do-while** loop, the loop body is always executed at least once, and the post-test condition determines whether the loop should be executed again.

在一个 do-while 循环中，循环体至少运行一次，后测试条件决定循环体是否应再次执行。

The fourth and last loop is the **for-in** loop. This loop has a very special purpose: it enumerates the named properties of any object. The basic format is as follows:

第四种也是最后一种循环称为 for-in 循环。此循环有一个非常特殊的用途：它可以枚举任何对象的命名属性。其基本格式如下：

```
for (var prop in object){
 //loop body
}
```

Each time the loop is executed, the prop variable is filled with the name of another property (a string) that exists on the object until all properties have been returned. The returned properties are both those that exist on the object instance and those inherited through its prototype chain.

每次循环执行，属性变量被填充以对象属性的名字（一个字符串），直到所有的对象属性遍历完成才返回。返回的属性包括对象的实例属性和它从原型链继承而来的属性。

## Loop Performance 循环性能

A constant source of debate regarding loop performance is which loop to use. Of the four loop types provided by JavaScript, only one of them is significantly slower than the others: the **for-in** loop.

循环性能争论的源头是应当选用哪种循环。在 JavaScript 提供的四种循环类型中，只有一种循环比其他循环明显要慢：for-in 循环。

Since each iteration through the loop results in a property lookup either on the instance or on a prototype, the **for-in** loop has considerably more overhead per iteration and is therefore slower than the other loops. For the same number of loop iterations, a **for-in** loop can end up as much as seven times slower than the other loop types. For this reason, it's recommended to avoid the **for-in** loop unless your intent is to iterate over an unknown number of object properties. If you have a finite, known list of properties to iterate over, it is faster to use one of the other loop types and use a pattern such as this:

由于每次迭代操作要搜索实例或原形的属性，for-in 循环每次迭代都要付出更多开销，所以比其他类型循环慢一些。在同样的循环迭代操作中，for-in 循环比其他类型的循环慢 7 倍之多。因此推荐的做法如下：除非你需要对数目不详的对象属性进行操作，否则避免使用 for-in 循环。如果你迭代遍历一个有限的，已知的属性列表，使用其他循环类型更快，可使用如下模式：

```
 var props = ["prop1", "prop2"],
i = 0;
while (i < props.length){
  process(object[props[i]]);
}
```

This code creates an array whose members are property names. The **while** loop is used to iterate over this small number of properties and process the appropriate member on **object**. Rather than looking up each and every property on **object**, the code focuses on only the properties of interest, saving loop overhead and time.

此代码创建一个由成员和属性名构成的队列。while 循环用于遍历这几个属性并处理所对应的对象成员，而不是遍历对象的每个属性。此代码只关注感兴趣的属性，节约了循环时间。

Aside from the **for-in** loop, all other loop types have equivalent performance characteristics such that it's not useful to try to determine which is fastest. The choice of loop type should be based on your requirements rather than performance concerns.

除 for-in 循环外，其他循环类型性能相当，难以确定哪种循环更快。选择循环类型应基于需求而不是性能。

If loop type doesn't contribute to loop performance, then what does? There are actually just two factors:

如果循环类型与性能无关，那么如何选择？其实只有两个因素：

• Work done per iteration

每次迭代干什么

• Number of iterations

迭代的次数

By decreasing either or both of these, you can positively impact the overall performance of the loop.

通过减少这两者中一个或者全部（的执行时间），你可以积极地影响循环的整体性能。

## Decreasing the work per iteration　减少迭代的工作量

It stands to reason that if a single pass through a loop takes a long time to execute, then multiple passes through the loop will take even longer. Limiting the number of expensive operations done in the loop body is a good way to speed up the entire loop.

不言而喻，如果一次循环迭代需要较长时间来执行，那么多次循环将需要更长时间。限制在循环体内进行耗时操作的数量是一个加快循环的好方法。

A typical array-processing loop can be created using any of the three faster loop types. The code is most frequently written as follows:

一个典型的数组处理循环，可使用三种循环的任何一种。最常用的代码写法如下：

```
//original loops
for (var i=0; i < items.length; i++){
  process(items[i]);
}
var j=0;
while (j < items.length){
  process(items[j++]);
}
var k=0;
do {
  process(items[k++]);
} while (k < items.length);
```

In each of these loops, there are several operations happening each time the loop body is executed:

在每个循环中，每次运行循环体都要发生如下几个操作：

1. One property lookup (**items.length**) in the control condition

在控制条件中读一次属性（items.length）

2. One comparison (**i < items.length**) in the control condition

在控制条件中执行一次比较（i < items.length）

3. One comparison to see whether the control condition evaluates to **true** (**i<items.length==true**)

比较操作，察看条件控制体的运算结果是不是 true（i < items.length == true）

4. One increment operation (**i++**)

一次自加操作（i++）

5. One array lookup (**items[i]**)

一次数组查找（items[i]）

6. One function call (**process(items[i])**)

一次函数调用（process(items[i])）

There's a lot going on per iteration of these simple loops, even though there's not much code. The speed at which the code will execute is largely determined by what **process**() does to each item, but even so, reducing the total number of operations per iteration can greatly improve the overall loop performance.

在这些简单的循环中，即使没有太多的代码，每次迭代也要进行许多操作。代码运行速度很大程度上由process()对每个项目的操作所决定，即使如此，减少每次迭代中操作的总数可以大幅度提高循环整体性能。

The first step in optimizing the amount of work in a loop is to minimize the number of object member and array item lookups. As discussed in Chapter 2, these take significantly longer to access in most browsers versus local variables or literal values. The previous examples do a property lookup for **items.length** each and every time through the loop. Doing so is wasteful, as this value won't change during the execution of the loop and is therefore an unnecessary performance hit. You can improve the loop performance easily by doing the property lookup once, storing the value in a local variable, and then using that variable in the control condition:

优化循环工作量的第一步是减少对象成员和数组项查找的次数。正如第 2 章讨论的，在大多数浏览器上，这些操作比访问局部变量或直接量需要更长时间。前面的例子中每次循环都查找 items.length。这是一种浪费，因为该值在循环体执行过程中不会改变，因此产生了不必要的性能损失。你可以简单地将此值存入一个局部变量中，在控制条件中使用这个局部变量，从而提高了循环性能：

```
//minimizing property lookups
for (var i=0, len=items.length; i < len; i++){
  process(items[i]);
}
var j=0,
count = items.length;
while (j < count){
  process(items[j++]);
}
var k=0,
num = items.length;
do {
  process(items[k++]);
} while (k < num);
```

Each of these rewritten loops makes a single property lookup for the array length prior to the loop executing. This allows the control condition to be comprised solely of local variables and therefore run much faster. Depending on the length of the array, you can save around 25% off the total loop execution time in most browsers (and up to 50% in Internet Explorer).

这些重写后的循环只在循环执行之前对数组长度进行一次属性查询。这使得控制条件只有局部变量参与运算，所以速度更快。根据数组的长度，在大多数浏览器上你可以节省大约 25%的总循环时间（在 Internet Explorer 可节省 50%）。

You can also increase the performance of loops by reversing their order. Frequently, the order in which array items are processed is irrelevant to the task, and so starting at the last item and processing toward the first item is an acceptable alternative. Reversing loop order is a common performance optimization in programming languages but generally isn't very well understood. In JavaScript, reversing a loop does result in a small performance improvement for loops, provided that you eliminate extra operations as a result:

你还可以通过改变他们的顺序提高循环性能。通常，数组元素的处理顺序与任务无关，你可以从最后一个开始，直到处理完第一个元素。倒序循环是编程语言中常用的性能优化方法，但一般来说不太容易理解。在 JavaScript 中，倒序循环可以略微提高循环性能，只要你消除因此而产生的额外操作：

```
//minimizing property lookups and reversing
for (var i=items.length; i--; ){
  process(items[i]);
}
var j = items.length;
while (j--){
  process(items[j]]);
}
var k = items.length-1;
do {
  process(items[k]);
} while (k--);
```

The loops in this example are reversed and combine the control condition with the decrement operation. Each control condition is now simply a comparison against zero. Control conditions are compared against the value **true**, and any nonzero number is automatically coerced to **true**, making zero the equivalent of **false**. Effectively, the control condition has been changed from two comparisons (is the iterator less than the total and is that equal to **true**?) to just a single comparison (is the value **true**?). Cutting down from two comparisons per iteration to one speeds up the loops even further. By reversing loops and minimizing property lookups, you can see execution times that are up to 50%–60% faster than the original.

例子中使用了倒序循环，并在控制条件中使用了减法。每个控制条件只是简单地与零进行比较。控制条件与 true 值进行比较，任何非零数字自动强制转换为 true，而零等同于 false。实际上，控制条件已经从两

次比较（迭代少于总数吗？它等于 true 吗？）减少到一次比较（它等于 true 吗？）。将每个迭代中两次比较减少到一次可以大幅度提高循环速度。通过倒序循环和最小化属性查询，你可以看到执行速度比原始版本快了 50%-60%。

As a comparison to the originals, here are the operations being performed per iteration for these loops:

与原始版本相比，每次迭代中只进行如下操作：

1. One comparison (**i == true**) in the control condition

   在控制条件中进行一次比较（i == true）

2. One decrement operation (**i--**)

   一次减法操作（i--）

3. One array lookup (**items[i]**)

   一次数组查询（items[i]）

4. One function call (**process(items[i])**)

   一次函数调用（process(items[i])）

The new loop code has two fewer operations per iteration, which can lead to increasing performance gains as the number of iterations increases.

新循环代码每次迭代中减少两个操作，随着迭代次数的增长，性能将显著提升。

**Decreasing the number of iterations** 减少迭代次数

Even the fastest code in a loop body will add up when iterated thousands of times. Additionally, there is a small amount of performance overhead associated with executing a loop body, which just adds to the overall execution time. Decreasing the number of iterations throughout the loop can therefore lead to greater performance gains. The most well known approach to limiting loop iterations is a pattern called Duff's Device.

即使循环体中最快的代码，累计迭代上千次（也将是不小的负担）。此外，每次运行循环体时都会产生一个很小的性能开销，也会增加总的运行时间。减少循环的迭代次数可获得显著的性能提升。最广为人知的限制循环迭代次数的模式称作“达夫设备”。

Duff's Device is a technique of unrolling loop bodies so that each iteration actually does the job of many iterations. Jeff Greenberg is credited with the first published port of Duff's Device to JavaScript from its original implementation in C. A typical implementation looks like this:

达夫设备是一个循环体展开技术，在一次迭代中实际上执行了多次迭代操作。Jeff Greenberg 被认为是将达夫循环从原始的 C 实现移植到 JavaScript 中的第一人。一个典型的实现如下：

```
//credit: Jeff Greenberg
var iterations = Math.floor(items.length / 8),
startAt = items.length % 8,
i = 0;
do {
  switch(startAt){
    case 0: process(items[i++]);
    case 7: process(items[i++]);
    case 6: process(items[i++]);
    case 5: process(items[i++]);
    case 4: process(items[i++]);
    case 3: process(items[i++]);
    case 2: process(items[i++]);
    case 1: process(items[i++]);
  }
  startAt = 0;
} while (--iterations);
```

The basic idea behind this Duff's Device implementation is that each trip through the loop is allowed a maximum of eight calls to **process**(). The number of iterations through the loop is determined by dividing the total number of items by eight. Because not all numbers are evenly divisible by eight, the **startAt** variable holds the remainder and indicates how many calls to **process**() will occur in the first trip through the loop. If there were 12 items, then the first trip through the loop would call **process**() 4 times, and then the second trip would call **process**() 8 times, for a total of two trips through the loop instead of 12.

达夫设备背后的基本理念是：每次循环中最多可 8 次调用 process()函数。循环迭代次数为元素总数除以 8。因为总数不一定是 8 的整数倍，所以 startAt 变量存放余数，指出第一次循环中应当执行多少次 process()。比方说现在有 12 个元素，那么第一次循环将调用 process()4 次，第二次循环调用 process()8 次，用 2 次循环代替了 12 次循环。

A slightly faster version of this algorithm removes the **switch** statement and separates the remainder processing from the main processing:

此算法一个稍快的版本取消了 switch 表达式，将余数处理与主循环分开：

```
//credit: Jeff Greenberg
var i = items.length % 8;
while(i){
  process(items[i--]);
}
i = Math.floor(items.length / 8);
while(i){
  process(items[i--]);
  process(items[i--]);
  process(items[i--]);
  process(items[i--]);
  process(items[i--]);
  process(items[i--]);
  process(items[i--]);
  process(items[i--]);
}
```

Even though this implementation is now two loops instead of one, it runs faster than the original by removing the **switch** statement from the loop body.

虽然此代码中使用两个循环替代了先前的一个，但它去掉了循环体中的 switch 表达式，速度更快。

Whether or not it's worthwhile to use Duff's Device, either the original or the modified version, depends largely on the number of iterations you're already doing. In cases where the loop iterations are less than 1,000, you're likely to see only an insignificant amount of performance improvement over using a regular loop construct. As the number of iterations increases past 1,000, however, the efficacy of Duff's Device increases significantly. At 500,000 iterations, for instance, the execution time is up to 70% less than a regular loop.

是否值得使用达夫设备，无论是原始的版本还是修改后的版本，很大程度上依赖于迭代的次数。如果循环迭代次数少于 1'000 次，你可能只看到它与普通循环相比只有微不足道的性能提升。如果迭代次数超过 1'000 次，达夫设备的效率将明显提升。例如 500'000 次迭代中，运行时间比普通循环减少到 70%。

## Function-Based Iteration 基于函数的迭代

The fourth edition of ECMA-262 introduced a new method on the native array object call **forEach**(). This method iterates over the members of an array and runs a function on each. The function to be run on each item is passed into **forEach**() as an argument and will receive three arguments when called, which are the array item value, the index of the array item, and the array itself. The following is an example usage:

ECMA-262 标准第四版介绍了本地数组对象的一个新方法 forEach()。此方法遍历一个数组的所有成员，并在每个成员上执行一个函数。在每个元素上执行的函数作为 forEach()的参数传进去，并在调用时接收三个参数，它们是：数组项的值，数组项的索引，和数组自身。下面是用法举例：

```
items.forEach(function(value, index, array){
  process(value);
});
```

The **forEach**() method is implemented natively in Firefox, Chrome, and Safari. Additionally, most JavaScript libraries have the logical equivalent:

forEach()函数在 Firefox，Chrome，和 Safari 中为原生函数。另外，大多数 JavaScript 库都有等价实现：

```
//YUI 3
Y.Array.each(items, function(value, index, array){
  process(value);
});
//jQuery
jQuery.each(items, function(index, value){
  process(value);
});
//Dojo
dojo.forEach(items, function(value, index, array){
  process(value);
});
//Prototype
items.each(function(value, index){
  process(value);
});
//MooTools
$each(items, function(value, index){
  process(value);
});
```

Even though function-based iteration represents a more convenient method of iteration, it is also quite a bit slower than loop-based iteration. The slowdown can be accounted for by the overhead associated with an extra method being called on each array item. In all cases, function-based iteration takes up to eight times as long as loop-based iteration and therefore isn't a suitable approach when execution time is a significant concern.

尽管基于函数的迭代显得更加便利，它还是比基于循环的迭代要慢一些。每个数组项要关联额外的函数调用是造成速度慢的原因。在所有情况下，基于函数的迭代占用时间是基于循环的迭代的八倍，因此在关注执行时间的情况下它并不是一个合适的办法。

## Conditionals　条件表达式

Similar in nature to loops, conditionals determine how execution flows through JavaScript. The traditional argument of whether to use **if-else** statements or a **switch** statement applies to JavaScript just as it does to other languages. Since different browsers have implemented different flow control optimizations, it is not always clear which technique to use.

与循环相似，条件表达式决定 JavaScript 运行流的走向。其他语言使用 if-else 或者 switch 表达式的传统观点也适用于 JavaScript。由于不同的浏览器针对流程控制进行了不同的优化，使用哪种技术并不总是很清楚。

### if-else Versus switch　if-else 与 switch 比较

The prevailing theory on using **if-else** versus **switch** is based on the number of conditions being tested: the larger the number of conditions, the more inclined you are to use a **switch** instead of **if-else**. This typically comes down to which code is easier to read. The argument is that **if-else** is easier to read when there are fewer conditions and **switch** is easier to read when the number of conditions is large. Consider the following:

使用 if-else 或者 switch 的流行理论是基于测试条件的数量：条件数量较大，倾向于使用 switch 而不是 if-else。这通常归结到代码的易读性。这种观点认为，如果条件较少时，if-else 容易阅读，而条件较多时 switch 更容易阅读。考虑下面几点：

```
 if (found){
   //do something
 } else {
   //do something else
 }
switch(found){
   case true:
     //do something
     break;
   default:
    //do something else
}
```

Though both pieces of code perform the same task, many would argue that the **if-else** statement is much easier to read than the **switch**. Increasing the number of conditions, however, usually reverses that opinion:

虽然两个代码块实现同样任务，很多人会认为 if-else 表达式比 witch 表达式更容易阅读。如果增加条件体的数量，通常会扭转这种观点：

```
 if (color == "red"){
   //do something
} else if (color == "blue"){
   //do something
} else if (color == "brown"){
   //do something
} else if (color == "black"){
   //do something
} else {
   //do something
}
switch (color){
   case "red":
     //do something
     break;
   case "blue":
     //do something
     break;
   case "brown":
     //do something
     break;
   case "black":
     //do something
     break;
   default:
     //do something
}
```

Most would consider the **switch** statement in this code to be more readable than the **if-else** statement.

大多数人会认为这段代码中的 switch 表达式比 if-else 表达式可读性更好。

As it turns out, the **switch** statement is faster in most cases when compared to **if-else**, but significantly faster only when the number of conditions is large. The primary difference in performance between the two is that the incremental cost of an additional condition is larger for **if-else** than it is for **switch**. Therefore, our natural inclination to use **if-else** for a small number of conditions and a **switch** statement for a larger number of conditions is exactly the right advice when considering performance.

事实证明，大多数情况下 switch 表达式比 if-else 更快，但只有当条件体数量很大时才明显更快。两者间的主要性能区别在于：当条件体增加时，if-else 性能负担增加的程度比 switch 更多。因此，我们的自然倾向认为条件体较少时应使用 if-else 而条件体较多时应使用 switch 表达式，如果从性能方面考虑也是正确的。

Generally speaking, **if-else** is best used when there are two discrete values or a few different ranges of values for which to test. When there are more than two discrete values for which to test, the **switch** statement is the most optimal choice.

一般来说，if-else 适用于判断两个离散的值或者判断几个不同的值域。如果判断多于两个离散值，switch 表达式将是更理想的选择。

## Optimizing if-else 优化 if-else

When optimizing **if-else**, the goal is always to minimize the number of conditions to evaluate before taking the correct path. The easiest optimization is therefore to ensure that the most common conditions are first. Consider the following:

优化 if-else 的目标总是最小化找到正确分支之前所判断条件体的数量。最简单的优化方法是将最常见的条件体放在首位。考虑下面的例子：

```
 if (value < 5) {
   //do something
} else if (value > 5 && value < 10) {
   //do something
} else {
   //do something
}
```

This code is optimal only if **value** is most frequently less than 5. If **value** is typically greater than or equal to 10, then two conditions must be evaluated each time before the correct path is taken, ultimately increasing the average amount of time spent in this statement. Conditions in an **if-else** should always be ordered from most likely to least likely to ensure the fastest possible execution time.

这段代码只有当 value 值经常小于 5 时才是最优的。如果 value 经常大于等于 10，那么在进入正确分支之前，必须两次运算条件体，导致表达式的平均时间提高。if-else 中的条件体应当总是按照从最大概率到最小概率的顺序排列，以保证理论运行速度最快。

Another approach to minimizing condition evaluations is to organize the **if-else** into a series of nested **if-else** statements. Using a single, large **if-else** typically leads to slower overall execution time as each additional condition is evaluated. For example:

另外一种减少条件判断数量的方法是将 if-else 组织成一系列嵌套的 if-else 表达式。使用一个单独的一长串的 if-else 通常导致运行缓慢，因为每个条件体都要被计算。例如：

```
 if (value == 0){
   return result0;
} else if (value == 1){
   return result1;
} else if (value == 2){
  return result2;
} else if (value == 3){
  return result3;
} else if (value == 4){
  return result4;
} else if (value == 5){
  return result5;
} else if (value == 6){
  return result6;
} else if (value == 7){
  return result7;
} else if (value == 8){
  return result8;
} else if (value == 9){
  return result9;
} else {
  return result10;
}
```

With this **if-else** statement, the maximum number of conditions to evaluate is 10. This slows down the average execution time if you assume that the possible values for **value** are evenly distributed between 0 and 10. To minimize the number of conditions to evaluate, the code can be rewritten into a series of nested **if-else** statements, such as:

在这个 if-else 表达式中，所计算条件体的最大数目是 10。如果假设 value 的值在 0 到 10 之间均匀分布，那么会增加平均运行时间。为了减少条件判断的数量，此代码可重写为一系列嵌套的 if-else 表达式，例如：

```
if (value < 6){
  if (value < 3){
    if (value == 0){
      return result0;
    } else if (value == 1){
      return result1;
    } else {
      return result2;
    }
  } else {
    if (value == 3){
      return result3;
    } else if (value == 4){
      return result4;
    } else {
      return result5;
    }
  }
} else {
  if (value < 8){
    if (value == 6){
      return result6;
    } else {
      return result7;
    }
  } else {
    if (value == 8){
      return result8;
    } else if (value == 9){
      return result9;
    } else {
      return result10;
    }
  }
}
```

The rewritten **if-else** statement has a maximum number of four condition evaluations each time through. This is achieved by applying a binary-search-like approach, splitting the possible values into a series of ranges to check and then drilling down further in that section. The average amount of time it takes to execute this code is roughly half of the time it takes to execute the previous if-else statement when the values are evenly distributed between 0 and 10. This approach is best when there are ranges of values for which to test (as opposed to discrete values, in which case a **switch** statement is typically more appropriate).

在重写的 if-else 表达式中，每次抵达正确分支时最多通过四个条件判断。它使用二分搜索法将值域分成了一系列区间，然后逐步缩小范围。当数值范围分布在 0 到 10 时，此代码的平均运行时间大约是前面那个版本的一半。此方法适用于需要测试大量数值的情况（相对离散值来说 switch 更合适）。

**Lookup Tables** 查表法

Sometimes the best approach to conditionals is to avoid using **if-else** and **switch** altogether. When there are a large number of discrete values for which to test, both **if-else** and **switch** are significantly slower than using a lookup table. Lookup tables can be created using arrays or regular objects in JavaScript, and accessing data from a lookup table is much faster than using **if-else** or **switch**, especially when the number of conditions is large (see Figure 4-1).

有些情况下要避免使用 if-else 或 switch。当有大量离散值需要测试时，if-else 和 switch 都比使用查表法要慢得多。在 JavaScript 中查表法可使用数组或者普通对象实现，查表法访问数据比 if-else 或者 switch 更快，特别当条件体的数目很大时（如图 4-1）。

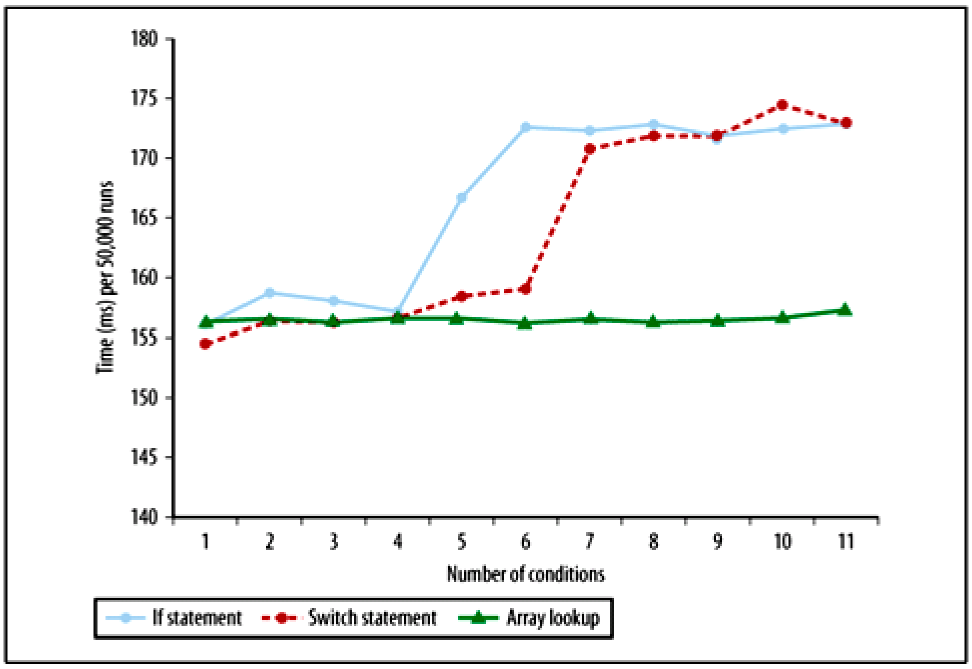


Figure 4-1. Array item lookup versus using if-else or switch in Internet Explorer 7

图 4-1　Internet Explorer 7 中数组查询与 if-else 或 switch 的比较

Lookup tables are not only very fast in comparison to **if-else** and **switch**, but they also help to make code more readable when there are a large number of discrete values for which to test. For example, **switch** statements start to get unwieldy when large, such as:

与 if-else 和 switch 相比，查表法不仅非常快，而且当需要测试的离散值数量非常大时，也有助于保持代码的可读性。例如，当 switch 表达式很大时就变得很笨重，诸如：

```
switch(value){
  case 0:
    return result0;
  case 1:
    return result1;
  case 2:
    return result2;
  case 3:
    return result3;
  case 4:
    return result4;
  case 5:
    return result5;
  case 6:
    return result6;
  case 7:
    return result7;
  case 8:
    return result8;
  case 9:
    return result9;
  default:
    return result10;
}
```

The amount of space that this **switch** statement occupies in code is probably not proportional to its importance. The entire structure can be replaced by using an array as a lookup table:

switch 表达式代码所占的空间可能与它的重要性不成比例。整个结构可以用一个数组查表替代：

```
 //define the array of results
var results = [result0, result1, result2, result3, result4, result5, result6, result7, result8, result9, result10]
//return the correct result
return results[value];
```

When using a lookup table, you have completely eliminated all condition evaluations. The operation becomes either an array item lookup or an object member lookup. This is a major advantage for lookup tables: since there are no conditions to evaluate, there is little or no additional overhead as the number of possible values increases.

当使用查表法时，必须完全消除所有条件判断。操作转换成一个数组项查询或者一个对象成员查询。使用查表法的一个主要优点是：由于没有条件判断，当候选值数量增加时，很少，甚至没有增加额外的性能开销。

Lookup tables are most useful when there is logical mapping between a single key and a single value (as in the previous example). A **switch** statement is more appropriate when each key requires a unique action or set of actions to take place.

查表法最常用于一个键和一个值形成逻辑映射的领域（如前面的例子）。一个 switch 表达式更适合于每个键需要一个独特的动作，或者一系列动作的场合。

## Recursion 递归

Complex algorithms are typically made easier by using recursion. In fact, there are some traditional algorithms that presume recursion as the implementation, such as a function to return factorials:

复杂算法通常比较容易使用递归实现。事实上，有些传统算法正是以递归实现的，诸如阶乘函数：

```
function factorial(n){
  if (n == 0){
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

The problem with recursive functions is that an ill-defined or missing terminal condition can lead to long execution times that freeze the user interface. Further, recursive functions are more likely to run into browser call stack size limits.

递归函数的问题是，一个错误定义，或者缺少终结条件可导致长时间运行，冻结用户界面。此外，递归函数还会遇到浏览器调用栈大小的限制。

## Call Stack Limits 调用栈限制

The amount of recursion supported by JavaScript engines varies and is directly related to the size of the JavaScript call stack. With the exception of Internet Explorer, for which the call stack is related to available system memory, all other browsers have static call stack limits. The call stack size for the most recent browser versions is relatively high compared to older browsers (Safari 2, for instance, had a call stack size of 100). Figure 4-2 shows call stack sizes over the major browsers.

JavaScript 引擎所支持的递归数量与 JavaScript 调用栈大小直接相关。只有 Internet Explorer 例外，它的调用栈与可用系统内存相关，其他浏览器有固定的调用栈限制。大多数现代浏览器的调用栈尺寸比老式浏览器要大（例如 Safari 2 调用栈尺寸是 100）。图 4-2 显示出主流浏览器的调用栈大小。

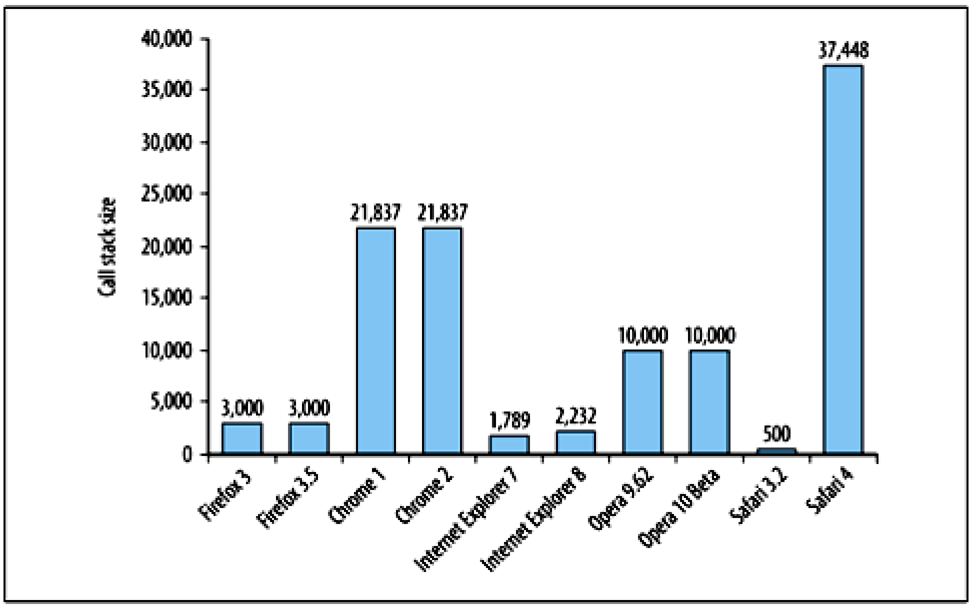


Figure 4-2. JavaScript call stack size in browsers

图 4-2 浏览器的 JavaScript 调用栈尺寸

When you exceed the maximum call stack size by introducing too much recursion, the browser will error out with one of the following messages:

当你使用了太多的递归，超过最大调用栈尺寸时，浏览器会出错并弹出以下信息：

- Internet Explorer: “Stack overflow at line x”

• Firefox: “Too much recursion”

• Safari: “Maximum call stack size exceeded”

• Opera: “Abort (control stack overflow)”

Chrome is the only browser that doesn't display a message to the user when the call stack size has been exceeded.

Chrome 是唯一不显示调用栈溢出错误的浏览器。

Perhaps the most interesting part of stack overflow errors is that they are actual JavaScript errors in some browsers, and can therefore be trapped using a try-catch statement. The exception type varies based on the browser being used. In Firefox, it's an **InternalError**; in Safari and Chrome, it's a **RangeError**; and Internet Explorer throws a generic **Error** type. (Opera doesn't throw an error; it just stops the JavaScript engine.) This makes it possible to handle such errors right from JavaScript:

关于调用栈溢出错误，最令人感兴趣的部分大概是：在某些浏览器中，他们的确是 JavaScript 错误，可以用一个 try-catch 表达式捕获。异常类型因浏览器而不同。在 Firefox 中，它是一个 InternalError；在 Safari 和 Chrome 中，它是一个 RangeError；在 Internet Explorer 中抛出一个一般性的 Error 类型。（Opera 不抛出错误；它终止 JavaScript 引擎）。这使得我们能够在 JavaScript 中正确处理这些错误：

```
try {
  recurse();
} catch (ex){
  alert("Too much recursion!");
}
```

If left unhandled, these errors bubble up as any other error would (in Firefox, it ends up in the Firebug and error consoles; in Safari/Chrome it shows up in the JavaScript console), except in Internet Explorer. IE will not only display a JavaScript error, but will also display a dialog box that looks just like an alert with the stack overflow message.

如果不管它，那么这些错误将像其他错误一样冒泡上传（在 Firefox 中，它结束于 Firebug 和错误终端；在 Safari/Chrome 中它显示在 JavaScript 终端上），只有 Internet Explorer 例外。IE 不会显示一个 JavaScript 错误，但是会弹出一个提示堆栈溢出信息的对话框。

## Recursion Patterns　递归模式

When you run into a call stack size limit, your first step should be to identify any instances of recursion in the code. To that end, there are two recursive patterns to be aware of. The first is the straightforward recursive pattern represented in the factorial() function shown earlier, when a function calls itself. The general pattern is as follows:

当你陷入调用栈尺寸限制时，第一步应该定位在代码中的递归实例上。为此，有两个递归模式值得注意。首先是直接递归模式为代表的前面提到的 factorial()函数，即一个函数调用自身。其一般模式如下：

```
 function recurse(){
   recurse();
}
recurse();
```

This pattern is typically easy to identify when errors occur. A second, subtler pattern involves two functions:

当发生错误时，这种模式比较容易定位。另外一种模式称为精巧模式，它包含两个函数：

```
 function first(){
   second();
}
function second(){
   first();
}
first();
```

In this recursion pattern, two functions each call the other, such that an infinite loop is formed. This is the more troubling pattern and a far more difficult one to identify in large code bases.

在这种递归模式中，两个函数互相调用对方，形成一个无限循环。这是一个令人不安的模式，在大型代码库中定位错误很困难。

Most call stack errors are related to one of these two recursion patterns. A frequent cause of stack overflow is an incorrect terminal condition, so the first step after identifying the pattern is to validate the terminal condition. If the terminal condition is correct, then the algorithm contains too much recursion to safely be run in the browser and should be changed to use iteration, memoization, or both.

大多数调用栈错误与这两种模式之一有关。常见的栈溢出原因是一个不正确的终止条件，所以定位模式错误的第一步是验证终止条件。如果终止条件是正确的，那么算法包含了太多层递归，为了能够安全地在浏览器中运行，应当改用迭代，制表，或两者兼而有之。

译者注：memoization，没错，就是这么个单词，译为“制表”，不是 memorization！

memoization，又称 tabulation，源自键盘上的 tab 键。

后面章节还会详解，简而言之，就是用一个数组栈记录每次递归的结果，如果某值曾经计算过，那么直接从数组栈中以查表法获得结果，而不必重复计算。

**Iteration 迭代**

Any algorithm that can be implemented using recursion can also be implemented using iteration. Iterative algorithms typically consist of several different loops performing different aspects of the process, and thus introduce their own performance issues. However, using optimized loops in place of long-running recursive functions can result in performance improvements due to the lower overhead of loops versus that of executing a function.

任何可以用递归实现的算法都可以用迭代实现。迭代算法通常包括几个不同的循环，分别对应算法过程的不同方面，也会导致自己的性能为题。但是，使用优化的循环替代长时间运行的递归函数可以提高性能，因为运行一个循环比反复调用一个函数的开销要低。

As an example, the merge sort algorithm is most frequently implemented using recursion. A simple JavaScript implementation of merge sort is as follows:

例如，合并排序算法是最常用的以递归实现的算法。一个简单的 JavaScript 实现的合并排序算法如下：

```
function merge(left, right){
  var result = [];
  while (left.length > 0 && right.length > 0){
    if (left[0] < right[0]){
      result.push(left.shift());
    } else {
      result.push(right.shift());
    }
  }
  return result.concat(left).concat(right);
}
function mergeSort(items){
  if (items.length == 1) {
    return items;
  }
  var middle = Math.floor(items.length / 2),
  left = items.slice(0, middle),
  right = items.slice(middle);
  return merge(mergeSort(left), mergeSort(right));
}
```

The code for this merge sort is fairly simple and straightforward, but the **mergeSort**() function itself ends up getting called very frequently. An array of **n** items ends up calling **mergeSort**() 2 * n –1 times, meaning that an array with more than 1,500 items would cause a stack overflow error in Firefox.

这个合并排序代码相当简单直接，但是 mergeSort()函数被调用非常频繁。一个具有 n 个项的数组总共调用 mergeSort()达 2 * n - 1 次，也就是说，对一个超过 1500 个项的数组操作，就可能在 Firefox 上导致栈溢出。

Running into the stack overflow error doesn't necessarily mean the entire algorithm has to change; it simply means that recursion isn't the best implementation. The merge sort algorithm can also be implemented using iteration, such as:

程序陷入栈溢出错误并不一定要修改整个算法；它只是意味着递归不是最好的实现方法。合并排序算法还可以用迭代实现，如下：

```
//uses the same mergeSort() function from previous example
function mergeSort(items){
  if (items.length == 1) {
    return items;
  }
  var work = [];
  for (var i=0, len=items.length; i < len; i++){
    work.push([items[i]]);
  }
  work.push([]); //in case of odd number of items
  for (var lim=len; lim > 1; lim = (lim+1)/2){
    for (var j=0,k=0; k < lim; j++, k+=2){
      work[j] = merge(work[k], work[k+1]);
    }
    work[j] = []; //in case of odd number of items
  }
  return work[0];
}
```

This implementation of **mergeSort**() does the same work as the previous one without using recursion. Although the iterative version of merge sort may be somewhat slower than the recursive option, it doesn't have the same call stack impact as the recursive version. Switching recursive algorithms to iterative ones is just one of the options for avoiding stack overflow errors.

此 mergeSort()实现与前面的函数实现同样功能而没有使用递归。虽然迭代版本的合并排序可能比递归版本的慢一些，但它不会像递归版本那样影响调用栈。将递归算法切换为迭代只是避免栈溢出错误的方法之一。

**Memoization** 制表

Work avoidance is the best performance optimization technique. The less work your code has to do, the faster it executes. Along those lines, it also makes sense to avoid work repetition. Performing the same task multiple times is a waste of execution time. Memoization is an approach to avoid work repetition by caching previous calculations for later reuse, which makes memoization a useful technique for recursive algorithms.

减少工作量就是最好的性能优化技术。代码所做的事情越少，它的运行速度就越快。根据这些原则，避免重复工作也很有意义。多次执行相同的任务也在浪费时间。制表，通过缓存先前计算结果为后续计算所重复使用，避免了重复工作。这使得制表成为递归算法中有用的技术。

When recursive functions are called multiple times during code execution, there tends to be a lot of work duplication. The **factorial**() function, introduced earlier in "Recursion" on page 73, is a great example of how work can be repeated multiple times by recursive functions. Consider the following code:

当递归函数多次被调用时，重复工作很多。在 factorial()函数中（在前面介绍过的阶乘函数），是一个递归函数重复多次的典型例子。考虑下面的代码：

```
var fact6 = factorial(6);
var fact5 = factorial(5);
var fact4 = factorial(4);
```

This code produces three factorials and results in the **factorial**() function being called a total of 18 times. The worst part of this code is that all of the necessary work is completed on the first line. Since the factorial of 6 is equal to 6 multiplied by the factorial 5, the factorial of 5 is being calculated twice. Even worse, the factorial of 4 is being calculated three times. It makes far more sense to save those calculations and reuse them instead of starting over anew with each function call.

此代码生成三个阶乘结果，factorial()函数总共被调用了 18 次。此代码中最糟糕的部分是，所有必要的计算已经在第一行代码中执行过了。因为 6 的阶乘等于 6 乘以 5 的阶乘，所以 5 的阶乘被计算了两次。更糟糕的是，4 的阶乘被计算了三次。更为明智的方法是保存并重利用它们的计算结果，而不是每次都重新计算整个函数。

You can rewrite the **factorial**() function to make use of memoization in the following way:

你可以使用制表技术来重写 factorial()函数，如下：

```
function memfactorial(n){
  if (!memfactorial.cache){
    memfactorial.cache = {
      "0": 1,
      "1": 1
    };
  }
  if (!memfactorial.cache.hasOwnProperty(n)){
    memfactorial.cache[n] = n * memfactorial (n-1);
  }
  return memfactorial.cache[n];
}
```

The key to this memoized version of the factorial function is the creation of a cache object. This object is stored on the function itself and is prepopulated with the two simplest factorials: 0 and 1. Before calculating a factorial, this cache is checked to see whether the calculation has already been performed. No cache value means the calculation must be done for the first time and the result stored in the cache for later usage. This function is used in the same manner as the original **factorial**() function:

这个使用制表技术的阶乘函数的关键是建立一个缓存对象。此对象位于函数内部，并预置了两个最简单的阶乘：0 和 1。在计算阶乘之前，首先检查缓存中是否已经存在相应的计算结果。没有对应的缓冲值说明这是第一次进行此数值的计算，计算完成之后结果被存入缓存之中，以备今后使用。此函数与原始版本的 factorial()函数用法相同。

```
 var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

This code returns three different factorials but makes a total of eight calls to **memfactorial**(). Since all of the necessary calculations are completed on the first line, the next two lines need not perform any recursion because cached values are returned.

此代码返回三个不同的阶乘值，但总共只调用 memfactorial()函数八次。既然所有必要的计算都在第一行代码中完成了，那么后两行代码不会产生递归运算，因为直接返回缓存中的数值。

The memoization process may be slightly different for each recursive function, but generally the same pattern applies. To make memoizing a function easier, you can define a **memoize**() function that encapsulates the basic functionality. For example:

制表过程因每种递归函数而略有不同，但总体上具有相同的模式。为了使一个函数的制表过程更加容易，你可以定义一个 memoize()函数封装基本功能。例如：

```
 function memoize(fundamental, cache){
   cache = cache || {};
   var shell = function(arg){
     if (!cache.hasOwnProperty(arg)){
       cache[arg] = fundamental(arg);
     }
     return cache[arg];
   };
   return shell;
}
```

This **memoize**() function accepts two arguments: a function to memoize and an optional cache object. The cache object can be passed in if you'd like to prefill some values; otherwise a new cache object is created. A shell function is then created that wraps the original (**fundamental**) and ensures that a new result is calculated only if it has never previously been calculated. This shell function is returned so that you can call it directly, such as:

此 memoize()函数接收两个参数：一个用来制表的函数和一个可选的缓存对象。如果你打算预设一些值，那么就传入一个预定义的缓存对象；否则它将创建一个新的缓存对象。然后创建一个外壳函数，将原始函数（fundamential）包装起来，确保只有当一个此前从未被计算过的值传入时才真正进行计算。计算结果由此外壳函数返回，你可以直接调用它，例如：

```
//memoize the factorial function
var memfactorial = memoize(factorial, { "0": 1, "1": 1 });
//call the new function
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

Generic memoization of this type is less optimal that manually updating the algorithm for a given function because the **memoize**() function caches the result of a function call with specific arguments. Recursive calls, therefore, are saved only when the shell function is called multiple times with the same arguments. For this reason, it's better to manually implement memoization in those functions that have significant performance issues rather than apply a generic memoization solution.

这种通用制表函数与人工更新算法相比优化较少，因为 memoize()函数缓存特定参数的函数调用结果。当代码以同一个参数多次调用外壳函数时才能节约时间（译者注：如果外壳函数内部还存在递归，那么内部的递归就不能享用这些中间运算结果了）。因此，当一个通用制表函数存在显著性能问题时，最好在这些函数中人工实现制表法。

## Summary 总结

Just as with other programming languages, the way that you factor your code and the algorithm you choose affects the execution time of JavaScript. Unlike other programming languages, JavaScript has a restricted set of resources from which to draw, so optimization techniques are even more important.

正如其他编程语言，代码的写法和算法选用影响 JavaScript 的运行时间。与其他编程语言不同的是，JavaScript 可用资源有限，所以优化技术更为重要。

• The for, while, and do-while loops all have similar performance characteristics, and so no one loop type is significantly faster or slower than the others.

for，while，do-while 循环的性能特性相似，谁也不比谁更快或更慢。

• Avoid the for-in loop unless you need to iterate over a number of unknown object properties.

除非你要迭代遍历一个属性未知的对象，否则不要使用 for-in 循环。

• The best ways to improve loop performance are to decrease the amount of work done per iteration and decrease the number of loop iterations.

改善循环性能的最好办法是减少每次迭代中的运算量，并减少循环迭代次数。

• Generally speaking, switch is always faster than if-else, but isn't always the best solution.

一般来说，switch 总是比 if-else 更快，但并不总是最好的解决方法。

• Lookup tables are a faster alternative to multiple condition evaluation using if-else or switch.

当判断条件较多时，查表法比 if-else 或者 switch 更快。

• Browser call stack size limits the amount of recursion that JavaScript is allowed to perform; stack overflow errors prevent the rest of the code from executing.

浏览器的调用栈尺寸限制了递归算法在 JavaScript 中的应用；栈溢出错误导致其他代码也不能正常执行。

• If you run into a stack overflow error, change the method to an iterative algorithm or make use of memoization to avoid work repetition.

如果你遇到一个栈溢出错误，将方法修改为一个迭代算法或者使用制表法可以避免重复工作。

The larger the amount of code being executed, the larger the performance gain realized from using these strategies.

运行的代码总量越大，使用这些策略所带来的性能提升就越明显。

# 第五章 Strings and Regular Expressions 字符串和正则表达式

Practically all JavaScript programs are intimately tied to strings. For example, many applications use Ajax to fetch strings from a server, convert those strings into more easily usable JavaScript objects, and then generate strings of HTML from the data. A typical program deals with numerous tasks like these that require you to merge, split, rearrange, search, iterate over, and otherwise handle strings; and as web applications become more complex, progressively more of this processing is done in the browser.

几乎所有 JavaScript 程序都与字符串操作紧密相连。例如，许多应用程序使用 Ajax 从服务器获取字符串，将这些字符串转换成更易用的 JavaScript 对象，然后从数据中生成 HTML 字符串。一个典型的程序需要处理若干这样的任务，合并，分解，重新排列，搜索，遍历，以及其他方法处理字符串。随着网页应用越来越复杂，越来越多的此类任务将在浏览器中完成。

In JavaScript, regular expressions are essential for anything more than trivial string processing. A lot of this chapter is therefore dedicated to helping you understand how regular expression engines internally process your strings and teaching you how to write regular expressions that take advantage of this knowledge.

在 JavaScript 中，正则表达式是必不可少的东西，它的重要性远超过琐碎的字符串处理。本章使用相当篇幅帮助您了解正则表达式引擎处理字符串的原理，并讲授如何利用这些知识书写正则表达式。

Also in this chapter, you'll learn about the fastest cross-browser methods for concatenating and trimming strings, discover how to increase regex performance by reducing backtracking, and pick up plenty of other tips and tricks for efficiently processing strings and regular expressions.

通过本章内容，您还将学到关于连接、修整字符串的最快的跨浏览器方法，探索如何通过减少回溯来提高正则表达式的性能，并挑选了一些关于高效处理字符串和正则表达式的技巧。

## String Concatenation 字符串连接

String concatenation can be surprisingly performance intensive. It's a common task to build a string by continually adding to the end of it in a loop (e.g., when building up an HTML table or an XML document), but this sort of processing is notorious for its poor performance in some browsers.

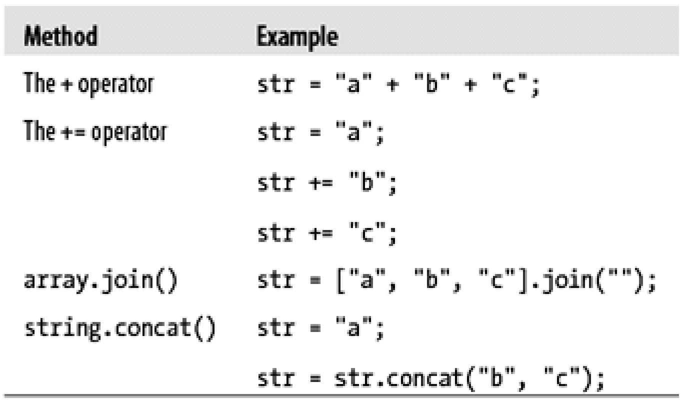
字符串连接表现出惊人的性能紧张。通常一个任务通过一个循环，向字符串末尾不断地添加内容，来创建一个字符串（例如，创建一个 HTML 表或者一个 XML 文档），但此类处理在一些浏览器上表现糟糕而遭人痛恨。

So how can you optimize these kinds of tasks? For starters, there is more than one way to merge strings (see Table 5-1).

那么你怎样优化此类任务呢？首先，有多种方法可以合并字符串（见表 5-1）。

Table 5-1. String concatenation methods

表 5-1　字符串连接函数



| Method | Example |
|---|---|
| The + operator | `str = "a" + "b" + "c";` |
| The += operator | `str = "a";` |
| | `str += "b";` |
| | `str += "c";` |
| `array.join()` | `str = ["a", "b", "c"].join("");` |
| `string.concat()` | `str = "a";` |
| | `str = str.concat("b", "c");` |

All of these methods are fast when concatenating a few strings here and there, so for casual use, you should go with whatever is the most practical. As the length and number of strings that must be merged increases, however, some methods start to show their strength.

当连接少量字符串时，所有这些函数都很快，临时使用的话，可选择最熟悉的使用。当合并字符串的长度和数量增加之后，有些函数开始显示出自己的威力。

**Plus (+) and Plus-Equals (+=) Operators　加和加等于操作**

These operators provide the simplest method for concatenating strings and, in fact, all modern browsers except IE7 and earlier optimize them well enough that you don't really need to look at other options. However, several techniques maximize the efficiency of these operators.

这些操作符提供了连接字符串的最简单方法，事实上，除 IE7 和它之前的所有现代浏览器都对此优化得很好，所以你不需要寻找其他方法。然而，有些技术可以最大限度地提高这些操作的效率。

First, an example. Here's a common way to assign a concatenated string:

首先，看一个例子。这是连接字符串的常用方法：

```
str += "one" + "two";
```

When evaluating this code, four steps are taken:

此代码执行时，发生四个步骤：

1. A temporary string is created in memory.

内存中创建了一个临时字符串。

2. The concatenated value "**onetwo**" is assigned to the temporary string.

临时字符串的值被赋予"onetwo"。

3. The temporary string is concatenated with the current value of **str**.

临时字符串与 str 的值进行连接。

4. The result is assigned to **str**.

结果赋予 str。

This is actually an approximation of how browsers implement this task, but it's close.

这基本上就是浏览器完成这一任务的过程。

The following code avoids the temporary string (steps 1 and 2 in the list) by directly appending to **str** using two discrete statements. This ends up running about 10%–40% faster in most browsers:

下面的代码通过两个离散表达式直接将内容附加在 str 上避免了临时字符串（上面列表中第 1 步和第 2 步）。在大多数浏览器上这样做可加快 10%-40%：

```
str += "one";
str += "two";
```

In fact, you can get the same performance improvement using one statement, as follows:

实际上，你可以用一行代码就实现这样的性能提升，如下：

```
str = str + "one" + "two";
// equivalent to str = ((str + "one") + "two")
```

This avoids the temporary string because the assignment expression starts with **str** as the base and appends one string to it at a time, with each intermediary concatenation performed from left to right. If the concatenation were performed in a different order (e.g., **str = "one" + str + "two"**), you would lose this optimization. This is because of the way that browsers allocate memory when merging strings. Apart from IE, browsers try to expand the memory allocation for the string on the left of an expression and simply copy the second string to the end of it (see Figure 5-1). If, in a loop, the base string is furthest to the left, you avoid repeatedly copying a progressively larger base string.

这就避免了使用临时字符串，因为赋值表达式开头以 str 为基础，一次追加一个字符串，从左至右依次连接。如果改变连接顺序（例如，str = "one" + str + "two"），你会失去这种优化。这与浏览器合并字符串时分配内存的方法有关。除 IE 以外，浏览器尝试扩展表达式左端字符串的内存，然后简单地将第二个字符串拷贝到它的尾部（如图 5-1）。如果在一个循环中，基本字符串位于最左端，就可以避免多次复制一个越来越大的基本字符串。

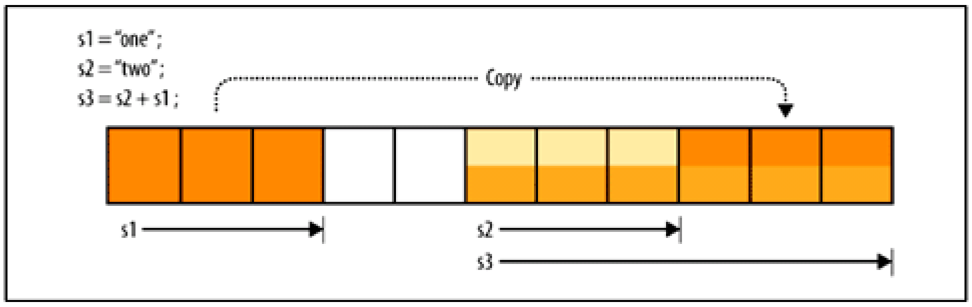


Figure 5-1. Example of memory use when concatenating strings: s1 is copied to the end of s2 to create s3; the base string s2 is not copied

图 5-1　连接字符串时的内存使用情况：s1 复制到 s2 的尾部形成 s3；基本字符串 s2 没有被复制

These techniques don't apply to IE. They have little, if any, effect in IE8 and can actually make things slower in IE7 and earlier. That's because of how IE executes concatenation under the hood. In IE8's implementation, concatenating strings merely stores references to the existing string parts that compose the new string. At the last possible moment (when you actually use the concatenated string), the string parts are each copied into a new "real" string, which then replaces the previously stored string references so that this assembly doesn't have to be performed every time the string is used.

这些技术并不适用于 IE。它们几乎没有任何作用，在 IE8 上甚至比 IE7 和早期版本更慢。这与 IE 执行连接操作的机制有关。在 IE8 中，连接字符串只是记录下构成新字符串的各部分字符串的引用。在最后时刻（当你真正使用连接后的字符串时），各部分字符串才被逐个拷贝到一个新的“真正的”字符串中，然后用它取代先前的字符串引用，所以并非每次使用字符串时都发生合并操作。

IE7 and earlier use an inferior implementation of concatenation in which each pair of concatenated strings must always be copied to a new memory location. You'll see the potentially dramatic impact of this in the upcoming section "Array Joining". With the pre-IE8 implementation, the advice in this section can make things slower since it's faster to concatenate short strings before merging them with a larger base string (thereby avoiding the need to copy the larger string multiple times). For instance, with **largeStr = largeStr + s1 + s2**, IE7 and earlier must copy the large string twice, first to merge it with s1, then with s2. Conversely, **largeStr += s1 + s2** first merges the two smaller strings and then concatenates the result with the large string. Creating the intermediary string of **s1 + s2** is a much lighter performance hit than copying the large string twice.

IE7 和更早的浏览器在连接字符串时使用更糟糕的实现方法，每连接一对字符串都要把它们复制到一块新分配的内存中。你会在后面的“数组联结”一节中看到它潜在的巨大影响。针对 IE8 之前的实现方式，本节的建议反而会使代码更慢，因为合并多个短字符串比连接一个大字符串更快（避免多次拷贝那些大字符串）。例如，largeStr = largeStr + s1 + s2 语句，在 IE7 和更早的版本中，必须将这个大字符串拷贝两次，首先与 s1 合并，然后再与 s2 合并。相反，largeStr = s1 + s2 首先将两个小字符串合并起来，然后将结果返回给大字符串。创建中间字符串 s1 + s2 与两次拷贝大字符串相比，性能冲击要轻得多。

**Firefox and compile-time folding Firefox 和编译期合并**

When all strings concatenated in an assignment expression are compile-time constants, Firefox automatically merges them at compile time. Here's a way to see this in action:

在赋值表达式中所有字符串连接都属于编译期常量，Firefox 自动地在编译过程中合并它们。这里有一个方法可看到这一过程：

```
 function foldingDemo() {
   var str = "compile" + "time" + "folding";
   str += "this" + "works" + "too";
   str = str + "but" + "not" + "this";
}
alert(foldingDemo.toString());

// In Firefox, you'll see this:
// function foldingDemo() {
//    var str = "compiletimefolding";
//    str += "thisworkstoo";
//    str = str + "but" + "not" + "this";
// }
```

When strings are folded together like this, there are no intermediary strings at runtime and the time and memory that would be spent concatenating them is reduced to zero. This is great when it occurs, but it doesn't help very often because it's much more common to build strings from runtime data than from compile-time constants.

当字符串是这样合并在一起时，由于运行时没有中间字符串，所以连接它们的时间和内存可以减少到零。这种功能非常了不起，但它并不经常起作用，因为通常从运行期数据创建字符串而不是从编译期常量。

## Array Joining 数组联结

The **Array.prototype.join** method merges all elements of an array into a string and accepts a separator string to insert between each element. By passing in an empty string as the separator, you can perform a simple concatenation of all elements in an array.

Array.prototype.join 方法将数组的所有元素合并成一个字符串，并在每个元素之间插入一个分隔符字符串。如果传递一个空字符串作为分隔符，你可以简单地将数组的所有元素连接起来。

Array joining is slower than other methods of concatenation in most browsers, but this is more than compensated for by the fact that it is the only efficient way to concatenate lots of strings in IE7 and earlier.

在大多数浏览器上，数组联结比连接字符串的其他方法更慢，但是事实上，为一种补偿方法，在 IE7 和更早的浏览器上它是连接大量字符串唯一高效的途径。

The following example code demonstrates the kind of performance problem that array joining solves:

下面的示例代码演示了可用数组联结解决的性能问题：

```
var str = "I'm a thirty-five character string.",
newStr = "",
appends = 5000;
while (appends--) {
  newStr += str;
}
```

This code concatenates 5,000 35-character strings. Figure 5-2 shows how long it takes to complete this test in IE7, starting with 5,000 concatenations and then gradually increasing that number.

此代码连接 5'000 个长度为 35 的字符串。图 5-2 显示出在 IE7 中执行此测试所需的时间，从 5'000 次连接开始，然后逐步增加连接数量。

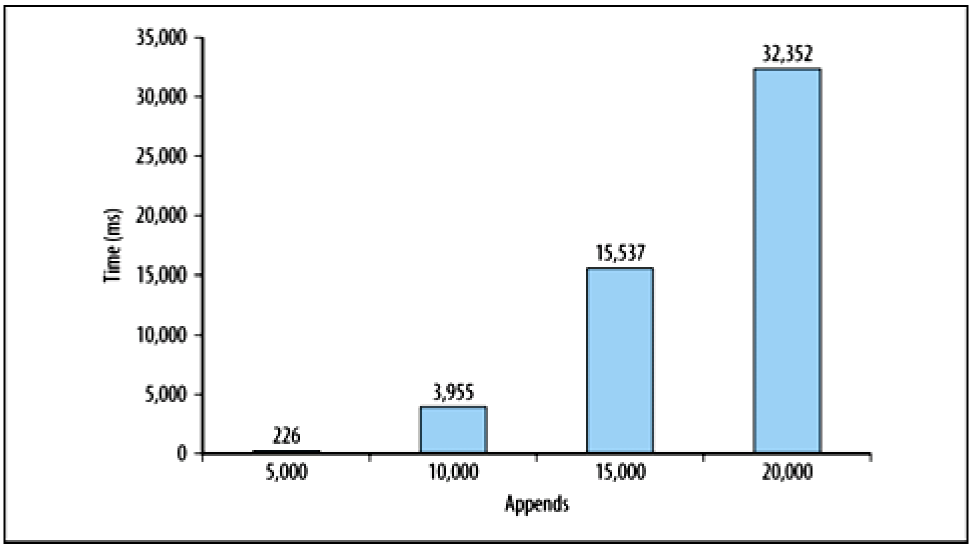


Figure 5-2. Time to concatenate strings using += in IE7

图 5-2 IE7 中使用+=连接字符串所用的时间

IE7's naive concatenation algorithm requires that the browser repeatedly copy and allocate memory for larger and larger strings each time through the loop. The result is quadratic running time and memory consumption.

IE7 天真的连接算法要求浏览器在循环过程中反复地为越来越大的字符串拷贝和分配内存。结果是以平方关系递增的运行时间和内存消耗。

The good news is that all other modern browsers (including IE8) perform far better in this test and do not exhibit the quadratic complexity that is the real killer here. However, this demonstrates the impact that seemingly simple string concatenation can have; 226 milliseconds for 5,000 concatenations is already a significant performance hit that would be nice to reduce as much as possible, but locking up a user's browser for more than 32 seconds in order to concatenate 20,000 short strings is unacceptable for nearly any application.

好消息是所有其他的现代浏览器（包括 IE8）在这个测试中表现良好，不会呈现平方关系的复杂性递增，这是真正的杀手级改善。然而，此程序演示了看似简单的字符串连接所产生的影响。5'000 次连接用去 226 毫秒已经是一个显著的性能冲击了，应当尽可能地缩减这一时间，但锁定用户浏览器长达 32 秒，只是为了连接 20'000 个短字符串，则对任何应用程序来说都是不能接受的。

Now consider the following test, which generates the same string via array joining:

现在考虑下面的测试，它使用数组联结生成同样的字符串：

```
 var str = "I'm a thirty-five character string.",
strs = [],
newStr,
appends = 5000;
while (appends--) {
  strs[strs.length] = str;
}
newStr = strs.join("");
```

Figure 5-3 shows this test's running time in IE7.

图 5-3 显示出 IE7 上进行此测试所用的时间。

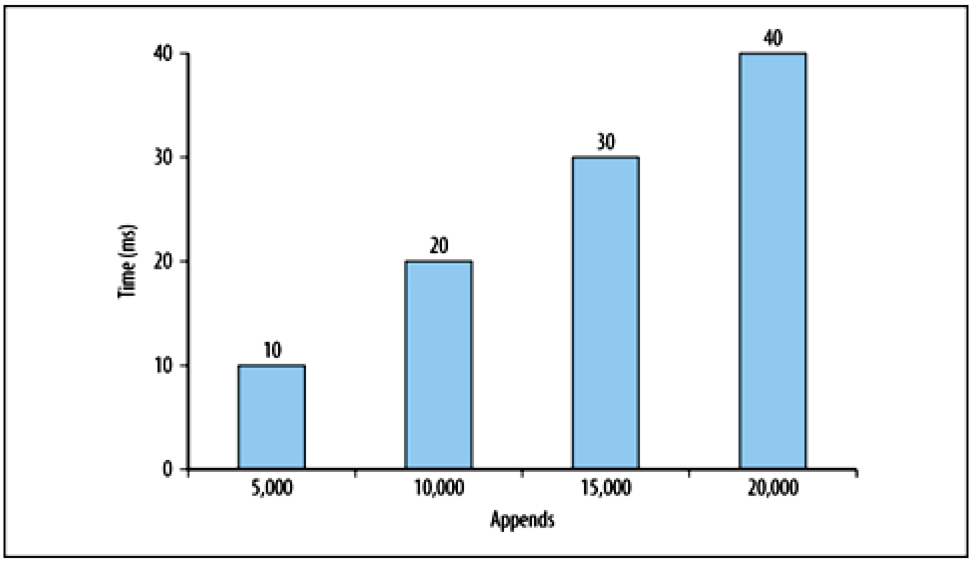


Figure 5-3. Time to concatenate strings using array joining in IE7

图 5-3 IE7 中使用数组连接来连接字符串所用的时间

This dramatic improvement results from avoiding repeatedly allocating memory for and copying progressively larger and larger strings. When joining an array, the browser allocates enough memory to hold the complete string, and never copies the same part of the final string more than once.

这一难以置信的改进结果是因为避免了重复的内存分配和拷贝越来越大的字符串。当联结一个数组时，浏览器分配足够大的内存用于存放整个字符串，也不会超过一次地拷贝最终字符串的同一部分。

**String.prototype.concat**

The native string **concat** method accepts any number of arguments and appends each to the string that the method is called on. This is the most flexible way to concatenate strings because you can use it to append just one string, a few strings at a time, or an entire array of strings.

原生字符串连接函数接受任意数目的参数，并将每一个参数都追加在调用函数的字符串上。这是连接字符串最灵活的方法，因为你可以用它追加一个字符串，或者一次追加几个字符串，或者一个完整的字符串数组。

```
// append one string
str = str.concat(s1);
// append three strings
str = str.concat(s1, s2, s3);
// append every string in an array by using the array
// as the list of arguments
str = String.prototype.concat.apply(str, array);
```

Unfortunately, **concat** is a little slower than simple + and += operators in most cases, and can be substantially slower in IE, Opera, and Chrome. Moreover, although using **concat** to merge all strings in an array appears similar to the array joining approach discussed previously, it's usually slower (except in Opera), and it suffers from the same potentially catastrophic performance problem as + and += when building large strings in IE7 and earlier.

不幸的是，大多数情况下 concat 比简单的+和+=慢一些，而且在 IE，Opera 和 Chrome 上大幅变慢。此外，虽然使用 concat 合并数组中的所有字符串看起来和前面讨论的数组联结差不多，但通常它更慢一些（Opera 除外），而且它还潜伏着灾难性的性能问题，正如 IE7 和更早版本中使用+和+=创建大字符串那样。

## Regular Expression Optimization 正则表达式优化

Incautiously crafted regexes can be a major performance bottleneck (the upcoming section, "Runaway Backtracking" on page 91, contains several examples showing how severe this can be), but there is a lot you can do to improve regex efficiency. Just because two regexes match the same text doesn't mean they do so at the same speed.

粗浅地编写正则表达式是造成性能瓶颈的主要原因（后面“回溯失控”一节有一些例子说明这是多么严重的问题），但还有很多可以改进正则表达式效率的地方。两个正则表达式匹配相同的文本并不意味着他们具有同等的速度。

Many factors affect a regex's efficiency. For starters, the text a regex is applied to makes a big difference because regexes spend more time on partial matches than obvious nonmatches. Each browser's regex engine also has different internal optimizations.

许多因素影响正则表达式的效率。首先，正则表达式适配的文本千差万别，部分匹配时比完全不匹配所用的时间要长。每种浏览器的正则表达式引擎也有不同的内部优化。

Regex optimization is a fairly broad and nuanced topic. There's only so much that can be covered in this section, but what's included should put you well on your way to understanding the kinds of issues that affect regex performance and mastering the art of crafting efficient regexes.

正则表达式的优化是一个相当广泛和细致入微的话题。本节讨论尽其所能，希望这些内容有助于您理解影响正则表达式性能的各种问题和掌握编写高效正则表达式的艺术。

Note that this section assumes you already have some experience with regular expressions and are primarily interested in how to make them faster. If you're new to regular expressions or need to brush up on the basics, numerous resources are available on the Web and in print. *Regular Expressions Cookbook* (O'Reilly) by Jan Goyvaerts and Steven Levithan (that's me!) is written for people who like to learn by doing, and covers JavaScript and several other programming languages equally.

请注意，本节假设您已经具有正则表达式经验，主要关注于如何使它们更快。如果您是正则表达式的新手，或者还需要复习一下基础，网上和书上都有许多资源。《Regular Expressions Cookbook》（O'Reilly）

由 Jan Goyvaerts 和 Steven Levithan（本文作者！）为那些勇于实践的人们编写，涵盖了 JavaScript 和其他编程语言。

**How Regular Expressions Work** 正则表达式工作原理

In order to use regular expressions efficiently, it's important to understand how they work their magic. The following is a quick rundown of the basic steps a regex goes through:

为了有效地使用正则表达式，重要的是理解它们的工作原理。下面是一个正则表达式处理的基本步骤：

Step 1: Compilation

第一步：编译

When you create a regex object (using a regex literal or the RegExp constructor), the browser checks your pattern for errors and then converts it into a native code routine that is used to actually perform matches. If you assign your regex to a variable, you can avoid performing this step more than once for a given pattern.

当你创建了一个正则表达式对象之后（使用一个正则表达式直接量或者 RegExp 构造器），浏览器检查你的模板有没有错误，然后将它转换成一个本机代码例程，用于执行匹配工作。如果你将正则表达式赋给一个变量，你可以避免重复执行此步骤。

Step 2: Setting the starting position

第二步：设置起始位置

When a regex is put to use, the first step is to determine the position within the target string where the search should start. This is initially the start of the string or the position specified by the regex's **lastIndex** property, but when returning here from step 4 (due to a failed match attempt), the position is one character after where the last attempt started.

当一个正则表达式投入使用时，首先要确定目标字符串中开始搜索的位置。它是字符串的起始位置，或者由正则表达式的 lastIndex 属性指定，但是当它从第四步返回到这里的时候（因为尝试匹配失败），此位置将位于最后一次尝试起始位置推后一个字符的位置上。

Optimizations that browser makers build into their regex engines can help avoid a lot of unnecessary work at this stage by deciding early that certain work can be skipped. For instance, if a regex starts with ^, IE and Chrome can usually determine that a match cannot be found after the start of a string and avoid foolishly searching subsequent positions. Another example is that if all possible matches contain **x** as the third character, a smart implementation may be able to determine this, quickly search for the next **x**, and set the starting position two characters back from where it's found (e.g., recent versions of Chrome include this optimization).

浏览器厂商优化正则表达式引擎的办法是，在这一阶段中通过早期预测跳过一些不必要的工作。例如，如果一个正则表达式以^开头，IE 和 Chrome 通常判断在字符串起始位置上是否能够匹配，然后可避免愚蠢地搜索后续位置。另一个例子是匹配第三个字母是 x 的字符串，一个聪明的办法是先找到 x，然后再将起始位置回溯两个字符（例如，最近的 Chrome 版本实现了这种优化）。

Step 3: Matching each regex token

第三步：匹配每个正则表达式的字元

Once the regex knows where to start, it steps through the text and the regex pattern. When a particular token fails to match, the regex tries to backtrack to a prior point in the match attempt and follow other possible paths through the regex.

正则表达式一旦找好起始位置，它将一个一个地扫描目标文本和正则表达式模板。当一个特定字元匹配失败时，正则表达式将试图回溯到扫描之前的位置上，然后进入正则表达式其他可能的路径上。

Step 4: Success or failure

第四步：匹配成功或失败

If a complete match is found at the current position in the string, the regex declares success. If all possible paths through the regex have been attempted but a match was not found, the regex engine goes back to step 2 to try again at the next character in the string. Only after this cycle completes for every character in the string (as well as the position after the last character) and no matches have been found does the regex declare overall failure.

如果在字符串的当前位置上发现一个完全匹配，那么正则表达式宣布成功。如果正则表达式的所有可能路径都尝试过了，但是没有成功地匹配，那么正则表达式引擎回到第二步，从字符串的下一个字符重新尝试。只有字符串中的每个字符（以及最后一个字符后面的位置）都经历了这样的过程之后，还没有成功匹配，那么正则表达式就宣布彻底失败。

Keeping this process in mind will help you make informed decisions about the types of issues that affect regex performance. Next up is a deeper look into a key feature of the matching process in step 3: backtracking.

牢记这一过程将有助于您明智地判别那些影响正则表达式性能问题的类型。接下来我们深入剖析第三步中匹配过程的关键点：回溯。

**Understanding Backtrack 理解回溯**

In most modern regex implementations (including those required by JavaScript), backtracking is a fundamental component of the matching process. It's also a big part of what makes regular expressions so expressive and powerful. However, backtracking is computationally expensive and can easily get out of hand if you're not careful. Although backtracking is only part of the overall performance equation, understanding how it works and how to minimize its use is perhaps the most important key to writing efficient regexes. The next few sections therefore cover the topic at some length.

在大多数现代正则表达式实现中（包括 JavaScript 所需的），回溯是匹配过程的基本组成部分。它很大程度上也是正则表达式如此美好和强大的根源。然而，回溯计算代价昂贵，如果你不够小心的话容易失控。虽然回溯是整体性能的唯一因素，理解它的工作原理，以及如何减少使用频率，可能是编写高效正则表达式最重要的关键点。因此后面几节用较长篇幅讨论这个话题。

As a regex works its way through a target string, it tests whether a match can be found at each position by stepping through the components in the regex from left to right. For each quantifier and alternation, a decision must be made about how to proceed. With a quantifier (such as *, +?, or {2,}), the regex must decide when to try matching additional characters, and with alternation (via the | operator), it must try one option from those available.

当一个正则表达式扫描目标字符串时，它从左到右逐个扫描正则表达式的组成部分，在每个位置上测试能不能找到一个匹配。对于每一个量词和分支，都必须决定如何继续进行。如果是一个量词（诸如*，+?，或者{2,}），正则表达式必须决定何时尝试匹配更多的字符；如果遇到分支（通过|操作符），它必须从这些选项中选择一个进行尝试。

Each time the regex makes such a decision, it remembers the other options to return to later if necessary. If the chosen option is successful, the regex continues through the regex pattern, and if the remainder of the regex is also successful, the match is complete. But if the chosen option can't find a match or anything later in the regex fails, the regex backtracks to the last decision point where untried options remain and chooses one. It continues on like this until a match is found or all possible permutations of the quantifiers and alternation options in the regex have been tried unsuccessfully, at which point it gives up and moves on to start this process all over at the next character in the string.

每当正则表达式做出这样的决定，如果有必要的话，它会记住另一个选项，以备将来返回后使用。如果所选方案匹配成功，正则表达式将继续扫描正则表达式模板，如果其余部分匹配也成功了，那么匹配就结束了。但是如果所选择的方案未能发现相应匹配，或者后来的匹配也失败了，正则表达式将回溯到最后一个决策点，然后在剩余的选项中选择一个。它继续这样下去，直到找到一个匹配，或者量词和分支选项的所有可能的排列组合都尝试失败了，那么它将放弃这一过程，然后移动到此过程开始位置的下一个字符上，重复此过程。

Alternation and backtracking 分支和回溯

Here's an example that demonstrates how this process plays out with alternation.

下面的例子演示了这一过程是如何处理分支的。

```
/h(ello|appy) hippo/.test("hello there, happy hippo");
```

This regex matches "hello hippo" or "happy hippo". It starts this test by searching for an **h**, which it finds immediately as the first character in the target string. Next, the subexpression_r(ello|appy) provides two ways to proceed. The regex chooses the leftmost option (alternation always works from left to right), and checks whether **ello** matches the next characters in the string. It does, and the regex is also able to match the following space character. At that point, though, it reaches a dead end because the **h** in **hippo** cannot match the **t** that comes next in the string. The regex can't give up yet, though, because it hasn't tried all of its options, so it backtracks to the last decision point (just after it matched the leading **h**) and tries to match the second alternation option. That doesn't work, and since there are no more options to try, the regex determines that a match cannot be found starting from the first character in the string and moves on to try again at the second character. It doesn't find an **h** there, so it continues searching until it reaches the 14th character, where it matches the h in "happy". It then steps through the alternatives again. This time **ello** doesn't match, but after backtracking and trying the second alternative, it's able to continue until it matches the full string "happy hippo" (see Figure 5-4). Success.

此正则表达式匹配“hello hippo”或“happy hippo”。测试一开始，它要查找一个 h，目标字符串的第一个字母恰好就是 h，它立刻就被找到了。接下来，子表达式（ello|appy）提供了两个处理选项。正则表达式选择最左边的选项（分支选择总是从左到右进行），检查 ello 是否匹配字符串的下一个字符。确实匹配，然后正则表达式又匹配了后面的空格。然而在这一点上它走进了死胡同，因为 hippo 中的 h 不能匹配字符串中的下一个字母 t。此时正则表达式还不能放弃，因为它还没有尝试过所有的选择，随后它回溯到最后一个检查点（在它匹配了首字母 h 之后的那个位置上）并尝试匹配第二个分支选项。但是没有成功，而且也没有更多的选项了，所以正则表达式认为从字符串的第一个字符开始匹配是不能成功的，因此它从第二个字符开始，重新进行查找。它没有找到 h，所以就继续向后找，直到第 14 个字母才找到，它匹配 happy 的那个 h。然后它再次进入分支过程。这次 ello 未能匹配，但是回溯之后第二次分支过程中，它匹配了整个字符串“happy hippo”（如图 5-4）。匹配成功了。

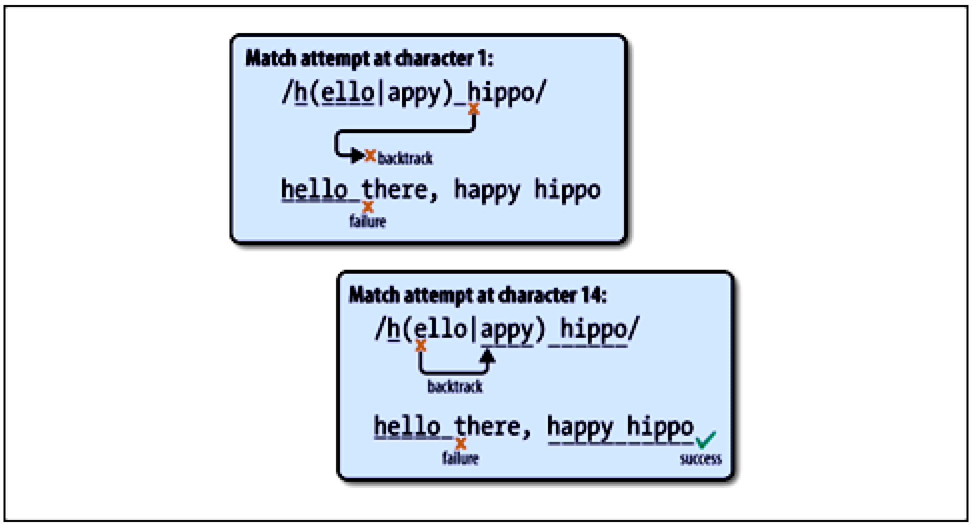


Figure 5-4. Example of backtracking with alternation

图 5-4 分支回溯的例子

## Repetition and backtracking 重复与回溯

This next example shows how backtracking works with repetition quantifiers.

下一个例子显示了带重复量词的回溯。

```
var str = "<p>Para 1.</p>" +
        "<img src='smiley.jpg'>" +
        "<p>Para 2.</p>" +
        "<div>Div.</div>";
/<p>.*<\/p>/i.test(str);
```

Here, the regex starts by matching the three literal characters **<p>** at the start of the string. Next up is **.***. The dot matches any character except line breaks, and the greedy asterisk quantifier repeats it zero or more times—as many times as possible. Since there are no line breaks in the target string, this gobbles up the rest of the string! There's still more to match in the regex pattern, though, so the regex tries to match <. This doesn't work at the end of the string, so the regex backtracks one character at a time, continually trying to match <, until it gets back to the < at the beginning of the **</div>** tag. It then tries to match \/ (an escaped backslash), which works, followed by **p**, which doesn't. The regex backtracks again, repeating this process until it eventually matches the **</p>** at the end of the second paragraph. The match is returned successfully, spanning（译者注：大概是 scanning） from the start of the first paragraph until the end of the last one, which is probably not what you wanted.

正则表达式一上来就匹配了字符串开始的三个字母<p>。然后是.*。点号匹配除换行符以外的任意字符，星号这个贪婪量词表示重复零次或多次——匹配尽量多的次数。因为目标字符串中没有换行符，它将吞噬剩下的全部字符串！不过正则表达式模板中还有更多内容需要匹配，所以正则表达式尝试匹配<。它在字符串末尾匹配不成功，所以它每次回溯一个字符，继续尝试匹配<，直到它回到</div>标签的<位置。然后它尝试匹配\/（转义反斜杠），匹配成功，然后是 p，匹配不成功。正则表达式继续回溯，重复此过程，直到第二段末尾时它终于匹配了</p>。匹配返回成功，它从第一段头部一直扫描到最后一个的末尾，这可能不是你想要的结果。

You can change the regex to match individual paragraphs by replacing the greedy ***** quantifier with the lazy (aka nongreedy) ***?**. Backtracking for lazy quantifiers works in the opposite way. When the regex **/<p>.*?<\/p>/** comes to the **.*?**, it first tries to skip this altogether and move on to matching **<\/p>**. It does so because ***?** repeats its preceding element zero or more times, as few times as possible, and the fewest possible times it can repeat is zero. However, when the following < fails to match at this point in the string, the regex backtracks and tries to match the next fewest number of characters: one. It continues backtracking forward like this until the **<\/p>** that follows the quantifier is able to fully match at the end of the first paragraph.

你可以将正则表达式中的贪婪量词*改为懒惰（又名非贪婪）量词*?，以匹配单个段落。懒惰量词的回溯工作以相反方式进行。当正则表达式/<p>.*?<\/p>/推进到.*?时，它首先尝试全部跳过然后继续匹配<\/p>。它这么做是因为*?匹配零次或多次，但尽可能少重复，尽可能少的话那么它就可以重复零次。但是，当随后的<在字符串的这一点上匹配失败时，正则表达式回溯并尝试下一个最小的字符数：一个。它继续像这样向前回溯到第一段的末尾，在那里量词后面的<\/p>得到完全匹配。

You can see that even if there was only one paragraph in the target string and therefore the greedy and lazy versions of this regex were equivalent, they would go about finding their matches differently (see Figure 5-5).

如果目标字符串只有一个段落，你可以看到此正则表达式的贪婪版本和懒惰版本是等价的，但是他们尝试匹配的过程不同（如图 5-5）。

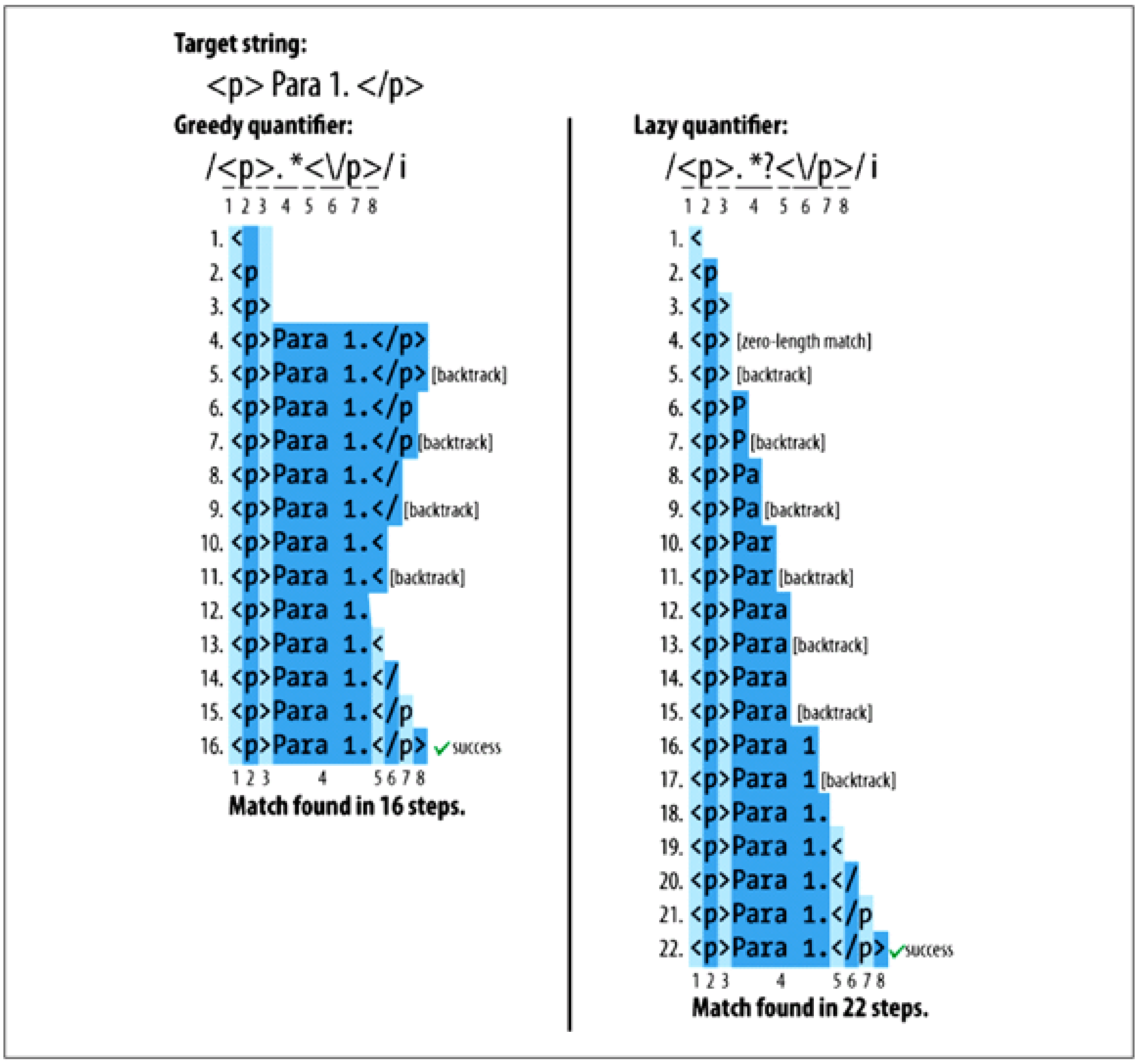


Figure 5-5. Example of backtracking with greedy and lazy quantifiers

图 5-5　回溯与贪婪量词和懒惰量词

**Runaway Backtracking**　回溯失控

When a regular expression stalls your browser for seconds, minutes, or longer, the problem is most likely a bad case of runaway backtracking. To demonstrate the problem, consider the following regex, which is designed to match an entire HTML file. The regex is wrapped across multiple lines in order to fit the page. Unlike most other regex flavors, JavaScript does not have an option to make dots match any character, including line breaks, so this example uses **[\s\S]** to match any character.

当一个正则表达式占用浏览器上秒，上分钟或者更长时间时，问题原因很可能是回溯失控。为说明此问题，考虑下面的正则表达式，它的目标是匹配整个 HTML 文件。此表达式被拆分成多行是为了适合页面显示。不像其他大多数正则表达式那样，JavaScript 没有选项可使点号匹配任意字符，包括换行符，所以此例中以[\s\S]匹配任意字符。

```
/<html>[\s\S]*?<head>[\s\S]*?<title>[\s\S]*?<\/title>[\s\S]*?<\/head>
[\s\S]*?<body>[\s\S]*?<\/body>[\s\S]*?<\/html>/
```

This regex works fine when matching a suitable HTML string, but it turns ugly when the string is missing one or more required tags. If the **</html>** tag is missing, for instance, the last **[\s\S]*?** expands to the end of the string since there is no **</html>** tag to be found, and then, instead of giving up, the regex sees that each of the previous **[\s\S]*?** sequences remembered backtracking positions that allow them to expand further. The regex tries expanding the second-to-last **[\s\S]*?**—using it to match the **</body>** tag that was previously matched by the literal **<\/body>** pattern in the regex— and continues to expand it in search of a second **</body>** tag until the end of the string is reached again. When all of that fails, the third-to-last **[\s\S]*?** expands to the end of the string, and so on.

此正则表达式匹配正常 HTML 字符串时工作良好，但是如果目标字符串缺少一个或多个标签时，它就会变得十分糟糕。例如</html>标签缺失，那么最后一个[\s\S]*?将扩展到字符串的末尾，因为在那里没有发现</html>标签，然后并没有放弃，正则表达式将察看此前的[\s\S]*?队列记录的回溯位置，使它们进一步扩大。正则表达式尝试扩展倒数第二个[\s\S]*?——用它匹配</body>标签，就是此前匹配过正则表达式模板<\/body>的那个标签——然后继续查找第二个</body>标签直到字符串的末尾。当所有这些步骤都失败了，倒数第三个[\s\S]*?将被扩展直至字符串的末尾，依此类推。

## The solution: Be specific 解决方法：具体化

The way around a problem like this is to be as specific as possible about what characters can be matched between your required delimiters. Take the pattern "**.*?**", which is intended to match a string delimited by double-quotes. By replacing the overly permissive **.*?** with the more specific **[^"\r\n]***, you remove the possibility that backtracking will force the dot to match a double-quote and expand beyond what was intended.

此类问题的解决办法在于尽可能具体地指出分隔符之间的字符匹配形式。例如模板".*?"用于匹配双引号包围的一个字符串。用更具体的[^"\rn]*取代过于宽泛的.*?，就去除了回溯时可能发生的几种情况，如尝试用点号匹配引号，或者扩展搜索超出预期范围。

With the HTML example, this workaround is not as simple. You can't use a negated character class like **[^<]** in place of **[\s\S]** because there may be other tags between those you're searching for. However, you can reproduce the effect by repeating a noncapturing group that contains a negative lookahead (blocking the next required tag) and the **[\s\S]** (any character) metasequence. This ensures that the tags you're looking for fail at every intermediate position, and, more importantly, that the **[\s\S]** patterns cannot expand beyond where the tags you are blocking via negative lookahead are found. Here's how the regex ends up looking using this approach:

在 HTML 的例子中解决办法不是那么简单。你不能使用否定字符类型如[^<]替代[\s\S]因为在搜索过程中可能会遇到其他类型的标签。但是，你可以通过重复一个非捕获组来达到同样效果，它包含一个回顾（阻塞下一个所需的标签）和[\s\S]（任意字符）元序列。这确保中间位置上你查找的每个标签都会失败，然后，更重要的是，[\s\S]模板在你在回顾过程中阻塞的标签被发现之前不能被扩展。应用此方法后正则表达式最终修改如下：

```
/<html>(?:(?!<head>)[\s\S])*<head>(?:(?!<title>)[\s\S])*<title>
(?:(?!<\/title>)[\s\S])*<\/title>(?:(?!<\/head>)[\s\S])*<\/head>
(?:(?!<body>)[\s\S])*<body>(?:(?!<\/body>)[\s\S])*<\/body>
(?:(?!<\/html>)[\s\S])*<\/html>/
```

Although this removes the potential for runaway backtracking and allows the regex to fail at matching incomplete HTML strings in linear time, it's not going to win any awards for efficiency. Repeating a lookahead for each matched character like this is rather inefficient in its own right and significantly slows down successful matches. This approach works well enough when matching short strings, but since in this case the lookaheads may need to be tested thousands of times in order to match an HTML file, there's another solution that works better. It relies on a little trick, and it's described next.

虽然这样做消除了潜在的回溯失控，并允许正则表达式匹配不完整 HTML 字符串失败时，其使用时间与文本长度呈线性关系，但是它的效率并没有提高。像这样为每个匹配字符多次前瞻缺乏效率，而且成功

匹配过程也相当慢。匹配较短字符串时此方法相当不错，但匹配一个 HTML 文件可能需要前瞻并测试上千次。另外一种解决方案更好，它使用了一点小技巧，如下：

## **Emulating atomic groups using lookahead and backreferences** 使用前瞻和后向引用列举原子组

Some regex flavors, including .NET, Java, Oniguruma, PCRE, and Perl, support a feature called atomic grouping. Atomic groups—written as **(?>…)**, where the ellipsis represents any regex pattern—are noncapturing groups with a special twist. As soon as a regex exits an atomic group, any backtracking positions within the group are thrown away. This provides a much better solution to the HTML regex's backtracking problem: if you were to place each **[\s\S]*?** sequence and its following HTML tag together inside an atomic group, then every time one of the required HTML tags was found, the match thus far would essentially be locked in. If a later part of the regex failed to match, no backtracking positions would be remembered for the quantifiers within the atomic groups, and thus the **[\s\S]*?** sequences could not attempt to expand beyond what they already matched.

一些正则表达式引擎，如.NET，Java，Oniguruma，PCRE，Perl，支持一种称作原子组的属性。原子组，写作(?>…)（译者注：有的书上称“贪婪子表达式”），省略号表示任意正则表达式模板——非捕获组和一个特殊的扭曲。存在于原子组中的正则表达式组中的任何回溯点都将被丢弃。这就为 HTML 正则表达式的回溯问题提供了一个更好的解决办法：如果你将[\s\S]*?序列和它后面的 HTML 标记一起放在一个原子组中，每当所需的 HTML 标签被发现一次，这次匹配基本上就被锁定了。如果正则表达式的后续部分匹配失败，原子组中的量词没有记录回溯点，因此[\s\S]*?序列就不能扩展到已匹配的范围之外。

That's great, but JavaScript does not support atomic groups or provide any other feature to eliminate needless backtracking. It turns out, though, that you can emulate atomic groups by exploiting a little-known behavior of lookahead: that lookaheads are atomic groups. The difference is that lookaheads don't consume any characters as part of the overall match; they merely check whether the pattern they contain can be matched at that position. However, you can get around this by wrapping a lookahead's pattern inside a capturing group and adding a backreference to it just outside the lookahead.Here's what this looks like:

这是了不起的技术。但是，JavaScript 不支持原子组，也不提供其他方法消除不必要的回溯。不过，你可以利用前瞻过程中一项鲜为人知的行为来模拟原子组：前瞻也是原子组。不同的是，前瞻在整个匹配过程中，不消耗字符；它只是检查自己包含的模板是否能在当前位置匹配。然而，你可以避开这点，在捕获组中包装一个前瞻模板，在前瞻之外向它添加一个后向引用。它看起来是下面这个样子：

```
(?=(pattern to make atomic))\1
```

This construct is reusable in any pattern where you want to use an atomic group. Just keep in mind that you need to use the appropriate backreference number if your regex contains more than one capturing group.

在任何你打算使用原子组的模式中这个结构都是可重用的。只要记住，你需要使用适当的后向引用次数如果你的正则表达式包含多个捕获组。

Here's how this looks when applied to the HTML regex:

HTML 正则表达式使用此技术后修改如下：

```
/<html>(?=([\s\S]*?<head>))\1(?=([\s\S]*?<title>))\2(?=([\s\S]*?
<\/title>))\3(?=([\s\S]*?<\/head>))\4(?=([\s\S]*?<body>))\5
(?=([\s\S]*?<\/body>))\6[\s\S]*?<\/html>/
```

Now, if there is no trailing **</html>** and the last **[\s\S]*?** expands to the end of the string, the regex immediately fails because there are no backtracking points to return to. Each time the regex finds an intermediate tag and exits a lookahead, it throws away all backtracking positions from within the lookahead. The following backreference simply rematches the literal characters found within the lookahead, making them a part of the actual match.

现在如果没有尾随的</html>那么最后一个[\s\S]*?将扩展至字符串结束，正则表达式将立刻失败因为没有回溯点可以返回。正则表达式每次找到一个中间标签就退出一个前瞻，它在前瞻过程中丢弃所有回溯位置。下一个后向引用简单地重新匹配前瞻过程中发现的字符，将他们作为实际匹配的一部分。

**Nested quantifiers and runaway backtracking** 嵌套量词和回溯失控

So-called nested quantifiers always warrant extra attention and care in order to ensure that you're not creating the potential for runaway backtracking. A quantifier is nested when it occurs within a grouping that is itself repeated by a quantifier (e.g., **(x+)***).

所谓嵌套量词总是需要额外的关注和小心，以确保没有掩盖回溯失控问题。嵌套量词指的是它出现在一个自身被重复量词修饰的组中（例如(x+)*）。

Nesting quantifiers is not actually a performance hazard in and of itself. However, if you're not careful, it can easily create a massive number of ways to divide text between the inner and outer quantifiers while attempting to match a string.

嵌套量词本身并不会造成性能危害。然而，如果你不小心，它很容易在尝试匹配字符串过程中，在内部量词和外部量词之间，产生一大堆分解文本的方法。

As an example, let's say you want to match HTML tags, and you come up with the following regex:

例如，假设你想匹配的 HTML 标签，使用了下面的正则表达式：

```
/<(?:[^>"']|"[^"]*"|'[^']*')*>/
```

This is perhaps overly simplistic, as it does not handle all cases of valid and invalid markup correctly, but it might work OK if used to process only snippets of valid HTML. Its advantage over even more naive solutions such as **/<[^>]*>/** is that it accounts for **>** characters that occur within attribute values. It does so using the second and third alternatives in the noncapturing group, which match entire double- and single-quoted attribute values in single steps, allowing all characters except their respective quote type to occur within them.

这也许过于简单，因为它不能正确处理所有情况的有效和无效标记，但它处理有效 HTML 片段时应该没什么问题。与更加幼稚的/<[^>]*>/相比，它的优势在于涵盖了出现在属性值中的>符号。在非捕获组中它不使用第二个和第三个分支，它们匹配单引号和双引号包围的属性值，除特定的引号外允许所有字符出现。

So far, there's no risk of runaway backtracking, despite the nested ***** quantifiers. The second and third alternation options match exactly one quoted string sequence per repetition of the group, so the potential number of backtracking points increases linearly with the length of the target string.

到目前为止还没有回溯失控的危险，尽管遇到了嵌套量词*。分组的每次重复过程中，第二和第三分支选项严格匹配一个带引号的字符串，所以潜在的回溯点数目随目标字符串长度而线性增长。

However, look at the first alternative in the noncapturing group: **[^>"']**. This can match only one character at a time, which seems a little inefficient. You might think it would be better to add a **+** quantifier at the end of this character class so that more than one suitable character can be matched during each repetition of the group—and at positions within the target string where the regex finds a match—and you'd be right. By matching more than one character at a time, you'd let the regex skip many unnecessary steps on the way to a successful match.

但是，察看非捕获组的第一个分支：[^>"']，每次只匹配一个字符，似乎有些效率低下。你可能认为在字符类后面加一个+量词会更好些，这样一来每次组重复过程就可以匹配多于一个的字符了。正则表达式可以在目标字符串的那个位置上发现一个匹配。你是对的，通过每次匹配多个字符，你让正则表达式在成功匹配的过程中跳过许多不必要的步骤。

What might not be as readily apparent is the negative consequence such a change could lead to. If the regex matches an opening < character, but there is no following > that would allow the match attempt to complete successfully, runaway backtracking will kick into high gear because of the huge number of ways the new inner quantifier can be combined with the outer quantifier (following the noncapturing group) to match the text that follows <. The regex must try all of these permutations before giving up on the match attempt. Watch out!

没有什么比这种改变所带来的负面效应更显而易见了。如果正则表达式匹配一个<字符，但后面没有>，却可以令匹配成功完成，回溯失控就会进入快车道，因为内部量词和外部量词的排列组合产生了数量巨大的分支路径（跟在非捕获组之后）用以匹配<之后的文本。正则表达式在最终放弃匹配之前必须尝试所有的排列组合。要当心啊！

**From bad to worse.** 从坏到更坏

For an even more extreme example of nested quantifiers resulting in runaway backtracking, apply the regex **/(A+A+)+B/** to a string containing only **A**s. Although this regex would be better written as **/AA+B/**, for the sake of discussion imagine that the two **A**s represent different patterns that are capable of matching some of the same strings.

关于嵌套量词导致回溯失控一个更极端的例子是，在一大串 A 上应用正则表达式/(A+A+)+B/。虽然这个正则表达式写成/AA+B/更好，但为了讨论方便，设想一下两个 A 能够匹配同一个字符串的多少种模板。

When applied to a string composed of 10 **A**s ("**AAAAAAAAAA**"), the regex starts by using the first **A+** to match all 10 characters. The regex then backtracks one character, letting the second **A+** match the last one. The

grouping then tries to repeat, but since there are no more **A**s and the group's **+** quantifier has already met its requirement of matching at least once, the regex then looks for the **B**. It doesn't find it, but it can't give up yet, since there are more paths through the regex that haven't been tried. What if the first **A+** matched eight characters and the second matched two? Or if the first matched three characters, the second matched two, and the group repeated twice? How about if during the first repetition of the group, the first **A+** matched two characters and the second matched three; then on the second repetition the first matched one and the second matched four? Although to you and me it's obviously silly to think that any amount of
backtracking will produce the missing **B**, the regex will dutifully check all of these futile options and a lot more. The worst-case complexity of this regex is an appalling O(2n), or two to the n^th power, where *n* is the length of the string. With the 10 **A**s used here, the regex requires 1,024 backtracking steps for the match to fail, and with 20 **A**s, that number explodes to more than a million. Thirty-five **A**s should be enough to hang Chrome, IE, Firefox, and Opera for at least 10 minutes (if not permanently) while they process the more than 34 billion backtracking steps required to invalidate all permutations of the regex. The exception is recent versions of Safari, which are able to detect that the regex is going in circles and quickly abort the match (Safari also imposes a cap of allowed backtracking steps, and aborts match attempts when this is exceeded).

当应用在一个由 10 个 A 组成的字符串上（“AAAAAAAAAA”），正则表达式首先使用第一个 A+匹配了所有 10 个字符。然后正则表达式回溯一个字符，让第二个 A+匹配最后一个字符。然后这个分组试图重复，但是没有更多的 A 而且分组中的+量词已经符合匹配所需的最少一次，因此正则表达式开始查找 B。它没有找到，但是还不能放弃，因为还有许多路径没有被测试过。如果第一个 A+匹配 8 个字符，第二个 A+匹配 2 个字符会怎么样呢？或者第一个匹配 3 个，第二个匹配 2 个，分组重复两次，又会怎么样呢？如果在分组的第一遍重复中，第一个 A+匹配 2 个字符，第二个 A+匹配 3 个字符，然后第二遍重复中，第一个匹配 1 个，第二个匹配 4 个，又怎么样呢？虽然你我都不会笨到认为多次回溯后可以找到那个并不存在的 B，但是正则表达式只会忠实地一次又一次地检查所有这些无用的选项。此正则表达式最坏情况的复杂性是一个惊人的 O(2n)，也就是 2 的 n 次方。n 表示字符串的长度。在 10 个 A 构成的字符串中，正则表达式需要 1024 次回溯才能确定匹配失败，如果是 20 个 A，该数字剧增到一百万以上。25 个 A 足以挂起 Chrome，IE，Firefox，和 Opera 至少 10 分钟（如果还没死机的话）用以处理超过三千四百万次回溯以排除正则表达式的各种排列组合。唯一的例外是最新的 Safari，它能够检测正则表达式陷入了循环，并快速终止匹配（Safari 还限定了回溯的次数，超出则终止匹配尝试）。

The key to preventing this kind of problem is to make sure that two parts of a regex cannot match the same part of a string. For this regex, the fix is to rewrite it as **/AA+B/**, but the issue may be harder to avoid with complex regexes. Adding an emulated atomic group often works well as a last resort, although other solutions, when possible, will most likely keep your regexes easier to understand. Doing so for this regex looks like **/((?=(A+A+))\2)+B/**, and completely removes the backtracking problem.

预防此类问题的关键是确保正则表达式的两个部分不能对字符串的同一部分进行匹配。这个正则表达式可重写为/AA+B/，但复杂的正则表达式可能难以避免此类问题。增加一个模拟原子组往往作为最后一招使用，虽然还有其他解决办法，如果可能的话，尽可能保持你的正则表达式简单易懂。如果这么做此正则表达式将改成/((?=(A+A+))\2)+B/，就彻底消除了回溯问题。

## A Note on Benchmarking 测试基准说明

Because a regex's performance can be wildly different depending on the text it's applied to, there's no straightforward way to benchmark regexes against each other. For the best result, you need to benchmark your regexes on test strings of varying lengths that match, don't match, and nearly match.

因为正则表达式性能因应用文本不同而产生很大差异，没有简单明了的方法可以测试正则表达式之间的性能差别。为得到最好的结果，你需要在各种字符串上测试你的正则表达式，包括不同长度，能够匹配的，不能匹配的，和近似匹配的。

That's one reason for this chapter's lengthy backtracking coverage. Without a firm understanding of backtracking, you won't be able to anticipate and identify backtracking-related problems. To help you catch runaway backtracking early, always test your regexes with long strings that contain partial matches. Think about the kinds of strings that your regexes will nearly but not quite match, and include those in your tests.

这也是本章长篇大论回溯的原因之一。如果没有确切理解回溯，就无法预测和确定回溯相关问题。为帮助你早日把握回溯失控，总是用包含特殊匹配的长字符串测试你的正则表达式。针对你的正则表达式构思一些近似但不能完全匹配的字符串，将他们应用在你的测试中。

## More Ways to Improve Regular Expression Efficiency 提高正则表达式效率的更多方法

The following are a variety of additional regex efficiency techniques. Several of the points here have already been touched upon during the backtracking discussion.

下面是一写提高正则表达式效率的技术。几个技术点已经在回溯部分讨论过了。

Focus on failing faster

关注如何让匹配更快失败

Slow regex processing is usually caused by slow failure rather than slow matching. This is compounded by the fact that if you're using a regex to match small parts of a large string, the regex will fail at many more positions than it will succeed. A change that makes a regex match faster but fail slower (e.g., by increasing the number of backtracking steps needed to try all regex permutations) is usually a losing trade.

正则表达式处理慢往往是因为匹配失败过程慢，而不是匹配成功过程慢。如果你使用正则表达式匹配一个很大字符串的一小部分，情况更为严重，正则表达式匹配失败的位置比匹配成功的位置要多得多。如果一个修改使正则表达式匹配更快但失败更慢（例如，通过增加所需的回溯次数去尝试所有分支的排列组合），这通常是一个失败的修改。

Start regexes with simple, required tokens

正则表达式以简单的，必需的字元开始

Ideally, the leading token in a regex should be fast to test and rule out as many obviously nonmatching positions as possible. Good starting tokens for this purpose are anchors (**^** or **$**), specific characters (e.g., **x** or **\u263A**), character classes (e.g., **[a-z]** or shorthands like **\d**), and word boundaries (**\b**). If possible, avoid starting regexes with groupings or optional tokens, and avoid top-level alternation such as **/one|two/** since that forces the regex to consider multiple leading tokens. Firefox is sensitive to the use of any quantifier on leading tokens, and is better able to optimize, e.g., **\s\s*** than **\s+** or **\s{1,}**. Other browsers mostly optimize away such differences.

最理想的情况是，一个正则表达式的起始字元应当尽可能快速地测试并排除明显不匹配的位置。用于此目的好的起始字元通常是一个锚（^或$），特定字符（例如 x 或\u363A），字符类（例如，[a-z]或速记符例如\d），和单词边界（\b）。如果可能的话，避免以分组或选择字元开头，避免顶级分支例如/one|two/，

因为它强迫正则表达式识别多种起始字元。Firefox 对起始字元中使用的任何量词都很敏感，能够优化的更好，例如，以\s\s*替代\s+或\s{1,}。其他浏览器大多优化掉这些差异。

Make quantified patterns and their following token mutually exclusive

编写量词模板，使它们后面的字元互相排斥

When the characters that adjacent tokens or subexpressions are able to match overlap, the number of ways a regex will try to divide text between them increases. To help avoid this, make your patterns as specific as possible. Don't use "**.*?**" (which relies on backtracking) when you really mean "**[^"\r\n]***".

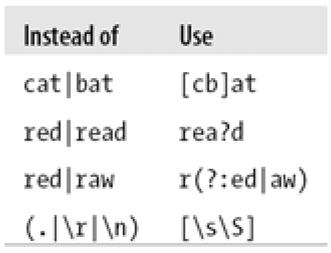
当字符与字元毗邻或子表达式能够重叠匹配时，一个正则表达式尝试分解文本的路径数量将增加。为避免此现象，尽量具体化你的模板。当你想表达“[^"\r\n]*”时不要使用“.*?”（依赖回溯）。

Reduce the amount and reach of alternation

减少分支的数量，缩小它们的范围

Alternation using the | vertical bar may require that all alternation options be tested at every position in a string. You can often reduce the need for alternation by using character classes and optional components, or by pushing the alternation further back into the regex (allowing some match attempts to fail before reaching the alternation). The following table shows examples of these techniques.

分支使用 | ，竖线，可能要求在字符串的每一个位置上测试所有的分支选项。你通常可通过使用字符类和选项组件减少对分支的需求，或将分支在正则表达式上的位置推后（允许到达分支之前的一些匹配尝试失败）。下表列出这些技术的例子。



| Instead of | Use |
|---|---|
| cat\|bat | [cb]at |
| red\|read | rea?d |
| red\|raw | r(?:ed\|aw) |
| (.\|\r\|\n) | [\s\S] |

Character classes are faster than alternation because they are implemented using bit vectors (or other fast implementations) rather than backtracking. When alternation is necessary, put frequently occurring alternatives first if this doesn't affect what the regex matches. Alternation options are attempted from left to right, so the more frequently an option is expected to match, the sooner you want it to be considered.

字符类比分支更快，因为他们使用位向量实现（或其他快速实现）而不是回溯。当分支必不可少时，将常用分支放在最前面，如果这样做不影响正则表达式匹配的话。分支选项从左向右依次尝试，一个选项被匹配上的机会越多，它被检测的速度就越快。

Note that Chrome and Firefox perform some of these optimizations automatically, and are therefore less affected by techniques for hand-tuning alternation.

注意 Chrome 和 Firefox 自动执行这些优化中的某些项目，因此较少受到手工调整的影响。

Use noncapturing groups

使用非捕获组

Capturing groups spend time and memory remembering backreferences and keeping them up to date. If you don't need a backreference, avoid this overhead by using a noncapturing group—i.e., **(?:…)** instead of **(…)**. Some people like to wrap their regexes in a capturing group when they need a backreference to the entire match. This is unnecessary since you can reference full matches via, e.g., element zero in arrays returned by **regex.exec()** or **$&** in replacement strings.

捕获组花费时间和内存用于记录后向引用，并保持它们是最新的。如果你不需要一个后向引用，可通过使用非捕获组避免这种开销——例如，(?:…)替代（…）。有些人当他们需要一个完全匹配的后向引用时，喜欢将他们的正则表达式包装在一个捕获组中。这是不必要的，因为你能够通过其他方法引用完全匹配，例如，使用 regex.exec()返回数组的第一个元素，或替换字符串中的$&。

Replacing capturing groups with their noncapturing kin has minimal impact in Firefox, but can make a big difference in other browsers when dealing with long strings.

用非捕获组取代捕获组在 Firefox 中影响很小，但在其他浏览器上处理长字符串时影响很大。

Capture interesting text to reduce postprocessing

捕获感兴趣的文字，减少后处理

As a caveat to the last tip, if you need to reference parts of a match, then, by all means, capture those parts and use the backreferences produced. For example, if you're writing code to process the contents of quoted strings matched by a regex, use /**"([^"]*)"**/ and work with backreference one, rather than using /**"[^"]*"**/ and manually stripping the quote marks from the result. When used in a loop, this kind of work reduction can save significant time.

最后一个告诫，如果你需要引用匹配的一部分，应当通过一切手段，捕获那些片断，再使用后向引用处理。例如，如果你写代码处理一个正则表达式所匹配的引号中的字符串内容，使用/"([^"]*)"/然后使用一次后向引用，而不是使用/"[^"]*"/然后从结果中手工剥离引号。当在循环中使用时，减少这方面的工作可以节省大量时间。

Expose required tokens

暴露所需的字元

In order to help regex engines make smart decisions about how to optimize a search routine, try to make it easy to determine which tokens are required. When tokens are used within subexpressions or alternation, it's harder for regex engines to determine whether they are required, and some won't make the effort to do so. For instance, the regex /^**(ab|cd)**/ exposes its start-of-string anchor. IE and Chrome see this and prevent the regex from trying to find matches after the start of a string, thereby making this search near instantaneous regardless of string length.

However, because the equivalent regex **/(^ab|^cd)/** doesn't expose its ^ anchor, IE doesn't apply the same optimization and ends up pointlessly searching for matches at every position in the string.

为帮助正则表达式引擎在如何优化查询例程时做出明智的决策，应尽量简单地判断出那些必需的字元。当字元应用在子表达式或者分支中，正则表达式引擎很难判断他们是不是必需的，有些引擎并不作此方面努力。例如，正则表达式/^(ab|cd)/暴露它的字符串起始锚。IE 和 Chrome 会注意到这一点，并阻止正则表达式尝试查找字符串头端之后的匹配，从而使查找瞬间完成而不管字符串长度。但是，由于等价正则表达式/(^ab|^cd)/不暴露它的^锚，IE 无法应用同样的优化，最终无意义地搜索字符串并在每一个位置上匹配。

Use appropriate quantifiers

使用适当的量词

As described in the earlier section “Repetition and backtracking” on page 90, greedy and lazy quantifiers go about finding matches differently, even when they match the same strings. Using the more appropriate quantifier type (based on the anticipated amount of backtracking) in cases where they are equally correct can significantly improve performance, especially with long strings.

正如前一节《重复和回溯》所讨论过的那样，贪婪量词和懒惰量词即使匹配同样的字符串，其查找匹配过程也是不同的。在确保正确等价的前提下，使用更合适的量词类型（基于预期的回溯次数）可以显著提高性能，尤其在处理长字符串时。

Lazy quantifiers are particularly slow in Opera 9.x and earlier, but Opera 10 removes this weakness.

懒惰量词在 Opera 9.x 和更早版本上格外缓慢，但 Opera 10 消除了这一弱点。

Reuse regexes by assigning them to variables

将正则表达式赋给变量，以重用它们

Assigning regexes to variables lets you avoid repeatedly compiling them. Some people go overboard, using regex caching schemes that aim to avoid ever compiling a given pattern and flag combination more than once. Don't bother; regex compilation is fast, and such schemes likely add more overhead than they evade. The important thing is to avoid repeatedly recompiling regexes within loops. In other words, don't do this:

将正则表达式赋给变量以避免对它们重新编译。有人做的太过火，使用正则表达式缓存池，以避免对给定的模板和标记组合进行多次编译。不要过分忧虑，正则表达式编译很快，这样的缓存池所增加的负担可能超过他们所避免的。重要的是避免在循环体中重复编译正则表达式。换句话说，不要这样做：

```
while (/regex1/.test(str1)) {
  /regex2/.exec(str2);
  ...
}
```

Do this instead:

替代以如下做法：

```
var regex1 = /regex1/,
regex2 = /regex2/;
while (regex1.test(str1)) {
  regex2.exec(str2);
  ...
}
```

Split complex regexes into simpler pieces

将复杂的正则表达式拆分为简单的片断

Try to avoid doing too much with a single regex. Complicated search problems that require conditional logic are easier to solve and usually more efficient when broken into two or more regexes, with each regex searching within the matches of the last. Regex monstrosities that do everything in one pattern are difficult to maintain, and are prone to backtracking-related problems.

尽量避免一个正则表达式做太多的工作。复杂的搜索问题需要条件逻辑，拆分为两个或多个正则表达式更容易解决，通常也更高效，每个正则表达式只在最后的匹配结果中执行查找。在一个模板中完成所有工作的正则表达式怪兽很难维护，而且容易引起回溯相关的问题。

**When Not to Use Regular Expressions** 什么时候不应该使用正则表达式

When used with care, regexes are very fast. However, they're usually overkill when you are merely searching for literal strings. This is especially true if you know in advance which part of a string you want to test. For instance, if you want to check whether a string ends with a semicolon, you could use something like this:

小心使用它，正则表达式是非常快的。然而，当你只是搜索文字字符串时它们经常矫枉过正。尤其当你事先知道了字符串的哪一部分将要被测试时。例如，如果你想检查一个字符串是不是以分号结束，你可以使用：

```
endsWithSemicolon = /;$/.test(str);
```

You might be surprised to learn, though, that none of the big browsers are currently smart enough to realize in advance that this regex can match only at the end of the string. What they end up doing is stepping through the entire string. Each time a semicolon is found, the regex advances to the next token (**$**), which checks whether the match is at the end of the string. If not, the regex continues searching for a match until it finally makes its way through the entire string. The longer your string (and the more semicolons it contains), the longer this takes.

你可能觉得很奇怪，虽说当前没有哪个浏览器聪明到这个程度，能够意识到这个正则表达式只能匹配字符串的末尾。最终它们所做的将是一个一个地测试了整个字符串。每当发现了一个分号，正则表达式就前

进到下一个字元（$），检查它是否匹配字符串的末尾。如果不是这样的话，正则表达式就继续搜索匹配，直到穿越了整个字符串。字符串的长度越长（包含的分号越多），它占用的时间也越长。

In this case, a better approach is to skip all the intermediate steps required by a regex and simply check whether the last character is a semicolon:

这种情况下，更好的办法是跳过正则表达式所需的所有中间步骤，简单地检查最后一个字符是不是分号：

```
endsWithSemicolon = str.charAt(str.length - 1) == ";";
```

This is just a bit faster than the regex-based test with small target strings, but, more importantly, the string's length no longer affects the time needed to perform the test.

目标字符串很小时，这种方法只比正则表达式快一点，但更重要的是，字符串的长度不再影响执行测试所需要的时间。

This example used the **charAt** method to read the character at a specific position. The string methods **slice**, **substr**, and **substring** work well when you want to extract and check the value of more than one character at a specific position. Additionally, the **indexOf** and **lastIndexOf** methods are great for finding the position of literal strings or checking for their presence. All of these string methods are fast and can help you avoid invoking the overhead of regular expressions when searching for literal strings that don't rely on fancy regex features.

这个例子使用 charAt 函数在特定位置上读取字符。字符串函数 slice，substr，和 substring 可用于在特定位置上提取并检查字符串的值。此外，indexOff 和 lastIndexOf 函数非常适合查找特定字符串的位置，或者判断它们是否存在。所有这些字符串操作函数速度都很快，当您搜索那些不依赖正则表达式复杂特性的文本字符串时，它们有助于您避免正则表达式带来的性能开销。

## String Trimming 字符串修剪

Removing leading and trailing whitespace from a string is a simple but common task. Although ECMAScript 5 adds a native string trim method (and you should therefore start to see this method in upcoming browsers), JavaScript has not historically included it. For the current browser crop, it's still necessary to implement a trim method yourself or rely on a library that includes it.

去除字符串首尾的空格是一个简单而常见的任务。虽然 ECMAScript 5 添加了原生字符串修剪函数（你应该可以在即将出现的浏览器中看到它们），到目前为止 JavaScript 还没有包含它。对当前的浏览器而言，有必要自己实现一个修剪函数，或者依靠一个包含此功能的库。

Trimming strings is not a common performance bottleneck, but it serves as a decent case study for regex optimization since there are a variety of ways to implement it.

修剪字符串不是一个常见的性能瓶颈，但作为学习正则表达式优化的例子有多种实现方法。

**Trimming with Regular Expressions 用正则表达式修剪**

Regular expressions allow you to implement a trim method with very little code, which is important for JavaScript libraries that focus on file size. Probably the best all-around solution is to use two substitutions—one to remove leading whitespace and another to remove trailing whitespace. This keeps things simple and fast, especially with long strings.

正则表达式允许你用很少的代码实现一个修剪函数，这对 JavaScript 关心文件大小的库来说十分重要。可能最好的全面解决方案是使用两个子表达式：一个用于去除头部空格，另一个用于去除尾部空格。这样处理简单而迅速，特别是处理长字符串时。

```
 if (!String.prototype.trim) {
   String.prototype.trim = function() {
     return this.replace(/^\s+/, "").replace(/\s+$/, "");
   }
}
// test the new method...
// tab (\t) and line feed (\n) characters are
// included in the leading whitespace.
var str = " \t\n test string ".trim();
alert(str == "test string"); // alerts "true"
```

The if block surrounding this code avoids overriding the **trim** method if it already exists, since native methods are optimized and usually far faster than anything you can implement yourself using a JavaScript function. Subsequent implementations of this example assume that this conditional is in place, though it is not written out each time.

if 语句避免覆盖 trim 函数如果它已经存在，因为原生函数进行了优化，通常远远快于你用 JavaScript 自己写的函数。后面的例子假设这个条件已经判断过了，所以不是每次都写上。

You can give Firefox a performance boost of roughly 35% (less or more depending on the target string's length and content) by replacing **/\s+$/** (the second regex) with **/\s\s*$/**. Although these two regexes are functionally identical, Firefox provides additional optimization for regexes that start with a nonquantified token. In other browsers, the difference is less significant or is optimized differently altogether. However, changing the regex that matches at the beginning of strings to **/^\s\s*/** does not produce a measurable difference, because the leading ^ anchor takes care of quickly invalidating nonmatching positions (precluding a slight performance difference from compounding over thousands of match attempts within a long string).

你可以给 Firefox 一个大约 35%的性能提升（或多或少依赖于目标字符串的长度和内容）通过将/\s+$/（第二个正则表达式）替换成/\s\s*$/。虽然这两个正则表达式的功能完全相同，Firefox 却为那些以非量词字元开头的正则表达式提供额外的优化。在其他浏览器上，差异不显著，或者优化完全不同。然而，改变正则表达式，在字符串开头匹配/^\s\s*/不会产生明显差异，因为^锚需要照顾那些快速作废的非匹配位置（避免一个轻微的性能差异，因为在一个长字符串中可能产生上千次匹配尝试）。

Following are several more regex-based trim implementations, which are some of the more common alternatives you might encounter. You can see cross-browser performance numbers for all trim implementations described here in Table 5-2 at the end of this section. There are, in fact, many ways beyond those listed here that you can write a regular expression to help you trim strings, but they are invariably slower (or at least less consistently decent cross-browser) than using two simple substitutions when working with long strings.

以下是几个基于正则表达式的修剪实例，这是你可能会遇到的几个常见的替代品。你可以在本节末尾表 5-2 中看到这里讨论的 trim 实例在不同浏览器上的性能。事实上，除这里列出的之外还有许多方法，你可以写一个正则表达式来修剪字符串，但它们在处理长字符串时，总比用两个简单的表达式要慢（至少在跨浏览器时缺乏一致性）。

```
 // trim 2
String.prototype.trim = function() {
  return this.replace(/^\s+|\s+$/g, "");
}
```

This is probably the most common solution. It combines the two simple regexes via alternation, and uses the **/g** (global) flag to replace all matches rather than just the first (it will match twice when its target contains both leading and trailing whitespace). This isn't a terrible approach, but it's slower than using two simple substitutions when working with long strings since the two alternation options need to be tested at every character position.

这可能是最通常的解决方案。它通过分支功能合并了两个简单的正则表达式，并使用/g（全局）标记替换所有匹配，而不只是第一个（当目标字符串首尾都有空格时它将匹配两次）。这并不是一个可怕的方法，但是对长字符串操作时，它比使用两个简单的子表达式要慢，因为两个分支选项都要测试每个字符位置。

```
 // trim 3
String.prototype.trim = function() {
  return this.replace(/^\s*([\s\S]*?)\s*$/, "$1");
}
```

This regex works by matching the entire string and capturing the sequence from the first to the last nonwhitespace characters (if any) to backreference one. By replacing the entire string with backreference one, you're left with a trimmed version of the string.

这个正则表达式的工作原理是匹配整个字符串，捕获从第一个到最后一个非空格字符之间的序列，记入后向引用 1。然后使用后向引用 1 替代整个字符串，就留下了这个字符串的修剪版本。

This approach is conceptually simple, but the lazy quantifier inside the capturing group makes the regex do a lot of extra work (i.e., backtracking), and therefore tends to make this option slow with long target strings. After the regex enters the capturing group, the **[\s\S]** class's lazy ***?** quantifier requires that it be repeated as few times as possible. Thus, the regex matches one character at a time, stopping after each character to try to match the remaining **\s*$** pattern. If that fails because nonwhitespace characters remain somewhere after the current position in the string, the regex matches one more character, updates the backreference, and then tries the remainder of the pattern again.

这个方法概念简单，但捕获组里的懒惰量词使正则表达式进行了许多额外操作（例如，回溯），因此在操作长目标字符串时很慢。进入正则表达式捕获组时，[\s\S]类的懒惰量词*?要求它尽可能地减少重复次数。因此，这个正则表达式每匹配一个字符，都要停下来尝试匹配余下的\s*$模板。如果字符串当前位置之后存在非空格字符导致匹配失败，正则表达式将匹配一个或多个字符，更新后向引用，然后再次尝试模板的剩余部分。

Lazy repetition is particularly slow in Opera 9.x and earlier. Consequently, trimming long strings with this method in Opera 9.64 performs about 10 to 100 times slower than in the other big browsers. Opera 10 fixes this longstanding weakness, bringing this method's performance in line with other browsers.

在 Opera 9.x 和更早版本中懒惰重复特别慢。因此，这个方法在 Opera 9.64 上比其它大型浏览器慢了 10 到 100 倍。Opera 10 修正了这个长期存在的弱点，将此方法的性能提高到与其它浏览器相当的水平。

```
// trim 4
String.prototype.trim = function() {
  return this.replace(/^\s*([\s\S]*\S)?\s*$/, "$1");
}
```

This is similar to the last regex, but it replaces the lazy quantifier with a greedy one for performance reasons. To make sure that the capturing group still only matches up to the last nonwhitespace character, a trailing **\S** is required. However, since the regex must be able to match whitespace-only strings, the entire capturing group is made optional by adding a trailing question mark quantifier.

这个表达式与上一个很像，但出于性能原因以贪婪量词取代了懒惰量词。为确保捕获组只匹配到最后一个非空格字符，必需尾随一个\S。然而，由于正则表达式必需能够匹配全部由空格组成的字符串，整个捕获组通过尾随一个?量词而成为可选组。

Here, the greedy asterisk in **[\s\S]*** repeats its any-character pattern to the end of the string. The regex then backtracks one character at a time until it's able to match the following **\S**, or until it backtracks to the first character matched within the group (after which it skips the group).

在此，[\s\S]*中的贪婪量词"*"表示重复方括号中的任意字符模板直至字符串结束。然后正则表达式每次回溯一个字符，直到它能够匹配后面的\S，或者直到回溯到第一个字符而匹配整个组（然后它跳过这个组）。

Unless there's more trailing whitespace than other text, this generally ends up being faster than the previous solution that used a lazy quantifier. In fact, it's so much faster that in IE, Safari, Chrome, and Opera 10, it even beats using two substitutions. That's because those browsers contain special optimization for greedy repetition of character classes that match any character. The regex engine jumps to the end of the string without evaluating intermediate characters (although backtracking positions must still be recorded), and then backtracks as appropriate. Unfortunately, this method is considerably slower in Firefox and Opera 9, so at least for now, using two substitutions still holds up better cross-browser.

如果尾部空格不比其它字符串更多，它通常比前面那些使用懒惰量词的方案更快。事实上，它在 IE，Safari，Chrome 和 Opear 10 上如此之快，甚至超过使用两个子表达式的方案。因为这些浏览器包含特殊优化，专门服务于为字符类匹配任意字符的贪婪重复操作。正则表达式引擎直接跳到字符串末尾而不检查中间的字符（尽管回溯点必需被记下来），然后适当回溯。不幸的是，这种方法在 Firefox 和 Opera 9 上非常慢，所以到目前为止，使用两个子表达式仍然是更好的跨浏览器方案。

```
// trim 5
String.prototype.trim = function() {
  return this.replace(/^\s*(\S*(\s+\S+)*)\s*$/, "$1");
}
```

This is a relatively common approach, but there's no good reason to use it since it's consistently one of the slowest of the options shown here, in all browsers. It's similar to the last two regexes in that it matches the entire string and replaces it with the part you want to keep, but because the inner group matches only one word at a time, there are a lot of discrete steps the regex must take. The performance hit may be unnoticeable when trimming short strings, but with long strings that contain many words, this regex can become a performance problem.

这是一个相当普遍的方法，但没有很好的理由使用它，因为它在所有浏览器上都是这里列出所有方法中最慢的一个。它类似最后两个正则表达式，它匹配整个字符串然后用你打算保留的部分替换这个字符串，因为内部组每次只匹配一个单词，正则表达式必需执行大量的离散步骤。修剪短字符串时性能冲击并不明显，但处理包含多个词的长字符串时，这个正则表达式可以成为一个性能问题。

Changing the inner group to a noncapturing group—i.e., changing **(\s+\S+)** to **(?:\s+\S+)**—helps a bit, slashing roughly 20%–45% off the time needed in Opera, IE, and Chrome, along with much slighter improvements in Safari and Firefox. Still, a noncapturing group can't redeem this implementation. Note that the outer group cannot be converted to a noncapturing group since it is referenced in the replacement string.

将内部组修改为一个非捕获组——例如，将(\s+\S+)修改为（?:\s+\S+）——有一点帮助，在 Opera，IE 和 Chrome 上缩减了大约 20%-45%的处理时间，在 Safari 和 Firefox 上也有轻微改善。尽管如此，一个非捕获组不能完全代换这个实现。请注意，外部组不能转换为非捕获组，因为它在被替换的字符串中被引用了。

**Trimming Without Regular Expressions** 不使用正则表达式修剪

Although regular expressions are fast, it's worth considering the performance of trimming without their help. Here's one way to do so:

虽然正则表达式很快，还是值得考虑没有它们帮助时修剪字符串的性能。有一种方法这样做：

```
// trim 6
String.prototype.trim = function() {
  var start = 0,
      end = this.length - 1,
      ws = " \n\r\t\f\x0b\xa0\u1680\u180e\u2000\u2001\u2002\u2003
\u2004\u2005\u2006\u2007\u2008\u2009\u200a\u200b\u2028\u2029\u202f
\u205f\u3000\ufeff";
  while (ws.indexOf(this.charAt(start)) > -1) {
    start++;
  }
  while (end > start && ws.indexOf(this.charAt(end)) > -1) {
    end--;
  }
  return this.slice(start, end + 1);
}
```

The **ws** variable in this code includes all whitespace characters as defined by ECMAScript 5. For efficiency reasons, copying any part of the string is avoided until the trimmed version's start and end positions are known.

此代码中的 ws 变量包括 ECMAScript 5 中定义的所有空白字符。出于效率原因，在得到修剪区的起始和终止位置之前避免拷贝字符串的任何部分。

It turns out that this smokes the regex competition when there is only a bit of whitespace on the ends of the string. The reason is that although regular expressions are well suited for removing whitespace from the beginning of a string, they're not as fast at trimming from the end of long strings. As noted in the section “When Not to Use Regular Expressions” on page 99, a regex cannot jump to the end of a string without considering characters along the way. However, this implementation does just that, with the second **while** loop working backward from the end of the string until it finds a nonwhitespace character.

当字符串末尾只有少量空格时，这种情况使正则表达式陷入疯狂工作。原因是，尽管正则表达式很好地去除了字符串头部的空格，它们却不能同样快速地修剪长字符串的尾部。正如《什么时候不应该使用正则表达式》一节所提到的那样，一个正则表达式不能跳到字符串的末尾而不考虑沿途字符。然而，本实现正是如此，在第二个 while 循环中从字符串末尾向前查找一个非空格字符。

Although this version is not affected by the overall length of the string, it has its own weakness: long leading and trailing whitespace. That's because looping over characters to check whether they are whitespace can't match the efficiency of a regex's optimized search code.

虽然本实现不受字符串总长度影响，但它有自己的弱点：（它害怕）长的头尾空格。因为循环检查字符是不是空格在效率上不如正则表达式所使用的优化过的搜索代码。

### A Hybrid Solution　混合解决方案

The final approach for this section is to combine a regex's universal efficiency at trimming leading whitespace with the nonregex method's speed at trimming trailing characters.

本节中最后一个办法是将两者结合起来，用正则表达式修剪头部空格，用非正则表达式方法修剪尾部字符。

```
// trim 7
String.prototype.trim = function() {
  var str = this.replace(/^\s+/, ""),
  end = str.length - 1,
  ws = /\s/;
  while (ws.test(str.charAt(end))) {
    end--;
  }
  return str.slice(0, end + 1);
}
```

This hybrid method remains insanely fast when trimming only a bit of whitespace, and removes the performance risk of strings with long leading whitespace and whitespaceonly strings (although it maintains the weakness for strings with long trailing whitespace). Note that this solution uses a regex in the loop to check whether characters at the end of the string are whitespace. Although using a regex for this adds a bit of performance overhead, it lets you defer the list of whitespace characters to the browser for the sake of brevity and compatibility.

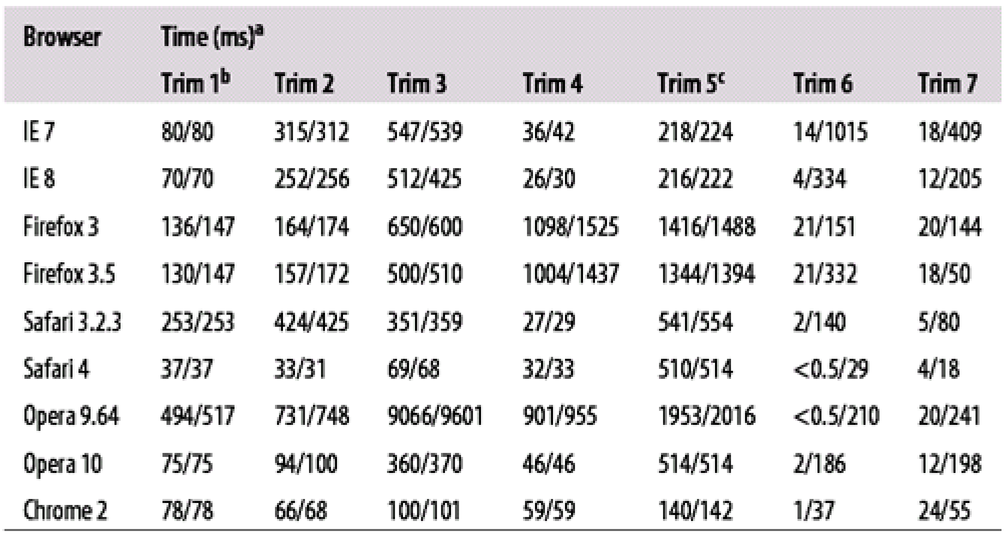
当只修剪一个空格时，此混合方法巨快无比，并去除了性能上的风险，诸如以长空格开头的字符串，完全由空格组成的字符串（尽管它在处理尾部长空格的字符串时仍然具有弱点）。请注意，此方案在循环中使用正则表达式检测字符串尾部的字符是否空格，尽管使用正则表达式增加了一点性能负担，但它允许你根据浏览器定义空格字符列表，以保持简短和兼容性。

The general trend for all trim methods described here is that overall string length has more impact than the number of characters to be trimmed in regex-based solutions, whereas nonregex solutions that work backward from the end of the string are unaffected by overall string length but more significantly affected by the amount of whitespace to trim. The simplicity of using two regex substitutions provides consistently respectable performance cross-browser with varying string contents and lengths, and therefore it's arguably the best all-around solution. The hybrid solution is exceptionally fast with long strings at the cost of slightly longer code and a weakness in some browsers for long, trailing whitespace. See Table 5-2 for all the gory details.

所有修剪方法总的趋势是：在基于正则表达式的方案中，字符串总长比修剪掉的字符数量更影响性能；而非正则表达式方案从字符串末尾反向查找，不受字符串总长的影响，但明显受到修剪空格数量的影响。简单地使用两个子正则表达式在所有浏览器上处理不同内容和长度的字符串时，均表现出稳定的性能。因此它可以说是最全面的解决方案。混合解决方案在处理长字符串时特别快，其代价是代码稍长，在某些浏览器上处理尾部长空格时存在弱点。表 5-2 是所有的测试细节。

Table 5-2. Cross-browser performance of various trim implementations

表 5-2　不同 trim 版本在各种浏览器上的性能



| Browser | Time (ms)[a] | | | | | | |
|---|---|---|---|---|---|---|---|
| | Trim 1[b] | Trim 2 | Trim 3 | Trim 4 | Trim 5[c] | Trim 6 | Trim 7 |
| IE 7 | 80/80 | 315/312 | 547/539 | 36/42 | 218/224 | 14/1015 | 18/409 |
| IE 8 | 70/70 | 252/256 | 512/425 | 26/30 | 216/222 | 4/334 | 12/205 |
| Firefox 3 | 136/147 | 164/174 | 650/600 | 1098/1525 | 1416/1488 | 21/151 | 20/144 |
| Firefox 3.5 | 130/147 | 157/172 | 500/510 | 1004/1437 | 1344/1394 | 21/332 | 18/50 |
| Safari 3.2.3 | 253/253 | 424/425 | 351/359 | 27/29 | 541/554 | 2/140 | 5/80 |
| Safari 4 | 37/37 | 33/31 | 69/68 | 32/33 | 510/514 | <0.5/29 | 4/18 |
| Opera 9.64 | 494/517 | 731/748 | 9066/9601 | 901/955 | 1953/2016 | <0.5/210 | 20/241 |
| Opera 10 | 75/75 | 94/100 | 360/370 | 46/46 | 514/514 | 2/186 | 12/198 |
| Chrome 2 | 78/78 | 66/68 | 100/101 | 59/59 | 140/142 | 1/37 | 24/55 |

**a** Reported times were generated by trimming a large string (40 KB) 100 times, first with 10 and then 1,000 spaces added to each end.

报告时间是修剪一个大字符串（40KB）100 次所用的时间，每个字符串以 10 个空格开头，以 1'000 个空格结尾。

**b** Tested without the **/\s\s*$/** optimization.

测试时关闭/\s\s*$/优化

**c** Tested without the noncapturing group optimization.

测试时关闭非捕获组优化

## Summary 总结

Intensive string operations and incautiously crafted regexes can be major performance obstructions, but the advice in this chapter helps you avoid common pitfalls.

密集的字符串操作和粗浅地编写正则表达式可能是主要性能障碍，但本章中的建议可帮助您避免常见缺陷。

• When concatenating numerous or large strings, array joining is the only method with reasonable performance in IE7 and earlier.

当连接数量巨大或尺寸巨大的字符串时，数组联合是 IE7 和它的早期版本上唯一具有合理性能的方法。

• If you don't need to worry about IE7 and earlier, array joining is one of the slowest ways to concatenate strings. Use simple + and += operators instead, and avoid unnecessary intermediate strings.

如果你不关心 IE7 和它的早期版本，数组联合是连接字符串最慢的方法之一。使用简单的+和+=取而代之，可避免（产生）不必要的中间字符串。

• Backtracking is both a fundamental component of regex matching and a frequent source of regex inefficiency.

回溯既是正则表达式匹配功能基本的组成部分，又是正则表达式影响效率的常见原因。

• Runaway backtracking can cause a regex that usually finds matches quickly to run slowly or even crash your browser when applied to partially matching strings. Techniques for avoiding this problem include making adjacent tokens mutually exclusive, avoiding nested quantifiers that allow matching the same part of a string more than one way, and eliminating needless backtracking by repurposing the atomic nature of lookahead.

回溯失控发生在正则表达式本应很快发现匹配的地方，因为某些特殊的匹配字符串动作，导致运行缓慢甚至浏览器崩溃。避免此问题的技术包括：使相邻字元互斥，避免嵌套量词对一个字符串的相同部分多次匹配，通过重复利用前瞻操作的原子特性去除不必要的回溯。

• A variety of techniques exist for improving regex efficiency by helping regexes find matches faster and spend less time considering nonmatching positions (see “More Ways to Improve Regular Expression Efficiency” on page 96).

提高正则表达式效率的各种技术手段，帮助正则表达式更快地找到匹配，以及在非匹配位置上花费更少时间（见《更多提高正则表达式效率的方法》）。

• Regexes are not always the best tool for the job, especially when you are merely searching for literal strings.

正则表达式并不总是完成工作的最佳工具，尤其当你只是搜索一个文本字符串时。

• Although there are many ways to trim a string, using two simple regexes (one to remove leading whitespace and another for trailing whitespace) offers a good mix of brevity and cross-browser efficiency with varying string contents and lengths. Looping from the end of the string in search of the first nonwhitespace characters, or combining this technique with regexes in a hybrid approach, offers a good alternative that is less affected by overall string length.

虽然有很多方法来修整一个字符串，使用两个简单的正则表达式（一个用于去除头部空格，另一个用于去除尾部空格）提供了一个简洁、跨浏览器的方法，适用于不同内容和长度的字符串。从字符串末尾开始循环查找第一个非空格字符，或者在一个混合应用中将此技术与正则表达式结合起来，提供了一个很好的替代方案，它很少受到字符串整体长度的影响。

# 第六章 Responsive Interfaces 响应接口

There's nothing more frustrating than clicking something on a web page and having nothing happen. This problem goes back to the origin of transactional web applications and resulted in the now-ubiquitous "please click only once" message that accompanies most form submissions. A user's natural inclination is to repeat any action that doesn't result in an obvious change, and so ensuring responsiveness in web applications is an important performance concern.

没有什么比点击页面上的东西却什么也没发生更令人感到挫折了。这个问题又回到了原始网页交互程序和现在已无处不在的提交表单时弹出的“请勿重复提交”消息上面。用户自然倾向于重复尝试这些不发生明显变化的动作，所以确保网页应用程序的响应速度也是一个重要的性能关注点。

Chapter 1 introduced the browser UI thread concept. As a recap, most browsers have a single process that is shared between JavaScript execution and user interface updates. Only one of these operations can be performed at a time, meaning that the user interface cannot respond to input while JavaScript code is executed and vice versa. The user interface effectively becomes "locked" when JavaScript is executing; managing how long your JavaScript takes to execute is important to the perceived performance of a web application.

第一节介绍了浏览器 UI 线程概念。总的来说，大多数浏览器有一个单独的处理进程，它由两个任务所共享：JavaScript 任务和用户界面更新任务。每个时刻只有其中的一个操作得以执行，也就是说当 JavaScript 代码运行时用户界面不能对输入产生反应，反之亦然。或者说，当 JavaScript 运行时，用户界面就被“锁定”了。管理好 JavaScript 运行时间对网页应用的性能很重要。

## The Browser UI Thread 浏览器 UI 线程

The process shared by JavaScript and user interface updates is frequently referred to as the browser UI thread (though the term "thread" is not necessarily accurate for all browsers). The UI thread works on a simple queuing system where tasks are kept until the process is idle. Once idle, the next task in the queue is retrieved and executed. These tasks are either JavaScript code to execute or UI updates to perform, which include redraws and reflows (discussed in Chapter 3). Perhaps the most interesting part of this process is that each input may result in one or more tasks being added to the queue.

JavaScript 和 UI 更新共享的进程通常被称作浏览器 UI 线程（虽然对所有浏览器来说“线程”一词不一定准确）。此 UI 线程围绕着一个简单的队列系统工作，任务被保存到队列中直至进程空闲。一旦空闲，队列中的下一个任务将被检索和运行。这些任务不是运行 JavaScript 代码，就是执行 UI 更新，包括重绘和重排版（在第三章讨论过）。此进程中最令人感兴趣的部分是每次输入均导致一个或多个任务被加入队列。

Consider a simple interface where a button click results in a message being displayed on the screen:

考虑这样一个简单的接口：按下一个按钮，然后屏幕上显示出一个消息：

```
<html>
  <head>
    <title>Browser UI Thread Example</title>
  </head>
  <body>
    <button onclick="handleClick()">Click Me</button>
    <script type="text/javascript">
      function handleClick(){
        var div = document.createElement("div");
        div.innerHTML = "Clicked!";
        document.body.appendChild(div);
      }
    </script>
  </body>
</html>
```

When the button in this example is clicked, it triggers the UI thread to create and add two tasks to the queue. The first task is a UI update for the button, which needs to change appearance to indicate it was clicked, and the second is a JavaScript execution task containing the code for **handleClick**(), so that the only code being executed is this method and anything it calls. Assuming the UI thread is idle, the first task is retrieved and executed to update the button's appearance, and then the JavaScript task is retrieved and executed. During the course of execution, **handleClick**() creates a new **<div>** element and appends it to the **<body>** element, effectively making another UI change. That means that during the JavaScript execution, a new UI update task is added to the queue such that the UI is updated once JavaScript execution is complete. See Figure 6-1.

当例子中的按钮被点击时，它触发 UI 线程创建两个任务并添加到队列中。第一个任务是按钮的 UI 更新，它需要改变外观以指示出它被按下了，第二个任务是 JavaScript 运行任务，包含 handleClick()的代码，被

运行的唯一代码就是这个方法和所有被它调用的方法。假设 UI 线程空闲，第一个任务被检查并运行以更新按钮外观，然后 JavaScript 任务被检查和运行。在运行过程中，handleClick()创建了一个新的<div>元素，并追加在<body>元素上，其效果是引发另一次 UI 改变。也就是说在 JavaScript 运行过程中，一个新的 UI 更新任务被添加在队列中，当 JavaScript 运行完之后，UI 还会再更新一次。如图 6-1。

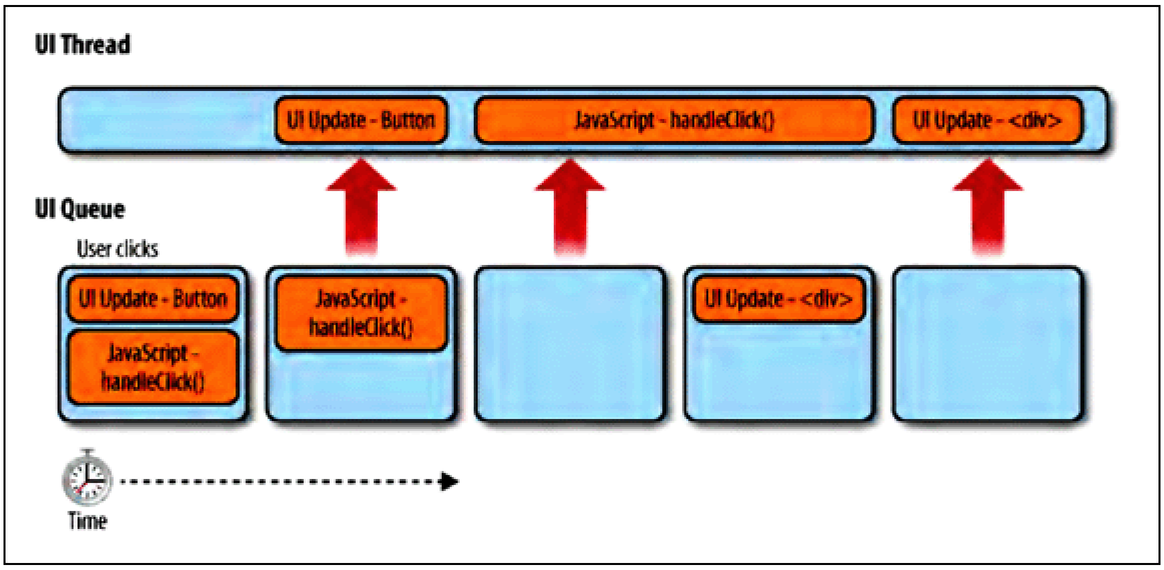


Figure 6-1. UI thread tasks get added as the user interacts with a page

图 6-1　用户与页面交互时向 UI 线程增加一条任务

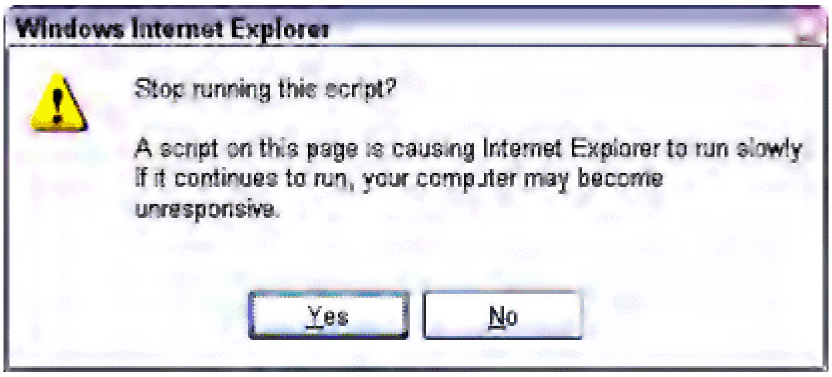
When all UI thread tasks have been executed, the process becomes idle and waits for more tasks to be added to the queue. The idle state is ideal because all user actions then result in an immediate UI update. If the user tries to interact with the page while a task is being executed, not only will there not be an immediate UI update, but a new task for a UI update may not even be created and queued. In fact, most browsers stop queuing tasks for the UI thread while JavaScript is executing, which means that it is imperative to finish JavaScript tasks as quickly as possible so as not to adversely affect the user's experience.

当所有 UI 线程任务执行之后，进程进入空闲状态，并等待更多任务被添加到队列中。空闲状态是理想的，因为所有用户操作立刻引发一次 UI 更新。如果用户企图在任务运行时与页面交互，不仅没有即时的 UI 更新，而且不会有新的 UI 更新任务被创建和加入队列。事实上，大多数浏览器在 JavaScript 运行时停止 UI 线程队列中的任务，也就是说 JavaScript 任务必须尽快结束，以免对用户体验造成不良影响。

**Browser Limits　浏览器限制**

Browsers place limits on the amount of time that JavaScript take to execute. This is a necessary limitation to ensure that malicious coders can't lock up a user's browser or computer by performing intensive operations that will never end. There are two such limits: the call stack size limit (discussed in Chapter 4) and the long-running script limit. The long-running script limit is sometimes called the long-running script timer or the runaway script timer, but the basic idea is that the browser keeps track of how long a script has been running and will stop it once a certain limit is hit. When the limit is reached, a dialog is displayed to the user, such as the one in Figure 6-2.

浏览器在 JavaScript 运行时间上采取了限制。这是一个有必要的限制，确保恶意代码编写者不能通过无尽的密集操作锁定用户浏览器或计算机。此类限制有两个：调用栈尺寸限制（第四章讨论过）和长时间脚本限制。长运行脚本限制有时被称作长运行脚本定时器或者失控脚本定时器，但其基本思想是浏览器记录一个脚本的运行时间，一旦到达一定限度时就终止它。当此限制到达时，浏览器会向用户显示一个对话框，如图 6-2 所示。


Windows Internet Explorer

Stop running this script?

A script on this page is causing Internet Explorer to run slowly If it continues to run, your computer may become unresponsive.

Figure 6-2. Internet Explorer's long-running script warning dialog is displayed when more than 5 million statements have been executed

图 6-2 Internet Explorer 的长运行脚本警告对话框，当运行超过 5 百万条语句时显示

There are two ways of measuring how long a script is executing. The first is to keep track of how many statements have been executed since the script began. This approach means that the script may run for different periods of time on different machines, as the available memory and CPU speed can affect how long it takes to execute a single statement. The second approach is to track the total amount of time that the script has been executing. The amount of script that can be processed within a set amount of time also varies based on the user's machine capabilities, but the script is always stopped after a set amount of time. Not surprisingly, each browser has a slightly different approach to long-running script detection:

有两种方法测量脚本的运行时间。第一个方法是统计自脚本开始运行以来执行过多少语句。此方法意味着脚本在不同的机器上可能会运行不同的时间长度，可用内存和 CPU 速度可以影响一条独立语句运行所花费的时间。第二种方法是统计脚本运行的总时间。在特定时间内可运行的脚本数量也因用户机器性能差异而不同，但脚本总是停在固定的时间上。毫不奇怪，每个浏览器对长运行脚本检查方法上略有不同：

• Internet Explorer, as of version 4, sets a default limit of 5 million statements; this limit is stored in a Windows registry setting called
HKEY_CURRENT_USER\Software\Microsoft\InternetExplorer\Styles\MaxScriptStatements.

Internet Explorer，在第 4 版中，设置默认限制为 5 百万条语句；此限制存放在 Windows 注册表中，叫做

HKEY_CURRENT_USER\Software\Microsoft\InternetExplorer\Styles\MaxScriptStatements

• Firefox has a default limit of 10 seconds; this limit is stored in the browser's configuration settings (accessible by typing **about:config** in the address box) as the dom.max_script_run_time key.

Firefox 默认限制为 10 秒钟，此限制存放在浏览器的配置设置中（在地址栏中输入 about:config）键名为 dom.max_script_run_time。

• Safari has a default limit of 5 seconds; this setting cannot be altered, but you can disable the timer by enabling the Develop menu and selecting Disable Runaway JavaScript Timer.

Safari 默认限制为 5 秒钟，此设置不能改变，但你可以关闭此定时，通过启动 Develop 菜单并选择"禁止失控 JavaScript 定时器"。

• Chrome has no separate long-running script limit and instead relies on its generic crash detection system to handle such instances.

Chrome 没有独立的长运行脚本限制，替代以依赖它的通用崩溃检测系统来处理此类实例。

• Opera has no long-running script limit and will continue to execute JavaScript code until it has finished, though, due to Opera's architecture, this will not cause system instability while the execution is completed.

Opera 没有长运行脚本限制，将继续运行 JavaScript 代码直至完成，由于 Opera 的结构，当运行结束时它并不会导致系统不稳定。

When the browser's long-running script limit is reached, a dialog is displayed to the user, regardless of any other error-handling code on the page. This is a major usability issue because most Internet users are not technically savvy and would therefore be confused about the meaning of the error message as well as which option (to stop the script or allow it to continue) is appropriate.

当浏览器的长时间脚本限制被触发时，有一个对话框显示给用户，而不管页面上的任何其他错误处理代码。这是一个主要的可用性问题，因为大多数互联网用户并不精通技术，会被错误信息所迷惑，不知道应该选择哪个选项（停止脚本或允许它继续运行）。

If your script triggers this dialog in any browser, it means the script is simply taking too long to complete its task. It also indicates that the user's browser has become unresponsive to input while the JavaScript code is continuing to execute. From a developer's point of view, there is no way to recover from a long-running script dialog's appearance; you can't detect it and therefore can't adjust to any issues that might arise as a result. Clearly, the best way to deal with long-running script limits is to avoid them in the first place.

如果你的脚本在浏览器上触发了此对话框，意味着脚本只是用太长的时间来完成任务。它还表明用户浏览器在 JavaScript 代码继续运行状态下无法响应输入。从开发者观点看，没有办法改变长运行脚本对话框的外观，你不能检测到它，因此不能用它来提示可能出现的问题。显然，长运行脚本最好的处理办法首先是避免它们。

### How Long Is Too Long? 多久才算“太久”？

Just because the browser allows a script to continue executing up to a certain number of seconds doesn't mean you should allow it do so. In fact, the amount of time that your JavaScript code executes continuously should be much smaller than the browser-imposed limits in order to create a good user experience. Brendan Eich, creator of JavaScript, is quoted as having once said, "[JavaScript] that executes in whole seconds is probably doing something wrong…."

浏览器允许脚本继续运行直至某个固定的时间，这并不意味着你可以允许它这样做。事实上，你的 JavaScript 代码持续运行的总时间应当远小于浏览器实施的限制，以创建良好的用户体验。Brendan Eich，JavaScript 的创造者，引用他的话说，“[JavaScript]运行了整整几秒钟很可能是做错了什么……”

If whole seconds are too long for JavaScript to execute, what is an appropriate amount of time? As it turns out, even one second is too long for a script to execute. The total amount of time that a single JavaScript operation should take (at a maximum) is 100 milliseconds. This number comes from research conducted by Robert Miller in 1968. Interestingly, usability expert Jakob Nielsen noted in his book Usability Engineering (Morgan Kaufmann, 1994) that this number hasn't changed over time and, in fact, was reaffirmed in 1991 by research at Xerox-PARC.

如果整整几秒钟对 JavaScript 运行来说太长了，那么什么是适当的时间？事实证明，即使一秒钟对脚本运行来说也太长了。一个单一的 JavaScript 操作应当使用的总时间（最大）是 100 毫秒。这个数字根据 Robert Miller 在 1968 年的研究。有趣的是，可用性专家 Jakob Nielsen 在他的著作《可用性工程》（Morgan Kaufmann，1944）上注释说这一数字并没有因时间的推移而改变，而且事实上在 1991 年被 Xerox-PARC（施乐公司的帕洛阿尔托研究中心）的研究中重申。

Nielsen states that if the interface responds to user input within 100 milliseconds, the user feels that he is "directly manipulating the objects in the user interface." Any amount of time more than 100 milliseconds means the user feels disconnected from the interface. Since the UI cannot update while JavaScript is executing, the user cannot feel in control of the interface if that execution takes longer than 100 milliseconds.

Nielsen 指出如果该接口在 100 毫秒内响应用户输入，用户认为自己是“直接操作用户界面中的对象。”超过 100 毫秒意味着用户认为自己与接口断开了。由于 UI 在 JavaScript 运行时无法更新，如果运行时间长于 100 毫秒，用户就不能感受到对接口的控制。

A further complication is that some browsers won't even queue UI updates while JavaScript is executing. For example, if you click a button while some JavaScript code is executing, the browser may not queue up the UI update to redraw the button as pressed or any JavaScript initiated by the button. The result is an unresponsive UI that appears to "hang" or "freeze."

更复杂的是有些浏览器在 JavaScript 运行时不将 UI 更新放入队列。例如，如果你在某些 JavaScript 代码运行时点击按钮，浏览器可能不会将重绘按钮按下的 UI 更新任务放入队列，也不会放入由这个按钮启动的 JavaScript 任务。其结果是一个无响应的 UI，表现为“挂起”或“冻结”。

Each browser behaves in roughly the same way. When a script is executing, the UI does not update from user interaction. JavaScript tasks created as a result of user interaction during this time are queued and then executed, in order, when the original JavaScript task has been completed. UI updates caused by user interaction are automatically skipped over at this time because the priority is given to the dynamic aspects of the page. Thus, a button clicked while a script is executing will never look like it was clicked, even though its **onclick** handler will be executed.

每种浏览器的行为大致相同。当脚本执行时，UI 不随用户交互而更新。此时 JavaScript 任务作为用户交互的结果被创建被放入队列，然后当原始 JavaScript 任务完成时队列中的任务被执行。用户交互导致的 UI 更新被自动跳过，因为优先考虑的是页面上的动态部分。因此，当一个脚本运行时点击一个按钮，将看不到它被按下的样子，即使它的 onclick 句柄被执行了。

Even though browsers try to do something logical in these cases, all of these behaviors lead to a disjointed user experience. The best approach, therefore, is to prevent such circumstances from occurring by limiting any JavaScript task to 100 milliseconds or less. This measurement should be taken on the slowest browser you must support (for tools that measure JavaScript performance, see Chapter 10).

尽管浏览器尝试在这些情况下做一些符合逻辑的事情，但所有这些行为导致了一个间断的用户体验。因此最好的方法是，通过限制任何 JavaScript 任务在 100 毫秒或更少时间内完成，避免此类情况出现。这种测量应当在你要支持的最慢的浏览器上执行（关于测量 JavaScript 性能的工具，参见第十章）。

## Yielding with Timers 用定时器让出时间片

Despite your best efforts, there will be times when a JavaScript task cannot be completed in 100 milliseconds or less because of its complexity. In these cases, it's ideal to yield control of the UI thread so that UI updates may occur. Yielding control means stopping JavaScript execution and giving the UI a chance to update itself before continuing to execute the JavaScript. This is where JavaScript timers come into the picture.

尽管你尽了最大努力，还是有一些 JavaScript 任务因为复杂性原因不能在 100 毫秒或更少时间内完成。这种情况下，理想方法是让出对 UI 线程的控制，使 UI 更新可以进行。让出控制意味着停止 JavaScript 运行，给 UI 线程机会进行更新，然后再继续运行 JavaScript。于是 JavaScript 定时器进入了我们的视野。

## Timer Basics　定时器基础

Timers are created in JavaScript using either **setTimeout()** or **setInterval()**, and both accept the same arguments: a function to execute and the amount of time to wait (in milliseconds) before executing it. The **setTimeout()** function creates a timer that executes just once, whereas the **setInterval()** function creates a timer that repeats periodically.

在 JavaScript 中使用 setTimeout()或 setInterval()创建定时器，两个函数都接收一样的参数：一个要执行的函数，和一个运行它之前的等待时间（单位毫秒）。setTimeout()函数创建一个定时器只运行一次，而 setInterval()函数创建一个周期性重复运行的定时器。

The way that timers interact with the UI thread is helpful for breaking up long-running scripts into shorter segments. Calling **setTimeout()** or **setInterval()** tells the JavaScript engine to wait a certain amount of time and then add a JavaScript task to the UI queue. For example:

定时器与 UI 线程交互的方式有助于分解长运行脚本成为较短的片断。调用 setTimeout()或 setInterval()告诉 JavaScript 引擎等待一定时间然后将 JavaScript 任务添加到 UI 队列中。例如：

```
function greeting(){
  alert("Hello world!");
}
setTimeout(greeting, 250);
```

This code inserts a JavaScript task to execute the **greeting()** function into the UI queue after 250 milliseconds have passed. Prior to that point, all other UI updates and JavaScript tasks are executed. Keep in mind that the second argument indicates when the task should be added to the UI queue, which is not necessarily the time that it will be executed; the task must wait until all other tasks already in the queue are executed, just like any other task. Consider the following:

此代码将在 250 毫秒之后，向 UI 队列插入一个 JavaScript 任务运行 greeting()函数。在那个点之前，所有其他 UI 更新和 JavaScript 任务都在运行。请记住，第二个参数指出什么时候应当将任务添加到 UI 队列之中，并不是说那时代码将被执行。这个任务必须等到队列中的其他任务都执行之后才能被执行。考虑下面的例子：

```
var button = document.getElementById("my-button");
button.onclick = function(){
  oneMethod();
  setTimeout(function(){
    document.getElementById("notice").style.color = "red";
  }, 250);
};
```

When the button in this example is clicked, it calls a method and then sets a timer. The code to change the **notice** element's color is contained in a timer set to be queued in 250 milliseconds. That 250 milliseconds starts from the time at which **setTimeout()** is called, not when the overall function has finished executing. So if **setTimeout()** is called at a point in time **n**, then the JavaScript task to execute the timer code is added to the UI queue at **n + 250**. Figure 6-3 shows this relationship when the button in this example is clicked.

在这个例子中当按钮被点击时，它调用一个方法然后设置一个定时器。用于修改 notice 元素颜色的代码被包含在一个定时器设备中，将在 250 毫秒之后添加到队列。250 毫秒从调用 setTimeout()时开始计算，而不是从整个函数运行结束时开始计算。如果 setTimeout()在时间点 n 上被调用，那么运行定时器代码的 JavaScript 任务将在 n+250 的时刻加入 UI 队列。图 6-3 显示出本例中按钮被点击时所发生事件之间的关系。

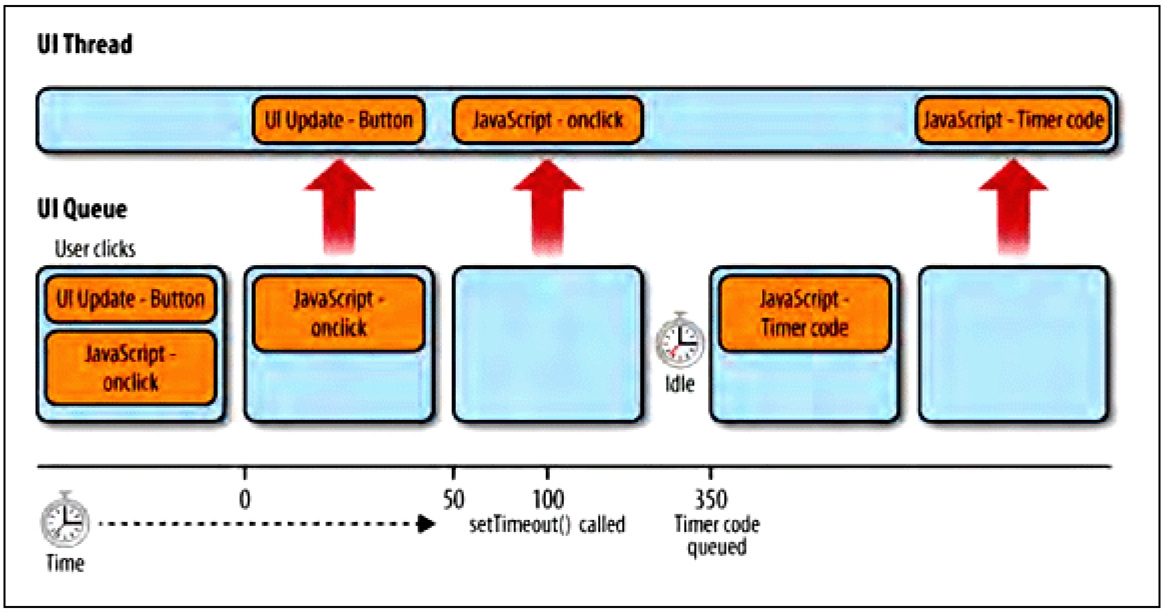


Figure 6-3. The second argument of setTimeout() indicates when the new JavaScript task should be inserted into the UI queue

图 6-3　setTimeout()的第二个参数指出何时将新的 JavaScript 任务插入到 UI 队列中

Keep in mind that the timer code can never be executed until after the function in which it was created is completely executed. For example, if the previous code is changed such that the timer delay is smaller and there is another function call after the timer is created, it's possible that the timer code will be queued before the **onclick** event handler has finished executing:

请记住，定时器代码只有等创建它的函数运行完成之后，才有可能被执行。例如，如果前面的代码中定时器延时变得更小，然后在创建定时器之后又调用了另一个函数，定时器代码有可能在 onclick 事件处理完成之前加入队列：

```
var button = document.getElementById("my-button");
button.onclick = function(){
  oneMethod();
  setTimeout(function(){
    document.getElementById("notice").style.color = "red";
  }, 50);
  anotherMethod();
};
```

If **anotherMethod**() takes longer than 50 milliseconds to execute, then the timer code is added to the queue before the **onclick** handler is finished. The effect is that the timer code executes almost immediately after the **onclick** handler has executed completely, without a noticeable delay. Figure 6-4 illustrates this situation.

如果 anotherMethod()执行时间超过 50 毫秒，那么定时器代码将在 onclick 处理完成之前加入到队列中。其结果是等 onclick 处理运行完毕，定时器代码立即执行，而察觉不出其间的延迟。图 6-4 说明了这种情况。

In either case, creating a timer creates a pause in the UI thread as it switches from one task to the next. Consequently, timer code resets all of the relevant browser limits, including the long-running script timer. Further, the call stack is reset to zero inside of the timer code. These characteristics make timers the ideal cross-browser solution for long-running JavaScript code.

在任何一种情况下，创建一个定时器造成 UI 线程暂停，如同它从一个任务切换到下一个任务。因此，定时器代码复位所有相关的浏览器限制，包括长运行脚本时间。此外，调用栈也在定时器代码中复位为零。这一特性使得定时器成为长运行 JavaScript 代码理想的跨浏览器解决方案。

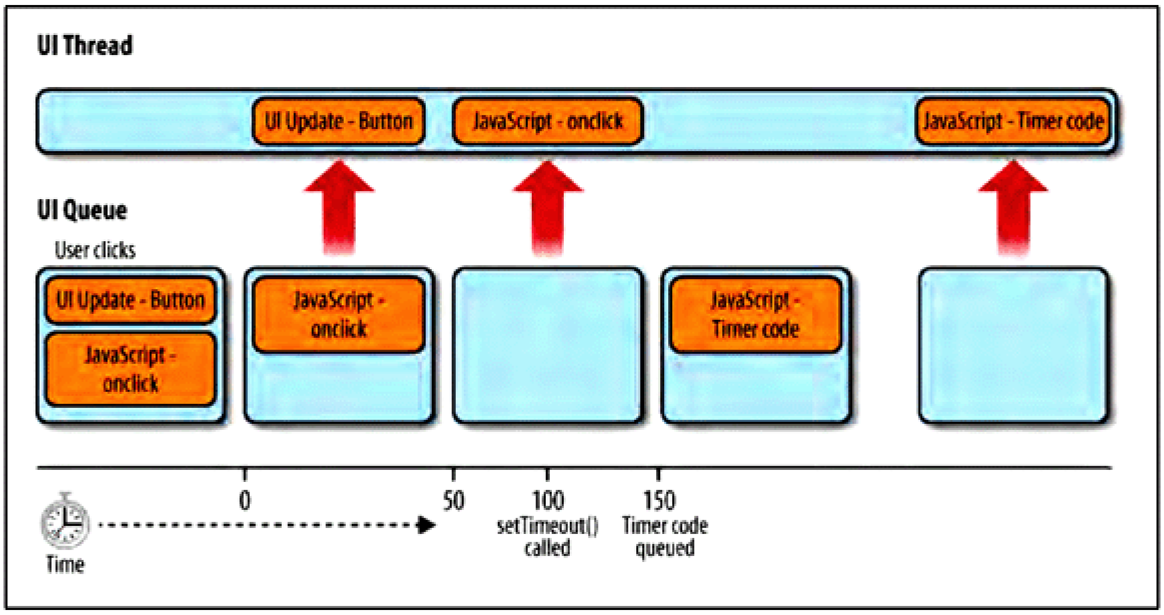


Figure 6-4. There may be no noticeable delay in timer code execution if the function in which setTimeout() is called takes longer to execute than the timer delay

图 6-4 如果调用 setTimeout()的函数又调用了其他任务，耗时超过定时器延时，定时器代码将立即被执行，它与主调函数之间没有可察觉的延迟

## Timer Precision 定时器精度

JavaScript timer delays are often imprecise, with slips of a few milliseconds in either direction. Just because you specify 250 milliseconds as the timer delay doesn't necessarily mean the task is queued exactly 250 milliseconds after **setTimeout()** is called. All browsers make an attempt to be as accurate as possible, but oftentimes a slip of a few milliseconds in either direction occurs. For this reason, timers are unreliable for measuring actual time passed.

JavaScript 定时器延时往往不准确，快慢大约几毫秒。仅仅因为你指定定时器延时 250 毫秒，并不意味任务将在调用 setTimeout()之后精确的 250 毫秒后加入队列。所有浏览器试图尽可能准确，但通常会发生几毫秒滑移，或快或慢。正因为这个原因，定时器不可用于测量实际时间。

Timer resolution on Windows systems is 15 milliseconds, meaning that it will interpret a timer delay of 15 as either 0 or 15, depending on when the system time was last updated. Setting timer delays of less than 15 can cause browser locking in Internet Explorer, so the smallest recommended delay is 25 milliseconds (which will end up as either 15 or 30) to ensure a delay of at least 15 milliseconds.

在 Windows 系统上定时器分辨率为 15 毫秒，也就是说一个值为 15 的定时器延时将根据最后一次系统时间刷新而转换为 0 或者 15。设置定时器延时小于 15 将在 Internet Explorer 中导致浏览器锁定，所以最小值建议为 25 毫秒（实际时间是 15 或 30）以确保至少 15 毫秒延迟。

This minimum timer delay also helps to avoid timer resolution issues in other browsers and on other systems. Most browsers show some variance in timer delays when dealing with 10 milliseconds or smaller.

此最小定时器延时也有助于避免其他浏览器和其他操作系统上的定时器分辨率问题。大多数浏览器在定时器延时小于 10 毫秒时表现出差异性。

## Array Processing with Timers 在数组处理中使用定时器

One common cause of long-running scripts is loops that take too long to execute. If you've already tried the loop optimization techniques presented in Chapter 4 but haven't been able to reduce the execution time enough, then timers are your next optimization step. The basic approach is to split up the loop's work into a series of timers.

一个常见的长运行脚本就是循环占用了太长的运行时间。如果你已经尝试了第四章介绍的循环优化技术，但还是不能缩减足够的运行时间，那么定时器就是你的下一个优化步骤。其基本方法是将循环工作分解到定时器序列中。

Typical loops follow a simple pattern, such as:

典型的循环模式如下：

```
for (var i=0, len=items.length; i < len; i++){
  process(items[i]);
}
```

Loops with this structure can take too long to execute due to the complexity of **process()**, the size of **items**, or both. In my book Professional JavaScript for Web Developers, Second Edition (Wrox 2009), I lay out the two determining factors for whether a loop can be done asynchronously using timers:

这样的循环结构运行时间过长的原因有二，process()的复杂度，items 的大小，或两者兼有。在我的藏书《Professional JavaScript for Web Developers》第二版（Wrox 2009）中，列举了是否可用定时器取代循环的两个决定性因素：

• Does the processing have to be done synchronously?

此处理过程必须是同步处理吗？

• Does the data have to be processed sequentially?

数据必须按顺序处理吗？

If the answer to both of these questions is "no," then the code is a good candidate for using timers to split up the work. A basic pattern for asynchronous code execution is:

如果这两个回答都是“否”，那么代码将适于使用定时器分解工作。一种基本异步代码模式如下：

```
 var todo = items.concat(); //create a clone of the original
setTimeout(function(){
   //get next item in the array and process it
   process(todo.shift());
   //if there's more items to process, create another timer
   if(todo.length > 0){
     setTimeout(arguments.callee, 25);
   } else {
     callback(items);
   }
}, 25);
```

The basic idea of this pattern is to create a clone of the original array and use that as a queue of items to process. The first call to **setTimeout()** creates a timer to process the first item in the array. Calling **todo.shift()** returns the first item and also removes it from the array. This value is passed into **process()**. After processing the item, a check is made to determine whether there are more items to process. If there are still items in the **todo** array, there are more items to process and another timer is created. Because the next timer needs to run the same code as the original, **arguments.callee** is passed in as the first argument. This value points to the anonymous function in which the code is executing. If there are no further items to process, then a **callback()** function is called.

这一模式的基本思想是创建一个原始数组的克隆，将它作为处理对象。第一次调用 setTimeout()创建一个定时器处理队列中的第一个项。调用 todo.shift()返回它的第一个项然后将它从数组中删除。此值作为参数传给 process()。然后，检查是否还有更多项需要处理。如果 todo 队列中还有内容，那么就再启动一个定时器。因为下个定时器需要运行相同的代码，所以第一个参数传入 arguments.callee。此值指向当前正在运行的匿名函数。如果不再有内容需要处理，将调用 callback()函数。

Because this pattern requires significantly more code that a regular loop, it's useful to encapsulate this functionality. For example:

此模式与循环相比需要更多代码，可将此功能封装起来。例如：

```
function processArray(items, process, callback){
  var todo = items.concat(); //create a clone of the original
  setTimeout(function(){
    process(todo.shift());
    if (todo.length > 0){
      setTimeout(arguments.callee, 25);
    } else {
      callback(items);
    }
  }, 25);
}
```

The **processArray()** function implements the previous pattern in a reusable way and accepts three arguments: the array to process, the function to call on each item, and a callback function to execute when processing is complete. This function can be used as follows:

processArray()函数以一种可重用的方式实现了先前的模板，并接收三个参数：待处理数组，对每个项调用的处理函数，处理结束时执行的回调函数。该函数用法如下：

```
var items = [123, 789, 323, 778, 232, 654, 219, 543, 321, 160];
function outputValue(value){
  console.log(value);
}
processArray(items, outputValue, function(){
  console.log("Done!");
});
```

This code uses the **processArray()** method to output array values to the console and then prints a message when all processing is complete. By encapsulating the timer code inside of a function, it can be reused in multiple places without requiring multiple implementations.

此代码使用 processArray()方法将数组值输出到终端，当所有处理结束时再打印一条消息。通过将定时器代码封装在一个函数里，它可在多处重用而无需多次实现。

## Splitting Up Tasks 分解任务

What we typically think of as one task can often be broken down into a series of subtasks. If a single function is taking too long to execute, check to see whether it can be broken down into a series of smaller functions that complete in smaller amounts of time. This is often as simple as considering a single line of code as an atomic task, even though multiple lines of code typically can be grouped together into a single task. Some functions are already easily broken down based on the other functions they call. For example:

我们通常将一个任务分解成一系列子任务。如果一个函数运行时间太长，那么查看它是否可以分解成一系列能够短时间完成的较小的函数。可将一行代码简单地看作一个原子任务，多行代码组合在一起构成一个独立任务。某些函数可基于函数调用进行拆分。例如：

```
function saveDocument(id){
  //save the document
  openDocument(id)
  writeText(id);
  closeDocument(id);
  //update the UI to indicate success
  updateUI(id);
}
```

If this function is taking too long, it can easily be split up into a series of smaller steps by breaking out the individual methods into separate timers. You can accomplish this by adding each function into an array and then using a pattern similar to the array-processing pattern from the previous section:

如果函数运行时间太长，它可以拆分成一系列更小的步骤，把独立方法放在定时器中调用。你可以将每个函数都放入一个数组，然后使用前一节中提到的数组处理模式：

```
function saveDocument(id){
  var tasks = [openDocument, writeText, closeDocument, updateUI];
  setTimeout(function(){
    //execute the next task
    var task = tasks.shift();
    task(id);
    //determine if there's more
    if (tasks.length > 0){
      setTimeout(arguments.callee, 25);
    }
  }, 25);
}
```

This version of the function places each method into the tasks array and then executes only one method with each timer. Fundamentally, this now becomes an array-processing pattern, with the sole difference that processing an item involves executing the function contained in the item. As discussed in the previous section, this pattern can be encapsulated for reuse:

这个版本将每个方法放入任务数组，然后在每个定时器中调用一个方法。从根本上说，现在它成为数组处理模式，只有一点不同：处理函数就包含在数组项中。正如前面一节所讨论的，此模式也可封装重用：

```
function multistep(steps, args, callback){
  var tasks = steps.concat(); //clone the array
  setTimeout(function(){
    //execute the next task
    var task = tasks.shift();
    task.apply(null, args || []);
    //determine if there's more
    if (tasks.length > 0){
      setTimeout(arguments.callee, 25);
    } else {
      callback();
    }
  }, 25);
}
```

The **multistep()** function accepts three arguments: an array of functions to execute, an array of arguments to pass into each function when it executes, and a callback function to call when the process is complete. This function can be used like the following:

multistep()函数接收三个参数：用于执行的函数数组，为每个函数提供参数的参数数组，当处理结束时调用的回调函数。函数用法如下：

```
function saveDocument(id){
  var tasks = [openDocument, writeText, closeDocument, updateUI];
  multistep(tasks, [id], function(){
    alert("Save completed!");
  });
}
```

Note that the second argument to **multistep()** must be an array, so one is created containing just **id**. As with array processing, this function is best used when the tasks can be processed asynchronously without affecting the user experience or causing errors in dependent code.

注意传给 multistep()的第二个参数必须是数组，它创建时只包含一个 id。正如数组处理那样，使用此函数的前提条件是：任务可以异步处理而不影响用户体验或导致依赖代码出错。

## Timed Code 限时运行代码

Sometimes executing just one task at a time is inefficient. Consider processing an array of 1,000 items for which processing a single item takes 1 millisecond. If one item is processed in each timer and there is a delay of 25 milliseconds in between, that means the total amount of time to process the array is (25 + 1) × 1,000 = 26,000 milliseconds, or 26 seconds. What if you processed the items in batches of 50 with a 25-millisecond delay between them? The entire processing time then becomes (1,000 / 50) × 25 + 1,000 = 1,500 milliseconds, or 1.5 seconds, and the user is still never blocked from the interface because the longest the script has executed continuously is 50 milliseconds. It's typically faster to process items in batches than one at a time.

有时每次只执行一个任务效率不高。考虑这样一种情况：处理一个拥有 1'000 个项的数组，每处理一个项需要 1 毫秒。如果每个定时器中处理一个项，在两次处理之间间隔 25 毫秒，那么处理此数组的总时间是(25 + 1) × 1'000 = 26'000 毫秒，也就是 26 秒。如果每批处理 50 个，每批之间间隔 25 毫秒会怎么样呢？整个处理过程变成(1'000 / 50) × 25 + 1'000 = 1'500 毫秒，也就是 1.5 秒，而且用户也不会察觉界面阻塞，因为最长的脚本运行只持续了 50 毫秒。通常批量处理比每次处理一个更快。

If you keep 100 milliseconds in mind as the absolute maximum amount of time that JavaScript should be allowed to run continuously, then you can start optimizing the previous patterns. My recommendation is to cut that number in half and never let any JavaScript code execute for longer than 50 milliseconds continuously, just to make sure the code never gets close to affecting the user experience.

如果你记住 JavaScript 可连续运行的最大时间是 100 毫秒，那么你可以优化先前的模式。我的建议是将这个数字削减一半，不要让任何 JavaScript 代码持续运行超过 50 毫秒，只是为了确保代码永远不会影响用户体验。

It's possible to track how long a piece of code has been running by using the native **Date** object. This is the way most JavaScript profiling works:

可通过原生的 Date 对象跟踪代码的运行时间。这是大多数 JavaScript 分析工具所采用的工作方式：

```
 var start = +new Date(),
stop;
someLongProcess();
stop = +new Date();
```

```
if(stop-start < 50){
  alert("Just about right.");
} else {
  alert("Taking too long.");
}
```

Since each new **Date** object is initialized with the current system time, you can time code by creating new **Date** objects periodically and comparing their values. The plus operator (+) converts the **Date** object into a numeric representation so that any further arithmetic doesn't involve conversions. This same basic technique can be used to optimize the previous timer patterns.

由于每个新创建的 Data 对象以当前系统时间初始化，你可以周期性地创建新 Data 对象并比较它们的值，以获取代码运行时间。加号（+）将 Data 对象转换为一个数字，在后续的数学运算中就不必再转换了。这一技术也可用于优化以前的定时器模板。

The **processArray()** method can be augmented to process multiple items per timer by adding in a time check:

processArray()方法通过一个时间检测机制，可在每个定时器中执行多次处理：

```
function timedProcessArray(items, process, callback){
  var todo = items.concat(); //create a clone of the original
  setTimeout(function(){
    var start = +new Date();
    do {
      process(todo.shift());
    } while (todo.length > 0 && (+new Date() - start < 50));
    if (todo.length > 0){
      setTimeout(arguments.callee, 25);
    } else {
      callback(items);
    }
    }, 25);
}
```

The addition of a **do-while** loop in this function enables checking the time after each item is processed. The array will always contain at least one item when the timer function executes, so a post-test loop makes more sense than a pretest one. When run in Firefox 3, this function processes an array of 1,000 items, where **process()** is an empty function, in 38–43 milliseconds; the original **processArray()** function processes the same array in over 25,000 milliseconds. This is the power of timing tasks before breaking them up into smaller chunks.

此函数中添加了一个 do-while 循环，它在每个数组项处理之后检测时间。定时器函数运行时数组中存放了至少一个项，所以后测试循环比前测试更合理。在 Firefox 3 中，如果 process()是一个空函数，处理一个 1'000 个项的数组需要 38 - 34 毫秒；原始的 processArray()函数处理同一个数组需要超过 25'000 毫秒。这就是定时任务的作用，避免将任务分解成过于碎小的片断。

## Timers and Performance 定时器与性能

Timers can make a huge difference in the overall performance of your JavaScript code, but overusing them can have a negative effect on performance. The code in this section has used sequenced timers such that only one timer exists at a time and new ones are created only when the last timer has finished. Using timers in this way will not result in performance issues.

定时器使你的 JavaScript 代码整体性能表现出巨大差异，但过度使用它们会对性能产生负面影响。本节中的代码使用定时器序列，同一时间只有一个定时器存在，只有当这个定时器结束时才创建一个新的定时器。以这种方式使用定时器不会带来性能问题。

Performance issues start to appear when multiple repeating timers are being created at the same time. Since there is only one UI thread, all of the timers compete for time to execute. Neil Thomas of Google Mobile researched this topic as a way of measuring performance on the mobile Gmail application for the iPhone and Android.

当多个重复的定时器被同时创建会产生性能问题。因为只有一个 UI 线程，所有定时器竞争运行时间。Google Mobile 的 Neil Thomas 将此问题作为测量性能的方法进行研究，针对 iPhone 和 Android 上运行的移动 Gmail 程序。

Thomas found that low-frequency repeating timers—those occurring at intervals of one second or greater—had little effect on overall web application responsiveness. The timer delays in this case are too large to create a bottleneck on the UI thread and are therefore safe to use repeatedly. When multiple repeating timers are used with a much greater frequency (between 100 and 200 milliseconds), however, Thomas found that the mobile Gmail application became noticeably slower and less responsive.

Thomas 发现低频率的重复定时器——间隔在 1 秒或 1 秒以上——几乎不影响整个网页应用的响应。这种情况下定时器延迟远超过使 UI 线程产生瓶颈的值，因此可安全地重复使用。当多个重复定时器使用更高的频率（间隔在 100 到 200 毫秒之间），Thomas 发现移动 Gmail 程序明显变慢，反应较差。

The takeaway from Thomas's research is to limit the number of high-frequency repeating timers in your web application. Instead, Thomas suggests creating a single repeating timer that performs multiple operations with each execution.

Thomas 研究的言外之意是，要在你的网页应用中限制高频率重复定时器的数量。同时，Thomas 建议创建一个单独的重复定时器，每次执行多个操作。

## Web Workers 网页工人线程

Since JavaScript was introduced, there has been no way to execute code outside of the browser UI thread. The web workers API changes this by introducing an interface through which code can be executed without taking time on the browser UI thread. Originally part of HTML 5, the web workers API has been split out into its own specification (http://www.w3.org/TR/workers/); web workers have already been implemented natively in Firefox 3.5, Chrome 3, and Safari 4.

自 JavaScript 诞生以来，还没有办法在浏览器 UI 线程之外运行代码。网页工人线程 API 改变了这种状况，它引入一个接口，使代码运行而不占用浏览器 UI 线程的时间。作为最初的 HTML 5 的一部分，网页

工人线程 API 已经分离出去成为独立的规范（http://www.w3.org/TR/workers/）。网页工人线程已经被 Firefox 3.5，Chrome 3，和 Safari 4 原生实现。

Web workers represent a potentially huge performance improvement for web applications because each new worker spawns its own thread in which to execute JavaScript. That means not only will code executing in a worker not affect the browser UI, but it also won't affect code executing in other workers.

网页工人线程对网页应用来说是一个潜在的巨大性能提升，因为新的工人线程在自己的线程中运行 JavaScript。这意味着，工人线程中的代码运行不仅不会影响浏览器 UI，而且也不会影响其它工人线程中运行的代码。

**Worker Environment** 工人线程运行环境

Since web workers aren't bound to the UI thread, it also means that they cannot access a lot of browser resources. Part of the reason that JavaScript and UI updates share the same process is because one can affect the other quite frequently, and so executing these tasks out of order results in a bad user experience. Web workers could introduce user interface errors by making changes to the DOM from an outside thread, but each web worker has its own global environment that has only a subset of JavaScript features available. The worker environment is made up of the following:

由于网页工人线程不绑定 UI 线程，这也意味着它们将不能访问许多浏览器资源。JavaScript 和 UI 更新共享同一个进程的部分原因是它们之间互访频繁，如果这些任务失控将导致糟糕的用户体验。网页工人线程修改 DOM 将导致用户界面出错，但每个网页工人线程都有自己的全局运行环境，只有 JavaScript 特性的一个子集可用。工人线程的运行环境由下列部分组成：

• A navigator object, which contains only four properties: **appName**, **appVersion**, **userAgent**, and **platform**

一个浏览器对象，只包含四个属性：appName, appVersion, userAgent, 和 platform

• A **location** object (same as on **window**, except all properties are read-only)

一个 location 对象（和 window 里的一样，只是所有属性都是只读的）

• A **self** object that points to the global worker object

一个 self 对象指向全局工人线程对象

• An **importScripts()** method that is used to load external JavaScript for use in the worker

一个 importScripts()方法，使工人线程可以加载外部 JavaScript 文件

• All ECMAScript objects, such as **Object**, **Array**, **Date**, etc.

所有 ECMAScript 对象，诸如 Object，Array，Data，等等。

• The **XMLHttpRequest** constructor

XMLHttpRequest 构造器

• The **setTimeout()** and **setInterval()** methods

setTimeout()和 setInterval()方法

• A **close()** method that stops the worker immediately

close()方法可立即停止工人线程

Because web workers have a different global environment, you can't create one from any JavaScript code. In fact, you'll need to create an entirely separate JavaScript file containing just the code for the worker to execute. To create a web worker, you must pass in the URL for the JavaScript file:

因为网页工人线程有不同的全局运行环境，你不能在 JavaScript 代码中创建。事实上，你需要创建一个完全独立的 JavaScript 文件，包含那些在工人线程中运行的代码。要创建网页工人线程，你必须传入这个 JavaScript 文件的 URL：

```
var worker = new Worker("code.js");
```

Once this is executed, a new thread with a new worker environment is created for the specified file. This file is downloaded asynchronously, and the worker will not begin until the file has been completely downloaded and executed.

此代码一旦执行，将为指定文件创建一个新线程和一个新的工人线程运行环境。此文件被异步下载，直到下载并运行完之后才启动工人线程。

**Worker Communication** 工人线程交互

Communication between a worker and the web page code is established through an event interface. The web page code can pass data to the worker via the **postMessage()** method, which accepts a single argument indicating the data to pass into the worker.There is also an **onmessage** event handler that is used to receive information from the worker. For example:

工人线程和网页代码通过事件接口进行交互。网页代码可通过 postMessage()方法向工人线程传递数据，它接收单个参数，即传递给工人线程的数据。此外，在工人线程中还有 onmessage 事件句柄用于接收信息。例如：

```
var worker = new Worker("code.js");
worker.onmessage = function(event){
  alert(event.data);
};
worker.postMessage("Nicholas");
```

The worker receives this data through the firing of a **message** event. An **onmessage** event handler is defined, and the event object has a **data** property containing the data that was passed in. The worker can then pass information back to the web page by using its own **postMessage()** method:

工人线程从 message 事件中接收数据。这里定义了一个 onmessage 事件句柄，事件对象具有一个 data 属性存放传入的数据。工人线程可通过它自己的 postMessage()方法将信息返回给页面。

```
//inside code.js
self.onmessage = function(event){
  self.postMessage("Hello, " + event.data + "!");
};
```

The final string ends up in the **onmessage** event handler for the worker. This messaging system is the only way in which the web page and the worker can communicate.

最终的字符串结束于工人线程的 onmessage 事件句柄。消息系统是页面和工人线程之间唯一的交互途径。

Only certain types of data can be passed using **postMessage()**. You can pass primitive values (**strings**, **numbers**, **Booleans**, **null**, and **undefined**) as well as instances of **Object** and **Array**; you cannot pass any other data types. Valid data is serialized, transmitted to or from the worker, and then deserialized. Even though it seems like the objects are being passed through directly, the instances are completely separate representations of the same data. Attempting to pass an unsupported data type results in a JavaScript error.

只有某些类型的数据可以使用 postMessage()传递。你可以传递原始值（string，number，boolean，null 和 undefined），也可以传递 Object 和 Array 的实例，其它类型就不允许了。有效数据被序列化，传入或传出工人线程，然后反序列化。即使看上去对象直接传了过去，实例其实是同一个数据完全独立的表述。试图传递一个不支持的数据类型将导致 JavaScript 错误。

**Loading External Files** 加载外部文件

Loading extra JavaScript files into a worker is done via the **importScripts()** method, which accepts one or more URLs for JavaScript files to load. The call to **importScripts()** is blocking within the worker, so the script won't continue until all files have been loaded and executed. Since the worker is running outside of the UI thread, there is no concern about UI responsiveness when this blocking occurs. For example:

当工人线程通过 importScripts()方法加载外部 JavaScript 文件，它接收一个或多个 URL 参数，指出要加载的 JavaScript 文件网址。工人线程以阻塞方式调用 importScripts()，直到所有文件加载完成并执行之后，脚本才继续运行。由于工人线程在 UI 线程之外运行，这种阻塞不会影响 UI 响应。例如：

```
//inside code.js
importScripts("file1.js", "file2.js");
self.onmessage = function(event){
  self.postMessage("Hello, " + event.data + "!");
};
```

The first line in this code includes two JavaScript files so that they will be available in the context of the worker.

此代码第一行包含两个 JavaScript 文件，它们将在工人线程中使用。

**Practical Uses** 实际用途

Web workers are suitable for any long-running scripts that work on pure data and that have no ties to the browser UI. This may seem like a fairly small number of uses, but buried in web applications there are typically some data-handling approaches that would benefit from using a worker instead of timers.

网页工人线程适合于那些纯数据的，或者与浏览器 UI 没关系的长运行脚本。它看起来用处不大，而网页应用程序中通常有一些数据处理功能将受益于工人线程，而不是定时器。

Consider, for example, parsing a large JSON string (JSON parsing is discussed further in Chapter 7). Suppose that the data is large enough that parsing takes at least 500 milliseconds. That is clearly too long to allow JavaScript to run on the client, as it will interfere with the user experience. This particular task is difficult to break into small chunks with timers, so a worker is the ideal solution. The following code illustrates usage from a web page:

考虑这样一个例子，解析一个很大的 JSON 字符串（JSON 解析将在后面第七章讨论）。假设数据足够大，至少需要 500 毫秒才能完成解析任务。很显然时间太长了以至于不能允许 JavaScript 在客户端上运行它，因为它会干扰用户体验。此任务难以分解成用于定时器的小段任务，所以工人线程成为理想的解决方案。下面的代码说明了它在网页上的应用：

```
var worker = new Worker("jsonparser.js");
//when the data is available, this event handler is called
worker.onmessage = function(event){
  //the JSON structure is passed back
  var jsonData = event.data;
  //the JSON structure is used
  evaluateData(jsonData);
};
//pass in the large JSON string to parse
worker.postMessage(jsonText);
```

The code for the worker responsible for JSON parsing is as follows:

工人线程的代码负责 JSON 解析，如下：

```
 //inside of jsonparser.js
//this event handler is called when JSON data is available
self.onmessage = function(event){
  //the JSON string comes in as event.data
  var jsonText = event.data;
  //parse the structure
  var jsonData = JSON.parse(jsonText);
  //send back to the results
  self.postMessage(jsonData);
};
```

Note that even though **JSON.parse()** is likely to take 500 milliseconds or more, there is no need to write any additional code to split up the processing. This execution takes place on a separate thread, so you can let it run for as long as the parsing takes without interfering with the user experience.

请注意，即使 JSON.parse()可能需要 500 毫秒或更多时间，也没有必要添加更多代码来分解处理过程。此处理过程发生在一个独立的线程中，所以你可以让它一直运行完解析过程而不会干扰用户体验。

The page passes a JSON string into the worker by using **postMessage()**. The worker receives the string as **event.data** in its **onmessage** event handler and then proceeds to parse it. When complete, the resulting JSON object is passed back to the page using the worker's **postMessage()** method. This object is then available as **event.data** in the page's **onmessage** event handler. Keep in mind that this presently works only in Firefox 3.5 and later, as Safari 4 and Chrome 3's implementations allow strings to be passed only between page and worker.

页面使用 postMessage()将一个 JSON 字符串传给工人线程。工人线程在它的 onmessage 事件句柄中收到这个字符串也就是 event.data，然后开始解析它。完成时所产生的 JSON 对象通过工人线程的 postMessage()方法传回页面。然后此对象便成为页面 onmessage 事件句柄的 event.data。请记住，此工程只能在 Firefox 3.5 和更高版本中运行，而 Safari 4 和 Chrome 3 中，页面和工人线程之间只允许传递字符串。

Parsing a large string is just one of many possible tasks that can benefit from web workers. Some other possibilities are:

解析一个大字符串只是许多受益于网页工人线程的任务之一。其它可能受益的任务如下：

• Encoding/decoding a large string

编/解码一个大字符串

• Complex mathematical calculations (including image or video processing)

复杂数学运算（包括图像或视频处理）

• Sorting a large array

给一个大数组排序

Any time a process takes longer than 100 milliseconds to complete, you should consider whether a worker solution is more appropriate than a timer-based one. This, of course, is based on browser capabilities.

任何超过 100 毫秒的处理，都应当考虑工人线程方案是不是比基于定时器的方案更合适。当然，还要基于浏览器是否支持工人线程。

## Summary 总结

JavaScript and user interface updates operate within the same process, so only one can be done at a time. This means that the user interface cannot react to input while JavaScript code is executing and vice versa. Managing the UI thread effectively means ensuring that JavaScript isn't allowed to run so long that the user experience is affected. To that end, the following should be kept in mind:

JavaScript 和用户界面更新在同一个进程内运行，同一时刻只有其中一个可以运行。这意味着当 JavaScript 代码正在运行时，用户界面不能响应输入，反之亦然。有效地管理 UI 线程就是要确保 JavaScript 不能运行太长时间，以免影响用户体验。最后，请牢记如下几点：

• No JavaScript task should take longer than 100 milliseconds to execute. Longer execution times cause a noticeable delay in updates to the UI and negatively impact the overall user experience.

JavaScript 运行时间不应该超过 100 毫秒。过长的运行时间导致 UI 更新出现可察觉的延迟，从而对整体用户体验产生负面影响。

• Browsers behave differently in response to user interaction during JavaScript execution.
Regardless of the behavior, the user experience becomes confusing and disjointed when JavaScript takes a long time to execute.

JavaScript 运行期间，浏览器响应用户交互的行为存在差异。无论如何，JavaScript 长时间运行将导致用户体验混乱和脱节。

• Timers can be used to schedule code for later execution, which allows you to split up long-running scripts into a series of smaller tasks.

定时器可用于安排代码推迟执行，它使得你可以将长运行脚本分解成一系列较小的任务。

• Web workers are a feature in newer browsers that allow you to execute JavaScript code outside of the UI thread, thus preventing UI locking.

网页工人线程是新式浏览器才支持的特性，它允许你在 UI 线程之外运行 JavaScript 代码而避免锁定 UI。

The more complex the web application, the more critical it is to manage the UI thread in a proactive manner. No JavaScript code is so important that it should adversely affect the user's experience.

网页应用程序越复杂，积极主动地管理 UI 线程就越显得重要。没有什么 JavaScript 代码可以重要到允许影响用户体验的程度。

# 第七章 Ajax 异步 JavaScript 和 XML

Ajax is a cornerstone of high-performance JavaScript. It can be used to make a page load faster by delaying the download of large resources. It can prevent page loads altogether by allowing for data to be transferred between the client and the server asynchronously. It can even be used to fetch all of a page's resources in one HTTP request. By choosing the correct transmission technique and the most efficient data format, you can significantly improve how your users interact with your site.

Ajax 是高性能 JavaScript 的基石。它可以通过延迟下载大量资源使页面加载更快。它通过在客户端和服务器之间异步传送数据，避免页面集体加载。它还用于在一次 HTTP 请求中获取整个页面的资源。通过选择正确的传输技术和最有效的数据格式，你可以显著改善用户与网站之间的互动。

This chapter examines the fastest techniques for sending data to and receiving it from the server, as well as the most efficient formats for encoding data.

本章考察从服务器收发数据最快的技术，以及最有效的数据编码格式。

## Data Transmission 数据传输

Ajax, at its most basic level, is a way of communicating with a server without unloading the current page; data can be requested from the server or sent to it. There are several different ways of setting up this communication channel, each with its own advantages and restrictions. This section briefly examines the different approaches and discusses the performance implications of each.

Ajax，在它最基本的层面，是一种与服务器通讯而不重载当前页面的方法，数据可从服务器获得或发送给服务器。有多种不同的方法构造这种通讯通道，每种方法都有自己的优势和限制。本节简要地介绍这些不同方法，并讨论各自对性能的影响。

### Requesting Data 请求数据

There are five general techniques for requesting data from a server:

有五种常用技术用于向服务器请求数据：

• XMLHttpRequest (XHR)

• Dynamic script tag insertion　动态脚本标签插入

• iframes

• Comet

• Multipart XHR　　　　　多部分的 XHR

The three that are used in modern high-performance JavaScript are XHR, dynamic script tag insertion, and multipart XHR. Use of Comet and iframes (as data transport techniques) tends to be extremely situational, and won't be covered here.

在现代高性能 JavaScript 中使用的三种技术是 XHR，动态脚本标签插入和多部分的 XHR。使用 Comet 和 iframe（作为数据传输技术）往往是极限情况，不在这里讨论。

**XMLHttpRequest**

By far the most common technique used, XMLHttpRequest (XHR) allows you to asynchronously send and receive data. It is well supported across all modern browsers and allows for a fine degree of control over both the request sent and the data received. You can add arbitrary headers and parameters (both GET and POST) to the request, and read all of the headers returned from the server, as well as the response text itself. The following is an example of how it can be used:

目前最常用的方法中，XMLHttpRequest（XHR）用来异步收发数据。所有现代浏览器都能够很好地支持它，而且能够精细地控制发送请求和数据接收。你可以向请求报文中添加任意的头信息和参数（包括 GET 和 POST），并读取从服务器返回的头信息，以及响应文本自身。以下是使用示例：

```
var url = '/data.php';
var params = [
  'id=934875',
  'limit=20'
];
var req = new XMLHttpRequest();
req.onreadystatechange = function() {
  if (req.readyState === 4) {
    var responseHeaders = req.getAllResponseHeaders(); // Get the response headers.
    var data = req.responseText; // Get the data.
    // Process the data here...
  }
}
req.open('GET', url + '?' + params.join('&'), true);
req.setRequestHeader('X-Requested-With', 'XMLHttpRequest'); // Set a request header.
req.send(null); // Send the request.
```

This example shows how to request data from a URL, with parameters, and how to read the response text and headers. A **readyState** of 4 indicates that the entire response has been received and is available for manipulation.

此例显示了如何从 URL 请求数据，使用参数，以及如何读取响应报文和头信息。readyState 等于 4 表示整个响应报文已经收并完可用于操作。

It is possible to interact with the server response as it is still being transferred by listening for **readyState** 3. This is known as streaming, and it is a powerful tool for improving the performance of your data requests:

readyState 等于 3 则表示此时正在与服务器交互，响应报文还在传输之中。这就是所谓的"流"，它是提高数据请求性能的强大工具：

```
req.onreadystatechange = function() {
  if (req.readyState === 3) { // Some, but not all, data has been received.
    var dataSoFar = req.responseText;
    ...
  }
  else if (req.readyState === 4) { // All data has been received.
    var data = req.responseText;
    ...
  }
}
```

Because of the high degree of control that XHR offers, browsers place some restrictions on it. You cannot use XHR to request data from a domain different from the one the code is currently running under, and older versions of IE do not give you access to **readyState** 3, which prevents streaming. Data that comes back from the request is treated as either a string or an XML object; this means large amounts of data will be quite slow to process.

由于 XHR 提供了高级别的控制，浏览器在上面增加了一些限制。你不能使用 XHR 从当前运行的代码域之外请求数据，而且老版本的 IE 也不提供 readyState 3，它不支持流。从请求返回的数据像一个字符串或者一个 XML 对象那样对待，这意味着处理大量数据将相当缓慢。

Despite these drawbacks, XHR is the most commonly used technique for requesting data and is still the most powerful. It should be the one you look to first.

尽管有这些缺点，XHR 仍旧是最常用的请求数据技术，也是最强大的。它应当成为你的首选。

**POST versus GET when using XHR.　使用 XHR 时，应使用 POST 还是 GET**

When using XHR to request data, you have a choice between using POST or GET. For requests that don't change the server state and only pull back data (this is called an idempotent action), use GET. GET requests are cached, which can improve performance if you're fetching the same data several times.

当使用 XHR 请求数据时，你可以选择 POST 或 GET。如果请求不改变服务器状态只是取回数据（又称作幂等动作）则使用 GET。GET 请求被缓冲起来，如果你多次提取相同的数据可提高性能。

POST should be used to fetch data only when the length of the URL and the parameters are close to or exceed 2,048 characters. This is because Internet Explorer limits URLs to that length, and exceeding it will cause your request to be truncated.

只有当 URL 和参数的长度超过了 2'048 个字符时才使用 POST 提取数据。因为 Internet Explorer 限制 URL 的长度，过长将导致请求（参数）被截断。

**Dynamic script tag insertion　动态脚本标签插入**

This technique overcomes the biggest limitation of XHR: it can request data from a server on a different domain. It is a hack; instead of instantiating a purpose-built object, you use JavaScript to create a new script tag and set its source attribute to a URL in a different domain.

该技术克服了 XHR 的最大限制：它可以从不同域的服务器上获取数据。这是一种黑客技术，而不是实例化一个专用对象，你用 JavaScript 创建了一个新脚本标签，并将它的源属性设置为一个指向不同域的 URL。

```
 var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName_r('head')[0].appendChild(scriptElement);
```

But dynamic script tag insertion offers much less control than XHR. You can't send headers with the request. Parameters can only be passed using GET, not POST. You can't set timeouts or retry the request; in fact, you won't necessarily know if it fails. You must wait for all of the data to be returned before you can access any of it. You don't have access to the response headers or to the entire response as a string.

但是动态脚本标签插入与 XHR 相比只提供更少的控制。你不能通过请求发送信息头。参数只能通过 GET 方法传递，不能用 POST。你不能设置请求的超时或重试，实际上，你不需要知道它是否失败了。你必须等待所有数据返回之后才可以访问它们。你不能访问响应信息头或者像访问字符串那样访问整个响应报文。

This last point is especially important. Because the response is being used as the source for a script tag, it must be executable JavaScript. You cannot use bare XML, or even bare JSON; any data, regardless of the format, must be enclosed in a callback function.

最后一点非常重要。因为响应报文被用作脚本标签的源码，它必须是可执行的 JavaScript。你不能使用裸 XML，或者裸 JSON，任何数据，无论什么格式，必须在一个回调函数之中被组装起来。

```
 var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName_r('head')[0].appendChild(scriptElement);
```

```
function jsonCallback(jsonString) {
  var data = ('(' + jsonString + ')');
  // Process the data here...
}
```

In this example, the **lib.js** file would enclose the data in the **jsonCallback** function:

在这个例子中，lib.js 文件将调用 jsonCallback 函数组装数据：

```
jsonCallback({ "status": 1, "colors": [ "#fff", "#000", "#ff0000" ] });
```

Despite these limitations, this technique can be extremely fast. The response is executed as JavaScript; it is not treated as a string that must be further processed. Because of this, it has the potential to be the fastest way of getting data and parsing it into something you can access on the client side. We compare the performance of dynamic script tag insertion with the performance of XHR in the section on JSON, later in this chapter.

尽管有这些限制，此技术仍然非常迅速。其响应结果是运行 JavaScript，而不是作为字符串必须被进一步处理。正因为如此，它可能是客户端上获取并解析数据最快的方法。我们比较了动态脚本标签插入和 XHR 的性能，在本章后面 JSON 一节中。

Beware of using this technique to request data from a server you don't directly control. JavaScript has no concept of permission or access control, so any code that you incorporate into your page using dynamic script tag insertion will have complete control over the page. This includes the ability to modify any content, redirect users to another site, or even track their actions on this page and send the data back to a third party. Use extreme caution when pulling in code from an external source.

请小心使用这种技术从你不能直接控制的服务器上请求数据。JavaScript 没有权限或访问控制的概念，所以你的页面上任何使用动态脚本标签插入的代码都可以完全控制整个页面。包括修改任何内容、将用户重定向到另一个站点，或跟踪他们在页面上的操作并将数据发送给第三方。使用外部来源的代码时务必非常小心。

## Multipart XHR  多部分 XHR

The newest of the techniques mentioned here, multipart XHR (MXHR) allows you to pass multiple resources from the server side to the client side using only one HTTP request. This is done by packaging up the resources (whether they be CSS files, HTML fragments, JavaScript code, or base64 encoded images) on the server side and sending them to the client as a long string of characters, separated by some agreed-upon string. The JavaScript code processes this long string and parses each resource according to its mime-type and any other "header" passed with it.

这里介绍最新的技术，多部分 XHR（MXHR）允许你只用一个 HTTP 请求就可以从服务器端获取多个资源。它通过将资源（可以是 CSS 文件，HTML 片段，JavaScript 代码，或 base64 编码的图片）打包成一个由特定分隔符界定的大字符串，从服务器端发送到客户端。JavaScript 代码处理此长字符串，根据它的媒体类型和其他“信息头”解析出每个资源。

Let's follow this process from start to finish. First, a request is made to the server for several image resources:

让我们从头到尾跟随这个过程。首先，发送一个请求向服务器索取几个图像资源：

```
var req = new XMLHttpRequest();
req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = function() {
  if (req.readyState == 4) {
    splitImages(req.responseText);
  }
};
req.send(null);
```

This is a very simple request. You are asking for data from **rollup_images.php**, and once you receive it, you send it to the function **splitImages**.

这是一个非常简单的请求。你向 rollup_images.php 要求数据，一旦你收到返回结果，就将它交给函数 splitImages 处理。

Next, on the server, the images are read and converted into strings:

下一步，服务器读取图片并将它们转换为字符串：

```
// Read the images and convert them into base64 encoded strings.
$images = array('kitten.jpg', 'sunset.jpg', 'baby.jpg');
foreach ($images as $image) {
  $image_fh = fopen($image, 'r');
  $image_data = fread($image_fh, filesize($image));
  fclose($image_fh);
    $payloads[] = base64_encode($image_data);
  }（译者注：疑原文有误，此括号多余）
}
// Roll up those strings into one long string and output it.
$newline = chr(1); // This character won't appear naturally in any base64 string.
echo implode($newline, $payloads);
```

This piece of PHP code reads three images and converts them into long strings of base64 characters. They are concatenated using a single character, Unicode character 1, and output back to the client.

这段 PHP 代码读取三个图片，并将它们转换成 base64 字符串。它们之间用一个简单的字符，UNICODE 的 1，连接起来，然后返回给客户端。

Once on the client side, the data is processed by the **splitImages** function:

然后回到客户端，此数据由 splitImage 函数处理：

```
function splitImages(imageString) {
  var imageData = imageString.split("\u0001");
  var imageElement;
  for (var i = 0, len = imageData.length; i < len; i++) {
    imageElement = document.createElement('img');
    imageElement.src = 'data:image/jpeg;base64,' + imageData[i];
    document.getElementById('container').appendChild(imageElement);
    }
}
```

This function takes the concatenated string and splits it up again into three pieces. Each piece is then used to create an image element, and that image element is inserted into the page. The image is not converted from a base64 string back to binary data; instead it is passed to the image element using a **data: URL** and the **image/jpeg** mime-type.

此函数将拼接而成的字符串分解为三段。每段用于创建一个图像元素，然后将图像元素插入页面中。图像不是从 base64 转换成二进制，而是使用 data:URL 并指定 image/jpeg 媒体类型。

The end result is that three images have been passed to the browser as a single HTTP request. This could be done with 20 images or 100; the response would be larger, but it would still take only one HTTP request. It can also be expanded to other types of resources. JavaScript files, CSS files, HTML fragments, and images of many types can all be combined into one response. Any data type that can be handled as a string by JavaScript can be sent. Here are functions that will take strings for JavaScript code, CSS styles, and images and convert them into resources the browser can use:

最终结果是：在一次 HTTP 请求中向浏览器传入了三张图片。也可以传入 20 张或 100 张，响应报文会更大，但也只是一次 HTTP 请求。它也可以扩展至其他类型的资源。JavaScript 文件，CSS 文件，HTML 片段，许多类型的图片都可以合并成一次响应。任何数据类型都可作为一个 JavaScript 处理的字符串被发送。下面的函数用于将 JavaScript 代码、CSS 样式表和图片转换为浏览器可用的资源：

```
function handleImageData(data, mimeType) {
  var img = document.createElement('img');
  img.src = 'data:' + mimeType + ';base64,' + data;
  return img;
}
function handleCss(data) {
  var style = document.createElement('style');
  style.type = 'text/css';
  var node = document.createTextNode(data);
  style.appendChild(node);
  document.getElementsByTagName_r('head')[0].appendChild(style);
}
function handleJavaScript(data) {
  (data);
}
```

As MXHR responses grow larger, it becomes necessary to process each resource as it is received, rather than waiting for the entire response. This can be done by listening for **readyState** 3:

由于 MXHR 响应报文越来越大，有必要在每个资源收到时立刻处理，而不是等待整个响应报文接收完成。这可以通过监听 readyState 3 实现：

```
 var req = new XMLHttpRequest();
var getLatestPacketInterval, lastLength = 0;
req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = readyStateHandler;
req.send(null);
function readyStateHandler{
  if (req.readyState === 3 && getLatestPacketInterval === null) {
    // Start polling.
    getLatestPacketInterval = window.setInterval(function() {
      getLatestPacket();
    }, 15);
  }
  if (req.readyState === 4) {
    // Stop polling.
    clearInterval(getLatestPacketInterval);
    // Get the last packet.
    getLatestPacket();
  }
}
function getLatestPacket() {
  var length = req.responseText.length;
  var packet = req.responseText.substring(lastLength, length);
  processPacket(packet);
  lastLength = length;
}
```

Once **readyState** 3 fires for the first time, a timer is started. Every 15 milliseconds, the response is checked for new data. Each piece of data is then collected until a delimiter character is found, and then everything is processed as a complete resource.

当 readyState 3 第一次发出时，启动了一个定时器。每隔 15 毫秒检查一次响应报文中的新数据。数据片段被收集起来直到发现一个分隔符，然后一切都作为一个完整的资源处理。

The code required to use MXHR in a robust manner is complex but worth further study. The complete library can be easily be found online at http://techfoolery.com/mxhr/.

以健壮的方式使用 MXHR 的代码很复杂但值得进一步研究。完整的库可参见 http://techfoolery.com/mxhr/。

There are some downsides to using this technique, the biggest being that none of the fetched resources are cached in the browser. If you fetch a particular CSS file using MXHR and then load it normally on the next page, it will not be in the cache. This is because the rolled-up resources are transmitted as a long string and then split up by the JavaScript code. Since there is no way to programmatically inject a file into the browser's cache, none of the resources fetched in this way will make it there.

使用此技术有一些缺点，其中最大的缺点是以此方法获得的资源不能被浏览器缓存。如果你使用 MXHR 获取一个特定的 CSS 文件然后在下一个页面中正常加载它，它不在缓存中。因为整批资源是作为一个长字符串传输的，然后由 JavaScript 代码分割。由于没有办法用程序将文件放入浏览器缓存中，所以用这种方法获取的资源也无法存放在那里。

Another downside is that older versions of Internet Explorer don't support **readyState** 3 or **data: URL**s. Internet Explorer 8 does support both of them, but workarounds must still be used for Internet Explorer 6 and 7.

另一个缺点是：老版本的 Internet Explorer 不支持 readyState 3 或 data: URL。Internet Explorer 8 两个都支持，但在 Internet Explorer 6 和 7 中必须设法变通。

Despite these downsides, there are still situations in which MXHR significantly improves overall page performance:

尽管有这些缺点，但某些情况下 MXHR 仍然显著提高了整体页面的性能：

• Pages that contain a lot of resources that aren't used elsewhere on the site (and thus don't need to be cached), especially images

网页包含许多其他地方不会用到的资源（所以不需要缓存），尤其是图片

• Sites that already use a unique rolled-up JavaScript or CSS file on each page to reduce HTTP requests; because it is unique to each page, it's never read from cache unless that particular page is reloaded

网站为每个页面使用了独一无二的打包的 JavaScript 或 CSS 文件以减少 HTTP 请求，因为它们对每个页面来说是独一的，所以不需要从缓存中读取，除非重新载入特定页面

Because HTTP requests are one of the most extreme bottlenecks in Ajax, reducing the number needed has a large effect on overall page performance. This is especially true when you are able to convert 100 image requests into a single multipart XHR request. Ad hoc testing with large numbers of images across modern browsers has shown this technique to be 4 to 10 times faster than making individual requests. Run these tests for yourself at http://techfoolery.com/mxhr/.

由于 HTTP 请求是 Ajax 中最极端的瓶颈之一，减少其需求数量对整个页面性能有很大影响。尤其是当你将 100 个图片请求转化为一个 MXHR 请求时。Ad hoc 在现代浏览器上测试了大量图片，其结果显示出此技术比逐个请求快了 4 到 10 倍。你可以自己运行这个测试：http://techfoolery.com/mxhr/。

**Sending Data　发送数据**

There are times when you don't care about retrieving data, and instead only want to send it to the server. You could be sending off nonpersonal information about a user to be analyzed later, or you could capture all script errors that occur and send the details about them to the server for logging and alerting. When data only needs to be sent to the server, there are two techniques that are widely used: XHR and beacons.

有时你不关心接收数据，而只要将数据发送给服务器。你可以发送用户的非私有信息以备日后分析，或者捕获所有脚本错误然后将有关细节发送给服务器进行记录和提示。当数据只需发送给服务器时，有两种广泛应用的技术：XHR 和灯标。

**XMLHttpRequest**

Though primarily used for requesting data from the server, XHR can also be used to send data back. Data can be sent back as GET or POST, as well as in any number of HTTP headers. This gives you an enormous amount of flexibility. XHR is especially useful when the amount of data you are sending back exceeds the maximum URL length in a browser. In that situation, you can send the data back as a POST:

虽然 XHR 主要用于从服务器获取数据，它也可以用来将数据发回。数据可以用 GET 或 POST 方式发回，以及任意数量的 HTTP 信息头。这给你很大灵活性。当你向服务器发回的数据量超过浏览器的最大 URL 长度时 XHR 特别有用。这种情况下，你可以用 POST 方式发回数据：

```
var url = '/data.php';
var params = [
  'id=934875',
  'limit=20'
];
var req = new XMLHttpRequest();
req.onerror = function() {
  // Error.
};
req.onreadystatechange = function() {
  if (req.readyState == 4) {
    // Success.
  }
};
req.open('POST', url, true);
req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
req.setRequestHeader('Content-Length', params.length);
req.send(params.join('&'));
```

As you can see in this example, we do nothing if the post fails. This is usually fine when XHR is used to capture broad user statistics, but if it's crucial that the data makes it to the server, you can add code to retry on failure:

正如你在这个例子中看到的，如果失败了我们什么也不做。当我们用 XHR 捕获登陆用户统计信息时这么做通常没什么问题，但是，如果发送到服务器的是至关重要的数据，你可以添加代码在失败时重试：

```
function xhrPost(url, params, callback) {
  var req = new XMLHttpRequest();
  req.onerror = function() {
    setTimeout(function() {
      xhrPost(url, params, callback);
    }, 1000);
  };
  req.onreadystatechange = function() {
    if (req.readyState == 4) {
      if (callback && typeof callback === 'function') {
        callback();
      }
    }
  };
  req.open('POST', url, true);
  req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
  req.setRequestHeader('Content-Length', params.length);
  req.send(params.join('&'));
}
```

When using XHR to send data back to the server, it is faster to use GET. This is because, for small amounts of data, a GET request is sent to the server in a single packet. A POST, on the other hand, is sent in a minimum of two packets, one for the headers and another for the POST body. A POST is better suited to sending large amounts of data to the server, both because the extra packet won't matter as much and because of Internet Explorer's URL length limit, which makes long GET requests impossible.

当使用 XHR 将数据发回服务器时，它比使用 GET 要快。这是因为对少量数据而言，向服务器发送一个 GET 请求要占用一个单独的数据包。另一方面，一个 POST 至少发送两个数据包，一个用于信息头。另一个用于 POST 体。POST 更适合于向服务器发送大量数据，即因为它不关心额外数据包的数量，又因为 Internet Explorer 的 URL 长度限制，它不可能使用过长的 GET 请求。

**Beacons** 灯标

This technique is very similar to dynamic script tag insertion. JavaScript is used to create a new **Image** object, with the **src** set to the URL of a script on your server. This URL contains the data we want to send back in the GET format of key-value pairs. Note that no **img** element has to be created or inserted into the DOM.

此技术与动态脚本标签插入非常类似。JavaScript 用于创建一个新的 Image 对象，将 src 设置为服务器上一个脚本文件的 URL。此 URL 包含我们打算通过 GET 格式传回的键值对数据。注意并没有创建 img 元素或者将它们插入到 DOM 中。

```
var url = '/status_tracker.php';
var params = [
  'step=2',
  'time=1248027314'
];
(new Image()).src = url + '?' + params.join('&');
```

The server takes this data and stores it; it doesn't have to send anything back to the client, since the image isn't actually displayed. This is the most efficient way to send information back to the server. There is very little overhead, and server-side errors don't affect the client side at all.

服务器取得此数据并保存下来，而不必向客户端返回什么，因此没有实际的图像显示。这是将信息发回服务器的最有效方法。其开销很小，而且任何服务器端错误都不会影响客户端。

The simplicity of image beacons also means that you are restricted in what you can do. You can't send POST data, so you are limited to a fairly small number of characters before you reach the maximum allowed URL length. You can receive data back, but in very limited ways. It's possible to listen for the **Image** object's **load** event, which will tell you if the server successfully received the data. You can also check the width and height of the image that the server returned (if an image was returned) and use those numbers to inform you about the server's state. For instance, a width of 1 could be "success" and 2 could be "try again."

简单的图像灯标意味着你所能做的受到限制。你不能发送 POST 数据，所以你被 URL 长度限制在一个相当小的字符数量上。你可以用非常有限的方法接收返回数据。可以监听 Image 对象的 load 事件，它可以告诉你服务器端是否成功接收了数据。你还可以检查服务器返回图片的宽度和高度（如果返回了一张图片）并用这些数字通知你服务器的状态。例如，宽度为 1 表示"成功"，2 表示"重试"。

If you don't need to return data in your response, you should send a response code of **204 No Content** and no message body. This will prevent the client from waiting for a message body that will never come:

如果你不需要为此响应返回数据，那么你应当发送一个 204 No Content 响应代码，无消息正文。它将阻止客户端继续等待永远不会到来的消息体：

```
 var url = '/status_tracker.php';
var params = [
  'step=2',
  'time=1248027314'
];
var beacon = new Image();
beacon.src = url + '?' + params.join('&');
beacon.onload = function() {
  if (this.width == 1) {
    // Success.
  }
  else if (this.width == 2) {
    // Failure; create another beacon and try again.
  }
};
beacon.onerror = function() {
  // Error; wait a bit, then create another beacon and try again.
};
```

Beacons are the fastest and most efficient way to send data back to the server. The server doesn't have to send back any response body at all, so you don't have to worry about downloading data to the client. The only downside is that it you are limited in the type of responses you can receive. If you need to pass large amounts of data back to the client, use XHR. If you only care about sending data to the server (with possibly a very simple response), use image beacons.

灯标是向服务器回送数据最快和最有效的方法。服务器根本不需要发回任何响应正文，所以你不必担心客户端下载数据。唯一的缺点是接收到的响应类型是受限的。如果你需要向客户端返回大量数据，那么使用 XHR。如果你只关心将数据发送到服务器端（可能需要极少的回复），那么使用图像灯标。

## Data Formats 数据格式

When considering data transmission techniques, you must take into account several factors: feature set, compatibility, performance, and direction (to or from the server). When considering data formats, the only scale you need for comparison is speed.

在考虑数据传输技术时，你必须考虑这些因素：功能集，兼容性，性能，和方向（发给服务器或者从服务器接收）。在考虑数据格式时，唯一需要比较的尺度的就是速度。

There isn't one data format that will always be better than the others. Depending on what data is being transferred and its intended use on the page, one might be faster to download, while another might be faster to parse. In this section, we create a widget for searching among users and implement it using each of the four major categories of data formats. This will require us to format a list of users on the server, pass it back to the browser, parse that list into a native JavaScript data structure, and search it for a given string. Each of the data formats will be compared based on the file size of the list, the speed of parsing it, and the ease with which it's formed on the server.

没有哪种数据格式会始终比其他格式更好。根据传送什么数据、用于页面上什么目的，某种格式可能下载更快，另一种格式可能解析更快。在本节中，我们创建了一个窗口小部件用于搜索用户信息并用四种主流的数据格式实现它。这要求我们在服务器端格式化一个用户列表，将它返回给浏览器，将列表解析成 JavaScript 数据格式，并搜索特定的字符串。每种数据格式将比较列表的文件大小，解析速度，和服务器上构造它们的难易程度。

**XML**

When Ajax first became popular, XML was the data format of choice. It had many things going for it: extreme interoperability (with excellent support on both the server side and the client side), strict formatting, and easy validation. JSON hadn't been formalized yet as an interchange format, and almost every language used on servers had a library available for working with XML.

当 Ajax 开始变得流行起来它选择了 XML 数据格式。有很多事情是围绕着它做的：极端的互通性（服务器端和客户端都能够良好支持），格式严格，易于验证。（那时）JSON 还没有正式作为交换格式，几乎所有的服务器端语言都有操作 XML 的库。

Here is an example of our list of users encoded as XML:

这里是用 XML 编码的用户列表的例子：

```
<?xml version="1.0" encoding='UTF-8'?>
<users total="4">
  <user id="1">
```

```
    <username>alice</username>
    <realname>Alice Smith</realname>
    <email>alice@alicesmith.com</email>
  </user>
  <user id="2">
    <username>bob</username>
    <realname>Bob Jones</realname>
    <email>bob@bobjones.com</email>
  </user>
  <user id="3">
    <username>carol</username>
    <realname>Carol Williams</realname>
    <email>carol@carolwilliams.com</email>
  </user>
  <user id="4">
    <username>dave</username>
    <realname>Dave Johnson</realname>
    <email>dave@davejohnson.com</email>
  </user>
</users>
```

Compared to other formats, XML is extremely verbose. Each discrete piece of data requires a lot of structure, and the ratio of data to structure is extremely low. XML also has a slightly ambiguous syntax. When encoding a data structure into XML, do you make object parameters into attributes of the object element or independent child elements? Do you make long, descriptive tag names, or short ones that are efficient but indecipherable? Parsing this syntax is equally ambiguous, and you must know the layout of an XML response ahead of time to be able to make sense of it.

与其他格式相比，XML 极其冗长。每个离散的数据片断需要大量结构，所以有效数据的比例非常低。而且 XML 语法有些轻微模糊。当数据结构编码为 XML 之后，你将对象参数放在对象元素的属性中，还

是放在独立的子元素中？标签名使用长命名或者短小高效却难以辨认的名字？语法解析同样含混，你必须先知道 XML 响应报文的布局，然后才能搞清楚它的含义。

In general, parsing XML requires a great deal of effort on the part of the JavaScript programmer. Aside from knowing the particulars of the structure ahead of time, you must also know exactly how to pull apart that structure and painstakingly reassemble it into a JavaScript object. This is far from an easy or automatic process, unlike the other three data formats.

一般情况下，解析 XML 要占用 JavaScript 程序员相当一部分精力。除了要提前知道详细结构之外，你还必须确切地知道如何解开这个结构然后精心地将它们写入 JavaScript 对象中。这远非易事且不能自动完成，不像其他三种数据格式那样。

Here is an example of how to parse this particular XML response into an object:

下面是如何将特定 XML 报文解析到对象的例子：

```
function parseXML(responseXML) {
  var users = [];
  var userNodes = responseXML.getElementsByTagName_r('users');
  var node, usernameNodes, usernameNode, username,
  realnameNodes, realnameNode, realname,
  emailNodes, emailNode, email;
  for (var i = 0, len = userNodes.length; i < len; i++) {
    node = userNodes[i];
    username = realname = email = '';
    usernameNodes = node.getElementsByTagName_r('username');
    if (usernameNodes && usernameNodes[0]) {
      usernameNode = usernameNodes[0];
      username = (usernameNodes.firstChild) ?
      usernameNodes.firstChild.nodeValue : '';
    }
    realnameNodes = node.getElementsByTagName_r('realname');
```

```
     if (realnameNodes && realnameNodes[0]) {
       realnameNode = realnameNodes[0];
       realname = (realnameNodes.firstChild) ?
       realnameNodes.firstChild.nodeValue : '';
     }
     emailNodes = node.getElementsByTagName_r('email');
    if (emailNodes && emailNodes[0]) {
       emailNode = emailNodes[0];
       email = (emailNodes.firstChild) ?
       emailNodes.firstChild.nodeValue : '';
     }
     users[i] = {
       id: node.getAttribute('id'),
       username: username,
       realname: realname,
       email: email
     };
   }
   return users;
}
```

As you can see, it requires checking each tag to ensure that it exists before reading its value. It is heavily dependent on the structure of the XML.

正如你所看到的，在读值之前，它需要检查每个标签以保证它存在。这在很大程度上依赖于 XML 的结构。

A more efficient approach would be to encode each of the values as an attribute of the <**user**> tag. This results in a smaller file size for the same amount of data. Here is an example of the user list with the values encoded as attributes:

一个更有效的方式是将每个值都存储为<user>标签的属性。数据相同而文件尺寸却更小。这个例子中的用户列表，将值放置在标签属性中：

```
<?xml version="1.0" encoding='UTF-8'?>
<users total="4">
  <user id="1-id001" username="alice" realname="Alice Smith" email="alice@alicesmith.com" />
  <user id="2-id001" username="bob" realname="Bob Jones" email="bob@bobjones.com" />
  <user id="3-id001" username="carol" realname="Carol Williams" email="carol@carolwilliams.com" />
  <user id="4-id001" username="dave" realname="Dave Johnson" email="dave@davejohnson.com" />
</users>
```

Parsing this simplified XML response is significantly easier:

解析此简化版 XML 响应报文要容易得多：

```
function parseXML(responseXML) {
  var users = [];
  var userNodes = responseXML.getElementsByTagName_r('users');
  for (var i = 0, len = userNodes.length; i < len; i++) {
    users[i] = {
      id: userNodes[i].getAttribute('id'),
      username: userNodes[i].getAttribute('username'),
      realname: userNodes[i].getAttribute('realname'),
      email: userNodes[i].getAttribute('email')
    };
  }
  return users;
}
```

**XPath**

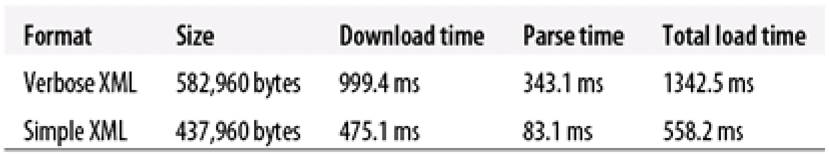
Though it is beyond the scope of this chapter, XPath can be much faster than **getElementsByTagName** when parsing an XML document. The caveat is that it is not universally supported, so you must also write fallback code using the older style of DOM traversal. At this time, DOM Level 3 XPath has been implemented by Firefox, Safari, Chrome, and Opera. Internet Explorer 8 has a similar but slightly less advanced interface.

虽然它超出了本章内容的范围，但 XPath 在解析 XML 文档时比 getElementsByTagName 快得多。需要注意的是，它并未得到广泛支持，所以你必须使用古老风格的 DOM 遍历方法编写备用代码。现在，DOM 级别 3 的 XPath 已经被如下浏览器实现：Firefox，Safari，Chrome，和 Opera。Internet Explorer 8 有一个类似的但略微先进的接口。

**Response sizes and parse times　响应报文大小和解析时间**

Let's take a look at the performance numbers for XML in the following table.

让我们来看一看下表中的 XML 性能数据：



| Format | Size | Download time | Parse time | Total load time |
|---|---|---|---|---|
| Verbose XML | 582,960 bytes | 999.4 ms | 343.1 ms | 1342.5 ms |
| Simple XML | 437,960 bytes | 475.1 ms | 83.1 ms | 558.2 ms |

As you can see, using favoring attributes over child tags leads to a smaller file size and a significantly faster parse time. This is mostly due to the fact that you don't have to walk the DOM on the XML structure as much, and can instead simply read attributes.

正如你所看到的，与使用子标签相比，使用属性时文件尺寸更小，特别是解析时间更快。其原因很大程度上基于这样的事实：你不需要在 XML 结构上访问 DOM 那么多次，而只是简单地读取属性。

Should you consider using XML? Given its prevalence in public APIs, you often have no choice. If the data is only available in XML, you roll up your sleeves and write code to parse it. But if there is any other format available, prefer that instead. The performance numbers you see here for verbose XML are extremely slow compared to more advanced techniques. For browsers that support it, XPath would improve the parse time, but at the cost of writing and maintaining three separate code paths (one for browsers that support DOM Level 3 XPath, one for Internet Explorer 8, and one for all other browsers). The simple XML format compares more favorably, but is still an order of magnitude slower than the fastest format. XML has no place in high-performance Ajax.

你是否考虑使用 XML？鉴于开发 API 如此流行使，你经常别无选择。如果数据只有 XML 格式可用，那么你卷起袖子写代码解析它吧。但是如果有其他格式可用，那么宁愿取代它。你在这里看到的标准 XML 的性能数据与更先进的技术相比，显得太慢了。如果浏览器支持的话，XPath 可改善解析时间，但代价是编写并维护三个代码分支（为支持 DOM 级别 3 的 XPath 的浏览器写一个，为 Internet Explorer 8 写一个，为其他浏览器写一个）。简化 XML 格式更有利，但比那些最快的格式还是慢一个数量级。在高性能 Ajax 中没有 XML 的地位。

**JSON**

Formalized and popularized by Douglas Crockford, JSON is a lightweight and easy-to-parse data format written using JavaScript object and array literal syntax. Here is an example of the user list written in JSON:

通过 Douglas Crockford 的发明与推广，JSON 是一个轻量级并易于解析的数据格式，它按照 JavaScript 对象和数组字面语法所编写。下例是用 JSON 书写的用户列表：

```
[
  {"id":1, "username":"alice", "realname": "Alice Smith", "email":"alice@alicesmith.com"},
  {"id":2, "username":"bob", "realname": "Bob Jones", "email":"bob@bobjones.com"},
  {"id":3, "username":"carol", "realname": "Carol Williams","email":"carol@carolwilliams.com"},
  {"id":4, "username":"dave", "realname": "Dave Johnson", "email":"dave@davejohnson.com"}
]
```

The users are represented as objects, and the list of users is an array, just as any other array or object would be written out in JavaScript. This means that when **eval**ed or wrapped in a callback function, JSON data is executable JavaScript code. Parsing a string of JSON in JavaScript is as easy as using **()**:

用户表示为一个对象，用户列表成为一个数组，与 JavaScript 中其他数组或对象的写法相同。这意味着如果它被包装在一个回调函数中，JSON 数据可称为能够运行的 JavaScript 代码。在 JavaScript 中解析 JSON 可简单地使用()：

```
function parseJSON(responseText) {
  return ('(' + responseText + ')');
}
```

Just as with XML, it is possible to distill this format into a simpler version. In this case, we can replace the attribute names with shortened (though less readable) versions:

正如 XML 那样，它也可以提炼成一个更简单的版本。这种情况下，我们可将名字缩短（尽管可读性变差）：

```
[
  { "i": 1, "u": "alice", "r": "Alice Smith", "e": "alice@alicesmith.com" },
  { "i": 2, "u": "bob", "r": "Bob Jones", "e": "bob@bobjones.com" },
  { "i": 3, "u": "carol", "r": "Carol Williams", "e": "carol@carolwilliams.com" },
  { "i": 4, "u": "dave", "r": "Dave Johnson", "e": "dave@davejohnson.com" }
]
```

This gives us the same data with less structure and fewer bytes overall to transmit to the browser. We can even take it a step further and remove the attribute names completely. This format is even less readable than the other two and is much more brittle, but the file size is much smaller: almost half the size of the verbose JSON format.

它将相同的数据以更少的结构和更小的字节尺寸传送给浏览器。更进一步，我们干脆完全去掉属性名。与其他两种格式相比，这种格式可读性更差，但也更利索，文件尺寸非常小：大约只有标准 JSON 格式的一半。

```
[
  [ 1, "alice", "Alice Smith", "alice@alicesmith.com" ],
  [ 2, "bob", "Bob Jones", "bob@bobjones.com" ],
  [ 3, "carol", "Carol Williams", "carol@carolwilliams.com" ],
  [ 4, "dave", "Dave Johnson", "dave@davejohnson.com" ]
]
```

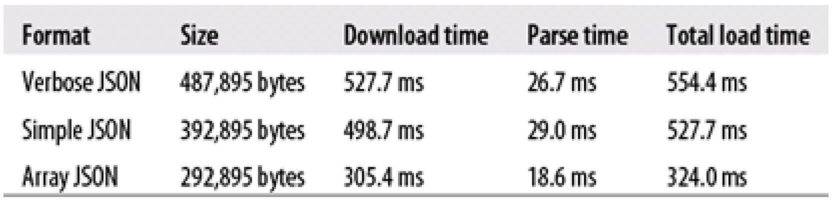
Successful parsing requires that the order of the data must be maintained. That being said, it is trivial to convert this format into one that maintains the same attribute names as the first JSON format:

解析过程需要保持数据的顺序。也就是说，它在进行格式转换时必须保持和第一个 JSON 格式一样的属性名：

```
function parseJSON(responseText) {
  var users = [];
  var usersArray = ('(' + responseText + ')');
    for (var i = 0, len = usersArray.length; i < len; i++) {
    users[i] = {
      id: usersArray[i][0],
      username: usersArray[i][1],
      realname: usersArray[i][2],
      email: usersArray[i][3]
    };
  }
  return users;
}
```

In this example, we use **()** to convert the string into a native JavaScript array. That array of arrays is then converted into an array of objects. Essentially, you are trading a smaller file size and faster **()** time for a more complicated parse function. The following table lists the performance numbers for the three JSON formats, transferred using XHR.

在这个例子中，我们使用()将字符串转换为一个本地 JavaScript 数组。然后再将它转换到一个对象数组，本质上讲，你用一个更复杂的解析函数换取了较小的文件尺寸和更快的()时间。下表列出这三种 JSON 格式的性能数据，以 XHR 传输。



| Format | Size | Download time | Parse time | Total load time |
|---|---|---|---|---|
| Verbose JSON | 487,895 bytes | 527.7 ms | 26.7 ms | 554.4 ms |
| Simple JSON | 392,895 bytes | 498.7 ms | 29.0 ms | 527.7 ms |
| Array JSON | 292,895 bytes | 305.4 ms | 18.6 ms | 324.0 ms |

JSON formed using arrays wins every category, with the smallest file size, the fastest average download time, and the fastest average parse time. Despite the fact that the parse function has to iterate through all 5,000 entries in the list, it is still more than 30% faster to parse.

数组形式的 JSON 在每一项中均获胜，它文件尺寸最小，下载最快，平均解析时间最短。尽管解析函数不得不遍历列表中所有 5'000 个单元，它还是快出了 30%。

**JSON-P**

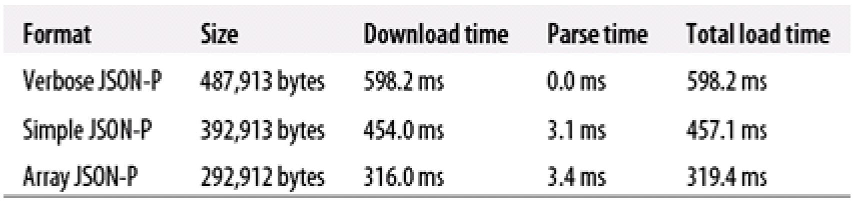
The fact that JSON can be executed natively has several important performance implications. When XHR is used, JSON data is returned as a string. This string is then evaluated using **()** to convert it into a native object. However, when dynamic script tag insertion is used, JSON data is treated as just another JavaScript file and executed as native code. In order to accomplish this, the data must be wrapped in a callback function. This is known as "JSON with padding," or JSON-P. Here is our user list formatted as JSON-P:

事实上 JSON 可被本地执行有几个重要的性能影响。当使用 XHR 时 JSON 数据作为一个字符串返回。该字符串使用()转换为一个本地对象。然而，当使用动态脚本标签插入时，JSON 数据被视为另一个 JavaScript 文件并作为本地码执行。为做到这一点，数据必须被包装在回调函数之中。这就是所谓的"JSON 填充"或 JSON-P。下面是我们用 JSON-P 格式书写的用户列表：

```
parseJSON([
  {"id":1, "username":"alice", "realname":"Alice Smith", "email":"alice@alicesmith.com"},
  {"id":2, "username":"bob", "realname":"Bob Jones", "email":"bob@bobjones.com"},
  {"id":3, "username":"carol", "realname":"Carol Williams", "email":"carol@carolwilliams.com"},
  {"id":4, "username":"dave", "realname":"Dave Johnson", "email":"dave@davejohnson.com"}
]);
```

JSON-P adds a small amount to the file size with the callback wrapper, but such an increase is insignificant compared to the improved parse times. Since the data is treated as native JavaScript, it is parsed at native JavaScript speeds. Here are the same three JSON formats transmitted as JSON-P.

JSON-P 因为回调包装的原因略微增加了文件尺寸，但与其解析性能的改进相比这点增加微不足道。由于数据作为本地 JavaScript 处理，它的解析速度像本地 JavaScript 一样快。下面是 JSON-P 传输三种 JSON 数据的时间：



| Format | Size | Download time | Parse time | Total load time |
|---|---|---|---|---|
| Verbose JSON-P | 487,913 bytes | 598.2 ms | 0.0 ms | 598.2 ms |
| Simple JSON-P | 392,913 bytes | 454.0 ms | 3.1 ms | 457.1 ms |
| Array JSON-P | 292,912 bytes | 316.0 ms | 3.4 ms | 319.4 ms |

File sizes and download times are almost identical to the XHR tests, but parse times are almost 10 times faster. The parse time for verbose JSON-P is zero, since no parsing is needed; it is already in a native format. The same is true for simple JSON-P and array JSON-P, but each had to be iterated through to convert it to the format that verbose JSON-P gives you naturally.

文件大小和下载时间与 XHR 测试基本相同，而解析时间几乎快了 10 倍。标准 JSON-P 的解析时间为 0，因为根本用不着解析，它已经是本地格式了。简化版 JSON-P 和数组 JSON-P 也是如此，只是每种都需要转换成标准 JSON-P 直接给你的那种格式。

The fastest JSON format is JSON-P formed using arrays. Although this is only slightly faster than JSON transmitted using XHR, that difference increases as the size of the list grows. If you are working on a project that requires a list with 10,000 or 100,000 elements in it, favor JSON-P over JSON.

最快的 JSON 格式是使用数组的 JSON-P 格式。虽然这只比使用 XHR 的 JSON 略快，但是这种差异随着列表尺寸的增大而增大。如果你所从事的项目需要一个 10'000 或 100'000 个单元构成的列表，那么 JSON-P 比 JSON 好很多。

There is one reason to avoid using JSON-P that has nothing to do with performance: since JSON-P must be executable JavaScript, it can be called by anyone and included in any website using dynamic script tag insertion. JSON, on the other hand, is not valid JavaScript until it is **eval**ed, and can only be fetched as a string using XHR.

Do not encode any sensitive data in JSON-P, because you cannot ensure that it will remain private, even with random URLs or cookies.

还有一个与性能无关的原因要避免使用 JSON-P：因为 JSON-P 必须是可执行的 JavaScript，它使用动态脚本标签注入技术可在任何网站中被任何人调用。从另一个角度说，JSON 在运行之前并不是有效的 JavaScript，使用 XHR 时只是被当作字符串获取。不要将任何敏感的数据编码为 JSON-P，因为你无法确定它是否包含私密信息，或者包含随机的 URL 或 cookie。

**Should you use JSON?  你应该使用 JSON 吗？**

JSON has several advantages when compared to XML. It is a much smaller format, with less of the overall response size being used as structure and more as data. This is especially true when the data contains arrays rather than objects. JSON is extremely interoperable, with encoding and decoding libraries available for most server-side languages. It is trivial to parse on the client side, allowing you to spend more time writing code to actually do something with the data. And, most importantly for web developers, it is one of the best performing formats, both because it is relatively small over the wire and because it can be parsed so quickly. JSON is a cornerstone of high-performance Ajax, especially when used with dynamic script tag insertion.

与 XML 相比 JSON 有许多优点。这种格式小得多，在总响应报文中，结构占用的空间更小，数据占用的更多。特别是数据包含数组而不是对象时。JSON 与大多数服务器端语言的编解码库之间有着很好的互操作性。它在客户端的解析工作微不足道，使你可以将更多写代码的时间放在其他数据处理上。对网页开发者来说最重要的是，它是表现最好的格式之一，即因为在线传输相对较小，也因为解析十分之快。JSON 是高性能 Ajax 的基石，特别是使用动态脚本标签插入时。

**HTML**

Often the data you are requesting will be turned into HTML for display on the page. Converting a large data structure into simple HTML can be done relatively quickly in JavaScript, but it can be done much faster on the server. One technique to consider is forming all of the HTML on the server and then passing it intact to the client; the JavaScript can then simply drop it in place with **innerHTML**. Here is an example of the user list encoded as HTML:

通常你所请求的数据以 HTML 返回并显示在页面上。JavaScript 能够比较快地将一个大数据结构转化为简单的 HTML，但是服务器完成同样工作更快。一种技术考虑是在服务器端构建整个 HTML 然后传递给客户端，JavaScript 只是简单地下载它然后放入 innerHTML。下面是用 HTML 编码用户列表的例子：

```
<ul class="users">
  <li class="user" id="1-id002">
    <a href="http://www.site.com/alice/" class="username">alice</a>
    <span class="realname">Alice Smith</span>
    <a href="mailto:alice@alicesmith.com" class="email">alice@alicesmith.com</a>
  </li>
  <li class="user" id="2-id002">
    <a href="http://www.site.com/bob/" class="username">bob</a>
    <span class="realname">Bob Jones</span>
    <a href="mailto:bob@bobjones.com" class="email">bob@bobjones.com</a>
  </li>
  <li class="user" id="3-id002">
    <a href="http://www.site.com/carol/" class="username">carol</a>
    <span class="realname">Carol Williams</span>
    <a href="mailto:carol@carolwilliams.com" class="email">carol@carolwilliams.com</a>
  </li>
  <li class="user" id="4-id002">
    <a href="http://www.site.com/dave/" class="username">dave</a>
    <span class="realname">Dave Johnson</span>
    <a href="mailto:dave@davejohnson.com" class="email">dave@davejohnson.com</a>
  </li>
</ul>
```

The problem with this technique is that HTML is a verbose data format, more so even than XML. On top of the data itself, you could have nested HTML tags, each with IDs, classes, and other attributes. It's possible to have the HTML formatting take up more space than the actual data, though that can be mitigated by using as few tags and attributes as possible. Because of this, you should use this technique only when the client-side CPU is more limited than bandwidth.

此技术的问题在于，HTML 是一种详细的数据格式，比 XML 更加冗长。在数据本身的最外层，可有嵌套的 HTML 标签，每个都具有 ID，类，和其他属性。HTML 格式可能比实际数据占用更多的空间，尽管可通过尽量少用标签和属性缓解这一问题。正因为这个原因，你只有当客户端 CPU 比带宽更受限时才使用此技术。

On one extreme, you have a format that consists of the smallest amount of structure required to parse the data on the client side, such as JSON. This format is extremely quick to download to the client machine; however, it takes a lot of CPU time to convert this format into HTML to display on the page. A lot of string operations are required, which are one of the slowest things you can do in JavaScript.

一种极端情况是，你有一种格式包含最少数量的结构，需要在客户端解析数据，例如 JSON。这种格式下载到客户机非常快，然而它需要很多 CPU 时间转化成 HTML 以显示在页面上。这需要很多字符串操作，而字符串操作也是 JavaScript 最慢的操作之一。

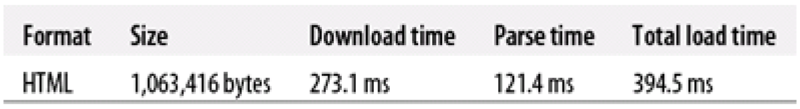
On the other extreme, you have HTML created on the server. This format is much larger over the wire and takes longer to download, but once it's downloaded, displaying it on the page requires a single operation:

另一种极端情况是，你在服务器上创建 HTML。这种格式在线传输数据量大，下载时间长，但一旦下载完，只需一个操作就可以显示在页面上：

```
document.getElementById('data-container').innerHTML = req.responseText;
```

The following table shows the performance numbers for the user list encoded using HTML. Keep in mind the main different between this format and all others: "parsing" in this case refers to the action of inserting the HTML in the DOM. Also, HTML cannot be easily or quickly iterated through, unlike a native JavaScript array.

下表显示了使用 HTML 编码用户列表时的性能数据。请记住这种格式与其他几种格式的差别："解析"在这种情况下指的是将 HTML 插入 DOM 的操作。此外，HTML 不能像本地 JavaScript 数组那样轻易迅速地进行迭代操作。



| Format | Size | Download time | Parse time | Total load time |
|---|---|---|---|---|
| HTML | 1,063,416 bytes | 273.1 ms | 121.4 ms | 394.5 ms |

As you can see, HTML is significantly larger over the wire, and also takes a long time to parse. This is because the single operation to insert the HTML into the DOM is deceptively simple; despite the fact that it is a single line of code, it still takes a significant amount of time to load that much data into a page. These performance numbers do deviate slightly from the others, in that the end result is not an array of data, but instead HTML elements displayed on a page. Regardless, they still illustrate the fact that HTML, as a data format, is slow and bloated.

正如你所看到的，HTML 传输数据量明显偏大，也需要长时间来解析。因为将 HTML 插入到 DOM 的单一操作看似简单，尽管它只有一行代码，却仍需要时间向页面加载很多数据。与其他测试相比这些性能数据确实有轻微的偏差，最终结果不是数据数组，而是显示在页面上的 HTML 元素。无论如何，它们仍显示出 HTML 的一个事实：作为数据格式，它缓慢而且臃肿。

**Custom Formatting　自定义格式**

The ideal data format is one that includes just enough structure to allow you to separate individual fields from each other. You can easily make such a format by simply concatenating your data with a separator character:

最理想的数据格式只包含必要的结构，使你能够分解出每个字段。你可以自定义一种格式只是简单地用一个分隔符将数据连结起来：

```
Jacob;Michael;Joshua;Matthew;Andrew;Christopher;Joseph;Daniel;Nicholas;Ethan;William;Anthony;Ryan;Davi
d;Tyler;John
```

These separators essentially create an array of data, similar to a comma-separated list. Through the use of different separators, you can create multidimensional arrays. Here is our user list encoded as a character-delimited custom format:

这些分隔符基本上创建了一个数据数组，类似于一个逗号分隔的列表。通过使用不同的分隔符，你可以创建多维数组。这里是用自定义的字符分隔方式构造的用户列表：

```
1:alice:Alice Smith:alice@alicesmith.com;
2:bob:Bob Jones:bob@bobjones.com;
```

```
3:carol:Carol Williams:carol@carolwilliams.com;
4:dave:Dave Johnson:dave@davejohnson.com
```

This type of format is extremely terse and offers a very high data-to-structure ratio (significantly higher than any other format, excluding plain text). Custom formats are quick to download over the wire, and they are fast and easy to parse; you simply call **split()** on the string, using your separator as the argument. More complex custom formats with multiple separators require loops to split all the data (but keep in mind that these loops are extremely fast in JavaScript). **split()** is one of the fastest string operations, and can typically handle separator-delimited lists of 10,000+ elements in a matter of milliseconds. Here is an example of how to parse the preceding format:

这种格式非常简洁，与其他格式相比（不包括纯文本），其数据/结构比例明显提高。自定义格式下载迅速，易于解析，只需调用字符串的 split()将分隔符作为参数传入即可。更复杂的自定义格式具有多种分隔符，需要在循环中分解所有数据（但是请记住，在 JavaScript 中这些循环是非常快的）。split()是最快的字符串操作之一，通常可以在数毫秒内处理具有超过 10'000 个元素的“分隔符分割”列表。下面的例子给出解析上述格式的方法：

```
function parseCustomFormat(responseText) {
  var users = [];
  var usersEncoded = responseText.split(';');
  var userArray;
  for (var i = 0, len = usersEncoded.length; i < len; i++) {
    userArray = usersEncoded[i].split(':');
    users[i] = {
      id: userArray[0],
      username: userArray[1],
      realname: userArray[2],
      email: userArray[3]
    };
  }
  return users;
}
```

When creating you own custom format, one of the most important decisions is what to use as the separators. Ideally, they should each be a single character, and they should not be found naturally in your data. Low-number ASCII characters work well and are easy to represent in most server-side languages. For example, here is how you would use ASCII characters in PHP:

当创建你的自定义格式时，最重要的决定是采用何种分隔符。理想情况下，它应当是一个单字符，而且不能存在于你的数据之中。ASCII 字符表中前面的几个字符在大多数服务器端语言中能够正常工作而且容易书写。例如，下面讲述如何在 PHP 中使用 ASCII 码：

```
function build_format_custom($users) {
  $row_delimiter = chr(1); // \u0001 in JavaScript.
  $field_delimiter = chr(2); // \u0002 in JavaScript.
  $output = array();
  foreach ($users as $user) {
    $fields = array($user['id'], $user['username'], $user['realname'], $user['email']);
    $output[] = implode($field_delimiter, $fields);
  }
  return implode($row_delimiter, $output);
}
```

These control characters are represented in JavaScript using Unicode notation (e.g., \u0001). The **split()** function can take either a string or a regular expression as an argument. If you expect to have empty fields in your data, then use a string; if the delimiter is passed as a regular expression, **split()** in IE ignores the second delimiter when two are right next to each other. The two argument types are equivalent in other browsers.

这些控制字符在 JavaScript 中使用 Unicode 标注（例如，\u0001）表示。split()函数可以用字符串或者正则表达式作参数。如果你希望数据中存在空字段，那么就使用字符串；如果分隔符是一个正则表达式，IE 中的 split()将跳过相邻两个分隔符中的第二个分隔符。这两种参数类型在其他浏览器上等价。

```
// Regular expression delimiter.
var rows = req.responseText.split(/\u0001/);
```

```
// String delimiter (safer).
var rows = req.responseText.split("\u0001");
```

Here are the performance numbers for a character delimited custom format, using both XHR and dynamic script tag insertion:

这里是字符分隔的自定义格式的性能数据，使用 XHR 和动态脚本标签注入：

| Format | Size | Download time | Parse time | Total load time |
|---|---|---|---|---|
| Custom Format (XHR) | 222,892 bytes | 63.1 ms | 14.5 ms | 77.6 ms |
| Custom Format (script insertion) | 222,912 bytes | 66.3 ms | 11.7 ms | 78.0 ms |

Either XHR or dynamic script tag insertion can be used with this format. Since the response is parsed as a string in both cases, there is no real difference in performance. For very large datasets, it's hands down the fastest format, beating out even natively executed JSON in parse speed and overall load time. This format makes it feasible to send huge amounts of data to the client side in a very short amount of time.

XHR 和动态脚本标签注入都可以使用这种格式。两种情况下都要解析字符串，在性能上没有实质上的差异。对于非常大的数据集，它是最快的传输格式，甚至可以在解析速度和下载时间上击败本机执行的 JSON。用此格式向客户端传送大量数据只用很少的时间。

## Data Format Conclusions 数据格式总结

Favor lightweight formats in general; the best are JSON and a character-delimited custom format. If the data set is large and parse time becomes an issue, use one of these two techniques:

总的来说越轻量级的格式越好，最好是 JSON 和字符分隔的自定义格式。如果数据集很大或者解析时间成问题，那么就使用这两种格式之一：

• JSON-P data, fetched using dynamic script tag insertion. This treats the data as executable JavaScript, not a string, and allows for extremely fast parsing. This can be used across domains, but shouldn't be used with sensitive data.

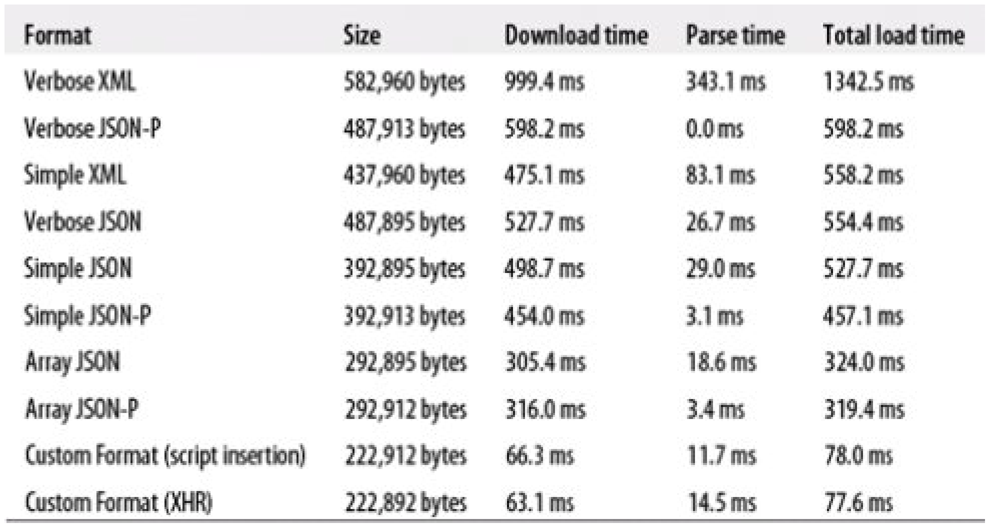
JSON-P 数据，用动态脚本标签插入法获取。它将数据视为可运行的 JavaScript 而不是字符串，解析速度极快。它能够跨域使用，但不应涉及敏感数据。

• A character-delimited custom format, fetched using either XHR or dynamic script tag insertion and parsed using split(). This technique parses extremely large datasets slightly faster than the JSON-P technique, and generally has a smaller file size.

字符分隔的自定义格式，使用 XHR 或动态脚本标签插入技术提取，使用 split()解析。此技术在解析非常大数据集时比 JSON-P 技术略快，而且通常文件尺寸更小。

The following table and Figure 7-1 show all of the performance numbers again (in order from slowest to fastest), so that you can compare each of the formats in one place. HTML is excluded, since it isn't directly comparable to the other formats.

下表和图 7-1 再次显示了所有方法的性能数据（按照从慢到快的顺序），你可以在此比较每种格式的优劣。HTML 未包括，因为它与其他格式不能直接比较。



| Format | Size | Download time | Parse time | Total load time |
|---|---|---|---|---|
| Verbose XML | 582,960 bytes | 999.4 ms | 343.1 ms | 1342.5 ms |
| Verbose JSON-P | 487,913 bytes | 598.2 ms | 0.0 ms | 598.2 ms |
| Simple XML | 437,960 bytes | 475.1 ms | 83.1 ms | 558.2 ms |
| Verbose JSON | 487,895 bytes | 527.7 ms | 26.7 ms | 554.4 ms |
| Simple JSON | 392,895 bytes | 498.7 ms | 29.0 ms | 527.7 ms |
| Simple JSON-P | 392,913 bytes | 454.0 ms | 3.1 ms | 457.1 ms |
| Array JSON | 292,895 bytes | 305.4 ms | 18.6 ms | 324.0 ms |
| Array JSON-P | 292,912 bytes | 316.0 ms | 3.4 ms | 319.4 ms |
| Custom Format (script insertion) | 222,912 bytes | 66.3 ms | 11.7 ms | 78.0 ms |
| Custom Format (XHR) | 222,892 bytes | 63.1 ms | 14.5 ms | 77.6 ms |

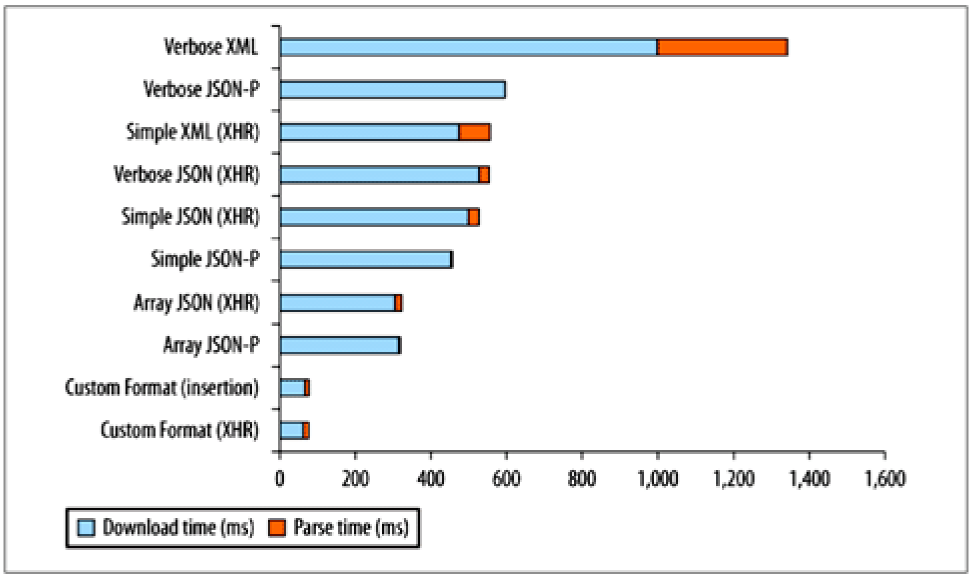


Figure 7-1. A comparison of data format download and parse times

图 7-1　各种数据格式下载和解析的时间

Keep in mind that these numbers are from a single test run in a single browser. The results should be used as general indicators of performance, not as hard numbers. You can run these tests yourself at http://techfoolery.com/formats/.

请注意，这些数据只是在一个浏览器上进行一次测试获得的。此结果可用作大概的性能指标，而不是确切的数字。你可以自己运行这些测试，位于：http://techfoolery.com/formats/。

## Ajax Performance Guidelines　Ajax 性能向导

Once you have selected the most appropriate data transmission technique and data format, you can start to consider other optimization techniques. These can be highly situational, so be sure that your application fits the profile before considering them.

一旦你选择了最合适的数据传输技术和数据格式，那么就开始考虑其他的优化技术吧。这些技术要根据具体情况使用，在考虑它们之前首先应确认你的应用程序是否能够适合这些概念。

### Cache Data　缓存数据

The fastest Ajax request is one that you don't have to make. There are two main ways of preventing an unnecessary request:

最快的 Ajax 请求就是你不要用它。有两种主要方法避免发出一个不必要的请求：

• On the server side, set HTTP headers that ensure your response will be cached in the browser.

在服务器端，设置 HTTP 头，确保返回报文将被缓存在浏览器中。

• On the client side, store fetched data locally so that it doesn't have be requested again.

在客户端，于本地缓存已获取的数据，不要多次请求同一个数据。

The first technique is the easiest to set up and maintain, whereas the second gives you the highest degree of control.

第一种技术最容易设置和维护，而第二个给你最大程度的控制。

**Setting HTTP headers 设置 HTTP 头**

If you want your Ajax responses to be cached by the browser, you must use GET to make the request. But simply using GET isn't sufficient; you must also send the correct HTTP headers with the response. The **Expires** header tells the browser how long a response can be cached. The value is a date; after that date has passed, any requests for that URL will stop being delivered from cache and will instead be passed on to the server. Here is what an **Expires** header looks like:

如果你希望 Ajax 响应报文能够被浏览器所缓存，你必须在发起请求时使用 GET 方法。但这还不充分，你必须在响应报文中发送正确的 HTTP 头。Expires 头告诉浏览器应当缓存响应报文多长时间。其值是一个日期，当过期之后任何对该 URL 发起的请求都不再从缓存中获得，而要重新访问服务器。一个 Expires 头如下：

```
Expires: Mon, 28 Jul 2014 23:30:00 GMT
```

This particular **Expires** header tells the browser to cache this response until July 2014. This is called a far future **Expires** header, and it is useful for content that will never change, such as images or static data sets.

这种特殊的 Expires 头告诉浏览器缓存此响应报文直到 2014 年 7 月。这就是所谓的遥远未来 Expires 头，用于那些永不改变的内容，例如图片和静态数据集。

The date in an **Expires** header is a GMT date. It can be set in PHP using this code:

Expires 头中的日期是 GMT 日期。它在 PHP 中使用如下代码设置：

```
$lifetime = 7 * 24 * 60 * 60; // 7 days, in seconds.
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

This will tell the browser to cache the file for 7 days. To set a far future **Expires** header, set the lifetime to something longer; this example tells the browser to cache the file for 10 years:

这将告诉浏览器缓存此数据 7 天。要设置一个遥远未来 Expires 头，将它的生命期设得更长，下面的例子告诉浏览器缓存文件 10 年：

```
$lifetime = 10 * 365 * 24 * 60 * 60; // 10 years, in seconds.
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

An **Expires** header is the easiest way to make sure your Ajax responses are cached on the browser. You don't have to change anything in your client-side code, and can continue to make Ajax requests normally, knowing that the browser will send the request on to the server only if the file isn't in cache. It's also easy to implement on the server side, as all languages allow you to set headers in one way or another. This is the simplest approach to ensuring your data is cached.

一个 Expires 头是确保浏览器缓存 Ajax 响应报文最简单的方法。你不需要改变客户端的任何代码，可继续正常地使用 Ajax 请求，确信浏览器只当文件不在缓存之中时才将请求发送给服务器。这在服务器端也很容易实现，所有的语言都允许你通过某种方法设置信息头。这是保证你的数据被缓存的最简单方法。

**Storing data locally 本地存储数据**

Instead of relying on the browser to handle caching, you can also do it in a more manual fashion, by storing the responses you receive from the server. This can be done by putting the response text into an object, keyed by the URL used to fetch it. Here is an example of an XHR wrapper that first checks to see whether a URL has been fetched before:

除了依赖浏览器处理缓存之外，你还可以用手工方法实现它，直接存储那些从服务器收到的响应报文。可将响应报文存放在一个对象中，以 URL 为键值索引它。这是一个 XHR 封装，它首先检查一个 URL 此前是否被取用过：

```
 var localCache = {};
function xhrRequest(url, callback) {
  // Check the local cache for this URL.
  if (localCache[url]) {
    callback.success(localCache[url]);
    return;
  }
  // If this URL wasn't found in the cache, make the request.
  var req = createXhrObject();
  req.onerror = function() {
    callback.error();
  };
  req.onreadystatechange = function() {
    if (req.readyState == 4) {
      if (req.responseText === '' || req.status == '404') {
        callback.error();
        return;
      }
      // Store the response on the local cache.
      localCache[url] = req.responseText;
      callback.success(req.responseText);
    }
  };
  req.open("GET", url, true);
  req.send(null);
}
```

Overall, setting an **Expires** header is a better solution. It's easier to do and it caches responses across page loads and sessions. But a manual cache can be useful in situations where you want to be able to programmatically expire a cache and fetch fresh data. Imagine a situation where you would like to use cached data for every request, except when the user takes an action that causes one or more of the cached responses to become invalid. In this case, removing those responses from the cache is trivial:

总的来说，设置一个 Expires 头是更好的解决方案。这比较容易，而且其缓存内容可以跨页面或者跨对话。而一个手工缓存可以用程序废止缓存内容并获取新的数据。设想一种情况，你为每个请求缓存数据，用户可能触发某些动作导致一个或多个响应报文作废。这种情况下从缓存中删除报文十分简单：

```
delete localCache['/user/friendlist/'];
delete localCache['/user/contactlist/'];
```

A local cache also works well for users browsing on mobile devices. Most of the browsers on such devices have small or nonexistent caches, and a manual cache is the best option for preventing unnecessary requests.

本地缓存也可很好地工作于移动设备上。此类设备上的浏览器缓存小或者根本不存在，手工缓存成为避免不必要请求的最佳选择。

## Know the Limitations of Your Ajax Library 了解 Ajax 库的限制

All JavaScript libraries give you access to an Ajax object, which normalizes the differences between browsers and gives you a consistent interface. Most of the time this is a very good thing, as it allows you to focus on your project rather than the details of how XHR works in some obscure browser. However, in giving you a unified interface, these libraries must also simplify the interface, because not every browser implements each feature. This prevents you from accessing the full power of **XMLHttpRequest**.

所有 JavaScript 库允许你访问一个 Ajax 对象，它屏蔽浏览器之间的差异，给你一个一致的接口。大多数情况下这非常好，因为它使你可以关注你的项目，而不是那些古怪的浏览器上 XHR 的工作细节。然而，为了给你一个统一的接口，这些库必须简化接口，因为不是所有浏览器都实现了每个功能。这使得你不能访问 XMLHttpRequest 的完整功能。

Some of the techniques we covered in this chapter can be implemented only by accessing the XHR object directly. Most notable of these is the streaming feature of multipart XHR. By listening for **readyState 3**, we can start to slice up a large response before it's completely received. This allows us to handle pieces of the response in real time, and it is one of the reasons that MXHR improves performance so much. Most JavaScript libraries, though, do not give you direct access to the **readystatechange** event. This means you must wait until the entire response is received (which may be a considerable amount of time) before you can start to use any part of it.

本章中介绍的一些技术只能通过直接访问 XHR 对象实现。值得注意的是在多部分 XHR 技术中要用到流功能。通过监听 readyState 3，我们在一个大的响应报文没有完全接收之前就开始解析它。这使我们可以实时处理报文片断，这也是 MXHR 能够大幅度提高性能的原因之一。不过大多数 JavaScript 库不允许你直接访问 readystatechange 事件。这意味着你必须等待整个响应报文接收完（可能是一个相当长的时间）然后你才能使用它。

Using the **XMLHttpRequest** object directly is not as daunting as it seems. A few quirks aside, the most recent versions of all major browsers support the **XMLHttpRequest** object in the same way, and all offer access to the different **readyStates**. You can support older versions of IE with just a few more lines of code. Here is an example of a function that will return an XHR object, which you can then interact with directly (this is a modified version of what the YUI 2 Connection Manager uses):

直接使用 XMLHttpRequest 对象并非像它看起来那么恐怖。除一些个别行为之外，所有主流浏览器的最新版本均以同样方式支持 XMLHttpRequest 对象，均可访问不同的 readyStates。如果你要支持老版本的 IE，只需要多加几行代码。下面例子中的函数返回一个 XHR 对象，你可以直接调用（这是 YUI 2 连接管理器中修改后的版本）：

```
function createXhrObject() {
  var msxml_progid = [
    'MSXML2.XMLHTTP.6.0',
    'MSXML3.XMLHTTP',
    'Microsoft.XMLHTTP', // Doesn't support readyState 3.
    'MSXML2.XMLHTTP.3.0', // Doesn't support readyState 3.
  ];
  var req;
  try {
    req = new XMLHttpRequest(); // Try the standard way first.
  }
  catch(e) {
    for (var i = 0, len = msxml_progid.length; i < len; ++i) {
      try {
        req = new ActiveXObject(msxml_progid[i]);
        break;
      }
      catch(e2) { }
    }
  }
  finally {
    return req;
  }
}
```

This will first try the versions of **XMLHttp** that do support **readyState 3**, and then fall back to the ones that don't in case those versions aren't available.

它首先尝试支持 readyState 3 的 XMLHttpRequest，然后回落到那些不支持此状态的版本中。

Interacting directly with the XHR object also reduces the amount of function overhead, further improving performance. Just beware that by forgoing the use of an Ajax library, you may encounter some problems with older and more obscure browsers.

直接操作 XHR 对象减少了函数开销，进一步提高了性能。只是放弃使用 Ajax 库，你可能会在古怪的浏览器上遇到一些问题。

## Summary 总结

High-performance Ajax consists of knowing the specific requirements of your situation and selecting the correct data format and transmission technique to match.

高性能 Ajax 包括：知道你项目的具体需求，选择正确的数据格式和与之相配的传输技术。

As data formats, plain text and HTML are highly situational, but they can save CPU cycles on the client side. XML is widely available and supported almost everywhere, but it is extremely verbose and slow to parse. JSON is lightweight and quick to parse (when treated as native code and not a string), and almost as interoperable as XML. Character-delimited custom formats are extremely lightweight and the quickest to parse for large datasets, but may take additional programming effort to format on the server side and parse on the client side.

作为数据格式，纯文本和 HTML 是高度限制的，但它们可节省客户端的 CPU 周期。XML 被广泛应用普遍支持，但它非常冗长且解析缓慢。JSON 是轻量级的，解析迅速（作为本地代码而不是字符串），交互性与 XML 相当。字符分隔的自定义格式非常轻量，在大量数据集解析时速度最快，但需要编写额外的程序在服务器端构造格式，并在客户端解析。

When requesting data, XHR gives you the most control and flexibility when pulling from the page's domain, though it treats all incoming data as a string, potentially slowing down the parse times. Dynamic script tag insertion, on the other hand, allows for cross-domain requests and native execution of JavaScript and JSON, though it offers a less robust interface and cannot read headers or response codes. Multipart XHR can be used to reduce the number of requests, and can handle different file types in a single response, though it does not cache the resources received. When sending data, image beacons are a simple and efficient approach. XHR can also be used to send large amounts of data in a POST.

当从页面域请求数据时，XHR 提供最完善的控制和灵活性，尽管它将所有传入数据视为一个字符串，这有可能降低解析速度。另一方面，动态脚本标签插入技术允许跨域请求和本地运行 JavaScript 和 JSON，虽然它的接口不够安全，而且不能读取信息头或响应报文代码。多部分 XHR 可减少请求的数量，可在一

次响应中处理不同的文件类型，尽管它不能缓存收到的响应报文。当发送数据时，图像灯标是最简单和最有效的方法。XHR 也可用 POST 方法发送大量数据。

In addition to these formats and transmission techniques, there are several guidelines that will help your Ajax appear to be faster:

除这些格式和传输技术之外，还有一些准则有助于进一步提高 Ajax 的速度：

• Reduce the number of requests you make, either by concatenating JavaScript and CSS files, or by using MXHR.

减少请求数量，可通过 JavaScript 和 CSS 文件打包，或者使用 MXHR。

• Improve the perceived loading time of your page by using Ajax to fetch less important files after the rest of the page has loaded.

缩短页面的加载时间，在页面其它内容加载之后，使用 Ajax 获取少量重要文件。

• Ensure your code fails gracefully and can handle problems on the server side.

确保代码错误不要直接显示给用户，并在服务器端处理错误。

• Know when to use a robust Ajax library and when to write your own low-level Ajax code.

学会何时使用一个健壮的 Ajax 库，何时编写自己的底层 Ajax 代码。

Ajax offers one of the largest areas for potential performance improvements on your site, both because so many sites use asynchronous requests heavily and because it can offer solutions to problems that aren't even related to it, such as having too many resources to load. Creative use of XHR can be the difference between a sluggish, uninviting page and one that responds quickly and efficiently; it can be the difference between a site that users hate to interact with and one that they love.

Ajax 是提升你网站潜在性能之最大的改进区域之一，因为很多网站大量使用异步请求，又因为它提供了许多不相关问题的解决方案，这些问题诸如，需要加载太多资源。对 XHR 的创造性应用是如此的与众

不同，它不是呆滞不友好的界面，而是响应迅速且高效的代名词；它不会引起用户的憎恨，谁见了它都会爱上它。

# 第八章　Programming Practices　编程实践

Every programming language has pain points and inefficient patterns that develop over time. The appearance of these traits occurs as people migrate to the language and start pushing its boundaries. Since 2005, when the term "Ajax" emerged, web developers have pushed JavaScript and the browser further than it was ever pushed before. As a result, some very specific patterns emerged, both as best practices and as suboptimal ones. These patterns arise because of the very nature of JavaScript on the Web.

每种编程语言都有痛点，而且低效模式随着时间的推移不断发展。其原因在于，越来越多的人们开始使用这种语言，不断扩种它的边界。自 2005 年以来，当术语“Ajax”出现时，网页开发者对 JavaScript 和浏览器的推动作用远超过以往。其结果是出现了一些非常具体的模式，即有优秀的做法也有糟糕的做法。这些模式的出现，是因为网络上 JavaScript 的性质决定的。

## Avoid Double Evaluation　避免二次评估

JavaScript, like many scripting languages, allows you to take a string containing code and execute it from within running code. There are four standard ways to accomplish this: **eval_r()**, the **Function()** constructor, **setTimeout()**, and **setInterval()**. Each of these functions allows you to pass in a string of JavaScript code and have it executed. Some examples:

JavaScript 与许多脚本语言一样，允许你在程序中获取一个包含代码的字符串然后运行它。有四种标准方法可以实现：eval_r()，Function()构造器，setTimeout()和 setInterval()。每个函数允许你传入一串 JavaScript 代码，然后运行它。例如：

```
var num1 = 5,
num2 = 6,
//eval_r() evaluating a string of code
result = eval_r("num1 + num2"),
//Function() evaluating strings of code
```

```
sum = new Function("arg1", "arg2", "return arg1 + arg2");
//setTimeout() evaluating a string of code
setTimeout("sum = num1 + num2", 100);
//setInterval() evaluating a string of code
setInterval("sum = num1 + num2", 100);
```

Whenever you're evaluating JavaScript code from within JavaScript code, you incur a double evaluation penalty. This code is first evaluated as normal, and then, while executing, another evaluation happens to execute the code contained in a string. Double evaluation is a costly operation and takes much longer than if the same code were included natively.

当你在 JavaScript 代码中执行（另一段）JavaScript 代码时，你付出二次评估的代价。此代码首先被评估为正常代码，然后在执行过程中，运行字符串中的代码时发生另一次评估。二次评估是一项昂贵的操作，与直接包含相应代码相比将占用更长时间。

As a point of comparison, the time it takes to access an array item varies from browser to browser but varies far more dramatically when the array item is accessed using **eval_r()**. For example:

作为一个比较点，不同浏览器上访问一个数组项所占用的时间各有不同，但如果使用 eval_r()访问其结果将大相径庭。例如：

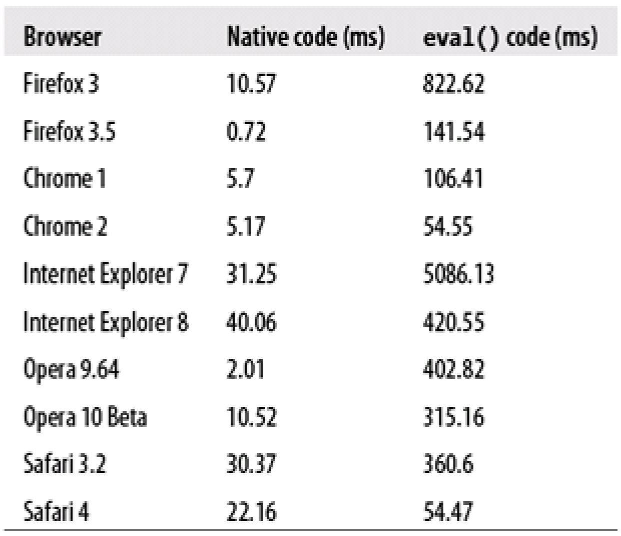
```
//faster
var item = array[0];
//slower
var item = eval_r("array[0]");
```

The difference across browsers becomes dramatic if 10,000 array items are read using eval_r() instead of native code. Table 8-1 shows the different times for this operation.

如果使用 eval_r()代替直接代码访问 10'000 个数组项，在不同浏览器上的差异非常巨大。表 8-1 显示了这些操作所用的时间。

Table 8-1. Speed comparison of native code versus eval_r() for accessing 10,000 array items

表 8-1　直接代码与 eval_r()访问 10'000 个数组项的速度比较



| Browser | Native code (ms) | eval() code (ms) |
|---|---|---|
| Firefox 3 | 10.57 | 822.62 |
| Firefox 3.5 | 0.72 | 141.54 |
| Chrome 1 | 5.7 | 106.41 |
| Chrome 2 | 5.17 | 54.55 |
| Internet Explorer 7 | 31.25 | 5086.13 |
| Internet Explorer 8 | 40.06 | 420.55 |
| Opera 9.64 | 2.01 | 402.82 |
| Opera 10 Beta | 10.52 | 315.16 |
| Safari 3.2 | 30.37 | 360.6 |
| Safari 4 | 22.16 | 54.47 |

This dramatic difference in array item access time is due to the creation of a new interpreter/compiler instance each time **eval_r()** is called. The same process occurs for **Function()**, **setTimeout()**, and **setInterval()**, automatically making code execution slower.

访问数组项时间上的巨大差异，是因为每次调用 eval_r()时要创建一个新的解释/编译实例。同样的过程也发生在 Function()，setTimeout()和 setInterval()上，自动使代码执行速度变慢。

Most of the time, there is no need to use **eval_r()** or **Function()**, and it's best to avoid them whenever possible. For the other two functions, **setTimeout()** and **setInterval()**, it's recommended to pass in a function as the first argument instead of a string. For example:

大多数情况下，没必要使用 eval_r()或 Function()，如果可能的话，尽量避免使用它们。至于另外两个函数，setTimeout()和 setInterval()，建议第一个参数传入一个函数而不是一个字符串。例如：

```
setTimeout(function(){
  sum = num1 + num2;
}, 100);
setInterval(function(){
  sum = num1 + num2;
}, 100);
```

Avoiding double evaluation is key to achieving the most optimal JavaScript runtime performance possible.

避免二次评估是实现最优化的 JavaScript 运行时性能的关键。

**Use Object/Array Literals** 使用对象/数组直接量

There are multiple ways to create objects and arrays in JavaScript, but nothing is faster than creating object and array literals. Without using literals, typical object creation and assignment looks like this:

在 JavaScript 中有多种方法创建对象和数组，但没有什么比创建对象和数组直接量更快了。如果不使用直接量，典型的对象创建和赋值是这样的：

```
//create an object
var myObject = new Object();
myObject.name = "Nicholas";
myObject.count = 50;
myObject.flag = true;
myObject.pointer = null;
//create an array
var myArray = new Array();
myArray[0] = "Nicholas";
myArray[1] = 50;
myArray[2] = true;
myArray[3] = null;
```

Although there is technically nothing wrong with this approach, literals are evaluated faster. As an added bonus, literals take up less space in your code, so the overall file size is smaller. The previous code can be rewritten using literals in the following way:

虽然在技术上这种做法没有什么不对，直接量赋值很快。作为一个额外的好处，直接量在你的代码中占用较少空间，所以整个文件尺寸可以更小。上面的代码可用直接量重写为下面的样式：

```
 //create an object
var myObject = {
  name: "Nicholas",
  count: 50,
  flag: true,
  pointer: null
};
//create an array
var myArray = ["Nicholas", 50, true, null];
```

The end result of this code is the same as the previous version, but it is executed faster in almost all browsers (Firefox 3.5 shows almost no difference). As the number of object properties and array items increases, so too does the benefit of using literals.

此代码的效果与前面的版本相同，但在几乎所有浏览器上运行更快（在 Firefox 3.5 上几乎没区别）。随着对象属性和数组项数的增加，使用直接量的好处也随之增加。

**Don't Repeat Work** 不要重复工作

One of the primary performance optimization techniques in computer science overall is work avoidance. The concept of work avoidance really means two things: don't do work that isn't required, and don't repeat work that has already been completed. The first part is usually easy to identify as code is being refactored. The second part—not repeating work—is usually more difficult to identify because work may be repeated in any number of places and for any number of reasons.

在计算机科学领域最重要的性能优化技术之一是避免工作。避免工作的概念实际上意味着两件事：不要做不必要的工作，不要重复做已经完成的工作。第一部分通常认为代码应当重构。第二部分——不要重复工作——通常难以确定，因为工作可能因为各种原因而在很多地方被重复。

Perhaps the most common type of repeated work is browser detection. A lot of code has forks based on the browser's capabilities. Consider event handler addition and removal as an example. Typical cross-browser code for this purpose looks like the following:

也许最常见的重复工作类型是浏览器检测。大量代码依赖于浏览器的功能。以事件句柄的添加和删除为例，典型的跨浏览器代码如下：

```
function addHandler(target, eventType, handler){
  if (target.addEventListener){ //DOM2 Events
    target.addEventListener(eventType, handler, false);
  } else { //IE
    target.attachEvent("on" + eventType, handler);
  }
}
function removeHandler(target, eventType, handler){
  if (target.removeEventListener){ //DOM2 Events
    target.removeEventListener(eventType, handler, false);
  } else { //IE
    target.detachEvent("on" + eventType, handler);
  }
}
```

The code checks for DOM Level 2 Events support by testing for **addEventListener()** and **removeEventListener()**, which is supported by all modern browsers except Internet Explorer. If these methods don't exist on the **target**, then IE is assumed and the IE-specific methods are used.

此代码通过测试 addEventListener()和 removeEventListener()检查 DOM 级别 2 的事件支持情况，它能够被除 Internet Explorer 之外的所有现代浏览器所支持。如果这些方法不存在于 target 中，那么就认为当前浏览器是 IE，并使用 IE 特有的方法。

At first glance, these functions look fairly optimized for their purpose. The hidden performance issue is in the repeated work done each time either function is called. Each time, the same check is made to see whether a certain method is present. If you assume that the only values for **target** are actually DOM objects, and that the user doesn't magically change his browser while the page is loaded, then this evaluation is repetitive. If **addEventListener()** was present on the first call to **addHandler()** then it's going to be present for each subsequent call. Repeating the same work with every call to a function is wasteful, and there are a couple of ways to avoid it.

乍一看，这些函数为实现它们的目的已经足够优化。隐藏的性能问题在于每次函数调用时都执行重复工作。每一次，都进行同样的检查，看看某种方法是否存在。如果你假设 target 唯一的值就是 DOM 对象，而且用户不可能在页面加载时魔术般地改变浏览器，那么这种判断就是重复的。如果 addHandler()一上来就调用 addEventListener()那么每个后续调用都要出现这句代码。在每次调用中重复同样的工作是一种浪费，有多种办法避免这一点。

## Lazy Loading 延迟加载

The first way to eliminate work repetition in functions is through lazy loading. Lazy loading means that no work is done until the information is necessary. In the case of the previous example, there is no need to determine which way to attach or detach event handlers until someone makes a call to the function. Lazy-loaded versions of the previous functions look like this:

第一种消除函数中重复工作的方法称作延迟加载。延迟加载意味着在信息被使用之前不做任何工作。在前面的例子中，不需要判断使用哪种方法附加或分离事件句柄，直到有人调用此函数。使用延迟加载的函数如下：

```
function addHandler(target, eventType, handler){
  //overwrite the existing function
  if (target.addEventListener){ //DOM2 Events
    addHandler = function(target, eventType, handler){
      target.addEventListener(eventType, handler, false);
    };
  } else { //IE
    addHandler = function(target, eventType, handler){
      target.attachEvent("on" + eventType, handler);
    };
  }
  //call the new function
  addHandler(target, eventType, handler);
}
function removeHandler(target, eventType, handler){
  //overwrite the existing function
  if (target.removeEventListener){ //DOM2 Events
    removeHandler = function(target, eventType, handler){
      target.addEventListener(eventType, handler, false);
    };
  } else { //IE
    removeHandler = function(target, eventType, handler){
      target.detachEvent("on" + eventType, handler);
    };
  }
  //call the new function
  removeHandler(target, eventType, handler);
}
```

These two functions implement a lazy-loading pattern. The first time either method is called, a check is made to determine the appropriate way to attach or detach the event handler. Then, the original function is overwritten with a new function that contains just the appropriate course of action. The last step during that first function call is to execute the new function with the original arguments. Each subsequent call to **addHandler()** or **removeHandler()** avoids further detection because the detection code was overwritten by a new function.

这两个函数依照延迟加载模式实现。这两个方法第一次被调用时，检查一次并决定使用哪种方法附加或分离事件句柄。然后，原始函数就被包含适当操作的新函数覆盖了。最后调用新函数并将原始参数传给它。以后再调用 addHandler()或者 removeHandler()时不会再次检测，因为检测代码已经被新函数覆盖了。

Calling a lazy-loading function always takes longer the first time because it must run the detection and then make a call to another function to accomplish the task. Subsequent calls to the same function, however, are much faster since they have no detection logic. Lazy loading is best used when the function won't be used immediately on the page.

调用一个延迟加载函数总是在第一次使用较长时间，因为它必须运行检测然后调用另一个函数以完成任务。但是，后续调用同一函数将快很多，因为不再执行检测逻辑了。延迟加载适用于函数不会在页面上立即被用到的场合。

**Conditional Advance Loading** 条件预加载

An alternative to lazy-loading functions is conditional advance loading, which does the detection upfront, while the script is loading, instead of waiting for the function call. The detection is still done just once, but it comes earlier in the process. For example:

除延迟加载之外的另一种方法称为条件预加载，它在脚本加载之前提前进行检查，而不等待函数调用。这样做检测仍只是一次，但在此过程中来的更早。例如：

```
var addHandler = document.body.addEventListener ?
  function(target, eventType, handler){
    target.addEventListener(eventType, handler, false);
  }:
  function(target, eventType, handler){
    target.attachEvent("on" + eventType, handler);
  };
var removeHandler = document.body.removeEventListener ?
  function(target, eventType, handler){
    target.removeEventListener(eventType, handler, false);
  }:
  function(target, eventType, handler){
    target.detachEvent("on" + eventType, handler);
  };
```

This example checks to see whether **addEventListener()** and **removeEventListener()** are present and then uses that information to assign the most appropriate function. The ternary operator returns the DOM Level 2 function if these methods are present and otherwise returns the IE-specific function. The result is that all calls to **addHandler()** and **removeHandler()** are equally fast, as the detection cost occurs upfront.

这个例子检查 addEventListener()和 removeEventListener()是否存在，然后根据此信息指定最合适的函数。三元操作符返回 DOM 级别 2 的函数，如果它们存在的话，否则返回 IE 特有的函数。然后，调用 addHandler() 和 removeHandler()同样很快，虽然检测功能提前了。

Conditional advance loading ensures that all calls to the function take the same amount of time. The trade-off is that the detection occurs as the script is loading rather than later. Advance loading is best to use when a function is going to be used right away and then again frequently throughout the lifetime of the page.

条件预加载确保所有函数调用时间相同。其代价是在脚本加载时进行检测。预加载适用于一个函数马上就会被用到，而且在整个页面生命周期中经常使用的场合。

## Use the Fast Parts 使用速度快的部分

Even though JavaScript is often blamed for being slow, there are parts of the language that are incredibly fast. This should come as no surprise, since JavaScript engines are built in lower-level languages and are therefore compiled. Though it's easy to blame the engine when JavaScript appears slow, the engine is typically the fastest part of the process; it's your code that is actually running slowly. There are parts of the engine that are much faster than others because they allow you to bypass the slow parts.

虽然 JavaScript 经常被指责缓慢，然而此语言的某些部分具有难以置信的快速。这不足为奇因为 JavaScript 引擎由低级语言构建。虽然 JavaScript 速度慢很容易被归咎于引擎，然而引擎通常是处理过程中最快的部分，实际上速度慢的是你的代码。引擎的某些部分比其它部分快很多，因为它们允许你绕过速度慢的部分。

### Bitwise Operators 位操作运算符

Bitwise operators are one of the most frequently misunderstood aspects of JavaScript. General opinion is that developers don't understand how to use these operators and frequently mistake them for their Boolean equivalents. As a result, bitwise operators are used infrequently in JavaScript development, despite their advantages.

位操作运算符是 JavaScript 中经常被误解的内容之一。一般的看法是，开发者不知道如何使用这些操作符，经常在布尔表达式中误用。结果导致 JavaScript 开发中不常用位操作运算符，尽管它们具有优势。

JavaScript numbers are all stored in IEEE-754 64-bit format. For bitwise operations, though, the number is converted into a signed 32-bit representation. Each operator then works directly on this 32-bit representation to achieve a result. Despite the conversion, this process is incredibly fast when compared to other mathematical and Boolean operations in JavaScript.

JavaScript 中的数字按照 IEEE-754 标准 64 位格式存储。在位运算中，数字被转换为有符号 32 位格式。每种操作均直接操作在这个 32 位数上实现结果。尽管需要转换，这个过程与 JavaScript 中其他数学和布尔运算相比还是非常快。

If you're unfamiliar with binary representation of numbers, JavaScript makes it easy to convert a number into a string containing its binary equivalent by using the **toString()** method and passing in the number 2. For example:

如果你对数字的二进制表示法不熟悉，JavaScript 可以很容易地将数字转换为字符串形式的二进制表达式，通过使用 toString()方法并传入数字 2（做参数）。例如：

```
 var num1 = 25,
num2 = 3;
alert(num1.toString(2)); //"11001"
alert(num2.toString(2)); // "11"
```

Note that this representation omits the leading zeros of a number.

请注意，该表达式消隐了数字高位的零。

There are four bitwise logic operators in JavaScript:

JavaScript 中有四种位逻辑操作符：

Bitwise AND  位与

Returns a number with a 1 in each bit where both numbers have a 1

两个操作数的位都是 1，结果才是 1

Bitwise OR　位或

Returns a number with a 1 in each bit where either number has a 1

有一个操作数的位是 1，结果就是 1

Bitwise XOR　位异或

Returns a number with a 1 in each bit where exactly one number has a 1

两个位中只有一个 1，结果才是 1

Bitwise NOT　位非

Returns 1 in each position where the number has a 0 and vice versa

遇 0 返回 1，反之亦然

These operators are used as follows:

这些操作符用法如下：

```
//bitwise AND
var result1 = 25 & 3; //1
alert(result.toString(2)); //"1"
//bitwise OR
var result2 = 25 | 3; //27
alert(resul2.toString(2)); //"11011"
//bitwise XOR
var result3 = 25 ^ 3; //26
alert(resul3.toString(2)); //"11000"
//bitwise NOT
var result = ~25; //-26
alert(resul2.toString(2)); //"-11010"
```

There are a couple of ways to use bitwise operators to speed up your JavaScript. The first is to use bitwise operations instead of pure mathematical operations. For example, it's common to alternate table row colors by calculating the modulus of 2 for a given number, such as:

有许多方法可以使用位运算符提高 JavaScript 的速度。首先可以用位运算符替代纯数学操作。例如，通常采用对 2 取模运算实现表行颜色交替显示，例如：

```
for (var i=0, len=rows.length; i < len; i++){
  if (i % 2) {
    className = "even";
  } else {
    className = "odd";
  }
  //apply class
}
```

Calculating mod 2 requires the number to be divided by 2 to determine the remainder. If you were to look at the underlying 32-bit representation of numbers, a number is even if its first bit is 0 and is odd if its first bit is 1. This can easily be determined by using a bitwise AND operation on a given number and the number 1. When the number is even, the result of bitwise AND 1 is 0; when the number is odd, the result of bitwise AND 1 is 1. That means the previous code can be rewritten as follows:

计算对 2 取模，需要用这个数除以 2 然后查看余数。如果你看到 32 位数字的底层（二进制）表示法，你会发现偶数的最低位是 0，奇数的最低位是 1。如果此数为偶数，那么它和 1 进行位与操作的结果就是 0；如果此数为奇数，那么它和 1 进行位与操作的结果就是 1。也就是说上面的代码可以重写如下：

```
for (var i=0, len=rows.length; i < len; i++){
  if (i & 1) {
    className = "odd";
  } else {
    className = "even";
  }
```

```
    //apply class
}
```

Although the code change is small, the bitwise AND version is up to 50% faster than the original (depending on the browser).

虽然代码改动不大，但位与版本比原始版本快了 50%（取决于浏览器）。

The second way to use bitwise operators is a technique known as a bitmask. Bitmasking is a popular technique in computer science when there are a number of Boolean options that may be present at the same time. The idea is to use each bit of a single number to indicate whether or not the option is present, effectively turning the number into an array of Boolean flags. Each option is given a value equivalent to a power of 2 so that the mask works. For example:

第二种使用位操作的技术称作位掩码。位掩码在计算机科学中是一种常用的技术，可同时判断多个布尔选项，快速地将数字转换为布尔标志数组。掩码中每个选项的值都等于 2 的幂。例如：

```
var OPTION_A = 1;
var OPTION_B = 2;
var OPTION_C = 4;
var OPTION_D = 8;
var OPTION_E = 16;
```

With the options defined, you can create a single number that contains multiple settings using the bitwise OR operator:

通过定义这些选项，你可以用位或操作创建一个数字来包含多个选项：

```
var options = OPTION_A | OPTION_C | OPTION_D;
```

You can then check whether a given option is available by using the bitwise AND operator. The operation returns 0 if the option isn't set and 1 if the option is set:

你可以使用位与操作检查一个给定的选项是否可用。如果该选项未设置则运算结果为 0，如果设置了那么运算结果为 1：

```
//is option A in the list?
if (options & OPTION_A){
  //do something
}
//is option B in the list?
if (options & OPTION_B){
  //do something
}
```

Bitmask operations such as this are quite fast because, as mentioned previously, the work is happening at a lower level of the system. If there are a number of options that are being saved together and checked frequently, bitmasks can help to speed up the overall approach.

像这样的位掩码操作非常快，正因为前面提到的原因，操作发生在系统底层。如果许多选项保存在一起并经常检查，位掩码有助于加快整体性能。

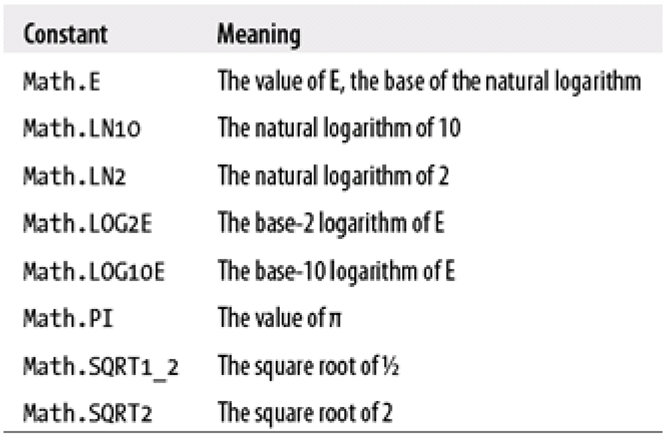
**Native Methods** 原生方法

No matter how optimal your JavaScript code is, it will never be faster than the native methods provided by the JavaScript engine. The reason for this is simple: the native parts of JavaScript—those already present in the browser before you write a line of code—are all written in a lower-level language such as C++. That means these methods are compiled down to machine code as part of the browser and therefore don't have the same limitations as your JavaScript code.

无论你怎样优化 JavaScript 代码，它永远不会比 JavaScript 引擎提供的原生方法更快。其原因十分简单：JavaScript 的原生部分——在你写代码之前它们已经存在于浏览器之中了——都是用低级语言写的，诸如 C++。这意味着这些方法被编译成机器码，作为浏览器的一部分，不像你的 JavaScript 代码那样有那么多限制。

A common mistake of inexperienced JavaScript developers is to perform complex mathematical operations in code when there are better performing versions available on the built-in **Math** object. The **Math** object contains properties and methods designed to make mathematical operations easier. There are several mathematical constants available:

经验不足的 JavaScript 开发者经常犯的一个错误是在代码中进行复杂的数学运算，而没有使用内置 Math 对象中那些性能更好的版本。Math 对象包含专门设计的属性和方法，使数学运算更容易。这里是一些数学常数：



| Constant | Meaning |
|---|---|
| `Math.E` | The value of E, the base of the natural logarithm |
| `Math.LN10` | The natural logarithm of 10 |
| `Math.LN2` | The natural logarithm of 2 |
| `Math.LOG2E` | The base-2 logarithm of E |
| `Math.LOG10E` | The base-10 logarithm of E |
| `Math.PI` | The value of π |
| `Math.SQRT1_2` | The square root of ½ |
| `Math.SQRT2` | The square root of 2 |

Each of these values is precalculated, so there is no need for you to calculate them yourself. There are also methods to handle mathematical calculations:

这里的每个数值都是预计算好的，所以你不需要自己来计算它们。还有一些处理数学运算的方法：

| Method | Meaning |
|---|---|
| `Math.abs(num)` | The absolute value of *num* |
| `Math.exp(num)` | Math.E$^{num}$ |
| `Math.log(num)` | The logarithm of *num* |
| `Math.pow(num,power)` | $num^{power}$ |
| `Math.sqrt(num)` | The square root of *num* |
| `Math.acos(x)` | The arc cosine of *x* |
| `Math.asin(x)` | The arc sine of *x* |
| `Math.atan(x)` | The arc tangent of *x* |
| `Math.atan2(y,x)` | The arc tangent of *y/x* |
| `Math.cos(x)` | The cosine of *x* |
| `Math.sin(x)` | The sine of *x* |
| `Math.tan(x)` | The tangent of *x* |

Using these methods is faster than recreating the same functionality in JavaScript code. Whenever you need to perform complex mathematical calculations, look to the **Math** object first.

使用这些函数比同样功能的 JavaScript 代码更快。当你需要进行复杂数学计算时，首先查看 Math 对象。

Another example is the Selectors API, which allows querying of a DOM document using CSS selectors. CSS queries were implemented natively in JavaScript and truly popularized by the jQuery JavaScript library. The jQuery engine is widely considered the fastest engine for CSS querying, but it is still much slower than the native methods. The native **querySelector()** and **querySelectorAll()** methods complete their tasks, on average, in 10% of the time it takes for JavaScript-based CSS querying. Most JavaScript libraries have now moved to use the native functionality when available to speed up their overall performance.

另一个例子是选择器 API，可以像使用 CSS 选择器那样查询 DOM 文档。CSS 查询被 JavaScript 原生实现并通过 jQuery 这个 JavaScript 库推广开来。jQuery 引擎被认为是最快的 CSS 查询引擎，但是它仍比原生方法慢。原生的 querySelector()和 querySelectorAll()方法完成它们的任务时，平均只需要基于 JavaScript 的 CSS 查询 10%的时间。大多数 JavaScript 库已经使用了原生函数以提高它们的整体性能。

Always use native methods when available, especially for mathematical calculations and DOM operations. The more work that is done with compiled code, the faster your code becomes.

当原生方法可用时，尽量使用它们，尤其是数学运算和 DOM 操作。用编译后的代码做越多的事情，你的代码就会越快。

## Summary 总结

JavaScript presents some unique performance challenges related to the way you organize your code. As web applications have become more advanced, containing more and more lines of JavaScript to function, some patterns and antipatterns have emerged. Some programming practices to keep in mind:

JavaScript 提出了一些独特的性能挑战，关系到你组织代码的方法。网页应用变得越来越高级，包含的 JavaScript 代码越来越多，出现了一些模式和反模式。请牢记以下编程经验：

• Avoid the double evaluation penalty by avoiding the use of **eval_r()** and the **Function()** constructor. Also, pass functions into **setTimeout()** and **setInterval()** instead of strings.

通过避免使用 eval_r()和 Function()构造器避免二次评估。此外，给 setTimeout()和 setInterval()传递函数参数而不是字符串参数。

• Use object and array literals when creating new objects and arrays. They are created and initialized faster than nonliteral forms.

创建新对象和数组时使用对象直接量和数组直接量。它们比非直接量形式创建和初始化更快。

• Avoid doing the same work repeatedly. Use lazy loading or conditional advance loading when browser-detection logic is necessary.

避免重复进行相同工作。当需要检测浏览器时，使用延迟加载或条件预加载。

• When performing mathematical operations, consider using bitwise operators that work directly on the underlying representation of the number.

当执行数学远算时，考虑使用位操作，它直接在数字底层进行操作。

• Native methods are always faster than anything you can write in JavaScript. Use native methods whenever available.

原生方法总是比 JavaScript 写的东西要快。尽量使用原生方法。

As with many of the techniques and approaches covered in this book, you will see the greatest performance gains when these optimizations are applied to code that is run frequently.

本书涵盖了很多技术和方法，如果将这些优化应用在那些经常运行的代码上，你将会看到巨大的性能提升。

# 第九章 Building and Deploying High-Performance JavaScript Applications

# 创建并部署高性能 JavaScript 应用程序

According to a 2007 study by Yahoo!'s Exceptional Performance team, 40%–60% of Yahoo!'s users have an empty cache experience, and about 20% of all page views are done with an empty cache (http://yuiblog.com/blog/2007/01/04/performance-research-part-2/). In addition, another more recent study by the Yahoo! Search team, which was independently confirmed by Steve Souders of Google, indicates that roughly 15% of the content delivered by large websites in the United States is served uncompressed.

根据 Yahoo!卓越性能团队在 2007 年进行的研究，40%-60%的 Yahoo!用户没有使用缓存的经验，大约 20%页面视图不使用缓存（http://yuiblog.com/blog/2007/01/04/performance-research-part-2/）。另外，由 Yahoo!研究小组发现，并由 Google 的 Steve Souders 所证实的一项最新研究表明，大约 15%的美国大型网站所提供的内容没有压缩。

These facts emphasize the need to make sure that JavaScript-based web applications are delivered as efficiently as possible. While part of that work is done during the design and development cycles, the build and deployment phase is also essential and often overlooked. If care is not taken during this crucial phase, the performance of your application will suffer, no matter how much effort you've put into making it faster.

这些事实强调有必要确保那些基于 JavaScript 的网页应用尽量高效地发布。虽然部分工作在设计开发过程中已经完成，但构建和部署过程也很重要且往往被忽视。如果在这个关键过程中不够小心，你应用程序的性能将受到影响，无论你怎样努力使它更快。

The purpose of this chapter is to give you the necessary knowledge to efficiently assemble and deploy a JavaScript-based web application. A number of concepts are illustrated using Apache Ant, a Java-based build tool that has quickly become an industry standard for building applications for the Web. Toward the end of the chapter, a custom agile build tool written in PHP5 is presented as an example.

本章的目的是给你必要的知识，有效地组织并部署基于 JavaScript 的 Web 应用程序。一些概念使用 Apache Ant 进行说明，它是一个基于 Java 的创建工具，并很快成为开发网页应用程序的工业标准。在本章末尾，给出了一个用 PHP5 写的定制灵活的开发工具的例子。

## Apache Ant

Apache Ant (http://ant.apache.org/) is a tool for automating software build processes. It is similar to **make**, but is implemented in Java and uses XML to describe the build process, whereas **make** uses its own Makefile format. Ant is a project of the Apache Software Foundation (http://www.apache.org/licenses/).

Apache Ant（http://ant.apache.org/）是一个自动构建软件的工具。它类似于 make，但在 Java 中实现，并使用 XML 来描述生成过程，而 make 使用它自己的 Makefile 文件格式。Ant 是 Apache 软件基金会的一个项目：（http://www.apache.org/licenses/）。

The main benefit of Ant over **make** and other tools is its portability. Ant itself is available on many different platforms, and the format of Ant's build files is platform independent.

Ant 与 make 等其他工具相比，优势在于它的可移植性。Ant 本身可用在许多不同平台上，Ant 开发文件的格式与平台无关。

An Ant build file is written in XML and named build.xml by default. Each build file contains exactly one project and at least one target. An Ant target can depend on other targets.

默认的 Ant 开发文件为 XMl 格式的 build.xml。每个开发文件只包含一个项目和至少一个目标体。一个 Ant 目标体可依赖于其他目标体。

Targets contain task elements: actions that are executed atomically. Ant comes with a great number of built-in tasks, and optional tasks can be added if needed. Also, custom tasks can be developed in Java for use in an Ant build file.

目标体包含任务元素是一些自动运行的动作。Ant 配有大量内置任务，如果需要还可以添加可选任务。此外，Ant 创建文件中用到的自定义任务可用 Java 开发。

A project can have a set of properties, or variables. A property has a name and a value. It can be set from within the build file using the **property** task, or might be set outside of Ant. A property can be evaluated by placing its name between **${** and **}**.

一个项目可有一个属性或变量的集合。一个属性有一个名字和一个值。它可以在开发文件中使用 property 任务设置，或者在 Ant 之外设置。引用属性的方法是：将属性名放在${和}之间。

The following is an example build file. Running the default target (**dist**) compiles the Java code contained in the source directory and packages it as a JAR archive.

下面是一个开发文件的例子。运行默认目标（dist）编译源码目录中的 Java 代码并封装为一个 JAR 文档。

```
 <?xml version="1.0" encoding="UTF-8"?>
<project name="MyProject" default="dist" basedir=".">


  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>


  <target name="init">
```

```
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>


  <target name="compile" depends="init" description="compile the source">
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>


  <target name="dist" depends="compile" description="generate the distribution">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>
    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>


  <target name="clean" description="clean up">
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>


</project>
```

Although Apache Ant is used to illustrate the core concepts of this chapter, many other tools are available to build web applications. Among them, it is worth noting that Rake (http://rake.rubyforge.org/) has been gaining popularity in recent years. Rake is a Rubybased build program with capabilities similar to make. Most notably, Rakefiles (Rake's version of Makefiles) are written using standard Ruby syntax, and are therefore platform-independent.

虽然这里使用 Apache Ant 来说明本章的核心概念，还是有很多其它工具可用于开发网页应用程序。其中，值得一提的是 Rake（http://rake.rubyforge.org/）已在最近几年获得普及。最值得注意的是，Rakefile（Rake 版的 Makefile）使用标准 Ruby 语法书写，因此具有平台无关性。

## Combining JavaScript Files 合并 JavaScript 文件

According to Yahoo!'s Exceptional Performance team, the first and probably most important guideline for speeding up your website, especially for first-time visitors, is to reduce the number of HTTP requests required to render the page (http://yuiblog.com/blog/2006/11/28/performance-research-part-1/). This is where you should start looking for optimizations because combining assets usually requires a fairly small amount of work and has the greatest potential benefit for your users.

根据 Yahoo!卓越性能团队的研究，第一个也是最重要的提高网站速度的准则，特别针对那些第一次访问网站的用户，是减少渲染页面所需的 HTTP 请求的数量（http://yuiblog.com/blog/2006/11/28/performance-research-part-1/）。这是你优化工作的入手点，因为合并资源通常能够以相当少的工作为用户赢得最大的潜在利益。

Most modern websites use several JavaScript files: usually a small library, which contains a set of utilities and controls to simplify the development of richly interactive web applications across multiple browsers, and some site-specific code, split into several logical units to keep the developers sane. CNN (http://www.cnn.com/), for example, uses the Prototype and Script.aculo.us libraries. Their front page displays a total of 12 external scripts and more than 20 inline script blocks. One simple optimization would be to group some, if not all, of this code into one external JavaScript file, thereby dramatically cutting down the number of HTTP requests necessary to render the page.

大多数现代网站使用多个 JavaScript 文件：通常包括一个小型库，它是一个工具和控件集合以简化跨浏览器富交互网页应用程序开发，还有一些网站相关的代码，被分割成几个逻辑单元使开发者保持清晰。例如 CNN（http://www.cnn.com/），使用 Prototype 和 Script.aculo.us 库。它的首页显示了 12 个外部脚本和超

过 20 个内联脚本块。一个简单的优化是将某些脚本合并成一个外部 JavaScript 文件，而不是全部，从而大大降低渲染页面所需 HTTP 请求的数量。

Apache Ant provides the ability to combine several files via the **concat** task. It is important, however, to remember that JavaScript files usually need to be concatenated in a specific order to respect dependencies. Once these dependencies have been established, using a **filelist** or a combination of **fileset** elements allows the order of the files to be preserved. Here is what the Ant target looks like:

Apache Ant 通过 concat 任务提供合并几个文件的能力。这很重要，但是要记住 JavaScript 文件通常需要按照依赖关系的特定顺序进行连接。一旦创建了依赖关系，使用 filelist 或组合使用 fileset 元素可将这些文件次序保存下来。Ant 目标体的样子如下：

```
 <target name="js.concatenate">
   <concat destfile="${build.dir}/concatenated.js">
     <filelist dir="${src.dir}"

       files="a.js, b.js"/>
     <fileset dir="${src.dir}"

       includes="*.js"

       excludes="a.js, b.js"/>
   </concat>
</target>
```

This target creates the file concatenated.js under the build directory, as a result of the concatenation of a.js, followed by b.js, followed by all the other files under the source directory in alphabetical order.

此目标体在开发目录下创建 concatenated.js 文件，它首先连接 a.js，然后是 b.js，然后是源目录下按字母顺序排列的其它文件。

Note that if any of the source files (except possibly the last one) does not end with either a semicolon or a line terminator, the resulting concatenated file may not contain valid JavaScript code. This can be fixed by instructing Ant to check whether each concatenated source file is terminated by a newline, using the **fixlastline** attribute:

注意所有源文件中（可能除了最后一个）如果不是以分号或行终止符结束的，那么合并文件的结果可能不成为有效的 JavaScript 代码。可这样修正：指示 Ant 检查每个源文件是否以新行结束，使用 fixlastline 属性：

```
 <concat destfile="${build.dir}/concatenated.js" fixlastline="yes">
   ...
</concat>
```

## Preprocessing JavaScript Files 预处理 JavaScript 文件

In computer science, a preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers. The amount and kind of processing done depends on the nature of the preprocessor; some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of fully fledged programming languages.—http://en.wikipedia.org/wiki/Preprocessor

在计算机科学中，预处理器的任务是将输入数据处理成另一种编程语言所使用的数据。它输出的就是我们所说的经过预处理的输入数据，它通常被一些后续程序所使用，例如编译器。预处理的数量和类型与预处理器的性质有关，有些预处理器只能处理简单文本替换和宏扩展，而另一些则成为完全成熟的编程语言。（维基百科）

—http://en.wikipedia.org/wiki/Preprocessor

Preprocessing your JavaScript source files will not make your application faster by itself, but it will allow you to, among other things, conditionally instrument your code in order to measure how your application is performing.

预处理您的 JavaScript 源文件并不会使你的程序更快，但它允许你在代码中加入其它语言才有的一些特性，例如用条件体插入一些测试代码，来衡量你应用程序的性能。

Since no preprocessor is specifically designed to work with JavaScript, it is necessary to use a lexical preprocessor that is flexible enough that its lexical analysis rules can be customized, or else use one that was designed to work with a language for which the lexical grammar is close enough to JavaScript's own lexical grammar. Since the C programming language syntax is close to JavaScript, the C preprocessor (**cpp**) is a good choice. Here is what the Ant target looks like:

由于没有专门为 JavaScript 设计的预处理器，有必要使用一个词法预处理器，它足够灵活，可定制其词法分析规则，或者使用一个为某种语言设计的工具，其词法语法与 JavaScript 自己的词法语法足够接近。由于 C 语言语法接近 JavaScript，C 预处理器（cpp）就是一个很好的选择。Ant 目标体如下：

```
<target name="js.preprocess" depends="js.concatenate">
  <apply executable="cpp" dest="${build.dir}">
    <fileset dir="${build.dir}"

      includes="concatenated.js"/>
    <arg line="-P -C -DDEBUG"/>
    <srcfile/>
    <targetfile/>
    <mapper type="glob"

      from="concatenated.js"

      to="preprocessed.js"/>
  </apply>
</target>
```

This target, which depends on the **js.concatenate** target, creates the file **preprocessed.js** under the build directory as a result of running **cpp** on the previously concatenated file. Note that **cpp** is run using the standard **-P** (inhibit generation of line markers) and **-C** (do not discard comments) options. In this example, the **DEBUG** macro is also defined.

这一目标体依赖于 js.concatenate 目标，它在前面的连接文件中运行 cpp，其结果是在开发目录下创建 preprocessed.js 文件。注意 cpp 使用标准-P（抑制线标记生成）和-c（不删除注释）选项。在这个例子中还定义了 DEBUG 宏。

With this target, you can now use the macro definition (**#define**, **#undef**) and the conditional compilation (**#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**) directives directly inside your JavaScript files, allowing you, for example, to conditionally embed (or remove) profiling code:

有了这个目标体，你现在可以直接在 JavaScript 文件中使用宏定义（#define, #undef）和条件编译（#if, #ifdef, #ifndef, #else, #elif, #endif）指令。例如，你可以使用条件体嵌入（或删除）测试代码：

```
#ifdef DEBUG


(new YAHOO.util.YUILoader({
  require: ['profiler'],
  onSuccess: function(o) {
    YAHOO.tool.Profiler.registerFunction('foo', window);
  }
})).insert();


#endif
```

If you plan to use multiline macros, make sure you use the Unix end-of-line character (**LF**). You may use the **fixcrlf** Ant task to automatically fix that for you.

如果你打算使用多行宏，请确保你使用了 Unix 的行结束符（LF）。你可用 Ant 任务 fixcrlf 自动修复它们。

Another example, not strictly related to performance but demonstrating how powerful JavaScript preprocessing can be, is the use of "variadic macros" (macros accepting a variable number of arguments) and file inclusion to implement JavaScript assertions. Consider the following file named include.js:

另一个例子，不太严格，但说明了 JavaScript 预编译可以多么强大，它使用了"传参数的宏"（宏接收可变数据的参数）和文件包含，以实现 JavaScript 断言。考虑下面这个 include.js 文件：

```
#ifndef _INCLUDE_JS_
#define _INCLUDE_JS_


#ifdef DEBUG
function assert(condition, message) {
  // Handle the assertion by displaying an alert message
  // possibly containing a stack trace for example.
}
#define ASSERT(x, ...) assert(x, ## __VA_ARGS__)
#else
#define ASSERT(x, ...)
#endif


#endif
```

You can now write JavaScript code that looks like the following:

现在你可以像下面这样书写 JavaScript 代码：

```
#include "include.js"


function myFunction(arg) {
  ASSERT(YAHOO.lang.isString(argvar), "arg should be a string");
  ...
#ifdef DEBUG
  YAHOO.log("Log this in debug mode only");
#endif
  ...
}
```

The assertion and the extra logging code appear only when the DEBUG macro is set during development. These statements disappear in the final production build.

断言和额外的日志代码只出现在开发过程设置的 DEBUG 宏块中。这些声明不会出现在最终产品中。

**JavaScript Minification JavaScript 紧凑**

JavaScript minification is the process by which a JavaScript file is stripped of everything that does not contribute to its execution. This includes comments and unnecessary whitespace. The process typically reduces the file size by half, resulting in faster downloads, and encourages programmers to write better, more extensive inline documentation.

JavaScript 紧凑指的是剔除一个 JavaScript 文件中一切运行无关内容的过程。包括注释和不必要的空格。该处理通常可将文件尺寸缩减到一半，其结果是下载速度更快，并鼓励程序员写出更好，更详细的内联文档。

JSMin (http://www.crockford.com/javascript/jsmin.html), developed by Douglas Crockford, remained the standard in JavaScript minification for a long time. However, as web applications kept growing in size and complexity, many felt it was time to push JavaScript minification a step further. This is the main reason behind the development of the YUI Compressor (http://developer.yahoo.com/yui/compressor/), a tool that performs all kinds of smart operations in order to offer a higher level of compaction than other tools in a completely safe way. In addition to stripping comments and unnecessary whitespace, the YUI Compressor offers the following features:

JSMin（http://www.crockford.com/javascript/jsmin.html），由 Douglas Crockford 开发，它保持了 JavaScript 紧凑标准很长一段时间。然而随着网络应用程序在规模和复杂性上不断增长，许多人认为 JavaScript 紧凑

应当再向前推进一步。这是开发 YUI 压缩器的主要原因（http://developer.yahoo.com/yui/compressor/）它提供了所有类型的智能操作，为了提供比其它工具更高级的紧凑操作并且以完全安全的方法实现。除了剔除注释和不必要的空格，YUI 压缩器还提供以下功能：

• Replacement of local variable names with shorter (one-, two-, or three-character) variable names, picked to optimize gzip compression downstream

将局部变量名替换以更短的形式（1 个，2 个，或 3 个字符），以优化后续的 gzip 压缩工作

• Replacement of bracket notation with dot notation whenever possible

(e.g., **foo["bar"]** becomes **foo.bar**)

尽可能将中括号操作符替换为点操作符（例如 foo:["bar"]变成 foo.bar）

• Replacement of quoted literal property names whenever possible

(e.g., **{"foo":"bar"}** becomes **{foo:"bar"}**)

尽可能替换引号直接量属性名（例如{"foo":"bar"}变成{foo:"bar"}）

• Replacement of escaped quotes in strings (e.g., **'aaa\'bbb'** becomes **"aaa'bbb"**)

替换字符串中的转义引号（例如'aaa\'bbb'变成"aaa'bbb"）

• Constant folding (e.g., **"foo"+"bar"** becomes **"foobar"**)

常量折叠（例如"foo"+"bar"变成"foobar"）

Running your JavaScript code through the YUI Compressor results in tremendous savings compared to JSMin without any further action. Consider the following numbers on the core files of the YUI library (version 2.7.0, available at http://developer.yahoo.com/yui/):

与 JSMin 相比，通过 YUI 压缩器极大节省了您的 JavaScript 代码而无需更多操作。下面的数字是 YUI 库的核心代码（2.7.0 版，下载地址：http://developer.yahoo.com/yui/）：

| | |
|---|---|
| Raw yahoo.js, dom.js and event.js | 192,164 bytes |
| yahoo.js, dom.js and event.js + JSMin | 47,316 bytes |
| **yahoo.js, dom.js and event.js + YUI Compressor** | **35,896 bytes** |

In this example, the YUI Compressor offers 24% savings out of the box compared to JSMin. However, there are things you can do to increase the byte savings even further. Storing local references to objects/values, wrapping code in a closure, using constants for repeated values, and avoiding **eval** (and its relatives, the **Function** constructor, **setTimeout**, and **setInterval** when used with a string literal as the first argument), the **with** keyword, and JScript conditional comments all contribute to making the minified file smaller. Consider the following function, designed to toggle the **selected** class on the specified DOM element (220 bytes):

在这个例子中，YUI 压缩器与 JSMin 相比节省了 24%空间。然而，你还可以进一步节省空间。将局部引用存储在对象/值中，用闭包封装代码，使用常量替代重复值，避免 eval（以及相似的 Function 构造器，setTimeout 和 setInterval 使用字符串直接量作为第一个参数），with 关键字，JScript 条件注释，都有助于进一步精缩文件。考虑下面的函数，用来转换特定 DOM 元素的 selected 类（220 字节）：

```
function toggle (element) {
  if (YAHOO.util.Dom.hasClass(element, "selected")){
    YAHOO.util.Dom.removeClass(element, "selected");
  } else {
    YAHOO.util.Dom.addClass(element, "selected");
  }
}
```

The YUI Compressor will transform this code into the following (147 bytes):

YUI 压缩器将此代码转换如下（147 字节）：

```
function
toggle(a){if(YAHOO.util.Dom.hasClass(a,"selected")){YAHOO.util.Dom.removeClass(a,"selected")}else{YAHOO.util.Dom.addClass(a,"selected")}};
```

If you refactor the original version by storing a local reference to **YAHOO.util.Dom** and using a constant for the "**selected**" value, the code becomes (232 bytes):

如果你重构原始代码，将 YAHOO.util.Dom 存入一个局部引用，并使用常量存放"select"值，代码将变成如下的样子（232 字节）：

```
function toggle (element) {
  var YUD = YAHOO.util.Dom, className = "selected";
  if (YUD.hasClass(element, className)){
    YUD.removeClass(element, className);
  } else {
    YUD.addClass(element, className);
  }
}
```

This version shows even greater savings after minification using the YUI Compressor (115 bytes):

此版本在经过 YUI 压缩器紧凑处理之后将变得更小（115 字节）：

```
function toggle(a){var
c=YAHOO.util.Dom,b="selected";if(c.hasClass(a,b)){c.removeClass(a,b)}else{c.addClass(a,b)}};
```

The compaction ratio went from 33% to 48%, which is a staggering result given the small amount of work needed. However, it is important to note that gzip compression, happening downstream, may yield conflicting results; in other words, the smallest minified file may not always give the smallest gzipped file. That strange result is a direct consequence of lowering the amount of redundancy in the original file. In addition, this kind of microoptimization incurs a small runtime cost because variables are now used in place of literal values, thus requiring additional lookups. Therefore, I usually recommend not abusing these techniques, although it may still be worth considering them when serving content to user agents that don't support (or advertise their support for) gzip compression.

压缩率从 33%变为 48%，只需要少量工作就得到惊人的结果。然而，要注意后续的 gzip 压缩，可能会产生相反的结果。换句话说，最小的紧凑文件并不总是给出最小的 gzip 压缩文件。这种奇怪的结果是降低原文件的冗余量造成的。此外，这类微观优化还导致了一个很小的运行期负担，因为变量替代了直接量，所以需要额外的查找。因此，通常建议不要滥用这些技术，虽然从服务内容到用户代理不支持（或声称支持）gzip 压缩时，它们还是值得考虑的。

In November 2009, Google released an even more advanced minification tool called the Closure Compiler (http://code.google.com/closure/compiler/). This new tool goes further than the YUI Compressor when using its advanced optimizations option. In this mode, the Closure Compiler is extremely aggressive in the ways that it transforms code and renames symbols. Although it yields incredible savings, it requires the developer to be very careful and to ensure that the output code works the same way as the input code. It also makes debugging more difficult because almost all of the symbols are renamed. The Closure library does come with a Firebug extension, named the

Closure Inspector (http://code.google.com/closure/compiler/docs/inspector.html), that provides a mapping between the obfuscated symbols and the original symbols. Nevertheless, this extension is not available on browsers other than Firefox, which may be a problem when debugging browser-specific code paths, and debugging still remains harder than with other, less aggressive minification tools.

2009 年 11 月，Google 发布了一个更先进的紧凑工具闭包编译器 http://code.google.com/closure/compiler/ 这种工具比 YUI 压缩器更进一步，当使用其先进优化选项时。在这种模式下，闭包编译器以极端霸道的方式转换代码并修改符号名。虽然它产生了难以置信的压缩率，但它要求开发者必须非常小心以确保输出代码与输入代码等价。它还使得调试更为困难，因为几乎所有符号都被改名了。此闭包库以一个 Firebug 扩展的形式发布，命名为闭包察看器（Closure Inspector）

(http://code.google.com/closure/compiler/docs/inspector.html)，并提供了一个转换后符号名和原始符号名之间的对照表。不过，这个扩展不能用于 Firefox 之外的浏览器，所以对那些浏览器相关的代码来说是个问题，而且和那些不这么霸道的紧凑工具相比，调试工作更困难。

## Buildtime Versus Runtime Build Processes 开发过程中的编译时和运行时

Concatenation, preprocessing, and minification are steps that can take place either at buildtime or at runtime. Runtime build processes are very useful during development, but generally are not recommended in a production environment for scalability reasons. As a general rule for building high-performance applications, everything that can be done at buildtime should not be done at runtime.

连接，预处理，和紧凑既可以在编译时发生，也可以在运行时发生。在开发过程中，运行时创建过程非常有用，但由于扩展性原因一般不建议在产品环境中使用。开发高性能应用程序的一个普遍规则是，只要能够在编译时完成的工作，就不要在运行时去做。

Whereas Apache Ant is definitely an offline build program, the agile build tool presented toward the end of this chapter represents a middle ground whereby the same tool can be used during development and to create the final assets that will be used in a production environment.

Apache Ant 无疑是一种脱机开发程序，而本章末尾出现的灵巧开发工具代表了中间路线，同样的工具即可用于开发期创建最终断言，也可用于产品环境。

**JavaScript Compression   JavaScript 压缩**

When a web browser requests a resource, it usually sends an **Accept-Encoding** HTTP header (starting with HTTP/1.1) to let the web server know what kinds of encoding transformations it supports. This information is primarily used to allow a document to be compressed, enabling faster downloads and therefore a better user experience. Possible values for the **Accept-Encoding** value tokens include: **gzip**, **compress**, **deflate**, and **identity** (these values are registered by the Internet Assigned Numbers Authority, or IANA).

当网页浏览器请求一个资源时，它通常发送一个 Accept-Encoding 的 HTTP 头（以 HTTP/1.1 开始）让网页服务器知道传输所支持的编码类型。此信息主要用于允许文档压缩以获得更快下载速度，从而改善用户体验。Accept-Encoding 的取值范围是：gzip，compress，deflate，和 identity（这些值已经在以太网地址分配机构（即 IANA）注册）。

If the web server sees this header in the request, it will choose the most appropriate encoding method and notify the web browser of its decision via the **Content-Encoding** HTTP header.

如果网页服务器在请求报文中看到这些信息头，它将选择适当的编码方法，并通过 Content-Encoding 的 HTTP 头通知浏览器。

**gzip** is by far the most popular encoding. It generally reduces the size of the payload by 70%, making it a weapon of choice for improving the performance of a web application. Note that gzip compression should be used primarily on text responses, including JavaScript files. Other file types, such as images or PDF files, should not be gzipped, because they are already compressed and trying to compress them again is a waste of server resources.

gzip 大概是目前最流行的编码格式。它通常可将有效载荷减少到 70%，成为提高网页应用性能的有力武器。注意 gzip 压缩器主要用于文本报文，包括 JavaScript 文件。其他文件类型，如图片和 PDF 文件，不应该使用 gzip 压缩，因为它们已经压缩，如果试图再次压缩只会浪费服务器资源。

If you use the Apache web server (by far the most popular), enabling gzip compression requires installing and configuring either the **mod_gzip** module (for Apache 1.3 and available at http://www.schroepl.net/projekte/mod_gzip/) or the **mod_deflate** module (for Apache 2).

如果您使用 Apache 网页服务器（目前最流行的），启用 gzip 压缩功能需要安装并配置 mod_gzip 模块（针对 Apache 1.3，位于 http://www.schroepl.net/projekte/mod_gzip/）或者 mod_deflate 模块（针对 Apache 2）。

Recent studies done independently by Yahoo! Search and Google have shown that roughly 15% of the content delivered by large websites in the United States is served uncompressed. This is mostly due to a missing **Accept-Encoding** HTTP header in the request, stripped by some corporate proxies, firewalls, or even PC security software. Although gzip compression is an amazing tool for web developers, one must be mindful of this fact and strive to write code as concisely as possible. Another technique is to serve alternate JavaScript content to users who are not going to benefit from gzip compression but could benefit from a lighter experience (although users should be given the choice to switch back to the full version).

由 Yahoo!搜索和 Google 独立完成的最新研究表明，美国大型网站提供的内容中有大约 15%未经过压缩。大多数因为在请求报文中缺少 Accept-Encoding 的 HTTP 头，它被一些公司代理、防火墙、甚至 PC 安全软件剔除了。虽然 gzip 压缩是一个惊人的网页开发工具，但还是要注意到这个事实，尽量书写简洁的代码。另一种技术是提供替代的 JavaScript 内容，使那些不能受益于 gzip 压缩的用户，可以使用更简单的用户体验（用户可以选择切换回完整版本）。

To that effect, it is worth mentioning Packer (http://dean.edwards.name/packer/), a JavaScript minifier developed by Dean Edwards. Packer is able to shrink JavaScript files beyond what the YUI Compressor can do. Consider the following results on the jQuery library (version 1.3.2, available at http://www.jquery.com/):

为此，值得提到 Packer (http://dean.edwards.name/packer/)，由 Dean Edwards 开发的一个 JavaScript 紧凑工具。Packer 对 JavaScript 压缩能够超过 YUI 压缩器的水平。考虑下面对 jQuery 库的压缩结果（版本 1.3.2，下载地址 http://www.jquery.com/）：

| | |
|---|---|
| jQuery | 120,180 bytes |
| jQuery + YUI Compressor | 56,814 bytes |
| **jQuery + Packer** | **39,351 bytes** |
| Raw jQuery + gzip | 34,987 bytes |
| **jQuery + YUI Compressor + gzip** | **19,457 bytes** |
| **jQuery + Packer + gzip** | **19,228 bytes** |

After gzipping, running the jQuery library through Packer or the YUI Compressor yields very similar results. However, files compressed using Packer incur a fixed runtime cost (about 200 to 300 milliseconds on my modern laptop). Therefore, using the YUI Compressor in combination with gzipping always gives the best results. However, Packer can be used with some success for users on slow lines that don't support gzip compression, for whom the cost of unpacking is negligible compared to the cost of downloading large amounts of code. The only downside to serving different JavaScript content to different users is increased QA costs.

经过 gzip 压缩之后，jQuery 库经过 Packer 或 YUI 压缩器产生的结果非常相近。然而，使用 Packer 压缩文件导致一个固定的运行时代价（在我的不落后的笔记本电脑上大约是 200 至 300 毫秒）。因此，使用 YUI 压缩器和 gzip 结合总能给出最佳结果。然而，Packer 可用于网速不高或者不支持 gzip 压缩的情况，解压缩的代价与下载大量代码的代价相比微不足道。为不同用户提供不同 JavaScript 的唯一缺点是质量保证成本的增加。

## Caching JavaScript Files 缓存 JavaScript 文件

Making HTTP components cacheable will greatly improve the experience of repeat visitors to your website. As a concrete example, loading the Yahoo! home page (http://www.yahoo.com/) with a full cache requires 90%

fewer HTTP requests and 83% fewer bytes to download than with an empty cache. The round-trip time (the elapsed time between the moment a page is requested and the firing of the **onload** event) goes from 2.4 seconds to 0.9 seconds (http://yuiblog.com/blog/2007/01/04/performance-research-part-2/). Although caching is most often used on images, it should be used on all static components, including JavaScript files.

使 HTTP 组件可缓存将大大提高用户再次访问网站时的用户体验。一个具体的例子是，加载 Yahoo!主页时（http://www.yahoo.com/），和不使用缓存相比，使用缓存将减少 90%的 HTTP 请求和 83%的下载字节。往返时间（从请求页面开始到第一次 onload 事件）从 2.4 秒下降到 0.9 秒 (http://yuiblog.com/blog/2007/01/04/performance-research-part-2/)。虽然图片经常使用缓存，但它应当被使用在所有静态内容上，包括 JavaScript 文件。

Web servers use the **Expires** HTTP response header to let clients know how long a resource can be cached. The format is an absolute timestamp in RFC 1123 format. An example of its use is: **Expires: Thu, 01 Dec 1994 16:00:00 GMT**. To mark a response as "never expires," a web server sends an **Expires** date approximately one year in the future from the time at which the response is sent. Web servers should never send **Expires** dates more than one year in the future according to the HTTP 1.1 RFC (RFC 2616, section 14.21).

网页服务器使用 Expires 响应报文 HTTP 头让客户端知道缓存资源的时间。它是一个 RFC 1123 格式的绝对时间戳。例如：**Expires: Thu, 01 Dec 1994 16:00:00 GMT**。要将响应报文标记为"永不过期"，网页服务器可以发送一个时间为请求时间之后一年的 Expires 数据。根据 HTTP 1.1 RFC（RFC 2616，14.21 节）的要求，网页服务器发送的 Expires 时间不应超过一年。

If you use the Apache web server, the **ExpiresDefault** directive allows you to set an expiration date relative to the current date. The following example applies this directive to images, JavaScript files, and CSS stylesheets:

如果你使用 Apache 网页服务器，ExpiresDefault 指令允许你根据当前时间设置过期时间。下面的例子将此指令用于图片，JavaScript 文件，和 CSS 样式表：

```
 <FilesMatch "\.(jpg|jpeg|png|gif|js|css|htm|html)$">
   ExpiresActive on
   ExpiresDefault "access plus 1 year"
</FilesMatch>
```

Some web browsers, especially when running on mobile devices, may have limited caching capabilities. For example, the Safari web browser on the iPhone does not cache a component if its size is greater than 25KB uncompressed (see http://yuiblog.com/blog/2008/02/06/iphone-cacheability/) or 15KB for the iPhone 3.0 OS. In those cases, it is relevant to consider a trade-off between the number of HTTP components and their cacheability by splitting them into smaller chunks.

某些网页浏览器，特别是那些移动设备上的浏览器，可能有缓存限制。例如，iPhone 的 Safari 浏览器不能缓存解压后大于 25K 的组件（见 http://yuiblog.com/blog/2008/02/06/iphone-cacheability/），在 iPhone 3.0 操作系统上不能大于 15K。在这种情况下，应衡量 HTTP 组件数量和它们的可缓存性，考虑将它们分解成更小的块。

You can also consider using client-side storage mechanisms if they are available, in which case the JavaScript code must itself handle the expiration.

如果可能的话，你还可以考虑客户端存储机制，让 JavaScript 代码自己来处理过期。

Finally, another technique is the use of the HTML 5 offline application cache, implemented in Firefox 3.5, Safari 4.0, and on the iPhone beginning with iPhone OS 2.1. This technology relies on a manifest file listing the resources to be cached. The manifest file is declared by adding a **manifest** attribute to the **<html>** tag (note the use of the HTML 5 DOCTYPE):

最后，另一种技术是使用 HTML 5 离线应用程序缓存，它已经在如下浏览器中实现：Firefox 3.5，Safari 4.0，从 iPhone OS 2.1 开始以后的版本。此技术依赖于一个配置文件，列出应当被缓存的资源。此配置文件通过<html>标签的 manifest 属性（注意要使用 HTML 5 的 DOCTYPE）：

```
<!DOCTYPE html>
<html manifest="demo.manifest">
```

The manifest file uses a special syntax to list offline resources and must be served using the **text/cache-manifest** mime type. More information on offline web application caching can be found on the W3C website at http://www.w3.org/TR/html5/offline.html.

此配置文件使用特殊语法列出离线资源，必须使用 text/cache-manifest 指出它的媒体类型。更多关于离线网页应用缓存的信息参见 W3C 的网站 http://www.w3.org/TR/html5/offline.html。

**Working Around Caching Issues　关于缓存问题**

Adequate cache control can really enhance the user experience, but it has a downside: when revving up your application, you want to make sure your users get the latest version of the static content. This is accomplished by renaming static resources whenever they change.

充分利用缓存控制可真正提高用户体验，但它有一个缺点：当应用程序更新之后，你希望确保用户得到静态内容的最新版本。这通过对改动的静态资源进行重命名实现。

Most often, developers add a version or a build number to filenames. Others like to append a checksum. Personally, I like to use a timestamp. This task can be automated using Ant. The following target takes care of renaming JavaScript files by appending a timestamp in the form of **yyyyMMddhhmm**:

大多情况下，开发者向文件名中添加一个版本号或开发编号。有人喜欢追加一个校验和。个人而言，我更喜欢时间戳。此任务可用 Ant 自动完成。下面的目标体通过附加一个 yyyyMMddhhmm 格式的时间戳重名名 JavaScript 文件：

```
<target name="js.copy">
  <!-- Create the time stamp -->
  <tstamp/>
  <!-- Rename JavaScript files by appending a time stamp -->
  <copy todir="${build.dir}">
    <fileset dir="${src.dir}" includes="*.js"/>
    <globmapper from="*.js" to="*-${DSTAMP}${TSTAMP}.js"/>
  </copy>
</target>
```

**Using a Content Delivery Network　使用内容传递网**

A content delivery network (CDN) is a network of computers distributed geographically across the Internet that is responsible for delivering content to end users. The primary reasons for using a CDN are reliability, scalability, and above all, performance. In fact, by serving content from the location closest to the user, CDNs are able to dramatically decrease network latency.

内容传递网络（CDN）是按照地理分布的计算机网络，通过以太网负责向最终用户分发内容。使用 CDN 的主要原因是可靠性，可扩展性，但更主要的是性能。事实上，通过地理位置上最近的位置向用户提供服务，CDN 可以极大地减少网络延迟。

Some large companies maintain their own CDN, but it is generally cost effective to use a third-party CDN provider such as Akamai Technologies (http://www.akamai.com/) or Limelight Networks (http://www.limelightnetworks.com/).

一些大公司维护它们自己的 CDN，但通常使用第三方 CDN 更经济一些，如 Akamai 科技（http://www.akamai.com）或 Limelight 网络（http://www.limelightnetworkds.com）。

Switching to a CDN is usually a fairly simple code change and has the potential to dramatically improve end-user response times.

切换到 CDN 通常只需改变少量代码，并可能极大地提高最终用户响应速度。

It is worth noting that the most popular JavaScript libraries are all accessible via a CDN. For example, the YUI library is served directly from the Yahoo! network (server name is yui.yahooapis.com, details available at http://developer.yahoo.com/yui/articles/hosting/), and jQuery, Dojo, Prototype, Script.aculo.us, MooTools, YUI, and other libraries are all available directly via Google's CDN (server name is ajax.googleapis.com, details available at http://code.google.com/apis/ajaxlibs/).

值得注意的是，最流行的 JavaScript 库都可以通过 CDN 访问。例如，YUI 直接从 Yahoo!网络获得（服务器名是 **yui.yahooapis.com**，http://developer.yahoo.com/yui/articles/hosting/）。jQuery，Dojo，Prototype，Script.aculo.us，MooTools，YUI，还有其他库都可以直接通过 Google 的 CDN 获得（服务器名是 **ajax.googleapis.com**，http://code.google.com/apis/ajaxlibs/）。

## Deploying JavaScript Resources 部署 JavaScript 资源

The deployment of JavaScript resources usually amounts to copying files to one or several remote hosts, and also sometimes to running a set of shell commands on those hosts, especially when using a CDN, to distribute the newly added files across the delivery network.

部署 JavaScript 资源通常需要拷贝文件到一个或多个远程主机，有时在那些主机上运行一个 shell 命令集，特别当使用 CDN 时，通过传递网络分发最新添加的文件。

Apache Ant gives you several options to copy files to remote servers. You could use the **copy** task to copy files to a locally mounted filesystem, or you could use the optional **FTP** or **SCP** tasks. My personal preference is to go directly to using the **scp** utility, which is available on all major platforms. Here is a very simple example demonstrating this:

Apache Ant 提供给你几个选项用于将文件复制到远程服务器上。你可以使用 copy 任务将文件复制到一个本地挂载的文件系统，或者使用 FTP 或 SCP 任务。我个人喜欢直接使用 scp 工具，因为所有主流平台都支持它。这是一个非常简单的例子：

```
 <apply executable="scp" fail parallel="true">
   <fileset dir="${build.dir}" includes="*.js"/>
  <srcfile/>
   <arg line="${live.server}:/var/www/html/"/>
</apply>
```

Finally, in order to execute shell commands on a remote host running the SSH daemon, you can use the optional **SSHEXEC** task or simply invoke the **ssh** utility directly, as demonstrated in the following example, to restart the Apache web server on a Unix host:

最终，为了在远程主机上运行 shell 命令启动 SSH 服务，你可以使用 SSHEXEC 任务选项或简单地直接调用 ssh 工具，正如下面的例子中那样，在 Unix 主机上重启 Apache 网页服务：

```
 <exec executable="ssh" fail>
   <arg line="${live.server}"/>
```

```
  <arg line="sudo service httpd restart"/>
</exec>
```

**Agile JavaScript Build Process 灵巧的 JavaScript 开发过程**

Traditional build tools are great, but most web developers find them very cumbersome because it is necessary to manually compile the solution after every single code change. Instead, it's preferable to just have to refresh the browser window and skip the compilation step altogether. As a consequence, few web developers use the techniques outlined in this chapter, resulting in applications or websites that perform poorly. Thankfully, it is fairly simple to write a tool that combines all these advanced techniques, allowing web developers to work efficiently while still getting the most performance out of their application.

传统开发工具很强大，但大多数网页开发人员嫌它们太麻烦，因为在每次修改之后都需要手工编译解决方案。开发人员更喜欢另一种方法，跳过整个编译步骤，直接刷新浏览器窗口。因此，几乎没有网页开发者使用本章之外的技术而导致应用程序或网站表现不佳。幸运的是，写一个综合上述优点的工具十分简单，它允许网页开发者在应用程序之外获得最佳性能。

**smasher** is a PHP5 application based on an internal tool used by Yahoo! Search. It combines multiple JavaScript files, preprocesses them, and optionally minifies their content. It can be run from the command line, or during development to handle web requests and automatically combine resources on the fly. The source code can be found at http://github.com/jlecomte/smasher, and contains the following files:

smasher 是一个 PHP5 应用程序，基于 Yahoo!搜索所使用的一个内部工具。它合并多个 JavaScript 文件，预处理它们，根据选项对它们内容进行紧凑处理。它可以从命令行运行，或者在开发过程中处理网页请求自动合并资源。源码可在 http://github.com/jlecomte/smasher 下载，包含以下文件：

**smasher.php**

Core file 核心文件

**smasher.xml**

Configuration file 配置文件

**smasher**

Command-line wrapper  命令行封装

**smasher_web.php**

Web server entry point  网页服务入口

**smasher** requires an XML configuration file containing the definition of the groups of files it will combine, as well as some miscellaneous information about the system. Here is an example of what this file looks like:

smasher 需要一个 XML 配置文件包含要合并的文件组的定义，以及一些有关系统的信息。下面是这个文件的一个例子：

```
<?xml version="1.0" encoding="utf-8"?>
<smasher>
  <temp_dir>/tmp/</temp_dir>
  <root_dir>/home/jlecomte/smasher/files/</root_dir>
  <java_bin>/usr/bin/java</java_bin>
  <yuicompressor>/home/jlecomte/smasher/yuicompressor-2-4-2.jar</yuicompressor>


  <group id="yui-core">
    <file type="css" src="reset.css" />
     <file type="css" src="fonts.css" />
    <file type="js" src="yahoo.js" />
    <file type="js" src="dom.js" />
    <file type="js" src="event.js" />
  </group>


  <group id="another-group">
    <file type="js" src="foo.js" />
    <file type="js" src="bar.js" />
    <macro name="DEBUG" value="1" />
  </group>
  ...
</smasher>
```

Each **group** element contains a set of JavaScript and/or CSS files. The **root_dir** top-level element contains the path to the directory where these files can be found. Optionally, **group** elements can also contain a list of preprocessing macro definitions.

每一个 group 元素包含一个 JavaScript 或 CSS 文件集合。该 root_dir 顶层元素包含查找这些文件的路径。group 元素还可通过选项来包含一个预处理宏定义列表。

Once this configuration file has been saved, you can run **smasher** from the command line. If you run it without any of the required parameters, it will display some usage information before exiting. The following example shows how to combine, preprocess, and minify the core YUI JavaScript files:

一旦这个配置文件保存下来，你就可以在命令行运行 smasher。如果你不加任何参数运行它，它将在退出之前显示一些有用的信息。下面的例子演示了如何合并，预处理，和紧凑处理 YUI 核心 JavaScript 文件：

```
$ ./smasher -c smasher.xml -g yui-core -t js
```

If all goes well, the output file can be found in the working directory, and is named after the group name (**yui-core** in this example) followed by a timestamp and the appropriate file extension (e.g., yui-core-200907191539.js).

如果一切正常，可以在工作目录找到输出文件，它的名字以组名开头（这个例子中为 yui-core）后面跟着一个时间戳和适当的文件扩展名（例如，yui-core-200907191539.js）。

Similarly, you can use **smasher** to handle web requests during development by placing the file smasher_web.php somewhere under your web server document root and by using a URL similar to this one:

同样，你可以使用 smasher 在开发过程中处理网页请求，将文件 smasher_web.php 放在你的网页服务根文档中，使用类似这样的 URL：

http://<host>/smasher_web.php?conf=smasher.xml&group=yui-core&type=css&nominify

By using different URLs for your JavaScript and CSS assets during development and in production, it is now possible to work efficiently while still getting the most performance out of the build process.

在开发和产品中，为你的 JavaScript 和 CSS 资源使用不同的 URL，现在它可以在开发过程之外获得最佳性能。

## Summary 总结

The build and deployment process can have a tremendous impact on the performance of a JavaScript-based application. The most important steps in this process are:

开发和部署过程对基于 JavaScript 的应用程序可以产生巨大影响，最重要的几个步骤如下：

• Combining JavaScript files to reduce the number of HTTP requests

合并 JavaScript 文件，减少 HTTP 请求的数量

• Minifying JavaScript files using the YUI Compressor

使用 YUI 压缩器紧凑处理 JavaScript 文件

• Serving JavaScript files compressed (gzip encoding)

以压缩形式提供 JavaScript 文件（gzip 编码）

• Making JavaScript files cacheable by setting the appropriate HTTP response headers and work around caching issues by appending a timestamp to filenames

通过设置 HTTP 响应报文头使 JavaScript 文件可缓存，通过向文件名附加时间戳解决缓存问题

• Using a Content Delivery Network to serve JavaScript files; not only will a CDN improve performance, it should also manage compression and caching for you

使用内容传递网络（CDN）提供 JavaScript 文件，CDN 不仅可以提高性能，它还可以为你管理压缩和缓存

All these steps should be automated using publicly available build tools such as Apache Ant or using a custom build tool tailored to your specific needs. If you make the build process work for you, you will be able to greatly improve the performance of web applications or websites that require large amounts of JavaScript code.

所有这些步骤应当自动完成，不论是使用公开的开发工具诸如 Apache Ant，还是使用自定义的开发工具以实现特定需求。如果你使这些开发工具为你服务，你可以极大改善那些大量使用 JavaScript 代码的网页应用或网站的性能。

# 第十章　Tools　工具

Having the right software is essential for identifying bottlenecks in both the loading and running of scripts. A number of browser vendors and large-scale websites have shared techniques and tools to help make the Web faster and more efficient. This chapter focuses on some of the free tools available for:

当确定脚本加载和运行时的瓶颈所在时，合手的工具是必不可少的。许多浏览器厂商和大型网站分享了一些技术和工具，帮助开发者使网页更快，效率更高。本章关注于这些免费工具：

**Profiling** 性能分析

Timing various functions and operations during script execution to identify areas for optimization

在脚本运行期定时执行不同函数和操作，找出需要优化的部分

**Network analysis** 网络分析

Examining the loading of images, stylesheets, and scripts and their effect on overall page load and rendering

检查图片，样式表，和脚本的加载过程，汇报它们对整个页面加载和渲染的影响

When a particular script or application is performing less than optimally, a profiler can help prioritize areas for optimization. This can get tricky because of the range of supported browsers, but many vendors now provide a profiler along with their debugging tools. In some cases, performance issues may be specific to a particular browser; other times, the symptoms may occur across multiple browsers. Keep in mind that the optimizations applied to one browser might benefit other browsers, but they might have the opposite effect as well. Rather than assuming which functions or operations are slow, profilers ensure that optimization time is spent on the slowest areas of the system that affect the most browsers.

当一个特定的脚本或应用程序没有达到最优状态时，一个性能分析器有助于安排优化工作的先后次序。不过，因为浏览器支持的范围不同，这可能变得很麻烦，但许多厂商在他们的调试工具中提供了性能分析器。有些情况下，性能问题可能与特定浏览器有关，其他情况下，这些症状可能出现在多个浏览器。请记住，在一个浏览器上所进行的优化可能适用于其他浏览器，也可能产生相反的效果。性能分析工具确保优化工作花费在系统中最慢，影响大多数浏览器的地方，而不是去判定那些函数或操作缓慢。

While the bulk of this chapter focuses on profiling tools, network analyzers can be highly effective in helping to ensure that scripts and pages are loading and running as quickly as possible. Before diving into tweaking code, you should be sure that all scripts and other assets are being loaded optimally. Image and stylesheet loading can affect the loading of scripts, depending on how many concurrent requests the browser allows and how many assets are being loaded.

虽然本章大多数内容关注于性能分析工具，其实网络分析工具可以极大提高分析效率，以确保脚本和页面尽可能快地加载运行。在调整代码之前，您应该确保脚本和其他资源的加载过程已经优化过了。图片和样式表加载会影响脚本加载，这取决于浏览器允许多少并发请求，有多少资源需要加载。

Some of these tools provide tips on how to improve the performance of web pages. Keep in mind that the best way to interpret the information these tools provide is to learn more about the rationale behind the rules. As with most rules, there are exceptions, and a deeper understanding of the rules allows you to know when to break them.

这里的一些工具提供了如何优化网页性能的秘诀。请记住，要充分利用这些工具所提供的信息，首先要深入了解这些规则背后的理由。正如大多数规则一样，总会有例外发生，深入理解这些规则使得您知道什么情况下应当突破规则。

## JavaScript Profiling JavaScript 性能分析

The tool that comes with all JavaScript implementations is the language itself. Using the **Date** object, a measurement can be taken at any given point in a script. Before other tools existed, this was a common way to time script execution, and it is still occasionally useful. By default the **Date** object returns the current time, and subtracting one **Date** instance from another gives the elapsed time in milliseconds. Consider the following example, which compares creating elements from scratch with cloning from an existing element (see Chapter 3, DOM Scripting):

此工具与所有 JavaScript 实例与生俱来，正是语言自身。使用 Data 对象可以测量脚本的任何部分。在其它工具出现之前，测试脚本运行时间是一种常用手段，现在仍然会不时用到。通常使用 Data 返回当前时间，然后减去另一个 Data 值以得到以毫秒为单位的时间差。考虑下面的例子，它比较创建新元素和克隆已有元素所用的时间（参见第三章，DOM 编程）：

```
 var start = new Date(),
     count = 10000,
     i, element, time;


for (i = 0; i < count; i++) {
   element = document.createElement ('div');
}


time = new Date() - start;
alert('created ' + count + ' in ' + time + 'ms');


start = new Date();
for (i = 0, i < count; i++) {
   element = element.cloneNode(false);
}
```

```
time = new Date() - start;
alert('created ' + count + ' in ' + time + 'ms');
```

This type of profiling is cumbersome to implement, as it requires manually instrumenting your own timing code. A **Timer** object that handles the time calculations and stores the data would be a good next step.

此类性能分析十分繁琐，它需要手动添加定时器代码。可定义一个 Timer 对象处理时间计算并存放那些下一步会用到的时间值。

```
 Var Timer = {
   _data: {},


   start: function(key) {
     Timer._data[key] = new Date();
   },


   stop: function(key) {
     var time = Timer._data[key];
     if (time) {
       Timer._data[key] = new Date() - time;
     }
   },


   getTime: function(key) {
     return Timer._data[key];
   }
};
```

```
Timer.start('createElement');
for (i = 0; i < count; i++) {
   element = document.createElement ('div');
}


Timer.stop('createElement');
alert('created ' + count + ' in ' + Timer.getTime('createElement');
```

As you can see, this still requires manual instrumentation, but provides a pattern for building a pure JavaScript profiler. By extending the **Timer** object concept, a profiler can be constructed that registers functions and instruments them with timing code.

正如你看到的，这样做仍需要手工添加代码，但提供了一个建立纯 JavaScript 性能分析的模式。通过扩展 Timer 对象的概念，一个性能分析工具可以在构造时注册函数并在计时代码中调用它们。

**YUI Profiler YUI 分析器**

The YUI Profiler (http://developer.yahoo.com/yui/profiler/), contributed by Nicholas Zakas, is a JavaScript profiler written in JavaScript. In addition to timer functionality, it provides interfaces for profiling functions, objects, and constructors, as well as detailed reports of the profile data. It enables profiling across various browsers and data exporting for more robust reporting and analysis.

YUI 分析器（http://developer.yahoo.com/yui/profiler/），由 Nicholas Zakas 提供，是用 JavaScript 编写的 JavaScript 分析器。除了计时功能，它还提供了用于函数、对象、和构造器的性能分析接口，还包括性能分析数据的详细报告。它可以跨浏览器工作，其输出数据可提供更强大的报告和分析。

The YUI Profiler provides a generic timer that collects performance data. **Profiler** provides static methods for starting and stopping named timings and retrieving profile data.

YUI 分析器提供一个通用定时器用于收集性能数据。Profiler 提供一些静态函数，用于启动和停止命名定时器，以及获取性能数据。

```
 var count = 10000, i, element;
Y.Profiler.start('createElement');


for (i = 0; i < count; i++) {
   element = document.createElement ('div');
}


Y.Profiler.stop('createElement');
alert('created ' + count + ' in ' + Y.Profiler.getAverage('createElement') + 'ms');
```

This clearly improves upon the inline **Date** and **Timer** approach and provides additional profile data regarding the number of times called, as well as the average, minimum, and maximum times. This data is collected and can be analyzed alongside other profile results.

很明显，它改进了内联 Data 和 Timer 方法，提供额外的性能数据包括调用次数，平均时间，最小时间，最大时间等。这些数据收集起来可以与其他测试结果综合分析。

Functions can be registered for profiling as well. The registered function is instrumented with code that collects performance data. For example, to profile the global **initUI** method from Chapter 2, all that is required is the name:

函数分析只需要注册一下。注册的函数被收集性能数据的代码调用。例如，要分析第二章提到的全局 initUI 方法，仅仅需要传入它的名字：

```
 Y.Profiler.registerFunction("initUI");
```

Many functions are bound to objects in order to prevent pollution of the global namespace. Object methods can be registered by passing the object in as the second argument to **registerFunction**. For example, assume an object called **uiTest** that implements two **initUI** approaches as **uiTest.test1** and **uiTest.test2**. Each can be registered individually:

许多函数是与对象绑定的，以防止污染全局命名空间。对象方法也可以通过 reguisterFunction 注册，只要将对象作为第二个参数传入。例如，假设一个称作 uiTest 的对象实现了两个方法，分别为 uiTest.test1 和 uiTest.test2，每个方法都可以独立注册：

```
Y.Profiler.registerFunction("test1", uiTest);
Y.Profiler.registerFunction("test2", uiTest);
```

This works well enough, but doesn't really scale for profiling many functions or an entire application. The **registerObject** method automatically registers every method bound to the object:

一切正常，但还是不能真正测量多个函数或整个应用程序。registerObject 方法自动注册绑定到对象的每一个方法：

```
Y.Profiler.registerObject("uiTest", uiTest);
```

The first argument is the name of the object (for reporting purposes), and the second is the object itself. This will instrument profiling for all of the uiTest methods.

第一个参数是对象的名字（用于报告），第二个参数是对象本身。它将分析 uiTest 的所有方法。

Objects that rely on prototype inheritance need special handling. YUI's profiler allows the registration of a constructor function that will instrument all methods on all instances of the object:

那些从原形继承的对象需要特殊处理。YUI 分析工具允许注册构造器函数，它可以调用对象的所有实例中的所有方法：

```
Y.Profiler.registerConstructor("MyWidget", myNameSpace);
```

Now every function on each instance of **myNameSpace.MyWidget** will be measured and reported on. An individual report can be retrieved as an object:

现在，所有 myNameSpace.MyWidget 实例的每个函数都将被测量并记入报告。一个独立的报告可像获取对象那样获取：

```
var initUIReport = Y.Profiler.getReport("initUI");
```

This provides an object containing the profile data, including an array of points, which are the timings for each call, in the order they were called. These points can be plotted and analyzed in other interesting ways to examine the variations in time. This object has the following fields:

这样得到一个包含分析数据的对象，它包含一个由时间点构成的数组，它们按照调用顺序排列。这些时间点可用于绘图或者用其他感兴趣的方法进行分析，以检查时间上的变化。这个对象具有如下字段：

```
{
  min: 100,
  max: 250,
  calls: 5,
  avg: 120,
  points: [100, 200, 250, 110, 100]
};
```

Sometimes you may want only the value of a particular field. Static **Profiler** methods provide discrete data per function or method:

有时您只关心其中的某些字段。静态 Profiler 方法提供每个函数或方法的离散数据：

```
var uiTest1Report = {
  calls: Y.Profiler.getCalls("uiTest.test1"),
  avg: Y.Profiler.getAvg("uiTest.test1")
};
```

A view that highlights the slowest areas of the code is really what is needed in order to properly analyze a script's performance. A report of all registered functions called on the object or constructor is also available:

一个视图高亮显示出代码中最慢的部分，那也是真正需要分析脚本性能的地方。另外一个功能可报告出对象或构造器所调用的所有已注册的函数：

```
var uiTestReport = Y.Profiler.getReport("uiTest");
```

This returns an object with the following data:

它返回的对象包含如下数据：

```
{
 test1: {
   min: 100,
   max: 250,
   calls: 10,
   avg: 120
 },
 test2:
   min: 80,
   max: 210,
   calls: 10,
   avg: 90
 }
};
```

This provides the opportunity to sort and view the data in more meaningful ways, allowing the slower areas of the code to be scrutinized more closely. A full report of all of the current profile data can also be generated. This, however, may contain useless information, such as functions that were called zero times or that are already meeting performance expectations. In order to minimize this type of noise, an optional function can be passed in to filter the data:

还可以排序以及采用更有意义的方法察看数据，使代码中速度慢的部分得到更密切的检查。获得所有分析数据的完整报告会包含许多无用信息，诸如那些调用次数为零的函数，或者那些性能已经达到预期指标的函数。为降低这些干扰，可传入一个选择函数来过滤这些数据：

```
 var fullReport = Y.Profiler.getFullReport(function(data) {
   return (data.calls > 0 && data.avg > 5);
};
```

The Boolean value returned will indicate whether the function should be included in the report, allowing the less interesting data to be suppressed.

其返回的布尔值用于指出该函数是否应当加入到报告之中，让不感兴趣的数据被抑制掉。

When finished profiling, functions, objects, and constructors can be unregistered individually, clearing the profile data:

当分析完成后，函数，对象，还有构造器应当分别注销，清理分析数据：

```
 Y.Profiler.unregisterFunction("initUI");
Y.Profiler.unregisterObject("uiTests");
Y.Profiler.unregisterConstructor("MyWidget");
```

The **clear()** method keeps the current profile registry but clears the associated data. This function can be called individually per function or timing:

clear()方法保留当前分析目标的注册状态，但清除相关数据。此函数可在每个函数或计时中单独调用：

```
 Y.Profiler.clear("initUI");
```

Or all data may be cleared at once by omitting the name argument:

如果不传参数，那么所有数据都会被一次性清理：

```
 Y.Profiler.clear();
```

Because it is in JSON format, the profile report data can be viewed in any number of ways. The simplest way to view it is on a web page by outputting as HTML. It can also be sent to a server, where it can be stored in a database for more robust reporting. This is especially useful when comparing various optimization techniques across browsers.

因为它使用 JSON 格式，所以分析报告有多种察看方法。最简单的办法就是在网页上输出为 HTML。还可以将它发送到服务器，存入数据库，以实现更强大的报告功能。特别当比较不同的跨浏览器优化技术时特别有用。

It is worth noting that anonymous functions are especially troublesome for this type of profiler because there is no name to report with. The YUI Profiler provides a mechanism for instrumenting anonymous functions, allowing them to be profiled. Registering an anonymous function returns a wrapper function that can be called instead of the anonymous function:

没有什么比匿名函数更难以分析了，因为它们没有名字。YUI 分析器提供了一种调用匿名函数的机制，使得它们可以被分析。注册一个匿名函数会返回一个封装函数，可以调用它而不是调用匿名函数：

```
var instrumentedFunction =
  Y.Profiler.instrument("anonymous1", function(num1, num2){
    return num1 + num2;
  });
instrumentedFunction(3, 5);
```

This adds the data for the anonymous function to the Profiler's result set, allowing it to be retrieved in the same manner as other profile data:

它将匿名函数的数据添加到 Profiler 的返回集中，获取它的方式与其他分析数据相同：

```
var report = Y.Profiler.getReport("anonymous1");
```

## Anonymous Functions 匿名函数

Depending on the profiler, some data can be obscured by the use of anonymous functions or function assignments. As this is a common pattern in JavaScript, many of the functions being profiled may be anonymous, making it difficult or impossible to measure and analyze. The best way to enable profiling of anonymous functions is to name them. Using pointers to object methods rather than closures will allow the broadest possible profile coverage.

使用匿名函数或函数分配会造成分析器的数据模糊。由于这是 JavaScript 的通用模式，许多被分析的函数可能是匿名的，对它们测量和分析很困难或根本无法进行。分析匿名函数的最佳办法是给它们取个名字。使用指针指向对象方法而不是闭包，可以实现最广泛的分析覆盖。

Compare using an inline function:

比较两种方法，其中一个使用内联函数：

```
 myNode.onclick = function() {
   myApp.loadData();
};
```

with a method call:

另一个使用方法调用：

```
 myApp._onClick = function() {
   myApp.loadData();
};
myNode.onclick = myApp._onClick;
```

Using the method call allows any of the reviewed profilers to automatically instrument the **onclick** handler. This is not always practical, as it may require significant refactoring in order to enable profiling.

使用函数调用可使回顾式分析器自动调用 onclick 句柄。这不总是一种实用的方法，因为它可能需要对代码进行大量重构：

For profilers that automatically instrument anonymous functions, adding an inline name makes the reports more readable:

为了让分析器能够自动调用匿名函数，添加一个内联名称使报告更加可读：

```
myNode.onclick = function myNodeClickHandler() {
  myApp.loadData();
};
```

This also works with functions declared as variables, which some profilers have trouble gleaning a name from:

当函数被定义为变量时也可使用这种方法，有些分析器在拾取名称时会遇到麻烦：

```
var onClick = function myNodeClickHandler() {
  myApp.loadData();
};
```

The anonymous function is now named, providing most profilers with something meaningful to display along with the profile results. These names require little effort to implement, and can even be inserted automatically as part of a debug build process.

此匿名函数现在被命名了，使大多数分析器的分析结果显示出有意义的内容。这些命名工作几乎不需要什么工作量，而且可以用开发调试工具自动插入。

**Firebug**

Firefox is a popular browser with developers, partially due to the Firebug addon (available at http://www.getfirebug.com/), which was developed initially by Joe Hewitt and is now maintained by the Mozilla Foundation. This tool has increased the productivity of web developers worldwide by providing insights into code that were never before possible.

对开发人员来说，Firefox 是一个时髦的浏览器，部分原因是 Firebug 插件（http://www.getfirebug.com/）由 Joe Hewitt 首创现在由 Mozilla 基金会维护。此工具具有前所未有的代码洞察力，提高了全世界网页开发者的生产力。

Firebug provides a console for logging output, a traversable DOM tree of the current page, style information, the ability to introspect DOM and JavaScript objects, and more. It also includes a profiler and network analyzer, which will be the focus of this section. Firebug is also highly extensible, enabling custom panels to be easily added.

Firebug 提供了一个控制台日志输出，当前页面的 DOM 树显示，样式信息，能够反观 DOM 和 JavaScript 对象，以及更多功能。它还包括一个性能和网络分析器，这是本节的重点。Firebug 易于扩展，可添加自定义面板。

**Console Panel Profiler 控制台面板分析器**

The Firebug Profiler is available as part of the Console panel (see Figure 10-1). It measures and reports on the execution of JavaScript on the page. The report details each function that is called while the profiler is running, providing highly accurate performance data and valuable insights into what may be causing scripts to run slowly.

Firebug 分析器是控制台面板的一部分（如图 10-1）。它测量并报告页面中运行的 JavaScript。当分析器运行时，报告深入到每个被调用函数的细节，提供高精确的性能数据和变量察看功能，（有助于）找出可能导致脚本运行变慢的原因。

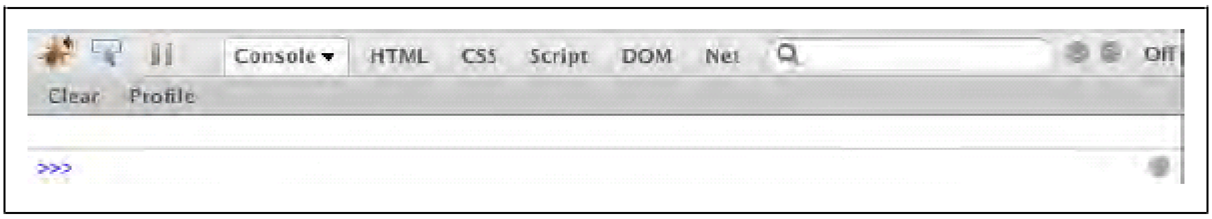
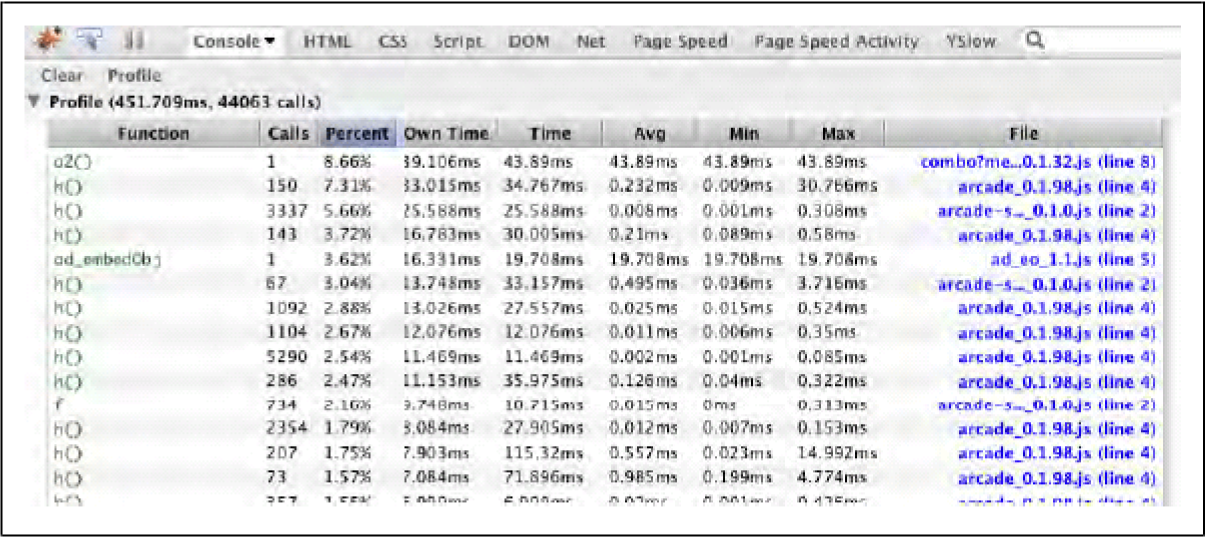


Figure 10-1. FireBug Console panel

图 10-1 FireBug 控制台面板

One way to run a profile is by clicking the Profile button, triggering the script, and clicking the Profile button again to stop profiling. Figure 10-2 shows a typical report of the profile data. This includes Calls, the number of times the function was called; Own Time, the time spent in the function itself; and Time, the overall time spent in a function and any function it may have called. The profiling is instrumented at the browser chrome level, so there is minimal overhead when profiling from the Console panel.

点击 Profile 按钮可启动分析过程，触发脚本，再次点击 Profile 按钮可停止分析。图 10-2 显示了一个典型的分析数据报告。它包括 Calls：函数被调用的次数；Own Time：函数自身运行花费的时间；Time：函数花费的总时间，包括被它调用的函数所花费的时间总和。性能分析过程在浏览器底层调用，所以从控制台面板启动分析时性能开销很小。



▼ Profile (451.709ms, 44063 calls)

| Function | Calls | Percent | Own Time | Time | Avg | Min | Max | File |
|---|---|---|---|---|---|---|---|---|
| a2() | 1 | 8.66% | 39.106ms | 43.89ms | 43.89ms | 43.89ms | 43.89ms | combo?me...0.1.32.js (line 8) |
| h() | 150 | 7.31% | 33.015ms | 34.767ms | 0.232ms | 0.009ms | 30.766ms | arcade_0.1.98.js (line 4) |
| h() | 3337 | 5.66% | 25.588ms | 25.588ms | 0.008ms | 0.001ms | 0.308ms | arcade-s..._0.1.0.js (line 2) |
| h() | 143 | 3.72% | 16.783ms | 30.005ms | 0.21ms | 0.089ms | 0.58ms | arcade_0.1.98.js (line 4) |
| ad_embedObj | 1 | 3.62% | 16.331ms | 19.708ms | 19.708ms | 19.708ms | 19.708ms | ad_eo_1.1.js (line 5) |
| h() | 67 | 3.04% | 13.748ms | 33.157ms | 0.495ms | 0.036ms | 3.716ms | arcade-s..._0.1.0.js (line 2) |
| h() | 1092 | 2.88% | 13.026ms | 27.557ms | 0.025ms | 0.015ms | 0.524ms | arcade_0.1.98.js (line 4) |
| h() | 1104 | 2.67% | 12.076ms | 12.076ms | 0.011ms | 0.006ms | 0.35ms | arcade_0.1.98.js (line 4) |
| h() | 5290 | 2.54% | 11.469ms | 11.469ms | 0.002ms | 0.001ms | 0.085ms | arcade_0.1.98.js (line 4) |
| h() | 286 | 2.47% | 11.153ms | 35.975ms | 0.126ms | 0.04ms | 0.322ms | arcade_0.1.98.js (line 4) |
| f | 734 | 2.16% | 9.748ms | 10.715ms | 0.015ms | 0ms | 0.313ms | arcade-s..._0.1.0.js (line 2) |
| h() | 2354 | 1.79% | 8.084ms | 27.905ms | 0.012ms | 0.007ms | 0.153ms | arcade_0.1.98.js (line 4) |
| h() | 207 | 1.75% | 7.903ms | 115.32ms | 0.557ms | 0.023ms | 14.992ms | arcade_0.1.98.js (line 4) |
| h() | 73 | 1.57% | 7.084ms | 71.896ms | 0.985ms | 0.199ms | 4.774ms | arcade_0.1.98.js (line 4) |
| [illegible] | 357 | 1.55% | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |

Figure 10-2. Firebug Profile panel

图 10-2　Firebug 性能面板

## Console API　终端 API

Firebug also provides a JavaScript interface to start and stop the profiler. This allows more precise control over which parts of the code are being measured. This also provides the option to name the report, which is valuable when comparing various optimization techniques.

Firebug 还提供了 JavaScript 接口用于启动和停止分析器。这可精确控制测量某部分代码。它还提供选项以命名报告，在比较不同的优化技术时特别有价值。

```
console.profile("regexTest");
regexTest('foobar', 'foo');
console.profileEnd();
console.profile("indexOfTest");
```

```
indexOfTest('foobar', 'foo');
console.profileEnd();
```

Starting and stopping the profiler at the more interesting moments minimizes side effects and clutter from other scripts that may be running. One thing to keep in mind when invoking the profiler in this manner is that it does add overhead to the script. This is primarily due to the time required to generate the report after calling **profileEnd()**, which blocks subsequent execution until the report has been generated. Larger reports will take longer to generate, and may benefit from wrapping the call to **profileEnd()** in a **setTimeout**, making the report generation asynchronous and unblocking script execution.

在兴趣点上启动和停止分析器，可减少副作用和其他运行脚本造成的干扰。有一点要记住，以这种方法调用分析器会增加脚本的开销。主要是因为调用 profileEnd()需要花费时间来生成报告，它阻塞后续执行直到报告生成完毕。较大报告需要更长时间来生成，更好的做法是将 profileEnd()调用封装在 setTimeout 中，使报告生成过程可以异步进行而不阻塞脚本运行。

After ending the profile, a new report is generated, showing how long each function took, the number of times called, the percent of the total overhead, and other interesting data. This will provide insight as to where time should be spent optimizing function speeds and minimizing calls.

分析完成之后，生成了一份新的报告，显示出每个函数占用了多长时间，被调用的次数，占总开销的百分比，还有其它感兴趣的数据。这些数据为功夫应当花在优化函数速度上，还是减少调用次数上提供了依据。

Like the YUI Profiler, Firebug's **console.time()** function can help measure loops and other operations that the profiler does not monitor. For example, the following times a small section of code containing a loop:

正如 YUI 分析器，Firebug 的 console.time()函数有助于测量循环和其他分析器不能监视的操作。例如，下面对一小段包含循环的代码进行计时：

```
 console.time("cache node");
for (var box = document.getElementById("box"),
    i = 0;
```

```
     i < 100; i++) {
   value = parseFloat(box.style.left) + 10;
   box.style.left = value + "px";
}
console.timeEnd("cache node");
```

After ending the timer, the time is output to the Console. This can be useful when comparing various optimization approaches. Additional timings can be captured and logged to the Console, making it easy to analyze results side by side. For example, to compare caching the node reference with caching a reference to the node's style, all that is needed is to write the implementation and drop in the timing code:

在定时器结束之后，时间被输出到控制带上。这可用于比较各种优化方法。控制台可以捕获并记录额外的计时结果，因此很容易并排（显示）分析结果。例如，比较缓存节点引用和缓存节点样式的引用，都需要在计时程序中添加实现代码：

```
 console.time("cache style");
for (var style = document.getElementById("box").style,
      i = 0;
      i < 100; i++) {
   value = parseFloat(style.left) + 10;
   style.left = value + "px";
}
console.timeEnd("cache style");
```

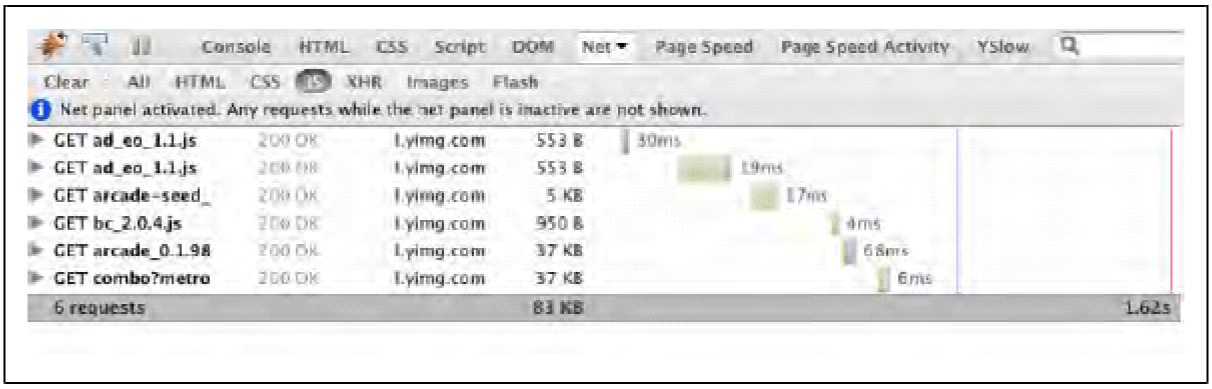
The Console API gives programmers the flexibility to instrument profiling code at various layers, and consolidates the results into reports that can be analyzed in many interesting ways.

控制台 API 使程序员能够灵活地调用不同层次的分析代码，并将结果汇总在报告中，可以用许多感兴趣的方法进行分析。

**Net Panel　网络面板**

Often when encountering performance issues, it is good to step back from your code and take a look at the larger picture. Firebug provides a view of network assets in the Net panel (Figure 10-3). This panel provides a visualization of the pauses between scripts and other assets, providing deeper insight into the effect the script is having on the loading of other files and on the page in general.

通常，当遇到性能问题时，最好从代码中退出来，看看更大的图景。Firebug 在网络面板中提供了一个网络资源视图（如图 10-3）。此面板提供了脚本和其他资源的静态视图，可深入探查脚本对其它文件加载造成的影响和对页面造成的一般影响。



Net panel activated. Any requests while the net panel is inactive are not shown.

```
GET ad_eo_1.1.js      200 OK   l.yimg.com   553 B   30ms
GET ad_eo_1.1.js      200 OK   l.yimg.com   553 B   19ms
GET arcade-seed_      200 OK   l.yimg.com     5 KB  17ms
GET bc_2.0.4.js       200 OK   l.yimg.com   950 B   4ms
GET arcade_0.1.98     200 OK   l.yimg.com    37 KB  68ms
GET combo?metro       200 OK   l.yimg.com    37 KB  6ms
6 requests                                   83 KB          1.62s
```

Figure 10-3. Firebug Net panel

图 10-3 Firebug 网络面板

The colored bars next to each asset break the loading life cycle into component phases (DNS lookup, waiting for response, etc.). The first vertical line (which displays as blue) indicates when the page's **DOMContentLoaded** event has fired. This event signals that the page's DOM tree is parsed and ready. The second vertical line (red) indicates when the window's **load** event has fired, which means that the DOM is ready and all external assets have completed loading. This gives a sense as to how much time is spent parsing and executing versus page rendering.

每个资源后面的彩条将加载过程分解为组件阶段（DNS 察看，等待响应，等等）。第一条垂直线（显示为蓝色）指出页面的 DOMContentLoaded 事件发出的时间。此事件表明页面的 DOM 树已经解析并准备好了。第二条垂直线（红色）指出 window 的 load 事件发出的时间，它表示 DOM 已准备好并且所有外部资源已完成加载。这样就给出了一个场景，关于解析和运行以及页面渲染所花费的时间。

As you can see in the figure, there are a number of scripts being downloaded. Based on the timeline, each script appears to be waiting for the previous script prior to starting the next request. The simplest optimization to improve loading performance is to reduce the number of requests, especially script and stylesheet requests, which can block other assets and page rendering. When possible, combine all scripts into a single file in order to minimize the total number of requests. This applies to stylesheets and images as well.

正如你在图中看到的，下载了很多脚本。在时间线上，每个脚本看上去要等待前面的脚本首先启动下一个请求。提高加载性能的最简单的优化办法是减少请求数量，特别是脚本和样式表请求，它们会阻塞其它资源和页面渲染。如果可能的话，将所有脚本合并为一个文件，以减少总的请求数量。这种方法对样式表和图片同样有用。

## Internet Explorer Developer Tools IE 开发人员工具

As of version 8, Internet Explorer provides a development toolkit that includes a profiler. This toolkit is built into IE 8, so no additional download or installation is required. Like Firebug, the IE profiler includes function profiling and provides a detailed report that includes the number of calls, time spent, and other data points. It adds the ability to view the report as a call tree, profile native functions, and export the profile data. Although it lacks a network analyzer, the profiler can be supplemented with a generic tool such as Fiddler, which is outlined later in this chapter. See http://msdn.microsoft.com/en-us/library/dd565628(VS.85).aspx for more details.

Internet Explorer 8 提供了一个开发工具包，它包含一个分析器。此工具包内建于 IE 8，所以不需要额外的下载和安装。像 Firebug 一样，IE 分析器包括函数分析和细节报告，可以指出调用次数，花费时间，还有其它数据点。它能够以调用树形式察看报告，分析原生函数，并导出分析数据。虽然它没有网络分析器，但是此分析器可以增补一个通用工具注入 Fiddler，它将在本章稍后部分介绍。更多详细信息参见 http://msdn.microsoft.com/en-us/library/dd565628(VS.85).aspx

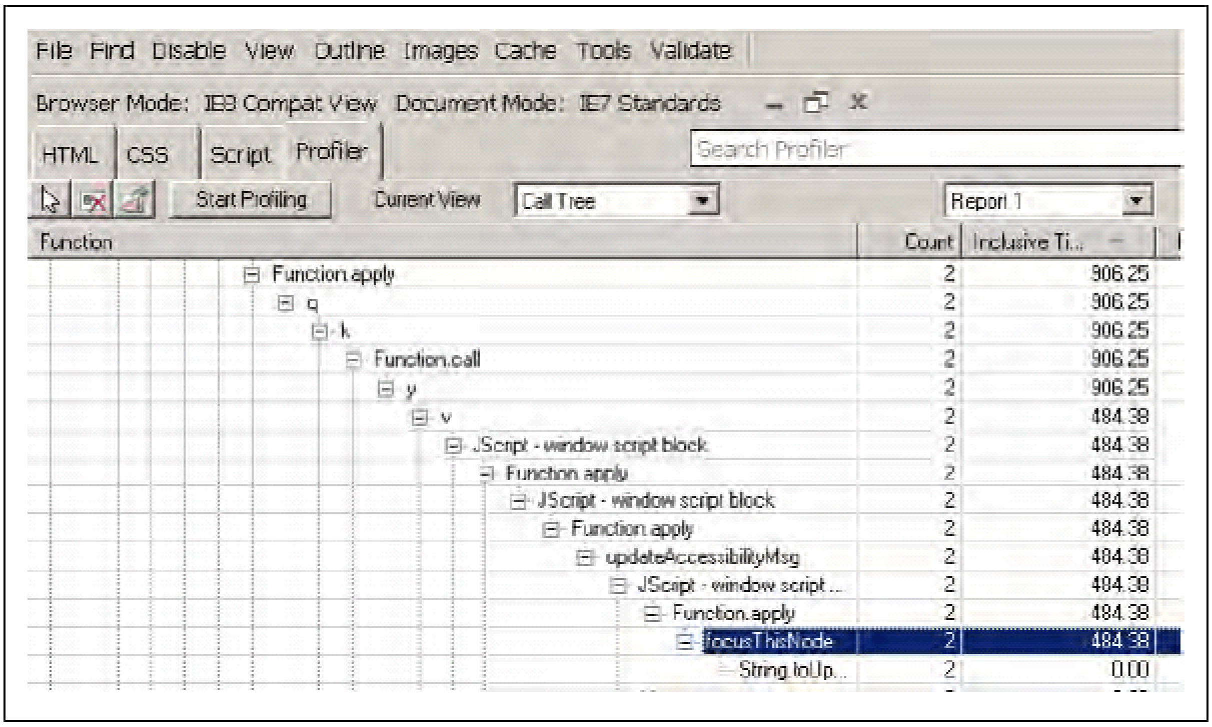
IE 8's Profiler can be found with the Developer Tools (Tools → Developer Tools). After pressing the Start Profiling button, all subsequent JavaScript activity is monitored and profiled. Clicking Stop Profiling (same button, new label) stops the profiler and generates a new report. By default, F5 starts the profiler and Shift-F5 ends it.

IE 8 的性能分析器（译者注：简体中文版称作“探查器”）可以在开发人员工具中找到（工具菜单 → 开发人员工具）。在按下 Start Profiling 按钮（译者注：简体中文版称作“开始配置文件”）之后，所有后续的

JavaScript 活动都会被监视并分析。点击 Stop Profiling（同一个按钮，只是上面的文字标签变了）（译者注：简体中文版称作"停止配置文件"）将停止分析器并生成一个新的报告。默认快捷键是 F5 启动分析器，Shift-F5 停止它。

The report provides both a flat function view of the time and duration of each call and a tree view showing the function call stack. The tree view allows you to walk through the call stack and identify slow code paths (see Figure 10-4). The IE profiler will use the variable name when no name is available for the function.

报告显示了一个平面函数视图，包含每次调用的时间和持续时间。还有一个树状视图，显示了函数调用栈。树形视图使你可以遍历察看调用栈并定位出缓慢代码的路径（参见图 10-4）。IE 分析器将使用函数的变量名来显示匿名函数。


File Find Disable View Outline Images Cache Tools Validate
Browser Mode: IE8 Compat View Document Mode: IE7 Standards
HTML CSS Script Profiler
Search Profiler
Start Profiling
Current View
Call Tree
Report 1

| Function | Count | Inclusive Ti... |
|---|---|---|
| Function.apply | 2 | 906.25 |
| q | 2 | 906.25 |
| k | 2 | 906.25 |
| Function.call | 2 | 906.25 |
| y | 2 | 906.25 |
| v | 2 | 484.38 |
| JScript - window script block | 2 | 484.38 |
| Function.apply | 2 | 484.38 |
| JScript - window script block | 2 | 484.38 |
| Function.apply | 2 | 484.38 |
| updateAccessibilityMsg | 2 | 484.38 |
| JScript - window script ... | 2 | 484.38 |
| Function.apply | 2 | 484.38 |
| focusThisNode | 2 | 484.38 |
| String.toUp... | 2 | 0.00 |

Figure 10-4. IE 8 Profiler call tree

图 10-4 IE 8 分析器的调用树

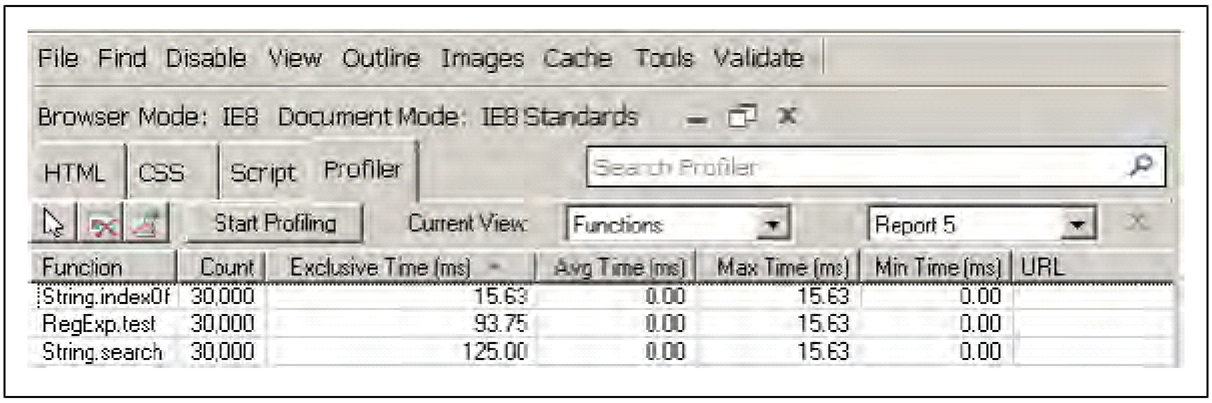
The IE Profiler also provides insight into native JavaScript object methods. This allows you to profile native objects in addition to implementation code, and makes it possible to do things such as compare **String::indexOf** with **RegExp::test** for determining whether an HTML element's **className** property begins with a certain value:

IE 分析器还可以观察原生 JavaScript 对象方法。你可以添加调用代码来分析原生对象，那么就能够比较 String::indexOf 和 RegExp::test，用于确定一个 HTML 元素的 className 属性是否具有特定值。

```
 var count = 10000,
     element = document.createElement_x('div'),
     result, i, time;


element.className = 'foobar';


for (i = 0; i < count; i++) {
   result = /^foo/.test(element.className);
}


for (i = 0; i < count; i++) {
   result = element.className.search(/^foo/);
}


for (i = 0; i < count; i++) {
   result = (element.className.indexOf('foo') === 0);
}
```

As seen in Figure 10-5, there appears to be a wide variation in time between these various approaches. Keep in mind that the average time of each call is zero. Native methods are generally the last place to look for optimizations, but this can be an interesting experiment when comparing approaches. Also keep in mind that with numbers this small, the results can be inconclusive due to rounding errors and system memory fluctuations.

如图 10-5 所示，这些不同方法所用的时间差异很大。注意，每次调用的平均时间是零。原生函数通常是最后寻找优化的地方，但这是一个有趣的比较实验。同时请注意，由于这个数字很小，可能由于舍入误差和系统内存波动而无法得出确切结论。


File Find Disable View Outline Images Cache Tools Validate
Browser Mode: IE8 Document Mode: IE8 Standards
HTML CSS Script Profiler
Search Profiler
Start Profiling
Current View: Functions
Report 5

| Function | Count | Exclusive Time (ms) | Avg Time (ms) | Max Time (ms) | Min Time (ms) | URL |
|---|---|---|---|---|---|---|
| String.indexOf | 30,000 | 15.63 | 0.00 | 15.63 | 0.00 | |
| RegExp.test | 30,000 | 93.75 | 0.00 | 15.63 | 0.00 | |
| String.search | 30,000 | 125.00 | 0.00 | 15.63 | 0.00 | |

Figure 10-5. Profile results for native methods

图 10-5 原生方法的分析结果

Although the IE Profiler does not currently offer a JavaScript API, it does have a console API with logging capabilities. This can be leveraged to port the **console.time()** and **console.timeEnd()** functions over from Firebug, allowing the same tests to run in IE.

虽然 IE 分析器目前不提供 JavaScript 的 API，但它有一个具有日志功能的控制台 API。可以将 console.time() 和 console.timeEnd()函数从 Firebug 上移植过来，从而在 IE 上进行同样的测试。

```
if (console && !console.time) {
  console._timers = {};
  console.time = function(name) {
    console._timers[name] = new Date();
  };
  console.timeEnd = function(name) {
    var time = new Date() - console._timers[name];
  console.info(name + ': ' + time + 'ms');
  };
}
```

## Safari Web Inspector　Safari 网页监察器

Safari, as of version 4, provides a profiler in addition to other tools, including a network analyzer, as part of its bundled Web Inspector. Like the Internet Explorer Developer Tools, the Web Inspector profiles native functions and provides an expandable call tree. It also includes a Firebug-like console API with profiling functionality, and a Resource panel for network analysis.

Safari 4 提供了分析器等工具，包括网络分析器，它作为网页检查器的一部分。正如 Internet Explorer 开发人员工具那样，网页检查器可以分析原生函数并提供一个可以展开的调用树（视图）。它还包括一个类似 Firebug 具有分析功能的控制台 API，网络分析器还具有一个资源面板。

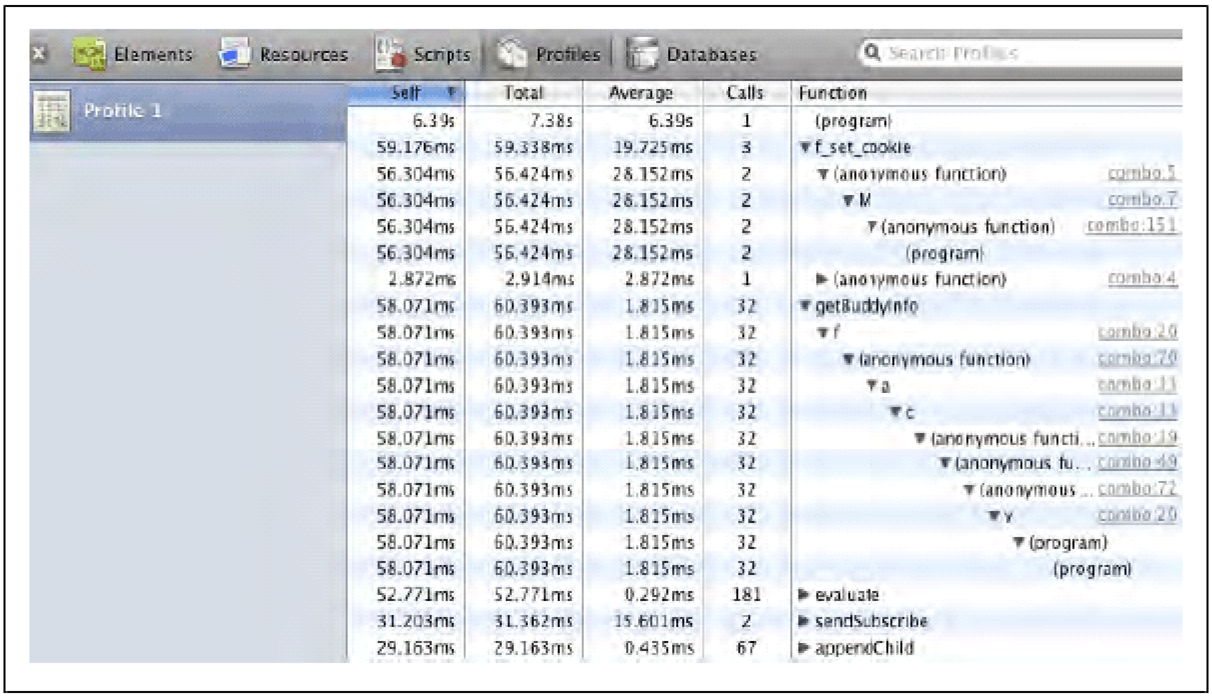
To access the Web Inspector, first make sure that the Develop menu is available. The Develop menu can be enabled by opening Preferences → Advanced and checking the "Show Develop menu in menu bar" box. The Web Inspector is then available under Develop → Show Web Inspector (or the keyboard shortcut Option-Command-I).

要访问网页检查器，首先确定 Develop 菜单是否可用。可通过 Preferences → Advanced 菜单命令，选中"在菜单栏上显示开发菜单"。然后可以用 Develop → Show Web Inspector 打开网页检查器（键盘快捷命令是 Option-Command-I）。

### Profiles Panel　分析面板

Clicking the Profile button brings up the Profile panel (Figure 10-6). Click the Enable Profiling button to enable the Profiles panel. To start profiling, click the Start Profiling button (the dark circle in the lower right). Click Stop Profiling (same button, now red) to stop the profile and show the report.

点击 Profile 按钮打开分析面板（如图 10-6）。点击 Enable Profiling 按钮启用分析面板。点击 Start Profiling 按钮开始分析（右下方的暗色圆圈）。点击 Stop Profiling（同一个按钮，现在是公司）停止分析并便是报告。



| Self | Total | Average | Calls | Function | |
|---|---|---|---|---|---|
| 6.39s | 7.38s | 6.39s | 1 | (program) | |
| 59.176ms | 59.338ms | 19.725ms | 3 | ▼f_set_cookie | |
| 56.304ms | 56.424ms | 28.152ms | 2 | ▼(anonymous function) | combo:5 |
| 56.304ms | 56.424ms | 28.152ms | 2 | ▼M | combo:7 |
| 56.304ms | 56.424ms | 28.152ms | 2 | ▼(anonymous function) | combo:151 |
| 56.304ms | 56.424ms | 28.152ms | 2 | (program) | |
| 2.872ms | 2.914ms | 2.872ms | 1 | ▶(anonymous function) | combo:4 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼getBuddyInfo | |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼f | combo:20 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼(anonymous function) | combo:70 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼a | combo:11 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼c | combo:11 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼(anonymous functi... | combo:19 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼(anonymous fu... | combo:49 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼(anonymous ... | combo:72 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼Y | combo:20 |
| 58.071ms | 60.393ms | 1.815ms | 32 | ▼(program) | |
| 58.071ms | 60.393ms | 1.815ms | 32 | (program) | |
| 52.771ms | 52.771ms | 0.292ms | 181 | ▶evaluate | |
| 31.203ms | 31.362ms | 15.601ms | 2 | ▶sendSubscribe | |
| 29.163ms | 29.163ms | 0.435ms | 67 | ▶appendChild | |

Figure 10-6. Safari Web Inspector Profile panel

图 10-6　Safari 的网页检查器分析面板

Safari has emulated Firebug's JavaScript API (**console.profile()**, **console.time()**, etc.) in order to start and stop profiling programmatically. The functionality is the same as Firebug's, allowing you to name reports and timings for better profile management.

Safari 模仿了 Firebug 的 JavaScript API（console.profile()，console.time()，等等），可以用程序启动和停止分析功能。此功能与 Firebug 的完全相同，允许你对报告和计时进行命名以提供更好的分析管理。

Safari provides both a Heavy (bottom-up) view of the profiled functions and a Tree (top-down) view of the call stack. The default Heavy view sorts the slowest functions first and allows traversal up the call stack, whereas the Tree view allows drilling from the top down into the execution path of the code from the outermost caller. Analyzing the call tree can help uncover more subtle performance issues related to how one function might be calling another.

Safari 提供了一个重视图（由下而上）用于函数分析，和一个树视图（由上而下）用于显示调用栈。默认的重视图将最慢的函数排在前面，并允许遍历调用栈，而树视图可从上至下地从最外层调用进入代码的运行路径。分析调用树有助于揭露与函数调用方法相关的性能问题。

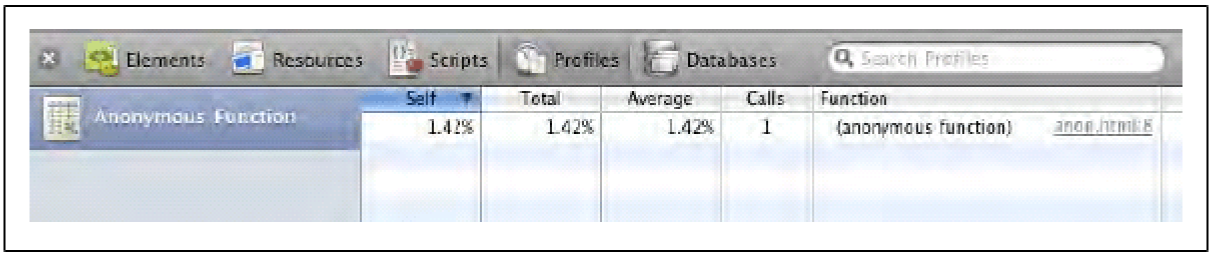
Safari has also added support for a property called **displayName** for profiling purposes. This provides a way to add names to anonymous functions that will be used in the report output. Consider the following function assigned to the variable **foo**:

Safari 还增加了一个名为 displayName 的属性专用于分析目的。它提供了一种为匿名函数在输出报告中添加名称的方法。考虑下面这个赋值给变量 foo 的函数：

```
var foo = function() {
  return 'foo!';
};
console.profile('Anonymous Function');
foo();
console.profileEnd();
```

As shown in Figure 10-7, the resulting profile report is difficult to understand because of the lack of function names. Clicking on the URL to the right of the function shows the function in the context of the source code.

如图 10-7 所示，因为缺少函数名，分析报告难以理解。点击函数右面的 URL 以源码方式显示出函数内容。



| Self | Total | Average | Calls | Function |
|---|---|---|---|---|
| 1.42% | 1.42% | 1.42% | 1 | (anonymous function) anon.html:8 |

Figure 10-7. Web Inspector Profile panel showing anonymous function

图 10-7　网页检查器的分析面板显示匿名函数

Adding a **displayName** will make the report readable. This also allows for more descriptive names that are not limited to valid function names.

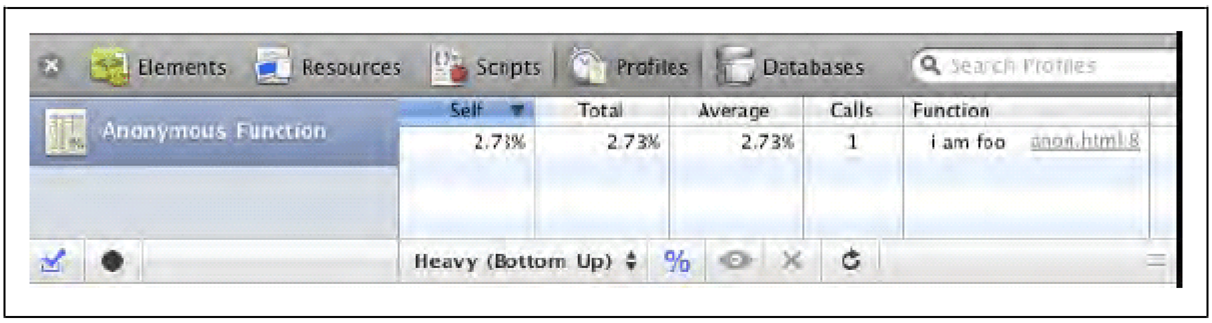
添加一个 displayName 属性将使报告变得可读。可添加更具有描述意义的名字而不仅限于函数名。

```
var foo = function() {
  return 'foo!';
};
foo.displayName = 'I am foo';
```

As shown in Figure 10-8, the **displayName** now replaces the anonymous function. However, this property is available only in Webkit-based browsers. It also requires refactoring of truly anonymous functions, which is not advised. As discussed earlier, adding the name inline is the simplest way to name anonymous functions, and this approach works with other profilers:

如图 10-8 所示，displayName 现在取代了匿名函数。但是，此属性仅适用于基于 Webkit 的浏览器。它还要求重构真正的匿名函数，所以不建议这么做。正如前面所讨论过的，添加内联名称是命名匿名函数最简单的方法，而且可以在其他分析器上工作。

```
var foo = function foo() {
  return 'foo!';
};
```



| Self | Total | Average | Calls | Function |
|---|---|---|---|---|
| 2.73% | 2.73% | 2.73% | 1 | i am foo anon.html:8 |

Figure 10-8. Web Inspector Profile panel showing displayName

图 10-8　网页检查器的分析面板显示了 displayName 属性

**Resources Panel　资源面板**

The Resources panel helps you better understand how Safari is loading and parsing scripts and other external assets. Like Firebug's Net panel, it provides a view of the resources, showing when a request was initiated and how long it took. Assets are conveniently color-coded to enhance readability. Web Inspector's Resources panel separates the size charting from time, minimizing the visual noise (see Figure 10-9).

资源面板可帮助您更好地理解 Safari 加载和解析脚本以及其他外部资源的方式。就像 Firebug 的网络面板，它提供了一个资源视图，显示出一个发起的请求以及它持续了多长时间。资源以不同颜色显示以方便察看。网页检查器的资源面板将尺寸表与时间表分开，缩小了视觉上的干扰（如图 10-9）。

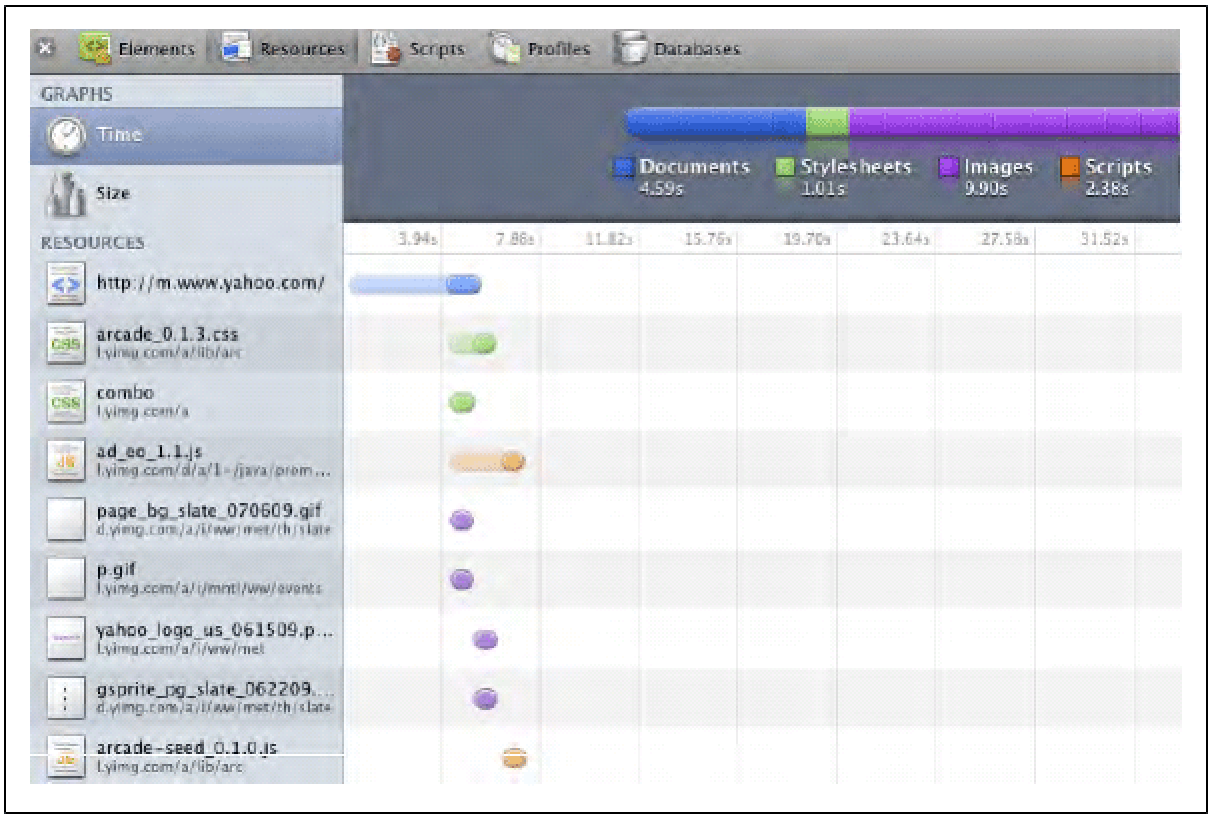


Figure 10-9. Safari Resources panel

图 10-9　Safari 的资源面板

Notice that unlike some browsers, Safari 4 is loading scripts in parallel and not blocking. Safari gets around the blocking requirement by ensuring that the scripts execute in the proper order. Keep in mind that this only applies to scripts initially embedded in HTML at load; dynamically added scripts block neither loading nor execution (see Chapter 1).

请注意，不像某些浏览器那样，Safari 4 能够并行加载脚本而不会互相阻塞。Safari 绕过被阻塞的请求，还要确保脚本按照正确的顺序执行。请记住，这仅适用于 HTML 加载时嵌入的那些初始脚本，动态添加的脚本块不会被加载，也不会被运行（参见第一章）。

**Chrome Developer Tools   Chrome** 开发人员工具

Google has also provided a set of development tools for its Chrome browser, some of which are based on the WebKit/Safari Web Inspector. In addition to the Resources panel for monitoring network traffic, Chrome adds a Timeline view of all page and network events. Chrome includes the Web Inspector Profiles panel, and adds the ability to take "heap" snapshots of the current memory. As with Safari, Chrome profiles native functions and implements the Firebug Console API, including **console.profile** and **console.time**.

Google 也为它的 Chrome 浏览器提供了一套开发工具集，有一些基于 WebKit/Safari 网页检查器。除了监视网络流量的资源面板之外，Chrome 为所有页面和网络事件添加了一个时间线视图。Chrome 包含网页检查器的分析面板，增加了对当前“堆内存”的快照功能。正如 Safari 那样，Chrome 能够分析原生函数并实现了 Firebug 的控制台 API，包括 console.profile 和 console.time。

As shown in Figure 10-10, the Timeline panel provides an overview of all activities, categorized as either "Loading", "Scripting," or "Rendering". This enables developers to quickly focus on the slowest aspects of the system. Some events contain a subtree of other event rows, which can be expanded or hidden for more or less detail in the Records view.

如图 10-10 所示，时间线面板提供了所有活动的概况，按类别分为“加载”，“脚本”，或“渲染”。这使得开发人员可以快速定位系统中速度最慢的部分。某些事件包含其他事件行的子树，在报告视图中可以展开或隐藏以显示更多或更少的细节。

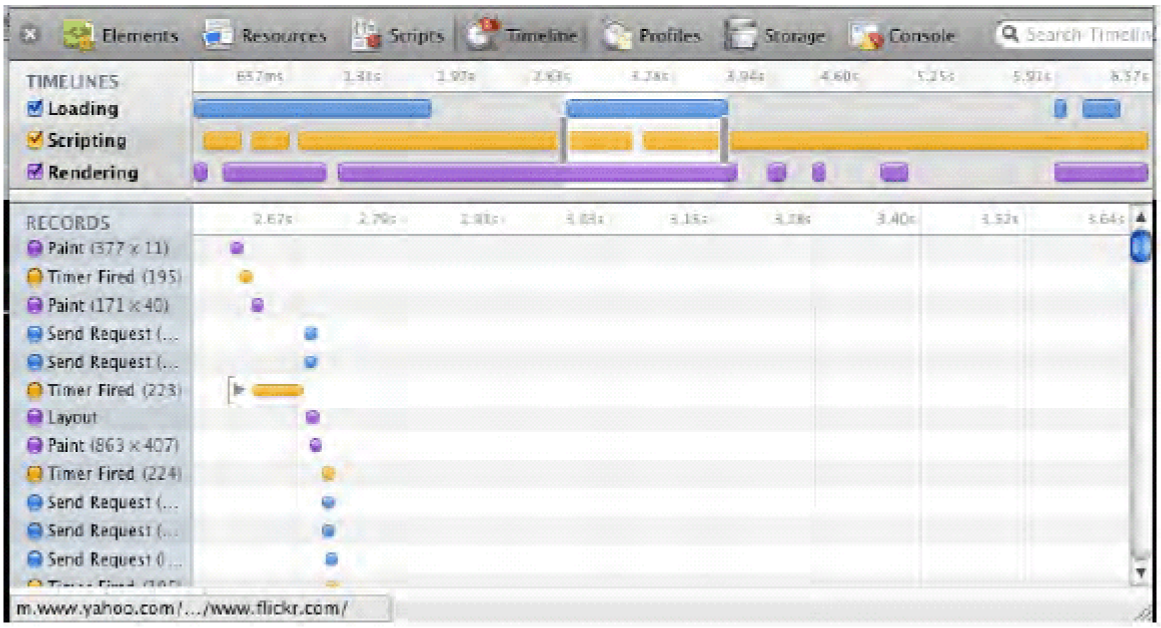
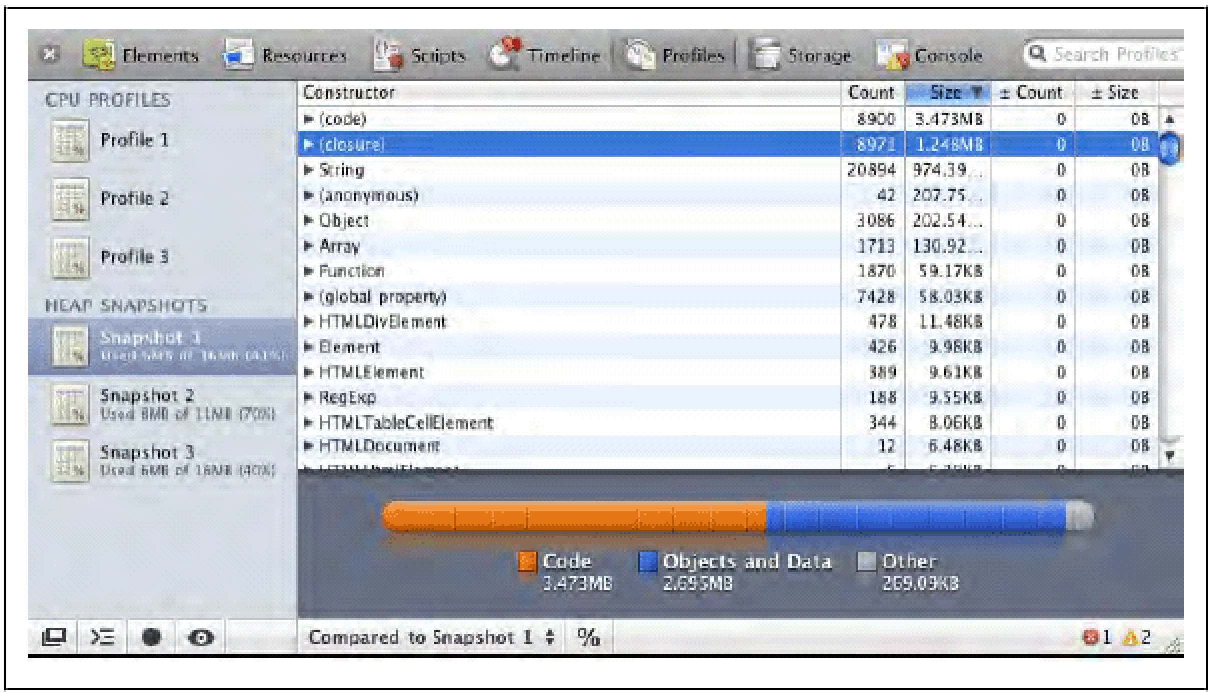


Figure 10-10. Chrome Developer Tools Timeline panel

图 10-10 Chrome 开发者工具的时间线面板

Clicking the eye icon on Chromes Profiles panel takes a snapshot of the current JavaScript memory heap (Figure 10-11). The results are grouped by constructor, and can be expanded to show each instance. Snapshots can be compared using the "Compared to Snapshot" option at the bottom of the Profiles panel. The +/- Count and Size columns show the differences between snapshots.

点击 Chrome 分析面板上的眼睛图标，可获得当前 JavaScript 堆内存的快照（图 10-11）。其结果按照构造器分组，可以展开察看每个实例。快照可使用性能面板底部的“比较快照”选项来进行比较。带+/-号的 Count 列和 Size 列显示出快照之间的差异。



| Constructor | Count | Size ▼ | ± Count | ± Size |
|---|---|---|---|---|
| ▸ (code) | 8900 | 3.473MB | 0 | 0B |
| ▸ (closure) | 8971 | 1.248MB | 0 | 0B |
| ▸ String | 20894 | 974.39... | 0 | 0B |
| ▸ (anonymous) | 42 | 207.75... | 0 | 0B |
| ▸ Object | 3086 | 202.54... | 0 | 0B |
| ▸ Array | 1713 | 130.92... | 0 | 0B |
| ▸ Function | 1870 | 59.17KB | 0 | 0B |
| ▸ (global property) | 7428 | 58.03KB | 0 | 0B |
| ▸ HTMLDivElement | 478 | 11.48KB | 0 | 0B |
| ▸ Element | 426 | 9.98KB | 0 | 0B |
| ▸ HTMLElement | 389 | 9.61KB | 0 | 0B |
| ▸ RegExp | 188 | 9.55KB | 0 | 0B |
| ▸ HTMLTableCellElement | 344 | 8.06KB | 0 | 0B |
| ▸ HTMLDocument | 12 | 6.48KB | 0 | 0B |

🟧 Code 3.473MB
🟦 Objects and Data 2.695MB
▨ Other 269.03KB

Figure 10-11. Chrome Developer Tools JavaScript heap snapshot

图 10-11　Chrome 开发人员工具的 JavaScript 堆内存快照

## Script Blocking　脚本阻塞

Traditionally, browsers limit script requests to one at a time. This is done to manage dependencies between files. As long as a file that depends on another comes later in the source, it will be guaranteed to have its dependencies ready prior to execution. The gaps between scripts may indicate script blocking. Newer browsers such as Safari 4, IE 8, Firefox 3.5, and Chrome have addressed this by allowing parallel downloading of scripts but blocking execution, to ensure dependencies are ready. Although this allows the assets to download more quickly, page rendering is still blocked until all scripts have executed.

传统上，浏览器每次只能发出一个脚本请求。这样做是为了管理文件之间的以来关系。只要一个文件依赖于另一个在源码中靠后的文件，它所依赖的文件将保证在它运行之前被准备好。脚本之间的差距表明脚本被阻塞了。新式浏览器诸如 Safari 4，IE 8，Firefox 3.5，和 Chrome 解决这个问题的办法是允许并行下载，但阻塞式运行，以保证依赖体已经准备好了。虽然这使得资源下载更快，页面渲染仍旧会阻塞，直至所有脚本都被执行。

Script blocking may be compounded by slow initialization in one or more files, which could be worthy of some profiling, and potentially optimizing or refactoring. The loading of scripts can slow or stop the rendering of the page, leaving the user waiting. Network analysis tools can help identify and optimize gaps in the loading of assets. Visualizing these gaps in the delivery of scripts gives an idea as to which scripts are slower to execute. Such scripts may be worth deferring until after the page has rendered, or possibly optimizing or refactoring to reduce the execution time.

脚本阻塞将因为一个或多个文件初始化缓慢而变得更加严重，值得对它做某些类型的分析，并有可能优化或重构。脚本加载会减慢或停止页面渲染，造成用户等待。网络分析工具有助于找出并优化加载资源之间差距。以图形显示出传送脚本时的差异，可找出那些运行较慢的脚本。这些脚本也许应该推迟到页面渲染之后再加载，或者尽可能优化或重构以减少运行时间。

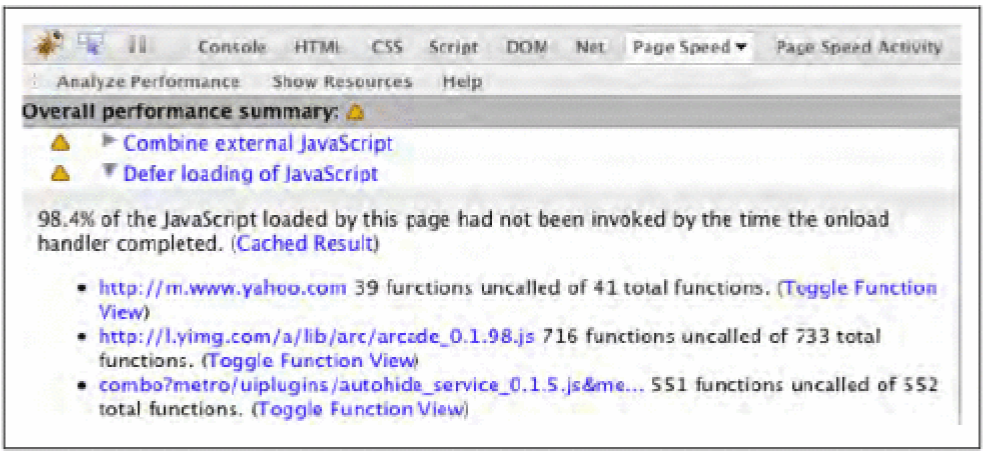
**Page Speed**

Page Speed is a tool initially developed for internal use at Google and later released as a Firebug addon that, like Firebug's Net panel, provides information about the resources being loaded on a web page. However, in addition to load time and HTTP status, it shows the amount of time spent parsing and executing JavaScript, identifies deferrable scripts, and reports on functions that aren't being used. This is valuable information that can help identify areas for further investigation, optimization, and possible refactoring. Visit http://code.google.com/speed/page-speed/ for installation instructions and other product details.

Page Speed 最初是 Google 内部开发所使用的一个工具，后来作为 Firebug 插件发布，像 Firebug 的网络面板一样提供了关于页面资源加载的信息。然而，除了加载时间和 HTTP 状态，它还显示 JavaScript 解析和运行所花费的时间，指出造成延迟的脚本，并报告那些没有被使用的函数。这些有价值的信息可帮助确定进一步调查的方向，优化，以及可能的重构。访问 http://code.google.com/speed/page-speed/关于安装及其他产品细节。

The Profile Deferrable JavaScript option, available on the Page Speed panel, identifies files that can be deferred or broken up in order to deliver a smaller initial payload. Often, very little of the script running on a page is required to render the initial view. In Figure 10-12 you can see that a majority of the code being loaded is not used prior to the window's **load** event firing. Deferring code that isn't being used right away allows the initial page to load much faster. Scripts and other assets can then be selectively loaded later as needed.

Page Speed 面板上的分析延迟 JavaScript 选项，指出哪些文件可造成延迟或者可以拆分为一个较小的初始化载荷。通常，页面上运行的脚本极少需要渲染初始视图。在图 10-12 中您可以看到，大部分代码加载之后，不会在 window 发出 load 事件之前被用到。延迟（加载）这些不会被立刻用到的代码可使得初始化页面加载过程更快。如果需要的话，脚本和其它资源可以有选择地稍后加载。



Overall performance summary:

▶ Combine external JavaScript

▼ Defer loading of JavaScript

98.4% of the JavaScript loaded by this page had not been invoked by the time the onload handler completed. (Cached Result)

- http://m.www.yahoo.com 39 functions uncalled of 41 total functions. (Toggle Function View)
- http://l.yimg.com/a/lib/arc/arcade_0.1.98.js 716 functions uncalled of 733 total functions. (Toggle Function View)
- combo?metro/uiplugins/autohide_service_0.1.5.js&me... 551 functions uncalled of 552 total functions. (Toggle Function View)

Figure 10-12. Page Speed deferrable JavaScript summary

图 10-12　Page Speed 的延迟 JavaScript 总结

Page Speed also adds a Page Speed Activity panel to Firebug. This panel is similar to Firebug's own Net panel, except that it provides more granular data about each request. This includes a breakdown of each script's life cycle, including parse and execution phases, giving a detailed account of the gaps between scripts. This can help identify areas where profiling and possible refactoring of the files are needed. As seen in the legend, Figure 10-13 shows the amount of time spent parsing the script in red and the time executing in blue. A long execution time may be worth looking into more closely with a profiler.

Page Speed 还为 Firebug 增加了一个页面速度活动面板。此面板类似于 Firebug 自己的网络面板，不过它为每个请求提供了更加精细的数据。其中包括每个脚本生命周期的统计分析——解析和运行阶段，给出了脚本之间时间差异的详细报告。这有助于分析特定区域并在需要的情况下重构这些文件。正如传说中的图 10-13 显示出红色的脚本解析时间和蓝色的运行时间。运行时间太长的脚本更需要我们用分析器深入研究。

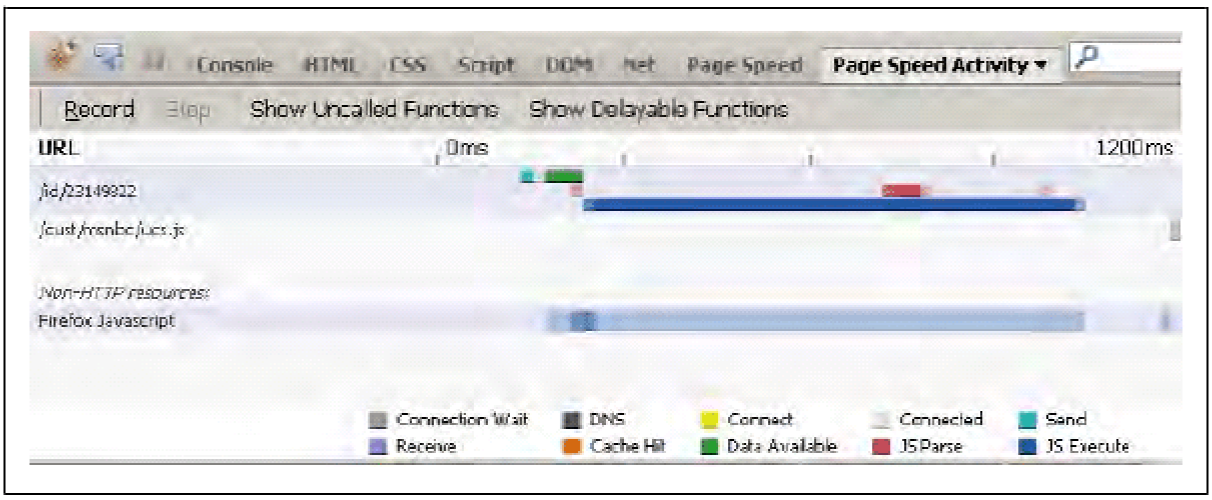
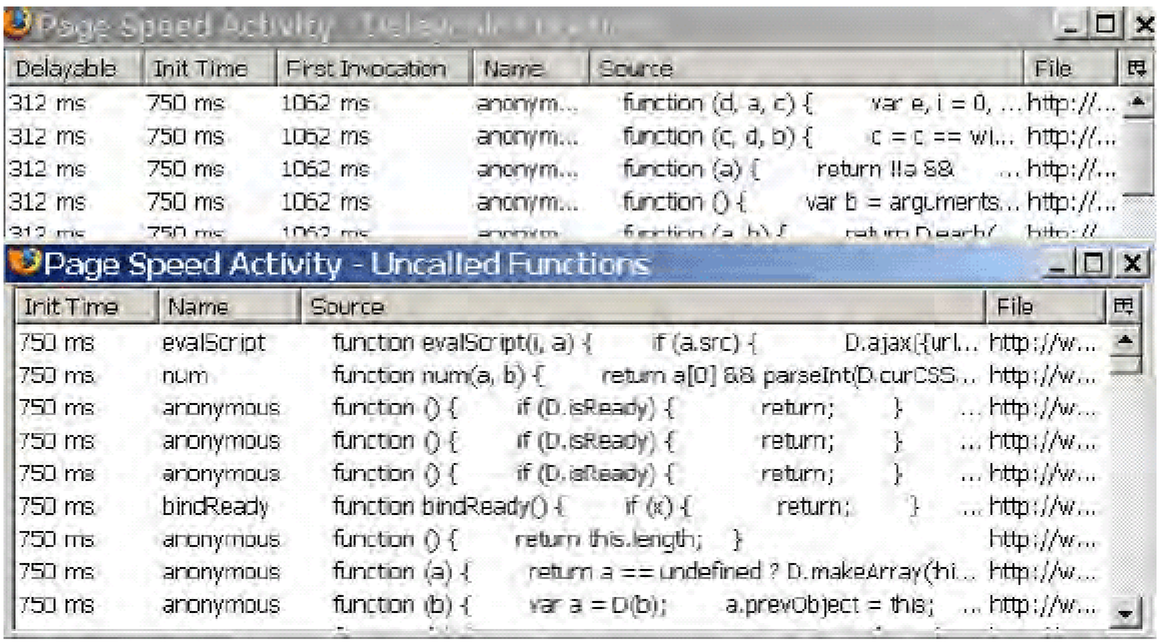


Figure 10-13. Page Speed parse and execution times

图 1-=13　Page Speed 的解析和运行时间

There may be significant time spent parsing and initializing scripts that are not being used until after the page has rendered. The Page Speed Activity panel can also provide a report on which functions were not called at all and which functions may be delayable, based on the time they were parsed versus the time they were first called (Figure 10-14).

可能有大量时间花费在解析和初始化脚本上，而这些脚本在页面渲染之后还没有用到。页面速度动作面板还提供报告指出哪些函数从来没有被调用过，哪些函数可以延迟使用，基于它们的解析时间以及它们第一次被调用的时间（如图 10-14）。



| Delayable | Init Time | First Invocation | Name | Source | File |
|---|---|---|---|---|---|
| 312 ms | 750 ms | 1062 ms | anonym... | function (d, a, c) { var e, i = 0, ... | http://... |
| 312 ms | 750 ms | 1062 ms | anonym... | function (c, d, b) { c = c == wi... | http://... |
| 312 ms | 750 ms | 1062 ms | anonym... | function (a) { return !!a && ... | http://... |
| 312 ms | 750 ms | 1062 ms | anonym... | function () { var b = arguments... | http://... |
| 312 ms | 750 ms | 1062 ms | anonym | function (a, b) { return D.each( | http:// |

| Init Time | Name | Source | File |
|---|---|---|---|
| 750 ms | evalScript | function evalScript(i, a) { if (a.src) { D.ajax({url... | http://w... |
| 750 ms | num | function num(a, b) { return a[0] && parseInt(D.curCSS... | http://w... |
| 750 ms | anonymous | function () { if (D.isReady) { return; } ... | http://w... |
| 750 ms | anonymous | function () { if (D.isReady) { return; } ... | http://w... |
| 750 ms | anonymous | function () { if (D.isReady) { return; } ... | http://w... |
| 750 ms | bindReady | function bindReady() { if (x) { return; } ... | http://w... |
| 750 ms | anonymous | function () { return this.length; } | http://w... |
| 750 ms | anonymous | function (a) { return a == undefined ? D.makeArray(thi... | http://w... |
| 750 ms | anonymous | function (b) { var a = D(b); a.prevObject = this; ... | http://w... |

Figure 10-14. Reports for delayable and uncalled functions

图 10-14　可延迟函数和未调用函数的报告

These reports show the amount of time spent initializing the function that are either never called or that could be called later. Consider refactoring code to remove uncalled functions and to defer code that isn't needed during the initial render and setup phase.

此报告显示了那些从未被调用过或者以后才会被调用的函数初始化所用的总时间。考虑重构代码删除那些未被调用的函数，并且延迟加载那些在初始渲染和设置阶段用不到的代码。

**Fiddler**

Fiddler is an HTTP debugging proxy that examines the assets coming over the wire and helps identify any loading bottlenecks. Created by Eric Lawrence, this is a general purpose network analysis tool for Windows that provides detailed reports on any browser or web request. Visit http://www.fiddler2.com/fiddler2/ for installation and other information.

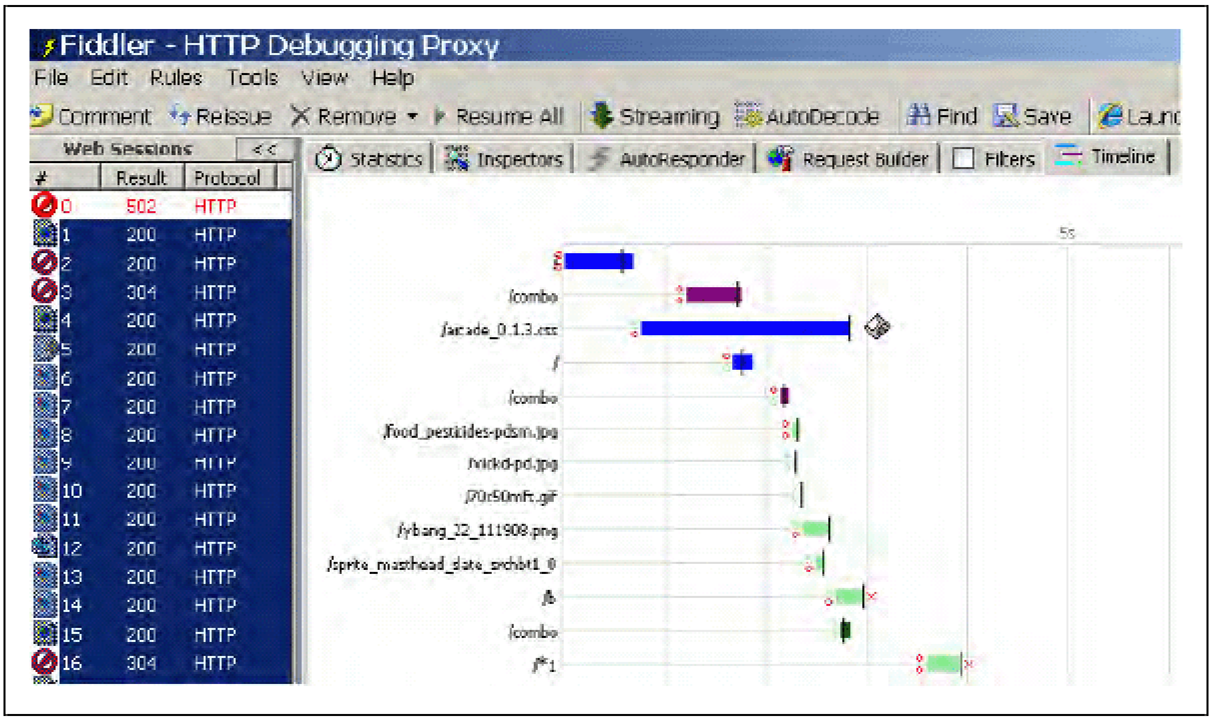
Fiddler 是一个 HTTP 调试代理，检查资源在线传输情况，以定位加载瓶颈。它由 Eric Lawrence 创建，是一个 Windows 下通用的网络分析工具，可为任何浏览器或网页请求给出详细报告。其安装和其它信息参见 http://www.fiddler2.com/fiddler2/。

During installation, Fiddler automatically integrates with IE and Firefox. A button is added to the IE toolbar, and an entry is added under Firefox's Tools menu. Fiddler can also be started manually. Any browser or application that makes web requests can be analyzed. While running, all WinINET traffic is routed through Fiddler, allowing it to monitor and analyze the performance of downloaded assets. Some browsers (e.g., Opera and Safari) do not use WinINET, but they will detect the Fiddler proxy automatically, provided that it is running prior to launching the browser. Any program that allows for proxy settings can be manually run through Fiddler by pointing it at the Fiddler proxy (127.0.0.1, port: 8888).

在安装过程中，Fiddler 与 IE 和 Firefox 自动集成。IE 工具栏上将添加一个按钮，Firefox 的工具菜单中将增加一个菜单项。Fiddler 还可以手工启动。任何浏览器或应用程序发起的网页请求都能够分析。它运行时，所有 WinINET 通信都通过 Fiddler 进行路由，允许它监视并分析资源下载的性能。某些浏览器（例如 Opera 和 Safari）不使用 WinINET，但它们会自动检测 Fiddler 代理，倘若它在浏览器启动之前正在运行的话。任何能够设置代理的程序都可以手工设置指定它使用 Fiddler 代理（127.0.0.1，端口：8888）。

Like Firebug, Web Inspector, and Page Speed, Fiddler provides a waterfall diagram that provides insights as to which assets are taking longer to load and which assets might be affecting the loading of other assets (Figure 10-15).

像 Firebug，网页检查器，Page Speed 一样，Fiddler 提供一个瀑布图，可深入察看哪些资源占用了较长加载时间，哪些资源可能影响了其它资源加载（图 10-15）。


Fiddler - HTTP Debugging Proxy
File Edit Rules Tools View Help
Comment Reissue Remove Resume All Streaming AutoDecode Find Save Launc
Web Sessions
Statistics Inspectors AutoResponder Request Builder Filters Timeline
5s
/combo
/arcade_0.1.3.css
/
/combo
/food_pesticides-pdsm.jpg
/nickd-pd.jpg
/70c50mft.gif
/ybang_32_111908.png
/sprite_masthead_date_srchbt1_0
/b
/combo
/*1

| # | Result | Protocol |
|---|---|---|
| 0 | 502 | HTTP |
| 1 | 200 | HTTP |
| 2 | 200 | HTTP |
| 3 | 304 | HTTP |
| 4 | 200 | HTTP |
| 5 | 200 | HTTP |
| 6 | 200 | HTTP |
| 7 | 200 | HTTP |
| 8 | 200 | HTTP |
| 9 | 200 | HTTP |
| 10 | 200 | HTTP |
| 11 | 200 | HTTP |
| 12 | 200 | HTTP |
| 13 | 200 | HTTP |
| 14 | 200 | HTTP |
| 15 | 200 | HTTP |
| 16 | 304 | HTTP |

Figure 10-15. Fiddler waterfall diagram

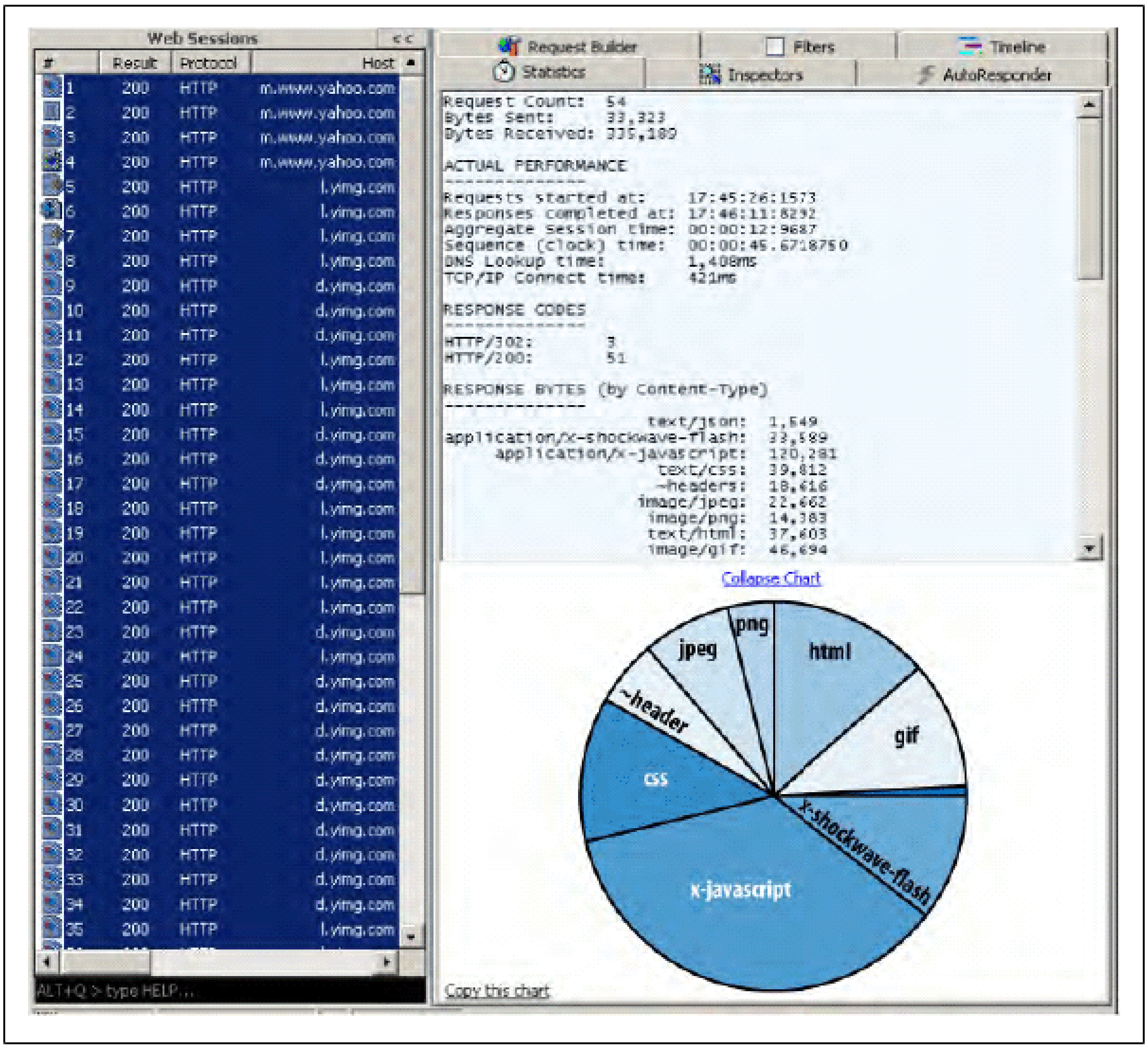
图 10-15 Fiddler 的瀑布图

Selecting one or more resources from the panel on the left shows them in the main view. Click the Timeline tab to visualize the assets over the wire. This view provides the timing of each asset relative to other assets, which allows you to study the effects of different loading strategies and makes it more obvious when something is blocking.

在主视窗的左侧面板中选择一个或多个资源。点击时间线标签可以看到通过网络的资源。此视图提供了每个相关联资源之间的计时信息，使你可以研究不同加载策略的效果，以及使阻塞的原因更加明显。

The Statistics tab shows a detailed view of the actual performance of all selected assets—giving insight into DNS Lookup and TCP/IP Connect times—as well as a breakout of the size of and type of the various assets being requested (Figure 10-16).

统计标签显示了所有选择资源实际性能的细节视图——包括 DNS 解析和 TCP/IP 连接的时间——以及所请求的各种资源的大小、类型等详细信息（图 10-16）。



| # | Result | Protocol | Host |
|---|---|---|---|
| 1 | 200 | HTTP | m.www.yahoo.com |
| 2 | 200 | HTTP | m.www.yahoo.com |
| 3 | 200 | HTTP | m.www.yahoo.com |
| 4 | 200 | HTTP | m.www.yahoo.com |
| 5 | 200 | HTTP | l.yimg.com |
| 6 | 200 | HTTP | l.yimg.com |
| 7 | 200 | HTTP | l.yimg.com |
| 8 | 200 | HTTP | l.yimg.com |
| 9 | 200 | HTTP | d.yimg.com |
| 10 | 200 | HTTP | d.yimg.com |
| 11 | 200 | HTTP | d.yimg.com |
| 12 | 200 | HTTP | l.yimg.com |
| 13 | 200 | HTTP | l.yimg.com |
| 14 | 200 | HTTP | l.yimg.com |
| 15 | 200 | HTTP | d.yimg.com |
| 16 | 200 | HTTP | d.yimg.com |
| 17 | 200 | HTTP | d.yimg.com |
| 18 | 200 | HTTP | l.yimg.com |
| 19 | 200 | HTTP | l.yimg.com |
| 20 | 200 | HTTP | l.yimg.com |
| 21 | 200 | HTTP | l.yimg.com |
| 22 | 200 | HTTP | l.yimg.com |
| 23 | 200 | HTTP | d.yimg.com |
| 24 | 200 | HTTP | l.yimg.com |
| 25 | 200 | HTTP | d.yimg.com |
| 26 | 200 | HTTP | d.yimg.com |
| 27 | 200 | HTTP | d.yimg.com |
| 28 | 200 | HTTP | d.yimg.com |
| 29 | 200 | HTTP | d.yimg.com |
| 30 | 200 | HTTP | d.yimg.com |
| 31 | 200 | HTTP | d.yimg.com |
| 32 | 200 | HTTP | d.yimg.com |
| 33 | 200 | HTTP | d.yimg.com |
| 34 | 200 | HTTP | d.yimg.com |
| 35 | 200 | HTTP | l.yimg.com |

```
Request Count:   54
Bytes Sent:      33,323
Bytes Received:  335,189

ACTUAL PERFORMANCE
--------------
Requests started at:     17:45:26:1573
Responses completed at:  17:46:11:8292
Aggregate Session time:  00:00:12:9687
Sequence (clock) time:   00:00:45.6718750
DNS Lookup time:         1,408ms
TCP/IP Connect time:     421ms

RESPONSE CODES
--------------
HTTP/302:        3
HTTP/200:        51

RESPONSE BYTES (by Content-Type)
--------------
                    text/json:  1,649
application/x-shockwave-flash:  33,589
     application/x-javascript:  120,281
                     text/css:  39,812
                    ~headers:  18,616
                  image/jpeg:  22,662
                   image/png:  14,383
                   text/html:  37,603
                   image/gif:  46,694
```

Figure 10-16. Fiddler Statistics tab

图 10-16 Fiddler 的统计图表

This data helps you decide which areas should be investigated further. For example, long DNS Lookup and TCP/IP Connect times may indicate a problem with the network. The resource chart makes it obvious which types of assets comprise the bulk of the page load, identifying possible candidates for deferred loading or profiling (in the case of scripts).

这些数据帮助你决策哪些地方应当进行更深入的调查。例如，DNS 解析和 TCP/IP 连接时间过长可能意味着网络问题。资源图表中可以明显地看出哪种类型的资源在页面加载中比例较大，找出哪些可能需要延迟加载，或者需要进一步分析（如果是脚本类型）。

**YSlow**

The YSlow tool provides performance insights into the overall loading and execution of the initial page view. This tool was originally developed internally at Yahoo! by Steve Souders as a Firefox addon (via GreaseMonkey). It has been made available to the public as a Firebug addon, and is maintained and updated regularly by Yahoo! developers. Visit http://developer.yahoo.com/yslow/ for installation instructions and other product details.

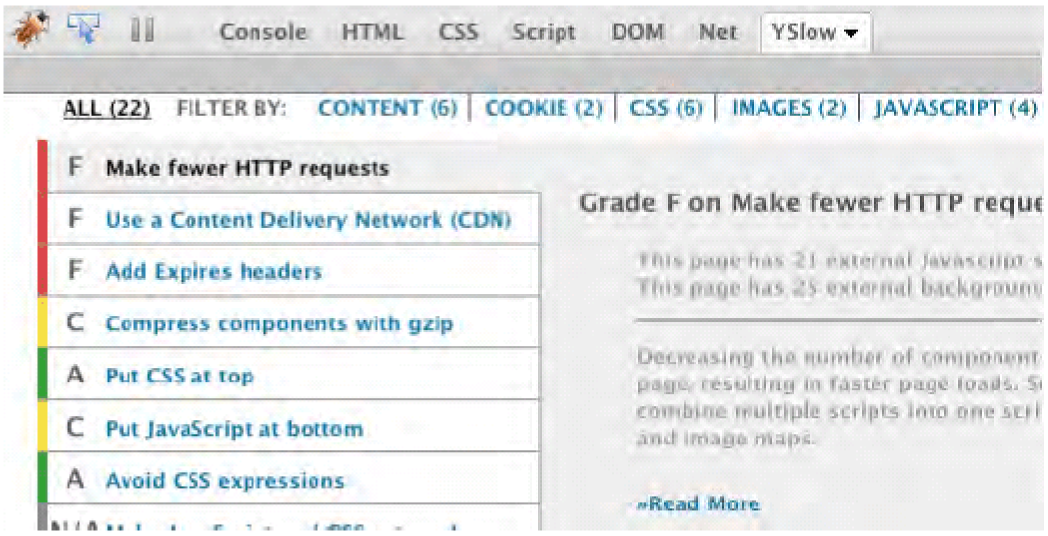
YSlow 工具能够深入视察初始页面视图整体加载和运行过程的性能。它最初由 Yahoo!内部的 Steve Souders 开发，作为 Firefox 插件（通过 GreaseMonkey）。它已经发布为一个 Firebug 插件，由 Yahoo!开发人员维护并定期更新。安装及其他产品细节参见 http://developer.yahoo.com/yslow/。

YSlow scores the loading of external assets to the page, provides a report on page performance, and gives tips for improving loading speed. The scoring it provides is based on extensive research done by performance experts. Applying this feedback and reading more about the details behind the scoring helps ensure the fastest possible page load experience with the minimal number of resources.

YSlow 为页面加载的外部资源评分，为页面性能提供报告，并给出提高加载速度的建议。它提供的评分基于性能专家们广泛的研究。运用这些反馈信息，并阅读评分背后更多的细节，有助于以最小的资源数量确保最快的页面加载体验。

Figure 10-17 shows YSlow's default view of a web page that has been analyzed. It will make suggestions for optimizing the loading and rendering speed of the page. Each of the scores includes a detailed view with additional information and an explanation of the rule's rationale.

图 10-17 显示了 YSlow 默认的网页分析视图。它将提供关于优化下载和页面渲染速度的建议。每一个评分都包含一个细节视图提供附加信息，以及对规则理由的解释。


Console HTML CSS Script DOM Net YSlow

ALL (22) FILTER BY: CONTENT (6) | COOKIE (2) | CSS (6) | IMAGES (2) | JAVASCRIPT (4)

F Make fewer HTTP requests
F Use a Content Delivery Network (CDN)
F Add Expires headers
C Compress components with gzip
A Put CSS at top
C Put JavaScript at bottom
A Avoid CSS expressions

Grade F on Make fewer HTTP reque

This page has 21 external javascript s
This page has 25 external backgroun

Decreasing the number of component
page, resulting in faster page loads. S
combine multiple scripts into one scri
and image maps.

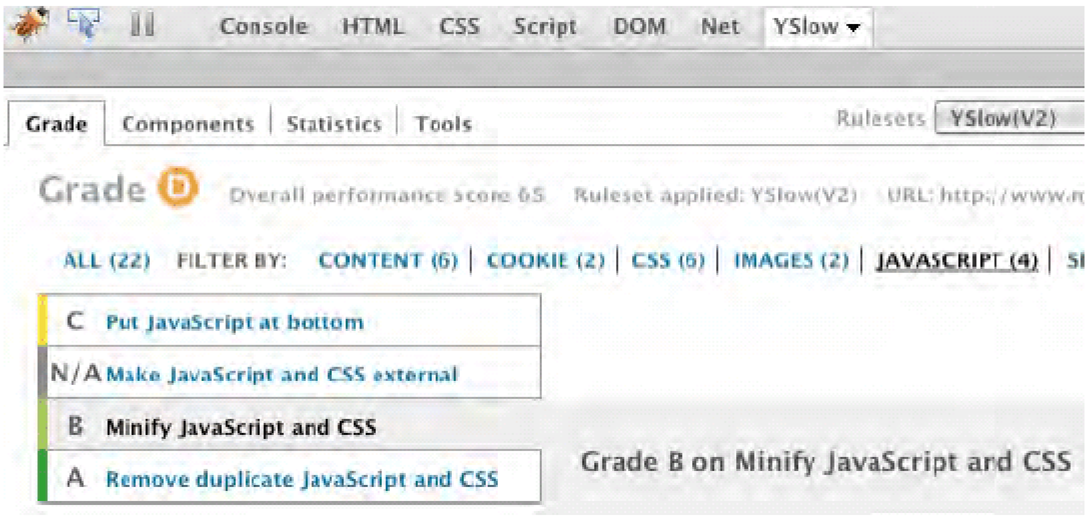
»Read More

Figure 10-17. YSlow: All results

图 10-17 YSlow：全部结果

In general, improving the overall score will result in faster loading and execution of scripts. Figure 10-18 shows the results filtered by the JAVASCRIPT option, with some advice about how to optimize script delivery and execution.

一般情况下，提高整体评分将意味着更快的加载和脚本运行。图 10-18 显示出由 JAVASCRIPT 选项过滤后的结果，还有一些建议，关于如何优化脚本的传输和运行。


Console HTML CSS Script DOM Net YSlow

Grade Components Statistics Tools Rulesets YSlow(V2)

Grade D Overall performance score 65 Ruleset applied: YSlow(V2) URL: http://www.n

ALL (22) FILTER BY: CONTENT (6) | COOKIE (2) | CSS (6) | IMAGES (2) | JAVASCRIPT (4) | SI

C Put JavaScript at bottom
N/A Make JavaScript and CSS external
B Minify JavaScript and CSS
A Remove duplicate JavaScript and CSS

Grade B on Minify JavaScript and CSS

Figure 10-18. YSlow: JavaScript results

图 10-18 YSlow：JavaScript 结果

When interpreting the results, keep in mind that there may be exceptions to consider. These might include deciding when to make multiple requests for scripts versus combining into a single request, and which scripts or functions to defer until after the page renders.

在分析结果时，请记住要考虑到一些意外情况。包括决定是否将多个脚本请求合并成一个单独请求，以及哪些脚本或函数应当在页面渲染之后延迟加载。

**dynaTrace Ajax Edition Ajax 版的 dynaTrace**

The developers of dynaTrace, a robust Java/.NET performance diagnostic tool, have released an "Ajax Edition" that measures Internet Explorer performance (a Firefox version is coming soon). This free tool provides an end-to-end performance analysis, from network and page rendering to runtime scripts and CPU usage. The reports display all aspects together, so you can easily find where any bottlenecks may be occurring. The results can be exported for further dissection and analysis. To download, visit http://ajax.dynatrace.com/pages/.

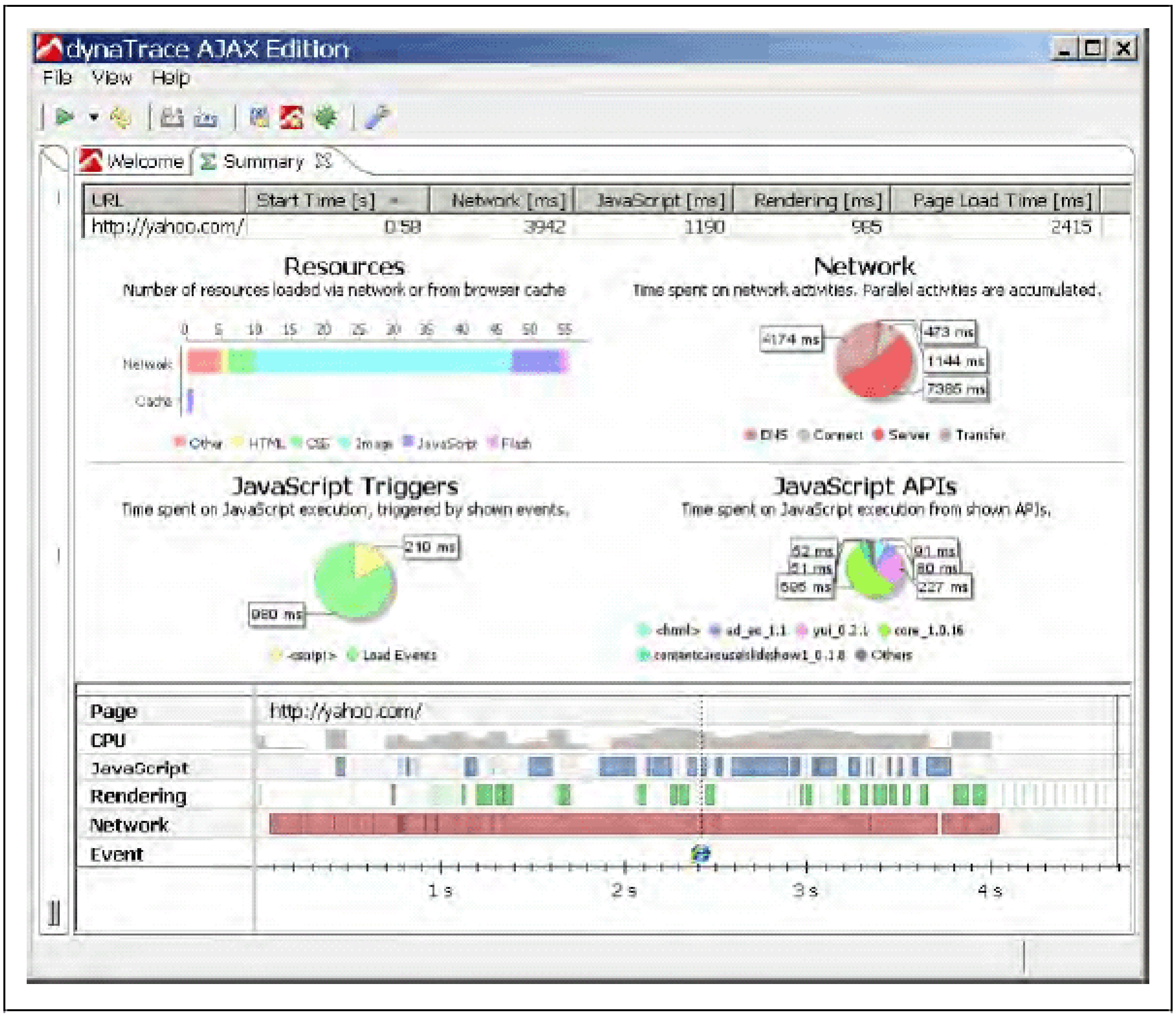
dynaTrace 是一个强大的 Java/.NET 性能诊断工具，它的开发人员已经发布了一个"Ajax 版"用于测量 Internet Explorer 的性能（Firefox 版很快就会出现）。这个免费工具提供了一个"终端到终端"性能分析器，从网络和页面渲染，到脚本运行时间和 CPU 占用率都能分析。报告显示将所有信息汇总在一起，所以你可以容易地发现瓶颈之所在。结果可导出用于进一步剖析。它可在这里下载：http://ajax.dynatrace.com/pages/。

The Summary report shown in Figure 10-19 provides a visual overview that allows you to quickly determine which area or areas need more attention. From here you can drill down into the various narrower reports for more granular detail regarding that particular aspect of performance.

总结报告如图 10-19 所示，提供了一个图形化的概貌，使您马上知道哪些区域需要更多关注。从这里您可以深入到各种具体的报告中，察看某一方面性能的更多细节。

The Network view, shown in Figure 10-20, provides a highly detailed report that breaks out time spent in each aspect of the network life cycle, including DNS lookup, connection, and server response times. This guides you to the specific areas of the network that might require some tuning. The panels below the report show the request and response headers (on the left) and the actual request response (on the right).

网络视图如图 10-20 所示，提供了关于网络生命周期每个阶段花费时间的非常详细的报告，包括 DNS 解析，连接，和服务器响应时间。它指引你进入网络中可能需要调整的特定区域。下面面板中的报告显示了请求和响应报文头（左侧）和实际请求的响应（右侧）。



| URL | Start Time [s] | Network [ms] | JavaScript [ms] | Rendering [ms] | Page Load Time [ms] |
|---|---|---|---|---|---|
| http://yahoo.com/ | 0.58 | 3942 | 1190 | 585 | 2415 |

Resources: Number of resources loaded via network or from browser cache

Network: Time spent on network activities. Parallel activities are accumulated.

JavaScript Triggers: Time spent on JavaScript execution, triggered by shown events.

JavaScript APIs: Time spent on JavaScript execution from shown APIs.

Figure 10-19. dynaTrace Ajax Edition: Summary report

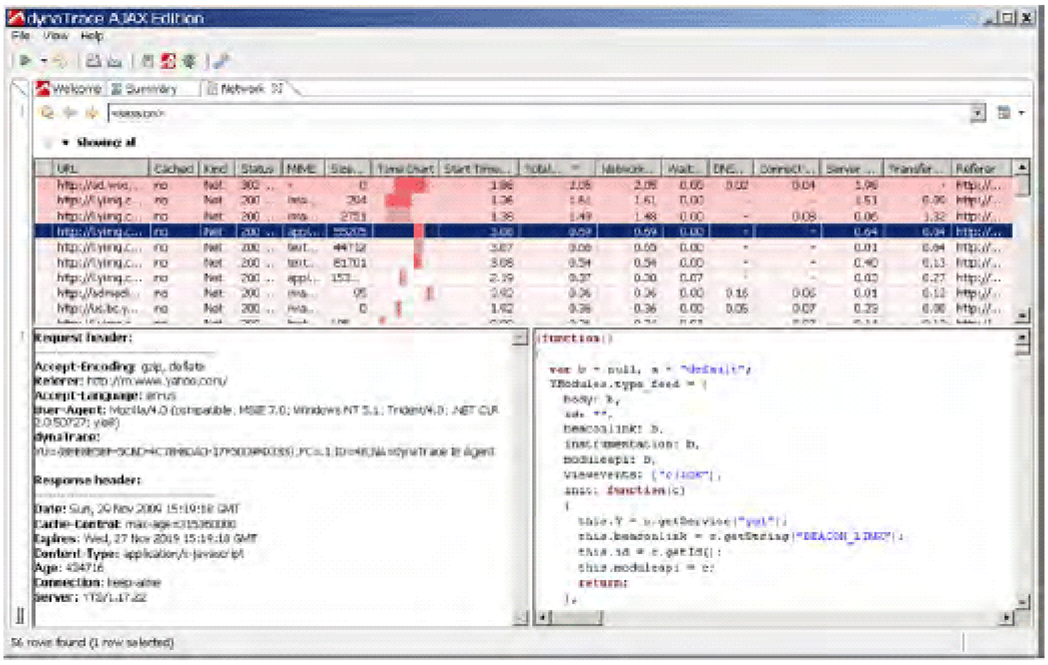
图 10-19 dynaTrace Ajax Edition：总结报告

Selecting the JavaScript Triggers view brings up a detailed report on each event that fired during the trace (see Figure 10-21). From here you can drill into specific events ("load", "click", "mouseover", etc.) to find the root cause of runtime performance issues.

选择 JavaScript 触发器视图将看到跟踪过程中所发出的每个事件的详细报告（如图 10-21）。从这里你可以深入到每个特定的事件中（“load”，“click”，“mouseover”等等）去发现运行时性能问题的根本原因。

This view includes any dynamic (Ajax) requests that a event may be triggering and any script "callback" that may be executed as a result of the request. This allows you to better understand the overall performance perceived by your users, which, because of the asynchronous nature of Ajax, might not be obvious in a script profile report.

此视图包括一个事件可能触发的任意动态（Ajax）请求，以及作为请求结果而运行的任意脚本“回调”。这使您更好地理解用户所体会到的整体性能，由于 Ajax 的异步特性，在一个脚本分析报告中可能不怎么明显。



| URL | Cached | Kind | Status | MIME | Size... | Time Chart | Start Time... | Total... | Network... | Wait... | DNS... | Connect... | Server... | Transfer... | Referer |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| http://ad.wsq... | no | Net | 302 ... | - | 0 | | 1.86 | 2.05 | 2.05 | 0.05 | 0.02 | 0.04 | 1.96 | - | http://... |
| http://l.yimg.c... | no | Net | 200 ... | ima... | 204 | | 1.36 | 1.61 | 1.61 | 0.00 | | | 1.53 | 0.00 | http://... |
| http://l.yimg.c... | no | Net | 200 ... | ima... | 2751 | | 1.38 | 1.49 | 1.48 | 0.00 | - | 0.08 | 0.06 | 1.32 | http://... |
| http://l.yimg.c... | no | Net | 200 ... | appl... | 55205 | | 3.00 | 0.59 | 0.59 | 0.00 | - | - | 0.64 | 0.04 | http://... |
| http://l.yimg.c... | no | Net | 200 ... | text... | 44712 | | 3.07 | 0.05 | 0.05 | 0.00 | - | - | 0.01 | 0.04 | http://... |
| http://l.yimg.c... | no | Net | 200 ... | text... | 81701 | | 3.08 | 0.54 | 0.54 | 0.00 | - | - | 0.40 | 0.13 | http://... |
| http://l.yimg.c... | no | Net | 200 ... | appl... | 153... | | 2.39 | 0.37 | 0.30 | 0.07 | - | - | 0.03 | 0.27 | http://... |
| http://sdmspd... | no | Net | 200 ... | ima... | 95 | | 3.92 | 0.36 | 0.36 | 0.00 | 0.16 | 0.06 | 0.01 | 0.12 | http://... |
| http://us.bc.y... | no | Net | 200 ... | ima... | 0 | | 1.92 | 0.36 | 0.36 | 0.00 | 0.05 | 0.07 | 0.23 | 0.06 | http://... |

**Request header:**

**Accept-Encoding:** gzip, deflate
**Referer:** http://m.www.yahoo.com/
**Accept-Language:** en-us
**User-Agent:** Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; yie8)
**dynaTrace:** [illegible] dynaTrace IE Agent

**Response header:**

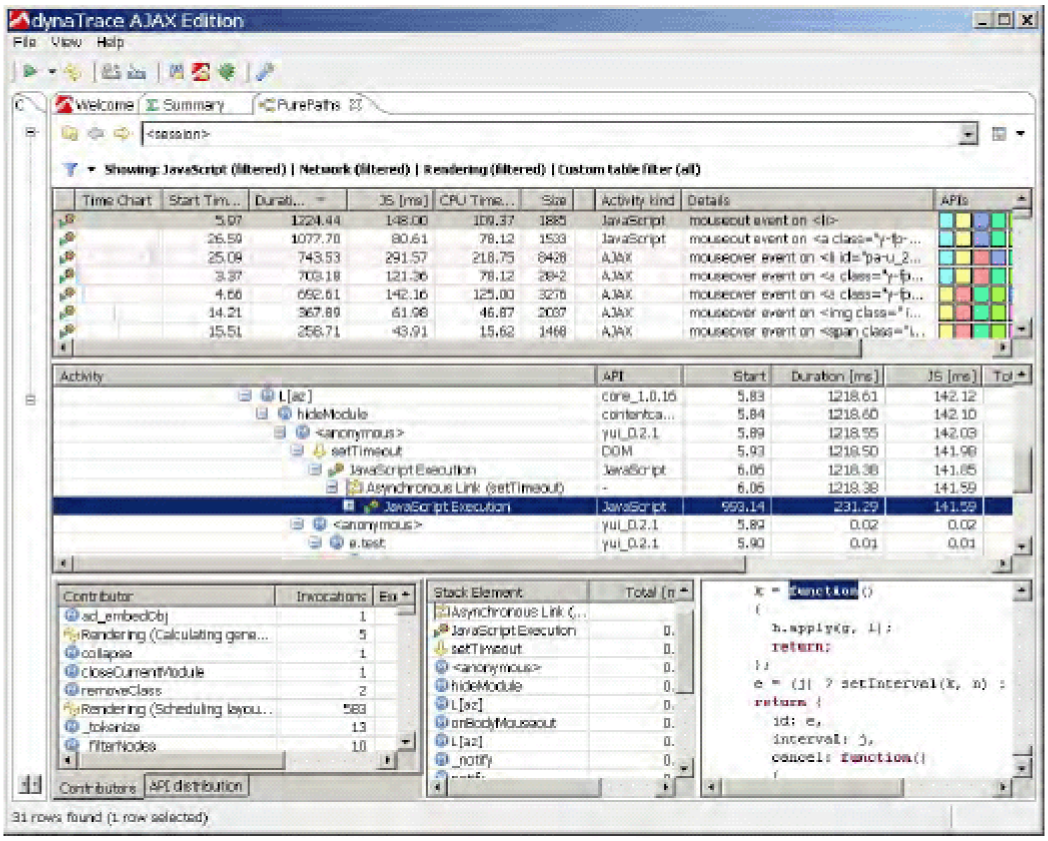
**Date:** Sun, 29 Nov 2009 15:19:18 GMT
**Cache-Control:** max-age=315360000
**Expires:** Wed, 27 Nov 2019 15:19:18 GMT
**Content-Type:** application/x-javascript
**Age:** 434716
**Connection:** keep-alive
**Server:** YTS/1.17.22

```
(function()
{
  var b = null, a = "default";
  YModules.type_feed = {
    body: b,
    id: "",
    beaconlink: b,
    instrumentation: b,
    moduleapi: b,
    viewevents: ["click"],
    init: function(c)
    {
      this.Y = c.getService("yui");
      this.beaconlink = c.getString("BEACON_LINK");
      this.id = c.getId();
      this.moduleapi = c;
      return;
    },
```

56 rows found (1 row selected)

Figure 10-20. dynaTrace Ajax Edition: Network report

图 10-20 dynaTrace Ajax Edition：网络报告



Showing: JavaScript (filtered) | Network (filtered) | Rendering (filtered) | Custom table filter (all)

| Time Chart | Start Tim... | Durati... | JS [ms] | CPU Time... | Size | Activity kind | Details |
|---|---|---|---|---|---|---|---|
| | 5.97 | 1224.44 | 148.00 | 109.37 | 1885 | JavaScript | mouseout event on <li> |
| | 26.59 | 1077.70 | 80.61 | 78.12 | 1533 | JavaScript | mouseout event on <a class="y-fp-... |
| | 25.09 | 743.53 | 291.57 | 218.75 | 8428 | AJAX | mouseover event on <li id="pa-u_2... |
| | 3.37 | 703.18 | 121.36 | 78.12 | 2842 | AJAX | mouseover event on <a class="y-fp... |
| | 4.66 | 692.61 | 142.16 | 125.00 | 3276 | AJAX | mouseover event on <a class="y-fp... |
| | 14.21 | 367.89 | 61.98 | 46.87 | 2037 | AJAX | mouseover event on <img class="i... |
| | 15.51 | 258.71 | 43.91 | 15.62 | 1468 | AJAX | mouseover event on <span class="i... |

| Activity | API | Start | Duration [ms] | JS [ms] |
|---|---|---|---|---|
| L[az] | core_1.0.16 | 5.83 | 1218.61 | 142.12 |
| hideModule | contentca... | 5.84 | 1218.60 | 142.10 |
| <anonymous> | yui_0.2.1 | 5.89 | 1218.55 | 142.03 |
| setTimeout | DOM | 5.93 | 1218.50 | 141.98 |
| JavaScript Execution | JavaScript | 6.06 | 1218.38 | 141.85 |
| Asynchronous Link (setTimeout) | - | 6.06 | 1218.38 | 141.59 |
| JavaScript Execution | JavaScript | 993.14 | 231.29 | 141.59 |
| <anonymous> | yui_0.2.1 | 5.89 | 0.02 | 0.02 |
| e.test | yui_0.2.1 | 5.90 | 0.01 | 0.01 |

| Contributor | Invocations |
|---|---|
| ad_embedObj | 1 |
| Rendering (Calculating gene... | 5 |
| collapse | 1 |
| closeCurrentModule | 1 |
| removeClass | 2 |
| Rendering (Scheduling layou... | 583 |
| _tokenize | 13 |
| filterNodes | 10 |

| Stack Element | Total (n |
|---|---|
| Asynchronous Link (... | |
| JavaScript Execution | 0. |
| setTimeout | 0. |
| <anonymous> | 0. |
| hideModule | 0. |
| L[az] | 0. |
| onBodyMouseout | 0. |
| L[az] | 0. |
| _notify | 0. |

```
x = function()
{
  h.apply(g, i);
  return;
};
e = (j) ? setInterval(x, n) :
return {
  id: e,
  interval: j,
  cancel: function()
```

31 rows found (1 row selected)

Figure 10-21. dynaTrace Ajax Edition PurePaths panel

图 10-21 Ajax 版 dynaTrace 的 PurePaths 面板

## Summary 总结

When web pages or applications begin to feel slow, analyzing assets as they come over the wire and profiling scripts while they are running allows you to focus your optimization efforts where they are needed most.

当网页或应用程序变慢时，分析网上传来的资源，分析脚本的运行性能，使你能够集中精力在那些需要努力优化的地方。

• Use a network analyzer to identify bottlenecks in the loading of scripts and other page assets; this helps determine where script deferral or profiling may be needed.

使用网络分析器找出加载脚本和其它页面资源的瓶颈所在，这有助于决定哪些脚本需要延迟加载，或者进行进一步分析。

• Although conventional wisdom says to minimize the number of HTTP requests, deferring scripts whenever possible allows the page to render more quickly, providing users with a better overall experience.

传统的智慧告诉我们应尽量减少 HTTP 请求的数量，尽量延迟加载脚本以使页面渲染速度更快，向用户提供更好的整体体验。

• Use a profiler to identify slow areas in script execution, examining the time spent in each function, the number of times a function is called, and the callstack itself provides a number of clues as to where optimization efforts should be focused.

使用性能分析器找出脚本运行时速度慢的部分，检查每个函数所花费的时间，以及函数被调用的次数，通过调用栈自身提供的一些线索来找出哪些地方应当努力优化。

• Although time spent and number of calls are usually the most valuable bits of data, looking more closely at how functions are being called might yield other optimization candidates.

虽然花费时间和调用次数通常是数据中最有价值的点，还是应当仔细察看函数的调用过程，可能发现其它优化方法。

These tools help to demystify the generally hostile programming environments that modern code must run in. Using them prior to beginning optimization will ensure that development time is spent focusing on the right problems.

这些工具在那些现代代码所要运行的编程环境中不再神秘。在开始优化工作之前使用它们，确保开发时间用在解决问题的刀刃上。

（全文终）