**第8章 BOM**

本章内容

* 理解window对象---BOM核心
* 控制窗口、框架和弹出窗口
* 利用location对象中的页面信息
* 使用navigator对象了解浏览器

ECMAScript是JavaScript的核心，但如果要在Web中使用JavaScript，那么BOM（浏览器对象模型）怎无疑才是真正的核心。BOM提供了很多对象，用于访问浏览器的功能，这些功能与任何网页内容无关。多年来，缺少事实上的规范导致BOM既有意思又有问题，因为浏览器提供的商会按照各自的想法随意去扩展它。于是，浏览器之间共有的对象就成为了事实上的标准。这些对象在浏览器中得以存在，很大程度上是由于它们提供了与浏览器的互操作性。W3C为了把浏览器中的JavaScript最基本的部分标准化，已经将BOM的主要方面纳入了HTML5的规范中。

8.1 window对象

BOM的核心对象是window，它表示浏览器的一个实例。在浏览器中，window对象有双重角色，它既是JavaScript访问浏览器窗口的一个接口，又是ECMAScript规定的Global对象。这意味在网页中定义的任何一个对象、变量和函数，都以window作用Global对象，因此有权访问parseInt()等方法。

8.1.1 全局作用域

由于window对象同时扮演着ECMAScript中Global对象的角色，因此所有在全局作用域中声明的变量、函数都会变成window对象的属性和方法。来看下面的例子。

1.       var age = 33;

2.       function sayAge() {

3.         alert(this.age);

4.       }

5.

6.       alert(window.age);    // 33

7.       sayAge();             // 33

8.       window.sayAge();      // 33

9.

我们在全局作用域中定义了一个变量age和一个函数sayAge()，它们被自动归在了window对象名下。于是，可以通过window.age访问age，可以通过window.sayAge()访问函数sayAge()。 由于sayAge()存在于全局作用域中，因此this.age被映射到window.age，最终显示仍然是正确的结果。

抛开全局变量会成为window对象的属性不谈，定义全局变量与在window对象上直接定义属性还是有一点差别：全局变量不能通过delete操作符删除，而直接window对象上定义的属性可以。

例如：

1.       var age = 29;

2.       window.color = "red";

3.

4.       // 在IE < 9时抛出错误，在其它所有浏览器中都返回false

5.       delete window.age;

6.

7.       // 在IE < 9时抛出错误，在其它所有浏览器中都返回true

8.       delete window.color;

9.

10.       alert(window.age); // 29

11.       alert(window.color); // undefined

12.

DeleteOperatorExample01.html

刚才使用var语句添加的window属性有一个名为[[Configurable]]的特性，这个特性的值被设置为false，因此这样定义的属性不可以通过delete操作符删除。IE8及更早的版本在遇到使用delete删除window属性的语句时，不管该属性最初是如何创建的，都会抛出错误，以示警告。IE9及更高版本不会抛出错误。

另外，还要记住一件事：尝试访问未声明的变量会抛出错误，但是通过查询window对象，可以知道某个可能未声明的变量是否存在。例如：

1.       // 这里抛出错误，因为oldvalue未定义

2.       var newValue = oldValue;

3.

4.       // 这里不会抛出错误，因为这是一次属性查询

5.       // newValue的值是undefined

6.       var newValue = window.oldValue;

7.

本章后面将要讨论的很多全局JavaScript对象（如location和navigator）实际上都是window对象的属性。

Windows Mobile平台的IE浏览器不允许通过window.property = value之类的形式，直接在window对象上创建新的属性和方法。可是，在全局作用域中声明的所有变量和函数，照样会编程window对象的成员。

8.1.2 窗口关系及框架

如果页面中包含框架，则每个框架都拥有自己的window对象，并且保存在frame集合中。在frames集合中，可以通过数值索引（从0开始，从左至右，从上到下）或者框架名称来访问相应的window对象。每个window对象都有一个name属性，其中包含框架的名称。下面是一个包含框架的页面：

1. <html>

2.   <head>

3.     <title>Frameset Example</title>

4.   </head>

5.   <frameset rows="160,\*">

6.     <frame src="./frame.html" name="topFrame">

7.       <frameset cols="50%,50%">

8.         <frame src="./anotherFrame.html" name="lefFrame">

9.         <frame src="./yetAnotherFrame.html" name="rightFrame">

10.       </frameset>

11.     </frameset>

12. </html>

13.

FramesetExample01.html

以上代码创建了一个框架集，其中一个框架居上，两个框架居下。对这个例子而言，可以通过window.frames[0]或者window.frames[“topFrame”]来引用上方的框架。不过，恐怕你最好使用top而非window来引用这些框架（例如，通过top.frames[0]）。

我们知道，top对象始终执行最高（最外）层的框架，也就是浏览器窗口。使用它可以确保在一个框架中正确地访问另一个框架。因为对于在一个框架中编写的任何代码来说，其中的window对象指向的都是那个框架的特定实例，而非最高层的框架。图8-1展示了在最高层窗口中，通过代码来访问前面例子中每个框架的不同方式。

文本

描述已自动生成

图8-1

与top相对的另一个window对象是parent。顾名思义，parent（父）对象始终指向当前框架的直接上层框架。在某些情况下，parent有可能等于top；但在没有框架的情况下，parent一定等于top（此时它们都等于window）。再看下面的例子。

1. <!DOCTYPE html>

2. <html lang="en">

3. <head>

4.   <meta charset="UTF-8">

5.   <meta name="viewport" content="width=device-width, initial-scale=1.0">

6.   <title>Document</title>

7. </head>

8. <frameset rows="100,\*">

9.   <frame src="./frame.html" name="topFrame">

10.   <frameset cols="40%,60%">

11.       <frame src="./anotherFrame.html" name="lefFrame">

12.       <frame src="./anotherFrameset.html" name="rightFrame">

13.   </frameset>

14. </frameset>

15. </html>

16.

frameset1.html

这个框架集中的一个框架包含了另一个框架集，该框架集的代码如下所示。

1. <!DOCTYPE html>

2. <html lang="en">

3. <head>

4.   <meta charset="UTF-8">

5.   <meta name="viewport" content="width=device-width, initial-scale=1.0">

6.   <title>Document</title>

7. </head>

8. <frameset cols="40%,60%">

9.     <frame src="./red.html" name="redFrame">

10.     <frame src="./blue.html" name="blueFrame">

11. </frameset>

12. </html>

13.

anotherframeset.html

浏览器在加载完第一个框架集以后，会继续将第二个框架集加载到rightFrame中。如果代码位于redFrame（或blueFrame）中，那么parent对象指向的是rightFrame。可是，如果代码位于topFrame中，则parent指向的是top，因为topFrame的直接上层框架就是最外层框架。图8-2展示了在将前面例子加载到浏览器之后，不同window对象的值。

图片包含 表格

描述已自动生成

图8-2

注意，除非最高层窗口是通过window.open()打开的（本章后面将会讨论），否则其window对象的name属性不会包含任何值。

与框架有关的最后一个对象self，它始终指向window；实际上，self和window对象可以互换使用。引入self对象的目的只是为了与top和parent对象对应起来，因此它不格外包含其他值。所有这些对象都是window对象的属性，可以通过window.parent、window.top等形式来访问。同时，这也意味着可以将不同层次的window对象连缀起来，例如window.parent.parent.frames[0]。

在使用框架的情况下，浏览器中会存在多个Global对象。在每个框架中定义的全局变量会自动成为框架中window对象的属性。由于每个window对象都包含原生类型的构造函数，因此每个框架都有一套自己的构造函数，这些构造函数一一对应，但并不相等。例如，top.Object并不等于top.frames[0].Object。这个问题会影响到对跨框架传递的对象使用instanceof操作符。

8.1.3 窗口位置

用来确定和修改window对象位置的属性和方法有很多。IE、Safari、Opera和Chrome都提供了screenLeft和screenTop属性，分别用于表示窗口相对于屏幕左边和上边的位置。Firefox则在screenX和screenY属性中提供相同的窗口位置信息，Safari和Chrome也同时支持这两个属性。Opera虽然也支持screenX和screenY属性，但与screenLeft和screenTop属性并不对应，因此建议大家不要在Opera中使用它们。使用下列代码可以跨浏览器取得窗口左边和上边的位置。

1.       var leftPos =

2.         typeof window.screenLeft == "number"

3.           ? window.screenLeft

4.           : window.screenX;

5.

6.       var topPos =

7.         typeof window.screenTop == "number"

8.           ? window.screenTop

9.           : window.screenY;

10.

WindowPositionExample01.html

这个例子运用二元操作符首先确定screenLeft和screenTop属性是否存在，如果是（在IE、Safari、Opera和Chrome），则取得这两个属性的值。如果不存在（Firefox中），则取得screenX和screenY的值。

在使用这些值的过程中，还必须注意一些小问题。在IE、Opera中，screenLeft和screenTop中保存的是从屏幕左边和上边到由window对象表示的页面可见区域的距离。换句话说，如果window对象是最外层对象，而且浏览器窗口紧贴屏幕最上端---即y轴坐标为0，那么screenTop的值就是位于页面可见区域上方的浏览器工具栏的像素高度。但是，在Chrome、Firefox和Safari中，screenY或screenTop中保存的是整个浏览器窗口相对于屏幕的坐标值，即在窗口的y轴坐标为0时返回0。

更让人捉摸不透是，Firefox、Safari和Chrome始终返回页面中每个框架的top.screenX和top.screenY值。即使在页面由于被设置了外边距而发生偏移的情况下，相对于window对象使用screenX和screenY每次也都会返回相同的值。而IE和Opera则会给出框架相对于屏幕边界的精确坐标值。

最终结果，就是无法在跨浏览器的条件下取得窗口左边和上边的精确坐标值。然而，使用moveTo()和moveBy()方法倒是有可能将窗口精确地移动到一个新位置。这两个方法都接受两个参数，其中moveTo()接受的是新位置的x和y坐标值，而moveBy()接受的是在水平和垂直方向上移动的像素数。下面来看几个例子：

1.       // 将窗口移动到屏幕左上角

2.       window.moveTo(0, 0);

3.

4.       // 将窗口向下移动100像素

5.       window.moveBy(0, 100);

6.

7.       // 将窗口移动到（200，300）

8.       window.moveTo(200, 300);

9.

10.       // 将窗口像左移动50像素

11.       window.moveBy(-50, 0);

12.

需要注意的是，这两个方法可能会被浏览器禁用；在Opera和IE7（及更高版本）中默认就是禁用的。另外，这两个方法都不适用于框架，只能对最外层的window对象使用。

8.1.4 窗口大小

跨浏览器确定一个窗口的大小不是一件简单的事。IE9+、Firefox、Safari、Opera和Chrome均为此提供了4个属性：innerWidth、innerHeight、outerWidth和outerHeight。在IE9+、Safari和Firefox中，outerWidth和outerHeight返回浏览器窗口本身的尺寸（无论是从最外层的window对象还是从某个框架访问）。在Opera中，这两个属性的值表示页面视图容器的大小。而innerWidth和innerHeight则表示该容器中页面视图区的大小（减去边框宽度）。在Chrome中，outerWidth，outerHeight与innerWidth，innerHeight返回相同的值，即视口（viewport）大小而非浏览器窗口大小。

IE8及更早版本没有提供取得当前浏览器窗口尺寸的属性：不过，它通过DOM提供了页面可见区域的相关信息。

在IE9+、Firefox、Safari、Opera和Chrome中，document.documentElement.clientWidth和document.documentElement.clientHeight中保存了页面视口信息。在IE6中，这些属性必须在标准模式下才有效；如果是混杂模式，就必须通过document.body.clientWidth和document.body.clientHeight取得形同信息。而对于混杂模式下的Chrome，则无论通过document.documentElement还是document.body中的clientWidth和clientHeight属性，都可以取得视口的大小。

虽然最终无法确定浏览器窗口本身的大小，但却可以取得页面视口的大小，如下所示。

1.       var pageWidth = window.innerWidth, pageHeight = window.innerHeight;

2.

3.       if (typeof pageWidth != "number") {

4.         if (document.compatMode == "CSS1Compat") {

5.           pageWidth = document.documentElement.clientWidth;

6.           pageHeight = document.documentElement.clientHeight;

7.         } else {

8.           pageWidth = document.body.clientWidth;

9.           pageHeight = document.body.clientHeight;

10.         }

11.       }

12.

13.       console.log(pageWidth, pageHeight);

14.

WindowSizeExample01.html

在以上代码中，我们首先将window.innerWidth和window.innerHeight的值分别赋给了pageWidth和pageHeight。然后检查pageWidth中保存的是不是一个数值；如果不是，则通过检查document.compatMode（这个属性将在第10章全面讨论）来确定页面是否处于标准模式。如果是，则分别使用document.documentElement.clientWidth和document.documentElement.clientHeight的值。否则，就是用document.body.clientWidth和document.body.clientHeight的值。

对于移动设备，window.innerWidth和window.innerHeight保存着可见视口，也就是屏幕上可见页面区域的大小。移动IE浏览器不支持这些属性，但通过document.documentElement.clientWidth和document.documentElement.clientHeight提供了相同的信息。随着页面的缩放，这些值也会相应变化。

在其他移动浏览器中，document.documentElement度量的是布局视口，即渲染后页面的实际大小（与可见视口不同，可见视口只是整个页面中的一小部分）。移动IE浏览器把布局视口的信息保存在document.body.clientWidth和document.body.clientHeight中。这些值不会随着页面缩放变化。

由于与桌面浏览器间存在这些差异，最好是先检查一下用于是否在使用移动设备，然后再决定使用哪个属性。

有关移动设备视口的话题比较复杂，有很多非常规的情形，也有各种各样的建议。移动开发咨询师Peter-Paul Koch记述了他对这个问题的研究：<http://t.cn/zOZs0Tz>。如果你在做移动web开发，推荐你读一下这篇文章。

另外，使用resizeTo()和resizeBy()方法可以调整浏览器窗口的大小。这两个方法都接受两个参数，其中resizeTo()接收浏览器窗口的新宽度和新高度，而resizeBy()接收新窗口与原窗口的宽度和高度之差。来看下面的例子。

1. // 调整到100x100

2. window.resizeTo(100, 100);

3.

4. // 调整到200x150

5. window.resizeBy(100, 50);

6.

7. // 调整到300x300

8. window.resizeTo(300, 300);

9.

需要注意的是，这两个方法与移动窗口的方法类似，也有可能被浏览器禁用；而且，在Opera和IE7（及更高版本）中默认就是禁用的。另外，这两个方法同样不适用于框架，而且只能对最外层的window对象使用。

8.1.5 导航和打开窗口

使用window.open()方法既可以导航到一个特定的URL，也可以打开一个新的浏览器窗口。这个方法可以接收4个参数：要加载的URL、窗口目标、一个特性字符串以及一个表示新页面是否取代浏览器历史记录中当前加载页面的布尔值。通常只须传递第一个参数，最后一个参数只在打开新窗口的情况下使用。

如果为window.open()传递第二个参数，而且该参数是已有窗口或框架的名称，那么就会在具有该名称的窗口或框架中加载第一个参数指定的URL。看下面的例子。

1.       // 等同于 <a href="http://www.baidu.com" target="topFrame"></a>

2.       window.open("http://www.wrox.com/", "topFrame");

3.

调用这行代码，就如同用户点击了href属性为http://www.wrox.com，target属性为“topFrame”的链接。如果有一个名叫“topFrame”的窗口或者框架，就会在该窗口或框架加载这个URL；否则，就会创建一个新窗口并将其命名为“topFrame”。此外，第二个参数也可以是下列任何一个特殊的窗口名称；\_self、\_parent、\_top和\_blank。

1. **弹出窗口**

如果给window.open()传递的第二个参数并不是一个已经存在的窗口或框架，那么该方法就会根据在第三个参数位置上传入的字符串创建一个新窗口或新标签页。如果没有传入第三个参数，那么就会打开一个带有全部默认设置（工具栏、地址栏和状态栏等）的新浏览器窗口（或者打开一个新标签页-根据浏览器设置）。在不打开新窗口的情况下，会忽略第三个参数。

第三个参数是一个逗号分隔的设置字符串，表示在新窗口中都显示哪些特性。下表列出了可以出现在这个字符串中的设置选项。

|  |  |  |
| --- | --- | --- |
| 设置 | 值 | 说明 |
| fullscreen | yes或no | 表示浏览器窗口是否最大化。仅限IE |
| height | 数值 | 表示新窗口的高度，不能小于100 |
| left | 数值 | 表示新窗口的左坐标。不能是负值 |
| location | yes或no | 表示是否在新窗口中显示地址栏。不同浏览器的默认值不同。如果设置为no，地址栏可能会隐藏，也可能会被禁用（取决于浏览器） |
| menubar | yes或no | 表示是否在浏览器窗口中显示菜单栏。默认值为no |
| resizable | yes或no | 表示是否可以通过拖动浏览器窗口的边框改变其大小。默认值为no |
| scrollbars | yes或no | 表示如果内容在视口中显示不下，是否允许滚动。默认值为no |
| status | yes或no | 表示是否在浏览器窗口中显示状态栏。默认值为no |
| toolbar | yes或no | 表示是否在浏览器窗口中显示工具栏。默认值为no |
| top | 数值 | 表示新窗口的上坐标。不能是负值 |
| width | 数值 | 表示新窗口的宽度。不能小于100 |

表中所列的部分或全部设置选项，都可以通过逗号分隔的名值对列表来指定。其中，名值对以等号表示（注意，整个特性字符串中不允许出现空格），如下面的例子所示。

1.       window.open("http://www.wrox.com/", "wroxWindow", "height=400,width=400,top=10,left=10,resizable=yes");

2.

这行代码会打开一个新的可以调整大小的窗口，窗口初始化大小为400x400像素，并且距屏幕上沿和左边各10个像素。

window.open()方法会返回一个指向新窗口的引用。引用的对象与其他window对象大致相似，但我们可以对其进行更多的控制。例如，这些浏览器在默认情况下可能不允许我们针对主浏览器窗口调整大小或移动位置，但却允许我们针对通过window.open()创建的窗口调整大小或移动位置。通过这个返回的对象，可以像操作其他窗口一样操作新打开的窗口，如下所示。

1.       // 等同于 <a href="http://www.baidu.com" target="topFrame"></a>

2.       var wroxWin = window.open(

3.         "http://www.wrox.com/",

4.         "wroxWindow",

5.         "height=400,width=400,top=10,left=10,resizable=yes"

6.       );

7.

8.       // 移动位置

9.       wroxWin.moveTo(500, 500);

10.

11.       // 调整大小

12.       wroxWin.resizeTo(500, 500);

13.

14.       **wroxWin.close()**;

15.

但是，这个方法仅适用于通过window.open()打开的弹出窗口，对于浏览器的主窗口，如果没有得到用户的允许是不能关闭它的。不过，弹出窗口倒是可以调用top.close()在不经用户允许的情况下关闭自己。弹出窗口关闭之后，窗口的引用仍然还在，但除了像下面这样检测其closed属性之外，已经没有其他用处了。

1. wroxWin.close();

2. alert(wroxWin.closed); // true

3.

新创建的window对象有一个opener属性，其中保存着打开它的原始窗口对象。这个属性只在弹出窗口中的最外层window对象（top）中有定义。而且指向调用window.open()的窗口或框架。例如：

1. var wroxWin = window.open("http://www.wrox.com/", "wroxWindow", "height=400,width=400,top=0,left=0,resizable=yes,location=no");

2. alert(wroxWin.opener == window); // true

3.

虽然弹出窗口中有一个指针指向打开它的原始窗口，但原始窗口中并没有这样的指针指向弹出窗口。窗口并不跟踪记录它们打开的弹出窗口，因此我们只能在必要的时候自己来手动实现跟踪。

有些浏览器（如IE8和Chrome）会在独立的进程中运行每个标签页。当一个标签页打开另一个标签页时，如果两个window对象之间需要彼此通信。那么新标签就不能运行在独立的进程中。在Chrome，将新创建的标签页的opener属性设置为null，即表示在单独的进程中运行新标签页，如下所示。

1. var wroxWin = window.open("http://www.wrox.com/", "wroxWindow", "height=400,width=400,top=0,left=0,resizable=yes,location=no");

2. wroxWin.opener = null;

3.

将opener属性设置为null就是告诉浏览器新创建的标签页不需要与打开它的标签页进行通信，因此可以在独立的进程中运行。标签页之间的联系一旦切断，将没有办法恢复。

1. **安全限制**

曾经有一段时间，广告商在网上使用弹出窗口达到了肆无忌惮的程度。他们经常把弹出窗口打扮成系统对话框的模样，引诱用户去点击其中的广告。由于看起来像是系统对话框，一般用户很难分辨是真是假。为了解决这个问题，有些浏览器开始在弹出窗口配置方面增加限制。

Windows XP SP2对弹出窗口施加了多方面的安全限制，包括不允许在屏幕之外创建弹出窗口、不允许将弹出窗口移动到屏幕以外、不允许关闭状态栏等。IE7则增加了更多的安全限制，如不允许关闭地址栏、默认情况下不允许移动弹出窗口或调整其大小。Firefox 1从一开始就不支持修改状态栏，因此无论给window.open()传入什么样的特性字符串，弹出窗口中都会无一例外地显示状态栏。后来的Firefox 3又强制始终在弹出窗口中显示地址栏。Opera只会在主浏览器窗口中打开弹出窗口，但不允许它们出现在可能与系统对话框混淆的地方。

此外，有的浏览器只根据用户操作来创建弹出窗口。这样一来，在页面尚未加载完成时调用window.open()的语句根本不会执行，而且还可能会将错误消息显示给用户。换句话说，只能通过单击或者击键来打开弹出窗口。

对于那些不是用户有意打开的弹出窗口，Chrome采取了不同的处理方式。它不会像其他浏览器那样简单地屏蔽这些弹出窗口，而是只显示它们的标题栏，并把它们放在浏览器窗口的右下角。

在打开计算机硬盘中的网页时，IE会解除对弹出窗口的某些限制。但是在服务器上执行这些代码会受到对弹出窗口的限制。

1. **弹出窗口的屏蔽程序**

大多数浏览器都内置有弹出窗口屏蔽程序，而没有内置此类程序的浏览器，也可以安装Yahoo! Toolbar等带有内置屏蔽程序的实用工具。结果就是用户可以将绝大多数不想看到的弹出窗口屏蔽掉。于是，在弹出窗口被屏蔽时，就因该考虑两种可能性。如果是浏览器内置的屏蔽程序阻止的弹出窗口，那么window.open()很可能会返回null。此时，只要检查这个返回的值就可以确定弹出窗口是否被屏蔽了，如下面的例子所示。

1.       var wroxWin = window.open("http://www.wrox.com", "\_blank");

2.       if (wroxWin === null) {

3.         alert("The popup was blocked!");

4.       }

5.

如果是浏览器扩展或其他程序阻止的弹出窗口，那么window.open()通常会抛出一个错误。因此，要想准确地检测出弹出窗口是否被屏蔽，必须在检测返回值的同时，将对window.open()的调用封装在一个try-catch块中，如下所示。

1.       var blocked = false;

2.       try {

3.         var wroxWin = window.open("http://www.wrox.com", "\_blank");

4.         if (wroxWin == null) {

5.           blocked = true;

6.         }

7.       } catch (ex) {

8.         blocked = true;

9.       }

10.

11.       if (blocked) {

12.         alert("The popup was blocked!");

13.       }

14.

PopupBlockerExample01.html

在任何情况下，以上代码都可以检测出调用window.open()打开的弹出窗口是不是被屏蔽了。但要注意的是，检测出弹出窗口是否被屏蔽只是一方面，它并不会阻止浏览器显示与被屏蔽的弹出窗口有关的消息。

8.1.6 间歇调用和超时调用

JavaScript是单线程语言，但它允许通过设置超时值和间歇值来调度代码在特定的时刻执行前者是在指定的时间过后执行代码，而后者则是每隔指定的时间就执行一次代码。

超时调用需要使用window对象的setTimeOut()方法，它接受两个参数：要执行的代码和以毫秒表示的时间（即在执行代码前需要等待多少毫秒）。其中，第一个参数可以是一个包含JavaScript代码的字符串（就和在eval()函数中使用的字符串一样），也可以是一个函数。例如，下面对setTimeOut()的两次调用都会在一秒钟后显示一个警告框。

1.       // 不建议传递字符串

2.       setTimeout("alert('Hello World!')", 1000);

3.

4.       // 推荐的调用方式

5.       setTimeout(function () {

6.         alert("Hello World!");

7.       }, 1000);

8.

TimeoutExample01.html

虽然这两种调用方式都没有问题，但由于传递字符串可能导致性能损失，因此不建议以字符串作为第一个参数。

第二个参数是一个表示等待多长时间的毫秒数，但经过该时间后指定的代码不一定会执行。JavaScript是一个单线程的解释器，因此一定时间内只能执行一段代码。为了控制要执行的代码，就有一个JavaScript任务队列。这些任务会按照将它们添加到队列的顺序执行。setTimeOut()的第二个参数告诉JavaScript再过多长时间把当前任务添加到队列中。如果队列是空的，那么添加的代码会立即执行；如果队列不是空的，那么它就要等前面的代码执行完了以后再执行。

调用setTimeOut()之后，该方法会返回一个数值ID，表示超时调用。这个超时调用ID是计划执行代码的唯一标识符，可以通过它来取消超时调用。要取消尚未执行的超时调用计划，可以调用clearTimeOut()方法并将相应的超时调用ID作为参数传递给它，如下所示。

1.       var timeoutId = setTimeout(function () {

2.         alert("Hello world!");

3.       }, 1000);

4.

5.       clearTimeout(timeoutId);

6.

TimeoutExample02.html

只要再指定的时间尚未过去之前调用clearTimeout()，就可以完全取消超时调用。前面的代码在设置超时调用之后马上又调用了clearTimeout()，结果就跟什么也没有发生一样。

超时调用的代码都是在全局作用域中执行的，因此函数中this的值在非严格模式下指向window对象，在严格模式下是undefined。

间歇调用与超时调用类似，只不过他会按照指定的时间间隔重复执行代码，直至间歇调用被取消或者页面被卸载。设置间歇调用的方法是setInterval()，它接受的参数与setTimeout()相同：要执行的代码（字符串或函数）和每次执行之前需要等待的毫秒数。下面来看一个例子。

1.       // 不建议传递字符串

2.       setInterval("alert('Hello World!')", 10000);

3.

4.       // 推荐的调用方式

5.       setInterval(function () {

6.         alert("Hello World!");

7.       }, 10000);

8.

IntervalExample01.html

调用setInterval()方法同样也会返回一个间歇调用ID，该ID可用于在将来某个时刻取消间歇调用。要取消尚未执行的间歇调用，因为在不加干涉的情况下，间歇调用将会一直执行到页面卸载。以下是一个常见的使用间歇调用的例子。

1.       var num = 0;

2.       var max = 10;

3.       var intervalId = null;

4.

5.       function incrementNumber() {

6.         num++;

7.         if (num == max) {

8.           clearInterval(intervalId);

9.           alert("Done");

10.         }

11.       }

12.

13.       intervalId = setInterval(incrementNumber, 500);

14.

IntervalExample02.html

在这个例子中，变量num每半秒钟递增一次，当递增到最大值时，就会取消先前设定的间歇调用。这个模式也可以使用超时调用来实现，如下所示。

1.       var num = 0;

2.       var max = 10;

3.

4.       function incrementNumber() {

5.         num++;

6.         if (num < max) {

7.           setTimeout(incrementNumber, 500);

8.         } else {

9.           alert("Done");

10.         }

11.       }

12.

13.       setTimeout(incrementNumber, 500);

14.

TimeoutExample03.html

可见，在使用超时调用时，没有必要跟踪超时调用ID，因为每次执行代码之后，如果不再设置另一次超时调用，调用就会自行停止。一般认为，使用超时调用来模拟间歇调用的是一种最佳模式。在开发环境下，很少使用真正的间歇调用，原因是后一个间歇调用可能会在前一个间歇调用结束之前启动。而像前面示例中那样使用超时调用，则完全可以避免这一点。所以，最好不要使用间歇调用。

8.1.7 系统对话框

浏览器通过alert()、confirm()和promp()方法调用系统对话框向用户显示消息。系统对话框与在浏览器中显示的网页没有关系，也不包含HTML。它们的外观由操作系统及（或）浏览器设置决定，而不是CSS决定。此外，通过这几个方法打开的对话框都是同步和模态的。也就是说，显示这些对话框的时候代码会停止执行，而关掉这些对话框后代码又会恢复执行。

本书各章经常会用到alert()方法，这个方法接受一个字符串并将其显示给用户。具体来说，调用alert()方法的结果就是向用户显示一个系统对话框，其中包含指定的文本和一个OK（“确定”）按钮。例如，alert(“Hello World!”)会在Windows XP系统的IE中生成如图8-3所示的对话框。

图片包含 背景图案

描述已自动生成

图 8-3

第二种对话框是调用confirm()方法生成的。从向用户显示消息的方面来看，这种“确认”对话框很像是一个“警告”对话框。但二者的主要区别在于“确认”对话框除了显示OK按钮外，还会显示一个Cancel（“取消”）按钮，两个按钮可以让用户决定是否执行给定的操作。例如，confirm(“Are you sure?”)会显示如图8-4所示的确认对话框。

图片包含 图形用户界面

描述已自动生成

图 8-4

为了确定用户是单击了OK还是Cancel，可以检查confirm()方法返回的布尔值：true表示单击了OK，false表示单击了Cancel或单击了右上角的X按钮。确认对话框的典型用法如下。

1.       if (confirm("Are you sure?")) {

2.         alert("I\'m so glad you\'re sure!");

3.       } else {

4.         alert("I\'m sorry to hear you\'re not sure.");

5.       }

6.

在这个例子中，第一行代码（if条件语句）会向用户显示一个确认对话框。如果用户单击了OK，则通过一个警告框向用户显示消息I’m so glad you’re sure!。如果用户单击的是Cancel按钮，则通过警告框显示I’m sorry to hear you’re not sure.。这种模式经常在用户想要执行删除操作的时候使用，例如删除电子邮件。

最后一种对话框是通过prompt()方法生成的，这是一个“提示”框，用于提示用户输入一些文本。提示框除了显示OK和Cancel按钮之外，还会显示一个文本输入域，以供用户在其中输入内容。prompt()方法接受两个参数：要显示给用户的文本提示和文本输入域的默认值（可以是一个空字符串）。调用prompt(“What\’s your name?”, “Michael”)会得到如图8-5所示的对话框。

图形用户界面, 文本, 应用程序

描述已自动生成

图 8-5

如果用户单击了OK按钮，则prompt()返回文本输入域的值；如果用户单击了Cancel或没有单击OK而是通过其他方式关闭了对话框，则该方法返回null。下面是一个例子。

1.       var result = prompt("What\'s your name?", "Michael");

2.       if (result != null) {

3.         alert("Welcome, " + result);

4.       }

5.

综上所述，这些系统对话框很适合向用户显示消息并请用户作出决定。由于不涉及HTML、CSS或JavaScript，因此它们是增强Web应用程序的一种便捷方式。

除了上述三种对话框之外，Google Chrome浏览器还引入了一种新特性。如果当前脚本在执行过程中会打开两个或多个对话框，那么从第二个对话框开始，每个对话框中都会显示一个复选框，以便用户阻止后续的对话框显示，除非用户刷新页面（见图8-6）。

图形用户界面, 文本, 应用程序

描述已自动生成

图 8-6

如果用户勾选了其中的复选框，并且关闭了对话框，那么除非用户刷新页面，所有后续的系统对话框（包括警告框、确认框和提示框）都会被屏蔽。Chrome没有就对话框是否显示向开发人员提供任何信息。由于浏览器会在空闲时重置对话框计数器，因此如果两次独立的用户操作分别打开两个警告框，那么这两个警告框中都不会显示复选框。而如果是同一次用户操作会生成两个警告框，那么第二个警告框中就会显示复选框。这个新特性出现以后，IE9和Firefox 4也实现了它。

还有两个可以通过JavaScript打开的对话框，即“查找”和“打印”。这两个对话框都是异步显示的，能够将控制权立即交还给脚本。这两个对话框与用户通过浏览器菜单的“查找”和“打印”命令打开的对话框相同。而在JavaScript中则可以像下面这样通过window对象的find()和print()方法打开它们。

1.       // 显示“打印”对话框

2.       window.print();

3.

4.       // 显示“查找”对话框

5.       window.find();

6.

这两个方法同样不会就用户在对话框中的操作给出任何信息，因此它们的用处有限。另外，既然这两个对话框是异步显示的，那么Chrome的对话框计数器就不会将它们计算在内，所以它们也不会受用户禁用后续对话框显示的影响。

8.2 location对象

location是最有用的BOM对象之一，它提供了与当前窗口中加载的文档有关的信息，还提供了一些导航功能。事实上，location对象是很特别的一个对象，因为它既是window对象的属性，也是document对象的属性；换句话说，window.location和document.location引用的是同一个对象。location对象的用处不只表现在它保存着当前文档的信息。还表现在它将URL解析为独立的片段，让开发人员可以通过不同的属性访问这些片段。下表列出了location对象的所有属性（注：省略了每个属性前面的location前缀）。

|  |  |  |
| --- | --- | --- |
| 属性名 | 例子 | 说明 |
| hash | “#contents” | 返回URL中的hash（），如果URL中不包含散列，则返回空字符串。 |
| host | “www.wrox.com:80” | 返回服务器名称和端口号（如果有） |
| hostname | “www.wrox.com” | 返回不带端口号的服务器名称 |
| href | “http://www.wrox.com” | 返回当前加载页面的完整URL。而location对象的toString()方法也返回这个值。 |
| pathname | “/WileyCDA/” | 返回URL中的目录和（或）文件名。 |
| port | “8080” | 返回URL中指定的端口号。如果URL中不包含端口号，则这个属性返回空字符串。 |
| protocol | “http:” | 返回页面使用的协议。通常是http:或https: |
| search | “?q=javascript” | 返回URL的查询字符串。这个字符串以问号开头。 |

8.2.1 查询字符串参数

虽然通过上面的属性可以访问到location对象的大多数信息，但其中访问URL包含的查询字符串的属性并不方便。尽管location.search返回从问号到URL末尾的所有内容，但却没有办法逐个访问其中的每个查询字符串参数。为此，可以像下面这样创建一个函数，用以解析查询字符串，然后返回包含所有参数的一个对象：

1.       function getQueryStringArgs() {

2.         // 取得查询字符串并去掉开头的问号

3.         var qs = location.search.length > 0 ? location.search.substring(1) : "";

4.

5.         // 保存数据对象

6.         args = {};

7.

8.         // 取得每一项

9.         items = qs.length ? qs.split("&") : [];

10.         (item = null), (name = null), (value = null);

11.         // 在for循环中使用

12.         i = 0, len = items.length;

13.         for (i = 0; i < len; i++) {

14.           item = items[i].split("=");

15.           name = decodeURIComponent(item[0]);

16.           value = decodeURIComponent(item[1]);

17.

18.           if (name.length) {

19.             args[name] = value;

20.           }

21.         }

22.

23.         return args;

24.       }

25.

LocationExample01.html

这个函数的第一步是先去掉查询字符串开头的问号。当然，前提是location.search中必须要包含一或多个字符。然后，所有参数将被保存在args对象中，该对象以字面量形式创建。接下来，根据和号（＆）来分割查询字符串，并返回name＝value格式的字符串数组。下面的for循环会迭代这个数组，然后再根据等号分割每一项，从而返回第一项是参数名，第二项是参数值的数组。再使用decodeURLComponent分别解码name和value（因为查询字符串应该是被编码过的）。最后，将name作为args对象的属性，将value作为相应属性的值。下面给出了使用这个函数的示例。

1.       // 假设查询字符串是?q=javascript&num=10

2.       var args = getQueryStringArgs();

3.       alert(args["q"]); // Javascript

4.       alert(args["num"]); // 10

5.

可见，每个查询字符串参数都成了返回对象的属性。这样就极大地方便了对每个参数的访问。

8.2.2 位置操作

使用location对象可以通过很多方式来改变浏览器的位置。首先，也是最常用的方式，就是使用assign方法并为其传递一个URL，如下所示。

1. location.assign(“http://www.wrox.com”);

2.

这样，就可以立即打开新URL并在浏览器的历史记录中生成一条记录。如果是将location.href或window.location设置为一个URL值，也会以该值调用assign()方法。例如，下列两行代码与显示调用assign()方法的效果完全一样。

1.       window.location = "http://www.wrox.com";

2.       location.href = "http://www.wrox.com";

3.

在这些改变浏览器位置的方法中，最常用的是设置location.href属性。

另外，修改location对象的其他属性也可以改变当前加载的页面。下面的例子展示了通过将hash、search、hostname、pathname和port属性设置为新值来改变URL。

1.       // 假设初始URL为http://www.wrox.com/WileyCDA/

2.       // 将URL修改为http://www.wrox.com/WileyCDA/?q=javascript

3.       location.search = "?q=javascript";

4.       // 将URL修改为http://www.yahoo.com/WileyCDA/

5.       location.hostname = "www.yahoo.com";

6.       // 将URL修改为http://www.yahoo.com/mydir/

7.       location.pathname = "mydir";

8.       // 将URL修改为http://www.yahoo.com:8080/WileyCDA/

9.       location.port = 8080;

10.

在IE8、Firefox 1、safari、Opera9+和Chrome中，修改hash的值会在浏览器的历史记录中生成一条新记录。在IE的早期版本中，hash属性不会在用户单击“后退”和“前进”按钮时被更新，而只会在用户单击包含hash的URL时才会被更新。

当通过上述任何一种方式修改URL之后，浏览器的历史记录中就会生成一条新记录，因此用户通过单击“后退”按钮都会导航到前一个页面。要禁用这种行为，可以使用replace()方法。这个方法只接受一个参数，即要导航到的URL；结果虽然会导致浏览器位置改变，但不会在历史记录中生成新记录。在调用replace()方法之后，用户不能回到前一个页面，来看下面的例子：

1.     <p>You Won't be able to get back here</p>

2.     <script type="text/javascript">

3.       setTimeout(function () {

4.         location.replace("http://www.wrox.com");

5.       }, 1000);

6.     </script>

7.

LocationReplaceExample01.html

如果将这个页面加载到浏览器中，浏览器就会在1秒钟后重新定向到[www.wrox.com](http://www.wrox.com)。然后，“后退”按钮将处于禁用状态，如果不重新输入完整的URL，则无法返回示例页面。

与位置有关的最后一方法时reload()，作用是重新加载当前显示的页面。如果调用reload()时不传递任何参数，页面就会以最有效的方式重新加载。也就是说，如果页面自上次请求以来并没有改变过，页面就会从浏览器缓存中重新加载。如果要强制从服务器重新加载，则需要像下面这样为该方法传递参数true。

1.       location.reload();    // 重新加载（有可能从缓存中加载）

2.       location.reload(true);  // 重新加载（从服务器重新加载）

3.

位于reload()调用之后的代码可能会也可能不会执行，这要取决于网络延迟或系统资源等因素。为此，最好将reload()放在代码的最后一行。

8.3 navigator对象

最早由Netscape Navigator 2.0引入的navigator对象，现在已经成为识别客户端浏览器的事实标准。虽然其他浏览器也通过其他方式相同或相似的信息（例如，IE中的window.clientFormation和Opera中的window.opera）。但navigator对象却是所有支持JavaScript的浏览器所共有的。与其他BOM对象的情况一样，每个浏览器中的navigator对象也都有一套自己的属性。下表列出了存在于所有浏览器中的属性和方法，以及支持它们的浏览器版本。

|  |  |
| --- | --- |
| 属性和方法 | 说明 |
| appCodeName | 浏览器的名称。通常都是Mozilla，即使在非Mozilla浏览器中也是如此 |
| appMinorVersion | 次版本信息 |
| appName | 完整的浏览器名称 |
| appVersion | 浏览器的版本。一般不与实际的浏览器版本对应 |
| buildID | 浏览器编译版本 |
| cookieEnabled | 表示cookie是否启用 |
| cpuClass | 客户端计算机中使用的CPU类型（x86、68K、Alpha、PPC或Other） |
| javaEnabled() | 表示当前浏览器是否启用了Java |
| language | 浏览器的主语言 |
| mimeTypes | 在浏览器中注册的MIME类型数组 |
| onLine | 表示浏览器是否连接到了因特网 |
| opsProfile | 似乎早就不用了。查不到相关文档 |
| oscpu | 客户端计算机的操作系统或使用的CPU |
| platform | 浏览器所在的系统平台 |
| plugins | 浏览器中安装的插件信息的数组 |
| preference() | 设置用户的首选项 |
| product | 产品名称（如Gecko） |
| productSub | 关于产品的次要信息（如Gecko的版本） |
| registerContentHandler() | 针对特定的MIME类型将一个站点注册为处理程序 |
| registerProtocolHandler() | 针对特定的协议将一个站点注册为处理程序 |
| securityPolicy | 已经废弃。安全策略的名称。为了与Netscape Navigator4向后兼容而保留下来 |
| systemLanguage | 操作系统的语言 |
| taintEnabled() | 已经废弃。表示是否允许变量被修改（taint）。为了与netscap Navigator3向后兼容而保留下来 |
| userAgent | 浏览器的用户带来字符串 |
| userLanguage | 操作系统的默认语言 |
| userProfile | 借以访问用户个人信息的对象 |
| vendor | 浏览器的品牌 |
| vendorSub | 有关供应商的次要信息 |

表中的这些navigator对象的属性通常用于检测显示网页的浏览器类型（第9章会详细讨论）。

8.3.1 检测插件

检测浏览器中是否安装了特定的插件是一种最常见的检测例程。对于非IE浏览器，可以使用plugins数组来达到这个目的。该数组中的每一项都包含下列属性。

* name：插件的名字。
* description：插件的描述。
* filename：插件的文件名。
* length：插件所处理的MIME类型数量。

一般来说，name属性中会包含检测插件必需的所有信息，但有时候也不完全如此。在检测插件时，需要像下面这样循环迭代每个插件并将插件的name与给定的名字进行比较。

1.       // 检测插件（在IE中无效）

2.       function hasPlugin(name) {

3.         name = name.toLowerCase();

4.         for (var i = 0; i < navigator.plugins; i++) {

5.           if (navigator.plugins[i].name.toLowerCase().indexOf(name) > -1) {

6.             return true;

7.           }

8.         }

9.         return false;

10.       }

11.

12.       // 检测Flash

13.       alert(hasPlugin("Flash"));

14.

15.       // 检测QuickTime

16.       alert(hasPlugin("QuickTime"));

17.

PluginsDetectionExample01.html

这个hasPlugin()函数接受一个参数：要检测的插件名。第一步是将传入的名称转换为小写形式，以便于比较。然后，迭代plugins数组，通过indexOf()检测每个name属性，以确定传入的名称是否出现在字符串的某个地方。比较的字符串都使用小写形式可以避免因大小写不一致导致的错误。而传入的参数应该尽可能具体，以避免混淆。应该说，像Flash和QuickTime这样的字符串就比较具体了，不容易导致混淆。在Firefox、Safari、Opera和Chrome中可以使用这种方法来检测插件。

每个插件对象本身也是一个MimeType对象的数组，这些对象可以通过方括号语法来访问。每个MimeType对象有4个属性：包含MIME类型描述的description、回指插件对象的enabledPlugin、表示与MIME类型对应的文件扩展名的字符串suffixes（以逗号分隔）和表示完整的MIME类型字符串的type。

检查IE中的插件比较麻烦，因为IE不支持Netscape式的插件。在IE中检测插件的唯一方式就是使用专有的ActiveXObject类型，并尝试创建一个特定插件的实例。IE是以COM对象的方式实现插件的，而COM对象使用唯一标识符来标识。因此，要想检查特定的插件，就必须知道其COM标识符。例如，Flash的标识符是ShockwaveFlash.ShockwaveFlash。知道唯一标识符之后，就可以编写类似下面的函数来检测IE中是否安装了相应插件了。

1. // 检测IE中的插件

2. function hasIEPlugin(name) {

3.     try {

4.         new ActiveXObject(name);

5.         return true;

6.     } catch (ex) {

7.         return false;

8.     }

9. }

10.

11. // 检测Flash

12. alert(hasIEPlugin("ShockwaveFlash.ShockwaveFlash"));

13.

14. // 检测QuickTime

15. alert(hasIEPlugin("QuickTime.QuickTime"));

16.

PluginDetectingExample02.html

在这个例子中，函数hasIEPlugin()只接受一个COM标识符作为参数。在函数内部，首先会尝试创建一个COM对象的实例。这样，如果实例化成功，则函数返回true；否则，如果抛出了错误，则执行catch块，结果就会返回false。例子最后检测IE中是否安装了Flash和QuickTime插件。

鉴于检测这两种插件的方法差别太大，因此典型的做法是针对每个插件分别创建检测函数，而不是使用前面介绍的通用检测方法。来看下面的例子。

1. // 检测所有浏览器中的Flash

2. function hasFlash() {

3.     var result = hasPlugin("Flash");

4.     if (!result) {

5.         result = hasIEPlugin("ShockwaveFlash.Shockwave.Flash");

6.     }

7.

8.     return result;

9. }

10.

11. // 检测所有浏览器中的QuickTime

12. function hasQuickTime() {

13.     var result = hasPlugin("QuickTime");

14.     if (!result) {

15.         result = hasIEPlugin("ShockwaveFlash.SockwaveFlash");

16.     }

17.     return result;

18. }

19.

20. // 检测Flash

21. alert(hasFlash());

22.

23. // 检测QuickTime

24. alert(hasQuickTime());

25.

上面的代码中定义了两个函数：hasFlash()和hasQuickTime()。每个函数都先尝试使用不针对IE的插件检测方法。如果返回了false（在IE中会这样），那么再使用针对IE的插件检测方法。如果IE的插件检测方法再返回false，则整个方法也将返回false。只要任何一次检测返回true，整个方法都会返回true。

plugins集合有一个叫refresh()的方法，用于刷新plugins以反映最新安装的插件。这个方法接受一个参数：表示是否应该重新加载页面的一个布尔值。如果将这个值设置为true，则会重新加载包含插件的所有页面；否则，只更新plugins集合，不重新加载页面。

8.3.2 注册处理程序

Firefox 2为navigator对象新增了registerContentHandler()和registerProtocolHandler()方法（这两个方法是在HTML5中定义的，相关内容将在第22章讨论）。这两个方法可以让一个站点指明它可以处理特定类型的信息。随着RSS阅读器和在线电子邮件程序的兴起，注册处理程序就为像桌面应用程序一样默认使用这些在线应用程序提供了一种方式。

其中，registerContentHandler()方法接收三个参数：要处理的MIME类型、可以处理该MIME类型的页面的URL以及应用程序的名称。举个例子，要将一个站点注册为处理RSS源的处理程序，可以使用如下代码。

1. navigator.registerContentHandler("application/rss+xml", "http://www.somereader.com?feed=%s", "Some Reader");

2.

第一个参数是RSS源的MIME类型。第二个参数是应该接收RSS源URL的URL，其中的%s表示RSS源URL，由浏览器自动插入。当下一次请求RSS源时，浏览器就会打开指定的URL，而相应的Web应用程序将以适当方式来处理该请求。

Firefox 4及之前版本只允许在registerContentHandler()方法中使用三个MIME类型：application/rss+xml、application/atom+xml和application/vnd.mozilla.maybe.feed。这三个MIME类型的作用都一样。即为RSS或ATOM新闻源（feed）注册处理程序。

类似的调用方式也适用于registerProtocolHandler()方法，它也接收三个参数：要处理的协议（例如，mailto或ftp）、处理该协议的页面的URL和应用程序的名称。例如，要将一个应用程序注册为默认的邮件客户端，可以使用如下代码。

1. navigator.registerProtocolHandler("mailto", "http://www.baidu.com?cmd=%s", "Some Mail Client");

2.

这个例子注册了一个mailto协议的处理程序，该程序指向一个基于web的电子邮件客服端。同样，第二个参数仍然是处理相应请求的URL，而%s则表示原始的请求。

Firefox 2虽然实现了registerProtocolHandler()，但该方法还不能用。Firefox 3完整实现这个方法。

8.4 screen对象

JavaScript中有几个对象在编程中用处不大，而screen对象就是其中之一。screen对象基本上只用来表明客户端的能力，其中包括浏览器窗口外部的显示器的信息，如像素的宽度和高度等。每个浏览器中的screen对象都包含着各不相同的属性，下表列出了所有属性及支持相应属性的浏览器。

|  |  |
| --- | --- |
| 属性 | 说明 |
| availHeight | 屏幕的像素高度减系统部件高度之后的值（只读） |
| availLeft | 未被系统部件占用的最左侧的像素值（只读） |
| availTop | 未被系统部件占用的最上方的像素值（只读） |
| availWidth | 屏幕的像素宽度减系统部件宽度之后的值（只读） |
| bufferDepth | 读、写用于呈现屏外位图的位数。 |
| colorDepth | 用于表现颜色的位数；多数系统都是32（只读） |
| deviceXDPI | 屏幕实际的水平DPI |
| deviceYDPI | 屏幕实际的垂直DPI |
| fontSmoothingEnabled | 表示是否启动了字体平滑（只读） |
| height | 屏幕的像素高度 |
| left | 当前屏幕距左边的像素距离 |
| logicalXDPI | 屏幕逻辑的水平DPI |
| logicalYDPI | 屏幕逻辑的垂直DPI |
| pixelDepth | 屏幕的位深（只读） |
| top | 当前屏幕距上边的像素距离 |
| updateInterval | 读、写以毫秒表示的屏幕刷新时间间隔 |
| width | 屏幕的像素宽度 |

这些信息经常集中出现在测定客户端能力的站点跟踪工具中，但通常不会用于影响功能。不过，有时候也可能会用到其中的信息来调整浏览器窗口大小，使其占据屏幕的可用空间，例如：

1. window.resizeTo(screen.availWidth, screen.availHeight);

2.

前面曾经提到过，许多浏览器都会禁用调整浏览器窗口大小的能力，因此上面这行代码不一定在所有环境下都有效。

涉及移动设备的屏幕大小时，情况有点不一样。运行IOS的设备始终会像是把设备竖着拿在手里一样，因此返回的值768 x 1024。而Android设备则会相应调用screen.width和screen.height的值。

8.5 history对象

history对象保存着用户上网的历史记录，从窗口被打开的那一刻算起。因为history是window对象的属性，因此每个浏览器窗口、每个标签页乃至每个框架，都有自己的history对象与特定的window对象关联。处于安全方面的考虑，开发人员无法得知用户浏览过的URL。不过，借由用户访问过的页面列表，同样可以在不知道实际URL的情况下实现后退与前进。

使用go()方法可以在用户的历史记录中任意跳转，可以向后也可以向前。这个方法接受一个参数，表示向后或向前跳转的页面数的一个整数值。负数表示向后跳转（类似于单击浏览器的“后退”按钮），正数表示向前跳转（类似于单击浏览器的“前进”按钮）。来看下面的例子。

1. // 后退一页

2. history.go(-1);

3.

4. // 前进一页

5. history.go(1);

6.

7. // 前进两页

8. history.go(2);

9.

也可以给go()方法传递一个字符串参数，此时浏览器会跳转到历史记录中包含该字符串的第一个位置---可能后退，也可能前进，具体要看哪个位置最近。如果历史记录中不包含该字符串，那么这个方法什么也不做，例如：

1. // 跳转到最近的wrox.com页面

2. history.go("wrox.com");

3.

4. // 跳转到最近的nczonline.net页面

5. history.go("nczonline.net");

6.

另外，可以使用两个简写方法back()和forward()来替代go()。顾名思义，这两个方法可以模仿浏览器的“后退”和“前进”按钮。

1. // 后退一页

2. history.back();

3.

4. // 前进一页

5. history.forward();

6.

除了上述几个方法外，history对象还有一个length属性，保存着历史记录的数量。这个数量包括所有历史记录，即所有向后和向前的记录。对于加载到窗口、标签页或框架中的第一个页面而言，history.length等于0。通过下面这样测试该属性的值，可以确定用户是否一开始就打开了你的页面。

1. if (history.length == 0) {

2.   // 这应该是用户打开窗口后第一个页面。

3. }

4.

虽然history并不常用，但在创建自定义的“后退”和“前进”按钮，以及检测当前页面是不是用户历史记录中的第一个页面时，还是必须使用它。

8.6 小结

浏览器对象模型（BOM）以window对象为依托，表示浏览器窗口以及页面可见区域。同时，window对象还是ECMAScript中的global对象，因而所有全局变量和函数都是它的属性，且所有原生构造函数及其他函数也都存在于它的命名空间下。本章讨论了下列BOM的组成部分。

* 在使用框架时，每个框架都有自己的window对象以及所有原生构造函数及其他函数的副本。每个框架都保存在frames集合中，可以通过位置或通过名称来访问。
* 有一些窗口指针，可以用来引用其他框架，包括父框架。
* top对象始终指向最外围的框架，也就是整个浏览器窗口。
* parent对象标识包含当前框架的框架，而self对象则回指window。
* 使用location对象可以通过编程方式来访问浏览器的导航系统。设置相应的属性，可以逐段或整体性地修改浏览器的URL。
* 调用replace()方法可以导航到一个新URL，同时该URL会替换浏览器历史记录中当前显示的页面。
* navigator对象提供了与浏览器有关的信息。到底提供哪些信息，很大程度上取决于用户的浏览器；不过，也有一些公共的属性（如userAgent）存在于所有浏览器中。

BOM中还有两个对象：screen和history，但它们的功能有限。screen对象中保存着与客户端显示器有关的信息，这些信息一般只用于站点分析。history对象为访问浏览器的历史记录开了一个小缝隙，开发人员可以据此判断历史记录的数量，也可以在历史记录中向后或向前导航到任意页面。