

第 8 章

BOM

本章内容

- 理解 window 对象——BOM 的核心
- 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）则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 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，它既是通过 JavaScript 访问浏览器窗口的一个接口，又是 ECMAScript 规定的 Global 对象。这意味着在网页中定义的任何一个对象、变量和函数，都以 window 作为其 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。

```
var age = 29;
function sayAge(){
    alert(this.age);
}

alert(window.age);    //29
sayAge();             //29
window.sayAge();      //29
```

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 访问变量 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`, 最终显示的仍然是正确的结果。

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, 而直接在 `window` 对象上定义的属性可以。例如:



```
var age = 29;
window.color = "red";

//在 IE < 9 时抛出错误, 在其他所有浏览器中都返回 false
delete window.age;

//在 IE < 9 时抛出错误, 在其他所有浏览器中都返回 true
delete window.color; //returns true

alert(window.age); //29
alert(window.color); //undefined
```

DeleteOperatorExample01.htm

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:

```
//这里会抛出错误, 因为 oldValue 未定义
var newValue = oldValue;

//这里不会抛出错误, 因为这是一次属性查询
//newValue 的值是 undefined
var newValue = window.oldValue;
```

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或方法。可是, 在全局作用域中声明的所有变量和函数, 照样会变成 `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 `frames` 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:



```
<html>
<head>
  <title>Frameset Example</title>
</head>
<frameset rows="160,*">
  <frame src="frame.htm" name="topFrame">
  <frameset cols="50%,50%">
    <frame src="anotherframe.htm" name="leftFrame">
    <frame src="yetanotherframe.htm" name="rightFrame">
  </frameset>
</frameset>
</html>
```

FramesetExample01.htm

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` 对象始终指向最高（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。

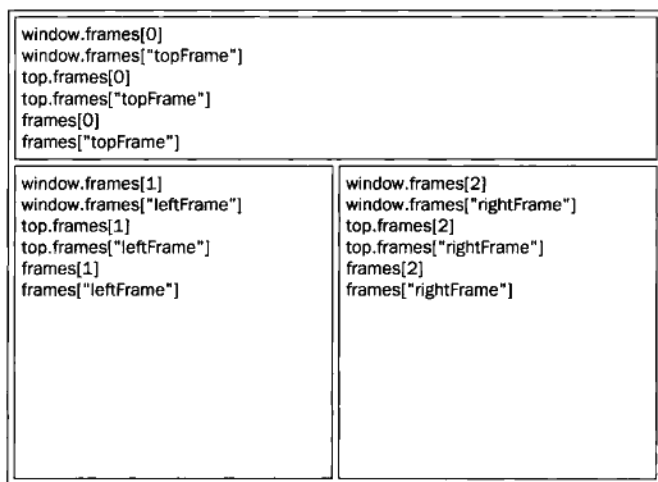


图 8-1

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。



```
<html>
<head>
  <title>Frameset Example</title>
```

```
</head>
<frameset rows="100,*">
  <frame src="frame.htm" name="topFrame">
  <frameset cols="50%,50%">
    <frame src="anotherframe.htm" name="leftFrame">
    <frame src="anotherframeset.htm" name="rightFrame">
  </frameset>
</frameset>
</html>
```

frameset1.htm

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。

```
<html>
<head>
  <title>Frameset Example</title>
</head>
<frameset cols="50%,50%">
  <frame src="red.htm" name="redFrame">
  <frame src="blue.htm" name="blueFrame">
</frameset>
</html>
```

anotherframeset.htm

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 对象指向的就是 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。

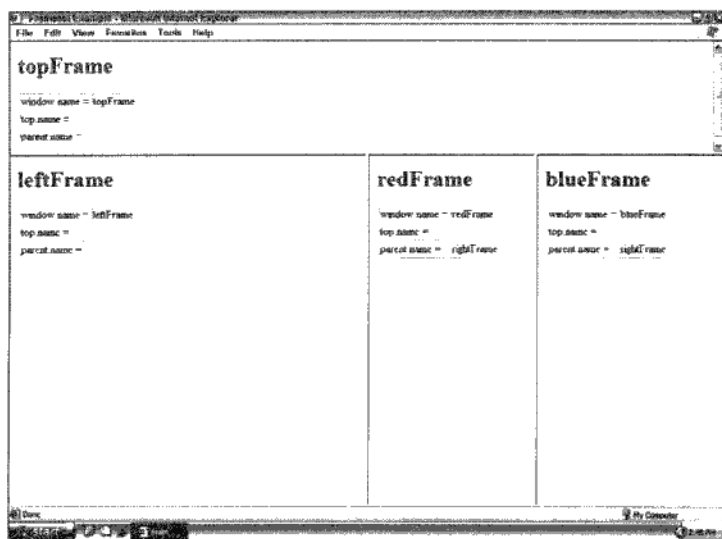


图 8-2

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是 `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。



```
var leftPos = (typeof window.screenLeft == "number") ?  
    window.screenLeft : window.screenX;  
var topPos = (typeof window.screenTop == "number") ?  
    window.screenTop : window.screenY;
```

WindowPositionExample01.htm

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 中), 则取得这两个属性的值。如果不存在 (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 和 Chrome 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 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。这两个方法都接收两个参数, 其中

`moveTo()` 接收的是新位置的 x 和 y 坐标值, 而 `moveBy()` 接收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:

```
//将窗口移动到屏幕左上角
window.moveTo(0,0);

//将窗向下移动 100 像素
window.moveBy(0,100);

//将窗口移动到(200,300)
window.moveTo(200,300);

//将窗口向左移动 50 像素
window.moveBy(-50,0);
```

需要注意的是, 这两个方法可能会被浏览器禁用; 而且, 在 Opera 和 IE 7 (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^①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 IE、Firefox、Safari、Opera 和 Chrome 中, `document.documentElement.clientWidth` 和 `document.documentElement.clientHeight` 中保存了页面视口的信息。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` 取得相同信息。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。



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

if (typeof pageWidth != "number"){
    if (document.compatMode == "CSS1Compat"){
        pageWidth = document.documentElement.clientWidth;
        pageHeight = document.documentElement.clientHeight;
    } else {
        pageWidth = document.body.clientWidth;
        pageHeight = document.body.clientHeight;
    }
}
```

WindowSizeExample01.htm

^① 这里所谓的“页面视图容器”指的是 Opera 中单个标签页对应的浏览器窗口。

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, 就使用 `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, 最好是先检测一下用户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, 推荐你读一读这篇文章。

另外, 使用 `resizeTo()` 和 `resizeBy()` 方法可以调整浏览器窗口的大小。这两个方法都接收两个参数, 其中 `resizeTo()` 接收浏览器窗口的新的宽度和新高度, 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。

```
//调整到 100×100
window.resizeTo(100, 100);

//调整到 200×150
window.resizeBy(100, 50);

//调整到 300×300
window.resizeTo(300, 300);
```

需要注意的是, 这两个方法与移动窗口位置的方法类似, 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, 最后一个参数只在不打开新窗口的情况下使用。

如果为 `window.open()` 传递了第二个参数, 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。

```
//等同于< a href="http://www.wrox.com" target="topFrame"></a>
window.open("http://www.wrox.com/", topFrame);
```

调用这行代码,就如同用户单击了 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 或_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	表示是否在浏览器窗口中显示地址栏。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。

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

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


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

```
//调整大小  
wroxWin.resizeTo(500,500);
```

```
//移动位置  
wroxWin.moveTo(100,100);
```

调用 `close()` 方法还可以关闭新打开的窗口。

```
wroxWin.close();
```

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,已经没有任何用处了。

```
wroxWin.close();  
alert(wroxWin.closed); //true
```

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:

```
var wroxWin = window.open("http://www.wrox.com/", "wroxWindow",  
    "height=400,width=400,top=10,left=10,resizable=yes");  
  
alert(wroxWin.opener == window); //true
```

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,那么新标签页就不能运行在独立的进程中。在 Chrome 中,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。

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

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。

2. 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还是假。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 中的 IE6 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信息显示给用户。换句话说，只能通过单击或者按键来打开弹出窗口。

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。

3. 弹出窗口屏蔽程序

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弹出窗口屏蔽掉。于是，在弹出窗口被屏蔽时，就应该考虑两种可能性。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，只要检测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。

```
var wroxWin = window.open("http://www.wrox.com", "_blank");
if (wroxWin == null){
    alert("The popup was blocked!");
}
```

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。



```
var blocked = false;

try {
    var wroxWin = window.open("http://www.wrox.com", "_blank");
    if (wroxWin == null){
        blocked = true;
    }
} catch (ex){
    blocked = true;
}

if (blocked){
    alert("The popup was blocked!");
}
```

PopupBlockerExample01.htm

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，检测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时间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。



```
// 不建议传递字符串!
setTimeout("alert('Hello world!') ", 1000);

// 推荐的调用方式
setTimeout(function() {
    alert("Hello world!");
}, 1000);
```

TimeoutExample01.htm

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。

```
// 设置超时调用
var timeoutId = setTimeout(function() {
    alert("Hello world!");
}, 1000);

// 注意：把它取消
clearTimeout(timeoutId);
```

TimeoutExample02.htm

只要是在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，只不过它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。



```
//不建议传递字符串!
setInterval ("alert('Hello world!') ", 10000);

//推荐的调用方式
setInterval (function() {
    alert("Hello world!");
}, 10000);
```

IntervalExample01.htm

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，可以使用 `clearInterval()` 方法并传入相应的间歇调用 ID。取消间歇调用的重要性要远远高于取消超时调用，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。

```
var num = 0;
var max = 10;
var intervalId = null;

function incrementNumber() {
    num++;

    //如果执行次数达到了 max 设定的值，则取消后续尚未执行的调用
    if (num == max) {
        clearInterval(intervalId);
        alert("Done");
    }
}

intervalId = setInterval(incrementNumber, 500);
```

IntervalExample02.htm

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。



```
var num = 0;
var max = 10;

function incrementNumber() {
    num++;

    //如果执行次数未达到 max 设定的值，则设置另一次超时调用
    if (num < max) {
        setTimeout(incrementNumber, 500);
    } else {
        alert("Done");
    }
}

setTimeout(incrementNumber, 500);
```

TimeoutExample03.htm

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 `prompt()` 方法可以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, 而不是由 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。

通常使用 `alert()` 生成的“警告”对话框向用户显示一些他们无法控制的消息, 例如错误消息。而用户只能在看完消息后关闭对话框。

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 8-3

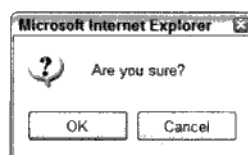


图 8-4

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。

```
if (confirm("Are you sure?")) {  
    alert("I'm so glad you're sure! ");  
} else {  
    alert("I'm sorry to hear you're not sure. ");  
}
```

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调用 `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。提示框中除了显示 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。

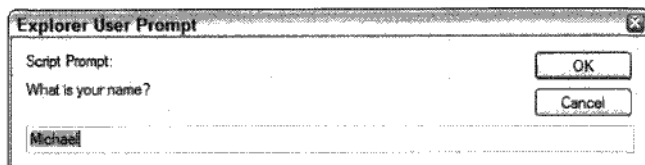


图 8-5

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。

```
var result = prompt("What is your name? ", "");
if (result !== null) {
    alert("Welcome, " + result);
}
```

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)。

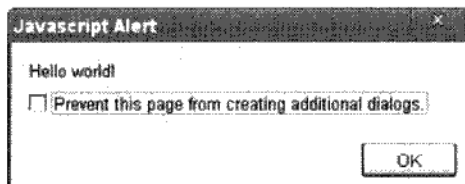


图 8-6

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。

```
//显示“打印”对话框
window.print();

//显示“查找”对话框
window.find();
```

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(#号后跟零或多个字符),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:

```
function getQueryStringArgs(){  
    //取得查询字符串并去掉开头的问号  
    var qs = (location.search.length > 0 ? location.search.substring(1) : ""),  
    //保存数据的对象  
    args = {},  
    //取得每一项  
    items = qs.length ? qs.split("&") : [],  
    item = null,  
    name = null,
```

```

        value = null,

        //在 for 循环中使用
        i = 0,
        len = items.length;

    //逐个将每一项添加到 args 对象中
    for (i=0; i < len; i++){
        item = items[i].split("=");
        name = decodeURIComponent(item[0]);
        value = decodeURIComponent(item[1]);

        if (name.length) {
            args[name] = value;
        }
    }

    return args;
}

```

LocationExample01.htm

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，然后再根据等于号分割每一项，从而返回第一项为参数名，第二项为参数值的数组。再使用 `decodeURIComponent()` 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。

```

//假设查询字符串是?q=javascript&num=10

var args = getQueryStringArgs();

alert(args["q"]);    //"javascript"
alert(args["num"]); //"10"

```

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。

```
location.assign("http://www.wrox.com");
```

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，下列两行代码与显式调用 `assign()` 方法的效果完全一样。

```

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

```

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。

```
//假设初始 URL 为 http://www.wrox.com/WileyCDA/  
  
//将 URL 修改为"http://www.wrox.com/WileyCDA/#section1"  
location.hash = "#section1";  
  
//将 URL 修改为"http://www.wrox.com/WileyCDA/?q=javascript"  
location.search = "?q=javascript";  
  
//将 URL 修改为"http://www.yahoo.com/WileyCDA/"  
location.hostname = "www.yahoo.com";  
  
//将 URL 修改为"http://www.yahoo.com/mydir/"  
location.pathname = "mydir";  
  
//将 URL 修改为"http://www.yahoo.com:8080/WileyCDA/"  
location.port = 8080;
```

每次修改 location 的属性 (hash 除外), 页面都会以新 URL 重新加载。



在 IE8、Firefox 1、Safari 2+、Opera 9+ 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:



```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>You won't be able to get back here</title>  
  </head>  
  <body>  
    <p>Enjoy this page for a second, because you won't be coming back here.</p>  
    <script type="text/javascript">  
      setTimeout(function () {  
        location.replace("http://www.wrox.com/");  
      }, 1000);  
    </script>  
  </body>  
</html>
```

LocationReplaceExample01.htm

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。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个方法是 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。

```
location.reload();           //重新加载 (有可能从缓存中加载)
location.reload(true);       //重新加载 (从服务器重新加载)
```

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提供了相同或相似的信息 (例如, IE 中的 window.clientInformation 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。

属性或方法	说 明	IE	Firefox	Safari/ Chrome	Opera
appName	浏览器的名称。通常都是 Mozilla, 即使在非 Mozilla 浏览器中也是如此	3.0+	1.0+	1.0+	7.0+
appMinorVersion	次版本信息	4.0+	-	-	9.5+
appName	完整的浏览器名称	3.0+	1.0+	1.0+	7.0+
appVersion	浏览器的版本。一般不与实际的浏览器版本对应	3.0+	1.0+	1.0+	7.0+
buildID	浏览器编译版本	-	2.0+	-	-
cookieEnabled	表示 cookie 是否启用	4.0+	1.0+	1.0+	7.0+
cpuClass	客户端计算机中使用的 CPU 类型 (x86、68K、Alpha、PPC 或其他)	4.0+	-	-	-
javaEnabled()	表示当前浏览器中是否启用了 Java	4.0+	1.0+	1.0+	7.0+
language	浏览器的主语言	-	1.0+	1.0+	7.0+
mimeType	在浏览器中注册的 MIME 类型数组	4.0+	1.0+	1.0+	7.0+
onLine	表示浏览器是否连接到了因特网	4.0+	1.0+	-	9.5+
opsProfile	似乎早就不用了。查不到相关文档	4.0+	-	-	-
oscpu	客户端计算机的操作系统或使用的 CPU	-	1.0+	-	-
Platform	浏览器所在的系统平台	4.0+	1.0+	1.0+	7.0+
plugins	浏览器中安装的插件信息的数组	4.0+	1.0+	1.0+	7.0+
preference()	设置用户的首选项	-	1.5+	-	-
product	产品名称 (如 Gecko)	-	1.0+	1.0+	-
productSub	关于产品的次要信息 (如 Gecko 的版本)	-	1.0+	1.0+	-
register- ContentHandler()	针对特定的 MIME 类型将一个站点注册为处理程序	-	2.0+	-	-
register- ProtocolHandler()	针对特定的协议将一个站点注册为处理程序	-	2.0	-	-
securityPolicy	已经废弃。安全策略的名称。为了与 Netscape Navigator 4 向后兼容而保留下来	-	1.0+	-	-

(续)

属性或方法	说 明	IE	Firefox	Safari/ Chrome	Opera
systemLanguage	操作系统的语言	4.0+	-	-	-
taintEnabled()	已经废弃。表示是否允许变量被修改 (taint)。为了与 Netscape Navigator 3 向后兼容而保留下来	4.0+	1.0+	-	7.0+
userAgent	浏览器的用户代理字符串	3.0+	1.0+	1.0+	7.0+
userLanguage	操作系统的默认语言	4.0+	-	-	7.0+
userProfile	借以访问用户个人信息的对象	4.0+	-	-	-
vendor	浏览器的品牌	-	1.0+	1.0+	-
vendorSub	有关供应商的次要信息	-	1.0+	1.0+	-

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。



```
//检测插件 (在 IE 中无效)
function hasPlugin(name){
    name = name.toLowerCase();
    for (var i=0; i < navigator.plugins.length; i++){
        if (navigator.plugins[i].name.toLowerCase().indexOf(name) > -1){
            return true;
        }
    }
    return false;
}

//检测 Flash
alert(hasPlugin("Flash"));

//检测 QuickTime
alert(hasPlugin("QuickTime"));
```

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 MIME 类型字符串的 `type`。

检测 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 中是否安装相应插件了。

```
//检测 IE 中的插件
function hasIEPlugin(name){
    try {
        new ActiveXObject(name);
        return true;
    } catch (ex){
        return false;
    }
}

//检测 Flash
alert(hasIEPlugin("ShockwaveFlash.ShockwaveFlash"));

//检测 QuickTime
alert(hasIEPlugin("QuickTime.QuickTime"));
```

PluginDetectionExample02.htm

在这个例子中，函数 `hasIEPlugin()` 只接收一个 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。之所以要在 `try-catch` 语句中进行实例化，是因为创建未知 COM 对象会导致抛出错误。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。

```
//检测所有浏览器中的 Flash
function hasFlash(){
    var result = hasPlugin("Flash");
    if (!result){
        result = hasIEPlugin("ShockwaveFlash.ShockwaveFlash");
    }
}
```

```
        return result;
    }

    //检测所有浏览器中的 QuickTime
    function hasQuickTime(){
        var result = hasPlugin("QuickTime");
        if (!result){
            result = hasIEPlugin("QuickTime.QuickTime");
        }
        return result;
    }

    //检测 Flash
    alert(hasFlash());

    //检测 QuickTime
    alert(hasQuickTime());
```

PluginDetectionExample03.htm

上面代码中定义了两个函数：hasFlash()和 hasQuickTime()。每个函数都是先尝试使用不针对 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 集合有一个名叫 refresh()的方法，用于刷新 plugins 以反映最新安装的插件。这个方法接收一个参数：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

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使用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。

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

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, 要想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。

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

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 的电子邮件客户端。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	IE	Firefox	Safari/ Chrome	Opera
<code>availHeight</code>	屏幕的像素高度减系统部件高度之后的值 (只读)	√	√	√	√
<code>availLeft</code>	未被系统部件占用的最左侧的像素值 (只读)		√	√	
<code>availTop</code>	未被系统部件占用的最上方的像素值 (只读)		√	√	
<code>availWidth</code>	屏幕的像素宽度减系统部件宽度之后的值 (只读)	√	√	√	√
<code>bufferDepth</code>	读、写用于呈现屏外位图的位数	√			
<code>colorDepth</code>	用于表现颜色的位数; 多数系统都是 32 (只读)	√	√	√	√
<code>deviceXDPI</code>	屏幕实际的水平 DPI (只读)	√			
<code>deviceYDPI</code>	屏幕实际的垂直 DPI (只读)	√			
<code>fontSmoothingEnabled</code>	表示是否启用了字体平滑 (只读)	√			
<code>height</code>	屏幕的像素高度	√	√	√	√
<code>left</code>	当前屏幕距左边的像素距离		√		
<code>logicalXDPI</code>	屏幕逻辑的水平 DPI (只读)	√			
<code>logicalYDPI</code>	屏幕逻辑的垂直 DPI (只读)	√			
<code>pixelDepth</code>	屏幕的位深 (只读)		√	√	√

(续)

属 性	说 明	IE	Firefox	Safari/ Chrome	Opera
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：

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

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，因此返回的值是 768 × 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。出于安全方面的考虑，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 的情况下实现后退和前进。

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。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。

```
//后退一页
history.go(-1);

//前进一页
history.go(1);

//前进两页
history.go(2);
```

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：

```
//跳转到最近的wrox.com页面
history.go("wrox.com");

//跳转到最近的nczonline.net页面
history.go("nczonline.net");
```

另外，还可以使用两个简写方法 back() 和 forward() 来代替 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。


```
//后退一页
history.back();

//前进一页
history.forward();
```

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。通过像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。

```
if (history.length == 0){
    //这应该是用户打开窗口后的第一个页面
}
```

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。



当页面的 URL 改变时, 就会生成一条历史记录。在 IE8 及更高版本、Opera、Firefox、Safari 3 及更高版本以及 Chrome 中, 这里所说的改变包括 URL 中 hash 的变化 (因此, 设置 location.hash 会在这些浏览器中生成一条新的历史记录)。

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, 且所有原生的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 对象表示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。