（续）

设　置	值	说　明
Menubar	"yes"或"no"	表示是否显示菜单栏。默认为"no"
resizable	"yes"或"no"	表示是否可以拖动改变新窗口大小。默认为"no"
scrollbars	"yes"或"no"	表示是否可以在内容过长时滚动。默认为"no"
status	"yes"或"no"	表示是否显示状态栏。不同浏览器的默认值也不一样
toolbar	"yes"或"no"	表示是否显示工具栏。默认为"no"
top	数值	新窗口的 *y* 轴坐标。这个值不能是负值
width	数值	新窗口的宽度。这个值不能小于 100

这些设置需要以逗号分隔的名值对形式出现，其中名值对以等号连接。（特性字符串中不能包含空格。）来看下面的例子：

```
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可缩放的新窗口，大小为 400 像素×400 像素，位于离屏幕左边及顶边各 10 像素的位置。

window.open()方法返回一个对新建窗口的引用。这个对象与普通 window 对象没有区别，只是为控制新窗口提供了方便。例如，某些浏览器默认不允许缩放或移动主窗口，但可能允许缩放或移动通过 window.open()创建的窗口。跟使用任何 window 对象一样，可以使用这个对象操纵新打开的窗口。

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

// 缩放
wroxWin.resizeTo(500, 500);

// 移动
wroxWin.moveTo(100, 100);
```

还可以使用 close()方法像这样关闭新打开的窗口：

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

这个方法只能用于 window.open()创建的弹出窗口。虽然不可能不经用户确认就关闭主窗口，但弹出窗口可以调用 top.close()来关闭自己。关闭窗口以后，窗口的引用虽然还在，但只能用于检查其 closed 属性了：

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

新创建窗口的 window 对象有一个属性 opener，指向打开它的窗口。这个属性只在弹出窗口的最上层 window 对象（top）有定义，是指向调用 window.open()打开它的窗口或窗格的指针。例如：

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

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

虽然新建窗口中有指向打开它的窗口的指针，但反之则不然。窗口不会跟踪记录自己打开的新窗口，因此开发者需要自己记录。

在某些浏览器中，每个标签页会运行在独立的进程中。如果一个标签页打开了另一个，而 window 对象需要跟另一个标签页通信，那么标签便不能运行在独立的进程中。在这些浏览器中，可以将新打开的标签页的 opener 属性设置为 null，表示新打开的标签页可以运行在独立的进程中。比如：

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

wroxWin.opener = null;
```

把 opener 设置为 null 表示新打开的标签页不需要与打开它的标签页通信，因此可以在独立进程中运行。这个连接一旦切断，就无法恢复了。

2. 安全限制

弹出窗口有段时间被在线广告用滥了。很多在线广告会把弹出窗口伪装成系统对话框，诱导用户点击。因为长得像系统对话框，所以用户很难分清这些弹窗的来源。为了让用户能够区分清楚，浏览器开始对弹窗施加限制。

IE 的早期版本实现针对弹窗的多重安全限制，包括不允许创建弹窗或把弹窗移出屏幕之外，以及不允许隐藏状态栏等。从 IE7 开始，地址栏也不能隐藏了，而且弹窗默认是不能移动或缩放的。Firefox 1 禁用了隐藏状态栏的功能，因此无论 window.open()的特性字符串是什么，都不会隐藏弹窗的状态栏。Firefox 3 强制弹窗始终显示地址栏。Opera 只会在主窗口中打开新窗口，但不允许它们出现在系统对话框的位置。

此外，浏览器会在用户操作下才允许创建弹窗。在网页加载过程中调用 window.open()没有效果，而且还可能导致向用户显示错误。弹窗通常可能在鼠标点击或按下键盘中某个键的情况下才能打开。

> **注意** IE 对打开本地网页的窗口再弹窗解除了某些限制。同样的代码如果来自服务器，则会施加弹窗限制。

3. 弹窗屏蔽程序

所有现代浏览器都内置了屏蔽弹窗的程序，因此大多数意料之外的弹窗都会被屏蔽。在浏览器屏蔽弹窗时，可能会发生一些事。如果浏览器内置的弹窗屏蔽程序阻止了弹窗，那么 window.open()很可能会返回 null。此时，只要检查这个方法的返回值就可以知道弹窗是否被屏蔽了，比如：

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

在浏览器扩展或其他程序屏蔽弹窗时，window.open()通常会抛出错误。因此要准确检测弹窗是否被屏蔽，除了检测 window.open()的返回值，还要把它用 try/catch 包装起来，像这样：

```
let blocked = false;

try {
  let wroxWin = window.open("http://www.wrox.com", "_blank");
  if (wroxWin == null){
```

```
      blocked = true;
    }
  } catch (ex){
    blocked = true;
  }
  if (blocked){
    alert("The popup was blocked!");
  }
```

无论弹窗是用什么方法屏蔽的，以上代码都可以准确判断调用 window.open()的弹窗是否被屏蔽了。

> **注意**　检查弹窗是否被屏蔽，不影响浏览器显示关于弹窗被屏蔽的消息。

12.1.7　定时器

JavaScript 在浏览器中是单线程执行的，但允许使用定时器指定在某个时间之后或每隔一段时间就执行相应的代码。setTimeout()用于指定在一定时间后执行某些代码，而 setInterval()用于指定每隔一段时间执行某些代码。

setTimeout()方法通常接收两个参数：要执行的代码和在执行回调函数前等待的时间（毫秒）。第一个参数可以是包含 JavaScript 代码的字符串（类似于传给 eval()的字符串）或者一个函数，比如：

```
// 在 1 秒后显示警告框
setTimeout(() => alert("Hello world!"), 1000);
```

第二个参数是要等待的毫秒数，而不是要执行代码的确切时间。JavaScript 是单线程的，所以每次只能执行一段代码。为了调度不同代码的执行，JavaScript 维护了一个任务队列。其中的任务会按照添加到队列的先后顺序执行。setTimeout()的第二个参数只是告诉 JavaScript 引擎在指定的毫秒数过后把任务添加到这个队列。如果队列是空的，则会立即执行该代码。如果队列不是空的，则代码必须等待前面的任务执行完才能执行。

调用 setTimeout()时，会返回一个表示该超时排期的数值 ID。这个超时 ID 是被排期执行代码的唯一标识符，可用于取消该任务。要取消等待中的排期任务，可以调用 clearTimeout()方法并传入超时 ID，如下面的例子所示：

```
// 设置超时任务
let timeoutId = setTimeout(() => alert("Hello world!"), 1000);

// 取消超时任务
clearTimeout(timeoutId);
```

只要是在指定时间到达之前调用 clearTimeout()，就可以取消超时任务。在任务执行后再调用 clearTimeout()没有效果。

> **注意**　所有超时执行的代码（函数）都会在全局作用域中的一个匿名函数中运行，因此函数中的 this 值在非严格模式下始终指向 window，而在严格模式下是 undefined。如果给 setTimeout()提供了一个箭头函数，那么 this 会保留为定义它时所在的词汇作用域。

setInterval()与 setTimeout()的使用方法类似，只不过指定的任务会每隔指定时间就执行一

次，直到取消循环定时或者页面卸载。setInterval()同样可以接收两个参数：要执行的代码（字符串或函数），以及把下一次执行定时代码的任务添加到队列要等待的时间（毫秒）。下面是一个例子：

```
setInterval(() => alert("Hello world!"), 10000);
```

> **注意** 这里的关键点是，第二个参数，也就是间隔时间，指的是向队列添加新任务之前等待的时间。比如，调用 setInterval()的时间为 01:00:00，间隔时间为 3000 毫秒。这意味着 01:00:03 时，浏览器会把任务添加到执行队列。浏览器不关心这个任务什么时候执行或者执行要花多长时间。因此，到了 01:00:06，它会再向队列中添加一个任务。由此可看出，执行时间短、非阻塞的回调函数比较适合 setInterval()。

setInterval()方法也会返回一个循环定时 ID，可以用于在未来某个时间点上取消循环定时。要取消循环定时，可以调用 clearInterval()并传入定时 ID。相对于 setTimeout()而言，取消定时的能力对 setInterval()更加重要。毕竟，如果一直不管它，那么定时任务会一直执行到页面卸载。下面是一个常见的例子：

```
let num = 0, intervalId = null;
let max = 10;

let incrementNumber = function() {
  num++;

  // 如果达到最大值，则取消所有未执行的任务
  if (num == max) {
    clearInterval(intervalId);
    alert("Done");
  }
}

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

在这个例子中，变量 num 会每半秒递增一次，直至达到最大限制值。此时循环定时会被取消。这个模式也可以使用 setTimeout()来实现，比如：

```
let num = 0;
let max = 10;
let incrementNumber = function() {
  num++;

  // 如果还没有达到最大值，再设置一个超时任务
  if (num < max) {
    setTimeout(incrementNumber, 500);
  } else {
    alert("Done");
  }
}

setTimeout(incrementNumber, 500);
```

注意在使用 setTimeout()时，不一定要记录超时 ID，因为它会在条件满足时自动停止，否则会自动设置另一个超时任务。这个模式是设置循环任务的推荐做法。setIntervale()在实践中很少会在生产环境下使用，因为一个任务结束和下一个任务开始之间的时间间隔是无法保证的，有些循环定时任

务可能会因此而被跳过。而像前面这个例子中一样使用 setTimeout()则能确保不会出现这种情况。一般来说，最好不要使用 setInterval()。

12.1.8　系统对话框

使用 alert()、confirm()和 prompt()方法，可以让浏览器调用系统对话框向用户显示消息。这些对话框与浏览器中显示的网页无关，而且也不包含 HTML。它们的外观由操作系统或者浏览器决定，无法使用 CSS 设置。此外，这些对话框都是同步的模态对话框，即在它们显示的时候，代码会停止执行，在它们消失以后，代码才会恢复执行。

alert()方法在本书示例中经常用到。它接收一个要显示给用户的字符串。与 console.log 可以接收任意数量的参数且能一次性打印这些参数不同，alert()只接收一个参数。调用 alert()时，传入的字符串会显示在一个系统对话框中。对话框只有一个“OK”（确定）按钮。如果传给 alert()的参数不是一个原始字符串，则会调用这个值的 toString()方法将其转换为字符串。

警告框（alert）通常用于向用户显示一些他们无法控制的消息，比如报错。用户唯一的选择就是在看到警告框之后把它关闭。图 12-1 展示了一个警告框。

图　12-1

第二种对话框叫确认框，通过调用 confirm()来显示。确认框跟警告框类似，都会向用户显示消息。但不同之处在于，确认框有两个按钮：“Cancel”（取消）和“OK”（确定）。用户通过单击不同的按钮表明希望接下来执行什么操作。比如，confirm("Are you sure?")会显示图 12-2 所示的确认框。

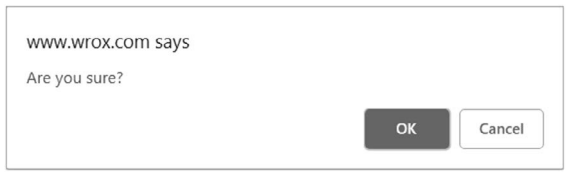

图　12-2

要知道用户单击了 OK 按钮还是 Cancel 按钮，可以判断 confirm()方法的返回值：true 表示单击了 OK 按钮，false 表示单击了 Cancel 按钮或者通过单击某一角上的 X 图标关闭了确认框。确认框的典型用法如下所示：

```
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，第一行代码向用户显示了确认框，也就是 if 语句的条件。如果用户单击了 OK 按钮，

则会弹出警告框显示"I'm so glad you're sure!"。如果单击了 Cancel，则会显示"I'm sorry to hear you're not sure."。确认框通常用于让用户确认执行某个操作，比如删除邮件等。因为这种对话框会完全打断正在浏览网页的用户，所以应该在必要时再使用。

最后一种对话框是提示框，通过调用 prompt()方法来显示。提示框的用途是提示用户输入消息。除了 OK 和 Cancel 按钮，提示框还会显示一个文本框，让用户输入内容。prompt()方法接收两个参数：要显示给用户的文本，以及文本框的默认值（可以是空字符串）。调用 prompt("What is your name?", "Jake")会显示图 12-3 所示的提示框。

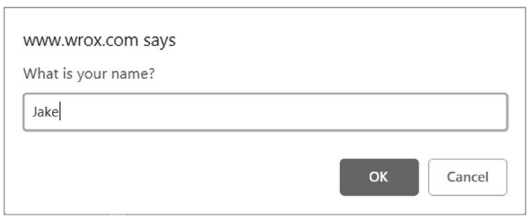

图 12-3

如果用户单击了 OK 按钮，则 prompt()会返回文本框中的值。如果用户单击了 Cancel 按钮，或者对话框被关闭，则 prompt()会返回 null。下面是一个例子：

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

这些系统对话框可以向用户显示消息、确认操作和获取输入。由于不需要 HTML 和 CSS，所以系统对话框是 Web 应用程序最简单快捷的沟通手段。

很多浏览器针对这些系统对话框添加了特殊功能。如果网页中的脚本生成了两个或更多系统对话框，则除第一个之外所有后续的对话框上都会显示一个复选框，如果用户选中则会禁用后续的弹框，直到页面刷新。

如果用户选中了复选框并关闭了对话框，在页面刷新之前，所有系统对话框（警告框、确认框、提示框）都会被屏蔽。开发者无法获悉这些对话框是否显示了。对话框计数器会在浏览器空闲时重置，因此如果两次独立的用户操作分别产生了两个警告框，则两个警告框上都不会显示屏蔽复选框。如果一次独立的用户操作连续产生了两个警告框，则第二个警告框会显示复选框。

JavaScript 还可以显示另外两种对话框：find()和 print()。这两种对话框都是异步显示的，即控制权会立即返回给脚本。用户在浏览器菜单上选择“查找”（find）和“打印”（print）时显示的就是这两种对话框。通过在 window 对象上调用 find()和 print()可以显示它们，比如：

```
// 显示打印对话框
window.print();

// 显示查找对话框
window.find();
```

这两个方法不会返回任何有关用户在对话框中执行了什么操作的信息，因此很难加以利用。此外，因为这两种对话框是异步的，所以浏览器的对话框计数器不会涉及它们，而且用户选择禁用对话框对它们也没有影响。

12.2 location 对象

location 是最有用的 BOM 对象之一，提供了当前窗口中加载文档的信息，以及通常的导航功能。这个对象独特的地方在于，它既是 window 的属性，也是 document 的属性。也就是说，window.location 和 document.location 指向同一个对象。location 对象不仅保存着当前加载文档的信息，也保存着把 URL 解析为离散片段后能够通过属性访问的信息。这些解析后的属性在下表中有详细说明（location 前缀是必需的）。

假设浏览器当前加载的 URL 是 http://foouser:barpassword@www.wrox.com:80/WileyCDA/?q=javascript#contents，location 对象的内容如下表所示。

属　性	值	说　明
location.hash	"#contents"	URL 散列值（井号后跟零或多个字符），如果没有则为空字符串
location.host	"www.wrox.com:80"	服务器名及端口号
location.hostname	"www.wrox.com"	服务器名
location.href	"http://www.wrox.com:80/WileyCDA/?q=javascript#contents"	当前加载页面的完整 URL。location 的 toString() 方法返回这个值
location.pathname	"/WileyCDA/"	URL 中的路径和（或）文件名
location.port	"80"	请求的端口。如果 URL 中没有端口，则返回空字符串
location.protocol	"http:"	页面使用的协议。通常是"http:"或"https:"
location.search	"?q=javascript"	URL 的查询字符串。这个字符串以问号开头
location.username	"foouser"	域名前指定的用户名
location.password	"barpassword"	域名前指定的密码
location.origin	"http://www.wrox.com"	URL 的源地址。只读

12.2.1 查询字符串

location 的多数信息都可以通过上面的属性获取。但是 URL 中的查询字符串并不容易使用。虽然 location.search 返回了从问号开始直到 URL 末尾的所有内容，但没有办法逐个访问每个查询参数。下面的函数解析了查询字符串，并返回一个以每个查询参数为属性的对象：

```
let getQueryStringArgs = function() {
  // 取得没有开头问号的查询字符串
  let qs = (location.search.length > 0 ? location.search.substring(1) : ""),
    // 保存数据的对象
    args = {};

  // 把每个参数添加到 args 对象
  for (let item of qs.split("&").map(kv => kv.split("="))) {
    let name = decodeURIComponent(item[0]),
      value = decodeURIComponent(item[1]);
    if (name.length) {
      args[name] = value;
    }
  }
```

```
  return args;
}
```

这个函数首先删除了查询字符串开头的问号，当然前提是 location.search 必须有内容。解析后的参数将被保存到 args 对象，这个对象以字面量形式创建。接着，先把查询字符串按照&分割成数组，每个元素的形式为 name=value。for 循环迭代这个数组，将每一个元素按照=分割成数组，这个数组第一项是参数名，第二项是参数值。参数名和参数值在使用 decodeURIComponent()解码后（这是因为查询字符串通常是被编码后的格式）分别保存在 name 和 value 变量中。最后，name 作为属性而 value 作为该属性的值被添加到 args 对象。这个函数可以像下面这样使用：

```
// 假设查询字符串为?q=javascript&num=10

let args = getQueryStringArgs();

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

现在，查询字符串中的每个参数都是返回对象的一个属性，这样使用起来就方便了。

URLSearchParams

URLSearchParams 提供了一组标准 API 方法，通过它们可以检查和修改查询字符串。给 URLSearchParams 构造函数传入一个查询字符串，就可以创建一个实例。这个实例上暴露了 get()、set()和 delete()等方法，可以对查询字符串执行相应操作。下面来看一个例子：

```
let qs = "?q=javascript&num=10";

let searchParams = new URLSearchParams(qs);

alert(searchParams.toString());  // " q=javascript&num=10"
searchParams.has("num");         // true
searchParams.get("num");         // 10

searchParams.set("page", "3");
alert(searchParams.toString());  // " q=javascript&num=10&page=3"

searchParams.delete("q");
alert(searchParams.toString());  // " num=10&page=3"
```

大多数支持 URLSearchParams 的浏览器也支持将 URLSearchParams 的实例用作可迭代对象：

```
let qs = "?q=javascript&num=10";

let searchParams = new URLSearchParams(qs);

for (let param of searchParams) {
  console.log(param);
}
// ["q", "javascript"]
// ["num", "10"]
```


12.2.2 操作地址

可以通过修改 location 对象修改浏览器的地址。首先，最常见的是使用 assign()方法并传入一个 URL，如下所示：

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

这行代码会立即启动导航到新 URL 的操作，同时在浏览器历史记录中增加一条记录。如果给 location.href 或 window.location 设置一个 URL，也会以同一个 URL 值调用 assign()方法。比如，下面两行代码都会执行与显式调用 assign()一样的操作：

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

在这 3 种修改浏览器地址的方法中，设置 location.href 是最常见的。

修改 location 对象的属性也会修改当前加载的页面。其中，hash、search、hostname、pathname 和 port 属性被设置为新值之后都会修改当前 URL，如下面的例子所示：

```
// 假设当前 URL 为 http://www.wrox.com/WileyCDA/

// 把 URL 修改为 http://www.wrox.com/WileyCDA/#section1
location.hash = "#section1";

// 把 URL 修改为 http://www.wrox.com/WileyCDA/?q=javascript
location.search = "?q=javascript";

// 把 URL 修改为 http://www.somewhere.com/WileyCDA/
location.hostname = "www.somewhere.com";

// 把 URL 修改为 http://www.somewhere.com/mydir/
location.pathname = "mydir";

// 把 URL 修改为 http://www.somewhere.com:8080/WileyCDA/
location.port = 8080;
```

除了 hash 之外，只要修改 location 的一个属性，就会导致页面重新加载新 URL。

> **注意**　修改 hash 的值会在浏览器历史中增加一条新记录。在早期的 IE 中，点击“后退”和“前进”按钮不会更新 hash 属性，只有点击包含散列的 URL 才会更新 hash 的值。

在以前面提到的方式修改 URL 之后，浏览器历史记录中就会增加相应的记录。当用户单击“后退”按钮时，就会导航到前一个页面。如果不希望增加历史记录，可以使用 replace()方法。这个方法接收一个 URL 参数，但重新加载后不会增加历史记录。调用 replace()之后，用户不能回到前一页。比如下面的例子：

```
<!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>
    setTimeout(() => location.replace("http://www.wrox.com/"), 1000);
  </script>
</body>
</html>
```

浏览器加载这个页面 1 秒之后会重定向到 www.wrox.com。此时，“后退”按钮是禁用状态，即不能返回这个示例页面，除非手动输入完整的 URL。

最后一个修改地址的方法是 reload()，它能重新加载当前显示的页面。调用 reload()而不传参数，页面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没有修改过，浏览器可能会从缓存中加载页面。如果想强制从服务器重新加载，可以像下面这样给 reload()传个 true：

```
location.reload();      // 重新加载，可能是从缓存加载
location.reload(true); // 重新加载，从服务器加载
```

脚本中位于 reload()调用之后的代码可能执行也可能不执行，这取决于网络延迟和系统资源等因素。为此，最好把 reload()作为最后一行代码。

12.3 navigator 对象

navigator 是由 Netscape Navigator 2 最早引入浏览器的，现在已经成为客户端标识浏览器的标准。只要浏览器启用 JavaScript，navigator 对象就一定存在。但是与其他 BOM 对象一样，每个浏览器都支持自己的属性。

> **注意** navigator 对象中关于系统能力的属性将在第 13 章详细介绍。

navigator 对象实现了 NavigatorID、NavigatorLanguage、NavigatorOnLine、NavigatorContentUtils、NavigatorStorage、NavigatorStorageUtils、NavigatorConcurrentHardware、NavigatorPlugins 和 NavigatorUserMedia 接口定义的属性和方法。

下表列出了这些接口定义的属性和方法：

属性/方法	说　明
activeVrDisplays	返回数组，包含 ispresenting 属性为 true 的 VRDisplay 实例
appCodeName	即使在非 Mozilla 浏览器中也会返回"Mozilla"
appName	浏览器全名
appVersion	浏览器版本。通常与实际的浏览器版本不一致
battery	返回暴露 Battery Status API 的 BatteryManager 对象
buildId	浏览器的构建编号
connection	返回暴露 Network Information API 的 NetworkInformation 对象
cookieEnabled	返回布尔值，表示是否启用了 cookie
credentials	返回暴露 Credentials Management API 的 CredentialsContainer 对象
deviceMemory	返回单位为 GB 的设备内存容量
doNotTrack	返回用户的“不跟踪”（do-not-track）设置
geolocation	返回暴露 Geolocation API 的 Geolocation 对象
getVRDisplays()	返回数组，包含可用的每个 VRDisplay 实例
getUserMedia()	返回与可用媒体设备硬件关联的流
hardwareConcurrency	返回设备的处理器核心数量
javaEnabled	返回布尔值，表示浏览器是否启用了 Java
language	返回浏览器的主语言
languages	返回浏览器偏好的语言数组