第 3 章　DOM

本章内容

● 节点的概念

●四个非常实用的　DOM 方法： getElementById　、getElementsByTagName　、getAttribute 和 setAttribute

终于要与 DOM 面对面了。能够向大家介绍　DOM 是笔者的荣幸，我非常乐于带领大家通过 DOM 的眼睛去看世界。

3.1　文档： DOM 中的“ D”

DOM 是“ Document Object Model　”（文档对象模型）的首字母缩写。如果没有 document（文档）， DOM 也就无从谈起。当创建了一个网页并把它加载到　Web 浏览器中时， DOM 就在幕后悄然而生。它将根据你编写的网页文档创建一个文档对象。

在人类语言中，“ 对象 ”这个词的含义往往不那么明确和具体，它几乎可以用来称呼任何一种客观存在的事物。　但在程序设计语言中，　“ 对象 ”这个词的含义非常明确和具体。

3.2　对象： DOM 中的“ O”

在上一章的末尾，我们向大家展示了几个　JavaScript　对象的例子。你们应该还记得，“ 对象 ”是一种独立的数据集合　。与某个特定对象相关联的变量被称为这个对象的属性；可以通过某个特定对象去调用的函数被称为这个对象的方法。

JavaScript　语言里的对象可以分为三种类型：

●用户定义对象　（user-defined　object　）：由程序员自行创建的对象。　本书不讨论这种对象。

●内建对象　（native　object　）：内建在　JavaScript　语言里的对象，如　Array 、Math　和 Date 等。

●宿主对象 （host object ）：由浏览器提供的对象。

在 JavaScript 语言的发展初期，程序员在编写 JavaScript 脚本时经常需要用到一些非常重要的宿主对象，它们当中最基础的是 window对象。

window 对象对应着浏览器窗口本身，这个对象的属性和方法通常被统称为 BOM(浏览器对象模型）——但我觉得称之为 Window Object Model （窗口对象模型）更为贴切。 BOM向程序员提供了 window.open 和 window.blur 等方法，你们在上网冲浪时看到的各种弹出窗口和下拉菜单——其数量之多已经到了泛滥成灾的地步——几乎都是由这些方法负责创建和处理的。难怪 JavaScript 会有一个不好的名声！

值得庆幸的是，在这本书里我们不需要与 BOM打太多的交道。我们将把注意力集中在浏览器窗口的内部而不是浏览器窗口本身。 我们将着重探讨如何对网页的内容进行处理，而用来实现这一目标的载体就是 document对象。

在本书的后续内容里，我们将尽可能地只讨论 document 对象的属性和方法。

现在，我们已经对 DOM中的字母“ D”（document，文档）和字母“ O”（object ，对象）做了解释，那么字母“ M”又代表着什么呢？

3.3 模型： DOM 中的“ M”

DOM中的“ M”代表着“ Model ”（模型），但说它代表着“ Map”（地图）也未尝不可。 模型也好， 地图也罢， 它们的含义都是某种事物的表现形式。 就像一个模型火车代表着一列真正的火车、 一张城市街道图代表着一个实际存在的城市那样， DOM代表着被加载到浏览器窗口里的当前网页：浏览器向我们提供了当前网页的地图（或者说模型），而我们可以通过 JavaScript 去读取这张地图。

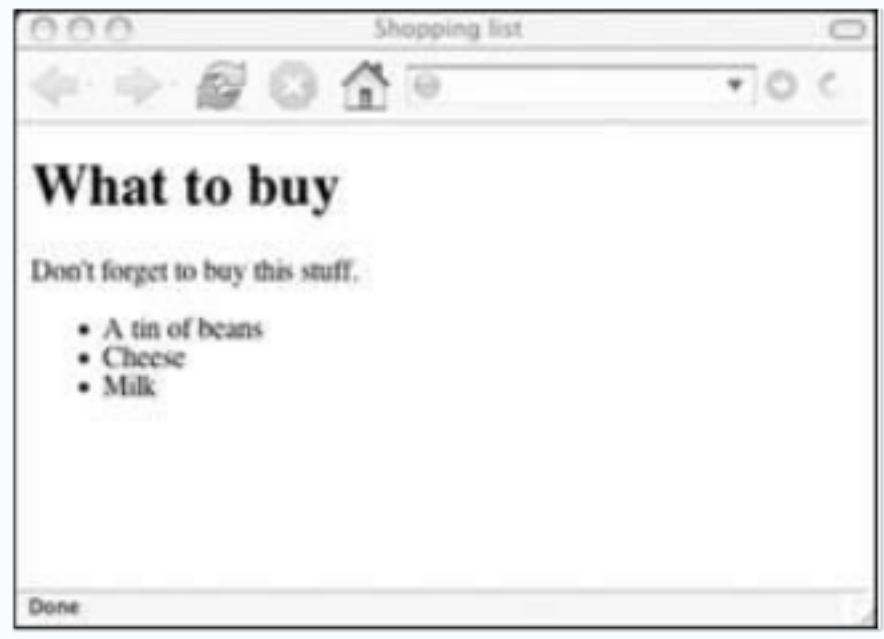
既然是地图，就必须有诸如方向、等高线和比例尺之类的记号。要想看懂和使用地图，就必须知道这些记号的含义和用途——这个道理同样适用于 DOM。要想从 DOM获得信息，我们必须先把各种用来表示和描述一份文档的记号弄明白。

DOM把一份文档表示为一棵树（这里所说的“ 树 ”是数学意义上的概念），这是我们理解和运用这一模型的关键。更具体地说，　　DOM把文档表示为一棵家谱树。

家谱树本身又是一种模型。家谱树的典型用法是表示一个人类家族的谱系并使用 parent （父）、 child （子）、 sibling （兄弟）等记号来表明家族成员之间的关系。 家谱树可以把一些相当复杂的关系简明地表示出来：　　一位特定的家族成员既是某些成员的父辈，又是另一位成员的子辈，同时还是另一位成员的兄弟。

类似于人类家族谱系的情况，家谱树模型也非常适合用来表示一份用　　(X)HTML 语言编写出来的文档。

请看下面这份非常基本的网页，它的内容是一份购物清单。




```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
➥ charset=utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li>Cheese</li>
      <li>Milk</li>
    </ul>
  </body>
</html>
```

这份文档可以用图　3-1 中的模型来表示。

我们来分析一下这个网页的结构。这种分析不仅可以让我们了解它是由哪些元素构成的，还可以让我们了解为什么图 3-1 中的模型可以如此完美地把它表示出来。在对 Doctype 做出声明后，这份文档首先打开了一个 <html> 标签，而这个网页里的所有其他元素都包含在这个元素里。因为所有其他的元素都包含在其内部，所以这个 <html> 标签既没有父辈，也没有兄弟。如果这是一棵真正的树的话，这个 <html> 标签显然就是树根。

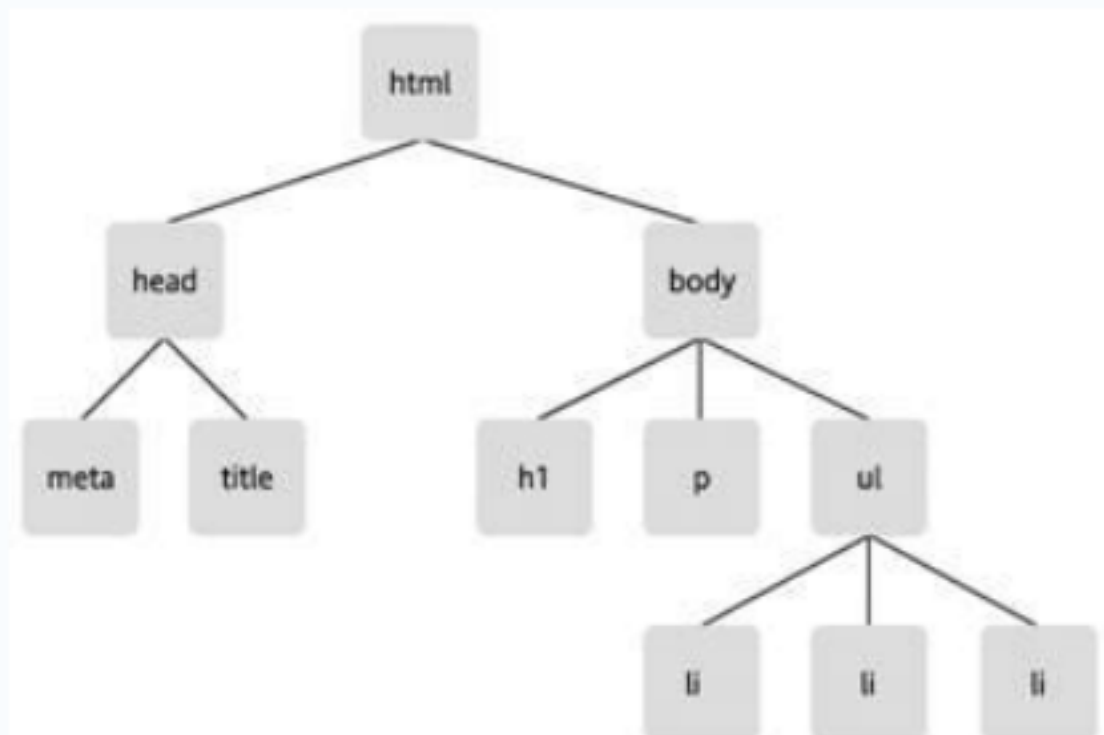


图 3-1 把网页中的元素表示为一棵家谱树

这正是图 3-1 中的家谱树以 html 为根元素的原因。毫无疑问，html 元素完全可以代表整个文档。

如果在这份文档里更深入一层，我们将发现 <head>和<body>两个分枝。它们存在于同一层次且互不包含，所以它们是兄弟关系。它们有着共同的父元素 <html>，但又各有各的子元素，所以它们本身又是其他一些元素的父元素。

<head>元素有两个子元素： <meta>和<title> （这两个元素是兄弟关系）。 <body>元素有三个子元素： <h1>、<p>和 （这三个元素是兄弟关系）。如果继续深入下去，我们将发现 <ul> 也是一个父元素。它有三个子元素，它们都是 <li> 元素。

利用这种简单的家谱关系记号，我们可以把各元素之间的关系简明清晰地表达出来。

例如， <h1>和 之间是什么关系？答案是它们是兄弟关系。

那么 <body>和 之间又是什么关系？ <body>是 的父元素， 是<body>的一个子元素。

如果把各种文档元素想像成一棵家谱树上的各个节点的话，我们就可以用同样的记号来描述 DOM。不过，与使用“ 家谱树 ”这个术语相比，把一份文档称为一棵“ 节点树 ”更准确。

3.3.1 节点

节点（node）这个名词来自网络理论，它代表着网络中的一个连接点。网络是由节点构成的集合。

在现实世界里，一切事物都由原子构成。原子是现实世界的节点。但原子本身还可以进一步分解为更细小的亚原子微粒。这些亚原子微粒同样是节点。

DOM也是同样的情况。文档也是由节点构成的集合，只不过此时的节点是文档树上的树枝和树叶而已。

在 DOM里存在着许多不同类型的节点。就像原子包含着亚原子微粒那样，有些 DOM节点类型还包含着其他类型的节点。

1. 元素节点

DOM的原子是元素节点（ element node ）。

在描述刚才那份“ 购物清单 ”文档时，我们使用了诸如 <body>、<p>和 之类的元素。如果把 Web上的文档比作一座大厦， 元素就是建造这座大厦的砖块，这些元素在文档中的布局形成了文档的结构。

各种标签提供了元素的名字。文本段落元素的名字是“ p”，无序清单元素的名字是“ ul ”，列表项元素的名字是“ li ”。

元素可以包含其他的元素。在我们的“ 购物清单 ”文档里，所有的列表项元素都包含在一个无序清单元素的内部。 事实上，没有被包含在其他元素里的唯一元素是 <html> 元素。它是我们的节点树的根元素。

2．文本节点

元素只是不同节点类型中的一种。如果一份文档完全由一些空白元素构成，它将有一个结构， 但这份文档本身将不会包含什么内容。 在网上， 内容决定着一切，没有内容的文档是没有任何价值的，而绝大多数内容都是由文本提供的。

在“ 购物清单 ”例子里， <p>元素包含着文本“ Don’t forget to buy this stuff. ”。它是一个文本节点（ text node ）。

在 XHTML文档里，文本节点总是被包含在元素节点的内部。但并非所有的元素节点都包含有文本节点。 在“ 购物清单 ”文档里， 元素没有直接包含任何文本节点——它包含着其他的元素节点 （一些 <li> 元素），后者包含着文本节点。

3．属性节点

还存在着其他一些节点类型。例如，注释就是另外一种节点类型。但我们这里还想向大家再多介绍一种节点类型。

元素都或多或少地有一些属性，属性的作用是对元素做出更具体的描述。例如，几乎所有的元素都有一个 title 属性，而我们可以利用这个属性对包含在元素里的东西做出准确的描述：

```
<p title="a gentle reminder">Don’t forget to buy this stuff.</p>
```

在 DOM中， title="a gentle reminder" 是一个属性节点（ attribute node ），如图 3-2 所示。因为属性总是被放在起始标签里， 所以属性节点总是被包含在元素节点当中。

并非所有的元素都包含着属性，但所有的属性都会被包含在元素里。

图 3-2　一个元素节点，它包含着一个属性节点和一个文本节点

在前面的“购物清单”示例文档里，我们可以清楚地看到那个无序清单元素（<ul>）有个 id 属性。如果曾经使用过 CSS，你们对 id 和 class 之类的属性应该不会感到陌生。 不过，为了照顾那些对 CSS还不太熟悉的读者， 我们下面将简要地重温几个最基本的 CSS概念。

4. CSS：层叠样式表

DOM并不是人们与网页的结构打交道的唯一手段。我们还可以通过 CSS（层叠样式表）告诉浏览器应该如何显示一份文档的内容。

类似于 JavaScript 脚本，对样式的声明既可以嵌在文档的 <head>部分（这需要用到 <style> 标签），也可以放在另外一个样式表文件里。利用 CSS对某个元素的样式做出声明的语法与 JavaScript 函数的定义语法很相似：

```
selector {
 property: value;
}
```

在样式声明里， 我们可以对浏览器在显示各有关元素时使用的颜色、 字体和字号做出定义，如下所示：

```
p {
  color: yellow;
  font-family: "arial", sans-serif;
  font-size: 1.2em;
}
```

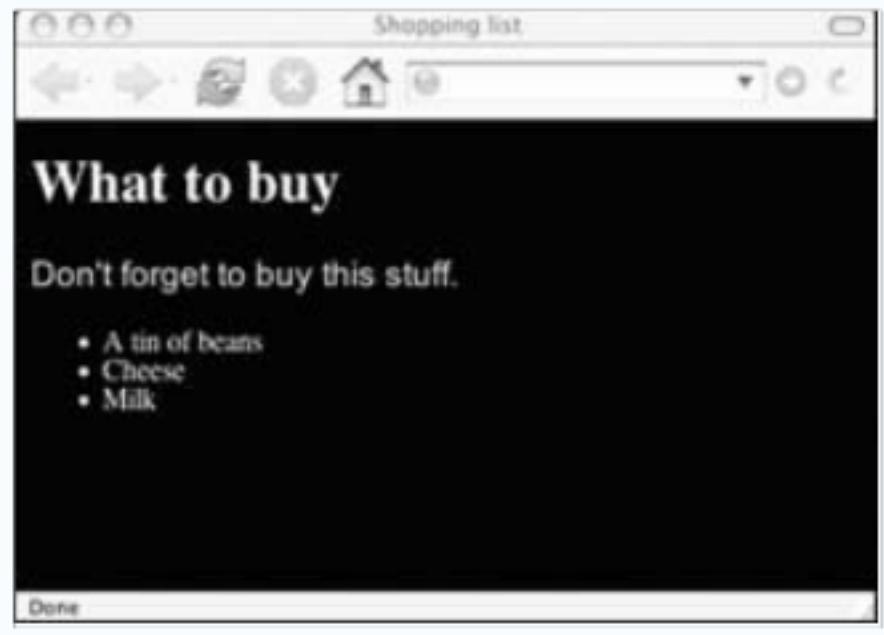
继承（inheritance ）是 CSS技术中的一项强大功能。类似于 DOM，CSS也把文档的内容视为一棵节点树。节点树上的各个元素将继承其父元素的样式属性。

例如，如果我们为 body 元素定义了一些颜色或字体，包含在 body 元素里的所有元素都将自动获得——也就是继承，那些样式：

```
body {
  color: white;
  background-color: black;
}
```

这些颜色将不仅作用于那些被直接包含在 <body>标签里的内容，还将作用于那些嵌套在 body 元素内部的所有元素。

这里是把刚才定义的样式应用在“购物清单”示例文档上而得到的网页显示效果。



在某些场合，当把样式应用于一份文档时，我们其实只想让那些样式只作用于某个特定的元素。例如，我们只想让某一段文本变成某种特殊的颜色和字体，但不想让其他段落受到影响。 为了获得如此精细的控制， 我们将需要在文档里插入一些能够把这段文本与其他段落区别开来的特殊标志。

为了把某一个或某几个元素与其他元素区别开来，我们需要使用 class 属性或 id 属性之一。

l class 属性

所有的元素都有 class 属性，不同的元素可以有同样的 class 属性值。如下所示：

```
<p class="special">This paragraph has the special class</p>
<h2 class="special">So does this headline</h2>
```

在样式表里，我们可以像下面这样为 class 属性值相同的所有元素定义一种共享的样式：

```
.special {
  font-style: italic;
}
```

我们还可以像下面这样利用 class 属性为一种特定类型的元素定义一种独享的样式：

```
h2.special {
  text-transform: uppercase;
}
```

l id 属性

id 属性的用途是给网页里的某个元素加上一个独一无二的标识符，如下所示：

```
<ul id="purchases">
```

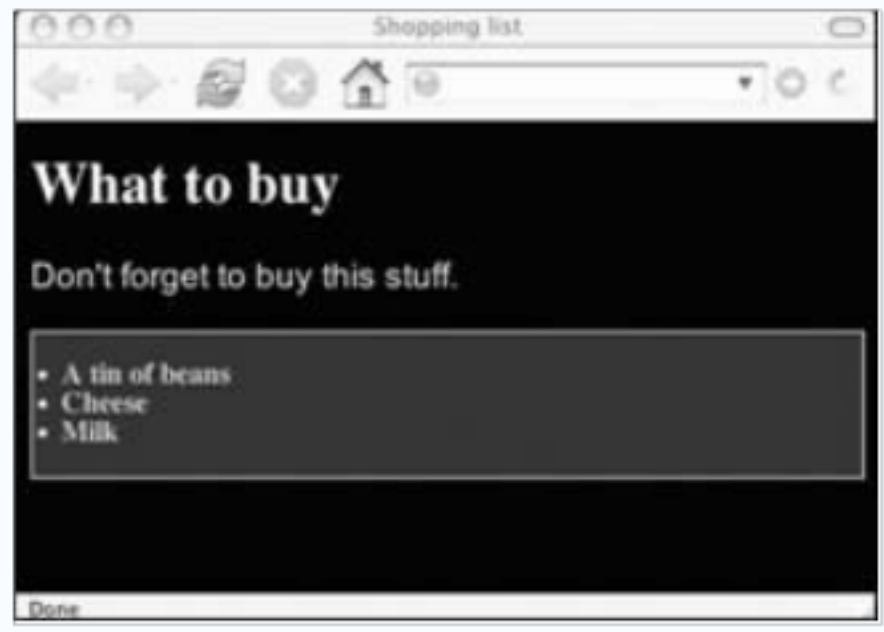
在样式表里，我们可以像下面这样为有着特定 id 属性值的元素定义一种独享的样式：

```
#purchases {
  border: 1px solid white;
  background-color: #333;
  color: #ccc;
  padding: 1em;
}
```

每个元素只能有一个 id 属性值，不同的元素必须有不同的 id 属性值。不过，我们可以在样式表里像下面这样，利用 id 属性为包含在某给定元素里的其他元素定义样式，而如此定义出来的样式将只作用于包含在这个给定元素里的有关元素：

```
#purchases li {
  font-weight: bold;
}
```

这里是把刚才利用 id 属性定义的样式应用在“购物清单”示例文档上而得到的网页显示效果。



id 属性就像是一个挂钩， 它一头连着文档里的某个元素， 另一头连着 CSS样式表里的某个样式。 DOM也可以使用这种挂钩，事实上，有不少 DOM操作必须借助于这种挂钩才能完成。

3.3.2 getElementById() 方法

DOM提供了一个名为 getElementById() 的方法，这个方法将返回一个与那个有着给定 id 属性值的元素节点相对应的对象。请注意， JavaScript 语言区分字母的大小写情况， 所以大家在写出“ getElementById ”时千万不要把大小写弄错了。如果把它错写成“ GetElementById ”或“ getElementbyid ”， 你将无法得到你真正想要的东西。

这个方法是与 document对象相关联的函数 。在脚本代码里，函数名的后面必须跟有一组圆括号， 这组圆括号包含着函数的参数。 getElementById() 方法只有一个参数： 你想获得的那个元素的 id 属性值，这个 id 值必须放在单引号或双引号里。

```
document.getElementById(id)
```

下面是一个例子：

```
document.getElementById("purchases")
```

这个调用将返回一个对象，这个对象对应着 document对象里的一个独一无二的元素，那个元素的 HTML id 属性值是 purchases 。再说一遍， getElementById()

方法将返回一个对象。你们可以用 typeof 操作符来验证这一点。 typeof 操作符可以告诉我们它的操作数是不是一个字符串、数值、函数、布尔值或对象 。

下面是把一些 JavaScript 语句插入到前面给出的“购物清单”文档之后得到的一份代码清单，新增的代码（黑体字部分）出现在 </body>结束标签之前。顺便说一句，我本人并不赞成把 JavaScript 代码直接嵌入一份文档的做法，但它不失为一种简便快捷的测试手段：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
➥ charset=utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li>Cheese</li>
      <li>Milk</li>
    </ul>
<script type="text/javascript">
alert(typeof document.getElementById("purchases"));
</script>
  </body>
</html>
```

把上面这些代码保存为一个 XHTML文件。当在你们的 Web浏览器里加载这个 XHTML文件时，屏幕上将弹出一个如下所示的 alter 对话框，它向你们报告 document.getElementById ("purchases") 的类型——它是一个对象。不仅如此，如果用上述办法去检查其他元素节点的类型，你们也会看到类似的 alert 对话框。

事实上，文档中的每一个元素都对应着一个对象。利用 DOM提供的方法，我们可以把与这些元素相对应的任何一个对象筛选出来。

一般来说，我们用不着为文档里的每一个元素都分别定义一个独一无二的 id 值；那也太小题大做了。要知道，即使某个元素没有独一无二的 id 值，我们也可以利用 DOM提供的另一个方法把与之对应的对象准确无误地筛选出来。

3.3.3 getElementsByTagName() 方法

getElementsByTagName()方法将返回一个对象数组，每个对象分别对应着文档里有着给定标签的一个元素。类似于 getElementById() ，这个方法也是只有一个参数的函数，它的参数是 (X)HTML标签的名字：

```
element.getElementsByTagName(tag)
```

它与 getElementById() 方法有许多相似之处，但有一点要特别提醒大家注意： getElements- ByTagName() 方法返回的是一个数组；你们在编写脚本时千万注意不要把这两个方法弄混了。

下面是一个例子：

```
document.getElementsByTagName("li")
```

这个调用将返回一个对象数组，每个对象分别对应着 document对象中的一个列表项（ li ）元素。与任何其他的数组一样，我们可以利用 length 属性查出这个数组里的元素个数。

首先，在上一小节给出的 XHTML示例文档里把 <script> 标签中的 alter 语句替换为下面这条语句：

```
alert(document.getElementsByTagName("li").length);
```

然后，用浏览器里重新加载那个 XHTML文件，你们就会看到这份示例文档里的列表项元素的个数： 3。这个数组里的每个元素都是一个对象。可以通过利用一个循环语句和 typeof 操作符去遍历这个数组的办法来验证这一点。例如，你们可以试试下面这个 for 循环：

```
for (var i=0; i < document.getElementsByTagName("li").length; i++) {
  alert(typeof document.getElementsByTagName("li")[i]);
}
```

请注意，即使在整个文档里只有一个元素有着给定的标签名，getElementsByTagName()方法也将返回一个数组。此时，那个数组的长度是 1。

你们或许早就觉得用键盘反复敲入 document.getElementsByTagName("li")是件很麻烦的事情，而这些长长的字符串会让代码变得越来越难以阅读。有个简单的办法可以减少不必要的打字量并改善代码的可读性：只要把 document.getElementsByTagName("li") 赋值给一个变量即可。

请在上一小节给出的 XHTML 示例文档里把 <script> 标签中的 alter 语句替换为下面这些语句：

```
var items = document.getElementsByTagName("li");
for (var i=0; i < items.length; i++) {
  alert(typeof items[i]);
}
```

现在，当用浏览器里重新加载那个 XHTML 文件时，你们将看到三个 alert 对话框，显示在这三个 alert 对话框里的消息都是“ object ”。

getElementsByTagName()方法允许我们把一个通配符当为它的参数，而这意味着文档里的每个元素都将在这个函数所返回的数组里占有一席之地。通配符（星号字符“ * ”）必须放在引号里，这是为了让通配符与乘法操作符有所区别。如果你想知道某份文档里总共有多少个元素节点，像下面这样以通配符为参数去调用 getElementsByTagName()方法是最简便的办法：

```
alert(document.getElementsByTagName("*").length);
```

我们还可以把 getElementById() 和 getElementsByTagName()方法结合起来运用。例如，我们刚才给出的几个例子都是通过 document 对象调用 getElementsByTagName()方法的，如果只想知 道其 id 属性值是 purchase 的元素包含着多少个列表项 （li ）的话，你就必须通过一个更具体的对象去调用这个方法，如下所示：

```
var shopping = document.getElementById("purchases");
var items = shopping.getElementsByTagName("*");
```

在这两条语句执行完毕后，items 数组将只包含其 id 属性值是 purchase 的无序清单里的元素。具体到我们这个例子，items 数组的长度刚好与这份文档里的列表项元素的总数相等：

```
alert (items.length);
```

如果还需要更多的证据，下面这些语句将证明 items 数组里的每个值确实是一个对象：

```
for (var i=0; i < items.length; i++) {
  alert(typeof items[i]);
}
```

3.4 趁热打铁

在看过那么多显示着单词“ object ”的 alert 对话框后，你们很可能不愿意再看到它。如果真是如此，我的目的也就达到了——我想通过这些 alert 对话框强调这样一个事实： 文档中的每个元素节点都是一个对象。 我现在要告诉大家的是，这些对象中的每个都为我们准备了一系列非常有用的方法， 而这一切当然都要归功于 DOM。利用这些已经预先定义好的方法，我们不仅可以检索出关于文档里的任何一个对象的信息，甚至还可以改变某些元素的属性。

下面是对本章此前学习内容的一个简要总结：

- 一份文档就是一棵节点树。
- 节点分为不同的类型：元素节点、属性节点和文本节点等。
- getElementById() 方法将返回一个对象，该对象对应着文档里的一个特定的元素节点。
- getElementsByTagName() 方法将返回一个对象数组， 它们分别对应着文档里的一个特定的元素节点。
- 这些节点中的每个都是一个对象。

接下来，我们将向大家介绍几个与这些对象相关联的属性和方法。

3.4.1 getAttribute() 方法

至此，我们已经向大家介绍了两种检索特定元素节点的办法：一种是使用 getElementById() 方法，另一种是使用 getElementsByTagName()方法。在找到那个元素后， 我们就可以利用 getAttribute() 方法把它的各种属性的值查询出来。

getAttribute() 方法是一个函数。它只有一个参数——你打算查询的属性的名字：

```
object.getAttribute(attribute)
```

不过， getAttribute() 方法不能通过 document对象调用，这与我们此前介绍过的其他方法不同。我们只能通过一个元素节点对象调用它。

例如，你可以把它与 getElementsByTagName()方法结合起来，去查询每个 <p>元素的 title 属性，如下所示：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i < paras.length; i++ ) {
  alert(paras[i].getAttribute("title"));
}
```

如果把上面这段代码插入到前面给出的“ 购物清单 ”示例文档的末尾，并在 Web 浏览器里重新加载这个页面，屏幕上将弹出一个显示着文本消息“ a gentle reminder ”的 alter 对话框。

在“ 购物清单 ”文档里只有一个带有 title 属性的 <p>元素。假如这份文档还有一个或更多个不带 title 属性的 <p>元素, 则相应的 getAttribute("title") 调用将返回 null 。null 是 JavaScript 语言中的空值， 其含义是“ 你说的这个东西不存在 ”。如果你们想亲自验证一下这件事， 请先把下面这段文本插入到 “ 购物清单 ”文档中的现有文本段落之后：

```
<p>This is just a test</p>
```

然后重新加载这个页面。 这一次， 你们将看到两个 alter 对话框， 而第二个对话框将是一片空白或者是只显示着单词“ null ”——具体情况要取决于你的 Web 浏览器将如何显示 null 值。

可以修改我们的脚本，让它只在 title 属性存在时才弹出一条消息。我们将增加一条 if 语句来检查 getAttribute() 方法的返回值是不是 null 。趁着这个机会，我们还增加了几个变量以提高脚本的可读性：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text != null) {
    alert(title_text);
  }
}
```

现在，如果重新加载这个页面，你们将只会看到一个显示着“ a gentle reminder ”消息的 alter 对话框，如下所示。

我们甚至可以把这段代码缩得更短一些。当检查某项数据是否是 null 值时，我们其实是在检查它是否存在。 这种检查可以简化为直接把被检查的数据用做 if 语句的条件。 if (something) 与 if (something != null) 完全等价， 但前者显然更为简明。此时，如果 something 存在，则 if 语句的条件将为真；如果 something 不存在，则 if 语句的条件将为假。

具体到这个例子，只要我们把 if (title_text != null) 替换为 if (title_text) ，我们就可以得到更简明的代码。此外，为了进一步增加代码的可读性，

我们还可以趁此机会把 alter 语句与 if 语句写在同一行上，这可以让它们更接近于我们日常生活中的英语句子：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text) alert(title_text);
}
```

3.4.2 setAttribute() 方法

我们此前介绍给大家的所有方法都只能用来检索信息。 setAttribute() 方法与它们有一个本质上的区别：它允许我们对属性节点的值做出修改。

类似于 getAttribute() 方法，setAttribute() 方法也是一个只能通过元素节点对象调用的函数，但 setAttribute() 方法需要我们向它传递两个参数：

```
object.setAttribute(attribute,value)
```

在下面的例子里， 第一条语句将把 id 属性值是 purchase 的元素检索出来， 第二条语句将把这个元素的 title 属性值设置为 a list of goods ：

```
var shopping = document.getElementById("purchases");
shopping.setAttribute("title","a list of goods");
```

我们可以利用 getAttribute() 方法来证明这个元素的 title 属性值确实发生了变化：

```
var shopping = document.getElementById("purchases");
alert(shopping.getAttribute("title"));
shopping.setAttribute("title","a list of goods");
alert(shopping.getAttribute("title"));
```

上面这些语句将在屏幕上弹出两个 alert 对话框：第一个 alter 对话框出现在 setAttribute() 方法被调用之前，它将是一片空白或显示着单词“ null ”；第二个出现在 title 属性值被设置之后，它将显示着“ a list of goods ”消息。

在上例中，我们设置了一个现有节点的 title 属性，但这个属性原先并不存在。这意味着我们发出的 setAttribute() 调用实际完成了两项操作：先把这个属性创建出来，然后再对其值进行设置。如果我们把 setAttribute() 方法用在元素节点的某个现有属性上，这个属性的当前值将被覆盖。

在“ 购物清单 ”示例文档里， <p>元素已经有了一个 title 属性，这个属性的值是 a gentle reminder 。我们可以用 setAttribute() 方法来改变它的当前值：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text) {
    paras[i].setAttribute("title","brand new title text");
    alert(paras[i].getAttribute("title"));
  }
}
```

上面这段代码将先从文档里把已经带有 title 属性的 <p>元素全部检索出来，然后把它们的 title 属性值全部修改为 brand new title text 。具体到“购物清单”文档，属性值 a gentle reminder 将被覆盖。

这里有一个非常值得关注的细节： 通过 setAttribute() 方法对文档做出的修改，将使得文档在浏览器窗口里的显示效果和 / 或行为动作发生相应的变化，但我们在通过浏览器的 view source （查看源代码）选项去查看文档的源代码时看到的仍将是原来的属性值——也就是说， setAttribute() 方法做出的修改不会反映在文档本身的源代码里。 这种“表里不一”的现象源自 DOM 的工作模式： 先加载文档的静态内容、 再以动态方式对它们进行刷新， 动态刷新不影响文档的静态内容。这正是 DOM 的真正威力和诱人之处： 对页面内容的刷新不需要最终用户在他们的浏览器里执行页面刷新操作就可以实现。

3.5 小结

在这一章里，我们向大家介绍了 DOM 提供的四个方法：

- getElementById() 方法
- getElementsByTagName() 方法
- getAttribute() 方法
- setAttribute() 方法

这四个方法是将要编写的许多 DOM 脚本的基石。

DOM 还提供了许多其他的属性和方法，如 nodeName、nodeValue、chileNodes、nextSibling 和 parentNode 等，但现在还不是它们出场的时候——我打算在本书后面的有关章节中选择一些适当的时机把它们依次介绍给大家。 我在这里列出它们的目的只是为了激起大家的学习兴趣。

本章内容比较偏重于理论。在看过那么多的 alter 对话框之后，相信大家都迫不及待地想通过其他一些东西去进一步了解和测试 DOM。而我也正想通过一个案例来进一步展示 DOM 的强大威力。

在下一章中，我将带领大家利用本章介绍的四个 DOM 方法去创建一个基于 JavaScript 的图片库。

现在，是时候利用 DOM 去完成一些简单的任务了。在这一章中，我将带领大家用 JavaScript 和 DOM 去建立一个图片库， 并将其称为“ JavaScript 美术馆 ”。我们将按照以下步骤来完成这一案例：

- 编写一份优秀的标记语言文档。
- 编写一个 JavaScript 函数以显示用户想要查看的图片。
- 在标记语言文档里触发一个调用 JavaScript 图片显示的函数。
- 对这个 JavaScript 函数的功能进行扩展。这需要用到几个新的 DOM 方法。

把一些图片发布到网上的办法很多。最容易想到的办法是把所有的图片都放到同一个网页里。 不过，如果你打算发布的图片比较多的话， 这个页面很快就会因为变得过于巨大而失去吸引力。要知道，虽然这种网页本身的 HTML 代码不会多到哪儿去， 但算上那些图片可就不一样了。 我们必须面对的现实是： 数据量越大，网页的下载时间就越长， 但愿意等待很长时间去下载一个网页的人却没有几个。

因此，为每张图片分别创建一个网页的解决方案显然更值得考虑。你的图片库将不再是一个体积庞大、 难以下载的网页， 而是许多尺寸合理、 便于下载和浏览的页面。 不过，这一解决方案并非尽善尽美： 首先， 为每张图片分别制作一个网页需要花费不少的时间和精力； 其次， 需要在每个网页上提供一些链接， 来给出当前图片在整个图片库里的位置以方便人们从当前图片转到其他的图片。

如果你想两全其美，利用 JavaScript 来创建图片库将是最佳的选择：把整个图片库的浏览链接集中安排在你的图片库主页里， 只在用户点击了这个主页里的某个图片链接时才把相应的图片传送给他。

4.1 编写标记语言文档

为了完成“ JavaScript 美术馆”案例，我特意用数码相机拍摄了几张照片，并把它们修整成最适合于用浏览器来查看的尺寸，即 400 像素宽× 300 像素高。

第一项工作是为这些图片创建一个链接清单。因为我没打算让这些图片按照某种特殊的顺序排列，所以将使用一个无序清单元素（ ）来列出那些链接。如果想让自己的图片按顺序排列的话， 应该使用一个排序清单元素 （<ol>）去组织图片链接。

下面就是我编写出来的 HTML文档：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Image Gallery</title>
</head>
<body>
  <h1>Snapshots</h1>
  <ul>
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display">
➥ Fireworks</a>
    </li>
    <li>
      <a href="images/coffee.jpg" title="A cup of black coffee">
➥ Coffee</a>
    </li>
    <li>
      <a href="images/rose.jpg" title="A red, red rose">Rose</a>
    </li>
    <li>
      <a href="images/bigben.jpg" title="The famous clock">
➥ Big Ben</a>
    </li>
  </ul>
</body>
</html>
```

我将把这份文档保存为 gallery.html 文件，并把图片集中保存在子目录 images 里。我的 images 子目录和 gallery.html 文件位于同一个子目录下。在 gallery.html 文件里，无序清单元素中的每个链接分别指向不同的图片。在浏览器窗口里点击某个链接就可以转到相应的图片， 但从图片重新返回到链接清单目前还必须借助于浏览器的 Back（后退）按钮。下面是这个基本的链接清单在浏览器窗口里的显示效果。

这是一个相当令人满意的网页，但它的默认行为还不太理想。下面是我觉得还需要改进的几个地方：

- 当点击某个链接时，我希望能留在这个网页而不是转到另一个窗口。
- 当点击某个链接时，我希望能在这个网页上同时看到那张图片以及原有的图片清单。

下面是我为了实现上述希望而需要完成的几项改进：

- 通过增加一个“占位符”图片的办法在这个主页上为图片预留一个浏览区域。
- 在点击某个链接时，将拦截这个网页的默认行为。
- 在点击某个链接时，将把“占位符”图片替换为与那个链接相对应的图片。

我们先来解决“占位符”图片的问题。我选用了一个类似于名片的图片，你们可以根据个人喜好来决定选用的图片，即使选用一个空白图片也没问题。

把下面这些代码插入到图片名单的末尾：

```
<img id="placeholder" src="images/placeholder.gif"
➥ alt="my image gallery" />
```

我对这个图片的 id 属性进行了设置，这将使我可以通过一个外部的样式表对图片的显示位置和显示效果加以控制。例如，我可以让这个图片出现在链接清单的旁边而不是它的下方。我还可以在自己的 JavaScript 代码里使用这个 id 值。下面是这个页面在增加了“占位符”图片后的显示效果。

现在，我的 HTML文档已经准备好了。接下来的工作是编写一些必要的 Java Script 代码。

4.2 编写 JavaScript 函数

为了把“占位符”图片替换为想要查看的图片，需要改变它的 src 属性。setAttribute() 方法是完成这项工作的最佳选择，而我将利用这个方法写一个函数。那个函数只有一个参数， 即我想查看的那张图片的链接； 它将通过改变“占位符”图片的 src 属性的办法将其替换为我想查看的那张图片。

首先，需要给函数起一个好的名字。我想让它既能提醒我这个函数的用途，又足够简明扼要。我决定把这个函数命名为 showPic() 。还需要给这个函数的参数起一个名字，我决定把它命名为 whichpic ：

```
function showPic(whichpic)
```

whichpic 代表着一个元素节点； 具体地说，那将是一个指向某个图片的 <a>元素。我需要知道那个图片的文件路径， 这个路径可以通过在 whichpic 元素上调用 getAttribute() 方法的办法查出来——只要把 "href" 作为参数传递给 getAttribute() 方法，就可以知道那个图片的文件路径了：

```
whichpic.getAttribute("href")
```

我将把这个路径存入一个变量以便在后面的语句里使用它。 我决定把这个变量命名为 source ：

```
var source = whichpic.getAttribute("href");
```

接下来，还需要把“ 占位符 ”图片检索出来，这种事对 getElementById() 方法来说不过是小菜一碟：

```
document.getElementById("placeholder")
```

我不想重复敲入“ document. getElementById("placerholder") ”这么长的字符串，所以将把这个元素赋值给一个变量并将其命名为 placerholder ：

```
var placeholder = document.getElementById("placeholder");
```

现在，已经声明并赋值了两个变量： source 和 placerholder 。它们可以让我的脚本简明易读。

我将使用 setAttribute() 方法对 placerholder 元素的 src 属性进行刷新。还记得吗， 这个方法有两个参数： 一个是打算对之进行设置的属性， 另一个是这个属性的新属性值。 具体到这个例子， 因为我想对 src 属性进行设置， 所以第一个参数是 "src" ；至于第二个参数，也就是 src 属性的新属性值，我已经把它保存在 source 变量里了：

```
placeholder.setAttribute("src",source);
```

这显然要比下面这么冗长的代码更容易阅读和理解：

```
document.getElementById("placeholder").setAttribute("src",
➥ whichpic.getAttribute("href"));
```

4.2.1 DOM 之前的解决方案

其实，不使用 setAttribute() 方法也可以改变某个图片的 src 属性。

setAttribute() 方法是“ 第 1 级 DOM ”（DOM Level 1）的组成部分之一，这个方法可以对任意元素节点的任意属性进行设置。在“ 第 1 级 DOM ”出现之前，

程序员只能通过另外一种办法对一部分元素的属性进行设置，　　这个办法至少在目前还可以用来改变某些属性的值。

例如，如果想改变某个输入元素的　value 属性，可以使用如下所示的办法：

```
element.value = "the new value"
```

这与下面这条语句的效果是等价的：

```
element.setAttribute("value","the new value");
```

类似的办法也可以用来改变图片的　src 属性。例如，在我的图片库脚本里，　完全可以用下面这条语句来代替　setAttribute()　　调用：

```
placeholder.src = source;
```

我个人更喜欢使用　setAttribute()　　方法。这起码可以让我不必费心去记忆哪些元素的哪些属性可以用哪些　DOM之前的方法去设置。 虽然用那些老办法可以毫无问题地对文档里的图片、表单和其他一些元素的属性进行设置，但　　setAttribute() 方法的优势在于它可以对文档中的任何一个元素的任何一个属性做出修改。

“第 1 级 DOM”的另一个优势是可移植性更好。那些老方法只适用于　　Web文档，　DOM则适用于任何一种标记语言。虽然这种差异对我们这个例子没有影响，但我希望大家能够牢牢记住这一点：　DOM是一种适用于多种环境和多种程序设计语言的通用型　API。如果想把从本书学到的　DOM技巧运用在　Web浏览器以外的应用环境里，严格遵守“第　1 级 DOM”能够让你们避免与兼容性有关的任何问题。

4.2.2　showPic()　函数的代码清单

下面是 showPic() 函数完整的代码清单：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
}
```

接下来的任务是把这个　JavaScript　函数与我们的　HTML文档结合起来。

4.3　JavaScript 函数的调用

我需要把刚编写出来 showPic() 函数与图片库 HTML文档结合起来。最简单的办法是把这个函数用一组 <script> 标签插入到那个文档的 <head>部分，但我认为这种做法有点儿目光短浅： 如果今后想把这同一函数用在多个页面上的话， 我将不得不反复多次地进行剪贴操作。 为今后考虑， 更有远见的办法是先把这个函数存入一个外部文件，然后在每一份需要用到这个函数的 HTML文档的 <head>部分插入一个链接来引用这个外部文件。

以.js 作为文件扩展名，把这个函数存入一个文本文件。完全可以把这种文件命名为你们喜欢的任何东西， 但我习惯于用这些文件所包含的函数的名字来命名它们——我给这个文件起的名字是 showPic.js 。

就像我刚才决定把所有的图片集中存放在 images 子目录里那样， 把所有的 JavaScript 脚本文件集中存放在一个子目录里也将是个好主意。 我创建了一个名为 scripts 的子目录并把 showPic.js 文件保存到其中。

现在，需要在图片库 HTML文档里插入一个链接来引用这个 JavaScript 脚本文件。我将把下面这条语句插入那份 HTML文档的 <head>部分（选择的插入位置是<head>标签之后）：

```
<script type="text/javascript" src="scripts/showPic.js"></script>
```

有了这条语句，把 showPic() 函数与图片库 HTML文档结合起来的任务就已经完成了一半——我还需要为 HTML文档里的每个图片链接增加一个函数调用动作，否则 showPic() 函数将永远也不会被调用。我将通过增加一个事件处理函数（ event handler ）的办法来完成这项工作。

事件处理函数

事件处理函数的作用是，在预定事件发生时让预先安排好的 JavaScript 代码开始执行，用术语来说就是 “ 触发一个动作 ”。例如，如果想在鼠标指针悬停在某个元素上时触发一个动作， 就需要使用 onmouseover事件处理函数； 如果想在鼠标指针离开某个元素时触发一个动作，就需要使用 onmouseout事件处理

函数。在“ JavaScript 美术馆”案例里，我想在用户点击某个链接时触发一个动作，所以需要使用 onclick 事件处理函数。

我想通过 onclick 事件处理函数去触发的动作是调用 showPic() 函数，而要想调用这个函数，就必须向它传递一个参数：一个带有 href 属性的元素节点。在图片库 HTML文档里，当我把 onclick 事件处理函数嵌入一些链接时，我需要把那些链接本身用作 showPic() 函数的参数。

有个非常简单但又非常有效的办法可以做到这一点：使用 JavaScript 语言中的 this 关键字。这个关键字的含义是“这个对象”。具体到正在讨论的这个例子，我将使用 this 来表示“这个 <a>元素节点”：

```
showPic(this)
```

综上所述，我将使用 onclick 事件处理函数去调用 showPic(this) 方法。使用事件处理函数调用 JavaScript 代码的语法如下所示：

```
event = "JavaScript statement(s)"
```

请注意，在如上所示的语法里，JavaScript 代码是包含在一对引号之间的：我们可以把任意数量的 JavaScript 语句放在这对引号之间，只要把各条语句用分号隔开即可。

下面这条语句将完成“使用 onclick 事件处理函数调用 showPic(this) 方法”的任务：

```
onclick = "showPic(this);"
```

不过，如果只把上面这个事件处理函数插入到 HTML文档中的链接里去的话，将遇到这样一个问题：在 onclick 事件发生时，不仅 showPic(this) 函数将被调用，链接被点击时的默认行为也将发生。这意味着用户还是会被带到另外一个图片查看窗口里去，而这是我不希望发生的事情。我需要阻止这种默认行为的发生。

要想达到这一目的，我们必须对事件处理函数的工作机制有进一步的了解：在给某个元素添加了事件处理函数后，一旦发生预定事件，相应的 JavaScript

代码就会得到执行；那些 JavaScript 代码可以返回一个结果，而这个结果将被传递回那个事件处理函数。 例如，我们可以给某个链接添加一个 onclick 事件处理函数，并让这个处理函数所触发的 JavaScript 代码返回布尔值 true 或 false 。这样一来，当这个链接被点击时，如果那段 JavaScript 代码返回给 onclick 事件处理函数的值是 true ， onclick 事件处理函数将认为“ 这个链接被点击了 ”；反之，如果那段 JavaScript 代码返回给 onclick 事件处理函数的值是 false ， onclick 事件处理函数将认为“ 这个链接没有被点击 ”。

可以通过下面这个简单测试去验证这一结论：

```
<a href="http://www.example.com" onclick="return false;">Click me</a>
```

当点击这个链接时，因为 onclick 事件处理函数所触发的 JavaScript 代码返回给它的值是 false ，所以这个链接在被点击时的默认行为将不会发生。

同样道理，如果像下面这样在 onclick 事件处理函数所触发的 JavaScript 代码里增加一条 return false 语句的话，我就可以不让用户被他们所点击的链接带到另外一个图片查看窗口里去：

```
onclick = "showPic(this); return false;"
```

下面是最终完成的 onclick 事件处理函数在图片库 HTML文档里的样子：

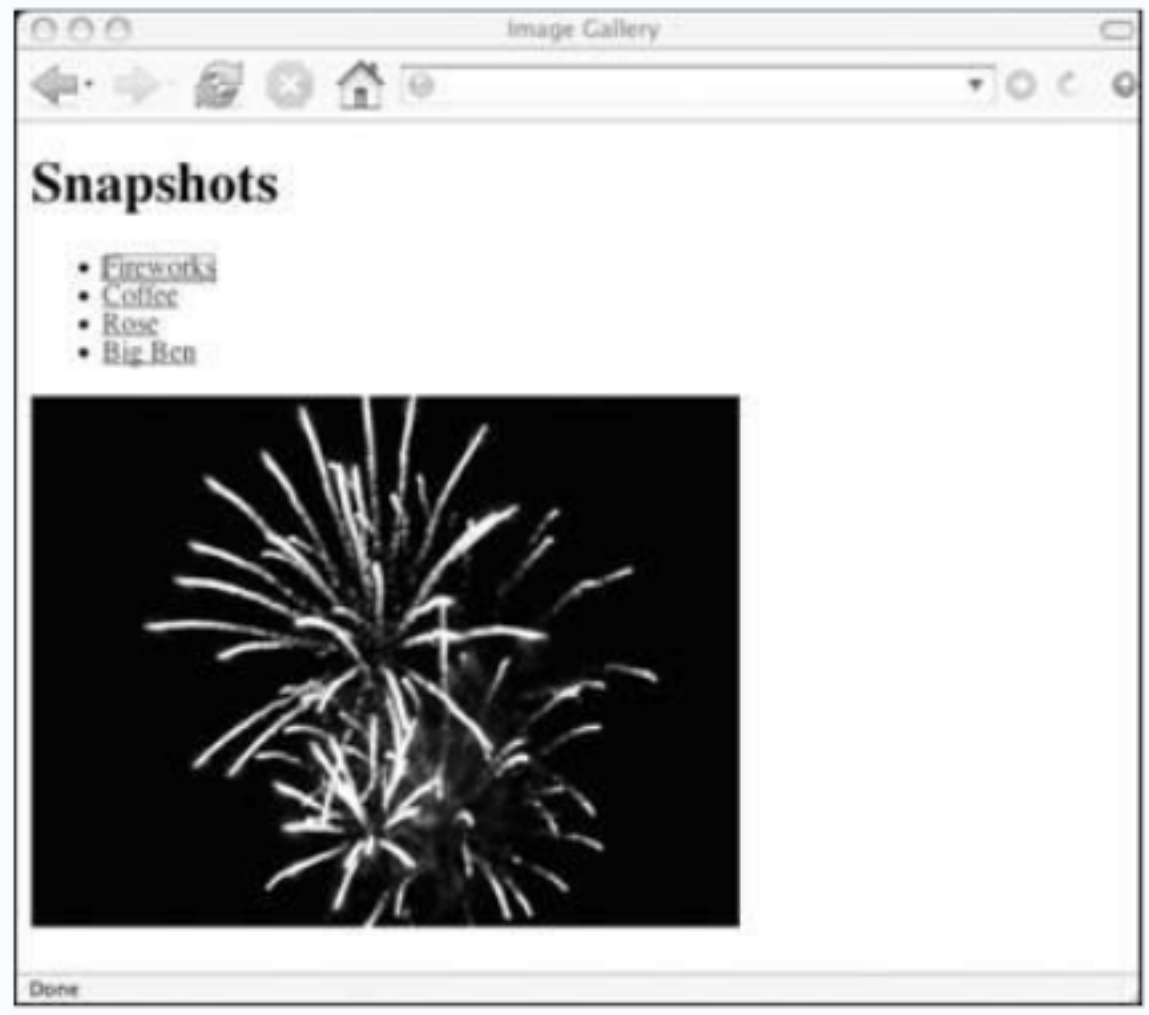
```
<li>
    <a href="images/fireworks.jpg" onclick="showPic(this);
➥ return false;" title="A fireworks display">Fireworks</a>
</li>
```

接下来，只要在图片库 HTML文档里把这个 onclick 事件处理函数添加到每个链接上即可。 这当然有些麻烦， 但眼下只能这么做——我将在后面的内容里向大家介绍一个可以避免这种麻烦的办法。下面是图片库 HTML文档在我以手动方式把这个 onclick 事件处理函数添加到各个有关链接上之后的样子：

```
<li>
    <a href="images/fireworks.jpg" onclick="showPic(this);
➥ return false;" title="A fireworks display">Fireworks</a>
  </li>
  <li>
    <a href="images/coffee.jpg" onclick="showPic(this);
➥ return false;" title="A cup of black coffee">Coffee</a>
```

```
  </li>
  <li>
    <a href="images/rose.jpg" onclick="showPic(this); return false;"
➥ title="A red, red rose">Rose</a>
  </li>
  <li>
    <a href="images/bigben.jpg" onclick="showPic(this); return false;"
➥ title="The famous clock">Big Ben</a>
  </li>
```

现在，如果把这个页面加载到 Web浏览器里，你们将看到一个能够正常工作的 “ JavaScript 美术馆 ”：在如下所示的页面里，不管在图片列表里的哪个链接上点击鼠标，都将在这同一个页面里看到相应的图片。



4.4 对 JavaScript 函数进行功能扩展

在同一个网页上切换显示不同的图片并不是什么新鲜事。有着这类效果的网页和脚本早在 W3C推出它们的标准化 DOM和 JavaScript 语言之前就已经出现了，如今更是得到了广泛的流行。

在这种情形下，如果你们想让自己与众不同，就必须另辟蹊径——你们有没有想过在同一个网页上切换显示不同的文本？我可不是在开玩笑，利用 JavaScript 语言和 DOM，确实可以做到这一点。

图片库 HTML文档里的每个图片链接都有一个 title 属性。我想把这个属性的值提取出来并让它们伴随相应的图片一同显示在网页上。 title 属性的值可以

用 getAttribute() 方法轻而易举地提取出来，如果把下面这条语句添加到 showPic() 函数的开头的话，就可以把 title 属性的值保存到一个变量里：

```
var titletext = whichPic.getAttribute("title");
```

解决了提取 title 属性值的问题，还需要把它插入到 HTML 文档中的适当位置。为了完成这一工作，我需要用到几个新的 DOM 属性。

4.4.1 childNodes 属性

childNodes 属性让我们可以从给定文档的节点树里把任何一个元素的所有子元素检索出来。

childNodes 属性将返回一个数组，这个数组包含给定元素节点的全体子元素：

```
element.childNodes
```

假设我们需要把某个文档的 body 元素的全体子元素检索出来。首先，我们将使用 getElementsByTagName()方法来得到 body 元素。因为每份文档只有一个 body 元素，所以它将是 getElementsByTagName("body") 方法所返回的数组中的第一个（也是唯一一个）元素：

```
var body_element = document.getElementsByTagName("body")[0];
```

现在，变量 body_element 已经指向了那个文档的 body 元素。接下来，可以用如下所示的语法记号把 body 元素的全体子元素检索出来：

```
body_element.childNodes
```

写出这个记号显然要比敲入下面这个长长的字符串要简明得多：

```
document.getElementsByTagName("body")[0].childNodes
```

顺便说一句，body 元素还有一个更简单的专用记号：

```
document.body
```

现在，已经知道如何把 body 元素的全体子元素检索出来了，我们来看看这些信息的用途。

首先，我们可以精确地查出 body 元素有多少个子元素。因为 childNodes 属性返回的是一个数组，所以可以用数组数据类型的 length 属性查出它所包含的元素的个数：

```
body_element.childNodes.length;
```

请把下面这个小函数添加到 showPic.js 文件里：

```
function countBodyChildren() {
  var body_element = document.getElementsByTagName("body")[0];
  alert (body_element.childNodes.length);
}
```

这个简单的小函数将弹出一个 alert 对话框，其显示内容是 body 元素的子元素的总数。

我想让这个函数在页面加载时执行，而这需要使用 onload 事件处理函数。把下面这条语句添加到代码段的末尾：

```
window.onload = countBodyChildren;
```

这条语句将在页面加载时调用 countBodyChildren 函数。

在 Web浏览器里刷新 gallery.html 文件。你们将看到一个 alert 对话框，其显示内容是 body 元素的子元素的总数。那个数字很可能会让你们大吃一惊。

4.4.2 nodeType 属性

根据 gallery.html 文件的结构，body 元素应该只有 3 个子元素：一个 h1 元素、一个 ul 元素和一个 imag 元素。可是，countBodyChildren() 函数给出来的数字却远大于此。之所以会这样，是因为文档树的节点类型并非只有元素节点一种。

由 childNodes 属性返回的数组包含着所有类型的节点，除了所有的元素节点，所有的属性节点和文本节点也包含在其中。事实上，文档里几乎每一样东西

都是一个节点——甚至连空格和换行符都会被解释为节点，　而它们也全都包含在 childNodes 属性所返回的数组当中。

因此， countBodyChildren() 函数的返回结果才会是一个相当大的数字。

还好，我们可以利用 nodeType属性来区分文档里的各个节点。这个属性可以让我们知道自己正在与哪一种节点打交道。　不过，这个属性返回的是一个数字而不是像“ element ”或“ attribute ”那样的英文字符串。

下面是 nodeType属性的调用语法：

```
node.nodeType
```

为了验证这一点， 请把 countBodyChildren() 函数中的 alter 语句替换为下面这条语句，这样一来，我们就可以知道 body_element 元素的 nodeType属性了：

```
alert(body_element.nodeType);
```

在 Web浏览器里刷新 gallery.html 文件，将看到一个显示数字“ 1 ”的 alert 对话框。换句话说，元素节点的 nodeType 属性值是 1。

nodeType属性总共有 12 种可取值，但其中仅有 3 种具有实用价值：元素节点、属性节点和文本节点的 nodeType 属性值分别是 1、2 和 3。

- 元素节点的 nodeType 属性值是 1。
- 属性节点的 nodeType 属性值是 2。
- 文本节点的 nodeType 属性值是 3。

这个意味着，可以让我们的函数只对某种特定类型的节点进行处理。例如，完全可以编写出一个只对元素节点进行处理的函数。

4.4.3 在 HTML 文档里增加一段描述性文本

为了让我的“ JavaScript 美术馆 ”变得与众不同， 我决定给它增加一个文本节点。我想在显示图片时把这个文本节点的值替换为一个来自某个属性节点 （某个图片链接的 title 属性）的值。

首先，需要为打算显示的文本安排显示位置。我将在 gallery.html 文件里增加一个新的文本段。我决定把它安排在 <img>标签之前。我将为它设置一个独一无二的 id 值，这样就能在 JavaScript 函数里方便地引用它了：

```
<p id="description">Choose an image.</p>
```

上面这条语句将把 <p>元素的 id 属性设置为 description （描述），这个单词可以让这个元素的用途一目了然。如下所示，包含在此元素里的文本现在是“ Choose an image. ”，但我打算在切换显示图片时把这段文本同时替换为相应的描述性文字。

我想达到的预期效果是： 在“ JavaScript 美术馆 ”里的某个图片链接被点击时，不仅要把“ 占位符 ”图片替换为那个链接的 href 属性所指向的图片，还要把这段文本同时替换为那个链接的 title 属性值。为了实现这一想法，需要对 showPic() 函数做一些改进。

4.4.4 用 JavaScript 代码改变 <p> 元素的文本内容

为了把“ JavaScript 美术馆 ”里的图片说明在某个图片链接被点击时， 动态地替换为那个链接的 title 属性值，我需要对 showPic() 函数做一些修改。

下面是 showPic() 函数现在的样子：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
}
```

首先，我需要在 showPic() 函数里增加一些语句来提取 whichpic 对象的 title 属性值。我将把提取到的 title 属性值存入 text 变量。这些事可以轻而易举地利用 getAttribute() 方法完成：

```
var text = whichpic.getAttribute("title");
```

接下来，为了让自己能在代码里方便地引用那段 id 属性值等于 description 的文本，我决定创建一个新的变量来存放它：

```
var description = document.getElementById("description");
```

下面是 showPic() 函数在我给它增加了上述两个变量之后的样子：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
}
```

我们的下一个任务是实现文本的切换显示效果。

4.4.5 nodeValue 属性

如果想改变某个文本节点的值，那就使用 DOM 提供的 nodeValue 属性，它的用途正是检索（和设置）节点的值：

```
node.nodeValue
```

但这里有个大家必须注意的细节：在用 nodeValue 属性检索 description 对象的值时，你得到的并不是包含在这个段落里的文本。可以用下面这条 alert 语句来验证这一点：

```
alert (description.nodeValue);
```

这个调用将返回一个 null 值。<p>元素的 nodeValue 属性值是一个空值， 而我们这里需要的是 <p>元素所包含的文本的值。

包含在 <p>元素里的文本是另一种节点，它在 DOM 里是 <p>元素的第一个子节点。换句话说，如果想获得 <p>元素的文本内容，就必须检索它的第一个子节点的 nodeValue 属性值。

下面这条 alert 语句可以找到我们想要的内容：

```
alert(description.childNodes[0].nodeValue);
```

这个调用的返回值才是我们正在寻找的“ Choose an image. ”。这个值来自 childNodes[] 数组的第一个（下标是 0）元素。

4.4.6 firstChild 和 lastChild 属性

数组元素 childNodes[0] 有个更直观易读的同义词。无论何时何地，只要需要访问 childNodes[] 数组的第一个元素，我们都可以把它写成 firstChild ，这个记号本身是一个属性：

```
node.firstChild
```

这种用法与下面这个语法记号完全等价：

```
node.childNodes[0]
```

单词“ firstChild ”不仅更加简短，还更加具有可读性。

DOM 还提供了一个与之对应的 lastChild 属性：

```
node.lastChild
```

这个语法记号代表着 childNodes[] 数组的最后一个元素。如果不想通过 lastChild 属性去访问这个节点，我们将不得不使用如下所示的语法记号：

```
node.childNodes[node.childNodes.length-1]
```

与简明易懂的 lastChild 相比，这么复杂的语法记号恐怕没人会喜欢。

4.4.7 利用 nodeValue 属性刷新 <p> 元素的文本内容

现在，我们将回到 showPic() 函数。我将对 id 属性值等于 description 的<p>元素所包含的文本节点的 nodeValue 属性进行刷新。

具体到这个 id 属性值等于 description 的<p>元素，因为它只有一个子节点，所以选用 firstChild 属性或是选用 lastChild 属性的效果是完全一样的。既然如此，我决定选用 firstChild 属性。

我可以把 4.4.5 节里的 alter 语句改写为如下所示的样子：

```
alert(description.firstChild.nodeValue);
```

两条语句的效果完全一样（都将显示“ Choose an image. ”消息），但这里的代码显然更容易阅读和理解。

nodeValue 属性的用途并非仅限于此。我们不仅可以用它来检索某个节点的值，还可以用它来设置某个节点的值；后一种用法正是我目前最需要的。

还记得刚才在 showPic() 函数里的 text 变量吗？当“ JavaScript 美术馆”页面上的某个图片链接被点击时， showPic() 函数会把这个链接的 title 属性值传递给 text 变量。而我现在将用 text 变量去刷新 id 值等于 description 的那个<p>元素的第一个子节点的 nodeValue 属性值，如下所示：

```
description.firstChild.nodeValue = text;
```

下面是我为了改进 showPic() 函数而给它添加的三条新语句：

```
var text = whichpic.getAttribute("title");
var description = document.getElementById("description");
description.firstChild.nodeValue = text;
```

如果用日常生活中的语言来说，这三条语句的含义依次是：

- 当“ JavaScript美术馆 ”页面上的某个图片链接被点击时， 这个链接的 title 属性值将被提取并保存到 text 变量中。
- 找到那个 id="description" 的<p> 元素，并把这个对象保存到变量 description 里去。
- 把 description 对象的第一个子节点的 nodeValue 属性值设置为变量 text 的值。

下面是完成上述改进后的 showPic() 函数的代码清单：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
  description.firstChild.nodeValue = text;
}
```

如果想测试一下这些扩展功能，请把改进后的 showPic() 函数存入 showPic.js 文件，然后在浏览器里刷新 gallery.html 文档。现在，点击这个网页上的某个图片链接时，你们将看到两种效果：“占位符”图片将被替换为这个链接所指向的一张新图片，同时图片下方的描述性文字也将被替换为这个链接的 title 属性值，如下所示。

你们可以在 http://friendsofed.com/ 网站上找到“JavaScript 美术馆”脚本文件和 HTML文档。我在示例中用到的图片也可以在那里找到，但我建议大家找一些自己的图片来测试这个脚本，那样会更有意思。

如果想让这个“JavaScript 美术馆”变得更美观，可以再给它增加一个像下面这样的样式表：

```
body {
  font-family: "Helvetica","Arial",sans-serif;
  color: #333;
  background-color: #ccc;
  margin: 1em 10%;
}
h1 {
  color: #333;
  background-color: transparent;
}
a {
  color: #c60;
  background-color: transparent;
  font-weight: bold;
  text-decoration: none;
}
ul {
  padding: 0;
}
li {
  float: left;
  padding: 1em;
  list-style: none;
}
```

请把这些 CSS代码存入 layout.css 文件，并把这个文件存放到 styles 子目录里。然后，就可以在 gallery.html 文档的 <head>部分用一个 <link> 标签来引用这个文件了，如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Image Gallery</title>
  <script type="text/javascript" src="scripts/showPic.js"></script>
  <link rel="stylesheet" href="styles/layout.css"
➥ type="text/css" media="screen" />
</head>
<body>
  <h1>Snapshots</h1>
  <ul>
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display"
```

```
➡ onclick="showPic(this); return false;">Fireworks</a>
  </li>
  <li>
    <a href="images/coffee.jpg" title="A cup of black coffee"
➡ onclick="showPic(this); return false;">Coffee</a>
  </li>
  <li>
    <a href="images/rose.jpg" title="A red, red rose"
➡ onclick="showPic(this); return false;">Rose</a>
  </li>
  <li>
    <a href="images/bigben.jpg" title="The famous clock"
➡ onclick="showPic(this); return false;">Big Ben</a>
  </li>
 </ul>
 <img id="placeholder" src="images/placeholder.gif"
➡ alt="my image gallery" />
 <p id="description">Choose an image.</p>
</body>
</html>
```

下面是“ JavaScript　美术馆 ”在经过上面那个简单的样式表修饰之后的显示效果。

4.5　小结

本章向大家介绍了一个简单的 JavaScript 应用案例。在实现这个案例的过程中，我们还向大家介绍了 DOM提供的几个新属性，它们是：

- childNodes
- nodeType
- nodeValue
- firstChild
- lastChild

本章的学习重点有两个：一是如何利用 DOM所提供的方法去编写图片库脚本；二是如何利用事件处理函数把 JavaScript 代码与网页集成在一起。

从表面上看，我们的“ JavaScript 美术馆”已经大获成功，但它实际上还有许多地方值得改进，而那将是随后两章里的讨论重点。

在下一章中，我将向大家介绍一些 JavaScript 脚本编程方面的原则和良好习惯，我希望它们能让你们领悟这样一个道理： 通往终点的过程与终点本身同样重要。

在那之后，我将向你们演示如何把那些编程原则和良好习惯应用到“ JavaScript 美术馆”案例上。