第 5 章　文档对象模型(DOM)

文档对象模型（Document Object Model：DOM），最初是 W3C 为了解决浏览器混战时代不同浏览器环境之间的差别而制定的模型标准，主要是针对 IE 和 Netscape Navigator。W3C 解释为：“文档对象模型（DOM）是一个能够让程序和脚本动态访问和更新文档内容、结构和样式的语言平台，提供了标准的 HTML 和 XML 对象集，并有一个标准的接口来访问并操作它们。”它使得程序员可以很快捷地访问 HTML 或 XML 页面上的标准组件，如元素、样式表、脚本等等并作相应的处理。DOM 标准推出之前，创建前端 Web 应用程序都必须使用 Java Applet 或 ActiveX 等复杂的组件，现在基于 DOM 规范，在支持 DOM 的浏览器环境中，Web 开发人员可以很快捷、安全地创建多样化、功能强大的 Web 应用程序。本章只讨论 HTML DOM。

5.1 DOM 概述

文档对象模型定义了 JavaScript 可以进行操作的浏览器，描述了文档对象的逻辑结构及各功能部件的标准接口。主要包括如下方面：

- 核心 JavaScript 语言参考（数据类型、运算符、基本语句、函数等）
- 与数据类型相关的核心对象（String、Array、Math、Date 等数据类型）
- 浏览器对象（window、location、history、navigator 等）
- 文档对象（document、images、form 等）

JavaScript 使用两种主要的对象模型：浏览器对象模型（BOM）和文档对象模型（DOM），前者提供了访问浏览器各个功能部件，如浏览器窗口本身、浏览历史等的操作方法；后者则提供了访问浏览器窗口内容，如文档、图片等各种 HTML 元素以及这些元素包含的文本的操作方法。在早期的浏览器版本中，浏览器对象模型和文档对象模型之间没有很大的区别。观察下面的简单 HTML 代码：

```
//源程序 5.1
<!DOCTYPE HTML PUBLIC   "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
   <meta http-equiv=content-type content="text/html; charset=gb2312">
   <title> First Page!</title>
</head>
<body>
<h1>Test!</h1>
<!--NOTE!-->
<p>Welcome to<em> DOM </em>World! </p>
<ul>
   <li>Newer</li>
</ul>
</body>
</html>
```

在 DOM 模型中，浏览器载入这个 HTML 文档时，它以树的形式对这个文档进行描述，其中各 HTML 的每个标记都作为一个对象进行相关操作，如图 5.1 所示。

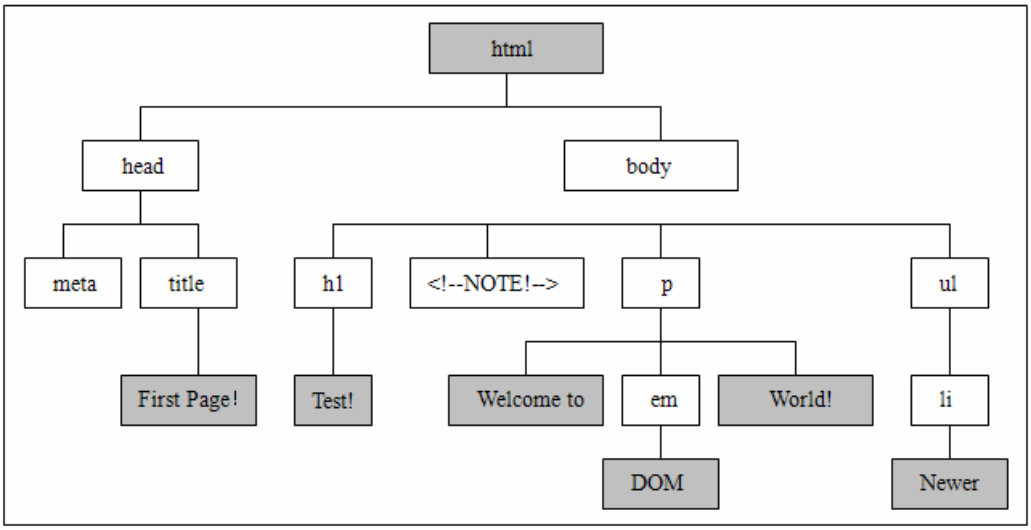

图 5.1 实例的家谱树

可以看出，html 为根元素对象，可代表整个文档，head 和 body 两个分支，位于于同一层次，为兄弟关系，存在同一父元素对象，但又有各自的子元素对象。

在支持脚本的浏览器发展过程中，出现了如下 6 种不同的文档对象模型，如表 5.1 所示：

表 5.1 文档对象模型的各个版本及浏览器支持

文档对象模型	浏览器支持
Basic Object Model（基本对象模型）	NN2，NN3，IE3/J1，IE3/J2，NN4，IE4，IE5，NN6，IE5.5，IE6，Moz1，Safari1
Basic Plus Images（基本附加图像）	NN3，IE3.01（Only for Mac），NN4，IE4，IE5，NN6，IE5.5，IE6，Moz1，Safari1
NN4 Extensions（NN4扩展）	NN4
IE4 Extensions（IE4扩展）	IE4，IE5，IE5.5，IE6(所有版本的一些功能需要Win32 OS)
IE5 Extensions（IE5扩展）	IE5，IE5.5，IE6(所有版本的一些功能需要Win32 OS)
W3C DOM（W3C文档对象模型I、II）	IE5，NN6，IE5.5，Moz1，Safari1（均为部分）

术语：IE4 表示 Internet Explorer 4，NN4 表示 Netscape Navigator 4，Moz1 表示 Mozilla1，其余类推

DOM 不同版本的存在给客户端程序员带来了很多的挑战，编写当前浏览器中最新对象模型支持的 JavaScript 脚本相对比较容易，但如果使用早期版本的浏览器访问这些网页，将会出现不支持某种属性或方法的情况。如果要使设计的网页能运行于绝大多数浏览器中，显而易见将是个难题。因此，W3C DOM 对这些问题做了一些标准化工作，新的文档对象模型继承了许多原始的对象模型，同时还提供了文档对象引用的新方法。

下面介绍在所有支持脚本的浏览器中均可实现的最低公用标准的文档对象模型：基本对象模型。

5.1.1 基本对象模型

基本对象模型提供了一个非常基础的文档对象层次结构（如图 5.2 所示），并最先受到 NN2 的脚本支持。在该模型中，window 位于对象层次的最高级，包括全部的 document 对象，同时具有其他对象所没有的属性和方法，document 就是浏览器载入的 HTML 页面，其上的链接和表单元素如按钮等交互性元素被作为有属性、方法和事件处理程序的元素对象来对待。由于功能十分有限，JavaScript 主要应用于简单的网页操作，如表单合法性验证、获

取程序最后一次修改的时间等等。

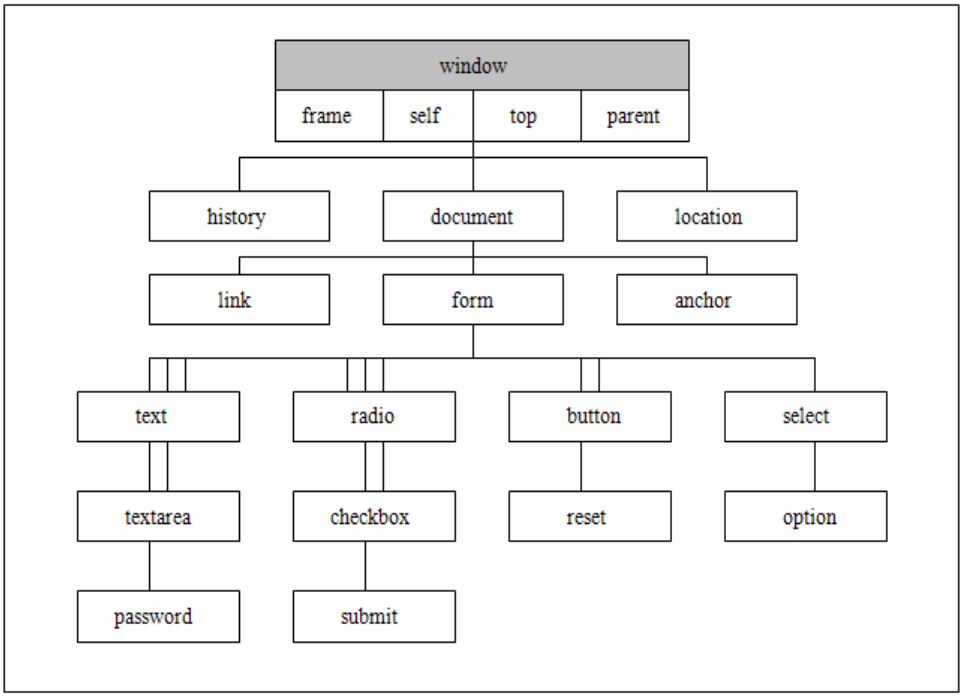

图 5.2 基本文档对象模型

IE3 及其他更高版本的浏览器实现了来自 NN2 的基本对象模型，因此，NN2 后续的浏览器版本的文档对象模型本质是相同的，只不过添加了其他的 window 对象及其操作方法，同时提供了引用原始对象的新方法，如 navigator 和 screen 对象等。

5.1.2 浏览器扩展

在各个版本浏览器中，文档对象模型都有其特殊的地方。一般来说，每发布一个新版本的浏览器，浏览器厂商都会以各种方式扩展 document 对象，新版本修订了老版本的程序错误，同时添加了对象的属性、方法及事件处理程序等，不断扩充原有的功能。

当然，从新对象模型可以更快捷地执行更多任务的技术层面上来看，每次的浏览器版本更新绝对不是一件坏事，但不同浏览器的对象模型朝着不同方向发展，却给 Web 程序员将应用程序在不同浏览器之间移植方面带来了相当的难度，导致 Web 应用程序的跨平台性较差。下面讨论文档对象模型发展过程中主要浏览器版本的对象模型，特别强调各种版本的文档对象模型的新特性以及它们和常用编程任务之间的关系。

1. Netscape Navigator 浏览器

基本对象模型最先在 NN2 种获得支持，虽然功能很有限，这也为文档对象模型的发展奠定了坚实的基础。在 NN3 中通过访问嵌入对象、Applet 应用程序、插件等，使第一个简单、类似于 DHTML 的应用程序的出现成为可能，且脚本语言能访问更多的文档属性和方法。表 5.2 中列出了 NN3 中的 document 对象新增的主要内容：

表 5.2 NN3 中document对象新增主要内容

属性	附加说明
applets[]	文本中的applets数组，用<applet>和</applet>标记

embeds[]	文本中嵌入对象的数组
images[]	文本中图像的数组，用<img>和</img>标记
plugins[]	浏览器中的插件数组
domain	包含web服务器主机名的字符串，仅能改变为更一般的主机名

NN3 中的文档对象模型如图 5.3 所示，其中灰色框内为 NN3 中 document 对象新增内容。

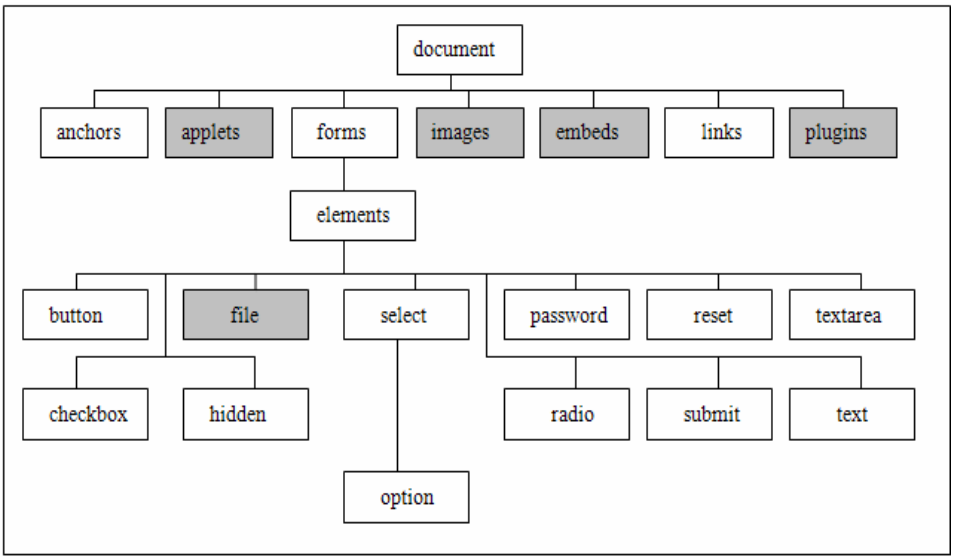

图 5.3 NN3 中 document 对象新增内容

NN3 中增加的最重要对象就是 images 对象，可通过 document.images 得到文档的一个 image 数组，然后通过一下语句进行操作：

```
document.images[n].src=…
```

images 对象的大多数属性都是只读的，而 src 可读可写，典型应用是图片翻转程序，如下代码所示：

```
//源程序 5.2
<!DOCTYPE HTML PUBLIC   "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<title>Sample</title>
</head>
<body>
<a href="#"
    onmouseover="document.images[0].src='01.jpg'"
    onmouseout="document.images[0].src='02.jpg'">
<img src="01.jpg">
</a>
</body>
</html>
```

在 NN4 之前，Web 应用程序基本不具有动态性，在 NN4 中，新增<layer>标记支持，改进了 Netscape 事件模型、style 对象及其操作方法，表 5.3 中列出了 NN3 中的 Document 对象新增的主要内容：

表 5.3 NN4 中document对象新增主要内容

属性	附加说明

classes	针对HTML标记，创建或使用带有class属性集的CSS样式
Ids	针对HTML标记，创建或使用带有id属性集的CSS样式
tags	针对任意的HTML标记，创建或使用CSS样式
layers[]	文本中的层数组，<layer>标记或 定位元素对象

NN4 中的文档对象模型如图 5.4 所示，其中灰色框内为 NN4 中 document 对象新增内容。

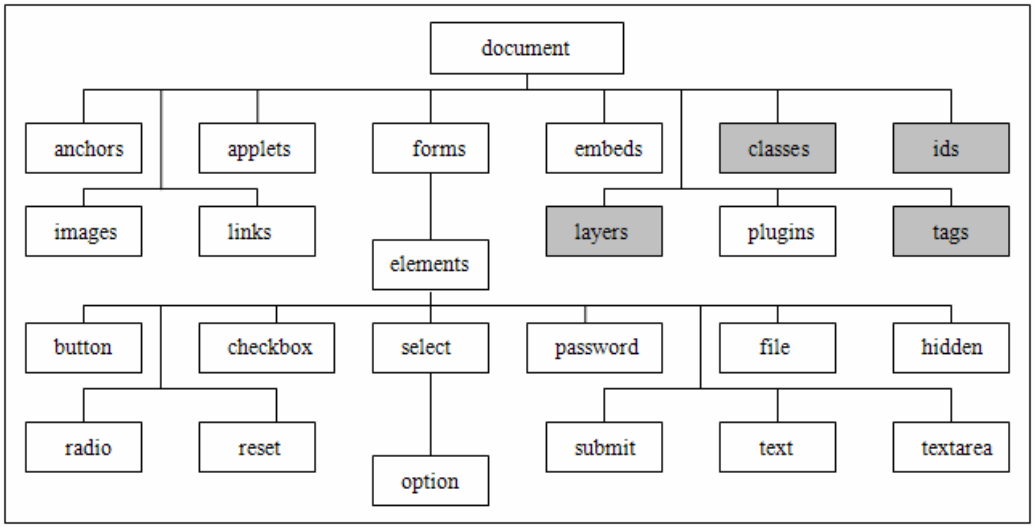

图 5.4 NN4 中 document 对象新增内容

新增对象 layer(层)是一个容器，可以容纳自己的文档，从而拥有自己的 document 对象。当然，这个文档从属于主文档。JavaScript 通过操作它的属性和方法，可以动态改变 layer 的尺寸、位置、隐藏与否等。如果有多个层，还可以更改其堆栈顺序，并且首次允许层覆盖文档中的其它元素。假如需要访问 layer 标记为“MyLayer”的层中标记为“MyPicture”的 image 对象的 src 属性，可通过如下方式访问：

```
document.MyLayer.document.MyPicture.src
```

但是在 W3C DOM 中，layer 对象没被吸收为标准对象，而是用定位 div 对象和 span 对象代替，同时赋予了新的属性、方法和事件处理程序。在 NN4 中，document.layers[]返回文档的 layers[]数组，而在其它浏览器中，则返回 undefined，这也提供了一个鉴别 NN4 浏览器的有效方法。

tags[]属性可以在全局范围内操作某个 HTML 元素对象的样式，语法如下：

```
document.tags.tagName.propertyName
```

其中，tagName 指 HTML 元素对象，propertyName 指要访问的 CSS 属性。例如：

```
<h1 class="site-name title">会员管理系统</h1>
```

如果要改变<h1></h1>之间文本对象的颜色或其它属性，可以通过以下简单的方法：

```
document.tags.h1.color="red"
```

NN6 是 Netscape 浏览器发展的里程碑，它向前兼容 DOM Level 0，也即 W3C 的 DOM 标准，并合并了早期文档对象模型中被广泛使用的特性。同时，它也部分遵循 DOM Level 1 和 DOM Level 2 标准，主要包括 W3C 对于 HTML、XML、CSS 和事件的对象模型。同时它放弃了 NN4 支持但差不多是其特有的扩展内容，如<layer>标记以及对应的 JavaScript 对象，打破了 windows 程序“向下兼容”的规律，导致很多老版本浏览器上支持的脚本在 NN6 中无效。

2. Internet Explorer 浏览器

IE3 是 IE 家族较早支持文档对象模型的浏览器，其对象模型基于 5.1.1 节的基本对象模

型，但是扩展了几个属性，如 frame[]数组等。IE3 中对象模型如图 5.5 所示。

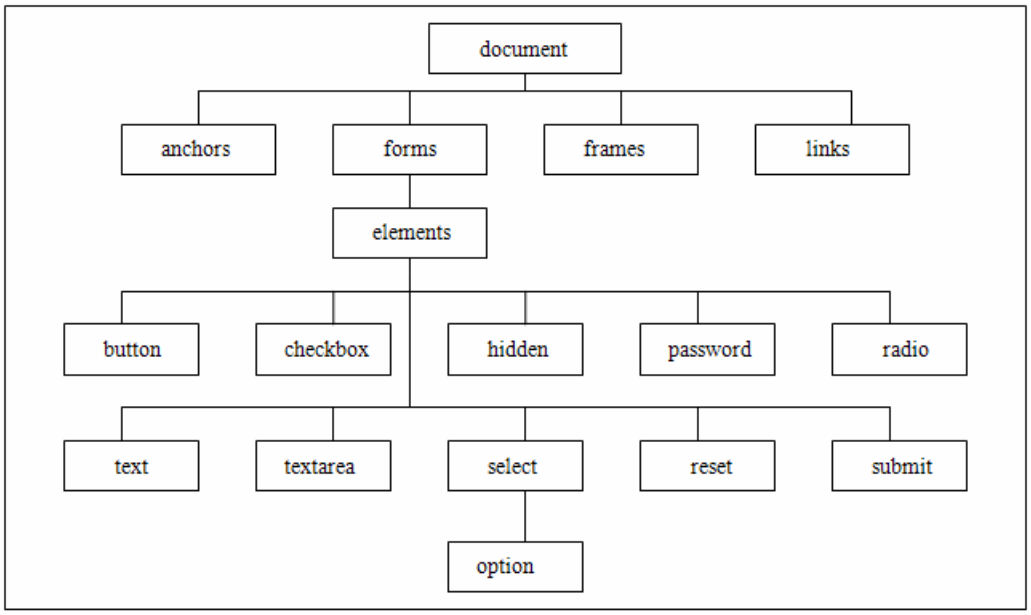

图 5.5 IE3 对象模型结构

IE4 时代，JavaScript 脚本被广泛地运用于 Web 应用程序来实现网页的动态，同时它将每个 HTML 元素都表示为对象。IE4 支持 NN2 和 IE3 的文档对象模型，同时具有许多新的、和 NN4 完全不一样的 document 对象特性，在此，Netscape Navigator 和 IE 这两种使用最为广泛的浏览器开始分道扬镳。表 5.4 列出了 IE4 中的新文本属性。

表 5.4 IE4 中的新文本属性

属性	附加说明
all[]	文档中所有HTML标记的数组
children[]	对象所有子标记数组
embeds[]	文档中嵌入对象的数组，用<embeds>标记
images[]	文档中图像对象数组，用<img>标记
scripts[]	文档的脚本数组，用<script>标记
styleSheets[]	文档中样式（style）对象数组，用<style>标记
applets[]	文档中所有applets数组，用<applets>标记

IE4 中最重要的是引入了 JavaScript 功能部件 document.all[]集合，通过它可以访问文档中的所有对象，通过其特有的检索方式，返回和索引相匹配的对象集合。document.all[]集合拆散了文档对象的层次结构，可对 HTML 文档的任意对象进行快速和简易的访问。IE4 中可通过多种方式快捷访问指定的对象或对象集合：

```
document.all[3];
document.all["name"];
document.all.item("name");
document.all.tags("p");
```

IE4 中给文本对象添加了许多非常使用的新属性和操作的方法，可见 IE4 使动态 Web 应用程序成为现实，提供了对象动态操作以及 HTML 和文本中任意插入、修改和删除等方法。见表 5.5 和表 5.6。

表 5.5 IE4 中文本对象的新属性

属性	附加说明

all[]	对象包含的所有标记的集合
children[]	对象直接派生的标记的集合
innerHTML	包含由对象的标记中包含的HTML内容的字符串，对多数HTML标记可写
innerText	包含由对象的标记中包含的文本内容的字符串，对多数HTML标记可写
outerHTML	包含标记中包含的HTML内容的字符串，对多数HTML标记可写
outerText	包含标记中包含的文本内容的字符串，对多数HTML标记可写
parentElement	对父对象的引用
className	包含对象的CSS类的字符串
style	包含对象的CSS属性的style对象
tagName	包含与对象相关的HTML标记名字的字符串

表 5.6 IE4 中文本对象的新方法

方法	附加说明
Click()	模拟鼠标单击，触发onClick()事件处理器
getAttribute()	返回所指标记的HTML属性的参数
setAttribute()	设置所指标记的HTML属性的参数
removeAttribute()	从标记中删除HTML属性的参数
insertAdjacentHTML()	在标记的前面、后面或当中插入HTML语句
insertAdjacentText()	在标记的前面、后面或当中插入文本

IE4 对象模型结构如图 5.6 所示，其中灰色框内的为 IE4 中 document 对象新增内容。

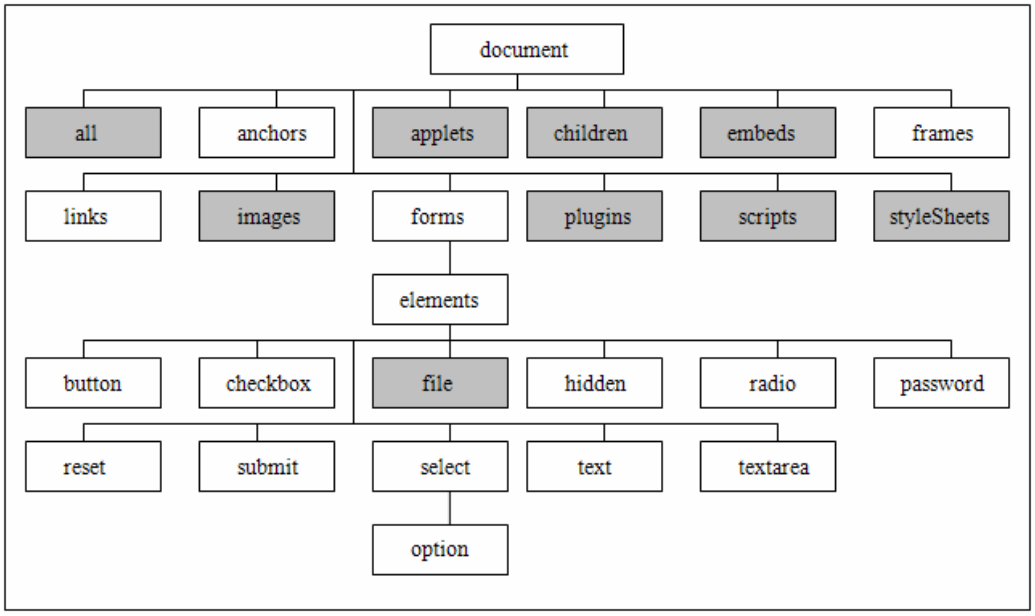

图 5.6 IE4 对象模型结构

下面一段代码综合了 IE4 中文本对象操作的新方法：

```
//源程序 5.3
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<title>Sample</title>
</head>
<body>
<br>
<h1 id="MyTest" align="center">My Test String!<h1><br>
```

```
<form name="MyTestForm" id="MyTestForm">
   <input type="button" value="左对齐" onclick="document.all['MyTest'].align='left'">
   <input type="button" value="右对其" onclick="document.all['MyTest'].align='right'">
   <input type="button" value="中间对齐" onclick="document.all['MyTest'].align='center'"><br>
   <input type="button" value="字体红色" onclick="document.all['MyTest'].style.color='red'">
   <input type="button" value="字体绿色" onclick="document.all['MyTest'].style.color='green'">
   <input type="button" value="字体黑色" onclick="document.all['MyTest'].style.color='black'"><br>
   <input type="text" name="用户内容" id="userText" size="30">
   <input type="button" value="改变字符串内容"
   onclick="document.all['MyTest'].innerText=document.MyTestForm.userText.value"><br>
</form>
</body>
</html>
```

IE5 文档对象模型中与 IE4 极其相似，但对 IE4 进行了功能扩展，增加了对象的可用属性和方法，使得它更为强大，具有更强的文档操作能力。同时，IE5 中的事件处理器数目也大大增加，达到 40 多种，从专门的鼠标和键盘事件到进行剪贴、复制的事件。IE5 中支持两种新的功能部件：DHTML 行为和 HTML 应用程序，前者允许程序员定义可被任何元素反复使用的 DHTML 成分，后者则表现得更像真正的程序而不是 Web 应用程序。

IE5.5、6、7 在继续 IE4 文档对象模型的基础上，在实现 W3C DOM 规范的同时，继续添加只能在 IE 内核浏览器中运行的功能部件，包括新的属性、方法和事件处理程序。从 IE6 开始，完全符合 CSS1 和 DOM Level 1 标准。

较之其他浏览器，IE 对 W3C DOM 标准贯彻得不是很完全，尚有许多有待完善的地方。

3. Opera、Mozilla 和其他浏览器

由于 IE 的漏洞及运行速度问题，很多工程专业的使用者采用 Opera、Mozilla 及其它的浏览器。这些浏览器一般非常严格执行 W3C 和 ECMA 标准，而不采用 IE 和 NN 的特有对象模型(一些也提供对 NN2 和 IE3 对象模型的支持)，主要致力于在 W3C 标准基础上进行开发。

在针对内容和样式的 HTML 和 CSS 的公用规范推出后，DOM 承诺建立统一的范例，用于生成不仅能和网络，而且能和离线的商业或者技术系统进行交互的文档。Netscape 承诺并已经在 NN6 中彻底贯彻了这一标准。Mozilla 具有最好的 DOM 支持，实现了完整的 DOM Level 1、几乎所有的 DOM Level 2 以及部分 DOM Level 3。在 Mozilla 之后，Opera 和 Safari 也在完全支持该标准上做出了相当的努力，极大缩小了与标准之间的差距，支持几乎所有的 DOM Level 1 和大部分 DOM Level 2。

5.1.3 W3C DOM

客户端 Web 应用程序开发人员面对的最大障碍在于 DOM 有很多不同的版本，同时在浏览器版本更替过程中，对象模型又不是统一的，如果需要在不同浏览器环境中运行该网页，将会发现对象的很多属性或方法，甚至某些对象都不起作用。W3C 文档对象模型(DOM)是一个中立的接口语言平台，为程序以及脚本动态地访问和更新文档内容，结构以及样式提供一个通用的标准。它将把整个页面(HTML 或 XML)规划成由节点分层构成的文档，页面的每个部分都是一个节点的衍生物，从而使开发者对文档的内容和结构具有空前的控制力，用 DOM API 可以轻松地删除、添加和替换指定的节点。

DOM 规范必须适应 HTML 的已知结构，同时适应 XML 文档的未知结构。DOM 的概念主要有：

- 核心 DOM：指定类属类型，将带有标记的文档看成树状结构并据此对文档进行相

关操作；

- DOM 事件：包括使用者熟悉的鼠标、键盘事件，同时包括 DOM 特有的事件，当操作文档对象模型中的各元素对象时发生。
- HTML DOM：提供用于操作 HTML 文档以及类似于 JavaScript 对象模型语法的功能部件，在核心 DOM 的基础上支持对所有 HTML 元素对象进行操作。
- XML DOM：提供用于操作 XML 文档的特殊方法，在核心 DOM 的基础上支持对 XML 元素如进程指导、名称空间、CDATA 扇区项等的操作。
- DOM CSS：提供脚本编程实现 CSS 的接口。

在 W3C DOM 规范发展历史上，主要出现了三种不同的版本，下面作简要的介绍。

5.1.4 W3C DOM 规范级别

DOM 规范是一个逐渐发展的概念，规范的发行通常与浏览器发行的时间不很一致，导致任何特定的浏览器的发行版本都只包括最近的 W3C 版本。W3C DOM 经历了三个阶段。

DOM Level 1 是 W3C 于 1998 年 10 月提出的第一个正式的 W3C DOM 规范。它由 DOM Core 和 DOM HTML 两个模块构成。前者提供了基于 XML 的文档的结构图，以方便访问和操作文档的任意部分；后者添加了一些 HTML 专用的对象和方法，从而扩展了 DOM Core。DOM Level 1 的主要目标是合理规划文档的结构。它的最大缺陷就是忽略了事件模型，其中包括 NN2 和 IE3 中最简单的事件模型。

DOM Level 2 基于 DOM Level 1 并扩展了 DOM Level 1，添加了鼠标和用户界面事件、范围、遍历（重复执行 DOM 文档的方法）、XML 命名空间、文本范围、检查文档层次的方法等新概念，并通过对象接口添加了对 CSS 的支持。同时引入几个新模块，用以处理新的接口类型，包括：

- DOM 视图——描述跟踪文档的各种视图（即 CSS 样式化之前和 CSS 样式化之后的文档）的接口；
- DOM 事件——描述事件的接口；
- DOM 样式表——描述处理基于 CSS 样式的接口；
- DOM 遍历和范围——描述遍历和操作文档树的接口。

DOM Level 3 引入了以统一的方式载入和保存文档的方法（包含在新模块 DOM Load and Save 中）以及验证文档（DOM Validation）的方法，从而进一步扩展了 W3C DOM 规范。在 DOM Level 3 中，DOM Core 被扩展为支持所有的 XML 1.0 特性，包括 XML Infoset、XPath 和 XML Base，从而改善了 DOM 对 XML 的支持。

由于 DOM Level 1 在当前浏览器版本中获得最广泛的支持，本章余下部分只针对 DOM Level 1 版本进行讨论。

> 注意：DOM 发展历史上曾出现了 DOM Level 0 的说法，其实 W3C DOM 规范并无此版本，只是 DOM 标准的一个具有试验性质的初级 DOM，主要指第四代浏览器中支持的原始 DHTML，包括对图像进行链接、显示以及在客户端进行表单的数据合法性验证。

5.2 文档对象模型的层次

文档对象模型具有层次结构，由于 JavaScript 是基于对象的编程语言，而不是面向对象的编程语言，所以在 JavaScript 编程中不必考虑类及类的实例、继承等等晦涩难懂的编程术

语，只需充分了解不同浏览器中文档对象模型的层次结构。引用对象的能力决定了代码的功能，而对象则依赖于其在文档对象模型中的层次。知道了对象在文档对象模型中所处的层次，就可以用 JavaScript 准确定位并操作该对象。

举一个很简单的例子：假如一个班的同学在上素描课，指导老师觉得 Group1 的 TOM 的画图纸 BackColor 为 LightGray 比较恰当，就说：“Group1 的 TOM，将你的画图纸的 BackColor 改为 LightGray 好吗？”，TOM 听到老师的指示，把画图纸的 BackColor 改为 LightGray。在基于对象的编程方法中，“TOM”是 Object，“ChangeBackColor”为 Command，“LightGray”是 Parameters，可以通过如下的方法实现：

目标对象层次：

```
Group1.TOM
```

ChangeBackColor 可认为是 TOM 的一种方法，所以 TOM 及其方法的完整引用如下：

```
Group1.TOM.ChangeBackColor( )
```

方法需要一种参数去决定是改变成 Color 集合里的什么颜色，所以准确表示该模型的完整命令就是：

```
Group1.TOM.ChangeBackColor(Color.LightGray)
```

该实例的层次结构如图 5.7 所示。

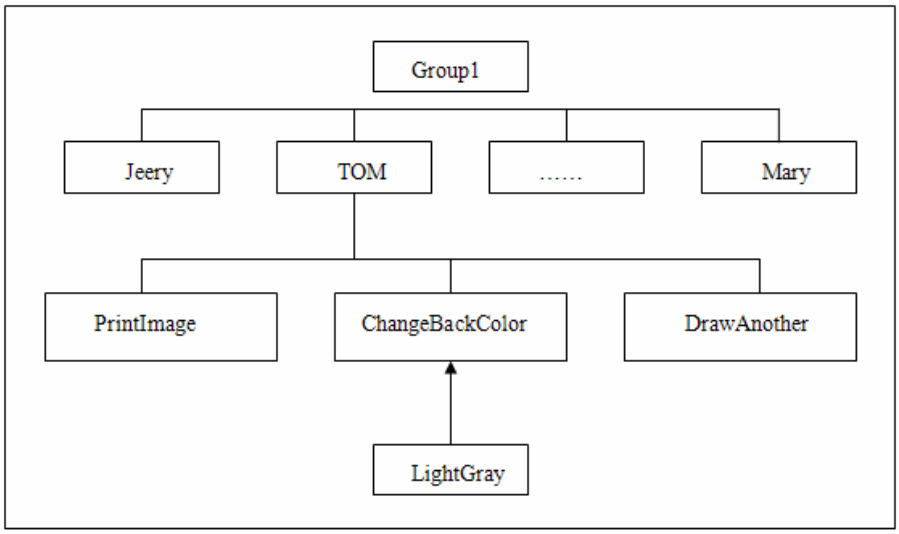

图 5.7 实例中的对象访问结构

如果引用 TOM.ChangeBackColor(Color.LightGray)来表述上述事件，而且班上其它组也有叫 TOM 的，这就很容易带来识别上的混乱。从上述意义来讲，搞清楚文档对象模型对准确定位文档对象并施加相应的操作相当重要。知道了对象所属层次，就知道了访问对象的途径，从而获得对象的操作方法。

5.3 文档对象的产生过程

在面向对象或基于对象的编程语言中，指定对象的作用域越小，对象位置的假定也就越多。对客户端 JavaScript 脚本而言，其对象一般不超过浏览器，脚本不会访问计算机硬件、操作系统、其他程序等其他超出浏览器的对象。

HTML 文档载入时，浏览器解释其代码，当遇到自身支持的 HTML 元素对象对应的标记时，就按 HTML 文档载入的顺序在内存中创建这些对象，而不管 JavaScript 脚本是否真正

运行这些对象。对象创建后，浏览器为这些对象提供专供 JavaScript 脚本使用的可选属性、方法和处理程序。通过这些属性、方法和处理程序，Web 开发人员就能动态操作 HTML 文档内容，下面代码演示如何动态改变文档的背景颜色：

```
//源程序 5.4
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<script language="javascript">
<!--
function changeBgClr(value)
{
   document.body.style.backgroundColor=value
}
//-->
</script>
</head>
<body>
<form>
   <input type=radio value=red onclick="changeBgClr(this.value)">red
   <input type=radio value=green onclick="changeBgClr(this.value)">green
   <input type=radio value=blue onclick="changeBgClr(this.value)">blue
</form>
</body>
</html>
```

其中 document.body.style.backgroundColor 语句表示访问当前 document 对象固有子对象 body 的样式子对象 style 的 backgroundColor 属性。

> 注意：如果创建一个多框架页面，则直到浏览器载入所有框架时，某个框架内的脚本才能与其它框架进行通信。

5.4 对象的属性和方法

DOM 将文档表示为一棵枝繁叶茂的家谱树，如果把文档元素想象成家谱树上的各个节点的话，可以用同样的记号来描述文档结构模型，在这种意义上讲，将文档看成一棵“节点树”更为准确。在充分认识这棵树之前，先来了解节点的概念。

5.4.1 何谓节点

所谓节点(node)，表示某个网络中的一个连接点，换句话说，网络是节点和连线的集合在 W3C DOM 中，每个容器、独立的元素或文本块都被看着一个节点，节点是 W3C DOM 的基本构建块。当一个容器包含另一个容器时，对应的节点之间有父子关系。同时该节点树遵循 HTML 的结构化本质，如<html>元素包含<head>、<body>，前者又包含<title>，后者包含各种块元素等。DOM 中定义了 HTML 文档中 6 种相关节点类型。所有支持 W3C DOM 的浏览器（IE5+，Moz1 和 Safari 等）都实现了前 3 种常见的类型，其中 Moz1 实现了所有类型。如表 5.7 所示：

表 5.7 DOM定义的HTML文档节点类型

节点类型数值	节点类型	附加说明	实例
1	元素(Element)	HTML标记元素	…
2	属性(Attribute)	HTML标记元素的属性	color="red"
3	文本(Text)	被HTML标记括起来的文本段	Hello World!
8	注释(Comment)	HTML注释段	<!--Comment-->
9	文档(Document)	HTML文档根文本对象	<html>
10	文档类型(DocumentType)	文档类型	<!DOCTYPE HTML PUBLIC "…">

注意：IE6 内核浏览器中属性（attribute）类型在 IE6 版本中才获得支持。

具体来讲，DOM 节点树中的节点有元素节点、文本节点和属性节点等三种不同的类型，下面具体介绍。

1. 元素节点(element node)

在 HTML 文档中，各 HTML 元素如<body>、<p>、<ul>等构成文档结构模型的一个元素对象。在节点树中，每个元素对象又构成了一个节点。元素可以包含其它的元素，例如在下面的“购物清单”代码中：

```
<ul id="purchases">
  <li>Beans</li>
  <li>Cheese</li>
  <li>Milk</li>
</ul>
```

所有的列表项元素<li>都包含在无序清单元素<ul>内部。其中节点树中<html>元素是节点树的根节点。

2. 文本节点(text node)

在节点树中，元素节点构成树的枝条，而文本则构成树的叶子。如果一份文档完全由空白元素构成，它将只有一个框架，本身并不包含什么内容。没有内容的文档是没有价值的，而绝大多数内容由文本提供。在下面语句中：

```
<p>Welcome to<em> DOM </em>World! </p>
```

包含“Welcome to” 、“DOM” 、 “World!”三个文本节点。在 HTML 中，文本节点总是包含在元素节点的内部，但并非所有的元素节点都包含或直接包含文本节点，如“购物清单”中，<ul>元素节点并不包含任何文本节点，而是包含着另外的元素节点，后者包含着文本节点，所以说，有的元素节点只是间接包含文本节点。

3. 属性节点(attribute node)

HTML 文档中的元素或多或少都有一些属性，便于准确、具体地描述相应的元素，便于进行进一步的操作，例如：

```
<h1 class="Sample">Welcome to DOM World！</h1>
<ul id="purchases">…</ul>
```

这里 class="Sample"、id="purchases"都属于属性节点。因为所有的属性都是放在元素标签里，所以属性节点总是包含在元素节点中。

注意：并非所有的元素都包含属性，但所有的属性都被包含在元素里。

5.4.2 对象属性

属性一般定义对象的当前设置，反映对象的可见属性，如 checkbox 的选中状态，也可

能是不很明显的信息，如提交 form 的动作和方法。在 DOM 模型中，文档对象有许多初始属性，可以是一个单词、数值或者数组，来自于产生对象的 HTML 标记的属性设置。如果标记没有显式设置属性，浏览器使用默认值来给标记的属性和相应的 JavaScript 文本属性赋值。DOM 文档对象主要有如下重要属性，如表 5.8 所示：

表 5.8 文档对象的属性

节点属性	附加说明
nodeName	返回当前节点名字
nodeValue	返回当前节点的值，仅对文本节点
nodeType	返回与节点类型相对应的值，如表5.8
parentNode	引用当前节点的父节点，如果存在的话
childNodes	访问当前节点的子节点集合，如果存在的话
firstChild	对标记的子节点集合中第一个节点的引用，如果存在的话
lastChild	对标记的子节点集合中最后一个节点的引用，如果存在的话
previousSibling	对同属一个父节点的前一个兄弟节点的引用
nextSibling	对同属一个父节点的下一个兄弟节点的引用
attributes	返回当前节点（标记）属性的列表
ownerDocument	指向包含节点（标记）的HTML document对象

注意：firstchild 和 lastchild 指向当前标记的子节点集合内的第一个和最后一个子节点，但是多数情况下使用 childNodes 集合，用循环遍历子节点。如果没有子节点，则 childNodes 长度为 0。

例如如下 HTML 语句：

```
<p id="p1">Welcome to<B> DOM </B>World! </p>
```

可以用如图 5.8 的节点树表示，并标出节点之间的关系。

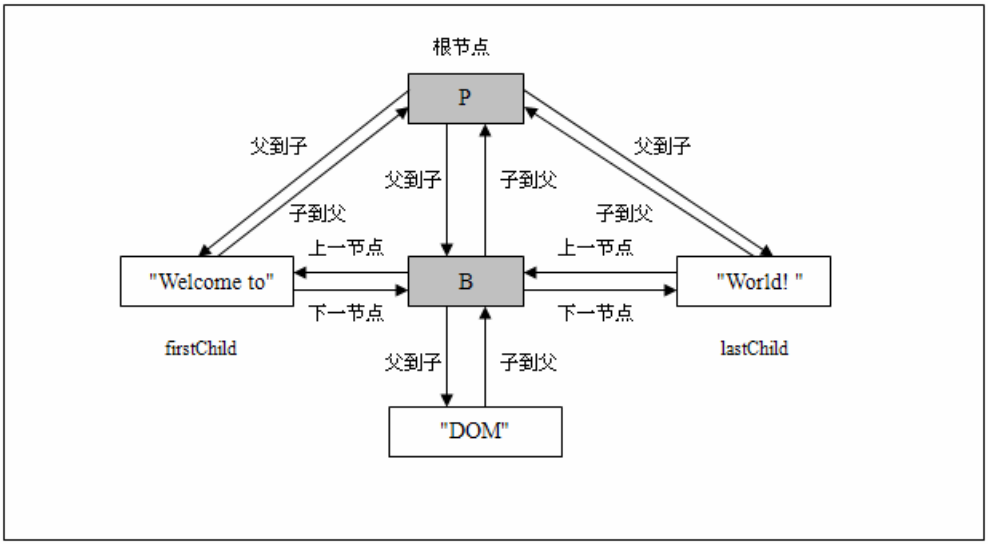

图 5.8 例句的节点树表示

下面的代码演示如何在节点树中按照节点之间的关系检索出各个节点：

```
//源程序 5.5
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
```

```
<title> First Page!</title>
</head>
<body>
<p id="p1">Welcome to<B> DOM </B>World! </p>
<script language="JavaScript" type="text/javascript">
<!--
//输出节点属性
function nodeStatus(node)
{
   var temp="";
   if(node.nodeName!=null)
   {
      temp+="nodeName: "+node.nodeName+"\n";
   }
   else temp+="nodeName: null!\n";
   if(node.nodeType!=null)
   {
      temp+="nodeType: "+node.nodeType+"\n";
   }
   else temp+="nodeType: null\n";
   if(node.nodeValue!=null)
   {
      temp+="nodeValue: "+node.nodeValue+"\n\n";
   }
   else temp+="nodeValue: null\n\n";
   return temp;
}
//处理并输出节点信息
//返回 id 属性值为 p1 的元素节点
var currentElement=document.getElementById('p1');
var msg=nodeStatus(currentElement);
//返回 p1 的第一个孩子，即文本节点“Welcome to”
currentElement=currentElement.firstChild;
msg+=nodeStatus(currentElement);
//返回文本节点“Welcome to”的下一个同父节点，即元素节点 B
currentElement=currentElement.nextSibling;
msg+=nodeStatus(currentElement);
//返回元素节点 B 的第一个孩子，即文本节点“DOM”
currentElement=currentElement.firstChild;
msg+=nodeStatus(currentElement);
//返回文本节点“DOM”的父节点，即元素节点 B
currentElement=currentElement.parentNode;
msg+=nodeStatus(currentElement);
//返回元素节点 B 的同父节点，即文本节点“Welcome to”
currentElement=currentElement.previousSibling;
msg+=nodeStatus(currentElement);
//返回文本节点“Welcome to”的父节点，即元素节点 P
currentElement=currentElement.parentNode;
msg+=nodeStatus(currentElement);
//返回元素节点 P 的最后一个孩子，即文本节点“World!”
currentElement=currentElement.lastChild;
msg+=nodeStatus(currentElement);
//输出节点属性
```

```
alert(msg);
//-->
</script>
</body>
</html>
```

运行上述代码，结果如图 5.9 所示，null 指某个节点没有对应的属性。

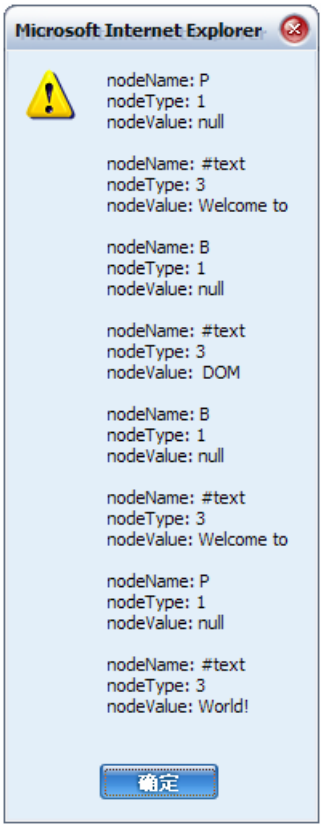

图 5.9 节点数的遍历方法实例

> 注意：遍历浏览器载入 HTML 文档形成的节点树时，可通过 document.documentElement 属性来定位根节点，即<html>标记。

在准确定位节点树中的某个节点后，就可以使用对象的方法来操作这个节点，下面介绍对象(节点)的操作方法。

5.4.3 对象方法

对象方法是脚本给该对象的命令，可以有返回值，也可没有，且不是每个对象都有方法定义。DOM 中定义了操作节点的一系列行之有效的方法，让 Web 应用程序开发者真正做到随心所欲地操作 HTML 文档中各个元素对象，先来了解 id 属性和 class 属性。

5.4.4 id 属性和 class 属性

在图 5.1 所示的家谱树中，HTML 文档载入时各元素对象都被标注成节点，同时根据浏览器载入的顺序，自动分配一个序号，可用 document.all[ ]直接访问。考察如下的实例：

```
//源程序 5.6
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title> First Page!</title>
</head>
<body>
<h5>Test!</h5>
<!--NOTE!-->
<p>Welcome to<em> DOM </em>World! </p>
<ul>
   <li>Newer</li>
</ul>
<hr>
<br>
<script language="JavaScript" type="text/javascript">
<!--
   var i,origlength;
   //获取 document.all[ ]数组的长度
   origlength=document.all.length;
   document.write('document.all.length='+origlength+"<br>");
   //循环输出各节点的 tagName 属性值
   for(i=0;i<origlength;i++)
   {
      document.write("document.all["+i+"]="+document.all[i].tagName+"<br>");
   }
//-->
</script>
</body>
</html>
```

程序运行结果如图 5.10 所示，可以看出浏览器按载入顺序为每个 HTML 元素都分配了一个序号来访问对应的元素节点。

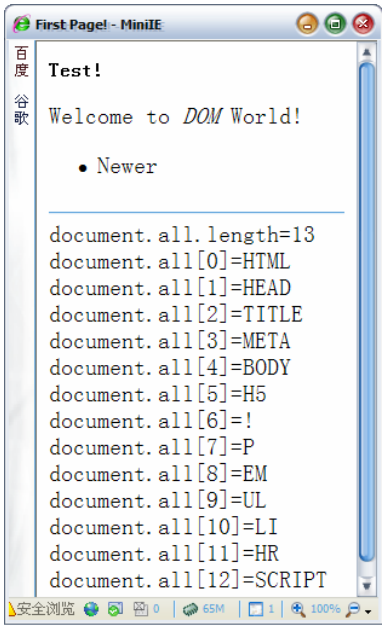

图 5.10 HTML 载入生成的 document.all[]数组

> 注意：如果使用 document.all.length 对循环进行检查，程序会出现死循环，因为每次输出正在检查的元素时，document.all[]元素个数都在增加。

由于不能直观看出 document.all[]数组中各元素的序号分布，甚至在文档最终完成之前根本无法获知节点树的架构，这种访问方法显然不能很方便、快捷地进行元素节点访问，现在引入 id 属性和 class 属性。

id 属性的用途是给 HTML 文档中的某个特定的元素对象加上独一无二（对当前文档而言）的标识符，便于精确访问这个元素对象。例如：

```
<p id="p1">Hello World!</p>
```

在 CSS 中，可以为有着特定 id 的元素对象定义一种独享的样式：

```
#p1{
   border:1px solid white;
   background-color:#333;
   color:#ccc;
   padding:1em;
}
```

每个元素对象只能有一个 id 属性值，不同的元素对象必须有不同的 id 属性值。也可利用 id 属性为包含在某给定元素里的其他元素定义样式，这样定义的样式将只作用于包含在给定元素里的指定元素。在前述的“购物清单”代码中，可通过如下方式定义
和
之间文本的样式：

```
#purchases li{
   fontweight:bold;
}
```

id 属性就是一座桥梁，连接着文档中的某个元素和 CSS 样式表中的特定样式，同时实现元素对象的相关操作。

所有的元素都有 class 属性，不同的元素可以有相同的 class 属性值，例如：

```
<p class="MyClass">The First Line</p>
<h1 class="MyClass">The Second Line</h1>
```

可以通过如下方式定义<p>和<h1>的共享样式：

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

同时，也可以通过 h1.special 方式定位第二个文本对象并改变它的样式，获取更为精确的控制。

5.4.5 getElementById( )方法

该方法返回与指定 id 属性值的元素节点相对应的对象，对应着文档里一个特定的元素节点（元素对象）。该方法是与 document 对象相关联的函数，其语法如下：

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

其中 id 为要定位的对象 id 属性值。

下面的例子演示 getElementById( )方法的使用，同时可以看出其返回一个对象(object)，而不是数值、字符串等。

```
//源程序 5.7
<!DOCTYPE HTML PUBLIC   "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
```

```
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title> First Page!</title>
</head>
<body>
<ul id="purchases">
   <li>Beans</li>
   <li>Cheese</li>
   <li>Milk</li>
</ul>
<script language="JavaScript" type="text/javascript">
<!--
document.write(typeof document.getElementById("purchases"));
//-->
</script>
</body>
</html>
```

一般来说，我们不必为 HTML 文档中的每一个元素对象都定义一个独一无二的 id 属性值，也可通过下面的 getElementByTagName( )方法准确定位文档中特定的元素。

注意： 1、JavaScript 对大小写敏感， getElementById 写成 GetElementById 、getelementById 等都不对。
2、typeof 返回数据的类型，如数值、对象、字符串等。

5.4.6 getElementByTagName( )方法

该方法返回文档里指定标签 tag 的元素对象数组，与上述的 getElementById( )方法返回对象不同，且返回的对象数组中每个元素分别对应文档里一个特定的元素节点（元素对象）。其语法如下：

```
element. getElementByTagName(tag)
```

其中 tag 为指定的标签。下面给出的例子演示该方法返回的是对象数组，而不是对象。

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

将上述的代码替换前面购物清单<script></script>之间的语句，可以看出该方法返回对象(object)数组，长度为 3。再看下面的代码：

```
var shoplist=document.getElementById("purchases");
var items=shoplist.getElementByTagName("*")
var i=items.length;
```

以上语句运行后，items 数组将只包含 id 属性值为 purchases 的无序清单里的元素，i 返回 3，与列表项元素个数相同。

由于对象数组中定位对象需要事先知道对象对应的下标号，DOM 提供了直接通过元素对象名称进行访问的方法，即 getElementByName( )方法。

注意：可以用参数“*”来获取文档中所有的元素，但此时 IE6 内核的浏览器并不犯会所有的元素，必须使用 document.all[]来获取。

5.4.7 getElementByName( )方法

相对于 id 属性值，旧版本的 HTML 文档更习惯于对<form>、<select>等元素节点使用 name 属性。此时需要用到文档对象的 getElementByName( )方法来定位。该方法返回指定名称 name 的节点序列，其语法如下：

```
Document. getElementByName(name)
```

其中 name 为指定要定位的元素对象的名字，下面的代码演示其使用方法：

```
var MyList=document.getElementByName("MyTag");
var temp=" ";
for(var i=0;i<MyList.length;i++)
{
    temp+="nodeName: "+node.nodeName+"\n";
    temp+="nodeType: "+node.nodeType+"\n";
    temp+="nodeValue: "+node.nodeValue+"\n";
}
return temp;
```

在准确定位到特定元素对象后，可通过 getAttribute( )方法将它的各种属性值查询出来。

> 注意：Opera 7.5、IE 6.0 及使用 IE6 内核的浏览器在使用此方法时有很大的不同，首先，它们返回 id 为 name 的元素；其次，仅仅检查<input>和<img>元素。该方法由于浏览器的支持问题，不是很常见。

5.4.8 getAttribute( )方法

该方法返回目标对象指定属性名称的某个属性值。语法如下：

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

其中 attribute 为对象指定要搜索的属性，下面的代码演示其使用方法：

```
//源程序 5.8
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title>Sample Page!</title>
</head>
<body>
<p title="First Sample">This is the first Sample!</p>
<script language="JavaScript" type="text/javascript">
<!--
var objSample=document.getElementByTagName("p");
for(var i=0;i<objSample.length;i++)
{
  document.write(objSample[i].getAttribute("title"));
}
//-->
</script>
</body>
</html>
```

上述代码通过 objSample.length 控制循环，遍历整个文档的<p>标记。运行结果显示为“First Sample”。

以上从节点定位到获得其指定的属性值，都只能检索信息。下面介绍指定节点的属性值进行修改的途径：setAttribute( )方法。

5.4.9 setAttribute( )方法

该方法可以修改任意元素节点指定属性名称的某个属性值，语法如下：

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

类似于 getAttribute( )方法，setAttribute( )方法也只能通过元素节点对象调用，但是需要传递两个参数：

- attribute：指定目标节点要修改的属性
- value：属性修改的目标值

下面的代码演示其功能：

```
//源程序 5.9
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title>Sample Page!</title>
</head>
<body>
<ul id="purchases">
   <li>Beans</li>
   <li>Cheese</li>
   <li>Milk</li>
</ul>
<script language="JavaScript" type="text/javascript">
<!--
   var shoplist=document.getAttributeById("purchases");
   document.write(shoplist.getAttribute("title"));
   shoplist.setAttribute("title", "New List");
//-->
</script>
</body>
</html>
```

运行结果显示 null 和 New List，因为 id 属性值为 purchases 的 ul 元素节点的 title 属性在 shoplist.setAttribute("title","New List")代码运行之前根本不存在，所以显示 null；运行后，修改 title 属性为“New List”。这意味着至少完成了两个步骤：

（1）创建 ul 元素节点的 title 属性；

（2）设置刚创建的 title 属性值；

当然，如果 title 属性值本来就存在，运行 shoplist.setAttribute("title","New List")后，title 原来的属性值被“New List”覆盖。

> 注意：通过 setAttribute( )方法对文档做出的修改，将使浏览器窗口的显示效果、行为动作等发生相应的变化，这是一个动态的过程。但是这种修改并不反应到文档本身的物理内容上。这由 DOM 的工作模式决定：先加载文档静态内容，再以动态的方式对文档进行刷新，动态刷新不影响文档的静态内容。客户端用户不需要手动执行页面刷新操作就能动态刷新页面。

5.4.10 removeAttribute( )方法

该方法可以删除任意元素节点指定的属性，语法如下：

```
object.removeAttribute(name)
```

类似于 getAttribute( )和 setAttribute( )方法，removeAttribute( )方法也只能通过元素节点对象调用。其中 name 标示要删除的属性名称，例如：

```
//源程序 5.10
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title>Sample Page!</title>
<script language="JavaScript" type="text/javascript">
<!--
function TestEvent()
{
   document.MyForm.text1.removeAttribute("disabled");
}
//-->
</script>
</head>
<body>
<form name="MyForm">
   <input type="text" name="text1" value="red" disabled>
   <input type="button" name="MyButton" value="MyTestButton" onclick="TestEvent()">
</form>
</body>
</html>
```

运行上述代码，单击“MyTestButton” 按钮之前，文本框 text1 显示为只读属性；单击“MyTestButton”按钮后，触发 TestEvent( )函数执行核心语句：

```
document.MyForm.text1.removeAttribute("disabled");
```

该语句删除 text1 的 disabled 属性，文本框变为可用状态。

5.5 附加的节点处理方法

由于文本节点具有易于操纵、对象明确等特点，DOM Level 1 提供了非常丰富的节点处理方法，如表 5.9 所示：

表 5.9 DOM中的节点处理方法

操作类型	方法原型	附加说明
生成节点	createElement(tagName)	创建由tagName指定类型的标记
	CreateTextNode(string)	创建包含字符创string的文本节点
	createAttribute(name)	针对节点创建由name指定的属性，不常用
	createComment(string)	创建由字符串string指定的文本注释
插入和添加节点	appendChild(newChild)	添加子节点newChild到目标节点上
	insertBefore(newChild,targetChild)	将新节点newChild插入目标节点targetChild之前
复制节点	cloneNode(bool)	复制节点自身，由逻辑量bool确定是否复制子节点

删除和替换节点	removeChild(childName)	删除由childName指定的节点
	replaceChild(newChild,oldChild)	用新节点newChild替换旧节点oldChild
文本节点操作	insertData(offset,string)	从由offset指定的位置插入string值
	appendData(string)	将string值插入到文本节点的末尾处
	deleteData(offset,count)	从由offset指定的位置删除count个字符
	replaceData(offset,count,string)	从由offset指定的位置用string代替count个字符
	splitText(offset)	从由offset指定的位置将文本节点分成两个文本节点，左边更新为原始节点，右边的返回到新节点
	substringData(offset,count)	返回从offset指定的位置开始的count个字符

DOM 中指定的节点处理方法，提供了 Web 应用程序开发者快捷、动态更新 HTML 页面的途径。下面通过具体实例来说明各种方法的使用。

5.5.1 生成节点

DOM 中提供的方法生成新节点的操作非常简单，语法分别如下：

```
MyElement=document.createElement("h1")
MyTextNode=document.createTextNode("My Text Node!")
MyComment=document.createComment("My Comment!")
```

下面的实例演示如何生成新节点并验证：

```
//源程序 5.11
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title>Sample Page!</title>
</head>
<script language="JavaScript" type="text/javascript">
<!--
function nodeStatus(node)
{
  var temp="";
  if(node.nodeName!=null)
  {
    temp+="nodeName: "+node.nodeName+"\n";
  }
  else temp+="nodeName: null!\n";
  if(node.nodeType!=null)
  {
    temp+="nodeType: "+node.nodeType+"\n";
  }
  else temp+="nodeType: null\n";
  if(node.nodeValue!=null)
  {
    temp+="nodeValue: "+node.nodeValue+"\n\n";
  }
  else temp+="nodeValue: null\n\n";
  return temp;
}
function MyTest( )
{
  //产生 p 元素节点和新文本节点
```

```
   var newParagraph = document.createElement("p");
   var newTextNode= document.createTextNode(document.MyForm.MyField.value);
   var msg=nodeStatus(newParagraph);
   msg+=nodeStatus(newTextNode)
   alert(msg);
   return;
}
//-->
</script>
<body>
<form name="MyForm">
   <input type="text" name="MyField" value="My Sample">
   <input type="button" value="TestButton" onclick="MyTest()">
</body>
</html>
```

上面代码运行后，单击“TestButton”按钮，触发 MyTest()函数，结果如图 5.11 所示。

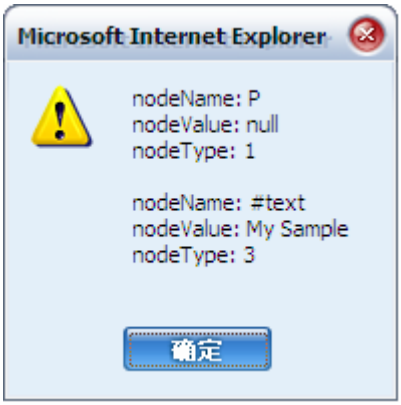

图 5.11 实例生成的元素节点和文本节点

生成节点后，要将节点添加到 DOM 树中，下面介绍插入和添加节点的方法。

5.5.2 插入和添加节点

把新创建的节点插入到文档的节点树最简单的方法就是让它成为该文档某个现有节点的子节点，appendChild(newChild)作为要添加子节点的节点的方法被调用，将一个标识为 newChild 的节点添加到它的子节点的末尾。语法如下：

```
object.appendChild(newChild)
```

下面的实例演示如何在节点树中插入节点：

```
//源程序 5.12
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title>Sample Page!</title>
</head>
<script language="JavaScript" type="text/javascript">
<!--
```

```
function nodeStatus(node)
{
   var temp="";
   if(node.nodeName!=null)
   {
      temp+="nodeName: "+node.nodeName+"\n";
   }
   else temp+="nodeName: null!\n";
   if(node.nodeType!=null)
   {
      temp+="nodeType: "+node.nodeType+"\n";
   }
   else temp+="nodeType: null\n";
   if(node.nodeValue!=null)
   {
      temp+="nodeValue: "+node.nodeValue+"\n\n";
   }
   else temp+="nodeValue: null\n\n";
   return temp;
}
function MyTest( )
{
   //产生 p 元素节点和新文本节点，并将文本节点添加为 p 元素节点的最后一个子节点
   var newParagraph = document.createElement("p");
   var newTextNode= document.createTextNode(document.MyForm.MyField.value);
   newParagraph.appendChild(newTextNode);
   var msg=nodeStatus(newParagraph);
   msg+=nodeStatus(newTextNode);
   msg+=nodeStatus(newParagraph.firstChild);
   alert(msg);
   return;
}
//-->
</script>
<body>
<form name="MyForm">
   <input type="text" name="MyField" value="My Sample">
   <input type="button" value="TestButton" onclick="MyTest()">
</body>
</html>
```

程序运行结果如图 5.12 所示。

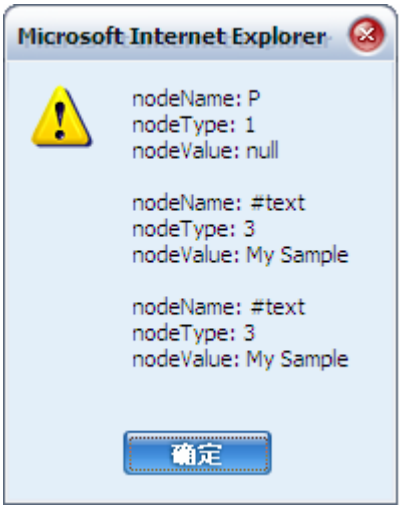

图 5.12 改写后的实例生成的元素节点和文本节点

明显看出使用 newParagraph.appendChild(newTextNode)语句后，节点 newTextNode 和节点 newParagraph.firstChild 表示同一节点，证明生成的文本节点已经添加到<p>元素节点的子节点列表中。

insertBefore(newChild,targetChild)方法将文档中一个新节点 newChild 插入到原始节点 targetChild 前面，语法如下：

```
parentElement.insertBefore(newChild,targetChild)
```

调用此方法之前，要明白三点：

- 要插入的新节点 newChild
- 目标节点 targetChild
- 这两个节点的父节点 parentElement

其中，parentElement=targetChild.parentNode，且父节点必须是元素节点。以下面的语句为例：

```
<p id="p1">Welcome to<B> DOM </B>World! </p>
```

其表示的节点树如图 5.8 所示。下面的代码演示如何在文本节点“Welcome to”之前添加一个同父文本节点“NUDT YSQ”：

```
//源程序 5.13
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title> First Page!</title>
</head>
<body>
<p id="p1">Welcome to<B> DOM </B>World! </p>
<script language="JavaScript" type="text/javascript">
<!--
function nodeStatus(node)
{
   var temp="";
   if(node.nodeName!=null)
   {
```

```
    temp+="nodeName: "+node.nodeName+"\n";
  }
  else temp+="nodeName: null!\n";
  if(node.nodeType!=null)
  {
    temp+="nodeType: "+node.nodeType+"\n";
  }
  else temp+="nodeType: null\n";
  if(node.nodeValue!=null)
  {
    temp+="nodeValue: "+node.nodeValue+"\n\n";
  }
  else temp+="nodeValue: null\n\n";
  return temp;
}
//输出节点树相关信息
//返回 id 属性值为 p1 的元素节点
var parentElement=document.getElementById('p1');
var msg="insertBefore 方法之前:\n"
msg+=nodeStatus(parentElement);
//返回 p1 的第一个孩子，即文本节点“Welcome to”
var targetElement=parentElement.firstChild;
msg+=nodeStatus(targetElement);
//返回文本节点“Welcome to”的下一个同父节点，即元素节点 B
var currentElement=targetElement.nextSibling;
msg+=nodeStatus(currentElement);
//返回元素节点 P 的最后一个孩子，即文本节点“World!”
currentElement=parentElement.lastChild;
msg+=nodeStatus(currentElement);
//生成新文本节点“NUDT YSQ”，并插入到文本节点“Welcome to”之前
var newTextNode= document.createTextNode("NUDT YSQ");
parentElement.insertBefore(newTextNode,targetElement);
msg+="insertBefore 方法之后:\n"+nodeStatus(parentElement);
//返回 p1 的第一个孩子，即文本节点“NUDT YSQ”
targetElement=parentElement.firstChild;
msg+=nodeStatus(targetElement);
//返回文本节点“Welcome to”的下一个同父节点，即元素节点“Welcome to”
var currentElement=targetElement.nextSibling;
msg+=nodeStatus(currentElement);
//返回元素节点 P 的最后一个孩子，即文本节点“World!”
currentElement=parentElement.lastChild;
msg+=nodeStatus(currentElement);
//输出节点属性
alert(msg);
//-->
</script>
</body>
</html>
```

输出信息按照父节点、第一个子节点、下一个子节点、最后一个子节点的顺序显示，程序运行结果如图 5.13 所示。

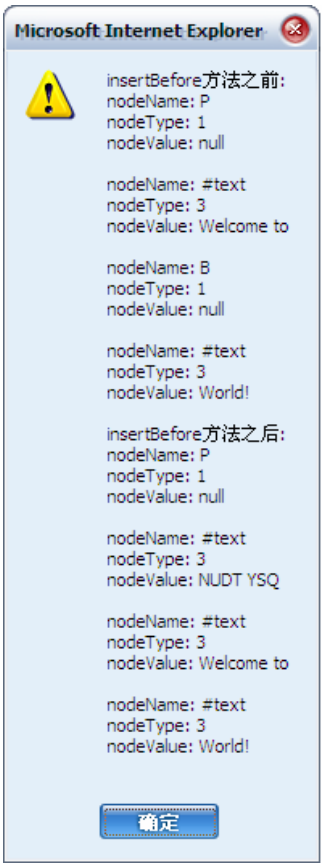

图 5.13 使用 insertBefore( )方法在目标节点之前插入新节点

可以很直观看出文本节点“Welcome to”在作为 insertBefore( )方法的目标节点后，在其前面插入文本节点“NUDT YSQ”作为<p>元素节点的第一子节点。

DOM 本身并没有提供类似 insertBefore(newChild,targetChild)方法在节点之后插入新节点方法 insertAfter(newChild,targetChild)，但是可以通过如下形式实现：

```
function insertAfter(newChild,targetChild)
{
    var parentElement=targetChild.parentNode;
    //检查目标节点是否是父节点的最后一个子节点
    //是：直接按 appendChild( )方法插入新节点
    if(parentElement.lastChild==targetChild)
    {
        parentElement.appendChild(newChild);
    }
    //不是：使用目标节点的 nextSibling 属性定位到它的下一同父节点，按 insertBefore( )方法操作
    else
        parentElement.insertBefore(newChild,targetElement.nextSibling);
}
```

将源程序 5.13 中新节点插入语句：

```
parentElement.insertBefore(newTextNode,targetElement);
```

改写为：

```
insertAfter(newTextNode,targetElement);
```

程序运行结果如图 5.14 所示。

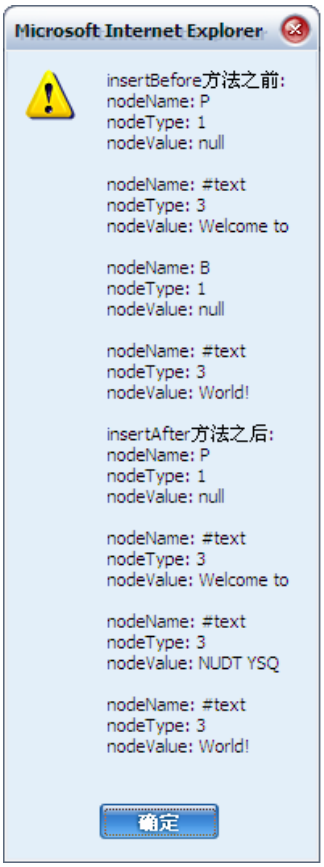

图 5.14 使用 insertAfter( )函数在目标节点之后插入新节点

可以直观看出，insertAfter( )函数实现了在目标节点后面插入同级子节点的功能，扩展了 DOM 关于节点插入和添加的方法。

5.5.3 复制节点

有时候并不需要生成或插入新的节点，而只是复制就可以达到既定的目标。DOM 提供 cloneNode( )方法来复制特定的节点，语法如下：

```
clonedNode=targetNode.cloneNode(bool)
```

其中参数 bool 为逻辑量：

- bool=1 或 true：表示复制节点自身的同时复制节点所有的子节点；
- bool=0 或 false：表示仅仅复制节点自身。

下面的实例演示使用如何复制节点并将其插入到节点树中：

```
//源程序 5.14
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title> First Page!</title>
</head>
<body>
<p id="p1">Welcome to<B> DOM </B>World! </p>
<script language="JavaScript" type="text/javascript">
```

```
<!--
function nodeStatus(node)
{
   var temp="";
   if(node.nodeName!=null)
   {
      temp+="nodeName: "+node.nodeName+"\n";
   }
   else temp+="nodeName: null!\n";
   if(node.nodeType!=null)
   {
      temp+="nodeType: "+node.nodeType+"\n";
   }
   else temp+="nodeType: null\n";
   if(node.nodeValue!=null)
   {
      temp+="nodeValue: "+node.nodeValue+"\n\n";
   }
   else temp+="nodeValue: null\n\n";
   return temp;
}
//输出节点树相关信息
//返回 id 属性值为 p1 的元素节点
var parentElement=document.getElementById('p1');
var msg="insertBefore 方法之前:\n"
msg+=nodeStatus(parentElement);
//返回 p1 的第一个孩子，即文本节点“Welcome to”
var targetElement=parentElement.firstChild;
msg+=nodeStatus(targetElement);
//返回文本节点“Welcome to”的下一个同父节点，即元素节点 B
var currentElement=targetElement.nextSibling;
msg+=nodeStatus(currentElement);
//返回元素节点 P 的最后一个孩子，即文本节点“World!”
currentElement=parentElement.lastChild;
msg+=nodeStatus(currentElement);
//通过目标节点的 cloneNode( )方法产生复制后的新节点
var ClonedNode=targetElement.cloneNode(false);
//使用父节点的 insertBefore( )方法将新节点插入到目标节点的前面
parentElement.insertBefore(ClonedNode,targetElement);
msg+="insertBefore 方法之后:\n"+nodeStatus(parentElement);
//返回 p1 的第一个孩子，即克隆的新文本节点“Welcome to”
targetElement=parentElement.firstChild;
msg+=nodeStatus(targetElement);
//返回文本节点“Welcome to”的下一个同父节点，即元素节点“Welcome to”
var currentElement=targetElement.nextSibling;
msg+=nodeStatus(currentElement);
//返回元素节点 P 的最后一个孩子，即文本节点“World!”
currentElement=parentElement.lastChild;
msg+=nodeStatus(currentElement);
//输出节点属性
alert(msg);
//-->
</script>
```

```
</body>
</html>
```

程序运行结果如图 5.15 所示。

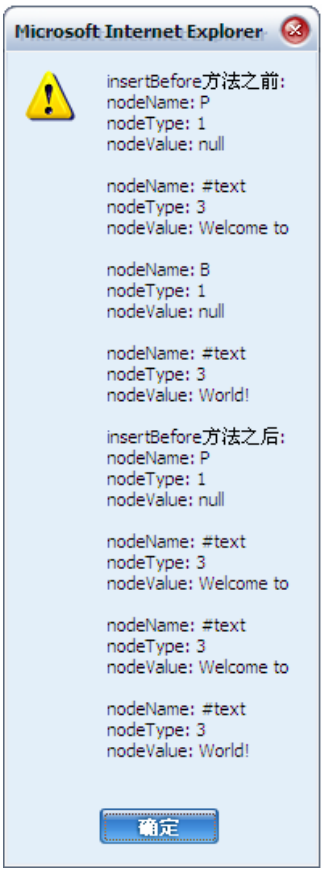

图 5.15 使用 cloneNode( )方法进行节点复制操作

注意：HTML 文档中，空元素不会改变文档的外观，因为浏览器经常对那些无内容的元素进行最小化。

5.5.4 删除和替换节点

可以在节点树中生成、添加、复制一个节点，当然也可以删除节点树中特定的节点。DOM 提供 removeChild()方法来进行删除操作，语法如下：

```
removeNode=object.removeChild(name)
```

参数 name 指明要删除的节点名称，该方法返回所删除的节点对象。

下面的实例演示如何使用 removeChild()方法删除节点：

```
//源程序 5.15
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title> First Page!</title>
</head>
<body>
<p id="p1">Welcome to<B> DOM </B>World! </p>
```

```
<script language="JavaScript" type="text/javascript">
<!--
function nodeStatus(node)
{
   var temp="";
   if(node.nodeName!=null)
   {
      temp+="nodeName: "+node.nodeName+"\n";
   }
   else temp+="nodeName: null!\n";
   if(node.nodeType!=null)
   {
      temp+="nodeType: "+node.nodeType+"\n";
   }
   else temp+="nodeType: null\n";
   if(node.nodeValue!=null)
   {
      temp+="nodeValue: "+node.nodeValue+"\n\n";
   }
   else temp+="nodeValue: null\n\n";
   return temp;
}
var parentElement=document.getElementById('p1');
var msg="父节点：\n"+nodeStatus(parentElement);
//返回元素节点 P 的最后一个孩子，即文本节点“World!”
currentElement=parentElement.lastChild;
msg+="删除前:lastChild:\n"+nodeStatus(currentElement);
//删除节点 P 的最后一个孩子，即文本节点“World!”，最后一个孩子变为 B
parentElement.removeChild(currentElement);
currentElement=parentElement.lastChild;
msg+="删除后:lastChild:\n"+nodeStatus(currentElement);
//输出节点属性
alert(msg);
//-->
</script>
</body>
</html>
```

程序运行结果如图 5.16 所示。

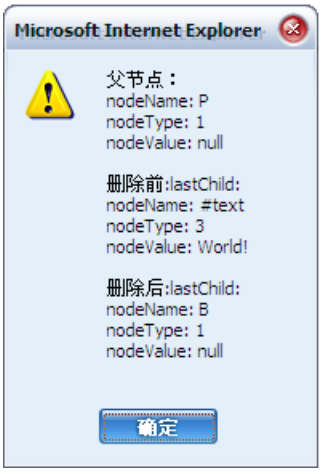

图 5.16 使用 removeChild( )方法删除子节点

DOM 中使用 replaceChild( )来替换指定的节点，语法如下：

```
object.replaceChild(newChild,oldChild)
```

其中参数：

- newChild：新添加的节点
- oldChild：被替换的目标节点

将下面的代码替换源程序 5.15 中的输出部分：

```
var parentElement=document.getElementById('p1');
var msg="父节点：\n"+nodeStatus(parentElement);
//返回元素节点 P 的最后一个孩子，即文本节点“World!”
currentElement=parentElement.lastChild;
msg+="替换前:lastChild:\n"+nodeStatus(currentElement);
//生成将用来替换的新文本节点 newTextNode
var newTextNode= document.createTextNode("NUDT YSQ");
parentElement.replaceChild(newTextNode,currentElement)
//替换文本节点“World!”为“NUDT YSQ”
currentElement=parentElement.lastChild;
msg+="替换后:lastChild:\n"+nodeStatus(currentElement);
//输出节点属性
alert(msg);
```

程序运行结果如图 5.17 所示。

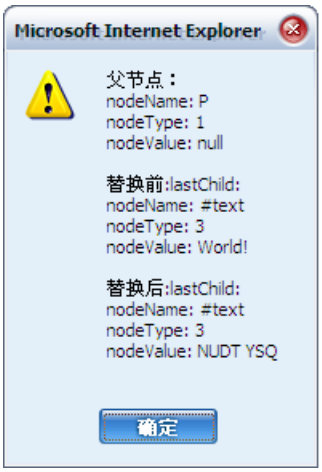

图 5.17 使用 replaceChild( )方法替换指定节点

如果只是要改变文本节点的内容，只需要给该节点的 nodeValue 属性赋一个新字符串值即可，将下面的代码替换源程序 5.15 中的输出部分：：

```
var parentElement=document.getElementById('p1');
var msg="父节点:\n"+nodeStatus(parentElement);
//返回 p1 的第一个孩子，即文本节点“Welcome to”
var targetElement=parentElement.firstChild;
msg+="nodeValue 更改之前:\n"+nodeStatus(targetElement);
//将文本节点“Welcome to”赋新值“Verify String!”
targetElement.nodeValue= "Verify String!";
//返回 p1 的第一个孩子，即新文本节点“Verify String!”
msg+="nodeValue 更改之后:\n"+nodeStatus(targetElement);
//输出节点属性
```

```
alert(msg);
```
程序运行结果如图 5.18 所示。

图 5.18 修改文本节点的 nodeValue 值更改文本内容

当节点内容完全是文本时，此方法修改节点内容最为直接。如果不存在文本节点，则不起任何作用。下面专门介绍 DOM 中文本节点的特有操作方法。

> 注意：通过 createTextNode( )方法产生的文本节点没有任何内在样式，如果要改变文本节点的外观及文本，就必须修改该文本节点的父节点的 style 属性。执行样式更改和内容变化的浏览器将自动刷新此网页，以适应文本节点样式和内容的变化。

5.5.5 文本节点操作

针对文本节点，DOM 提供了多种方法来操作其文本属性，下面的例子演示各个函数的具体使用方法：

```
//源程序 5.16
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title> First Page!</title>
</head>
<body>
<p id="p1">Congratulations</p>
<script language="JavaScript" type="text/javascript">
<!--
function nodeStatus(node)
{
   var temp="";
   if(node.length!=0)
   {
      temp+="nodeLength: "+node.length+"\n";
   }
   else temp+="nodeLength: 0\n\n";
```

```
   if(node.data!="")
   {
      temp+="nodeData: "+node.data+"\n\n";
   }
   else temp+="nodeData: null\n\n";
   return temp;
}
//返回 id 属性值为 p1 的元素节点
var parentElement=document.getElementById('p1');
//返回 p1 的第一个孩子，即文本节点“Congratulations”，并输出其信息
var targetElement=parentElement.firstChild;
var msg="●原始文本节点内容:\n"+nodeStatus(targetElement);
//使用 targetElement 的 data 属性直接修改其内容
targetElement.data="Welcome you";
msg+="●使用 targetElement.data="Welcome you":\n"+nodeStatus(targetElement);
//使用 appendData( )方法在文本节点内容后面添加字符串
targetElement.appendData(' to NUDT!');
msg+="●使用 targetElement.appendData(' to NUDT!'):\n"+nodeStatus(targetElement);
//使用 insertData( )方法在文本节点内容某个位置添加字符串
targetElement.insertData(0,'YSQ ');
msg+="●使用 targetElement.insertData(0,'YSQ '):\n"+nodeStatus(targetElement);
//使用 deleteData( )方法在文本节点内容中指定位置删除指定长度的字符串
targetElement.deleteData(0,4);
msg+="●使用 targetElement.deleteData(0,4):\n"+nodeStatus(targetElement);
//使用 replaceData( )方法用指定字符串替换文本节点内容指定位置和长度的字符串
targetElement.replaceData(0,7,'JHX Let');
msg+="●使用 targetElement.replaceData(0,7,'JHX Let'):\n"+nodeStatus(targetElement);
//使用 splitText( )方法将文本节点内容从指定位置分为两部分，左边为原始节点，右边赋值给新节点
var tempNode=targetElement.splitText(8);
msg+="●使用 targetElement.splitText(8):\n";
msg+="原始文本节点: \n"+nodeStatus(targetElement);
msg+="新的文本节点: \n"+nodeStatus(tempNode);
//使用 substringData( )方法返回文本节点内容中指定位置和长度的字符串
msg+="●使用 targetElement.substringData(0,3):\n"+targetElement.substringData(0,3);
//输出文本节点属性变化过程
alert(msg);
//-->
</script>
</body>
</html>
```

程序运行结果如图 5.19 所示。

图 5.19 DOM 中关于文本节点操作的方法实例

注意：浏览器对这些方法的支持不太规则，在完整支持 DOM Level 1 规范的浏览器中能正常工作。

5.6 对象的事件处理程序

事件由浏览器动作如浏览器载入文档或用户动作诸如敲击键盘、滚动鼠标等触发，而事件处理程序则说明一个对象如何响应事件。在早期支持 JavaScript 脚本的浏览器中，事件处理程序是作为 HTML 标记的附加属性加以定义的，其形式如下：

```
<input type="button" name="MyButton" value="Test Event" onclick="MyEvent()">
```

下面的例子演示鼠标单击时事件的触发过程：

```
//源程序 5.17
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title>Sample Page!</title>
<script language="JavaScript">
←
//MyButtonA 的 onclick 事件触发函数
function MyEventA()
{
```

```
   alert("MyButtonA is clicked!");
}
//MyButtonB 的 onclick 事件触发函数
function MyEventB()
{
   alert("MyButtonB is clicked!");
}
//-->
</script>
</head>
<body>
<form name=”MyForm” method="post" action="OtherPage.asp">
   <input type="button" name="MyButtonA" value="Test Event A" onclick="MyEventA()">
   <input type="button" name="MyButtonB" value="Test Event B" onclick="MyEventB()">
</form>
</body>
</html>
```

结果显示，鼠标单击“MyButtonA”按钮，弹出警告框“MyButtonA is clicked!”；单击“MyButtonB”按钮，弹出警告框“MyButtonB is clicked!”，可以看出这个方法模拟了事件所表示的动作。

在 DOM 中，可以通过调用对象事件处理程序的方式来触发这个事件：

```
document.MyForm.MyButtonA.onclick()
```

同时，可以将事件处理程序作为对象的属性来触发事件，例如：

```
//源程序 5.18
<!DOCTYPE HTML PUBLIC   "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<html>
<head>
<meta http-equiv=content-type content="text/html; charset=gb2312">
<title>Sample Page!</title>
<script language="JavaScript">
<!--
//将 MyEvent( )事件处理程序赋予 MyText 的 onfocus 事件
function TestEvent()
{
   document.MyForm.MyText.onfocus=MyEvent();
}
//onfocus 事件触发结果
function MyEvent()
{
   alert("MyEvent is Trigger!");
}
//-->
</script>
</head>
<body>
<form name="MyForm">
   <input type="text" name="MyText">
   <input type="button" name="MyButton" value="Test Button" onclick="TestEvent()">
</form>
</body>
</html>
```

上面代码的核心语句 document.MyForm.MyText.onfocus=MyEvent()表示将自定义的函数 MyEvent( )赋给对象的事件处理程序 onfocus( )，即给 onfocus( )赋予了节点属性的特征。

注意：由于对事件处理程序属性做的任何变化都会随着文档的重载而消失，因此建议将事件处理程序作为脚本的一部分，像方法一样调用。

5.7 浏览器兼容性策略

由于各大浏览器厂商对 DOM 和 CSS 标准支持的程度不同，导致同样的 HTML 页面在不同浏览器环境中解析不到同样的视觉界面，同时出现了有很多的 CSS 解析 bug，如 IE 盒模型 bug、IE 浮动 3px bug 等，这就是浏览器兼容性问题。

在浏览器版本不断更替的发展过程中，主要浏览器厂商都发现他们的浏览器实现的 CSS 特性与随后发布的 DOM 和 CSS 标准有所不同。为了消除浏览器之争，打破私有代码代码的不兼容性，让 Web 标准体系里的代码在所有的浏览器上都能得到正常解析，实现 Web 应用程序的跨平台性，主要浏览器厂商如 Microsoft、Mozilla 等共同商定由 Web 应用程序开发者在 HTML 页面的<head>元素中自由选择是否添加<!DOCTYPE>标记，以确定文档是遵循旧的 quirks 方式还是标准兼容模型。

可以使用如下的方法在<head>元素中包含<!DOCTYPE >标记来实现浏览器的兼容性：

```
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html140/strict.dtd">
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0 Frameset//EN"
"http://www.w3.org/TR/REC-html140/frameset.dtd">
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html140/loose.dtd">
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD XHTML 1.0//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR//xhtml1/DTD/xhtml1-frameset.dtd">
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR//xhtml1/DTD/xhtml1-transitional.dtd">
<!DOCTYPE HTML PUBLIC  "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

由于 IE5.5 以下版本的浏览器忽略了标准兼容模型，他们不适用<!DOCTYPE>标记而用旧的 quirks 方式。在目前主流浏览器中，所有基于 Mozilla、WinIE6 和 MacIE5 及以上版本的浏览器中都实现了浏览器兼容性策略，从而使用<!DOCTYPE>标记更有可能通过不同浏览器实现相同的视觉界面。

注意：关于盒模型 bug 和移动 3px bug 原理及解决方法请参考请见 W3C 中国网站：http://www.w3cn.org/

5.8 本章小结

本章介绍了 DOM（文档对象模型），它是 JavaScript 脚本与 HTML 文档、CSS 样式表之间联系的纽带。支持 DOM 的浏览器在载入 HTML 文档时按照 DOM 规范将文档节点化形成节点树，JavaScript 通过 DOM 提供的诸如 getElementById( )、removeAttribute( )等方法，可对节点树中的任何已节点化的元素进行访问和修改属性等操作，并通过 createTextNode( )、

appendChild( )等方法迅速生成新文本节点并进行相关操作，甚至动态生成指定的 HTML 文档。

各大浏览器厂商不同程度支持 DOM 规范的推广，但都没有得到很好的支持。Web 应用程序开发者在开发普适当前主流浏览器的 Web 应用程序的时候，就必须充分了解浏览器对 DOM 的支持情况，并编写组织很有条理的 HTML 文档，因为在组织很差的 HTML 文档中执行操作 DOM 的脚本代码，会出现“不可预测的结果”（W3C）。在主流浏览器全面支持 DOM 规范之前，可使用 DOM 中比较基础但得到支持的 DOM Level 1 规范。下面几章我们将集中精力讨论 JavaScript 脚本中数据类型的对象，如 String、Array、Math 等。