

## 第 10 章

# DOM

### 本章内容

- 理解包含不同层次节点的 DOM
- 使用不同的节点类型
- 克服浏览器兼容性问题及各种陷阱

**D**OM (文档对象模型) 是针对 HTML 和 XML 文档的一个 API (应用程序编程接口)。DOM 描绘了一个层次化的节点树, 允许开发人员添加、移除和修改页面的某一部分。DOM 脱胎于 Netscape 及微软公司创始的 DHTML (动态 HTML), 但现在它已经成为表现和操作页面标记的真正的跨平台、语言中立的方式。

1998 年 10 月 DOM 1 级规范成为 W3C 的推荐标准, 为基本的文档结构及查询提供了接口。本章主要讨论与浏览器中的 HTML 页面相关的 DOM1 级的特性和应用, 以及 JavaScript 对 DOM1 级的实现。IE、Firefox、Safari、Chrome 和 Opera 都非常完善地实现了 DOM。



注意, IE 中的所有 DOM 对象都是以 COM 对象的形式实现的。这意味着 IE 中的 DOM 对象与原生 JavaScript 对象的行为或活动特点并不一致。本章将较多地谈及这些差异。

## 10.1 节点层次

DOM 可以将任何 HTML 或 XML 文档描绘成一个由多层节点构成的结构。节点分为几种不同的类型, 每种类型分别表示文档中不同的信息及 (或) 标记。每个节点都拥有各自的特点、数据和方法, 另外也与其他节点存在某种关系。节点之间的关系构成了层次, 而所有页面标记则表现为一个以特定节点为根节点的树形结构。以下面的 HTML 为例:

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

可以将这个简单的 HTML 文档表示为一个层次结构, 如图 10-1 所示。

文档节点是每个文档的根节点。在这个例子中, 文档节点只有一个子节点, 即<html>元素, 我们

称之为文档元素。文档元素是文档的最外层元素，文档中的其他所有元素都包含在文档元素中。每个文档只能有一个文档元素。在 HTML 页面中，文档元素始终都是<html>元素。在 XML 中，没有预定义的元素，因此任何元素都可能成为文档元素。

每一段标记都可以通过树中的一个节点来表示：HTML 元素通过元素节点表示，特性（attribute）通过特性节点表示，文档类型通过文档类型节点表示，而注释则通过注释节点表示。总共有 12 种节点类型，这些类型都继承自一个基类型。

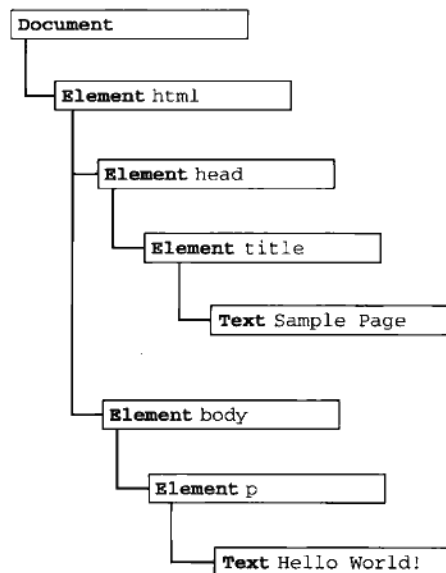


图 10-1

### 10.1.1 Node 类型

DOM1 级定义了一个 Node 接口，该接口将由 DOM 中的所有节点类型实现。这个 Node 接口在 JavaScript 中是作为 Node 类型实现的；除了 IE 之外，在其他所有浏览器中都可以访问到这个类型。JavaScript 中的所有节点类型都继承自 Node 类型，因此所有节点类型都共享着相同的基本属性和方法。

每个节点都有一个 nodeType 属性，用于表明节点的类型。节点类型由在 Node 类型中定义的下列 12 个数值常量来表示，任何节点类型必居其一：

- ❑ Node.ELEMENT\_NODE(1);
- ❑ Node.ATTRIBUTE\_NODE(2);
- ❑ Node.TEXT\_NODE(3);
- ❑ Node.CDATA\_SECTION\_NODE(4);
- ❑ Node.ENTITY\_REFERENCE\_NODE(5);
- ❑ Node.ENTITY\_NODE(6);
- ❑ Node.PROCESSING\_INSTRUCTION\_NODE(7);
- ❑ Node.COMMENT\_NODE(8);

```
❑ Node.DOCUMENT_NODE(9);  
❑ Node.DOCUMENT_TYPE_NODE(10);  
❑ Node.DOCUMENT_FRAGMENT_NODE(11);  
❑ Node.NOTATION_NODE(12);
```

通过比较上面这些常量，可以很容易地确定节点的类型，例如：

```
if (someNode.nodeType == Node.ELEMENT_NODE){    //在 IE 中无效  
    alert("Node is an element.");  
}
```

这个例子比较了 `someNode.nodeType` 与 `Node.ELEMENT_NODE` 常量。如果二者相等，则意味着 `someNode` 确实是一个元素。然而，由于 IE 没有公开 `Node` 类型的构造函数，因此上面的代码在 IE 中会导致错误。为了确保跨浏览器兼容，最好还是将 `nodeType` 属性与数字值进行比较，如下所示：

```
if (someNode.nodeType == 1){    //适用于所有浏览器  
    alert("Node is an element.");  
}
```

并不是所有节点类型都受到 Web 浏览器的支持。开发人员最常用的就是元素和文本节点。本章后面将详细讨论每个节点类型的受支持情况及使用方法。

### 1. nodeName 和 nodeValue 属性

要了解节点的具体信息，可以使用 `nodeName` 和 `nodeValue` 这两个属性。这两个属性的值完全取决于节点的类型。在使用这两个值以前，最好是像下面这样先检测一下节点的类型。

```
if (someNode.nodeType == 1){  
    value = someNode.nodeName;    //nodeName 的值是元素的标签名  
}
```

在这个例子中，首先检查节点类型，看它是不是一个元素。如果是，则取得并保存 `nodeName` 的值。对于元素节点，`nodeName` 中保存的始终是元素的标签名，而 `nodeValue` 的值则始终为 `null`。

### 2. 节点关系

文档中所有的节点之间都存在这样或那样的关系。节点间的各种关系可以用传统的家族关系来描述，相当于把文档树比喻成家谱。在 HTML 中，可以将 `<body>` 元素看成是 `<html>` 元素的子元素；相应地，也就可以将 `<html>` 元素看成是 `<body>` 元素的父元素。而 `<head>` 元素，则可以看成是 `<body>` 元素的同胞元素，因为它们都是同一个父元素 `<html>` 的直接子元素。

每个节点都有一个 `childNodes` 属性，其中保存着一个 `NodeList` 对象。`NodeList` 是一种类数组对象，用于保存一组有序的节点，可以通过位置来访问这些节点。请注意，虽然可以通过方括号语法来访问 `NodeList` 的值，而且这个对象也有 `length` 属性，但它并不是 `Array` 的实例。`NodeList` 对象的独特之处在于，它实际上是基于 DOM 结构动态执行查询的结果，因此 DOM 结构的变化能够自动反映在 `NodeList` 对象中。我们常说，`NodeList` 是有生命、有呼吸的对象，而不是在我们第一次访问它们的某个瞬间拍摄下来的一张快照。

下面的例子展示了如何访问保存在 `NodeList` 中的节点——可以通过方括号，也可以使用 `item()` 方法。

```
var firstChild = someNode.childNodes[0];  
var secondChild = someNode.childNodes.item(1);  
var count = someNode.childNodes.length;
```

无论使用方括号还是使用 `item()` 方法都没有问题，但使用方括号语法看起来与访问数组相似，因此颇受一些开发人员的青睐。另外，要注意 `length` 属性表示的是访问 `NodeList` 的那一刻，其中包含的节点数量。我们在本书前面介绍过，对 `arguments` 对象使用 `Array.prototype.slice()` 方法可以将其转换为数组。而采用同样的方法，也可以将 `NodeList` 对象转换为数组。来看下面的例子：

```
//在 IE8 及之前版本中无效
var arrayOfNodes = Array.prototype.slice.call(someNode.childNodes,0);
```

除 IE8 及更早版本之外，这行代码能在任何浏览器中运行。由于 IE8 及更早版本将 `NodeList` 实现为一个 COM 对象，而我们不能像使用 JavaScript 对象那样使用这种对象，因此上面的代码会导致错误。要想在 IE 中将 `NodeList` 转换为数组，必须手动枚举所有成员。下列代码在所有浏览器中都可以运行：

```
function convertToArray(nodes){
    var array = null;
    try {
        array = Array.prototype.slice.call(nodes, 0); //针对非 IE 浏览器
    } catch (ex) {
        array = new Array();
        for (var i=0, len=nodes.length; i < len; i++){
            array.push(nodes[i]);
        }
    }

    return array;
}
```

这个 `convertToArray()` 函数首先尝试了创建数组的最简单方式。如果导致了错误（说明是在 IE8 及更早版本中），则通过 `try-catch` 块来捕获错误，然后手动创建数组。这是另一种检测怪癖的形式。

每个节点都有一个 `parentNode` 属性，该属性指向文档树中的父节点。包含在 `childNodes` 列表中的所有节点都具有相同的父节点，因此它们的 `parentNode` 属性都指向同一个节点。此外，包含在 `childNodes` 列表中的每个节点相互之间都是同胞节点。通过使用列表中每个节点的 `previousSibling` 和 `nextSibling` 属性，可以访问同一列表中的其他节点。列表中第一个节点的 `previousSibling` 属性值为 `null`，而列表中最后一个节点的 `nextSibling` 属性的值同样也为 `null`，如下面的例子所示：

```
if (someNode.nextSibling === null){
    alert("Last node in the parent's childNodes list.");
} else if (someNode.previousSibling === null){
    alert("First node in the parent's childNodes list.");
}
```

当然，如果列表中只有一个节点，那么该节点的 `nextSibling` 和 `previousSibling` 都为 `null`。

父节点与其第一个和最后一个子节点之间也存在特殊关系。父节点的 `firstChild` 和 `lastChild` 属性分别指向其 `childNodes` 列表中的第一个和最后一个节点。其中，`someNode.firstChild` 的值始终等于 `someNode.childNodes[0]`，而 `someNode.lastChild` 的值始终等于 `someNode.childNodes[someNode.childNodes.length-1]`。在只有一个子节点的情况下，`firstChild` 和 `lastChild` 指向同一个节点。如果没有子节点，那么 `firstChild` 和 `lastChild` 的值均为 `null`。明确这些关系能够对我们查找和访问文档结构中的节点提供极大的便利。图 10-2 形象地展示了上述关系。

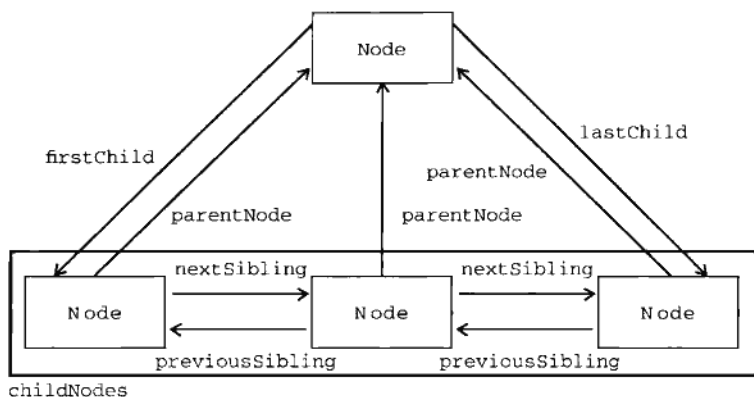


图 10-2

在反映这些关系的所有属性当中，childNodes 属性与其他属性相比更方便一些，因为只须使用简单的关系指针，就可以通过它访问文档树中的任何节点。另外，hasChildNodes() 也是一个非常有用的方法，这个方法在节点包含一或多个子节点的情况下返回 true；应该说，这是比查询 childNodes 列表的 length 属性更简单的方法。

所有节点都有的最后一个属性是 ownerDocument，该属性指向表示整个文档的文档节点。这种关系表示的是任何节点都属于它所在的文档，任何节点都不能同时存在于两个或更多个文档中。通过这个属性，我们可以不必在节点层次中通过层层回溯到达顶端，而是可以直接访问文档节点。



虽然所有节点类型都继承自 Node，但并不是每种节点都有子节点。本章后面将会讨论不同节点类型之间的差异。

### 3. 操作节点

因为关系指针都是只读的，所以 DOM 提供了一些操作节点的方法。其中，最常用的方法是 appendChild()，用于向 childNodes 列表的末尾添加一个节点。添加节点后，childNodes 的新增节点、父节点及以前的最后一个子节点的关系指针都会相应地得到更新。更新完成后，appendChild() 返回新增的节点。来看下面的例子：

```
var returnedNode = someNode.appendChild(newNode);  
alert(returnedNode == newNode); //true  
alert(someNode.lastChild == newNode); //true
```

如果传入到 appendChild() 中的节点已经是文档的一部分了，那结果就是将该节点从原来的位置转移到新位置。即使可以将 DOM 树看成是由一系列指针连接起来的，但任何 DOM 节点也不能同时出现在文档中的多个位置上。因此，如果在调用 appendChild() 时传入了父节点的第一个子节点，那么该节点就会成为父节点的最后一个子节点，如下面的例子所示。

```
//someNode 有多个子节点  
var returnedNode = someNode.appendChild(someNode.firstChild);  
alert(returnedNode == someNode.firstChild); //false  
alert(returnedNode == someNode.lastChild); //true
```



如果需要把节点放在 `childNodes` 列表中某个特定的位置上, 而不是放在末尾, 那么可以使用 `insertBefore()` 方法。这个方法接受两个参数: 要插入的节点和作为参照的节点。插入节点后, 被插入的节点会变成参照节点的前一个同胞节点 (`previousSibling`), 同时被方法返回。如果参照节点是 `null`, 则 `insertBefore()` 与 `appendChild()` 执行相同的操作, 如下面的例子所示。

```
//插入后成为最后一个子节点
returnedNode = someNode.insertBefore(newNode, null);
alert(newNode == someNode.lastChild); //true

//插入后成为第一个子节点
var returnedNode = someNode.insertBefore(newNode, someNode.firstChild);
alert(returnedNode == newNode); //true
alert(newNode == someNode.firstChild); //true

//插入到最后一个子节点前面
returnedNode = someNode.insertBefore(newNode, someNode.lastChild);
alert(newNode == someNode.childNodes[someNode.childNodes.length-2]); //true
```

前面介绍的 `appendChild()` 和 `insertBefore()` 方法都只插入节点, 不会移除节点。而下面要介绍的 `replaceChild()` 方法接受的两个参数是: 要插入的节点和要替换的节点。要替换的节点将由这个方法返回并从文档树中被移除, 同时由要插入的节点占据其位置。来看下面的例子。

```
//替换第一个子节点
var returnedNode = someNode.replaceChild(newNode, someNode.firstChild);

//替换最后一个子节点
returnedNode = someNode.replaceChild(newNode, someNode.lastChild);
```

在使用 `replaceChild()` 插入一个节点时, 该节点的所有关系指针都会从被它替换的节点复制过来。尽管从技术上讲, 被替换的节点仍然还在文档中, 但它在文档中已经没有了位置。

如果只想移除而非替换节点, 可以使用 `removeChild()` 方法。这个方法接受一个参数, 即要移除的节点。被移除的节点将成为方法的返回值, 如下面的例子所示。

```
//移除第一个子节点
var formerFirstChild = someNode.removeChild(someNode.firstChild);

//移除最后一个子节点
var formerLastChild = someNode.removeChild(someNode.lastChild);
```

与使用 `replaceChild()` 方法一样, 通过 `removeChild()` 移除的节点仍然为文档所有, 只不过在文档中已经没有了位置。

前面介绍的四个方法操作的都是某个节点的子节点, 也就是说, 要使用这几个方法必须先取得父节点 (使用 `parentNode` 属性)。另外, 并不是所有类型的节点都有子节点, 如果在不支持子节点的节点上调用了这些方法, 将会导致错误发生。

#### 4. 其他方法

有两个方法是所有类型的节点都有的。第一个就是 `cloneNode()`, 用于创建调用这个方法的节点的一个完全相同的副本。`cloneNode()` 方法接受一个布尔值参数, 表示是否执行深复制。在参数为 `true` 的情况下, 执行深复制, 也就是复制节点及其整个子节点树; 在参数为 `false` 的情况下, 执行浅复制, 即只复制节点本身。复制后返回的节点副本属于文档所有, 但并没有为它指定父节点。因此, 这个节点副本就成为了一个“孤儿”, 除非通过 `appendChild()`、`insertBefore()` 或 `replaceChild()` 将它

添加到文档中。例如，假设有下列的 HTML 代码。

```
<ul>
  <li>item 1</li>
  <li>item 2</li>
  <li>item 3</li>
</ul>
```

如果我们已经将<ul>元素的引用保存在了变量 `myList` 中，那么通常下列代码就可以看出使用 `cloneNode()` 方法的两种模式。

```
var deepList = myList.cloneNode(true);
alert(deepList.childNodes.length);    //3 (IE < 9) 或 7 (其他浏览器)

var shallowList = myList.cloneNode(false);
alert(shallowList.childNodes.length); //0
```

在这个例子中，`deepList` 中保存着一个对 `myList` 执行深复制得到的副本。因此，`deepList` 中包含 3 个列表项，每个列表项中都包含文本。而变量 `shallowList` 中保存着对 `myList` 执行浅复制得到的副本，因此它不包含子节点。`deepList.childNodes.length` 中的差异主要是因为 IE8 及更早版本与其他浏览器处理空白字符的方式不一样。IE9 之前的版本不会为空白符创建节点。



`cloneNode()` 方法不会复制添加到 DOM 节点中的 JavaScript 属性，例如事件处理程序等。这个方法只复制特性、（在明确指定的情况下也复制）子节点，其他一切都不会复制。IE 在此存在一个 bug，即它会复制事件处理程序，所以我们建议在复制之前最好先移除事件处理程序。

我们要介绍的最后一个方法是 `normalize()`，这个方法唯一的作用就是处理文档树中的文本节点。由于解析器的实现或 DOM 操作等原因，可能会出现文本节点不包含文本，或者接连出现两个文本节点的情况。当在某个节点上调用这个方法时，就会在该节点的后代节点中查找上述两种情况。如果找到了空文本节点，则删除它；如果找到相邻的文本节点，则将它们合并为一个文本节点。本章后面还将进一步讨论这个方法。

## 10.1.2 Document 类型

JavaScript 通过 `Document` 类型表示文档。在浏览器中，`document` 对象是 `HTMLDocument`（继承自 `Document` 类型）的一个实例，表示整个 HTML 页面。而且，`document` 对象是 `window` 对象的一个属性，因此可以将其作为全局对象来访问。`Document` 节点具有下列特征：

- ☐ `nodeType` 的值为 9；
- ☐ `nodeName` 的值为 "#document"；
- ☐ `nodeValue` 的值为 `null`；
- ☐ `parentNode` 的值为 `null`；
- ☐ `ownerDocument` 的值为 `null`；
- ☐ 其子节点可能是一个 `DocumentType`（最多一个）、`Element`（最多一个）、`ProcessingInstruction` 或 `Comment`。

Document 类型可以表示 HTML 页面或者其他基于 XML 的文档。不过，最常见的应用还是作为 HTMLDocument 实例的 document 对象。通过这个文档对象，不仅可以取得与页面有关的信息，而且还能操作页面的外观及其底层结构。



在 Firefox、Safari、Chrome 和 Opera 中，可以通过脚本访问 Document 类型的构造函数和原型。但在所有浏览器中都可以访问 HTMLDocument 类型的构造函数和原型，包括 IE8 及后续版本。

## 1. 文档的子节点

虽然 DOM 标准规定 Document 节点子节点可以是 DocumentType、Element、ProcessingInstruction 或 Comment，但还有两个内置的访问其子节点的快捷方式。第一个就是 documentElement 属性，该属性始终指向 HTML 页面中的 <html> 元素。另一个就是通过 childNodes 列表访问文档元素，但通过 documentElement 属性则能更快捷、更直接地访问该元素。以下面这个简单的页面为例。

```
<html>
  <body>

  </body>
</html>
```

这个页面在经过浏览器解析后，其文档中只包含一个子节点，即 <html> 元素。可以通过 documentElement 或 childNodes 列表来访问这个元素，如下所示。

```
var html = document.documentElement;    //取得对<html>的引用
alert(html === document.childNodes[0]); //true
alert(html === document.firstChild);    //true
```

这个例子说明，documentElement、firstChild 和 childNodes[0] 的值相同，都指向 <html> 元素。

作为 HTMLDocument 的实例，document 对象还有一个 body 属性，直接指向 <body> 元素。因为开发人员经常要使用这个元素，所以 document.body 在 JavaScript 代码中出现的频率非常高，其用法如下。

```
var body = document.body;    //取得对<body>的引用
```

所有浏览器都支持 document.documentElement 和 document.body 属性。

Document 另一个可能的子节点是 DocumentType。通常将 <!DOCTYPE> 标签看成一个与文档其他部分不同的实体，可以通过 doctype 属性（在浏览器中是 document.doctype）来访问它的信息。

```
var doctype = document.doctype;    //取得对<!DOCTYPE>的引用
```

浏览器对 document.doctype 的支持差别很大，可以给出如下总结。

- ❑ IE8 及之前版本：如果存在文档类型声明，会将其错误地解释为一个注释并把它当作 Comment 节点；而 document.doctype 的值始终为 null。
- ❑ IE9+ 及 Firefox：如果存在文档类型声明，则将其作为文档的第一个子节点；document.doctype 是一个 DocumentType 节点，也可以通过 document.firstChild 或 document.childNodes[0] 访问同一个节点。
- ❑ Safari、Chrome 和 Opera：如果存在文档类型声明，则将其解析，但不作为文档的子节点。document.doctype 是一个 DocumentType 节点，但该节点不会出现在 document.childNodes 中。



由于浏览器对 `document.doctype` 的支持不一致，因此这个属性的用处很有限。

从技术上说，出现在 `<html>` 元素外部的注释应该算是文档的子节点。然而，不同的浏览器在是否解析这些注释以及能否正确处理它们等方面，也存在很大差异。以下面简单的 HTML 页面为例。

```
<!-- 第一条注释 -->
<html>
  <body>

  </body>
</html>
<!-- 第二条注释 -->
```

看起来这个页面应该有 3 个子节点：注释、`<html>` 元素、注释。从逻辑上讲，我们会认为 `document.childNodes` 中应该包含与这 3 个节点对应的 3 项。但是，现实中的浏览器在处理位于 `<html>` 外部的注释方面存在如下差异。

- ❑ IE8 及之前版本、Safari 3.1 及更高版本、Opera 和 Chrome 只为第一条注释创建节点，不为第二条注释创建节点。结果，第一条注释就会成为 `document.childNodes` 中的第一个子节点。
- ❑ IE9 及更高版本会将第一条注释创建为 `document.childNodes` 中的一个注释节点，也会将第二条注释创建为 `document.childNodes` 中的注释子节点。
- ❑ Firefox 以及 Safari 3.1 之前的版本会完全忽略这两条注释。

同样，浏览器间的这种不一致性也导致了位于 `<html>` 元素外部的注释没有什么用处。

多数情况下，我们都用不着在 `document` 对象上调用 `appendChild()`、`removeChild()` 和 `replaceChild()` 方法，因为文档类型（如果存在的话）是只读的，而且它只能有一个元素子节点（该节点通常早就已经存在了）。

## 2. 文档信息

作为 `HTMLDocument` 的一个实例，`document` 对象还有一些标准的 `Document` 对象所没有的属性。这些属性提供了 `document` 对象所表现的网页的一些信息。其中第一个属性就是 `title`，包含着 `<title>` 元素中的文本——显示在浏览器窗口的标题栏或标签页上。通过这个属性可以取得当前页面的标题，也可以修改当前页面的标题并反映在浏览器的标题栏中。修改 `title` 属性的值不会改变 `<title>` 元素。来看下面的例子。

```
//取得文档标题
var originalTitle = document.title;

//设置文档标题
document.title = "New page title";
```

接下来要介绍的 3 个属性都与对网页的请求有关，它们是 `URL`、`domain` 和 `referrer`。`URL` 属性中包含页面完整的 `URL`（即地址栏中显示的 `URL`），`domain` 属性中只包含页面的域名，而 `referrer` 属性中则保存着链接到当前页面的那个页面的 `URL`。在没有来源页面的情况下，`referrer` 属性中可能会包含空字符串。所有这些信息都存在于请求的 `HTTP` 头部，只不过是通过这些属性让我们能够在 `JavaScript` 中访问它们而已，如下面的例子所示。

```
//取得完整的 URL
var url = document.URL;

//取得域名
```

```
var domain = document.domain;  
  
//取得来源页面的 URL  
var referrer = document.referrer;
```

URL 与 domain 属性是相互关联的。例如,如果 document.URL 等于 http://www.wrox.com/WileyCDA/, 那么 document.domain 就等于 www.wrox.com。

在这 3 个属性中,只有 domain 是可以设置的。但由于安全方面的限制,也并非可以给 domain 设置任何值。如果 URL 中包含一个子域名,例如 p2p.wrox.com,那么就只能将 domain 设置为“wrox.com”(URL 中包含“www”,如 www.wrox.com 时,也是如此)。不能将这个属性设置为 URL 中不包含的域,如下面的例子所示。

```
//假设页面来自 p2p.wrox.com 域  
  
document.domain = "wrox.com";           // 成功  
  
document.domain = "nczonline.net";       // 出错!
```

当页面中包含来自其他子域的框架或内嵌框架时,能够设置 document.domain 就非常方便了。由于跨域安全限制,来自不同子域的页面无法通过 JavaScript 通信。而通过将每个页面的 document.domain 设置为相同的值,这些页面就可以互相访问对方包含的 JavaScript 对象了。例如,假设有一个页面加载自 www.wrox.com,其中包含一个内嵌框架,框架内的页面加载自 p2p.wrox.com。由于 document.domain 字符串不一样,内外两个页面之间无法相互访问对方的 JavaScript 对象。但如果将这两个页面的 document.domain 值都设置为“wrox.com”,它们之间就可以通信了。

浏览器对 domain 属性还有一个限制,即如果域名一开始是“松散的”(loose),那么不能将它再设置为“紧绷的”(tight)。换句话说,在将 document.domain 设置为“wrox.com”之后,就不能再将其设置回“p2p.wrox.com”,否则将会导致错误,如下面的例子所示。

```
//假设页面来自于 p2p.wrox.com 域  
  
document.domain = "wrox.com";           //松散的(成功)  
  
document.domain = "p2p.wrox.com";       //紧绷的(出错!)
```

所有浏览器中都存在这个限制,但 IE8 是实现这一限制的最早的 IE 版本。

### 3. 查找元素

说到最常见的 DOM 应用,恐怕就要数取得特定的某个或某组元素的引用,然后再执行一些操作了。取得元素的操作可以使用 document 对象的几个方法来完成。其中,Document 类型为此提供了两个方法:getElementById() 和 getElementsByTagName()。

第一个方法,getElementById(),接收一个参数:要取得的元素的 ID。如果找到相应的元素则返回该元素,如果不存在带有相应 ID 的元素,则返回 null。注意,这里的 ID 必须与页面中元素的 id 特性(attribute)严格匹配,包括大小写。以下面的元素为例。

```
<div id="myDiv">Some text</div>
```

可以使用下面的代码取得这个元素:

```
var div = document.getElementById("myDiv");           //取得<div>元素的引用
```

但是，下面的代码在除 IE7 及更早版本之外的所有浏览器中都将返回 null。

```
var div = document.getElementById("mydiv"); //无效的 ID (在 IE7 及更早版本中可以)
```

IE8 及较低版本不区分 ID 的大小写，因此 "myDiv" 和 "mydiv" 会被当作相同的元素 ID。

如果页面中多个元素的 ID 值相同，getElementById() 只返回文档中第一次出现的元素。IE7 及较低版本还为此方法添加了一个有意思的“怪癖”：name 特性与给定 ID 匹配的表单元素 (<input>、<textarea>、<button>及<select>) 也会被该方法返回。如果有哪个表单元素的 name 特性等于指定的 ID，而且该元素在文档中位于带有给定 ID 的元素前面，那么 IE 就会返回那个表单元素。来看下面的例子。

```
<input type="text" name="myElement" value="Text field">
<div id="myElement">A div</div>
```

基于这段 HTML 代码，在 IE7 中调用 document.getElementById("myElement")，结果会返回 <input> 元素；而在其他所有浏览器中，都会返回对 <div> 元素的引用。为了避免 IE 中存在的这个问题，最好的办法是不让表单字段的 name 特性与其他元素的 ID 相同。

另一个常用于取得元素引用的方法是 getElementsByTagName()。这个方法接受一个参数，即要取得元素的标签名，而返回的是包含零或多个元素的 NodeList。在 HTML 文档中，这个方法会返回一个 HTMLCollection 对象，作为一个“动态”集合，该对象与 NodeList 非常类似。例如，下列代码会取得页面中所有的 <img> 元素，并返回一个 HTMLCollection。

```
var images = document.getElementsByTagName("img");
```

这行代码会将一个 HTMLCollection 对象保存在 images 变量中。与 NodeList 对象类似，可以使用方括号语法或 item() 方法来访问 HTMLCollection 对象中的项。而这个对象中元素的数量则可以通过其 length 属性取得，如下面的例子所示。

```
alert(images.length); //输出图像的数量
alert(images[0].src); //输出第一个图像元素的 src 特性
alert(images.item(0).src); //输出第一个图像元素的 src 特性
```

HTMLCollection 对象还有一个方法，叫做 namedItem()，使用这个方法可以通过元素的 name 特性取得集合中的项。例如，假设上面提到的页面中包含如下 <img> 元素：

```

```

那么就可以通过如下方式从 images 变量中取得这个 <img> 元素：

```
var myImage = images.namedItem("myImage");
```

在提供按索引访问项的基础上，HTMLCollection 还支持按名称访问项，这就为我们取得实际想要的元素提供了便利。而且，对命名的项也可以使用方括号语法来访问，如下所示：

```
var myImage = images["myImage"];
```

对 HTMLCollection 而言，我们可以向方括号中传入数值或字符串形式的索引值。在后台，对数值索引就会调用 item()，而对字符串索引就会调用 namedItem()。

要想取得文档中的所有元素，可以向 getElementsByTagName() 中传入 "\*"。在 JavaScript 及 CSS 中，星号 (\*) 通常表示“全部”。下面看一个例子。

```
var allElements = document.getElementsByTagName("*");
```

仅此一行代码返回的 `HTMLCollection` 中，就包含了整个页面中的所有元素——按照它们出现的先后顺序。换句话说，第一项是 `<html>` 元素，第二项是 `<head>` 元素，以此类推。由于 IE 将注释 (Comment) 实现为元素 (Element)，因此在 IE 中调用 `getElementsByName("*")` 将会返回所有注释节点。



虽然标准规定标签名需要区分大小写，但为了最大限度地与既有 HTML 页面兼容，传给 `getElementsByName()` 的标签名是不需要区分大小写的。但对于 XML 页面而言（包括 XHTML），`getElementsByName()` 方法就会区分大小写。

第三个方法，也只有 `HTMLDocument` 类型才有的方法，是 `getElementsByName()`。顾名思义，这个方法会返回带有给定 `name` 特性的所有元素。最常使用 `getElementsByName()` 方法的情况是取得单选按钮；为了确保发送给浏览器的值正确无误，所有单选按钮必须具有相同的 `name` 特性，如下面的例子所示。

```
<fieldset>
  <legend>Which color do you prefer?</legend>
  <ul>
    <li><input type="radio" value="red" name="color" id="colorRed">
      <label for="colorRed">Red</label></li>
    <li><input type="radio" value="green" name="color" id="colorGreen">
      <label for="colorGreen">Green</label></li>
    <li><input type="radio" value="blue" name="color" id="colorBlue">
      <label for="colorBlue">Blue</label></li>
  </ul>
</fieldset>
```

如这个例子所示，其中所有单选按钮的 `name` 特性值都是 "color"，但它们的 ID 可以不同。ID 的作用在于将 `<label>` 元素应用到每个单选按钮，而 `name` 特性则用以确保三个值中只有一个被发送给浏览器。这样，我们就可以使用如下代码取得所有单选按钮：

```
var radios = document.getElementsByName("color");
```

与 `getElementsByName()` 类似，`getElementsByName()` 方法也会返回一个 `HTMLCollection`。但是，对于这里的单选按钮来说，`namedItem()` 方法则只会取得第一项（因为每一项的 `name` 特性都相同）。

#### 4. 特殊集合

除了属性和方法，`document` 对象还有一些特殊的集合。这些集合都是 `HTMLCollection` 对象，为访问文档常用的部分提供了快捷方式，包括：

- ❑ `document.anchors`，包含文档中所有带 `name` 特性的 `<a>` 元素；
- ❑ `document.applets`，包含文档中所有的 `<applet>` 元素，因为不再推荐使用 `<applet>` 元素，所以这个集合已经不建议使用了；
- ❑ `document.forms`，包含文档中所有的 `<form>` 元素，与 `document.getElementsByTagName("form")` 得到的结果相同；
- ❑ `document.images`，包含文档中所有的 `<img>` 元素，与 `document.getElementsByTagName("img")` 得到的结果相同；
- ❑ `document.links`，包含文档中所有带 `href` 特性的 `<a>` 元素。

这个特殊集合始终都可以通过 `HTMLDocument` 对象访问到，而且，与 `HTMLCollection` 对象类似，集合中的项也会随着当前文档内容的更新而更新。

## 5. DOM 一致性检测

由于 DOM 分为多个级别，也包含多个部分，因此检测浏览器实现了 DOM 的哪些部分就十分必要了。document.implementation 属性就是为此提供相应信息和功能的对象，与浏览器对 DOM 的实现直接对应。DOM1 级只为 document.implementation 规定了一个方法，即 hasFeature()。这个方法接受两个参数：要检测的 DOM 功能的名称及版本号。如果浏览器支持给定名称和版本的功能，则该方法返回 true，如下面的例子所示：

```
var hasXmlDom = document.implementation.hasFeature("XML", "1.0");
```

下表列出了可以检测的不同的值及版本号。

功 能	版 本 号	说 明
Core	1.0、2.0、3.0	基本的DOM，用于描述表现文档的节点树
XML	1.0、2.0、3.0	Core的XML扩展，添加了对CDATA、处理指令及实体的支持
HTML	1.0、2.0	XML的HTML扩展，添加了对HTML特有元素及实体的支持
Views	2.0	基于某些样式完成文档的格式化
StyleSheets	2.0	将样式表关联到文档
CSS	2.0	对层叠样式表1级的支持
CSS2	2.0	对层叠样式表2级的支持
Events	2.0、3.0	常规的DOM事件
UIEvents	2.0、3.0	用户界面事件
MouseEvents	2.0、3.0	由鼠标引发的事件（click、mouseover等）
MutationEvents	2.0、3.0	DOM树变化时引发的事件
HTMLEvents	2.0	HTML4.01事件
Range	2.0	用于操作DOM树中某个范围的对象和方法
Traversal	2.0	遍历DOM树的方法
LS	3.0	文件与DOM树之间的同步加载和保存
LS-Async	3.0	文件与DOM树之间的异步加载和保存
Validation	3.0	在确保有效的前提下修改DOM树的方法

尽管使用 hasFeature() 确实方便，但也有缺点。因为实现者可以自行决定是否与 DOM 规范的不同部分保持一致。事实上，要想让 hasFeature() 方法针对所有值都返回 true 很容易，但返回 true 有时候也不意味着实现与规范一致。例如，Safari 2.x 及更早版本会在没有完全实现某些 DOM 功能的情况下也返回 true。为此，我们建议多数情况下，在使用 DOM 的某些特殊的功能之前，最好除了检测 hasFeature() 之外，还同时使用能力检测。

## 6. 文档写入

有一个 document 对象的功能已经存在很多年了，那就是将输出流写入到网页中的能力。这个能力体现在下列 4 个方法中：write()、writeln()、open() 和 close()。其中，write() 和 writeln() 方法都接受一个字符串参数，即要写入到输出流中的文本。write() 会原样写入，而 writeln() 则会在字符串的末尾添加一个换行符（\n）。在页面被加载的过程中，可以使用这两个方法向页面中动态地加入内容，如下面的例子所示。





```
<html>
<head>
  <title>document.write() Example</title>
</head>
<body>
  <p>The current date and time is:
  <script type="text/javascript">
    document.write("<strong>" + (new Date()).toString() + "</strong>");
  </script>
  </p>
</body>
</html>
```

*DocumentWriteExample01.htm*

这个例子展示了在页面加载过程中输出当前日期和时间的代码。其中，日期被包含在一个<strong>元素中，就像在 HTML 页面中包含普通的文本一样。这样做会创建一个 DOM 元素，而且可以在将来访问该元素。通过 write() 和 writeln() 输出的任何 HTML 代码都将如此处理。

此外，还可以使用 write() 和 writeln() 方法动态地包含外部资源，例如 JavaScript 文件等。在包含 JavaScript 文件时，必须注意不能像下面的例子那样直接包含字符串 "</script>"，因为这会导致该字符串被解释为脚本块的结束，它后面的代码将无法执行。



```
<html>
<head>
  <title>document.write() Example 2</title>
</head>
<body>
  <script type="text/javascript">
    document.write("<script type=\"text/javascript\" src=\"file.js\">" +
      "</script>");
  </script>
</body>
</html>
```

*DocumentWriteExample02.htm*

即使这个文件看起来没错，但字符串 "</script>" 将被解释为与外部的 <script> 标签匹配，结果文本"); 将会出现在页面中。为避免这个问题，只需把这个字符串分开写即可；第 2 章也曾经提及这个问题，解决方案如下。

```
<html>
<head>
  <title>document.write() Example 3</title>
</head>
<body>
  <script type="text/javascript">
    document.write("<script type=\"text/javascript\" src=\"file.js\">" +
      "<\\script>");
  </script>
</body>
</html>
```

*DocumentWriteExample03.htm*

字符串"</script>"不会被当作外部<script>标签的关闭标签,因而页面中也就不会出现多余的内容了。

前面的例子使用 document.write() 在页面被呈现的过程中直接向其中输出了内容。如果在文档加载结束后再调用 document.write(), 那么输出的内容将会重写整个页面, 如下面的例子所示:

```
<html>
<head>
  <title>document.write() Example 4</title>
</head>
<body>
  <p>This is some content that you won't get to see because it will be overwritten.</p>
  <script type="text/javascript">
    window.onload = function(){
      document.write("Hello world!");
    };
  </script>
</body>
</html>
```

*DocumentWriteExample04.htm*

在这个例子中, 我们使用了 window.onload 事件处理程序 (事件将在第 13 章讨论), 等到页面完全加载之后延迟执行函数。函数执行之后, 字符串 "Hello world!" 会重写整个页面内容。

方法 open() 和 close() 分别用于打开和关闭网页的输出流。如果是在页面加载期间使用 write() 或 writeln() 方法, 则不需要用到这两个方法。



严格型 XHTML 文档不支持文档写入。对于那些按照 application/xml+xml 内容类型提供的页面, 这两个方法也同样无效。

### 10.1.3 Element 类型

除了 Document 类型之外, Element 类型就要算是 Web 编程中最常用的类型了。Element 类型用于表现 XML 或 HTML 元素, 提供了对元素标签名、子节点及特性的访问。Element 节点具有以下特征:

- ☐ nodeType 的值为 1;
- ☐ nodeName 的值为元素的标签名;
- ☐ nodeValue 的值为 null;
- ☐ parentNode 可能是 Document 或 Element;
- ☐ 其子节点可能是 Element、Text、Comment、ProcessingInstruction、CDATASection 或 EntityReference。

要访问元素的标签名, 可以使用 nodeName 属性, 也可以使用 tagName 属性; 这两个属性会返回相同的值 (使用后者主要是为了清晰起见)。以下面的元素为例:

```
<div id="myDiv"></div>
```

可以像下面这样取得这个元素及其标签名:

```
var div = document.getElementById("myDiv");
alert(div.tagName);      //"DIV"
alert(div.tagName == div.nodeName); //true
```

这里的元素标签名是 `div`，它拥有一个值为 `"myDiv"` 的 ID。可是，`div.tagName` 实际上输出的是 `"DIV"` 而非 `"div"`。在 HTML 中，标签名始终都以全部大写表示；而在 XML（有时候也包括 XHTML）中，标签名则始终会与源代码中的保持一致。假如你不确定自己的脚本将会在 HTML 还是 XML 文档中执行，最好是在比较之前将标签名转换为相同的大小写形式，如下面的例子所示：

```
if (element.tagName == "div"){ //不能这样比较，很容易出错！
    //在此执行某些操作
}

if (element.tagName.toLowerCase() == "div"){ //这样最好（适用于任何文档）
    //在此执行某些操作
}
```

这个例子展示了围绕 `tagName` 属性的两次比较操作。第一次比较非常容易出错，因为其代码在 HTML 文档中不管用。第二次比较将标签名转换成了全部小写，是我们推荐的做法，因为这种做法适用于 HTML 文档，也适用于 XML 文档。



可以在任何浏览器中通过脚本访问 `Element` 类型的构造函数及原型，包括 IE8 及之前版本。在 Safari 2 之前版本和 Opera 8 之前的版本中，不能访问 `Element` 类型的构造函数。

## 1. HTML 元素

所有 HTML 元素都由 `HTMLElement` 类型表示，不是直接通过这个类型，也是通过它的子类型来表示。`HTMLElement` 类型直接继承自 `Element` 并添加了一些属性。添加的这些属性分别对应于每个 HTML 元素中都存在的下列标准特性。

- `id`，元素在文档中的唯一标识符。
- `title`，有关元素的附加说明信息，一般通过工具提示条显示出来。
- `lang`，元素内容的语言代码，很少使用。
- `dir`，语言的方向，值为 `"ltr"`（left-to-right，从左至右）或 `"rtl"`（right-to-left，从右至左），也很少使用。
- `className`，与元素的 `class` 特性对应，即为元素指定的 CSS 类。没有将这个属性命名为 `class`，是因为 `class` 是 ECMAScript 的保留字（有关保留字的信息，请参见第 1 章）。

上述这些属性都可以用来取得或修改相应的特性值。以下面的 HTML 元素为例：

```
<div id="myDiv" class="bd" title="Body text" lang="en" dir="ltr"></div>
```

*HTMLElementsExample01.htm*

元素中指定的所有信息，都可以通过下列 JavaScript 代码取得：

```
var div = document.getElementById("myDiv");
alert(div.id);           // "myDiv"
alert(div.className);    // "bd"
alert(div.title);        // "Body text"
alert(div.lang);         // "en"
alert(div.dir);          // "ltr"
```

当然，像下面这样通过为每个属性赋予新的值，也可以修改对应的每个特性：



```
div.id = "someOtherId";
div.className = "ft";
div.title = "Some other text";
div.lang = "fr";
div.dir = "rtl";
```

*HTMLElementsExample01.htm*

并不是对所有属性的修改都会在页面中直观地表现出来。对 id 或 lang 的修改对用户而言是透明不可见的（假设没有基于它们的值设置的 CSS 样式），而对 title 的修改则只会在鼠标移动到这个元素之上时才会显示出来。对 dir 的修改会在属性被重写的那一刻，立即影响页面中文本的左、右对齐方式。修改 className 时，如果新类关联了与此前不同的 CSS 样式，那么就会立即应用新的样式。

前面提到过，所有 HTML 元素都是由 HTMLElement 或者其更具体的子类型来表示的。下表列出了所有 HTML 元素以及与之关联的类型（以斜体印刷的元素表示已经不再推荐使用了）。注意，表中的这些类型在 Opera、Safari、Chrome 和 Firefox 中都可以通过 JavaScript 访问，但在 IE8 之前的版本中不能通过 JavaScript 访问。

元 素	类 型	元 素	类 型
A	HTMLAnchorElement	EM	HTMLElement
ABBR	HTMLElement	FIELDSET	HTMLFieldSetElement
ACRONYM	HTMLElement	FONT	HTMLFontElement
ADDRESS	HTMLElement	FORM	HTMLFormElement
APPLET	HTMLAppletElement	FRAME	HTMLFrameElement
AREA	HTMLAreaElement	FRAMESET	HTMLFrameSetElement
B	HTMLElement	H1	HTMLHeadingElement
BASE	HTMLBaseElement	H2	HTMLHeadingElement
BASEFONT	HTMLBaseFontElement	H3	HTMLHeadingElement
BDO	HTMLElement	H4	HTMLHeadingElement
BIG	HTMLElement	H5	HTMLHeadingElement
BLOCKQUOTE	HTMLQuoteElement	H6	HTMLHeadingElement
BODY	HTMLBodyElement	HEAD	HTMLHeadElement
BR	HTMLBRElement	HR	HTMLHRElement
BUTTON	HTMLButtonElement	HTML	HTMLHtmlElement
CAPTION	HTMLTableCaptionElement	I	HTMLElement
CENTER	HTMLElement	IFRAME	HTMLIFrameElement
CITE	HTMLElement	IMG	HTMLImageElement
CODE	HTMLElement	INPUT	HTMLInputElement
COL	HTMLTableColElement	INS	HTMLModElement
COLGROUP	HTMLTableColElement	ISINDEX	HTMLIsIndexElement
DD	HTMLElement	KBD	HTMLElement
DEL	HTMLModElement	LABEL	HTMLLabelElement
DFN	HTMLElement	LEGEND	HTMLLegendElement
DIR	HTMLDirectoryElement	LI	HTMLLIElement
DIV	HTMLDivElement	LINK	HTMLLinkElement
DL	HTMLDListElement	MAP	HTMLMapElement
DT	HTMLElement	MENU	HTMLMenuElement

(续)

元 素	类 型	元 素	类 型
META	HTMLMetaElement	STRONG	HTMLHeadingElement
NOFRAMES	HTMLIFrameElement	STYLE	HTMLStyleElement
NOSCRIPT	HTMLScriptElement	SUB	HTMLTextElement
OBJECT	HTMLObjectElement	SUP	HTMLTextElement
OL	HTMLListElement	TABLE	HTMLTableElement
OPTGROUP	HTMLOptGroupElement	TBODY	HTMLTableSectionElement
OPTION	HTMLOptionElement	TD	HTMLTableCellElement
P	HTMLParagraphElement	TEXTAREA	HTMLTextAreaElement
PARAM	HTMLParamElement	TFOOT	HTMLTableSectionElement
PRE	HTMLPreElement	TH	HTMLTableCellElement
Q	HTMLQuoteElement	THEAD	HTMLTableSectionElement
S	HTMLTextElement	TITLE	HTMLTitleElement
SAMP	HTMLTextElement	TR	HTMLTableRowElement
SCRIPT	HTMLScriptElement	TT	HTMLTextElement
SELECT	HTMLSelectElement	U	HTMLTextElement
SMALL	HTMLTextElement	UL	HTMLListElement
SPAN	HTMLTextElement	VAR	HTMLTextElement
STRIKE	HTMLTextElement		

表中的每一种类型都有与之相关的特性和方法。本书将会讨论其中很多类型。

## 2. 取得特性

每个元素都有一或多个特性，这些特性的用途是给出相应元素或其内容的附加信息。操作特性的 DOM 方法主要有三个，分别是 `getAttribute()`、`setAttribute()` 和 `removeAttribute()`。这三个方法可以针对任何特性使用，包括那些以 `HTMLElement` 类型属性的形式定义的特性。来看下面的例子：

```
var div = document.getElementById("myDiv");
alert(div.getAttribute("id"));           // "myDiv"
alert(div.getAttribute("class"));        // "bd"
alert(div.getAttribute("title"));        // "Body text"
alert(div.getAttribute("lang"));         // "en"
alert(div.getAttribute("dir"));          // "ltr"
```

注意，传递给 `getAttribute()` 的特性名与实际的特性名相同。因此要想得到 `class` 特性值，应该传入 `"class"` 而不是 `"className"`，后者只有在通过对象属性访问特性时才用。如果给定名称的特性不存在，`getAttribute()` 返回 `null`。

通过 `getAttribute()` 方法也可以取得自定义特性（即标准 HTML 语言中没有的特性）的值，以下的元素为例：

```
<div id="myDiv" my_special_attribute="hello!"></div>
```

这个元素包含一个名为 `my_special_attribute` 的自定义特性，它的值是 `"hello!"`。可以像取得其他特性一样取得这个值，如下所示：

```
var value = div.getAttribute("my_special_attribute");
```

不过，特性的名称是不区分大小写的，即 `"ID"` 和 `"id"` 代表的都是同一个特性。另外也要注意，根据 HTML5 规范，自定义特性应该加上 `data-` 前缀以便验证。



任何元素的所有特性，也都可以通过 DOM 元素本身的属性来访问。当然，HTML 元素也会有 5 个属性与相应的特性一一对应。不过，只有公认的（非自定义的）特性才会以属性的形式添加到 DOM 对象中。以下面的元素为例：

```
<div id="myDiv" align="left" my_special_attribute="hello!"></div>
```

因为 id 和 align 在 HTML 中是 <div> 的公认特性，因此该元素的 DOM 对象中也将存在对应的属性。不过，自定义特性 my\_special\_attribute 在 Safari、Opera、Chrome 及 Firefox 中是不存在的；但 IE 却会为自定义特性也创建属性，如下面的例子所示：

```
alert(div.id);           // "myDiv"
alert(div.my_special_attribute); // undefined (IE 除外)
alert(div.align);        // "left"
```

*ElementAttributesExample02.htm*

有两类特殊的特性，它们虽然有对应的属性名，但属性的值与通过 getAttribute() 返回的值并不相同。第一类特性就是 style，用于通过 CSS 为元素指定样式。在通过 getAttribute() 访问时，返回的 style 特性值中包含的是 CSS 文本，而通过属性来访问它则会返回一个对象。由于 style 属性是用于以编程方式访问元素样式的（本章后面讨论），因此并没有直接映射到 style 特性。

第二类与众不同的特性是 onclick 这样的事件处理程序。当在元素上使用时，onclick 特性中包含的是 JavaScript 代码，如果通过 getAttribute() 访问，则会返回相应代码的字符串。而在访问 onclick 属性时，则会返回一个 JavaScript 函数（如果未在元素中指定相应特性，则返回 null）。这是因为 onclick 及其他事件处理程序属性本身就应该被赋予函数值。

由于存在这些差别，在通过 JavaScript 以编程方式操作 DOM 时，开发人员经常不使用 getAttribute()，而是只使用对象的属性。只有在取得自定义特性值的情况下，才会使用 getAttribute() 方法。

在 IE7 及以前版本中，通过 getAttribute() 方法访问 style 特性或 onclick 这样的事件处理特性时，返回的值与属性的值相同。换句话说，getAttribute("style") 返回一个对象，而 getAttribute("onclick") 返回一个函数。虽然 IE8 已经修复了这个 bug，但不同 IE 版本间的不一致性，也是导致开发人员不使用 getAttribute() 访问 HTML 特性的一个原因。

### 3. 设置特性

与 getAttribute() 对应的方法是 setAttribute()，这个方法接受两个参数：要设置的特性名和值。如果特性已经存在，setAttribute() 会以指定的值替换现有的值；如果特性不存在，setAttribute() 则创建该属性并设置相应的值。来看下面的例子：

```
div.setAttribute("id", "someOtherId");
div.setAttribute("class", "ft");
div.setAttribute("title", "Some other text");
div.setAttribute("lang", "fr");
div.setAttribute("dir", "rtl");
```

*ElementAttributesExample01.htm*

通过 `setAttribute()` 方法既可以操作 HTML 特性也可以操作自定义特性。通过这个方法设置的特性名会被统一转换为小写形式，即“ID”最终会变成“id”。

因为所有特性都是属性，所以直接给属性赋值可以设置特性的值，如下所示。

```
div.id = "someOtherId";  
div.align = "left";
```

不过，像下面这样为 DOM 元素添加一个自定义的属性，该属性不会自动成为元素的特性。

```
div.mycolor = "red";  
alert(div.getAttribute("mycolor")); //null (IE 除外)
```

这个例子添加了一个名为 `mycolor` 的属性并将它的值设置为“red”。在大多数浏览器中，这个属性都不会自动变成元素的特性，因此想通过 `getAttribute()` 取得同名特性的值，结果会返回 `null`。可是，自定义属性在 IE 中会被当作元素的特性，反之亦然。



在 IE7 及以前版本中，`setAttribute()` 存在一些异常行为。通过这个方法设置 `class` 和 `style` 特性，没有任何效果，而使用这个方法设置事件处理程序特性时也一样。尽管到了 IE8 才解决这些问题，但我们还是推荐通过属性来设置特性。

要介绍的最后一个方法是 `removeAttribute()`，这个方法用于彻底删除元素的特性。调用这个方法不仅会清除特性的值，而且也会从元素中完全删除特性，如下所示：

```
div.removeAttribute("class");
```

这个方法并不常用，但在序列化 DOM 元素时，可以通过它来确切地指定要包含哪些特性。



IE6 及以前版本不支持 `removeAttribute()`。

#### 4. attributes 属性

`Element` 类型是使用 `attributes` 属性的唯一一个 DOM 节点类型。`attributes` 属性中包含一个 `NamedNodeMap`，与 `NodeList` 类似，也是一个“动态”的集合。元素的每一个特性都由一个 `Attr` 节点表示，每个节点都保存在 `NamedNodeMap` 对象中。`NamedNodeMap` 对象拥有下列方法。

- ❑ `getNamedItem(name)`：返回 `nodeName` 属性等于 `name` 的节点；
- ❑ `removeNamedItem(name)`：从列表中移除 `nodeName` 属性等于 `name` 的节点；
- ❑ `setNamedItem(node)`：向列表中添加节点，以节点的 `nodeName` 属性为索引；
- ❑ `item(pos)`：返回位于数字 `pos` 位置处的节点。

`attributes` 属性中包含一系列节点，每个节点的 `nodeName` 就是特性的名称，而节点的 `nodeValue` 就是特性的值。要取得元素的 `id` 特性，可以使用以下代码。

```
var id = element.attributes.getNamedItem("id").nodeValue;
```

以下是使用方括号语法通过特性名称访问节点的简写方式。

```
var id = element.attributes["id"].nodeValue;
```

也可以使用这种语法来设置特性的值，即先取得特性节点，然后再将其 `nodeValue` 设置为新值，如下所示。

```
element.attributes["id"].nodeValue = "someOtherId";
```

调用 `removeNamedItem()` 方法与在元素上调用 `removeAttribute()` 方法的效果相同——直接删除具有给定名称的特性。下面的例子展示了两个方法间唯一的区别，即 `removeNamedItem()` 返回表示被删除特性的 `Attr` 节点。


```
var oldAttr = element.attributes.removeNamedItem("id");
```

最后，`setNamedItem()` 是一个很常用的方法，通过这个方法可以为元素添加一个新特性，为此需要为它传入一个特性节点，如下所示。

```
element.attributes.setNamedItem(newAttr);
```

一般来说，由于前面介绍的 `attributes` 的方法不够方便，因此开发人员更多的会使用 `getAttribute()`、`removeAttribute()` 和 `setAttribute()` 方法。

不过，如果想要遍历元素的特性，`attributes` 属性倒是可以派上用场。在需要将 DOM 结构序列化为 XML 或 HTML 字符串时，多数都会涉及遍历元素特性。以下代码展示了如何迭代元素的每一个特性，然后将它们构造成 `name="value" name="value"` 这样的字符串格式。



```
function outputAttributes(element){  
    var pairs = new Array(),  
        attrName,  
        attrValue,  
        i,  
        len;  
  
    for (i=0, len=element.attributes.length; i < len; i++){  
        attrName = element.attributes[i].nodeName;  
        attrValue = element.attributes[i].nodeValue;  
        pairs.push(attrName + "=\"" + attrValue + "\"");  
    }  
    return pairs.join(" ");  
}
```

*ElementAttributesExample03.htm*

这个函数使用了一个数组来保存名值对，最后再以空格为分隔符将它们拼接起来（这是序列化长字符串时的一种常用技巧）。通过 `attributes.length` 属性，`for` 循环会遍历每个特性，将特性的名称和值输出为字符串。关于以上代码的运行结果，以下是两点必要的说明。

- 针对 `attributes` 对象中的特性，不同浏览器返回的顺序不同。这些特性在 XML 或 HTML 代码中出现的先后顺序，不一定与它们出现在 `attributes` 对象中的顺序一致。
- IE7 及更早的版本会返回 HTML 元素中所有可能的特性，包括没有指定的特性。换句话说，返回 100 多个特性的情况会很常见。

针对 IE7 及更早版本中存在的问题，可以对上面的函数加以改进，让它只返回指定的特性。每个特性节点都有一个名为 `specified` 的属性，这个属性的值如果为 `true`，则意味着要么是在 HTML 中指定了相应特性，要么是通过 `setAttribute()` 方法设置了该特性。在 IE 中，所有未设置过的特性的该属性值都为 `false`，而在其他浏览器中根本不会为这类特性生成对应的特性节点（因此，在这些浏览器中，任何特性节点的 `specified` 值始终为 `true`）。改进后的代码如下所示。



```
function outputAttributes(element){
    var pairs = new Array(),
        attrName,
        attrValue,
        i,
        len;

    for (i=0, len=element.attributes.length; i < len; i++){
        attrName = element.attributes[i].nodeName;
        attrValue = element.attributes[i].nodeValue;
        if (element.attributes[i].specified) {
            pairs.push(attrName + "=\"" + attrValue + "\"");
        }
    }
    return pairs.join(" ");
}
```

*ElementAttributesExample04.htm*

这个经过改进的函数可以确保即使在 IE7 及更早的版本中，也会只返回指定的特性。

## 5. 创建元素

使用 `document.createElement()` 方法可以创建新元素。这个方法只接受一个参数，即要创建元素的标签名。这个标签名在 HTML 文档中不区分大小写，而在 XML（包括 XHTML）文档中，则是区分大小写的。例如，使用下面的代码可以创建一个 `<div>` 元素。

```
var div = document.createElement("div");
```

在使用 `createElement()` 方法创建新元素的同时，也为新元素设置了 `ownerDocument` 属性。此时，还可以操作元素的特性，为它添加更多子节点，以及执行其他操作。来看下面的例子。

```
div.id = "myNewDiv";
div.className = "box";
```

在新元素上设置这些特性只是给它们赋予了相应的信息。由于新元素尚未被添加到文档树中，因此设置这些特性不会影响浏览器的显示。要把新元素添加到文档树，可以使用 `appendChild()`、`insertBefore()` 或 `replaceChild()` 方法。下面的代码会把新创建的元素添加到文档的 `<body>` 元素中。

```
document.body.appendChild(div);
```

*CreateElementExample01.htm*

一旦将元素添加到文档树中，浏览器就会立即呈现该元素。此后，对这个元素所作的任何修改都会实时反映在浏览器中。

在 IE 中可以以另一种方式使用 `createElement()`，即为这个方法传入完整的元素标签，也可以包含属性，如下面的例子所示。

```
var div = document.createElement("<div id=\"myNewDiv\" class=\"box\"></div >");
```

这种方式有助于避开在 IE7 及更早版本中动态创建元素的某些问题。下面是已知的一些这类问题。

- 不能设置动态创建的 `<iframe>` 元素的 `name` 特性。
- 不能通过表单的 `reset()` 方法重设动态创建的 `<input>` 元素（第 13 章将讨论 `reset()` 方法）。

- 动态创建的 type 特性值为 "reset" 的 <button> 元素重设不了表单。
  - 动态创建的一批 name 相同的单选按钮彼此毫无关系。name 值相同的一组单选按钮本来应该用于表示同一选项的不同值，但动态创建的一批这种单选按钮之间却没有这种关系。
- 上述所有问题都可以通过在 createElement() 中指定完整的 HTML 标签来解决，如下面的例子所示。

```
if (client.browser.ie && client.browser.ie <=7){  
    //创建一个带 name 特性的 iframe 元素  
    var iframe = document.createElement("<iframe name=\"myframe\"></iframe>");  
  
    //创建 input 元素  
    var input = document.createElement("<input type=\"checkbox\">");  
  
    //创建 button 元素  
    var button = document.createElement("<button type=\"reset\"></button>");  
  
    //创建单选按钮  
    var radio1 = document.createElement("<input type=\"radio\" name=\"choice\" \" +  
        \"value=\"1\">");  
    var radio2 = document.createElement("<input type=\"radio\" name=\"choice\" \" +  
        \"value=\"2\">");  
}
```

与使用 createElement() 的惯常方式一样，这样的用法也会返回一个 DOM 元素的引用。可以将这个引用添加到文档中，也可以对其加以增强。但是，由于这样的用法要求使用浏览器检测，因此我们建议只在需要避开 IE 及更早版本中上述某个问题的情况下使用。其他浏览器都不支持这种用法。

## 6. 元素的子节点

元素可以有任意数目的子节点和后代节点，因为元素可以是其他元素的子节点。元素的 childNodes 属性中包含了它的所有子节点，这些子节点有可能是元素、文本节点、注释或处理指令。不同浏览器在看待这些节点方面存在显著的不同，下面的代码为例。

```
<ul id="myList">  
    <li>Item 1</li>  
    <li>Item 2</li>  
    <li>Item 3</li>  
</ul>
```

如果是 IE 来解析这些代码，那么 <ul> 元素会有 3 个子节点，分别是 3 个 <li> 元素。但如果是在其他浏览器中，<ul> 元素都会有 7 个元素，包括 3 个 <li> 元素和 4 个文本节点（表示 <li> 元素之间的空白符）。如果像下面这样将元素间的空白符删除，那么所有浏览器都会返回相同数目的子节点。

```
<ul id="myList"><li>Item 1</li><li>Item 2</li><li>Item 3</li></ul>
```

对于这段代码，<ul> 元素在任何浏览器中都会包含 3 个子节点。如果需要通过 childNodes 属性遍历子节点，那么一定不要忘记浏览器间的这一差别。这意味着在执行某项操作以前，通常都要先检查一下 nodeType 属性，如下面的例子所示。

```
for (var i=0, len=element.childNodes.length; i < len; i++){  
    if (element.childNodes[i].nodeType == 1){  
        //执行某些操作  
    }  
}
```



这个例子会循环遍历特定元素的每一个子节点，然后只在子节点的 `nodeType` 等于 1（表示是元素节点）的情况下，才会执行某些操作。

如果想通过某个特定的标签名取得子节点或后代节点该怎么办呢？实际上，元素也支持 `getElementsByTagName()` 方法。在通过元素调用这个方法时，除了搜索起点是当前元素之外，其他方面都跟通过 `document` 调用这个方法相同，因此结果只会返回当前元素的后代。例如，要想取得前面 `<ul>` 元素中包含的所有 `<li>` 元素，可以使用下列代码。

```
var ul = document.getElementById("myList");  
var items = ul.getElementsByTagName("li");
```

要注意的是，这里 `<ul>` 的后代中只包含直接子元素。不过，如果它包含更多层次的后代元素，那么各个层次中包含的 `<li>` 元素也都会返回。

### 10.1.4 Text 类型

文本节点由 `Text` 类型表示，包含的是可以照字面解释的纯文本内容。纯文本中可以包含转义后的 HTML 字符，但不能包含 HTML 代码。`Text` 节点具有以下特征：

- `nodeType` 的值为 3；
- `nodeName` 的值为 `"#text"`；
- `nodeValue` 的值为节点所包含的文本；
- `parentNode` 是一个 `Element`；
- 不支持（没有）子节点。

可以通过 `nodeValue` 属性或 `data` 属性访问 `Text` 节点中包含的文本，这两个属性中包含的值相同。对 `nodeValue` 的修改也会通过 `data` 反映出来，反之亦然。使用下列方法可以操作节点中的文本。

- `appendData(text)`：将 `text` 添加到节点的末尾。
- `deleteData(offset, count)`：从 `offset` 指定的位置开始删除 `count` 个字符。
- `insertData(offset, text)`：在 `offset` 指定的位置插入 `text`。
- `replaceData(offset, count, text)`：用 `text` 替换从 `offset` 指定的位置开始到 `offset+count` 为止处的文本。
- `splitText(offset)`：从 `offset` 指定的位置将当前文本节点分成两个文本节点。
- `substringData(offset, count)`：提取从 `offset` 指定的位置开始到 `offset+count` 为止处的字符串。

除了这些方法之外，文本节点还有一个 `length` 属性，保存着节点中字符的数目。而且，`nodeValue.length` 和 `data.length` 中也保存着同样的值。

在默认情况下，每个可以包含内容的元素最多只能有一个文本节点，而且必须确实有内容存在。来看几个例子。

```
<!-- 没有内容，也就没有文本节点 -->  
<div></div>  
  
<!-- 有空格，因而有一个文本节点 -->  
<div> </div>  
  
<!-- 有内容，因而有一个文本节点 -->  
<div>Hello World!</div>
```

上面代码给出的第一个<div>元素没有内容，因此也就不存在文本节点。开始与结束标签之间只要存在内容，就会创建一个文本节点。因此，第二个<div>元素中虽然只包含一个空格，但仍然有一个文本子节点；文本节点的 nodeValue 值是一个空格。第三个<div>也有一个文本节点，其 nodeValue 的值为"Hello World!"。可以使用以下代码来访问这些文本子节点。

```
var textNode = div.firstChild; //或者 div.childNodes[0]
```

在取得了文本节点的引用后，就可以像下面这样来修改它了。

```
div.firstChild.nodeValue = "Some other message";
```

*TextNodeExample01.htm*

如果这个文本节点当前存在于文档树中，那么修改文本节点的结果就会立即得到反映。另外，在修改文本节点时还要注意，此时的字符串会经过 HTML（或 XML，取决于文档类型）编码。换句话说，小于号、大于号或引号都会像下面的例子一样被转义。

```
//输出结果是"Some &lt;strong&gt;other&lt;/strong&gt; message"  
div.firstChild.nodeValue = "Some <strong>other</strong> message";
```

*TextNodeExample02.htm*

应该说，这是在向 DOM 文档中插入文本之前，先对其进行 HTML 编码的一种有效方式。



在 IE8、Firefox、Safari、Chrome 和 Opera 中，可以通过脚本访问 Text 类型的构造函数和原型。

### 1. 创建文本节点

可以使用 document.createTextNode() 创建新文本节点，这个方法接受一个参数——要插入节点中的文本。与设置已有文本节点的值一样，作为参数的文本也将按照 HTML 或 XML 的格式进行编码。

```
var textNode = document.createTextNode("<strong>Hello</strong> world!");
```

在创建新文本节点的同时，也会为其设置 ownerDocument 属性。不过，除非把新节点添加到文档树中已经存在的节点中，否则我们不会在浏览器窗口中看到新节点。下面的代码会创建一个<div>元素并向其中添加一条消息。

```
var element = document.createElement("div");  
element.className = "message";  
  
var textNode = document.createTextNode("Hello world!");  
element.appendChild(textNode);  
  
document.body.appendChild(element);
```

*TextNodeExample03.htm*

这个例子创建了一个新<div>元素并为它指定了值为"message"的 class 特性。然后，又创建了一个文本节点，并将其添加到前面创建的元素中。最后一步，就是将这个元素添加到了文档的<body>元素中，这样就可以在浏览器中看到新建的元素和文本节点了。

一般情况下，每个元素只有一个文本子节点。不过，在某些情况下也可能包含多个文本子节点，如下面的例子所示。



```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

var anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);
```

*TextNodeExample04.htm*

如果两个文本节点是相邻的同胞节点，那么这两个节点中的文本就会连起来显示，中间不会有空格。

## 2. 规范化文本节点

DOM 文档中存在相邻的同胞文本节点很容易导致混乱，因为分不清哪个文本节点表示哪个字符串。另外，DOM 文档中出现相邻文本节点的情况也不在少数，于是就催生了一个能够将相邻文本节点合并的方法。这个方法是由 Node 类型定义的（因而在所有节点类型中都存在），名叫 `normalize()`。如果在一个包含两个或多个文本节点的父元素上调用 `normalize()` 方法，则会将所有文本节点合并成一个节点，结果节点的 `nodeValue` 等于将合并前每个文本节点的 `nodeValue` 值拼接起来的值。来看一个例子。



```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

var anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);

document.body.appendChild(element);

alert(element.childNodes.length);    //2

element.normalize();
alert(element.childNodes.length);    //1
alert(element.firstChild.nodeValue); // "Hello world!Yippee!"
```

*TextNodeExample05.htm*


浏览器在解析文档时永远不会创建相邻的文本节点。这种情况只会作为执行 DOM 操作的结果出现。



在某些情况下，执行 `normalize()` 方法会导致 IE6 崩溃。不过，在 IE6 后来的补丁中，可能已经修复了这个问题（未经证实）。IE7 及更高版本中不存在这个问题。

### 3. 分割文本节点

Text 类型提供了一个作用与 `normalize()` 相反的方法: `splitText()`。这个方法会将一个文本节点分成两个文本节点, 即按照指定的位置分割 `nodeValue` 值。原来的文本节点将包含从开始到指定位置之前的内容, 新文本节点将包含剩下的文本。这个方法会返回一个新文本节点, 该节点与原节点的 `parentNode` 相同。来看下面的例子。



```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);

var newNode = element.firstChild.splitText(5);
alert(element.firstChild.nodeValue);    //"Hello"
alert(newNode.nodeValue);               //" world!"
alert(element.childNodes.length);       //2
```

*TextNodeExample06.htm*

在这个例子中, 包含 "Hello world!" 的文本节点被分割为两个文本节点, 从位置 5 开始。位置 5 是 "Hello" 和 "world!" 之间的空格, 因此原来的文本节点将包含字符串 "Hello", 而新文本节点将包含文本 "world!" (包含空格)。

分割文本节点是从文本节点中提取数据的一种常用 DOM 解析技术。

### 10.1.5 Comment 类型

注释在 DOM 中是通过 Comment 类型来表示的。Comment 节点具有下列特征:

- ☐ `nodeType` 的值为 8;
- ☐ `nodeName` 的值为 "#comment";
- ☐ `nodeValue` 的值是注释的内容;
- ☐ `parentNode` 可能是 Document 或 Element;
- ☐ 不支持 (没有) 子节点。

Comment 类型与 Text 类型继承自相同的基类, 因此它拥有除 `splitText()` 之外的所有字符串操作方法。与 Text 类型相似, 也可以通过 `nodeValue` 或 `data` 属性来取得注释的内容。

注释节点可以通过其父节点来访问, 以下面的代码为例。



```
<div id="myDiv"><!--A comment --></div>
```

在此, 注释节点是 `<div>` 元素的一个子节点, 因此可以通过下面的代码来访问它。

```
var div = document.getElementById("myDiv");
var comment = div.firstChild;
alert(comment.data);    //"A comment"
```

*CommentNodeExample01.htm*

另外,使用 `document.createComment()` 并为其传递注释文本也可以创建注释节点,如下面的例子所示。

```
var comment = document.createComment("A comment ");
```

显然,开发人员很少会创建和访问注释节点,因为注释节点对算法鲜有影响。此外,浏览器也不会识别位于 `</html>` 标签后面的注释。如果要访问注释节点,一定要保证它们是 `<html>` 元素的后代(即位于 `<html>` 和 `</html>` 之间)。



在 Firefox、Safari、Chrome 和 Opera 中,可以访问 `Comment` 类型的构造函数和原型。在 IE8 中,注释节点被视作标签名为 `!` 的元素。也就是说,使用 `getElementsByTagName()` 可以取得注释节点。尽管 IE9 没有把注释当成元素,但它仍然通过一个名为 `HTMLCommentElement` 的构造函数来表示注释。

### 10.1.6 CDATASection 类型

`CDATASection` 类型只针对基于 XML 的文档,表示的是 `CDATA` 区域。与 `Comment` 类似,`CDATASection` 类型继承自 `Text` 类型,因此拥有除 `splitText()` 之外的所有字符串操作方法。`CDATASection` 节点具有下列特征:

- ☐ `nodeType` 的值为 4;
- ☐ `nodeName` 的值为 `"#cdata-section"`;
- ☐ `nodeValue` 的值是 `CDATA` 区域中的内容;
- ☐ `parentNode` 可能是 `Document` 或 `Element`;
- ☐ 不支持(没有)子节点。

`CDATA` 区域只会出现在 XML 文档中,因此多数浏览器都会把 `CDATA` 区域错误地解析为 `Comment` 或 `Element`。以下面的代码为例:

```
<div id="myDiv"><![CDATA[This is some content.]]></div>
```

这个例子中的 `<div>` 元素应该包含一个 `CDATASection` 节点。可是,四大主流浏览器无一能够这样解析它。即使对于有效的 XHTML 页面,浏览器也没有正确地支持嵌入的 `CDATA` 区域。

在真正的 XML 文档中,可以使用 `document.createCDATASection()` 来创建 `CDATA` 区域,只需为其传入节点的内容即可。



在 Firefox、Safari、Chrome 和 Opera 中,可以访问 `CDATASection` 类型的构造函数和原型。IE9 及之前版本不支持这个类型。

### 10.1.7 DocumentType 类型

`DocumentType` 类型在 Web 浏览器中并不常用,仅有 Firefox、Safari 和 Opera 支持它<sup>①</sup>。Document-

<sup>①</sup> Chrome 4.0 也支持 `DocumentType` 类型。



Type 包含着与文档的 doctype 有关的所有信息，它具有下列特征：

- ☐ nodeType 的值为 10；
- ☐ nodeName 的值为 doctype 的名称；
- ☐ nodeValue 的值为 null；
- ☐ parentNode 是 Document；
- ☐ 不支持（没有）子节点。

在 DOM1 级中，DocumentType 对象不能动态创建，而只能通过解析文档代码的方式来创建。支持它的浏览器会把 DocumentType 对象保存在 document.doctype 中。DOM1 级描述了 DocumentType 对象的 3 个属性：name、entities 和 notations。其中，name 表示文档类型的名称；entities 是由文档类型描述的实体的 NamedNodeMap 对象；notations 是由文档类型描述的符号的 NamedNodeMap 对象。通常，浏览器中的文档使用的都是 HTML 或 XHTML 文档类型，因而 entities 和 notations 都是空列表（列表中的项来自行内文档类型声明）。但不管怎样，只有 name 属性是有用的。这个属性中保存的是文档类型的名称，也就是出现在<!DOCTYPE 之后的文本。以下面严格型 HTML 4.01 的文档类型声明为例：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">
```

DocumentType 的 name 属性中保存的就是 "HTML"：

```
alert(document.doctype.name);           //"HTML"
```

IE 及更早版本不支持 DocumentType，因此 document.doctype 的值始终都等于 null。可是，这些浏览器会把文档类型声明错误地解释为注释，并且为它创建一个注释节点。IE9 会给 document.doctype 赋正确的对象，但仍然不支持访问 DocumentType 类型。

## 10.1.8 DocumentFragment 类型

在所有节点类型中，只有 DocumentFragment 在文档中没有对应的标记。DOM 规定文档片段（document fragment）是一种“轻量级”的文档，可以包含和控制节点，但不会像完整的文档那样占用额外的资源。DocumentFragment 节点具有下列特征：

- ☐ nodeType 的值为 11；
- ☐ nodeName 的值为 "#document-fragment"；
- ☐ nodeValue 的值为 null；
- ☐ parentNode 的值为 null；
- ☐ 子节点可以是 Element、ProcessingInstruction、Comment、Text、CDATASection 或 EntityReference。

虽然不能把文档片段直接添加到文档中，但可以将它作为一个“仓库”来使用，即可以在里面保存将来可能会添加到文档中的节点。要创建文档片段，可以使用 document.createDocumentFragment() 方法，如下所示：

```
var fragment = document.createDocumentFragment();
```

文档片段继承了 Node 的所有方法，通常用于执行那些针对文档的 DOM 操作。如果将文档中的节点添加到文档片段中，就会从文档树中移除该节点，也不会从浏览器中再看到该节点。添加到文档片段中的新节点同样也不属于文档树。可以通过 `appendChild()` 或 `insertBefore()` 将文档片段中内容添加到文档中。在将文档片段作为参数传递给这两个方法时，实际上只会将文档片段的所有子节点添加到相应位置上；文档片段本身永远不会成为文档树的一部分。来看下面的 HTML 示例代码：

```
<ul id="myList"></ul>
```

假设我们想为这个 `<ul>` 元素添加 3 个列表项。如果逐个地添加列表项，将会导致浏览器反复渲染（呈现）新信息。为避免这个问题，可以像下面这样使用一个文档片段来保存创建的列表项，然后再一次性将它们添加到文档中。



```
var fragment = document.createDocumentFragment();
var ul = document.getElementById("myList");
var li = null;

for (var i=0; i < 3; i++){
    li = document.createElement("li");
    li.appendChild(document.createTextNode("Item " + (i+1)));
    fragment.appendChild(li);
}

ul.appendChild(fragment);
```

*DocumentFragmentExample01.htm*

在这个例子中，我们先创建一个文档片段并取得了对其 `<ul>` 元素的引用。然后，通过 `for` 循环创建 3 个列表项，并通过文本表示它们的顺序。为此，需要分别创建 `<li>` 元素、创建文本节点，再把文本节点添加到 `<li>` 元素。接着使用 `appendChild()` 将 `<li>` 元素添加到文档片段中。循环结束后，再调用 `appendChild()` 并传入文档片段，将所有列表项添加到 `<ul>` 元素中。此时，文档片段的所有子节点都被删除并转移到了 `<ul>` 元素中。

### 10.1.9 Attr 类型


元素的特性在 DOM 中以 `Attr` 类型来表示。在所有浏览器中（包括 IE8），都可以访问 `Attr` 类型的构造函数和原型。从技术角度讲，特性就是存在于元素的 `attributes` 属性中的节点。特性节点具有下列特征：

- ❑ `nodeType` 的值为 11；
- ❑ `nodeName` 的值是特性的名称；
- ❑ `nodeValue` 的值是特性的值；
- ❑ `parentNode` 的值为 `null`；
- ❑ 在 HTML 中不支持（没有）子节点；
- ❑ 在 XML 中子节点可以是 `Text` 或 `EntityReference`。

尽管它们也是节点，但特性却不被认为是 DOM 文档树的一部分。开发人员最常使用的是 `getAttribute()`、`setAttribute()` 和 `removeAttribute()` 方法，很少直接引用特性节点。

Attr 对象有 3 个属性: name、value 和 specified。其中, name 是特性名称 (与 nodeName 的值相同), value 是特性的值 (与 nodeValue 的值相同), 而 specified 是一个布尔值, 用以区别特性是在代码中指定的, 还是默认的。

使用 document.createAttribute() 并传入特性的名称可以创建新的特性节点。例如, 要为元素添加 align 特性, 可以使用下列代码:



```
var attr = document.createAttribute("align");
attr.value = "left";
element.setAttributeNode(attr);
alert(element.attributes["align"].value);           //"left"
alert(element.getAttributeNode("align").value);      //"left"
alert(element.getAttribute("align"));                 //"left"
```

*AttrExample01.htm*

这个例子创建了一个新的特性节点。由于在调用 createAttribute() 时已经为 name 属性赋了值, 所以后面就不必给它赋值了。之后, 又把 value 属性的值设置为 "left"。为了将新创建的特性添加到元素中, 必须使用元素的 setAttributeNode() 方法。添加特性之后, 可以通过下列任何方式访问该特性: attributes 属性、getAttributeNode() 方法以及 getAttribute() 方法。其中, attributes 和 getAttributeNode() 都会返回对应特性的 Attr 节点, 而 getAttribute() 则只返回特性的值。



我们并不建议直接访问特性节点。实际上, 使用 getAttribute()、setAttribute() 和 removeAttribute() 方法远比操作特性节点更为方便。

## 10.2 DOM 操作技术

很多时候, DOM 操作都比较简明, 因此用 JavaScript 生成那些通常原本是用 HTML 代码生成的内容并不麻烦。不过, 也有一些时候, 操作 DOM 并不像表面上看起来那么简单。由于浏览器中充斥着隐藏的陷阱和不兼容问题, 用 JavaScript 代码处理 DOM 的某些部分要比处理其他部分更复杂一些。不过, 也有一些时候, 操作 DOM 并不像表面上看起来那么简单。

### 10.2.1 动态脚本

使用<script>元素可以向页面中插入 JavaScript 代码, 一种方式是通过其 src 特性包含外部文件, 另一种方式就是用这个元素本身来包含代码。而这一节要讨论的动态脚本, 指的是在页面加载时不存在, 但将来的某一时刻通过修改 DOM 动态添加的脚本。跟操作 HTML 元素一样, 创建动态脚本也有两种方式: 插入外部文件和直接插入 JavaScript 代码。

动态加载的外部 JavaScript 文件能够立即运行, 比如下面的<script>元素:

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

这个<script>元素包含了第 9 章的客户端检测脚本。而创建这个节点的 DOM 代码如下所示:

```
var script = document.createElement("script");
script.type = "text/javascript";
```

```
script.src = "client.js";  
document.body.appendChild(script);
```

显然, 这里的 DOM 代码如实反映了相应的 HTML 代码。不过, 在执行最后一行代码把<script>元素添加到页面中之前, 是不会下载外部文件的。也可以把这个元素添加到<head>元素中, 效果相同。整个过程可以使用下面的函数来封装:

```
function loadScript(url){  
    var script = document.createElement("script");  
    script.type = "text/javascript";  
    script.src = url;  
    document.body.appendChild(script);  
}
```

然后, 就可以通过调用这个函数来加载外部的 JavaScript 文件了:

```
loadScript("client.js");
```

加载完成后, 就可以在页面中的其他地方使用这个脚本了。问题只有一个: 怎么知道脚本加载完成呢? 遗憾的是, 并没有什么标准方式来探知这一点。不过, 与此相关的一些事件倒是可以派上用场, 但要取决于所用的浏览器, 详细讨论请见第 13 章。

另一种指定 JavaScript 代码的方式是行内方式, 如下面的例子所示:

```
<script type="text/javascript">  
    function sayHi(){  
        alert("hi");  
    }  
</script>
```

从逻辑上讲, 下面的 DOM 代码是有效的:

```
var script = document.createElement("script");  
script.type = "text/javascript";  
script.appendChild(document.createTextNode("function sayHi(){alert('hi');}"));  
document.body.appendChild(script);
```

在 Firefox、Safari、Chrome 和 Opera 中, 这些 DOM 代码可以正常运行。但在 IE 中, 则会导致错误。IE 将<script>视为一个特殊的元素, 不允许 DOM 访问其子节点。不过, 可以使用<script>元素的 text 属性来指定 JavaScript 代码, 像下面的例子这样:



```
var script = document.createElement("script");  
script.type = "text/javascript";  
script.text = "function sayHi(){alert('hi');}";  
document.body.appendChild(script);
```

*DynamicScriptExample01.htm*

经过这样修改之后的代码可以在 IE、Firefox、Opera 和 Safari 3 及之后版本中运行。Safari 3.0 之前的版本虽然不能正确地支持 text 属性, 但却允许使用文本节点技术来指定代码。如果需要兼容早期版本的 Safari, 可以使用下列代码:

```
var script = document.createElement("script");  
script.type = "text/javascript";  
var code = "function sayHi(){alert('hi');}";
```

```
try {  
    script.appendChild(document.createTextNode("code"));  
} catch (ex){  
    script.text = "code";  
}  
document.body.appendChild(script);
```

这里，首先尝试标准的 DOM 文本节点方法，因为除了 IE（在 IE 中会导致抛出错误），所有浏览器都支持这种方式。如果这行代码抛出了错误，那么说明是 IE，于是就必须使用 text 属性了。整个过程可以用以下函数来表示：



```
function loadScriptString(code){  
    var script = document.createElement("script");  
    script.type = "text/javascript";  
    try {  
        script.appendChild(document.createTextNode(code));  
    } catch (ex){  
        script.text = code;  
    }  
    document.body.appendChild(script);  
}
```

下面是调用这个函数的示例：

```
loadScriptString("function sayHi(){alert('hi');}");
```

[DynamicScriptExample02.htm](#)

以这种方式加载的代码会在全局作用域中执行，而且当脚本执行后将立即可用。实际上，这样执行代码与在全局作用域中把相同的字符串传递给 eval() 是一样的。

## 10.2.2 动态样式

能够把 CSS 样式包含到 HTML 页面中的元素有两个。其中，<link>元素用于包含来自外部的文件，而<style>元素用于指定嵌入的样式。与动态脚本类似，所谓动态样式是指在页面刚加载时不存在的样式；动态样式是在页面加载完成后动态添加到页面中的。

我们下面这个典型的<link>元素为例：

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

使用 DOM 代码可以很容易地动态创建出这个元素：

```
var link = document.createElement("link");  
link.rel = "stylesheet";  
link.type = "text/css";  
link.href = "style.css";  
var head = document.getElementsByTagName("head")[0];  
head.appendChild(link);
```

以上代码在所有主流浏览器中都可以正常运行。需要注意的是，必须将<link>元素添加到<head>而不是<body>元素，才能保证在所有浏览器中的行为一致。整个过程可以用以下函数来表示：

```
function loadStyles(url){  
    var link = document.createElement("link");
```



```
link.rel = "stylesheet";
link.type = "text/css";
link.href = url;
var head = document.getElementsByTagName("head")[0];
head.appendChild(link);
}
```

调用 loadStyles() 函数的代码如下所示:

```
loadStyles("styles.css");
```

加载外部样式文件的过程是异步的, 也就是加载样式与执行 JavaScript 代码的过程没有固定的次序。一般来说, 知不知道样式已经加载完成并不重要; 不过, 也存在几种利用事件来检测这个过程是否完成的技术, 这些技术将在第 13 章讨论。

另一种定义样式的方式是使用<style>元素来包含嵌入式 CSS, 如下所示:

```
<style type="text/css">
body {
    background-color: red;
}
</style>
```

按照相同的逻辑, 下列 DOM 代码应该是有效的:

```
var style = document.createElement("style");
style.type = "text/css";
style.appendChild(document.createTextNode("body{background-color:red}"));
var head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

*DynamicStyleExample01.htm*

以上代码可以在 Firefox、Safari、Chrome 和 Opera 中运行, 在 IE 中则会报错。IE 将<style>视为一个特殊的、与<script>类似的节点, 不允许访问其子节点。事实上, IE 此时抛出的错误与向<script>元素添加子节点时抛出的错误相同。解决 IE 中这个问题的办法, 就是访问元素的 styleSheet 属性, 该属性又有一个 cssText 属性, 可以接受 CSS 代码 (第 13 章将进一步讨论这两个属性), 如下面的例子所示。

```
var style = document.createElement("style");
style.type = "text/css";
try{
    style.appendChild(document.createTextNode("body{background-color:red}"));
} catch (ex){
    style.styleSheet.cssText = "body{background-color:red}";
}
var head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

与动态添加嵌入式脚本类似, 重写后的代码使用了 try-catch 语句来捕获 IE 抛出的错误, 然后再使用针对 IE 的特殊方式来设置样式。因此, 通用的解决方案如下。

```
function loadStyleString(css){
    var style = document.createElement("style");
```

```
style.type = "text/css";
try{
    style.appendChild(document.createTextNode(css));
} catch (ex){
    style.styleSheet.cssText = css;
}
var head = document.getElementsByTagName("head")[0];
head.appendChild(style);
}
```

DynamicStyleExample02.htm

调用这个函数的示例如下:

```
loadStyleString("body{background-color:red}");
```

这种方式会实时地向页面中添加样式, 因此能够马上看到变化。



如果专门针对 IE 编写代码, 务必小心使用 `styleSheet.cssText` 属性。在重用同一个 `<style>` 元素并再次设置这个属性时, 有可能会造成浏览器崩溃。同样, 将 `cssText` 属性设置为空字符串也可能导致浏览器崩溃。我们希望 IE 中的这个 bug 能够在将来被修复。

## 10.2.3 操作表格

`<table>` 元素是 HTML 中最复杂的结构之一。要想创建表格, 一般都必须涉及表示表格行、单元格、表头等方面的标签。由于涉及的标签多, 因而使用核心 DOM 方法创建和修改表格往往都免不了要编写大量的代码。假设我们要使用 DOM 来创建下面的 HTML 表格。

```
<table border="1" width="100%">
  <tbody>
    <tr>
      <td>Cell 1,1</td>
      <td>Cell 2,1</td>
    </tr>
    <tr>
      <td>Cell 1,2</td>
      <td>Cell 2,2</td>
    </tr>
  </tbody>
</table>
```

要使用核心 DOM 方法创建这些元素, 得需要像下面这么多的代码:

```
//创建 table
var table = document.createElement("table");
table.border = 1;
table.width = "100%";

//创建 tbody
var tbody = document.createElement("tbody");
```

```
table.appendChild(tbody);

//创建第一行
var row1 = document.createElement("tr");
tbody.appendChild(row1);
var cell1_1 = document.createElement("td");
cell1_1.appendChild(document.createTextNode("Cell 1,1"));
row1.appendChild(cell1_1);
var cell2_1 = document.createElement("td");
cell2_1.appendChild(document.createTextNode("Cell 2,1"));
row1.appendChild(cell2_1);

//创建第二行
var row2 = document.createElement("tr");
tbody.appendChild(row2);
var cell1_2 = document.createElement("td");
cell1_2.appendChild(document.createTextNode("Cell 1,2"));
row2.appendChild(cell1_2);
var cell2_2 = document.createElement("td");
cell2_2.appendChild(document.createTextNode("Cell 2,2"));
row2.appendChild(cell2_2);

//将表格添加到文档主体中
document.body.appendChild(table);
```

显然，DOM 代码很长，还有点不太好懂。为了方便构建表格，HTML DOM 还为<table>、<tbody>和<tr>元素添加了一些属性和方法。

为<table>元素添加的属性和方法如下。

- caption: 保存着对<caption>元素（如果有）的指针。
- tBodies: 是一个<tbody>元素的 HTMLCollection。
- tFoot: 保存着对<tfoot>元素（如果有）的指针。
- tHead: 保存着对<thead>元素（如果有）的指针。
- rows: 是一个表格中所有行的 HTMLCollection。
- createTHead(): 创建<thead>元素，将其放到表格中，返回引用。
- createTFoot(): 创建<tfoot>元素，将其放到表格中，返回引用。
- createCaption(): 创建<caption>元素，将其放到表格中，返回引用。
- deleteTHead(): 删除<thead>元素。
- deleteTFoot(): 删除<tfoot>元素。
- deleteCaption(): 删除<caption>元素。
- deleteRow(pos): 删除指定位置的行。
- insertRow(pos): 向 rows 集合中的指定位置插入一行。

为<tbody>元素添加的属性和方法如下。

- rows: 保存着<tbody>元素中行的 HTMLCollection。
- deleteRow(pos): 删除指定位置的行。
- insertRow(pos): 向 rows 集合中的指定位置插入一行，返回对新插入行的引用。

为<tr>元素添加的属性和方法如下。

- `cells`: 保存着<tr>元素中单元格的 `HTMLCollection`。
- `deleteCell(pos)`: 删除指定位置的单元格。
- `insertCell(pos)`: 向 `cells` 集合中的指定位置插入一个单元格, 返回对新插入单元格的引用。

使用这些属性和方法, 可以极大地减少创建表格所需的代码数量。例如, 使用这些属性和方法可以将前面的代码重写如下 (加阴影的部分是重写后的代码)。

```
//创建 table
var table = document.createElement("table");
table.border = 1;
table.width = "100%";

//创建 tbody
var tbody = document.createElement("tbody");
table.appendChild(tbody);

//创建第一行
tbody.insertRow(0);
tbody.rows[0].insertCell(0);
tbody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1,1"));
tbody.rows[0].insertCell(1);
tbody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2,1"));

//创建第二行
tbody.insertRow(1);
tbody.rows[1].insertCell(0);
tbody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1,2"));
tbody.rows[1].insertCell(1);
tbody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2,2"));

//将表格添加到文档主体中
document.body.appendChild(table);
```

在这次的代码中, 创建<table>和<tbody>的代码没有变化。不同的是创建两行的部分, 其中使用了 HTML DOM 定义的表格属性和方法。在创建第一行时, 通过<tbody>元素调用了 `insertRow()` 方法, 传入了参数 0——表示应该将插入的行放在什么位置上。执行这一行代码后, 就会自动创建一行并将其插入到<tbody>元素的位置 0 上, 因此就可以马上通过 `tbody.rows[0]` 来引用新插入的行。

创建单元格的方式也十分相似, 即通过<tr>元素调用 `insertCell()` 方法并传入放置单元格的位置。然后, 就可以通过 `tbody.rows[0].cells[0]` 来引用新插入的单元格, 因为新创建的单元格被插入到了这一行的位置 0 上。

总之, 使用这些属性和方法创建表格的逻辑性更强, 也更容易看懂, 尽管技术上这两套代码都是正确的。

## 10.2.4 使用 NodeList

理解 `NodeList` 及其“近亲”`NamedNodeMap` 和 `HTMLCollection`, 是从整体上透彻理解 DOM 的关键所在。这三个集合都是“动态的”; 换句话说, 每当文档结构发生变化时, 它们都会得到更新。因此, 它们始终都会保存着最新、最准确的信息。从本质上说, 所有 `NodeList` 对象都是在访问 DOM 文档时实时运行的查询。例如, 下列代码会导致无限循环:

```
var divs = document.getElementsByTagName("div"),
    i,
    div;

for (i=0; i < divs.length; i++){
    div = document.createElement("div");
    document.body.appendChild(div);
}
```

第一行代码会取得文档中所有<div>元素的 HTMLCollection。由于这个集合是“动态的”，因此只要有新<div>元素被添加到页面中，这个元素也会被添加到该集合中。浏览器不会将创建的所有集合都保存在一个列表中，而是在下一次访问集合时再更新集合。结果，在遇到上例中所示的循环代码时，就会导致一个有趣的问题。每次循环都要对条件 `i < divs.length` 求值，意味着会运行取得所有<div>元素的查询。考虑到循环体每次都会创建一个新<div>元素并将其添加到文档中，因此 `divs.length` 的值在每次循环后都会递增。既然 `i` 和 `divs.length` 每次都会同时递增，结果它们的值永远也不会相等。

如果想要迭代一个 `NodeList`，最好是使用 `length` 属性初始化第二个变量，然后将迭代器与该变量进行比较，如下面的例子所示：

```
var divs = document.getElementsByTagName("div"),
    i,
    len,
    div;

for (i=0, len=divs.length; i < len; i++){
    div = document.createElement("div");
    document.body.appendChild(div);
}
```

这个例子中初始化了第二个变量 `len`。由于 `len` 中保存着对 `divs.length` 在循环开始时的一个快照，因此就会避免上一个例子中出现的无限循环问题。在本章演示迭代 `NodeList` 对象的例子中，使用的都是这种更为保险的方式。

一般来说，应该尽量减少访问 `NodeList` 的次数。因为每次访问 `NodeList`，都会运行一次基于文档的查询。所以，可以考虑将从 `NodeList` 中取得的值缓存起来。

## 10.3 小结

DOM 是语言中立的 API，用于访问和操作 HTML 和 XML 文档。DOM1 级将 HTML 和 XML 文档形象地看作一个层次化的节点树，可以使用 JavaScript 来操作这个节点树，进而改变底层文档的外观和结构。

DOM 由各种节点构成，简要总结如下。

- 最基本的节点类型是 `Node`，用于抽象地表示文档中一个独立的部分；所有其他类型都继承自 `Node`。
- `Document` 类型表示整个文档，是一组分层节点的根节点。在 JavaScript 中，`document` 对象是 `Document` 的一个实例。使用 `document` 对象，有很多种方式可以查询和取得节点。
- `Element` 节点表示文档中的所有 HTML 或 XML 元素，可以用来操作这些元素的内容和特性。
- 另外还有一些节点类型，分别表示文本内容、注释、文档类型、CDATA 区域和文档片段。



访问 DOM 的操作在多数情况下都很直观，不过在处理<script>和<style>元素时还是存在一些复杂性。由于这两个元素分别包含脚本和样式信息，因此浏览器通常会将它们与其他元素区别对待。这些区别导致了在针对这些元素使用 innerHTML 时，以及在创建新元素时的一些问题。

理解 DOM 的关键，就是理解 DOM 对性能的影响。DOM 操作往往是 JavaScript 程序中开销最大的部分，而因访问 NodeList 导致的问题为最多。NodeList 对象都是“动态的”，这就意味着每次访问 NodeList 对象，都会运行一次查询。有鉴于此，最好的办法就是尽量减少 DOM 操作。