**第10章 DOM**

本章内容

* 理解包含不同层次的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

DOM（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描述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及微软公司创立的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DOM1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性并不一致。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和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：

1. <html>

2.   <head>

3.     <title>Sample Page</title>

4.   </head>

5.   <body>

6.     <p>Hello World!</p>

7.   </body>

8. </html>

9.

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。

图示

描述已自动生成

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：

1.     if (someNode.nodeType == Node.ELEMENT\_NODE) { // 在IE中无效

2.       alert("Node is an element");

3.     }

4.

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。然后，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：

1. if (someNode.nodeType == 1) { // 适用于所有浏览器

2.       alert("Node is an element");

3.     }

4.

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。

1. if (someNode.nodeType == 1) {

2.       value = someNode.nodeName; // nodeName的值是元素的标签名

3.     }

4.

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中保存的始终都是元素的标签名，而nodeValue的值则使用为null。

1. **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是有生命的、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果访问保存在NodeList中的节点---可以通过方括号，也可以使用item()方法。

1.     var firstChild = someNode.childNodes[0];

2.     var secondChild = someNode.childNodes.item(1);

3.     var count = someNode.childNodes.length;

4.

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：

1.     // 在IE8及之前的版本中无效

2.     var arrayOfNodes = Array.prototype.slice.call(someNode.childNodes, 0);

3.

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Jscript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：

1.     function convertToArray(nodes) {

2.       var array = null;

3.       try {

4.         array = Array.prototype.slice(nodes, 0); // 针对非IE浏览器

5.       } catch (ex) {

6.         array = new Array();

7.         for (var i = 0, len = nodes.length; i < len; i++) {

8.           array.push(nodes[i]);

9.         }

10.       }

11.

12.       return array;

13.     }

14.

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属性和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：

1.       if (someNode.nextSibling === null) {

2.         alert("Last node in the parent's childNodes list.");

3.       } else if (someNode.previousSibling === null) {

4.         alert("First node is the parent's childNodes list.");

5.       }

6.

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.childeNodes.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。

手机屏幕截图

中度可信度描述已自动生成

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个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。

1. **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：

1.       var returnedNode = someNode.appendChild(newNode);

2.       alert(returnedNode == newNode); // true

3.       alert(someNode.lastChild == newNode); // true

4.

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。

1.       // someNode有多个子节点

2.       var returnedNode = someNode.appendChild(someNode.firstChild);

3.       alert(returnedNode == someNode.firstChild); // false

4.       alert(returnedNode == someNode.lastChild); // true

5.

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。

1.       // 插入后成为最后一个子节点

2.       returnedNode = someNode.insertBefore(newNode, null);

3.       alert(returnNode == someNode.lastChild); // true

4.

5.       // 插入后成为第一个子节点

6.       returnedNode = someNode.insertBefore(newNode, someNode.firstChild);

7.       alert(returnNode == newNode); // true

8.       alert(newNode == someNode.firstChild); // true

9.

10.       // 插入到最后一个子节点前面

11.       returnNode = someNode.insertBefore(newNode, someNode.lastChild);

12.       alert(newNode == someNode.childNodes[someNode.childNodes.length - 2]); // true

13.

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。

1.       // 替换第一个子节点

2.       var returnedNode = someNode.replaceChild(newNode, someNode.firstChild);

3.

4.       // 替换最后一个子节点

5.       returnedNode = someNode.replaceChild(newNode, someNode.lastChild);

6.

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自己的位置。

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，既要移除的节点。被移除的节点将成为方法的返回值。如下面的例子所示。

1.       // 移除第一个子节点

2.       var returnedNode = someNode.removeChild(someNode.firstChild);

3.

4.       // 移除最后一个子节点

5.       var returnedNode = someNode.removeChild(someNode.lastChild);

6.

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自己的位置。

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。

1. **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，假设有下面的HTML代码。

1.       <ul>

2.         <li>item 1</li>

3.         <li>item 2</li>

4.         <li>item 3</li>

5.       </ul>

6.

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。

1. 1.       var deepList = myList.cloneNode(true);

2. 2.       alert(deepList.childNodes.length); // 3 （IE < 9）或 7（其他浏览器）

3. 3.

4. 4.       var shallowList = myList.cloneNode(false);

5. 5.       alert(shallowList.childNodes.length); // 0

6. 6.

在这个例子中，deepList中保存着一个对myList执行深度复制得到的副本。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串的方式不一样。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节点的子节点可以是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<hmtl>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。

1. <html>

2.   <body>

3.

4.   </body>

5. </html>

6.

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。

1.       var html = document.documentElement; // 取得对<html>的引用

2.       alert(html === document.childNodes[0]); // true

3.       alert(html === document.firstChild); // true

4.

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。

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

2.

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（在浏览器中式document.doctype）来访问它的信息。

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

2.

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。以下面接单的HTML页面为例。

1. <!-- 第一条注释 -->

2. <html>

3.   <body>

4.

5.   </body>

6. </html>

7. <!-- 第二条注释 -->

8.

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，第一条注释就成为了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）。

1. **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对象所表示的网页的一些信息。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属性的值会改变<title>元素。来看下面的例子。

1.       // 取得文档标题

2.       var originalTitle = document.title;

3.

4.       // 设置文档标题

5.       document.title = "New page title";

6.

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。所有这些信息都能存在于请求的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。

1.       // 取得完整的URL

2.       var url = document.URL;

3.

4.       // 取得域名

5.       var domain = document.domain;

6.

7.       // 取得来源页面的URL

8.       var referrer = document.referrer;

9.

URL与domain属性是相互关联的。例如，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domian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。

1.       // 假设页面来自p2p.wrox.com域

2.       document.domain = "wrox.com"; // success

3.

4.       document.domain = "nczonline.net"; // failure

5.

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，这些页面就可以相互访问对方包含的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，即如果域名一开始就是“松散的”（loose），那么不能将它再设置为“紧绷的”（tight）。话句话说，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。

1.       // 假设页面来自于p2p.wrox.com域

2.       document.domain = "wrox.com"; // 松散的（成功）

3.       document.domain = "p2p.wrox.com"; // 紧绷的（出错了！）

4.

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。

1. **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。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。

1. <div id="myDiv">Some text</div>

2.

可以使用下面的代码取得这个元素：

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

2.

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。

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

2.

IE8及较低版本不区分ID的大小写，因此“myDiv”和“mydiv”会被当做相同元素的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（<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。

1.     <input type="text" name="myElement" value="Text field">

2.     <div id="myElement">A div</div>

3.

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，下列代码会取得页码中所有的<img>元素，并返回一个HTMLCollection。

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

2.

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。

1.       alert(images.length); // 输出图像的数量

2.       alert(images[0].src); // 输出第一个图像元素的src特性

3.       alert(images.item(0).src); // 输出第一个图像元素的src特性

4.

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：

1. <img src="myimage.gif" alt="" name="myImage">

2.

那么就可以通过如下方式从images变量中取得这个<img>元素：

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

2.

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：

1. var myImage = images["myImage"];

2.

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。

1. var allElements = document.getElementsByTagName("\*");

2.

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getElementsByTagName(“\*”)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getElementsByTagName()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），getElementsTagName()方法就会区分大小写。

第三个方法，也是只有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。

1.     <fieldset>

2.       <legend>Which color do you prefer?</legend>

3.       <ul>

4.         <li>

5.           <input type="radio" value="red" name="color" id="colorRed">

6.           <label for="colorRed">Red</label>

7.         </li>

8.         <li>

9.           <input type="radio" value="green" name="color" id="colorGreen">

10.           <label for="colorRed">Green</label>

11.         </li>

12.         <li>

13.           <input type="radio" value="blue" name="color" id="colorBlue">

14.           <label for="colorRed">Blue</label>

15.         </li>

16.       </ul>

17.     </fieldset>

18.

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：

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

2.

与getElementsByTagName()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）。

1. **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，为了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.form，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。

1. **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的名称和版本的功能，则该方法返回true，如下面的例子所示：

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

2.

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 | 样式表关联的文档 |
| 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。

1. **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。

1. <!DOCTYPE html>

2. <html lang="en">

3.   <head>

4.     <meta charset="UTF-8" />

5.     <meta name="viewport" content="width=device-width, initial-scale=1.0" />

6.     <title>document.write() Example 1</title>

7.   </head>

8.   <body>

9.     <p>The current date and time is:

10.       <script type="text/javascript">

11.         **document.write("<strong>" + (new Date()).toString() + "</strong>");**

12.       </script>

13.     </p>

14.     <script type="text/javascript">

15.

16.     </script>

17.   </body>

18. </html>

19.

DocumentWriteExample01.html

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()方法输出的任何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析为脚本块的结束，它后面的代码将无法执行。

1. <!DOCTYPE html>

2. <html lang="en">

3.   <head>

4.     <meta charset="UTF-8" />

5.     <meta name="viewport" content="width=device-width, initial-scale=1.0" />

6.     <title>document.write() Example 2</title>

7.   </head>

8.   <body>

9.     <script type="text/javascript">

10.       document.write("<script type='text\/javascript' src='file.js'" +

11.         "</script>");

12.     </script>

13.   </body>

14. </html>

15.

DocumentWriteExample02.html

即使这个文件看起来没错，但字符串“</script>”将被解析为与外部<script>标签匹配，结果文本");将会出现在页面中。为避免这个问题，只需加入转义字符\即可；第2章也曾经提及这个问题，

解决方案如下。

1. <!DOCTYPE html>

2. <html lang="en">

3.   <head>

4.     <meta charset="UTF-8" />

5.     <meta name="viewport" content="width=device-width, initial-scale=1.0" />

6.     <title>document.write() Example 3</title>

7.   </head>

8.   <body>

9.     <script type="text/javascript">

10.       document.write("<script type='text\/javascript' src='file.js'" +

11.         "**<\/script>**");

12.     </script>

13.   </body>

14. </html>

15.

DocumentWriteExample03.html

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：

1. <!DOCTYPE html>

2. <html lang="en">

3.   <head>

4.     <meta charset="UTF-8" />

5.     <meta name="viewport" content="width=device-width, initial-scale=1.0" />

6.     <title>document.write() Example 4</title>

7.   </head>

8.   <body>

9.     <p>This is some content that you won't get to see because it till be overwritten.</p>

10.     <script type="text/javascript">

11.       window.onload = function () {

12.         document.write("Hello World");

13.       };

14.     </script>

15.   </body>

16. </html>

17.

DocumentWriteExample04.html

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+xhmtl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节点具有一下特征：

* 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：

1. <div id="myDiv"></div>

2.

可以像下面这样取得这个元素及其标签名：

1.       var div = document.getElementById("myDiv");

2.       alert(div.tagName); // "DIV"

3.       alert(div.tagName == div.nodeName); // true

4.

这里的元素标签名是div，它拥有一个值为“myDiv”的ID。但是，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：

1.       if (element.tagName == "div") { // 不能这样比较，很容易出错！

2.         // 在此执行某些操作

3.       }

4.

5.       if (element.tagName.toLowerCase() == "div") { // 这样最好（适用于任何文档）

6.         // 在此执行某些操作

7.       }

8.

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及之前的版本。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：

1. <div id="myDiv" class="bd" title="body text" lang="en" dir="ltr"></div>

2.

HTMLElementsExample01.html

元素中指定的所有信息，都可通过下列JavaScript代码取得：

1.       var div = document.getElementById("myDiv");

2.       alert(div.id); // myDiv

3.       alert(div.className); // bd

4.       alert(div.title); // Body text

5.       alert(div.lang); // en

6.       alert(div.dir); // ltr

7.

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：

1.       div.id = "someOtherId";

2.       div.className = "ft";

3.       div.title = "Some other text";

4.       div.lang = "fr";

5.       div.dir = "rtl";

6.

HTMLElementsExample01.html

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 |
| META | HTMLMetaElement | STRONG | HTMLElement |
| NOFRAMES | HTMLElement | STYLE | HTMLStyleElement |
| NOSCRIPT | HTMLElement | SUB | HTMLElement |
| OBJECT | HTMLObjectElement | SUP | HTMLElement |
| OL | HTMLOListElement | 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*** | ***HTMLElement*** | TITLE | HTMLTitleElement |
| SAMP | HTMLElement | TR | HTMLTableRowElement |
| SCRIPT | HTMLScriptElement | TT | HTMLElement |
| SELECT | HTMLSelectElement | ***U*** | ***HTMLElement*** |
| SMALL | HTMLElement | UL | HTMLUListElement |
| SPAN | HTMLElement | VAR | HTMLElement |
| ***STRIKE*** | ***HTMLElement*** |  |  |

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。

1. **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removeAtrribute()。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：

1.       var div = document.getElementById("myDiv");

2.       alert(div.getAttribute("id")); // "myDiv"

3.       alert(div.getAttribute("class")); // "bd"

4.       alert(div.getAttribute("title")); // "Body text"

5.       alert(div.getAttribute("lang")); // "en"

6.       alert(div.getAttribute("dir")); // "ltr"

7.

注意，传递给getAttribute()的特性名与实际的特性名相同。因此想要得到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，以下面的元素为例：

1. <div id="myDiv" my\_special\_attribute="hello!"></div>

2.

这个元素包含了一个名为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：

1. var value = div.getAttribute("my\_special\_attribute");

2.

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，HTMLElement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：

1. <div id="myDiv" align="left" my\_special\_attribute="hello!"></div>

2.

因为id与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和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：

1.       alert(div.id); // myDiv

2.       alert(div.my\_special\_attribute); // undefined（IE除外）

3.       alert(div.align); // left

4.

ElementAttributesExample02.html

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本身就应该被赋予函数值。

由于存在这些差异，在通过JavaScript以编程方式操作DOM时，开发人员经常不使用getAttribute()，而是只是用对象的属性。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。

1. **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：

1.       div.setAttribute("id", "someOtherId");

2.       div.setAttribute("class", "ft");

3.       div.setAttribute("title", "Some other text");

4.       div.setAttribute("lang", "fr");

5.       div.setAttribute("dir", "rtl");

6.

ElementAttributesExample01.html

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。

1.       div.id = "someOtherId";

2.       div.align = "left";

3.

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

1.       div.mycolor = "red";

2.       alert(div.getAttribute("mycolor")); // null（IE除外）

3.

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楚特性的值，而且也会从元素中完全删除特性，如下所示：

1.       div.removeAttribute("class");

2.

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()。

1. **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，可以用以下代码。

1. var id = div.attributes.getNamedItem("id").nodeValue;

2.

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。

1. var id = div.attributes["id"].nodeValue;

2.

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。

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

2.

调用removeNamedItem()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。

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

2.

最后，setNamedItem()是一个很不常用的方法，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。

1. element.attributes.setNamedItem(newAttr);

2.

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属性倒是可以排上用场。在需要将DOM解构序列化为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。

1.       function outputAttributes(element) {

2.         var pairs = new Array(),

3.           attrName,

4.           attrValue,

5.           i,

6.           len;

7.         for (i = 0, len = element.attributes.length; i < len; i++) {

8.           attrName = element.attributes[i].nodeName;

9.           attrValue = element.attributes[i].nodeValue;

10.           pairs.push(attrName + "=\"" + attrValue + "\"");

11.         }

12.

13.         pairs.join(" ");

14.       }

15.

ElementAttributesExample03.html

这个函数使用了一个数字来保存名值对，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。

1.       function outputAttributes(element) {

2.         var pairs = new Array(),

3.           attrName,

4.           attrValue,

5.           i,

6.           len;

7.         for (i = 0, len = element.attributes.length; i < len; i++) {

8.           attrName = element.attributes[i].nodeName;

9.           attrValue = element.attributes[i].nodeValue;

10.           **if (element.attributes[i].specified) {**

11.             **pairs.push(attrName + '="' + attrValue + '"');**

12.           **}**

13.         }

14.

15.         pairs.join(" ");

16.       }

17.

ElementAttributesExample04.html

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。

1. **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。

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

2.

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。

1.       div.id = "myNewDiv";

2.       div.className = "box";

3.

在新元素上设置这些特性只是给他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。

1. document.appendChild(div);

2.

CreateElementExample01.html

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。

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

2.

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属性值为“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。

1.       if (client.brower.ie && client.brower.ie <= 7) {

2.         // 创建一个带name特性的iframe元素

3.         var iframe = document.createElement("<iframe name=\"myFrame\"></iframe>");

4.

5.         // 创建input元素

6.         var input = document.createElement("<input type=\"checkbox\">");

7.

8.         // 创建button元素

9.         var button = document.createElement("<button type=\"reset\"></button>");

10.

11.         // 创建单选按钮

12.         var radio1 = document.createElement("<input type=\"radio\" name=\"choice\" value=\"1\">");

13.         var radio2 = document.createElement("<input type=\"radio\" name=\"choice\" value=\"2\">");

14.       }

15.

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。

1. **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，以下面的代码为例。

1.     <ul id="myList">

2.       <li>Item 1</li>

3.       <li>Item 2</li>

4.       <li>Item 3</li>

5.     </ul>

6.

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。

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

2.

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，通常都要先检查以下nodeType属性，如下面的例子所示。

1.       for (var i = 0, len = element.childNodes.length; i < len; i++) {

2.         if (element.childNodes[i].nodeType == 1) {

3.           // 执行某项操作

4.         }

5.       }

6.

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。

1.       var ul = document.getElementById("myList");

2.       var items = ul.getElementsByTagName("li");

3.

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，包含的是可以按照字面量解释得纯文本内容。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量。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。

1.     <!-- 没有内容，也就没有文本节点 -->

2.     <div></div>

3.

4.     <!-- 有空格，因而有一个文本节点 -->

5.     <div> </div>

6.

7.     <!-- 有内容，因而有一个文本节点 -->

8.     <div>Hello World！</div>

9.

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。

1. var textNode = div.firstChild; // 或者div.ChildNodes[0]

2.

在取得了文本节点的引用后，就可像下面这样来修改它了。

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

2.

TextNodeExample01.html

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。

1.       // 输出结果就是 "Some <strong>other</strong> message"

2.       div.firstChild.nodeValue = "Some <strong>other</strong> message";

3.

TextNodeExample02.html

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。

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

2.

在创建文本节点的同时，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。

1.       var element = document.createElement("div");

2.       element.className = "message";

3.

4.       var textNode = document.createTextNode("Hello World!");

5.       element.appendChild(textNode);

6.

7.       document.body.appendChild(element);

8.

TextNodeExample03.html

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创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。

1.       var element = document.createElement("div");

2.       element.className = "message";

3.

4.       var textNode = document.createTextNode("Hello World!");

5.       element.appendChild(textNode);

6.

7.       **var anotherTextNode = document.createTextNode("Yippee");**

8.       **element.appendChild(anotherTextNode);**

9.

10.       document.body.appendChild(element);

11.

TextNodeExample04.html

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。

1. **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（因为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。

1.       var element = document.createElement("div");

2.       element.className = "message";

3.

4.       var textNode = document.createTextNode("Hello World!");

5.       element.appendChild(textNode);

6.

7.       var anotherTextNode = document.createTextNode("Yippee");

8.       element.appendChild(anotherTextNode);

9.

10.       document.body.appendChild(element);

11.

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

13.

14.       element.normalize(); // 1

15.       alert(element.childNodes.length);

16.       alert(element.firstChild.nodeValue); // Hello World!Yippee

17.

TextNodeExample05.html

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的崩溃。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。

1. **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隔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。

1.       var element = document.createElement("div");

2.       element.className = "message";

3.

4.       var textNode = document.createTextNode("Hello World!");

5.       element.appendChild(textNode);

6.

7.       document.body.appendChild(element);

8.

9.       **var newNode = textNode.splitText(5);**

10.       **alert(element.firstChild.nodeValue);** // "Hello"

11.       **alert(newNode.nodeValue);** // " world!"

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

13.

TextNodeExample06.html

这个例子中，包含“Hello world！”的文本节点被分隔为两个两个文本节点，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。

1.     <div id="myDiv"><!-- A Comment --></div>

2.

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。

1.       var element = document.createElement("div");

2.       var comment = element.firstChild;

3.       alert(comment.data); // "A Comment"

4.

CommentNodeExample01.html

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。

1.       var comment = document.createComment("A Comment");

2.

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，因为多数浏览器都会把CDATA区域错误地解析为Comment或Element。以下面的代码为例：

1.     <div id="myDiv">**<![CDATA[This is some conten.]]>**</div>

2.

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及之前的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。DocumentType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HTML4.01的文档类型声明为例：

1. <!DOCTYPE HTML PUBLIC

2. "-//W3C//DTD HTML 4.01//EN"

3. "http://www.3c.org/TR/html4/strict.dtd">

4.  alert(docuemt.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，既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：

1. var fragment = document.createDocumentFragment();

2.

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：

1. <ul id="myList"></ul>

2.

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，然后再一次型将它们添加到文档中。

1.     var fragment = document.createDocumentFragment();

2.     var ul = document.getElementById("myList");

3.

4.     var li = null;

5.     for (var i = 0; i < 3; i++) {

6.       li = document.createElement("li");

7.       li.appendChild(document.createTextNode("Item " + (i + 1)));

8.       fragment.appendChild(li);

9.     }

10.

11.     ul.appendChild(fragment);

12.

DocumentFragmentExample01.html

在这个例子中，我们先创建一个文档片段并取得了对<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2；
* nodeName的值就是特性的名称；
* nodeValue的值就是特性的值；
* 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：

1.     var attr = document.createAttribute("align");

2.     attr.value = "right"

3.

4.     element.setAttributeNode(attr);

5.     alert(element.attributes["align"].value);

6.     alert(element.getAttribute("align"));

7.     alert(element.getAttributeNode("align").value);

8.

AttrExample01.html

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“right”。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生成的那些通常原本是用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。

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：

1.   <script type="text/javasc " src="client.js"></script>

2.

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：

1.     var script = document.createElement("script");

2.     script.type = "text/javascript";

3.     script.src = "client.js";

4.     document.body.appendChild(script);

5.

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：

1.       function loadScript(url) {

2.         var script = document.createElement("script");

3.         script.type = "text/javascript";

4.         script.src = url;

5.         document.body.appendChild(script);

6.       }

7.

然后，就可以通过调用这个函数来加载外部的JavaScript文件了：

1.       loadScript("client.js");

2.

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：

1.     <script type="text/javascript">

2.       function sayHi() {

3.         alert("hi");

4.       }

5.     </script>

6.

从逻辑上讲，下面的DOM代码是有效的：

1.       var script = document.createElement("script");

2.       script.type = "text/javascript";

3.       **script.appendChild(document.createTextNode("function sayHi() { alert('hi'); } sayHi();"));**

4.       document.body.appendChild(script);

5.

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：

1.       var script = document.createElement("script");

2.       script.type = "text/javascript";

3.       **script.text = "function sayHi() { alert('hi'); } sayHi();";**

4.       document.body.appendChild(script);

5.

DynamicScriptExample01.html

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：

1.       var script = document.createElement("script");

2.       script.type = "text/javascript";

3.       var code = "function sayHi() { alert('hi'); } sayHi();";

4.       **try {**

5.         **script.appendChild(document.createTextNode(code));**

6.       **} catch (ex) {**

7.         **script.text = code;**

8.       **}**

9.       document.body.appendChild(script);

10.

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：

1.       function loadScriptString(code) {

2.         var script = document.createElement("script");

3.         script.type = "text/javascript";

4.         try {

5.           script.appendChild(document.createTextNode(code));

6.         } catch (ex) {

7.           script.text = code;

8.         }

9.         document.body.appendChild(script);

10.       }

11.

下面是调用这个函数的示例：

1.       loadScriptString("function sayHi() { alert('Hi'); } sayHi();");

2.

DynamicScriptString02.html

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。

我们以下面这个典型的<link>元素为例：

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

2.

使用DOM代码可以很容易地动态创建出这个元素：

1.       var link = document.createElement("link");

2.       link.rel = "stylesheet";

3.       link.type = "text/css";

4.       link.href = "styles.css";

5.       var head = document.getElementsByTagName("head")[0];

6.

7.       head.appendChild(link);

8.

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：

1.       function loadStyles(url) {

2.         var link = document.createElement("link");

3.         link.rel = "stylesheet";

4.         link.type = "text/css";

5.         link.href = url;

6.         var head = document.getElementsByTagName("head")[0];

7.

8.         head.appendChild(link);

9.       }

10.

调用loadStyles()函数的代码如下所示：

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：

1.     <style type="text/css">

2.       body {

3.         background-color: red;

4.       }

5.     </style>

6.

按照相同的逻辑，下列DOM代码应该是有效的：

1.       var style = document.createElement("style");

2.       style.type = "text/css";

3.       style.appendChild(document.createTextNode("body { background-color: red; }"));

4.       var head = document.getElementsByTagName("head")[0];

5.       head.appendChild(style);

6.

DynamicStyleExample01.html

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章将进一步讨论这个两个属性），如下面的例子所示。

1.       var style = document.createElement("style");

2.       style.type = "text/css";

3.       **try {**

4.

5.         style.appendChild(document.createTextNode("body { background-color: red; }"));

6.       **} catch (ex) {**

7.         **style.styleSheet.cssText= "body { background-color: red; }";**

8.       **}**

9.       var head = document.getElementsByTagName("head")[0];

10.       head.appendChild(style);

11.

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。

1.       function loadStyleString(css) {

2.         var style = document.createElement("style");

3.         style.type = "text/css";

4.         try {

5.           console.log(style);

6.           style.appendChild(

7.             document.createTextNode(css)

8.           );

9.         } catch (ex) {

10.           style.styleSheet.cssText = css;

11.         }

12.         var head = document.getElementsByTagName("head")[0];

13.         head.appendChild(style);

14.       }

15.

DynamicStyleExample02.html

调用这个函数的示例如下：

1.       loadStyleString("body { background-color: yellow; }");

2.

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，有可能会导致浏览器崩溃。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。

1.     <table border="1" width="100%">

2.       <tbody>

3.         <tr>

4.           <td>Cell 1, 1</td>

5.           <td>Cell 2, 1</td>

6.         </tr>

7.         <tr>

8.           <td>Cell 1, 2</td>

9.           <td>Cell 2, 2</td>

10.         </tr>

11.       </tbody>

12.     </table>

13.

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：

1.       // 创建table

2.       var table = document.createElement("table");

3.       table.border = 1;

4.       table.width = "100%";

5.

6.       // 创建tbody

7.       var tbody = document.createElement("tbody");

8.       table.appendChild(tbody);

9.

10.       // 创建第一行

11.       var row1 = document.createElement("tr");

12.       tbody.appendChild(row1);

13.       var cell1\_1 = document.createElement("td");

14.       cell1\_1.appendChild(document.createTextNode("Cell 1, 1"))

15.       row1.appendChild(cell1\_1);

16.       var cell1\_2 = document.createElement("td");

17.       cell1\_2.appendChild(document.createTextNode("Cell 2, 1"))

18.       row1.appendChild(cell1\_2);

19.

20.       // 创建第二行

21.       var row2 = document.createElement("tr");

22.       tbody.appendChild(row2);

23.       var cell2\_1 = document.createElement("td");

24.       cell2\_1.appendChild(document.createTextNode("Cell 2, 1"))

25.       row2.appendChild(cell2\_1);

26.       var cell2\_2 = document.createElement("td");

27.       cell2\_2.appendChild(document.createTextNode("Cell 2, 1"))

28.       row2.appendChild(cell2\_2);

29.

30.       document.body.appendChild(table);

31.

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<tfoot>元素（如果有）的指针。
* tHead：保存着<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）。

1.       // 创建table

2.       var table = document.createElement("table");

3.       table.border = 1;

4.       table.width = "100%";

5.

6.       // 创建tbody

7.       var tbody = document.createElement("tbody");

8.       table.appendChild(tbody);

9.

10.       // 创建第一行

11.       tbody.insertRow(0);

12.       tbody.rows[0].insertCell(0);

13.       tbody.rows[0].cells[0].appendChild(document.createTextNode("Cell 1, 1"));

14.       tbody.rows[0].insertCell(1);

15.       tbody.rows[0].cells[1].appendChild(document.createTextNode("Cell 2, 1"));

16.

17.       // 创建第二行

18.       tbody.insertRow(1);

19.       tbody.rows[1].insertCell(0);

20.       tbody.rows[1].cells[0].appendChild(document.createTextNode("Cell 1, 2"));

21.       tbody.rows[1].insertCell(1);

22.       tbody.rows[1].cells[1].appendChild(document.createTextNode("Cell 2, 2"));

23.

24.       document.body.appendChild(table);

25.

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，每当文档解构发生变化时，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：

1.       var divs = document.getElementsByTagName("div"),

2.         i,

3.         div;

4.       for (i = 0; i < divs.length; i++) {

5.         div = document.createElement("div");

6.         document.body.appendChild(div);

7.       }

8.

第一行代码中会取得文档中所有<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：

1.       var divs = document.getElementsByTagName("div"),

2.         i,

3.         len,

4.         div;

5.       **for (i = 0, len = divs.length; i < len; i++) {**

6.         div = document.createElement("div");

7.         document.body.appendChild(div);

8.       }

9.

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线循环问题。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，因而访问NodeList导致的问题最多。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。