**第11章 DOM扩展**

本章内容

* 理解Selectors API
* 使用HTML5 DOM扩展

了解专有的DOM扩展

尽管DOM作为API已经非常完善了，但为了实现更多的功能，仍然会有一些标准或专有的扩展。2008年之前，浏览器中几乎所有的DOM扩展都是专有的。此后，W3C着手将一些已经成为事实标准的专有扩展标准化并写入规范当中。

对DOM的两个主要的扩展是Selectors API（选择符API）和HTML5。这两个扩展都源自开发社区，而将某些常见的做法及API标准化一直是众望所归。此外，还有一个不那么引人瞩目的Element traversal（元素遍历）规范，为了DOM添加了一些属性。虽然前述两个主要规范（特别是HTML5）已经涵盖了大量的DOM扩展，但专有扩展依然存在。本章也会介绍专有的DOM扩展。

11.1 选择符API

众多JavaScript库中最常用的一项功能，就是根据CSS选择符选择与某个模式匹配的DOM元素。实际上，jQuery（www.jquery.com）的核心就是通过css选择符查询DOM文档取得元素的引用，从而抛开了getElementById()和getElementsByTagName()。

Selectors API（www.w3.org/TR/selectors-api/）是由W3C发起制定的一个标准，致力于让浏览器原生支持CSS查询。所有实现这一功能的JavaScript库都会写一个基础的CSS解析器然，然后再使用已有的DOM方法查询文档并找到匹配的节点。尽管开发人员在不只疲惫地改进这一过程的性能，但到头来都只能通过运行JavaScript代码来完成查询操作。而把这个功能变成原生API之后，解析和树查询操作可以在浏览器内部通过编译后的代码来完成，极大地改善了性能。

Selectors API Level 1的核心是两个方法：querySelector()和querySelectorAll()。在兼容的浏览器中，可以通过Document及Element类型的实例调用它们。目前已完全支持Selectors API Level 1的浏览器有IE8+、Firefox 3.5+、Safari 3.1+、Chrome和Opera 10+。

11.1.1 querySelector()方法

querySelector()方法接收一个CSS选择符，返回与该模式匹配的第一个元素，如果没有找到匹配的元素，返回null。请看下面的例子。

1.       // 取得body元素

2.       var body = document.querySelector("body");

3.

4.       // 取得ID为“myDiv”的元素

5.       var myDiv = document.querySelector("#myDiv");

6.

7.       // 取得类为“selected”的第一个元素

8.       var selected = document.querySelector(".selected");

9.

10.       // 取得类为“button”的第一个图像元素

11.       var img = document.querySelector("img.button");

12.

SelectorsAPIExample01.html

通过Document类型调用querySelector()方法时，会在该文档元素的范围内查找匹配的元素。而通过Element类型调用querySelector()方法时，只会在该元素后代元素的范围内查找匹配的元素。

CSS选择符可以简单也可以复杂，视情况而定。如果传入了不被支持的选择符，querySelector()会抛出错误。

11.1.2 querySelectorAll()方法

querySelectorAll()方法接收的参数与querySelector()方法一样，都是一个CSS选择符，但返回的是所有匹配的元素而不仅仅是一个元素。这个方法返回的是一个NodeList的实例。

具体来说，返回的值实际上是带有所有属性和方法的NodeList，而其底层实现则类似于一组元素的快照，而非不断对文档进行搜索的动态查询。这样实现可以避免使用NodeList对象通常会引起的大多数性能问题。

只要传给querySelectorAll()方法的CSS选择符有效，该方法都会返回一个NodeList对象，而不管找到多少匹配的元素。如果没有找到匹配的元素，NodeList就是空的。

与querySelector()类似，能够调用querySelectorAll()方法的类型包括Document、DocumentFragment和Element。下面是几个例子。

1.       // 取得某<div>中的所有<em>元素（类似于getElementsByTagName("em")）

2.       var ems = document.getElementById("myDiv").querySelectorAll("em");

3.

4.       // 取得类为“selected”的所有元素

5.       var selecteds = document.querySelectorAll(".selected");

6.

7.       // 取得所有<p>元素中的所有<strong>元素

8.       var strongs = document.querySelectorAll("p strong");

9.

SelectorsAPIExample02.html

要取得返回的NodeList中的每一个元素，可以使用item()方法，也可以使用方括号语法，比如：

1.       var i, len, strong;

2.       for (i = 0, len = strongs.length; i < len; i++) {

3.         strong = strongs[i]; // 或者strongs.item(i)

4.         string.className = "important";

5.       }

6.

同样与querySelector()类似，如果传入了浏览器不支持的选择符或者选择符中有语法错误，querySelectorAll()会抛出错误。

11.1.3 matchesSelector()方法

Selectors API Level 2规范为Element类型新增了一个方法matchesSelector()。这个方法接收一个参数，即CSS选择符，如果调用元素与该选择符匹配，返回true；否则，返回false。看例子。

1.       if (document.body.matchesSelector("boddy.page1")) {

2.         // true

3.       }

4.

在取得某个元素引用的情况下，使用这个方法能够方便地检测它是否会被querySelector()或querySelectorAll()方法返回。

截至2011年年中，还没有浏览器支持matchesSelector()方法；不过，也有一些实验性的实现。IE9+通过msMatchesSelector()支持该方法，Firefox 3.6+通过mozMatchesSelector()支持该方法，Safari 5+和Chrome通过webkitMatchesSelector()支持该方法。因此，如果你想使用这个方法，最好是编写一个包装函数。

1.       function matchesSelector(element, selector) {

2.         if (element.matchesSelector) {

3.           return element.matchesSelector(selector);

4.         } else if (element.msMatchesSelector) {

5.           return element.msMatchesSelector(selector);

6.         } else if (element.mozMatchesSelector) {

7.           return element.mozMatchesSelector(selector);

8.         } else if (element.webkitMatchesSelector) {

9.           return element.webkitMatchesSelector(selector);

10.         } else {

11.           throw new Error("Not supported.")

12.         }

13.       }

14.

15.       if (matchesSelector(document.body, "body.page1")) {

16.         // 执行操作

17.       }

18.

SelectorsAPIExample03.html

11.2 元素遍历

对于元素间的空格，IE9及之前版本不会返回文本节点，而其他所有浏览器都会返回文本节点。这样，就导致了在使用childNodes和firstChild等属性时的行为不一致。为了弥补这一差异，而同时又保持DOM规则不变，Element Tranversal规范（www.w3.org/TR/ElementTranversal）新定义了一组属性。

Element Tranversal API为DOM元素添加了以下5个属性。

* childElementCount：返回子元素（不包括文本节点和注释）的个数。
* firstElementChild：指向第一个子元素；firstChild的元素版。
* lastElementChild：指向最后一子元素；lastChild的元素版。
* previousElementSibling：指向前一个同辈元素；previousSibling的元素版。
* nextElementSibling：指向后一个同辈元素；nextSibling的元素版。

支持的浏览器为DOM元素添加了这些属性，利用这些元素不必担心空白文本节点，从而可以更方便地查找DOM元素了。

下面来看一个例子。过去，要跨浏览器遍历某元素的所有子元素，需要像下面这样写代码。

1.       var i,

2.         len,

3.         child = element.firstChild;

4.       while (child != element.lastChild) {

5.         if (child.nodeType == 1) { // 检查是不是元素

6.           processChild(child);

7.         }

8.         child = child.nextSibling;

9.       }

10.

而使用Element Tranversal新增的元素，代码会更简洁。

1.       var i,

2.         len,

3.         child = element.**firstElementChild**;

4.       while (child != element.**lastElementChild**) {

5.         processChild(child); // 已知其是元素

6.         child = child.**nextElementSibling**;

7.       }

8.

支持Element Tranversal规范的浏览器有IE9+、Firefox 3.5+、Safari 4+、Chrome和Opera。

11.3 HTML5

对于传统HTML而言，HTML5是一个叛逆。所有之前的版本对JavaScript接口的描述都不过三言两语，主要篇幅都用于都用于定义标记，与JavaScript相关的内容一概交由DOM规范去定义。

而HTML5规范则围绕如何使用新增标记定义了大量JavaScript API。其中一些API与DOM重叠，定义了浏览器应该支持的DOM扩展。

因为HTML5涉及的面非常广，本节只讨论与DOM节点相关的内容。HTML5的其他相关内容将在本书其他章节中穿插介绍。

11.3.1 与类相关的扩充

HTML4在web开发领域得到广泛采用后导致了一个很大的变化，即class属性用的越来越多，一方面可以通过它为元素添加样式，另一方面还可以用它表示元素的语义。于是，自然就有很多JavaScript代码会来操作CSS类，比如动态修改类或者搜索文档中具有给定类或给定的一组类的元素，等等。为了让开发人员适应并增加对class属性的新认识，HTML5新增了很多API，致力于简化CSS类的用法。

1. **getElementsByClassName()方法**

HTML5添加的getElementsByClassName()方法是最受人欢迎的一个方法，可以通过document对象及所有HTML元素调用该方法。这个方法最早出现在JavaScript库中，是通过既有的DOM功能实现的，而原生的实现具有极大的性能优势。

getElementsByClassName()方法接收一个参数，即一个包含一或多个类名的字符串，返回带有指定类的所有元素的NodeList。传入多个类名时，类名的先后顺序不重要。来看下面的例子。

1.       // 取得所有类中包含username和current的元素，类名的先后顺序无所谓

2.       var allCurrentUsername = document.getElementsByClassName("username current");

3.

4.       // 取得ID为myDiv的元素中带有类名selected的所有元素

5.       var selected = document.getElementById("myDiv").getElementsByClassName("selected");

6.

调用这个方法时，只有位于调用元素子树中的元素才会返回。在document对象上调用getElementsByClassName()始终会返回与类名匹配的所有元素，在元素上调用该方法就只会返回后代元素中匹配的元素。

使用这个方法可以更方便地为带有某些类的元素添加事件处理程序，从而不必再局限于使用ID或标签名。不过别忘了，因为返回的对象是NodeList，所以使用这个方法与使用getElementsByTagName()以及其他返回NodeList的DOM方法都具有同样的性能问题。

支持getElementsByClassName()方法的浏览器有IE 9+、Firefox 3+、Safari 3.1+、Chrome和Opera 9.5+。

1. **classList属性**

在操作类名是时，需要通过className属性添加、删除和替换类名。因为className中是一个字符串，所以即使只修改字符串一部分，也必须每次都设置整个字符串的值。比如，以下面的HTML代码为例。

1.     <div class="bd user disabled">...</div>

2.

这个<div>元素一共有三个类名。要从中删除一个类名，需要把这三个类名拆开，删除不想要的那个，然后再把其他类名拼成一个新字符串。请看下面的例子。

1.       // 删除user类

2.

3.       // 首先，取得类名字符串并拆分成数组

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

5.       var classNames = div.className.split(/\s+/);

6.

7.       // 找到要删除的类名

8.       var pos = -1,

9.         i,

10.         len;

11.       for (i = 0, len = classNames.length; i < len; i++) {

12.         if (classNames[i] == "user") {

13.           pos = i;

14.           break;

15.         }

16.       }

17.

18.       // 删除类名

19.       classNames.splice(i, 1);

20.

21.       // 把剩下的类名拼接成字符串并重新设置

22.       div.className = classNames.join(" ");

23.

为了从<div>元素的class属性中删除“user”，以上这些代码都是必须的。必须得通过类似的算法替换类名并确认元素中是否包含该类名。添加类名可以通过拼接字符串完成，但必须要通过检测确定不会多次添加相同得类名。很多JavaScript库都实现了这个方法，以简化这些操作。

HTML5新增了一种操作类名得方式，可以让操作更简单也更安全，那就是为所有元素添加classList属性。这个classList属性是新集合类型DOMTokenList的实例。与其他DOM集合类似，DOMTokenList有一个表示自己包含多少元素的length属性，而要取得每个元素可以使用item()方法，也可以使用方括号语法。此外，这个新类型还定义如下方法。

* add(value)：将给定的字符串值添加到列表中。如果值已经存在，就不添加了。
* contains(value)：表示列表中是否存在给定的值，如果存在则返回true，否则返回false。
* remove(value)：从列表中删除给定的字符串。
* toggle(value)：如果列表中已经存在给定的值，删除它；如果列表中没有给定的值，添加它。

这样，前面那么多行代码用下面这一行代码就可以代替了：

1.       div.classList.remove("user");

2.

以上代码能够确保其他类名不受此次修改的影响。其他方法也能极大地减少类似基本操作的复杂性，如下面的例子所示。

1.       // 删除disabled类

2.       div.classList.remove("disabled");

3.

4.       // 添加current类

5.       div.classList.add("current");

6.

7.       // 切换user类

8.       div.classList.toggle("user");

9.

10.       // 确定元素中是否包含既定的类名

11.       if (div.classList.contains("bd") && !div.classList.contains("disabled")) {

12.         // 执行操作

13.       }

14.

15.       // 迭代类名

16.       for (var i = 0, len = div.classList.length; i < len; i ++) {

17.         doSomething(div.classList[i])

18.       }

19.

有了classList属性，除非你需要全部删除所有类名，或者完全重写元素的class属性，否则也就用不到className属性了。

支持classList属性的浏览器有Firefox 3.5+和Chrome。

11.3.2 焦点管理

HTML5也添加了辅助管理DOM焦点的功能。首先就是document.activeElement属性，这个属性始终会引用DOM中当前获得了焦点的元素。元素获得焦点的方式有页面加载、用户输入（通常是通过按Tab键）和在代码中调用focus()方法。来看几个例子。

1.       var button = document.getElementById("myButton");

2.       button.focus();

3.       alert(document.activeElement === button); // true

4.

默认情况下，文档刚刚加载完成时，document.activeElement中保存的是document.body元素的引用。文档加载期间，document.activeElement的值为null。

另外就是新增了document.hasFocus()方法，这个方法用于确定文档是否获得了焦点。

1.       var button = document.getElementById("myButton");

2.       button.focus();

3.       alert(document.hasFocus()); // true

4.

通过检测文档是否获得了焦点，可以知道用户是不是正在与页面交互。

查询文档获知哪个元素获得了焦点，以及确定文档是否获得了焦点，这两个功能最重要的用途是提高Web应用的无障碍性。无障碍Web应用的一个主要标志就是恰当的焦点管理，而确切地知道哪个元素获得了焦点是一个极大的进步，至少我们不用再像过去那样靠猜测了。

实现了这两个属性的浏览器包括IE 4+、Firefox 3+、Safari 4+、Chrome和Opera 8+。

11.3.3 HTMLDocument的变化

HTML5扩展了HTMLDocument，增加了新的功能。与HTML5中新增的其他DOM扩展类似，这些变化同样基于那些已经得到很多浏览器完美支持的专有扩展。所以，尽管这些扩展被写入标准的时间相对不长，但很多浏览器很早就已经支持这些功能了。

1. **readyState属性**

IE4最早为document对象引入了readyState属性。然后，其他浏览器也都陆续添加这个属性，最终HTML5把这个属性纳入了标准当中。Document的readyState属性有两个可能的值：

* loading，正在加载文档；
* complete，已经加载完文档。

使用document.readyState的最恰当方式，就是通过它来实现一个指示文档已经加载完成的指示器。在这个属性得到广泛支持之前，要实现这样一个指示器，必须借助onload事件处理程序设置一个标签，表明文档已经加载完毕。document的readyState属性的基本用法如下。

1.       if (document.readyState == "complete") {

2.         // 执行操作

3.       }

4.

支持readyState属性的浏览器有IE4+、Firefox 3.6+、Safari、Chrome和Opera 9+。

1. **兼容模式**

自从IE6开始区分渲染页面的模式是标准的还是混杂的，检测页面的兼容模式就成为浏览器的必要功能。IE为此给document添加了一个名为compatMode的属性，这个属性就是为了告诉开发人员浏览器采用了哪种渲染模式。就像下面例子中所展示的那样，在标准模式下，document.compatMode的值等于“CSS1Compat”，而在混杂模式下，document.compatMode的值等于“BackCompat”。

1.       if (document.compatMode == "CSS1Compat") {

2.         alert("Standards mode");

3.       } else {

4.         alert("Quirks mode");

5.       }

6.

后来，陆续实现这个属性的浏览器有Firefox、Safari 3.1+、Opera和Chrome。最终，HTML5也把这个属性纳入标准，对其实现做出了明确规定。

1. **head属性**

作为对document.body引用文档的<body>元素的补充，HTML5新增了document.head属性，引用文档的<head>元素。要引用文档的<head>元素，可以结合使用这个属性和另一种后备方法。

1.       var head = document.head || document.getElementsByName("head")[0];

2.

如果可用，就是用document.head，否则仍然使用getElementsByTagName()方法。

实现document.head属性的浏览器包括Chrome和Safari 5。

11.3.4 字符集属性

HTML5新增了几个与文档字符集有关的属性。其中，charset属性表示文档中实际使用的字符集，也可以用来指定新字符集。默认情况下，这个属性的值为“UTF-16”，但可以通过<meta>元素、响应头部或直接设置charset属性修改这个值。来看一个例子。

1.       alert(document.charset); // UTF-16

2.       document.charset = "UTF-8";

3.

另一个属性是defaultCharset，表示根据默认浏览器及操作系统的设置，当前文档默认的字符集应该是什么。如果文档没有使用默认的字符集，那charset和defaultCharset属性的值可能会不一样，例如：

1.       if (document.charset != document.defaultCharset) {

2.         alert("Custom character set being used.");

3.       }

4.

通过这两个属性可以得到文档使用的字符编码的具体信息，也能对字符编码进行准确地控制。运行适当的情况下，可以保证用户正常查看页面或使用应用。

支持document.charset属性的浏览器有IE、Safari、Opera和Chrome。Firefox支持document.Characterset。支持document.defaultCharset属性的浏览器有IE、Safari和Chrome。

11.3.5 自定义数据属性

HTML5规定可以为元素添加非标准的属性，但要添加前缀data-，目的是为元素提供与渲染无关的信息，或者提供语义信息。这些属性可以任意添加、随便命名，只要以data-开头即可。来看一个例子。

1.     <div id="myDiv" **data-appId="1234" data-myname="Nicholas"**>...</div>

2.

添加了自定义属性之后，可以通过元素的dataset属性来访问自定义属性的值。dataset属性的值是DOMStringMap的一个实例，也就是一个名值对儿的映射。这个映射中，每个data-name形式的属性都会有一个对应的属性，只不过属性名没有data-前缀（比如，自定义属性是data-myname，那映射中对应的属性就是myname）。还是看一个例子吧。

1.       // 本例中使用的方法仅用于演示

2.

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

4.

5.       // 取得自定义属性的值

6.       var appId = div.dataset.appId;

7.       var myName = div.dataset.myname;

8.

9.       // 设置值

10.       div.dataset.appId = "23456";

11.       div.dataset.myname = "Michael";

12.

13.       // 有没有myname值呢？

14.       if (div.dataset.myname) {

15.         alert("Hello, " + div.dataset.myname);

16.       }

17.

如果需要给元素添加一些不可见的数据以便进行其他处理，那就要用到自定义数据属性。在跟踪链接或混搭应用中，通过自定义数据属性能方便地知道点击来自页面中的哪个部分。

在编写本书时，支持自定义数据属性的浏览器有Firefox 6+和Chrome。

11.3.6 插入标记

虽然DOM为操作节点提供了细致入微的控制手段，但在需要给文档插入大量新HTML标记的情况下，通过DOM操作仍然非常麻烦，因为不仅要创建一系列DOM节点，而且还要小心地按照正确的顺序把它们连接起来。相对而言，使用插入标记的技术，直接插入HTML字符串不仅更简单，速度也更快。以下与插入标记相关的DOM扩展已经纳入了HTML5规范。

1. **innerHTML属性**

在读模式下，innerHTML属性返回与调用元素的所有子节点（包括元素、注释和文本节点）对应的HTML标记。在写模式下，innerHTML会根据指定的值创建新的DOM树，然后用这个DOM树完全替换调用元素原先的所有子节点。下面是一个例子。

1.     <div id="content">

2.       <p>This is a <strong>paragraph</strong> with a list of following it.</p>

3.       <ul>

4.         <li>Item 1</li>

5.         <li>Item 2</li>

6.         <li>Item 3</li>

7.       </ul>

8.     </div>

9.

对于上面的<div>元素来说，它的innerHTML属性会返回如下字符串。

1. <p>This is a <strong>paragraph</strong> with a list of following it.</p> <ul> <li>Item 1</li> <li>Item 2</li> <li>Item 3</li> </ul>

2.

但是，不同浏览器返回的文本格式会有所不同。IE和Opera会将所有标签转换为大写形式，而Safari、Chrome和Firefox则会原原本本地按照原先文档中（或者指定这些标签时）的格式返回HTML，包括空格和缩进。不要指望所有浏览器返回的innerHTML值完全相同。

在写模式下，innerHTML的值会被解析为DOM子树，替换调用元素原来的所有子节点。因为它的值被认为是HTML，所以其中的所有标签都会按照浏览器处理HTML的标准方式转换为元素（同样，这里的转换结果也因浏览器而异）。如果设置的值仅是文本而没有HTML标签，那么结果就是设置纯文本，如下所示。

1.       div.innerHTML = "Hello world!";

2.

为innerHTML设置的包含HTML的字符值与解析后innerHTML的值大不相同。来看下面的例子。

1.       div.innerHTML = "Hello & welcome, <b>\"reader\"!</b>";

2.

以上操作得到的结果如下：

1. <div>Hello & welcome, <b>"reader"</b></div>

2.

设置了innerHTML之后，可以像访问文档中的其他节点一样访问新创建的节点。

为innerHTML设置HTML字符串后，浏览器会将这个字符串解析为相应的DOM树。因此设置了innerHTML之后，再从中读取HTML字符串，会得到与设置时不一样的结果。原因在于返回的字符串是根据原始HTML字符串创建的DOM树经过序列化之后的结果。

使用innerHTML属性也有一些限制。比如，在大多数浏览器中，通过innerHTML插入<script>元素并不会执行其中的脚本。IE8及更早版本是唯一能在这种情况下执行脚本的浏览器，但必须满足一些条件。一是必须为<script>元素指定defer属性，二是<script>元素必须位于（微软所谓的） “有作用域的元素”（scoped element）之后。<script>元素被认为是“无作用域的元素”（NoScoped element），也就是在页面中看不到的元素，与<style>元素或注释类似。如果通过innerHTML插入的字符串开头就是一个“无作用域的元素”，那么IE会在解析这个字符串前先删除该元素。换句话说，以下代码达不到目的：

1.       div.innerHTML = "<script defer>alert('he');<\/script>" // 无效

2.

此时，innerHTML字符串一开始（而且整个）就是一个“无作用域的元素”，所以这个字符串会变成空字符串。如果想插入这段脚本，必须在前面添加一个“有作用域的元素”，可以是一个文本节点，也可以是一个没有结束标签的元素如<input>。例如，下面这几行代码都可以正常执行：

1.       div.innerHTML = "\_<script defer>alert('he');<\/script>";

2.       div.innerHTML = "<div> <\/div><script defer>alert('he');<\/script>";

3.       div.innerHTML = "<input type=\"hidden\"><script defer>alert('he');<\/script>";

4.

第一行代码会在<script>元素前插入一个文本节点。事后，为了不影响页面显示，你可能需要移除这个文本节点。第二行代码采用的方法类似，只不过使用的是一个包含非换行空格的<div>元素。如果仅仅插入一个空的<div>元素，还是不行；必须要包含一点儿内容，浏览器才会创建文本节点。同样，为了不影响页面布局，恐怕还得移除这个节点。第三行代码使用的是一个隐藏的<input>域，也能达到相同的效果。不过，由于隐藏的<input>域不影响页面布局，因此这种方式在大多数情况下都是首选。

大多数浏览器都支持以直观的方式通过innerHTML插入<style>元素，例如：

1.       div.innerHTML = "<style type=\"text/css\">body { background-color: red; }<\/style>";

2.

但在IE8及更早版本中，<style>也是一个“没有作用域的元素”，因此必须像下面这样给它前置一个“有作用域的元素”：

1.       div.innerHTML = "\_<style type=\"text/css\">body { background-color: red; }<\/style>";

2.       div.removeChild(div.firstChild);

3.

并不是所有元素都支持innerHTML属性。不支持innerHTML的元素有：<color>、<colgroup>、<frameset>、<head>、<html>、<style>、<table>、<tbody>、<thead>、<tfoot>和<tr>。此外，在IE8及更早版本中，<title>元素也没有innerHTML属性。

Firefox对在内容类型为application/xhtml+xml的XHTML文档中设置innerHTML有严格的限制。在XHTML文档中使用innerHTML时，XHTML代码必须完全符合要求。如果代码格式不正确，设置innerHTML将会静默地失败。

无论什么时候，只要使用innerHTML从外部插入HTML，都应该首先以可靠的方式处理HTML。IE8为此提供了window.toStaticHTML()方法，这个方法接收一个参数，即一个HTML字符串；返回一个经过无害处理后的版本---从源HTML中删除所有脚本节点和事件处理程序属性。下面就是要给例子：

1.       var text = "<a href=\"#\" onclick=\"alert('hi')\">Click Me</a>";

2.       var sanitized = window.toStaticHTML(text); // Internet Explorer 8 only

3.       alert(sanitized);

4.

这个例子将一个HTML链接字符串传给了toStaticHTML()方法，得到的无害版本中去掉了onclick属性。虽然目前只有IE8原生支持这个方法，但我们还是建议读者在通过innerHTML插入代码之前，尽可能先手工检查一下其中的文本内容。

1. **outerHTML属性**

在读模式下，outerHTML返回调用它的元素及所有子节点的HTML标签。在写模式下，outerHTML会根据指定的HTML字符串创建新的DOM子树，然后用这个DOM子树完全替换调用元素。

1.     <div id="content">

2.       <p>This is a <strong>paragraph</strong> with a list of following it.</p>

3.       <ul>

4.         <li>Item 1</li>

5.         <li>Item 2</li>

6.         <li>Item 3</li>

7.       </ul>

8.     </div>

OtherHTMLExample01.html

如果在<div>元素上调用outerHTML，会返回与上面相同的代码，包括<div>本身。不过，由于浏览器解析和解释HTML标记的不同，结果也可能会有所不同。（这里的不同与使用innerHTML属性时存在的差异性质是一样的。）

使用outerHTML属性以下面这种方式设置值：

1.       div.outerHTML = "<p>This is paragraph.<\/p>"

2.

这行代码完成的操作与下面这些DOM脚本代码一样：

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

2.       p.appendChild(document.createTextNode("This is paragraph."));

3.       div.parentNode.replaceChild(p, div);

4.

结果，就是新创建的<p>元素会取代DOM树中的<div>元素。

支持outerHTML属性的浏览器有IE4+、Safari4+、Chrome和Opera8+。Firefox 7及之前版本都不知支持outerHTML属性。

1. **insertAdjacentHTML()**

插入标记的最后一个新增方式是insertAdjacentHTML()方法。这个方法最早也是在IE中出现的，它接收两个参数：插入位置和要插入的HTML文本。第一个参数必须是下列值之一：

* “beforebegin”，在当前元素之前插入一个紧邻的同辈元素；
* “afterbegin”，在当前元素之下插入一个新的子元素或在第一个子元素之前再插入新的子元素；
* “beforeend”，在当前元素之下插入一个新的子元素或在最后一个子元素之后再插入新的子元素；
* “afterend”，在当前元素之后插入一个紧邻的同辈元素。

注意，这些值都必须是小写形式。第二个参数是一个HTML字符串（与innerHTML和outerHTML的值相同），如果浏览器无法解析该字符串，就会抛出错误。以下是这个方法的基本用法示例。

1.       // 作为前一个同辈元素插入

2.       div.insertAdjacentElement("beforebegin", "<p>This is paragraph.<\/p>")

3.

4.       // 作为第一个子元素插入

5.       div.insertAdjacentElement("afterbegin", "<p>This is paragraph.<\/p>")

6.

7.       // 作为最后一子元素插入

8.       div.insertAdjacentElement("beforeend", "<p>This is paragraph.<\/p>")

9.

10.       // 作为后一个同辈元素插入

11.       div.insertAdjacentElement("afterend", "<p>This is paragraph.<\/p>")

12.

1. **内存与性能问题**

使用本节介绍的方法替换子节点可能会导致浏览器的内存占用问题，尤其是在IE中，问题更加明显。在删除带有事件处理程序或引用了其他JavaScript对象子树时，就有可能导致内存占用问题。假设某个元素有一个事件处理程序（或者引用了一个JavaScript对象作为属性），在使用前述某个属性将该元素从文档树中删除后，元素与事件处理程序（或JavaScript对象）之间的绑定关系在内存中并没有一并删除。如果这种情况频繁出现，页面占用的内存数量就会明显增加。因此，在使用innerHTML、outerHTML属性和insertAdjacentHTML()方法时，最好先手工删除要被替换的元素的所有事件处理程序和JavaScript对象属性（第13章将进一步讨论事件处理程序）。

不过，使用这几个属性---特别是使用innerHTML，仍然还是可以为我们提供很多遍历的。一般来说，在插入大量新HTML标记时，使用innerHTML属性与通过多次DOM操作先创建节点再指定它们之间的关系相比，效率要高得多。这是因为在设置innerHTML或outerHTML时，就会创建一个HTML解析器。这个解析器是在浏览器级别的代码（通常是C++编写的）基础上运行的，因此比执行JavaScript快得多。不可避免地，创建和销毁HTML解析器也会带来性能损失，所以最好能够将设置innerHTML或outerHTML的次数控制在合理的范围内。例如，下列代码使用innerHTML创建了很多列表项：

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

2.         ul.innerHTML += "<li>" + values[i] + "<\/li>"; // 要避免这种频繁操作！！

3.       }

4.

这种每次循环都设置一次innerHTML的做法效率很低。而且，每次循环还要从innerHTML中读取一次信息，就意味着每次循环要访问两次innerHTML。最好的做法是单独构建字符串，然后再一次性地将结果字符串赋值给innerHTML，像下面这样：

1.       var itemsHTML = "";

2.       for (var i = 0, len = values.length; i < len; i++) {

3.         itemsHTML += "<li>" + values[i] + "<\/li>"; // 要避免这种频繁操作！！

4.       }

5.

6.       ul.innerHTML = itemsHTML;

7.

这个例子的效率要高得多，因为它只对innerHTML执行了一次赋值操作。

11.3.7 scrollIntoView()方法

如何滚动页面也是DOM规范没有解决的一个问题。为了解决这个问题，浏览器实现了一些方法，以方便开发人员更好地控制页面滚动。在各种专有方法中，HTML5最终选择了scrollIntoView()作为标准方法。

scrollInteView()可以在所有HTML元素上调用，通过滚动浏览器窗口或某个容器元素，调用元素就可以出现在视口中。如果给这个方法传入true作为参数，或者不传入任何参数，那么窗口滚动之后会让调用元素的顶部与视口顶部尽可能平齐。如果传入false作为参数，调用元素会尽可能全部出现在视口中，（可能的话，调用元素的底部会与视口顶部平齐。）不过顶部不一定平齐，例如：

1.       // 让元素可见

2.       document.forms[0].scrollIntoView();

3.

当页面发生变化时，一般会用这个方法来吸引用户的注意力。实际上，为某个元素设置焦点也会导致浏览器滚动并显示出获得焦点的元素。

支持scrollIntoView()方法的浏览器有IE、Firefox、Safari和Opera。

11.4 专有扩展

虽然所有浏览器开发商都知晓坚持标准的重要性，但在发现某项功能缺失时，这些开发商都会一如既往地向DOM中添加专有扩展，以弥补功能上的不足。表面上看，这种各行其事的做法似乎不太好，但实际上专有扩展为Web开发领域提供了很多重要的功能，这些功能最终都在HTML5规范中得到了标准化。

即便如此，仍然还有大量专有的DOM扩展没有成为标准。但这并不是说它们将来不会被写进标准，而只是说在编写本书的时候，它们还是专有功能，而且只得到了少数浏览器的支持。

11.4.1 文档模式

IE8引入了新的概念叫“文档模式”（document mode）。页面的文档模式决定了可以使用什么功能。换句话说，文档模式决定了你可以使用哪个级别的CSS，可以在JavaScript中使用哪些API，以及如何对待文档类型（doctype）。到了IE9，总共有以下4中文档模式。

* IE5：以混杂模式渲染页面（IE5的默认模式就是混杂模式）。IE8及更高版本中的新功能都无法使用。
* IE7：以IE7标准模式渲染页面。IE8及更高版本中的新功能都无法使用。
* IE8：以IE8标准模式渲染页面。IE8中的新功能都可以使用，因此可以使用Selectors API、更多CSS2级选择符和某些CSS3功能，还有一些HTML5的功能。不过IE9中的新功能无法使用。
* IE9：以IE9标准模式渲染页面。IE9中的新功能都可以使用，比如ECMAScript 5、完整的CSS3以及更多HTML5功能。这个文档模式是最高级的模式。

要理解IE8及更高版本的工作原理，必须理解文档模式。

要强制浏览器以某种模式渲染页面，可以使用HTTP头部信息X-UA-Compatible，或通过等价的<meta>标签来设置：

1.     <meta http-equiv="X-UA-Compatible" content="IE=IEVersion">

2.

注意，这里IE的版本（IEVersion）有以下一些不同的值，而且这些值并不一定与上述4种文档模式对应。

* Edge：始终以最新的文档模式来渲染页面。忽略文档类型声明。对于IE8，始终保持以IE8标准模式渲染页面。对于IE9、则以IE9标准模式渲染页面。
* EmulateIE9：如果有文档类型声明，则以IE9标准模式渲染页面，否则将文档模式设置为IE5。
* EmulateIE8：如果有文档类型声明，则以IE8标准模式渲染页面，否则将文档模式设置为IE5。
* EmulateIE7：如果有文档类型声明，则以IE7标准模式渲染页面，否则将文档模式设置为IE5。
* 9：强制以IE9标准模式渲染页面，忽略文档类型声明。
* 8：强制以IE8标准模式渲染页面，忽略文档类型声明。
* 7：强制以IE7标准模式渲染页面，忽略文档类型声明。
* 5：强制将文档模式设置为IE5，忽略文档类型声明。

比如，要想让文档模式像在IE7种一样，可以使用下面这行代码：

1.     <meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7">

2.

如果不打算考虑文档类型声明，而直接使用IE7标准模式，那么可以使用下面这行代码：

1.     <meta http-equiv="X-UA-Compatible" content="IE=7">

2.

没有规定说必须在页面中设置X-UA-Compatible。默认情况下，浏览器会通过文档类型声明来确定使用最佳的可用文档模式，还是使用混杂模式。

通过document.documentMode属性可以知道给定页面使用的是什么文档模式。这个属性是IE8中新增的，它会返回使用的文档模式的版本号（在IE9中，可能返回的版本号为5、7、8、9）：

1.       var mode = document.documentMode;

2.

知道页面采用的是什么文档模式，有助于理解页面的行为方式。无论在什么模式下，都可以访问这个属性。

11.4.2 children属性

由于IE9之前的版本与其他浏览器在处理文本节点的空白符时有差异，因此就出现了children属性。这个属性是HTMLCollection的实例，只包含元素中同样还是元素的子节点。除此之外，children属性与childNodes没有什么区别，即在元素只包含元素子节点时，这两个属性的值相同。下面是访问children属性的示例代码：

1.       var childCount = element.children.length;

2.       var firstChild = element.children[0];

3.

支持children属性的浏览器有IE5、Firefox 3.5、Safari 2（但有bug）、Safari 3（完全支持）、Opera8和Chrome（所有版本）。IE8及更早版本的children属性中也会包含注释节点，但IE9之后的版本则只返回元素节点。

11.4.3 contains()方法

在实际开发中，经常需要知道某个节点是不是另一个节点的后代。IE为此率先引入了contains()方法，以便不通过在DOM文档树中查找即可获得这个信息。调用contains()方法的应该是祖先节点，也就是搜索开始的节点，这个方法接收一个参数，既要检测的后代节点。如果被检测的节点是后代节点，该方法返回true；否则，返回false。以下是一个例子：

1.       alert(document.documentElement.contains(document.body));

2.

这个例子测试了<body>元素是不是<html>元素的后代，在格式正确的HTML页面中，以上代码返回true。支持contains()方法的浏览器有IE、Firefox 9、Safari、Opara和Chrome。

使用DOM Level 3 compareDocumentPosition()也能够确定节点间的关系。支持这个方法的浏览器有IE9+、Firefox、Safari、Opera 9.5+和Chrome。如前所述，这个方法用于确定两个节点间的关系，返回一个表示该关系的位掩码（bitmask）。下表列出了这个位掩码的值。

|  |  |
| --- | --- |
| 掩码 | 节点关系 |
| 1 | 无关（给定的节点不在当前文档中） |
| 2 | 居前（给定的节点在DOM树中位于参考节点之前） |
| 4 | 居后（给定的节点在DOM树中位于参考节点之后） |
| 8 | 包含（给定的节点是参考节点的祖先 |
| 16 | 被包含（给定的节点是参考节点的后代） |

为模仿contains()方法，应该关注的是掩码16。可以对compareDocumentPosition()的结果执行按位与，以确定参考节点（调用compareDocumentPosition()方法的当前节点）是否包含给定的节点（传入的节点）。来看下面的例子：

1.       var result = document.documentElement.compareDocumentPosition(document.body);

2.       alert(!!(result & 16));

3.

执行上面的代码后，结果会变成20（表示“居后”的4加上表示“被包含”的16）。对掩码16执行按位操作会返回一个非零数值，而两个逻辑非操作符会将该数值转换为布尔值。

使用一些浏览器及能力检测，就可以写出如下所示的一个通用的contains函数：

1.       function contains(refNode, otherNode) {

2.         if (

3.           typeof refNode.contains == "function" &&

4.           (!client.engine.webkit || client.engine.webkit >= 522)

5.         ) {

6.           return refNode.contains(otherNode);

7.         } else if (typeof refNode.compareDocumentPosition == "funciton") {

8.           return !!(refNode.compareDocumentPosition(otherNode) & 16);

9.         } else {

10.           var node = otherNode.parentNode;

11.           do {

12.             if (node === refNode) {

13.               return true;

14.             } else {

15.               node = node.parentNode;

16.             }

17.           } while (node != null)

18.           return false;

19.         }

20.       }

21.

这个函数组合使用了三种方式来确定一个节点是不是另一个节点的后代。函数的第一个参数是参考节点，第二个参数是要检查的节点。在函数体内，首先检测refNode中是否存在contains()方法（能力检测）。这一部分代码还检查了当前浏览器所用的WebKit版本号。如果方法存在而且不是WebKit（!client.engine.webkit），则继续执行代码。否则如果浏览器是WebKit且至少是Safari3（WebKit版本号为522或更高），那么也可以继续执行代码。在WebKit版本号小于522的Safari浏览器中，contains()方法不能正常使用。

接下来检查是否存在compareDocumentPosition()方法，而函数的最后一步则是自otherNode开始向上遍历DOM结构，以递归方式取得parentNode，并检查其是否与refNode相等。在文档的顶端，parentNode的值等于null，于是循环结束。这是针对旧版本Safari设计的一个后备策略。

11.4.4 插入文本

前面介绍过，IE原来专有的插入标记的属性innerHTML和outerHTML已经被HTML5纳入规范。但另外两个插入文本的专有属性则没有这么好的运气。这两个没有被HTML5看中的属性是innerText和outerText。

1. innerText

通过innerText属性可以操作元素中包含的所有文本内容，包括子文档树中的文本。在通过innerText读取值时，它会按照由浅入深的顺序，将子文档树中的所有文本拼接起来。在通过innerText写入值时，结果会删除元素的所有子节点，插入包含相应文本值的文本节点。来看下面这个HTML代码示例。

1.     <div id="content">

2.       <p>This is a <strong>paragraph</strong> with a list of following it.</p>

3.       <ul>

4.         <li>Item 1</li>

5.         <li>Item 2</li>

6.         <li>Item 3</li>

7.       </ul>

8.     </div>

9.

InnerTextExample01.html

对于这个例子中的<div>元素而言，其innerText属性会返回下列字符串：

1. This is a paragraph with a list of following it.

2.

3. Item 1

4. Item 2

5. Item 3

6.

由于不同浏览器处理空白符的方式不同，因此输出的文本可能会也可能不会包含原始HTML代码中的缩进。

使用innerText属性设置这个<div>元素的内容，则只需一行代码：

1.       div.innerText = "Hello World!";

2.

InnerTextExample02.html

执行这行代码后，页面的HTML代码就会变成如下所示。

1.     <div id="content">Hello World!</div>

2.

设置innerText属性移除了先前存在的所有子节点，完全改变了DOM子树。此外，设置innerText属性的同时，也对文本中存在的HTML语法字符（小于号、大于号、引号及和号）进行了编码。再看一个例子。

1.       div.innerText = "Hello & welcome, <b>\"reader\"<\/b>";

2.

InnerTextExample03.html

运行以上代码之后，会得到如下所示的结果。

<div id="content">Hello &amp; welcome, &lt;b&gt;"reader"&lt;/b&gt;</div>

设置innerText永远只会生成当前节点的一个子文本节点，而为了确保只生成一个子文本节点，就必须要对文本进行HTML编码。利用这一点，可以通过innerText属性过滤掉HTML标签。方法是将innerText设置为等于innerText，这样就可以去掉所有HTML标签，比如：

1. div.innerText = div.innerText;

2.

执行这行代码后，就用原来的文本内容替换了容器元素中的所有内容（包括子节点，因而也就去掉了HTML标签）。

支持innerText属性的浏览器包括IE4+、Safari 3+、Opera 8+和Chrome。Firefox虽然不支持innerText，但支持作用类似的textContent属性。textContent是DOM Level 3规定的一个属性，其他支持textContent属性的浏览器还有IE9+、Safari 3+、Opera 10+和Chrome。为了确保跨浏览器兼容，有必要编写一个类似于下面的函数来检测可以使用哪个属性。

1.       function getInnerText(element) {

2.         return (typeof element.textContent == "string") ? element.textContent : element.innerText;

3.       }

4.

5.       function setInnerText(element, text) {

6.         if (typeof element.textContent == "string") {

7.           element.textContent = text;

8.         } else {

9.           element.innerText = text;

10.         }

11.       }

12.

InnerTextExample05.html

这两个函数都接收一个元素作为参数，然后检查这个元素是不是有textContent属性。如果有，那么typeof element.textContent应该是“string”；如果没有，那么这两个函数就会改为使用innerText。可以像下面这样调用这两个函数。

1.       setInnerText(div, "Hello World!");

2.       alert(getInnerText(div));

3.

使用这两个函数可以确保在不同的浏览器中使用正确的属性。

实际上，innerText与textContent返回的内容并不完全一样。比如，innerText会忽略行内的样式和脚本，而textContent则会像返回其他文本一样返回行内样式和脚本代码。避免跨浏览器兼容问题的最佳途径，就是从不包含行内样式或行内脚本的DOM子树副本或DOM片段中读取文本。

1. outerText

除了作用范围扩大到了包含调用它的节点之外，outerText与innerText基本上没有多大区别。在读取文本值时，outerText与innerText的结果完全一样。但在写模式下，outerText就完全不同了：outerText不只是替换调用它的元素的子节点，而是会替换整个元素（包括子节点）。比如：

1.       div.outerText = "Hello World!";

2.

这行代码实际上相当于如下两行代码：

1.       var text = document.createTextNode("Hello World!");

2.       div.parentNode.replaceChild(text, div);

3.

本质上，新的文本节点会完全取代调用outerText的元素。此后，该元素就从文档中被删除，无法访问。

支持outerText属性的浏览器有IE4+、Safari 3+、Opera 8+和Chrome。由于这个属性会导致调用它的元素不存在，因此并不常用。我们也建议读者尽可能不用使用这个属性。

11.4.5 滚动

如前所述，HTML 5之前的规范并没有就与页面滚动相关的API做出任何规定。但HTML5在将scollIntoView()纳入规范之后，仍然还有其他几个专有方法可以在不同的浏览器中使用。下面列出的几个方法都是对HTMLElement类型的扩展，因此在所有元素中都可以调用。

* scrollIntoViewIfNeeded(alignCenter)：只在当前元素在视口中不可见的情况下，才滚动浏览器窗口或容器元素，最终让它可见。如果当前元素在视口中可见，这个方法什么也不做。如果将可选的alignCenter参数设置为true，则表示尽量将元素显示在视口中部（垂直方向）。Safari和Chrome实现了这个方法。
* scrollByLines(lineCount)：将元素的内容滚动指定的行高，lineCount值可以是正值，也可以是负值。Safari和Chrome实现了这个方法。
* scrollByPages(pageCount)：将元素的内容滚动指定的页面高度，具体高度由元素的高度决定。Safari和Chrome实现了这个方法。

希望大家要注意的是，scrollIntoView()和scrollIntoViewIfNeeded()的作用对象是元素的容器，而scrollByLines()和scrollByPages影响的则是元素自身。下面还是来看几个示例吧。

1.       // 将页面主体滚动5行

2.       document.body.scrollByLines(5);

3.

4.       // 在当前元素下不可见的时候，让它进入浏览器的视口

5.       document.images[0].scrollIntoViewIfNeeded();

6.

7.       // 将页面主题往回滚动 1 页

8.       document.body.scrollByPages(-1);

9.

由于scrollIntoView()是唯一一个所有浏览器都支持的方法，因此还是这个方法最常用。

11.5 小结

虽然DOM为与XML及HTML文档交互制定了一系列核心API，但仍然有几个规范对标准的DOM进行了扩展。这些扩展中有很多原来是浏览器专有的，但后来成为了事实标准，于是其他浏览器也都提供了相同的实现。本章介绍的三个这方面的规范如下。

* Selectors API，定义了两个方法，让开发人员能够基于CSS选择符从DOM中取得元素，这两个方法是querySelector()和querySelectorAll()。
* Element Traversal，为DOM元素定义了额外的属性，让开发人员能够更方便地从一个元素跳到另一元素。之所以会出现这个扩展，是因为浏览器处理DOM元素间空白符的方式不一样。
* HTML5，为标准的DOM定义了很多扩展功能。其中包括在innerHTML属性这样的事实标准基础上提供的标准定义，以及为管理焦点、设置字符集、滚动页面而规定的扩展API。

虽然目前DOM扩展的数量还不多，但随着Web技术的发展，相信一定还会涌现出更多扩展来。很多浏览器都在试验专有的扩展，而这些扩展一旦获得认可，就能成为“伪”标准，甚至会被收录到规范的更新版本中。