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 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，然后再使用已有的 DOM 方法查询文档并找到匹配的节点。尽管库开发人员在不知疲倦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 IE 8+、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。

```
//取得 body 元素
var body = document.querySelector("body");

//取得 ID 为"myDiv"的元素
var myDiv = document.querySelector("#myDiv");

//取得类为"selected"的第一个元素
var selected = document.querySelector(".selected");

//取得类为"button"的第一个图像元素
var img = document.body.querySelector("img.button");
```

SelectorsAPIExample01.htm

通过 Doument 类型调用 querySelector()方法时，会在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。

```
//取得某<div>中的所有<em>元素（类似于 getElementsByTagName("em")）
var ems = document.getElementById("myDiv").querySelectorAll("em");

//取得类为"selected"的所有元素
var selecteds = document.querySelectorAll(".selected");

//取得所有<p>元素中的所有<strong>元素
var strongs = document.querySelectorAll("p strong");
```

SelectorsAPIExample02.htm

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：

```
var i, len, strong;
for (i=0, len=strongs.length; i < len; i++){
    strong = strongs[i];    //或者 strongs.item(i)
    strong.className = "important";
}
```

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。

```
if (document.body.matchesSelector("body.page1")){
    //true
}
```

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。IE 9+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。

```
function matchesSelector(element, selector){
    if (element.matchesSelector){
        return element.matchesSelector(selector);
    } else if (element.msMatchesSelector){
        return element.msMatchesSelector(selector);
    } else if (element.mozMatchesSelector){
        return element.mozMatchesSelector(selector);
    } else if (element.webkitMatchesSelector){
        return element.webkitMatchesSelector(selector);
    } else {
        throw new Error("Not supported.");
    }
}

if (matchesSelector(document.body, "body.page1")){
    //执行操作
}
```

SelectorsAPIExample03.htm

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 规范不变，Element Traversal 规范（www.w3.org/TR/ElementTraversal/）新定义了一组属性。

Element Traversal 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：指向最后一个子元素；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。

```
var i,
    len,
    child = element.firstChild;
while(child != element.lastChild){
    if (child.nodeType == 1){    //检查是不是元素
       processChild(child);
    }
    child = child.nextSibling;
}
```

而使用 Element Traversal 新增的元素，代码会更简洁。

```
var i,
    len,
    child = element.firstElementChild;
while(child != element.lastElementChild){
    processChild(child);    //已知其是元素
    child = child.nextElementSibling;
}
```

支持 Element Traversal 规范的浏览器有 IE 9+、Firefox 3.5+、Safari 4+、Chrome 和 Opera 10+。

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` 属性用得越来越多，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。

```
//取得所有类中包含"username"和"current"的元素，类名的先后顺序无所谓
var allCurrentUsernames = document.getElementsByClassName("username current");

//取得ID为"myDiv"的元素中带有类名"selected"的所有元素
var selected = document.getElementById("myDiv").getElementsByClassName("selected");
```

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+。

2. classList 属性

在操作类名时，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。

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

这个 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。

```
//删除"user"类

//首先，取得类名字符串并拆分成数组
var classNames = div.className.split(/\s+/);

//找到要删的类名
var pos = -1,
    i,
    len;
for (i=0, len=classNames.length; i < len; i++){
    if (classNames[i] == "user"){
        pos = i;
        break;
    }
}

//删除类名
classNames.splice(i,1);

//把剩下的类名拼成字符串并重新设置
div.className = classNames.join(" ");
```

为了从 元素的 class 属性中删除"user"，以上这些代码都是必需的。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的类名。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的方式，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：

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

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。

```
//删除"disabled"类
div.classList.remove("disabled");

//添加"current"类
div.classList.add("current");

//切换"user"类
div.classList.toggle("user");

//确定元素中是否包含既定的类名
if (div.classList.contains("bd") && !div.classList.contains("disabled")){
    //执行操作
)

//迭代类名
for (var i=0, len=div.classList.length; i < len; i++){
    doSomething(div.classList[i]);
}
```

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 3.6+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。

```
var button = document.getElementById("myButton");
button.focus();
alert(document.activeElement === button);    //true
```

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。

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

```
button.focus();
alert(document.hasFocus());  //true
```

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的包括 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.readyState 属性的基本用法如下。

```
if (document.readyState == "complete"){
    //执行操作
}
```

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

2. 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"。

```
if (document.compatMode == "CSS1Compat"){
    alert("Standards mode");
} else {
    alert("Quirks mode");
}
```

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。

3. 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。

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

如果可用，就使用 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。

```
alert(document.charset); //"UTF-16"
document.charset = "UTF-8";
```

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：

```
if (document.charset != document.defaultCharset){
    alert("Custom character set being used.");
}
```

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、Firefox、Safari、Opera 和 Chrome。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。

```
<div id="myDiv" data-appId="12345" data-myname="Nicholas"></div>
```

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。在这个映射中，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。

```
//本例中使用的方法仅用于演示

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

//取得自定义属性的值
var appId = div.dataset.appId;
var myName = div.dataset.myname;

//设置值
div.dataset.appId = 23456;
div.dataset.myname = "Michael";

//有没有"myname"值呢?
if (div.dataset.myname){
```

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

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。

```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

对于上面的 元素来说，它的 innerHTML 属性会返回如下字符串。

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

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。

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

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

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

以上操作得到的结果如下：

```
<div id="content">Hello &amp;amp; welcome, <b>&amp;quot;reader&amp;quot;!</b></div>
```

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”（NoScope 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：

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

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：

```
div.innerHTML = "_<script defer>alert('hi');<\/script>";
div.innerHTML = "<div> </div><script defer>alert('hi');<\/script>";
div.innerHTML = "<input type=\"hidden\"><script defer>alert('hi');<\/script>";
```

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 元素。如果仅仅插入一个空的 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：

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

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”：

```
div.innerHTML = "_<style type=\"text/css\">body {background-color: red; }</style>";
div.removeChild(div.firstChild);
```

并不是所有元素都支持 innerHTML 属性。不支持 innerHTML 的元素有：<col>、<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。下面就是一个例子：

```
var text = "<a href=\"#\" onclick=\"alert('hi')\">Click Me</a>";
var sanitized = window.toStaticHTML(text);        //Internet Explorer 8 only
alert(sanitized);    //"<a href=\"#\">Click Me</a>"
```

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。

2. 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。下面是一个例子。

```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

OuterHTMLExample01.htm

如果在 元素上调用outerHTML，会返回与上面相同的代码，包括 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：

```
div.outerHTML = "<p>This is a paragraph.</p>";
```

这行代码完成的操作与下面这些DOM脚本代码一样：

```
var p = document.createElement("p");
p.appendChild(document.createTextNode("This is a paragraph."));
div.parentNode.replaceChild(p, div);
```

结果，就是新创建的<p>元素会取代DOM树中的 元素。

支持outerHTML属性的浏览器有IE4+、Safari 4+、Chrome和Opera 8+。Firefox 7及之前版本都不支持outerHTML属性。

3. insertAdjacentHTML()方法

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。

```
//作为前一个同辈元素插入
element.insertAdjacentHTML("beforebegin", "<p>Hello world!</p>");

//作为第一个子元素插入
element.insertAdjacentHTML("afterbegin", "<p>Hello world!</p>");

//作为最后一个子元素插入
element.insertAdjacentHTML("beforeend", "<p>Hello world!</p>");

//作为后一个同辈元素插入
element.insertAdjacentHTML("afterend", "<p>Hello world!</p>");
```

支持 insertAdjacentHTML()方法的浏览器有 IE、Firefox 8+、Safari、Opera 和 Chrome。

4. 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便利的。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：

```
for (var i=0, len=values.length; i < len; i++){
    ul.innerHTML += "<li>" + values[i] + "</li>"; //要避免这种频繁操作！！
}
```

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：

```
var itemsHtml = "";
```

```
for (var i=0, len=values.length; i < len; i++){
    itemsHtml += "<li>" + values[i] + "</li>";
}
ul.innerHTML = itemsHtml;
```

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。

scrollIntoView()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：

```
//让元素可见
document.forms[0].scrollIntoView();
```

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 引入了一个新的概念叫“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 种文档模式。

- 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：

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

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 中一样，可以使用下面这行代码：

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

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：

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

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是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）：

```
var mode = document.documentMode;
```

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。无论在什么文档模式下，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中的空白符时有差异，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：

```
var childCount = element.children.length;
var firstChild = element.children[0];
```

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，即要检测的后代节点。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：

```
alert(document.documentElement.contains(document.body));     //true
```

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、Opera 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：

```
var result = document.documentElement.compareDocumentPosition(document.body);
alert(!!(result & 16));
```

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成布尔值。

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：

```
function contains(refNode, otherNode){
    if (typeof refNode.contains == "function" &&
            (!client.engine.webkit || client.engine.webkit >= 522)){
        return refNode.contains(otherNode);
    } else if (typeof refNode.compareDocumentPosition == "function"){
        return !!(refNode.compareDocumentPosition(otherNode) & 16);
    } else {
        var node = otherNode.parentNode;
        do {
            if (node === refNode){
                return true;
            } else {
                node = node.parentNode;
            }
        } while (node !== null);
```

```
            return false;
        }
    }
```

ContainsExample02.htm

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 Safari 3（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。在文档树的顶端，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 属性

通过 innertText 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。

```
<div id="content">
    <p>This is a <strong>paragraph</strong> with a list following it.</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
</div>
```

InnerTextExample01.htm

对于这个例子中的 元素而言，其 innerText 属性会返回下列字符串：

```
This is a paragraph with a list following it.
Item 1
Item 2
Item 3
```

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 元素的内容，则只需一行代码：

```
div.innerText = "Hello world!";
```

InnerTextExample02.htm

执行这行代码后，页面的 HTML 代码就会变成如下所示。

```
<div id="content">Hello world!</div>
```

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。

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

InnerTextExample03.htm

运行以上代码之后，会得到如下所示的结果。

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

```
div.innerText = div.innerText;
```

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。

```
function getInnerText(element){
    return (typeof element.textContent == "string") ?
        element.textContent : element.innerText;
}

function setInnerText(element, text){
    if (typeof element.textContent == "string"){
        element.textContent = text;
    } else {
        element.innerText = text;
    }
}
```

InnerTextExample05.htm

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。

```
setInnerText(div, "Hello world!");
alert(getInnerText(div));    //"Hello world!"
```

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的样式和脚本代码。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。

2. outerText 属性

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：

```
div.outerText = "Hello world!";
```

这行代码实际上相当于如下两行代码：

```
var text = document.createTextNode("Hello world!");
div.parentNode.replaceChild(text, div);
```

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要使用这个属性。

11.4.5 滚动

如前所述，HTML5 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 scrollIntoView()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。

```
//将页面主体滚动 5 行
document.body.scrollByLines(5);
```

```
//在当前元素不可见的时候，让它进入浏览器的视口
document.images[0].scrollIntoViewIfNeeded();

//将页面主体往回滚动 1 页
document.body.scrollByPages(-1);
```

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个元素。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。