（续）

元　素	类　型	元　素	类　型
H5	HTMLHeadingElement	PRE	HTMLPreElement
H6	HTMLHeadingElement	Q	HTMLQuoteElement
HEAD	HTMLHeadElement	*S*	*HTMLElement*
HR	HTMLHRElement	SAMP	HTMLElement
HTML	HTMLHtmlElement	SCRIPT	HTMLScriptElement
I	HTMLElement	SELECT	HTMLSelectElement
IFRAME	HTMLIFrameElement	SMALL	HTMLElement
IMG	HTMLImageElement	SPAN	HTMLElement
INPUT	HTMLInputElement	*STRIKE*	*HTMLElement*
INS	HTMLModElement	STRONG	HTMLElement
ISINDEX	*HTMLIsIndexElement*	STYLE	HTMLStyleElement
KBD	HTMLElement	SUB	HTMLElement
LABEL	HTMLLabelElement	SUP	HTMLElement
LEGEND	HTMLLegendElement	TABLE	HTMLTableElement
LI	HTMLLIElement	TBODY	HTMLTableSectionElement
LINK	HTMLLinkElement	TD	HTMLTableCellElement
MAP	HTMLMapElement	TEXTAREA	HTMLTextAreaElement
MENU	*HTMLMenuElement*	TFOOT	HTMLTableSectionElement
META	HTMLMetaElement	TH	HTMLTableCellElement
NOFRAMES	HTMLElement	THEAD	HTMLTableSectionElement
NOSCRIPT	HTMLElement	TITLE	HTMLTitleElement
OBJECT	HTMLObjectElement	TR	HTMLTableRowElement
OL	HTMLOListElement	TT	HTMLElement
OPTGROUP	HTMLOptGroupElement	*U*	*HTMLElement*
OPTION	HTMLOptionElement	UL	HTMLUListElement
P	HTMLParagraphElement	VAR	HTMLElement
PARAM	HTMLParamElement		

这里列出的每种类型都有关联的属性和方法。本书会涉及其中的很多类型。

2. 取得属性

每个元素都有零个或多个属性，通常用于为元素或其内容附加更多信息。与属性相关的 DOM 方法主要有 3 个：getAttribute()、setAttribute()和 removeAttribute()。这些方法主要用于操纵属性，包括在 HTMLElement 类型上定义的属性。下面看一个例子：

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

注意传给 getAttribute()的属性名与它们实际的属性名是一样的，因此这里要传"class"而非

"className"（className 是作为对象属性时才那么拼写的）。如果给定的属性不存在，则 getAttribute() 返回 null。

getAttribute()方法也能取得不是 HTML 语言正式属性的自定义属性的值。比如下面的元素：

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

这个元素有一个自定义属性 my_special_attribute，值为"hello!"。可以像其他属性一样使用 getAttribute()取得这个属性的值：

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

注意，属性名不区分大小写，因此"ID"和"id"被认为是同一个属性。另外，根据 HTML5 规范的要求，自定义属性名应该前缀 data-以方便验证。

元素的所有属性也可以通过相应 DOM 元素对象的属性来取得。当然，这包括 HTMLElement 上定义的直接映射对应属性的 5 个属性，还有所有公认（非自定义）的属性也会被添加为 DOM 对象的属性。比如下面的例子：

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

因为 id 和 align 在 HTML 中是 元素公认的属性，所以 DOM 对象上也会有这两个属性。但 my_special_attribute 是自定义属性，因此不会成为 DOM 对象的属性。

通过 DOM 对象访问的属性中有两个返回的值跟使用 getAttribute()取得的值不一样。首先是 style 属性，这个属性用于为元素设定 CSS 样式。在使用 getAttribute()访问 style 属性时，返回的是 CSS 字符串。而在通过 DOM 对象的属性访问时，style 属性返回的是一个（CSSStyleDeclaration）对象。DOM 对象的 style 属性用于以编程方式读写元素样式，因此不会直接映射为元素中 style 属性的字符串值。

第二个属性其实是一类，即事件处理程序（或者事件属性），比如 onclick。在元素上使用事件属性时（比如 onclick），属性的值是一段 JavaScript 代码。如果使用 getAttribute()访问事件属性，则返回的是字符串形式的源代码。而通过 DOM 对象的属性访问事件属性时返回的则是一个 JavaScript 函数（未指定该属性则返回 null）。这是因为 onclick 及其他事件属性是可以接受函数作为值的。

考虑到以上差异，开发者在进行 DOM 编程时通常会放弃使用 getAttribute()而只使用对象属性。getAttribute()主要用于取得自定义属性的值。

3. 设置属性

与 getAttribute()配套的方法是 setAttribute()，这个方法接收两个参数：要设置的属性名和属性的值。如果属性已经存在，则 setAttribute()会以指定的值替换原来的值；如果属性不存在，则 setAttribute()会以指定的值创建该属性。下面看一个例子：

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

setAttribute()适用于 HTML 属性，也适用于自定义属性。另外，使用 setAttribute()方法设置的属性名会规范为小写形式，因此"ID"会变成"id"。

因为元素属性也是 DOM 对象属性，所以直接给 DOM 对象的属性赋值也可以设置元素属性的值，如下所示：

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

注意，在 DOM 对象上添加自定义属性，如下面的例子所示，不会自动让它变成元素的属性：

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

这个例子添加了一个自定义属性 mycolor 并将其值设置为"red"。在多数浏览器中，这个属性不会自动变成元素属性。因此调用 getAttribute()取得 mycolor 的值会返回 null。

最后一个方法 removeAttribute()用于从元素中删除属性。这样不单单是清除属性的值，而是会把整个属性完全从元素中去掉，如下所示：

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

这个方法用得并不多，但在序列化 DOM 元素时可以通过它控制要包含的属性。

4. attributes 属性

Element 类型是唯一使用 attributes 属性的 DOM 节点类型。attributes 属性包含一个 NamedNodeMap 实例，是一个类似 NodeList 的“实时”集合。元素的每个属性都表示为一个 Attr 节点，并保存在这个 NamedNodeMap 对象中。NamedNodeMap 对象包含下列方法：

- getNamedItem(*name*)，返回 nodeName 属性等于 *name* 的节点；
- removeNamedItem(*name*)，删除 nodeName 属性等于 *name* 的节点；
- setNamedItem(*node*)，向列表中添加 *node* 节点，以其 nodeName 为索引；
- item(*pos*)，返回索引位置 *pos* 处的节点。

attributes 属性中的每个节点的 nodeName 是对应属性的名字，nodeValue 是属性的值。比如，要取得元素 id 属性的值，可以使用以下代码：

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

下面是使用中括号访问属性的简写形式：

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

同样，也可以用这种语法设置属性的值，即先取得属性节点，再将其 nodeValue 设置为新值，如下所示：

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

removeNamedItem()方法与元素上的 removeAttribute()方法类似，也是删除指定名字的属性。下面的例子展示了这两个方法唯一的不同之处，就是 removeNamedItem()返回表示被删除属性的 Attr 节点：

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

setNamedItem()方法很少使用，它接收一个属性节点，然后给元素添加一个新属性，如下所示：

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

一般来说，因为使用起来更简便，通常开发者更喜欢使用 getAttribute()、removeAttribute()和 setAttribute()方法，而不是刚刚介绍的 NamedNodeMap 对象的方法。

attributes 属性最有用的场景是需要迭代元素上所有属性的时候。这时候往往是要把 DOM 结构序列化为 XML 或 HTML 字符串。比如，以下代码能够迭代一个元素上的所有属性并以 attribute1="value1" attribute2="value2"的形式生成格式化字符串：

```
function outputAttributes(element) {
  let pairs = [];

  for (let i = 0, len = element.attributes.length; i < len; ++i) {
    const attribute = element.attributes[i];
    pairs.push(`${attribute.nodeName}="${attribute.nodeValue}"`);
  }

  return pairs.join(" ");
}
```

这个函数使用数组存储每个名/值对，迭代完所有属性后，再将这些名/值对用空格拼接在一起。（这个技术常用于序列化为长字符串。）这个函数中的 for 循环使用 attributes.length 属性迭代每个属性，将每个属性的名字和值输出为字符串。不同浏览器返回的 attributes 中的属性顺序也可能不一样。HTML 或 XML 代码中属性出现的顺序不一定与 attributes 中的顺序一致。

5. 创建元素

可以使用 document.createElement()方法创建新元素。这个方法接收一个参数，即要创建元素的标签名。在 HTML 文档中，标签名是不区分大小写的，而 XML 文档（包括 XHTML）是区分大小写的。要创建 元素，可以使用下面的代码：

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

使用 createElement()方法创建新元素的同时也会将其 ownerDocument 属性设置为 document。此时，可以再为其添加属性、添加更多子元素。比如：

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

在新元素上设置这些属性只会附加信息。因为这个元素还没有添加到文档树，所以不会影响浏览器显示。要把元素添加到文档树，可以使用 appendChild()、insertBefore()或 replaceChild()。比如，以下代码会把刚才创建的元素添加到文档的<body>元素中：

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

元素被添加到文档树之后，浏览器会立即将其渲染出来。之后再对这个元素所做的任何修改，都会立即在浏览器中反映出来。

6. 元素后代

元素可以拥有任意多个子元素和后代元素，因为元素本身也可以是其他元素的子元素。childNodes 属性包含元素所有的子节点，这些子节点可能是其他元素、文本节点、注释或处理指令。不同浏览器在识别这些节点时的表现有明显不同。比如下面的代码：

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

在解析以上代码时，<ul>元素会包含 7 个子元素，其中 3 个是<li>元素，还有 4 个 Text 节点（表示<li>元素周围的空格）。如果把元素之间的空格删掉，变成下面这样，则所有浏览器都会返回同样数量的子节点：

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

所有浏览器解析上面的代码后，<ul>元素都会包含 3 个子节点。考虑到这种情况，通常在执行某个操作之后需要先检测一下节点的 nodeType，如下所示：

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

以上代码会遍历某个元素的子节点，并且只在 nodeType 等于 1（即 Element 节点）时执行某个操作。

要取得某个元素的子节点和其他后代节点，可以使用元素的 getElementsByTagName()方法。在元素上调用这个方法与在文档上调用是一样的，只不过搜索范围限制在当前元素之内，即只会返回当前元素的后代。对于本节前面<ul>的例子，可以像下面这样取得其所有的<li>元素：

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

这里例子中的<ul>元素只有一级子节点，如果它包含更多层级，则所有层级中的<li>元素都会返回。

14.1.4　Text 类型

Text 节点由 Text 类型表示，包含按字面解释的纯文本，也可能包含转义后的 HTML 字符，但不含 HTML 代码。Text 类型的节点具有以下特征：

- nodeType 等于 3；
- nodeName 值为"#text"；
- nodeValue 值为节点中包含的文本；
- parentNode 值为 Element 对象；
- 不支持子节点。

Text 节点中包含的文本可以通过 nodeValue 属性访问，也可以通过 data 属性访问，这两个属性包含相同的值。修改 nodeValue 或 data 的值，也会在另一个属性反映出来。文本节点暴露了以下操作文本的方法：

- appendData(*text*)，向节点末尾添加文本 *text*；
- deleteData(*offset, count*)，从位置 *offset* 开始删除 *count* 个字符；
- insertData(*offset, text*)，在位置 *offset* 插入 *text*；
- replaceData(*offset, count, text*)，用 *text* 替换从位置 *offset* 到 *offset* + *count* 的文本；
- splitText(*offset*)，在位置 *offset* 将当前文本节点拆分为两个文本节点；
- substringData(*offset, count*)，提取从位置 *offset* 到 *offset* + *count* 的文本。

除了这些方法，还可以通过 length 属性获取文本节点中包含的字符数量。这个值等于 nodeValue.length 和 data.length。

默认情况下，包含文本内容的每个元素最多只能有一个文本节点。例如：

```
<!-- 没有内容，因此没有文本节点 -->
<div></div>

<!-- 有空格，因此有一个文本节点 -->
```

```
<div> </div>

<!-- 有内容，因此有一个文本节点 -->
<div>Hello World!</div>
```

示例中的第一个 元素中不包含内容，因此不会产生文本节点。只要开始标签和结束标签之间有内容，就会创建一个文本节点，因此第二个 元素会有一个文本节点的子节点，虽然它只包含空格。这个文本节点的 nodeValue 就是一个空格。第三个 元素也有一个文本节点的子节点，其 nodeValue 的值为"Hello World!"。下列代码可以用来访问这个文本节点：

```
let textNode = div.firstChild; // 或 div.childNodes[0]
```

取得文本节点的引用后，可以像这样来修改它：

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

只要节点在当前的文档树中，这样的修改就会马上反映出来。修改文本节点还有一点要注意，就是 HTML 或 XML 代码（取决于文档类型）会被转换成实体编码，即小于号、大于号或引号会被转义，如下所示：

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

这实际上是在将 HTML 字符串插入 DOM 文档前进行编码的有效方式。

1. 创建文本节点

document.createTextNode()可以用来创建新文本节点，它接收一个参数，即要插入节点的文本。跟设置已有文本节点的值一样，这些要插入的文本也会应用 HTML 或 XML 编码，如下面的例子所示：

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

创建新文本节点后，其 ownerDocument 属性会被设置为 document。但在把这个节点添加到文档树之前，我们不会在浏览器中看到它。以下代码创建了一个 元素并给它添加了一段文本消息：

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

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

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

这个例子首先创建了一个 元素并给它添加了值为"message"的 class 属性，然后又创建了一个文本节点并添加到该元素。最后一步是把这个元素添加到文档的主体上，这样元素及其包含的文本会出现在浏览器中。

一般来说一个元素只包含一个文本子节点。不过，也可以让元素包含多个文本子节点，如下面的例子所示：

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

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

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

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

在将一个文本节点作为另一个文本节点的同胞插入后，两个文本节点的文本之间不会包含空格。

2. 规范化文本节点

DOM 文档中的同胞文本节点可能导致困惑，因为一个文本节点足以表示一个文本字符串。同样，DOM 文档中也经常会出现两个相邻文本节点。为此，有一个方法可以合并相邻的文本节点。这个方法叫 normalize()，是在 Node 类型中定义的（因此所有类型的节点上都有这个方法）。在包含两个或多个相邻文本节点的父节点上调用 normalize()时，所有同胞文本节点会被合并为一个文本节点，这个文本节点的 nodeValue 就等于之前所有同胞节点 nodeValue 拼接在一起得到的字符串。来看下面的例子：

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

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

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

document.body.appendChild(element);

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

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

浏览器在解析文档时，永远不会创建同胞文本节点。同胞文本节点只会出现在 DOM 脚本生成的文档树中。

3. 拆分文本节点

Text 类型定义了一个与 normalize()相反的方法——splitText()。这个方法可以在指定的偏移位置拆分 nodeValue，将一个文本节点拆分成两个文本节点。拆分之后，原来的文本节点包含开头到偏移位置前的文本，新文本节点包含剩下的文本。这个方法返回新的文本节点，具有与原来的文本节点相同的 parentNode。来看下面的例子：

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

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

document.body.appendChild(element);

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

在这个例子中，包含"Hello world!"的文本节点被从位置 5 拆分成两个文本节点。位置 5 对应"Hello"和"world!"之间的空格，因此原始文本节点包含字符串"Hello"，而新文本节点包含文本" world!"（包含空格）。

拆分文本节点最常用于从文本节点中提取数据的 DOM 解析技术。

14.1.5 Comment 类型

DOM 中的注释通过 Comment 类型表示。Comment 类型的节点具有以下特征：

- nodeType 等于 8；
- nodeName 值为"#comment"；
- nodeValue 值为注释的内容；
- parentNode 值为 Document 或 Element 对象；
- 不支持子节点。

Comment 类型与 Text 类型继承同一个基类（CharacterData），因此拥有除 splitText()之外 Text 节点所有的字符串操作方法。与 Text 类型相似，注释的实际内容可以通过 nodeValue 或 data 属性获得。

注释节点可以作为父节点的子节点来访问。比如下面的 HTML 代码：

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

这里的注释是 元素的子节点，这意味着可以像下面这样访问它：

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

可以使用 document.createComment()方法创建注释节点，参数为注释文本，如下所示：

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

显然，注释节点很少通过 JavaScrpit 创建和访问，因为注释几乎不涉及算法逻辑。此外，浏览器不承认结束的</html>标签之后的注释。如果要访问注释节点，则必须确定它们是<html>元素的后代。

14.1.6 CDATASection 类型

CDATASection 类型表示 XML 中特有的 CDATA 区块。CDATASection 类型继承 Text 类型，因此拥有包括 splitText()在内的所有字符串操作方法。CDATASection 类型的节点具有以下特征：

- nodeType 等于 4；
- nodeName 值为"#cdata-section"；
- nodeValue 值为 CDATA 区块的内容；
- parentNode 值为 Document 或 Element 对象；
- 不支持子节点。

CDATA 区块只在 XML 文档中有效，因此某些浏览器比较陈旧的版本会错误地将 CDATA 区块解析为 Comment 或 Element。比如下面这行代码：

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

这里 的第一个子节点应该是 CDATASection 节点。但主流的四大浏览器没有一个将其识别为 CDATASection。即使在有效的 XHTML 文档中，这些浏览器也不能恰当地支持嵌入的 CDATA 区块。

在真正的 XML 文档中，可以使用 document.createCDataSection()并传入节点内容来创建 CDATA 区块。

14.1.7　DocumentType 类型

DocumentType 类型的节点包含文档的文档类型（doctype）信息，具有以下特征：

- nodeType 等于 10；
- nodeName 值为文档类型的名称；
- nodeValue 值为 null；
- parentNode 值为 Document 对象；
- 不支持子节点。

DocumentType 对象在 DOM Level 1 中不支持动态创建，只能在解析文档代码时创建。对于支持这个类型的浏览器，DocumentType 对象保存在 document.doctype 属性中。DOM Level 1 规定了 DocumentType 对象的 3 个属性：name、entities 和 notations。其中，name 是文档类型的名称，entities 是这个文档类型描述的实体的 NamedNodeMap，而 notations 是这个文档类型描述的表示法的 NamedNodeMap。因为浏览器中的文档通常是 HTML 或 XHTML 文档类型，所以 entities 和 notations 列表为空。（这个对象只包含行内声明的文档类型。）无论如何，只有 name 属性是有用的。这个属性包含文档类型的名称，即紧跟在<!DOCTYPE 后面的那串文本。比如下面的 HTML 4.01 严格文档类型：

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

对于这个文档类型，name 属性的值是"html"：

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

14.1.8　DocumentFragment 类型

在所有节点类型中，DocumentFragment 类型是唯一一个在标记中没有对应表示的类型。DOM 将文档片段定义为“轻量级”文档，能够包含和操作节点，却没有完整文档那样额外的消耗。DocumentFragment 节点具有以下特征：

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

不能直接把文档片段添加到文档。相反，文档片段的作用是充当其他要被添加到文档的节点的仓库。可以使用 document.createDocumentFragment()方法像下面这样创建文档片段：

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

文档片段从 Node 类型继承了所有文档类型具备的可以执行 DOM 操作的方法。如果文档中的一个节点被添加到一个文档片段，则该节点会从文档树中移除，不会再被浏览器渲染。添加到文档片段的新节点同样不属于文档树，不会被浏览器渲染。可以通过 appendChild()或 insertBefore()方法将文档片段的内容添加到文档。在把文档片段作为参数传给这些方法时，这个文档片段的所有子节点会被添加到文档中相应的位置。文档片段本身永远不会被添加到文档树。以下面的 HTML 为例：

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

假设想给这个<ul>元素添加 3 个列表项。如果分 3 次给这个元素添加列表项，浏览器就要重新渲染 3 次页面，以反映新添加的内容。为避免多次渲染，下面的代码示例使用文档片段创建了所有列表项，然后一次性将它们添加到了<ul>元素：

```
let fragment = document.createDocumentFragment();
let ul = document.getElementById("myList");

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

ul.appendChild(fragment);
```

这个例子先创建了一个文档片段，然后取得了<ul>元素的引用。接着通过 for 循环创建了 3 个列表项，每一项都包含表明自己身份的文本。为此先创建<li>元素，再创建文本节点并添加到该元素。然后通过 appendChild()把<li>元素添加到文档片段。循环结束后，通过把文档片段传给 appendChild()将所有列表项添加到了<ul>元素。此时，文档片段的子节点全部被转移到了<ul>元素。

14.1.9 Attr 类型

元素数据在 DOM 中通过 Attr 类型表示。Attr 类型构造函数和原型在所有浏览器中都可以直接访问。技术上讲，属性是存在于元素 attributes 属性中的节点。Attr 节点具有以下特征：

- nodeType 等于 2；
- nodeName 值为属性名；
- nodeValue 值为属性值；
- parentNode 值为 null；
- 在 HTML 中不支持子节点；
- 在 XML 中子节点可以是 Text 或 EntityReference。

属性节点尽管是节点，却不被认为是 DOM 文档树的一部分。Attr 节点很少直接被引用，通常开发者更喜欢使用 getAttribute()、removeAttribute()和 setAttribute()方法操作属性。

Attr 对象上有 3 个属性：name、value 和 specified。其中，name 包含属性名（与 nodeName 一样），value 包含属性值（与 nodeValue 一样），而 specified 是一个布尔值，表示属性使用的是默认值还是被指定的值。

可以使用 document.createAttribute()方法创建新的 Attr 节点，参数为属性名。比如，要给元素添加 align 属性，可以使用下列代码：

```
let attr = document.createAttribute("align");
attr.value = "left";
element.setAttributeNode(attr);

alert(element.attributes["align"].value);       // "left"
alert(element.getAttributeNode("align").value); // "left"
alert(element.getAttribute("align"));           // "left"
```

在这个例子中，首先创建了一个新属性。调用 createAttribute()并传入"align"为新属性设置