

# 第 18 章

## JavaScript 与 XML

### 本章内容

- 检测浏览器对 XML DOM 的支持
- 理解 JavaScript 中的 XPath
- 使用 XSLT 处理器

**曾**几何时，XML 一度成为存储和通过因特网传输结构化数据的标准。透过 XML 的发展，能够清晰地看到 Web 技术发展的轨迹。DOM 规范的制定，不仅是为了方便在 Web 浏览器中使用 XML，也是为了在桌面及服务器应用程序中处理 XML 数据。此前，由于浏览器无法解析 XML 数据，很多开发人员都要动手编写自己的 XML 解析器。而自从 DOM 出现后，所有浏览器都内置了对 XML 的原生支持（XML DOM），同时也提供了一系列相关的技术支持。

### 18.1 浏览器对 XML DOM 的支持

在正式的规范诞生以前，浏览器提供商实现的 XML 解决方案不仅对 XML 的支持程度参差不齐，而且对同一特性的支持也各不相同。DOM2 级是第一个提到动态创建 XML DOM 概念的规范。DOM3 级进一步增强了 XML DOM，新增了解析和序列化等特性。然而，当 DOM3 级规范的各项条款尘埃落定之后，大多数浏览器也都实现了各自不同的解决方案。

#### 18.1.1 DOM2 级核心

我们在第 12 章曾经提到过，DOM2 级在 `document.implementation` 中引入了 `createDocument()` 方法。IE9+、Firefox、Opera、Chrome 和 Safari 都支持这个方法。想一想，或许你还记得可以在支持 DOM2 级的浏览器中使用以下语法来创建一个空白的 XML 文档：

```
var xmldom = document.implementation.createDocument(namespaceUri, root, doctype);
```

在通过 JavaScript 处理 XML 时，通常只使用参数 `root`，因为这个参数指定的是 XML DOM 文档元素的标签名。而 `namespaceUri` 参数则很少用到，原因是在 JavaScript 中管理命名空间比较困难。最后，`doctype` 参数用得就更少了。

因此，要想创建一个新的、文档元素为 `<root>` 的 XML 文档，可以使用如下代码：

```
var xmldom = document.implementation.createDocument("", "root", null);
```

```
alert(xmldom.documentElement.tagName);    //"root"
```

```
var child = xmldom.createElement("child");
xmldom.documentElement.appendChild(child);
```

这个例子创建了一个 XML DOM 文档，没有默认的命名空间，也没有文档类型。但要注意的是，尽管不需要指定命名空间和文档类型，也必须传入相应的参数。具体来说，给命名空间 URI 传入一个空字符串，就意味着未指定命名空间，而给文档类型传入 `null`，就意味着不指定文档类型。变量 `xmldom` 中保存着一个 DOM2 级 Document 类型的实例，带有第 12 章讨论过的所有 DOM 方法和属性。我们例子显示了文档元素的标签名，然后又创建并给文档元素添加了一个新的子元素。

要检测浏览器是否支持 DOM2 级 XML，可以使用下面这行代码：

```
var hasXmlDom = document.implementation.hasFeature("XML", "2.0");
```

在实际开发中，很少需要从头开始创建一个 XML 文档，然后再使用 DOM 文档为其添加元素。更常见的情况往往是将某个 XML 文档解析为 DOM 结构，或者反之。由于 DOM2 级规范没有提供这种功能，因此就出现了一些事实标准。

## 18.1.2 DOMParser 类型

为了将 XML 解析为 DOM 文档，Firefox 引入了 DOMParser 类型；后来，IE9、Safari、Chrome 和 Opera 也支持了这个类型。在解析 XML 之前，首先必须创建一个 DOMParser 的实例，然后再调用 `parseFromString()` 方法。这个方法接受两个参数：要解析的 XML 字符串和内容类型（内容类型始终都应该是 `"text/xml"`）。返回的值是一个 Document 的实例。来看下面的例子。

```
var parser = new DOMParser();
var xmldom = parser.parseFromString("<root><child></root>", "text/xml");

alert(xmldom.documentElement.tagName);    //"root"
alert(xmldom.documentElement.firstChild.tagName);    //"child"

var anotherChild = xmldom.createElement("child");
xmldom.documentElement.appendChild(anotherChild);

var children = xmldom.getElementsByTagName("child");
alert(children.length);    //2
```

---

### *DOMParserExample01.htm*

在这个例子中，我们把一个简单的 XML 字符串解析成了一个 DOM 文档。解析得到的 DOM 结构以 `<root>` 作为其文档元素，该元素还有一个 `<child>` 子元素。此后，就可以使用 DOM 方法对返回的这个文档进行操作了。

DOMParser 只能解析格式良好的 XML，因而不能把 HTML 解析为 HTML 文档。在发生解析错误时，仍然会从 `parseFromString()` 中返回一个 Document 对象，但这个对象的文档元素是 `<parsererror>`，而文档元素的内容是对解析错误的描述。下面是一个例子。

```
<parsererror xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">XML
Parsing Error: no element found Location: file:///I:/My%20Writing/My%20Books/
Professional%20JavaScript/Second%20Edition/Examples/Ch15/DOMParserExample2.htm Line
Number 1, Column 7: <sourcetext> & lt;root & gt; -----^</sourcetext > </parsererror>
```

Firefox 和 Opera 都会返回这种格式的文档。Safari 和 Chrome 返回的文档也包含 `<parsererror>` 元素，但该元素会出现在发生解析错误的地方。IE9 会在调用 `parseFromString()` 的地方抛出一个解析错误。由于存在这些差别，因此确定是否发生解析错误的最佳方式就是，使用一个 `try-catch` 语句块，如果没

有错误, 则通过 `getElementsByTagName()` 来查找文档中是否存在 `<parsererror>` 元素, 如下面的例子所示。



```
var parser = new DOMParser(),
    xmldom,
    errors;
try {
    xmldom = parser.parseFromString("<root>", "text/xml");
    errors = xmldom.getElementsByTagName("parsererror");
    if (errors.length > 0) {
        throw new Error("Parsing error!");
    }
} catch (ex) {
    alert("Parsing error!");
}
```


*DOMParserExample02.htm*

这例子显示, 要解析的字符串中缺少了闭标签 `</root>`, 而这会导致解析错误。在 IE9+ 中, 此时会抛出错误。在 Firefox 和 Opera 中, 文档元素将是 `<parsererror>`, 而在 Safari 和 Chrome 中, `<parsererror>` 是 `<root>` 的第一个子元素。调用 `getElementsByTagName("parsererror")` 能够应对这两种情况。如果这个方法返回了元素, 就说明有错误发生, 继而通过一个警告框显示出来。当然, 你还可以更进一步, 从错误元素中提取出错误信息。

### 18.1.3 XMLSerializer 类型

在引入 `DOMParser` 的同时, Firefox 还引入了 `XMLSerializer` 类型, 提供了相反的功能: 将 DOM 文档序列化为 XML 字符串。后来, IE9+、Opera、Chrome 和 Safari 都支持了 `XMLSerializer`。

要序列化 DOM 文档, 首先必须创建 `XMLSerializer` 的实例, 然后将文档传入其 `serializeToString()` 方法, 如下面的例子所示。




```
var serializer = new XMLSerializer();
var xml = serializer.serializeToString(xmldom);
alert(xml);
```

*XMLSerializerExample01.htm*

但是, `serializeToString()` 方法返回的字符串并不适合打印, 因此看起来会显得乱糟糟的。

`XMLSerializer` 可以序列化任何有效的 DOM 对象, 不仅包括个别的节点, 也包括 HTML 文档。将 HTML 文档传入 `serializeToString()` 以后, HTML 文档将被视为 XML 文档, 因此得到的代码也将是格式良好的。



如果将非 DOM 对象传入 `serializeToString()`, 会导致错误发生。

### 18.1.4 IE8 及之前版本中的 XML

事实上, IE 是第一个原生支持 XML 的浏览器, 而这一支持是通过 ActiveX 对象实现的。为了便于桌面应用程序开发人员处理 XML, 微软创建了 MSXML 库; 但微软并没有针对 JavaScript 创建不同的对


象，而只是让 Web 开发人员能够通过浏览器访问相同的对象。

第 8 章曾经介绍过 ActiveXObject 类型，通过这个类型可以在 JavaScript 中创建 ActiveX 对象的实例。同样，要创建一个 XML 文档的实例，也要使用 ActiveXObject 构造函数并为其传入一个表示 XML 文档版本的字符串。有 6 种不同的 XML 文档版本可以供选择。

- ❑ Microsoft.XmlDom: 最初随同 IE 发布；不建议使用。
- ❑ MSXML2.DOMDocument: 为方便脚本处理而更新的版本，建议仅在特殊情况下作为后备版本使用。
- ❑ MSXML2.DOMDocument.3.0: 为了在 JavaScript 中使用，这是最低的建议版本。
- ❑ MSXML2.DOMDocument.4.0: 在通过脚本处理时并不可靠，使用这个版本可能导致安全警告。
- ❑ MSXML2.DOMDocument.5.0: 在通过脚本处理时并不可靠，使用这个版本同样可能导致安全警告。
- ❑ MSXML2.DOMDocument.6.0: 通过脚本能够可靠处理的最新版本。

在这 6 个版本中，微软只推荐使用 MSXML2.DOMDocument.6.0 或 MSXML2.DOMDocument.3.0；前者是最新最可靠的版本，而后者则是大多数 Windows 操作系统都支持的版本。可以作为后备版本的 MSXML2.DOMDocument，仅在针对 IE5.5 之前的浏览器开发时才有必要使用。

通过尝试创建每个版本的实例并观察是否有错误发生，可以确定哪个版本可用。例如：



```
function createDocument(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.DOMDocument.6.0", "MSXML2.DOMDocument.3.0",
                        "MSXML2.DOMDocument"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                //跳过
            }
        }

        return new ActiveXObject(arguments.callee.activeXString);
    }
}
```

---

#### *IEXmlDomExample01.htm*

这个函数中使用 for 循环迭代了每个可能的 ActiveX 版本。如果版本无效，则创建新 ActiveXObject 的调用就会抛出错误；此时，catch 语句会捕获错误，循环继续。如果没有发生错误，则可用的版本将被保存在这个函数的 activeXString 属性中。这样，就不必在每次调用这个函数时都重复检查可用版本了——直接创建并返回对象即可。

要解析 XML 字符串，首先必须创建一个 DOM 文档，然后调用 loadXML() 方法。新创建的 XML 文档完全是一个空文档，因而不能对其执行任何操作。为 loadXML() 方法传入的 XML 字符串经解析之后会被填充到 DOM 文档中。来看下面的例子。

```

var xmlDoc = createDocument();
xmlDoc.loadXML("<root><child/></root>");

alert(xmlDoc.documentElement.tagName);      //"root"
alert(xmlDoc.documentElement.firstChild.tagName);  //"child"

var anotherChild = xmlDoc.createElement("child");
xmlDoc.documentElement.appendChild(anotherChild);

var children = xmlDoc.getElementsByTagName("child");
alert(children.length);    //2

```

IEXmlDomExample01.htm

在新 DOM 文档中填充了 XML 内容之后，就可以像操作其他 DOM 文档一样操作它了（可以使用任何方法和属性）。

如果解析过程中出错，可以在 `parseError` 属性中找到错误消息。这个属性本身是一个包含多个属性的对象，每个属性都保存着有关解析错误的某一方面信息。

- ❑ `errorCode`：错误类型的数值编码；在没有发生错误时值为 0。
- ❑ `filePos`：文件中导致错误发生的位置。
- ❑ `line`：发生错误的行。
- ❑ `linepos`：发生错误的行中的字符。
- ❑ `reason`：对错误的文本解释。
- ❑ `srcText`：导致错误的代码。
- ❑ `url`：导致错误的文件的 URL（如果有这个文件的话）。

另外，`parseError` 的 `valueOf()` 方法返回 `errorCode` 的值，因此可以通过下列代码检测是否发生了解析错误。

```

if (xmlDoc.parseError != 0){
    alert("Parsing error occurred.");
}

```

错误类型的数值编码可能是正值，也可能是负值，因此我们只需检测它是不是等于 0。要取得有关解析错误的详细信息也很容易，而且可以将这些信息组合起来给出更有价值的解释。来看下面的例子。

```

if (xmlDoc.parseError != 0){
    alert("An error occurred:\nError Code: "
        + xmlDoc.parseError.errorCode + "\n"
        + "Line: " + xmlDoc.parseError.line + "\n"
        + "Line Pos: " + xmlDoc.parseError.linepos + "\n"
        + "Reason: " + xmlDoc.parseError.reason);
}

```

IEXmlDomExample02.htm

应该在调用 `loadXML()` 之后、查询 XML 文档之前，检查是否发生了解析错误。

## 1. 序列化 XML

IE 将序列化 XML 的能力内置在了 DOM 文档中。每个 DOM 节点都有一个 `xml` 属性，其中保存着表示该节点的 XML 字符串。例如：

```

alert(xmlDoc.xml);

```


文档中的每个节点都支持这个简单的序列化机制，无论是序列化整个文档还是某个子文档树，都非常方便。

## 2. 加载 XML 文件

IE 中的 XML 文档对象也可以加载来自服务器的文件。与 DOM3 级中的功能类似，要加载的 XML 文档必须与页面中运行的 JavaScript 代码来自同一台服务器。同样与 DOM3 级规范类似，加载文档的方式也可以分为同步和异步两种。要指定加载文档的方式，可以设置 `async` 属性，`true` 表示异步，`false` 表示同步（默认值为 `true`）。来看下面的例子。

```
var xmldom = createDocument();  
xmldom.async = false;
```

在确定了加载 XML 文档的方式后，调用 `load()` 可以启动下载过程。这个方法接受一个参数，即要加载的 XML 文件的 URL。在同步方式下，调用 `load()` 后可以立即检测解析错误并执行相关的 XML 处理，例如：



```
var xmldom = createDocument();  
xmldom.async = false;  
xmldom.load("example.xml");  
  
if (xmldom.parseError != 0){  
    //处理错误  
} else {  
  
    alert(xmldom.documentElement.tagName); // "root"  
    alert(xmldom.documentElement.firstChild.tagName); // "child"  
  
    var anotherChild = xmldom.createElement("child");  
    xmldom.documentElement.appendChild(anotherChild);  
  
    var children = xmldom.getElementsByTagName("child");  
    alert(children.length);    // 2  
  
    alert(xmldom.xml);  
  
}
```

---

*IEXmlDomExample03.htm*

由于是以同步方式处理 XML 文件，因此在解析完成之前，代码不会继续执行，这样的编程工作要简单一点。虽然同步方式比较方便，但如果下载时间太长，会导致程序反应很慢。因此，在加载 XML 文档时，通常都使用异步方式。

在异步加载 XML 文件的情况下，需要为 XML DOM 文档的 `onreadystatechange` 事件指定处理程序。有 4 个就绪状态（`ready state`）。

- 1: DOM 正在加载数据。
- 2: DOM 已经加载完数据。
- 3: DOM 已经可以使用，但某些部分可能还无法访问。
- 4: DOM 已经完全可以使用。

在实际开发中，要关注的只有一个就绪状态：4。这个状态表示 XML 文件已经全部加载完毕，而且已经全部解析为 DOM 文档。通过 XML 文档的 `readyState` 属性可以取得其就绪状态。以异步方式加载 XML 文件的典型模式如下。



```

var xmldom = createDocument();
xmldom.async = true;

xmldom.onreadystatechange = function(){
    if (xmldom.readyState == 4){
        if (xmldom.parseError != 0){
            alert("An error occurred:\nError Code: "
                + xmldom.parseError.errorCode + "\n"
                + "Line: " + xmldom.parseError.line + "\n"
                + "Line Pos: " + xmldom.parseError.linepos + "\n"
                + "Reason: " + xmldom.parseError.reason);
        } else {

            alert(xmldom.documentElement.tagName); //"root"
            alert(xmldom.documentElement.firstChild.tagName); //"child"

            var anotherChild = xmldom.createElement("child");
            xmldom.documentElement.appendChild(anotherChild);

            var children = xmldom.getElementsByTagName("child");
            alert(children.length); //2

            alert(xmldom.xml);

        }
    }
};

xmldom.load("example.xml");

```

[IEXmlDomExample04.htm](#)

要注意的是，为 onreadystatechange 事件指定处理程序的语句，必须放在调用 load() 方法的语句之前；这样，才能确保在就绪状态变化时调用该事件处理程序。另外，在事件处理程序内部，还必须注意要使用 XML 文档变量的名称 (xmldom)，不能使用 this 对象。原因是 ActiveX 控件为预防安全问题不允许使用 this 对象。当文档的就绪状态变化为 4 时，就可以放心地检测是否发生了解析错误，并在未发生错误的情况下处理 XML 了。



虽然可以通过 XML DOM 文档对象加载 XML 文件，但公认的还是使用 XMLHttpRequest 对象比较好。有关 XMLHttpRequest 对象及 Ajax 的相关内容，将在第 21 章讨论。

## 18.1.5 跨浏览器处理 XML

很少有开发人员能够有福气专门针对一款浏览器做开发。因此，编写能够跨浏览器处理 XML 的函数就成为了常见的需求。对解析 XML 而言，下面这个函数可以在所有四种主要浏览器中使用。



```

function parseXml(xml){
    var xmldom = null;

    if (typeof DOMParser != "undefined"){
        xmldom = (new DOMParser()).parseFromString(xml, "text/xml");
    }
}

```

```

    var errors = xmldom.getElementsByTagName("parsererror");
    if (errors.length){
        throw new Error("XML parsing error:" + errors[0].textContent);
    }

    } else if (typeof ActiveXObject != "undefined"){
        xmldom = createDocument();
        xmldom.loadXML(xml);
        if (xmldom.parseError != 0){
            throw new Error("XML parsing error: " + xmldom.parseError.reason);
        }
    }

    } else {
        throw new Error("No XML parser available.");
    }

    return xmldom;
}

```

---

### *CrossBrowserXmlExample01.htm*

---

这个 `parseXml()` 函数只接收一个参数，即可解析的 XML 字符串。在函数内部，我们通过能力检测来确定要使用的 XML 解析方式。`DOMParser` 类型是受支持最多的解决方案，因此首先检测该类型是否有效。如果是，则创建一个新的 `DOMParser` 对象，并将解析 XML 字符串的结果保存在变量 `xmldom` 中。由于 `DOMParser` 对象在发生解析错误时不抛出错误（除 IE9+ 之外），因此还要检测返回的文档以确定解析过程是否顺利。如果发现了解析错误，则根据错误消息抛出一个错误。

函数的最后一部分代码检测了对 ActiveX 的支持，并使用前面定义的 `createDocument()` 函数来创建适当版本的 XML 文档。与使用 `DOMParser` 时一样，这里也需要检测结果，以防有错误发生。如果确实有错误发生，同样也需要抛出一个包含错误原因的错误。

如果上述 XML 解析器都不可用，函数就会抛出一个错误，表示无法解析了。

在使用这个函数解析 XML 字符串时，应该将它放在 `try-catch` 语句当中，以防发生错误。来看下面的例子。

```

var xmldom = null;

try {
    xmldom = parseXml("<root><child/></root>");
} catch (ex){
    alert(ex.message);
}

```

//进一步处理

---

### *CrossBrowserXmlExample01.htm*

---

对序列化 XML 而言，也可以按照同样的方式编写一个能够在四大浏览器中运行的函数。例如：

```

function serializeXml(xmldom){

    if (typeof XMLSerializer != "undefined"){
        return (new XMLSerializer()).serializeToString(xmldom);

    } else if (typeof xmldom.xml != "undefined"){

```



```
        return xmldom.xml;  
    } else {  
        throw new Error("Could not serialize XML DOM.");  
    }  
}
```

*CrossBrowserXmlExample02.htm*

这个 `serializeXml()` 函数接收一个参数，即要序列化的 XML DOM 文档。与 `parseXml()` 函数一样，这个函数首先也是检测受到最广泛支持的特性，即 `XMLSerializer`。如果这个类型有效，则使用它来生成并返回文档的 XML 字符串。由于 ActiveX 方案比较简单，只使用了一个 `xml` 属性，因此这个函数直接检测了该属性。如果上述两方面尝试都失败了，函数就会抛出一个错误，说明序列化不能进行。一般来说，只要针对浏览器使用了适当的 XML DOM 对象，就不会出现无法序列化的情况，因而也就没有必要在 `try-catch` 语句中调用 `serializeXml()`。结果，就只需如下一行代码即可：

```
var xml = serializeXml(xmldom);
```

只不过由于序列化过程的差异，相同的 DOM 对象在不同的浏览器下，有可能会得到不同的 XML 字符串。

## 18.2 浏览器对 XPath 的支持

XPath 是设计用来在 DOM 文档中查找节点的一种手段，因而对 XML 处理也很重要。但是，DOM3 级以前的标准并没有就 XPath 的 API 作出规定；XPath 是在 DOM3 级 XPath 模块中首次跻身推荐标准行列的。很多浏览器都实现了这个推荐标准，但 IE 则以自己的方式实现了 XPath。

### 18.2.1 DOM3 级 XPath

DOM3 级 XPath 规范定义了了在 DOM 中对 XPath 表达式求值的接口。要确定某浏览器是否支持 DOM3 级 XPath，可以使用以下 JavaScript 代码：

```
var supportsXPath = document.implementation.hasFeature("XPath", "3.0");
```

在 DOM3 级 XPath 规范定义的类型中，最重要的两个类型是 `XPathEvaluator` 和 `XPathResult`。`XPathEvaluator` 用于在特定的上下文中对 XPath 表达式求值。这个类型有下列 3 个方法。

- ❑ `createExpression(expression, nsresolver)`：将 XPath 表达式及相应的命名空间信息转换成一个 `XPathExpression`，这是查询的编译版。在多次使用同一个查询时很有用。
- ❑ `createNSResolver(node)`：根据 `node` 的命名空间信息创建一个新的 `XPathNSResolver` 对象。在基于使用命名空间的 XML 文档求值时，需要使用 `XPathNSResolver` 对象。
- ❑ `evaluate(expression, context, nsresolver, type, result)`：在给定的上下文中，基于特定的命名空间信息来对 XPath 表达式求值。剩下的参数指定如何返回结果。

在 Firefox、Safari、Chrome 和 Opera 中，`Document` 类型通常都是与 `XPathEvaluator` 接口一起实现的。换句话说，在这些浏览器中，既可以创建 `XPathEvaluator` 的新实例，也可以使用 `Document` 实例中的方法（XML 或 HTML 文档均是如此）。

在上面这三个方法中，`evaluate()` 是最常用的。这个方法接收 5 个参数：XPath 表达式、上下文

节点、命名空间求解器、返回结果的类型和保存结果的 XPathResult 对象（通常是 null，因为结果也会以函数值的形式返回）。其中，第三个参数（命名空间求解器）只在 XML 代码中使用了 XML 命名空间时有必要指定；如果 XML 代码中没有使用命名空间，则这个参数应该指定为 null。第四个参数（返回结果的类型）的取值范围是下列常量之一。

- XPathResult.ANY\_TYPE: 返回与 XPath 表达式匹配的数据类型。
- XPathResult.NUMBER\_TYPE: 返回数值。
- XPathResult.STRING\_TYPE: 返回字符串值。
- XPathResult.BOOLEAN\_TYPE: 返回布尔值。
- XPathResult.UNORDERED\_NODE\_ITERATOR\_TYPE: 返回匹配的节点集合，但集合中节点的次序不一定与它们在文档中的次序一致。
- XPathResult.ORDERED\_NODE\_ITERATOR\_TYPE: 返回匹配的节点集合，集合中节点的次序与它们在文档中的次序一致。这是最常用的结果类型。
- XPathResult.UNORDERED\_NODE\_SNAPSHOT\_TYPE: 返回节点集合的快照，由于是在文档外部捕获节点，因此对文档的后续操作不会影响到这个节点集合。集合中节点的次序不一定与它们在文档中的次序一致。
- XPathResult.ORDERED\_NODE\_SNAPSHOT\_TYPE: 返回节点集合的快照，由于是在文档外部捕获节点，因此对文档的后续操作不会影响到这个节点集合。集合中节点的次序与它们在文档中的次序一致。
- XPathResult.ANY\_UNORDERED\_NODE\_TYPE: 返回匹配的节点集合，但集合中节点的次序不一定与它们在文档中的次序一致。
- XPathResult.FIRST\_ORDERED\_NODE\_TYPE: 返回只包含一个节点的节点集合，包含的这个节点就是文档中第一个匹配的节点。

指定的结果类型决定了如何取得结果的值。下面来看一个典型的例子。

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);

if (result !== null) {
    var node = result.iterateNext();
    while(node) {
        alert(node.tagName);
        node = node.iterateNext();
    }
}
```

*DomXPathExample01.htm*

这个例子中为返回结果指定的是 XPathResult.ORDERED\_NODE\_ITERATOR\_TYPE，也是最常用的结果类型。如果没有节点匹配 XPath 表达式，evaluate() 返回 null；否则，它会返回一个 XPathResult 对象。这个 XPathResult 对象带有的属性和方法，可以用来取得特定类型的结果。如果节点是一个节点迭代器，无论是次序一致还是次序不一致的，都必须使用 iterateNext() 方法从节点中取得匹配的节点。在没有更多的匹配节点时，iterateNext() 返回 null。

如果指定的是快照结果类型（不管是次序一致还是次序不一致的），就必须使用 snapshotItem() 方法和 snapshotLength 属性，例如：

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);
if (result !== null) {
    for (var i=0, len=result.snapshotLength; i < len; i++) {
        alert(result.snapshotItem(i).tagName);
    }
}
```


---

### *DomXPathExample02.htm*

这里, snapshotLength 返回的是快照中节点的数量, 而 snapshotItem() 则返回快照中给定位置的节点 (与 NodeList 中的 length 和 item() 相似)。

#### 1. 单节点结果

指定常量 XPathResult.FIRST\_ORDERED\_NODE\_TYPE 会返回第一个匹配的节点, 可以通过结果的 singleNodeValue 属性来访问该节点。例如:



```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.FIRST_ORDERED_NODE_TYPE, null);

if (result !== null) {
    alert(result.singleNodeValue.tagName);
}
```

---

### *DomXPathExample03.htm*

与前面的查询一样, 在没有匹配节点的情况下, evaluate() 返回 null。如果有节点返回, 那么就可以通过 singleNodeValue 属性来访问它。

#### 2. 简单类型结果

通过 XPath 也可以取得简单的非节点数据类型, 这时候就要使用 XPathResult 的布尔值、数值和字符串类型了。这几个结果类型分别会通过 booleanValue、numberValue 和 stringValue 属性返回一个值。对于布尔值类型, 如果至少有一个节点与 XPath 表达式匹配, 则求值结果返回 true, 否则返回 false。来看下面的例子。

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.BOOLEAN_TYPE, null);
alert(result.booleanValue);
```

---

### *DomXPathExample04.htm*

在这个例子中, 如果有节点匹配 "employee/name", 则 booleanValue 属性的值就是 true。

对于数值类型, 必须在 XPath 表达式参数的位置上指定一个能够返回数值的 XPath 函数, 例如计算与给定模式匹配的所有节点数量的 count()。来看下面的例子。

```
var result = xmldom.evaluate("count(employee/name)", xmldom.documentElement,
                             null, XPathResult.NUMBER_TYPE, null);
alert(result.numberValue);
```

---

### *DomXPathExample05.htm*

以上代码会输出与 "employee/name" 匹配的节点数量 (即 2)。如果使用这个方法的时候没有指定与前例类似的 XPath 函数, 那么 numberValue 的值将等于 NaN。



对于字符串类型，`evaluate()`方法会查找与 XPath 表达式匹配的第一个节点，然后返回其第一个子节点的值（实际上是假设第一个子节点为文本节点）。如果没有匹配的节点，结果就是一个空字符串。来看一个例子。

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.STRING_TYPE, null);
alert(result.stringValue);
```

*DomXPathExample06.htm*

这个例子的输出结果中包含着与“`element/name`”匹配的元素的第一个子节点中包含的字符串。

### 3. 默认类型结果

所有 XPath 表达式都会自动映射到特定的结果类型。像前面那样设置特定的结果类型，可以限制表达式的输出。而使用 `XPathResult.ANY_TYPE` 常量可以自动确定返回结果的类型。一般来说，自动选择的结果类型可能是布尔值、数值、字符串值或一个次序不一致的节点迭代器。要确定返回的是什么结果类型，可以检测结果的 `resultType` 属性，如下面的例子所示。

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                             XPathResult.ANY_TYPE, null);

if (result !== null) {
    switch(result.resultType) {
        case XPathResult.STRING_TYPE:
            //处理字符串类型
            break;

        case XPathResult.NUMBER_TYPE:
            //处理数值类型
            break;

        case XPathResult.BOOLEAN_TYPE:
            //处理布尔值类型
            break;

        case XPathResult.UNORDERED_NODE_ITERATOR_TYPE:
            //处理次序不一致的节点迭代器类型
            break;

        default:
            //处理其他可能的结果类型
    }
}
```

显然，`XPathResult.ANY_TYPE` 可以让我们更灵活地使用 XPath，但是却要求有更多的处理代码来处理返回的结果。


### 4. 命名空间支持

对于利用了命名空间的 XML 文档，`XPathEvaluator` 必须知道命名空间信息，然后才能正确地进行求值。处理命名空间的方法也不止一种。我们以下的 XML 代码为例。

```
<?xml version="1.0" ?>
<wrox:books xmlns:wrox="http://www.wrox.com/">
  <wrox:book>
    <wrox:title>Professional JavaScript for Web Developers</wrox:title>
    <wrox:author>Nicholas C. Zakas</wrox:author>
  </wrox:book>
  <wrox:book>
    <wrox:title>Professional Ajax</wrox:title>
    <wrox:author>Nicholas C. Zakas</wrox:author>
    <wrox:author>Jeremy McPeak</wrox:author>
    <wrox:author>Joe Fawcett</wrox:author>
  </wrox:book>
</wrox:books>
```

在这个 XML 文档中,所有元素定义都来自 `http://www.wrox.com/` 命名空间,以前缀 `wrox` 标识。如果要对这个文档使用 XPath,就需要定义要使用的命名空间;否则求值将会失败。

处理命名空间的第一种方法是通过 `createNSResolver()` 来创建 `XPathNSResolver` 对象。这个方法接受一个参数,即文档中包含命名空间定义的节点。对于前面的 XML 文档来说,这个节点就是文档元素 `<wrox:books>`,它的 `xmlns` 特性定义了命名空间。可以把这个节点传递给 `createNSResolver()`,然后可以像下面这样在 `evaluate()` 中使用返回的结果。



```
var nsresolver = xmldom.createNSResolver(xmldom.documentElement);

var result = xmldom.evaluate("wrox:book/wrox:author",
                             xmldom.documentElement, nsresolver,
                             XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

alert(result.snapshotLength);
```

---

*DomXPathExample07.htm*

在将 `nsresolver` 对象传入到 `evaluate()` 之后,就可以确保它能够理解 XPath 表达式中使用的 `wrox` 前缀。读者可以试一试使用相同的表达式,如果不使用 `XPathNSResolver` 的话,就会导致错误。

处理命名空间的第二种方法就是定义一个函数,让它接收一个命名空间前缀,返回关联的 URI,例如:

```
var nsresolver = function(prefix){
  switch(prefix){
    case "wrox": return "http://www.wrox.com/";
    //其他前缀
  }
};

var result = xmldom.evaluate("count(wrox:book/wrox:author)",
                             xmldom.documentElement, nsresolver, XPathResult.NUMBER_TYPE, null);

alert(result.numberValue);
```

---

*DomXPathExample08.htm*

在不确定文档中的哪个节点包含命名空间定义的情况下,这个命名空间解析函数就可以派上用场了。只要你知道前缀和 URI,就可以定义一个返回该信息的函数,然后将它作为第三个参数传递给 `evaluate()` 即可。

## 18.2.2 IE 中的 XPath

IE 对 XPath 的支持是内置在基于 ActiveX 的 XML DOM 文档对象中的，没有使用 DOMParser 返回的 DOM 对象。因此，为了在 IE9 及之前的版本中使用 XPath，必须使用基于 ActiveX 的实现。这个接口在每个节点上额外定义了两个的方法：`selectSingleNode()` 和 `selectNodes()`。其中，`selectSingleNode()` 方法接受一个 XPath 模式，在找到匹配节点时返回第一个匹配的节点，如果没有找到匹配的节点就返回 `null`。例如：

```
var element = xmldom.documentElement.selectSingleNode("employee/name");

if (element !== null){
    alert(element.xml);
}
```

---

*IEXPathExample01.htm*

这里，会返回匹配“employee/name”的第一个节点。上下文节点是 `xmldom.documentElement`，因此就调用了该节点上的 `selectSingleNode()`。由于调用这个方法可能会返回 `null` 值，因而有必要在使用返回的节点之前，先检查确定它不是 `null`。

另一个方法 `selectNodes()` 也接收一个 XPath 模式作为参数，但它返回与模式匹配的所有节点的 `NodeList`（如果没有匹配的节点，则返回一个包含零项的 `NodeList`）。来看下面的例子。

```
var elements = xmldom.documentElement.selectNodes("employee/name");
alert(elements.length);
```

---

*IEXPathExample02.htm*

对这个例子而言，匹配“employee/name”的所有元素都会通过 `NodeList` 返回。由于不可能返回 `null` 值，因此可以放心地使用返回的结果。但要记住，既然结果是 `NodeList`，而其包含的元素可能会动态变化，所以每次访问它都有可能得到不同的结果。

IE 对 XPath 的支持非常简单。除了能够取得一个节点或一个 `NodeList` 外，不可能取得其他结果类型。

### IE 对命名空间的支持

要在 IE 中处理包含命名空间的 XPath 表达式，你必须知道自己使用的命名空间，并按照下列格式创建一个字符串：

```
"xmlns:prefix1='uri1' xmlns:prefix2='uri2' xmlns:prefix3='uri3'"
```

然后，必须将这个字符串传入到 XML DOM 文档对象的特殊方法 `setProperty()` 中，这个方法接收两个参数：要设置的属性名和属性值。在这里，属性名应该是“`SelectionNamespaces`”，属性值就是按照前面格式创建的字符串。下面来看一个在 DOM XPath 命名空间中对 XML 文档求值的例子。

```
xmldom.setProperty("SelectionNamespaces", "xmlns:wrox='http://www.wrox.com/'");

var result = xmldom.documentElement.selectNodes("wrox:book/wrox:author");
alert(result.length);
```

---

*IEXPathExample03.htm*

对于这个 DOM XPath 的例子来说, 如果不提供命名空间解析信息, 就会在对表达式求值时导致一个错误。

### 18.2.3 跨浏览器使用 XPath

鉴于 IE 对 XPath 功能的支持有限, 因此跨浏览器 XPath 只能保证达到 IE 支持的功能。换句话说, 也就是要在其他使用 DOM3 级 XPath 对象的浏览器中, 重新创建 `selectSingleNode()` 和 `selectNodes()` 方法。第一个函数是 `selectSingleNode()`, 它接收三个参数: 上下文节点、XPath 表达式和可选的命名空间对象。命名空间对象应该是下面这种字面量的形式。

```
{
  prefix1: "uri1",
  prefix2: "uri2",
  prefix3: "uri3"
}
```

以这种方式提供的命名空间信息, 可以方便地转换为针对特定浏览器的命名空间解析格式。下面给出了 `selectSingleNode()` 函数的完整代码。

```
function selectSingleNode(context, expression, namespaces){
  var doc = (context.nodeType != 9 ? context.ownerDocument : context);


  if (typeof doc.evaluate != "undefined"){
    var nsresolver = null;
    if (namespaces instanceof Object){
      nsresolver = function(prefix){
        return namespaces[prefix];
      };
    }

    var result = doc.evaluate(expression, context, nsresolver,
                              XPathResult.FIRST_ORDERED_NODE_TYPE, null);
    return (result != null ? result.singleNodeValue : null);
  } else if (typeof context.selectSingleNode != "undefined"){
    //创建命名空间字符串
    if (namespaces instanceof Object){
      var ns = "";
      for (var prefix in namespaces){
        if (namespaces.hasOwnProperty(prefix)){
          ns += "xmlns:" + prefix + "=" + namespaces[prefix] + " ";
        }
      }
      doc.setProperty("SelectionNamespaces", ns);
    }
    return context.selectSingleNode(expression);
  } else {
    throw new Error("No XPath engine found.");
  }
}
```

这个函数首先要确定 XML 文档，以便基于该文档对表达式求值。由于上下文节点可能是文档，所以必须要检测 `nodeType` 属性。此后，变量 `doc` 中就会保存对 XML 文档的引用。然后，可以检测文档中是否存在 `evaluate()` 方法，即是否支持 DOM3 级 XPath。如果支持，接下来就是检测传入的 `namespaces` 对象。在这里使用 `instanceof` 操作符而不是 `typeof`，是因为后者对 `null` 也返回 "object"。然后将 `nsresolver` 变量初始化为 `null`，如果提供了命名空间信息的话，就将其改为一个函数。这个函数是一个闭包，它使用传入的 `namespaces` 对象来返回命名空间的 URI。此后，调用 `evaluate()` 方法，并对其结果进行检测，在确定是节点之后再返回该结果。

在这个函数针对 IE 的分支中，需要检查 `context` 节点中是否存在 `selectSingleNode()` 方法。与 DOM 分支一样，这里的第一步是有选择地构建命名空间信息。如果传入了 `namespaces` 对象，则迭代其属性并以适当格式创建一个字符串。注意，这里使用了 `hasOwnProperty()` 方法来确保对 `Object.prototype` 的任何修改都不会影响到当前函数。最后，调用原生的 `selectSingleNode()` 方法并返回结果。

如果前面两种方法都没有得到支持，这个函数就会抛出一个错误，表示找不到 XPath 处理引擎。下面是使用 `selectSingleNode()` 函数的示例。



```
var result = selectSingleNode(xmlDom.documentElement, "wrox:book/wrox:author",
                               { wrox: "http://www.wrox.com/" });
alert(serializeXml(result));
```

---

*CrossBrowserXPathExample01.htm*

---

类似地，也可以创建一个跨浏览器的 `selectNodes()` 函数。这个函数接收与 `selectSingleNode()` 相同的三个参数，而且大部分逻辑都相似。为了便于看清楚，我们用加粗字体突出了这两个函数的差别所在。

```
function selectNodes(context, expression, namespaces){
    var doc = (context.nodeType != 9 ? context.ownerDocument : context);

    if (typeof doc.evaluate != "undefined"){
        var nsresolver = null;
        if (namespaces instanceof Object){
            nsresolver = function(prefix){
                return namespaces[prefix];
            };
        }

        var result = doc.evaluate(expression, context, nsresolver,
                                   XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);
        var nodes = new Array();

        if (result != null){
            for (var i=0, len=result.snapshotLength; i < len; i++){
                nodes.push(result.snapshotItem(i));
            }
        }

        return nodes;
    } else if (typeof context.selectNodes != "undefined"){
        //创建命名空间字符串
        if (namespaces instanceof Object){
            var ns = "";
```



```

        for (var prefix in namespaces){
            if (namespaces.hasOwnProperty(prefix)){
                ns += "xmlns:" + prefix + "=" + namespaces[prefix] + " ";
            }
        }
        doc.setProperty("SelectionNamespaces", ns);
    }
    var result = context.selectNodes(expression);
    var nodes = new Array();


    for (var i=0,len=result.length; i < len; i++){
        nodes.push(result[i]);
    }

    return nodes;
} else {
    throw new Error("No XPath engine found.");
}
}

```

*CrossBrowserXPathExample02.htm*

很明显，其中有很多逻辑都与 `selectSingleNode()` 方法相同。在函数针对 DOM 的部分，使用了有序快照结果类型，然后将结果保存在了一个数组中。为了与 IE 的实现看齐，这个函数应该在没找到匹配项的情况下也返回一个数组，因而最终都要返回数组 `nodes`。在函数针对 IE 的分支中，调用了 `selectNodes()` 方法并将结果复制到了一个数组中。因为 IE 返回的是一个 `NodeList`，所以最好将节点都复制到一个数组中，这样就可以确保在不同浏览器下，函数都能返回相同的数据类型。使用这个函数的示例如下：



```

var result = selectNodes(xmlDom.documentElement, "wrox:book/wrox:author",
                        { wrox: "http://www.wrox.com/" });
alert(result.length);

```

*CrossBrowserXPathExample02.htm*

为了求得最佳的浏览器兼容性，我们建议在 JavaScript 中使用 XPath 时，只考虑使用这两个方法。

## 18.3 浏览器对 XSLT 的支持

XSLT 是与 XML 相关的一种技术，它利用 XPath 将文档从一种表现形式转换成另一种表现形式。与 XML 和 XPath 不同，XSLT 没有正式的 API，在正式的 DOM 规范中也没有它的位置。结果，只能依靠浏览器开发商以自己的方式来实现它。IE 是第一个支持通过 JavaScript 处理 XSLT 的浏览器。


### 18.3.1 IE 中的 XSLT

与 IE 对其他 XML 功能的支持一样，它对 XSLT 的支持也是通过 ActiveX 对象实现的。从 MSXML 3.0（即 IE6.0）时代起，IE 就支持通过 JavaScript 实现完整的 XSLT 1.0 操作。IE9 中通过 `DOMParser` 创建的 DOM 文档不能使用 XSLT。

#### 1. 简单的 XSLT 转换

使用 XSLT 样式表转换 XML 文档的最简单方式，就是将它们分别加到一个 DOM 文档中，然后再

使用 `transformNode()` 方法。这个方法存在于文档的所有节点中，它接受一个参数，即包含 XSLT 样式表的文档。调用 `transformNode()` 方法会返回一个包含转换信息的字符串。来看一个例子。



```
//加载 XML 和 XSLT (仅限于 IE)
xmldom.load("employees.xml");
xsltDom.load("employees.xslt");

//转换
var result = xmldom.transformNode(xsltDom);
```

---

*IEXsltExample01.htm*

这个例子加载了一个 XML 的 DOM 文档和一个 XSLT 样式表的 DOM 文档。然后，在 XML 文档节点上调用了 `transformNode()` 方法，并传入 XSLT。变量 `result` 中最后就会保存一个转换之后得到的字符串。需要注意的是，由于是在文档节点级别上调用的 `transformNode()`，因此转换是从文档节点开始的。实际上，XSLT 转换可以在文档的任何级别上进行，只要在想要开始转换的节点上调用 `transformNode()` 方法即可。下面我们来看一个例子。

```
result = xmldom.documentElement.transformNode(xsltDom);
result = xmldom.documentElement.childNodes[1].transformNode(xsltDom);
result = xmldom.getElementsByTagName("name")[0].transformNode(xsltDom);
result = xmldom.documentElement.firstChild.lastChild.transformNode(xsltDom);
```

如果不是在文档元素上调用 `transformNode()`，那么转换就会从调用节点上面开始。不过，XSLT 样式表则始终都可以针对调用节点所在的整个 XML 文档，而无需更换。

## 2. 复杂的 XSLT 转换

虽然 `transformNode()` 方法提供了基本的 XSLT 转换能力，但还有使用这种语言的更复杂的方式。为此，必须要使用 XSL 模板和 XSL 处理器。第一步是要把 XSLT 样式表加载到一个线程安全的 XML 文档中。而这可以通过使用 ActiveX 对象 `MSXML2.FreeThreadedDOMDocument` 来做到。这个 ActiveX 对象与 IE 中常规的 DOM 支持相同的接口。此外，创建这个对象时应该尽可能使用最新的版本。例如：

```
function createThreadSafeDocument(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.FreeThreadedDOMDocument.6.0",
            "MSXML2.FreeThreadedDOMDocument.3.0",
            "MSXML2.FreeThreadedDOMDocument"],
            i, len;

        for (i=0, len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                //跳过
            }
        }

        return new ActiveXObject(arguments.callee.activeXString);
    }
}
```


---

*IEXsltExample02.htm*

除了签名不同之外，线程安全的 XML DOM 文档与常规 XML DOM 文档的使用仍然是一样的，如下所示：

```
var xsltdom = createThreadSafeDocument();
xsltdom.async = false;
xsltdom.load("employees.xslt");
```

在创建并加载了自由线程的 DOM 文档之后，必须将它指定给一个 XSL 模板，这也是一个 ActiveX 对象。而这个模板是用来创建 XSL 处理器对象的，后者则是用来转换 XML 文档的。同样，也需要使用最新版本来创建这个对象，如下所示：



```
function createXSLTemplate(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.XSLTemplate.6.0",
                        "MSXML2.XSLTemplate.3.0",
                        "MSXML2.XSLTemplate"],
            i, len;
        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                //跳过
            }
        }
    }
    return new ActiveXObject(arguments.callee.activeXString);
}
```

*IEXsltExample02.htm*

使用这个 createXSLTemplate() 函数可以创建这个对象最新版本的实例，用法如下：


```
var template = createXSLTemplate();
template.stylesheet = xsltdom;

var processor = template.createProcessor();
processor.input = xmldom;
processor.transform();

var result = processor.output;
```

*IEXsltExample02.htm*

在创建了 XSL 处理器之后，必须将要转换的节点指定给 input 属性。这个值可以是一个文档，也可以是文档中的任何节点。然后，调用 transform() 方法即可执行转换并将结果作为字符串保存在 output 属性中。这些代码实现了与 transformNode() 相同的功能。



XSL 模板对象的 3.0 和 6.0 版本存在显著的差别。在 3.0 版本中，必须给 input 属性指定一个完整的文档；如果指定的是节点，就会导致错误。而在 6.0 版本中，则可以为 input 属性指定文档中的任何节点。

使用 XSL 处理器可以对转换进行更多的控制,同时也支持更高级的 XSLT 特性。例如, XSLT 样式表可以接受传入的参数,并将其用作局部变量。以下面的样式表为例:



```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:param name="message"/>

  <xsl:template match="/">
    <ul>
      <xsl:apply-templates select="*" />
    </ul>
    <p>Message: <xsl:value-of select="$message" /></p>
  </xsl:template>


  <xsl:template match="employee">
    <li><xsl:value-of select="name" />,
      <em><xsl:value-of select="@title" /></em></li>
  </xsl:template>

</xsl:stylesheet>
```

---

*employees.xslt*

这个样式表定义了一个名为 message 的参数,然后将该参数输出到转换结果中。要设置 message 的值,可以在调用 transform() 之前使用 addParameter() 方法。addParameter() 方法接收两个参数:要设置的参数名称(与在<xsl:param>的 name 特性中指定的一样)和要指定的值(多数情况下是字符串,但也可以是数值或布尔值)。下面就是这样一个例子。



```
processor.input = xmldom.documentElement;
processor.addParameter("message", "Hello World!");
processor.transform();
```

---

*IEXsltExample03.htm*

通过设置参数的值,这个值就可以在输出中反映出来。

XSL 处理器的另一个高级特性,就是能够设置一种操作模式。在 XSLT 中,可以使用 mode 特性为模板定义一种模式。在定义了模式后,如果没有将<xsl:apply-templates>与匹配的 mode 特性一起使用,就不会运行该模板。下面来看一个例子。

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"/>

  <xsl:param name="message"/>

  <xsl:template match="/">
    <ul>
      <xsl:apply-templates select="*" />
    </ul>
    <p>Message: <xsl:value-of select="$message" /></p>
  </xsl:template>
```

```

<xsl:template match="employee">
  <li><xsl:value-of select="name" />,
    <em><xsl:value-of select="@title" /></em></li>
</xsl:template>


<xsl:template match="employee" mode="title-first">
  <li><em><xsl:value-of select="@title" /></em>,
    <xsl:value-of select="name" /></li>
</xsl:template>

</xsl:stylesheet>

```

*employees3.xslt*

这个样式表定义了一个模板，并将其 `mode` 特性设置为 "title-first"（即“先显示职位”）。在这个模板中，首先会输出员工的职位，其次才输出员工的名字。为了使用这个模板，必须也要将 `<xsl:apply-templates>` 元素的模式设置为 "title-first"。在使用这个样式表时，默认情况下其输出结果与前面一样，先显示员工的名字，再显示员工的职位。但是，如果在使用这个样式表时，使用 JavaScript 将模式设置为 "title-first"，那么结果就会先输出员工的职位。在 JavaScript 中使用 `setStartMode()` 方法设置模式的例子如下。



```

processor.input = xmldom;
processor.addParameter("message", "Hello World!");
processor.setStartMode("title-first");
processor.transform();

```


*IEXsltExample05.htm*

`setStartMode()` 方法只接受一个参数，即要为处理器设置的模式。与 `addParameter()` 一样，设置模式也必须在调用 `transform()` 之前进行。

如果你打算使用同一个样式表进行多次转换，可以在每次转换之后重置处理器。调用 `reset()` 方法后，就会清除原先的输入和输出属性、启动模式及其他指定的参数。调用 `reset()` 方法的例子如下：

```
processor.reset(); //准备下一次转换
```

因为处理器已经编译了 XSLT 样式表，所以与使用 `transformNode()` 相比，这样进行重复转换的速度会更快一些。



MSXML 只支持 XSLT 1.0。由于微软的战略重点转移到了 .NET Framework，因而 MSXML 的开发被停止了。我们希望在不久的将来，能够通过 JavaScript 访问 XML 和 XSLT .NET 对象。

## 18.3.2 XSLTProcessor 类型

Mozilla 通过在 Firefox 中创建新的类型，实现了 JavaScript 对 XSLT 的支持。开发人员可以通过 `XSLTProcessor` 类型使用 XSLT 转换 XML 文档，其方式与在 IE 中使用 XSL 处理器类似。因为这个类型是率先出现的，所以 Chrome、Safari 和 Opera 都借鉴了相同的实现，最终使 `XSLTProcessor` 成为了通过 JavaScript 进行 XSLT 转换的事实标准。

与 IE 的实现类似，第一步也是加载两个 DOM 文档，一个基于 XML，另一个基于 XSLT。然后，创建一个新 XSLTProcessor 对象，并使用 importStylesheet() 方法为其指定一个 XSLT，如下面的例子所示。


```
var processor = new XSLTProcessor()  
processor.importStylesheet(xsltdom);
```

---

*XsltProcessorExample01.htm*

最后一步就是执行转换。这一步有两种不同的方式，如果想返回一个完整的 DOM 文档，可以调用 transformToDocument()。而通过调用 transformToFragment() 则可以得到一个文档片段对象。一般来说，使用 transformToFragment() 的唯一理由，就是你还想把返回的结果添加到另一个 DOM 文档中。

在使用 transformToDocument() 时，只要传入 XML DOM，就可以将结果作为一个完全不同的 DOM 文档来使用。来看下面的例子。



```
var result = processor.transformToDocument(xmlDom);  
alert(serializeXml(result));
```

---

*XsltProcessorExample01.htm*

而 transformToFragment() 方法接收两个参数：要转换的 XML DOM 和应该拥有结果片段的文档。换句话说，如果你想将返回的片段插入到页面中，只要将 document 作为第二个参数即可。下面来看一个例子。

```
var fragment = processor.transformToDocument(xmlDom, document);  
var div = document.getElementById("divResult");  
div.appendChild(fragment);
```

---

*XsltProcessorExample02.htm*

这里，处理器创建了一个由 document 对象拥有的片段。这样，就可以将返回的片段添加到页面中已有的<div>元素中了。

在 XSLT 样式表的输出格式为 "xml" 或 "html" 的情况下，创建文档或文档片段会非常有用。不过，在输出格式为 "text" 时，我们通常只希望得到转换的文本结果。可惜的是，没有方法能够直接返回文本。当输出格式为 "text" 时调用 transformToDocument()，仍然会返回一个完整的 XML 文档，但这个文档的内容在不同浏览器中却不一样。例如，Safari 会返回一个完整的 HTML 文档，而 Opera 和 Firefox 则会返回一个只包含一个元素的文档，这个元素中包含着输出的文本。

使用 transformToFragment() 方法可以解决这个问题，这个方法返回的是只包含一个子节点的文档片段，而子节点中包含着结果文本。然后，使用下列代码就可以取得其中的文本。

```
var fragment = processor.transformToFragment(xmlDom, document);  
var text = fragment.firstChild.nodeValue;  
alert(text);
```

以上代码能够在支持的浏览器中一致地运行，而且能够恰好返回转换得到的输出文本。

### 1. 使用参数

XSLTProcessor 也支持使用 setParameter() 来设置 XSLT 的参数，这个方法接收三个参数：命名空间

URI、参数的内部名称和要设置的值。通常，命名空间 URI 都是 null，而内部名称就是参数的名称。另外，必须在调用 transformToDocument() 或 transformToFragment() 之前调用这个方法。下面来看例子。

```
var processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World! ");
var result = processor.transformToDocument(xmlldom);
```

*XsltProcessorExample03.htm*

还有两个与参数有关的方法，getParameter() 和 removeParameter()，分别用于取得和移除当前参数的值。这两个方法都要接受命名空间参数（同样，通常是 null）和参数的内部名称。例如：

```
var processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);
processor.setParameter(null, "message", "Hello World! ");

alert(processor.getParameter(null, "message"));    //输出 "Hello World!"
processor.removeParameter(null, "message");

var result = processor.transformToDocument(xmlldom);
```

这两个方法并不常用，提供它们只是为了方便起见。

## 2. 重置处理器

每个 XSLTProcessor 的实例都可以重用，以便使用不同的 XSLT 样式表执行不同的转换。重置处理器时要调用 reset() 方法，这个方法会从处理器中移除所有参数和样式表。然后，你就可以再次调用 importStylesheet()，以加载不同的 XSLT 样式表，如下面的例子所示。

```
var processor = new XSLTProcessor()
processor.importStylesheet(xsltdom);

//执行转换

processor.reset();
processor.importStylesheet(xsltdom2);

//再执行转换
```

在需要基于多个样式表进行转换时，重用 XSLTProcessor 可以节省内存。

## 18.3.3 跨浏览器使用 XSLT

IE 对 XSLT 转换的支持与 XSLTProcessor 的区别实在太大，因此要想重新实现二者所有这方面的功能并不现实。因此，跨浏览器兼容性最好的 XSLT 转换技术，只能是返回结果字符串。为此在 IE 中只需在上下文节点上调用 transformNode() 即可，而在其他浏览器中则需要序列化 transformToDocument() 操作的结果。下面这个函数可以在 IE、Firefox、Chrome、Safari 和 Opera 中使用。

```
function transform(context, xslt){
    if (typeof XSLTProcessor != "undefined"){
        var processor = new XSLTProcessor();
        processor.importStylesheet(xslt);
```

```

        var result = processor.transformToDocument(context);
        return (new XMLSerializer()).serializeToString(result);

    } else if (typeof context.transformNode != "undefined") {
        return context.transformNode(xslt);
    } else {
        throw new Error("No XSLT processor available.");
    }
}

```

*CrossBrowserXsltExample01.htm*

这个 transform() 函数接收两个参数：要执行转换的上下文节点和 XSLT 文档对象。首先，它检测是否有 XSLTProcessor 类型的定义，如果有则使用该类型来进行转换。在调用 transformToDocument() 方法之后，将返回的结果序列化为字符串。如果上下文节点中有 transformNode() 方法，则调用该方法并返回结果。与本章中其他的跨浏览器函数一样，transform() 也会在 XSLT 处理器无效的情况下抛出错误。下面是使用这个函数的示例。

```
var result = transform(xmlDom, xsltDom);
```

使用 IE 的 transformNode() 方法，可以确保不必使用线程安全的 DOM 文档进行转换。



注意，由于不同浏览器的 XSLT 引擎不一样，因此转换得到的结果在不同浏览器间可能会稍有不同，也可能会差别很大。因此，不能绝对依赖在 JavaScript 中使用 XSLT 进行转换的结果。

## 18.4 小结

JavaScript 对 XML 及其相关技术有相当大的支持。然而，由于缺乏规范，共同的功能却存在一些不同的实现。DOM2 级提供了创建空 XML 文档的 API，但没有涉及解析和序列化。既然规范没有对这些功能作出规定，浏览器提供商就各行其是，拿出了自己的实现方案。IE 采取了下列方式。

- 通过 ActiveX 对象来支持处理 XML，而相同的对象也可以用来构建桌面应用程序。
- Windows 携带了 MSXML 库，JavaScript 能够访问这个库。
- 这个库中包含对基本 XML 解析和序列化的支持，同时也支持 XPath 和 XSLT 等技术。

Firefox 为处理 XML 的解析和序列化，实现了两个新类型，简介如下。

- DOMParser 类型比较简单，其对象可以将 XML 字符串解析为 DOM 文档。
- XMLSerializer 类型执行相反的操作，即将 DOM 文档序列化为 XML 字符串。

由于 Firefox 中的类型比较简单，用户众多，IE9、Opera、Chrome 和 Safari 都相继实现了相同的类型。因此，这些类型也就成为了 Web 开发中的事实标准。

DOM3 级引入了一个针对 XPath API 的规范，该规范已经由 Firefox、Safari、Chrome 和 Opera 实现。这些 API 可以让 JavaScript 基于 DOM 文档运行任何 XPath 查询，并且能够返回任何数据的结果。IE 以自己的方式实现了对 XPath 的支持；具体来说，就是两个方法：selectSingleNode() 和 selectNodes()。虽然与 DOM3 级 API 相比还存在诸多限制，但使用这两个方法仍然能够执行基本的 XPath 功能，即在 DOM 文档中查找节点或节点集合。



与 XML 相关的最后一种技术是 XSLT，没有公开发布的标准针对这种技术的功能定义相应的 API。Firefox 为通过 JavaScript 处理转换创建了 `XSLTProcessor` 类型；此后不久，Safari、Chrome、和 Opera 也都实现了同样的类型。IE 则针对 XSLT 提供了自己的方案，一个是简单的 `transformNode()` 方法，另一个是较为复杂的模板/处理器手段。

目前，IE、Firefox、Chrome 和 Opera 都能够较好地支持 XML。虽然 IE 的实现与其他浏览器相比差异比较大，但仍然还是有较多的公共功能可供我们实现跨浏览器的方案。

# 第 19 章

## E4X

### 本章内容

- E4X 新增的类型
- 使用 E4X 操作 XML
- 语法的变化

2002 年，由 BEA Systems 为首的几家公司建议为 ECMAScript 增加一项扩展，以便在这门语言中添加原生的 XML 支持。2004 年 6 月，E4X（ECMAScript for XML）以 ECMA-357 标准的形式发布；2005 年 12 月又发布了修订版。E4X 本身不是一门语言，它只是 ECMAScript 语言的可选扩展。就其本身而言，E4X 为处理 XML 定义了新的语法，也定义了特定于 XML 的对象。

尽管浏览器实现这个扩展标准的步伐非常缓慢，但 Firefox 1.5 及更高版本则支持几乎全部 E4X 标准。本章主要讨论 Firefox 对 E4X 的实现。

### 19.1 E4X 的类型

作为对 ECMAScript 的扩展，E4X 定义了如下几个新的全局类型。

- XML：XML 结构中的任何一个独立的部分。
- XMLList：XML 对象的集合。
- Namespace：命名空间前缀与命名空间 URI 之间的映射。
- QName：由内部名称和命名空间 URI 组成的一个限定名。

E4X 定义的这个 4 个类型可以表现 XML 文档中的所有部分，其内部机制是将每一种类型（特别是 XML 和 XMLList）都映射为多个 DOM 类型。

#### 19.1.1 XML 类型

XML 类型是 E4X 中定义的一个重要的新类型，可以用它来表现 XML 结构中任何独立的部分。XML 的实例可以表现元素、特性、注释、处理指令或文本节点。XML 类型继承自 Object 类型，因此它也继承了所有对象默认的所有属性和方法。创建 XML 对象的方式不止一种，第一种方式是像下面这样调用其构造函数：

```
var x = new XML();
```

这行代码会创建一个空的 XML 对象，我们能够向其中添加数据。另外，也可以向构造函数中传入一个 XML 字符串，如下面的例子所示：