

# 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：

```
var x = new XML("<employee position='Software Engineer'><name>Nicholas " +  
    "Zakas</name></employee>");
```

传入到构造函数中的 XML 字符串会被解析为分层的 XML 对象。除此之外，还可以向构造函数中传入 DOM 文档或节点，以便它们的数据可以通过 E4X 来表现，语法如下：

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

虽然这些创建 XML 对象的方式都还不错，但最强大也最吸引人的方法，则是使用 XML 字面量将 XML 数据直接指定给一个变量。XML 字面量就是嵌入到 JavaScript 代码中的 XML 代码。下面来看一个例子。

```
var employee = <employee position="Software Engineer">  
    <name>Nicholas C. Zakas</name>  
</employee>;
```

*XMLTypeExample01.htm*

在这个例子中，我们将一个 XML 数据结构直接指定给了一个变量。这种简洁的语法同样可以创建一个 XML 对象，并将它赋值给 employee 变量。



Firefox 对 E4X 的实现不支持解析 XML 的开头代码（prolog）。无论 `<?xml version="1.0" ?>` 出现在传递给 XML 构造函数的文本中，还是出现在 XML 字面量中，都会导致语法错误。

XML 类型的 `toXMLString()` 方法会返回 XML 对象及其子节点的 XML 字符串表示。另一方面，该类型的 `toString()` 方法则会基于不同 XML 对象的内容返回不同的字符串。如果内容简单（纯文本），则返回文本；否则，`toString()` 方法与 `toXMLString()` 方法返回的字符串一样。来看下面的例子。

```
var data = <name>Nicholas C. Zakas</name>;  
alert(data.toString());           //"Nicholas C. Zakas"  
alert(data.toXMLString());       //"<name>Nicholas C. Zakas</name>"
```

使用这两个方法，几乎可以满足所有序列化 XML 的需求。

## 19.1.2 XMLList 类型

XMLList 类型表现 XML 对象的有序集合。XMLList 的 DOM 对等类型是 NodeList，但与 Node 和 NodeList 之间的区别相比，XML 和 XMLList 之间的区别是有意设计得比较小的。要显式地创建一个 XMLList 对象，可以像下面这样使用 XMLList 构造函数：

```
var list = new XMLList();
```

与 XML 构造函数一样，也可以向其中传入一个待解析的 XML 字符串。这个字符串可以不止包含一个文档元素，如下面的例子所示：

```
var list = new XMLList("<item/><item/>");
```

*XMLListTypeExample01.htm*

结果，保存在这个 list 变量中的 XMLList 就包含了两个 XML 对象，分别是两个 `<item/>` 元素。


还可以使用加号 (+) 操作符来组合两个或多个 XML 对象, 从而创建 XMLList 对象。加号操作符在 E4X 中已经被重载, 可以用于创建 XMLList, 如下所示:

```
var list = <item/> + <item/> ;
```

这个例子使用加号操作符组合了两个 XML 字面量, 结果得到一个 XMLList。同样的组合操作也可以使用特殊的 <> 和 </> 语法来完成, 此时不使用加号操作符, 例如:

```
var list = <><item/><item/></>;
```

尽管可以创建独立的 XMLList 对象, 但是这类对象通常是在解析较大的 XML 结构的过程中捎带着被创建出来的。来看下面的例子:



```
var employees = <employees>
  <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
  </employee>
  <employee position="Salesperson">
    <name>Jim Smith</name>
  </employee>
</employees>;
```

*XMLListTypeExample02.htm*

以上代码定义的 employees 变量中包含着一个 XML 对象, 表示 <employees/> 元素。由于这个元素又包含两个 <employee/> 元素, 因而就会创建相应的 XMLList 对象, 并将其保存在 employees.employee 中。然后, 可以使用方括号语法及位置来访问每个元素:

```
var firstEmployee = employees.employee[0];
var secondEmployee = employees.employee[1];
```

每个 XMLList 对象都有 length() 方法, 用于返回对象中包含的元素数量。例如:

```
alert(employees.employee.length()); //2
```

注意, length() 是方法, 不是属性。这一点是故意与数组和 NodeList 相区别的。

E4X 有意模糊 XML 和 XMLList 类型之间的区别, 这一点很值得关注。实际上, 一个 XML 对象与一个只包含一个 XML 对象的 XMLList 之间, 并没有显而易见的区别。为了减少两者之间的区别, 每个 XML 对象也同样有一个 length() 方法和一个由 [0] 引用的属性 (返回 XML 对象自身)。

XML 与 XMLList 之间的这种兼容性可以简化 E4X 的使用, 因为有些方法可以返回任意一个类型。

XMLList 对象的 toString() 和 toXMLString() 方法返回相同的字符串值, 也就是将其包含的 XML 对象序列化之后再拼接起来的结果。

### 19.1.3 Namespace 类型

E4X 中使用 Namespace 对象来表现命名空间。通常, Namespace 对象是用来映射命名空间前缀和命名空间 URI 的, 不过有时候并不需要前缀。要创建 Namespace 对象, 可以像下面这样使用 Namespace 构造函数:

```
var ns = new Namespace();
```

而传入 URI 或前缀加 URI, 就可以初始化 Namespace 对象, 如下所示:



```
var ns = new Namespace("http://www.wrox.com/");           //没有前缀的命名空间
var wrox = new Namespace("wrox", "http://www.wrox.com/"); //wrox 命名空间
```

---

*NamespaceTypeExample01.htm*

---

可以使用 `prefix` 和 `uri` 属性来取得 `Namespace` 对象中的信息:

```
alert(ns.uri);           //"http://www.wrox.com/"
alert(ns.prefix);        //undefined
alert(wrox.uri);         //"http://www.wrox.com/"
alert(wrox.prefix);      //"wrox"
```

---

*NamespaceTypeExample01.htm*

---

在没有给 `Namespace` 对象指定前缀的情况下, `prefix` 属性会返回 `undefined`。要想创建默认的命名空间, 应该将前缀设置为空字符串。

如果 XML 字面量中包含命名空间, 或者通过 XML 构造函数解析的 XML 字符串中包含命名空间信息, 那么就会自动创建 `Namespace` 对象。然后, 就可以通过前缀和 `namespace()` 方法来取得对 `Namespace` 对象的引用。来看下面的例子:

```
var xml = <wrox:root xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello World!</wrox:message>
</wrox:root>;

var wrox = xml.namespace("wrox");
alert(wrox.uri);
alert(wrox.prefix);
```

---

*NamespaceTypeExample02.htm*

---

在这个例子中, 我们以 XML 字面量的形式创建了一个包含命名空间的 XML 片段。而表现 `wrox` 命名空间的 `Namespace` 对象可以通过 `namespace("wrox")` 取得, 然后就可以访问这个对象的 `uri` 和 `prefix` 属性了。如果 XML 片段中有默认的命名空间, 那么向 `namespace()` 中传入空字符串, 即可取得相应的 `Namespace` 对象。

`Namespace` 对象的 `toString()` 方法始终会返回命名空间 URI。

## 19.1.4 QName 类型

`QName` 类型表现的是 XML 对象的限定名, 即命名空间与内部名称的组合。向 `QName` 构造函数中传入名称或 `Namespace` 对象和名称, 可以手工创建新的 `QName` 对象, 如下所示:

```
var wrox = new Namespace("wrox", "http://www.wrox.com/");
var wroxMessage = new QName(wrox, "message"); //表示"wrox:message"
```

---

*QNameTypeExample01.htm*

---

创建了 `QName` 对象之后, 可以访问它的两个属性: `uri` 和 `localName`。其中, `uri` 属性返回在创建对象时指定的命名空间的 URI (如果未指定命名空间, 则返回空字符串), 而 `localName` 属性返回限定名中的内部名称, 如下面的例子所示:

```
alert(wroxMessage.uri);           //"http://www.wrox.com/"
alert(wroxMessage.localName);     //"message"
```

### QNameTypeExample01.htm

这两个属性是只读的，如果你想修改它们的值，会导致错误发生。QName 对象重写了 toString() 方法，会以 uri::localName 形式返回一个字符串，对于前面的例子来说，就是"http://www.wrox.com/:message"。

在解析 XML 结构时，会为表示相应元素或特性的 XML 对象自动创建 QName 对象。可以使用这个 XML 对象的 name() 方法取得与该 XML 对象关联的 QName 对象，如下面的例子所示：

```
var xml = < wrox:root xmlns:wrox="http://www.wrox.com/" >
    <wrox:message>Hello World!</wrox:message>
</wrox:root> ;

var wroxRoot = xml.name();
alert(wroxRoot.uri);           //"http://www.wrox.com/"
alert(wroxRoot.localName);     //"root"
```

### QNameTypeExample02.htm

这样，即便没有指定命名空间信息，也会根据 XML 结构中的元素和特性创建一个 QName 对象。使用 setName() 方法并传入一个新 QName 对象，可以修改 XML 对象的限定名，如下所示：

```
xml.setName(new QName("newroot"));
```

通常，这个方法会在修改相应命名空间下的元素标签名或特性名时用到。如果该名称不属于任何命名空间，则可以像下面这样使用 setLocalName() 方法来修改内部名称：

```
xml.setLocalName("newtagname");
```

## 19.2 一般用法

在将 XML 对象、元素、特性和文本集合到一个层次化对象之后，就可以使用点号加特性或标签名的方式来访问其中不同的层次和结构。每个子元素都是父元素的一个属性，而属性名与元素的内部名称相同。如果子元素只包含文本，则相应的属性只返回文本，如下面的例子所示。

```
var employee = <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
</employee>;
alert(employee.name); //"Nicholas C. Zakas"
```

以上代码中的<name/>元素只包含文本。访问 employee.name 即可取得该文本，而在内部需要定位到<name/>元素，然后返回相应文本。由于传入到 alert() 时，会隐式调用 toString() 方法，因此显示的是<name/>中包含的文本。这就使得访问 XML 文档中包含的文本数据非常方便。如果有多个元素具有相同的标签名，则会返回 XMLList。下面再看一个例子。

```
var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
```

```

        <name>Jim Smith</name>
    </employee>
</employees>;

alert(employees.employee[0].name);    //"Nicholas C. Zakas"
alert(employees.employee[1].name);    //"Jim Smith"

```

这个例子访问了每个<employee/>元素并返回了它们<name/>元素的值。如果你不确定子元素的内部名称，或者你想访问所有子元素，不管其名称是什么，也可以像下面这样使用星号(\*)。

```

var allChildren = employees.*;    //返回所有子元素，不管其名称是什么
alert(employees.*[0].name);    //"Nicholas C. Zakas"

```

*UsageExample01.htm*

与其他属性一样，星号也可能返回 XML 对象，或返回 XMLList 对象，这要取决于 XML 结构。

要达到同样的目的，除了属性之外，还可以使用 child() 方法。将属性名或索引值传递给 child() 方法，也会得到相同的值。来看下面的例子。

```

var firstChild = employees.child(0);    //与 employees.*[0] 相同
var employeeList = employees.child("employee");    //与 employees.employee 相同
var allChildren = employees.child("**");    //与 employees.* 相同

```

为了再方便一些，还有一个 children() 方法始终返回所有子元素。例如：

```

var allChildren = employees.children();    //与 employees.* 相同

```

而另一个方法 elements() 的行为与 child() 类似，区别仅在于它只返回表示元素的 XML 对象。例如：

```

var employeeList = employees.elements("employee");    //与 employees.employee 相同
var allChildren = employees.elements("**");    //与 employees.* 相同

```

这些方法为 JavaScript 开发人员提供了访问 XML 数据的较为熟悉的语法。

要删除子元素，可以使用 delete 操作符，如下所示：

```

delete employees.employee[0];
alert(employees.employee.length());    //1

```

显然，这也正是将子节点看成属性的一个主要的优点。

## 19.2.1 访问特性

访问特性也可以使用点语法，不过其语法稍有扩充。为了区分特性名与子元素的标签名，必须在名称前面加上一个@字符。这是从 XPath 中借鉴的语法；XPath 也是使用@来区分特性和标签的名称。不过，结果可能就是这种语法看起来比较奇怪，例如：

```

var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;

alert(employees.employee[0].@position);    //"Software Engineer"

```

*AttributesExample01.htm*

与元素一样，每个特性都由一个属性来表示，而且可以通过这种简写语法来访问。以这种语法访问特性会得到一个表示特性的 XML 对象，对象的 toString() 方法始终会返回特性的值。要取得特性的名称，可以使用对象的名方法。

另外，也可以使用 child() 方法来访问特性，只要传入带有@前缀的特性的名称即可。

```
alert(employees.employee[0].child("@position")); // "Software Engineer"
```

---

*AttributesExample01.htm*

由于访问 XML 对象的属性时也可以使用 child()，因此必须使用@字符来区分标签名和特性名。

使用 attribute() 方法并传入特性名，可以只访问 XML 对象的特性。与 child() 方法不同，使用 attribute() 方法时，不需要传入带@字符的特性名。下面是一个例子。

```
alert(employees.employee[0].attribute("position")); // "Software Engineer"
```

---

*AttributesExample01.htm*

这三种访问特性的方式同时适用于 XML 和 XMLList 类型。对于 XML 对象来说，会返回一个表示相应特性的 XML 对象；对 XMLList 对象来说，会返回一个 XMLList 对象，其中包含列表中所有元素的特性 XML 对象。对于前面的例子而言，employees.employee.@position 返回的 XMLList 将包含两个对象：一个对象表示第一个 <employee/> 元素中的 position 特性，另一个对象表示第二个元素中的同一特性。

要取得 XML 或 XMLList 对象中的所有特性，可以使用 attributes() 方法。这个方法会返回一个表示所有特性的 XMLList 对象。使用这个方法与使用@\*的结果相同，如下面的例子所示。

```
// 下面两种方式都会取得所有特性
var atts1 = employees.employee[0].@*;
var atts2 = employees.employee[0].attributes();
```

在 E4X 中修改特性的值与修改属性的值一样非常简单，只要像下面这样为特性指定一个新值即可。

```
employees.employee[0].@position = "Author"; // 修改 position 特性
```

修改的特性会在内部反映出来，换句话说，此后再序列化 XML 对象，就会使用新的特性值。同样，为特性赋值的语法也可以用来添加新特性，如下面的例子所示。

```
employees.employee[0].@experience = "8 years"; // 添加 experience 特性
employees.employee[0].@manager = "Jim Smith"; // 添加 manager 特性
```

由于特性与其他 ECMAScript 属性类似，因此也可以使用 delete 操作符来删除特性，如下所示。

```
delete employees.employee[0].@position; // 删除 position 特性
```

通过属性来访问特性极大地简化了与底层 XML 结构交互的操作。

## 19.2.2 其他节点类型

E4X 定义了表现 XML 文档中所有部分的类型，包括注释和处理指令。在默认情况下，E4X 不会解析注释或处理指令，因此这些部分不会出现在最终的对象层次中。如果想让解析器解析这些部分，可以像下面这样设置 XML 构造函数的下列两个属性。

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;
```

在设置了这两个属性之后，E4X 就会将注释和处理指令解析到 XML 结构中。

由于 XML 类型可以表示所有节点，因此必须有一种方式来确定节点类型。使用 `nodeKind()` 方法可以得到 XML 对象表示的类型，该访问可能会返回 "text"、"element"、"comment"、"processing-instruction" 或 "attribute"。以下面的 XML 对象为例。

```
var employees = <employees>
  <?Don't forget the donuts?>
  <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
  </employee>
  <!--just added-->
  <employee position="Salesperson">
    <name>Jim Smith</name>
  </employee>
</employees> ;
```

我们可以通过下面的表格来说明 `nodeKind()` 返回的节点类型。

语 句	返 回 值
<code>employees.nodeKind()</code>	"element"
<code>employees.*[0].nodeKind()</code>	"processing-instruction"
<code>employees.employee[0].@position.nodeKind()</code>	"attribute"
<code>employees.employee[0].nodeKind()</code>	"element"
<code>employees.*[2].nodeKind()</code>	"comment"
<code>employees.employee[0].name.*[0].nodeKind()</code>	"text"

不能在包含多个 XML 对象的 `XMLList` 上调用 `nodeKind()` 方法；否则，会抛出一个错误。

可以只取得特定类型的节点，而这就要用到下列方法。

- ❑ `attributes()`：返回 XML 对象的所有特性。
- ❑ `comments()`：返回 XML 对象的所有子注释节点。
- ❑ `elements(tagName)`：返回 XML 对象的所有子元素。可以通过提供元素的 `tagName`（标签名）来过滤想要返回的结果。
- ❑ `processingInstructions(name)`：返回 XML 对象的所有处理指令。可以通过提供处理指令的 `name`（名称）来过滤想要返回的结果。
- ❑ `text()`：返回 XML 对象的所有文本子节点。

上述的每一个方法都返回一个包含适当 XML 对象的 `XMLList`。

使用 `hasSimpleContent()` 和 `hasComplexContent()` 方法，可以确定 XML 对象中是只包含文本，还是包含更复杂的内容。如果 XML 对象中只包含子文本节点，则前一个方法会返回 `true`；如果 XML 对象的子节点中有任何非文本节点，则后一个方法返回 `true`。来看下面的例子。

```
alert(employees.employee[0].hasComplexContent()); //true
alert(employees.employee[0].hasSimpleContent()); //false
alert(employees.employee[0].name.hasComplexContent()); //false
alert(employees.employee[0].name.hasSimpleContent()); //true
```

利用这些方法，以及前面提到的其他方法，可以极大地方便查找 XML 结构中的数据。

## 19.2.3 查询

实际上，E4X 提供的查询语法在很多方面都与 XPath 类似。取得元素或特性值的简单操作是最基本



的查询。在查询之前,不会创建表现 XML 文档结构中不同部分的 XML 对象。从底层来看,XML 和 XMLList 的所有属性事实上都是查询的结果。也就是说,引用不表现 XML 结构中某一部分的属性仍然会返回 XMLList;只不过这个 XMLList 中什么也不会包含。例如,如果基于前面的 XML 示例执行下列代码,则返回的结果就是空的。

```
var cats = employees.cat;  
alert(cats.length()); //0
```

### QueryingExample01.htm

这个查询想要查找<employees/>中的<cat/>元素,但这个元素并不存在。上面的第一行代码会返回一个空的 XMLList 对象。虽然返回的是空对象,但查询可以照常进行,而不会发生异常。

前面我们看到的大多数例子都使用点语法来访问直接的子节点。而像下面这样使用两个点,则可以进一步扩展查询的深度,查询到所有后代节点。

```
var allDescendants = employees..*; //取得<employees/>的所有后代节点
```

上面的代码会返回<employees/>元素的所有后代节点。结果中将会包含元素、文本、注释和处理指令,最后两种节点的有无取决于在 XML 构造函数上的设置(前面曾经讨论过);但结果中不会包含特性。要想取得特定标签的元素,需要将星号替换成实际的标签名。

```
var allNames = employees..name; //取得作为<employees/>后代的所有<name/>节点
```

同样的查询可以使用 descendants() 方法来完成。在不给这个方法传递参数的情况下,它会返回所有后代节点(与使用..\*相同),而传递一个名称作为参数则可以限制结果。下面就是这两种情况的例子。

```
var allDescendants = employees.descendants(); //所有后代节点  
var allNames = employees.descendants("name"); //后代中的所有<name/>元素
```

还可以取得所有后代元素中的所有特性,方法是使用下列任何一行代码。

```
var allAttributes = employees..@*; //取得所有后代元素中的所有特性  
var allAttributes2 = employees.descendants("@*"); //同上
```

与限制结果中的后代元素一样,也可以通过用完整的特性名来替换星号达到过滤特性的目的。例如:

```
var allAttributes = employees..@position; //取得所有 position 特性  
var allAttributes2 = employees.descendants("@position"); //同上
```

除了访问后代元素之外,还可以指定查询的条件。例如,要想返回 position 特性值为 "Salesperson" 的所有<employee/>元素,可以使用下面的查询:

```
var salespeople = employees.employee.@position == "Salesperson";
```

同样的语法也可以用于修改 XML 结构中的某一部分。例如,可以将第一位销售员(salesperson)的 position 特性修改为 "Senior Salesperson",代码如下:

```
employees.employee.@position == "Salesperson"[0].@position = "Senior Salesperson";
```

注意,圆括号中的表达式会返回一个包含结果的 XMLList,而方括号返回其中的第一项,然后我们重写了@position 属性的值。

使用 `parent()` 方法能够在 XML 结构上溯, 这个方法会返回一个 XML 对象, 表示当前 XML 对象的父元素。如果在 `XMLList` 上调用 `parent()` 方法, 则会返回列表中所有对象的公共父元素。下面是一个例子。

```
var employees2 = employees.employee.parent();
```

这里, 变量 `employees2` 中包含着与变量 `employees` 相同的值。在处理来源未知的 XML 对象时, 经常会用到 `parent()` 方法。

## 19.2.4 构建和操作 XML

将 XML 数据转换成 XML 对象的方式有很多种。前面曾经讨论过, 可以将 XML 字符串传递到 XML 构造函数中, 也可以使用 XML 字面量。相对而言, XML 字面量方式更方便一些, 因为可以在字面量中嵌入 JavaScript 变量, 语法是使用花括号 `{ }`。可以将 JavaScript 变量嵌入到字面量中的任意位置上, 如下面的例子所示。

```
var tagName = "color";
var color = "red";
var xml = <{tagName}>{color}</{tagName}>;

alert(xml.toXMLString()); // "<color>red</color>"
```

---

*XMLConstructionExample01.htm*

在这个例子中, XML 字面量的标签名和文本值都是使用花括号语法插入的。有了这个语法, 就可以省去在构建 XML 结构时拼接字符串的麻烦。

E4X 也支持使用标准的 JavaScript 语法来构建完整的 XML 结构。如前所述, 大多数必要的操作都是查询, 而且即便元素或特性不存在也不会抛出错误。在此基础上更进一步, 如果将一个值指定给一个不存在的元素或特性, E4X 就会首先在底层创建相应的结构, 然后完成赋值。来看下面的例子。

```
var employees = <employees/>;
employees.employee.name = "Nicholas C. Zakas";
employees.employee.@position = "Software Engineer";
```

---

*XMLConstructionExample02.htm*

这个例子一开始声明了 `<employees/>` 元素, 然后在这个元素基础上开始构建 XML 结构。第二行代码在 `<employees/>` 中创建了一个 `<employee/>` 元素和一个 `<name/>` 元素, 并指定了文本值。第三行代码添加了一个 `position` 特性并为该特性指定了值。此时构建的 XML 结构如下所示。

```
<employees>
  <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
  </employee>
</employees>
```

当然, 使用加号操作符也可以再添加一个 `<employee/>` 元素, 如下所示。

```
employees.employee += <employee position="Salesperson">
  <name>Jim Smith</name>
</employee>;
```

---

*XMLConstructionExample02.htm*


最终构建的 XML 结构如下所示：

```
<employees>
  <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
  </employee>
  <employee position="Salesperson">
    <name>Jim Smith</name>
  </employee>
</employees>
```

除了上面介绍的基本的 XML 构建语法之外，还有一些类似 DOM 的方法，简介如下。

- ❑ `appendChild(child)`：将给定的 *child* 作为子节点添加到 XMLList 的末尾。
- ❑ `copy()`：返回 XML 对象副本。
- ❑ `insertChildAfter(refNode, child)`：将 *child* 作为子节点插入到 XMLList 中 *refNode* 的后面。
- ❑ `insertChildBefore(refNode, child)`：将 *child* 作为子节点插入到 XMLList 中 *refNode* 的前面。
- ❑ `prependChild(child)`：将给定的 *child* 作为子节点添加到 XMLList 的开始位置。
- ❑ `replace(propertyName, value)`：用 *value* 值替换名为 *propertyName* 的属性，这个属性可能是一个元素，也可能是一个特性。
- ❑ `setChildren(children)`：用 *children* 替换当前所有的子元素，*children* 可以是 XML 对象，也可能是 XMLList 对象。

这些方法既非常有用，也非常容易使用。下列代码展示了这些方法的用途。



```
var employees = <employees>
  <employee position="Software Engineer">
    <name>Nicholas C. Zakas</name>
  </employee>
  <employee position="Salesperson">
    <name>Jim Smith</name>
  </employee>
</employees>

employees.appendChild(<employee position="Vice President">
  <name>Benjamin Anderson</name>
</employee>);

employees.prependChild(<employee position="User Interface Designer">
  <name>Michael Johnson</name>
</employee>);

employees.insertChildBefore(employees.child(2),
  <employee position="Human Resources Manager">
    <name>Margaret Jones</name>
  </employee>);

employees.setChildren(<employee position="President">
  <name>Richard McMichael</name>
</employee> +
  <employee position="Vice President">
    <name>Rebecca Smith</name>
  </employee>);
```

以上代码首先在员工列表的底部添加了一个名为 Benjamin Anderson 的副总统 (vice president)。然后, 在员工列表顶部又添加了一个名为 Michael Johnson 的界面设计师。接着, 在列表中位置为 2 的员工——此时这个员工是 Jim Smith, 因为他前面还有 Michael Johnson 和 Nicholas C. Zakas——之前又添加了一个名为 Margaret Jones 的人力资源部经理。最后, 所有这些子元素都被总统 Richard McMichael 和副总统 Rebecca Smith 替代。结果 XML 如下所示。

```
<employees>
  <employee position="President">
    <name>Richard McMichael</name>
  </employee>
  <employee position="Vice President">
    <name>Rebecca Smith</name>
  </employee>
</employees>
```

熟练运用这些技术和方法, 就能够使用 E4X 执行任何 DOM 风格的操作。

## 19.2.5 解析和序列化

E4X 将解析和序列化数据的控制放在了 XML 构造函数的一些设置当中。与 XML 解析相关的设置有如下三个。

- ☐ ignoreComments: 表示解析器应该忽略标记中的注释。默认设置为 true。
- ☐ ignoreProcessingInstructions: 表示解析器应该忽略标记中的处理指令。默认设置为 true。
- ☐ ignoreWhitespace: 表示解析器应该忽略元素间的空格, 而不是创建表现这些空格的文本节点。默认设置为 true。

这三个设置会影响对传入到 XML 构造函数中的字符串以及 XML 字面量的解析。

另外, 与 XML 数据序列化相关的设置有如下两个。

- ☐ prettyIndent: 表示在序列化 XML 时, 每次缩进的空格数量。默认值为 2。
- ☐ prettyPrinting: 表示应该以方便人类认读的方式输出 XML, 即每个元素重起一行, 而且子元素都要缩进。默认设置为 true。

这两个设置将影响到 toString() 和 toXMLString() 的输出。

以上五个设置都保存在 settings 对象中, 通过 XML 构造函数的 settings() 方法可以取得这个对象, 如下所示。

```
var settings = XML.settings();
alert(settings.ignoreWhitespace);    //true
alert(settings.ignoreComments);      //true
```

*ParsingAndSerializationExample01.htm*

通过向 setSettings() 方法中传入包含全部 5 项设置的对象, 可以一次性指定所有设置。在需要临时改变设置的情况下, 这种设置方式非常有用, 如下所示。

```
var settings = XML.settings();
XML.prettyIndent = 8;
XML.ignoreComments = false;
```

```
//执行某些处理
XML.setSettings(settings);    //重置前面的设置
```

而使用 `defaultSettings()` 方法则可以取得一个包含默认设置的对象，因此任何时候都可以使用下面的代码重置设置。

```
XML.setSettings(XML.defaultSettings());
```

## 19.2.6 命名空间

E4X 提供了方便使用命名空间的特性。前面曾经讨论过，使用 `namespace()` 方法可以取得与特定前缀对应的 `Namespace` 对象。而通过使用 `setNamespace()` 并传入 `Namespace` 对象，也可以为给定元素设置命名空间。来看下面的例子。

```
var messages = <messages>
    <message>Hello world!</message>
</messages>;
messages.setNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

调用 `setNamespace()` 方法后，相应的命名空间只会应用到调用这个方法元素。此时，序列化 `messages` 变量会得到如下结果。

```
<wrox:messages xmlns:wrox="http://www.wrox.com/">
    <message>Hello world!</message>
</wrox:messages>
```

可见，由于调用了 `setNamespace()` 方法，`<messages/>` 元素有了 `wrox` 命名空间前缀，而 `<message/>` 元素则没有变化。

如果只想添加一个命名空间声明，而不想改变元素，可以使用 `addNamespace()` 方法并传入 `Namespace` 对象，如下面的例子所示。

```
messages.addNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

在将这行代码应用于原先的 `<messages/>` 元素时，就会创建如下所示的 XML 结构。

```
<messages xmlns:wrox="http://www.wrox.com/">
    <message>Hello world!</message>
</messages>
```

调用 `removeNamespace()` 方法并传入 `Namespace` 对象，可以移除表示特定命名空间前缀和 URI 的命名空间声明；注意，必须传入丝毫不差的表示命名空间的 `Namespace` 对象。例如：

```
messages.removeNamespace(new Namespace("wrox", "http://www.wrox.com/"));
```

这行代码可以移除 `wrox` 命名空间。不过，引用前缀的限定名不会受影响。

有两个方法可以返回与节点相关的 `Namespace` 对象的数组：`namespaceDeclarations()` 和 `inScopeNamespaces()`。前者返回在给定节点上声明的所有命名空间的数组，后者返回位于给定节点作用域中（即包括在节点自身和祖先元素中声明的）所有命名空间的数组。如下面的例子所示：

```
var messages = <messages xmlns:wrox="http://www.wrox.com/">
    <message>Hello world!</message>
</messages>;
```

```

alert(messages.namespaceDeclarations()); // "http://www.wrox.com"
alert(messages.inScopeNamespaces()); // ",http://www.wrox.com"

alert(messages.message.namespaceDeclarations()); // ""
alert(messages.message.inScopeNamespaces()); // ",http://www.wrox.com"

```

这里，`<messages/>`元素在调用 `namespaceDeclarations()` 时，会返回包含一个命名空间的数组，而在调用 `inScopeNamespaces()` 时，则会返回包含两个命名空间的数组。作用域中的这两个命名空间，分别是默认命名空间（由空字符串表示）和 `wrox` 命名空间。在 `<message/>` 元素上调用这些方法时，`namespaceDeclarations()`，会返回一个空数组，而 `inScopeNamespaces()` 方法返回的结果与在 `<messages/>` 元素上调用时的返回结果相同。

使用双冒号 (`::`) 也可以基于 `Namespace` 对象来查询 XML 结构中具有特定命名空间的元素。例如，要取得包含在 `wrox` 命名空间中的所有 `<message/>` 元素，可以参考下面的代码。

```

var messages = <messages xmlns:wrox="http://www.wrox.com/">
    <wrox:message>Hello world!</message>
</messages>;
var wroxNS = new Namespace("wrox", "http://www.wrox.com/");
var wroxMessages = messages.wroxNS::message;

```

这里的双冒号表示返回的元素应该位于其中的命名空间。注意，这里使用的是 JavaScript 变量，而不是命名空间前缀。

还可以为某个作用域中的所有 XML 对象设置默认命名空间。为此，要使用 `default xml namespace` 语句，并将一个 `Namespace` 对象或一个命名空间 URI 作为值赋给它。例如：

```

default xml namespace = "http://www.wrox.com/";

function doSomething(){
    // 只为这个函数设置默认的命名空间
    default xml namespace = new Namespace("your", "http://www.yourdomain.com");
}

```

在 `doSomething()` 函数体内设置默认命名空间并不会改变全局作用域中的默认 XML 命名空间。在给定作用域中，当所有 XML 数据都需要使用特定的命名空间时，就可以使用这个语句，从而避免多次引用命名空间的麻烦。

## 19.3 其他变化

为了与 ECMAScript 做到无缝集成，E4X 也对语言基础进行了一些修改。其中之一就是引入了 `for-each-in` 循环，以便迭代遍历每一个属性并返回属性的值，如下面的例子所示。

```

var employees = <employees>
    <employee position="Software Engineer">
        <name>Nicholas C. Zakas</name>
    </employee>
    <employee position="Salesperson">
        <name>Jim Smith</name>
    </employee>
</employees>;

```



```
for each (var child in employees){
    alert(child.toXMLString());
}
```

*ForEachInExample01.htm*

在这个例子的 for-each-in 循环中，<employees/>的每个子节点会依次被赋值给 child 变量，其中包括注释、处理指令和/或文本节点。要想返回特性节点，则需要对一个由特性节点组成的 XMLList 对象进行操作，如下所示。

```
for each (var attribute in employees.@*){ //遍历特性
    alert(attribute);
}
```

虽然 for-each-in 循环是在 E4X 中定义的，但这个语句也可以用于常规的数组和对象，例如：

```
var colors = ["red","green","blue"];
for each(var color in colors){
    alert(color);
}
```

*ForEachInExample01.htm*

对于数组，for-each-in 循环会返回数组中的每一项。对于非 XML 对象，这个循环返回对象每个属性的值。

E4X 还添加了一个全局函数，名叫 isXMLName()。这个函数接受一个字符串，并在这个字符串是元素或特性的有效内部名称的情况下返回 true。在使用未知字符串构建 XML 数据结构时，这个函数可以为开发人员提供方便。来看下面的例子。

```
alert(isXMLName("color")); //true
alert(isXMLName("hello world")); //false
```

如果你不确定某个字符串的来源，而又需要将该字符串用作一个内部名称，那么最好在使用它之前先通过 isXMLName() 检测一下是否有效，以防发生错误。

E4X 对标准 ECMAScript 的最后一个修改是 typeof 操作符。在对 XML 对象或 XMLList 对象使用这个操作符时，typeof 返回字符串 "xml"。但在对其他对象使用这个操作符时，返回的都是 "object"，例如：

```
var xml = new XML();
var list = new XMLList();
var object = {};

alert(typeof xml); //"xml"
alert(typeof list); //"xml"
alert(typeof object); //"object"
```

多数情况下，都没有必要区分 XML 和 XMLList 对象。在 E4X 中，这两个对象都被看成是基本数据类型，因而也无法通过 instanceof 操作符来将它们区分开来。

## 19.4 全面启用 E4X

鉴于 E4X 在很多方面给标准 JavaScript 带来了不同，因此 Firefox 在默认情况下只启用 E4X 中与其

他代码能够相安无事的那些特性。要想完整地启用 E4X，需要将<script>标签的 type 特性设置为 "text/javascript;e4x=1"，例如：

```
<script type="text/javascript;e4x=1" src="e4x_file.js"></script>
```

在打开这个“开关”之后，就会全面启用 E4X，从而能够正确地解析嵌入在 E4X 字面量中的注释和 CData 片段。在没有完整启用 E4X 的情况下使用注释和/或 CData 片段会导致语法错误。

## 19.5 小结

E4X 是以 ECMA-357 标准的形式发布的对 ECMAScript 的一个扩展。E4X 的目的是为操作 XML 数据提供与标准 ECMAScript 更相近的语法。E4X 具有下列特征。

- 与 DOM 不同，E4X 只用一个类型来表示 XML 中的各种节点。
  - XML 对象中封装了对所有节点都有用的数据和行为。为表现多个节点的集合，这个规范定义了 XMLList 类型。
  - 另外两个类型，Namespace 和 QName，分别表现命名空间和限定名。
- 为便于查询 XML 结构，E4X 还修改了标准的 ECMAScript 语法，修改的地方如下。
- 使用两个点 (..) 表示要匹配所有后代元素，使用 @ 字符表示应该返回一或多个特性。
  - 星号字符 (\*) 是一个通配符，可以匹配任意类型的节点。
  - 所有这些查询都可以通过一组执行相同操作的方法来实现。

到 2011 年底，Firefox 还是唯一一个支持 E4X 的浏览器。尽管没有其他浏览器提供商承诺会实现 E4X，但在服务器上，由于 BEA Workshop for WebLogic 和 Yahoo! YQL 的推动，E4X 已经取得了不小的成功。