```
      throw new Error("Parsing error!");
    }
  } catch (ex) {
    console.log("Parsing error!");
  }
```

这个例子中解析的 XML 字符串少一个</root>标签，因此会导致解析错误。IE 此时会抛出错误。Firefox 和 Opera 此时会返回 document 元素为<parsererror>的文档，而在 Chrome 和 Safari 返回的文档中，<parsererror>是<root>的第一个子元素。调用 getElementsByTagName("parsererror") 可适用于后两种情况。如果该方法返回了任何元素，就说明有错误，会弹警告框给出提示。当然，此时可以进一步解析出错误信息并显示出来。

### 22.1.3 XMLSerializer 类型

与 DOMParser 相对，Firefox 也增加了 XMLSerializer 类型用于提供相反的功能：把 DOM 文档序列化为 XML 字符串。此后，XMLSerializer 也得到了所有主流浏览器的支持。

要序列化 DOM 文档，必须创建 XMLSerializer 的新实例，然后把文档传给 serializeToString() 方法，如下所示：

```
let serializer = new XMLSerializer();
let xml = serializer.serializeToString(xmldom);
console.log(xml);
```

serializeToString()方法返回的值是打印效果不好的字符串，因此肉眼看起来有点困难。

XMLSerializer 能够序列化任何有效的 DOM 对象，包括个别节点和 HTML 文档。在把 HTML 文档传给 serializeToString()时，这个文档会被当成 XML 文档，因此得到的结果是格式良好的。

> **注意**　如果给 serializeToString()传入非 DOM 对象，就会导致抛出错误。

## 22.2 浏览器对 XPath 的支持

XPath 是为了在 DOM 文档中定位特定节点而创建的，因此它对 XML 处理很重要。在 DOM Level 3 之前，XPath 相关的 API 没有被标准化。DOM Level 3 开始着手标准化 XPath。很多浏览器实现了 DOM Level 3 XPath 标准，但 IE 决定按照自己的方式实现。

### 22.2.1 DOM Level 3 XPath

DOM Level 3 XPath 规范定义了接口，用于在 DOM 中求值 XPath 表达式。要确定浏览器是否支持 DOM Level 3 XPath，可以使用以下代码：

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

虽然这个规范定义了不少类型，但其中最重要的两个是 XPathEvaluator 和 XPathResult。XPathEvaluator 用于在特定上下文中求值 XPath 表达式，包含三个方法。

- createExpression(*expression, nsresolver*)，用于根据 XPath 表达式及相应的命名空间计算得到一个 XPathExpression，XPathExpression 是查询的编译版本。这适合于同样的查询要运行多次的情况。

- createNSResolver(*node*)，基于 *node* 的命名空间创建新的 XPathNSResolver 对象。当对使用名称空间的 XML 文档求值时，需要 XPathNSResolver 对象。
- evaluate(*expression, context, nsresolver, type, result*)，根据给定的上下文和命名空间对 XPath 进行求值。其他参数表示如何返回结果。

Document 类型通常是通过 XPathEvaluator 接口实现的，因此可以创建 XPathEvaluator 的实例，或使用 Document 实例上的方法（包括 XML 和 HTML 文档）。

在上述三个方法中，使用最频繁的是 evaluate()。这个方法接收五个参数：XPath 表达式、上下文节点、命名空间解析器、返回的结果类型和 XPathResult 对象（用于填充结果，通常是 null，因为结果也可能是函数值）。第三个参数，命名空间解析器，只在 XML 代码使用 XML 命名空间的情况下有必要。如果没有使用命名空间，这个参数也应该是 null。第四个参数要返回值的类型是如下 10 个常量值之一。

- XPathResult.ANY_TYPE：返回适合 XPath 表达式的数据类型。
- 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：返回匹配节点的集合，但集合中节点的顺序可能与它们在文档中的顺序不一致。
- XPathResult.ORDERED_NODE_ITERATOR_TYPE：返回匹配节点的集合，集合中节点的顺序与它们在文档中的顺序一致。这是非常常用的结果类型。
- XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE：返回节点集合的快照，在文档外部捕获节点，因此对文档的进一步修改不会影响该节点集合。集合中节点的顺序可能与它们在文档中的顺序不一致。
- XPathResult.ORDERED_NODE_SNAPSHOT_TYPE：返回节点集合的快照，在文档外部捕获节点，因此对文档的进一步修改不会影响这个节点集合。集合中节点的顺序与它们在文档中的顺序一致。
- XPathResult.ANY_UNORDERED_NODE_TYPE：返回匹配节点的集合，但集合中节点的顺序可能与它们在文档中的顺序不一致。
- XPathResult.FIRST_ORDERED_NODE_TYPE：返回只有一个节点的节点集合，包含文档中第一个匹配的节点。

指定的结果类型决定了如何获取结果的值。下面是一个典型的示例：

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

if (result !== null) {
  let element = result.iterateNext();
  while(element) {
    console.log(element.tagName);
    node = result.iterateNext();
  }
}
```

这个例子使用了 XPathResult.ORDERED_NODE_ITERATOR_TYPE 结果类型，也是最常用的类型。如果没有节点匹配 XPath 表达式，evaluate()方法返回 null；否则，返回 XPathResult 对象。返回的 XPathResult 对象上有相应的属性和方法用于获取特定类型的结果。如果结果是节点迭代器，无论

有序还是无序，都必须使用 iterateNext()方法获取结果中每个匹配的节点。在没有更多匹配节点时，iterateNext()返回 null。

如果指定了快照结果类型（无论有序还是无序），都必须使用 snapshotItem()方法和 snapshotLength 属性获取结果，如以下代码所示：

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

这个例子中，snapshotLength 返回快照中节点的数量，而 snapshotItem()返回快照中给定位置的节点（类似于 NodeList 中的 length 和 item()）。

### 22.2.2　单个节点结果

XPathResult.FIRST_ORDERED_NODE_TYPE 结果类型返回匹配的第一个节点，可以通过结果的 singleNodeValue 属性获取。比如：

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

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

与其他查询一样，如果没有匹配的节点，evaluate()返回 null。如果有一个匹配的节点，则要使用 singleNodeValue 属性取得该节点。这对 XPathResult.FIRST_ORDERED_NODE_TYPE 也一样。

### 22.2.3　简单类型结果

使用布尔值、数值和字符串 XPathResult 类型，可以根据 XPath 获取简单、非节点数据类型。这些结果类型返回的值需要分别使用 booleanValue、numberValue 和 stringValue 属性获取。对于布尔值类型，如果至少有一个节点匹配 XPath 表达式，booleanValue 就是 true；否则，booleanValue 为 false。比如：

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                  XPathResult.BOOLEAN_TYPE, null);
console.log(result.booleanValue);
```

在这个例子中，如果有任何节点匹配"employee/name"，booleanValue 属性就等于 true。

对于数值类型，XPath 表达式必须使用返回数值的 XPath 函数，如 count()可以计算匹配给定模式的节点数。比如：

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

以上代码会输出匹配"employee/name"的节点数量（比如 2）。如果在这里没有指定 XPath 函数，numberValue 就等于 NaN。

对于字符串类型，evaluate()方法查找匹配 XPath 表达式的第一个节点，然后返回其第一个子节点的值，前提是第一个子节点是文本节点。如果不是，就返回空字符串。比如：

```
let result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
                  XPathResult.STRING_TYPE, null);
console.log(result.stringValue);
```

这个例子输出了与"employee/name"匹配的第一个元素中第一个文本节点包含的文本字符串。

### 22.2.4 默认类型结果

所有 XPath 表达式都会自动映射到特定类型的结果。设置特定结果类型会限制表达式的输出。不过，可以使用 XPathResult.ANY_TYPE 类型让求值自动返回默认类型结果。通常，默认类型结果是布尔值、数值、字符串或无序节点迭代器。要确定返回的结果类型，可以访问求值结果的 resultType 属性，如下例所示：

```
let 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:
      // 处理无序节点迭代器类型
      break;

    default:
      // 处理其他可能的结果类型

  }
}
```

使用 XPathResult.ANY_TYPE 可以让使用 XPath 变得更自然，但在返回结果后则需要增加额外的判断和处理。

### 22.2.5 命名空间支持

对于使用命名空间的 XML 文档，必须告诉 XPathEvaluator 命名空间信息，才能进行正确求值。处理命名空间的方式有很多，看下面的示例 XML 代码：

```
<?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，所有元素的命名空间都属于 http://www.wrox.com/，都以 `wrox` 前缀标识。如果想使用 XPath 查询该文档，就需要指定使用的命名空间，否则求值会失败。

第一种处理命名空间的方式是通过 `createNSResolver()`方法创建 `XPathNSResolver` 对象。这个方法只接收一个参数，即包含命名空间定义的文档节点。对上面的例子而言，这个节点就是 `document` 元素`<wrox:books>`，其 `xmlns` 属性定义了命名空间。为此，可以将该节点传给 `createNSResolver()`，然后得到的结果就可以在 `evaluate()`方法中使用：

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

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

console.log(result.snapshotLength);
```

把 `nsresolver` 传给 `evaluate()`之后，可以确保 XPath 表达式中使用的 `wrox` 前缀能够被正确理解。假如不使用 `XPathNSResolver`，同样的表达式就会导致错误。

第二种处理命名空间的方式是定义一个接收命名空间前缀并返回相应 URI 的函数，如下所示：

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

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

console.log(result.numberValue);
```

在并不知晓文档的哪个节点包含命名空间定义时，可以采用这种定义命名空间解析函数的方式。只要知道前缀和 URI，就可以定义这样一个函数，然后把它作为第三个参数传给 `evaluate()`。

## 22.3　浏览器对 XSLT 的支持

可扩展样式表语言转换（XSLT，Extensible Stylesheet Language Transformations）是与 XML 相伴的一种技术，可以利用 XPath 将一种文档表示转换为另一种文档表示。与 XML 和 XPath 不同，XSLT 没有与之相关的正式 API，正式的 DOM 中也没有涵盖它。因此浏览器都以自己的方式实现 XSLT。率先在 JavaScript 中支持 XSLT 的是 IE。

### 22.3.1　`XSLTProcessor` 类型

Mozilla 通过增加了一个新类型 `XSLTProcessor`，在 JavaScript 中实现了对 XSLT 的支持。通过使

用 XSLTProcessor 类型，开发者可以使用 XSLT 转换 XML 文档，其方式类似于在 IE 中使用 XSL 处理器。自从 XSLTProcessor 首次实现以来，所有浏览器都照抄了其实现，从而使 XSLTProcessor 成了通过 JavaScript 完成 XSLT 转换的事实标准。

与 IE 的实现一样，第一步是加载两个 DOM 文档：XML 文档和 XSLT 文档。然后，使用 importStyleSheet()方法创建一个新的 XSLTProcessor，将 XSLT 指定给它，如下所示：

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

最后一步是执行转换，有两种方式。如果想返回完整的 DOM 文档，就调用 transformToDocument()；如果想得到文档片段，则可以调用 transformToFragment()。一般来说，使用 transformToFragment()的唯一原因是想把结果添加到另一个 DOM 文档。

如果使用 transformToDocument()，只要传给它 XML DOM，就可以将结果当作另一个完全不同的 DOM 来使用。比如：

```
let result = processor.transformToDocument(xmldom);
console.log(serializeXml(result));
```

transformToFragment()方法接收两个参数：要转换的 XML DOM 和最终会拥有结果片段的文档。这可以确保新文本片段可以在目标文档中使用。比如，可以把 document 作为第二个参数，然后将创建的片段添加到其页面元素中。比如：

```
let fragment = processor.transformToFragment(xmldom, document);
let div = document.getElementById("divResult");
div.appendChild(fragment);
```

这里，处理器创建了由 document 对象所有的片段。这样就可以将片段添加到当前页面的 元素中了。

如果 XSLT 样式表的输出格式是"xml"或"html"，则创建文档或文档片段理所当然。不过，如果输出格式是"text"，则通常意味着只想得到转换后的文本结果。然而，没有方法直接返回文本。在输出格式为"text"时调用 transformToDocument()会返回完整的 XML 文档，但这个文档的内容会因浏览器而异。比如，Safari 返回整个 HTML 文档，而 Opera 和 Firefox 则返回只包含一个元素的文档，其中输出就是该元素的文本。

解决方案是调用 transformToFragment()，返回只有一个子节点、其中包含结果文本的文档片段。之后，可以再使用以下代码取得文本：

```
let fragment = processor.transformToFragment(xmldom, document);
let text = fragment.firstChild.nodeValue;
console.log(text);
```

这种方式在所有支持的浏览器中都可以正确返回转换后的输出文本。

### 22.3.2 使用参数

XSLTProcessor 还允许使用 setParameter()方法设置 XSLT 参数。该方法接收三个参数：命名空间 URI、参数本地名称和要设置的值。通常，命名空间 URI 是 null，本地名称就是参数名称。setParameter()方法必须在调用 transformToDocument()或 transformToFragment()之前调用。例子如下：

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

与参数相关的还有两个方法：getParameter()和 removeParameter()。它们分别用于取得参数的当前值和移除参数的值。它们都以一个命名空间 URI（同样，一般是 null）和参数的本地名称为参数。比如：

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

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

let result = processor.transformToDocument(xmldom);
```

这几个方法并不常用，只是为了操作方便。

### 22.3.3　重置处理器

每个 XSLTProcessor 实例都可以重用于多个转换，只是要使用不同的 XSLT 样式表。处理器的 reset()方法可以删除所有参数和样式表。然后，可以使用 importStylesheet()方法加载不同的 XSLT 样表，如下所示：

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

// 执行某些转换

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

// 再执行一些转换
```

在使用多个样式表执行转换时，重用一个 XSLTProcessor 可以节省内存。

## 22.4　小结

浏览器对使用 JavaScript 处理 XML 实现及相关技术相当支持。然而，由于早期缺少规范，常用的功能出现了不同实现。DOM Level 2 提供了创建空 XML 文档的 API，但不能解析和序列化。浏览器为解析和序列化 XML 实现了两个新类型。

- DOMParser 类型是简单的对象，可以将 XML 字符串解析为 DOM 文档。
- XMLSerializer 类型执行相反操作，将 DOM 文档序列化为 XML 字符串。

基于所有主流浏览器的实现，DOM Level 3 新增了针对 XPath API 的规范。该 API 可以让 JavaScript 针对 DOM 文档执行任何 XPath 查询并得到不同数据类型的结果。

最后一个与 XML 相关的技术是 XSLT，目前并没有规范定义其 API。Firefox 最早增加了 XSLTProcessor 类型用于通过 JavaScript 处理转换。

# 第23章 JSON

本章内容

- 理解 JSON 语法
- 解析 JSON
- JSON 序列化


正如上一章所说，XML 曾经一度成为互联网上传输数据的事实标准。第一代 Web 服务很大程度上是以 XML 为基础的，以服务器间通信为主要特征。可是，XML 也并非没有批评者。有的人认为 XML 过于冗余和啰唆。为解决这些问题，也出现了几种方案。不过 Web 已经朝着它的新方向进发了。

2006 年，Douglas Crockford 在国际互联网工程任务组（IETF，The Internet Engineering Task Force）制定了 JavaScript 对象简谱（JSON，JavaScript Object Notation）标准，即 RFC 4627。但实际上，JSON 早在 2001 年就开始使用了。JSON 是 JavaScript 的严格子集，利用 JavaScript 中的几种模式来表示结构化数据。Crockford 将 JSON 作为替代 XML 的一个方案提出，因为 JSON 可以直接传给 `eval()` 而不需要创建 DOM。

理解 JSON 最关键的一点是要把它当成一种数据格式，而不是编程语言。JSON 不属于 JavaScript，它们只是拥有相同的语法而已。JSON 也不是只能在 JavaScript 中使用，它是一种通用数据格式。很多语言都有解析和序列化 JSON 的内置能力。

## 23.1 语法

JSON 语法支持表示 3 种类型的值。

- **简单值**：字符串、数值、布尔值和 `null` 可以在 JSON 中出现，就像在 JavaScript 中一样。特殊值 `undefined` 不可以。
- **对象**：第一种复杂数据类型，对象表示有序键/值对。每个值可以是简单值，也可以是复杂类型。
- **数组**：第二种复杂数据类型，数组表示可以通过数值索引访问的值的有序列表。数组的值可以是任意类型，包括简单值、对象，甚至其他数组。

JSON 没有变量、函数或对象实例的概念。JSON 的所有记号都只为表示结构化数据，虽然它借用了 JavaScript 的语法，但是千万不要把它跟 JavaScript 语言混淆。

### 23.1.1 简单值

最简单的 JSON 可以是一个数值。例如，下面这个数值是有效的 JSON：

```
5
```

这个 JSON 表示数值 5。类似地，下面这个字符串也是有效的 JSON：

```
"Hello world!"
```

JavaScript 字符串与 JSON 字符串的主要区别是，JSON 字符串必须使用双引号（单引号会导致语法错误）。

布尔值和 null 本身也是有效的 JSON 值。不过，实践中更多使用 JSON 表示比较复杂的数据结构，其中会包含简单值。

### 23.1.2　对象

对象使用与 JavaScript 对象字面量略为不同的方式表示。以下是 JavaScript 中的对象字面量：

```
let person = {
  name: "Nicholas",
  age: 29
};
```

虽然这对 JavaScript 开发者来说是标准的对象字面量，但 JSON 中的对象必须使用双引号把属性名包围起来。下面的代码与前面的代码是一样的：

```
let object = {
  "name": "Nicholas",
  "age" : 29
};
```

而用 JSON 表示相同的对象的语法是：

```
{
  "name": "Nicholas",
  "age": 29
}
```

与 JavaScript 对象字面量相比，JSON 主要有两处不同。首先，没有变量声明（JSON 中没有变量）。其次，最后没有分号（不需要，因为不是 JavaScript 语句）。同样，用引号将属性名包围起来才是有效的 JSON。属性的值可以是简单值或复杂数据类型值，后者可以在对象中再嵌入对象，比如：

```
{
  "name": "Nicholas",
  "age": 29,
  "school": {
    "name": "Merrimack College",
    "location": "North Andover, MA"
  }
}
```

这个例子在顶级对象中又嵌入了学校相关的信息。即使整个 JSON 对象中有两个属性都叫"name"，但它们属于两个不同的对象，因此是允许的。同一个对象中不允许出现两个相同的属性。

与 JavaScript 不同，JSON 中的对象属性名必须始终带双引号。手动编写 JSON 时漏掉这些双引号或使用单引号是常见错误。

### 23.1.3　数组

JSON 的第二种复杂数据类型是数组。数组在 JSON 中使用 JavaScript 的数组字面量形式表示。例如，以下是一个 JavaScript 数组：

```
let values = [25, "hi", true];
```

在 JSON 中可以使用类似语法表示相同的数组：

```
[25, "hi", true]
```

同样，这里没有变量，也没有分号。数组和对象可以组合使用，以表示更加复杂的数据结构，比如：

```
[
  {
    "title": "Professional JavaScript",
    "authors": [
      "Nicholas C. Zakas",
      "Matt Frisbie"
    ],
    "edition": 4,
    "year": 2017
  },
  {
    "title": "Professional JavaScript",
    "authors": [
      "Nicholas C. Zakas"
    ],
    "edition": 3,
    "year": 2011
  },
  {
    "title": "Professional JavaScript",
    "authors": [
      "Nicholas C. Zakas"
    ],
    "edition": 2,
    "year": 2009
  },
  {
    "title": "Professional Ajax",
    "authors": [
      "Nicholas C. Zakas",
      "Jeremy McPeak",
      "Joe Fawcett"
    ],
    "edition": 2,
    "year": 2008
  },
  {
    "title": "Professional Ajax",
    "authors": [
      "Nicholas C. Zakas",
      "Jeremy McPeak",
      "Joe Fawcett"
    ],
    "edition": 1,
    "year": 2007
  },
  {
    "title": "Professional JavaScript",
    "authors": [
      "Nicholas C. Zakas"
    ],
```