2.1.5　XHTML 中的变化

可扩展超文本标记语言（XHTML，Extensible HyperText Markup Language）是将 HTML 作为 XML 的应用重新包装的结果。与 HTML 不同，在 XHTML 中使用 JavaScript 必须指定 `type` 属性且值为 `text/javascript`，HTML 中则可以没有这个属性。XHTML 虽然已经退出历史舞台，但实践中偶尔可能也会遇到遗留代码，为此本节稍作介绍。

在 XHTML 中编写代码的规则比 HTML 中严格，这会影响使用`<script>`元素嵌入 JavaScript 代码。下面的代码块虽然在 HTML 中有效，但在 XHML 中是无效的。

```
<script type="text/javascript">
  function compare(a, b) {
    if (a < b) {
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
</script>
```

在 HTML 中，解析`<script>`元素会应用特殊规则。XHTML 中则没有这些规则。这意味着 `a < b` 语句中的小于号（`<`）会被解释成一个标签的开始，并且由于作为标签开始的小于号后面不能有空格，这会导致语法错误。

避免 XHTML 中这种语法错误的方法有两种。第一种是把所有小于号（`<`）都替换成对应的 HTML 实体形式（`<`）。结果代码就是这样的：

```
<script type="text/javascript">
  function compare(a, b) {
    if (a &lt; b) {
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
</script>
```

这样代码就可以在 XHTML 页面中运行了。不过，缺点是会影响阅读。好在还有另一种方法。

第二种方法是把所有代码都包含到一个 CDATA 块中。在 XHTML（及 XML）中，CDATA 块表示文档中可以包含任意文本的区块，其内容不作为标签来解析，因此可以在其中包含任意字符，包括小于号，并且不会引发语法错误。使用 CDATA 的格式如下：

```
<script type="text/javascript"><![CDATA[
  function compare(a, b) {
    if (a < b) {
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
]]></script>
```

在兼容 XHTML 的浏览器中，这样能解决问题。但在不支持 CDATA 块的非 XHTML 兼容浏览器中则不行。为此，CDATA 标记必须使用 JavaScript 注释来抵消：

```
<script type="text/javascript">
//<![CDATA[
  function compare(a, b) {
    if (a < b) {
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
//]]>
</script>
```

这种格式适用于所有现代浏览器。虽然有点黑科技的味道，但它可以通过 XHTML 验证，而且对 XHTML 之前的浏览器也能优雅地降级。

> **注意** XHTML 模式会在页面的 MIME 类型被指定为"application/xhtml+xml"时触发。并不是所有浏览器都支持以这种方式送达的 XHTML。

2.1.6 废弃的语法

自 1995 年 Netscape 2 发布以来，所有浏览器都将 JavaScript 作为默认的编程语言。type 属性使用一个 MIME 类型字符串来标识<script>的内容，但 MIME 类型并没有跨浏览器标准化。即使浏览器默认使用 JavaScript，在某些情况下某个无效或无法识别的 MIME 类型也可能导致浏览器跳过（不执行）相关代码。因此，除非你使用 XHTML 或<script>标签要求或包含非 JavaScript 代码，最佳做法是不指定 type 属性。

在最初采用 script 元素时，它标志着开始走向与传统 HTML 解析不同的流程。对这个元素需要应用特殊的解析规则，而这在不支持 JavaScript 的浏览器（特别是 Mosaic）中会导致问题。不支持的浏览器会把<script>元素的内容输出到页面上，从而破坏页面的外观。

Netscape 联合 Mosaic 拿出了一个解决方案，对不支持 JavaScript 的浏览器隐藏嵌入的 JavaScript 代码。最终方案是把脚本代码包含在一个 HTML 注释中，像这样：

```
<script><!--
  function sayHi(){
    console.log("Hi!");
  }
//--></script>
```

使用这种格式，Mosaic 等浏览器就可以忽略<script>标签中的内容，而支持 JavaScript 的浏览器则必须识别这种模式，将其中的内容作为 JavaScript 来解析。

虽然这种格式仍然可以被所有浏览器识别和解析，但已经不再必要，而且不应该再使用了。在 XHTML 模式下，这种格式也会导致脚本被忽略，因为代码处于有效的 XML 注释当中。

2.2　行内代码与外部文件

虽然可以直接在 HTML 文件中嵌入 JavaScript 代码，但通常认为最佳实践是尽可能将 JavaScript 代码放在外部文件中。不过这个最佳实践并不是明确的强制性规则。推荐使用外部文件的理由如下。

- **可维护性**。JavaScript 代码如果分散到很多 HTML 页面，会导致维护困难。而用一个目录保存所有 JavaScript 文件，则更容易维护，这样开发者就可以独立于使用它们的 HTML 页面来编辑代码。
- **缓存**。浏览器会根据特定的设置缓存所有外部链接的 JavaScript 文件，这意味着如果两个页面都用到同一个文件，则该文件只需下载一次。这最终意味着页面加载更快。
- **适应未来**。通过把 JavaScript 放到外部文件中，就不必考虑用 XHTML 或前面提到的注释黑科技。包含外部 JavaScript 文件的语法在 HTML 和 XHTML 中是一样的。

在配置浏览器请求外部文件时，要重点考虑的一点是它们会占用多少带宽。在 SPDY/HTTP2 中，预请求的消耗已显著降低，以轻量、独立 JavaScript 组件形式向客户端送达脚本更具优势。

比如，第一个页面包含如下脚本：

```
<script src="mainA.js"></script>
<script src="component1.js"></script>
<script src="component2.js"></script>
<script src="component3.js"></script>
...
```

后续页面可能包含如下脚本：

```
<script src="mainB.js"></script>
<script src="component3.js"></script>
<script src="component4.js"></script>
<script src="component5.js"></script>
...
```

在初次请求时，如果浏览器支持 SPDY/HTTP2，就可以从同一个地方取得一批文件，并将它们逐个放到浏览器缓存中。从浏览器角度看，通过 SPDY/HTTP2 获取所有这些独立的资源与获取一个大 JavaScript 文件的延迟差不多。

在第二个页面请求时，由于你已经把应用程序切割成了轻量可缓存的文件，第二个页面也依赖的某些组件此时已经存在于浏览器缓存中了。

当然，这里假设浏览器支持 SPDY/HTTP2，只有比较新的浏览器才满足。如果你还想支持那些比较老的浏览器，可能还是用一个大文件更合适。

2.3　文档模式

IE5.5 发明了文档模式的概念，即可以使用 `doctype` 切换文档模式。最初的文档模式有两种：**混杂模式**（quirks mode）和**标准模式**（standards mode）。前者让 IE 像 IE5 一样（支持一些非标准的特性），后者让 IE 具有兼容标准的行为。虽然这两种模式的主要区别只体现在通过 CSS 渲染的内容方面，但对 JavaScript 也有一些关联影响，或称为副作用。本书会经常提到这些副作用。

IE 初次支持文档模式切换以后，其他浏览器也跟着实现了。随着浏览器的普遍实现，又出现了第三种文档模式：**准标准模式**（almost standards mode）。这种模式下的浏览器支持很多标准的特性，但是没有标准规定得那么严格。主要区别在于如何对待图片元素周围的空白（在表格中使用图片时最明显）。

混杂模式在所有浏览器中都以省略文档开头的 doctype 声明作为开关。这种约定并不合理，因为混杂模式在不同浏览器中的差异非常大，不使用黑科技基本上就没有浏览器一致性可言。

标准模式通过下列几种文档类型声明开启：

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

<!-- XHTML 1.0 Strict -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- HTML5 -->
<!DOCTYPE html>
```

准标准模式通过过渡性文档类型（Transitional）和框架集文档类型（Frameset）来触发：

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

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

<!-- XHTML 1.0 Transitional -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- XHTML 1.0 Frameset -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

准标准模式与标准模式非常接近，很少需要区分。人们在说到“标准模式”时，可能指其中任何一个。而对文档模式的检测（本书后面会讨论）也不会区分它们。本书后面所说的**标准模式**，指的就是除混杂模式以外的模式。

2.4 <noscript>元素

针对早期浏览器不支持 JavaScript 的问题，需要一个页面优雅降级的处理方案。最终，<noscript>元素出现，被用于给不支持 JavaScript 的浏览器提供替代内容。虽然如今的浏览器已经 100%支持 JavaScript，但对于禁用 JavaScript 的浏览器来说，这个元素仍然有它的用处。

<noscript>元素可以包含任何可以出现在<body>中的 HTML 元素，<script>除外。在下列两种情况下，浏览器将显示包含在<noscript>中的内容：

- 浏览器不支持脚本；
- 浏览器对脚本的支持被关闭。

任何一个条件被满足，包含在<noscript>中的内容就会被渲染。否则，浏览器不会渲染<noscript>中的内容。

下面是一个例子：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  <script defer="defer" src="example1.js"></script>
  <script defer="defer" src="example2.js"></script>
  </head>
  <body>
  <noscript>
    <p>This page requires a JavaScript-enabled browser.</p>
  </noscript>
  </body>
</html>
```

这个例子是在脚本不可用时让浏览器显示一段话。如果浏览器支持脚本，则用户永远不会看到它。

2.5　小结

JavaScript 是通过<script>元素插入到 HTML 页面中的。这个元素可用于把 JavaScript 代码嵌入到 HTML 页面中，跟其他标记混合在一起，也可用于引入保存在外部文件中的 JavaScript。本章的重点可以总结如下。

- 要包含外部 JavaScript 文件，必须将 src 属性设置为要包含文件的 URL。文件可以跟网页在同一台服务器上，也可以位于完全不同的域。
- 所有<script>元素会依照它们在网页中出现的次序被解释。在不使用 defer 和 async 属性的情况下，包含在<script>元素中的代码必须严格按次序解释。
- 对不推迟执行的脚本，浏览器必须解释完位于<script>元素中的代码，然后才能继续渲染页面的剩余部分。为此，通常应该把<script>元素放到页面末尾，介于主内容之后及</body>标签之前。
- 可以使用 defer 属性把脚本推迟到文档渲染完毕后再执行。推迟的脚本原则上按照它们被列出的次序执行。
- 可以使用 async 属性表示脚本不需要等待其他脚本，同时也不阻塞文档渲染，即异步加载。异步脚本不能保证按照它们在页面中出现的次序执行。
- 通过使用<noscript>元素，可以指定在浏览器不支持脚本时显示的内容。如果浏览器支持并启用脚本，则<noscript>元素中的任何内容都不会被渲染。

第3章 语言基础

本章内容

- ❑ 语法
- ❑ 数据类型
- ❑ 流控制语句
- ❑ 理解函数

任何语言的核心所描述的都是这门语言在最基本的层面上如何工作，涉及语法、操作符、数据类型以及内置功能，在此基础之上才可以构建复杂的解决方案。如前所述，ECMA-262 以一个名为 ECMAScript 的伪语言的形式，定义了 JavaScript 的所有这些方面。

ECMA-262 第 5 版（ES5）定义的 ECMAScript，是目前为止实现得最为广泛（即受浏览器支持最好）的一个版本。第 6 版（ES6）在浏览器中的实现（即受支持）程度次之。到 2017 年底，大多数主流浏览器几乎或全部实现了这一版的规范。为此，本章接下来的内容主要基于 ECMAScript 第 6 版。

3.1 语法

ECMAScript 的语法很大程度上借鉴了 C 语言和其他类 C 语言，如 Java 和 Perl。熟悉这些语言的开发者，应该很容易理解 ECMAScript 宽松的语法。

3.1.1 区分大小写

首先要知道的是，ECMAScript 中一切都区分大小写。无论是变量、函数名还是操作符，都区分大小写。换句话说，变量 `test` 和变量 `Test` 是两个不同的变量。类似地，`typeof` 不能作为函数名，因为它是一个关键字（后面会介绍）。但 `Typeof` 是一个完全有效的函数名。

3.1.2 标识符

所谓**标识符**，就是变量、函数、属性或函数参数的名称。标识符可以由一或多个下列字符组成：

- ❑ 第一个字符必须是一个字母、下划线（_）或美元符号（$）；
- ❑ 剩下的其他字符可以是字母、下划线、美元符号或数字。

标识符中的字母可以是扩展 ASCII（Extended ASCII）中的字母，也可以是 Unicode 的字母字符，如 À 和 Æ（但不推荐使用）。

按照惯例，ECMAScript 标识符使用驼峰大小写形式，即第一个单词的首字母小写，后面每个单词的首字母大写，如：

```
firstSecond
myCar
doSomethingImportant
```

虽然这种写法并不是强制性的，但因为这种形式跟 ECMAScript 内置函数和对象的命名方式一致，所以算是最佳实践。

> **注意**　关键字、保留字、true、false 和 null 不能作为标识符。具体内容请参考 3.2 节。

3.1.3　注释

ECMAScript 采用 C 语言风格的注释，包括单行注释和块注释。单行注释以两个斜杠字符开头，如：

```
// 单行注释
```

块注释以一个斜杠和一个星号（/*）开头，以它们的反向组合（*/）结尾，如：

```
/* 这是多行
注释 */
```

3.1.4　严格模式

ECMAScript 5 增加了严格模式（strict mode）的概念。严格模式是一种不同的 JavaScript 解析和执行模型，ECMAScript 3 的一些不规范写法在这种模式下会被处理，对于不安全的活动将抛出错误。要对整个脚本启用严格模式，在脚本开头加上这一行：

```
"use strict";
```

虽然看起来像个没有赋值给任何变量的字符串，但它其实是一个预处理指令。任何支持的 JavaScript 引擎看到它都会切换到严格模式。选择这种语法形式的目的是不破坏 ECMAScript 3 语法。

也可以单独指定一个函数在严格模式下执行，只要把这个预处理指令放到函数体开头即可：

```
function doSomething() {
  "use strict";
  // 函数体
}
```

严格模式会影响 JavaScript 执行的很多方面，因此本书在用到它时会明确指出来。所有现代浏览器都支持严格模式。

3.1.5　语句

ECMAScript 中的语句以分号结尾。省略分号意味着由解析器确定语句在哪里结尾，如下面的例子所示：

```
let sum = a + b       // 没有分号也有效，但不推荐
let diff = a - b;     // 加分号有效，推荐
```

即使语句末尾的分号不是必需的，也应该加上。记着加分号有助于防止省略造成的问题，比如可以避免输入内容不完整。此外，加分号也便于开发者通过删除空行来压缩代码（如果没有结尾的分号，只删除空行，则会导致语法错误）。加分号也有助于在某些情况下提升性能，因为解析器会尝试在合适的位置补上分号以纠正语法错误。

多条语句可以合并到一个 C 语言风格的代码块中。代码块由一个左花括号（{）标识开始，一个右花括号（}）标识结束：

```
if (test) {
  test = false;
  console.log(test);
}
```

if 之类的控制语句只在执行多条语句时要求必须有代码块。不过，最佳实践是始终在控制语句中使用代码块，即使要执行的只有一条语句，如下例所示：

```
// 有效，但容易导致错误，应该避免
if (test)
  console.log(test);

// 推荐
if (test) {
  console.log(test);
}
```

在控制语句中使用代码块可以让内容更清晰，在需要修改代码时也可以减少出错的可能性。

3.2 关键字与保留字

ECMA-262 描述了一组保留的**关键字**，这些关键字有特殊用途，比如表示控制语句的开始和结束，或者执行特定的操作。按照规定，保留的关键字不能用作标识符或属性名。ECMA-262 第 6 版规定的所有关键字如下：

```
break       do          in          typeof
case        else        instanceof  var
catch       export      new         void
class       extends     return      while
const       finally     super       with
continue    for         switch      yield
debugger    function    this
default     if          throw
delete      import      try
```

规范中也描述了一组**未来的保留字**，同样不能用作标识符或属性名。虽然保留字在语言中没有特定用途，但它们是保留给将来做关键字用的。

以下是 ECMA-262 第 6 版为将来保留的所有词汇。

始终保留：

```
enum
```

严格模式下保留：

```
implements  package     public
interface   protected   static
let         private
```

模块代码中保留：

```
await
```

这些词汇不能用作标识符，但现在还可以用作对象的属性名。一般来说，最好还是不要使用关键字和保留字作为标识符和属性名，以确保兼容过去和未来的 ECMAScript 版本。

3.3　变量

ECMAScript 变量是松散类型的，意思是变量可以用于保存任何类型的数据。每个变量只不过是一个用于保存任意值的命名占位符。有 3 个关键字可以声明变量：var、const 和 let。其中，var 在 ECMAScript 的所有版本中都可以使用，而 const 和 let 只能在 ECMAScript 6 及更晚的版本中使用。

3.3.1　var 关键字

要定义变量，可以使用 var 操作符（注意 var 是一个关键字），后跟变量名（即标识符，如前所述）：

```
var message;
```

这行代码定义了一个名为 message 的变量，可以用它保存任何类型的值。（不初始化的情况下，变量会保存一个特殊值 undefined，下一节讨论数据类型时会谈到。）ECMAScript 实现变量初始化，因此可以同时定义变量并设置它的值：

```
var message = "hi";
```

这里，message 被定义为一个保存字符串值 hi 的变量。像这样初始化变量不会将它标识为字符串类型，只是一个简单的赋值而已。随后，不仅可以改变保存的值，也可以改变值的类型：

```
var message = "hi";
message = 100;  // 合法，但不推荐
```

在这个例子中，变量 message 首先被定义为一个保存字符串值 hi 的变量，然后又被重写为保存了数值 100。虽然不推荐改变变量保存值的类型，但这在 ECMAScript 中是完全有效的。

1. var 声明作用域

关键的问题在于，使用 var 操作符定义的变量会成为包含它的函数的局部变量。比如，使用 var 在一个函数内部定义一个变量，就意味着该变量将在函数退出时被销毁：

```
function test() {
  var message = "hi"; // 局部变量
}
test();
console.log(message); // 出错!
```

这里，message 变量是在函数内部使用 var 定义的。函数叫 test()，调用它会创建这个变量并给它赋值。调用之后变量随即被销毁，因此示例中的最后一行会导致错误。不过，在函数内定义变量时省略 var 操作符，可以创建一个全局变量：

```
function test() {
  message = "hi";     // 全局变量
}
test();
console.log(message); // "hi"
```

去掉之前的 var 操作符之后，message 就变成了全局变量。只要调用一次函数 test()，就会定义这个变量，并且可以在函数外部访问到。

> **注意** 虽然可以通过省略 var 操作符定义全局变量，但不推荐这么做。在局部作用域中定义的全局变量很难维护，也会造成困惑。这是因为不能一下子断定省略 var 是不是有意而为之。在严格模式下，如果像这样给未声明的变量赋值，则会导致抛出 ReferenceError。

如果需要定义多个变量，可以在一条语句中用逗号分隔每个变量（及可选的初始化）：

```
var message = "hi",
    found = false,
    age = 29;
```

这里定义并初始化了 3 个变量。因为 ECMAScript 是松散类型的，所以使用不同数据类型初始化的变量可以用一条语句来声明。插入换行和空格缩进并不是必需的，但这样有利于阅读理解。

在严格模式下，不能定义名为 eval 和 arguments 的变量，否则会导致语法错误。

2. var 声明提升

使用 var 时，下面的代码不会报错。这是因为使用这个关键字声明的变量会自动提升到函数作用域顶部：

```
function foo() {
  console.log(age);
  var age = 26;
}
foo();  // undefined
```

之所以不会报错，是因为 ECMAScript 运行时把它看成等价于如下代码：

```
function foo() {
  var age;
  console.log(age);
  age = 26;
}
foo();  // undefined
```

这就是所谓的“提升”（hoist），也就是把所有变量声明都拉到函数作用域的顶部。此外，反复多次使用 var 声明同一个变量也没有问题：

```
function foo() {
  var age = 16;
  var age = 26;
  var age = 36;
  console.log(age);
}
foo();  // 36
```

3.3.2 let 声明

let 跟 var 的作用差不多，但有着非常重要的区别。最明显的区别是，let 声明的范围是块作用域，而 var 声明的范围是函数作用域。

```
if (true) {
  var name = 'Matt';
  console.log(name); // Matt
}
console.log(name);   // Matt
```