1.2.2　DOM

文档对象模型（DOM，Document Object Model）是一个应用编程接口（API），用于在 HTML 中使用扩展的 XML。DOM 将整个页面抽象为一组分层节点。HTML 或 XML 页面的每个组成部分都是一种节点，包含不同的数据。比如下面的 HTML 页面：

```
<html>
    <head>
        <title>Sample Page</title>
    </head>
    <body>
        <p> Hello World!</p>
    </body>
</html>
```

这些代码通过 DOM 可以表示为一组分层节点，如图 1-2 所示。

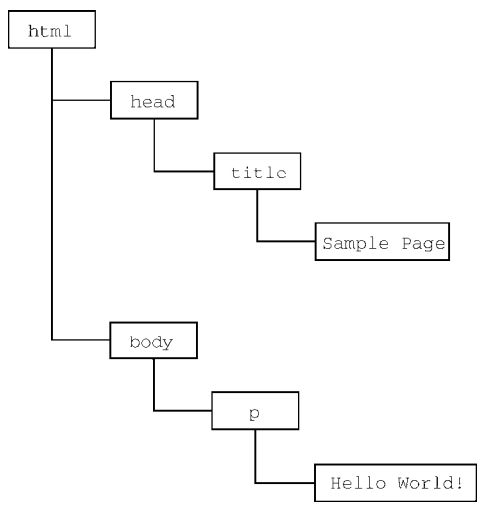

图　1-2

DOM 通过创建表示文档的树，让开发者可以随心所欲地控制网页的内容和结构。使用 DOM API，可以轻松地删除、添加、替换、修改节点。

1. 为什么 DOM 是必需的

在 IE4 和 Netscape Navigator 4 支持不同形式的动态 HTML（DHTML）的情况下，开发者首先可以做到不刷新页面而修改页面外观和内容。这代表了 Web 技术的一个巨大进步，但也暴露了很大的问题。由于网景和微软采用不同思路开发 DHTML，开发者写一个 HTML 页面就可以在任何浏览器中运行的好日子就此终结。

为了保持 Web 跨平台的本性，必须要做点什么。人们担心如果无法控制网景和微软各行其是，那么 Web 就会发生分裂，导致人们面向浏览器开发网页。就在这时，万维网联盟（W3C，World Wide Web Consortium）开始了制定 DOM 标准的进程。

2. DOM 级别

1998 年 10 月，DOM Level 1 成为 W3C 的推荐标准。这个规范由两个模块组成：DOM Core 和 DOM

HTML。前者提供了一种映射 XML 文档，从而方便访问和操作文档任意部分的方式；后者扩展了前者，并增加了特定于 HTML 的对象和方法。

> **注意** DOM 并非只能通过 JavaScript 访问，而且确实被其他很多语言实现了。不过对于浏览器来说，DOM 就是使用 ECMAScript 实现的，如今已经成为 JavaScript 语言的一大组成部分。

DOM Level 1 的目标是映射文档结构，而 DOM Level 2 的目标则宽泛得多。这个对最初 DOM 的扩展增加了对（DHTML 早就支持的）鼠标和用户界面事件、范围、遍历（迭代 DOM 节点的方法）的支持，而且通过对象接口支持了层叠样式表（CSS）。另外，DOM Level 1 中的 DOM Core 也被扩展以包含对 XML 命名空间的支持。

DOM Level 2 新增了以下模块，以支持新的接口。

- **DOM 视图**：描述追踪文档不同视图（如应用 CSS 样式前后的文档）的接口。
- **DOM 事件**：描述事件及事件处理的接口。
- **DOM 样式**：描述处理元素 CSS 样式的接口。
- **DOM 遍历和范围**：描述遍历和操作 DOM 树的接口。

DOM Level 3 进一步扩展了 DOM，增加了以统一的方式加载和保存文档的方法（包含在一个叫 DOM Load and Save 的新模块中），还有验证文档的方法（DOM Validation）。在 Level 3 中，DOM Core 经过扩展支持了所有 XML 1.0 的特性，包括 XML Infoset、XPath 和 XML Base。

目前，W3C 不再按照 Level 来维护 DOM 了，而是作为 DOM Living Standard 来维护，其快照称为 DOM4。DOM4 新增的内容包括替代 Mutation Events 的 Mutation Observers。

> **注意** 在阅读关于 DOM 的资料时，你可能会看到 DOM Level 0 的说法。注意，并没有一个标准叫“DOM Level 0”，这只是 DOM 历史中的一个参照点。DOM Level 0 可以看作 IE4 和 Netscape Navigator 4 中最初支持的 DHTML。

3. 其他 DOM

除了 DOM Core 和 DOM HTML 接口，有些其他语言也发布了自己的 DOM 标准。下面列出的语言是基于 XML 的，每一种都增加了该语言独有的 DOM 方法和接口：

- 可伸缩矢量图（SVG，Scalable Vector Graphics）
- 数学标记语言（MathML，Mathematical Markup Language）
- 同步多媒体集成语言（SMIL，Synchronized Multimedia Integration Language）

此外，还有一些语言开发了自己的 DOM 实现，比如 Mozilla 的 XML 用户界面语言（XUL，XML User Interface Language）。不过，只有前面列表中的语言是 W3C 推荐标准。

4. Web 浏览器对 DOM 的支持情况

DOM 标准在 Web 浏览器实现它之前就已经作为标准发布了。IE 在第 5 版中尝试支持 DOM，但直到 5.5 版才开始真正支持，该版本实现了 DOM Level 1 的大部分。IE 在第 6 版和第 7 版中都没有实现新特性，第 8 版中修复了一些问题。

网景在 Netscape 6（Mozilla 0.6.0）之前都不支持 DOM。Netscape 7 之后，Mozilla 把开发资源转移

到开发 Firefox 浏览器上。Firefox 3+支持全部的 Level 1、几乎全部的 Level 2，以及 Level 3 的某些部分。（Mozilla 开发团队的目标是打造百分之百兼容标准的浏览器，他们的工作也得到了应有的回报。）

支持 DOM 是浏览器厂商的重中之重，每个版本发布都会改进支持度。下表展示了主流浏览器支持 DOM 的情况。

浏　览　器	DOM 兼容
Netscape Navigator 1~4.*x*	—
Netscape 6+（Mozilla 0.6.0+）	Level 1、Level 2（几乎全部）、Level 3（部分）
IE2~4.*x*	—
IE5	Level 1（很少）
IE5.5~8	Level 1（几乎全部）
IE9+	Level 1、Level 2、Level 3
Edge	Level 1、Level 2、Level 3
Opera 1~6	—
Opera 7~8.*x*	Level 1（几乎全部）、Level 2（部分）
Opera 9~9.9	Level 1、Level 2（几乎全部）、Level 3（部分）
Opera 10+	Level 1、Level 2、Level 3（部分）
Safari 1.0.*x*	Level 1
Safari 2+	Level 1、Level 2（部分）、Level 3（部分）
iOS Safari 3.2+	Level 1、Level 2（部分）、Level 3（部分）
Chrome 1+	Level 1、Level 2（部分）、Level 3（部分）
Firefox 1+	Level 1、Level 2（几乎全部）、Level 3（部分）

> **注意**　上表中兼容性的状态会随时间而变化，其中的内容仅反映本书写作时的状态。

1.2.3　BOM

IE3 和 Netscape Navigator 3 提供了**浏览器对象模型**（BOM）API，用于支持访问和操作浏览器的窗口。使用 BOM，开发者可以操控浏览器显示页面之外的部分。而 BOM 真正独一无二的地方，当然也是问题最多的地方，就是它是唯一一个没有相关标准的 JavaScript 实现。HTML5 改变了这个局面，这个版本的 HTML 以正式规范的形式涵盖了尽可能多的 BOM 特性。由于 HTML5 的出现，之前很多与 BOM 有关的问题都迎刃而解了。

总体来说，BOM 主要针对浏览器窗口和子窗口（frame），不过人们通常会把任何特定于浏览器的扩展都归在 BOM 的范畴内。比如，下面就是这样一些扩展：

- 弹出新浏览器窗口的能力；
- 移动、缩放和关闭浏览器窗口的能力；
- `navigator` 对象，提供关于浏览器的详尽信息；
- `location` 对象，提供浏览器加载页面的详尽信息；

- screen 对象，提供关于用户屏幕分辨率的详尽信息；
- performance 对象，提供浏览器内存占用、导航行为和时间统计的详尽信息；
- 对 cookie 的支持；
- 其他自定义对象，如 XMLHttpRequest 和 IE 的 ActiveXObject。

因为在很长时间内都没有标准，所以每个浏览器实现的都是自己的 BOM。有一些所谓的事实标准，比如对于 window 对象和 navigator 对象，每个浏览器都会给它们定义自己的属性和方法。现在有了 HTML5，BOM 的实现细节应该会日趋一致。关于 BOM，本书会在第 12 章再专门详细介绍。

1.3 JavaScript 版本

作为网景的继承者，Mozilla 是唯一仍在延续最初 JavaScript 版本编号的浏览器厂商。当初网景在将其源代码开源时（项目名为 Mozilla Project），JavaScript 在其浏览器中最后的版本是 1.3。（前面提到过，1.4 版是专门为服务器实现的。）因为 Mozilla Foundation 在持续开发 JavaScript，为它增加新特性、关键字和语法，所以 JavaScript 的版本号也在不断递增。下表展示了 Netscape/Mozilla 浏览器发布的历代 JavaScript 版本。

浏 览 器	JavaScript 版本
Netscape Navigator 2	1.0
Netscape Navigator 3	1.1
Netscape Navigator 4	1.2
Netscape Navigator 4.06	1.3
Netscape 6+（Mozilla 0.6.0+）	1.5
Firefox 1	1.5
Firefox 1.5	1.6
Firefox 2	1.7
Firefox 3	1.8
Firefox 3.5	1.8.1
Firefox 3.6	1.8.2
Firefox 4	1.8.5

这种版本编号方式是根据 Firefox 4 要发布 JavaScript 2.0 决定的，在此之前版本号的每次递增，反映的是 JavaScript 实现逐渐接近 2.0 建议。虽然这是最初的计划，但 JavaScript 的发展让这个计划变得不可能。JavaScript 2.0 作为一个目标已经不存在了，而这种版本号编排方式在 Firefox 4 发布后就终止了。

> **注意** Netscape/Mozilla 仍然沿用这种版本方案。而 IE 的 JScript 有不同的版本号规则。这些 JScript 版本与上表提到的 JavaScript 版本并不对应。此外，多数浏览器对 JavaScript 的支持，指的是实现 ECMAScript 和 DOM 的程度。

1.4　小结

JavaScript 是一门用来与网页交互的脚本语言，包含以下三个组成部分。

- ECMAScript：由 ECMA-262 定义并提供核心功能。
- 文档对象模型（DOM）：提供与网页内容交互的方法和接口。
- 浏览器对象模型（BOM）：提供与浏览器交互的方法和接口。

JavaScript 的这三个部分得到了五大 Web 浏览器（IE、Firefox、Chrome、Safari 和 Opera）不同程度的支持。所有浏览器基本上对 ES5（ECMAScript 5）提供了完善的支持，而对 ES6（ECMAScript 6）和 ES7（ECMAScript 7）的支持度也在不断提升。这些浏览器对 DOM 的支持各不相同，但对 Level 3 的支持日益趋于规范。HTML5 中收录的 BOM 会因浏览器而异，不过开发者仍然可以假定存在很大一部分公共特性。

第2章 HTML中的JavaScript

本章内容

- 使用<script>元素
- 行内脚本与外部脚本的比较
- 文档模式对 JavaScript 有什么影响
- 确保 JavaScript 不可用时的用户体验

将 JavaScript 引入网页，首先要解决它与网页的主导语言 HTML 的关系问题。在 JavaScript 早期，网景公司的工作人员希望在将 JavaScript 引入 HTML 页面的同时，不会导致页面在其他浏览器中渲染出问题。通过反复试错和讨论，他们最终做出了一些决定，并达成了向网页中引入通用脚本能力的共识。当初他们的很多工作得到了保留，并且最终形成了 HTML 规范。

2.1 <script>元素

将 JavaScript 插入 HTML 的主要方法是使用<script>元素。这个元素是由网景公司创造出来，并最早在 Netscape Navigator 2 中实现的。后来，这个元素被正式加入到 HTML 规范。<script>元素有下列 8 个属性。

- async：可选。表示应该立即开始下载脚本，但不能阻止其他页面动作，比如下载资源或等待其他脚本加载。只对外部脚本文件有效。
- charset：可选。使用 src 属性指定的代码字符集。这个属性很少使用，因为大多数浏览器不在乎它的值。
- crossorigin：可选。配置相关请求的 CORS（跨源资源共享）设置。默认不使用 CORS。crossorigin="anonymous"配置文件请求不必设置凭据标志。crossorigin="use-credentials"设置凭据标志，意味着出站请求会包含凭据。
- defer：可选。表示脚本可以延迟到文档完全被解析和显示之后再执行。只对外部脚本文件有效。在 IE7 及更早的版本中，对行内脚本也可以指定这个属性。
- integrity：可选。允许比对接收到的资源和指定的加密签名以验证子资源完整性（SRI，Subresource Integrity）。如果接收到的资源的签名与这个属性指定的签名不匹配，则页面会报错，脚本不会执行。这个属性可以用于确保内容分发网络（CDN，Content Delivery Network）不会提供恶意内容。
- language：废弃。最初用于表示代码块中的脚本语言（如"JavaScript"、"JavaScript 1.2"或"VBScript"）。大多数浏览器都会忽略这个属性，不应该再使用它。
- src：可选。表示包含要执行的代码的外部文件。

- type：可选。代替 language，表示代码块中脚本语言的内容类型（也称 MIME 类型）。按照惯例，这个值始终都是"text/javascript"，尽管"text/javascript"和"text/ecmascript"都已经废弃了。JavaScript 文件的 MIME 类型通常是"application/x-javascript"，不过给 type 属性这个值有可能导致脚本被忽略。在非 IE 的浏览器中有效的其他值还有"application/javascript"和"application/ecmascript"。如果这个值是 module，则代码会被当成 ES6 模块，而且只有这时候代码中才能出现 import 和 export 关键字。

使用<script>的方式有两种：通过它直接在网页中嵌入 JavaScript 代码，以及通过它在网页中包含外部 JavaScript 文件。

要嵌入行内 JavaScript 代码，直接把代码放在<script>元素中就行：

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

包含在<script>内的代码会被从上到下解释。在上面的例子中，被解释的是一个函数定义，并且该函数会被保存在解释器环境中。在<script>元素中的代码被计算完成之前，页面的其余内容不会被加载，也不会被显示。

在使用行内 JavaScript 代码时，要注意代码中不能出现字符串</script>。比如，下面的代码会导致浏览器报错：

```
<script>
  function sayScript() {
    console.log("</script>");
  }
</script>
```

浏览器解析行内脚本的方式决定了它在看到字符串</script>时，会将其当成结束的</script>标签。想避免这个问题，只需要转义字符“\”[①]即可：

```
<script>
  function sayScript() {
    console.log("<\/script>");
  }
</script>
```

这样修改之后，代码就可以被浏览器完全解释，不会导致任何错误。

要包含外部文件中的 JavaScript，就必须使用 src 属性。这个属性的值是一个 URL，指向包含 JavaScript 代码的文件，比如：

```
<script src="example.js"></script>
```

这个例子在页面中加载了一个名为 example.js 的外部文件。文件本身只需包含要放在<script>的起始及结束标签中间的 JavaScript 代码。与解释行内 JavaScript 一样，在解释外部 JavaScript 文件时，页面也会阻塞。（阻塞时间也包含下载文件的时间。）在 XHTML 文档中，可以忽略结束标签，比如：

```
<script src="example.js"/>
```

以上语法不能在 HTML 文件中使用，因为它是无效的 HTML，有些浏览器不能正常处理，比如 IE。

① 此处的转义字符指在 JavaScript 中使用反斜杠“\”来向文本字符串添加特殊字符。——编者注

注意 按照惯例，外部 JavaScript 文件的扩展名是.js。这不是必需的，因为浏览器不会检查所包含 JavaScript 文件的扩展名。这就为使用服务器端脚本语言动态生成 JavaScript 代码，或者在浏览器中将 JavaScript 扩展语言（如 TypeScript，或 React 的 JSX）转译为 JavaScript 提供了可能性。不过要注意，服务器经常会根据文件扩展来确定响应的正确 MIME 类型。如果不打算使用.js 扩展名，一定要确保服务器能返回正确的 MIME 类型。

另外，使用了 src 属性的<script>元素不应该再在<script>和</script>标签中再包含其他 JavaScript 代码。如果两者都提供的话，则浏览器只会下载并执行脚本文件，从而忽略行内代码。

<script>元素的一个最为强大、同时也备受争议的特性是，它可以包含来自外部域的 JavaScript 文件。跟<img>元素很像，<script>元素的 src 属性可以是一个完整的 URL，而且这个 URL 指向的资源可以跟包含它的 HTML 页面不在同一个域中，比如这个例子：

```
<script src="http://www.somewhere.com/afile.js"></script>
```

浏览器在解析这个资源时，会向 src 属性指定的路径发送一个 GET 请求，以取得相应资源，假定是一个 JavaScript 文件。这个初始的请求不受浏览器同源策略限制，但返回并被执行的 JavaScript 则受限制。当然，这个请求仍然受父页面 HTTP/HTTPS 协议的限制。

来自外部域的代码会被当成加载它的页面的一部分来加载和解释。这个能力可以让我们通过不同的域分发 JavaScript。不过，引用了放在别人服务器上的 JavaScript 文件时要格外小心，因为恶意的程序员随时可能替换这个文件。在包含外部域的 JavaScript 文件时，要确保该域是自己所有的，或者该域是一个可信的来源。<script>标签的 integrity 属性是防范这种问题的一个武器，但这个属性也不是所有浏览器都支持。

不管包含的是什么代码，浏览器都会按照<script>在页面中出现的顺序依次解释它们，前提是它们没有使用 defer 和 async 属性。第二个<script>元素的代码必须在第一个<script>元素的代码解释完毕才能开始解释，第三个则必须等第二个解释完，以此类推。

2.1.1 标签位置

过去，所有<script>元素都被放在页面的<head>标签内，如下面的例子所示：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  <script src="example1.js"></script>
  <script src="example2.js"></script>
  </head>
  <body>
  <!-- 这里是页面内容 -->
  </body>
</html>
```

这种做法的主要目的是把外部的 CSS 和 JavaScript 文件都集中放到一起。不过，把所有 JavaScript 文件都放在<head>里，也就意味着必须把所有 JavaScript 代码都下载、解析和解释完成后，才能开始渲染页面（页面在浏览器解析到<body>的起始标签时开始渲染）。对于需要很多 JavaScript 的页面，这会导致页面渲染的明显延迟，在此期间浏览器窗口完全空白。为解决这个问题，现代 Web 应用程序通常将所有 JavaScript 引用放在<body>元素中的页面内容后面，如下面的例子所示：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  </head>
  <body>
  <!-- 这里是页面内容 -->
  <script src="example1.js"></script>
  <script src="example2.js"></script>
  </body>
</html>
```

这样一来，页面会在处理 JavaScript 代码之前完全渲染页面。用户会感觉页面加载更快了，因为浏览器显示空白页面的时间短了。

2.1.2　推迟执行脚本

HTML 4.01 为<script>元素定义了一个叫 defer 的属性。这个属性表示脚本在执行的时候不会改变页面的结构。也就是说，脚本会被延迟到整个页面都解析完毕后再运行。因此，在<script>元素上设置 defer 属性，相当于告诉浏览器立即下载，但延迟执行。

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  <script defer src="example1.js"></script>
  <script defer src="example2.js"></script>
  </head>
  <body>
  <!-- 这里是页面内容 -->
  </body>
</html>
```

虽然这个例子中的<script>元素包含在页面的<head>中，但它们会在浏览器解析到结束的</html>标签后才会执行。HTML5 规范要求脚本应该按照它们出现的顺序执行，因此第一个推迟的脚本会在第二个推迟的脚本之前执行，而且两者都会在 DOMContentLoaded 事件之前执行（关于事件，请参考第 17 章）。不过在实际当中，推迟执行的脚本不一定总会按顺序执行或者在 DOMContentLoaded 事件之前执行，因此最好只包含一个这样的脚本。

如前所述，defer 属性只对外部脚本文件才有效。这是 HTML5 中明确规定的，因此支持 HTML5 的浏览器会忽略行内脚本的 defer 属性。IE4~7 展示出的都是旧的行为，IE8 及更高版本则支持 HTML5 定义的行为。

对 defer 属性的支持是从 IE4、Firefox 3.5、Safari 5 和 Chrome 7 开始的。其他所有浏览器则会忽略这个属性，按照通常的做法来处理脚本。考虑到这一点，还是把要推迟执行的脚本放在页面底部比较好。

> **注意**　对于 XHTML 文档，指定 defer 属性时应该写成 defer="defer"。

2.1.3　异步执行脚本

HTML5 为<script>元素定义了 async 属性。从改变脚本处理方式上看，async 属性与 defer 类

似。当然，它们两者也都只适用于外部脚本，都会告诉浏览器立即开始下载。不过，与 defer 不同的是，标记为 async 的脚本并不保证能按照它们出现的次序执行，比如：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  <script async src="example1.js"></script>
  <script async src="example2.js"></script>
  </head>
  <body>
  <!-- 这里是页面内容 -->
  </body>
</html>
```

在这个例子中，第二个脚本可能先于第一个脚本执行。因此，重点在于它们之间没有依赖关系。给脚本添加 async 属性的目的是告诉浏览器，不必等脚本下载和执行完后再加载页面，同样也不必等到该异步脚本下载和执行后再加载其他脚本。正因为如此，异步脚本不应该在加载期间修改 DOM。

异步脚本保证会在页面的 load 事件前执行，但可能会在 DOMContentLoaded（参见第 17 章）之前或之后。Firefox 3.6、Safari 5 和 Chrome 7 支持异步脚本。使用 async 也会告诉页面你不会使用 document.write，不过好的 Web 开发实践根本就不推荐使用这个方法。

> **注意** 对于 XHTML 文档，指定 async 属性时应该写成 async="async"。

2.1.4 动态加载脚本

除了<script>标签，还有其他方式可以加载脚本。因为 JavaScript 可以使用 DOM API，所以通过向 DOM 中动态添加 script 元素同样可以加载指定的脚本。只要创建一个 script 元素并将其添加到 DOM 即可。

```
let script = document.createElement('script');
script.src = 'gibberish.js';
document.head.appendChild(script);
```

当然，在把 HTMLElement 元素添加到 DOM 且执行到这段代码之前不会发送请求。默认情况下，以这种方式创建的<script>元素是以异步方式加载的，相当于添加了 async 属性。不过这样做可能会有问题，因为所有浏览器都支持 createElement()方法，但不是所有浏览器都支持 async 属性。因此，如果要统一动态脚本的加载行为，可以明确将其设置为同步加载：

```
let script = document.createElement('script');
script.src = 'gibberish.js';
script.async = false;
document.head.appendChild(script);
```

以这种方式获取的资源对浏览器预加载器是不可见的。这会严重影响它们在资源获取队列中的优先级。根据应用程序的工作方式以及怎么使用，这种方式可能会严重影响性能。要想让预加载器知道这些动态请求文件的存在，可以在文档头部显式声明它们：

```
<link rel="preload" href="gibberish.js">
```