汕学派JS学习 第一第二章

第一章：

JavaScript 是一种专门为网络交互而设计的脚本语言，由下列三个不同的部分组成：

虽然 JavaScript 和 ECMAScript 通常都被人们用来表达相同的含义，但 JavaScript 的含义却比 ECMA-262 中规定的要多得多。没错，一个完整的 JavaScript 实现应该由下列三个不同的部分组成（见图 1-1）。

- 核心（ECMAScript）
- 文档对象模型（DOM）
- 浏览器对象模型（BOM）

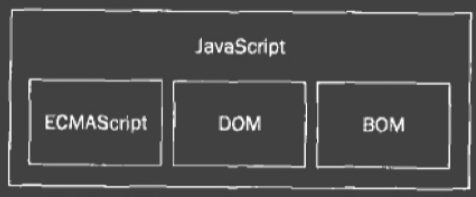

图 1-1

1.ECMAScript 由ECMA-262定义，提供核心语言功能
2.文档对象模型（DOM），提供访问和操作网页内容的方法和接口

1.2.2 文档对象模型（DOM）

文档对象模型（DOM，Document Object Model）是针对 XML 但经过扩展用于 HTML 的应用程序编程接口（API，Application Programming Interface）。DOM 把整个页面映射为一个多层节点结构。HTML 或 XML 页面中的每个组成部分都是某种类型的节点，这些节点又包含着不同类型的数据。看下面这个 HTML 页面：

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

在 DOM 中，这个页面可以通过见图 1-2 所示的分层节点图表示。

通过 DOM 创建的这个表示文档的树形图，开发人员获得了控制页面内容和结构的主动权。借助 DOM 提供的 API，开发人员可以轻松自如地删除、添加、替换或修改任何节点。

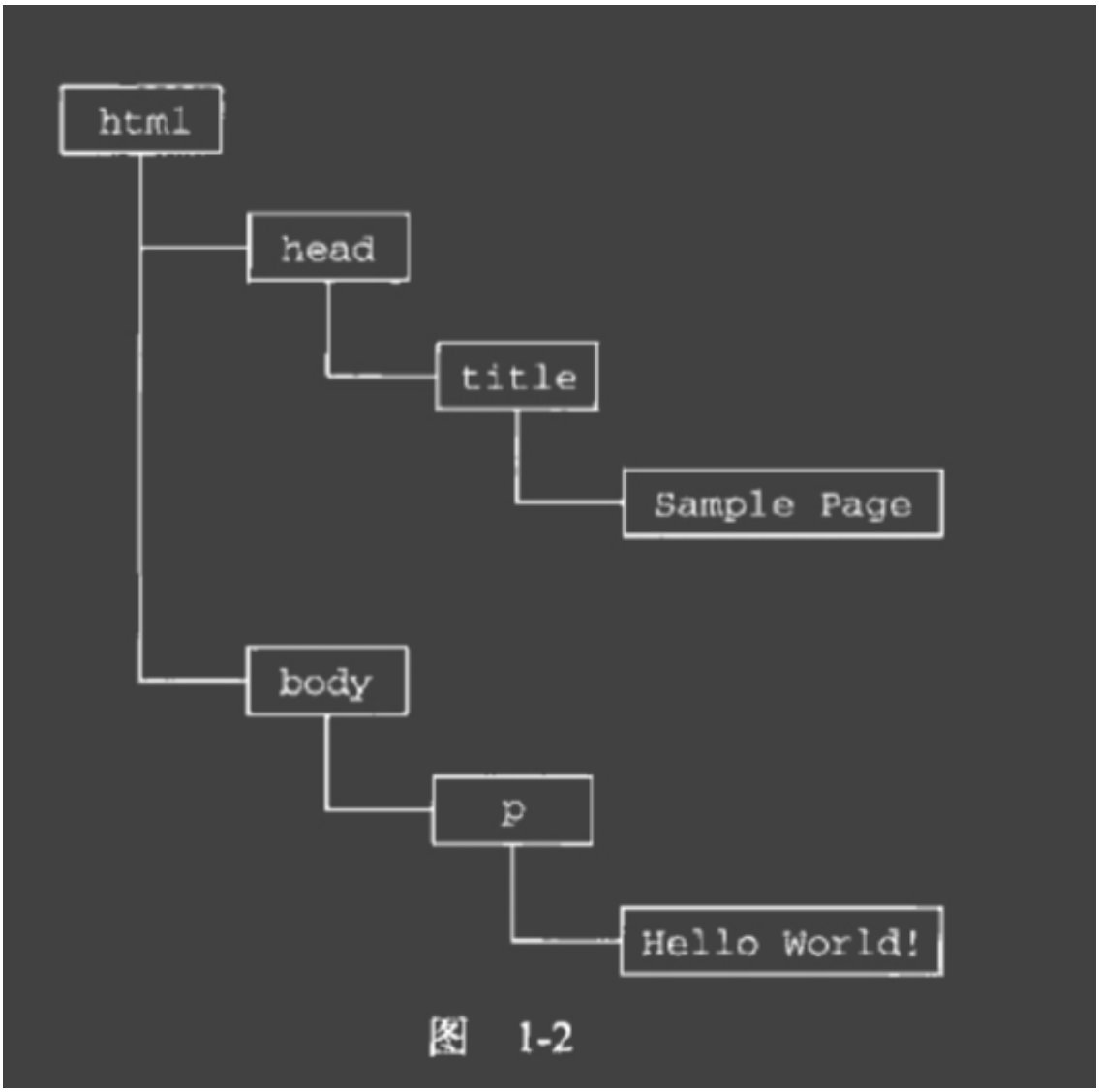

图 1-2

3.浏览器对象模型（BOM），提供与浏览器交互的方法和接口

1.2.3 浏览器对象模型（BOM）

Internet Explorer 3 和 Netscape Navigator 3 有一个共同的特色，那就是支持可以访问和操作浏览器窗口的浏览器对象模型（BOM，Browser Object Model）。开发人员使用 BOM 可以控制浏览器显示的页面以外的部分。而 BOM 真正与众不同的地方（也是经常会导致问题的地方），还是它作为 JavaScript 实现的一部分但却没有相关的标准。这个问题在 HTML5 中得到了解决，HTML5 致力于把很多 BOM 功能写入正式规范。HTML5 发布后，很多关于 BOM 的困惑烟消云散。

从根本上讲，BOM 只处理浏览器窗口和框架；但人们习惯上也把所有针对浏览器的 JavaScript 扩展算作 BOM 的一部分。下面就是一些这样的扩展：

- 弹出新浏览器窗口的功能；
- 移动、缩放和关闭浏览器窗口的功能；
- 提供浏览器详细信息的 navigator 对象；
- 提供浏览器所加载页面的详细信息的 location 对象；
- 提供用户显示器分辨率详细信息的 screen 对象；
- 对 cookies 的支持；
- 像 XMLHttpRequest 和 IE 的 ActiveXObject 这样的自定义对象。

由于没有 BOM 标准可以遵循，因此每个浏览器都有自己的实现。虽然也存在一些事实标准，例如要有 window 对象和 navigator 对象等，但每个浏览器都会为这两个对象乃至其他对象定义自己的属性和方法。现在有了 HTML5，BOM 实现的细节有望朝着兼容性越来越高的方向发展。第 8 章将深入讨论 BOM。

第二章：

把JS插入到html页面中需要使用`<script>`元素。使用该元素可以把JS嵌入到html页面中。
注意：

2.1.1 标签的位置

按照惯例，所有<script>元素都应该放在页面的<head>元素中，例如：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" src="example1.js"></script>
    <script type="text/javascript" src="example2.js"></script>
  </head>
  <body>
    <!-- 这里放内容 -->
  </body>
</html>
```

这种做法的目的就是把所有外部文件（包括 CSS 文件和 JavaScript 文件）的引用都放在相同的地方。可是，在文档的<head>元素中包含所有 JavaScript 文件，意味着必须等到全部 JavaScript 代码都被下载、解析和执行完成以后，才能开始呈现页面的内容（浏览器在遇到<body>标签时才开始呈现内容）。对于那些需要很多 JavaScript 代码的页面来说，这无疑会导致浏览器在呈现页面时出现明显的延迟，而延迟期间的浏览器窗口中将是一片空白。为了避免这个问题，现代 Web 应用程序一般都把全部 JavaScript 引用放在<body>元素中页面的内容后面，如下例所示：

1.在包含外部js文件时，必须将SRC属性设置为指向相应文件的URL。这个文件既可以是包含它的页面位于同一个服务器上的文件，也可以是其他任何域中的文件。
2.所有`<script>`元素都会按照他们在页面中出现的先后顺序依次被解析。在不使用defer和async属性的情况下，只有在解析完前面`<script>`元素中的代码之后，才会解析后面`<script>`元素中的代码。
3.由于浏览器会先解析不是用defer属性的`<script>`元素中的代码，然后再解析后面的内容。所以一般应该把`<script>`元素放在页面最后，`</body>`标签前面。

2.1.2 延迟脚本

HTML 4.01 为<script>标签定义了 defer 属性。这个属性的用途是表明脚本在执行时不会影响页面的构造。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>元素中设置 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 type="text/javascript" defer="defer" src="example1.js"></script>
    <script type="text/javascript" defer="defer" src="example2.js"></script>
  </head>
  <body>
    <!-- 这里放内容 -->
  </body>
</html>
```

在这个例子中，虽然我们把<script>元素放在了文档的<head>元素中，但其中包含的脚本将延迟到浏览器遇到</html>标签后再执行。HTML5 规范要求脚本按照它们出现的先后顺序执行，因此第一个延迟脚本会先于第二个延迟脚本执行，而这两个脚本会先于 DOMContentLoaded 事件（详见第 13 章）执行。在现实当中，延迟脚本并不一定会按照顺序执行，也不一定会在 DOMContentLoaded 事件触发前执行，因此最好只包含一个延迟脚本。

前面提到过，defer 属性只适用于外部脚本文件。这一点在 HTML5 中已经明确规定，因此支持 HTML5 的实现会忽略给嵌入脚本设置的 defer 属性。IE4～IE7 还支持对嵌入脚本的 defer 属性，但 IE8 及之后版本则完全支持 HTML5 规定的行为。

IE4、Firefox 3.5、Safari 5 和 Chrome 是最早支持 defer 属性的浏览器。其他浏览器会忽略这个属性，像平常一样处理脚本。为此，把延迟脚本放在页面底部仍然是最佳选择。

在 XHTML 文档中，要把 defer 属性设置为 defer="defer"。

4.使用defer属性可以让脚本在文档中完全呈现之后再执行，延迟脚本总是按照指定他们的顺序执行。

5.使用async属性可以表示当前脚本不必再等待其他脚本，也不必阻塞文档呈现。不能保证异步脚本按照它们在页面中出现的顺序执行。

2.1.3 异步脚本

HTML5 为<script>元素定义了 async 属性。这个属性与 defer 属性类似，都用于改变处理脚本

的行为。同样与 defer 类似，async 只适用于外部脚本文件，并告诉浏览器立即下载文件。但与 defer 不同的是，标记为 async 的脚本并不保证按照指定它们的先后顺序执行。例如：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" async src="example1.js"></script>
    <script type="text/javascript" async src="example2.js"></script>
  </head>
  <body>
    <!-- 这里放内容 -->
  </body>
</html>
```

在以上代码中，第二个脚本文件可能会在第一个脚本文件之前执行。因此，确保两者之间互不依赖非常重要。指定 async 属性的目的是不让页面等待两个脚本下载和执行，从而异步加载页面其他内容。为此，建议异步脚本不要在加载期间修改 DOM。

异步脚本一定会在页面的 load 事件前执行，但可能会在 DOMContentLoaded 事件触发之前或之后执行。支持异步脚本的浏览器有 Firefox 3.6、Safari 5 和 Chrome。

在 XHTML 文档中，要把 async 属性设置为 async="async"。

6.使用 `<noscript>` 元素可以指定在不支持脚本的浏览器中显示替代内容，但在启用了脚本的情况下，浏览器不会显示 `<noscript>` 元素中的任何内容。

2.4 <noscript>元素

早期浏览器都面临一个特殊的问题，即当浏览器不支持 JavaScript 时如何让页面平稳地退化。对这个问题的最终解决方案就是创造一个<noscript>元素，用以在不支持 JavaScript 的浏览器中显示替代的内容。这个元素可以包含能够出现在文档<body>中的任何 HTML 元素——<script>元素除外。包含在<noscript>元素中的内容只有在下列情况下才会显示出来：

- 浏览器不支持脚本；
- 浏览器支持脚本，但脚本被禁用。

符合上述任何一个条件，浏览器都会显示<noscript>中的内容。而在除此之外的其他情况下，浏览器不会呈现<noscript>中的内容。

请看下面这个简单的例子：

```
<html>
  <head>
    <title>Example HTML Page</title>
    <script type="text/javascript" defer="defer" src="example1.js"></script>
    <script type="text/javascript" defer="defer" src="example2.js"></script>
  </head>
  <body>
    <noscript>
      <p>本页面需要浏览器支持（启用）JavaScript。
    </noscript>
  </body>
</html>
```

这个页面会在脚本无效的情况下向用户显示一条消息。而在启用了脚本的浏览器中，用户永远也不会看到它——尽管它是页面的一部分。