CHAPTER

第 1 章

十年 JavaScript

几乎每本讲 JavaScript的书都会用很多的篇幅讲 JavaScript的源起与现状。本书也需要这样吗？

不。我虽然也这样想过，但我不打算让读者去读一些能够从 Wiki中摘抄出来的文字，或者在很多书籍中都可以看到的、千篇一律的文字。所以，我来写写我与 JavaScript的故事。在这个过程中，你会看到一个开发者在每个阶段对 JavaScript的认识，同时可以知道这本书的由来。

当然，一个人的历史，在一门语言的历史面前显得是那样的不足以道。因此除非编好故事性内容，对本章的前 3个小节，你也可以选择跳过去。

1.1 网页中的代码

1.1.1 新鲜的玩意儿

1996 年末，公司老板 P&J 找我去给他的一个朋友帮忙，做一些网页。那时事实上还没有说要做成网站。在那个时代，中国的 IT 人中可能还有 2/3 的触网者在玩一种叫“电子公告板（Bulletin Board System，BBS）”的东西——这与现在的 BBS 很不一样，它是一种利用现有电话网组成的 PC-BBS 系统，使用基于 Telnet 的终端登入操作。而另外 1/3 的触网者可能已经开始了互联网之旅，知道了像主页（Home Page）、超链接（Hyper Link）这样的一些东西。

我最开始做的网页只用于展示信息，是一个个单纯的、静态的网页，并通过一些超链接连接起来。当时网页开发的环境并不好（像现在的 Dreamweaver 这类程序，那时只能是梦想），因此我只能用记事本（notepad.exe）来写 HTML。当时显示这些.htm 文件的浏览器就是 Netscape Navigator 3。

我很快就遇到了麻烦，因为 P&J 的朋友说希望让浏览网页的用户们能做更多的事，例如搜索等操作。我笑着说："如果在电子公告板上，写段脚本就可以了；但在互联网上面，却要做很多的工作。"

事实上我并不知道要做多少的工作。我随后查阅的资料表明：不但要在网页中放一些表单让浏览者提交信息，还要在网站的服务器上写些代码来响应这些提交信息。我向那位先生摊开双手，说："如果你真的想要这样做，那么我们可能需要三个月，或者更久。因为我还必须学习一些新鲜的玩意儿才行。"

那时的触网者，对这些"新鲜的玩意儿"的了解还几乎是零。因此，这个想法很自然地被搁置了。而我后来（1997 年）被调到成都，终于有更多的机会接触 Internet，而且浏览器环境已经换成了 Internet Explorer 4.0。

那是一个美好的时代。通过互联网络，大量的新东西很快被传递进来。我终于有机会了解一些新的技术名词，例如 CSS 和 JavaScript。当时（1997 年 12 月）HTML 4.0 的标准已经确定，浏览器的兼容性开始变得更好，Internet Explorer（以下简称 IE）也越来越有取代 Netscape Navigator（以下简称 NN）而一统天下的趋势。除了这些，我还对在 Delphi 中进行 ISAPI CGI 和 ISAPI Filter 开发的技术也展开了深入的学习。

1.1.2 第一段在网页中的代码

1998 年，我被调回到河南郑州，成为一名专职程序员，任职于当时一家开发反病毒软件的公司，主要工作是用 Delphi 做 Windows 环境下的开发。而当时我的个人兴趣之一，就是"做一个个人网站"。那时大家都对"做主页"很感兴趣，我的老朋友傅贵[1]就专门写了一套代码，以方便普通互联网用户将自己的主页放到"个人空间"里。同时，他还为这些个人用户提供了公共的 BBS 程序和一些其他的服务器端代码。但我并不满足于这些，我满脑子想的是做一个"自己的网站"。我争取到了一台使用 IIS 4.0 的服务器，由于有 ISAPI CGI 这样的服务器端技术，因此一年多前的那个"如何让浏览者提交信息"

[1] 在中国互联网技术论坛的早期，傅贵先生创建了著名的"Delphi/C++ Builder论坛"，他也是"中国开发网 ORG（cndev.org）"的创建者。

的问题已经迎刃而解。而当时更先进的浏览器端开发技术也已经出现，例如 Java Applet。我当时便选择了一个 Java Applet 来做“网页菜单”。

在当时，在 IE 中显示 Java Applet 之前需要装载整个 JVM（Java Virtual Machine，Java 虚拟机）。这对于现在的 CPU 来说，已经不是什么大不了的负担了，但当时这个过程却非常漫长。在这个“漫长的过程”中，网页显示一片空白，因此浏览者可能在看到一个“漂亮的菜单”之前就跑掉了。

为此我不得不像做 Windows 桌面应用程序一样，弄一个“闪屏窗口”放在前面。这个窗口只用于显示“Loading...”这样的文字（或图片）。而同时，我在网页中加入一个<APPLET>标签，使得 JVM 能偷偷地载入到浏览器中。然而，接下来的问题是：这个过程怎么结束呢？

我当时能找到的所有 Java Applet 都没有“在 JVM 载入后自动链接到其他网页”的能力。但其中有一个能力可以支持一种状态查询，它能在一个名为 `isInited` 的属性中返回状态 `True` 或 `False`。

这时，我需要在浏览器中查询到这种状态，如果是 `True`，就可以结束“Loading”过程，进入到真正的主页中去。由于 JVM 已经偷偷地载入过了，因此“漂亮的菜单”就能很快地显示出来。由于我得不到 Java Applet 的 Java 源代码并重写这个 Applet 去切换网址，因此这个“访问 Java Applet 的属性”的功能就需要用一种在浏览器中的技术来实现了。

这时跳到我面前的东西，就是 JavaScript。我为此而写出的代码如下：

```
<script language="JavaScript">
function checkInited() {
  if (document.MsgApplet.isInited) {
    self.location.href = "mainpage.htm";
  }
}
setInterval("checkInited()", 50)
</script>
```

1.1.3 最初的价值

JavaScript 最初被开发人员接受，其实是一种无可奈何的选择。

首先，网景公司（Netscape Communications Corporation）很早就意识到：网络需要一种集成的、统一的、客户端到服务端的解决方案。为此 Netscape 提出了 LiveWire 的概

念[2]，并设计了当时名为 LiveScript 的语言用来在服务器上创建类似于 CGI 的应用程序；与此同时，网景公司也意识到他们的浏览器 NN 中需要一个脚本语言的支持，解决类似于“在向服务器提交数据之前进行验证”的问题。1995 年 4 月，网景公司招募了 Brendan Eich，希望 Brendan Eich 来实现这样的一种语言，实现“使网页活动起来（Making Web Pages Come Alive）”。到了 1995 年 9 月，在发布 NN 2.0 Beta 时，LiveScript 最早作为一种“浏览器上的脚本语言”被推到网页制作人员的面前；随后，在 9 月 18 日，网景公司宣布在其服务器端产品“LiveWire Server Extension Engine”中将包含一个该语言的服务器端（Server-side）版本[3]。

在这时，Sun 公司的 Java 语言大行其道。Netscape 决定在服务器端与 Sun 进行合作，这种合作后来扩展到浏览器，推出了名为 Java Applet 的“小应用”。而 Netscape 也借势将 LiveScript 改名，于 1995 年 12 月 4 日，在与 Sun 公司共同发布声明中首次使用了“JavaScript”这个名字，称之为一种“面向企业网络和互联网的、开放的、跨平台的对象脚本语言”[4]。从这种定位来看，最初的 JavaScript 在一定程度上是为了解决浏览器与服务器之间统一开发而实现的一种语言。

微软在浏览器方面是一个后来者。因此，它不得不在自己的浏览器中加入 JavaScript 的支持。但为了避免冲突，微软使用了“JScript”这个名字。微软在 1996 年 8 月发布 IE 3 时，提供了相当于 NN 3 的 JavaScript 脚本语言支持，但同时也提供了自己的 VBScript。

当 IE 与 NN 进行那场著名的“浏览器大战”的时候，没有人能够看到结局。因此要想做一个“可以看的网页”，只能选择一个在两种浏览器上都能运行的脚本语言。这就使得 JavaScript 成为唯一可能正确的答案。当时，几乎所有的书籍都向读者宣导“兼容浏览器是一件天大的事”。为了这种兼容，一些书籍甚至要求网页制作人员最好不要用 JavaScript，“让所有的事，在服务器上使用 Perl 或 CGI 去做好了”。

然而，随着 IE 4.0 的推出以及缘于 DHTML（Dynamic HML，动态网页）带来的诱惑，一切都发生了改变。

---

[2] LiveWire是 Netscape公司的一个通用的 Web开发环境，仅仅支持 Netscape Enterprise Server和 Netscape FastTrack Server，而不支持其他的 Web服务器。这种技术在服务器端通过内嵌于网页的 LiveScript代码，使用名为 database、DbPool、Cursor 等的一组对象来存取 LiveWire Database。作为整套的解决方案，Netscape在客户端网页上也提供 LiveScript脚本语言的支持，除了访问 Array、String等这些内置对象之外，也可以访问 window等浏览器对象。LiveScript后来发展为 JavaScript，而 LiveWire架构也成为所有 Web服务器提供 SP（Server Page）技术的蓝本。例如，在 IIS中的 ASP，以及更早期的 IDC（Internet Database Connect）。

[3] 该产品即是后来在 1996年 3月发布的 Netscape Enterprise Server 2.0。参见：
http://wp.netscape.com/newsref/pr/newsrelease41.html
http://wp.netscape.com/newsref/pr/newsrelease98.html

[4] 参见：
http://wp.netscape.com/comprod/columns/techvision/innovators_be.html
http://wp.netscape.com/newsref/pr/newsrelease67.html

1.2 用 JavaScript 来写浏览器上的应用

1.2.1 我要做一个聊天室

大概是在 1998 年 12 月中旬，我的个人网站完工了。这是一个文学网站，这个网站在浏览器上用到了 Java Applet 和 JavaScript，并且为 IE 4.0 的浏览器提供了一个称为“搜索助手”的浮动条（FloatBar），用于快速地向服务器提交查询文章的请求。而服务器上则使用了用 Delphi 开发的 ISAPI/CGI，运行于当时流行的 Windows NT 上的 IIS 系统。

我接下来冒出的想法是：要做一个聊天室。因为在我的个人网站中，论坛、BBS 等都有其他网站免费提供，唯独没有聊天室。

1999 年春节期间，我在四川的家中开始做这个聊天室并完成了原型系统（我称之为 beta 0）；一个月后，这个聊天室的 beta 1 终于在互联网上架站运行（如图 1-1 所示）。

图 1-1 聊天室的 beta 1 的界面

这个聊天室的功能集设定见表 1-1。

表 1-1 聊天室 beta 1 的功能集设定

分类	功能	概要
用户	设定名字、昵称、肖像	用户功能按钮 1，对话框
	上传照片并显示给所有人	用户功能按钮 1，对话框
	更改名字、昵称、肖像	用户功能按钮 2~4，快速更换
消息	在当前房间发消息	普通聊天
	向指定用户发消息	用户功能按钮 6，私聊功能
	向所有房间发消息	用户功能按钮 6，通告功能
房间	转到指定房间	用户功能按钮 6，对话框
	创建新房间	房间功能按钮 1
	房间更名	房间功能按钮 2
	设定房间初始化串	房间功能按钮 3
界面	显示/不显示昵称	用户功能按钮 5，动态切换
工具	按名称查找房间	用户功能按钮 7，对话框
	按照 ID/名字查找用户	用户功能按钮 7，对话框
	踢人	房间功能按钮 4

在这个聊天室的右上角有一个“隐藏帧”，是用 Frameset 来实现的。这是最早期实现 Web RPC（Remote Procedure Call）的方法，那时网页开发还不推荐使用 IFrame，也没有后来风行的 AJAX。因此从浏览器下方的状态栏中，我们也可以看到这个聊天室在调用服务器上的.dll——这就是那个用 Delphi 写的 ISAPI CGI。当时我还不知道 PHP，而且 ASP 也并不那么流行。

这个聊天室在浏览器上大量地使用了 JavaScript。一方面，它用于显示聊天信息、控制 CSS 显示和实现界面上的用户交互；另一方面，我用它实现了一个 Command Center，将浏览器中的行为编码成命令发给服务器的 ISAPI CGI。这些命令被服务器转发给聊天室中的其他用户，目标用户浏览器中的 JavaScript 代码能够解释这些命令并执行类似于“更名”、“更新列表”之类的功能——服务器上的 ISAPI 基本上只用于中转命令，因此效率非常高。你可能已经注意到，这其实与现在的 AJAX 的思想如出一辙。

虽然这个聊天室在 beta 0 时还尝试支持了 NN 4，但在 beta 1 时就放弃了，因为 IE 4 提供的 DHTML 模型已经可以使用 insertAdjacentHTML 动态更新网页了，而 NN 4 仍只能调用 `document.write` 来修改页面。

1.2.2 Flash 的一席之地

我所在的公司也发现了互联网上的机会，成立了互联网事业部。我则趁机提出了一个庞大的计划，名为 JSVPS（JavaScripts Visual Programming System）。

JSVPS 在服务器端表现为 `dataCenter` 与 `dataBaseCenter`。前者用于类似于聊天室的即时数据交互，后者则用于类似于论坛中的非即时数据交互。在浏览器端，JSVPS 提出了开发网页编辑器和 JavaScript 组件库的设想。

这时微软的 IE 4.x 已经从浏览器市场拿到了超过 70%的市场份额，开始试图把 Java Applet 从它的浏览器中赶走。这一策略所凭借的，便是微软在 IE 中加入的 ActiveX 技术。于是 Macromedia Flash 就作为一个 ActiveX 插件挤了进来。Flash 在图形矢量表达能力和开发环境方面表现优异，使当时的 Java Applet 优势全失。一方面微软急于从桌面环境挤走 Java，以应对接下来在.NET 与 Java 之间的语言大战；另一方面 Flash 与 Dreamweaver 当时只是网页制作工具，因此微软并没有放在眼里，就假手 Flash 赶走了 Java Applet。

Dreamweaver 系列的崛起，使得网页制作工具的市场变得几乎没有了悬念。主力放在 Java Applet 的工具，如 HotDog 等都纷纷下马；而纯代码编辑的工具，如国产的 CutePage 则被 Dreamweaver 慢慢地蚕食着市场。同样的原因，JSVPS 项目在浏览器端“开发网页编辑器”的设想最终未能实施，而“JavaScript 组件库”也因为市场不明朗而一直不能投入开发。

服务器端的 `dataCenter` 与 `dataBaseCenter` 都成功地投入了商用。此后，我在聊天室上花了更多的精力。到 2001 年下半年，它已经开始使用页签形式来管理多房间同时聊天，并加入语言过滤、表情、行为和用户界面定制等功能。而且，通过对核心代码的分离，聊天室已经衍生出“Web 即时通信工具”和“网络会议室”这样的版本。

2002 年初，聊天室发布的最终版本（ver 2.8）的功能设定已经远远超出了现在网上所见的 Web 聊天室的功能集。图 1-2 中，聊天室最终版本的界面包括颜色选取器、本地历史记录、多房间管理、分屏过滤器、音乐、动作、表情库和 Outlook 样式的工具栏，以及中间层叠的窗体，都是由 DHTML 与 CSS 来动态实现的。在后台驱动这一切的，就是 JavaScript。

图 1-2 聊天室最终版本的界面

我没有选择这时已经开始流行的 Flash，因为用 DHTML 做聊天室的界面效果并不逊于 Flash，也因为在 RWC 与 RIA 的战争中，我选择了前者。

1.2.3 RWC 与 RIA 之争

追溯 RWC 的历史，就需要从“动态网页”说起。在 1997 年 10 月发布的 IE 4 中，微软提供了 JScript 3，这包括当时刚刚发布的 ECMAScript Editon 1，以及尚未发布的 JavaScript 1.3 的很多特性。最重要的是，微软颇有创见地将 CSS、HTML 与 JavaScript 技术集成起来，提出了 DHTML 开发模型（Dynamic HTML Model），这使得几乎所有的网页都开始倾向于“动态（Dynamic）”起来。

开始，人们还很小心地使用着脚本语言，但当微软用 IE 4 在浏览器市场击败网景之后，很多人发现：没有必要为 10%的人去多写 90%的代码。因此，“兼容”和“标准”变得不再重要。于是 DHTML 成了网页开发的事实标准，以至于后来由 W3C 提出的 DOM（Document Object Model）在很长一个阶段中都没有产生任何影响。

这时，成熟的网页制作模式，使得一部分人热衷于创建更有表现能力和实用价值的

网页，他们把这样的浏览器和页面叫做“Rich Web Client”，简称 RWC。Rich Web 的概念产于何时已经不可考，但 Erik Arvidsson 一定是这其中的先行者。他拥有一个知名的个人网站 WebFX（http://webfx.eae.net/）。从 1997 开始，他在 WebFX 上公布关于浏览器上开发体验的文章和代码。他可能是最早通过 JavaScript+DHTML 实现 menu、tree 及 tooltip 的人。1998 年末，他就已经在个人网站上实现了一个著名的 WebFX Dynamic WebBoard，如图 1-3 所示。这套界面完整地模仿了 Outlook，因而是在 Rich Web Client 上实现类 Windows 界面的经典之作。

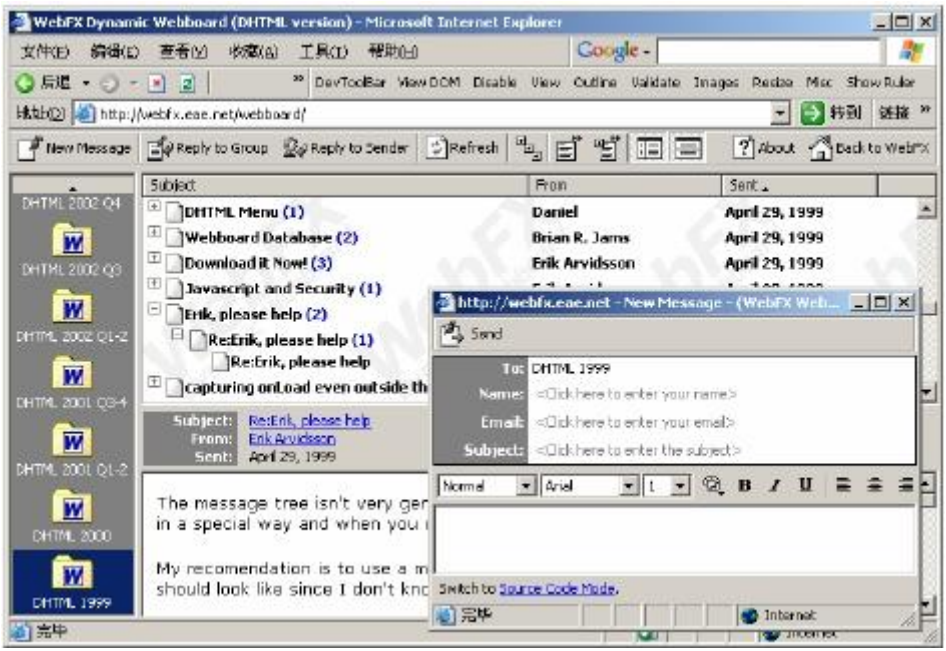


图 1-3 WebFX Dynamic WebBoard 的仿 Outlook 界面

在这时盛行的 Flash 也需要一种脚本语言来表现动态的矢量图形。因此，Macromedia 很自然地在 Flash 2 中开始加入一种名为 Action 的脚本支持。在 Flash 3 时，该脚本参考了 JavaScript 的实现，变得更为强大。随后 Macromedia 又干脆以 JavaScript 作为底本完成了自己的 ActionScript，并加入到 Flash 5 中。随着 ActionScript 被浏览器端开发人员逐渐接受，这种语言也日渐成熟，于是 Macromedia 开始提出自己的对“浏览器端开发”的理解。这就是有名的 RIA（Rich Internet Application）。

这样一来，RIA 与 RWC 分争“富浏览器客户端应用（Rich Web-client Application，RWA）”市场的局面出现了——微软开始尝到自己种下的苦果：一方面它通过基于 ActiveX 技术的 Flash 赶走了 Java Applet，另一方面却又使得 Dreamweaver 和 Flash 日渐坐大，实

在是“前门拒虎，后门进狼”。微软用丢失网页编辑器和网页矢量图形事实标准的代价，换取了在开发工具（如 Virtual Studio .NET）和语言标准（如 CLS，即 Common Language Specification）方面的成功。而这个代价的直接表现之一，就是 RIA 对 RWC 的挑战。

RIA 的优势非常明显，在 Dreamweaver UltraDev 4.0 发布之后，Macromedia 成为网页编辑、开发类工具市场的领先者。而在服务器端，有基于 Server Page 思路的 ColdFusion、优秀的 J2EE 应用服务器 JRun 和面向 RIA 模式的 Flash 组件环境 Flex。这些构成了完整的 B/S 三层开发环境。然而似乎没有人能容忍 Macromedia 独享浏览器开发市场，并试图染指服务器端的局面，所以 RIA 没有得到足够的商业支持。另外，ActionScript 也离 JavaScript 越来越远，既不受传统网页开发者的青睐，而对以设计人员为主体的 Flash 开发者来讲又设定了过高的门槛。

RWC 的状况则更加尴尬。因为 JavaScript 中尽管有非常丰富的、开放的网络资源，但却找不到一套兼容的、标准的开发库，也找不到一套规范的对象模型（DOM 与 DHTML 纷争不断），甚至连一个统一的代码环境都不存在（没有严格规范的 Host 环境）。

在 RIA 热捧浏览器上的 Rich Application 市场的同时，自由的开发者们则在近乎疯狂地挖掘 CSS、HTML 和 DOM 中的宝藏，试图从中寻找到 RWC 的出路。支持这一切的，是 JavaScript 1.3~1.5，以及在 W3C 规范下逐渐成熟的 Web 开发基础标准。在这整个过程中，RWC 都只是一种没有实现的、与 RIA 的商业运作进行着持续抗争的理想而已。

1.3 没有框架与库的语言能怎样发展呢

1.3.1 做一个框架

聊天室接下来的发展几乎停滞了。我在 RWC 与 RIA 之争中选择了 RWC，但也同时面临了 RWC 的困境：找不到一个统一的框架或底层环境。因此，聊天室如果再向下发展，也只能是在代码堆上堆砌代码而已。

因此，整个 2003 年，我基本上都没有再碰过浏览器上的开发。2004 年初的时候，我到一家新的公司（Jxsoft Corporation）任职。这家公司的主要业务都是 B/S 架构上的开发，于是我提出“先做易做的 1/2”的思路，打算通过提高浏览器端开发能力，来加强公司在 B/S 架构开发中的竞争力。

于是我得到很丰富的资源，来主持一个名为 WEUI（Web Enterprise UI Component

Framework）项目的开发工作。这个项目的最初设想，跟 JSVPS 一样是个庞然大物（似乎我总是喜欢如图 1-4 所示的这类庞大的构想）。

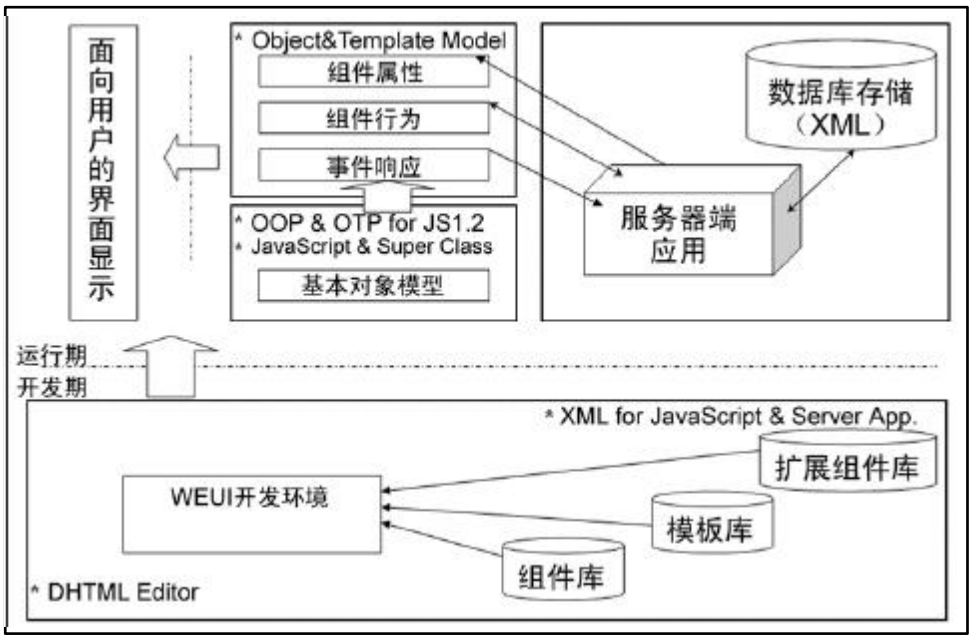


图 1-4 WEUI 基本框架和技术概览

WEUI 包括了 B/S 两端的设计，甚至还有自己的一个开发环境。而真正做起来的时候，则是从 WEUI OOP Framework 开始的。这是因为 JavaScript 语言没有真正的“面向对象编程（Object Oriented Programming，OOP）”框架。

在我所收集的资料中，第一个提出 OOP JavaScript 概念的是 Brandon Myers，他在一个名为 Dynapi[5]的开源项目工作中，提出了名为“SuperClass”的概念和原始代码。在 2001 年 3 月，Bart Bizon 按照这个思路发起了开源项目 SuperClass，放在 SourceForge 上。这份代码维护到 ver 1.7b。半年后，Bart Bizon 放弃了 SuperClass 并重新发起 JSClass 项目，这成为 JavaScript 早期框架中的代表作品。

后来许多 JavaScript OOP Framework 都不约而同地采用了与 SuperClass 类同的方法——使用“语法解释器”来解决框架问题。然而前面提到过的实现了“类 Outlook 界面”的 Erik Arvidsson 则采用了另一种思路：使用 JavaScript 原生代码（native code）在执行期建立框架，并将这一方法用在了另一个同样著名的项目 Bindows 上。

[5] Dynapi是早期最负盛名的 JavaScript开源项目中的一个，它创建得比 Bindows项目更早，参与者也更多。

对于中国的一部分的 JavaScript 爱好者来说，RWC 时代就开始于《程序员》2004 年第 5 期的一篇名为《王朝复辟还是浴火重生——The Return of Rich Client》的文章。这篇文章讲的就是 Bindows 在浏览器上的不凡表现，如图 1-5 所示。

图 1-5 Bindows 在浏览器上的不凡表现

Bindows 可能也是赶上了好时候，这年的 MS Teched 就有好几个专场来讲述智能客户端（Smart Client）。而智能客户端的基本思想就是跨平台的、弹性的富客户端（Rich Client）。因此“丰富的浏览器表现”立即成为“时新”的开发需求，以 Bindows 为代表的 RWC 也因此成为国内开发者和需求方共同关注的焦点。

WEUI v1.0 内核的研发工作大概就结束于此时。我在这个阶段中主要负责的就是 JavaScript OOP Language Core 的开发，并基本完成了对 JavaScript 语言在 OOP 方面的补

充。而接下来，另外的两名开发人员[6]则分别负责 Application Framework 与 Database Layer 的开发，他们的工作完成于 2004 年 8 月。紧接着 WEUI 就被应用到一个商业项目的前期开发中了。WEUI 很快显示出它在浏览器端的开发优势：它拥有完整的 OOP 框架与“基本够用”的组件库，为构建大型的浏览器端应用系统的可行性提供了实证。

WEUI 在开发环境和服务器端上没有得到投入。这与 JSVPS 有着基本相同的原因：没有需求。于是从 2004 年底开始，我就着手于以用户界面（UI）组件库为主要目标的 WEUI v2.0 的开发，直到 2005 年 3 月。

1.3.2 重写框架的语言层

Qomo 项目于 2005 年末启动，它自一开始便立意于继承和发展 WEUI 框架。为此我联系了 WEUI 原项目组以及产品所有的公司，并获得了基于该项目开源的授权。

从 WEUI 到 Qomo 转变之初，我只是试图整理一套有关 WEUI 的文档，并对 WEUI 内核中有关 OOP 的部分做一些修补。因此在这个阶段，我用了一段时间撰写公开文档来讲述 JavaScript 的基本技术，这包括一组名为《JavaScript 面向对象的支持》的文章。而这个过程正好需要我深入地分析 JavaScript 对象机制的原理，以及这种原理与 Qomo 项目中对 OOP 进行补充的技术手段之间的关系。然而这个分析的过程，让我汗如雨下：在此以前，我一直在用一种基于 Delphi 的面向对象思想的方式，来理解 JavaScript 中的对象系统的实现。这种方式完全忽略了 JavaScript 的“原型继承”系统的特性，不但弃这种特性优点于不顾，而且很多实现还与它背道而驰。换而言之，WEUI 中对于 OOP 的实现，不但不是对 JavaScript 的补充，反而是一种伤害。

于是，我决定重写 WEUI 框架的语言层。不过在做出这个决定时，我仍然没有意识到 WEUI 的内部其实还存在着非常多的问题——这其中既有设计方面的，也有实现方面的问题。但我已经决定在 WEUI 向 Qomo 转化的过程中，围绕这些（已显现或正潜藏着的）问题开始努力了。

Qomo Field Test 1.0 发布于 2006 年 2 月中旬，它其实只包括一个`$import()`函数的实现，用于装载其他模块。两个月之后终于发布了 beta 1，已经包括了兼容层、命名空间，以及 OOP、AOP、IOP 三种程序设计框架基础。这时 Qomo 项目组发展到十余人，部分人员已经开始参与代码的编写和审查工作了。我得到了 Zhe[7]的有力支持，他几乎独立完成了兼容层框架以及其在 Mozilla、Safari 等引擎上的兼容代码。很多开源界的，或者对

[6] 他们分别是周鹏飞（leon）与周劲羽（yygw）。我们三人都姓周，实在是巧合。鹏飞现在是微软的软件工程师，而劲羽则领导了 Delphi界非常有名的开源项目 CnPack & CnWizard的开发，现居河南许昌。

[7] Zhe的全名是方哲，爱好研究跨平台兼容问题。他提出并且正在实现基于 QNX6系统模块化网络架构的 CAN栈。

JavaScript 方面有丰富经验的朋友对 Qomo 提出了他们的建议，包括我后来的同事 hax[8]等。这些过程贯穿于整个 Qomo 开发过程之中。

在经历过两个 beta 之后，Qomo 赶在 2007 年 2 月前发布了 v1.0 final。这个版本包括了 Builder 系统、性能分析与测试框架，以及公共类库。此外，该版本也对组件系统的基本框架做出了设计，并发布了 Qomo 的产品路线图，从而让 Qomo 成为一个正式可用的、具有持续发展潜力的框架系统。

回顾 WEUI 至 Qomo 的发展历程，后者不单单是前者的一个修改版本，而几乎是在相同概念模型上的、完全不同的技术实现。Qomo 摒弃了对特殊的或具体语言环境相关特性的依赖，更加深刻地反映了 JavaScript 语言自身的能力。不但在结构上与风格上更为规范，而且在代码的实用性上也有了更大的突破。即使不讨论这些（看起来有些像宣传词的）因素，仅以我个人而言，正是在 Qomo 项目的发展过程中，加深了我对 JavaScript 的函数式、动态语言特性的理解，也渐而渐之地丰富了本书的内容。

1.3.3 富浏览器端开发与 AJAX

事情很快发生了变化——起码，看起来时代已经变了。因为从 2005 年开始，几乎整个 B/S 开发界都在热情地追捧一个名词：AJAX。

AJAX 中的“J”就是指 JavaScript，它明确地指出这是一种基于 JavaScript 语言实现的技术框架。但事实上它很简单，如果你发现它的真相不过是“使用一个对象的方法而已”，那么你可能会不屑一顾。因为如果在 C++、Java 或 Delphi 中，有人提出一个名词/概念（例如 Biby），说“这是如何使用一个对象的技术”，那绝不可能得到如 AJAX 般的待遇。

然而，就是这种“教你如何用一个对象”的技术在 2005 年至 2006 年之间突然风行全球。Google 基于 AJAX 构建了 Gmail；微软基于 AJAX 提出了 Atlas；Yahoo 发布了 YUI（Yahoo! User Interface）；IBM 则基于 Eclipse 中的 WTP（Web Tools Project）发布了 ATF（AJAX Toolkit Framework）……一夜之间，原本在技术上对立或者竞争的公司都不约而同地站到了一起：它们不得不面对这种新技术给互联网带来的巨大机会。

事实上在 AJAX 的早期就有人注意到这种技术的本质不过是同步执行。而同步执行其实在 AJAX 出现之前就已经应用得很广泛：在 IE 等浏览器上采用“内嵌帧（IFrame）”载入并执行代码；在 Netscape 等不支持 IFrame 技术的浏览器中采用“层（Layer）”来载

[8] hax的全名是贺师俊，他领导着一个名为 PIES的 JavaScript开源项目。

入并执行代码。这其中就有 JSRS（JavaScript Remote Scripting），它的第一个版本发布于 2000 年 8 月。

但是以类似于用 JSRS 的技术来实现的 HTTP-RPC 方案存在两个问题：

- Iframe/Layer 标签在浏览器中没有得到广泛的支持，也不为 W3C 标准所认可。
- HTTP-RPC 没有提出数据层的定义和传输层的确切实施方案，而是采用 B/S 两端应用自行约定协议。

然而这仍然只是表面现象。JSRS 一类的技术方案存在先天的不足：它仅仅是技术方案，并不是应用框架，也没有任何商业化公司去推动这种技术。而 AJAX 一开始就是具有成熟商业应用模式的框架，而且许多公司快速地响应了这种技术并基于它创建了各自的“同步执行”的解决方案和编程模型。因此真正使 AJAX 浮上水面的并不是“一个 `XMLHttpRequest` 对象的使用方法”，也并不因为它是“一种同步和异步载入远程代码与数据的技术”，而是框架和商业标准所带来的推动力量。

这时人们似乎已经忘却了 RWC。而 W3C 却回到了这个技术名词上，并在三个主要方面对 RWC 展开了标准化的工作：

- 复合文档格式（Compound Document Formats，CDF）。
- Web 标准应用程序接口（Web APIs）。
- Web 标准应用程序格式（Web Application Formats）。

这其中，CDF 是对 AJAX 中的“X”即 XML）提出标准；而 Web APIs 则试图对“J”（即 JavaScript）提出标准。所以事实上，无论业界如何渲染 AJAX 或者其他的技术模型或框架，Web 上的技术发展方向仍然会落足到“算法+结构”这样的模式上来。这种模式在浏览器上的表现，后者是由 XML/XHTML 标准化来实现的，而前者就是由 JavaScript 语言来驱动的。

微软当然不会错过这样的机会。微软开始意识到 Flash 已经成为“基于浏览器的操作平台”这一发展方向上不可忽视的障碍，因此一方面发展 Altas 项目，为 .NET Framework+ASP.NET+VS.NET 这个解决方案解决 RWC 开发中的现实问题，另一方面，启动被称为“Flash 杀手”的 Silverlight 项目对抗 Adobe 在企业级、门户级富客户端开发中推广 RIA 思想。

现在，编程语言体系也开始发生根本性的动摇。Adobe 购得 Macromedia 之后，把基于 JavaScript 规范的 ActionScript 回馈给开源界，与 Mozilla 开始联手打造 JavaScript 2；SUN 在 Java 6 的 JSR-223 中直接嵌入了来自 Mozilla 的 Rhino JavaScript 引擎，随后 Java 自己也开源了。在另一边，微软借助.NET 虚拟执行环境在动态执行上天生的优势，全力

推动 DLR（Dynamic Language Runtime），其中包括了 Ruby、Python、JavaScript 和 VB 等多种具有动态的、函数式特性的语言实现，这使得.NET Framework 一路冲进了动态语言开发领域的角斗场。

1.4 语言的进化

1.4.1 Qomo 的重生

这时的 Qomo 已经相当完整地实现了一门“高级语言”的大多数特性，我一时间便觉得 Qomo 失去了它应有的方向。原本作为产品的一个组成部分的 WEUI 是有着它的商业目的的，而 Qomo v1.x~v2.0 的整个过程中也有着“开源框架”这样的追求。但当这些目的渐渐远去的时候，Qomo 作为一个没有商业和社区推动的项目，该如何发展呢？

我已经不止一次地关注到了 Qomo 核心部分的复杂性。在[redacted]omo v2 以前，这些复杂性由浏览器兼容、代码组织形式和语言实现技术三个方面构成。举例来说，考虑到 Qomo 的代码包可以自由地拼装，因此 Interface 层的实现就必须要能够完整地从整个框架中剥离出去；另一方面，Interface 层又必须依赖于 OOP 层中所设计的对象实现框架。这使得 Qomo v2 不得不在 Loader 框架中加入了一种类似于编译器的“内联（Inline）”技术，也就是在打包的时候，将一些代码直接插入到指定的位置，以黏合跨层次之间的代码[redacted]

Inline 带来的恶果之一，就是原本的 Object.js 被分成了 9 个片断——当使用不同的选项来打包的时候，这些片断被直接拼接到一个大的文件中；当使用动态加载方式装入 Qomo 的时候，则可以用“Inline”的方式在 eval()代码文本的同时插入这些片断。

我意识到我在触碰一种新的技术的边界。这一技术的核心问题是：一个框架的组织原则与实现之间的矛盾。

Qomo 到底应该设计为何种框架？是为应用而设计，还是仅仅围绕语言特性的扩展？在不同的选择之下，Qomo 又应该被实现成什么样子？

在 2007 年末，我开启了一个新的项目，名为 QoBean。

1.4.2 QoBean 是对语言的重新组织

QoBean 将问题直接聚焦于“语言实现”，开始讨论 JavaScript 语言自身特性的架构方

式、扩展能力及新语言扩展的可能性。缘于这一设定，QoBean 将 Qomo 中的语言层单独地拿了出来，并设定了一些基本原则：

- 不讨论浏览器层面的问题。
- 在 ECMAScript 规范的基础上实现，以保证可移植性。
- 可以完全、透明地替换 Qomo 中的语言层。
- 从语言原子做起。

“从语言原子做起”意味着它必须回归到对 JavaScript 语言的重新思考，即究竟什么才是 JavaScript 语言的“原子”。

“语言原子”这个词我最早读自李战的《悟透 Delphi》。不过这本书终究没有出版，而李战兄后来写了一本《悟透 JavaScript》，算是完成了他的悟透。至于引发我关于 JavaScript 原子问题的思考的，则是在本书第 1 版出版的前后，与起步软件的宋兴烈谈到 JavaScript 的一些特性，而他便提议将这些东西视做“原子特性”。

“这些东西”其实只有两个，其一是对象，其二是函数。它们初次在 QoBean 中的应用，便是“以极小的代价实现 Qomo 的整个类继承框架的完整体系”。而这一尝试的结果是：原本在 Qomo 中用了 565 行代码来实现的 Object.js，在 QoBean 的描述中却只用了 20 行。进一步地，在新的 QoBean 中，为 OOP 语言层做的概念描述也只剩下了两行代码：

```
MetaObject = Function;
MetaClass = Function;
```

元语言的概念在 Qomo/QoBean 中渐渐浮现出来。2008 年 7 月，亦即是在发布 QoBean alpha1 之后的半年，我为 QoBean 撰写了两篇博客文章：QoBean 的元语言系统（一）、二）。QoBean 也在此时基本完成了对“Qomo 的语言层”的重新组织。

事实上，QoBean 本质上是探讨了 JavaScript 这门语言的一种扩展模式，即基于语言自身的原子特性进行二次实现的能力。这与后来的其他一些语言扩展，有着完全不同的路径，以及基本相同的目的。

我们在探索的是 JavaScript 这门语言的边界。

1.4.3 JavaScript 作为一门语言的进化

JavaScript 之父 Brendan Eich 曾说：“ 我们最初利用 JavaScript 的目的，是让客户端的应用不必从服务器重新加载页面即可回应用户的输入信息，并且提供一种功能强大的图形工具包给脚本编写者。”这包括在客户端的两个方面的功能，第一是用户交互，第二是用户界面。而展现与交互，正是现在对“前端职能”的两个主要定义。所以这个语言的

最初构想，与它现在所应用的主要领域是悄然契合的。

但在新千年之后，浏览器取代传统的操作系统桌面渐渐成为热门的“客户端”解决方案，AJAX 在这时作为一种客户端技术对这一技术选型起到了推波助澜的作用。与此同时，开发人员觉察到 JavaScript 作为一门语言，在客户端实现技术中难以有足够丰富的实现能力。于是语言级别的扩展纷纷出现：在代码组织上，开始有了命名空间；在运行效率上，有了编译压缩；在标准化方面，有了 Common JS；在语言扩展上，有了在 JavaScript 中嵌入的解释语言……

对于传统的高级程序设计语言来讲，这一切是再自然不过了。然而 JavaScript 毕竟只是一种脚本语言，这些“附加的”扩展与“第三方的”实现事实上带来了更大的混乱。于是大公司开始提出种种“统一框架”，或者兼容并包的“整合方向”。在这一选择中，大公司首先解决的是“自己的”问题，这是由于在他们的种种产品、产品线中尤其需要这样一种统一的、标准的方案，以避免重复投入带来的开发与维护成本。因此，“商业化产品+自主的 JavaScript 开发包”成为一种时兴的产品模式。在这一产品模式中，由于 JavaScript 本身不具有源码保护的特性，因此源码开放既是战略，又是手段，也是不得已而为之的事。

现在我们有机会看到 Plam 手机平台上完整的 Web OS 的代码，也有机会直接读到 FireBug 中全部用于支持引擎级调试的代码，我们手边还有类似于 YUI、Dojo 等的企业级框架可用。然而，一切还是一如既往的乱。因为，这一切只是基于语言的扩展，而非语言或其基础库中的特性。造成这一事实的原因，既源于 JavaScript 自身初始设计的混合式特性，也因为 ECMA（European CompAter Manufacturing Association，欧洲计算机制造协会）在这门语言的标准化工作上的滞后与反复，如图 1-6 所示。

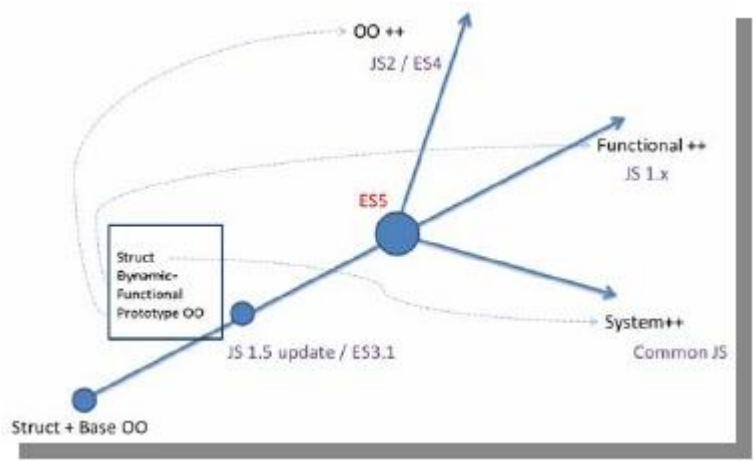


图 1-6 Java Script 的语言特性发展及其标准化过程

JavaScript 在发布 v1.0 的时候，仅有结构化语言的特性以及一些基础的面向对象特性（即 Struct + Base OO），这一切直到 v1.3 的版本时才得以本质性的改观，使得这一语言成为包括动态语言、函数式语言特性在内的特性丰富的混合语言。这一切被 ECMA 通过标准化的形式确定下来，形成了以 JavaScript 1.5 和 ECMAScript Ed3 版本（即 JS 1.5/ES3.1）为代表的，一直到我们今天仍在使用的 JavaScript 语言。

随后 ECMA 启动了一个工作组来进行下一代 JavaScript 的标准化工作，称为 ECMAScript Ed4（亦即是 ES4/JS2），这个小组的工作始于 2003 年 3 月。然而他们搞砸了整件事，他们几乎把 JS2 设计成了一门新的、大的、复杂的语言。因此，一方面一些需要“丰富的语言特性”的厂商不遗余力地推进着这一标准，另一方面它又被崇尚轻捷的前端开发人员诟病不已。迟至 2009 年年底，ECMA Ed4 的标准化小组终于宣布：ES4 的标准化暂停，并将基于 ES4 的后续工作称为 ECMAScript Harmony。接下来，为了解决“互联网开发需要更新的、标准化的 JavaScript 语言”这一迫切问题，不久他们就发布了 ECMAScript Ed5 这一版本，亦即是 ES5。严格地说，ES5 与 ES4 基本没有什么关系，而是对 ES3.1 所代表的语言方向的一个补充。

换言之，ES5 没有改变 JavaScript 1.x 的语言特性；ES4 则是一门集语言生产商所有创想之大成，而又与 JavaScript 1.x 倡导的语言风格相去甚远的语言。历史证明，我们暂弃了后者——过去 10 年，我们都未能将 JavaScript 1.x 推进到 v2.0 版本。

由于 ES5 秉承了 JavaScript 设计的原始思想，因此基于 ES5 又展开了新一轮的语言进化的角力（图 1-6 中的三条虚线试图说明，语言最初的特性设计是这一切的根源）：

- 第一个方向由 Microsoft、Adobe 等大厂商所倡导，沿着 ES4——或称之为 JS2、ECMAScript Harmony 的方向，向更加丰富的面向对象（OO++）特性发展，主要试图解决大型系统开发中所需要的复杂的对象层次结构、类库及框架；
- 第二个方向，则由包括 Brendan Eich 本人在内的语言开发者、研究者主导，他们力图增强 JavaScript 语言的函数式特性（Functional++），因为这样的语言特性在解决许多问题时来得比结构化的、面向对象的语言更优雅有效，而且从语言角度看来，函数式更为“纯粹”。
- 在第三个方向上，Common JS 等开发组意识到 JS 1.x 在应用于浏览器之外的开发场景中，以及在组织大型项目方面显得无力。而将这一问题归结起来，就是“缺乏基础运行框架和运行库”，于是通过参考传统的、大型的、系统级的应用开发语言，尝试性地提出在 JavaScript 中的同等解决方案（System++）。

具体的引擎或框架已经不再是被关注的话题。ES4 的失败给整个 JavaScript 领域带来的思考是：我们究竟需要一种怎样的语言？

1.5 为 JavaScript 正名

至 2005 年，JavaScript 就已经诞生 10 年了。然而 10 年之后，这门语言的发明者 Brendan Eich 还在向这个世界解释“JavaScript 不是 Java，也不是脚本化的 Java（Java Scription）”。

这实在是计算机语言史上最罕见的一件事了。因为如今几乎所有的 Web 页面中都同时包含了 JavaScript 与 HTML，而后者从一开始就被人们接受，前者却用了 10 年都未能向开发人员说清楚“自己是什么”。

Brendan Eich 在这份名为“JavaScript 这十年（JavaScript at Ten Years）[9]”的演讲稿中，重述了这门语言的早期历史：Brendan Eich 自 1995 年 4 月受聘于网景公司，开始实现一种名为“魔卡（Mocha）”——JavaScript 最早的开发代号或名称的语言；仅两个月之后，为了迎合 Netscape 的 Live 战略而更名为 LiveScript；到了 1995 年年末，又为了迎合市场对 Java 语言的热情，正式地、也是遗憾地更名为 JavaScript，并随网景浏览器推出[10]。

Brendan 在这篇演讲稿最末一行写道：“不要让营销决定语言名称（Don't let Marketing name your language）”。一门被误会了 10 年的语言的名字之争，是不是就此结束了呢？仍然不是。因为这 10 年来，JavaScript 的名字已经越来越乱，更多市场的因素困扰着这门语言，好像“借用 Java 之名”已经成了扔不掉的黑锅。

1.5.1 JavaScript

我们先说正式的、标准的名词：JavaScript。它实际是指两个东西：

- 一种语言的统称。该语言由 Brendan Eich 发明，最早用于 Netscape 浏览器。
- 上述语言的一种规范化的实现。在 JavaScript 1.3 之前，网景公司将他们在 Netscape 浏览器上的该语言规范的具体实现直接称为 JavaScript，并一度以“ Client-Side JavaScript”与“Server-Side JavaScript”区分该语言在浏览器 NN 与 NWS（Netscape Web Server）上的实现，但后来他们改变了这个做法。

[9] Brendan's keynote "JavaScript at Ten Years（Powerpoint）" for the ACM ICFP 2005 conference：http://developer.mozilla.org/en/docs/JavaScript_Language_Resources

[10] 部分信息引自于以下文献：
《The History of JavaScript》：http://inventors.about.com/od/jstartinventions/a/JavaScript.htm
《JavaScript Tutorial Part I》：http://www.techiwarehouse.com/

1.5.2 Core JavaScript

Core JavaScript 这个名词早在 1996 年（或更早之前）就被定义过，但它直到 1998 年 10 月由网景公司发布 JavaScript 1.3 时才被正式提出来。准确地说，它是指由网景公司和后来的开源组织 Mozilla，基于 Brendan Eich 最初版本的 JavaScript 引擎而发展出来的脚本引擎，是 JavaScript 规范的一个主要的实现者、继承者和发展者。

Core JavaScript 的定义如图 1-7 所示。

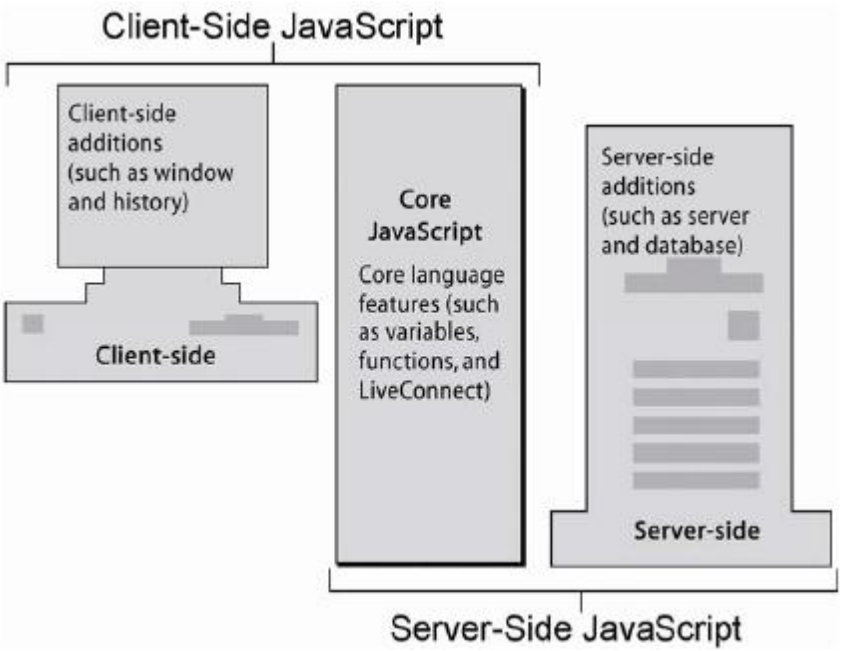


图 1-7 官方手册中有关 Core JavaScript 的概念说明

在 JavaScript 1.3 发布时，Netscape 意识到他们不能仅仅以 Client/Server 来区分 JavaScript，因为市面已经出现了很多种 JavaScript。于是他们做了一些小小的改变：在发布手册时，分别发布“Core JavaScript Guide”和“Client-Side JavaScript Guide”。前者是指语言定义与语法规范，后者则是该语言的一种应用环境与应用方法。

所以事实上，自 1.3 版本开始，Core JavaScript 1.x 与 JavaScript 1.x 是等义的，换言之，我们现在常说的 JavaScript 1.x，就是指 Core JavaScript，而并不包括 Client-Side JavaScript。不过，源于一些历史的因素，在 Core JavaScript 中会有一部分关于“LiveConnect 技术”的叙述及规范。这在其他（所有的）JavaScript 规范与实现中均是不具备的。

不幸的是，Apple 公司有一个基于 KJS 实现的 JavaScript 引擎，名为 JavaScriptCore，

属于 WebKit 项目的一个组成部分，WebKit 项目所实现的产品就是著名的开源跨平台浏览器 Safari。所以在了解 Core JavaScript 同时，还需强调它与 JavaScriptCore 的不同。

1.5.3 SpiderMonkey JavaScript

Brendan Eich 编写的 JavaScript 引擎最后由 Mozilla 贡献给了开源界，SpiderMonkey 便是这个产品开发中的、开源项目的名称（code-name，即项目代码名）。为了与我们通常讲述的 JavaScript 语言区分开来，我们使用 SpiderMonkey 来特指上述由 Netscape 实现的、Mozilla 和开源社区维护的引擎及其规范。

Mozilla Firefox 4.0 以后的版本对 SpiderMonkey 进行了较大的更新，大量使用 JIT（Just In Time）编译技术来提升引擎性能[11]。并且从 Firefox 4.0 版本开始，Mozilla 发布了 JavaScript 1.8.5 版本，开始支持 ES5 规范下的语言特性。

在本书此后的描述中，凡称及 SpiderMonkey JavaScript 的，将是特指于此；凡称及 JavaScript 的，将是泛指 JavaScript 这一种语言的实现。

1.5.4 ECMAScript

JavaScript 的语言规范由网景公司提交给 ECMA 去审定，并在 1997 年 6 月发布了名为 ECMAScript Edition 1 的规范，或者称为 ECMA-262。4 个月后，微软在 IE 4.0 中发布了 JScript 3.0，宣称成为第一个遵循 ECMA 规范来实现的 JavaScript 脚本引擎。

而因为计划改写整个浏览器引擎的缘故，网景公司晚了整整一年才推出“完全遵循 ECMA 规范”的 JavaScript 1.3。请注意到这样一个问题：网景公司首先开发了 JavaScript 并提交 ECMA 标准化，但在市场的印象中，网景公司的 Core JavaScript 1.3比微软的 JScript 3.0 晚了一年实现 ECMA 所定义的 JavaScript 规范。这直接导致了一个恶果：JScript 成为 JavaScript 语言的事实标准。

在本书此后的描述中，我们将基于 ECMAScript Edition 3 的规范来讲述 JavaScript。凡未特别指明的叙述中，所谓 JavaScript 即是指“一种符合 ECMAScript Edition 3 规范的 JavaScript 实现”。

[11] 事实上从 Firefox 3.5中便开始加入 JIT编译引擎，这使得从 Firefox 4.0至 6.x版本中的 SpiderMonkey总是同时带有两套 JIT 引擎，分别称为 TraceMonkey和 JagerMonkey，来做执行期的优化。在 Firefox 7之后的版本中，（计划）将采用第三套 JIT引擎，即 IonMonkey。

1.5.5 JScript

微软于 1996 年在 IE 中实现了一个与网景浏览器类似的脚本引擎，微软把它叫做 JScript 以示区别，结果 JScript 这个名字一直用到现在。

直到 JScript 3.0 之后，JavaScript 语言的局面才显得明朗起来，如图 1-8 所示。

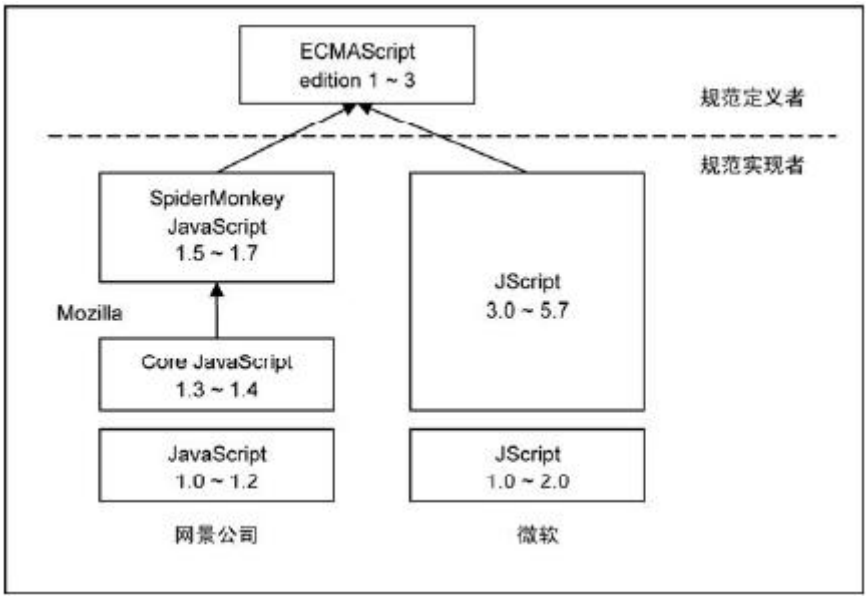


图 1-8 JScript 与 JavaScript 各版本间的关系

由于 JScript 成为 JavaScript 语言的事实标准，再有 IE 浏览器几乎占尽市场（如果我们现在不是在讨论 JavaScript，你也可以把这个因果颠倒过来），因此在 1999 年之后，Web 页面上出现的脚本代码基本上都是基于 JScript 开发的，而 Core JavaScript 1.x 却变成了“（事实上的）被兼容者”。

直到 2005 年前后，源于 W3C、ECMA 对网页内容与脚本语言标准化的推动，以及 Mozilla Firefox 成功地返回浏览器市场，Web 开发人员开始注重所编写的脚本代码是否基于 JavaScript 的——即是 ECMAScript 的标准规范，这成为了新一轮语言之争的起点。

1.5.6 总述

JavaScript 这个名词的多种含义见表 1-2。

表 1-2 名词“JavaScript”的多种含义

含义	详述
JavaScipt脚本语言	一种语言的统称，由 ECMAScript 262规范。涵盖 Core JavaScript、JScript、ActionScript等，而非特指其一
浏览器 JavaScript	包括 DOM、BOM模型等在内的对象体系，但不确指具体脚本环境。是目前 JavaScript最为广泛的应用环境，（在 Netscape/Mozilla系列中的浏览器 JavaScript）也被称为 Client-Side JavaScript
Core JavaScript	也称 SpiderMonkey JavaScript，主要指 Netscape/Mozilla系列的浏览器环境中的 JavaScript，是该语言的主要规范之一
JScript	仅指 IE系列的浏览器环境中的 JavaScript，是使用最广泛的一种JavaScript脚本语言实现和该语言主要规范之一。另外，在 Windows Script Host（WSH）、Active Server Page（ASP）等脚本开发环境中，也包括 JScript这一语言。而在.NET Framework环境中提供的则称为 JScript .NET，是基于 ECMAScritp Ed4标准实现的 JScript版本

1.6 JavaScript 的应用环境

在此前的内容中，我讨论的都是 JavaScript 语言及其规范，而并非该语言的应用环境。在大多数人看来，JavaScript 的应用环境都是 Web 浏览器，这也的确是该语言最早的设计目标。然而从很早开始，JavaScript 语言就已经在其他的复杂应用环境中使用，并受这些应用环境的影响而发展出新的语言特性了[12]。

JavaScript 的应用环境，主要由宿主环境（host environment）与运行期环境构成。其中，宿主环境是指外壳程序（Shell）和 Web 浏览器等，而运行期环境则是由 JavaScript 引擎内建的。图 1-9 是由它们共同构建的对象编程系统的基本结构。

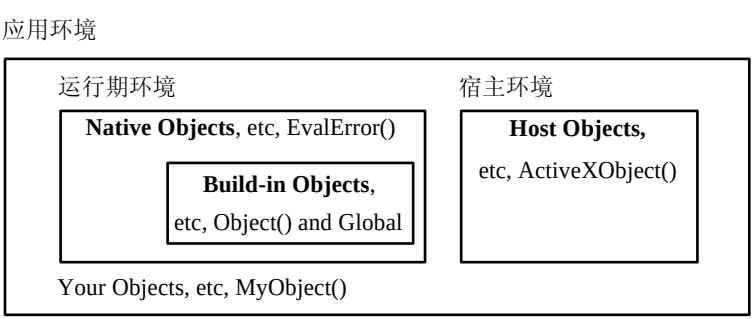


图 1-9 由宿主与运行期构成的应用环境

[12] 正是这些复杂的应用环境推动了 JavaScript 2的到来。按照 ECMAEdition 4标准规范小组的说明，ECMAEdition 4主要面对的问题就在于 JavaScript 1.x没有足够的抽象能力和语言机制，因而难以胜任大型编程系统环境下的开发。

1.6.1 宿主环境

JavaScript 是一门设计得相对“原始”一点点的语言，它被创生时的最初目标仅仅是为 Netscape 提供一个在浏览器与服务器间都能统一使用的开发语言。简单地说，它原来是想让 B/S 架构下的开发人员用起来都舒服那么一点点的。这意味着最初的设计者希望 JavaScript 语言是跨平台的，能够提供“端到端（side to side）”的整体解决方案。

然而，事实上这非常难做到，因为不同的平台提供的“可执行环境”不同。而宿主环境就是为了隔离代码、语言与具体的平台而提出的一个设计。一方面我们不能让浏览器上拥有一个巨大无比的运行期环境（例如像虚拟机那么大），另一方面服务器端又需要一个较强大的环境，因此 JavaScript 就被设计成了需要“宿主环境”的语言[13]。

ECMAScript 规范并没有对宿主环境提出明确的定义。比如说，它没有提出标准输入输出（stdin、stdout）需要确切地实现在哪个对象中。为了弥补这个问题，RWC 在 WebAPIs 规范中首先就提出了“需要一个 window 对象”的浏览器环境。这意味着在 RWC 或者浏览器端，是以 window 对象及其中的 Document 对象来提供输入输出的。

但这仍然不是全部的真相。因为“RWC 规范下的宿主环境”，并不等同于“JavaScript 规范下的宿主环境”。本书并不打算讨论与特定浏览器相关的细节问题，因此我们事实上在说的是 JavaScript 的一个公共语言环境，或者说公共的宿主环境的定义。作为程序运行过程中对输入输出的基本要求，本书设定宿主环境在全局应当支持如表 1-3 所示的方法。

表 1-3 本书对宿主环境在全局方法上的简单设定

方法	含义	备注
`alert(sMessage)`	显示一个消息文本（字符串），并等待用户一次响应。调用者将忽略响应的返回信息	
`write(sText, ...)`	输出一段文本，多个参数将被连接成单个字符串文本	（*注 1）
`writeln(sText, ...)`	（同 `write`）输出一段文本，多个参数将被连接成单个字符串文本，并在文本末尾追加一个换行符（`\n`）	

*注 1：`write()`与 `writeln()`在浏览器中是 Document对象的方法。为遵循这一惯例，在本书的所有测试范例中并不直接使用这两个方法。但这里保留了它们，以描述宿主环境的标准输入输出。

对于不同的宿主来说，这些方法依赖于不同的对象层次的“顶层对象（或全局对象）”。例如浏览器宿主依赖于 window 对象，而 WSH 宿主则依赖于 WScript 对象。但在本书中，调用这些方法时将略去这个对象。因此，至少它看起来很像是 Global 对象上的方法（事实上，大多数的宿主默认“顶层对象”不需要使用全名的约定）。

[13] 虚拟机（Virtual Machine）是另一种隔离语言与平台环境的手段。Java与.NET Framework都以虚拟机的方式提供运行环境。在 JavaScript 2中，几乎所有的实现者都在宿主环境的基础上采用了虚拟机的方案。在这种方案中，宿主的作用是提供混合语言编程的能力和跨语言的对象系统，而虚拟机则着眼于跨平台的、语言无关的虚拟执行环境。

下面的代码说明在具体的宿主环境中如何实现本书所适用的 alert()方法。例如：

```
//示例 1： .NET Framework中的 JScript 8.0，（当前的）顶层对象取决于import语句
//（注：JScript.NET中的脚本需要编译执行）
import System.Windows.Forms;
function alert(sMessage) {
  MessageBox.Show(sMessage);
}
alert('Hello, World!')

//示例 2：浏览器环境中使用的顶层对象是 window
alert('Hello, World!');

//示例 3：WSH环境中使用的顶层对象是 WScript，但必须使用全名
function alert(sMessage) {
  WScript.Echo(sMessage);
}
alert('Hello, World!');
```

1.6.2 外壳程序

外壳程序（shell）是宿主的一种。不过在其他一些文档中并不这样解释，而是试图将宿主与外壳分别看待。这其中的原因在于将“跨语言宿主”与“应用宿主”混为一谈。

Windows 环境中，微软提供的 WSH（Windows Script Host）是一种跨语言宿主，在该宿主环境中提供一个公共的对象系统，并提供装载不同的编程语言引擎的能力。如此一来，WSH 可以让多个语言使用同一套对象——这些对象由一些 COM 组件来实现并注册到 Windows 系统中。所以，我们在 IE 浏览器中看到，既可以用 VBScript 操作网页中的对象，也可以用 JScript 来操作它。基本上来讲，IE 浏览器采用的是与 WSH 完全相同的宿主实现技术。

多数 JavaScript 引擎会提供一个用于演示的外壳程序。该外壳程序通过一种命令行交互式界面来展示引擎的能力，在 UNIX/Linux 系统中编程的开发人员会非常习惯这种环境，而在 Windows 中编程的开发人员则不然。在这种环境下，可以像调试器中的单步跟踪一样，展示出许多引擎内部的细节。图 1-10 是 SpiderMonkey JavaScript 随引擎同时发布一个外壳程序，它就是（该脚本引擎的）一个应用宿主。

如同引擎提供的这种外壳程序一样，我们一般所见的 Shell 是指一种简单的应用宿主，它只负责提供一个宿主应用环境：包括对象和与对象运行相关的操作系统进程。但是在另外一些情况下，“外壳（而不是外壳程序）”和“宿主”也被赋予一些其他的含义。例如在 WSH 中，“宿主”是指整个宿主环境和提供该环境的技术，而 Shell 则是其中的一个可编程对象（WScript.WshShell）——封装了 Windows 系统的功能（如注册表、文件系统等）的一个“外壳对象”，而非“外壳程序”。

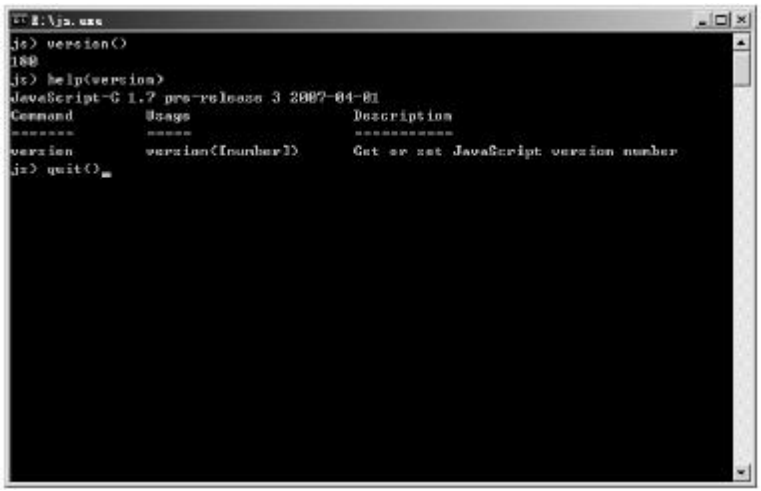
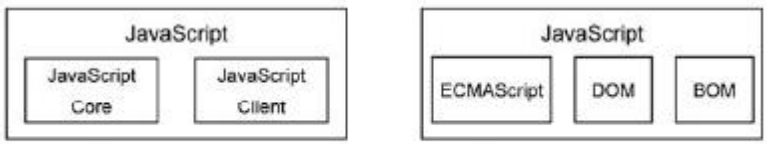


图 1-10 SpiderMonkey JavaScript 提供的外壳程序

讨论脚本引擎本身时，我们并不强调宿主环境的形式是 WSH 这种“使用跨语言宿主技术构建的脚本应用环境”，还是 SpiderMonkey JavaScript 所提供的这种“交互式命令行程序”。我们只强调：脚本引擎必须运行在一个宿主之中，并由该宿主创建和维护脚本引擎实例的“运行期环境（runtime）”。

1.6.3 运行期环境

在不同的书籍中对 JavaScript 运行期环境的阐释是不一致的。例如在《JavaScript 权威指南》中，它由 JavaScript 内核（Core）和客户端（Client）JavaScript 两部分构成；而在《JavaScript 高级程序设计》中，它被描述成由核心（ECMAScript）、文档对象模型（DOM）、浏览器对象模型（BOM）三个部分组成，如图 1-11 所示）。



（a）偏向 Netscape、Mozilla 的《JavaScript 权威指南》 （b）偏向 Internet Explorer 的《JavaScript 高级程序设计》

图 1-11 对“运行期环境”的不同解释

本书是从引擎的角度讨论 JavaScript 的，因此在本书看来，与浏览器相关的内容都属于“应用环境”：属于宿主环境或属于用户编程环境。图 1-9 由宿主与运行期构成的应用环境”表达了这种关系。在这样的关系中，运行期环境是由宿主通过脚本引擎（JavaScript

Engines）创建的[14]。图 1-12 说明应用程序，是宿主在这里可以看成一个应用程序，是如何创建运行期环境[15]的。

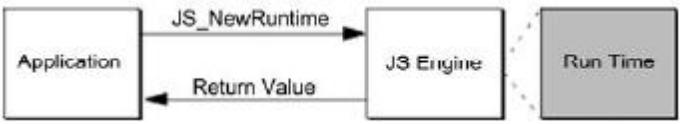


图 1-12 应用（宿主）通过引擎创建“运行期环境”的过程

这相当于是说，在本书中讲述的运行期环境，是特指由引擎创建的初始应用环境。这样解释运行期环境的特点，而并不强调（或包括）在应用、宿主或用户代码混杂作用的、运行过程中的应用环境。在初始状态下的运行期环境主要包括：

- 一个对宿主的约定。
- 一个引擎内核。
- 一组对象和 API。
- 一些其他的规范。

换而言之，这是指一个引擎自身的能力。不过即使如此，不同的 JavaScript 脚本引擎所提供的语言特性也并不一致。因此，在本书中若非特别说明，JavaScript 是指一种通用的、跨平台和跨环境的语言，并不特指某种特定的宿主环境或者运行环境。也就是说，它是指 ECMAScript 262 所描述的语言规范。目前最常见的实现 ECMAScript 262-3 或 JavaScript 1.5 以上版本规范的引擎包括如表 1-4 所示的几种[16]。

表 1-4 最常见的一些 JavaScript 引擎（部分）

引擎	应用	语言	备注
SpiderMonkey	Mozilla	C	
JavaScriptCore	Safari	C++	基于 KDE发布的 KJS，由 Apple公司支持
Rhino	Java	Java	主要应用于 IBM、Sun等的 Java平台
JScript	Windows		Windows环境，以及 IE
Narcissus		JavaScript	（*注 1）

*注 1：Brendan Eich为验证 JavaScript语言的自实现能力而写的一套代码，被称为“JS implemented in JS”。有许多项目基于该代码进行扩展，例如 NarrativEJS基于该项目实现了 JavaScript上的解释器、编译器和扩展语法。

[14] 从物理实现的角度上来看，一些常见的 JavaScript Engines包括在 Windows和 Internet Explorer中的 jscript.dll，以及在 Mozilla Firefox中的 js3250.dll。在绝大多数基于 JavaScript 1.x实现的系统中，运行期环境是由这样的脚本引擎决定的；但在 JavaScript 2中，由于虚拟机的存在，运行期环境被赋予了更为复杂的含义。

[15] 引用自《JavaScript C Engine s Guide》：http://developer.mozilla.org/en/docs/JavaScript_C_Engine_Embedder's_Guide。

[16] 本书附录将以图表的形式展示更为复杂的脚本引擎的继承关系，以及其规范的发展。