动态语言技术
精品书廊

Broadview
www.broadview.com.cn

JAVASCRIPT
语言精髓与编程实践

周爱民 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

目录

代　序

学两种语言

—《我的程序语言实践》节选—

《程序设计语言——实践之路》一书对“语言”有一个分类法，将语言分类为“说明式”与“命令式”两种。Delphi 以及 C、C++、Java、C#等都被分在“命令式”语言范型的范畴；“函数式”语言则是“说明式”范型中的一种。如今回顾自己对语言的学习，其实十年也就学会了两种语言：一种是命令式的 Pascal/Delphi，另一种则是说明式的 JavaScript。当然从语言的实现方式来看，一种是静态的，一种是动态的。

这便是我程序员生涯的全部了。

我毕竟不是计算机科学的研究者，而只是其应用的实践者，因此我从一开始就缺乏对“程序”的某些科学的或学术层面上的认识是很正常的。也许有些人一开始就认识到程序便是如此，或者一种语言就应当是这样构成和实现的，那么可能他是从计算机科学走向应用，故而比我了解得多些。而我，大概在十年前学习编程，以及在后来很多年的实践中，仅被要求“写出代码”，而从未被要求了解“什么是语言”。所以我才会后知后觉，才会在很长的时间里迷失于那些精细的、沟壑纵横的语言表面而不自知。然而一如我现在所见到，与我曾相同地行进于那些沟壑的朋友，仍然在持续地迷惑着、盲目着，全然无觉于沟壑之外的瑰丽与宏伟。

前些天写过一篇 BLOG，是推荐那篇“十年学会编程”的。那篇文章道出了我在十年编程实践之后，对程序语言的最深刻的感悟。我们学习语言其实不必太多，深入一两种就可以了。如果在一种类型的语言上翻来覆去，例如不断地学 C、Delphi、Java、C#……无非是求生存、讨生活，或者用以装点个人简历，于编程能力的提高是不大的。更多的人，因为面临太多的语言选择而浅尝辄止，多年之后仍远离程序根本，成为书写代码的机器，把书写代码的行数、程序个数或编程年限作为简历中最显要的成果。这在明眼人看来，无过是熟练的砌砖工而已。

我在《大道至简》中说“如今我已经不再专注于语言”。其实在说完这句话之后，我就已经开始了对 JavaScript 的深入研究。在如此深入地研究一种语言，进而与另一种全然有别的语言比较补充之后，我对“程序＝算法＋结构”有了更深刻的理解与认识——尽管这句名言从来未因我的认识而变化过，从来未因说明与命令的编程方式而变化过，也从来未因动态与静态的实现方法而变化过。

动静之间，不变的是本质。我之所以写这篇文章，并非想说明这种本质是什么抑或如何得到，只是期望读者能在匆忙的行走中，时而停下了脚步，远远地观望一下目标罢了。

而我此刻，正在做一个驻足观望的路人甲。

周爱民

前　言

语言

语言是一种交流的工具，这约定了语言的“工具”本质，以及“交流”的功用。“工具”的选择只在于“功用”是否能达到，而不在于工具是什么。

在数千年之前，远古祭师手中的神杖就是他们与神交流的工具。祭师让世人相信他们敬畏的是神，而世人只需要相信那柄神杖。于是，假如祭师不小心丢掉了神杖，就可以堂而皇之地再做一根。甚至，他们可以随时将旧的换成更新或更旧的神杖，只要他们宣称这是一根更有利于通神的杖。对此，世人往往做出迷惑的表情，或者欢欣鼓舞的情状。今天，这种表情或情状一样地出现在大多数程序员的脸上，出现在他们听闻到新计算机语言被创生的时刻。

神杖换了，祭师还是祭师，世人还是会把头叩得山响。祭师掌握了与神交流的方法（如果真如同他们自己说的那样的话），而世人只看见了神杖。

所以，泛义的工具是文明的基础，而确指的工具却是愚人的器物。

计算机语言有很多种分类方法，例如高级语言或者低级语言。其中一种分类方法，就是“静态语言”和“动态语言”——事物就是如此，如果用一对绝对反义的词来分类，就相当于概含了事物的全体。当然，按照中国人中庸平和的观点，以及保守人士对未知可能性的假设，我们还可以设定一种中间态：半动态语言。你当然也可以叫它半静态语言，这个随便你。

所以，我们现在是在讨论一种很泛义的计算机语言工具。至少在眼下，它（在分类概念中）概含了计算机语言的二分之一。当然，限于我自身的能力，我只能讨论一种确指的工具，例如JavaScript。但我希望你由此看到的是计算机编程方法的基础，而不是某种愚人的器物。JavaScript 的生命力可能足够顽强，我假定它比 C 还顽强，甚至比你我的生命都顽强。但它只是愚人的器物，因此反过来说：它能不能长久地存在都并不重要，重要的是它能不能作为这“二分之一的泛义”来供我们讨论。

分类法

新打开一副扑克牌，我们总看到它被整齐的排在那里，从 A 到 K 及大小王。接下来，我们将它一分为二，然后交叉在一起；再分开，再交叉……但是在重新开局之前，你是否注意到：在上述过程中，牌局的复杂性其实不是由“分开”这个动作导致的，而是由“交叉”这个动作导致的。

所以分类法本身并不会导致复杂性。就如同一副新牌只有四套 A~K，我们可以按十三牌面来分类，也可以按四种花色来分类。当你从牌盒里把它们拿出来的时候，无论它们是以哪种

方式分类的，这副牌都不混乱。混乱的起因，在于你交叉了这些分类。

同样的道理，如果世界上只有动态、静态两种语言，或者真有半动态语言而你又有明确的“分类法”，那么开发人员将会迎来清醒明朗的每一天：我们再也不需要花更多的时间去学习更多的古怪语言了。

然而，第一个问题便来自于分类本身。因为“非此即彼”的分类必然导致特性的缺失——如果没有这样“非此即彼”的标准就不可能形成分类，但特性的缺失又正是开发人员所不能容忍的。

我们一方面吃着碗里，一方面念着锅里。即使锅里漂起来的那片菜叶未见得有碗里的肉好吃，我们也一定要捞起来尝尝。而且大多数时候，由于我们吃肉吃腻了嘴，因此会觉得那片菜叶味道其实更好。所以首先是我们的个性，决定了我们做不成绝对的素食者或肉食者。

当然，更有一些人说我们的确需要一个新的东西来使我们更加强健。但不幸的是，大多数提出这种需求的人，都在寻求纯质银弹[1]或混合毒剂[2]。无论如何，他们要么相信总有一种事物是完美武器，或者更多的特性放在一起就变成了魔力的来源。

我不偏向两种方法之任一。但是我显然看到了这样的结果，前者是我们在不断地创造并特化某种特性，后者是我们在不断地混合种种特性。

更进一步地说，前者在产生新的分类法以试图让武器变得完美，后者则通过混淆不同的分类法，以期望通过突变而产生奇迹。

二者相同之处，都在于需要更多的分类法。

函数式语言就是来源于另外的一种分类法。不过要说明的是，这种分类法是计算机语言的原力之一。基本上来说，这种分类法在电子计算机的实体出现以前就已经诞生了。这种分类法的基础是“运算产生结果，还是运算影响结果”。前一种思想产生了函数式语言（如 LISP）所在的“说明式语言”这一分类，后者则产生了我们现在常见的 C、C++等语言所在的“命令式语言”这一分类。

然而我们已经说过，人们需要更多的分类的目的，是要么找到类似银弹的完美武器，要么找到混合毒剂。所以一方面很多人宣称“函数式是语言的未来”，另一方面也有很多人把这种分类法与其他分类法混在一起，于是变成了我们这本书所要讲述的“动态函数式语言”——当然，毋庸置疑的是：还会有更多的混合法产生。因为保罗·格雷厄姆（Paul Graham）[3]已经做过这样的总结：

> 二十年来，开发新编程语言的一个流行的秘诀是：取 C 语言的计算模式，逐渐地往上加 LISP 模式的特性，例如运行时类型和无用单元收集。

然而这毕竟只是“创生一种新语言”的魔法。那么，到底有没有让我们在这浩如烟海的语言家族中，找到学习方法的魔法呢？

我的答案是：看清语言的本质，而不是试图学会一门语言。当然，这看起来非常概念化。甚至有人说我可能是从某本教材中抄来的，另外一些人又说我试图在这本书里宣讲类似于我那本《大道至简》里的老庄学说[4]。

其实这很冤枉。我想表达的意思不过是：如果你想把一副牌理顺，最好的法子，是回到它

---

[1] 参见《人月神话》，美国弗雷德里克•布鲁克斯（Frederick P. Brooks, Jr.）著。
[2] 参见《蓝精灵》，比利时皮埃尔•居里福特（Pierre Culliford，Peyo）著。
[3] 保罗•格雷厄姆是计算机程序语言 Arc 的设计者，著有多本关于程序语言，以及创业方面的书籍。
[4] 这是一本软件工程方面的书，但往往被人看成是医学书籍或有人希望从中求取养生之道。

的分类法上，要么从 A 到 K 整理，要么按四个花色整理[5]。毕竟，两种或更多种分类法作用于同一事物，只会使事物混淆而不是弄得更清楚。

因此，本书从语言特性出发，把动态与静态、函数式与非函数式的语言特性分列出来。先讲述每种特性，然后再讨论如何去使用（例如交叉）它们。

特性

无论哪种语言（或其他工具）都有其独特的特性，以及借鉴自其他语言的特性。有些语言通体没有“独特特性”，只是另外一种语言的副本，这更多的时候是为了“满足一些人使用语言的习惯”。还有一些语言则基本上全是独特的特性，这可能导致语言本身不实用，但却是其他语言的思想库。

我们已经讨论过这一切的来源。

对于 JavaScript 来说，除了动态语言的基本特性之外，它还有着与其创生时代背景密切相关的一些语言特性。直到昨天[6]，JavaScript 的创建者还在小心翼翼地增补着它的语言特性。JavaScript 轻量的、简洁的、直指语言本实的特性集设计，使它成为解剖动态语言的有效工具。这个特性集包括：

- 一套参考过程式语言惯例的语法；
- 一套以原型继承为基础的对象系统；
- 一套支持自动转换的弱类型系统；
- 动态语言与函数式语言的基本特性。

需要强调的是，JavaScript 1.x 非常苛刻地保证这些特性是相应语言领域中的最小特性集（或称之为“语言原子”），这些特性在 JavaScript 中相互混合，通过交错与补充而构成了丰富的、属于 JavaScript 自身的语言特性。

本书的主要努力之一，就是分解出这些语言原子，并重现将它们混合在一起的过程与方法。通过从复杂性到单一语言特性的还原过程，让读者了解到语言的本实，以及“层出不穷的语言特性”背后的真相。

技巧

技巧是“技术的取巧之处”，所以根本上来说，技巧也是技术的一部分。很多人（也包括我）反对技巧的使用，是因为难以控制，并且容易破坏代码的可读性。

哪种情况下代码是需要“易于控制”和“可读性强”的呢？通常，我们认为在较大型的工程下需要“更好的控制代码”；在更多人共同开发的项目代码上要求“更好的可读性”。然而，反过来说，在一些更小型的、不需要更多人参与的项目中，“适度的”使用技巧是否就可以接受呢？

这取决于“需要、能够”维护这个代码的人对技巧的理解。这包括：

- 技巧是一种语言特性，还是仅特定版本所支持或根本就是 BUG；

[5] 不过这都将漏掉了两张王牌。这正是问题之所在，因为如果寻求“绝对一分为二的方法”，那么应该分为“王牌”和“非王牌”。但这往往不被程序员或扑克牌玩家们采用，因为极端复杂性才是他们的毕生目标。

[6] 在 JavaScript 2——这种把银弹涂上毒剂以试图用单发手枪击杀恐龙的构想发布之前的“昨天”。

- 技巧是不是唯一可行的选择，有没有不需要技巧的实现；
- 技巧是为了实现功能，而不是为了表现技巧而出现在代码中的。

即使如此，我仍然希望每一个技巧的使用都有说明，甚至示例。如果维护代码的人不能理解该技巧，那么连代码本身都失去了价值，更何论技巧存在这份代码中的意义呢？

所以本书中的例子的确要用到许多“技巧”，但我一方面希望读者能明白，这是语言内核或框架内核实现过程中必须的，另一方面也希望读者能从这些技巧中学习到它原本的技术和理论，以及活用的方法。

然而对于很多人来说，本书在讲述一个完全不同的语言类型。在这种类型的语言中，本书所讲述的一切，都只不过是“正常的方法”；在其他类型的一些语言中，这些看起来就成了技巧。例如在 JavaScript 中要改变一个对象方法指向的代码非常容易，并且是语言本身赋予的能力；而在 Delphi/C++中，却成了“破坏面向对象设计”的非正常手段。

所以你最好能换一个角度来看待本书中讲述的“方法”。无论它对你产生多大的冲击，你应该先想到的是这些方法的价值，而不是它对于“你所认为的传统”的挑战。事实上，这些方法，在另一些“同样传统”的语言类型中，已经存在了足够长久的时间——如同“方法”之与“对象”一样，原本就是那样“（至少看起来）自然而然”地存在于它所在的语言体系之中。

语言特性的价值依赖于环境而得彰显。横行的螃蟹看起来古怪，但据说那是为了适应一次地磁反转。螃蟹的成功在于适应了一次反转，失败（我们是说导致它这样难看）之处，也在于未能又一次反转回来。

这本书

你当然可以置疑：为什么要有这样的一本书？是的，这的确是一个很好的问题。

首先，这本书并不讲 Web 浏览器（Web Browser，例如 Internet Explorer）。这可能令人沮丧。但的确如此。尽管在很多人看来，JavaScript 就是为浏览器而准备的一种轻量的语言，并认为它离开了 DOM、HTML、CSS 就没有意义。在同样的“看法”之下，国内外的书籍在谈及 JavaScript 时，大多会从“如何在 Web 页面上验证一个输入框值的有效性”讲起。

是的，最初我也是这样认为的。因为本书原来就是出自我在写《B 端开发》一书的过程之中。《B 端开发》是一本讲述“在浏览器（Browser）上如何用 JavaScript 开发”的书。然而，《B 端开发》写到近百页就放下了，因为我觉得应该写一本专门讲 JavaScript 的书，这更重要。

所以，现在你将要看到的这本书就与浏览器无关。在本书中我会把 JavaScript 提升到与 Java、C# 或 Delphi 一样的高度，来讲述它的语言实现与扩展。尽管在本书的最后一部分内容中，讲述了名为“Qomo”的完整的 JavaScript 框架，这有助于你在浏览器上构建大型应用[7]。但是，嗯，本书不讲浏览器，不讲 Web，也并不讲“通常概念下的”AJAX。

JavaScript 是一门语言，有思想的、有内涵的、有灵魂的语言。如果你不意识到这一点，那么你可能永远都只能拿它来做那个“验证一个输入框值的有效性”的代码。本书讲述 JavaScript 的这些思想、核心、灵魂，以及如何去丰富它的血肉。最为核心的内容是在第 2 至 6 章，包括：

- 以命令式为主的一般化的 JavaScript 语言特性，以及其对象系统；

[7] 如果你试图构建基于 Web 的大型应用（例如基于 AJAX 的工程），那么你可以从 Qomo 中得益良多。它可以成倍地提高你的开发工效，有利于你实现更多的、更有价值的应用特性。（广告结束）

- 动态、函数式语言，以及其他语言特性在 JavaScript 的表现与应用；
- 使用动态函数式特性来扩展 JavaScript 的特性与框架。

在撰述这些内容的整个过程中，我一直在试图给这本书找到一个适合的读者群体，但我发现很难。因为通常的定义是低级、中级与高级，然而不同的用户对自己的“等级”的定义标准并不一样。在这其中，有“十年学会编程”的谦谨者，也有“三天学会某某语言”的速成家。所以，我认为这样定位读者的方式是徒劳的。

如果你想知道自己是否适合读这本书，建议你先看一下目录，然后试读一二章节，可以先选读一些在你的知识库中看来很新鲜的，以及一些你自认为已经非常了解的内容。通过对比，你应该知道这本书会给你带来什么。

不过我需要强调一些东西。这本书不是一本让你“学会某某语言”的书，也不是一本让初学者“学会编程”的书。阅读本书，你至少应该有一点编程经验（例如半年至一年），而且要摈弃某些偏见（例如 C 语言天下无敌或 JavaScript 是新手玩具）。

最后，你至少要有一点耐心与时间。

CHAPTER

第 1 章

十年 JavaScript

几乎每本讲 JavaScript 的书都会用很多的篇幅来讲 JavaScript 的源起与现状。本书也需要这样的一个开篇吗？

不。我虽然也想过这样，但我不打算让读者去读一些能够从 Wiki 中摘抄出来的文字，或者在不同的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，也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，甚至显得是那样的不足以道。因此除了故事性与可读性之外，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，中国可能还有 2/3 的 IT 人在玩一种叫“电子公告板（BBS，Bulletin Board System）”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。而另外的 1/3 可能就已经开始了互联网之旅，知道了主页（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。当时的网页开发的环境并不好（像现在的 Dreamweaver 这种东西，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，例如搜索什么的。我笑着说：如果在电子公告板（PC-BBS）上，写段脚本就可以了；但在互联网上面，却要做很多的工作。

我事实上并不知道要做多少的工作。我随后查阅的资料表明：我们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。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”们，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。而我在后来（1997 年）被调到成都，终于有更多的机会接触 Internet 网络，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，大量的新东西被很快传递进来。我终于有机会了解一些新的技术名词，例如 CSS 和 JavaScript。HTML 4.0 的标准已经确定（1997 年 12 月），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）而一统天下的形势。除了这些，我还对在 Delphi 中进行 ISAPI CGI 和 ISAPI Filter 的开发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，我调回到河南郑州，成为一名专职程序员，任职于当时的一家反病毒软件公司，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[8]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 程序和其他的一些服务器端代码。但我并不满足这些，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”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”。

[8] 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。

但是在当时，在 IE 中显示 Java Applet 之前需要装载整个的 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`，我就可以结束 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。因为我得不到 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[9]，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 Netscape Navigator 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。于 1995 年 4 月，他们招募了 Brendan Eich，希望 Brendan Eich 来实现这样的一种语言，以“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 最早被作为一种“浏览器上的脚本语言”给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[10]。

而在这时，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，

[9] 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）。

[10] 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

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”[11]。从这种定位来看，最初的 JavaScript 一定程度上是为了解决浏览器与服务器之间统一开发而被实现的一种语言。

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 带来的诱惑，一切都发生了改变。

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。

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

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。而服务器则使用了 Delphi 来开发的 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：我要做一个聊天室。因为在我的个人网站中，包括论坛、BBS 等都有网站免费提供，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（我称为 beta 0）；又一个月后，这个聊天室的 beta 1 终于在互联网上架站运行（见图 1-1）。

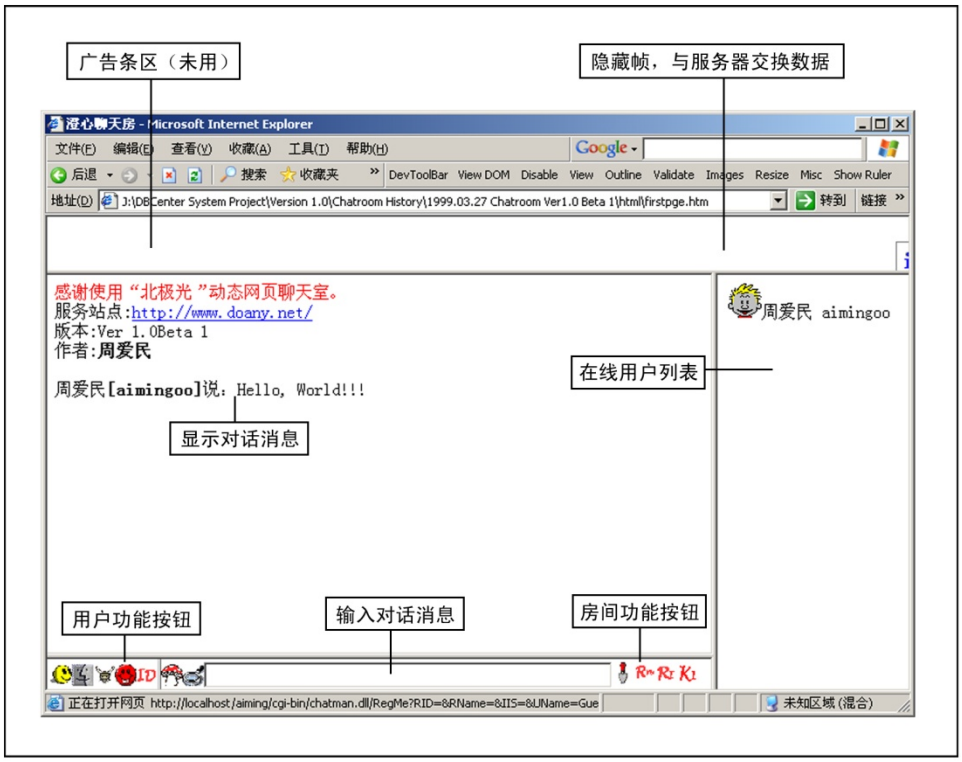


图 1-1　聊天室的 beta 1 的界面

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
	照片上传并显示给所有人	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 6，对话框
	按照 ID/名字查找用户	用户功能按钮 6，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。当时还没有 PHP，而 ASP 也只是刚刚出现，并不成熟。

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，例如 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”也因为市场不明朗一直不能投入开发。

而在服务器端的 `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 的示例中，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（DHTML，Dynamic HTML）”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。

在开始，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，以致于后来由 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，在 WebFX 上公布他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（见图 1-3）。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。

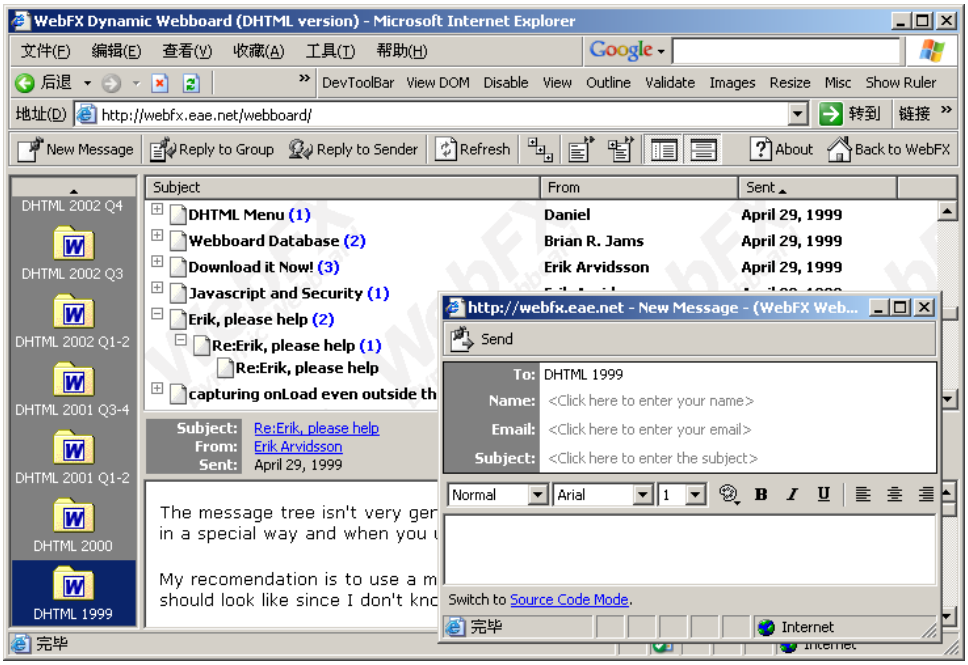


图 1-3　WebFX Dynamic WebBoard 的仿 Outlook 界面

而在这时盛行的 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）”的市场的局面出现了——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（例如 Virtual Studio .NET）和语言标准（例如 CLS，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。另一方面，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。

但 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 Web 开发基础标准。而在这整个的过程中，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：我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。

于是，整个的 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）。

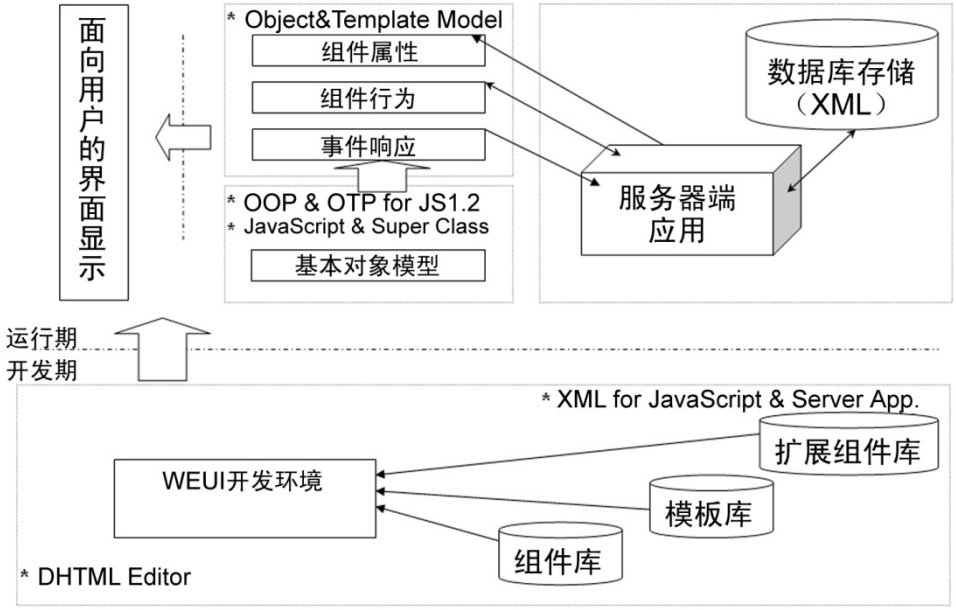


图 1-4　WEUI 基本框架和技术概览

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（OOP，Object Oriented Programming）”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[12]的开源项目工作中，提出了名为“SuperClass”的概念和原始代码。后来，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。

后来许多的 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。

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《王朝复辟还是浴火重生——The Return of Rich Client》。这篇文章讲的就是 Bindows（见图 1-5）。

[12] 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。

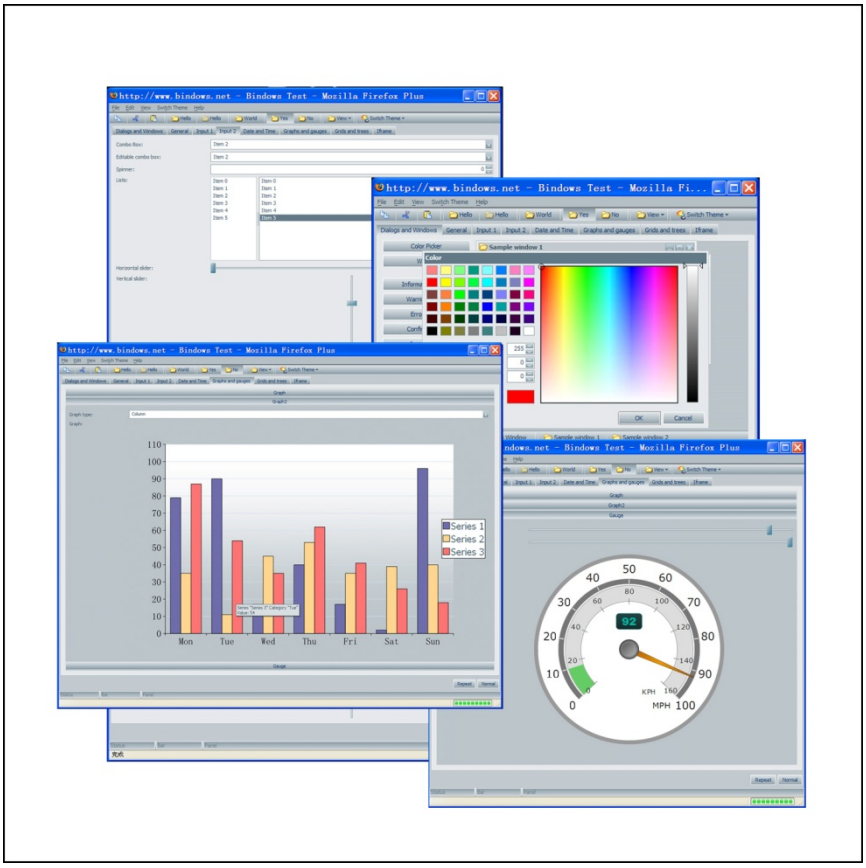


图 1-5　Bindows 在浏览器上的不凡表现

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（Rich Web Client）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[13]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，我就着手以 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

[13] 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。

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。然而这一个分析的过程，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 的有力支持[14]，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 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[15]等。这些过程贯穿于整个的 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（RWC）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 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

[14] 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。
[15] hax 的全名是贺师俊，他领导着一个名为 PIES 的 JavaScript 开源项目。

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的技术带来的巨大的网络机会。

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 Internet Explorer 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。JSRS 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，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、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 为 JavaScript 正名

至 2005 年，JavaScript 就已经诞生十年了。然而十年之后，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，前者却用了十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）[16]”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[17]。

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）”。一门被误会了十年的语言的名字之争，是不是就此结束了呢？

仍然不是。因为这十年来，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。

1.4.1 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。

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

[17] 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/

- 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（Netscape Navigator）与 NWS（Netscape Web Server）上的实现——但后来他们改变了这个做法。

1.4.2 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 1-6 所示。

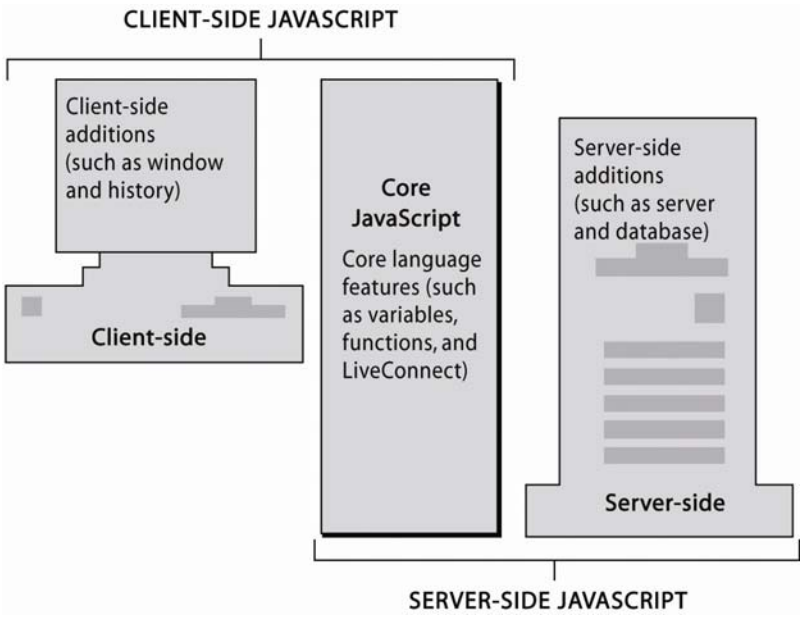


图 1-6　官方手册中有关 Core JavaScript 的概念说明

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。

然而不幸的是，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。

1.4.3 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，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。目前 SpiderMonkey JavaScript 已经发布了 1.7 版本的项目代码。

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

1.4.4 ECMAScript

JavaScript 的语言规范由网景公司提交给 ECMA（European Computer Manufacturing Association，欧洲计算机制造协会）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。四个月后，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”。

1.4.5 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 1-7 所示。

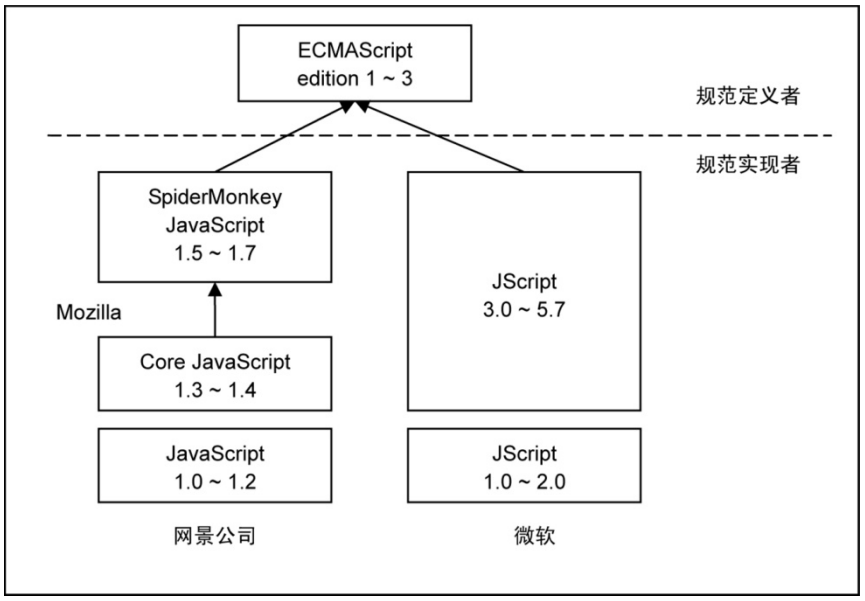


图 1-7　JScript 与 JavaScript 各版本间的关系

然而由于 JScript 成为 JavaScript 语言的事实标准，再有 Internet Explorer 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——亦即是 ECMAScript 的标准规范，这成为了新一轮语言之争的起点。

1.4.6 总述

JavaScript 这个名词的多种含义如表 1-2 所示。

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。概含 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 Cliet-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 Internet Explorer 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

1.5　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 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新的语言特性了[18]。

JavaScript 的应用环境，主要由宿主环境与运行期环境构成。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 1-8 说明由它们共同构建的对象编程系统的基本结构：

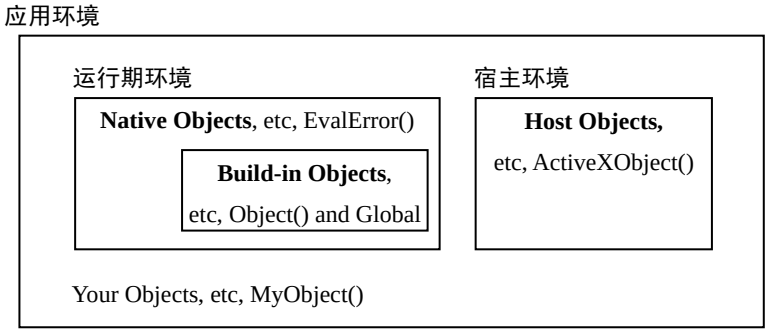


图 1-8　由宿主与运行期构成的应用环境

[18] 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，因而难于胜任大型编程系统环境下的开发。

1.5.1 宿主环境（host environment）

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于做到，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[19]。

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	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, ...)`	输出一个段文本，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）。

[19] 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!');
```

1.5.2 外壳程序（Shell）

外壳程序是宿主的一种。不过在其他的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 1-9 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”是指整个的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（例如注册表、文件系统等）的一个“外壳对象”，而非“外壳程序”。

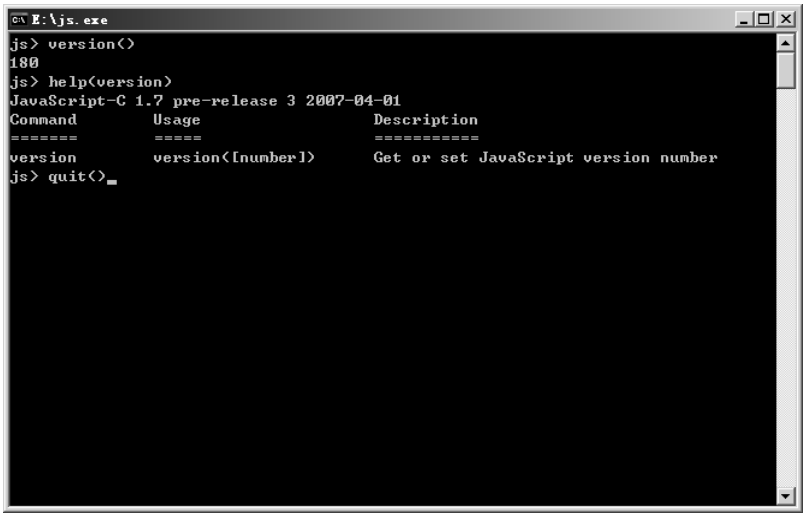


图 1-9　SpiderMonkey JavaScript 提供的外壳程序

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）”。

1.5.3　运行期环境（runtime）

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（见图 1-10）。

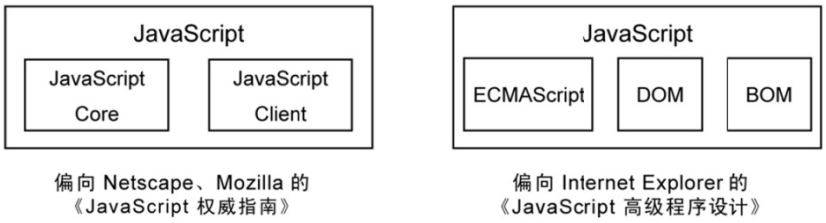
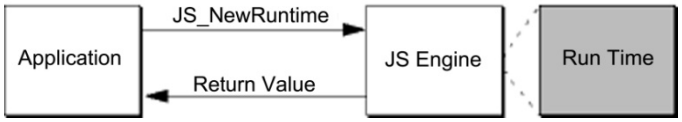


图 1-10　对“运行期环境”的不同解释

本书是从引擎的角度讨论 JavaScript，因此在本书看来，与浏览器相关的内容都是属于“应用环境”：属于宿主环境或属于用户编程环境。本小节开始位置的“图 1-8　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[20]。图 1-11 说明应用程序——宿主在这里可以看成一个应用程序——如何创建运行期环境[21]。



[20] 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。

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

图 1-11　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[22]：

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 Internet Explorer
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。

[22] 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。

CHAPTER

第 4 章

JavaScript 的函数式语言特性

函数式程序设计始于 LISP。LISP 的程序和数据都用表来表示，甚至这种语言的名字本身也就是“表处理系统（List Processor）”的缩写形式。著名的函数式语言 Scheme 是 LISP 的一种方言。

——《程序设计语言概念和结构》，Ravi Sethi

4.1 概述

通常来讲，函数式语言被认为是基于“数学函数”的一种语言。当我们开始用数学领域中的抽象概念来解释函数式语言时，问题被放大（或缩小、聚焦）为下面两个描述[23]：

- 数学函数是集合 A（称为定义域）中成员到集合 B（称为值域）中成员的映射；
- 函数式程序设计是通过数学函数的定义、应用的说明和求值完成运算过程的。

第一句话基本上等于什么都没说，它的含义完全等同于“函数＝从问题中找到答案”。而第二句话的“定义和应用说明”基本上等于第一句话，所以相当于说：函数式程序设计是“计算函数”——还是等于什么也没有说。

但是这些古怪的文字的确是在阐述函数式语言的精髓。为了减轻你的痛苦（但绝非轻视你的智商），我换个说法来陈述它们：如果表达式“1+1=2”中的“+”被理解为求值函数，那么所谓函数式语言，就是通过连续表达式运算求值的语言；既然上面的表达式可以算出结果“=2”，那么函数式语言自然也可以通过不停地求值找到问题的答案。

[23] 基本概念引用自《程序设计语言原理》（Robert W. Sebesta 著），但并未复录原文的概念陈述。

4.1.1 从代码风格说起

在一些语言中，连续运算被认为是不良好的编程习惯。我们被要求运算出一个结果值，先放到中间变量中，然后拿中间变量继续参与运算。

其中的原因之一，在于容易形成良好的代码风格。这个原因被阐释得非常多。例如我们被教育说，不应该这样写代码[24]：

```
child = (!LC && !RC) ? 0 : (!LC ? RC : LC);
```

而应该把它写成下面这样：

```
if (LC ==0 && RC == 0)
  child = 0;
else if (LC == 0)
  child = RC;
else
  child = LC;
```

我承认我们应该写更良好风格的代码，我也曾经深受自己代码风格不良好之苦并幡然醒悟。但是上面这个问题的本质，真的是“追求更漂亮的代码风格（style）”吗？

例如我曾经有一个困扰，就是如何写 LISP/Scheme 的代码，才会有“更良好的风格”？下面这段代码是一段 LISP 语言的示例：

```
; LISP Example function
(DEFUN equal_lists ( lis1 lis2)
  (COND
    ((ATOM lis1) (EQ lis1 lis2))
    ((ATOP lis2) NIL)
    ((equal_lists (CAR lis1) (CAR lis2))
      (equal_lists (CDR lis1) (CDR lis2)))
    (T NIL)
  )
)
```

然而答案是：没有比上面这个示例更良好的 LISP 语言风格了（当然，你愿意用四个空格替换两个空格，或者把括号写在一行的后面之类，是一种习惯而非“更良好风格”的必要前提）。由此看来：不同语言中所谓的“良好风格”看起来是并没有统一标准的。

所以说，语言风格的好坏只是判断“是否连续运算”的一个并不要重要的方面。

4.1.2 为什么常见的语言不赞同连续求值

在另一个方面，“不支持连续运算”这种编程习惯（和代码风格）其实是为了更加符合冯·诺依曼的计算机体系的设计。在这一体系的程序设计观念中，我们应这样写代码：

[24] 《程序设计实践》Brian W. Kernighan 和 Rob Pike 著，裘宗燕译。

```
var desktop = new Destktop();
var chair1 = new Chair();
var chair2 = new Chair();
var me = new Man();

var myHome = new Home();
myHome.concat(desktop);
myHome.concat(chair1);
myHome.concat(chair2);
myHome.concat(me);
myHome.show(room);
```

看看，我们费尽心力才创建了一个有桌子、椅子和人的房子，并进而有了个家，但这个家的简陋条件，实在是比监狱还差。然而我们已经付出了如此多的代码（还不包括那些类的声明与实现），因此我们如果要创建一个更加漂亮而有生气的家，上面这样的代码我们得写很多年。

为什么我们要这样写代码呢？因为我们从面向过程、面向对象一路走来，根本上就是在冯·诺依曼的体系上发展。在这个体系上，我们首先就被告知：运算数要先放到寄存器里，然后再参与 CPU 运算。于是我们得到了结论，汇编语言应该这样写：

```
MOV EAX, $0044C8B8
CALL @InitExe
```

接下来，我们就看到过程式语言这样写：

```
var
  value_1: integer;
  value_2: integer;

begin
  value_1 := 100;
  value_2 := 1000;
  writeln(value_1 * value_2);
end.
```

然后，我们就看到了面向对象的语言应该这样写：

```
var
  value_1: TIntegerClass;
  value_2: TIntegerClass;
var
  calc : TCalculator;

begin
  calc := TCalculator.Create();
  value_1 := TIntegerClass.Create(100);
  value_1 := TIntegerClass.Create(1000);

  calc.calc(value_1, value_2);
  calc.show();
end.
```

在冯·诺依曼体系下，我们就是这样做事的。所以在《程序设计语言——实践之路》这本书中，将面向对象与面向过程都归类为“命令式”语言，着实不妄。

综合上一小节的讨论来看，一方面，冯·诺依曼体系对存储的理解从根本上规范了我们的代码风格；另一方面，语言环境是风格限定与编程习惯形成的重要前提。

因此对于一种语言来说，某种风格可能是非常漂亮的，但对于另一种来说，可能根本就无法实现这种风格。从形式上讲，如果我们以过程式代码的风格来看 LISP 代码，那么除了还存有缩进之外，几乎毫无美观之处。

然而，事实上只有这种风格才能满足函数式语言的特性设定——因此问题的根源并不在于“代码是否更加漂亮”，而是 LISP——这种函数式语言——本身的某些特性需要“这样一种”复杂的代码风格，如同冯·诺依曼体系需要“那样一种”风格一样。

4.1.3 函数式语言的渊源

上述 LISP（这种函数式语言）的代码风格所表达的基本语言特征之一就是连续运算：运算一个输入，产生一个输出，输出的结果即是下一个运算的输入。在连续运算过程中，无需中间变量来“寄存”。因此从理论上来说：函数式语言不需要寄存器或变量赋值。

然而为什么“连续求值”会成为函数式语言的基本特性呢？或者说，这些影响到函数式语言的代码风格的特性是什么呢？要了解这一问题的实质，需要更远地回溯“函数式”语言的起源。我们得先回答一个问题：

这种语言是如何产生的呢？

1930 年前后，在第一台电子计算机还没有问世之前，有四位著名的人物展开了对形式化运算系统的研究。他们力图通过这种所谓的“形式系统”，来证明一个重要的命题：可以用简单的数学法则表达现实系统。这四个人分别是阿兰·图灵、约翰·冯·诺依曼、库尔特·哥德尔和阿隆左·丘奇。

在 1936 年，图灵提出了现在称为“图灵机”的形式系统。图灵机概念中提出了通过 0、1 运算系统来解决复杂问题。接下来，在 1939 年，阿坦纳索夫研制成功第一台电子计算机 ABC，其中采用了电路开合来代表 0、1 ，运用电子管和电路执行逻辑运算。再接下来，在 1945 年，冯·诺依曼等人基于当时计算机系统 ENIAC（Electronic Numerical Integrator And Computer，电子数字积分计算机）的研究成果，提出了 EDVAC 体系设计[25]，以及其上的编码程序、纸带存储与输入。该设计方案完全实现了图灵的科学预见与构思[26]。

[25] 《存储程序通用电子计算机方案——EDVAC（Electronic Discrete variable Automatic Computer，离散变量自动电子计算机）》是一份设计方案，而非（当时的）物理实现。EDVAC 方案直到 1950 年以后才被实现。

[26] 电子计算机的历史一直存在很多争议，如今这些争议已经被澄清。这一部分的文字请参见袁传宽教授在《人物》杂志 2007 年 10 月和 11 月期中的一组文章《计算机世界第一人——艾兰·图灵》和《被遗忘的计算机之父——阿坦纳索夫》。

我们现在最常见的通用编程环境，就是构架于冯·诺依曼在 EDVAC 中的设计，该设计包括五大部件：运算器 CA、逻辑控制器 CC、存储器 M、输入装置 I 和输出装置 O。其中，运算器基于的理论是 0、1 运算，而存储器 M 和输入输出装置 I/O 则依赖于 0、1 存储。因此基于冯·诺依曼体系架构的程序设计语言，必然面临这样的物理环境——具有存储系统（例如内存、硬盘等）的计算机体系，并依赖存储（这里指内存）进行运算。后来有人简单地归结这样的运算系统：通过修改内存来反映运算的结果。

然而，我们应用计算机的目的，是进行运算并产生结果。所以其实运算才是本质，而“修改内存”只不过是这种运算规则的“副作用”，或者说是“表现运算效果的一种手段”。因此相对于基于图灵机模型提出的运算范型，阿隆左·丘奇所提出的运算系统更加趋近“运算才是本质”观点。

这是一种被称为 Lambda 演算的形式系统。这个系统本质上就是一种虚拟的机器的编程语言——而不是虚拟的机器，它的基础是一些以函数为参数和返回值的函数[27]。注意，我们在这里一定要强调“基础是一些‘以函数为参数和返回值’的函数”这一特性。

这种运算模式却一直没有被实现。大约在冯·诺依曼等人的 EDVAC 报告提出的十年之后，一位 MIT 的教授 John McCarthy[28]对阿隆左·丘奇的工作产生了兴趣。在 1958 年，他公开了表处理语言 LISP。该语言其实就是对阿隆左·丘奇的 Lambda 演算的实现。

但是，这时的 LISP 工作在冯·诺依曼计算机上！——很明显，这时只有这样的计算机系统——更加准确地说，LISP 系统当时是作为 IBM 704 机器上的一种解释器而出现的。

所以从函数式语言的鼻祖——LISP 开始，函数式语言就是运行在解释环境而非编译环境中的。而究其根源，还在于冯·诺依曼体系的计算机系统是基于存储与指令系统的，而并不是基于（类似 Lambda 演算的）连续运算的。

函数式语言强调运算过程，这也依赖于运行该系统的平台的运算特性。由于我们的确是将计算机设计成了冯·诺依曼的体系，所以在过去很长的时间里，你看不到一个计算机（硬件）系统宣称在机器指令级别上支持了函数式语言。直到 1973 年，MIT 人工智能实验室的一组程序员开发了被称为“LISP 机器”的硬件。

阿隆左·丘奇的 Lambda 演算终于得以硬件实现！

现在让我们回到最初的话题：为什么可以将语言分成命令式和说明式语言？是的，从语言学分类来说，这是两种不同类型的计算范型；从硬件系统来说，它们依赖于各自不同的计算机系统。如同函数式与命令式语言，这些分类之间存在着本质的差异。

---

[27] 《函数式编程另类指南》（Functional Programming For The Rest of Us），Vyacheslav Akhmechet 著，lihaitao 译。

[28] John McCarthy 被称为人工智能之父，是 1971 年（第 6 届）图灵奖得主。

然而现在我们每个人手中的电脑毕竟都不是名为“LISP 机器”的硬件——支持大量“运算函数”的 RISC（复杂指令集）已经失败了，精简指令集带来了更少的指令和更确切的运算法则：放到寄存器里，然后再交由 CPU 运算。我们不能寄期望一种基于 A 范型实现的计算机系统同时（在物理特性上的、完美的）支持 B 范型。换言之，不能指望在 X86 指令集中出现适宜于 Lambda 演算的指令、逻辑或者物理设计。

于是当前的现实变成了这样：我们大多数人都在使用基于冯·诺依曼体系的命令式语言，但为了获得特别的计算能力或者编程特性，这些语言也在逻辑层来实现一种适宜于函数式语言范型的环境。这一方面产生了类似于 JavaScript 这样的多范型语言，另一方面则产生了类似于.NET 或 JVM 的、能够进行某些函数式运算的虚拟机环境[29]。

4.2 函数式语言中的函数

并不是一个语言支持函数，这个语言就可以叫做“函数式语言”。函数式语言中的“函数（function）”除了能被调用之外，还具有一些其他的性质。这包括：

- 函数是运算元；
- 在函数内保存数据；
- 函数内的运算对函数外无副作用。

我们下面分述函数在 JavaScript 中的这三种特性。

4.2.1 函数是运算元

大多数语言都支持将函数作为运算元参与运算。不过由于对函数的理解不同，因此它们的运算效果也不一样。例如在 C、Pascal 这些命令式语言中，函数是一个指针，对函数指针的运算可以包括赋值、调用和地址运算。由于这种情况下函数被理解为指针，因此也可以作为函数参数进行传值（地址值），比较常见的情况是函数 A 的声明中，允许传出一个回调函数 B 的指针。下例是这样一个 Win32 API 的声明：

```
/**
 * Pascal语言声明的EnumWindows()
 */
function EnumWindows(lpEnumFunc: EnumWindowsProc;
  lParam: LPARAM): BOOL; stdcall

/**
 * C语言声明的EnumWindows()
 */
BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

[29] 自.NET 3.0 开始，C#开始支持 Lambda 表达式特性；而 JVM 中，则要等到 Java 7 以后。

由于这里的 lpEnumFunc 将传入一个地址指针，这个地址显然可能来自另一个进程空间，或者当前进程无效的存储地址。因此这种函数调用过程中，以地址值为数据的参数传递，大大增加了系统的风险。同时，基于地址指针值进行的运算，也带来了“内存访问违例”的隐患。

当 JavaScript 中的函数作为参数时，也是传递引用的，但并没有地址概念。由于彻底地杜绝了地址运算，也就没了上述的隐患。“函数调用”实质上是一个普通的运算符，因此所谓“传入参数”可以被理解为运算元。由此的结论是，（作为“传入参数”的）函数只有运算元的含义而没有地址含义，“函数参数”与普通参数并没有什么特别不同。

4.2.2 在函数内保存数据

函数式语言的函数可以保存内部数据的状态。在某些命令式语言中也有类似的性质，但与函数式语言存在根本不同。以（编译型、X86 平台上的）命令式语言来说，由于代码总是在代码段中执行，而代码段不可写，因此函数中的数据只能是静态数据。这种特性通常与编译器或某些特定算法的专用数据绑定在一起（例如跳转表）。

除了这种情况之外，在命令式语言中，函数内部的私有变量（局部变量）是不能被保存的。从程序执行的方式来讲，局部变量在栈上分配，在函数执行结束后，所占用的栈被释放。因此函数内的数据不可能被保存。

在 JavaScript 的函数中，函数内的私有变量可以被修改，而且当再次“进入”到该函数内部时，这个被修改的状态仍将持续。下面的例子中，函数 `set_value()`用于修改值，函数 `get_value` 用于获取值，我们将这两个函数置入到 `MyFunc()`内部，用来考察该函数内部的数据状态：

```
var set, get;

function MyFunc() {
  // 初值
  var value = 100;

  // 内部的函数, 用于该问 MyValue
  function set_value(v) {
    value = v;
  }
  function get_value() {
    return value;
  }

  // 将内部函数公布到全局
  set = set_value;
  get = get_value;
}
```

```
// 测试一
// 显示输出值: 100
MyFunc();
alert( get() );

// 测试二
// 显示输出值: 300
set(300);
alert( get() );
```

测试一表明：函数 `MyFunc()`在执行结束后，内部数据值仍保持它的状态。而测试二则表明：`set_value()`将影响到内部数据，这种影响后的状态也被保持。

在函数内保持数据的特性被称为“闭包（Closure）”，我们将在“4.6　闭包”中更详细地讨论它。不过显而易见的好处是，如果一个数据能够在函数内持续保存，那么该函数（作为构造器时）赋给实例的方法就可以使用这些数据进行运算；而在多个实例间，由于数据存在于不同的闭包中，因此不会产生相互影响——以面向对象的术语来解释，就是说不同的实例拥有各自的私有数据（复制自某个公共的数据），多个实例之间不存在可共享的类成员。下例说明这个特性：

```
function MyObject() {
  var value = 100;
  this.setValue = function(v) {
    value = v;
  }
  this.showValue = function() {
    alert(value);
  }
}
var
  obj1 = new MyObject();
  obj2 = new MyObject();

// 测试: obj2 的置值不会影响到 obj1
// 显示结果值: 100;
obj2.setValue(300);
obj1.showValue();
```

4.2.3　函数内的运算对函数外无副作用

运算对函数外无副作用，是函数式语言应当达到的一种特性。然而在 JavaScript 中这项特性只能通过开发人员的编程习惯来保证。

所谓运算对函数外无副作用，含义在于：

- 函数使用入口参数进行运算，而不修改它（作为值参数而不是变量参数使用）；
- 在运算过程中不会修改函数外部的其他数据的值（例如全局变量）；
- 运算结束后通过函数返回向外部系统传值。

这样的函数在运算过程中对外部系统是无副作用的。然而我们注意到 JavaScript 允许在函数内部引用和修改全局变量，甚至可以声明全局变量。这一点其实是破坏它的函数式特性的。除此之外，JavaScript 也允许在函数内修改对象和数组的成员。这使得函数并不是仅仅通过它的返回值来影响系统，因此也不是正确的函数式特性。

所以在 JavaScript 中，只能通过开发人员的习惯来实现这一特性。这包括两点：不修改全局变量，以及不在函数内修改对象和数组成员——这些成员应该由对象方法而非对象系统外的其他函数来修改。

当把“不在函数内修改对象成员”这个原则，与面向对象系统的另一个特性结合起来的时

候，系统的稳定性就大大地增强了。这个特性就是通过接口（interface）向暴露系统，以及通过读写器（get&setter）访问对象属性（attribute）。由于在这种对象系统中，对象向外部系统展现的都是接口方法（以及读写器方法），从而有效地避免了外部系统“直接修改对象成员”。

在这里补充面向对象系统的这一特性，是强调函数式中的“函数”所要求的“无副作用”这个特性，其实可以与面向对象系统很好地结合起来。二者并不矛盾，在编程习惯上也并非格格不入。

4.3 从运算式语言到函数式语言

现在让我们回到最开始的话题：为什么“连续求值”会成为函数式语言的基本特性呢？这是因为函数式语言是基于对 Lambda 演算的实现而产生的，其基本运算模型就是：

- （表达式）运算产生结果；
- 结果（值）用于更进一步的运算。

至于从 LISP 开始引用的“函数”这个概念，其实在演算过程中只有“结果（值）”的价值：它是一组运算的封装，产生的效果是返回一个可供后续运算的值。因此我们应该认识到，函数式语言中所谓的“函数”并不是真正的精髓，真正的精髓在于“运算”，而函数只是封装“运算”的一种手段。

到了这里，我们如果“假设系统的结果只是一个值”，那么“我们必然可以通过一系列连续的运算来得到这个值”。不过，这句话要分成两半来看，我们下面先讲“连续运算”，然后再来讨论“结果只是一个值”的问题。

4.3.1 JavaScript 中的几种连续运算

4.3.1.1 连续赋值

在 JavaScript 中，一种常见的情况就是连续赋值：

```
var a = b = c = d = 100;
```

我们把它写成下面这种格式，可能会让人更能理解（我并不是想说明这种语法风格更好。当然，如果你想要为每个变量做注释，可能这是个不错的主意）：

```
1  var a =
2      b =
3      c =
4      d = 100;
```

第 4 行的表达式“`d = 100`”被首先运算。因为表达式有返回值，所以得到了运算结果“`100`”。接下来，该值参与下一个赋值表达式运算，变成了“`c = 100`”。如此类推，我们得到了连续赋值的效果。

所以，在别的某些语言中（例如 Pascal），连续赋值可能是一种“新奇的语法特性”，但在 JavaScript 语言中，它不过是一种连续表达式运算的效果。

4.3.1.2 三元表达式的连用

我们前面提到过三元表达式（`?:`），这个表达式在 C 语言里是一种并不非常推荐的语句。因此对于刚才那个例子：

```
child = (!LC && !RC) ? 0 : (!LC ? RC : LC);
```

总有人会在书籍中故意弱化或丑化这个表达式的表现。其实开始的那段代码，一般人并不会那样写，即使要写成三元表达式，也应该如下：

```
child = (LC || RC) ? ( LC ? LC : RC ) : 0;
```

在函数式语言中，三元表达式其实不但应该使用，甚至被推荐连用。因为这样能够充分发挥连续运算的特性：

```
1  var objType = _get_from_Input();
2  var cls = ((objType == 'String') ? String :
3           (objType == 'Array') ? Array :
4           (objType == 'Number') ? Number :
5           (objType == 'Boolean') ? Boolean :
6           (objType == 'RegExp') ? RegExp :
7           Object
8  );
9  var obj = new cls();
```

在这个例子里，我们也可以看到一种良好的代码书写风格（至于第 8 行的括号是放在第 7 行的最末或新起一行，可以看成一种习惯）。但我们充分利用了表达式求值的特性：第 2～6 行的每个三元表达式的第三个运算元，其实都是下一行运算的返回结果。

显然，“运算产生值并参与运算”这一特性，使得上述的代码成为可能。否则，你可能需要写下面这样的代码（当然，你可能现在仍旧认为下面这样的代码风格更漂亮）：

```
var objType = _get_from_Input();

switch (objType) {
  case 'String': {
    obj = new String();
    break;
  }

  case 'Number': {
    // ...
  }

  // ...
  default: {
    obj = new Object();
  }
}
```

一部分理解了面向对象编程的“多态性”的开发人员可能会主张下面的代码：

```
var cls;
var objType = _get_from_Input();

switch (objType) {
  case 'String': {
    cls = String;
    break;
  }

  case 'Number': {
    // ...
  }

  // ...
  default: {
    cls = Object;
  }
}

var obj = new cls();
```

熟悉模式的开发人员则不慌不忙地提出他们的观点：

```
// ...
// (对于不同的语言，以上省略 10~50 行类工厂的实现代码)

var objType = _get_from_Input();
var fac = new Factiory();
var cls = fac.getClass(objType);
var obj = new cls();
```

我们不必去评述这几种代码实现的优劣。就如同代码风格一样，在不同的体系之下，存在不同的评判标准。但现在的问题是：你在使用一门函数式语言，然而你的代码利用函数式语言的特性了吗？

4.3.1.3 一些运算连用

在前面这个三元表达式中的例子中，我们说明了行代码：

```
child = (!LC && !RC) ? 0 : (!LC ? RC : LC);
```

至少可以被改成如下的形式：

```
child = (LC || RC) ? ( LC ? LC : RC ) : 0;
```

然后我们又说，可以通过更好的代码格式化，来使得三元表达式连用的代码可读性得到提升。例如：

```
child = !( LC || RC ) ? 0
       :( LC ? LC
       : RC );
```

但事实上在 JavaScript 中，一些语法约定基本不需要用户开发人员写上面这样的代码。正如这个例子，我们无非是想要得到 `LC`、`RC` 和 `0` 值之一。这可以借助连续的逻辑“或（`||`）”运算来得到。上面的代码等效于：

```
child = LC || RC || 0;
```

这行代码中，等号右边的表达式的意思是说：

- 如果 `LC` 能被转换成逻辑“`true`”（值为真），则运算返回 `LC` 的实际值；否则，
- 如果 `RC` 值为真，则返回 `RC` 的实际值；否则，
- ……
- 直到表达式结束，返回表达式最后一个运算元的实际值。

而这样的运算结果，正是我们需要得到的 `child` 值。

这段代码其实也正好说明了 JavaScript 中逻辑运算的实质——逻辑“或”运算并非为布尔值专设。按照上面规则，对布尔值进行“或（`||`）”运算的效果，只是一个特例而已。

4.3.1.4 函数与方法的调用

在前面我们使用了一个不非常恰当的例子。因为孤立来看，我们要得到一个对象的类类型，并不需要用那样复杂的三元表达式，用类厂可能的确是不错的主意。但我说那是一个孤立的问题。因为我们忽视了另外一项 JavaScript 特性：对象的构造、函数与方法的调用等，本质上都是表达式运算，而非语句。

举例来说，我们可以用下面的代码完成对象的构造：

```
var obj = new ( (obj=='String') ? String : Object );
```

这行代码是用一个运算来作为 new 运算的入口参数——注意 new 不是语句的语法关键字，而是运算符。

所以在 JavaScript 中，我们事实上可以用下面的连续运算来完成上一小节的示例：

```
var obj = new (
  (objType == 'String') ? String :
  (objType == 'Array') ? Array :
  (objType == 'Number') ? Number :
  (objType == 'Boolean') ? Boolean :
  (objType == 'RegExp') ? RegExp :
  Object
);
```

这样，连续的一组三元表达式的运算结果，成为了 new 运算符的输入，而 new 运算符后面的一对括号“( )”，在这里起到的是强制运算符的作用。最后，new 运算的结果（构造一个对象实例），被作为赋值运算的运算元，然后赋给了变量 obj。

接下来的代码将会更加有趣。让我们对上面的代码做一点小修改：

```
10  alert(
11    (new (
12       (objType == 'String') ? String :
13       (objType == 'Array') ? Array :
14       (objType == 'Number') ? Number :
15       (objType == 'Boolean') ? Boolean :
16       (objType == 'RegExp') ? RegExp :
17       Object
18     )
19    ).toString()
20  )
```

是的，我故意将代码分隔成这个样子，以使你更清楚地看到运算的层次。我们看到 new 运算在第 9 行得到了运算结果：一个对象实例。然后它被一对强制运算符给包括了起来（第 2~10 行）。强制运算的结果还是返回该实例，——我们在这里只是需要取得一个语法上清晰的效果[30]——而该实例接下来就调用了一下方法 toString()。

如果方法调用不是表达式而是语句，那么上面这样的代码就不可能被写出来。所以，第 10 行代码的实质是，刚被创建的对象实例：

- 先通过点运算符“.”进行了一次对象属性 toString 存取，
- 然后通过运算符“( )”进行了一次方法调用。

这两次运算的结果，返回了对象的序列化值。接下来这个值被送入 alert()函数的参数表（第 1~11 行代码），最终显示输出。

[30] 这里的强制运算是不必须的。因为“.”运算的优先级低于“new”运算，所以这里并不存在歧义。本例中使用了加强制运算，只是为了更好地表示运算次序。另外，事实上“.”具有最低的优先级次序。

4.3.2 运算式语言

在本小节中，我们将讨论一种新的编程范型：运算式语言。它满足说明式语言的两个特性：一是陈述运算，二是求值。

不同的运算式语言的编程能力是不同的，在本节中我们将列举两个这种类型的语言。需要注意的是，它们一开始时并不是以一个语言范型出现的——而更像是某个体系中的小功能而已。

4.3.2.1 运算的实质，是值运算

将“值运算”换个说法，就是“求值”。如果说“运算的实质，就是求值”，那么大家会觉得顺理成章。但是，这里的“值”如果是指“值类型”的数据呢？

在该设问中，我们其实已经向程序设计语言的本质走得更近了一步。为了说明这一点，我们先来考察一下 JavaScript 的各种运算的结果类型。表 4-1 对此作出了完整的分析。

表 4-1　JavaScript 的各种运算的结果类型（目标）

分类	名称	符号	说明	运算元	目标
计算运算	加法	+	将两个数相加	number	number
	减法	-	对两个表达式执行减法操作		
	乘法	*	将两个数相乘		
	除法	/	将两个数相除并返回一个数值结果		
	取余	%	将两个数相除，并返回余数		
	递增	++	给变量加一		
	递减	--	将变量减一		
	一元正值	+			
	一元取反	-	表示一个数值表达式的相反数		
按位运算	按位与	&	对两个表达式执行按位与操作		
	按位左移	<<	将一个表达式的各位向左移		
	按位非	~	对一个表达式执行按位取非操作		
	按位或	|	对两个表达式指定按位或操作		
	按位右移	>>	将一个表达式的各位向右移，保持符号不变		
	按位异或	^	对两个表达式执行按位异或操作		
	无符号右移	>>>	在表达式中对各位进行无符号右移操作		

续表

分类	名称	符号	说明	运算元	目标
逻辑运算	逻辑与	&&	对两个表达式执行逻辑与操作	boolean	boolean
	逻辑非	!	对表达式执行逻辑非操作		
	逻辑或	||	对两个表达式执行逻辑或操作		
字串	连接	+	连接字符或字符串	string	string
函数	函数调用	( )	调用函数并返回结果值	function	(*注 1)
比较	比较	(章节 2.3.4)	返回比较结果	(任意)	boolean
赋值运算	赋值	=	将一个值赋给变量	(任意)	(*注 1)
	复合赋值	(章节 2.3.5)	运算并将结果值赋给变量	(值类型)	(值类型)
对象	对象构造	new	创建一个新对象	function	object
	对象检查	instanceof	返回一个 Boolean 值，表明对象是否为特定类的一个实例	object	boolean
	成员存取	[ ]或.	存取对象的成员	object (string) (标识符)	(*注 1)
	成员删除	delete	删除对象的属性，或删除数组中的一个元素		boolean
	成员检查	in	检查一个对象成员是否存在	object (string)	boolean
其他	typeof	typeof	返回运算元数据类型的字符串	(任意)	string
表达式逻辑	三元条件(*)	?:	根据条件执行两个表达式之一	boolean (表达式)	(*注 2)
	优先级	( )	包含运算符的执行优先级信息的列表	(表达式)	
	逗号	,	使两个表达式连续执行		
	void	void	避免一个表达式返回值		undefined

*注 1：取决于具体的值、变量或对象成员的数据类型。

*注 2：是表达式之间的运算关系，结果只对表达式产生影响。

在表 4-1 中，最令人惊讶的结论是：所有的运算都产生“值类型”的结果值[31]。正因为“运算都产生值类型的结果”，且“所有的逻辑语句结构都可以被消灭”，所以结论是：“系统的结果必然是值，并且可以通过一系列的运算来得到这一结果”[32]。

我们知道，计算机其实只能表达值数据。任何复杂的现象（例如界面、动画或模拟现实），在运算系统看来其实只是某种输出设备对数值的理解而已，运算系统只需要得到这些数值，至于如何展示，则是另一个物理系统（或其他运算系统）来负责的事情。

所以运算的实质其实是值的运算。至于像“指针”、“对象”这样抽象结构，在运算系统来看，其实只是定位到“值”以进行后续运算的工具而已——换言之，它们是不参与“求值”运算的。到这里，读者应该明白为什么表 4-1 中的结果类型“必然是值”了。

4.3.2.2 有趣的运算：在 Internet Explorer 和 J2EE 中

在 Internet Explorer 中，层叠样式表（CSS，Cascading Style Sheet）中会有一些运算过程，用于设定一些特殊的样式属性，例如颜色和 URL 地址：

[31] “注 1”所标示的几处运算，要么是使用引用类型来定位值，要么就是像赋值这样在本章中不讨论的运算。

[32] 这是一项重要的结论。尽管在这里没有展开讲述，但如果读者愿意了解一些计算系统基本模型方面的知识，可以从该项结论为出发点，了解一些关于函数式和数据流式语言的特性。例如 VAL 这种语言，一方面它是典型的数据流式语言，另一方面它也具有某些函数式特性。此外，“如何消灭逻辑语句结构”的问题，我们会在下一节中予以详述。

```
<style>
/* 设置字体颜色 */
DIV {
 COLOR: rgb(127, 127, 0);
}
/* 设置背景图片的 url 地址 */
TABLE {
 BACKGROUND: url(http://127.0.0.1/bg.png);
}
</style>
```

这些过程是符合 CSS 规范的。但是 Internet Explorer 5.0 及更高版本的浏览器对此做出了一些扩展，它们使用一个名为 `expression` 的过程，来表明 CSS 属性需要通过一个计算过程来得到值。具体来说，如下例：

```
<style>
SPAN {
 BORDER: 1 solid red;
 POSITION: absolute;
}

#span_left {
 LEFT: 0px;
 WIDTH: 300px;
}

#span_right {
 LEFT: 300px;
 WIDTH: expression(document.body.clientWidth - 300);
}
</style>
```

我们可以用接下来的 HTML 代码来展示这个样式表的效果：

```
<body>
<span id="span_left">300px</span>
<span id="span_right" onresize="this.innerText = this.clientWidth + 'px'">
</span>
</body>
```

其效果如图 4-1 所示。

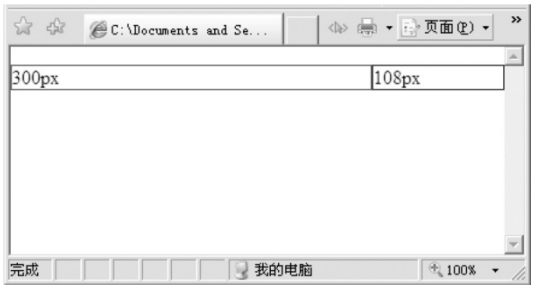


图 4-1　上述样式表在 Internet Explorer 浏览器中的效果

在上面的样式表中，我们用一个表达式运算来使 `span_right` 的宽度总是随当前网页的宽度而变化。我们的目的是要使 `left/right` 两个`<SPAN>`标签动态地填充网页上的左右两个部分。因此事实上我们也可以用如下方式写 `span_right` 的样式表：

```
/* 上例的一个更好的版本 */
#span_right {
 LEFT: expression(document.getElementById('span_left').currentStyle.width);
 WIDTH: expression(document.body.clientWidth -
   parseInt(this.currentStyle.left));
}
```

这些是 Internet Explorer 上的非常强大的功能。事实上，在 IE 6.0 中，有一个名为 IE 7 的开源项目就利用这种特性，为 IE 6.0 上的 CSS 样式表实现了与 IE 7 等同的功能。

那么，Internet Explorer 中是如何扩展这样的一个功能的呢？

如果我们更加完整地考察这个 `expresstion()`过程，就会发现它的一些独特之处，包括：

- 它是 JavaScript 语法的脚本代码；
- 它可以访问整个的文档对象模型（DOM，Document Object Model）；
- 它只能是一个表达式，或一组用逗号分隔的表达式；
- 它不能使用任何语句和语句分隔符（分号）；
- 它可以声明并使用函数（函数中也可以出现语句），但函数只用于值运算；
- 整个表达式的运算结果是值（用于赋给样式表属性）。

我们综合这些特性就可以发现，这个 `expresstion()`中所包括的，其实是一种：

- 消灭了语句的、
- 用表达式来运算求值的

JavaScript 语言的简化版本。

这是真正有趣的地方。事实上我们可以通过样式表中的 expresstion()过程来完成所有的工作，而无需单独写其他的脚本代码。换而言之，此处的 expresstion()已经具备了整个 JavaScript 语言的编程能力——不过在这里我们还要强调 expresstion()允许声明和使用函数这一特性。关于这一点，我们下一小节还会重新提及。

这一类的特性也出现在 J2EE 这种大型的语言系统中。在 JSTL 1.0 中，为了方便存取数据而自定义了这样一种语言（只能用在 JSTL 标签中），要求以#开始，将变量或表达式放在一对大括号之间，例如：

```
#{...}
```

由于它可以直接访问 faces-config.xml 中定义的名称、客户端 Request 中的参数，或者是 JavaBeans 中的成员等数据，因此可以写出这样的代码来：

```
<f:view>
  名称: <h:outputText value="#{userBean.name}, #{param.name}"/>
</f:view>
```

这个语言名为“JavaServer Faces(JSF) Expression Language(EL)”，JSEL 中还可以使用数值运算、逻辑运算、关系运算，以及数组和对象成员存取等表达式。和刚才我们提到的 CSS 中的 expresstion()过程一样，JSEL 提供的也是一个运算求值的结果；并且在表达式运算过程中，不会出现语句这种语法元素。

这一类的语言，被称为表达式语言（Expression Language，EL）。前面提到过，我们可以消灭语言中的陈述运算逻辑的三种语句（顺序、分支和循环），并使代码具有完全等同的编程能力。所以我们也看到表达式语言具有充备的程序设计能力，是一种极端精华的编程范型。

为了将这个范型与直译的“表达式（Expression）”区分开来，在随后的文字中我们将称之为“运算式语言（范型）”——事实上也存在这样的翻译，但这种翻译一般强调的是名词性质的表达式。而我们在这里使用这个名词，主要强调“通过运算求值来实现程序设计”的这样一个编程范型[33]。

4.3.3 如何消灭掉语句

读者应当注意到：由连续运算来组织代码与用顺序语句来组织代码，是两种不同的风格。对于“运算式语言”这种新的语言风格，如果要让它成为一种纯粹的且完备的语言（范型），那么我们需要让它仅通过“表达式运算”就能完成全部的程序逻辑，这包括其他语言中的三种基本逻辑结构：顺序、分支与循环——是的，我们在讲述命令式语言时提到过这三种基本逻辑结构，它既用于组织代码，亦用于陈述逻辑。同样，运算式语言也需要这两种能力（如果不考虑代码写得多么凌乱、难懂的话，我们可以忽略前者）。

因此，为了让“（纯粹地）连续运算”能实现足够复杂的系统，我们要在消减掉“语句”这个语法元素的同时，通过表达式来陈述三种基本逻辑。我们将看到：在运算式语言中，语句是可以被消灭掉的。

4.3.3.1 通过表达式消灭分支语句

单个分支的 IF 条件语句，可以被转换成布尔表达式。例如：

[33] Expression Language 通常被译作“表达式语言”，以这种方式称述对象时，主要说明它是一种叙述表达式规格、性质和功能的语言，一般不作为程序设计语言，因此也不会指称某种编程范型，例如正则表达式（RegExp）是一种表达式语言，但并不是程序设计语言。在本书中，“运算式语言（Expression Language）”是确指一种程序设计语言范型，它通过处理表达式求值来完成整个程序设计过程。

```
/**
 * 示例 1: 消灭条件分支语句(无 else 分支)
 */
if (tag > 1) {
  alert('true');
}

// 转换成
(tag > 1) && alert('true');
```

IF 条件分支语句（一个或两个分支），总是可以被转换（三元）条件表达式。例如下面的代码：

```
/**
 * 示例 2: 消灭条件分支语句
 */

// 1. 无 else 分支
if (tag > 1) {
  alert('true');
}

// 转换成
(tag > 1) ? alert('true') : null;

// 2. 有 else 分支
if (tag > 1) {
  alert('true');
}
else {
  alert('false');
}

// 转换成
(tag > 1) ? alert('true') : alert('false');
```

由于一个多重分支语句可以被转换成 IF 条件分支语句的连用，例如：

```
/**
 * 示例 3: 多重分支语句与 IF 语句连用的等效性
 */
switch (value) {
  100:
  200: alert('value is 200 or 100'); break;
  300: alert('value is 300'); break;
  default: alert('I don\'t know.');
}

// 等效于
if (value == 100 || value == 200) {
  alert('value is 200 or 100');
}
else if (value == 300) {
  alert('value is 300');
}
else {
  alert('I don\'t know.')
}
```

因此 SWITCH 语句与 IF 语句连用等效。而 IF 语句连用则可以用三元条件表达式连续运算来替代：

```
/**
 * 示例 4: 消灭 IF 语句连用
 * (参考示例 3)
 */

// ... (if 语句连用示例代码略)
// 转换为
(value = 100 || value == 200) ? alert('value is 200 or 100')
  : (value = 300) ? alert('value is 300')
  : alert('I don\'t know.');
```

4.3.3.2 通过函数递归消灭循环语句

放开易用性不论，常见的三种循环语句 while、do..while 和 for 是可以互换的，对这一点我想无需论述。因此，下面只以 do..while 语句为例，来讨论循环语句的问题。

循环语句可以通过函数递归来模拟，这一点其实也是经过证实的，如下例：

```
/**
 * 示例 1: 通过函数递归来模拟循环语句
 */
var loop = 100;
var i = loop;

do {
  // do something...
  i--;
}
while (i > 0);
```

```
// 用函数递归模拟上述循环语句
function foo(i) {
  // do something...
  if (--i > 0) foo(i);
}
foo(loop);

// 用函数递归模拟上述循环语句(更能展现函数式语言特性的)
void function(i) {
  // do something...
  (--i > 0) && arguments.callee(i);
}(loop);
```

但是，如果用函数来模拟循环，那么必然存在一个问题，就是栈溢出。循环语句的一个良好特性就是开销很小，而在函数的递归调用过程中，由于需要为每次函数调用保留私有数据和上下文环境，因此将消耗大量的栈空间。

但是递归中也可以存在不占用栈的情况，这就是尾递归。简单地讲，尾递归是指在一个函数的执行序列的最后一个表达式中出现的递归调用。由于这个递归是最后一个表达式，那么当前函数不需要为下一次调用保持栈和运算的上下文环境。换而言之，这种情况下，递归函数的多次调用中要么使用同一个栈（和上下文环境），要么根本就不使用栈（和上下文环境）。

一个简单的实现方法就是：尾递归相当于在函数尾部发生的一个（无需返回的）跳转指令。由于这种特性，所以满足尾递归的函数，就可以在不消耗栈和上下文环境的情况下，用来替代循环语句。关于该理论，在 SICP[34]中有过详细的解释，而在现实中，Scheme、Erlang 等语言都将尾递归作为一种重要的特性内置于编译器中。这些编译器内置尾递归（或强调必须使用严格的尾递归）的原因在于：在函数式等编程范型中，通过编译器的优化，可以无需“（循环）语句”这种编程元素来实现高性能的迭代运算。

然而不幸的是，目前已知的 JavaScript 的解释环境中并不支持这种特性。因此，我们在这里讨论函数式时，可以说“能够通过函数递归来消灭循环语句”，但在不支持尾递归（及其优化技术）的 JavaScript 中，这种实现将以大量栈和内存消耗为代价。

4.3.3.3 其他可以被消灭的语句

由于可以不使用循环和 `switch`，所以标签语句也就没有存在的价值——事实上函数式语言中另外有与“标签语句”类似的、函数式的实现。与此相同的，流程控制中的一般子句（`break` 和 `continue`）也没有存在的价值。

前面说过函数式语言可以不使用寄存器，因此这事实上只需要值声明，而不需要变量声明——值参与运算，变量其实是值的寄存。所以在函数式语言中，变量声明语句也是不需要的。

所以你会看到，在函数式语言中，除了值声明和函数中的返回子句之外，其他的语句都是可以被消灭的。但是，为什么这两种语句不能被消灭呢？我想，我不需要再给出答案了吧——如果你认真地读过本章节的话。

4.4 函数：对运算式语言的补充和组织

我们现在面临着一个新的词汇“运算式语言”（EL，Expression Language）。之所以需要强调它，是因为（目前）没有任何书籍将函数式语言的基础归结到这种运算范型中。

前面讲到，运算式语言的特点在于强调“求值运算”。我们也列举了 JavaScript 中的表达

[34] 《计算机程序的构造和解释》（Structure and Interpretation of Computer Programs），Abelson,H.等著，裘宗燕译。

式运算符，看到这些运算符处理的运算元都是“值类型数据”。由此我们可以认为：JavaScript 语言特性中的某个子集，可以作为一个最小化的运算式语言来使用——事实上我们也看到在 Internet Explorer 中将 JavaScript 的一个子集作为 CSS 的表达式（expression）来使用。

我们也讨论到，如果要在运算式语言中完成全部逻辑，那么它需要通过表达式或者运算来填补“被消灭掉的语句”。更确切地来说，它必须有能力实现“顺序、分支和循环”三种基本逻辑结构。在 JavaScript 中，可以通过“连续运算符（逗号）”和“三元条件表达式”来替代顺序和分支。因此，最后的问题是：如何在表达式级别实现循环？

我们也在前面给出了答案：使用函数递归来消灭循环语句。在这一个小节里，将进一步讨论这个话题。

4.4.1 函数是必要的补充

因为表达式运算是“求值运算”，所以有且仅有“当函数只存在值含义，且没有副作用”时，该函数才能作为表达式的运算元参与运算，并起到替代循环语句的作用。

显然，根据上面所述的“函数式语言中的函数”的特性，它确实可以充当这样的角色。因此，在一个纯粹的、完备的运算式语言中，函数是一个必要的补充。

但这只是理论上的、理想化的假设：如果“函数”是满足函数式语言的三个特性的话。而正如我们在前面所讨论的那样，JavaScript 中的函数事实上仍然存在副作用。即使除去这个我们寄期望于“程序员的习惯”来解决的问题不谈，也还有一个无法回避的问题：JavaScript 中的函数并不支持尾递归。

不过这仍不会对 JavaScript 中的运算式语言特性构成太多的负面影响。因为 JavaScript 是一种多范型语言，所以事实上它可以在一个函数内部使用循环语句——当然，这会使它在表面上看起来并没有彻底地消除循环语句。

为什么说是“表面上看起来”呢？因为函数在这里只有值的含义，所以函数只有返回的值在影响系统的运算。由此的推论是：无论函数内部如何实现，事实上并不会影响到其外部的系统（也包括编程范型的纯洁性）。

当然，对于 JavaScript 来说，使用“包含循环语句的函数”来实现纯粹的运算式语言编程时，仍然需要抑制函数的副作用。关于这一点，我们已经在前面讲述过了。

当在 JavaScript 中需要一种纯粹的“运算式语言”时，函数是一个必要的补充。这首先体现在对循环逻辑的封装上。在“尾递归”与“利用多范型特性来包含循环语句”这两种方案上，JavaScript（非常偷懒地）选择了后者。但只要不产生副作用，我们仍然可以承认这是一种纯粹的运算式范型。

作为更加具体的实例，我们可以在 Internet Explorer 样式表表达式中使用这样的技术。下例中，我们将通过循环查找顶层元素（`class=boxSpan`）来决定指定元素（`id= rightSpan`）的大小。这样复杂的逻辑，仍然可以通过在样式表中使用如下的表达式来实现：

```
<style>
SPAN {
 BORDER: 1 solid red;
 POSITION: absolute;
}
.nb {
 BORDER-WIDTH: 0px;
}
.boxSpan {
 LEFT: 0px;
 WIDTH: 90%;
 HEIGHT: 600px;
}

#rightSpan {
 BORDER: 1 solid blue;
 WIDTH: 100px;
 LEFT: expression(
  (function(el, className) {
     while (el = el.parentElement &&
            el.className != className) { /* null loop */ };
     return (el ? el.clientWidth : 0);
   }
  )(this, 'boxSpan') - parseInt(this.currentStyle.width)
 );
}
</style>
```

这个样式表的效果可以使用如下 HTML 代码来展示：

```
<body onload="document.getElementById('rightSpan').fireEvent('onmove')">
<span class="boxSpan" >
  <!-- 注：这里无论使用多少层次的 span，均不会影响到 rightSpan 的运算效果。 -->
  <span class="nb"><span class="nb"><span class="nb">
     <span id="rightSpan"
      onmove="this.innerText = this.currentStyle.left"></span>
  </span></span></span>
</span>
</body>
```

效果如图 4-2 所示。

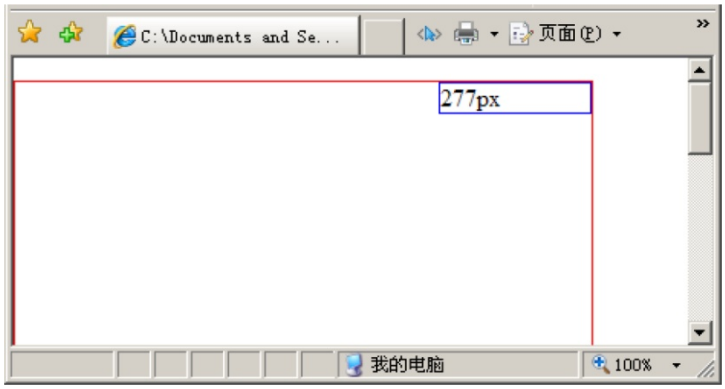


图 4-2　函数对 Internet Explorer 样式表表达式的补充及其效果

我们看到，蓝色块（rightSpan）总是动态的浮在外框（boxSpan）的右上角。为了增加难度，我们在这里设定外框是使用了 `class=boxSpan`，因此不能用 `getElementById` 这种便捷的方法来找到它。为此，我们在样式表的 `expression()`中声明了一个匿名函数并立即调用它，该匿名函数的返回值直接参与了后续的表达式运算。

正如我们所强调的那样，在匿名函数中我们使用了循环语句，但这并没有影响到外部表达式运算的纯洁性。因为对于这个表达式来说，该函数是如何实现的并没有关系，有且仅有它的值参与了运算的过程。

4.4.2　函数是代码的组织形式

我们当然可以使用连续的表达式运算来完成足够复杂的系统，这一点在前面已经论证过了。但是如果我们真的要这样去做，那么跟试图用一条无限长的穿孔纸带来完成复杂系统并没有区别——在代码（连续的表达式）达到某种长度之后，我们将难于阅读和调试，最终系统将因为复杂性（而不是可计算性）而崩溃。

在大型系统中，“良好的代码组织”也是降低复杂性的重要手段。对于运算式语言来说，实现良好的代码组织的有效途径之一，就是使用函数。如前所述，函数具有值特性、可运算、无副作用，因此在运算式范型中引入这样一个概念，并不会导致运算规则的任何变化。

所以，我们可以用函数来封装一组表达式，并更好地格式化它的代码风格。从语义上来讲，一个函数调用过程其实只相当于表达式运算中的一个求值：

```
// 表达式一
a = v1 + v2 * v3 - v4

// 示例 1：表达式一等效于如下带匿名函数的表达式
a = v1 + (function() {
  return v2 * v3 - v4;
})()

// 示例 2：等效于如下表达式
a = v1 + (
  v2 * v3 - v4;
)

// 示例 3：使用具名函数的例子
function calc() {
  return v2 * v3 - v4;
}
a = v1 + calc()
```

对于运算式的语法解释过程来说，示例 1 与示例 2 之间的区别仅在于：一个是用匿名函数调用来求值，另一个是通过强制运算符“括号()”来求值——当然，将前者改成具名的，在语

义上也没有变化（如示例 3）。

所以在运算式语言中，函数不但是消减循环语句的一个必要补充，也是一种消减代码复杂性的组织形式。

4.4.3 重新认识“函数”

我们看到，为了实现足够复杂的系统，运算式语言需要“函数”来组织代码和消减循环语句。在前面的行文中，我们花了很长的篇幅，以命令式语言中的函数（function）的概念，来解释了运算式语言的这种需求。当然，这种函数除了“名字（function）”跟命令式语言中用得一样之外，也具有三种特别的“函数式”特性。

但是我们事实上从来没有正式地解释过“函数式”是什么意思，只是反过来澄清过“并不是一个语言支持函数，这个语言就可以叫做函数式语言”。那么，如果要下个定义的话，我们是否能总结前文，说“函数式语言是一种用‘函数’来消减循环语句和组织代码的运算式语言”呢？更深层的问题是：运算式是不是函数式的基础，而函数式又是不是运算式的某个分支呢？

产生这些问题的症结其实在于“函数式语言”中的这个“函数”，并不是我们在命令式语言中看到的例程（函数 function 和过程 procedure），也不是我们在 JavaScript 中看到的 `function` 关键字或 `Function` 类型。所以，仅凭“JavaScript 中函数是第一型的”就推论出“JavaScript 是函数式语言”，这种推论是不严谨的，或者说根本就是不正确的。

在认识“函数式语言”之前，必须明确这个“函数”的含义。

4.4.3.1 “函数”==“lambda”

让我们先来看看“Vyacheslav Akhmechet”[35]对 lambda 的一个解释：

> “我在学习函数式编程的时候，很不喜欢术语 lambda，因为我没有真正理解它的意义。在这个环境里，lambda 是一个函数，那个希腊字母（λ）只是方便书写的数学记法。每当你听到 lambda 时，只要在脑中把它翻译成函数即可。”

简单地说，就是：函数==lambda。所以更复杂的概念，例如“lambda 演算（lambda calculus）”其实就是一套用于研究函数定义、函数应用和递归的系统。

从数学上，已经论证过 lambda 运算是一个图灵等价的运算系统；从历史上，我们已经知道函数式语言就是基于 lambda 运算而产生的运算范型。所以，在本质上来讲，函数式语言中的函数这个概念，其实应该是“lambda（函数）”，而不是在我们现在的通用语言（我指的是像 C、Pascal 这样的命令式语言）中讲到的 `function`。

4.4.3.2 当运算符等义于某个函数时

我们来看一段普通的 C 代码（以下设 `bTrue` 为布尔值 `true`）：

```
// 示例 1：普通的 C 代码
if (bTrue) {
  v = 3 + 4;
}
else {
  v = 3 * 4;
}
```

为了让代码简洁些，我们可以写成这样（所谓简洁是指忽略函数声明的部分）：

[35] 《函数式编程另类指南》的作者。

```
// 示例 2：使用函数的普通的 C 代码
function calc(b, x, y) {
  if (b) {
    return x + y;
  }
  else {y;
    return x *
  }
}
// 等效于示例 1 的运算
v = calc(bTrue, 3, 4);
```

我们说上面这两种写法都是命令式语言的。下面我们将 JavaScript 作为“运算式语言”，用表达式来重写一下：

```
// 示例 3：使用表达式的 JavaScript 代码
v = (bTrue ? 3+4 : 3*4);
```

接下来我们提出一个问题，既然在这个表达式中，值 3 与值 4 是重复出现的，那么可不可以像示例 2 一样处理成参数呢？当然，也是可以的：

```
// 示例 4：使用函数来消减掉一次传参数
function f_add(x, y) {
  return x + y;
}
function f_mul(x, y) {
  return x * y;
}
// 与示例 3 等义的代码
v = (bTrue ? f_add : f_mul)(3, 4);
```

我们注意示例 4 中一个问题：`f_add()`与 `f_mul()`其实本身并没有运算行为，而只是将“`+`”和“`*`”运算的结果直接返回。换而言之，事实上这里的“`+`”与“`*`”运算符就分别等义于 `f_add()`与 `f_mul()`这两个函数。

所以对于上面代码，除开赋值运算符之外的“求值表达式”部分，我们改写成如下（当然，下面的代码并不能被正常执行，但形式上与示例 4 是一致的）：

```
// 示例 5
(bTrue ? "+" : "*")(3, 4);
```

最后，我们改变一下代码书写习惯（改变书写代码的习惯其实对很多开发人员来说甚为艰难，但我们这里只是尝试一下而已）。新的代码风格是这样约定的：

- 表达式由运算符和运算元构成，用括号包含起来；
- 运算元之间的分隔符使用空格；
- 对于任何表达式来说，运算符必须写在前面，然后再写运算元。

注意我们这里没有改变任何逻辑，而只是换用了新的书写方法和顺序。那么新的代码应该写成这样：

```
((?: bTrue "+" "*") 3 4)
```

我们再接着约定：

- 对于三元表达式（`?:`）来说，`?:`号改用 `if` 来标识（至于三个运算元，按前面的规则，跟在运算符后面并用空格分隔即可）；
- 运算符可以用作运算元（这意味着“`+`”和“`*`”中的字符串引号可去掉）；
- 对于布尔值 `true` 来说，使用`#f` 标识。

这一过程中我们只是换用了新的符号标识系统。新的代码应该写成这样：

```
// 示例 6: 新的代码风格
(if #f + *) 3 4)
```

到这里，我最终给出答案：示例 6 其实是一行 Scheme 语言代码。而 Scheme 是 LISP 语言的一个变种，是一种完全的、纯粹的“函数式语言”。

从一个 JavaScript 表达式，到一行 Scheme 代码的过程中，我们只做出了一个假设：

如果运算符等义于某个函数。

如果我们再把这个过程逆转回来，那么同样也可以说：在 JavaScript 中可以用函数来替换运算符。这个例子可以写成这样：

```
// 示例 7: JavaScript 中用函数替代运算符
function f_mul(x, y) {
  return x * y;
}
function f_add(x, y) {
  return x + y;
}
function f_if(b, v1, v2) {
  return b ? v1 : v2;
}

//与示例 4 完全等义的代码
f_if(bTrue, f_mul, f_add)(3, 4);
```

我们再一次回顾前面说到的“函数式语言中的函数”的三个特性：

- 函数是运算元；
- 在函数内保存数据；
- 函数内的运算对函数外无副作用。

我们看到，示例 7 中的 `f_mul()`、`f_add()`和 `f_if()`完全满足了这三个特性。最后，我们得到结论：“当运算符等义于某个（lambda）函数”时，我们前面所讲述的运算式语言其实就是一种“非常纯粹的”函数式语言了。

4.4.4 JavaScript 语言中的函数式编程

在上面提到的这行 Scheme 代码中：

```
(if #f + *) 3 4)
```

“`if`”、“`+`”和“`*`”都是函数，而“`#f`”、“`3`”和“`4`”都是运算数（或者说值）。所以整个的 Scheme 语言的编程模式，就变得非常简单：

```
(function [arguments])
```

也就是说，整个编程的模式被简化了函数（function）与其参数（arguments）的运算，而在这个模式上的连续运算就构成了系统——整个系统不再需要第二种编程范型或冗余规则（例如赋值等）。

在上面的例子中，我们将 JavaScript 中的某些特性作为新范型——一种（基于表达式的）“运算式语言”来讨论，并证明它具有计算系统的完整的逻辑与运算能力。然后我们引入了“函数式语言中的函数”，说明“如果运算符等义于某个（lambda）函数”，那么所谓的运算式语言也就成了“函数式语言”了——我们一直在讲述的其实就是函数式语言，而所谓“运算式语言范型”，无非是我们偷梁换柱的一个名词罢了。

我们也知道，不能通过重新定义 JavaScript 规范来使得它表现出这种“函数式语言的纯粹性”。因此我们需要提炼并阐述一个 JavaScript 中的最小特性集，使它满足“函数式语言”的特性，以便在后文更好地讨论这些特性。

这样的一个特性集是：

- 在函数外消除语句，只使用表达式和函数，通过连续求值来组织代码；
- 在值概念上，函数可作为运算元参与表达式运算；
- 在逻辑概念上，函数等义于表达式运算符，其参数是运算元，返回运算结果；
- 函数严格强调无副作用。

这样的语言特性集的要点在于：关注运算，以及运算之间的关系。使用者必须认识到：连续运算的结果就是我们想要的系统目标。因而我们无可避免地要去面对一种“连续运算”的代码风格，我们的选择仅在于：把这种风格写得漂亮点，或放弃说“函数式语言不是我想要的”。

——当然，很明显我在这里写这本书并不是想要达到后一个目标。

4.5 JavaScript 中的函数

我们在前面已经约定，在 JavaScript 中使用函数式风格编程，应首先用表达式连续运算来组织代码，并且注意如下概念：在运算元中，除了 JavaScript 中默认的数据类型，重点强调函数可以作为运算元参与运算；在运算符中，强调函数可以等义于一个运算符。

我们已经在“2.3 JavaScript 的语法：表达式运算”中全面地讨论过表达式运算。在本小节中，我们将详细讨论 JavaScript 中的函数的种种特性。这些特性是 JavaScript 的函数能够成为“函数语言中的（lambda）函数”的根源——或表现为某些不足。当然，反过来说，正是为了让 JavaScript 成为一种函数式语言，设计者才为 function 这种类型添加了这些语言特性。

4.5.1 可变参数与值参数传递

在一些语言中，函数入口参数有多种调用约定，例如值参数、变量参数、传出参数，等等。常见的如表 4-2 所示。

表 4-2 较常见的函数调用约定

调用约定	传参顺序	清除参数责任	寄存器传参	实现目的	其他
Pascal	由左至右	例程自身	否	与旧有过程兼容	较少使用
Cdecl	由右至左	调用者	否	与 c/c++模块交互	一些语言默认使用该约定
Stdcall	由右至左	例程自身	否	Windows API	WinAPI 通常使用该约定

在 JavaScript 中，函数参数值只支持一种调用约定。它的特点表现为：

- 从左至右向函数传入参数；
- 传入参数的引用，函数内对它的任何修改都不会被传出；
- 传入参数个数（相对于函数声明时的形式参数）是可变的。

传入可变个数的一个简单(却并不常用的)技巧是用来替代函数内的局部变量声明。例如：

```
function Array.prototype.clear(len) {
  len = this.length;
  if (len > 0) {
    this.splice(0, len);
  }
}
```

对于该示例中的 clear()方法，很显然 len 是完全不需要的一个形式参数。既然它只用于在函数内暂存一下“this.length”的值，那么显然可以声明为一个局部变量。因此在这里，它被声明成函数形式参数 len 的唯一可能的理由只是：省掉 var 这个关键字[36]。

包括 JavaScript 自身的许多内部函数或方法，也都会经常用到“传入可变个数的参数”这种技巧。例如：

```
// 字符串 split 方法, separator 参数默认值为','. 当 limit 默认时返回整个数组.
stringObj.split([separator[, limit]])

// 数组的 concat 方法, item1..itemN 参数个数是可变的
arrayObj.concat([item1[, item2[, ... [, itemN]]]])
```

在具体到函数的代码实现时，对于 split()这种形式的函数声明，通常我们会将形式参数都声明出来。例如（str 参数是为了避免声明成 string 的对象方法）：

```
function s_split(str, separator, limit) {
  // ...
}
```

接下来，s_split()按如下规则来理解 separator：

- 如果值未传入，或者为 undefined，则返回包含单个 str 元素的数组；
- 如果值为空字符串，则将字符串分解成单个字符的字符串数组；
- 如果值为其他值，则转换为字符串，并使用该字符串分解 str。

对于第三项规则，举一个实例来说，如果传入 null，null 转换成字符串将是'null'，因此如下调用将返回数组['_', '_']：

```
s_split('_null_', null);
```

[36] 我并不赞用这种改变函数形式参数语义的“奇怪用法”，但这的确在某些代码包中出现，并且也是经常被置疑的用法。在这里讲到它，更多的是释疑，而非认同。

当实际函数调用时，如果默认指定参数（没有传入实际值），则它的值为 undefined。因此识别的代码如下：

```
// 示例 1: 对于可变参数的识别方法(1)
function s_split(str, separator, limit) {
  if (separator === undefined) {
     myArray = [str];
  }
  else {
    // ...
    // split to myArray
  }
  // 注: 当 limit 值大于结果数组长度时，将返回数组本身
  if (limit !== undefined &&
      limit < myArray.length) {
    myArray.length = limit;
  }
  return myArray;
}
```

现在出现了一个问题：在使用默认参数（不传入实际值）与使用 undefined 作为参数时，该参数的值都是 undefined！那么我们如何在代码中区分下面这两种情况呢？

```
// 参数传入 1
s_split(str, undefined);
// 参数传入 2
s_split(str);
```

理论上，对于 split()这个函数的设计，两种调用方法应该返回完全相同的结果。但是我们的问题只是（在某些可能的情况下）检测二者的不同，因而需要一个这样的方法。JavaScript 对此给出的方法是：使用 arguments.length 来检测实际传入参数的个数，如：

```
// 示例 2: 对于可变参数的识别方法(2)
function s_split(str, separator, limit) {
  // ...
  alert(arguments.length);
}
```

```
// 输出: 2
s_split(str, undefined);
// 输出: 1
s_split(str);
```

arguments 也用于在缺少形式参数的情况下进行处理，例如前面提到的 arrarObj.concat() 方法。形式上我们可以如下这样声明一个类似的函数：

```
function a_concat(arr) { // item1, item2 , ... [, itemN]
  // ...
}
```

我们没有办法对 item1...itemN 写出形式参数来，因此只能略去不写。这在具体的代码实现时，应该用如下的方法处理：

```
// 示例 3: 对缺少形式参数的识别方法
function a_concat(arr) {
  for (var i=1; i<arguments.length; i++) {
    if (arguments[i] instanceof Array) {
       //...
    }
    else {
     arr[arr.length] = arguments[i];
    }
  }
}
```

即使是在有形式参数的情况下，也仍然可以使用 arguments 来访问。例如上例中，第 0 个形式参数为 arr，但也可以使用 arguments[0]来访问它，而且二者完全等效——这种“完全等效”还包括对 arguments 的修改。例如：

```
// 示例 4: 访问 arguments 数组与形式参数完全等效
function myFunc(value) {
  arguments[0] = 100;
  alert(value);
}
// 输出: 100;
myFunc('abc');
```

在这个例子中我们注意到：不论是通过 arguments 还是形式参数，都可以修改入口参数的值。对于其他的一些语言来说，这个修改能否影响到原始值，取决于参数声明的形式：

```
// Pascal 风格 1: 可以在函数内修改参数
function Func_1(var str: String): string;
// Pascal 风格 2: 不能在函数内修改参数
function Func_2(const str: String): string;
```

JavaScript 不提供上述第一种风格的参数声明方式，以及其他类似于 in、out 等约束的参数声明方式。JavaScript 的函数参数声明，等效于上述的第二种风格。也就是说，你永远不可能在 JavaScript 中通过修改参数来影响到函数外部的变量——不过这里必须强调的是，你可以通过成员赋值或方法调用，来影响对象的成员。例如：

```
obj = {
  value: 100
}
function myCalc(obj) {
  //...
  obj.value = 2 * obj.value;
}
// 一般的代码风格
myCalc(obj);
alert(obj.value);
```

在这种情况下，“函数没有对外的副作用”只能是程序员的编程习惯上的约束。因此如果你试图让代码“更加函数式”，那么更合理的做法是：用方法来封装成员存取，并在某个函数之外去调用方法。因而对于上面的例子，相对良好的函数式风格应该如下：

```
obj = {
  value: 100,
```

```
  setValue: function(value) {
    return (this.value = value);
  },
  getValue: function() {
    return this.value;
  }
}
// 函数式风格的代码
function myCalc(v) {
  // ...
  return v*2;
}
// 输出: 200
alert(
  obj.setValue(myCalc(obj.getValue()))
);
```

当然，如果确定 `myCalc()`是 `obj` 对象的特定逻辑，那么上面的代码也可以是：

```
obj = {
  value: 100,
  calc: function() {
    return this.value * 2;
  }
}
// 输出: 200
alert(obj.calc());
```

4.5.2 非惰性求值

在下面这个例子中，代码的两个输出值将是什么呢？

```
/**
 * 示例 1: 在参数中使用表达式时的求值规则
 */
var i = 100;
// 输出 A
alert( i+=20, i*=2, 'value: '+i );
// 输出 B
alert( i );
```

在这个例子中，输出 A 时会显示数值“120”，输出 B 则会显示数值“240”。对于第一个输出，我们大致是可以理解的：因为 `alert()`只接受一个参数，这个参数的值为“`i+=20`”的运算结果，因此是 `120`。

尽管 `alert()`并没有接受第二、三个参数，但用于传入第二个参数值的表达式“`i*=2`”却完成了运算，并实际地向 `alert()`传入了该值——只不过 `alert()`没有使用它而已。由于该值已经完成了运算，所以在这行代码结束后，变量 `i` 已经被赋值为 `240` 了。于是，输出 B 将显示数值 `240`。

这里就体现了 JavaScript 的“非惰性求值”的特性——对于函数来说，如果一个参数是需要用到时，才会完成求值（或取值），那么它就是“惰性求值”的，反之则是“非惰性求值”。而 JavaScript 使用“非惰性求值”的很大一部分原因，在于它的表达式还支持赋值，这也就意味着表达式会产生对系统的副作用。类似于这行代码：

```
alert( i+=20, i*=2, 'value: '+i );
```

在语义上就是应该对系统产生副作用的。但如果参数是“惰性求值”的，那么只有当 `alert()`——或者别的什么函数——使用到第二个参数时，表达式才会完成运算，也才会产生副作用。然而，应该注意到的这种（被假设的）情况是的实质是：表达式在语义上“将会”产生副作用，而副作用是否发生却只能取决于参数“是否被使用”。

我们总不可能让静态的语义理解，与动态的执行效果发生这种冲突：程序员写出代码却不

知道它会何时产生副作用。因此，在“允许赋值表达式存在”的这种情况下，更加合理的做法的确是像 JavaScritp 那样采用“非惰性求值”。这样，函数被调用时的形式就决定了它将产生哪些副作用，以及将传入哪些值到函数中以参与运算——尽管这些值可能根本就不会被用到。

显然，你应该也已经看到了问题：由于值可能根本不被函数内部用到，因此它的运算也可能是完全无意义的：既不产生副作用，也不被使用。例如上面例子中的参数三，它完成了：

- 将数值 `i` 转换成字符串；并
- 与字符串`'value: '`合并成新字符串。

这样两个运算，然而运算结果却根本不被使用——这显然是一种 CPU 资源的浪费。

反过来讨论一下“惰性求值”。如果在上述的表达式中不存在副作用，例如我们采用如下的参数传入：

```
/**
 * 示例 2: 在参数的表达式中消除副作用
 */
var i = 100;
alert( i+20, i*2, 'value: '+i );
alert( i );
```

那么对于第一个 `alert()`调用来说，第二、三个参数“是否运算后再传入函数”就并不重要，即使它们完全不运算（你可以想象成没有这两个参数），也不会对第二个 `alert()`调用造成任何影响。

这意味着在“惰性求值”的语言中，如果参数不被用到，那么它无论是直接量还是某个表达式，都不会占用 CPU 资源，而执行器引擎也可以将它直接优化掉。作为一个更加明显的例子，下面的代码在“支持惰性求值的语言”中将什么也不会发生，而由于 JavaScript 是“非惰性求值”的语言，因此会进入死循环：

```
function lock() {
  while (true);
}

function NullFunction() {
}

alert( NullFunction(
  /* arguments..., */
  lock()
));
```

如果上面的代码中还传入更多的参数，那么这所有的参数都将被运算——无论它们有多大的运算量，并最后进入那个 `lock()`循环。然而，我们知道 `NullFunction()`其实什么也不做、什么行为也没有，我们调用了一个什么行为也没有的函数，系统却死锁了。

这是 JavaScipt 为它支持多语言范型，尤其是支持一种表达式与函数都“可能”对外产生副作用的语言范型（例如命令式语言）而付出的代价。尽管一般开发人员不会写出上例这样的“用来死锁”的代码，但是 JavaScript 也因此丢掉了函数式语言的一种很好的特性。

4.5.3 函数是第一型

在中文中解释“第一型（first-class data types）”会是一件比较艰难的事。这个词汇更完整的说法是“第一类数据类型”，其中的“第一类（first-class）”是有确定含义的修饰词。采用相同构词法的概念还有“第一类值（first-class values）”、“第一类函数/实体（first-class functions/entity）”、“第一类表示类型（first-class representation types）”、“第一类自然数据类型（first-class natural data types）”，等等。

在上述所有概念中，所谓“第一类（first-class）”所表明的意思，强调指称目标是“不可分解的、最高级别的、不被重述的”等。在一些解释中，直接套用社会学中的“一等公民（first-class citizens）”来阐释 first-class，虽然同样未能说清楚，但起码让人有了直观的概念[37]。

与一般程序设计语言中的数据类型概念比较，“基础类型（或元类型）”和“first-class data types”的概念是近同的。所谓基础类型，是指在语言中用来组织、声明其他复合类型的基本元素，它在语言 / 语法解释器级别存在，无需用户代码重述。换而言之，“第一型（first-class data types）”通常是指基础类型。更加直观地说，它表现为如下特性[38]：

- 能够表达为匿名的直接量（立即值）；
- 能被变量存储；
- 能被其他数据结构存储；
- 有独立而确定的名称（如语法关键字）；
- 可（与其他数据实体）比较的；
- 可作为例程参数传递；
- 可作为函数结果值返回；
- 在运行期可创建；
- 能够以序列化的形式表达；
- 可（以自然语言的形式）读的；
- 能在分布的或运行中的进程中传递与存储；
- ……

很显然 JavaScript 能够满足上述全部特性——但大多数宿主并非独立进程或分布式系统，因此最后一条特性无法体现。不过，因为 JavaScript 函数能够序列化成字符串并跨进程与主机传递，因此它天生具备这种在分布系统中进行传递与存储的能力。

要让 JavaScript 中的“函数（function）”能够替代运算符，并起到“Scheme 函数（Scheme 函数式语言中的函数）”的作用，其最重要的一条前提就是“让函数可以作为运算元”。也就是说，（如前面列举的特性，）既可以作为数据值存储与向函数传入传出，又可以作为函数来执行调用。这其实意味着它必然是“第一型的”。而“函数既可以是运算符，也可以是运算元（被运算的数据）”——亦即是函数可以作为函数的参数（运算符可以作为运算元）这一特性，在函数式语言中有一个专门的名词，叫“高阶函数”。

所以事实上“高阶函数”与“函数是第一型”两个特性是相依存的。JavaScript 中的第一型就是指六种基本类型：`undefined`、`string`、`boolean`、`number`、`object` 和 `function`。

4.5.4 函数是一个值

我们经常听到有人说“JavaScript 中所有的东西都是对象”。其实，从函数式语言的角度来看却正好相反，变成了“所有的东西都是值”。函数是第一型——可以作为值来使用、传递等，也正好充分证明了上述这个观点。

---

[37] 事实上，“first-class data types”最早确实引申自“一等公民（first-class citizens）”这个概念。它出自英国科学家 Christopher Strachey 在 1960 年的一篇论文《Functions as First-class Citizens》。Christopher 是指称语义的奠基者、分时系统（Compactable Time-Sharing System，CTSS）概念的创立者，据说他的名字与 C 语言中的“C”有着一些关系。

[38] 引自：http://en.wikipedia.org/wiki/First-class_object。

因为函数是值，所以函数可以被作为值来存储到变量，也可声明它的直接量。例如：

```
// 将一个函数直接量赋值给变量 func 存储
var func = function() {
  //...
}

// 声明一个(命名的)函数变量
function myFunc() {
  // ...
}
```

因为函数是值，所以函数可以直接参与表达式运算。例如：

```
// 直接参与布尔运算
if (!myFunc) {
  // ...
}

// 与字符串等其他类型数据混合运算
value = 'the function context: ' + myFunc;
```

因为函数是值，所以它也可以作为其他函数的参数传入，或者作为结果值传出——上一小节说这是“高阶函数”的特性，放在本小节中，作为“值的特性”来解释，也是一样的。例如：

```
// 函数作为参数传入
function test(foo) {
  // 函数参与表达式运算, 以及作为函数返回值传出
  return (foo !== myFunc ? foo  : function() {
    // ...
  });
}
// 将 test()函数调用的返回值作为新的函数调用
test(myFunc)();
```

4.5.5 可遍历的调用栈

调用栈是 JavaScript 的一个很有用的特性，许多高级的 JavaScript 功能都要用它来扩展。这些特殊的技巧将在本书更靠后的章节中去讲述，本小节主要讲述这个特性本身。

一个静态的、未调用的函数只是一个值。一旦它（例如函数 F）调用时，系统就将当前正在运行的函数（例如函数 A）入栈，并保留函数 A 的执行指针。在函数 F 执行完之后，函数 A 出栈，并继续执行指针后的代码。例如这样的代码：

```
function F() {
  // ...
}
function A() {
  F(x, y, z);
}
```

下面的形式代码说明上述代码执行过程中栈的构造，以及函数调用的初始化过程。先声明形式代码中用到的内部例程如下：

```
function BaseArray() {
  // 基本数组的构造器, 拥有 length 属性, 支持下标存取
}
function _stack_push(cp) {
  // 指定的代码指针(cp, Code Point)入栈
}
function _stack_pop() {
  // 指定的代码指针出栈, 返回一个 cp
}
function _set_cp(cp) {
  // 让引擎从指定的 cp 位置开始执行, 引擎会分析 cp 位置所在的
  // 闭包(closure)并装载该闭包环境
}
function _cp_function(foo) {
  // 将指定函数 foo 装入引擎, 并获取代码指针
}
```

以下为形式代码“F(x,y,z);”的形式代码：

```
// 以下为在函数 A()中调用时的形式代码
function _A() {
  _stack_push($$CP);

  // F(x,y,z)
  F.arguments = new BaseArray(x, y, z);
  F.arguments.callee = F;
  F.caller = _A;
  _set_cp(_cp_function(F));

  _set_cp(_stack_pop().next());
}
```

这个形式代码中最重要的是 callee 与 caller 的赋值，这就是 JavaScript 为用户可访问的调用栈准备的全部。

4.5.5.1 callee：我是谁

callee 在 JScript 5.5、JavaScript 1.2 以下版本中是没有实现的，即使是现在它也用得不多。但由于匿名函数和函数可被重写的特性的存在，使得 callee 变得不可或缺。下面两例说明在没有 callee 成员时的问题：

```
21  // 示例 1: 匿名函数调用自身
22  void function() {
23    // 在这里如何调用函数自身?
24  }();
25
26  // 示例 2: 函数如果在递归过程中被重写
27  var loop = 0;
28  function myFunc() {
29    otherFunc();
30    alert(loop++);
31    myFunc();
32  }
33
34  function otherFunc() {
35    myFunc = null;
36  }
37
38  myFunc();
```

在示例 2 中，另一种常见的（类似）情况可能是 myFunc 是一个时钟触发过程[39]。虽然 otherFunc()的代码写得有点不可理喻，但它表明的意思可能只是：“myFunc”这个全局变量名被第三方代码重写。很显然 loop 值只能被正常显示一次，在第一次迭代结束后，myFunc()值被重写了，于是当代码执行到第 11 行时，便出错了。看起来，解决的法子是不要使用全局变量名，但如果我们不给 myFunc()一个全局变量名，那么问题就回到了示例 1——我们如何调用匿名函数自身呢？

所以从 JavaScript 1.2 开始使 arguments 对象拥有一个成员：callee，该成员总是指向该参数对象（arguments）的创建者函数。由于 arguments 总可以在函数内部直接访问，因此也就总可以在函数内部识别“我是谁”。所以如果不考虑函数名重写或匿名的问题，下面的等式是成立的：

```
function myFunc() {
  // 下列等式成立
  arguments.callee === myFunc.arguments.callee;
  arguments.callee === myFunc;
}
```

这样一来，无论是否匿名，函数的递归调用就总可以写成：

[39] 当我们把这个问题放在时钟触发的代码中时，整个问题会变得更为随机，往往这种错误是不可重现的。

```
void function() {
  // ...(略)
  arguments.callee();
}();
```

4.5.5.2 caller：谁呼（叫）我

在函数内部识别自身（我是谁），只是解决了匿名递归的问题，而并非我们说到的“遍历调用栈”的问题。如果我们要遍历函数 A 调用函数 B（以及调用函数 A 自身）时所形成的调用栈，那么我们就需要用到 `caller` 这个成员。从前面的形式代码中我们可以看到，这个成员是 `Function` 的一个属性。

所以我们先讲述 `callee` 属性的原因仅在于：如果我们要遍历栈，则必须先找到指定的（包括匿名函数在内的）函数，然后才能使用 `caller` 来访问它[40]。下面先给出遍历调用栈的公共代码：

```
// 用于获取函数名的正则表达式
var _r_function = /^function\b\s*([$\w\u00FF-\uFFFF]+)\s*\(/m;

// callback 函数, 用于显示函数的信息
function showIt(func) {
  var name = (!(func instanceof Function) ? 'unknow type' :
    _r_function.test(func.toString()) ?  RegExp.$1 :
    'anonymous');
  alert('-> ' + name);
}

// 遍历调用栈
function enumStack(callback) {
  var f = arguments.callee;
  while (f.caller) {
    callback( f = f.caller );
  }
}
```

下面的代码则演示如何使用这些函数来遍历调用栈：

```
 function level_n() {
  enumStack(showIt);
}

function level_2() {
  // ...
  level_n();
}

function test() {
  level_2();
}

test();
```

运行该示例，我们将看到如下的输出信息：

```
-> level_n
-> level_2
-> test
```

亦即是 `enumStack()`函数被调用时的栈信息——排除 `enumStack()`函数自身。

接下来我们提一个问题：既然 `caller` 是 `Function` 的一个成员。那么在栈上如果出现两次、多次同一个函数，该函数的 `caller` 又如何处理呢？——更加严重的是，如果这个函数是来自几次不同的调用，又怎么办呢？

答案是：JavaScript 中无法识别这种情况，并导致遍历出错。如下例：

```
var i = 0;
```

[40] JScript 中，也可以直接用 `arguments.caller` 来访问。但这是非标准的用法。

```
var i_max = 1;  // <-- 当这里置值大于 1 时，将导致 enumStack()进入死循环
function f1() {
  f2();
}
function f2() {
  if (++i < i_max) {
    f1();
  }
  enumStack(showIt);
}

f1();
```

我们把这个例子构造得稍稍复杂了一点，其逻辑基本上就是：`f1()`调用 `f2()`，然后 `f2()`再调用 `f1()`。我们使用变量 `i` 来决定调用的次数，因此这个循环调用也是可以中断的。当我们设置 `i_max` 为 1 时，则在 `f2()`中不会重新调用 `f1()`，因此它输出的栈信息是：

```
-> f2
-> f1
```

当我们设置 `i_max` 大于 `1`（例如置值 `2`）时，则正确的栈应该是：

```
-> f2  // caller 指向 f1
-> f1  // caller 指向 f2
-> f2  // caller 指向 f1
-> f1  // caller 指向 host
```

而当列举到第二次时，`f1.caller`、`f2.caller` 就开始相互指向对方，从而导致死循环：

```
-> f2  // caller 指向 f1
-> f1  // caller 指向 f2
-> f2  // caller 指向 f1  // <-- 由于指向的 f1 仍是上一次迭代中的函数实例，导致下述结果
-> f1  // caller 指向 f2  // <---这里的 host，在第二行的位置被覆盖了
-> (死循环)...
```

因此，看起来非常漂亮的列举栈的功能，在这里的例子中失效了。当然，对于 JavaScript 自身来讲，这并不是问题。因为在引擎内部会有一个真实的、不依赖函数引用

或函数名的调用栈。缘于引擎内部和真实脚本环境并不一致，所以我们可以在调试器中看到上例的正常调用栈，而在脚本代码中，却没有任何办法来列举它[41]。

也许是因为这样的原因，尽管在 JScript、SpiderMonkey JavaScript 等引擎纷纷实现了该方法，但在 Safari 项目的 WebKit 引擎却没有在 `Function` 对象中实现 `caller` 属性[42]。从 WebKit 项目组得到的反馈是：ECMAScript 的标准中反对使用该属性——很遗憾，这也是事实。

4.6 闭包

让 JavaScript 的“纯函数式风格代码”看起来好看一些的基本决窍在于：用表达式运算来实现局部逻辑并完成最终的运算与输出；用函数声明来组织整体的代码。一个简单、直观的代码块示例如下：

```
function f1(){
  // expressions
}

function f2() {
  // expressions
}

// main()
alert(
  f1() +  f2()
);
```

这当然意味着“函数”将会大量出现。而我们需要保证“函数式风格”的纯粹性，以免回归到命令式语言的老路上去，就需要强调“在内部保存数据和对外无副作用”这两大特性——这在 JavaScript 中都是通过“函数闭包（Function Closure）”来实现的。

除了函数闭包之外，在 JavaScript 中还存在一种特殊的“对象闭包（Object Closure）”。它是与 `with` 语句实现直接相关的一种闭包。因为这种特殊性，我们将在“4.6.9.2　对象闭包带来的可见性效果”小节单独地讨论它。而在其他章节中的所谓“闭包”，都是特指函数闭包。

[41] 关于这个问题，我们在后面关于 JScript 5 的兼容实现中，在讨论到 `call/apply` 的实现及其限制时，会再次提到这个问题。我们的处理方法，是在 `_safe_caller()` 函数中，加入列举层数的限制。当然，这仍不能避免栈上出现同一函数时，`_safe_caller()` 失效的问题。

[42] Safari V3 中的 JavaScriptCore 引擎已经添加了该属性，但是 Opera 的 JavaScript 引擎中至今仍然没有这样的实现。

4.6.1 什么是闭包

闭包（Closure）与函数有着紧密的关系，以至于许多人将函数与闭包等同起来讨论，但结果却总是讨论不清楚。事实上在 JavaScript 中，一个函数只是一段静态的代码、脚本文本，因此它是一个代码书写时，以及编译期的、静态的概念；而闭包则是函数的代码在运行过程中的一个动态环境，是一个运行期的、动态的概念。由于引擎对每个函数建立其独立的上下文环境，因此当函数被再次执行或者通过某种方法进入函数体内时，就可以得到闭包内的全部信息。

闭包具有两个特点：第一是闭包作为与函数成对的数据，在函数执行过程中处于激活（即可访问）状态；第二是闭包在函数运行结束后，保持运行过程的最终数据状态。因此函数的闭包总的来说决定了两件事：闭包所对应的函数代码如何访问数据，以及闭包内的数据何时销毁。对于前者来说，涉及作用域（可见性）的问题；对于后者来说，涉及数据引用的识别。

闭包包括的是函数运行实例的引用、环境（environment，用来查找全局变量的表），以及一个由所有 upvalue 引用组成的数组。图 4-3 说明了这种结构关系。

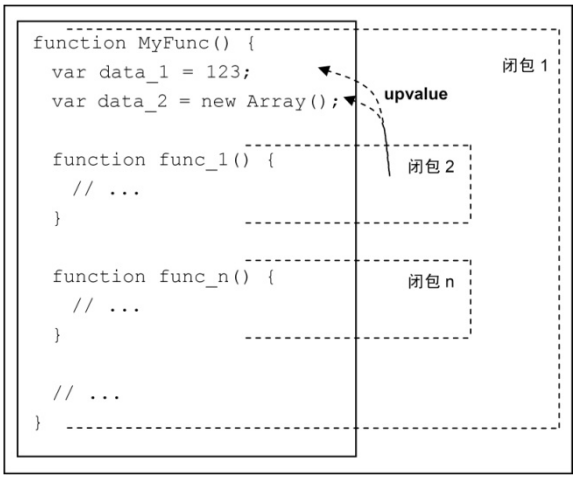


图 4-3　闭包及其相关概念之间的关系

图 4-3 并不能很好地传达出“闭包是运行期概念”这样的信息，仅能从静态的视觉效果上说明闭包、子函数闭包、upvalue 之间的关系。一些在图中未能标示的信息包括：

- 在运行过程中，子函数闭包（闭包 2～n）可以访问 upvalue；
- 同一个函数中的所有子函数（闭包 2～n），访问一份相同值的 upvalue。

下例简单说明了第二条特性：

```
funtion MyFunc(){
  var data = 100;
  function func_1() {
    data = data * 5;
  }
  function func_n() {
    alert(data);
  }

  func_1();
  func_n();
}

// 由于 func_n 与 func_1 使用相同的 upvalue 变量 data. 因此在 func_n()中可以
// 显示 func_1()对该值的修改。返回结果值: 500
MyFunc();
```

4.6.2 什么是函数实例与函数引用

在书写代码的过程中，函数只是一段代码文本。对编译语言来说，这段文本总被编译成确定的代码，并放在确定的内存位置执行。因此在编译语言里，一段代码文本与运行期的代码实例实际上是等同的、一对一的概念。又由于函数可以被多个不同的变量引用，所以一个函数的代码块在运行期可对应于多个变量（函数入口地址指针），如图 4-4 所示：

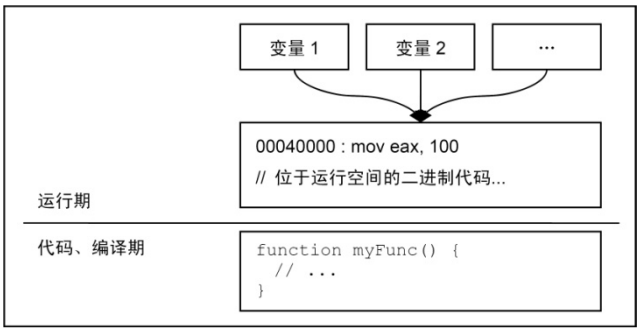


图 4-4　编译语言中函数代码与运行期（函数实例和引用）之间的关系

在 JavaScript 中也可以具有这种关系。下例说明这种情况（一段函数有多个引用）：

```
function myFunc() {
  // ...
}

var f1 = myFunc;
var f2 = myFunc;

// 返回值 true，表明变量 f1 与 f2 指向同一个函数实例．反过来说，也就是 f1 与 f2 是
// 该函数实例的多个引用．
alert( f1===f2 );
```

但在 JavaScript 中，却并不是单单只有这种关系。更为复杂的情况是：一个函数代码可以有多份函数实例，一个函数实例可以有多个函数引用。因此变成了图 4-5 的样子。

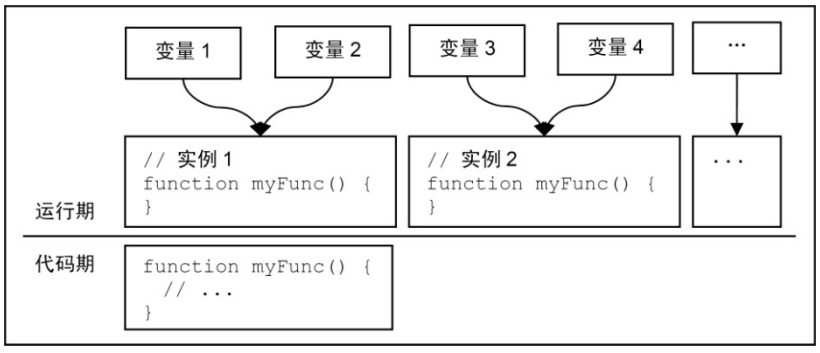


图 4-5　函数代码块的实例与引用之间可以存在多对多的关系

下面这个例子，说明在 JavaScript 中同一个函数代码块可以有多个函数实例：

```
function MyObject() {
  function func() {
    // ...
  }

  this.doFunc = func;
}

var obj1 = new MyObject();
var obj2 = new MyObject();

// 显示 false，表明这是两个不同的函数实例
alert( obj1.doFunc === obj2.doFunc )
// 显示 true，表明两个函数实例的代码块完成相同
```

```
alert( obj1.doFunc.toString() == obj2.doFunc.toString() )
```

也许有读者认为上例是 new 运算的效果，但事实上这与 new 运算无关，也与 JavaScript 的对象系统无关。下例说明这一点：

```
function myFunc() {
  function func() {
    // ...
  }

  this.doFunc = func;
}
var obj = new Object();

// 1. 进入 myFunc，取 func()的一个实例
myFunc.call(obj);
// 2. 套取函数实例的一个引用，暂存
var func = obj.doFunc;
// 3. 再次进入 myFunc，取 func()的一个实例
myFunc.call(obj);
// 4. 比较两次取得的函数实例，结果显示 false，表明是不同的实例
alert( func === obj.doFunc );
// 5. 显示 true，表明两个函数实例的代码块完成相同
alert( func.toString() == obj.doFunc.toString() )
```

4.6.3 （在被调用时，）每个函数实例至少拥有一个闭包

许多书籍总不能把函数与闭包讲清楚，其中的原因之一，就是不能将函数（代码块）、函数实例与函数引用分开来。关键在于：闭包是对应于运行期的函数实例的，而不是对应函数（代码块）的。由于闭包对应于函数实例，那么我们只需要分析哪些情况下产生实例，就可以清楚地知道运行的闭包环境。

前面已经列举过，下面的情况只产生引用不产生实例：

```
// 示例 1: 没有函数实例产生
function myFunc() {
}
var f1 = myFunc;
var f2 = myFunc;

// 显示 true
alert( f1 === f2 );
```

对象的实例只持有原型中的方法的一个引用，因此也不产生（方法）函数的实例：

```
// 示例 2: 没有函数实例产生
function MyObject() {
}
MyObject.prototype.method = function() { /* ... */ };
var obj1 = new MyObject();
var obj2 = new MyObject();

// 显示 true
alert( obj1.method === obj2.method );
```

下面的例子也是常用的构造对象的方法，但它会产生多个函数实例（与示例 2 的原型构造方法正好不同）：

```
// 示例 3: 有函数实例产生
function MyObject() {
  this.method = function() { /* ... */ };
}

var obj1 = new MyObject();
var obj2 = new MyObject();
// 显示 false;
aerlt( obj1.method === obj2.method );
```

可见，常见的两种构造对象的方法，产生对象实例的效果并不一样（我们这里强调的是方法的闭包性质）。下面的示例，就交叉利用了示例 2 与示例 3 两种不同特性，来实现构造器及其原型，从而在 JavaScript 中构造出复杂的对象：

```
// 构造器函数
function MyObject() {
  var instance_data = 100;
```

```
  this.getInstanceData = function() {
    return instance_data;
  }

  this.setInstanceData = function(v) {
    instance_data = v;
  }
}
// 使用一个匿名函数去修改构造器的原型 MyObject.prototype, 以
// 访问该匿名函数中的 upvalue.
void function() {
 var class_data = 5;

  this.getClassData = function() {
    return class_data;
  }

  this.setClassData = function(v) {
    class_data = v;
  }
}.call(MyObject.prototype);

// 创建对象
var obj1 = new MyObject();
var obj2 = new MyObject();

// 输出 100
// 表明 obj2 与 obj1 的 getInstanceData 是不同函数实例，因此在访问不同闭包的 upvalue
obj1.setInstanceData(10);
alert(obj2.getInstanceData());

// 输出 20
// 表明 obj 与 obj1 的 getClassData 是同一个函数实例，因此在访问相同的 upvalue
obj1.setClassData(20);
alert(obj2.getClassData());
```

除了构造对象实例的情况，我们也常常在函数中将内部函数作为返回值。下面的示例，随函数的执行次数产生多个 `MyFunc()`函数实例：

```
// 示例 4: 有函数实例产生
function aFunc() {
  function MyFunc() {
    // ...
  }
  return MyFunc;
}
var f1 = aFunc();
var f2 = aFunc();

// 显示 false
alert( f1 === f2 )
```

下面两个例子与示例 4 中的 `aFunc()`一样，都将在多次调用中，产生多个函数实例：

```
// 示例 5: 有函数实例产生
function aFunc_1() {
  var MyFunc = function () {
    // ...
  }
  return MyFunc;
}

// 示例 6: 有函数实例产生
function aFunc_2() {
  return function() {
    // ...
  };
}
```

我们也有机会返回同一个函数实例，只是比较麻烦一点：

```
// 示例 7: 有函数实例产生
var aFunc_3 = function () {
  var MyFunc = function () {
    // ...
  }

  // 返回一个函数到 aFunc_3
  return function() {
    return MyFunc;
  }
}()

// 多次调用 aFunc_3()将得到 MyFunc 的同一个实例
var f3 = aFunc_3();
var f4 = aFunc_3();

// 显示 true
alert( f3 === f4 )
```

在这个例子中，我们实际上在 aFunc_3 内部创建了一个匿名函数，这个匿名函数（即使是多个实例）访问 upvalue 时，将会得到相同的数据。因此多次调用 aFunc_3 时，就会得到同一个 MyFunc()实例——匿名函数的 upvalue 中的那个实例。

上面的各种形式在各个 JavaScript 的代码包中都经常出现。但本质上就是“产生”与“不产生”函数实例两种形式，进而形成两种闭包、两种保护数据和提供运行上下文环境的形式。

4.6.4 函数闭包与调用对象

《JavaScript 权威指南》（第四版）对“调用对象”作出了三点说明：

- 其 4.6 小节中说，对象属性与变量没有本质差异；
- 其 4.6.1 小节中说，全局变量其实是“全局对象（global object）”的属性；
- 其 4.6.2 小节中说，局部变量其实是“调用对象（call object）”的属性。

为了解释上述话题，《JavaScript 权威指南》引入了“JavaScript 执行环境（执行上下文，Execution Context）的概念[43]。这其中所说到的“全局对象（Global Object）”与“调用对象（Call Object）”其实都是指图 4-6 中的 ScriptObject 结构。

[43] 在我早期的一些公开文档中，会把执行环境与对象上下文（Object Context）混同起来，根本的原因便在于我把“全局对象／局部对象（global object／call object）”与 JavaScript 中的 Object()对象实例混作一谈。事实上，在 JavaScript 环境中，对象实例是没有、也并不需要“上下文环境”的。因为对象是存储系统，不是执行系统，其主要运算是成员存取操作“.”和“[ ]”——我们甚至讨论过“在 JavaScript 中并没有真正的方法调用运算”（参见“2.5　面向对象编程的语法概要”）。这种对象实质上只是一个“名字－值”对照表（其自身也可作为引用与值参的运算）。但这些与“函数”完全无关，与运行期的“上下文环境”也完全无关。

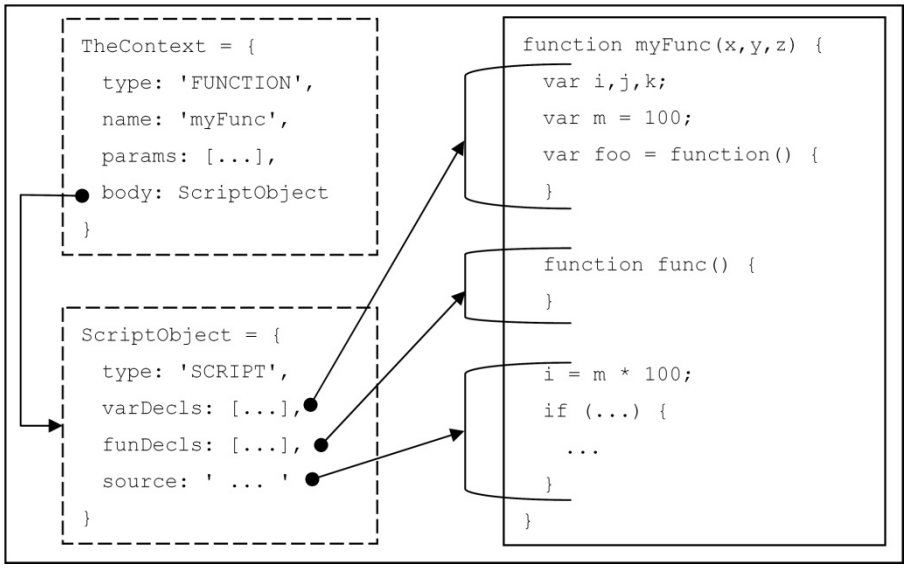


图 4-6　闭包相关元素（调用对象、上下文环境等）的内部数据结构

图 4-6 中，TheContext 结构只描述函数作为对象时的外在表现，例如名称是 `myFunc`，参数有哪几个，等等。它的 body 指向一个 ScriptObject，这个 ScriptObject 就包含了函数代码体中全部的语法分析结构，包括：内部变量表（`varDecls`）和内嵌函数表（`funDecls`），以及除此之外的全部代码（source）。

所谓“调用对象”，是指当前正在调用函数的 ScriptObject 结构；而所谓“全局对象”，是指系统全局环境下的一个 ScriptObject 结构。下面讲述有关于该结构的基本规则。

4.6.4.1 “调用对象”的局部变量维护规则

规则一：在函数开始执行时，varDecls 中所有值将被置为 undefined。因此我们无论如何访问函数，变量初始值总是 undefined。例如：

```
// 变量初始化
function myFunc() {
  alert(i);
  var i = 100;
}

// 输出值总是 undefined
myFunc();
myFunc();
```

由于 varDecls 总在语法分析阶段就已经创建好了，因此在 myFunc()内部，即使是在“var i”这个声明之前访问该变量，也不会有语法错误，而是访问到位于 ScriptObject 结构内 varDecls 中该变量的初值：undefined。由于 varDecls 总在执行前被初始化，因此第二次调用 myFunc()时，值仍是 undefined。

规则二：函数执行结束并退出时，varDecls 不被重置。正因为 varDecls 不被重置，所以 JavaScript 中的函数能够提供“在函数内保存数据”这种函数式语言特性。需要说明的是，该规则与规则一并不冲突。上例中第二次调用 myFunc()时，没有显示“在函数内所保存的数据（该例中是数值 100）”的原因是：第二次执行函数时，在进入函数前 varDecls 被再次初始化了。

如果我们不再次执行该函数，而通过一些方法来考察该函数的内部，就可以看到该数据被保存了——这就是“4.2.2　在函数内保存数据”所讲述的内容和示例效果。

规则三：函数内数据持续（即“在函数内保存数据”）的生存周期，取决于该函数实例是否存在活动引用——如果没有，则“调用对象（ScriptObject）”被销毁（即内存回收）。

4.6.4.2 “全局对象”的变量维护规则

我们上面例举的是一个函数内部的局部变量、嵌套函数和代码。ScriptObject 结构对于“全局对象”来说，仍然是完全适用的。唯一不同的是，从代码的语义分析上，我们找不到一个全局的“TheContext”结构。

不过在 JavaScript 引擎内部，这个全局的 TheContext 结构仍然存在。这种情况下，我们可以把全局代码看作是某个函数中的“SCRIPT”代码块。基本结构如图 4-7 所示。

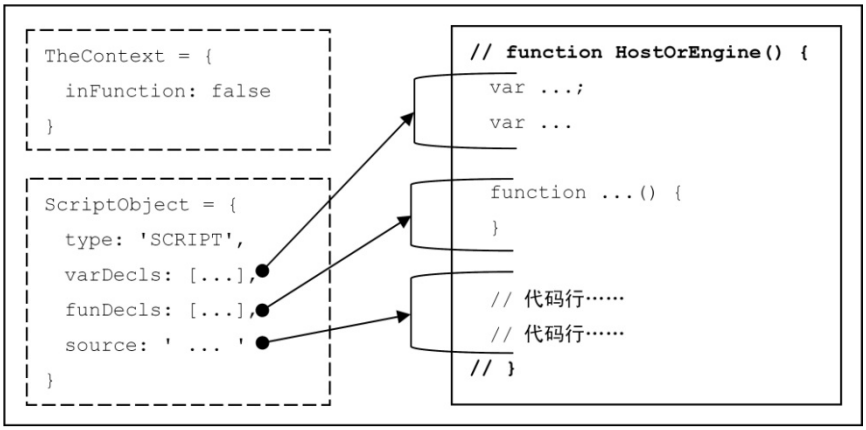


图 4-7　全局闭包相关元素（全局对象、上下文环境等）的内部数据结构

全局的变量维护规则与关于“调用对象”的变量维护规则并不冲突。但由于（虚构的）`HostOrEngine()`存在着特殊性，因此：

- 由于该函数从来不被再次进入，因此不会被重新初始化；
- 由于该函仅有一个被系统持有的实例，因此它自身和内部数据总不被销毁。

需要强调的是，由于这些特殊性，下面关于生存周期的讨论中将不包括“全局对象”。

4.6.4.3　函数闭包与“调用对象”的生存周期

读者应该注意到，上述的“调用对象”其实在语法分析期就可以得到。有了这个在语法分析期得到的 `ScriptObject` 作为原型，接下来事情就好办了。因为在运行期该函数实例有一个函数闭包，所以在执行它时，引擎将会：

- 创建一个函数实例；然后，
- 为该函数实例创建一个闭包；然后，
- 为该函数实例（及其闭包）的运行环境从 `ScriptObject` 复制一个“调用对象”。

因此这三个运行期结构总是一体出现的，一般情况下看起来就像处于同一个生存周期之中。虽然在后续的章节中你会看到其他特殊的情况，但在这里，读者先理解为同一生存周期亦无不可——我们先讨论这种闭包与“调用对象”的生存周期，依赖于具体函数实例被引用、释放引用和销毁的周期的情况。图 4-8 结合示例代码做进一步解释：

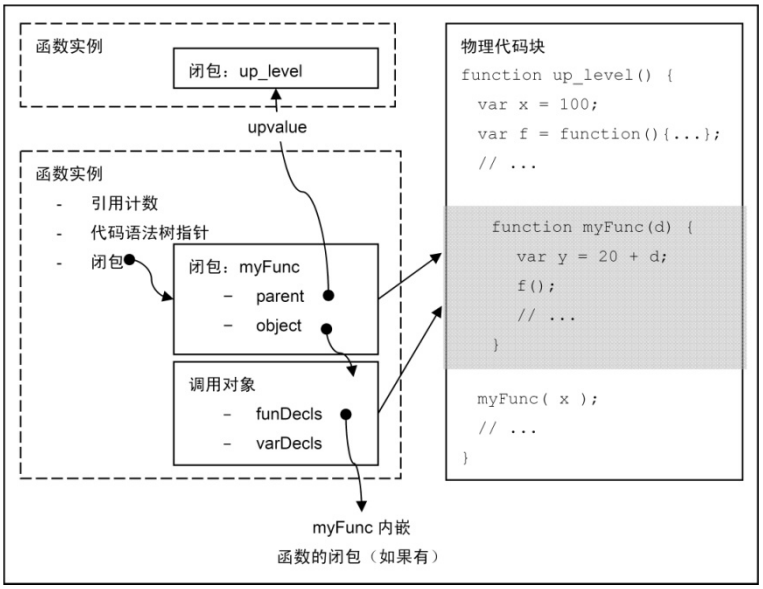


图 4-8　示例代码块在运行期的数据结构分析

在这个例子中，myFunc()访问了两个存在依赖的变量。其一是参数 d，通过函数入口传入。这种情况下，它可能在函数内被持有，或仅作求值运算。但在这个例子中：

```
// 该语句仅将参数 d 用作值运算，在 myFunc()闭包退出后 d 的引用状态将被复位
var y = 20 + d;
```

这说明在下面的语句中传入的变量 x，在 myFunc()中其实并没有被持有：

```
myFunc( x );
```

但第二个变量 f 就与此不同。变量 f 是在闭包中通过 upvalue 访问到的 up_level()中的变量。接下来的问题是：由于 f 是闭包外的引用，因此当函数 myFunc()还存活时，函数 up_level()也必须存活。而且由于 JavaScript 是动态语言，因此在 up_level()的代码体中可能随时会修改变量 f；又由于 myFunc()可能被外部其他变量引用，因此在 up_level()执行结束后，变量 f 也不应该被清除（这正是函数式语言中函数闭包的特性）。

因此一个闭包 A 使用了 upvalue，那么 upvalue 所在的闭包 B，就为 A 所依赖。

除了在闭包内通过标识符显式地引用 upvalue，从而导致闭包与闭包间的生存周期关联之外，还有一种导致关联的情况。这正好也与前面提到的一个例子有关系。我们在讨论参数 d 时，说到过“（参数）可能在函数内被持有，或者仅作求值运算”。但是我们上面的例子只讲述了“作求值运算”，而未讨论导致引用的情况。而这种“导致引用的情况”，也导致闭包间产生关联关系。例如：

```
function up_level() {
  var f = function(){...};
  function myFunc(foo) {
     var _foo = foo;
     // ...
     function aFunc() {
       _foo();
     }
  }

  myFunc(f);
}
```

在这个例子中，函数 f 被 myFunc()中的变量_foo 持有了一个引用。尽管“对函数外无副作用”，且函数 f 是通过入口参数传递的，但仍然不可避免地进行了引用计数。在这里也应该注意到一个细节，前例中使用的代码是：

```
var x = 100;
// ...
function myFunc(d) {
  var y = 20 + d;   // d 是入口参数，传入值类型变量 x
  // ...
```

而在本例中使用的代码是：

```
var f = function() {...};
// ...
function myFunc(foo) {
  var _foo = foo;   // foo 是入口参数，传入引用类型变量 f
  // ...
```

我们在两个例子中，传入的数据类型和在函数内使用的方法都不一致。真正导致闭包持有外部变量的原因是：当 foo 是一个引用，且被闭包内的某个变量赋值、传递或收集（例如数组的 push()操作）。

由于是动态语言的缘故，一个变量 / 入口参数是否为引用类型只能在运行期动态地获得，而无法通过代码的上下文语义来分析。所以我们不能找到更好的、差异更明显的例子来对比上述两种情况。而读者应知道，这个“引用与释放引用”的运算是运行期的、难以静态推演的行为。这便足够了。

4.6.4.4 引用与泄漏

在前面，我们讨论了闭包间基本的引用规则。我们也讨论到闭包与调用对象的生存周期，基本上可以归结为“函数实例”的生存周期。

在编译语言，以及以编译成二进制代码为主的静态语言中，函数体总在文件的代码段中，并在运行期被装入标志为“可执行”的内存区。因此，事实上我们不会认为代码、函数等会有“生存周期”。我们在大多数时候会认为“引用类型的数据结构”——例如指针、对象等——具有引用、生存周期和泄漏的问题。

我们也同样说过，在 JavaScript 中的函数也是一个引用类型的“数据 / 值”——尽管我们没有从这个角度上去论证它必然存在引用、生存周期与泄漏的问题，但事实上的确可以这样做。

除了这个原因之外，我们也应该注意到，由于全局函数事实上也可以视为宿主对象的属性（例如浏览器中的 window 对象），因此很多情况下，JavaScript 中的函数都具有“作为方法时依赖于对象生存周期”和“作为闭包时依赖函数实例生存周期”这样双重的复杂性。

当然，也有独立于对象系统而存在的“纯粹的函数”，例如匿名函数、内嵌函数等。很多时候它们只用于运算，而并不赋值到某个对象成员（从而变成对象方法）。但如果一个函数不被显式 / 隐式地赋值，那么它的函数实例在使用之后就必然被废弃——因为没有被引用。换而言之，引用正是“赋值”这样的运算带来的。

也许有人会置疑：函数返回值是否有引用效果呢？例如：

```
function myFunc() {
   function foo() {
     //...
   }
   return foo;
}
```

这个例子其实可以用于证明函数返回（return 子句）并不导致引用。因为如果我们这样来使用 myFunc()函数：

```
myFunc()();
```

那么函数 foo()在返回后立即被执行了，然后就会释放。因此 return 子句并不是导致引用的必然条件。相反，如果我们下面这样使用：

```
func = myFunc();     // <-- 在这里产生一个引用
func();
```

那么将产生引用，而后 myFunc()与 foo()的生存周期就将依赖于变量 func 的生存周期了。

除了上述这种被变量引用的情况，在 JavaScript 中最常见的其实是对象属性的引用。这主要指如下四种情况下：

- 对象在构造时，使用 this 引用进入构造器函数；或
- 对象在调用方法时，使用 this 引用进入函数；或
- 某个函数使用 call/apply 方法调用，并传入某个对象作为 this 引用；或
- 调用一个函数时，对象从参数入口传入。

在下面的代码中，我们以第四种情况为例来讲述一下函数实例被创建和引用的过程（注意下例中的匿名函数共产生了三个函数实例，但表现出来的生命周期并不一致）：

```
function MyObject(obj) {
  var foo = function() {
    // ...
  }

  if (!obj) return;
  obj.method = foo;
}

// 示例 1
MyObject();

// 示例 2
MyObject(new Object());

// 示例 3
var obj = new Object();
MyObject(obj);
```

在示例 1 中，`MyObject()`被调用，在函数内部有一个匿名函数的实例被创建，并被赋值给 `foo` 变量，但因为参数 `obj` 为 `undefined`，所以该函数实例没有被返回到 `MyObject()`函数外。因此 `MyObject()`执行结束后，闭包内的数据未被外部引用，闭包随即销毁（这与内存回收算法有关，因此这里并不强调实时销毁），`foo` 指向的匿名函数也被销毁。

在示例 2 中，传入参数 `obj` 是一个有效的对象，于是匿名函数被赋值给 `obj.method`，因此建立了一个引用。在 `MyObject()`执行结束的时候，该匿名函数与 `MyObject()`都不能被销毁。但随后，由于传入的对象未被任何变量引用，因此随即销毁了；而后 `obj.method` 的引用得以释放。这时 `foo` 指向的匿名函数没有任何引用、`MyObject()`内部也没有其他数据被引用，因此开始 `MyObject()`闭包的销毁过程。

在示例 3 中开始的过程与示例 2 一致，但由于 `obj` 是一个在 `MyObject()`之外具有独立生存周期的外部变量，JavaScript 引擎必须对这种持有 `MyObject()`闭包中的 `foo` 变量（所指向的匿名函数实例）的关联关系加以持续地维护，直到该变量被销毁，或它的指定方法（`obj.method`）被重置、删除[44]时，它对 `foo` 的引用才会得以释放。例如：

```
// 重新置值时, 关联关系被清除
obj.method = new Function();

// 删除成员时, 关联关系也被清除
delete obj.method;

// 变量销毁(或重新置值)导致的关联关系清除
obj = null;
```

对于示例 3，在对象销毁时，该对象持有的所有函数的闭包将失去对该对象的引用。而当一个函数实例的所有引用者都被销毁时，函数实例（及其闭包、调用对象）被销毁。

[44] 该项不适用于使用 `var` 声明的变量，因为 `var` 声明的变量不能被 `delete` 操作删除。

这看起来很完美：在纯粹的函数式语言中，在一个闭包中构造的对象只能被自己持有，或者通过函数返回以便被其他闭包引用——这显然是能被引擎感知的——所以函数与闭包总能在确知的情况下被销毁。然而在 JavaScript 中，由于函数内无法做到无副作用，因此它必须为每一个在函数内创建的对象创建一个引用列表，只有当所有函数内的对象（不仅仅是内嵌函数）都不再被引用时，该函数才处于自由（free）的状态。

然而有些对象总不能被销毁（例如在 DOM 中存在的泄漏），或在销毁时不能通知到 JavaScript 引擎，因此也就有些 JavaScript 闭包总不能被销毁。这是在某些具体的宿主环境中，常常因为宿主的使用方法导致 JavaScript 存在泄漏的根源。一个通常的例子是：

```
<!-- code form codeproject.com, uses in IE, By volkan.ozcelik. -->
<script type="text/javascript">
function LeakMemory(){
  var parentDiv = document.createElement("<div onclick='foo()'>");
  parentDiv.bigString = new Array(1000).join(
                        new Array(2000).join("XXXXX"));
}
</script>

<body>
<input type="button" value="Memory Leaking Insert"
  onclick="LeakMemory()" />
</body>
```

这个例子中，`LeakMemory()`中创建了一个 DOM 对象 `parentDiv`，然后该对象又持有了在 `LeakMemoery()`中创建的一个大数组。当 `LeakMemory()`函数退出时，因为 DOM 并不释放临时的 `parentDiv` 对象，所以 JavaScript 也不能释放 `LeakMemory` 闭包和那个创建的大数组，于是产生了内存泄漏[45]。

4.6.5 函数实例拥有多个闭包的情况

一般情况下，一个函数实例只有一个闭包，在闭包中的数据（闭包上下文）没有被引用时，该函数实例与闭包就被同时回收了——我们在此前也主要讲述这种情况。但也存在函数实例有多个闭包的情况——这非常罕见。下面特别构造了这样一个例子：

```
39  var checker;
40
41  function myFunc() {
42    if (checker) {
43      checker();
44    }
45
46    alert('do myFunc: ' + str);
47    var str = 'test.';
48
49    if (!checker) {
50      checker = function() {
51        alert('do Check:' + str);
52      }
53    }
54
55    return arguments.callee;
56  }
57
58  // 连续执行两次 myFunc()
59  myFunc()();
```

在这个例子中，`myFunc` 函数将执行两次。在第 21 行的第一次执行结束时，在第 17 行代码处将“函数实例自身的一个引用”（`callee`）作为结果值返回；接下来，在 21 行处会遇到第

[45] 这个泄漏在 IE 7 中已经被修复。

二个函数调用运算符，于是 myFunc 函数就被第二次调用了。由于第二次执行的只是一个引用，因此这个示例中 myFunc 只创建过一个函数实例。

运行这个例子将输出三个信息：

```
do myFunc:  undefined
do Check: test.
do myFunc: undefined
```

其中第一、三个信息都由第 8 行代码输出。输出 undefined 的原因在于：

- 函数被调用时，函数内的局部变量被声明并被初始化为 undefined；
- 局部变量表被保存在该函数闭包中的 varDecls 域中。

第 8 行代码用于检测该值——并且我们强调它的值的确是 undefined。但是我们也注意到第二个输出信息是“test.”，根据代码流程，该值是第二次调用 myFunc 时输出的——因为在第一次调用 myFunc 时，checker 还被有被赋过值呢。

但所输出的 checker 值，却是在第一次调用中被完成赋值的——它是一个局部函数引用；并且在它的闭包中，还引用了 upvalue 变量 str。由于这个变量是第一次调用的 myFunc 函数的闭包中的，因此在第一次 myFunc 调用结束后，闭包并没有被销毁——闭包中存在被其他对象引用的变量 / 数据。

所以，myFunc 第二次调用并以函数形式调用全局变量 checker 时，checker 实际上是输出了“第一次 myFunc 调用过程中形成的闭包”中的 str——显然，这个值是“test.”。这就是第二个输出信息的由来。

这个例子传达出的信息是：

- JavaScript 中函数实例可能拥有多个闭包；
- JavaScript 中函数实例与闭包的生存周期是分别管理的；
- JavaScript 中函数被调用——即“()”运算时总是初始化一个闭包；而上次调用中的闭包是否销毁，取决于该闭包中是否有被（其他闭包）引用的变量 / 数据。

我们注意到这里提及“函数实例与闭包的生存周期是分别管理的”。因此一个函数实例（以及其可能的多个引用）的生存周期，与闭包是没有直接关系的。换而言之，会存在函数变量没有持有闭包的情况。这是因为：

- 闭包创建自函数执行开始之时；接下来，
- 在执行中闭包没有被其他对象引用；接下来，
- 在函数执行结束时闭包被销毁了。

而这时函数实例及其引用（例如变量、对象成员、数组元素等）都还存在，只是没有与之对应的闭包了。所以第 4.6.3 小节的标题是“（在被调用时，）每个函数实例至少拥有一个闭包”，以强调“在调用过程中”这样的事实。

本小节则另外强调，函数实例在没有被调用时可能有对应的（一个或多个）闭包，也可能没有闭包——当然，还可能闭包失效了，但未被引擎的内存管理器回收，而这就不在本书要讨论的范围中了。

4.6.6 语句或语句块中的闭包问题

一般情况下，当一个函数实例被创建时，它唯一对应的一个闭包也就被创建。在下面的代码中，由于外部的构造器函数被执行两次，因此内部的 `foo` 函数也被创建了两个函数实例（以及闭包）并赋值给 `this` 对象的成员：

```
function MyObject() {
  function foo() {
  }

  this.method = foo;
}

obj1 = new MyObject();
obj2 = new MyObject();

// 显示 false, 表明产生了两个函数实例
alert( obj1.method === obj2.method );
```

这在函数之外（语句级别）也具有完全相同的表现。在下面的例子中，多个匿名函数的实例被赋给了 `obj` 的成员：

```
var obj = new Object();
for (var i=0; i<5; i++) {
  obj['method' + i] = function() {
    //...
  };
}
// 显示 false, 表明产生了多个函数实例
alert( obj.method2 === obj.method3 );
```

尽管是多个实例，但它们仍然是共享同一个外层函数闭包中的 upvalue 值——在上例中，外层函数闭包指的是全局闭包。因此下面例子所表现出来的，仍然只是闭包中对 upvalue 值访问的规则，而并非闭包或函数实例的创建规则“出现了某种特例”[46]：

```
var obj = new Object();
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
   obj[e] = function(){
      alert(events[e]);
   };
}
// 显示 false, 表明是不同的函数实例
alert( obj.m1 === obj.m2 );

// 方法 m1()与 m2()都输出相同值
// 其原因，在于两个方法都访问全局闭包中的同一个 upvalue 值 e
obj.m1();
obj.m2();
```

在这个例子中，方法 m1 与 m2 究竟输出何值，取决于前面的 for..in 语句在最后一次迭代中对变量 e 的置值。某些引擎中总保证 for..in 的顺序与 events 中声明时的属性顺序一致（例如 SpiderMonkey），但也有一些引擎并没有这项约定。因此上例在不同的引擎中表现的结果未必一致，但 m1()与 m2()输出值总是相同的。

按照这段代码的本意，应该是每个函数实例输出不同值。对这个问题的处理方法之一，是再添加一个外层函数，利用“在函数内保存数据”的特性来为内部函数保存一个可访问的 upvalue：

```
var obj = new Object();
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
   obj[e] = function(aValue) {  // 闭包 lv1
     return function() {  // 闭包 lv2
       alert(events[aValue]);
     }
   }(e);
}

/* 或用如下代码, 在闭包内通过局部变量保存数据
for (e in events) {
   function() {  // 闭包 lv1
     var aValue = e;
     obj[e] = function() {  // 闭包 lv2
       alert(events[aValue]);
     }
   }();
}
*/

// 下面将输出不同的值
obj.m1();
obj.m2();
```

由于闭包 lv2 引用了闭包 lv1 中的入口参数，因此两个闭包存在了关联关系。在 obj 的方法未被清除之前，两个闭包都不会被销毁，但 lv1 为 lv2 保存了一个可供访问的 upvalue——这除了私有变量、函数之外，也包括它的传入参数。

很明显，上例的问题在于多了一层闭包，因此增加了系统消耗。不过这并非不可避免。我们要看清楚这个问题，其本质是要一个地方来保存 for..in 列举中的每一个值 e，而并非是“需要一个闭包来添加一个层”。那么，既然列举过程中产生了不同的函数实例，自然也可以将值 e 交给这些函数实例自己去保存：

```
var obj = new Object();
```

[46] 这个例子引自一份存有误解的网络文档（http://blog.csdn.net/g9yuayon/archive/2007/04/18/1568980.aspx）。

```
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
  (obj[e] = function() {
     // arguments.callee 指向匿名函数自身
     alert(events[arguments.callee.aValue]);
   }
  ).aValue = e;
}

// 下面将输出不同的值
obj.m1();
obj.m2();
```

4.6.7 闭包中的标识符（变量）特例

在下面两个小节中：

- 3.2.4.5 变量作用域中的次序问题
- 3.2.4.6 变量作用域与变量的生存周期

在对函数内声明的变量进行阐述的时候，说变量标识符在定义后值为 `undefined`，并可以在函数内访问。例如：

```
var name = 'test';
function myFunc() {
  // 输出 1
  alert(name);

  // 输出 2
  var name = 100;
  alert(name);
}
```

在这个例子中，输出 1 不会访问到全局变量 `name`，是因为 `name` 已经在函数闭包内被声明（但未被赋值）过；输出 2 则输出该局部变量 `name` 赋值后的值。

但是完整地说来，函数闭包内的标识符系统应该包括：

- this；
- 局部变量（varDecls）；
- 函数形式参数名（argsName）；
- arguments；
- 函数名（funcName）。

其中 arguments 是语言内定的标识符，无需声明即可使用。

上面列举标识符系统时是按优先顺序的。这种优先顺序（及其隐含规则）甚至超出了我们前面讲到过的变量作用域规则、闭包规则。首先我们来看看下面这个例子：

```
// 示例 1: 输出值'hi', 而非函数 foo.toString()
function foo(foo) {
  alert(foo);
}
foo('hi');

// 示例 2: 输出数值 100 的类型'number', 而非参数对象 arguments 的类型'object'
function foo2(arguments) {
  alert(typeof arguments);
}
foo2(100);

// 示例 3: 输出参数对象 arguments 的类型'object'
// (注: 在 JScript 中, arguments 作为函数名可以被声明, 但调用该函数导致脚本引擎崩溃)
function arguments() {
  alert(typeof arguments);
}
arguments();
```

这几个示例表明：

- 形式参数名优先于内置对象 arguments；
- 内置对象 arguments 优先于函数名；
- 综合上两项（以及参考示例 1 的效果），形式参数名优先于函数名。

所以，从优先级上来看是“argsName > arguments > funcName”。但是如果与局部变量 varDecls 比较来看，又会是如何呢？下例说明这种情况：

```
// 示例 4: 输出值'test'
function foo(str) {
  var str;
  alert(str);
}
foo('test');
```

这个例子中，变量 str 显然没有作为局部变量（varDecls）处理，而是函数入口处的形式参数，所以才输出值'test'。但接下来这个例子却又有点不同：

```
// 示例 5: 输出值'member'
function foo(str) {
  var str = 'member';
  alert(str);
}
foo('test');
```

这个例子表明：局部变量（varDecls）优先级高于函数形式参数。这又与上一个例子是矛盾的。比较来看，我们可以直观地认为：

- 当形式参数名与未赋值的局部变量名重复时，取形式参数值；
- 当形式参数与有值的局部变量名重复时，取局部变量值。

我并不确知这样实现的真实原因，但是这显然带来了一种语义上的合理性：

```
// 示例 6: 输出值'test'
function foo(str) {
  var str = str;
  alert(str);
}
foo('test');
```

如果仅认为“局部变量（varDecls）优先级高于函数形式参数”，那么该代码“var str=str”赋值运算的左右两侧都应该是局部变量 str；而局部变量声明后初值为 undefined，所以该行代码的结果“（看起来）应该”是 undefined。

然而这可能与我们的目的不一致[47]，我们的目的，可能是将入口参数 str 赋值局部变量 str。那么，就只能是上述两条假设都同时能立，才能使接下来这行“alert（str）”既访问局部变量，又能输出有效值——“字符串'test'”。这一语义上的合理性是：赋值运算符的左侧是局部变量——这是由 var 关键字决定的，而右侧是形式参数——这是因为此时局部变量 str“尚未赋值”。

这样一来，赋值语句就成立了，接下来 str 变成有值的局部变量，而 alert()也就会访问它了。不过，这里的“有或没有赋值”是指显式的赋值操作。仅以变量声明来讲，它是会有一个 undefined 初值的，而“是不是 undefined”并不是上述判断的条件。

4.6.8 函数对象的闭包及其效果

这里说的“函数对象”，是特指使用 Function()构造器创建的函数。它与函数直接量声明、匿名函数不同，它在任意位置创建的实例，都处于全局闭包中。亦即是说，Function()的实例的 upvalue 总指向全局闭包。下例说明这一点：

```
var value = 'this is global.';

function myFunc() {
  var value = 'this is local.';
  var foo = new Function('\
    alert(value);\
  ');

  foo();
}

// 显示'this is global.', 表明 foo 访问到全局闭包中的 value 变量
myFunc();
```

这其中的原因，在于 Function()构造器传入的参数全部都是字符串，因此不必要与函数局部变量建立引用。由此带来的一个便利是：在任意多层的函数内部，都可以通过 Function()

[47] 当然，开发人员预期目的未必如此，但语言设计者总得为这样的代码而设定一个“可被理解的语义”。

创建函数而不至于导致闭包不能释放。所以，对于上一小节的示例，更加妥当的写法应该是：

```
var obj = new Object();
var events = {m1: "clicked", m2: "changed"};

for (e in events) {
  obj[e] = new Function('alert(events["' + e + '"])');
}

// 下面将输出不同的值
obj.m1();
obj.m2();
```

这样，系统就不会维护多余的闭包，也不会在函数实例上添加多余的成员了。不过你也应该注意到，这里用于构造函数的字符串

```
  'alert(events["' + e + '"])'
```

中，变量 e 是一个字符串。这的确是一个限制：在 Function()构造器中，在通过这个字符串参数来保证没有对变量的引用时，变量是先转换为字符串值的——因此，如果该变量在转换时会失去原意（例如 Object/ActiveXObject 等），那么这个方法显然不可行。

除开上述的限制，你还是可以拿这个特性来做很多事的。例如下面的函数用来为你的函数定制一个特别的 toString()效果：

```
function myFunction(name) {
  var context = [
    "return 'function ", name, "()\\n",
    "{\\n",
    "  [custom function]",
    "\\n}'"
  ];
  return new Function(context.join(''));
}

function aFunc() {
  //...
  //...
  //...
}

// 显示默认的 toString 效果
alert(aFunc);

// 显示一个定制的 toString 效果
aFunc.toString = myFunction('aFunc');
alert(aFunc);
```

在这个例子中，因为使用了 new Function()，所以在 myFunction()中产生的函数实例不会引用它的入口参数 name。这样一来，无论多少次调用 myFuncion()都不会建立更多的闭包——如果写成下面的代码，尽管结果一致，但系统需要维护的 myFunction()实例数或闭包数却会大量增加：

```
function myFunction(name) {
  return function() {
    return 'function ' + name +
      '()\n{\n  [custom function]\n}';
  }
}
```

4.6.9 闭包与可见性

作为动态语言实现上的一种结果，语法作用域、变量作用域、变量生存周期等在 JavaScript 中变成了不同的概念。本书分别在下面的章节中讲述过它们。

- 语法作用域：语法形式上的结构化效果，讲述于“3.2.2 模块化的层次：语法作用域”；

- 变量作用域：亦即变量的可见性[48]，讲述于“3.2.4　模块化的效果：变量作用域”；
- 变量生存周期：变量分配存储到销毁的过程，讲述于“4.6.4　函数闭包与调用对象”。

除了变量作用域（及其可见性）之外， JavaScript 中还存在对象成员的可见性问题——这是与 `with` 语句相关的语言特性。本章节将引入“对象闭包”的概念来解释这一特性。

4.6.9.1　函数闭包带来的可见性效果

在表现上来看，JavaScript 对可见性的规定是：

- 变量在全局声明（用或不用 `var`），则在全局可见。
- 变量在代码任何位置隐式声明（不用 `var`），都在全局可见。
- 变量在函数内显式声明（用 `var`），则在函数内可见。

如图 4-9 所示。

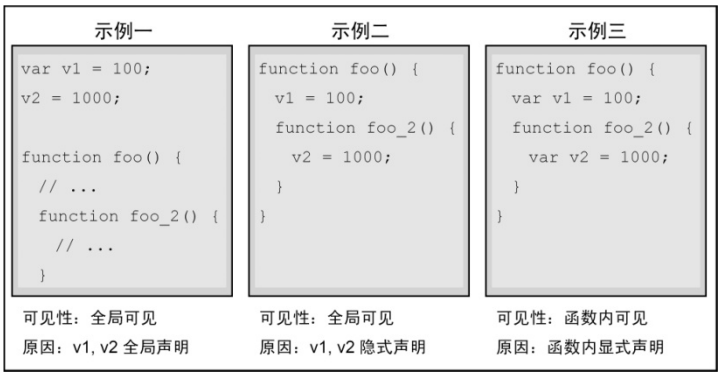


图 4-9　JavaScript 对可见性的规定

以下两条是对上述规则的补充。

- 可见性传递：变量 A 在函数内可见，则在其所有内嵌函数中可见。如图 4-10 所示。

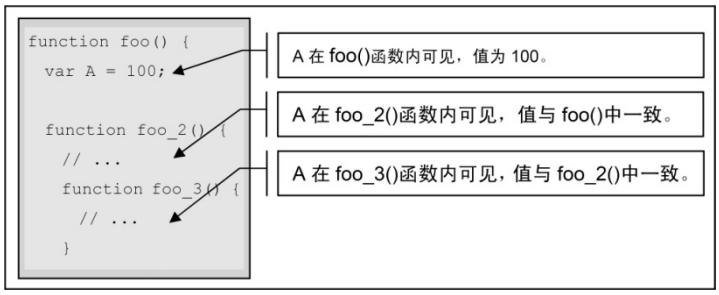


图 4-10　可见性传递规则

- 可见性覆盖：在函数内显式声明变量 A，将覆盖当前可见的同名函数。如图 4-11 所示。

[48] 标识符可见性包括对象成员标识符、标签标识符、变量标识符等的可见性，因此它与变量作用域并不严格等义，而我们这里只强制变量的可见性。

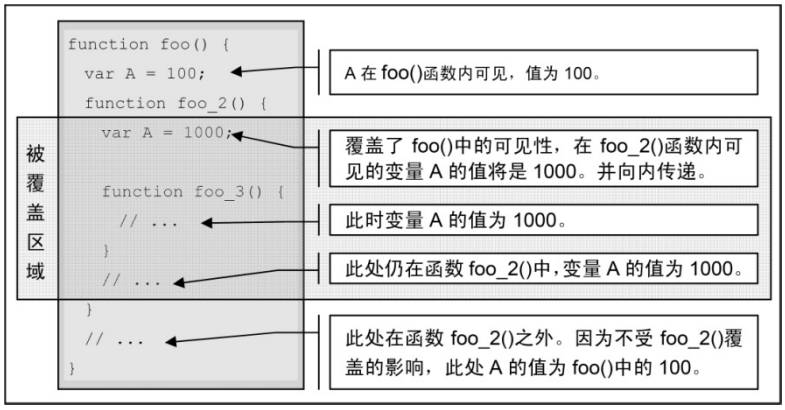


图 4-11　可见性覆盖规则

但在具体的实现上，JavaScript 只要求在语法分析期：

- 将一个函数代码体中所有用 var 关键字声明的变量记入它自己的 ScriptObject 的 varDecls 域，然后，
- 设定访问规则“如果当前函数的 ScriptObject.varDecls 域中不存在该变量声明，则通过‘闭包.parent’来取得上一层函数的 ScriptObject.varDecls 作为 upvalue”

那么必然会得到 JavaScript 中那些“看起来非常复杂”的语言特性。以图 4-10 和图 4-11 为例：

- 可见性传递的实质是 foo_2()与 foo_3()都能访问到来自于 foo() 的同一份 ScriptObject.varDecls；
- 可见性覆盖的实质是 foo_3()在 foo_2()的 ScriptObject.varDecls 中找到了名为 A 的变量，因而不必再向上层回溯；
- “变量在代码任何位置隐式声明（不用 var），都在全局可见”的实质，是该变量标识符不在函数所有 parent 的 ScriptObject.varDecls 中存在时，必然回溯至顶层的全局函数闭包（通常实现为宿主对象）的中的 varDecls，并在该位置“隐式地”声明了一个变量。

综上所述，我们在第 3 章讲述“JavaScript 的非函数式语言特性”时，讨论到的 JavaScript 中的“变量作用域”几个问题，包括（参见“3.2.4　模块化的效果：变量作用域”）：

- 只有表达式、局部和全部三种作用域；
- 变量生存周期不依赖于语法的声明顺序；
- 生存周期只有两个——函数内的局部执行期间和函数外引擎的全局执行期间。

其中除了“表达式作用域”依赖于匿名函数的特性之外，其他的所有现象其实都是“函数闭包”这种特性带来的效果。更加确切地说，它们仅仅依赖于在“4.6.4　函数闭包与调用对象”中讲述的调用对象（ScriptObject）中的 varDecls 域的访问规则。

最后补充一点，正是由于每一个函数实例都有一份 ScriptObject 的副本，因此“不同的闭包访问到的私有变量不一致”。

4.6.9.2　对象闭包带来的可见性效果

所谓“对象闭包”，是指使用 with 语句时与 with()所指示对象相关的闭包，该闭包被动态创建并添加到执行环境当前的闭包链顶端[49]。如果我们将闭包的可见性理解为“闭包的 upvalue 系统”与“闭包内的标识符系统”，对象闭包与函数闭包在前者上是一致的，但对后者的处理并不相同。表 4-3 说明了这种差异：

表 4-3　函数闭包与对象闭包的差异

标识符系统	函数闭包	对象闭包	说明
this	有	没有	（*注 1）
局部变量（varDecls）	有		（*注 2）
函数形式参数名（argsName）	有	没有	仅与函数声明相关

[49] 在 SpiderMonkey 中，with 语句不是打开对象闭包的唯一方式。首先，用 Object.eval()可以让一段代码执行在对象闭包中，例如“obj.eval('value = 100')”；将在 obj 的对象闭包中执行代码。此外，Script()对象也可以强制在一个对象的闭包中执行代码，例如“new Script('value = 100').exec(obj)”就与上面的作用是一致的。

arguments	有	没有	仅与函数调用过程相关
函数名/对象名（funcName/objName）	有	没有	对象直接量不能具名
对象成员名	没有	有	（*注 3）

*注 1：在对象闭包中使用 this 引用，将通过 upvalue 访问到它上一层的函数中的 this 引用。

*注 2：对象闭包中的 var 声明效果存在引擎差异。

*注 3：在函数闭包中，arguments 也是函数的一个属性。但函数内能直接访问 arguments，是因为函数闭包将一个引用了该属性的变量添加到闭包标识符系统中，与（函数作为对象时的）“对象闭包”并没有什么关系。

在对象闭包中用 var 声明变量的效果，与具体的引擎实现有密切关系。在目前的考察中，JScript、SpiderMonkey JavaScript、Adobe ActionScript 等都将 with 语句中的 var 关键字理解为所在函数闭包中的变量声明。如下例：

```
// 示例 1
function foo() {
  with (this) {
    var value;
  }
  alert(value);
}
// 显示'undefined'
foo();
```

由于“var value”被理解为 foo()函数中的一个变量声明，且该变量初值为 undefined，因此该示例显示'undefined'。

当该关键字 var 被理解为 foo()中的变量声明时，就存在了一个问题：如果 with 所指示的对象闭包中存在同名的属性，那么 value 应当操作哪个标识符？例如：

```
// 示例 2
var aObj = { value: 'hello' };
function foo() {
  with (aObj) {
    var value = 1000;
    alert(aObj.value); // 显示值: 1000
  }
  alert(value);
}
// 显示'undefined'
foo();
```

对于这个例子，上述脚本引擎采用了一致的策略：由于对象 aObj 存在 value 属性，因此“value = 1000”将向 aObj.value 置值。也就是说，由于当前最顶层的闭包是 aObj 的对象闭包，且该闭包存在“value”这个标识符（或属性名），因此应当向它置值。

表面看起来这很合理，但它与上一项处理策略是冲突的。因为：

- “var value”向 foo()函数的闭包声明了一个局部变量；而
- “value = 1000”操作的是 aObj 对象闭包中的一个属性

这使得在不同的阶段中，同一行语句存在面向两种不同目标的语义与执行效果。这显然是不合理的。

为了避免上述的困惑，一些引擎选用了别的方案。例如用 Apple Safari 中的 JavaScriptCore（基于 KJS 引擎）来测试示例 2 时，第二个输出就会显示值“1000”而不是“undefined”。这是因为 KJS 引擎系列将“var value = 1000”作为一个语义来理解，所以即使 aObj.value 属性存在，也将操作 foo()函数闭包中的 value。

无论如何，上述这些引擎都认为在对象闭包中声明的标识符总是位于上层的函数闭包中的——换句话说，因为只有函数闭包具有 varDelcs，所以也只有它才能接受标识符声明[50]。但除此之外，也有一些脚本引擎在对象闭包中解释 var 的语义——这也意味着该引擎会在执行过程中使用 var 动态声明变量，例如 DMonkey——一个用 Delphi 实现 JavaScript 引擎的开源项目。下面的代码说明在 DMonkey 中，with 语句中的 var 在对象闭包中声明了一个'value'变量标识符，且该标识符不是“对象的一个成员”：

[50] KJS 与 SpiderMonkey JavaScript 引擎共同存在的这种前提，使它们在处理 with 语句所打开的对象闭包中存在“条件化函数声明（Conditional Function Declaration）”时达成了一致。关于条件化函数声明，请参见“5.4.2.1 语法声明与语句含义不一致的问题”。

```
// 示例 3: 测试 DMonkey 中的效果
var value = 10;
var aObj = { value: 'hello' };
function foo() {
  with (aObj) {
    var value = 1000;
    alert(aObj.value);  // 输出字符串: 'hello'
    alert(value); // 输出值: 1000
  }
  // 上面的 var 只作用于对象闭包内部, 所以这里的 value 指向了函数外的全局变量, 输出值: 10
  alert(value);
}
foo();
```

DMonkey 与 KJS 相同之处在于“`var value = 1000`”语句都不会影响到 `aObj.value` 的值；不同的是，KJS 认为该行语句的效果作用于函数闭包 `foo()`，而 DMonkey 认为效果作用于 `aObj` 的对象闭包中。

对象闭包存在一个最明显的易用性问题，就是对“对象直接量”闭包中如何访问该对象自身。这个问题在匿名函数闭包中同样存在，但是我们总可以通过 `arguments.callee` 来访问到该匿名函数。然而对象闭包中没有 `arguments`，也没有（对该对象有效的）`this` 引用，那么下面的代码有什么办法去访问到 `with()`操作的对象呢：

```
with ( {} ) {  // <-- 这里声明了一个匿名的对象直接量
  // <--这里如何访问对象直接量自身呢?
}
```

在 `with` 语句中访问对象自身至少有三个目的：其一是为该对象添加成员，其二是访问下标（对于数组）或无法用标识符存取的成员，其三是使用类似于 `for..in` 这种需要操作对象的语句。这些问题可以通过将对象直接量赋给一个标识符的方法来解决，例如：

```
var x;
with ( x={} ) {  // <-- 这里声明了一个匿名的对象直接量
  // 通过变量 x 访问对象引用
}
```

也可以通过下面的 `self()`函数来解决：

```
function self(x) {
  return x.self = x;
}

// 将所声明的对象直接量用作 self 的参数
with ( self({}) ) {
  // 通过 self 成员访问对象引用
  self.value = 100;
  for (var i in self) {
    // ...
  }
}
```

函数闭包与对象闭包既有相关性，也有各自的独立性。对象闭包总是动态添加在闭包链顶端，而函数闭包则依赖于函数声明时的、静态的语法作用域限制。因此，二者可能出现不完全一致的情况，这很容易让人产生（至少在表面上的）困惑。例如：

```
60  /**
61    * 示例 4: 两种闭包的交叉作用
62    */
63  var obj = { value: 200 };
64  var value = 1000;
65  with (obj) { // <-- 对象闭包
66    function foo() {  // <-- 具名函数 foo()的闭包
67       value *= 2;    // <-- 依赖于函数静态位置所决定的闭包链
68    }
69    foo();
70  }
71
72  // 显示 200
73  alert(obj.value);
74  // 显示 2000
75  alert(value);
```

在第 6 行，我们可能预期是要限定第 6~11 行中所有对 `value` 存取都指向 `obj.value`。但由于第 7~9 行在一个函数（的闭包）作用域里，因此 `with` 的限定事实上只对第 10 行有效（尽管第 10 行并不直接存取 `obj.value`）。于是，我们看到第 8 行通过变量作用域的影响而访问到了全局变量 `value`：使它的值乘 2，变成了 `2000`。

这段代码中同时出现了两种闭包。在形式上来看，`foo()`的函数闭包位于 `obj` 的对象闭包中，因此 `foo()`中访问 `value` 时应该通过闭包链来存取到 `obj.value`。但事实上，`foo()`函数的闭包链是在语法分析期就决定了的：它被添加在全局闭包之后；而另一方面，`with()`在运行期打开了 `obj` 的对象闭包，由于 `with` 语句也处在全局闭包中，因此 `obj` 的对象闭包就被“静态地”添加到了全局闭包之后。结果是：`obj` 对象闭包与 `foo()`函数现在处于并列位置。所以在第 8 行就无法访问到 `obj.value` 了——`foo()`函数的闭包链上找不到 `obj` 的对象闭包。

另一种存在两种闭包的情况是：在函数中打开函数对象自身的闭包。例如：

```
function foo() {
  with (arguments.callee) {
    // 这里即处理 foo()函数的函数闭包中，又处于它作为对象时的对象闭包中
  }
}
foo();
```

我们在“7.12　使用对象闭包来重置重写”的最后部分，在讲述到有关 `Function.prototype.bind()`的技术时，其实就用到了这种两种闭包同时存在的效果，尽管它没有带来特别的“好处”，但的确让我们看到了两种闭包不同的性质。

4.6.9.3 匿名函数的闭包与可见性效果

有一种做法可以避免上面提到的“foo()函数中不能访问 obj.value”的问题。这种做法是：将函数 foo()添加为对象 obj 的方法。例如：

```
76  (部分代码参见 4.6.9.2 小节的示例 4)
77  with (obj) { // <-- 对象闭包
78    obj.foo = function() { // <-- 匿名函数的闭包
79       value *= 2;     // <-- 依赖于函数闭包所在的当前闭包链
80    }
81    obj.foo();
82  }
```

这样一来，obj.value 值将变成 400，而全局的 value 值不变。这是因为匿名函数的闭包也是如同对象闭包一样，动态地添加到当前闭包链的顶端——而代码执行到第 3 行时，“当前闭包”正好是 with 所打开的 obj 对象闭包。

由于这里的“添加到闭包链顶端”的行为只是（引擎针对于）匿名函数自身的行为，所以它与上述代码的赋值操作无关。也就是说，即使该匿名函数没有添加为对象 obj 的方法——而仅是即用即声明，那么它所操作的仍然是对象闭包中的 value 值。例如：

```
83  (部分代码参见上例)
84  with (obj) { // <-- 对象闭包
85    void function() { // <-- 函数闭包
86       value *= 2;
87    }();
88  }
```

更细致地追究这个问题，其实匿名函数“添加到闭包链顶端”也与“执行”无关，而是在它作为一个直接量被“创建”时，由引擎动态添加在闭包链上的。也就是说，匿名函数直接量的创建位置决定了它所在闭包链的位置。例如：

```
function foo() {
  function foo2() {
    // foo2()函数内的局部变量声明
    // ...
    var msg = 'hello.';

    // 使外部变量 m 引用到该匿名函数
    m = function(varName) {
      return eval(varName);
    }
  }
  foo2();

  // aVarName 是 foo2()中的任意局部变量名, 例如'msg'
  var aFormatStr = 'the value is: ${aVarName}';
```

```
  var m;
  var rx = /\$\{(.*?)\}/g;
  alert(aFormatStr.replace(rx, function($0, varName) {
    return m(varName);
  }));
}
foo();
```

这个例子中，可以通过改变 aFormatStr 中的'aVarName'来取得任意 foo2()函数中的局部变量的值。而前提是：让外部代码持有 foo2()的一个匿名函数的引用。通过这种方法，foo2()函数无论在任何位置——包括 foo()的闭包内，或者任何 foo()无法访问到的位置，函数 foo()都可以访问到它的内部成员[51]。

[51] 这一技术被内置于 Qomo V2 的内核，并由此带来了一种类似于 PHP 中的允许 echo()输出带有变量名的字符串，并动态将这些变量转换为值的技术。

CHAPTER

第 7 章

一般性的动态函数式语言技巧

相对于某些特性更纯粹的语言，JavaScript 中看起来非常技巧化的处理方法，事实上是充分利用了函数式、原型继承、动态语言等复合的语言特性的“一般性技术”。由于涉及了过多的、不同分类系统下的语言特性，这些技巧与其他语言不相容，或者看起来颇为另类。然而作为语言的研习者，应有能力剖析这些技术，并从中找到所谓技巧的根本。

本章用条目的方式，归纳一些在 JavaScript 中常见的技巧。除了指出这些技术所涉及的语言特性（其绝大多数技巧的特性是复合的）之外，也给出它的应用、限制与更加复杂的变化。

7.1 消除代码的全局变量名占用

说明：

解决函数外执行的代码（行）可能占用大量全局变量名的问题。

示例背景：

一段代码运行在全局闭包中时，会占用一些全局标识符名。当后续代码使用到相同标识符名时，可能导致不可预测的逻辑错误。因此可以将这些代码“包装”在一个匿名函数闭包中，使用“声明即运行”的技术，既完成代码执行，又将代码的副作用限制在函数内部的有限范围之中。

示例代码：

```
/**
 * 1. 基本示例
 */
void function() {
  // ...
}();

/**
 * 2. 带参数的示例
 */
void function(x, y, z) {
  //...
}(10, 20, 300);

/**
 * 3. 执行后不会被释放的示例
 */
void function() {
  // 将一个内部的(引用类型的)变量赋给全局变量或其成员
  String.prototype.aMethod = function() {
    // ...
  }
}();
```

示例说明：

我们在“2.4.1.3　函数调用语句”中讲到过上述的例子，主要是从词法分析的角度来解释它。但是，这个例子应当被称为“声明即调用的匿名函数”。它主要解决的问题是：用闭包隔离代码并执行。

如同模式中的单例一样，这种函数只被创建一个函数实例，并立即执行。如果可能，它也将在执行后立即被释放和回收。由于从语法上讲，函数块本身具有“声明逻辑相关的连续代码”的作用，因此你也可以用这种技巧来（结构化地）组织代码并为它（这个匿名函数）添加有意义的注释。这比将代码行分散在全局闭包的各处要好。

如果在这种匿名函数内部，将一些函数内引用赋值给了外部的、全局的变量引用，例如某些内部对象、DOM 对象的成员，那么这个匿名函数在执行后并不会被释放（示例 3）。这会导致匿名函数的生存周期依赖于该外部变量——被重写或删除。

因此如果的确有这样的需求，应当将更多的外部操作集中在同一个匿名函数中，并减少该函数中无关的逻辑代码，例如在同一个“声明即调用的匿名函数”内为 `String.prototype` 扩展多个原型方法。

7.2 一次性的构造器

说明：

只使用一次的构造器函数。

示例背景：

一个构造器函数在使用一次之后，标识符即被重写。但可以保证“<实例>.constructor”正确地指向构造器，且允许使用 new 运算来创建更多的实例。

示例代码：

```
/**
 * 1. 基本示例: 使用标识符重写
 */
Instance = function() { ... }
Instance.prototype = {
  // 直接量声明
  ...,
  // 维护原型: 置构造器属性
  constractor: Instance
}
Instance = new Instance();
Instance2 = new Instance.constructor();

/**
 * 2. 使用匿名函数
 */
Instance = new function() {
  // or arguments.callee.prototype
  var proto = this.constructor.prototype;

  // 维护原型 proto
  // ...
}

/**
 * 3. 在示例 2 中使用参数
 */
Instance = new function(x,y) {
  //...
}(1,2);
```

示例说明：

示例 1 使用重写标识符的方法来解释了这一技巧的主要目的，也同时说明了如何有效地在重写原型时维护 constructor 属性。当对象系统正确维护 constructor 属性时，我们就总可以使用“new <实例>.constructor()”来创建更多的实例，而不必依赖于构造器函数的名字。

匿名函数同样具有“一次性”的特点，因此示例 2 做了一个相同功能的演示。在示例 2 中，依赖于：

- 构造器函数的原型的 constructor 属性默认指向自身；
- 实例（示例 2 中的 this 引用）从原型中继承 constructor 属性。

这两个性质来找到匿名的构造器函数，并操作它的原型引用。此外也可以依赖参数对象的属性 arguments.callee 来找到该构造器，这是在匿名函数内部访问函数自身的标准方法。与示例 1 相同的是，如果示例 2 不是修改而是重写原型的话，那么它也需要维护好原型的 constructor 属性。

示例 3 说明在 new 中使用参数表的特殊性。这与函数调用不同的是，我们不要以为这里的匿名函数后面有一个参数表，而使用一对括号来表明函数调用。例如：

```
new (<匿名函数>(1, 2));
```

这样的代码事实上是可能会导致语法异常的。因为这个所谓的“参数表”实际上是 new 运算符的一部分，而非构造器函数的一部分。因此它的语法含义事实上是：

```
new (<匿名函数>)(1, 2);
```

关于这部分的语义分析细节，请参见“2.5.1.1　使用构造器创建对象实例”。

7.3　对象充当识别器

说明：

用一个对象来充当函数调用界面上的，或针对特殊成员的识别器。

示例背景：

该技巧利用了对象实例在全局唯一性的特性（包括直接量空白对象“{}!=={}”）。

我们在一个函数闭包内声明的局部变量，是不会被外部代码直接访问到的，但能在该闭包中、更内层的子函数闭包中通过 upvalue 访问。因此我们可以用该局部变量作为一个识别器，来辨识函数是在内部亦或外部调用。

除了在函数调用界面上的识别外，我们也可以利用这种技巧来弥补 in 运算不能有效甄别对象内部成员的问题。

示例代码：

```
142 /**
143  * 1. 用作函数调用界面上的识别器
144  */
145 var myFunc = function() {
146   var handle = {};
147   function _myFunc(x,y,z) {
148     if (arguments[0] === handle)
149       // 是内部调用...
150     else
151       // 是外部调用...
152   }
153
154   // 内部调用测试（可传入更多参数）
155   _myFunc(handle, 1, 2, 3);
156
157   return _myFunc;
158 }
159 // 外部调用测试(不能访问变量 handle)
160 myFunc(1, 2, 3);
```

```
161
162 /**
163  * 2. 用作对象成员识别器
164  */
165 var obj = function() {
166   var yes = {};
167   var tbl = { v1: yes, v2: yes, v3: yes };
168   return {
169     query: function(id) { return tbl[id] === yes }
170   }
171 }();
172 // 测试: 查询指定 id 是否存在
173 obj.query('v1');
174 obj.query('constructor');
```

示例说明：

与示例 1 相同的模式，被应用在 Qomo 的切面与多投事件特性的实现中。在这两个 Qomo 特性中，内部函数_myFunc()掌握着一些特殊的信息（例如更内部的变量或受保护的信息）。一方面，在某些有关_myFunc()的内部调用时需要索取该信息，另一方面，又要避免它的外部引用 myFunc()在调用时索取该信息。这种情况下，示例 1 使用一个内部或外部调用的识别器 handle 来识别函数调用的上下文环境。

另一个识别函数调用的上下文环境的方法，是直接检测 arguments.callee.caller 以判定是否来自某个函数调用。这种方法在 Qomo 中也有应用，但它不适合对批量调用者的识别，另外也存在限制：调用者函数的标识符必须能被_myFunc()访问。

示例 2 被应用在 Qomo 的切面系统的实现中检测某个标识名是否存在。示例 2 创建的内部检索表 tbl 是一个对象，而不是使用通常的索引数组。对对象的成员做检测的方法，通常是使用 in 运算。但是 in 运算并不排除 construtor 这种基本属性，以及通过原型继承来的属性，因此往往要花大量代码消除误判。在本示例中，通过识别器变量 yes，能有效地避免这些问题，并具有与 in 运算相同的性能。此外，在 tbl 表声明时，可以覆盖任何内部的或继承自原型的属性，这不会使运算效率或准确性受到影响。

本技巧在检测 handle 与 yes 等识别器变量时，必须使用“===”运算符。

7.4 识别 new 运算进行的构造器调用

说明：

在一个函数中识别当前调用是使用 new 来进行的构造器调用，还是普通的函数或方法调用。

示例背景：

如果一个函数被设计为既可以用 new 运算来产生对象实例，又可以作为普通函数调用，或者可以作为对象方法调用，那么如何识别当前究竟在哪种环境下被调用呢？

示例代码：

```
175 /**
176  * 1. 识别当前函数是否使用 new 运算来构建实例
177  */
178 function MyObject() {
179   if (this instanceof arguments.callee) {
180     // 是使用 new 运算构建实例
181   }
182   else {
183     // 是普通函数调用
184   }
185 }
```

```
var obj = new MyObject();

/**
 * 2. 识别当前函数是否使用作为方法调用
 */
function foo() {
  if (this === window) {
    // 是普通函数调用
  }
  else {
    // 是方法调用
  }
}
obj.aMethod = foo;
obj.aMethod();
```

示例说明：

当使用 new 运算时，构建过程运行在函数自身的闭包中，而且新构建的对象总是构造器的一个实例。因此，示例 1 使用 arguments.callee 来找到这个构造器函数自身，并检测 this 对象是否为它的实例，从而可以检测是否运行在 new 运算的实例构建过程中。

但这种方法存在一个（唯一的）问题。如果用户代码试图将构造器函数作为它的实例的一个方法，那么示例 1 就会误判。例如：

```
obj.aMethod = MyObject;
obj.aMethod(); // <-- 这个方法调用会被识别为 new()构建过程
```

这时，除非用户代码愿意使用更为复杂的构造过程，在 MyObject()中加入更多的识别代码并与构造器原型等采用某种特别的约定，否则上述的误判问题是不可能解决的。

除了示例 1 所示的方法之外，用户代码也可以用一种简洁的方法来识别 new 运算中的构造器调用：

```
  if (this.constructor === arguments.callee) {
    ...
```

这种方法依赖于构建器原型的 constructor 属性，因此要求 MyObject 的原型或该原型的 constructor 成员未被重写过。这种方法虽然有些限制，但仍然是常常用到的。

示例 2 利用了一条简单的规则：在普通函数调用中，this 引用指向宿主对象——在浏览器环境中是 window，如果是在其他环境中，请参考相关文档。

当用户代码使用 aFunction.apply 或 aFunction.call 来调用函数自身时，可能会显式地指定 this 引用为 null、undefined 等无意义的值。这种情况下，JavaScript 引擎内部会忽略该值，并以宿主对象作为 this 引用传入函数。因此我们不必额外地检测这些值。