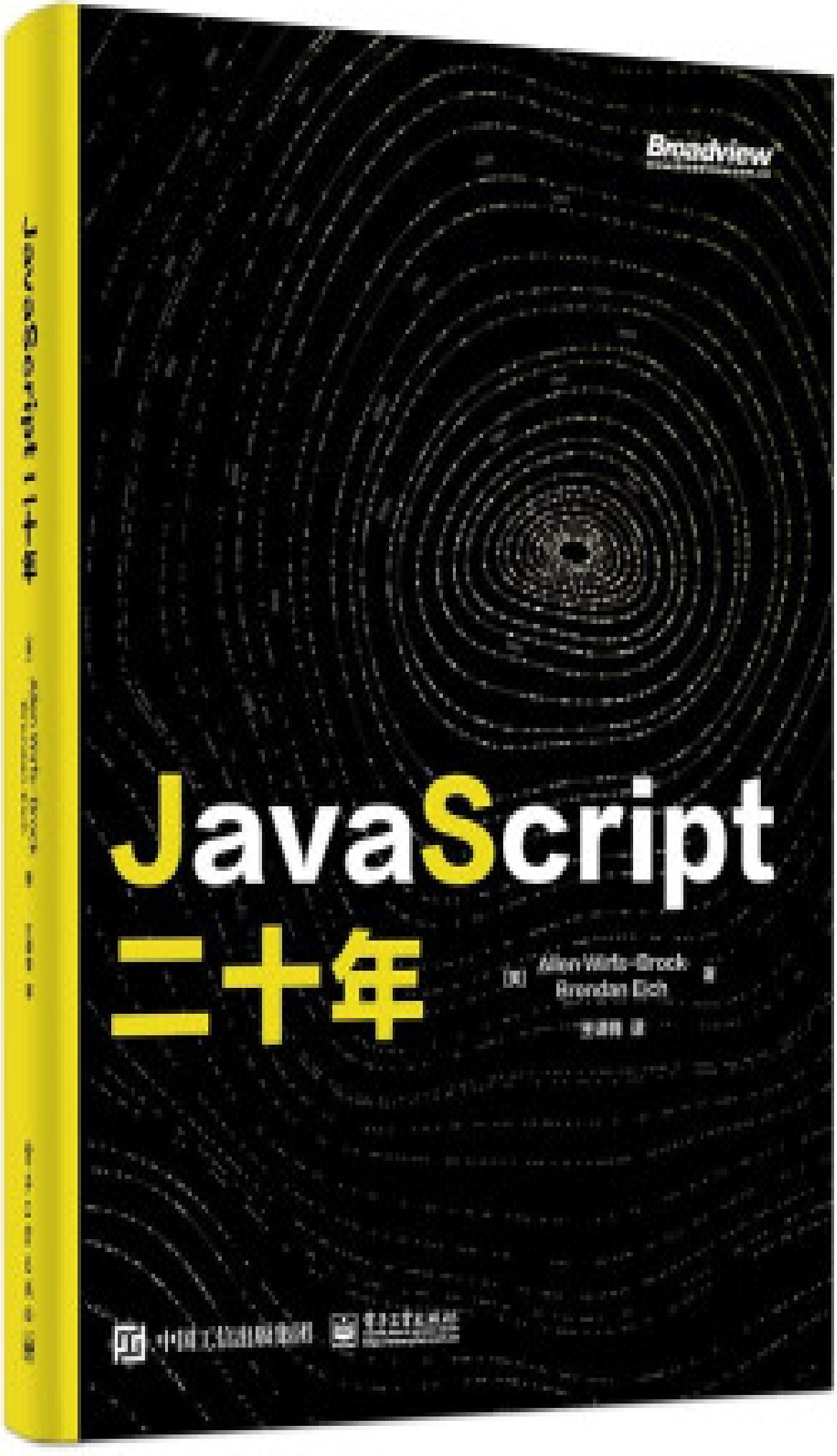
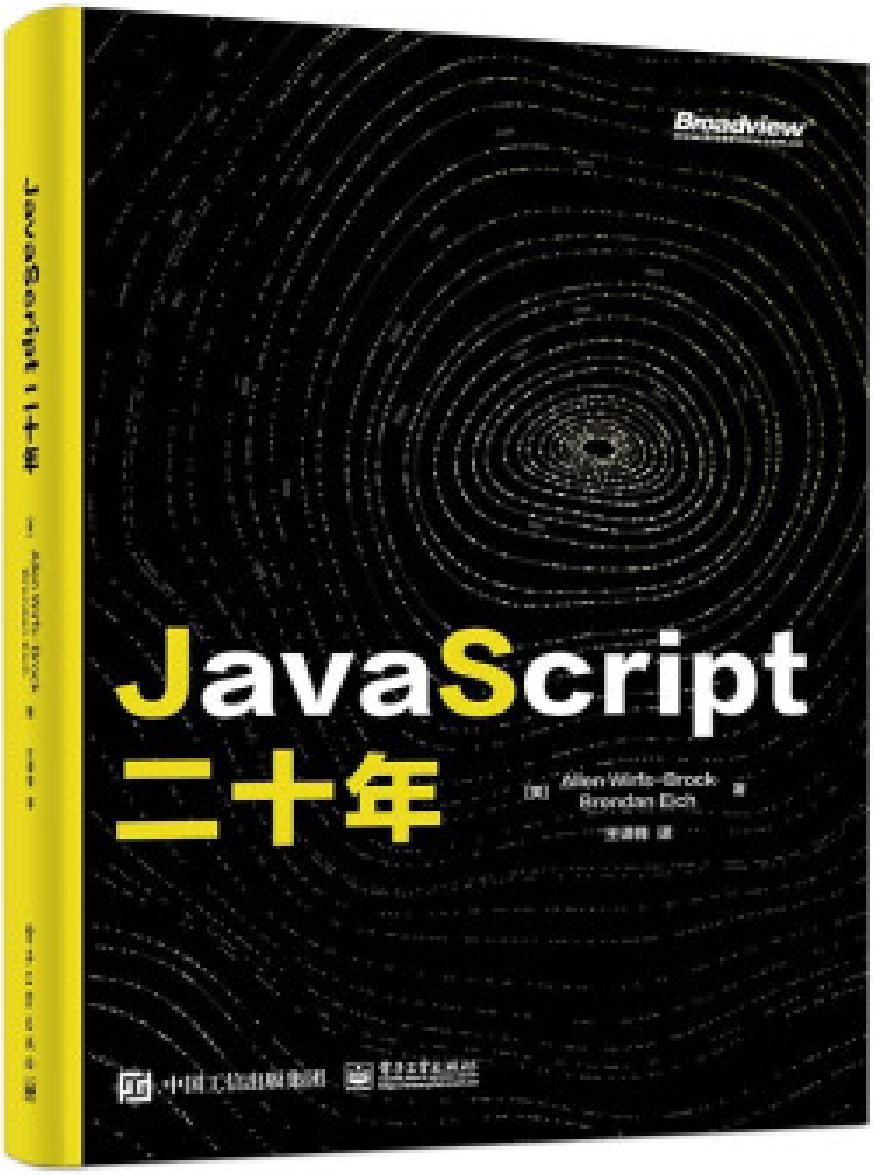

JavaScript
二十年
[美] Allen Wirfs-Brock
Brendan Eich
著
李松峰 译
Broadview
中国工信出版集团
电子工业出版社

# JavaScript 二十年



# JavaScript 二十年§




本项目已与博文视点合作推出纸质版，其版税收入将有 10% 捐献给 Mozilla，点击购买。

## 原文简介§

本书由 JavaScript 之父 Brendan Eich 与 ES6 规范首席作者 Allen Wirfs-Brock 联合编写，详细记载和解读了自 1995 年语言诞生到 2015 年 ES6 规范制定为止，共计 20 年的 JavaScript 语言演化历程。全书不仅讲解了大量语言技术细节层面的演进，更复盘了更高层面上规范制定与标准博弈中的历史成败，是一部讲述人类如何在商业与技术上的竞争合作中促进产业发展的故事。

这个故事相当漫长而复杂。全文分为四个部分，每部分都对应 JavaScript 演化历程中的一个主要阶段。各部分之间还有一段简短的插曲，介绍彼时的开发者们是如何看待与使用 JavaScript 的。

这四个部分依次如下：

1. （The Origins of JavaScript），介绍了 JavaScript 的创建与早期发展，包括语言的诞生背景、命名方式、初始特性及其设计理念等。这一节还追溯了它在 Netscape 与其他公司最初的演化，例如微软的 JScript。
2. 创立标准（Creating a Standard），介绍了从 JavaScript 到 ECMAScript 标准的历程。这主要涵盖 JavaScript 标准化工作的启动、规范的创建、相关贡献者以及决策方式等。
3. 改革失败（Failed Reformations），介绍了在 Eich 离开后，缺乏「仁慈独裁者」的 ECMAScript 委员会修改语言的失败尝试。这主要涉及委员会的分裂、对 ES4 的两轮投入，以及 Flash 与 ActionScript 在其中的渊源等。
4. 继往开来（Modernizing JavaScript），介绍了 2009 年 ES5 与 2015 年 ES6 这两个成功标准背后的故事，主要包括对 ES5 与 ES6 的目标、重大基础性更改与重要新特性的介绍与回顾。

# 目录§

        - 
            - 
            - 
            - 
            - 
            - 
            - 
            - 
                - 
                - 
                - 
                - 
                - 
                - 
                - 
                - 
        - 
        - 
- 
- 创立标准
    - 寻找场地
    - 首次 TC39 会议
    - 编写规范
    - 命名标准
    - ISO 快速通道
    - 定义 ECMAScript 3
- 插曲：JavaScript 不需要 Java
    - 布道师
    - 富互联网应用与 AJAX
    - 浏览器博弈论
- 改革失败
    - 不满于成功
    - 对 ES4 的第一轮尝试
    - 另一条死路
    - Flash 与 ActionScript
    - 对 ES4 的第二轮尝试
        - 重置 TC39-TG1

# 许可§




Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: the first 20 years. Proc. ACM Program. Lang. 4, HOPL, Article 77 (June 2020), 189 pages. DOI:https://doi.org/10.1145/3386327


1. JavaScript 二十年
    1. 原文简介
    2. 目录
    3. 许可

# 语言诞生 · JavaScript 二十年

## JavaScript 二十年

- 

- ## JavaScript 二十年

- 
- 
- 



# 语言诞生§

# 史前时代§

万维网的概念与基础技术，是 1989-1991 年间由 CERN 的 Tim Berners-Lee [2003] 创造的。Web 技术在高能物理圈内流通了几年，但并未在物理社区外引起强烈反响。它真正引发关注的契机，还是 1992-1993 年开发的 *Mosaic*[g]。这款由本科生 Marc Andreessen 和伊利诺伊大学香槟分校超算中心（NCSA）的 Eric Bina 研发的应用，本质上定义了「Web 浏览器」这一全新软件类别。

NCSA Mosaic 是不仅易装易用，而且带有图形界面的 Web 客户端。它在物理学界之外普及了万维网的概念，传播相当广泛。到 1994 年初，商业资本开始通过获得 Mosaic 代码许可或从头研发仿 Mosaic 式浏览器的方式，争相加入这波浏览器浪潮。SGI（硅谷图形公司）的创始人 Jim Clark 拉到了风险投资，并招来了 Marc Andreessen 和 Eric Bina 两人。在 1994 年 4 月，他们共同创立了一家公司。这家公司最终定名为 Netscape（网景通讯），目标是推出世界上最流行的浏览器来替代 Mosaic。为此，Netscape 从零开始研发了下一代 Mosaic 式浏览器 *Netscape Navigator*[g]，它于 1994 年 10 月起开始发行。到 1995 年初，Netscape Navigator 达到了初始目标，正在迅速地取代 Mosaic。

Tim Berners-Lee 的 Web 技术的核心，是使用*声明式的*[g] HTML 标记语言来描述文档，将它们呈现为网页。但业界对于能方便最终用户编排应用操作的*脚本语言*[g] [Ousterhout 1997]，也展示出了相当大的兴趣。这些语言诸如微软 Office 中的 Visual Basic 和苹果 AppleScript [Cook 2007] 之类，其设计目标并非用于实现应用核心的复杂数据结构和算法组件。相反地，它们为用户提供了将此类应用组件「粘合」在一起的新方式。在 Netscape 扩大万维网受众范围的途中，一个重要的问题就是脚本语言「是否应该」与「如何」集成到网页中。

## Brendan Eich 加入网景§

Brendan Eich[4] 于 1985 年在伊利诺伊大学香槟分校硕士毕业，然后立即入职了 SGI 公司，主要从事 Unix 内核和网络层的工作。1992 年，他在离开 SGI 后加盟了 MicroUnity。这是一家资金雄厚的新兴公司，致力于开发视频媒体处理器。在这两家公司，他都实现了用于支持内核与网络编程任务的小型专用语言。在 MicroUnity，他还在 GCC *编译器*[g]上做了些工作。

1995 年初，Brendan Eich 被 Netscape 以「在浏览器里写 Scheme」[5]为诱饵打动而跳槽了。但当 Eich 于 1995 年 4 月 3 日加入 Netscape 时，他发现公司在产品营销与编程语言上的现状都很复杂。Netscape 在 1994 年底拒绝了微软的低价收购要约。此后 Netscape 管理层预计自己将直面微软「拥抱，扩展，灭绝」战略 [Wikipedia 2019] 的攻击。在盖茨的直接领导下，微软已经迅速意识到它们即将推出的封闭生态信息应用 Blackbird 项目 [Anderson 2007]，在跨平台 Web 的兴起之下将无足轻重。因此，盖茨的「互联网浪潮」备忘录 [Gates 1995] 将微软的战略从 Blackbird 重新引导到了 *Internet Explorer*[g] 与一整套服务器产品上，以应对 Netscape 的攻城略地。

网页脚本语言的备选项，包括 Scheme 这样的研究型语言，Perl / Python / Tcl 这样基于 Unix 的实用型语言，以及微软 Visual Basic 这样的专有语言。Brendan Eich 希望的是在浏览器中实现 Scheme。但在 1995 年初，Sun（太阳微系统公司）开始为当时尚未发布的[6] Java 发起了游击营销活动 [Byous 1998]。Sun 和 Netscape 迅速达成协议，决定将 Java 集成到 Netscape 2 中。Eich 回忆说，Marc Andreessen 在 Netscape 会议上的口号是「Netscape 加 Java 干掉 Windows」。在 1995 年 5 月 23 日 Sun 的 Java 发布会上，Netscape 宣布了他们授权 Sun 的 Java 技术 [Netscape 1995a] 在浏览器中使用的意向。

Netscape 内部的这项快速决策，使得对 Scheme / Perl / Python / Tcl / Visual Basic 等脚本语言的选型都受到了严重的阻碍，它们在商业利益和（或）上市时间的角度上看都是不可行的。对 Netscape 和 Sun 的高层，尤其是 Marc Andreessen 和 Sun 的 Bill Joy 来说，他们认为唯一可行的方法是设计实现一门「小语言」[7]来补充 Java。

对这一决策的怀疑者在 Sun 占支配地位，在 Netscape 也占多数。他们质疑是否需要这样一门更简单的脚本语言：Java 是否不适合脚本编

写？如何解释为什么两种语言比一种更好？Netscape 是否具备创建新语言的专业能力？

第一个反对意见很容易反驳。1995 年春季的 Java 并不适合初学者使用，人们必须将 Java 主程序的代码体放在包内部*类*[g]声明下名为 `main` 的静态*方法*[g]中，还必须为所有参数、返回值和变量声明静态*类型*[g]。从 Visual Basic 与 Visual C++ 互补，以及许多 Unix 语言与原生代码组件互补的经验来看，Java 明显对于「胶水」脚本编写者来说还不够简单。

克服第二个反对意见的依据，则是对微软产品的参考。对于专业的 Windows 应用程序员，微软向他们出售 Visual C++。而对于业余爱好者、兼职程序员、设计师、会计师和其他人员，微软提供了 Visual Basic 作为脚本语言。这样那些经验不足的兼职程序员就可以「胶合」定制使用由 Visual C++ 构建的组件了。名为「Visual Basic for Applications」（VBA）的 Visual Basic 版本已经集成到了微软 Office 中，以支持这些应用的用户扩展和脚本需求。

克服了前两个反对意见后，Marc Andreessen 提出了浏览器脚本语言的代号「Mocha」。据 Eich 说，这一提议还希望在适当时候将该语言重命名为「JavaScript」。这种 Java 的辅助语言必须「看起来像 Java」，保持易用性并「基于对象」，而不是像 Java 这样基于类。

只剩下最后一个反对意见了：Netscape 是否具备创建有效脚本语言的专业知识，并应用到 1995 年 9 月的 Netscape 2 beta 上？Brendan Eich 的任务就是通过创建 Mocha 来证明这一点。

## Mocha 的故事§

随着 Java 发布的临近，Brendan Eich 认为时间至关重要。双鸟在林不如一鸟在手，因此他在 1995 年 5 月[8]花了连续十天进行第一个 *Mocha*[g] 实现的原型设计。这项工作赶在了可行性论证的最后期限之前完成。

这个 Demo 包括语言的最小实现，并最小化地集成到了 Netscape 2 浏览器的 pre-alpha 版本中。

Eich 的原型是在 SGI Indy Unix 工作站 [Netfreak 2019] 上开发的，使用了一个手写的词法分析器和递归下降解析器。这个解析器发出的是字节码指令，而不是语法分析树（parse tree）。字节码*解释器*[g]简单而缓慢[9]。

字节码特性源于 Netscape LiveWire 服务器[10]的需求，其开发人员甚至在将 Mocha 原型化之前就希望将其嵌入。这支团队的前 Borland 管理和工程人员都坚信动态脚本语言的未来，但他们希望使用字节码而非源码解析的方式，加快服务器应用的加载速度。

Marc Andreessen 强调，Mocha 应该非常易于使用，任何人都可以直接在 HTML 文档中编写几行。Sun 和 Netscape 的高层管理人员则重申了 Mocha 应该「看起来像 Java」的要求，明确排除了 BASIC 式的东西。但这种 Java 式的外表也带来了对 Java 式行为的期望，这种期望影响了语言*对象*[g]模型的设计，以及原始类型（如 `boolean` / `int` / `double` / `string` 等）的语义。

在外表接近 Java 的要求之外，Brendan Eich 可以自由选择大多数语言设计细节。加入 Netscape 后，他探索了一些「易于使用」与「教育用途」的语言，包括 HyperTalk 语言 [Apple Computer 1988]，Logo 语言 [Papert 1980] 和 Self 语言 [Ungar and Smith 1987]。所有人都认可 Mocha 将会「基于对象」但没有类。因为支持类将花费很长时间，并有与 Java 竞争的风险。出于对 Self 的认可，Eich 选择使用带有单个原型链接的*委托*[g]机制，来创建动态的对象模型。他认为这样可以节省实现成本，但最后还是没有足够时间在 Mocha 原型中暴露该机制。

对象是通过为*构造函数*[g]应用 `new` 运算符的方式创建的。名为 `Object` 的默认对象构造函数，与其他内建对象一起内置在环境中。每个对象由零个或多个属性组成。每个*属性*[g]都有一个名称（也叫*属性键*[g]）和一个值，该值可以是*函数*[g]、对象或其他几种内建数据类型之一。可以通过为未使用的属性键赋值的方式，来创建出新属性。属性没有可见性或赋值限制，构造函数还可以提供一组初始属性。创建对象后，也可以将其他属性添加上去。LiveWire 团队特别喜欢这种非常动态的手法。

尽管 Scheme 的诱惑已经不再，Brendan Eich 仍然发现 Lisp 式的函数一等公民[g]概念很有吸引力。函数一等公民对应的这套工具深受 Scheme 习惯用法的启发，方法不必被包含在类中。这包括支持顶层的子程序、将函数作为参数传递、对象上的方法，以及事件处理器（event handler）。由于时间限制，函数表达式（也叫 *lambda 表达式*[g]，或简称 lambda）被延期，但在语法中得以保留。事件处理器和对象方法通过向 Java（在 C++ 之后）借鉴的 `this` 关键字得以统一。在所有函数中，它都用于表示该函数在作为方法被调用时的上下文对象。

在与 Marc Andreessen 以及一些早期的 Netscape 工程师[11]做非正式讨论的激励之下，这个原型支持了 `eval` 函数。它可以解析执行包含程序的字符串。直觉上，这种动态的「字符串到程序」编程对 Web 浏览器和服务器上的某些应用很重要[12]。不过，支持 `eval` 的决策立刻带来了相应的后果。一些场景需要函数通过类似 Java 的 `toString` 方法，将其源码反编译为字符串。为此 Eich 选择在十天冲刺[13]内实现字节码反编译器，因为不论将源码放在主存储器（RAM 或 ROM）还是从辅助存储器（硬盘等）中恢复，对某些需支持的目标体系结构而言，代价都可能过于昂贵。对于受 Intel 8086 16 位分段内存模型约束的 Windows 3.1 计算机而言，情况尤其如此。因为对于内存中无边界或大型的结构体，需要覆盖并手动管理内存中的多个段。

十天结束时，原型在一次全体 Netscape 工程人员的会议上进行了演示（图 2）。演示获得了成功，这使人们对于交付更加完整且集成度更高的 Netscape 2 感到过分乐观。Netscape 2 的首个 beta 版本计划于当年 9 月发布。Brendan Eich 在那个夏天的主要工作，则是将 Mocha 更全面地集成到浏览器中。这需要设计实现使 Mocha 程序能与网页交互的 API。同时，他还需要将语言的原型实现转变为可交付的软件，并响应早期内部用户的错误报告、更改建议与特性需求。

这个十天创建 Mocha 故事的更多细节，可以参见 Brendan Eich 的复述 [Eich 2008c, 2011d; JavaScript Jabber 2014; Walker 2018]。通过互联网档案馆，还可以获得 Mocha 生产版本的源码 [Netscape 1997b]。Jamie Zawinski [1999] 的「Netscape 宿舍」也描述了在此期间作为 Netscape 软件开发者的工作经历。

# JavaScript 1.0 与 1.1§

Netscape 和 Sun 于 1995 年 12 月 4 日在联合新闻稿 [Netscape and Sun 1995; Appendix F] 中发布了 JavaScript。通稿中 JavaScript 被描述为「一种对象脚本语言」，可用于编写脚本来动态地「修改 Java 对象的属性和行为」。它将作为「Java 的补充，方便进行在线应用开发」。尽管它们的技术设计只有表面上的相似，两家公司还是试图在 Java 和 JavaScript 语言间建立牢固的品牌联系。这种名称上的相似性及其带来的两种语言具备密切联系的暗示，长期以来都是导致混乱的根源之一。

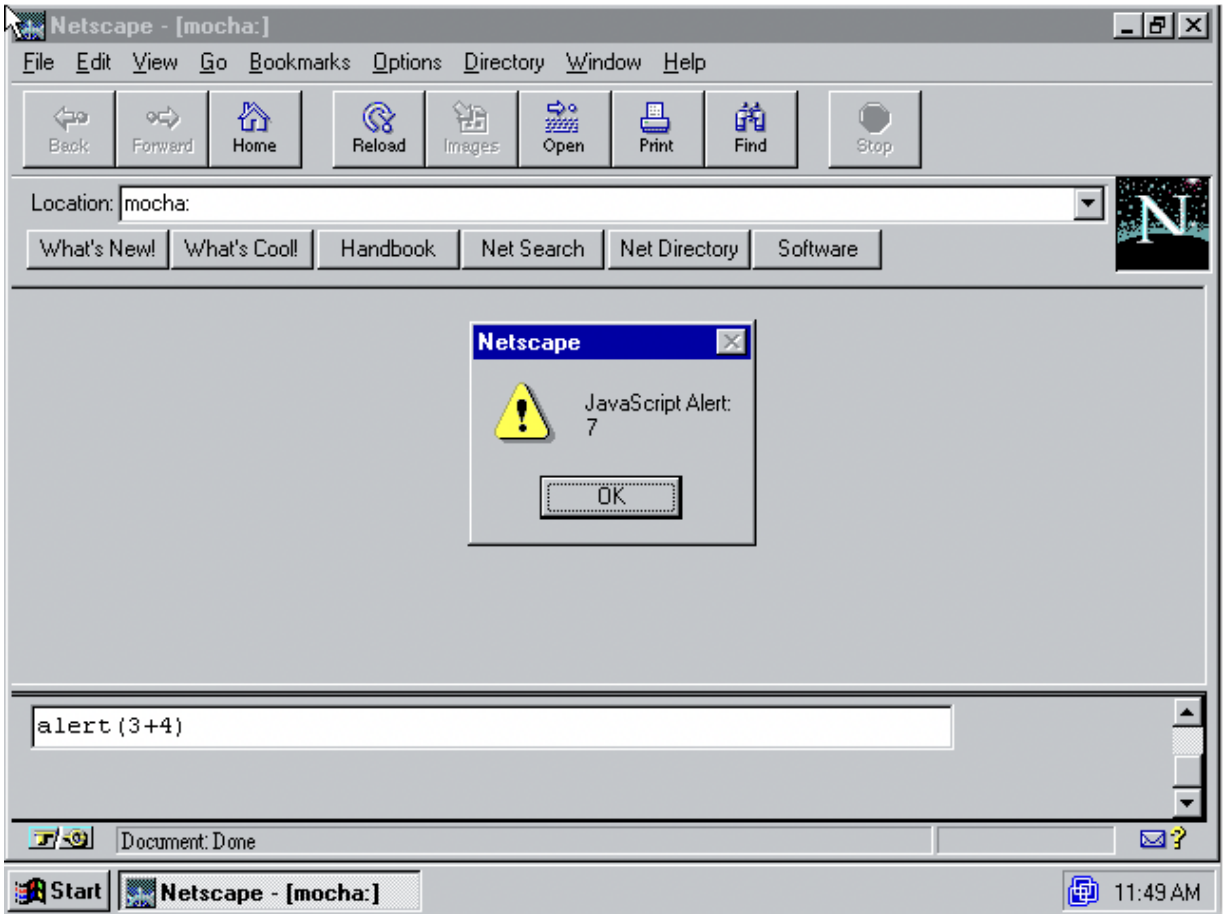



图 2. Mocha 控制台。Brendan Eich 的 Mocha 初始 Demo，其所演示的功能是在 SGI Unix 工作站的 Netscape 2 pre-alpha 中运行的「Mocha 控制台」。这个 Mocha 控制台除了名称改变之外，基本按原样发布在了

Netscape 2 正式版中。这是在 Windows 95 上运行的 Netscape 2.02 的屏幕截图。可以通过在浏览器地址栏中键入 `mocha:` 来激活这个 Mocha 控制台——正式版 Netscape 2 已将其更改为 `javascript:`，但 `mocha:` 仍然有效。激活控制台后，浏览器会打开两个页面框架。在下部文本框中键入的 Mocha 表达式，其求值运行后的效果会体现在上方页面中。这一示例展示了调用内置 `alert` 函数来获得表达式计算值的弹出窗口。原始演示版本的弹出窗口显示的是「Mocha Alert」，而不是「JavaScript Alert」。

以「LiveScript」名称发布的 JavaScript，最初于 1995 年 9 月在 Netscape Navigator 2.0 的第一个 beta 版本 [Netscape 1995b] 中公开。该版本后还有四个 beta 版本，然后才是 1996 年 3 月发布的 Navigator 2.0 正式版。这个正式版支持了 JavaScript 1.0。而 Netscape Enterprise Server 2.0 也在 3 月发布 [Netscape 1996f]，将 JavaScript 1.0 集成到了其 LiveWire 服务端的脚本组件中。

JavaScript 只是 Netscape Navigator 中一个相对较小的功能，因此其开发受到了 Navigator 2.0 整体规划的约束。该计划要求在 1995 年 8 月冻结特性。JavaScript 1.0 的特性集，实际上是划出了当年 8 月 Mocha 实现里正在开发或即将开发的特性。尽管 Eich 在整个 Navigator 2.0 发布历程中都在继续修复最初 Mocha 实现中的 bug，但相对于设想中的语言设计而言，这一特性集并不完整，仍然存在各种疑难 bug 和边界条件下的特殊行为。Brendan Eich 在 1.0 发行前不久接受了采访 [Shah 1996]，他回应了 JavaScript 作为 Java 附属品的官方定位，以及初始发布版本的仓促性：

> BE（Brendan Eich）：我希望它（JavaScript）可以由其他厂商基于我和 Bill Joy 正在起草的规范来实现。我希望看到它保持小巧，但能在 Web 上随处可见，成为把对 HTML 元素的操作与 Java applet 等其他组件粘合在一起时的首选方式。
>
> BE：……据我所知，最常见的用途是使页面更智能，更生动。比如可以根据一天中的时间，在单击链接时加载不同的 *URL*[g]。
>
> ……
>
> BE：隧道的尽头是光明的。现在 JavaScript 的单人秀成分还太重，2.0（Netscape Navigator 版本，译者注）会包含许多烦人的小

bug。我希望所有重大错误都有解决方法，我也已经花了很多时间与开发者一起寻找 bug 及其解法。

我将继续通过修正错误、添加新特性，并尝试使 JavaScript 在所有平台上保持一致的方式，来完成 2.1 版本。我不知道 2.1 版本具体何时交付，但可以保证它会在明年秋天前发布——我们这里前进得很快。

JavaScript 1.0 [Netscape 1996d] 是一种简单的*动态类型*[g]语言，它支持数字、字符串与布尔值、一等公民函数，以及对象数据类型。从语法上看，JavaScript 与 Java 一样属于 C 家族，其控制流语句借鉴了 C，其表达式语法也包括了大多数 C 的数字运算符。JavaScript 1.0 有一个小的内置函数库，其源码通常直接嵌入 HTML 文件中，但其内置库包含一个 `eval` 函数，可以解析并求值编码到字符串中的 JavaScript 源码。整个 JavaScript 1.0 是一门非常精简的语言。图 3 总结了一些缺失的特性。对于现代 JavaScript 程序员而言，这些特性的遗漏可能令人惊讶。

| | |
|---|---|
| 独立的 `Array` 对象类型 | `Array` 字面量 |
| 正则表达式 | 对象字面量 |
| 对 `undefined` 的全局绑定 | `===` 运算符 |
| `typeof`, `void`, `delete` 运算符 | `in`, `instanceof` 运算符 |
| `do-while` 语句 | `switch` 语句 |
| `try-catch-finally` 语句 | `break`/`continue` 到标签 |
| 嵌套函数声明 | 函数表达式 |
| 函数的 `call` 和 `apply` 方法 | 函数的 `prototype` 属性 |
| 基于原型的继承 | 对内置原型对象的访问 |
| 循环垃圾回收[g] | HTML `<script>` 标签的 `src` 属性 |

图 3. JavaScript 1.0 中未涉及的 JavaScript 常用特性（约 2010 年时）。

1996 年初，代号「Atlas」的 Netscape Navigator 3.0 开发工作启动 [Netscape 1996g]，并于 1996 年 8 月发布。Brendan Eich 在此期间得以继续开发那些当 1995 年 8 月的 2.0 版本特性冻结时，还不够完整或缺失的特性。直到 Navigator 3.0 中发布 JavaScript 1.1 [Netscape 1996a, e]

时，JavaScript 的初始定义和开发才算完成。以下各节概述了 JavaScript 1.0/1.1 语言的设计。

## JavaScript 语法§

JavaScript 1.0 的语法直接以 C 语言 [ANSI X3 1989] 为基础，有一些地方受到了 *AWK*[g] 语言 [Aho et al. 1988] 的启发。一个脚本（script）就是一系列的语句（statement）和声明（declaration）。与 C 不同的是，JavaScript 的语句并不限于在函数体内出现。在 JavaScript 1.0 中，脚本源码嵌入在由 `<script></script>` 标签包围的 HTML 文档中。

JavaScript 1.0 中受 C 启发的语句包括：表达式语句；`if` 条件语句；`for` 和 `while` 循环语句；非顺序控制流的 `break`、`continue` 和 `return` 语句；以及语句块（支持使用由 `{}` 分隔的语句序列，就像使用单条语句一样）。`if`，`for` 和 `while` 语句都是复合语句[14]。JavaScript 1.0 并未包含 C 的 `do-while` 语句，`switch` 语句，语句标签与 `goto` 语句。

在基本的 C 语句全家桶基础上，JavaScript 1.0 添加了两个复合语句，用于访问其对象数据类型的属性。受 AWK 启发的 `for-in` 语句可以遍历对象的*属性键*[g]。而在 `with` 语句[15]的语句体内，可以把某个对象的属性名称当作变量来访问。由于属性可能被动态添加（在更高版本的语言中还可以被删除），因此可见变量的*绑定*[g]可能会随 `with` 语句体中的执行过程而发生变化。

JavaScript 中的声明（declaration）并未遵循 C 或 Java 的风格。JavaScript 是动态类型的，没有语言层面的类型名称作为识别声明的语法前缀。相反地，JavaScript 的声明使用关键字作为前缀。JavaScript 1.0 有两种形式的声明，即 `function` 声明和 `var` 声明。`function` 声明[16]的语法是直接从 AWK 借鉴的，定义了单个可调用函数的名称、形参和语句主体。`var` 声明可以引入一个或多个变量绑定，并能选择性地为变量赋值。所有的 `var` 声明都被视为语句，并可在任何语句上下文中出现，包括语句块中。在 JavaScript 1.0/1.1 中，函数声明则只能

在脚本的顶层出现，并且不支持嵌套。`var` 声明也可以出现在函数体内。由这类声明定义的变量，属于函数的局部变量。

与 C 不同的是，JavaScript 1.0 的语句块并未引入声明作用域的概念。在函数体内的语句块中，`var` 声明对这整个函数体均局部可见。位于函数外部块中的 `var` 声明则具备全局*作用域*[8]。如果向作用域内不存在 `function` 或 `var` 声明的变量名赋值，则会隐式创建具有该名称的全局变量。事实证明这种行为是导致错误的重要原因，因为如果拼写错了已声明的变量，也会静默地创建名称错误的新变量。

JavaScript 与传统 C 语法还有一个重要区别，那就是它对语句末尾分号的处理。C 将分号视为强制性的语句终止符，而 JavaScript 则允许在分号是行中最后一个有效字符时，省略这个用于终止语句的分号。这种行为的确切规则并未包含在 JavaScript 1.0 文档中。《Netscape 2.0 手册》在描述各种 JavaScript 语句形式时也并未展示分号，它只说明「一条语句可能跨越多行。如果每条语句之间用分号分隔，则可能在一行上出现多条语句 [Netscape 1996d]」。手册的 JavaScript 代码示例使用了无分号的编码风格，如下所示：

```
var a, x, y
var r = 10
with (Math) {
  a = PI * r * r
  x = r * cos(PI)
  y = r * sin(PI / 2)
}
```

这种不使用分号就可以编写 JavaScript 代码的特性，称为自动分号插入（ASI）。ASI 在 JavaScript 程序员间仍然存在争议。相当一部分程序员仍然更喜欢以无分号风格编码，而其他人则从不使用 ASI。

## 数据类型与表达式§

JavaScript 1.0/1.1 是一种动态类型语言，具有五种基本数据类型：数字、字符串、布尔值、对象和函数。这里的「动态类型」意味着运行时类型信息与每条数据相关联，而不是与诸如变量之类的「值的容器」相关联。运行时类型检查可确保操作仅应用于各操作所支持的数据值上。

布尔值、字符串和数字是不可变（immutable）的值。布尔类型具有两个值，分别为 `true` 和 `false`。字符串值由 8 位字符编码的不可变序列组成，没有 Unicode 支持。数字类型由所有可能的 IEEE 754 [IEEE 2008] 双精度二进制 64 位浮点值组成，不同之处在于仅暴露了一个规范（canonical）的 `NaN` 值。某些运算会特殊处理与「无符号 32 位整数」和「有符号 32 位二进制补码整数」相对应的数字值。Mocha 内部使用了此类整数值的替代表示形式，但只有一个正式的数字数据类型。

JavaScript 1.0 有两个特殊值，用于表示「缺少有用的数据值」。未初始化的变量会被设置为特殊值 *undefined*[17]。这也是程序在尝试访问对象中尚不存在的属性时所返回的值。在 JavaScript 1.0 中，可以通过声明和访问未初始化变量的方式，获取到 *undefined* 这个值。而值 `null` 则旨在表示某个预期存在对象值的上下文里「没有对象」。它是根据 Java 的 `null` 值建模的，有助于将 JavaScript 与 Java 实现的对象进行集成。在整个历史上，同时存在这样两个相似但又有显著不同的值导致了 JavaScript 程序员的困惑，很多人不确定应在何时使用哪个。

JavaScript 1.0 的表达式语法基本上复制自 C，使用了一组相同的运算符（operator）与优先级规则。这里主要省略的部分是 C 的指针和与类型相关的运算符，以及一元的 `+` 运算符。二元的 `+` 运算符被重载，以执行数字加法与字符串连接。移位和按位逻辑运算符可以对有符号的 32 位二进制补码整数进行位级的操作。如有必要，操作数将被截断为整数，并取模减少到 32 位的值。`>>` 运算符可以对 32 位整数值执行符号扩展的算术右移。JavaScript 还添加了从 Java 借鉴的 `>>>` 运算符，用于执行无符号的右移运算。

JavaScript 1.1 添加了 `delete`，`typeof` 和 `void` 运算符。在 JavaScript 1.1 中，`delete` 运算符仅会将其对应的变量或对象属性操作数设为 `null` 值。`typeof` 运算符会返回一个字符串，该字符串标识其操作数的原始类型。可能的字符串值包括
`"undefined"`、`"object"`、`"function"`、`"boolean"`、`"string"`、`"number`

"，或一个由实现环境决定的字符串值，以此来标示宿主对象的种类。令人困惑的是，`typeof null` 会返回字符串值 `"object"` 而不是 `"null"`。其实也可以说这与 Java 保持了一致，因为 Java 的所有值都是对象，而 `null` 本质上是表达「没有对象」的对象。但是，Java 缺少与 `typeof` 运算符等效的特性，并使用 `null` 作为未初始化变量的默认值。根据 Brendan Eich 的回忆，`typeof null` 的值是原始 Mocha 实现中*抽象泄漏*[g]的结果。`null` 的运行时值使用了与对象值相同的内部标记值进行编码，因此 `typeof` 运算符的实现就直接返回了 `"object"`，而无需任何额外的特殊处理。实践表明，这种选择对 JavaScript 程序员带来了很大的麻烦。他们通常想在尝试访问某个值的属性之前，先测试这个值是否确实是一个对象。但光是测试值的类型是否为 `"object"` 并不足以保护属性访问，因为尝试访问 `null` 的属性也会产生运行时错误。

`void` 运算符仅求值其操作数，然后返回 *undefined*。访问 *undefined* 的一种常见手法是 `void 0`。引入 `void` 运算符是为了作为辅助，以便定义那些会在单击时执行 JavaScript 代码的 HTML 超链接。例如：

```
<a href="javascript:void usefulFunction()">
  Click to do something useful
</a>
```

这里 `href` *属性*[g]的值应为一个 URL，而 `javascript:` 是浏览器可识别的特殊 URL 协议。这意味着要对后面的 JavaScript 代码求值，并使用将其转换为字符串的结果，就像使用由常规 `href` URL 获取的响应文档那样。除非获得 *undefined*，否则 `<a>` 元素将尝试继续处理该响应文档。通常 Web 开发者想要的只是在单击链接时对 JavaScript 表达式求值而已。给表达式加上前缀 `void` 即可允许以这种方式使用该表达式，避免 `<a>` 元素的进一步处理。

C 和 JavaScript 表达式之间的最大区别，是 JavaScript 运算符会自动将其操作数隐式转换为运算符领域内的数据类型。JavaScript 1.1 添加了一种可配置的机制，用于将任意对象转换为数字或字符串值。图 4 总结了 JavaScript 1.1 的隐式类型转换（coercion）规则。

| From - To | function | object | number | boolean | string |
|---|---|---|---|---|---|
| **undefined** | error | null | error | false | `"undefined"` |

| From - To | function | object | number | boolean | string |
|---|---|---|---|---|---|
| **function** | N/C | Function object | valueOf/error | valueOf/true | decompile |
| **object (not null)** | Function object | N/C | valueOf/error | valueOf/true | toString/valueOf[1] |
| **object (null)** | error | N/C | 0 | false | `"null"` |
| **number (zero)** | error | null | N/C | false | `"0"` |
| **number (nonzero)** | error | Number | N/C | true | default |
| **number (NaN)** | error | Number | N/C | false[2] | `"NaN"` |
| **number (+Infinity)** | error | Number | N/C | true | `"+Infinity"` |
| **number (-Infinity)** | error | Number | N/C | true | `"-Infinity"` |
| **boolean (false)** | error | Boolean | 0 | N/C | `"false"` |
| **boolean (true)** | error | Boolean | 1 | N/C | `"true"` |
| **string (empty)** | error | String | error[3] | false | N/C |
| **string (non-empty)** | error | String | number/error | true | N/C |

- 若结果以斜杠分隔，表示 JavaScript 会先尝试前者，若未成功则使用后者。
- **N/C** 表示不需转换（No Conversion Necessary）。
- **decompile** 表示一份包含函数独有源码的字符串。
- **toString** 表示调用 toString 方法的结果。
- **valueOf** 表示在 valueOf 方法能为目标类型返回值时，对其进行调用的结果。

- **number** 表示在字符串为有效整数或浮点数字面量时，其相应的数值。
- [1] 如果 valueOf 没有返回字符串，则进行默认的对象到字符串转换。
- [2] 在 Navigator 3.0 所用的 JavaScript 1.1 中，会将 NaN 转换为 true。
- [3] 在 Navigator 3.0 所用的 JavaScript 1.1 中，会将空字符串转换为 0。

图 4. Eich 和 McKinney 在 JavaScript 1.1 初始规范中提出的隐式类型转换规则 [1996, page 23]，最终标准化的规则与此略有不同。这是对原始表格的复制，存在一些排版上的细微差别。脚注 3 并未出现在原文中。

## 对象§

JavaScript 1.0 的对象是关联数组，其元素称为属性。每个属性都有一个字符串键和一个值，该值可以是任何 JavaScript 数据类型。属性可以被动态添加。JavaScript 1.0/1.1 不支持从对象中删除属性。

只要某个属性的键字符串符合标识符的语法规则，就可以用形如 `obj.prop0` 的点符号（dot notation）来访问它。所有属性都可以使用方括号表示法（bracket notation）来访问，包括那些键不符合标识符规则的属性。其中用方括号括起来的表达式将被求值，并转换为用作属性键的字符串。例如当 `n` 的值为 `0` 时，`obj["prop" + n]` 等效于 `obj.prop0`。赋值给不存在的属性会创建一个新属性，访问不存在的属性通常会返回 *undefined*。但是在 JavaScript 1.0/1.1 中，如果使用方括号表示法访问不存在的属性值，并且属性键是非负整数的字符串表示形式，则会返回 `null` 值。

属性既可以用作数据存储，也可以将行为与对象关联。那些值为函数的属性，可以作为对象的方法被调用。而作为对象方法被调用的函

数，则可以通过关键字 this 的动态绑定来访问该对象。

要想创建对象，可以将 new 运算符应用于内置函数或用户自定义的函数。那些意图以这种方式被使用的函数，则称为构造函数（constructor）。构造函数通常会将属性添加到新对象。这些属性既可以是数据，也可以是方法。内置的构造函数 Object 可以用于创建最初没有属性的新对象。图 5 展示了如何使用 Object 构造函数或用户定义的构造函数，来创建新对象。

```
// 使用 Object 构造函数
var p1 = new Object;
p1.x = 0;
p1.y = 0;

// 使用自定义的构造函数
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p2 = new Point(0, 0);
```

图 5. JavaScript 1.0 中创建对象的可选方式。属性既可以在对象被 Object 创建之后添加，也可以通过自定义构造函数在创建对象时添加。

JavaScript 1.0 还有一个内置的 Array 构造函数，但使用 Object 与 Array 构造函数所创建的对象只有一个可见的区别，那就是为该对象显示的调试字符串（形如 "[object Object]" 之类，译者注）。在 JavaScript 1.0 中，Array 构造函数创建的对象没有 length 属性。

通过将整数值作为键来创建属性的方式，可以对任何对象实现类似数组的索引行为。这样的对象还可以带有非整数键对应的属性：

```
var a = new Object; // 或者 new Array
a[0] = "zero";
a[1] = "one";
a[2] = "two";
a.length = 3;
```

JavaScript 1.0 中没有对象*继承*[g]的概念。程序必须分别将所有属性添加到每个新对象上，这通常是通过为程序所使用的每个「类对象」

（class object）定义一个构造函数的方式来实现的。图 6 展示了基于 JavaScript 1.0 定义的简单 Point 抽象。

```
// 定义出作为方法被使用的函数
function ptSum(pt2) {
  return new Point(this.x + pt2.x, this.y + pt2.y);
}
function ptDistance(pt2) {
  return Math.sqrt(Math.pow(pt2.x - this.x, 2) + Math.pow(pt2.y
- this.y, 2));
}

// 定义 Point 构造函数
function Point(x, y) {
  // 创建并初始化新对象的数据属性
  this.x = x;
  this.y = y;

  // 为每个对象实例添加方法
  this.sum = ptSum;
  this.distance = ptDistance;
}
var origin = new Point(0, 0); // 创建 Point 对象
```

图 6. 使用 JavaScript 1.0 定义的 Point 抽象，每个实例对象具备自己的方法属性。

在这个示例中值得注意的重要之处如下：

- 每个方法都必须定义为全局可见的函数。这类函数的名称是必需的，而且其名称不应与用于定义其他「类抽象」（class-like abstraction）方法函数的名称冲突（`ptSum`，`ptDistance`）。
- 构造对象时，必须为每个方法创建一个对象属性，并将其值初始化为相应的全局函数。
- 方法是通过属性名称（`origin.distance`）而非声明的全局名称（`ptDistance`）被调用的。

JavaScript 1.1 不再需要直接在每个新实例上创建方法属性。它通过函数对象名为 `prototype` 的属性，将*原型*[9]对象与构造函数关联起来。《JavaScript 1.1 指南》[Netscape 1996e] 将 `prototype` 描述为「由所有该类型对象共享的属性」。这是个模糊的描述，更好的表述可能是这

样的：原型是一种特殊的对象，其自身属性与所有「由构造函数创建的对象」所共享。

对这种共享机制没有更进一步的说明，但可以发现原型对象具备如下特征：

- 访问对象属性时，如果这个属性的名称在「与对象构造函数相关联的原型」上已被定义，那么将返回原型对象的属性值。
- 对原型对象属性的添加或修改，对于通过「与原型相关联的构造函数」创建的现有对象，是立即可见的。
- 为对象属性赋值时，会*遮盖*[g18]在「与对象构造函数相关联的原型」上定义的同名属性值。

对于语言内置的 `Object.prototype` 对象，其所有属性都可以通过对任何对象的属性访问来获取到，除非该属性已被对象或其原型遮盖。

图 7 展示了 JavaScript 1.1 中对图 6 简单 Point 抽象的定义。

```
// 定义出作为方法被使用的函数
function ptSum(pt2) {
  return new Point(this.x + pt2.x, this.y + pt2.y);
}
function ptDistance(pt2) {
  return Math.sqrt(Math.pow(pt2.x - this.x, 2) + Math.pow(pt2.y
- this.y, 2));
}

// 定义 Point 构造函数
function Point(x, y) {
  // 创建并初始化新对象的数据属性
  this.x = x;
  this.y = y;
}

// 添加方法到共享的原型对象
Point.prototype.sum = ptSum;
Point.prototype.distance = ptDistance;

var origin = new Point(0, 0); // 创建 Point 对象
```

图 7. 使用 JavaScript 1.1 定义的 Point 抽象。实例对象从 `Point.prototype` 对象上继承方法，而不是在每个实例上定义方法属

性。

这里的不同之处在于，方法仅在原型对象上挂载了一次，而不是在构造每个实例对象时重复挂载。由原型对象提供给某个对象的属性称为*继承属性*[g]，而直接在对象上定义的属性则称为*自有属性*[g]。自有属性会遮盖同名的继承属性。

原型对象的属性通常是方法。在这种情况下，构造函数提供的原型发挥的是与 C++ 中的虚函数表（vtable）或 Smalltalk 中的 MethodDictionary 相同的作用，也就是将通用的行为与一组对象相关联。构造函数实际上充当的是类对象（class object）的角色，其原型相当于与类实例共享方法的容器。这是一种对 JavaScript 1.1 对象模型的合理解释，当然也不是唯一的解释。

对构造函数原型属性的命名，清楚地表明 Brendan Eich 考虑了另一种对象模型。该模型的灵感来自于 Self 编程语言 [Ungar and Smith 1987]。在 Self 中，新对象是通过「部分克隆某些种类的原型对象」的方式来创建的。每个克隆体都有一个指回其原型的 `parent` 链接，这样原型就可以提供能在其所有克隆体之间共享的功能了。JavaScript 1.1 的对象模型可以看作是 Self 模型的一种变体。在原型中，原型对象可以通过构造函数被间接访问到，而 `new` 运算符将从原型中克隆出新实例。这些克隆出的实例，会*继承*[g]那些在原型对象属性上通用共享的功能。一些 JavaScript 程序员将此机制称为「原型继承[g]」。这是一种委托机制的形式。一些 JavaScript 程序员还使用带引号的「类式继承[g]」概念，来指代 Java 和许多其他面向对象语言中使用的继承风格。

JavaScript 1.1 的文档 [Netscape 1996e] 并未完全描述这两个对象模型。它维护的是一个与 1995 年 12 月 Netscape / Sun 新闻稿一致的营销故事。JavaScript 被定位为一种用于「脚本式编写对象交互」的语言，而对象抽象的实际定义（类定义）将用 Java 编写。此时原生 JavaScript 的对象抽象能力尚且限于次要特性。这些次要特性仅引起了微小的关注，有很多并未被文档化。

## 函数对象§

在 JavaScript 1.0/1.1 中，函数定义（function definition）会创建并命名一个可调用的函数。JavaScript 函数是一等（first-class）的对象值。在 `function` 声明中提供的名称会被定义为全局变量，类似于顶层代码中的 `var` 声明。而它的值则是函数对象，可以赋值给变量、设置为属性值、在函数调用中作为参数传递，以及作为函数的返回值。因为函数也是对象，所以在它们上面同样可以定义属性。以下示例展示了如何将属性添加到函数对象上：

```
function countedHello() {
  alert("Hello , World!");
  countedHello.callCount++; // 增加该函数的 callCount 属性
}
countedHello.callCount = 0; // 将计数器与函数相关联
for (var i = 0; i < 5; i++) countedHello();
alert(countedHello.callCount); // 显示 5
```

函数需要用形式参数列表（formal parameter list）来声明。但参数列表的大小，并不会限制调用函数时可传递的参数数量。如果调用函数时传递的实参（实际参数，argument）数量少于其声明的形参（形式参数，parameter）数量，那么多余的形参将被设置为 *undefined*。而如果传递的实参数量超过形参数量，则会对额外的实参求值，但无法通过形参名称获得这些值。不过在执行函数体期间，还可以使用类似数组的实参对象（arguments object）作为函数对象 `arguments` 属性的值。调用函数时传递的所有实参，都可以用作 `arguments` 对象的整数键（integer-keyed）属性。这样一来，就可以支持可变长度参数列表的函数了。

## 内置库§

JavaScript 1.0 附带了具备内置函数、对象和构造函数的库（library）。在这个库定义的通用对象[19]和函数之中，有少量属于通用，而有大量则是宿主特定（host-specific）的。在 Netscape Navigator 中，*宿主对象*[g]提供的模型表达了当前 HTML 文档的一部分。这些 API 最终被称为级别 0 的文档对象模型（DOM）[Koch 2003; Netscape

1996b]。而对于 Netscape Enterprise Server，宿主对象支持客户端与服务端之间的通信，管理客户端与服务端之间的会话（session）状态，以及对文件与数据库的访问。这种服务端宿主对象的设计，并没有在 Netscape 服务器产品以外的地方被采用。

JavaScript 的早期设计，很大程度上受到了浏览器平台需求的驱动。在早期 JavaScript 版本对应的 Netscape 文档中，并没有明确区分库中的元素是意图「独立于宿主环境」还是「依赖宿主」。不过，DOM 和其他浏览器平台 API 的设计、演变和标准化，已经足够构成它们自己的重要故事了。本文仅在与 JavaScript 的总体设计相关时，才会提及与浏览器相关的问题。

JavaScript 1.0 仅具有两个通用的对象类，即 `String` 和 `Date`。此外还有一个单例全局对象 `Math`，其属性是常用的数学常量和函数。

在 JavaScript 1.0 程序中，对于某些不活跃或实现得不完整的类，也可以看到它们的构造函数，前提是程序知道该如何访问它们。

JavaScript 1.1 完成了这些特性的实现，并文档化记录了它们的存在。图 8 总结了 JavaScript 1.0 和 1.1 中定义的那些与宿主无关的类。

| 基础对象 | | 属性 | |
|---|---|---|---|
| **1.0** | **1.1** | **1.0** | **1.1 新增** |
| (global functions) | | eval, isNaN[1], parseFloat[2], parseInt[2] | |
| ~~Array~~[3] | Array | | join, reverse, sort, toString |
| ~~Boolean~~[3] | Boolean | | toString |

| 基础对象 | | 属性 | |
|---|---|---|---|
| **1.0** | **1.1** | **1.0** | **1.1 新增** |
| Date | | getDate, getDay, getHours, getMinutes, getMonth, getSeconds, getTime, getTimezoneOffset, getYear, setDate, setHours, setMinutes, setMonth, setSeconds, setTime, setYear, toGMTString, toLocaleString, Date.parse, Date.UTC | toString |
| (function objects) | | arguments, length, caller | |
| ~~Function~~[3] | Function | | prototype, toString |
| Math | | E, LN2, LN10, LOG2E, LOG10E, PI, SQRT1_2, SQRT2, abs, acos, asin, atan, ceil, cos, exp, floor, log, max, min, pow, random[1], round, sin, sqrt, tan | |
| Object | | | constructor, eval, toString, valueOf |

| 基础对象 | | 属性 | |
|---|---|---|---|
| **1.0** | **1.1** | **1.0** | **1.1 新增** |
| ~~Number~~[3] | Number | | toString, Number.NaN, Number.MAX_VALUE, Number.MIN_VALUE, Number.NEGATIVE_INFINITY, Number.POSITIVE_INFINITY |
| (string values) | | length | |
| | String | charAt[4], indexOf, lastIndexOf, ~~split~~[3], substring, toLowerCase, toUpperCase, (plus 13 HTML wrapper methods) | split, toString, valueOf |

- [1] 在 1.0 中仅于 Unix 平台可用。
- [2] 在 1.0 中的行为，视宿主操作系统不同而不同。
- [3] 在 1.0 中存在，但缺乏实用性或 bug 较多。
- [4] 在 1.0 中这些方法是字符串值的属性。在 1.1 中它们是 String.prototype 的属性。

图 8. JavaScript 1.0/1.1 中宿主独立的内置库。

`String` 类提供了 `length` 属性和 6 个对不可变字符串值进行操作的通用方法，它们会在适当的时候返回新的字符串值。JavaScript 1.0 的 `String` 类还包括 13 种方法，用于使用各种 HTML 标签来包装字符串值。这个例子说明了 JavaScript 1.0/1.1 中「与宿主相关的特性」和「通用特性」之间的模糊界限。JavaScript 1.0 没有提供全局 `String` 构造函数，所有字符串值都是使用字符串字面量、运算符或内置函数创建的。JavaScript 1.1 添加了全局 `String` 构造函数和 `split` 方法。

`Date` 类用于表示日历日期和时间。JavaScript 1.0 的 `Date` 是直接按照 Java 1.0 [Gosling et al. 1996] 中的 `java.util.Date` 类而实现的，连 bug 都保持了一致。这里包括了一些编码细节，如使用以 GMT 时间 1970 年 1 月 1 日 00:00:00 为中心的毫秒级分辨率时间值，在外部以 0-11 编

号的月份，以及 Java 设计中存在的 2000 年歧义。这个设计决策的理由，是与 Java 互操作性方面的需求。唯一被排除的 Java 方法是 `equal`，`before` 和 `after`。这里并没有使用它们的必要，因为 JavaScript 具备隐式类型转换（automatic coercion）转换能力，可以将数字关系运算符直接与 Date 对象一起使用。

除了 `Object` 之外，`Date` 是 JavaScript 1.0 中唯一可用的内置构造函数。另外除了类的实例方法之外，`Date` 也是唯一在构造函数对象上暴露方法的类。那些浏览器特定（browser-specific）的类则都没有暴露出构造函数。

对内置库和宿主提供的对象而言，它们的属性具有一些特殊的性质。这些性质是那些由 JavaScript 程序员自定义的属性所不具备的。比如，有的方法属性不会被 `for-in` 语句枚举，而某些属性会被 `delete` 运算符忽略，或具有只读的值。访问或修改某些这样的属性时，会产生具有可见副作用的特殊行为。

JavaScript 1.1 加入了可用的 `Array` 类。由 `Array` 构造函数创建的对象，可以用于表示由整数索引且起点为零的多个异类（heterogeneous）向量。数组元素作为对象属性表示，它们的键是其整数下标的字符串表示形式。数组对象还具有 `length` 属性，这一属性的值由构造函数初始化设置。每当访问大于或等于当前 `length` 值的元素索引时，就会更新 `length` 属性的值。因此，数组对象的元素数量可以动态增长。

## 执行模型§

在 Netscape 2 和后续的浏览器中，HTML 网页都可能包含多个 `<script>` 元素。加载页面后，浏览器将为 HTML 文档创建一个新的 JavaScript 执行环境和全局上下文。全局上下文包括了全局对象，这个对象的属性键涵盖了（由 JavaScript 内置库与宿主环境所提供的）内置函数与变量的名称，以及脚本中定义的全局变量和函数。

在 Netscape 2 中，每个 <script> 元素里的 JavaScript 代码都会按照它们在页面 HTML 文件中的出现顺序，逐个解析和求值。在后来的浏览器中，还可以标记 <script> 元素以支持延迟求值（deferred evaluation）。这使得浏览器可以在等待从网络上请求 JavaScript 代码的同时，继续处理 HTML。但不论在哪种情况下，浏览器一次都只会求值一个脚本。脚本之间通常共享同一个全局对象。由脚本创建的全局变量和函数，对所有后续脚本均可见。每个脚本都会运行到完成（run to completion），而不会被抢占（preëmption）或中断（interruption）。早期浏览器的这一特性已成为 JavaScript 的一条基本原理。脚本是执行的基本单位。每个脚本的执行一旦开始，就会持续到它完成为止。在脚本内部，不必担心其他脚本的并发执行，因为这种情况不会发生。

Netscape 2 还引入了网页框架（Web page frame）的概念[20]。页框（frame）是网页的一个区域，可以在其中载入单独的 HTML 文档。页面上的所有页框都会共享相同的 JavaScript 执行环境，每个页框在这一环境中都具有单独的全局上下文。在不同页框中加载的脚本对应不同的全局对象、不同的内置对象，以及不同的全局变量与函数。不过，全局上下文并没有独立的地址空间。JavaScript 执行环境对应单个用于存储对象的地址空间（address space），这一空间会在环境内的所有页框之间共享。由于所有对象都在同一个地址空间中，对象的引用可能经由不同页框内的 JavaScript 代码互相传递，从而混杂来自不同全局上下文的对象。这可能会导致让人意想不到的行为。图 9 中的 JavaScript 1.1 示例说明了这一点。

```
// 只要在其他页框内求值 new Object()
// 就会让 alien 变量引用到在那里创建的对象
var alien = createNewObjectInADifferentFrame();
var native = new Object(); // 在当前页框创建对象
Object.prototype.sharedProperty = "each frame has distinct
built-ins";
alert(native.sharedProperty); // each frame has distinct built-
ins
alert(alien.sharedProperty); // undefined
```

图 9. JavaScript 1.1 示例，表明即便不同 HTML 页框的内置对象不同，对象也可以互通。

每个页框都有独立的 `Object` 构造函数和 `Object.prototype`。它们所提供的属性，由该构造函数创建的所有对象所继承。向某个页框的 `Object.prototype` 添加属性，不会使该属性对其他页框内由 `Object` 构造函数创建的对象可见。

交互式的 JavaScript 网页是事件驱动的应用。其中的事件循环（event loop）由浏览器实现。HyperCard [Apple Computer 1988] 启发了 Brendan Eich 在最初的 Netscape 2 DOM [Netscape 1996c] 设计中使用事件的概念。最初，事件主要是由用户交互触发的。但在现代浏览器中事件有很多种，其中只有一些是源自用户的。

执行完网页定义的所有脚本后，页面的 JavaScript 环境将保持活跃状态，等待事件发生。事件处理器可以与浏览器提供的对象相关联，这包括了许多 DOM 对象。一个事件处理器也就是一个 JavaScript 函数，能响应事件的发生而被调用。将函数赋值给浏览器对象的某些特定属性，就能使该函数成为与这一属性相关联的事件处理器。例如与可点击的指点设备（鼠标）相对应的对象，就具备可设置的 `onclick` 属性。也可以使用一段 JavaScript 代码，直接在 HTML 元素中定义 JavaScript 事件处理器。例如：

```
<button onclick="doSomethingWhenClicked()">
  Click me
</button>
```

处理完 HTML 元素后，浏览器将创建一个 JavaScript 函数，并将其赋为按钮对象 `onclick` 属性的值。`onclick` 的代码片段会被用作函数体。当被 JavaScript 事件处理器监听的事件发生时，它将被放入未决（pending）事件池中。一旦没有正在执行的 JavaScript 代码，浏览器就会从事件池中获取一个未决事件，并调用与其关联的函数。和脚本一样，事件处理器函数也是运行到完成为止的。

## 迷惑行为与 Bug§

JavaScript 有一些令人感到奇特或意外的特性。它们之中有些是故意为之，有些则是在最初的 Mocha 10 天冲刺期间做出的快速设计决策的产物。JavaScript 1.0 也有 bug 和未完成的半成品特性。

## 冗余声明§

JavaScript 允许作用域内存在多个具有相同名称的声明。函数内部声明的所有同名变量名称，都会对应到同一个变量绑定。这个绑定在整个函数体中都是可见的。例如以下就是个有效的函数定义：

```
function f(x, x) { // x 对应第二个形参，忽略第一个 x
  var x; // 和第二个形参相同的绑定
  for (var x in obj) { // 和第二个形参相同的绑定
    var x = 1, x = 2; // 和第二个形参相同的绑定
  }
  var x = 3; // 和第二个形参相同的绑定
}
```

函数 `f` 中所有的 `var` 声明都会指向相同的变量绑定，也就是函数第二个形参的绑定。在函数的形参列表中，同一名称可以多次出现。在执行函数体之前，由 `var` 声明定义的变量都会初始化为 *undefined*，但名称与形参名相同的 `var` 变量则不在此列。在这种情况下，变量初始值会与「为同名形参传递的实参」相同。`var` 声明的初始化过程（包括冗余声明在内）与「为初始化后的变量赋值」的语义相同。它们在函数体内按正常执行顺序，依次在（初始化阶段）到达时执行。

脚本中可能有多个具有相同名称的 `function` 声明。在发生这种情况时，具有该名称的最后一个函数声明将被提升（hoist）到脚本顶部，并用这个名称初始化全局变量。所有其他同名的 `function` 声明都将被忽略。如果同时存在相同名称的全局 `function` 声明和全局 `var` 声明，它们都会指向相同的变量。在执行流程中遇到初始化器（即字面量）时，所有带初始化器的 `var` 声明都会覆盖函数值。

## 隐式类型转换与 == 运算符§

隐式类型转换旨在降低最初采用 JavaScript 作为简单脚本语言的入门障碍。但随着 JavaScript 逐渐演变为通用语言，事实证明它是导致混淆和编码错误的重要来源，对 == 运算符来说尤其如此。在最初的 10 天冲刺之后，添加到 Mocha 中的一些有问题的转换规则，原本是为了响应 alpha 用户的请求，以简化 JavaScript 同 HTTP / HTML 的集成。例如，Netscape 的内部用户要求使用 == 来比较包含字符串值 `"404"` 的 HTTP 状态码与数字 404。他们还要求在数字上下文中将空字符串自动转换为 `0`，从而为 HTML 表单的空字段提供默认值。这些类型转换规则带来了一些意外，例如 `1 == '1'` 且 `1 == '1.0'`，但 `'1' != '1.0'`。

JavaScript 1.0 还会在 `if` 语句的断言内，将 = 运算符视为 ==。例如：

```
// JavaScript 1.0-1.2
if (a = 0) alert("true"); // 这两条语句是等价的
if (a == 0) alert("true");
```

## 32 位算术§

JavaScript 的按位逻辑运算符，会对编码为 IEEE double 浮点数的 32 位值进行运算。按位运算符首先将整数截断，然后在执行按位运算前为其操作数做模转换，获得 32 位二进制补码值。因此，可以通过表达式 `x|0`，将数字值 `x` 强制转换为 32 位值，其中 `|` 是按位逻辑或运算符。基于这种手法，我们就能按以下步骤执行 32 位的带符号加法：

```
function int32bitAdd(x, y) {
  return ((x | 0) + (y | 0)) | 0; // 将结果 32 位截断的加法
}
```

可以使用类似的模式来执行无符号 32 位算术运算，但这时应使用无符号右移运算符 `>>>0` 来代替 `|0`。

## this 关键字§

每个函数都有一个隐式的 `this` 形参。将函数作为方法调用时，这个参数会被设置为用于访问该方法的对象。这和大多数面向对象语言中的 `this`（或 `self`）含义相同。但是 JavaScript 在「关联到对象的方法」与「独立函数」这两者之间，使用了单一的定义形式。这使 `this` 导致了许多程序员的困惑和 bug。

当直接调用函数而未为其限定（qualify）对象时，`this` 将被隐式设置为全局对象。而全局对象的属性包括了程序的所有全局变量。因此在直接调用函数时，`this` 所限定的属性引用，等价于对全局变量的引用。因为对 `this` 的处理取决于函数的调用方式，所以相同的 `this` 引用在不同的调用场景下，可能具有不同的含义。例如：

```
function setX(value) {
  this.x = value;
}
var obj = new Object;
obj.setX = setX; // 将 setX 作为 obj 的方法

obj.setX(42); // 将 setX 作为方法调用
alert(obj.x); // 显示 42

setX(84); // 直接调用 setX
alert(x); // 获取全局变量 x，显示 84
alert(obj.x); // 显示 42
```

由于某些 HTML 会将 JavaScript 代码段隐式转换成作为方法调用的函数，因此 `this` 引起了进一步的混乱。例如：

```
<button name="B" onclick="alert(this.name + ' clicked')">
  Click me
</button>
```

当执行事件处理器时，它将触发按钮的 onclick 方法。这时 this 指向按钮对象，然后 this.name 会检索其 name 属性的值。

## Arguments 对象§

函数的 arguments 对象与它的形参联系在一起——在 arguments 对象的数字索引属性与函数的形参之间，存在着动态的映射。对 arguments 对象属性的更改，也会更改相应形参的值。并且可以发现对形参的更改，也会对相应的 arguments 对象属性生效：

```
// JavaScript 1.0-1.1
f(1, 2);
function f(argA, argB) {
  alert(argA); // 显示 1
  alert(f.arguments[0]); // 显示 1
  f.arguments[0] = "one";
  alert(argA); // 显示 one
  argB = "two";
  alert(f.arguments[1]); // 显示 two
  alert(f.arguments.argB); // 显示 two
}
```

如以上示例的最后一行所示，还可以将形参名称作为 arguments 对象的属性键，以此来访问形参。

从概念上说，在调用函数时，应该为这次触发的函数创建一个新的 arguments 对象，并将该函数对象 arguments 属性的值设置为这个新 arguments 对象。但在 JavaScript 1.0/1.1 中，函数对象和 arguments 对象是相同的对象：

```
// JavaScript 1.0-1.1
function f(a, b) {
  if (f == f.arguments) alert("f and f.arguments are the same
object");
}
```

```
if (f.arguments == null) alert("but only while a call to f is
active");
```

理想情况下，函数的 arguments 对象只能在其函数体内访问。这是通过在函数调用返回时，自动将函数的 arguments 属性设置为 null 来部分实现的。但假设有两个函数 f1 和 f2，如果 f1 调用 f2，那么 f2 就可以通过对 f1.arguments 求值的方式，访问到 f1 的实参。

arguments 对象还有一个名为 caller 的属性。这个 caller 属性的值是「触发当前函数调用」的函数对象。但如果是最外层的函数调用，这个值则为 null。通过使用 caller 和 arguments，任何函数都可以检查当前调用栈上的函数及其实参，甚至还可以修改调用栈上函数的形参值。还有一个具备相同含义的 caller 属性可以通过函数对象直接访问，而无需通过 arguments 对象。

## 对数值属性键的特殊处理§

在 JavaScript 1.0 中，方括号在与整数键一起使用时具有不寻常的语义。在某些情况下，带方括号的整数键会按照属性的创建顺序，来依次访问对象的属性。如果对象上尚不存在具有该键的属性，并且该整数值 n 小于对象属性的总数，那么就会使用属性顺序来访问对象。在这种情况下，将会访问在该对象上创建的第 n 个属性（起点为零），例如：

```
// JavaScript 1.0
var a = new Object; // 或者 new Array
a[0] = "zero";
a[1] = "one";
a.p1 = "two";

alert(a[2]); // 显示 two
a[2] = "2";
alert(a.p1); // 显示 2
```

JavaScript 1.1 删除了对方括号的这种特殊处理。

## 原始值的属性§

在 JavaScript 1.0 中，数字和布尔值没有属性。并且在尝试访问它们或为其分配属性时，会产生错误消息。字符串值的行为则类似于具有属性的对象，但它们除了只读的 `length` 属性之外，都共享一组相同的属性和值。例如：

```
// JavaScript 1.0
"xyz".prop = 42; // 设置所有字符串的 prop 属性为 42
alert("xyz".prop); // 显示 42
alert("abc".prop); // 显示 42
```

在 JavaScript 1.1 中，对数字、布尔值或字符串值做属性访问或赋值时，会使用内置的 Number / Boolean / String 构造函数隐式创建「包装器对象」（wrapper object）。属性访问是在包装器（wrapper）上执行的，并且通常会从其内置原型来访问继承的属性。通过自动调用 `valueOf` 和 `toString` 方法执行的类型转换，使得在大多数情况下，包装器可以被视为原始值来使用。还可以通过赋值的方式，在包装器对象上创建新属性。但隐式创建的包装器，通常会在赋值后立即不可访问。例如：

```
// JavaScript 1.1
"xyz".prop = 42; // 设置字符串包装器的 prop 属性为 42
alert("xyz".prop); // 隐式创建另一个包装器，显示 undefined
var abc = new String("abc"); // 显式创建一个包装器对象

alert(abc + "xyz"); // 隐式将包装器转为字符串，显示 abcxyz
abc.prop = 42; // 在包装器对象上创建属性
alert(abc.prop); // 显示 42
```

## JavaScript 中的 HTML 注释§

Netscape 1 和 Mosaic 浏览器在遇到 HTML `<script>` 元素时所做的操作，引起了 Netscape 2 中潜在的 JavaScript 互操作性问题。那些较旧但仍被广泛使用的浏览器，在显示网页时会以文本形式显示 `<script>` 正文，亦即实际的 JavaScript 源码。在这些浏览器中，可以用 HTML 注释[21]将脚本主体括起，从而避免出现这种情况。例如：

```
<!-- Mosaic and Netscape 1 -->
<script>
  <!-- 这是包住脚本体的 HTML 注释
    alert("this is a message from JavaScript"); // 对旧浏览器不可见
    // 下一行结束 HTML 注释
  -->
</script>
```

基于这种编码模式，Netscape 1 和 Mosaic 中的 HTML 解析器会将整个脚本主体识别为 HTML 注释，而不去显示它。但按照最初的 Mocha 实现方式，这会使得浏览器无法将脚本解析为 JavaScript，因为 HTML 注释的分隔符（delimiter）在 JavaScript 代码中属于无效语法。为避免该问题，Brendan Eich 使 JavaScript 1.0 支持用 `<!--` 作为单行注释的开始，和 `//` 等效。他没有让 `-->` 成为可识别的 JavaScript 注释分隔符，因为在它前面加上 `//` 即可。这样一来就可以实现脚本的向后兼容支持了，如下所示：

```
<!-- Mosaic, Netscape 1, and Netscape 2 with JavaScript 1.0 -->
<script>
  <!-- 这既是旧浏览器中的 HTML 注释，也是一条 JS 单行注释
  alert("this is a message from JavaScript"); // 对旧浏览器不可见
  // 下一行既结束了 HTML 注释，也是一条 JS 单行注释
  //-->
</script>
```

尽管 `<!--` 注释并未记录为正式的 JavaScript 语法，但 Web 开发者已使用了它们，并且其他浏览器的 JavaScript 实现也支持它。结果 `<!--` 成为了事实上的 *Web Reality*[g]。二十年后的 2015 年，它终于被添加到了 ECMAScript 标准中——笑到最后的总是 Web Reality。

# 微软 JScript[22]§

在 Netscape 和 Sun 公开发布 JavaScript 的同一周，微软宣布它准备令 Visual Basic 成为「用 Visual Basic Script 来创建万维网应用的标准」[Wingfield 1995]。微软在 1996 年 5 月 29 日的 Internet Explorer 3.0 Beta 新闻稿 [Microsoft 1996] 中，正式宣布了对 JavaScript 的支持：

> **ActiveX 脚本**。凭借对 Visual Basic® Script 和 JavaScript 的原生支持，Microsoft Internet Explorer 3.0 提供了最为全面且语言无关（language-independent）的脚本能力。Microsoft Internet Explorer 可以扩展出对其他脚本语言的支持，例如 REXX、CGI 和 PERL。网页设计师可以将任何脚本语言插入 HTML 代码中，创建出将 ActiveX 控件、Java Applet 以及其他软件组件连接在一起的交互式页面。

自 1995 年 10 月 Robert Welland 加入微软 Internet Explorer（IE）团队开始，JScript 的开发工作就启动了。Welland 之前曾为苹果的 Newton 掌上电脑和 NewtonScript 编程语言 [Smith 1995] 工作。NewtonScript 是基于原型的面向对象语言，其设计受 Self 语言的影响。Welland 与 NewtonScript 的首席设计师 Walter Smith 以及该项目的顾问 David Ungar 密切合作，因此 Welland 非常熟悉 Self 和 Ungar 关于基于原型的语言的想法。在离开苹果后，Welland 一直在考虑该如何将脚本添加到浏览器，这也使得他最后被招来实现 Internet Explorer 的脚本能力。

当 Robert Welland 来到微软时，他被告知的是应该将 Visual Basic 放入 IE 中，但当他与微软 DevDiv 开发者工具部门[g]的 Visual Basic 团队讨论时，对方表示这需要花费两年时间。因此他和 Sam McKelvie 快速尝试了使 VBA[23] 在 IE 2 中运行的工作，但发现它太过于复杂而无法与浏览器的对象模型集成。Welland 在 Netscape 2 公开测试版中研究了 LiveScript / JavaScript，并开始尝试实现一个针对 JavaScript 的简单字节码解释器，而后 McKelvie 对其进行了改进。Welland 发现，DevDiv 部门的 Peter Kukol 已经编写了一个 JavaScript 解析器[24]，可以用来生成字节码。于是 Welland 和 McKelvie 将他们的解释器、Kukol 的解析器和 Patrick Dussud 编写的垃圾收集器连接了起来，构成了 JScript 的基础。

微软的 DevDiv 部门负责微软所有编程语言和开发者工具的开发。因此，在 Windows 部门 IE 团队工作的 Robert Welland 和 Sam McKelvie

要想参与新语言实现的开发，在政治上是敏感的。而对于 IE 是否应该支持 JavaScript 的问题，也存在着内部争议。DevDiv 希望集中精力将 Visual Basic 用于脚本，并将 Java 用于应用程序。但 IE 团队的目标是使 IE 3 与 Netscape 3 兼容，这就涉及了对 JavaScript 的支持。微软对于不得不支持 JavaScript 并不满意，但为时已晚，已经无法忽略它了。最后的折衷方案是，IE 和微软整体上将同时支持 JavaScript 和 Visual Basic 用于脚本编写，而脚本语言部分的职责属于 DevDiv。IE 和 Windows 团队的职责，则是把脚本能力集成到浏览器和其他产品中。

1996 年 1 月，Sam McKelvie 转入 DevDiv，而 Robert Welland 留在了 IE 团队。同样在 1 月，Shon Katzenberger 从 Microsoft Word 团队调入 DevDiv，从事脚本研发工作。Katzenberger 接管了解释器的职责，并在 Visual Basic 团队的帮助下，获得了能在同一款解释器上运行的 Visual Basic 脚本化子集。这被称为 Visual Basic Script（VBS）。

Welland 和 McKelvie 将脚本系统打包在一起，覆盖了对 JScript 和 VBS 的支持。这是一个可嵌入的组件，后来被称为 Active Scripting。该组件于 1996 年作为 IE3 和微软 Web 服务器产品 IIS 的一部分而提供。在 IIS 中，它为 Active Server Pages（ASP）提供了服务器端脚本支持。Active Scripting 随后成为了 Microsoft Windows 的标准组件，到 2019 年仍可用于支持旧版应用程序。

IE 团队非常重视与 Netscape 的竞争。他们希望当时作为 Active Scripting 一部分的脚本调试器能够吸引到 JavaScript Web 开发者使用 IE，因为 Netscape 没有 JavaScript 调试器。但他们也了解到，与 Netscape 浏览器的网站互操作性对于推广 IE 至关重要。于是 Shon Katzenberger 和其他人针对数千个使用 JavaScript 的网站运行了 IE 3 的开发版本，并将结果同 Netscape 2 与 Netscape 3 做了比较。每当发现差异时，Katzenberger 都必须对 Netscape JavaScript 的行为做逆向工程，以了解其不同之处。其中有些行为让他们非常吃惊。当他们发现在 Netscape 的实现中 HTML 页框竟然共享一个公共的对象地址空间并可以自由交换对象时，更是尤其让他们震惊。IE 已将页框实现为隔离的环境，因此需要大量的重新设计才能使对象在其中传递。

在整个 JScript 的开发过程中，适当语言规范的缺乏一直是个问题。Welland 回忆说在整个开发历程里，领导 IE3 开发的 Thomas Reardon

会抓住一切机会，就 JavaScript 语言规范的缺失而斥责 Netscape 同行。

## 从 Mocha 到 SpiderMonkey§

在 1995 年全年和 1996 年的大部分时间里，Brendan Eich 都是唯一全职从事 *JavaScript 引擎*[g][25]开发工作的 Netscape 开发者。在 1996 年 8 月发布的 Netscape 3.0 版本中，JavaScript 1.1 仍然主要包含 1995 年 5 月的 10 天原型代码。在发布这个版本后，Eich 认为是时候偿还*引擎*[g]的技术债[26]，并努力使 JavaScript「成为一门更干净的语言」了。但 Netscape 管理层则希望他研究语言规范。他们对微软针对 JavaScript 规范缺失的批评很敏感，并认为即将开始的语言标准化进程需要这样一份规范作为输入。Eich 拒绝了，他想把重新实现 Mocha 作为开始。要想编写规范，他需要的是仔细检查 Mocha 的实现。他认为在检查 Mocha 时重写 Mocha 是最有效率的方法，这也能让他在初始的设计错误被纳入规范前纠正它们。

由于对辩论感到沮丧，Brendan Eich 离开办公室，在家工作了两个星期。在此期间，他重新设计实现了 JavaScript 引擎的核心。此举的收获是一个更快、更可靠和更灵活的运行引擎。他舍弃了将 JavaScript 值表示为*差异联合体*[g]的实践，改为使用包含即时原始值的标记指针（tagged pointer）。他还实现了诸如嵌套函数、函数表达式和 `switch` 语句之类的特性，这些特性从未在原始引擎中实现过。基于引用计数的内存管理器也被替换成了基于标记 / 清除算法的垃圾收集器。

当 Eich 返回办公室时，新引擎已经取代了 Mocha。Chris Houck 这位早期的 Netscape 开发者也参与了进来，成为了 JavaScript 团队的第二位专职成员。Houck 根据电影《Beavis and Butt-Head Do America》[Judge et al. 1996] 中的桥段，将新引擎命名为「*SpiderMonkey*[g]」[27]。Clayton Lewis 加入团队担任经理，并聘请来了 Norris Boyd。技术作家 Rand McKinny 被派来协助 Eich 编写规范。

Brendan Eich 继续将语言增强为 JavaScript 1.2，以使其成为 Netscape 4.0 的一部分。它于 1996 年 12 月发布了第一个 beta 版本，而正则表达式则添加到了 1997 年 4 月的 beta 版本中。各平台上的 Netscape 4 生产版本于 6 月起开始释出，并于 1997 年下半年进行了分发。

SpiderMonkey 所实现的 JavaScript 1.2 语言和内置库，相对于 JavaScript 1.0/1.1 有了显著的增强。图 10 列出了JavaScript 1.2 中主要的新特性 [Netscape 1997c]。

```
* do 语句
* 语句标签，以及 break/continue 到标签
* switch 语句
* 嵌套函数声明（使用词法作用域）
* 函数表达式（lambda 表达式）
* 消除原本由 == 运算符所执行的隐式类型转换
* 可妥善删除属性的 delete 运算符
* 对象字面量
* 数组字面量
* 正则表达式字面量
* 可进行正则表达式匹配的 RegExp 对象
* 所有对象上的 __proto__ 伪属性
* 新数组方法 push, pop, shift, unshift, splice, concat, slice
* 新字符串方法 charCodeAt
* 基于 RegExp 的 fromCharCode (ISO latin-1), match, replace,
search, substr, split
* 函数的 arity 属性
* 将函数及其 arguments 对象拆分为不同对象
* 函数的形参与局部声明，可作为 arguments 对象上的属性名
* arguments.callee
* watch/unwatch 函数
* import/export 语句与脚本签名
```

图 10. JavaScript 1.2 的新特性。

在 JavaScript 1.2 中，大多数新加入的库都来自于其他流行语言现有特性的启发。数组 `concat` 和 `slice` 方法的灵感来自 Python 的序列操作，而 `push` / `pop` / `shift` / `unshift` / `splice` 方法都直接根据同名的 Perl 数组函数建模。Python 还启发了字符串的 `concat` / `slice` / `search` 方法。字符串的 `match` / `replace` / `substr` 来自 Perl。Java 启发了 `charCodeAt` 方法。至于正则表达式的字符串匹配语法和语义，借鉴的则还是 Perl。

JavaScript 1.2 在语句层面所添加的内容，提供了以前熟悉 C 系列语言的程序员所期望的语句。`do` 语句直接复制了 C 语言 `do` 语句的语法和类似的语义，这在 JavaScript 1.0 中遗漏了。带标签的语句以及 `break` / `continue` 到某个标签名的功能，则是直接按照 Java 中的相同特性建模的。它们允许从多级嵌套的循环和 `switch` 语句中尽早脱离（early escape），也可以在非迭代的代码块里这么做。JavaScript 1.2 的 `switch` 语句包含了对 `case` 选择器表达式的编译期求值 [Eich et al. 1998, jsemit.c lines 757-776]，这同 C 与 Java 是一致的。

在 JavaScript 1.0/1.1 中，函数只能定义在脚本顶层的全局声明中。JavaScript 1.2 支持把函数通过局部声明的形式，定义在另一个封闭函数中。这样的内部函数定义可以嵌套到任意层级。内部函数具备词法作用域，它们的局部声明会遮盖外部作用域中具有相同名称的声明。在 JavaScript 1.0/1.1 中，可以对变量和函数做前向引用，因为语言在逻辑上将顶级的 `var` 和 `function` 声明「提升」到了脚本的开头，而函数局部的 `var` 声明也会被「提升」到函数体的开头。类似地在 JavaScript 1.2 中，嵌套的 `function` 声明也会被提升到封闭函数体的开头。如果有多个具有相同名称的 `function` 声明，那么就将封闭函数体源码中最后出现的那个声明与该名称绑定。

JavaScript 1.2 还提供了 lambda 表达式支持，这是通过允许函数定义作为表达式原语的方式来实现的。它们称为「函数表达式」，并在语法上与函数声明相同，只是函数名称变成了可选的。如果存在函数名称，语言则会出于绑定目的，将函数表达式视为提升后的 `function` 声明。不带函数名称的函数表达式则会定义一个匿名函数。不论在哪种情况下，函数表达式的每次运行时求值都会创建一个新的闭包（closure）。新的 `callee` 属性被添加到了 `arguments` 对象上，使得此类闭包可以递归引用自己。

数组字面量和对象字面量[28]都受到了 Python 中类似特性的启发。数组字面量为创建和初始化数组对象的元素提供了简洁的语法，让 JavaScript 程序员可以编写如下内容：

```
// JavaScript 1.2
var p2 = [1, 2, 4, 8, 16, 32, 64];
```

而不必这样：

```
// JavaScript 1.1
var p2 = new Array();
p2[0] = 1;
p2[1] = 2;
p2[2] = 4;
// etc.
```

类似地，对象字面量提供了用于创建对象并将属性与之关联的简洁语法。通过对象字面量，程序员可以编写如下内容：

```
// JavaScript 1.2
var origin = { x: 0, y: 0 };
```

而不必这样：

```
// JavaScript 1.0
var origin = new Object;
origin.x = 0;
origin.y = 0;
```

对象字面量和函数表达式的组合，简化了对包含方法的无类（classless）对象的定义。例如：

```
// JavaScript 1.2
function Point(x, y) {
  return {
    x: x,
    y: y,
    distance: function (another) {
      return Math.sqrt(Math.pow(this.x - another.x, 2)
        + Math.pow(this.y - another.y, 2)
      );
    }
  }
}
var origin = new Point(0, 0);
alert(origin.distance(new Point(5, 5)));
```

将对象字面量和函数表达式的组合，也提供了一种更方便的方法来定义原型对象。另外添加的地方还有 `__proto__` 伪属性（pseudo-property），这个伪属性使 JavaScript 程序能动态访问并修改每个对象（用来访问继承属性）的内部引用[29]。通过使用 `__proto__`，程序可以动态构造任意深度的属性继承层次结构，并动态指定对象该从何处继承属性。

最终，某些 JavaScript 1.2 的更改被证明是错误的。`import` 和 `export` 语句旨在与 Netscape 4 中兼容 Java 的脚本签名机制 [Netscape 1997a] 一起使用。对于签名后的脚本，它们之中定义的全局变量对该脚本是私有的，但使用 `export` 语句可以显式导出其中的函数。非 Netscape 浏览器从未采用过此特性。

尽管用户需求促生了 JavaScript 1.0/1.1 中 == 运算符的隐式类型转换规则，但一些用户仍发现该行为令人惊讶和混乱。Brendan Eich 决定消除 JavaScript 的大多数隐式类型转换，以修复 == [Netscape 1997a; Rein 1997]。如果两个操作数都不是相同的原始类型（数字，字符串，布尔值，对象），那么 == 将返回 `false`。

JavaScript 1.2 希望通过 `<script>` 标签的 `version` 属性，来应对 JavaScript 1.0 和 1.1 的语义更改。但是到 JavaScript 1.2 生产版本发布时，这种形式的版本管理对 Web 开发者来说已变得难以维护 [Rein 1997]，对于需要工作在非 Netscape 浏览器上的网页来说尤其是这样。这些浏览器都维护了自己的 JavaScript 实现。

# 插曲：风评被害§

从诞生之初，JavaScript 一直受到舆论的激烈批评。一些批评针对的是这门语言基本的设计决策，例如动态类型或隐式类型转换等设计细节。其他批评者对于它与 HTML 的集成方式，或对于它暴露浏览器安全漏洞的风险 [Fair 1998]，也存在着巨大的反对意见。Robert Cailliau [Wikinews 2007] 称 JavaScript 为「计算史上最可怕的糟粕」，并说：「我只知道一种比 C 更糟糕的编程语言，那就是 Javascript（原文如此）。」Bert Bos 在 W3C 研讨会上 [2005] 将 JavaScript 描述为「有史以来最糟糕的发明」。

对许多新手程序员而言，浏览器中的 JavaScript 让他们首次接触到了常见的编程问题，例如浮点运算的挑战等。他们通常认为这些问题是 JavaScript 特有的。许多经验丰富的程序员将 JavaScript 与熟悉的编程语言（或由于名称混淆而与 Java）进行比较，并发现 JavaScript 的不

足。介绍 JavaScript 怪癖的文章 [Cardy 2011] 以及相关网站（例如 `wtfjs.com` [Leroux 2010]）一度在 Web 上十分流行。

# 创立标准 · JavaScript 二十年

## JavaScript 二十年

- 

- ## JavaScript 二十年

- 
- 
- 



# 创立标准§

# 寻找场地§

当 1995 年 Mocha 项目开始时，要想确保网页在不同浏览器之间的兼容性，需要的显然已经是规范化的标准了。Netscape 和 Sun 在发布 JavaScript 时 [1995] 也指出了这一点：

> Netscape 和 Sun 计划向万维网联盟（W3C）和互联网工程任务组（IETF）提议，将 JavaScript 作为开放的互联网脚本语言标准。

然而，W3C 和 IETF 都不适合创建独立于厂商的 JavaScript 规范。IETF 关注的重点是互联网的协议和数据格式，而非编程语言。W3C 则是一个新组织，其技术领导者对于向 Web 技术栈中添加命令式编程语言并不感兴趣。譬如 Berners-Lee 的协作者 Robert Cailliau 在一次采访中 [Wikinews 2007] 就这么说过：

> 比如说，我很确信我们需要把一门编程语言内置进去。但是以 Tim（Berners-Lee）为代表的开发者们相当反对，认为它必须保持完全的声明式。

在 1996 年初，浏览器技术正处于「互联网节奏」[30][Iansiti and MacCormack 1997] 的发展速度下。但是，语言的标准化进程常以缓慢而闻名，并且还容易引起争议。鉴于微软正认真对待浏览器竞争，Netscape 和 Sun 担心微软可能企图主导 Web 脚本标准的开发，从而把标准重新聚焦到基于 Visual Basic 的语言上。在 1996 年春天，Netscape 和 Sun 需要找到一个公认的标准开发组织，在它的保护下快速起草 JavaScript 标准。这个过程可以由微软参与，但不能由微软主导。Netscape 的标准专家 Carl Cargill 认识 Ecma 国际组织（Ecma International）的秘书长 Jan van den Beld，并朝这个方向推动了 JavaScript 的标准化。Ecma 对自己的定位是以业务为中心的标准组织，旨在将官僚主义流程最小化，从而把标准制定时间减至最少。由于国际标准组织（ISO）认可 Ecma 国际，Ecma 的标准可以通过快速通道来成为 ISO 标准。除了 Cargill 的人脉以外，Sun 也已经是 Ecma 的会员。它认为 Ecma 在微软反对下仍然坚持发布 Windows API 标准的行为，已经证明了其独立性 [LaMonica 1995]。

在 1996 年春天和夏天，Netscape、Sun 和 Jan van den Beld 做了非正式的联系和讨论。当年 9 月，Ecma 协调委员会（Ecma Co-ordinating[31]）[1996b] 考量了 Netscape 对启动 JavaScript 标准化活动的请求，并授权于 1996 年 11 月 4 日至 5 日在硅谷举行启动会议。Netscape 也正式申请 [Sampath 1996] 成为 Ecma 的准会员（Associate Member[32]）。10 月 30 日，Ecma 发表了对「JavaScript 项目启动会议」的开放邀请 [Ecma International 1996a]。如果获得足够兴趣，它将为活动组织出一个新的 Ecma 技术委员会（Technical Committee）。Ecma 使用数字来标记旗下的技术委员会，而下一个可用数字是 39。在 1996 年 12 月，Ecma 大会在其半年一度的 GA（General Assembly）大会上批准了 TC39 及其工作宣言的创立。与此同时，微软也作为标准会员（Ordinary Member）加入了 Ecma。

## 首次 TC39 会议§

TC39 的组织会议于 1996 年 11 月 21 日至 22 日在加州山景城的 Netscape 办公室举行，根据记载 [TC39 1996] 共有 30 位与会者（图 11）。会议开始前，Netscape 核心技术副总裁 David Stryker 和代表 Ecma 的 Jan van den Beld 分别致了欢迎辞。Stryker 表达了对委员会所创建规范的愿景，希望规范与当前实现只有最小程度的偏差，并能将超出规范范畴的语言扩展留待未来考量。

| | |
|---|---|
| 执行主席 | Mr. J. van den Beld |
| 秘书 | Mr. J. van den Beld (SG ECMA) |
| 参会者 | Mr. Cargill (Netscape), Ms. Converse (Netscape), Mr. Eich (Netscape), Mr. Fisher (NIST), Mr. Gardner (Borland), Mr. Krull (Borland), Mr. Ksar (HP), Mr. Lenkov (HP), Mr. Lie (W3C), Mr. Luu (Mainsoft), Mr. Mathis (Pithecanthropus, JTC1/SC22), Mr. Matzke (Apple), Mr. Murarka (Spyglass), Ms. Nguyen |

| | |
|---|---|
| | (Netscape), Mr. Noorda (Nombas), Mr. Palay (Silicon Graphics), Mr. Reardon (Microsoft), Mr. Robinson (Sun), Mr. Singer (IBM), Mr. Smilonich (Unysis), Mr. Smith (Digital), Mr. Stryker (Netscape), Ms. Thompson (Unisys), Mr. Urquhart (Sun), Mr. Veale (Borland), Mr. Welland (Microsoft), Mr. White (AAC Group, Microsoft), Mr. Willingmyre (GTW Associates, Microsoft), Mr. Wiltamuth (Microsoft). |
| 缺席 | Mr. Huffadine (Callscan) |

图 11. 首次 TC39 会议的参会者 [TC39 1996]。

微软 Internet Explorer 开发团队的负责人 Thomas Reardon 则建议委员会不要将 HTML 对象模型的内置库纳入规范中，从而「避免重复」。这些内容应留给 W3C。这一建议被委员会接受，并对委员会的早期成功至关重要。因为尽管 Netscape 和微软的核心语言特性非常相似，它们的 HTML API 却大不相同。这条 TC39 只开发独立于平台 / 宿主环境标准的决定，一直以来都是 TC39 的核心行动准则之一。Reardon 讨论了微软在尝试使 JScript 与 Netscape 完全兼容时遇到的困难，并强调了制定形式化语言规范的必要性。但他也告诫说，对于能增加价值的竞争性实现，规范同样应当为其留有余地。

会议拟议的内容包括 Netscape、Sun、微软和 Nombas 公司的技术演讲，以及在成立新的 Ecma 技术委员会并开始起草标准语言规范时，实际所需的各类组织活动。但在会议上 Sun 表示它不需要做演讲，因此 Borland International 公司的演讲被添加到了议程中。

在会议的开始阶段，Netscape 和 Borland 都分发了技术规范草案，但微软则没有。在 Thomas Reardon 的发言中，他说微软已制定了自己的初步规范并保存了文件。Reardon 表示他们暂时还没有时间完成拷贝，但明天就会有可用的副本。因此微软的技术演讲移到了会议的第二天。

Brendan Eich 参加了会议，但 Netscape 的技术演讲是由 Anh Nguyen 进行的，介绍了 Eich 和 C. Rand McKinny 为 JavaScript 1.1 编写的《JavaScript 语言规范》的初稿 [1996]。Netscape 向 Ecma 贡献了该文档，作为标准化工作的基础文档之一。Nguyen 解释说，Netscape

Navigator 3 中的 JavaScript 1.1 与 Netscape 2 中的初始 JavaScript 版本有一些差异。Netscape 的规范使用类似于 ANSI C 语言标准 [ANSI X3 1989] 的 BNF 表示法来描述语言语法。它使用非正式的叙述（prose）来定义大多数语义，并使用表格来描述语言的隐式类型转换（coercion）规则。

Borland 研发了JavaScript 和 JavaScript IDE 的服务端实现 [Lazar 1997]，其演讲专注于他们的实现中已经完成或纳入规划的几种语言扩展 [Borland International 1996]。主要的扩展包括类定义、try / catch / finally 异常处理、类 C 的 switch 语句、作为一等值的代码块、数组字面量、类 C 的预处理器，以及许多内置库的新增特性（包括一些 IO 特性在内）。Borland 还指出了他们在尝试与 Netscape 的实现互相兼容时遇到的困难，并表示需要更正式的规范，以确保实现之间的互操作性。

Nombas 的 Brent Noorda 介绍了该公司定位于脚本语言的 Cmm（C minus minus）产品。Cmm 的表层语法和某些语义与 JavaScript 1.0 非常相似，后来，Nombas 将其 Cmm 实现发展成了面向嵌入式应用的 ECMAScript 实现 [Noorda 2012]。

从第一天会议休会起，微软的 Robert Welland 就开始了工作 [Welland et al. 2018, at +8:30]。Thomas Reardon 关于「没有时间做拷贝」的托词其实是种拖延策略，为的是让 Welland 有更多时间来处理微软的规范。为会议创建规范文档的任务之前已分配给了微软的技术作家，但当 Welland 在出差参会之际收到文档时，他发现这份文档甚至不足以作为初步的语言规范，并不想把这样的文档交给委员会。然而当他在会议开始前看到 Netscape 的文档时，他感觉 Netscape 的文档同样写得不充分，不希望它成为制定标准的唯一基础文档。于是 Welland 和 Reardon 决定拖延一天，从而在会议第二天开始时准备出更好的文档。

会议结束后，Robert Welland 回到了他以前做 NewtonScript 时的同事 Walter Smith 的家中。Walter Smith 也供职于微软，但还住在湾区。他们通宵工作，将微软的文档改成了一份过得去的 JavaScript 核心语言初步规范。他们的规范还借鉴了 ANSI C 标准的大部分语法，并用一张表来表达类型转换规则。但是，Welland 希望其余部分的语义也能被更形式化地确定。他想到了 [Welland et al. 2018, at +10:10]《LISP 1.5 程序员手册》[McCarthy and Levin 1965] 中描述 Lisp 解释器语义的

一种风格。在这种风格下，每种句法形式都紧跟着对「如何为该语法求值」的精确描述。某些情况下，语义还会使用伪代码来表示[33]。Welland 决定使用带有编号步骤的类似伪代码，来描述对 JavaScript 语义的求值。

Welland 和 Smith 根据当时的 JScript 实现，在文档中添加了语义。对于不确定的地方，他们会回退参考自己先前在 Self 和 NewtonScript 上的经验，找到从那个角度出发有价值的表述。这份文档包含了一张用于表示数组的对象图，它在对属性继承进行建模时，看起来非常像 Self。到第二天早上，他们觉得做出的文档对于当作起点来说已经足够好了。于是他们制作了副本，由 Welland 在第二天会议开始时进行了分发。这份文档就是《JScript 语言规范 0.1 版》[Welland et al. 1996]，成为了微软贡献到 Ecma 的基础文档。

当 Robert Welland 进行演讲时，他惊喜地发现与会人员普遍更喜欢他的文档，并同意需要更正式的规范来确保实现的可互通性。但这里的共识并非等待另一种尚未确定的正式规范，而是通过整合 Netscape、微软和 Borland 贡献的规范来创建标准的初始草案，然后努力使最终的草案更完整而精确。委员会首先创建了一份问题列表 [Appendix G]，其中的问题需要在第一版标准中得以解决或澄清。鉴于共提交了两份被提议作为基准的文档，所以必须选择其中一份来开始编辑。Netscape 的文档是使用 FrameMaker 编写的，而微软的文档则使用 Word 编写。Ecma 的代表解释说，他们的内部编辑流程使用 Word。于是令 Welland 惊讶的是，委员会同意将微软的贡献作为基准文档。

委员会选举出了最初的主席团成员（图 12），并设定了非常激进的目标：在 1997 年 1 月的下一次会议上准备出初稿，在 1997 年 4 月准备出最终稿，进而在 1997 年 6 月的 Ecma GA 大会上批准该标准。他们安排了后续约每六周一次的会议，并着手建立私人邮件列表和 FTP 服务器[34]。

| | |
|---|---|
| 主席 | **Mr. G. Robinson** (Sun) |
| 副主席 | **Mr. C. Cargill** (Netscape) |
| 副主席 | **Mr. S. Wiltamuth** (Microsoft) |
| 主编 | **Mr. M. Gardner** (Borland) - 待确认 |
| 助理编辑 | **Mr. A. Murarka** (Spyglass) - 待确认 |

图 12. 首次 TC39 会议选举出的主席团成员 [TC39 1996]。

TC39 的第二次会议 [1997e] 于 1997 年 1 月 14 日至 15 日举行，共有 22 位与会者，其中包括 5 位并不属于 Ecma 会员的访客。Jan van den Beld 宣布 TC39 的建立已得到 Ecma GA 大会的确认。他强调说，TC39 必须尽快开始遵守有关会员资格和参与条件的 Ecma 规则，开发 Ecma 标准的贡献者必须是某个 Ecma 会员组织的代表。

会议的主要技术内容，是对标准第一稿进行审查和讨论 [TC39 1997c]。Borland 的 Michael Gardner 和 Randy Solton 通过整合 Netscape、微软和 Borland 的贡献，创建出了这份文档。由于 Spyglass 公司没有加入 Ecma，因此 Anup Murarka 没有参加第一稿的开发。会议将所有三个实现中完全相同的特性认定为不存在争议，并确定了特性不同之处以便后续协调。

那些特定于具体实现的特性，在「扩展提案」（Proposed Extensions）附录中列出。委员会讨论了对扩展的处理方式，并商定对于当时实现中共有的核心特性，其优先级将高于所有扩展。另外委员会还达成一致，认为规范应规避需要修改现有应用的改动，这也最终成为了未来对标准的修订中重要的设计准则。

为满足紧迫的时间表，委员会成立了一个特设技术工作组。这个工作组获得了与编辑一起工作的授权，以便填补丢失的材料，并解决规范中突出的技术问题。小组将进行电子化交流，每周轮流以当面会议和电话会议的形式开会，并由 Scott Wiltamuth 担任书记员。TC39 会议于 1 月 15 日上午 10:30 休会，其余时间用于特设技术工作组的会议。

会议结束后，Borland 决定不加入 Ecma，因此 Michael Gardner 无法继续担任编辑。Sun 请来了 Guy Steele，他从 1997 年 1 月下旬开始担任编辑，一直到 1997 年 9 月发布第一版标准为止。

## 编写规范§

Michael Gardner 和 Randy Solton 在 11 月的会议之后，立即开始了制订第一份规范草案的工作，并在接下来的六周中取得了显著进展。除 Gardner 和 Solton 以外，首份草案的技术贡献者还包括如下：Brendan Eich（Netscape）、C. Rand McKinney（Netscape）、Donna Converse（Netscape）、Shon Katzenberger（微软）和 Robert Welland（微软）。

Robert Welland 返回 Redmond 后，将他的 JScript 0.1 规范交接给了 Shon Katzenberger，以继续开发语言语义 [Welland et at. 2018, at +12:02]。数学博士出身的 Katzenberger 对形式化表示法感到满意。他发现伪代码概念（Appendix P）在描述 JavaScript 语义方面相当有效，其详细程度在他眼里是足以确保互操作性的。Katzenberger 成为了微软对标准开发的主要技术贡献者。他将草稿与现有实现相对照，并为未覆盖到的部分附加编写伪代码算法，从而扩展了 Welland 和 Smith 的深夜工作。然后，他将自己修改后的新材料发送给 Borland 的编辑，以纳入正式草案。2018 年 Katzenberger 在接受采访时 [Welland et at. 2018, at +21:16]，表示他对编辑过程中的改动有时会无意破坏自己的算法而有所不满。当 Guy Steele 可以担任编辑时，他感到相当高兴。

1 月 10 日的草案 [TC39 1997c] 建立了规范的基本结构（图 13），并确定了用于定义语言的许多基础技术、约定和惯用语。在 20 年后的 ECMAScript 标准版本中，这些概念有许多仍在使用。

| **1997 年 1 月 10 日的草案** | **ECMA-262 第 1 版** |
|---|---|
| | Scope |
| | Conformance |
| | Reference |
| | Overview |
| Notational Conventions | Notational Conventions |
| Source Text | Source Text |
| Lexical Conventions | Lexical Conventions |
| Types | Types |
| Type Conversion | Type Conversion |
| Variables | Execution Contexts |
| Expressions | Expressions |

| 1997 年 1 月 10 日的草案 | ECMA-262 第 1 版 |
|---|---|
| Statements | Statements |
| Function Definition | Function Definition |
| Program | Program |
| Native ECMAScript Objects | Native ECMAScript Objects |
| | Errors |

图 13. ECMAScript 标准的结构。

草案中对语法约定的描述，主要来自 Netscape 的规范。但至于表达式与语句级语法的结构，以及产生式（production）的名称，则在很大程度上遵循了微软规范中的用法。在两份贡献出的规范中，表达式语法在细微的细节层面上有所不同，例如函数调用的优先级、对象的创建（new 运算符），以及对象属性访问表达式的元素等。

这份草案试图将自动分号插入（ASI）的规则，精确地定义为用来「校正解析错误」的过程。语句的语法包括了显式的分号，用于终止所有非复合语句。如果没有 ASI，那么缺失分号将会产生解析错误。ASI 规范定义了 JavaScript 解析器何时必须通过「假设存在分号」并重新解析的方式，来尝试纠正此类解析错误。第一版草案中的 ASI 规则并不完整，这在后来的 ECMAScript 规范草案和发行版中进行了完善。

1 月 10 日的草案中包含了 Shon Katzenberger 的伪代码算法（例如图 14），用于定义各种语言结构的语义。具体算法的组成，则包括了带顺序编号的步骤，以及步骤之间的简单条件控制流。每个步骤都包含一些*命令式*[g]的叙述。步骤的叙述用英语编写，并结合了规范中针对常见动作所定义的基本词汇。可以在规范内的其他算法中命名和「调用」这些算法。

```
4.4.7 GetValue(V)

1. If Type(V) is not a Reference, return V.
2. Call GetBase(V)
3. If Result(2) is null, generate a runtime error.
4. Call the [[Get]] method of Result(2), passing GetProperty(V)
for the property name and GetAccess(V) for the access mode.
5. Return Result(4).
```

图 14. 在 2007 年 1 月 10 日的 ECMAScript 规范中 [TC39 1997c, §4.7.4]，一个具名的伪代码算法。原始文档中的步骤 2 末尾少了一个句号。

草案还定义了算法中使用的数据类型。ECMAScript 程序中可见值的类型包括 Number、Boolean、String、Object、Undefined 和 Null。另外还有 Reference、Completion 和 List 类型的值用于定义语言语义，ECMAScript 程序无法直接接触到它们。

对象类型的规范引入了属性*标记*[g]的概念，用于控制如何访问或修改各个属性。规范共定义了七种不同的标记：ReadOnly、ErrorOnWrite、DontEnum、NotImplicit、NotExplicit、Permanent 和 Internal。最后，ErrorOnWrite、NotImplicit 和 NotExplicit 被移除，而 Permanent 则被重命名为 DontDelete。具有 Internal 标记的属性会保留与对象相关联的内部状态，但这对 ECMAScript 程序并不直接可见。这种*内部属性*[g]的用途是保存状态。对于实现对象语义，或者实现内置对象与宿主对象的唯一行为，这些状态都是必需的。

一并引入的概念还包括*内部方法*[g]，这是用于定义对象基本行为的算法。对于某些内部方法，可以用替代性的定义来指定不同种类的对象（例如 Array 对象），从而支持它们在行为上的变化。内部方法的接口，实质上是简单*元对象协议*[g]的规范 [Kiczales et al. 1991]。

在规范中，内部方法和内部属性的名称被括在双括号中，形如 `[[Foo]]`。1 月 10 日的草案定义了内部方法 `[[Get]]`、`[[Put]]`、`[[HasProperty]]`、`[[Construct]]`、`[[Call]]` 和内部属性 `[[Prototype]]`。在第一次形式化表达对象属性访问、原型继承和函数调用的语义时，用到的就是这些内部方法。到 ES1 完成时，又添加了 `[[CanPut]]` 和 `[[Delete]]` 内部方法。

第一稿的目录中既包含了原生（内置）ECMAScript 对象，也包含了由浏览器和 Web 服务端宿主环境提供的对象。但是这些部分在 1 月 10 日的草案中仍然留空。草案中有 20 个条目被明确标记为「问题」，它们是许多附录中描述的潜在语言扩展的补充。

1 月 10 日的草案，是 1997 年 1 月 15 日首次技术工作组会议上讨论的基础。会议做出了一些重要的决定 [Wiltamuth 1997a]，其中包括：

- 初始标准的范畴，将不涉及特定于宿主的库对象与函数。例如那些应由浏览器和 Web 服务端宿主提供的规范。
- 只有在完整的规范草案可用后，才考虑对当前语言的扩展。
- 逗号和 ? 运算符不会传播（propagate）引用值，因此它们既不能在赋值运算的左侧使用，也不能作为函数调用的 `this` 值。
- 标识符中不允许使用非 ASCII Unicode 字符。
- 字符串值支持使用 NUL（U+0000）字符。
- 全局函数和变量声明会创建可枚举、可删除的属性，而规范中定义的内置对象属性则默认为不可枚举但可删除的。

在第一次工作组会议上未解决的问题包括：多次赋值的求值顺序、对继承的只读属性赋值时的语义，以及如何适应 1970 年之前的日期值。

工作组（图 15）在 1997 年 4 月中旬之前定期开会，研究了一系列的主要和次要问题，并审查了编辑编写的工作草案文本。有九次工作会议留下了记录 [Wiltamuth 1997a, b, c, d, e, f, g, h, i]。参加了一些工作组会议的 Richard Gabriel 在个人交流中回忆说，这些会议期间的互动并不罕见。Guy Steele 会询问一些边界条件下特性行为的问题。有时 Brendan Eich 会说「我不知道」，有时 Eich 和 Shon Katzenberger 可能不太确定或产生分歧。在这种时候，他们会在各自的实现中尝试测试用例。如果得到相同的答案，这个答案就会成为被确定下来的行为；如果出现差异，他们将会就问题讨论到达成共识为止。

| | |
|---|---|
| Scott Wiltamuth (note taker) | Microsoft |
| Brendan Eich | Netscape |
| Shon Katzenberger | Microsoft |
| Michael Gardner (1st draft co-editor) | Borland |
| Randy Solton (1st draft co-editor) | Borland |
| Clayton Lewis | Netscape |
| Guy Steele (editor) | Sun |

图 15. ES1 规范工作组的定期参与者。

在第一份 Gardner 和 Solton 起草的规范草案之后，Guy Steele 在 1997 年 2 月 27 日至 5 月 2 日之间，向整个委员会发布了另外七份草案，其余的工作草案则在工作组内分发。除了 Ecma GA 大会的最终草案

[TC39 1997b] 之外，每份草案都包含详细的问题解决日志 [TC39 1997d]。

规范制定过程中的某些问题，对语言的使用产生了长期的影响。比如有个受到持续讨论的问题是这样的：短路布尔运算符 `&&`和 `||` 在遇到可转换为布尔值的操作数时，是应该求值为其中一个操作数的实际值（所谓「Perl 风格」），还是 `true` 或 `false` 的布尔值（所谓「Java 风格」）。Brendan Eich 最初的实现主要使用了「Perl 风格」的语义，但少数情况下也有「Java 风格」的行为。微软和 Borland 则已经实现了完整的「Java 风格」语义。最终决定是一致采用「Perl 风格」。

这个决定直接促成了几年后广泛使用的 JavaScript 惯用法。布尔运算符将 `null` 和 `undefined` 的值转换为 `false`，并将所有的对象引用转换为 `true`。这就带来了如图 16 所示的手法，它为对象属性和可选的函数参数提供了默认值。

```
function f(options) {
  options = options || getDefaultOptionsObject();
  // 如果传递了 options 对象，那么就使用它
  // 否则使用默认的一组配置
  //  ...
}
```

图 16. ECMAScript 1 中为函数形参赋予默认值的手法。

Brendan Eich 回忆说，他希望加入 JavaScript 1.2 中自己对 `==` 运算符语义的更改，以消除其类型转换问题。Shon Katzenberger 成功地说服了他，理由是鉴于会破坏大量现有 Web 页面，现在做这种更改已经为时已晚。Eich 在 JavaScript 1.3 的 SpiderMonkey 版本中恢复了原始的语义。

TC39 的第三次会议是 1997 年 3 月 18 日至 19 日举行的。这是 6 月 Ecma GA 大会前最后一次排定的 TC39 正式会议，目标是让标准的第一版能获得接受和批准。为了满足这份时间表，TC39 需要在这次会议上投票，以将标准提交给 GA 大会。

在 3 月 12 日，标准的 0.12 版本草案 [TC39 1997a] 分发给了全体委员会，并在 3 月 14 日的工作组会议上进行了讨论 [TC39 1997f]。这份草案在技术上已经接近完成，只是 Date 对象的复杂定义仍然只是简单的

一组标题。Shon Katzenberger 提出了关于规范质量的完整提案。经过讨论和审查，这份提案也可以被纳入规范。从 1 月 10 日草案完成的两个月以来，这份文档包含的实质性页面已从 41 页增加到了 96 页。0.12 版草案中除了缺少 Date 规范外，其问题跟踪附录中还有 8 个内部「问题」标签和 6 个重要条目。工作组会议还讨论了大约 12 个需要在规范中解决的其他问题。

由于 Scott Wiltamuth 保证所有问题都不会遗留下争议，并且完整的草案可以在 3 月底完成，因此 TC39 一致同意将草案交给 Ecma GA 大会，以进行 6 月的赞成投票。工作组被赋予的职责是收尾规范，并与 Ecma 秘书处的工作人员一起制定出符合其时间表和格式要求的最终草案。草案的完成比 Wiltamuth 的估计多花了一个月的时间。在 1997 年 5 月 2 日完成最终草案 [TC39 1997b] 前，工作组内部又分发了三份中间草案。最终草案于 5 月 5 日分发给了 GA 大会成员。最终草案符合 Ecma 的文档约定，并包含了 Richard Gabriel 对语言的*非规范性*[g]概述。GA 大会在 1997 年 6 月的会议上同意在稍作编辑更改后，将草案发布为《Ecma 标准 ECMA-262 第 1 版》，并将其提交到了 ISO 快速通道流程中。编辑更改完成后，草案于 1997 年 9 月 10 日分发给了 TC39。《ECMA-262 第 1 版》[Steele 1997] 在 9 月 16 日至 17 日的 TC39 会议上 [1997h] 正式发布。

## 命名标准§

自标准化过程开始起，语言命名上的隐患就已经埋下了。Netscape 起的最初名称「LiveScript」基于它和 Sun 的战略合作伙伴关系而替换成了「JavaScript」。Sun 将「JavaScript」注册成了商标，并将其许可给了 Netscape。尽管 Sun 支持 Netscape 脚本语言的标准化工作，但他们也积极保护与 Java 有关的商标。Sun 似乎不太可能将对「JavaScript」商标的控制权交给标准组织。

在第一次 TC39 会议上，与会者邀请 Sun 提供「JavaScript」名称，并同意在找到更合适的名称之前，先使用「ECMAScript」作为占位名称。Scott Wiltamuth 的任务是收集名称建议并检查其可用性。

Wiltamuth [1997j] 列出了 16 种潜在可行的名称，以及 14 种由于现有商标或用法冲突而被认为不可行的名称。一项民意测验确定了排名最高的候选名称：LiveScript、ScriptJ、EZScript、Xpresso / Expresso / Espresso。会议要求 Netscape 和 Sun 的代表确定 LiveScript 和 JavaScript 的可用性。在此期间，规范草案中继续使用「ECMAScript」。

Sun 确认了 [TC39 1997f] 不会将「JavaScript」许可给 Ecma，而 Netscape 则表示对于使用 LiveScript 作为标准名称，没有法律上的异议[35]。根据这一反馈，TC39 同意与 Netscape 合作以保护 LiveScript 的权利，并且 Ecma 将会审查商标的注册。但在收到 Netscape 的书面确认前，ECMAScript 仍将在规范草案中使用。

提交给 Ecma GA 大会的标准草案，仍然使用 ECMAScript 作为语言名称。在 GA 大会上 [Ecma International 1997]，有人担心在标准标题中使用商标名称的适当性，因为标准的目的是使所有实现该标准的公司享有平等的地位。由于 Netscape 决定不将 LiveScript 正式转让给 Ecma，因此后者无法使用 LiveScript 作为语言的名称。大会批准了带着「ECMAScript」占位名称的标准，并指示 TC39 在 9 月前解决命名问题。

命名问题在 7 月的 TC39 会议上 [1997g] 进行了讨论。Scott Wiltamuth 建议使用「RDScript」[36]，而 Carl Cargill 则建议采用「ECMAScript」作为永久名称。还曾有关于是否需要名称的讨论，因为也许以「ECMA-262」（该规范的 Ecma 文档编号）作为名称就足够了。7 月的会议最后没有解决任何问题。但到了 9 月，TC39 [1997h] 同意使用「ECMAScript」作为语言名称来发布标准。

几个月后，在为是否批准 ECMA-262 为 ISO 标准做决定性投票时，美国国家标准机构（ANSI）评论指出 [TC39 1998e]：「这种语言的任何实现都不太可能被称为 ECMAScript。这在现在和将来都会使用户感到困惑。困惑之处包括标准的含义，以及语言引擎对标准的支持。」事实证明这一预测是正确的。全世界一直在使用「JavaScript」名称来标识这门由浏览器实现的语言，这个名称已经包含在了 HTML `<script>` 元素的规范中。Brendan Eich [2006b] 后来表达了他对命名问题的看法：「ECMAScript 一直是个没人要的商品名，听起来像是种皮肤病。」

# ISO 快速通道§

JavaScript 初始标准化的最后一步，是使 Ecma 规范被接受为国际标准组织（ISO）标准。1997 年 9 月，第一版 ECMA-262 已提交进入 ISO/IEC 快速通道流程 [TC39 1997h]。Guy Steele 随后辞去了项目编辑的职务，由 IBM 的 Mike Cowlishaw 接任。

ISO/IEC 投票产生了来自丹麦、法国、日本、荷兰和美国的国家标准机构的 27 页评论 [TC39 1998e]。这其中还包括了对 TC39 [1998b] 提交的现存错误表的评论。大多数评论指出了快速创建 ECMA-262 时遗漏的次要编辑问题。同时报告的还有一些更重要的技术问题，涉及 Date 对象的 2000 年过渡支持，以及 Unicode 与语言的集成。

在 TC39 的投入下，Mike Cowlishaw 准备了一份《意见处置报告》。这份报告在投票决议会议上获得了审核和接受 [TC39 1998a]。1998 年 7 月，即将出镜的修订版规范发布到了 ISO/IEC，并寄给了各 Ecma 标准会员。后者批准了该修订规范，是为《ECMA-262 第 2 版》[Cowlishaw 1998]。

# 定义 ECMAScript 3§

在第一次 TC39 会议上，涌现出了许多对 JavaScript 1.0/1.1 语言的扩展，其中一些扩展也合并到了语言规范的初稿中。但是 TC39 技术工作组同意优先完成基本语言规范，而后才考虑新特性。因此对第一版来说，大部分可能的扩展都归入了规范草案的附录中 [TC39 1997a, Appendix B]。

到 1997 年 7 月的 TC39 会议 [1997g] 时，第一版的工作已接近完成。委员会考虑的重点转移到了下一版规范中所应包含的新特性。Netscape 已经表明了其 Netscape 4.0 的发展方向，其中会将 SpiderMonkey 引擎与 JavaScript 1.2 的扩展相结合。Scott Wiltamuth 则提出了微软 [1997] 关于「ECMAScript 2.0」的初步提案，其中包括

switch 语句、do while 语句，以及带有标签的 break 和 continue 语句。一并包含的还有 === 和 !== 运算符，以及将 caller 属性添加到 arguments 对象。微软的 Andrew Clinick [1997] 提出了一份单独的提案，希望增加条件编译支持。微软在 10 月将 JScript 3.0 作为 Internet Explorer 4.0 的组件发布时，确定了「第二版」的起点。图 17 列出了截至 1997 年底，由 Netscape [1997c] 和微软 [2009b] 浏览器为 ECMAScript 第一版实现的主要扩展。

| 特性 | JavaScript 1.2 | JScript 3.0 | ECMA-262 第 3 版 |
|---|---|---|---|
| do 语句 | ✓ | ✓ | ✓ |
| break/continue 到标签 | ✓ | ✓ | ✓ |
| switch 语句 | ✓ | ✓ | ✓ |
| 嵌套函数 | ✓ | ✓ | ✓ |
| 函数表达式 | ✓ | ✓ | ✓ |
| 对象字面量 | ✓ | ✓ | ✓ |
| 数组字面量 | ✓ | ✓ | ✓ |
| === 和 !== | | ✓ | ✓ |
| 正则表达式字面量 | ✓ | ✓ | ✓ |
| delete 运算符 | ✓ | ✓ | ✓ |
| 所有对象上的 __proto__ 伪属性 | ✓ | | |
| 数组方法 concat, slice | ✓ | ✓ | ✓ |
| 数组方法 push, pop, shift, splice, unshift | ✓ | | ✓ |
| 带有继承元素的稀疏数组 | ✓ | | ✓ |
| 使用正则表达式的字符串方法 fromCharCode, match, replace, search, substr, split | ✓ | ✓ | ✓ |
| 字符串方法 charCodeAt | ✓ | | ✓ |
| 正则表达式方法 compile, exec, test | ✓ | ✓ | ✓ |
| 正则表达式属性 $1...$9, input | ✓ | ✓ | |

| 特性 | JavaScript 1.2 | JScript 3.0 | ECMA-262 第 3 版 |
|---|---|---|---|
| 正则表达式全局属性 `lastMatch`, `lastParen`, `leftContext`, `rightContext` | ✓ | | |
| 带有本地声明属性的 `arguments` 对象 | ✓ | ✓ | |
| `arguments.callee` | ✓ | | ✓ |
| `arguments.caller` | ✓ | ✓ | |
| `watch/unwatch` 函数 | ✓ | | |
| `import/export` 语句与脚本签名 | ✓ | | |
| 条件编译 | | ✓ | |
| `debugger` 关键字 | | ✓ | |

图 17. 主流浏览器在 1997 年对 ECMA-262 第一版的扩展。它们中的多数最终包含在了 ECMA-262 第三版中。

TC39 的正式会议已经改由代表成员公司的小组与项目经理参加，转为了管理和战略会议。而整个委员会的大部分技术工作，都发生在非正式技术工作组中。在 7 月的会议上，TC39 商定了开发第二版的一系列步骤。委员会还达成了共识，认为技术工作组有责任定义工作项目、特性提案和验收标准。第二版分配到的时间要比第一版更多，以使草案进一步成熟并获得外部反馈。第二版规范初稿的目标日期是 1997 年 12 月。在 9 月的会议上 [TC39 1997h]，人们还同意第二版规范必须向后兼容那些符合第一版规范的程序。

在做出这些决定时，ISO 快速通道流程尚未开始。这时还没有人知道，由此产生的更改将需要发布新版 ECMA-262 标准，才能与 ISO 版本保持一致。在 1998 年初，一度有两个成员互相重叠的工作组，分别负责两份单独的规范草案。显然，这里的「第二版」（提交给 ISO 的 Edition 2）和「第二版」（包含新特性的 Version 2）已经不大可能合并。但是尽管 TC39 代表们已经知道这个版本可能会发布为「第三版」，他们还是继续将下一轮功能性工作叫做「第二版」或「V2」。

像这样 TC39 的内部版本命名与最终的发布术语相冲突的情况，后面还会发生。

到 1997 年底，技术工作组的参与者发生了重大变化。图 18 列出了 1998 年间在工作组会议记录中出现的个人。在开发第一版的工作组常规参与者中，只有 Clayton Lewis 仍然保持活跃。Brendan Eich 在 1998 年 2 月参加了一次会议，而后成为了 Mozilla 项目 [Mozilla Organization 1998] 的联合创始人，致力于开源 Netscape 浏览器的代码，由 Waldemar Horwat 接任 TC39 的 Netscape 语言设计负责人。无独有偶，微软的 Katzenberger 也在休假后转入其他项目，Herman Venter 和 Rok Yu 接替了他代表微软在 TC39 的职责。

| | | | |
|---|---|---|---|
| Norris Boyd | Netscape | Drew Lytle | Microsoft |
| Andrew Clinick | Microsoft | Karl Matzke | SunSoft |
| Mike Cowlishaw | IBM | Mick McCabe | Netscape |
| Jeff Dyer | Nombas | Dave Ragget | HP/W3C |
| Bill Gibbons | Netscape | Herman Venter | Microsoft |
| Waldemar Horwat | Netscape | Rok Yu | Microsoft |
| Mike Ksar | HP | Chris Weight | Microsoft |
| Clayton Lewis | Netscape | | |

图 18. 1998 年 TC39 技术工作组的经常性参与者。

在 1997 年 10 月，技术工作组为能包含在第二版中的特性列出了清单（Appendix H）。在这里获得认可而列出的特性中，除了一些例外，主要都是 Netscape JavaScript 1.2 和微软 JScript 3.0 特性的结合。还有 `toSource` 也包括在内，对应于 Brendan Eich 为 JavaScript 1.3 开发的对象序列化与持久性方案[37]。其他已在设想中但缺乏共识的特性则另外列出。与第一版一样，工作组的大部分注意力都集中在精确指定已实现的特性，并解决实现之间存在的差异。但是，商定的特性列表里还包括异常处理机制、`instanceof` 运算符，以及尚未实现的所有其他特性。开发这些特性将需要某种设计工作，这在第一版中是不必的。图 19 列出了一些 1998 年前的浏览器所没有的特性，这些特性最终都包含在了 ES3 中。

```
* try-catch-finally 和异常对象
* instanceof 和 in 运算符
```

```
* 对象原型方法：hasInstance, hasOwnProperty, isPrototypeOf,
propertyIsEnumerable
* undefined 的全局绑定
* toFixed, toExponential, toPrecision
* URI 处理器函数
* 标识符中的 Unicode 字符
* 基础的 I18N 方法：Object toLocaleString; Array toLocaleString;
Number toLocaleString; String localeCompare, toLocaleLowerCase,
toLocaleUpperCase; Date toLocaleDateString, toLocaleTimeString
```

图 19. 1998 年前的浏览器所缺乏的 ES3 新特性。它们中的一些在 TC39 开发 ES3 时就集成到了浏览器里。

技术工作组按每月面对面开会的节奏设定了规划。Mike Cowlishaw [1999b; Appendix I] 维护了一份文档，以跟踪规范各部分的当前状态。状态指示器如下：「自 V1 起未更改」、「尚未准备就绪」、「需要讨论」，「特性已接受」和「内容已达成共识」。状态「特性已接受」表示委员会对规范中定义的功能性表示同意，状态「内容已同意」则表示实际的规范文本已经过审核而被接受。

Bill Gibbons 是新规范工作草案的编辑。每次会议都有一个介绍和讨论各种提案和未解决问题的议程。提案被提出的形式，则通常是提交新的或修订后的算法规范文本。会议还进行了一般状态审核，由与会人员讨论自上次会议以来确定的问题。当就提案或问题解决方案达成协议时，Gibbons 会将其纳入工作草案。V2 版本的第一份完整草案 [Cowlishaw et al. 1998] 发布于 1998 年 4 月，基于 ECMA-262 第一版，其中没有包含任何为 ECMA-262 第二版（ISO 版本）同时开发的更改。工作草案的标题页指出，这里包含的是 Netscape 和微软提交的拟议更改。在 9 月 ISO 版本完成后，Gibbons 将 ES2 更改合并到了当前的 V2 工作草案中。

当时 Unicode 仍然是一种新技术，语言设计人员还在探索将其集成到编程语言中的最佳实践。有个需要特别关注的问题，即如何处理 Unicode 的各种正规化（normalization）形式，这些形式允许对行为等效的字符序列进行替代编码。ES1 对 Unicode 的支持很少。惠普的 Tom McFarland 参加 1998 年 5 月的会议后提交了一份备忘录 [McFarland 1998]，指出了他认为与*国际化*[g]（I18N）有关的许多问题，以及如何将 Unicode 更好地集成到 ECMAScript 中。经过几次会议的讨论，TC39 在 1998 年 11 月建立了一个由 IBM 的 Richard Gillam

[1998] 主持的「I18N 工作组」。I18N 小组很快决定将重点放在针对核心语言的少量基本 I18N 特性上 [Gillam et al. 1999b]，并将关于国际化和本地化更复杂的内容推迟，将它们纳入单独定义的可选库中 [Gillam et al. 1999a, b]。但直到 2012 年，这些类库的规范 [Lindenberg 2012] 才得以完成。除了为核心语言添加了少量区域特定（locale-specific）特性外，I18N 小组还解决了如何将非拉丁字符合并到标识符中的问题。它推荐 ECMAScript 规范对于提供给实现的源代码，可以假定其均采用 Unicode 的正规形式 C（Normal Form C）来编写。这在很大程度上避免了正规化问题。它还选择不对核心语言中的 Unicode 正规化提供任何支持，并把对正规化的编程支持推迟纳入可选库中。

V2 的主要任务，是为语言设计异常处理机制。1998 年 2 月 [TC39 1998c]，微软的 Herman Venter 和 Netscape 的 Waldemar Horwat 均提出了设计草案。两种设计都多少参考了 Java 的 try-catch-finally 语句语法，但它们和 Java 在语法和语义上都存在着显著的差异。

在微软的设计 [Venter 1998b] 中，任何值都可以作为异常抛出，并且 `try` 语句具有单个 `catch` 子句，它声明了一个初始化为「被捕获的异常值」的局部变量。从 `try` 块传播的所有异常都会被无条件捕获，没有 `finally`。

Netscape 的设计 [Horwat 1998] 还允许将任何值作为异常抛出。但在这种设计中，`try` 语句可能具有多个 `catch` 子句[38]，其中带有将 `instanceof` 用作鉴别符（discriminator）的语法，以确定要执行哪个 `catch` 的子句。如果没有 `catch` 子句与异常匹配，那么在执行 `finally` 子句后，还会继续在调用栈中传播异常。`instanceof` 鉴别符最终被 `if` 鉴别符[39]所取代，它会将表达式求值为布尔值，以确定是否选中了想要的 `catch`。

在 1998 年 2 月的会议上 [TC39 1998d]，委员会同意使用 `try` 和 `catch` 关键字，并且 `throw` 语句可以传播任何值（不仅是特定内置异常类的实例）来表示异常。在 1998 年 3 月的工作组会议上，Waldemar Horwat 主张加入 `finally` 子句，并同意进一步研究相应实现的细节。4 月的工作草案 [Cowlishaw et al. 1998] 合并了 Netscape 的设计，但当时尚未解决的问题包括：对 `finally` 的支持、`catch` 变量绑定的作用域、是否允许多个 `catch` 子句、是否应该将 `instanceof` 用作 `catch` 的选择器，以及是否应自动重新抛出未被选中的异常。图 20 提供了一些

示例，展示了微软的提案、Netscape 修改后的提案，以及最终在 ES3 中确定的语法。注意 Netscape 的设计使用了单独的选择器表达式来选择 `catch` 子句。但在微软和最终的 ES3 设计中，则需要使用单个 `catch` 块中的用户逻辑来区分不同的异常。

```
// 微软的设计
try {
  doSomething();
} catch (var e) {
  if (e == "thing")
    console.log("a thing")
  else if (e == 42)
    console.log("42")
  else {
    console.log(e);
    cleanup();
    throw e; // 重新 throw
  }
// 没有 finally 语法
}
cleanup();


// Netscape 的设计
try {
  doSomething();
} catch (e if e == "thing") {
  console.log("a thing")
} catch (e2 if e2 == 42) {
  console.log("42")
} catch (e3) {
  console.log(e3);
  throw e3; // 重新 throw
} finally {
  cleanup();
}


// 第 3 版规范的最终设计
try {
  doSomething();
} catch (e) {
  if (e == "thing")
    console.log("a thing")
  else if (e == 42)
    console.log("42")
```

```
  else {
    console.log(e);
    throw e; // 重新 throw
  }
} finally {
  cleanup();
}
```

图 20. 异常处理的几种设计。在这些示例中，`doSomething` 函数可能抛出两种异常，它们在当前函数继续执行前都需要单独处理。所有其他异常都被「重新抛出」以传播给当前函数的 `caller`。当前函数还具备 `cleanup` 流程，不管 `doSomething` 是否抛出异常都会执行。

直到 1999 年 9 月对标准草案进行最终技术审查 [TC39 1999b] 前，语言是否应支持多个 `catch` 子句的问题一直没有得到解决。这个特性最终推迟留待未来考虑。同样在最后的审查中，委员会才就标准将定义的内置异常类达成了共识。

在将 Java 和其他*静态类型*[8]基于类的语言中的特性，适配到使用动态类型和原型继承的 JavaScript 时，委员会遇到了一些困难。像 `catch` 子句的守卫表达式（guard expression）就是这其中的一个例子。在 Java 中，要由哪个 `catch` 子句处理抛出的异常，是通过无副作用的「子类型包含测试」来确定的。这种测试完全依赖静态声明的类层次结构，可以在实际恢复调用栈现场（call stack unwinding）之前执行。但是 JavaScript 则既没有正式的类概念，也没有静态的类层次结构。由于委员会已经决定支持抛出任何类型的值作为异常，故而要想在 JavaScript 的 `catch` 子句中区分出任意的值，就需要求值任意的守卫表达式，这其中可能包含赋值和函数调用。但是，对表达式的求值需要建立适当的词法和动态环境，并且每次对守卫表达式的求值都可能产生副作用，这些副作用可能会改变后续守卫表达式的求值结果。在一份中立提案中，Waldemar Horwat [1998] 提出了一种复杂的叙述性规范，它允许实现者决定「何时」以及「以何种顺序」来对 `catch` 到的守卫表达式求值，甚至还允许多次对单个守卫表达式求值。Horwat 希望使调试器在恢复现场前，能够确定是否还有「未被处理的抛出异常」。幸好这个设计未被接受，因为随后的经验表明，这种实现方式上的差异，是网页在兼容多个浏览器时互操作性问题的重要来源。

另一个 TC39 难以将语言的概念和构造从 Java 转换为 JavaScript 的例子，则是 `instanceof` 运算符。在 Java 中，`instanceof` 是一个二元运

算符，用于测试其左操作数的对象是否为右操作数的「类实例」或「子类实例」。Herman Venter [1998a] 最初提出的 `instanceof` 提案限制了右操作数仅限标识符，这样就完全模仿了 Java 的语法。但是 JavaScript 本质上没有类的概念，并且还有多种创建新对象的方法。Venter 的提案假定使用构造函数模式作为测试 `instanceof` 的基础。这样一来，右操作数就可以动态地求值到构造函数对象，而这是个一等的函数值。由于这样的右操作数是一等的值而非类型引用，因此提案不久就泛化支持了在该位置上出现表达式。`instanceof` 的运行时语义被定义为：遍历左操作数的原型继承链，搜索值为右操作数 `prototype` 属性当前值的对象。对于许多简单的构造函数，这将会匹配到那些将 `new` 运算符应用到它们上面而创建的对象。

具备 Java 背景的新 JavaScript 程序员会认为 `instanceof` 是区分各种对象的可靠方法，但许多经验丰富的 JavaScript 程序员会都避免使用它。这是因为构造函数返回的对象未必能通过动态的 `instanceof` 测试，并且由于对象元结构的可变性，对 `instanceof` 的重复应用可能不是幂等的。如果要测试的对象来自与构造函数不同的 HTML 页框，测试也可能失败。最后，即使结果为真，被测试的对象仍然可能没有由构造函数创建的数据和行为属性。

ES3 包含了内部函数声明和函数表达式，它们与 JavaScript 1.2 中最初引入的概念相似。函数声明被明确排除在 `{}` 语句块之外，也不能作为子语句使用。Waldemar Horwat [2008b] 后来解释了原因：

1. 将这类声明提升到最高层级（像 `var` 那样）的做法是无效的。因为在这样的函数能捕获的作用域里，可以包含尚不存在的变量。ES3 没有局部作用域，但确实有会导致相同问题的异常作用域。当我们考虑将语言扩展为支持常量和动态（即运行时）类型注释后的场景时，情况还会变得更糟——这样的函数可以捕获尚未创建的常量，甚至还可以捕获尚未计算出类型的变量！

2. 可以选择等到遇到此类声明时再绑定它们，这样也确实可行。但我们不想仅出于对函数的支持，就在 ES3 中实现这样的本地绑定。

3. 在这类声明位于 `if` 语句的子语句位置时，规划中的设想是仅在 `if` 表达式为真（对 `else` 子句为假）时创建这些声明，

> 并将其放入最接近的封闭块级作用域内。这就构成了某种形式的条件编译。而一个语句块如果前面有标记（attribute），那它就是一个非作用域块，这个块会把标记分配给它所包含的定义。于是这样就可能把多个定义附加到一条 `if` 语句了。

主要的浏览器都忽略了这些意见，选择继续在块内实现函数声明。然而，每种实现都为这些声明发明了自己的独特语义。十五年后，这为 ES6 [TC39 2013b, Function In Block Options; §21.3.2] 的设计者带来了重大的问题。

到 1999 年春季，第三版规范明显还无法在 6 月的 GA 大会上获得批准，但等到 12 月可能还有机会。在 3 月，工作组进行了分类 [Clinick 1999]，以识别出那些为达成 12 月目标而需要砍掉或推迟的特性。被永久性移除的特性包括：`__proto__` 属性、`#` 变量、用于堆栈实化（stack reification）的调用对象（call object），以及显式的闭包对象。推迟到可能在未来版本中加入的特性则包括：原子操作、异常 `catch` 的守卫、条件编译、日期标量、十进制小数运算、泛型序列运算符、可选的 I18N 库、外部函数接口（FFI）、基于 `toSource` 的对象持久化、对数值单位的语法和运算支持，以及可扩展的字面量语法。

工作组在 1999 年 5 月至 1999 年 9 月间举行了四次会议，以解决有关第三版规范最终草案的问题。在此期间必须解决的重大设计问题包括：正则表达式匹配语义算法规范的创建、一组内置异常类型的确定、函数表达式绑定语义的确定，以及将 Unicode 支持合并到语言中时的细节。

1999 年 8 月 8 日，Mike Cowlishaw [1999c] 发布了最终的「E3 草案状态」，展示了所有状态为「内容已同意」或「自 V1 以来未更改」的章节。8 月 25 日，Bill Gibbons [1999] 发布了「第三版 3 最终草案」，并离开委员会开始了新工作。Herman Venter 和 Waldemar Horwat 负责将所有剩余的更改纳入草案。

在最后的 ES3 开发会议 [TC39 1999b] 中，Horwat 准备了很长的笔记清单，以标识对次要编辑和技术问题的更正，这其中只有少数变化会影响 JavaScript 程序员的日常。内置异常 `ConversionError` 和 `RegExpError` 被移除，由 `TypeError` 和 `SyntaxError` 取代。

对于 *FunctionExpression*[40]（函数表达式）中允许在函数名称位置出现的可选标识符，8 月的草案没有为其指定任何含义。例如：

```
function fact(n) { throw "wrong fact" }; // 函数声明
var lambdaFact = function fact(n) { // 这个函数表达式，是否应该绑定到 fact 上?
  return n <= 1 ? 1 : fact(n - 1);
};
lambdaFact(5); // 应该递归还是抛出异常?
```

在这份草案中，调用 `lambdaFact` 会抛出异常。这是因为这里 *FunctionExpression* 起始位置的 `fact` 名称，并没有为 `fact` 创建词法绑定。在 9 月的会议上达成了对规范的修订意见，会为这个名称创建一个到相应函数的本地名称绑定，这个绑定只在 *FunctionExpression* 的语句体内可见。

在最后时刻还有个最令人惊讶的新增特性，即 Waldemar Horwat 在会议上提出的「函数合并」（joined functions）。只要实现支持该特性，就可以在如下情况时重复返回相同的函数闭包对象：

```
function getClosure() { return function () {/* 没有对自由变量的引用 */ } }
var firstTime = getClosure();
var secondTime = getClosure();

// 下面的比较是 true 还是 false 由实现决定
console.log(firstTime === secondTime); // 是否是相同对象?
```

Waldemar Horwat 担心闭包创建的开销，并认为这个改动将可以让实现在某些常见情况下复用闭包。Herman Venter 表示了一些担忧，但在会议结束时同意支持这个改动。这本可能造成一个重大的设计错误，因为随后 Web 浏览器上的经验表明，这种特性所允许的某种在实现间可见的差异，可能会妨碍网站在不同浏览器上的正常工作。幸运的是，并没有浏览器实现函数合并特性，它在 2009 年也从 ES5 规范中删除。

由于在字符串字面量中，对八进制常量（以 `0` 开头的数字写法）和八进制转义序列的使用不被提倡，它们从 *规范的*[g]标准中移到了非规范性的附录 B[41]（Annex B）中。一并移至附录 B 的内容包括：与 Y2K 不兼容的 Date 方法、`escape` 和 `unescape` 字符串函数，以及字符串方

法 `substr`。这些特性都已被认定为过时，但仍被网站使用。此举背后的设想，在于特性一旦在标准的非规范性附录 B 中列出，即表明它们已被废弃而不应继续使用，各实现均有权最终删除它们。这是个幼稚的期望。TC39 成员尚未意识到，浏览器实现者们非常不愿意删除网页上实际可能用到的任何特性（不论是否标准化）——某些网页永远不会消失。

在审查并解决了所有未解决的问题后，TC39 一致接受规范，认为它已经完备，遵从并纳入了会议中所提出的更改要求。Waldemar Horwat 和 Herman Venter 准备了最终文档 [TC39 1999e]，并于 1999 年 10 月 13 日将其交给了 Ecma 秘书处。最终草案中有一张表，其中列出了 ECMA-262 前三个版本的所有贡献者（图 21），包括内容创作、技术会议参与，以及通过电子邮件的贡献。

| | | | |
|---|---|---|---|
| Mike Ang | Gary Fisher | Clayton Lewis | Sam Ruby |
| Christine Begle | Richard Gabriel | Drew Lytle | Dario Russi |
| Norris Boyd | Michael Gardner | Bob Mathis | David Singer |
| Carl Cargill | Bill Gibbons | Karl Matzke | Randy Solton |
| Andrew Clinick | Richard Gillam | Mike McCabe | Guy Steele |
| Donna Converse | Waldemar Horwat | Tom McFarland | Michael Turyn |
| Mike Cowlishaw | Shon Katzenberger | Anh Nguyen | Herman Venter |
| Chris Dollin | Cedric Krumbein | Brent Noorda | George Wilingmyre |
| Jeff Dyer | Mike Ksar | Andy Palay | Scott Wiltamuth |
| Brendan Eich | Roger Lawrence | Dave Raggett | Rok Yu |
| Chris Espinosa | Steve Leach | Gary Robinson | |

图 21. ECMA-262 第 1、2、3 版的技术贡献者。

在 11 月，最终草案中有一些较小的编辑和技术错误被确定并更正 [TC39 1999a]。其中最值得注意之处，在于微软发现当为了符合最终草案，用正则表达式来改动 JScript 的 `String.replace` 实现时，许多网

站（包括 microsoft.com 在内）会出现问题。TC39 同意更改规范，从而与微软之前的实现相匹配。

1999 年 12 月 16 日，Ecma GA 大会 [Ecma International 1999] 批准了该规范，是为《ECMA-262 第 3 版》[Cowlishaw 1999a]。自 2000 年 3 月起，Waldemar Horwat [2003b] 维护了一份非正式的 ES3 勘误表。主流浏览器陆续在 2000 年发布了与 ES3 兼容的版本。微软的 JScript 5.5 作为 IE 5.5 的一部分于 2000 年 7 月发布，而 Netscape 的 JavaScript 1.5 则作为 Netscape 6 的一部分于 2000 年 11 月发布。直到 2009 年 12 月为止，《ECMA-262 第 3 版》都没有被更新的版本替代。在此期间，浏览器并不能自动更新，并且许多用户只有在拥有新计算机或新版操作系统时，才会更新浏览器。等到 Web 开发者可以假设所有用户都使用支持 ES3 的浏览器时，已经过去了将近十年。

# 插曲：JavaScript 不需要 Java§

最初，JavaScript 被认为是 Java 的辅助脚本语言，所有复杂的编程任务都将使用 Java 来完成。但是随着对 JavaScript 的熟悉，Web 开发者们开始意识到他们其实只要有 JavaScript 就够了。

## 布道师§

随着浏览器中 JavaScript 的使用量增加，JavaScript 教育者和布道师应运而生。这其中最具影响力的人物之一，就是 Douglas Crockford。从一篇简短的线上文章《JavaScript：世界上最容易被误解的编程语言》[Crockford 2001a] 开始，他就承担起了改变软件开发社区对 JavaScript 态度的任务。Crockford 在他的另一篇文章中解释说：

> 当 JavaScript 刚出现时，我认为它不值得关注。很久之后我重新审视了它，发现隐藏在浏览器中的是一门出色的编程语言。我最早的态度来源于 Sun 和 Netscape 对 JavaScript 的最初定位。为了避免将 JavaScript 定位为 Java 的竞争对手，他们对 JavaScript 做了许多错误的陈述。这些虚假宣传一直在针对（友善度）和业余爱好者市场的大量不良 JavaScript 书籍中流传。

Douglas Crockford [2001d; 2002a; 2003; 2006] 揭示了 JavaScript 类似于 Scheme 的闭包和类似于 Self 的对象机制，并说明了该如何使用它们。但他并没有掩盖 JavaScript 的缺陷和怪癖。除了识别出这些特性之外，Crockford [2001e; 2002d] 还创建并推广了 JSLINT [Crockford 2001b]，这是第一个广泛使用的 JavaScript linter[42] 程序。另外，Crockford [2001c; 2019b] 还为 JavaScript 开发者引入了压缩[43]（minimization）的概念，并创建了 JSMIN 工具。他写了一本畅销书 [Crockford 2008b]，告诉程序员该如何使用 JavaScript 的优点并规避缺点。最后，他成为了 JavaScript 标准化工作的参与者。

Crockford 倡导简单性，他意识到可以通过使用 JavaScript 对象和数组字面量语法子集的形式，实现独立于语言的数据交换格式，从而避免 XML 的复杂性。他将这种被广泛使用的格式命名为「JavaScript Object Notation」或「JSON」[Crockford 2002b, c; Crockford 2019a]。这种简单的格式可以很容易地在任何语言中解析，但在 JavaScript 中尤其容易处理，因为有 `eval` 函数可以将 JSON 数据转换为 JavaScript 对象[44]。

# 富互联网应用与 AJAX§

早期的交互式 Web 应用主要是基于表单的。用户会将数据输入 HTML 表单，然后由浏览器传输回 Web 服务器，在服务端处理数据并更新数据库，最后将更新的 HTML 文稿传输回浏览器显示。JavaScript 在浏览器端用于基本的输入数据验证，以及对服务端生成的 HTML 做简单的动态更改。这种 Web 应用的形式后来被表述为 Web 1.0[45]。

一些应用程序具有高度的交互性，需要丰富的低延迟用户界面。于是不可避免地，有些开发者想要开发具备这些特性的 Web 应用。当 Netscape 在 1995 年将 Java 和 JavaScript 引入 Web 浏览器时，其计划是 Java 将成为实现复杂交互式 Web 应用的主要语言，而 JavaScript 将主要用于基于表单的应用中 [Shah 1996]。在 1990 年代末和 2000 年代初，许多「富互联网应用」[Allaire 2002] 被构建为 Java Applets。

在 1997 年，微软发布了其企业电子邮件客户端的 Web 版本，这就是被实现为 Web 1.0 风格的 Outlook Web Access（OWA）[Bilic 2007, Van Eaton 2005] 应用。而后，OWA 1.0 被交互更丰富的版本所接替。这个新版本使用了动态 HTML（Dynamic HTML[46]）和一个名为 XMLHTTP [Hopmann 2006] 的新浏览器 API。XMLHTTP 使得网页上的 JavaScript 代码能与服务端来回异步传输数据，而无需完全重新加载网页。DHTML 和 XMLHTTP 的组合，使得 Web 页面在每个会话中只需加载一次，然后即可作为支持远程访问数据和服务的交互式应用而运行。

在 2000 年代上半叶，许多组织都使用过类似的技术来构建 Web 应用。但直到 Google 用它来实现 GMail、Google Maps 和其他应用后，这种 Web 应用风格才广为人知。Jesse James Garrett [2005] 创造了「AJAX」一词来形容它。AJAX 和使用它构建的社交媒体应用，成为了 *Web 2.0*[g] 时代的标志。

Web 2.0 和 AJAX 的出现，是 JavaScript 在 Web 开发中用途的主要转折点。JavaScript 的角色逐渐由一门用来将动态元素添加到静态页面的语言，变为了一门用来对复杂的富互联网应用（RIA）进行编码的语言。

同时，浏览器的生态系统正变得越来越复杂，总有各式各样市场份额很低的新浏览器出现。Netscape（在被 AOL 收购后）和微软（在获得市场主导地位后）逐渐放弃了对浏览器的活跃开发，这为新浏览器的出现创造了机会。*Firefox*[g47] [Mozilla 2004]、*Opera*[g] [Opera 2013]、苹果 *Safari*[g] [Melton 2003]，以及最后的谷歌 *Chrome*[g] [Kennedy 2008] 逐渐占据了有意义的市场份额。

新的浏览器都实现了对 JavaScript ES3 规范的支持，也支持被 W3C 部分指定的浏览器平台 API。但由于平台规范并不够完整和精确，大多

数新浏览器都以各种方式扩展或修改了平台的 API。并且尽管这些新浏览器不断涌现，许多用户仍在使用过时的 Internet Explorer 和 Netscape 版本。这些版本有很多 bug，并缺乏对最新语言特性和平台 API 的支持。

在一个重要的维度上，Web 浏览器与大多数其他应用平台有所不同，那就是应用程序以源码形式分发，以便在用户提供的环境中立即执行。这与传统的方案是不同的。在传统方案中，开发者可以选择特定版本的编译器和运行时库，然后在以二进制形式将其部署给用户前，构建和测试其应用。Douglas Crockford[48] 在一些演讲中，将 Web 开发的这一特色描述为：由用户（通常不知情地）选择语言的处理器。Web 开发者需要确保其 Web 页面和 Web 应用能在最终用户选择的任何浏览器上正常工作。

解决浏览器差异的一种方法，是为每个不兼容的浏览器创建单独的应用版本。然后 Web 服务器就可以在收到网页请求时，根据浏览器提供的标识信息，将不同版本发送到不同的浏览器。但是大多数应用的源码通常都由其所有版本共享，只有很小的变化会用来解决浏览器的差异。这就产生了维护应用程序多个（基本相同的）版本时的开发和运营挑战。

一种避免应用源码出现多个不同版本的方法，是维护单个源文件。当应用在浏览器中运行时，它会动态选择出特定于浏览器的变体版本。这里对变体的选择方式，主要是基于惯用的代码序列，包括执行浏览器嗅探（识别出特定的浏览器版本）或功能测试（识别出某种特性或 bug 是否存在）。

在 AJAX 应用复杂性与浏览器兼容问题的背景下，用于简化 Web 应用构建的框架和库应运而生。早期的框架包括 Prototype [Stephenson et al. 2007]、MooTools [Proietti 2006] 和 Dojo [Russell et al. 2005]，而其中最受欢迎 [W[3]Techs 2010] 的是 jQuery [Resig 2006]。这些早期的框架与库通常为 AJAX 应用提供了基础结构支撑，并为简化编码实现常见任务提供了高层面的抽象。它们还通过内部处理和隐藏许多浏览器特性变体的方式，解决了许多兼容问题。

这样一种特殊的库，已经重要到了要创造新词汇来表示它的程度。Remy Sharp [2010] 提出了「polyfill[g]」一词，它所描述的库提供了「应

由浏览器提供但仍然缺失」的 API 支持。设计良好的 polyfill 会动态检查它所提供的特性是否在环境中已经可用。只有在缺少内置支持或不兼容的情况下，polyfill 才会自行将其置入环境。早期的 polyfill 库专注于使浏览器更具互操作性，其手段主要是隐藏早期浏览器竞争中留下的遗留特性变体，或在旧浏览器中支持新的浏览器特性。如果一个特性在某种流行的浏览器中存在，但在其他流行的浏览器中却不存在，那么 polyfill 可以使 Web 应用使用相同的代码在所有浏览器上运行。随着浏览器兼容性的改善，polyfill 则成为了一种常见手法，用来尽早用上浏览器和 JavaScript 的新特性。在 Web 新特性的设计过程中，polyfill 库的创建变得十分普遍。除了对开发者有用外，通过 polyfill 还能收集到宝贵的开发者反馈，从而支持新特性和 API 的设计。

当 JavaScript 应用是朴素地将独立创建的几个部分组合而成时，命名冲突十分常见。许多框架和库提供了某种模块化机制，这通常是通过使用命名空间对象（namespace objects）和立即执行的函数表达式（IIFE[49]）来实现的。命名空间对象只是个单例对象，其主要用途是提供对函数或变量的限定（qualified）名称访问。JavaScript 1.0 的内置 Math 对象就是命名空间对象。命名空间对象的限制之一在于，它之中的所有名称都是公共的。要克服这个限制，可以将命名空间对象与 IIFE 相结合，如图 22 所示。

```
// 使用模块模式定义 services
var Services = function () {
  var privateJobCount = 0; // 「模块」的私有状态
  return { // 命名空间对象
    jobCount: function () { return privateJobCount },
    job1: function () { privateJobCount++ }
  }
}(); // Services 被初始化为调用该函数时的返回值

// 从命名空间里获取实体
Services.job1();
```

图 22. JavaScript 模块模式的示例。这里的 `Services` 函数封装了私有的实现。`Services` 会在被调用并返回命名空间对象时初始化，命名空间对象的属性暴露了「模块」的公共接口。

模块模式有几个变体，但基本概念都是用 IIFE（或有时用命名函数）的词法作用域，来封装一系列函数的某些私有状态。IIFE 会返回一个命名空间对象，其属性就是封装后需要支持被公开访问的函数。

通常认为 Douglas Crockford 普及了模块模式，但它很可能是由许多 JavaScript 程序员独立发现的。

# 浏览器博弈论§

在*浏览器大战*[2] [Borland 2003] 期间，Netscape 和微软都尝试在引入新网站能力上实现超越式的创新。他们都试图说服开发者使用其独有的特性，并开展了「在『XXX』上效果最佳」的营销活动。但如果网站无法在用户首选的浏览器下正常工作，浏览器用户会很不高兴。而且 Web 开发者也不喜欢为不同浏览器维护网站的多个版本。

即使微软为了赢得 Netscape 的市场份额，在技术和非技术方面都进行了大量投资，人们仍然意识到 JavaScript 的发展除了竞争外还需要合作。1997 年 7 月，在第一版 ECMA-262 的工作即将完成前的 TC39 会议上，微软的 Scott Wiltamuth 提出了关于未来 ECMAScript 开发的合作承诺（图 23）。

一种不同的工作方式
微软在 ECMAScript 标准上的承诺

* 我们将把影响 ECMAScript 的新想法拿上组织的台面，而非保持机密。
* 我们将实现组织内达成一致意见后的想法。
* 我们将遵守组织内的架构原则，而非发布无视原则或与其矛盾的替代品。
* 我们将不会在首先提交到 ECMA 前，发布 ECMAScript 的扩展。
* 我们将实现所有 ECMA 批准的 ECMAScript 标准。
* 我们将明确标识出所有我们目前支持但尚未批准的 ECMAScript 特性。

图 23. 微软在 1997 年 7 月 TC39 会议上的承诺 [TC39 1997g]。

Brendan Eich 回忆说在某个时候，他意识到市场的务实性严重限制了浏览器实现者能用来改善其产品的举措。例如：

- 破坏性变更（甚至 bug 修复）可能赶走用户。
- 新浏览器必须遵从于现有的浏览器。
- 如果仅在一个浏览器中进行创新，那将是浪费。
- 第一个吃螃蟹的浏览器，可能反而会丢失市场份额。

Eich 意识到这种情况很可能属于纳什均衡 [Nash 1950]，因此创造了「浏览器博弈论」一词，用以描述浏览器实现者所受到的约束。

第一个约束有时会用「不要破坏 Web！」的口号来表述。网页通常以 HTML 和 JavaScript 源码的形式存储在服务器上。每次用户访问页面时，浏览器都会对其进行重新解释。这些页面中有很多并非由其原始创建者维护，但仍在活跃使用中，其中还包括一些具有持续效用或历史重要性的文档。一旦浏览器解释源代码的方式发生*破坏性变更*[8]，就可能导致某个页面变得难以辨认或无法正常工作。如果变化仅在单个浏览器上发生，那么用户可以切换到使用其他浏览器。如果这种变化在浏览器中普遍存在，那么这部分失去维护的 Web 就会永久损坏。这个事实也限制了 Web 标准的开发。一旦浏览器实现者认为某个标准所引入的特性（或授权做出的改动）会使得现存的大量 Web 内容失效，那么这个标准就将被忽略。

如今，浏览器开发者普遍认识到作为 Web 及其开放标准基础的兼容要求，限制了他们通过单方面平台创新进行竞争的能力。浏览器「可以并且确实」会在实现的质量（如性能、安全性、可靠性和可用性）上进行竞争。但要想提高浏览器作为应用平台的基本技术能力，通常需要所有主流浏览器之间的合作。

浏览器博弈论是 JavaScript 演化的重要因素。它还可以提供一个理解 JavaScript 为何成功的视角，并解释 JavaScript 历史上许多创新的成败缘由。

1. 创立标准
    1. 寻找场地
    2. 首次 TC39 会议
    3. 编写规范
    4. 命名标准
    5. ISO 快速通道
    6. 定义 ECMAScript 3
2. 插曲：JavaScript 不需要 Java

1. 布道师
2. 富互联网应用与 AJAX
3. 浏览器博弈论

# 改革失败 · JavaScript 二十年


## JavaScript 二十年


- 


- JavaScript 二十年


- 
- 
- 



# 改革失败§

# 不满于成功§

千禧年到来之际，以万维网（World Wide Web）为代表的互联网，正对世界产生着巨大的影响 [Miniwatts Marketing Group 2019]。随着 Netscape、微软和其他浏览器厂商不断增强浏览器的实用性，Web 得以迅速发展。Web 的成功与其持续演化的诉求，催生了 Ecma TC39 和 W3C 等工作组。这些组织中有些参与者是行业专家，他们并未直接参与浏览器开发。这些人的兴趣集中在理想化的未来 Web 上。从这个角度来看，现有的实用主义 Web 技术被当作了对未来的障碍。

1998 年 5 月，W3C 举办了名为「塑造 HTML 的未来」的研讨会。研讨记录中的结论如下：

> 在讨论中，人们一致认为进一步扩展 HTML 4.0 是困难的，将 4.0 转为 XML 应用也会是困难的。要克服这些限制，我们提议的方法是重新开始使用基于 XML 标签集的下一代 HTML。对于更好地适配数据库与工作流应用，以及对于支持小型 / 移动设备上更广泛而多样的特性，研讨会上都表达了相应的需求。模块化的 HTML 将为此提供这所需的灵活性 [W3C 1998]。

研讨会中 IBM 代表 David Singer [1998] 的演讲则更加直言不讳，他说「我们知道 HTML 的未来应该是这样的：讨厌、粗野而贫乏。」

在 ES3 即将完成时，TC39 也处于类似的处境中。借着 ES3，ECMAScript 规范也算与 Netscape 和微软浏览器中提供的 JavaScript 特性接轨了。并且至少在当时（早期），浏览器厂商并未过多引导干涉语言的未来规划。与 1995 年的 Netscape 不同的是，现在的 TC39 已经不必再规避类似 Java 的特性了。一些 TC39 的参与者意识到了对第二代浏览器脚本语言的需求 [Raggett 1999b; TC39 1999c; Appendix J]，这样的一门语言可以纠正原始 JavaScript 中的设计错误，并提供满足专业软件开发者需求的特性，而非仅仅满足非专业的脚本编写者。打造新一代 ECMAScript 的目标集中在了 ECMA-262 的第四版上。这个版本在 TC39 内部最初被称为「E4」，后来则称为「ES4」。

> TC39 对 ES4 的尝试共进行了两轮，本文中用「初版 ES4」和「新版 ES4」区分它们。

# 对 ES4 的第一轮尝试§

自首次 TC39 会议上 Borland [1996] 提出在语言中添加类（class）定义的提案起，人们一直希望尝试在 JavaScript 中添加新特性，以便应对大型程序的复杂性。Netscape 的 JavaScript 1.2 支持运行加密签名后的脚本，它们可以通过 `import` 和 `export` 声明 [Netscape 1997a] 相互集成。微软的 JScript 3 则包含了条件编译特性 [Clinick 1997]。1998 年 2 月版的《ECMAScript 展望一览表》[TC39 1998c] 将「包（package）概念」列为了 V2 的可选项。这类用于支持大型项目开发的特性较早从 ES3 特性集中移除，但 TC39 仍然在并行地进行着这些工作。

与其相关的首个重要提案来自 Dave Raggett，他是由惠普赞助的 W3C 研究员。当时 Raggett 正在 W3C 开发一项名为「Spice」的提案，以改善 HTML、CSS 和 JavaScript 的集成。这份提案的早期版本 [Raggett 1998c] 于 1998 年 2 月提交给了 TC39。除了与 HTML 和 CSS 相集成的特性外，Raggett 的初始提案中还包括了一种用于声明原型对象的构想。这种构想与 Borland 的类声明提案相似，增加了用于将事件处理器与原型对象声明式关联的能力。这份提案还包括了用于定义「库」（library）和从库中导入各种定义的设计，如下所示：

```
// 1998 年 2 月的 Spice 提案
import document, block, Inline from
"http://www.w3.org/Style/std.lib";

prototype Link extends Inline
{
   href = "http://www.w3.org/";
   when onmousedown
   {
      document.load(this.href);
   }
}
```

根据 1998 年 3 月的会议记录 [TC39 1998d]，当时会上讨论了 Dave Raggett 最初提交的 Spice。记录中指出对其的「初始反馈是负面的」。于是 Raggett 与来自 HP Labs 的两位语言设计师 Chris Dollin 和 Steve Leach 一起继续发展他的提案。在 9 月，Raggett 提交了一组新的文档 [Raggett et al. 1998] 来描述扩展后的 Spice 提案。实际上，这个提

案是一门不兼容 ECMAScript 的替代性语言，甚至用基于闭合关键字的语句语法替代了花括号分隔的 C 式语法。

在 1998 年 11 月，Spice 的设计者与来自 Netscape 和微软的 TC39 代表之间举行了一次私人会议。而后在当月的 TC39 工作组会议上，Dave Raggett [1998a] 介绍了经过修订的 Spice 提案。工作组会议上，虽然 TC39 成员对于「替换当时的语句语法」或「立即尝试集成对 CSS 的声明式支持」都缺乏兴趣，但他们有兴趣用 Spice 提案中的某些概念来扩展 ECMAScript，例如类、数值单位[50]、类型和模块机制等。Raggett 在讨论中指出，一旦类似特性添加到了 ECMAScript 中，惠普就不太可能继续开发 Spice[51]。

为了制定出一份能在 1999 年 1 月提交给整个 TC39 的的提案，委员会成立了一个新的 TC39 Spice 工作组。委员会认为，必须使用已经保留的 Java 关键字来定义支持新核心概念的新特性，并且类的语义应该与 Java 类似。各种数值单位应该基于类来定义，这需要增加对运算符重载的支持。

Spice 工作组的首次电话会议在 1998 年 12 月的第一周举行。12 月 10 日，Dave Raggett [1998b] 在会议的基础上分发了一份新文档，它主要涉及包和数字单元，但还更广泛地探讨了包括类和接口在内的类型声明。它的重点更多是语法而非语义。这份文档的设计基于 *名义化类型系统*[g]，包括如下：名义化的内置基本类型、同质（homogeneous）的数组类型、类（class）类型（其中的子类均为 nominal subtype）、接口（interface）类型，以及表示需要进行动态类型检查的 `any` 类型。在语法上，它探索了将类型与变量绑定相关联的新方法。文档仍然假定使用 `var` 关键字用于变量声明，并探讨了两种类型注解的风格。其中一种 C 式的风格将类型表达式作为变量名声明的前缀，而另一种 Pascal 式的风格则在变量名声明后书写冒号和类型表达式。图 24 中举例说明了这两种替代方案。

```
// C 风格的声明可选形式
var float x, int[] y, z; // z 的类型是什么？
var float x, int[] y, int[] z; // 是这样吗？
var float x, int[] y, any z; // 或者是这样吗？

// Pascal 风格的可选形式
var x: float, y: int[], z; // z 的类型是 any
```

图 24. 在初版 ES4 中，基于 C 和 Pascal 语法所衍生出的几种类型注解的可选形式。

对类和接口所定义的语法大致遵循 Java，包括了对 `public`、`private`、`protected` 和默认（包级）可见性修饰符的完整补充。语言底层的元对象结构未涵盖在内，但这里的元对象模型已经必须与当时的 JavaScript 原型继承模型存在隐式区别了。这份文档提出了关于如何区分「使用声明出的静态类型信息的早期绑定（early binding）成员访问」和「没有静态类型信息的延迟绑定（late binding）成员访问」的问题。文档还探索了属性的动态添加[52]，认为可以在类中禁用这一能力。

相关的设计讨论发生在 1999 年 1 月和 2 月 [Raggett 1999b, c]，主要与类、*类型注解*[g]和作用域有关。Chris Dollin、Waldemar Horwat 和 Herman Venter 是首要的参与者。许多讨论都涉及用类所定义出的对象的性质，以及类成员访问的语义。Dollin 和 Venter 主要倾向于使用类似 Java 的语义，其中类实例的结构由类声明静态确定，并且成员的可访问性都能基于类型信息静态地决定，可以在文档网站上查找到。Horwat 则主要倾向于使用更具动态性的模型，其中即使存在类型注解，也会使用不可靠的动态查找来访问成员。要想满足现有 JavaScript 程序员的预期，并兼容那些使用了「基于原型的特设类（ad hoc class）」的已有代码，似乎都需要动态语义。这里涉及的特性包括可选的类型注解，以及 *expando properties*[g]。此外 Horwat 认为，动态语义与脚本的性质更加一致，脚本编程天生地涉及从多个来源动态组装出代码，并使用独立于引用方脚本做版本控制的库。Horwat [1999b] 在描述成员查找可选方案的文档中，总结了使用静态方案和动态方案之间的区别。

在 2 月的会议上，Waldemar Horwat [1999a] 展示了他的「JavaScript 2.0」规范。他说这是一份最早为 Netscape 编写的实验性设计[53]，但与 TC39 最近讨论的内容相匹配[54]。规范中包含了具有大量机器级数字类型的名义化类型系统，类似 Java 的类成员可见性规则，以及带有显式 import 支持的包。它还具有许多更新颖的特性，包括：类扩展声明、包成员的声明级版本控制、nullable 和 non-nullable 的类型，以及一等公民的类型值。JavaScript 2.0 提出了一种「流式执行模型」[Raggett 1999d]，取代了之前 JavaScript 版本的声明提升（declaration-hoisting）语义。在这种语义下，声明要在执行过程中遇到时才会被处理。例如

`if` 语句可以用来选择性地声明变量，或者用来选择带有不同类型注解的声明。像这样「一等公民类型值」和「声明的流式执行」的结合，会使得在某些情况下无法进行完整的静态类型检查。

JavaScript 2.0 并未尝试与原始 JavaScript（甚至还有尚未完成的 ECMAScript 3）完全向后兼容。在向 TC39 介绍 JavaScript 2.0 时，Waldemar Horwat 表示：「至少，你应该能写出在 ECMAScript 1.0 和 2.0（ES4）中都能工作的代码。完全向后兼容会非常痛苦 [Raggett 1999c]。」例如可选类型注解所带来的语法复杂性，会导致难以在换行符上支持自动分号插入。Horwat 对向后兼容性的解决方案是在实现中提供多个编译器。他认为根据语言版本切换编译器，要比使用具有严格前向兼容性的单一语言更可取。

在 1999 年剩下的时间中，TC39 的大部分注意力都集中在完成 ES3 上。但是在 3 月，它发布了《未来特性展望表》[TC39 1999c]，介绍了可能在 ES3 之后发布的特性。Spice 工作组则转为了一个模块化子组，并继续时常举行关于初版 ES4 的会议 [Raggett 1999a, d; TC39 1999a]。到 11 月，TC39 已经将主要注意力转移到了「第 4 版」上，并更新了 ES3 之后的未来特性展望表（见图 25），于是这一步伐加快了。TC39 主席 [Lewis 1999a] 在 1999 年 11 月的报告中描述了初版 ES4 的目标：

> ECMAScript 2.0 是一个进取且大幅改进的 ECMAScript 语言规范，委员会希望在 2000 年实现标准化（当然这个野心可能过大）。ECMAScript 2.0 的主要目标是为「大规模编程」提供支持——这也就是说，要支持开发由多人构建的程序，并可能是史上第一次要在用户的桌面上组装出这些程序。

```
* 模块化增强：classes, types, modules, libraries, packages 等
* 国际化（I18N）相关：
  - 国际化库 [可能作为独立的 ECMA 技术报告]
  - 日历
* 十进制小数（对 Number 对象的增强或替代）
* Catch guards（带类型）
* 对原子（线程安全）操作的讨论或定义（可能非规范性）
* 其他各类更新 [除模块化相关外]：
  - 声明修饰符的扩展机制
  - 可扩展的语法（如用 # 表达 RGB 值）
```

- 单位语法和算术库
- Here 文档（长字符串常量）

图 25. TC39 在 1999 年 11 月的《未来特性展望表》[TC39 1999d] 中设想的初版 ES4 特性。

在 2000 年 1 月的会议上 [Raggett 2000]，微软砍掉了一些特性，希望能赶在 2000 年 12 月发布规范的第四版。微软的主要兴趣是添加静态类型注解并保持向后兼容性，包括对自动分号插入的支持。Venter 介绍了对 ES3 规范的一系列更改，他认为这些更改足以支持类型注解。但此时在类型系统性质方面仍然存在很多不确定性，包括：类的语义、包的语义、命名空间的语义，以及如何将静态和动态的语言概念集成到单个语言中。

在 2000 年 6 月 22 日，微软 [2000b] 发布了 .NET Framework，这是微软为应对 Sun 在 Java 平台上的竞争所做出的回应。微软 .NET 是一个多语言的应用开发平台，除了主要语言 C# 外，它还支持 Visual Basic 和 JavaScript 等其他语言的方言。消息发布后，第一个 .NET 预览版[55] 在 7 月的微软专业开发者大会上发布 [Microsoft 2000a]。预览版包含了早期版本的 JScript .NET [Clinick 2000]。与浏览器中的 JavaScript 不同，JScript .NET 是一门面向 .NET 公共语言运行时（CLR）的预编译语言，其内部使用了 .NET 的类型系统。Internet Explorer 并不支持 JScript .NET（或通称为 .NET）。相反地，JScript .NET 一开始就可以使用各种 .NET 框架组件来构建桌面、服务端和命令行应用。JScript .NET 宣称自己与 ES3 规范兼容，但由于其设计目标并非运行为浏览器编写的 JavaScript 代码，因此它并不需要严格的向后兼容性。除了 ES3 特性外，JScript .NET 还添加了可选的静态类型注解、包含成员可见性属性的类和接口声明，以及支持显式 import 的包。根据微软 Andrew Clinick [2000] 的说法，这些新特性是与其他 Ecma TC39 成员一起设计的。他还告诫说，根据正在进行的 TC39 讨论，设计的细节可能会改变。

在 .NET 于 2000 年 6 月发布前，微软的 Herman Venter 并不能与 Waldemar Horwat 或其他 TC39 成员讨论 .NET 或 JScript .NET。当年 8 月，Horwat 和 Venter 私下见面，试图就完成 ES4 标准达成足够的一致。Horwat [2000] 在会议笔记上记录了对 43 个问题与分歧的讨论，其中总结了以下讨论：

> 概括说来，Herman 正在为服务端准备 JScript 的实现，并希望冻结这门语言，使其易于与微软 .NET 运行时互操作。Waldemar 担心这门语言对浏览器的适用性，并希望保留语言的动态性。他认为这是 ECMAScript 的与众不同之处。Waldemar 担心语言向 Java 或 C# 发展，因为他认为在这两者擅长的领域里，几乎不需要另一种新语言。而且对于静态编程来说，新语言最终能获得的结果也比不上 C# 语言。Herman 还建议在新的服务端项目上使用 C# 而非 JScript，并将新的 JScript 视作一种针对「已经习惯使用 JScript 编程的开发者」的语言。

Horwat [2003a] 从 JavaScript 2.0 文档中 fork 出了一份单独的《ECMAScript 4 Netscape 提案》文档，然后将这份文档用作正在进行的初版 ES4 开发的工作草案。JavaScript 2.0 文档则继续并行维护，包括了 TC39 尚未同意加入的其他特性。

微软希望 .NET 及其语言能被视为标准化的技术。Ecma 组织在这方面有口皆碑，可以很容易地将专有技术转移到标准轨道上，并且微软对 TC39 的工作方式也感到满意。因此，微软向 Ecma 建议扩展 TC39 的职责范围，并在其中将 .NET 标准化。于是 TC39 被重新定义为 Ecma 面向「编程环境」的技术委员会，正在进行的 ECMAScript 开发活动在 TC39 中被降级到了 TG 任务组（Task Group）状态，称为 TC39-TG1。Ecma 成立了其他的 TC39 任务组，以开发 CLR 和 C# 的标准。

这次创建 ECMAScript 规范第四版的尝试还会再进行三年。但事后看来，JScript .NET 的发布已经敲响了这项工作的「丧钟」。到 2000 年 6 月，Netscape 已经输掉了「浏览器大战」[Borland 2003]，其浏览器市场份额下降到了 14% 以下 [Reuters 2000]。在被美国在线公司收购后，它正逐渐失去人手，被迫以更少的资源运营，只能艰难地继续更新其浏览器。

微软凭借 IE 在竞争中取胜，并最终获得了 90% 以上的市场份额。它对继续增强自己没有专有控制权的 Web 编程平台的兴趣不大。在微软内部，研发资源从增强开放的浏览器技术（例如 ECMAScript）转到了开发专有的微软技术（如 Windows Presentation Framework[56] [Microsoft 2016]）上，微软希望它们能最终淘汰和取代开放的 Web 技术。在 .NET 的编程语言领域，微软专注于 C# 和 VisualBasic .NET。在这种情况下 JScript .NET 的重要性，仅仅取决于它能使多少 JavaScript 程序员迁移到 .NET 平台。

TG1 继续开展会议，讨论特定问题，并更新规范草案[57]。微软与 Netscape 在类型系统的性质方面存在着重大的持续分歧。Waldemar Horwat 在 MIT 轻量语言研讨会上发表了有关 JavaScript 2.0 设计的论文 [Horwat 2001]，他将 JavaScript 2.0 描述为具有「强大的动态类型」能力的语言。他进一步解释说，在 JavaScript 2.0 中，所有变量都具有关联的类型，这些类型限制了可以存储在其中的值，但是类型约束的检查必须在运行时进行。在一般情况下，JavaScript 2.0 的一等公民类型值和隐式的 downcast[58] 会导致无法对程序进行静态类型检查。

TG1 召开会议的频率和出席人数都在逐渐降低。Chris Dollin 于 2001 年 6 月参加了最后一次会议，而 Herman Venter 参加的最后一次 TC39-TG1 会议则是在 2002 年 6 月。2003 年 7 月 15 日，美国在线宣布将解散 Netscape，并解雇了包括 Waldemar Horwat 在内的大多数员工。在同一周举行的 TG1 会议上，Horwat 辞去了 ES4 编辑的职务。TG1 的其余成员决定将精力集中在为 ECMAScript 开发 XML 支持上，并中止 ES4 的工作，直到 XML 项目完成并可以确定出新的编辑为止。

## 另一条死路§

在 1990 年代中期到后期，人们对软件组件（component）的概念产生了浓厚的兴趣，并提出与实现了几种软件组件模型。这其中包括来自对象管理组织（OMG）的 CORBA、微软的 COM，以及 Sun 的 JavaBeans。一般意义上，软件组件模型是一种模块化方案，提供了一种可以「描述、发现和使用基于对象的软件模块」的方式。在 1997 年 7 月的 TC39 会议上 [1997g]，代表 Oracle 的 Jim Tressa 作了有关组件脚本语言的 OMG 提案的介绍。据称在那次会议上，IBM、Netscape、Oracle 和其他公司都有兴趣对基于 ECMAScript 的提案作出回应，但是 OMG 最终制定的规范并非基于 ECMAScript。

「ECMAScript 组件」概念希望发布特定于 JavaScript 的组件模型，以便在浏览器和其他 JavaScript 宿主中使用。它定义了一份 XML 模式（schema）和词汇表，以描述 JavaScript 组件和一组实现者应遵守的约定。这项工作的赞助者是 NetObjects[59] 和 Netscape 这两家公司。

NetObjects 的 Richard Wagner [1998] 于 1998 年 6 月向 Ecma GA 大会进行了初步介绍。在同一次会议上，相应的技术规范草案 [Wagner and Shapley 1998] 被提交给了 TC39。这份文档经过了三份草案的修改，然后提交给了 Ecma GA。它被批准为 Ecma 标准，并发布为了 ECMA-290 [Wagner 1999]。然而根据记录，这份标准并没有被实际实现过。根据 TC39 的建议，Ecma GA 大会在 2009 年投票决定撤回 ECMA-290 标准 [Ecma International 2009b]。

ECMAScript 第三版中的「精简模式（compact profile）项目」为 ES3 的一个动态程度较低的子集定义出了一种语言*模式*[g]，这可以使得资源受限环境中的 JavaScript 实现仍然符合 ECMAScript 规范。这一规范的创建 [Raggett 2000] 是由 Ecma 之外的 WMLScript 推动的，目标是定义用于手机应用[60]的 JavaScript 方言 [Lewis 1999b]。精简模式包含了 ES3 的所有特性，但允许实现移除对 `with` 语句的支持。实现也可以移除对 `eval` 和 `Function` 构造函数的支持。精简模式还支持让内置库的对象不可变的实现，这样可以进行预编译，或提供基于 ROM 的实现。Ecma GA 大会批准了 ECMA-327 精简模式标准 [Vartiainen 2001]。与 ECMA-290 不同的是，ECMA-327 实际上已经在某些环境中实现了。但是随着新版 ECMA-262 的发布，人们对更新 ECMA-327 缺乏兴趣。ECMA-262 的最新版本已经使用在了资源非常受限的环境上。如果针对这类环境的实现需要移除某些特性，直接做就是了。实际上对于大多数资源受限的应用而言，并没有证据表明实现之间需要具备完美的 JavaScript 互操作性。Ecma GA 大会在 2015 年投票决定撤回 ECMA-327 标准 [Ecma International 2015b]。

在 2002 年，TC39-TG1 将大部分注意力转移到了开发「ECMAScript for XML」的规范上。所谓 E4X 是一个单独的 Ecma 标准，它向 ES3 添加了语法扩展，从而支持对 XML 文档的处理。相应的 ECMA-357 版本 [Ecma International 2004; Schneider et al. 2005] 分别于 2004 年和 2005 年发布。Firefox 是唯一实现 E4X 的浏览器，因此就像浏览器博弈论中指出的那样，这个能力很少被使用。到了 2015 年，由于 E4X 扩展与 ECMAScript 2015 不兼容，ECMA-357 这项 Ecma 标准也被撤回 [Ecma International 2015b]。

## Flash 与 ActionScript§

Macromedia 公司的 *Flash*[g]（后来被 Adobe 收购）在 2000 年代初问世，成为了人们在构造富互联网应用时，对 Java 和 JavaScript 的流行替代品。Flash 最初是一个基于时间轴的动画产品，建立在 Jonathan Gay [2006] 工作的基础上。Flash 附带了视觉创作工具，它能将基于动画的应用编译为二进制文件，这些文件由 Flash Player 解释执行。播放器组件可以通过浏览器的插件扩展 API 集成到浏览器中。在巅峰时期，几乎所有浏览器用户都安装了 Flash 播放器 [Adobe 2013]。

最初的 Flash 创作主要是偏视觉化的，但它的功能还包括编写简短的文本「动作」（action），以定义对各种时间轴事件的响应。在 1999 年 5 月发布的 Flash 4 中，Gary Grossman 将 Flash 动作演变成了一种简单的动态类型脚本语言，其语法与 JavaScript 相似。随着 2000 年 Flash 5 的发布，这门脚本语言成为了 ECMAScript 3 的方言，并被命名为「ActionScript」。*ActionScript*[g] 1.0 支持大多数 ES3 语句和基于原型的对象，但缺乏对正则表达式的支持，具有非标准的 `eval` 函数，这种 `eval` 只能求值一组受限的变量访问表达式，此外还有各种其他细微的语义差异。由于 ActionScript 代码被编译为仅在 Flash Player 环境中运行，因此并不必严格遵守 ECMAScript 规范的语义。例如在 ActionScript 1.0 中，`var` 声明的作用域是最接近它的封闭块，而非整个封闭函数。

在 2003 年，ActionScript 2.0 作为 Flash MX 开发环境和 Flash Player 6 的组件发布。它扩展了 ActionScript 1.0，支持类声明、接口声明、声明中的类型注解，以及用于访问其他脚本中定义的类的 import 语句。其中，类的类型注解、类声明和接口声明的语法大致遵循了初版 ES4 与 JS2 规范草案中使用的语法，但语义则大大简化。对类型注解的使用是可选的。类型检查属于仅限编译期的特性 [Macromedia 2003]。如果提供了类型注解，语言会在编译时执行类似 Java 的名义化类型检查。但在生成代码前，类型信息会被擦除。ActionScript 2.0 使用了与 ActionScript 1.0 相同的虚拟机，并执行基本的运行时安全检查。程序可以按违反名义化类型系统规则的方式来动态修改对象，只要这些更改不会触发任何运行时安全检查即可。

在 2003 年，Flash 在 Web 开发中获得了广泛的应用，这带来了复杂的大型 ActionScript 应用，其中有一些遇到了性能问题。与当时的大多数 ECMAScript 语言设计者和实现者一样，Macromedia 团队认为[61]动态类型（尤其是原始类型）导致了主要的性能瓶颈，并且正在探索向

ActionScript 运行时添加静态类型的方法。大约在同一时间，自 1998 年以来一直担任 TC39 代表的 Jeff Dyer 加入了 Macromedia。Dyer 确认了 TC39 对静态类型持有相同的观点。这种在基于虚拟机的语言中应用静态类型的观点广为流传，受到了对静态类型 Java 虚拟机（JVM）设计的强烈影响。Macromedia 的 Jonathan Gay 和 Lee Thornason 研发了实验性的 Maelstrom 项目，旨在研究 JVM 是否可以集成到 Flash 中，并用作静态类型版 ActionScript 的运行时。这个实验非常成功，以至于 Macromedia 向 Sun 就有关将 Java 2 Micro Edition（J2ME）版本 JVM 用于 Flash 的许可进行了洽谈。他们想使用 J2ME 的理由，是因为标准版 Java 运行时太大，无法嵌入 Flash Web 下载。但是 Macromedia 这项对使用 Java Micro Edition 技术的提议，与 Sun 的 Java 许可策略并不相符。于是 Edwin Smith 经过大量工作，创建了一系列概念验证性的虚拟机。这些虚拟机帮助 Macromedia 构建了自己的静态类型 JVM 式虚拟机 AVM2 [Adobe 2007]，并在其上运行了新版本的 ActionScript。这种新语言是由 Gary Grossman，Jeff Dyer 和 Edwin Smith 设计的，它受到了 Horwat 的初版 ES4 / JS2 规范草案的重度影响。但是与 JScript .NET 一样，ActionScript 3.0 是初版 ES4 设计的简化。它不像 JS2 那样动态，并且与 JScript .NET 不同的是，它不受 .NET 类型模型的约束。另外 ActionScript 3.0 与 JScript .NET 还有一点相似之处，那就是它们都不会重度受制于旧版的兼容性问题。Flash 会同时附带用于支持 ActionScript 3.0 的 AVM2，以及用于支持 ActionScript 1.0 和 2.0 的 AVM1。这项创建新版 ActionScript 和新虚拟机的工作耗时三年才得以完成，相应产物在 2006 年作为 Flash Player 9 的组件而发布，最终于 2007 年交付。在工作完成之际，Adobe 收购了 Macromedia，而 Flash 则成为了 Adobe Flash。

## 对 ES4 的第二轮尝试§

虽然初版 ES4 的开发工作在 2003 年停滞了，但 Web 上对 JavaScript 的使用仍在继续增长。不到一年内，TG1 成员就再次开始考虑设计一个被称为「ES4」的新版本 ECMAScript 了。

## 重置 TC39-TG1§

Macromedia 于 2003 年 11 月成为了 Ecma 会员，Jeff Dyer 则成为了 TC39 的代表之一。Macromedia 此举的意图是很明显的，因为 ActionScript 3 的设计受到了 TG1 最初开发的 ES4 规范的强烈影响。对 Macromedia 而言，让 ActionScript 的设计与将来的 ECMAScript 规范保持一致非常重要，并且他们也需要 TG1 考虑来自 ActionScript 的需求和先例。

2004 年春季，Mozilla 基金会发布了 Firefox 浏览器的技术预览版，有望在年底之前发布 Firefox 1.0。Mozilla 的首席技术官 Brendan Eich 对开放 Web 的未来感到担忧。此时业界对基于浏览器的 Web 应用的兴趣正在迅速增长，但当时最新的浏览器标准并不足以支持交互足够丰富的应用。像 Flash 和微软 WPF 与 .NET 这样封闭的专有应用平台，正在竞相取代 HTML / CSS / JavaScript 的 Web 技术栈，但负责开放 Web 的标准化组织并未响应这一挑战。1998 年，W3C [W3C 1998] 决定停止发展 HTML，转而支持基于 XML 的替代方法。但是 XHTML 在语法和语义上都不兼容 HTML，并没有被浏览器厂商和 Web 开发者普遍接受。同样地，Ecma TC39-TG1 对发展 ECMAScript 规范的尝试也陷入困境，它的注意力已经转移到了设计 ECMAScript 的 XML 支持上。Web 技术社区的一些成员担心「ECMAScript 已死」[Schulze 2004b]。

在这个时候，Brendan Eich [2004] 站了出来，促进了 WHATWG（Web 超文本应用技术工作组）[Hickson 2004] 的成立，这个工作组专注于 HTML 的未来。他还开始重新介入 TG1。Eich 于 2004 年 3 月会见了 Ecma 秘书长 [Marcey 2004]。当年 5 月，Mozilla 基金会申请了 Ecma 会员资格。在 2004 年 6 月，Eich 自 1998 年 2 月以来首次参加了 TG1 会议 [Schulze 2004a]。

在 6 月的会议 [Schulze 2004b] 上，TG1 的召集人（Convener）职责从微软的 Rok Yu 移交给了 Macromedia 的 William Schulze。Jeff Dyer 则成为了 ECMA-262 的编辑。代表们再次致力于完成 ECMAScript 规范的第四版，但决定不再继续投入 Waldemar Horwat 的初版 ES4 草案。

根据 Schulze 的报告，「初版 ES4 太过于笼统而宽泛，难以完成或获得采用」。取而代之地，成员们同意采取「一种更为增量的途径」[Schulze 2004a]。基于这种方式，新版语言可以集成到包括 ActionScript 在内的现有实现中。被列为候选待集成的特性包括：包、命名空间、条件属性、运行时类型检查，以及 XML 支持。这份列表包括了原有初版 ES4 草案中一些最复杂的部分，但成员们仍然认可了新版 ES4 的 12 个月开发周期。Dyer 同意准备一份介绍变更计划的草案，以在 2004 年 10 月的会议上进行介绍。

TG1 暂时还无法处理这些新需求。在 2004 年下半年和 2005 年的大部分时间里，委员会的大部分注意力都集中在修订 E4X 规范 [Schneider et al. 2005] 上，这是 ISO 快速通道流程的一部分。直到 2005 年 10 月，委员会才对新版 ES4 开始了认真的工作。但在这段时间里，Brendan Eich 熟悉了 ECMAScript 当时的标准化状态，并开始在会议演讲和博客文章中公开表达其对下一版的想法 [Eich 2005a, b]。在 2005 年 9 月的会议 [TC39-TG1 2005] 上，Eich 成为了 TG1 的召集人，开始推动新版 ES4 的开发。

## 重新设计 ES4§

在 2005 年 10 月的博客文章中，Brendan Eich [2005d] 列举了下一轮 ES4 工作的四个目标，如下所述：

- 让第 4 版重新朝向当前的语言发展。这样一来，「基于原型的委托」就不再属于残留的兼容模式，而是组成语言对象系统的动态部分。这个对象系统中所包含的类，可以带有不会被隐藏或覆盖的固定成员。
- 允许标准实现者让语言自举[62]，从而表达出所有「原生」对象使用的「元对象协议」黑魔法（参见 ECMA-262 第 3 版第 15 节），包括读取、设置、调用 、构造等操作，以及对属性标记（如可枚举性）的控制。

- 加入类型注解以支持大规模编程，前提是不破坏现有版本与新版本之间的互操作性。XUL[63] 框架和现代 Web 应用都越来越需要这样的特性。
- 修复其他长期困扰着几乎所有 JS 开发者的问题。

据 Eich 所述，他的预期是在 2006 年底之前完成这项工作，其中包括初步实现以及对互操作性的测试。

Brendan Eich [2005d] 在 2005 年 11 月的博客文章中简化了这些目标，如下所示：

1. 以更强大的类型和命名支持大型编程。
2. 支持自举、*自托管*[g] 和反射。
3. 保证向后兼容性，一些简化语言的更改则例外。

他还指出，标准的目标并不是让 ECMAScript 更像 Java 或任何其他语言，也不是让 ECMAScript 更易于被优化。在随后的演讲中，Eich [2006a] 认可了对初版 ES4 规范的批评，这其中也包括了对于是否需要「声明式的静态类型」或者「类定义」的质疑。对此 Eich 反驳说，对此什么都不做是不行的。他认为，随着 Web 开发者构建出日益复杂的应用，ES3 语言在未来十年内的扩展性会显得很差。他特别指出，要想支持这样的应用，需要的是一种保证语言不变性（invariance，描述类型严谨程度的概念，译者注）的类型系统，这样的类型系统可以选择性地执行静态检查。不过做这种改变的机会只有一次，所以现在就是时候了。

Brendan Eich 乐观地认为，在编程语言规范和类型系统领域的现代研究，可以帮助解决初版 ES4 原始工作中某些领域的问题。在 2006 年初，他招来 Dave Herman 加入了 TG1 新版 ES4 的设计团队[64]。Herman 是美国东北大学的博士研究生，当时正致力于开发 ES3 的操作语义（operational semantics，用于保证程序在数学上严谨性的概念，译者注）。通过 Herman 的推荐，Eich 还邀请了圣克鲁斯大学教授 Cormac Flanagan 加入。Flanagan 是混合类型系统 [Flanagan 2006] 领域的专家。大约在同一时间，Opera Web 浏览器上的软件架构师 Lars Thomas Hansen 成为了 TG1 的经常性参与者。Herman、Hansen 和 Flanagan 都与美国东北大学的编程语言研究社区有着直接或间接的联系。

| | |
|---|---|
| Jeff Dyer | Adobe[65] |
| Brendan Eich | Mozilla |
| Cormac Flanagan | University of California, Santa Cruz |
| Lars T Hansen | Opera/Adobe |
| Dave Herman | Northeastern University |
| Graydon Hoare[66] | Mozilla |
| Edwin Smith | Adobe |

图 26. 2006 年新版 ES4 的核心设计团队。

2005 年末，TG1 为新版 ES4 项目制定了每周电话会议和每月面对面会议的时间表。图 26 列出了 2006 年的新版 ES4 核心设计团队。这些人定期参加会议，参与关键决策，并不断做出重要贡献。来自 Adobe、Mozilla 和其他组织的其他人员偶尔会参加会议和（或）做出贡献，但很少积极参与这个项目。

在 JS2 / ES4 的第一轮开发中，要想进行与现有 ECMAScript 程序不兼容的更改是非常容易的。它假定在浏览器中，HTML `<script>` 元素里的版本信息，可以用来选择语言的不同版本。在与新版 ES4 有关的新工作中，人们进一步地意识到了变更的潜在影响，但仍然希望能够通过版本控制的方式，来纠正委员眼中属于早期 JavaScript 设计错误的内容。Brendan Eich 曾经在他的博客文章和演讲中谈及这种可能性，但也有一些 TG1 成员提出了反对意见。Douglas Crockford 代表雅虎在 2006 年 7 月的 TG1 会议 [TC39-TG1 2006c] 上指出，「向后兼容是困难且重要的」。不过对于雅虎来说安全还是最大的问题。如果是为了解决与安全相关的问题，那么也可以忍受后向的不兼容性。微软的 Pratap Lakshman 则表示：「向后兼容属于最高优先级。除非为了修复安全性问题，否则向后兼容性不应该被破坏。」

2005 年，Brendan Eich 在 ICFP（函数式编程的学术会议，译者注）发表了纪念 JavaScript 十周年的主题演讲。演讲后的问答环节中，他对 Python 做出了积极的评价 [Danvy 2005]。他甚至还推测对于较大规模的 Web 脚本来说，Python 可能比 JavaScript 更好。在接下来的一年里他都在做游说，希望在新版 ES4 中再加入一些根据等价的 Python 特性直接建模的特性，包括迭代器、生成器、*解构*赋值和数组推导式。他还提倡使用具备块级作用域的 `let` 和 `const` 关键字来声明变量，从而

替代函数级作用域的 `var` 声明。在很大程度上，它们与人们为新版 ES4 提出的其他（所谓）更复杂的「大规模编程」特性并没有关系。这些特性被添加到了基于 SpiderMonkey 的 JavaScript 1.7 引擎 [Mozilla 2006a] 中，相应的 Firefox 2 浏览器版本于 2006 年 10 月发布。但是这些特性未被其他浏览器接纳，因此并未在 XUL 之外的地方获得广泛的使用。

Eich 担心其他浏览器厂商（尤其是微软）会选择非常缓慢地接纳新版 ES4 的 JavaScript 改进。另外还有一个令人担忧之处，那就是 JavaScript 引擎可能无法继续提高性能，满足不了 AJAX Web 应用涌现出的需求。有种能解决这两个问题的方法，即打造出支持设想中新版 ES4 规范的高性能开源 JavaScript 引擎。为此，Eich 说服了 Adobe 基于开源许可将其 AVM2 引擎实现贡献给了 Mozilla。Mozilla 将获得的代码库命名为「Tamarin」[Mozilla 2006b]。在后面几个月中，Mozilla 发布了 [Eich 2007a] 两个项目：一个是旨在用 Tamarin 代码库替代 SpiderMonkey 的 ActionMonkey，而另一个是基于 Tamarin 的 JavaScript 引擎 ScreamingMonkey，它可以当作 Internet Explorer 的第三方插件扩展来安装。这两个项目都没有完成。

在执行这些工程行动之际，TG1 同时也在继续致力于新版 ES4 的设计。新版 ES4 的主要目标是提供一种类型系统和类型注解符号，用来在大型程序中验证对复杂数据抽象的使用。只要是恰当地写出的程序，都应该能在部署前做静态类型分析。但这样的类型系统不仅需要能处理新程序和现有未加注解的程序，还要能处理当前语言所支持的对象动态结构变化。在 2006 年的大部分时间里，委员会都在理解这些需求的含义，并尝试设计出一种类型系统来适应它们 [TC39-TG1 2006a, d]。

委员会的工作起点，是 ActionScript 3 规范 [Macromedia 2005] 中具备非正式描述的类型系统。这是一个名义化类型系统，其中的类和接口类型与 Java 类似，而泛型则是在这基础上加入的。ActionScript 3 支持为声明添加类型注解，而对于缺少显式类型注解的声明，语言还引入了一种通用类型。ActionScript 3 规范没有明确加入「函数子类型」的概念，并且对类和接口子类型的定义也并不完整。语言还具有严格模式，这个模式下会使用具有类型注解的声明和有限的类型推断，来执行提前（ahead-of-time）的静态类型检查。另外也有一种标准模式，可以用来根据类型注解来动态验证实际数据值。

Dave Herman 和 Cormac Flanagan 早期提出的建议，是使用契约模型（contract model）[Findler and Felleisen 2002] 来更好地统一严格和标准模式，以及类型化和非类型化的声明。随着工作的进行，结构化类型（structural type）[TC39 ES4 2006d] 也被加入了进来，用于处理对象和数组字面量。一并加入的还有用于处理数组类型的参数化类型。TG1 在内部[67] Wiki 网站 [TC39 ES4 2007f] 上为此考虑并记录了许多可选的方法 [TC39 ES4 2006b]。Herman 和 Flanagan 还试验了类型系统的形式化 [TC39 ES4 2007a]。到 2007 年初，语言设计仍不完整，但已演变到覆盖了许多现代类型概念，包括函数类型和协变 / 逆变（co/contra-variance）[TC39 ES4 2007b]。语言还需要支持可选类型（optional typing）和历史遗留的动态类型程序，这个现实需求一直是复杂性的重要来源。

在整个 2006 年和 2007 年的大部分时间里，TG1 继续致力于制定新提案和完善现有提案，最终在内部 Wiki 上整理出了一份列表。这其中包括了 54 份已批准的提案，它们都被规划进了新版 ES4 规范里。另外的 26 个提案则被推迟或删除。

在 Brendan Eich 发现了 Dave Herman 介绍如何对 ES3 的形式化语义进行文档化实验的网页 [Herman 2005] 后，Herman 也被招募进了 TG1。2006 年 2 月的 TG1 会议 [TC39-TG1 2006b] 上，Herman 介绍了用于规范化编程语言的形式化技术。他解释说，除了为实现者提供指导之外，形式化的规范还提供了一种在规范中查找和纠正错误的方法。有人担心 ECMAScript 实现者和规范的其他使用者是否能阅读这种形式化的规范。对此 Herman 认为，基于操作语义的形式化是可以变得非常易读的。在接下来的几个月中，Herman 尝试使用 Maude [Clavel et al. 2003]、Stratego [Visser 2001] 和 PLT Redex [Matthews et al. 2004] 等工具来确定 ECMAScript 的语义，但最终发现它们都不够令人满意。在同一时期，TG1 还讨论了 [TC39-TG1 2006e] 根据参考实现（Reference Implementation）来定义语言的可能性。还有一种可能性是专门为 ECMAScript 设计一种新的形式化规范语言。在 2006 年 10 月的会议上，TG1 讨论了这种语言的可能语法和语义。最后 Cormac Flanagan 指出，工作组现在已经在讨论定义两种语言（规范语言和新版 ECMAScript）的工作。于是工作组很快同意使用现有的语言来为新版 ES4 编写定义解释器，并迅速决定使用 SML[68] 语言 [Milner et al. 1997]。到 11 月中旬，TG1 已经为此搭建了工具和基础架构，相关成

员也在着手编写解释器。Herman 和 Flanagan [2007] 描述了这对委员会工作风格的影响，如下所述：

> 在我们选择了确定性的解释器后，委员会的互动方式就发生了很大变化，从每月一天半的讨论型会议转为了为期 3 天的 *hackathon*[g]。这中途也会进行技术讨论，涉及语言设计和实现时的各种极端情况。这些问题都被逐一发现和解决。

## 阻力§

微软几乎没有参与重启新版 ES4。虽然 DevDiv 在组织上远离了负责 IE 的微软 Windows 团队，但 JScript 的开发始终由 DevDiv 负责。在 2000 年初，DevDiv 为支持 .NET 计划进行了重组，其中的 C# 产品部门同时负责 JScript .NET 和 IE 中更传统的 JScript 引擎。这也包括了参加 ECMAScript 标准化活动的责任。但由于客户对 JScript .NET 的接受程度较弱，而且 Windows 团队对于增强 IE 的兴趣不大，因此与 JScript 和 ECMAScript 相关的工作，在 C# 团队中属于低优先级的事项。

在 2000 年代，微软通常将战略上重要的开发工作放在华盛顿州 Redmond 市的总部里，并经常将更多的战术项目分配到世界各地的其他分部中。在 2006 财年（2005 年 7 月至 2006 年 6 月）里，微软 DevDiv 决定将所有与 JScript 和 ECMAScript 相关工作的职责交接给位于印度 Hyderabad 市的印度研发中心（IDC）。DevDiv 之前已经将类似 Java 的 J# .NET 产品交接给了 IDC [Prasanna 2002]。到 2006 年春季，交接工作得以基本完成。在 TG1 上代表微软的任务，则交给了曾经在 J# 团队工作过，并参与过 Ecma C# 标准工作组 TC39-TG3 的 Pratap Lakshman。Lakshman 在 2006 年 4 月第一次远程参加了 TG1 会议，并开始参加电话会议和一些面对面的会议。但在此期间，他并不是新版 ES4 开发工作的重要贡献者。

本文作者之一 Allen Wirfs-Brock 于 2003 年加入微软，担任软件架构师，负责研究新 IDE 体系结构的探索性项目。在加入微软前，他已经

在 Smalltalk 编程语言和开发环境方面工作了二十多年。Wirfs-Brock 曾是首个商用 Smalltalk 虚拟机实现 [Caudill and Wirfs-Brock 1986] 的首席开发者。他致力于增强 Smalltalk 的特性以支持大型编程，设计了标准的 Smalltalk 异常处理系统，并编写了 ANSI Smalltalk 标准 [ANSI X3J20 1998] 中的语言定义部分。

到 2006 年底，IDE 项目似乎已经进入了正轨，Wirfs-Brock 也开始寻找新的机会。这时 DevDiv 内部对于动态语言的兴趣正在增加。由于还没有单独的 DevDiv 产品组负责动态语言，各个产品组的经理都抢着想揽下这份工作。Wirfs-Brock 当时担任资深架构师，向 Visual Basic 产品组经理 Julia Liuson 汇报，为她提供动态语言技术和机遇方面的建议。

Allen Wirfs-Brock 的新岗位从 2007 年 1 月的第一周起开始。在一次偶然的谈话中，Liuson 问他是否了解 JavaScript。Wirfs-Brock 回忆说他的回复大致是这样：「我不太了解，只知道这是一种用于网页的动态语言，我认为它与 Self 有一定关系。」随后 Liuson 将显示器转了过来，给他看了一封她刚刚收到的电子邮件，问他是否有什么想法。

这封邮件是 Pratap Lakshman 发给所有 DevDiv 产品组经理的，希望获知他对 Ecma TC39 正在开发的一个新 JavaScript 标准所应采取的立场。据 Wirfs-Brock 回忆，Lakshman 的信息说新的标准基于 Adobe Flash。与当时的浏览器相比，这将会是个实质性的变化。Lakshman 说，TC39 正在开发的是一门强大的语言，对 Web 来说可能太复杂了。他还列举了一长串新特性和变化，包括基于类的静态类型、结构化类型、参数化类型，以及方法重载。他还说，修订后的语言将通过 Standard ML 编写的参考实现来定义。

Allen Wirfs-Brock 回复 Julia Liuson 说，这听起来像是一次彻底的重新设计。根据他的经验，通过增加静态类型来改进动态语言的尝试很少成功。他对 JavaScript 或 Web 开发还不够了解，无法给出更确切的意见。不过，他提出要进一步研究一下。

Wirfs-Brock 花了几天时间来熟悉 JavaScript、当时的 ES3 规范以及公开 Wiki 快照中的 TG1 提案 [TC39 ES4 2007f]。他与 Lakshman、IE 团队的软件架构师和从事 Web 应用开发的微软工程师都进行了交谈。他意识到 JavaScript 在 Web 上发挥的作用，明显属于 Richard Gabriel [1990]「Worse Is Better」理念的实例。它最早只是个最低限度上的创

造，并以一种分散的方式成长，现在则已经深深地根植在了 Web 之中。相比之下，新版 ES4 的努力在 Wirfs-Brock 看来，则属于被 Gabriel 称为「做正确的事」的项目，不太可能获得成果。哪怕取得了成果，也会对 Web 造成很大的破坏。作为结论，他认为在技术上负责任的做法，是尝试让 ECMAScript 的演化重新回到增量演进的道路上。

鉴于微软当时对 Web 浏览器技术缺乏战略兴趣，Wirfs-Brock 认为 DevDiv 管理层不太可能有兴趣将资源分配给与 Web 浏览器相关的工作。他决定在向 DevDiv 内部公开时，需要关注新版 ES4 倘若成功所可能带来的后果。他确定的主要关注点是 Adobe 在 ActionScript 3 语言定义和虚拟机方面的贡献。DevDiv 特别关注的地方是 .NET 平台和 C# 旗舰语言，这些产品的主要客户是企业级应用的开发者。虽然 .NET 的主要竞争对手是 Sun 公司的 Java 平台，但 DevDiv 也开始将 Adobe 公司基于 ActionScript 的 Flash 和 Flex 产品视为 .NET 的竞争对手。Wirfs-Brock 预计新版 ES4 如果成功落地，可以将 ActionScript 转变为一线企业级语言，其功能和实用性可以与 C# 或 Java 相媲美。基于这一点，再加上 JavaScript 作为 Web 开发主要语言的标准化，可以推测出新语言可能对微软的语言和开发者产品造成严重的竞争威胁。

Allen Wirfs-Brock 写了一份备忘录，说明了这些担忧，并建议微软在 TG1 内部积极开展工作，试图将 TG1 重新引导到对 ECMAScript 标准增量、非破坏性演进的道路上。到 1 月中旬，这个建议获得采纳，Wirfs-Brock 则被授权执行该建议。2007 年 1 月 18 日，Pratap Lakshman 在 TG1 内部邮件列表上发布了一条消息 [TC39 2003]，介绍 Wirfs-Brock 为新的微软 TG1 代表。

3 月份的 TG1 面对面会议将由微软主办，Wirfs-Brock 决定在这次会议上首次参会。但他还觉得，必须尽快打消委员会对「微软支持新版 ES4 工作」态度的认识。他要求 Pratap Lakshman 在 2 月的会议上传达这一信息。Lakshman 照做了，并在 TG1 的内部 Wiki 上发布了一个页面 [Lakshman 2007a]，提出了一种描述简化的 ES4 浏览器模式（browser profile）的设想。他报告说自己收到的回应相当不友好，但在一次茶歇时间，Douglas Crockford 找到了他，提出雅虎愿意和微软一起反对新版 ES4。

Allen Wirfs-Brock 联系了 Douglas Crockford，他们同意一起合作制定微软 - 雅虎的联合提案，以替代新版 ES4 项目。Crockford [2002d] 此

前曾发布过一小套关于修改 ECMAScript 语言的建议，其目的是通过纠正原始设计中的错误和不便，从而使语言「更好一点」。Wirfs-Brock 和 Crockford 同意将这些建议作为联合提案在技术上的出发点。Pratap Lakshman 则提出了一份进行最小化修改的提案 [Lakshman 2007b]，其中能纳入 Crockford 所建议的许多 ES3 改动，相当于他对自己「浏览器模式」设想的后续行动。同时，Wirfs-Brock 与 Crockford 和 Lakshman 合作，起草了一份更正式的提案，并在微软和雅虎内部流传，以供内部批准。在 3 月 21 到 23 日的 TG1 会议前，他们于 2007 年 3 月 15 日发布了提案 [Crockford et al. 2007]。Crockford 将提案通过 TG1 的内部邮件列表进行了分发。

这份提案名为《关于重新聚焦 TC39-TG1 对 ECMAScript 第三版规范维护的提案》，其开篇段落如下：

> 我们认为，目前 TC39-TG1 正在开发的 ECMAScript 4 规范，与目前的标准完全不同，它本质上是一种新的语言。对于一个被广泛使用的标准化语言的修订版来说，这样剧烈的改动是不合适的。而且鉴于目前 ECMAScript 第三版在 AJAX 式 Web 应用中的广泛采用，这样的改动也是不合理的。我们认为，基于目前的语言设计工作，TC39-TG1 内部无法达成共识。然而，我们相信可以找到一个替代性的解决方案，并将这一提案作为可能的解决途径。

这份提案建议，TG1 应围绕三个工作项目进行重组。第一个工作项目是维护当时的 ECMAScript 语言，即由第三版规范定义的 ECMAScript 语言。维护工作将包括：澄清第三版规范中未明确的部分，纳入新特性（如 Mozilla 的 JavaScript 1.6 / 1.7 中的新特性），以及一些如 Crockford 所列举的小型修正和改进。第二个工作项目是为 ActionScript 起草一份标准定义。第三个工作项目是为浏览器定义一种新的编程语言，这门语言可以与 ECMAScript 共存，同时不受 ECMAScript 兼容性的限制。提案还提出了将工作项目二和三合并的可能性。它建议将这两者分配给一个新的 TC39 工作组，而不是 TG1 工作组。

正如预期的那样，TG1 内部邮件列表[69]上对此的反应普遍是负面的，但它确实显示出苹果的 Maciej Stachowiak [2007b] 也对新版 ES4 的发展方向持保留意见。Brendan Eich [2007b] 是最有分量的回应者，他为静态类型和其他新版 ES4 的特性辩护，认为这些特性对于增强性能和

大型应用的结构化至关重要。他还质疑了微软和雅虎提出这份提案的动机 [Eich 2007c]。

随着 3 月会议日期的临近，电子邮件上的讨论愈演愈烈。Pratap Lakshman 要求将会议第二天的大部分时间用于讨论微软和雅虎的提案。Brendan Eich 反驳说讨论一个小时就应该足够了。他和 Jeff Dyer 都表示希望将会议的大部分时间继续作为新版 ES4 的 hackathon。并且 Eich 和 Dyer 都认为，新版 ES4 的开发代表了微软帮助建立的 TG1 长期以来形成的共识，并质疑微软和雅虎现在试图打破这一共识是否合适。Allen Wirfs-Brock 对此回复说，现在共识已经被打破了，因为微软和雅虎是 Ecma 三个标准会员（Ordinary Member）中的两个，他们都经常参加 TG1。

3 月份会议第二天 [TC39-TG1 2007c] 的出席人数比平时多。除了 Allen Wirfs-Brock 和 Pratap Lakshman 外，微软的代表还有 Scott Isaacs 和 Chris Wilson。Isaacs 是微软「live.com」的框架架构师，还是 DHTML[70] 的初始开发者之一。Wilson 则是 Internet Explorer 的平台架构师，并积极参与 W3C Web 标准的制定。Isaacs 和 Douglas Crockford 都谈到了在浏览器 ECMAScript 实现的互操作性不佳的情况下，Web 应用开发上的困难。Crockford 认为，更完整的 ES3 级特性规范将有助于消除互操作性问题，从而提高 Web 的稳定性。Isaacs 特别关注的是，应当尽量减少新的语言语法扩展，因为这些新的扩展可能导致旧浏览器在执行新网页时出现解析错误。Isaacs 和 Crockford 都强调了 Web 应用中安全和隐私功能的重要性。对此 Eich、Dyer 和 Graydon Hoare 则反驳说，要想构建更稳定、更安全、更高性能的浏览器编程环境，新版 ES4 的类型系统是必需的基础。Wirfs-Brock 认为，进化后的「ES3.1」规范将有助于稳定 Web，并为 ES4 的实现和流行提供时间。Eich 担心这只是一种拖延策略，让微软有时间建立他们基于 .NET 的富互联网应用平台[71]，从而与基于标准的 HTML / CSS / JavaScript 平台进行竞争。他警告说，现在社区里已经有很多人对 ES4 充满了热情，如果微软和雅虎强行推迟开发 ES4，会给微软和雅虎带来负面影响。

最终，大家一致认为，开发「ES3.1」规范可能有一定的价值，微软和雅虎可以在 TG1 的背景下进行工作。这也就是 Wirfs-Brock 在筹备会议时所希望的结果。新版 ES4 的支持者坚持认为 ES3.1 必须是新版 ES4 的一个子集，其规范必须使用为新版 ES4 开发的规范风格。

Wirfs-Brock 对这些限制并不太担心，因为他仍然认为新版 ES4 规范不太可能完成并发布。

会后，Pratap Lakshman、Allen Wirfs-Brock 和 Douglas Crockford 开始着手定义 ES3.1 项目。Wirfs-Brock 和 Crockford 在 3 月 29 日举行了会议，并同意 Lakshman 应起草一份初步提案，在 4 月 TG1 会议前分发。Crockford 提出了一些设计准则，并建议 3.1 规范采用与 ES3 规范相同的风格。这与 3 月会议上达成的共识有冲突，但在新版 ES4 规范的最终形式尚未确定的情况下，使用相同的规范形式也是有问题的。

4 月 15 日，Pratap Lakshman 以《ES3.1 提案工作草案》[Lakshman et al. 2007] 为题，在 Wiki 上发布了一些页面。它列出了一系列目标、前后向兼容性要求，以及设计准则（图 27）。它还包括了大约 20 个修复、修改和新特性的描述，这些特性都是候选的，其中有许多来自于 Douglas Crockford 的《ECMAScript 修改建议》文档。他于 4 月初更新了这份文档，并在 ES3.1 开发过程中进行了两次更新 [Crockford 2007b, c, d]。

目标
1. 通过重写规范来提高实现的一致性，提高规范的严谨性和明确性，并纠正已知的模糊点或不够规范之处。
2. 在标准中添加已被实现和使用的常见扩展（特别是大多数 JavaScript 1.6 和 1.7 的特性）。
3. 纳入增量扩展，着重支持当前的使用经验和最佳实践。
4. 采用影响较小的语言修改，纠正已知的性能或可靠性问题。
5. 将确定有问题的特性标记为废弃。
6. 最大限度地提高 ES3 和 ES3.1 之间，以及 ES3.1 和 ES4 之间的前向和后向兼容性。

设计准则
1. 主要重点是纠正已知的错误和澄清已知的歧义。
2. 只有在以下情况下才考虑新特性：
   a. 不引入新的语法
   b. 提供重要的新价值
3. 倾向于现有实现方案中已被证明的特性。
4. 如果已有特性会造成重大的安全性或可靠性问题，则可能被标记为废弃。
   a. 考虑废弃那些会导致重大性能问题的低价值特性。

图 27. ES3.1 的初始目标和设计准则 [Lakshman et al. 2007]。

在 4 月的会议 [TC39-TG1 2007a] 上，委员会讨论了 ES3.1 工作草案。新版 ES4 开发者的主要关注点是 ES3.1 工作与新版 ES4 规范之间的关系。他们希望 ES3.1 工作遵循他们打算在新版 ES4 中使用的 ML 参考实现规范技术。ES3.1 小组反驳说，为一个规范的维护版本完全改变规范技术，其意义似乎并不大。Jeff Dyer 最后建议，鉴于观点的不同，ES3.1 的人应该继续他们手头的工作。但他也提出了警告，认为在 ES3 规范的背景下所做的工作，对小组的其他成员没有什么价值。

在 2007 年春夏之交的其余时间里，这两个小组基本上都持续在各自的项目上工作。ES3.1 小组分析了现有的 ES3 规范及其实现，以确定由于规范化程度不够或未能遵循规范而存在的互操作性问题 [Lakshman 2007c; Wirfs-Brock 2007b; Wirfs-Brock and Crockford 2007]。新版 ES4 小组则以其 ML 参考实现为工具，继续完善他们的各种提案。

新版 ES4 项目的时间安排仍然非常紧迫。在 2007 年 5 月初，一份提交给 Ecma 共同管理委员会的报告 [Miller 2007] 指出，新版 ES4 规范的最终草案将在 2007 年 10 月前完成，以便 Ecma GA 大会在 12 月批准它。2007 年 6 月 8 日，Dave Herman [2007; Appendix K] 在 *Lambda the Ultimate*[72] 博客上宣布 ES4 参考实现的「M0」版本[73]已经可用。

在 6 月的会议 [TC39-TG1 2007b] 上，有人呼吁立即启动 ES4 的规范编写进程。但当时仍有重大的技术设计问题未能解决，也经常发现新的问题。例如在 7 月的会议 [Eich 2007d] 上人们意识到，新版语言在对结构化类型做运行时类型检查时还有重大问题。

9 月 7 日的 TG1 召集人报告指出，想在 2007 年完成新版规范是不现实的，新的完成日期被推后一年至 2008 年 9 月。报告中还介绍 Lars Hansen 将担任新版 ES4 的编辑。这份报告既没有提到正在进行的 ES3.1 工作，也没有提到雅虎和微软对新版 ES4 的保留。

9 月会议 [TC39-TG1 2007d] 的目标之一，是接受、拒绝或推迟 ES4 Wiki 上所有未决的新版 ES4 提案。从新版 ES4 工作组的角度来看，这包括了被标记为「维护 ES3」的提案，这是 ES3.1 工作的总括性提案。Jeff Dyer 在会上的立场是，这份提案需要在当天被接受或拒绝（并在 Wiki 上标明）。如果被否决，该提案将不再作为 TG1 的工作项目。从会议记录中可以看出，他不认为提案有可能被接受。Brendan Eich 的立场则更为微妙。作为新版 ES4 的公开支持者，他认为 ES3.1 的努力让他分心，并非常怀疑微软的动机。他不希望 ES3.1 的开发与

新版 ES4 竞争，并建议 ES3.1 的支持者们考虑离开 TG1，看看 TC39 是否愿意为他们建立一个新的任务组。然而作为 TG1 的召集人，他希望找到一种避免组织分裂的方法。他建议，ES3.1 小组的工作成果可以作为 Ecma 技术报告来发表，或发表为其他一些不太正式、非 ISO、非标准轨道的文档。整个谈话过程非常激烈，对新版 ES4 和 ES3.1 的支持者来说都很紧张。Pratap Lakshman 一度沮丧地表示：「不论是全部还是部分，我们都既不支持也不同意目前的 ES4 提案。我们打算继续与有兴趣的任务组成员合作，制定一份对现行规范进行更多增量修订的提案。」尽管这不是个非常政治化的声明，但它也反映了微软的立场，只是在范围涉及全部新版 ES4 这一点上略有出入。最后「维护 ES3」提案状态的问题得以变通解决，方法是将关于 ES3.1 的页面从 Wiki 的「提案」命名空间下移到了新的「ES3.1」命名空间下。然而，ES3.1 和新版 ES4 的支持者们在目标上的冲突仍然存在，相关言论很快就公开化了 [Kanaracus 2007]。

## 寻求和谐§

2007 年期间，活跃的 TG1 参与者开始增加，这其中部分原因在于 ES3.1 和新版 ES4 小组都努力鼓励新成员和目前不活跃的成员参加会议。在当年春季，先前并不活跃的 TG1 成员 IBM 和苹果，也开始更经常地派代表参加 TG1 会议，并参与在线讨论。Google 作为标准会员加入了 Ecma，并任命 Waldemar Horwat 为其 GA 代表和 TG1 代表领导。Dojo 基金会作为非营利会员加入，由 Alex Russell 和 Chris Zyp 代表。Allen Wirfs-Brock 和 Douglas Crockford 都鼓励对象能力（OCAP）[Miller 2006] 语言[74]专家 Mark S. Miller 参与进来。Miller 曾在 Google 工作，他开始以 Google 代表的身份参加会议。一些新的与会者为小组带来了属于 Web 开发者的视角，在以前小组一直是由语言设计者和引擎实现者主导的。

2007 年初，TG1 的目标是在 10 月前完成新版 ES4 规范。这一目标没有实现，但 Lars Hansen [2007e] 在 10 月完成了一份文件，其初稿 [Hansen 2007b] 名为《ECMAScript 第四版语言概述》。这不是一份详

细的规范，而是对语言主要特性的 40 页总结。其摘要的第一段是这样描述新版 ES4 语言的：

> 第四版的 ECMAScript 语言（ES4）代表了 ECMA 在 1999 年批准的 ECMA-262 标准语言第三版（ES3）的重要演变。ES4 与 ES3 兼容，并增加了重要的设施，用于大型程序设计（类、接口、命名空间、包、程序单元、可选的类型注解，以及可选的静态类型检查和验证）、演化式编程和脚本编写（结构化类型、鸭子类型、类型定义和方法多重派发）、数据结构构建（参数化类型、getter / setter 和元级方法）、控制抽象（适当的尾调用、迭代器和生成器）以及类型自省（类型元对象和堆栈标记）。

最终证明，这份文档是对人们设想中新版 ES4 语言的最佳整体描述。然而，Allen Wirfs-Brock [2007c] 和 Douglas Crockford [2007a] 都对「ECMAScript 第四版」这一名称被不加限定地使用表示担心，这暗示了其所描述的语言已非常接近最终批准的 Ecma 标准。此外这份文档在导言中宣称，其整体设计代表了 Ecma TC39-TG1 的共识，并未提及任何 TG1 中对新版 ES4 在设计上的不同意见。在沟通后，Hansen 同意在文档标题前加上「拟议」字样，并在文档导言中插入了一段话，指出 TG1 中有少数成员对该设计的标准化表示反对。在新版 ES4 小组成员为分发概览文件和参考实现代码而建立的网站 [TC39 ES4 2007c] 上，人们也提出了类似的意见。这些事件增加了 ES3.1 支持者对新版 ES4 支持者们的担忧，担心他们继续公开宣传 ES4，同时继续无视或贬低 ES3.1 的开发。

Allen Wirfs-Brock 经常与微软的企业标准小组保持联系，其中包括 Ecma 共同协调委员会（CC）的成员 Isabelle Valet-Harper。协调委员会 [Ecma International 2007b] 关注的是，TG1 将外部托管的私有 Wiki 作为文档和会议记录的载体，Ecma 秘书处和一般会员无法访问它们。秘书处要求 TG1 将议程、会议记录和重要文档格式化，以便发表到对 Ecma 内部成员专用的网站上。TG1 决定，对此最简单的办法是将整个 TG1 的 Wiki 网站都设为公开可读 [TC39 2007]。

在 2007 年 10 月的 Ecma CC 会议 [Ecma International 2007a] 上，委员会讨论了 TC39-TG1 的运作问题。在 2001 年之前，TC39 的章程只涉及 ECMAScript。2001 年它进行了扩张，包含了更多的编程语言和平台，其中每种都由一个基本独立的 TG 任务组负责。ECMAScript 的开发工作则交由 TC39-TG1 负责。Ecma 秘书处一般侧重于监督和支持

TC 技术委员会一级的活动，而非监督 TG 工作组。在 2007 年，TG1 工作组的独立运作似乎缺乏 TC39 或秘书处的监督。一些协调委员会成员担心，TG1 可能没有完全遵守 Ecma 的政策和程序。会议还讨论了 TG1 工作组内部据称对其当时的工作缺乏共识的问题。人们讨论了一种可能的解决办法，那就是将 TC39-TG1 升格为正式的 TC 技术委员会，这样它将得到秘书处更大的监督。时任 Ecma 主席 John Neumann 同意出席 2007 年 11 月的 TG1 会议，试图使情况明朗化。

这次会议 [TC39 2007] 主要用于表达协调委员会的关切，并讨论了 ES3.1 和新版 ES4 项目之间显著缺乏共识的问题。John Neumann 强调了他对 TG1 在向 Ecma 传达会议通知、议程、会议记录和关键文档时缺乏沟通的关切，并坚持认为这种情况需要改变。他还提出了警告，认为从 Ecma 委员会的角度来看，TG1 在某些情况下过于公开。特别是 Ecma 管理部门内也有人担心，TG1 成员之间的分歧会在网络博客和论坛上公开争论。Neumann 宣布，他将建议把与 ECMAScript 相关的活动再次作为 TC39 的唯一关注点。总体上看来，TC39-TG1 将被重命名为 TC39。这将使与 ECMAScript 相关的工作在 Ecma 内部得到更多的关注，并使 Ecma 秘书处能直接为其提供支持和监督。TC39 内其他正在开展工作的任务组，则将转入新成立的 TC49 任务组。这一改组在 2007 年 12 月的 Ecma GA 大会上获得批准。自 2008 年 1 月起，TC39-TG1 再次成为了 TC39。

11 月的会议还讨论了 TC39 的后续章程。Douglas Crockford 提议，应该有个新项目来定义一种*安全的 ECMAScript*[g]（SES），以支持 *mashup*[g] 和其他注重安全性的应用。Allen Wirfs-Brock [2007] 则发布了一份新的《微软立场声明》，重申了微软的呼吁，即采取循序渐进的方式推进 ES3 语言和规范的发展，而非继续现有的新版 ES4 工作。Crockford 宣布雅虎支持这一立场。Lars Hansen 断言称「3.1 的提案一直停滞，最终在 9 月份被搁置，我们在这里致力于 ES4 而非 3.1」。Brendan Eich 也认为自 4 月以来 ES3.1 都没有多少进展。Wirfs-Brock 不接受 ES3.1 被搁置的说法，指出已有多份分析 ES3 互操作性问题的文档 [Lakshman 2007c; Wirfs-Brock 2007a, b; Wirfs-Brock and Crockford 2007]，它们都属于对 ES3.1 开发的投入。

为了评估对 TC39 三项可能的开发活动的兴趣，委员会进行了一次投票调查。与会的所有人（代表九个组织）都支持继续开展新版 ES4 的工作。继续开发 ES3.1 的工作得到了微软、雅虎、苹果、谷歌和

Mozilla 的支持。安全 ECMAScript 的启动工作得到了微软、雅虎、苹果和谷歌的支持。从 Ecma 的角度来看，这一支持度已足以证明在新的 TC39 中推进这些活动是合理的。不过微软也支持新版 ES4，这并不符合其立场文件中的说法。Allen Wirfs-Brock 回忆说，他认为不必在这一点上更进一步，因为他仍然预测对新版 ES4 的努力最后不会成功。

在 2007 年 12 月的 Ecma GA 大会后，Isabelle Valet-Harper 与 Allen Wirfs-Brock 讨论了谁可能是新 TC39 的合适主席。Brendan Eich 无法担任主席，因为 Ecma 当时的规则要求 TC 主席必须是一名来自标准会员的代表，而 Mozilla 则是一个非营利性会员。Wirfs-Brock 和 Valet-Harper 达成了一致，认为理想的主席应该是对 ES4、ES3.1 或任何其他可能的 TC39 项目没有利益相关或意见的人。Valet-Harper 建议微软和 Adobe 本着合作的精神，分别与 John Neumann 签约，由 John Neumann 代表他们，并共同提名他担任 TC39 主席。这一想法获得了 Adobe 的同意，并在 2008 年 1 月的 TC39 会议上得以宣布。在 2008 年 3 月的会议上，Neumann 被正式选为 TC39 主席。

2007 年 11 月，Lars Hansen [2007c] 编写了一份《编辑报告》，并提出了新的时间表，目标是在 2008 年 10 月前完成新版 ES4 的最终草案，并在 2008 年 12 月将其作为 Ecma 标准发布。他还写了一篇论文 [Hansen 2007a]，总结了新版 ES4 与 ES3 的有意不兼容之处，并写了一篇关于如何用新版 ES4 的渐进式类型支持演化式编程的教程 [Hansen 2007d]。2008 年 2 月，Jeff Dyer [2008a] 发布了一份新的工作计划，目标仍然定在 12 月，而中间的草案将在 5 月、7 月和 9 月完成。Hansen 和 Dyer [2008] 还发布了一份题为《在拟议的 ECMAScript 4 中将推迟的特性》的立场声明。声明中认为，当时的新版 ES4 计划包括了一些「奇怪的、未经证实的或代价高昂的」特性。而推迟实现这些特性：

> 将会大大增加在 2008 年完成规范的可能性，增加社区的参与度，有助于保持实现的复杂性可控，降低标准化的风险，并在一定程度上减少 TG1 成员之间的分歧。

声明中建议推迟的特性包括：数值转换、int 和 uint、十进制小数运算、运算符重载、泛型函数、`wrap`[75]、堆栈标记、生成器、尾调用、nullability、程序单位、重构的 `with`、修订后的 `eval` 和命名空间过滤

器。在说明了这些特性被推迟的原因之后，声明提出了 Adobe 对 ECMAScript 未来发展的修订后意见：

> 我们认为 ES 相比于我们目前所看到的 ES4，应该更多地以一种零散的方式发展。从 E262-3 的发布到 E262-4 的发布已经过去了 9 年，这本身并不是一次性引入大量新特性的有效理由。每个特性都必须有其重要性，而且必须基于经验来指导我们。即便如此，本文并不主张接纳一个被注水的「ES3.1」（其实应该叫「ES3.01」）。我们主张现在就采用 80% 完成度的解决方案「ES3.8」，然后计划在不久的将来，当这些需求更明确的时候，再发展到满足新的需求。

在 TC39 的会议记录中，或者在 TC39 的内部或公开的电子邮件渠道中，都没有关于这份立场文件的实质性讨论记录。唯一有记录的回应是 IBM 对于排除十进制小数运算的建议表示反对。在同一时期，还出现了大量对新版 ES4 在设计、方法论和开发进程等各方面的批评意见。这些批评意见被发布到了 `es4-discuss`[g] 邮件列表中，其中一些批评来自于有影响力的框架开发者，以及苹果和谷歌的 ECMAScript 实现者。2008 年 3 月，新版 ES4 的设计者们发现 [Dyer 2008b] 用于定义模块的新版 ES4 包抽象存在着根本性的语义问题，在 5 月还发现了命名空间的问题 [Stachowiak 2008b]。

在 2008 年的春天，Lars Hansen 发布了新版 ES4 个别规范部分的初稿以征求反馈意见。5 月 16 日，Hansen [2008] 公布了他的规范初稿 [Hansen et al. 2008a, b, c]。

> 随本文附上的是拟议的 ECMAScript 第四版规范相当不完整的初稿。这份草案包括一个简短的介绍，语言的表层语法，以及对核心语义的描述——包括值、存储、类型、名称、作用域和名称解析。更多的内容将在准备就绪后陆续发布，可能（或多或少地）每两个月发布一次。

在同一时期，ES3.1 子组开始开发一个由 ES3 规范衍生的规范。来自 Google、IBM、Dojo 基金会和苹果等组织的参与者不断扩大。ES3.1 规范的初稿于 5 月 28 日发布 [Lakshman 2008; Lakshman et al. 2008]。

在 2008 年 5 月 29 日到 30 日的会议上，这两份规范都由其编辑进行了介绍。详细的讨论时间被推迟到了 7 月份的会议上，以使成员们有时

间阅读规范。从进展速度、剩余的写作量以及尚未解决的设计问题的数量来看，新版 ES4 的最终规范明显不可能在 2008 年 12 月前完成，其发布时间更可能是 2009 年 6 月或 2009 年 12 月。而对 ES3.1 来说，为了使其能在 2008 年 12 月前完成，所有主要的设计决策都需要在 2008 年 7 月会议前完成。将其发布时间定于 2009 年 6 月似乎是个较为现实的目标。

2008 年 6 月底，John Neumann 组织了一次电话会议，与会者包括 Brendan Eich、Allen Wirfs-Brock、Douglas Crockford、Adobe 的 Dan Smith[76]，以及 Adobe 的 Ecma GA 代表 David McAllister。McAllister 和 Smith 宣布，Adobe 将停止对新版 ES4 的支持，分配给它的工作人员将转到其他事项中去。在场的每个人都明白，这标志着新版 ES4 已经告终，下面应该为此仔细安排一份更为公开的声明。他们同意在 7 月份即将召开的 的 TC39 会议上向全体 TC39 成员宣读这一决定，并在会上决定如何发表公开声明。而对 Eich 来说，Adobe 已经事先告知了他这一决定。他对此表示同意，并希望 TC39 的所有成员都能围绕着完成 ES3.1 的工作，制定出一个不受过去 ES4 设计决定限制的共同计划。他也同意在即将举行的会议上提出这一设想。接下来的会议将于 7 月 23 日至 25 日在挪威奥斯陆举行。这次会议的议程被修订 [TC39 2008f]，将「ECMAScript 的和谐化（Harmonization ）」列为了第一个新议题。

在 2018 年的电子邮件讨论中，Jeff Dyer 和 Lars Hansen 回忆说，退出是他们与经理 Dan Smith 协商后做出的决定。他们已经确信，新版 ES4 不太可能完成。他们认为，ES3.1 工作组成员的反对意见使得新版 ES4 的工作停滞不前，而且很明显，在 ES3 现状基础上进行修复的方法在 TC39 中已经成为了主流，已经不再有空间来整合 ActionScript 3 的静态特性了。

Cormac Flanagan 在 2019 年的个人交流中推测，Adobe 的退出确实表明了新版 ES4 存在的问题。他的事后感想还包括以下几点：

- 为 ES4 计划的大量语言扩展（回头看来）是种高风险、非保守的做法。
- 在标准化进程中，主要由于静态类型系统的加入，标准涉及到了最前沿的语言技术（十多年后的 2019 年，这里仍然有一些未解决的研究和性能问题 [Greenman et al. 2019]）。在 TFP 07 上发表的《空间高效的渐进式类型》论文 [Herman et al. 2011] 受到了

ES4 中的性能问题的启发，这或许也体现了 ES4 工作的研究性质。
- 在 TC39 中，围绕 ES4 的「是否买入」担忧虽然一直是个问题，但从来都不是致命的。
- 尽管在后来的版本中被废弃，但基于 ML 的参考规范是个可行的设想。现在回想起来，也许从参考规范开始实现 ES3 会更好。

Douglas Crockford [2008c] 在一篇博文中，则将新版 ES4 的失败归咎于「未经证实的过度创新」：

> 事实证明，标准机构不是创新的好地方。这是属于实验室和初创公司的领域。标准必须通过共识来起草。标准必须是没有争议的。如果一个特性过于模糊，以至于无法形成共识，那么它就不应该成为标准化的备选项。正由于这个原因，「委员会设计」是个贬义词。标准化机构不应该参与到设计的工作中来。他们应该坚持认真制定规范，这也是一项重要而艰巨的工作。

Allen Wirfs-Brock 回忆说，当 Adobe 宣布退出新版 ES4 时，他感到松了一口气。他知道，微软负责 IE 浏览器的高管们已经意识到，放弃对 IE 浏览器的投资是个战略错误。当时 IE 的市场份额正大量流失到 Firefox 浏览器，而且高管们也知道谷歌正准备推出新的浏览器。微软正在主动面对 Web 开发者中所流传的「微软反对 Web 技术进步」的观点。微软对新版 ES4 的反对（特别是经由 Brendan Eich 和微软 Chris Wilson 的公开争论所暴露出的事实）也更加印证了这种观点。到 2008 年 6 月，Wirfs-Brock 担心微软可能会因为纯粹的商业原因而改变决定：与其公开反对，不如顺其自然地支持新版 ES4。

在奥斯陆举行的 TC39 会议 [TC39 2008g] 上，委员会的大部分时间都在围绕一套共同可实现的目标来解释「和谐化 TC39」的概念。总体计划是这样的：整个委员会的重点是在 2009 年完成 ES3.1 版本的发布，同时合作规划一个更重要的后续版本，即代号「和谐」（Harmony）的项目，它不受十年来 ES4 设计决策的限制。会议上讨论了一些特性算或不算「和谐的」，但在会上或会后与未参会的 TC39 成员的邮件讨论中，都没有人对基本计划提出严重的反对意见。会后编写的一份白皮书 [Eich et al. 2008] 总结了实施这一计划的步骤：

1. 在各方的充分合作下，将工作重点放在 ES3.1 上，目标是在明年初实现两个具备互操作性的实现。

2. 在 ES3.1 以外的下一步工作上进行合作，这将包括在语义和句法创新方面比目前 ES4 的建议更适度的语法扩展。
3. 删除 ES4 中的「包」、「命名空间」和「早期绑定」等概念。
4. 改写 ES4 中的其他目标和想法，以保持委员会的共识。这其中包括了「类」概念，它应当根据现有的 ES3 概念结合拟议的 ES3.1 扩展来实现。

8 月 13 日，Brendan Eich [2008b; Appendix M] 通过电子邮件向 `es4-discuss` 邮件列表发送了一份略为个性化的白皮书。8 月 19 日，Ecma 国际 [2008] 发布了一份简短的新闻稿，宣布 TC39 将把工作重点放在 ES3.1 上。8 月 15 日，Eich 录制了一个播客 [Openweb 2008]。他在其中解释了自己在技术层面和现实层面上对新版 ES4 失败的看法，以及他对 TC39 内部「和谐的未来」的希望。在播客开始不久，他说「通过命名空间来统一早期绑定和延迟绑定的尝试已经失败了。」后来他做了详细的说明：

> 首先我们把 ES4 的包给砍掉了，这是我们砍的。然后我们把 ES4 的命名空间也砍掉了，这也是我们砍的。我们这样做不是为了讨好 3.1。我们这样做是因为命名空间的问题。
>
> ……
>
> 这并不是让步，也不是对立斗争——这（新版 ES4）确实是个很好的尝试，它试着把事情统一起来，回到 Waldemar Horwat 的规范（也许甚至是 Common Lisp）去尝试命名空间和包，然后认识到它们不适合 Web。

# 插曲：认真对待 JavaScript§

从 20 世纪 90 年代末开始，TC39 成员试图将 JavaScript 作为一种面向专业程序员的语言进行重新设计。到 2000 年代末，浏览器和其他相关平台的开发者们终于意识到，JavaScript 是他们平台中需要认真对待的工程部分。

# JavaScript 性能革命§

当 Brendan Eich 在 1995 年 5 月构建 Mocha 时，性能既不是个关注点，也不是个目标。当时还没有任何 JavaScript 程序，为其预期的程序只要能对「基于其他更高效的语言实现的对象」做简单组合就够了。在当时的设想中，JavaScript 并不是用来编写哪怕稍复杂点的算法的。早期的 JavaScript 引擎使用简单的字节码解释器或解析树求值器来直接解释 JavaScript 函数，并使用简单的内存管理方案。它们没有利用 20 世纪 80 年代和 90 年代初为 Lisp、Smalltalk、Self 和其他动态语言开发的复杂的高性能实现技术。对 Netscape / Mozilla 的 SpiderMonkey 和微软的 JScript 引擎而言，其基本架构在十年来基本没有变化。在十年间，新的 ES3 级语言特性得以加入，安全问题也得到了解决。但无论这期间有什么性能提升，它们都可以归功于摩尔定律 [Moore 1975] 下的硬件性能进步。在这一时期的大部分时间里，维护浏览器的 JavaScript 引擎不过是一位软件开发者的兼职工作。

在 2000 年代的前半段，AJAX 式大型 Web 应用的出现，开始严重突破了第一代引擎的性能限制。到 2006 与 2007 年，Web 开发者对性能问题的呼声越来越高，浏览器厂商也开始派出团队来解决其 JavaScript 引擎的性能限制。对性能的度量是提高性能的重要起点，苹果的 *WebKit*[g] 团队为此创建了 SunSpider JavaScript 基准测试套件（benchmark suite）[Stachowiak 2007a]。SunSpider 远非完美，由相对较小的测试用例组成，但它来自于实际的 Web 应用代码。在它发布后不久，Web 应用开发者社区就开始经常使用 SunSpider 来比较浏览器的 JavaScript 性能，并对结果进行讨论。浏览器博弈论基本上阻止了浏览器厂商以 JavaScript 特性为基础进行竞争，但他们「可以并且确实」开始在 JavaScript 性能上进行竞争。

不同厂商采取了不同的路线来实现高性能的 JavaScript 引擎。2006 年，Google 开始研发一款最终成为了 *Chrome*[g] 的浏览器。Lars Bak 领导了 Chrome 的 *V8*[g] JavaScript 引擎的开发。这个引擎建立在他所开发的 Smalltalk、Self 和 Java 虚拟机 [Google 2008b] 的技术基础上。当 Chrome 浏览器于 2008 年 9 月发布时，它成为了代表良好 JavaScript 性能的新基准。同时期的一份报告 [Hobbs 2008] 显示，在当时的 Firefox 版本中，V8 运行 Google 基准测试 [Google 2008a] 的速度比当

时发布的 SpiderMonkey 快约 10 倍[77]。然而在 SunSpider 基准测试中，V8 大约只快了 2 倍。

Mozilla 最初使用的路线称为 TraceMonkey [Gal et al. 2009]，这是基于加州大学欧文分校的 Andreas Gal 博士毕业作品的引擎。它使用了现有的 SpiderMonkey 解释器，并在其基础上增加了一个跟踪驱动的 JIT 编译器。这个编译器能动态识别出执行热点，并为其生成优化后的原生代码。苹果的 SquirrelFish Extreme [Stachowiak 2008a]（也就是 Nitro）则使用了受 Self 和高性能 Lua 实现启发的技术。微软最初试图逐步重新设计其遗留的 JScript 引擎，以便在 IE8 中使用。但在 IE9 中微软推出了 Chakra，这是一个基于 JIT 的全新 JavaScript 引擎 [Niyogi 2010]。

所有这些努力仅仅是优化 JavaScript 性能的起点。今天，每一个主流浏览器的开发都需要一个实质性的 JavaScript 团队，专注于性能、安全性和 ECMAScript 标准的新语言特性。这些团队所开发的每个引擎都是在兼容的开源许可下发布的。因此这些团队能在彼此的工作基础上分享想法，有时还可以分享完整的子系统。最快的 JavaScript 实现就是这样在他们的竞争中产生的。

## CommonJS 和 Node.js§

从诞生之初起，JavaScript 也会部署在服务端平台上，提供基本的脚本功能。然而每个平台都有所不同，提供了自己特有的 JavaScript API。在 JavaScript 诞生的前 15 年里，并没有一个通用的、领域独立的、可互操作的非浏览器 JavaScript 应用环境。2009 年 1 月，曾在 Adobe 和 Mozilla 工作过的 Khan Academy 开发者 Kevin Dangoor 决定改变这种状况。他写了一篇博客文章 [Dangoor 2009] 描述了这些问题，并邀请服务端 JavaScript 社区通过在线讨论组和 Wiki 参与到解决问题中来。一年后，他在一篇后续的博文 [Dangoor 2010] 中，将自己最初希望创造的东西总结为如下：

- 一个模块系统（module system）
- 一个跨解释器的标准库

- 若干个标准接口
- 一个包系统（package system）
- 一个包仓库（package repository）

在一周内，有 224 名成员加入了讨论组 [Kowal 2009a]，其中许多人表示有兴趣为这个项目做出贡献。这个提议最初被称为 ServerJS，但在 2009 年 8 月更名为 *CommonJS*[g]，因为这一技术的适用性已经超出了服务端的范畴。提议的重点在于编写规范，而非实现。

到 2009 年 4 月，小组获得了一份初步的模块规范 [CommonJS Project 2009]。这个 CommonJS 模块规范主要基于 Kris Kowal 和 Ihab Awad 的设计 [2009a]。一个 CommonJS 模块就是一个 JavaScript 函数体，其作用域内包括了多个变量绑定，这些绑定使得函数体内的代码能与其他模块进行交互。这一能力是由一个同步模块加载器实现的。模块加载器会获取模块的源码，用一个骨架函数定义包裹它们，接着解析并调用合成函数（synthesized function）来初始化该模块，并初始化它到其他模块的连接。如图 28 所示，模块级的作用域声明会成为合成函数的局部变量。模块系统的控制钩子则作为函数参数暴露出来，其值由加载器提供。`require` 参数是一个函数，它同步地对所请求的模块执行上下文加载过程，并返回其 `exports` 的值[78]。默认情况下，`exports` 的值是一个由加载器提供的对象。在 CommonJS 中，从模块的名称，到实际的 `exports` 值，再到模块所导出属性的名称和值，都可以被动态生成。这使得想要预先获知程序「需要哪些模块，以及有哪些实体在这些模块之间共享」变得困难，有时这甚至是无法实现的。

```
// moda.js - 源码
var modp = require("modp");
exports.n = modp.p++;
exports.modName = "prefix" + exports.n;


// modb.js - 源码
var modx = require(require("moda").modName);
var propName = Object.keys(modx)[0];
exports[propName] = modx[propName];


// moda.js - CJS 展开后
(function (exports, require, module) {
  var modp = require("modp");
  exports.n = modp.p++;
```

```
  exports.modName = "prefix" + exports.n;
});

// modb.js - CJS 展开后
(function (exports, require, module) {
  var modx = require(require("moda").modName);
  var propName = Object.keys(modx)[0];
  exports[propName] = modx[propName];
});
```

图 28. CommonJS 模块被模块加载器转换成了「实现模块模式的函数」。模块之间的共享，是通过动态构造出的 `exports` 对象上的属性来实现的。

CommonJS 模块的早期使用者之一，就是 2009 年初由 Ryan Dahl 开发的 *Node.js*[g]。在其设想中，Node.js 是个用于通过 JavaScript 构建服务端应用的开源平台，其能力足以处理大量的客户端同时连接。Node.js 支持一种异步的 I/O 模型，并为此提供了一个带有库的 JavaScript 编程环境。它连接起了常见的 POSIX 接口、JavaScript 回调，以及简化的浏览器事件循环，其整体实现主要包含了谷歌的 V8 JavaScript 引擎、一个 CommonJS 模块加载器，以及一组 C 语言实现的模块。这些模块提供了许多平台接口的非阻塞版本，包括 POSIX API 和其他高层面的文件和网络操作。Node.js 的首个公开版本是在 2009 年 5 月发布的 [Node Project 2009]。但直到 2009 年 11 月 Dahl [2009] 在 jsconf.eu 上做了一次演讲后，它才引起了人们的重视。此后不久，Dahl 被 Joyent 雇用。Joyent 负责管理和支持 Node.js 的进一步开发，直到 2015 年将其交接给 Node.js 基金会为止 [Node Foundation 2018]。

Node.js 最早被设想为一种用于构建服务端应用的技术。但它已经成为了一个平台，使 JavaScript 能作为通用编程语言，应用在包括小型嵌入式设备在内的各种平台上。Node.js 的 I/O 模块与高性能的 V8 引擎相结合，在能力上足以与 Python 和 Ruby 等其他动态应用语言相媲美，在性能上也往往更胜一筹，成为了编写命令行 JavaScript 应用时的事实标准。Node.js 使掌握了 JavaScript 的 Web 程序员能将其技能转移到其他类型的应用和非浏览器环境中。最初许多客户端 Web 应用的开发者们之所以使用 JavaScript 编程，是因为他们别无选择。而许多 Node.js 开发者选择使用它，反而是因为他们更喜欢用 JavaScript 编程。

# 成为浏览器通用运行时的 JavaScript§

JavaScript 这门语言属于一系列 Web 标准套件中的一部分，这些标准定义出了可互操作的浏览器平台。它是仅有的一门网页开发者们可以预期在每个浏览器中都能使用[79]的语言。如 Java、Adobe Flash 和微软 *Silverlight*[g] 等其他语言环境，都不属于这个标准平台的一部分，必须使用特定于浏览器的扩展机制来集成到浏览器中——前提是这门语言支持这个浏览器。通常情况下，语言引擎必须由浏览器用户单独安装，并且可能无法完全集成到浏览器的标准服务中，比如基于 DOM 的图形模型。

浏览器博弈论预测，任何「通过增加另一种编程语言来扩展标准浏览器平台」的尝试，其成功的可能性都极低。浏览器厂商需要大量投入来设计、实现和推广一种新的 Web 语言，却不能保证它能在 Web 开发者中流行。要想让这门语言获得接纳，需要所有的主流浏览器都同意支持一种由竞争对手设计，且用户群很小甚至根本不存在的语言。而且这种语言还将成为长期的维护负担。例如在 2011 年，Google 推出了 *Dart*[g] 语言，将其作为一种更好的 Web 编程语言进行推广 [Krill 2011]。Google 分发了一个实验性的 *Chromium*[g] 版本 [Google 2012a]（这是他们的 Chrome 浏览器的开源基础），其中包含了一个 Dart 虚拟机 [Google 2012b]，但这个虚拟机从未被纳入 Chrome 浏览器的生产版本或任何其他浏览器中。

随着 2005 年 AJAX 和 Web 2.0 风格应用的出现，Web 开发者开始编写更大型、更复杂的 Web 应用，其中有些人开始寻找一种能比 ES3 级的 JavaScript 更适合此类应用的编程语言。如果开发者需要编写的代码可以作为网页的一部分在任何浏览器中运行，而他们又需要或想要用 JavaScript 以外的语言编写代码，该怎么办呢？唯一的选择是使用 JavaScript 作为运行时支持的替代语言。这可能可以通过用 JavaScript 为替代语言编写一个解释器来实现。但在 2000 年代中期，JavaScript 引擎仍然是以相对较慢的解释器来实现的，而且 JavaScript 对编写高效解释器而言并不是种很好的语言。实现一个慢上加慢的解释器，显然并非有足够吸引力的解决方案。一种更可行的方法是通过「源对源翻译器」来承载替代语言，即经由编译器，将替代语言的源代码翻译成「可在浏览器 JavaScript 引擎上运行的 JavaScript 代码」。如果替代

语言的语义和 JavaScript 的语义之间有合理而接近的匹配，那么用这种方式编译出的程序，其运行时性能就可以相对接近于手写的 JavaScript。

2006 年 5 月公开发布的 Google Web Toolkit（GWT）[Google 2006]，是首个使用源到源翻译的 AJAX 工具套件。GWT 集成了 Java 到 JavaScript 的编译器，它被成功地应用于一些 Google 重要的对外 Web 应用，并在 Google 以外也得到了大量使用。GWT 的成功，证明了将 JavaScript 用于源对源翻译的可行性，许多其他语言的翻译器也随之而来。有一份文档记录了「可编译到 JS」的语言列表。它在 2011 年 1 月共有 19 个条目 [Ashkenas et al. 2011]，但到 2018 年则包括了 270 多种语言 [Ashkenas 2018]。这些语言要么被翻译到 JavaScript，要么以 JavaScript 为宿主。这些语言中有一些是玩具级或不完整的实现，然而其中也有许多是拥有大量用户的严肃编译器，甚至还有一个针对 JavaScript 的 Dart 编译器。

源对源翻译不仅被用来支持 Web 页面上的遗留语言，还为实验新语言和扩展 JavaScript 提供了一种手段。*CoffeeScript*[g] [Ashkenas 2010] 是最成功的源对源翻译器之一，它由 Jeremy Ashkenas 在 2009 年到 2010 年开发。在成为 Web 开发者前，Ashkenas 曾用 Ruby 语言编程，他更喜欢 Ruby 相对无标点符号的语法和 Python 式的留白缩进，而非 JavaScript 使用的 C 风格语法。他创建了 CoffeeScript 作为 JavaScript 的新表层语法，同时保留了 JavaScript 的底层运行时语义。Ashkenas [2009] 这样描述他在 CoffeeScript 上的工作：

> 长期以来，JavaScript 都把一个漂亮的对象模型隐藏在了 Java 式的语法中。CoffeeScript 试图通过偏重表达式而非语句的语法，减少标点符号的噪音，提供优雅的函数字面量，从而展示出 JavaScript 好的部分。像 `square: x => x * x` 这行 CoffeeScript，就可以编译成这样的 JavaScript：

```
var square = function (x) {
  return x * x;
};
```

除了「漂亮的函数」以外，CoffeeScript 还引入了许多提高编程便利性的语法糖，包括类声明和解构操作。这些内容都很容易转为 JavaScript 代码。许多 CoffeeScript 的特性与 ECMAScript Harmony 所考虑的特性

类似。CoffeeScript 验证了 JavaScript 程序员对这些特性的兴趣。CoffeeScript 很快就变得相当流行，并被许多主要网站的开发者所采用。但在 ES2015 普及后，它的使用量就逐渐减少了。

在 2011 年 5 月的 JSConf 大会上，Brendan Eich 与 Jeremy Ashkenas 一起分享了 CoffeeScript 及其在 JavaScript 的 Harmony 演化中的作用。在他的演讲中，Eich [2011c] 介绍了一个名为「转译器[g]」的术语，用来描述像 CoffeeScript 这样的源对源编译器。这并非「转译器」一词首次出现，但在 Eich 的演讲前，这个词并未被广泛地了解和使用。后来，这个概念开始在 JavaScript 开发者社区内外被普遍使用。

Alon Zakai [2011] 的 Emscripten 是一个能将 C/C++ 翻译成高效 JavaScript 代码的转译器。它的诞生前提，在于作者发现通过 JavaScript 的 32 位算术编码模式和二进制的 TypedArray 数据结构，可以定义出一个易于被基于 JIT 的 JavaScript 引擎优化的 C 语言执行环境。Emscripten 启发了 asm.js [Herman et al. 2014]，这是个定义了一组 JavaScript 代码模式的规范。相应编译器所生成的符合规范的 JavaScript 代码，都应该能被引擎识别和优化。asm.js 的成功进一步带来了 WebAssembly [Haas et al. 2017]，它以字节码级接口扩展了 JS 引擎，可以作为 C/C++ 和类似的低级语言的编译目标。

# 继往开来 · JavaScript 二十年


## JavaScript 二十年


- 


- JavaScript 二十年


- 
- 
- 



## 继往开来§

# 开发 ES3.1/ES5§

在 2007 年的大多数时间里，新版 ES4 工作组都认为对 ES3.1 的投入不过是企图阻挠新版 ES4 的竞争，其中并没有实质性的技术。但是，Douglas Crockford、Pratap Lakshman 和 Allen Wirfs-Brock 仍然致力于对 ES3 规范进行增量修改，从而保证规范与时俱进，并修复各种导致互操作性问题的隐患。在发布 ES3.1 的初始目标和设计原则，并提出语言特性级的改动 [Lakshman et al. 2007] 后，他们走出了工作的第一步，即全面了解当时 Web 浏览器中 JavaScript 的现状，以及真实的 Web 与 ES3 规范之间有何不同之处。

ES3.1 工作组有一个直接的关注点，那就是微软为 Internet Explorer 实现的 JScript 以不符合 Web 标准而闻名。为了验证这些 ECMAScript 相关问题的有效性与影响范围，Allen Wirfs-Brock 请 Pratap Lakshman 进行分析，以确定 IE 的 JScript 与 ES3 规范之间一共有哪些出入。这次分析于 2007 年 9 月完成，其成果是一份长达 87 页的报告，名为《JScript 到 ES3 的偏差》[Lakshman 2007c]。这份报告分为三个主要部分。在第一部分里，报告逐个确定了「当时的 JScript 实现」与「ES3 规范的明确要求」之间存在偏差的具体位置。对每个偏差，报告都提供了 ES3 中相应被违反之处的描述、用于观察偏差的测试用例，以及在当时最新版的 IE、Mozilla Firefox、Opera 和苹果 Safari 上执行测试的结果。这些浏览器是当时公认的「前四大」浏览器。如图 29 中的示例，就展示了一个被确定出的此类偏差。其中有些偏差为 IE 所特有，有些偏差在所有受测浏览器中均存在，还有些偏差在 IE 和其他若干浏览器中存在。

2.15 String.prototype.split: §15.4.4.14

ES3 陈述为「如果分隔符是一个包含捕获小括号的正则表达式，那么每次匹配分隔符时，捕获小括号的结果（包括任何未定义的结果在内）都会被拼接到输出数组中。」

JScript 忽略了捕获小括号。FF 输出了空字符串而不是 undefined。

示例

```
<script>
alert("A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?
```

```
([^<>]+)>/));
</script>
```

输出

```
IE: A,bold,and,coded
FF: A,,B,bold,/,B,and,,CODE,coded,/,CODE,
Opera: 和 FF 相同
Safari: 和 FF 相同
```

图 29. 一个记录在 JScript 偏差报告中的 ES3 偏差 [Lakshman 2007c]。

偏差报告的第二个主要部分，确定了所有在 ES3 规范中被明确定义为「行为依赖于实现」或定义不够充分之处。这部分也提供了测试用例，以及在四个主流浏览器上执行测试的结果。报告的最后一部分则描述了 IE 中实现的各类属于 ES3 规范扩展的特性。Wirfs-Brock [2007b] 还准备了一份列表，记录了 Firefox 中实现的 ES3 扩展。在 2007 年 8 月 16 日的会议上，Douglas Crockford 和 Allen Wirfs-Brock 讨论了这些文档的草案，相应产物是一系列 ES3.1 规范中的试验性变更 [Wirfs-Brock and Crockford 2007]。

ES3.1 的开发在 2008 年 1 月的 TC39 会议上正式启动。这次会议上探讨了规范的目标，其中另有几位 TC39 成员对参与开发工作也表示出了兴趣。2 月 11 日，Lakshman 向 TC39 的内部邮件列表发送了一条消息，呼吁对 ES3.1 行动的参与。这封邮件提醒人们注意去年夏天准备的偏差与互操作性文档，并请求对这些文档提供更多反馈。在 2 月 21 日举行的电话会议上，每周两次电话会议的工作时间表得以确定。与以前的 ES3.1 讨论相比，参与这些电话会议的人数明显更多。图 30 中列出了相应的经常性参与者。起初，人们通过直发邮件来交换和讨论提案，也有一些讨论在 `es4-discuss` 邮件论坛进行。然而，由于与新版 ES4 主题相关的流量很大，因此很难挑选出其中与 ES3.1 相关的主题。为此在 4 月，一个单独的 `es3.1-discuss`[80] 邮件论坛 [TC39 et al. 2008] 得以成立。之后大多数在会议前后对 ES3.1 设计的讨论，都移到了这个论坛来进行。

| | |
|---|---|
| Douglas Crockford | Yahoo! |
| Pratap Lakshman | Microsoft |
| Mark S. Miller | Google |
| Adam Peller | IBM |

| | |
|---|---|
| Sam Ruby | IBM |
| Allen Wirfs-Brock | Microsoft |
| Kris Zyp | The Dojo Foundation |

图 30. 2008 年 ES3.1 WG 会议的经常性参与者。

在最早的讨论主题 [TC39 2008d] 里，其中有一个主题是评估 ES3.1 的总体目标，以及在解决问题和添加新特性时所应遵循的设计准则。对此，微软 Live 团队开发者和其他一些 Web 框架开发者所主张的早期立场，是避免使用任何可能导致脚本「无法在现有或旧版浏览器上解析」的新语法扩展。但这种「不允许新语法」的规则带来了过多的限制，忽视了多种浏览器已经具备某些语法扩展的事实。这个讨论的成果，是基于四种最知名的浏览器（IE、Firefox、Opera 和 Safari）而得出的「四人三票」（3 out of 4）规则，这些浏览器在微软的 JScript 偏差文档中都已经被分析过了。当四种浏览器中有三种在某特性上达成了一致或具备共通行为时，ES3.1 规范就应该以此为准。这条规则引发了关于「ES3.1 应如何处理浏览器互操作性问题」的广泛讨论。

人们一致认为，ES3.1 的首要原则是「不要破坏 Web」，亦即确定出有哪些语言变更会改变现有「已与主流浏览器相兼容的」网页的行为。但是现存的网页已有数以亿计，它们实际上依赖了 ECMAScript 规范中的哪些部分呢？对规范的哪些改动会破坏 Web 呢？有一则来自浏览器实现者的趣闻，认为由于现有网页的庞大基数，任何兼容的浏览器特性（不论用法多晦涩或令人难以置信）都可能被某些现存的页面使用。基于这种观点，在所有四种主流浏览器中共通的特性不应被更改，而四种浏览器中有三种支持的特性则很有望被标准化。但对于那些四种浏览器中只有两种支持，或在所有浏览器中都不同的特性和行为，又该怎么办呢？显然，这类特性和行为对于现有的可交互 Web 并非必需，并且还可能会在标准化过程中被进一步修改。

工作组还发现，基本上所有 ECMAScript 规范中允许实现存在的可变性，都不利于创建可兼容的网页。传统的语言规范中可能会允许「特定于实现」的差异，以便为语言实现者提供灵活性，或者适应在不同实现中已知的差异。但这种场景和万维网「通过多种独立创建的 Web 浏览器，访问全球互通的 Web」的理念，在根本上就是不匹配的。ECMAScript 规范需要比传统语言规范更规范化、更详细，并且还需要尽可能消除现有实现上的差异。经过 2 月的初步讨论，Douglas

Crockford [2008a] 在 TC39 Wiki 上发布了修订后的 ES3.1 目标（图 31）。

1. 对浏览器实现的统一：考虑采纳 4 种浏览器品牌中已有 3 种实现的特性，或 4 种用户计算机中已有 3 种部署的特性，减少跨浏览器的不兼容性。
2. ES3.1 应通过减少易混淆或麻烦的结构来改进语言，使业余开发者受益。
3. ES3.1 应通过减少易混淆或麻烦的结构来改进语言，使主要网站受益。
4. ES4 应成为 ES3.1 的超集。
5. ES3.1 应为语言的安全子集提供良好基础。
6. ES3.1 应尝试纠正 ES3 中的错误。
7. ES3.1 新特性应需要具体演示。
8. ES3.1 可能会废弃（或选择性删除）影响性能、安全性和可靠性的特性。
9. ES3.1 应提供可虚拟化性，允许对宿主对象的模拟。

图 31. 2008 年 2 月 ES3.1 修订后的目标 [Crockford 2008a]。

在 2008 年 3 月的面对面会议上，工作组一致认为应该立即开始编写实际的 ES3.1 规范文档。Pratap Lakshman 在会议上提供了一份经过纠正的 ES3 规范，其改动来源于 Mozilla 维护的勘误表 [Horwat 2003b]。工作组同意将其用作 ES3.1 基础文档，并请 Lakshman 担任编辑。跟以前的版本一样，规范文档将使用微软 Word 编写。通过相对于第三版的修订追踪（change tracking）功能，可以跟踪规范的演变情况，从而进行评审，并确保这些改动能被重新集成到新版 ES4 中。工作组成员被分配到了对具体新特性的规范文本开发上（图 32）。工作完成后，Lakshman 会将它们合并到 master 草案中。

| | |
|---|---|
| Lakshman | 基于 Mozilla「数组扩展」的新数组方法，以及 `reduce` 和 `reduceRight` |
| Lakshman | 为字符串添加数组式的下标索引支持 |
| Lakshman | `Date` 的改进 |
| Lakshman | 严格模式下的属性访问语义 |
| Crockford | JSON 支持 |
| Crockford | Unicode 更新 |
| Peller | 推荐基于微软偏差文档的改动 |
| Ruby | 十进制小数 |
| Zyp | 对象字面量的 getter 和 setter |
| Wirfs-Brock | 用于创建和检视属性的静态方法 |

| | |
|---|---|
| Wirfs-Brock | 更新伪代码记号和约定 |
| Miller | 对象的 freeze 和 seal，并对整份规范从安全角度进行评审 |

图 32. 截至 2008 年 3 月 28 日的 ES3.1 工作组任务分配 [TC39 2008c]。

2008 年 5 月 29 日，Pratap Lakshman 在 TC39 Wiki 上发布了 ES3.1 规范的初稿。更新后的草案通常每周发布一次，而「评审草案」则在每次 TC39 会议之前两到三周发布。在 2008 年 5 月 29 日到 2009 年 3 月 2 日之间，共有 26 个中间草案发布。

长期以来，IBM 一直主张 JavaScript 需要支持十进制小数运算。从 1998 年 11 月 19 日的 TC39 工作组会议起，Mike Cowlishaw 就希望将这一特性包含在 ES3 和初版 ES4 中。当 IBM 重新参与 TC39，开始贡献新版 ES4 和 ES3.1 时，他们再次强烈建议引入对十进制小数的支持。来自 IBM 的成员向 TC39 明确了一点，即 IBM 的政策是「反对所有不支持十进制小数运算的新语言标准」。TC39 中的许多人都对这一目标的可行性表示怀疑，但是 Brendan Eich 支持 IBM，他指出 Firefox 最常报告的的错误来自于那些不了解二进制浮点运算语义的 JavaScript 开发者。Eich 帮助了 Sam Ruby 开始开发原型。他们使用 Mozilla 的 SpiderMonkey 引擎，将 IEEE 754-2008 浮点十进制小数实现为了新的原始数据类型，使其可以在混合模式（mixed-mode）表达式中与 Number 类型结合使用。在 2008 年 9 月和 2008 年 11 月的 ES3.1 草案中，已经纳入了相当完整的十进制小数特性规范。但在 2008 年 11 月 19 日至 20 日的 TC39 会议上，需要就 ES3.1 草案中所应保留或删除的新特性做出最终决定。会议的第一项议题便是十进制小数运算支持。委员会的结论是，十进制小数设计仍然太不成熟，并存在剩余的设计问题，这些问题在不延迟 ES3.1 的情况下是不太可能解决的。会议纪要 [TC39 2008a] 中记录了这些担忧，并总结如下：

> 由于存在这些问题，因此决定将对十进制小数的支持推迟到 ECMAScript 的 Harmony 修订版。与会者承认，当前在 ECMAScript 十进制小数提案的开发上已经有非常显著的进展，并要感谢 IBM 的 Sam Ruby 对开发所投入的努力。与会者鼓励 Sam 和其他 TC39 成员继续开发该提案，并对「完全集成且通用的」

> 十进制小数运算提案版本成为 Harmony 修订版的组成部分感到乐观。

在 2009 年 1 月发布的下一个评审草案中，没有关于十进制小数运算的资料。

由于微软将 JavaScript 的开发职责转移给了位于 Redmond 的新小组，并且 Pratap Lakshman 拒绝了调迁的机会，因此他在 2009 年 3 月 25 日至 26 日的会议 [TC39 2009d] 上宣布辞去 ECMA-262 编辑的职务。委员会任命 Allen Wirfs-Brock 接任他的编辑职位。

Wirfs-Brock 回忆说，在 TC39 会议期间的某个休息时间，他找到了 Brendan Eich，建议将 ES3.1 重命名为整数级的版本。关于新名称的说法是，ES3.1 已经成长为 ECMA-262 的全面修订，与之前的三个版本一样重要。而新版 ES4 虽然已经终止，但其相关作品已经广为宣传。因此如果将 ES3.1 指定为第 4 版，会对 JavaScript 开发者社区和 Web 搜索引擎造成混乱。作为替代，Wirfs-Brock 建议 Ecma 永久停用 ECMA-262 第 4 版，并发布 ES3.1 作为第 5 版。对此 Eich 表示同意。在会议恢复后，他们向委员会提出了这个想法，并获得了接受。在认可了对版本号的更新后，会上委员会还同意接受以当时最新的草案作为最终草案。2009 年 4 月 7 日，「最终草案」以第 5 版的名义发布 [Lakshman et al. 2009]。该草案发布后，还有五份发行候选（release-candidate）草案发布，其中包含了一些较小的技术和编辑更改。在 2009 年 8 月苹果发现 [Hunt 2009]，使 `arguments` 对象继承自 `Array.prototype` 的决定，会与 Prototype 框架产生意外的交互，从而破坏多个苹果网站和 NASA 网站。于是这一改动从最终规范中删除。

2009 年 9 月 23 日，TC39 [2009b] 投票确认了 ES5 的完成，并将其提交给 Ecma GA 大会供批准。Ecma GA 大会审核批准的最终草案于 2009 年 10 月 28 日发布。在第 3 版获得批准十年后，《ECMA-262 第 5 版》于 2009 年 12 月 3 日由 GA 大会批准 [Ecma International 2009a]。GA 大会投票收到 19 票赞成和 2 票反对。IBM 之所以投反对票，是因为标准未包含对十进制小数运算的支持。Intel 则表示他们投反对票，只是因为他们缺乏足够的时间对规范进行完整的知识产权审查。

《ECMA-262 第 5 版》是 ISO/IEC ECMAScript 标准的快速通道修订版，它经历了 ISO 国家机构的审核过程。Allen Wirfs-Brock 根据审核

过程中的反馈，将许多编辑上的修订纳入了规范。这份修订版于 2011 年 6 月作为《ECMA-262 第 5.1 版》和《ISO/IEC 16262 第 3 版》发布。

## ES5 技术设计§

尽管 ES3.1 最初的目标非常保守，ES5 仍包含多项技术创新。

### 严格模式§

ES5 严格模式直接源于 Douglas Crockford 在 JavaScript 的设计中「纠正错误和不便」的目标。其中的一些不便在当时会造成语法错误，它们在 ES5 中可以在不影响现有代码的前提下被修正。例如保留字（reserved words）既无法作为对象字面量的属性键，也无法在点号后使用。但是，仍有许多 JavaScript 的错误特性并不能被无条件修复，因为它们可能会改变现有代码的运行时行为，从而「破坏 Web」。严格模式的设想，则是使 JavaScript 开发者有机会在新代码或更新后的代码中，明确是否选择性使用（opt-in）包含了此类修复的语言方言。为此，浏览器将必须同时支持严格模式和原有非严格模式的代码。并且在理想情况下，严格模式应该能在各个独立函数的层面上选择性切换，以便现有脚本能逐步转换为使用严格模式。人们希望随着时间的流逝，严格模式能成为编写新代码的主要方言。但它该如何获得最初的采用，仍然是一个问题。有人认为要等到所有主流浏览器都实现 ES5 严格模式，可能会有相当大的延迟。而浏览器博弈论预测，如果严格模式会使脚本在某些流行的浏览器上无法使用，那么开发者将不会使用它。使严格模式符合减法原则（subtractive）可以规避这个问题。严格模式并没有向 ECMAScript 添加新特性；相反地，它删除了

有问题的特性。在不支持严格模式的浏览器上运行时，无错误的严格模式代码也应该能继续按开发者的预期工作。

严格模式的一个早期问题，是该如何选择性地启用它。严格模式所具备的细粒度选择性，需要一种易于嵌入到脚本中的机制来实现，而不能利用类似 `<script>` 元素属性的外部手段。ES4 中考虑过提供可放置在 ECMAScript 代码内的 `use` 指令，以此来选择各种模式。但这样的指令会违反 ES3.1「不允许新语法」的设计准则。还有一种可能性是使用特殊形式的注释作为指令。但是 ES3.1 工作组也不愿意为任何形式的注释赋予语法上的意义，因为 JavaScript 压缩工具（minimizer）会删除注释。但 Allen Wirfs-Brock 发现，ECMAScript 中的 *ExpressionStatement* 语法可以将任何表达式转换成有效的语句。只要某个表达式是显式或隐式地（通过 ASI）后跟分号，那么它就可以转换成有效的语句，对仅含字符串字面量常量的表达式而言也是如此。这意味着那些诸如 `"use-strict";` 的声明，在语法上也是有效的 ES3 代码。因为这行代码只是一个常量值，所以在 ES3 中对其求值也没有副作用。同时这也是一个*空操作*。选择使用这样的语句作为严格模式看起来相当安全，因为任何现有的 JavaScript 代码似乎都不太可能已经利用了这样的语句形式，并且在 ES3 的实现中加载 ES5 代码时，旧版实现也都会忽略这行代码的存在。工作组采纳了这个想法，决定只要使用 `"use-strict";` 形式的声明作为脚本或函数主体的第一条语句，就表示整个脚本或函数应使用严格模式下的语义来处理。

严格模式的主要目标之一，是显式捕获那些容易产生但在运行时并不明显的编码错误。严格模式中添加了如下的新运行时错误：

- 给未声明的标识符赋值。在旧版 JavaScript 中，对输错的变量名称进行赋值，会导致在全局对象上创建属性。
- 给只读的自有或继承属性赋值。在旧版 JavaScript 中，这种操作会静默地不生效。
- 尝试在不可扩展的对象上创建属性。这样的对象在 ES5 之前并不存在，但为了保持一致性，在 ES5 中的严格模式之外执行此操作时，也将会静默地不生效。
- 将 `delete` 运算符应用于不可删除的属性。在旧版 JavaScript 中，这时的 `delete` 会返回 `false`。
- 将 `delete` 运算符应用于变量引用会产生语法错误。在旧版 JavaScript 中，对于显式声明的变量，`delete` 会返回 `false`。如

果变量引用来自与 `with` 语句相配合的对象，或者属于全局对象的属性，那么它在旧版 JavaScript 中将被删除。

严格模式还会移除或修改那些可能使程序更混乱、更难优化或更不安全的特性：

- 禁用 `with` 语句。`with` 语句提供了一种变量引用的动态作用域形式，这种形式可能会造成困扰，并且不利于各实现中的优化。
- `eval` 函数不能动态添加新绑定到当前作用域。
- `eval` 和 `arguments` 不能用作变量名或参数名。
- 函数的 `arguments` 对象不与其形参相关联。作为替代，严格模式下的 `arguments` 对象是一个数组式（array-like）的对象，其元素是传递给函数的参数值的快照。修改其元素不会修改相应形参的值，反之亦然。
- 严格模式下，函数的 `arguments` 对象没有 `callee` 属性。将这样的 `arguments` 对象传递给其他代码时，不会再隐式转移出对其上函数的调用能力[81]。
- 严格模式下，不允许实现在函数的 `arguments` 对象上提供 `caller` 属性。 `caller` 属性是 ES3 的一个非标准但已广泛实现的扩展，它允许遍历函数的调用堆栈，获取到所有的调用者函数。
- 严格模式下，调用函数时如果没有提供 `this` 值，全局对象对其就不可见。

在 Douglas Crockford [2007d] 列出的错误和不便清单上，还有许多关于严格模式的特性，但它们都没有纳入 ES5 中。对于这些特性，要么是 TC39 无法就其是否不受欢迎达成一致，要么是发现该改动不符合减法原则。例如，尽管 Crockford 和其他许多人都不喜欢 JavaScript 的自动分号插入，但许多开发者都更喜欢在没有显式分号的情况下编码。再比如，将 `typeof null` 更改为返回非 `"object"` 的其他值，也不符合减法原则。

## Getter，Setter 和对象元操作§

从最早的 JavaScript 实现开始，内置对象和宿主对象中的某些属性就已具备一些特殊性质。而通过 JavaScript 代码所创建的对象，是无法应用它们的。例如某些属性具有只读的值，或无法使用 `delete` 运算符删除；内置对象和宿主对象的方法属性在由 `for-in` 语句枚举时会被跳过。在 ES1 中这些特殊语义的确定，是通过将 ReadOnly，DontDelete 和 DontEnum 这些标记（attribute）与规范中的对象模型相关联的方式来实现的。这些标记会通过伪代码来测试，伪代码中定义了它们所涉及的语言标记的语义。这些标记没有被具体化（reified）——在 JavaScript 中，并不存在能为「新创建或已有的」属性设置这些标记的语言特性。ES3 中添加了一个 `Object.prototype.propertyIsEnumerable` 方法，用于测试 DontEnum 标记是否存在。但规范中仍然没有对 ReadOnly 或 DontDelete 标记执行非破坏性测试的相应方法。类似地，有许多由浏览器 DOM 提供的宿主对象也暴露了一些属性，它们通常叫做「getter / setter 属性」。在 ES5 中，这些属性被命名为访问器属性（accessor properties），会在存取属性值时执行计算。由于缺少对这些特性的标准化支持，JavaScript 程序员既无法定义「遵守与内置或宿主对象相同约定」的库，也无法实现 polyfill 来可靠地模拟这类对象。

对这些问题的统一解决方案，构成了新版 ES5 特性中最大的一部分。这部分特性没有正式名称，它们被非正式地称为「静态对象函数[82]」（Static Object Functions）或「对象反射函数」（Object Reflection Functions）。Allen Wirfs-Brock [2008] 为这个特性集编写了设计原理文档，其中包含了用例与以下设计准则：

- 干净地将元层（meta）和应用层分开。
- 尽量降低 API 的复杂度，例如方法的数量和方法参数的复杂度。
- 专注于命名和参数设计上的易用性。
- 尝试复用设计中的基本元素[83]。
- 尽可能使程序员或语言实现能静态优化对该 API 的使用。

第一条准则不鼓励在 `Object.prototype` 中添加形如 `propertyIsEnumerable` 的新方法，这会进一步模糊元层和应用层的分离。作为替代，ES5 工作组决定把这些特性作为命名空间对象的属性，从而将它们与应用层对象分离。他们考虑添加一个名为 `Reflect` 的新内置全局对象作为命名空间对象，但又担心这会与现有代码的名

称冲突。最终，他们决定将新函数作为 `Object` 构造函数的属性，而不是 `Object.prototype` 的属性。将对象构造函数作为命名空间是一个不错的选择，因为它是一个已经存在的全局变量，并且在当前的语言实现和以前的标准版本中，都没有在其上定义任何属性。同时，它的名称也与重新考虑对象定义的想法相契合。

下一个问题是确定 API 的形式。基于第二条准则，ES5 设计者希望避免给每个「属性标记」与「访问器属性」分别设置单独的查询与赋值函数。设计者考虑了许多方法来将这一特性合并到少量函数中。一些可能性包括使用具有 Boolean 标记（如「read-only」）的位编码的单个函数，或者具有大量位置参数（positional parameters）的单个函数。但是这两种方法的易用性都不够好。使用可选的关键字参数（keyword arguments）或许可以解决这些易用性问题，但 ES5 中缺少关键字参数。

Allen Wirfs-Brock 建议使用描述符对象（descriptor object），这种对象的属性将与各种属性标记相对应。这种描述符可以用来定义和检查属性。Wirfs-Brock 的第一份草案[84]展示了一种可能的 API 示例，用于向名为 `obj` 的对象添加属性：

```
Object.addProperty(obj, { name: "pi", value: 3.14159, writable:
false });
```

在示例中，描述符被编码为对象的字面量。对于描述符上没有，但又与其他属性标记相对应的属性，则会使用这里提供的默认值。还有一个设想中的 `defineProperty` 函数也会接受类似的描述符，可以用来更改已有属性的标记值。`defineProperty` 不会修改不存在于描述符属性上的标记。最后，还可以通过调用 `getProperty` 来获取对象上任何已有属性的完整描述符。

Mark Miller 提出了改进意见，建议让这个 `defineProperty` 能支持「添加新属性」和「修改现有属性」的使用场景。Miller 还建议从属性描述符中删除 name 属性，将描述符包装在一个对象中，该对象的属性名就是目标对象中受影响的属性名。这样的「属性映射表」（property map）将允许通过单次调用定义出多个属性。例如，以下操作就定义出了名为 `x` 和 `y` 的属性：

```
Object.defineProperties(obj, {
  x: { value: 0, writable: true },
```

```
  y: { value: 0, writable: true }
});
```

Miller 建议移除 defineProperty，只保留 defineProperties 的形式，因为后者也很容易用于定义单个属性。但是，这种表达方式很难定义出具有计算名称（computed name）的属性。在 ES3.1 中并没有语法能将计算值放置在「对象字面量的属性名称」位置处。最后，ES3.1 既提供了能通过「将名称独立地传递为参数」来定义单个属性的 defineProperty，也提供了能通过属性映射表定义多个属性的 defineProperties。ES5 定义的整套对象反射函数如图 33 所示。

| 函数名 | 行为 |
| --- | --- |
| Object.create | 使用所提供的对象作为原型，创建一个新的对象。支持通过可选的属性映射表来添加属性。 |
| Object.defineProperty | 基于属性描述符创建一个新属性，或更新已有属性的定义。 |
| Object.defineProperties | 创建或更新属性映射表中一组属性的定义。 |
| Object.getOwnPropertyDescriptor | 返回某个具名属性的描述符对象，不存在该属性时则返回 undefined。 |
| Object.getOwnPropertyNames | 返回包含了某对象全部自有属性名字符串的 Array。 |
| Object.getPrototypeOf | 返回所传入对象的原型对象。 |
| Object.keys | 返回一个 Array，其中包含对象自有属性的字符串名称，这些属性均在使用 for-in 时可见。 |
| Object.preventExtensions | 阻止所有向对象上添加新属性的操作。 |

| 函数名 | 行为 |
| --- | --- |
| Object.seal | 阻止所有向对象上添加新属性的操作，并阻止对其自有属性定义的修改。 |
| Object.freeze | 封存对象，并冻结其所有自有数据属性的值。 |
| Object.isExtensible | 测试是否可为对象添加新自有属性。 |
| Object.isSealed | 测试某对象是否被封存。 |
| Object.isFrozen | 测试某对象是否被冻结。 |

图 33. ES5 对象反射函数。

访问器属性也能通过可选的属性描述符来支持。除了 value 属性外，还可以使用具有 get 和（或）set 属性的描述符来定义访问器属性。例如一个用于「拦截对数据属性存取操作」的访问器属性，就可以定义为如下：

```
Object.defineProperties(obj, {
  x: {
    set: function (value) { this.privateX = value }, // 公有访问
器属性
    get: function () { return this.privateX }
  },
  privateX: {
    value: 0,
    writable: true
  } // 「私有」数据属性
});
```

除了这种基于反射的接口外，ES3.1 还在语法上支持了用对象字面量来定义访问器属性。四种浏览器中有三种都已经实现了这种语法，因此它符合加入新语法的标准。在对象字面量中，可以通过函数来定义访问器属性，其中函数关键字 function 被 get 或 set 所替换，例如：

```
var obj = {
  privateX: 0, // 一个普通的属性
  set x(value) { this.privateX = value }, // 访问器属性 x 的
setter
  get x() { return this.privateX }, // 访问器属性 x 的 getter
```

```
  get negX() { return -this.privateX } // 只有 getter 的访问器
};
```

要支持这些新特性，需要扩展语言内部（最早在 ES1 中定义的）对象模型，通过对象反射 API 来部分开放它。这也为重新考虑对象模型的术语提供了契机。ES1 通过一个值和一组标记的方式来描述属性，这些标记包括 ReadOnly，DontEnum 和 DontDelete。ES1 中的标记是无状态的，它们是关联到属性的记号，以自身的存在与否来表达其含义。ES3.1 设计者则希望将这些标记作为属性描述符对象的属性。为此他们更改了内部模型，将 ES1 标记建模为与每个对象属性相关联的 Boolean 状态变量，并将属性值重新建模为另一个状态变量。而内部标记的命名约定，也更改成了与内部方法一致的双括号模式。为了支持访问器属性，内部对象模型上新添加了 `[[Get]]` 和 `[[Set]]` 标记，这些标记的值分别是在值被引用时调用的 getter 函数，以及在赋值时调用的 setter 函数（或者是表示默认函数的 `undefined`）。根据某个属性是否既具有 `[[Value]]` 标记又没有 `[[Get]]` 与 `[[Set]]` 标记，可以区分出数据属性和访问器属性。

为了支持访问器属性，需要更新 ES1 中 `[[Get]]`、`[[Put]]` 和 `[[CanPut]]` 内部方法的规范。为了支持对象反射 API 使用的属性描述符，还需要添加 `[[DefineOwnProperty]]`、`[[GetOwnProperty]]` 和 `[[GetProperty]]` 内部方法。但光有这个反射 API 还是不够。在 ES3.1 中，`for-in` 语句对属性键的枚举、`Object.getOwnPropertyNames` 方法，以及 `Object.keys` 函数，都仍然使用非形式化的叙述来定义语义。

设计对象反射 API 的最后一步，是为这些属性描述符对象中表示属性标记的词汇，确定出一致且可用的命名约定。尤其像 DontEnum 和 ReadOnly 之类的名称就缺乏内部一致性，这引来了对其易用性问题的关注，当它们被用作布尔值标志时更是如此。例如若将属性设为可枚举，就需要表达双重否定（将 DontEnum 设置为 `false`）。在 2008 年初，Neil Mix [2008b] 在与新版 ES4 有关的主题帖上建议，将「enumerable」、「writable」和「removable」（对应 DontDelete）作为标记名会更好。Mark Miller [2008b] 对这些名称表示赞赏，并提出了一条设计准则：标记名应说明它「允许什么」而非「拒绝什么」。他还建议遵循「默认拒绝」的最佳实践来保证安全性。当定义属性时，全部所需的标记都要显式地启用。

对象反射 API 提供了 ECMAScript 早期版本中没有的新能力。它允许程序更改现有属性的标记，包括在数据属性和访问器属性之间切换。这里的一个考量在于，是否需要额外的标记来禁用此类更改。对此可能的命名包括「dynamic」、「flexible」和「fixed」。但人们担心添加这样一个额外的 Boolean 属性标记后，对现有实现可能产生的影响。如果一个语言实现没有可用的额外比特位来表示该标记，要怎么办呢？最后 ES3.1 工作组意识到，对属性标记的更改，等效于先对属性的当前标记做原子查询，再删除该属性，最后重新创建具有相同名称但标记值已修改的属性。鉴于这种等效性，可以使用单个标记来表达是否启用删除和修改。于是 DontDelete 和 removable 标记被重命名为了「configurable」[85]，以此来代表这一含义。Mark Miller [2010b] 绘制了 ES5 属性标记的状态图 [Harel 2007]（图 34），并发布到了 ECMAScript Wiki 上。注意当 configurable 标记为 `false` 时，仍然可以将属性的 writable 标记从 `true` 更改为 `false`。这个反常之处的存在，是为了让安全*沙箱*能更改某些内置属性，使其从「不可配置但可写」变为「不可配置且不可写」。

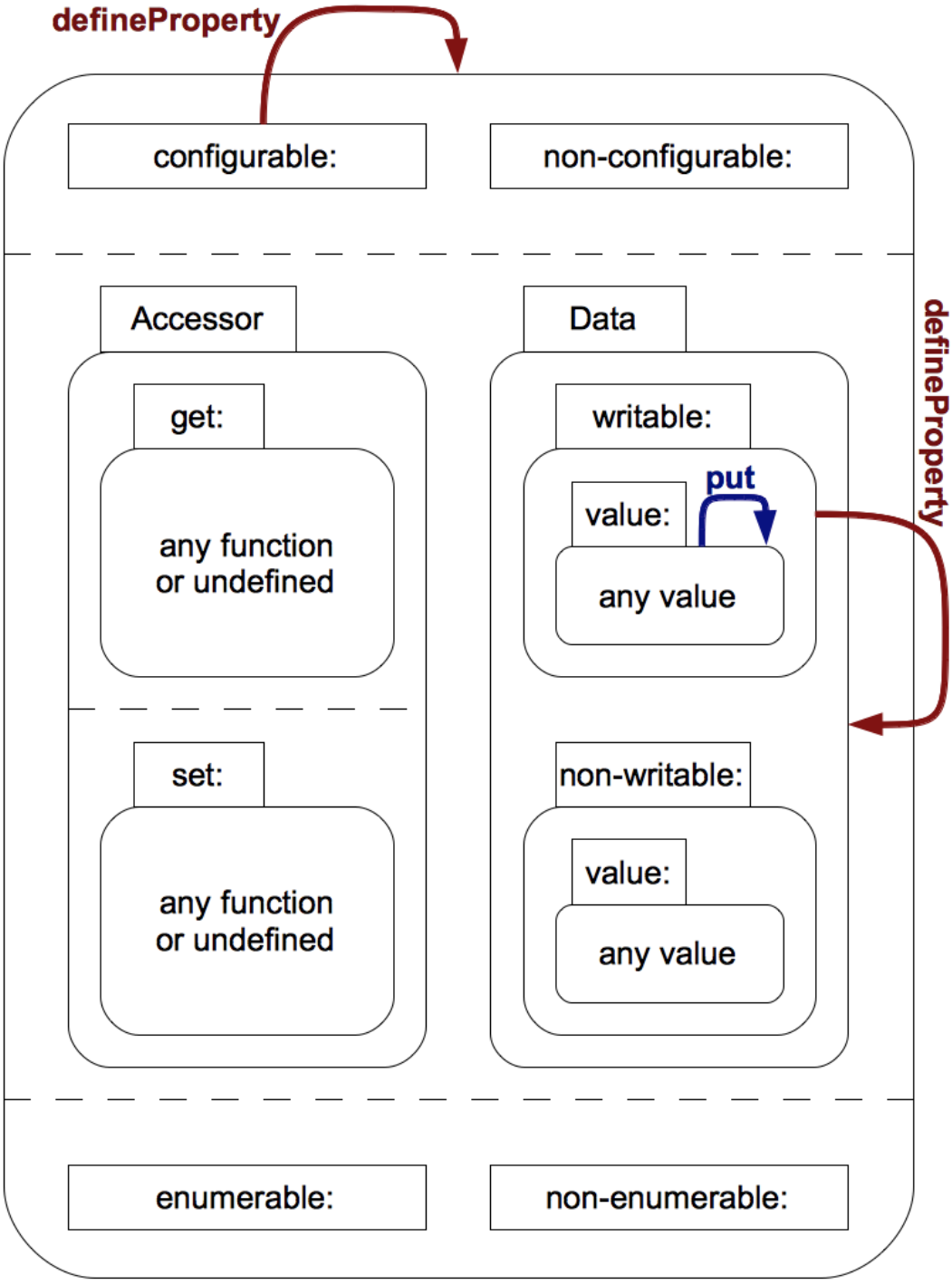



图 34. ES5 中属性的标记状态图 [Miller 2010b]。

作为在 JavaScript 应用中使用「基于原型风格的面向对象编程」范式的倡导者，Douglas Crockford 提倡使用名为 `beget` 的函数，来基于「显式提供的原型」创建对象。ES5 中的 `Object.create` 函数，实质上就是将属性映射表添加为第二个可选参数的 `beget` 函数，例如：

```
var point1 = beget(protoPoint); // 用 Crockford 风格创建一个
point
point1.x = 0;
point1.y = 0;
var point2 = Object.create(protoPoint, { // 使用 ES5 声明式风格
  x: { value: 0 },
  y: { value: 0 }
});
```

ES3 中 Crockford 的 `beget` 函数与 ES5 的对比。

Allen Wirfs-Brock 曾希望 JavaScript 程序员采用声明式风格，这样语言实现就可以识别出该模式，并据此优化对象的创建。然而在实践中，有个易用性问题妨碍了这种 ES5 模式的广泛应用。这个问题出在对默认属性标记的选择上。在 JavaScript 1.0 中，通过隐式赋值创建的属性具有与此等效的属性标记：`{writable: true, enumerable: true, configurable: true}`。但 ES5 属性描述符所遵循的「默认拒绝」策略，意味着在使用声明式风格的 `Object.create` 时，所有这些标记的默认值均为 `false`。例子如下所示：

```
// 以 Crockford 风格使用 Object.create
var point1 = Object.create(protoPoint);
point1.x = 0;
point1.y = 0;
// point1.x 的标记为
// writable: true, enumerable: true, configurable: true
// point1.y 的标记为
// writable: true, enumerable: true, configurable: true

// 以声明式风格使用 Object.create
var point2 = Object.create(protoPoint, {
  x: { value: 0 },
  y: { value: 0 }
});
// point2.x 的标记为
// writable: false, enumerable: false, configurable: false
// point2.y 的标记为
// writable: false, enumerable: false, configurable: false
```

要与 beget 示例的效果完全一致，使用 ES5 风格的 JavaScript 程序员就必须编写：

```
// 通过 ES5 与例行的标记值来创建 point 实例
var point2 = Object.create(protoPoint, {
  x: { value: 0, writable: true, enumerable: true,
configurable: true },
  y: { value: 0, writable: true, enumerable: true,
configurable: true }
});
```

对于大多数希望使用 JavaScript（传统意义上更为宽松的）默认值的程序员而言，这种表达方式过于繁琐。在实践中，人们通常使用 Object.create 的单参数形式来创建新对象，使用 Object.defineProperties 来定义和操作对象创建后的属性，很少使用 Object.create 的双参数形式来定义新对象的属性。

## 对象的完整性与安全性特性§

Netscape 3 所引入的 HTML <script> 元素 src 属性，使网页可以从多个 Web 服务器加载 JavaScript 代码。按最常见的说法，脚本会被加载到单个 JavaScript 执行环境中，在此它们共享同一个全局命名空间。跨站脚本也可以直接与此交互，这使得人们有条件创建 mashup 应用。跨站脚本的加载能力得到了广泛使用，并对支持基于广告的 Web 商业模式起了关键作用。但是跨站脚本既可能相互篡改与干扰，也可能如此影响原站点页面中的脚本。最后 Web 开发者们意识到，第三方脚本可能引发一些风险，比如窃取密码等用户机密数据，或者修改页面行为以欺骗用户。到 2007 年，人们发现 Web 广告代理商开始在暗中分发恶意广告。浏览器厂商开发了各种 HTML 和 HTTP 级别的特性来解决这一问题，例如内容安全策略（CSP）。但这种级别的特性并不能直接解决许多低层面的 JavaScript 漏洞 [Barth et al. 2009]。

当 Douglas Crockford [Adsafe 2007] 和 Mark Miller [Caja Project 2012; Miller et al. 2008] 参加 ES3.1 工作组时，他们都在积极开发用于支持

JavaScript 执行沙箱的技术，这些沙箱可用于安全地托管执行不受信任的第三方 JavaScript 代码。尽管 ES3.1 强势的向后兼容性需求意味着已无法消除许多已知的第三方脚本漏洞，但 Crockford 和 Miller 都力求消除那些可以在兼容前提下修补的漏洞，并继续添加新特性，以助于创建安全沙箱。在这之中，Mark Miller 对基于对象能力（object capability）[Miller 2006] 构建沙箱所需的特性尤其感兴趣。

这里最大的问题是 JavaScript 对象的可变性（mutability）。默认情况下，包括标准库对象在内的所有 JavaScript 对象，对于任意获取到了对其引用的代码而言，都是完全可变的。对象的属性和方法都可以被更改、赋值或删除。对于被直接引用的对象以及（从根级对象起）被间接引用的对象来说，情况都是这样的。尽管 ES3 中并没有方法能直接修改某个对象「到其原型对象」的引用，但是除 IE 之外的所有主流浏览器都已经实现了非标准属性 `__proto__`。通过该属性，可以修改对象的原型继承链。对于这种普遍存在的可变性，仅有的例外是 ES3 中带有 ReadOnly 或 DontDelete 标记的少数内置属性。

Mark Miller 和 Douglas Crockford 希望添加新能力，从而在将对象传递给不受信任的代码前，能锁定该对象的属性。这种能力可以用于保护需要暴露给沙箱的内置库对象，并让托管在沙箱内的代码能保护任何「需要被传递给不受信任的代码」的对象。通过将 DontDelete 标记重新设定为 Configurable 标记，并利用 `Object.defineProperty` 来使属性不可被修改与删除，语言提供了保护单个属性的基本能力。但这仍不足以防止不受信任的代码将新属性附加到传入其中的对象上。这种添加新属性的能力，使得不受信任的代码可以覆盖掉继承的行为，并有可能构建出用于泄漏私人数据的隐蔽通信渠道。在 ES5 中，这个问题是通过为每个对象关联一个名为 `[[Extensible]]` 的内部新状态来解决的。创建对象时，`[[Extensible]]` 默认被设置为 `true`。但如果将它设置为 `false`，那么新属性就无法添加到该对象上，此时语言实现也不允许提供任何用于修改对象 `[[Prototype]]` 的扩展。最后，一旦 `[[Extensible]]` 被设置为 `false`，就无法将其重置为 `true`。

`Object.isExtensible` 函数提供了一个用于查询对象 `[[Extensible]]` 状态的 API。`Object.preventExtensions` 函数能强制将 `[[Extensible]]` 设置为 `false`。而 `Object.freeze` 函数能很方便地将 `[[Extensible]]` 连同所有属性的 `[[Configurable]]` 与 `[[Writable]]` 标记都设置为 `false`，使对象的直接状态完全不可变。`Object.seal` 函数

则类似于 Object.freeze，只是它没有将 [[Writable]] 设置为 false。它能固定住对象的原型和属性集合，但仍然允许修改数据属性的值。

另一个被重点关注的问题，则是对全局对象的环境式访问（ambient access）。ECMAScript 将全局对象定义为了一个「属性位于全局作用域上」的对象，所有具名的标准库对象都作为全局对象的属性存在。并且大多数 JavaScript 的宿主环境，都会向全局对象添加特定于环境的对象与 API 函数。例如浏览器中的全局对象就和 window 对象相同，提供了对当前页面 DOM 对象与其他浏览器 API 的完全访问权限。一般而言，沙箱会限制对某些或所有全局对象属性的访问，或替代掉部分全局对象的属性。理论上，应该可以在所有沙箱代码外强制放置一个额外的词法作用域，通过设置这个作用域来实现这种效果。这种手段可以为某些全局对象属性提供替代性的绑定，或者通过提供值为 undefined 的遮盖式绑定，从而隔离这些属性。但是自 JavaScript 1.0 以来，始终有一种方法能访问到词法作用域隐藏不了的全局对象：

```
function getGlobalObject() {
  // 直接调用时，this 的值是全局对象
  return this;
}
getGlobalObject().document.write("pwned");
```

直到 ES5 前，直接调用函数（而非限定了对象的方法调用）的行为都会传入 null 作为隐式 this 参数，并且所有函数在被调用时，都会把值为 null 的 this 替换成全局对象。为了保证后向兼容性，现有代码中的这一行为是不能更改的。但 ES5 的严格模式则为新代码提供了「选择性使用新行为」的机会。在 ES5 中，严格模式下的函数永远不会用全局对象替换实际的 this 参数。沙箱可以只允许在其中运行严格模式的 JavaScript 代码，从而杜绝对全局对象的环境式访问。

在 ES5 的开发过程中，Web 上开始实际出现了如图 35 中示例的恶意攻击。ES3 规定使用对象字面量创建的对象继承自 Object.prototype，并且该对象字面量会使用 [[Put]] 内部方法，以设置新对象字面量中列出的属性。但是当使用 [[Put]] 将值分配给对象的属性时，需要查找原型继承链，来检查是否可以找到具有相同名称的属性。如果找到具有这一名称的 setter 属性，相应的 setter 函数就会执行。而如果在 Object.prototype 上设置这种 setter，那么只要尝试使用对象字面量形式创建一个与 setter 同名的属性，就都会调用到相应的 setter 函数，并为其传递该属性值。

```
// 假设我们已经发现某页面使用对象字面量
// 将一些有价值的信息存在 secret 属性中
function setupToStealSecret() {
  // 使用 ES5 前非标准的 getter / setter API
  // 在原型上定义一对 getter / setter
  Object.prototype.__defineSetter__("secret", function (val) {
    this.__harmlessSoundingName__ = val; // 将值存储在其他属性上
    exploitTheSecret(val, this)
  });
  Object.prototype.__defineGetter__("secret", function () {
    // 从另一个位置获取值，不会破坏原有代码逻辑
    return this.__harmlessSoundingName__;
  });
}

// 当代码使用具有 secret 属性的对象字面量定义对象时，秘密就会泄漏
var objectWithSecret = {
  secret: "password" // 这会触发继承的 setter
  // 可能还定义了其他属性
};
```

图 35. 使用 JavaScript 1.5 的 `__defineSetter__` 扩展的安全漏洞。通过在 `Object.prototype` 上定义 setter 属性，攻击者可以劫持使用对象字面量定义的特定属性的值。

对这个漏洞的修复，会产生对象字面量语义上的破坏性变更，但浏览器厂商愿意为修复这样的安全漏洞而做出改动。实际的规范更改很简单：不再使用 `[[Put]]` 语义来创建新对象的属性。ES5 使用了新的 `[[DefineOwnProperty]]` 内部方法，这个方法会始终忽略继承的属性，直接在对象上创建新的属性。

ES5 只能使 JavaScript 在安全方面前进一小步。当 ES5 的工作正在进行时，Douglas Crockford 建议在 TC39 内成立一个安全 ECMAScript（SES）工作组，其目的 [Crockford 2008d; TC39 2008b] 是探索开发一种新 ECMAScript 安全方言的可能性，这种方言不受后向兼容性的约束。SES 工作组在 2008 到 2009 年举行了四次会议，并评估了一些现有的 JavaScript 解决方案 [TC39 2008e]，以实现对不可信代码的安全求值。最后，TC39 放弃了对单独的新方言做标准化的想法，但诸如对象能力模型一类的 SES 概念则极大地影响了 Harmony 的研发。Ankur Taly [2011] 等人基于形式化手段，展示了严格模式和其

他 ES5 特性是如何支持「对 mashup 友好的安全 ECMAScript 子集」的。

## 活动对象（Activation Object）的移除§

在 ES5 之前，ECMAScript 规范已经明确要求使用 ECMAScript 对象来定义 ECMAScript 语言的作用域语义。每个*作用域轮廓*都由一个活动对象（AO）表示。活动对象也是普通的 ECMAScript 对象，其属性提供了变量和函数绑定，这些绑定是由与当前轮廓相对应的代码创建的。嵌套作用域被定义为一份活动对象的列表，可在其中依次搜索对某个引用的绑定。语言特点在于，引用绑定在访问「活动对象」和访问「用户程序定义出的对象属性」时，都会使用相同的属性访问语义运算符。ES1 及其后续规范指出，活动对象的概念仅用于纯粹的语言规范化，对 ECMAScript 程序而言是透明的。然而如果引擎完全符合规范，这种属性访问语义会导致出现一些边界情况下意料之外的行为。对于这些边界情况下的语义，实际实现则各有不同。

例如有一种意外情况，就是活动对象可能继承自 `Object.prototype`，而这是新创建对象的默认原型。这意味着 `Object.prototype` 的属性会被所有活动对象继承，并将作为每个活动对象的本地绑定。这会导致外部作用域中所有名称相同的绑定都被遮盖住。

对绑定的解析是动态发生的，其中会使用活动对象进行属性查找。因此只要在调用函数前，预先加入相应名称在 `Object.prototype` 上的绑定，任何在被调用函数中的自由引用都可以被拦截，例如：

```
// ES1-ES3

var originalArray = Array;
function AltArray() {
  // 用于替代内置的 Array 构造器
  // ...
}
// 调用一个函数，强制它使用 AltArray
Object.prototype.Array = AltArray;
```

```
somethingThatFreelyReferencesArray();
delete Object.prototype.Array; // 移除可选的 Array 绑定
```

另一种意外情况，是 ES3 中对 `try` 语句的 `catch` 子句形参的处理。此时的形参会在新作用域中作为「使用本地词法作用域」的绑定，而这个新作用域包含了 `catch` 子句的语句体。使用 ECMAScript 对象来表示作用域轮廓的手段，也给这一语义带来了问题。ES5 规范 [Lakshman and Wirfs-Brock 2009, Annex D] 对该问题的描述如下：

> 12.4：在第 3 版中，会以类似 `new Object()` 的形式创建出一个对象，作为解析「传递给 `try` 语句 `catch` 子句的异常形参」名称的作用域。如果实际的异常对象是一个函数，并且它在 `catch` 子句中被调用，那么作用域对象将被作为函数调用的 `this` 值传递。而后，函数体可以在 `this` 值上定义新属性。并且在函数返回后，这些属性名称将成为 `catch` 子句作用域内可见的标识符绑定。在第 5 版中，当将异常参数作为函数调用时，将把 `undefined` 作为 `this` 的值来传递。

在 2008 年的大部分时间里，工作组打算在新版本中引入 `const` 声明，因为尽管语义不同，这个特性在四种浏览器中也有三种支持。计划的目的是使 `const` 词法作用域缩小到块级，这有望进一步对早期规范版本中遗留的作用域模型施加压力。

为了解决这些问题，Allen Wirfs-Brock 在规范层面上开发了一种新的作用域与绑定模型。这个模型并不使用 ECMAScript 对象语义来定义标识符解析机制，并且引入了环境记录（environment record）的概念。环境记录包含单个作用域轮廓中的绑定，以及一些环境（environment），每个环境都是环境记录的有序列表。环境记录为在 ECMAScript 程序中某个位置做标识符解析提供了上下文。环境记录有不同的种类，它们可用于表示全局作用域、函数作用域、块级作用域，以及 `with` 语句的作用域。而所有环境都开放了一个规范级的通用协议，用于对单个绑定做定义、查找和值修改。规范中对于与「声明或访问变量」和「其他种类的绑定」相关的语言特性，都需要使用通用的环境记录协议。

不过，`const` 声明最终推迟到了未来的 harmony 规范版本中，因为工作组意识到过早纳入 `const` 可能会引入一些有问题的语义，从而妨碍将来更全面的块级作用域设计。新的作用域模型仍然在 ES5 中得以应

用，以解决与作用域相关的已知遗留问题，并为 ES6 中一组更全面的声明语句奠定了基础。

## 其他 ES5 特性§

除了图 33 中列出的对象反射函数外，ES5 还添加了以下的标准内置函数、方法和属性：

- `JSON.parse` 和 `JSON.stringify`，它们可以在对象与其 JSON 格式字符串之间做相互转换。
- 9 个新的 `Array.prototype` 方法：`indexOf`、`lastIndexOf`、`every`、`some`、`forEach`、`map`、`filter`、`reduce` 和 `reduceRight`。
- 1 个新的 `String.prototype` 方法：`trim`。
- `Date`：`Date.prototype.now` 方法与新扩展，用于解析和产生 ISO 8601 日期格式下的数据字符串。
- 新的 `Function.prototype` 方法 `bind`，以及函数实例上的 `name` 属性。

其他各类更改和增强包括：

- 修复 `with` 语句和 `catch` 子句形参作用域的语义。
- 使用 `[]` 语法对字符串做数组式的索引。
- 对正则表达式语法进行小幅修正。
- 每次求值正则表达式字面量时，都需要创建一个新的 `RegExp` 对象。
- 对错误的正则表达式字面量做早期错误报告。
- 全局对象中的 `undefined`、`NaN` 和 `Infinity` 属性具有只读的值。
- 要求对于所有规范中的算法，都用 `Object`、`Array` 等的内置初始值来替代当前值。
- 规范附录 D 和 E 中列出的各种非规范性语义修订。

# 实现与测试[86]§

在 2008 年 7 月于奥斯陆举行的 Ecma TC39 会议上，委员会同意在发布 ES3.1 之前，先获得两种互相兼容的实现。提出「两种互相兼容的实现」需求的主要原因，是确保 TC39 不会去对那些尚未被证明「技术上可行且与现有 Web 内容兼容」的内容进行标准化。Mozilla 承诺提供其中一种实现。由于微软的市场地位及其历来低频的浏览器更新，TC39 内部有一种强烈的情绪，认为微软应该公开浏览器宿主内的语言原型实现，以此作为 ES3.1 验证过程的一部分，展示它对 ES3.1 所应承担的责任。当时 TC39 计划在 2009 年 6 月的 Ecma GA 大会上，做好发布 ES3.1 的准备。这需要在 2009 年 3 月的 TC39 会议上，根据在 2 月到 3 月期限内所进行的互通性测试的结果，来决定出是否继续。当时还没有针对 ECMA-262 的官方一致性测试（conformance test）套件，自然也没有针对 ES3.1 新特性的测试用例，各种语言实现都具备自己的专用测试（ad hoc test）套件。另外除微软外的所有语言实现，也会使用 Mozilla 的 JavaScript 测试套件。微软对 Mozilla 测试套件所使用的 Mozilla 公共许可证有所顾虑，因此不会使用或贡献它。微软的首选测试套件应该使用 MIT 或 BSD 风格许可证，经由 Ecma 来提供。

在 2008 年 10 月，Pratap Lakshman 开始同时开发以 IE 为宿主的 ES3.1 实现，以及为其配套的测试套件。

已被实现出的测试用例将被贡献回社区。而整个测试套件的目标则是实现最大的代码覆盖率，这里的「代码」指的是规范中的伪代码。每个测试用例都以它在最新规范草案中的章节和算法步骤编号来命名，并放置在单独的 `.js` 源文件中。图 36 说明了测试文件所使用的命名约定。

| **sectionNumber-algorithmStepNumber-testNumber-s.js** | |
|---|---|
| **sectionNumber** | 规范中的章节号 |
| **algorithmStepNumber** | 某个算法步骤，其需求可由该测试用例验证 |

| sectionNumber-algorithmStepNumber-testNumber-s.js | |
|---|---|
| **testNumber** | 可选，应于该算法步骤具备多个测试用例时添加 |
| **-s** | 可选，应于测试用例面向严格模式时添加 |

图 36. 用于 esconform 测试用例文件的命名约定。每个文件包含一个测试，并以其在规范中所测试的伪代码步骤作为文件名。

Lakshman 实现了 900 多个测试用例，以及一个用于运行和报告各用例的简单测试工具。图 37 是其中一个测试用例的示例。

```
// Test Subclause 10.4.2 Algorithm 3 Step 1 Strict mode}
var testName =
"Eval code in strict mode-cannot instantiate variable in
calling context";

function testcase() {
    eval("'use strict ';var __10_4_2_3_1_s = 1");
    try{
        __10_4_2_3_1_s;
    } catch(e) {
        if (e instanceof ReferenceError)
        return true;
    }
}
```

图 37. 一个 ES5conform 测试。这个测试用例位于微软为 TC39 提供的初始 zip 文件中的 `10.4.2-3-1-s.js` 中 [Microsoft 2009a]。

在 2009 年 1 月的 TC39 会议 [Horwat 2009] 上，Pratap Lakshman 演示了 ES3.1 的原型。它将实验版 `JSCRIPT.dll` 集成到了刚发布的微软 IE 8 Release Candidate 1 版本中 。这次演示的内容包含了新语言特性与一致性测试套件。人们对这项工作大加赞赏，Waldemar Horwat 在会议记录中提到：「开发者们欣喜若狂」。

微软向 Ecma 贡献了这些测试，并在其开源项目门户 `codeplex.com` 上以「ES5conform」[2009] 的名义发布了它们。大致在同一时间，谷歌宣布 [Hansen 2009] 将发布他们在开发 Chrome 的 V8 JavaScript 引擎过

程中创建的开源 ES3 测试套件。这个测试套件被命名为「Sputnik」，包含了 5000 多个测试。

2010 年，ES5conform 和 Sputnik 成为了名为「Test262」的通用测试套件的核心，这一套件由 Ecma TC39 管理。像这样由 Ecma 技术委员会来维护和分发软件包，是一种根本性的改变。为了实现这一目标，必须要解决许多政策与许可证问题。Test262 开发过程中，最早的 ES5 阶段由 David Fugate 领导，到 ES6 阶段这一职责交给了 Brian Terlson。在 ES6 阶段后，Test262 由 Leo Balter 组织。现在，Test262 已经成为了 TC39 开发过程中不可或缺的一部分，每个 ECMAScript 新特性必须在测试后才能纳入 ECMAScript 标准。截至 2018 年 8 月 21 日，Test262 包含了 61877 个测试。Test262 的成功使得 TC39 相信，现在的规范已经不再需要配套的可执行文件了。

# 从 Harmony 到 ECMAScript 2015§

新版 ES4 的流产，使 TC39 自 1999 年以来终于能以相对干净的状态，来规划 JavaScript 未来的演进之路。TC39 不再考虑从头开始创造一种更好的语言，开始了一条走向成功的道路。只要花 7 年时间，就能抵达这条路的终点。

## 开始投入 Harmony§

TC39 的 Harmony 项目没有受限于 ES4 开发期间所做的决策，但仍然可以从中参考借鉴。虽然 TC39 仍然会被部分 ES5 项目中的决策所限制，但这项工作现在总体上与 Harmony 的预期方向一致。事实上，在 2008 年下半年和 2009 年的大部分时间里，TC39 在大部分会议时间里关注的都是 ES5。这也为整个委员会提供了一个机会，使他们能以 ES5 规范为起点，熟悉并投入 Harmony 上的工作。

## 稻草人（Strawman）与目标§

在 2008 年 8 月，ECMAScript Wiki 上出现了名为「Harmony 稻草人」的页面，`es4-discuss` 邮件列表也更名为 `es-discuss`。在 Harmony 项目提出后，`es-discuss` 上爆发了关于其潜在特性的新讨论。根据当时的工作流程，新的想法会在 `es-discuss` 或 TC39 会议上提出。如果 TC39 的成员认为某个想法有价值，他们会写出一份初步的设计或特性描述，并将其发布到稻草人 Wiki 页面上。随后这个「稻草人」将在 TC39 会议上进行展示。根据委员会的反应，该想法要么被放弃，要么被反复修改以继续完善。到 2008 年 11 月 21 日，稻草人 Wiki 页面 [TC39 Harmony 2008] 中共列出了以下条目：

- class
- const
- lambda
- 词法作用域
- 命名
- 返回到标签
- 类型

除了这里的 class 还是个占位符之外，所有条目都指向了一份由 Dave Herman 简要撰写的稻草人提案。

对于 Harmony 中可能的特性，人们进行了广泛的讨论。到 2009 年夏天，委员会决定进一步促进这项工作形成体系。在 2009 年 7 月的会议 [TC39 2009a] 上，TC39 成员决定是时候定义出 Harmony 的目标了。他们认为 ES3.1 的目标 [Crockford 2008a] 在此仍然适用，主要只是在其基础上做一些补充和改进。Brendan Eich [2009a] 发布了这些目标的新版本。其最终产物是如图 38 所示的「Harmony 目标说明」。

需求
1. 新特性需要具体的示例。
2. 保持语言对业余开发者的愉悦性。
3. 保留语言易于「从小规模开始迭代原型」的性质。

目标
1．成为如下场景下更好的语言：
　一、开发复杂的应用时。
　二、开发这些应用所依赖的库（可能包括 DOM）时。
　三、开发面向新版的代码生成器时。
2．切换到可测试的规范，理想情况下这对应于一个主要以 ES5 为宿主的定义解释器。
3．改善互操作性，尽可能采用事实上的标准。
4．尽可能保持版本号的简单和线性。
5．支持在对象层面上可静态验证的安全子集。

手段
1．尽量减少 ES5 之外所需的额外语义状态。
2．为以下维度提供语法上的便利：
　一、良好的抽象模式。
　二、高完整性的模式。
　三、经过净化后所定义出的核心语义。
3．通过可选的版本机制或前置杂注（pragma），去除易混淆或麻烦的结构：
　一、考虑使 Harmony 基于 ES5 严格模式。
4．支持虚拟化，允许对宿主对象的模拟。

图 38. 2009 年 7 月的 Harmony 目标说明 [Eich 2009a]。

## 倡导者模型§

Dave Herman 向委员会建议，认为委员会应该采用一种名为「倡导者模型」的开发方式[87]。基于这种模型，应由一位或一小组成员共同对一项单独的特性负责。倡导者（champion）需要写出最初的稻草人提案，并持续对其进行改进，直到提案可以被整合到实际规范中为止。从提出最早的稻草人提案起，倡导者还需要随提案发展向整个委员会做报告，并接受来自委员会和其他评审者的反馈。这些反馈意见也由倡导者消化，并据此决定是否对提案进行更新。基于倡导者模型，委员会应该就不会在倡导者报告过程中陷入「委员会设计」的行为了。不过最后仍然需要委员会全体达成一致，以决定将最终提案纳入规范。

委员会接受了 Herman 对倡导者模型的提案，并总体上有效地使用了这一模型。但这种机制也有崩溃的时候。这一时期的核心会员群体相对较小，技术能力也很强。他们有时根本抵挡不住「由委员会做一些设计」的诱惑，有时这其实是在提案上取得进展的最有效方式。有时会出现多位倡导者，他们会对某一特定特性或设计问题提出不同的解法和提案。在这种情况下，如果相互竞争的倡导者们不能就一份共同的提案达成一致，委员会就必须选择一个提案，或在某些情况下拒绝所有相互竞争的提案。

## 选择特性集§

在 2009 年、2010 年和 2011 年上半年的大部分时间里，TC39 的倡导者们都在致力于开发稻草人提案。他们与委员会一起审查这些提案，并试图获得必要的共识，以便将其推进到获得接受的状态。到 2009 年 8 月，稻草人页面 [TC39 Harmony 2009] 上的提案数量已从最初的 7 个发展到了 21 个。到 2010 年初，Harmony 特性集的大致形态开始出现。Brendan Eich [2010a] 将它们组织成了一系列主题（见图 39），并添加到了介绍 Harmony 目标的页面。到 2010 年 12 月，稻草人页面 [TC39 Harmony 2010b] 上的提案已经增加到了 66 个，另有 17 份提案 [TC39 Harmony 2010a] 已确认被推迟或放弃。到 2011 年 5 月初，稻草人页面 [TC39 Harmony 2011c; Appendix N] 有超过 100 个条目，而「已批准提案」的页面 [TC39 Harmony 2011a] 有 17 个条目。

主题
1. 模块化，换句话说即如何划分源码单元，以对外部用户隐藏内部细节
2. 隔离性，即阻止副作用传播，或仅允许特定引用来传播副作用
   * 零授权的制造者式模块（maker-style modules）
   * 其他涉及模块的「基础设施 / 上下文 / 内置特性」等的组合
   * 浏览器中缺乏隔离：多个互相连接的全局对象
3. 虚拟化，用于分层的客体代码托管，并连接不同的对象系统，特别是模拟宿主对象
   * 代理（Proxy）
   * 弱引用或 Ephemeron（类似 WeakMap 的数据结构，译者注）
4. 控制副作用，以便于较简单的迭代和状态机代码
   * 有限的 continuation 机制

```
   * 生成器与迭代器
5. 为库与工具赋能，这样 TC39 委员会就不会妨碍库的演进
   * Object.hashcode
   * 某种字节数组
   * 值类型（用于十进制小数运算等）
6. 语言改革，需要「更好的胡萝卜」来引导用户远离不好的形式
   * 块级作用域中的 let、const 和函数
   * 默认参数、剩余参数（rest parameter）和展开运算符（spread
operator）
   * 解构（destructuring）
7. 版本化，因为新语法是 Harmony 的一部分
   * 本主题意在尽量减少选择性使用的版本特性，从而简化迁移，并为未来的下一版
做准备
```

图 39. 2010 年的 Harmony 特性主题 [Eich 2010a]。

2009 年，Brendan Eich [TC39 2009b] 建议 TC39 将 2012 年 6 月作为 Ecma GA 通过「ES.next」的目标日期，并将特性冻结的目标日期定为 2011 年 5 月。随着 5 月目标日期的临近，规范明显还无法在 2012 年 6 月完成。但起草一份规范所承诺的特性列表以便专注于其开发，仍然有其意义所在。5 月会议 [TC39 2011b] 的大部分时间用于对稻草人列表进行分类，并就哪些剩余的稻草人提案将推进到「Harmony 提案」状态达成了共识。每份稻草人提案都先经过讨论，然后再去衡量是否有共识来推进它。在经过最低限度的审查后，一些提案获得推进，另一些则被拒绝。对于其他代表重要特性的提案，虽然委员会对当时相应的稻草人不够满意，但它们也得到了推进。这些提案被当作占位符，等待后续开发改进后的提案。如模块和类即均以此方式处理。最终的 Harmony 特性集并未在会议上被严格冻结。随着 ES.next 开发的继续，也有一些提案被加入和放弃。但此次会议所列出的提案清单，已经确立了后来 ES2015 的大致形态。图 40 列出了 5 月会议的参会者，附录 O 则展示了会后的 Harmony 提案页面 [TC39 Harmony 2011b]。

| | | | |
|---|---|---|---|
| Avner Aharon | Microsoft | Waldemar Horwat | Google |
| Douglas Crockford | Yahoo! (Phone) | Mark Miller | Google |
| Brendan Eich | Mozilla | John Neumann | Ecma |
| Cormac Flanagan | UCSC | Alex Russell | Google |

| | | | |
|---|---|---|---|
| David Fugate | Microsoft | Mike Samuel | Google |
| Dave Herman | Mozilla | István Sebestyén | Ecma |
| Luke Hoban | Microsoft | Sam Tobin-Hochstadt | Northeastern Univ |
| Bill Frants | Periwinkle (guest) | Allen Wirfs-Brock | Mozilla |

图 40. 2011 年 5 月 TC39 特性筛选会的参会者 [TC39 2011b]。

## 开始编写规范§

作为项目编辑，Allen Wirfs-Brock 全权负责根据 TC39 倡导者开发的 Harmony 提案，来创建 ES.next 规范文档。在微软，他的职责被分散在 TC39 相关工作与其他项目之间。2010 年 12 月，他离开微软加入 Mozilla，专注于 ES Harmony。

ES4 和 ES5 的经验使 Wirfs-Brock 明白，持续不断地对具体规范文档的开发，是完成新版标准的关键。2011 年 6 月 22 日，他怀着坚定的决心打开了最近完成的 ES5.1 规范的源文件，将封面页改为「第 6 版草案」，并将其保存为基准 ES6 规范草案。然后，他立即开始根据 5 月份的特性分类与委员会两年来做出的其他决定，在草案中编辑新材料。7 月 12 日，他发布了「ES.next 规范的第一份工作草案」[Wirfs-Brock et al. 2011a, b]。图 41 是该草案的变更摘要。这是委员会发布的 38 份草案中的第一份，最后一份草案则于 2015 年 4 月 14 日发布到了 Wiki 上 [Wirfs-Brock et al. 2015a, c]。

- `5.1.4` 引入补充语法的概念。
- `5.3` 引入静态语义规则的概念。
- `8.6.2` 等处取消了 `[[Class]]` 内部属性，增加了各种内部特征属性作为替代。
- `10.1.2` 定义了「扩展代码」的概念，即指可能使用新版 ES.next 语法的代码。一并重新定义的还有「严格代码」，即 ES5 严格

模式代码或扩展代码。

- 11.1.4 增加了在数组字面量中使用展开运算符的语法和语义。
- 11.1.5 增加了属性值简写的语法和语义，以及多种辅助抽象操作的语义。
- 11.2, 11.2.4 增加了参数列表中展开运算符的语法和语义。
- 11.13 增加了解构赋值运算符的语法和语义。
- 12.2 增加了 BindingPattern 语法和部分语义，以支持在声明与形参列表中的解构。
- 13 增加了对形参列表中剩余参数、参数默认值和模式解构的语法支持，并为它们提供了静态语义。但这种参数的实例化还未完成。对这类增强后的形参列表，也定义了实参列表的「长度」。
- 15 说明了此条目的函数规范，实际上是 [[Call]] 内部方法的定义。
- 15.2.4.2 重新规定 toString 不使用 [[Class]]。注意未来仍然需要增加一种明确的扩展机制。
- Annex B 改名为面向 Web 浏览器 ES 实现的规范化可选特性。

图 41. 首份 ES6 草案的变更日志 [Wirfs-Brock et al. 2011a, reformatted]。

## One JavaScript§

从 Harmony 项目启动起，TC39 就假定需要某种显式的「选择性使用」（opt-in）机制，来使用很多（甚至可能是所有）的新 Harmony 特性。这是从 ES4 时代延续下来的。在 ES4 时代，很多提案都包含了会使一些现有 JavaScript 程序失效的破坏性变更。Harmony 的进程对于纳入破坏性变更而言比较保守，但还是有所考虑的。在 Harmony 开发的前三年，具体的选择机制还没有确定，但也经常受到讨论。第一份 ES6 草案引入了「扩展代码」的概念，它是 ES5 严格代码的超集，但还没有包含对具体选择机制的描述。一些可供考虑的替代方案包括：使用 HTML <script> 元素属性从外部进行选择；使用新的 use mode 杂注语句；使用某种分隔的语法形式；添加一种类似于 "use strict" 的新指令等。有人担心这样下去将来会有多少种模式，难道

标准的每个大版本里都需要选择性地使用一种新模式吗？这似乎对语言用户和实现者而言都是个重大的复杂性负担。

Dave Herman [2011b] 在题为「ES6 不需要 opt-in」的 `es-discuss` 消息中认为，破坏性变更应该非常有限，并且仅限于在封装为 ES6 模块的代码内。绝大多数特性应该是非破坏性的，这样无论它们是否出现在模块中，都应该表现得完全一致。在某些情况下，这可能需要重新设计一些特性。在少数情况下，设想中的特性可能不得不为此而放弃。在对这条 `es-discuss` 消息的 150 多条回复中，这些想法逐渐得到了完善。在接下来的 TC39 会议上，Herman [2012] 做了一次名为「One JavaScript」的演讲，其中介绍了对这些想法的提炼。这里的关键在于，未来的程序员与 ECMAScript Harmony 的实现者们，应该能够用一种统一的 JavaScript 语言来思考，而不用考虑模式、版本或方言。TC39 有责任使 ES.next 的设计与此观点保持一致。会议的大部分时间都在讨论这条命题，以及它对各种 Harmony 特性的影响。大家的共识是尽量让「1JS」适用于 Harmony。在下一份规范草案 [Wirfs-Brock et al. 2012a] 中，扩展代码的概念被删除了。同时人们也做了各种其他的修改，以消除潜在的破坏性变更。

## Brendan 的梦想§

2011 年 1 月，在 Harmony 上投入了两年多的工作后，Brendan Eich [2011b] 发表了一篇名为《我的 Harmony 梦想》的博客文章，其中提出了一些关于语言进化和标准委员会的观点。文中核心则给出了他希望中「Harmony JavaScript 应该是什么样子」的示例。

> ……我想提出一个全新的 JavaScript Harmony 愿景。当然，这里的概念性尝试还（暂时）不够标准，但也不是一些随意而糟糕的衍生品。这些东西确实可能成为现实。如果有你们的帮助，它们会更有可能实现，并且能实现得更好（关于如何参与，可参见本文末尾）。

> 我正在模糊 Ecma TC39 目前的共识与我的想法之间的界限。这里的共识包括 Harmony 项目，以及 TC39 上一些人赞成的 Harmony 稻草人提案。我这么做是故意的，因为我认为 JS 需要一些新的概念上的完整性。它不需要安全的委员会设计，不管是那种「让我们把所有提案联合起来」的方法（这在 TC39 上是行不通的），还是盲目地「让我们求出提案间的交集，如果结果还是空集，那就这样算了吧」的方法（这也是行不通的，但这是更可能的坏结果），都是不可行的。

他介绍了各种场景下如何使用 ES5 特性进行编码的示例，以及如何在他梦想的 Harmony 中表达同等内容的替代性示例。这些设想中的例子，展示了 Harmony 提案的中间阶段，以及它们是如何演变成实际 ES2015 特性的。他提出的一些内容并未纳入 ES2015 中，大多数特性最后在某些方面发生了变化。另外也有必要做出其他的改动，因为 1JS 理念消除了对现有特性的语法和语义进行选择性修改的可能性。

为了解这些特性的演化，这里将比较 Brendan Eich 在 2011 年的「梦想」[88]和最终成为现实的 ES2015。

**梦想：绑定与作用域**。块级作用域的声明和自由变量引用，属于早期（解析时）错误：

```
let block_scoped = "yay!"
const REALLY = "srsly"
function later(f, t, type) {
  setTimeout(f, t, typo) // EARLY ERROR
}
```

**ES2015 现实**：支持块级作用域的 `let` 和 `const` 声明，但 1JS 令自由变量引用不属于早期错误。

**梦想：函数声明的改进**。消除 `function` 关键字，隐式 `return` 最后的表达式，即可为不存在自由变量的函数消除冗余闭包：

```
const #add(a, b) { a + b }
#(x) { x * x }
```

**ES2015 现实**：箭头函数取代了 `#` 符号，仅对带有表达式体的箭头函数采用隐式 `return`。对象字面量和类语句体中使用了简洁的方法。至于是否做对上层不可见的闭包优化，则交由实现决定：

```
const add = (a, b) => a + b // 表达式体隐式返回
x => x * x
x => { console.log(x); return x * x } // 语句体需要显式返回
// 对象字面量与类中的方法定义
class {
  add(a, b) { return a + b } // 不支持表达式体
}
```

**梦想：使用词法作用域的 this**。在 `#` 号函数中，`this` 基于词法作用域绑定：

```
function writeNodes() {
  this.nodes.forEach(#(node) {
    this.write(node)
  })
}
```

**ES2015 现实**：对于 `this` 和其他函数级作用域的隐式绑定，都会在箭头函数中使用词法绑定：

```
function writeNodes() {
  this.nodes.forEach(node => this.write(node))
}
```

**梦想：记录（record）与元组（tuple）**。支持不可变的数据结构，并支持内容层面的等价性：

```
const point = #{ x: 10, y: 20 }
point === #{ x: 10, y: 20 } // true
```

**ES2015 现实**：未支持。这一特性过于接近「可扩展的值类型」的概念，这在 Harmony 中并未获得充分开发。

**梦想：剩余参数、展开与解构**。支持可变长度参数列表的语法，可将数组展开到参数列表与数组字面量，并从数组和对象中提取组件。

```
function printf(format, ...args) {
  /* 将 args 作为真实数组使用 */
}
function construct(f, a) {
  return new f(...a)
}
```

```
let [first, second] = sequence
const { name, address, ...misc } = person
```

**ES2015 现实：**除了 ES2015 中不支持 ... 运算符的对象解构外，与设想完全相同。对象解构特性在后续版本中已经加入。

**梦想：模块**。一种简单的模块化设计，支持在浏览器中异步加载。

```
module M {
  module N = "http://N.com/N.js"
  export const K = N.K // N.K 的值
  exported export #add(x, y) { x + y }
}
```

**ES2015 现实：**每个文件一个模块，没有明确的模块定义定界符。支持更多的 `import` 和 `export` 形式。基于绑定而非模块间共享的值。

```
// http://M.com/M.js 的内容
export {K} from "http://N.com/N.js" // N.K 所 export 的绑定
export const add = (x, y) => x + y
```

**梦想：迭代**。对无括号的 `for-in` 语句进行扩展，使其能与「基于 proxy 的标准库」或「用户定义的生成器函数」所提供的迭代器一起工作。

```
module Iter = {"@std:Iteration"}
import Iter.{keys,values,items,range}
for k in keys(o) { append(o[k]) }
for v in values(o) { append(v) }
for [k,v] in items(o) { append(k, v) }
for x in o { append(x) }
#sqgen(n) { for i in range(n) yield i*i }
return [i * i for i in range(n)] // 数组推导
return (i * i for i in range(n)) // 生成器推导
```

**ES2015 现实：**1JS 鼓励使用 `for-of` 语句，以取代通过依赖模块和 proxy 来重载 `for-in` 的行为。内置的集合类也定义出了标准的 key / value / entries 协议。出于对未来前景的考量，推导式在 Harmony 开发的后期被放弃了。

```
for (k of o.keys()) append(o[k])
for (v of o.values()) append(v)
for ([k,v] of o.entries()) append(k, v)
```

```
for (x of o) append(x) // o 提供了其默认迭代器
function *sqgen(n) {for (let i of Array(n).keys) yield i*i } //
一个生成器
```

**梦想：无括号的语句**。这是更为现代的语法，在复合语句中取消了原先必需的小括号：

```
if x > y { alert("paren-free") }
if x > z return "brace -free"
if x > y { f() else if x > z { g() }
```

**ES2015 现实**：被认为过于激进而被 TC39 拒绝，未纳入规范。1JS 要求继续承认旧的语法形式，新旧形式的混合导致了设计和使用上额外的复杂性。

## 重新打造规范§

使用可执行、可测试的规范来表达 ECMAScript 语义的愿望，从新版 ES4 的工作中延续了下来。但使用 ML 作为规范语言的尝试已经被放弃了。在 Harmony 工作的早期，Allen Wirfs-Brock [2009] 提出了通过「以 ES5 JavaScript 编写的定义解释器」来确定 Harmony 的想法。这个想法甚至被列入了 Harmony 目标声明中（图 38）。但到 2010 年春天，在这个概念上仍然没有取得什么进展，TC39 成员对此方法也感到了更多的不确定性。而为 ES5（附录 P）所做的伪代码改进，已经消除了早期版本中伪代码存在的大部分可用性问题。并且 Test262 的进展也表明，一套全面的测试套件对于验证规范和实现同样有用。在 5 月的 TC39 会议 [2010] 上，人们再次讨论了规范的形式。当前现状对会议上的许多人来说仍然很有吸引力。苹果公司的 Oliver Hunt 发现，作为规范实现者，ES5 中的伪代码比他见过的任何可执行规范代码都更好用。于是会议一致决定继续使用伪代码来定义 Harmony。

对于项目编辑来说，创建规范并不仅仅是一件简单的集成任务。从理论上来说，提案应当由倡导者开发到「可以轻松集成到规范中」的程度。但在实践中，这种情况很少发生。一些倡导者对规范的结构或形

式不够熟悉，无法创建可集成的伪代码。另外一些人则没有必要的时间或专业知识来创建详细的语义规范。对于许多提案，Allen Wirfs-Brock 不得不设法将它们集成到规范中。这需要制定语义细节，并编写或重写提案在规范中的算法。

倡导者们往往会较为狭隘地关注自己的提案所定义的特性。好的提案会考虑到该特性如何与语言的现有特性交互。然而即使是最熟练的倡导者，也很难考虑他们的特性和「其他倡导者同时开发的其他提案」之间所有的潜在交互。所有特性都必须通过编辑，才能成为实际规范的一部分。所以 Wirfs-Brock 对于原有语言和所有 Harmony 提案如何结合在一起形成 ES6，有着最完整的看法。他特别关注跨越多个特性提案的交叉问题，并确保提案之间在语法和语义上的一致性。当整合已批准的提案时，他会试图将它们转化为一组可组合的正交特性 [Linsey 1993]。有时，这需要改变提案的语法或语义细节，甚至增加或删除重要特性。然后这些改变必须提交给倡导者，而且往往还要提交给整个委员会批准。

## 重组规范结构§

从 1997 年的第一版初稿（图 13）到 ES5.1 为止，ECMAScript 规范的组织结构基本没有变化。在编写 ES5 规范时，Allen Wirfs-Brock 发现规范中材料的基本排序令人困惑。他逐渐认识到规范实际上定义了三个独立的部分：

- 一个 ECMAScript 虚拟机，包括各种运行时实体及其语义。
- ECMAScript 语言的语法、语义，及其与虚拟机之间的映射。
- 所有 ECMAScript 程序都可以使用的各种标准库对象。

原始规范及其修订版将三部分交织在一起，掩盖了这一基本结构。Allen Wirfs-Brock 认为，将规范明确地组织成三部分结构将使其更容易理解，还能更清楚地介绍大量新的 ES6 材料。委员会对此表示同意。图 42 显示了 ES2015 规范的新组织结构与 ES5 规范之间的比较。

| 条目 | ECMA-262 第 5.1 版（245 页） | ECMA-262 第 6 版（545 页） |
|---|---|---|
| 1 | Scope | Scope |
| 2 | Conformance | Conformance |
| 3 | Normative References | Normative References |
| 4 | Overview | Overview |
| 5 | Conventions | Notational Conventions |
| 6 | Source Text | ECMAScript Data Types and Values |
| 7 | Lexical Conventions | Abstract Operations |
| 8 | Types | Executable Code and Execution Contexts |
| 9 | Type Conversion and Testing | Ordinary and Exotic Object Behaviors |
| 10 | Executable Code and Execution Contexts | ECMAScript Language: Source Code |
| 11 | Expressions | ECMAScript Language: Lexical Grammar |
| 12 | Statements | ECMAScript Language: Expressions |
| 13 | Function Definition | ECMAScript Language: Statements and Declarations |
| 14 | Program | ECMAScript Language: Functions and Classes |
| 15 | Standard Built-in ECMAScript Objects | ECMAScript Language: Scripts and Modules |
| 16 | Errors | Error Handling and Language Extensions |
| 17 | | ECMAScript Standard Built-in Objects |
| 18 | | The Global Object |
| 19 | | Fundamental Objects |
| 20 | | Numbers and Dates |
| 21 | | Text Processing |

| 条目 | ECMA-262 第 5.1 版（245 页） | ECMA-262 第 6 版（545 页） |
|---|---|---|
| 22 | | Indexed Collections |
| 23 | | Keyed Collections |
| 24 | | Structured Data |
| 25 | | Control Abstraction Objects |
| 26 | | Reflection |

图 42. 第五版和第六版规范的组织。在 ES6 规范中，第 6-9 条定义了虚拟机语义。第 10-15 条定义了语言，第 17-26 条定义了标准库。

## 新的术语§

ES6 为澄清和更新规范中使用的一些术语提供了机会。其中需要注意的一个领域，就是对象的命名规则。在 JavaScript 1.0 的实现中，JavaScript 程序可以访问特定于宿主和 JavaScript 引擎的对象。这些对象的基本语义，相比于用 ECMAScript 代码所能创建的对象，有着多种不同的区别。ES1 规范中使用了以下术语：「对象」、「原生对象」、「标准对象」、「内置对象」、「标准原生对象」、「内置原生对象」和「宿主对象」，以指代可以实现对象的各种方式。这些称呼之间的区别很微妙，但却没有特别的用处。人们不清楚这些类别中到底哪些允许特别的对象语义，也不清楚 JavaScript 程序员所创建的对象与其中的哪些相匹配。

ES6 的一个目标，是使大多数标准库和宿主对象能使用 JavaScript 代码进行「自托管的实现」。有了自托管的可能性，对象是由宿主提供、由引擎提供还是由程序提供，其中的区别就显得越来越不重要了。对象之间的语义差异要比「谁来提供它们」或「用来实现它们的技术」更为重要。

对此在术语上的基本需求，是区分具有正常语义的对象和具有反常（即不寻常）语义的对象。Douglas Crockford [TC39 2012b] 根据 Ecma

最高会员等级的名称，建议用「标准对象」来表示那些语义上使用 JavaScript 对象字面量或 `new Object()` 来创建的对象。凡是在语义上与普通对象语义有任何偏离的对象，都被称为「异质对象」。标准对象和异质对象都可能由宿主、引擎或应用程序员提供，也可能用 JavaScript 或其他语言来实现。

## 新的语义种类§

在 ES6 之前，除了那些定义标准库函数的算法之外，大多数伪代码算法都与语法产生式相关联，并指定了相应产生式的运行时求值语义。并没有必要对这些算法进行命名，因为它们是唯一与语法产生式相关联的语义。此外还有一些算法（如类型转换的算法和定义对象语义的内部方法）则没有直接与语法相关联。这些算法被赋予了名称，以便于从求值算法中引用。

ES6 引入了形如对象解构之类的新特性，它们具有复杂的行为，其规范必须横贯多种语法产生式。一些算法需要对解析树进行多次遍历以收集信息，或对跨越多个解析节点的求值步骤进行排序。还有一些常见的在语法上存在关联的行为，会为了保持一致性而在多种语言特性之间复用。为适应这些需求，ES6 规范中除了隐式命名的求值算法外，还可以将命名算法与解析节点关联起来。它们通过名称被其所关联的语法符号引用。通常这种命名算法是多态的，即一个同名算法被定义为多种语法产生式。实际选择的具体算法，取决于在解析特定源文本语法符号时所进行的推导。

为了最大限度地减少实现之间的差异，ECMA-262 的每一个后续版本都更精确地定义了错误条件，以及应在何时检测到它们。ES3 隐式地引入了「早期错误」的概念，并在 ES5 中进一步完善。所谓早期错误，指的是在脚本求值前就会被检测到并报告的错误。一旦检测到早期错误，就会阻止对脚本的求值。最常见的早期错误形式是语法错误。当脚本的源代码不能使用 ECMAScript 语法进行解析时，就会出现这种错误。语法错误隐含在了语法的定义中。ES3 引入了一些其他类型的早期错误，例如在 `break` 语句中引用了语句标签，而相应标签

在词法上没有包围住 `break` 语句时。ES5 严格模式中又增加了一些早期错误。尽管这些错误不属于解析错误，规范还是将大多数此类错误定义为语法错误，即对语言静态语义规则的违反。在 ES6 之前，多数这样的错误都通过位于求值算法附近的非正式叙述来确定，其他则通过使用伪代码来确定。这些伪代码会在求值算法中测试运行时的错误条件，然后基于叙述来说明该错误「可以或应该」作为早期错误报告。

ES6 特性中引入了更多种类的早期错误。例如，试图使用 `let` 或 `const` 声明来重复定义一个标识符，就属于早期错误。ES6 在语法中增加了「静态语义」（Static Semantic）子条目，用于一致地指定早期错误的触发条件。图 43 显示了一组早期错误定义的示例。如图所示，早期错误规则可以引用静态语义算法。静态语义算法使用与运行时算法相同的约定，只是它们可能不会引用 ECMAScript 环境的任何运行时状态——因为它们是在求值脚本之前应用的。这些静态语义早期错误规则和算法，仅限于使用和分析可从源代码中提取的信息，而无需执行源代码。运行时算法中可以调用静态语义算法，但静态语义算法不能调用运行时算法。

```
13.3.1.1 静态语义: Early Errors

LexicalDeclaration : LetOrConst BindingList ;
  * 如果 BindingList 的 BoundNames 包含 "let"，属于 Syntax Error。
  * 如果 BindingList 的 BoundNames 包含重复项，属于 Syntax Error。
LexicalBinding : BindingIdentifier Initializer (opt)
  * 如果 Initializer 不存在，且包含这条产生式的 LexicalDeclaration
对应的 IsConstantDeclaration 结果为 true，属于 Syntax Error。

...
13.3.1.3 静态语义: IsConstantDeclaration
LexicalDeclaration : LetOrConst BindingList ;
  1. 返回 LetOrConst 的 IsConstantDeclaration。
LetOrConst : let
  1. 返回 false。
LetOrConst : const
  1. 返回 true。
```

图 43. ES6 静态语义规则示例 [Wirfs-Brock 2015a, pages 194-195]。

# ES2015 语言特性§

Harmony 提案 Wiki 页面 [TC39 Harmony 2014] 最终版本中所列出的提案，被开发成了几十种语言及其标准库的新特性与扩展特性。典型提案在被纳入规范草案之前，要经过多次反复的迭代。有些提案在纳入规范草案后，还会继续进行改进。一些提案最终被放弃审议，或推迟到未来的版本中。

以下各节将深入探讨几项重要提案的发展历史，并总结其他重要特性的细节。

## Realms、Jobs、Proxies 和元对象编程（MOP）§

Harmony 的目标之一，在于使异质对象不分其为内置还是由宿主定义，均能实现自托管，并完全确定其由 Web 浏览器所实现的语义扩展机制。为支持这一目标，需要完善某些 ECMAScript「虚拟机」中现有的抽象，并进一步增加新的抽象，以确定新的（或不够明确的）语言特性。

「Realm」[Wirfs-Brock 2015a, pg. 72] 是一种新的规范抽象。引入它的目的，是为了支持在单个 ECMAScript 执行环境中描述多个全局命名空间的语义。Realm 能支持 HTML 页框的语义，这是 ECMAScript 自 ES1 以来一直忽略的浏览器特性。而「Job」[Wirfs-Brock 2015a, pg. 76] 这种规范抽象的加入，是为了确定性地定义 ECMAScript 执行环境该如何将多个脚本依次执行到完成（run-to-completion）。基于 Job 所提供的方法，能解释由浏览器和其他 JavaScript 宿主所提供的「事件派发」和「延迟回调」的语义。它们还为定义 ES2015 中 Promise 的语义建立了基础。

ES1 所提供的内部方法，基本上是个残缺的元对象协议。在对各种内置对象和宿主提供的对象做属性访问时，会有各类可见的语义区别。基于内部方法，可以将这些区别解释为它们在内部方法规范上的差异。但在 ES2015 之前，内部方法的语义还不够完整和规范，其使用也不够一致。为了「驯服」宿主对象，实现异质对象的自托管，并支持对象能力的*隔离层* [Van Cutsem and Miller 2013]。ES1 到 ES5 中所设计的内部方法，被转换成了一种明确的元对象编程（MOP）。

JavaScript 代码要想定义异质对象，就必须能为这些对象所用的内部方法提供相应的实现。这个特性是由 ES2015 中的 `Proxy` 对象 [Wirfs-Brock 2015a, pg. 495] 提供的。新版 ES4 提出了一种名为「catchalls」[TC39 ES4 2006a] 的机制，从而让 JavaScript 代码能逐对象地覆盖当「试图访问某个属性，或调用某个不存在的方法」时发生的默认动作。这个「catchalls」机制的目的，是改进 JavaScript 1.5 的非标准 `__noSuchMethod__` 机制 [Mozilla 2008a]。在 Harmony 中，Brendan Eich [2009b; 2009d] 引入了所谓的「动作方法」（action method）概念，使其能动态附加到对象上，从而令新版 ES4 的 catchalls 更进一步通用化。在对某个对象执行某些语言操作时，如果该对象上已定义了相应的动作方法，则会调用该方法。可用的动作集与 ES5 的内部方法集类似，但不是它们的直接映射。这里有个悬而未决的问题，即这些动作是在执行所有属性访问时触发，还是仅当访问不存在的属性时触发。Eich 所设计的用于将动作附加到对象上的 API，是以 ES5 对象反射函数为基础的：

```
// Harmony Catchall 提案

var peer = new Object;
Object.defineCatchAll(obj, {
  // 加入支持数组式行为的动作
  has: function (id) { return peer.hasOwnProperty(id); },
  get: function (id) { return peer[id]; },
  set: function (id, value) {
    if ((id >>> 0) === id && id >= peer.length) peer.length = 1
+ id;
    peer[id] = value
  },
  add: function (id) {
    Object.defineProperty(obj, id, {
      get: function () { return peer[id]; },
      set: function (value) { peer[id] = value; }
    })
```

```
  },
  // 其他动作的定义...
});
```

在这个例子中，属性 `has`、`get`、`set` 和 `add` 提供了动态附加到 `obj` 对象上的所有动作。各动作函数可在词法上共享对 `peer` 对象的访问，这就在 `obj` 和 `peer` 之间建立了一对一的关联。这些处理函数共同使用 `peer` 来支持对 `obj` 自有属性的存储。它们还会动态更新 `peer` 对象的 `length` 属性值，因此该值总比用作属性名的最大整数大 1。

在 Brendan Eich 的 catchall 提案之后不久，Tom Van Cutsem 和 Mark Miller [2010a; 2010b] 又提出了另一种设计。这就是「基于代理的 catchall 提案」[Van Cutsem 2009]，它定义了一套分层的对象交互 API。Proxy 提案的目的是支持对虚拟对象的定义，例如在安全的「基于对象能力」式系统中，定义出用于实现隔离的隔离层对象。TC39 基本认可了 Proxy 稻草人，并很快将其作为 Harmony 提案接受。

这份提案引入了 Proxy 对象的概念。提案没有扩展出具侵入性动作方法的基础对象，而是选择创建一个与处理器对象（handler object）相关联的 Proxy 对象，其中的方法称之为「trap」。Trap 会由语言操作而触发。通过处理器函数，可以完全定义出语言操作所用的对象行为。Trap 既可能是自包含的，也可能通过词法捕获的形式，与「对处理器函数可见的已有对象」一起使用。如下所示 [Van Cutsem and Miller 2010c]：

```
// 最早的 Harmony Proxy 提案

// 一个进行简单转发的代理
function makeHandler(obj) {
  return {
    has: function (name) { return name in obj; },
    get: function (rcvr, name) { return obj[name]; },
    set: function (rcvr, name, val) { obj[name] = val; return
true; },
    enumerate: function () {
      var res = []; for (name in obj) { res.push(name); };
return res;
    },
    delete: function (name) { return delete obj[name]; }
  };
}
```

```
var proxy = Proxy.create(makeHandler(o),
Object.getPrototypeOf(o));
```

在这个例子中，`makeHandler` 是用于创建处理器对象的辅助函数，其 trap 在词法上共享对「作为参数传递给 `makeHandler` 的对象」的访问。传递给 `makeHandler` 的对象可能是一个新创建的对象，这时它的作用类似于 catch-all 例子中的 `peer` 对象。另外，被传递的对象也可以是一个已有的对象。这时，trap 可以将部分或全部被截获的操作转发给该对象。在这种情况下，`obj` 对象的角色就相当于「被转发代理」的目标。

通过将 trap 方法放在处理器对象中的方式，可以避免它与基础对象属性的名称相冲突。提案中定义了 7 种基本 trap、6 种派生 trap[89]，以及 2 种针对函数对象的 trap。和 catchall 提案类似地，trap 和 ES5 的内部方法相接近，但也不是 ES5 内部方法的直接映射。ES5 中为 `[[GetOwnProperty]]` 和 `[[DefineOwnProperty]]` 内部方法建立了某些不可违背的一致性规则 [Wirfs-Brock 2011b, page 33]。而对 ES2015 来说，有个棘手的问题就是如何在实行[90]这些规则的同时，对「被冻结或密封的对象」与「不可配置的属性」进行虚拟化。

在对原始 Proxy 提案做原型建设后，Van Cutsem [2011] 宣布了重大修订：

> 几周前，Mark 和我坐在一起研究了 proxy 的一些现存问题，特别是如何让 proxy 更好地处理不可配置的属性和不可扩展的对象。其结果就是我们所说的「直接代理」：在我们的新提案中，proxy 总是另一个「目标」对象的包装器。只要以这种方式稍微转变我们对 proxy 的看法，很多早先开放的问题就不复存在了。并且这样一来，proxy 的开销在某些情况下可能会大大减少。

在「直接代理」的提案 [Van Cutsem and Miller 2011a, b, 2012] 中，目标对象（以下例子中的 `o`）类似于转发代理例子中传递给 `makeHandler` 的对象。它作为 Proxy 对象的内部状态而保存，并在调用 trap 时作为一个显式参数来传递。因为 Proxy 了解目标对象的信息，所以它可以在使用目标对象时，确保其符合必要的一致性规则。以下是直接代理版本的 Proxy 转发示例：

```
// Harmony 直接代理提案

// 一个进行简单直接转发的代理
var Proxy(o, {
  // 处理器对象
  has: function (target, name) {
    return Reflect.has(target, name)
  },
  get: function (target, name, rcvr) {
    return Reflect.get(target, name, rcvr)
  },
  set: function (target, name, val, rcvr) {
    return Reflect.set(target, name, val, rcvr)
  },
  enumerate: function (target) {
    return Reflect.enumerate(target)
  },
  // ...
});
```

这里 `Reflect` 对象的方法对应于标准的内部方法。它们使处理器函数能直接调用对象的内部方法，而非使用隐式调用它们的 JavaScript 代码序列。在直接代理的设计中，最初主要根据 ES5 的内部方法，定义出了 16 种不同的 trap。设计中还发现对于某些对象的内部操作，由于其没有用内部方法来定义，所以无法被 Proxy 拦截。Tom Van Cutsem、Mark Miller 和 Allen Wirfs-Brock 共同开发了 Harmony 内部方法和 Proxy 的 trap，使它们保持一致，并足以表达 ECMAScript 规范和宿主对象中所定义的所有对象行为。其具体的实现手段是增加新的内部方法，以及将一些不可截取的操作，重新定义为基础级、可捕获的常规方法调用。此外提案还定义了每个内部方法的关键一致性规则。ECMAScript 的实现和宿主都必须确保符合这些一致性规则，而 `Proxy` 可以对自托管的异质对象实行[91]这些规则。图 44 是对 ES2015 中元对象编程的概述：

| ES5 内部方法 | ES6 内部方法 | ES6 Proxy Traps 与反射方法 |
|---|---|---|
| [[Canput]] | | |
| [[DefaultValue]] | | |
| [[GetProperty]] | | |
| [[HasProperty]] | [[HasProperty]] | has |

| ES5 内部方法 | ES6 内部方法 | ES6 Proxy Traps 与反射方法 |
|---|---|---|
| [[Get]] | [[Get]] | get |
| [[GetOwnProperty]] | [[GetOwnProperty]] | getOwnPropertyDescriptor |
| [[Put]] | [[Set]] | set |
| [[Delete]] | [[Delete]] | deleteProperty |
| [[DefineOwnProperty]] | [[DefineOwnProperty]] | defineProperty |
| [[Call]] | [[Call]] | apply |
| [[Construct]] | [[Construct]] | construct |
| | [[Enumerate]] | enumerate |
| | [[OwnPropertyKeys]] | ownKeys |
| | [[GetPrototypeOf]] | getPrototypeOf |
| | [[SetPrototypeOf]] | setPrototypeOf |
| | [[IsExtensible]] | isExtensible |
| | [[PreventExtensions]] | preventExtensions |

图 44. ES6/ES2015 的元对象协议由规范级内部方法定义，并通过 `Proxy` 的 trap 和 `Reflect` 方法进行验证。

在直接代理的设计中，使用了一个封装过的目标对象。但它的设计目的并非提供目标对象的简易透明封装。与其表象相反，代理并不是一种用来记录属性访问或处理「方法未找到」问题的简单方式。为了支持这些用例而朴素实现的 Proxy 对象，通常是不可靠或有错误的。直接代理的核心使用场景，是对象的虚拟化和安全隔离层的创建。正如 Mark Miller [2018] 所解释的那样：

> Proxy 和 WeakMap 的最初设计动机，是支持隔离层的创建。单独使用的 proxy 不可能是透明的，也不能合理地达到接近透明的程度。隔离层能合理且几乎透明地模拟 realm 的边界。对于具备私有成员的类而言，这种模拟基本上是完美的。

## 块级声明作用域§

从初版 ES4 起，就有对加入块级声明作用域的诉求。具有类 C 式语言语法经验的程序员，会希望位于 {} 块中的声明属于该块中的局部变量。最早 JavaScript 1.0 中的 var 作用域规则令人惊讶，有时会掩盖严重的错误。其中的一个常见 bug 就是循环中闭包的问题：

```
// ES3

function f(x) { // 此函数有循环中闭包的 bug
  for (var p in x) {
    var v = doSomething(x, p);
    obj.setCallback(function (arg) { handle(v, p, arg) });
    // 全部在循环中创建的闭包都共享 v 和 p 的绑定
    // 而不是在每次迭代中使用不同的绑定
  }
}
```

这种手法在操作浏览器 DOM 的代码中很常见——即便是有经验的 JavaScript 程序员，有时也会忘记 var 声明不是块级作用域的。

除非破坏已有代码，否则现有的 var 声明是无法改变为块级作用域的。在新版 ES4 尝试中，已经确定使用关键字 let 和 const 作为声明，以满足对块级作用域的需求。关键字 let 用于定义可变的变量绑定，而 const 则用于定义不可变的常量绑定。它们的使用并不限于块，而是可以出现在任何能出现 var 声明的地方。新版 ES4 设计团队甚至还制作了写有标语「let 是新的 var」的T恤。Harmony 继承了 let 和 const 声明，但新版 ES4 工作中仍有许多相关的语义问题尚未得到解答。

ES5 曾考虑增加 const 声明。ES5 规范中包含了可用于确定块级声明绑定语义的抽象。但至于这些语义究竟该如何确定，则并不明显。下面的代码片段说明了一些问题。

```
// ES2015

{ // 外层块
  let x = "outer";
  { // 内层块
    console.log(x);
    var refX1 = function () { return x };
    console.log(refX1());
    const x = "inner";
    console.log(x);
```

```
    var refX2 = function () { return x };
    console.log(refX2());
  }
}
```

在 const 声明之前的内层块中，出现的对 x 的某些引用或所有引用，是否应该是编译时错误呢？还是说它们应该是运行时错误呢？如果它们不是错误，那么是否应该将其解析到 x 的外部绑定呢？或者说内层的 x 在初始化之前，是否应该以 undefined 为默认值？如果在 const 声明之前调用函数 refX1，是否应该和在声明之后调用函数一样，解析到同样的 x 绑定和相同的值呢？如果 x 的内层声明是一个 let 声明，上述所有问题仍然适用。针对这些情况下的引用，Waldemar Horwat [2008a] 描述了四种可能的语义：

- A1. 词法死区。在同一块中「文本上前于」（textually prior）变量定义而出现的引用，属于错误。
- A2. 词法窗口。在同一块中「文本上前于」变量定义而出现的引用，进入外部作用域。
- B1. 临时性死区。在同一块中「临时性前于」（temporally prior）变量定义而出现的引用，属于错误。
- B2. 临时性窗口。在同一块中「临时性前于」变量定义而出现的引用，进入外部作用域。

Horwat 感谢 Lars Hansen 将「死区」的概念引入讨论。术语「临时性前于」指的是运行时求值顺序。A2 和 B2 是不可取的，因为这使得块中同一名称在不同的位置，可以有不同的绑定。并且在 B2 的情况下，块中某处的名称甚至在不同的时刻，都可以有不同的绑定。A1 是不可取的，因为它妨碍了以这些声明形式来定义相互递归的函数。A2 的缺点在于，它需要对所有引用进行运行时初始化检查，不过这其中有许多可以被编译器基于相当简单的分析来安全地消除。但在花了近两年时间后 TC39 最终达成的共识，是认为新的词法声明形式应具有 B1 的临时性死区（TDZ）语义。这些语义可由下面这些规则来概括：

- 在一个作用域内，任何名称都只有唯一的一个绑定。
- let、const、class、import、块级函数声明和形参绑定在运行时是死的，直到初始化为止。
- 访问或赋值给一个未初始化的绑定，属于运行时错误。

在规范中，上述第一条规则表示为早期错误规则，另外两条则表示为运行时语义算法。

当 Allen Wirfs-Brock 开始将 `let` 和 `const` 集成到规范中时，他发现二者与传统的 `var` 和 `function` 声明之间，还存在着许多潜在的交互。这导致 TC39 又进行了一轮讨论，就下列补充规则达成了一致意见：

- 一个名称的多个 `var` 声明可以存在于任何层级的块嵌套中。它们都指向同一个绑定，其定义会被提升到最接近的外层函数或顶层全局作用域中（ES1 遗留语义）。
- 允许为同一名称进行多次 `var` 声明和函数 / 顶层全局作用域内的 `function` 声明，每个名称对应一个绑定（ES3 遗留语义）。
- 所有其他在同个作用域中的多重声明，都属于早期错误，包括 `var/let`、`let/let`、`let/const`、`let/function`、`class/function`、`const/class` 等。
- 如果一个块级的 `var` 声明名称，被提升到了任何同名的外层 `let`、`const`、`class`、`import` 或块级 `function` 声明之上，这也属于一个早期错误。
- 当创建绑定时，`var` 声明会被自动初始化为 `undefined`，因此对它们的访问没有 TDZ 限制。

另一组问题则涉及对全局声明的处理。在 ES2015 之前，所有的全局声明都会在宿主环境提供的全局对象上创建属性。但是对象属性并没有像实现临时性死区所需的那样，规定将一个属性标记为未初始化。有一份提案要求把全局层级上新 `const`、`let` 和 `class` 声明的出现，当作是 `var` 声明。这方面存在先例，因为一些 ES2015 之前的 JavaScript 引擎，已经以这种方式实现了 `const` 声明。然而这将导致在全局层级上使用新的声明时，会和其他位置上的使用不一致。相比之下 TC39 的共识，则是词法声明规则应尽可能一致地适用于所有类型的作用域。对于全局作用域，`var` 和 `function` 声明保留了创建全局对象属性的遗留行为，但所有其他声明形式，都会创建不影响全局对象属性的词法绑定。

新的规则不允许应用存在矛盾的 `var/let` 多重绑定，对类似的冲突而言也是这样的。但例外是那些不使用 `var` 或 `function` 声明创建的全局对象属性，它们不会导致多次声明之间的冲突。在这些情况下，一个全局的 `let/const/class` 声明会遮盖名称相同的全局对象属性。这暗含

了一条规则，即使用新声明定义的全局变量，不能在单独的脚本中多次定义。

仅仅增加块级作用域的 `let` 和 `const` 声明，还不足以完全消除循环中闭包的隐患。这里还有一个由 `for` 语句引入的变量作用域问题，即 `for (var p in x)`。ES2015 解决这个问题的方式，是允许在 for 语句的头部使用 `let` 和 `const` 来代替 `var`。以这种方式使用的 `let` 或 `const` 会在作用域轮廓中创建一个绑定，这个绑定会在循环体的每次迭代中重新创建。循环 `for (const p in x) {body}` 在*去糖化*之后，大致如下所示：

```
// ES2015

// for (const p in x) {body} 的去糖后近似表示
{ let $next;
  for ($next in x) {
    const p = $next;
    {body}
  }
}
```

为处理 C 风格的三表达式 `for` 语句而引入的词法绑定比较复杂，争议也较大。JavaScript 1.0 已经包含了使用 `var` 声明作为此类语句第一个表达式的能力，所以 `let` 或 `const` 声明在那里应该也可以使用。但是，这种声明所产生的约束力有多大呢？是应该有一个单独且生命周期为整个 `for` 语句的绑定，还是应该像 `for-in` 语句那样，为循环的每一次迭代建立一个单独的绑定呢？答案并不明确，因为常见的编码模式是利用第二、三个表达式或循环体中的代码，来更新所声明的循环变量的值，以便在循环的下一次迭代中使用。如果每次迭代都得到一个新的循环变量绑定，就需要自动使用上一次迭代的循环变量最终值，来初始化下一次迭代中的循环变量绑定。大多数类似 C 的语言，都采用了每条 `for` 语句对应一个单独绑定的方式，而非每次迭代对应一个绑定的方式，这也是 ES6 规范草案最初的做法。但是，这种方式仍然存在循环中闭包的问题。为此，对于使用 `let` 声明的三表达式语句，规范最终改为每次迭代使用一个绑定，并在迭代之间传递值。事实证明，对于第一个表达式中的 `const` 声明来说，使用每个循环语句唯一的绑定就足够了，因为此类变量的值不能被 `for` 头部或循环体中的其他表达式修改。

另一个重要的问题，是在语句块中声明函数时的语义。ES3 有意排除了（第 12 节）对块内函数声明的任何语法或语义规范。但各实现均忽略了这一指导，允许这样的声明——不幸的是，每个主流浏览器实现都为其赋予了不同的语义。不过在某些使用场景 [Terlson 2012] 下，这些语义之间所存在的重叠，是足够进行这样的函数声明，并在所有主流浏览器中都兼容地使用的。根据 ES2015 的词法声明规则，其中一些使用场景将被认为属于非法，或需要改变其含义。若在这些场景下实现新的规则，将会「破坏 Web」。这对严格模式来说不是问题，因为 ES5 已经禁止语言实现在严格模式代码中提供块级函数声明。对于非严格模式的代码，一种方法是效仿 ES3，不指定任何关于块级函数的内容——让每个实现来决定「是否以及如何」将块级函数声明与新的词法声明形式相整合。但这不利于互操作性，也与 1JS 的目标相悖 [TC39 2013b]。与其相反地，TC39 [2013a] 确定了少数几个用例，其中现有的块级函数具备互操作性且有实际用处，但根据新规则却会出现错误。例如：

```
// 兼容但非标准的 ES3 扩展

function f(bool) {
  if (bool == true) {
    function g() { /*do something*/ }
  }
  if (bool == true) g(); // 这在所有主流浏览器中均可用
}
```

对此的修复方法，是定义一些额外的非严格模式代码规则 [Wirfs-Brock 2015a, Annex B.3.3]。这些规则可以静态地检测那些特定的可互操作用例，并使其合法地与遗留网页相兼容。对于上面的例子，规则会把其代码当作这样：

```
// ES2015 附录 B 中的去糖化

function f(bool) {
  var g; // 如果顶层存在由 let 声明的 g，则属于早期错误
  function $setg(v) { g = v }
  if (bool == true) {
    function g() { /*do something*/ }
    $setg(g); // 将本地 g 设为顶层 g 的值
  }
  if (bool == true) g(); // 引用顶层 g
}
```

# 类§

在 2008 年 7 月发起 Harmony 工作的 TC39 会议上，相当多时间都用来讨论「是否应该以及如何」纳入类。在 ES4 的前后两次尝试中，为了开发复杂的类定义语法和语义，人们都付出了巨大的努力。而且这两次尝试中的设计，都需要新的运行时机制来支持。这些设计可以宽泛地描述为「受 Java 启发的类」。

Mark Miller [2008d] 认为，对于类抽象所需的大部分运行时机制，在 ES3 中已经基于 lambda 函数和词法捕获技术实现了。词法捕获技术类似于 Scheme [Dickey 1992; Sussman and Steele Jr 1975]，且由 Douglas Crockford [2008b, pages 52-55] 为适应 JavaScript 而进行了修改。这种「lambda 去糖化」的类定义风格，与模块模式实质上是一致的。它表明类只是一个小而轻的模块，其目的就是用来被多次实例化。Miller 称这种方法为「糖式类」（classes as sugar）。

Cormac Flanagan [2008] 将最初对类的讨论总结如下：

> EcmaScript（原文如此）需要提供对「具有数据抽象和隐藏的高完整性对象[92]」更好的支持，也需要更好地支持私有字段和方法……
>
> ……我们最初专注于一个简单的、极简的设计，它不支持继承或类型注解，并使用在实例中私有的数据。类名没有单独的命名空间，类对象是一种新的（一等公民）值。

Flanagan 提出的稻草人提案，使用了简单的类定义语法。如下所示：

```
// Flanagan 的 Harmony Class 稻草人

class Point (initialX , initialY) {
  private x = initialX;
  private y = initialY;
  public getX() { return x };
  public getY() { return y };
}
```

Cormac Flanagan 的提案内容并未完整地「去糖化」，并且包含的语义细节也很少。Mark Miller [2008c; 2009; 2010a] 用类似的表层语法设计对其进行了反驳。Miller 的提案进行了完整的去糖化，不需为类实例提供一种新的运行时对象。在 Miller 的设计中没有继承，所有的方法和实例变量都默认为私有访问。所有的方法和实例变量都被表示为逐实例的词法捕获声明，这些声明只能从类定义的代码体中直接访问。通过类实例对象的属性，提案支持从外部访问公有方法，并为公有实例变量提供了 `get` 访问器。从外部直接对实例变量赋值是不允许的，并且提案也不使用`this` 关键字。

Mark Miller 提出的「糖式类」提案所经常受到的一种批评，是认为它创造了太多的对象。具有 n 个方法的类在每次对象实例化时，除了实际的实例对象外，还会隐式创建 n 个特定于实例的闭包对象。对此 Miller 的立场是，去糖化只定义了可见的语义，而实现者可以自由开发技术，以避免创建闭包对象。然而委员会中有人对此表示怀疑，质疑实现者是否会开发此类优化。提案的另一个问题是缺乏对继承（或其他行为组合机制）的支持。为此 Miller 还开发了一些提案 [Miller 2010d, 2011a]，将组合性 Trait [Van Cutsem and Miller 2011c] 加入了他的类去糖化设计中。

对定义高完整性对象的支持，是委员会成员的首要任务。他们最关心的是可能试图窃取私人信息的恶意 Web 广告与 mashup。整个委员会都对此表示关切，但不一定要就此确定优先级。Waldemar Horwat [2010] 在 2010 年 9 月的 TC39 会议记录中指出：

> 小组内部关于目标的分歧：「高完整性」VS.「用更好的语法来支持人们已经在写的东西」VS. 也许有可能两者兼得。

Allen Wirfs-Brock 认为，如果让对象的创建变得不那么命令式，可能可以支持第二条目标。在经典的 JavaScript 中，最接近 Class 的是构造函数，它需要命令式地定义一个新对象的属性。对象字面量提供了一种更为声明式的方式来定义对象属性，但其缺乏与 ECMAScript 的内置类约定[93]相匹配的能力。也许对象字面量可以进行扩展，以更好地支持人们已经在写的东西，而不必引入「类」作为新的语言实体。

```
function tripleFactory(a, b, c) {
  return { // 这个对象字面量用于创建 triple 对象
    <proto: Array.prototype, // 由 proto 元属性设置继承的原型
```

```
    sealed>,                    // 用 Object.seal() 封住元属性
    0: a,
    1: b,
    2: c,
    var length const: 3, // var 会设置 [[enumerable]] 为 false
    // const 会设置 [[writable]] 为 false
    method toString() { // 方法是有函数值的数据属性
      // 并且其 [[ enumerable ]] 为 false
      return "triple(" + this[0] + "," + this[1] + "," +
this[2] + ")"
    },
    method sum(){ return this[0] + this[1] + this[2] }
  }
}
```

图 45. 基于 Wirfs-Brock 的 Harmony 扩展对象字面量提案的工厂函数。

在一组相关提案中，Wirfs-Brock [2011c; 2011d] 展示了如何扩展对象字面量，使其更为声明式，并消除在定义常规对象时使用 ES5 对象反射 API 的需求。例如，图 45 显示了在基于扩展对象字面量的*工厂函数*时，该如何定义具有显式原型、方法和私有属性的类。

Allen Wirfs-Brock 的提案还展示了对于扩展对象字面量的语法，该如何将其用作类定义的主体。在 2011 年 3 月的 TC39 演讲中 Wirfs-Brock [2011a] 提出，类定义应该能生成 ECMAScript 规范第 15 条[94]里内置库 Class 所使用的「构造函数、原型对象和实例对象」基本三要素，这在所有 ECMA-262 已有版本中都是通用的。与其将类定义去糖化为 lambda 表达式（糖化类）或一种新的运行时实体（受 Java 启发的类），不如将其去糖化为 JavaScript 程序员和框架作者们已经使用且熟悉的构造函数和原型继承对象。在会议上，大家对扩展对象字面量语法的许多细节有很大的意见分歧，但达成了一个宽松的共识，即核心类定义的语义，应该符合规范第 15 条中的构造函数、原型、实例三要素。

2011 年 5 月初，TC39 的 ES.next 特性冻结会议迅速临近，此时仍然有几个与类相关的稻草人提案在进行竞争。看起来委员会仍然未必有足够的共识，能使其中的某个提案被采纳。2011 年 5 月 10 日，Allen Wirfs-Brock 与 Mark Miller、Peter Hallam 和 Bob Nystrom 见了面。Hallam 和 Nystrom 是使用 Google 的 Traceur 转译器 [Traceur Project

2011b]，对 JavaScript 类支持进行原型设计的团队成员。他们的原型融合了 Wirfs-Brock 和 Miller 提案中的想法。会议的目标是取得足够的一致意见，以便能提出一份统一的提案。Bob Nystrom [2011] 在其会议报告中列出了许多一致意见，包括：

> ……构造函数、原型和实例这三要素，足以解决其他语言中的类所要解决的问题。Harmony 类语法的目的，并不是去要改变这些语义。相反地，它是要为这些语义提供一种简明而声明式的外表，以便体现程序员的意图，而非底层的命令式机制。
>
> ……对象是声明式和信息性的，函数则是命令式和行为式的。类的问题在于：「我们是否应将其建立在这些抽象的基础上。如果是的话，应该选择哪一个？」……
>
> 在我们的共识提案中，会通过结合这两种手段来解决这种宗教式的分歧：引入一种类似对象字面量的形式作为类体，再加上一个函数来作为构造器。

会后，Mark Miller [2011b] 创建了一份新的稻草人提案。尽管该提案中仍有许多细节缺乏共识，它在特性冻结会议 [TC39 2011b] 上仍然获得了接受。图 46 中作为示例的类定义，是基于 Miller 的特性冻结类提案而给出的：

```
class Monster extends Character {
  constructor(name, health) { // 构造器函数
    super();                  // 调用父类构造器
    public name = name; // 公有实例属性
    private health = health; // 私有实例变量
  }
  attack(target) { // 原型方法
    log('The monster attacks ' + target);
  }
  get isAlive() { // 原型 get 访问器
    return private(this).health > 0;
  }
  set health(value) { // 原型 set 访问器
    if (value < 0) {
      throw new Error('Health must be non-negative.')
    }
    private(this).health = value
  }
```

```
  public numAttacks = 0; // 原型数据属性
  public const attackMessage = 'The monster hits you!'; // 只读
}
```

图 46. 基于 Mark Miller [2011b] 统一化 Harmony Class 提案的类。

一个月后，Dave Herman [2011c] 在一篇题为「最小化的类」的 `es-discuss` 帖子中，对 class 提案的复杂性及其诸多分歧点给 ES.next 带来的时间风险表示了担忧。他提出了另一种最小化的设计，它只包含：带原型继承的类声明、构造器、声明式方法，并使用 `super` 关键字调用被继承的方法。被排除的是声明式属性、构造器属性、私有数据，以及其他任何有争议的内容。Herman 的建议在 2011 年 7 月的会议 [TC39 2011a] 上进行了讨论，但委员会决定将重点放在解决当时 Mark Miller 提案中的未决问题上。Brendan Eich [2012a] 后来写道：

> 去年夏天在 Redmond，最小化类有了一个很好的 TC39 支持子集。但我们当时卡在对「const 和 guard 使用前初始化的未来前景」的讨论上……

关于类的替代性设计 [Ashkenas 2011; Eich 2011a; Herman 2011a] 的持续在线讨论，促使 Dave Herman [2011d] 写了一份新的「最小类」稻草人提案。这份提案将 Herman 之前的帖子形式化，但增加了「静态」构造器数据和方法属性。在接下来的两次 TC39 会议上，几乎没有对 Herman 的最小化提案所进行的讨论，在解决计划中分歧的方面也没有什么进展。Brendan Eich [2012c] 对这个问题的描述如下：

> ……Waldemar 观察到的总体趋势是真实的：如果（提案的覆盖面）太小，就没有意义。而如果太大，我们又很难同意。我们需要「金发姑娘」（童话《金发姑娘和三字小熊》中的主人公，译者注）——恰到好处的温度和数量。

到 2012 年 3 月初，`es-discuss` 社区成员对于 TC39 明显无法完成 ES.next 中类的设计，表示出了越来越大的失望。Russell Leggett [2012] 在一篇题为「为类找到一个『安全』语法」[95]的文章中提出了这个问题：

> 我们是否能想出一种大家都认为「比没有好」的类语法，并注重于为将来的改进留出可能性呢？作为一种「安全语法」，这并不

> 意味着我们停止尝试寻找更好的语法。它只意味着如果我们还没有找到答案，那我们也仍然留着一些东西——这些东西我们可以在 ES7 中做得更好。

Leggett 的帖子在三天内收到了 119 个以正面为主的回复。它列出了一套「绝对最低的要求」，这与 Dave Herman 去年夏天的清单基本相同。Leggett 的贡献是创造了「安全学校」的隐喻。Allen Wirfs-Brock 对此立即表示支持，并创造了一份新的「最大化的最小」（max-min）版本 [Wirfs-Brock 2012d] 提案，用这个隐喻重新定义了 Herman 的最小化类提案。这里最大的技术变动，是移除了原提案中的构造器属性[96]。如果此时要将此「max-min」提案正式列入 2012 年 3 月 TC39 会议的议程，已经为时已晚。但 Allen Wirfs-Brock 和 Alex Russell 在会议结束时，领导了一次非正式讨论 [TC39 2012a]。总体来说，委员会对提案的接受度是积极的。但有几位成员就此表示担心，认为提案内容可能过少而不值得就此费心，或者可能会对他们考虑的未来扩展产生不利影响。当时没有试图就该提案达成共识，但 Wirfs-Brock 和 Russell 表示，任何更详细的内容都不可能进入 ES.next。

这份 max-min 提案正式列入了 2012 年 5 月的会议议程，并在会上进行了类似的讨论 [TC39 2012b]，其结果是类似的。与会人员正逐步就该提案达成共识，但还有一些关键人物缺席。由于时间上的压力，与会者一致认为，已经可以就原型和初步规范草案开展工作了。到 7 月会议 [TC39 2012c] 时，Allen Wirfs-Brock 已经写好了 max-min 类提案的规范文本，并准备了一套演示文稿 [Wirfs-Brock 2012b]，列举了他遇到的每项设计决策。他带领委员会逐条审查了每项决策，并记录了对某一备选方案的接受或共识。这种方法回避了就整个提案达成共识的问题，但却让委员会在细节设计层面参与了共识的形成。ES.next 规范的下一份草案 [Wirfs-Brock et al. 2012b, c] 包含了完整的 max-min 类设计，其中纳入了 7 月会议上做出的决策。对此没有人表示反对。

然而在 2014 年夏天，随着浏览器 JavaScript 引擎开发者开始实现 ES6 的类，确实出现了一条重要的反对意见。ES6 工作的长期目标之一，是提供一种「子类化」内置类的方法，如 `Array` [Kangax 2010] 和 Web 平台的 DOM 类。Allen Wirfs-Brock [2012c; 2012e] 写了一份 Harmony 稻草人文档，描述了为什么传统的 JavaScript 内置构造函数在进行子类化时会存在问题。内置的构造函数通常是使用语言实现所用的原生语言（如 C++）来定义的。它们会分配和初始化私有的对象表示，这

些私有对象的特殊结构也会被相关的内置方法所获知，这些方法也是用实现语言定义的。当使用 `new` 运算符直接调用内置构造函数时，这种方法是有效的。但当使用 JavaScript 特有的原型继承方案来「子类化」这样的构造函数时，`new` 运算符会被应用于子类构造函数（通常用 JavaScript 编码）上。它所分配出的是一个普通对象，而不是被继承的内置方法所期望的私有对象表示。Wirfs-Brock [2013] 在确定 max-min 类的语义时，试图避免这个问题。`new` 的语义被分割成了单独的分配阶段和初始化阶段。对象分配是由 `new` 首先调用一个特别命名的 `@@create` 方法来进行的。该方法通常由内置的父类提供，而不会被子类覆盖。对象初始化发生在分配之后，与子类的构造函数相协调。它通常会对其父类构造函数进行 `super` 调用，以执行所有特定于父类的必要初始化，然后再执行所有特定于子类的必要初始化。如果编码得当，这可以使内置的父类在将对象传递给子类构造函数之前，分配出其特殊的私有对象结构。子类构造函数可以使用其初始化代码，将子类属性添加到父类提供的对象中。

2014 年发现的问题在于，`@@create` 方法创建的对象是未初始化的。某个错误或恶意的类构造函数，可能会在未初始化的对象上调用内置的父类方法（很可能由 C++ 实现）——这可能导致灾难性的后果。Wirfs-Brock 曾假设所有这类对象都会在内部跟踪它们的初始化状态，并且需要相应的内置方法，来检查它们是否被应用了到一个未初始化的对象上。Mozilla 的 Boris Zbarsky [2014] 指出，浏览器中有数千种这样的方法，而在区分两阶段的设计中，需要为每个方法更新每个浏览器的 DOM 规范和实现。这促使了单阶段分配 / 初始化设计 [Wirfs-Brock et al. 2014c, d] 和另一份提案 [Herman and Katz 2014] 的发展。这份提案保留了两个阶段，但会将构造器参数传递给 `@@create` 方法和构造器。在 2014 年剩余的时间里，委员会对这些方案和其他替代方案进行了激烈的辩论。在某段时间，共识的缺乏一度可能推迟原定于 2015 年 6 月发布的 ES6，甚至迫使从该版本中完全移除类。然而在 2015 年 1 月，TC39 围绕单阶段设计的变体达成了共识 [TC39 2015a; Wirfs-Brock 2015b]。这一经验再次坚定了 TC39 的决心，要求更多、更早地由实现者对 ES6 后的新特性进行反馈。

## 模块§

ES4 设计的复杂部分之一，就是用于构建大型程序和库的「包和命名空间」结构。当新版 ES4 被放弃时，人们已经发现这些机制存在重大问题 [Dyer 2008b; Stachowiak 2008b]，它们显然不适合进入 Harmony。而当时有影响力的 JavaScript 开发者们所使用的，还是基于模块模式而缺乏泛用性的模块化解决方案 [Miraglia 2007; Yahoo! Developer Network 2008]。2009 年 1 月，Kris Kowal 和 Ihab Awad 向 TC39 [2009c] 提交了一份受模块模式启发的设计 [Awad and Kowal 2009; Kowal and Awad 2009a]。他们的设计最终演变成了 Node.js 中使用的 CommonJS 模块系统。

Kris Kowal 和 Ihab Awad 在他们最初的提案和随后的修订版 [Kowal 2009b; Kowal and Awad 2009b] 中，纳入了一些语法糖式的替代方案。这些方案可能会覆盖他们的模块设计，而不会改变提案的动态语义。Awad [2010a; 2010c] 随后开发了一份不同的提案，这份提案借鉴了 CommonJS 上的工作，以及 E 语言 [Miller et al. 2019] 的 Emaker 模块。这些 Emaker 模块正被与安全 ECMAScript 相关的 Caja 项目 [2012] 所使用。在 TC39 内部，这些提案被称为「一等公民式模块系统」，因为它们将模块表现为动态构造出的一等公民式运行时实体，这提供了一种新的计算抽象机制。例如在 Awad 的提案中，一个模块的多个实例可能同时存在，每个实例用不同的参数值初始化。

Brendan Eich [2009c] 描述了一种替代方法：

> Harmony 中的替代方案是一种特殊的语法形式。比如说 import 指令，它可以在程序解析（而非执行）时进行分析。这样语言实现可以在执行前预先加载好所有的依赖关系，以免在导入（或出现延迟的数据依赖）时阻塞。否则就要使用一种较不方便的非阻塞导入，以保留 JS「运行到完成」的执行模式。

这种替代方案被称为「静态」或「二等公民式模块」系统。这种模块系统提供了使应用代码结构化的机制，而非定义出新的计算抽象机制。对此 Sam Tobin-Hochstadt [2010] 解释说：

> ……在一个有状态的语言中，你会希望能在不改变其行为的情况下，将程序划分成模块。对于有一段有状态的代码，在你把它移到自己的模块中后，所造成的影响不应该多于任何其他重构。如果你需要反复创建新状态，ES 也提供了不错的机制。同样地，如

果有一个导入了 A 的模块，你可以把它拆分成两个都导入了 A 的新模块。像这种重构也不应该改变程序的工作方式。

Dave Herman 和 Sam Tobin-Hochstadt 为二等公民式 Harmony 模块开发了「简单模块」设计 [Herman 2010b, c, f; Herman and Tobin-Hochstadt 2011; Tobin-Hochstadt and Herman 2010]，其基本思想在于将模块视作「可共享词法绑定的代码单元」。新语法将用于划分出代码单元，并确定出哪些绑定将被共享。在 Awad [2010b] 建议 TC39 将工作重点放在 Herman 和 Tobin-Hochstadt 的提案上之前，TC39 对这两种方法的优点进行了广泛的讨论。

他们的设计中具有 `module` 声明，其中会为模块分配一个词法标识符。这要么会引入模块代码，要么会确定包含相应代码的外部资源。而对于具备 `export` 关键字前缀的声明，其绑定将被暴露到模块外部。例如：

```
// 最早的 Harmony 简单模块提案

module m1 { // 一个内部模块
  export var x = 0, y=0;
  export function f() {/* ... */};
}
module m2 { // 同个源文件内的另一个内部模块
  export const pi = 3.1415926;
}
// 用于确定外部模块的字符串字面量
module mx = load "http://example.com/js/x.js";

// ... 后续代码可导入并使用来自 m1，m2 和 mx 中的绑定
```

模块声明也可以进行嵌套。一个形如 `x.js` 的外部模块，可以只包含一个模块主体，而不必以模块声明语法包围它。`import` 声明用于使某个模块所导出的绑定，能在词法上被导入它的模块所访问。使用上述示例模块的代码，可能会有如下的 `import`：

```
// 最早的 Harmony 简单模块提案

import m1.{x, f};    // 从 m1 导入两个绑定
import m2.{pi: PI}; // 导入一个绑定并重命名，以便于本地访问
import mx.*;         // 导入所有由 mx 导出的绑定
import mx as X;      // 将 X 本地绑定到以 mx 导出字段为属性的对象
```

通过模块声明、字符串字面量形式的外部模块标识，以及声明式的导出 / 导入定义，可以静态地确定一组由相互依赖的模块组成的封闭集合。这些模块之间的共享词法绑定，可以在执行代码之前进行链接。循环依赖也是允许的。当执行开始时，模块会按照规定好的确定性顺序进行初始化。如果有任何无法初始化的循环依赖关系，TDZ 死区会确保抛出运行时错误。

模块语法发生了演变 [Herman et al. 2013]，但「模块具备共享词法绑定，且可静态链接」这一基本思想仍然存在。主要的语法变化之一，是取消了显式的模块声明语法、模块标识符，以及内部 / 嵌套模块。每个源文件对应一个 Harmony 模块，其中使用字面量形式的字符串资源标识符来进行识别。模块标识符的取消，需要改变 `import` 语法。另外通配符导入也被取消，因为它太容易出错。通配符导入被替换成了另一种形式，这种形式会将一组开放式的导入指令暴露为「单一命名空间下的对象属性」，而非作为单独的词法绑定。对于前述中的 `import` 示例，其基于最终版语法的表达是这样的：

```
// ES2015

import { x, f } from "m1.js"; // 从 m1 导入两个被导出的绑定
import { pi as PI } from "m2.js"; // 导入一个绑定并重命名，以便于本地访问
import * as X from "mx.js"; // 将 X 本地绑定到命名空间对象，其属性映射为 mx.js 所导出的字段

// 新增的导入形式
import "my.js"; // 仅为初始化副作用而导入 my.js
import z from "mz.js"; // 导入由 mz.js 所导出的唯一默认绑定
```

`module` 声明的取消和默认绑定 `import` 形式的增加，均属于设计的后期变化。Node.js 的普及出乎意料地迅速，它将 CommonJS 模块广泛暴露在了 JavaScript 开发者社区中。TC39 为此收到了负面的社区反馈 [Denicola 2014]，并担心 CommonJS 模块事实上的标准化，可能会给 Harmony 设计蒙上阴影。TC39 为此增加了 `export default` 形式，以适应那些习惯于在许多 CommonJS 模块中使用单体导出设计模式[97]的开发者。TC39 模块倡导者们也开始向 Node.js 开发者布道 [Katz 2014] Harmony 模块。

最初的「简单模块」提案包含了模块加载器 [Herman 2010e] 的概念，它提供了将模块整合到运行中的 JavaScript 程序时的语义。其目的在于由 ECMAScript 规范来定义出：模块的语言级语法和语义、模块加载的运行时语义，以及模块加载器的 API。这其中模块加载器的 API，能为 JavaScript 程序员提供「控制和扩展加载器语义」的机制。加载过程最终被设想 [Herman 2013b] 为一条由五个阶段组成的流水线，包括规范化、解析、获取、翻译和链接。加载器首先会对模块标识符进行规范化处理。然后它会通过对模块源码的检索和预处理，确定模块的相互依赖性，将导入和导出联系起来，最后再初始化相互依赖的模块。模块加载器的设计目标是高度的灵活性，以完全支持 Web 浏览器的异步 I/O 模式。在 2011 年的 JSConf 上，Dave Herman 展示了 [Leung 2011] 一个概念验证性的模块加载器。它扩展了加载过程中的翻译阶段，将 CoffeeScript 和 Scheme 代码加载为了运行在 JavaScript 网页之中的模块。

为了充分理解模块加载过程和该如何确定它，Dave Herman 与 Mozilla 的 Jason Orendorff 合作，使用 JavaScript 代码实现了一个模块加载器参考实现的原型 [Orendorff and Herman 2014]。2013 年 12 月，Herman [2013a] 完成了对 Orendorff 的 JavaScript 代码的初步改写，使其变成了规范伪代码。2014 年 1 月，Allen Wirfs-Brock [2014a] 将伪代码初步整合到了 ES6 草案中。但 Wirfs-Brock 发现模块加载器的异步性质，给 ECMAScript 规范增加了新的复杂性和潜在的不确定性。这种情况因加载器 API 而变得更糟，因为它允许用户程序在模块加载过程中注入任意的 JavaScript 代码。到 2014 年年中，异步模块加载的额外复杂性和 API 中一连串难以解决的设计问题，似乎已经危及了 ES6 在 2015 发布版本的目标。

在开发简单模块提案的早期阶段，Allen Wirfs-Brock [2010] 曾注意到模块作用域和链接的语义，可以从加载器管道中分离出来。在之前的 ECMA-262 版本中，规范已经定义了 JavaScript 源码的语法和语义，但并未涉及该如何访问它。这是由托管 JavaScript 引擎的环境来承担的责任。在 2014 年 9 月的 TC39 会议 [TC39 2014b] 上，Wirfs-Brock 认为类似的方法也可以适用于模块，这样 ECMA-262 就不需要包含模块加载管道的规范了。如果 ECMA-262 假定模块的源码都已经存在，那么只要规定各独立模块的语法和语义，以及该如何「将被导入和导出的绑定联系起来」的语义就足够了。浏览器等宿主环境可以提供异步加载管道，但其定义将与语言规范解耦。要移除加载器管道，也意

味着要移除加载器 API。TC39 接受了这一观点。Wirfs-Brock 也得以在 2014 年 10 月的规范草案 [Wirfs-Brock et al. 2014b] 中，纳入了基本完整的语言级模块规范。模块语义与加载器管道的分离，使得 WHATWG 能够专注于确定 ECMAScript 模块该如何与 Web 平台 [Denicola 2016] 集成。

## 箭头函数§

ES2015 引入了一种简洁的函数定义表达形式，通常称之为「箭头函数」。箭头函数的写法是以形参列表为起始，然后是 `=>` 标记和函数体。例如：

```
(a, b) => { return a + b }
```

如果只有一个形参，那么可以省略括号。而如果函数体是单条 `return` 语句，还可以省略括号和 `return` 关键字。例如：

```
x => x /* 一个 identity 函数 */
```

与其他函数定义形式不同的是，箭头函数不会重新绑定 `this` 和其他函数作用域内的隐式绑定。这使得箭头函数在「内层函数需访问其外层函数的隐式绑定」的情况下，显得非常方便。

设计箭头函数的主要动机，在于开发者经常需要编写冗长的函数表达式，以此作为平台和库 API 函数的回调参数。在 JavaScript 1.8 中，Mozilla [2008b] 实现了[98]「表达式闭包」，它保留了对`function` 关键字的使用，允许使用无括号的单个表达式体。TC39 讨论了一些类似但较短小的表示法，用诸如 λ、*ƒ*、\ 或 # 等符号 [Eich 2010b; TC39 Harmony 2010c] 来代替函数，但未能就其中任何一种方法达成共识。

TC39 [Herman 2008] 同时也对提供具有精简语义的「lambda 函数」感兴趣，比如支持*消栈的尾调用*和 Tennent [1981] 一致性原则[99]。其支持者们认为，这样的函数将会在实现由语言或库所定义的控制抽象时有

所用处。在 Harmony 进程早期，Brendan Eich [2008a] 在 `es-discuss` 上的一篇讨论贴中， 提出了一个最初由 Allen Wirfs-Brock 所提出的建议，即基于 Smalltalk 块语法的启发，采用一种简洁的 lambda 函数语法。例如 `{|a，b| a+b}` 就相当于 Herman 的 `lambda(a,b){a+b}`。Eich 的帖子引发了一场大规模但没有结论的线上讨论，话题涉及与（某种可能的）简明函数特性所相关的方方面面。作为关键总结，可以认为其中许多语法的灵感会带来解析或可用性上的问题，而且 JavaScript 的非本地控制转移语句——`return`、`break` 和 `continue`——会显著地使编写控制抽象的机制变得更加复杂。大多数 TC39 成员和 `es-discuss` 订阅者似乎主要对简洁的函数语法更感兴趣，而非对 Tennent 一致性感兴趣。

在这之后的 30 个月里，这方面都没有出现什么重大进展，直到 Brendan Eich [2011f; 2011g] 撰写了两份替代性的稻草人提案为止。这两份提案之中，有一份设计的是「箭头函数」，它参照了 CoffeeScript 中的类似特性。这份提案中有 `->` 和 `=>` 两种函数，它们具备各种语法和语义上的差异和选项。而另一份提案设计的，则是以 Smalltalk 和 Ruby 的块为模型的「块级 lambda」，它还支持 Tennent 一致性。在随后的 9 个月里，这两项提案及其备选方案在 `es-discuss` 和 TC39 会议上得到了广泛的讨论。有人担心如果要支持解析箭头函数，现有的 JavaScript 实现是否易于更新。这里的问题是箭头符号出现在整个结构的中间，而且它前面还有一个形参列表，因此可能会被有歧义地解析为括号表达式。对于块级 lambda 提案，有人担心 [Wirfs-Brock 2012a] 它所创建出的用户定义控制结构，并不能充分而完整地与内置的语法控制结构相集成。Brendan Eich 总体倾向于块级 lambda 提案，但随着 2012 年 3 月 TC39 会议的临近，他认为箭头函数更有可能被委员会接受。在会议上 [TC39 2012a]，他向委员会介绍了一套关于箭头函数最终设计基本特征的共识性决定 [Eich 2012b]。

## 其他特性§

除上述已经讨论过的内容外，重要的新语言特性还包括如下：

- 对象字面量的增强，包括计算属性名和简洁的方法语法。
- 在对象与数组的初始化声明和赋值运算符中使用解构。
- 形式参数增强，包括剩余参数、可选参数默认值，以及参数解构。
- 受 Python 启发的迭代器和生成器，但与其有显著的不同。
- `for-of` 语句，以及在新场景和改进后的场景下普遍使用的迭代器协议。
- 在字符串和正则表达式中支持完整的 Unicode。
- 支持嵌入领域特定语言（domain specific language）的模板字面量。
- 作为属性键使用的 `Symbol` 值。
- 二进制和八进制数字字面量。
- 消栈的尾调用[100]。

语言内置库的增强包括：

- 新的 `Array` 方法。
- `of` 和 `from` 构造器方法约定，用于创建数组和其他集合对象。
- 类型数组类，包括用于操作二进制数据的 `DataView` 和 `ArrayBuffer`。它们都基于 Khronos Group [2011] 规范中之前实现出的浏览器宿主对象，但与语言的其他部分有了更好的集成。类型数组现在还支持了大多数的 `Array` 方法。
- `Map` 和 `Set` 这类具有键的集合，以及 `WeakMap` 和 `WeakSet`。
- 额外的 `Math` 和 `Number` 函数。
- 用于复制对象属性的 `Object.assign` 函数。
- 用于延迟访问异步计算值的 `Promise` 类。
- 反映内部元对象协议的 `Reflect` 函数。

## 延期和被放弃的特性§

在 ES6 的开发过程中，TC39 还考虑了许多稻草人特性提案，但它们最终没有被纳入为 ES2015 的特性。这其中许多提案在最初提出后不久就被拒绝，但其他一些则曾属于重要的开发工作，有些甚至在最终

被从版本中移除之前，已经推进到了成为被接受的 Harmony 提案的程度。在被削减掉的内容中，有一些提案被放弃，另一些则被推迟，以便开展更多的工作，并可能考虑纳入未来的版本中。截至 ES2015 完成前不久，被削减的重要特性和开发工作主要包括以下内容：

- **推导式** [Herman 2010a, d, 2014a; TC39 2014a] 推导式原本可提供一种更简洁而声明式的方式，来创建出一个初始化后的数组，或定义出一个生成器函数。它基于 Python 和 JavaScript 1.7/1.8 中的类似特性。
- **模块加载器 API** [Herman 2013b] 模块加载器 API 原本可让 JavaScript 程序员动态介入模块加载器的处理过程。程序可能会使用该 API 来完成一些处理，比如在加载过程中插入一个转译器，或支持模块的动态定义。这个 API 和模块加载器一起被推迟。
- **Realms API** [Herman 2014b] Realm API 原本可使 JavaScript 程序员能在新的 Realms 中创建、补充和执行代码，它与模块加载器 API 密切相关。这一特性被推迟，以进行额外的设计工作。
- **模式匹配** [Herman 2011e; Rossberg 2013] 解构的通用化，它原本将包括受 Haskell 启发的可驳式匹配（refutable matching）。
- **Object.observe** [Arvidsson 2015; Klein 2015; Weinstein 2012] 一种复杂的数据绑定机制，可以在受监控对象的属性被修改时产生事件。
- **并行 JavaScript** [Hudson 2012, 2014] 又名 River Trail，是英特尔和 Mozilla 的一个联合项目，旨在使 JavaScript 程序员能够明确地利用处理器的 SIMD 能力。
- **值对象** [Eich 2013] 其目标是提供一种通用性的支持，以便定义出类似 Number 和 String 的新原始数据类型（包括运算符重载）。这可以允许库实现十进制小数、大整数等特性。
- **Guards** [Miller 2010c] 为声明添加的类似于类型的注解，可对其进行动态验证。

## Harmony 转译器§

转译器在 Harmony 特性的开发、测试和社区普及化过程中发挥了重要作用。在标准完成或浏览器完全支持之前，它们就能实现新特性的生产级使用。转译器对 JavaScript 开发者社区快速采用 ES2015 至关重要。支持 Harmony 的重要转译器包括：

- **Narcissus** [Eich et al. 2012] 是一个以 JavaScript 为宿主的 JavaScript 引擎，被 Mozilla Research 用于 ES6 语言实验。
- **Traceur** [Hallam and Russell 2011; Traceur Project 2011a] 是谷歌开发的一款转译器，用于实验 ES6 的早期特性。Traceur 提供了 ES6 语义的高保真实现，但由此产生的运行时开销使其在生产使用中缺乏吸引力。
- **Babel** [2015] 原名 6to5，是由 Sebastian McKenzie 开发的。当时 Sebastian McKenzie 是一名 17 岁的开发者，生活在澳大利亚的农村：「在 2014 年 9 月 28 日，我在复习高中考试时第一次提交代码到了 GitHub，内容是我当时在做的一个 JavaScript 库。」[McKenzie 2016] Babel 通过牺牲与规范草案之间的完全语义一致性，来将运行时开销最小化。它使人们能提前使用 ES2015 和其他实验性的 JavaScript 特性，使大多数 ES2015 级 JavaScript 代码能在旧的浏览器（或仅支持 ES5 的平台）上运行。然而，一些使用 Babel 的开发者开始依赖于实验性特性和不正确的语义。相对于后来的标准 ECMAScript 特性，还有一些被依赖的 Babel 变体特性已经过时（废弃）。这使得向原生实现的过渡变得更加困难，并且在少数情况下还造成了限制 TC39 设计灵活性的遗留问题。
- **TypeScript** [Microsoft 2019] 是微软一款使用自由许可证的语言产品，最初以带有 ES6+ 特性的 ES5 为目标，后来增加了 ES2015 作为编译目标。TypeScript 最重要的功能，是一个可选的静态分析类型系统与类型注解。它可以编译成人们惯用的动态类型 JavaScript 代码。在 2020 年，TypeScript 是编写带类型注解的 JavaScript 时的事实标准 [Greif and Benitte 2019]。

转译器的生产级使用（尤其是 Babel 和 TypeScript），是许多 JavaScript 开发团队内部大型文化转型的一部分。在这些团队中，JavaScript 已经被近似于当作传统的、具备开发和部署构建工具链的 AOT 编译型语言，而不是作为一个动态执行环境，加载并直接执行程序员的原始源代码。

# 完成 ECMAScript 2015§

在 2015 年 3 月的会议上，TC39 [2015b] 批准了当时的候选规范 [Wirfs-Brock et al. 2015b, c]，将其提交给了 Ecma GA 大会进行最终批准。Ecma GA 在 2015 年 6 月的会议上投票批准了它 [Ecma International 2015a]，并立即发布了 ECMA-262 第 6 版，是为《ECMAScript 2015 语言规范》[Wirfs-Brock 2015a]。

ECMAScript 2015 的开发和发布历时近 7 年，数百人为其开发做出了贡献。从 2008 年 7 月的会议（Harmony 工作开始之处）开始，到 2015 年 3 月的会议（候选规范获得批准之处）为止，委员会共召开了 41 次 TC39 会议。有 145 人亲自或通过电话参加了这些会议，具体参与程度不一。ES2015 的开发与 ES5/ES5.1、《ECMA-402 ECMAScript 国际化 API》、《ECMA-404 JSON 数据交换格式》以及 Test262 测试验证套件的开发相重叠。一些与会者的主要兴趣是其中的一项或多项工作。在 145 名与会者中，有 62 人只参加了一次会议，他们通常以观察员身份列席。

TC39 主席 John Neumann 和 Ecma 秘书长 István Sebestyén 为项目提供了行政上的支持，确保了会议的顺利进行。项目编辑 Allen Wirfs-Brock 在项目过程中发布了 38 份规范草案 [TC39 Harmony 2015]。有 7 人（图 47）实质上属于整个项目的技术贡献者。另有 35 名与会者（图 48）参加了 5 至 24 次会议，其中大多数人对项目作出了重要的技术贡献。在 ES2015 的开发过程中，数百名 JavaScript 开发者社区成员向 `es-discuss` 邮件列表 [TC39 et al. 2006] 发布了超过 36000 条消息，并在 TC39 的 bug 跟踪系统 [TC39 et al. 2016] 中，提交了 4000 多条与 ES2015 规范草案有关的工单。

| | |
|---|---|
| Allen Wirfs-Brock (Project Editor) | Microsoft, Mozilla |
| Brendan Eich | Mozilla, invited expert |
| Mark S. Miller | Google |
| Waldemar Horwat | Google |
| Dave Herman | Northeastern Univ, Mozilla |
| Douglas Crockford | Yahoo!, PayPal |

| | |
|---|---|
| Erik Arvidsson | Google |

图 47. TC39 技术贡献者，他们在整个 ES2015 开发工作中表现活跃。在此期间，每人至少参加了 41 次 TC39 会议中的 30 次。2009 年 5 月，Arvidsson 首次参加。Crockford 最后一次参加是在 2014 年 4 月。其余的人从头到尾都参加了项目。

| | | | |
|---|---|---|---|
| Sam Tobin-Hochstadt (24) | Andreas Rossberg (13) | Rafael Weinstein (10) | Chris Pine (7) |
| Alex Russell (21) | Oliver Hunt (12) | Jeff Dyer (8) | Mike Samuel (6) |
| Luke Hoban (20) | Norbert Lindenberg (12) | David Fugate (8) | Ihab Awad (5) |
| Cormac Flanagan (18) | Sam Ruby (12) | Domenic Denicola (7) | Reid Burke (5) |
| Yehuda Katz (17) | Brian Terlson (12) | Rick Hudson (7) | Andreas Gal (5) |
| Rick Waldron (17) | Sebastian Markbage (11) | Jafar Husain (7) | Peter Jensen (5) |
| Eric Ferraiuolo (15) | Jeff Morrison (11) | Dimitry Lomov (7) | Pratap Lakshman(5) |
| Tom Van Cutsem (14) | Rob Sayre (10) | Ben Newman (7) | Nicholas Malsakic (5) |
| Nebojsa Ćirić (13) | Matt Sweeney (10) | Caridy Patino (7) | |

图 48. 在制定 ES2015 期间，经常参加 TC39 会议的技术贡献者。这些数字反映了他们参加了多少次会议。

在 ES6 的开发过程中，人们对 TC39 的兴趣和参与度急剧增加，并在其完成后继续增加。2008 年 7 月，TC39 的第一次 Harmony 会议仅有代表 8 个组织的 13 人参加。2015 年 7 月的会议是在 ES2015 发布一个月后举行的，有代表 15 个组织的 34 名个人参会者（有些是远程参会）。在 2019 年 7 月的 TC39 会议上，有代表 24 个组织的 76 名参会者（46 人到场，30 人远程）。

## 准备 ES6 之后的未来§

2013 年和 2014 年，随着 ES6 开发工作的结束，TC39 开始考虑未来版本的开发该如何进行。ES6 进程中的一个关注点在于，其中一些特性的设计是在它们「能出现在已发布的 ECMAScript 标准中」之前几年就完成了的。这与大多数主流浏览器厂商所采用的「常青浏览器」概念相冲突。常青浏览器每隔几周就会更新一次，使错误修复和新特性能尽快推出。大多数 TC39 成员认为，ECMAScript 标准需要更快的更新周期，以便更好地适应浏览器的快速发展。

为此，委员会提出了以一年为发布周期的提议。这将使各个新特性在标准中迅速变得可用。每年发布版本还可以使规范中的错误迅速得到纠正，并消除了多年来对长篇勘误表的需要。按照标准组织的规范，每年一次的发布周期是非常快的，但 Ecma 同意接受这个时间表。

以年为单位进行更新，将要求 TC39 在开发新语言特性方面更加规范。有些设计工作仍然需要多年才能完成，因此这需要一个流程，以适应跨越多个年度发布周期的特性开发项目，并能协调不同特性之间重叠的开发周期。还有人担心，ES6 过于依赖一位编辑来完成大部分规范的编写工作。要想成功实现每年发布，倡议者需要为自己的特性进行大部分的规范编写。

Rafael Weinstein 和 Dimitry Lomov 发表了一份提案 [TC39 2013c; Weinstein and Lomov 2013]，建议在开发过程中，新特性提案要经过五个成熟阶段。后来 Weinstein 与 Allen Wirfs-Brock 合作，进一步定义和文档化了这一过程。附录 Q 是对新流程和发展阶段的描述。从 2014 年起，TC39 在 ES6 之后的所有工作中都遵循这一程序。截至 2020 年 6 月发表本文，TC39 在每年 6 月都成功发布了 ECMAScript 规范的新版本。

# 总结§

JavaScript 是一门以低预期要求来创建的语言。它的初衷是作为 Java 在浏览器内的一个辅助工具，适合初级网页开发者和兼职程序员使用。然而在很短的时间内，它就（在浏览器里）超过了 Java，成为了交互式网页的主要语言。尽管 JavaScript 发展的前二十年里充满了增强、改进、重新设计或取代它的失败尝试，但到这一时期结束时，JavaScript 已成为了世界上使用最广泛的编程语言——而且还不仅仅用于网页。除了使用 Node.js 和其他宿主构建的服务器应用外，JavaScript 还被用于构建桌面应用、移动设备应用、健身追踪器、机器人和众多嵌入式系统。它甚至是詹姆斯·韦伯太空望远镜的一部分。这个望远镜使用 Nombas 的 ES1 级嵌入式 JavaScript，作为其板载控制软件的一部分 [Dashevsky and Balzano 2008]。

JavaScript 的崛起是必然的吗？基于 Web 和浏览器博弈论中提出的可操作性要求，结论可能会倾向于演化出一种单一的主流网页编程语言，但并没有特别的理由说明这种语言必须是 JavaScript。其他语言也本可以填补这一角色。实际上纵观 JavaScript 的历史，有很多地方的结果都可能是不同的：

- 如果 Marc Andreessen 没有倡导开发浏览器脚本语言，会怎么样呢？
- 如果 Sun 公司的 Bill Joy 没有支持启动开发 Mocha 来作为 Java 的补充，会怎么样呢？
- 如果把开发 Mocha 的任务交给 Brendan Eich 以外的人，会怎么样呢？
- 如果 Eich 是一位更有经验的语言设计者或实现者，并总结认为 10 天内完成 demo 是一件不可能完成的任务，会怎么样呢？
- 如果 Eich 的编程能力不够，或者在语言设计上的野心太大，导致没能在 10 天内创建出 Mocha 的 demo，会怎么样呢？
- 如果 JavaScript 最初的设计中没有包含一等公民式的函数，会怎么样呢？
- 如果 Sun 或 Netscape 公司花大力气把 Java 与 HTML 更好地结合起来，而不是把 Java 作为一个孤立的环境来托管，会怎么样呢？
- 如果微软没有实现 JScript，而是更大力地推广它的 Visual Basic 替代方案，会怎么样呢？
- 如果微软在取得 90% 以上的浏览器市场份额后，继续投资浏览器语言技术，会怎么样呢？

- 如果 Macromedia/Adobe 推动将 ActionScript 2 或 3 作为浏览器的官方标准，而不是参与新版 ES4 的重新设计，会怎么样呢？
- 如果 TC39 内部没有出现反对新版 ES4 的声音，会怎么样呢？

如果，如果，如果……但这些事情其实都没有发生。实际上，面对嘲笑和有时甚至是激烈的批评，一个世代的浏览器实现者、引擎开发者、框架设计者、标准贡献者、工具构建者和 Web 应用程序员们，都找到了务实的方法来继续使用和增强 JavaScript，而且通常还不会破坏 Web。

Brendan Eich 在 2011 年一次名为「JSLOL」[Eich 2011e] 的会议演讲中，是这么描述 JavaScript 的：

> 最早他们说 JavaScript 没法做「富互联网应用」。
>
> 然后他们说 JavaScript 没法快起来。
>
> 然后他们说 JavaScript 没法修复语言问题。
>
> 然后他们说 JavaScript 没法做多核与 GPU 运算。
>
> 他们每次都错了！
>
> 我建议：**永远押宝在 JS**。

# 致谢§

HOPL-IV 项目委员会成员们协助了两位作者（图 49）。他们提供了修改指导、LATEX 技巧和详尽的评审，并对本文草稿做出了有价值的反馈。

以下同事参与了 JavaScript 和 ECMAScript 的开发，他们为本文所讨论的事件与技术提供了信息：Douglas Crockford、Jeff Dyer、Richard Gabriel、Bill Gibbons、Gary Grossman、Lars T. Hansen、Dave

Herman、Graydon Hoare、Yehuda Katz、Shon Katzenberger、Peter Kukol、Pratap Lakshman、Mark S. Miller、István Sebestyén、Mike Shaver、Brian Terlson、Tom Van Cutsem、Herman Venter、Rick Waldron 和 Robert Welland。

在稿件编写的各个阶段，对部分或全部稿件提供编辑反馈的 Beta 读者们包括：Jory Burson、Douglas Crockford、Jeff Dyer、Richard Gabriel、Lars T. Hansen、Dave Herman、Pratap Lakshman、Mathias Bynens、Axel Rauschmayer、Jonathan Sampson、Jon Steinhart、Tom Van Cutsem、Herman Venter、Rick Waldron、Rebecca Wirfs-Brock 和 Joseph Yoder。

Richard Gabriel、Rebecca Wirfs-Brock 和 Joseph Yoder 都参加了耗时多日的研讨会，在研讨会上，我们用全面的通读微调了论文的结构和语言。

记忆是不可靠的。因此准确的历史取决于能否获得原始文件。互联网档案馆和 Ecma 国际的内部档案，为本文提供了重要的原始资料。特别地，如果没有 Ecma 现任秘书长 István Sebestyén 的热情支持，本文是不可能完成的。Sebestyén 博士不仅确保了能让我们访问 Ecma 的内部档案，而且和我们一样都认为 Ecma 与 TC39 和 ECMAScript 有关的大部分文件档案，也都需要能通过网络公开访问。Ecma 的 Patrick Charollais 协助建立了 https://www.ecma-international.org/archive/ecmascript 网页。

最后，Allen Wirfs-Brock 要感谢 Pratap Lakshman 在 2007 年 1 月写的那封邮件。这是通向本文道路的起点。

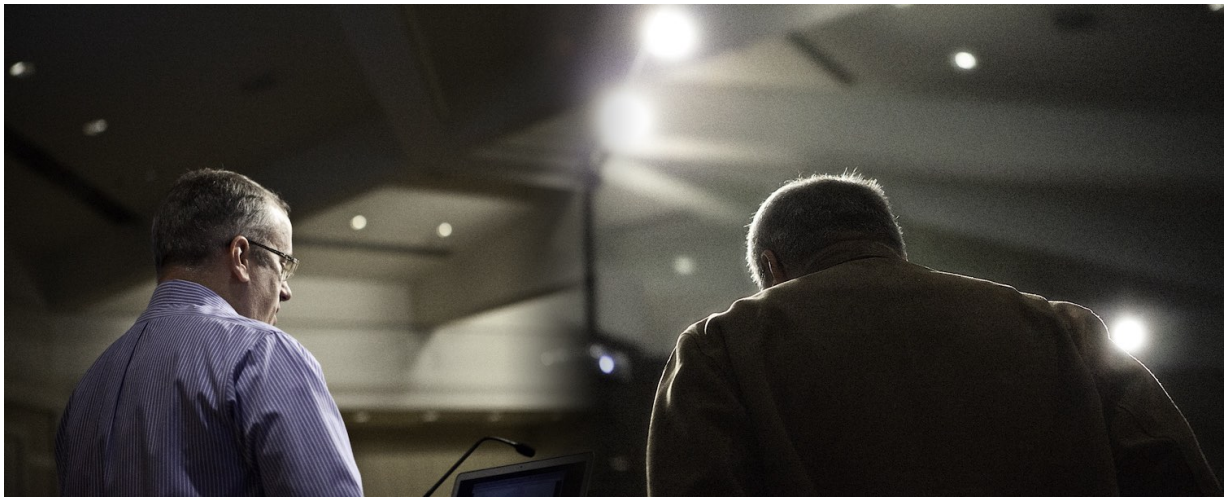


图 49. Brendan Eich 和 Allen Wirfs-Brock，2011 年。照片由 Richard P. Gabriel 提供。