

JavaScript Enlightenment



JavaScript 启示录

O'REILLY®

[美] Cody Lindley 著
徐涛 译

 人民邮电出版社
POSTS & TELECOM PRESS

O'REILLY®

JavaScript 启示录

[美] Cody Lindley 著
徐涛 译

点此购买: <http://item.jd.com/11416981.html>



更多精彩, 请关注人邮IT书坊 (ptpressitbook)
赠书福利不间断, 期待您的加入

人 民 邮 电 出 版 社
北 京

图书在版编目 (C I P) 数据

JavaScript启示录 / (美) 林德利 (Lindley, C.) 著
; 徐涛译. -- 北京 : 人民邮电出版社, 2014. 2
ISBN 978-7-115-33494-7

I. ①J… II. ①林… ②徐… III. ①JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2013)第252616号

版权声明

Copyright© 2013 by Q'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by Q'Reilly Media, Inc. and Posts & Telecom Press, 2014. Authorized translation of the English edition, 2013 Q'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版由 **Q'Reilly Media, Inc.** 授权人民邮电出版社出版。未经出版者书面许可, 对本书的任何部分不得以任何方式复制或抄袭。

版权所有, 侵权必究。

-
- ◆ 著 [美] Cody Lindley
 - 译 徐 涛
 - 责任编辑 陈冀康
 - 责任印制 程彦红 杨林杰
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 三河市海波印务有限公司印刷
 - ◆ 开本: 787×1000 1/16
 - 印张: 9.25
 - 字数: 163 千字 2014 年 2 月第 1 版
 - 印数: 1—3 000 册 2014 年 2 月河北第 1 次印刷

著作权合同登记号 图字: 01-2013-3680 号

定价: 35.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京崇工商广字第 0021 号

内容提要

JavaScript 是 Web 开发人员必须掌握的一门编程语言，但 JavaScript 语言及其相关技术正在变得越来越复杂。如何掌握 JavaScript 的基本概念和核心技术，往往让初学者和 JavaScript 新手感到束手无策。

本书力图在有限的篇幅内，通过考察原生 JavaScript 对象和所支持的细微差别，来给读者展现准确的 JavaScript 世界观，涉及对象、属性、复杂值、原始值、作用域、继承、this 关键字、head 对象等重要概念。本书帮助读者厘清这些概念，进而掌握应用它们的技术和技巧。

本书适合希望通过深入了解 JavaScript 对象来巩固对语言理解的高级初学者或中级 JavaScript 开发人员阅读，也适合准备研究 JavaScript 幕后知识的 JavaScript 库使用老手参考。

技术编辑简介

Michael Richardson

Michael Richardson 是一名 Web 和应用开发人员，居住在爱达荷州博伊西。很久以前，他在沙拉劳伦斯学院获得了创意写作的艺术硕士学位，并于 2003 年发表了名为《蘑菇电台计划(Plans for a Mushroom Radio)》的小说。最近，他在与他的妻子和孩子共度美好时光之余，还管理着自己的小型 Web 应用程序 Timeglider (<http://timeglider.com>)。

Kyle Simpson

Kyle Simpson 是一位来自得克萨斯州奥斯汀的 JavaScript 系统架构师。他主攻 JavaScript、Web 性能优化和“中端”应用程序架构。如果使用 JavaScript 或 Web 堆栈技术解决不了某个问题，他可能就会感到不耐烦了。他负责多个开源项目，包括 *LABjs*、HandlebarJS (<http://handlebars.com>) 和 BikechainJS (<http://bikechainjs.com>)。Kyle 是 Mozilla 开发工具团队里的一名软件工程师。

Nathan Smith

Nathan Smith (<http://sonspring.com>) 是在惠普工作的 UX 开发人员。他拥有亚斯毕理神学院的神学硕士学位。他在 20 世纪晚期开始建立网站，爱好手工编写 HTML、CSS 和 JavaScript 代码。他创建了 960 Grid System (<http://960.gs/>)，这是一种用于描绘、设计和编写页面布局的设计和 CSS 框架。他还创建了 Formalize (<http://formalize.com>)，一种使表格式样清晰的 JavaScript 和 CSS 框架。

Ben Nadel

Ben Nadel 是 Epicenter Consulting 的首席软件工程师，Epicenter Consulting 是一家位于曼哈顿的 Web 应用开发公司，专门从事创新定制软件，帮助客户转变其业务方式。他也是 Adobe 社区专家以及高级 ColdFusion 的 Adobe 认证专家。在业余时间，他在 www.bennadel.com 上撰写关于 Web 应用开发等方面的博客。

Ryan Florence

Ryan Florence (<http://ryanflorence.com>) 是来自犹他州盐湖城的前端 Web 开发人

员，自 20 世纪 90 年代早期就一直在建立网站。他尤其感兴趣的是将自己的体验提供给终端用户和延续项目的开发人员。**Ryan** 时常活跃在 **JavaScript** 社区，编写插件，帮助创建流行的 **JavaScript** 库，在各种会议和聚会上发表演讲，并将经验写在网上。他目前在 **Clock Four** 担任高级技术顾问。

Nathan Logan

Nathan Logan (<http://nathanlogan.com>) 是一名专业的 **Web** 开发人员，有 8 年的开发经验。他主攻客户端技术，但也涉及服务器端技术。他目前就职于 **Memolane**，与本书作者是同事。**Nathan** 有很贤惠的妻子和可爱的儿子，爱好骑山地自行车、泡温泉、吃辛辣食物，他是苏格兰人，信仰基督教。

作者简介

Cody Lindley 是一名客户端工程师(也称为前端开发人员)及 Flash 开发者。他在 HTML、CSS、JavaScript、Flash、客户端性能技术方面有丰富的工作经验(11 年以上)，这些技术都与 Web 开发有关。如果他不在编写客户端代码，就很可能是在研究界面/交互设计，或是在准备创作材料以及在各种会议上要用的演讲稿。当他不坐在电脑前时，他肯定正和他的妻子和孩子在爱达荷州的博伊西游玩，他们在博伊西可以进行三项全能运动训练、滑雪、骑山地自行车和公路自行车、登山、阅读、看电影或者讨论基督教世界观的理论依据。

译者简介

徐涛（网名：汤姆大叔；微博：@TomXuTao），微软最有价值专家（MVP）、项目经理、软件架构师，擅长大型互联网产品的架构与设计，崇尚敏捷开发模式，熟悉设计模式、前端技术、以及各种开源产品，曾获 MCP、MCSE、MCDBA、MCTS、MCITP、MCPD、PMP 认证。《JavaScript 编程精解》译者，博客地址：<http://www.cnblogs.com/tomxu>。

前言

简介

本书无关于 JavaScript 设计模式，也无关于 JavaScript 面向对象代码实现。本书的写作目的也不是鉴别 JavaScript 语言特点的好坏。本书并不是一本完整的参考指南。它面向的读者人群并不是编程新手或对 JavaScript 完全陌生的人员。同时，它也不是一本 JavaScript 攻略手册。关于上述这些方面的书籍都已经面世。

本书的撰写意图是通过考察原生 JavaScript 对象和不同环境对原生对象的支持的细微差别，来给读者展现准确的 JavaScript 世界观：复杂值、原始值、作用域、继承、head 对象等。我希望本书是关于 ECMAScript 第三版规范的简单易懂的总结，重点介绍 JavaScript 中对象的特性。

如果你是只使用过 JavaScript 库（如 jQuery、MooTools、Zepto、YUI、Dojo 等）的设计师或开发人员，我希望本书中的资料能够使你从 JavaScript 库用户转变成为 JavaScript 开发人员。

为什么要写这本书？

首先，我必须承认，写这本书是为了我自己。说实话，精心编制这些资料，这样我就可以品尝自己制作的“饮品”，并始终记得它的“味道”。换句话说，我想用自己的语言来编写参考书籍，以便在需要时用来唤起我的记忆。另外：

- JS 库会导致“黑匣子”综合征，它对某些方面是有益的，但对某些方面是不利的。有些事情完成得很快速和高效，但你却不知道如何或者为何要如此。当事情不顺利或性能成为问题时，如何以及为何就显得很重要了。事实上，在构建 Web 应用程序时（或只是一个优秀的注册表单），如果想要实现 JavaScript 库或框架，就应该打开“引擎盖”看看，了解“发动机”的情况。本书就是写给那些想要打开 JavaScript 这个“引擎盖”并不怕弄脏手的人的。
- Mozilla 提供了最新和最完整的 JavaScript 1.5 参考指南。我认为现在缺少的是一个从单一角度编写的易读文档，从而配合他们的参考指南使用。希望本书会

成为 JavaScript 方面的指南，告诉你“你需要知道什么”，并详述一些 Mozilla 指南未涉及的概念。

- 虽然 JavaScript 1.5 版本发布了很长一段时间，ES6 和 ES5 中的新特性固然要用，但我们希望将存在时间较久的有关 JavaScript 基础概念撰写成书。
- 有关编程语言的高级技术书籍通常都有大量的代码示例和无意义的漫谈。我更喜欢用直接切中要点的简短解释，使用可以立即运行的真实代码。我发明了一个新词“技术性薄片撷取 (technical thin-slicing)”，来描述我在本书中想要使用的东西。这就需要将复杂的主题精简为更小、更易读的概念，并采用最少的词汇以及全面、精准的代码示例。
- 大多数值得一读的 JavaScript 书籍都有 3 英寸厚。像 David Flanigan 等人所写的权威指南肯定有它们自己的一席之地，但我想要编写的书籍只专注于重要的内容，而不详述所有知识。

谁应该阅读本书？

本书面向两种人群。第一种是希望通过深入了解 JavaScript 对象来巩固对语言理解的高级初学者或中级 JavaScript 开发人员。第二种是准备研究 JavaScript 幕后知识的 JavaScript 库使用老手。本书不适合编程新手、JavaScript 库使用新手以及 JavaScript 开发新手。

为什么是 JavaScript 1.5 和 ECMAScript 第 3 版？

在本书中，我重点介绍的是 JavaScript 1.5 版本（相当于 ECMAScript 第 3 版），因为它是到目前为止实现最为广泛的 JavaScript 版本。本书的下一版肯定会转向日渐重要的 ES6 和 ES5 版本。

为什么没有涉及 Date()、Error()和 RegEx()对象？

正如前面所说的，本书不是关于 JavaScript 的详尽参考指南，相反，本书致力于将对象作为了解 JavaScript 的透镜。因此，我决定在书中不涉及 Date()、Error()和 RegEx()对象，尽管这些对象都很有用，但是掌握这些对象的细节内容对 JavaScript 对象的总体理解不会产生很大的影响。希望大家能够将在这里所学到的知识应用到 JavaScript 环境中可用的所有对象中。

在开始学习本书之前，了解本书所采用的各种模式是非常重要的。请不要跳过本节，因为它包含一些有助于阅读本书的重要信息。

使用 JS Bin 时，要记住的是，必须要手动执行代码（单击 **Render**），因为这不同于由 jsFiddle 执行的页面加载。

本书约定

本书使用下列排版约定（参照编码约定（P3））：

斜体(Italic)

表示第一次出现的专业词汇、链接(URL)、电子邮件地址、文件名和文件扩展名。

等宽字体(Constant width)

表示广义上的程序清单，包括变量或函数名、数据库、数据类型、环境变量、语句和关键字。

等宽粗体(Constant width bold)

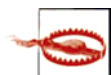
表示应该由用户逐字输入的命令或其他文本。

等宽斜体(Constant width italic)

表示应该由用户提供的值或取决于上下文的值。



这个图标表示提示、建议或一般说明。



这个图标表示警告或提醒。

代码示例

这本书是为了帮助你做好工作。一般来说，可以在你的程序和文档中使用本书的代码，无须联系我们获取许可。例如，使用本书中的几段代码写一个程序是不需要许可的；出售和散布 O’ Reilly 书中用例的光盘（CD-ROM）是需要许可的；通过引用本书用例和代码来回答问题是不需要许可的；在你的产品文档中引用本书中大量的用例代码是需要许可的。

我们赞赏但不强求注明信息来源。一条信息来源通常包括标题、作者、出版者和国际标准书号（ISBN），例如，“*JavaScript Enlightenment* by Cody Lindley (O’Reilly). Copyright 2013 Cody Lindley, 978-1-449-34288-3”。

如果你感到对示例代码的使用超出了正当引用或这里给出的许可范围，请随时通过

permissions@oreilly.com 联系我们。

Safari®在线图书



Safari 在线图书 (Safari Books Online: www.safaribooksonline.com)

是一家按需服务的数字图书馆, 提供来自世界领先作者的技术类和商业类专业参考书目和视频。

专业技术人员、软件开发人员、Web 设计师、商业和创意专家将 Safari 在线图书作为他们研究、解决问题、学习和认证培训的主要资源。

Safari 在线图书为组织、政府机构和个人提供一系列的产品组合和定价计划。用户可以在一个来自各个出版社的完全可搜索的数据库中访问成千上万的书籍、培训视频和正式出版前的手稿。这些出版社包括: O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、微软出版社、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。欲获得有关 Safari 在线图书的更多信息, 请在线访问我们。

联系我们

如果您对本书有意见和问题, 请联系出版社:

美国:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国:

北京西城区西直门南工街 2 号

成铭大厦 C 座 807 室 (100035)

奥莱利技术咨询 (北京) 有限公司

我们已将本书做成一个网页, 我们在那里列出了勘误表、示例及其他信息。您可以

打开网址 http://oreil.ly/javascript_enlightenment 访问这个页面。

如需对本书发布评论或提出技术问题，请发送电子邮件至 bookquestions@oreilly.com。

欲获知有关我们的图书、课程、会议及新闻的更多信息，请访问我们的网站 <http://www.oreilly.com>。

Facebook 网址: <http://facebook.com/oreilly>

Twitter 网址: <http://twitter.com/oreillymedia>

YouTube 网址: <http://www.youtube.com/oreillymedia>

目录

第 1 章	JavaScript 对象	1
1.1	创建对象	1
1.2	JavaScript 构造函数构建并返回对象实例	6
1.3	JavaScript 原生/内置对象构造函数	7
1.4	用户自定义/非原生对象构造函数	8
1.5	使用 new 操作符实例化构造函数	10
1.6	从构造函数创建字面量值	11
1.7	原始值（或简单值）	13
1.8	null、undefined、“string”、10、true 和 false 等原始值不是对象	14
1.9	如何存储和复制原始值	15
1.10	原始值比较采用值比较	16
1.11	原始值（String、Number、Boolean）在被用做对象时就像对象	17
1.12	复杂值（或组合值）	18
1.13	如何存储或复制复杂值	19
1.14	复杂对象比较采用引用比较	20
1.15	复杂对象具有动态属性	21
1.16	typeof 操作符	21
1.17	动态属性支持易变对象	23
1.18	构造函数实例都拥有指向其构造函数的 Constructor 属性	24
1.19	验证对象是否是特定构造函数的实例	26
1.20	构造函数创建的实例可拥有自己独立的属性（实例属性）	27
1.21	JavaScript 对象和 Object()对象	28
第 2 章	对象与属性	29
2.1	复杂对象可以将大多数 JavaScript 值作为属性	29
2.2	封装复杂对象	30
2.3	用点表示法或中括号表示法获取/设置/更新对象属性	31
2.4	删除对象属性	34
2.5	如何解决对象属性的引用	34
2.6	使用 hasOwnProperty 验证对象属性不是来自原型链	37
2.7	使用 in 操作符检查一个对象是否包含给定属性	37

2.8	使用 for in 循环枚举（循环遍历）对象的属性	38
2.9	宿主对象与原生对象	39
2.10	使用 Underscore.js 增强及扩展对象	40
第 3 章	Object()	43
3.1	Object()对象概要	43
3.2	Object()参数	44
3.3	Object()属性和方法	45
3.4	Object()对象实例属性和方法	45
3.5	使用对象字面量创建 Object()对象	46
3.6	所有对象都继承自 Object.prototype	47
第 4 章	Function()	49
4.1	Function()对象概要	49
4.2	Function()参数	50
4.3	Function()属性和方法	50
4.4	Function 对象实例属性和方法	51
4.5	函数总有返回值	51
4.6	函数是“一等公民”（不仅语法，还有值）	52
4.7	函数的参数传递	53
4.8	this 和 arguments 适用于所有函数	53
4.9	arguments.callee 属性	54
4.10	函数实例的 length 属性和 arguments.length	55
4.11	重定义函数参数	55
4.12	代码执行完成前取消函数执行	56
4.13	定义函数（语句、表达式或构造函数）	57
4.14	调用函数[函数、方法、构造函数或 call()和 apply()]	57
4.15	匿名函数	59
4.16	自调用的函数表达式	59
4.17	自调用的匿名函数语句	59
4.18	函数可以嵌套	60
4.19	给函数传递函数，从函数返回函数	61
4.20	函数定义之前调用（函数提升）	61
4.21	函数可以调用自身（递归）	62
第 5 章	head/全局对象	64
5.1	head/全局对象概要	64
5.2	head 对象内的全局函数	65

5.3	head 对象与全局属性、全局变量	65
5.4	引用 head 对象	67
5.5	head 对象是隐式的，通常不显式引用	67
第 6 章	this 关键字	69
6.1	this 概要及 this 如何引用对象	69
6.2	如何确定 this 值	70
6.3	在嵌套函数中用 this 关键字引用 head 对象	71
6.4	充分利用作用域链研究嵌套函数问题	73
6.5	使用 call()或 apply()控制 this 值	73
6.6	在用户自定义构造函数内部使用 this 关键字	75
6.7	原型方法内的 this 关键字引用构造函数实例	75
第 7 章	作用域和闭包	77
7.1	JavaScript 作用域概要	77
7.2	JavaScript 没有块作用域	78
7.3	在函数中用 var 声明变量，避免作用域陷阱	78
7.4	作用域链（词法作用域）	79
7.5	作用域链查找返回第一轮值	81
7.6	函数定义时确定作用域，而非调用时确定	81
7.7	闭包是由作用域链引起的	82
第 8 章	函数原型属性	84
8.1	原型链概要	84
8.2	为何要关注 prototype 属性	85
8.3	原型在所有 function()实例上都是标准的	85
8.4	默认的 prototype 属性是 Object()对象	86
8.5	将构造函数创建的实例链接至构造函数的 prototype 属性	87
8.6	原型链的最后是 Object.prototype	88
8.7	原型链返回在链中找到的第一个匹配结果	88
8.8	用新对象替换 prototype 属性会删除默认构造函数属性	89
8.9	继承原型属性的实例总是能够获得最新值	90
8.10	用新对象替换 prototype 属性不会更新以前的实例	91
8.11	用户自定义构造函数像原生构造函数一样原型继承	92
8.12	创建继承链	94
第 9 章	Array()	95
9.1	Array()对象概要	95
9.2	Array()参数	96

9.3	Array()属性和方法	96
9.4	数组对象实例属性和方法	96
9.5	创建数组	97
9.6	数组添加及更新	98
9.7	长度与索引	99
9.8	定义预定义长度的数组	100
9.9	可以通过设置数组长度添加或删除值	100
9.10	数组包含数组（多维数组）	101
9.11	遍历数组	101
第 10 章	String()	103
10.1	String()对象概要	103
10.2	String()参数	104
10.3	String()属性和方法	104
10.4	字符串对象实例属性和方法	104
第 11 章	Number()	106
11.1	Number()对象概要	106
11.2	整数和浮点数	106
11.3	Number()参数	107
11.4	Number()属性	108
11.5	数字对象实例属性和方法	108
第 12 章	Boolean()	109
12.1	Boolean()对象概要	109
12.2	Boolean()参数	109
12.3	Boolean()属性和方法	110
12.4	布尔对象实例属性和方法	110
12.5	非原始 false 布尔对象转换为 true	111
12.6	某些值是 false，其他都是 true	111
第 13 章	使用原始值：字符串、数字和布尔值	113
13.1	访问属性时原始值/字面量值被转换为对象	113
13.2	通常应使用原始字符串、数字和布尔值	115
第 14 章	null	116
14.1	null 值概要	116
14.2	typeof(null)的返回值为“object”	116
第 15 章	undefined	118
15.1	undefined 值概要	118

15.2	在全局作用域中定义 <code>undefined</code> 变量	119
第 16 章	Math 函数	120
16.1	内置 Math 对象概要	120
16.2	Math 属性和方法	120
16.3	Math 不是构造函数	122
16.4	Math 常数无法增大/改变	122
附录 A	回顾	123
附录 B	总结	126

JavaScript 对象

1.1 创建对象

在 JavaScript 中，对象为“王”：JavaScript 里的几乎所有东西都是对象或者用起来像对象。理解了对象，就能够理解 JavaScript。因此，让我们来查看一下 JavaScript 中的对象创建。

对象只是一组有命名值（也称为属性）集合的容器。在阅读 JavaScript 代码之前，让我们先来推理一下。以我自己为例，我们可以用简单的语言在表格中表达“cody”：

cody	
property	property value:
living	true
age	33
gender	male

上述表格中的“cody”一词只是一组属性名和所对应值的标签，这些属性和值构成了 cody。正如表格中所看到的：我还活着，33 岁，男性。

然而，JavaScript 不会用表格来表达，它是用对象来表达的，就像“cody”表格中包含的那样。将上述表格转化成实际的 JavaScript 对象应是这样的：

Fiddle 地址：<http://jsfiddle.net/javascriptenlightenment/ckVA5/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 创建 cody 对象
var cody = new Object();
```

```
// 为 cody 对象的各种属性赋值（使用点表示法）
cody.living = true;
cody.age = 33;
cody.gender = 'male';

console.log(cody); // 输出 Object {living = true, age = 33, gender = 'male'}

</script></body></html>
```

最重要的是要记住：对象只是属性的容器，每个属性都有一个名称和一个值。JavaScript 采用具有命名值属性的容器（即对象）这一概念作为在 JavaScript 中表达值的构建块。cody 对象是一个值，通过创建对象将这个值表示为 JavaScript 对象，给对象命名，然后将属性赋值给对象。

到目前为止，我们所讨论的 cody 对象只有静态信息。鉴于正在研究 JavaScript 编程语言，我们希望为 cody 对象编写代码，做一些真正有意义的事情。不然，也就只有一个类似 JSON (<http://www.json.org/>) 的数据库。为了使 cody 对象有生命力，需要添加一个属性：方法（*method*）。方法用于执行函数（*function*）。确切地讲（<http://bclary.com/2004/11/07/%23a-4.3.3>），在 JavaScript 中，方法是包含 Function() 对象的属性，其目的是对函数内部的对象进行操作。

更新 cody 表格，加入 getGender 方法，用浅显的英语表示如下：

cody	
property:	property value:
living	true
age	33
gender	male
getGender	return the value of gender

用 JavaScript 代码表示，更新后的上述“cody”表格中的 getGender 方法应是这样的：

Fiddle 地址：<http://jsfiddle.net/javascriptenlightenment/3gBT4/>

```
<!DOCTYPE html><html lang="en"><body><script>

var cody = new Object();
cody.living = true;
cody.age = 33;
cody.gender = 'male';
cody.getGender = function(){ return cody.gender;};
```

```
console.log(cody.getGender()); // 输出 'male'
```

```
</script></body></html>
```

`getGender` 方法是 `cody` 对象的属性，用于返回 `cody` 的其他属性值：存储在 `gender` 属性中的 "male" 值。必须知道的是，如果没有方法，除了用于存储静态属性以外，对象就没有其他太多用途。

截至目前，我们讨论的 `cody` 对象是一种 `Object()` 对象。通过使用调用 `Object()` 构造函数而得到的空对象创建了 `cody` 对象。将构造函数视为模板或饼干模具来生成预定义对象。对于 `cody` 对象示例，使用 `Object()` 构造函数来生成空对象，然后将它命名为 `cody`。鉴于 `cody` 是由 `Object()` 构造函数构造出的一个对象，所以将 `cody` 称为 `Object()` 对象。大家真正需要了解的是，除了创建像 `cody` 这样简单的 `Object()` 对象外，JavaScript 中的大多数值都是对象（"foo"、5 和 true 等原始值例外，但它们拥有等效包装器对象）。

由 `Object()` 构造函数创建的 `cody` 对象与通过 `String()` 构造函数创建的字符串对象没有太大的区别。为了证明其真实性，可检验和对比下面的代码：

Fiddle 地址：<http://jsfiddle.net/javascriptenlightenment/XcfC5/>

```
<!DOCTYPE html><html lang="en"><body><script>
var myObject = new Object(); // 生成一个 Object() 对象
myObject['0'] = 'f';
myObject['1'] = 'o';
myObject['2'] = 'o';

console.log(myObject); // 输出 Object { 0="f", 1="o", 2="o" }

var myString = new String('foo'); // 生成一个 String() 对象

console.log(myString); // 输出 foo { 0="f", 1="o", 2="o" }

</script></body></html>
```

正如这里所显示的，`myObject` 和 `myString` 都是对象！它们都可以有属性、继承属性，并且都由构造函数生成。包含 "foo" 字符串值的 `myString` 变量看似很简单，但令人惊讶的是，在其表面之下有一个对象结构。查看生成的这两个对象，你会发现这两个对象虽然在本质上是相同的，但在类型上是不同的。更重要的是，希望你开始了解，JavaScript 是使用对象来表示值的。



注意

看到字符串值采用 "foo" 对象形式,你可能会觉得很奇怪,因为通常字符串在 JavaScript 中被表示为原始值(如 `var myString = 'foo';`)。我在这里特地用一个字符串对象值来强调所有的东西都可以是对象,包括通常不将其视为对象的值(即字符串、数字、布尔值)。同时,我认为这有助于解释为什么有些人说 JavaScript 中的所有东西都可以是一个对象。

JavaScript 将 `String()` 和 `Object()` 构造函数作为其自身的语言,以便使 `String()` 对象和 `Object()` 对象的创建变得简单。但是作为 JavaScript 程序员,也可以创建同样强大的构造函数。如下代码,定义了一个非原生的自定义 `Person()` 构造函数,以便使用它创建 `people` 对象。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/zQDSw/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 定义 Person 构造函数,以便稍后创建自定义的 Person() 对象
var Person = function (living, age, gender) {
    this.living = living;
    this.age = age;
    this.gender = gender;
    this.getGender = function () { return this.gender; };
};

// 实例化 Person 对象,并将它保存到 cody 变量
var cody = new Person(true, 33, 'male');

console.log(cody);

/* 下面的 String() 构造函数由 JavaScript 自身所定义,有同样的模式。因为 string 构造函数是 JavaScript 内置的,获取字符串所要做的就是将它实例化,但无论是用原生的 String() 还是用户自定义的构造函数 Person(), 模式都是一样的 */

// 实例化一个 String 对象,并保存到 myString 变量
var myString = new String('foo');

console.log(myString);

</script></body></html>
```

用户自定义的 `Person()` 构造函数可以生成 `person` 对象,就像原生 `String()` 构造函数可以生成字符串对象一样。`Person()` 构造函数的能力和延展性并不比原生 `String()` 构造函数或 JavaScript 中的其他原生构造函数差。

还记得我们开始看到的 `cody` 对象是如何从 `Object()` 生成的吗？重要的是要注意，在上个代码示例中的 `Object()` 构造函数和 `new Person()` 构造函数可以产生相同的结果。两者都可以生成具有相同属性和属性方法的相同对象。查看下面两段代码，结果表明，尽管生成方式不同，`codyA` 和 `codyB` 却拥有相同的对象值。

Fiddle 地址：<http://jsfiddle.net/javascriptenlightenment/Du5YV/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 使用 Object() 构造函数创建 codyA 对象

var codyA = new Object();
codyA.living = true;
codyA.age = 33;
codyA.gender = 'male';
codyA.getGender = function () { return codyA.gender; };

console.log(codyA); // 输出 Object {living=true, age=33, gender="male", ...}

/* 下面创建了同样的 cody 对象，但不是使用原生的 Object() 构造函数创建一次性的 cody
对象，首先定义自己的 Person() 构造函数来创建 cody 对象（或者其他任意 Person 对象），然后使用
"new" 关键字进行实例化 */

var Person = function (living, age, gender) {
    this.living = living;
    this.age = age;
    this.gender = gender;
    this.getGender = function () { return this.gender; };
};
// 输出 Object {living=true, age=33, gender="male", ...}
var codyB = new Person(true, 33, 'male');

console.log(codyB);

</script></body></html>
```

`codyA` 和 `codyB` 对象之间的主要区别不在于对象本身，而是在于生成对象的构造函数。`codyA` 对象是使用 `Object()` 构造函数实例来产生的。`Person()` 构造函数创建了 `codyB`，但也可以把 `Person()` 构造函数当成一个强大的、集中定义的对象“工厂”，用于创建更多的 `Person()` 对象。为自定义对象创建自定义构造函数的同时，也为 `Person()` 实例创建了原型继承。

这两个方案都能够创建相同的复杂对象。这两个模式最常用于构造对象。

JavaScript 实际上是一种预包装若干原生对象构造函数的语言。这些构造函数用于

生成一些表达特定类型值（如数字、字符串、函数、对象、数组等）的复杂对象，同样，也可以通过 `Function()` 对象创建自定义的对象构造函数（例如 `Person()`）。不管是否是用于创建对象的模式，产生的最终结果通常都是创建一个复杂的对象。

理解对象的创建、本质、用法以及其原始等价代码是本书其余部分的重点。

1.2 JavaScript 构造函数构建并返回对象实例

构造函数的作用是创建多个共享特定特性和行为的对象。构造函数主要是一种用于生成对象的饼干模具，这些对象具有默认属性和属性方法。

如果说“构造函数只是一个函数”，那么我会说“你是对的，除非使用 `new` 关键字来调用该函数。”（如 `new String('foo')`）。如果使用 `new` 调用某函数，该函数则担任一个特殊的角色，JavaScript 给予该函数特殊待遇，将该函数的 `this` 值设置为正在构建的新对象。除了这个特殊行为，该函数还默认返回新创建的对象（即 `this`），而不是虚假值。该函数返回的新对象则被认为是构建该对象的构造函数的实例。

再次思考 `Person()` 构造函数，但这一次要仔细阅读下面代码中的注释，因为其内容强调了 `new` 关键字的作用。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/YPR6Q/>

```
<!DOCTYPE html><html lang="en"><body><script>

// Person 是一个构造函数，可以使用 new 关键字进行实例化

var Person = function Person(living, age, gender) {
    // 下面的 this 表示即将创建的新对象（即，this = new Object();）
    this.living = living;
    this.age = age;
    this.gender = gender;
    this.getGender = function () { return this.gender; };
    // 一旦该 Person 函数使用 new 关键字调用，就返回 this，而不是 undefined
};

// 实例化 Person 对象，命名为 cody
var cody = new Person(true, 33, 'male');

// cody 是一个对象，并且是 Person() 的一个实例
console.log(typeof cody); // 输出 object
console.log(cody); // 输出 cody 内部的属性和值
```

```
console.log(cody.constructor); // 输出 Person() 函数

</script></body></html>
```

上述代码利用了自定义构造函数（即 `Person()`）来创建 `cody` 对象。这与 `Array()` 构造函数创建 `Array()` 对象（如 `new Array()`）没有什么不同：

Fiddle 地址：<http://jsfiddle.net/javascriptenlightenment/cKa3a/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 实例化 Array 对象，命名为 myArray
var myArray = new Array(); // myArray 是 Array 的一个实例

// myArray 是一个对象，并且是 Array() 构造函数的一个实例
console.log(typeof myArray); // 输出 object! 什么? 是的，数组是 object 类型

console.log(myArray); // 输出[ ]

console.log(myArray.constructor); // 输出 Array()

</script></body></html>
```

在 JavaScript 中，大多数值（不包括原始值）都涉及正在被创建的对象，或者是从构造函数实例化的对象。构造函数返回的对象被称为实例。读者要熟悉这些语义，同样要熟悉利用构造函数来构建对象的模式。

1.3 JavaScript 原生/内置对象构造函数

JavaScript 语言包含 9 个原生（或内置）对象构造函数。JavaScript 使用这些对象来构建 JavaScript 语言。“构建”的意思是指，这些对象是用于表达 JavaScript 代码中的对象值，以及协调语言中的多个特性。因此，原生对象构造函数是多方面的，它们生成对象，但也被用于促进语言的编程约定的形成。例如，函数是 `Function()` 构造函数创建的对象，但作为构造函数，使用 `new` 关键字调用后，它们也可用于创建其他对象。

下面列出了 JavaScript 预包装的 9 个原生对象构造函数：

- `Number()`
- `String()`
- `Boolean()`

- Object()
- Array()
- Function()
- Date()
- RegExp()
- Error()

JavaScript 主要是由这 9 个对象（以及字符串、数字和布尔原始值）来创建的。深入理解这些对象，对充分利用 JavaScript 独特的编程力量和语言灵活性是非常关键的。



注意

- Math 对象在这里是很古怪的。它是一个静态对象，而不是构造函数，也就是说，我们不能这么做：`var x = new Math()`。但我们可以使用它，就好像它已经实例化好了一样（如 `Math.PI`）。实际上，Math 只是一个由 JavaScript 设置的对象命名空间，用于存储数学函数。
- 原生对象有时也被称为“全局对象”，因为它们是 JavaScript 中原生就可以使用的对象。不要混淆的术语是拥有 "head" 全局对象的 global object（全局对象），它是作用域链中的最高层级。例如，所有 Web 浏览器中都有的 window 对象。
- Number()、String() 和 Boolean() 构造函数不仅能够构建对象，而且能为字符串、数字和布尔值提供原始值，这取决于如何充分利用构造函数。如果直接调用这些构造函数，那么就会返回一个复杂对象。如果只是简单地在代码中表示一个数字、字符串或布尔值（5、“foo”和 true 等原始值），那么构造函数将返回一个原始值，而不是一个复杂的对象值。

1.4 用户自定义/非原生对象构造函数

正如在 `Person()` 构造函数中所看到的，我们可以创建自己的构造函数，从中可以生成不只一个，而是多个自定义对象。

下面列出了熟悉的 Person()构造函数:

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/GLMr8/>

```
<!DOCTYPE html><html lang="en"><body><script>
var Person = function (living, age, gender) {
    this.living = living;
    this.age = age;
    this.gender = gender;
    this.getGender = function () { return this.gender; };
};

var cody = new Person(true, 33, 'male');
console.log(cody); // 输出 Object {living=true, age=33, gender="male", ...}

var lisa = new Person(true, 34, 'female');
console.log(lisa); // 输出 Object {living=true, age=34, gender="female", ...}

</script></body></html>
```

正如你所看到的,通过传递特定的参数和调用 Person()构造函数,可以轻松地创建大量的特定 people 对象。当需要两个或三个以上具有相同属性但具有不同值的对象时,这样做是非常有用的。细想一下,这正是 JavaScript 使用原生对象所发挥的作用。Person()构造函数与 Array()构造函数一样也遵循同样的原则。因此 new Array('foo','bar')与 new Person(true, 33, 'male')确实没有多大差异。创建自定义构造函数只是在使用 JavaScript 本身在原生构造函数中所使用的相同的模式。



注意

- 虽然这不是必需的,但当创建将要与 new 操作符一起使用的自定义构造函数时,最佳做法是保持构造函数名称的第一个字符大写:是 Person(),而不是 person()。
- 关于构造函数比较复杂的一点就是 this 值在函数内部的使用方式。请记住,构造函数只是一个饼干模具,在将它与 new 关键字一起使用时,它会创建一个拥有构造函数内部定义的属性和值的对象。在使用 new 关键字时, this 值的字面意思是基于构造函数内部的语句创建的新对象/新实例。另一方面,如果创建一个构造函数,且不使用 new 关键字进行调用,那么 this 值将引用包含该函数的“父”对象。更多细节内容请参考第 6 章。
- 可以放弃使用 new 关键字和构造函数的概念,方法是使该函数显

式地返回一个对象。必须显式地编写构建并返回 `Object()` 对象的函数：`var myFunction = function(){return {prop: val}};`。但是，这样做会避开原型继承的使用。

1.5 使用 new 操作符实例化构造函数

构造函数从根本上说是用于创建预配置对象的饼干模具模板。以 `String()` 为例，这个函数在与 `new` 操作符 [`new String('foo')`] 一起使用时会创建基于 `String()` 模板的字符串实例。让我们来看一个示例。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/FKdsp/>

```
<!DOCTYPE html><html lang="en"><body><script>

var myString = new String('foo');

console.log(myString); // 输出 foo {0 = "f", 1 = "o", 2 = "o"}

</script></body></html>
```

上述代码创建了一个新的字符串对象，它是 `String()` 构造函数的一个实例。就像这样，我们在 JavaScript 中表达了一个字符串值。



注意

我并不是建议使用构造函数，而不使用等价方式：字面量/原始值，如 `var string="foo";`。但我建议，要理解字面量/原始值背后的内容。

如前所述，JavaScript 语言具有以下原生预定义的构造函数：`Number()`、`String()`、`Boolean()`、`Object()`、`Array()`、`Function()`、`Date()`、`RegExp()` 和 `Error()`。可以在任意一个构造函数上应用 `new` 操作符来实例化一个对象实例。如下代码，构建了这 9 类原生 JavaScript 对象。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/M9cWA/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 使用 new 关键字实例化每个原生构造函数

var myNumber = new Number(23);
var myString = new String('male');
var myBoolean = new Boolean(false);
```

```

var myObject = new Object();
var myArray = new Array('foo', 'bar');
var myFunction = new Function("x", "y", "return x*y");
var myDate = new Date();
var myRegExp = new RegExp('\bt[a-z]+\b');
var myError = new Error('Crap!');

// 输出/验证哪个构造函数创建了该对象
console.log(myNumber.constructor); // 输出 Number()
console.log(myString.constructor); // 输出 String()
console.log(myBoolean.constructor); // 输出 Boolean()
console.log(myObject.constructor); // 输出 Object()
console.log(myArray.constructor); // 在现在的浏览器中, 输出 Array()
console.log(myFunction.constructor); // 输出 Function()
console.log(myDate.constructor); // 输出 Date()
console.log(myRegExp.constructor); // 输出 RegExp()
console.log(myError.constructor); // 输出 Error()

</script></body></html>

```

通过使用 `new` 操作符, 告诉 JavaScript 解释器, 我们需要一个对应于构造函数实例的对象。例如, 在上述代码中, `Date()` 构造函数用于创建日期对象。`Date()` 构造函数是日期对象的饼干模具。也就是说, 它从 `Date()` 构造函数定义的默认模式中生成了日期对象。

至此, 大家应该熟悉从原生构造函数[例如 `new String('foo')`]和用户自定义构造函数[例如 `new Person(true, 33, 'male')`]创建对象实例的方法了。



注意

时刻记住, `Math` 是一个静态对象——其他方法的容器, 它不是使用 `new` 运算符的构造函数。

1.6 从构造函数创建字面量值

JavaScript 提供了叫做“字面量”的快捷方式——用于创建大多数原生对象值, 而不必使用 `new Foo()` 或 `new Bar()` 这样的方式。大多数情况下, 字面量语法与使用 `new` 操作符的效果相同。但是也有例外: `Number()`、`String()` 和 `Boolean()`, 请看下面的注释。

如果你有其他编程背景, 可能更熟悉用字面量方式创建对象。下面我使用 `new` 操

作符实例化原生 JavaScript 构造函数，然后创建相应的字面量等价形式。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/Nbkw4/>

```
<!DOCTYPE html><html lang="en"><body><script>

var myNumber = new Number(23); // 对象
var myNumberLiteral = 23; // 原始数字值，而非对象

var myString = new String('male'); // 对象
var myStringLiteral = 'male'; // 原始字符串值，而非对象

var myBoolean = new Boolean(false); // 对象
var myBooleanLiteral = false; // 原始布尔值，而非对象

var myObject = new Object();
var myObjectLiteral = {};

var myArray = new Array('foo', 'bar');
var myArrayLiteral = ['foo', 'bar'];

var myFunction = new Function("x", "y", "return x*y");
var myFunctionLiteral = function (x, y) { return x * y };

var myRegExp = new RegExp('\bt[a-z]+\b');
var myRegExpLiteral = /\bt[a-z]+\b/;

// 验证创建自同样的构造函数的字面量

console.log(myNumber.constructor, myNumberLiteral.constructor);
console.log(myString.constructor, myStringLiteral.constructor);
console.log(myBoolean.constructor, myBooleanLiteral.constructor);
console.log(myObject.constructor, myObjectLiteral.constructor);
console.log(myArray.constructor, myArrayLiteral.constructor);
console.log(myFunction.constructor, myFunctionLiteral.constructor);
console.log(myRegExp.constructor, myRegExpLiteral.constructor);

</script></body></html>
```

这里你需要注意的是，在一般情况下，使用字面量只是隐藏了与使用 `new` 操作符相同的基本过程。重要的是，它方便多了！

但在原始字符串、数字和布尔值方面，事情变得更复杂了。在这些情况下，字面量值具有原始值的特点，而不是复杂对象值的特点。请查看如下注意事项。



注意

在针对字符串、数字和布尔值使用字面量值时，只有在该值被视为对象的情况下才会创建实际的复杂对象。换句话说，在尝试使用与构造函数有关联的方法或检索属性（如 `var charactersInFoo = 'foo'.length`）之前，一直在使用原始数据类型。当这种情况发生时，JavaScript 会在幕后为字面量值创建一个包装器对象，以便将该值视为一个对象。调用方法以后，JavaScript 即抛弃包装器对象，该值返回字面量类型。这就是字符串、数字和布尔被认为是原始（或简单）数据类型的原因。我希望这能够澄清对“JavaScript 中的所有东西都是对象”与“JavaScript 中所有东西都用成对象”这两个概念的误解。

1.7 原始值（或简单值）

5、“foo”、true、false，以及 null 和 undefined 等 JavaScript 值都被视为原始值，因为它们是不可细化的。也就是说，数字是数字，字符是字符，布尔值则是 true 或 false，null 和 undefined 就是 null 和 undefined。这些值本身是很简单的，不能表示由其他值组成的值。

查看下面的代码，并思考一下字符串、数字、布尔值、null 和 undefined 值是否可以更加复杂。将它与你所知道的 Object()实例或 Array()实例或其他任何复杂对象进行比较。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/xUQTC/>

```
<!DOCTYPE html><html lang="en"><body><script>

var myString = 'string'
var myNumber = 10;
var myBoolean = false; // 可以是 true，也可以是 false
var myNull = null;
var myUndefined = undefined;

console.log(myString, myNumber, myBoolean, myNull, myUndefined);

// 假设一个像数组或对象这样的复杂对象可以由多个原始值组成，并由此变成一个复杂的多值集
var myObject = {
  myString: 'string',
  myNumber: 10,
  myBoolean: false,
  myNull: null,
  myUndefined: undefined
}
```



```
};

console.log(myObject);

var myArray = ['string', 10, false, null, undefined];

console.log(myArray);

</script></body></html>
```

很简单，原始值是表示 JavaScript 中可用的数据/信息的最底层形式（即最简单的形式）。



注意

- 与使用字面量语法创建值相反，在使用 `new` 关键字创建 `String()`、`Number()` 或 `Boolean()` 值时，创建的对象实际上是一个复杂对象。
- 了解 `String()`、`Number()` 和 `Boolean()` 构造函数是两种目的的构造函数，分别是用于创建字面量/原始值以及复杂值的，这是非常重要的。这些构造函数并不总是返回对象，相反，在不使用 `new` 操作符的情况下，它返回实际复杂对象值的原始表示。

1.8 null、undefined、"string"、10、true 和 false 等原始值不是对象

`null` 和 `undefined` 都是非常简单的值，它们不需要构造函数，也没有使用 `new` 操作符来将自己创建为 JavaScript 值。欲使用 `null` 或 `undefined`，只需将它们看做操作符来使用即可。从技术上来讲，从构造函数返回的字符、数字、布尔值并不是对象。

下面对原始值和其他原生 JavaScript 对象之间的差异进行了比较。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/ZwgqD/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 使用原始值的时候，不会创建对象，注意没有使用 new 关键字
var primitiveString1 = "foo";
var primitiveString2 = String('foo');
var primitiveNumber1 = 10;
var primitiveNumber2 = Number('10');
var primitiveBoolean1 = true;
```

```

var primitiveBoolean2 = Boolean('true');

// 确认不是对象类型
console.log(typeof primitiveString1, typeof primitiveString2);
// 输出 'string,string'
console.log(typeof primitiveNumber1, typeof primitiveNumber2);
// 输出 'number,number'
console.log(typeof primitiveBoolean1, typeof primitiveBoolean2);
// 输出 'boolean,boolean'

// 比较用于创建对象的构造函数和 new 关键字的用法

var myNumber = new Number(23);
var myString = new String('male');
var myBoolean = new Boolean(false);
var myObject = new Object();
var myArray = new Array('foo', 'bar');
var myFunction = new Function("x", "y", "return x * y");
var myDate = new Date();
var myRegExp = new RegExp('\\bt[a-z]+\\b');
var myError = new Error('Crap!');

// 输出 'object object object object object function object function object'
console.log(
  typeof myNumber,
  typeof myString,
  typeof myBoolean,
  typeof myObject,
  typeof myArray,
  typeof myFunction, // 注意所有的 function 对象返回的类型都是 function
  typeof myDate,
  typeof myRegExp, // 注意 RegExp() 的类型返回的是 function
  typeof myError
);

</script></body></html>

```

希望大家从前面的代码中可以掌握：原始值不是对象。原始值的特殊之处在于它们是用来表示简单值。

1.9 如何存储和复制原始值

原始值在“面值（face value）”中的存储和操作，理解这一点相当重要。这可能听起来很简单，但是这意味着如果将字符串值“foo”存储在 myString 变量中，那么从字面上讲，“foo”值同样存储于内存中。为什么了解这个知识很重要？因为一旦

开始操作（即复制）值，我们就必须了解它，因为原始值是真实值复制。

在下面的示例中，我们在变量 `myStringCopy` 中存储了 `myString` 值('foo')的一个副本，其值是真实值复制。即使我们改变原来的值，变量 `myStringCopy` 引用的复制值仍保持不变。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/Gh3dW/>

```
<!DOCTYPE html><html lang="en"><body><script>

var myString = 'foo' //创建原始值字符对象
var myStringCopy = myString; //复制字符并存储到新变量上

var myString = null; //操作存储在 myString 变量中的值

/* 原来的值从myString复制到了myStringCopy上,更新myString以后,再检测myStringCopy
的值即可证实这点*/

console.log(myString, myStringCopy); // 输出 'null foo'

</script></body></html>
```

这里的重点是，原始值是作为不可细化的值进行存储和操作的。引用它们会转移其值。在上面的示例中，我们复制或者克隆了 `myString` 值并存储到 `myStringCopy` 变量中。当更新 `myString` 值时，`myStringCopy` 值仍是旧 `myString` 值的副本。请记住这点，并对这里的技术细节和复杂对象进行比较（将在下面讨论）。

1.10 原始值比较采用值比较

可以通过比较原始值来确定其值在字面上是否相同。从逻辑上讲，如果将一个包含数值 10 的变量与另一个包含数值 10 的变量进行比较，JavaScript 将会认为它们是相等的，因为 10 与 10 是相同的（即 `10===10`）。同样，如果将原始字符串“foo”与另一个拥有“foo”值的原始字符串进行比较也会采用同样的方法。比较的结果是，根据它们的值，它们是彼此相等的（即 `'foo' === 'foo'`）。

在下面的代码中，我使用原始数字演示了“值比较”的概念，并将它与复杂数字对象进行比较。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/NewQU/>

```
<!DOCTYPE html><html lang="en"><body><script>
```

```

var price1 = 10;
var price2 = 10;
var price3 = new Number('10'); // 复杂数字对象，因为使用了 new
var price4 = price3;

console.log(price1 === price2); // 输出 true

// 输出 false，因为 price3 包含了一个复杂数字对象，而 price1 是原始值
console.log(price1 === price3);

// 输出 true，因为复杂对象采用引用比较，而不是值比较
console.log(price4 === price3);

// 如果使 price4 的值变为原始值，那又如何呢？
price4 = 10;
console.log(price4 === price3); /* 输出 false: price4 此时是原始值，而非复杂对象 */

</script></body></html>

```

这里的重点是，在进行比较时，原始值会去检查表示的值是否相等。当使用 `new` 关键字创建字符串、数字或布尔值时[如 `new Number('10')`]，这个值就不再是原始值。同样地，如果通过字面量语法创建的值进行比较，结果就不同了。这并不奇怪，因为原始值是按值来存储的（`10===10` 成立吗？），而复杂值是按引用来存储的（`price3` 和 `price4` 包含对相同值的引用吗？）。

1.11 原始值（String、Number、Boolean）在被用做对象时就像对象

原始值被当作构造函数创建的一个对象来使用时，JavaScript 会将其转换为一个对象，以便可以使用对象的特性（如方法），而后抛弃对象性质，并将它变回到原始值。下面的代码采用了原始值，并展示了将值作为对象使用时会发生什么事情。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/gSTNp/>

```

<!DOCTYPE html><html lang="en"><body><script>

// 生成原始值
var myNull = null;
var myUndefined = undefined;
var primitiveString1 = "foo";
var primitiveString2 = String('foo'); // 没有使用 new 因此获得的是原始值
var primitiveNumber1 = 10;
var primitiveNumber2 = Number('10'); // 没有使用 new，因此获得的是原始值

```

```

var primitiveBoolean1 = true;
var primitiveBoolean2 = Boolean('true'); // 没有使用 new, 因此获得的是原始值

/* 使用 toString()方法 (从 object.prototype 继承而来) 来演示: 当把原始值用作对象
时, 原始值就会转化成对象 */

// 输出 "string string"
console.log(primitiveString1.toString(), primitiveString2.toString());

// 输出 "number number"
console.log(primitiveNumber1.toString(), primitiveNumber2.toString());

// 输出 "boolean boolean"
console.log(primitiveBoolean1.toString(), primitiveBoolean2.toString());

// 因为 null 和 undefined 没有转化成对象, 也没有构造函数, 所以下面 2 条语句会抛出错误
console.log(myNull.toString());
console.log(myUndefined.toString());

</script></body></html>

```

上述代码示例中, 所有的原始值 (除 `null` 和 `undefined`) 都被转换为对象, 以便充分利用 `toString()` 方法。一旦调用和返回该方法, 对象就会转换成原始值。

1.12 复杂值 (或组合值)

原生对象构造函数 `Object()`、`Array()`、`Function()`、`Date()`、`Error()` 和 `RegExp()` 是复杂类型, 因为它们可以包含一个或多个原始值或复杂值。本质上, 复杂值可以由很多不同类型的 JavaScript 对象组成。可以这样说, 复杂对象其在内存中的大小是未知的, 因为复杂对象可以包含任何值, 而不是一个特定的已知值。在如下代码中, 我们创建了一个包含所有原始值的对象和一个包含所有原始值的数组。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/JeFqt/>

```

<!DOCTYPE html><html lang="en"><body><script>
var object = {
    myString: 'string',
    myNumber: 10,
    myBoolean: false,
    myNull: null,
    myUndefined: undefined
};

var array = ['string', 10, false, null, undefined];

```

```
// 相比简单的原始值，原始值不能表示复杂值，而复杂值可以封装任意 JavaScript 值
var myString = 'string';
var myNumber = 10;
var myBoolean = false;
var myNull = null;
var myUndefined = undefined;

</script></body></html>
```

这里的重点是，复杂值是各种值的组合，并且在复杂性和组合方面与原始值不同。



注意

“复杂对象”这一术语在其他作品中也被表达为“复合对象”或“引用类型”。如果还不太清楚，可以思考这些名称，它们都描述了 JavaScript 值的特性，不包括原始值在内。原始值不是“值引用”，不能表示其他值的组合（即一个由几部分或几个元素组成的东西）。另一方面，复杂对象是“值引用”，可以包含或封装其他值。

1.13 如何存储或复制复杂值

复杂值是通过引用进行存储和操作的，理解这一点相当重要。创建一个包含复杂对象的变量时，其值是内存中的一个引用地址。引用一个复杂对象时，使用它的名称（即变量或对象属性）通过内存中的引用地址获取该对象值。当考虑试图复制一个复杂值时会发生什么事的时候，这就非常重要了。下面创建一个对象，并将它保存在变量 `myObject` 中，然后将 `myObject` 中的这个值复制到变量 `copyOfMyObject` 中。实际上，这并不是复制对象，更像是复制对象的地址。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/hypZC/>

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {};
var copyOfMyObject = myObject; //没有复制值，而是复制了引用

myObject.foo = 'bar'; // 操作 myObject 中的值

/* 现在如果输出 myObject 和 copyOfMyObject，则均会输出 foo 属性，因为它们引用的是
同一个对象*/
console.log(myObject, copyOfMyObject); /* 输出 'Object { foo="bar"}
Object { foo="bar"}' */

</script></body></html>
```

我们必须认识到的是，与复制值的原始值不同，对象（也称为复杂值）是通过引用进行存储的。因此，复制的是引用（也称为地址），而不是实际的值。这意味着根本就没有复制对象。就像我说过的，复制的是内存堆栈中对象的地址或引用。在代码示例中，`myObject` 和 `copyOfMyObject` 都指向存储于内存中的同一个对象。

这里的重点是，当更改复杂值时，所有引用相同地址的变量的值都会被修改，因为复杂值是通过引用进行存储的。在上述代码示例中，更新 `myObject` 和 `copyOfMyObject` 中任何一个变量中的值，两者都会被更改。



注意

- 使用 `new` 关键字创建 `String()`、`Number()` 和 `Boolean()` 值时，或者这些值在幕后被转换成复杂对象时，值依然是按照值进行存储/复制的。因此，即使原始值可以被看做是复杂值，它们也不具备通过引用进行复制的特性。
- 若要真正地复制一个对象，必须要从旧的对象中提取值，并将提取的值注入新对象。

1.14 复杂对象比较采用引用比较

复杂对象只有在引用相同的对象（即有相同的地址）时才相等。包含相同对象的两个变量彼此不相等，因为它们并不指向同一个对象。

下面的示例中，`objectFoo` 和 `objectBar` 有相同的属性，实际上也是完全相同的对象，但通过 `===` 询问它们是否相等时，JavaScript 会告诉我们：它们是不相等的。

Fiddle 地址：<http://jsfiddle.net/javascriptenlightenment/g4CfS/>

```
<!DOCTYPE html><html lang="en"><body><script>

var objectFoo = { same: 'same' };
var objectBar = { same: 'same' };

// 输出 false, JS 并不关注它们的类型是否相同
console.log(objectFoo === objectBar);

// 复杂对象如何才能相等
var objectA = { foo: 'bar' };
var objectB = objectA;
```

```
console.log(objectA === objectB); // 输出 true, 因为它们引用的是同一个对象

</script></body></html>
```

这里的重点是, 指向内存中复杂对象的变量, 只有在引用相同“地址”的情况下才是相等的。相反, 两个单独创建的对象, 即使具有相同的类型并拥有完全相同的属性, 它们也是不相等的。

1.15 复杂对象具有动态属性

一个新变量, 指向现有的复杂对象, 并没有复制该对象。这就是复杂对象有时被称为引用对象的原因。复杂对象可以根据需求有任意多个引用, 即使对象改变, 它们也总是指向同一个对象。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/SSsVC/>

```
<!DOCTYPE html><html lang="en"><body><script>

var objA = { property: 'value' };
var pointer1 = objA;
var pointer2 = pointer1;

// 更新 objA.property, 所有引用(pointer1 和 pointer2)都被更新了
objA.property = null;

// 输出 'null null null', 因为 objA, pointer1 和 pointer2 引用的都是同一对象
console.log(objA.property, pointer1.property, pointer2.property);

</script></body></html>
```

复杂对象支持动态对象属性, 因为我们可以定义对象, 然后创建引用, 再更新对象, 并且所有指向该对象的变量都会“获得”更新。

1.16 typeof 操作符

typeof 操作符用于返回正在使用值的类型。但它返回的值并不一致, 或者说, 逻辑上不一致。下面的代码展示了使用 typeof 操作符所返回的值。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/QM95R/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 原始值
```



```

var myNull = null;
var myUndefined = undefined;
var primitiveString1 = "string";
var primitiveString2 = String('string');
var primitiveNumber1 = 10;
var primitiveNumber2 = Number('10');
var primitiveBoolean1 = true;
var primitiveBoolean2 = Boolean('true');

console.log(typeof myNull); // 输出 object, 需要注意!!!
console.log(typeof myUndefined); // 输出 undefined
console.log(typeof primitiveString1, typeof primitiveString2);
    // 输出 string string
console.log(typeof primitiveNumber1, typeof primitiveNumber2);
    // 输出 number number
console.log(typeof primitiveBoolean1, typeof primitiveBoolean2);
    // 输出 boolean boolean

// 复杂值
var myNumber = new Number(23);
var myString = new String('male');
var myBoolean = new Boolean(false);
var myObject = new Object();
var myArray = new Array('foo', 'bar');
var myFunction = new Function("x", "y", "return x * y");
var myDate = new Date();
var myRegExp = new RegExp('\\bt[a-z]+\\b');
var myError = new Error('Crap!');

console.log(typeof myNumber); // 输出 object
console.log(typeof myString); // 输出 object
console.log(typeof myBoolean); // 输出 object
console.log(typeof myObject); // 输出 object
console.log(typeof myArray); // 输出 object
console.log(typeof myFunction); // 输出 function, 需要注意!!!
console.log(typeof myDate); // 输出 object
console.log(typeof myRegExp); // 输出 function, 需要注意!!!
console.log(typeof myError); // 输出 object

</script></body></html>

```

在这些值上使用该操作符时，考虑到正在使用的值（原始值或复杂值）的类型，我们应该特别注意潜在的返回值。

1.17 动态属性支持易变对象

复杂对象是由动态属性构成的。这使得用户自定义对象和大多数原生对象可以产生突变。这意味着 JavaScript 中的大多数对象都可以随时更新或更改。正因为如此，可以通过增加原生对象，来改变 JavaScript 本身的原生预配置特性。然而，我并不是建议大家这样做，事实上，我认为不应该这样做，但我们不能掩盖这些用法的有用之处。

也就是说，我们可以在原生构造函数上存储属性，并在原型对象上，向原生对象额外添加新方法。

如下代码，改变了 String()构造函数和 String.prototype。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/QvbDw/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 利用 augmentedProperties 属性扩展内置的 String 构造函数 Function()
String.augmentedProperties = [];

if (!String.prototype.trimIT) { // 如果原型中没有 trimIT(),就添加它
    String.prototype.trimIT = function () {
        return this.replace(/^\s+|\s+$/g, '');
    }
    // 添加 trimIT 字符到 augmentedProperties 数组
    String.augmentedProperties.push('trimIT');
}
var myString = ' trim me ';
console.log(myString.trimIT()); // 调用自定义的 trimIT 方法, 输出 'trim me'

console.log(String.augmentedProperties.join()); // 输出 'trimIT'

</script></body></html>
```

希望大家能够明白，JavaScript 中的对象是动态的。这使得 JavaScript 中的对象是可以改变的。从本质上讲，整个 JavaScript 语言都可以变为自定义版本（例如 trimIT 方法）。再次说明，我不推荐这种做法，我只是指出它是 JavaScript 中对象特性的一部分。



注意

如果更改了 JavaScript 的原生内部运作机制，你可能会获得一个自定义版本的 JavaScript 来进行程序处理。但一定要谨慎，因为大多数人会认为 JavaScript 不管在什么时候使用，都是相同的。

1.18 构造函数实例都拥有指向其构造函数的 Constructor 属性

任何对象实例化时，都是在幕后将 `constructor` 属性创建为对象/实例的属性。这是指创建对象的构造函数。下面创建一个 `Object()` 对象，保存在变量 `foo` 中，然后验证 `constructor` 属性在创建的对象中是否可用。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/ZtewV/>

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = {};
console.log(foo.constructor === Object) /* 输出 true, 因为 object() 构建了
                                         foo */
console.log(foo.constructor) // 指向 Object() 构造函数

</script></body></html>
```

如下功能非常方便：如果正在使用一些实例，而无法看到是谁或者是什么创建了它（尤其是别人编写的代码），那么可以以此确定它是否是一个数组或一个对象等。

下面实例化 JavaScript 语言中的大多数预配置对象。请注意，在字面量/原始值上使用 `constructor` 属性能够指向正确的构造函数。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/yJqaf/>

```
<!DOCTYPE html><html lang="en"><body><script>

var myNumber = new Number('23');
var myNumberL = 23; // 字面量方式
var myString = new String('male');
var myStringL = 'male'; // 字面量方式
var myBoolean = new Boolean('true');
var myBooleanL = true; // 字面量方式
var myObject = new Object();
var myObjectL = {}; // 字面量方式
var myArray = new Array();
var myArrayL = []; // 字面量方式
var myFunction = new Function();
var myFunctionL = function () { }; // 字面量方式
var myDate = new Date();
var myRegExp = new RegExp('/./');
var myRegExpL = /. /; // 字面量方式
var myError = new Error();
```

```

console.log( // 所有这些都返回 true
    myNumber.constructor === Number,
    myNumberL.constructor === Number,
    myString.constructor === String,
    myStringL.constructor === String,
    myBoolean.constructor === Boolean,
    myBooleanL.constructor === Boolean,
    myObject.constructor === Object,
    myObjectL.constructor === Object,
    myArray.constructor === Array,
    myArrayL.constructor === Array,
    myFunction.constructor === Function,
    myFunctionL.constructor === Function,
    myDate.constructor === Date,
    myRegExp.constructor === RegExp,
    myRegExpL.constructor === RegExp,
    myError.constructor === Error
);

</script></body></html>

```

`constructor` 属性也适用于用户自定义的构造函数。如下代码中，我们定义了一个 `CustomConstructor()` 构造函数，然后使用 `new` 关键字调用构造函数来生成一个对象。一旦创建了对象，就可以使用 `constructor` 属性了。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/MDs2t/>

```

<!DOCTYPE html><html lang="en"><body><script>

var CustomConstructor = function CustomConstructor() { return 'Wow!'; };
var instanceOfCustomObject = new CustomConstructor();

// 输出 true
console.log(instanceOfCustomObject.constructor === CustomConstructor);

// 返回 CustomConstructor() 函数的一个引用
// 返回 'function() { return 'Wow!'; };'
console.log(instanceOfCustomObject.constructor);

</script></body></html>

```



注意

大家可能会感到困惑，当不返回对象时，原始值为何拥有指向构造函数的 `constructor` 属性。使用原始值的时候，依然调用了构造函数，因此原始值和构造函数依然有关系。然而，最终的结果是

一个原始值。

- 对用户自定义的构造函数表达式，如果想让 `constructor` 属性记录构造函数的实际名称，则必须给予构造函数表达式一个实际名称（如 `var Person = function Person(){};`）。

1.19 验证对象是否是特定构造函数的实例

通过使用 `instanceof` 操作符，可以确定（`true` 或 `false`）一个对象是否是特定构造函数的实例。

如下代码，要验证的是 `InstanceOfCustomObject` 对象是否是 `CustomConstructor` 构造函数的实例。它适用于用户自定义对象，同时也适用于用 `new` 操作符创建的原生对象。

Fiddle 地址：<http://jsfiddle.net/javascriptenlightenment/g9Tt6/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 用户自定义对象构造函数
var CustomConstructor = function () { this.foo = 'bar'; };

// 初始化 CustomConstructor 的实例
var instanceOfCustomObject = new CustomConstructor();
// 输出 true
console.log(instanceOfCustomObject instanceof CustomConstructor);

// 原生对象亦是如此
console.log(new Array('foo') instanceof Array) // 输出 true

</script></body></html>
```



注意

- 使用 `instanceof` 操作符时需要注意的一件事情是，任何时间判断一个对象是否是 `Object` 的实例时，它都将返回 `true`，因为所有对象都继承自 `Object()` 构造函数。
- 原始值使用对象包装器，判断实例时（如 `'foo' instanceof String` // 返回 `false`），`instanceof` 操作符将返回 `false`。如果使用 `new` 操作符创建字符串 `'foo'`，`instanceof` 操作符会返回 `true`。所以，请记住，`instanceof` 只适用于构造函数创建返回的复杂对象和实例。

1.20 构造函数创建的实例可拥有自己独立的属性（实例属性）

在 JavaScript 中，对象在任何时候都可以扩展（即动态属性）。正如前面提到的，确切地说，JavaScript 拥有易变对象（*mutable object*）。这意味着通过构造函数创建的对象可以扩展属性。

下面的代码通过 `Array()` 构造函数创建了一个实例，然后利用其属性进行扩展。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/RuQfJ/>

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = new Array();
myArray.prop = 'test';
console.log(myArray.prop) // 输出 'test'

</script></body></html>
```

通过 `Object()`、`RegExp()` 或者任何其他非原生构造函数，甚至是 `Boolean()`，都可以实现同一目的。

Fiddle 地址: <http://jsfiddle.net/javascriptenlightenment/GnbPf/>

```
<!DOCTYPE html><html lang="en"><body><script>

// 原生构造函数创建的对象也可以实现同一目的
var myString = new String();
var myNumber = new Number();
var myBoolean = new Boolean(true);
var myObject = new Object();
var myArray = new Array();
var myFunction = new Function('return 2+2');
var myRegExp = new RegExp('\bt[a-z]+\b');

myString.prop = 'test';
myNumber.prop = 'test';
myBoolean.prop = 'test';
myObject.prop = 'test';
myArray.prop = 'test';
myFunction.prop = 'test';
myRegExp.prop = 'test';

// 输出 'test', 'test', 'test', 'test', 'test', 'test', 'test'
console.log(myString.prop, myNumber.prop, myBoolean.prop, myObject.prop,
  myArray.prop, myFunction.prop, myRegExp.prop);
```

```
// 注意:原始值/字面量不能实现该功能
var myString = 'string';
var myNumber = 1;
var myBoolean = true;

myString.prop = true;
myNumber.prop = true;
myBoolean.prop = true;

// 输出 undefined, undefined, undefined
console.log(myString.prop, myNumber.prop, myBoolean.prop);

</script></body></html>
```

向通过构造函数创建的对象添加属性的情况并不少见。请记住：通过构造函数创建的对象实例只是普通对象。



注意

请记住，除了自己的属性外，实例还可以拥有继承自原型链的属性。或者，正如上述代码所示，实例化后属性被添加至构造函数，这强调了 JavaScript 中对象的动态特性。

1.21 JavaScript 对象和 Object()对象

不要将一般术语“JavaScript 对象”与 Object()对象混淆。前者指的是 JavaScript 中对象的概念。Object()对象（如 `var myObject = new Object()`）是 JavaScript 中表示的一个非常特殊的值。就像 Array()对象是一种被称为 *array* 的对象类型，Object()对象是一种被称为 *object* 的对象类型。重点是，Object()构造函数生成空的通用对象容器，该容器叫做 Object()对象。同样，Array()构造函数生成数组对象，该对象叫做 Array()对象。

在本书中，术语“JavaScript 对象”指的是 JavaScript 中的所有对象，因为 JavaScript 中的大部分值都可以像对象一样使用。这是由于大多数 JavaScript 值都是由能够生成特定类型对象的原生构造函数所创建的。

要记住的是，Object()对象是一种特定类型的值，它是一个通用的空对象。不要将其与术语“JavaScript 对象”混淆，“JavaScript 对象”用于表示 JavaScript 中可以作为对象的绝大多数值。

JavaScript 启示录

如果你是一名JavaScript新手或中级开发人员，本书将会帮助你巩固对JavaScript的理解，特别是当你使用JavaScript库的时候。在这本简明扼要的图书中，JavaScript专家Cody Lindley通过剖析JavaScript对象及其细微区别，准确地提供了这种语言的详细信息。

库和框架能够帮助我们快速有效地构建Web应用程序，但是当它们出错或性能出问题的时候，知道这些库和框架是如何以及为何能够工作，就显得至关重要了。如果你已经准备好了深入了解JavaScript的内部核心原理，那么本书正是你所需要的。

通过本书，你将能够：

- 得到可以立即运行的真实代码，帮助你概括和理解ECMA-262第3版；
- 剖析JavaScript对象的创建过程；
- 学习复杂值、原始值、作用域和继承特性；
- 理解head对象的重要性；
- 使用字符串、数字、布尔对象和值；
- 了解如何使用null值和内置的math对象；
- 深入了解细节——超越Mozilla的JavaScript 1.5参考指南。

Cody Lindley是一名客户端工程师（也称为前端开发人员）及Flash开发者。他在HTML、CSS、JavaScript、Flash、客户端性能技术方面有丰富的工作经验（超过13年），这些技术都与Web开发有关。他还是《jQuery Cookbook》一书的作者。

O'REILLY®
oreilly.com.cn

封面设计：Karen Montgomery，张健

O'Reilly Media, Inc. 授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

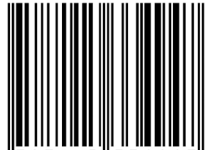
This Authorized Edition for sale only in the territory of People's Republic of China

(excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/程序设计/JavaScript

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33494-7



9 787115 334947 >