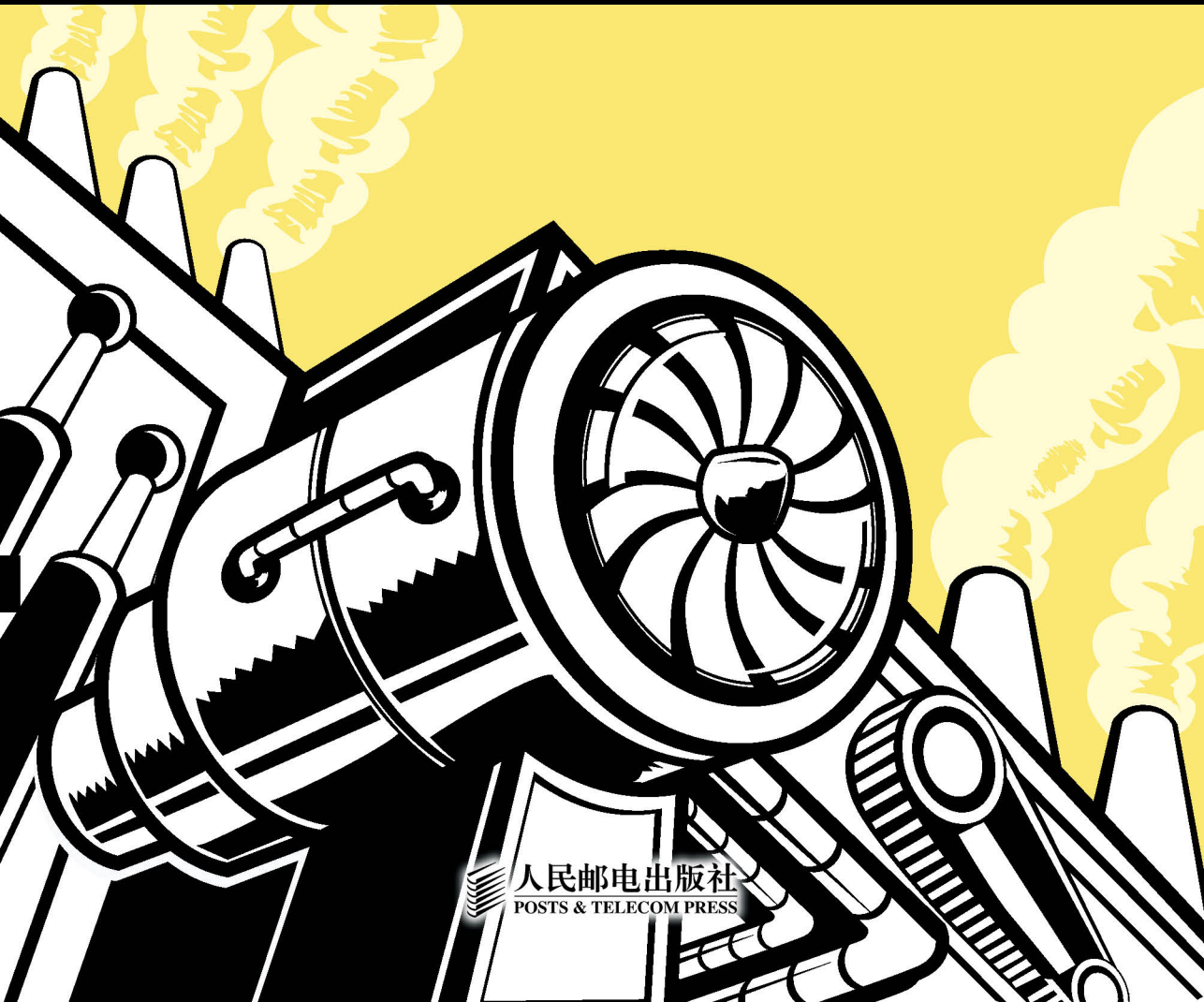
畅销书《JavaScript高级程序设计》作者Nicholas C. Zakas最新力作
《JavaScript启示录》和《jQuery Cookbook》作者Cody Lindley作序推荐

no starch press

JavaScript 面向对象精要

THE PRINCIPLES OF OBJECT-ORIENTED JAVASCRIPT

[美] Nicholas C. Zakas 著　　胡世杰 译



人民邮电出版社
POSTS & TELECOM PRESS

JavaScript 面向对象精要

［美］Nicholas C.Zakas　著

胡世杰　译

人 民 邮 电 出 版 社

北 京

版权声明

◆ 著　　［美］Nicholas C.Zakas
译　　胡世杰
责任编辑　陈冀康
责任印制　张佳莹
◆ 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
北京　　印刷有限公司印刷
◆ 开本：720×960　1/16
印张：7
字数：117 千字　　2015 年 3 月第 1 版
印数：1-0 000 册　　2015 年 3 月北京第 1 次印刷
著作权合同登记号　图字：01-2013-7464 号

定价：00.00 元

读者服务热线：（010）81055410　印装质量热线：（010）81055316
反盗版热线：（010）81055315

内容提要

本书关注面向对象的原理和 ES5 对象新特性，目的是帮助那些已经熟知面向对象编程的读者掌握这些概念是如何在 JavaScript 中工作的。

本书内容简洁而精妙。全书共 6 章，分别深入探讨了原始类型和引用类型、函数、对象、构造函数和原型对象、继承和对象模式等主题和特性。在本书中，你将学到 JavaScript 独特的面向对象的编程方式；抛弃类的概念和基于类的继承，学习基于原型的继承和构造函数。你将学会如何创建对象、定义自己的类型、使用继承以及其他各种操作来充分使用对象。总而言之，你将学到用 JavaScript 语言进行专业编程所需熟知的一切。

本书适合熟悉面向对象编程的概念并希望将其应用于 JavaScript 的开发者阅读，也适合 JavaScript 新手学习参考。

译者序

JavaScript 是一门流行多年的语言，到现在依然有着旺盛的生命力。这与无数开发人员的使用和完善是分不开的。希望本书能够为这些开发者开阔视野，带来一些更深层次的领悟。

本书作者是一位非常资深的 JavaScript 开发者、作者以及演讲者。他能够深入浅出地介绍 JavaScript 的面向对象的特性，使得本书尤其适用于那些已经熟悉面向对象或 JavaScript 两者之一的开发者。

在你阅读这本书时，就好像听见作者将 JavaScript 中面向对象的特征性娓娓道来，看见一幅精准的设计蓝图在面前徐徐展现，让你能够自始至终都沉醉在优美的技术世界里无法自拔，感受到每一个简洁设计背后的意图，甚至不由自主地去思考为什么最终的设计是现在这个样子？如果是让自己来设计又会做成什么样子？会不会有更好的设计？

无论你的答案如何，作者的目的都达到了。作者不仅是要告诉读者 JavaScript 是如何实现面向对象的特性的，更重要的是引导每一位读者思考为什么如此实现。而这，才是最有价值的东西。

这是本人的第一本译作。感谢好友高博推荐我这个机会，让我能够进入技术书籍的翻译这样一个领域。感谢陈冀康编辑的辛勤工作，这本译作的问世离不开他的审阅和校对。感谢好友徐章宁在翻译本书过程中给予我的建议，最后感谢我挚爱的妻子黄静对我由于倾力翻译而疏于家事的体谅和支持。希望本书的出版能够给你们带来快乐。

胡世杰
于上海
2015 年 1 月

作者简介

Nicholas C. Zakas 是 Box 公司的一位软件工程师。他因 JavaScript 最佳实践的写作和演讲而知名。在此之前，他作为 Yahoo！主页的资深前端工程师，在雅虎有 5 年的工作经验。他写了好几本书，包括 *Maintainable JavaScript*（O'Reilly Media, 2012）和 *Professional JavaScript for Web Developers*（Wrox, 2012）。

技术评审者简介

来自英国的 Angus Croll 现在是位于旧金山的推特的网页框架团队的一员，同时也是推特的开源框架 Flight 的合作开发者和主要维护者。他对 JavaScript 和文学有同等的兴趣爱好，并热情拥护艺术家和有创造力的思想家们更多地参与软件开发。Angus 经常在世界各地的研讨会上发言，现正在为 No Starch Press 编写两本书。

译者简介

胡世杰，1981 年生，硕士毕业于上海交通大学。目前在 EMC 中国卓越研发集团任高级开发工程师，在软件开发领域有超过 10 年的经验。对各种主流开发语言均有涉猎。平时热爱看书和游戏，是婚姻生活中的情感专家和沟通大师。

序

Nicholas Zakas 这个名字就是 JavaScript 开发的代名词。我可以用数页的篇幅漫谈他获得的专业荣誉，但我不准备这么做。Nicholas 是一位非常著名的顶级 JavaScript 开发者及作者，他不需要介绍。不过，在赞扬这本书之前，我打算先谈一些个人感受。

我和 Nicholas 的关系来自多年来以一个 JavaScript 小学生的身份研究他的书，读他的博客，听他的演讲，关注他的推特更新。我们的第一次会面是在几年前，当时我请他在一个 jQuery 研讨会上发言。结果他做了一场高质量的演讲。此后，我们一直在网络上保持着密切的联系。从那时起，我就越来越崇拜他，不仅仅因为他是 JavaScript 委员会的一位领导者和开发者。他的语言是如此优雅和使人深思，他的行为是如此亲切。他永远是一位乐于助人的开发者、诲人不倦的演讲者和超越自我的作者。当他演讲时，你应该倾听，不仅因为他是一位 JavaScript 专家，更因为他高尚的人品还在他的专业地位之上。

本书的标题和简介清楚地展现了 Nicholas 的意图：他写这本书是为了帮助那些对类（也就是 C++或 Java）习以为常的程序员转换到一个没有类的语言上去。他在本书中解释了如何在 JavaScript 中实现封装、聚合、继承以及多态。有基础的程序员想要转投面向对

象的 JavaScript 开发，这是一本理想的教材。如果你是一位其他语言的开发者，你会发现这是一本语言简洁而精妙的 JavaScript 方面的书。

不过，本书同时也为 JavaScript 程序员服务。很多 JavaScript 开发者对对象的理解仍停留在 ECMAScirpt 3（ES3）的版本，他们需要适当了解 ECMAScript 5（ES5）版本的对象功能。本书就可以满足他们的需求，帮他们弥补 ES3 对象和 ES5 对象之间的知识空白。

现在你可能在想，“没啥了不起的。介绍 ES5 的 JavaScript 书已经有好几本了。”好吧，确实如此。不过我相信本书是目前唯一一本在整个叙述过程中都将 ES5 对象视为一等公民并时刻深入对象本质的著作。本书不仅对 ES5 对象进行介绍，还介绍了在你学习 ES5 新特性的时候所需要了解的 ES3 和 ES5 之间的差异。

这是一本关注面向对象的原理和 ES5 对象新特性的书。作为一名作者，我坚信本书是我们在等待脚本环境更新至 ES6 之前的必读之作。

Cody Lindley（www.codylindley.com）

JavaScript Enlightenment[1]，*DOM Enlightenment* 和

jQuery Enlightenment 的作者

于博伊西，爱达荷

2013 年 12 月 16 日

[1] 编者注：中文版《Java Script 启示录》由人民邮电出版社于 2014 年 3 月出版（ISBN 978-115-33494-7，定价 35 元）。

致　　谢

我要感谢 Kate Matsudaira 让我相信自助出版一本 Ebook 是传递信息最好的方式。没有她的建议，我可能还在想我该如何处理这本书中包含的信息。

感谢 Rob Friesel 再次对本书的初稿提供的精彩反馈，以及 Cody Lindley 提供的建议。感谢 Angus Croll 对本书的定稿提供的技术评审——他的精益求精为本书增色不少。

感谢我在一次研讨会上遇到的 Bill Pollock，是他推动了这本书的出版。

前　言

大多数开发者将面向对象的编程联想为那些在学校中学到的基于类的语言，比如 C++和 Java。在用这些语言完成任务之前，必须先创建类，哪怕只是写一个简单的命令行程序。目前业界常用的设计模式也强调了基于类的概念。JavaScript 不使用类，这也是人们在学了 C++或 Java 之后再学习 JavaScript 时感到困惑的原因。

面向对象的语言有如下几种特性。

封装　数据可以和操作数据的功能组织在一起。这就是对象的定义，十分简单。

聚合　一个对象能够引用另一个对象。

继承　一个新创建的对象和另一个对象拥有同样的特性，而无需显式复制其功能。

多态　一个接口可被多个对象实现。

JavaScript 拥有上述全部特性，因为语言本身没有类的概念，所以某些特性可能不是以你所期望的方式实现的。乍一看，一个 JavaScript 程序可能像是一个用 C 来编写的面向过程的程序。如果写一个函数并传递一些参数，就有了一个看上去没有对象也可工作的脚本。但是仔细观察，你就会在点号的使用上发现对象的

存在。

很多面向对象的语言使用点号来访问对象的属性和方法，JavaScript 也不例外。但是你永远不需要在 JavaScript 中写一个类定义，导入一个包或包含一个头文件。你只是用你需要的数据类型开始编写代码，然后有无数种方法可以把它们组织在一起。可以用面向过程的方式编写 JavaScript，但它真正的威力是在你利用其面向对象的特性时才能展现的。而这就是本书要告诉你的。

不要搞错：你在传统的面向对象的语言中学到的很多概念都不一定适用于 JavaScript。初学者往往对此迷惑不已。你在阅读的过程中会迅速发现 JavaScript 的弱类型特性允许你用比其他语言更少的代码完成同样的任务。你无需预先设计好类就可以开始编写代码。需要一个具有某个字段的对象了？随时随地都可创建。忘了给那个对象添加一个方法？没关系——以后补上。

在本书中，你将学到 JavaScript 独特的面向对象的编程方式。抛弃类的概念和基于类的继承，学习基于原型的继承和功能类似的构造函数。你将学会如何创建对象，定义自己的类型，使用继承以及其他各种操作来充分使用对象。一句话，你将从专业级别理解和使用 JavaScript 程序所需知道的一切。享受它吧！

本书目标读者

本书的目的是帮助那些已经熟知面向对象编程的读者掌握这些概念是如何在 JavaScript 中工作的。只要你熟悉 Java、C#或其他面向对象的编程语言，那么本书就是为你而著。尤其是以下 3 种人群。

- 熟悉面向对象编程的概念并希望将其应用于 JavaScript 的开发者。
- 希望更有效地组织代码的网页应用程序开发者和 Node.js 开发者。
- 想深入了解 JavaScript 的开发新手。

本书不是写给那些从未编写过 JavaScript 的新手的。你需要对

如何编写和执行 JavaScript 代码有一个清楚的认识才能跟得上。

概览

第 1 章“原始类型和引用类型”介绍了 JavaScript 中的两种数据类型：原始类型和引用类型。你会学到它们之间的区别，知道为什么理解它们之间的区别对于理解整个 JavaScript 非常重要。

第 2 章“函数”解释了 JavaScript 函数的输入和输出。函数是 JavaScript 的一等公民，它们使得 JavaScript 成为一门有趣的语言。

第 3 章“理解对象”深度探索 JavaScript 对象的组成。JavaScript 对象的行为和其他语言的对象不同，深入了解对象的工作原理是掌握 JavaScript 语言的关键。

第 4 章“构造函数和原型对象”将目光聚焦于构造函数，拓展了之前对函数的讨论。所有的构造函数都是函数，但它们在使用上有一点区别。本章在探索这些区别之外还讨论了如何创建自定义类型。

第 5 章“继承”解释了 JavaScript 中的继承是如何实现的。JavaScript 里没有类，却不代表 JavaScript 里不能继承。在本章，你将学到原型继承以及它和类继承的区别。

第 6 章“对象模式”带你浏览常用的对象模式。JavaScript 拥有很多不同的方式来创建和组合对象，本章为你介绍其中最流行的几种模式。

帮助与支持

如果你对本书有任何疑问、评论或其他反馈，请访问 http://groups.google.com/group/zakasbooks 上的邮件列表。

目　　录

第 1 章

原始类型和引用类型

大多数开发者在使用 Java 或 C#等基于类的语言的过程中学会了面向对象编程。由于 JavaScript 没有对类的正式支持，这些开发者在学习 JavaScript 时往往会迷失方向。JavaScript 不需要在开头就定义好各种类，你可以在写代码的过程中根据需要创建数据结构。由于 JavaScript 缺少类，也就缺少用于对类进行分组的包。在 Java 中，包和类的名字不仅定义了对象的类型，也在工程中列出文件和目录的层次结构，JavaScript 编程就好像从一块空白石板开始：你可以在上面组织任何你想要的东西。些开发者选择模仿其他语言的结构，也有一些人则利用 JavaScript 的灵活性来创建一些全新的东西。对没有掌握 JavaScript 的人来说，这种选择的自由令人崩溃，然而一旦你熟悉了它，你会发现 JavaScript 是一种无比灵活的语言，可以很轻松地适应你的编程习惯。

为了便于开发者从传统的面向对象语言过渡，JavaScript 把对象作为语言的中心。几乎所有 JavaScript 的数据要么是一个对象要么从对象中获取。其实就连函数在 JavaScript 中也被视为对象，这使得它们成为 JavaScript 的一等公民。

使用和理解对象是理解整个 JavaScript 的关键。你可以在任何时候创建对象，在任何时候给对象添加、删除属性。JavaScript 对象是如此灵活，可以创造出其他语言不具有的独特而有趣的模式。

本章致力于鉴别和使用两种 JavaScript 基本数据类型：原始类型和引用类型。虽然两者都通过对象进行访问，但是理解它们行为之间的区别是非常重要的。

1.1 什么是类型

JavaScript 虽然没有类的概念，但依然存在两种类型：原始类型和引用类型。原始类型保存为简单数据值。引用类型则保存为对象，其本质是指向内存位置的引用。

为了让开发者能够把原始类型和引用类型按相同方式处理，JavaScript 花费了很大努力来保证语言的一致性。

其他编程语言用栈储存原始类型，用堆储存引用类型，JavaScript 则完全不同：它使用一个变量对象追踪变量的生存期。原始值被直接保存在变量对象内，而引用值则作为一个指针保存在变量对象内，该指针指向实际对象在内存中的存储位置。虽然看上去原始值和引用值一样，但是它们还是有区别的，本章稍后会介绍。

当然，原始类型和引用类型还有其他区别。

1.2 原始类型

原始类型代表照原样保存的一些简单数据，如 true 和 25。JavaScript 共有 5 种原始类型，如下。

boolean	布尔，值为 true 或 false
number	数字，值为任何整型或浮点数值
string	字符串，值为由单引号或双引号括出的单个字符或连续字符（JavaScript 不区分字符类型）
null	空类型，该原始类型仅有一个值：null
undefined	未定义，该原始类型仅有一个值：undefined（undefined 会被赋给一个还没有初始化的变量）

前 3 种类型（boolean，number 和 string）表现的行为类似，而后 2 种（null 和 undefined）则有一点区别，本章后面将会讨论。所有原始类型的值都有字面形式。字面形式是不被保存在变量中的值，如硬编码的姓名或价格。下面是每种类型使用字面形式的例子。

```
// strings
var name = "Nicholas";
var selection = "a";

// numbers
var count = 25;
var cost = 1.51;

// boolean
var found = true;

// null
var object = null;

// undefined
var flag = undefined;
var ref; // assigned undefined automatically
```

JavaScript 和许多其他语言一样，原始类型的变量直接保存原始值（而不是一个指向对象的指针）。当你将原始值赋给一个变量时，该值将被复制到变量中。也就是说，如果你使一个变量等于另一个时，每个变量有它自己的一份数据拷贝。例如，

```
var color1 = "red";
var color2 = color1;
```

这里，color1 被赋值为“red”。变量 color2 被赋予 color1 的值，这样变量 color2 中就保存了“red”。虽然 color1 和 color2 具有同样的值，但是两者毫无关联，改变 color1 的值不会影响 color2，反之亦然。这是因为存在两个不同的储存地址，每个变量拥有一个。图 1-1 展示了这段代码的变量对象。

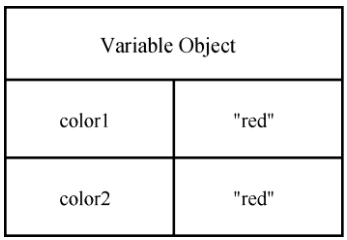


图 1-1 变量对象

因为每个含有原始值的变量使用自己的存储空间，一个变量的改变不会影响到其他变量。例如，

```
var color1 = "red";
var color2 = color1;

console.log(color1);   // "red"
console.log(color2);   // "red"

color1 = "blue";

console.log(color1);   // "blue"
console.log(color2);   // "red"
```

在这段代码中，color1 被改为“blue”，而 color2 还保有原来的值“red”。

1.2.1 鉴别原始类型

鉴别原始类型的最佳方法是使用 typeof 操作符。它可以被用在任何变量上，并返回一个说明数据类型的字符串。Typeof 操作符可用于字符串、数字、布尔和未定义类型。下面是 typeof 对不同原始类型的输出。

```
console.log(typeof "Nicholas");   // "string"
console.log(typeof 10);           // "number"
console.log(typeof 5.1);          // "number"
console.log(typeof true);         // "boolean"
console.log(typeof undefined);    // "undefined"
```

正如我们所期望的，对于字符串，typeof 将返回“string”，对于数字将返回“number”（无论整型还是浮点数），对于布尔类型将返回“Boolean”，对于未定义类型将则返回“undefined”。

至于空类型则有一些棘手。

下面那行代码的运行结果困扰了很多开发者。

```
console.log(typeof null);          // "object"
```

当你运行 typeof null 时，结果是“object”。但这是为什么呢？（其实这已经被设计和维护 JavaScript 的委员会 TC39 认定是一个错误。在逻辑上，你可以认为 null 是一个空的对象指针，所以结果为“object”，但这还是很令人困惑。）

判断一个值是否为空类型的最佳方法是直接和 null 比较，如下。

```
console.log(value === null);       // true or false
```

非强制转换比较

注意这段代码使用了三等号操作符（===）而不是双等号。原因是三等号在进行比较时不会将变量强制转换为另一种类型。为了理解这点，请看下面的例子。

```
console.log("5" == 5);               // true
console.log("5" === 5);              // false

console.log(undefined == null);      // true
console.log(undefined === null);     // false
```

当你使用双等号进行比较时，双等号操作符会在比较之前把字符串转换成数字，因此认为字符串“5”和数字 5 相等。三等号操作符认为这两个值的类型不同，因此不相等。同样原因，当你比较 undefined 和 null 时，双等号认为它们相等而三等号认为不相等。当你试图鉴别 null 时，使用三等号才能让你正确鉴别出类型。

1.2.2 原始方法

虽然字符串、数字和布尔是原始类型，但是它们也拥有方法（null 和 undefined 没有方法）。特别是字符串有很多方法，可以帮助你更好地使用它们。例如，

```
var name = "Nicholas";
var lowercaseName = name.toLowerCase();     // convert to lowercase
var firstLetter = name.charAt(0);           // get first character
var middleOfName = name.substring(2, 5);    // get characters 2-4

var count = 10;
var fixedCount = count.toFixed(2);          // convert to "10.00"
var hexCount = count.toString(16);          // convert to "a"

var flag = true;
var stringFlag = flag.toString();           // convert to "true"
```

注意 尽管原始类型拥有方法，但它们不是对象。JavaScript 使它们看上去像对象一样，以此来提供语言上的一致性体验，你会在本章后面看到这点。

1.3 引用类型

引用类型是指 JavaScript 中的对象，同时也是你在该语言中能找到的最接近类的东西。引用值是引用类型的实例，也是对象的同义词（本章后面将用对象指代引用值）。对象是属性的无序列表。属性包含键（始终是字符串）和值。如果一个属性的值是函数，它就被称为方法。JavaScript 中函数其实是引用值，除了函数可以运行以外，一个包含数组的属性和一个包含函数的属性没有什么区别。

当然，在使用对象前，你必须先创建它们。

1.3.1 创建对象

有时候，把 JavaScript 对象想象成图 1-2 中的哈希表可以帮助你更好地理解对象结构。

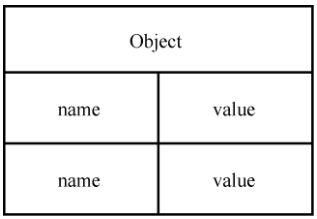


图 1-2 对象结构

JavaScript 有好几种方法可以创建

对象，或者说实例化对象。第一种是使用 new 操作符和构造函数。构造函数就是通过 new 操作符来创建对象的函数——任何函数都可以是构造函数。根据命名规范，JavaScript 中的构造函数用首字母大写来跟非构造函数进行区分。例如下列代码实例化一个通用对象，并将它的引用保存在 object 中。

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

因为引用类型不在变量中直接保存对象，所以本例中的 object 变量实际上并不包含对象的实例，而是一个指向内存中实际对象所在位置的指针（或者说引用）。这是对象和原始值之间的一个基本差别，原始值是直接保存在变量中的。

当你将一个对象赋值给变量时，实际是赋值给这个变量一个指针。这意味着，将一个变量赋值给另一个变量时，两个变量各获得了一份指针的拷贝，指向内存中的同一个对象。例如，

```
var object1 = new Object();
var object2 = object1;
```

这段代码先用 new 创建了一个对象并将其引用保存在 object1 中。然后将 object1 的值赋值给 object2。两个变量都指向第一行被创建的那个对象实例，如图 1-3 所示。

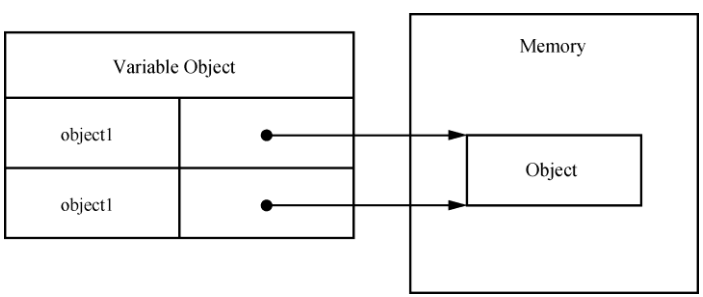


图 1-3 两个变量指向同一个对象

1.3.2 对象引用解除

JavaScript 语言有垃圾收集的功能，因此当你使用引用类型时无需担心内存分配。但最好在不使用对象时将其引用解除，让垃圾收集器对那块内存进行释放。解除引用的最佳手段是将对象变量置为 null。

```
var object1 = new Object();

// do something

object1 = null; // dereference
```

这里，对象 object1 被创建然后使用，最后设置为 null。当内存中的对象不再被引用后，垃圾收集器会把那块内存挪作它用（在那些使用几百万对象的巨型程序里，对象引用解除尤其重要）。

1.3.3 添加删除属性

在 JavaScript 中，对象另一个有趣的方面是你可以随时添加和删除其属性。例如，

```
var object1 = new Object();
var object2 = object1;

object1.myCustomProperty = "Awesome!";
console.log(object2.myCustomProperty); // "Awesome!"
```

这里，object1 上增加了 myCustomProperty 属性，值为“Awesome!”。该属性也可以被 object2 访问，因为 object1 和 object2 指向同一个对象。

注意 本例演示了 JavaScript 的一个独特的方面：可以随时修改对象，即使并没有在开始时定义它们。同时，你将会在本书后续内容中看到还存在很多方法阻止此类修改。

除了通用对象引用类型以外，JavaScript 还有其他一些内建类型任你使用。

1.4 内建类型实例化

你已经见过如何用 new Object()创建和使用通用对象。Object 类型只是 JavaScript 提供的少量内建引用类型之一。其他内建类型各有它们的特殊用途，可在任何时候被实例化。

这些内建类型如下。

Array 数组类型，以数字为索引的一组值的有序列表

Date	日期和时间类型
Error	运行期错误类型（还有一些更特别的错误的子类型）
Function	函数类型
Object	通用对象类型
RegExp	正则表达式类型

可以用 new 来实例化每一个内建引用类型，如下。

```
var items = new Array();
var now = new Date();
var error = new Error("Something bad happened.");
var func = new Function("console.log('Hi');");
var object = new Object();
var re = new RegExp("\\d+");
```

1.4.1 字面形式

内建引用类型有字面形式。字面形式允许你在不需要使用 new 操作符和构造函数显式创建对象的情况下生成引用值（你曾在本章前面见过原始类型的字面形式，包括字符串、数字、布尔、空类型和未定义）。

1.4.2 对象和数组字面形式

要用对象字面形式创建对象，可以在大括号内定义一个新对象及其属性。属性的组成包括一个标识符或字符串、一个冒号以及一个值。多个属性之间用逗号分隔。例如，

```
var book = {
    name: "The Principles of Object-Oriented JavaScript",
    year: 2014
};
```

属性名字也可以用字符串表示，特别是当你希望名字中包含空格或其他特殊字符时。

```
var book = {
    "name": "The Principles of Object-Oriented JavaScript",
    "year": 2014
};
```

本例等价于前例，仅在语法上有所区别。下例是另一种等价写法。

```
var book = new Object();
book.name = "The Principles of Object-Oriented JavaScript";
book.year = 2014;
```

上述 3 例的结果是一致的：一个具有两个属性的对象。写法完全取决于你。

注意 虽然使用字面形式并没有调用 new Object()，但是 JavaScript 引擎背后做的工作和 new Object()一样，除了没有调用构造函数。其他引用类型的字面形式也是如此。

定义数组的字面形式是在中括号内用逗号区分的任意数量的值。例如，

```
var colors = [ "red", "blue", "green" ];
console.log(colors[0]); // "red"
```

这段代码等价于：

```
var colors = new Array("red", "blue", "green")
console.log(colors[0]); // "red"
```

1.4.3 函数字面形式

基本上都要用字面形式来定义函数。考虑到在可维护性、易读性和调试上的巨大挑战，通常不会有人使用函数的构造函数，因此很少看到用字符串表示的代码而不是实际的代码。

使用字面形式创建函数更方便也更不容易出错，如下例。

```
function reflect(value) {
    return value;
}

// is the same as

var reflect = new Function("value", "return value;");
```

这段代码定义了 reflect()函数，它的作用是将任何传给它的参数返回。即使是这样一个简单的例子，使用字面形式都比构造函数的形式方便和易读。另外，用构造函数创建的函数没什么好的调试方法：JavaScript 调试器认不出这些函数，它们在程序里就好像黑盒一样。

1.4.4 正则表达式字面形式

JavaScript 允许用字面形式而不是使用 RegExp 构造函数定义正则表达式。它们看上去类似 Perl 中的正则表达式：模式被包含在两个“/”之间，第二个“/”后是由单字符表示的额外选项。例如，

```
var numbers = /\d+/g;

// is the same as

var numbers = new RegExp("\\d+", "g");
```

使用字面形式比较方便的一个原因是你不需要担心字符串中的转义字符。如果使用 RegExp 构造函数，传入模式的参数是一个字符串，你需要对任何反斜杠进行转义（这就是为什么字面形式使用“\d”而构造函数使用“\\”d 的原因）。在 JavaScript 中，除非需要通过一个或多个字符串动态构造正则表达式，否则都建议使用字面形式而不是构造函数。

总之，除了函数，对内建类型没什么正确或错误的实例化方法。很多开发者喜欢字面形式，另一些则喜欢用构造函数。你可以选择能令你觉得更舒服的那种。

1.5 访问属性

属性是对象中保存的名字和值的配对。点号是 JavaScript 中访问属性的最通用做法（就跟许多面向对象语言一样），不过也可以用中括号访问 JavaScript 对象的属性。

例如，下面的代码使用点号。

```
var array = [];
array.push(12345);
```

也可以如下例用中括号，方法的名字现在由中括号中的字符串表示。

```
var array = [];
array["push"](12345);
```

在需要动态决定访问哪个属性时，这个语法特别有用。例如下例的中括号允许你用变量而不是字符串字面形式来指定访问的属性。

```
var array = [];
var method = "push";
array[method](12345);
```

在这段代码中，变量 method 的值是“push”，因此在 array 上调用了 push()方法。这种能力极其有用，你会在本书中随处看到这种用法。记住一点：除了语法不同，在性能或其他方面点号和中括号都大致相同，唯一区别在于中括号允许你在属性名字上使用特殊字符。开发者通常认为点号更易读，所以你更多地看到点号而不是中括号。

1.6 鉴别引用类型

函数是最容易鉴别的引用类型，因为对函数使用 typeof 操作符时，返回值是“function”。

```
function reflect(value) {
    return value;
}

console.log(typeof reflect); // "function"
```

对其他引用类型的鉴别则较为棘手，因为对于所有非函数的引用类型，typeof 返回“object”。在处理很多不同类型的时候这帮不上什么忙。为了更方便地鉴别引用类型，可以使用 JavaScript 的 instanceof 操作符。

instanceof 操作符以一个对象和一个构造函数为参数。如果对象是构造函数所指定的类型的一个实例，instanceof 返回 true；否则返回 false，如下例。

```
var items = [];
var object = {};

function reflect(value) {
    return value;
}

console.log(items instanceof Array);        // true
console.log(object instanceof Object);      // true
console.log(reflect instanceof Function);   // true
```

本例用 instanceof 和构造函数测试了几个值，它们真正的类型都被正确鉴别出来（即使该构造函数并没有被用于创建该变量）。

instanceof 操作符可鉴别继承类型。这意味着所有对象都是 Object 的实例，因为所有引用类型都继承自 Object。

作为示范，下列代码用 instanceof 检查了之前那 3 种引用。

```
var items = [];
var object = {};

function reflect(value) {
    return value;
}

console.log(items instanceof Array);         // true
console.log(items instanceof Object);        // true
console.log(object instanceof Object);       // true
console.log(object instanceof Array);        // false
console.log(reflect instanceof Function);    // true
console.log(reflect instanceof Object);      // true
```

每种引用类型的对象都被正确鉴别为 Object 的实例。

1.7 鉴别数组

虽然 instanceof 可以鉴别数组，但是有一个例外会影响网页开发者：JavaScript 的值可以在同一个网页的不同框架之间传来传去。当你试图鉴别一个引用值的类型时，这就有可能成为一个问题，因为每一个页面拥有它自己的全局上下文——Object、Array 以及其他内建类型的版本。结果，当你把一个数组从一个框架传到另一个框架时，instanceof 就无法识别它，因为那个数组是来自不同框架的 Array 的实例。

为了解决这个问题，ECMAScript 5 引入了 Array.isArray()来明确鉴别一个值是否为 Array 的实例，无论该值来自哪里，该方法对来自任何上下文的数组都返回 true。如果你的环境兼容 ECMAScript 5，Array.isArray()是鉴别数组的最佳方法。

```
var items = [];

console.log(Array.isArray(items)); // true
```

大多数环境都在浏览器和 Node.js 中支持 Array.isArray()方法。IE8 或更早的版本不支持该方法。

1.8 原始封装类型

JavaScript 中一个最让人困惑的部分可能就是原始封装类型的概念。原始封装类型共有 3 种（String、Number 和 Boolean）。这些特殊引用类型的存在使得原始类型用起来和对象一样方便。(如果你不得不用独特的语法或切换为基于过程的编程方式来获取一个子字符串，那就太让人困惑啦)。

当读取字符串、数字或布尔值时，原始封装类型将被自动创建。例如，下列代码第一行，一个原始字符串的值被赋给 name。第二行代码把 name 当成一个对象，使用点号调用了 charAt 方法。

```
var name = "Nicholas";
var firstChar = name.charAt(0);
console.log(firstChar);                 // "N"
```

这是在背后发生的事情如下。

```
// what the JavaScript engine does
var name = "Nicholas";
var temp = new String(name);
var firstChar = temp.charAt(0);
temp = null;
console.log(firstChar);                 // "N"
```

由于第二行把字符串当成对象使用，JavaScript 引擎创建了一个字符串的实体让 charAt(0)可以工作。字符串对象的存在仅用于该语句并在随后被销毁（一种称为自动打包的过程）。为了测试这一点，试着给字符串添加一个属性看看它是不是对象。

```
var name = "Nicholas";
name.last = "Zakas";

console.log(name.last);                 // undefined
```

这段代码试图给字符串 name 添加 last 属性。代码运行时没有错误，但是属性却消失了。到底发生了什么？你可以在任何时候给一个真的对象添加属性，属性会保留至你手动删除它们。原始封装类型的属性会消失是因为被添加属性的对象立刻就被销毁了。

下面是在 JavaScript 引擎中实际发生的事情。

```
// what the JavaScript engine does
var name = "Nicholas";
var temp = new String(name);
temp.last = "Zakas";
temp = null;                        // temporary object destroyed

var temp = new String(name);
console.log(temp.last);             // undefined
temp = null;
```

实际上是在一个立刻就被销毁的临时对象上而不是字符串上添加了新的属性。之后当你试图访问该属性时，另一个不同的临时对象被创建，而新属性并不存在。虽然原始封装类型会被自动创建，在这些值上进行 instanceof 检查对应类型的返回值却都是 false。

```
var name = "Nicholas";
var count = 10;
var found = false;

console.log(name instanceof String);   // false
console.log(count instanceof Number);  // false
console.log(found instanceof Boolean); // false
```

这是因为临时对象仅在值被读取时创建。instanceof 操作符并没有真的读取任何东西，也就没有临时对象的创建，于是它告诉我们这些值并不属于原始封装类型。

你也可以手动创建原始封装类型，但有某些副作用。

```
var name = new String("Nicholas");
var count = new Number(10);
var found = new Boolean(false);

console.log(typeof name);              // "object"
console.log(typeof count);             // "object"
console.log(typeof found);             // "object"
```

如你所见，手动创建原始封装类型实际会创建出一个 object，这意味着 typeof 无法鉴别出你实际保存的数据的类型。

另外，使用 String、Number 和 Boolean 对象和使用原始值有一定区别。例如，下列代码使用了 Boolean 对象，对象的值是 false，但 console.log(“Found”)依然会被执行。这是因为一个对象在条件判断语句中总被认为是 true，无论该对象的值是不是等于 false。

```
var found = new Boolean(false);
if (found) {
    console.log("Found");            // this executes
}
```

手工创建的原始封装类型在其他地方也很容易让人误解，在大多数情况下都只会导致错误。所以，除非有特殊情况，你应该避免这么做。

1.9 总结

JavaScript 中虽然没有类，但是有类型。每个变量或数据都有一个对应的原始类型或引用类型。5 种原始类型（字符串、数字、布尔、空类型以及未定义）的值会被直接保存在变量对象中。除了空类型，都可以用 typeof 来鉴别。空类型必须直接跟 null 进行比较才能鉴别。

引用类型是 JavaScript 中最接近类的东西，而对象则是引用类型的实例。可以用 new 操作符或字面形式创建新对象。通常可以用点号访问属性和方法，也可以用中括号。函数在 JavaScript 中也是对象，可以用 typeof 鉴别它们。至于其他引用类型，你应该用 instanceof 和一个构造函数来鉴别。

为了让原始类型看上去更像引用类型，JavaScript 提供了 3 种原始封装类型：String、Number 和 Boolean。JavaScript 会在背后创建这些对象使得你能够像使用普通对象那样使用原始值，但这些临时对象在使用它们的语句结束时就立刻被销毁。虽然你也可以自己创建原始封装类型的实例，但是它们太容易令人误解，所以最好别这么干。