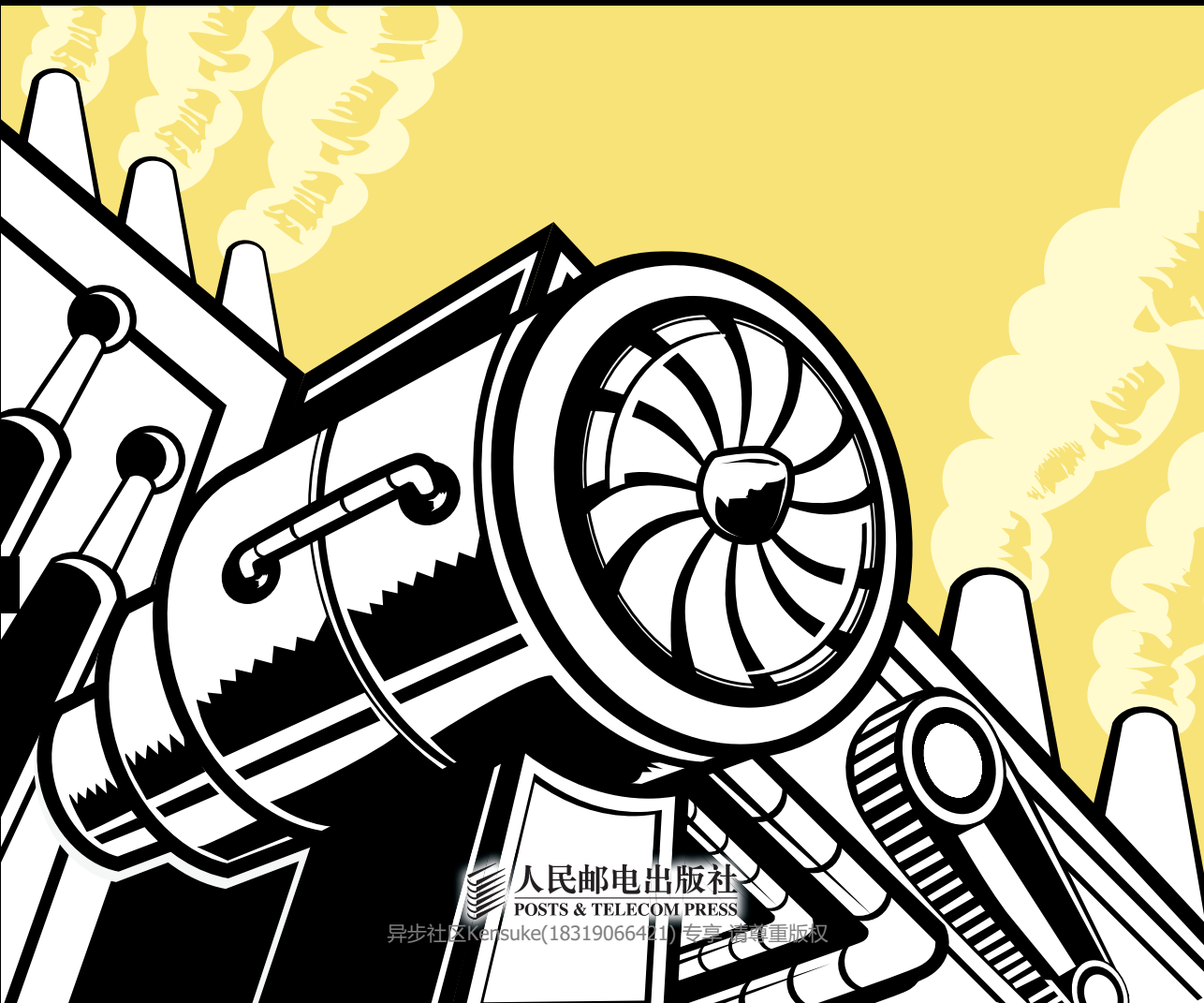
畅销书《JavaScript高级程序设计》作者Nicholas C. Zakas最新力作
《JavaScript启示录》和《jQuery Cookbook》作者Cody Lindley作序推荐

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

THE PRINCIPLES OF
OBJECT-ORIENTED
JAVASCRIPT

[美] Nicholas C. Zakas 著　　胡世杰 译

人 民 邮 电 出 版 社
北 京

异步社区电子书

感谢您购买异步社区电子书！异步社区已上架电子书 500 余种，社区还会经常发布福利信息，对社区有贡献的读者赠送免费样书券、优惠码、积分等等，希望您在阅读过程中，把您的阅读体验传递给我们，让我们了解读者心声，有问题我们会及时修正。

图书在版编目（CIP）数据

JavaScript面向对象精要 / (美) 尼古拉斯(Nicholas,C.Z.) 著 ; 胡世杰译. -- 北京 : 人民邮电出版社, 2015.4
ISBN 978-7-115-38384-6

Ⅰ. ①J… Ⅱ. ①尼… ②胡… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第026626号

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　焦志炜
◆ 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.ptptress.com.cn
三河市中晟雅豪印务有限公司印刷
◆ 开本：720×960　1/16
印张：7.25
字数：117 千字　2014 年 4 月第 1 版
印数：1 – 3 500 册　2014 年 4 月河北第 1 次印刷
著作权合同登记号　图字：01-2013-7464 号

定价：39.00 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。一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。

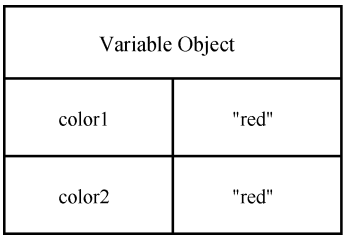


图 1-1 变量对象

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。

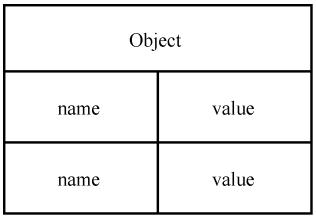


图 1-2 对象结构

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。

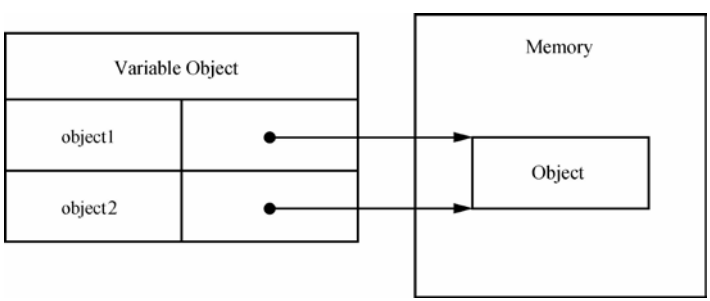


图 1-3 两个变量指向同一个对象

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。

第 2 章

函　数

我们已经在第 1 章讨论过，在 JavaScript 中，函数其实就是对象。使函数不同于其他对象的决定性特点是函数存在一个被称为[[Call]]的内部属性。内部属性无法通过代码访问而是定义了代码执行时的行为。ECMAScript 为 JavaScript 的对象定义了多种内部属性，这些内部属性都用双重中括号来标注。

[[Call]]属性是函数独有的，表明该对象可以被执行。由于仅函数拥有该属性，ECMAScript 定义 typeof 操作符对任何具有[[Call]]属性的对象返回“function”。这在过去曾经导致一些问题，因为某些浏览器曾经在正则表达式中包含[[Call]]属性，导致后者被错误鉴别为函数。现在，所有的浏览器行为都一致，typeof 不会再将正则表达式鉴别为函数了。

本章讨论在 JavaScript 中定义和执行函数的各种方法。由于函数是对象，它们的行为和其他语言中函数的行为不同，理解函数的行为是理解 JavaScript 的核心。

2.1 声明还是表达式

函数具有两种字面形式。第一种是函数声明，以 function 关键字开头，后面跟着函数的名字。函数的内容放在大括号内，例如，下面就是函数声明。

```
function add(num1, num2) {
    return num1 + num2;
}
```

第二种形式是函数表达式，function 关键字后面不需要加上函数的名字。这种函数被称为匿名函数，因为函数对象本身没有名字。取而代之的函数表达式通常会被一个变量或属性引用，下面就是函数表达。

```
var add = function(num1, num2) {
    return num1 + num2;
};
```

这段代码实际上将一个函数作为值赋值给变量 add。除了没有函数名并在最后多了一个分号以外，函数表达式几乎和函数声明完全一样。函数表达式的赋值通常在最后有一个分号，就如同其他对象的赋值一样。

虽然这两种形式颇为相似，但是它们有一个非常重要的区别。函数声明会被提升至上下文（要么是该函数被声明时所在的函数的范围，要么是全局范围）的顶部。这意味着你可以先使用函数后声明它们。例如，

```
var result = add(5, 5);

function add(num1, num2) {
    return num1 + num2;
}
```

这段代码看上去似乎会造成错误，但实际上可以工作。那是因为 JavaScript 引擎将函数声明提升至顶部来执行，就好像它被写成如下形式。

```
// how the JavaScript engine interprets the code
function add(num1, num2) {
   return num1 + num2;
}

var result = add(5, 5);
```

JavaScript 能对函数声明进行提升，这是因为引擎提前知道了函数的名字。而函数表达式仅能通过变量引用，因此无法提升。所以下面这段代码会导致错误。

```
// eror!
var result = add(5, 5);

var add = function(num1, num2) {
return num1 + num2;
};
```

只要你始终在使用函数之前定义它们，你就可以随意使用函数声明或表达式。

2.2 函数就是值

函数是 JavaScript 的一大重点，你可以像使用对象一样使用函数。也可以将它们赋给变量，在对象中添加它们，将它们当成参数传递给别的函数，或从别的函数中返回。基本上只要是可以使用其他引用值的地方，你就可以使用函数。这使得 JavaScript 的函数威力无穷。考虑下面的例子。

```
function sayHi() {
   console.log("Hi!");
}

sayHi();        // outputs "Hi!"

var sayHi2 = sayHi;

sayHi2();       // outputs "Hi!"
```

这段代码首先有一个函数声明 sayHi。然后有一个变量 sayHi2 被创建并被赋予 sayHi 的值。sayHi 和 sayHi2 现在指向同一个函数，两者都可以被执行，并具有相同结果。为了更好地理解这点，让我们来看一下用 Function 构造函数重写的具有相同功能的代码。

```
var sayHi = new Function("console.log(\"Hi!\");");

sayHi();         // outputs "Hi!"

var sayHi2 = sayHi;

sayHi2();        // outputs "Hi!"
```

Function 构造函数更加清楚地表明 sayHi 能够像其他对象一样被传来传去。只要你记住函数就是对象，很多行为就变得容易理解了。

例如，你可以将函数当成参数传递给其他的函数。JavaScript 数组的 sort()方法接受一个比较函数作为可选参数。每当数组中两个值需要进行比较时都会调用比较函数。如果第一个值小于第二个，比较函数返回一个负数。如果第一个值大于第二个，比较函数返回一个正数。如果两个值相等，函数返回 0。

在默认情况下，sort()将数组中每个对象转换成字符串然后进行比较。这意味着，你无法在不指定比较函数的情况下为数字的数组精确排序。例如，

```
var numbers = [ 1, 5, 8, 4, 7, 10, 2, 6 ];
numbers.sort(function(first, second) {
   return first - second;
});

console.log(numbers); // "[1, 2, 4, 5, 6, 7, 8, 10]"

numbers.sort();
console.log(numbers); // "[1, 10, 2, 4, 5, 6, 7, 8]"
```

在本例，被传递给 sort()的比较函数其实是一个函数表达式。请注意它没有名字，仅作为引用被传递给另一个函数（这使得它成为匿名函数）。比较函数对两个值进行相减以返回正确的结果。

作为对比，第二次 sort()不使用比较函数。结果和预期的不一样，1 后面跟着的是 10。这是因为默认的比较函数将所有值都转换成字符串进行比较。

2.3 参数

JavaScript 函数的另一个独特之处在于你可以给函数传递任意数量的参数却不造成错误。那是因为函数参数实际上被保存在一个被称为 arguments 的类似数组的对象中。如同一个普通的 JavaScript 数组，arguments 可以自由增长来包含任意个数的值，这些值可通过数字索引来引用。arguments 的 length 属性会告诉你目前有多少个值。

arguments 对象自动存在于函数中。也就是说，函数的命名参数不过是为了方便，并不真的限制了该函数可接受参数的个数。

注意 arguments 对象不是一个数组的实例，其拥有的方法与数组不同，Array.isArray(arguments)永远返回 false。

另一方面，JavaScript 也没有忽视那些命名参数。函数期望的参数个数保存在函数的 length 属性中。还记得吗？函数就是对象，所以它可以有属性。Length 属性表明了该函数的期望参数个数。了解函数的期望参数个数在 JavaScript 中是非常重要的，因为给它传递过多或过少的参数都不会抛出错误。

下面是一个简单的使用 arguments 和函数的期望参数个数的例子。注意实际传入的参数的数量不影响函数的期望参数个数。

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

console.log(reflect("Hi!"));        // "Hi!"
console.log(reflect("Hi!", 25));    // "Hi!"
console.log(reflect.length);   // 1

reflect = function() {
   return arguments[0];
};

console.log(reflect("Hi!"));        // "Hi!"
console.log(reflect("Hi!", 25));    // "Hi!"
console.log(reflect.length);   // 0
```

本例先定义了一个具有单一命名参数的 reflect()函数，但是当有两个参数传递给它时没有任何错误发生。由于只有一个命名参数，length 属性为 1。代码随后重新定义 reflect()为无命名参数的函数，它返回传入的第一个参数 arguments[0]。这个新版本的函数和前一个版本的输出一模一样，但它的 length 为 0。

因为使用了命名参数，reflect()的第一个实现容易理解（和在别的语言里一样）。使用 arguments 对象的版本有点让人莫名其妙，因为没有命名参数，你不得不浏览整个函数体来确定是否使用了参数。这就是为什么许多开发者尽可能避免使用 arguments 的原因。

不过，在某些情况下使用 arguments 比命名参数更有效。例如，假设你想创建一个函数接受任意数量的参数并返回它们的和。因为你不知道会有多少个参数，所以你无法使用命名参数。在这种情况下，使用 arguments 是最好的选择。

```
function sum() {

   var result = 0,
      i = 0,
      len = arguments.length;

   while (i <len) {
      result += arguments[i];
      i++;
}

   return result;
}

console.log(sum(1, 2));        // 3
console.log(sum(3, 4, 5, 6));  // 18
console.log(sum(50));               // 50
console.log(sum());            // 0
```

sum()函数接受任意数量的参数并在 while 循环中遍历它们的值来求和。这就和对一个数组中的数字求和一样。由于 result 初始值为 0，该函数就算没有参数也能正常工作。

2.4 重载

大多数面向对象语言支持函数重载，它能让一个函数具有多个签名。函数签名由函数的名字、参数的个数及其类型组成。因此，一个函数可以有一个接受一个字符串参数的签名和另一个接受两个数字参数的签名。JavaScript 语言根据实际传入的参数决定调用函数的哪个版本。

之前已经提过，JavaScript 函数可以接受任意数量的参数且参数类型完全没有限制。这说明 JavaScript 函数其实根本没有签名，因此也不存在重载。看看当你试图声明两个同名函数会发生什么。

```
function sayMessage(message) {
   console.log(message);
}

function sayMessage() {
   console.log("Default message");
}

sayMessage("Hello!");          // outputs "Default message"
```

如果这是其他的语言，sayMessage(“Hello!”)就会输出“Hello!”。然而在 JavaScript 里，当你试图定义多个同名的函数时，只有最后定义的有效，之前的函数声明被完全删除，只使用最后那个。下面，让我们用对象来帮助理解。

```
var sayMessage = new Function("message", "console.log(message);");

sayMessage = new Function("console.log(\"Default message\");");

sayMessage("Hello!");        // outputs "Default message"
```

这样看代码，前一个函数为什么不工作就一目了然了。对 sayMessage 连续赋了两次函数对象，第一个自然就丢失了。

JavaScript 函数没有签名这个事实不意味着你不能模仿函数重载。你可以用 arguments 对象获取传入的参数个数并决定怎么处理。例如，

```
function sayMessage(message) {

    if (arguments.length === 0) {
        message = "Default message";
}

    console.log(message);
}

sayMessage("Hello!");      // outputs "Hello!"
```

本例中，sayMessage()函数的行为视传入参数的个数而定。如果没有传入参数（arguments.length === 0），那么就使用默认的信息。否则使用第一个传入的参数为信息。和其他语言中的重载相比，这里有更多的人为介入，但是结果是相同的。如果你还想检查不同的数据类型，你可以用 typeof 和 instanceof。

注意 在实际使用中，检查命名参数是否为未定义比依靠 arguments.length 更常见。

2.5 对象方法

第 1 章中介绍了可以在任何时候给对象添加或删除属性。如果属性的值是函数，则该属性被称为方法。你可以像添加属性那样给对象添加方法。例如，在下面代码中，变量 person 被赋予了一个对象的字面形式，包含属性 name 和方法 sayName。

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(person.name);
    }
};

person.sayName();          // outputs "Nicholas"
```

注意定义数据属性和方法的语法完全相同——标识符后面跟着冒号和值。只不过 sayName 的值正好是一个函数。定义好以后你立刻就能在对象上调用方法 person.sayName()。

2.5.1 this 对象

你可能已经注意到前面例子中一些奇怪之处。sayName()方法直接引用了 person.name，在方法和对象间建立了紧耦合。有太多理由证明这是有问题的。首先，如果你改变变量名，你也必须要改变方法中引用的名字。其次，这种紧耦合使得同一个方法很难被不同对象使用。幸好 JavaScript 对此有一个解决办法。

JavaScript 所有的函数作用域内都有一个 this 对象代表调用该函数的对象。在全局作用域中，this 代表全局对象（浏览器里的 window）。当一个函数作为对象的方法被调用时，默认 this 的值等于那个对象。所以你应该在方法内引用 this 而不是直接引用一个对象。前例代码可以改写如下。

```
var person = {
   name: "Nicholas",
   sayName: function() {
      console.log(this.name);
   }
};

person.sayName(); // outputs "Nicholas"
```

这段代码和前面的版本输出相同，但是这一次，sayName()引用 this 而不是 person。这意味着你可以轻易改变变量名，甚至是将该函数用在不同对象上。

```
function sayNameForAll() {
   console.log(this.name);
}

var person1 = {
   name: "Nicholas",
   sayName: sayNameForAll
};

var person2 = {
   name: "Greg",
   sayName: sayNameForAll
};

var name = "Michael";

person1.sayName();   // outputs "Nicholas"
person2.sayName();   // outputs "Greg"

sayNameForAll();          // outputs "Michael"
```

本例先定义函数 sayNameForAll，然后以字面形式创建两个对象以 sayNameForAll 函数作为 sayName 方法。函数就是引用值，所以你可以把它们作为属性值赋给任意个对象。当 person1 调用 sayName()时，输出“Nicholas”；person2 则输出“Greg”。那是因为 this 在函数调用时才被设置，所以 this.name 是正确的。

本例最后部分定义了全局变量 name。全局变量被认为是全局对象的属性，所以当直接调用 sayNameForAll 时输出“Michael”。

2.5.2 改变 this

在 JavaScript 中，使用和操作函数中 this 的能力是良好地面向对象编程的关键。函数会在各种不同上下文中被使用，它们必须到哪都能正常工作。一般 this 会被自动设置，但是你可以改变它的值来完成不同的目标。有 3 种函数方法允许你改变 this 的值。（记住函数是对象，而对象可以有方法，所以函数也有。）

1. call()方法

第一个用于操作 this 的函数方法是 call()，它以指定的 this 值和参数来执行函数。call()的第一个参数指定了函数执行时 this 的值，其后的所有参数都是需要被传入函数的参数。假设你更新 sayNameForAll 让它接受一个参数，代码如下。

```
function sayNameForAll(label) {
    console.log(label + ":" + this.name);
}

var person1 = {
    name: "Nicholas"
};

var person2 = {
    name: "Greg"
};

var name = "Michael";

sayNameForAll.call(this, "global");          // outputs "global:Michael"
sayNameForAll.call(person1, "person1");  // outputs "person1:Nicholas"
sayNameForAll.call(person2, "person2");  // outputs "person2:Greg"
```

在本例中，sayNameForAll 接受一个 label 参数用于输出。然后该函数被调用 3 次。注意调用函数时在函数名后没有小括号，因为它被作为对象访问而不是被执行的代码。第一次调用使用全局 this 并传入参数“global”来输出“global:Michael”。之后两次调用分别使用 person1 和 person2。由于使用了 call()方法，你不需要将函数加入每个对象——你显式指定了 this 的值而不是让 JavaScript 引擎自动指定。

2. apply()方法

apply()是你可以用来操作 this 的第二个函数方法。apply()的工作方式和 call()完全一样，但它只接受两个参数：this 的值和一个数组或者类似数组的对象，内含需要被传入函数的参数（也就是说你可以把 arguments 对象作为 apply()的第二个参数）。你不需要像使用 call()那样一个个指定参数，而是可以轻松传递整个数组给 apply()。除此之外，call()和 apply()表现得完全一样。下例演示了 apply()的用法。

```
function sayNameForAll(label) {
    console.log(label + ":" + this.name);
}

var person1 = {
    name: "Nicholas"
};

var person2 = {
    name: "Greg"
};

var name = "Michael";

sayNameForAll.apply(this, ["global"]);        // outputs "global:Michael"
sayNameForAll.apply(person1, ["person1"]);    //                  outputs
"person1:Nicholas"
sayNameForAll.apply(person2, ["person2"]);    // outputs "person2:Greg"
```

这段代码借用了前例并用 apply()替换了 call()；结果完全相同。你通常会根据你手头已有的数据决定使用哪个方法。如果你已经有一个数组，用 apply()；如果你有的只是一个个单独的变量，则用 call()。

3. bind()方法

改变 this 的第三个函数方法是 bind()。ECMAScript 5 中新加的这个方法和之前那两个颇有不同。按惯例，bind()的第一个参数是要传给新函数的 this 的值。其他所有参数代表需要被永久设置在新函数中的命名参数。你可以在之后继续设置任何非永久参数。

下面的代码演示了两个使用 bind()的例子。创建 sayNameForPerson1()函数并将 person1 绑定为其 this 对象的值。然后创建了 sayNameForPerson2()并将 person2 绑定为其 this 对象的值，“person2”绑定为其第一个参数。

```
function sayNameForAll(label) {
   console.log(label + ":" + this.name);
}

var person1 = {
   name: "Nicholas"
};

var person2 = {
   name: "Greg"
};

// create a function just for person1
var sayNameForPerson1 = sayNameForAll.bind(person1);
sayNameForPerson1("person1");  // outputs "person1:Nicholas"

// create a function just for person2
var sayNameForPerson2 = sayNameForAll.bind(person2, "person2");
sayNameForPerson2();           // outputs "person2:Greg"

// attaching a method to an object doesn't change 'this'
person2.sayName = sayNameForPerson1;
person2.sayName("person2");         // outputs "person2:Nicholas"
```

sayNameForPerson1()没有绑定参数，所以你仍然需要传入 label 参数用于输出。sayNameForPerson2()不仅绑定 this 为 person2，同时也绑定了第一个参数为“person2”。这意味着你可以调用 sayNameForPerson2()而不传入任何额外参数。例子最后将 sayNameForPerson1()设置为 person2 的 sayName 方法。由于其 this 的值已经绑定，所以虽然 sayNameForPerson1 现在是 person2 的方法，它仍然输出 person1.name 的值。

2.6 总结

JavaScript 函数的独特之处在于它们同时也是对象，也就是说它们可以被访问、复制和覆盖，就像其他对象一样。JavaScript 中的函数和其他对象最大的区别在于它们有一个特殊的内部属性[[Call]]，包含了该函数的执行指令。typeof 操作符会在对象内查找这个内部属性，如果找到，它返回“function”。

函数的字面形式有两种：声明和表达式。函数声明是 function 关键字右边跟着函数名。函数声明会被提升至上下文的顶部。函数表达式可被用于任何可以使用值的地方，例如赋值语句、函数参数或另一个函数的返回值。

函数是对象，所以存在一个 Function 构造函数。你可以用 Function 构造函数创建新的函数，不过没有人会建议你这么做，因为它会使你的代码难以理解和调试。但是有时你可能不得不使用这种用法，例如在函数的真实形式直到运行时才能确定的时候。

为了理解 JavaScript 的面向对象编程，你需要好好理解它的函数。因为 JavaScript 没有类的概念，能够帮助你实现聚合和继承的就只有函数和其他对象了。

第 3 章

理解对象

尽管 JavaScript 里有大量内建引用类型，很可能你还是会频繁创建自己的对象。当你在这么做的时候，记得 JavaScript 中的对象是动态的，可在代码执行的任意时刻发生改变。基于类的语言会根据类的定义锁定对象，JavaScript 对象没有这种限制。

JavaScript 编程一大重点就是管理那些对象，这就是为什么理解对象如何运作是理解整个 JavaScript 的关键。本章后面会对此进行详细讨论。

3.1 定义属性

在第 1 章讲过，有两种创建自己的对象的方式：使用 Object 构造函数或使用对象的字面形式。例如，

```
var person1 = {
    name: "Nicholas"
};

var person2 = new Object();
person2.name = "Nicholas";

person1.age = "Redacted";
person2.age = "Redacted";

person1.name = "Greg";
person2.name = "Michael";
```

person1 和 person2 都是具有 name 属性的对象。然后两个对象都被赋予了 age 属性。你可以在对象定义之后立即给它添加属性，也可以等到后面。你创建的对象总是任你随意修改，除非你另外指明（见本章 3.7）。例子最后改变了两个对象 name 的值，属性的值也可以被随时修改。

当一个属性第一次被添加给对象时，JavaScript 在对象上调用一个名为[[Put]]的内部方法。[[Put]]方法会在对象上创建一个新节点来保存属性，就像第一次在哈希表上添加一个键一样。这个操作不仅指定了初始的值，也定义了属性的一些特征。所以在前例中，当属性 name 和 age 在每个对象上第一次被定义时，[[Put]]方法都在该对象上被调用了。

调用[[Put]]的结果是在对象上创建了一个自有属性。一个自有属性表明仅仅该指定的对象实例拥有该属性。该属性被直接保存在实例内，对该属性的所有操作都必须通过该对象进行。

注意 自有属性有别于原型属性，我们将在第 4 章讨论原型属性。

当一个已有的属性被赋予一个新值时，调用的是一个名为[[Set]]的方法。该方法将属性的当前值替换为新值。上例为 name 设置第

二个值时，调用了[[Set]]方法。图 3-1 分步显示了 person1 的 name 和 age 属性被改变时内部的变化。

图的第一部分显示了用对象字面形式创建的对象 person1，它在属性 name 上隐式调用了[[Put]]。对 person1.age 的赋值是在属性 age 上调用了[[Put]]。而给 person1.name 设置新值“Greg”则是在属性 name 上调用了[[Set]]，覆盖了已有的属性值。

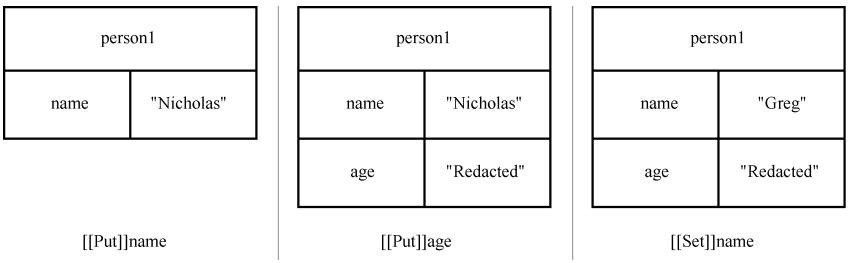


图 3-1 添加和改变一个对象的属性

3.2 属性探测

由于属性可以在任何时候添加，所以有时候就有必要检查对象是否已有一个属性。JavaScript 开发新手常错误地使用以下模式探测属性是否存在。

```
// unreliable
if(person1.age) {
   // do something with age
}
```

问题在于 JavaScript 的类型强制会影响该模式的输出结果。当 if 判断中的值是一个对象、非空字符串、非零数字或 true 时，判断会评估为真；而当值是一个 null、undefined、0、false、NaN 或空字符串时评估为假。由于一个对象属性可以包含这些假值，上例代码有可能导致错误的假定。例如，当 person1.age 为 0 时，虽然 age 属性存在，if 条件仍然无法满足。更加可靠的判断属性是否存在的方法是使用 in 操作符。

in 操作符在给定对象中查找一个给定名称的属性，如果找到则返回 true。实际上 in 操作符就是在哈希表中查找一个键是否存在。下例展示了用 in 检查 person1 对象的某些属性时会发生的情况。

```
console.log("name" in person1);     // true
console.log("age" in person1); // true
console.log("title" in person1);    // false
```

方法就是值为函数的属性，所以你一样可以检查一个方法是否存在。下例给 person1 添加了 sayName()方法并用 in 来确认方法的存在。

```
var person1 = {
   name: "Nicholas",
   sayName: function() {
      console.log(this.name);
   }
};

console.log("sayName" in person1); // true
```

在大多数情况下，in 操作符是探测对象中属性是否存在的最好的途径。它还有一个额外的好处就是不会评估属性的值。当此类评估会导致性能问题或错误时，这一点就尤其重要。

然而在某些情况下，你可能希望仅当一个属性是自有属性时才检查其是否存在。in 操作符会检查自有属性和原型属性，所以你不得不选择另一条途径：所有的对象都拥有的 hasOwnProperty()方法。该方法在给定的属性存在且为自有属性时返回 true。下例代码比较了 in 和 hasOwnProperty()针对 person1 的不同属性的结果。

```
var person1 = {
   name: "Nicholas",
   sayName: function() {
      console.log(this.name);
   }
};

console.log("name" in person1);                         // true
console.log(person1.hasOwnProperty("name"));            // true

console.log("toString" in person1);                     // true
console.log(person1.hasOwnProperty("toString"));        // false
```

在本例中，name 是 person1 的一个自有属性，所以 in 操作符和

hasOwnProperty()都返回 true。而 toString()方法则是一个所有对象都具有的原型属性。in 操作符对 toString 返回 true，而 hasOwnProperty()则返回 false。这是个重大区别，在第 4 章会深入讨论。

3.3 删除属性

正如属性可以在任何时候被添加到对象上，它们也可以在任何时候被移除。但设置一个属性的值为 null 并不能从对象中彻底移除那个属性，只是调用[[Set]]将 null 值替换了该属性原来的值而已，这点你已经在本章前面见过了。你需要使用 delete 操作符来彻底移除对象的一个属性。

delete 操作符针对单个对象属性调用名为[[Delete]]的内部方法。你可以认为该操作在哈希表中移除了一个键值对。当 delete 操作符成功时，它返回 true。（某些属性无法被移除，本章后续将详细讨论。）下例展示了 delete 操作符的用法。

```
var person1 = {
    name: "Nicholas"
};

console.log("name" in person1);          // true

delete person1.name;                  // true - not output
console.log("name" in person1);          // false
console.log(person1.name);               // undefined
```

本例中，name 属性被从 person1 中删除。操作完成后，in 操作符返回 false。试图访问一个不存在的属性将返回 undefined。图 3-2 展示了 delete 是如何影响一个对象的。

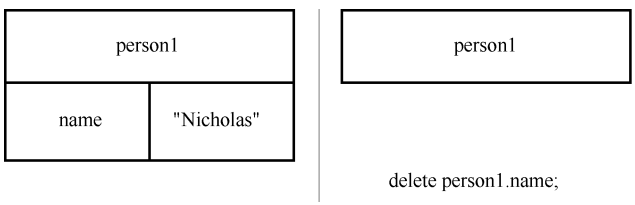


图 3-2 当你用 delete 删除 name 属性时，它将彻底从 person1 中消失

3.4 属性枚举

所有你添加的属性默认都是可枚举的，也就是说你可以用 for-in 循环遍历它们。可枚举属性的内部特征[[Enumerable]]都被设置为 true。for-in 循环会枚举一个对象所有的可枚举属性并将属性名赋给一个变量。下例的循环将输出 object 的属性的名字和值。

```
var property;

for(property in object) {
   console.log("Name: " + property);
   console.log("Value: " + object[property]);
}
```

for-in 循环每次迭代，object 的下一个可枚举属性的名字就被赋给变量 property，直到遍历完所有的可枚举属性。然后循环结束，代码继续执行。本例使用了中括号访问对象属性的值并输出到控制台。这是 JavaScript 的中括号的一个基本用法。

如果你只需要获取一个对象的属性列表以备程序将来使用，ECMAScript 5 引入了 Object.keys()方法，它可以获取可枚举属性的名字的数组，如下所示：

```
var properties = Object.keys(object);

// if you want to mimic for-in behavior
var i, len;

for(i=0, len=properties.length; i<len; i++){
   console.log("Name: " + properties[i]);
   console.log("Value: " + object[properties[i]]);
}
```

本例使用了 Object.keys()获取了某一对象的可枚举属性。然后用 for 循环遍历属性并输出它们的名字和值。通常操作一个属性名数组时会选用 Object.keys()，而当你不需要数组时则会选用 for-in。

注意 for-in 循环返回的和 Object.keys() 返回的可枚举属性有一个区别。for-in 循环同时也会遍历原型属性而 Object.keys()只返回自有（实例）属性。第 4 章会讨论原型属性和自有属性的区别。

并不是所有的属性都是可枚举的。实际上，对象的大部分原生方法的[[Enumerable]]特征都被置为false。你可以用propertyIsEnumerable()方法检查一个属性是否为可枚举的。每个对象都拥有该方法。

```
var person1 = {
    name: "Nicholas"
};

console.log("name" in person1);                          // true
console.log(person1.propertyIsEnumerable("name"));       // true

var properties = Object.keys(person1);

console.log("length" in properties);                        // true
console.log(properties.propertyIsEnumerable("length")); // false
```

这里，属性name是可枚举的，因为它是person1上的自定义属性。而properties数组的属性length则是不可枚举的，因为它是Array.prototype的内建属性。你会发现很多原生属性默认都是不可枚举的。

3.5 属性类型

属性有两种类型：数据属性和访问器属性。数据属性包含一个值，例如本章前面几例的name属性。[[Put]]方法的默认行为是创建数据属性，本章到目前为止用到的所有例子使用的都是数据属性。访问器属性不包含值而是定义了一个当属性被读取时调用的函数（称为getter）和一个当属性被写入时调用的函数（称为setter）。访问器属性仅需要getter或setter两者中的任意一个，当然也可以两者都有。

在对象字面形式中定义访问器属性有特殊的语法。

```
var person1 = {
   _name: "Nicholas",

get name() {
   console.log("Reading name");
   return this._name;
},

set name(value) {
   console.log("Setting name to %s", value);
   this._name= value;
   }
```

```
};

console.log(person1.name);       // "Reading name" then "Nicholas"

person1.name = "Greg";
console.log(person1.name);       // "Setting name to Greg" then "Greg"
```

本例定义了一个访问器属性 name。一个数据属性_name 保存了访问器属性的实际值。（前置下划线是一个约定俗成的命名规范，表示该属性被认为是私有的，实际上它还是公开的。）用于定义 name 的 getter 和 setter 的语法看上去很像函数但没有 function 关键字。特殊关键字 get 和 set 被用在访问器属性名字的前面，后面跟着小括号和函数体。getter 被期望返回一个值，而 setter 则接受一个需要被赋给属性的值作为参数。

虽然本例使用_name 来保存属性的数据，你也可以同样方便地将数据保存在变量甚至是另一个对象中。本例仅仅给属性的行为添加了日志记录。如果你只是需要保存数据，通常没有什么理由使用访问器属性——直接使用属性本身即可。但当你希望赋值操作会触发一些行为或读取的值需要通过计算所需的返回值得到时，访问器属性会非常有用。

注意 你并不一定要同时定义 getter 和 setter，可以选择定义其中之一。如果你仅定义 getter，该属性就变成只读，在非严格模式下试图写入将失败，而在严格模式下将抛出错误。如果你仅定义 setter，该属性就变成只写，在两种模式下试图读取都将失败。

3.6 属性特征

在 ECMAScript 5 之前没有办法指定一个属性是否可枚举。实际上根本没有办法访问属性的任何内部特征。为了改变这点，ECMAScript 5 引入了多种方法来和属性特征直接互动，同时也引入新的特征来支持额外的功能。现在可以创建出和 JavaScript 内建属性一样的自定义属性。本节将详细介绍数据和访问器属性的特征，下面就从它们的通用特征开始。

3.6.1 通用特征

有两个属性特征是数据和访问器属性都具有的。一个是[[Enumerable]]，决定了你是否可以遍历该属性。另一个是[[Configurable]]，决定了该属性是否可配置。你可以用 delete 删除一个可配置的属性，或随时改变它。（这也意味着可配置属性的类型可以从数据属性变成访问器属性或相反）你声明的所有属性默认都是可枚举、可配置的。

如果你想改变属性特征，可以使用 Object.defineProperty()方法。该方法接受 3 个参数：拥有该属性的对象、属性名和包含需要设置的特征的属性描述对象。属性描述对象具有和内部特征同名的属性但名字中不包含中括号。所以你可以用 enumerable 属性来设置[[Enumerable]]特征，用 configurable 属性来设置[[Configurable]]特征。例如，假设你想要让一个对象属性变成不可枚举且不可配置，方法如下。

```
var person1 = {
   name: "Nicholas"
};

Object.defineProperty(person1, "name", {
   enumerable: false
});

console.log("name" in person1);                          // true
console.log(person1.propertyIsEnumerable("name"));       // false

var properties = Object.keys(person1);
console.log(properties.length);                          // 0

Object.defineProperty(person1, "name", {
   configurable: false
});

// try to delete the Property
delete person1.name;
console.log("name" in person1);                          // true
console.log(person1.name);                               // "Nicholas"

Object.defineProperty(person1, "name", {                 // error!!!
   configurable: true
});
```

本例如通常一样定义了 name 属性，然后设置它的[[Enumerable]]特征为 false。基于这个新值的 propertyIsEnumerable()方法将返回

false。之后 name 被改为不可配置。从现在起，由于该属性不能被改变，试图删除 name 将会失败，所以 name 依然存在于 person1 中。对 name 再次调用 Object.defineProperty()也不会改变属性。person1 对象的属性 name 被有效地锁定。

最后几行代码试图重新定义 name 为可配置的。然而这将抛出错误。你无法将一个不可配置的属性变成可配置。同样，在不可配置的情况下试图将数据属性变为访问器属性或反向变更也会抛出错误。

注意 当 JavaScript 运行于严格模式，试图删除一个不可配置的属性将导致错误。而非严格模式则会失败。

3.6.2 数据属性特征

数据属性额外拥有两个访问器属性不具备的特征。第一个是[[Value]]，包含属性的值。当你在对象上创建属性时该特征被自动赋值。所有的属性的值都保存在[[Value]]中，哪怕该值是一个函数。

第二个特征是[[Writable]]，该特征是一个布尔值，指示该属性是否可以写入。所有的属性默认都是可写的，除非你另外指定。

通过这两个额外特征，你可以使用 Object.defineProperty()完整定义一个数据属性，即使该属性还不存在。考虑下例代码。

```
var person1 = {
   name: "Nicholas"
};
```

你已经在本章中多次看到这段代码。它给 person1 添加 name 属性并设置它的值。你可以用下面（更加罗嗦）的代码达到同样的效果。

```
var person1 = {};

Object.defineProperty(person1, "name", {
   value: "Nicholas",
   enumerable: true,
   configurable: true,
   writable: true
});
```

当 Object.defineProperty()被调用时，它首先检查属性是否存在。如果不存在，将根据属性描述对象指定的特征创建。在本例中，name 不是 person1 已有的属性，因此它被成功创建。

当你用 Object.defineProperty()定义新的属性时一定记得为所有的特征指定一个值，否则布尔型的特征会被默认设置为 false。下列代码创建的 name 属性就是不可枚举、不可配置、不可写的，这是因为在调用 Object.defineProperty()时没有显式指定这些特征为 true。

```
var person1 = {};

Object.defineProperty(person1, "name", {
    value: "Nicholas"
});

console.log("name" in person1);                          // true
console.log(person1.propertyIsEnumerable("name"));       // false

delete person1.name;
console.log("name" in person1);                          // true

person1.name = "Greg";
console.log(person1.name);                               // "Nicholas"
```

在这段代码中，除了读取 name 属性的值，无法对它做任何事情，因为其他操作都被锁定了。当你用 Object.defineProperty()改变一个已有的属性时，只有你指定的特征会被改变。

注意 在严格模式下试图改变不可写属性会抛出错误，而在非严格模式下则会失败。

3.6.3 访问器属性特征

访问器属性也有两个额外特征。访问器属性不需要存储值，因此也就没有[[Value]]和[[Writable]]。取而代之的是[[Get]]和[[Set]]，内含 getter 和 setter 函数。和对象字面形式的 getter 和 setter 一样，仅需要定义其中一个特征就可以创建一个访问器属性。

注意 如果你试图创建一个同时具有数据特征和访问器特征的属性，将会得到一个错误。

使用访问器属性特征比使用对象字面形式定义访问器属性的优势在于，你可以为已有的对象定义这些属性。如果你想要用对象字面形式，你只能在创建对象时定义访问器属性。

和数据属性一样，可以指定访问器属性是否为可配置、可枚举。看看以前的例子，如下。

```
var person1 = {
    _name: "Nicholas",

    get name() {
        console.log("Reading name");
        return this._name;
    },

    set name(value) {
        console.log("Setting name to %s", value);
        this._name= value;
    }
};
```

这段代码可以被改写成如下形式。

```
var person1 = {
    _name: "Nicholas"
};

Object.defineProperty(person1, "name", {
    get: function() {
        console.log("Reading name");
        return this._name;
    },
    set: function(value) {
         console.log("Setting name to %s", value);
         this._name= value;
    },
    enumerable: true,
    configurable: true
});
```

注意那个传给 Object.defineProperty()的属性描述对象中的 get 和 set 关键字，它们是包含函数的数据属性。在这里不能使用对象字面形式。

设置[[Enumerable]]和[[Configurable]]可以改变访问器属性的工作方式。例如，你可以像下面的代码一样来创建一个不可配置、不可枚举、不可写的属性。

```
var person1 = {
    _name: "Nicholas"
};
```

```
Object.defineProperty(person1, "name", {
    get: function() {
        console.log("Reading name");
        return this._name;
    }
});

console.log("name" in person1);                          // true
console.log(person1.propertyIsEnumerable("name"));       // false
delete person1.name;
console.log("name" in person1);                          // true

person1.name = "Greg";
console.log(person1.name);                               // "Nicholas"
```

在这段代码中，name 属性是一个只有 getter 的访问器属性。没有 setter，也没有任何特征被显式指定为 true，所以它的值只能被读取，不能被改变。

注意 和对象字面形式定义的访问器属性一样，在严格模式下试图写入没有 setter 的访问器属性会抛出错误，而在非严格模式下会失败。试图读取一个没有 getter 的访问器属性则总是返回 undefined。

3.6.4 定义多重属性

如果你使用 Object.defineProperties()而不是 Object.defineProperty()，可以为一个对象同时定义多个属性。这个方法接受两个参数：需要改变的对象和一个包含所有属性信息的对象。后者可以被看成一个哈希表，键是属性名，值是为该属性定义特征的属性描述对象。如下代码定义了两个属性。

```
var person1 = {};

Object.defineProperties(person1, {

    // data property to store data
    _name: {
        value: "Nicholas",
        enumerable: true,
        configurable: true,
        writable: true
    },

    // accessor property
```

```
    name: {
        get: function() {
            console.log("Reading name");
            return this._name;
        },
        set: function(value) {
            console.log("Setting name to %s", value);
            this._name= value;
        },
        enumerable: true,
        configurable: true
    }
});
```

本例定义了_name 数据属性和 name 访问器属性。你可以用 Object.defineProperties()定义任意数量的属性。甚至可以同时改变已有的属性并创建新属性。

3.6.5 获取属性特征

如果需要获取属性的特征，在 JavaScript 中可以用 Object.getOwnPropertyDescriptor()方法。正如其名，这个方法只可用于自有属性。它接受两个参数：对象和属性名。如果属性存在，它会返回一个属性描述对象，内含 4 个属性：configurable 和 enumerable，另外两个属性则根据属性类型决定。即使你从没有为属性显式指定特征，你依然会得到包含全部这些特征值的属性描述对象。下面的代码创建了一个属性并检查其特征。

```
var person1 = {
    name: "Nicholas"
};

var descriptor = Object.getOwnPropertyDescriptor(person1, "name");

console.log(descriptor.enumerable);          // true
console.log(descriptor.configurable);        // true
console.log(descriptor.writable);       // true
console.log(descriptor.value);          // "Nicholas"
```

这里，属性 name 作为对象字面形式的一部分被定义。调用 Object.getOwnPropertyDescriptor()方法返回的属性描述对象具有 4 个属性：enumerable、configurable、writable 和 value，即使它们从没有被 Object.defineProperty()显式定义。

3.7 禁止修改对象

对象和属性一样具有指导其行为的内部特征。其中，[[Extensible]]是一个布尔值，它指明该对象本身是否可以被修改。你创建的所有对象默认都是可扩展的，意味着新的属性可以随时被添加。相信你已经在本章很多地方看到了这一点。设置[[Extensible]]为 false，你就能禁止新属性的添加。有下列 3 种方法可以帮助你锁定对象属性。

3.7.1 禁止扩展

第一种方法是用 Object.preventExtensions()创建一个不可扩展的对象。该方法接受一个参数，就是你希望使其不可扩展的对象。一旦在一个对象上使用该方法，就永远不能再给它添加新的属性了。可以用 Object.isExtensible()来检查[[Extensible]]的值。请看下面代码。

```
var person1 = {
    name: "Nicholas"
};

console.log(Object.isExtensible(person1));        // true

Object.preventExtensions(person1);
console.log(Object.isExtensible(person1));        // false

person1.sayName = function() {
    console.log(this.name);
};

console.log("sayName" in person1);                // false
```

创建 person1 后，这段代码先检查其[[Extensible]]特征，然后将其变得不可扩展。由于不可扩展，sayName()方法永远无法被加到 person1 上。

注意 在严格模式下试图给一个不可扩展对象添加属性会抛出错误，而在非严格模式下则会失败。应该对不可扩展对象使用严格模式，这样，当一个不可扩展对象被错误使用时你就会知道。

3.7.2 对象封印

对象封印是创建不可扩展对象的第二种方法。一个被封印的对象是不可扩展的且其所有属性都不可配置。这意味着不仅不能给对

象添加新属性，也不能删除属性或改变其类型（从数据属性变成访问器属性或相反）。如果一个对象被封印，只能读写它的属性。

可以用 Object.seal()方法来封印一个对象。该方法被调用时，[[Extensible]]特征被置为 false，其所有属性的[[Configurable]]特征被置为 false。如下面所示，可以用 Object.isSealed()判断一个对象是否被封印。

```
var person1 = {
    name: "Nicholas"
};

console.log(Object.isExtensible(person1));    // true
console.log(Object.isSealed(person1));        // false

Object.seal(person1);
console.log(Object.isExtensible(person1));    // false
console.log(Object.isSealed(person1));        // true

person1.sayName = function() {
    console.log(this.name);
};

console.log("sayName" in person1);       // false

person1.name = "Greg";
console.log(person1.name);                    // "Greg"

delete person1.name;
console.log("name" in person1);               // true
console.log(person1.name);                    // "Greg"

var descriptor = Object.getOwnPropertyDescriptor(person1, "name");
console.log(descriptor.configurable);         // false
```

这段代码封印了 person1，因此不能在 person1 上添加或删除属性。所有的被封印对象都是不可扩展的对象，此时对 person1 使用 Object.isExtensible()方法将会返回 false，且试图添加 sayName()会失败。而且，虽然 person1.name 被成功改变为一个新值，但是删除它将会失败。

如果你熟悉 Java 或 C++语言，你也应该熟悉被封印对象。当你基于这两种语言的类创建对象时，无法给对象添加新的属性，但可以修改该属性的值。实际上，封印对象就是 JavaScript 在没有类的情况下允许你做同样的控制。

注意 确保对被封印对象使用严格模式，这样当有人误用该对象时，你会得到一个错误。

3.7.3 对象冻结

创建不可扩展对象的最后一种方法是冻结它。如果一个对象被冻结，则不能在其上添加或删除属性，不能改变属性类型，也不能写入任何数据属性。简而言之，被冻结对象是一个数据属性都为只读的被封印对象。被冻结对象无法解冻。可以用 Object.freeze()来冻结一个对象，用 Object.isFrozen()来判断一个对象是否被冻结，代码如下。

```
var person1 = {
    name: "Nicholas"
};

console.log(Object.isExtensible(person1));        // true
console.log(Object.isSealed(person1));            // false
console.log(Object.isFrozen(person1));            // false

Object.freeze(person1);
console.log(Object.isExtensible(person1));        // false
console.log(Object.isSealed(person1));            // true
console.log(Object.isFrozen(person1));            // true

person1.sayName = function() {
    console.log(this.name);
};

console.log("sayName" in person1);           // false

person1.name = "Greg";
console.log(person1.name);                   // "Nicholas"

delete person1.name;
console.log("name" in person1);              // true
console.log(person1.name);                   // "Nicholas"

var descriptor = Object.getOwnPropertyDescriptor(person1, "name");
console.log(descriptor.configurable);        // false
console.log(descriptor.writable);       // false
```

在本例中，person1 被冻结。被冻结对象也被认为是不可扩展对象和被封印对象，所以 Object.isExtensible()返回 false，而 Object.isSealed()则返回 true。属性 name 无法被改变，所以试图对其赋值为“Greg”的操作失败，后续的检查依旧返回“Nicholas”。

注意 被冻结对象仅仅只是对象在某个时间点上的快照。其用途有限且极少被使用。和所有不可扩展对象一样，应该对被冻结对象使用严格模式。

3.8 总结

将属性视为键值对，对象视为属性的哈希表有助于理解 JavaScript 对象。你可以使用点号或中括号访问对象的属性。可以随时用赋值的方式添加新属性，也可以在任何时候用 delete 操作符删除一个属性。你可以随时用 in 操作符检查对象中某个属性是否存在。如果是自有属性，还可以用 hasOwnProperty()，这个方法存在于所有对象中。所有对象属性默认都是可枚举的，这意味着它们会出现在 for-in 循环中或者被 Object.keys()获取。

属性有两种类型：数据属性和访问器属性。数据属性保存值，你可以读写它们。当数据属性保存了一个函数的值，该属性被认为是对象的一个方法。不同于数据属性，访问器属性不保存值；它们用 getter 和 setter 来进行指定的操作。可以用对象字面形式创建数据属性和访问器属性。

所有属性都有一些相关特征。这些特征定义了属性的工作模式。数据属性和访问器属性都具有[[Enumerable]]和[[Configurable]]特征。数据属性还具有[[Writable]]和[[Value]]特征，而访问器属性则具有[[Get]]和[[Set]]特征。[[Enumerable]]和[[Configurable]]默认对所有属性置为 true，[[Writable]]默认对数据属性置为 true。你可以用 Object.defineProperty()或 Object.defineProperties()改变这些特征，用 Object.getOwnPropertyDescriptor()获取它们。

有 3 种方式可以锁定对象的属性。Object.prevent Extensions()方法创建不可扩展的对象，无法在其上添加新的属性。如果你用 Object.seal()方法创建被封印对象，它不可扩展且其属性不可配置。Object.freeze()方法创建被冻结对象，它同时是一个被封印对象且其数据属性不可写。你要当心这些不可扩展对象并始终对它们使用严格模式，这样任何对其错误的使用都会抛出一个错误。

第 4 章

构造函数和原型对象

也许你在还没有理解构造函数和原型对象的时候就已经在JavaScript之路上走了很远，但直到你很好地掌握它们之前你不会真正欣赏这门语言。由于 JavaScript 缺乏类，它用构造函数和原型对象来给对象带来与类相似的功能。但是，这些相似的功能并不一定表现的跟类完全一致。在本章中，你会详细看到 JavaScript 如何使用构造函数和原型对象来创建对象。

4.1 构造函数

构造函数就是你用 new 创建对象时调用的函数。到目前为止，你已经见过好几次内建的 JavaScript 构造函数了，例如，Object、

Array 和 Function。使用构造函数的好处在于所有用同一个构造函数创建的对象都具有同样的属性和方法。如果想创建多个相同的对象，你可以创建自己的构造函数以及引用类型。

构造函数也是函数，你会用同样的方法定义它。唯一的区别是构造函数名应该首字母大写，以此区分于其他函数。下例定义了一个空的 Person 函数。

```
function Person() {
   // intentionally empty
}
```

这个函数就是个构造函数。构造函数和其他函数并没有绝对的语法上的区别。唯一的线索是首字母大写。

定义好构造函数以后，你就可以用它来创建对象，例如下面两个 Person 对象。

```
var person1 = new Person();
var person2 = new Person();
```

如果没有需要传递给构造函数的参数，你甚至可以忽略小括号，如下。

```
var person1 = new Person;
var person2 = new Person;
```

即使 Person 构造函数并没有显式返回任何东西，person1 和 person2 都会被认为是一个新的 Person 类型的实例。new 操作符会自动创建给定类型的对象并返回它们。这也意味着，你可以用 instanceof 操作符获取对象的类型。如下所示。

```
console.log(person1 instanceofPerson); // true
console.log(person2 instanceofPerson); // true
```

由于 person1 和 person2 被 Person 构造函数创建，用 instanceof 检查他们是否为 Person 类型时将返回 true。

你也可以用构造函数属性来检查一个对象的类型。每个对象在创建时都自动拥有一个构造函数属性，其中包含了一个指向其构造函数的引用。那些通过对象字面形式或 Object 构造函数创建出来的泛用对象，其构造函数属性指向 Object；那些通过自定义构造函数

创建出来的对象，其构造函数属性指向创建它的构造函数。如下例，person1 和 person2 的 constructor 属性是 Person。

```
console.log(person1.constructor === Person); // true
console.log(person2.constructor === Person); // true
```

console.log 函数对两个对象都输出 true，因为它们都是用 Person 构造函数创建的。

虽然对象实例及其构造函数之间存在这样的关系，但是还是建议你使用 instanceof 来检查对象类型。这是因为构造函数属性可以被覆盖，并不一定完全准确。

当然，一个空的构造函数并不是十分有用的。使用构造函数的目的是为了轻松创建许多拥有相同属性和方法的对象。为此，你只需要在构造函数内简单地给 this 添加任何你想要的属性即可，如下例。

```
function Person(name) {
   this.name = name;
   this.sayName= function() {
       console.log(this.name);
   };
}
```

该版本的 Person 构造函数接受一个命名参数 name 并将其赋给 this 对象的 name 属性。同时，构造函数还给对象添加了一个 sayName()方法。当你调用构造函数时，new 会自动创建 this 对象，且其类型就是构造函数的类型。（在本例中为 Person）构造函数本身不需要返回一个值，new 操作符会帮你返回。

现在可以使用 Person 构造函数来创建具有初始 name 属性的对象了。

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name);          // "Nicholas"
console.log(person2.name);          // "Greg"

person1.sayName();             // outputs "Nicholas"
person2.sayName();             // outputs "Greg"
```

每个对象都有自己的 name 属性，所以 sayName()方法可以根据不同对象返回不同的值。

注意 你也可以在构造函数中显式调用 return。如果返回的值是一个对象，它会代替新创建的对象实例返回。如果返回的值是一个原始类型，它会被忽略，新创建的对象实例会被返回。

构造函数允许你用一致的方式初始化一个类型的实例，在使用对象前设置好所有的属性。如下例，你也可以在构造函数中用 Object.defineProperty()方法来帮助我们初始化。

```
function Person(name) {

   Object.defineProperty(this, "name", {
      get: function() {
         return name;
      },
      set: function(newName) {
         name= newName;
      },
      enumerable: true,
      configurable: true
   });

   this.sayName= function() {
      console.log(this.name);
   };
}
```

在这个版本的 Person 构造函数中，name 属性是一个访问者属性，利用 name 参数来存取实际的值。之所以能这样做，是因为命名参数就相当于一个本地变量。

始终确保用 new 调用构造函数；否则，你就是在冒改变全局对象的风险，而不是创建一个新的对象。考虑如下代码中发生了什么。

```
var person1 = Person("Nicholas");        // note: missing "new"

console.log(person1 instanceofPerson);        // false
console.log(typeof person1);             // "undefined"
console.log(name);                       // "Nicholas"
```

当 Person 不是被 new 调用时，构造函数中的 this 对象等于全局 this 对象。由于 Person 构造函数依靠 new 提供返回值，person1 变量为 undefined。没有 new，Person 只不过是一个没有返回语句的函数。对 this.name 的赋值实际上是创建了一个全局变量 name 来保存

传递给 Person 的参数。第 6 章介绍的几个对象模式中将包括对这一问题的解决方法。

注意 在严格模式下，当你不通过 new 调用 Person 构造函数时会出现错误。这是因为严格模式并没有为全局对象设置 this。this 保持为 undefined，而当你试图为 undefined 添加属性时都会出错。

构造函数允许你给对象配置同样的属性，但是构造函数并没有消除代码冗余。在之前的例子中，每一个对象都有自己的 sayName() 方法。这意味着如果你有 100 个对象实例，你就有 100 个函数做相同的事情，只是使用的数据不同。

如果所有的对象实例共享同一个方法会更有效率，该方法可以使用 this.name 访问正确的数据，这就需要用到原型对象。

4.2 原型对象

可以把原型对象看作是对象的基类。几乎所有的函数（除了一些内建函数）都有一个名为 prototype 的属性，该属性是一个原型对象用来创建新的对象实例。所有创建的对象实例共享该原型对象，且这些对象实例可以访问原型对象的属性。例如，hasOwnProperty() 方法被定义在泛用对象 Object 的原型对象中，但却可以被任何对象当作自己的属性访问，如下例。

```
var book = {
   title: "The Principles of Object-Oriented JavaScript"
};

console.log("title" in book);                                    // true
console.log(book.hasOwnProperty("title"));                       // true
console.log("hasOwnProperty" in book);                           // true
console.log(book.hasOwnProperty("hasOwnProperty"));              //
false
console.log(Object.prototype.hasOwnProperty("hasOwnProperty"));  // true
```

即使 book 中并没有 hasOwnProperty()方法的定义，仍然可以通过 book.hasOwnProperty()访问该方法。这是因为该方法的定义存在于 Object.prototype 中。in 操作符对原型属性和自有属性都返回 true。

鉴别一个原型属性

你可以用这样一个函数去鉴别一个属性是否是原型属性。

```
function hasPrototypeProperty(object, name) {
    return name in object &&!object.hasOwnProperty(name);
}

console.log(hasPrototypeProperty(book, "title"));      //
false
console.log(hasPrototypeProperty(book, "hasOwnProperty")); //
true
```

如果某个属性 in 一个对象，但 hasOwnProperty()返回 false，那么这个属性就是一个原型属性。

4.2.1 [[Prototype]]属性

一个对象实例通过内部属性[[Prototype]]跟踪其原型对象。该属性是一个指向该实例使用的原型对象的指针。当你用 new 创建一个新的对象时，构造函数的原型对象会被赋给该对象的[[Prototype]]属性。图 4-1 显示了[[Prototype]]属性是如何让多个对象实例引用同一个原型对象来减少重复代码的。

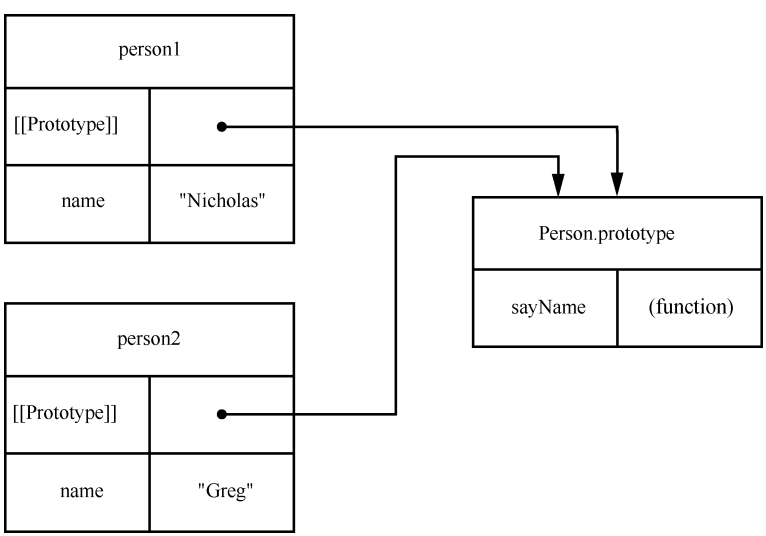


图 4-1 person1 和 person2 的[[Prototype]]属性指向同一个原型对象

你可以调用对象的 Object.getPrototypeOf()方法读取[[Prototype]]属性的值。下例代码检查一个泛用空对象的[[Prototype]]属性。

```
var object = {};
var prototype = Object.getPrototypeOf(object);

console.log(prototype === Object.prototype); // true
```

如上例，任何一个泛用对象，其[[Prototype]]属性始终指向 Object.prototype。

注意 大部分 JavaScript 引擎在所有对象上都支持一个名为_proto_的属性。该属性使你可以直接读写[[Prototype]]属性。Firefox、Safari、Chrome 和 Node.js 都支持该属性，且 ECMAScript 6 正在考虑将_proto_加入到标准中。

你也可以用 isPrototypeOf()方法检查某个对象是否是另一个对象的原型对象，该方法被包含在所有对象中。

```
var object = {};

console.log(Object.prototype.isPrototypeOf(object)); // true
```

因为 object 是一个泛用对象，它的原型是 Object.prototype，意味着本例中的 isPrototypeOf()方法应当返回 true。

当读取一个对象的属性时，JavaScript 引擎首先在该对象的自有属性中查找属性名字。如果找到则返回。如果自有属性中不包含该名字，则 JavaScript 会搜索[[Prototype]]中的对象。如果找到则返回。如果找不到，则返回 undefined。

下例首先创建了一个没有任何自有属性的对象。

```
var object = {};

console.log(object.toString());          // "[object Object]"

object.toString= function() {
    return "[object Custom]";
};

console.log(object.toString());          // "[object Custom]"

// delete own property
```

```
delete object.toString;
console.log(object.toString());          // "[object Object]"

// no effect - delete only works on own properties
delete object.toString;
console.log(object.toString());          // "[object Object]"
```

本例最初的 toString()方法来自原型对象，默认返回“[object Object]”。如果你定义一个名叫 toString()的自有属性，那么每次调用该对象 toString()方法都会调用该自有属性。自有属性会覆盖原型属性。仅当自有属性被删除时，原型属性才会再一次被使用。delete 操作符仅对自有属性起作用，你无法删除一个对象的原型属性。图 4-2 对本例做了很好的解释。

同时也揭示了一个重要概念：无法给一个对象的原型属性赋值。如你在图 4-2 中所见，对 toString 的赋值在对象上创建了一个新的自有属性，而不是改变原型属性。

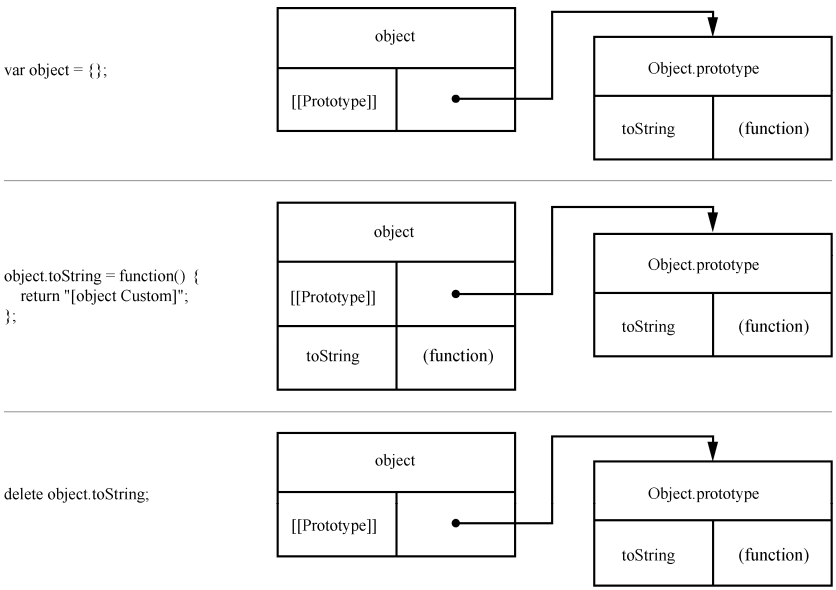


图 4-2 一个没有自有属性的对象（上）只有原型属性
为对象添加的 toString()属性（中）代替了原型属性，直到你删除它为止（下）

4.2.2 在构造函数中使用原型对象

原型对象的共享机制使得它们成为一次性为所有对象定义方法的理想手段。因为一个方法对所有的对象实例做相同的事，没理由每个实例都要有一份自己的方法。

将方法放在原型对象中并用 this 访问当前实例是更有效的做法。下例展现了新 Person 构造函数。

```
function Person(name) {
   this.name = name;
}

Person.prototype.sayName= function() {
    console.log(this.name);
};

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name);     // "Nicholas"
console.log(person2.name);     // "Greg"

person1.sayName();        // outputs "Nicholas"
person2.sayName();        // outputs "Greg"
```

在这个版本的 Person 构造函数中，sayName()被定义在原型对象上而不是构造函数中。创建出的对象和本章之前的例子中创建的对象别无二致，只不过 sayName()现在是一个原型属性而不是自有属性。在 person1 和 person2 调用 sayName()时，相对的 this 的值被分别赋上 person1 或 person2。

也可以在原型对象上存储其他类型的数据，但在存储引用值时需要注意。因为这些引用值会被多个实例共享，可能大家不希望一个实例能够改变另一个实例的值。下例显示当你不注意你的引用值实际指向哪里时会发生的情况。

```
function Person(name) {
   this.name = name;
}

Person.prototype.sayName= function() {
    console.log(this.name);
};
```

```
Person.prototype.favorites= [];

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

person1.favorites.push("pizza");
person2.favorites.push("quinoa");

console.log(person1.favorites);          // "pizza,quinoa"
console.log(person2.favorites);          // "pizza,quinoa"
```

favorites 属性被定义在原型对象上，意味着 person1.favorites 和 person2.favorites 指向同一个数组。你对任一 Person 对象的 favorites 插入的值都将成为原型对象上数组的元素。但这可能不是你期望的行为，所以在原型对象上定义时你需要非常小心。

虽然你可以在原型对象上一一添加属性，但是很多开发者会使用一种更简洁的方式：直接用一个对象字面形式替换原型对象，如下。

```
function Person(name) {
   this.name = name;
}

Person.prototype= {
    sayName: function() {
       console.log(this.name);
   },

    toString: function() {
       return "[Person " + this.name + "]";
   }
};
```

上段代码在原型对象上定义了两个方法，sayName()和 toString()。这种定义的方式非常流行，因为这种方式不需要多次键入 Person.prototype。但是有一个副作用需要注意。

```
var person1 = new Person("Nicholas");

console.log(person1 instanceof Person);      // true
console.log(person1.constructor === Person); // false
console.log(person1.constructor === Object); // true
```

使用对象字面形式改写原型对象改变了构造函数的属性，因此它现在指向 Object 而不是 Person。这是因为原型对象具有一个 constructor 属性，这是其他对象实例所没有的。当一个函数被创建时，它的 prototype 属性也被创建，且该原型对象的 constructor 属性指向该函数。当使用对象字面形式改写原型对象 Person.prototype 时，其 constructor 属性将被置为泛用对象 Object。为了避免这一点，需要在改写原型对象时手动重置其 constructor 属性，如下例。

```
function Person(name) {
   this.name = name;
}

Person.prototype= {
    constructor: Person,

    sayName: function() {
       console.log(this.name);
   },

    toString: function() {
       return "[Person " + this.name + "]";
   }
};

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1 instanceof Person);       // true
console.log(person1.constructor === Person);  // true
console.log(person1.constructor === Object);  // false

console.log(person2 instanceof Person);       // true
console.log(person2.constructor === Person);  // true
console.log(person2.constructor === Object);  // false
```

本例显式指定了原型对象的 constructor 属性。为了不忘记赋值，最好把它设为原型对象的第一个属性。

构造函数、原型对象和对象实例之间的关系最有趣的一个方面也许就是对象实例和构造函数之间没有直接联系。不过对象实例和原型对象以及原型对象和构造函数之间都有直接联系。图 4-3 描述了这个关系。

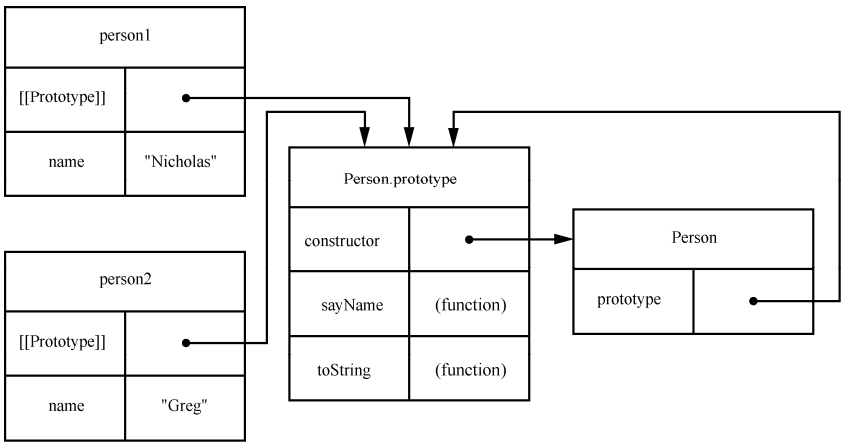


图 4-3 对象实例及其构造函数之间通过原型对象相连

这样的连接关系意味着，如果打断对象实例和原型对象之间的联系，那么也将打断对象实例和其构造函数的联系。

4.2.3 改变原型对象

给定类型的所有对象实例共享一个原型对象，所以你可以一次性扩充所有对象实例。记住，[[Prototype]]属性只是包含了一个指向原型对象的指针，任何对原型对象的改变都立即反映到所有引用它的对象实例上。这意味着你给原型对象添加的新成员都可以立即被所有已经存在的对象实例使用，如下例。

```
function Person(name) {
   this.name = name;
}

Person.prototype= {
    constructor: Person,

    sayName: function() {
       console.log(this.name);
    },

    toString: function() {
       return "[Person " + this.name + "]";
   }
};

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
```

```
console.log("sayHi" in person1);         // false
console.log("sayHi" in person2);         // false

// add a new method
Person.prototype.sayHi= function() {
    console.log("Hi");
};

person1.sayHi();                         // outputs "Hi"
person2.sayHi();                         // outputs "Hi"
```

在这段代码中，Person 类型一开始只有两个方法，sayName()和 toString()。然后在创建两个 Person 对象实例后给原型对象添加了 sayHi()。对命名属性的查找是在每次访问属性时发生的，所以可以做到无缝体验。

可以随时改变原型对象的能力在封印对象和冻结对象上有一个十分有趣的后果。当你在一个对象上使用 Object.seal()或 Object.freeze()时，完全是在操作对象的自有属性。你无法添加自有属性或改变冻结对象的自有属性，但仍然可以通过在原型对象上添加属性来扩展这些对象实例，如下例。

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

Object.freeze(person1);

Person.prototype.sayHi= function() {
    console.log("Hi");
};

person1.sayHi();                         // outputs "Hi"
person2.sayHi();                         // outputs "Hi"
```

本例中有两个 Person 的对象实例。person1 是冻结对象而 person2 是普通对象。当你在原型对象上添加 sayHi()时，person1 和 person2 都获得了这一新方法，这似乎不符合 person1 的冻结状态。其实，[[Prototype]]属性是对象实例的自有属性，属性本身被冻结，但其指向的值（原型对象）并没有冻结。

注意 实际开发 JavaScript 时，你可能不会频繁地使用原型对象。但是理解对象实例及其原型对象之间的关系是非常重要的，而像这样的奇怪例子有助于帮助你理解这些概念。

4.2.4 内建对象的原型对象

到这里，你可能会疑惑原型对象是否也允许你改变 JavaScript 引擎的标准内建对象。答案是 yes。所有内建对象都有构造函数，因此也都有原型对象给你去改变。例如，在所有数组上添加一个新的方法只需要简单地修改 Array.prototype 即可。

```
Array.prototype.sum= function() {
    return this.reduce(function(previous, current) {
       return previous + current;
   });
};

var numbers = [ 1, 2, 3, 4, 5, 6 ];
var result = numbers.sum();

console.log(result);               // 21
```

这个例子在 Array.prototype 上创建了一个名为 sum()的方法，该方法对数组的所有元素求和并返回。numbers 数组通过原型对象自动拥有了这个方法。在 sum()内部，this 指向数组的对象实例 numbers，于是该方法也可以自由使用数组的其他方法，比如 reduce()。

你可能还记得字符串、数字和布尔类型都有内建的原始封装类型来帮助我们像使用普通对象一样使用它们。如果改变原始封装类型的原型对象，你就可以给这些原始值添加更多的功能，如下例。

```
String.prototype.capitalize= function() {
    return this.charAt(0).toUpperCase() + this.substring(1);
};

var message = "hello world!";
console.log(message.capitalize()); // "Hello world!"
```

这段代码为字符串创建了一个名为 capitalize()的新方法。String 类型是字符串的原始封装类型，修改其原型对象意味着所有的字符串都自动获得这些改动。

注意 修改内建对象来试验各种功能是既有趣又好玩的事，但在生产环境中这么做可不是一个好主意。开发者们都期望一个内建对象具有一定的方法并表现出一定的行为。故意改变内建对象会破坏这种期望并导致其他开发者无法确定这些对象会如何工作。

4.3 总结

构造函数就是用 new 操作符调用的普通函数。你可以随时定义你自己的构造函数来创建多个具有同样属性的对象。可以用 instanceof 操作符或直接访问 constructor 属性来鉴别对象是被哪个构造函数创建的。

每一个函数都具有 prototype 属性，它定义了该构造函数创建的所有对象共享的属性。通常，共享的方法和原始值属性被定义在原型对象里，而其他属性都定义在构造函数里。constructor 属性实际上被定义在原型对象里供所有对象实例共享。

原型对象被保存在对象实例内部的[[Prototype]]属性中。这个属性是一个引用而不是一个副本。由于 JavaScript 查找属性的机制，你对原型对象的修改都立刻出现在所有对象实例中。当你试图访问一个对象的某个属性时，JavaScript 首先在自有属性里查找该名字，如果在自有属性中没有找到则查找原型属性。这样的机制意味着原型对象可以随时改变而引用它的对象实例则立即反映出这些改变。

内建对象也有可以被修改的原型对象。虽然不建议在生产环境中这么做，但它们可以被用来做实验以及验证新功能。

第 5 章

继　　承

学习如何创建对象是理解面向对象编程的第一步。第二步是理解继承。在传统面向对象语言中，类从其他类继承属性。然而在JavaScript 中，继承可以发生在没有类的继承关系的对象之间。这种继承的机制你已经熟悉了，就是原型对象。

5.1　原型对象链和 Object.prototype

JavaScript 内建的继承方法被称为原型对象链，又可称为原型对象继承。如你在第 4 章所学，原型对象的属性可经由对象实例访问，这就是继承的一种形式。对象实例继承了原型对象的属性。因为原型对象也是一个对象，它也有自己的原型对象并继承其属性。这就

是原型对象链：对象继承其原型对象，而原型对象继承它的原型对象，依此类推。

所有的对象，包括那些你自己定义的对象都自动继承自 Object，除非你另有指定（本章后续会讨论到）。更确切地说，所有对象都继承自 Object.prototype。任何以对象字面形式定义的对象，其[[Prototype]]的值都被设为 Object.prototype，这意味着它继承 Object.prototype 的属性，如下例中的 book。

```
var book = {
    title: "The Principles of Object-Oriented JavaScript"
};

var prototype = Object.getPrototypeOf(book);

console.log(prototype === Object.prototype); // true
```

book 的对象原型是 Object.prototype。这里不需要多余的代码来指定，因为这是创建新对象的默认行为。这个关系意味着 book 会自动接收来自 Object.prototype 的方法。

5.1.1 继承自 Object.prototype 的方法

前几章里用到的多个方法其实都是定义在 Object.prototype 上的。因此可以被其他对象继承。这些方法如下。

hasOwnProperty()	检查是否存在一个给定名字的自有属性
propertyIsEnumerable()	检查一个自有属性是否可枚举
isPrototypeOf()	检查一个对象是否是另一个对象的原型对象
valueOf()	返回一个对象的值表达
toString()	返回一个对象的字符串表达

这 5 种方法经由继承出现在所有对象中。当需要让对象在 JavaScript 中以一致的方式工作时，最后两个尤其重要，有时你甚至会想要自己定义它们。

1. valueOf()

每当一个操作符被用于一个对象时就会调用 valueOf()方法。valueOf()默认返回对象实例本身。原始封装类型重写了 valueOf()以使得它对 String 返回一个字符串，对 Boolean 返回一个布尔，对 Number 返回一个数字。类似的，Date 对象的 valueOf()方法返回一个 epoch 时间，单位是毫秒（正如 Date.prototype.getTime()所为）。这允许你写出下例代码来对 Date 做比较。

```
var now = new Date();
var earlier = new Date(2010, 1, 1);

console.log(now >earlier);          // true
```

本例中，now 是一个代表当前时间的 Date，而 earlier 是一个过去的时间。当使用大于操作符（>）时，在实际比较前，在两个对象上都调用了 valueOf()方法。你甚至可以对两个 Date 相减来获得它们在 epoch 时间上的差值。

如果你的对象也要这样使用操作符，你可以定义自己的 valueOf()方法。定义的时候你并没有改变操作符的行为，仅仅定义了操作符默认行为所使用的值。

2. toString()

一旦 valueOf()返回的是一个引用而不是原始值的时候，就会回退调用 toString()方法。另外，当 JavaScript 期望一个字符串时，也会对原始值隐式调用 toString()。例如，当加号操作符的一个边是一个字符串时，另一边会被自动转换成字符串。如果另一边是一个原始值，会自动被转换成一个字符串表达（例如，true 转换成“true”），如果另一边是一个引用值，则会调用 valueOf()。如果 valueOf()返回一个引用值，则调用 toString()。如下面代码。

```
var book = {
   title: "The Principles of Object-Oriented JavaScript"
};

var message = "Book = " + book;
console.log(message);          // "Book = [object Object]"
```

这段代码以“Book =”和 book 来构造字符串。因为 book 是一个对象，此时调用它的 toString()方法。该方法继承自 Object.prototype，大部分 JavaScript 引擎返回默认值“[object Object]”。如果你对这个值满意，就不需要改变对象的 toString()方法。但定义自己的 toString()方法有时可以为此类字符串转换提供包含更多信息的值。假设你想要之前的脚本记录书的名字，请看下例。

```
var book = {
   title: "The Principles of Object-Oriented JavaScript",
   toString: function() {
      return "[Book " + this.title + "]"
   }
};

var message = "Book = " + book;

// "Book = [Book The Principles of Object-Oriented JavaScript]"
console.log(message);
```

这段代码为 book 自定义的 toString()方法与继承来的版本相比，返回更有用的值。大多数时候，你不需要自定义 toString()方法，但必要时你该知道怎么做。

5.1.2 修改 Object.prototype

所有的对象都默认继承自 Object.prototype，所以改变 Object.prototype 会影响所有的对象，这是非常危险的。第 4 章告诫过你不要修改内建对象的原型对象，到了 Object.prototype，这个告诫就要加倍。查看下面的代码会发生什么。

```
Object.prototype.add= function(value) {
   return this + value;
};

var book = {
   title: "The Principles of Object-Oriented JavaScript"
};

console.log(book.add(5));           // "[object Object]5"
console.log("title".add("end"));        // "titleend"

// in a web browser
console.log(document.add(true));        // "[object HTMLDocument]true"
console.log(window.add(5));             // "[object Window]true"
```

添加 Object.prototype.add()会导致所有的对象都有了一个 add()方法，不管这样是不是合理。不仅仅给开发者，同时也给 JavaScript 委员会带来了问题：它不得不把新方法添加到各种不同的地方，因为给 Object.prototype 添加方法可能会带来不可预知的结果。

这个问题的另一方面在于给 Object.prototype 添加可枚举属性。在之前的例子里，Object.prototype.add()是一个可枚举属性，这意味着它会出现在 for-in 循环中，如下。

```
var empty = {};

for(var property in empty) {
   console.log(property);
}
```

这里，一个空对象依然会输出“add”作为其属性，就是因为它存在于其原型对象里且为可枚举属性。考虑到 JavaScript 中使用 for-in 的频繁程度，为 Object.prototype 添加可枚举属性会影响大量代码。因为这个原因，Douglas Crockford 推荐在 for-in 循环中始终使用 hasOwnProperty()，如下。

```
var empty = {};

for(var property in empty) {
   if (empty.hasOwnProperty(property)) {
      console.log(property);
   }
}
```

不过这个方法虽然可以有效过滤那些不想要的原型对象的属性，但也同时限制了 for-in 循环，使其只能用于自有属性，这也许不是你想要的。对你来说，最灵活的做法还是不要修改 Object.prototype。

5.2 对象继承

对象继承是最简单的继承类型。你唯一需要做的就是指定哪个对象是新对象的[[Prototype]]。对象字面形式会隐式指定 Object.prototype 为其[[Prototype]]，你也可以用 Object.create()方法显式指定。

Object.create()方法接受两个参数。第一个参数是需要被设置为新对象的[[Prototype]]的对象。第二个可选参数是一个属性描述对象，其格式如你在 Object.defineProperties()中使用的一样（见第 3 章）。考虑下面的代码。

```
var book = {
   title: "The Principles of Object-Oriented JavaScript"
};

// is the same as

var book = Object.create(Object.prototype, {
               title: {
                  configurable: true,
                  enumerable: true,
                  value: "The Principles of Object-Oriented JavaScript",
                  writable: true
               }
           });
```

两种声明具有相同的效果。第一种声明使用对象字面形式来定义一个具有单一属性 title 的对象。该对象自动继承自 Object.prototype，且其属性被默认设置为可配置、可枚举和可写。第二种声明使用 Object.create()显式做了同样的操作。两个 book 对象的行为完全一致。但你可能永远不会这样写出直接继承自 Object.prototype 的代码，毕竟那是默认行为。继承自其他对象则有趣多了，如下。

```
var person1 = {
   name: "Nicholas",
   sayName: function() {
      console.log(this.name);
   }
};

var person2 = Object.create(person1, {
   name: {
      configurable: true,
      enumerable: true,
      value: "Greg",
      writable: true
   }
});

person1.sayName();                              // outputs "Nicholas"
person2.sayName();                              // outputs "Greg"

console.log(person1.hasOwnProperty("sayName"));   // true
```

```
console.log(person1.isPrototypeOf(person2)); // true
console.log(person2.hasOwnProperty("sayName"));   // false
```

这段代码创建了一个对象 person1，具有一个 name 属性和一个 sayName()方法。对象 person2 继承自 person1，也就继承了 name 和 sayName()。然而 person2 在通过 Object.create()创建时还定义了一个自有属性 name。该自有属性隐藏并代替了原型对象的同名属性。所以，person1.sayName()输出“Nicholas”，而 person2.sayName()输出“Greg”。请记住，sayName()依然只存在于 person1 并被 person2 继承。

本例 person2 的继承链长于 person1。对象 person2 继承自 person1 而 person1 继承自 Object.prototype。如图 5-1 所示。

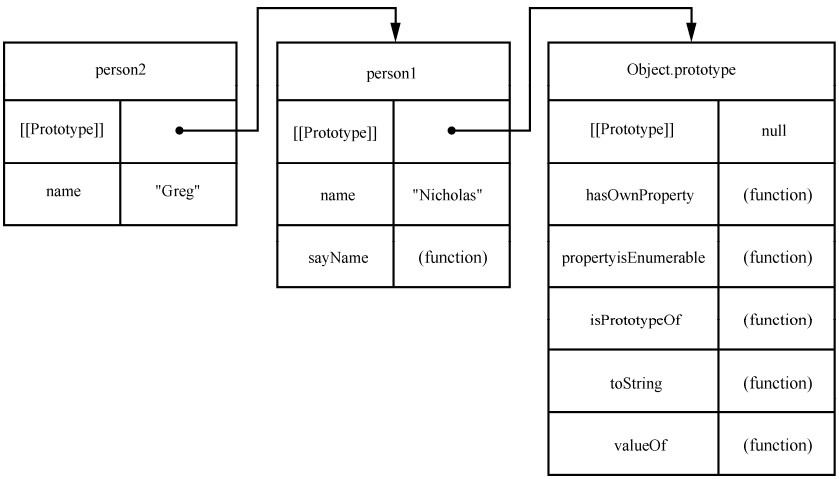


图 5-1 person2 的原型对象链包含 person1 和 Object.prototype

当访问一个对象的属性时，JavaScript 引擎会执行一个搜索过程。如果在对象实例上发现该属性（就是说是个自有属性），该属性值就会被使用。如果对象实例上没有发现该属性，则搜索[[Prototype]]。如果仍然没有发现，则继续搜索该原型对象的[[Prototype]]，直到继承链末端。末端通常是一个 Object.prototype，其[[Prototype]]被置为 null。

也可以通过 Object.create()创建[[Prototype]]为 null 的对象，如下所示。

```
var nakedObject= Object.create(null);

console.log("toString" in nakedObject);  // false
console.log("valueOf" in nakedObject);       // false
```

本例中的 nakedObject 是一个没有原型对象链的对象。这意味着 toString()和 valueOf()等内建方法都不存在于该对象上。实际上，该对象完全就是一个没有任何预定义属性的白板，这使得它成为一个完美的哈希容器，因为不会发生跟继承来的属性名字的冲突。除此之外这种对象也没有什么别的用处了，你不能把它当成一个其他继承自 Object.prototype 的对象一样使用。例如，无论何时当你对 nakedObject 使用操作符时，你都会得到一个“Cannot convert object to primitive value.”的错误。这只是一个有趣的 JavaScript 语言诡计，使你可以创建出一个没有原型对象的对象。

5.3 构造函数继承

JavaScript 中的对象继承也是构造函数继承的基础。还记得第 4 章提到，几乎所有的函数都有 prototype 属性，它可以被修改或替换。该 prototype 属性被自动设置为一个新的继承自 Object.prototype 的泛用对象，该对象有一个自有属性 constructor。实际上，JavaScript 引擎为你做了下面的事情。

```
// you write this
function YourConstructor() {
    // initialization
}

// JavaScript engine does this for you behind the scenes
YourConstructor.prototype= Object.create(Object.prototype, {
                              constructor: {
                                 configurable: true,
                                 enumerable: true,
                                 value: YourConstructor
                                 writable: true
                              }
                           });
```

你不需要做额外的工作，这段代码帮你把构造函数的 prototype 属性设置为一个继承自 Object.prototype 的对象。这意味着 Yourconstructor

创建出来的任何对象都继承自 Object.prototype。YourConstructor 是 Object 的子类，而 Object 是 YourConstructor 的父类。

由于 prototype 属性可写，你可以通过改写它来改变原型对象链。考虑下面的例子。

```
function Rectangle(length, width) {
   this.length= length;
   this.width= width;
}

Rectangle.prototype.getArea= function() {
   return this.length* this.width;
};

Rectangle.prototype.toString= function() {
   return "[Rectangle " + this.length+ "x" + this.width+ "]";
};

// inherits from Rectangle
function Square(size) {
   this.length= size;
   this.width= size;
}

Square.prototype= new Rectangle();
Square.prototype.constructor= Square;

Square.prototype.toString= function() {
   return "[Square " + this.length+ "x" + this.width+ "]";
};

var rect= new Rectangle(5, 10);
var square = new Square(6);

console.log(rect.getArea());                // 50
console.log(square.getArea());              // 36

console.log(rect.toString());               // "[Rectangle 5x10]"
console.log(square.toString());             // "[Square 6x6]"

console.log(rect instanceof Rectangle);  // true
console.log(rect instanceof Object);        // true

console.log(square instanceof Square);      // true
console.log(square instanceof Rectangle);   // true
console.log(square instanceof Object);      // true
```

这段代码里有两个构造函数：Rectangle 和 Square。Square 构造函数的 prototype 属性被改写为 Rectangle 的一个对象实例。此时不需要给 Rectangle 的调用提供参数，因为它们不需要被使用，而且如

果提供了，那么所有的 Square 的对象实例都会共享同样的维度。用这种方式改变原型对象链时，你需要确保构造函数不会在参数缺失时抛出错误（很多构造函数包含的初始化逻辑会需要参数）且构造函数不会改变任何全局状态，比如追踪有多少对象实例被创建等。Square.prototype 被改写后，其 constructor 属性会被重置为 Square。

然后，rect 作为 Rectangle 的对象实例被创建，而 square 则被作为 Square 的实例创建。两个对象都有 getArea()方法，因为那继承自 Rectangle.prototype。instanceof 操作符认为变量 square 同时是 Square、Rectangle 和 Object 的对象实例，因为 instanceof 使用原型对象链检查对象类型。如图 5-2 所示。

Square.prototype 并不真的需要被改写为一个 Rectangle 对象，毕竟 Rectangle 构造函数并没有真的为 Square 做什么必要的事情。事实上，唯一相关的部分是 Square.prototype 需要指向 Rectangle.prototype，使得继承得以实现。这意味着你可以用 Object.create()简化例子，代码如下。

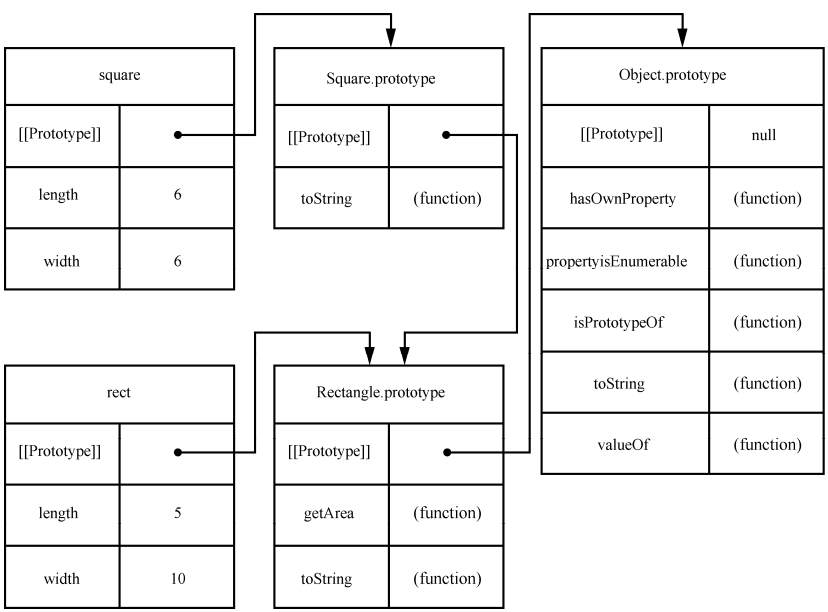


图 5-2 square 和 rect 的原型对象链显示两者都继承自 Rectangle.prototype 和 Object.prototype，但只有 square 继承自 Square.prototype

```
// inherits from Rectangle
function Square(size) {
    this.length= size;
    this.width= size;
}

Square.prototype= Object.create(Rectangle.prototype, {
                        constructor: {
                            configurable: true,
                            enumerable: true,
                            value: Square,
                            writable: true
                        }
                    });

Square.prototype.toString= function() {
    return "[Square " + this.length+ "x" + this.width+ "]";
};
```

在这个版本的代码中，Square.prototype 被改写为一个新的继承自 Rectangle.prototype 的对象，而 Rectangle 构造函数没有被调用。这意味着，你不再需要担心不加参数调用构造函数会导致的错误。除此之外，这段代码和前面的代码行为完全一致。原型对象链完好无缺，所有的 Square 对象实例都继承自 Rectangle.prototype 且其 constructor 属性也都在同样的地方被重置。

注意 *在对原型对象添加属性前要确保你已经改写了原型对象，否则在改写时会丢失之前添加的方法。*

5.4 构造函数窃取

由于 JavaScript 中的继承是通过原型对象链来实现的，因此不需要调用对象的父类的构造函数。如果你确实需要在子类构造函数中调用父类构造函数，那你就需要利用 JavaScript 函数工作的特性。

在第 2 章中学过的 call()和 apply()方法允许你在调用函数时提供一个不同的 this 值。那正好是构造函数窃取的关键。只需要在子类的构造函数中用 call()或者 apply()调用父类的构造函数，并将新创建的对象传进去即可。实际上，就是用自己的对象窃取父类的构造函数，如下例。

```
function Rectangle(length, width) {
   this.length= length;
   this.width= width;
}

Rectangle.prototype.getArea= function() {
   return this.length* this.width;
};

Rectangle.prototype.toString= function() {
   return "[Rectangle " + this.length+ "x" + this.width+ "]";
};

// inherits from Rectangle
function Square(size) {
   Rectangle.call(this, size, size);

   // optional: add new properties or override existing ones here
}

Square.prototype= Object.create(Rectangle.prototype, {
                     constructor: {
                         configurable: true,
                         enumerable: true,
                         value: Square,
                         writable: true
                      }
                  });

Square.prototype.toString= function() {
   return "[Square " + this.length+ "x" + this.width+ "]";
};

var square = new Square(6);

console.log(square.length);         // 6
console.log(square.width);          // 6
console.log(square.getArea()); // 36
```

Square 构造函数调用了 Rectangle 构造函数，并传入了 this 和 size 两次（一次作为 length，另一次作为 width）。这么做会在新对象上创建 length 和 width 属性并让它们等于 size，这是一种避免在构造函数里重新定义你希望继承的属性的手段。你可以在调用完父类的构造函数后继续添加新属性或覆盖已有的属性。

这个分两步走的过程在你需要完成自定义类型之间的继承时比较有用。你经常需要修改一个构造函数的原型对象，你也经常需要在子类的构造函数中调用父类的构造函数。一般来说，需要修改 prototype 来继承方法并用构造函数窃取来设置属性。由于这种做法模仿了那些基于类的语言的类继承，通常被称为伪类继承。

5.5 访问父类方法

在前面的例子中，Square 类型有自己的 toString()方法隐藏了其原型对象的 toString()方法。子类提供新功能覆盖父类的方法十分常见，但如果你还想访问父类的方法该怎么办呢？在其他语言中，可以用 super.toString()，但在 JavaScript 中没有类似的方式。代替的方法是在通过 call()或 apply()调用父类的原型对象的方法时传入一个子类的对象。如下例所示。

```
function Rectangle(length, width) {
   this.length= length;
   this.width= width;
}

Rectangle.prototype.getArea= function() {
   return this.length* this.width;
};

Rectangle.prototype.toString= function() {
   return "[Rectangle " + this.length+ "x" + this.height+ "]";
};

// inherits from Rectangle
function Square(size) {
   Rectangle.call(this, size, size);
}

Square.prototype= Object.create(Rectangle.prototype, {
                     constructor: {
                        configurable: true,
                        enumerable: true,
                        value: Square,
                        writable: true
                     }
                  });

// call the supertype method
Square.prototype.toString= function() {
   var text = Rectangle.prototype.toString.call(this);
   return text.replace("Rectangle", "Square");
};
```

在这个版本的代码中，Square.prototype.toString()通过 call()调用 Rectangle.prototype.toString()。该方法只需在返回文本结果前用“Square”替换“Rectangle”。这种做法看上去可能有一点冗长，但这是唯一的访问父类方法的手段。

5.6 总结

JavaScript 通过原型对象链支持继承。当将一个对象的[[Prototype]]设置为另一个对象时，就在这两个对象之间创建了一条原型对象链。所有的泛用对象都自动继承自 Object.prototype。如果你想要创建一个继承自其他对象的对象，你可以用 Object.create()指定[[Prototype]]为一个新对象。

可以在构造函数中创建原型对象链来完成自定义类型之间的继承。通过将构造函数的 prototype 属性设置为某一个对象，就建立了自定义类型对象和该对象的继承关系。构造函数的所有对象实例共享同一个原型对象，所以它们都继承自该对象。这个技术在继承其他对象的方法时工作得十分好，但你不能用原型对象继承自有属性。

为了正确继承自有属性，可以使用构造函数窃取。只需以 call()或 apply()调用父类的构造函数，就可以在子类里完成各种初始化。结合构造函数窃取和原型对象链是 JavaScript 中最常见的继承手段。由于和基于类的继承相似，这个组合经常被称为伪类继承。

可以通过直接访问父类原型对象的方式访问父类的方法。当你这么做时，你必须以 call()或 apply()执行父类方法并传入一个子类的对象。

第 6 章

对象模式

JavaScript 有很多创建对象的模式，完成工作的方式也不只一种。你可以随时定义自己的类型或自己的泛用对象。可以使用继承或混入等其他技术令对象间行为共享。也可以利用 JavaScript 高级技巧来阻止对象结构被改变。本章讨论的模式赐予你强大的管理和创建对象的能力，完全基于你自己的用例。

6.1 私有成员和特权成员

JavaScript 对象的所有属性都是公有的，且没有显式的方法指定某个属性不能被外界某个对象访问。然而，有时你可能不希望数据公有。例如，当一个对象使用一个值来决定某种状态，在对象不知情的情况下修改该值会让状态管理过程陷入混乱。一种避免它的方法是通过使用命名规则。例如，在不希望公有的属性名字前加上下划线（如 this._name）。还有很多其他方法不需要依赖命名规则，因此在阻止私有信息被修改方面也就更加“防弹”。

6.1.1 模块模式

模块模式是一种用于创建拥有私有数据的单件对象的模式。基本做法是使用立调函数表达（IIFE）来返回一个对象。IIFE 是一种被定义后立即调用并产生结果的函数表达，该函数表达可以包括任意数量的本地变量，它们在函数外不可见。因为返回的对象被定义在函数内部，对象的方法可以访问这些数据。（IIFE 定义的所有的对象都可以访问同样的本地变量）以这种方式访问私有数据的方法被称为特权方法。下面是模块模式的基本格式。

```
var yourObject= (function() {

   // private data variables

   return{
       // public methods and properties
   };

}());
```

该模式创建了一个匿名函数并立即执行。注意在函数尾部有额外的小括号，你可以用这种语法立刻执行匿名函数。这意味着这个函数仅存在于被调用的瞬间，一旦执行后立即就被销毁了。IIFE 是 JavaScript 一种非常流行的模式，部分原因就是它在模块模式中的应用。

模块模式允许你使用普通变量作为非公有对象属性。通过创建闭包函数作为对象方法来操作它们。闭包函数就是一个可以访问其作用域外部数据的普通函数。举例来说，当你在一个函数中访问一个全局对象，比如网页浏览器中的 window，该函数就是在访问其作用域外的变量。区别是，在模块模式里，变量是定义在 IIFE 中，而访问变量的函数也定义在 IIFE 中。如下。

```
var person = (function() {

   var age = 25;

   return{
      name: "Nicholas",

      getAge: function() {
         return age;
      },

      growOlder: function() {
         age++;
      }
   };

}());

console.log(person.name);      // "Nicholas"
console.log(person.getAge());  // 25

person.age= 100;
console.log(person.getAge());  // 25

person.growOlder();
console.log(person.getAge());  // 26
```

这段代码使用模块模式创建了 person 对象。变量 age 就是该对象的一个私有属性。它无法被外界直接访问，但可以通过对象方法来操作。该对象上有两个特权方法：getAge()读取变量 age 的值，growOlder()让 age 自增。这两个方法都可以直接访问变量 age，因为它们都是定义在同一个 IIFE 内。

模块模式还有一个变种叫暴露模块模式，它将所有的变量和方法都组织在 IIFE 的顶部，然后将它们设置到需要被返回的对象上。你可以用暴露模块模式改写前例，如下。

```
var person = (function() {

    var age = 25;

    function getAge() {
        return age;
    }

    function growOlder() {
        age++;
    }

    return{
        name: "Nicholas",
        getAge: getAge,
        growOlder: growOlder
    };

}());
```

在暴露模块模式中，age、getAge()和 growOlder()都被定义成 IIFE 的本地对象。然后 getAge()和 growOlder()函数被设置到返回的对象中，有效地对外界“暴露”了它们。这段代码和使用传统模块模式的前例一模一样；然而，有人更喜欢这种模式，因为它保证所有的变量和函数声明都在一处。

6.1.2 构造函数的私有成员

模块模式在定义单个对象的私有属性上十分有效，但对于那些同样需要私有属性的自定义类型又如何呢？你可以在构造函数中使用类似的模式来创建每个实例的私有数据。如下例所示。

```
function Person(name) {

    // define a variable only accessible inside of the Person constructor
    var age = 25;

    this.name = name;

    this.getAge= function() {
        return age;
    };

    this.growOlder= function() {
        age++;
    };
}
```

```
var person = new Person("Nicholas");

console.log(person.name);      // "Nicholas"
console.log(person.getAge());  // 25

person.age= 100;
console.log(person.getAge());  // 25

person.growOlder();
console.log(person.getAge());  // 26
```

在这段代码中，Person 构造函数有一个本地变量 age。该变量被用于 getAge()和 growOlder()方法。当你创建 Person 的一个实例时，该实例接受其自身的 age 变量、getAge()方法和 growOlder()方法。这种做法在很多方面都类似模块模式，构造函数创建一个本地作用域并返回 this 对象。在第 4 章讨论过，将方法直接放在对象实例上不如放在其原型对象里有效，但如果你需要实例私有的数据，这是唯一可行的手段。

如果你需要所有实例可共享的私有数据（就好像它被定义在原型对象里那样），可以结合模块模式和构造函数，如下。

```
var Person = (function() {

   // everyone shares the same age
   var age = 25;

function InnerPerson(name) {
   this.name = name;
}

InnerPerson.prototype.getAge= function() {
   return age;
};

InnerPerson.prototype.growOlder= function() {
   age++;
};

return InnerPerson;

}());

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name);          // "Nicholas"
console.log(person1.getAge()); // 25
```

```
console.log(person2.name);          // "Greg"
console.log(person2.getAge()); // 25

person1.growOlder();
console.log(person1.getAge()); // 26
console.log(person2.getAge()); // 26
```

在这段代码中，InnerPerson 构造函数被定义在一个 IIFE 中。变量 age 被定义在构造函数外并被两个原型对象的方法使用。IIFE 返回 InnerPerson 构造函数作为全局作用域里的 Person 构造函数。最终，Person 的全部实例得以共享 age 变量，所以在一个实例上的改变自动影响了另一个。

6.2 混入

JavaScript 中大量使用了伪类继承和原型对象继承，还有另一种伪继承的手段叫混入。一个对象在不改变原型对象链的情况下得到了另一个对象的属性被称为混入。第一个对象（接收者）通过直接复制第二个对象（提供者）的属性从而接收了这些属性。下面是传统的利用函数实现的混入。

```
function mixin(receiver, supplier) {
   for(var property in supplier) {
      if(supplier.hasOwnProperty(property)) {
         receiver[property] = supplier[property]
      }
   }

   return receiver;
}
```

函数 mixin()接受两个参数：接收者和提供者。该函数的目的是将提供者所有的可枚举属性复制给接收者。可以通过使用 for-in 循环迭代提供者的属性并将值设置给接收者的同名属性达成这一目的。记住这是浅拷贝，所以如果属性内包含的是一个对象，那么提供者和接收者将指向同一个对象。这个模式被广泛用于将一个 JavaScript 对象内已经存在的行为添加到另一个对象中去。

例如，可以通过混入而不是继承给一个对象添加事件支持。首先，假设你已经有一个支持事件的自定义类型。

```
function EventTarget(){
}

EventTarget.prototype= {

    constructor: EventTarget,

    addListener: function(type, listener){

        // create an array if it doesn't exist
        if(!this.hasOwnProperty("_listeners")) {
            this._listeners= [];
        }

        if(typeof this._listeners[type] == "undefined"){
            this._listeners[type] = [];
        }

        this._listeners[type].push(listener);
    },

    fire: function(event){

        if(!event.target){
            event.target= this;
        }

        if(!event.type){ // falsy
            throw new Error("Event object missing 'type' property.");
        }

        if(this._listeners&&this._listeners[event.type] instanceof Array){
            var listeners = this._listeners[event.type];
            for(var i=0, len=listeners.length; i<len; i++){
                listeners[i].call(this, event);
            }
        }
    },

    removeListener: function(type, listener){
        if(this._listeners&&this._listeners[type] instanceof Array){
        var listeners = this._listeners[type];
```

```
            for(var i=0, len=listeners.length; i<len; i++){
                if(listeners[i] === listener){
                    listeners.splice(i, 1);
                    break;
                }
            }
        }
    }
};
```

EventTarget 类型为任何对象提供基本的事件处理。你可以添加和删除监听者，也可以在对象上直接触发事件。事件监听者被存储在_listeners 属性中，该属性仅在 addListener()第一次被调用时创建（这让混入变得简单一点）。你可以像下面这样使用 EventTarget 的实例。

```
var target = new EventTarget();
target.addListener("message", function(event) {
    console.log("Message is " + event.data);
})

target.fire({
    type: "message",
    data: "Hello world!"
});
```

在 JavaScript 对象中支持事件十分有用。如果你想让另一个对象也支持事件，你有几种选择。首先，你可以创建一个新的 EventTarget 实例并添加任何你需要的属性，如下。

```
var person = new EventTarget();
person.name = "Nicholas";
person.sayName= function() {
    console.log(this.name);
    this.fire({ type: "namesaid", name: name });
};
```

在这段代码中，一个新的变量 person 作为 EventTarget 的实例被创建出来，然后添加各种跟 person 相关的属性。可惜的是，这意味着 person 实际上是一个 EventTarget 而不是一个 Object 或其他自定义类型。另外，你还需要承受手工添加一批新属性的开销。如果能有一种更加有组织的方法来干这件事就更好了。

解决这个问题的方法是使用伪类继承。

```
function Person(name) {
   this.name = name;
}

Person.prototype= Object.create(EventTarget.prototype);
Person.prototype.constructor= Person;

Person.prototype.sayName = function() {
   console.log(this.name);
   this.fire({ type: "namesaid", name: name });
};

var person = new Person("Nicholas");

console.log(person instanceof Person);       // true
console.log(person instanceof EventTarget); // true
```

在这个例子中，一个新的 Person 类型继承自 EventTarget。随后你可以在 Person 的原型对象上添加你需要的方法。然而，这还没有做到足够简洁，而且你会抱怨这个关系说不过去：一个 Person 是一种 EventTarget？通过使用混入，可以用最少的代码将这些新属性复制到原型对象中，如下。

```
Function Person(name) {
   this.name = name;
}

mixin(Person.prototype, new EventTarget());
mixin(Person.prototype, {
   constructor: Person,

   sayName: function() {
      console.log(this.name);
      this.fire({ type: "namesaid", name: name });
   }
});

var person = new Person("Nicholas");

console.log(person instanceof Person);       // true
console.log(person instanceof EventTarget); // false
```

这里，Person.prototype 混入了 EventTarget 的一个新实例来获取事件行为。然后，Person.prototype 又被混入了 constructor 和 sayName() 来完成原型对象的组装。由于本例中没有继承，Person 的实例不再是 EventTarget 的实例。

当然，有时候你可能需要使用一个对象的属性，但不想要伪类继承的构造函数。这时候，可以直接使用混入来创建自己的对象。

```
var person = mixin(new EventTarget(), {

   name: "Nicholas",

   sayName: function() {
      console.log(this.name);
      this.fire({ type: "namesaid", name: name });
   }

});
```

在这个例子中，一个 EvenTarget 实例混入了一些新的属性来创建 person 对象而没有改变 person 的原型对象链。

以这种方式使用混入时需要记住一件事，提供者的访问器属性会变成接收者的数据属性，这意味着如果你不当心，有可能改写它们。这是因为接收者的属性是被赋值语句而不是 Object.defineProperty()创建，提供者的属性当前的值被读取后赋值给接收者的同名属性。如下例。

```
var person = mixin(new EventTarget(), {

   get name() {
      return "Nicholas"
   },

   sayName: function() {
      console.log(this.name);
      this.fire({ type: "namesaid", name: name });
   }

});

console.log(person.name); // "Nicholas"

person.name = "Greg";
console.log(person.name); // "Greg"
```

这段代码定义了仅有 getter 的访问器属性 name。这意味着对该属性赋值应该不起作用。然而，由于在 person 对象里该访问器属性

变成了数据属性，你就有可能改写 name 的值。在调用 mixin()时，提供者 name 属性的值被读取后赋给接收者的 name 属性。在这个过程中没有机会定义一个新的访问器属性，从而使接收者的 name 属性成为了一个数据属性。

如果你想要访问器属性被复制成访问器属性，需要一个不同的 mixin()函数，如下。

```
function mixin(receiver, supplier) {
    Object.keys(supplier).forEach(function(property) {
        var descriptor = Object.getOwnPropertyDescriptor(supplier,
property);
        Object.defineProperty(receiver, property, descriptor);
    });

    return receiver;
}

var person = mixin(new EventTarget(), {

    get name() {
        return "Nicholas"
},

    sayName: function() {
        console.log(this.name);
        this.fire({ type: "namesaid", name: name });
    }

});

console.log(person.name);       // "Nicholas"

person.name = "Greg";
console.log(person.name);       // "Nicholas"
```

这个版本的 mixin()使用 Object.keys()获得提供者所有的可枚举自有属性。在这组属性上用 forEach()方法迭代，对提供者的每一个属性获得其属性描述符，然后通过 Object.defineProperty()添加给接收者，确保所有的属性相关信息都被传递给接收者，而不只是属性的值。这意味着 person 对象会有一个访问器属性名为 name，所以它无法被改写。

当然，这个版本的 mixin()只能工作在 ECMAScript 5 的 JavaScript 引擎上。如果你的代码需要工作在老版本的引擎上，可以将两种 mixin()结合到一个函数里。

```
function mixin(receiver, supplier) {

   if(Object.getOwnPropertyDescriptor) {

      Object.keys(supplier).forEach(function(property) {
         var descriptor = Object.getOwnPropertyDescriptor(supplier,
property);
         Object.defineProperty(receiver, property, descriptor);
      });

   } else {

      for(varproperty in supplier) {
         if(supplier.hasOwnProperty(property)) {
            receiver[property] = supplier[property]
         }
      }
   }

   return receiver;
}
```

这里，mixin()通过检查 Object.getOwnPropertyDescriptor()是否存在来决定 JavaScript 引擎是否支持 ECMAScript 5。如果支持则使用 ECMAScript 5 的版本。否则使用 ECMAScript 3 的版本。这个函数可同时被新老 JavaScript 引擎使用，因为它们会选取最合适的混入策略。

注意 Object.keys()只返回可枚举属性。如果还想要复制不可枚举属性，可以使用 Object.getOwnPropertyNames()来代替。

6.3 作用域安全的构造函数

构造函数也是函数，所以可以不用 new 操作符直接调用它们来改变 this 的值。在非严格模式下，this 被强制指向全局对象，这么做会导致无法预知的结果。而在严格模式下，构造函数会抛出一个错误。在第 4 章中，你遇到过如下的例子。

```
function Person(name) {
   this.name = name;
}
```

```
Person.prototype.sayName= function() {
   console.log(this.name);
};

var person1 = Person("Nicholas");        // note: missing "new"

console.log(person1 instanceof Person);  // false
console.log(typeof person1);             // "undefined"
console.log(name);                       // "Nicholas"
```

这个例子里，由于 Person 构造函数不是用 new 操作符调用的，我们创建了一个全局变量 name。这段代码运行于非严格模式，如果在严格模式下这么做会抛出一个错误。首字母大写的构造函数通常是在提醒你记得在前面加上 new 操作符，但是你就是想要这么用怎么办？很多内建构造函数，例如 Array 和 RegExp 不需要 new 也可以工作，这是因为它们被设计为作用域安全的构造函数。一个作用域安全的构造函数有没有 new 都可以工作，并返回同样类型的对象。

当用 new 调用一个函数时，this 指向的新创建的对象已经属于该构造函数所代表的自定义类型。也就是说，可以在函数内用 instanceof 来检查自己是否被 new 调用。

```
function Person(name) {
    if(this instanceof Person) {
       // called with "new"
    } else {
       // called without "new"
    }
}
```

使用这种模式，你可以根据 new 的使用与否来控制函数的行为。可能你想要在不同的情况下都表现出同样的行为（常常为了保护那些偶然忘记使用 new 的情况）。一个作用域安全的 Person 的版本如下。

```
function Person(name) {
    if(this instanceof Person) {
       this.name = name;
    } else {
       return new Person(name);
    }
}
```

对于这个构造函数，当自己是被 new 调用时则设置 name 属性。如果不是被 new 调用，则以 new 递归调用自己来为对象创建正确的实例。这么做，就能确保下面的行为一致了。

```
var person1 = new Person("Nicholas");
var person2 = Person("Nicholas");

console.log(person1 instanceof Person); // true
console.log(person2 instanceof Person); // true
```

这种不使用 new 创建新对象的做法已经相当常见。JavaScript 本身也提供了很多作用域安全的构造函数，例如 Object、Array、RegExp 和 Error。

6.4 总结

JavaScript 有很多不同的方式创建和组装对象。虽然 JavaScript 没有一个正式的私有属性的概念，但是你可以创建仅在对象内可访问的数据或函数。对于单件对象，你可以使用模块模式对外界隐藏数据。可以使用立调函数表达（IIFE）定义仅可被新创建的对象访问的本地变量和函数。特权方法是可以访问对象私有数据的方法。你还可以创建具有私有数据的构造函数，一种方法是在构造函数内定义变量，另一种方法是使用 IIFE 来创建所有实例共享的私有数据。

混入是一种给对象添加功能，同时避免继承的强有力的方式。混入将一个属性从一个对象复制到另一个，从而使得接收者在不需要继承提供者的情况下获得其功能。和继承不同，混入令你在创建对象后无法检查属性来源。因此，混入最适合被用于数据属性或小函数。若你想要获得更强大的功能且需要知道该功能来自哪里，继承仍然是我们推荐的做法。

作用域安全的构造函数是用不用 new 都可以被调用来生成新的对象实例的构造函数。这种模式之所以能工作，是因为 this 在构造函数一开始执行时就已经指向自定义类型的实例，你可以根据 new 的使用与否决定构造函数的行为。

索　引

Symbols

A

B

C

D

E

F

R

S

T

U

V

W

异步社区
人民邮电出版社
www.epubit.com.cn

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 

里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书 8 折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

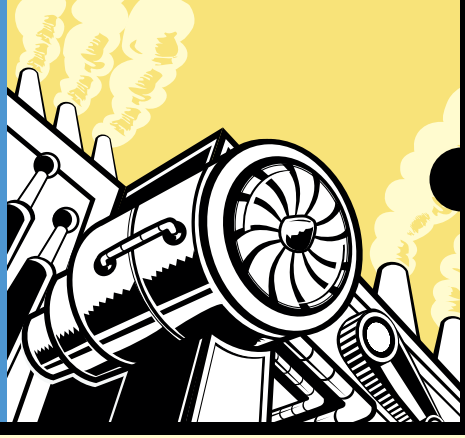
异步社区	微信服务号	微信订阅号	官方微博	QQ 群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn



你要是用过 C++ 或 Java 这样的传统面向对象语言，JavaScript 可能看上去一点都不面向对象。JavaScript 没有类的概念，而你甚至不需要定义任何对象就可以开始编写代码。但是别被这些假象蒙蔽了，实际上 JavaScript 是一种具有惊人威力且极具表达能力的面向对象的语言，它赋予你很大的设计决策权。

本书深入探讨了 JavaScript 的对象机制，揭示了这一语言对于继承以及其他关键特性的独特实现。

在本书中，你将会学到：

- 原始值和引用值的区别；
- JavaScript函数为何如此独特；
- 创建对象的各种方法；
- 如何定义你自己的构造函数；
- 如何理解和使用原型对象；
- 类型和对象的继承模式。

本书能够让有一定经验的开发者更深入地理解 JavaScript 语言，揭示 JavaScript 对象背后的秘密，从而帮助你写出更加清晰、灵活、高效的代码。

作者简介

Nicholas C. Zakas 是 Box 公司的一位软件工程师。他因关于 JavaScript 最佳实践的写作和演讲而知名。在此之前，作为 Yahoo！主页的资深前端工程师，他在雅虎工作了 5 年。他写了数本著名的 JavaScript 技术图书，包括《编写可维护的 JavaScript》和《JavaScript 高级程序设计》。

人民邮电出版社－信息技术分社
http://weibo.com/ptpitbooks

美术编辑：董志桢

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 978-7-115-38384-6