

*Learning JavaScript Design Patterns*



# JavaScript 设计模式

[美] Addy Osmani 著  
徐涛 译

O'REILLY®

人民邮电出版社  
POSTS & TELECOM PRESS

# JavaScript设计模式

通过阅读本书，你将学会如何把经典和现代设计模式应用到JavaScript语言中，来编写优美、结构化和可维护的代码。如果想让代码保持高效、更易于管理，并且能同步最新的最佳实践，那么本书正是为你打造的。

本书介绍了很多流行的设计模式，包括Module（模块）模式、Observer（观察者）模式、Facade（外观）模式和Mediator（中介者）模式；帮助你从现代Web应用程序开发者的角度来了解MVC、MVP、MVVM等现代架构模式的实用性。本书将带领你了解现代模块格式、有效为代码定义名称空间的方法和其他重要主题。

- 学习设计模式的结构以及如何编写设计模式；
- 了解不同的模式类别，包括创建型、结构型和行为模式；
- 介绍JavaScript中20多个经典和现代设计模式；
- 使用编写模块化代码的几种方法，包括Module模式、非同步模块定义（AMD）和CommonJS；
- 发掘在jQuery库中实现的设计模式；
- 学习流行的设计模式，从而编写可维护的jQuery插件。

Addy Osmani，谷歌Chrome团队的开发项目工程师，对JavaScript应用程序架构有着强烈的爱好。他创建了比较流行的项目，如TodoMVC，并对Modernizr和jQuery等其他开源项目也做出很大贡献。作为一位高产的博主（<http://addyosmani.com/blog>），Addy的文章经常出现在《JavaScript电子周刊》、《Smashing杂志》及很多其他出版物上。

“每一位JavaScript开发人员都应该阅读这本书。这是一本有关JavaScript设计模式的入门书，我们在将来会经常阅读和参考这本书。”

——Andree Hansson,  
首席前端开发人员

“Addy Osmani的这本书让很多开发人员可以了解JavaScript设计模式。MV\*和现代模块化模式部分将帮助开发者巩固他们对为了创建客户端Web应用程序可能已经在使用的技术和库的理解。”

——Eric Ferraiuolo,  
@ericf, YUI, Yahoo!



封面设计：Karen Montgomery, 张健

O'Reilly Media, Inc. 擁權人民郵電出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行  
This Authorized Edition for sale only in the territory of People's Republic of China  
(excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/程序设计/JavaScript

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)

**O'REILLY®**  
[oreilly.com.cn](http://oreilly.com.cn)

ISBN 978-7-115-31454-3



ISBN 978-7-115-31454-3

定价：49.00 元

# JavaScript 设计模式

[美] Addy Osmani 著

徐 涛 译

人 民 邮 电 出 版 社

北 京

## 图书在版编目（C I P）数据

JavaScript设计模式 / (美) 奥斯马尼 (Osmani, A.)  
著 ; 徐涛译. -- 北京 : 人民邮电出版社, 2013.6  
ISBN 978-7-115-31454-3

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

中国版本图书馆CIP数据核字(2013)第063408号

## 版权声明

Copyright© 2012 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2013. Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

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

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

版权所有，侵权必究

---

◆ 著 [美] Addy Osmani  
译 徐 涛  
责任编辑 陈冀康  
责任印制 程彦红 焦志炜  
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号  
邮编 100061 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
三河市海波印务有限公司印刷  
◆ 开本：787×1000 1/16  
印张：16  
字数：301 千字 2013 年 6 月第 1 版  
印数：1-3 000 册 2013 年 6 月河北第 1 次印刷  
著作权合同登记号 图字：01-2013-1022 号

---

定价：49.00 元

读者服务热线：(010) 67132692 印装质量热线：(010) 67129223  
反盗版热线：(010) 67171154

---

# 內容提要

设计模式是解决软件设计中常见问题的可复用方案。学习任何编程语言，设计模式都是一个令人兴奋和极具吸引力的话题。

本书是 JavaScript 设计模式的学习指南。全书分为 14 章。首先介绍了什么是模式、模式的结构、类别、模式的分类、如何编写模式等等；然后，集中介绍了很多流行的设计模式在 JavaScript 中的应用，包括 Module（模块）模式、Observer（观察者）模式、Facade（外观）模式和 Mediator（中介者）模式；最后，还探讨了模块化的 JavaScript 模式、jQuery 及其插件中的设计模式。

本书适合专业的 Web 开发人员和前端工程师阅读。通过阅读本书，他们将能够提高对设计模式的认识，并学会如何将设计模式应用到 JavaScript 编程语言中。

---

# 前言

设计模式是解决软件设计中常见问题的可复用方案。探索任何编程语言时，设计模式都是一个令人兴奋和极具吸引力的话题。

原因之一是：设计模式是许多先前开发人员总结出的经验，我们可以借鉴这些经验进行编程，以确保能够以优化的方式组织代码，为我们解决棘手的问题提供参考。

设计模式还是我们用来描述解决方案的常用词汇。当我们想要向其他人表述一种以代码形式构建解决方案的方式时，描述设计模式比描述语法和语义要简单得多。

在本书中，我们将探讨 JavaScript 编程语言中经典的与现代的设计模式的应用。

## 目标读者

本书的目标读者是专业开发人员，希望提高对设计模式的认识，并学会如何将设计模式应用到 JavaScript 编程语言中。

本书对有些概念（闭包、原型继承）只做了一些基本介绍，以便于理解。如果想要进一步了解这些概念，下面列出了一些推荐书目，方便大家阅读。

如果想要学习如何编写出美观、结构化和组织良好的代码，相信这本书就是为你而准备的。

## 致谢

本书中提到的很多设计模式是根据我的个人经验总结出来的，还有很多设计模式是 JavaScript 社区前辈总结出来的经验。本作品是众多开发人员的经验结合产物。与斯托扬·斯蒂凡诺夫为防止致谢名单中出现错漏的明智做法（在《JavaScript 模式》中）类似，我列出了致谢名单以及本书参考的文献。

如果参考文献中遗漏了任何著作或链接，我深表歉意。如果您与我联系，我一定会及时将您的著作或链接添加到参考文献中。

## 其他读物

虽然本书也面向初学者和中级开发人员，但也需要读者对 JavaScript 有基本的了解。如果想要深入了解该语言，我很乐意向您推荐以下书目。

- 《JavaScript 权威指南》，作者：David Flanagan
- 《JavaScript 编程精解》，作者：Marijn Haverbeke
- 《JavaScript 模式》，作者：Stoyan Stefanov
- 《编写可维护的 JavaScript》<sup>1</sup>，作者：Nicholas Zakas
- 《JavaScript 语言精粹》，作者：Douglas Crockford

## 本书约定

本书使用下列排版约定。

斜体 (*Italic*)

表示专业词汇、链接 (URL)、文件名和文件扩展名。

等宽字体 (Constant width)

表示广义上的计算机编码，它们包括变量或函数名、数据库、数据类型、环境变量、语句和关键字。

等宽粗体 (Constant width bold)

表示应该由用户按照字面引入的命令或其他文本。

---

<sup>1</sup> 编者注：《编写可维护的 JavaScript》已由人民邮电出版社出版 (ISBN9787115310088，定价 55 元，2013 年 3 月)。

等宽斜体 (*Constant width italic*)

表示应该由用户替换或取决于上下文的值。



这个图标表示提示、建议或一般说明。



这个图标表示警告或提醒。

## 代码示例

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

我们赞赏但不强求注明信息来源。一条信息来源通常包括标题、作者、出版者和国际标准书号 (ISBN)。例如：“Learning JavaScript Design Patterns by Addy Osmani (O'Reilly). Copyright 2012 Addy Osmani, 978-1-449-33181-8”。

如果你感到对示例代码的使用超出了正当引用或这里给出的许可范围，请随时通过 [permissions@oreilly.com](mailto:permissions@oreilly.com) 联系我们。

## Safari® 在线图书

Safari 在线图书 (Safari Books Online) 是一家按需服务的数字图书馆，提供来自领先出版商的技术类和商业类专业参考书目和视频。

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

Safari Books Online 为组织、政府机构和个人提供一系列的产品组合和定价计划。

用户可以在一个来自各个出版社的可完全搜索的数据库中访问成千上万的书籍、培训视频和正式出版前的手稿，这些出版社包括：O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、微软出版社、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等等。欲获得有关 Safari Books Online 的更多信息，请在线访问我们。

## 联系我们

关于本书的建议和疑问，可以与下面的出版社联系：

美国：

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）  
奥莱利技术咨询（北京）有限公司

我们将关于本书的勘误表，例子以及其他信息列在本书的网页上，网页地址是：

<http://www.oreilly.com/catalog/9781449302146>

如果要评论本书或者咨询关于本书的技术问题，请发邮件到：

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

想了解关于 O'Reilly 图书、课程、会议和新闻的更多信息，请访问以下网站：

<http://www.oreilly.com.cn>

<http://www.oreilly.com>

## 致谢

我要感谢热情的技术审稿人帮助我检查和改进本书，这些审稿人还包括整个社区里的同行。他们的知识和热情对这一工作做出了不可思议的贡献。这些技术审稿人的官方 tweet 和博客也经常为我提供一些想法和灵感，我竭诚向大家推荐他们的 tweet 和博客。

- Nicholas Zakas (<http://nczonline.net>, [@slicknet](http://twitter.com/slicknet)(<http://twitter.com/slicknet>))
- Andrée Hansson (<http://andreehansson.se>, [@peolanha](http://twitter.com/peolanha)(<http://twitter.com/peolanha>))
- Luke Smith (<http://lucassmith.name>, [@ls\\_n](http://twitter.com/ls_n)([http://twitter.com/ls\\_n](http://twitter.com/ls_n)))
- Eric Ferraiuolo (<http://ericf.me/>, [@ericf](http://ericf) (<http://ericf.me/>))
- Peter Michaux (<http://michaux.ca>, [@petermichaux](http://twitter.com/petermichaux))
- Alex Sexton (<http://alexsexton.com>, [@slexaxton](http://twitter.com/slexaxton)(<http://twitter.com/slexaxton>))

我还要感谢丽贝卡·墨菲 (<http://rebeccamurphey.com>, [@rmurphey](http://twitter.com/rmurphey) (<http://twitter.com/rmurphey>)) 给我写作的灵感，更重要的是，本书在 GitHub 和 O'Reilly 出版公司都可以获取。

最后，我要感谢我的妻子埃莉，她非常支持我的出书工作。

# 目录

第 1 章 介绍.....	1
第 2 章 什么是模式.....	3
我们每天都在使用模式.....	4
第 3 章 模式状态测试、Proto 模式及三法则 .....	6
第 4 章 设计模式的结构.....	8
第 5 章 编写设计模式.....	11
第 6 章 反模式.....	13
第 7 章 设计模式类别.....	15
第 8 章 设计模式分类.....	17
有关类（Class）的要点 .....	17
第 9 章 JavaScript 设计模式.....	20
9.1 Constructor（构造器）模式.....	21
9.1.1 对象创建 .....	21
9.1.2 基本 Constructor（构造器） .....	23
9.1.3 带原型的 Constructor（构造器） .....	24
9.2 Module（模块）模式.....	25
9.2.1 对象字面量 .....	25
9.2.2 Module（模块）模式 .....	27
9.2.3 Module 模式变化 .....	31
9.3 Revealing Module（揭示模块）模式 .....	36
9.3.1 优点 .....	38
9.3.2 缺点 .....	38
9.4 Singleton（单例）模式 .....	38
9.5 Observer（观察者）模式 .....	42
9.5.1 Observer（观察者）模式和 Publish/Subscribe（发布/订阅）模式的区别 .....	47
9.5.2 优点 .....	49
9.5.3 缺点 .....	49
9.5.4 Publish/Subscribe 实现 .....	49
9.6 Mediator（中介者）模式 .....	59

9.6.1	基本实现 .....	60
9.6.2	高级实现 .....	61
9.6.3	示例 .....	67
9.6.4	优点和缺点 .....	68
9.6.5	中介者 (Mediator) 与观察者 (Observer) .....	69
9.6.6	中介者 (Mediator) 与外观 (Facade) .....	69
9.7	Prototype (原型) 模式 .....	70
9.8	Command (命令) 模式 .....	73
9.9	Facade (外观) 模式 .....	75
	有关抽象的要点 .....	78
9.10	Factory (工厂) 模式 .....	78
9.10.1	何时使用 Factory 模式 .....	81
9.10.2	何时不应使用 Factory 模式 .....	81
9.10.3	Abstract Factory (抽象工厂) .....	81
9.11	Mixin 模式 .....	82
9.11.1	子类化 .....	83
9.11.2	Mixin (混入) .....	84
9.12	Decorator (装饰者) 模式 .....	88
9.12.1	伪经典 Decorator (装饰者) .....	91
9.12.2	使用 jQuery 的装饰者 .....	96
9.12.3	优点和缺点 .....	97
9.13	Flyweight (享元) 模式 .....	98
9.13.1	使用 Flyweight 模式 .....	98
9.13.2	Flyweight 和共享数据 .....	99
9.13.3	实现经典 Flyweight (享元) .....	99
9.13.4	转换代码以使用 Flyweight (享元) 模式 .....	103
9.13.5	基本工厂 .....	105
9.13.6	管理外部状态 .....	106
9.13.7	Flyweight (享元) 模式和 DOM .....	107
第 10 章	JavaScript MV*模式 .....	112
10.1	MVC .....	112
	Smalltalk-80 MVC .....	113
10.2	为 JavaScript 开发人员提供的 MVC .....	114
10.2.1	Model (模型) .....	114
10.2.2	View (视图) .....	116

10.2.3 Controller (控制器) .....	119
10.2.4 Spine.js 与 Backbone.js .....	120
10.3 MVC 为我们提供了什么 .....	122
10.4 JavaScript 中的 Smalltalk-80 MVC .....	122
10.4.1 深入挖掘 .....	123
10.4.2 总结 .....	123
10.5 MVP .....	124
10.5.1 Model、View 和 Presenter .....	124
10.5.2 MVP 或 MVC? .....	125
10.5.3 MVC、MVP 和 Backbone.js .....	126
10.6 MVVM .....	128
10.6.1 历史 .....	129
10.6.2 Model .....	129
10.6.3 View .....	130
10.6.4 ViewModel .....	133
10.6.5 小结：View 和 ViewModel .....	135
10.6.6 小结：ViewModel 和 Model .....	135
10.7 利与弊 .....	135
10.7.1 优点 .....	135
10.7.2 缺点 .....	136
10.8 使用更松散数据绑定的 MVVM .....	136
10.9 MVC、MVP 与 MVVM .....	141
10.10 Backbone.js 与 KnockoutJS .....	142
<b>第 11 章 模块化的 JavaScript 设计模式 .....</b>	<b>144</b>
11.1 脚本加载器要点 .....	145
11.2 AMD .....	145
11.2.1 模块入门 .....	146
11.2.2 使用 Dojo 的 AMD 模块 .....	150
11.2.3 AMD 模块设计模式 (Dojo) .....	151
11.2.4 使用 jQuery 的 AMD 模块 .....	152
11.2.5 AMD 总结 .....	155
11.3 CommonJS .....	155
11.3.1 入门指南 .....	156
11.3.2 使用多个依赖 .....	157
11.3.3 支持 CommonJS 的加载器和框架 .....	158

11.3.4 CommonJS 适用于浏览器吗? .....	158
11.3.5 延伸阅读.....	159
11.4 AMD 和 CommonJS: 互相竞争, 标准同效 .....	159
UMD: 用于插件的 AMD 和 CommonJS 兼容模块.....	160
11.5 ES Harmony.....	165
11.5.1 具有 Imports 和 Exports 的模块 .....	166
11.5.2 从远程数据源加载的模块 .....	167
11.5.3 模块加载器 API.....	167
11.5.4 用于服务器的类 CommonJS 模块.....	168
11.5.5 具有构造函数、getter 和 setter 的类 .....	168
11.5.6 ES Harmony 总结.....	169
11.5.7 延伸阅读.....	170
11.6 总结.....	170
<b>第 12 章 jQuery 中的设计模式 .....</b>	<b>171</b>
12.1 Composite (组合) 模式.....	171
12.2 Adapter (适配器) 模式 .....	173
12.3 Facade (外观) 模式 .....	174
12.4 Observer (观察者) 模式.....	177
12.5 Iterator (迭代器) 模式.....	180
12.6 延迟初始化 .....	181
12.7 Proxy (代理) 模式.....	183
12.8 Builder (生成器) 模式 .....	184
<b>第 13 章 jQuery 插件设计模式 .....</b>	<b>187</b>
13.1 模式.....	188
13.2 Lightweight Start 模式 .....	189
延伸阅读 .....	191
13.3 完整的 Widget Factory 模式 .....	191
延伸阅读 .....	194
13.4 嵌套命名空间插件模式 .....	194
延伸阅读 .....	196
13.5 自定义事件插件模式 (使用 Widget Factory) .....	196
延伸阅读 .....	198
13.6 使用 DOM-to-Object Bridge 模式的原型继承.....	198
延伸阅读 .....	200
13.7 jQuery UI Widget Factory Bridge 模式 .....	200

延伸阅读 .....	203
13.8 使用 Widget Factory 的 jQuery Mobile Widget .....	203
13.9 RequireJS 和 jQuery UI Widget Factory .....	206
13.9.1 用法 .....	208
13.9.2 延伸阅读 .....	209
13.10 全局选项和单次调用可重写选项（最佳选项模式） .....	209
延伸阅读 .....	211
13.11 高可配和高可变的插件模式 .....	211
延伸阅读 .....	213
13.12 是什么使插件超越模式 .....	213
13.12.1 质量 .....	214
13.12.2 代码风格 .....	214
13.12.3 兼容性 .....	214
13.12.4 可靠性 .....	214
13.12.5 性能 .....	214
13.12.6 文档 .....	215
13.12.7 维护的可能性 .....	215
13.13 总结 .....	215
13.14 命名空间模式 .....	215
13.15 命名空间基础 .....	216
13.15.1 单一全局变量 .....	216
13.15.2 命名空间前缀 .....	217
13.15.3 对象字面量表示法 .....	217
13.15.4 嵌套命名空间 .....	221
13.15.5 立即调用的函数表达式（IIFE） .....	222
13.15.6 命名空间注入 .....	224
13.16 高级命名空间模式 .....	226
13.16.1 自动嵌套的命名空间 .....	227
13.16.2 依赖声明模式 .....	229
13.16.3 深度对象扩展 .....	229
13.16.4 推荐 .....	232
第 14 章 总结 .....	233
附录 参考文献 .....	235

# 第1章

---

## 介绍

编写易于维护的代码，其中一个最重要方面是能够找到代码中重复出现的主题并优化它们。这也是设计模式有价值的地方。

在本书第1章，我们将探讨可应用于任何编程语言的设计模式发展史和重要性。如果大家已经看过或熟悉这段发展史，可以直接跳到第2章继续阅读。

设计模式来源于土木工程师克里斯托弗·亚历山大 ([http://en.wikipedia.org/wiki/Christopher\\_Alexander](http://en.wikipedia.org/wiki/Christopher_Alexander)) 的早期作品。他经常发表一些作品，内容是总结他在解决设计问题方面的经验，以及这些知识与城市和建筑模式之间有何关联。有一天，亚历山大突然发现，重复使用这些模式可以让某些设计构造取得我们期望的最佳效果。

亚历山大与萨拉-石川佳纯和穆雷·西尔弗斯坦合作创造了一种建筑模式语言，帮助设计师提高自己的设计能力，以解决任何规模的设计和建筑挑战。这就是亚历山大在1977年发表的一篇题为《建筑模式语言》的文章，其后又制作成一本完整的精装书出版 ([http://www.amazon.co.uk/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199/ref=sr\\_1\\_1?s=books&ie=UTF8&qid=1329440685&sr=1-1](http://www.amazon.co.uk/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199/ref=sr_1_1?s=books&ie=UTF8&qid=1329440685&sr=1-1))。

大约在30年前，软件工程师们开始把亚历山大编写的建筑设计原则纳入首个有关设计模式的文档中，成为初级开发人员改进编程技巧的指南。需要指出的是，设计模式背后的概念实际上自编程行业诞生以来就已经存在了，虽然是以一种不太正式的形式存在。

关于软件工程设计模式的最早和最具代表性的作品是 1995 年出版的《设计模式：可复用面向对象软件的基础》一书。该书的作者是 Erich Gamma ([http://en.wikipedia.org/wiki/Erich\\_Gamma](http://en.wikipedia.org/wiki/Erich_Gamma))、Richard Helm ([http://www.amazon.co.uk/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199/ref=sr\\_1\\_1?s=books&ie=UTF8&qid=1329440685&sr=1-1](http://www.amazon.co.uk/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199/ref=sr_1_1?s=books&ie=UTF8&qid=1329440685&sr=1-1))、Ralph Johnson ([http://en.wikipedia.org/wiki/Ralph\\_Johnson](http://en.wikipedia.org/wiki/Ralph_Johnson)) 和 John Vlissides (<http://en.wikipedia.org/wiki/John-Vlissides>)，他们以“四人组”著称（或简称为 GoF）。

GoF 发表的著作大大推动了设计模式概念在编程领域中的进一步发展，因为它描述了很多开发技术和误区，并列举了 23 个面向对象设计中最常用的经典设计模式。我们将在第 7 章进一步详细介绍这些设计模式。

我们将在本书中了解一些流行的 JavaScript 设计模式，并探讨为何某些设计模式比其他模式更适用于我们编写的程序。要记住，设计模式不仅适用于原生 JavaScript（即标准 JavaScript 代码），也适用于 jQuery 和 Dojo 等抽象库。在开始之前，我们先来了解一下软件设计中“模式”的确切定义。

## 第2章

---

# 什么是模式

模式是一种可复用的解决方案，可用于解决软件设计中遇到的常见问题，如在我们编写的 JavaScript 应用程序的实例中。另一种模式的方式是将解决问题的方法制作成模板，并且这些模板可应用于多种不同的情况。

那么，为什么了解和熟悉模式是很重要的？设计模式有三大好处。

**模式是已经验证的解决方案。**

它们为解决软件开发中遇到的问题提供可靠的方法，也就是使用已经验证的解决方案，这些解决方案体现了开发人员的经验及见解，他们为定义和改进这些方法提供了帮助，从而形成现在的模式。

**模式很容易被复用。**

模式通常是指一种立即可用的解决方案，可以对其进行修改以满足个人需求。该特性使得这些模式的功能非常强大。

**模式富有表达力。**

看到模式时，通常就表示有一个设置好的结构和表达解决方案的词汇，以帮助我们非常轻松地表达出所实现的大型解决方案。

模式不是一种确切的解决方案。重要的是，我们要知道模式的作用仅仅是为我们提供一个解决问题的方案。模式无法解决所有的设计问题，也无法取代优秀软件设计师的工作，但模式确实能够支持这些工作。接下来我们将了解一下模式的其他一些优点。

- **复用模式有助于防止在应用程序开发过程中小问题引发大问题。**这意味着当我们在已经验证的模式基础上编写代码时，可以在代码结构上少花点时间，从而有更多的时间专注于整体解决方案的质量。这是因为模式支持我们用更结构化和更有组织性的方式编写代码，从而避免以后因代码的整洁性问题而重构代码。
- **模式可以提供通用的解决方案，并且其记录方式不需要与某个特定问题挂钩。**这种通用的方法意味着不管现在开发的是哪种应用程序（在许多情况下是指编程语言），设计模式都可用于改进代码的结构。
- **某些模式确实能够通过避免代码复用来减少代码的总体资源占用量。**通过鼓励开发人员更密切地关注解决方案中可以即刻减少代码复用的部分，例如，减少类似处理过程的函数数量，用一个广义函数取而代之，那么就可以减小代码库的总大小。这也就是所谓的使代码更加简洁。
- **模式添加到开发人员的词汇中，会使沟通更快速。**
- **经常使用的模式可以逐步改进，因为其他开发人员使用这些模式后总结出的共同经验又贡献给了设计模式社区。**在某些情况下，这会创造出全新的设计模式，而在其他情况下，会对有关如何更好地使用特定模式的指导做出改进。这样可以确保基于模式的解决方案比临时解决方案更加强大。

## 我们每天都在使用模式

为了了解使用模式的好处，我们来研究一个非常简单的元素选择问题，我们通常使用jQuery库来解决这种问题。

试想，如果我们有一个脚本，想为页面上每个具有“foo”类（class属性）的DOM元素增加一个计数器，查询列表最简单有效的方法是什么？对，有几种不同的方法

可以解决这个问题。

- 在页面上选择所有元素并存储。接着过滤该集合并使用正则表达式（或另一种方式）来存储那些具有“foo”类的元素。
- 使用浏览器原生的 `querySelectorAll()` 等功能来选择所有具有“foo”类的元素。
- 同样地，使用原生特性 `getElementsByClassName()` 等功能来重新获得所需的集合。

那么，哪种方法最快？最快的实际上是第3种方法，它比其他方法 (<http://jsperf.com/getelementsbyclassname-vs-queryselectorall/5>) 快8至10倍。但在实际应用中，IE9以下的版本不支持第3种方法，因此必须使用第1种方法，而其他方法都行不通。

但使用jQuery的开发人员不必担心这个问题，因为通过使用Facade（外观）模式，它已经被抽象出来了。正如我们将在后面详细介绍的，该模式为若干更复杂的底层代码体提供了一套简单的抽象接口（例如`$el.css()`和`$el.animate()`）。正如我们所看到的，这意味着我们可以在“实现级”细节上花费更少的时间。

根据现有浏览器的支持范围，jQuery会在幕后选择最佳的元素选择方式，我们只需要使用抽象层即可。

我们可能都很熟悉jQuery的\$（“selector”）。用它选择页面上的HTML元素比需要手动处理的`getElementById()`、`getElementsByClassName()`、`getElementByTagName`等容易得多。

虽然我们知道`querySelectorAll()`可以解决这个问题，但让我们来比较一下使用jQuery的Facade（外观）模式接口与自己选择最佳路径这两种方式所花费的精力。根本就不用比！使用设计模式的抽象可以体现真实价值。

我们将在本书的后面章节探讨上述问题并了解更多设计模式。

## 第3章

# 模式状态测试、Proto 模式及三法则

请记住，并不是每一种算法、最佳实践或解决方案都代表一种完整的模式。这里可能会有一些遗漏的关键内容，在认真审查之前，设计模式社区通常不会认定它是一种真正的模式。即使展现在我们面前的某种内容似乎符合模式的标准，但也要经过一段时间的审查和多人的测试才能被视为真正的设计模式。

再一次回顾亚历山大所做的工作，他称设计模式应该是一种流程，也是一种“事物”。该定义比较模糊，他接着补充说，设计模式是一种能够创建“事物”的流程。这就是为什么模式通常专注于为视觉可识别结构寻址，即我们应该能够直观地描绘（或绘制）该结构，而且该结构是模式在实践中产生的。

在学习设计模式时，我们不时会碰到“原始模式（proto-pattern）”这个词：一种尚未通过“模式”测试的设计模式。原始模式是值得与社区分享的特殊解决方案，来自社区编程人员的心血，但因为产生的时间短，还没有机会对它们进行严格的审查。

另外，分享设计模式的个人可能没有时间或兴趣审查“模式”流程，可能只是发布原始模式的简单描述。这种类型的模式的简要描述或片段被称为 *patlet*。

完整记录合格模式的相关工作是相当艰巨的。回顾设计模式方面的一些最早期的作品，如果模式可以执行以下操作，就被认为是“优秀”的模式。

- **解决特殊问题。**模式不应该只是获取原则或策略，它们需要获取解决方案。这是作为一种优秀模式不可或缺的要素。

- **没有显而易见的解决方案。**我们通常会发现，解决问题的技术基本来自众所周知的基本原则。最好的设计模式通常会间接提供解决问题的方案——这被认为是解决与设计相关的最具有挑战性问题的必要方法。
- **描述业经验证的概念。**设计模式需要证明它们的作用与描述相一致，并且如果没有证明这一点，就不能认真考虑该设计。如果一种模式实际上是推测性的，那么只有那些勇敢的人才可能尝试使用它。
- **描述一种关系。**在某些情况下，看起来可能是一种模式描述了一种类型的模块。尽管一种实现看起来也是这样，但对模式的正式描述必须能够更深入地解释它与代码关系的系统结构和机制。

我们会理所当然地认为一种不符合设计准则的 proto 模式是不值得学习的，然而，这与事实相差甚远。很多 proto 模式其实是非常有用的。这里并不是说所有的 proto 模式都值得研究，但也有不少被遗忘的有用的原始模式能够在以后的项目中为我们提供帮助。牢记上述内容，发挥最佳判断力，你的选择过程就会很顺利。

成为有效模式的其中一个附加要求是，它们展示一些反复出现的现象。这通常可以限定在至少三个关键领域中，被称为三法则。使用该规则重现，则必须证明如下：

#### 适合性

如何才算是成功的模式？

#### 实用性

为什么模式被视为是成功的？

#### 适用性

一项设计是因为它有着广泛的适用性才能称得上是模式吗？如果是这样，需要解释其原因。

## 第4章

---

# 设计模式的结构

大家可能很想知道编写模式的作者是如何完成一种新模式的结构概述、实现和目的的。传统上，一种模式最初是以一种规则的形式呈现的，该规则建立下面这样的关系。

- 上下文
- 上下文里产生的元件系统
- 解决元件在上下文中自身问题的配置

基于这一点，现在让我们来看看对设计模式组件元素的总结。一种设计模式应该包括：

**模式名称**

**描述**

**上下文大纲**

模式在上下文中有效，以满足用户的需求。

**问题陈述**

问题的陈述已解决，因此我们可以理解模式的意图。

## **解决方案**

以可理解的步骤和看法解决用户问题的描述。

## **设计**

模式设计的描述，特别是与它交互的用户行为。

## **实现**

有关如何实现模式的指导

## **插图**

模式中类的可视表示（如图表）

## **示例**

最小形式的模式实现

## **辅助条件**

需要哪些其他模式来支持所描述模式的使用？

## **关系**

这种模式像哪些模式？它是否极力模仿其他模式呢？

## **已知的用法**

是广泛使用的模式？如果是这样，在哪里使用？如何使用？

## **讨论**

团队或者作者对该模式所带来巨大好处的想法。

在创建或维护解决方案时，设计模式是将企业或团队中的所有开发人员拉到同一战线上来的一种非常有效的方法。如果你或你的公司曾考虑研究自己的设计模式，请

记住，尽管在规划和设计阶段可能要花费大量的初始成本，但考虑到从投资中获得的价值，这样做也是相当值得的。但不要在研究现有新模式之前总是做深入研究，因为你可能会发现，直接使用或在现有已经验证的设计模式的基础上构建新模式比重新开发一种模式要更加有利。

## 第5章

---

# 编写设计模式

编写优秀的设计模式是一项具有挑战性的任务。模式不仅需要为最终用户提供大量的参考资料，还需要能够证明自己为何是必要的。

如果已经阅读了前面章节中的“什么是模式”，我们可能会认为当我们在各种地方看到它们的时候，这个定义本身应该能够帮助我们辨别模式。其实这是不完全正确的，对于我们正在查看的一段代码是否遵循一种固定模式或者只是偶然性地像一种模式，这点我们并不是很清楚。

当看到一个我们认为它可能正在使用某种模式的代码体时，我们应考虑写下这段代码里我们认为遵循现有特定模式的某些方面。

在模式分析的很多情况下，我们会发现我们只是在查看遵循好的原则和设计实践的代码，而这些原则和设计实践可能偶然会与模式的规则发生重叠。请记住：没有交互和明确规则的解决方案就不是模式。

如果你对编写自己的设计模式有兴趣，我推荐你向已经经历过这个过程并做得很好的人学习。要花时间从大量不同的设计模式描述中汲取信息，并吸收对自己有意义的知识。

发掘结构和语义——可以通过审查感兴趣模式的交互和上下文来实现，因此我们可以确定有助于在有用配置中将这些模式组织在一起的原则。

一旦熟悉了模式方面的大量信息，我们就可以使用现有的格式来开始编写模式，并

且看看自己能否想出新点子来改善它，或将自己的想法融入到里面。

近年来有个人就是这样做的，他是 Christian Heilmann，他采用了现有的模块（Module）模式，并对它做了一些有用的基本修改，以创建揭示模块（Revealing Module）模式（这是本书稍后将讨论的模式之一）。

如果你对创建新设计模式有兴趣，以下是我建议的几个要点。

### 模式的实用性有多少？

确保模式描述的是能够解决重复出现的问题的业经验证的解决方案，而不是未经验证的推测性解决方案。

牢记最佳实践。

作出的设计决策应该基于通过对最佳实践的理解而获得的原则。

设计模式对于用户来说应该是透明的。

设计模式对于任何类型的用户体验都应是完全透明的。它们主要是为使用它们的开发人员提供服务，而不应强制改变用户体验的行为，不使用模式，这些事情将不会发生。

要记住独创性在模式设计中不是重点。

编写模式时，我们不需要是已有解决方案的最初发现者，也不必担心我们的设计有一小部分与其他模式有重叠。如果我们的方法很强大，有广泛的适用性，那么它就有可能被认定为是一个有效的模式。

模式需要一批有说服力的示例。

好的模式描述需要伴随着一系列同样强有力的事例，以演示所编写模式的成功应用。为了展示模式的广泛应用，举出能够展示良好设计原则的示例是比较理想的。

编写模式是要在创建通用、具体及有用的设计方面找到一种细致的平衡。要努力确保所编写的模式能够覆盖最广泛的应用领域。希望本篇编写模式的简要介绍能够为大家提供一些见解，以帮助大家进一步学习本书后面的章节。

# 反模式

如果我们认为一种模式代表一种最佳实践，那么一种反模式就代表我们已经学到的教训。反模式这个术语是 1995 年由安德鲁·凯尼格在当年的 11 月 C++ 报告中创造的，是受“四人组”所著《设计模式》一书的启发。在凯尼格的报告中，他提出了反模式的两个概念。反模式是：

- 描述一种针对某个特定问题的不良解决方案，该方案会导致糟糕的情况发生；
- 描述如何摆脱前述的糟糕情况以及如何创造好的解决方案。

关于这个话题，亚历山大写到了有关实现在优秀设计结构和优秀上下文之间取得良好平衡所面临的困难：



这些要点是关于设计的过程；以及创造展示新物理顺序、组织和形式的物理对象的过程，与功能相对应。……每一个设计问题都是以在两个实体之间实现平衡为开始的，即：问题的形式和它的上下文。形式是解决问题的方案；上下文定义该问题。

虽然了解设计模式很重要，但理解反模式也同样很重要。让我们来探究其原因。创建应用程序时，一个项目的生命周期就会以此为起点；一旦完成了初始版本，就需要进行维护。最终方案的质量好坏取决于技能水平和团队投入的时间。这里的好坏是在上下文中考虑的，如果一个“完美的”设计应用于错误的上下文中，那么它就可能是一种反模式。

应用程序在进入生产环境并即将进入维护模式时会面临更大的挑战。之前没有研究过该应用程序的开发人员，在这样的系统上工作，可能会将不良设计意外地引入到项目中。如果说将不良实践创建为反模式，则能让开发人员有办法提前识别这些反模式，这样就可以避免常见错误的发生，它与下面这个方法相类似：设计模式为我们提供了识别有用途通用技术的方法。

总的来说，反模式是一种值得记录的不良设计。JavaScript 中的反模式示例如下所示。

- 在全局上下文中定义大量的变量污染全局命名空间。
- 向 `setTimeout` 或 `setInterval` 传递字符串，而不是函数，这会触发 `eval()` 的内部使用。
- 修改 `Object` 类的原型（这是一个特别不良的反模式）。
- 以内联形式使用 JavaScript，它是不可改变的。
- 在使用 `document.createElement` 等原生 DOM 方法更合适的情况下使用 `document.write`。多年以来 `document.write` 一直都是在被严重滥用，并有相当多的缺点，包括：如果在页面加载完成后执行 `document.write`，它实际上会重写我们所在的页面，而 `document.createElement` 则不会。访问此网站 (<http://jsfiddle.net/addyosmani/6T9vX/>) 可以看到它运行的实例。`document.write` 也无法与 XHTML 相适，这是选择像 `document.createElement` 这样更为 DOM 友好的方法比较有利的另一个原因。

了解反模式是成功的关键。一旦能够辨别这种反模式，我们将能够重构代码来防止它们的出现，这样我们解决方案的总体质量就能够立即得到改善。

# 设计模式类别

来自知名设计书籍的一个词汇——领域驱动术语，是这样描述的：



设计模式命名、抽象和标识是通用设计结构的主要方面，这些设计结构能被用于构造可复用的面向对象设计。设计模式确定所包含的类和实例、它们的角色、协作方式以及职责分配。

每一种设计模式都重点关注一个特定的面向对象设计问题或设计要点，描述何时使用它，在另一些约束条件下是否还能使用，以及使用的效果和利弊。由于我们最终要实现设计，设计模式还提供了示例……代码来阐明其实现。

虽然设计模式描述的是面向对象设计，但它们都基于实际的解决方案，这些方案的实现语言是主流面向对象的编程语言。

可以将设计模式划分成很多不同的类别。在这一节中，我们将回顾其中三种类别，并在详细探索具体类别之前，简要举出一部分这些类别的设计模式示例。

### 1. 创建型设计模式

创建型设计模式专注于处理对象创建机制，以适合给定情况的方式来创建对象。创建对象的基本方法可能导致项目复杂性增加，而这些模式旨在通过控制创建过程来解决这种问题。

属于这个类别的模式包括：Constructor（构造器）、Factory（工厂）、Abstract（抽象）、Prototype（原型）、Singleton（单例）和Builder（生成器）。

## 2. 结构型设计模式

结构型模式与对象组合有关，通常可以用于找出在不同对象之间建立关系的简单方法。这种模式有助于确保在系统某一部分发生变化时，系统的整个结构不需要同时改变。同时对于不适合因某一特定目的而改变的系统部分，这种模式也能够帮助它们完成重组。

属于这个类别的模式包括：Decorator（装饰者）、Facade（外观）、Flyweight（享元）、Adapter（适配器）和Proxy（代理）。

## 3. 行为设计模式

行为模式专注于改善或简化系统中不同对象之间的通信。

行为模式包括：Iterator（迭代器）、Mediator（中介者）、Observer（观察者）和Visitor（访问者）。

# 设计模式分类

根据我早期学习设计模式的经历，我个人发现表 8-1 中的内容能够非常有效地提醒我们这些模式能够提供什么。它涵盖“四人组”提到的 23 种设计模式。原始表格是由艾丽丝·尼尔森在 2004 年总结，我对一些必要的地方进行了修改，以使它与我们在本节中的讨论内容相符。

我推荐大家将该表格作为参考，但是要记住，这里还有很多未提及的其他模式，本书稍后将讨论这些模式。

## 有关类（Class）的要点

要记住，该表格中将会有“类”的概念。JavaScript 是一种无类语言，但可以使用函数来模拟类。

最常用的实现方法是定义一个 JavaScript 函数，然后使用 new 关键字创建新对象。使用 this 来定义对象的新属性和方法，如下所示：

```
// 一个 Car "class"
function Car(model) {

    this.model = model;
    this.color = "silver";
    this.year = "2012";

    this.getInfo = function () {
        return this.model + " " + this.year;
    }
}
```

```
};  
}
```

然后可以像这样使用我们上面定义的 car 构造函数来实例化该对象：

```
var myCar = new Car("ford");  
  
myCar.year = "2010";  
  
console.log(myCar.getInfo());
```

欲了解更多使用 JavaScript 来定义“类”的方法，请查看斯托扬·斯蒂凡诺夫发布的帖子（<http://www.phpied.com/3-ways-to-define-a-javascript-class/>）。

让我们继续来查看此表。

表 8-1

创建型模式	基于创建对象的概念
类	
工厂方法	基于接口数据或事件生成几个派生类的一个实例
对象	
抽象工厂	创建若干类系列的一个实例，无需详述具体的类
生成器	从表示中分离对象构建；总是创建相同类型的对象
原型	用于复制或克隆完全初始化的实例
单例	一个类在全局访问点只有唯一一个实例
结构型模式	基于构建对象块的想法
类	
适配器	匹配不同类的接口，因此类可以在不兼容接口的情况下共同工作
对象	
适配器	匹配不同类的接口，因此类可以在不兼容接口的情况下共同工作
桥接	将对象接口从其实现中分离，因此它们可以独立进行变化
组合	简单和复合对象的结构，使对象的总和不只是它各部分的总和
装饰	向对象动态添加备选的处理
外观	隐藏整个子系统复杂性的唯一一个类
享元	一个用于实现包含在别处信息的高效共享的细粒度实例
代理	占位符对象代表真正的对象
行为模式	基于对象在一起配合工作的方式

续表

类	
解释器	将语言元素包含在应用程序中的方法，以匹配预期语言的语法
模板方法	在方法中创建算法的 shell，然后将确切的步骤推到子类
对象	
职责链	在对象链之间传递请求的方法，以找到能够处理请求的对象
命令	将命令执行从其调用程序中分离的方法
迭代器	顺序访问一个集合中的元素，无需了解该集合的内部工作原理
中介者	在类之间定义简化的通信，以防止一组类显式引用彼此
备忘录	捕获对象的内部状态，以能够在以后恢复它
观察者	向多个类通知改变的方式，以确保类之间的一致性
状态	状态改变时，更改对象的行为
策略	在一个类中封装算法，将选择与实现分离
访问者	向类添加一个新的操作，无需改变类

## 第9章

# JavaScript 设计模式

在本章中，我们将探索一些经典与现代设计模式的 JavaScript 实现。

开发人员通常想知道他们是否应该在工作中使用一种“理想”的模式或模式集。这个问题没有明确的唯一答案，我们研究的每个脚本和 Web 应用程序可能都有它自己的个性化需求，我们需要思考模式的哪些方面能够为实现提供实际价值。

例如，一些项目可能会受益于观察者模式提供的解耦好处（这可以减少应用程序的某些部分对彼此的依赖度），而有些项目可能只是因为太小，而根本无需考虑解耦。

也就是说，一旦我们牢牢掌握了设计模式和与它们最为相配的具体问题，那么将它们集成到我们的应用程序架构中就会变得更加容易。

在本节中将要探索的模式包括：

- Constructor（构造器）模式；
- Module（模块）模式；
- Revealing Module（揭示模块）模式；
- Singleton（单例）模式；
- Observer（观察者）模式；

- Mediator（中介者）模式；
- Prototype（原型）模式；
- Command（命令）模式；
- Facade（外观）模式；
- Factory（工厂）模式；
- Mixin（混入）模式；
- Decorator（装饰者）模式；
- Flyweight（享元）模式。

## 9.1 Constructor（构造器）模式

在经典面向对象编程语言中，Constructor 是一种在内存已分配给该对象的情况下，用于初始化新创建对象的特殊方法。在 JavaScript 中，几乎所有东西都是对象，我们通常最感兴趣的是 *object* 构造器。

Object 构造器用于创建特定类型的对象——准备好对象以备使用，同时接收构造器可以使用的参数，以在第一次创建对象时，设置成员属性和方法的值（见图 9-1）。

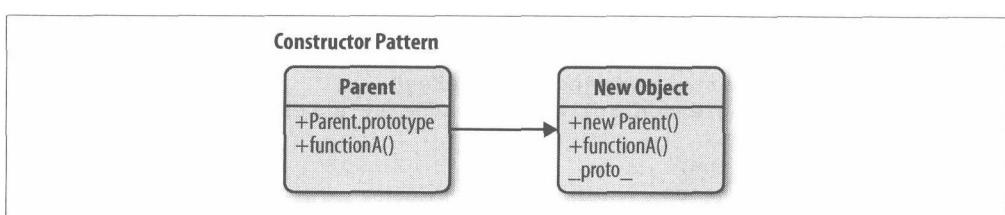


图 9-1 Constructor（构造器）模式

### 9.1.1 对象创建

在 JavaScript 中，创建新对象的两种常用方法如下所示：

```
//下面每种方式都将创建一个新的空对象  
  
var newObject = {};  
  
// object 构造器的简洁记法  
var newObject = new Object();
```

在 `Object` 构造器为特定的值创建对象封装，或者没有传递值时，它将创建一个空对象并返回它。

有四种方法可以将键值赋值给一个对象：

```
// ECMAScript 3 兼容方式  
  
// 1. “点”语法  
  
// 设置属性  
newObject.someKey = "Hello World";  
  
// 获取属性  
var key = newObject.someKey;  
  
// 2. 中括号语法  
  
// 设置属性  
newObject["someKey"] = "Hello World";  
  
// 获取属性  
var key = newObject["someKey"];  
  
// 只适用 ECMAScript 5 的方式  
// 更多信息查看: http://kangax.github.com/es5-compat-table/  
  
// 3. Object.defineProperty  
  
// 设置属性  
Object.defineProperty(newObject, "someKey", {  
    value: "for more control of the property's behavior",  
    writable: true,  
    enumerable: true,  
    configurable: true  
});  
  
// 如果上面的看着麻烦，可以使用下面的简便方式  
var defineProp = function (obj, key, value) {  
  
    config.value = value;  
    Object.defineProperty(obj, key, config);  
};
```

```
// 使用上述方式，先创建一个空的 person 对象
var person = Object.create(null);

// 然后设置各个属性
defineProp(person, "car", "Delorean");
defineProp(person, "dateOfBirth", "1981");
defineProp(person, "hasBeard", false);

// 4.Object.defineProperties

// 设置属性
Object.defineProperties(newObject, {

    "someKey": {
        value: "Hello World",
        writable: true
    },
    "anotherKey": {
        value: "Foo bar",
        writable: false
    }
});

// 可以用 1 和 2 中获取属性的方式获取 3 和 4 方式中的属性
```

正如我们将在本书稍后看到的，这些方法甚至可以用于继承，如下所示：

```
// 用法

// 创建赛车司机 driver 对象，继承于 person 对象
var driver = Object.create(person);

// 为 driver 设置一些属性
defineProp(driver, "topSpeed", "100mph");

// 获取继承的属性
console.log(driver.dateOfBirth);

// 获取我们设置的 100mph 的属性
console.log(driver.topSpeed);
```

## 9.1.2 基本 Constructor (构造器)

正如我们在前面所看到的，JavaScript 不支持类的概念，但它确实支持与对象一起用的特殊 `constructor`（构造器）函数。通过在构造器前面加 `new` 关键字，告诉 JavaScript 像使用构造器一样实例化一个新对象，并且对象成员由该函数定义。

在构造器内，关键字 `this` 引用新创建的对象。回顾对象创建，基本的构造器看起来

可能是这样的：

```
function Car(model, year, miles) {  
    this.model = model;  
    this.year = year;  
    this.miles = miles;  
  
    this.toString = function () {  
        return this.model + " has done " + this.miles + " miles";  
    };  
}  
  
// 用法  
  
// 可以创建 car 的新实例  
var civic = new Car("Honda Civic", 2009, 20000);  
var mondeo = new Car("Ford Mondeo", 2010, 5000);  
  
// 打开浏览器控制台，查看这些对象上调用的 toString() 方法的输出  
console.log(civic.toString());  
console.log(mondeo.toString());
```

上面是一个简单的构造器模式版本，但它确实有一些问题。其中一个问题，是它使继承变得困难，另一个问题是，`toString()`这样的函数是为每个使用 `Car` 构造器创建的新对象而分别重新定义的。这不是最理想的，因为这种函数应该在所有的 `Car` 类型实例之间共享。

值得庆幸的是，因为有很多 ES3 和 ES5 兼容替代方法能够用于创建对象，所以很容易解决这个限制问题。

### 9.1.3 带原型的 Constructor ( 构造器 )

JavaScript 中有一个名为 `prototype` 的属性。调用 JavaScript 构造器创建一个对象后，新对象就会具有构造器原型的所有属性。通过这种方式，可以创建多个 `Car` 对象，并访问相同的原型。因此我们可以扩展原始示例，如下所示：

```
function Car(model, year, miles) {  
    this.model = model;  
    this.year = year;  
    this.miles = miles;  
}  
// 注意这里我们使用 Object.prototype.newMethod 而不是 Object.prototype 是为了避免重新定义 prototype 对象  
Car.prototype.toString = function () {  
    return this.model + " has done " + this.miles + " miles";  
};
```

```
//用法：  
  
var civic = new Car("Honda Civic", 2009, 20000);  
var mondeo = new Car("Ford Mondeo", 2010, 5000);  
  
console.log(civic.toString());  
console.log(mondeo.toString());
```

现在 `toString()` 的单一实例就能够 在所有 `Car` 对象之间共享。

## 9.2 Module（模块）模式

模块是任何强大应用程序架构中不可或缺的一部分，它通常能够帮助我们清晰地分离和组织项目中的代码单元。

在 JavaScript 中，有几种用于实现模块的方法，包括：

- 对象字面量表示法
- Module 模式
- AMD 模块
- CommonJS 模块
- ECMAScript Harmony 模块

我们稍后将在本书第 11 章探索后三种方法。Module 模式在某种程度上是基于对象字面量，因此首先重新认识对象字面量是有意义的。

### 9.2.1 对象字面量

在对象字面量表示法中，一个对象被描述为一组包含在大括号 (`{ }` ) 中、以逗号分隔的 `name/value` 对。对象内的名称可以是字符串或标识符，后面跟着一个冒号。对象中最后的一个 `name/value` 对的后面不用加逗号，如果加逗号将会导致出错。

```
var myObjectLiteral = {  
    variableKey: variableValue,  
    functionKey: function () {
```

```
// ...
}
};
```

对象字面量不需要使用 `new` 运算符进行实例化，但不能用在一个语句的开头，因为开始的可能被解读为一个块的开始。在对象的外部，新成员可以使用如下赋值语句添加到对象字面量上，如：`myModule.property = "someValue";`

下面我们可以看到一个更完整的示例：使用对象字面量表示法定义的模块：

```
var myModule = {

    myProperty: "someValue",

    // 对象字面量可以包含属性和方法
    // 例如，可以声明模块的配置对象
    myConfig: {
        useCaching: true,
        language: "en"
    },

    // 基本方法
    myMethod: function () {
        console.log("Where in the world is Paul Irish today?");
    },

    // 根据当前配置输出信息
    myMethod2: function () {
        console.log("Caching is:" + (this.myConfig.useCaching) ? "enabled" : "disabled");
    },
    // 重写当前的配置
    myMethod3: function (newConfig) {

        if (typeof newConfig === "object") {
            this.myConfig = newConfig;
            console.log(this.myConfig.language);
        }
    }
};

// 输出: Where in the world is Paul Irish today?
myModule.myMethod();

//输出: enabled
myModule.myMethod2();

//输出: fr
myModule.myMethod3({
    language: "fr",
    useCaching: false
});
```

使用对象字面量有助于封装和组织代码，如果想进一步了解有关对象字面量的信息，丽贝卡·墨菲曾对这一主题进行了深入解析，可阅读其文章进行了解（地址：<http://rmurphrey.com/blog/2009/10/15/using-objects-to-organize-your-code/>）。

也就是说，如果我们选择了这种技术，我们可能同样也对 Module 模式感兴趣。它仍然使用对象字面量，但只是作为一个作用域函数的返回值。

## 9.2.2 Module（模块）模式

Module 模式最初被定义为一种在传统软件工程中为类提供私有和公有封装的方法。

在 JavaScript 中，Module 模式用于进一步模拟类的概念，通过这种方式，能够使一个单独的对象拥有公有/私有方法和变量，从而屏蔽来自全局作用域的特殊部分。产生的结果是：函数名与在页面上其他脚本定义的函数冲突的可能性降低（见图 9-2）。

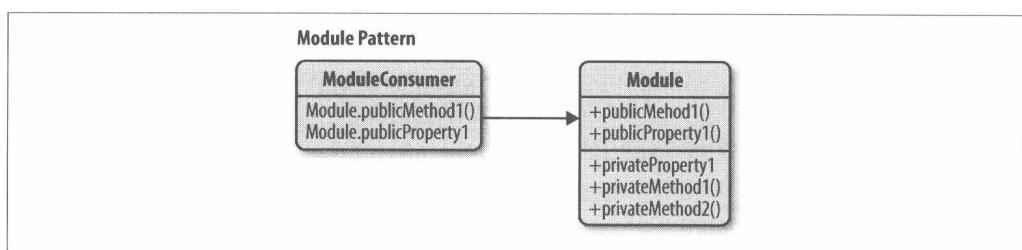


图 9-2 Module 模式

### 9.2.2.1 私有

Module 模式使用闭包封装“私有”状态和组织。它提供了一种包装混合公有/私有方法和变量的方式，防止其泄露至全局作用域，并与别的开发人员的接口发生冲突。通过该模式，只需返回一个公有 API，而其他的一切则都维持在私有闭包里。

这为我们提供了一个屏蔽处理底层事件逻辑的整洁解决方案，同时只暴露一个接口供应用程序的其他部分使用。该模式除了返回一个对象而不是一个函数之外，非常类似于一个立即调用的函数表达式<sup>1</sup>。

<sup>1</sup> IIFE (<http://benalman.com/news/2010/11/immediately-invoked-function-expression/>)。请参阅第 200 页的“命名空间模式”了解更多信息。

应该指出的是，在 JavaScript 中没有真正意义上的“私有”，因为不像有些传统语言，JavaScript 没有访问修饰符。从技术上来说，我们不能称变量为公有或是私有，因此我们需使用函数作用域来模拟这个概念。在 Module 模式内，由于闭包的存在，声明的变量和方法只在该模式内部可用。但在返回对象上定义的变量和方法，则对外部使用者都是可用的。

### 9.2.2.2 历史

从历史的角度来看，Module 模式最初是在 2003 年由多人共同开发出来的，其中包括理查德·康佛德 (<http://groups.google.com/group/comp.lang.javascript/msg/9f58bd11bd67d937>)。后来由道格拉斯·克劳克福德在其讲座中推广开来。除此之外，如果你曾体验过雅虎的 YUI 库，它的一些特性看起来可能相当熟悉，原因是在创建它们的组件时，Module 模式对 YUI 有很大的影响。

### 9.2.2.3 示例

让我们通过创建一个自包含的模块来看一下 Module 模式的实现。

```
var testModule = (function () {
    var counter = 0;

    return {
        incrementCounter: function () {
            return ++counter;
        },
        resetCounter: function () {
            console.log("counter value prior to reset: " + counter);
            counter = 0;
        }
    };
})();

//用法:
//增加计数器
testModule.incrementCounter();

// 检查计数器值并重置
//输出: 1
testModule.resetCounter();
```

在这里，代码的其他部分无法直接读取 `incrementCounter()` 或 `resetCounter()`。`counter` 变量

实际上是完全与全局作用域隔离的，因此它表现得就像是一个私有变量，它的存在被局限于模块的闭包内，因此唯一能够访问其作用域的代码就是这两个函数。上述方法进行了有效的命名空间设置，所以在测试代码中，所有的调用都需要加上前缀（如：“`testModule`”）。

使用 `Module` 模式时，可能会觉得它可以用来定义一个简单的模板来入门使用。下面是一个包含命名空间、公有和私有变量的 `Module` 模式：

```
var myNamespace = (function () {

    // 私有计数器变量
    var myPrivateVar = 0;

    // 记录所有参数的私有函数
    var myPrivateMethod = function (foo) {
        console.log(foo);
    };

    return {

        // 公有变量
        myPublicVar: "foo",

        // 调用私有变量和方法的公有函数
        myPublicFunction: function (bar) {

            // 增加私有计数器值
            myPrivateVar++;

            // 传入 bar 调用私有方法
            myPrivateMethod(bar);
        }
    };
})();
```

来看另一个示例，我们可以看到一个使用这种模式实现的购物车。模块本身是完全自包含在一个被称为 `basketModule` 的全局变量中。模块中的 `basket` 数组是私有的，因此应用程序的其他部分无法直接读取它。它只与模块的闭包一起存在，所以能够访问它的方法都是那些能够访问其作用域的方法（即 `addItem()`、`getItem()` 等）。

```
var basketModule = (function () {

    // 私有

    var basket = [];

    function doSomethingPrivate() {
```

```

    //...
}

function doSomethingElsePrivate() {
    //...
}

// 返回一个暴露出的公有对象
return {

    // 添加 item 到购物车
    addItem: function (values) {
        basket.push(values);
    },

    // 获取购物车里的 item 数
    getItemCount: function () {
        return basket.length;
    },
    // 私有函数的公有形式别名
    doSomething: doSomethingPrivate,

    // 获取购物车里所有 item 的价格总值
    getTotal: function () {

        var itemCount = this.getItemCount(),
            total = 0;

        while (itemCount--) {
            total += basket[itemCount].price;
        }

        return total;
    }
};
})();

```

在该模块中，可能已经注意到返回了一个 object。它会被自动赋值给 basketModule，以便我们可以与它交互，如下所示：

```

// basketModule 返回了一个拥有公用 API 的对象

basketModule.addItem({
    item: "bread",
    price: 0.5
});

basketModule.addItem({
    item: "butter",
    price: 0.3
});

```

```

// 输出: 2
console.log(basketModule.getItemCount());

// 输出: 0.8
console.log(basketModule.getTotal());

// 不过, 下面的代码不会正常工作

// 输出:undefined
// 因为 basket 自身没有暴露在公有的 API 里
console.log(basketModule.basket);

// 下面的代码也不会正常工作, 因为 basket 只存在于 basketModule 闭包的作用域里, 而不
是存在于返回的公有对象里
console.log(basket);

```

上述方法在 `basketModule` 内部都属于有效的命名空间设置。

请注意上面的 `basket` 模块中的作用域函数是如何包裹在所有函数的周围, 然后调用并立即存储返回值。这有很多优点, 包括:

- 只有我们的模块才能享有拥有私有函数的自由。因为它们不会暴露于页面的其余部分 (只会暴露于我们输出的 API), 我们认为它们是真正的私有。
- 鉴于函数往往已声明并命名, 在试图找到有哪些函数抛出异常时, 这将使得在调试器中显示调用堆栈变得更容易。
- 正如 T.J.Crowder 在过去所指出的, 根据环境, 它还可以让我们返回不同的函数。在过去, 我曾看到开发人员使用它来执行 UA 测试, 从而针对 IE 在他们的模块内提供一个代码路径, 但我们现在可以很容易地选择特征检测来实现类似的目的。

### 9.2.3 Module 模式变化

#### 9.2.3.1 引入混入

模式的这种变化演示了全局变量 (如: `jQuery`、`Underscore`) 如何作为参数传递给模块的匿名函数。这允许我们引入它们, 并按照我们所希望的为它们取个本地别名。

```

// 全局模块
var myModule = (function ($, _) {
    function privateMethod1() {
        $("container").html("test");
    }
    return {
        publicMethod1: privateMethod1
    };
});

```

```

        }

        function privateMethod2() {
            console.log(_.min([10, 5, 100, 2, 1000]));
        }

        return {
            publicMethod: function () {
                privateMethod1();
            }
        };
    // 引入 jQuery 和 Underscore
})(jQuery, _));
}

myModule.publicMethod();

```

### 9.2.3.2 引出

下一个变化允许我们声明全局变量，而不需实现它们，并可以同样地支持上一个示例中的全局引入的概念。

```

// 全局模块
var myModule = (function () {

    // 模块对象
    var module = {},
        privateVariable = "Hello World";

    function privateMethod() {
        // ...
    }

    module.publicProperty = "Foobar";
    module.publicMethod = function () {
        console.log(privateVariable);
    };

    return module;
})();

```

### 9.2.3.3 工具包和特定框架的 Module 模式实现

**Dojo**。提供了一种和对象一起用的便利方法 `dojo.setObject()`。其第一个参数是用点号分割的字符串，如 `myObj.parent.child`，它在 `parent` 对象中引用一个称为 `child` 的属性，`parent` 对象是在 `myObj` 内部定义。我们可以使用 `setObject()` 设置子级的值（比如属性等），

如果中间对象不存在的话，也可以通过点号分割将中间的字符作为中间对象进行创建。

例如，如果要将 `basket.core` 声明为 `store` 名称空间的对象，可以采用传统的方法来实现，如下所示：

```
var store = window.store || {};
if (!store["basket"]) {
    store.basket = {};
}
if (!store.basket["core"]) {
    store.basket.core = {};
}
store.basket.core = {
    // ...剩余的逻辑
};
```

或者，使用 Dojo 1.7（AMD 兼容的版本）和上述方法，如下所示：

```
require(["dojo/_base/customStore"], function (store) {
    // 使用 dojo.setObject()
    store.setObject("basket.core", (function () {
        var basket = [];

        function privateMethod() {
            console.log(basket);
        }
        return {
            publicMethod: function () {
                privateMethod();
            }
        };
    })());
});
```

欲了解更多有关 `dojo.setObject()` 的信息，请参阅官方文档 (<http://dojotoolkit.org/reference-guide/1.7/dojo/setObject.html>)。

**ExtJS**。对比那些使用 Sencha ExtJS 的人，你的运气会好一点，因为官方文档包含了一些示例 ([http://www.sencha.com/learn/legacy/Tutorial:Application\\_Layout\\_for\\_Beginners](http://www.sencha.com/learn/legacy/Tutorial:Application_Layout_for_Beginners))，演示了 EXTJS 框架下如何正确使用 Module 模式。

在这里，我们可以看到这样的一个示例：如何定义一个名称空间，然后填充一个包

含私有和公有 API 的模块。除了一些语义差异，它与如何在纯 JavaScript 中实现 **Module** 模式十分相近。

```
// 创建命名空间
Ext.namespace("myNameSpace");

// 创建应用程序
myNameSpace.app = function () {

    // 这里不要访问 DOM，因为元素还不存在
    // 私有变量

    var btn1,
        privVar1 = 11;

    // 私有函数
    var btn1Handler = function (button, event) {
        console.log("privVar1", privVar1);
        console.log("this.btn1Text=" + this.btn1Text);
    };

    // 公有对象
    return {
        // 公有属性，例如要转化的字符
        btn1Text: "Button 1",

        // 公有方法
        init: function () {

            if (Ext.Ext2) {

                btn1 = new Ext.Button({
                    renderTo: "btn1-ct",
                    text: this.btn1Text,
                    handler: btn1Handler
                });

            } else {

                btn1 = new Ext.Button("btn1-ct", {
                    text: this.btn1Text,
                    handler: btn1Handler
                });

            }
        }
    };
}();
```

**YUI。** 同样，在使用 YUI3 构建应用程序时，我们也可以实现 **Module** 模式。下面的示例在很大程度上基于由 Eric Miraglia 提出的原始 YUI Module 模式实现，但它

又与纯 JavaScript 版本截然不同。

```
Y.namespace("store.basket") = (function () {
    var myPrivateVar, myPrivateMethod;
    // 私有变量:
    myPrivateVar = "I can be accessed only within Y.store.basket.";
    // 私有方法:
    myPrivateMethod = function () {
        Y.log("I can be accessed only from within YAHOO.store.basket");
    }
    return {
        myPublicProperty: "I'm a public property.",
        myPublicMethod: function () {
            Y.log("I'm a public method.");
            // 在 basket 里, 可以访问到私有变量和方法
            Y.log(myPrivateVar);
            Y.log(myPrivateMethod());
        }
        // myPublicMethod 的原始作用域是 store, 所以可以使用 this 来访问公有成员
        Y.log(this.myPublicProperty);
    };
});()
```

**jQuery。**有许多方式可以将非 jQuery 插件代码包装在 Module 模式中。如果模块之间有多个共性, Ben Cherry 之前建议过一种实现, 在模块模式内部模块定义附件使用函数包装器。

在下面的示例中, 定义了 library 函数, 它声明一个新库, 并在创建新库(即模块)时将 init 函数自动绑定到 document.ready。

```
function library(module) {
    $(function () {
        if (module.init) {
            module.init();
        }
    });
    return module;
}
var myLibrary = library(function () {
    return {
```

```
init: function () {
    // module implementation
// 模块实现
}
);
})();

```

#### 9.2.3.4 优点

我们已经了解了单例模式如何有用，但为什么 Module 模式是一个好的选择呢？首先，相比真正封装的思想，它对于很多拥有面向对象背景的开发人员来说更加整洁，至少是从 JavaScript 的角度。

其次，它支持私有数据，因此，在 Module 模式中，代码的公有（public）部分能够接触私有部分，然而外界无法接触类的私有部分。

#### 9.2.3.5 缺点

Module 模式的缺点是：由于我们访问公有和私有成员的方式不同，当我们想改变可见性时，实际上我们必须要修改每一个曾经使用过该成员的地方。

我们也无法访问那些之后在方法里添加的私有成员。也就是说，在很多情况下，如果正确使用，Module 模式仍然是相当有用的，肯定可以改进应用程序的结构。

其他缺点包括：无法为私有成员创建自动化单元测试，bug 需要修正补丁时会增加额外的复杂性。为私有方法打补丁是不可能的。相反，我们必须覆盖所有与有 bug 的私有方法进行交互的公有方法。另外开发人员也无法轻易地扩展私有方法，所以要记住，私有方法并不像它们最初显现出来的那么灵活。

欲进一步阅读有关 Module 模式的信息，请参阅 Ben Cherry 更为深入的精彩文章 (<http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth>)。

## 9.3 Revealing Module（揭示模块）模式

现在对 Module 模式应该更加熟悉了，让我们来看一个稍有改进的版本——Christian Heilmann 的 Revealing Module 模式。

Revealing Module 模式的产生是因为 Heilmann 很不满意这种状况：当他想从另一个方法调用一个公有方法或访问公有变量时，必须要重复主对象的名称。他也不喜欢

使用 Module 模式时，必须要切换到对象字面量表示法来让某种方法变成公有方法。

他努力的结果就是创建了一个更新的模式，能够在私有范围内简单定义所有的函数和变量，并返回一个匿名对象，它拥有指向私有函数的指针，该函数是他希望展示为公有的方法。

有关如何使用 Revealing Module 模式的示例如下所示：

```
var myRevealingModule = function () {  
    var privateVar = "Ben Cherry",  
        publicVar = "Hey there!";  
  
    function privateFunction() {  
        console.log("Name:" + privateVar);  
    }  
  
    function publicSetName(strName) {  
        privateName = strName;  
    }  
  
    function publicGetName() {  
        privateFunction();  
    }  
  
    // 将暴露的公有指针指向到私有函数和属性上  
  
    return {  
        setName: publicSetName,  
        greeting: publicVar,  
        getName: publicGetName  
    };  
}();  
  
myRevealingModule.setName("Paul Kinlan");
```

如果你喜欢，该模式也可以用于展示拥有更具体命名方案的私有函数和属性：

```
var myRevealingModule = function () {  
    var privateCounter = 0;  
  
    function privateFunction() {  
        privateCounter++;  
    }  
  
    function publicFunction() {  
        publicIncrement();  
    }  
}
```

```
function publicIncrement() {
    privateFunction();
}

function publicGetCount() {
    return privateCounter;
}

// 将暴露的公有指针指向到私有函数和属性上

return {
    start: publicFunction,
    increment: publicIncrement,
    count: publicGetCount
};

}();

myRevealingModule.start();
```

### 9.3.1 优点

该模式可以使脚本语法更加一致。在模块代码底部，它也会很容易指出哪些函数和变量可以被公开访问，从而改善可读性。

### 9.3.2 缺点

该模式的一个缺点是：如果一个私有函数引用一个公有函数，在需要打补丁时，公有函数是不能被覆盖的。这是因为私有函数将继续引用私有实现，该模式并不适用于公有成员，只适用于函数。

引用私有变量的公有对象成员也遵守无补丁规则。

正因为如此，采用 Revealing Module 模式创建的模块可能比那些采用原始 Module 模式创建的模块更加脆弱，所以在使用时应该特别小心。

## 9.4 Singleton（单例）模式

Singleton（单例）模式被熟知的原因是因为它限制了类的实例化次数只能一次。从经典意义上来说，Singleton 模式，在该实例不存在的情况下，可以通过一个方法创建一个类来实现创建类的新实例；如果实例已经存在，它会简单返回该对象的引用。Singleton 不同于静态类（或对象），因为我们可以推迟它们的初始化，这通常是因为它们需要一些

信息，而这些信息在初始化期间可能无法获得。对于没有察觉到之前的引用的代码，它们不会提供方便检索的方法。这是因为它既不是对象，也不是由一个 Singleton 返回的“类”；它是一个结构。思考一下闭包变量为何实际上并不是闭包，而提供闭包的函数作用域是闭包。在 JavaScript 中，Singleton 充当共享资源命名空间，从全局命名空间中隔离开代码实现，从而为函数提供单一访问点。我们可以像如下这样实现一个 Singleton：

```
var mySingleton = (function () {
    // 实例保持了 Singleton 的一个引用
    var instance;
    function init() {
        // Singleton
        // 私有方法和变量
        function privateMethod() {
            console.log("I am private");
        }
        var privateVariable = "Im also private";
        var privateRandomNumber = Math.random();
        return {
            // 公有方法和变量
            publicMethod: function () {
                console.log("The public can see me!");
            },
            publicProperty: "I am also public",
            getRandomNumber: function () {
                return privateRandomNumber;
            }
        };
    };
    return {
        // 获取 Singleton 的实例，如果存在就返回，不存在就创建新实例
        getInstance: function () {
            if (!instance) {
                instance = init();
            }
            return instance;
        }
    };
})();

var myBadSingleton = (function () {
    // 实例保持了 Singleton 的一个引用
    var instance;
    function init() {
        // Singleton
        var privateRandomNumber = Math.random();
        return {
            getRandomNumber: function () {
                return privateRandomNumber;
            }
        };
    }
});
```

```

    };
    return {
        // 每次都创建新实例
        getInstance: function () {
            instance = init();
            return instance;
        }
    };
})();

var singleA = mySingleton.getInstance();
var singleB = mySingleton.getInstance();
console.log(singleA.getRandomNumber() === singleB.getRandomNumber()); // true

var badSingleA = myBadSingleton.getInstance();
var badSingleB = myBadSingleton.getInstance();
console.log(badSingleA.getRandomNumber() !== badSingleB.getRandomNumber()); // true

```

是什么使 Singleton 成为实例的全局访问入口(通常通过 MySingleton.getInstance()), 因为我们没有(至少在静态语言中)直接调用新的 MySingleton()。然而, 这在 JavaScript 中是可能的。在“四人组”所著的书中, 有关 Singleton 模式适用性的描述如下。

- 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
- 该唯一的实例应该是通过子类化可扩展的, 并且客户应该无需更改代码就能使用一个扩展的实例时。

这些观点另外关联到一个场景, 这里我们可能需要如下这样的代码:

```

mySingleton.getInstance = function () {
    if (this._instance == null) {
        if (isFoo()) {
            this._instance = new FooSingleton();
        } else {
            this._instance = new BasicSingleton();
        }
    }
    return this._instance;
};

```

在这里, getInstance 变得有点像 Factory(工厂)方法, 当访问它时, 我们不需要更新代码中的每个访问点。FooSingleton(上面)将是一个 BasicSingleton 的子类, 并将实现相同的接口。

为何延迟执行对于 Singleton 很重要?



在 C++ 中， Singleton 负责隔绝动态初始化顺序的不可预知性，将控制权归还给程序员。

值得注意的是类的静态实例（对象）和 Singleton 之间的区别：当 Singleton 可以作为一个静态的实例实现时，它也可以延迟构建，直到需要使用静态实例时，无需使用资源或内存。

如果我们有一个可以直接被初始化的静态对象，需要确保执行代码的顺序总是相同的（例如：在初始化期间 objCar 需要 objWheel 的情况），当我们有大量的源文件时，它并不能伸缩。

Singleton 和静态对象都是有用的，但是我们不应当以同样的方式过度使用它们，也不应过度使用其他模式。

在实践中，当在系统中确实需要一个对象来协调其他对象时， Singleton 模式是很有用的。在这里，大家可以看到在这个上下文中模式的使用：

```
var SingletonTester = (function () {  
  
    // options: 包含 singleton 所需配置信息的对象  
    // e.g var options = { name: "test", pointX: 5};  
    function Singleton( options ) {  
  
        // 如果未提供 options，则设置为空对象  
        options = options || {};  
  
        // 为 singleton 设置一些属性  
        this.name = "SingletonTester";  
  
        this.pointX = options.pointX || 6;  
  
        this.pointY = options.pointY || 10;  
    }  
  
    // 实例持有者  
    var instance;  
  
    // 静态变量和方法的模拟  
    var _static = {  
  
        name: "SingletonTester",
```

```

// 获取实例的方法，返回 singleton 对象的 singleton 实例
getInstance: function( options ) {
    if( instance === undefined ) {
        instance = new Singleton( options );
    }

    return instance;
}
};

return _static;

})();
var singletonTest = SingletonTester.getInstance({
    pointX: 5
});

// 记录 pointX 的输出以便验证
// 输出: 5
console.log( singletonTest.pointX );

```

**Singleton** 很有使用价值，通常当发现在 JavaScript 中需要它的时候，则表示我们可能需要重新评估我们的设计。

**Singleton** 的存在往往表明系统中的模块要么是系统紧密耦合，要么是其逻辑过于分散在代码库的多个部分。由于一系列的问题：从隐藏的依赖到创建多个实例的难度、底层依赖的难度等等，**Singleton** 的测试会更加困难。

Miller Medeiros 之前曾推荐过这篇优秀的文章 (<http://www.ibm.com/developerworks/ebservices/library/co-single/index.html>)，以进一步了解 **Singleton** 和它的各种问题。他还建议阅读有关这篇文章 (<http://misko.hevery.com/2008/10/21/dependency-injection-myth-reference-passing/>) 的评论，包括讨论 **Singleton** 如何能够加强紧密耦合。我很高兴再次推荐这些内容，因为这两种内容都对这种模式提出了许多重要的问题。

## 9.5 Observer（观察者）模式

**Observer**（观察者）是一种设计模式，其中，一个对象（称为 **subject**）维持一系列依赖于它（观察者）的对象，将有关状态的任何变更自动通知给它们（见图 9-3）。

当一个目标需要告诉观察者发生了什么有趣的事情，它会向观察者广播一个通知（可以包括与通知主题相关的特定数据）。

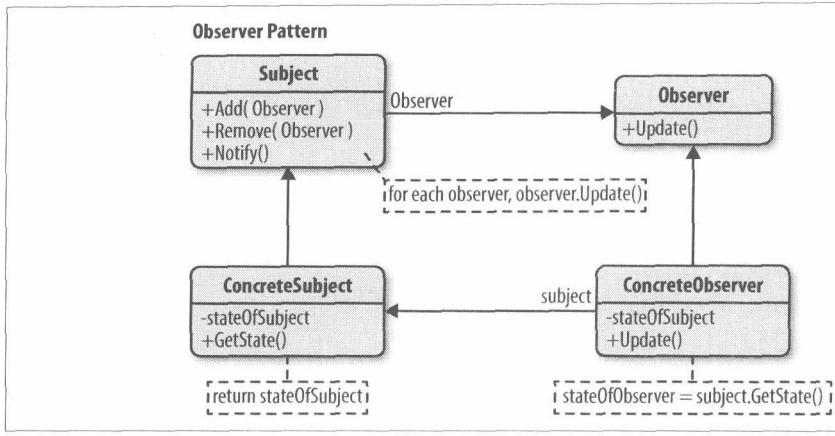


图 9-3 Observer 模式

当我们不再希望某个特定的观察者获得其注册目标发出的改变通知时，该目标可以将它从观察者列表中删除。

参考之前发布的设计模式定义通常很有用的，它与语言无关，以便久而久之使其使用和优势变得更有意义。“四人组”所著书籍（《设计模式：可复用面向对象软件的基础》）中提供的 Observer 模式的定义是：



“一个或多个观察者对目标的状态感兴趣，它们通过将自己依附在目标对象上以便注册所感兴趣的内容。目标状态发生改变并且观察者可能对这些改变感兴趣，就会发送一个通知消息，调用每个观察者的更新方法。当观察者不再对目标状态感兴趣时，它们可以简单地将自己从中分离。”

现在我们可以扩展所学到的内容来使用以下组件实现 Observer 模式：

### *Subject (目标)*

维护一系列的观察者，方便添加或删除观察者

### *Observer (观察者)*

为那些在目标状态发生改变时需获得通知的对象提供一个更新接口

### *ConcreteSubject (具体目标)*

状态发生改变时，向 Observer 发出通知，储存 ConcreteObserver 的状态

### *ConcreteObserver* (具体观察者)

存储一个指向 *ConcreteSubject* 的引用，实现 *Observer* 的更新接口，以使自身状态与目标的状态保持一致

首先，让我们来模拟一个目标可能拥有的一系列依赖 *Observer*:

```
function ObserverList(){
    this.observerList = [];
}

ObserverList.prototype.Add = function(obj){
    return this.observerList.push(obj);
};

ObserverList.prototype.Empty = function(){
    this.observerList = [];
};

ObserverList.prototype.Count = function(){
    return this.observerList.length;
};

ObserverList.prototype.Get = function (index){
    if (index > -1 && index <this.observerList.length) {
        return this.observerList[index];
    }
};

ObserverList.prototype.Insert = function(obj, index){
    var pointer = -1;

    if (index === 0){
        this.observerList.unshift(obj);
        pointer = index;
    } elseif (index === this.observerList.length){
        this.observerList.push(obj);
        pointer = index;
    }
    return pointer;
};

ObserverList.prototype.IndexOf = function(obj, startIndex){
    var i = startIndex, pointer = -1;

    while (i <this.observerList.length){
        if (this.observerList[i] === obj){
            pointer = i;
        }
        i++;
    }
}
```

```

        return pointer;
    };

ObserverList.prototype.RemoveIndexAt() = function (index){
    if (index === 0){
        this.observerList.shift();
    } else if(index === this.observerList.length - 1){
        this.observerList.pop();
    }
} else{
    this.observerList.splice(index, 1);
}
// 使用 extension 扩展对象
function extend(obj, extension){
    for (var key in obj){
        extension[key] = obj[key];
    }
}

```

接下来，让我们来模拟目标（Subject）和在观察者列表上添加、删除或通知观察者的能力。

```

function Subject() {
    this.observers = new ObserverList();
}

Subject.prototype.AddObserver = function(observer){
    this.observers.Add(observer);
};

Subject.prototype.RemoveObserver = function (observer){
    this.observers.RemoveIndexAt(this.observers.IndexOf(observer, 0));
};

Subject.prototype.Notify = function (context){
    var observerCount = this.observers.Count();
    for (var i = 0; i < observerCount; i++){
        this.observers.Get(i).Update(context);
    }
};

```

然后定义一个框架来创建新的 Observer。这里的 Update 功能将在后面的自定义行为部分进一步介绍。

```

// The Observer
function Observer(){
    this.Update = function (){
        // ...
    };
}

```

在使用上述 **Observer** 组件的样例应用程序中，定义如下：

- 用于向页面添加新可见 **checkbox** 的按钮
- 控制 **checkbox**，将充当一个目标，通知其他 **checkbox** 需要进行检查
- 用于添加新 **checkbox** 的容器

然后定义 **ConcreteSubject** 和 **ConcreteObserver** 处理程序，以便向页面添加新观察者，并实现更新界面。关于这些组件在示例上下文中的作用注释，请参阅下面的内容。

如下是 HTML 代码：

```
<button id="addNewObserver">Add New Observer checkbox</button>
<input id="mainCheckbox" type="checkbox"/>
<div id="observersContainer"></div>
```

如下是样例脚本：

```
// 引用 DOM 元素

var controlCheckbox = document.getElementById("mainCheckbox"),
    addBtn = document.getElementById("addNewObserver"),
    container = document.getElementById("observersContainer");

// 具体目标 Concrete Subject

// 利用 Subject 扩展 controlCheckbox
extend(new Subject(), controlCheckbox);

// 点击 checkbox 会触发通知到观察者上
controlCheckbox["onclick"] = new Function("controlCheckbox.Notify(
controlCheckbox.checked)");

addBtn["onclick"] = AddNewObserver;

// 具体观察者 Concrete Observer

function AddNewObserver() {
    // 创建需要添加的新 checkbox
    var check = document.createElement("input");
    check.type = "checkbox";

    // 利用 Observer 类扩展 checkbox
    extend(new Observer(), check);

    // 重写自定义更新行为
}
```

```

check.Update = function (value) {
    this.checked = value;
};

// 为主 subject 的观察者列表添加新的观察者
controlCheckbox.AddObserver(check);

// 将观察者附件到容器上
container.appendChild(check);
}

```

在本例中，我们了解了如何实现和使用 Observer 模式，包含目标（Subject）、观察者（Observer）、具体目标（ConcreteSubject）和具体观察者（ConcreteObserver）的概念。

### 9.5.1 Observer（观察者）模式和 Publish/Subscribe（发布/订阅）模式的区别

通常在 JavaScript 里，注重 Observer 模式是很有用的，我们会发现它一般使用一个被称为 Publish/Subscribe（发布/订阅）模式的变量来实现。虽然这些模式非常相似，但是它们之间的几点区别也是值得注意的。

Observer 模式要求希望接收到主题通知的观察者（或对象）必须订阅内容改变的事件，如图 9-4 所示。

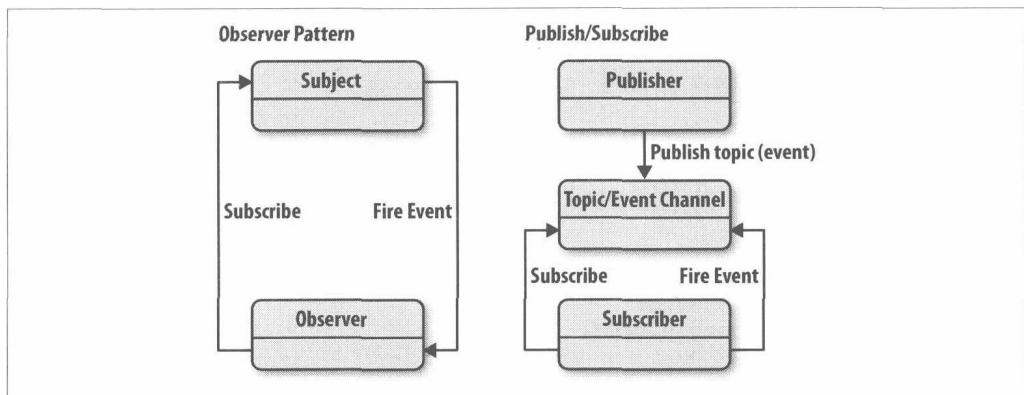


图 9-4 Publish/Subscribe

Publish/Subscribe 模式使用了一个主题/事件通道，这个通道介于希望接收到通知（订阅者）的对象和激活事件的对象（发布者）之间。该事件系统允许代码定义应用程序的特定事件，这些事件可以传递自定义参数，自定义参数包含订阅者所需的值。

其目的是避免订阅者和发布者之间产生依赖关系。

这与 Observer 模式不同，因为它允许任何订阅者执行适当的事件处理程序来注册和接收发布者发出的通知。

下面这个示例说明了如果有 publish()、subscribe() 和 unsubscribe() 的功能实现，是如何使用 Publish/Subscribe 模式的：

```
// 非常简单的 mail 处理程序

// 接收到的消息数量
var mailCounter = 0;

// 初始化订阅，名称是 inbox/newMessage

// 呈现消息预览
var subscriber1 = subscribe("inbox/newMessage", function (topic, data) {

    // debug 模式记录 topic
    console.log("A new message was received: ", topic);

    // 使用从目标 subject 传递过来的 data，一般呈现消息预览
    $(".messageSender").html(data.sender);
    $(".messagePreview").html(data.body);

});

// 另外一个订阅，使用同样的 data 数据用于不同的任务

// 通过发布者更新所接收消息的数量

var subscriber2 = subscribe("inbox/newMessage", function (topic, data) {

    $('.newMessageCounter').html(mailCounter++);

});

publish("inbox/newMessage", [
    sender: "hello@google.com",
    body: "Hey there! How are you doing today?"
]);

// 之后可以通过 unsubscribe 来取消订阅
// unsubscribe( subscriber1 );
// unsubscribe( subscriber2 );
```

这里的中心思想是促进松散耦合。通过订阅另一个对象的特定任务或活动，当任务/活动发生时获得通知，而不是单个对象直接调用其他对象的方法。

## 9.5.2 优点

Observer 模式和 Publish/Subscribe 模式鼓励我们努力思考应用程序不同部分之间的关系。它们也帮助我们识别包含直接关系的层，并且可以用目标集和观察者进行替换。实际上可以用于将应用程序分解为更小、更松散耦合的块，以改进代码管理和潜在的复用。

使用 Observer 模式背后的另一个动机是我们需要在哪里维护相关对象之间的一致性，而无需使类紧密耦合。例如，当一个对象需要能够通知其他对象，而无需在这些对象方面做假设时。

在使用任何一种模式时，动态关系可以在观察者和目标之间存在。这提供了很大的灵活性，当应用程序的不同部分紧密耦合时，这可不是很容易实现的。

虽然它可能不一直是所有问题的最佳解决方案，但这些模式仍是用于设计解耦性系统的最佳工具之一，应该视为所有 JavaScript 开发人员工具中的一个重要工具。

## 9.5.3 缺点

因此，这些模式的某些问题实际上源于它的主要好处。在 Publish/Subscribe 中，通过从订阅者中解耦发布者，它有时会很难保证应用程序的特定部分按照我们期望的运行。

例如，发布者可能会假设：一个或多个订阅者在监听它们。倘若我们假设订阅者需要记录或输出一些与应用程序处理有关的错误。如果订阅者执行日志崩溃了（或出于某种原因无法正常运行），由于系统的解耦特性，发布者就不会看到这一点。

这种模式的另一个缺点是：订阅者非常无视彼此的存在，并对变换发布者产生的成本视而不见。由于订阅者和发布者之间的动态关系，很难跟踪依赖更新。

## 9.5.4 Publish/Subscribe 实现

Publish/Subscribe 非常适用于 JavaScript 生态系统，这主要是因为在其核心，ECMAScript 实现是由事件驱动的。在浏览器环境下尤其如此，因为 DOM 将事件是作为脚本编程的主要交互 API。

也就是说，在实现代码里，无论是 ECMAScript 还是 DOM 都不会提供核心对象或

方法来创建自定义事件系统（或许除了 DOM3 CustomEvent 以外，它被绑定到 DOM，因此一般是无用的）。

幸运的是，流行的 JavaScript 库，比如 Dojo、jQuery（自定义事件）和 YUI 都拥有一些实用程序可以很容易实现 Publish/Subscribe 系统。如下是一些有关示例：

```
// Publish

// jQuery: $(obj).trigger("channel", [arg1, arg2, arg3]);
$( el ).trigger( "/login", [{username:"test", userData:"test"}] );

// Dojo: dojo.publish("channel", [arg1, arg2, arg3] );
dojo.publish( "/login", [{username:"test", userData:"test"}] );

// YUI: el.publish("channel", [arg1, arg2, arg3]);
el.publish( "/login", {username:"test", userData:"test"} );

// Subscribe

// jQuery: $(obj).on( "channel", [data], fn );
$( el ).on( "/login", function( event ){...} );

// Dojo: dojo.subscribe( "channel", fn );
var handle = dojo.subscribe( "/login", function(data){..} );

// YUI: el.on("channel", handler);
el.on( "/login", function( data ){...} );

// Unsubscribe

// jQuery: $(obj).off( "channel" );
$( el ).off( "/login" );

// Dojo: dojo.unsubscribe( handle );
dojo.unsubscribe( handle );

// YUI: el.detach("channel");
el.detach( "/login" );
```

对于那些希望使用采用纯 JavaScript（或其他库）的 Publish/Subscribe 模式的人来说，AmplifyJS (<http://amplifyjs.com/>) 包含了一个整洁、与库无关的实现，它可用于任何库或工具包。值得一看的类似语言有 Radio.js (<http://radio.uxder.com/>)、PubSubJS (<https://github.com/mroderick/PubSubJS>)、或 Peter Higgins 所写的 Pure JS PubSub (<https://github.com/phiggins42/bloody-jquery-plugins/blob/>)。

jQuery 开发人员更是有相当多的其他选择，可以选择使用众多完整实现中的一个，

从 Peter Higgins 的 jQuery 插件到 Ben Alman 在 GitHub 上的优化过的 Pub/Sub jQuerygist。如下仅列出几个相关链接。

- Ben Alman 的 Pub/Subgist

<https://gist.github.com/661855> (recommended) (推荐)

- Rick Waldron 上述所说的 jQuery 核心风格

<https://gist.github.com/705311>

- Peter Higgins 的插件

<http://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsu.js>

- AppendTo 在 AmplifyJS 里实现的 Pub/Sub

<http://amplifyjs.com>

- Ben Truyman 的 gist

<https://gist.github.com/826794>

所以我们现在能够正确了解有多少个纯 JavaScript 实现的 Observer 模式了，让我们来看一下在 GitHub 上发布的一个被称为 pubsubz (<https://github.com/addyosmani/pubsubz>) 的项目中一个极简版本的 Publish/Subscribe I。它展示了订阅和发布的概念，以及取消订阅的概念。

我选择了在这个代码的基础上展示我们的示例，因为它非常接近我们所期望的 JavaScript 版经典 Observer 模式所包括的方法命名和实现方式。

#### 9.5.4.1 Publish/Subscribe 实现

```
var pubsub = {};  
function (q) {  
    var topics = {},  
        subUid = -1;
```

```

// 发布或广播事件，包含特定的 topic 名称和参数（比如传递的数据）
q.publish = function (topic, args) {

    if (!topics[topic]) {
        return false;
    }

    var subscribers = topics[topic],
        len = subscribers ? subscribers.length : 0;

    while (len--) {
        subscribers[len].func(topic, args);
    }

    return this;
};

// 通过特定的名称和回调函数订阅事件，topic/event 触发时执行事件
q.subscribe = function (topic, func) {
    if (!topics[topic]) {
        topics[topic] = [];
    }

    var token = (++subUid).toString();
    topics[topic].push({
        token: token,
        func: func
    });
    return token;
};

// 基于订阅上的标记引用，通过特定 topic 取消订阅
q.unsubscribe = function (token) {
    for (var m in topics) {
        if (topics[m]) {
            for (var i = 0, j = topics[m].length; i < j; i++) {
                if (topics[m][i].token === token) {
                    topics[m].splice(i, 1);
                    return token;
                }
            }
        }
    }
    return this;
};
} (pubsub));

```

#### 9.5.4.2 使用上述实现

我们现在可以使用该实现来发布和订阅感兴趣的活动，如下所示（示例 9-1）：

### 示例 9-1 使用上述实现

```
// 另一个简单的消息处理程序

// 简单的消息记录器记录所有通过订阅者接收到的主题 (topic) 和数据

var messageLogger = function ( topics, data ) {
    console.log( "Logging: " + topics + ": " + data );
};

// 订阅者监听订阅的 topic，一旦该 topic 广播一个通知，订阅者就调用回调函数
var subscription = pubsub.subscribe( "inbox/newMessage", messageLogger );

// 发布者负责发布程序感兴趣的 topic 或通知，例如：

pubsub.publish( "inbox/newMessage", "hello world!" );

// 或者
pubsub.publish( "inbox/newMessage", [ "test", "a", "b", "c" ] );

// 或者
pubsub.publish( "inbox/newMessage", {
    sender: "hello@google.com",
    body: "Hey again!"
} );

// 如果订阅者不想被通知了，也可以取消订阅
// 一旦取消订阅，下面的代码执行后将不会记录消息，因为订阅者不再进行监听了
pubsub.publish( "inbox/newMessage", "Hello! are you still there?" );
```

#### 9.5.4.3 用户界面通知

接下来，假设我们有一个负责显示实时股票信息的 Web 应用程序。

该应用程序有一个显示股票统计的网格和一个显示最后更新点的计数器。当数据模型改变时，应用程序需要更新网格和计数器。在这种情况下，目标（它将发布主题/通知）就是数据模型，观察者就是网格和计数器。

当观察者接收到 Model（模型）自身已经改变的通知时，则可以相应地更新自己。

在我们的实现中，订阅者会监听 `newDataAvailable` 这个 topic，以探测是否有新的股票信息。如果新通知发布到这个 topic，它将触发 `gridUpdate` 向包含股票信息的网格添加一个新行。它还将更新一个 `last updated` 计数器来记录最后一次添加的数据（示例 9-2）。

## 示例 9-2 用户界面通知

```
// 在 newDataAvailable topic 上创建一个订阅
var subscriber = pubsub.subscribe( "newDataAvailable", gridUpdate );

// 返回稍后界面上要用到的当前本地时间
getCurrentTime = function (){

    var date = new Date(),
        m = date.getMonth() + 1,
        d = date.getDate(),
        y = date.getFullYear(),
        t = date.toLocaleTimeString().toLowerCase();

    return (m + "/" + d + "/" + y + " " + t);
};

// 向网格组件上添加新数据行
function addGridRow( data ) {

    // ui.grid.addRow( data );
    console.log( "updated grid component with:" + data );
}

// 更新网格上的最新更新时间
function updateCounter( data ) {
    // ui.grid.updateLastChanged( getCurrentTime() );
    console.log( "data last updated at: " + getCurrentTime() + " with "
+ data );
}

// 使用传递给订阅者的数据 data 更新网格
gridUpdate = function( topic, data ){

    if ( data !== "undefined" ) {
        grid.addGridRow( data );
        grid.updateCounter( data );
    }

};

// 下面的代码描绘了数据层，一般应该使用 ajax 请求获取最新的数据后，告知程序有最新数据
// 发布者更新 gridUpdate topic 来展示新数据项
pubsub.publish( "newDataAvailable", {
    summary: "Apple made $5 billion",
    identifier: "APPL",
    stockPrice: 570.91
});

pubsub.publish( "newDataAvailable", {
    summary: "Microsoft made $20 million",
    identifier: "MSFT",
    stockPrice: 30.85
});
```

#### 9.5.4.4 使用 Ben Alman 的 Pub/Sub 实现解耦应用程序

在接下来的电影评级示例中，我们将使用 Ben Alman 在 Publish/Subscribe 模式上的 jQuery 实现来展示我们如何解耦一个用户界面。需要注意的是，如何提交评级才会有新用户和评级数据同时发布通知的效果。

这是留给这些 topic 的订阅者来处理那些数据的。在我们的例子中，将新数据放入现有的数组中，然后使用 Underscore 库的`.template()`方法使用模板呈现它们。

如下是 HTML/模板代码（示例 9-3）：

示例 9-3 用于 Pub/Sub 的 HTML/模板代码

```
<script id="userTemplate" type="text/html">
  <li><%= name %></li>
</script>

<script id="ratingsTemplate" type="text/html">
  <li><strong><%= title %></strong> was rated <%= rating %>/5</li>
</script>

<div id="container">

  <div class="sampleForm">
    <p>
      <label for="twitter_handle">Twitter handle:</label>
      <input type="text" id="twitter_handle" />
    </p>
    <p>
      <label for="movie_seen">Name a movie you've seen this year:</label>
      <input type="text" id="movie_seen" />
    </p>
    <p>
      <label for="movie_rating">Rate the movie you saw:</label>
      <select id="movie_rating">
        <option value="1">1</option>
        <option value="2">2</option>
        <option value="3">3</option>
        <option value="4">4</option>
        <option value="5" selected>5</option>
      </select>
    </p>
    <p>
      <button id="add">Submit rating</button>
    </p>
  </div>
</div>
```

```
        </p>
    </div>

    <div class="summaryTable">
        <div id="users"><h3>Recent users</h3></div>
        <div id="ratings"><h3>Recent movies rated</h3></div>
    </div>
</div>
```

下面是 JavaScript 代码（示例 9-4）：

#### 示例 9-4 用于 Pub/Sub 的 JavaScript 代码

```
; (function ($) {

    // 订阅 new user 主题，提交评论的时候在用户列表上添加一个用户
    $.subscribe("/new/user", function (e, data) {

        var compiledTemplate;

        if (data) {
            compiledTemplate = _.template($("#userTemplate").html());
            $('#users').append(compiledTemplate(data));
        }

    });

    // 订阅 new rating 主题，rating 主题由 title 和 rating 组成。新 rating 添加到已有
    // 用户的 rating 列表上
    $.subscribe("/new/rating", function (e, data) {

        var compiledTemplate;

        if (data) {

            compiledTemplate = _.template($("#ratingsTemplate").html());
            $("#ratings").append(compiledTemplate(data));
        }

    });

    // 添加新用户处理程序
    $("#add").on("click", function (e) {

        e.preventDefault();

        var strUser = $("#twitter_handle").val(),
            strMovie = $("#movie_seen").val(),
            strRating = $("#movie_rating").val();

        // 通知程序，新用户有效
        $.publish("/new/user", { name: strUser });

    });

});
```

```
// 通知程序新 rating 评价有效
$.publish("/new/rating", { title: strMovie, rating: strRating });
});

})(jQuery);
```

#### 9.5.4.5 解耦基于 Ajax 的 jQuery 应用程序

在最后一个示例中，我们将看一下如何使用 Pub/Sub 解耦在早期开发过程中的代码，以此使我们省去一些可能繁琐的重构工作。

通常在侧重 Ajax 技术的应用程序中，一旦我们收到了请求的响应，我们就想据此实现不只一个特定动作。我们可以简单地向成功回调中添加所有的 post 请求逻辑，但这种方法存在一些缺点。

由于函数/代码之间互相依赖的增加，高度耦合的应用程序有时会增加复用函数所需的工作量。这意味着，如果我们只是想一次性获取一个结果集，在回调中对 post 请求逻辑进行硬编码可能是行得通的，但是，当我们需要对相同的数据源（和不同的端行为）进一步地进行 Ajax 调用，而没有多次重写部分代码，那就不那么合适了。我们可以从一开始就使用 pub/sub 来节省时间，而不必遍历调用相同数据源的每一层而后再对它们进行操作。

通过使用 **Observer**，我们还可以将不同事件降至我们所要的任何粒度级别，并轻松地根据这些事件分离应用程序范围内的通知，而使用其他模式完成这项工作的优雅度较低。

请注意在下面的样例中，当用户表示他想进行搜索查询时，是如何发出一个 topic 通知的，以及当请求返回并且有实际数据可用时，是如何发出另一个通知的。它让订阅者随后决定如何利用这些事件（或返回的数据）。它的好处是：如果我们愿意，我们可以有 10 个不同的订阅者以不同的方式使用返回的数据，但这对于 Ajax 层而言是无关紧要的。其唯一的责任是请求和返回数据，然后传递给任何想使用它的人。这种关注点分离能使代码的整体设计变得更加整洁。

下面是 HTML/模板代码（示例 9-5）：

#### 示例 9-5 用于 Ajax 的 HTML/模板

```
<form id="flickrSearch">
```

```
<input type="text" name="tag" id="query"/>

<input type="submit' name="submit' value="submit' />

</form>

<div id="lastQuery"></div>

<div id="searchResults"></div>

<script id="resultTemplate" type="text/html">
  <% _.each(items, function( item ){ %>
    <li><p></p></li>
  <% });%>
</script>
```

下面是 JavaScript 代码（示例 9-6）：

#### 示例 9-6 用于 Ajax 的 JavaScript 代码

```
; (function ($) {
  // 预编译模板，并使用闭包缓存它
  var resultTemplate = _.template($("#resultTemplate").html());

  // 订阅新搜索 tags 主题
  $.subscribe("/search/tags", function (e,tags) {
    $("#searchResults")
      .html("Searched for:" + tags + "");
  });

  // 订阅新搜索结果主题
  $.subscribe("/search/resultSet", function (e,results) {
    $("#searchResults").append(resultTemplate(results));
    $("#searchResults").append(compiled_template(results));
  });

  // 提交搜索请求，并在/search/tags 主题上发布 tags
  $("#flickrSearch").submit(function (e) {

    e.preventDefault();
    var tags = $(this).find("#query").val();

    if (!tags) {
      return;
    }
  });
});
```

```

    $.publish("/search/tags", [$._trim(tags)]);
});

// 订阅发布的新 tag，并且使用 tag 发起请求。一旦返回数据，将数据发布给应用程序的其他使用者
$.subscribe("/search/tags", function (e, tags) {

    $.getJSON("http://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=
    ck=?", {
        tags: tags,
        tagmode: "any",
        format: "json"
    },
    function (data) {

        if (!data.items.length) {
            return;
        }

        $.publish("/search/resultSet", data.items);
    });
});

}) (jaucry);

```

在应用程序设计中，Observer 模式在解耦多个不同脚本方面是非常有用的，如果你还没有使用它，我建议你了解一下这里提到的其中一个预先编写的实现，并试着使用一下。这是要入门了解的一个比较简单的设计模式，但同时也是最强大的设计模式之一。

## 9.6 Mediator（中介者）模式

在字典里，中介者是指“协助谈判和解决冲突的中立方”<sup>1</sup>。在本书设计模式里，中介者是一种行为设计模式，它允许我们公开一个统一的接口，系统的不同部分可以通过该接口进行通信。

如果一个系统的各个组件之间看起来有太多的直接关系，也许是时候需要一个中心控制点了，以便各个组件可以通过这个中心控制点进行通信。Mediator 模式促进松散耦合的方式是：确保组件的交互是通过这个中心点来处理的，而不是通过显式地

<sup>1</sup> Wikipedia (<http://en.wikipedia.org/wiki/Mediation>) ; Dictionary.com (<http://dictionary.reference.com/browse/mediator>)

引用彼此。这种模式可以帮助我们解耦系统并提高组件的可重用性。

现实世界的一个例子是典型的机场交通控制系统。机场控制塔（中介者）处理飞机的起飞和降落，因为所有通信（监听到或发出的通知）都是从飞机到控制塔，而不是飞机和飞机直接相互通信的。中央控制系统是该系统成功的关键，而这才是中介者在软件设计中所担任的真正角色（图 9-5）。

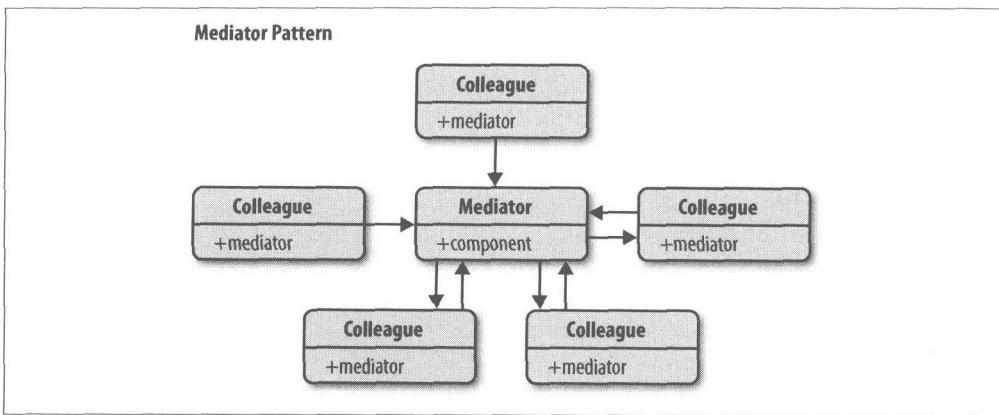


图 9-5 Mediator 模式

就实现而言，Mediator 模式本质上是 Observer 模式的共享目标。它假设该系统中对象或模块之间的订阅和发布关系被牺牲掉了，从而维护中心联络点。

它也可能被认为是额外的或者是用于应用程序间的通知，如不同子系统之间的通信，这些子系统本身就很复杂，且可能希望通过发布/订阅关系实现内部组件间的解耦。

另一个例子是 DOM 事件冒泡和事件委托。如果系统中所有的订阅针对的是文档 `document` 而不是单个 `node` 节点，则这个文档会有效地充当中介者。更高级别(`level`)的对象承担了向订阅者通知有关交互事件的责任，而不是绑定到单个节点的事件。

### 9.6.1 基本实现

可以在下面找到 Mediator 模式的简单实现，暴露了 `publish()` 和 `subscribe()` 方法来使用：

```
var mediator = (function () {  
    // 存储可被广播或监听的 topic  
    var topics = {};  
    ...  
});
```

```

// 订阅一个 topic，提供一个回调函数，一旦 topic 被广播就执行该回调
var subscribe = function (topic, fn) {

    if (!topics[topic]){
        topics[topic] = [];
    }
    topics[topic].push({ context: this, callback: fn });

    return this;
};

// 发布/广播事件到程序的剩余部分
var publish = function (topic){

    var args;

    if (!topics[topic]){
        return false;
    }

    args = Array.prototype.slice.call(arguments, 1);
    for (var i = 0, l = topics[topic].length; i < l; i++) {

        var subscription = topics[topic][i];
        subscription.callback.apply(subscription.context, args);
    }
    return this;
};

return {
    Publish: publish,
    Subscribe: subscribe,
    installTo: function (obj) {
        obj.subscribe = subscribe;
        obj.publish = publish;
    }
};
})();

```

## 9.6.2 高级实现

如果你对更高级的代码实现感兴趣，深入下去可以浏览到 Jack Lawson 的优秀 `Mediator.js` (<http://thejacklawson.com/Mediator.js/>) 的简洁版。除了其他改进以外，这个版本支持 topic 命名空间、订阅者删除和用于中介者的更强大的发布/订阅（Publish/Subscribe）系统。但如果你想跳过这些内容，则可以直接进入下一个示例

继续阅读。<sup>1</sup>

首先，让我们来实现订阅者的概念，可以考虑一个 Mediator 的 topic 注册实例。

通过生成对象实例，之后我们可以很容易地更新订阅者，而不需要注销并重新注册它们。订阅者可以写成构造函数，该函数接受三个参数：一个可被调用的函数 fn、一个 options 对象和一个 context（上下文）。

```
// 将 context 上下文传递给订阅者，默认上下文是 window 对象
(function (root){

    function guidGenerator() { /*..*/ }

    // 订阅者构造函数
    function Subscriber(fn, options, context) {

        if (!this instanceof Subscriber) {

            return new Subscriber(fn, context, options);

        } else {

            // guidGenerator() 是一个函数，用于为订阅者生成 GUID，以便之后很方便地引用它们。
            // 为了简洁，跳过具体实现

            this.id = guidGenerator();
            this.fn = fn;
            this.options = options;
            this.context = context;
            this.topic = null;

        }
    }
})();
```

Mediator 中的 topic 持有了一组回调函数和子 topic 列表，一旦 Mediator.Publish 方法在 Mediator 实例上被调用时，这些回调函数就会被触发。它还包含用于操作数据列表的方法。

```
// 模拟 Topic
// JavaScript 允许我们使用 Function 对象作为原型的结合与新对象和构造函数一起调用
function Topic( namespace ){

    if ( !this instanceof Topic ) {
```

---

<sup>1</sup> 感谢 Jack 提供优秀代码注释，在本节中提供了莫大的帮助。

```

        return new Topic( namespace );
    }else{

        this.namespace = namespace || "";
        this._callbacks = [];
        this._topics = [];
        this.stopped = false;

    }
}

// 定义 topic 的 prototype 原型，包括添加订阅者和获取订阅者的方式
Topic.prototype = {

    // 添加新订阅者
    AddSubscriber: function( fn, options, context ){

        var callback = new Subscriber( fn, options, context );

        this._callbacks.push( callback );

        callback.topic = this;

        return callback;
    },
    ...
}

```

我们的 Topic 实例作为一个参数传递给 Mediator 回调。然后可以 StopPropagation() 的简便方法来调用进一步的回调传播：

```

StopPropagation: function(){
    this.stopped = true;
},

```

当给定 GUID 标识符时，我们也可以很容易获取现有的订阅者：

```

GetSubscriber: function( identifier ){

    for(var x = 0, y = this._callbacks.length; x < y; x++ ){
        if( this._callbacks[x].id == identifier || this._callbacks[x].fn == identifier ){
            return this._callbacks[x];
        }
    }

    for( var z in this._topics ){
        if( this._topics.hasOwnProperty( z ) ){
            var sub = this._topics[z].GetSubscriber( identifier );
            if( sub !== undefined ){
                return sub;
            }
        }
    }
}

```

```
        }
    }
}

},
```

接下来，如果需要它们，我们可以提供简单方法来添加新 topic、检查现有 topic 或者获取 topic：

```
AddTopic: function( topic ){
    this._topics[topic] = new Topic( (this.namespace ? this.namespace + ":" :
 "") + topic );
},

HasTopic: function( topic ){
    return this._topics.hasOwnProperty( topic );
},

returnTopic: function( topic ){
    Return this._topics[topic];
},
```

如果不再需要订阅者，我们可以显式地删除它们。以下代码将通过它的子主题递归删除一位订阅者：

```
RemoveSubscriber: function( identifier ){

    if( !identifier ){
        this._callbacks = [];

        for( var z in this._topics ){
            if( this._topics.hasOwnProperty(z) ){
                this._topics[z].RemoveSubscriber( identifier );
            }
        }
    }

    for( var y = 0, x = this._callbacks.length; y < x; y++ ) {
        if( this._callbacks[y].fn == identifier || this._callbacks[y].id ==
 identifier ){
            this._callbacks[y].topic = null;
            this._callbacks.splice( y,1 );
            x--; y--;
        }
    }
},
```

接下来，我们将通过子 topic 递归向订阅者发布（Publish）任意参数。

```
Publish: function( data ){

    for( var y = 0, x = this._callbacks.length; y < x; y++ ) {

        var callback = this._callbacks[y], l;
        callback.fn.apply( callback.context, data );

        l = this._callbacks.length;

        if( l < x ) {
            y--;
            x = l;
        }
    }

    for( var x in this._topics ){
        if( !this.stopped ){
            if( this._topics.hasOwnProperty( x ) ){
                this._topics[x].Publish( data );
            }
        }
    }

    this.stopped = false;
}
};
```

这里暴露了我们将主要与之交互的 Mediator 实例。在这里，完成了事件在 topic 上的注册和移除。

```
function Mediator() {

    if ( !this instanceof Mediator) {
        return new Mediator();
    } else {
        this._topics = new Topic("");
    }
};
```

对于更高级的使用场景，我们可以让 Mediator 支持用于 inbox:messages:new:read 等主题 topic 的命名空间。在接下来的示例中，GetTopic 根据命名空间返回相应的主题实例。

```
Mediator.prototype = {

    GetTopic: function( namespace ){


```

```

var topic = this._topics,
    namespaceHierarchy = namespace.split( ":" );

if( namespace === "" ){
    return topic;
}

if( namespaceHierarchy.length > 0 ){
    for( var i = 0, j = namespaceHierarchy.length; i < j; i++ ){

        if( !topic.HasTopic( namespaceHierarchy[i] ) ){
            topic.AddTopic( namespaceHierarchy[i] );
        }

        topic = topic.ReturnTopic( namespaceHierarchy[i] );
    }
}

return topic;
},

```

在本小节中，我们定义了 `Mediator.Subscribe` 方法，它接受一个 `topic` 命名空间、一个可执行的 `fn` 函数、`options`，以及调用该函数的 `context` 上下文。如果 `topic` 不存在，则创建一个。

```

Subscribe: function( topicName, fn, options, context ){
    var options = options || {},
        context = context || {},
        topic = this.GetTopic( topicName ),
        sub = topic.AddSubscriber( fn, options, context );

    return sub;
},

```

继续下去，我们可以进一步定义用于访问特定订阅者或将订阅者从 `topic` 中递归删除的实用程序。

```

// 通过给定的订阅者 ID/命名函数和 topic 命名空间返回一个订阅者

GetSubscriber: function( identifier, topic ){
    return this.GetTopic( topic || "" ).GetSubscriber( identifier );
},

// 通过给定的订阅者 ID 或命名函数，从给定的 topic 命名空间递归删除订阅者

Remove: function( topicName, identifier ){
    this.GetTopic( topicName ).RemoveSubscriber( identifier );
},

```

主要的 Publish 方法允许我们向所选择的 topic 命名空间任意发布数据。

Topic 向下递归调用。例如，一个发往 inbox:messages 的帖子将被发至 inbox:messages:new 和 inbox:messages:new:read。如下所示：

```
Mediator.Publish( "inbox:messages:new", [args] );
Publish: function( topicName ){
    var args = Array.prototype.slice.call( arguments, 1 ),
        topic = this.GetTopic( topicName );

    args.push( topic );
    this.GetTopic( topicName ).Publish( args );
}
};
```

最后，我们可以很容易地将 Mediator 作为一个对象附加到 root 上：

```
root.Mediator = Mediator;
Mediator.Topic = Topic;
Mediator.Subscriber = Subscriber;

// 记住，这里可以传递任何内容。这里我传递了 window 对象作为 Mediator 的附加对象，但
// 也可以随时附加到其他对象上。
})( window );
```

### 9.6.3 示例

通过使用上述的任一种实现（简单的和高级的实现），我们可以建立一个简单的聊天记录系统，如下所示。

如下是 HTML 代码：

```
<h1>Chat</h1>
<form id="chatForm">
    <label for="fromBox">Your Name:</label>
    <input id="fromBox" type="text"/>
    <br />
    <label for="toBox">Send to:</label>
    <input id="toBox" type="text"/>
    <br />
    <label for="chatBox">Message:</label>
    <input id="chatBox" type="text"/>
    <button action="submit">Chat</button>
</form>

<div id="chatResult"></div>
```

如下是 JavaScript 代码：

```
$("#chatForm").on("submit", function (e) {
    e.preventDefault();

    // 从 UI 上获取 chat 的数据
    var text = $("#chatBox").val(),
        from = $("#fromBox").val();
        to = $("#toBox").val();

    // 将数据发布到 newMessage 主题上
    mediator.publish("newMessage", { message: text, from: from, to: to });

    // 将新消息附加到聊天结果记录上
    function displayChat(data) {
        var date = new Date(),
            msg = data.from + " said \"" + data.message + "\" to " + data.to;

        $("#chatResult")
            .prepend(msg + " (" + date.toLocaleTimeString() + ")");
    }

    // 记录消息日志
    function logChat(data) {
        if (window.console) {
            console.log(data);
        }
    }

    // 通过 mediator 订阅新提交的 newMessage 主题
    mediator.subscribe("newMessage", displayChat);
    mediator.subscribe("newMessage", logChat);

    // 如下代码仅在高级代码实现上可以使用

    function amITalkingToMyself(data) {
        return data.from === data.to;
    }

    function iAmClearlyCrazy(data) {
        $("#chatResult").prepend(data.from + " is talking to himself.");
    }

    mediator.Subscribe(amITalkingToMyself, iAmClearlyCrazy);
}
```

## 9.6.4 优点和缺点

Mediator 模式最大的好处是：它能够将系统中对象或组件之间所需的通信渠道从多对多减少到多对一。由于现在的解耦程度较高，添加新发布者和订阅者相对也容易多了。

或许使用这种模式最大的缺点是：它会引入单一故障点。将 Mediator 放置于模块之间可以导致性能下降，因为它们总是间接地进行通信。由于松耦合的性质，很难通过仅关注广播来确定一个系统如何作出反应。

也就是说，自我提醒解耦的系统有很多其他的优点：如果模块之间直接相互通信，模块的改变（如另一个模块抛出一个异常）容易让应用程序的其余部分产生多米诺效应。这个问题对解耦的系统来说就不是个大问题。

最后，紧密耦合会引起各种各样的问题，这只是另一个替代方案，但如果正确地实现，它也能很好地工作。

### 9.6.5 中介者 (Mediator) 与观察者 (Observer)

开发人员通常想知道 Mediator 模式和 Observer 模式之间的差异是什么。无可否认地，它们之间有一些重叠，让我们重新参考“四人组”作出的解释：



在 Observer 模式中，不存在封装约束的单一对象。Observer 和 Subject（目标）必须合作才能维持约束。Communication（通信）模式由观察者和目标互连的方式所决定：单一目标通常有很多观察者，有时一个目标的观察者是另一个观察者的目标。

Mediator 和 Observer 都能够促进松耦合；然而，Mediator 模式通过限制对象严格通过 Mediator 进行通信来实现这一目的。Observer 模式创建观察者对象，观察者对象向订阅它们的对象发布其感兴趣的事件。

### 9.6.6 中介者 (Mediator) 与外观 (Facade)

我们将简单提一下 Facade 模式，但出于引用目的，一些开发人员可能也想知道 Mediator 和 Facade 模式之间是否有相似点。它们都能够抽象出现有模块的功能，但是也有一些细微的差别。

Mediator 模块在它被模块显式引用的地方汇集这些模块之间的通信。从某种意义上说，这是多方向的。另一方面，Facade 模式仅是为模块或系统定义了一个较简单的接口，而没有添加任何额外的功能。系统中的其他模块不会直接关联外观，所以可以被视为单向的。

## 9.7 Prototype（原型）模式

“四人组”称 Prototype 模式为一种基于现有对象模板，通过克隆方式创建对象的模式。

我们可以认为 Prototype 模式是基于原型继承的模式，可以在其中创建对象，作为其他对象的原型。prototype 对象本身实际上是用作构造函数创建每个对象的蓝图。如果所用构造函数的原型包含一个名为 name 的属性（代码样例如下），那么由同样的构造函数创建的每个对象也会有同样的属性（见图 9-6）。

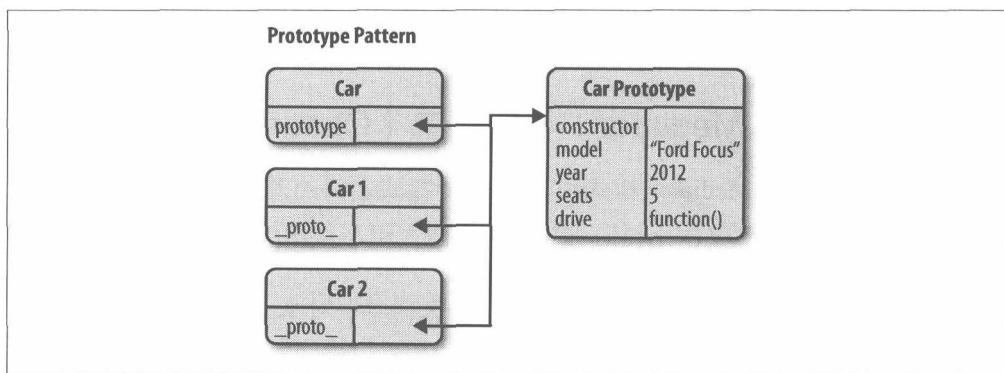


图 9-6 Prototype 模式

如果查看现有（非 JavaScript）文献对该模式的定义，我们可能会再一次发现对类的引用。现实情况是，原型继承避免和类（Class）一起使用。理论上没有“定义”的对象，也没有核心的对象。我们仅是创建现有功能对象的拷贝。

使用 Prototype 模式的其中一个好处是，我们获得的是 JavaScript 其本身所具有的原型优势，而不是试图模仿其他语言的特性。与其他设计模式一起使用时，情况并非总是如此。

模式不仅是一种实现继承的简单方法，它也可以带来性能提升：在一个对象中定义一个函数，它们都是由引用创建（因此所有子对象都指向相同的函数），而不是创建它们自己的单份拷贝。

如同 ECMAScript5 标准所定义的，那些有趣的、真正的原型继承要求使用 Object.create（我们在本节的前面曾看到过）。Object.create 创建一个对象，拥有指定原型和可选的属性（例如 Object.create（prototype, optionalDescriptorObjects））。

在下面的示例中可以看到它的演示：

```
var myCar = {  
    name: "Ford Escort",  
  
    drive: function () {  
        console.log("Weeeee. I'm driving!");  
    },  
  
    panic: function () {  
        console.log("Wait. How do you stop this thing?");  
    }  
  
};  
  
// 使用 Object.create 实例化一个新 car  
var yourCar = Object.create(myCar);  
  
// 现在可以看到一个对象是另外一个对象的原型  
console.log(yourCar.name);
```

Object.create 还允许我们轻松实现差异继承等高级概念，通过差异继承，对象可以直接继承自其他对象。我们之前已经了解到，Object.create 允许我们使用第二个提供的参数来初始化对象属性。例如：

```
var vehicle = {  
    getModel: function () {  
        console.log("The model of this vehicle is.." + this.model);  
    }  
};  
  
var car = Object.create(vehicle, {  
  
    "id": {  
        value: MY_GLOBAL.nextId(),  
        // writable:false, configurable:false 默认值  
        enumerable: true  
    },  
  
    "model": {  
        value: "Ford",  
        enumerable: true  
    }  
});
```

在这里，可以使用对象字面量在 Object.create 的第二个参数上初始化属性，并且对

象字面量采用的语法与 `Object.defineProperties` 和 `Object.defineProperty` 方法所使用的语法相似，我们之前已经了解过这些方法。

值得注意的是，在枚举对象的属性以及在 `hasOwnProperty()` 检查中包装循环内容时（Crockford 推荐），原型关系可能会引起麻烦。

如果希望在不直接使用 `Object.create` 的情况下实现 Prototype 模式，我们可以按照上面的示例模拟该模式，如下所示：

```
var vehiclePrototype = {

    init: function (carModel) {
        this.model = carModel;
    },

    getModel: function () {
        console.log("The model of this vehicle is.." + this.model);
    }
};

function vehicle(model) {

    function F() { };
    F.prototype = vehiclePrototype;

    var f = new F();
    f.init(model);
    return f;
}

var car = vehicle("Ford Escort");
car.getModel();
```



这个方案不允许用户以同样的方式定义只读属性（因为如果不小心，`vehiclePrototype` 可能会被改变）。

最后一种可供选择的 Prototype 模式实现可以是这样的：

```
var beget = (function () {

    function F() { }
    return function (proto) {
        F.prototype = proto;
        return new F();
    }
});
```

```
};  
})();
```

我们可以从 vehicle 函数引出这个方法。请注意，这里的 vehicle 模仿了一个构造函数，因为 Prototype 模式不包含任何初始化的概念，而不仅是将对象链接至原型。

## 9.8 Command（命令）模式

Command 模式旨在将方法调用、请求或操作封装到单一对象中，从而根据我们不同的请求对客户进行参数化和传递可供执行的方法调用。此外，这种模式将调用操作的对象与知道如何实现该操作的对象解耦，并在交换出具体类（对象）方面提供更大的整体灵活性。

用基于类的编程语言解释具体类是最恰当的，它们与抽象类的思想相关。一个抽象类定义一个接口，但不一定为它所有的成员函数提供实现。它作为一个基类，派生出其他类。实现缺失功能的派生类被称为一个具体的类（见图 9-7）。

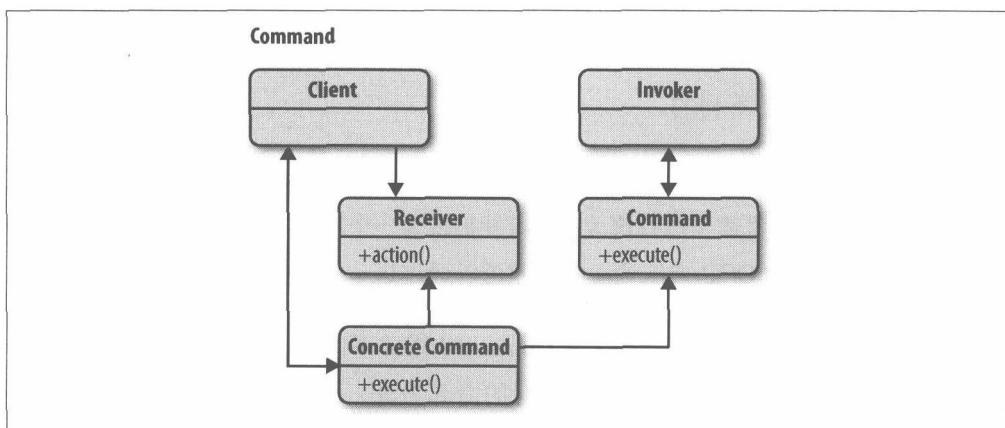


图 9-7 Command 模式

Command 模式背后的主要思想是：它为我们提供了一种分离职责的手段，这些职责包括从执行命令的任意地方发布命令以及将该职责转而委托给不同对象。

实施明智的、简单的命令对象将把 action 动作和调用该动作的对象绑定在一起。它们始终包括一个执行操作（如 run() 或 execute()）。所有具有相同接口的 Command 对象可以根据需要轻松交换，这被认为是该模式的一个更大好处。

为了演示 Command 模式，我们将创建一个简单的汽车购买服务。

```
(function () {
    var CarManager = {
        // 请求信息
        requestInfo: function (model, id) {
            return "The information for " + model + " with ID " + id + " is foobar";
        },
        // 订购汽车
        buyVehicle: function (model, id) {
            return "You have successfully purchased Item " + id + ", a " + model;
        },
        // 组织一个 view
        arrangeViewing: function (model, id) {
            return "You have successfully booked a viewing of " + model + "(" + id + ")";
        }
    };
})();
```

看一下上面的代码，它可以通过直接访问对象轻松地调用我们的 CarManager 方法。我们认为这里没有任何错误，从技术上讲，它是完全有效的 JavaScript。然而，它在有些情况下可能是不利的。

例如，试想如果 CarManager 里的核心 API 改变了会怎么样。这将要求程序里所有直接访问这些方法的对象都需要进行修改。这可能被视为一个耦合层，它实际上最大程度地违反了松耦合对象的 OOP 方法论。而我们可以通过进一步抽象 API 来解决这个问题。

现在让我们来扩展 CarManager，这样 Command 模式下的应用程序会产生如下结果：接受任意可以在 CarManager 对象上执行的命名方法，传递可以使用的任意数据，如 CarModel（模型）和 ID。

这是我们希望能够实现的内容：

```
CarManager.execute("buyVehicle", "Ford Escort", "453543");
```

按照这个结构，我们现在应该为 CarManager.execute 方法添加一个定义，如下所示：

```
CarManager.execute = function (name) {
    return CarManager[name] && CarManager[name].apply(CarManager,
        [].slice.call(arguments, 1));
};
```

因此我们最终的示例调用看起来是这样的：

```
CarManager.execute("arrangeViewing", "Ferrari", "14523");
CarManager.execute("requestInfo", "Ford Mondeo", "54323");
CarManager.execute("requestInfo", "Ford Escort", "34232");
CarManager.execute("buyVehicle", "Ford Escort", "34232");
```

## 9.9 Facade（外观）模式

当创建外观时，向外界展示的外表可能掩盖了一个非常不同的现实。这是我们下一个要查看的模式名称的灵感来源——Facade 模式。Facade 模式为更大的代码体提供了一个方便的高层次接口，能够隐藏其底层的真实复杂性。可以把它想成是简化 API 来展示给其他开发人员，通常都是可以提高可用性（见图 9-8）。

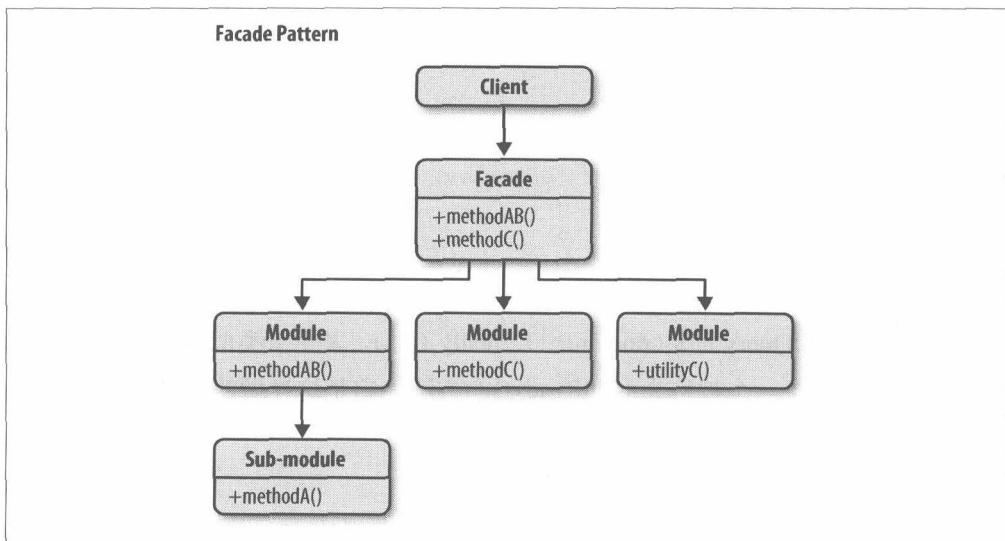


图 9-8 Facade 模式

Facade 是一种结构型模式，在 jQuery 等 JavaScript 库中经常可以看到，尽管一个实现可能支持具有广泛行为的方法，但却只有一个“外观”或这些方法的有限抽象能够提供给公众使用。

这使我们可以直接与 Facade 交互，而不是与幕后子系统交互。每当使用 jQuery 的 \$(el).css() 或 \$(el).animate() 方法时，实际上我们是在使用 Facade：一种更简单的公有接口，使我们不必手动在 jQuery 核心中调用很多内部方法以便实现某些行

为。这也避免了手动与 DOM API 交互并维护状态变量的需要。

jQuery 核心方法应该被认为是指间抽象。对于开发人员来说，更直接的事是 DOM API，外观可以使 jQuery 库很容易使用。

在我们学到的知识基础上，Facade 模式既能简化类的接口，也能将这个类从使用它的代码中解耦。这使我们能够间接与子系统交互，这种方式相比直接访问子系统有时不易犯错误。Facade 的优点包括易于使用和实现该模式时占用空间小。

让我们来看看运行中的模式。这是一个未优化的代码示例，但在这里，我们使用 Facade 来简化用于监听跨浏览器事件的接口。为此，创建一个可以用于某些代码的通用方法，该代码的任务是检查特性的存在，以便能够提供一个安全的、跨浏览器的兼容解决方案。

```
var addMyEvent = function (el, ev, fn) {  
  
    if (el.addEventListener) {  
        el.addEventListener(ev, fn, false);  
    } else if (el.attachEvent) {  
        el.attachEvent("on" + ev, fn);  
    } else {  
        el["on" + ev] = fn;  
    }  
};
```

我们都很熟悉的 jQuery 的 `$(document).ready(...)`，采用了类似的方式。在内部，它实际上是使用了一个被称为 `bindReady()` 的方法，它是这样做的：

```
bindReady: function () {  
    ...  
    if ( document.addEventListener ) {  
        // 使用便利的事件回调  
        document.addEventListener( "DOMContentLoaded", DOMContentLoaded,  
false );  
  
        // 可靠的 window.onload, 始终可用  
        window.addEventListener( "load", jQuery.ready, false );  
  
        // 如果是 IE 事件模型  
    } else if ( document.attachEvent ) {  
  
        document.attachEvent( "onreadystatechange", DOMContentLoaded );  
  
        // 可靠的 window.onload, 始终可用  
        window.attachEvent( "onload", jQuery.ready );  
    }  
};
```

...

这是 Facade 的另一个示例，其余部分仅仅使用了有限暴露的\$(document).ready(..) 接口，从而将更复杂的实现始终隐藏在视线之外。

但 Facade 不是必须单独使用的。它们也可以与其他模式集成，如 Module 模式。如下所示，Module 模式的实例包含很多已经定义的私有方法。然后使用 Facade 提供一个更简单的 API 来访问这些方法：

```
var module = (function () {

    var _private = {
        i: 5,
        get: function () {
            console.log("current value:" + this.i);
        },
        set: function (val) {
            this.i = val;
        },
        run: function () {
            console.log("running");
        },
        jump: function () {
            console.log("jumping");
        }
    };

    return {

        facade: function (args) {
            _private.set(args.val);
            _private.get();
            if (args.run) {
                _private.run();
            }
        }
    };
}());

// 输出: "running", 10
module.facade({ run: true, val: 10 });
```

在这个示例中，调用 `module.facade()` 实际上会在该模块中触发一系列的私有行为，但用户不会接触到。我们让 `facade` 变成一个不需要关注实现细节，而且更容易使用的一个特性。

## 有关抽象的要点

Facade 也有一些缺点，但值得注意的一个问题是性能。也就是说，我们必须确定 Facade 提供给实现的抽象是否包含隐性成本，如果是的话，这种成本是否是合理的。回到 jQuery 库部分，我们大多数人都知道，`getElementById ("identifier")` 和 `("#identifier")` 可以用于通过 ID 查询页面上的某个元素。

然而，你知道 `getElementById()` 本身的速度在高数量级下要快的多吗？看一下 jsPerf 测试，来查看每个浏览器级上的结果：<http://jsperf.com/getelementbyid-vs-jquery-id>。当然，现在我们必须要记住的是，jQuery（和 Sizzle，其选择器引擎）在幕后做了很多事情，以优化查询（一个 jQuery 对象，而不只是一个 DOM 节点被返回）。

这个特别的 Facade 给我们带来的挑战是：为了提供一种能够接受和解析多个查询类型的优雅选择器函数，其抽象会存在隐性成本。用户不需要访问 `jQuery.getById ("identifier")` 或 `jQuery.getbyClass ("identifier")` 等。也就是说，多年来性能的优劣已经在实践中检验过了，鉴于 jQuery 的成功，简单的 Facade 实际上能够为团队很好的效力。

当使用 Facade 模式时，要试着了解涉及的任何性能成本，并确认是否值得抽象。

## 9.10 Factory（工厂）模式

Factory 模式是另一种创建型模式，涉及创建对象的概念。其分类不同于其他模式的地方在于它不显式地要求使用一个构造函数。而 Factory 可以提供一个通用的接口来创建对象，我们可以指定我们所希望创建的工厂对象的类型（见图 9-9）。

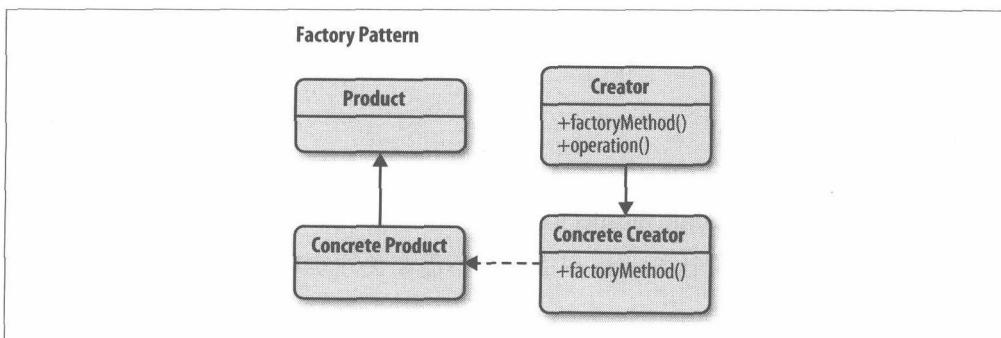


图 9-9 Factory 模式

假设有一个 UI 工厂，我们要创建一个 UI 组件的类型。不需要直接使用 new 运算符或者通过另一个创建型构造函数创建这个组件，而是要求 Factory 对象创建一个新的组件。我们通知 Factory 需要什么类型的对象（如“按钮”、“面板”），它会进行实例化，然后将它返回给我们使用。

如果对象创建过程相对比较复杂，这种方法特别有用，例如，如果它强烈依赖于动态因素或应用程序配置的话。

可以在 ExtJS 等 UI 库中找到此模式的示例，其中创建对象或组件的方法也有可能被归入子类了。

下面这个示例构建在之前的代码片段之上，使用 Constructor 模式逻辑来定义汽车。它展示了如何使用 Factory 模式来实现 vehicle 工厂：

```
// Types.js -本例构造函数的存放文件

// 定义 Car 构造函数
function Car(options) {

    // 默认值
    this.doors = options.doors || 4;
    this.state = options.state || "brand new";
    this.color = options.color || "silver";

}

// 定义 Truck 构造函数
function Truck(options) {

    this.state = options.state || "used";
    this.wheelSize = options.wheelSize || "large";
    this.color = options.color || "blue";
}

// FactoryExample.js

// 定义 vehicle 工厂的大体代码
function VehicleFactory() { }

// 定义该工厂 factory 的原型和试用工具，默认的 vehicleClass 是 Car
VehicleFactory.prototype.vehicleClass = Car;

// 创建新 Vehicle 实例的工厂方法
```

```

VehicleFactory.prototype.createVehicle = function (options) {

    if (options.vehicleType === "car") {
        this.vehicleClass = Car;
    } else {
        this.vehicleClass = Truck;
    }

    return new this.vehicleClass(options);
};

// 创建生成汽车的工厂实例
var carFactory = new VehicleFactory();
var car = carFactory.createVehicle({
    vehicleType: "car",
    color: "yellow",
    doors: 6});

// 测试汽车是由 vehicleClass 的原型 prototype 里的 Car 创建的
// 输出: true
console.log(car instanceof Car);

// 输出: Car 对象, color: "yellow", doors: 6, state:"brand new"
console.log(car);

```

在方法 1 中，我们修改了 VehicleFactory 实例来使用 Truck 类：

```

var movingTruck = carFactory.createVehicle({
    vehicleType: "truck",
    state: "like new",
    color: "red",
    wheelSize: "small"});

// 测试卡车是由 vehicleClass 的原型 prototype 里的 Truck 创建的
// 输出: true
console.log(movingTruck instanceof Truck);

// 输出: Truck 对象, color : "red", state: "like new" , wheelSize: "small"
console.log(movingTruck);

```

在方法 2 中，我们把 VehicleFactory 归入子类来创建一个构建 Truck 的工厂类：

```

function TruckFactory() { }
TruckFactory.prototype = new VehicleFactory();
TruckFactory.prototype.vehicleClass = Truck;

var truckFactory = new TruckFactory();

```

```
var myBigTruck = truckFactory.createVehicle({
    state: "omg..so bad.",
    color: "pink",
    wheelSize: "so big");

// 确认 myBigTruck 是由原型 Truck 创建的
// 输出: true
console.log(myBigTruck instanceof Truck);

// 输出: Truck 对象, color: pink", wheelSize: "so big", state: "omg. so bad"
console.log(myBigTruck);
```

### 9.10.1 何时使用 Factory 模式

Factory 模式应用于如下场景时是特别有用的：

- 当对象或组件设置涉及高复杂性时
- 当需要根据所在的不同环境轻松生成对象的不同实例时
- 当处理很多共享相同属性的小型对象或组件时
- 在编写只需要满足一个 API 契约（亦称鸭子类型）的其他对象的实例对象时。对于解耦是很有用的。

### 9.10.2 何时不应使用 Factory 模式

如果应用错误，这种模式会为应用程序带来大量不必要的复杂性。除非为创建对象提供一个接口是我们正在编写的库或框架的设计目标，否则我建议坚持使用显式构造函数，以避免不必要的开销。

由于对象创建的过程实际上是藏身接口之后抽象出来的，单元测试也可能带来问题，这取决于对象创建的过程有多复杂。

### 9.10.3 Abstract Factory ( 抽象工厂 )

了解抽象工厂模式也是有用的，它用于封装一组具有共同目标的单个工厂。它能够将一组对象的实现细节从一般用法中分离出来。

应当使用抽象工厂模式的情况是：一个系统必须独立于它所创建的对象的生成方

式，或它需要与多种对象类型一起工作。

既简单又容易理解的示例是车辆工厂，它定义了获取或注册车辆类型的方法。抽象工厂可以命名为 `AbstractVehicleFactory`。抽象工厂将允许对像 `car` 或 `truck` 这样的车辆类型进行定义，具体工厂只需要实现履行车辆契约的类（如 `Vehicle.prototype.drive` 和 `Vehicle.prototype.breakDown`）。

```
var AbstractVehicleFactory = (function () {  
  
    // 存储车辆类型  
    var types = {};  
  
    return {  
        getVehicle: function ( type, customizations ) {  
            var Vehicle = types[type];  
            return (Vehicle) ? return new Vehicle(customizations) : null;  
        },  
  
        registerVehicle: function ( type, Vehicle ) {  
            var proto = Vehicle.prototype;  
  
            // 只注册实现车辆契约的类  
            if ( proto.drive && proto.breakDown ) {  
                types[type] = Vehicle;  
            }  
  
            return AbstractVehicleFactory;  
        },  
    };  
})();  
  
// 用法：  
AbstractVehicleFactory.registerVehicle( "car", Car );  
AbstractVehicleFactory.registerVehicle( "truck", Truck );  
  
// 基于抽象车辆类型实例化一个新 car 对象  
var car = AbstractVehicleFactory.getVehicle( "car" , {  
    color: "lime green",  
    state: "like new" } );  
  
// 同理实例化一个新 truck 对象  
var truck = AbstractVehicleFactory.getVehicle( "truck" , {  
    wheelSize: "medium",  
    color: "neon yellow" } );
```

## 9.11 Mixin 模式

在 C++ 和 Lisp 等传统编程语言中，Mixin 是可以轻松被一个子类或一组子类继承功

能的类，目的是函数复用。

### 9.11.1 子类化

对于不熟悉子类化的开发人员来说，在深入研究 Mixin 和 Decorator 之前，将阅读初学者内容。

子类化这个术语是指针对一个新对象，从一个基础或超类对象中继承相关的属性。在传统的面向对象编程中，类 B 是从另外一个类 A 扩展得来。这里我们认为 A 是一个超类，B 是 A 的一个子类。因此，B 的所有实例从 A 处继承了相关方法。但是 B 仍然能够定义自己的方法，包括那些 A 最初所定义方法的重写。

A 中的一个方法，在 B 里已经被重写了，那么 B 还需要调用 A 中的这个方法吗，我们称此为方法链。B 需要调用构造函数 A（超类）吗，我们称此为构造函数链。

为了演示子类化，首先需要一个可以创建自己新实例的基本对象。让我们围绕一个人的概念来模拟子类化。

```
var Person = function (firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.gender = "male";  
};
```

下一步，指定一个新类（对象），它是现有 Person 对象的一个子类。想象一下，在继承 Person 超类上的属性同时，我们需要在 SuperHero 上添加另外不同的属性。由于超级英雄与平常人具有很多共同的特征（如姓名、性别），希望这应该能够充分说明子类化是如何工作的。

```
// Person 的新实例很容易像如下这样创建:  
var clark = new Person("Clark", "Kent");  
  
// 为超人 (Superhero) 定义一个子类构造函数  
var Superhero = function (firstName, lastName, powers) {  
  
    // 调用超类的构造函数，然后使用 .call() 方法进行调用从而进行初始化  
  
    Person.call(this, firstName, lastName);  
};
```

```

    // 最后，保存 powers，在正常 Person 里找不到的特性数组
    this.powers = powers;
};

SuperHero.prototype = Object.create(Person.prototype);
var superman = new Superhero("Clark", "Kent", ["flight", "heat-vision"]);
console.log(superman);

// 输出 Person 属性和 powers

```

`Superhero` 构造函数创建一个源于 `Person` 的对象。这种类型的对象拥有在链中比它们靠上对象的属性，如果我们已经在 `Person` 对象中设置默认值，`Superhero` 就能够重写所有继承的值，并且其对象本身也可以拥有特定的值。

## 9.11.2 Mixin（混入）

在 JavaScript 中，我们可以将继承 Mixin 看作为一种通过扩展收集功能的方式。我们定义的每个新对象都有一个原型，可以从中继承更多属性。原型可以继承于其他对象的原型，但更重要的是，它可以为任意数量的对象实例定义属性。可以利用这一点来促进函数复用（见图 9-10）。

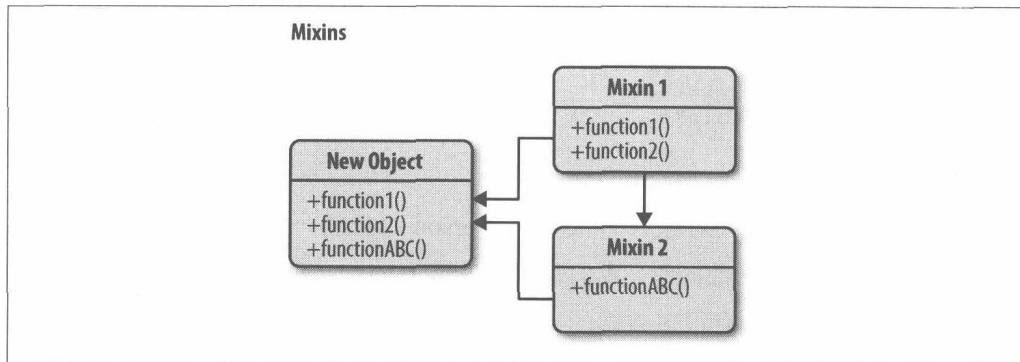


图 9-10 Mixin

Mixin 允许对象通过较低的复杂性借用（或继承）功能。由于该模式非常适用于 JavaScript 的对象原型，它为我们提供了一种相当灵活的方式，从不只一个 Mixin 中分享功能，但实际上很多功能是通过多重继承获得的。

它们可以被视为具有可以在很多其他对象原型中轻松共享属性和方法的对象。想象一下，我们在标准对象字面量中定义一个包含实用函数的 Mixin，如下所示：

```

var myMixins = {

    moveUp: function () {
        console.log("move up");
    },

    moveDown: function () {
        console.log("move down");
    },

    stop: function () {
        console.log("stop! in the name of love!");
    }

};

```

然后我们可以使用 Underscore.js 的`_.extend()`方法等辅助器轻松地扩展现有构造器函数的原型，以将上述行为包含进来：

```

// carAnimator 构造函数的大体代码
function carAnimator() {
    this.moveLeft = function () {
        console.log("move left");
    };
}

// personAnimator 构造函数的大体代码
function personAnimator() {
    this.moveRandomly = function () { /*...*/ };
}

// 使用 Mixin 扩展 2 个构造函数
_.extend(carAnimator.prototype, myMixins);
_.extend(personAnimator.prototype, myMixins);

// 创建 carAnimator 的新实例
var myAnimator = new carAnimator();
myAnimator.moveLeft();
myAnimator.moveDown();
myAnimator.stop();

// 输出：
// move left
// move down
// stop! in the name of love!

```

正如我们所看到的，这允许我们以通用方式轻松“混入”对象构造函数。

在下一个示例中，我们有两个构造函数：Car 和 Mixin。我们要做的是扩充（扩展的另一种说法）Car，以便它可以继承 Mixin 中定义的特定方法，即`driveForward()`

和 `driveBackward()`。这次，我们不会使用 `Underscore.js`。

本示例将演示如何扩展构造函数，不需要对我们可能拥有的每个构造函数都重复这个过程而将功能包含进来。

```
// 定义简单的 Car 构造函数
var Car = function (settings) {

    this.model = settings.model || "no model provided";
    this.color = settings.color || "no colour provided";
};

// Mixin
var Mixin = function () { };

Mixin.prototype = {

    driveForward: function () {
        console.log("drive forward");
    },
    driveBackward: function () {
        console.log("drive backward");
    },
    driveSideways: function () {
        console.log("drive sideways");
    }
};

// 通过一个方法将现有对象扩展到另外一个对象上
function augment(receivingClass, givingClass) {

    // 只提供特定的方法
    if (arguments[2]) {
        for (var i = 2, len = arguments.length; i < len; i++) {
            receivingClass.prototype[arguments[i]] = givingClass.
prototype [arguments[i]];
        }
    }
    // 提供所有方法
    else {
        for (var methodName in givingClass.prototype) {
            // 确保接收类不包含所处理方法的同名方法
            if (!Object.hasOwnProperty(receivingClass.prototype, methodName)) {
                receivingClass.prototype[methodName] = givingClass.
prototype[methodName];
            }
        }
    }
}

// 另一方式:
```

```

// if ( !receivingClass.prototype[methodName] ) {
// receivingClass.prototype[methodName] = givingClass.prototype[methodName];
// }
}
}

// 给 Car 构造函数增加"driveForward"和"driveBackward"两个方法
augment(Car, Mixin, "driveForward", "driveBackward");

// 创建一个新 Car
var myCar = new Car({
    model: "Ford Escort",
    color: "blue"
});

// 测试确保新增方法可用
myCar.driveForward();
myCar.driveBackward();

// 输出：
// drive forward
// drive backward

// 也可以通过不声明特定方法名的形式，将 Mixin 的所有方法都添加到 Car 里
augment(Car, Mixin);

var mySportsCar = new Car({
    model: "Porsche",
    color: "red"
});

mySportsCar.driveSideways();

// 输出：
// drive sideways

```

## 优点和缺点

Mixin 有助于减少系统中的重复功能及增加函数复用。当一个应用程序可能需要在各种对象实例中共享行为时，我们可以通过在 Mixin 中维持这种共享功能并专注于仅实现系统中真正不同的功能，来轻松避免任何重复。

也就是说，有关 Mixin 的缺点是稍有争议的。有些开发人员认为将功能注入对象原型中是一种很糟糕的想法，因为它会导致原型污染和函数起源方面的不确定性。在大型系统中，可能就会有这种情况。

我认为，强大的文档有助于将与混入函数来源有关的困惑减至最低，但对于每一种

模式，如果在实现期间多加注意，一切应该会很顺利。

## 9.12 Decorator（装饰者）模式

Decorator 是一种结构型设计模式，旨在促进代码复用。与 Mixin 相类似，它们可以被认为是另一个可行的对象子类化的替代方案。

通常，Decorator 提供了将行为动态添加至系统的现有类的能力。其想法是，装饰本身对于类原有的基本功能来说并不是必要的；否则，它就可以被合并到超类本身了。

装饰者可以用于修改现有的系统，希望在系统中为对象添加额外的功能，而不需要大量修改使用它们的底层代码。开发人员使用它们的一个共同原因是，应用程序可能包含需要大量不同类型对象的功能。想象一下，如果必须为一个 JavaScript 游戏定义数百种不同的对象构造函数会怎么样（见图 9-11）。

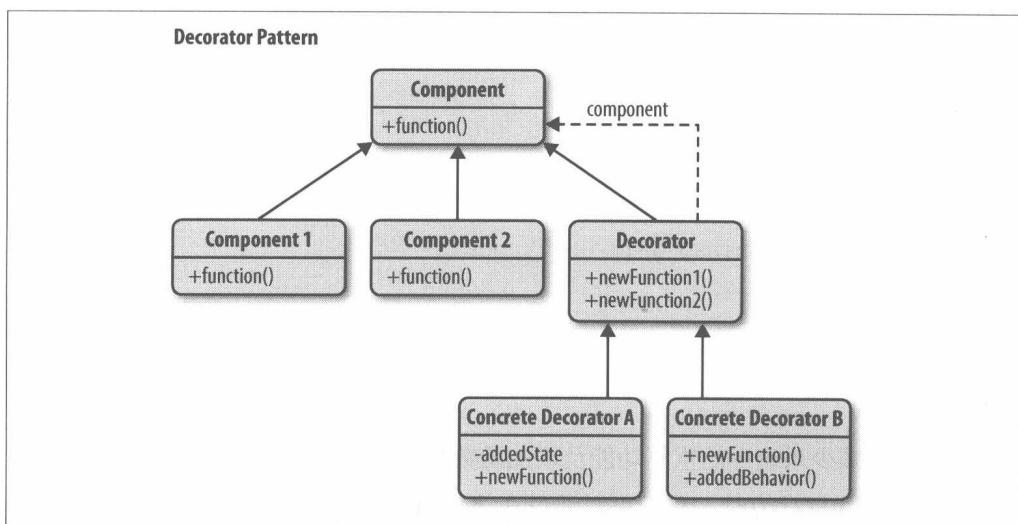


图 9-11 Decorator 模式

对象构造函数可以代表不同的玩家类型，每个类型都有不同的功能。魔戒游戏可能需要 Hobbit、Elf、Orc、Wizard、Mountain Giant、Stone Giant 等构造函数，甚至有可能数以百计。如果我们把功能作为因素计算，可以想象必须为每个能力类型组合创建子类——如：`HobbitWithRing`、`HobbitWithSword`、`HobbitWithRingAndSword`

等等。当计算越来越多的不同能力时，这并不是很实用，当然也是不可控的。

Decorator 模式并不严重依赖于创建对象的方式，而是关注扩展其额外功能。我们使用了一个单一的基本对象并逐步添加提供额外功能的 `decorator` 对象，而不是仅仅依赖于原型继承。这个想法是：向基本对象添加（装饰）属性或方法，而不是进行子类化，因此它较为精简。

在 JavaScript 中向对象添加新属性是一个非常简单的过程，所以带着这种想法，可以实现一个非常简单的 `decorator`，如下所示（示例 9-7 和示例 9-8）：

#### 示例 9-7 使用新功能装饰构造函数

```
// 车辆 vehicle 构造函数
function vehicle(vehicleType) {
    // 默认值
    this.vehicleType = vehicleType || "car";
    this.model = "default";
    this.license = "00000-000";
}

// 测试基本的 vehicle 实例
var testInstance = new vehicle("car");
console.log(testInstance);

// 输出：
// vehicle: car, model:default, license: 00000-000

// 创建一个 vehicle 实例进行装饰
var truck = new vehicle("truck");

// 给 truck 装饰新的功能
truck.setModel = function (modelName) {
    this.model = modelName;
};

truck.setColor = function (color) {
    this.color = color;
};

// 测试赋值操作是否正常工作
truck.setModel("CAT");
truck.setColor("blue");

console.log(truck);
```

```

// 输出:
// vehicle:truck, model:CAT, color: blue

// 下面的代码, 展示 vehicle 依然是不被改变的
var secondInstance = new vehicle("car");
console.log(secondInstance);

// 输出:
// vehicle: car, model:default, license: 00000-000

```

这种类型的简单实现是可行的, 但它并不能真正证明装饰者所提供的所有优势。为此, 首先要查阅一下改编的咖啡示例, 该示例来自 Freeman、Sierra 和 Bates 所著的一本名为《深入浅出设计模式》书籍, 它围绕的是模拟购买苹果笔记本。

### 示例 9-8 使用多个 Decorator 装饰对象

```

// 被装饰的对象构造函数
function MacBook() {

    this.cost = function () { return 997; };
    this.screenSize = function () { return 11.6; };
}

// Decorator 1
function Memory(macbook) {

    var v = macbook.cost();
    macbook.cost = function () {
        return v + 75;
    };
}

// Decorator 2
function Engraving(macbook) {

    var v = macbook.cost();
    macbook.cost = function () {
        return v + 200;
    };
}

// Decorator 3
function Insurance(macbook) {

    var v = macbook.cost();
    macbook.cost = function () {
        return v + 250;
    };
}

```

```
var mb = new MacBook();
Memory(mb);
Engraving(mb);
Insurance(mb);

// 输出: 1522
console.log(mb.cost());

// 输出: 11.6
console.log(mb.screenSize());
```

在这个示例中，Decorator 重写 MacBook() 超类对象的 cost() 函数来返回 MacBook 的当前价格加上特定的升级价格。

我们认为装饰作为并没有重写原始 Macbook 对象的构造函数方法（如 screenSize()），为 Macbook 定义的其他属性也一样，依然保持不变并完好无损。

实际上在前面的示例中没有已定义的接口，从创建者移动到接收者时，我们转移了确保一个对象符合接口要求的职责。

### 9.12.1 伪经典 Decorator（装饰者）

现在，我们要查看 Dustin Diaz 和 Ross Harmes 所著的《JavaScript 设计模式》(PJDP) 一书中提出的装饰者变体。

不像早些时候的一些示例，Diaz 和 Harmes 更关注如何在其他编程语言（如 Java 或 C++）中使用“接口”的概念实现装饰者，我们稍后将对其进行更详细地定义。



这个 Decorator 模式的特殊变体是用于引用目的。如果你发现它过于复杂，我建议选择前面介绍的较简单实现。

#### 9.12.1.1 接口

PJDP 将 Decorator 模式描述为一种用于在相同接口的其他对象内部透明地包装对象的模式。接口应该是对对象定义方法的一种方式，但是，它实际上并不直接指定如何实现这些方法。

接口还可以定义接收哪些参数，但这些都是可选的。

那么，我们为什么要在 JavaScript 中使用接口呢？其想法是：它们可以自我记录，并能促进可复用性。理论上，通过确保实现类保持和接口相同的改变，接口可以使代码变得更加稳定。

下面是使用鸭子类型在 JavaScript 中实现接口的一个示例，这种方法帮助确定一个对象是否是基于其实现方法的构造函数/对象的实例。

```
// 用事先定义好的接口构造函数创建接口，该函数将接口名称和方法名称作为参数
// 在 reminder 示例中，summary() 和 placeOrder() 描绘的功能，接口应该支持

var reminder = new Interface("List", ["summary", "placeOrder"]);

var properties = {
  name: "Remember to buy the milk",
  date: "05/06/2016",
  actions: {
    summary: function () {
      return "Remember to buy the milk, we are almost out!";
    },
    placeOrder: function () {
      return "Ordering milk from your local grocery store";
    }
  }
};

// 创建构造函数实现上述属性和方法

function Todo(config) {

  // 为了支持这些功能，接口示例需要检查这些功能

  Interface.ensureImplements(config.actions, reminder);

  this.name = config.name;
  this.methods = config.actions;
}

// 创建 Todo 构造函数的新实例

var todoItem = Todo(properties);

// 最后测试确保新增加的功能可用

console.log(todoItem.methods.summary());
console.log(todoItem.methods.placeOrder());

// 输出：
// Remember to buy the milk, we are almost out!
// Ordering milk from your local grocery store
```

在这个示例中，`Interface.ensureImplements` 提供了严格的功能检查。在这里（<https://gist.github.com/1057989>）可以找到它的代码以及 `Interface` 构造函数的代码。

接口的最大问题是，在 JavaScript 中没有为它们提供内置支持，试图模仿可能不太合适的另外一种语言特性是有风险的。可以在不花费大量性能成本的情况下使用享元接口，但我们将继续看一下使用相同概念的抽象装饰者。

### 9.12.1.2 抽象 Decorator（抽象装饰者）

为了演示该版本 Decorator 模式的结构，假设我们有一个超类，再次模拟 Macbook，以及模拟一个商店允许我们“装饰”苹果笔记本并收取增强功能的额外费用。

增强功能可以包括将内存升级到 4GB 或 8GB、雕刻、Parallels 或外壳。如果为每个增强选项组合使用单个子类来模拟它，看起来可能就是这样的：

```
var Macbook = function () {
    //...
};

var MacbookWith4GBRam = function () { },
    MacbookWith8GBRam = function () { },
    MacbookWith4GBRamAndEngraving = function () { },
    MacbookWith8GBRamAndEngraving = function () { },
    MacbookWith8GBRamAndParallels = function () { },
    MacbookWith4GBRamAndParallels = function () { },
    MacbookWith8GBRamAndParallelsAndCase = function () { },
    MacbookWith4GBRamAndParallelsAndCase = function () { },
    MacbookWith8GBRamAndParallelsAndCaseAndInsurance = function () { },
    MacbookWith4GBRamAndParallelsAndCaseAndInsurance = function () { };
```

...等等。

这将是一个不切实际的解决方案，因为每个可用的增强功能组合都需要一个新的子类。为了让事情变得简单点，而不需维护大量的子类，让我们来看看可以如何使用装饰者来更好地解决这个问题。

我们只需创建五个新的装饰者类，而不是需要之前看到的所有组合。在这些增强类上调用的方法将被传递给 `Macbook` 类。

在接下来的示例中，装饰者会透明地包装它们的组件，由于使用了相同的接口，它

们可以进行相互交换。

如下是我们将为 Macbook 定义的接口：

```
var Macbook = new Interface("Macbook",
    ["addEngraving",
    "addParallels",
    "add4GBRam",
    "add8GBRam",
    "addCase"]);
```

// Macbook Pro 可能需要如下这样来描述：

```
var MacbookPro = function () {
    // 实现 Macbook
};

MacbookPro.prototype = {
    addEngraving: function () {
    },
    addParallels: function () {
    },
    add4GBRam: function () {
    },
    add8GBRam: function () {
    },
    addCase: function () {
    },
    getPrice: function () {
        // 基本价格
        return 900.00;
    }
};
```

为了便于我们添加后期需要的更多选项，我们定义了一个具有默认方法的抽象装饰者类来实现 Macbook 接口，其余的选项则划入子类。抽象装饰者确保我们可以装饰出一个独立的，而且多个装饰者在不同组合下都需要的基类（还记得前面的示例吗？），而不需要为每一个可能的组合都派生子类。

```
// Macbook 装饰者抽象装饰者类

var MacbookDecorator = function (macbook) {
    Interface.ensureImplements(macbook, Macbook);
    this.macbook = macbook;
};

MacbookDecorator.prototype = {
    addEngraving: function () {
        return this.macbook.addEngraving();
    }
};
```

```

},
addParallels: function () {
    return this.macbook.addParallels();
},
add4GBRam: function () {
    return this.macbook.add4GBRam();
},
add8GBRam: function () {
    return this.macbook.add8GBRam();
},
addCase: function () {
    return this.macbook.addCase();
},
getPrice: function () {
    return this.macbook.getPrice();
}
};

```

上面的示例演示的是：Macbook decorator 接受一个对象作为组件。它使用了我们前面定义的 Macbook 接口，针对每个方法，在组件上会调用相同的方法。我们现在可以仅通过使用 MacbookDecorator 创建选项类；简单调用超类构造函数，必要时可以重写任何方法。

```

var CaseDecorator = function (macbook) {
    // 接下来调用超类的构造函数
    this.superclass.constructor(macbook);

};

// 扩展超类
extend(CaseDecorator, MacbookDecorator);

CaseDecorator.prototype.addCase = function () {
    return this.macbook.addCase() + "Adding case to macbook";
};

CaseDecorator.prototype.getPrice = function () {
    return this.macbook.getPrice() + 45.00;
};

```

正如我们可以看到的，其中的大部分内容都是很容易实现的。我们所做的是重写需要装饰的 addCase() 和 getPrice() 方法，首先执行该组件的原有方法，然后加上额外的内容（文本或价格）。到目前为止本节已展示了很多的信息了，让我们试着将所有内容整合到一个示例中，希望能够加强所学到的内容。

```

// 实例化 macbook
var myMacbookPro = new MacbookPro();

```

```
// 输出: 900.00
console.log(myMacbookPro.getPrice());

// 装饰 macbook
myMacbookPro = new CaseDecorator(myMacbookPro);

// 返回的将是 945.00
console.log(myMacbookPro.getPrice());
```

由于装饰者可以动态地修改对象，因此它们是一种用于改变现有系统的完美模式。有时候，为对象创建装饰者比维护每个对象类型的单个子类要简单一些。可以让可能需要大量子类对象的应用程序的维护变得更加简单。

## 9.12.2 使用 jQuery 的装饰者

与我们已经涉及的其他模式一样，也有一些使用 jQuery 实现的装饰者模式的示例。jQuery.extend()允许我们在运行时或者在随后一个点上动态地将两个或两个以上的对象（和它们的属性）一起扩展（或合并）为一个单一对象。

在这种情况下，一个目标对象可以用新功能来装饰，而不会在源/超类对象中破坏或重写现有的方法（虽然这是可以做到的）。

在接下来的示例中定义三个对象：defaults、options 和 settings。该任务的目的是为了装饰 defaults 对象，将 options 中的额外功能附加到 defaults 上。我们必须首先使 defaults 保持未接触状态，并且保持稍后可以访问其属性或函数的能力；然后，给 defaults 赋予使用装饰属性和函数的能力，这些装饰属性和函数是从 options 里获取的：

```
var decoratorApp = decoratorApp || {};
// 定义要使用的对象
decoratorApp = {

    defaults: {
        validate: false,
        limit: 5,
        name: "foo",
        welcome: function () {
            console.log("welcome!");
        }
    },
    options: {
        validate: true,
        name: "bar",
    }
};
```

```

        helloWorld: function () {
            console.log("hello world");
        }
    },
    settings: {},
    printObj: function (obj) {
        var arr = [],
            next;
        $.each(obj, function (key, val) {
            next = key + ": ";
            next += $.isPlainObject(val) ? printObj(val) : val;
            arr.push(next);
        });
        return "{" + arr.join(", ") + "}";
    }
};

// 合并 defaults 和 options，没有显式修改 defaults
decoratorApp.settings = $.extend({}, decoratorApp.defaults, decoratorApp.options);

// 这里所做的就是装饰可以访问 defaults 属性和功能的方式(options 也一样), defaults
本身未作改变

$("#log")
    .append(decoratorApp.printObj(decoratorApp.settings) +
        +decoratorApp.printObj(decoratorApp.options) +
        +decoratorApp.printObj(decoratorApp.defaults));

// settings -- { validate: true, limit: 5, name: bar,
// welcome: function () { console.log("welcome!"); }, }
// helloWorld: function () { console.log("hello!"); } }
// options -- { validate: true, name: bar,
// helloWorld: function () { console.log("hello!"); } }
// defaults -- { validate: false, limit: 5, name: foo,
// welcome: function () { console.log("welcome!"); } }

```

### 9.12.3 优点和缺点

开发人员喜欢使用这种模式，因为它使用时可以是透明的，并且也是相当灵活的：正如我们所看到的，对象可以被新行为包装或“装饰”，然后可以继续被使用，而不必担心被修改的基本对象。在一个更广泛的上下文中，这种模式也使我们不必依靠大量的子类来获得同样的好处。

但是在实现该模式时，也有一些缺陷是我们应该要注意的。如果管理不当，它会极大地复杂化应用程序架构，因为它向我们的命名空间引入了很多小型但类似的对

象。让人担心的是，除了对象变得难以管理，其他不熟悉这个模式的开发人员可能难以理解为什么使用它。

大量的评论或模式研究应该有助于解决后者的问题，但是，只要我们继续把握住在应用程序中使用装饰者的广度，在这两方面就应该可以做得很好。

## 9.13 Flyweight（享元）模式

Flyweight 模式是一种经典的结构型解决方案，用于优化重复、缓慢及数据共享效率较低的代码。它旨在通过与相关的对象共享尽可能多的数据来减少应用程序中内存的使用（如：应用程序配置、状态等，见图 9-12）。

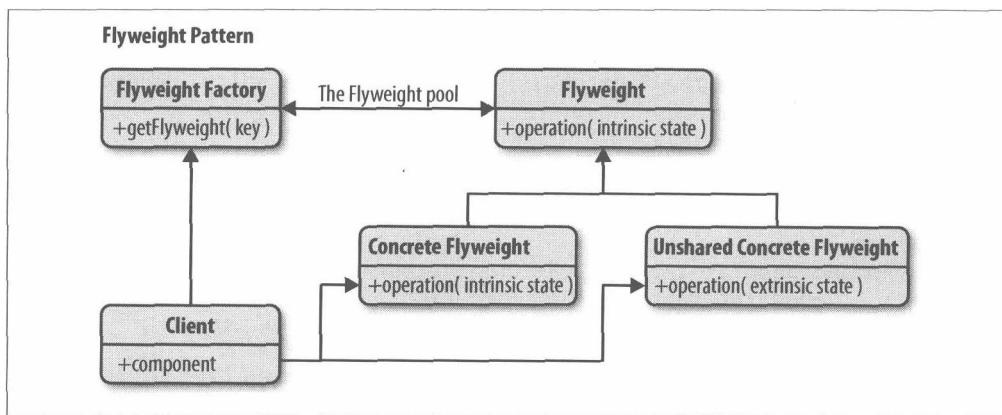


图 9-12 Flyweight 模式

该模式最早是由 Paul Calder 和 Mark Linton 于 1990 年构思出来，它以拳击重量级别命名，它包括重量不到 112 磅的拳手。Flyweight 这个名字是源自这一重量级别，因为它所指的是：模式旨在帮助我们实现的轻量级（内存占用）。

在实践中，Flyweight 数据共享会涉及获取多个对象使用的若干相似对象或数据结构，以及将这些数据放到一个单一的外部对象中。我们可以将该对象传递给依赖这些数据的对象，而不是在每一个对象都存储相同的数据。

### 9.13.1 使用 Flyweight 模式

Flyweight 模式的应用方式有两种。第一种是用于数据层，处理内存中保存的大量

相似对象的共享数据。

第二种是用于 DOM 层，Flyweight 可以用作中央事件管理器，来避免将事件处理程序附加到父容器中的每个子元素上，而是将事件处理程序附加到这个父容器上。

鉴于数据层是 Flyweight 模式最常使用的地方，我们首先要对它进行了解。

### 9.13.2 Flyweight 和共享数据

对于该应用程序，还有一些经典 Flyweight 模式的概念我们需要注意。在 Flyweight 模式中，有个有关两个状态的概念—内部和外部。对象中的内部方法可能需要内部信息，没有内部信息，它们就绝对无法正常运行。但外部信息是可以被删除的或是可以存储在外部的。

具有相同内部数据的对象可以被替换为一个由 factory 方法创建的单一共享对象。这使我们可以极大减少存储隐式数据的总数量。

这么做的好处是，我们能够密切关注已经被实例化的对象，这样新副本就只需要创建与现有对象不同的部分就可以了。

我们使用管理器来处理外部状态。如何实现管理器是不固定的，但有一种方法就是让管理器对象包含一个外部状态的中央数据库以及这些外部状态所属的享元对象。

### 9.13.3 实现经典 Flyweight (享元)

由于近年来 Flyweight 模式还没有在 JavaScript 中大量使用，很多给我们带来启发的相关实现都是来自 Java 和 C++。

享元模式的首次代码实现就是我所写的 JavaScript 实现，该实现基于维基百科 Flyweight 模式的 Java 示例 ([http://en.wikipedia.org/wiki/Flyweight\\_pattern](http://en.wikipedia.org/wiki/Flyweight_pattern))。

在这个实现中我们将利用三种类型的 Flyweight 组件，它们是：

#### *Flyweight* (享元)

描述一个接口，通过这个接口 flyweight 可以接受并作用于外部状态。

## *Concrete flyweight* (具体享元)

实现 Flyweight 接口，并存储内部状态。**Concrete Flyweight** 对象必须是可共享的，并能够控制外部状态。

## *Flyweight factory* (享元工厂)

创建并管理 flyweight 对象。确保合理地共享 flyweight，并将它们当作一组对象进行管理，并且如果我们需要单个实例时，可以查询这些对象。如果该对象已经存在则直接返回，否则，创建新对象并返回。

它们与实现中的下列定义相对应：

- **CoffeeOrder:** 享元
- **CoffeeFlavor:** 具体享元
- **CoffeeOrderContext:** 辅助器
- **CoffeeFlavorFactory:** 享元工厂
- **testFlyweight:** 享元的应用

## 鸭子补丁“实现”

鸭子补丁 (Duck punching) 使我们无需修改运行时源，就可以扩展一种语言或解决方案的功能。由于下一个解决方案要求使用 Java 关键字 (`implements`) 来实现接口，并且无法在原生 JavaScript 中找到，所以让我们首先对它进行鸭子补丁。

`Function.prototype.implementsFor` 作用于一个对象构造函数，并将接受一个父类（函数）或对象，或者使用普通继承（函数）或虚拟继承（对象）来继承它。

```
// 在 JS 里模拟纯虚拟继承 implement
Function.prototype.implementsFor = function (parentClassOrObject) {
    if (parentClassOrObject.constructor === Function)
    {

        // 正常继承
        this.prototype = new parentClassOrObject();
        this.prototype.constructor = this;
    }
}
```

```

        this.prototype.parent = parentClassOrObject.prototype;
    }
    else {
        // 纯虚拟继承
        this.prototype = parentClassOrObject;
        this.prototype.constructor = this;
        this.prototype.parent = parentClassOrObject;
    }
    return this;
};

```

通过使一个函数显式地继承一个接口，可以用它来为缺少的 `implements` 关键字打上补丁。在下面的代码里，`CoffeeFlavor` 实现了 `CoffeeOrder` 接口，且必须包含它的接口方法，以便将功能的实现赋值给对象。

```

// 享元对象
var CoffeeOrder = {

    // 接口
    serveCoffee: function (context) { },
    getFlavor: function () { }

};

// 实现 CoffeeOrder 的具体享元对象
function CoffeeFlavor(newFlavor) {

    var flavor = newFlavor;

    // 如果已经为某一功能定义了接口，则实现该功能
    if (typeof this.getFlavor === "function") {
        this.getFlavor = function () {
            return flavor;
        };
    }

    if (typeof this.serveCoffee === "function") {
        this.serveCoffee = function (context) {
            console.log("Serving Coffee flavor "
                + flavor
                + " to table number "
                + context.getTable());
        };
    }
}

// 为 CoffeeOrder 实现接口
CoffeeFlavor.implementsFor(CoffeeOrder);

// 处理 coffee 订单的 table 数

```

```

function CoffeeOrderContext(tableNumber) {
    return {
        getTable: function () {
            return tableNumber;
        }
    };
}

// 享元工厂对象
function CoffeeFlavorFactory() {
    var flavors = [],
        flavor;

    return {
        getcoffeeFlavor: function (flavorName) {
            flavor = flavors[flavorName];
            if (flavor === undefined) {
                flavor = new CoffeeFlavor(flavorName);
                flavors.pushc [flavorName], flavor]);
            }
            return flavor;
        },
        getTotalCoffeeFlavorsMade: function () {
            return flavors.length;
        }
    };
}

// 样例用法:
// testFlyweight()

function testFlyweight() {
    // 已订购的 flavor.
    var flavors = new CoffeeFlavor(),
        flavorFactory;

    // 订单 table
    tables = new CoffeeOrderContext(),

    // 订单数量
    ordersMade = 0,
    flavorFactory = new CoffeeFlavorFactory();

    function takeOrders(flavorIn, table) {
        flavors[ordersMade] = flavorFactory.getcoffeeFlavor(flavorIn);
        tables[ordersMade++] = new CoffeeOrderContext(table);
    }

    takeOrders("Cappuccino", 2);
    takeOrders("Cappuccino", 2);
    takeOrders("Frappe", 1);
}

```

```

takeOrders("Frappe", 1);
takeOrders("Xpresso", 1);
takeOrders("Frappe", 897);
takeOrders("Cappuccino", 97);
takeOrders("Cappuccino", 97);
takeOrders("Frappe", 3);
takeOrders("Xpresso", 3);
takeOrders("Cappuccino", 3);
takeOrders("Xpresso", 96);
takeOrders("Frappe", 552);
takeOrders("Cappuccino", 121);
takeOrders("Xpresso", 121);

for (var i = 0; i < ordersMade; ++i) {
    flavors[i].serveCoffee(tables[i]);
}
console.log(" ");
console.log("total CoffeeFlavor objects made: " + flavorFactory.
getTotalCoffeeFlavorsMade());
}

```

#### 9.13.4 转换代码以使用 Flyweight (享元) 模式

接下来，通过实现一个系统来管理图书馆中的所有书籍，让我们来继续了解一下享元。每本书的重要元数据可以被分解成如下形式：

- ID
- Title
- Author
- Genre
- Page count
- Publisher ID
- ISBN

我们还将需要使用以下属性来跟踪哪些成员已借出了哪些书籍，借书日期以及预计返还的日期。

- checkoutDate

- `checkoutMember`
- `dueReturnDate`
- `availability`

因此每本书在使用享元模式进行优化之前，都会按如下方式表示：

```
var Book = function (id, title, author, genre, pageCount, publisherID,
ISBN, checkoutDate, checkoutMember, dueReturnDate, availability) {

    this.id = id;
    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;
    this.checkoutDate = checkoutDate;
    this.checkoutMember = checkoutMember;
    this.dueReturnDate = dueReturnDate;
    this.availability = availability;

};

Book.prototype = {
    getTitle: function () {
        return this.title;
    },
    getAuthor: function () {
        return this.author;
    },
    getISBN: function () {
        return this.ISBN;
    },
    // 鉴于篇幅，其他属性就暂不列出了
    updateCheckoutStatus: function (bookID, newStatus, checkoutDate,
checkoutMember, newReturnDate) {

        this.id = bookID;
        this.availability = newStatus;
        this.checkoutDate = checkoutDate;
        this.checkoutMember = checkoutMember;
        this.dueReturnDate = newReturnDate;
    },
    extendCheckoutPeriod: function (bookID, newReturnDate) {
        this.id = bookID;
        this.dueReturnDate = newReturnDate;
    }
};
```

```

        },
        isPastDue: function (bookID) {
            var currentDate = new Date();
            return currentDate.getTime() > Date.parse(this.dueReturnDate);
        }
    );

```

刚开始对于少量书籍可能是行得通的，但是，当图书馆扩大到拥有一个更大的库存，并且每本书都有多个版本和副本时，就会发现随着时间的推移，管理系统运行得越来越慢。使用数以千计的书籍对象可能会淹没可用内存，但可以使用享元模式优化系统来改善这个问题。

现在可以将数据分成内部和外部状态，如下所示：与书籍对象（title、author 等）相关的数据是内部状态，而借出数据（checkoutMember、dueReturnDate 等）是外部状态。实际上这意味着，每个书籍属性组合只需要有一个 Book 对象。它仍然要处理相当多的对象，但比以前处理的对象明显减少了。

下面书籍元数据组合的单个实例将在指定书名的书籍副本之间共享。

```

// 享元优化版本
var Book = function (title, author, genre, pageCount, publisherID, ISBN) {

    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;
};


```

正如我们可以看到的，外部状态已被删除。图书馆借出有关的所有事情都将转移给管理器，由于对象数据现在已被分割，可以使用工厂进行实例化。

### 9.13.5 基本工厂

现在让我们来定义一个基本的工厂。首先，必须要检查一下指定书名的书是否已在系统内部创建。如果已经创建，则返回它；如果没有，就会创建并存储这本新书，以便以后可以访问它。这确保我们仅为每一个特定的内部数据块创建一个拷贝：

```

// 书籍工厂单例
var BookFactory = (function () {
    var existingBooks = {}, existingBook;
    return {
        createBook: function (title, author, genre, pageCount, publisherID, ISBN) {
            // 如果书籍之前已经创建，则找出并返回它
            // !!强制返回布尔值
            existingBook = existingBooks[ISBN];
            if (!!existingBook) {
                return existingBook;
            } else {
                // 如果没找到，则创建一个该书的新实例，并保存
                var book = new Book(title, author, genre, pageCount, publisherID, ISBN);
                existingBooks[ISBN] = book;
                return book;
            }
        }
    };
});

```

### 9.13.6 管理外部状态

接下来，我们需要存储从 Book 对象中删除的状态。幸运的是，可以使用管理器（我们会将它定义为一个单例）来封装它们。一个 Book 对象和借书成员的组合将被称为书籍记录。管理器会将它们存储起来，它还包括在 Book 类享元优化期间我们排除的与借出有关的逻辑。

```

// 书籍记录管理器单例
var BookRecordManager = (function () {

    var bookRecordDatabase = {};

    return {
        // 添加新书到图书馆系统
        addBookRecord: function (id, title, author, genre, pageCount,
publisherID, ISBN, checkoutDate,
        //checkoutMember, dueReturnDate, availability) {
            var book = bookFactory.createBook(title, author, genre, pageCount,
publisherID, ISBN);

            bookRecordDatabase[id] = {
                checkoutMember: checkoutMember,
                checkoutDate: checkoutDate,
                dueReturnDate: dueReturnDate,
                availability: availability,
                book: book

```

```

    };
},
updateCheckoutStatus: function (bookID, newStatus, checkoutDate,
    checkoutMember, newReturnDate) {
    var record = bookRecordDatabase[bookID];
    record.availability = newStatus;
    record.checkoutDate = checkoutDate;
    record.checkoutMember = checkoutMember;
    record.dueReturnDate = newReturnDate;
},
extendCheckoutPeriod: function (bookID, newReturnDate) {
    bookRecordDatabase[bookID].dueReturnDate = newReturnDate;
},
isPastDue: function (bookID) {
    var currentDate = new Date();
    return currentDate.getTime() > Date.parse(
        bookRecordDatabase[bookID].dueReturnDate);
}
};
});

```

这些代码修改的结果是，从 Book 类中提取的所有数据，现在被存储在 BookManager 单例（BookDatabase）的属性中，这比我们以前使用大量对象时的效率要高很多。现在与书籍出借相关的方法在这里成为了基础，因为它们处理的是外部数据，而不是内部数据。

这个过程给我们的最终解决方案上增加了一点复杂性，但与它所解决的性能问题相比，这只是一个小小的问题。它具有数据智能性，如果有 30 本完全相同的书，我们现在只需要存储它一次。同时，每个函数都占用内存。通过使用 Flyweight 模式，这些函数在一个地方（在管理器上）存在，而不是在每个对象上存在，从而节约更多的内存。

### 9.13.7 Flyweight（享元）模式和 DOM

文档对象模型（DOM）支持两种方式让对象检测事件：自上而下（事件捕捉）和自下而上（事件冒泡）。

在事件捕捉中，事件首先被最外层的元素捕捉，然后传播到最里面的元素。在事件冒泡中，事件被捕捉并传递给最里面的元素，然后传播到外部元素。

在这个上下文中描述享元的最好比喻之一是由 Gary Chisholm 编写的，它是类似这样的：



试着用池塘的方式思考一下享元。一条鱼张开它的嘴（事件），气泡升到表面（冒泡），当气泡到达表面（动作）时，一只坐在顶部的苍蝇飞走了。在本例中，我们可以很容易地把鱼张开嘴转换成点一个按钮，气泡转换成冒泡效应，苍蝇飞走可转换成运行一些功能。

引入冒泡用于处理这些情况：一个单一的事件（如一次点击）可能是由 DOM 层级的不同级别所定义的多个事件处理程序进行处理。上述事情发生时，事件冒泡先执行为最低层级特定元素定义的事件处理程序。此后，事件在冒泡到更高级元素之前，先冒泡到包含的这些元素上。

享元可以用来进一步调整事件冒泡过程，正如我们即将要看到的（示例 9-9）。

在第一个实际示例中，假设一个文档中有一些相似的元素，在用户对它们执行用户动作（如：点击、鼠标悬停）时执行同样相似的行为。

通常在构建我们自己的 `accordion` 组件、菜单或其他基于列表的小部件时，我们要做的就是将一个点击事件绑定至父容器（如`$('.ul li a').on(..)`）中的每个链接元素上。其实不需将点击绑定至多个元素，我们就可以很容易地将享元附加到容器的顶部，它可以监听来自下面的事件。然后这些事情可以使用逻辑进行处理，逻辑与否简单取决于要求是否简单或复杂。

由于之前经常提到的组件类型的每个部分都有相同的重复标记（如 `accordion` 的每一节代码），有很大的机会是：被点击的每个元素的行为都和附近其他带有同名样式（class）元素的行为非常相似。利用这些信息，我们将使用享元来构建一个基本的 `accordion`。

在 jQuery 用户将初始化点击绑定到一个容器 `div` 的同时，这里使用了一个 `stateManager` 命名空间来封装我们的享元逻辑。为了确保页面上没有其他相似逻辑处理程序附加在 `div` 容器上，刚开始就应用 `unbind` 事件。

现在要确定容器中的哪个子元素被点击，我们利用一个 `target` 检查，它提供了一个对被点击元素的引用，和父元素无关。然后，我们利用此信息来处理单击事件，而不是在页面加载时将事件绑定至特定的子元素上。

## 示例 9-9 集中事件处理

如下是 HTML 代码：

```
<div id="container">
    <div class="toggle" href="#">More Info (Address)
        <span class="info">
            This is more information
        </span></div>
    <div class="toggle" href="#">Even More Info (Map)
        <span class="info">
            <iframe src="http://www.map-generator.net/extmap.php?name=London&address=london%2C%20england&width=500...gt;"></iframe>
        </span>
    </div>
</div>
```

如下是 JavaScript 代码：

```
var stateManager = {

    fly: function () {
        var self = this;

        $("#" + container).unbind().on("click", function (e) {
            var target = $(e.originalTarget || e.srcElement);
            if (target.is("div.toggle")) {
                self.handleClick(target);
            }
        });
    },

    handleClick: function (elem) {
        elem.find("span").toggle("slow");
    }
};
```

这里的好处在于，我们将很多独立的动作转变成一个共享的动作（可能会节省内存）。

在第二个示例中，我们可以通过使用具有 jQuery 的享元模式进一步提高性能。

James Padolsey 之前写了一篇名为《76 bytes for faster jQuery》的文章，文中他提醒我们：每次 jQuery 触发一个回调，无论何种类型（过滤器、每个、事件处理程序），我们都能够通过 this 关键字访问函数的上下文（DOM 元素与它相关）。

可惜的是，我们中的很多人都已经习惯了在\$( )或jQuery( )中包装 this 这个想法，这意味着每次构建 jQuery 的新实例都不是必要的。而不是像如下这样做：（示例 9-10）

### 示例 9-10 使用 Flyweight 进行性能优化

```
$(“div”).on(“click”, function () {
    console.log(“You clicked: ” + $(this).attr(“id”));
});

// 我们需要避免使用 DOM 元素创建 jQuery 对象（像上面的代码那样），直接像下面这样使用
DOM 元素即可：

$(“div”).on(“click”, function () {
    console.log(“You clicked: ” + this.id);
});
```

James 希望在下列上下文中使用 jQuery 的 `jQuery.text`；但是，他不同意的观点是：在每个迭代循环里创建新的 jQuery 对象。

```
$(“a”).map(function () {
    return $(this).text();
});
```

在冗余的包装方面（这里可能是使用 jQuery 实用方法的情况下），最好使用 `jQuery.methodName`（如：`jQuery.text`），而不是 `jQuery.fn.methodName`（如：`jQuery.fn.text`）。其中，`methodName` 代表一个实用程序，例如 `each()` 或 `text`。这样做不需要在每次调用我们的函数时，都调用更高一级的抽象或创建一个新的 jQuery 对象，因为 `jQuery.methodName` 是库本身在底层抽象所使用的方法，以助力 `jQuery.fn.methodName`。

因为不是所有的 jQuery 方法都有相应的单节点函数，所以 Padolsey 想出了 `jQuery.single` 工具这一概念。

这里的看法是：单一的 jQuery 对象被创建，用于每次对 `jQuery.single` 的调用（实际上意味着只有一个 jQuery 对象被创建）。可以在下面找到它的实现，由于我们是将多个可能对象的数据合并到一个更加集中的单一结构中，这在技术上讲也是享元。

```
jQuery.single = (function (o) {
    var collection = jQuery([1]);
    return function (element) {
        // 将元素赋值给集合：
        collection[0] = element;

        // 返回集合：
        return collection;
    };
});
```

使用链接的示例如下所示：

```
$( "div" ).on( "click", function () {  
    var html = jQuery.single(this).next().html();  
    console.log(html);  
});
```



虽然我们可能相信，简单缓存 jQuery 代码可能会提供相等的性能受益，Padolsey 称仍然值得使用`$.single`，并且它可以表现的更好。这并不是说不需要使用任何缓存，只是要注意这种方法是对我们有帮助的。要进一步了解`$.single`方面的细节，我建议大家阅读 Padolsey 的完整文章。

## 第 10 章

# JavaScript MV\* 模式

在本节中，我们将回顾三个非常重要的架构模式：MVC（模型-视图-控制器）和 MVP（模型-视图-表示器）和 MVVM（模型-视图-视图模型）。在过去，这三种模式都被大量用于构建桌面和服务器端应用程序，只是在最近几年它才被应用于 JavaScript。

由于大多数正在使用这些模式的 JavaScript 开发人员都选择使用 Backbone.js 等库来实现 MVC/MV\* 式的结构，我们将比较现代解决方案对 MVC 的运用与经典解决方案在这些模式上的运用有何不同。

首先来了解一下基础知识。

### 10.1 MVC

MVC 是一种架构设计模式，它通过关注点分离鼓励改进应用程序组织。它强制将业务数据（Model）与用户界面（View）隔离，第三个组件（Controller）仍然管理逻辑和用户输入。这种模式最初是由 Trygve Reenskaug ([http://en.wikipedia.org/wiki/Trygve\\_Reenskaug](http://en.wikipedia.org/wiki/Trygve_Reenskaug)) 在研究 Smalltalk-80 (1979) 期间设计出来的，在 Smalltalk-80 (1979) 中，它最初被称为模型-视图-控制器-编辑器（Model- View- Controller-Editor）。1995 年出版的《设计模式：可复用面向对象软件的基础》（亦称“四人组”之书）继续对 MVC 进行了深入的阐述，在推广使用方面发挥了重要作用。

## Smalltalk-80 MVC

重要的是要了解最初 MVC 模式的目的解决什么问题，因为它自产生之初已发生了很大的变化。早在 70 年代，图形用户界面是少之又少，一个被称为分离表示 (<http://martinfowler.com/eaaDev/uiArchs.html>) 的概念开始被用作在领域对象和表示对象之间做清晰划分的方法，领域对象在现实世界中塑造概念（如一张图片、一个人），表示对象呈现给用户屏幕。

MVC 的 Smalltalk-80 实现进一步延伸了这个概念，并包含一个从用户界面分离出应用程序逻辑的目的。它的概念是：解耦应用程序的这些部分，同时也将允许应用程序中的其他接口实现 Model 复用。关于 Smalltalk-80 的 MVC 架构有一些值得注意的有趣地方：

- Model 代表特定于领域的数据，不了解用户界面（View（视图）和 Controller（控制器））。当一个 Model（模型）改变时，它会通知它的观察者。
- View 描绘的是 Model 的当前状态。Observer 模式用于让 View 了解 Model 什么时候更新或修改。
- Presentation 由 View 关注，但不只是单个 View（视图）和 Controller（控制器），屏幕上显示的每个部分或者元素都需要 View-Controller 对。
- Controller 在这个 View-Controller 对中的作用是处理用户交互（如按键和点击等动作），为 View 做决定。

当开发人员了解到 Observer 模式（现在通常作为发布/订阅（Publish/Subscribe）变异来实现）在几十年前就是 MVC 架构的一部分时，他们有时会很惊讶。在 Smalltalk-80 的 MVC 中，View 观察 Model。正如前面所提到的，一旦 Model 发生变化，View 也随之作出反应。简单的一个例子就是一个由股票市场数据支持的应用程序。为了让应用程序变得有用，在 Model 中对数据的任何更改都应使 View 立即刷新。

Martin Fowler (<http://martinfowler.com>) 多年来在写作 MVC 的起源方面做的很出色，如果大家有兴趣进一步了解 Smalltalk-80 的 MVC 历史，推荐大家阅读他的作品。

## 10.2 为 JavaScript 开发人员提供的 MVC

我们已经回顾了 1970 年代的开发历史，现在让我们回到现在。在现代，MVC 模式已被应用于各种各样的编程语言中，包括与我们最有关联的语言：JavaScript。JavaScript 现在有很多框架，支持 MVC（或它之上的变异，我们称为 MV\* 家族），从而允许开发人员轻松向应用程序中添加结构。

这些框架包括诸如 Backbone、Ember.js 和 AngularJS。考虑到避免“意大利面条式”代码的重要性，由于它的无结构性，它描述的是非常难以阅读或维护的代码，现代 JavaScript 开发人员必须要了解该模式提供的内容。这使我们能够真正了解这些框架可以让我们以不同的方式进行工作（见图 10-1）。

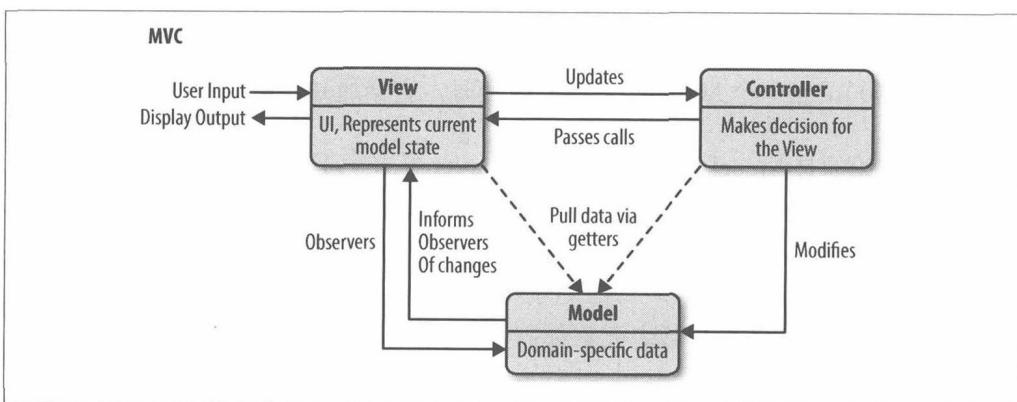


图 10-1 MVC 模式

我们都知道 MVC 是由三个核心组件组成，如以下部分中所描述的。

### 10.2.1 Model (模型)

Model 管理应用程序的数据。Model 不涉及用户界面，也不涉及表示层，而是代表应用程序可能需要的独特形式的数据。当 Model (模型) 改变时（如它更新时），它通常会通知它的观察者（如 View (视图)，我们即将要介绍的概念 Model (模型) 发生了改变，这样它们就可以做出相应的反应。

为了进一步了解 Model (模型)，假设我们有一个 JavaScript 图片库应用程序。在图片库中，一个图片可以有它自己的 Model (模型)，因为它代表了一种独特的特定领域数据。

这样一种 Model（模型）可能包含相关的属性，如标题、图像源以及额外的元数据。一个特定的图片将存储于一个 Model（模型）的实例中，Model（模型）也可以是可复用的。下面我们可以看到一个示例：使用 Backbone 实现的一个非常简单的 Model（模型）。

```
var Photo = Backbone.Model.extend({  
  // photo 的默认属性  
  defaults: {  
    src: "placeholder.jpg",  
    caption: "A default image",  
    viewed: false  
  },  
  
  // 确保每个 photo 都有一个 src  
  initialize: function () {  
    this.set({ "src": this.defaults.src });  
  }  
});
```

Model（模型）内置的功能在框架中各不相同，但它们通常是支持属性验证，这里的属性代表 Model（模型）的属性，例如 Model（模型）标识符。当在实际应用程序中使用 Model（模型）时，我们一般也要求 Model（模型）具持久化。持久化可以允许我们编辑和更新 Model（模型），保存其最新状态在内存中、用户的localStorage 数据存储中或者与数据库同步。

此外，一个 Model（模型）可能有多个观察它的 View（视图）。比如说，图片 Model（模型）包含元数据，比如它的位置（经度和纬度），图片中的朋友（一系列标识符），和一系列标签，开发人员可以决定提供单个 View（视图）来显示这三个方面。

现代 MVC/MV\* 框架很少提供将 Model（模型）组织在一起的方法（如，在 Backbone 中，这些群体被称为“集合”）。在集合组里管理 Model（模型）允许我们根据从组中收到的通知编写应用程序逻辑，该组中包含的所有 Model（模型）都应被改变。这使我们无需手动观察单个 Model（模型）实例。

在这里我们可以看到将 Model（模型）分组到一个简化的 Backbone 集合中的示例。

```
var PhotoGallery = Backbone.Collection.extend({  
  // 引用到集合模型  
  model: Photo,
```

```
// 过滤所有被查看过的图片
viewed: function () {
    return this.filter(function (photo) {
        return photo.get("viewed");
    });
},
// 过滤所有未被查看过的图片
unviewed: function () {
    return this.without.apply(this, this.viewed());
}
);
```

MVC 旧版本可能也涉及到管理应用程序状态 *state* 的 Model（模型）概念。在 JavaScript 应用程序中，*state* 有不同的含义，通常指的是当前的“状态”——如在一个固定点上用户屏幕上的 View（视图）或子视图（带有特定数据）。在讨论单页面应用程序时通常会讨论状态，同时也需要模拟状态的概念。

总而言之，Model（模型）主要是与业务数据有关。

### 10.2.2 View（视图）

视图是 Model（模型）的可视化表示，表示当前状态的筛选视图。Smalltalk 视图是关于绘制和维护一个位图，而 JavaScriptView（视图）是关于构建和维护一个 DOM 元素。

一个 View（视图）通常检测一个 Model（模型），并在 Model（模型）更改时进行通知，使 View（视图）本身能够相应的更新。设计模式方面的文章通常将 View（视图）指为“愚钝的”，因为它们对应用程序中 Model（模型）和控制器的了解是有限的。

用户可以与 View（视图）交互，包括读取和编辑 Model（模型），即，在 Model（模型）中获取或设置属性值。由于 View（视图）是表示层，通常我们能够以一种友好的方式进行编辑和更新。例如，我们在前面所讨论的图片库应用程序，可以通过“编辑” View（视图）为 Model（模型）编辑提供便利，在该 View（视图）中，选择了特定图片的用户可以编辑它的元数据。

更新 Model（模型）的实际任务其实是在 Controller（控制器）上（我们即将要介绍）。

让我们使用纯 JavaScript 示例实现来进一步研究 View（视图）。下面可以看到一个创建单一图片 View（视图）的函数，实现了一个 Model（模型）实例和一个 Controller（控制器）实例。

我们在 View（视图）内定义一个 render() 实用工具，它负责使用 JavaScript 模板引擎（Underscore 模板）渲染 photoModel 的内容，并更新由 photoEl 引用的 View（视图）的内容。

然后 photoModel 将 render() 回调作为一个订阅者添加进去，以便在 Model（模型）改变时，可以通过 Observer 模式触发 View（视图）更新。

你可能想知道，这里的用户交互在哪里发挥作用了。当用户单击 View（视图）中的任何元素时，View（视图）没有责任去了解下一步要做什么。它依赖于一个 Controller（控制器）来为它做出这个决定。在我们的样例实现中，这是通过向 photoEl 添加一个事件监听器来实现的，该监听器会处理返回给 Controller（控制器）的单击行为，包括连同 Model（模型）信息一起传递，以备不时之需。

该架构的优点是：在应用程序所需功能方面，每个组件都发挥着自己的单独作用。

```
var buildPhotoView = function (photoModel, photoController) {  
    var base = document.createElement("div"),  
        photoEl = document.createElement("div");  
  
    base.appendChild(photoEl);  
  
    var render = function () {  
        // 使用模板库（例如 Underscore）为 photo 实体生成 HTML  
        photoEl.innerHTML = _.template("#photoTemplate", {  
            src: photoModel.getSrc()  
        });  
    };  
  
    photoModel.addSubscriber(render);  
  
    photoEl.addEventListener("click", function () {  
        photoController.handleEvent("click", photoModel);  
    });  
  
    var show = function () {  
        photoEl.style.display = "";  
    };  
  
    var hide = function () {  
        photoEl.style.display = "none";  
    };  
  
    return {  
        showView: show,  
        hideView: hide  
    };  
};
```

```
};  
};
```

## 模板

在支持 MVC/MV\* 的 JavaScript 框架上下文中，有必要再简要讨论一下我们在上一节中谈及过的 JavaScript 模板及其与 View（视图）的关系。

长期以来我们都是了解到（已经过验证）手工创建大量的 HTML 标记并通过字符串拼接是非常不好的性能实践。开发者会遭受遍历非正常格式数据的折磨，在嵌套的 div 中对它进行包装，使用 `document.write` 等过时的技术将“模板”注入到 DOM 中。因为通常这意味着要在标准标记里保持脚本标记，很快就会变得难以阅读，更重要的是，这样的灾难会持续，尤其是在创建代码量很大的应用程序时。

JavaScript 模板解决方案（如 Handlebars.js 和 Mustache）通常是用于将 View（视图）模板定义为包含模板变量的标记（或者是存储于外部，或者是使用自定义类型的 script 标签，如 `text/template`）。可以使用变量语法对变量进行定界（例如`{{name}}`），框架通常非常智能，可以接受 JSON 形式的数据（Model 模型实例可以被转换成这种形式），这样的话，我们则只需要关注保持整洁的 Model（模型）和模板。大部分数据绑定工作都是由框架自身来完成，这有很多好处，尤其是选择在外部存储模板时，因为构建更大的应用程序时，它可以让路给按需动态加载的模板。

这里，我们可以看到 HTML 模板的两个示例（示例 10-1 和示例 10-2）。一个使用流行的 Handlebars.js 框架来实现，另一个则是使用 Underscore 的模板来实现。

### 示例 10-1 Handlebars.js 代码

```
<li class="photo">  
  <h2>{{caption}}</h2>  
    
  <div class="meta-data">  
    {{metadata}}  
  </div>  
</li>
```

### 示例 10-2 Underscore.js 微型模板

```
<li class="photo">  
  <h2><%= caption %></h2>  
  
```

```
<div class="meta-data">
<%= metadata %>
</div>
</li>
```

注意，模板本身并不是 View（视图）。使用 StrutsModel 2 架构的开发人员可能感觉模板就是 View（视图），但实际上不是。View（视图）是一个用于检测 Model（模型）并保持可视化表示更新的对象。模板可能是一种指定部分或甚至所有 View（视图）对象的声明方式，这样它就可以从模板规范中生成。

值得一提的是，在经典的 web 开发中，在独立 View（视图）之间导航需要使用页面刷新。但在单页面的 JavaScript 应用程序中，一旦通过 Ajax 从服务器获取数据，数据可以只在同一页面的新 View（视图）中动态呈现，而不需要任何此类刷新。因此导航的角色就落到了路由身上，它协助管理应用程序状态（如，允许用户将他们已导航到的特定 View（视图）加入书签）。由于路由既不是 MVC 的一部分，也不会出现在每一个 MVC 的框架中，因此本节没有对其进行更详细地介绍。

简而言之，View（视图）是应用程序数据的可视化表示。

### 10.2.3 Controller（控制器）

Controller（控制器）是 Model（模型）和 View（视图）之间的中介，当用户操作 View（视图）时，它通常负责更新 Model（模型）。

在图片库应用程序中，Controller（控制器）将负责处理用户对特定图片 View（视图）的编辑更改，当用户完成编辑后，更新一个特定的图片 Model（模型）。

请记住，Controller（控制器）在 MVC 中扮演一个角色：View（视图）策略模式的简易化。在策略模式方面，View（视图）在其判断力下将委托给 Controller（控制器）进行操作，这是策略模式的工作原理。当 View（视图）认为合适时，View（视图）可以委托 Controller（控制器）来处理用户事件。当 View（视图）认为合适时，View（视图）也可以委托 Controller（控制器）处理 Model（模型）更改事件，但这不是 Controller（控制器）原来的角色。

在大多数 JavaScriptMVC 框架中，与我们通常所认为的“MVC”相比最逊色的地方，就是它具有 Controller（控制器）。其原因各有不同，但以我的真实想法，是因

为框架作者最初看到的是 MVC 的服务器端解释，意识到它在客户端并不是 1:1 转变，他们觉得重新解释 MVC 中的 C 更有意义。但问题是，这是很主观的，会增加对理解经典 MVC 模式和 Controller（控制器）在现代框架中的角色的复杂性。

例如，让我们简要地回顾一下流行架构框架 Backbone.js 的架构。Backbone 包含 Model（模型）和 View（视图）（有点类似于我们前面看过的 content）；但它实际上并没有真正的 Controller（控制器）。其 View（视图）和路由的行为与 Controller（控制器）有点类似，但它们实际上都不是 Controller（控制器）。

在这方面，可能与在官方文档或博客帖子中提到的内容相反，Backbone 既不是真正的 MVC/MVP，也不是 MVVM 框架。实际上把它视为以自己的方式来架构的 MV\* 家族的一个成员会更好。当然这样做也没有错，但重要的是要区分经典 MVC 和 MV\*，我们应当开始依靠前者经典著作的建议，来帮助后者。

## 10.2.4 Spine.js 与 Backbone.js

### 10.2.4.1 Spine.js

我们现在知道，当用户更新 View（视图）时，习惯上 Controller（控制器）会负责更新 Model（模型）。有趣的是，在编写时，最流行的 JavaScript MVC/MV\* 框架（Backbone）的 Controller（控制器）没有自己明确的概念。

因此我们可以从另一个 MVC 框架来审视 Controller（控制器），以理解实现方面的差别，并进一步演示非传统框架如何接手 Controller（控制器）的角色。为此，让我们来看一下有关 Spine.js 的示例 Controller（控制器）。

在本示例中，有一个叫做 PhotosController 的 Controller（控制器），它将负责管理应用程序中的单个图片。这将确保在 View（视图）更新时（如用户编辑图片元数据时），相应的 Model（模型）也相应更新。

我们不会深入研究 Spine.js，只是浅尝辄止的研究它的 Controller（控制器）如何完成下列工作。

```
// 在 Spine 中，通过继承 Spine.Controller 创建控制器
```

```
var PhotosController = Spine.Controller.sub({
  init: function () {
    this.item.bind("update", this.proxy(this.render));
    this.item.bind("destroy", this.proxy(this.remove));
  },
  render: function () {
    // 处理模板
    this.replace($("#photoTemplate").tmpl(this.item));
    return this;
  },
  remove: function () {
    this.el.remove();
    this.release();
  }
});
```

在 Spine 中，Controller（控制器）被认为是应用程序的粘合剂，添加及响应 DOM 事件，渲染模板，并确保 View（视图）和 Model（模型）保持同步（在我们知道它是 Controller（控制器）的情况下是合理的）。

在上面的示例中，我们所做的工作是使用 render() 和 remove() 在 update 和 destroy 事件中设置监听器。当图片项更新时，重新渲染 View（视图）来反映这些元数据的变化。同样，如果这张图片从图片库中删除，要从 View（视图）中把它删除。在 render() 函数中，使用 Underscore 微型模板（通过\_.template()）来呈现一个 ID 为#photoTemplate 的 JavaScript 模板。它只是返回编译后的 HTML 字符串，用于填充 photoEl 的内容。

它为我们提供的是一个轻量级的简单方式，来管理 Model（模型）和 View（视图）之间的变化。

#### 10.2.4.2 Backbone.js

在本节的后面部分，我们将回顾 Backbone 和传统 MVC 之间的区别，但是现在，让我们把注意力集中在 Controller（控制器）上。

在 Backbone 中，Backbone.View 和 Backbone.Router 一起承担 Controller（控制器）的责任。之前，Backbone 确实有它自己的 Backbone.Controller，但这个组件的命名在它使用的上下文中没有意义，于是后来改名为 Router。

Router 拥有更多一点的 Controller（控制器）责任，因为它可以为 Model（模型）

绑定事件，使我们的 View（视图）响应 DOM 事件和 DOM 渲染。正如 Tim Branyen（另一个来自 Bocoup 的 Backbone 贡献者）先前所指出的，它可以不需要 Backbone.Router，所以使用路由范式来思考它的方法可能是：

```
var PhotoRouter = Backbone.Router.extend({  
  routes: { "photos/:id": "route" },  
  
  route: function (id) {  
    var item = photoCollection.get(id);  
    var view = new PhotoView({ model: item });  
  
    $('.content').html(view.render().el);  
  }  
});
```

总而言之，这一节的收获是：Controller（控制器）管理应用程序中 Model（模型）和 View（视图）之间的逻辑和协调。

## 10.3 MVC 为我们提供了什么

MVC 中的这种关注点分离有利于进一步简化应用程序功能的模块化，并能够实现：

- 整体维护更容易。数据中心是否改变，什么时候需要更新应用程序这点很清楚，这意味着是 Model（模型）或者也可能是 Controller（控制器）的变化，或者仅仅是视觉，这意味着 View（视图）的改变。
- 解耦 Model（模型）和 View（视图），意味着它能够更直接地编写业务逻辑的单元测试。
- 在整个应用程序中，底层 Model（模型）和 Controller（控制器）代码的重复（例如我们可能已经使用的）被消除了。
- 取决于应用程序的大小和角色的分离，这种模块性可以让负责核心逻辑的开发人员和负责用户界面的开发人员同时工作。

## 10.4 JavaScript 中的 Smalltalk-80 MVC

虽然大多数的现代 JavaScript 框架试图向 MVC 范式发展，以便更好地适应 web 应

用程序开发的不同需求，在 Smalltalk-80 中发现有一种框架，试图坚持纯粹形式的模式。Peter Michaux 编写的 Maria.js (<https://github.com/petermichaix/maria>) 提供了一个忠于 MVC 最初形式的实现：Model 是模型、View 是视图，Controller 只是控制器。而有些开发人员可能会觉得 MV\* 框架应该能够解决更多的问题，如果你选择使用最初 MVC 的 JavaScript 实现，这是一个很有用的参考。

### 10.4.1 深入挖掘

阅读到本书的这个部分，我们应该对 MVC 模式所提供的内容有了基本的了解，但是这里仍然有一些有趣的信息值得我们注意。

GoF（四人组）没有将 MVC 作为一种设计模式，而认为它是一系列用于构建用户界面的类。在他们看来，MVC 实际上是三个经典设计模式的变体：Observer（观察者）模式、Strategy（策略）模式及 Composite（组合）模式。根据 MVC 在框架中的实现方式，它也可以使用 Factory（工厂）模式和 Template（模板）模式。“四人组”的书提到，在使用 MVC 时，这些模式是很有用的额外模式。

正如我们已经讨论过的，Model（模型）表示应用程序数据，而 View（视图）表示在屏幕上向用户显示了什么内容。因此，MVC 依赖于 Observer（观察者）模式来实现它的一些核心通信（令人惊讶的是，很多有关 MVC 模式的文章都没有涉及到）。当 Model（模型）被改变时，它通知其观察者（View）一些内容已经更新，这也许是在 MVC 中最重要的关系。这种关系的观察者本质上也是促进多个 View（视图）被附加到同一个 Model（模型）中的因素。

对于有兴趣了解更多有关 MVC 的自然解耦的开发人员来说（再次提醒，取决于实现），模式的其中一个目标是帮助定义主题 topic 和它的观察者之间的一对多关系。当主题改变时，它的观察者也会更新。View（视图）和 Controller（控制器）有一个稍微不同的关系。Controller（控制器）帮助 View（视图）应对不同的用户输入，是策略模式的榜样。

### 10.4.2 总结

在回顾了经典 MVC 模式后，我们现在应该了解了如何在应用程序中整洁地分离关注点。也应该了解了 JavaScript 的 MVC 框架在解释 MVC 模式方面如何不同，虽然非常容易变化，但依然要分享一下原始模式所必须有的一些基本概念。

评估一个新的 JavaScript MVC/MV\* 框架时, 请记住: 它可以用于退后, 以及审查如何接近架构(具体地说, 它是如何支持实现模型、视图、控制器或其他方法的), 因为这可以更好的帮助我们理解应如何使用框架。

## 10.5 MVP

模型-视图-表示器(MVP)是MVC设计模式的一种衍生模式, 专注于改进表示逻辑。它是在1990年代由Telligent公司(<http://en.wikipedia.org/wiki/Telligent>)创造的, 当时他们正在研究一个用于C++ CommonPoint环境的模型。

MVC和MVP在多个组件中都有关注点分离目标, 但两者之间有一些基本的区别。

出于总结的目的, 我们将关注最适合用于web架构的MVP版本。

### 10.5.1 Model、View 和 Presenter

MVP中的P代表表示器。这是一个包含用于View(视图)的用户界面业务逻辑的组件。与MVC不同, 来自View(视图)的调用将委托给表示器, 表示器是从View(视图)中解耦, 通过接口与它对话。这允许我们做各种有用的事情, 如可以在单元测试中模拟View(视图)(见图10-2)。

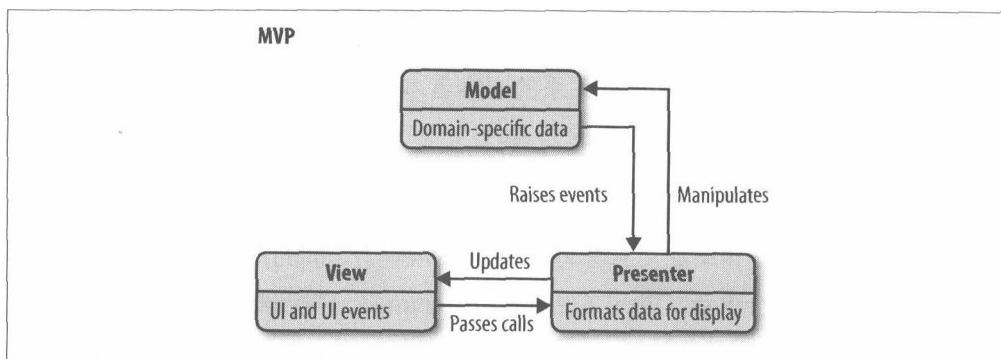


图 10-2 MVP 模式

最常见的MVP实现是使用被动View(视图)(从各方面讲, 它都是“哑”的)的MVP, 包含一点或零逻辑。如果MVC和MVP是不同的, 这是因为C和P完成不同的事情。在MVP中, 当Model(模型)变化时, 监控Model(模型)和更新View

(视图)。P 将 Model (模型) 有效地绑定至 View (视图)，这是以前在 MVC 中 Controller (控制器) 的责任。

由 View (视图) 进行请求，表示器执行任何与用户请求有关的工作，并将数据回传给它们。在这方面，它们检索数据并操作数据，并确定应该如何在 View (视图) 中显示这些数据。在有些实现中，表示器还与服务层交互来持久化数据 Model (模型)。Model (模型) 可能会触发事件，表示器的角色是订阅它们，这样就可以更新 View (视图)。在这种被动架构中，没有直接数据绑定的概念。View (视图) 暴露了 setter 设置器，表示器可以用它来设置数据。

MVC 这种变化的好处是它能够提高应用程序的可测试性，并在 View (视图) 和 Model (模型) 之间提供更清晰的分离。然而这并不是没有代价的，由于在这种模式中缺乏数据绑定支持，往往意味着不得不单独关注这个任务。

虽然被动视图 (<http://martinfowler.com/eaaDev/PassiveScreen.html>) 的常见实现是用于实现一个接口的 View (视图)，这里有些变化，包括将 View (视图) 从表示器进一步解耦的事件使用。由于我们在 JavaScript 中没有接口，在这里我们使用协议而不是显式接口。在技术上这仍然是一个 API，从这个角度来看，我们称它为接口也可能是公平的。

还有一个 MVP 的监管控制器 (<http://martinfowler.com/eaaDev/SupervisingPresenter.html>) 变体，这与 MVC 和 MVVM 模式 ([http://en.wikipedia.org/wiki/Model\\_View\\_ViewModel](http://en.wikipedia.org/wiki/Model_View_ViewModel)) 较为接近，因为它直接在 View (视图) 上提供 Model (模型) 的数据绑定。键值检测 (KVO) 插件 (如 Derick Bailey 的 Backbone.ModelBinding 插件) 往往会将 Backbone 从被动 View (视图) 中引出，进而引入监督控制器或 MVVM 变体中。

## 10.5.2 MVP 或 MVC?

MVP 最常用于企业级应用程序，这种应用程序需要尽可能多地重用表示逻辑。具有非常复杂的 View (视图) 和大量用户交互的应用程序可能发现 MVC 在这里并不完全符合要求，因为解决这个问题可能意味着要极度依赖于多个控制器。在 MVP 中，所有这些复杂的逻辑可以封装在一个表示器中，这可以大大简化维护工作。

由于 MVP 里的 View (视图) 是通过接口被定义的，从技术上讲，接口是系统和

**View**（视图）（而不是表示器）之间唯一的接触点，这种模式可以让开发人员能够编写表示逻辑，而无需等待设计师为应用程序设计布局和图形。

根据不同的实现，使用 MVP 进行自动化单元测试可能会比 MVC 更容易。经常被测试的原因是，表示器可以被当作一个完整的用户界面模拟，因此它可以独立于其他组件而进行单元测试。以我的经验，这的确取决于我们实现 MVP 的语言（为 JavaScript 项目选择 MVP，与 ASP.NET 实现的 MVP 之间有很大的差异）。

最后，考虑到它们之间的差别主要是语义上的，我们对 MVC 的潜在担忧对 MVP 来说可能是有效的。只要能够清晰地分离 Model（模型）、View（视图）和 Controller（控制器）（或表示器）的关注点，我们就应该能够实现大多数相同的受益，不管我们所选择的模式是什么。

### 10.5.3 MVC、MVP 和 Backbone.js

只有很少的架构式 JavaScript 框架要求以经典形式实现 MVC 或 MVP 模式，因为很多 JavaScript 开发人员认为 MVC 和 MVP 不是互斥的（实际上在查看 ASP.NET 或 GWT 等 web 框架时，我们更可能看到的是 MVP 被严格实现了）。这是因为在应用程序中额外的表示器/View（视图）逻辑是可能存在的，但它仍然是 MVC 的风格。

来自波士顿在 Bocoup 任职的 Backbone 贡献者 Irene Ros (<http://ireneros.com/>) 认同这种思维方式，因为当她将 View（视图）分离至它们自己的不同组件中时，她需要一些能够对它们进行组装的东西。它可以是 Controller（控制器）路由（如本书稍后将讨论的 Backbone.Router），或者响应获取数据的回调。

也就是说，与 MVC 相比，一些开发人员确实认为 Backbone.js 更符合 MVP 的描述。他们的观点是：

- 与 Controller（控制器）相比，MVP 中的表示器更好地描述了 Backbone.View（View 模板和绑定至模板的数据之间的层）
- 该 Model（模型）适合 Backbone.Model（它与 MVC 中的 Model（模型）没有什么大的不同）
- View（视图）最能表示模板（如 Handlebars/Mustache 标记模板）

对它们的回答是：View 也可以只是 View（根据 MVC），Backbone 非常灵活，可以用于多种目的。MVC 中的 V 和 MVP 中的 P 都可以通过 Backbone.View 来完成，因为它们能够达到两个目的：两者都呈现原子组件并装配那些被其他 View（视图）渲染的组件。

我们还了解到，在 Backbone 中，Controller（控制器）的责任与 Backbone.View 和 Backbone.Router 共享，在下面的示例中，我们可以了解这些方面都是真实的。

Backbone PhotoView 使用 Observer 模式来“订阅”ViewModel（视图模型）的变化（见 this.model.bind ("change" ,...) 代码行）。它还处理 render()方法中的模板，但与其他实现不同的是，用户交互也在 View 中进行处理（参见 events）。

```
var PhotoView = Backbone.View.extend({  
    //... 此处省略。  
    tagName: "li",  
  
    // 通过模板函数将 photo 模板的内容传递进来，为单个 photo 对象进行缓存  
    template: _.template($("#photo-template").html()),  
  
    // 给一个项指定 DOM 事件  
    events: {  
        "click img": "toggleViewed"  
    },  
  
    // PhotoView 监听模型的改变，重写呈现。鉴于在**Photo** and a **PhotoView**  
    // 之间只是一对一的关系，为了方便，我们在模型上设置了直接引用  
  
    initialize: function () {  
        this.model.on("change", this.render, this);  
        this.model.on("destroy", this.remove, this);  
    },  
  
    // 重写呈现 photo 实体  
    render: function () {  
        $(this.el).html(this.template(this.model.toJSON()));  
        return this;  
    },  
  
    // 反转模型的 viewed 状态  
    toggleViewed: function () {  
        this.model.viewed();  
    }  
});
```

另一种(完全不同的)观点是:Backbone更像我们前面看到的Smalltalk-80MVC(<http://martinfowler.com/eaaDev/uiArchs.html#ModelViewController>)。

鉴于 Backbone 博客 Derick Bailey 曾提出的观点 (<http://lostechies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework/>), 最好不要强迫 Backbone 来匹配任何特定的设计模式。设计模式应该被认为是有关如何构建应用程序的灵活指南, 在这方面, Backbone 既不适合 MVC, 也不适合 MVP。反而, 它借用多个架构模式中一些最好的概念, 并创建一个运行良好的灵活框架。

但值得了解的是, 这些概念起源于哪里以及产生的原因, 所以希望我对 MVC 和 MVP 的解释会有所帮助。采用 Backbone 方式的 MV\*有助于其应用程序架构的风格。大多数结构型 JavaScript 框架都在经典模式上采取自己的实现, 不管是有意或意外, 但重要的是, 它们能够帮助我们开发有组织性、整洁并易于维护的应用程序。

## 10.6 MVVM

MVVM (模型-视图-视图模型) 是一种基于 MVC 和 MVP 的架构模式, 它试图更清晰地将用户界面 (UI) 开发从应用程序的业务逻辑与行为中分离。为此, 很多这种模式的实现都要利用声明式数据绑定来实现将 View (视图) 工作从其他层分离。

这有助于在同一个代码库中 UI 和开发工作的同时进行。UI 开发人员在其文档标记 (HTML) 内编写到 ViewModel 的绑定, 其中的 Model 和 ViewModel 都是由研究应用程序逻辑的开发人员来进行维护 (见图 10-3)。

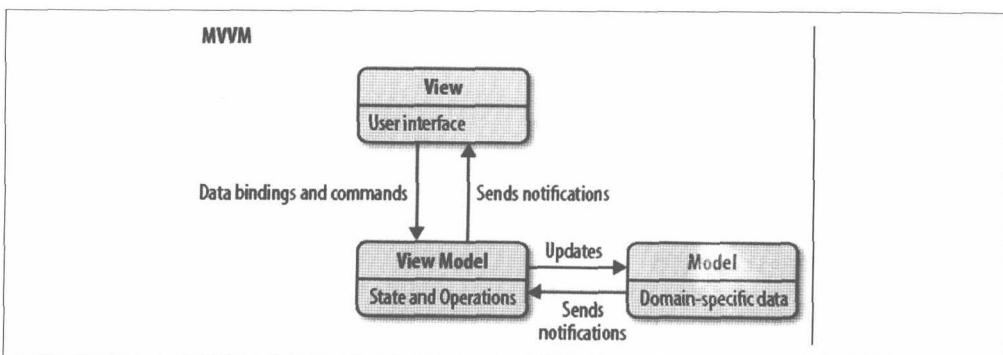


图 10-3 MVVM 模式

## 10.6.1 历史

MVVM(按名称)最初是由微软在使用 Windows Presentation Foundation (WPF ([http://en.wikipedia.org/wiki/Windows\\_Presentation\\_Foundation](http://en.wikipedia.org/wiki/Windows_Presentation_Foundation))) 和 Silverlight (<http://www.microsoft.com/silverlight/>) 时定义, 2005 年 John Grossman (<http://blogs.msdn.com/b/johngrossman/>) 在一篇关于阿瓦隆 (WPF 的代号) 的博客文章中正式宣布了它的存在。它在 Adobe Flex 社区中也受到了一定的欢迎, 它是在只能使用 MVC 情况下的另一种替代方法。

在微软采用 MVVM 这个名称之前, 在社区中发生了一场有关从 MVP 转向 MVPM 的运动: Model View PresentationModel ([http://blogs.adobe.com/paulw/archives/2007/10/presentation\\_pa\\_3.html](http://blogs.adobe.com/paulw/archives/2007/10/presentation_pa_3.html))。Martin Fowler 面向对它感兴趣的读者, 在 2004 年写了一篇有关 PresentationModels 的文章 (<http://martinfowler.com/eaaDev/PresentationModel.html>)。在写本文之前, PresentationModel ([http://www.infragistics.com/community/blogs/craig\\_shoemaker/archive/2009/11/03/learning-model-view-model-and-presentation-model.aspx](http://www.infragistics.com/community/blogs/craig_shoemaker/archive/2009/11/03/learning-model-view-model-and-presentation-model.aspx)) 的想法就已经存在很久了, 但它被认为是极大的思想突破, 并极大地助力了 MVPM 的推广。

在微软宣布 MVVM 为 MVPM 的替代方法后, 在“alt.net”圈引起了相当大的骚动。很多人称微软在 GUI 世界的主导地位为他们提供了接管整个社区的机会, 出于营销的目的, 只要他们乐意, 可以重命名现有的概念。一些进步的人群认为, MVVM 和 MVPM 实际上是相同的想法, 只是封装稍有不同。

近年来, MVVM 一直是以结构框架的形式在 JavaScript 中实现, 如 KnockoutJS (<http://knockoutjs.com/>)、Kendo MVVM (<http://www.kendoui.com/web/roadmap.aspx>) 和 Knockback.js (<https://github.com/kmalakoff/knockback>), 整个社区对此反映都很积极。

现在让我们来回顾组成 MVVM 的这三个组件。

## 10.6.2 Model

和 MV\* 家族的其他成员一样, MVVM 中的 Model (模型) 表示应用程序将会使用的特定领域数据或信息。有关特定领域数据的典型示例可能是一个用户帐户 (例如名字、头像、电子邮件) 或一个音乐带 (例如标题、年、专辑)。

Model（模型）保存着信息，但是通常不处理行为。它们不会格式化信息或影响数据在浏览器中显示的方式，因为这不是它们的责任。数据格式化是由 View 来处理的，而行为被认为是业务逻辑，应该封装在与 Model（模型）交互的另一层中：ViewModel。

这个规则的唯一例外是验证，对于 Model 来说是可接受的，以验证用于定义或更新现有 Model（模型）的数据（例如，输入的电子邮件地址是否满足特定正则表达式的要求？）。

在 KnockoutJS 中，Model 被归入上述定义，但通常会使用 Ajax 调用服务器端的服务来进行 Model（模型）数据的读写。

要构建一个简单的 Todo 应用程序，代表单个 Todo 项的 KnockoutJSModel（模型）可能会如下所示：

```
var Todo = function (content, done) {
    this.content = ko.observable(content);
    this.done = ko.observable(done);
    this.editing = ko.observable(false);
};
```

你可能会注意到在上面的代码片段中，我们在 KnockoutJS 命名空间 ko 上调用了 observables()方法。在 KnockoutJS 中，observables 是特殊的 JavaScript 对象，可以通知用户有关变化并自动检测依赖。这样在 Model 属性的值被修改时，就可以使 Model 和 ViewModel 保持同步。

### 10.6.3 View

与 MVC 一样，View 实际上仅是与用户进行交互的应用程序的一部分。它们是一个交互式 UI，描绘 ViewModel 的状态。从这个意义上说，View（视图）被认为是主动而不是被动的，对于 MVC 和 MVP 中的 View（视图）这也是正确的。在 MVC、MVP 和 MVVM 中，View（视图）也可以是被动的，但这意味着什么呢？

被动 View 只输出显示，并不接收任何用户输入。这种 View（视图）在应用程序中，可能也没有真正的 Model（模型）知识，并且可以被表示器控制。MVVM 的主动 View 包含数据绑定、事件和行为，需要对 ViewModel 有了解。虽然这些行为可以被映射到属性，但 View 仍负责处理 ViewModel 的事件。

重要的是要牢记，View 并不负责处理状态；它仅仅是让状态与 ViewModel 保持同步。

KnockoutJSView 只是一个具有声明式绑定的 HTML 文档，它将绑定到 ViewModel。KnockoutJSView 显示来自 ViewModel 的信息，向它传递命令（例如，用户点击一个元素）并在 ViewModel 的状态变化时进行更新。使用 ViewModel 数据生成 HTML 的模板也可以用于这一目的。

通过一个简短的初始示例，我们可以了解 JavaScript MVVM 框架 KnockoutJS 如何定义 ViewModel 以及在标记中 ViewModel 的相关绑定：

如下是用于 ViewModel 的代码：

```
var aViewModel = {  
    contactName: ko.observable("John")  
};
```

如下是用于 View 的代码：

```
<input id="source" data-bind="value: contactName, valueUpdate: "keyup" /></p>  
  
<div data-bind="visible: contactName().length >10">  
    You have a really long name!  
</div>
```

文本输入框（ID 为 source）从 contactName 获得它的初始值，当 contactName 更改时，文本框的值会自动更新。由于数据绑定是双向的，向文本框输入内容时，同样也将相应地更新 contactName 的值，因此这两个值始终是保持同步的。

该实现是特定于 KnockoutJS，而且包含“*You have a really long name!*（你有一个很长的名字！）”的<div>还包含简单的验证（再次以数据绑定的形式）。如果输入文本超过 10 个字符，它就会显示；否则，它将保持隐藏。

回到 Todo 应用程序来看一个更高级的示例。精简 KnockoutJS View（视图），包括所有必要的数据绑定，代码如下所示。

```
<div id="todoapp">  
    <header>  
        <h1>Todos</h1>  
        <input id="new-todo" type="text" data-bind="value: current,  
            valueUpdate: 'afterkeydown', enterKey: add"  
            placeholder="What needs to be done?" />  
    </header>  
    <section id="main" data-bind="block: todos().length">
```

```

<input id="toggle-all" type="checkbox" data-bind="checked: allCompleted">
<label for="toggle-all">Mark all as complete</label>

<ul id="todo-list" data-bind="foreach: todos">

    <!-- item -->
    <li data-bind="css: { done: done, editing: editing }">
        <div class="view" data-bind="event: { dblclick: $root.editItem }">
            <input class="toggle" type="checkbox" data-bind="checked: done">
            <label data-bind="text: content"></label>
            <a class="destroy" href="#" data-bind="click: $root.remove"></a>
        </div>
        <input class="edit" type="text"
               data-bind="value: content, valueUpdate: 'afterkeydown',
               enterKey: $root.stopEditing, selectAndFocus: editing,
               event: { blur: $root.stopEditing }"/>
    </li>
</ul>

</section>
</div>

```

注意，这个 HTML 的基本布局相对比较直观，包含一个用于添加新项目的文本输入框（new-todo），将所有条目标记为完成的开关标记，以及 Todo 列表，Todo 列表内包含多个 li 元素，每个 li 都包含一个可绑定 Todo 项的模板。

上面 HTML 中的数据绑定可以被分解如下：

- 文本输入框 new-todo 具有 current 属性的数据绑定，这里是所添加当前项的值的存储位置。ViewModel（稍后介绍）观察 current 属性，并且与 add 事件也有对应的绑定。按下回车键时，add 事件被触发，然后 ViewModel 可以去除 current 值前后的空格，并将它添加到 Todo 列表中。
- 如果点击输入复选框 toggle-all，所有的当前项都可以标记成已完成。勾选此项，就会触发 ViewModel 中的 allCompleted 事件。
- li 项有一个 done class。当一个 task 的 class 被标记为 done 时，相应的 class editing 也会生效。如果双击该项，将执行 \$root.editItem 回调。
- checkbox 复选框具有 toggle class，用于显示 done 属性的状态。
- 包含 Todo 项（content）的文本值的标签 Label。

- 还有一个删除按钮，点击它时，将调用\$root.remove 回调。
- 用于编辑模式下的文本输入框还保存 Todo 项 content 的值。enterKey 事件将设置 editing 属性为 true 或 false。

#### 10.6.4 ViewModel

可以将 ViewModel 作为一个专门的 Controller，充当数据转换器。它将 Model 信息转变为 View 信息，还将命令从 View 传递到 Model。

例如，假设我们有一个包含 date 属性的 unix 格式模型（如 1333832407）。Model（模型）不需要知道用户的日期 View（视图）（如 04/07/2012 @ 5:00pm），这里只是将地址转变为它的显示格式即可，Model（模型）仅保存原始格式的数据。View 包含格式化后的日期，ViewModel 充当两者之间的中间人。

在这个意义上，ViewModel 可能被看作为 Model，而不是 View，但它可以处理大部分的 View 显示逻辑。ViewModel 可能还会暴露一些方法，用于帮助保持 View 的状态，基于 View 上的操作来更新 Model（模型），并触发 View 上的事件。

总之，ViewModel 位于 UI 层的后面。它暴露了 View 所需的数据（从 Model 那里），可以被视为 View 数据和操作的源头。

KnockoutJS 将 ViewModel 解释为可以在 UI 上执行的数据和操作的表示。它不是 UI 本身，也不是数据 Model（模型），而是一个用于保存用户正在使用且尚未保存数据的层。Knockout 的 ViewModel 可以在不需要 HTML 的情况下由 JavaScript 对象来实现。这个用于实现的抽象方法可以让它们保持简单，这意味着可以根据需要更容易地管理更复杂的行为。

因此 Todo 应用程序的部分 KnockoutJS ViewModel 可能如下所示：

```
// 主要 ViewModel
var ViewModel = function ( todos ) {
    var self = this;

    // 将传递的数组转化到 Todo 对象的数组
    self.todos = ko.observableArray(
        ko.utils.arrayMap( todos, function ( todo ) {
            return new Todo( todo.content, todo.done );
        })
    );
}
```

```

});;
// 临时保存新输入的 todo
self.current = ko.observable();
// 回车时，添加新输入的 todo
self.add = function ( data, event ) {
    var newTodo, current = self.current().trim();
    if (current) {
        newTodo = new Todo( current );
        self.todos.push( newTodo );
        self.current("");
    }
};
// 删除单个 todo
self.remove = function ( todo ) {
    self.todos.remove( todo );
};

// 删除所有已完成的 todo
self.removeCompleted = function () {
    self.todos.remove(function (todo) {
        return todo.done();
    });
};

// 可写的 computed 监控，用于标记所有已完成/未完成的 todo
self.allCompleted = ko.computed({
    // 基于所有 todo 的 done 标记，永远返回 true/false
    read: function () {
        return !self.remainingCount();
    },
});

// 设置所有的 todo 于新值 (true/false)
write: function ( newValue ) {
    ko.utils.arrayForEach( self.todos(), function ( todo ) {
        // 即便 value 值不变，也要设置，因为这种情况下，不会通知订阅者的
        todo.done( newValue );
    });
};

// 编辑一个 todo 项
self.editItem = function( item ) {
    item.editing( true );
};

..

```

在这里，我们基本上提供了添加、编辑或删除项目所需的方法，以及标记所有剩余项目为已完成的逻辑。需要注意的是，与前面的 `ViewModel` 示例相比唯一的差别是可观察的数组。在 KnockoutJS 中，如果我们希望探测和响应单个对象的变化，我们会使用 `observables`。但如果我们希望探测和响应一组事物的变化，我们可以使用 `observableArray`。有关如何使用可观察数组的简单示例，如下所示：

```
// 定义初始化的空数组
var myObservableArray = ko.observableArray();

// 添加一个值到数组上，并且通知观察者
myObservableArray.push('A new todo item');
```

完整的 Knockout.js Todo 应用程序示例可以在 TodoMVC (<http://todomvc.com/>) 中找到。

### 10.6.5 小结：View 和 ViewModel

View 和 ViewModel 之间通过数据绑定和事件进行通信。正如 ViewModel 初始示例中所示，ViewModel 不只暴露 Model 属性，还会访问其他方法和验证类的特性。

View 处理自己的用户界面事件，必要时将它们映射到 ViewModel。Model 和 ViewModel 上的属性通过双向数据绑定进行同步和更新。

触发器（数据触发器）也可以使我们进一步地对 Model 属性的状态变化做出反应。

### 10.6.6 小结：ViewModel 和 Model

ViewModel 似乎是完全负责 MVVM 中的 Model，但这种关系中有一些微妙之处值得我们注意。ViewModel 可以为数据绑定而暴露 Model 或 Model 属性，也可以包含接口，用于获取和操作在 View 中暴露的属性。

## 10.7 利与弊

我们现在希望能够更好地了解什么是 MVVM 以及它的工作原理。现在让我们回顾一下采用这种模式的优点和缺点。

### 10.7.1 优点

- MVVM 使得 UI 和为 UI 提供驱动的行为模块的并行开发变得更容易。
- MVVM 使 View 抽象化，从而减少代码背后所需的业务逻辑量。
- ViewModel 在单元测试中的使用比在事件驱动代码中的使用更加容易。
- 不需要考虑 UI 自动化和交互就可以测试 ViewModel（更多时候是 Model，而不是 View）。

## 10.7.2 缺点

- 对于简单的 UI 来说，使用 MVVM 有些大材小用。
- 数据绑定可以是声明性的，使用也很方便，但比命令式代码更难调试，在命令式代码中，我们只需设置断点。
- 大型应用程序中的数据绑定可以产生大量的标记。我们不想看到的情况是：绑定比被绑定的对象还要繁重。
- 在较大型应用程序中，预先设计大量的 ViewModel 可能更加困难。

## 10.8 使用更松散数据绑定的 MVVM

具有 MVC 或 MVP 开发背景的 JavaScript 开发人员，审查 MVVM 并抱怨它的真实关注点的分离并不罕见。也就是说，大量的内联数据绑定仍存在于 View 的 HTML 标记中。

我必须承认，当我第一次审查 MVVM（如 KnockoutJS、Knockback）实现时，令我感到惊讶的是，任何开发人员都想要回到那个将逻辑（JavaScript）与标记混合在一起的旧时光，并发现它很快就变得不可维护。但现实是，MVVM 这样做，是出于很多有说服力的理由（我们已经介绍过），包括帮助设计师更容易地在标记上绑定逻辑。

对于我们中的纯粹主义者来说，由于自定义绑定 provider 这一特性的出现，你会很高兴地了解到，我们现在可以大大减少对数据绑定的依赖，这个特性是在 KnockoutJS 1.3 中推出，并可用于所有版本。

KnockoutJS 默认情况下具有一个数据绑定 provider，用于寻找任何具有 `data-bind` 属性的元素，如在下例中所示。

```
<input id="new-todo" type="text" data-bind="value: current, valueUpdate:  
"afterkeydown",  
enterKey: add" placeholder="What needs to be done?" />
```

当 provider 查找一个拥有该属性的元素时，provider 对其进行解析，然后使用当前

数据上下文将它转变成 binding 对象。这是 KnockoutJS 一贯的工作方式，使我们能够以声明式地将绑定添加至元素，KnockoutJS 在这一层将该元素绑定至数据。

一旦开始创建不再是微不足道的 View 时，我们可能会遇到大量的元素和属性 (attribute)，它们在标记中的绑定会变得难以管理。但使用自定义绑定 provider 后，它就再也不是问题了。

绑定 provider 最感兴趣的两件事：

- 如果给定一个 DOM 节点，它是否包含任何数据绑定？
- 如果节点满足第一点，绑定对象在当前的数据上下文中看起来像是什么？

绑定 provider 实现两个功能：

#### *nodeHasBindings*

接受一个 DOM 节点，但该 DOM 节点并不一定要有元素。

#### *getbindings*

返回一个对象，该对象表示应用于当前数据上下文的绑定。

该自定义绑定 provider 的大体结构，如下所示：

```
var ourBindingProvider = {
    nodeHasBindings: function (node) {
        // 返回 true/false
    },
    getBindings: function (node, bindingContext) {
        // 返回绑定对象
    }
};
```

在开始细化这个 provider 之前，让我们先简要讨论一下数据绑定属性中的逻辑。

如果在使用 Knockout 的 MVVM 时发现自己对应用程序逻辑过分嵌入到 View 中的想法感到不满意，我们可以对其进行更改。我们将名称绑定赋值给元素，可以实现像 CSS 类这样的任务。Ryan Niemeyer (<http://www.knockmeout.net/>) 之前曾建议使用

`data-class`, 以避免混淆表示类和数据类, 因此我们要使 `nodeHasBindings` 函数支持:

```
// 元素有任何绑定么?
function nodeHasBindings(node) {
    return node.getAttribute ? node.getAttribute("data-class") : false;
};
```

接下来, 我们需要一个很有用的 `getBindings()` 函数。由于我们坚持采用 CSS 类的想法, 为什么不考虑像 CSS 的 `class` 支持空格分隔, 实现在不同元素之间分享绑定?

首先来回顾一下绑定将会是什么样子。我们创建一个对象来保存它们, 属性名称需要匹配我们希望在数据类中使用的键。

通过应用自定义绑定 provider, 将 KnockoutJS 应用程序从使用传统数据绑定的应用程序转变成 unobtrusive 绑定的应用程序不需要太多的工作量。我们只需拉出所有的 `data-bind` 属性, 用 `data-class` 属性替换它们, 然后按照如下方式将绑定放入 `binding` 对象:

```
var viewModel = new ViewModel(todos || []),
    bindings = {

        newTodo: {
            value: viewModel.current,
            valueUpdate: "afterkeydown",
            enterKey: viewModel.add
        },
        taskTooltip: { visible: viewModel.showTooltip },
        checkAllContainer: { visible: viewModel.todos().length },
        checkAll: { checked: viewModel.allCompleted },

        todos: { foreach: viewModel.todos },
        todoListItem: function () { return { css: { editing: this.
editing } }; },
        todoListWrapper: function () {
            return { css: { done: this.done } };
        },
        todoCheckBox: function () { return { checked: this.done }; },
        todoContent: function () { return { text: this.content, event:
{ dblclick: this.edit } }; },
        todoDestroy: function () { return { click: viewModel.remove }; },
        todoEdit: function () { return {

            value: this.content,
            valueUpdate: "afterkeydown",
            enterKey: this.stopEditing,
            event: { blur: this.stopEditing } };

        },
    };
};
```

```

todoCount: { visible: viewModel.remainingCount },
remainingCount: { text: viewModel.remainingCount },
remainingCountWord: function () {
    return { text: viewModel.getLabel(viewModel.remainingCount) };
},
todoClear: { visible: viewModel.completedCount },
todoClearAll: { click: viewModel.removeCompleted },
completedCount: { text: viewModel.completedCount },
completedCountWord: function () {
    return { text: viewModel.getLabel(viewModel.completedCount) };
},
todoInstructions: { visible: viewModel.todos().length }
};

....
```

但在上面的代码片段中有两行是缺失的：我们仍然需要 `getBindings` 函数，它将遍历 `data-class` 属性中的每个键，并创建每一个键的结果对象。如果我们检测到绑定对象是一个函数，通过 `this` 上下文使用当前数据调用该对象。完整的定制绑定 provider 如下所示：

```

// 可以创建 bindingProvider，使用不同于 data-bind 的属性
ko.customBindingProvider = function( bindingObject ) {
    this.bindingObject = bindingObject;

    // 判断元素是否有任何绑定
    this.nodeHasBindings = function( node ) {
        return node.getAttribute ? node.getAttribute( "data-class" ) : false;
    };
};

// 返回给定 node 节点和绑定上下文 (bindingContext) 的绑定
this.getBindings = function( node, bindingContext ) {

    var result = {},
        classes = node.getAttribute( "data-class" );

    if ( classes ) {
        classes = classes.split( " " );
        // 评估每个 class，构建单个对象返回
        for ( var i = 0, j = classes.length; i < j; i++ ) {

            var bindingAccessor = this.bindingObject[classes[i]];
            if ( bindingAccessor ) {
                var binding = typeof bindingAccessor ===
                    "function" ? bindingAccessor.call(bindingContext.$data) :
                bindingAccessor;
                ko.utils.extend(result, binding);
            }
        }
    }
}
```

```
        return result;
    };
};


```

因此，bindings 对象的最后几行代码可以被定义为：

```
// 设置 ko 的当前绑定 provider 为我们的自定义绑定 provider
ko.bindingProvider.instance = new ko.customBindingProvider( bindings );

// 绑定 ViewModel 的新实例到页面上
ko.applyBindings( viewModel );

})();
```

我们要做的是有效地定义绑定处理程序的构造函数，它接受一个对象（绑定），以便用于查找绑定。我们可以使用 `data-class` 重写应用程序 View 的标记，如下所示：

```
<div id="create-todo">
    <input id="new-todo" data-class="newTodo" placeholder=
        "What needs to be done?"/>
    <span class="ui-tooltip-top" data-class="taskTooltip" style=
        "display: none;">
        Press Enter to save this task</span>
</div>
<div id="todos">
    <div data-class="checkAllContainer">
        <input id="check-all" class="check" type="checkbox" data-
class=""
            checkAll"/>
        <label for="check-all">Mark all as complete</label>
    </div>
    <ul id="todo-list" data-class="todos">
        <li data-class="todoListItem">
            <div vclass="todo" data-class="todoListWrapper">
                <div vclass="display">
                    <input class="check" type="checkbox" data-class=
                        "to do CheckBox"/>
                    <div class="todo-content" data-class="todoContent" style=
                        "cursor: pointer;"> </div>
                    <span class="todo-destroy" data-class="todoDestroy"></span>
                </div>
                <div class="edit">
                    <input class="todo-input" data-class="todoEdit" />
                </div>
            </div>
        </li>
    </ul>
</div>
```

Neil Kerkin 使用上面的代码组合了一个完整的 TodoMVC 示例程序，可以点击[此处](#)

进行访问和了解。

前面的解释看起来可能需要很多工作，但现在由于我们编写了一个通用的 `getBindings` 方法，可以很简单地复用该方法了，并且使用 `data-class` 不需要使用严格的数据绑定来编写 KnockoutJS 应用程序。最终结果是使自定义数据绑定从 View 转移到绑定对象上，从而使绑定标记更加简洁。

## 10.9 MVC、MVP 与 MVVM

MVP 和 MVVM 均是 MVC 的衍生品。MVC 与其衍生品之间的主要区别是每一层对其他层的依赖，以及它们是如何紧密地互相绑定的。

在 MVC 中，View 位于架构之上，与 Controller（控制器）相邻。Model 位于 Controller（控制器）之下，因此 View 了解 Controller（控制器），Controller（控制器）了解 Model。在这里，View 能够直接访问 Model。但是，向 View 暴露完整的 Model 可能会带来安全性和性能成本，这取决于应用程序的复杂性。MVVM 尝试避免这些问题。

在 MVP 中，Controller（控制器）的作用被 Presenter 所替代。表示器与 View（视图）位于同一位置，监听 View 和 Model 的事件，并调解它们之间的行动。与 MVVM 不同，它没有使用将 View 绑定至 ViewModel 的机制，因此我们转而依赖每个 View 来实现用于让 Presenter 与 View 进行交互的接口。

因此，MVVM 允许我们创建 Model 的特定于 View 的子集，它们可以包含状态和逻辑信息，无需向 View 暴露整个 Model。与 MVP 的 Presenter 不同，引用 View 时不需要 ViewModel。View 可以绑定到 ViewModel 上的属性，而属性会将 Model 所包含的数据暴露给 View。如前所述，View 的抽象意味着它背后的代码所要求的逻辑更少了。

MVVM 其中的一个缺点是：在 ViewModel 和 View 之间需要进行解释，这会带来性能成本。解释的复杂性也是可以变化的：它可以像复制数据一样简单，或是像 View 所看到的那种形式进行操作一样复杂。MVC 没有这种问题，因为整个 Model 都是可用的，这种操作是可以避免的。

## 10.10 Backbone.js 与 KnockoutJS

了解 MVC、MVP 和 MVVM 之间的细微区别是非常重要的，但是开发人员最终会问到，基于我们所学，他们是否应该考虑使用 KnockoutJS，而不是 Backbone。以下几个要点在这里应该会有帮助。

- 这两个库都设有不同的目标，它通常不会像是选择 MVC 或 MVVM 那样简单。
- 如果数据绑定和双向沟通都是你的主要关注点，那么 KnockoutJS 一定是要选择的方向。实际上，存储在 DOM 节点中的任何属性或值都可以使用这种方法被映射到 JavaScript 对象。
- Backbone 的优势在于它易于与 RESTful 服务相集成，而 KnockoutJS Model 只是 JavaScript 对象，用于更新 Model 的代码必须由开发人员编写。
- KnockoutJS 重点关注自动化 UI 绑定，如果尝试使用 Backbone 来完成这项工作，就需要更详细的自定义代码。这对于 Backbone 本身不是问题，因为它有意尝试脱离 UI。但 Knockback 试图协助解决这个问题。
- 通过使用 KnockoutJS，我们可以将自己的函数绑定至 ViewModel observables，任何时候 observable 发生变化时，都会执行这些 observables。这使我们能够实现与在 Backbone 中相同级别的灵活性。
- Backbone 内置有一个强大的路由解决方案，而 KnockoutJS 却没有提供可用的路由选择。但如果需要，我们可以使用 Ben Alman 的 BBQ 插件(<http://benalman.com/projects/jquery-bbq-plugin/>) 或像 Miller Medeiros 的 Crossroads (<http://millermedeiros.github.com/crossroads.js/>) 这样的优秀独立路由系统来轻松地替代这一行为。

最后，我个人发现 KnockoutJS 更适用于小型应用程序，而在创建大型应用程序时，Backbone 的特性集确实发挥了它的作用。也就是说，很多开发人员都使用这两种框架来编写复杂性各不相同的的应用程序，我建议大家在尝试决定哪种框架最适于当前项目之前，先小范围地试用这两种框架。

欲阅读更多有关 MVVM 或 Knockout 的文章，推荐以下资料：

- MVVM 的优势

(<http://www.silverlightshow.net/news/The-Advantages-of-MVVM.aspx>)

- SO: MVVM 的问题是什么？ (<http://stackoverflow.com/questions/883895/what-are-the-problems-of-the-mvvm-pattern>)
- MVVM 解释 (<http://www.codeproject.com/Articles/100175/Model- View- ViewModel-MVVM-Explained>)
- 如何比较 MVC 与 MVVM？ (<http://www.quora.com/Pros- and-cons- of-MVVM-framework-and-how-I-can-campare-it-with-MVC>)
- KnockoutJS 中的定制绑定 (<http://www.knockmeout.net/2011/09/ ko-13- preview-part-2-custom-binding.html>)
- 利用 TodoMVC 探索 Knockout (<http://gratdevel.blogspot.co.uk/2012/02/exploring-todomvc-and-knockoutjs-with.html>)

## 第 11 章

---

# 模块化的 JavaScript 设计模式

在可扩展 JavaScript 的世界里，如果我们说一个应用程序是模块化（*modular*）的，那么通常意味着它是由一系列存储于模块中的高度解耦、不同的功能片段组成。在可能的情况下，通过移除依赖（*dependencies*），松耦合（<http://arguments.callee.info/2009/05/18/javascript-design-patterns--mediator/>）可以使应用程序的可维护性更加简单。如果有效地实现了这点，就会很容易地了解一部分如何影响另一个部分。

与一些更传统的编程语言不同，当前版本的 JavaScript (ECMA-262) (<http://www.ecma-international.org/publications/standards/Ecma-262.htm>) 并不以简洁、有组织性的方式为开发人员提供引入这种代码模块的方式。这是有关规范的一个关注点，一直没有给予重点关注，直到最近几年对更多有组织性的 JavaScript 应用程序的需求变得更加明显。

目前开发人员只能求助于本书前部分介绍的模块 (<http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth>) 或对象字面量模式 (<http://rmurphey.com/2009/10/15/using-objects-to-organize-your-code>) 的各种变体。通过使用很多这种变体，DOM 中的模块脚本被串在一起，其命名空间由单个全局对象描述，在这里，仍有可能引起架构中的命名冲突。除了使用手动或第三方工具解决，还没有一种清晰的方式能够处理依赖管理。

这些问题的原生解决方案将加入 ES Harmony (<http://wiki.ecmascript.org/doku.php?id=harmony:modules>) ——JavaScript 的下一版本，好消息是，编写模块化 JavaScript

代码再简单不过了，现在就可以开始编写。

在本节中，我们将介绍编写 JavaScript 的三种方式：*AMD*、*CommonJS* 以及 JavaScript 下一个版本 *Harmony* 的有关建议。

## 11.1 脚本加载器要点

不介绍有名的脚本加载器 (<http://msdn.microsoft.com/en-us/magazine/hh227261.aspx>)，就很难讨论 AMD 和 CommonJS 模块。在撰写本文时，脚本加载是实现一个目标的手段，如今这个成为模块化 JavaScript 的目标可以用于应用程序，因此，不得不使用兼容脚本加载器。为了使本节内容发挥最大作用，我建议大家对脚本加载工具的工作原理要有基本的了解，那么模块化方式的解释在上下文中就能够讲得通。

有很多优秀的加载器用于 AMD 和 CommonJS 方式中模块的加载，但我个人比较喜欢 RequireJS (<http://requirejs.org/>) 和 curl.js (<https://github.com/unscriptable/curl>)。有关这些工具的完整教程不在本书范围之内，但我推荐大家阅读 John Hann 所写的有关 curl.js (<http://unscriptable.com/2011/03/30/curl-js-yet-another-amd-loader/>) 的文章以及 James Burke 的 API 文档 RequireJS (<http://requirejs.org/docs/api.html>) 来了解更多相关技术。

从生产的角度来看，在使用这种模块时，建议使用用于连接脚本的优化工具（如 RequireJS 优化器）来进行部署。有趣的是，通过使用 Almond (<https://github.com/jrburke/almond>) AMD，RequireJS 不需要转入已部署站点，我们可能会认为是脚本加载器可以很容易地转移到开发之外。

也就是说，James Burke 很可能会说，页面加载以后能够动态地加载脚本仍然有用武之地，RequireJS 也可以支持这一点。记住这些要点，让我们开始使用它吧。

## 11.2 AMD

异步模块定义 (AMD) 的整体目标是提供模块化的 JavaScript 解决方案，以便开发人员使用。它诞生于使用 XHR+eval 的 Dojo 开发经验，这种格式的支持者希望能够避免未来的任何解决方案受到过去解决方案缺点的影响。

AMD 模块格式本身就是对定义模块的建议，其模块和依赖都可以进行异步加载 (<http://dictionary.reference.com/browse/asynchronous>)。它有很多独特的优点，包括可以异步，并且其本质上具有高度灵活性，消除了代码和模块之间可能惯有的紧耦合。很多开发人员都喜欢使用它，可以将它作为实现 ES Harmony 模块系统 (<http://wiki.ecmascript.org/doku.php?id=harmony:modules>) 的可靠奠基石。

AMD 最开始是 CommonJS 中模块格式的草案规范，但由于它没有达到广泛一致，这种格式的进一步发展就转移到了 amdjs 社区 (<https://github.com/amdjs>)。

如今，它被用于 Dojo、MooTools、Firebug、甚至是 jQuery 等项目。虽然有时会看到 *CommonJS AMD format* 这个术语的使用，但最好是只将它作为 AMD 或 Async 模块支持，因为并不是所有 CommonJS 的参与者都希望使用它。



曾经该建议被称为 Modules Transport/C，但由于该规范并不针对传输当前 CommonJS 模块，而是用于定义模块，它使得选择 AMD 命名约定更有意义。

### 11.2.1 模块入门

关于 AMD 有两个关键概念是值得我们注意的，它们是用于模块定义的 `define` 方法和用于处理依赖加载的 `require` 方法。使用以下方式，`define` 用于定义已命名或未命名模块：

```
define(  
    module_id /*可选*/,  
    [dependencies] /*可选*/,  
    definition function/*function for instantiating the module or object  
实例化模块或对象的函数*/  
);
```

正如代码注释所表明的，`module_id` 是一个可选参数，它通常只在非 AMD 连接工具被使用时才需要（可能会有其他一些边界情况也会使用它）。当遗漏这个参数时，我们称这个模块为匿名 (*anonymous*) 的。

当使用这个匿名模块时，模块身份的概念是 DRY，以便更容易避免文件名和代码重复。因为代码变得更轻便了，不需要修改代码本身或改变其模块 ID，就可以将

它很容易地移动到其他位置（或在文件系统附加）。将 `module_id` 想象成类似文件夹路径的概念。



开发人员可以仅通过使用 AMD 优化器在多个环境中运行相同的代码，AMD 优化器在 CommonJS 环境（比如 r.js (<https://github.com/jrburke/rjs/>)）下工作。

回到 `define` 方法，`dependencies` 参数表示我们定义模块所需的依赖数组，第三个参数（`definitionfunction` 或 `factory function`）是用于执行实例化模块的函数。一个准系统模块可以定义如下：

#### 示例 11-1 了解 AMD: `define()`

```
// 演示目的，定义 module_id (myModule)
define("myModule",
    ["foo", "bar"],
    // 模块定义函数，依赖 (foo 和 bar) 作为参数映射到函数上
    function (foo, bar) {
        // 返回定义的模块输出（例如，要暴露的内容）
        // 在这里创建你的模块

        var myModule = {
            doStuff: function () {
                console.log("Yay! Stuff");
            }
        };

        return myModule;
    });
}

// 另外一种方式
define("myModule",
    ["math", "graph"],
    function (math, graph) {

        // 注意和 AMD 不太一样，通过特定的语法以不同的形式定义模块
        return {
            plot: function (x, y) {
                return graph.drawPie(math.randomGrid(x, y));
            }
        };
    });
}
```

另一方面，`require`通常用于加载顶级JavaScript文件或模块的代码，我们是否希望动态获取依赖呢。下面是它的使用示例。

### 示例 11-2 了解 AMD: `require()`

```
// 本例中, foo 和 bar 是两个外部模块, 两个模块加载以后的输出作为回调函数的参数传入( foo 和 bar ), 以便可以方便的访问
```

```
require(["foo", "bar"], function (foo, bar) {
    // 剩余的代码
    foo.doSomething();
});
```

下面是一个动态加载依赖的示例（示例 11-3）：

### 示例 11-3 动态加载依赖

```
define(function (require) {
    var isReady = false, foobar;

    // 注意模块内部里的 require 定义
    require(["foo", "bar"], function (foo, bar) {
        isReady = true;
        foobar = foo() + bar();
    });

    // 依然返回一个模块
    return {
        isReady: isReady,
        foobar: foobar
    };
});
```

下面是一个定义 AMD 兼容插件的示例（示例 11-4）：

### 示例 11-4 了解 AMD: 插件

```
// 使用 AMD 可以加载任意格式的内容 (包括文本文件和 HTML)
// 这种方式可以用于模板依赖, 以便在页面加载的时候进行做换肤方面的工作

define( ["./templates", "text!./template.md","css!./template.css" ],
    function( templates, template ){
        console.log( templates );
        // 利用模板继续处理
    }
);
```



虽然上面的示例中包含 `css!` 用于加载 CSS 依赖，但是一定要记住，这种方法会发出一些警告，例如当 CSS 完全被加载时，它不一定完全生效。取决于如何处理创建过程，它也可能会使 CSS 作为一个依赖文件而被包含在优化的文件中，因此，在将 CSS 作为加载依赖使用的情况下，一定要谨慎，如果有兴趣尝试，可以从此处 (<https://github.com/VIISON/RequireCSS>) 获取 VIISON 的 RequireJS 的 CSS 插件。

该示例可以简单地看作是 `requirejs(["app/myModule"], function(){});`，表明加载器的顶级全局对象被使用。这里演示了如何使用不同的 AMD 加载器加载顶级模块；然后，通过使用 `define()` 函数，如果它接受一个本地模块参数，那么所有 `require([])` 的示例都适用于 curl.js 和 RequireJS 这两种类型的加载器。

#### 示例 11-5 使用 RequireJS 加载 AMD 模块

```
require(["app/myModule"],  
  
        function( myModule ) {  
  
            // 开始主模块，顺序加载其他模块  
            var module = new myModule();  
            module.doStuff();  
        });
```

#### 示例 11-6 使用 curl.js 加载 AMD 模块

```
curl(["app/myModule.js"],  
  
      function( myModule ) {  
  
          // 开始主模块，顺序加载其他模块  
          var module = new myModule();  
          module.doStuff();  
      });
```

下面是具有延迟依赖（异步依赖）的模块代码：

```
// 本代码应该兼容 jQuery 的 Deferred 或 futures.js 实现，或者其他类似实现  
  
define(["lib/Deferred"], function (Deferred) {  
    var defer = new Deferred();  
  
    require(["lib/templates/?index.html", "lib/data/?stats"],  
           function (template, data) {
```

```
        defer.resolve({ template: template, data: data });
    }
);
return defer.promise();
});
```

## 11.2.2 使用 Dojo 的 AMD 模块

数组中将任何模块依赖定义为第一个参数，并提供一个回调（factory），一旦依赖加载，它将执行该模块。例如：

```
define(["dijit/Tooltip"], function( Tooltip ){
    // dijit tooltip 可以在局部使用了
    new Tooltip(...);

});
```

注意模块的匿名特性，Dojo 异步加载器、RequireJS 或标准 dojo.require() 模块加载器 (<http://livedocs.dojotoolkit.org/dojo/require>) 都可以实现匿名特性。

有一些关于模块引用的有趣陷阱，我们这里有必要进行了解。虽然 AMD 式的模块引用通过使用匹配的参数定义依赖列表，但这并没有得到较旧版本的 Dojo 1.6 构建系统的支持——它只适用于 AMD 兼容的加载器。例如：

```
define(["dojo/cookie", "dijit/Tooltip"], function( cookie, Tooltip ){
    var cookieValue = cookie( "cookieName" );
    new Tooltip(...);

});
```

这与嵌套的命名空间相比有很多优点，因为模块不再每次都直接引用完整的命名空间；我们所需要的仅仅是依赖中的 dojo/cookie 路径，使之作为一个参数，就可以被该变量引用。这使我们无需在应用程序中多次导入 dojo。

最后需要注意的是，如果希望继续使用老版的 Dojo 构建系统或希望将老版模块改造成新式的 AMD 风格，下面这个更详细的版本可以使得迁移变得比较容易。注意，dojo 和 dijit 也被引用为依赖：

```
define(["dojo", "dijit", "dojo/cookie", "dijit/Tooltip"], function( dojo,
dijit ){
    var cookieValue = dojo.cookie( "cookieName" );
```

```
    new dijit.Tooltip(...);
});
```

### 11.2.3 AMD 模块设计模式 ( Dojo )

正如前面部分所述，设计模式可以非常有效地改进创建解决方案的方法，以便解决常见的开发问题。JohnHann (<http://twitter.com/unscriptable>) 已经给出了一些非常优秀的有关 AMD 模块设计模式的演示文稿，包括 Singleton、Decorator、Mediator 等等，我强烈推荐大家查看他制作的幻灯片 (<http://unscriptable.com/code/AMD-module-patterns/#0>)。

可以在下面找到 AMD 设计模式的一些实现（示例 11-7 和示例 11-8）：

#### 示例 11-7 Decorator 模式

```
// mylib/UpdatableObservable:a becoratorfor dojo/store/Observable 的装饰者
define(["dojo", "dojo/store/Observable"], function (dojo, Observable) {
    return function UpdatableObservable(store) {
        var observable = dojo.isFunction(store.notify) ? store :
            new Observable(store);

        observable.updated = function (object) {
            dojo.when(object, function (itemOrArray) {
                dojo.forEach([], concat(itemOrArray), this.notify, this);
            });
        };

        return observable;
    };
});

// mylib/UpdatableObservable 的使用者

define(["mylib/UpdatableObservable"], function (makeUpdatable) {
    var observable,
        updatable,
        someItem;

    // 让 observable stroe 变成可更新
    updatable = makeUpdatable(observable); //New 操作符是可选的

    // 如果要改变数据，传入参数调用.update() 即可
    //updatable.updated( updatedItem );
});
```

#### 示例 11-8 Adapter 模式

```
// mylib/Array"适配 each 功能 (模仿 jQuery)
```

```

define(["dojo/_base/lang", "dojo/_base/array"], function (lang, array) {
    return lang.delegate(array, {
        each: function (arr, lambda) {
            array.forEach(arr, function (item, i) {
                lambda.call(item, i, item); // 就像 jQuery 的 each
            });
        }
    });
});
// 使用该适配器 Adapter
// "myapp/my-module":
define(["mylib/Array"], function (array) {
    array.each(["uno", "dos", "tres"], function (i, esp) {
        // 这里, `this` == item
    });
});

```

## 11.2.4 使用 jQuery 的 AMD 模块

与 Dojo 不同, jQuery 只有一个文件; 但由于库具有基于插件的特性, 我们可以演示使用 jQuery 定义 AMD 模块有多么简单, 如下所示。

```

define(["js/jquery.js", "js/jquery.color.js", "js/underscore.js"],
    function( $, colorPlugin, _ ){
        // 这里我们传入了 jQuery, color 插件和 Underscore, 三者在全局作用域都不
        // 可访问, 但却可以在内部轻松访问
        // 随机产生颜色数字, 选择数组的第一个项进行操作

        var shuffleColor = _.first(_.shuffle("#666", "#333", "#111"));
        // 将页面上所有带有 class 为 item 的元素都进行背景颜色的改变动画操作, 使用
        shuffleColor 这个颜色
        $(".item").animate( {"backgroundColor": shuffleColor} );

        // 返回的值可以用于其他模块
        return {};
    });

```

但是, 这个示例中缺少了一些内容: 注册的概念。

### 11.2.4.1 将 jQuery 注册为异步兼容模块

jQuery 1.7 的其中一个关键特性是支持将 jQuery 注册为异步模块。有很多兼容性的脚本加载器 (包括 RequireJS 和 curl) 能够使用异步模块方式加载模块, 这意味着值需要很少插件页面即可运行。

如果开发人员想要使用 AMD，并不希望其 jQuery 版本泄漏到全局空间中，她应在使用 jQuery 的顶层模块中调用 noConflict。此外，因为多个版本的 jQuery 可以在一个页面上，需要特别注意 AMD 加载器必须为此负责，因此 jQuery 只在识别这些问题的 AMD 加载器上进行注册，这些问题由指定 define.amd.jQuery 的加载器来表明。RequireJS 和 curl 是完成这种工作的两种加载器。

命名的 AMD 提供了一种安全的方式，可以安全稳健地用于大多数用例。

```
//在 document 对象中，负责各个 jQuery 全局实例，便于测试.noConflict

var jQuery = this.jQuery || "jQuery",
$ = this.$ || "$",
originaljQuery = jQuery,
original$ = $;
define(["jquery"], function ($) {
    $(".items").css("background", "green");
    return function () { };
});
```

#### 11.2.4.2 为何 AMD 是编写模块化 JavaScript 的更好选择

我们已经回顾了一些代码示例，了解了 AMD 的能力。它似乎给我们展现的不仅仅是一个典型的 Module 模式，但它为什么是模块化应用程序开发的更好选择？

- 对于如何完成灵活模块的定义提供了明确的建议。
- 比很多人依赖的现有全局命名空间和<script>标签解决方案都更加简洁。有一种简洁的方式可以声明可能的独立模块和依赖。
- 封装模块定义，帮助我们避免全局命名空间被污染。
- 可以说它比一些替代解决方案运行的更好（如我们稍后将看到的 CommonJS）。它没有跨域、本地或调试的问题，也不依赖于服务器端工具。大多数 AMD 加载器支持浏览器中的加载模块，无需构建流程。
- 提供一个“transport”方法，将多个模块包括在单个文件中。CommonJS 等其他方法尚未在 transport 格式方面达成一致。
- 如果需要，可以延迟加载脚本。



上述的很多内容也可能被说成 YUI 的模块加载策略。

#### 11.2.4.3 延伸阅读

AMD 的 RequireJS 指南

(<http://requirejs.org/docs/whyamd.html>)

加载 AMD 模块最快的方式是什么？(<http://unscriptable.com/2011/09/21/what-is-the-fastest-way-to-load-amd-modules/>)

AMD 与 CommonJS，更好的格式是什么？(<http://unscriptable.com/2011/09/30/amd-versus-cjs-whats-the-best-format/>)

AMD 比 CommonJS 更适用于 Web 模块 (<http://blog.millermedeiros.com/amd-is-better-for-the-web-than-commonjs-modules/>)

未来是模块不是框架

(<http://unscriptable.com/code/Modules-Frameworks/>)

AMD 不再是 CommonJS 规范 ([http://groups.google.com/group/commonjs/browse\\_thread/thread/96a0963bcb4ca78f/cf73db49ce267ce1?lnk=gst#](http://groups.google.com/group/commonjs/browse_thread/thread/96a0963bcb4ca78f/cf73db49ce267ce1?lnk=gst#))

发明 JavaScript 模块格式和脚本加载器 (<http://tagneto.blogspot.com/2011/04/on-inventing-js-module-formats-and.html>)

AMD 邮件列表 (<http://groups.google.com/group/amd-implement>)

#### 11.2.4.4 支持 AMD 的脚本加载器和框架

浏览器中：

- RequireJS <http://requirejs.org>
- curl.js <http://github.com/unscriptable/curl>

- bdLoad <http://bdframework.com/bdLoad>
- Yabble <http://github.com/jbrantly/yabble>
- PINF <http://github.com/pinf/loader-js>
- (and more)

服务器端：

- RequireJS <http://requirejs.org>
- PINF <http://github.com/pinf/loader-js>

### 11.2.5 AMD 总结

在多个项目中使用 AMD 后，我的结论是，创建严格应用程序的开发人员可能希望在更好的模块格式中，它能够勾选很多复选框。它使我们不需要担心全局对象，并支持命名模块，不需要将服务器转换为函数，可以很好地用于依赖管理。

它对于使用 Backbone.js、ember.js 或其他结构化框架进行模块化开发也是一个不错的补充方式，以保持应用程序的组织性。

在 Dojo 和 CommonJS 的世界里，AMD 已经被深入探讨了长达约两年的时间，我们知道它需要时间去成熟和发展。我们也知道它久经沙场，很多大公司使用它来创建重要的应用程序（IBM、BBC iPlayer），因此，如果不能用了，很可能是他们现在抛弃它了，但目前还没有。

这就是说，AMD 仍有可以改进的地方。已经使用过一段时间的开发人员可能觉得 AMD 样例文件/包装器代码是一个很恼人的东西。虽然我也有这方面的担心，但是 Volo (<https://github.com/volojs/volo>) 等工具可以帮助我们解决这些问题，我认为，总的来说，使用 AMD 的优点远远多于其缺点。

## 11.3 CommonJS

CommonJS 模块建议指定一个简单的 API 来声明在浏览器外部工作的模块（如在服

务器上)。与 AMD 不同, 它试图包含更广泛的引人关注的问题, 如 IO、文件系统、promises 等等。

CommonJS 起初在 2009 年由 Kevin Dangoor 启动的一个项目中被称为 ServerJS, 最近该格式在一个志愿者工作小组 CommonJS (<http://www.commonjs.org/>) 的努力下已变得更加正式化, 这个小组旨在设计、标准化和规范 JavaScript API。到目前为止, 他们已努力获得对模块 (<http://www.commonjs.org/specs/modules/1.0/>) 和包 (<http://wiki.commonjs.org/wiki/Packages/1.0>) 标准的批准。

### 11.3.1 入门指南

从结构的角度来看, CommonJS 模块是 JavaScript 中的可复用部分, 导出特定对象, 以便可以用于任何依赖代码。与 AMD 不同, 在这种模块周围通常是没有函数封装器的 (所以我们在这里看不到 `define`)。

CommonJS 模块基本上包含两个主要部分: 自由变量 `exports`, 它包含了一个模块希望其他模块能够使用的对象, 以及 `require` 函数, 模块可以使用该函数导入 (`import`) 其他模块的导出 (`exports`) (示例 11-9、示例 11-10 和示例 11-11)。

#### 示例 11-9 了解 CommonJS: `require()` 和导出

```
// package/lib 是我们需要的一个依赖
var lib = require( "package/lib" );

// 我们模块的行为
function foo(){
    lib.log( "hello world!" );
}

// 导出 (暴露) foo 给其他模块
exports.foo = foo;
```

#### 示例 11-10 导出的基本消耗

```
// package/lib 是我们需要的一个依赖
var lib = require( "package/lib" );

// 我们模块的行为
function foo(){
    lib.log( "hello world!" );
}
```

```
// 导出（暴露）foo 给其他模块  
exports.foo = foo;
```

#### 示例 11-11 第一个 CommonJS 示例的 AMD 等效代码

```
define(function(require){  
    var lib = require("package/lib");  
  
    // 我们模块的行为  
    function foo(){  
        lib.log("hello world!");  
    }  
  
    // 导出（暴露）foo 给其他模块  
    return {  
        foobar: foo  
    };  
});
```

AMD 可以完成该项工作，因为 AMD 支持简化的 CommonJS 包装 (<http://requirejs.org/docs/whyamd.html#sugar>) 功能。

### 11.3.2 使用多个依赖

#### app.js:

```
var modA = require("./foo");  
var modB = require("./bar");  
  
exports.app = function () {  
    console.log("Im an application!");  
}  
  
exports.foo = function () {  
    return modA.helloWorld();  
}
```

#### bar.js:

```
exports.name = "bar";
```

#### foo.js:

```
require("./bar");  
exports.helloWorld = function () {  
    return "Hello World!!"  
}
```

### 11.3.3 支持 CommonJS 的加载器和框架

浏览器中：

- curl.js <http://github.com/unscriptable/curl>
- SproutCore 1.1 <http://sproutcore.com>
- PINF <http://github.com/pinf/loader-js>

服务器端：

- Node <http://nodejs.org>
- Narwhal <https://github.com/trobinson/narwhal>
- Persevere <http://www.persvr.org/>
- Wakanda <http://www.wakandasoft.com/>

### 11.3.4 CommonJS 适用于浏览器吗？

有些开发人员认为 CommonJS 更适用于服务器端开发，这是目前在 Harmony 前期发展中应将哪种格式作为真正的标准方面存在一定分歧的其中一个原因。一些关于 CommonJS 的反对言论包括：很多 CommonJS API 具有面向服务器特性，这是我们可能无法在 Javascript 的浏览器水平上实现的特性，例如，由于其功能的本质特征，*io*、*system* 和 *js* 都被认为是无法实现的。

也就是说，它可用于了解如何构造 CommonJS 模块，以便我们能够更好地了解在定义模块时如何更好地使用它们，这些模块可能被用于任何地方。在客户端和服务器都含有应用程序的模块包括验证、转换和模板引擎。当一个模块可以用于服务器端环境中时，一些开发人员在选择使用哪种格式方面倾向于选择 CommonJS，而在其他情况下会使用 AMD。

AMD 模块能够使用插件，并能够定义更细粒度的东西，如构造函数和函数，这是说得通的。但如果我们试图从中获得构造函数，CommonJS 模块则仅能够定义不易

使用的对象。

尽管这超出了本节的范围，你可能还会注意到，在讨论 AMD 和 CommonJS 时，我们提到了不同类型的 `require` 方法。相似命名规范令人担忧的一点是混淆性，社区目前正在划分全局 `require` 函数的优点。John Hann 在这里的建议是，重命名全局加载器方法可能会更有意义（如库的名称），而不是称它为 `require`，因为它可能无法告知用户全局 `require` 和内部 `require` 之间的差异。正是因为这个原因，`curl.js` 等加载器使用 `curl()`，而不是 `require`。

### 11.3.5 延伸阅读

阐明 CommonJS 模块

(<http://dailyjs.com/2010/10/18/modules/>)

JavaScript 的发展

(<http://www.slideshare.net/davidpadbury/javascript-growing-up>)

CommonJS 中的 RequireJS 要点

(<http://requirejs.org/docs/commonjs.html>)

一步步学习 Node.js 和 CommonJS——创建自定义模块(<http://elegantcode.com/2011/2/04/taking-baby-steps-with-node-js-commonjs-and-creating-custom-modules/>)

用于浏览器的异步 CommonJS 模块 (<http://www.sitepen.com/blog/2010/07/16/synchronous-commonjs-modules-for-the-browser-and-introducing-transporter/>)

CommonJS 邮件列表

(<http://groups.google.com/group/commonjs>)

## 11.4 AMD 和 CommonJS：互相竞争，标准同效

AMD 和 CommonJS 都是有效的模块格式，有不同的最终目标。

AMD 采用浏览器优先的开发方法，选择异步行为和简化的向后兼容性，但是它没有任何文件 I/O 的概念。它支持对象、函数、构造函数、字符串、JSON 以及很多其他类型的模块，在浏览器中原生运行。其使用非常的灵活。

另一方面，CommonJS 采用服务器优先方法，假定同步行为，没有全局概念这个精神包袱，并试图迎合未来技术（在服务器上）。我们这样做的意思是，由于 CommonJS 支持非包装模块，它可以更接近下一代 ES Harmony 规范，让我们摆脱 AMD 强制执行的 `define()` 包装器。但 CommonJS 模块仅将对象作为模块给予支持。

## UMD：用于插件的 AMD 和 CommonJS 兼容模块

对于希望创建在浏览器和服务端环境中都能工作的模块的开发人员来说，现有的解决方案可能有一点不足。为了解决这个问题，我与 James Burke 和其他开发者共同创建了通用模块定义（UMD；<https://github.com/umdjs/umd>）。

UMD 是一种实验模块格式，支持在客户端和服务器端环境中工作的模块定义，并且编写时所有或大多数流行的脚本加载技术在这些环境时中都是可用的。虽然一种新的模块格式的概念可能是令人怯步的，但为了全面对其进行了解，我们将对 UMD 进行简要介绍。

我们最初是通过查看 AMD 规范支持的简化 CommonJS 包装器开始定义 UMD。对于希望编写类似 CommonJS 模块的开发人员来说，可以使用以下 CommonJS 兼容的格式：

### 基本 AMD 混合格式

```
define(function (require, exports, module) {  
  
    var shuffler = require("lib/shuffle");  
    exports.randomize = function (input) {  
        return shuffler.shuffle(input);  
    }  
});
```

然而，需要注意的是，如果模块不包含依赖数组，它只会被用作 CommonJS 模块，定义函数最少包含一个参数。它在一些设备（如 PS3）上也不会正常工作。欲了解更多有关上述包装器的信息，请参阅 <http://requirejs.org/docs/api.html#cjsmodule>。

再进一步，我们想提供一些不同的模式，它们不仅能够与 AMD 和 CommonJS 一起使用，同时还能解决开发人员希望开发的这种模块在其他环境中的常见兼容性问题。

我们在下面看到的这种变体允许我们使用 CommonJS、AMD 或者浏览器全局对象来创建一个模块。

### 使用 CommonJS、AMD 或者浏览器全局对象创建模块

定义模块 commonJsStrict，它依赖于另一个称为 b 的模块。该模块的名称隐含在文件名中，处理文件名称和导出全局对象的最佳做法是使用相同的名称。

如果模块 b 也在浏览器中使用相同类型的样例文件，它将创建一个已使用的全局对象.b。如果我们不希望支持浏览器全局对象，可以删除 root 并将 this 作为第一个参数传递给顶部函数。

```
(function (root, factory) {
    if (typeof exports === 'object') {
        // CommonJS
        factory(exports, require('b'));
    } else if (typeof define === 'function' && define.amd) {
        // AMD. 注册为一个匿名模块
        define(['exports', 'b'], factory);
    } else {
        // 浏览器全局对象
        factory((root.commonJsStrict = {}), root.b);
    }
}(this, function (exports, b) {
    // 附件属性到 exports 对象上，以定义导出的 module 属性。
    exports.action = function () { };
}));
```

UMD 资源库包含的变体包括在浏览器中运行最好的模块，最能用于导出的模块，可以很好地用于 CommonJS 运行时的模块，甚至是那些最适合定义 jQuery 插件的模块，我们接下来就对其进行了解。

### 所有环境中都起作用的 jQuery 插件

UMD 提供两种使用 jQuery 插件的模式：一种模式可以定义能够与 AMD 和浏览器

全局配合良好的插件，另一种模式还可以在 CommonJS 环境中工作。jQuery 不可能用于大多数 CommonJS 环境中，因此要记住，除非我们正在使用的这种环境确实能够与它配合良好。

现在，我们将定义一个插件，它由核心 core 和核心扩展 extension 组成。

- 插件将由一个核心 core 和核心扩展 extension 组成。
- 核心插件被加载到\$.core 命名空间中，然后可以通过命名空间模式使用插件扩展轻松对其进行扩展。通过脚本标签自动加载的插件位于 core 下的 plugin 命名空间内（即\$.core.plugin.methodName()）。

该模式很好使用，因为插件扩展可以访问在基类中定义的属性和方法，或者稍作调整、重写默认行为，以便扩展后能够实现更多功能。也不需要加载器来让它们充分发挥作用。

欲了解其中的更多细节，请参阅下面代码示例中的注释。

#### usage.html:

```
<script type="text/javascript" src="jquery-1.7.2.min.js"></script>
<script type="text/javascript" src="pluginCore.js"></script>
<script type="text/javascript" src="pluginExtension.js"></script>

<script type="text/javascript">

$(function() {

    // 插件 core 暴露到本例的 core 命名空间下，这里首先进行缓存
    var core = $.core;

    // 使用 core 内置的功能高亮页面上所有的 div 为黄色
    core.highlightAll();

    // 访问 plugin 命名空间下的插件，设置页面的第一个 div 的背景色为绿色
    core.plugin.setGreen( "div:first" );
    // 这里首先调用 core 下的 highlight 方法，之后设置最后一个 div 的颜色为错误红色，
    // errorColor 是定义 core/plug-in 下。
    // 查看后面的代码，就会发现调用这些属性和方法是多么简单
    core.plugin.setRed("div:last");
});

});
```

```
</script>
```

### pluginCore.js:

```
// Module/plug-in core
// 注: 模块封装的代码支持多种模块化格式和规范, 通过映射到指定的规范格式上来实现
// 我们的模块实际功能是在下面实现的
// 如果需要依赖, 可以很容易声明, 应该像前面的 AMD 模块示例一样能用

(function (name, definition) {
    var theModule = definition(),
        // this 在这里是“安全”的
        hasDefine = typeof define === "function" && define.amd,
        // hasDefine = typeof define === "function",
        hasExports = typeof module !== "undefined" && module.exports;

    if (hasDefine) { // AMD Module
        define(theModule);
    } else if (hasExports) { // Node.js Module
        module.exports = theModule;
    } else {
        // 赋值给通用命名空间或者直接赋值给全局对象 (window)
        (this.jQuery || this.ender || this.$ || this)[name] = theModule;
    }
})("core", function () {
    var module = this;
    module.plugins = [];
    module.highlightColor = "yellow";
    module.errorColor = "red";

    // 定义 core 模块, 返回公有 API

    // 这里的 highlight 就是上面例子中的 core.highlightAll()方法, 不同的插件可以
    // 高亮不同的颜色
    module.highlight = function (el, strColor) {
        if (this.jQuery) {
            jQuery(el).css("background", strColor);
        }
    }
    return {
        highlightAll: function () {
            module.highlight("div", module.highlightColor);
        }
    };
});
```

### pluginExtension.js:

```
// Extension to module core
```

```

(function (name, definition) {
    var theModule = definition(),
        hasDefine = typeof define === "function",
        hasExports = typeof module !== "undefined" && module.exports;

    if (hasDefine) { // AMD Module
        define(theModule);
    } else if (hasExports) { // Node.js Module
        module.exports = theModule;
    } else {

        // 根据平面文件/全局模块扩展，赋值给通用命名空间或者直接赋值给全局对象 (window)
        var obj = null,
            namespaces,
            scope;

        obj = null;
        namespaces = name.split(".");
        scope = (this.jQuery || this.ender || this.$ || this);

        for (var i = 0; i < namespaces.length; i++) {
            var packageName = namespaces[i];
            if (obj && i == namespaces.length - 1) {
                obj[packageName] = theModule;
            } else if (typeof scope[packageName] === "undefined") {
                scope[packageName] = {};
            }
            obj = scope[packageName];
        }
    }
})("core.plugin", function () {

    // 定义我们的模块并返回公有 API
    // 如果想扩充 highlight 方法的功能，这里的代码可以很容易适配 core 中的方法，以及重写和扩展 core 的现有功能
    return {
        setGreen: function (el) {
            highlight(el, "green");
        },
        setRed: function (el) {
            highlight(el, errorColor);
        }
    };
});

```

UMD 的目的并不是取代 AMD 和 CommonJS，仅是为希望让代码在更多环境中运

行的开发人员提供一些额外帮助。欲获得更多有关信息或对该实验格式提供建议，请参阅 <https://github.com/umdjs/umd>。

## 延伸阅读

使用 AMD 加载器编写和管理模块化 JavaScript，John Hann (<http://unscriptable.com/code/Using-AMD-loaders/#0>)

解密 CommonJS 模块，Alex Young

(<http://dailyjs.com/2010/10/18/modules/>)

AMD 模块模式：单例，John Hann

(<http://unscriptable.com/2011/09/22/amd-module-patterns-singleton/>)

在任何地方都可以运行的 JavaScript 模块样例代码，Kris Zyp (<http://www.sitepen.com/blog/2010/09/30/run-anywhere-javascript-modules-boilerplate-code/>)

JavaScript 模块和 jQuery 标准和建议，James Burke (<http://tagneto.blogspot.com/2010/02/standards-and-proposals-for-javascript.html>)

## 11.5 ES Harmony

TC39 (<http://www.ecma-international.org/memento/TC39.htm>)，用于定义 ECMAScript 语法和语义的标准体，在过去几年一直密切关注着 JavaScript 在大规模开发中的使用情况。一个可能需要他们一直关注的领域是支持更高级的模块，以满足现代 JavaScript 开发人员的需求。

因此，现在有很多对该语言的扩展建议，包括能够在客户端和服务器上运行的灵活模块 (<http://wiki.ecmascript.org/doku.php?id=harmony:modules>)，模块加载器 ([http://wiki.ecmascript.org/doku.php?id=harmony:module\\_loaders](http://wiki.ecmascript.org/doku.php?id=harmony:module_loaders)) 等等 (<http://wiki.ecmascript.org/doku.php?id=harmony:proposals>)。在本节中，我们将使用下一代 ES 中建议的语法，做一些代码示例，以便可以了解接下来会发生什么。



虽然 Harmony 仍处于建议阶段，我们已经可以尝试 ES.next 的（部分）特性，感谢谷歌的 Traceur 编译器 (<http://code.google.com/p/traceur-compiler/>)。它能够解决用于编写模块化 JavaScript 的原生支持问题。

想在一分钟内启动并运行 Traceur，请阅读入门指南 (<http://code.google.com/p/traceur-compiler/wiki/GettingStarted>)。如果你有兴趣学习更多关于这个项目知识，JSConf 上的演示文稿 (<http://traceur-compiler.googlecode.com/svn/branches/v0.10/presentation/index.html>) 也是值得一看的。

### 11.5.1 具有 Imports 和 Exports 的模块

阅读过有关 AMD 和 CommonJS 模块的这几个章节后，你可能会熟悉模块依赖（导入）和模块导出（或是允许其他模块调用的公共 API/变量）的概念。在 ES.next 中，已经以一种更简洁的方式提出了这些概念，使用 `import` 关键字指定其依赖。`export` 与我们所预想的差别不是很大，很多开发人员会再往下查看代码示例，并很快掌握它们的用法。

- `import` 声明绑定一个模块，作为局部变量导出，并可能被重新命名，以避免名称冲突/矛盾。
- `export` 声明：声明一个外部可见模块的本地绑定，这样其他模块可以获取该导出，但不能修改它们。有趣的是，模块可以导出子模块，但是不能导出在其他地方定义的模块。我们还可以重命名导出，因此它们的外部名称与本地名称不同。

```
module staff{
    // 指定可以被其他模块调用的公有导出
    export var baker = {
        bake: function( item ){
            console.log( "Woo! I just baked " + item );
        }
    }
}

module skills{
    export var specialty = "baking";
    export var experience = "5 years";
```

```
}

module cakeFactory{

    // 指定依赖
    import baker from staff;

    // 使用通配符导入所有内容
    import * from skills;

    export var oven = {
        makeCupcake: function( toppings ){
            baker.bake( "cupcake", toppings );
        },
        makeMuffin: function( mSize ){
            baker.bake( "muffin", size );
        }
    }
}
```

## 11.5.2 从远程数据源加载的模块

该模块还建议支持基于远程的模块（例如第三方库），以便使从外部位置载入模块简单化。下面这个示例显示导入我们上面定义的模块以及其使用：

```
module cakeFactory from "http://addyosmani.com/factory/cakes.js";
cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

## 11.5.3 模块加载器 API

模块加载器建议一个动态的 API 在严格控制的上下文中加载模块。在加载器上支持的特征包括用于加载模块的 `load(url, moduleInstance, error)`、`createModule(object, globalModuleReferences)` 等等 ([http://wiki.ecmascript.org/doku.php?id=harmony:module\\_loaders](http://wiki.ecmascript.org/doku.php?id=harmony:module_loaders))。

下面是动态加载我们最初定义的模块的另一个示例。请注意，与上一个示例中我们从远程数据源导入模块的做法不同，该模块加载器 API 更适用于动态上下文。

```
Loader.load("http://addyosmani.com/factory/cakes.js",
    function (cakeFactory) {
        cakeFactory.oven.makeCupcake("chocolate");
    });
}
```

## 11.5.4 用于服务器的类 CommonJS 模块

对于对服务器环境更感兴趣的开发人员来说，ES.next 建议的模块系统不仅仅查看浏览器中的模块。例如，在这里我们可以看到一个在服务器上使用的类 CommonJS 模块：

```
// io/File.js
export function open( path ) { ... };
export function close( hnd ) { ... };

// compiler/LexicalHandler.js
module file from "io/File";

import { open, close } from file;
export function scan( in ) {
    try {
        var h = open( in ) ...
    }
    finally { close( h ) }
}

module lexer from "compiler/LexicalHandler";
module stdlib from "@std";

//... scan(cmdline[0]) ...
```

## 11.5.5 具有构造函数、getter 和 setter 的类

类的概念在纯粹主义者中间一直是一个有争议的问题，到目前为止，我们要么是求助于 JavaScript 的原型性质 (<http://javascript.crockford.com/prototypal.html>)，或是使用框架或抽象，它们能够提供使用 *class* 定义的形式，以一种脱语法糖的形式达到相同的原型行为方式。

在 Harmony 中，建议类与构造函数和（最终）一些真正私有的内容一起使用。在下面的示例中，提供了内联注释来帮助了解如何实现类的结构化。

通读后，你可能也注意到，这里缺少单词“函数”。这不是输入错误：TC39 有意识地减少 *function* 关键字的滥用，希望这将有助于简化我们编写的代码。

```
class Cake{
    // 使用关键字 constructor 定义类的构造函数，接着是公有的参数和内部声明
```

```

constructor( name, toppings, price, cakeSize ){
    public name = name;
    public cakeSize = cakeSize;
    public toppings = toppings;
    private price = price;
}

// 作为 ES.next 的一部分，减少不必要的 function 定义，可以看到去除了 function
// 定义，而是定义标识符和参数的形式来实现方法

addTopping( topping ){
    public( this ).toppings.push( topping );
}

// Getter 可以在标识符/方法名称签名加上 get 来实现
get allToppings(){
    return public( this ).toppings;
}

get qualifiesForDiscount(){
    return private( this ).price > 5;
}

// 和 getter 类似，setter 也是通过在标识符/方法名称签名加上 set 来实现
set cakeSize( cSize ){
    if( cSize < 0 ){
        throw new Error( "Cake must be a valid size - "
            + "either small, medium or large" );
    }
    public( this ).cakeSize = cSize;
}

}

```

### 11.5.6 ES Harmony 总结

如前所述，Harmony 可能会有一些很优秀的新增加功能，以简化模块化应用程序的开发，并处理依赖管理等问题。

目前，在现今的浏览器中使用 Harmony 语法的最佳选择是通过 transpiler，如谷歌 Traceur (<http://code.google.com/p/traceur-compiler/>) 或 Esprima (<https://code.google.com/p/esprima/>)。还有 RequireHM (<https://github.com/addyosmani/require-hm>) 等项目允许我们使用具有 AMD 的 Harmony 模块。但在最终规范发布之前，我们的首选

是 AMD（对于浏览器内置模块）和 CommonJS（对于用在服务器上的那些模块）。

### 11.5.7 延伸阅读

初识即将出现的 JavaScript 模块 (<http://www.2ality.com/2011/03/first-look-at-upcoming-javascript.html>)

David Herman 在 JavaScript/ES.Next 方面的见解（视频）(<http://blog.mozilla.org/dherman/2011/02/23/my-js-meetup-talk/>)

ES Harmony 模块建议

(<http://wiki.ecmascript.org/doku.php?id=harmony:modules>)

ES Harmony 模块语义/结构原理 ([http://wiki.ecmascript.org/doku.php?id=harmony:modules\\_rationale](http://wiki.ecmascript.org/doku.php?id=harmony:modules_rationale))

ES Harmony 类建议

(<http://wiki.ecmascript.org/doku.php?id=harmony:classes>)

## 11.6 总结

在这一节中，我们回顾了使用现代模块方式编写模块化 JavaScript 的几个可用选择。

这些方式相比仅使用 Module 模式有很多优点，包括：无需管理全局变量，更好地支持静态和动态依赖管理，改进与脚本加载器的兼容性，服务器上模块更好的兼容性等等。

简而言之，我建议大家尝试使用本章中建议的做法，因为这些方式提供大量的功能和灵活性，可以极大地帮助我们更好的组织应用程序。

# jQuery 中的设计模式

jQuery 目前是最流行的 JavaScript DOM 操作库，并提供了一个用于以安全、跨浏览器的方式与 DOM 进行交互的抽象层。有趣的是，库也可以作为演示设计模式如何可以有效地用于创建可读并容易使用 API 的示例。

在很多情况下，编写 jQuery 的核心贡献者并没有打算开始使用特定的模式，但它们仍存在，并且有一定的借鉴意义。让我们来看看其中的一些模式以及它们是如何在 API 中使用的。

## 12.1 Composite（组合）模式

*Composite* 模式描述了一组对象，可以使用与处理对象的单个实例同样的方式来进行处理。

这使我们能够以统一的方式来处理单个对象和组合，这意味着不管我们正在处理的是一个项目还是一千个项目，都可以应用同样的行为。

在 jQuery 中，把方法应用于元素或元素集合时，可以用统一的方式来处理它们，因为这两种选择都返回 jQuery 对象。

使用下面的 jQuery 选择器的代码示例演示了这种情况。在这里，可以向单一元素添加一个 `active` 类（`class`）（如具有唯一 ID 的元素）或一组具有相同标签名或类的元素，而无需额外的工作。

```
// Single elements
$("#singleItem").addClass("active");
$("#container").addClass("active");

// Collections of elements
$("div").addClass("active");
$(".item").addClass("active");
$("input").addClass("active");
```

jQuery `addClass()` 实现可以直接使用原生 `for` 循环（或 jQuery 的 `jQuery.each()`/`jQuery.fn.each()`）来遍历一个集合，以将这种方法应用于单个 item 或组。通过阅读 jQuery 源代码，我们可以看到确实就是这样处理的：

```
addClass: function( value ) {
    var classNames, i, l, elem,
        setClass, c, cl;

    if ( jQuery.isFunction( value ) ) {
        return this.each(function( j ) {
            jQuery( this ).addClass( value.call(this, j, this.className) );
        });
    }

    if ( value && typeof value === "string" ) {
        classNames = value.split( rspace );

        for ( i = 0, l = this.length; i < l; i++ ) {
            elem = this[ i ];

            if ( elem.nodeType === 1 ) {
                if ( !elem.className && classNames.length === 1 ) {
                    elem.className = value;
                } else {
                    setClass = " " + elem.className + " ";
                    for ( c = 0, cl = classNames.length; c < cl; c++ ) {
                        if ( !~setClass.indexOf( " " + classNames[ c ] + " " ) ) {
                            setClass += classNames[ c ] + " ";
                        }
                    }
                    elem.className = jQuery.trim( setClass );
                }
            }
        }
    }
}
```

```
    return this;
}
```

## 12.2 Adapter（适配器）模式

*Adapter* 模式将对象或类的接口 (*interface*) 转变为与特定的系统兼容的接口。

适配器基本上允许类或函数在一起运行，通常并不是因为它们的不兼容接口。

适配器将接口的调用转变为原始接口的调用，实现这个目的所需的代码通常是很小的。

我们可能用过的适配器示例是 jQuery 的 `jQuery.fn.css()` 方法。它不仅有助于标准化接口，以展示样式如何应用于多种浏览器，使我们能够轻松使用简单的语法适配浏览器在后台实际支持的语法。

```
// 跨浏览器透明度
// opacity: 0.9; Chrome 4+, FF2+, Saf3.1+, Opera 9+, IE9, iOS 3.2+, Android 2.1+
// filter: alpha(opacity=90); IE6-IE8

// 设置 opacity
$(".container").css({ opacity: .5 });

// 获取 opacity
var currentOpacity = $(".container").css('opacity');
```

相应的 jQuery 核心 `cssHook` 使上面的代码得以实现，如下所示：

```
get: function( elem, computed ) {
    // IE 使用 filter 获取 opacity
    return ropacity.test(
        computed && elem.currentStyle ?
            elem.currentStyle.filter : elem.style.filter) || "" ) ?
        ( parseFloat( RegExp.$1 ) / 100 ) + "" :
        computed ? "1" : "";
    },

set: function( elem, value ) {
    var style = elem.style,
        currentStyle = elem.currentStyle,
        opacity = jQuery.isNumeric( value ) ?
            "alpha(opacity=" + value * 100 + ")" : "",
        filter = currentStyle && currentStyle.filter || style.filter || "";
```

```

// 如果没有 layout, IE 在 opacity 方面有问题
// 强制设置 zoom level
style.zoom = 1;

// 如果设置 opacity 为 1, 并且没有其他的 filter 存在, 尝试删除 filter 属性
if ( value >= 1 && jQuery.trim( filter.replace( ralpha, "" ) ) === "" ) {

    // 设置 filter 属性为 null, "", "" 时, 该属性依然存在
    style.removeAttribute( "filter" );

    // 如果没有 filter 样式属性了, 也就完成了
    if ( currentStyle && !currentStyle.filter ) {
        return;
    }
}

// 否则, 设置新 filter 值
style.filter = ralpha.test( filter ) ?
    filter.replace( ralpha, opacity ) :
    filter + " " + opacity;
}
};


```

## 12.3 Facade（外观）模式

如本书前面部分所示, *Facade* 模式为更大的(可能更复杂)的代码体提供了一个更简单的接口。

我们可以经常在 jQuery 库中发现 Facade, 它使开发人员能够方便地访问实现, 以处理 DOM 操作、动画以及特别有趣的跨浏览器 Ajax。

下面是 jQuery 的\$.ajax()外观:

```

$.get( url, data, callback, dataType );
$.post( url, data, callback, dataType );
$.getJSON( url, data, callback );
$.getScript( url, callback );

```

这些是在后台进行转变:

```

// $.get()
$.ajax({
    url: url,
    data: data,

```

```

    dataType: dataType
}).done(callback);

// $.post
$.ajax({
  type: "POST",
  url: url,
  data: data,
  dataType: dataType
}).done(callback);

// $.getJSON()
$.ajax({
  url: url,
  dataType: "json",
  data: data,
}).done(callback);

// $.getScript()
$.ajax({
  url: url,
  dataType: "script",
}).done(callback);

```

更有趣的是，上面的外观实际上是自身功能的外观，在后台隐藏大量的复杂性。

这是因为 jQuery 核心中的 `jQuery.ajax()` 实现是一段非凡代码，至少可以这么说。至少，它规范了 XHR(XMLHttpRequest)之间的跨浏览器差异，使我们易于执行常见的 HTTP 操作（如 `get`、`post` 等），使用（延迟异步）`Deferreds`，等。

由于需要一整章的内容来介绍与上述外观有关的所有代码，这里只是列出用于规范 XHR 的 jQuery 核心中的代码：

```

// 创建 xhrs 的函数
function createStandardXHR() {
  try {
    return new window.XMLHttpRequest();
  } catch (e) { }
}

function createActiveXHR() {
  try {
    return new window.ActiveXObject("Microsoft.XMLHTTP");
  } catch (e) { }
}

// 创建 request 请求对象

```

```

jQuery.ajaxSettings.xhr = window.ActiveXObject ?
    * 微软在 IE7 里没有正确实现 XMLHttpRequest (不能请求本地文件)
    * 所以如果 ActiveXObject 可用, 优先使用
    * 另外 IE7/IE8i 的 XMLHttpRequest 也许被禁用掉, 所以需要一个 fallback
    */
function () {
    return !this.isLocal && createStandardXHR() || createActiveXHR();
} :

// 在其他所有浏览器上, 使用标准的 XMLHttpRequest 对象
createStandardXHR;
...

```

下面的代码块也是实际 jQuery XHR (jqXHR) 实现的 high level 实现, 实际上它是我们最常与之交互的方便外观:

```

// 请求远程文档
jQuery.ajax({
    url: url,
    type: type,
    dataType: "html",
    data: params,
    // 请求完成时的回调 (内部使用了 responseText )
    complete: function( jqXHR, status, responseText ) {
        // 保存 jqXHR 里的 responseText
        responseText = jqXHR.responseText;
        // 如果成功, 注入 HTML 到匹配的元素里
        if ( jqXHR.isResolved() ) {
            // 获取真实的 response 响应, 如果在 ajaxSetting.ajax 里设置了 dataFilter 的话
            jqXHR.done(function( r ) {
                responseText = r;
            });
            // 判断是否指定了选择器
            self.html( selector ?
                // 创建虚拟 div 保存结果
                jQuery("")
                    // 注入文档里的内容, 删除 script 标记避免 IE 里的“拒绝访问”
                    .append(responseText.replace(rscript, ""))
                // 定位指定的元素
                .find(selector) :

                // 如果没有, 则注入完整的结果
                responseText );
        }

        if ( callback ) {
            self.each( callback, [ responseText, status, jqXHR ] );
        }
    }
}

```

```
        }
    });

    return this;
}

```

## 12.4 Observer（观察者）模式

我们之前了解的另一种模式是 Observer (Publish/Subscribe) 模式，这里是系统中的对象可订阅其他对象的地方，并在感兴趣的事件发生时获得通知。

jQuery 核心具有对 Publish/Subscribe 类系统的内置支持已经有几年了，它被称为 *自定义事件*。

jQuery 早期版本中，使用 `jQuery.bind()` (`subscribe`)、`jQuery.trigger()` (`publish`) 和 `jQuery.unbind()` (`unsubscribe`) 可以访问这些自定义事件，但在最新版本中，可以使用 `jQuery.on()`、`jQuery.trigger()` 和 `jQuery.off()` 来实现这一目的。

这里我们可以看到练习示例：

```
// 等价于 subscribe(topicName, callback)
$(document).on("topicName", function () {
    //...执行相应行为
});

// 等价于 publish(topicName)
$(document).trigger("topicName");

// 等价于 unsubscribe(topicName)
$(document).off("topicName");

```

调用 `jQuery.on()` 和 `jQuery.off()` 最终要通过 jQuery 事件系统。与 Ajax 类似，由于其实现相对较长，我们可以了解自定义事件的实际事件处理程序在哪里以及如何附加的：

```
jQuery.event = {

    add: function( elem, types, handler, data, selector ) {

        var elemData, eventHandle, events,
            t, tns, type, namespaces, handleObj,
            handleObjIn, quick, handlers, special,
```

```

    ...
    // 如果是第一个元素的话，初始化该元素的事件结构和主处理程序
    events = elemData.events;
    if ( !events ) {
        elemData.events = events = {};
    }
    ...

    // 处理通过空格分离的多个事件
    // jQuery(...).bind("mouseover mouseout", fn);
    types = jQuery.trim( hoverHack(types) ).split( " " );
    for ( t = 0; t < types.length; t++ ) {

        ...

        // 如果是第一个， 初始化事件处理队列
        handlers = events[ type ];
        if ( !handlers ) {
            handlers = events[ type ] = [];
            handlers.delegateCount = 0;

            // 如果指定的事件处理程序返回 false, 只使用 addEventListener/attachEvent
            if ( !special.setup || special.setup.call( elem, data,
                //namespaces, eventHandle ) === false ) {
                // 绑定全局事件处理程序到元素上
                if ( elem.addEventListener ) {
                    elem.addEventListener( type, eventHandle, false );

                } else if ( elem.attachEvent ) {
                    elem.attachEvent( "on" + type, eventHandle );
                }
            }
        }
    }
}

```

针对那些更喜欢使用 Observer 模式传统命名方案的人群，Ben Alman (<https://gist.github.com/661855>) 在上述方法周围创建了一个简单的包装器，提供访问 `jQuery.publish()`、`jQuery.subscribe` 和 `jQuery.unsubscribe` 方法的权限。我曾在本书的前面部分提到了它们，但我们在下面可以看到完整的包装器代码。

```

(function ($) {

    var o = $({});

    $.subscribe = function () {
        o.on.apply(o, arguments);
    };
})

```

```

$.unsubscribe = function () {
    o.off.apply(o, arguments);
};

$.publish = function () {
    o.trigger.apply(o, arguments);
};

}(jQuery));

```

在最新版本的 jQuery 中，可以使用多用途回调对象（`jQuery.Callbacks`），让用户基于回调函数列表编写新的解决方案。使用这种功能编写的这种解决方案是另一种 Publish/Subscribe 系统。其实现如下所示：

```

var topics = {};

jQuery.Topic = function (id) {
    var callbacks,
        topic = id && topics[id];
    if (!topic) {
        callbacks = jQuery.Callbacks();
        topic = {
            publish: callbacks.fire,
            subscribe: callbacks.add,
            unsubscribe: callbacks.remove
        };
        if (id) {
            topics[id] = topic;
        }
    }
    return topic;
};

```

也可以像下面这样使用：

```

// 订阅者
$.Topic("mailArrived").subscribe(fn1);
$.Topic("mailArrived").subscribe(fn2);
$.Topic("mailSent").subscribe(fn1);

// 发布者
$.Topic("mailArrived").publish("hello world!");
$.Topic("mailSent").publish("woo! mail!");

// 这里，当"mailArrived"通知发布时，"hello world!"推送到fn1和fn2上
// "mailSent"通知发布时，"woo! mail!"也推送到fn1上了

```

```
// 输出:  
// hello world!  
// fn2 输出: hello world!  
// woo! mail!
```

## 12.5 Iterator（迭代器）模式

Iterator 是一种设计模式，其中，迭代器（允许我们遍历集合的所有元素的对象）顺序访问聚合对象的元素，无需公开其基本形式。

迭代器封装特定迭代如何发生的内部结构。对于 jQuery 的 `jQuery.fn.each()` 迭代器，我们实际上能够使用 `jQuery.each()` 后面的底层代码来遍历一个集合，而不需要阅读或理解提供这种功能的后台工作代码。

这种模式可以被视为一种特殊的 facade，我们显式地处理与迭代相关的问题。

```
$ .each( [ "john", "dave", "rick", "julian" ] , function( index, value ) {  
    console.log( index + ": " + value);  
});  
  
$( "li" ).each( function ( index ) {  
    console.log( index + ": " + $( this ).text());  
});
```

这里我们可以看到 `jQuery.fn.each()` 的代码：

```
// 为每个匹配的元素执行一个 callback 回调  
each: function( callback, args ) {  
    return jQuery.each( this, callback, args );  
}
```

随后是 `jQuery.each()` 后面的代码，它处理了两种遍历对象的方法：

```
each: function( object, callback, args ) {  
    var name, i = 0,  
        length = object.length,  
        isObj = length === undefined || jQueryisFunction( object );  
  
    if ( args ) {  
        if ( isObj ) {  
            for ( name in object ) {  
                if ( callback.apply( object[ name ], args ) === false ) {  
                    break;  
                }  
            }  
        }  
    }  
},
```

```

        }
    } else {
        for ( ; i < length; ) {
            if ( callback.apply( object[ i++ ], args ) === false ) {
                break;
            }
        }
    }

// 特殊的，快速处理的场景，适用于大多数 each 遍历
} else {
    if ( isObj ) {
        for ( name in object ) {
            if ( callback.call( object[ name ], name, object[ name ] ) ===
false ) {
                break;
            }
        }
    } else {
        for ( ; i < length; ) {
            if ( callback.call( object[ i ], i, object[ i++ ] ) === false ) {
                break;
            }
        }
    }
}

return object;
};

```

## 12.6 延迟初始化

延迟初始化是一种设计模式，它能够延迟昂贵的过程，直到它的第一个实例需要时。其中一个示例就是 jQuery 中的`.ready()`函数，当 DOM 准备就绪时，它仅执行一次回调。

```

$( document ).ready( function () {

    // DOM 就绪之前 ajax 请求不会执行

    var jqxhr = $.ajax({
        url: "http://domain.com/api/",
        data: "display=latest&order=ascending"
    })
    .done( function( data ) ){

```

```

        $(".status").html( "content loaded" );
        console.log( "Data output:" + data );
    });

});

```

jQuery.fn.ready()是由jQuery.bindReady()驱动，如下所示：

```

bindReady: function() {
    if ( readyList ) {
        return;
    }
    readyList = jQuery.Callbacks( "once memory" );

    // 浏览器事件发生后，这里调用$(document).ready()
    if ( document.readyState === "complete" ) {
        // 异步处理允许脚本延迟执行
        return setTimeout( jQuery.ready, 1 );
    }

    // Mozilla, Opera 和 webkit 支持该事件
    if ( document.addEventListener ) {
        // 使用事件处理回调
        document.addEventListener( "DOMContentLoaded", DOMContentLoaded,
false );

        // window.onload 后的事件，永远可用
        window.addEventListener( "load", jQuery.ready, false );
    }

    // 如果 IE 事件模型可用
} else if ( document.attachEvent ) {
    // 确保 onload 之前触发事件，可能晚点，但安全，对 iframe 也是
    document.attachEvent( "onreadystatechange", DOMContentLoaded );

    // window.onload 后的事件，永远可用
    window.attachEvent( "onload", jQuery.ready );
}

// 如果是 IE，并且不是 frame，继续检查文档是否就绪
var toplevel = false;

try {
    toplevel = window.frameElement == null;
} catch(e) {}

if ( document.documentElement.doScroll && toplevel ) {
    doScrollCheck();
}
},

```

虽然它不是直接用于 jQuery 核心，但一些开发人员可能也很熟悉通过 jQuery 插件 (<http://www.appelsiini.net/projects/lazyload>) 进行的 LazyLoading 的概念。

延迟加载（LazyLoading）实际上与延迟初始化是相同的，是一种在需要时加载页面上额外数据的技术（例如，当用户翻到页面的最后时）。近年来，这种模式已经变得相当优秀，目前可以在 Twitter 和 Facebook UI 中找到它的身影。

## 12.7 Proxy（代理）模式

有些时候，我们有必要控制对象后面的访问权限和上下文，这就是 Proxy 模式有用的地方。

当一个昂贵的对象应被实例化时，Proxy 模式可以帮助我们对其进行控制，提供高级的方法来引用对象，或修改对象，让它在特定的上下文中以一种特殊方式发挥作用。

在 jQuery 核心中，存在 `jQuery.proxy()` 方法，它接受函数作为参数，并返回一个始终具有特定上下文的新对象。这确保了函数中的 `this` 值是我们所需要的值。

在下面的示例中它就很有用，在 `click` 事件处理程序内使用计时器时使用它。想象在添加任何计时器之前我们有以下处理程序：

```
$("button").on("click", function () {
    // 这里的函数，this 是指被点击的元素
    $(this).addClass("active");
});
```

如果我们想在添加 `active` 类之前添加指定延迟，我们可以使用 `setTimeout()` 来实现这一目的。可惜的是，这个解决方案有一个小问题：无论函数传递给 `setTimeout()` 什么，在该函数内都将会用于 `this` 的不同值。它将引用 `window` 对象，这不是我们所希望的。

```
$("button").on("click", function () {
    setTimeout(function () {
        // this 没有引用我们的元素，而是引用到 window 对象上了
        $(this).addClass("active");
    });
});
```

为了解决这个问题，我们可以使用 `jQuery.proxy()` 来实现代理模式类型。通过使用我们希望赋给 `this` 的函数和值来调用它，它实际上会返回一个函数，它会在正确的上下文保留该值。如下所示：

```
$("button").on("click", function () {  
    setTimeout($.proxy(function () {  
        // 这里的 this 即引用到我们所想的元素上了  
        $(this).addClass("active");  
    }, this), 500);  
  
    // 后面传递给$.proxy()的 this 所引用的就是我们的 DOM 元素  
});
```

可以在下面代码中找到 `jQuery` 的 `jQuery.proxy()` 实现：

```
// 绑定函数到上下文 (context) 上，参数可选  
proxy: function( fn, context ) {  
    if ( typeof context === "string" ) {  
        var tmp = fn[ context ];  
        context = fn;  
        fn = tmp;  
    }  
  
    // 快速判断 fn 是否可以调用，如果不可以，返回 undefined  
    if ( !jQueryisFunction( fn ) ) {  
        return undefined;  
    }  
  
    // 模拟绑定  
    var args = slice.call( arguments, 2 ),  
        proxy = function() {  
            return fn.apply( context, args.concat( slice.call( arguments ) ) );  
        };  
  
    // 为原始的处理程序设置一个唯一的 guid  
    // so it can be removed  
    proxy.guid = fn.guid = fn.guid || proxy.guid || jQuery.guid++;  
    return proxy;  
}
```

## 12.8 Builder（生成器）模式

使用 DOM 时，我们通常想动态创建新元素——一种增加复杂性的过程，取决于我们希望所创建元素包含的最终标记、属性和特性。

复杂的元素在定义时需要特别注意，特别是如果我们想灵活地从字面上定义元素的最终标记（可能会很混乱）或转而选择更易读的面向对象路由。拥有用于创建复杂的 DOM 对象并独立于对象本身的机制能够提供这种灵活性，而这正是 Builder 模式所提供的。

Builder 使我们能够仅通过指定对象的类型和内容就可以创建复杂的对象，而无需显式创建或表示该对象。

使用 jQuery 美元符号可以实现这项功能，因为它提供了很多不同的方式来动态创建新 jQuery（和 DOM）对象，或者是通过为元素、部分标记和内容传递完整的标记，或者是使用 jQuery 来创建：

```
$(" <div class= \"foo\">bar</div> " );  
  
$(" <p id=\"test\">foo <em>bar</em></p> ").appendTo("body" );  
  
var newParagraph = $(" <p /> ").text( "Hello world" );  
$( "<input />" )  
    .attr({ "type": "text", "id": "sample" });  
    .appendTo("#container");
```

下面是 jQuery 核心的内部 `jQuery.prototype` 方法中的一个片段，有助于从传递给 `jQuery()` 选择器的标记中创建 jQuery 对象。

不管 `document.createElement` 是否是用于创建新元素，该元素的引用（已发现或创建）被注入到返回的对象中，因此之后马上就可以轻松使用 `.attr()` 等高级方法。

```
// 处理: $(html) -> $(array)  
if ( match[1] ) {  
    context = context instanceof jQuery ? context[0] : context;  
    doc = ( context ? context.ownerDocument || context : document );  
  
    // 如果仅仅传入一个单独的字符串并且是单独的标签，则创建元素并且忽略剩余部分  
    ret = rsingleTag.exec( selector );  
  
    if ( ret ) {  
        if ( !jQuery.isPlainObject( context ) ) {  
            selector = [ document.createElement( ret[1] ) ];  
            jQuery.fn.attr.call( selector, context, true );  
        } else {  
            selector = [ doc.createElement( ret[1] ) ];
```

```
        }

    } else {
        ret = jQuery.buildFragment( [ match[1] ], [ doc ] );
        selector = ( ret.cacheable ? jQuery.clone(ret.fragment)
        : ret.fragment ).childNodes;
    }

    return jQuery.merge( this, selector );
}
```

# jQuery 插件设计模式

jQuery 插件开发已经历多年的发展，不再是只有一种插件编写方法，而是有很多种方法。事实上，某些插件设计模式或许比其他模式更适于解决特定的问题或更好地用于某组件。

一些开发人员可能希望使用 jQuery UI widget factory (<http://ajpiano.com/widgetfactory/>)；它可以很好地用于复杂、灵活的 UI 组件。有些人则可能不这样认为。

有些人构建其插件时可能更像是在构建模块（类似于模块模式）或使用更现代的模块格式，如 AMD。

有些人可能会希望自己的插件能够利用原型继承的力量。其他人可能想使用自定义事件或 Publish/Subscribe 来传达插件与剩余应用程序之间的信息，诸如此类。

注意到为创建通用 jQuery 插件样例文件而做出的一系列努力之后，我开始思考插件模式。这种样例文件在理论上是一个伟大的想法，而事实上，我们很少以一种固定的方式并一直使用单一的模式来编写插件。

假设我们已经尝试在某个时候编写自己的 jQuery 插件，喜欢将有效的内容拼在一起。这是起作用的。它完成了它需要完成的任务，但或许我们会觉得它的结构可以更好。或许它可以更加灵活，或可以解决更多开发人员经常遇到的问题。如果这一切听起来都很熟悉，那么你可能会发现这一章是很有用的。在本书中，我们将探索一些其他开发人员使用的运行良好的 jQuery 插件模式。



### 注意：

本章是针对中级到高级开发人员，虽然我们在开始前将简要地回顾一些 jQuery 插件的基础知识。

如果你认为现在还没有做好准备，我很乐意为你推荐官方 jQuery 插件/程序编写指南 (<http://docs.jquery.com/Plugins/Authoring>)，Ben Alman 的插件设计指南 (<http://msdn.microsoft.com/en-us/magazine/ff696759.aspx>) 和 Remy Sharp 的“编写糟糕 jQuery 插件的信号” (<http://remysharp.com/2010/06/03/signs-of-a-poorly-written-jquery-plugin/>)”作为学习本节内容开始之前的阅读材料。

## 13.1 模式

jQuery 插件有几个具体的规则，这是其在整个社区内实现方式多样性的原因之一。在最基本的层面上，我们只需向 jQuery 的 `jQuery.fn` 对象添加一个新函数属性来编写插件，如下所示：

```
$ .fn.myPluginName = function () {  
    // 插件逻辑  
};
```

它非常的简洁，但下面的代码将是创建插件的更好方式：

```
(function ($) {  
    $ .fn.myPluginName = function () {  
        // 插件逻辑  
    };  
})(jQuery);
```

在这里，我们在一个匿名函数中包装了插件逻辑。为了确保使用\$标志作为快捷变量，不在 jQuery 和其他 JavaScript 库之间产生冲突，我们只需将它传递给这个闭包，可以将 jQuery 对象映射到美元符号上。这将确保它不受执行范围之外的任何事物的影响。

另一种编写该模式的方式是使用 `jQuery.extend()`，它使我们能够一次性定义多个函数，有时会在语义上使其更有意义：

```
(function ($) {  
    $.extend($.fn, {
```

```
myplugin: function () {
    // 插件逻辑
}
});
})(jQuery);
```

我们现在已经了解了 jQuery 插件的一些基本知识，但要更进一步掌握它还有更多的事要做。*Lightweight Start* 是我们将要探索的第一个完整的插件设计模式，它涵盖了一些我们在基本日常插件开发中可以使用的最佳实践，重视值得应用的常见陷阱。



我将会介绍下列的大多数模式，建议大家阅读代码中的注释，因为它们将在为什么要应用某些最佳实践方面提供更多的见解。

我还要提到的是，没有前期的工作、投入和 jQuery 社区中其他成员的建议，所有这一切都不可能实现。每种模式我都在注释中列了出来，这样如果你有兴趣，就可以仔细研究他们的个人工作。

## 13.2 Lightweight Start 模式

让我们开始深入研究遵循最佳实践的插件模式的基础（包括 jQuery 插件程序编写指南中的最佳实践）。这种模式对于插件开发新手或只是想实现简单功能（如 Utility 插件）的开发人员来说是很理想的模式。*Lightweight start* 使用下列内容：

- 常见最佳实践，例如，放在函数调用之前的分号（我们将在下面的注释中了解其原因）。
- `window`、`document` 和 `undefined` 作为参数传入。
- 基本的默认对象。
- 简单的插件构造函数，用于与初始化创建相关的逻辑，以及用于所使用元素的赋值。
- 扩展有默认值的选项。
- 构造函数周围的 `lightweight` 包装器，帮助避免多实例等问题。

- 遵守 jQuery 核心格式指南，以实现最优可读性。

```
/*
 * jQuery lightweight plugin boilerplate
 * Original author: @ajpiano
 * Further changes, comments: @addyosmani
 * Licensed under the MIT license
 */

// 函数调用之前的分号是为了安全的目的，防止前面的其他插件没有正常关闭。
; (function ($, window, document, undefined) {

    // 这里使用的 undefined 是 ECMAScript 3 里的全局变量 undefined，是可以修改的。undefined 没有真正传进来，以便可以确保该值是真正的 undefined。ES5 里，undefined 是不可修改的。
    // window 和 document 传递进来作为局部变量存在，而非全局变量，因为这可以加快解析流程以及影响最小化（尤其是同时引用一个插件的时候）
    // 创建默认值

    var pluginName = "defaultPluginName",
        defaults = {
            propertyName: "value"
        };
    or
    // 真正的插件构造函数
    function Plugin(element, options) {
        this.element = element;

        // jQuery 有个 extend 方法用于将两个或多个对象合并在一起，在第一个对象里进行排序。第一个对象通常为空，因为我们不想为插件的新实例影响默认的 option 选项。
        this.options = $.extend({}, defaults, options);

        this._defaults = defaults;
        this._name = pluginName;

        this.init();
    }

    Plugin.prototype.init = function () {
        // 这里处理初始化逻辑
        // 已经可以访问 DOM，并且通过实例访问 options，例如 this.element,
this.options
    };

    // 真正的插件包装，防止出现多实例
    $[fn][pluginName] = function (options) {
        return this.each(function () {
            if (!$.data(this, "plugin_" + pluginName)) {
                $.data(this, "plugin_" + pluginName,

```

```
        new Plugin(this, options));
    }
});
})(jQuery, window, document);
```

用法：

```
$("#elem").defaultPluginName({
    propertyName: "a custom value"
});
```

## 延伸阅读

插件/程序编写，jQuery

(<http://docs.jquery.com/Plugins/Authoring>)

编写糟糕 jQuery 插件的信号，Remy Sharp

(<http://remysharp.com/2010/06/03/signs-of-a-poorly-written-jquery-plugin/>)

如何创建你自己的 jQuery 插件，Elijah Manor

(<http://msdn.microsoft.com/en-us/magazine/ff608209.aspx>)

jQuery 插件风格及其重要性，Ben Almon

(<http://msdn.microsoft.com/en-us/magazine/ff696759.aspx>)

创建第一个 jQuery 插件，第 2 部分，Andrew Wirick

(<http://enterprisejquery.com/2010/07/create-your-first-jquery-plugin-part-2-revising-your-plugin/>)

## 13.3 完整的 Widget Factory 模式

jQuery 插件程序编写指南是一本介绍插件开发的好书，它不会让那些必须定期处理的常见插件平台任务变得难以理解。

jQuery UIWidget Factory 是该问题的解决方案，帮助我们基于面向对象原则创建复

杂、有状态的插件。它还简化了与插件实例之间的通信，混编一些我们在使用基本的插件时将不得不编写的重复任务。

有状态的插件帮助我们追踪其当前状态，还使我们能够在初始化后改变插件的属性。

**Widget Factory** 最优秀的一个地方是大多数 jQuery UI 库实际上都是将它作为其组件的基础来使用。这意味着如果我们正在寻找超越该模式结构的进一步指导，我们只需要查看 GitHub 上的 jQuery UI 库 (<https://github.com/jquery/jquery-ui>) 就足够了。

这个 jQuery UI Widget Factory 模式涵盖了几乎所有被支持的默认工厂方法，包括触发事件。根据上一个模式，所有使用的方法都包含了注释，内联注释内给出了进一步的指导。

```
/*
 * jQuery UI Widget-factory plugin boilerplate (for 1.8/9+)
 * Author: @addyosmani
 * Further changes: @peolanha
 * Licensed under the MIT license
 */

; (function ($, window, document, undefined) {

    // 在指定的命名空间下创建 widget，命名空间可作为额外的参数传递进来
    // $.widget( "namespace.widgetname", (optional) -已经存在的可被继承的
    // widget 原型，是一个对象字面量

    $.widget("namespace.widgetname", {

        // 可以作为默认值的 Options
        options: {
            someValue: null
        },

        // 创建 widget (例如创建元素，应用主题，绑定时间等)
        _create: function () {

            // _create 在第一次调用 widget 的时候创建，在此防止 widget 初始化代
            // 码，然后通过 this.element 可以访问调用该 widget 的元素。
            // 可以通过 this.options this.element.addStuff() 访问上面定义
            // 的 options
        },

        // 销毁插件实例，清理 widget 在 DOM 上的修改
    });
});
```

```

        destroy: function () {
            // this.element.removeStuff();
            // 对 UI 1.8, destroy 必须调用基类的 widget
            $Widget.prototype.destroy.call(this);
            // 对 UI 1.9, 定义 _destroy 代替, 不必担心调用基类的 widget
        },
        methodB: function (event) {
            // _trigger 派发插件用户可以订阅的回调 callback
            // 用法: _trigger( "callbackName" , [eventObject], [uiObject] )
            // 例如 this._trigger( "hover" , e /*where e.type == "mouseenter" */ ,
        { hovered: $(e.target)} );
            this._trigger("methodA", event, {
                key: value
            });
        },
        methodA: function (event) {
            this._trigger("dataChanged", event, {
                key: value
            });
        },
        // 响应用户通过 option 方法修改的任何值
        _setOption: function (key, value) {
            switch (key) {
            case "someValue":
                // this.options.someValue = doSomethingWith( value );
                break;
            default:
                // this.options[ key ] = value;
                break;
            }
            // 对于 UI 1.8, 必须手工调用基类 widget 中的 _setOption
            $Widget.prototype._setOption.apply(this, arguments);
            // 对 UI 1.9, 可以使用 _super 代替
            // this._super( "_setOption" , key, value );
        }
    );
} (jQuery, window, document);

```

用法:

```

var collection = $("#elem").widgetName({
    foo: false
});

collection.widgetName("methodB");

```

## 延伸阅读

jQuery UIWidget Factory

(<http://ajpiano.com/widgetfactory/#slide1>)

有状态的插件和 Widget Factory 介绍, Doug Neiner

(<http://msdn.microsoft.com/en-us/magazine/ff706600.aspx>)

Widget Factory (已解释), Scott Gonzalez

(<http://wiki.jqueryui.com/w/page/12138135/Widget%20factory>)

了解 jQuery UI Widget: 指南, Hacking at 0300 (<http://bililite.com/blog/understanding-jquery-ui-widgets-a-tutorial/>)

## 13.4 嵌套命名空间插件模式

如前所述, 处理代码的命名空间是一种避免在全局命名空间内与其他对象和变量产生冲突的方法。它们之所以重要, 是因为如果另一个页面上的脚本与我们使用相同的变量或插件名称, 我们想防止插件崩溃。作为全局命名空间的好公民, 鉴于同样的问题, 我们也必须竭尽所能不阻止其他开发人员执行脚本。

JavaScript 并不真的像其他语言那样设有命名空间的内置支持, 但它确实拥有能够用于达到类似效果的对象。采用顶级对象作为命名空间的名称, 我们可以很容易地检查页面上另一个具有相同名称对象的存在。如果这种对象不存在, 那么我们会定义它; 如果它确实存在, 那么我们简单地利用插件对它进行扩展。

对象 (或者是对象字面量) 可以用于创建嵌套命名空间, 如 `namespace.subnamespace.pluginName` 等。但为简单起见, 下面的命名空间样例文件应为我们开始了解这些概念提供一切所需。

```
/*
 * jQuery namespaced "Starter" plugin boilerplate
 * Author: @dougneiner
```

```

* Further changes: @addyosmani
* Licensed under the MIT license
*/
;

(function ($) {
    if (!$.myNamespace) {
        $.myNamespace = {};
    };

    $.myNamespace.myPluginName = function (el, myFunctionParam, options) {
        // 避免作用域问题，使用 base 代替 this 来引用内部事件和函数中的 this
        var base = this;

        // 访问 jQuery 和元素的 DOM 版本
        base.$el = $(el);
        base.el = el;

        // 为 DOM 对象添加一个反向引用
        base.$el.data("myNamespace.myPluginName", base);

        base.init = function () {
            base.myFunctionParam = myFunctionParam;

            base.options = $.extend({}, 
                $.myNamespace.myPluginName.defaultOptions, options);

            // 这里放置初始化代码
        };

        // 示例函数，取消注释即可使用
        // base.functionName = function( parameters ){
        // 
        // };
        // 运行初始化函数
        base.init();
    };

    $.myNamespace.myPluginName.defaultOptions = {
        myDefaultValue: ""
    };

    $.fn.myNamespace_myPluginName = function
        (myFunctionParam, options) {
        return this.each(function () {
            (new $.myNamespace.myPluginName(this,
                myFunctionParam, options));
        });
    };
}
;
```

```
    };  
}) (jQuery);
```

用法：

```
$("#elem").mynamespace_myPluginName({  
    myDefaultValue: "foobar"  
});
```

## 延伸阅读

JavaScript 中的命名空间，Angus Croll

(<http://javascriptweblog.wordpress.com/2010/12/07/namespacing-in-javascript/>)

使用\$.fn jQuery 命名空间，Ryan Florence

(<http://ryanflorence.com/use-your-fn-jquery-namespace/>)

JavaScript 命名空间，Peter Michaux

(<http://michaux.ca/articles/javascript-namespacing>)

JavaScript 中的模块和命名空间，Axel Rauschmayer

(<http://www.2ality.com/2011/04/modules-and-namespaces-in-javascript.html>)

## 13.5 自定义事件插件模式（使用 Widget Factory）

在这本书的第 9 章中，我们讨论了 Observer 模式，然后涵盖了 jQuery 自定义事件的支持，提供了用于实现 Publish/Subscribe 的类似解决方案。编写 jQuery 插件时可以使用同样的模式。

这里的基本思想是，当应用程序中发生有趣的事情时，页面中的对象可以发布事件通知。然后其他对象订阅（或监听）这些事件并作出相应的反应。这导致应用程序的逻辑被进一步地解耦，因为每个对象不再需要直接与另一个对象通信。

在下面的 jQuery UIWidget Factory 模式中，我们将实现一个基本的基于事件的自定

义 Publish/Subscribe 系统，它允许插件从其他的应用程序中订阅事件通知，该 Publish/Subscribe 系统将负责发布它们。

```
/*
 * jQuery custom-events plugin boilerplate
 * Author: DevPatch
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

// 本模式，使用 jQuery 自定义事件添加 pub/sub 能力到 widget 上。
// 每个 widget 都能发布特定的事件并且可以订阅其他事件。这种方式显著解耦了 widget，使得它们的函数独立

; (function ($, window, document, undefined) {
    $widget("ao.eventStatus", {
        options: {

        },
        _create: function () {
            var self = this;

            //self.element.addClass( "my-widget" );

            //订阅到 "myEventStart"
            self.element.on("myEventStart", function (e) {
                console.log("event start");
            });

            //订阅到 "myEventEnd"
            self.element.on("myEventEnd", function (e) {
                console.log("event end");
            });

            //取消订阅到 "myEventStart"
            //self.element.off( "myEventStart", function(e){
            //    //console.log( "unsubscribed to this event" );
            //});
        },
        destroy: function () {
            $.Widget.prototype.destroy.apply(this, arguments);
        },
    });
})(jQuery, window, document);
```

```
// 发布事件通知  
// $(".my-widget").trigger("myEventStart");  
// $(".my-widget").trigger("myEventEnd");
```

用法：

```
var el = $("#elem");  
el.eventStatus();  
el.eventStatus().trigger("myEventStart");
```

## 延伸阅读

jQuery UI Widget 之间的通信， Benjamin Sternthal

(<http://www.devpatch.com/2010/03/communication-between-jquery-ui-widgets/>)

## 13.6 使用 DOM-to-Object Bridge 模式的原型继承

如前所述，在 JavaScript 中，我们看不到在其他经典编程语言中所拥有的类的传统概念，但它确实有原型继承。通过原型继承，对象可以继承自另一个对象。我们可以将这个概念应用于 jQuery 插件开发中。

Yepnope.js 作者 Alex Sexton (<http://alexsexton.com/>) 和 jQuery 团队成员 Scott Gonzalez (<http://scottgonzalez.com/>) 已详细地探讨过这个主题。总之，他们发现，对于有组织的模块化开发，从插件生成过程中清晰分离定义插件逻辑的对象本身是有益的。

益处是：测试插件代码变得非常容易，并且我们也能够调整后台事物的工作方式，而无需改变我们实现的任何对象 API 的使用方式。

在 Sexton 所写的关于这个主题的文章中，他实现了一个桥接，使我们能够将一般逻辑附加至特定的插件，我们已在下面的模式中实现了这一操作。

这种模式的另一个优势是：我们不需要一直重复相同的插件初始化代码，从而确保维持 DRY 开发背后的概念。一些开发人员可能还发现这种模式比其他模式更容易阅读。

```
/*!  
 * jQuery prototypal inheritance plugin boilerplate  
 * Author: Alex Sexton, Scott Gonzalez
```

```

 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

// myObject -一个描述我们想模拟的概念, 例如一辆汽车 car
var myObject = {
    init: function (options, elem) {
        // 将默认 options 与传递进来的 options 混入在一起
        this.options = $.extend({}, this.options, options);

        // 保存元素引用, jQuery 引用和正常引用都保存
        this.elem = elem;
        this.$elem = $(elem);

        // 构建 DOM 的初始化结构
        this._build();

        // 返回 this, 以便可以链式使用
        return this;
    },
    options: {
        name: "No name"
    },
    _build: function () {
        //this.$elem.html( "<h1>" +this.options.name+ "</h1>" );
    },
    myMethod: function (msg) {
        // 可以直接访问相关的和缓存的 jQuery 元素
        // this.$elem.append( "<p>" +msg+ "</p>" );
    }
};

// 测试 Object.create, 为不支持的浏览器提供 create 支持
if (typeof Object.create !== "function") {
    Object.create = function (o) {
        function F() { }
        F.prototype = o;
        return new F();
    };
}

// 基于定义的对象上创建插件
$.plugin = function (name, object) {
    $.fn[name] = function (options) {
        return this.each(function () {
            if (!$.data(this, name)) {

```

```
        $.data(this, name, Object.create(object).init(  
            options, this));  
    }  
});  
};  
};
```

用法：

```
$.plugin("myobj", myObject);

$("#elem").myobj({ name: "John" });

var collection = $("#elem").data("myobj");
collection.myMethod("I am a method");
```

延伸阅读

使用继承模式来组织大型 jQuery 应用， Alex Sexton

(<http://alexsexton.com/blog/2010/02/using-inheritance-patterns-to-organize-large-jquery-applications/>)

如何使用 jQuery 或其他框架管理大型应用程序（进一步讨论）, Alex Sexton

(<http://www.slideshare.net/SlexAxton/how-to-manage-large-jquery-apps>)

需要使用原型继承的实例， Neeraj Singh

(<http://www.slideshare.net/SlexAxton/how-to-manage-large-jquery-apps>)

JavaScript 中的原型继承, Douglas Crockford

(<http://javascript.crockford.com/prototypal.html>)

## 13.7 jQuery UI Widget Factory Bridge 模式

如果你喜欢基于上一个设计模式中的对象生成插件这种方式，那么你可能会对jQuery UI Widget Factory 中发现的`$.widget.bridge` 方法感兴趣。

这个桥接基本上是作为 JavaScript 对象（使用`$.widget` 创建）和 jQuery 核心 API 之

间的中间层，提供更为内置的解决方案，以实现基于对象的插件定义。实际上，我们可以使用自定义构造函数创建有状态的插件。

此外，`$.widget.bridge` 提供了很多其他的功能，包括：

- 像在经典 OOP 中一样，进行公有和私有方法的处理，与预期一致（即，暴露了公有方法，而隐藏了私有方法）。
- 对多个初始化的自动保护。
- 已传递对象实例的自动生成，及它们在所选择元素内部`$data` 缓存中的存储。
- 初始化后可以更改选项。

欲进一步了解有关如何使用这种模式的信息，请参阅下面的内联注释：

```
/*
 * jQuery UI Widget factory "bridge" plugin boilerplate
 * Author: @erichynds
 * Further changes, additional comments: @addyosmani
 * Licensed under the MIT license
 */

// "widgetName" 对象构造器
// required: 必须接收 2 个参数,
// options: 配置选项参数
// element: 在之上创建实例的 DOM 元素
var widgetName = function (options, element) {
    this.name = "myWidgetName";
    this.options = options;
    this.element = element;
    this._init();
}
// "widgetName" 原型
widgetName.prototype = {

    // widget 第一次调用的时候, _create 会自动运行
    _create: function () {
        // creation code
        // 创建代码
    },
}
```

```

    // required: 插件的初始化逻辑代码放在_init中，第一次创建实例或初始化之后再次
尝试初始化widget（通过桥接）时，会触发_init
    _init: function () {
        // 初始化代码
    },
    // required: 使用桥接的对象必须包含一个option。深度初始化，改变options的
逻辑放在这里
    option: function (key, value) {
        // 可选：如果不需要则可以忽略获取/修改options的深度初始化
        // 用法：$("#foo").bar({ cool:false });

        if ($.isPlainObject(key)) {
            this.options = $.extend(true, this.options, key);
        }

        // 用法：$("#foo").option("cool"); - getter
        // else if (key && typeof value === "undefined") {
        //     return this.options[key];
        }

        // 用法：$("#foo").bar("option", "baz", false );
        // else {
        //     this.options[key] = value;
        }

        // required: option必须返回当前实例，在元素上重写初始化实例时，首先调用
option，然后链式到_init方法
        return this;
    },
    // 注意公有方法没有使用下划线
    publicFunction: function () {
        console.log("public function");
    },
    // 私有方法使用了下划线
    _privateFunction: function () {
        console.log("private function");
    }
};

```

用法：

```

// 在foo命名空间下，连接widget对象到jQuery的API
$.widget.bridge("foo", widgetName);

// 创建widget的实例
var instance = $("#foo").foo({

```

```
        baz: true
    });

// elem 的数据上已经存在 widget 实例了
// 输出: #elem
console.log(instance.data("foo").element);

// 桥接允许我们调用公有方法
// 输出: "public method"
instance.foo("publicFunction");

// 桥接阻止调用内部方法
instance.foo("_privateFunction");
```

## 延伸阅读

在 Widget Factory 之外使用\$.widget.bridge, Eric Hynds

(<http://www.erichynds.com/jquery/using-jquery-ui-widget-factory-bridge/>)

## 13.8 使用 Widget Factory 的 jQuery Mobile Widget

jQuery Mobile 是一种 jQuery 项目框架，鼓励在流行移动设备、平台及桌面上进行 web 应用程序的设计。我们只需编写一次代码，它就应能够在很多目前的 A-、B- 和 C- 级浏览器中运行，而不是为每个设备或操作系统编写单独的应用程序。

jQuery Mobile 背后的基本原理也可以应用到插件和小部件开发中。

有趣的是，在下一个模式中，虽然在编写“移动”优化的小部件时有些细微的差别，但那些很早以前就熟悉使用 jQuery UIWidgetFactory 模式的人应该能够很快掌握这种模式。

下面移动优化过的小部件与我们之前看到标准 UI 小部件模式相比有一些有趣的差别：

- `$.mobile.widget` 被引用为现有的小部件原型，从中用于继承。对于标准窗口小部件，传入任何这样的原型对于基本开发都是不必要的，但使用此这种特定的 jQuery Mobile widget 原型为进一步的“选项（options）”格式提供了内部访问。

- 在`_create()`中提供了官方 jQuery Mobile widget 如何处理元素选择的指南，以选择更适于 jQM 标记的基于角色的方法。这并不是说不推荐标准选择，只是考虑到 jQuery Mobile 页面的结构，这种方法可能更有意义。
- 注释形式中还提供了指南，以便在`pagecreate`上应用插件方法，并通过数据角色和数据属性选择插件应用程序。

```
/*
 * (jQuery mobile) jQuery UI Widget-factory plugin boilerplate (for 1.8/9+)
 * Author: @scottjehl
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

;(function ( $, window, document, undefined ) {

    // 在我们选择的命名空间下定义一个 widget，这里使用了第一个参数里的 mobile
    $.widget( "mobile.widgetName", $.mobile.widget, {

        // Options 作为默认值
        options: {
            foo: true,
            bar: false
        },

        _create: function() {
            // widget 第一次调用的时候，_create 会自动运行，在这里放置 widget 的
            // 初始化代码，然后即可访问在其之上调用 widget 的元素。
            // 上面定义的 options 可以通过 this.options 进行访问

            // var m = this.element,
            // p = m.parents( ":jqmData(role='page')"),
            // c = p.find( ":jqmData(role='content')");
        },

        // 私有方法和属性以下划线开头
        _dosomething: function(){ ... },

        // 下面这样的公有方法可以在外部进行调用
        // $("#myelem").foo( "enable", arguments );

        enable: function() { ... },

        // 销毁实例化后的插件，并且清理 DOM 上 widget 所做的修改
        destroy: function () {
            // this.element.removeStuff();
        }
    });
});
```

```

        // 对于 UI 1.8, destroy 必须调用基类的 widget
        $Widget.prototype.destroy.call( this );
        //对于 UI 1.9, 定义_destroy代替, 不必担心调用基类的 widget
    },

    methodB: function ( event ) {
        // _trigger 派发插件用户可以订阅的回调 callback
        // signature: _trigger( "callbackName" , [eventObject],
        // [uiObject] )
        // e.g. this._trigger( "hover", e /*where e.type ==
        // "mouseenter"*/, { hovered: $(e.target)} );
        this._trigger( "methodA", event, {
            key: value
        });
    },

    methodA: function ( event ) {
        this._trigger( "dataChanged", event, {
            key: value
        });
    },

    // 响应用户通过 option 方法修改的任何值
    _setOption: function ( key, value ) {
        switch ( key ) {
        case "someValue":
            // this.options.someValue = doSomethingWith( value );
            break;
        default:
            // this.options[ key ] = value;
            break;
        }
    }

    // 对于 UI 1.8, 必须手工调用基类 widget 中的 _setOption
    $Widget.prototype._setOption.apply(this, arguments);
    // 对 UI 1.9, 可以使用 _super 代替
    // this._super( "_setOption", key, value );
}

})( jQuery, window, document );

```

用法:

```

var instance = $( "#foo" ).widgetName({
    foo: false
});

instance.widgetName( "methodB" );

```

每当 jQuery Mobile 中的新页面创建时，我们还可以自动初始化这个小部件。当 jQuery Mobile 页面（通过 `data-role="page"` 属性发现的页面）第一次初始化时，jQuery Mobile 的页面插件派发一个 `create` 事件。当新页面创建时，我们可以监听该事件（称为 `pagecreate`）并自动运行插件。

```
$ (document).on("pagecreate", function ( e ) {  
    // 这里，e.target 引用的是创建的页面 (pagecreate 事件的 target)，所以可以很方便地在该页面上查找匹配所指定选择器的元素，然后在元素上面调用我们的插件。  
    // 下面的代码表示在所有 data-role 属性为 foo 的元素上调用 foo 插件  
    $(e.target).find( "[data-role='foo']" ).foo( options );  
  
    // 或者，稍微好点的方案，通过配置的 data-attribute 命名空间编写选择器  
    $( e.target ).find( ":jqmData(role='foo')" ).foo( options );  
});
```

我们现在可以简单地引用脚本，它包含运行 jQueryMobile 网站页面的小部件和 `pagecreate` 绑定，并且它将像任何其他 jQueryMobile 插件那样自动运行。

## 13.9 RequireJS 和 jQuery UI Widget Factory

如第 11 章中所述，RequireJS 是 AMD 兼容脚本加载器，它提供了一个用于将应用程序逻辑封装在可管理模块内部的简洁解决方案。它能够以正确的顺序加载模块（通过其 `order` 插件），通过其优秀的 `r.js` 优化器简化混合脚本的过程，并在每个模块的基础上提供定义动态依赖。

在下列样例文件模式中，我们将演示如何定义 AMD（因此是 RequireJS）兼容的 jQuery UI 小部件，如下：

- 允许小部件模块依赖的定义，建立在前面介绍的 jQuery UI Widget Factory 模式的基础上。
- 演示了传入 HTML 模板资产的方法，以创建模板化的小部件（使用 Underscore.js 微模板）。
- 包括调整快速提示，如果我们希望稍后将它传递给 RequireJS 优化器，我们就可以把它添加到小部件模块中。

```

/*
 * jQuery UI Widget + RequireJS module boilerplate (for 1.8/9+)
 * Authors: @jrburke, @addyosmani
 * Licensed under the MIT license
 */

// Note from James:
//
// 假设我们使用了 RequireJS 和 jQuery 文件，下面的文件都在同一目录下
//
// - require-jquery.js
// - jquery-ui.custom.min.js (custom jQuery UI build with widget factory)
// - templates/
// - asset.html
// - ao.myWidget.js

// 然后可以像如下这样构建 widget 小部件了：

// ao.myWidget.js file:
define("ao.myWidget", ["jquery", "text!templates/asset.html", "underscore",
"jquery-ui.custom.min"], function ($, assetHtml, _) {

    // 在我们指定的命名空间下定义 widget，这里使用 ao 做演示
    $.widget("ao.myWidget", {

        // Options 作为默认值
        options: {},

        // 设置 widget (例如创建元素，应用主题，绑定事件等)
        _create: function () {

            // widget 第一次调用的时候，_create 会自动运行，在这里放置 widget
            // 的初始化代码，然后即可访问在其之上调用 widget 的元素。
            // 上面定义的 options 可以通过 this.options 进行访问// this.
            element.addStuff();

            // this.element.addStuff();

            // We can then use Underscore templating with
            // with the assetHtml that has been pulled in
            // var template = _.template( assetHtml );
            // this.content.append( template() );
        }

        // 销毁实例化后的插件，并且清理 DOM 上 widget 所做的修改
        destroy: function () {
            // this.element.removeStuff();
            // 对于 UI 1.8, destroy 必须调用基类的 widget
            $.Widget.prototype.destroy.call(this);
        }
    });
});

```

```

    //对于 UI 1.9, 定义 _destroy 代替, 不必担心调用基类的 widget
    },

    methodB: function (event) {
        // _trigger 派发插件用户可以订阅的回调 callback
        // signature: _trigger( "callbackName" , [eventObject],
        // [uiObject] )
        this._trigger("methodA", event, {
            key: value
        });
    },

    methodA: function (event) {
        this._trigger("dataChanged", event, {
            key: value
        });
    },
}

//响应用户通过 option 方法修改的任何值
_setOption: function (key, value) {
    switch (key) {
    case "someValue":
        // this.options.someValue = doSomethingWith( value );
        break;
    default:
        // this.options[ key ] = value;
        break;
    }
}

// 对于 UI 1.8, 必须手工调用基类 widget 中的 _setOption
$.Widget.prototype._setOption.apply(this, arguments);
// 对 UI 1.9, 可以使用 _super 代替
// this._super( "_setOption", key, value );
}
);
}
);

```

### 13.9.1 用法

下面是 index.html 代码:

```
<script data-main="scripts/main" src="requirejs">
</script>
```

下面是 main.js 代码:

```
require({
    paths: {
```

```
        "jquery": "https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
jquery.min",
        "jqueryui": "https://ajax.googleapis.com/ajax/libs/jqueryui/1.
8.18/jquery-ui.min",
        "boilerplate": "../patterns/jquery.widget-factory.requirejs.
boilerplate"
    }
}, ["require", "jquery", "jqueryui", "boilerplate"],
function (req, $) {

    $(function () {

        var instance = $("#elem").myWidget();
        instance.myWidget("methodB");
    });
});
});
```

### 13.9.2 延伸阅读

和 jQuery 一起使用 RequireJS, Rebecca Murphrey

(<http://jqfundamentals.com/#example-10.5>)

使用 jQuery 和 RequireJS 的快速模块化代码, James Burke

(<http://speakerrate.com/talks/2983-fast-modular-code-with-jquery-and-requirejs>)

jQuery 最好的朋友, Alex Sexton

(<http://jquerysbestfriends.com/#slide1>)

使用 RequireJS 管理依赖, Ruslan Matveev

(<http://www.angrycoding.com/2011/09/managing-dependencies-with-requirejs.html>)

## 13.10 全局选项和单次调用可重写选项（最佳选项模式）

下一个模式，我们来看看为插件定义的最优化方式的配置选项和默认值。大多数人都很熟悉的定义插件选项的方法是传递带有默认值的对象字面量给\$.extend()，比如

基本插件样例文件中所演示的。

然而，如果我们使用可定制化选项的插件，以便用户能够重写全局配置或单次调用的配置，那么就可以进一步优化相应的结构。

相反，通过引用在插件命名空间中显式定义的选项对象（例如`$fn.pluginName.options`），并在开始被调用时，将它与任何传递到该插件的选项合并，用户可以选择在插件初始化时传递选项或在插件之外重写选项（如下面演示的那样）。

```
/*
 * jQuery "best options" plugin boilerplate
 * Author: @cowboy
 * Further changes: @addyosmani
 * Licensed under the MIT license
 */

;(function ( $, window, document, undefined ) {

    $.fn.pluginName = function ( options ) {

        // 这是重写默认选项的最佳实践，显示定义$.fn.pluginName.options，然后传递第二个options，两者合并在一起使用。
        // 这样在全局使用或者单次调用方面都可以使用

        options = $.extend( {}, $.fn.pluginName.options, options );

        return this.each(function () {

            var elem = $(this);

            });

    };
    // 全局重写选项
    // 显示通过重写插件的默认options来达到目的
    // 也可以只改变其中一个键值，比如$.fn.pluginName.options.key="otherval";

    $.fn.pluginName.options = {

        key: "value",
        myMethod: function ( elem, param ) {
        }
    };
})( jQuery, window, document );
```

用法：

```
$("#elem").pluginName({
  key: "foobar"
});
```

## 延伸阅读

jQuery Pluginization (<http://benalman.com/talks/jquery-pluginization.html>) 和 Ben Alman 一起写的 gist (<https://gist.github.com/472783/e8bf47340413129a8abe5fac55c83336efb5d4e1>)

## 13.11 高可配和高可变的插件模式

在这种模式中，类似于 Alex Sexton 的原型继承插件模式，插件的逻辑不是嵌套在 jQuery 插件中。而是使用构造函数和在其原型上定义的对象字面量来定义插件逻辑。然后 jQuery 用于插件对象的实际实例化。

通过使用两个小技巧可以进入定制化的下一个阶段，其中一个技巧已经在前面的模式中看到过：

- 可以以全局和元素的单个集合重写选项。
- 通过 HTML5 数据属性可以在单个元素级别上定制选项（如下所示）。改进插件行为可以应用于一组元素，但要进行内联定制，而不需要实例化每个元素让其拥有不同的默认值。

我们不认为后者选项的使用很频繁，但是它可以是一个更简洁的解决方案（如果我们不介意内联方法的话）。如果你想知道它可以用在哪里，想象为一组元素编写一个拖拽插件。我们可以像下面这样定制其选项：

```
$(".item-a").draggable({ "defaultPosition": "top-left" });
$(".item-b").draggable({ "defaultPosition": "bottom-right" });
$(".item-c").draggable({ "defaultPosition": "bottom-left" });
//等等
```

但使用模式内联方法，如下代码是可以实现的：

```
$( ".items" ).draggable();

html
<li class="item" data-plugin-options="{"defaultPosition":"top-left"}> </div>
```

```
<li class="item" data-plugin-options="{"defaultPosition": "bottom-left"}>  
</div>
```

等等。我们很可能会优先考虑这些方法之一，但它仅是另一个值得注意的变体。

```
/*  
 * "Highly configurable" mutable plugin boilerplate  
 * Author: @markdalgleish  
 * Further changes, comments: @addyosmani  
 * Licensed under the MIT license  
 */  
  
// 关于该模式，Alex Sexton 说该插件的逻辑并没有嵌套在 jQuery 插件里，而是仅仅使用  
jQuery 用于初始化。  
  
; (function ($, window, document, undefined) {  
  
    // 插件构造器  
    var Plugin = function (elem, options) {  
        this.elem = elem;  
        this.$elem = $(elem);  
        this.options = options;  
  
        // 下一行代码是 HTML5 data 属性的高级应用，用于为单个元素定义自定义属性。例如：  
        // <div class="item" data-plugin-options="{"message": "Goodbye  
        World!"}></div>  
        this.metadata = this.$elem.data("plugin-options");  
    };  
  
    // 插件原型  
    Plugin.prototype = {  
        defaults: {  
            message: "Hello world!"  
        },  
  
        init: function () {  
            // 将默认选项进行扩展，options 是对象字面量  
            this.config = $.extend({}, this.defaults, this.options,  
                this.metadata);  
  
            // 用法样例：  
            // 为每个实例设置 message:  
            // $("#elem").plugin( { message: "Goodbye World!" } );  
            // 或者  
            // var p = new Plugin( document.getElementById( "elem" ) ,  
            // { message: "Goodbye World!" }).init()  
            // 或者，设置全局的默认 message:  
    };  
});
```

```
// Plugin.defaults.message = "Goodbye World!"  
  
this.sampleMethod();  
return this;  
},  
  
sampleMethod: function () {  
    // 例如，显示当前配置的 message  
    // console.log(this.config.message);  
    }  
}  
  
Plugin.defaults = Plugin.prototype.defaults;  
  
$.fn.plugin = function (options) {  
    return this.each(function () {  
        new Plugin(this, options).init();  
    });  
};  
  
// 可选: window.Plugin = Plugin;  
  
})(jQuery, window, document);
```

用法:

```
$("#elem").plugin({  
    message: "foobar"  
});
```

## 延伸阅读

创建高可配的 jQuery 插件，Mark Dalgleish

(<http://markdalgleish.com/2011/05/creating-highly-configurable-jquery-plugins/>)

编写高可配的 jQuery 插件，第 2 部分，Mark Dalgleish

(<http://markdalgleish.com/2011/09/html5data-creating-highly-configurable-jquery-plugins-part-2/>)

## 13.12 是什么使插件超越模式

总而言之，设计模式仅仅是编写可维护 jQuery 插件的一个方面。还有很多其他的

因素是值得考虑的，我想分享一下我自己选择第三方插件来解决一些其他问题的标准。希望这将有助于提高插件项目的整体质量。

### 13.12.1 质量

遵守你所编写的 JavaScript 和 jQuery 的最佳实践。是否努力使用 jsHint 或 jsLint 编写插件代码？编写的插件是否是最佳的？

### 13.12.2 代码风格

插件是否遵守一贯的代码风格指南？如 jQuery 核心风格指南（[http://docs.jquery.com/JQuery\\_Core\\_Style\\_Guidelines](http://docs.jquery.com/JQuery_Core_Style_Guidelines)），如果不是，你编写的代码是否至少是相对清晰和易读呢？

### 13.12.3 兼容性

插件与 jQuery 的哪些版本兼容？最近是否使用了 jQuery-git 最新发布版或最新稳定版进行了测试？如果是在 jQuery 1.6 出现之前编写的插件，那么可能会有属性和特性方面的问题，因为在发布时这些内容已经做了修改。

新版本的 jQuery 做了一些改进，为 jQuery 项目提供了改进核心库内容的机会。当我们寻找更好的做事方式时，它出现了偶然破坏（主要是主版本）。我希望看到插件作者在必要时更新他们的代码，或者至少用新版本来测试他们的插件，以确保一切都按照预期工作。

### 13.12.4 可靠性

插件应该有它自己的单元测试。这些不仅能够证明它实际上是按预期的运行，而且也能够改进设计，而不影响最终用户的使用。我认为单元测试对于任何用于生产环境中的重要 jQuery 插件都是必不可少的，它们并不是那么难以编写。

欲寻找使用 QUnit 进行 JavaScript 自动化测试方面的优秀指南，你可能会对 Jörn Zaefferer (<http://bassistance.de/>) 的《使用 QUnit 进行的自动化 JavaScript 测试 (<http://msdn.microsoft.com/en-us/magazine/gg749824.aspx>)》感兴趣。

### 13.12.5 性能

如果插件需要执行的任务包含广泛的处理或大量的 DOM 操作，我们应该遵循最佳

实践来确定基准点，以有助于最小化这类工作。使用 [jsPerf.com](http://jsPerf.com) 来测试代码的片段，以判断它在不同浏览器中的执行情况，并发现需要进一步优化的地方。

### 13.12.6 文档

如果目的是便于其他开发人员使用该插件，那么要确保提供良好的文档记录。记录 API 以及如何使用插件。插件支持哪些方法和选项？它是否有用户需要注意的陷阱？如果用户无法弄清如何使用该插件，他们很可能会寻找另一种替代方法。文档也会极大地帮助我们为插件代码编写注释。迄今为止，这是你可以为其他开发人员能够提供的最好礼物。如果有人觉得他们可以浏览你的代码库，可以充分使用或改进它，就说明你干得漂亮。

### 13.12.7 维护的可能性

发布一个插件时，要估计它可能需要的维护和支持时间是多少。我们都喜欢在社区内分享我们的插件，但我们需要预估自己解答问题、解决问题并不断改进的能力。仅通过在 *README* 文件的前面陈述项目维护支持的意图就可以做到这一点。

## 13.13 总结

到目前为止，在这一章中，我们已经探讨了一些省时的设计模式和最佳实践，以用于改进 jQuery 插件的编写方式。有一些模式比其他模式更适用于某些用例，但我希望总的来说这些模式都是有用处的。

要记住，在选择某种模式时，重要的是它的实用性。不要为了使用而使用插件模式，而是要花时间了解其底层结构，并确定它是否能够很好地解决问题或适用于你要创建的组件。

## 13.14 命名空间模式

在本节中，我们将探讨 JavaScript 中的命名空间模式。命名空间可以被认为是唯一标识符下代码单元的逻辑分组。该标识符可以在很多命名空间中被引用，并且每个标识符自身可以包含它自己的嵌套（或子）命名空间层级。

在应用程序开发中，我们有很多重要的原因去使用命名空间。在 JavaScript 中，它们帮助我们防止与全局命名空间中的其他对象或变量产生冲突。它们也可用于帮助组织代码库中的功能块，这样它就更易于引用和使用。

处理任何重要脚本或应用程序的命名空间都是至关重要的，因为在页面上的另一个脚本使用与我们相同的变量或方法名称的情况下，它对于防止代码冲突是非常重要的。由于现在第三方标签的数量会经常注入到页面中，在某种程度上，这可能是在我们的工作中需要解决的一个常见问题。作为全局命名空间中行为端正的“公民”，由于相同的问题，尽力不去阻止执行其他开发者的脚本也是很重要的。

JavaScript 确实像其他语言那样没有用于命名空间的内置支持，但是它有可以用于实现类似效果的对象和闭包。

## 13.15 命名空间基础

几乎可以在任何重要的 JavaScript 应用程序中都能找到命名空间。除非我们正在使用一个简单的代码片段，否则我们必须竭尽所能确保我们正在实现的命名空间是正确的，因为它不仅容易上手，并且也会避免第三方代码影响我们自己的代码。我们将在这节中考察的模式是：

1. 单一全局变量
2. 前缀命名空间
3. 对象字面量表示法
4. 嵌套命名空间
5. 立即调用的函数表达式
6. 名称空间注入

### 13.15.1 单一全局变量

JavaScript 中一个流行的命名空间模式是选择一个全局变量作为主要的引用对象。

其框架实现如下所示，其中我们返回一个拥有函数和属性的对象：

```
var myApplication = (function () {
    function(){
        //...
    },
    return{
        //...
    }
})();
```

虽然这种方法适用于某些情况，但单一全局变量模式最大的挑战是确保没有其他人像我们一样在页面中使用相同的全局变量名称。

### 13.15.2 命名空间前缀

正如 Peter Michaux 所提到的，上述问题的其中一个解决方案是使用命名空间前缀。这本质上是一个简单的概念，但其思想是选择一个我们希望使用的独特前缀命名空间（在该示例中是 `myApplication_`），然后在前缀后面定义任何方法、变量或其他对象，如下所示：

```
var myApplication_propertyA = {};
var myApplication_propertyB = {};
function myApplication_myMethod() {
    //...
}
```

从减少全局作用域内的某个特殊变量的几率的角度来看，它是有效的，但是记住，唯一命名的对象可以有相同的效果。

除此以外，这种模式的最大问题是，一旦应用程序开始扩展，它会产生大量的全局对象。在全局命名空间内，它对任何其他开发人员都没有使用的这个前缀有严重的依赖，所以如果选择使用这种模式一定要特别注意。

欲获得更多有关 Peter 对单一全局变量模式的看法，请阅读他的精彩文章：<http://michaux.ca/articles/javascript-namespacing>。

### 13.15.3 对象字面量表示法

我们在 Module 模式部分介绍的对象字面量表示法可以被认为是包含一组键值对的

对象，每一对键和值由冒号进行分隔，键也可以代表新的命名空间。

```
var myApplication = {  
    // 正如如下代码，我们可以很容易的为此对象字面量定义功能  
    getInfo: function () {  
        //...  
    },  
  
    // 但也可以让其支撑进一步的对象命名空间，来包含其他想包含的数据  
    models: {},  
    views: {  
        pages: {}  
    },  
    collections: {}  
};
```

我们还可以选择将属性直接添加至命名空间：

```
myApplication.foo = function () {  
    return "bar";  
}  
  
myApplication.utils = {  
    toString: function () {  
        //...  
    },  
    export: function () {  
        //...  
    }  
}
```

对象字面量的优点是：不会污染全局命名空间，并在逻辑上协助组织代码和参数。如果你希望创建易读、可以扩展到支持深层嵌套的结构，它们是很有用的。与简单的全局变量不同，对象字面量也通常会考虑到同名变量的存在的测试，因此冲突发生的概率明显降低。

下一个示例演示了不同的方法，可以查看一个对象命名空间是否存在，如果不存在就对它进行定义。

```
// 下面的代码并没有检测 myApplication 在全局命名空间内是否存在，很差的代码实践，因为该代码可能重新改写了以及存在的别的同名对象  
var myApplication = {};  
  
// 下面的代码均检测了变量和命名空间是否存在，如果存在就使用该实例，否则为 myApplication 创建一个新实例
```

```
//  
// Option 1: var myApplication = myApplication || {};  
// Option 2 if( !MyApplication ){ MyApplication = {} };  
// Option 3: window.myApplication || ( window.myApplication = {} );  
// Option 4: var myApplication = $.fn.myApplication = function() {};  
// Option 5: var myApplication = myApplication === undefined ? {} :  
myApplication;
```

你会经常看到开发人员选择选项 1 或选项 2——这两个选项都是直接可用的，并在最终结果方面都是相同的。

选项 3 假设你在使用全局命名空间，它也可以写成：

```
myApplication || (myApplication = {});
```

这种变体假设 `myApplication` 已经初始化，因此它对于参数/参数场景是唯一真正有用的，如下面的示例所示：

```
function foo() {  
    myApplication || ( myApplication = {} );  
}  
  
// myApplication 还没有初始化，所以 foo 抛出引用错误  
  
foo();  
  
// However accepting myApplication as an argument  
  
function foo( myApplication ) {  
    myApplication || ( myApplication = {} );  
}  
  
foo();  
  
// 尽管 myApplication 等于 undefined，但不会出错，因为 myApplication 的值是 {}  
  
// 第 4 种方式对编写 jQuery 插件非常有用，下面是定义插件的形式：  
var myPlugin = $.fn.myPlugin = function() { ... };  
  
// Then later rather than having to type:  
$.fn.myPlugin.defaults = {};  
  
// 可以这样做  
myPlugin.defaults = {};
```

这会带来更好的压缩性（代码压缩），并可以减少作用域查询。

选项 5 有点类似于选项 4，不过是一个长格式，评估 `myApplication` 在内联是否是

`undefined` 的，如果不是，将它定义为一个对象；如果是，设置为 `myApplication` 的现有值。

只有考虑全面性时才会这样显示，但在大多数情况下，选项 1-4 将能够大大满足大多数需求。

当然还有大量用于对象字面量如何以及在何处组织和创建代码的变体。对于希望暴露特定的自封闭模块嵌套 API 的小型应用程序，你会发现自己已经使用了我们在本书前面介绍的 **Revealing Module 模式**：

```
var namespace = (function () {  
    // 定义局部作用域  
    var privateMethod1 = function () { /* ... */ },  
        privateMethod2 = function () { /* ... */ }  
        privateProperty1 = "foobar";  
  
    return {  
  
        // 这里返回的对象字面量可以包含很多层嵌套，不过这种方式对小程序是最有用的，比如我的个人程序里限制作用域  
        publicMethod1: privateMethod1,  
  
        // 拥有公有属性的嵌套命名空间  
        properties:{  
            publicProperty1: privateProperty1  
        },  
  
        // 其他嵌套命名空间  
        utils:{  
            publicMethod2: privateMethod2  
        }  
        ...  
    }  
}());
```

对象字面量的好处是，它们为我们提供了非常优雅的键/值语法；我们能够轻松地封装应用程序中任何不同的逻辑或功能，与其他内容进行清晰分离，并为扩展代码提供坚实的基础。

```
var myConfig = {  
  
    language: "english",
```

```

        defaults: {
            enableGeolocation: true,
            enableSharing: false,
            maxPhotos: 20
        },
        theme: {
            skin: "a",
            toolbars: {
                index: "ui-navigation-toolbar",
                pages: "ui-custom-toolbar"
            }
        }
    }
}

```

请注意，JSON 是对象字面量表示法的一个子集，它与上述对象字面量介绍（如，JSON 键必须是字符串）之间只有微小的语法差异。如果出于任何原因，希望使用 JSON 来存储配置数据（如，在发送到后端时为实现更简单的存储），你可以大胆去做。欲了解更多有关对象字面量模式的信息，我推荐阅读 [Rebecca Murphrey](#) 关于该主题的优秀文章，因为里面包含了一些我们没有触及的领域。

### 13.15.4 嵌套命名空间

对象字面量模式的扩展是嵌套命名空间。它是另一种冲突风险较低的常见模式，因为即使一个命名空间已经存在，它也不太可能有同样的嵌套子对象。

下面代码看起来熟悉吗？

```
YAHOO.util.Dom.getElementsByClassName("test");
```

Yahoo 的旧版本 YUI 库通常使用嵌套对象命名空间模式。我在 AOL 任工程师时，我们也在很多更大型的应用程序中使用这种模式。嵌套命名空间的示例实现如下：

```

var myApp = myApp || {};
// 定义嵌套子对象时，检测是否存在
myApp.routers = myApp.routers || {};
myApp.model = myApp.model || {};
myApp.model.special = myApp.model.special || {};
// 嵌套命名空间可以很复杂：
// myApp.utilities.charting.html5.plotGraph(/*..*/);

```

```
// myApp.modules.financePlanner.getSummary();  
// myApp.services.social.facebook.realtimeStream.getLatest();
```



以上内容与 YUI3 将命名空间作为模块处理不同，YUI3 使用拥有较短命名空间的沙箱 API 主机对象。

我们也可以选择声明新的嵌套命名空间/属性作为索引属性，如下所示：

```
myApp["routers"] = myApp["routers"] || {};  
myApp["models"] = myApp["models"] || {};  
myApp["controllers"] = myApp["controllers"] || {};
```

这两个选项都很易读，且有组织性，并提供相对安全的应用程序命名空间的方式，采用与我们在其他语言中使用的类似的方式。但唯一真正需要告诫的是，它需要浏览器的 JavaScript 引擎首先定位 myApp 对象，然后进一步挖掘，直到它达到我们实际希望使用的函数。

这可能意味着查找工作量的增加，但是，Juriy Zaytsev (<http://twitter.com/kangax>) 等开发人员之前已经测试并发现单一对象命名空间与“嵌套”方法之间的性能差异是很微不足道的。

### 13.15.5 立即调用的函数表达式 (IIFE)

在本书前面部分，我们简要介绍了立即调用函数表达式的概念；IIFE (<http://benalman.com/news/2010/11/immediately-invoked-function-expression/>) 实际上是匿名函数，被定义后立即被调用。如果这听起来很熟悉，那是因为你以前可能见到过它被引用为自动执行的（或自我调用）的匿名函数，但是我认为 Ben Alman 的 IIFE 命名更加准确。在 JavaScript 中，由于变量和函数都是在这样一个只能在内部进行访问的上下文中被显式地定义，函数调用提供了一种实现私有的便捷方式。

IIFE 是用于封装应用程序逻辑的常用方法，以保护它免受全局名称空间的影响，但在命名空间的世界里也有它们的使用。

可以在下面找到 IIFE 的示例：

```
// 匿名的立即调用的函数表达式  
(function () { /*...*/ })();
```

```
//立即调用的有名函数表达式  
(function foobar() { /*..*/ })();  
  
//自执行函数  
function foobar() { foobar(); }
```

第一个示例的扩展版本如下所示：

```
var namespace = namespace || {};  
  
//这里命名空间对象作为函数参数进行传递，然后在此对象上赋值公有方法和属性  
(function (o) {  
    o.foo = "foo";  
    o.bar = function () {  
        return "bar";  
    };  
}) (namespace);  
  
console.log(namespace);
```

在具有可读性的同时，该示例可以进行极大的扩展，以解决常见的开发问题，如定义私有级别（公有/私有函数和变量）以及方便的命名空间扩展。让我们来看一些更多的代码：

```
//命名空间（我们的命名空间名称）和undefined作为参数传递，确保：  
//1. 命名空间可以在局部进行修改，而不重写函数外面的上下文  
//2. undefined的参数值确保是undefined，主要是避免ES5规范里定义的可修改的undefined  
  
; (function (namespace, undefined) {  
  
    //私有属性  
    var foo = "foo",  
        bar = "bar";  
  
    //公有方法和属性  
    namespace.foobar = "foobar";  
    namespace.sayHello = function () {  
        speak("hello world");  
    };  
  
    //私有方法  
    function speak(msg) {  
        console.log("You said: " + msg);  
    };  
  
    //在全局命名空间内检测namespace是否存在，如果不存在，给window.namespace  
    //赋值一个对象字面量
```

```
})(window.namespace = window.namespace || {});  
  
// 如下方式可以测试命名空间的属性和方法  
  
// 输出: foobar  
console.log(namespace.foobar);  
  
// 输出: hello world  
namespace.sayHello();  
  
// 赋值新属性  
namespace.foobar2 = "foobar";  
  
// 输出: foobar  
console.log(namespace.foobar2);
```

可扩展性当然是任何可伸缩命名空间模式的关键，使用 IIFE 可以轻松实现这一目的。在下面的示例中，“命名空间”再次作为参数传递给匿名函数，然后进行扩展（或装饰）以拥有更多功能：

```
// 给 namespace 命名空间扩展更多新功能  
(function( namespace, undefined ){  
    // public method  
    // 公有方法  
    namespace.sayGoodbye = function () {  
        console.log( namespace.foo );  
        console.log( namespace.bar );  
        speak( "goodbye" );  
    }  
})( window.namespace = window.namespace || {});  
  
// 输出 goodbye  
namespace.sayGoodbye();
```

如果大家想获得更多关于这种模式的信息，我推荐阅读 Ben 写的有关 IIFE 的帖子 (<http://benalman.com/news/2010/11/immediately-invoked-function-expression/>)。

### 13.15.6 命名空间注入

命名空间注入是 IIFE 的另一个变体，我们从函数包装器内部为一个特定的命名空间“注入”方法和属性，使用 this 作为命名空间代理。这种模式的好处是易于将功能行为应用到多个对象或命名空间，当应用一组构建代码的基本方法时，该模式也是很有用的（例如 getter 和 setter）。

这种模式的缺点是：可能有更容易或更优的方法来实现这一目的（如深对象扩展/

合并), 我在本节的前面部分介绍过。

下面我们可以看到该模式运行的示例, 我们用它来填充两个命名空间的行为: 一个是最初定义的(utils), 另一个我们动态创建作为 utils 功能分配的一部分(称为 tools 的新命名空间)。

```
var myApp = myApp || {};
myApp.utils = {};

(function () {
    var val = 5;

    this.getValue = function () {
        return val;
    };

    this.setValue = function (newVal) {
        val = newVal;
    }

    // 也定义一个新的子命名空间
    this.tools = {};
});

// apply 方法将行为应用到 utils 命名空间上
// 为 utils 命名空间注册新的行为

// 可以通过 utilities 模块定义
(function () {
    this.diagnose = function () {
        return "diagnosis";
    }
}).apply(myApp.utils.tools);

// 同样的方式在普通的 IIFE 上扩展新功能, 仅仅将上下文作为参数传递并修改, 而不是仅仅
// 使用 this

// 用法:

// 输出命名空间
console.log(myApp);

// 输出: 5
console.log(myApp.utils.getValue());

// 设置 val 的值并返回
```

```
myApp.utils.setValue(25);
console.log(myApp.utils.getValue());

// 测试其他层次的功能
console.log(myApp.utils.tools.diagnose());
```

Angus Croll 还建议了使用调用 API 来实现上下文和参数自然分离的想法。该模式感觉更像是一个模块创建者，但作为模块，它还提供一个封装解决方案，为了周到考虑，我们将对它进行简要介绍：

```
// 定义稍后要用到的命名空间
var ns = ns || {},
    ns2 = ns2 || {};

// 模块、命名空间创建者
var creator = function (val) {

    var val = val || 0;

    this.next = function () {
        return val++;
    };

    this.reset = function () {
        val = 0;
    }
}

creator.call(ns);

// ns.next, ns.reset 此时已经存在
creator.call(ns2, 5000);
// ns2 包含了相同的方法，但是值被重写为 5000 了
```

如前所述，这种类型的模式是用于为多个模块或命名空间指定一个类似的功能基本集，但我只建议在对象/闭包内显式声明功能无法实现直接访问时再使用它。

## 13.16 高级命名空间模式

现在来研究一些我在大型应用程序开发中发现的不可估价的高级模式和实用程序，其中一些需要重新考虑应用程序命名空间的传统方法。在这里指出，我不是主张使用以下的任何命名空间方式，而是我发现在实践中它们是有效的命名方式。

### 13.16.1 自动嵌套的命名空间

正如我们所了解的，嵌套名称空间能够为代码单元提供有组织的结构层级。这种名称空间的示例可以是：*application.utilities.drawing.canvas.2d*。也可以使用对象字面量模式将它扩展为：

```
var application = {
    utilities: {
        drawing: {
            canvas: {
                2d: {
                    //...
                }
            }
        }
    }
};
```

该模式的突出挑战是：希望创建的每个附加层都需要将另一个对象定义为顶层命名空间中一些父对象的子对象。当需要多层时，它就会变得特别费力，因为应用程序的复杂性在增加。

如何解决这个问题呢？在 JavaScript 模式中，Stoyan Stefanov (<http://www.jspatterns.com/>) 提出了一个非常明智的方法在现有的全局变量中自动定义嵌套命名空间。他建议了一种便利方法，接受字符串参数作为一个嵌套，解析它，并自动用所需的对象填充基本名称空间。

他建议使用的方法如下所示，我已经对它进行了更新，使其成为一个通用函数，在遇到多个命名空间时更容易复用：

```
// 顶级命名空间分配到对象字面量上
var myApp = myApp || {};

// 通用函数，解析命名空间字符串，并自动生成嵌套的命名空间
function extend(ns, ns_string) {
    var parts = ns_string.split("."),
        parent = ns,
        pl;

    pl = parts.length;

    for (var i = 0; i < pl; i++) {
        // 属性如果不存在，则创建它
        if (!parent[parts[i]]) {
            parent[parts[i]] = {};
            parent = parent[parts[i]];
        }
    }
}
```

```

        if (typeof parent[parts[i]] === "undefined") {
            parent[parts[i]] = {};
        }
        parent = parent[parts[i]];
    }
    return parent;
}
// 用法：
// 给 myApp 扩展支持深度嵌套命名空间
var mod = extend(myApp, "modules.module2");

// 带有嵌套路径的对象输出
console.log(mod);

// 测试 mod 的实例，也可以用于测试 myApp 命名空间外部作为一个克隆
// 输出：true

console.log(mod === myApp.modules.module2);

// 使用 extend 对嵌套命名空间的进一步测试
extend(myApp, "moduleA.moduleB.moduleC.moduleD");
extend(myApp, "longer.version.looks.like.this");
console.log(myApp);

```

图 13-1 显示 Chrome Developer Tools 输出。

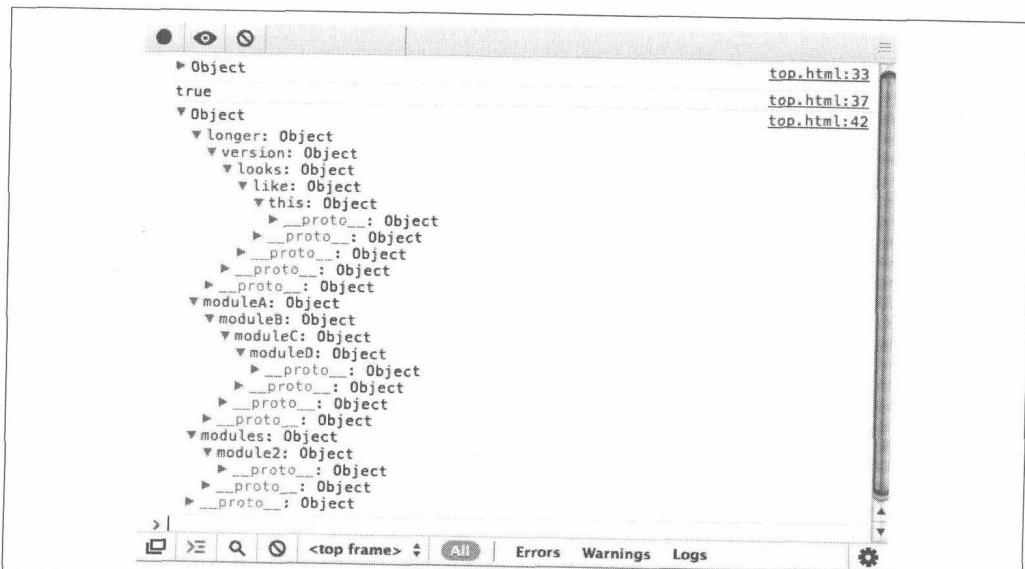


图 13-1 Chrome Developer Tools 输出

之前我们必须为其命名空间将各种嵌套显式声明为对象，现在使用单一的、简洁的

代码行就已经可以很容易实现这一目的。

### 13.16.2 依赖声明模式

现在将探讨嵌套的命名空间模式的微小扩展，我们将它称作依赖声明模式。我们都应该，对象的局部引用可以减少整体查找时间，但是让我们将它应用到命名空间中，来看看它在实践中是什么样的：

```
// 访问嵌套命名空间的统一方式
myApp.utilities.math.fibonacci(25);
myApp.utilities.math.sin(56);
myApp.utilities.drawing.plot(98, 50, 60);

// 局部/缓存引用
var utils = myApp.utilities,
maths = utils.math,
drawing = utils.drawing;

// 简便访问命名空间
maths.fibonacci(25);
maths.sin(56);
drawing.plot(98, 50, 60);

// 上述代码在成百上千次调用时，局部引用的命名空间和嵌套命名空间的性能相比显著提高
```

在这里使用局部变量几乎总是比使用顶级全局快（如 `myApp`）。它也比访问每个后续线上的嵌套属性/子命名空间更加方便和高效，并可以改进更复杂应用程序的可读性。

Stoyan 建议通过函数或模块在其函数作用域，定义局部命名空间（使用单变量模式），并称它为依赖声明模式。它提供的好处之一是减少定位依赖并解析它们，我们应该有一个在必要时将模块动态加载至命名空间的可扩展架构。

在我看来，这种模式在模块化级别上运行时的效果最好，局部化命名空间，以便用于一组方法。在每个函数级上局部化命名空间，特别是在命名空间依赖明显重叠的地方，我建议如有可能，应避免使用这种方法。相反，要进一步定义它，让它们都访问相同的引用。

### 13.16.3 深度对象扩展

自动命名空间的另一种方法是深度对象扩展。使用对象字面量表示法定义的命名空

间可以与其他对象（或命名空间）一起很容易地被扩展（或合并），这样就可以在相同的命名空间下访问这些命名空间的属性和函数。

这是实现现代 JavaScript 框架的简化方法（如 jQuery 的`$.extend`），但是，如果要使用原生 JS 扩展对象（命名空间），下面的例程可能会有所帮助。

```
// extend.js
// Andrew Dupont, 编写, Addy Osmani 优化

function extend(destination, source) {

    var toString = Object.prototype.toString,
        objTest = toString.call({});

    for (var property in source) {
        if (source[property] && objTest === toString.call(source[property])) {
            destination[property] = destination[property] || {};
            extend(destination[property], source[property]);
        } else {
            destination[property] = source[property];
        }
    }
    return destination;
};

console.group("objExtend namespacing tests");

// 定义一个顶级命名空间
var myNS = myNS || {};

// 1. 使用 utils 对象扩展命名空间
extend(myNS, {
    utils: {
    }
});

console.log("test 1", myNS);
// 此时 myNS.utils 已存在

// 2. 使用多次路径进行扩展(namespace.hello.world.wave)
extend(myNS, {
    hello: {
        world: {
            wave: {
                test: function () {
                    //...
                }
            }
        }
    }
});
```

```

        }
    }
});

// 测试直接赋值是否正常工作
myNS.hello.test1 = "this is a test";
myNS.hello.world.test2 = "this is another test";
console.log("test 2", myNS);

// 3. 如果要添加的命名空间在 myNS 里已经存在，那该如何？(例如 library)，要确保扩展
期间命名空间不被重写

myNS.library = {
    foo: function () { }
};

extend(myNS, {
    library: {
        bar: function () {
            //...
        }
    }
});

// 确认扩展的操作安全，myNS 现在也包含了 library.foo, library.bar
console.log("test 3", myNS);

// 4. 如果不想输入整个命名空间，而是只想输入部分，结果如何？

var shorterAccess1 = myNS.hello.world;
shorterAccess1.test3 = "hello again";
console.log("test 4", myNS);

// 正确，myApp.hello.world.test3 的值是"hello again"

console.groupEnd();

```



上面的实现不是对所有对象都是跨浏览器兼容的，应只当做概念验证来考虑。你可能会发现 Underscore.js 的 extend()方法是首先要学习的简单、且浏览器兼容性好的实现：<http://documentcloud.github.com/underscore/docs/underscore.html#section-67>。另外，可以在这里找到从核心中提取的 jQuery \$.extend()方法：<https://github.com/addyosmani/jquery.parts>。

对于将在应用程序中使用 jQuery 的开发人员来说，可以使用\$.extend 实现完全相同的对象名称空间可扩展性，如下所示：

```
// 顶级命名空间
var myApp = myApp || {};

// 直接赋值嵌套的命名空间
myApp.library = {
    foo: function () {
        //...
    }
};

// 将其他命名空间深度扩展（合并）到现有命名空间很有趣，也就是说命名空间有同名的名称
// 但不同的函数签名: $.extend( deep, target, object1, object2 )
$.extend(true, myApp, {
    library: {
        bar: function () {
            //...
        }
    }
});

console.log("test", myApp);
// myApp 此时不仅包含了 library.foo() 方法，也包含了 library.bar() 方法，没有东西
被重写掉
```

出于周全考虑，请在这里查阅本节命名空间实现剩余代码部分的 jQuery \$.extend 等效代码。

#### 13.16.4 推荐

回顾我们在本节中已经探讨的命名空间模式，我个人将在大多数应用程序使用的选项是采用对象字面量模式的嵌套对象命名空间。在可能的情况下，我将使用自动化嵌套命名空间来实现这个目的，但这只是个人喜好而已。

IIFE 和单一全局变量可能对中小型应用程序有效，但是，需要命名空间和深度层级命名空间的大型代码库要求有一个能够提高可读性和规模性的简洁解决方案。我认为这种模式能够实现所有这些目标。

我也推荐尝试一些建议的高级实用程序方法来进行命名空间扩展，因为从长期来看它们确实能够节约时间。

## 第 14 章

---

# 总结

这就是对 JavaScript 和 jQuery 中的设计模式探险。希望对您有所帮助。

这些设计模式，使我们很容易地，站在过去几十年中为挑战性问题定义解决方案和架构的开发人员的肩膀上想问题。本书内容应会提供足够的信息，帮助你在自己的脚本、插件和 Web 应用程序中开始使用我们介绍的这些模式。

重要的是我们要了解这些模式，但也要知道如何并且何时使用它们。在使用它们之前要先研究每种模式的利与弊。花时间来试验这些模式，以完全了解它们能够提供的东西，并基于模式对应用程序的真正价值来作出使用判断。

如果我提起了你进一步研究这个领域的兴趣，并且你也想了解更多有关设计模式的信息，很多该领域的优秀文章可以用于了解通用软件开发，当然还有 JavaScript。

我很乐意推荐如下内容：

企业应用架构模式， Martin Fowler

( <http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420> )

JavaScript 模式， Stoyan Stefanov

( [http://www.amazon.com/JavaScript-Patterns-Stoyan-Stefanov/dp/0596806752/ref=sr\\_1\\_](http://www.amazon.com/JavaScript-Patterns-Stoyan-Stefanov/dp/0596806752/ref=sr_1_)

[1?ie=UTF8&s=books&qid=1289759956&sr=1-1](http://www.douban.com/subject/1289759956/)

感谢大家阅读本书。欲获得更多有关 JavaScript 方面的学习材料, 请阅读我在  
<http://addyosmani.com> 上的博客或者 Twitter@addyosmani。

祝你在 JavaScript 中的探索之旅顺利完成, 再见!

# 参考文献

1. Robert C Martin, “Design Principles and Design Patterns”.

([http://www.objectmentor.com/resources/articles/Principles\\_and\\_Patterns.pdf](http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf))

2. Ralph Johnson, “Special Issue on Patterns and Pattern Languages”

(<http://www.cs.wustl.edu/%7Eschmidt/CACM-editorial.html>), ACM.

3. Hillside Engineering Design Patterns Library.

(<http://hillside.net/patterns/>)

4. Ross Harmes and Dustin Diaz, “Pro JavaScript Design Patterns”.

(<http://jsdesignpatterns.com/>)

5. Design Pattern Definitions.

([http://en.wikipedia.org/wiki/Design\\_Patterns](http://en.wikipedia.org/wiki/Design_Patterns))

6. Patterns and Software Terminology.

(<http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>)

7. Jeff Juday, “Reap the benefits of Design Patterns”.

(<http://artides.techrepublic.com/article/reap-the-benefits-of-design-patterns-in-software-development/5173591>)

8. Subramanyan, “Guhan, JavaScript Design Patterns”.

(<http://www.slideshare.net/rmsguhan/javascript-design-patterns>)

9. James Moaoriello, “What Are Design Patterns and Do I Need Them?”.

(<http://www.developer.com/design/article.php/1474561/What-Are-Design-Patterns-and-Do-I-Need-Them.htm>)

10. Alex Barnett, “Software Design Patterns”.

(<http://alexbarnett.net/blog/archive/2007/07/20/software-design-patterns.aspx>)

11. Gunni Rode, “Evaluating Software Design Patterns”.

(<http://www.rode.dk/thesis/>)

12. SourceMaking Design Patterns.

([http://sourcemaking.com/design\\_patterns](http://sourcemaking.com/design_patterns))

13. “The Singleton”

(<http://prototyp.ical.ly/index.php/2007/03/01/javascript-design-patterns-1-the-singleton/>), Prototyp.ical.

14. Stoyan Stevanov, “JavaScript Patterns”.

(<http://www.slideshare.net/stoyan/javascript-patterns>)

15. Design Pattern Implementations in JavaScript; discussion

(<http://stackoverflow.com/questions/24642/what-are-some-examples-of-design-pattern-implementations-using-javascript>), Stack Overflow.

16. Jared Spool, “The Elements of a Design Pattern”.

([http://www.uie.com/articles/elements\\_of\\_a\\_design\\_pattern/](http://www.uie.com/articles/elements_of_a_design_pattern/))

17. Examples of Practical JS Design Patterns; discussion

(<http://stackoverflow.com/questions/3722820/examples-of-practical-javascript-object-oriented-design-patterns>), Stack Overflow.

18. Nicholas Zakkas, “Design Patterns in JavaScript Part 1”.

(<http://www.webreference.com/programming/javascript/ncz/column5/index.html>)

19. Design Patterns in jQuery

(<http://stackoverflow.com/questions/3631039/design-patterns-used-in-the-jquery-library>), Stack Overflow.

20. Elyse Neilson, “Classifying Design Patterns By AntiClue”.

(<http://www.anticlue.net/archives/000198.htm>)

21. Douglas Schmidt, “Design Patterns, Pattern Languages, and Frameworks”.

(<http://www.cs.wustl.edu/%7Eschmidt/patterns.html>)

22. Christian Heilmann, “Show Love To The Module Pattern”.

(<http://christianheilmann.com/2007/07/24/show-love-to-the-module-pattern/>)

23. Mike G., “JavaScript Design Patterns”.

(<http://www.lovemikeg.com/2010/09/29/javascript-design-patterns/>)

24. Anoop Mashudanan, “Software Designs Made Simple”.

(<http://www.scribd.com/doc/16352479/Software-Design-Patterns-Made-Simple>)

25. Klaus Komenda, “JavaScript Design Patterns”.

(<http://www.klauskomenda.com/code/javascript-programming-patterns/>)

26. Introduction to the JavaScript Module Pattern.

(<https://www.unleashed-technologies.com/blog/2010/12/09/introduction-javascript-module-design-pattern>)

27. Design Patterns Explained.

(<http://c2.com/cgi/wiki?DesignPatterns>)

28. Mixins explained.

(<http://en.wikipedia.org/wiki/Mixin>)

29. Working with GoF’s Design Patterns In JavaScript.

([http://aspalliance.com/1782\\_Working\\_with\\_GoFs\\_Design\\_Patterns\\_in\\_JavaScript\\_Programming.all](http://aspalliance.com/1782_Working_with_GoFs_Design_Patterns_in_JavaScript_Programming.all))

30. Using Object.create.

(<http://stackoverflow.com/questions/2709612/using-object-create-instead-of-new>)

31. t3knomanster, JavaScript Design Patterns.

(<http://t3knomanser.livejournal.com/922171.html>)

32. Working with GoF Design Patterns In JavaScript Programming.

([http://aspalliance.com/1782\\_Working\\_with\\_GoFs\\_Design\\_Patterns\\_in\\_JavaScript\\_Programming.7](http://aspalliance.com/1782_Working_with_GoFs_Design_Patterns_in_JavaScript_Programming.7))

33. JavaScript Advantages of Object Literal

(<http://stackoverflow.com/questions/1600130/javascript-advantages-of-object-literal>), Stack Overflow.

34. Liam McLennan, “JavaScript Class Patterns”.

(<http://geekswithblogs.net/liammclennan/archive/2011/02/06/143842.aspx>)

35. Understanding proxies in jQuery.

(<http://stackoverflow.com/questions/4986329/understanding-proxy-in-jquery>)

36. Observer Pattern Using JavaScript.

(<http://www.codeproject.com/Articles/13914/Observer-Design-Pattern-Using-JavaScript>)

37. Speaking on the Observer pattern.

(<http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html>)

38. Singleton examples in JavaScript

([http://www.hardcode.nl/subcategory\\_1/article\\_526-singleton-examples-in-javascript.htm](http://www.hardcode.nl/subcategory_1/article_526-singleton-examples-in-javascript.htm)), Hardcode.nl.

39. Design Patterns by Gamma

(<http://exciton.cs.rice.edu/javaresources/DesignPatterns/>), Helm supplement.

---

# 作者简介

Addy Osmani 是谷歌 Chrome 团队的开发工程师，对 JavaScript 应用程序架构有着强烈的爱好。他创建了比较流行的项目，如 TodoMVC，并对 Modernizr 和 jQuery 等其它开放源代码项目也做出很大贡献。作为一位高产的博主(<http://addyosmani.com/blog>)，Addy 的文章经常出现在《JavaScript 电子周刊》、《Smashing 杂志》及很多其它出版物上。他目前负责的项目包括 Yeoman（为现代 web 开发人员提供的工作流）、Backbone Aura 和 jQuery UI Bootstrap。他在 <http://addyosmani.com> 上持续更新关于 Web 开发的博客，供那些希望更多地了解他的工作的人阅读。

---

# 译者简介

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

# 封面介绍

本书封面上的动物是一只雉鹃，拉丁名：*Dromococcyx phasianellus*。雉鹃属于鸟纲，原产于尤卡坦半岛与巴西之间的森林地区，南至哥伦比亚也有可能发现它的足迹。在这个许多鸟类物种因栖息地遭破坏而濒临灭绝的时代，雉鹃却能够努力保持住“无危物种”的保护状况。

雉鹃有一条长尾巴，羽冠较短，呈深棕色。雉鹃以昆虫为食，捕食时，它拍打着翅膀发出响声，然后飞向前用喙在地上啄食。虽然它是一种食虫动物，但它也捕食小蜥蜴和雏鸟。

像许多其他杜鹃一样，雉鹃也将自己的卵产在别的鸟类的巢里。当卵孵化时，寄主会将雉鹃的后代视为己出，幼雏也会牢记自己的寄主。出于本能，雉鹃的幼雏会将寄主的卵推出鸟巢，为自己腾出空间。与欧洲的杜鹃不同，雉鹃并不是专性巢寄生，它仍然有构建自己巢穴的能力。

封面图片版画，来源不详。