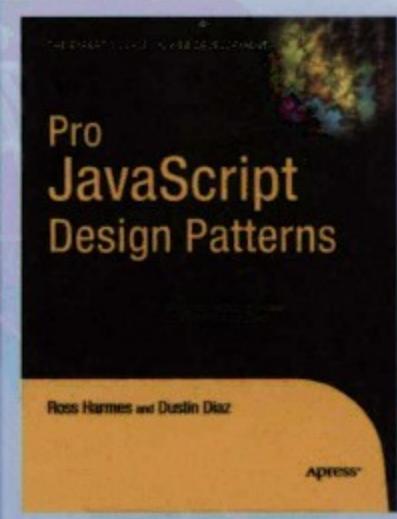


Pro JavaScript Design Patterns

# JavaScript 设计模式

[美] Ross Harmes 著  
Dustin Diaz  
谢廷晟 译

- 从这里开始，真正掌握JavaScript的精髓
- Google和Yahoo专家联手揭秘世界顶尖公司的技术内幕
- Amazon全五星盛誉图书



人民邮电出版社  
POSTS & TELECOM PRESS

**TURING** 图灵程序设计丛书 Web开发系列

Pro JavaScript Design Patterns

# JavaScript 设计模式

[美] Ross Harmes  
Dustin Diaz 著

人民邮电出版社  
北京

## 图书在版编目 (CIP) 数据

JavaScript 设计模式 / (美) 哈梅斯 (Harmes, R.),  
(美) 迪亚斯 (Diaz, D.) 著; 谢廷晟译. —北京: 人民  
邮电出版社, 2009.1

(图灵程序设计丛书)

书名原文: Pro JavaScript Design Patterns

ISBN 978-7-115-19128-1

I. J... II. ①哈…②迪…③谢 III. JAVA 语言 - 程序  
设计 IV. TP312

中国版本图书馆CIP数据核字 (2008) 第171032号

## 内 容 提 要

本书共有两部分。第一部分给出了实现具体设计模式所需要的面向对象特性的基础知识，主要包括接口、封装和信息隐藏、继承、单体模式等内容。第二部分则专注于各种具体的设计模式及其在 JavaScript 语言中的应用，主要介绍了工厂模式、桥接模式、组合模式、门面模式等几种常见的模式。为了让每一章中的示例都尽可能地贴近实际应用，书中同时列举了一些 JavaScript 程序员最常见的任务，然后运用设计模式使其解决方案变得更模块化、更高效并且更易维护，其中较为理论化的例子则用于阐明某些要点。

本书适合各层次的 Web 前端开发人员阅读和参考，也适合有 C++/Java/C# 背景的服务器端程序员学习。

图灵程序设计丛书

## JavaScript设计模式

◆ 著 [美]Ross Harmes Dustin Diaz

译 谢廷晟

责任编辑 傅志红

执行编辑 杨爽

◆ 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 电子函件 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: 800×1000 1/16

印张: 16.75

字数: 396千字

2009年1月第1版

印数: 1-4000册

2009年1月河北第1次印刷

著作权合同登记号 图字: 01-2008-5001号

ISBN 978-7-115-19128-1/TP

定价: 45.00 元

读者服务热线: (010)88593802 印装质量热线: (010)67129223

反盗版热线: (010)67171154

# 版 权 声 明

Original English language edition, entitled *Pro JavaScript Design Patterns* by Ross Harmes and Dustin Diaz, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705.

Copyright © 2008 by Ross Harmes and Dustin Diaz. Simplified Chinese-language edition copyright © 2009 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

# 对本书的赞誉

本书道前人所未道，引导你从编写代码转变为设计代码。书中绝大部分示例代码都来自YUI等实战项目，并进行了深入剖析。强烈推荐。

——Nicholas C. Zakas，著名JavaScript专家，Yahoo前端工程师，  
畅销书《JavaScript高级程序设计》作者

本书绝对值得细细品读，提供了大量有益而且有趣的JavaScript实现代码，其中对JavaScript面向对象特性的阐述是我读过的书中最出色的。

——Jeff Wilcox，微软Silverlight核心程序员

本书是JavaScript成熟的标志。两位作者详细剖析了面向对象实现的底层机制，并演示了在实战中如何灵活运用设计模式，获得优美的设计方案。

——Tiff Fehr，MSNBC.com用户体验工程师

想成为JavaScript专家？本书千万不能错过！

——Amazon.com读者评论

本书介绍的技术将使JavaScript如虎添翼。如果你要用JavaScript开发较大规模的程序，本书必不可少。

——JavaRanch.com

毫不夸张地说，这是我有生以来读到的最好的一本JavaScript图书。作者讲授了大量专家级的经验。

——Mostafa Farghaly，埃及程序员

本书是Web程序员深刻理解各种Ajax/JavaScript框架的宝典。

——theopensourcery.com

对于有一定经验的JavaScript程序员，本书绝对物超所值。想知道Google和Yahoo内部怎样开发企业级的程序吗？仔细研读吧。

——James Stewart，资深Web工程师

本书是市面上最好的一本关于JavaScript的书，适合想更多地了解JavaScript和设计模式的人。

——Amazon.com读者评论

还在怀疑JavaScript的强大？本书将使你大开眼界。

——Dzone.com

# 译者序

设计模式对于程序员来说并不是一个陌生话题。在Erich Gamma等人合著的经典著作《设计模式》出版之后，十几年间陆续出现了许多这方面的专著。不过它们大都结合Java和C++等传统的面向对象语言进行讲解，而讲述设计模式在动态语言中的实现的书则较为罕见。在早期的JavaScript编程实践中，这种语言只被用于做点为网页涂脂抹粉的小差事；程序的规模很小，也很简单。那个时候恐怕没有人会想到把设计模式用到这种“玩具语言”编写的程序中。随着Ajax技术的兴起，Web应用的许多逻辑都从服务器端转移到客户端执行，客户端JavaScript程序的作用越来越重要，其规模和复杂程度也越来越大，人们也越来越多地把面向对象方法应用到JavaScript程序设计中。在此背景下，有许多人开始研究设计模式在JavaScript程序设计中的应用，网上也陆续出现了一些关于这个话题的零星讨论。但是到目前为止，系统地探讨面向对象的程序设计模式在JavaScript语言中的实现的书，只此一本。（Michael Mahemoff所著的《Ajax设计模式》一书总结的是运用Ajax技术开发Web应用的各种设计模式，虽然也涉及大量JavaScript编程，但它与本书关注的焦点不同。本书讨论的是一些通用的面向对象设计模式在JavaScript中的实现，属于更基础性的东西，它们不仅仅适用于Web客户端编程。）

JavaScript这种语言与Java等传统的面向对象语言有很大的不同。它的动态性、词法作用域和基于原型的继承机制等特点可能会让很多初次接触它的程序员都有点不习惯，而且由于语言设计上的一些不完善，许多在传统面向对象语言中只是举手之劳的事在JavaScript中却不得不依靠hack手法来实现。这也许就是那些已经熟知设计模式在Java等语言中实现方式的程序员也需要本书的原因。本书第一部分着重讲述了面向对象技术在JavaScript中的实现方法。这对于对JavaScript只有过初步了解的人非常有用（当然，本书不适合对JavaScript一窍不通的读者。他们应该先找一本JavaScript基础教材来看看，比如人民邮电出版社出版的《JavaScript基础教程》）。Java和C++编程老手们在学完这部分内容之后，想必应该能够在JavaScript程序设计中自行应用各种经典的设计模式了。不过不同的人可能会有一些不同的做法，因此继续看看本书第二部分，借鉴一下作者的方法也不无益处。对于那些从未学过设计模式的JavaScript程序员来说，本书的重要性更是毋庸置疑。不过，坦率地说，要想深入学习设计模式仅看本书是不够的。取代Gamma等人的《设计模式》并不是本书的目标。

就个人的感觉而言，我觉得本书最大的遗憾之处在于作者在讲述各种设计模式的实现时没有使用原型式继承，而是选择了类式继承。毕竟原型式继承才是JavaScript面向对象编程最自然的继承实现方式。不过正如作者所言，“有些人似乎天生就容易被原型式继承的简洁性吸引，而另一

些人却对更面熟的类式继承情有独钟”，这只是个人口味的问题。考虑到很多人对原型式继承都非常陌生，作者的这种选择可以理解。

在本书的翻译过程中，我先后两次厚着脸皮向本书的执行编辑杨爽女士请求推迟交稿时间，她总是和颜悦色地告诉我：多花点时间不要紧，翻译质量才是最重要的。在此我要感谢杨爽女士的宽容，并感谢她和本书的文字编辑罗凌云女士认真负责的编辑工作。此外，李松峰、贺师俊、郭晓刚、刘江、麦天志、肖鹏、周琦、崔驰坤、米全喜、任斌、霍泰稳、张一宁和徐毅等朋友在本书的翻译过程中曾提供了许多宝贵意见，在此我一并表示感谢。由于我的水平有限，翻译不当之处在所难免。谨请各位读者朋友不吝赐教。建议您在图灵公司的网站（[www.turingbook.com](http://www.turingbook.com)）或互动出版网（[www.china-pub.com](http://www.china-pub.com)）上本书的网页上提出自己的意见。

谢廷晟

2008年秋

# 前　　言

目前，JavaScript到了一个转折关头。这门语言和它的用户都已经成熟起来。人们也开始认识到：它是一个复杂的课题，值得进一步研究。

设计模式运用在程序设计中已经有些年头了。它们最早被整理记录于Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides（绰号“四人帮”（the Gang of Four，后文中简写为Gof））合著的*Design Patterns*<sup>①</sup>一书中，现已被应用到各种各样的面向对象语言中。设计模式的魅力之一体现在它们被应用于各种语言和语法上时所表现出的一致性上。其基本结构是相同的，只是细节略有差别。例如，把一个用Java实现的模式转换为C++形式就很容易。

但是对JavaScript来说情况则有所不同。尽管所有那些能力JavaScript都有，但它们往往并非这种语言的正式部分，因而必须借助于一些晦涩的技巧和不那么直观的技术来模仿。这些年来，人们找到了许多方法用该语言来完成其设计者都未曾预计到的任务。那些常见的面向对象特性同样也要靠这样的手段实现。

本书收集整理了这些技巧和技术。第一部分提供了一个实现具体设计模式所需要的面向对象特性的基础。第二部分则专注于各种具体的设计模式及其在JavaScript语言中的应用。

为了让每一章中的示例都尽可能地贴近实际应用，我们花了不少心思。我们尽量列举一些JavaScript程序员最常见的任务，然后运用设计模式使其解决方案变得更模块化、更高效并且更易维护。而那些较为理论化的例子则用于阐明某些要点。我们知道，本书的价值最终将取决于它与你的日常工作和项目的紧密联系程度。

希望你能喜欢这本书。JavaScript是一种极其复杂而又灵活的语言，而且很适合于动手实践。你可以随时试试我们提供的示例代码。要是你找到实现某种设计模式的新方法，或者某种旧技术的新用途，请告诉我们一声。本书的附属网站<http://jsdesignpatterns.com>和Apress出版社的网站<http://www.apress.com>提供了更多信息以及可供下载的示例代码。

## 目标读者

本书主要面向两类读者。第一类读者是懂一点JavaScript并且想要加深对它的认识的Web开发人员或前端工程师，尤其是那些想要增进其对于JavaScript的面向对象能力的理解，学习如何提高其代码的模块化程度、可维护性和效率的人，他们可以从书中学到用JavaScript进行面向对象程序

① 中文版名为《设计模式——可复用面向对象软件的基础》（机械工业出版社，2004）。——译者注

设计的基本知识，还能学到各种具体的设计模式，懂得应该在什么场合使用这些设计模式，以及如何实现它们。这类读者已经比较熟悉JavaScript的基本语法，他们会更多地关注那些关于如何按特定设计模式重构现有代码的部分，以及对于每种模式的适用场合的说明。

第二类读者是一些主要使用Java和C++等服务器端编程语言的程序员，相对而言，他们对JavaScript比较陌生，但又希望能把自己在设计模式和面向对象程序设计方面的知识应用到客户端程序设计中。他们可以从本书中学到如何在JavaScript中实现接口、继承和封装等常见的面向对象特性。这类读者会发现书中的代码尤其有用，因为他们可能不太熟悉JavaScript和其他面向对象语言在语法方面的差别。也许他们对各种具体的设计模式已经比较熟悉，所以本书讲述面向对象技术在JavaScript中的实现方法的第一部分对他们可能更有意义。

那些对JavaScript和面向对象程序设计都不太熟悉的读者可能很难看懂某些示例中的代码。这并不是一本入门级的书，它假定读者已经具备一定的程序设计知识。尽管如此，我们还是尽可能地使自己的讲解做到深入浅出，让不同层次的读者都易于理解。

## 本书结构

本书分为两部分。第一部分讲述JavaScript面向对象程序设计的基础知识，其中的各章应该按顺序阅读。每一章都建立在前面一章的基础上，并且假定你已经读过之前的各章。读者最好先通读这一部分的所有章节，因为第二部分的各章都要用到第一部分讲述的技术，而且有时不会再加以说明。

第二部分讲述具体的设计模式及其在JavaScript中的实际应用，其中的各章可以按任意顺序阅读。有些章引用了其他章的内容，被引用的内容既有第一部分的，也有第二部分的，我们都给出了引文的章号。

## 第一部分

**第1章（富有表现力的JavaScript）**揭示了JavaScript语言富有表现力的特点。从中你可以体会到，这种语言允许你用各种各样的编程风格来完成同样的任务，还允许你在面向对象编程的过程中借用函数式编程的概念来丰富其实现方式。这一章解释了究竟为什么应该使用设计模式，以及它们在JavaScript程序设计中的运用是如何使代码更高效、更易于处理的。

**第2章（接口）**分析了其他面向对象语言实现接口的方式，并用JavaScript对它们在这方面的最佳特性进行了模仿。文中探讨了接口检查的各种可行方式，并给出了一个可用来检查对象是否具有必要方法的可重用的类。

**第3章（封装和信息隐藏）**探讨了在JavaScript中创建对象的各种不同方式，以及每种方式中用以创建公用（public）、私用（private）和受护（protected）方法<sup>①</sup>的可行技术。文中还对经过复杂封装的对象的适用场合进行了讨论。

<sup>①</sup> JavaScript并不像许多面向对象语言那样提供public、private和protected关键字及相关的特性。第3章中用一些技术模仿了其他语言对public成员和private成员的实现，但没有模仿protected成员的实现。——译者注

**第4章（继承）**讲述了在JavaScript中用以创建子类的各种技术。其中既讲了类式继承，也讲了原型式继承，还说明了它们各自的适用场合。文中还讲述了掺元类（mixin class）<sup>①</sup>，以及如何用它替代多亲继承（multiple inheritance）<sup>②</sup>。

**第5章（单体模式）**讨论了JavaScript中的单体模式、命名空间、代码组织，以及可以用来根据运行时环境动态定义方法的分支技术（branching）。其中还谈到了工厂模式等可以受益于单体的模式。

**第6章（方法的链式调用）**考察了JavaScript对方法进行链式调用的能力，以及这种做法为什么能得到更清晰、简练的代码。文中用这种技术创建了一个小小的JavaScript库，并把其中的方法和没有利用链式调用技术实现的对应方法进行了比较。

## 第二部分

**第7章（工厂模式）**讨论工厂模式。这种模式有助于消除那些彼此实例化对方的类之间的耦合，并改而用一个方法来确定要实例化哪个类。文中既讨论了另外用一个类（通常是一个单体）来生成实例的简单工厂模式，也讨论了用子类来确定一个成员对象应该是哪种具体类实例的较复杂的工厂模式。

**第8章（桥接模式）**讨论了一种既能把两个对象连接在一起，又能避免二者间的强耦合的方法。桥接元素把两个对象连接起来，同时又允许它们独立变化。文中演示了如何用桥接元素把函数松散地绑定到事件。这一章还创建了一个异步连接队列，用以示范桥接模式在保持代码的简洁性方面的作用。

**第9章（组合模式）**讨论了一种非常适合于创建Web上的动态用户界面的设计模式：组合模式。文中演示了如何使用这种模式来达到只用一条命令即可在许多对象上激发复杂或递归性的行为的目的，以及如何用它把一系列对象组织为复杂的层次体系（hierarchy）。文中还逐一说明了实现组合模式所必经的一系列步骤，并讨论了该模式的适用场合。

**第10章（门面模式）**讨论了一种用来为对象创建一个更完善的接口的方法。门面模式可以用来把现有接口转换为一个更便于使用的接口。文中解释了为什么大多数JavaScript库都是为这种语言在具体浏览器中的实现提供的一个门面。这一章也显示了如何用这种模式创建便利方法和事件工具库。

**第11章（适配器模式）**讨论了一种可以让现有接口契合实际需要的模式。适配器也称包装器（wrapper），用来把不匹配的接口替换为一个可用于现有系统中的接口。文中探讨了如何用适配器弥合JavaScript库间的差异并简化从一种库过渡到另一种库的过程。其中考察了一个电子邮件API，并创建了一个有助于升级到新版本的适配器。

**第12章（装饰者模式）**讨论了一种可以为对象添加特性而又不必创建新的子类的方法。装饰者模式用于把对象透明地包装到另一种具有相同接口的对象中。这一章考察了装饰者的结构以及

① 也译为“混入类”。——译者注

② 也译“多重继承”，但感觉不够准确。

如何将其与工厂模式结合使用以自动创建内嵌对象。我们还创建了一个性能分析器，以示范如何用装饰者模式动态实现接口。

**第13章（享元模式）**讨论了另一种用于优化目的的模式：享元模式。文中示范了如何通过把大批独立对象转变为少量共享对象，从而大幅削减实现应用软件所需的对象数目。这一章还创建了一个Web日历和一个可重用的工具提示类，以示范如何按享元模式对类进行重构。

**第14章（代理模式）**讨论了可用于控制对对象的访问的代理模式。文中显示了如何为本体（real subject）<sup>①</sup>创建一个代理对象以作为其替身，并使其能被远程访问，还探讨了代理模式的种种用法，其中包括推迟对其创建需要耗用大量计算资源的类的实例化。这一章还创建了一个可用来推迟任何类的加载的通用类。

**第15章（观察者模式）**讨论了一种对对象的状态进行观察，并且当它发生变化时能得到通知的方法。观察者模式也称发布者-订阅者模式（publisher-subscriber pattern），用于让对象对事件进行监听以便对其作出响应。文中以报业为例说明了观察者模式的各种运作方式，还讨论了在使用一个动画库时可以订阅的各种事件。

**第16章（命令模式）**讨论了一种对方法调用进行封装的方式。借助命令模式，可以对方法调用进行参数化和传递，然后在需要的时候再加以执行。文中说明了这种模式的各种应用场合，其中包括创建用户界面——尤其是那种需要不受限制的取消操作的用户界面。这一章还讨论了命令模式的结构，并提供了几个示范其在JavaScript中的用法的实际例子。

**第17章（职责链模式）**讨论了用来消除请求的发送者和接收者之间的耦合的职责链模式。文中说明了在JavaScript中如何用这种模式处理事件的捕获和冒泡，如何创建弱耦合模块和优化事件监听器的绑定。

## 预备知识

为了使书中的代码示例更加清晰、目标更加明确，我们使用了一些便利函数来执行安装事件监听器、派生子类、处理Cookie、引用HTML元素等任务。我们没有使用YUI或jQuery等库所提供的类似功能，其目的在于避免让我们的代码依赖于其他库，以便读者可以将其与自己喜欢的任何库结合使用。各种主流JavaScript库都有与我们使用的函数相对应的函数。这些便利函数的完整代码可以从本书的网站<http://jsdesignpatterns.com>和Apress出版社的网站<http://www.apress.com>下载。下面是这些函数的简要说明。

- `$(id)`，根据ID值获取HTML元素的引用。其参数可以是字符串，也可以是字符串的数组。
- `addEvent(obj, type, func)`，把函数func作为元素obj的事件监听器。type表示该函数所要监听的事件。
- `addLoadEvent(func)`，将函数func关联到window对象的load事件。
- `getElementsByClass(searchClass, node, tag)`，获取所有class属性值为searchClass的元素的引用。node和tag这两个参数可有可无，它们可以用来缩小搜索范围。函数的返回值

<sup>①</sup> 也译为“实体”或“主体”。——译者注

是一个数组。

- `insertAfter(parent, node, referenceNode)`, 插入元素`node`, 其父元素为`parent`, 其位置在`referenceNode`之后。
- `getCookie(name)`, 获取与名为`name`的Cookie相关联的字符串。
- `setCookie(name, value, expires, path, domain, secure)`, 把与名为`name`的Cookie相关联的字符串设置为`value`。除`name`和`value`外的其他参数均可有可无。
- `deleteCookie(name)`, 将名为`name`的Cookie的过期时间设置到过去。<sup>①</sup>
- `clone(object)`, 创建`object`的一个副本。用于原型式继承。见第4章。
- `extend(subClass, superClass)`, 执行一些必要的工作, 使`subClass`成为`superClass`的子类。见第4章。
- `augment(receivingClass, givingClass)`, 将`givingClass`中的方法输入`receivingClass`中。见第4章。

## 下载代码

本书的附属网站<http://jsdesignpatterns.com>和Apress出版社的网站<http://www.apress.com>都有zip文件格式的示例代码供下载。<sup>②</sup>

## 联系作者

你可以通过[dustin@jsdesignpatterns.com](mailto:dustin@jsdesignpatterns.com)和[ross@jsdesignpatterns.com](mailto:ross@jsdesignpatterns.com)与作者联系。

## 致谢

感谢认真负责的技术审稿人Simon Willison。要不是他, 本书的准确性、实用性和趣味性都得大打折扣。他不厌其烦地为每一章提供了令人惊喜的反馈。

感谢那些在百忙之中抽空阅读本书草稿并提供意见和更正的同事和合作伙伴。Dave Marr和Ernest Delgado更是在排除打字错误、技术错误和病句方面发挥了重要作用。同时, 也要感谢Lindsey Simon和Robert Otani, 他们层出不穷的JavaScript幽默为我们提供了不少帮助。

感谢我们的朋友和家人。面对我们没完没了的有关写作和晦涩的技术细节的唠叨, 他们总是很有耐心。他们的支持是我们前进的动力。

最后, 真心感谢Apress出版社负责本书的工作人员Chris Mills、Tom Welsh、Dominic Shakeshaft、Richard Dal Porto, 还有Jennifer Whipple, 他们的耐心、毅力和善解人意尤其值得赞赏并且令人难忘。

<sup>①</sup> 即删除这个Cookie。——译者注

<sup>②</sup> 示例代码也可以在图灵网站([www.turingbook.com](http://www.turingbook.com))本书网页免费注册下载。——编者注

# 目 录

## 第一部分 面向对象的 JavaScript

### 第1章 富有表现力的 JavaScript ..... 2

1.1	JavaScript 的灵活性 ..... 2
1.2	弱类型语言 ..... 5
1.3	函数是一等对象 ..... 5
1.4	对象的易变性 ..... 6
1.5	继承 ..... 8
1.6	JavaScript 中的设计模式 ..... 8
1.7	小结 ..... 8

### 第2章 接口 ..... 10

2.1	什么是接口 ..... 10
2.1.1	接口之利 ..... 10
2.1.2	接口之弊 ..... 11
2.2	其他面向对象语言处理接口的方式 ..... 11
2.3	在 JavaScript 中模仿接口 ..... 13
2.3.1	用注释描述接口 ..... 13
2.3.2	用属性检查模仿接口 ..... 14
2.3.3	用鸭式辨型模仿接口 ..... 16
2.4	本书采用的接口实现方法 ..... 17
2.5	Interface 类 ..... 17
2.5.1	Interface 类的使用场合 ..... 18
2.5.2	Interface 类的用法 ..... 19
2.5.3	示例：使用 Interface 类 ..... 19
2.6	依赖于接口的设计模式 ..... 21
2.7	小结 ..... 22

### 第3章 封装和信息隐藏 ..... 23

3.1	信息隐藏原则 ..... 23
3.1.1	封装与信息隐藏 ..... 23
3.1.2	接口扮演的角色 ..... 24
3.2	创建对象的基本模式 ..... 24

### 3.2.1 门户大开型对象 ..... 25

3.2.2	用命名规范区别私用成员 ..... 28
3.2.3	作用域、嵌套函数和闭包 ..... 29
3.2.4	用闭包实现私用成员 ..... 30

### 3.3 更多高级对象创建模式 ..... 32

3.3.1	静态方法和属性 ..... 32
3.3.2	常量 ..... 35
3.3.3	单体和对象工厂 ..... 36
3.4	封装之利 ..... 36
3.5	封装之弊 ..... 37
3.6	小结 ..... 37

### 第4章 继承 ..... 39

4.1	为什么需要继承 ..... 39
4.2	类式继承 ..... 39
4.2.1	原型链 ..... 40
4.2.2	extend 函数 ..... 41
4.3	原型式继承 ..... 43
4.3.1	对继承而来的成员的读和写 的不对等性 ..... 44
4.3.2	clone 函数 ..... 46

4.4	类式继承和原型式继承的对比 ..... 47
4.5	继承与封装 ..... 47
4.6	掺元类 ..... 48
4.7	示例：就地编辑 ..... 49
4.7.1	类式继承解决方案 ..... 50
4.7.2	原型式继承解决方案 ..... 53
4.7.3	掺元类解决方案 ..... 56

### 4.8 继承的适用场合 ..... 59

### 4.9 小结 ..... 60

### 第5章 单体模式 ..... 61

5.1	单体的基本结构 ..... 61
-----	------------------

5.2 划分命名空间	62	8.3 用桥接模式联结多个类	103
5.3 用作特定网页专用代码的包装器的单体	64	8.4 示例：构建 XHR 连接队列	103
5.4 拥有私用成员的单体	66	8.4.1 添加核心工具	103
5.4.1 使用下划线表示法	66	8.4.2 添加观察者系统	105
5.4.2 使用闭包	67	8.4.3 开发队列的基本框架	106
5.4.3 两种技术的比较	69	8.4.4 实现队列	108
5.5 惰性实例化	70	8.4.5 哪些地方用了桥接模式	112
5.6 分支	73	8.5 桥接模式的适用场合	113
5.7 示例：用分支技术创建 XHR 对象	74	8.6 桥接模式之利	113
5.8 单体模式的适用场合	76	8.7 桥接模式之弊	113
5.9 单体模式之利	77	8.8 小结	114
5.10 单体模式之弊	77	<b>第 9 章 组合模式</b>	115
5.11 小结	77	9.1 组合对象的结构	115
<b>第 6 章 方法的链式调用</b>	78	9.2 使用组合模式	115
6.1 调用链的结构	78	9.3 示例：表单验证	116
6.2 设计一个支持方法链式调用的 JavaScript 库	81	9.3.1 汇合起来	121
6.3 使用回调从支持链式调用的方法 获取数据	83	9.3.2 向 FormItem 添加操作	121
6.4 小结	84	9.3.3 向层次体系中添加类	121
<b>第二部分 设计模式</b>		9.3.4 添加更多操作	123
<b>第 7 章 工厂模式</b>	86	9.4 示例：图片库	124
7.1 简单工厂	86	9.5 组合模式之利	127
7.2 工厂模式	89	9.6 组合模式之弊	127
7.3 工厂模式的适用场合	91	9.7 小结	127
7.3.1 动态实现	91	<b>第 10 章 门面模式</b>	128
7.3.2 节省设置开销	91	10.1 一些你可能已经知道的门面元素	128
7.3.3 用许多小型对象组成一个大对象	92	10.2 JavaScript 库的门面性质	129
7.4 示例：XHR 工厂	92	10.3 用作便利方法的门面元素	129
7.4.1 专用型连接对象	94	10.4 示例：设置 HTML 元素的样式	131
7.4.2 在运行时选择连接对象	95	10.5 示例：设计一个事件工具	132
7.5 示例：RSS 阅读器	97	10.6 实现门面模式的一般步骤	133
7.6 工厂模式之利	100	10.7 门面模式的适用场合	134
7.7 工厂模式之弊	100	10.8 门面模式之利	134
7.8 小结	100	10.9 门面模式之弊	134
<b>第 8 章 桥接模式</b>	101	10.10 小结	135
8.1 示例：事件监听器	101	<b>第 11 章 适配器模式</b>	136
8.2 桥接模式的其他例子	102	11.1 适配器的特点	136

11.4	示例：适配电子邮件 API	139	13.7	享元模式的适用场合	177
11.4.1	用适配器包装 Web 邮件 API	143	13.8	实现享元模式的一般步骤	177
11.4.2	从 fooMail 转向 dedMail	144	13.9	享元模式之利	178
11.5	适配器模式的适用场合	144	13.10	享元模式之弊	179
11.6	适配器模式之利	145	13.11	小结	179
11.7	适配器模式之弊	145			
11.8	小结	145			
<b>第 12 章 装饰者模式</b>		<b>146</b>	<b>第 14 章 代理模式</b>		<b>180</b>
12.1	装饰者的结构	146	14.1	代理的结构	180
12.1.1	接口在装饰者模式中的角色	149	14.1.1	代理如何控制对本体的访问	180
12.1.2	装饰者模式与组合模式的 比较	150	14.1.2	虚拟代理、远程代理和保护 代理	183
12.2	装饰者修改其组件的方式	150	14.1.3	代理模式与装饰者模式的 比较	184
12.2.1	在方法之后添加行为	150	14.2	代理模式的适用场合	184
12.2.2	在方法之前添加行为	151	14.3	示例：网页统计	184
12.2.3	替换方法	152	14.4	包装 Web 服务的通用模式	187
12.2.4	添加新方法	153	14.5	示例：目录查找	189
12.3	工厂的角色	156	14.6	创建虚拟代理的通用模式	192
12.4	函数装饰者	158	14.7	代理模式之利	195
12.5	装饰者模式的适用场合	159	14.8	代理模式之弊	195
12.6	示例：方法性能分析器	159	14.9	小结	196
12.7	装饰者模式之利	162			
12.8	装饰者模式之弊	163	<b>第 15 章 观察者模式</b>		<b>197</b>
12.9	小结	163	15.1	示例：报纸的投送	197
<b>第 13 章 享元模式</b>		<b>164</b>	15.1.1	推与拉的比较	197
13.1	享元的结构	164	15.1.2	模式的实践	198
13.2	示例：汽车登记	164	15.2	构建观察者 API	200
13.2.1	内在状态和外在状态	165	15.2.1	投送方法	200
13.2.2	用工厂进行实例化	166	15.2.2	订阅方法	201
13.2.3	封装在管理器中的外在状态	167	15.2.3	退订方法	202
13.3	管理外在状态	168	15.3	现实生活中的观察者	202
13.4	示例：Web 日历	168	15.4	示例：动画	202
13.4.1	把日期对象转化为享元	170	15.5	事件监听器也是观察者	203
13.4.2	外在数据保存在哪里	171	15.6	观察者模式的适用场合	204
13.5	示例：工具提示对象	171	15.7	观察者模式之利	205
13.5.1	未经优化的 Tooltip 类	171	15.8	观察者模式之弊	205
13.5.2	作为享元的 Tooltip	173	15.9	小结	205
13.6	保存实例供以后重用	175			
			<b>第 16 章 命令模式</b>		<b>206</b>
			16.1	命令的结构	206
			16.1.1	用闭包创建命令对象	207

16.1.2 客户、调用者和接收者	208	第 17 章 职责链模式	225
16.1.3 在命令模式中使用接口	208	17.1 职责链的结构	225
16.2 命令对象的类型	209	17.2 传递请求	230
16.3 示例：菜单项	210	17.3 在现有层次体系中实现职责链	233
16.3.1 菜单组合对象	211	17.4 事件委托	234
16.3.2 命令类	213	17.5 职责链模式的适用场合	234
16.3.3 汇合起来	214	17.6 图片库的进一步讨论	235
16.3.4 添加更多菜单项	215	17.6.1 用职责链提高组合对象 的效率	236
16.4 示例：取消操作和命今日志	216	17.6.2 为图片添加标签	237
16.4.1 使用命今日志实现不可逆 操作的取消	220	17.7 职责链模式之利	240
16.4.2 用于崩溃恢复的命今日志	222	17.8 职责链模式之弊	240
16.5 命令模式的适用场合	222	17.9 小结	241
16.6 命令模式之利	223	索引	242
16.7 命令模式之弊	223		
16.8 小结	223		

## 第一部分

# 面向对象的 JavaScript

### 本部分内容

- 第 1 章 富有表现力的 JavaScript
- 第 2 章 接口
- 第 3 章 封装和信息隐藏
- 第 4 章 继承
- 第 5 章 单体模式
- 第 6 章 方法的链式调用

# 富有表现力的JavaScript

JavaScript是现在最流行、应用最广泛的语言之一。由于所有现代浏览器都嵌入了JavaScript解释器，所以在大多数地方都能见到其身影。作为一种语言，它在我们的日常生活中起着非常重要的作用，支持着我们访问的网站，帮助Web呈现出多姿多彩的界面。

那为什么有些人还把它看作一种玩具式的语言，认为它不值得职业程序员关注呢？我们认为其原因在于，人们没有认清这种语言的全部能力及其在当今的编程世界中的独特性。JavaScript是一种极富表现力的语言，它具有一些C家族语言所罕见的特性。

本章将探讨一些令JavaScript如此富有表现力的特性。从中你可以体会到，这种语言允许你用各种方式完成同样的任务，还允许你在面向对象编程的过程中借用函数式编程中的概念来丰富其实现方式。本章解释了究竟为什么应该使用设计模式，以及它们在JavaScript程序设计的运用是如何使代码更高效、更易于处理的。

## 1.1 JavaScript 的灵活性

JavaScript最强大的特性是其灵活性。作为JavaScript程序员，只要你愿意，可以把程序写得很简单，也可以写得很复杂。这种语言也支持多种不同的编程风格。你既可以采用函数式编程风格，也可以采用更复杂一点的面向对象编程风格。即使你根本不懂函数式编程或面向对象编程，也能写出较为复杂的程序。使用这种语言，哪怕只采用编写一个个简单的函数的方式，你也能高效地完成任务。这可能是某些人把JavaScript视同玩具的原因之一，但我们却认为这是一个优点。程序员只要使用这种语言的一个很小的、易于学习的子集就能完成一些有用的任务。这也意味着当你成长为一个更高级的程序员时，JavaScript在你手中的威力也在增长。

JavaScript允许你模仿其他语言的编程模式和惯用法。它也形成了自己的一些编程模式和惯用法。那些较为传统的服务器端编程语言具有的面向对象特性，JavaScript都有。

先来看一个用不同方法完成同样的任务的例子：启动和停止一个动画。如果你看不懂这些例子，别担心。我们在此使用的所有模式和技术都会在本书后面进行讲解。目前你可以把这一部分看作一个演示在JavaScript中用不同方法完成同一任务的实际例子。

如果你习惯于过程式的程序设计，那么可以这样做：

```
/* Start and stop animations using functions. */

function startAnimation() {
    ...
}

function stopAnimation() {
    ...
}
```

这种做法很简单，但你无法创建可以保存状态并且具有一些仅对其内部状态进行操作的方法的动画对象。下面的代码定义了一个类，你可以用它创建这种对象：

```
/* Anim class. */

var Anim = function() {
    ...
};

Anim.prototype.start = function() {
    ...
};

Anim.prototype.stop = function() {
    ...
};

/* Usage. */

var myAnim = new Anim();
myAnim.start();
...
myAnim.stop();
```

上述代码定义了一个名为Anim的类，并把两个方法赋给该类的prototype属性。第3章将详细讲述这种技术。如果你更喜欢把类的定义封装在一条声明中，则可改用下面的代码：

```
/* Anim class, with a slightly different syntax for declaring methods. */

var Anim = function() {
    ...
};

Anim.prototype = {
    start: function() {
        ...
    },
    stop: function() {
        ...
    }
};
```

这在传统的面向对象程序员看来可能更眼熟一点，他们习惯于看到类的方法声明内嵌在类的

声明之中。要是你以前用过这样的编程风格，可能想尝试一下下面的示例。同样，如果代码中有些地方你看不懂，不必为此而焦虑：

```
/* Add a method to the Function object that can be used to declare methods. */

Function.prototype.method = function(name, fn) {
    this.prototype[name] = fn;
};

/* Anim class, with methods created using a convenience method. */

var Anim = function() {
    ...
};

Anim.method('start', function() {
    ...
});

Anim.method('stop', function() {
    ...
});
```

Function.prototype.method用于为类添加新方法。它有两个参数。第一个是字符串，表示新方法的名称；第二个是用作新方法的函数。

你可以进一步修改Function.prototype.method，使其可被链式调用。这只需要让它返回this值即可（方法的链式调用技术将在第6章中讲述）：

```
/* This version allows the calls to be chained. */

Function.prototype.method = function(name, fn) {
    this.prototype[name] = fn;
    return this;
};

/* Anim class, with methods created using a convenience method and chaining. */

var Anim = function() {
    ...
};

Anim.
method('start', function() {
    ...
});
method('stop', function() {
    ...
});
```

你已经见识了完成同一项任务的5种不同方法，它们的风格略有差异。基于自己的编程背景，你可能觉得其中的某种方法比别的方法更为合意。这是件好事：JavaScript允许你用最适合于手头项目的编程风格进行工作。不同的风格在代码篇幅、编码效率和执行性能方面各有其特点。本书

的第一部分将讨论所有这些风格。

## 1.2 弱类型语言

在JavaScript中，定义变量时不必声明其类型。但这并不意味着变量没有类型。一个变量可以属于几种类型之一，这取决于其包含的数据。JavaScript中有3种原始类型：布尔型、数值型和字符串类型（不区分整数和浮点数是JavaScript与大多数其他主流语言的一个不同之处）。此外，还有对象类型和包含可执行代码的函数类型，前者是一种复合数据类型（数组是一种特殊的对象，它包含着一批值的有序集合）。最后，还有空类型（null）和未定义类型（undefined）这两种数据类型。原始数据类型按值传送，而其他数据类型则按引用传送。如果不了解这一点的话，你很可能会碰到一些意想不到的问题。

与其他弱类型语言一样，JavaScript中的变量可以根据所赋的值改变类型。原始类型之间也可以进行类型转换。`toString`方法可以把数值或布尔值转变为字符串。`parseFloat`和`parseInt`函数可以把字符串转变为数值。双重“非”操作可以把字符串或数值转变为布尔值：

```
var bool = !!num;
```

弱类型的变量带来了极大的灵活性。因为JavaScript会根据需要进行类型转换，所以一般说来，你不用为类型错误操心。

## 1.3 函数是一等对象

在JavaScript中，函数是一等对象。它们可以存储在变量中，可以作为参数传给其他函数，可以作为返回值从其他函数传出，还可以在运行时进行构造。在与函数打交道时，这些特性带来了极大的灵活性和极强的表达能力。在阅读本书时你会体会到，这正是用以构建传统的面向对象框架的基础。

可以用`function() { ... }`这样的语法创建匿名函数。它们没有函数名，但可以被赋给变量。下面是一个匿名函数的示例：

```
/* An anonymous function, executed immediately. */
```

```
(function() {
  var foo = 10;
  var bar = 2;
  alert(foo * bar);
})();
```

这个函数在定义之后便立即执行，甚至不用赋给一个变量。出现在函数声明之后的一对括号立即对函数进行了调用。括号中空无一物，但也并不是非得如此：

```
/* An anonymous function with arguments. */
```

```
(function(foo, bar) {
  alert(foo * bar);
})(10, 2);
```

这个匿名函数与前一个等价，只不过变量没有在函数内部用var声明，而是作为参数从外部传入而已。这个函数也可以返回一个值。这个返回值可以被赋给一个变量：

```
/* An anonymous function that returns a value. */

var baz = (function(foo, bar) {
    return foo * bar;
})(10, 2);

// baz will equal 20.
```

匿名函数最有趣的用途是用来创建闭包。闭包（closure）是一个受到保护的变量空间，由内嵌函数生成。JavaScript具有函数级的作用域。这意味着定义在函数内部的变量在函数外部不能被访问。JavaScript的作用域又是词法性质的（lexically scoped）。这意味着函数运行在定义它的作用域中，而不是在调用它的作用域中。把这两个因素结合起来，就能通过把变量包裹在匿名函数中而对其加以保护。你可以这样创建类的私用（private）变量：

```
/* An anonymous function used as a closure. */

var baz;

(function() {
    var foo = 10;
    var bar = 2;
    baz = function() {
        return foo * bar;
    };
})();

baz(); // baz can access foo and bar, even though it is executed outside of the
       // anonymous function.
```

7 变量foo和bar定义在匿名函数中。因为函数baz定义在这个闭包中，所以它能访问这两个变量，即使是在该闭包执行结束后。这是一个复杂的话题，本书中会多次涉及。在第3章讨论封装的时候将对这种技术详加讲解。

## 1.4 对象的易变性

在JavaScript中，一切都是对象（除了那三种原始数据类型。即便是这些类型，在必要的时候也会被自动包装为对象），而且所有对象都是易变的（mutable）。这意味着你能使用一些在大多数别的语言中不允许的技术，例如为函数添加属性：

```
function displayError(message) {
    displayError.numTimesExecuted++;
    alert(message);
}
displayError.numTimesExecuted = 0;
```

这也意味着你可以对先前定义的类和实例化的对象进行修改：

```
/* Class Person. */

function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype = {
    getName: function() {
        return this.name;
    },
    getAge: function() {
        return this.age;
    }
}

/* Instantiate the class. */

var alice = new Person('Alice', 93);
var bill = new Person('Bill', 30);

/* Modify the class. */

Person.prototype.getGreeting = function() {
    return 'Hi ' + this.getName() + '!';
};

/* Modify a specific instance. */

alice.displayGreeting = function() {
    alert(this.getGreeting());
}
```

在这个例子中，类的getGreeting方法是在已经创建了类的两个实例之后才添加的，但这两个实例仍然能获得这个方法，其原因在于prototype对象的工作机制。对象alice还得到了displayGreeting方法，而别的实例却没有。

与对象的易变性相关的还有内省（introspection）的概念。你可以在运行时检查对象所具有的属性和方法，还可以使用这种信息动态实例化类和执行其方法（这种技术称为反射（reflection）），甚至不需要在开发时知道它们的名称。这些技术在动态脚本编程中发挥着重要作用，而静态语言（例如C++）则缺乏这样的特性。

本书中大多数用来模仿传统的面向对象特性的技术都依赖于对象的易变性和反射。如果你习惯使用C++或Java这类语言，可能会觉得这很奇怪，因为在那些语言中，不能对已经实例化的对象进行扩展，也不能对已经定义好的类进行修改。而在JavaScript中，任何东西都可以在运行时修改。这是一个强有力的工具，许多在别的语言中无法办到的事都能借助于它而办到。当然，这也有其不利之处。你可以定义一个具有一套方法的类，却不能肯定这些方法在以后总是完好如初。这是JavaScript中很少进行类型检查的原因之一。这个问题将在第2章讲述鸭式辨型（duck typing）和接口检查时进行探讨。

## 1.5 继承

继承在JavaScript中不像在别的面向对象语言中那样简单。JavaScript使用的是基于对象的（原型式（prototypal））继承，它可以用来模仿基于类的（类式（classical））继承。这两种范型本书都会讲述，编写代码时用哪一种都行，根据手头任务的实际情况，有时其中的某种会更适合一些。它们在性能上也有不同的表现，这也是在进行选择时需要考虑的重要因素。这个复杂的话题将在第4章中进行探讨。

## 1.6 JavaScript 中的设计模式

1995年，GoF合作出版了一本名为《设计模式》的书。这本书整理记录了对象间相互作用的各种方式，并针对不同类型的对象创造了一套通用术语。用以创建这些不同类型的对象的套路被称为设计模式（design pattern）。出于通用性的考虑，书中使用了一种在一定程度上独立于语言的方式来描述这些模式。本书就是专门讨论这些模式在JavaScript中的应用的。  
9

JavaScript强大的表现力赋予了程序员在运用设计模式编写代码时极大的创造性。在JavaScript中使用设计模式主要有如下3原因。

(1) 可维护性。设计模式有助于降低模块间的耦合程度。这使对代码进行重构和换用不同的模块变得更容易，也使程序员在大型团队中的工作以及与其他程序员的合作变得更容易。

(2) 沟通。设计模式为处理不同类型的对象提供了一套通用的术语。程序员因此可以更简明地描述自己的系统的工作方式。你不用进行冗长的说明，往往这样一句话就足够了：“它使用了工厂模式”。每个模式都有自己的名称，这意味着你可以在较高层面上对其进行讨论，而不必涉足过多的细节。

(3) 性能。本书讲述的某些模式是起优化作用的模式。它们可以大幅提高程序的运行速度，并减少需要传送到客户端的代码量。这方面最重要的例子是享元模式（第13章）和代理模式（第14章）。

你也可能出于如下两个理由而不使用设计模式。

(1) 复杂性。获得可维护性往往要付出代价，那就是代码可能会变得更加复杂、更难被程序设计新手理解。

(2) 性能。尽管某些模式能提升性能，但多数模式对代码的性能都有所拖累。这种拖累可能微不足道，也可能完全不能接受，这取决于项目的具体需求。

实现设计模式比较容易，而懂得应该在什么时候使用什么模式则较为困难。未搞懂设计模式的用途就盲目套用，是一种不安全的做法。你应该尽量保证所选用的模式就是最恰当的那种，并且不要过度牺牲性能。

## 1.7 小结

JavaScript的丰富表现力是其力量之源。即使这种语言缺少一些有用的内置特性，拜其灵活性所赐，你也能自己加入这些特性。完成一项任务可以有多种方式，你可以根据自己的技术背景和

喜好选择编写代码的方式。

JavaScript是弱类型语言。程序员在定义变量时并不指定其类型。函数是一等对象，并且可以动态创建，因此你可以创建闭包。所有对象和类都是易变的，可以在运行时修改。可供使用的继承范型有两种，即原型式继承和类式继承，它们各有其优缺点。

JavaScript中的设计模式颇有助益，但其不当应用也会产生负面效果。在JavaScript这类轻灵的语言中，过度复杂的架构会很快把应用程序拖入泥沼。你使用的编程风格和选择的设计模式应该与所要完成的具体工作相称。

**接**口是面向对象JavaScript程序员的工具箱中最有用的工具之一。GoF在《设计模式》一书中提出的可重用面向对象设计的第一条原则中就说道：“针对接口而不是实现编程”，这个概念的基本性由此可见一斑。

问题在于，JavaScript中没有内置的创建或实现接口的方法。它也没有内置的方法可以用于判断一个对象是否实现了与另一个对象相同的一套方法，这使对象很难互换使用。好在JavaScript有着出色的灵活性，因此添加这些特性并非难事。

本章将考察其他面向对象的语言中实现接口的方法，并对它们在这方面最突出的特性进行模仿。我们先讨论在JavaScript中实现接口的各种方法，最后设计出一个可重用的类，用于检查对象是否具有必要的方法。

## 2.1 什么是接口

接口提供了一种用以说明一个对象应该具有哪些方法的手段。尽管它可以表明（或至少是暗示）这些方法的语义，但它并不规定这些方法应该如何实现。例如，如果一个接口包含有一个名为`setName`的方法，那么你有理由认为这个方法的实现应该具有一个字符串参数，并且会把这个参数赋给一个`name`变量。

有了这个工具，你就能按对象提供的特性对它们进行分组。例如，即使一批对象彼此存在着极大的差异，只要它们都实现了`Comparable`接口，那么在`object.compare(anotherObject)`方法中就可以互换使用这些对象。你还可以使用接口开发不同的类之间的共同性。如果把原本要求以一个特定的类为参数的函数改为要求以一个特定的接口为参数的函数，那么任何实现了该接口的对象都可以作为参数传递给它。这样一来，彼此不相关的对象也可以被同等对待。

### 2.1.1 接口之利

在面向对象的JavaScript中，接口有些什么作用呢？既定的一批接口具有自我描述性，并能促进代码的重用。接口可以告诉程序员一个类实现了哪些方法，从而帮助其使用这个类。如果你熟悉一个特定的接口，那么就已经知道如何使用任何实现了它的类，从而更有可能重用现有的类。

接口还有助于稳定不同的类之间的通信方式。如果事先知道了接口，你就能减少在集成两个对象的过程中出现的问题。借助于它，你可以事先就说明你希望一个类具有哪些特性和操作。一

个程序员可以针对所需要的类定义一个接口，并把它转交给另一个程序员。第二个程序员可以随心所欲地编写自己的代码，只要他定义的类实现了那个接口就行。这在大型项目中尤其有用。

测试和调试因此也能变得更轻松。在JavaScript这种弱类型语言中，类型不匹配错误很难跟踪。使用接口可以让这种错误的查找变得更容易一点，因为此时如果一个对象不像所要求的类型，或者没有实现必要的方法，那么你会得到包含有用信息的明确的错误提示。这样一来，逻辑错误可以被限制在方法自身，而不是在对象的构成之中。接口还能让代码变得更稳固，因为对接口的任何改变在所有实现它的类中都必须体现出来。如果接口添加了一个操作，而某个实现它的类并没有相应地添加这个操作，那么你肯定会立即见到一个错误。

### 2.1.2 接口之弊

接口并非没有缺点。JavaScript是一种具有极强表现力的语言，这主要得益于其弱类型的特点。而接口的使用则在一定程度上强化了类型的作用。这降低了语言的灵活性。

JavaScript并没有提供对接口的内置支持，而试图模仿其他语言内置的功能总会有一些风险。JavaScript中没有Interface这个关键词，因此，不管你用什么方法实现接口，它总是与C++和Java这些语言中所用的方法大相径庭，这加大了初涉JavaScript时所遇到的困难。

JavaScript中任何实现接口的方法都会对性能造成一些影响，在某种程度上这得归咎于额外的方法调用的开销。我们的实现方法中使用了两个for循环来遍历所需要的每一个接口中的每一个方法。对于大型接口和需要实现许多不同接口的对象，这种检查可能要花点时间，从而对性能造成负面影响。如果你在乎这个问题，那么可以在开发完成之后剔除这种代码，或者将其执行与一个调试标志关联起来，这样在运营环境中它就不会执行。但要注意不要过早进行优化处理。Firebug这类性能分析器可以帮助你判断是否真有必要剔除接口代码。

接口使用中的最大问题在于，无法强迫其他程序员遵守你定义的接口。在其他语言中，接口的概念是内置的，如果某人定义了实现一个接口的类，那么编译器会确保该类的确实现了这个接口。而在JavaScript中则必须用手工的办法保证某个类实现了一个接口。编码规范和辅助类可以提供一些帮助，但无法彻底根除这个问题。如果项目的其他程序员不认真对待接口，那么这些接口的使用是无法得到强制性保证的。除非项目的所有人都同意使用接口并对其进行检查，否则接口的很多价值都无从体现。

## 2.2 其他面向对象语言处理接口的方式

这里先概览一下三种广泛使用的面向对象语言处理接口的方式。你会发现它们的办法大体相似。稍后在2.5节中创建Interface类时，我们会尽量模仿它们的功能。

Java使用接口的方式是面向对象语言中比较典型的，所以我们先从Java开始。下面是java.io包中的一个接口：

```
public interface DataOutput {  
    void writeBoolean(boolean value) throws IOException;  
    void writeByte(int value) throws IOException;
```

```

    void writeChar(int value) throws IOException;
    void writeShort(int value) throws IOException;
    void writeInt(int value) throws IOException;
    ...
}

```

它列出了一个类应该实现的一批方法，包括方法的参数和可能会抛出的异常。每一行都像是一个方法声明，只不过是以一个分号而不是一对大括号结尾的。

创建一个实现这个接口的类需要使用关键字 `implements`：

```

public class DataOutputStream extends FilterOutputStream implements DataOutput {
    public final void writeBoolean (boolean value) throws IOException {
        write (value ? 1 : 0);
    }
    ...
}

```

该类声明并具体实现了接口中列出的每一个方法。漏掉任何一个方法都会导致在编译时显示错误。下面是Java编译器在发现一个接口错误时可能产生的输出信息：

---

MyClass should be declared abstract; it does not define writeBoolean(boolean) in MyClass.<sup>①</sup>

---

PHP的语法与此类似：

```

interface MyInterface {
    public function interfaceMethod($argumentOne, $argumentTwo);
}

class MyClass implements MyInterface {
    public function interfaceMethod($argumentOne, $argumentTwo) {
        return $argumentOne . $arguemntTwo;
    }
}

class BadClass implements MyInterface {
    // No method declarations.
}

// BadClass causes this error at run-time:
// Fatal error: Class BadClass contains 1 abstract methods and must therefore be
// declared abstract (MyInterface::interfaceMethod)

```

C#中也差不多：

```

interface MyInterface {
    string interfaceMethod(string argumentOne, string argumentTwo);
}

class MyClass : MyInterface {

```

<sup>①</sup> MyClass应被声明为抽象类；它没有定义WriteBoolean (boolean) 方法。——译者注

```

public string interfaceMethod(string argumentOne, string argumentTwo) {
    return argumentOne + argumentTwo;
}

class BadClass : MyInterface {
    // No method declarations.
}

// BadClass causes this error at compile-time:
// BadClass does not implement interface member MyInterface.interfaceMethod()

```

上述语言使用接口的方式大体相似。接口结构包含的信息说明了需要实现什么方法以及这些方法应该具有什么参数。类的定义明确地声明它们实现了这个接口（通常是使用`implements`关键字）。一个类可以实现不止一个接口。如果接口中的某个方法没有被实现，则会产生一个错误。在有的语言中错误产生在编译时，而在有的语言中则产生在运行时。错误消息向用户提供了三种信息：类名，接口名和未被实现的方法名。

显然我们不能如法炮制，因为JavaScript没有`interface`和`implements`关键字，也不在运行时对接口约定是否得到遵守进行检查。但是我们可以通过使用辅助类和显式地进行检查模仿出它们的大部分特性。

## 2.3 在 JavaScript 中模仿接口

下面我们将探讨在JavaScript中模仿接口的三种方法：注释法、属性检查法和鸭式辨型法。没有哪种技术是完美的，但三者结合使用基本上可以令人满意。

### 2.3.1 用注释描述接口

用注释模仿接口是最简单的方法，但效果却是最差的。这种方法模仿其他面向对象语言中的做法，使用了`interface`和`implements`关键字，但把它们放在注释中，以免引起语法错误。下面的示例演示了如何把这些关键字添加到代码中以描述所要求的方法：

```

/*
interface Composite {
    function add(child);
    function remove(child);
    function getChild(index);
}

interface FormItem {
    function save();
}

*/

```

```

var CompositeForm = function(id, method, action) { // implements Composite, FormItem
    ...
};

// Implement the Composite interface.

CompositeForm.prototype.add = function(child) {
    ...
};

CompositeForm.prototype.remove = function(child) {
    ...
};

CompositeForm.prototype.getChild = function(index) {
    ...
};

// Implement the FormItem interface.

CompositeForm.prototype.save = function() {
    ...
};

```

这种模仿并不是很好。它没有为确保CompositeForm真正实现了正确的方法集而进行检查，也不会抛出错误以告知程序员程序中有问题。说到底它主要还是属于程序文档范畴。在这种做法中，对接口约定的遵守完全依靠自觉。

尽管如此，这种方法也有其优点。它易于实现，不需要额外的类或函数。它可以提高代码的可重用性，因为现在那些类实现的接口都有说明，程序员可以把它们与其他实现了同样接口的类互换使用。这种方法并不影响文件尺寸或执行速度，因为它所用的注释可以在对代码进行部署时不费吹灰之力地予以剔除。但是，由于不会提供错误消息，它对测试和调试没有什么帮助。  
15

### 2.3.2 用属性检查模仿接口

第二种方法要更严谨一点。所有类都明确地声明自己实现了哪些接口，那些想与这些类打交道的对象可以针对这些声明进行检查。那些接口自身仍然只是注释，但现在你可以通过检查一个属性得知某个类自称实现了什么接口：

```

/*
interface Composite {
    function add(child);
    function remove(child);
    function getChild(index);
}

interface FormItem {
    function save();
}

```

```

}

/*
 * ...
 */

var CompositeForm = function(id, method, action) {
    this.implementsInterfaces = ['Composite', 'FormItem'];
    ...
};

// ...
function addForm(formInstance) {
    if(!implements(formInstance, 'Composite', 'FormItem')) {
        throw new Error("Object does not implement a required interface.");
    }
    ...
}

// The implements function, which checks to see if an object declares that it
// implements the required interfaces.

function implements(object) {
    for(var i = 1; i < arguments.length; i++) { // Looping through all arguments
        // after the first one.
        var interfaceName = arguments[i];
        var interfaceFound = false;
        for(var j = 0; j < object.implementsInterfaces.length; j++) {
            if(object.implementsInterfaces[j] == interfaceName) {
                interfaceFound = true;
                break;
            }
        }
        if(!interfaceFound) {
            return false; // An interface was not found.
        }
    }
    return true; // All interfaces were found.
}

```

在这个例子中，CompositeForm宣称自己实现了Composite和FormItem这两个接口，其做法是把这两个接口的名称加入一个名为implementsInterfaces的数组。类显式声明自己支持什么接口。任何一个要求其参数属于特定类型的函数都可以对这个属性进行检查，并在所需接口未在声明之列时抛出一个错误。

这种方法有几个优点。它对类所实现的接口提供了文档说明。如果需要的接口不在一个类宣称支持的接口之列，你会看到错误消息。通过利用这些错误，你可以强迫其他程序员声明这些接口。

这种方法的主要缺点在于它并未确保类真正实现了自称实现的接口。你只知道它是否说自己

实现了接口。在创建一个类时声明它实现了一个接口，但后来在实现该接口所规定的方法时却漏掉其中的某一个，这一种错误很常见。此时所有检查都能通过，但那个方法却并不存在，这将在代码中埋下一个隐患。另外，显式声明类所支持的接口也需要一些额外工作。

### 2.3.3 用鸭式辨型模仿接口

其实，类是否声明自己支持哪些接口并不重要，只要它具有这些接口中的方法就行。鸭式辨型（这个名称来自James Whitcomb Riley的名言：“像鸭子一样走路并且嘎嘎叫的就是鸭子”）正是基于这样的认识。它把对象实现的方法集作为判断它是不是某个类的实例的唯一标准。这种技术在检查一个类是否实现了某个接口时也可大显身手。这种方法背后的观点很简单：如果对象具有与接口定义的方法同名的所有方法，那么就可以认为它实现了这个接口。你可以用一个辅助函数来确保对象具有所有必需的方法：

```
// Interfaces.

var Composite = new Interface('Composite', ['add', 'remove', 'getChild']);
var FormItem = new Interface('FormItem', ['save']);

// CompositeForm class

var CompositeForm = function(id, method, action) {
    ...
};

...

function addForm(formInstance) {
    ensureImplements(formInstance, Composite, FormItem);
    // This function will throw an error if a required method is not implemented.
    ...
}
```

17

与另外两种方法不同，这种方法并不借助于注释。其各个方面都是可以强制实施的。`ensureImplements`函数需要至少两个参数。第一个参数是想要检查的对象。其余参数是据以对那个对象进行检查的接口。该函数检查其第一个参数代表的对象是否实现了那些接口所声明的所有方法。如果发现漏掉了任何一个方法，它就会抛出错误，其中包含了所缺少的那个方法和未被正确定义的接口的名称等有用信息。这种检查可以用在代码中任何需要确保某个对象实现了某个接口的地方。在本例中，`addForm`函数仅当一个表单对象支持所有必要的方法时才会对其执行添加操作。

尽管鸭式辨型可能是上述三种方法中最有用的一种，但它也有一些缺点。在这种方法中，类并不声明自己实现了哪些接口，这降低了代码的可重用性，并且也缺乏其他两种方法那样的自我描述性。它需要使用一个辅助类（`Interface`）和一个辅助函数（`ensureImplements`）。而且，它只关心方法的名称，并不检查其参数的名称、数目或类型。

## 2.4 本书采用的接口实现方法

本书综合使用了第一种和第三种方法。我们用注释声明类支持的接口，从而提高代码的可重用性及其文档的完善性。我们还用辅助类Interface及其类方法Interface.ensureImplements来对对象实现的方法进行显式检查。如果对象未能通过检查，这个方法将返回一条有用的消息。

下面是一个结合使用Interface类与注释的示例：

```
// Interfaces.

var Composite = new Interface('Composite', ['add', 'remove', 'getChild']);
var FormItem = new Interface('FormItem', ['save']);

// CompositeForm class

var CompositeForm = function(id, method, action) { // implements Composite, FormItem
    ...
};

...
...

function addForm(formInstance) {
    Interface.ensureImplements(formInstance, Composite, FormItem);
    // This function will throw an error if a required method is not implemented,
    // halting execution of the function.
    // All code beneath this line will be executed only if the checks pass.
    ...
}
```

Interface.ensureImplements起着严格把关的作用。如果它发现问题，就会抛出一个错误。这个错误要么被其他代码捕捉到并得到处理，要么中断程序的执行。无论是哪种情况，程序员都能立即知道代码中存在问题，并且知道其来源位置。

## 2.5 Interface 类

下面是本书中使用的Interface类的定义：

```
// Constructor.
```

```
var Interface = function(name, methods) {
    if(arguments.length != 2) {
        throw new Error("Interface constructor called with " + arguments.length +
            "arguments, but expected exactly 2.");
    }
}
```

```
this.name = name;
this.methods = [];
for(var i = 0, len = methods.length; i < len; i++) {
    if(typeof methods[i] !== 'string') {
```

```

        throw new Error("Interface constructor expects method names to be "
            + "passed in as a string.");
    }
    this.methods.push(methods[i]);
}
};

// Static class method.

Interface.ensureImplements = function(object) {
    if(arguments.length < 2) {
        throw new Error("Function Interface.ensureImplements called with " +
            arguments.length + "arguments, but expected at least 2.");
    }

    for(var i = 1, len = arguments.length; i < len; i++) {
        var interface = arguments[i];
        if(interface.constructor !== Interface) {
            throw new Error("Function Interface.ensureImplements expects arguments"
                + "two and above to be instances of Interface.");
        }

        for(var j = 0, methodsLen = interface.methods.length; j < methodsLen; j++) {
            var method = interface.methods[j];
            if(!object[method] || typeof object[method] !== 'function') {
                throw new Error("Function Interface.ensureImplements: object "
                    + "does not implement the " + interface.name
                    + " interface. Method " + method + " was not found.");
            }
        }
    }
};

```

19

从中可以看到，该类的所有方法对其参数都有严格的要求，如果参数未能通过检查，将导致错误的抛出。我们特地加入这种检查的目的在于：如果没有错误被抛出，那么你可以肯定接口已经得到了正确的声明和实现。

### 2.5.1 Interface 类的使用场合

严格的类型检查并不总是明智的。许多JavaScript程序员根本不用接口或它所提供的那种检查，也照样一千多年。接口在运用设计模式实现复杂系统的时候最能体现其价值。它看似降低了JavaScript的灵活性，而实际上，因为使用接口可以降低对象间的耦合程度，所以它提高了代码的灵活性。接口的使用可以让函数变得更灵活，因为你既能向函数传递任何类型的参数，又能保证它只会使用那些具有必要方法的对象。某些场合下接口很有用处。

在有许多程序员参与的大型项目中，接口起着至关重要的作用。程序员常常需要使用还未编写出来的API，或者需要提供一些占位代码（stub）以免延误开发进度。接口在这种场合中的重要性表现在许多方面。它们记载着API，可作为程序员正式交流的工具。在占位代码被替换为最

终的API时，你立刻就能知道所需方法是否得到了实现。在开发过程中，如果API发生了变化，只要新的API实现了同样的接口，它就能天衣无缝地替换原有API。

现在项目中用到来自因特网上的、你无法直接控制的代码的情况越来越普遍。部署在外部环境中的程序库以及搜索、电子邮件、地图等服务的API都是这类代码的例子。即使它们有着可信的来源，也必须谨慎使用，确保其变化不会在自己的代码中引起问题。一种应对之策是为所依赖的每一个API创建一个Interface对象，然后对接收到的每一个对象都进行检查，以确保其正确实现了那些接口：

```
var DynamicMap = new Interface('DynamicMap', ['centerOnPoint', 'zoom', 'draw']);

function displayRoute(mapInstance) {
    Interface.ensureImplements(mapInstance, DynamicMap);
    mapInstance.centerOnPoint(12, 34);
    mapInstance.zoom(5);
    mapInstance.draw();
    ...
}
```

在这个示例中，displayRoute函数要求传入的参数具有3个特定方法。通过使用一个Interface对象和调用Interface.ensureImplements方法，可以确保这些方法已经得到实现，否则你将见到一个错误。这个错误可以用一个try/catch块捕获，然后可能会被用于发送一条Ajax请求，将外部API引起的问题告知用户。这有助于提高mash-up应用系统的稳定性和安全性。

## 2.5.2 Interface 类的用法

判断在代码中使用接口是否划算是最重要（也是最困难）的一步。对于小型的、不太费事的项目来说，接口的好处也许并不明显，只是徒增其复杂度而已。你需要自行权衡其利弊。如果认为在项目中使用接口利大于弊，那么可以参照如下使用说明。

- (1) 将Interface类纳入HTML文件。你可以在本书的附属网站<http://jsdesignpatterns.com/>上下载到Interface.js文件。
- (2) 逐一检查代码中所有以对象为参数的方法。搞清代码的正常运转要求这些对象参数具有哪些方法。
- (3) 为你需要的每一个不同的方法集创建一个Interface对象。
- (4) 剔除所有针对构造器的显式检查。因为我们使用的是鸭式辨型，所以对象的类型不再重要。
- (5) 以Interface.ensureImplements取代原来的构造器检查。

20

这样做有什么好处？答案是代码的耦合程度降低了。因为现在你不再依赖于任何特定的类的实例，而是检查所需要的特性是否都已就绪（不管其具体如何实现）。由此你在对代码进行优化和重构时将拥有更大的自由。

## 2.5.3 示例：使用 Interface 类

假设你要创建一个类，它可以将一些自动化测试结果转化为适于在网页上查看的格式。该类

的构造器以一个TestResult类的实例为参数。它会应客户的请求对这个TestResult对象所封装的数据进行格式化，然后输出。这个ResultFormatter类最初的实现如下：

```
// ResultFormatter class, before we implement interface checking.

var ResultFormatter = function(resultsObject) {
    if(!(resultsObject instanceof TestResult)) {
        throw new Error("ResultsFormatter: constructor requires an instance "
            + "of TestResult as an argument.");
    }
    this.resultsObject = resultsObject;
};

ResultFormatter.prototype.renderResults = function() {
    var dateOfTest = this.resultsObject.getDate();
    var resultsArray = this.resultsObject.getResults();

    var resultsContainer = document.createElement('div');

    var resultsHeader = document.createElement('h3');
    resultsHeader.innerHTML = 'Test Results from ' + dateOfTest.toUTCString();
    resultsContainer.appendChild(resultsHeader);

    var resultsList = document.createElement('ul');
    resultsContainer.appendChild(resultsList);

    for(var i = 0, len = resultsArray.length; i < len; i++) {
        var listItem = document.createElement('li');
        listItem.innerHTML = resultsArray[i];
        resultsList.appendChild(listItem);
    }

    return resultsContainer;
};
```

21

该类的构造器会对参数进行检查，以确保其的确为TestResult类的实例。如果参数达不到要求，构造器将抛出一个错误。有了这样的保证，在编写renderResults方法时，你就可以认定有getDate和getResults这两个方法可供使用。嗯……当真是这样吗？在构造器中只对resultsObject是否为TestResult类的实例进行了检查。实际上这并不能保证所需的方法得到了实现。TestResult类可能会被修改，致使其不再拥有getDate方法。在此情况下，构造器中的检查仍能通过，但renderResults方法却会失灵。

此外，构造器中的这个检查施加了一些不必要的限制。它不允许使用其他类的实例作参数，哪怕它们原本可以如愿发挥作用。例如，有一个名为WeatherData的类也拥有getDate和getResults这两个方法。它本来可以被ResultFormatter类用得非常好的，但是那个显式类型检查（它使用的是instanceOf运算符）会阻止使用WeatherData类的任何实例。

问题的解决办法是删除那个使用instanceOf的检查，并用接口代替它。首先，我们需要创建

这个接口：

```
// ResultSet Interface.

var ResultSet = new Interface('ResultSet', ['getDate', 'getResults']);
```

这行代码创建了一个Interface对象的新实例。第一个参数是接口的名称，第二个参数是一个字符串数组，其中的每个字符串都是一个必需的方法的名称。有了这个接口之后，就可以用接口检查替代instanceOf检查了：

```
// ResultFormatter class, after adding Interface checking.
```

```
var ResultFormatter = function(resultsObject) {
  Interface.ensureImplements(resultsObject, ResultSet);
  this.resultsObject = resultsObject;
};

ResultFormatter.prototype.renderResults = function() {
  ...
};
```

renderResults方法保持不变。而构造器则被改为使用ensureImplements方法而不是instanceOf运算符。现在这个构造器可以接受WeatherData或其他任何实现了所需方法的类的实例。我们只修改了几行ResultFormatter类的代码，就让那个检查变得更准确（它要求参数对象实现所有必需的方法），而且也更宽容（它允许使用任何匹配该接口的对象）。

22

## 2.6 依赖于接口的设计模式

下面列出的设计模式（后面的章节会对它们进行讨论）尤其依赖接口。

- **工厂模式。** 对象工厂所创建的具体对象会因具体情况而异。使用接口可以确保所创建的这些对象可以互换使用。也就是说，对象工厂可以保证其生产出来的对象都实现了必需的方法。
- **组合模式。** 如果不用接口你就不可能使用这个模式。组合模式的中心思想在于可以将对象群体与其组成对象同等对待。这是通过让它们实现同样的接口来做到的。如果不进行某种形式的鸭式辨型或类型检查，组合模式就会失去大部分作用。
- **装饰者模式。** 装饰者通过透明地为另一对象提供包装而发挥作用。这是通过实现与另外那个对象完全相同的接口而做到的。对于外界而言，一个装饰者和它所包装的对象看不出有什么区别。我们将使用Interface类来确保所创建的装饰者对象实现了必需的方法。
- **命令模式。** 代码中所有的命令对象都要实现同一批方法（它们通常被命名为execute、run或undo）。通过使用接口，你为执行这些命令对象而创建的类可以不必知道这些对象具体是什么，只要知道它们都实现了正确的接口即可。藉此你可以创建出模块化程度很高而耦合程度很低的用户界面和API。

本书通篇都要用到接口这个重要概念。你应该斟酌一下，看看自己的项目是否需要用到它。

## 2.7 小结

本章讨论了接口在一些流行的面向对象语言中的使用和实现方式，而且说明了接口概念的各种不同实现方式都有一些共同特性：它们都提供一种规定必需方法的手段；它们都提供一种检查这些方法是否确实得到实现的手段，并且在结果为否定的时候能提供有用的信息。在JavaScript中可以结合使用文档手段（注释）、辅助类和鸭式辨型来模仿这些特性。使用接口的难点在于判断是否有必要使用它。它并不总是不可或缺的。灵活性是JavaScript最强大的特色之一。强制进行不必要的严格类型检查会损害这种灵活性。谨慎地使用Interface类有助于创建更健壮的类和更稳定的代码。

23

24

# 封装和信息隐藏

对象创建私用成员是任何面向对象语言中最基本和有用特性之一。通过将一个方法或属性声明为私用的，可以让对象的实现细节对其他对象保密以降低对象之间的耦合程度，可以保持数据的完整性并对其修改方式加以约束。在代码有许多人参与设计的情况下，这也可以使代码更可靠、更易于调试。简而言之，封装是面向对象的设计的基石。

尽管JavaScript是一种面向对象的语言，它并不具备用以将成员声明为公用或私用的任何内置机制。与讲述接口的前一章一样，我们将自己想办法实现这种特性。目前有几种办法可以用来创建具有公用、私用和特权（privileged）方法的对象，它们各有其优缺点。我们也会讨论在哪些场合下JavaScript程序员能够受益于经过复杂封装的对象。

## 3.1 信息隐藏原则

我们先举个例子来说明信息隐藏原则。假设每天晚上同事都会向你提交一份关于当天收益的报告单。这个接口定义得很清楚：你要求对方提供信息，而你的同事搜集原始数据，计算收益，然后向你汇报。就算你跳槽去了别家公司，由于这个接口依然存在，所以你的接替者很容易用同样的方式要求得到信息汇报。

有一天，你希望能比听取同事的汇报更频繁地得到这种信息。你找到了原始数据的存放地点，并且自己搜集和计算相关数据。事情原本很顺利，但后来情况发生了变化。数据原来采用逗号分隔值格式，但现在改为采用XML格式。而且，计算方法也会根据会计和税收法规进行调整，而你并不精通这些法规。如果你要辞职，那么首先必须教会接替者这些工作，这比起从同事那里请求最终计算结果可要麻烦多了。

在上述场景中，你对产生收益信息的内部细节形成了依赖。如果这种内部实现发生变化，你必须重新学习整个系统并从头开始。用面向对象设计的术语来说，你与原始数据之间形成了强耦合。信息隐藏原则有助于减轻系统中两个参与者之间的依赖性。它指出，两个参与者必须通过明确的通道传送信息。在本例中，这些通道就是对象间的接口。

### 3.1.1 封装与信息隐藏

封装与信息隐藏之间是什么关系呢？你可以把它们视为同一个概念的两种表述。信息隐藏是目的，而封装则是藉以达到这个目的的技术。本章着重讨论一些在JavaScript中进行封装的具体例

子。

封装（encapsulation）可以被定义为对对象的内部数据表现形式和实现细节进行隐藏。要想访问封装过的对象中的数据，只有使用已定义的操作这一种办法。通过封装可以强制实施信息隐藏。许多面向对象语言都使用关键字来说明某些方法和属性应被隐藏。例如在Java中，用关键字private来声明一个方法，可以确保只有该对象内部的代码才能执行它。但在JavaScript中没有这样的关键字，我们将使用闭包的概念来创建只允许从对象内部访问的方法和属性。这比使用关键字的办法更复杂（也更费解），但它也能获得同样的最终效果。

### 3.1.2 接口扮演的角色

在向其他对象隐藏信息的过程中接口是如何发挥作用的呢？接口提供了一份记载着可供公众访问的方法的契约。它定义了两个对象间可以具有的关系。只要接口不变，这个关系的双方都是可替换的。你不一定非得使用像第2章中定义的那种严格的接口，但大多数情况下，你将发现对可以使用的方法加以记载会很有好处。不是有了接口就万事大吉，你应该避免公开未定义于接口中的方法。否则其他对象可能会对那些并不属于接口的方法产生依赖，而这是不安全的。因为这些方法随时都可能发生改变或被删除，从而导致整个系统失灵。

一个理想的软件系统应该为所有类定义接口。这些类只向外界提供它们实现的接口中规定的方法，任何别的方法都留作自用。其所有属性都是私用的，外界只能通过接口中定义的存取操作与之打交道。但实际的系统很少能真正达到这样的境界。优质的代码应尽量向这个目标靠拢，但又不能过于刻板，把那些并不需要这些特性的简单项目复杂化。

## 3.2 创建对象的基本模式

本节将讨论创建对象的各种不同方式及其特点。JavaScript中创建对象的基本模式有3种。  
门户大开型（fully exposed）对象创建方式是最简单的一种，但它只能提供公用成员。第二种做法在此方面有所改进，它使用下划线来表示方法或属性的私用性。第三种做法使用闭包来创建真正私用的成员，这些成员只能通过一些特权方法访问。

23

**注解** 不能简单地说这些定义类的模式中哪种是“正确的”。它们都有各自的利弊。每一种做法都有可能适合你，具体取决于项目的需要。

以Book类为例。假设你接到这样一项任务：创建一个用来存储关于一本书的数据的类，并为其实现一个以HTML形式显示这些数据的方法。你只负责创建这个类，别人会创建并使用其实例。它会被这样使用：

```
// Book(isbn, title, author)
var theHobbit = new Book('0-395-07122-4', 'The Hobbit', 'J. R. R. Tolkien');
theHobbit.display(); // Outputs the data by creating and populating an HTML element.
```

### 3.2.1 门户大开型对象

实现Book类最简单的做法是按传统方式创建一个类，用一个函数来做其构造器。我们称其为门户大开型对象，因为它的所有属性和方法都是公开的、可访问的。这些公用属性需要使用this关键字来创建：

```
var Book = function(isbn, title, author) {
    if(isbn == undefined) throw new Error('Book constructor requires an isbn.');
    this.isbn = isbn;
    this.title = title || 'No title specified';
    this.author = author || 'No author specified';
}

Book.prototype.display = function() {
    ...
};
```

在构造器中，如果检查到没有提供ISBN，将会抛出一个错误。这是因为display方法要求书籍对象有一个准确的ISBN，否则就不能找到相应的图片，也不能生成一个用于购书的链接。title和author参数都是可选的，所以要准备默认值以防它们未被提供。逻辑“或”运算符“||”在此用于提供后备值。如果提供了title或author，那么运算符左边的运算数的求值结果为true，因此这个运算数会被作为运算结果返回。如果没有提供title或author，那么左边的运算数的求值结果为false，作为运算结果返回的是右边的运算数。

乍一看这个类似乎符合一切需要。但其最大的问题是无法检验ISBN数据的完整性，而不完整的ISBN数据有可能导致display方法失灵。这破坏了你与其他程序员之间的契约。如果Book对象在创建时没有抛出任何错误，那么display方法应该能正常工作才对，但是由于没有进行完整性检查，这就不一定了。为了解决这个问题，下面的版本强化了对ISBN的检查：

```
var Book = function(isbn, title, author) {
    if(!this.checkIsbn(isbn)) throw new Error('Book: Invalid ISBN.');
    this.isbn = isbn;
    this.title = title || 'No title specified';
    this.author = author || 'No author specified';
}

Book.prototype = {
    checkIsbn: function(isbn) {
        if(isbn == undefined || typeof isbn != 'string') {
            return false;
        }

        isbn = isbn.replace(/-/,''); // Remove dashes.
        if(isbn.length != 10 && isbn.length != 13) {
            return false;
        }

        var sum = 0;
        if(isbn.length === 10) { // 10 digit ISBN.
```

```

If(!isbn.match(/\d{9}/)) { // Ensure characters 1 through 9 are digits.
    return false;
}

for(var i = 0; i < 9; i++) {
    sum += isbn.charAt(i) * (10 - i);
}
var checksum = sum % 11;
if(checksum === 10) checksum = 'X';
if(isbn.charAt(9) != checksum) {
    return false;
}
}

else { // 13 digit ISBN.
    if(!isbn.match(/^\d{12}/)) { // Ensure characters 1 through 12 are digits.
        return false;
    }

    for(var i = 0; i < 12; i++) {
        sum += isbn.charAt(i) * ((i % 2 === 0) ? 1 : 3);
    }
    var checksum = sum % 10;
    if(isbn.charAt(12) != checksum) {
        return false;
    }
}

return true; // All tests passed.
},
display: function() {
    ...
};

};


```

上述代码中添加了一个checkIsbn方法，以保证ISBN是一个具有正确的位数和校验和的字符串。因为现在该类有了两个方法，所以Book.prototype被设为一个对象字面量，这样在定义多个方法的时候就不用在每个方法前面都加上Book.prototype了。这两种定义方法的做法效果是相同的，本章都会用到。

现在情况看起来有所改善。在创建对象的时候可以对ISBN的有效性进行检查，这可以确保display方法能正常工作。但是现在又出现了一个问题。假设另一个程序员认识到一本书可能会有多个版本，每个版本都有自己的ISBN。他设计了一个用来在这些不同版本之中进行选择的算法，并在实例化书籍对象之后直接用它修改其isbn属性：

```

theHobbit.isbn = '978-0261103283';
theHobbit.display();

```

即使能在构造器中对数据的完整性进行检验，你对其他程序员会把什么样的值直接赋给isbn属性还是毫无控制。为了保护内部数据，你为每个属性都提供了取值器（accessor）和赋值器

(mutator) 方法。取值器方法（通常以getAttributeName这种形式命名）用于获取属性值，而赋值器方法（通常以setAttributeName这种形式命名）则用于设置属性值。通过使用赋值器，你可以在把一个新值真正赋给属性之前进行各种检验。下面是加入了取值器和赋值器之后的新版Book对象：

```
var Publication = new Interface('Publication', ['getIsbn', 'setIsbn', 'getTitle',
    'setTitle', 'getAuthor', 'setAuthor', 'display']);

var Book = function(isbn, title, author) { // implements Publication
    this.setIsbn(isbn);
    this.setTitle(title);
    this.setAuthor(author);
}

Book.prototype = {
    checkIsbn: function(isbn) {
        ...
    },
    getIsbn: function() {
        return this.isbn;
    },
    setIsbn: function(isbn) {
        if(!this.checkIsbn(isbn)) throw new Error('Book: Invalid ISBN.');
        this.isbn = isbn;
    },
    getTitle: function() {
        return this.title;
    },
    setTitle: function(title) {
        this.title = title || 'No title specified';
    },
    getAuthor: function() {
        return this.author;
    },
    setAuthor: function(author) {
        this.author = author || 'No author specified';
    },
    display: function() {
        ...
    }
};
```

注意，上述代码中还定义了一个接口。从现在开始，其他程序员应该只使用这个接口中定义的方法与对象打交道。构造器中也调用了赋值器方法。因为没有必要重复实现同样的检验，所以你在对象的内部也使用这些方法。

这是使用门户大开型对象创建方式所能得到的最好结果。里面包含了一个明确定义的接口、

一些对数据具有保护作用的取值器和赋值器方法，以及一些有效性检验方法。尽管这个设计方案有这样一些特性，它还是存在一个漏洞。虽然我们为设置属性提供了赋值器方法，但那些属性仍然是公开的，可以被直接设置，而在这种方案中却无法阻止这种行为。不管是出于有意（虽然程序员知道那个接口，但却未予理睬）还是无意（程序员不知道他不应该直接对其进行设置），isbn属性都可能会被设置为一个无效值。

尽管这种创建对象的模式存在上述缺陷，它也有许多优点。它易于使用，JavaScript编程新手很快就能学会。创建这样的对象不要求你深入理解作用域或调用链的概念。由于所有方法和属性都是公开的，派生子类和进行单元测试也很容易。唯一的弊端在于无法保护内部数据，而且取值器和赋值器方法也引入了严格说来并非必不可少的额外的代码（在JavaScript文件大小很重要的场合，这可能是一个应该掂量一下的问题）。

### 3.2.2 用命名规范区别私用成员

本节讨论通过使用命名规范模仿私用成员的模式。这种方法致力于解决上一节中遇到的一个问题，即无法阻止其他程序员无意中绕过所有检验步骤。从本质上说这种模式与门户大开型对象创建模式如出一辙，只不过在一些方法和属性的名称前加了下划线以示其私用性而已。

```
var Book = function(isbn, title, author) { // implements Publication
    this.setIsbn(isbn);
    this.setTitle(title);
    this.setAuthor(author);
}

Book.prototype = {
    checkIsbn: function(isbn) {
        ...
    },
    getIsbn: function() {
        return this._isbn;
    },
    setIsbn: function(isbn) {
        if(!this.checkIsbn(isbn)) throw new Error('Book: Invalid ISBN.');
        this._isbn = isbn;
    },
    getTitle: function() {
        return this._title;
    },
    setTitle: function(title) {
        this._title = title || 'No title specified';
    },
    getAuthor: function() {
        return this._author;
    },
}
```

```

setAuthor: function(author) {
  this._author = author || 'No author specified';
},
display: function() {
  ...
};

```

在这个例子中，所有属性都已重新命名。每个属性的名称前都加了一个下划线，表示它是私用属性。由于下划线在JavaScript中可以用作标识符的第一个字符，所以它们仍然是有效的变量名。

这种命名规范也可以应用于方法。假设有一个程序员在使用你的类时颇费周折，因为他老是碰到“Invalid ISBN（无效ISBN）”错误。他可能会使用公用方法checkIsbn，把所有可用于校验和位上的字符（只有10个）依次尝试一遍，直到通过测试，然后用那个结果创建一个Book实例。你应该阻止这种做法，因为这样找到的ISBN仍可能是无效的。为此你可以将这个方法的声明从：

```

checkIsbn: function(isbn) {
  ...
},

```

改为：

```

_checkIsbn: function(isbn) {
  ...
},

```

即使如此，仍可能有程序员以一种钻空子的心态来使用这个函数，但他们是在无意之中用到这个函数的可能性很小。

下划线的这种用法是一个众所周知的命名规范，它表明一个属性（或方法）仅供对象内部使用，直接访问它或设置它可能会导致意想不到的后果。这有助于防止程序员对它的无意使用，却不能防止对它的有意使用。后一个目标的实现需要有真正私用性的方法。

这种创建对象的模式具有门户大开型对象创建模式的所有优点，而且比后者少了一个缺点。但是，它只是一种约定，只有在得到遵守时才有效果，而且并没有什么强制性手段可以保证这一点。所以它并不是真正可以用来隐藏对象内部数据的解决方案。它主要适用于非敏感性的内部方法和属性，也即，那些因为未见于公开的接口，所以类的大多数使用者都不会关心的方法和属性。

### 3.2.3 作用域、嵌套函数和闭包

在讨论真正的私用性方法和属性的实现技术之前，我们先花点时间解释一下这种技术背后的原理。在JavaScript中，只有函数具有作用域。也就是说，在一个函数内部声明的变量在函数外部无法访问。私用属性就其本质而言就是你希望在对象外部无法访问的变量，所以为实现这种拒访性而求助于作用域这个概念是合乎情理的。定义在一个函数中的变量在该函数的内嵌函数中是可以访问的。下面这个示例说明了JavaScript中作用域的特点：

```
function foo() {
  var a = 10;

  function bar() {
    a *= 2;
  }

  bar();
  return a;
}
```

在这个示例中，`a`定义在函数`foo`中，但函数`bar`可以访问它，因为`bar`也定义在`foo`中。`bar`在执行过程中将`a`设置为`a`乘以2。当`bar`在`foo`中被调用时它能够访问`a`，这可以理解。但是如果`bar`是在`foo`外部被调用呢？

```
function foo() {
  var a = 10;

  function bar() {
    a *= 2;
    return a;
  }

  return bar;
}

var baz = foo(); // baz is now a reference to function bar.
baz(); // returns 20.
baz(); // returns 40.
baz(); // returns 80.

var blat = foo(); // blat is another reference to bar.
blat(); // returns 20, because a new copy of a is being used.
```

32

在上述代码中，所返回的对`bar`函数的引用被赋给变量`baz`。这个函数现在是在`foo`外部被调用，但它依然能够访问`a`。这是因为JavaScript中的作用域是词法性的。函数是运行在定义它们的作用域中（本例中是`foo`内部的作用域），而不是运行在调用它们的作用域中。只要`bar`被定义在`foo`中，它就能访问在`foo`中定义的所有变量，即使`foo`的执行已经结束。

这就是闭包的一个例子。在`foo`返回后，它的作用域被保存下来，但只有它返回的那个函数能够访问这个作用域。在前面的示例中，`baz`和`blat`各有这个作用域及`a`的一个副本，而且只有它们自己能对其进行修改。返回一个内嵌函数是创建闭包最常用的手段。

### 3.2.4 用闭包实现私用成员

现在回过来看手头的那个问题：你需要创建一个只能在对象内部访问的变量。闭包用于这个目的看来是再合适不过了，因为借助于闭包你可以创建只允许特定函数访问的变量，而且这些变量在这些函数的各次调用之间依然存在。为了创建私用属性，你需要在构造器函数的作用域中定义相关变量。这些变量可以被定义于该作用域中的所有函数访问，包括那些特权方法：

```

var Book = function(newIsbn, newTitle, newAuthor) { // implements Publication

    // Private attributes.
    var isbn, title, author;

    // Private method.
    function checkIsbn(isbn) {
        ...
    }

    // Privileged methods.
    this.getIsbn = function() {
        return isbn;
    };

    this.setIsbn = function(newIsbn) {
        if(!checkIsbn(newIsbn)) throw new Error('Book: Invalid ISBN.');
        isbn = newIsbn;
    };

    this.getTitle = function() {
        return title;
    };

    this.setTitle = function(newTitle) {
        title = newTitle || 'No title specified';
    };

    this.getAuthor = function() {
        return author;
    };

    this.setAuthor = function(newAuthor) {
        author = newAuthor || 'No author specified';
    };

    // Constructor code.
    this.setIsbn(newIsbn);
    this.setTitle(newTitle);
    this.setAuthor(newAuthor);
};

// Public, non-privileged methods.
Book.prototype = {
    display: function() {
        ...
    }
};

```

那么这与我们先前讲过的其他创建对象的模式有什么不同呢？在其他使用Book的例子中，我们在创建和引用对象的属性时总要使用this关键字。而在本例中，我们用var声明这些变量。这意味着它们只存在于Book构造器中。checkIsbn函数也是用同样的方式声明的，因此成了一个私

用方法。

需要访问这些变量和函数的方法只需声明在Book中即可。这些方法被称为特权方法(*privileged method*)，因为它们是公用方法，但却能够访问私用属性和方法。为了在对象外部能访问这些特权函数，它们的前面都被加上了关键字this。因为这些方法定义于Book构造器的作用域中，所以它们能访问到私用属性。引用这些属性时并没有使用this关键字，因为它们不是公开的。所有取值器和赋值器方法都被改为不加this地直接引用这些属性。

任何不需要直接访问私用属性的方法都可以像原来那样在Book.prototype中声明。display就是这类方法中的一个。它不需要直接访问任何私用属性，因为它可以通过调用getIsbn或getTitle来进行间接访问。只有那些需要直接访问私用成员的方法才应该被设计为特权方法。但特权方法太多又会占用过多内存，因为每个对象实例都包含了所有特权方法的新副本。

用这种方式创建的对象可以具有真正私用的属性。其他程序员不可能直接访问他们创建的Book实例的任何内部数据。由于他们不得不通过赋值器方法设置属性的值，所以属性会得到什么样的值尽在你的掌控之下。

这种对象创建模式解决了其他模式中的所有问题，但它也有自己的一些弊端。在门户大开型对象创建模式中，所有方法都创建在原型对象中，因此不管生成多少对象实例，这些方法在内存中只存在一份。而采用本节讨论的做法，每生成一个新的对象实例都将为每一个私用方法和特权方法生成一个新的副本。这会比其他做法耗费更多内存，所以只宜用在需要真正的私用成员的场合。这种对象创建模式也不利于派生子类，因为所派生出的子类不能访问超类的任何私用属性或方法。相比之下，在大多数语言中，子类都能访问超类的所有私用属性和方法<sup>①</sup>。故在JavaScript中用闭包实现私用成员导致的派生问题被称为“继承破坏封装(*inheritance breaks encapsulation*)”。如果你创建的类以后可能会需要派生出子类，那么最好还是采用这里所讨论的三种对象创建模式中的前两种。

34

### 3.3 更多高级对象创建模式

在前面你已经学到了创建对象的3种基本模式，本节将再讨论一些高级一点的模式。本书第2部分中会更详细地讨论各种具体的模式，这里只是对其中的一些模式作一个简介。

#### 3.3.1 静态方法和属性

前面所讲的作用域和闭包的概念可用于创建静态成员，包括公用的和私用的。大多数方法和属性所关联的是类的实例，而静态成员所关联的则是类本身。换句话说，静态成员是在类的层次上操作，而不是在实例的层次上操作。每个静态成员都只有一份。你稍后将会看到，静态成员是直接通过类对象访问的。

<sup>①</sup> 这种说法不正确。至少在Java、C++和C#这三种主流的面向对象语言中，子类是不能直接访问父类的私用成员的，它们只能直接访问父类的公用成员和受护成员。——译者注

下面是添加了静态属性和方法的Book类：

```

var Book = (function() {

    // Private static attributes.
    var numOfBooks = 0;

    // Private static method.
    function checkIsbn(isbn) {
        ...
    }

    // Return the constructor.
    return function(newIsbn, newTitle, newAuthor) { // implements Publication

        // Private attributes.
        var isbn, title, author;

        // Privileged methods.
        this.getIsbn = function() {
            return isbn;
        };
        this.setIsbn = function(newIsbn) {
            if(!checkIsbn(newIsbn)) throw new Error('Book: Invalid ISBN.');
            isbn = newIsbn;
        };
        this.getTitle = function() {
            return title;
        };
        this.setTitle = function(newTitle) {
            title = newTitle || 'No title specified';
        };
        this.getAuthor = function() {
            return author;
        };
        this.setAuthor = function(newAuthor) {
            author = newAuthor || 'No author specified';
        };

        // Constructor code.
        numOfBooks++; // Keep track of how many Books have been instantiated
                      // with the private static attribute.
        if(numOfBooks > 50) throw new Error('Book: Only 50 instances of Book can be '
            + 'created.');

        this.setIsbn(newIsbn);
        this.setTitle(newTitle);
    }
});

```

```

        this.setAuthor(newAuthor);
    }
})();

// Public static method.
Book.convertToTitleCase = function(inputString) {
    ...
};

// Public, non-privileged methods.
Book.prototype = {
    display: function() {
        ...
    }
};

```

这与3.2.4节中创建的类大体相似，但也有一些重要区别。这里的私用成员和特权成员仍然被声明在构造器中（分别使用var和this关键字）。但那个构造器却从原来的普通函数变成了一个内嵌函数，并且被作为包含它的函数的返回值赋给变量Book。这就创建了一个闭包，你可以把静态的私用成员声明在里面。位于外层函数声明之后的一对空括号很重要，其作用是代码一载入就立即执行这个函数（而不是在调用Book构造函数时）。这个函数的返回值是另一个函数，它被赋给Book变量，Book因此成了一个构造函数。在实例化Book时，所调用的是这个内层函数。外层那个函数只是用于创建一个可以用来存放静态私用成员的闭包。

在本例中，checkIsbn被设计为静态方法，原因是为Book的每个实例都生成这个方法的一个新副本毫无道理。此外还有一个静态属性numOfBooks，其作用在于跟踪Book构造器的总调用次数。本例利用这个属性将Book实例的个数限制为不超过50个。

这些私用的静态成员可以从构造器内部访问，这意味着所有私用函数和特权函数都能访问它们。与其他方法相比，它们有一个明显的优点，那就是在内存中只会存放一份。因为其中那些静态方法被声明在构造器之外，所以它们不是特权方法，不能访问任何定义在构造器中的私用属性。定义在构造器中的私用方法能够调用那些私用静态方法，反之则不然。要判断一个私用方法是否应该被设计为静态方法，一条经验法则是看它是否需要访问任何实例数据。如果它不需要，那么将其设计为静态方法会更有效率（从内存占用的意义上来说），因为它只会被创建一份。

创建公用的静态成员则容易得多，只需直接将其作为构造函数这个对象的属性<sup>①</sup>创建即可，前述代码中的方法convertToTitleCase就是一例。这实际上相当于把构造器作为命名空间来使用。

<sup>①</sup> 注意这里所说的属性与前面所说的**属性**不是一个意思。之前所说的**属性**（在本注中表示为粗体，以示区分）大致相当于C++中类的数据成员（而方法则相当于C++中类的函数成员）。实际上，在JavaScript中，对象是一组属性的无序集合，这些属性可以存放各种类型的数据，如果一个属性存放的数据是一个函数，那么它也被称为该对象的一个方法。先前所说的**属性**实际上是指对象的非方法性属性，本书中“属性”一词基本上都用于这一含义，只有此处是一个例外（之所以不得不这样做，是因为如果不把这个词的话，很难既把这个句子的意思表达清楚而又不引起误解）。——译者注

**注解** 在JavaScript中，除那三种原始类型外，所有其他类型的变量都是对象（即使是三种原始类型，在必要的时候也会被自动包装为对象）。这意味着函数也是对象。因为对象在本质上是一些散列表，所以任何时候都可以为其添加成员。其结果就是函数也可以像其他对象一样具有属性和方法，而且这些属性和方法随时可以添加。

所有公用静态方法如果作为独立的函数来声明其实也同样简单，但最好还是像这样把相关行为集中在一起。这些方法用于与类这个整体相关的任务，而不是与类的任一特定实例相关的任务。它们并不直接依赖于对象实例中包含的任何数据。

### 3.3.2 常量

常量只不过是一些不能被修改的变量。在JavaScript中，可以通过创建只有取值器而没有赋值器的私用变量来模仿常量。因为常量往往是在开发时进行设置，而且不因对象实例的不同而变化，所以将其作为私用静态属性来设计是合乎情理的。假设Class对象有一个名为UPPER\_BOUND的常量，那么为获取这个常量而进行的方法调用如下所示：

```
Class.getUPPER_BOUND();
```

为了实现这个取值器，需要使用我们还未讲过的特权静态方法：<sup>①</sup>

```
var Class = (function() {
    // Constants (created as private static attributes).
    var UPPER_BOUND = 100;
    // Constructor.
    var ctor = function (constructorArgument) {
        ...
        // Privileged static method.
        ctor.getUPPER_BOUND = function() {
            return UPPER_BOUND;
        };
        ...
        // Return the constructor.
        return ctor;
    }();
});
```

37

如果需要使用许多常量，但你不想为每个常量都创建一个取值器方法，那么可以创建一个通用的取值器方法：

```
var Class = (function() {
    // Private static attributes.
```

<sup>①</sup> 原书下面这段代码和再下面那段代码有严重错误。我已经进行了修改。——译者注

```

var constants = {
    UPPER_BOUND: 100,
    LOWER_BOUND: -100
};

// Constructor.
var ctor = function(constructorArgument) {
    // Privileged static method.
    ctor.getConstant = function(name) {
        return constants[name];
    }
    ...
    // Return the constructor.
    return ctor;
}();

```

然后可以这样使用这个取值器以获得一个常量：

```
Class.getConstant('UPPER_BOUND');
```

### 3.3.3 单体和对象工厂

其他还有一些模式也使用闭包来创建受保护的变量空间。在这方面最突出的两个是单体模式和工厂模式。本书后面部分会详细讨论这两种模式，因为它们使用了本章中的概念来实现信息隐藏，所以在此先简要介绍一下。

**38** 单体模式使用一个由外层函数返回的对象字面量来公开特权成员，而私用成员则被保护性地封装在外层函数的作用域中。它使用的技术与我们前面所讲的一样：外层函数在定义之后立即执行，其结果被赋给一个变量。在本章前面的例子中外层函数返回的都是一个函数，而单体模式中外层函数返回的则是一个对象字面量。这种创建受保护的命名空间的方法非常简便和直观。第5章将对单体进行更详细的讨论。

对象工厂也可以使用闭包来创建具有私用成员的对象。其最简形式就是一个类构造器，本章讨论的所有技术都可以用在上面。第7章详细讨论了工厂模式。

## 3.4 封装之利

要是创建对象时不用操心闭包和特权方法这些东西，事情会简单得多。在一个理想的世界里，所有方法都可以是公开的，而其他程序员只会使用接口中规定的那些方法。那么，不厌其烦地隐藏实现细节究竟能为你带来什么好处呢？

封装保护了内部数据的完整性。通过将数据的访问途径限制为取值器和赋值器这两个方法，可以获得对取值和赋值的完全控制。这可以减少其他函数所需的错误检查代码的数量，并确保数

据不会处于无效状态。另外，对象的重构因此可以变得更轻松。因为用户不知道对象的内部细节，所以你可以随心所欲地修改对象内部使用的数据结构和算法，对此外人不会知道，他们也不必知道。

通过只公开那些在接口中规定的方法，可以弱化模块间的耦合。这是面向对象设计最重要的原则之一。尽可能地提高对象的独立性可以带来很多好处。它提高了对象的可重用性，使其在必要的时候可以被替换。使用私用变量也有助于避免命名空间冲突。如果一个变量在代码中其他地方都不能被访问，你就不用老是担心它是否与程序中其他地方的对象或函数重名并因此造成问题。封装还使你可以大幅改动对象的内部细节，而不会影响到其他部分的代码。总的来说，代码的修改变得更轻松，因为你对它所带来的影响已经了如指掌。如果对象的内部数据都是公开的话，你不可能完全清楚代码的修改会带来什么结果。

## 3.5 封装之弊

私用方法很难进行单元测试。因为它们及其内部变量都是私用的，所以在对象外部无法访问到它们。这个问题没有什么很好的应对之策。你要么通过使用公用方法来提供访问途径（这样就葬送了使用私用方法所带来的大多数好处），要么设法在对象内部定义并执行所有单元测试。最好的解决办法是只对公用方法进行单元测试。这应该能覆盖到所有私用方法，尽管对它们的测试只是间接的。这种问题不是JavaScript所独有的，只对公用方法进行单元测试是一种广为接受的处理方式。

使用封装意味着不得不与复杂的作用域链打交道，而这会使错误调试更加困难。一般说来这不算什么大问题，但有时候你会很难区分来自不同作用域的大批同名变量。这个问题不是经过封装的对象所特有的，但实现私用方法和属性所需的闭包会让它变得更加复杂。39

过度封装也是一个潜在的问题。如果你对其他程序员对你的类的需求了解得并不透彻，那么对防止他们修改类的内部细节的热衷可能会过于严厉。预测人们会怎样使用你的代码不容易。封装可能会损害类的灵活性，致使其无法被用于某些你未曾想到过的目的。

最大的问题在于JavaScript中实现封装的困难。其实现过程需要用到的一些复杂对象模式对于大多数编程新手而言太不直观。JavaScript本来就是一门与多数面向对象语言大相径庭的语言，而封装技术涉及的调用链和定义后立即执行的匿名函数等概念更是加大了学习难度。此外，封装技术的应用还使不熟悉特定模式的人难以理解既有代码。注释和程序文档可以提供一些帮助，但并不能完全解决这个问题。如果你想使用这些模式，那么应该确保与你合作的其他程序员也理解它们。

## 3.6 小结

本章讨论了信息隐藏的概念以及如何用封装这种手段来实现它。因为JavaScript没有对封装提供内置的支持，所以其实现必须依赖于一些其他技术。如果能够确信其他程序员只会使用接口中规定的方法，或者并非迫切需要保持内部数据的完整性，那就可以使用门户大开型对象。

命名规范可以用来引导其他程序员，使其明白哪些方法是不宜直接访问的内部方法。如果需要真正的私用成员，那就只能使用闭包了。通过创建一个受保护的变量空间，可以实现公用、私用和特权成员，以及静态类成员和常量。本书后面的多数章节都依赖于这些基本技术，因此它们值得反复玩味一下。只要理解了JavaScript中作用域的特点，你就能模仿出各种面向对象的技术。

# 继承

**在** JavaScript中继承是一个非常复杂的话题，比其他任何面向对象的语言中的继承都复杂得多。在大多数其他面向对象语言中，继承一个类只需使用一个关键字即可。与它们不同，在JavaScript中要想达到~~传承公用成员的目的~~，需要采取一系列措施。更有甚者，JavaScript属于使用原型式继承（我们会向你证明这其实是一个极大的优点）的少数语言之一。得益于这种语言的灵活性，你既可使用标准的基于类的继承，也可使用更微妙一些（但也可能更有效一些）的原型式继承。

本章将讨论在JavaScript中创建子类的各种技术以及它们的适用场合。

## 4.1 为什么需要继承

在开始实际摆弄代码之前，我们先得搞清使用继承能带来什么好处。一般说来，在设计类的时候，我们希望能减少重复性的代码，并且尽量弱化对象间的耦合。使用继承符合前一个设计原则的需要。借助这种机制，你可以在现有类的基础上进行设计并充分利用它们已经具备的各种方法，而对设计进行修改也更为轻松。假设你需要让几个类都拥有一个按特定方式输出类结构的 `toString` 方法，当然可以用复制加粘贴的办法把定义 `toString` 方法的代码添加到每一个类中，但这样做的话，每当需要改变这个方法的工作方式时，你将不得不在每一个类中重复同样的修改。反之，如果你先创建一个 `ToStringProvider` 类，然后让那些类继承这个类，那么 `toString` 这个方法只需在一个地方声明即可。

让一个类继承另一个类可能会导致二者产生强耦合，也即一个类依赖于另一个类的内部实现。我们将讨论一些有助于避免这种问题的技术，其中包括用掺元类为其他类提供方法这种技术。

## 4.2 类式继承

JavaScript可以被装扮成使用类式继承的语言。通过用函数来声明类、用关键字 `new` 来创建实例，JavaScript中的对象也能惟妙惟肖地模仿Java或C++中的对象。下面是JavaScript中一个简单的类声明：

```
/* Class Person. */

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

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

首先要做的是创建构造函数。按惯例，其名称就是类名，首字母应大写。在构造函数中，创建实例属性要使用关键字this。类的方法则被添加到其prototype对象中，就像例中的Person.prototype.getName那样。要创建该类的实例，只需结合关键字new调用这个构造函数即可：

```
var reader = new Person('John Smith');
reader.getName();
```

然后你可以访问所有的实例属性，也可以调用所有的实例方法。这是JavaScript中一个非常简单的类的例子。

### 4.2.1 原型链

创建继承Person的类则要复杂一些：

```
/* Class Author. */

function Author(name, books) {
  Person.call(this, name); // Call the superclass's constructor in the scope of this.
  this.books = books; // Add an attribute to Author.
}

Author.prototype = new Person(); // Set up the prototype chain.
Author.prototype.constructor = Author; // Set the constructor attribute to Author.
Author.prototype.getBooks = function() { // Add a method to Author.
  return this.books;
};
```

让一个类继承另一个类需要用到许多行代码（不像大多数别的面向对象的语言中那样只用一个关键字extend即可）。首先要做的是像前一个示例中那样创建一个构造函数。在构造函数中，调用超类的构造函数，并将name参数传给它。这行代码需要解释一下。在使用new运算符的时候，系统会为你做一些事。它先创建一个空对象，然后调用构造函数，在此过程中这个空对象处于作用域链的最前端。而在Author函数中调用超类的构造函数时，你必须手工完成同样的任务。“Person.call(this, name)”这条语句调用了Person构造函数，并且在此过程中让那个空对象（用this代表）处于作用域链的最前端，而name则被作为参数传入。

下一步是设置原型链。尽管相关代码比较简单，但这实际上是一个非常复杂的话题。前面已经说过，JavaScript没有extend关键字。但是在JavaScript中每个对象都有一个名为prototype的属

性，这个属性要么指向另一个对象，要么是`null`<sup>①</sup>。在访问对象的某个成员时（比如`reader.getName`），如果这个成员未见于当前对象，那么JavaScript会在`prototype`属性所指的对象<sup>②</sup>中查找它。如果在那个对象中也没有找到，那么JavaScript会沿着原型链向上逐一访问每个原型对象，直到找到这个成员（或已经查过原型链最顶端的`Object.prototype`对象）。这意味着为了让一个类继承另一个类，只需将子类的`prototype`设置为指向超类的一个实例即可。这与其他语言中的继承机制迥然不同，可能会非常令人费解，而且有违直觉。

为了让`Author`继承`Person`，必须手工将`Author`的`prototype`设置为`Person`的一个实例。最后一个步骤是将`prototype`的`constructor`属性重设为`Author`（因为把`prototype`属性设置为`Person`的实例时，其`constructor`属性被抹除了）<sup>③</sup>。

尽管本例中为实现继承需要额外使用三行代码，但是创建这个新的子类的实例与创建`Person`的实例没什么不同：

```
var author = [];
author[0] = new Author('Dustin Diaz', ['JavaScript Design Patterns']);
author[1] = new Author('Ross Harms', ['JavaScript Design Patterns']);

author[1].getName();
author[1].getBooks();
```

由此可见，类式继承的所有复杂性只限于类的声明，创建新实例的过程仍然很简单。

## 4.2.2 extend 函数

为了简化类的声明，可以把派生子类的整个过程包装在一个名为`extend`的函数中。它的作用与其他语言中的`extend`关键字类似，即基于一个给定的类结构创建一个新的类：

```
/* Extend function. */

function extend(subClass, superClass) {
    var F = function() {};
    F.prototype = superClass.prototype;
    subClass.prototype = new F();
```

- 
- ① 这种说法并不正确。每个对象都有一个原型对象，但这并不意味着每个对象都有一个`prototype`属性（实际上只有函数对象才有这个属性）。在创建一个对象时，JavaScript会自动将其原型对象设置为其构造函数的`prototype`属性所指的对象。应该注意的是，构造函数本身也是一个对象，它也有自己的原型对象，但这个原型对象并不是它的`prototype`属性所指向的那个对象。函数作为一个对象，其构造函数是`Function`。因此，构造函数的原型对象实际上是`Function.prototype`所指的对象。关于原型对象的详细说明，参见Flanagan所著《JavaScript权威指南》一书9.2节。——译者注
  - ② 由于前一注释中所说的原因，这里所说的“`prototype`属性所指的对象”应该改称“其原型对象”。后面出现的类似问题将直接在文中修改，不再用译注的形式注明。后文中如果说“`prototype`对象”，指的是`prototype`属性所指的对象；如果说“原型对象”，则是指一个对象的原型对象。——译者注
  - ③ 定义一个构造函数时，其默认的`prototype`对象是一个`Object`类型的实例，其`constructor`属性会被自动设置为该构造函数本身。如果手工将其`prototype`设置为另一个对象，那么新对象自然不会具有原对象的`constructor`值，所以需要重新设置其`constructor`属性。——译者注

```

    subClass.prototype.constructor = subClass;
}

```

这个函数所做的事与先前我们手工做的一样。它设置了prototype，然后再将其constructor重设为恰当的值。作为一项改进，它添加了一个空函数F，并将用它创建的一个对象实例插入原型链中。这样做可以避免创建超类的新实例，因为它可能会比较庞大，而且有时超类的构造函数有一些副作用，或者会执行一些需要进行大量计算的任务。

使用了extend函数后，前面那个Person/Author例子变成了这个样子：

```

/* Class Person. */

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

Person.prototype.getName = function() {
  return this.name;
}

/* Class Author. */

function Author(name, books) {
  Person.call(this, name);
  this.books = books;
}

extend(Author, Person);

Author.prototype.getBooks = function() {
  return this.books;
};

```

本例不像先前那样手工设置prototype和constructor属性，而是通过在类声明之后（在向prototype添加任何方法之前）立即调用extend函数来达到同样的目的。唯一的问题是超类（Person）的名称被固化在了Author类的声明之中。更好的做法是像下面这样用一种更具普适性的方式来自引用父类：

```

/* Extend function, improved. */

function extend(subClass, superClass) {
  var F = function() {};
  F.prototype = superClass.prototype;
  subClass.prototype = new F();
  subClass.prototype.constructor = subClass;

  subClass.superclass = superClass.prototype;
  if(superClass.prototype.constructor == Object.prototype.constructor) {
    superClass.prototype.constructor = superClass;
  }
}

```

这个版本要长一点，它提供了superclass属性，这个属性可以用来弱化Author与Person之间的耦合。该函数的前面4行与前一版本相同。它的最后3行代码则用来确保超类的constructor属性<sup>①</sup>已被正确设置（即使超类就是Object类本身），在用这个新的superclass属性调用超类的构造函数时这个问题很重要：

```
/* Class Author. */

function Author(name, books) {
  Author.superclass.constructor.call(this, name);
  this.books = books;
}
extend(Author, Person);

Author.prototype.getBooks = function() {
  return this.books;
};
```

有了superclass属性，就可以直接调用超类中的方法。这在既要重定义超类的某个方法而又想访问其在超类中的实现时可以派上用场。例如，为了用一个新的getName方法重定义Person类中的同名方法，你可以先用Author.superclass.getName获得作者的名字，然后在此基础上添加其他信息：

```
Author.prototype.getName = function() {
  var name = Author.superclass.getName.call(this);
  return name + ', Author of ' + this.getBooks().join(', ');*/
};
```

## 4.3 原型式继承

原型式继承与类式继承截然不同。我们发现在谈到它的时候，最好忘掉自己关于类和实例的一切知识，只从对象的角度来思考。用基于类的办法来创建对象包括两个步骤：首先，用一个类的声明定义对象的结构；第二，实例化该类以创建一个新对象。用这种方式创建的对象都有一套该类的所有实例属性的副本。每一个实例方法都只存在一份，但每个对象都有一个指向它的链接。

使用原型式继承时，并不需要用类来定义对象的结构，只需直接创建一个对象即可。这个对象随后可以被新的对象重用，这得益于原型链查找的工作机制。该对象被称为原型对象（prototype object），这是因为它为其他对象应有的模样提供了一个原型。这正是原型式继承这个名称的由来。

下面我们将使用原型式继承来重新设计Person和Author：

```
/* Person Prototype Object. */

var Person = {
  name: 'default name',
  getName: function() {
    return this.name;
  }
};
```

<sup>①</sup> 严格地说，是超类的prototype的constructor属性。——译者注

这里并没有使用一个名为Person的构造函数来定义类的结构，Person现在是一个对象字面量。它是所要创建的其他各种类Person对象的原型对象。其中定义了所有类Person对象都要具备的属性和方法，并为它们提供了默认值。方法的默认值可能不会被改变，而属性的默认值一般都会被改变：

```
var reader = clone(Person);
alert(reader.getName()); // This will output 'default name'.
reader.name = 'John Smith';
alert(reader.getName()); // This will now output 'John Smith'.
```

clone函数（稍后在4.3.2节中会详细说明）可以用来创建新的类Person对象。它会创建一个空对象，而该对象的原型对象被设置成为Person。这意味着在这个新对象中查找某个方法或属性时，如果找不到，那么查找过程会在其原型对象中继续进行。

你不必为创建Author而定义一个Person的子类，只要执行一次克隆即可：

```
/* Author Prototype Object. */

var Author = clone(Person);
Author.books = []; // Default value.
Author.getBooks = function() {
  return this.books;
}
```

然后你可以重定义该克隆中的方法和属性。可以修改在Person中提供的默认值，也可以添加新的属性和方法。这样一来就创建了一个新的原型对象，你可以将其用于创建新的类Author对象：

```
var author = [];

author[0] = clone(Author);
author[0].name = 'Dustin Diaz';
author[0].books = ['JavaScript Design Patterns'];

author[1] = clone(Author);
author[1].name = 'Ross Harmes';
author[1].books = ['JavaScript Design Patterns'];

author[1].getName();
author[1].getBooks();
```

### 4.3.1 对继承而来的成员的读和写的不对等性

前面说过，为了有效地使用原型式继承，你必须忘记有关类式继承的一切。这里就是一个例子。在类式继承中，Author的每一个实例都有一份自己的books数组副本。你可以用代码author[1].books.push('New Book Title')为其添加元素。但是对于使用原型式继承方式创建的类Author对象来说，由于原型链接的工作方式，这种做法并非一开始就能行得通。一个克隆并非其原型对象的一份完全独立的副本，它只是一个以那个对象为原型对象的空对象而已。克隆刚被创建时，author[1].name其实是一个返指最初的Person.name的链接。对于从原型对象继承而来的成

员，其读和写具有内在的不对等性。在读取author[1].name的值时，如果你还没有直接为author[1]实例定义name属性的话，那么所得到的是其原型对象的同名属性值。而在写入author[1].name的值时，你是在直接为author[1]对象定义一个新属性。

下面这个示例显示了这种不对等性：

```
var authorClone = clone(Author);
alert(authorClone.name); // Linked to the primitive Person.name, which is the
                        // string 'default name'.
authorClone.name = 'new name'; // A new primitive is created and added to the
                                // authorClone object itself.
alert(authorClone.name); // Now linked to the primitive authorClone.name, which
                        // is the string 'new name'.
```

```
authorClone.books.push('new book'); // authorClone.books is linked to the array
                                    // Author.books. We just modified the
                                    // prototype object's default value, and all
                                    // other objects that link to it will now
                                    // have a new default value there.
authorClone.books = []; // A new array is created and added to the authorClone
                      // object itself.
authorClone.books.push('new book'); // We are now modifying that new array.
```

这也说明了为什么必须为通过引用传递的数据类型的属性创建新副本。在上面的例子中，向authorClone.books数组添加新元素实际上是把这个元素添加到Author.books数组中。这可不是什么好事，因为你对那个值的修改不仅会影响到Author，而且会影响到所有继承了Author但还未改写那个属性的默认值的对象。在改变所有那些数组和对象的成员之前，必须先为其创建新的副本。稍不留神，你可能就会忘记这个问题并改动原型对象的值。这种错误必须尽量避免，因为这类问题调试起来会非常费时。在这类场合中，可以使用hasOwnProperty方法来区分对象的实际成员和它继承而来的成员。

有时原型对象自己也含有子对象。如果想覆盖其子对象中的一个属性值，你不得不重新创建整个子对象。这可以通过将该子对象设置为一个空对象字面量，然后对其进行重塑而办到。但这意味着克隆出来的对象必须知道其原型对象的每一个子对象的确切结构和默认值。为了尽量弱化对象之间的耦合，任何复杂的子对象都应该使用方法来创建：

```
var CompoundObject = {
  string1: 'default value',
  childObject: {
    bool: true,
    num: 10
  }
}
var compoundObjectClone = clone(CompoundObject);

// Bad! Changes the value of CompoundObject.childObject.num.
compoundObjectClone.childObject.num = 5;
```

```
// Better. Creates a new object, but compoundObject must know the structure
// of that object, and the defaults. This makes CompoundObject and
// compoundObjectClone tightly coupled.
compoundObjectClone.childObject = {
  bool: true,
  num: 5
};
```

在这个例子中，为compoundObjectClone对象新添了一个childObject属性，并修改了它所指向的对象的num属性。问题在于compoundObjectClone必须知道childObject具有两个默认值分别为true和10的属性。更好的办法是用一个工厂方法来创建childObject：

```
// Best approach. Uses a method to create a new object, with the same structure and
// defaults as the original.
```

```
var CompoundObject = {};
CompoundObject.string1 = 'default value',
CompoundObject.createChildObject = function() {
  return {
    bool: true,
    num: 10
  }
};
CompoundObject.childObject = CompoundObject.createChildObject();

var compoundObjectClone = clone(CompoundObject);
compoundObjectClone.childObject = CompoundObject.createChildObject();
compoundObjectClone.childObject.num = 5;
```

### 4.3.2 clone 函数

在前面的例子中用来创建克隆对象的奇妙函数究竟是个什么样子呢？答案如下：

```
/* Clone function. */

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

clone函数首先创建了一个新的空函数F，然后将F的prototype属性设置为作为参数object传入的原型对象。由此可以体会到JavaScript最初的设计者的用意。prototype属性就是用来指向原型对象的，通过原型链接机制，它提供了到所有继承而来的成员的链接。该函数最后通过把new运算符作用于F创建出一个新对象<sup>①</sup>，然后把这个新对象作为返回值返回。函数所返回的这个克隆结果是一个以给定对象为原型对象的空对象。

<sup>①</sup> new F相当于new F()。——译者注

## 4.4 类式继承和原型式继承的对比

类式继承和原型式继承是大相径庭的两种继承范型，它们生成的对象也有不同的行为方式。两种继承范型都各有其优缺点，为了判断在特定场合下应该使用哪一种，你需要对此有所了解。

包括JavaScript程序员在内的整个程序员群体对类式继承都比较熟悉。几乎所有用面向对象方式编写的JavaScript代码中都用到了这种继承范型。如果你设计的是一个供众人使用的API，或者可能会有不熟悉原型式继承的其他程序员基于你的代码进行设计，那么最好还是使用类式继承。在各种流行语言中只有JavaScript使用原型式继承，因此可能很多人从来没有用过这种继承。而对象具有到自己的原型对象的反向链接，这也是一个令人困惑的机制。那些没有完全理解原型式继承的程序员会把它视为某种反向继承，即父类继承子类。即使事实并非如此，原型式继承仍然是一个令人费解的话题。但是，因为JavaScript中的类式继承仅仅是对真正基于类的继承的一种模仿，所以那些高级JavaScript程序员总有一天需要懂得原型式继承的工作机制。有人认为隐瞒这一事实其实是弊大于利。

原型式继承更能节约内存。原型链读取成员<sup>①</sup>的方式使得所有克隆出来的对象都共享每个属性和方法的唯一一份实例，只有在直接设置了某个克隆出来的对象的属性和方法时，情况才会有变化。与此相比，在类式继承方式中创建的每一个对象在内存中都有自己的一套属性（和私用方法）的副本。原型式继承在这方面的节约效果很突出。这种继承也比类式继承显得更为简练，它只用到了一个clone函数，不像后者那样需要为每一个想扩展的类写上好几行像SuperClass.call(this, arg)和SubClass.prototype = new SuperClass这样的晦涩代码（当然，这几行代码中的一部分也可以被塞到extend函数中）。不要把原型式继承的简洁看作是简陋，它的力量蕴涵在其简洁性之中。

该使用类式继承还是原型式继承也许主要还是取决于你更喜欢哪种范型。有些人似乎天生就容易被原型式继承的简洁性吸引，而另一些人却对更熟悉的类式继承情有独钟。本书介绍的每一种设计模式中都可以使用这两种继承范型。为了便于理解，在讲述后面的设计模式时我们主要使用类式继承，但这两种继承范型在整本书中都是可以互换使用的。

## 4.5 继承与封装

在本章中到目前为止基本没提到过封装对继承的影响。从现有的类派生出一个子类时，只有公用和特权成员会被承袭下来。这与其他面向对象语言中的情况类似。以Java为例，其子类就无法访问到父类的私用方法；为了将一个方法遗传给子类，必须在定义它时使用关键字protected。

由于这个原因，门户大开型类是最适合于派生子类的。它们的所有成员都是公开的，因此可以被遗传给子类。如果某个成员需要稍加隐藏，你可以使用下划线符号规范。

在派生具有真正的私用成员的类时，因为其特权方法是公用的，所以它们会被遗传下来。籍此可以在子类中间接访问父类的私用属性，但子类自身的实例方法都不能直接访问这些私用属

<sup>①</sup> 指在原型链中查找成员。——译者注

性。父类的私用成员只能通过这些既有的特权方法进行访问，你不能在子类中添加能够直接访问它们的新的特权方法。

## 4.6 捎元类

有一种重用代码的方法不需要用到严格的继承。如果想把一个函数用到多个类中，可以通过扩充（augmentation）的方式让这些类共享该函数。其实际做法大体为：先创建一个包含各种通用方法的类，然后再用它扩充其他类。这种包含通用方法的类称为捎元类（mixin class）。它们通常不会被实例化或直接调用。其存在的目的只是向其他类提供自己的方法。咱们最好还是用一个示例来演示一下：

```
/* Mixin class. */

var Mixin = function() {};
Mixin.prototype = {
  serialize: function() {
    var output = [];
    for(key in this) {
      output.push(key + ':' + this[key]);
    }
    return output.join(',');
  }
};
```

这个Mixin类只有一个名为serialize的方法。这个方法遍历this对象的所有成员并将它们的值组织为一个字符串输出（这只是一个简化的例子。更健壮的版本参见Douglas Crockford的JSON库（<http://json.org/json.js>）中的toJSONString方法）。这种方法可能在许多不同类型的类中都会用到，但没有必要让这些类都继承Mixin，把这个方法的代码复制到这些类中也并不明智。最好还是用augment函数把这个方法添加到每一个需要它的类中：

```
augment(Author, Mixin);

var author = new Author('Ross Harmes', ['JavaScript Design Patterns']);
var serializedString = author.serialize();
```

50

在此我们用Mixin类中的所有方法扩充了Author类。Author类的实例现在就可以调用serialize方法了。这可以被视为多亲继承（multiple inheritance）在JavaScript中的一种实现方式。C++和Python这类语言允许子类继承多个超类。这在JavaScript中是不允许的，因为一个对象只能拥有一个原型对象。不过，由于一个类可以用多个捎元类加以扩充，所以这实际上实现了多继承的效果。

augment函数很简单。它用一个for..in循环遍访予类（giving class）的prototype中的每一个成员，并将其添加到受类（receiving class）的prototype中。如果受类中已经存在同名成员，则跳过这个成员，转而处理下一个。受类中的成员不会被改写：

```
/* Augment function. */

function augment(receivingClass, givingClass) {
  for(methodName in givingClass.prototype) {
    if(!receivingClass.prototype[methodName]) {
      receivingClass.prototype[methodName] = givingClass.prototype[methodName];
    }
  }
}
```

这个函数还可以再改进一下。如果掺元类中包含许多方法，但你只想复制其中的一两个，那么上面这个版本的augment函数是无法满足需要的。下面的新版本会检查是否存在额外的可选参数，如果存在，则只复制那些名称与这些参数匹配的方法：

```
/* Augment function, improved. */

function augment(receivingClass, givingClass) {
  if(arguments[2]) { // Only give certain methods.
    for(var i = 2, len = arguments.length; i < len; i++) {
      receivingClass.prototype[arguments[i]] = givingClass.prototype[arguments[i]];
    }
  } else { // Give all methods.
    for(methodName in givingClass.prototype) {
      if(!receivingClass.prototype[methodName]) {
        receivingClass.prototype[methodName] = givingClass.prototype[methodName];
      }
    }
  }
}
```

现在就可以用`augment(Author, Mixin, 'serialize');`这条语句来达到只为Author类添加一个`serialize`方法的目的了。如果想添加更多方法，只要把它们的名称作为参数传入即可。

用一些方法来扩充一个类有时比让这个类继承另一个类更合适。这是一种避免出现重复性代码的轻便的解决办法。不过适合这种方案的场合并不是很多。只有那些通用到足以使其在彼此大不相同的各种类中都能派上用场的方法才适合于共享（要是那些类彼此的差异不是那么大，那么普通的继承往往更合适）。

## 4.7 示例：就地编辑

我们将提供这个示例的三种解决方案，它们分别演示了类式继承、原型式继承和掺元类的用法。假设你的任务是编写一个用于创建和管理就地编辑域的可重用的模块化API（就地编辑（edit-in-place）是指网页上的一段普通文本被点击后就变成一个配有一些按钮的表单域，以便用户就地对这段文本进行编辑）。使用这个API，用户应该能够为对象<sup>①</sup>分配一个唯一的ID值，能够

<sup>①</sup> 指前面所说的就地编辑域。——译者注

为它提供一个默认值，并且能够指定其在页面上的目标位置。用户还应该在任何时候都可以访问到这个域的当前值，并且可以选择具体使用的编辑域（比如多行文本框或单行文本框）。

### 4.7.1 类式继承解决方案

我们先用类式继承创建这个API：

```
/* EditInPlaceField class. */

function EditInPlaceField(id, parent, value) {
    this.id = id;
    this.value = value || 'default value';
    this.parentElement = parent;

    this.createElements(this.id);
    this.attachEvents();
};

EditInPlaceField.prototype = {
    createElements: function(id) {
        this.containerElement = document.createElement('div');
        this.parentElement.appendChild(this.containerElement);

        this.staticElement = document.createElement('span');
        this.containerElement.appendChild(this.staticElement);
        this.staticElement.innerHTML = this.value;

        this.fieldElement = document.createElement('input');
        this.fieldElement.type = 'text';
        this.fieldElement.value = this.value;
        this.containerElement.appendChild(this.fieldElement);

        this.saveButton = document.createElement('input');
        this.saveButton.type = 'button';
        this.saveButton.value = 'Save';
        this.containerElement.appendChild(this.saveButton);

        this.cancelButton = document.createElement('input');
        this.cancelButton.type = 'button';
        this.cancelButton.value = 'Cancel';
        this.containerElement.appendChild(this.cancelButton);
        this.convertToText();
    },
    attachEvents: function() {
        var that = this;
        addEvent(this.staticElement, 'click', function() { that.convertToEditable(); });
        addEvent(this.saveButton, 'click', function() { that.save(); });
        addEvent(this.cancelButton, 'click', function() { that.cancel(); });
    },
}
```

```
convertToEditable: function() {
    this.staticElement.style.display = 'none';
    this.fieldElement.style.display = 'inline';
    this.saveButton.style.display = 'inline';
    this.cancelButton.style.display = 'inline';

    this.setValue(this.value);
},
save: function() {
    this.value = this.getValue();
    var that = this;
    var callback = {
        success: function() { that.convertToText(); },
        failure: function() { alert('Error saving value.'); }
    };
    ajaxRequest('GET', 'save.php?id=' + this.id + '&value=' + this.value, callback);
},
cancel: function() {
    this.convertToText();
},
convertToText: function() {
    this.fieldElement.style.display = 'none';
    this.saveButton.style.display = 'none';
    this.cancelButton.style.display = 'none';
    this.staticElement.style.display = 'inline';

    this.setValue(this.value);
},
setValue: function(value) {
    this.fieldElement.value = value;
    this.staticElement.innerHTML = value;
},
getValue: function() {
    return this.fieldElement.value;
}
};
```

要创建一个就地编辑域，只需实例化这个类即可：

```
var titleClassical = new EditInPlaceField('titleClassical', $('#doc'), 'Title Here');
var currentTitleText = titleClassical.getValue();
```

上述语句创建了EditPlaceField类（稍后我们将从它派生出新类）的一个实例，它用一个span标签显示文字，并用一个单行文本框作为文字的编辑区。它还具有一些配置方法(createElements、attachEvents)、一些用于转换和保存的内部方法(convertToEditable、save、cancel、convertToText)以及一对取值器和赋值器(getValue、setValue)。如果是实际使用中的代码，你最好为其中的每个HTML元素指定一个特定的class属性值，以便用CSS来设置它们的样式。为简洁起见，我们并没有这样做。

接下来我们要创建一个使用多行文本框而不是单行文本框的类。这个EditInPlaceArea类与EditInPlaceField类有很多共同之处，所以我们将前者作为后者的子类处理，以免编写重复性的代码：

```
/* EditInPlaceArea class. */

function EditInPlaceArea(id, parent, value) {
    EditInPlaceArea.superclass.constructor.call(this, id, parent, value);
}
extend(EditInPlaceArea, EditInPlaceField);

// Override certain methods.

EditInPlaceArea.prototype.createElements = function(id) {
    this.containerElement = document.createElement('div');
    this.parentElement.appendChild(this.containerElement);

    this.staticElement = document.createElement('p');
    this.containerElement.appendChild(this.staticElement);
    this.staticElement.innerHTML = this.value;

    this.fieldElement = document.createElement('textarea');
    this.fieldElement.value = this.value;
    this.containerElement.appendChild(this.fieldElement);

    this.saveButton = document.createElement('input');
    this.saveButton.type = 'button';
    this.saveButton.value = 'Save';
    this.containerElement.appendChild(this.saveButton);

    this.cancelButton = document.createElement('input');
    this.cancelButton.type = 'button';
    this.cancelButton.value = 'Cancel';
    this.containerElement.appendChild(this.cancelButton);
    this.convertToText();
};

EditInPlaceArea.prototype.convertToEditable = function() {
    this.staticElement.style.display = 'none';
    this.fieldElement.style.display = 'block';
    this.saveButton.style.display = 'inline';
    this.cancelButton.style.display = 'inline';

    this.setValue(this.value);
};

EditInPlaceArea.prototype.convertToText = function() {
    this.fieldElement.style.display = 'none';
    this.saveButton.style.display = 'none';
    this.cancelButton.style.display = 'none';
    this.staticElement.style.display = 'block';
}
```

```
this.setValue(this.value);  
};
```

上述代码中用extend函数实现子类的派生，然后在子类中重定义了父类中的一些方法，以体现出二者的差别。这个新类用一个多行文本框取代父类中的单行文本框，用一个p标签取代父类中的span标签。

类式继承技术用在这个案例中看起来很理想。从EditInPlaceField派生子类只是举手之劳，用不了多少代码。要体现子类与父类的差别，只需在子类中重定义一些父类的方法或添加一些新方法即可。我们还可以把这种就地编辑域与别的输出方式关联起来，为此需要另外创建一个子类，并在其中重定义父类的save方法。因为此例中类之间的差异很小，所以这种严格的继承是一种理想的选择。

## 4.7.2 原型式继承解决方案

尽管类式继承和原型式继承有根本性的差别，但从改用原型式继承完成这个任务的过程中你会发现两种方案的最终代码非常相似：

```
/* EditInPlaceField object. */  
  
var EditInPlaceField = {  
    configure: function(id, parent, value) {  
        this.id = id;  
        this.value = value || 'default value';  
        this.parentElement = parent;  
  
        this.createElements(this.id);  
        this.attachEvents();  
    },  
    createElements: function(id) {  
        this.containerElement = document.createElement('div');  
        this.parentElement.appendChild(this.containerElement);  
        this.staticElement = document.createElement('span');  
        this.containerElement.appendChild(this.staticElement);  
        this.staticElement.innerHTML = this.value;  
  
        this.fieldElement = document.createElement('input');  
        this.fieldElement.type = 'text';  
        this.fieldElement.value = this.value;  
        this.containerElement.appendChild(this.fieldElement);  
  
        this.saveButton = document.createElement('input');  
        this.saveButton.type = 'button';  
        this.saveButton.value = 'Save';  
        this.containerElement.appendChild(this.saveButton);  
  
        this.cancelButton = document.createElement('input');  
        this.cancelButton.type = 'button';
```

```

this.cancelButton.value = 'Cancel';
this.containerElement.appendChild(this.cancelButton);

this.convertToText();
},
attachEvents: function() {
    var that = this;
    addEvent(this.staticElement, 'click', function() { that.convertToEditable(); });
    addEvent(this.saveButton, 'click', function() { that.save(); });
    addEvent(this.cancelButton, 'click', function() { that.cancel(); });
},
convertToEditable: function() {
    this.staticElement.style.display = 'none';
    this.fieldElement.style.display = 'inline';
    this.saveButton.style.display = 'inline';
    this.cancelButton.style.display = 'inline';

    this.setValue(this.value);
},
save: function() {
    this.value = this.getValue();
    var that = this;
    var callback = {
        success: function() { that.convertToText(); },
        failure: function() { alert('Error saving value.'); }
    };
    ajaxRequest('GET', 'save.php?id=' + this.id + '&value=' + this.value, callback);
},
cancel: function() {
    this.convertToText();
},
convertToText: function() {
    this.fieldElement.style.display = 'none';
    this.saveButton.style.display = 'none';
    this.cancelButton.style.display = 'none';
    this.staticElement.style.display = 'inline';

    this.setValue(this.value);
},
setValue: function(value) {
    this.fieldElement.value = value;
    this.staticElement.innerHTML = value;
},
getValue: function() {
    return this.fieldElement.value;
}
};

```

上述代码中并没有创建类，而是创建了一个对象。原型式继承不使用构造函数，所以类式继承方案中的构造函数中的代码在这个方案中被移到了一个名为configure的方法中。除此之外，这个方案中的代码与前一方案几乎一模一样。根据EditInPlaceField这个原型对象创建新对象的方式与对类进行实例化大不相同：

```
var titlePrototypal = clone(EditInPlaceField);
titlePrototypal.configure('titlePrototypal', $('#doc'), 'Title Here');
var currentTitleText = titlePrototypal.getValue();
```

这里不再使用new运算符，而是使用clone函数来创建一个对象副本，然后再对这个副本进行配置。此后就可以象对待前面的titleClassical对象一样与titlePrototypal这个对象打交道了。这两个对象基本上没有什么分别，你可以用同样的API来处理它们。

创建这个对象的子对象也要用到clone函数：

```
/* EditInPlaceArea object. */

var EditInPlaceArea = clone(EditInPlaceField);

// Override certain methods.

EditInPlaceArea.createElements = function(id) {
    this.containerElement = document.createElement('div');
    this.parentElement.appendChild(this.containerElement);

    this.staticElement = document.createElement('p');
    this.containerElement.appendChild(this.staticElement);
    this.staticElement.innerHTML = this.value;

    this.fieldElement = document.createElement('textare');
    this.fieldElement.value = this.value;
    this.containerElement.appendChild(this.fieldElement);

    this.saveButton = document.createElement('input');
    this.saveButton.type = 'button';
    this.saveButton.value = 'Save';
    this.containerElement.appendChild(this.saveButton);

    this.cancelButton = document.createElement('input');
    this.cancelButton.type = 'button';
    this.cancelButton.value = 'Cancel';
    this.containerElement.appendChild(this.cancelButton);

    this.convertToText();
};

EditInPlaceArea.convertToEditable = function() {
    this.staticElement.style.display = 'none';
    this.fieldElement.style.display = 'block';
    this.saveButton.style.display = 'inline';
    this.cancelButton.style.display = 'inline';
}
```

```

        this.setValue(this.value);
    };
EditInPlaceArea.convertToText = function() {
    this.fieldElement.style.display = 'none';
    this.saveButton.style.display = 'none';
    this.cancelButton.style.display = 'none';
    this.staticElement.style.display = 'block';

    this.setValue(this.value);
};

```

上述代码只是先创建EditInPlaceField对象的一个副本，然后改写其中的一些方法。EditInPlaceArea这个原型对象可以像第一个原型对象<sup>①</sup>一样使用和克隆。实际上，你还可以如法炮制，在此基础上创建新的原型对象，为此只需对其进行克隆并在新对象中进行一些修改即可。

原型式继承技术看来也很适合这个案例，其原因与类式继承类似。这两种方案唯一的差别在于创建类（对象）以及从它们派生子对象（实例）的方式。大多数代码（包括所有方法）在两种方案中完全相同。由此可见从其中一种范型转到别一种范型有多么容易。不是所有的转变都有这样轻松，特别是在涉及具有大量数组或对象类型的成员的类和对象时，但通常你只需在语法方面进行一些调整即可。

在这个案例中，原型式继承相对于类式继承并未表现出什么优越性。其中的对象都没有用到多少默认值，因此新方案的使用没有节省什么内存。我们很难说哪种方案更好，它们在此例中难分伯仲。

58

### 4.7.3 掺元类解决方案

这次我们要使用掺元类来处理这个问题。我们首先创建一个包含了所有要共享的方法的掺元类，然后再创建一个新类，并使用augment函数来让这个新类共享到那些方法。

```

/* Mixin class for the edit-in-place methods. */

var EditInPlaceMixin = function() {};
EditInPlaceMixin.prototype = {
    createElements: function(id) {
        this.containerElement = document.createElement('div');
        this.parentElement.appendChild(this.containerElement);

        this.staticElement = document.createElement('span');
        this.containerElement.appendChild(this.staticElement);
        this.staticElement.innerHTML = this.value;

        this.fieldElement = document.createElement('input');
        this.fieldElement.type = 'text';
        this.fieldElement.value = this.value;
        this.containerElement.appendChild(this.fieldElement);
    }
};

```

<sup>①</sup> 指EditInPlaceField。——译者注

```

this.saveButton = document.createElement('input');
this.saveButton.type = 'button';
this.saveButton.value = 'Save';
this.containerElement.appendChild(this.saveButton);

this.cancelButton = document.createElement('input');
this.cancelButton.type = 'button';
this.cancelButton.value = 'Cancel';
this.containerElement.appendChild(this.cancelButton);

this.convertToText();
},
attachEvents: function() {
  var that = this;
  addEvent(this.staticElement, 'click', function() { that.convertToEditable(); });
  addEvent(this.saveButton, 'click', function() { that.save(); });
  addEvent(this.cancelButton, 'click', function() { that.cancel(); });
},
convertToEditable: function() {
  this.staticElement.style.display = 'none';
  this.fieldElement.style.display = 'inline';
  this.saveButton.style.display = 'inline';
  this.cancelButton.style.display = 'inline';
  this.setValue(this.value);
},
save: function() {
  this.value = this.getValue();
  var that = this;
  var callback = {
    success: function() { that.convertToText(); },
    failure: function() { alert('Error saving value.'); }
  };
  ajaxRequest('GET', 'save.php?id=' + this.id + '&value=' + this.value, callback);
},
cancel: function() {
  this.convertToText();
},
convertToText: function() {
  this.fieldElement.style.display = 'none';
  this.saveButton.style.display = 'none';
  this.cancelButton.style.display = 'none';
  this.staticElement.style.display = 'inline';

  this.setValue(this.value);
},
setValue: function(value) {
  this.fieldElement.value = value;
}

```

```

    this.staticElement.innerHTML = value;
},
getValue: function() {
    return this.fieldElement.value;
}
};

```

这个掺元类只定义了一些方法。要创建一个职能类，需要先创建一个构造函数，然后再调用 augment：

```

/* EditInPlaceField class. */

function EditInPlaceField(id, parent, value) {
    this.id = id;
    this.value = value || 'default value';
    this.parentElement = parent;

    this.createElements(this.id);
    this.attachEvents();
};

augment(EditInPlaceField, EditInPlaceMixin);

```

60

随后即可像类式继承方案中那样对这个类进行实例化。要创建使用多行文本框的类，你不是从EditInPlaceField派生子类，而是另行创建一个新类（的构造函数）并用同样的掺元类扩充它。但是在扩充这个新类之前，先要为其定义一些方法。由于它们是在扩充之前定义的，所以不会被覆盖。

```

/* EditInPlaceArea class. */

function EditInPlaceArea(id, parent, value) {
    this.id = id;
    this.value = value || 'default value';
    this.parentElement = parent;

    this.createElements(this.id);
    this.attachEvents();
};

// Add certain methods so that augment won't include them.

EditInPlaceArea.prototype.createElements = function(id) {
    this.containerElement = document.createElement('div');
    this.parentElement.appendChild(this.containerElement);

    this.staticElement = document.createElement('p');
    this.containerElement.appendChild(this.staticElement);
    this.staticElement.innerHTML = this.value;

    this.fieldElement = document.createElement('textare');
    this.fieldElement.value = this.value;
}

```

```

this.containerElement.appendChild(this.fieldElement);

this.saveButton = document.createElement('input');
this.saveButton.type = 'button';
this.saveButton.value = 'Save';
this.containerElement.appendChild(this.saveButton);

this.cancelButton = document.createElement('input');
this.cancelButton.type = 'button';
this.cancelButton.value = 'Cancel';
this.containerElement.appendChild(this.cancelButton);

this.convertToText();

};

EditInPlaceArea.prototype.convertToEditable = function() {
    this.staticElement.style.display = 'none';
    this.fieldElement.style.display = 'block';
    this.saveButton.style.display = 'inline';
    this.cancelButton.style.display = 'inline';
    this.setValue(this.value);
};

EditInPlaceArea.prototype.convertToText = function() {
    this.fieldElement.style.display = 'none';
    this.saveButton.style.display = 'none';
    this.cancelButton.style.display = 'none';
    this.staticElement.style.display = 'block';

    this.setValue(this.value);
};

augment(EditInPlaceArea, EditInPlaceMixin);

```

这个案例中可以使用掺元类技术，但是这种方案不如前面两种。虽然使用这些技术最终创建的对象大体相同，但是从条理性的角度来看，严格的继承方案比扩充方案更加清楚。掺元类非常适合于组织那些彼此迥然不同的类所共享的方法。但本例中的掺元类却是在为两个非常相似的类提供所有方法。前面两个方案中的代码要更容易维护一些，因为开发者一眼就能看出那些方法的来源以及类和对象的组织方式。

实现可以用在各类对象中的通用方法的共享才是掺元类如鱼得水的领域。这方面的例子包括将对象序列化为字符串形式的方法和输出对象的状态以供调试之用的方法。掺元类还可以用来模仿其他面向对象语言中的枚举或迭代器。

## 4.8 继承的适用场合

继承会使代码变得更加复杂、更难被JavaScript新手理解，所以只应该用在其带来的好处胜过缺点的那些场合。它的主要好处表现在代码的重用方面。通过建立类或对象之间的继承关系，有些方法我们只需要定义一次即可。同样，如果需要修改这些方法或排查其中的错误，那么由于其

定义只出现在一个位置，所以非常有利于节省时间和精力。

各种继承范型都有自己的优缺点。在内存效率比较重要的场合原型式继承（及clone函数）是最佳选择。如果与对象打交道的都是些只熟悉其他面向对象语言中的继承机制的程序员，那么最好使用类式继承（及extend函数）。这两种方法都适合于类间差异较小的类层次体系（hierarchy）。如果类之间的差异较大，那么用掺元类中的方法来扩充这些类往往是一种更合理的选择。

比较简单的JavaScript程序很少需要用到这种程度的抽象。只有那些有许多程序员参与的大型项目才需要这种代码组织手段。

## 4.9 小结

本章讨论了继承的优点和缺点，以及让一个类或对象继承另一个类或对象的三种方法。类式继承试图模仿C++和Java等面向对象语言中类的继承方式。它最适合于内存效率要求不高或程序员不熟悉那种不太知名的原型式继承的场合。子类派生过程中大多数令人困惑的步骤可以用extend函数封装起来。

原型式继承的工作机制是先创建一些对象然后再对其进行克隆，从而得到创建子类和实例的等效结果。一旦你明白了其中的道理，这种继承方式用起来会非常顺手，而且用这种办法创建的对象往往有较高的内存效率，这是因为它们会共享那些未被改写的属性和方法。那些包含着数组或对象类型成员的对象的克隆会有一些麻烦之处，但这个问题可以通过使用一个方法来设置那些属性的默认值加以解决。创建一个克隆对象的所有事宜由clone函数负责处理。

掺元类提供了一条既能让对象和类共享一些方法又不需要让它们结成父子关系的途径。如果你想让各种彼此有着较大差异的类共享一些通用方法，那么这正是掺元类的用武之地。augment函数允许你选择共享掺元类中的全部方法还是部分方法。

使用这三种技术可以创建出复杂的对象层次体系，其简练性堪与任何别的面向对象语言媲美。对于编程新手来说，JavaScript中的继承不那么好懂或直观。它是一种通过研究这种语言的底层特性而发展起来的高级技术，但是它可以通过使用几个便利函数而得以简化。这种技术非常适合于创建供其他程序员使用的API。

**单**体 (singleton) 模式是JavaScript中最基本但又最有用的模式之一，它可能比其他任何模式都更常用。这种模式提供了一种将代码组织为一个逻辑单元的手段，这个逻辑单元中的代码可以通过单一的变量进行访问。通过确保单体对象只存在一份实例，你就可以确信自己的所有代码使用的都是同样的全局资源。

单体类在JavaScript中有许多用途。它们可以用来划分命名空间，以减少网页中全局变量的数目。它们还可以在一种名为分支 (branching) 的技术中用来封装浏览器之间的差异（借助分支技术，你在使用各种常用的工具函数时就不必再操心浏览器嗅探的事）。更重要的是，借助于单体模式，你可以把代码组织得更为一致，从而使其更容易阅读和维护。

这种模式在JavaScript中非常重要，也许比在其他任何语言中都更重要。在网页上使用全局变量有很大的风险，而用单体对象创建的命名空间则是清除这些全局变量的最佳手段之一。仅此一个原因你就该掌握这种模式，更别说它还有许多别的用途。本章将介绍其中最重要的一些用途。

## 5.1 单体的基本结构

较高级的单体模式我们还是放到本章后面再讲，这里先讨论最基本的类型。最简单的单体实际上就是一个对象字面量，它把一批有一定关联的方法和属性组织在一起：

```
/* Basic Singleton. */

var Singleton = {
    attribute1: true,
    attribute2: 10,

    method1: function() {

    },
    method2: function(arg) {

    }
};
```

在这个示例中，所有那些成员现在都可以通过变量Singleton来访问。为此可以使用圆点运算符：

```
Singleton.attribute1 = false;
var total = Singleton.attribute2 + 5;
var result = Singleton.method1();
```

这个单体对象可以被修改。你可以为其添加新成员，这一点与别的对象字面量没什么不同。你也可以用`delete`运算符删除其现有成员。这实际上违背了面向对象设计的一条原则：类可以被扩展，但不应该被修改。JavaScript中的所有对象都是易变的，这正是它与C++和Java等别的面向对象语言的区别之一。你不必为此忧心忡忡（Python、Ruby和Smalltalk都允许在定义了类之后又对其进行修改），但是你应该清楚在这种语言中无法阻止对象的修改。如果某些变量需要保护，那么你可以像第3章所演示的那样将其定义在闭包之中。

你可能还是没发觉这种单体对象与普通对象字面量有什么不同。按传统的定义，单体是一个只能被实例化一次并且可以通过一个众所周知的访问点访问的类。要是严格地按这个定义来说，前面的例子所示的并不是一个单体，因为它不是一个可实例化的类。我们打算把单体模式定义得更广义一些：单体是一个用来划分命名空间并将一批相关方法和属性组织在一起的对象，如果它可以被实例化，那么它只能被实例化一次。

对象字面量只是用以创建单体的方法之一。本章后面要介绍的那些方法所创建的单体看起来才更像其他面向对象语言中的单体类。另外，并非所有对象字面量都是单体。如果它只是用来模仿关联数组或容纳数据的话，那就显然不是单体。但如果它是用来组织一批相关方法和属性的话，那就可能是单体。其区别主要在于设计者的意图。

## 5.2 划分命名空间

单体对象由两个部分组成：包含着方法和属性成员的对象自身，以及用于访问它的变量。这个变量通常是全局性的，以便在网页上任何地方都能直接访问到它所指向的单体对象。这是单体模式的一个要点。虽然按定义单体不必是全局性的，但它应该在各个地方都能被访问。因为单体对象的所有内部成员都被包装在这个对象中，所以它们不是全局性的。由于这些成员只能通过这个单体对象变量进行访问，因此在某种意义上，可以说它们被单体对象圈在了一个命名空间中。

命名空间是可靠的JavaScript编程的一个重要工具。在JavaScript中什么都可以被改写，程序员一不留神就会擦除一个变量、函数甚至整个类，而自己却毫无察觉。这种错误查找起来非常费时：

```
/* Declared globally. */

function findProduct(id) {
  ...
}

...

// Later in your page, another programmer adds...
var resetProduct = $('#reset-product-button');
var findProduct = $('#find-product-button'); // The findProduct function just got
                                             // overwritten.
```

有个问题虽然与这个例子没有直接关系，但却值得提一句。函数中声明变量时使用的var关键字很重要，如果不使用它，那么变量将被声明为全局性的，因此更容易干扰到全局命名空间中的其他代码。

现在回头来看前面的例子。为了避免无意中改写变量，最好的解决办法之一是用单体对象将代码组织在命名空间之中。下面是前面的例子用单体模式改良后的结果：

```
/* Using a namespace. */

var MyNamespace = {
    findProduct: function(id) {
        ...
    },
    // Other methods can go here as well.
}

...

// Later in your page, another programmer adds...
var resetProduct = $('reset-product-button');
var findProduct = $('find-product-button'); // Nothing was overwritten.
```

现在findProduct函数是MyNamespace中的一个方法，它不会被全局命名空间中声明的任何新变量改写。要注意，该方法仍然可以从各个地方访问。不同之处在于现在其调用方式不是findProduct(id)，而是MyNamespace.findProduct(id)。还有一个好处就是，这可以让其他程序员大体知道这个方法的声明地点及其作用。用命名空间把类似的方法组织到一起，也有助于增强代码的文档性<sup>①</sup>。

**注解** MyNamespace是个糟糕的单体名字，我们在此只是用它来表示这个对象字面量起着命名空间的作用。命名空间的名字应该能够说明其中包含的代码的用途。在本例中，ProductTools这个名字要更恰当一点。

命名空间还可以进一步分割。现在网页上的JavaScript代码往往不止有一个来源。其中除了你写的代码外，还会有库代码、广告代码和徽章代码。这些变量都出现在网页的全局命名空间中。为了避免冲突，可以定义一个用来包含自己的所有代码的全局对象：

```
/* GiantCorp namespace. */
var GiantCorp = {};
```

然后可以分门别类地把自己的代码和数据组织到这个全局对象中的各个对象（单体）中：

```
GiantCorp.Common = {
    // A singleton with common methods used by all objects and modules.
};
```

<sup>①</sup> 原文为“document your code”。意思是说，命名空间的使用有助于让用户了解代码的组织结构及其各部分的用途，这实际上就是提供了关于代码的说明信息。

```

GiantCorp.ErrorCodes = {
    // An object literal used to store data.
};

GiantCorp.PageHandler = {
    // A singleton with page specific methods and attributes.
};

```

来源于外部的代码与GiantCorp这个变量发生冲突的可能性很小。如果真有冲突，其造成的问题会非常明显，所以很容易被发现。想到自己办事牢靠，没有把全局命名空间搞得一片狼藉，你大可高枕无忧。你只是在全局命名空间中加入了一个变量，这是一个JavaScript程序员可望获得的最小地盘。

### 5.3 用作特定网页专用代码的包装器的单体

你已经了解了如何把单体作为命名空间使用，现在我们再介绍单体模式的一个特殊用途。在那种拥有许多网页的网站中，有些JavaScript代码是所有网页都要用到的，它们通常被存放在独立的文件中；而有些代码则是某个网页专用的，不会被用到其他地方。最好把这两种代码分别包装在自己的单体对象中。

用来包装各个网页专用的代码的单体通常看起来都差不多。它需要封装一些数据（也许是作为常量）、为各网页特有的行为定义一些方法以及定义初始化方法。涉及DOM中特有元素的大多数代码，比如添加事件监听器的代码，只有在这些元素加载之后才能工作。你可以通过创建一个init方法并将其关联到窗口的load事件（或类似的其他事件，比如由此派生出来的DOMContentLoaded事件或DOMContentLoaded事件<sup>①</sup>），将所有这些初始化代码组织到一个地方。

下面是用来包装特定网页专用代码的单体的骨架：

```

/* Generic Page Object. */

Namespace.PageName = {

    // Page constants.
    CONSTANT_1: true,
    CONSTANT_2: 10,

    // Page methods.
    method1: function() {
    },
    method2: function() {
    },

    // Initialization method.
    init: function() {
    }
};

```

<sup>①</sup> 详情参见<http://peter.michaux.ca/article/553>。

```

    }
}

```

```

// Invoke the initialization method after the page loads.
addLoadEvent(Namespace.PageName.init);

```

我们用一个Web开发中很常见的任务为例示范一下它的用法。我们经常想要用JavaScript为表单添加功能。出于平稳退化方面的考虑，通常先创建一个不依赖于JavaScript的、使用普通提交机制完成任务的纯HTML网页。然后再用JavaScript控制表单的行为，以提供额外的特性。

下面的单体会查找并劫持（hijack）一个特定的表单：

```

/* RegPage singleton, page handler object. */

GiantCorp.RegPage = {

    // Constants.
    FORM_ID: 'reg-form',
    OUTPUT_ID: 'reg-results',

    // Form handling methods.
    handleSubmit: function(e) {
        e.preventDefault(); // Stop the normal form submission.

        var data = {};
        var inputs = GiantCorp.RegPage.formEl.getElementsByTagName('input');

        // Collect the values of the input fields in the form.
        for(var i = 0, len = inputs.length; i < len; i++) {
            data[inputs[i].name] = inputs[i].value;
        }

        // Send the form values back to the server.
        GiantCorp.RegPage.sendRegistration(data);
    },
    sendRegistration: function(data) {
        // Make an XHR request and call displayResult() when the response is
        // received.
        ...
    },
    displayResult: function(response) {
        // Output the response directly into the output element. We are
        // assuming the server will send back formatted HTML.
        GiantCorp.RegPage.outputEl.innerHTML = response;
    },

    // Initialization method.
    init: function() {
        // Get the form and output elements.
        GiantCorp.RegPage.formEl = $(GiantCorp.RegPage.FORM_ID);
    }
};

```

```

GiantCorp.RegPage.outputEl = $(GiantCorp.RegPage.OUTPUT_ID);

// Hijack the form submission.
addEvent(GiantCorp.RegPage.formEl, 'submit', GiantCorp.RegPage.handleSubmit);
}

};

// Invoke the initialization method after the page loads.
addLoadEvent(GiantCorp.RegPage.init);

```

上述代码中首先假定GiantCorp命名空间已经作为一个空的对象字面量被创建好了。如若不然，代码的第一行就会引发一个错误。下面这行代码可以防止这种错误，如果GiantCorp还不存在，它就会定义这个对象，其中使用的逻辑“或”运算符可以在未找到一个属性时为其提供一个默认值：

```
var GiantCorp = window.GiantCorp || {};
```

在前面的例子中，我们把所关注的两个HTML元素的ID值作为常量保存起来，这是因为在程序执行期间它们不会发生变化。

其中的初始化方法先获取那两个HTML元素的引用，然后把它们作为单体的新属性保存起来。这没问题，你可以在运行时添加或删除单体的成员。这个方法还把一个方法关联到表单的submit事件。此后当表单被提交时，其正常行为会被阻止（这是e.preventDefault()的作用），取而代之的是收集表单数据并用Ajax方式将其发回服务器的操作。

## 5.4 拥有私用成员的单体

我们在第3章中讨论过几种创建类的私用成员的做法。使用真正的私用方法的一个缺点在于它们比较耗费内存，因为每个实例都具有方法的一份新副本。不过，由于单体对象只会被实例化一次，因此为其定义真正的私用方法时不必顾虑内存方面的问题。尽管如此，创建伪私用成员还是更容易一些，所以我们先谈谈这种做法。

### 5.4.1 使用下划线表示法

70

在单体对象内创建私用成员最简单、最直截了当的办法是使用下划线表示法。这可以让其他程序员知道相关方法或属性是私用的，只在对象内部使用。在单体对象中使用下划线表示法是一种告诫其他程序员不要直接访问特定成员的简明办法：

```

/* DataParser singleton, converts character delimited strings into arrays. */

GiantCorp.DataParser = {
    // Private methods.
    _stripWhitespace: function(str) {
        return str.replace(/\s+/, '');
    },
    _stringSplit: function(str, delimiter) {

```

```

        return str.split(delimiter);
    },

    // Public method.
    stringToArray: function(str, delimiter, stripWS) {
        if(stripWS) {
            str = this._stripWhitespace(str);
        }
        var outputArray = this._stringSplit(str, delimiter);
        return outputArray;
    };
}

```

这个例子中的单体对象有一个公用方法`stringToArray`。该方法的参数包括一个字符串、一个分隔符，以及一个用以指示是否要删除所有空白字符的可选的布尔值。它的工作主要靠`_stripWhitespace`和`_stringSplit`这两个私用方法完成。这两个方法不是该单体有记载的接口的一部分，以后更新时不见得还会存在，所以它们不应该被公开。将它们设计为私用方法，重构所有内部代码时就不必担心会殃及别人的程序。比如说，后来你检查了一下这个对象，觉得`_stringSplit`没有必要作为一个单独的函数存在。你可以将其彻底删除，而且因为这是用下划线标记的私用方法，可以确信没人会直接调用它（如果真有这样的人，他们活该遭受报应）。

`stringToArray`方法中用`this`访问单体中的其他方法。这是访问单体中其他成员的最简便的做法。但这样做也有一点风险，因为`this`并不一定就指向`GiantCorp.DataParser`。例如，如果把某个方法用作事件监听器，那么其中的`this`可能会指向`window`对象，这意味着`_stripWhitespace`和`_stringSplit`这两个方法不会被找到。虽然大多数JavaScript库都会为事件关联进行作用域校正，但还是使用全名`GiantCorp.DataParser`访问单体内的其他成员更保险一点。

## 5.4.2 使用闭包

在单体对象中创建私用成员的第二种办法需要借助闭包。这与第3章中创建真正的私用成员的做法非常相似，但也有一个重要的区别。先前的做法是把变量和函数定义在构造函数体内（不使用`this`关键字）以使其成为私用成员，此外还在构造函数体内定义了所有的特权方法并用`this`关键字使其可被外界访问。每生成一个该类的实例时，所有声明在构造函数内的方法和属性都会再次创建一份。这可能会非常低效。

71

因为单体只会被实例化一次，所以你不用担心自己在构造函数中声明了多少成员。每个方法和属性都只会被创建一次，所以你可以把它们都声明在构造函数内部（因此也就位于同一个闭包中）。先前你所见的单体都是这样的对象字面量：

```
/* Singleton as an Object Literal. */
```

```
MyNamespace.Singleton = {};
```

现在我们用一个在定义之后立即执行的函数创建单体：

```
/* Singleton with Private Members, step 1. */
```

```
MyNamespace.Singleton = function() {
    return {};
}();
```

上述两个例子中所创建的两个MyNamespace.Singleton完全相同。要注意在第二个例子中并没有把一个函数赋给MyNamespace.Singleton。那个匿名函数返回一个对象，而赋给MyNamespace.Singleton变量的正是这个对象。为了立即执行这个匿名函数，只需在其定义的最后那个大括号后面放上一对圆括号即可。

有些程序员喜欢在那个匿名函数定义之外再套上一对圆括号，以表示它会在声明之后立即执行。这在所创建的单体较为庞大时尤其有用，因为你只要瞟一眼就能看出该函数只是用来创建一个闭包。额外加上这对圆括号后，前面创建单体那个例子就变成下面的样子：

```
/* Singleton with Private Members, step 1. */

MyNamespace.Singleton = (function() {
    return {};
})();
```

你可以像以前那样把公用成员添加到作为单体返回的那个对象字面量中：

```
/* Singleton with Private Members, step 2. */

MyNamespace.Singleton = (function() {
    return { // Public members.
        publicAttribute1: true,
        publicAttribute2: 10,
        publicMethod1: function() {
            ...
        },
        publicMethod2: function(args) {
            ...
        }
    };
})();
```

72

要是这样得到的结果与直接使用一个对象字面量没什么区别，那又何必劳神加上那层函数包装呢？原因在于这个包装函数创建了一个可以用来添加真正的私用成员的闭包。任何声明在这个匿名函数中（但不是在那个对象字面量中）的变量或函数都只能被在同一个闭包中声明的其他函数访问。这个闭包在匿名函数执行结束后依然存在，所以在其中声明的函数和变量总能从匿名函数所返回的对象内部（并且也只能从内部）访问。

下面的代码示范了在那个匿名函数中添加私用成员的做法：

```
/* Singleton with Private Members, step 3. */

MyNamespace.Singleton = (function() {
    // Private members.
    var privateAttribute1 = false;
```

```

var privateAttribute2 = [1, 2, 3];

function privateMethod1() {
    ...
}

function privateMethod2(args) {
    ...
}

return { // Public members.
    publicAttribute1: true,
    publicAttribute2: 10,

    publicMethod1: function() {
        ...
    },
    publicMethod2: function(args) {
        ...
    };
}();

```

这种单体模式又称模块模式（module pattern）<sup>①</sup>，指的是它可以把一批相关方法和属性组织为模块并起到划分命名空间的作用。

73

### 5.4.3 两种技术的比较

现在回到DataParser这个例子中来，看看如何在其实现中使用真正的私用成员。现在我们不再为每个私用方法名称的开头添加一个下划线，而是把这些方法定义在闭包中：

```

/* DataParser singleton, converts character delimited strings into arrays. */
/* Now using true private methods. */

GiantCorp.DataParser = (function() {
    // Private attributes.
    var whitespaceRegex = /\s+/;

    // Private methods.
    function stripWhitespace(str) {
        return str.replace(whitespaceRegex, '');
    }
    function stringSplit(str, delimiter) {
        return str.split(delimiter);
    }

    // Everything returned in the object literal is public, but can access the
    // members in the closure created above.

```

<sup>①</sup> 详情参见<http://yuiblog.com/blog/2007/06/12/module-pattern/>。

```

return {
  // Public method.
  stringToArray: function(str, delimiter, stripWS) {
    if(stripWS) {
      str = stripWhitespace(str);
    }
    var outputArray = stringSplit(str, delimiter);
    return outputArray;
  }
};

})(); // Invoke the function and assign the returned object literal to
// GiantCorp.DataParser.

```

现在这些私用方法和属性可以直接用其名称访问，不必在其前面加上“`this.`”或“`GiantCorp.DataParser.`”，这些前缀只用于访问单体对象的公用成员。

这种模式与使用下划线表示法的模式相比有几点优势。把私用成员放到闭包中可以确保其不会在单体对象之外被使用。你可以自由地改变对象的实现细节，这不会殃及别人的代码。还可以用这种办法对数据进行保护和封装（尽管单体很少被这样用，除非那些数据只能被保存在一个地方）。

在使用这种模式时，你可以享受到真正的私用成员带来的所有好处，而不必付出什么代价，这是因为单体类只会被实例化一次。单体模式之所以是JavaScript中最流行、应用最广泛的模式之一，原因即在于此。

74

**警告** 要记住单体的公用成员和私用成员的声明语法有所不同，前者被声明在对象字面量内部而后者并非如此。私用属性必须用`var`声明，否则它将成为全局性的。私用方法则按`function funcName(args) {...}`这样的形式声明，在最后一个大括号之后不需要使用分号。公用属性和方法分别按`attributeName: attributeValue`和`methodName: function(args) {...}`这样的形式声明，如果后面还要声明别的成员的话，那么该声明的后面应该加上一个逗号。

## 5.5 惰性实例化

前面所讲的单体模式的各种实现方式有一个共同点：单体对象都是在脚本加载时被创建出来。对于资源密集型的或配置开销甚大的单体，也许更合理的做法是将其实例化推迟到需要使用它的时候。这种技术被称为惰性加载（lazy loading），它最常用于那些必须加载大量数据的单体。而那些被用作命名空间、特定网页专用代码包装器或组织相关实用方法的工具的单体最好还是立即实例化。

这种惰性加载单体的特别之处在于，对它们的访问必须借助于一个静态方法。应该这样调用其方法：`Singleton.getInstance().methodName()`，而不是这样调用：`Singleton.methodName()`。`getInstance`方法会检查该单体是否已经被实例化。如果还没有，那么它将创建并返回其实例。

如果单体已经实例化过，那么它将返回现有实例。下面我们从前面那个拥有真正的私用成员的单体的基本框架出发示范一下如何把普通单体转化为惰性加载单体：

```
/* Singleton with Private Members, step 3. */

MyNamespace.Singleton = (function() {
    // Private members.
    var privateAttribute1 = false;
    var privateAttribute2 = [1, 2, 3];

    function privateMethod1() {
        ...
    }
    function privateMethod2(args) {
        ...
    }

    return { // Public members.
        publicAttribute1: true,
        publicAttribute2: 10,

        publicMethod1: function() {
            ...
        },
        publicMethod2: function(args) {
            ...
        }
    };
})();
```

75

这段代码还没有进行任何修改。转化工作的第一步是把单体的所有代码移到一个名为 constructor 的方法中：

```
/* General skeleton for a lazy loading singleton, step 1. */

MyNamespace.Singleton = (function() {

    function constructor() { // All of the normal singleton code goes here.
        // Private members.
        var privateAttribute1 = false;
        var privateAttribute2 = [1, 2, 3];

        function privateMethod1() {
            ...
        }
        function privateMethod2(args) {
            ...
        }

        return { // Public members.
```

```

    publicAttribute1: true,
    publicAttribute2: 10,
    ...
    publicMethod1: function() {
        ...
    },
    publicMethod2: function(args) {
        ...
    }
}
})();

```

这个方法不能从闭包外部访问，这是件好事，因为我们想全权控制其调用时机。公用方法getInstance就是用来实现这种控制的。为了使其成为公用方法，只需将其放到一个对象字面量中并返回该对象即可：

*/\* General skeleton for a lazy loading singleton, step 2. \*/*

76

```

MyNamespace.Singleton = (function() {
    function constructor() { // All of the normal singleton code goes here.
        ...
    }
}
```
```

```

    return {
        getInstance: function() {
            // Control code goes here.
        }
    }
})
```
```

现在开始编写用于控制单体类实例化时机的代码。它需要做两件事。第一，它必须知道该类是否已经被实例化过。第二，如果该类已经实例化过，那么它需要掌握其实例的情况，以便能返回这个实例。办这两件事需要用到一个私用属性和已有的私用方法constructor：

*/\* General skeleton for a lazy loading singleton, step 3. \*/*

```

MyNamespace.Singleton = (function() {
    var uniqueInstance; // Private attribute that holds the single instance.

    function constructor() { // All of the normal singleton code goes here.
        ...
    }

    return {
        getInstance: function() {
            if(!uniqueInstance) { // Instantiate only if the instance doesn't exist.
                uniqueInstance = constructor();
            }
            return uniqueInstance;
        }
    }
})
```
```

```

    }
    return uniqueInstance;
}
})();

```

把一个单体转化为惰性加载单体后，你必须对调用它的代码进行修改。在本例中，像这样的方法调用：

```
MyNamespace.Singleton.publicMethod1();
```

应该被改为下面的形式：

```
MyNamespace.Singleton.getInstance().publicMethod1();
```

惰性加载单体的缺点之一在于其复杂性。用于创建这种类型的单体的代码并不直观，而且不易理解（不过良好的文档可以提供帮助）。如果你需要创建一个延迟实例化的单体，那么最好为其编写一条注释解释这样做的原因，以免别人把它简化为普通单体。

顺便提一句，如果觉得命名空间名称太长，可以创建一个别名来简化它。这种别名只不过是一个保存了对特定对象的引用的变量。在本例中，可以把MyNamespace.Singleton简化为MNS：

```
var MNS = MyNamespace.Singleton;
```

这样做会创建一个全局变量，所以最好还是把它声明在一个特定网页专用代码包装器单体中。在存在单体嵌套的情况下，会出现一些作用域方面的问题。在这种场合下访问其他成员最好使用完全限定名（比如GiantCorp.SingletonName）而不是this。

## 5.6 分支

分支 (branching) 是一种用来把浏览器间的差异封装到在运行期间进行设置的动态方法中的技术。举个例来说，假设我们需要创建一个返回XHR对象的方法。这种XHR对象在大多数浏览器中是XMLHttpRequest类的实例，而在IE早期版本中则是某种ActiveX类的实例。这样一个方法通常会进行某种浏览器嗅探或对象探测。如果不使用分支技术，那么每次调用这个方法时，所有那些浏览器嗅探代码都要再次运行。要是这个方法的调用很频繁，那么这样做会严重缺乏效率。

更有效做法是只在脚本加载时一次性地确定针对特定浏览器的代码。这样一来，在初始化完成之后，每种浏览器都只会执行针对它的JavaScript实现而设计的代码。能够在运行时动态确定函数代码的能力，正是JavaScript的高度灵活性和强大表现能力的一种体现。这种类型的优化很容易理解，它能提高调用这些函数的效率。

你可能一时还难以明白分支这个话题与单体模式有什么关系。在前面所讲的三种模式中，单体对象的所有代码都是在运行时确定的。这在用闭包创建私用成员的模式中最容易看出来：

```
MyNamespace.Singleton = (function() {
    return {};
})();
```

那个匿名函数在运行时得以执行，其返回的对象字面量被赋给MyNamespace.Singleton 变量。

我们可以创建两个不同的对象字面量，并根据某种条件将其中之一赋给那个变量：

```
/* Branching Singleton (skeleton). */

MyNamespace.Singleton = (function() {
    var objectA = {
        method1: function() {
            ...
        },
        method2: function() {
            ...
        }
    };
    var objectB = {
        method1: function() {
            ...
        },
        method2: function() {
            ...
        }
    };

    return (someCondition) ? objectA : objectB;
})();
```

78

上述代码中创建了两个对象字面量，它们拥有相同的一套方法。对于使用这个单体的程序员来说，赋给MyNamespace.Singleton究竟是哪个对象无关紧要，因为这两个对象实现了同样的接口，可以执行同样的任务，不同之处仅仅在于对象的方法具体使用的代码。你并不是只能使用两个分支，只要有理由，你也可以创建具有三四个分支的单体。据以在分支中进行选择的条件值在运行时进行确定。这种条件通常是某种能力检测的结果，意在确保运行代码的JavaScript环境实现了所需要的特性。如若不然，则将改而使用应变代码（fallback code）。

分支技术并不总是更高效的选择。在前面的例子中，有两个对象（objectA和objectB）被创建出来并保存在内存中，但派上用场的只有一个。在考虑是否使用这种技术的时候，你必须在缩短计算时间（因为判断该使用哪个对象的代码只会执行一次）和占用更多内存这一利一弊之间权衡一下。下一个例子就属于适合采用分支技术的情况，因为其中的分支对象较小而判断使用哪个对象的开销较大。

## 5.7 示例：用分支技术创建 XHR 对象

在本例中我们要创建一个单体，它有一个用来生成XHR对象实例的方法（在第7章中还有一个更高级的版本）。我们首先应该判断需要多少分支。因为所能实例化的对象只有3种不同类型，所以我们需要3个分支。这些分支分别按其返回的XHR对象类型命名：

```
/* SimpleXhrFactory singleton, step 1. */

var SimpleXhrFactory = (function() {
```

```

// The three branches.
var standard = {
  createXhrObject: function() {
    return new XMLHttpRequest();
  }
};
var activeXNew = {
  createXhrObject: function() {
    return new ActiveXObject('Msxml2.XMLHTTP');
  }
};
var activeXOld = {
  createXhrObject: function() {
    return new ActiveXObject('Microsoft.XMLHTTP');
  }
};

}();

```

79

这3个分支各含一个对象字面量，它们都有一个名为createXhrObject的方法。这个方法所做的只是返回一个可以用来执行异步请求的新对象。

创建分支型单体的第2步是根据条件将3个分支中某一分支的对象赋给那个变量。其具体做法是逐一尝试每种XHR对象，直到遇到一个当前JavaScript环境所支持的对象为止：

```

/* SimpleXhrFactory singleton, step 2. */

var SimpleXhrFactory = (function() {

  // The three branches.
  var standard = {
    createXhrObject: function() {
      return new XMLHttpRequest();
    }
  };
  var activeXNew = {
    createXhrObject: function() {
      return new ActiveXObject('Msxml2.XMLHTTP');
    }
  };
  var activeXOld = {
    createXhrObject: function() {
      return new ActiveXObject('Microsoft.XMLHTTP');
    }
  };

  // To assign the branch, try each method; return whatever doesn't fail.
  var testObject;
  try {

```

```

    testObject = standard.createXhrObject(); *
    return standard; // Return this if no error was thrown.
}
catch(e) {
  try {
    testObject = activeXNew.createXhrObject();
    return activeXNew; // Return this if no error was thrown.
  }
  catch(e) {
    try {
      testObject = activeXOld.createXhrObject();
      return activeXOld; // Return this if no error was thrown.
    }
    catch(e) {
      throw new Error('No XHR object found in this environment.');
    }
  }
}
})();

```

这个单体现在就可以用来生成XHR对象的实例。使用该API的程序员只要调用SimpleXhrFactory.createXhrObject()就能得到适合特定的运行时环境的XHR对象。用了分支技术后，所有那些特性嗅探代码都只会执行一次，而不是每生成一个对象就要执行一次。

这是一种非常有效的技术，它适用于任何只有在运行时才能确定具体实现的情况。第7章讨论工厂模式时我们还会进一步研究这个话题。

## 5.8 单体模式的适用场合

从为代码提供命名空间和增强其模块性这个角度来说，你应该尽量多使用单体模式。单是JavaScript中最有用的模式之一，几乎适用于所有大大小小的项目。在简单的快餐型项目中，你可以只是把单体用作命名空间，将自己的所有代码组织在一个全局变量名下。在稍大、稍复杂一点的项目中，单体可以用来把相关代码组织在一起以便日后维护，或者用来把数据或代码安置在一个众所周知的单一位置。在大型或复杂的项目中，它可以起到优化作用：那些开销较大却又很少使用的组件可以被包装到惰性加载单体中，而针对特定环境的代码则可以被包装到分支型单体中。

很少见到有哪个项目用不到某种形式的单体模式。JavaScript的灵活性使单体可以被用于多种不同任务。可以说，这种模式在JavaScript中的重要性大大超过它在其他语言中的重要性。这主要是因为它可以用来创建命名空间以减少全局变量的数目。这种作用对于JavaScript非常重要，因为这种语言中的全局变量比其他语言中的更有危险性。网页包含的JavaScript代码往往有着五花八门的来源，其编写者形形色色，所以全局变量和函数很容易被改写，从而导致你的代码失灵。可以解决这种问题的单体模式无疑是程序员们工具箱中的一大利器。

## 5.9 单体模式之利

单体模式的主要好处在于它对代码的组织作用。把相关方法和属性组织在一个不会被多次实例化的单体中，可以使代码的调试和维护变得更轻松。描述性的命名空间还可以增强代码的自我说明性，有利于新手阅读和理解。把你的方法包裹在单体中，可以防止它们被其他程序员误改，还可以防止全局命名空间被一大堆变量弄得一团糟。单体可以把你的代码与第三方的库代码和广告代码隔离开来，从而在整体上提高网页的稳定性。

单体模式的一些高级变体可以在开发周期的后期用于对脚本进行优化，提升其性能。使用惰性实例化技术，可以直到需要一个对象的时候才创建它，从而减少那些不需要它的用户承受的不必要的内存消耗（还可能包括带宽消耗）。分支技术则可以用来创建高效的方法，不用管浏览器或环境的兼容性如何。通过根据运行时的条件确定赋给单体变量的对象字面量，你可以创建出为特定环境量身定制的方法，这种方法不会在每次调用时都一再浪费时间去检查运行环境。

## 5.10 单体模式之弊

由于单体模式提供的是一种单点访问，所以它有可能导致模块间的强耦合。这是这种模式受到的主要批评，这个批评也很中肯。有时创建一个可实例化的类更为可取，哪怕它只会被实例化一次。因为这种模式可能会导致类间的强耦合，所以它也不利于单元测试。你无法单独测试一个调用了来自单体的方法的类，而只能把它与那个单体作为一个单元一起测试。单体最好还是留给定义命名空间和实现分支型方法这些用途。在这些情况下，耦合不是什么问题。

有时某种更高级的模式会更符合任务的需要。与惰性加载单体相比，虚拟代理能给予你对类实例化方式更多的控制权。你也可以用一个真正的对象工厂来取代分支型单体（虽说这个工厂可能也是一个单体）。不要对本书中那些更特别的模式抱有畏难情绪，不要仅仅因为单体“够可以了”就选择使用它。你应该确保所选择的模式适合自己的任务。

## 5.11 小结

单体模式是JavaScript中最基本的模式之一。它不仅可以像你在本章所看到的那样单独使用，还能以这样或那样的形式与本书所讲的大多数模式配合使用。例如，对象工厂就可以被设计为单体，组合对象的所有子对象也可以被封装进一个单体命名空间中。本书讲的就是如何创建可重用的模块化代码。寻找组织和说明代码的各种方法是迈向这个目标的最重要的步骤之一。单体在这方面很有用处。把你的代码包装在一个单体中，就不必担心别人在使用它们时会改写到他们自己的全局变量，这是向创建供大众使用的API这个方向迈出的一大步。这也是成为一个值得信赖的高级JavaScript程序员所要经历的第一步。

# 方法的链式调用

**本**章研究的是JavaScript对方法进行链式调用的能力。应用程序开发人员可以使用一些简单技术来改进自己的代码编写工作。你可以写一些函数来处理各种常见任务，以节省时间；也可以改进一下代码的实现方式。最后，你可以把方法的链式调用技术用到自己所写的整个JavaScript库中，把自己喜欢的方法串连起来调用。

链式调用其实只不过是一种语法招数。它能让你通过重用一个初始操作来达到用少量代码表达复杂操作的目的。这种技术包含两个部分：一个创建代表HTML元素的对象的工厂（第7章将详细讲述工厂模式），以及一批对这个HTML元素执行某些操作的方法。每一个这种方法都可以在方法名前附上一个圆点后加入调用链中。方法的链式调用可以被视为选择网页上的一个元素并对其进行一个或多个操作的过程。

我们先看个小小例子。通过对比“之前和之后”的代码，你就能对链式调用的概念有个初步认识。这个例子使用了一些预先定义的实用函数，这些函数具体如何实现对于这个例子来说并不重要。例中我们要获取一个ID值为example的元素的引用，然后为其指派一个事件监听器。当这个元素被点击时，事件监听器会将其文本颜色设置为绿色，然后显示该元素：

```
// Without chaining:  
addEvent($('example'), 'click', function() {  
    setStyle(this, 'color', 'green');  
    show(this);  
});  
  
// With chaining:  
$('example').addEvent('click', function() {  
    $(this).setStyle('color', 'green').show();  
});
```

83

## 6.1 调用链的结构

对\$函数你已经很熟悉了。它通常返回一个HTML元素或一个HTML元素的集合，如下所示：

```
function $() {  
    var elements = [];  
    for (var i = 0, len = arguments.length; i < len; ++i) {  
        var element = arguments[i];  
        if (typeof element === 'string') {  
            element = document.getElementById(element);
```

```

    }
    if (arguments.length === 1) {
        return element;
    }
    elements.push(element);
}
return elements;
}

```

但是，如果把这个函数改造为一个构造器，把那些元素作为数组保存在一个实例属性中，并让所有定义在构造器函数的prototype属性所指对象中的方法都返回用以调用方法的那个实例的引用，那么它就具有了进行链式调用的能力。先别说得太远。我们首先需要把这个\$函数改为一个工厂方法，它负责创建支持链式调用的对象。这个函数应该能接受元素数组形式的参数，以便我们能够使用与原来一样的公用接口。修改后的代码如下所示：

```

(function() {
    // Use a private class.
    function _$(els) {
        this.elements = [];
        for (var i = 0, len = els.length; i < len; ++i) {
            var element = els[i];
            if (typeof element === 'string') {
                element = document.getElementById(element);
            }
            this.elements.push(element);
        }
    }
    // The public interface remains the same.
    window.$ = function() {
        return new _$(arguments);
    };
})();

```

84

由于所有对象都会继承其原型对象的属性和方法，所以我们可以让定义在原型对象中的那几个方法都返回用以调用方法的实例对象的引用，这样就可以对那些方法进行链式调用。想好这一点，我们现在就动手在\$\_这个私用构造函数的prototype对象中添加方法，以便实现链式调用：

```

(function() {
    function _$(els) {
        // ...
    }
    _$.prototype = {
        each: function(fn) {
            for (var i = 0, len = this.elements.length; i < len; ++i) {
                fn.call(this, this.elements[i]);
            }
            return this;
        },
        setStyle: function(prop, val) {

```

```

        this.each(function(el) {
            el.style[prop] = val;
        });
        return this;
    },
    show: function() {
        var that = this;
        this.each(function(el) {
            that.setStyle('display', 'block');
        });
        return this;
    },
    addEvent: function(type, fn) {
        var add = function(el) {
            if (window.addEventListener) {
                el.addEventListener(type, fn, false);
            }
            else if (window.attachEvent) {
                el.attachEvent('on'+type, fn);
            }
        };
        this.each(function(el) {
            add(el);
        });
        return this;
    }
};
window.$ = function() {
    return new _$(arguments);
};
}();

```

85

看看该类的每一行方法的最后一行，你会发现它们都以“return this;”结束。这会将用以调用方法的对象传给调用链上的下一个方法。支持链式调用的接口带来的可能性是无穷的。现在你可以像这样编写代码：

```

$(window).addEvent('load', function() {
    $('#test-1', '#test-2').show().
        setStyle('color', 'red').
        addEvent('click', function(e) {
            $(this).setStyle('color', 'green');
        });
});

```

这会把一个事件监听器关联到window对象的load事件。它执行的时候会立即显示ID值为test-1和test-2的两个元素并将其中的文字设置为红色，随后，它会为这两个元素添加click事件监听器，其作用是在它们被点击时将文字设置为绿色。代码虽然只有一点点，却表达了相当多的东西。

精通jQuery这个JavaScript库的读者应该比较熟悉这种接口。在这种编程方式中，window对象或某个HTML元素是调用链的锚点，所有操作都挂系在上面。在前面的例子中有两条调用链：一条为window对象添加load事件监听器，另一条则设置ID值为test-1和test-2的两个元素的样式并为其添加click事件监听器。几乎所有现有的实用程序都可以用这种方式来加以改造，以便进行链式调用。这将在下一节中详细讨论。

## 6.2 设计一个支持方法链式调用的 JavaScript 库

前面对\$函数的改造只提供了对几个最常用的实用函数进行链式调用的支持，但是你可以尽情对其扩充。设计一个JavaScript库需要深思熟虑。一个库不一定要有成百上千行代码，它的功能决定了它的大小。你可以借鉴一下各种JavaScript库中包含的那些最常用的特性。表6-1列出了几乎所有JavaScript库都有的基本特性。

表6-1 常见于大多数JavaScript库中的特性

| 特    性 | 说    明                  |
|--------|-------------------------|
| 事件     | 添加和删除事件监听器；对事件对象进行规范化处理 |
| DOM    | 类名管理；样式管理               |
| Ajax   | 对XMLHttpRequest进行规范化处理  |

要是对那个私用的\$\_构造函数进行扩充，把这些东西包括进去，那么其伪码大致是下面这个样子：

```
// Include syntactic sugar to help the development of our interface.
Function.prototype.method = function(name, fn) {
    this.prototype[name] = fn;
    return this;
};

(function() {
    function _$(els) {
        // ...
    }
    /* Events
     * addEvent
     * getEvent
    */
    _$.method('addEvent', function(type, fn) {
        // ...
    }).method('getEvent', function(e) {
        // ...
    });
    /* DOM
     * addClass
     * removeClass
    */
});
```

```

        * replaceClass
        * hasClass
        * getStyle
        * setStyle
    */
    method('addClass', function(className) {
        // ...
    }).method('removeClass', function(className) {
        // ...
    }).method('replaceClass', function(oldClass, newClass) {
        // ...
    }).method('hasClass', function(className) {
        // ...
    }).method('getStyle', function(prop) {
        // ...
    }).method('setStyle', function(prop, val) {
        // ...
    });
/*
AJAX
 * load. Fetches an HTML fragment from a URL and inserts it into an element.
*/
method('load', function(uri, method) {
    // ...
});
window.$ = function() {
    return new _$(arguments);
});
})();

```

87

搭好这个API的架子之后，现在该着重考虑的是它的使用者可能是些什么人，以及他们会在什么情况下使用它。如果某个现有的API已经定义了一个\$函数，那么我们的这个库会将其改写。有个简单的解决办法是在源代码中为\$函数另取一个名字。但是，如果你是从一个现有的源代码库（source-code repository）中取得的源代码，那么每次从这个代码库中获取更新版本的代码之后，你都不得不再次为那个函数改名，所以这个解决方案并不理想。在这种情况下，更好的解决办法是像下面这样添加一个安装器（installer）：

```

Function.prototype.method = function(name, fn) {
    // ...
};

(function() {
    function _$(els) {
        // ...
    }
    _$.method('addEvent', function(type, fn) {
        // ...
    });
});

window.installHelper = function(scope, interface) {

```

```

scope[interface] = function() {
    return new _$(arguments);
}
})();

```

用户可能会这样使用它：

```

installHelper(window, '$');

$('example').show();

```

下面是个更复杂的例子，它演示了如何把这种功能添加到一个事先定义好的命名空间对象中：

```

// Define a namespace without overwriting it if it already exists.
window.com = window.com || {};
com.example = com.example || {};
com.example.util = com.example.util || {};

installHelper(com.example.util, 'get');

(function() {
    var get = com.example.util.get;
    get('example').addEvent('click', function(e) {
        get(this).addClass('hello');
    });
})();

```

88

## 6.3 使用回调从支持链式调用的方法获取数据

有时把方法串连起来调用并不是个好主意。链式调用很适合于赋值器方法，但对于取值器方法，你可能会希望它们返回你要的数据而不是返回this。不过，如果你把链式调用作为首要目标，希望所有方法的使用方式保持一致的话，那么变通办法还是有的：你可以利用回调技术来返回所要的数据。下面的例子同时展示了这两种做法。其中，API类使用了普通的取值器（它中断了调用链），而API2类则使用了回调方法：<sup>①</sup>

```

// Accessor without function callbacks: returning requested data in accessors.
window.API = window.API || function() {
    var name = 'Hello world';
    // Privileged mutator method.
    this.setName = function(newName) {
        name = newName;
        return this;
    };
    // Privileged accessor method.

```

<sup>①</sup> 原书中这个例子的代码有严重错误。我已经做了修改。——译者注

```

this.getName = function() {
    return name;
};

// Implementation code.
var o = new API;
console.log(o.getName()); // Displays 'Hello world'.
console.log(o.setName('Meow').getName()); // Displays 'Meow'.

// Accessor with function callbacks.
window.API2 = window.API2 || function() {
    var name = 'Hello world';
    // Privileged mutator method.
    this.setName = function(newName) {
        name = newName;
        return this;
    };
    // Privileged accessor method.
    this.getName = function(callback) {
        callback.call(this, name);
        return this;
    };
};

// Implementation code.
var o2 = new API2;
o2.getName(console.log).setName('Meow').getName(console.log);
// Displays 'Hello world' and then 'Meow'.

```

89

90

## 6.4 小结

在JavaScript中对象是作为引用被传递的。所以你可以让每个方法都传回对象的引用。如果让一个类的每个方法都返回this值，那么它就成了一个支持方法的链式调用的类。这种编程风格有助于简化代码的编写工作，并且在某种程度上可以让代码更加简洁、易读。很多时候使用链式调用可以避免多次重复使用一个对象变量，从而减少代码量。如果想让类的接口保持一致，让赋值器和取值器方法都支持链式调用，那么你可以在取值器中使用回调技术。

# *Part 2*

第二部分

## 设计模式

### 本部分内容

- 第7章 工厂模式
- 第8章 桥接模式
- 第9章 组合模式
- 第10章 门面模式
- 第11章 适配器模式
- 第12章 装饰者模式
- 第13章 享元模式
- 第14章 代理模式
- 第15章 观察者模式
- 第16章 命令模式
- 第17章 职责链模式

# 工厂模式

一个类或对象中往往包含别的对象。在创建这种成员对象时，你可能习惯于使用常规方式，也即用new关键字和类构造函数。问题在于这会导致相关的两个类之间产生依赖性。本章将讲述一种有助于消除这两个类之间的依赖性的模式，它使用一个方法来决定究竟要实例化哪个具体的类。我们既要讨论简单工厂模式，也要讨论更复杂的工厂模式。前者另外使用一个类（通常是一个单体）来生成实例，而后者则使用子类来决定一个成员变量应该是哪个具体的类的实例。

## 7.1 简单工厂

最好用一个例子来说明简单工厂模式的概念。假设你想开几个自行车商店，每个店都有几种型号的自行车出售。这可以用一个类来表示：

```
/* BicycleShop class. */

var BicycleShop = function() {};
BicycleShop.prototype = {
    sellBicycle: function(model) {
        var bicycle;

        switch(model) {
            case 'The Speedster':
                bicycle = new Speedster();
                break;
            case 'The Lowrider':
                bicycle = new Lowrider();
                break;
            case 'The Comfort Cruiser':
                default:
                    bicycle = new ComfortCruiser();
        }

        Interface.ensureImplements(bicycle, Bicycle);
        bicycle.assemble();
        bicycle.wash();

        return bicycle;
    }
};
```

`sellBicycle`方法根据所要求的自行车型号用`switch`语句创建一个自行车的实例。各种型号的自行车实例可以互换使用，因为它们都实现了`Bicycle`接口：

**注解** 接口在工厂模式中起着很重要的作用。如果不对对象进行某种类型检查以确保其实现了必需的方法，工厂模式带来的好处也就所剩无几了。在所有这些例子中，你可以创建一些对象并且对它们一视同仁，那是因为你可以确信它们都实现了同样一批方法。

```
/* The Bicycle interface. */

var Bicycle = new Interface('Bicycle', ['assemble', 'wash', 'ride', 'repair']);

/* Speedster class. */

var Speedster = function() { // implements Bicycle
  ...
};

Speedster.prototype = {
  assemble: function() {
    ...
  },
  wash: function() {
    ...
  },
  ride: function() {
    ...
  },
  repair: function() {
    ...
  }
};
```

要出售某种型号的自行车，只要调用`sellBicycle`方法即可：

```
var californiaCruisers = new BicycleShop();
var yourNewBike = californiaCruisers.sellBicycle('The Speedster');
```

在情况发生变化之前，这倒也挺管用。但要是你想在供货目录中加入一款新车型又会怎么样呢？你得为此修改`BicycleShop`的代码，哪怕这个类的实际功能实际上并没有发生改变——依旧是创建一个自行车的新实例，组装它，清洗它，然后把它交给顾客。更好的解决办法是把`sellBicycle`方法中“创建新实例”这部分工作转交给一个简单工厂对象：

```
/* BicycleFactory namespace. */

var BicycleFactory = {
  createBicycle: function(model) {
    var bicycle;

    switch(model) {
      case 'The Speedster':
```

```

        bicycle = new Speedster();
        break;
    case 'The Lowrider':
        bicycle = new Lowrider();
        break;
    case 'The Comfort Cruiser':
    default:
        bicycle = new ComfortCruiser();
    }

Interface.ensureImplements(bicycle, Bicycle);
return bicycle;
}
);

```

BicycleFactory是一个单体，用来把createBicycle方法封装在一个命名空间中。这个方法返回一个实现了Bicycle接口的对象，然后你可以照常对其进行组装和清洗：

```

/* BicycleShop class, improved. */

var BicycleShop = function() {};
BicycleShop.prototype = {
    sellBicycle: function(model) {
        var bicycle = BicycleFactory.createBicycle(model);

        bicycle.assemble();
        bicycle.wash();

        return bicycle;
    }
};

```

95

这个BicycleFactory对象可以供各种类用来创建新的自行车实例。有关可供车型的所有信息都集中在一个地方管理，所以添加更多车型很容易：

```

/* BicycleFactory namespace, with more models. */

var BicycleFactory = {
    createBicycle: function(model) {
        var bicycle;

        switch(model) {
            case 'The Speedster':
                bicycle = new Speedster();
                break;
            case 'The Lowrider':
                bicycle = new Lowrider();
                break;
            case 'The Flatlander':
                bicycle = new Flatlander();
                break;
        }
    }
};

```

```

        case 'The Comfort Cruiser':
    default:
        bicycle = new ComfortCruiser();
    }

    Interface.ensureImplements(bicycle, Bicycle);
    return bicycle;
}
};


```

BicycleFactory就是简单工厂的一个很好的例子。这种模式把成员对象的创建工作转交给一个外部对象。这个外部对象可以像本例中一样是一个简单的命名空间，也可以是一个类的实例。如果负责创建实例的方法的逻辑不会发生变化，那么一般说来用单体或静态类方法创建这些成员实例是合乎情理的。但如果要提供几种不同品牌的自行车，那么更恰当的做法是把这个创建方法实现在一个类中，并从该类派生出一些子类。

## 7.2 工厂模式

真正的工厂模式与简单工厂模式的区别在于，它不是另外使用一个类或对象来创建自行车（就像前面的例子中所做的那样），而是使用一个子类。按照正式定义，工厂是一个将其成员对象的实例化推迟到子类中进行的类。我们还是以BicycleShop为例来说明简单工厂和工厂模式之间的差别。

我们打算让各个自行车商店自行决定从哪个生产厂家进货。出于这个原因，单单一个BicycleFactory对象将无法提供需要的所有自行车实例。我们可以把BicycleShop设计为抽象类，让子类根据各自的进货渠道实现其createBicycle方法，：

```

/* BicycleShop class (abstract). */

var BicycleShop = function() {};
BicycleShop.prototype = {
    sellBicycle: function(model) {
        var bicycle = this.createBicycle(model);

        bicycle.assemble();
        bicycle.wash();

        return bicycle;
    },
    createBicycle: function(model) {
        throw new Error('Unsupported operation on an abstract class.');
    }
};

```

这个类中定义了createBicycle方法，但真要调用这个方法的话，会抛出一个错误。现在BicycleShop是一个抽象类，它不能被实例化，只能用来派生子类。设计一个经销特定自行车生产厂家产品的子类需要扩展BicycleShop，重定义其中的createBicycle方法。下面是两个子类的

例子，其中一个子类代表的商店从Acme公司进货，而另一个则从General Products公司进货：

```
/* AcmeBicycleShop class. */

var AcmeBicycleShop = function() {};
extend(AcmeBicycleShop, BicycleShop);
AcmeBicycleShop.prototype.createBicycle = function(model) {
    var bicycle;

    switch(model) {
        case 'The Speedster':
            bicycle = new AcmeSpeedster();
            break;
        case 'The Lowrider':
            bicycle = new AcmeLowrider();
            break;
        case 'The Flatlander':
            bicycle = new AcmeFlatlander();
            break;
        case 'The Comfort Cruiser':
        default:
            bicycle = new AcmeComfortCruiser();
    }
    Interface.ensureImplements(bicycle, Bicycle);
    return bicycle;
};

/* GeneralProductsBicycleShop class. */

var GeneralProductsBicycleShop = function() {};
extend(GeneralProductsBicycleShop, BicycleShop);
GeneralProductsBicycleShop.prototype.createBicycle = function(model) {
    var bicycle;

    switch(model) {
        case 'The Speedster':
            bicycle = new GeneralProductsSpeedster();
            break;
        case 'The Lowrider':
            bicycle = new GeneralProductsLowrider();
            break;
        case 'The Flatlander':
            bicycle = new GeneralProductsFlatlander();
            break;
        case 'The Comfort Cruiser':
        default:
            bicycle = new GeneralProductsComfortCruiser();
    }
    Interface.ensureImplements(bicycle, Bicycle);
}
```

```

    return bicycle;
};

}

```

这些工厂方法生成的对象都实现了Bicycle接口，所以在其他代码眼里它们完全可以互换。自行车的销售工作还是与以前一样，只是现在所开的商店可以是Acme或General Products自行车专商店：

```

var alecsCruisers = new AcmeBicycleShop();
var yourNewBike = alecsCruisers.sellBicycle('The Lowrider');

var bobsCruisers = new GeneralProductsBicycleShop();
var yourSecondNewBike = bobsCruisers.sellBicycle('The Lowrider');

```

因为两个生产厂家生产的自行车款式完全相同，所以顾客买车时可以不用关心车究竟是哪家生产的。要是他们只想要Acme生产的自行车，他们可以去Acme专商店买。

增加对其他生产厂家的支持很简单，只要再创建一个BicycleShop的子类并重定义其createBicycle工厂方法即可。我们也可以对各个子类进行修改，以支持相关厂家其他型号的产品。这是工厂模式最重要的特点。对Bicycle进行一般性操作的代码可以全部写在父类BicycleShop中，而对具体的Bicycle对象进行实例化的工作则被留到子类中。一般性的代码被集中在一个位置，而个体性的代码则被封装在子类中。

98

## 7.3 工厂模式的适用场合

创建新对象最简单的办法是使用new关键字和具体类。只有在某些情况下，创建和维护对象工厂所带来的额外复杂性才是物有所值。本节概括了这些场合。

### 7.3.1 动态实现

如果需要像前面自行车的例子一样，创建一些用不同方式实现同一接口的对象，那么可以使用一个工厂方法或简单工厂对象来简化选择实现的过程。这种选择可以是明确进行的也可以是隐含的。前者如自行车那个例子，顾客可以选择需要的自行车型号；而下一节所讲的XHR工厂那个例子则属于后者，该例中所返回的连接对象的类型取决于所探查到的带宽和网络延时等因素。在这些场合下，你通常要与一系列实现了同一个接口、可以被同等对待的类打交道。这是JavaScript中使用工厂模式的最常见的原因。

### 7.3.2 节省设置开销

如果对象需要进行复杂并且彼此相关的设置，那么使用工厂模式可以减少每种对象所需的代码量。如果这种设置只需要为特定类型的所有实例执行一次即可，这种作用尤其突出。把这种设置代码放到类的构造函数中并不是一种高效的做法，这是因为即便设置工作已经完成，每次创建新实例的时候这些代码还是会执行，而且这样做会把设置代码分散到不同的类中。工厂方法非常适合于这种场合。它可以在实例化所有需要的对象之前先一次性地进行设置。无论有多少不同的类会被实例化，这种办法都可以让设置代码集中在一个地方。

如果所用的类要求加载外部库的话，这尤其有用。工厂方法可以对这些库进行检查并动态加载那些未找到的库。这些设置代码只存在于一个地方，因此以后改起来也方便得多。

### 7.3.3 用许多小型对象组成一个大对象

工厂方法可以用来创建封装了许多较小对象的对象。考虑一下自行车对象的构造函数。自行车包含着许多更小的子系统：车轮、车架、传动部件以及车闸等。如果你不想让某个子系统与较大的那个对象之间形成强耦合，而是想在运行时从许多子系统中进行挑选的话，那么工厂方法是一个理想的选择。使用这种技术，某天你可以为售出的所有自行车配上某种链条，要是第二天找到另一种更中意的链条，可以改而采用这个新品种。实现这种改变很容易，因为这些自行车类的构造函数并不依赖于某种特定的链条品种。本章后面RSS阅读器的例子演示了工厂模式在这方面的用途。

## 7.4 示例：XHR 工厂

用Ajax技术发起异步请求是现在Web开发中的一个常见任务。用于发起请求的对象是某种类的实例，具体是哪种类取决于用户的浏览器。如果代码中需要多次执行Ajax请求，那么明智的做法是把创建这种对象的代码提取到一个类中，并创建一个包装器来包装在实际发起请求时所要经历的一系列步骤。简单工厂非常适合这种场合，它可以用根据浏览器能力的不同生成一个 XMLHttpRequest或ActiveXObject实例。

```
/* AjaxHandler interface. */

var AjaxHandler = new Interface('AjaxHandler', ['request', 'createXhrObject']);

/* SimpleHandler class. */

var SimpleHandler = function() {} // implements AjaxHandler
SimpleHandler.prototype = {
    request: function(method, url, callback, postVars) {
        var xhr = this.createXhrObject();
        xhr.onreadystatechange = function() {
            if(xhr.readyState !== 4) return;
            if(xhr.status === 200) ?
                callback.success(xhr.responseText, xhr.responseXML) :
                callback.failure(xhr.status);
        };
        xhr.open(method, url, true);
        if(method !== 'POST') postVars = null;
        xhr.send(postVars);
    },
    createXhrObject: function() { // Factory method.
        var methods = [
            function() { return new XMLHttpRequest(); },
            function() { return new ActiveXObject('Msxml2.XMLHTTP'); },
        ];
        for(var i = 0; i < methods.length; i++) {
            if(methods[i]()) return methods[i]();
        }
    }
};
```

```

function() { return new ActiveXObject('Microsoft.XMLHTTP'); }
];

for(var i = 0, len = methods.length; i < len; i++) {
  try {
    methods[i]();
  }
  catch(e) {
    continue;
  }
  // If we reach this point, method[i] worked.
  this.createXhrObject = methods[i]; // Memoize the method.
  return methods[i];
}

// If we reach this point, none of the methods worked.
throw new Error('SimpleHandler: Could not create an XHR object.');
}
};

request这个便利函数负责执行发出请求和处理响应结果所需的一系列操作。它先创建一个

```

100

XHR对象并对其进行配置，然后再发送请求。这里所关注的是用于创建XHR对象的代码。

`createXhrObject`这个工厂方法根据当前环境的具体情况返回一个XHR对象。在首次执行时，它会依次尝试三种用于创建XHR对象的不同方法，一旦遇到一种管用的，它就会返回所创建的对象并将自身改为用以创建那个对象的函数。这个新函数于是摇身一变成了`createXhrObject`方法。这种技术被称为memoizing<sup>①</sup>，它可以用来创建存储着复杂计算的函数和方法，以免再次进行这种计算。所有那些复杂的设置代码只会在方法首次执行时被调用一次，此后就只有针对当前浏览器的代码会被执行。假如前面的代码运行在一个实现了`XMLHttpRequest`类的浏览器中，那么第二次执行时的`createXhrObject`方法实际上是这个样子：

```
createXhrObject: function() { return new XMLHttpRequest(); }
```

`memoizing`技术可以提高代码的效率，因为所有设置和检测代码都只会执行一次。工厂方法是这种代码的理想封装工具，不管代码运行在什么平台上，它都能返回正确的对象。这一任务涉及的复杂性由此被集中在一个地方。

用`SimpleHandler`类发起异步请求的过程很简单，只要创建该类的一个实例，调用它的`request`方法即可：

```

var myHandler = new SimpleHandler();
var callback = {
  success: function(responseText) { alert('Success: ' + responseText); },
  failure: function(statusCode) { alert('Failure: ' + statusCode); }
};
myHandler.request('GET', 'script.php', callback);

```

<sup>①</sup> memoization是Donald Michie根据拉丁语中的memorandum（意为“记住”。该词也见于现代英语，不过却是“备忘录”等意思）杜撰的一个词。相应的动词、过去分词和ing形式分别为memoiz、memoized和memoizing。——译者注

### 7.4.1 专用型连接对象

这个例子可以进一步扩展，把工厂模式用在两个地方，以便根据网络条件创建专门的请求对象。在创建XHR对象时已经用过了简单工厂模式。另一个工厂则用来返回各种处理器类，它们都派生自SimpleHandler。

首先要做的是创建两个新的处理器类。QueuedHandler会在发起新的请求之前先确保所有请求都已经成功处理。而OfflineHandler则会在用户处于离线状态时把请求缓存起来。

```
/* QueuedHandler class. */

var QueuedHandler = function() { // implements AjaxHandler
    this.queue = [];
    this.requestInProgress = false;
    this.retryDelay = 5; // In seconds.
};

extend(QueuedHandler, SimpleHandler);
QueuedHandler.prototype.request = function(method, url, callback, postVars,
    override) {
    if(this.requestInProgress && !override) {
        this.queue.push({
            method: method,
            url: url,
            callback: callback,
            postVars: postVars
        });
    }
    else {
        this.requestInProgress = true;
        var xhr = this.createXhrObject();
        var that = this;
        xhr.onreadystatechange = function() {
            if(xhr.readyState !== 4) return;
            if(xhr.status === 200) {
                callback.success(xhr.responseText, xhr.responseXML);
                that.advanceQueue();
            }
            else {
                callback.failure(xhr.status);
                setTimeout(function() { that.request(method, url, callback, postVars, true); },
                    that.retryDelay * 1000);
            }
        };
        xhr.open(method, url, true);
        if(method !== 'POST') postVars = null;
        xhr.send(postVars);
    }
};
```

```

QueuedHandler.prototype.advanceQueue = function() {
  if(this.queue.length === 0) {
    this.requestInProgress = false;
    return;
  }
  var req = this.queue.shift();
  this.request(req.method, req.url, req.callback, req.postVars, true);
};

```

QueueHandler的request方法与SimpleHandler的看上去差不多，但在允许发起新请求之前它会先检查一下，以确保当前没有别的请求正在处理。如果有哪个请求未能成功处理，那么它还会在指定的时间间隔之后再次重复这个请求，直到该请求被成功处理为止。

OfflineHandler要简单一点：

```

/* OfflineHandler class. */

var OfflineHandler = function() { // implements AjaxHandler
  this.storedRequests = [];
};

extend(OfflineHandler, SimpleHandler);
OfflineHandler.prototype.request = function(method, url, callback, postVars) {
  if(XhrManager.isOffline()) { // Store the requests until we are online.
    this.storedRequests.push({
      method: method,
      url: url,
      callback: callback,
      postVars: postVars
    });
  } else { // Call SimpleHandler's request method if we are online.
    this.flushStoredRequests();
    OfflineHandler.superclass.request(method, url, callback, postVars);
  }
};

OfflineHandler.prototype.flushStoredRequests = function() {
  for(var i = 0, len = storedRequests.length; i < len; i++) {
    var req = storedRequests[i];
    OfflineHandler.superclass.request(req.method, req.url, req.callback,
      req.postVars);
  }
};

```

102

XhrManager.isOffline方法（稍后会详细介绍）的作用在于判断用户是否处于在线状态。OfflineHandler只有在用户处于在线状态时才会使用SimpleHandler的request方法实际发起请求。而且一旦探测到用户处于在线状态，它还会立即执行所有缓存中的请求。

#### 7.4.2 在运行时选择连接对象

现在该用到工厂模式了。因为程序员根本不可能知道各个最终用户实际面临的网络条件，

所以不可能要求他们在开发过程中选择使用哪个处理器类，而是应该用一个工厂在运行时选择最合适的类。程序员只需要调用这个工厂方法并使用其返回的对象即可。因为所有这些处理器类都实现了AjaxHandler接口，所以它们可以被同等对待。接口是相同的，区别只在于其实现：

```
/* XhrManager singleton. */

var XhrManager = {
  createXhrHandler: function() {
    var xhr;
    if(this.isOffline()) {
      xhr = new OfflineHandler();
    }
    else if(this.isHighLatency()) {
      xhr = new QueuedHandler();
    }
    else {
      xhr = new SimpleHandler()
    }

    Interface.ensureImplements(xhr, AjaxHandler);
    return xhr
  },
  isOffline: function() { // Do a quick request with SimpleHandler and see if
    ... // it succeeds.
  },
  isHighLatency: function() { // Do a series of requests with SimpleHandler and
    ... // time the responses. Best done once, as a
    // branching function.
  }
};
```

现在程序员就可以使用这个工厂方法，而不必实例化一个特定的类了：

```
var myHandler = XhrManager.createXhrHandler();
var callback = {
  success: function(responseText) { alert('Success: ' + responseText); },
  failure: function(statusCode) { alert('Failure: ' + statusCode); }
};
myHandler.request('GET', 'script.php', callback);
```

`createXhrHandler`方法返回的各种对象都具有我们所需要的一些方法。而且，因为它们都派生自`SimpleHandler`，所以`createXhrObject`这个复杂的方法只需要在这个类中实现一次即可，那些子类可以使用这个方法。`OfflineHandler`中还有多处使用了`SimpleHandler`的`request`方法，这进一步实现了代码的重用。

这里省略了`isOffline`和`isHighLatency`方法的实现细节。在实际实现这些方法的时候，需要先编写一个方法，它会用`setTimeout`安排执行一些异步请求，并记录它们的往返时间。只要这些

请求中的任何一个得到回应，`isOffline`方法就会返回`false`，否则它会返回`true`。而`isHighLatency`方法会检查请求得到回应所经历的时间，并根据其长短来决定该返回`true`还是`false`。这些方法的具体实现都是些细枝末节，这里不再赘述。

## 7.5 示例：RSS 阅读器

下面要设计的是一个用来在网页上显示来自RSS源的最新信息的小工具。我们打算重用一些现有模块，比如前面例子中的XHR处理器，而不是从头做起。最终得到的是一个RSS阅读器对象，它的成员对象包括一个XHR处理器对象、一个显示对象以及一个配置对象。

由于我们只想跟RSS容器对象打交道，所以用一个工厂来实例化这些内部对象并把它们组装到一个RSS阅读器对象中。使用工厂方法的好处在于，我们创建的RSS阅读器类不会与那些成员对象紧密耦合在一起。我们可以使用任何一个显示模块，只要它实现了必要的方法就行，因此大可不必让阅读器类套牢在某个特定的显示类上。

有了这个工厂方法，只要愿意，我们可以随时换掉任何一个模块，不管是在开发期间还是在运行期间。使用这个API的程序员得到的还是一个完整的RSS阅读器对象，其中所有的成员对象都已经实例化并配置完毕，但其中涉及的类之间的耦合都比较松散，因此可以随意更换。

先来看看要在工厂方法中进行实例化的那些类。XHR处理器类你已经见过了。本例使用`XhrManager.createXhrHandler`方法创建所用的处理器对象。下一个类是显示类。为了满足RSS阅读器类的需要，它需要实现几个方法。下面是一个实现了那些必要方法的显示类，它把输出内容包装为一个无序列表：

```
/* DisplayModule interface. */

var DisplayModule = new Interface('DisplayModule', ['append', 'remove', 'clear']);

/* ListDisplay class. */

var ListDisplay = function(id, parent) { // implements DisplayModule
    this.list = document.createElement('ul');
    this.list.id = id;
    parent.appendChild(this.list);
};

ListDisplay.prototype = {
    append: function(text) {
        var newEl = document.createElement('li');
        this.list.appendChild(newEl);
        newEl.innerHTML = text;
        return newEl;
    },
    remove: function(el) {
        this.list.removeChild(el);
    },
    clear: function() {
        this.list.innerHTML = '';
    }
};
```

```

    };
}

```

下一个要用到的是配置对象。这只是一个对象字面量，它包含着一些供阅读器类及其成员对象使用的设置：

```
/* Configuration object. */
```

```

var conf = {
  id: 'cnn-top-stories',
  feedUrl: 'http://rss.cnn.com/rss/cnn_topstories.rss',
  updateInterval: 60, // In seconds.
  parent: $('#feed-readers')
};

```

105

这些类都由一个名为FeedReader的类组合使用。它用XHR处理器从RSS源获取XML格式的数据并用一个内部方法对其进行解析，然后用显示模块将解析出来的信息输出到网页上：

```

/* FeedReader class. */

var FeedReader = function(display, xhrHandler, conf) {
  this.display = display;
  this.xhrHandler = xhrHandler;
  this.conf = conf;
  this.startUpdates();
};

FeedReader.prototype = {
  fetchFeed: function() {
    var that = this;
    var callback = {
      success: function(text, xml) { that.parseFeed(text, xml); },
      failure: function(status) { that.showError(status); }
    };
    this.xhrHandler.request('GET', 'feedProxy.php?feed=' + this.conf.feedUrl, callback);
  },
  parseFeed: function(responseText, responseXML) {
    this.display.clear();
    var items = responseXML.getElementsByTagName('item');
    for(var i = 0, len = items.length; i < len; i++) {
      var title = items[i].getElementsByTagName('title')[0];
      var link = items[i].getElementsByTagName('link')[0];
      this.display.append('<a href="' + link.firstChild.data + '">' +
        title.firstChild.data + '</a>');
    }
  },
  showError: function(statusCode) {
    this.display.clear();
    this.display.append('Error fetching feed.');
  }
};

```

```

stopUpdates: function() {
  clearInterval(this.interval);
},
startUpdates: function() {
  this.fetchFeed();
  var that = this;
  this.interval = setInterval(function() { that.fetchFeed(); }, this.conf.updateInterval * 1000);
}
};

```

XHR请求中使用的feedProxy.php脚本是一个代理。可以用它从外部域获取数据，以应对JavaScript的同源限制机制。那种允许从任何URL获取数据的开放性代理容易被滥用，因此应该避免使用。像本例中这样使用代理时，应该硬性设定一个允许访问的URL的白名单（whitelist），拒绝对未列入其中的URL的访问。

现在还差一个部分，即把所有这些类和对象拼装起来的那个工厂方法。它被实现为一个简单工厂，如下所示：

```

/* FeedManager namespace. */

var FeedManager = {
  createFeedReader: function(conf) {
    var displayModule = new ListDisplay(conf.id + '-display', conf.parent);
    Interface.ensureImplements(displayModule, DisplayModule);

    var xhrHandler = XhrManager.createXhrHandler();
    Interface.ensureImplements(xhrHandler, AjaxHandler);

    return new FeedReader(displayModule, xhrHandler, conf);
  }
};

```

这个工厂方法先实例化所需要的模块，确认它们实现了正确的方法，然后把它们传递给FeedReader构造函数。

这个示例中的工厂方法能带来什么好处呢？使用这个API的程序员当然可以手工创建一个FeedReader对象，而不必借助FeedManager.createFeedReader方法。但使用这个工厂方法，可以把FeedReader类所需的复杂设置封装起来，并且可以确保其成员对象都实现了所需接口。它还把对所使用的特定模块的硬性设定（ListDisplay和XhrManager.createXhrHandler）集中在一个位置。哪天要是想使用ParagraphDisplay和QueuedHandler，做起来也同样简单，只要改改这个工厂方法内部的代码就行。你也可以添加一些代码，像XHR处理器示例中所示的那样在运行期间从一些可用的模块中进行选择。总而言之，这是一个阐明“用许多小型对象组成一个大对象”这个用途的绝佳示例。它使用工厂模式，先创建出所有要用到的对象，然后再生成并返回那个作为容器的FeedReader类型大对象。本书网站<http://jsdesignpatterns.com/>所提供的第7章的示例代码中有一个上述代码的可运行版本，它内嵌在一个网页之中。

## 7.6 工厂模式之利

工厂模式的主要好处在于消除对象间的耦合。通过使用工厂方法而不是new关键字及具体类，你可以把所有实例化代码集中在一个位置。这可以大大简化更换所用的类或在运行期间动态选择所用的类的工作。在派生子类时它也提供了更大的灵活性。使用工厂模式，你可以先创建一个抽象的父类，然后在子类中创建工作方法，从而把成员对象的实例化推迟到更专门化的子类中进行。

107

所有这些好处都与面向对象设计的这两条原则有关：弱化对象间的耦合；防止代码的重复。在一个方法中进行类的实例化，可以消除重复性的代码。这是在用一个对接口的调用取代一个具体的实现。这些都有助于创建模块化的代码。

## 7.7 工厂模式之弊

可能有人禁不住想把工厂方法当万金油用，把普通的构造函数扔在一边。这并不值得提倡。如果根本不可能另外换用一个类，或者不需要在运行期间在一系列可互换的类中进行选择，那就不应该使用工厂方法。大多数类最好使用new关键字和构造函数公开地进行实例化。这可以让代码更简单易读。你可以一眼就看到调用的是什么构造函数，而不必去查看某个工厂方法以便知道实例化的是什么类。工厂方法在其适用场合非常有用，但切勿滥用。如果拿不定主意，那就不要用，因为以后在重构代码时还有机会使用工厂模式。

## 7.8 小结

本章讨论了简单工厂和工厂模式。我们以自行车商店为例说明了二者之间的差别：简单工厂通常另外使用一个类或对象封装实例化操作，而真正的工厂模式则要实现一个抽象的工厂方法并把实例化工作推迟到子类中进行。这种模式有几种明确的应用场合。它主要用在所实例化的类的类型不能在开发期间确定，而只能在运行期间确定的情况下。此外，如果存在着许多具有复杂的设置开销的相关对象，或者想创建一个包含了一些成员对象的类但又想避免把它们紧密耦合在一起的话，那么这也是这种模式的用武之地。不要盲目地把工厂模式用于所有的实例化任务。但如果运用得当的话，它将是JavaScript程序员手中的一大利器。

108

**在**实现API的时候，桥接模式非常有用。实际上，这也许是被用得最不够充分的模式之一。在所有模式中，这种模式最容易立即付诸实施。在设计一个JavaScript API的时候，可以用这个模式来弱化它与使用它的类和对象之间的耦合。按GoF的定义，桥接模式的作用在于“将抽象与其实现隔离开来，以便二者独立变化”。这种模式对于JavaScript中常见的事件驱动的编程大有裨益。

如果你刚进入JavaScript API开发的世界，那么很可能要创建许多获取方法（getter）、设置方法（setter）、请求方法（requester）以及别的基于动作的方法。无论它们是用来创建Web服务API还是普通的取值器（accessor）方法和赋值器（mutator）方法，在实现过程中桥接模式都有助于保持API代码的简洁。

## 8.1 示例：事件监听器

桥接模式最常见和实际的应用场合之一是事件监听器回调函数。假设有一个名为getBeerById的API函数，它根据一个标识符返回有关某种啤酒的信息。当然，在Web应用软件中，你希望在用户执行某种操作（比如点击一个元素）时获取这种信息。那个被点击的元素很可能具有啤酒的标识符信息，它可能是作为元素自身的ID保存，也可能是作为别的自定义属性保存。下面是一种做法：

```
addEvent(element, 'click', getBeerById);
function getBeerById(e) {
    var id = this.id;
    asyncRequest('GET', 'beer.uri?id=' + id, function(resp) {
        // Callback response.
        console.log('Requested Beer: ' + resp.responseText);
    });
}
```

可以看出这是一个只能工作在浏览器中的API。根据事件监听器回调函数的工作机制，事件对象自然会被作为第一个参数传递给这个函数。在本例中并没有使用这个参数，而只是从this对

<sup>①</sup> 注意本章中所说的“接口”，基本上都是指API（应用编程接口）一词中那个接口，而不是第2章所说的接口的意思。后面的各章有时也有这种情况，读者当能自行分辨，故不再赘述。——译者注

象获取ID。如果你要对这个API函数做单元测试，或更有甚者，在命令行环境中执行它，那就只有祝你好运了。对于API开发来说，最好从一个优良的API开始，不要把它与任何特定的实现搅在一起。毕竟，我们希望所有人都能获取到啤酒的信息：

```
function getBeerById(id, callback) {
  // Make request for beer by ID, then return the beer data.
  asyncRequest('GET', 'beer.uri?id=' + id, function(resp) {
    // callback response
    callback(resp.responseText);
  });
}
```

你想必也会同意：这个版本要实用得多。从逻辑上讲，把ID作为参数传递给一个名为getBeerById的函数是合情合理的。正如大多数“getter”函数所做的那样，这里使用了一个回调函数。向服务器请求提供信息时，回应结果总是通过一个回调函数返回。现在，我们将针对接口而不是实现进行编程（就像第2章所说的那样），用桥接模式把抽象隔离开来。看看修改后的那个事件监听器：

```
addEvent(element, 'click', getBeerByIdBridge);
function getBeerByIdBridge (e) {
  getBeerById(this.id, function(beer) {
    console.log('Requested Beer: '+beer);
  });
}
```

有了这层桥接元素，这个API的适用范围大大拓宽了，这给了你更大的设计自由。因为现在getBeerById并没有和事件对象捆绑在一起，你可以在单元测试中运行这个API。只需提供一个ID和回调函数，看哪：唾手可得的啤酒！此外，你也可以在命令行环境中（比如使用Firebug或Venkman）对这个接口进行快速测试。

## 8.2 桥接模式的其他例子

除了在事件回调函数与接口之间进行桥接外，桥接模式也可以用于连接公开的API代码和私用的实现代码。此外，它还可以用来把多个类联结在一起。从类的角度来看，这意味着把接口作为公开的代码编写，而把类的实现作为私用代码编写。

如果一个公用的接口抽象了一些也许应该属于私用性的（尽管在此情况下它不一定非得是私用的）较复杂的任务，那么可以使用桥接模式来收集某些私用性的信息。可以用一些具有特殊权利的方法作为桥梁以便访问私用变量空间，而不必冒险下到具体实现的浑水中。这一特例中的桥接性函数又称特权函数，第3章对此有详细说明。

```
var Public = function() {
  var secret = 3;
  this.privilegedGetter = function() {
    return secret;
  };
};
```

```
var o = new Public;
var data = o.privilegedGetter();
```

## 8.3 用桥接模式联结多个类

在现实生活中桥梁可以把多种事物联结起来，在JavaScript中也是如此：

```
var Class1 = function(a, b, c) {
    this.a = a;
    this.b = b;
    this.c = c;
}
var Class2 = function(d) {
    this.d = d;
};

var BridgeClass = function(a, b, c, d) {
    this.one = new Class1(a, b, c);
    this.two = new Class2(d);
};
```

这看起来很像是——适配器。

的确如此。但要注意到本例中实际上并没有客户系统要求提供数据。它只不过是用来接纳大量数据并将其发送给责任方的一种辅助性手段。此外，BridgeClass也不是一个客户系统已经实现的现有接口。引入这个类的目的只不过是要桥接一些类而已。

有人可能会争辩说引入这个桥接类完全是为了方便，实际上它就是一个门面类（facade）的角色。但是，这里使用桥接模式是为了让Class1和Class2能够独立于BridgeClass而发生改变，这与门面类是不同的。

## 8.4 示例：构建 XHR 连接队列

在本例中我们要构建一个Ajax请求队列。这个对象把请求存储在浏览器内存中的一个队列化数组中。刷新队列时每个请求都会按“先入先出”的顺序被发送给一个后端的Web服务。如果次序事关紧要，那么在Web应用程序中使用队列化系统是有好处的。另外队列还有一个好处：可以通过从队列中删除请求来实现应用程序的“取消”功能。电子邮件程序、富文本编辑器或任何涉及因用户输入引起的频繁动作的系统都是适用的例子。最后，连接队列可以帮助用户克服慢速网络连接带来的不便，甚至可以允许他们离线工作。当然，要想把请求发给服务器，还是需要重建连接。

队列系统开发出来后，我们会找出那些存在强耦合的抽象的部分，然后用桥接模式把抽象与实现分开。在此你几乎可以立即看出使用桥接模式的好处。

### 8.4.1 添加核心工具

在着手工作之前需要先准备一些核心工具函数。因为要通过XMLHttpRequest与服务器通信，

所以需要用下面这个asyncRequest函数（第11章讲适配器模式时也要用到它）解决浏览器的差异问题：

```

var asyncRequest = (function() {
    function handleReadyState(o, callback) {
        var poll = window.setInterval(
            function() {
                if (o && o.readyState == 4) {
                    window.clearInterval(poll);
                    if (callback) {
                        callback(o);
                    }
                }
            },
            50
        );
    }
    var getXHR = function() {
        var http;
        try {
            http = new XMLHttpRequest();
            getXHR = function() {
                return new XMLHttpRequest();
            };
        } catch(e) {
            var msxml = [
                'MSXML2.XMLHTTP.3.0',
                'MSXML2.XMLHTTP',
                'Microsoft.XMLHTTP'
            ];
            for (var i = 0, len = msxml.length; i < len; ++i) {
                try {
                    http = new ActiveXObject(msxml[i]);
                    getXHR = function() {
                        return new ActiveXObject(msxml[i]);
                    };
                    break;
                }
                catch(e) {}
            }
        }
        return http;
    };
    return function(method, uri, callback, postData) {
        var http = getXHR();
        http.open(method, uri, true);
        handleReadyState(http, callback);
        http.send(postData || null);
    };
}

```

```

        return http;
    };
})();

```

有了下面的代码，我们就可以用类似于讲述链式调用的第6章中的风格进行开发：

```

Function.prototype.method = function(name, fn) {
    this.prototype[name] = fn;
    return this;
};

```

最后要添加的是两个新的数组方法：`forEach`和`filter`。这是对`Array`的原型对象的扩展。它们是在JavaScript 1.6语言核心中引入的，而目前大多数浏览器使用的还是1.5版的语言核心。我们先检查一下浏览器有没有实现这些方法，如果没有，就添加进去：

```

if ( !Array.prototype.forEach ) {
    Array.method('forEach', function(fn, thisObj) {
        var scope = thisObj || window;
        for ( var i = 0, len = this.length; i < len; ++i ) {
            fn.call(scope, this[i], i, this);
        }
    });
}

if ( !Array.prototype.filter ) {
    Array.method('filter', function(fn, thisObj) {
        var scope = thisObj || window;
        var a = [];
        for ( var i = 0, len = this.length; i < len; ++i ) {
            if ( !fn.call(scope, this[i], i, this) ) { continue; }
            a.push(this[i]);
        }
        return a;
    });
}

```

Mozilla开发人员中心网站 ([http://developer.mozilla.org/en/docs/New\\_in\\_JavaScript\\_1.6#Array\\_extras](http://developer.mozilla.org/en/docs/New_in_JavaScript_1.6#Array_extras)) 上有关于这些数组方法的详细说明。

## 8.4.2 添加观察者系统

观察者系统在监听队列向客户发送的事件通知的过程中扮演着一个关键角色。关于观察者的详细信息参见第15章。现在我们要添加的是一个基本的系统：

```

window.DED = window.DED || {};
DED.util = DED.util || {};
DED.util.Observer = function() {
    this.fns = [];
}

```

```

DED.util.Observer.prototype = {
  subscribe: function(fn) {
    this.fns.push(fn);
  },
  unsubscribe: function(fn) {
    this.fns = this.fns.filter(
      function(el) {
        if (el !== fn) {
          return el;
        }
      }
    );
  },
  fire: function(o) {
    this.fns.forEach(
      function(el) {
        el(o);
      }
    );
  }
};

```

### 8.4.3 开发队列的基本框架

在这一特定系统中，我们希望该队列具有一些关键特性。首先，它必须是一个真正的队列，遵从先入先出的基本规则。与此不同，栈则有先入后出的特点，因此不是我们的目标。

因为这是一个用于存储待发请求的连接队列，所以你可能希望设置“重试”的次数限制。此外，根据每个队列的请求的大小，你可能也希望能设置“超时”限制。

最后，我们应该能够向队列添加新请求和清空队列，当然，还要能够刷新队列。此外还应该可以从队列中删除请求，这种操作称为出列（*dequeue*）：

```

114 DED.Queue = function() {
  // Queued requests.
  this.queue = [];

  // Observable Objects that can notify the client of interesting moments
  // on each DED.Queue instance.
  this.onComplete = new DED.util.Observer();
  this.onFailure = new DED.util.Observer();
  this.onFlush = new DED.util.Observer();

  // Core properties that set up a frontend queueing system.
  this.retryCount = 3;
  this.currentRetry = 0;
  this.paused = false;
  this.timeout = 5000;
  this.conn = {};
  this.timer = {};
}

```

```

};

DED.Queue.

method('flush', function() {
  if (!this.queue.length > 0) {
    return;
  }
  if (this.paused) {
    this.paused = false;
    return;
  }
  var that = this;
  this.currentRetry++;
  var abort = function() {
    that.conn.abort();
    if (that.currentRetry == that.retryCount) {
      that.onFailure.fire();
      that.currentRetry = 0;
    } else {
      that.flush();
    }
  };
  this.timer = window.setTimeout(abort, this.timeout);
  var callback = function(o) {
    window.clearTimeout(that.timer);
    that.currentRetry = 0;
    that.queue.shift();
    that.onFlush.fire(o.responseText);
    if (that.queue.length == 0) {
      that.onComplete.fire();
      return;
    }
    // recursive call to flush
    that.flush();
  };
  this.conn = asyncRequest(
    this.queue[0]['method'],
    this.queue[0]['uri'],
    callback,
    this.queue[0]['params']
  );
}).

method('setRetryCount', function(count) {
  this.retryCount = count;
}).

method('setTimeout', function(time) {
  this.timeout = time;
}).

method('add', function(o) {
  this.queue.push(o);
});

```

```

    }).
    method('pause', function() {
      this.paused = true;
    }).
    method('dequeue', function() {
      this.queue.pop();
    }).
    method('clear', function() {
      this.queue = [];
    });
  );
}

```

这段代码乍一看可能有点令人望而却步，不过你可以快速浏览一下DED.Queue类，看看其主要方法：flush、setRetryCount、setTimeout、add、pause、dequeue和clear。queue属性是一个数组字面量，用于保存对每一个请求的引用。add和dequeue这类方法所做的只是对这个数组进行push和pop操作。flush方法则会把请求发送出去并将它们移出数组。

#### 8.4.4 实现队列

队列系统的实现如下：

```

var q = new DED.Queue;
// Reset our retry count to be higher for slow connections.
q.setRetryCount(5);
// Decrease timeout limit because we still want fast connections to benefit.
q.setTimeout(1000);
// Add two slots.
q.add({
  method: 'GET',
  uri: '/path/to/file.php?ajax=true'
});
q.add({
  method: 'GET',
  uri: '/path/to/file.php?ajax=true&woe=me'
});
// Flush the queue.
q.flush();
// Pause the queue, retaining the requests.
q.pause();
// Clear our queue and start fresh.
q.clear();
// Add two requests.
q.add({
  method: 'GET',
  uri: '/path/to/file.php?ajax=true'
});
q.add({
  method: 'GET',
  uri: '/path/to/file.php?ajax=true&woe=me'
});
// Remove the last request from the queue.

```

```

q.dequeue();
// Flush the queue again.
q.flush();

```

迄今为止一切还算顺利。这个队列应该不难理解。但现在你可能会寻思其中有哪些地方用了桥接模式。实际上到目前为止还没有用到这个模式。但到了实现的时候，你会看到桥接元素无处不在。下面的代码示范了客户系统的实现：

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Ajax Connection Queue</title>
    <script src="utils.js"></script>
    <script src="queue.js"></script>
    <script type="text/javascript">
      addEvent(window, 'load', function() {
        // Implementation.
        var q = new DED.Queue();
        q.setRetryCount(5);
        q.setTimeout(3000);

        var items = $('#items');
        var results = $('#results');
        var queue = $('#queue-items');

        // Keeping track of my own requests as a client.
        var requests = [];

        // Notifier for each request that is being flushed.
        q.onFlush.subscribe(function(data) {
          results.innerHTML = data;
          requests.shift();
          queue.innerHTML = requests.toString();
        });
        // Notifier for any failures.
        q.onFailure.subscribe(function() {
          results.innerHTML += ' <span style="color:red;">Connection Error!</span>';
        });
        // Notifier of the completion of the flush.
        q.onComplete.subscribe(function() {
          results.innerHTML += ' <span style="color:green;">Completed!</span>';
        });
        var actionDispatcher = function(element) {
          switch (element) {
            case 'flush':
              q.flush();
              break;
            case 'dequeue':

```

```

        q.dequeue();
        requests.pop();
        queue.innerHTML = requests.toString();
        break;
    case 'pause':
        q.pause();
        break;
    case 'clear':
        q.clear();
        requests = [];
        queue.innerHTML = '';
        break;
    }
};

var addRequest = function(request) {
    var data = request.split('-')[1];
    q.add({
        method: 'GET',
        uri: 'bridge-connection-queue.php?ajax=true&s=' + data,
        params: null
    });
    requests.push(data);
    queue.innerHTML = requests.toString();
};

addEvent(items, 'click', function(e) {
    var e = e || window.event;
    var src = e.target || e.srcElement;
    try {
        e.preventDefault();
    }
    catch (ex) {
        e.returnValue = false;
    }
    actionDispatcher(src.id);
});

var adders = $('#adders');
addEvent(adders, 'click', function(e) {
    var e = e || window.event;
    var src = e.target || e.srcElement;
    try {
        e.preventDefault();
    }
    catch (ex) {
        e.returnValue = false;
    }
    addRequest(src.id);
});
});

```

```
</script>
<style type="text/css" media="screen">
    body { font: 100% georgia,times,serif; }
    h1, h2 { font-weight: normal; }
    #queue-items { height: 1.5em; }
    #add-stuff {
        padding: .5em;
        background: #ddd;
        border: 1px solid #bbb;
    }
    #results-area { padding: .5em; border: 1px solid #bbb; }
</style>
</head>
<body id="example">
    <div id="doc">
        <h1>Ajax Connection Queue</h1>
        <div id="queue-items"></div>
        <div id="add-stuff">
            <h2>Add Requests to Queue</h2>
            <ul id="adders">
                <li><a href="#" id="action-01">Add "01" to Queue</a></li>
                <li><a href="#" id="action-02">Add "02" to Queue</a></li>
                <li><a href="#" id="action-03">Add "03" to Queue</a></li>
            </ul>
        </div>
        <h2>Other Queue Actions</h2>
        <ul id="items">
            <li><a href="#" id="flush">Flush</a></li>
            <li><a href="#" id="dequeue">Dequeue</a></li>
            <li><a href="#" id="pause">Pause</a></li>
            <li><a href="#" id="clear">Clear</a></li>
        </ul>
        <div id="results-area">
            <h2>Results: </h2>
            <div id="results"></div>
        </div>
    </div>
</body>
</html>
```

这段代码会生成一个相当朴实的用户界面，如图8-1所示。

图8-1中顶部的区域用于向DED.Queue添加新请求，底部的区域用于执行其他方法。往队列中加入一些请求之后，你会看到大致如图8-2所示的画面。

反复点击Dequeue这个链接，删除DED.Queue实例中最后的3个请求。结果如图8-3所示。

点击Flush这个链接，送出两个请求，然后点击Pause。结果如图8-4所示。

所有请求处理完毕之后，Results区域会提示：队列处理完毕，所有请求均已发出。注意图8-5中02是队列中最后一个送出的请求。

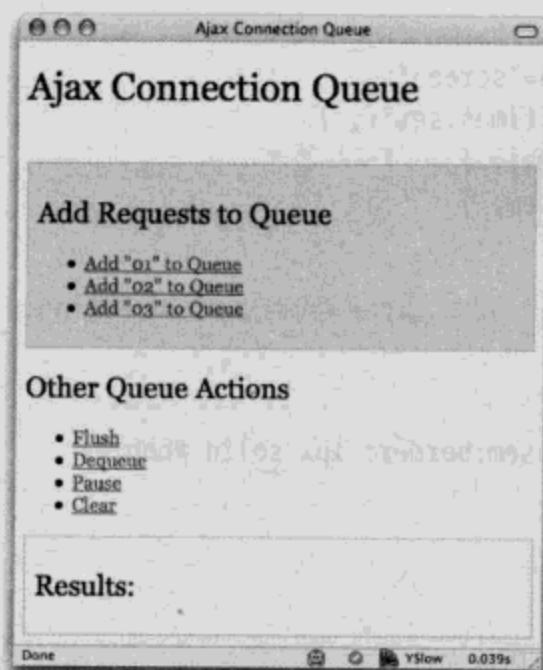


图 8-1

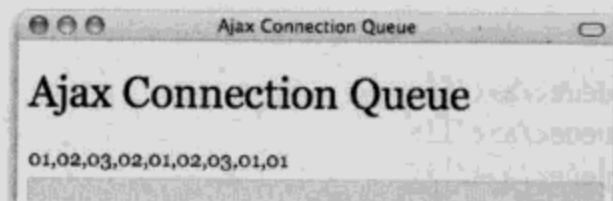


图 8-2

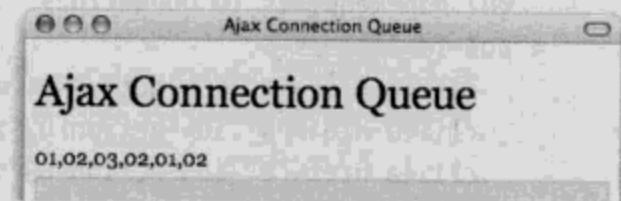


图 8-3

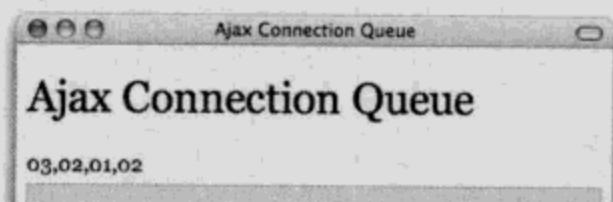


图 8-4

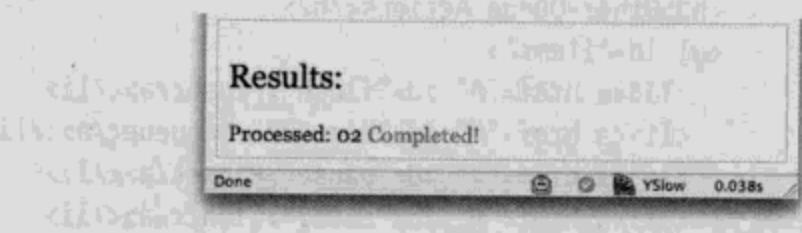


图 8-5

#### 8.4.5 哪些地方用了桥接模式

这个应用程序中到处都有桥接元素的身影。既然要设计一个无可挑剔的队列接口，那就得在所有恰当的地方用上桥接模式。最明显的是，事件监听器回调函数并不直接与队列打交道，而是使用了桥接函数，这些桥接函数控制着动作工厂并料理数据输入事宜。

不过，有一个地方可以再做些改进。用户点击一个链接以添加一个请求的时候，代码先要执行一些基本逻辑，然后把所点击的元素的ID传递给addRequest函数。但这个参数值并不是addRequest函数所期待的。它需要的应该是一个普通的数值ID而不是混合型的字符串。因此，我们可以把代码从这个形式：

```
var addRequest = function(request) {
  var data = request.split('-')[1];
  // etc...
};
```

改为这个形式：

```
var addRequest = function(data) {  
    // etc...  
};
```

毕竟，addRequest函数实际上只是要查一下数据中有些什么而已，不是吗？现在可以添加一个中间性的桥梁函数：

```
var addRequestFromClick = function(request) {  
    addRequest(request.split('-')[0]);  
};
```

在供用户执行刷新和暂停操作的部分，我们提供了一个动作调度函数，其作用就是桥接用户操作所包含的输入信息并将其委托给恰当的处理代码。在DOM脚本编程中这种技术也称为事件委托（event delegation）。click事件所代表的用户操作与DED.Queue实现完全分开的结果就是，后者的方法并不直接与那些事件耦合在一起，因此你可以在任何地方执行这些方法。你可以在JavaScript控制台命令行环境中调用它们，也可以对它们进行单元测试。它们也可以被关联到mouseover或focus等其他浏览器事件。其具体用法五花八门，客户系统只需提供桥接性元素即可。

## 8.5 桥接模式的适用场合

很难想象，不使用桥接模式的事件驱动编程会是什么样子。但是JavaScript编程新手们常常沉迷于事件驱动开发的函数式风格，忘了编写接口——哪怕面对的是复杂操作。判断什么地方应该使用桥接模式通常很简单。假如有下面的代码：

```
$('#example').onclick = function() {  
    new RichTextEditor();  
};
```

从中你无法看出那个编辑器要显示在什么地方、它有些什么配置选项以及应该怎样修改它。这里的要诀是要让接口“可桥接（bridgeable）”，实际上也就是可适配（adaptable。参见第11章）。

在现实生活中，桥梁对于城市建设与城中街道的连通至关重要。城区相当于模块，而街道则相当于把它们连接在一起的方法。道路的可用性往往影响着该片区的人口数量。同样，你向客户提供的接口极有可能影响到模块的受欢迎程度。

## 8.6 桥接模式之利

掌握如何在软件开发中实现桥接模式，受益的不只是你，还有那些负责维护你的作品的人。把抽象与其实现隔离开，有助于独立地管理软件的各组成部分。Bug也因此更容易查找，而软件发生严重故障的可能性也减小了。说到底，桥接元素应该是粘合每一个抽象的粘合因子。

## 8.7 桥接模式之弊

在我们看来，这种模式并没有多少真正的缺点。前面讲述它的优点的时候已经提过，它只会让API更加健壮、提高组件的模块化程度并促成更简洁的客户系统实现。不过，这些益处的确是

有代价的。每使用一个桥接元素都要增加一次函数调用，这对应用程序的性能会有一些负面影响。此外，它们也提高了系统的复杂程度，在出现问题时这会导致代码更难调试。大多数情况下桥接模式都非常有用，但注意不要滥用。举个例来说，如果一个桥接函数被用于连接两个函数，而其中某个函数根本不会在桥接函数之外被调用，那么此时这个桥接函数就不是非要不可，你可以放心将它删除。

## 8.8 小结

用GoF的话来说，桥接模式“将抽象与其实现隔离开来，以便二者独立变化”。它可以促进代码的模块化、促成更简洁的实现并提高抽象的灵活性。它可以用来把一组类和函数连接起来，而且提供了一种借助于特权函数访问私用数据的手段。

**组**合模式是一种专为创建Web上的动态用户界面而量身定制的模式。使用这种模式，可以用一条命令在多个对象上激发复杂的或递归的行为。这可以简化粘合性代码，使其更容易维护，而那些复杂行为则被委托给各个对象。

组合模式为操劳过度的JavaScript程序员带来了两大好处。

(1) 你可以用同样的方法处理对象的集合与其中的特定子对象。组合对象（composite）与组成它的对象实现了同一批操作。对组合对象执行的这些操作将向下传递到所有的组成对象（constituent object），这样一来所有的组成对象都会执行同样的操作。在存在大批对象的情况下，这是一种非常有效的技术。藉此你可以不着痕迹地用一组对象替换一个对象，反之亦然。这有助于弱化各个对象之间的耦合。

(2) 它可以用来把一批子对象组织成树形结构，并且使整棵树都可被遍历。所有组合对象都实现了一个用来获取其子对象的方法。借助这个方法，你可以隐藏实现的细节并随心所欲地组织子对象。任何使用这个对象的代码都不会对其内部实现形成依赖。

本章将示范在JavaScript中实现组合模式的方法，并讨论其适用场合。

125

## 9.1 组合对象的结构

如图9-1所示，在组合对象的层次体系中有两种类型的对象：叶对象和组合对象。这是一个递归定义，但这正是组合模式如此有用的原因所在。一个组合对象由一些别的组合对象和叶对象组成。其中只有叶对象不再包含子对象。叶对象是组合对象中最基本的元素，也是各种操作的落实地点。

## 9.2 使用组合模式

只有同时具备如下两个条件时才适合使用组合模式：

- 存在一批组织成某种层次体系的对象（具体

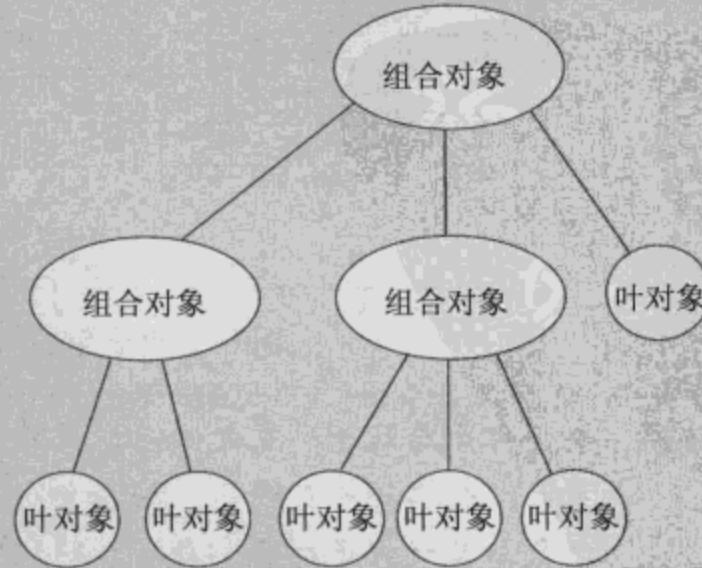


图9-1 组合设计模式的基本结构

的结构在开发期间可能无法得知)。

- 希望对这批对象或其中的一部分对象实施一个操作。

组合模式擅长于对大批对象进行操作。它专为组织这类对象并把操作从一个层次向下一层次传递而设计。藉此你可以弱化对象间的耦合并可互换地使用一些类或实例。按这种模式编写的代码模块化程度更高, 也更容易维护。

### 9.3 示例: 表单验证

假设你接手了一个新项目。乍一看任务很简单: 创建一个表单, 要求可以保存、恢复和验证其中的值。随便一个半吊子Web开发人员都能搞定, 不是吗? 问题在于, 这个表单中元素的内容和数目都是完全未知的, 而且会因用户而异。图9-2显示了一个典型的例子。那种紧密耦合到Name和Address这类特定表单域的validate函数不会管用, 因为在开发期间无法得知要验证哪些域。这正是组合模式可以大显身手的地方。

First name:	<input type="text"/>
Last name:	<input type="text"/>
Address:	<input type="text"/>
City:	<input type="text"/>
State:	<input type="text"/>
ZIP code:	<input type="text"/>
Day phone:	<input type="text"/>
Evening phone:	<input type="text"/>

First name:	<input type="text"/>
Last name:	<input type="text"/>
Email address:	<input type="text"/>
Website:	<input type="text"/>
Day phone:	<input type="text"/>
Evening phone:	<input type="text"/>
Mobile phone:	<input type="text"/>
Fax:	<input type="text"/>

图9-2 不同用户可能会看到不同的表单

首先, 我们应该逐一鉴别表单的各个组成元素, 判断它属于组合对象还是叶对象(参见图9-3)。表单最基本的构成要素是用户用以输入数据的域, 它们由input、select和textarea标签生成。用于组织相关域的fieldset标签属于上面的一个层次。位于最顶层的是表单自身。

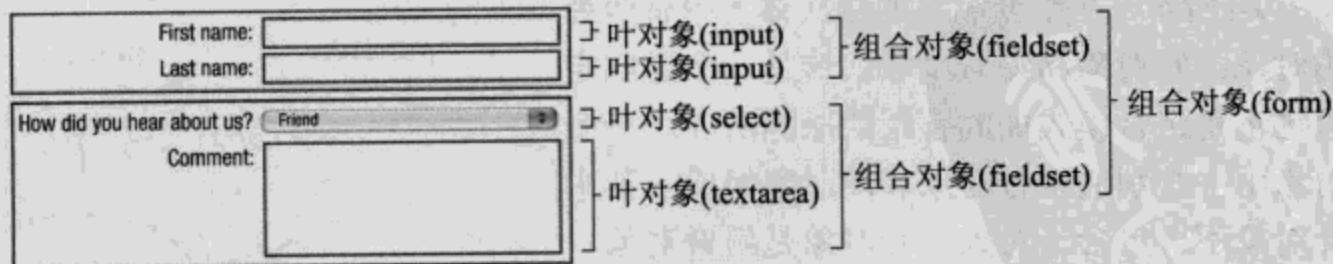


图9-3 把表单的基本元素分为组合对象和叶对象

**注解** 组合对象与其子对象之间应该具有一种具一(HAS-A)关系, 而不是一种为一(IS-A)关系。表单包含着域集(fieldset), 而后者又包含着域。域并不是域集的子类。由于组合对象与其所有子对象都具有相同的接口, 这可能会诱使人们把它们看成是超类和子类的关系, 但事实并非如此。叶对象并没有继承其上一级组合对象。

我们的第一个任务是创建一个动态表单并实现save和validate操作。表单中实际拥有的域会因用户而异，所以单单save或validate函数是不可能满足每一个人的需要的。我们希望设计一个模块化的表单，以便将来任何时候都能为其添加各种元素，而不用修改save和validate函数。

我们不想为表单元素的每一种可能组合编写一套方法，而是决定让这两个方法与表单域自身关联起来。也就是说，让每个域都知道如何保存和验证自己：

```
nameFieldset.validate();
nameFieldset.save();
...
```

这里的难点在于如何同时在所有域上执行这些操作。我们不想使用迭代结构的代码一一访问那些数目未知的域，而是打算用组合模式来简化代码。要保存所有域，只需这样一次调用即可：

```
topForm.save();
```

topForm对象将在其所有子对象上递归调用save方法。实际的save操作只会发生在底层的叶对象上。组合对象只起到一个传递调用的作用。现在你已经大体了解了组合对象的组织方式，下面该看看其实现代码了。

首先要做的是创建那些组合对象和叶对象需要实现的两个接口：

```
var Composite = new Interface('Composite', ['add', 'remove', 'getChild']);
var FormItem = new Interface('FormItem', ['save']);
```

目前FormItem接口只要求实现一个save函数，但稍后我们还会对其进行扩充。图9-4显示了待实现的类的UML类图。

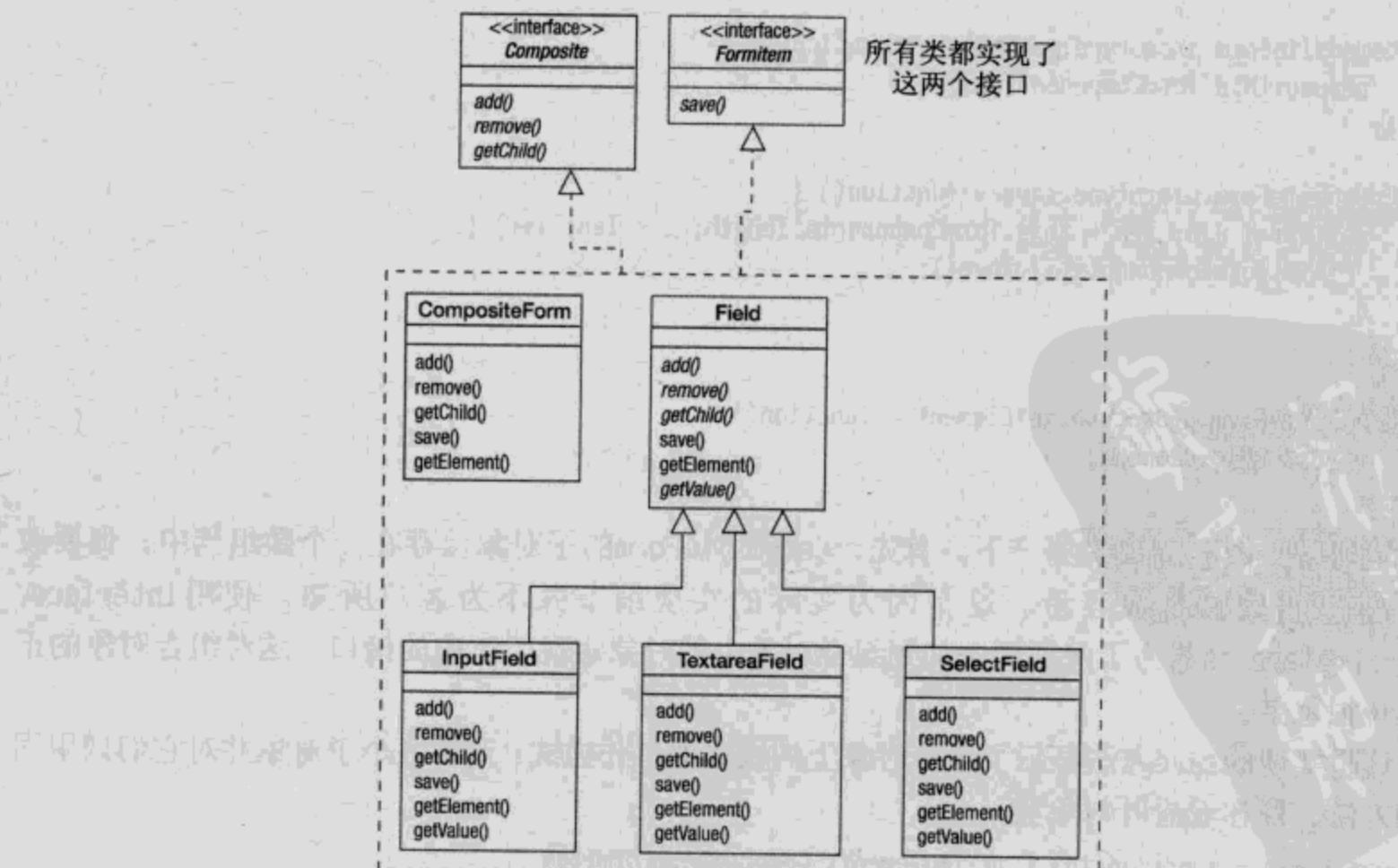


图9-4 需要实现的类

CompositeForm类的代码如下：

```

var CompositeForm = function(id, method, action) { // implements Composite, FormItem
    this.formComponents = [];

    this.element = document.createElement('form');
    this.element.id = id;
    this.element.method = method || 'POST';
    this.element.action = action || '#';
};

CompositeForm.prototype.add = function(child) {
    Interface.ensureImplements(child, Composite, FormItem);
    this.formComponents.push(child);
    this.element.appendChild(child.getElement());
};

CompositeForm.prototype.remove = function(child) {
    for(var i = 0, len = this.formComponents.length; i < len; i++) {
        if(this.formComponents[i] === child) {
            this.formComponents.splice(i, 1); // Remove one element from the array at
  // position i.
            break;
        }
    }
};

CompositeForm.prototype.getChild = function(i) {
    return this.formComponents[i];
};

CompositeForm.prototype.save = function() {
    for(var i = 0, len = this.formComponents.length; i < len; i++) {
        this.formComponents[i].save();
    }
};

CompositeForm.prototype.getElement = function() {
    return this.element;
};

```

这里有些东西需要说明一下。首先，CompositeForm的子对象保存在一个数组当中，但要改用别的数据结构也很容易。这是因为实际的实现细节并不为客户所知。使用Interface.ensureImplements是为了保证要添加到组合对象中的对象实现了正确的接口。这对组合对象的正常运转很重要。

这里实现的save方法显示了组合对象上的操作的工作方式：遍访各个子对象并对它们调用同样的方法。现在看看叶对象类：

```

var Field = function(id) { // implements Composite, FormItem
    this.id = id;

```

```

    this.element;
};

Field.prototype.add = function() {};
Field.prototype.remove = function() {};
Field.prototype.getChild = function() {};

Field.prototype.save = function() {
    setCookie(this.id, this.getValue());
};
Field.prototype.getElement = function() {
    return this.element;
};

Field.prototype.getValue = function() {
    throw new Error('Unsupported operation on the class Field.');
};

```

这个类将被各个叶对象类继承。它将Composite接口中的方法实现为空函数，这是因为叶节点不会有子对象。你也可以考虑让这几个函数抛出异常。

**警告** 这里用最简单的方式实现了save方法。实际上把用户的原始数据存放在Cookie中是一个非常糟糕的做法。其原因有多个。首先，用户计算机上的Cookie很容易被篡改，所以数据的有效性无法得到保证。其次，存储在Cookie中的数据有大小限制，因此用户的数据可能不会被全部保存下来。再者，这样做也会影响到性能，这是因为在每次请求中Cookie都会作为HTTP头被一起发送。

save方法用getValue方法获得所要保存的对象值，后一方法在各个子类中的实现各不相同。使用save方法，不用提交表单也能保存表单的内容。对于那种长长的表单来说这尤其有用，因为用户可以把数据保存下来，等以后再回来完成表单的填写：

```

var InputField = function(id, label) { // implements Composite, FormItem
    Field.call(this, id);

    this.input = document.createElement('input');
    this.input.id = id;

    this.label = document.createElement('label');
    var labelTextNode = document.createTextNode(label);
    this.label.appendChild(labelTextNode);

    this.element = document.createElement('div');
    this.element.className = 'input-field';
    this.element.appendChild(this.label);
    this.element.appendChild(this.input);
};

extend(InputField, Field); // Inherit from Field.

```

```
InputField.prototype.getValue = function() {
    return this.input.value;
};
```

InputField是Field的子类之一。它的大多数方法都是从Field继承而来，但它也实现了针对input标签的getValue方法的代码。TextareaField和SelectField也实现了自己特有的getValue方法：

```
var TextareaField = function(id, label) { // implements Composite, FormItem
    Field.call(this, id);

    this.textarea = document.createElement('textarea');
    this.textarea.id = id;

    this.label = document.createElement('label');
    var labelTextNode = document.createTextNode(label);
    this.label.appendChild(labelTextNode);

    this.element = document.createElement('div');
    this.element.className = 'input-field';
    this.element.appendChild(this.label);
    this.element.appendChild(this.textarea);
};

extend(TextareaField, Field); // Inherit from Field.

TextareaField.prototype.getValue = function() {
    return this.textarea.value;
};

var SelectField = function(id, label) { // implements Composite, FormItem
    Field.call(this, id);

    this.select = document.createElement('select');
    this.select.id = id;

    this.label = document.createElement('label');
    var labelTextNode = document.createTextNode(label);
    this.label.appendChild(labelTextNode);

    this.element = document.createElement('div');
    this.element.className = 'input-field';
    this.element.appendChild(this.label);
    this.element.appendChild(this.select);
};

extend(SelectField, Field); // Inherit from Field.

SelectField.prototype.getValue = function() {
    return this.select.options[this.select.selectedIndex].value;
};
```

### 9.3.1 汇合起来

这是组合模式大放光彩的地方。无论有多少表单域，对整个组合对象执行操作只需一个函数调用即可：

```
var contactForm = new CompositeForm('contact-form', 'POST', 'contact.php');

contactForm.add(new InputField('first-name', 'First Name'));
contactForm.add(new InputField('last-name', 'Last Name'));
contactForm.add(new InputField('address', 'Address'));
contactForm.add(new InputField('city', 'City'));
contactForm.add(new SelectField('state', 'State', stateArray));
// var stateArray =[{'al', 'Alabama'}, ...];
contactForm.add(new InputField('zip', 'Zip'));
contactForm.add(new TextareaField('comments', 'Comments'));

addEvent(window, 'unload', contactForm.save);
```

可以把`save`的调用绑定到某个事件上，也可以用`setInterval`周期性地调用这个函数。为这个组合对象添加其他操作也很简单。如下一节所示，验证数据、恢复先前保存的数据以及将表单重设为默认状态这些操作都可以如法炮制。

### 9.3.2 向 FormItem 添加操作

现在基本框架已经设计好了，要为`FormItem`接口添加其他操作很容易。首先是修改这个接口：

```
var FormItem = new Interface('FormItem', ['save', 'restore']);
```

然后是在叶对象类上实现这些操作。在本例中只要为超类`Field`添加这些操作以供那些子类继承即可：

```
Field.prototype.restore = function() {
  this.element.value = getCookie(this.id);
};
```

最后，为组合对象类添加同样的操作：

```
CompositeForm.prototype.restore = function() {
  for(var i = 0, len = this.formComponents.length; i < len; i++) {
    this.formComponents[i].restore();
  }
};
```

在实现中加入下面这行代码后就可以在窗口加载时恢复所有表单域的值：

```
addEvent(window, 'load', contactForm.restore);
```

### 9.3.3 向层次体系中添加类

到目前为止只有一个组合对象类。如果设计目标要求对操作的调用有更多粒度上的控制，那么，可以添加更多层次的组合对象类，而不必改变其他类。假设我们想要对表单的某些部分执行

save和restore操作，而不影响到其他部分，有一个解决办法是逐一在各个域上执行这些操作：

```
firstName.restore();
lastName.restore();
...
...
```

但在不知道表单具体会有哪些域的情况下，这种方法并不管用。在层次体系中创建新的层次是一个更好的选择。我们可以把域组织在域集（fieldset）中，每一个域集都是一个实现了FormItem接口的组合对象。在域集上调用restore将导致在其所有子对象上调用restore。

创建CompositeFieldset类并不要求为此修改其他类。因为composite接口隐藏了所有内部实现细节，我们可以自由选用某种数据结构来存储子对象。作为示范，我们在此将使用一个对象来存储CompositeFieldset的子对象，而不是像CompositeForm那样使用数组：

```
var CompositeFieldset = function(id, legendText) { // implements Composite, FormItem
    this.components = {};

    this.element = document.createElement('fieldset');
    this.element.id = id;

    if(legendText) { // Create a legend if the optional second
        // argument is set.
        this.legend = document.createElement('legend');
        this.legend.appendChild(document.createTextNode(legendText));
        this.element.appendChild(this.legend);
    }
};

CompositeFieldset.prototype.add = function(child) {
    Interface.ensureImplements(child, Composite, FormItem);
    this.components[child.getElement().id] = child;
    this.element.appendChild(child.getElement());
};

CompositeFieldset.prototype.remove = function(child) {
    delete this.components[child.getElement().id];
};

CompositeFieldset.prototype.getChild = function(id) {
    if(this.components[id] != undefined) {
        return this.components[id];
    }
    else {
        return null;
    }
};

CompositeFieldset.prototype.save = function() {
    for(var id in this.components) {
        if(!this.components.hasOwnProperty(id)) continue;
        this.components[id].save();
    }
};
```

```

};

CompositeFieldset.prototype.restore = function() {
    for(var id in this.components) {
        if(!this.components.hasOwnProperty(id)) continue;
        this.components[id].restore();
    }
};

CompositeFieldset.prototype.getElement = function() {
    return this.element;
};

```

CompositeFieldset的内部细节与CompositeForm截然不同，但是因为它与其他类实现了同样的接口，所以也能用在组合当中。只要对实现代码做少量修改即可获得这个新功能：

```
var contactForm = new CompositeForm('contact-form', 'POST', 'contact.php');
```

```

var nameFieldset = new CompositeFieldset('name-fieldset');
nameFieldset.add(new InputField('first-name', 'First Name'));
nameFieldset.add(new InputField('last-name', 'Last Name'));
contactForm.add(nameFieldset);

var addressFieldset = new CompositeFieldset('address-fieldset');
addressFieldset.add(new InputField('address', 'Address'));
addressFieldset.add(new InputField('city', 'City'));
addressFieldset.add(new SelectField('state', 'State', stateArray));
addressFieldset.add(new InputField('zip', 'Zip'));
contactForm.add(addressFieldset);

```

```
contactForm.add(new TextareaField('comments', 'Comments'));
```

```
body.appendChild(contactForm.getElement());
```

```

addEvent(window, 'unload', contactForm.save);
addEvent(window, 'load', contactForm.restore);

```

```

addEvent('save-button', 'click', nameFieldset.save);
addEvent('restore-button', 'click', nameFieldset.restore);

```

现在我们用域集对一部分域进行了组织。也可以直接把域加入表单之中（那个评论文本框就是一例），这是因为表单不在乎其子对象究竟是组合对象还是叶对象，只要它们实现了恰当的接口就行。在contactForm上执行任何操作仍然会导致在其所有子对象上（继而又在这些子对象的子对象上）执行同样的操作，因此系统并没有损失什么功能。这样做的收获是在表单的一个子集上执行这些操作的能力。

### 9.3.4 添加更多操作

前面已经开了一个好头。用同样的方法还可以添加更多操作。可以为Field的构造函数增加

一个参数，用以表明该域是否必须填写，然后基于这个属性实现一个验证方法。可以修改restore方法，以便在域没有保存过数据的情况下将其值设置为默认值。甚至还可以添加一个submit方法，用Ajax请求把所有的值发送到服务器端。由于使用了组合模式，添加这些操作并不需要知道表单具体是什么样子。

## 9.4 示例：图片库

在表单的例子中，由于HTML的限制，组合模式并没有得到充分利用。例如，你不能在表单中嵌套表单，而只能嵌套域集。真正的组合对象是可以内嵌在同类对象之中的。现在我们研究的是使用组合模式构建用户界面的另一个案例，但在这个示例中，任何位置都可以换用任何对象。同样，本例中也要用JavaScript对象来包装HTML元素。

这次的任务是创建一个图片库。我们希望能有选择地隐藏或显示图片库的特定部分。这可能是单独的图片，也可能是图片库。其他操作以后还可以添加，现在我们只关注hide和show操作。需要的类只有两个：用作图片库的组合对象类和用于图片本身的叶对象类：

```
var Composite = new Interface('Composite', ['add', 'remove', 'getChild']);
var GalleryItem = new Interface('GalleryItem', ['hide', 'show']);

// DynamicGallery class.

var DynamicGallery = function(id) { // implements Composite, GalleryItem
    this.children = [];

    this.element = document.createElement('div');
    this.element.id = id;
    this.element.className = 'dynamic-gallery';
}

DynamicGallery.prototype = {
    // Implement the Composite interface.

    add: function(child) {
        Interface.ensureImplements(child, Composite, GalleryItem);
        this.children.push(child);
        this.element.appendChild(child.getElement());
    },
    remove: function(child) {
        for(var node, i = 0; node = this.getChildAt(i); i++) {
            if(node == child) {
                this.children.splice(i, 1);
                break;
            }
        }
        this.element.removeChild(child.getElement());
    },
}
```

```

get Child: function(i) {
    return this.children[i];
},
// Implement the GalleryItem interface.

hide: function() {
    for(var node, i = 0; node = this.getChild(i); i++) {
        node.hide();
    }
    this.element.style.display = 'none';
},
show: function() {
    this.element.style.display = 'block';
    for(var node, i = 0; node = this.getChild(i); i++) {
        node.show();
    }
},
// Helper methods.

getElement: function() {
    return this.element;
}
};

```

在上面的代码中，首先定义的是组合对象类和叶对象类应该实现的接口。除了常规的组合对象方法外，这些类要定义的操作只包括hide和show。接下来定义的是组合对象类。由于DynamicGallery只是对div元素的包装，所以图片库中可以再嵌套图片库，而我们因此也就只需要用到一个组合对象类。

这里为DynamicGallery的prototype设置方法的做法与前例略有不同。我们没有按DynamicGallery.prototype.methodName的形式设置方法，而是把一个对象字面量赋给prototype属性，这个对象包含了所有要设置的方法。在一次性定义多个方法时这种做法的好处在于不必在每个方法名之前都加上DynamicGallery.prototype。以后添加其他方法时，我们还是可以继续使用原来那种更啰嗦一些的做法的。

你也许很想用DOM自身作为保存子元素的数据结构。它已经拥有addChild和removeChild方法，还有childNodes属性，对于存储和获取组合对象的子对象来说这原本非常理想。问题在于这种做法要求每个相关DOM节点都要具有一个反指其包装对象的引用，以便实现所要求的操作。而在某些浏览器中这会导致内存泄漏。一般说来，最好避免让DOM对象反过来引用JavaScript对象。出于这种考虑，本例中使用一个数组来保存子对象。

叶节点也非常简单。它是对image元素的包装，并且实现了hide和show方法：

```

// GalleryImage class.

var GalleryImage = function(src) { // implements Composite, GalleryItem

```

```

this.element = document.createElement('img');
this.element.className = 'gallery-image';
this.element.src = src;
}

GalleryImage.prototype = {

    // Implement the Composite interface.

    add: function() {},      // This is a leaf node, so we don't
    remove: function() {},   // implement these methods, we just
    getChild: function() {}, // define them.

    // Implement the GalleryItem interface.

    hide: function() {
        this.element.style.display = 'none';
    },
    show: function() {
        this.element.style.display = ''; // Restore the display attribute to its
   // previous setting.
    },

    // Helper methods.

    getElement: function() {
        return this.element;
    }
};

```

这是一个演示组合模式的工作方式的好例子。每个类都很简单，但由于有了这样一种层次体系，我们就可以执行一些复杂操作。GalleryImage类的构造函数会创建一个img元素。这个类定义中的其余部分由空的组合对象方法（因为这是叶节点）和GalleryItem要求的操作组成。现在我们可以使用这两个类来管理图片：

```

var topGallery = new DynamicGallery('top-gallery');

topGallery.add(new GalleryImage('/img/image-1.jpg'));
topGallery.add(new GalleryImage('/img/image-2.jpg'));
topGallery.add(new GalleryImage('/img/image-3.jpg'));

var vacationPhotos = new DynamicGallery('vacation-photos');

for(var i = 0; i < 30; i++) {
    vacationPhotos.add(new GalleryImage('/img/vac/image-' + i + '.jpg'));
}

topGallery.add(vacationPhotos);
topGallery.show();      // Show the main gallery,
vacationPhotos.hide(); // but hide the vacation gallery.

```

在组织图片时，DynamicGallery这个组合对象类你想用多少次都行。因为由DynamicGallery实例化的组合对象可以嵌套在同类对象中，所以只用这两个类的实例就能建造出任意大的层次体系。你也可以对这个层次体系中的任意部分执行各种操作。在正确建造好图片库的层次体系之后，像“显示在海边和山区的度假照片，但不包括2004年的”这样的操作，只要几行代码即可实现。

## 9.5 组合模式之利

使用组合模式，简单的操作也能产生复杂的结果。你不必编写大量手工遍历数组或其他数据结构的粘合代码，只需对最顶层的对象执行操作，让每一个子对象自己传递这个操作即可。这对于那些再三执行的操作尤其有用。

在组合模式中，各个对象之间的耦合非常松散。只要它们实现了同样的接口，那么改变它们的位置或互换它们只是举手之劳。这促进了代码的重用，也有利于代码重构。

用组合模式组织起来的对象形成了一个出色的层次体系。每当对顶层组合对象执行一个操作时，实际上是在对整个结构进行深度优先的搜索以查找节点。而创建组合对象的程序员对这些细节一无所知。在这个层次体系中添加、删除和查找节点都非常容易。

## 9.6 组合模式之弊

组合对象的易用性可能掩盖了它所支持的每一种操作的代价。由于对组合对象调用的任何操作都会被传递到它的所有子对象，如果这个层次体系很大的话，系统的性能将会受到影响。程序员可能一时还不会察觉到：像topGallery.show()这样一个方法调用会引起一次对整个树结构的遍历。在文档中说明一下这个情况很有必要。

在本章的两个示例中，组合对象类和节点类都被用作HTML元素的包装工具，这只是该模式的用法之一，但也是一种常见的用法。在此情况下，组合对象也必须遵守HTML的使用规则。例如，表格就很难转化为一个组合对象，这是因为表格标签中只能包含几种特定的标签。而且，此时的叶节点并不是那么显而易见。表格单元格本也可以被看作叶节点，但它们内部也可能会包含一些其他元素。这些限制降低了组合对象的有用性，也有损代码的模块性。按这种方式使用组合模式时，一定要权衡一下其利弊。139

组合模式的正常运作需要用到某种形式的接口。接口检查越严格，组合对象类也就越可靠。这会为系统增加一些复杂性，但不会太多。如果系统中已经使用了某种形式的接口或鸭式辨型（比如Interface类），那就没问题。否则就应该在代码中加入类型检查。

## 9.7 小结

如果应用得当，那么组合模式是一种非常管用的模式。它把一批子对象组织为树型结构，只要一条命令就可以操作树中的所有对象。它提高了代码的模块化程度，而且便于代码重构和对象的更换。这种模式特别适合于动态的HTML用户界面，在它的帮助下，你可以在不知道用户界面的最终格局的情况下进行开发。对于每一个JavaScript程序员，它都是最有用的模式之一。140

**门**面模式有两个作用：一是简化类的接口；二是消除类与使用它的客户代码之间的耦合。在JavaScript中，门面模式常常是开发人员最亲密的朋友。它是几乎所有JavaScript库的核心原则。通过创建一些便利方法让复杂系统变得更加简单易用，门面模式可以使库提供的工具更容易理解。使用这种模式，程序员可以间接地与一个子系统打交道，与直接访问子系统相比，这样做更不容易出错。

门面模式简化了诸如错误记录或跟踪页面视图统计数据这类常用的或重复性的任务。通过添加一些便利方法（这种方法是对原有的一些方法的组合利用），它还可以让对象的功能显得更加完善。

门面模式可用于简化复杂接口。它可以在幕后为你进行错误检查、清除不再需要的大对象，以及用一种更加易用的方式展现对象的功能。

门面模式并非必不可少。同样的任务不用它也能完成。这是一种组织性的模式，它可以用来修改类和对象的接口，使其更便于使用。它可以让程序员过得更轻松，使他们的代码变得更容易管理。

## 10.1 一些你可能已经知道的门面元素

想想计算机桌面上的那些快捷方式图标，它们就是在扮演一个把用户引导至某个地方的接口的角色，如果不借助于它们的话，那些地方找起来会很麻烦。对于那种嵌得比较深的文件或目录，如果每次使用时都要去找一次，实在是一件令人厌烦的事。基于GUI的操作系统就是计算机上的数据和功能的一个门面。每次点击、拖动和移动某个东西时，实际上是在跟一个门面打交道，间接地执行一些幕后的命令。

鉴于你可能已经有过一些JavaScript使用经验并且在网上找过适用于各种浏览器的处理事件监听器的方法，也许见过这样的代码：

```
function addEvent(el, type, fn) {  
    if (window.addEventListener) {  
        el.addEventListener(type, fn, false);  
    }  
    else if (window.attachEvent) {  
        el.attachEvent('on' + type, fn);  
    }  
    else {  
        el['on' + type] = fn;  
    }  
}
```

```
    el['on' + type] = fn;
}
}
```

开发人员在浏览器中使用JavaScript的很大一部分原因就在于事件监听器。因为JavaScript是一种事件驱动的语言，所以JavaScript应用程序中要是连一个事件监听器都没有的话，是一件奇怪的事。不过这种事的确可能存在。在有些高级应用中，JavaScript只被用作一种输出文本和生成DOM节点的编程环境。即便如此，这种语言的力量在很大程度上还是来自于把动作和事件关联起来的能力。这是它很有用的一个方面。

addEvent函数是一个基本的门面，有了它，就有了一种为DOM节点添加事件监听器的简便方法。你不用再为每次为一个元素添加事件监听器时都得针对浏览器间的差异进行检查而烦恼。有了这个便利方法，你大可把与添加事件监听器有关的各种低层细节抛在脑后，而把心思集中在如何构建自己的应用系统上。

在理想状态中，只需要使用addEventListener函数就行。由于并不是所有常见浏览器实现了这个函数，所以我们必须在代码中安排一些分支以便照顾到这个函数不存在的情况。上面的例子中那个addEvent函数会根据浏览器检查的结果选择适用的技术添加事件监听器。有了它，就可以把检查代码封装在一个地方，这可以让实现代码变得更简洁。这是一个应用门面模式对付一组设计得比较糟糕的API的案例，其具体做法就是用一个精心设计的API来包装它们。

## 10.2 JavaScript 库的门面性质

JavaScript库是为人设计的。设计它们的目的在于节省时间、简化常见任务和提供比每个浏览器都实现了的内置JavaScript函数更易于使用的接口。在大量使用DOM脚本编程的浏览器环境中，的确需要有JavaScript库才能对付。如今的Web应用程序开发要求你必须尽量提高编程效率。要做到这一点，最简单的办法就是创建自己的工具函数集或使用Prototype、jQuery或YUI这些第三方JavaScript库。

142

## 10.3 用作便利方法的门面元素

门面模式给予开发人员的另一个好处表现在对函数的组合上。这些组合而得的函数又叫便利函数（convenience function）。下面是一个纯粹形式化的例子：

```
function a(x) {
  // do stuff here...
}

function b(y) {
  // do stuff here...
}

function ab(x, y) {
  a(x);
  b(y);
}
```

你可能会想为什么不一开始就把所有功能都放到函数ab中去。答案是分别提供a、b和ab这几

个函数可以获得更多粒度控制和灵活性。组合a和b可能会对应用程序造成破坏或者产生意想不到的结果。以DOM脚本编程中经常用到的两个普通事件方法为例：

- event.stopPropagation()
- event.preventDefault()

第一个方法stopPropagation的功能是中止事件沿DOM树向上冒泡的传播过程<sup>①</sup>。第二个方法preventDefault的功能是阻止浏览器针对一个事件的默认行为。它 can 以用来防止对链接的点击导致浏览器导航到一个新的页面，也可以用来阻止表单的提交。因为不同的浏览器厂商为这两个功能提供的接口略有差异，所以现在摆在我们面前的就是一个用门面模式实现便利方法的理想案例：

```
var DED = window.DED || {};
DED.util = {
    stopPropagation: function(e) {
        if (ev.stopPropagation) {
            // W3 interface
            e.stopPropagation();
        } else {
            // IE's interface
            e.cancelBubble = true;
        }
    },
    preventDefault: function(e) {
        if (e.preventDefault) {
            // W3 interface
            e.preventDefault();
        } else {
            // IE's interface
            e.returnValue = false;
        }
    },
    /* our convenience method */
    stopEvent: function(e) {
        DED.util.stopPropagation(e);
        DED.util.preventDefault(e);
    }
};
```

尽管看起来很像，但门面模式并不是适配器模式。适配器（第11章将会详细讲述）是一种包装器，用来对接口进行适配以便在不兼容系统中使用它。而创建门面元素则是图个方便。它并不用于达到与需要特定接口的客户系统打交道这个目的，而是用于提供一个简化的接口。

<sup>①</sup> 严格地说，它不仅可以中止冒泡过程，也能中止捕获过程，这取决于事件监听器是用在事件传播的那个阶段。

——译者注

## 10.4示例：设置HTML元素的样式

设置HTML元素的样式是DHTML的核心内容，也是其初衷。要设置HTML元素的样式，只要对样式对象的特定属性赋值即可。通常，在这方面浏览器的差异问题基本上没有多大影响，也无关紧要。例如，如果要将ID为content的div元素的文本颜色设置为红色，可以使用下面的代码：

```
var element = document.getElementById('content');
element.style.color = 'red';
```

如果想把font-size属性值设置为16px，可以这么做：

```
element.style.fontSize = '16px';
```

现在假设要一次设置几个元素的某个样式。这是一个合理的要求。如果有三个ID分别为foo、bar和baz的元素，并且想把它们的文本颜色都设置为红色，可以这样做：

```
var element1 = document.getElementById('foo');
element1.style.color = 'red';
```

```
var element2 = document.getElementById('bar');
element2.style.color = 'red';
```

```
var element3 = document.getElementById('baz');
element3.style.color = 'red';
```

像这样不停地写getElementById并且为每一个元素设置同样的属性和值有点乏味。门面模式可以在此派上用场。为方便起见，我们这就着手创建一个接口，以简化成批设置一组元素的工作。既然你已经了解了各种关键因素，现在我们要采用一种逆向的工作方式，先写出使用待设计的便利方法的代码，然后再编写这个方法本身：

```
setStyle(['foo', 'bar', 'baz'], 'color', 'red');
```

可以看出，要创建的函数名为setStyle，这里传递给它的第一个参数是一个包含着三个ID值的数组。第二个参数是要设置的样式属性，而第三个参数则是该属性的值。了解了接口后，我们现在可以给出一个具体的实现。下面的函数就是一个门面元素，它可以满足我们的需要：

```
function setStyle(elements, prop, val) {
    for (var i = 0, len = elements.length-1; i < len; ++i) {
        document.getElementById(elements[i]).style[prop] = val;
    }
}
```

这个方法已经很有用，但要是我们还可以一次设置多个元素的多个样式就更好了，那样的话就不必反复使用setStyle方法。例如，在设置元素的position时，往往也希望设置它们的top和left样式属性。更有甚者，有些样式属性彼此密切相关，比如边距（margin）和衬距（padding）、字号和行高、前景色和背景色等等。如果使用前面的setStyle方法，那么代码大体会是下面这个样子：

```
setStyle(['foo'], 'position', 'absolute');
setStyle(['foo'], 'top', '50px');
setStyle(['foo'], 'left', '300px');
```

我们可以设计一个更复杂一些的接口，把所有逻辑都组合在另一个门面元素中，以便用一次函数调用就能处理所有这些问题。这个门面元素内部也要使用setStyle，但客户代码对此一无所知。我们把它命名为setCSS：

```
setCSS(['foo'], {
  position: 'absolute',
  top: '50px',
  left: '300px'
});
```

对象字面量的名/值对表示形式看起来甚至更接近于CSS语法。setCSS方法的实现如下：

```
function setCSS(el, styles) {
  for (var prop in styles) {
    if (!styles.hasOwnProperty(prop)) continue;
    setStyle(el, prop, styles[prop]);
  }
}
```

145

现在我们就可以像这样一次设置多个元素的多种样式：

```
setCSS(['foo', 'bar', 'baz'], {
  color: 'white',
  background: 'black',
  fontSize: '16px',
  fontFamily: 'georgia, times, serif'
});
```

## 10.5 示例：设计一个事件工具

前面曾经说过，在处理跨浏览器的开发问题时，最好创建一些门面函数。如果要设计一个大型库，那么最好把其中所有的工具元素拢在一起，这样更好用，访问起来也更简单。鉴于各种浏览器在事件处理方面表现出来的大量差异，开发一个事件工具很有必要。

我们先从一个基本的框架开始。这里要用到单体模式。它位于DED.util命名空间中，包含着我们要设计的各个静态方法：

```
DED.util.Event = {
  // bulk goes here...
};
```

接下来我们将着手解决开发人员在与事件打交道时都会碰到的一些常见问题，比如怎样获得事件目标元素和事件对象。当然，我们也会利用本章前面用来处理事件传播和事件默认行为的代码。下面是一个粗略的框架：

```
DED.util.Event = {
  getEvent: function(e) { },
  getTarget: function(e) { },
  stopPropagation: function(e) { },
  preventDefault: function(e) { },
  stopEvent: function(e) { }
};
```

下面的代码对相关对象的能力和特性进行检查并加入一些代码分支，以图弥合浏览器之间的差异。其结果是创建了5个门面方法，这是一个更为一致的接口，有了它我们的工作会变得更轻松：

```
DED.util.Event = {
    getEvent: function(e) {
        return e || window.event;
    },
    getTarget: function(e) {
        return e.target || e.srcElement;
    },
    stopPropagation: function(e) {
        if (e.stopPropagation) {
            e.stopPropagation();
        } else {
            e.cancelBubble = true;
        }
    },
    preventDefault: function(e) {
        if (e.preventDefault) {
            e.preventDefault();
        } else {
            e.returnValue = false;
        }
    },
    stopEvent: function(e) {
        this.stopPropagation(e);
        this.preventDefault(e);
    }
};
```

现在这个事件工具已经设计好了，可以像这样与前面写好的addEvent函数结合使用：

```
addEvent($('example'), 'click', function(e) {
    // Who clicked me.
    console.log(DED.util.Event.getTarget(e));
    // Stop propagating and prevent the default action.
    DED.util.Event.stopEvent(e);
});
```

## 10.6 实现门面模式的一般步骤

找准自己的应用程序中感觉适合使用门面方法的地方后，就可以着手加入便利方法了。这些函数的名称应经仔细考虑，与它们的用途要相称。对于那种由几个函数组合而成的函数，一个简单的办法就是把相关函数的名称串连成一个函数名，并采用camel大写规范，或者也可以使用thisFunctionAndThatFunction这种形式。

147

处理浏览器API的不一致性属于另一种情况，此时要做的就是把分支代码放在新创建的门面函数中，辅以对象检查或浏览器嗅探等技术。为这种函数取名有点令人挠头，因为你面临的是实现类似功能的函数在不同浏览器中碰巧有着不同名称的情况。这种浏览器称为pageX的，另一种浏览器称为clientX；这种浏览器称为addEventListener的，另一种浏览器称为attachEvent。这里所能给出的最好建议是：取一个好认的名字，并在代码文档中说明该门面函数的用途。

## 10.7 门面模式的适用场合

判断是否应该应用门面模式的关键在于辨认那些反复成组出现的代码。如果函数b出现在函数a之后这种情况经常出现，那么也许你应该考虑添加一个把这两个函数组合起来的门面函数。

在自己的核心工具代码中加入门面函数的另一个可能目的是应对JavaScript内置函数在不同浏览器中的不同表现。这样做并不是因为不能直接使用这些API，而是在处理跨浏览器的差异问题时最好的解决办法就是把这些差异抽取到门面方法中。它们可以提供一个更一致的接口。addEventListener函数就是一例。

## 10.8 门面模式之利

使用门面模式的目的就是要让程序员过得更轻松一些。编写一次组合代码，然后就可以反复使用它，这有助于节省时间和精力。它们可以替你把硬骨头啃掉，并且提供了一个处理常见问题和任务的简化接口。

门面方法方便了开发人员，并且提供了较高层的功能，如果不用门面模式的话这些功能实现起来可能会乏味而又费力。它们还能降低对外部代码的依赖程度，这为应用系统的开发增加了一些额外的灵活性。通过使用门面模式，可以避免与下层子系统紧密耦合。这样就可以对这个系统进行修改而不会影响到客户代码。

## 10.9 门面模式之弊

有时候门面元素也会带来一些不必要的额外负担。方便的东西不一定就得用。门面模式常常会被滥用。在使用你心仪的门面函数之前请三思。搞不好你就是在小题大做。举个例来说，你不会因为电冰箱很方便而且能装很多食品就带它去野营，你也不会去租一台拖拉机来耕自己的花园。在实施一些套路之前应该认真掂量一下其实用性，因为它们不易觉察的破坏性和昂贵代价可能会使应用程序步履蹒跚。有时相比一个庞杂的门面函数，其组成函数在粒度方面更有吸引力。这是因为门面函数可能常常会执行一些你并不需要的任务。

对于简单的个人网站或少量营销网页来说，仅为工具提示和弹出式窗口这样一点增强行为就导入整个JavaScript库可能并不明智。此时也许可以考虑只使用少许简单的门面元素而不是一个满是这类东西的库。说到底，这得由你自己决定，看这种模式是否可行。

## 10.10 小结

门面模式可用来创建便利函数，这些函数为执行各种复杂任务提供了一个简单的接口。它们使代码更容易维护和理解<sup>①</sup>。它们还能弱化子系统和客户代码的耦合。便利方法有助于简化常见的重复性任务，以及把经常相伴出现的常用函数组合在一起。这个模式在DOM脚本编程这种需要面对各种不一致的浏览器接口的环境中很常用。

148

---

<sup>①</sup> 原文为 “They help keep your code maintainable, understandable, and abstraction-oriented.” 其中的 “abstraction-oriented” 一词非常令人费解，所以译文中没有译出。——译者注

# 适配器模式

**适**配器模式可用来在现有接口和不兼容的类之间进行适配。使用这种模式的对象又叫包装器（wrapper），因为它们是在用一个新的接口包装另一个对象。许多时候创建适配器对程序员和接口的设计人员都有好处。在设计类的时候往往回遇到有些接口不能与现有API一同使用的情况。借助于适配器，你不用直接修改这些类也能使用它们。本章将考察一些这类场合，并探讨用适配器模式连接对象的各种方式。

## 11.1 适配器的特点

适配器可以被添加到现有代码中以协调两个不同的接口。如果现有代码的接口能很好地满足需要，那就可能没有必要使用适配器。但要是现有接口对于手头的工作来说不够直观或实用，那么可以使用适配器来提供一个更简洁或更丰富（option-rich）的接口。

从表面上看，适配器模式很像门面模式。它们都要对别的对象进行包装并改变其呈现的接口。二者的差别在于它们如何改变接口。门面元素展现的是一个简化的接口，它并不提供额外的选择，而且有时为了方便完成常见任务它还会做出一些假定。而适配器则要把一个接口转换为另一个接口，它并不会滤除某些能力，也不会简化接口。如果客户系统期待的API不可用，那就需要用到适配器。

适配器可被实现为不兼容的方法调用之间的一个代码薄层。如果你有一个具有3个字符串参数的函数，但客户系统拥有的却是一个包含三个字符串元素的数组，此时就可以用一个适配器来衔接二者。

假设你有一个对象，还有一个以三个字符串为参数的函数：

```
var clientObject = {  
    string1: 'foo',  
    string2: 'bar',  
    string3: 'baz'  
};  
function interfaceMethod(str1, str2, str3) {  
    ...  
}
```

为了把clientObject作为参数传递给interfaceMethod，需要用到适配器。我们可以这样创建一个：

```
function clientToInterfaceAdapter(o) {
    interfaceMethod(o.string1, o.string2, o.string3);
}
```

现在就可以把整个对象传递给这个函数：

```
clientToInterfaceAdapter(clientObject);
```

clientToInterfaceAdaper函数的作用就在于对interfaceMethod函数进行包装，并把传递给它的参数转换为后者需要的形式。

## 11.2 适配原有实现

在某些情况下，从客户一方对代码进行修改是不可能的。有些程序员因此索性避免创建API。如果现有接口发生了改变，那么客户代码也必须进行相应的修改后才能使用这个新接口，否则整个应用系统就有失灵的危险。在引入新接口之后，一般说来最好向客户方提供一些可为其实现新接口的适配器。

以PC硬件为例，PS2插口是连接鼠标和键盘的标准接口。多年以来几乎所有PC都带有这种接口，鼠标和键盘的设计人员（用本章的术语来讲就是客户）因此就有了一个固定的设计目标。后来硬件工程师们发明了可以完全替代PS2接口的技术，他们改用USB系统来支持键盘、鼠标和其他外设。

但现在问题来了。对于设计主板的工程师来说，消费者有没有USB键盘都无所谓。他们决定不再支持PS2插口，以便降低成本（并且节省空间）。键盘和鼠标的设计人员这下意识到，要想卖掉他们以前针对PS2接口生产的成千上万套键盘和鼠标，需要有适配器的支持才行。大家熟悉的PS2-to-USB适配器于是就应运而生了。

## 11.3 示例：适配两个库

现在可供选择的JavaScript库非常多。库的用户应该认真分析一下，看看哪套工具集最适合自己的需要，以及它们对自己的开发会带来什么影响。此外，需要考虑的因素还包括其他开发人员的编程风格、实现的难易程度以及是否存在与现有代码的冲突和不兼容问题。

即使原来已经选好了库，开发团队以后仍可能会出于性能、安全或设计的考虑，在不改动已有代码的前提下更换所用的库。有时公司甚至可能会为了帮助开发新手而提供一套中间性的适配器，比如说，为了帮助他们从自己熟悉的另一套API过渡到现在使用的API。

在最简单的情况下，创建适配器库往往是一个比改写所有代码更好的选择。下面的例子要实现的是从Prototype库的\$函数到YUI（Yahoo! User Interface）的get方法的转换。这两个函数的功能比较相似，不过请先看看它们在接口方面的差别：

```
// Prototype $ function.
function $() {
    var elements = new Array();
    for(var i = 0; i < arguments.length; i++) {
        var element = arguments[i];
```

```

if(typeof element == 'string')
    element = document.getElementById(element);
if(arguments.length == 1)
    return element;
elements.push(element);
}
return elements;
}

/* YUI get method. */
YAHOO.util.Dom.get = function(el) {
    if(YAHOO.lang.isString(el)) {
        return document.getElementById(el);
    }
    if(YAHOO.lang.isArray(el)) {
        var c = [];
        for(var i = 0, len = el.length; i < len; ++i) {
            c[c.length] = YAHOO.util.Dom.get(el[i]);
        }
        return c;
    }
    if(el) {
        return el;
    }
    return null;
};

```

二者的区别主要在于：get具有一个参数，这个参数可以是一个HTML元素、字符串或者由字符串或HTML元素组成的数组，与此不同，\$函数没有正式列出参数，而是允许客户传入任意数目的参数，不管是字符串还是HTML元素都行。

如果你需要从使用Prototype的\$函数改为使用YUI的get方法（或者相反），那么用于这个用途的适配器会是什么样子呢？其实现简单得令人吃惊：

```

function PrototypeToYUIAdapter() {
    return YAHOO.util.Dom.get(arguments);
}
function YUIToPrototypeAdapter(el) {
    return $.apply(window, el instanceof Array ? el:[el]);
}

```

151

注意观察适配器是如何包装被适配方法的。有了这些适配器，现有的客户系统就可以继续使用其熟悉的API。如果Prototype库的用户想利用YUI的方法，那么只要通过把\$函数插接到适配器函数来适配自己现有的全部代码即可。用不着为此修改任何原有的方法。对于从Prototype改投YUI的人来说，只要添加下面这行代码即可：

```
$ = PrototypeToYUIAdapter;
```

而那些从YUI改投Prototype的人则应该使用下面的代码：

```
YAHOO.util.Dom.get = YUIToPrototypeAdapter;
```

## 11.4 示例：适配电子邮件 API

本例研究的是一个Web邮件API，它可以用来接收、发送邮件并执行一些别的任务。我们将采用类Ajax技术从服务器获取消息，然后将消息详情载入DOM。在完成这个应用程序接口之后，我们将讨论如何为其编写包装函数，以便那些期待一个不同接口的客户也能使用这个API。

要事先办，先看一下整个应用系统：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Mail API Demonstration</title>
    <style type="text/css" media="screen">
      body {
        font: 62.5% georgia,times,serif;
      }
      #doc {
        margin: 0 auto;
        width: 500px;
        font-size: 1.3em;
      }
    </style>
    <script src="lib-utils.js"></script>
    <script type="text/javascript">
      // application utilities
      var DED = {};
      DED.util = {
        substitute: function (s, o) {
          return s.replace(/{{[^{}]*}}/g,
            function (a, b) {
              var r = o[b];
              return typeof r === 'string' || typeof r === 'number' ? r : a;
            }
          );
        },
        asyncRequest: (function() {
          function handleReadyState(o, callback) {
            var poll = window.setInterval(
              function() {
                if(o && o.readyState == 4) {
                  window.clearInterval(poll);
                  if (callback) {
                    callback(o);
                  }
                }
              }
            );
          }
        })
      };
    </script>
  </head>
  <body>
    <div id="doc">
      <h1>Mail API Demonstration</h1>
      <p>This page demonstrates how to use the Mail API to receive and send e-mail messages. It uses the lib-utils.js file for utility functions, and the Mail API itself for message handling. The Mail API is implemented using XMLHttpRequest and DOM manipulation, making it suitable for both server-side and client-side applications. The application shows how to receive a message, display its contents, and then send a reply message back to the server. The code is modular and can be easily adapted for different environments or clients. The lib-utils.js file contains various utility functions, such as a substitute function for replacing placeholders in strings, and an asynchronous request function for handling XMLHttpRequests. The Mail API itself provides methods for receiving messages, sending messages, and managing message threads. The application demonstrates how to use these methods to build a simple e-mail client. The code is well-commented and follows best practices for web development. It's a good example of how to use modern web technologies to build complex applications.
    </div>
  </body>
</html>
```

```

        }
    },
    50
);
}

var getXHR = function() {
    var http;
    try {
        http = new XMLHttpRequest;
        getXHR = function() {
            return new XMLHttpRequest;
        };
    }
    catch(e) {
        var msxml = [
            'MSXML2.XMLHTTP.3.0',
            'MSXML2.XMLHTTP',
            'Microsoft.XMLHTTP'
        ];
        for (var i=0, len = msxml.length; i < len; ++i) {
            try {
                http = new ActiveXObject(msxml[i]);
                getXHR = function() {
                    return new ActiveXObject(msxml[i]);
                };
                break;
            }
            catch(e) {}
        }
    }
    return http;
};

return function(method, uri, callback, postData) {
    var http = getXHR();
    http.open(method, uri, true);
    handleReadyState(http, callback);
    http.send(postData || null);
    return http;
};
})()
}

// dedMail application interface.
var dedMail = (function() {
    function request(id, type, callback) {
        DED.util.asyncRequest(
            'GET',
            'mail-api.php?ajax=true&id=' + id + '&type=' + type,
            function(o) {
                callback(o.responseText);
            }
        );
    }
    return {
        request: request
    };
})();

```

```

        }
    );
}

return {
    getMail: function(id, callback) {
        request(id, 'all', callback);
    },
    sendMail: function(body, recipient) {
        // Send mail with body text to the supplied recipient.
    },
    save: function(id) {
        // Save a draft copy with the supplied email ID.
    },
    move: function(id, destination) {
        // Move the email to the supplied destination folder.
    },
    archive: function(id) {
        // Archive the email. This can be a basic facade method that uses
        // the move method, hard-coding the destination.
    },
    trash: function(id) {
        // This can also be a facade method which moves the message to
        // the trash folder.
    },
    reportSpam: function(id) {
        // Move message to spam folder and add sender to the blacklist.
    },
    formatMessage: function(e) {
        var e = e || window.event;
        try {
            e.preventDefault();
        }
        catch(ex) {
            e.returnValue = false;
        }
        var targetEl = e.target || e.srcElement;
        var id = targetEl.id.toString().split('-')[1];
        dedMail.getMail(id, function(msgObject) {
            var resp = eval('(' + msgObject + ')');
            var details = '<p><strong>From:</strong> ' + resp.from + '<br>';
            details += '<strong>Sent:</strong> ' + resp.date + '</p>';
            details += '<p><strong>Message:</strong><br>';
            details += resp.message + '</p>';
            $('#message-pane').innerHTML = DED.util.substitute(details, resp);
        })
    }
};

// Set up mail implementation.
addEvent(window, 'load', function() {

```

```

var threads = getElementsByClass('thread', 'a');
for (var i=0, len=threads.length; i<len; ++i) {
    addEvent(threads[i], 'click', dedMail.formatMessage);
}
});
</script>
</head>

<body>
<div id="doc">
<h1>Email Application Interface</h1>
<ul>
<li>
<a class="thread" href="#" id="msg-1">
    load message Sister Sonya
</a>
</li>
<li>
<a class="thread" href="#" id="msg-2">
    load message Lindsey Simon
</a>
</li>
<li>
<a class="thread" href="#" id="msg-3">
    load message Margaret Stooart
</a>
</li>
</ul>
<div id="message-pane"></div>
</div>
</body>
</html>

```

155

在深入代码的细节之前先看看图11-1，它是点击了其中一个消息条目后的最终输出结果的屏幕截图。这可以给你一个有关你即将分析研究的系统的初步印象。

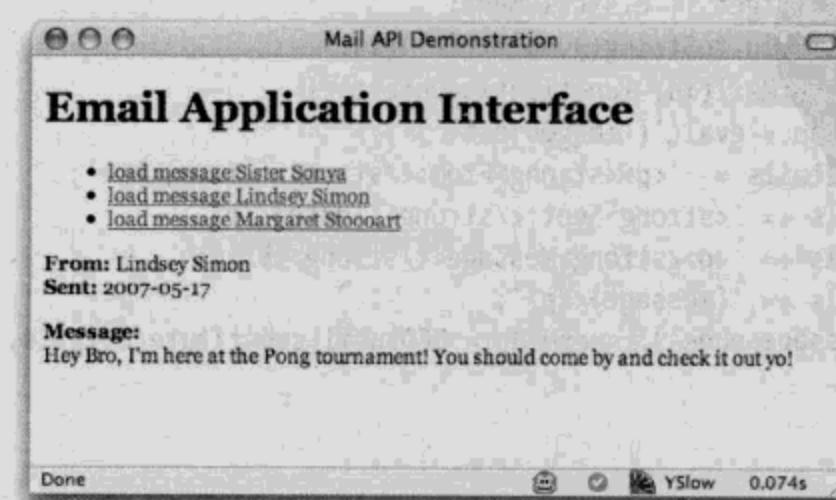


图 11-1

你首先注意到的可能是这个网页中导入了一个基本工具集（lib-utils.js），函数getElementsByClass、\$和addEvent都是在这个工具集中定义的。接下来，我们在DED.util命名空间中添加了一些应用程序工具，这个应用程序的开发需要用到它们。DED.util.substitute方法的用途是用第二个参数传入的对象的属性值替换第一个参数传入的字符串中的相关部分。下面是一个例子：

```
var substitutionObject = {
  name: "world",
  place: "Google"
};

var text = 'Hello {name}, welcome to {place}';
var replacedText = DED.util.substitute(text, substitutionObject);
console.log(replacedText);
// produces "Hello world, welcome to Google"
```

下一个工具函数asyncRequest用于向服务端发出Ajax请求。另外请注意，针对浏览器的差异，函数中创建XMLHttpRequest对象的部分使用了分支技术。在第一次调用getXHR函数以获取XHR对象之后，这个函数被重新定义。这样一来，对浏览器差异的检查只会执行一次，而不是每次调用时都要进行。由于减少了对象检查的次数，因此这种技术能够大幅提高应用程序的运行速度。

最后来看dedMail这个单体对象：

```
var dedMail = (function() { ... })();
```

这个对象提供了getMail、sendMail、move和archive等常见的邮件处理方法。注意这个例子中只实现了getMail方法的逻辑，这个方法根据所提供的ID从服务器获取邮件。消息加载结束后，作为参数传给getMail方法的回调函数callback将被调用，传给这个回调函数的参数是服务器回应的文本内容。其实你也可以用发布/订阅模式来监听ready事件，但这里采用的是进行XHR调用时很常见的函数式风格。这只是接口开发人员的个人偏好问题。

156

### 11.4.1 用适配器包装 Web 邮件 API

现在这个应用程序接口已经创建完毕，可以在客户代码中对它们进行调用。事情看起来很顺利：你使用了其中的方法，谨慎地测试了回调函数，并且对数据对象进行解析，然后将其载入DOM。不过请等一下。实验工程（experimental engineering）小组那边的人已经用原来的fooMail系统写好了他们的代码，不过，他们也愿意利用dedMail这个更先进的新接口。问题在于，他们的方法<sup>①</sup>要求提供的是HTML片段。其构造函数<sup>②</sup>也只接受一个ID值。而且他们的getMail函数只有回调函数这一个参数。这多少有点老土（dedMail小组的工程师们也这样想），不过那些使用fooMail的工程师无疑可以从dedMail的性能测试中受益。最后，他们希望不用改写全部代码。所以，我们的决定是：要有适配器。

<sup>①</sup> 从上下文看，是指fooMail中用作getMail方法参数的那个回调函数。——译者注

<sup>②</sup> 作者语焉不详。估计应该是指fooMail构造函数。——译者注

### 11.4.2 从fooMail转向dedMail

与Prototype和YUI的适配器那个例子一样，从fooMail转向dedMail相对而言应该很简单。在充分了解了提供方（supplier）和接受方（receiver）的情况后，你就可以截取来自提供方的逻辑，然后用接受方能够理解的方式对其进行转换。

先来看一段使用fooMail这个API的代码。

```
fooMail.getMail(function(text) {
    $('message-pane').innerHTML = text;
});
```

注意，getMail方法以一个回调方法为参数，这个回调函数在被调用时得到的参数是包含着发信人姓名、发信日期和信件内容的一段文本。这不算理想，但使用fooMail的工程师们不想冒破坏现有应用系统的危险对此进行修改。你可以像下面这样为他们写一个简单的适配器，这样他们就不必改变自己的原有代码<sup>①</sup>：

```
var dedMailtoFooMailAdapter = {};
dedMailtoFooMailAdapter.getMail = function(id, callback) {
    dedMail.getMail(id, function(resp) {
        var resp = eval('('+'+resp+')');
        var details = '<p><strong>From:</strong> {from}<br>';
        details += '<strong>Sent:</strong> {date}</p>';
        details += '<p><strong>Message:</strong><br>';
        details += '{message}</p>';
        callback(DED.util.substitute(details, resp));
    });
}
// Other methods needed to adapt dedMail to the fooMail interface.
...
// Assign the adapter to the fooMail variable.
fooMail = dedMailtoFooMailAdapter;
```

这段代码中用dedMailtoFooMailAdapter这个单体对象改写了fooMail对象。该单体对象实现了一个getMail方法。这个方法内部在调用那个回调函数时会把一段HTML文本作为参数正确地传给它。

## 11.5 适配器模式的适用场合

适配器适用于客户系统期待的接口与现有API提供的接口不兼容这种场合。它只能用来协调语法上的差异问题。适配器所适配的两个方法执行的应该是类似的任务，否则的话它就解决不了

<sup>①</sup> 作者的这段代码有严重的错误。这个适配器既然是用来让客户以fooMail系统的接口来使用dedMail系统中的功能的，那么其getMail方法的参数列表就应该与fooMail.getMail的参数列表一致，即只以一个回调函数为参数。然而dedMailtoFooMailAdapter.getMail的参数列表却与dedMail.getMail一样，这是不行的。另外，这个适配器应命名为fooMailtoDedMailAdapter才对。实际上，要想改正这个错误，必须对fooMail这个对象有进一步了解才行。然而作者披露的相关信息太少，所以我也无法为读者提供一个正确的适配器版本。——译者注

问题。如果客户想要的是一个不同的接口，比如说一个他们用起来更容易一些的接口，那么也可以为此而使用适配器。就像桥接元素和门面元素一样，通过创建适配器，可以把抽象与其实现隔离开来，以便二者独立变化。

## 11.6 适配器模式之利

本章通篇都在讲：适配器有助于避免大规模改写现有客户代码。其工作机制是：用一个新的接口对现有类的接口进行包装，这样客户程序就能使用这个并非为其量身打造的类而又毋需为此大动手术。

## 11.7 适配器模式之弊

可能有些工程师不想使用适配器，其原因主要在于他们实际上需要彻底重写代码。有人认为适配器是一种不必要的开销，完全可以通过重写现有代码避免。此外适配器模式也会引入一批需要支持的新工具。如果现有API还未定形，或者新接口还未定形（这更有可能），那么适配器可能不会一直管用。在设计键盘的硬件工程师创造PS2-to-USB适配器这个案例中，成千上万的键盘上面的PS2插头不会再有什么变化，而USB接口则成了新的标准，所以这种适配器才有意义。但是在软件开发这一行中，情况并非总是如此。

## 11.8 小结

适配器模式是一种很有用的技术，它可以用来对类和对象进行包装，以便向客户代码提供其期待的接口。应用这种技术，你可以在不影响现有实现的前提下利用新的更好的接口。作为一个实现者（implementer），你可以根据自己的需要定制接口。这种模式的确会引入一些新代码，不过，在涉及大型系统和遗留框架的情况下，它的优点往往比缺点更突出。

# 装饰者模式

**本**章讨论的是一种为对象增添特性的技术，它并不使用创建新子类这种手段。装饰者模式（decorator pattern）可用来透明地把对象包装在具有同样接口的另一对象之中。这样一来，你可以给一个方法添加一些行为，然后将方法调用传递给原始对象。相对于创建子类来说，使用装饰者对象是一种更灵活的选择。这种模式特别适合JavaScript（在本章后面讲到动态接口时你就会体会到这一点），因为通常JavaScript代码并不怎么依赖对象的类型。

## 12.1 装饰者的结构

装饰者可用于为对象增加功能。它可以用来替代大量子类。为了准确说明这个概念，我们将进一步分析第7章所讲的那个自行车商店的例子。你上次见到AcmeBicycleShop类的时候，顾客可以购买的自行车有4种型号。后来这家商店开始为每一种自行车提供一些额外的特色配件。现在顾客再加点钱就可以买到带前灯、尾灯、前挂货篮或铃铛的自行车。每一种可选配件都会影响到售价和车的组装方法。这个问题最基本的解决办法是为选件的每一种组合创建一个子类：

```
var AcmeComfortCruiser = function() { ... }; // The superclass for all of the
  // other comfort cruisers
var AcmeComfortCruiserWithHeadlight = function() { ... };
var AcmeComfortCruiserWithTaillight = function() { ... };
var AcmeComfortCruiserWithHeadlightAndTaillight = function() { ... };
var AcmeComfortCruiserWithBasket = function() { ... };
var AcmeComfortCruiserWithHeadlightAndBasket = function() { ... };
var AcmeComfortCruiserWithTaillightAndBasket = function() { ... };
var AcmeComfortCruiserWithHeadlightTaillightAndBasket = function() { ... };
var AcmeComfortCruiserWithBell = function() { ... };
...
...
```

但是这种办法根本行不通，原因很简单：这需要实现100多个类（那4个父类每个都要派生24个子类，再加上父类本身）。而且这样做的话你不得不对工厂方法进行修改，以便能创建分别属于这100个子类的自行车来卖给顾客。你恐怕不想让自己的余生都花在维护成100多个子类上，所以，得想点更好的办法才是。

装饰者模式对于实现这些选择再合适不过了。你不用为自行车和选件的每一种组合创建一个子类，而只需创建4个新类（一个类针对一种选件）即可。这些类与那4种自行车类一样都要实现Bicycle接口，但它们只被用作这些自行车类的包装类。在这些选件类上进行的方法调用将被转

到它们包装的自行车类上，有时会稍有修改。

在这个例子中，选件类就是装饰者，而自行车类是它们的组件（component）。装饰者对其组件进行了透明包装，二者可以互换使用，这是因为它们实现了同样的接口。下面我们来看应该怎样实现自行车装饰者类。首先要改一下接口，加入一个getPrice方法：

```
/* The Bicycle interface. */

var Bicycle = new Interface('Bicycle', ['assemble', 'wash', 'ride', 'repair',
    'getPrice']);
```

所有自行车类和选件装饰者都要实现这个接口。AcmeComfortCruiser类大致是这个样子（不需要为使用装饰者而进行什么修改）：

```
/* The AcmeComfortCruiser class. */

var AcmeComfortCruiser = function() { // implements Bicycle
    ...
};

AcmeComfortCruiser.prototype = {
    assemble: function() {
        ...
    },
    wash: function() {
        ...
    },
    ride: function() {
        ...
    },
    repair: function() {
        ...
    },
    getPrice: function() {
        return 399.00;
    }
};
```

我们不关心除getPrice方法外的其他方法的实现细节。本节稍后定义那4个选件类时你就会明白其原因。这些选件类的作用基本上就是传递发生在它们身上的方法调用。为了简化这个任务，也为了方便以后增添更多选件，我们将创建一个抽象类BicycleDecorator，所有选件类都从此派生。它提供了Bicycle接口所要求的各个方法的默认版本：

```
/* The BicycleDecorator abstract decorator class. */

var BicycleDecorator = function(bicycle) { // implements Bicycle
    Interface.ensureImplements(bicycle, Bicycle);
    this.bicycle = bicycle;
}

BicycleDecorator.prototype = {
    assemble: function() {
        return this.bicycle.assemble();
    },
    wash: function() {
        return this.bicycle.wash();
    },
    ride: function() {
        return this.bicycle.ride();
    },
    repair: function() {
        return this.bicycle.repair();
    },
    getPrice: function() {
        return this.bicycle.getPrice();
    }
};
```

```

    },
    wash: function() {
        return this.bicycle.wash();
    },
    ride: function() {
        return this.bicycle.ride();
    },
    repair: function() {
        return this.bicycle.repair();
    },
    getPrice: function() {
        return this.bicycle.getPrice();
    }
};

```

几乎没有比这更简单的装饰者类了。它的构造函数接受一个对象参数，并将其用作该装饰者的组件。该类实现了Bicycle接口，它所实现的每一个方法所做的只是在其组件上调用同名方法。乍一看这与组合模式的工作方式非常相似，不过我们会在12.1.2节中解释二者的差别。BicycleDecorator类是所有选件类的超类。对于那些不需要修改的方法，选件类只要使用从BicycleDecorator继承而来的版本即可，而这些方法又会在组件上调用同样的方法，因此选件类对于任何客户代码都是透明的。

到了这里，装饰者开始变得有趣了。有了BicycleDecorator，创建各种选件类很容易，只需要调用超类的构造函数并改写某些方法即可。下面是HeadlightDecorator类的代码：

```

/* HeadlightDecorator class. */

var HeadlightDecorator = function(bicycle) { // implements Bicycle
    // Call the superclass's constructor.
    HeadlightDecorator.superclass.constructor.call(this, bicycle);
}
extend(HeadlightDecorator, BicycleDecorator); // Extend the superclass.
HeadlightDecorator.prototype.assemble = function() {
    return this.bicycle.assemble() + ' Attach headlight to handlebars.';
};
HeadlightDecorator.prototype.getPrice = function() {
    return this.bicycle.getPrice() + 15.00;
};

```

这个类很简单。它重定义了需要进行装饰的两个方法。本例中装饰这些方法的做法是，先执行组件的方法，然后在此基础上附加一些装饰元素。assemble方法中附加的是一条指示，而getPrice方法则是把前灯的价格计入总价当中。

现在一切就绪，该来看看怎么使用装饰者了。要创建一辆带前灯的自行车，首先应该创建自行车的实例，然后以该自行车对象为参数实例化前灯选件。在此之后，应该只使用这个HeadlightDecorator对象，你完全可以将其视为一辆自行车，而把它是一个装饰者对象这件事抛在脑后：

```

var myBicycle = new AcmeComfortCruiser(); // Instantiate the bicycle.
alert(myBicycle.getPrice()); // Returns 399.00

```

```
myBicycle = new HeadlightDecorator(myBicycle); // Decorate the bicycle object.
alert(myBicycle.getPrice()); // Now returns 414.00
```

上面的代码中第3行最为关键。这里用来存放那个HeadlightDecorator实例的不是另一个变量，而是用来存放自行车实例的同一变量。这意味着此后将不能再访问原来的那个自行车对象，不过没关系，你以后不再需要这个对象。那个装饰者完全可以和自行车对象互换使用。这也意味着你可以随心所欲地嵌套使用多种装饰者。假如你创建一个TaillightDecorator类，那么可以将其与HeadlightDecorator结合使用：

```
/* TaillightDecorator class. */

var TaillightDecorator = function(bicycle) { // implements Bicycle
    // Call the superclass's constructor.
    TaillightDecorator.superclass.constructor.call(this, bicycle);
}
extend(TaillightDecorator, BicycleDecorator); // Extend the superclass.
TaillightDecorator.prototype.assemble = function() {
    return this.bicycle.assemble() + ' Attach taillight to the seat post.';
};
TaillightDecorator.prototype.getPrice = function() {
    return this.bicycle.getPrice() + 9.00;
};

var myBicycle = new AcmeComfortCruiser(); // Instantiate the bicycle.
alert(myBicycle.getPrice()); // Returns 399.00

myBicycle = new TaillightDecorator(myBicycle); // Decorate the bicycle object
  // with a taillight.
alert(myBicycle.getPrice()); // Now returns 408.00

myBicycle = new HeadlightDecorator(myBicycle); // Decorate the bicycle object
  // again, now with a headlight.
alert(myBicycle.getPrice()); // Now returns 423.00
```

你可以如法炮制，创建对应于前挂货篮和铃铛的装饰者。通过在运行期间动态应用装饰者，可以创建出具有所有需要的特性的对象，而不用去维护那100个不同的子类。要是前灯的价格发生变化，你只要在HeadlightDecorator类这一个地方予以更新即可。维护工作因此也更容易管理得多。

162

### 12.1.1 接口在装饰者模式中的角色

装饰者模式颇多得益于接口的使用。装饰者最重要的特点之一就是它可以用来替代其组件。在本例中，这就是说任何原来使用AcmeComfortCruiser实例的地方，都可以使用HeadlightDecorator实例，为此不必对代码进行任何修改。这是通过确保所有装饰者对象都实现了Bicycle接口而达到的。

接口在此发挥着两个方面的作用。首先，它说明了装饰者必须实现哪些方法，这有助于防止开发过程中的错误。通过创建一个具有一批固定方法的接口，你所面对的就不再是一个游移不定的目标。此外，它还可以在新版工厂方法（参见12.3节）中用来确保所创建的对象都实现了必需的方法。

如果装饰者对象与其组件不能互换使用，它就丧失了其功用。这是装饰者模式的关键特点，要注意防止装饰者和组件出现接口方面的差异。这种模式的好处之一就是可以透明地用新对象装饰现有系统中的对象，而这并不会改变代码中的其他东西。只有装饰者和组件实现了同样的接口才能做到这一点。

### 12.1.2 装饰者模式与组合模式的比较

从BicycleDecorator类这个例子可以看到，装饰者模式和组合模式之间有许多共同点。装饰者对象和组合对象都是用来包装别的对象（那些对象在组合模式中称为子对象，而在装饰者模式中称为组件），它们都与所包装的对象实现同样的接口并且会把任何方法调用传递给这些对象。像BicycleDecorator这种极其基本的装饰者甚至可以被视为一个简单的组合对象。那么这二者之间又有什么区别呢？

组合模式是一种结构型模式，用于把众多子对象组织为一个整体。藉此程序员与大批对象打交道时可以将它们当作一个对象来对待，并将它们组织为层次性的树。通常它并不修改方法调用，而只是将其沿组合对象与子对象的链向下传递，直到到达并落实在叶对象上。

装饰者模式也是一种结构型模式，但它并非用于组织对象，而是用于在不修改现有对象或从其派生子类的前提下为其增添职责。在一些较简单的例子中，装饰者会透明而不加修改地传递所有方法调用，不过，创建装饰者的目的就在于对方法进行修改。HeadlightDecorator就修改了assemble和getPrice方法，其做法是先传递方法调用，然后修改其返回结果。

尽管简单的组合对象可等同于简单的装饰者，这二者却有着不同的焦点。组合对象并不修改方法调用，其着眼点在于组织子对象。而装饰者存在的唯一目的就是修改方法调用而不是组织子对象，因为子对象只有一个。虽然这两种模式的结构惊人地相似，但它们的任务完全不同，并无混淆之虞。

## 12.2 装饰者修改其组件的方式

装饰者的作用就在于以某种方式对其组件对象的行为进行修改。本节将介绍一些这方面的做法。

### 12.2.1 在方法之后添加行为

在方法之后添加行为是最常见的修改方法的做法。具体而言就是先调用组件的方法，并在其返回后实施一些附加的行为。HeadlightDecorator的getPrice方法就是一个简单的例子：

```
HeadlightDecorator.prototype.getPrice = function() {
    return this.bicycle.getPrice() + 15.00;
};
```

在这个例子中，首先对组件调用getPrice方法，然后再把前灯的价钱加在该方法调用所返回的价钱上，最后将其结果作为总价返回。这一过程可以根据需要多次重复。作为示范，下面将创建一辆带有两个前灯和一个尾灯的自行车：

```
var myBicycle = new AcmeComfortCruiser(); // Instantiate the bicycle.
alert(myBicycle.getPrice()); // Returns 399.00

myBicycle = new HeadlightDecorator(myBicycle); // Decorate the bicycle object
// with the first headlight.
myBicycle = new HeadlightDecorator(myBicycle); // Decorate the bicycle object
// with the second headlight.
myBicycle = new TaillightDecorator(myBicycle); // Decorate the bicycle object
// with a taillight.

alert(myBicycle.getPrice()); // Now returns 438.00
```

该调用栈大体是这样的：在TaillightDecorator对象（这是最外层的装饰者）上调用getPrice方法，这将转而在较外层的HeadlightDecorator对象上调用getPrice方法，这样继续下去，直到到达AcmeComfortCruiser对象并返回其价格。每个装饰者都会加上自己的价格，然后将和返回给相邻的外层。最后结果是438.00。

assemble方法是另一个在方法之后添加行为的例子。这个方法并不是把多个数值累加为最终价格，而是在之前的组装指示后面添加新的指示。其最终结果是对组装整部自行车所要经历的一系列步骤的描述，包括最后安装那些由装饰者所代表的部件的步骤。

这是修改组件方法最常见的做法。它在保留原有行为的基础上添加一些额外行为或修改返回结果。

### 12.2.2 在方法之前添加行为

如果行为修改发生在执行组件方法之前，那么要么必须把装饰者行为安排在调用组件方法之前，要么必须设法修改传递给组件方法的参数值。下面的例子实现了一个提供车架颜色选择的装饰者：

```
/* FrameColorDecorator class. */

var FrameColorDecorator = function(bicycle, frameColor) { // implements Bicycle
    // Call the superclass's constructor.
    FrameColorDecorator.superclass.constructor.call(this, bicycle);
    this.frameColor = frameColor;
}
extend(FrameColorDecorator, BicycleDecorator); // Extend the superclass.
FrameColorDecorator.prototype.assemble = function() {
    return 'Paint the frame ' + this.frameColor + ' and allow it to dry. ' +
        this.bicycle.assemble();
};
FrameColorDecorator.prototype.getPrice = function() {
    return this.bicycle.getPrice() + 30.00;
};
```

这里有两点与先前看到的装饰者不同。首先是该装饰者多了frameColor这个新状态。这个状态是通过构造函数新增的一个参数设置的。第二点差别在于本例中assemble方法添加的步骤出现在其他组装指示之前而不是之后。不管是在组件方法之前还是之后添加行为，它们都是装饰者的有效实现方式。装饰者并非只能在组件方法调用之后进行修改或执行代码。相反，它可以在调用组件方法之前执行代码，或修改要传递给该方法的参数。装饰者也可以增添一些用以实现其提供的附加特性的属性（如本例的frameColor）。

在用FrameColorDecorator装饰过的对象上调用assemble方法时，新的指示会被添加到开头部分：

```

var myBicycle = new AcmeComfortCruiser(); // Instantiate the bicycle.
myBicycle = new FrameColorDecorator(myBicycle, 'red'); // Decorate the bicycle
// object with the frame color.
myBicycle = new HeadlightDecorator(myBicycle); // Decorate the bicycle object
// with the first headlight.
myBicycle = new HeadlightDecorator(myBicycle); // Decorate the bicycle object
// with the second headlight.
myBicycle = new TaillightDecorator(myBicycle); // Decorate the bicycle object
// with a taillight.

alert(myBicycle.assemble());
/* Returns:
   "Paint the frame red and allow it to dry. (Full instructions for assembling
   the bike itself go here) Attach headlight to handlebars. Attach headlight
   to handlebars. Attach taillight to the seat post."
*/

```

165

### 12.2.3 替换方法

有时为了实现新行为必须对方法进行整体替换。在此情况下，组件方法不会被调用（或虽然被调用但其返回值会被抛弃）。作为这种修改的一个例子，下面我们将创建一个用来实现自行车的终生保修的装饰者：

```

/* LifetimeWarrantyDecorator class. */

var LifetimeWarrantyDecorator = function(bicycle) { // implements Bicycle
  // Call the superclass's constructor.
  LifetimeWarrantyDecorator.superclass.constructor.call(this, bicycle);
}

extend(LifetimeWarrantyDecorator, BicycleDecorator); // Extend the superclass.
LifetimeWarrantyDecorator.prototype.repair = function() {
  return 'This bicycle is covered by a lifetime warranty. Please take it to '
    + 'an authorized Acme Repair Center.';
};

LifetimeWarrantyDecorator.prototype.getPrice = function() {
  return this.bicycle.getPrice() + 199.00;
};

```

这个装饰者把组件的repair方法替换为一个新方法，而组件的方法则再也不会被调用。装饰者也可以根据某种条件决定是否替换组件方法，在条件满足时替换方法，否则就使用组件的方法。

下面这个例子创建的装饰者用于实现规定了保修期的保修:

```
/* TimedWarrantyDecorator class. */

var TimedWarrantyDecorator = function(bicycle, coverageLengthInYears) {
    // implements Bicycle
    // Call the superclass's constructor.
    TimedWarrantyDecorator.superclass.constructor.call(this, bicycle);
    this.coverageLength = coverageLengthInYears;
    this.expDate = new Date();
    var coverageLengthInMs = this.coverageLength * 365 * 24 * 60 * 60 * 1000;
    this.expDate.setTime(this.expDate.getTime() + coverageLengthInMs);
}

extend(TimedWarrantyDecorator, BicycleDecorator); // Extend the superclass.

TimedWarrantyDecorator.prototype.repair = function() {
    var repairInstructions;
    var currentDate = new Date();
    if(currentDate < this.expDate) {
        repairInstructions = 'This bicycle is currently covered by a warranty.' +
            'Please take it to an authorized Acme Repair Center.';
    }
    else {
        repairInstructions = this.bicycle.repair();
    }
    return repairInstructions;
};

TimedWarrantyDecorator.prototype.getPrice = function() {
    return this.bicycle.getPrice() + (40.00 * this.coverageLength);
};
```

这个例子中的getPrice和repair方法都会因保修期的长短而有所变化。如果尚在保修期内，你会得到“把自行车送到维修中心”这样的维修指示，否则被调用的将是组件的repair方法。

在此之前的那些装饰者的应用顺序并不重要。但是，它们都必须放在最后应用，或至少要放在所有其他修改repair方法的装饰者之后应用。在使用替换组件方法的装饰者时，必须留意用装饰者包装自行车的顺序。使用工厂方法可以简化这一使用过程，但不管是否使用工厂方法，如果顺序事关紧要，那么装饰者就失去了部分灵活性。本节之前所讲的所有装饰者按任何顺序应用都可以正常发挥作用，因此可以根据需要透明而又动态地添加它们。而在引入替换组件方法的装饰者之后，必须设法确保按正确的顺序应用装饰者。

#### 12.2.4 添加新方法

前面的例子所讲的修改都发生在接口所定义的方法中（组件也具有这些方法），但这并不是一种必然的要求。装饰者也可以定义新方法，不过，要想稳妥地实现这一点不容易。要想使用这些新方法，外围代码首先必须知道有这样一些新方法。由于这些新方法并不是在接口中定义的，而且它们是动态添加的，因此有必要进行类型检查，以验明用于包装组件对象的最外层装饰者。

与用新方法装饰组件对象相比，对现有方法进行修改更容易实施，而且更不容易出错，这是因为采用后一种做法时，被装饰的对象用起来与之前没什么不同，外围代码也就不需要修改。

话虽如此，在装饰者中添加新方法有时也是为类增添功能的一种强有力的手段。我们可以用这种装饰者为自行车对象增添一个按铃方法。这是一个新功能，没有装饰者自行车就不可能执行这个任务：

```
/* BellDecorator class. */

var BellDecorator = function(bicycle) { // implements Bicycle
    BellDecorator.superclass.constructor.call(this, bicycle); // Call the superclass's constructor.
}
extend(BellDecorator, BicycleDecorator); // Extend the superclass.
BellDecorator.prototype.assemble = function() {
    return this.bicycle.assemble() + ' Attach bell to handlebars.';
};
BellDecorator.prototype.getPrice = function() {
    return this.bicycle.getPrice() + 6.00;
};
BellDecorator.prototype.ringBell = function() {
    return 'Bell rung.';
};
```

167

这与先前讲过的装饰者非常相似，差别只在于它实现了ringBell这个未见于接口中的方法。用这个对象装饰的自行车现在有了新的功能：

```
var myBicycle = new AcmeComfortCruiser(); // Instantiate the bicycle.
myBicycle = new BellDecorator(myBicycle); // Decorate the bicycle object
  // with a bell.
alert(myBicycle.ringBell()); // Returns 'Bell rung.'
```

BellDecorator必须放在最后应用，否则这个新方法将无法访问。这是因为其他装饰者只能传递它们知道的方法，也即那些定义在接口中的方法。由于其他装饰者都不知道ringBell方法，如果你在添加了铃铛之后再添加前灯的话，那么BellDecorator中定义的新方法实际上会被HeadlightDecorator掩盖：

```
var myBicycle = new AcmeComfortCruiser(); // Instantiate the bicycle.
myBicycle = new BellDecorator(myBicycle); // Decorate the bicycle object
  // with a bell.
myBicycle = new HeadlightDecorator(myBicycle); // Decorate the bicycle object
  // with a headlight.
alert(myBicycle.ringBell()); // Method not found.
```

这个问题有几种解决办法。你可以在接口中添加ringBell方法，并在BicycleDecorator超类中实现它，这样一来外层的装饰者对象就会传递这个方法。这个办法并不理想，因为这样一来所有实现Bicycle接口的对象都必须实现这个方法，哪怕它是一个空方法或者根本不会被使用。另一种解决办法是用一个设置过程（set process）来创建装饰者，它可以确保如果使用了BellDecorator对象的话，这个对象一定是处于最外层的装饰者。作为一个临时性措施这个方案还不

错，但要是还有另外一个装饰者也实现了一个新方法的话，这就不管用了，因为这样一来，一次只能使用其中的一个，否则总有一个装饰者定义的所有新方法会被掩盖。

最好的解决办法是在BicycleDecorator的构造函数中添加一些代码，它们对组件对象进行检查，并为其拥有的每一个方法创建一个通道方法（pass-through method）。这样一来，如果在BellDecorator（或任何其他实现了新方法的装饰者）外再裹上另一个装饰者的话，内层装饰者定义的新方法仍然可以访问。下面的代码就采用了这个解决方案：

```
/* The BicycleDecorator abstract decorator class, improved. */

var BicycleDecorator = function(bicycle) { // implements Bicycle
    this.bicycle = bicycle;
    this.interface = Bicycle;

    // Loop through all of the attributes of this.bicycle and create pass-through
    // methods for any methods that aren't currently implemented.
    outerloop: for(var key in this.bicycle) {
        // Ensure that the property is a function.
        if(typeof this.bicycle[key] !== 'function') {
            continue outerloop;
        }
        // Ensure that the method isn't in the interface.
        for(var i = 0, len = this.interface.methods.length; i < len; i++) {
            if(key === this.interface.methods[i]) {
                continue outerloop;
            }
        }
        // Add the new method.
        var that = this;
        (function(methodName) {
            that[methodName] = function() {
                return that.bicycle[methodName]();
            };
        })(key);
    }
}
BicycleDecorator.prototype = {
    assemble: function() {
        return this.bicycle.assemble();
    },
    wash: function() {
        return this.bicycle.wash();
    },
    ride: function() {
        return this.bicycle.ride();
    },
    repair: function() {

```

```

        return this.bicycle.repair();
    },
    getPrice: function() {
        return this.bicycle.getPrice();
    }
};

```

在这个例子中，接口中的方法都如通常一样定义在BicycleDecorator的prototype中。BicycleDecorator构造函数对组件对象进行检查，并为所找到的每一个未见于接口中的方法创建一个新的通道方法。这样一来，外层的装饰者就不会掩盖内层装饰者定义的新方法，你可以自由自在地创建各种实现新方法的装饰者，而不用担心这些新方法无法访问。

## 12.3 工厂的角色

前几节中已经提到，装饰者的应用顺序有时很重要。在理想情况下，装饰者应该能够以一种完全与顺序无关的方式创建，然而这并不总是能够办到。如果必须确保某种特定顺序，那么可以为此使用工厂对象。实际上，不管顺序是否要紧，工厂都很适合于创建装饰对象。本节将重写第7章中创建的AcmeBicycleShop类的createBicycle方法，以便用户可以指定自行车要配的选件。这些选件将被转化为装饰者，并在方法返回之前被应用到新创建的自行车对象上。

原来的AcmeBicycleShop类如下所示：

```

/* Original AcmeBicycleShop factory class. */

var AcmeBicycleShop = function() {};
extend(AcmeBicycleShop, BicycleShop);
AcmeBicycleShop.prototype.createBicycle = function(model) {
    var bicycle;

    switch(model) {
        case 'The Speedster':
            bicycle = new AcmeSpeedster();
            break;
        case 'The Lowrider':
            bicycle = new AcmeLowrider();
            break;
        case 'The Flatlander':
            bicycle = new AcmeFlatlander();
            break;
        case 'The Comfort Cruiser':
            default:
                bicycle = new AcmeComfortCruiser();
    }

    Interface.ensureImplements(bicycle, Bicycle);
    return bicycle;
};

```

这个类的改进版允许用户指定想为自行车配的选件。在这里，使用工厂模式可以统揽各种类

(既包括自行车类也包括装饰者类)。把所有这些信息保存在一个地方，用户就可以把实际的类名与客户代码隔离开，这样以后添加新类或修改现有类也就更容易。下面是这个改进版的代码：

```

/* AcmeBicycleShop factory class, with decorators. */

var AcmeBicycleShop = function() {};
extend(AcmeBicycleShop, BicycleShop);
AcmeBicycleShop.prototype.createBicycle = function(model, options) {
    // Instantiate the bicycle object.
    var bicycle = new AcmeBicycleShop.models[model]();

    // Iterate through the options and instantiate decorators.
    for(var i = 0, len = options.length; i < len; i++) {
        var decorator = AcmeBicycleShop.options[options[i].name];
        if(typeof decorator !== 'function') {
            throw new Error('Decorator ' + options[i].name + ' not found.');
        }
        var argument = options[i].arg;
        bicycle = new decorator(bicycle, argument);
    }

    // Check the interface and return the finished object.
    Interface.ensureImplements(bicycle, Bicycle);
    return bicycle;
};

// Model name to class name mapping.
AcmeBicycleShop.models = {
    'The Speedster': AcmeSpeedster,
    'The Lowrider': AcmeLowrider,
    'The Flatlander': AcmeFlatlander,
    'The Comfort Cruiser': AcmeComfortCruiser
};

// Option name to decorator class name mapping.
AcmeBicycleShop.options = {
    'headlight': HeadlightDecorator,
    'taillight': TaillightDecorator,
    'bell': BellDecorator,
    'basket': BasketDecorator,
    'color': FrameColorDecorator,
    'lifetime warranty': LifetimeWarrantyDecorator,
    'timed warranty': TimedWarrantyDecorator
};

```

170

如果顺序很重要，那么可以添加一些代码，在使用选件数组实例化装饰者之前对其进行排序。用工厂实例化自行车对象有许多好处。首先，不必了解自行车和装饰者的各种类名，所有这些信息都封装在AcmeBicycleShop类中。因此添加自行车型号和选件非常容易，只要把它们添加到AcmeBicycleShop.models或AcmeBicycleShop.options数组中即可。作为一个示范，我们来比较一

下创建带装饰的自行车对象的两种不同做法。第一种做法不使用工厂：

```
var myBicycle = new AcmeSpeedster();
myBicycle = new FrameColorDecorator(myBicycle, 'blue');
myBicycle = new HeadlightDecorator(myBicycle);
myBicycle = new TaillightDecorator(myBicycle);
myBicycle = new TimedWarrantyDecorator(myBicycle, 2);
```

171

采用这种直接实例化对象的做法，与客户代码紧密耦合在一起的类不下5个。与此相比，下面所示的第二种做法使用了工厂，与客户代码耦合在一起的只有一个类，即那个工厂本身：

```
var alecsCruisers = new AcmeBicycleShop();
var myBicycle = alecsCruisers.createBicycle('The Speedster', [
  { name: 'color', arg: 'blue' },
  { name: 'headlight' },
  { name: 'taillight' },
  { name: 'timed warranty', arg: 2 }
]);
```

该工厂会对经历了最后一道装饰的对象进行接口检查，以确保它实现了正确的接口。这意味着你可以相信它创建出来的对象能做你希望它做的事。这也意味着任何使用createBicycle方法的代码只管使用它创建出来的对象就行，不必关心它是一个自行车对象还是一个装饰者对象，这是由于它们实现了同样的接口，因此对于客户代码来说它们没有本质上的区别。

最后要说的是，如果有必要的话，工厂可以对选件进行排序。某些装饰者修改组件方法的方式决定了它们需要最先或最后被应用，在此情况下工厂的这种作用尤其有用。那种会替换组件方法而不是对其进行扩充的装饰者需要放在最后创建，以确保其成为最外层的装饰者。

## 12.4 函数装饰者

装饰者并不局限于类。你也可以创建用来包装独立的函数和方法的装饰者。在某些语言中这是一种常见的技术。这种技术在Python中应用很广，以至于函数装饰者已经成了其语言核心中的内置成分。

下面是一个简单的函数装饰者的例子，这个装饰者包装了另一个函数，其作用在于将被包装者的返回结果改为大写形式：

```
function upperCaseDecorator(func) {
  return function() {
    return func.apply(this, arguments).toUpperCase();
  }
}
```

这个装饰者可以用来创建新函数，后者执行起来与普通函数没什么不同。下面的例子先定义了一个普通函数，然后将其装饰为一个新函数：

```
function getDate() {
  return (new Date()).toString();
}
getDateCaps = upperCaseDecorator(getDate);
```

getDateCaps函数的调用方式与其他函数没什么不一样。它会以大写形式返回调用结果：

```
alert(getDate()); // Returns Wed Sep 26 2007 20:11:02 GMT-0700 (PDT)
alert(getDateCaps()); // Returns WED SEP 26 2007 20:11:02 GMT-0700 (PDT)
```

172

在前面那个函数装饰者的定义中，`func.apply`的作用在于执行被包装的函数。这意味着它也可以用来包装方法：

```
BellDecorator.prototype.ringBellLoudly =
  upperCaseDecorator(BellDecorator.prototype.ringBell);

var myBicycle = new AcmeComfortCruiser();
myBicycle = new BellDecorator(myBicycle);

alert(myBicycle.ringBell()); // Returns 'Bell rung.'
alert(myBicycle.ringBellLoudly()); // Returns 'BELL RUNG.'
```

函数装饰者在对另一个函数的输出应用某种格式或执行某种转换这方面很有用处。例如，你可以创建一个用来包装自行车类的`assemble`方法的函数装饰者，让它用别的语言返回组装指示（不过这会是一个非常庞大的装饰者）。你也可以创建一个函数装饰者，用它包装那种返回数值的函数，并将该数值转换为别的数制形式。函数装饰者给程序员带来了极大的灵活性，而实现它所需要的代码比大而全的类装饰者少得多。

## 12.5 装饰者模式的适用场合

如果需要为类增添特性或职责，而从该类派生子类的解决办法并不实际的话，就应该使用装饰者模式。派生子类之所以会不实际，最常见的原因是需要增添的特性的数量和组合要求使用大量子类。自行车商店的例子就说明了这一点。这个例子中涉及7种不同的自行车选件，其中一些选件你还可以应用多次，这意味着如果不采用装饰者模式的话，要达到同样的目的需要数以千计的子类。从这个意义上讲，装饰者模式甚至可以被视为一种优化模式，因为在此场合下它节省的代码量可达几个数量级。

如果需要为对象增添特性而又不想改变使用该对象的代码的话，也可以采用装饰者模式。因为装饰者可以动态而又透明地修改对象，所以它们很适合于修改现有系统这一任务。相比卷入创建和维护子类的麻烦，创建和应用一些装饰者往往要省事得多。

## 12.6 示例：方法性能分析器

装饰者模式擅长于为各种对象增添新特性。本例要创建的装饰者可以用来包装任何对象，以便为其提供方法性能分析（method profiling）功能。我们打算在每个方法调用的前后添加一些代码，分别用于启动计时器和停止计时器并报告结果。这个装饰者必须完全透明，这样它才能应用于任何对象而又不干扰其正常的代码执行。它也应该能适用于任何对象，无论其实现的是什么接口。我们将先创建一个实现了计时功能的速成版装饰者，然后再将其改造为通用型装饰者。

173

我们需要一个用来测试的样例类。下面的ListBuilder类就是用于这个目的，其唯一的功能是在网页上创建一个有序列表：

```
/* ListBuilder class. */
var ListBuilder = function(parent, listLength) {
    this.parentEl = $(parent);
    this.listLength = listLength;
};
ListBuilder.prototype = {
    buildList: function() {
        var list = document.createElement('ol');
        this.parentEl.appendChild(list);

        for(var i = 0; i < this.listLength; i++) {
            var item = document.createElement('li');
            list.appendChild(item);
        }
    }
};
```

我们首先要创建一个专用于这个ListBuilder类的装饰者，它将记录执行buildList方法所耗用的时间。我们用console.log输出这些结果。运行这段代码时，要知道并非所有浏览器都实现了console对象：

```
/* SimpleProfiler class. */
var SimpleProfiler = function(component) {
    this.component = component;
};
SimpleProfiler.prototype = {
    buildList: function() {
        var startTime = new Date();
        this.component.buildList();
        var elapsedTime = (new Date()).getTime() - startTime.getTime();
        console.log('buildList: ' + elapsedTime + ' ms');
    }
};
```

SimpleProfiler是ListBuilder的一个装饰者。它也实现了ListBuilder中的buildList这个方法，并且在传递该方法调用的语句前后添加了一些计时用的代码。调用SimpleProfiler的方法时，将输出调用组件的同名方法所耗费的时间。为了测试这个装饰者，我们可以创建一个包含5000个元素的列表，看看要花多长时间：

```
var list = new ListBuilder('list-container', 5000); // Instantiate the object.
list = new SimpleProfiler(list); // Wrap the object in the decorator.
list.buildList(); // Creates the list and displays "buildList: 298 ms".
```

明白该装饰者的工作机制之后，现在要考虑的是如何对其进行通用化改造，使其可用于任何对象。为了达到这个目的，必须让装饰者逐一检查组件对象的所有属性，为找到的每一个方法创

建一个通道方法。这些通道方法也必须包含启动和停止计时器的代码：

```
/* MethodProfiler class. */
var MethodProfiler = function(component) {
    this.component = component;
    this.timers = {};

    for(var key in this.component) {
        // Ensure that the property is a function.
        if(typeof this.component[key] !== 'function') {
            continue;
        }

        // Add the method.
        var that = this;
        (function(methodName) {
            that[methodName] = function() {
                that.startTimer(methodName);
                var returnValue = that.component[methodName].apply(that.component,
                    arguments);
                that.displayTime(methodName, that.getElapsedTime(methodName));
                return returnValue;
            };
        })(key);
    }
}

MethodProfiler.prototype = {
    startTimer: function(methodName) {
        this.timers[methodName] = (new Date()).getTime();
    },
    getElapsedTime: function(methodName) {
        return (new Date()).getTime() - this.timers[methodName];
    },
    displayTime: function(methodName, time) {
        console.log(methodName + ': ' + time + ' ms');
    }
};
```

先从简单的部分说起。prototype中的方法负责执行计时任务。其中startTimer用于获取起始时间并以毫秒为单位保存其值<sup>①</sup>。getElapsedTime会读取起始时间，然后用当前时间减去该时间以得到耗费的时间。displayTime用于输出方法的名称以及执行该方法所耗费的时间（以毫秒为单位）。

需要仔细看看那个构造函数，特别是那个for..in循环。这个循环逐一检查组件对象的每一个属性，跳过不是方法的属性，如果遇到方法属性，则为装饰者添加一个同名的新方法。这样添加的新方法中的代码会启动计时器、调用组件的同名方法（把所有参数传递给它，并保存其返回值）、停止并显示计时器以及返回先前保存下来的组件的同名方法的返回值。这种方法的声明被

<sup>①</sup> 即自格林尼治标准时间1970年1月1日0时起到该时刻所经历的毫秒数。——译者注

包装在一个匿名函数中，其目的在于保留正确的methodName变量值<sup>①</sup>。

在测试该装饰者之前，先为ListBuilder添加一个新方法removeLists，并修改一下buildList方法。现在这两个方法都有一个参数，它可以用米说明相关列表是有序列表还是无序列表。这个ListBuilder与其第一个版本已经有了足够大的差别，用一个MethodProfiler对象来装饰它，看看结果如何：

```
var list = new ListBuilder('list-container', 5000);
list = new MethodProfiler(list);
list.buildList('ol'); // Displays "buildList: 301 ms".
list.buildList('ul'); // Displays "buildList: 287 ms".
list.removeLists('ul'); // Displays "removeLists: 10 ms".
list.removeLists('ol'); // Displays "removeLists: 12 ms".
```

这个例子出色地应用了装饰者模式。那个性能分析器完全透明，它可以为各种对象添加功能，为此并不需要从那些对象派生子类。只使用这一个装饰者类即可轻而易举地对各种各样的对象进行装饰。因为它可以透明地执行任何对它进行的方法调用，所以其他代码并不知道而且也不关心它的存在。你甚至可以在其基础上再应用其他装饰者，以实现别的功能或进行其他类型的性能跟踪。

## 12.7 装饰者模式之利

装饰者是在运行期间为对象增添特性或职责的有力工具。在自行车商店那个例子中，通过使用装饰者，你可以动态地为自行车对象添加可选的特色配件。在只有部分对象需要这些特性的情况下装饰者模式的好处尤为突出。如果不采用这种模式，那么要想实现同样的效果必须使用大量子类。

<sup>①</sup> 这个问题可以用一个简化的例子来说明：

```
var aFns = new Array(2);
for (var i = 0; i < aFns.length; ++i) {
    aFns[i] = function() { alert(i); };
}
aFns[0](); aFns[1]();
```

上例中的两个输出结果都是2。函数aFns[0]和aFns[1]拥有相同的定义作用域（即定义它们的作用域是同一个），所以它们引用的是这个作用域中的同一个变量i。在循环结束后这个i的值是2，所以两个函数执行的结果都是输出2这个值。如果把代码改为：

```
var aFns = new Array(2);
for (var i = 0; i < aFns.length; ++i) {
    (function(nIndex) { aFns[nIndex] = function() { alert(nIndex); }; })(i);
}
aFns[0](); aFns[1]();
```

那么其输出结果分别为0和1。循环体中的那个匿名函数执行了两次，每次各有一个不同的调用对象（call object）。参见《JavaScript权威指南》，所以定义函数aFns[0]和aFns[1]的作用域并不相同。因此，虽然这两个函数都引用了名为nIndex的变量，但实际上它们所引用的变量是存在于不同的作用域中的两个同名变量。在定义aFns[0]的作用域中nIndex为0，而在定义aFns[1]的作用域中nIndex为1，这就是输出结果分别为0和1的原因。——译者注

装饰者的运作过程是透明的，这就是说你可以用它包装其他对象，然后继续按之前使用那些对象的方法来使用它。从MethodProfiler这个示例中可以看到，这一切甚至可以动态实现，不用事先知道组件对象的接口。在为现有对象添砖加瓦这方面，装饰者模式为程序员带来了极大的灵活性。

## 12.8 装饰者模式之弊

装饰者模式的缺点主要表现在两个方面。首先，在遇到用装饰者包装起来的对象时，那些依赖于类型检查的代码会出问题。尽管在JavaScript中很少使用严格的类型检查，但是如果你的代码中执行了这样的检查，那么装饰者是无法匹配所需要的类型的。通常装饰者对客户代码来说是完全透明的，不过，在这种情况下，客户代码就能感知装饰者与其组件的不同。

第二，使用装饰者模式往往会增加架构的复杂程度。这种模式常常要引入许多小对象，它们看起来比较相似（参见自行车商店一例），而实际功能却大相径庭。装饰者模式往往不太容易理解，对于那些不熟悉这种模式的开发人员而言尤其如此。此外，实现具有动态接口的装饰者（如MethodProfiler）涉及的语法细节有时也会令人生畏。在设计一个使用了装饰者模式的架构时，你必须多花点心思，确保自己的代码有良好的文档说明，并且容易理解。

## 12.9 小结

本章讲述了一种既不用创建子类，又能透明、动态地为对象增添功能的设计模式。装饰者模式可以在不修改类定义的前提下用来为具体对象添加特性。我们再次研究了自行车商店的例子，并用工厂模式来创建具有多种可定制选件的自行车。我们讨论了装饰者修改其包装的对象的各种做法，以及与每种做法相关的一些注意事项。作为一个实用性的练习，我们还创建了一个具有动态接口的装饰者，它可以用来记录执行一个对象的方法所耗费的时间。

只要懂得装饰者模式的工作机制，你就会明白它是多么有用。我们用7个装饰者就完成了本来需要几千个子类才能完成的任务，这足以说明问题。鉴于装饰者的完全透明性，使用这种模式时你不用老是担心系统会因此而失灵或产生不兼容问题。这是一种不用重新定义对象就能对其进行扩充的简便手段。

176

177

**本**章要探讨的是另一种优化模式——享元（flyweight）模式。它最适合于解决因创建大量类似对象而累及性能的问题。这种模式在JavaScript中尤其有用，因为复杂的JavaScript代码可能很快就会用光浏览器的所有可用内存。通过把大量独立对象转化为少量共享对象，可以降低运行Web应用程序所需的资源数量。这种技术带来的好处可大可小。对于那些可能一连用上几天也不会重新加载的大型应用系统，任何减少内存用量的技术都有非常显著的效果。而对于那些不会在浏览器中打开那么长时间的小型网页，内存的节约就没那么重要。

## 13.1 享元的结构

刚接触享元模式的时候其工作机制可能很难理解。我们先对其结构作一鸟瞰，然后再详细讲解各个部分。

享元模式用于减少应用程序所需对象的数量。这是通过将对象的内部状态划分为内在数据（intrinsic data）和外在数据（extrinsic data）两类而实现的。内在数据是指类的内部方法所需要的信息，没有这种数据的话类就不能正常运转。外在数据则是可以从类身上剥离并存储在其外部的信息。我们可以将内在状态相同的所有对象替换为同一个共享对象，用这种方法可以把对象数量减少到不同内在状态的数量。

创建这种共享对象需要使用工厂，而不是普通的构造函数。这样做可以跟踪到已经实例化的各个对象，从而仅当所需对象的内在状态不同于已有对象时才创建一个新对象。对象的外在状态被保存在一个管理器对象中。在调用对象的方法时，管理器会把这些外在状态作为参数传入。

下面我们逐一详细讲解这些细节。

## 13.2 示例：汽车登记

假设要开发一个系统，用以代表一个城市的所有汽车。你需要保存每一辆汽车的详细情况（品牌、型号和出厂日期）及其所有权的详细情况（车主姓名、车牌号和最近登记日期）。当然，你决定把每辆汽车表示为一个对象：

```

/* Car class, un-optimized. */

var Car = function(make, model, year, owner, tag, renewDate) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.owner = owner;
    this.tag = tag;
    this.renewDate = renewDate;
};

Car.prototype = {
    getMake: function() {
        return this.make;
    },
    getModel: function() {
        return this.model;
    },
    getYear: function() {
        return this.year;
    },

    transferOwnership: function(newOwner, newTag, newRenewDate) {
        this.owner = newOwner;
        this.tag = newTag;
        this.renewDate = newRenewDate;
    },
    renewRegistration: function(newRenewDate) {
        this.renewDate = newRenewDate;
    },
    isRegistrationCurrent: function() {
        var today = new Date();
        return today.getTime() < Date.parse(this.renewDate);
    }
};

```

这个系统最初表现不错。但是随着城市人口的增长，你发现它一天天地变慢了。数以十万计的汽车对象耗尽了可用的计算资源。要想优化这个系统，可以采用享元模式减少所需对象的数目。

优化工作的第一步是把内在状态与外在状态分开。

### 13.2.1 内在状态和外在状态

将对象数据划分为内在和外在部分的过程有一定的随意性。既要维持每个对象的模块性，又想把尽可能多的数据作为外在数据处理。划分依据的选择多少有些主观性。在本例中，车的自然数据（品牌、型号和出厂日期）属于内在数据，而所有权数据（车主姓名、车牌号和最近登记日期）则属于外在数据。这意味着对于品牌、型号和出厂日期的每一种组合，只需要一个汽车对象就行。这个数目还是不少，不过与之前相比已经少于几个数量级。每个品牌-型号-出厂日期组合

对应的那个实例将被所有该类型汽车的车主共享。下面是新版Car类的代码（我们将在13.2.3节中说明外在数据的去向）：

```
/* Car class, optimized as a flyweight. */

var Car = function(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
};

Car.prototype = {
    getMake: function() {
        return this.make;
    },
    getModel: function() {
        return this.model;
    },
    getYear: function() {
        return this.year;
    }
};
```

上面的代码删除了所有外在数据。所有处理登记事宜的方法都被转移到一个管理器对象中（不过，也可以将这些方法留在原地，并为其增加对应于各种外在数据的参数）。因为现在对象的数据已被分为两大部分，所以必须用工厂来实例化它。

### 13.2.2 用工厂进行实例化

这个工厂很简单。它会检查之前是否已经创建过对应于指定品牌-型号-出厂日期组合的汽车，如果存在这样的汽车那就返回它，否则就创建一辆新车，并把它保存起来供以后使用。这就确保了对应于每个唯一的内在状态，只会创建一个实例：

```
/* CarFactory singleton. */

var CarFactory = (function() {
    var createdCars = {};

    return {
        createCar: function(make, model, year) {
            // Check to see if this particular combination has been created before.
            if(createdCars[make + '-' + model + '-' + year]) {
                return createdCars[make + '-' + model + '-' + year];
            }
            // Otherwise create a new instance and save it.
            else {
                var car = new Car(make, model, year);
                createdCars[make + '-' + model + '-' + year] = car;
                return car;
            }
        }
    };
});
```

```

        }
    }
};

})(());

```

### 13.2.3 封装在管理器中的外在状态

要完成这种优化还需要一个对象。所有那些从Car对象中删除的数据必须有个保存地点，我们用一个单体来做封装这些数据的管理器。原先的每一个Car对象现在都被分割为外在数据及其所属的共享汽车对象的引用这样两部分。Car对象与车主数据的组合称为汽车记录（car record）。管理器存储着这两方面的信息。它还包含着从原先的Car类删除的方法：

```

/* CarRecordManager singleton. */

var CarRecordManager = (function() {

    var carRecordDatabase = {};

    return {
        // Add a new car record into the city's system.
        addCarRecord: function(make, model, year, owner, tag, renewDate) {
            var car = CarFactory.createCar(make, model, year);
            carRecordDatabase[tag] = {
                owner: owner,
                renewDate: renewDate,
                car: car
            };
        },
        // Methods previously contained in the Car class.
        transferOwnership: function(tag, newOwner, newTag, newRenewDate) {
            var record = carRecordDatabase[tag];
            record.owner = newOwner;
            record.tag = newTag;
            record.renewDate = newRenewDate;
        },
        renewRegistration: function(tag, newRenewDate) {
            carRecordDatabase[tag].renewDate = newRenewDate;
        },
        isRegistrationCurrent: function(tag) {
            var today = new Date();
            return today.getTime() < Date.parse(carRecordDatabase[tag].renewDate);
        }
    };
})();

```

从Car类剥离的所有数据现在都保存在CarRecordManager这个单体的私用属性carRecordDatabase中。这个carRecordDatabase对象要比以前使用的一大批对象高效得多。那些处理所有权

事宜的方法现在也被封装在这个单体中，因为它们处理的都是外在数据。

可以看出，这种优化是以复杂性为代价的。原先有的只是一个类，而现在却变成了一个类和两个单体对象。把一个对象的数据保存在两个不同的地方这种做法也有点令人困惑。但与所解决的性能问题相比，这两点都只是小问题。如果运用得当，那么享元模式能够显著地提升程序的性能。

### 13.3 管理外在状态

管理享元对象的外在数据有许多不同的方法。使用管理器对象是一种常见做法，这种对象有一个集中管理的数据库（centralized database），用于存放外在状态及其所属的享元对象。汽车登记那个示例就采用了这种方案。其优点在于简单、容易维护。这也是一种比较轻便的方案，因为用来保存外在数据的只是一个数组或对象字面量。我们在后面那个工具提示对象示例中还要使用这种方案。

另一种管理外在状态的办法是使用组合模式。借助第9章所讲的这种模式，你可以用对象自身的层次体系来保存信息，而不需要另外使用一个集中管理的数据库。组合对象的叶节点全都可以是享元对象，这样一来这些享元对象就可以在组合对象层次体系中的多个地方被共享。对于大型的对象层次体系这非常有用，因为同样的数据用这种方案来表示时所需对象的数量要少得多。

### 13.4 示例：Web日历

为了演示用组合对象来保存外在状态的具体做法，下面我们要创建一个Web日历。首先实现的是一个未经优化的、未使用享元的版本。这是一个大型组合对象，位于最顶层的是代表年份的组合对象。它封装着代表月份的组合对象，而后者又封装着代表日期的叶对象。这是一个简单的例子，它会按顺序显示每月中的各天，还会按顺序显示一年中的各个月：

```
/* CalendarItem interface. */

var CalendarItem = new Interface('CalendarItem', ['display']);

/* CalendarYear class, a composite. */

var CalendarYear = function(year, parent) { // implements CalendarItem
    this.year = year;
    this.element = document.createElement('div');
    this.element.style.display = 'none';
    parent.appendChild(this.element);

    function isLeapYear(y) {
        return (y > 0) && !(y % 4) && ((y % 100) || !(y % 400));
    }

    this.months = [];
    // The number of days in each month.
```

```

this.numDays = [31, isLeapYear(this.year) ? 29 : 28, 31, 30, 31, 30, 31, 31, 31, 30,
               31, 30, 31];
for(var i = 0, len = 12; i < len; i++) {
  this.months[i] = new CalendarMonth(i, this.numDays[i], this.element);
}
};

CalendarYear.prototype = {
  display: function() {
    for(var i = 0, len = this.months.length; i < len; i++) {
      this.months[i].display(); // Pass the call down to the next level.
    }
    this.element.style.display = 'block';
  }
};

/* CalendarMonth class, a composite. */

var CalendarMonth = function(monthNum, numDays, parent) { // implements CalendarItem
  this.monthNum = monthNum;
  this.element = document.createElement('div');
  this.element.style.display = 'none';
  parent.appendChild(this.element);

  this.days = [];
  for(var i = 0, len = numDays; i < len; i++) {
    this.days[i] = new CalendarDay(i, this.element);
  }
};

CalendarMonth.prototype = {
  display: function() {
    for(var i = 0, len = this.days.length; i < len; i++) {
      this.days[i].display(); // Pass the call down to the next level.
    }
    this.element.style.display = 'block';
  }
};

/* CalendarDay class, a leaf. */

var CalendarDay = function(date, parent) { // implements CalendarItem
  this.date = date;
  this.element = document.createElement('div');
  this.element.style.display = 'none';
  parent.appendChild(this.element);
};

CalendarDay.prototype = {
  display: function() {
    this.element.style.display = 'block';
    this.element.innerHTML = this.date;
  }
};

```

这段代码的问题在于，你不得不为每一年创建365个CalendarDay对象。要创建一个显示10年的日历，需要实例化几千个CalendarDay对象。这些对象固然不大，但是无论什么类型的对象，如果其数目如此之多的话，都会给浏览器带来资源压力。更有效的方法是无论日历要显示多少年，都只用一个CalendarDay对象来代表所有日期。

### 13.4.1 把日期对象转化为享元

把CalendarDay对象转化为享元对象的过程很简单。首先，修改CalendarDay类本身，除去其中保存的所有数据，让这些数据（日期和父元素）成为外在数据：

```
/* CalendarDay class, a flyweight leaf. */

var CalendarDay = function() {} // implements CalendarItem
CalendarDay.prototype = {
  display: function(date, parent) {
    var element = document.createElement('div');
    parent.appendChild(element);
    element.innerHTML = date;
  }
};
```

接下来，创建日期对象的单个实例。所有CalendarMonth对象中都要使用这个实例。这里本来也可以像第一个示例那样使用工厂来创建该类的实例，不过，因为这个类只需要创建一个实例，所以直接实例化它就行：

```
/* Single instance of CalendarDay. */

var calendarDay = new CalendarDay();
```

现在外在数据成了display方法的参数，而不是类的构造函数的参数。这是享元的典型工作方式。因为在此情况下有些（或全部）数据被保存在对象之外，要想实现与之前同样的功能就必须把它们提供给各个方法。

最后，CalendarMonth类也要略作修改。原来用CalendarDay类构造函数创建该类实例的那个表达式被替换为calendarDay对象，而那些原本提供给CalendarDay类构造函数的参数现在被转而提供给display方法：

```
/* CalendarMonth class, a composite. */

var CalendarMonth = function(monthNum, numDays, parent) { // implements CalendarItem
  this.monthNum = monthNum;
  this.element = document.createElement('div');
  this.element.style.display = 'none';
  parent.appendChild(this.element);

  this.days = [];
  for(var i = 0, len = numDays; i < len; i++) {
    this.days[i] = calendarDay;
  }
}
```

```

);
CalendarMonth.prototype = {
  display: function() {
    for(var i = 0, len = this.days.length; i < len; i++) {
      this.days[i].display(i, this.element);
    }
    this.element.style.display = 'block';
  }
};

```

### 13.4.2 外在数据保存在哪里

本例没有像前面的例子那样使用一个中心数据库来保存所有从享元对象剥离的数据。实际上，其他类基本上没做什么修改：CalendarYear根本没有变，而CalendarMonth只改了两行。这都是因为组合对象的结构本身就已经包含了所有的外在数据。由于月份对象中的所有日期对象被依次存放在一个数组中，所以它知道每一个日期对象的状态，从CalendarDay构造函数中剔除的两种数据都已经存在于CalendarMonth对象中。

这就是组合模式与享元模式配合得如此完美的原因。组合对象通常拥有大量叶对象，它还保存着许多可作为外在数据处理的数据。叶对象通常只包含极少的内在数据，所以很容易被转化为共享资源。

## 13.5 示例：工具提示对象

在JavaScript对象需要创建HTML内容这种情况下，享元模式特别有用。那种会生成DOM元素的对象如果数目众多的话，会占用过多内存，使网页陷入泥沼。采用享元模式后，只需创建少许这种对象即可，所有需要这种对象的地方都可以共享它们。工具提示就是一个典型的例子。

工具提示就是在桌面应用程序中把鼠标指针移到工具图标上时出现的那种浮动文本框。它提供了一些说明信息，这样用户不用点击工具图标也能知道其用途。这在Web应用程序中非常有用，也很容易用JavaScript实现。

### 13.5.1 未经优化的 Tooltip 类

先看看未使用享元模式的Tooltip类：

```

/* Tooltip class, un-optimized. */

var Tooltip = function(targetElement, text) {
  this.target = targetElement;
  this.text = text;
  this.delayTimeout = null;
  this.delay = 1500; // in milliseconds.

  // Create the HTML.
  this.element = document.createElement('div');
  this.element.style.display = 'none';
}

```

```

this.element.style.position = 'absolute';
this.element.className = 'tooltip';
document.getElementsByTagName('body')[0].appendChild(this.element);
this.element.innerHTML = this.text;

// Attach the events.
var that = this; // Correcting the scope.
addEvent(this.target, 'mouseover', function(e) { that.startDelay(e); });
addEvent(this.target, 'mouseout', function(e) { that.hide(); });

};

Tooltip.prototype = {
  startDelay: function(e) {
    if(this.delayTimeout == null) {
      var that = this;
      var x = e.clientX;
      var y = e.clientY;
      this.delayTimeout = setTimeout(function() {
        that.show(x, y);
      }, this.delay);
    }
  },
  show: function(x, y) {
    clearTimeout(this.delayTimeout);
    this.delayTimeout = null;
    this.element.style.left = x + 'px';
    this.element.style.top = (y + 20) + 'px';
    this.element.style.display = 'block';
  },
  hide: function() {
    clearTimeout(this.delayTimeout);
    this.delayTimeout = null;
    this.element.style.display = 'none';
  }
};

```

187

Tooltip类的构造函数为mouseover和mouseout事件注册了事件监听器。这里有一个问题：这些事件监听器通常是作为触发事件的HTML元素的方法执行的，这意味着其中的this关键字指向的是该元素而不是那个Tooltip对象<sup>①</sup>，因此startDelay和hide方法是不能通过this关键字访问到的。为了解决这个问题，我们要了点小花招，具体做法是声明一个名为that的新变量，然后把this赋给它。that是一个普通变量，它不会因事件监听器是否作为HTML元素的方法调用而变，所以我们可以通过它调用Tooltip的方法。

<sup>①</sup> 用传统的方法（即设置HTML元素的onmouseover和onmouseout等属性）添加的事件处理器，都是作为HTML元素的方法调用的。用DOM事件模型中的addEventListener方法添加的事件处理器，通常也是作为HTML元素的方法调用的（不过DOM标准并没有对此做出规定）。但是用IE事件模型中的attachEvent方法添加的事件处理器却是作为全局函数调用的，所以此时事件处理器中的this关键字不是指向引发事件的HTML元素，而是指向全局对象，也即window所指的那个对象。——译者注

这个类的用法很简单。只管创建其实例，并把网页上的某个元素的引用和想要显示的文字传递给它的构造函数就行。这里使用了\$函数，它可以根据一个元素的ID值获取对元素的引用：

```
/* Tooltip usage. */

var linkElement = $('#link-id');
var tt = new Tooltip(linkElement, 'Lorem ipsum...');
```

但是，如果网页上有几百个甚至几千个元素需要用到工具提示，那会出现什么情况呢？这意味着将会出现成百上千个Tooltip类的实例，它们每个都有自己的属性、DOM元素和样式。这是非常低效的。

既然工具提示一次只能显示一个，那么为每一个工具提示对象创建一份HTML内容并没有意义。如果把Tooltip对象实现为享元，那么它只要有一个实例就行，可以让管理器对象把要显示的文字作为外在数据提供给它的方法。

### 13.5.2 作为享元的 Tooltip

把Tooltip类转化为享元需要做三件事：把外在数据从Tooltip对象中删除；创建一个用来实例化Tooltip的工厂；创建一个用来保存外在数据的管理器。在这个例子中我们还可以发挥一点创造性，用一个单体同时扮演工厂和管理器的角色。此外，由于外在数据可以作为事件监听器的一部分保存，因此没有必要使用一个中心数据库。

首先是把外在数据从Tooltip类中删除：

```
/* Tooltip class, as a flyweight. */

var Tooltip = function() {
    this.delayTimeout = null;
    this.delay = 1500; // in milliseconds.

    // Create the HTML.
    this.element = document.createElement('div');
    this.element.style.display = 'none';
    this.element.style.position = 'absolute';
    this.element.className = 'tooltip';
    document.getElementsByTagName('body')[0].appendChild(this.element);
};

Tooltip.prototype = {
    startDelay: function(e, text) {
        if(this.delayTimeout == null) {
            var that = this;
            var x = e.clientX;
            var y = e.clientY;
            this.delayTimeout = setTimeout(function() {
                that.show(x, y, text);
            }, this.delay);
        }
    }
};
```

```

    },
    show: function(x, y, text) {
        clearTimeout(this.delayTimeout);
        this.delayTimeout = null;
        this.element.innerHTML = text;
        this.element.style.left = x + 'px';
        this.element.style.top = (y + 20) + 'px';
        this.element.style.display = 'block';
    },
    hide: function() {
        clearTimeout(this.delayTimeout);
        this.delayTimeout = null;
        this.element.style.display = 'none';
    }
};

```

上面的Tooltip类删除了原来的构造函数的所有参数以及注册事件处理器的代码。而startDelay和show方法则各增加了一个新的参数，这样一来，要显示的文字就可以作为外在数据传给它们。

下一步是创建用作工厂和管理器的那个单体。我们把Tooltip类的声明放在TooltipManager这个单体中，这样它就不能在别的地方被实例化：

```

/* TooltipManager singleton, a flyweight factory and manager. */

var TooltipManager = (function() {
    var storedInstance = null;

    /* Tooltip class, as a flyweight. */

    var Tooltip = function() {
        ...
    };
    Tooltip.prototype = {
        ...
    };

    return {
        addTooltip: function(targetElement, text) {
            // Get the tooltip object.
            var tt = this.getTooltip();
            // Attach the events.
            addEvent(targetElement, 'mouseover', function(e) { tt.startDelay(e, text); });
            addEvent(targetElement, 'mouseout', function(e) { tt.hide(); });
        },
        getTooltip: function() {
            if(storedInstance == null) {
                storedInstance = new Tooltip();
            }
        }
    };
});

```

```

        return storedInstance;
    }
}

})();

```

这个单体有两个方法，分别体现了它的两种角色。getTooltip是工厂方法，它与你之前见到过的其他享元的生成方法差不多。addTooltip则是管理器方法，它先获取一个Tooltip对象，然后分别把两个匿名函数注册为目标元素的mouseover和mouseout事件监听器。这个例子用不着创建中心数据库，因为那两个匿名函数中生成的闭包已经保存了外在数据。

现在创建工具提示的代码看起来与之前的有点不一样。在这里应该调用addTooltip方法，而不是实例化Tooltip：

```

/* Tooltip usage. */

TooltipManager.addTooltip($('link-id'), 'Lorem ipsum...');


```

把Tooltip转化为享元的收获何在？现在需要生成的DOM元素已减至一个。这很重要。假如你想为工具提示添加阴影或iframe垫片（iframe shim）<sup>①</sup>等特性，那么每个Tooltip对象需要生成5到10个DOM元素。要是不把它实现为享元的话，网页将被成百上千个工具提示压垮。此外，享元模式的应用还减少了对象内部保存的数据。有了这两点改进，现在你想创建多少工具提示都行（当然，要在合理的范围内），用不着担心为此会冒出几千个挤来挤去的Tooltip实例。

## 13.6 保存实例供以后重用

模式对话框是享元模式的另一个适用场合。与工具提示一样，对话框对象也封装着数据和HTML内容。不过，后者包含的DOM元素要多得多，因此尽可能地减少其实例个数更显重要。问题在于网页上可能会同时出现不止一个对话框。实际上，你无法确切知道究竟需要多少对话框。既然如此，那又怎能得知需要用到多少实例呢？

因为运行期间需要用到的实例的确切数目无法在开发期间确定，所以不能对实例的个数加以限制，而只能要用多少就创建多少，然后把它们保存起来供以后使用。这样就不用再次承受其创建过程中的开销，而且所创建的实例的数目也刚好能满足需要。

在这个示例中，DialogBox对象的实现细节并不重要。你只需要知道，它是资源密集型的对象，应该尽量少实例化。该类的基本框架以及它实现的接口如下所示：

190

<sup>①</sup> iframe垫片是一种用来解决IE中的窗口化控件（windowed control）带来的问题的偏招。IE 7之前的IE中的select和IE 5.5之前的IE中的iframe都是所谓窗口化控件，CSS属性z-index对它们不起作用。即使把一个绝对定位的而且不透明的HTML元素定位到窗口化控件前方，这种控件也会穿透该HTML元素显示出来。这给Web前端开发中菜单和对话框的实现造成了很大的麻烦。好在从IE 5.5开始iframe就不再是窗口化控件，而且它还有一个不同于其他元素的特点，那就是它能遮住其后方的窗口化控件。因此在IE 5.5和IE 6中，在需要把一个HTML元素（比如div）定位到窗口化控件前方时，可以把一个与该元素大小相同的iframe定位到同样的位置，并且让其在z轴上位于那个HTML元素后方。这样一来，iframe可以遮住后方的窗口化控件，而它本身又会被那个HTML元素遮住，从效果上看就像是那个HTML元素遮住了后方的窗口化控件一样。这种iframe就称为iframe垫片。——译者注

```

/* DisplayModule interface. */

var DisplayModule = new Interface('DisplayModule', ['show', 'hide', 'state']);

/* DialogBox class. */

var DialogBox = function() { // implements DisplayModule
    ...
};

DialogBox.prototype = {
    show: function(header, body, footer) { // Sets the content and shows the
        ...
        // dialog box.
    },
    hide: function() { // Hides the dialog box.
        ...
    },
    state: function() { // Returns 'visible' or 'hidden'.
        ...
    }
};

```

该类实现了DisplayModule接口中定义的3个方法（show、hide和save），这里并不关心其具体实现。本例的重点在于那个控制享元数量的管理器。该管理器需要3个部件：一个用来显示对话框的方法、一个用来检查当前网页上正在使用的对话框的数目的方法，以及一个用来保存所生成的对话框的数据结构。我们用一个单体来包装这些部件，以确保管理器的唯一性<sup>①</sup>：

```

/* DialogBoxManager singleton. */

var DialogBoxManager = (function() {
    var created = []; // Stores created instances.

    return {
        displayDialogBox: function(header, body, footer) {
            var inUse = this.numberInUse(); // Find the number currently in use.
            if(inUse > created.length) {
                created.push(this.createDialogBox()); // Augment it if need be.
            }
            created[inUse].show(header, body, footer); // Show the dialog box.
        },
        createDialogBox: function() { // Factory method.
            var db = new DialogBox();
            return db;
        },
        numberInUse: function() {
            var inUse = 0;

```

<sup>①</sup> 这段代码有一些问题。首先，第7行那个if语句的判别条件应该从“inUse > created.length”改为“inUse >= created.length”。其次，即便已经改正了前面的错误，displayDialogBox这个方法的逻辑仍然不正确。它应该逐一检查数组的各个元素，用遇到的第一个当前未被使用的对话框来显示给定内容。如果所有对话框都正被使用，那么再重新创建一个对话框对象，把它添加到数组的最后，然后用这个新对象显示给定内容。——译者注

```

for(var i = 0, len = created.length; i < len; i++) {
  if(created[i].state() === 'visible') {
    inUse++;
  }
}
return inUse;
})();
```

```

这个管理器把已经创建出来的对话框对象保存在数组`created`中，以便于重用。`numberInUse`方法用于获取现有`DialogBox`对象中当前正被使用的对象的个数，它通过检查`DialogBox`对象的状态判断其是否正被使用。`displayDialogBox`方法会先检查这个数字是否不小于数组的长度，并且只有在不能重用现有实例的情况下才创建新实例。

这个示例比工具提示那个要复杂一点，但它们的工作原理是相同的。总结起来就是：通过把外在数据从资源密集型对象剥离以实现对这种对象的重用；用一个管理器控制对象的个数并保存外在数据；所生成的实例的个数应该刚好够用，并且在实例化开销较大的情况下，这些实例应被保存起来供以后重用。这种技术类似于服务端语言中的SQL连接池。在后一种技术中，仅当现有连接都在使用当中时才会创建新连接。

## 13.7 享元模式的适用场合

要想把对象转化为享元需要满足一些前提条件。最重要的一个条件就是，网页中必须使用了大量资源密集型对象。如果只会用到少许这类对象，那么这种优化并不划算。那么“大量”到底有多大？实际上浏览器内存和CPU的使用率都会制约你所能创建的对象的数量。只要对象的数目已经多到开始引起这方面的问题，那就足以应用享元模式了。

第二个条件是这些对象中所保存的数据至少有一部分能被转化为外在数据。这就是说必须能够把存储在对象内部的部分数据分离出来，然后将其作为参数提供给各个方法。此外，将这些数据存储在对象外部所占用的资源应该相对较少，否则这种做法对于性能的提升实际上毫无意义。那些包含着大量基础性代码（infrastructure code）和HTML内容的对象可能比较适合这种优化。如果一个对象只不过是一些数据和访问这些数据的方法的容器，那么结论就不是那么回事了。

最后一个条件是，将外在数据分离出去之后，独一无二的对象的数目相对较少。最理想的情况是只存在一个独一无二的对象，比如日历和工具提示那两个示例。尽管并非什么时候都能把实例数目削减到一个，但是应该尽量减少独一无二的对象的个数。在每一个这种独一无二的对象都需要有多个副本的情况下（比如对话框那个例子），这尤其重要。

192

## 13.8 实现享元模式的一般步骤

能满足上述三个条件的程序就适合用享元模式进行优化。几乎所有的享元都是按同样的步骤实现的。

(1) 将所有外在数据从目标类剥离。具体做法是尽可能多地删除该类的属性，所删除的应该是那种因实例而异的属性。构造函数的参数也要这样处理。这些参数应该被添加到该类的各个方法。这些外在数据现在不再保存在类的内部，而是由管理器提供给类的方法。经过这样的处理后，目标类应该依然具有与之前一样的功能。唯一的区别在于数据的来源发生了变化。

(2) 创建一个用来控制该类的实例化的工厂。这个工厂应该掌握该类所有已创建出来的独一无二的实例。其具体做法之一是用一个对象字面量<sup>①</sup>来保存每一个这类对象的引用，并以用来生成这些对象的参数的唯一性组合作为它们的索引。这样一来，每次要求工厂提供一个对象时，它会先检查那个对象字面量，看看以前是否请求过这个对象。如果是，那么只要返回那个现有对象的引用就行。否则它会创建一个新对象并将其引用保存在那个对象字面量中，然后返回这个对象。另一种做法称为对象池（pooling），这种技术用数组来保存所创建的对象的引用。它适合于注重可用对象的数量而不是那些单独配置的实例（uniquely configured instance）的场合。这种技术可用来将所实例化的对象的数目维持在最低值。工厂会处理根据内在数据创建对象的所有事宜。

(3) 创建一个用来保存外在数据的管理器。该管理器对象负责控制处理外在数据的种种事宜。在实施优化之前，要是需要一个目标类的实例，你会把所有数据传给构造函数以创建其新实例。而现在要是需要一个实例，你会调用管理器的某个方法，把所有数据都提供给它。这个方法会分辨内在数据和外在数据。它把内在数据提供给工厂对象以创建一个对象（或者，如果已经存在这样一个对象的话，则重用该对象）。外在数据则被保存在管理器内的一个数据结构中。管理器随后会根据需要将这些数据提供给共享对象的方法，其效果就如同该类有许多实例一样。

## 13.9 享元模式之利

享元模式可以把网页的资源负荷降低几个数量级。在工具提示那个示例中，`Tooltip`对象（以及它所生成的HTML元素）的数量被削减至一个。如果网页使用了成百上千个工具提示（对于桌面风格的大型应用程序来说这很常见），那么所能节省的资源相当可观。即使享元模式的应用无法将实例的个数削减到一个，你仍能从中获益不少。

实现这种节省并不需要大量修改原有代码。在创建了管理器、工厂和享元之后，你需要对代码进行的修改只不过是从直接实例化目标类改为调用管理器对象的某个方法。如果你创建的享元是其他程序员使用的API，那么他们只要稍稍改变自己调用API的方式就能享受到享元带来的好处。这正是享元模式出色的地方。你只要优化一下自己的API，所有使用它的人都能从中受益。如果优化的是整个网站中都在使用的一个库，那么用户会立即体会到其速度的显著提升。

<sup>①</sup> 其实在这一段中作者不应该用“对象字面量”这个词，而应该改用“对象”。这里简单解释一下其原因。在语句“`var obj = {};`”中，所谓对象字面量其实只是“`{}`”这一部分，它只是一个值。相比之下，`obj`是一个变量。把一个对象字面量赋给它，只不过是让它指向系统通过解析那个对象字面量而生成的对象而已，并不会因此就把它变成一个对象字面量。以后为`obj`增添属性时，改变的是它所指的那个对象，而不是对象字面量“`{}`”。再举一例。在C语句“`int n = 2;`”中，`n`是一个变量，而2则是一个字面量。这条语句执行之后`n`的值为2。如果在此之后再执行语句“`n = 3;`”，其结果是把变量`n`的值改为3，而不是把程序中的那个字面量2改为3。——译者注

## 13.10 享元模式之弊

享元模式只不过是一种优化模式。其作用是在满足一系列严格条件的前提下提高代码的运行效率。它不是什么地方都能用，也不应该被到处乱用。如果把它用在不必要的地方，其结果反而有损代码的运行效率。这种模式在优化代码的同时，也提高了其复杂程度，这会给调试和维护造成困难。

它之所以会妨碍调试，是因为现在可能出错的地方变成了3个：管理器、工厂和享元。相比之下，先前需要操心的只有一个对象。而且现在追踪数据问题也很困难，因为数据的来源并不总是很清楚。如果工具提示显示的文字不正确，那么其原因究竟是传入的文字不正确，还是工具提示这个共享资源没有清除上次使用的文字呢？这些类型的错误可能会造成很大的麻烦。

这种优化也会使维护变得更加困难。现在你面对的不是由封装着数据的对象构成的清晰的架构，而是一堆又碎又乱的东西，其中的数据至少分两处保存。你最好对一份数据是内在数据或外在数据的原因加以注明，因为以后负责维护代码的人对此可能并不清楚。

这些缺点并非洪水猛兽，它们的存在只是表明只有在必要的时候才应该进行这种优化。你必须在运行效率和可维护性之间进行权衡，然而这种权衡正是工程学的精髓所在。既然如此，要是拿不准是否需要使用享元模式，那么你很可能并不需要它。享元模式适合的是系统资源已经用得差不多而且明显需要进行某种优化这样一类场合。这正是其利大于弊的时候。

## 13.11 小结

本章讨论了享元模式的结构、用法和益处。这纯粹是一种优化模式，目的在于提高系统的性能和代码的效率——尤其是使用内存的效率。其实现手法是，将所有可保存在外部的数据从一个现有的类中剥离，这样一来，该类的每一个独一无二的实例就成为可在多处共享的资源。一个享元对象足以替代原来的多个对象。

要想像这样共用享元对象，必须添加一些新对象。其中包括一个工厂对象。它负责控制目标类的实例化，并且把实例的数目限制在刚好够用的范围。它还应该把先前创建的实例保存起来，以便以后要用到同样的对象时进行重用。需要添加的还包括一个管理器对象。它负责保存外在数据并将其提供给享元的方法。这样既能保留目标类的原有功能，又能大幅削减所需对象的数量。

如果运用得当，那么享元模式可以显著提高系统性能，并且大幅减少所需资源。否则，它只会让代码变得更加复杂、更难调试和维护，即便有点性能提高，也不足以补偿这些负面作用。在使用这种模式之前，必须确保你的程序满足必要的条件，而且确保这样做在性能上的收获能够超过在代码复杂程度上付出的代价。

这种模式对JavaScript程序员特别有用，因为它可以用来减少网页上所要使用的DOM元素的数量，要知道这些元素需要耗费许多内存。结合使用这种模式与组合模式等组织型模式可以开发出功能丰富的复杂Web应用系统，它们可以平稳地运行在任何现代JavaScript环境中。

194

195

**本**章讨论的是代理模式。代理(proxy)是一个对象，它可以用来控制对另一对象的访问。它与另外那个对象实现了同样的接口，并且会把任何方法调用传递给那个对象。另外那个对象通常称为本体(real subject)。代理可以代替其本体被实例化，并使其可被远程访问。它还可以把本体的实例化推迟到真正需要的时候，对于实例化比较费时的本体，或者因尺寸较大以至于不用时不宜保存在内存中的本体，这特别有用。在处理那些需要较长时间才能把数据载入用户界面的类时，代理也大有裨益。

## 14.1 代理的结构

代理模式最基本的形式是对访问进行控制。代理对象和另一个对象(本体)实现的是同样的接口。实际上工作还是本体在做，它才是负责执行所分派的任务的那个对象或类。代理对象所做的不外乎节制对本体的访问。要注意，代理对象并不会在另一对象的基础上添加方法或修改其方法(就像装饰者那样)，也不会简化那个对象的接口(就像门面元素那样)。它实现的接口与本体完全相同，所有对它进行的方法调用都会被传递给本体。

### 14.1.1 代理如何控制对本体的访问

那种根本不实现任何访问控制的代理最简单。它所做的只是把所有方法调用传递到本体。这种代理毫无用处，但它也可提供一个进一步发展的基础。

在下面的例子中，我们将创建一个代表图书馆的类。该类封装了一个Book对象(定义见第3章)目录：

```
/* From chapter 3. */

var Publication = new Interface('Publication', ['getIsbn', 'setIsbn', 'getTitle',
    'setTitle', 'getAuthor', 'setAuthor', 'display']);
var Book = function(isbn, title, author) { ... } // implements Publication

/* Library interface. */

var Library = new Interface('Library', ['findBooks', 'checkoutBook', 'returnBook']);
```

```

/* PublicLibrary class. */

var PublicLibrary = function(books) { // implements Library
    this.catalog = {};
    for(var i = 0, len = books.length; i < len; i++) {
        this.catalog[books[i].getIsbn()] = { book: books[i], available: true };
    }
};

PublicLibrary.prototype = {
    findBooks: function(searchString) {
        var results = [];
        for(var isbn in this.catalog) {
            if(!this.catalog.hasOwnProperty(isbn)) continue;
            if(searchString.match(this.catalog[isbn].getTitle()) ||
                searchString.match(this.catalog[isbn].getAuthor())) {
                results.push(this.catalog[isbn]);
            }
        }
        return results;
    },
    checkoutBook: function(book) {
        var isbn = book.getIsbn();
        if(this.catalog[isbn]) {
            if(this.catalog[isbn].available) {
                this.catalog[isbn].available = false;
                return this.catalog[isbn];
            } else {
                throw new Error('PublicLibrary: book ' + book.getTitle() +
                    ' is not currently available.');
            }
        } else {
            throw new Error('PublicLibrary: book ' + book.getTitle() + ' not found.');
        }
    },
    returnBook: function(book) {
        var isbn = book.getIsbn();
        if(this.catalog[isbn]) {
            this.catalog[isbn].available = true;
        } else {
            throw new Error('PublicLibrary: book ' + book.getTitle() + ' not found.');
        }
    }
};

```

这个类非常简单。它可以用来查书、借书和还书。下面是一个没有实现任何访问控制的 PublicLibrary类的代理：

```

/* PublicLibraryProxy class, a useless proxy. */

var PublicLibraryProxy = function(catalog) { // implements Library
    this.library = new Publiclibrary(catalog);
};

PublicLibraryProxy.prototype = {
    findBooks: function(searchString) {
        return this.library.findBooks(searchString);
    },
    checkoutBook: function(book) {
        return this.library.checkoutBook(book);
    },
    returnBook: function(book) {
        return this.library.returnBook(book);
    }
};

```

PublicLibraryProxy与PublicLibrary实现了同样的接口和同一批方法。这个类在实例化时会创建一个PublicLibrary实例并将其作为属性保存。如果调用该类的某个方法，它会通过这个属性在其PublicLibrary实例上调用同名方法。这种类型的代理也可以通过检查本体的接口并为每一个方法创建对应方法这样一种方式动态地创建。这与第12章中创建具有动态接口的装饰者的方式类似。

前面已经说过，这种类型的代理没有什么用处。在各种类型的代理中，虚拟代理（virtual proxy）是最有用的类型之一。虚拟代理用于控制对那种创建开销很大的本体的访问。它会把本体的实例化推迟到有方法被调用的时候，有时还会提供关于实例化状态的反馈。它还可以在本体被加载之前扮演其替身的角色。作为一个例子，假设PublicLibrary的实例化很慢，不能在网页加载的时候立即完成。我们可以为其创建一个虚拟代理，让它把PublicLibrary的实例化推迟到必要的时候：

```

/* PublicLibraryVirtualProxy class. */

var PublicLibraryVirtualProxy = function(catalog) { // implements Library
    this.library = null;
    this.catalog = catalog; // Store the argument to the constructor.
};

PublicLibraryVirtualProxy.prototype = {
    _initializeLibrary: function() {
        if(this.library === null) {
            this.library = new PublicLibrary(this.catalog);
        }
    },
    findBooks: function(searchString) {
        this._initializeLibrary();
        return this.library.findBooks(searchString);
    },
    checkoutBook: function(book) {
        this._initializeLibrary();
    }
};

```

```

        return this.library.checkoutBook(book);
    },
    returnBook: function(book) {
        this._initializeLibrary();
        return this.library.returnBook(book);
    }
};

```

PublicLibraryProxy和PublicLibraryVirtualProxy之间的关键区别在于后者不会立即创建PublicLibrary的实例。PublicLibraryVirtualProxy会把构造函数的参数保存起来，直到有方法被调用时才真正执行本体的实例化。这样一来，如果图书馆对象一直未被用到，那么它就不会被创建出来。虚拟代理通常具有某种能触发本体的实例化的事件。在本例中，方法调用就是触发因素。

### 14.1.2 虚拟代理、远程代理和保护代理

对于JavaScript程序员来说，虚拟代理可能是最有用的代理类型。下面我们简要介绍一下其他代理类型，并说明为什么它们对JavaScript编程不那么适用。

远程代理（remote proxy）用于访问位于另一个环境中的对象。在Java中，这意味着另一个虚拟机中的对象，或者是地球另一端的某台计算机中的对象。远程对象一般都长期存在，任何时候都可以从任何其他环境中进行访问。这种类型的代理很难照搬到JavaScript中。其原因有两个。首先，通常JavaScript运行时环境不可能长期存在。大多数JavaScript环境都委身于Web浏览器，因此随着用户的网上冲浪活动，通常每过几分钟运行时环境就会加载或卸载一次。第二，在JavaScript中无法建立到另一个运行时环境的套接字连接以访问其变量空间，即便它能长期存在。与此最接近的做法只是用JSON对方法调用进行序列化，然后用Ajax技术将结果发送给某个资源。

远程代理的一种更有可能的用途是控制对其他语言中的本体的访问。这种本体可能是一个Web服务资源，也可能是一个PHP对象。在此情况下，很难说你所用的究竟是什么模式。它既可以被视为适配器，也可以被视为远程代理。因为这是一个灰色地带，所以有必要为这种模式确定一个名称。我们决定选择远程代理这个名称，原因在于这个名称更具说明性，也更准确，而且这里所说的模式更接近于代理模式而不是适配器模式。第一个例子中对此有更多的讨论<sup>①</sup>。

保护代理也不容易照搬到JavaScript中。在其他语言中，它通常用来根据客户的身份控制对特定方法的访问。假设要为PublicLibrary类添加一些用来增加或删除目录中的书的方法。在Java中你会用一个保护代理来限制对这些方法的访问，它只允许某些类型的客户（比如图书管理员）调用这些方法，而其他类型的客户则没有这样的权利。但是在JavaScript中，你无法判断调用方法的客户的类型，因此也就不可能实现这种模式。

出于上述原因，本章集中讨论的是虚拟代理和远程代理。

200

<sup>①</sup> 原文为“*We discuss this distinction more in the first example*”。这非常令人困惑，因为本章的第一个例子与此无关。后面的例子也没有专门讨论这方面的问题。——译者注

### 14.1.3 代理模式与装饰者模式的比较

代理在许多方面都很像装饰者。装饰者和虚拟代理都要对其他对象进行包装，都要实现与被包装对象相同的接口，而且都要把方法调用传递给被包装对象。那么二者究竟有什么区别呢？

最大的区别在于装饰者会对被包装对象的功能进行修改或扩充，而代理只不过是控制对它的访问。除了有时可能会添加一些控制代码之外，代理并不会对传递给本体的方法调用进行修改。而装饰者就是为修改方法而生的。另一个区别表现在被包装对象的创建方式上。在装饰者模式中，被包装对象的实例化过程是完全独立的。这个对象创建出来之后，你可以随意为其裹上一个或更多装饰者。而在代理模式中，被包装对象的实例化是代理的实例化过程的一部分。在某些类型的虚拟代理中，这种实例化受到严格控制，它必须在代理内部进行。此外，代理不会像装饰者那样互相包装。它们一次只使用一个。

## 14.2 代理模式的适用场合

关于代理应该什么时候用，最清楚的说明可以在虚拟代理的定义中找到：虚拟代理是一个对象，用于控制对一个创建开销昂贵的资源的访问。虚拟代理是一种优化模式。如果有些类或对象需要使用大量内存保存其数据，而你并不需要在实例化完成之后立即访问这些数据，或者，其构造函数需要进行大量计算那就应该使用虚拟代理将设置开销的产生推迟到真正需要使用数据的时候。代理还可以在设置的进行过程中提供类似于“正在加载……”这样的消息，这可以形成一个反应积极的用户界面，以免让用户面对一个没有任何反馈的空白页面发呆，不知道究竟发生了什么事。

远程代理则没有这样清楚的用例。如果需要访问某种远程资源的话，那么最好是用一个类或对象来包装它，而不是一遍又一遍地手工设置XMLHttpRequest对象。问题在于应该用什么类型的对象来包装这个资源呢？这主要是个命名问题。如果包装对象实现了远程资源的所有方法，那它就是一个远程代理。如果它会在运行期间增添一些方法，那它就是一个装饰者。如果它简化了该远程资源（或多个远程资源）的接口，那它就是一个门面。远程代理是一种结构型模式，它提供了一个访问位于其他环境中的资源的原生JavaScript API（native JavaScript API）。

总而言之，如果有些类或对象的创建开销较大，而且不需要在实例化完成后立即访问其数据，那么应该使用虚拟代理。如果你有某种远程资源，并且要为该资源提供的所有功能实现对应的方法，那么应该使用远程代理。

## 14.3 示例：网页统计

本例将创建一个远程代理，它包装了一个用来提供网页统计数据的Web服务。这个Web服务由一系列URL组成，它们各相当于一个拥有可选参数的方法。它在服务器端用什么语言实现并不重要。数据将以JSON格式返回。下面是这个Web服务实现的5个方法：

```
http://mydomain.com/stats/getPageviews/  
http://mydomain.com/stats/getUniques/  
http://mydomain.com/stats/getBrowserShare/
```

```
http://mydomain.com/stats/getTopSearchTerms/
http://mydomain.com/stats/getMostVisitedPages/
```

这几个方法都有用来限制搜集统计数据的时间范围的可选参数（名为startDate和endDate），对于前面的4个方法还可以要求只要特定网页的统计数据。

你希望在整个网站中都能显示这些统计数据，但只在用户需要的时候才显示。目前的做法是为每个网页进行手工XHR调用：

```
/* Manually making the calls. */

var xhrHandler = XhrManager.createXhrHandler();

/* Get the pageview statistics. */

var callback = {
  success: function(responseText) {
    var stats = eval('(' + responseText + ')'); // Parse the JSON data.
    displayPageviews(stats); // Display the stats on the page.
  },
  failure: function(statusCode) {
    throw new Error('Asynchronous request for stats failed.');
  }
};
xhrHandler.request('GET', '/stats/getPageviews/?page=index.html', callback);

/* Get the browser statistics. */

var callback = {
  success: function(responseText) {
    var stats = eval('(' + responseText + ')'); // Parse the JSON data.
    displayBrowserShare(stats); // Display the stats on the page.
  },
  failure: function(statusCode) {
    throw new Error('Asynchronous request for stats failed.');
  }
};
xhrHandler.request('GET', '/stats/getBrowserShare/?page=index.html', callback);
```

要是能够把这些调用包装在一个对象中就好了，这个对象应该展现出一个用来访问数据的原生JavaScript接口。这样就不会有前例中那样多的重复性代码。这个对象需要实现那个Web服务中的5个方法。每个方法都会执行对Web服务的XHR调用以获取数据，然后将其提供给回调函数。

首先要做的是定义Web服务的接口。其目的在于以后有需要的时候能够换用其他类型的代理：

```
/* PageStats interface. */

var PageStats = new Interface('PageStats', ['getPageviews', 'getUniques',
  'getBrowserShare', 'getTopSearchTerms', 'getMostVisitedPages']);
```

然后定义远程代理StatsProxy本身：

```

/* StatsProxy singleton. */

var StatsProxy = (function() { // implements PageStats

    /* Private attributes. */

    var xhrHandler = XhrManager.createXhrHandler();
    var urls = {
        pageviews: '/stats/getPageviews/',
        uniques: '/stats/getUniques/',
        browserShare: '/stats/getBrowserShare/',
        topSearchTerms: '/stats/getTopSearchTerms/',
        mostVisitedPages: '/stats/getMostVisitedPages/'
    };

    /* Private methods. */

    function xhrFailure() {
        throw new Error('StatsProxy: Asynchronous request for stats failed.');
    }

    function fetchData(url, dataCallback, startDate, endDate, page) {
        var callback = {
            success: function(responseText) {
                var stats = eval('(' + responseText + ')');
                dataCallback(stats);
            },
            failure: xhrFailure
        };

        var getVars = [];
        if(startDate != undefined) {
            getVars.push('startDate=' + encodeURIComponent(startDate));
        }
        if(endDate != undefined) {
            getVars.push('endDate=' + encodeURIComponent(endDate));
        }
        if(page != undefined) {
            getVars.push('page=' + page);
        }

        if(getVars.length > 0) {
            url = url + '?' + getVars.join('&');
        }

        xhrHandler.request('GET', url, callback);
    }

    /* Public methods. */
}

```

```

return {
    getPageviews: function(callback, startDate, endDate, page) {
        fetchData(urls.pageviews, callback, startDate, endDate, page);
    },
    getUniques: function(callback, startDate, endDate, page) {
        fetchData(urls.uniques, callback, startDate, endDate, page);
    },
    getBrowserShare: function(callback, startDate, endDate, page) {
        fetchData(urls.browserShare, callback, startDate, endDate, page);
    },
    getTopSearchTerms: function(callback, startDate, endDate, page) {
        fetchData(urls.topSearchTerms, callback, startDate, endDate, page);
    },
    getMostVisitedPages: function(callback, startDate, endDate) {
        fetchData(urls.mostVisitedPages, callback, startDate, endDate);
    }
};
})();

```

这段代码使用了单体模式的两种较高级的形式，这样可以创建私用属性和方法。接口所需要的那些方法被定义为公用方法，而助理方法则被定义为私用方法。所有公用方法都调用了fetchData这个辅助方法，前面的手工实现版本中那些重复性的代码都被集中到这个方法中。

本例中使用远程代理有什么好处呢？现在实现代码与Web服务的耦合变得更松散，而重复性代码也大大减少了。对待StatsProxy对象与对待别的JavaScript对象没什么两样，你可以随意用它进行查询。不过，这的确显露了这种方法的一个弊端。远程代理，根据其定义，应该能掩盖数据的实际来源。即使你可以将其视为本地资源，它实际上还是要对服务器进行访问，根据用户的连接速度，这种访问耗费的时间少则几毫秒，多则几秒。在设计远程代理时，注明一下这种性能问题很有必要。在本例中这个问题可以通过借助回调函数进行异步调用稍加缓解，这样程序的执行就不会因为要等待调用结果而被阻塞。但是回调函数的存在多少暴露了一些下层的实现细节，因为如果不与外部服务通信的话是不需要使用回调函数的。

204

## 14.4 包装 Web 服务的通用模式

我们可以从上面的例子中提炼出一个更加通用的Web服务包装模式。尽管在设计这种代理时其具体实现细节会因Web服务的类型而异，但是这个通用模式可以为你提供一个一般性的框架。由于JavaScript的同源性限制，Web服务代理所包装的服务必须部署在使用代理的网页所在的域中。这里使用的不是一个单体，而是一个拥有构造函数的普通类，以便以后进行扩展：

```

/* WebserviceProxy class */

var WebserviceProxy = function() {
    this.xhrHandler = XhrManager.createXhrHandler();
};

WebserviceProxy.prototype = {
    _xhrFailure: function(statusCode) {

```

```

        throw new Error('StatsProxy: Asynchronous request for stats failed.');
    },
    _fetchData: function(url, dataCallback, getVars) {
        var that = this;
        var callback = {
            success: function(responseText) {
                var obj = eval('(' + responseText + ')');
                dataCallback(obj);
            },
            failure: that._xhrFailure
        };
        var getVarArray = [];
        for(varName in getVars) {
            getVarArray.push(varName + '=' + getVars[varName]);
        }
        if(getVarArray.length > 0) {
            url = url + '?' + getVarArray.join('&');
        }
        xhrHandler.request('GET', url, callback);
    }
};

```

使用这个通用模式时，只需从WebserviceProxy派生一个子类，然后再借助\_fetchData方法实现需要的方法即可。如果把StatsProxy类实现为WebserviceProxy的子类，其结果大致如下：

```

/* StatsProxy class. */

var StatsProxy = function() {} // implements PageStats
extend(StatsProxy, WebserviceProxy);
/* Implement the needed methods. */

StatsProxy.prototype.getPageviews = function(callback, startDate, endDate,
    page) {
    this._fetchData('/stats/getPageviews/', callback, {
        'startDate': startDate,
        'endDate': endDate,
        'page': page
    });
};

StatsProxy.prototype.getUniques = function(callback, startDate, endDate,
    page) {
    this._fetchData('/stats/getUniques/', callback, {
        'startDate': startDate,
        'endDate': endDate,
        'page': page
    });
};

StatsProxy.prototype.getBrowserShare = function(callback, startDate, endDate,
    page)

```

```

page) {
  this._fetchData('/stats/getBrowserShare/', callback, {
    'startDate': startDate,
    'endDate': endDate,
    'page': page
  });
};

StatsProxy.prototype.getTopSearchTerms = function(callback, startDate,
  endDate, page) {
  this._fetchData('/stats/getTopSearchTerms/', callback, {
    'startDate': startDate,
    'endDate': endDate,
    'page': page
  });
};

StatsProxy.prototype.getMostVisitedPages = function(callback, startDate,
  endDate) {
  this._fetchData('/stats/getMostVisitedPages/', callback, {
    'startDate': startDate,
    'endDate': endDate
  });
};

```

## 14.5 示例：目录查找

这次的任务是为公司网站的主页添加一个可搜索的员工目录。它应该模仿实际的员工花名册中的页面，从A开始，显示其姓氏以特定字母开头的所有员工。由于这个网页的访问量很大，所以这个解决方案必须尽量节约带宽。我们不希望这个小小的特性拖累整个网页。

206

因为在这个问题中网页的大小很重要，所以我们决定只为那些需要查看员工资料的用户加载这种数据（要知道其数据量相当大）。这样一来，那些不关心这种信息的用户就不用下载额外的数据。这是虚拟代理可以大显身手的地方，因为它能够把需要占用大量带宽的资源的加载推迟到必要的时候。我们还打算在加载员工目录的过程中向用户提供一些提示信息，以免他们盯着一个空白屏幕，猜想是不是网站出了什么问题。这种任务非常适合虚拟代理。

首先要做的是创建代理的那个本体类。它负责获取员工数据并生成用于在网页上显示这些数据的HTML内容，其显示格式类似于电话号码簿：

```

/* Directory interface. */

var Directory = new Interface('Directory', ['showPage']);

/* PersonnelDirectory class, the Real Subject */

var PersonnelDirectory = function(parent) { // implements Directory
  this.xhrHandler = XhrManager.createXhrHandler();
  this.parent = parent;
  this.data = null;
  this.currentPage = null;
}

```

```

var that = this;
var callback = {
  success: that._configure,
  failure: function() {
    throw new Error('PersonnelDirectory: failure in data retrieval.');
  }
}
xhrHandler.request('GET', 'directoryData.php', callback);
};

PersonnelDirectory.prototype = {
  _configure: function(responseText) {
    this.data = eval('(' + responseText + ')');
    ...
    this.currentPage = 'a';
  },
  showPage: function(page) {
    $('page-' + this.currentPage).style.display = 'none';
    $('page-' + page).style.display = 'block';
    this.currentPage = page;
  }
};

```

207

该类的构造函数会发出一个XHR请求以获取员工数据。其`_configure`方法会在数据返回的时候被调用，它会生成HTML元素并向其中填入数据（为简洁起见，该方法的大部分内容已被省略）。该类实现了一个目录应有的所有功能。那么为什么还要使用代理呢？原因在于，这个类在实例化过程中会加载大量数据。如果在网页加载的时候实例化这个类，那么每一个用户都不得不加载这些数据，即使他根本不使用员工目录。代理的作用就是推迟这个实例化过程。

下面先勾勒出虚拟代理类的大体轮廓，它包含了该类需要的所有方法。本例中需要实现的只有`showPage`方法和构造函数：

```

/* DirectoryProxy class, just the outline. */

var DirectoryProxy = function(parent) { // implements Directory

};

DirectoryProxy.prototype = {
  showPage: function(page) {

  }
};

```

下一步是先将这个类实现为一个无用的代理，它的每个方法所做的只是调用本体的同名方法：

```

/* DirectoryProxy class, as a useless proxy. */

var DirectoryProxy = function(parent) { // implements Directory
  this.directory = new PersonnelDirectory(parent);
};

```

```
DirectoryProxy.prototype = {
  showPage: function(page) {
    return this.directory.showPage(page);
  }
};
```

现在这个代理可以代替PersonnelDirectory的实例使用。它们可以透明地互换。不过，在此情况下你丝毫没有享受到虚拟代理的好处。要想发挥虚拟代理的作用，需要创建一个用来实例化本体的方法，并注册一个用来触发这个实例化过程的事件监听器：

```
/* DirectoryProxy class, as a virtual proxy. */

var DirectoryProxy = function(parent) { // implements Directory
  this.parent = parent;
  this.directory = null;
  var that = this;
  addEvent(parent, 'mouseover', that._initialize); // Initialization trigger.
};

DirectoryProxy.prototype = {
  _initialize: function() {
    this.directory = new PersonnelDirectory(this.parent);
  },
  showPage: function(page) {
    return this.directory.showPage(page);
  }
};
```

208

现在DirectoryProxy类的构造函数不再实例化本体，而是把这个工作推迟到\_initialize中进行。我们注册了一个事件监听器，作为这个方法的触发器。触发器的作用在于通知代理对象用户需要实例化其本体，它可以有许多实现选择。在本例中，一旦用户把鼠标指针移到目录的父容器上方，本体就会被实例化。在更复杂的解决方案中，可以先为目录生成一个空白的用户界面，一旦某个表单域处于焦点之下，它就会被初始化后的本体透明地取代。

这个例子已经接近于完工。剩下的任务只有一件，那就是提示用户当前正在加载员工目录，并且在本体创建完毕之前阻止任何方法调用：

```
/* DirectoryProxy class, with loading message. */

var DirectoryProxy = function(parent) { // implements Directory
  this.parent = parent;
  this.directory = null;
  this.warning = null;
  this.interval = null;
  this.initialized = false;
  var that = this;
  addEvent(parent, 'mouseover', that._initialize); // Initialization trigger.
};

DirectoryProxy.prototype = {
  _initialize: function() {
    this.warning = document.createElement('div');
```

```

this.parent.appendChild(this.warning);
this.warning.innerHTML = 'The company directory is loading...';

this.directory = new PersonnelDirectory(this.parent);
var that = this;
this.interval = setInterval(function() { that._checkInitialization(); }, 100);
},
_checkInitialization: function() {
  if(this.directory.currentPage != null) {
    clearInterval(this.interval);
    this.initialized = true;
    this.parent.removeChild(this.warning);
  }
},
showPage: function(page) {
  if(!this.initialized) {
    return;
  }
  return this.directory.showPage(page);
}
};

```

209

阻止对showPage的调用很容易，只需要检查一下initialized属性，仅当其值为true的时候才允许调用本体的这个方法。相比之下，在对象的加载过程中显示一条提示信息则要麻烦一点。怎样才能知道那个类什么时候加载完毕呢？可以考虑在本体类中定义一个自定义事件，然后让代理订阅这个事件。不过在本例中我们选择了一种比较简单的技术。因为PersonnelDirectory实例的currentPage属性只有在数据加载完毕之后才会被设置，所以我们每隔100毫秒检查一次这个属性，直到发现它已经被设置了为止。此时即可清除加载提示并将代理标记为已初始化。

这个虚拟代理到此已经设计完毕。这只是一个用来说明这种代理的工作机制的非常简单的例子。要是设计一个更复杂的版本的话，那个初始化检查还可以设计得再健壮一些，实例化过程的触发器也可以设计得更精巧一些。每个代理都会因为你所期待的用户交互方式而有所不同。下面要讲的是一个动态虚拟代理，可以用它作模板创建自己的代理。

## 14.6 创建虚拟代理的通用模式

JavaScript是一种非常灵活的语言。得益于此，你可以创建一个动态虚拟代理，它会检查提供给它的类的接口，创建自己的对应方法，并且将该类的实例化推迟到某些预定条件得到满足的时候。作为第一步，下面先创建这个动态代理类的壳体以及\_initialize和\_checkInitialization这两个方法。这是一个抽象类，需要派生子类并进行一些配置才能正常工作：

```

/* DynamicProxy abstract class, incomplete. */

var DynamicProxy = function() {
  this.args = arguments;
  this.initialized = false;
}

```

```

};

DynamicProxy.prototype = {
    _initialize: function() {
        this.subject = {}; // Instantiate the class.
        this.class.apply(this.subject, this.args);
        this.subject.__proto__ = this.class.prototype;

        var that = this;
        this.interval = setInterval(function() { that._checkInitialization(); }, 100);
    },
    _checkInitialization: function() {
        if(this._isInitialized()) {
            clearInterval(this.interval);
            this.initialized = true;
        }
    },
    _isInitialized: function() { // Must be implemented in the subclass.
        throw new Error('Unsupported operation on an abstract class.');
    }
};

```

210

该类的构造函数你基本上可以暂且不管，稍后我们还会回头充实它。这个类实现了三个方法。`_initialize`方法用于触发本体的实例化过程。它可以被关联到各种触发器或条件。`_checkInitialization`方法每隔一段预定的时间会被调用一次，它会调用`_isInitialized`方法，如果返回值为`true`，就将`initialized`属性设置为`true`。在初始化完成之前，代理将阻止对本体的所有方法的调用。而`_isInitialized`方法就是用来判断代理的初始化是否已经完成的。子类必须实现这个方法，因为对应于不同的本体这个方法会有所不同。

现在需要在构造函数中添加一些代码，以便针对本体类中的每一个方法为代理创建一个相应的方法。这与动态装饰者那个例子中的代码非常相似，但其中也有一些重要区别：

```

/* DynamicProxy abstract class, complete. */

var DynamicProxy = function() {
    this.args = arguments;
    this.initialized = false;

    if(typeof this.class != 'function') {
        throw new Error('DynamicProxy: the class attribute must be set before ' +
            'calling the super-class constructor.');
    }

    // Create the methods needed to implement the same interface.
    for(var key in this.class.prototype) {
        // Ensure that the property is a function.
        if(typeof this.class.prototype[key] !== 'function') {
            continue;
        }
    }
}

```

```

// Add the method.
var that = this;
(function(methodName) {
    that[methodName] = function() {
        if(!that.initialized) {
            return
        }
        return that.subject[methodName].apply(that.subject, arguments);
    };
})(key);
}
};

DynamicProxy.prototype = {
    _initialize: function() {
        this.subject = {}; // Instantiate the class.
        this.class.apply(this.subject, this.args);
        this.subject.__proto__ = this.class.prototype;

        var that = this;
        this.interval = setInterval(function() { that._checkInitialization(); }, 100);
    },
    _checkInitialization: function() {
        if(this._isInitialized()) {
            clearInterval(this.interval);
            this.initialized = true;
        }
    },
    _isInitialized: function() { // Must be implemented in the subclass.
        throw new Error('Unsupported operation on an abstract class.');
    }
};

```

最重要的区别在于，这里是在对本体类的prototype中的方法进行逐一检查，而不是对本体对象本身进行检查。这是因为此时本体还未被实例化，自然还不存在本体对象，因此在决定需要实现些什么方法时检查的是本体类而不是本体对象。在这个过程中所添加的每一个方法都由两个部分组成：先执行的是一个检查，其目的在于确保本体已经初始化；随后是对本体中同名方法的调用。

要想使用这个类，必须先从它派生子类。为了演示其用法，我们创建了一个TestProxy类，它被用作虚构的TestClass的代理：

```

/* TestProxy class. */

var TestProxy = function() {
    this.class = TestClass;
    var that = this;
    addEvent($('test-link'), 'click', function() { that._initialize(); });
    // Initialization trigger.
    TestProxy.superclass.constructor.apply(this, arguments);
}

```

```

    };
    extend(TestProxy, DynamicProxy);
    TestProxy.prototype._isInitialized = function() {
        ... // Initialization condition goes here.
    };
}

```

在子类中必须做的事有4件：将this.class设置为本体类；创建某种实例化触发器（本例的设计是在点击一个链接时进行实例化）；调用超类的构造函数（就像所有子类都要做的那样）；实现\_isInitialized方法（它应该根据本体是否已初始化返回true或false）。

这个动态代理会把本体的实例化推迟到你认为必要的时候。在实例化完成之前，代理的所有公用方法什么事都不会做。这个类可以用来包装那些需要大量计算或较长时间才能实例化的类。

## 14.7 代理模式之利

不同类型的代理有不同的好处。借助远程代理，可以把远程资源当作本地JavaScript对象使用。其益处显而易见。它减少了为访问远程资源而不得不编写的粘合性代码的数量，并且为此提供了单一的接口。如果远程资源提供的API发生了变化，需要修改的代码只有一处。它还把与远程资源相关的所有数据统一保存在一个地方，其中包括资源的URL、数据格式、命令和响应的结构。如果需要访问多个Web服务，那么可以先创建一个抽象的通用远程代理类，然后针对每一种要访问的Web服务派生出一个子类。

虚拟代理则有着截然不同的作用。它并不会减少重复性的代码和提高对象的模块性，这与本书所讲的大多数模式都不一样。实际上它还会在网页中增加一些代码，而这些代码并非必不可少。这种模式的作用体现在效率方面。它是一种优化模式，只有当资源的创建或保有开销较大，因此需要使用代理来控制它们的创建时间和方式时，才可派上用场。在这类场合下它表现得非常出色。借助这种模式，你可以使用本体的所有功能而不必操心其实例化的事。它还可以在本体加载完毕之前显示“正在加载”这样的提示信息或者显示一个虚设用户界面（dummy user interface）。在速度比较重要的网页中，虚拟代理可以用来把大对象的实例化推迟到其他元素加载完毕之后。这往往能给最终用户带来一种速度大幅提升的感觉。如果虚拟代理包装的资源没有被用到，那么它根本就不会被加载。虚拟代理的主要好处就在于，你可以用它代替其本体，而不用操心实例化开销的问题。

## 14.8 代理模式之弊

不同类型的代理具有不同的好处，但它们的弊端却是相同的。代理刻意掩盖了大量复杂行为。以远程代理为例，其背后的复杂行为包括发出XHR请求、等待响应、对响应结果进行解析以及输出收到的数据。在使用远程代理的程序员眼里，它可能就像一个本地资源，但访问它所花的时间却比访问本地资源要多出几个数量级。而且，它需要和回调函数结合使用，因为让方法直接返回结果是行不通的，这给代码增加了一定的复杂性，并且进一步拆穿了其本地资源的假象。此外，远程代理只有在能够与远程资源通信的条件下才能工作，因此其可靠性也得打点折扣。与设计模式的大多数问题一样，这里所说的问题也能通过精心编撰的程序文档加以消除。

(至少也可以减轻其不利影响)。如果程序员能明确自己在性能和可靠性方面的目标，那么在代理使用的问题上他们可以便宜行事。

对虚拟代理来说情况也是如此。它掩盖了推迟本体的实例化的逻辑。使用这种代理的程序员并不清楚有哪些操作会触发对象的实例化。这些实现细节没有必要披露出来。不过，如果程序员以为立刻就能访问到本体的话，他们可能难免吃上一惊。在此情况下高质量的文档也有一些帮助。  
213

要是用在恰当的场合，这些代理都非常有用。但如果勉强使用的话，它们也会坏事，因为此时它们给项目带来的是不必要的复杂性和代码。因为代理与其本体完全可以互换，所以如果没有令人信服的理由使用代理的话，最好还是选择直接访问本体这种简单得多的办法。在为创建一个代理费心劳力之前，请确保你确实需要它提供的特性。

## 14.9 小结

本章讨论了代理模式的各种形式。每一种形式的代理都以自己的方式控制对资源的访问。这种资源被称为本体。

保护代理根据客户身份控制对本体方法的访问。本章没有涉及这种代理，因为它无法在JavaScript中实现。

远程代理负责控制对远程资源的访问。在Java这样的语言中，远程代理所做的就是连接到一个持久存在的Java虚拟机并传递方法调用。在客户端JavaScript中不可能有这样的用法。不过，远程代理在封装用其他语言编写的Web服务方面很有用处。借助这种代理，远程资源也能像本地资源一样访问。

虚拟代理用于控制对创建或保有开销较大的类或对象的访问。它在JavaScript中非常有用，因为最终用户的浏览器可供代码运行的内存可能不多。虚拟代理也有助于解决本体的加载过程比较缓慢所带来的问题，它可以先为最终用户提供“正在加载”这样的提示信息，或者先提供一个虚拟用户界面，这样用户在最终的实际用户界面载入之前可以先与之互动。

代理任何时候都可以被替换为本体。它会增加项目的复杂性。除非它能降低你的代码的冗余程度、提高其模块化程度或运行效率，否则不要使用它。如果运用得当，那么代理能够大大简化对资源的访问，这是其他方法难以办到的。  
214

**在**事件驱动的环境中，比如浏览器这种持续寻求用户关注的环境中，观察者模式（又名发布者-订阅者（publisher-subscriber）模式）是一种管理人与其任务之间的关系（确切地讲，是对象及其行为和状态之间的关系）的得力工具。用JavaScript的话来说，这种模式的实质就是你可以对程序中某个对象的状态进行观察，并且在其发生改变时能够得到通知。

观察者模式中存在两个角色：观察者和被观察者。本书一般倾向于称其为发布者和订阅者。这种模式在JavaScript中有几种不同的实现方式，本章将对其中的一些实现方式进行考察。不过我们首先要说明一下发布者和订阅者这两种角色。下一节的例子以报业为例说明了观察者模式的工作方式。

## 15.1 示例：报纸的投送

在报纸行业中，发行和订阅的顺利进行有赖于一些关键性的角色和行为。首先是读者。他们都是订阅者（subscriber），是与你我一样的人。我们消费数据并且根据读到的消息做出反应。我们可以选择自己的居住地点，让报社把报纸送到自己家中。这个活动中的另一个角色是发行方（publisher）。他们负责出版诸如*San Francisco Chronicle*、*New York Times*和*Sacramento Bee*这样的报纸。

确定了各方的身份之后，我们就可以分析每一方的职责所在。作为报纸的订阅者，我们有一些事要做。数据到来的时候我们收到通知。我们消费数据。然后我们根据数据做出反应。只要报纸到了订阅者手中，他们就可以自行处置。有些人读完之后会将其扔在一边，有些人会向朋友或家人转述其中的新闻，甚至还有一些人会把报纸送回去。总而言之，订阅者要从发行方接收数据。

发行方则要发送数据。在本例中，发行方也是投送方（deliver）。一般说来，一个发行方很可能有许多订阅者，同样，一个订阅者也很可能会订阅多家报社的报纸。问题的关键在于，这是一种多对多的关系，需要一种高级的抽象策略，以便订阅者能够彼此独立地发生改变，而发行方能够接受任何有消费意向的订阅者。

### 15.1.1 推与拉的比较

对于报社来说，只为给几个订阅者投送报纸就满世界跑是不划算的。而纽约市的居民也不可能特意飞到旧金山去拿自己订的*San Francisco Chronicle*，要知道这份报纸可以直接投送到他们家

门口。

订阅者要想拿到报纸的话有两种投送方式可选：推或拉。在推环境中，发行方很可能会雇佣投送人员四处送报。换句话说，他们把自己的报纸推出去，让订阅者收取。在拉环境中，规模较小的本地报社可能会在订阅者家附近的街角提供自己的数据，供订阅者“拉”。那些成长型发行方没有足够的资源进行大规模投送，因此采用拉方案，让订阅者到当地的杂货店或自动售货机那里“拿”报，对于它们来说往往是个优化投送环节的好办法。

### 15.1.2 模式的实践

在JavaScript中有多种方法可以实现发布者-订阅者模式。在展示那些示例之前，我们先确保各种角色的扮演者（对象）及其行为（方法）都已就绪。

- 订阅者可以订阅和退订。他们还要接收。他们可以在“由人投送（being delivered to）”和“自己收取（receiving for themselves）”之间进行选择。
- 发布者负责投送。他们可以在“送出（giving）”和“由人取（being taken from）”之间进行选择。

下面是一个展示发布者和订阅者之间的互动过程的高层数例。它是Sells方法（Sellsian approach）<sup>①</sup>的一个示范。这种技术类似于测试驱动的开发（TDD），不过它要求先写实现代码，就像API已经写好了一样。为了让这些代码成为可运转的真正实现，程序员需要完成各种该做的工作，API由此形成：

```
/* From
http://pluralsight.com/blogs/dbox/archive/2007/01/24/45864.aspx
*/
/*
 * Publishers are in charge of "publishing" i.e. creating the event.
 * They're also in charge of "notifying" (firing the event).
*/
var Publisher = new Observable;

/*
 * Subscribers basically... "subscribe" (or listen).
 * Once they've been "notified" their callback functions are invoked.
*/
var Subscriber = function(news) {
    // news delivered directly to my front porch
};
Publisher.subscribeCustomer(Subscriber);

/*
 * Deliver a paper:
 * sends out the news to all subscribers.
```

<sup>①</sup> 这不是一个正式的术语，而是Don Box发明的一个词，用来指Chris Sells所说的一种开发方法。下面的代码中开头部分的注释中那个URL就是该词的出处。——译者注

```

*/
Publisher.deliver('extre, extre, read all about it');

/*
 * That customer forgot to pay his bill.
*/
Publisher.unSubscribeCustomer(Subscriber);

```

在这个模型中，可以看出发布者处于明显的主导地位。它们负责登记其顾客，而且有权停止为其投送。最后，新的报纸出版后它们会将其投送给顾客。

上面的代码创建了一个新的可观察（observable）对象。它有三个实例方法：subscribeCustomer、unSubscribeCustomer和deliver。subscribeCustomer方法以一个代表订阅者的回调函数为参数。deliver方法在调用过程中将通过这些回调函数把数据发送给每一个订阅者。

下面的例子处理的是同一类问题，但发布者和订阅者之间的互动方式有所不同：

```

/*
 * Newspaper Vendors
 * setup as new Publisher objects
*/
var NewYorkTimes = new Publisher;
var AustinHerald = new Publisher;
var SfChronicle = new Publisher;

/*
 * People who like to read
 * (Subscribers)
 *
 * Each subscriber is set up as a callback method.
 * They all inherit from the Function prototype Object.
*/
var Joe = function(from) {
  console.log('Delivery from '+from+' to Joe');
};
var Lindsay = function(from) {
  console.log('Delivery from '+from+' to Lindsay');
};
var Quadaras = function(from) {
  console.log('Delivery from '+from+' to Quadaras');
};

/*
 * Here we allow them to subscribe to newspapers
 * which are the Publisher objects.
 * In this case Joe subscribes to the NY Times and
 * the Chronicle. Lindsay subscribes to NY Times
 * Austin Herald and Chronicle. And the Quadaras
 * respectfully subscribe to the Herald and the Chronicle
*/

```

```

Joe.
  subscribe(NewYorkTimes).
  subscribe(SfChronicle);

Lindsay.
  subscribe(AustinHerald).
  subscribe(SfChronicle).
  subscribe(NewYorkTimes);

Quadaras.
  subscribe(AustinHerald).
  subscribe(SfChronicle);

/*
 * Then at any given time in our application, our publishers can send
 * off data for the subscribers to consume and react to.
 */
NewYorkTimes.
  deliver('Here is your paper! Direct from the Big apple');

AustinHerald.
  deliver('News').
  deliver('Reviews').
  deliver('Coupons');

SfChronicle.
  deliver('The weather is still chilly').
  deliver('Hi Mom! I\'m writing a book');

```

在这个例子中，发布者的创建方式和订阅者接收数据的方式没有多少改变，但拥有订阅和退订权的一方变成了订阅者。当然，负责发送数据的还是发布者一方。

本例中的发布者是Publisher类的实例。它有一个deliver方法。而作为订阅者的函数对象则拥有subscribe和unsubscribe两个方法。订阅者只是普通的回调函数，那两个方法是通过扩展Function的prototype而加入的<sup>①</sup>。

下面我们将一步一步地构建出符合需要的API。

## 15.2 构建观察者 API

在明确了观察者模式中的核心成员之后，现在可以着手构建其API了。首先，我们需要一个

218 发布者的构造函数，它为该类实例定义了一个类型为数组的属性，用来保存订阅者的引用：

```

function Publisher() {
  this.subscribers = [];
}

```

### 15.2.1 投送方法

所有Publisher实例都应该能够投送数据。只要把deliver方法添加到Publisher的prototype

<sup>①</sup> 因为任何函数都是Function的实例，所以在Function.prototype中添加的新方法会被所有函数继承。当然，这里把subscribe和unsubscribe添加到Function.prototype中只是为了省事，在实际项目中不应该这样做。——译者注

中，它就能够被所有Publisher对象共享：

```
Publisher.prototype.deliver = function(data) {
  this.subscribers.forEach(
    function(fn) {
      fn(data);
    }
  );
  return this;
};
```

这个方法使用JavaScript 1.6中新增的数组方法forEach（参见“Mozilla开发人员中心”网站<http://developer.mozilla.org/>）逐一处理每一个订阅者。forEach方法会对一个“草垛（haystack）”从头到尾访问一遍，把每一根“针（needle）”、“针”的索引和整个数组提供给一个回调方法<sup>①</sup>。订阅者数组中的每根“针”都是一个回调函数，比如Joe、Lindsay和Quadaras。

deliver方法把this用作返回值，因此可以对该方法进行链式调用（参见第6章），以连续不断地投送数据。

### 15.2.2 订阅方法

下一步是给予订阅者订阅的能力：

```
Function.prototype.subscribe = function(publisher) {
  var that = this;
  var alreadyExists = publisher.subscribers.some(
    function(el) {
      return el === that;
    }
  );
  if ( !alreadyExists ) {
    publisher.subscribers.push(this);
  }
  return this;
};
```

这段代码为Function的prototype添加了一个以Publisher对象为参数的subscribe方法，因此所有函数都能调用这个方法。subscribe方法先定义了一个that变量，并把this赋给它。后面用作数组的some方法参数的那个匿名函数将通过闭包机制访问到这个变量，从而访问到用以调用subscribe方法的那个函数对象。some也是JavaScript 1.6中新增的数组方法，它以一个回调函数为参数。some方法逐一访问数组的各个元素，并以其为参数调用那个回调函数。只要至少有一次调用回调函数时返回true，则some方法返回true，否则some方法将返回false。subscribe方法把some方法的返回值赋给变量alreadyExists，然后根据这个变量的值决定是否为指定的发布者添加一

<sup>①</sup> 这里所谓的“草垛”和“针”分别指数组及其元素。这里所说的回调函数，是指forEach方法的那个参数。这个回调函数本身又有三个参数，依次为当前数组元素（即“针”）、当前数组元素的索引值、数组。本例中的那个回调函数是一个匿名函数，它只使用了第一个参数。注意，本段最后那句话“订阅者数组中的每根‘针’都是一个回调函数”中的“回调函数”与前面说的那个回调函数不是一回事。——译者注

个订阅者。最后，`subscribe`方法返回`this`值，以便支持该方法的链式调用。

### 15.2.3 退订方法

`unsubscribe`方法可供订阅者用来停止对指定发布者的观察：

```
Function.prototype.unsubscribe = function(publisher) {
  var that = this;
  publisher.subscribers = publisher.subscribers.filter(
    function(el) {
      return el !== that;
    }
  );
  return this;
};
```

有些订阅者在监听到某种一次性的事件之后会在回调阶段立即退订该事件。其做法大致如下：

```
var publisherObject = new Publisher;

var observerObject = function(data) {
  // process data
  console.log(data);
  // unsubscribe from this publisher
  arguments.callee.unsubscribe(publisherObject);
};

observerObject.subscribe(publisherObject);
```

## 15.3 现实生活中的观察者

在现实世界中，观察者模式对于那种由许多JavaScript程序员合作开发的大型程序特别有用。它可以提高API的灵活性，使并行开发的多个实现能够彼此独立地进行修改。作为开发人员，你可以对自己的应用程序中什么是“令人感兴趣的时刻”做出决定。你所能监听的不再只是`click`、`load`、`blur`和`mouseover`等浏览器事件。在富用户界面（rich UI）应用程序中，`drag`（拖动）、`drop`（拖放）、`moved`（移动）、`complete`（完成）和`tabSwitch`（标签切换）都可能是令人感兴趣的事件。它们都是在普通浏览器事件的基础上抽象出来的可观察事件，可由发布者对象向其监听者广播。

220

## 15.4 示例：动画

动画是在应用程序中实现可观察对象的一个很好的起点。眨眼间你就能想出至少3个可观察到的时刻：开始、结束和进行中。在本例中，我们将分别称之为`onStart`、`onComplete`和`onTween`。下面的代码演示了用前面编写的`Publisher`工具实现这些事件的过程：

```
// Publisher API
var Animation = function(o) {
  this.onStart = new Publisher,
  this.onComplete = new Publisher,
```

```

this.onTween = new Publisher;
};

Animation.

method('fly', function() {
  // begin animation
  this.onStart.deliver();
  for ( ... ) { // loop through frames
    // deliver frame number
    this.onTween.deliver(i);
  }
  // end animation
  this.onComplete.deliver();
});

// setup an account with the animation manager
var Superman = new Animation({...config properties...});

// Begin implementing subscribers
var putOnCape = function(i) { };
var takeOffCape = function(i) { };

putOnCape.subscribe(Superman.onStart);
takeOffCape.subscribe(Superman.onComplete);

// fly can be called anywhere
Superman.fly();
// for instance:
addEvent(element, 'click', function() {
  Superman.fly();
});

```

可以看到，如果你是负责实现为超人披上斗篷和解下斗篷的功能的人的话，这种运作方式还真不错。借助于发布者，你可以知道超人什么时候准备起飞以及什么时候回到地面。你只需要预订这些时刻的通知便万事大吉。

221

## 15.5 事件监听器也是观察者

在DOM脚本编程环境中的高级事件模式中，事件监听器说到底就是一种内置的观察者。事件处理器（handler）与事件监听器（listener）并不是一回事。前者说穿了就是一种把事件传给与其关联的函数的手段<sup>①</sup>。而且在这种模型中一种事件只能指定一个回调方法。而在监听器模式中，一个事件可以与几个监听器关联。每个监听器都能独立于其他监听器而改变。打个比方，对 *San Francisco Chronicle* 这家报社来说，其订阅者 Joe 订没订 *New York Times* 都无所谓。同样，Joe 也不在乎 Lindsay 是否也订了 *San Francisco Chronicle*。每一方都只管处理自己的数据和相关的行为。

<sup>①</sup> 原文为“a handler is essentially a means of passing the event along to a function to which it is assigned”。这种说法其实不太严谨。读者不必太过当真。——译者注

例如，使用事件监听器，可以让多个函数响应同一个事件：

```
// example using listeners
var element = $('example');
var fn1 = function(e) {
  // handle click
};
var fn2 = function(e) {
  // do other stuff with click
};

addEvent(element, 'click', fn1);
addEvent(element, 'click', fn2);
```

但用事件处理器就办不到：

```
// example using handlers
var element = document.getElementById('b');
var fn1 = function(e) {
  // handle click
};
var fn2 = function(e) {
  // do other stuff with click
};

element.onclick = fn1;
element.onclick = fn2;
```

在第一个例子中，由于使用的是事件监听器，所以click事件发生时fn1和fn2都会被调用。而第二个例子使用的是事件处理器，其中第二次对onclick赋值的结果是fn1被fn2取代，因此click事件发生时只会调用fn2，不会调用fn1。

言归正传。监听器和观察者之间的共同之处显而易见。实际上它们互为同义语。它们都订阅特定的事件，然后等待事件的发生。事件发生时，订阅方的回调函数会得到通知。传给它们的参数是一个事件对象，其中包含着事件发生时间、事件类型和事件发源地等有用的信息。

## 15.6 观察者模式的适用场合

如果希望把人的行为和应用程序的行为分开，那么观察者模式正适用于这种场合。最好不要实现一些与用户操作绑在一起而且来源于浏览器的东西，比如click、mouseover或keypress之类的基本DOM事件<sup>①</sup>。对于那些只关心动画的开始，或者错别字的发现（在拼写检查应用程序）的程序员而言，那些事件提供不了什么有用信息。

举个例来说，用户点击导航系统的一个标签（tab）时，会打开一个包含着更多相关信息的菜单。当然你可以直接监听这个click事件，不过这需要知道监听的是哪个元素。这样做的另一

<sup>①</sup> 原文为“*It's best not to implement something that is tied to user interaction and originates from the browser, such as basic DOM events like click, mouseover, or keypress*”。坦率地说，我认为这不太说得通。我姑且按字面意思译出来。请读者自行判断。——译者注

一个弊端是你的实现与click事件直接绑在了一起。比监听click事件更好的做法是：创建一个可观察的onTabChange对象，并且在特定事件发生时通知所有观察者。如果菜单改为在鼠标指向标签时或者标签处于焦点之下时打开，那么这个onTabChange对象会替你处理这种改变<sup>①</sup>。

## 15.7 观察者模式之利

观察者模式是开发基于行为（action-based）的大型应用程序的有力手段。在一次浏览器会话期间，应用程序中可能会断断续续地发生几十次、几百次甚至几千次各种事件。你可以削减为事件注册监听器的次数，让可观察对象借助一个事件监听器替你处理各种行为并将信息委托（delegate）给它的所有订阅者，从而降低内存消耗和提高互动性能。这样一来，就不用没完没了地为同样的元素增添新的事件监听器，这有利于减少系统开销并提高程序的可维护性。

## 15.8 观察者模式之弊

使用这种观察者接口的一个不利之处在于创建可观察对象所带来的加载时间开销。这可以通过使用惰性加载技术加以化解，具体而言就是把新的可观察对象的实例化推迟到需要发送事件通知的时候。这样一来，订阅者在事件尚未创建的时候就能订阅它，应用程序的初始加载时间也就不会受到影响。

## 15.9 小结

观察者模式是对应用系统进行抽象的有力手段。你可以定义一些事件供其他开发人员使用，而并不需要为此深入了解他们的代码。一个事件可以被5个订阅者订阅，而一个订阅者也可以订阅5个不同的事件。对于浏览器这类互动环境来说这非常理想。现在的Web应用程序越来越大，在此背景下，作为一种提高代码的可维护性和简洁性的有力手段，可观察对象的作用更显突出。这种模式的应用有助于防止第三方开发人员和合作伙伴因为对你的应用程序的细节了解得太多而把事情搞糟。实践一下观察者模式吧，这可以给你的朋友和上司留下深刻印象。

在Publisher那个工具中，我们开发的是一种“推”系统。发布者广播事件的方式是把数据推给每个订阅者。你可以自己尝试写个工具，让每个订阅者从发布者那里“拉”数据。我们可以给你一点提示：先为订阅者实现一个“拉”方法，该方法以一个Publisher对象为参数。

223

<sup>①</sup> 也即，只需要为此修改onTabChange的实现细节即可，不必修改onTabChange的观察者。——译者注

# 命 令 模 式

**本**章研究的是一种封装方法调用的方式。命令模式与普通函数有所不同。它可以用来对方法调用进行参数化处理和传送，经这样处理过的方法调用可以在任何需要的时候执行。它也可以用来消除调用操作（action）的对象和实现操作的对象之间的耦合，这为各种具体的类的更换带来了极大的灵活性。这种模式可以用在许多不同场合，不过它在创建用户界面这一方面非常有用，特别是在需要不受限的（unlimited）取消（undo）操作的时候。它还可以用来替代回调函数，因为它能够提高在对象之间传递的操作的模块化程度。

在后面的几节中，我们将讨论命令模式的结构，并提供一些演示其在JavaScript中的用法的示例。我们还会指出哪些地方最适合使用命令模式，以及哪些场合不应该使用它。

## 16.1 命令的结构

最简形式的命令对象是一个操作和用以调用这个操作的对象的结合体。所有的命令对象都有一个执行操作（execute operation），其用途就是调用命令对象所绑定的操作（action）。在大多数命令对象中，这个操作是一个名为execute或run的方法。使用同样接口的所有命令对象都可以被同等对待，并且可以随意互换。这是命令模式的魅力之一。

为了演示命令模式的典型用法，我们来考察一个有关动态用户界面的例子。假设你有一个广告公司，你想设计一个网页，客户可以在上面执行一些与自己的账户相关的操作，比如启用和停用某些广告。因为不知道其中的具体广告数量，所以你想设计一个尽可能灵活的用户界面（UI）。为此你打算用命令模式来弱化按钮之类的用户界面元素与其操作之间的耦合。

首先要做的是定义一个所有命令对象都必须实现的接口：

```
/* AdCommand interface. */  
  
var AdCommand = new Interface('AdCommand', ['execute']);
```

接下来需要定义两个类，分别用来封装广告的start方法和stop方法：

```
/* StopAd command class. */  
  
var StopAd = function(adObject) { // implements AdCommand  
    this.ad = adObject;  
};  
StopAd.prototype.execute = function() {
```

```

    this.ad.stop();
};

/* StartAd command class. */

var StartAd = function(adObject) { // implements AdCommand
    this.ad = adObject;
};
StartAd.prototype.execute = function() {
    this.ad.start();
};

```

这是两个非常典型的命令类。它们的构造函数以另一个对象为参数，而它们实现的execute方法则要调用该对象的某个方法。现在有了两个可用在用户界面中的类，它们具有相同的接口。你不知道也不关心adObject的具体实现细节，只要它实现了start和stop方法就行。借助于命令模式，可以实现用户界面对象与广告对象的隔离。

下面的代码创建的用户界面中，用户名下的每个广告都有两个按钮，分别用于启动和停止广告的轮播：

```

/* Implementation code. */

var ads = getAds();
for(var i = 0, len = ads.length; i < len; i++) {
    // Create command objects for starting and stopping the ad.
    var startCommand = new StartAd(ads[i]);
    var stopCommand = new StopAd(ads[i]);

    // Create the UI elements that will execute the command on click.
    new UiButton('Start ' + ads[i].name, startCommand);
    new UiButton('Stop ' + ads[i].name, stopCommand);
}

```

UiButton类的构造函数有两个参数，一个是按钮上的文字，另一个是命令对象。它会在网页上生成一个按钮，该按钮被点击时会执行那个命令对象的execute方法。这个类也不需要知道所用命令对象的确切实现。因为所有命令对象都实现了execute方法，所以可以把任何一种命令对象提供给UiButton，后者应该知道如何跟它打交道。这有助于创建高度模块化和低度耦合的用户界面。

226

### 16.1.1 用闭包创建命令对象

还有另外一种办法可以用来封装函数。这种办法不需要创建一个具有execute方法的对象，而是把想要执行的方法包装在闭包中。如果想要创建的命令对象像前例中那样只有一个方法，那么这种办法尤其方便。现在你不再调用execute方法，因为那个命令可以作为函数直接执行。这样做还可以省却作用域和this关键字的绑定这方面的烦恼。

下面的代码用这种技术重写了前面的例子：

```

/* Commands using closures. */

function makeStart(adObject) {
    return function() {
        adObject.start();
    };
}

function makeStop(adObject) {
    return function() {
        adObject.stop();
    };
}

/* Implementation code. */

var startCommand = makeStart(ads[0]);
var stopCommand = makeStop(ads[0]);

startCommand(); // Execute the functions directly instead of calling a method.
stopCommand();

```

这些命令函数可以像命令对象一样四处传递，并且在需要的时候执行。它们是正式的命令对象类的简单替代品，但并不适合于需要多个命令方法的场合，比如本章后面实现取消功能的那个示例。

### 16.1.2 客户、调用者和接收者

到此你对命令模式已经有了一个大概了解，现在我们再做点正式说明。这个系统中有三个参与者：客户（client）、调用者（invoking object）和接收者（receiving object）。客户负责实例化命令并将其交给调用者。在前面的例子中，for循环中的代码就是客户。它通常被包装为一个对象，但也不是非这样不可。调用者接过命令并将其保存下来。它会在某个时候调用该命令对象的execute方法，或者将其交给另一个潜在的调用者。前例中的调用者就是UIButton类创建的按钮。用户点击它的时候，它就会调用命令对象的execute方法。接收者则是实际执行操作的对象。调用者进行“commandObject.execute()”这种形式的调用时，它所调用的方法将转而以“receiver.action()”这种形式调用恰当的方法。前例中的接收者就是广告对象，它所能执行的操作要么是start方法，要么是stop方法。

**227** 什么参与者执行什么任务有时不太好记。这里再重复一遍：客户创建命令；调用者执行该命令；接收者在命令执行时执行相应操作。除客户外的其他两个参与者的名称在一定程度上揭示了其作用，这有助于记忆。

所有使用命令模式的系统都有客户和调用者，但不一定有接收者。有些复杂（但是模块化程度较低）的命令并不调用接收者的方法，而是自己执行一些复杂查询或命令。我们将在16.2节中详细讨论这种类型的命令。

### 16.1.3 在命令模式中使用接口

命令模式需要用到某种类型的接口。接口的作用在于确保接收者实现了所需要的操作，以及

命令对象实现了正确的执行操作（它可能有各种各样的名称，不过通常叫execute、run，在某些情况下也可能叫undo）。不进行这种检查的代码会比较脆弱，容易在运行期间出现很难排查的错误。你可以在自己的代码中统一定义一个Command接口，但凡使用命令对象的地方，都要检查它是否实现了这个接口。这样一来，其中所有命令对象的执行操作都具有相同的名称，因此无需修改即可交换使用。这个接口大体形如：

```
/* Command interface. */

var Command = new Interface('Command', ['execute']);
```

有了这个接口，你就可以用类似于下面的代码检查命令对象是否实现了正确的执行操作：

```
/* Checking the interface of a command object. */

// Ensure that the execute operation is defined. If not, a descriptive exception
// will be thrown.
Interface.ensureImplements(someCommand, Command);

// If no exception is thrown, you can safely invoke the execute operation.
someCommand.execute();
```

如果用闭包来创建命令函数，那么这种检查甚至更简单，只需要检查该命令是否为函数即可：

```
If(typeof someCommand != 'function') {
    throw new Error('Command isn't a function');
}
```

在第一组示例中，为了简单起见，并没有进行接口检查。但本章后面的示例都进行了这种检查。我们强烈建议你也使用接口检查。

## 16.2 命令对象的类型

所有类型的命令对象执行的都是同样的任务：隔离调用操作的对象与实际实施操作的对象。这个定义所涵盖的区间有两种极端情况。前面创建的那种命令对象属于区间的一端，这种情况下的命令对象所起的作用只不过是把现有接收者的操作（广告对象的start和stop方法）与调用者（按钮）绑定在一起。这类命令对象最简单，其模块化程度也最高。它们与客户、接收者和调用者之间只是松散地耦合在一起：

```
/* SimpleCommand, a loosely coupled, simple command class. */

var SimpleCommand = function(receiver) { // implements Command
    this.receiver = receiver;
};

SimpleCommand.prototype.execute = function() {
    this.receiver.action();
};
```

位于区间另一端的则是那种封装着一套复杂指令的命令对象。这种命令对象实际上没有接收者，因为它自己提供了操作的具体实现。它并不把操作委托给接收者实现，所有用于实现相关操作的代码都包含在其内部：

```

/* ComplexCommand, a tightly coupled, complex command class. */

var ComplexCommand = function() { // implements Command
    this.logger = new Logger();
    this.xhrHandler = XhrManager.createXhrHandler();
    this.parameters = {};
};

ComplexCommand.prototype = {
    setParameter: function(key, value) {
        this.parameters[key] = value;
    },
    execute: function() {
        this.logger.log('Executing command');
        var postArray = [];
        for(var key in this.parameters) {
            postArray.push(key + '=' + this.parameters[key]);
        }
        var postString = postArray.join('&');
        this.xhrHandler.request(
            'POST',
            'script.php',
            function() {},
            postString
        );
    }
};

```

在这两种极端之间存在一个灰色地带。有些命令对象不但封装了接收者的操作，而且其 execute方法中也具有一些实现代码。这类命令对象位于定义区间的中间地段：

```

/* GreyAreaCommand, somewhere between simple and complex. */

var GreyAreaCommand = function(receiver) { // implements Command
    this.logger = new Logger();
    this.receiver = receiver;
};

GreyAreaCommand.prototype.execute = function() {
    this.logger.log('Executing command');
    this.receiver.prepareAction();
    this.receiver.action();
};

```

这些类型的命令对象各有各的用处，它们都能在项目中找到自己的位置。简单命令对象一般用来消除两个对象（接收者和调用者）之间的耦合，而复杂命令对象则一般用来封装不可分的（atomic）或事务性（transactional）的指令。本章着重讨论简单命令对象。

### 16.3 示例：菜单项

这个示例演示了如何用最简类型的命令对象构建模块化的用户界面。我们将设计一个用来生成桌面应用程序风格的菜单栏的类，并且通过使用命令对象，让这些菜单执行各种各样的操作。借助

于命令模式，我们可以把调用者（菜单项）与接收者（实际执行操作的对象）隔离开。那些菜单项不必了解接收者的用法，它们只要知道所有命令对象都实现了一个execute方法就行。这意味着同样的命令对象也可以被工具栏图标等其他用户界面元素使用，而且并不需要为此进行修改。

这里没有给出接收者类的实现代码。其出发点在于你只需要知道接收者有些什么操作可供调用即可。图16-1显示了各种接收者类及其实现的方法。

| FileActions | EditActions | InsertActions |
|-------------|-------------|---------------|
| open()      | cut()       | textBlock()   |
| close()     | copy()      |               |
| save()      | paste()     |               |
| saveAs()    | delete()    |               |

| HelpActions |
|-------------|
| showHelp()  |

图16-1 接收者类支持的方法

前面说过，接口在命令模式中起着非常重要的作用。这种作用在本例中尤其突出，这是因为我们还要为菜单使用组合模式，而组合对象又严重依赖于接口。本例定义了三个接口：

```
/* Command, Composite and MenuObject interfaces. */

var Command = new Interface('Command', ['execute']);
var Composite = new Interface('Composite', ['add', 'remove', 'getChild',
    'getElement']);
var MenuObject = new Interface('MenuObject', ['show']);
```

### 16.3.1 菜单组合对象

接下来要实现的是MenuBar、Menu和MenuItem类。作为一个整体，它们要能显示所有可用操作，并且根据要求调用这些操作。MenuBar和Menu都是组合对象类，而MenuItem则是叶类。MenuBar类保存着所有的Menu实例：

```
/* MenuItem class, a composite. */

var MenuItem = function() { // implements Composite, MenuObject
    this.element = document.createElement('li');
    this.element.style.listStyleType = 'none';
};

MenuItem.prototype = {
    add: function(item) {
        item.parent = this;
        this.element.appendChild(item.element);
    },
    remove: function(item) {
        item.parent.removeChild(item.element);
    },
    getChild: function(index) {
        return this.children[index];
    }
};
```

```

},
getElement: function() {
    return this.element;
},

show: function() {
    this.element.style.display = 'block';
    for(name in this.menus) { // Pass the call down the composite.
        this.menus[name].show();
    }
}
};

```

MenuBar是一个很简单的组合对象类。它会生成一个无序列表标签，并且提供了向这个列表中添加菜单对象的方法。Menu类与此大体类似，不过它管理的是MenuItem实例：

```

/* Menu class, a composite. */

var Menu = function(name) { // implements Composite, MenuObject
    this.name = name;
    this.items = {};
    this.element = document.createElement('li');
    this.element.innerHTML = this.name;
    this.element.style.display = 'none';
    this.container = document.createElement('ul');
    this.element.appendChild(this.container);
};

Menu.prototype = {
    add: function(menuItemObject) {
        Interface.ensureImplements(menuItemObject, Composite, MenuObject);
        this.items[itemObject.name] = menuItemObject;
        this.container.appendChild(this.items[itemObject.name].getElement());
    },
    remove: function(name) {
        delete this.items[name];
    },
    getChild: function(name) {
        return this.items[name];
    },
    getElement: function() {
        return this.element;
    },

    show: function() {
        this.element.style.display = 'block';
        for(name in this.items) { // Pass the call down the composite.
            this.items[name].show();
        }
    }
};

```

值得一提的是，Menu类的items属性只起着一个查找表的作用，它不会保存菜单项的次序信息。菜单项的次序由DOM负责保持。每一条新添加的菜单项，都被添加在已有菜单项之后。如果要求能对菜单项的次序进行重排，那么可以把items属性实现为数组。

真正让人感兴趣的是MenuItem类。这是系统中的调用者类。MenuItem的实例被用户点击时，会调用与其绑定在一起的命令对象。为此需要先确保传入构造函数的命令对象实现了execute方法，然后再在为MenuItem对象对应的锚标签注册的click事件处理器中加入调用它的语句：

```
/* MenuItem class, a leaf. */

var MenuItem = function(name, command) { // implements Composite, MenuObject
    Interface.ensureImplements(command, Command);
    this.name = name;
    this.element = document.createElement('li');
    this.element.style.display = 'none';
    this.anchor = document.createElement('a');
    this.anchor.href = '#'; // To make it clickable.
    this.element.appendChild(this.anchor);
    this.anchor.innerHTML = this.name;

    addEvent(this.anchor, 'click', function(e) { // Invoke the command on click.
        e.preventDefault();
        command.execute();
    });
};

MenuItem.prototype = {
    add: function() {},
    remove: function() {},
    getChild: function() {},
    getElement: function() {
        return this.element;
    },

    show: function() {
        this.element.style.display = 'block';
    }
};
```

232

命令模式的作用在此开始显现出来。你可以创建一个包含着许多菜单的非常复杂的菜单栏，而每个菜单都包含着一些菜单项。这些菜单项对如何执行自己所绑定的操作一无所知，它们也不需要知道那些细节。它们唯一需要知道的就是命令对象有一个execute方法。

每个MenuItem都与一个命令对象绑定在一起。这个命令对象不能再改变，因为它被封装在一个事件监听器的闭包中。如果想改变菜单项所绑定的命令，必须另外创建一个新的MenuItem对象。

### 16.3.2 命令类

MenuCommand这个命令类非常简单，几乎没有比这更简单的命令类了。其构造函数的参数就是将被作为操作而调用的方法。因为JavaScript可以把对方法的引用作为参数传递，所以命令类只要

把这个引用保存下来，然后在execute方法的执行过程中调用它即可。这实际上就是一个函数的包装对象。

```
/* MenuCommand class, a command object. */

var MenuCommand = function(action) { // implements Command
    this.action = action;
};

MenuCommand.prototype.execute = function() {
    this.action();
};
```

如果action方法内部需要用到this关键字，那么它必须被包装在一个匿名函数中。如下所示<sup>①</sup>：

233 var someCommand = new MenuCommand(function() { myObj.someMethod(); });

### 16.3.3 汇合起来

这个复杂架构的最终结果中的实现代码很容易理解，其各个部分之间的耦合也比较松散。你需要做的就是创建MenuBar类的一个实例，然后为它添加一些Menu和MenuItem对象。其中每个MenuItem对象都绑定了一个命令对象：

```
/* Implementation code. */

/* Receiver objects, instantiated from existing classes. */
var fileActions = new FileActions();
var editActions = new EditActions();
var insertActions = new InsertActions();
var helpActions = new HelpActions();

/* Create the menu bar. */
var appMenuBar = newMenuBar();

/* The File menu. */
var fileMenu = new Menu('File');

var openCommand = new MenuCommand(fileActions.open);
var closeCommand = new MenuCommand(fileActions.close);
var saveCommand = new MenuCommand(fileActions.save);
var saveAsCommand = new MenuCommand(fileActions.saveAs);
```

<sup>①</sup> 作者的这句话非常含混。这个问题的实质是，作为参数传递给MenuCommand的构造函数的那个函数内部不应该使用this关键字。如果一个函数内部使用了this，那么这就意味着其设计者希望它被作为一个对象的方法调用，而不是作为一个普通函数调用。在本例中，如果用作MenuCommand构造函数的参数那个函数内部使用了this的话，那么在execute方法内部调用它时，这个this实际上指向的是所生成的那个MenuCommand实例，这基本上不可能是该函数设计者的初衷。如果想用MenuCommand包装一个名为someMethod的函数，但又希望调用其execute方法的结果是把someMethod函数作为另外某个对象的方法调用，那么提供给MenuCommand构造函数的参数就不能是someMethod，而应该是另外一个新函数，这个函数内部将把someMethod作为那个对象（该对象可以通过闭包机制而被引用到）的方法调用。——译者注

```

fileMenu.add(new MenuItem('Open', openCommand));
fileMenu.add(new MenuItem('Close', closeCommand));
fileMenu.add(new MenuItem('Save', saveCommand));
fileMenu.add(new MenuItem('Save As...', saveAsCommand));

appMenuBar.add(fileMenu);

/* The Edit menu. */
var editMenu = new Menu('Edit');

var cutCommand = new MenuCommand(editActions.cut);
var copyCommand = new MenuCommand(editActions.copy);
var pasteCommand = new MenuCommand(editActions.paste);
var deleteCommand = new MenuCommand(editActions.delete);

editMenu.add(new MenuItem('Cut', cutCommand));
editMenu.add(new MenuItem('Copy', copyCommand));
editMenu.add(new MenuItem('Paste', pasteCommand));
editMenu.add(new MenuItem('Delete', deleteCommand));

appMenuBar.add(editMenu);

/* The Insert menu. */
var insertMenu = new Menu('Insert');
var textBlockCommand = new MenuCommand(insertActions.textBlock);
insertMenu.add(new MenuItem('Text Block', textBlockCommand));

appMenuBar.add(insertMenu);

/* The Help menu. */
var helpMenu = new Menu('Help');

var showHelpCommand = new MenuCommand(helpActions.showHelp);
helpMenu.add(new MenuItem('Show Help', showHelpCommand));

appMenuBar.add(helpMenu);

/* Build the menu bar. */
document.getElementsByTagName('body')[0].appendChild(appMenuBar.getElement());
appMenuBar.show();

```

234

### 16.3.4 添加更多菜单项

要是以后想为菜单再增添一些菜单项，这很容易办到。例如，区区如下两行代码就能在Insert菜单中添加一个图像命令（假设InsertActions类已实现了需要的操作）：

```

var imageCommand = new MenuCommand(insertActions.image);
insertMenu.add(new MenuItem('Image', imageCommand));

```

这个菜单系统实现了接受用户请求的对象与实现相关操作的对象的隔离。命令模式非常适合

用来构建用户界面，这是因为这种模式可以把执行具体工作的类与生成用户界面的类隔离开来。在这种模式中，甚至可以让多个用户界面元素共用同一个接收者或命令对象。既然命令可以作为一等对象进行传递和重用，那么它自然应该能够反复执行——甚至是被不同的调用者反复执行。

## 16.4 示例：取消操作和命令日志

还有一个方法也经常被实现为命令对象，那就是undo。借助这个方法，调用者可以回滚用execute执行的操作。undo方法可以用来实现不受限制的取消功能。只需要把执行过的命令对象压入栈顶即可实现对命令执行历史的记录。如果用户想撤销最近的操作，他可以点击“取消”按钮，这会从栈中弹出最近那个命令并调用该命令的undo方法。用户可以这样取消先前执行过的所有操作，直到栈底。

为了演示用命令模式实现不受限制的取消操作的方法，我们下面要设计一个类似于Etch A Sketch的游戏<sup>①</sup>。游戏界面中有四个移动按钮，其功能分别是把指针向上、下、左、右4个方向移动10个像素。此外还有一个取消按钮，它可以用来撤销操作。首先我们必须修改一下Command接口，为其添加一个undo方法：

```
/* ReversibleCommand interface. */

var ReversibleCommand = new Interface('ReversibleCommand', ['execute', 'undo']);

接下来要做的是创建4个命令类，它们分别用来向上、下、左、右4个方向移动指针：

/* Movement commands. */

var MoveUp = function(cursor) { // implements ReversibleCommand
    this.cursor = cursor;
};

MoveUp.prototype = {
    execute: function() {
        cursor.move(0, -10);
    },
    undo: function() {
        cursor.move(0, 10);
    }
};

var MoveDown = function(cursor) { // implements ReversibleCommand
    this.cursor = cursor;
};

MoveDown.prototype = {
    execute: function() {

```

<sup>①</sup> 这是一种儿童玩具，国内有人称为“魔术画板”。经典的Etch A Sketch是一块中空的平板，内装一些金属粉末。其正面大部分覆盖着一块透明的有机玻璃薄板，在下边左右两角各有一个控制手柄。将正面朝下抖动平板，即可让金属粉末均匀地吸附在有机玻璃内表面。然后儿童可以通过操纵两个手柄控制平板内部的一个笔尖移动。笔尖移动路线上的金属粉末会被刮落，从而显出线条。——译者注

```
        cursor.move(0, 10);
    },
    undo: function() {
        cursor.move(0, -10);
    }
};

var MoveLeft = function(cursor) { // implements ReversibleCommand
    this.cursor = cursor;
};
MoveLeft.prototype = {
    execute: function() {
        cursor.move(-10, 0);
    },
    undo: function() {
        cursor.move(10, 0);
    }
};

var MoveRight = function(cursor) { // implements ReversibleCommand
    this.cursor = cursor;
};
MoveRight.prototype = {
    execute: function() {
        cursor.move(10, 0);
    },
    undo: function() {
        cursor.move(-10, 0);
    }
};
```

236

这些代码很简单。execute方法向合适的方向移动指针，而undo方法则向相向的方向把指针移回去。这是一个相当典型的带取消功能的命令。相关的操作必须易于逆转，而且不用知道系统原来的状态。

最后，我们还需要有用作调用者的按钮和实际负责实现指针移动的接收者。先来看看接收者：

```
/* Cursor class. */

var Cursor = function(width, height, parent) {
    this.width = width;
    this.height = height;
    this.position = { x: width / 2, y: height / 2 };

    this.canvas = document.createElement('canvas');
    this.canvas.width = this.width;
    this.canvas.height = this.height;
    parent.appendChild(this.canvas);

    this.ctx = this.canvas.getContext('2d');
    this.ctx.fillStyle = '#cc0000';
```

```

        this.move(0, 0);
    };
    Cursor.prototype.move = function(x, y) {
        this.position.x += x;
        this.position.y += y;

        this.ctx.clearRect(0, 0, this.width, this.height);
        this.ctx.fillRect(this.position.x, this.position.y, 3, 3);
    };
}

```

Cursor类实现了命令类所要求的操作。本例中涉及的操作只不过是在特定位置绘制一个方块。调用命令的是网页上的按钮。本例需要的按钮有两种：一种是用来调用execute方法的命令按钮，另一种是用来调用undo方法的取消按钮。

在讲按钮类之前，先来看看如何用装饰者模式进一步提高命令类的模块化程度。我们需要在系统中的某个地方加入一些用来把执行过的命令压入栈的代码。这些代码可以写在用户界面类（即按钮类）中，但这样一来就必须在每一个用户界面类中重复这些代码。如果想把这些命令用于快捷键的话，那就还要再次实现这种入栈代码。更好的办法是用一个实现了这些代码的装饰者来包装每一个命令。这样我们就可以把命令对象传递给任意的用户界面元素，不必担心它是否实现了入栈代码。

下面这个装饰者的作用就是在执行一个命令之前先将其压栈：

```

/* UndoDecorator class. */

var UndoDecorator = function(command, undoStack) { // implements ReversibleCommand
    this.command = command;
    this.undoStack = undoStack;
};

UndoDecorator.prototype = {
    execute: function() {
        this.undoStack.push(this.command);
        this.command.execute();
    },
    undo: function() {
        this.command.undo();
    }
};

```

这是装饰者模式的出色运用。藉此我们可以在保留原有接口的前提下为命令增添新的特性。在本例中这些装饰者对象可以与所有命令对象互换使用。

现在来看看用户界面类。这些类负责生成必要的HTML元素，并且为其注册click事件监听器，这些监听器要么调用execute方法要么调用undo方法：

```

/* CommandButton class. */

var CommandButton = function(label, command, parent) {
    Interface.ensureImplements(command, ReversibleCommand);
    this.element = document.createElement('button');

```

```
this.element.innerHTML = label;
parent.appendChild(this.element);

addEvent(this.element, 'click', function() {
    command.execute();
});

/*
 * UndoButton class.
 */

var UndoButton = function(label, parent, undoStack) {
    this.element = document.createElement('button');
    this.element.innerHTML = label;
    parent.appendChild(this.element);
    addEvent(this.element, 'click', function() {
        if(undoStack.length === 0) return;
        var lastCommand = undoStack.pop();
        lastCommand.undo();
    });
};
```

238

像UndoDecorator类一样，UndoButton类的构造函数也需要把命令栈作为参数传入。这个栈其实就是一个数组。调用经UndoDecorator对象装饰过的命令对象的execute方法时这个命令对象会被压入栈。为了执行取消操作，取消按钮会从命令栈中弹出最近的命令并调用其undo方法。这将逆转刚执行过的操作。

与使用命令模式的大多数示例一样，本例的实现代码也非常简单。只需要实例化Cursor和所有命令，创建使用这些命令的按钮，再创建一个空白栈即可：

```
/*
 * Implementation code.
 */

var body = document.getElementsByTagName('body')[0];
var cursor = new Cursor(400, 400, body);
var undoStack = [];

var upCommand = new UndoDecorator(new MoveUp(cursor), undoStack);
var downCommand = new UndoDecorator(new MoveDown(cursor), undoStack);
var leftCommand = new UndoDecorator(new MoveLeft(cursor), undoStack);
var rightCommand = new UndoDecorator(new MoveRight(cursor), undoStack);

var upButton = new CommandButton('Up', upCommand, body);
var downButton = new CommandButton('Down', downCommand, body);
var leftButton = new CommandButton('Left', leftCommand, body);
var rightButton = new CommandButton('Right', rightCommand, body);
var undoButton = new UndoButton('Undo', body, undoStack);
```

这段代码生成的用户界面包含一幅画布（canvas）<sup>①</sup>和5个按钮。点击4个命令按钮中的任何

<sup>①</sup> canvas是HTML 5草案中新增的一个标签，用于提供一些绘图功能。现在已有一些浏览器（不含IE）支持这个标签。——译者注

一个都会向相应方向移动指针。而点击取消按钮则会撤销最近一次移动。

### 16.4.1 使用命今日志实现不可逆操作的取消

前面讨论的取消操作针对的都是移动指针这类容易逆转的操作。对于那些本质上不可逆的操作，要想实现不受限制的取消就困难得多。例如，要是希望修改一下前面的示例，使它更像Etch A Sketch游戏，那就需要在指针后面留下一串尾迹。在画布上画线很容易，不过要取消这条线的绘制是不可能的。从一个点到另一个点的移动这种操作具有精确的对立操作，执行后者的结果看起来就像前者被逆转了一样。但是对于从A到B画一条线这种操作，从B到A再画一条线是无法逆转前一操作的，这只不过是在第一条线的上方又画了一条线而已。  
239

取消这种操作的唯一办法是清除状态，然后把之前执行过的操作（不含最近那个）依次重做一遍。这很容易办到，为此需要把所有执行过的命令记录在栈中。要想取消一个操作，需要做的就是从栈中弹出最近那个命令并弃之不用，然后清理画布并从头开始重新执行记录下来的所有命令。使用命今日志实现取消操作的系统不要求那些命令都是可逆命令。因此在本例中可以继续使用原来的Command接口。

本例在原先的例子基础上所做的修改很小。由于大多数代码都保持原封不动，所以我们只讨论那些需要修改的地方。第一个变化是我们删除了所有命令对象的undo方法，原因是命令对象代表的操作现在不再可逆。下面是其中的一个命令类，其undo方法已被删除：

```
/* Movement commands. */

var MoveUp = function(cursor) { // implements Command
    this.cursor = cursor;
};
MoveUp.prototype = {
    execute: function() {
        cursor.move(0, -10);
    }
};
```

接下来，最大的变化发生在Cursor类的代码中。原来用来记录命令的栈undoStack现在成了该类的内部属性，其名称也改成了commandStack。而UndoDecorator类和所有其他对undoStack的引用都被删除。新的Cursor类如下所示：

```
/* Cursor class, with an internal command stack. */

var Cursor = function(width, height, parent) {
    this.width = width;
    this.height = height;
    this.commandStack = [];

    this.canvas = document.createElement('canvas');
    this.canvas.width = this.width;
    this.canvas.height = this.height;
    parent.appendChild(this.canvas);
```

```
this.ctx = this.canvas.getContext('2d');
this.ctx.strokeStyle = '#cc0000';
this.move(0, 0);
};

Cursor.prototype = {
  move: function(x, y) {
    var that = this;
    this.commandStack.push(function() { that.lineTo(x, y); });
    this.executeCommands();
  },
  lineTo: function(x, y) {
    this.position.x += x;
    this.position.y += y;
    this.ctx.lineTo(this.position.x, this.position.y);
  },
  executeCommands: function() {
    this.position = { x: this.width / 2, y: this.height / 2 };
    this.ctx.clearRect(0, 0, this.width, this.height); // Clear the canvas.
    this.ctx.beginPath();
    this.ctx.moveTo(this.position.x, this.position.y);
    for(var i = 0, len = this.commandStack.length; i < len; i++) {
      this.commandStack[i]();
    }
    this.ctx.stroke();
  },
  undo: function() {
    this.commandStack.pop();
    this.executeCommands();
  }
};

```

240

这里新增了3个新方法。原有的move方法已被修改，它现在所做的就是把操作压入命令栈，然后调用executeCommands方法。实际执行操作并画线的是lineTo方法。executeCommands负责重置canvas元素，然后依次执行命令栈中保存的操作。undo方法则会删除栈中最近的那条命令，然后调用executeCommands以重建系统状态。

在Cursor类中似乎有些命名不当问题。那个commandStack包含的实际上并非命令对象，而是对函数的引用。而executeCommands方法实际上也并没有调用任何命令对象的execute方法，它调用的只不过是栈中保存的函数。那么我们需要回头改改这些扎眼的属性和方法名称吗？其实不必。命令也有许多不同类型，这里所用的命令没有封装在对象中并不意味着它们不能算命令。用闭包封装的方法调用也是完全有效的命令。真正重要的是它们的接口要一致，这样executeCommands方法不必知道它们做什么用也能调用它们。因为这里所用的命令只是一些函数引用，所以executeCommands只需要在其后面加上“()”就能调用它们。这再一致不过了。

其他的都是一些鸡毛蒜皮的修改。所有对undoStack的引用都被删除，UndoButton中相关元素的click事件监听器的代码也改了：

```
/* UndoButton class. */
```

241

```
var UndoButton = function(label, parent, cursor) {
    this.element = document.createElement('button');
    this.element.innerHTML = label;
    parent.appendChild(this.element);
    addEvent(this.element, 'click', function() {
        cursor.undo();
    });
};
```

实现代码与原来基本相同，唯一的变化是删除了undoStack，而传给UndoButton构造函数的参数也变成了一个Cursor实例：

```
/* Implementation code. */

var body = document.getElementsByTagName('body')[0];
var cursor = new Cursor(400, 400, body);

var upCommand = new MoveUp(cursor);
var downCommand = new MoveDown(cursor);
var leftCommand = new MoveLeft(cursor);
var rightCommand = new MoveRight(cursor);

var upButton = new CommandButton('Up', upCommand, body);
var downButton = new CommandButton('Down', downCommand, body);
var leftButton = new CommandButton('Left', leftCommand, body);
var rightButton = new CommandButton('Right', rightCommand, body);
var undoButton = new UndoButton('Undo', body, cursor);
```

现在我们有了一个带有不受限的取消功能的在线Etch A Sketch。因为按钮发出的命令都是模块化的，所以很容易增添一些新按钮，比如画圆的按钮或画笑脸符号 (:-)) 的按钮。由于现在不再要求操作必须可逆，因此我们可以实现一些更加复杂的行为。

#### 16.4.2 用于崩溃恢复的命今日志

命今日志的一个有趣用途是在程序崩溃后恢复其状态。在前面这个示例中，可以用XHR把经过序列化处理的命令记录到服务器上。用户下次访问该网页的时候，系统可以找出这些命令并用其将画布上的图案精确恢复到浏览器关闭时的状态。这可以替用户把应用程序状态保管下来，以便其撤销先前的任何一次浏览器会话中执行的操作。如果应用系统比较复杂，那么这种类型的命今日志会有很大的存储需求。为此你可以提供一个按钮，用户可以用它提交 (commit) 到当时为止的所有操作，从而清空命令栈。

### 16.5 命令模式的适用场合

命令模式的主要用途是把调用对象（用户界面、API和代理等）与实现操作的对象隔离开。照此而言，凡是两个对象间的互动方式需要有更高的模块化程度时都可以用到这种模式。这是一种组织型模式，几乎可以应用于任何系统。不过，最能体现其效用的还是那种需要对操作进行规范化处理

的场合。有了这种规范化处理，一个类或调用者也能调用多种方法，而且不需要先为此了解那些方法。许多用户界面元素都非常符合这样的特征，比如前面例子中的那种菜单。命令模式可以彻底消除用户界面元素与负责实际工作的类之间的耦合。这意味着这些元素可以被重用于任何网页或项目中，因为它们可以完全独立于那些类而使用。而那些负责实际工作的类也能被不同的用户界面元素使用。你可以为某个操作创建一个命令对象，然后用菜单项、工具图标和键盘快捷键调用这个对象。

可以受益于命令模式的还有其他一些特别场合。这种模式可以用来封装用于XHR调用或其他延迟性调用场合的回调函数。用一个回调函数命令代替回调函数，可以把多条函数调用封装为一个单位。有了命令对象的帮助，在应用程序中实现取消机制几乎是一件不足挂齿的事。实现不受限制的取消机制需要把执行过的命令保存在栈中。这种命令日志甚至可以用来取消本质上不可逆的操作。它还可以在任何应用程序崩溃之后用来恢复其整体状态。

## 16.6 命令模式之利

使用命令模式的主要好处有两个。首先，如果运用得当，它可以提高程序的模块化程度和灵活性。第二，有了它，实现取消和状态恢复等复杂的有用特性非常容易。

有人可能会争辩说：命令不过就是一个不必要地复杂化了的方法，在大多数情况下一个赤条条的方法也能取代它。这种观点仅对命令模式的那些无关痛痒的实现（trivial implementations）成立。命令对象具有的特性比普通的方法引用多得多。它可以被参数化处理，而且将那些参数保存起来以供多次调用。你可以为它定义的方法不只有execute，还可以是undo等别的方法，这样一来同样的操作就可以用不同的方式执行。还可以为其定义与操作相关的元数据，这些元数据可用于对象内省（introspection）或事件日志等目的。命令对象是经过封装的方法调用，它因为这种封装而拥有了方法调用本身所不具备的许多特性。

## 16.7 命令模式之弊

与其他任何模式一样，如果用法有误或者用得勉强的话，命令模式也会对程序造成损害。如果一个命令对象只包装了一个方法调用，而且其唯一目的就是这层对象包装的话，那么这种做法是一种浪费。如果你不需要命令模式给予的任何额外特性，也不需要具有一致接口的类所带来的模块性，那么直接使用方法引用而不是完整的命令对象也许更恰当。命令对象也会增加代码调试的难度，这是因为在应用了命令模式之后原有的方法之上又多了一层可能出错的代码。如果命令对象是运行期间动态创建的而你又难以确定它包含着什么操作的话，情况尤其如此。命令对象都具有同样的接口并且可以随意更换这一点是把双刃剑。在调试复杂的应用程序时它们很难跟踪。

## 16.8 小结

本章研究了命令模式。这是一种用来封装单个操作（discrete action）的结构型模式<sup>①</sup>。其封

<sup>①</sup> 本书的作者在这方面不太严谨。他在16.5节称命令模式为“组织型模式”，但在这里又称其为“结构型模式”。GoF的《设计模式》一书中把命令模式归入行为模式一类。——译者注

装的操作可能是单个方法调用这么简单，也可能是整个子程序那么复杂。经封装的操作可以作为一等的对象进行传送。命令对象主要用于消除调用者与接收者之间的耦合，这有助于创建高度模块化的调用者，它们对所调用的操作不需要任何了解。这种模式也给了程序员实现接收者的自由，他们不必担心接收者能否用在某套用户界面中。有些复杂的用户特性用命令模式很容易实现，不受限制的取消和程序崩溃之后的状态恢复就是这方面的两个例子。它们还可以用来实现事务，具体做法是把命令保存在栈中，隔一段时间提交一次。

命令模式最大的优点在于，只要是能够在execute方法中实现的操作，不管它们有多复杂或者彼此的差异有多大，都能以与任何别的命令完全相同的方式进行传送和调用。借助这种模式，  
244 代码的重用几乎可以达到一种不受限制的程度，这可以大大节省程序员的时间和精力。

命令模式的实现非常简单，只要实现一个命令接口，再实现一个具体的命令类即可。命令接口中包含一个execute方法，该方法将执行命令的具体操作。命令类继承自命令接口，并实现execute方法。这样，命令类就可以执行具体的命令操作了。命令类通常还包含一个接收者对象，以便在execute方法中调用接收者的操作。命令类也可以包含一些状态信息，以便在execute方法中使用。命令类通常还包含一个撤销方法，以便在命令被撤销时能够恢复原来的状态。

命令模式的一个常见应用是在文本编辑器中。在文本编辑器中，命令通常表示为“插入文本”、“删除文本”、“复制文本”、“粘贴文本”等操作。这些操作都可以抽象为命令对象，然后通过命令队列来管理。当用户执行某个操作时，会生成一个命令对象并将其加入命令队列。当用户想要撤销操作时，可以从命令队列中取出最后一个命令对象并调用其撤销方法。这样，命令模式就实现了文本编辑器中的撤销功能。

命令模式的一个常见应用是在文本编辑器中。在文本编辑器中，命令通常表示为“插入文本”、“删除文本”、“复制文本”、“粘贴文本”等操作。

# 职责链模式

**本**章考察的是职责链模式，它可以用来消除请求的发送者和接收者之间的耦合。这是通过实现一个由隐式地对请求进行处理的对象组成的链而做到的。链中的每个对象可以处理请求，也可以将其传给下一个对象。JavaScript内部就使用了这种模式来处理事件捕获和冒泡的问题。我们接下来将研究如何用这种模式创建耦合更松散的模块和优化事件绑定。

## 17.1 职责链的结构

职责链由多个不同类型的对象组成。发送者(sender)是发出请求的对象，而接收者(receiver)则是链中那些接收这种请求并且对其进行处理或传递的对象。请求本身有时也是一个对象，它封装着与操作有关的所有数据。其典型的运转流程大致是：

- 发送者知道链中的第一个接收者。它向这个接收者发出请求。
- 每一个接收者都对请求进行分析，然后要么处理它，要么将其往下传。
- 每一个接收者知道的其他对象只有一个，即它在链中的下家(successor)。
- 如果没有任何接收者处理请求，那么请求将从链上离开。不同的实现对此有不同的反应，既可能无声无息，也可能抛出一个错误。

为了说明职责链模式的组织方式及其作用，我们先回顾一下第14章所讲的那个图书馆的示例。例中的PublicLibrary类保存着一个按图书的ISBN索引的图书目录。如果知道书的ISBN，那么查书很容易。但是按书的主题或类别查书却很困难。下面我们要实现几种目录对象，以便按不同标准对图书进行分类。

245

先来看看要用的接口：

```
/* Interfaces. */
```

```
var Publication = new Interface('Publication', ['getIsbn', 'setIsbn', 'getTitle',
    'setTitle', 'getAuthor', 'setAuthor', 'getGenres', 'setGenres', 'display']);
var Library = new Interface('Library', ['addBook', 'findBooks', 'checkoutBook',
    'returnBook']);
var Catalog = new Interface('Catalog', ['handleFilingRequest', 'findBooks',
    'setSuccessor']);
```

与以前相比，Publication接口只是多了两个新方法：getGenres和setGenres。Library接口新增了一个为图书馆增添藏书的方法，而用于查书、借书和还书那三个方法是原有的。Catalog接

口是新增的。它将用来创建保存图书对象的类。本例中将对图书分类编目。Catalog接口有三个方法：handleFilingRequest会根据传给它的图书是否符合特定标准而决定是否将其编入内部目录中；findBooks会根据某些参数对内部目录进行搜索；setSuccessor则会设定职责链中的下一环。

现在来看一下Book和PublicLibrary这两个将被重用的类。它们都需要略做修改，以便实现分类编目：

```
/* Book class. */

var Book = function(isbn, title, author, genres) { // implements Publication
    ...
}
```

Book类现在多了一个用来说明图书所属类别的参数。它也实现了getGenres和setGenres方法，不过这里省略了其具体实现，因为它们就是一对简单的取值器和赋值器方法。

```
/* PublicLibrary class. */

var PublicLibrary = function(books) { // implements Library
    this.catalog = {};
    for(var i = 0, len = books.length; i < len; i++) {
        this.addBook(books[i]);
    }
};

PublicLibrary.prototype = {
    findBooks: function(searchString) {
        var results = [];
        for(var isbn in this.catalog) {
            if(!this.catalog.hasOwnProperty(isbn)) continue;
            if(this.catalog[isbn].getTitle().match(searchString) ||
                this.catalog[isbn].getAuthor().match(searchString)) {
                results.push(this.catalog[isbn]);
            }
        }
        return results;
    },
    checkoutBook: function(book) {
        var isbn = book.getIsbn();
        if(this.catalog[isbn]) {
            if(this.catalog[isbn].available) {
                this.catalog[isbn].available = false;
                return this.catalog[isbn];
            }
            else {
                throw new Error('PublicLibrary: book ' + book.getTitle() +
                    ' is not currently available.');
            }
        }
        else {
            throw new Error('PublicLibrary: book ' + book.getTitle() + ' not found.');
        }
    }
};
```

```

        }
    },
    returnBook: function(book) {
        var isbn = book.getIsbn();
        if(this.catalog[isbn]) {
            this.catalog[isbn].available = true;
        }
        else {
            throw new Error('PublicLibrary: book ' + book.getTitle() + ' not found.');
        }
    },
    addBook: function(newBook) {
        this.catalog[newBook.getIsbn()] = { book: newBook, available: true };
    }
};

```

除了用于添加图书的代码被移到了addBook这个新方法中以外，PublicLibrary目前还没有其他改变。本例中稍后还会对addBook和findBooks方法进行修改。

安顿好原有类之后，现在该来实现目录对象了。在为这些对象编写代码之前，先来设想一下其用法。所有用于判断一本书是否应该编入某个特定目录的代码都封装在Catalog类中。这意味着需要在PublicLibrary对象中把每一本书都提供给每一种分类目录进行处理：

```

/* PublicLibrary class, with hard-coded catalogs for genre. */

var PublicLibrary = function(books) { // implements Library
    this.catalog = {};
    this.biographyCatalog = new BiographyCatalog();
    this.fantasyCatalog = new FantasyCatalog();
    this.mysteryCatalog = new MysteryCatalog();
    this.nonFictionCatalog = new NonFictionCatalog();
    this.sciFiCatalog = new SciFiCatalog();

    for(var i = 0, len = books.length; i < len; i++) {
        this.addBook(books[i]);
    }
};

PublicLibrary.prototype = {
    findBooks: function(searchString) { ... },
    checkoutBook: function(book) { ... },
    returnBook: function(book) { ... },
    addBook: function(newBook) {
        // Always add the book to the main catalog.
        this.catalog[newBook.getIsbn()] = { book: newBook, available: true };

        // Try to add the book to each genre catalog.
        this.biographyCatalog.handleFilingRequest(newBook);
        this.fantasyCatalog.handleFilingRequest(newBook);
        this.mysteryCatalog.handleFilingRequest(newBook);
        this.nonFictionCatalog.handleFilingRequest(newBook);
    }
};

```

```

        this.sciFiCatalog.handleFilingRequest(newBook);
    }
};


```

前面这段代码可以奏效，不过其中固化了对5个不同类的依赖。如果想增加更多图书类别，那就需要修改构造函数和addBook方法这两处的代码。此外，把这些目录类别固化在构造函数中也没有多大意义，因为PublicLibrary的不同实例可能希望拥有完全不同的一套分类目录。而你也不可能在对象实例化之后再修改其支持的类别。这些理由都充分说明了前面的方法并不可取。下面来看职责链模式能为它带来什么改进：

```

/* PublicLibrary class, with genre catalogs in a chain of responsibility. */

var PublicLibrary = function(books, firstGenreCatalog) { // implements Library
    this.catalog = {};
    this.firstGenreCatalog = firstGenreCatalog;

    for(var i = 0, len = books.length; i < len; i++) {
        this.addBook(books[i]);
    }
};

PublicLibrary.prototype = {
    findBooks: function(searchString) { ... },
    checkoutBook: function(book) { ... },
    returnBook: function(book) { ... },
    addBook: function(newBook) {
        // Always add the book to the main catalog.
        this.catalog[newBook.getIsbn()] = { book: newBook, available: true };
        // Try to add the book to each genre catalog.
        this.firstGenreCatalog.handleFilingRequest(newBook);
    }
};

```

248

这个改进很明显。现在需要保存的只是指向分类目录链中第一个环节的引用。如果想把一本新书编入各种分类目录中，只需要将其传给链中第一个目录对象即可。第一个目录要么把它编入自己的目录（如果它符合所需标准），要么不编，然后，该目录会将该请求传递给下一个目录。因为一本图书可以属于不止一个类别，所以每个目录都会把请求往下传递。

现在不再有固化在代码中的依赖。所有分类目录都在外部实例化，因此不同的PublicLibrary实例能够使用不同的分类。你随时都可以在链中加入新的目录。下面的例子显示了其用法：

```

// Instantiate the catalogs.
var biographyCatalog = new BiographyCatalog();
var fantasyCatalog = new FantasyCatalog();
var mysteryCatalog = new MysteryCatalog();
var nonFictionCatalog = new NonFictionCatalog();
var sciFiCatalog = new SciFiCatalog();

// Set the links in the chain.
biographyCatalog.setSuccessor(fantasyCatalog);

```

```
fantasyCatalog.setSuccessor(mysteryCatalog);
mysteryCatalog.setSuccessor(nonFictionCatalog);
nonFictionCatalog.setSuccessor(sciFiCatalog);

// Give the first link in the chain as an argument to the constructor.
var myLibrary = new PublicLibrary(books, biographyCatalog);

// You can add links to the chain whenever you like.
var historyCatalog = new HistoryCatalog();
sciFiCatalog.setSuccessor(historyCatalog);
```

这个例子中，原来的链上有5个环节，第6个环节是后来加的。这意味着图书馆中每增加一本书都会通过调用链上第一个环节的handleFilingRequest方法发起对该书的编目请求。该请求将沿目录链逐一经过6个目录，最后从链尾离开。链上新增的任何目录都会被挂到链尾。

前面我们已经考察了使用职责链模式的动机以及与其使用相关的一般结构，但还没有研究过链上的对象本身。这些对象都具有一些共同特征。它们都拥有一个指向链上下一个对象（被称为下家（successor））的引用。对于链上最后一个对象，这是一个空引用。链上的对象至少都要实现一个共同的方法，即负责处理请求的方法。这些对象不用像前面的例子中那样属于同一个类，但是它们必须实现同样的接口。通常它们分别属于一个类（它实现了所有方法的默认版本）的各种子类。分类目录对象就是这样实现的：

```
/* GenreCatalog class, used as a superclass for specific catalog classes. */

var GenreCatalog = function() { // implements Catalog
    this.successor = null;
    this.catalog = [];
};

GenreCatalog.prototype = {
    _bookMatchesCriteria: function(book) {
        return false; // Default implementation; this method will be overriden in
                      // the subclasses.
    },
    handleFilingRequest: function(book) {
        // Check to see if the book belongs in this category.
        if(this._bookMatchesCriteria(book)) {
            this.catalog.push(book);
        }
        // Pass the request on to the next link.
        if(this.successor) {
            this.successor.handleFilingRequest(book);
        }
    },
    findBooks: function(request) {
        if(this.successor) {
            return this.successor.findBooks(request);
        }
    },
    setSuccessor: function(successor) {
```

```

    if(Interface.ensureImplements(successor, Catalog) {
        this.successor = successor;
    }
}
};

```

这个超类提供了所有必需方法的默认实现，它们可以被各种子类继承。子类只需要重写findBooks（参见下一节）和\_bookMatchesCriteria这两个方法。其中后一个方法是一个伪私用方法，它负责判断一本书是否应被编入相关分类目录。GenreCatalog类提供了这两个方法的最简实现，以防子类没有重写它们。

从这个超类派生一个分类目录子类很简单：

```

/* SciFiCatalog class. */

var SciFiCatalog = function() {} // implements Catalog
extend(SciFiCatalog, GenreCatalog);
SciFiCatalog.prototype._bookMatchesCriteria = function(book) {
    var genres = book.getGenres();
    if(book.getTitle().match(/space/i)) {
        return true;
    }
    for(var i = 0, len = genres.length; i < len; i++) {
        var genre = genres[i].toLowerCase();
        if(genres === 'sci-fi' || genres === 'scifi' || genres === 'science fiction') {
            return true;
        }
    }
    return false;
};

```

250

这段代码先创建了一个空构造函数，让其继承GenreCatalog，然后实现了\_bookMatchesCriteria方法。它的这个方法对图书的书名和类别进行检查，判断是否二者中有一个能够匹配某些搜索用词（search term）。这只是一个非常简单的实现。更健壮的方案需要检查更多搜索用词。

## 17.2 传递请求

在链上传递请求有许多不同方法可供选择。最常见的做法要么是使用一个专门的请求对象，要么是根本不使用参数，只依靠方法自身传递消息。不用参数调用方法是最简单的办法。我们将在17.6节的实用示例中考察这种技术。在前面的例子中，我们使用了另一种常见技术，即把图书对象作为请求进行传递。图书对象封装了在判断链上哪些环节应该将其编入它们的目录时需要的所有数据。这属于将现有对象作为请求对象进行重用的情况。而在本节中，我们将实现分类目录的findBooks方法，并将考察如何使用专门的请求对象来在链上的各个环节之间传递数据。

首先我们需要修改一下PublicLibrary的findBooks方法，以便可以根据类别来缩小搜索范围。如果调用该方法时提供了可选的genres参数，那么搜索将只在属于其指定类别的图书中进行：

```

/* PublicLibrary class. */

var PublicLibrary = function(books) { // implements Library
    ...
};

PublicLibrary.prototype = {
    findBooks: function(searchString, genres) {
        // If the optional genres argument is given, search for books only in
        // those genres. Use the chain of responsibility to perform the search.
        if(typeof genres === 'object' && genres.length > 0) {
            var requestObject = {
                searchString: searchString,
                genres: genres,
                results: []
            };
            var responseObject = this.firstGenreCatalog.findBooks(requestObject);
            return responseObject.results;
        }
        // Otherwise, search through all books.
        else {
            var results = [];
            for(var isbn in this.catalog) {
                if(!this.catalog.hasOwnProperty(isbn)) continue;
                if(this.catalog[isbn].getTitle().match(searchString) ||
                   this.catalog[isbn].getAuthor().match(searchString)) {
                    results.push(this.catalog[isbn]);
                }
            }
            return results;
        }
    },
    checkoutBook: function(book) { ... },
    returnBook: function(book) { ... },
    addBook: function(newBook) { ... }
};

```

251

`findBooks`方法创建了一个用来封装与请求相关的所有信息的对象，这些信息包括将要搜索的一组类别、搜索用词和一个用来保存查找结果的空数组。这里有一个明显的问题：这些信息作为独立的参数进行传递也很容易，那为什么还要费心创建这样一个对象呢？创建这个对象的主要原因是，把所有数据集中在一起后管理起来要方便得多。这些信息需要在通过链上各个环节的过程中保持完好无缺，把它们封装在一个对象中更利于做到这一点。在本例中，这个对象也会作为返回值返回给客户代码。如果要一次在链上发起多个搜索，那么这样的设计也有助于把搜索结果与所搜索的用词和类别保存在一起。

现在我们要实现`GenreCatalog`这个超类中的`findBooks`方法。这个方法将被用在所有子类中，它不需要被重写。其代码有点复杂，我们会逐步进行详细说明：

```
/* GenreCatalog class, used as a superclass for specific catalog classes. */
```

```

var GenreCatalog = function() { // implements Catalog
    this.successor = null;
    this.catalog = [];
    this.genreNames = [];
};

GenreCatalog.prototype = {
    _bookMatchesCriteria: function(book) { ... },
    handleFilingRequest: function(book) { ... },
    findBooks: function(request) {
        var found = false;
        for(var i = 0, len = request.genres.length; i < len && !found; i++) {
            for(var j = 0, nameLen = this.genreNames.length; j < nameLen; j++) {
                if(this.genreNames[j] === request.genres[i]) {
                    found = true; // This link in the chain should handle
                                  // the request.
                    break;
                }
            }
        }

        if(found) { // Search through this catalog for books that match the search
                    // string and aren't already in the results.
        outerloop: for(var i = 0, len = this.catalog.length; i < len; i++) {
            var book = this.catalog[i];
            if(book.getTitle().match(request.searchString) ||
               book.getAuthor().match(request.searchString)) {
                for(var j = 0, requestLen = request.results.length; j < requestLen; j++) {
                    if(request.results[j].getIsbn() === book.getIsbn()) {
                        continue outerloop; // The book is already in the results; skip it.
                    }
                }
                request.results.push(book); // The book matches and doesn't already
                                            // appear in the results. Add it.
            }
        }
    }

    // Continue to pass the request down the chain if the successor is set.
    if(this.successor) {
        return this.successor.findBooks(request);
    }
    // Otherwise, we have reached the end of the chain. Return the request
    // object back up the chain.
    else {
        return request;
    }
},
setSuccessor: function(successor) { ... }
};

```

这个方法可以分为三大部分。第一部分逐一检查请求对象中的每一个类别名称，看看其是否与对象中保存的一组类别名称中的某一个匹配。如果匹配，那么代码的第二部分会逐一检查目录中的所有图书，看看其书名和作者姓名是否与搜索用词匹配。与搜索用词匹配的图书将被添加到请求对象中的results数组中，不过其前提是该数组中还没有这本书。在最后一部分，如果当前目录对象不是链上的最后一环，那么请求将被沿目录链继续下传，否则它将返回请求对象。最终请求对象将从链尾开始沿目录链逐环向上返回，直到返回给客户代码。

在超类GenreCatalog中，用于保存类别名称的属性genreNames是一个空数组，在子类中必须为其填入一些具体的类别名称。下面是SciFiCatalog类的实现代码：

```
/* SciFiCatalog class. */

var SciFiCatalog = function() { // implements Catalog
    this.genreNames = ['sci-fi', 'scifi', 'science fiction'];
};

extend(SciFiCatalog, GenreCatalog);
SciFiCatalog.prototype._bookMatchesCriteria = function(book) { ... };
```

通过把请求封装为一个对象，可以使它更容易管理。在GenreCatalog类的findBooks方法这种复杂代码中尤其如此。它有助于让搜索用词、类别和搜索结果在通过链上所有必须经过的环节的过程中保持完好无缺。

## 17.3 在现有层次体系中实现职责链

前面例子中的职责链是从头开始创建出来的，但是在在一个现有的对象层次体系中实现这种模式往往更容易。在此情况下它经常与组合模式搭配使用。由于组合模式已经建立了一个对象层次体系，因此在此基础上添加一些用来处理（或传递）请求的方法很简单。

不过要注意，在这种结合中方法的工作机制相对于组合模式中的一般机制有所偏离。在组合模式中，组合对象与叶对象实现了同样的接口。对组合对象进行的任何方法调用都会被传递到所有子对象——包括叶对象和那些本身也是组合对象的子对象。方法调用到达叶对象之后，这些对象就会实际执行操作并完成工作。

但在组合模式结合了职责链模式之后，方法调用就不再总是不加分辨地往下一直传到叶对象。此时每一层都要对请求进行分析，以判断当前对象应该处理它还是应该把它往下传。组合对象实际上也会承担部分工作，而不是单纯依靠叶对象执行所有操作。

这两种模式的结合看似让代码增加了一些复杂性，其实通过重用现有的层次体系来实现职责链有很多好处。这样一来，就不用单独实现一些对象来作链上的环节，也不用手工设定下家对象。这些事早已安排妥当。此外，通过规定在某些场合下一些方法调用可以在层次体系中较高的层次上得到处理，并且阻止较低层次的节点和叶节点获悉这些调用，可以让组合模式变得更加健壮。

对于较深的层次体系这种作用尤其明显。假设有个组合对象层次体系有5个层次，其中每个组合对象都有5个子对象。那么叶对象总共有625个，而所有对象加起来一共是781个。通常情况下，所有方法调用都要传递到每一个对象，这意味着该操作将经过156个组合对象，最终由625

个叶节点执行。要是该方法能在第二层进行处理，那么它只需要经过一个对象，然后由5个对象执行。这样带来的节省多达两个数量级。

254 职责链模式和组合模式的结合对双方都是一种优化。由于职责链是现成的，所以设置代码的数量和用于职责链的额外对象的数目都减少了。由于在组合层次体系中某个方法可能会在高层得到处理，所以在整个树上执行该方法所需的计算量也降低了。本章后面的实用示例就结合使用了这两种模式，其目的在于让HTML元素层次体系上的方法调用更有效率。

## 17.4 事件委托

JavaScript语言使用了职责链模式来决定如何处理事件。事件被触发时（以点击事件为例）要经历两个阶段。第一个阶段是事件捕获（event capturing）阶段。在此期间，事件会沿HTML层次体系向下传播，从最顶层开始，历经各个子元素，直到到达被点击的元素。从这时起将开始第二个阶段，即事件冒泡（event bubbling）。在这个阶段事件会历经同一批元素升回到顶层祖先。绑定在事件经过的这些元素上的事件监听器既可以停止事件传播，也可以让其继续沿层次体系向上或向下传播。这里传递的请求对象称为事件对象（event object），它包含着与事件有关的所有信息，比如事件名称和最初激发事件的元素。

因为事件模型本质上是作为一个职责链实现的，所以你在前面学到的有关这个模式的一些经验也可以应用于事件处理方面。其中一条就是，最好在层次体系的较高层次上处理请求。假设有一个无序列表包含着几打列表项。与其为其中的每一个li元素绑定一个click事件监听器，不如只为ul元素绑定一个这种事件监听器。尽管两种做法都能实现同样的目标，而且访问到的都是完全相同的事件对象，但是如果采用只为ul元素绑定一个事件监听器这种做法的话，脚本会运行得更快，内存消耗得更少，而且以后维护起来也更容易。这种技术称为事件委托（event delegation），这是职责链方面的知识有助于优化代码的情况之一。

## 17.5 职责链模式的适用场合

适用职责链模式的场合有几种。在图书馆的例子中，我们想发出对某本图书进行分类的请求。我们事先不知道如果可以的话它应该被分类到哪一个目录，也不知道可用目录的数目和类型。为了解决这些问题，我们使用了一个目录链，其中的每一个目录都会把图书对象沿链传递给其下家。

255 这个例子说明了可以受益于职责链模式的场合。如果事先不知道在几个对象中有哪些能够处理请求，那么这就属于应该使用职责链的情况。如果这批处理器对象在开发期间不可知，而是需要动态指定的话，那么也应该使用这种模式。该模式还可以用在对于每个请求都不止有一个对象可以对它进行处理这种情况下。例如，在图书馆的例子中，每本图书都可以被分类到不止一个目录。该请求可以先被一个对象处理，然后继续往下传，在链上可能后面还有另一个对象会处理它。

使用这种模式，可以把特定的具体类与客户隔离开，并代之以一条由弱耦合的对象组成的链，它将隐式地对请求进行处理。这有助于提高代码的模块化程度和可维护性。

## 17.6 图片库的进一步讨论

我们在第9章中曾通过设计一个层次化的图片库来说明组合模式的用法。在此我们将继续讨论这个例子，看看职责链模式如何进一步提高其效率并为其增加各种特性。首先，我们将使用组合对象层次体系来重新实现hide和show方法，在此过程中会结合使用职责链。接下来，我们将显示如何把图片动态加入该层次体系中的任意层次，其具体做法是从顶层开始将请求逐级下传。

这个图片库仅由两个类组成：DynamicGallery是组合对象类，而GalleryImage是叶类。下面是这些类原来在第9章中的实现：

```
/* Interfaces. */

var Composite = new Interface('Composite', ['add', 'remove', 'getChild']);
var GalleryItem = new Interface('GalleryItem', ['hide', 'show']);

/* DynamicGallery class. */

var DynamicGallery = function(id) { // implements Composite, GalleryItem
    this.children = [];
    this.element = document.createElement('div');
    this.element.id = id;
    this.element.className = 'dynamic-gallery';
}
DynamicGallery.prototype = {
    add: function(child) {
        Interface.ensureImplements(child, Composite, GalleryItem);
        this.children.push(child);
        this.element.appendChild(child.getElement());
    },
    remove: function(child) {
        for(var node, i = 0; node = this.getChildAt(i); i++) {
            if(node == child) {
                this.children.splice(i, 1);
                break;
            }
        }
        this.element.removeChild(child.getElement());
    },
    getChild: function(i) {
        return this.children[i];
    },
    hide: function() {
        for(var node, i = 0; node = this.getChildAt(i); i++) {
            node.hide();
        }
        this.element.style.display = 'none';
    },
    show: function() {
```

```

this.element.style.display = '';
for(var node, i = 0; node = this.getChildAt(i); i++) {
    node.show();
}
};

getElement: function() {
    return this.element;
}
};

/* GalleryImage class. */

var GalleryImage = function(src) { // implements Composite, GalleryItem
    this.element = document.createElement('img');
    this.element.className = 'gallery-image';
    this.element.src = src;
}
GalleryImage.prototype = {
    add: function() {},           // This is a leaf node, so we don't
    remove: function() {},        // implement these methods, we just
    getChild: function() {},     // define them.

    hide: function() {
        this.element.style.display = 'none';
    },
    show: function() {
        this.element.style.display = '';
    },
    getElement: function() {
        return this.element;
    }
};

```

### 17.6.1 用职责链提高组合对象的效率

在组合对象中，`hide`和`show`方法先对本层次的一个样式属性进行设置，然后将调用传递给所有子对象。这是一种缜密的做法，但是效率不高。由于元素的`display`样式属性会被其所有子元素继承，因此没有必要把方法调用向层次体系的下层继续传递。更好的做法是将这些方法作为沿职责链传递的请求实现。

这样做需要知道什么时候应该停止请求以及什么时候应该将它传递给子节点。这正是职责链模式的核心：懂得什么时候处理请求以及什么时候传递请求。每个组合对象节点和叶节点都有两种状态：显示和隐藏。`hide`请求根本不必传递，因为用CSS隐藏组合节点将自动隐藏其所有子节点。`show`请求则总是需要传递，因为无法预先得知组合对象节点的所有子节点的状态。我们做的第一个优化是从`hide`方法中删除将方法调用传递给子节点的那部分代码：

```

/* DynamicGallery class. */

var DynamicGallery = function(id) { // implements Composite, GalleryItem
    ...
}

DynamicGallery.prototype = {
    add: function(child) { ... },
    remove: function(child) { ... },
    getChild: function(i) { ... },
    hide: function() {
        this.element.style.display = 'none';
    },
    show: function() { ... },
    getElement: function() { ... }
};

```

现在该组合对象层次体系中的任何一段都可以被视为一条职责链。在传递hide或show请求的时候，你并不知道或关心具体将由哪些对象执行那些实际实现隐藏或显示的操作，因为请求处理是隐式的。

## 17.6.2 为图片添加标签

前面的例子是一个用来显示如何用职责链优化组合对象的极其简单的案例。我们将通过为图片添加标签进一步阐明这个概念。标签是一个描述性的标题，可以用来对图片进行分类。图片和图片库都可以添加标签。为图片库添加标签实际上相当于让其中的所有图片都使用这个标签。你可以在层次体系的任何层次上搜索具有指定标签的所有图像。这正是职责链的优化可资利用的地方。如果在搜索过程中遇到一个具有所请求的标签的组合对象节点，那就可以停止请求并将该节点的所有叶级后代节点作为搜索结果返回：

```

var Composite = new Interface('Composite', ['add', 'remove', 'getChild',
    'getAllLeaves']);
var GalleryItem = new Interface('GalleryItem', ['hide', 'show', 'addTag',
    'getPhotosWithTag']);

```

我们在接口中添加了三个方法。其中addTag将为用以调用它的对象及其所有子对象添加一个标签。getPhotosWithTag将返回一个由具有特定标签的所有图片组成的数组。对任何组合对象调用getAllLeaves将返回其所有叶级后代节点组成的数组，而对叶节点调用这个方法则返回一个由它自身组成的数组。因为addTag最简单，所以我们先从它开始讲起：

```

/* DynamicGallery class. */

var DynamicGallery = function(id) { // implements Composite, GalleryItem
    this.children = [];
    this.tags = [];
    this.element = document.createElement('div');
    this.element.id = id;
    this.element.className = 'dynamic-gallery';
}

```

```

DynamicGallery.prototype = {
  ...
  addTag: function(tag) {
    this.tags.push(tag);
    for(var node, i = 0; node = this.getChildAt(i); i++) {
      node.addTag(tag);
    }
  },
  ...
};

/* GalleryImage class. */

var GalleryImage = function(src) { // implements Composite, GalleryItem
  this.element = document.createElement('img');
  this.element.className = 'gallery-image';
  this.element.src = src;
  this.tags = [];
}
GalleryImage.prototype = {
  ...
  addTag: function(tag) {
    this.tags.push(tag);
  },
  ...
};

```

我们在组合对象类和叶类中都添加了一个名为tags的数组属性。它保存着代表标签的字符串。在叶类的addTag方法中只需要把作为参数传入的字符串加入tags数组中即可。而在组合对象类的这个方法中，除了这样做之外，还要像任何其他普通组合对象方法那样把请求在层次体系中向下传递。尽管把标签给予一个组合对象实际上相当于把该标签给予其所有子对象，但我们还是必须为每一个子对象添加该标签。这是因为搜索可能会从层次体系中的任何层次开始，如果没有为每一个叶节点添加标签的话，那么从较低层次开始的搜索可能会错过在层次体系中的较高层次上分配的标签。

getPhotosWithTag方法是职责链发挥优化作用的地方。我们将分别讲述每个类中的这个方法。首先来看组合对象类：

```

/* DynamicGallery class. */

var DynamicGallery = function(id) { // implements Composite, GalleryItem
  ...
};

DynamicGallery.prototype = {
  ...
  getAllLeaves: function() {
    var leaves = [];
    for(var node, i = 0; node = this.getChildAt(i); i++) {
      leaves = leaves.concat(node.getAllLeaves());
    }
  }
};

```

```

    }
    return leaves;
},
getPhotosWithTag: function(tag) {
    // First search in this object's tags; if the tag is found here, we can stop
    // the search and just return all the leaf nodes.
    for(var i = 0, len = this.tags.length; i < len; i++) {
        if(this.tags[i] === tag) {
            return this.getAllLeaves();
        }
    }
}

// If the tag isn't found in this object's tags, pass the request down
// the hierarchy.
for(var results = [], node, i = 0; node = this.getChildAt(i); i++) {
    results = results.concat(node.getPhotosWithTag(tag));
}
return results;
},
...
};

```

这段代码实际上为DynamicGallery添加了两个方法，不过它们的关系很密切，你稍后便可看到。getPhotosWithTag方法是按职责链的风格实现的。它首先要判断当前对象是否能处理请求。其具体做法是在当前对象的tags数组中检查指定的标签。如果能找到，那就表明层次体系中当前这个组合对象的所有子对象也都具有这个标签。此时即可停止搜索，然后在这个层次上处理请求。如果找不到指定标签，则将请求传递给每一个子对象，并返回结果。

260

getAllLeaves方法用于获取特定组合对象的所有叶级后代节点并将它们组织为一个数组返回。它是作为一个普通的组合对象方法实现的，也就是说同样的方法调用会被传递到每一个子对象。

这些方法在叶类中的实现相当简单。它们返回的结果数组都只包含叶对象自身（但getPhotosWithTag方法在当前叶对象的标签不匹配指定标签时返回一个空数组）：

```

/* GalleryImage class. */

var GalleryImage = function(src) { // implements Composite, GalleryItem
    ...
};

GalleryImage.prototype = {
    ...
    getAllLeaves: function() { // Just return this.
        return [this];
    },
    getPhotosWithTag: function(tag) {
        for(var i = 0, len = this.tags.length; i < len; i++) {
            if(this.tags[i] === tag) {
                return [this];
            }
        }
    }
};

```

```

    }
}

return []; // Return an empty array if no matches were found.
},
...
);

```

我们来分析一下在本例中使用职责链有什么收获以及组合模式是如何为此提供帮助的。如果把getPhotosWithTag作为一个组合对象方法实现，让它只负责把方法调用传递给每个子对象，那么每个子对象又得把自己的所有标签逐一与所搜索的标签进行比较。而我们的做法使用了职责链模式来确定是否可以早一点结束搜索。在最差的情况下，最后仍然是叶节点在处理请求，但要是某个组合对象节点具有那个标签的话，那么请求可能在层次体系中往上几个层次的地方就能得到处理。

不过这些好处并非没有代价。我们仍然需要获取在层次体系中位于具有匹配标签的组合对象之下的每一个图片对象。这需要使用getAllLeaves方法，它所耗费的计算量比getPhotosWithTag少得多。我们的办法也适用于其他组合对象方法。实际上，组合对象方法需要耗费的计算量越大，在层次体系中较高的层次上处理请求并使用某种辅助方法（比如getAllLeaves）提高请求的处理效率所带来的好处也就越大。

## 17.7 职责链模式之利

**261** 借助职责链模式，可以动态选择由哪个对象处理请求。这意味着你可以使用只有在运行期间才能知道的条件来把任务分派给最恰当的对象。图片库的例子已经表明，这可以比试图在开发期间静态指定处理请求的对象高效得多。你还可以使用这个模式消除发出请求的对象与处理请求的对象之间的耦合。藉此你可以在模块的组织方面获得更大的灵活性，而且在重构和修改代码的时候不用担心会把类名固化在算法中。

在已经有现成的链或层次体系的情况下，职责链模式更加有效。与组合模式的结合使用就属于这种情况。你可以重用组合对象的结构来传递请求，直到找到一个可以处理请求的对象。在此情况下，不用编写粘合性代码来实例化那些对象或建立链，因为那些东西早已准备妥当。藉此可以实现把请求转给恰当的处理程序的方法。

## 17.8 职责链模式之弊

由于在职责链模式中，请求与具体的处理程序被隔离开来，因此无法保证它一定会被处理，而不是径直从链尾离开。这种模式使用的接收者是隐式的（implicit），因此无法得知如果请求能够得到处理的话具体将由哪个对象处理它。这个问题可以通过创建一个通用的（catch-all）接收者并将其添加到所有链的尾端来解决，但是这个办法很繁琐，而且这样一来就失去了可以随时在链尾添加新环节的灵活性。

职责链与组合对象类的搭配使用可能有点令人困惑。人们原本期望，组合对象节点完全可以与叶节点互换使用，而且客户代码看不出其中的差别，所有方法调用都被组合对象往层次体系的

下层传递。但职责链方法改变了这个约定。在引入职责链之后，有些方法会在组合对象这里进行处理，而不会继续往下传。要想让这些方法可以与叶方法互换，其代码的编写会很棘手。它们的效率很高，但为此付出的代价是代码的复杂性。

## 17.9 小结

本章考察了一种用来隔离请求的发送者和接收者的技术。职责链模式可以用来消除客户与处理程序之间的耦合，并生成由可以对请求进行处理的对象组成的链。使用这种模式，你可以编写代码在运行期间选择最合适的目标来作接收者，从而避免把接收者绑定到具体的类。如果有现成的链或层次体系可供利用，那么实现这个模式只是举手之劳。组合对象层次体系就非常适合于这一任务，而且它自身也会因为职责链的引入而更有效率。

这个模式的正确设置和使用都很复杂，因此只应该用在必要的地方。此外，它还是一种隐式的处理程序，因此无法得知处理请求的具体是链上的哪个环节。请求有可能根本不会被处理。但是在那些可以受益于职责链模式的场合下，比如前面为图片添加标签那个例子中，该模式可以提高算法的效率并降低其计算量。

262

# 索引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

## 数字和符号

\$ function (\$ 函数), 151-152

## A

accessor methods (取值器方法), 29, 38, 89-90  
action method (action方法), 233  
ActiveXObject, 100  
adapters/adapter pattern (适配器/适配器模式), 111  
adapting libraries (example) (对库进行适配 (示例)), 150-152  
benefits of (之利), 149, 158  
characteristics of (特点), 149-150  
creating (创建), 150  
drawbacks of (之弊), 158  
email API (example) (电子邮件API (示例)), 152-158  
for existing implementations (适配原有实现), 150  
vs. facade pattern (与门面模式的对比), 144, 149  
when to use (适用场合), 158  
addEvent function (addEvent函数), 142, 146-147  
addEventListener function (addEventListener函数), 142  
addForm function (addForm函数), 18  
addRequest function (addRequest函数), 122  
Ajax, 99  
Ajax request queue (Ajax请求队列), 111-122  
AjaxHandler interface (AjaxHandler接口), 103  
aliases (别名), 78  
animation, using observer pattern (动画, 使用观察者模式), 221  
anonymous functions (匿名函数), 6-8  
API class (API类), 89  
API2 class (API2类), 89  
APIs  
development, using bridges (开发, 使用桥接模式),

109-110

using adapter pattern with (使用适配器模式), 152-158  
array methods (数组方法), 113-114  
asymmetry, in prototypal inheritance (原型式继承中的不对等性), 46-48  
asyncRequest function (asyncRequest函数), 156  
attribute checking, emulating interfaces with (属性检查, 用于模拟接口), 16-17  
attributes, static (属性, 静态的), 35-37  
augment method (augment方法), 51, 59-63

## B

behaviors (行为)  
adding after method (在方法之后添加行为), 164  
adding before method (在方法之前添加行为), 165  
branching (分支), 65, 78-82  
bridges/bridge pattern (桥接元素/桥接模式)  
benefits of (之利), 123  
connecting multiple classes with (用以联结多个类), 111  
drawbacks of (之弊), 123  
event listener callbacks with (事件监听器回调与), 109-110  
introduction to (介绍), 109  
uses of (应用), 110, 122  
XHR connection queue (example) (XHR连接队列 (示例)), 111-122  
browsers, encapsulation of differences (浏览器, 对其差异的封装), 78-81  
buttons (按钮)  
command (命令), 237-239  
undo (销消), 237

## C

- C#, interfaces in (C#, 接口在其中的用法), 14  
 callbacks (回调函数), 213  
   event listener (事件监听器), 109-110, 142, 222  
   using to retrieve data from chained methods (用以从链式调用的方法中获取数据), 89-90  
 catch-all receivers (通用接收者), 262  
 chain of responsibility pattern (职责链模式)  
   benefits of (之利), 261  
   composite pattern and (与组合模式的关系), 254-262  
   drawbacks of (之弊), 262  
   event delegation (事件委托), 255  
   image gallery (example) (图片库 (示例)), 256-261  
   implementing, in existing hierarchy (实现, 在现有层次体系中), 254-255  
   introduction to (介绍), 245  
   objects in (其中的对象), 249-251  
   passing requests in (传入请求), 251-254  
   structure of (其结构), 245-251  
   using with composite pattern (与组合模式结合使用), 257-262  
   when to use (适用场合), 255-256  
 chains/chaining (方法调用链/链式调用), 89  
   building chainable JavaScript library (设计支持链式调用的JavaScript库), 86, 88  
   callbacks and (回调在其中的应用), 89-90  
   introduction to (介绍), 83  
   structure of (其结构), 84-86  
 \_checkInitialization method (\_checkInitialization方法), 211  
 checkISBN method (checkISBN方法), 29, 37  
 child objects (子对象), 47-48, 57  
 class declarations (类的声明), 42  
 classes (类)  
   composite (组合对象), 231-232  
   interfaces for (其接口), 26  
   mixin (掺元), 50-51, 59-62  
   using bridges to connect multiple (用桥接元素连接多个类), 110-111  
   参见各种具体的类  
 classical inheritance (类式继承), 42-45  
   edit-in-place field using (用于实现就地编辑域), 52-55  
   use of (适用场合), 62  
   vs. prototypal (与原型式继承的对比), 49  
 click handlers (click事件处理器), 241  
 click listeners (click事件监听器), 238  
 clients, in command pattern (客户, 命令模式中的), 227-228  
 clone function (clone函数), 46-49, 57-58, 62  
 closures (闭包), 7, 33-36  
   concept of (概念), 26  
   creating commands with (用以创建命令), 227  
   to hide information (用于隐藏信息), 33-38  
   using for private members in singleton objects (用于实现单体对象中的私用成员), 71-73  
 code (代码)  
   bridge between public and private (桥接公用和私用代码), 110  
   loosely coupled (弱耦合的), 234  
   organization of (其组织), 65, 81  
   page-specific (特定网页专用的), 68-70  
   reusability of (其可重用性), 11, 39, 41, 50-51  
 code duplication, prevention of (代码的重复, 如何防止), 107  
 command buttons (命令按钮), 237-239  
 command class (命令类), 233  
 command objects/pattern (命令对象/模式), 23  
   benefits of (之利), 243  
   building user interfaces with (用以构建用户界面), 230-235  
   clients in (其中的客户), 227-228  
   with decorator pattern (结合装饰者模式), 237-239  
   drawbacks of (之弊), 243  
   execute method (execute方法), 235-239  
   introduction to (介绍), 225  
   invoking object in (其中的调用者), 227-228  
   logging and (日志与), 235, 237-242  
   menu items (example) (菜单项 (示例)), 230-235  
   receiving object in (其中的接收者), 227-228  
   structure of (其结构), 225-228  
   undo method (undo方法), 235-242  
   using interfaces with (在其中使用接口), 228, 230  
   when to use (适用场合), 242-243  
 commands (命令)  
   creating with closures (用闭包创建), 227  
   reusing (重用), 235  
   参见各种具体的命令  
 comments, describing interfaces with (注释, 用以说明接口), 14-15  
 components (组件)  
   adding new methods (添加新方法), 167-169  
   replacing methods of (替换其方法), 166-167  
 composite classes (组合对象类), 231-232

- composite elements (组合元素), 127-131  
 composite pattern (组合模式), 23  
   adding more classes to hierarchy (向层次体系中添加类), 133-136  
   benefits of (之利), 125, 139  
   chain of responsibility pattern and (职责链模式与), 254-262  
   vs. decorator pattern (与装饰者模式的对比), 163  
   drawbacks of (之弊), 139-140  
   form validation (example) (表单验证 (示例)), 127-136  
   image gallery (example) (图片库 (示例)), 136-139  
   for managing extrinsic state (用于管理外在数据), 183-186  
   structure of (其结构), 126  
   using (用法), 126  
 CompositeFieldset class (CompositeFieldset类), 134-136  
 configure method (configure方法), 57  
 connection objects (连接对象)  
   choosing, at run-time (在运行期间选择), 103-104  
   specialized (专用型), 101-102  
 constants (常量), 37-38  
 constructor attribute (constructor属性), 45  
 constructors (构造函数), 42  
 convenience functions (便利函数), 143-144, 147  
 cookies, 131  
 core utilities, for request queue (用于请求队列的核心工具), 112-114  
 crash recovery (崩溃恢复), 242  
 createXhrHandler method (createXhrHandler方法), 104  
 createXhrObject method (createXhrObject方法), 101  
 Cursor class (Cursor类), 237, 240-241
- D**
- data (数据)  
   extrinsic (外在的), 179, 182-183, 186-189, 193  
   intrinsic (内在的), 179  
   retrieval, from chained methods (从支持链式调用的方法获取), 89-90  
 data encapsulation (数据封装)  
   data integrity and (数据完整性与), 39  
   in manager (管理器中的), 182-183  
 debugging (调试)  
   command pattern and (命令模式与), 243  
   encapsulation and (封装与), 39  
   flyweight pattern and (享元模式与), 194  
   interfaces and (接口与), 12  
   decorator objects (装饰者对象), 201  
   order of (其顺序), 168-169  
   using factory objects to create (用工厂对象创建), 169-172  
 decorator pattern (装饰者模式), 23  
   adding behavior after method (在方法之后添加行为), 164  
   adding behavior before method (在方法之前添加行为), 165  
   adding new methods (添加新方法), 167-169  
   benefits of (之利), 176  
   with command pattern (与命令模式结合), 237-239  
   vs. composite pattern (与组合模式的对比), 163  
   drawbacks of (之弊), 176  
   for functions (用于函数), 172-173  
   method profiler (example) (方法性能分析器 (示例)), 173-176  
   modifying behavior of components with (用以修改组件行为), 164-169  
   vs. proxy pattern (与代理模式的对比), 201  
   replacing methods (替换方法), 166-167  
   role of interface in (接口在其中的角色), 163  
   structure of (其结构), 159-163  
   use of (适用场合), 159, 173  
 decoupling (消除耦合), 262  
 dedMail, migrating from fooMail to (从fooMail转向dedMail), 157-158  
 delete keyword (delete关键字), 66  
 deliver method (deliver方法), 219  
 dependencies, hard-coded (依赖, 固化的), 249  
 dequeue (出列), 114  
 design patterns (设计模式) 参见patterns  
 DHTML, 144-145  
 dialog boxes, flyweight pattern for (对话框, 应用享元模式), 190-192  
 directory lookup (example) (目录查找 (示例)), 206-210  
 display method (display方法), 27-29  
 DOM elements, reducing needed (减少需要的DOM元素), 186-190  
 duck typing, emulating interfaces with (鸭式辨型, 用以模拟接口), 17-18  
 dynamic implementations (动态实现), 99  
 dynamic user interface (动态用户接口), 225-226  
 dynamic virtual proxy, creating (动态虚拟代理, 创建),

210-213

## E

edit-in-place field (example) (就地编辑 (示例)), 52  
 using classical inheritance (使用类式继承), 52-55  
 using mixin classes (使用掺元类), 59-62  
 using prototypal inheritance (使用原型式继承), 55-58  
 efficiency, of virtual proxies (效率, 虚拟代理的), 213  
 email API, adapter pattern with (电子邮件API, 与适配器模式), 152-158  
 encapsulation (封装)  
   benefits of (之利), 39  
   defined (定义), 26  
   drawbacks of (之弊), 39-40  
   vs. information hiding (与信息隐藏的对比), 26  
   inheritance and (继承与), 49-50  
 encapsulation functions (封装函数), 226-227, 243  
 ensureImplements function (ensureImplements函数), 18  
 event bubbling (事件冒泡), 255  
 event capturing (事件捕获), 255  
 event delegation (事件委托), 122, 255  
 event listeners (事件监听器)  
   with bridge pattern (与桥接模式), 109-110  
   as observers (作为观察者), 222  
   use of, in JavaScript (在JavaScript中的应用), 142  
 event objects (事件对象), 255  
 event utility, creating with facade pattern (事件工具, 用门面模式创建), 146-147  
 event-driven environments (事件驱动的环境), 215  
 execute method (execute方法), 226, 229, 235-239  
 extend function (extend函数), 43-45, 55, 62  
 extend keyword (extend关键字), 42  
 externally hosted libraries (部署在外部环境中的库), 20  
 extrinsic data (外在数据), 179, 192  
   encapsulation of (其封装), 182-183  
   removing from class (从类中删除), 188-189  
   storage of (其保存), 186, 193  
 extrinsic state (外在状态), 180-186

## F

facades/facade pattern (门面/门面模式), 201  
 vs. adapter pattern (与适配器模式的对比), 144, 149  
 addEvent function (addEvent函数), 142, 146-147  
 benefits of (之利), 148  
 vs. bridges (之弊), 111

common examples of (常见例子), 141-142  
 as convenient methods (用作便利方法), 143-144  
 creating event utility (example) (创建事件工具 (示例)), 146-147  
 drawbacks of (之弊), 148  
 introduction to (介绍), 141  
 JavaScript libraries as (JavaScript库的门面性质), 142  
 setting styles on HTML elements(example) (设置HTML元素的样式 (示例)), 144-145  
 steps for implementing (实现步骤), 147  
 uses of (应用), 141, 148  
 factory method, changing function into (把函数改为工厂方法), 84  
 factory pattern (工厂模式), 23, 38, 83  
   benefits of (之利), 107  
   creating decorator objects with (用以创建装饰者对象), 169-172  
   creating, for flyweight pattern (创建, 为实现享元模式), 193  
   drawbacks of (之弊), 108  
   example (示例), 99-104  
   instantiation using (用以进行实例化), 181  
   RSS reader (example) (RSS阅读器 (示例)), 104-107  
   simple (简单工厂), 93-96  
   true (真正的工厂模式), 96-98  
   uses of (应用), 99  
 fieldsets (域集), 128  
 findProduct function (findProduct函数), 67  
 flyweight manager (享元管理器), 193  
 flyweight pattern (享元模式)  
   calendar (example) (日历 (示例)), 185-186  
   car registrations (example) (汽车登记 (示例)), 179-183  
   composite pattern with (与组合模式结合), 183-186  
   converting objects to (把对象转化为), 185  
   drawbacks of (之弊), 194  
   encapsulation of extrinsic data (外在数据的封装), 182-183  
   extrinsic state and (外在状态与), 180-186  
   for dialog boxes (用于对话框), 190-192  
   instantiation using a factory (用工厂实例化), 181  
   intrinsic state and (内在数据与), 180-181  
   introduction to (介绍), 179  
   steps for implementing (实现步骤), 193  
   storing instances for reuse (保存实例以供重用), 190-192  
   structure of (其结构), 179

tooltip objects (example) (工具提示对象 (示例)), 186-190  
 web calendar (example) (Web日历 (示例)), 183-185  
 when to use (适用场合), 192.  
 fooMail, migrating to dedMail from (从dedMail转向 fooMail), 157-158  
 form elements (表单元素), 127-128  
 form validation, using composite pattern (表单验证, 使用组合模式), 127-136  
 FormItem interface (FormItem接口), 128, 133  
 fully exposed objects (门户大开型对象), 27-30  
 functions (函数)  
     combining, with facades (用门面组合), 143-144  
     convenience (便利), 143-144  
     decorator pattern for (针对函数的装饰模式), 172-173  
     as first-class objects (作为一等对象), 6-8  
     in JavaScript (JavaScript中的), 6-8  
     modifying into factory method (修改为工厂方法), 84  
     nested (嵌套), 32-33, 36  
     as objects (作为对象), 6-8, 37  
     privileged (特权函数), 110  
     scope of (其作用域), 33  
     singletons as (创建单体), 72-73  
     参见各个具体的函数

**G**

getValue method (getValue方法), 131-132  
 getXHR function (getXHR函数), 156  
 GUI-based operating systems (基于GUI的操作系统), 141

**H**

handleFilingRequest method (handleFilingRequest方法), 249  
 hard-coded dependencies (固化的依赖性), 249  
 HAS-A relationships (HAS-A (具一) 关系), 128  
 hasOwnProperty method (hasOwnProperty方法), 47  
 hierarchical structures, composite objects and (层次性结构, 组合对象与), 139  
 HTML elements (HTML元素)  
     JavaScript objects as wrappers for (JavaScript对象作为其包装器), 136-139  
     setting styles on, with facade pattern (设置其样式, 用门面模式), 144-145

**I**

image gallery (图片库)  
     adding tags to photos in (为图片添加标签), 258-261  
     using chain of responsibility pattern (使用职责链模式), 256-261  
     using composite pattern (使用组合模式), 136-139  
 implements keyword (implements关键字), 13-14  
 information hiding (信息隐藏)  
     encapsulation and (封装与), 26, 39-40  
     principle of (信息隐藏原则), 25-26  
     role of interface in (接口在其中的角色), 26  
     using closures (使用闭包), 33-38  
     using naming conventions (使用命名规范), 30-32  
 inheritance (继承), 9, 35  
     classical (类式), 42-45, 49, 52-55, 62  
     classical vs. prototypal (类式继承与原型式继承的比较), 49  
     encapsulation and (封装与), 49-50  
     introduction to (介绍), 41  
     prototypal (原型式), 41, 45-49, 55-58  
     when to use (适用场合), 41, 62  
 init method (init方法), 68  
 \_initialize method (\_initialize方法), 211  
 instanceof check (instanceOf检查), 22  
 instances, storing for reuse (实例, 保存以供重用), 190-192  
 instantiation (实例化)  
     lazy (惰性实例化), 75-78  
     using a factory (使用工厂), 181  
 Interface class (Interface类), 18-22  
 Interface helper class (Interface助理类), 18  
 interface keyword (interface关键字), 14  
 Interface objects (Interface对象), 20  
 Interface.ensureImplements class method  
     (Interface.ensureImplements类方法), 18  
 interfaces (接口)  
     benefits of using (使用接口的好处), 11  
     in C# (C#中的), 14  
     for command pattern (用于命令模式), 228-230  
     for composite objects (用于组合对象), 140  
     for decorator pattern (用于装饰者模式), 163  
     defined (定义), 11  
     describing with comments (用注释说明), 14-15  
     drawbacks of using (使用接口的不利之处), 12  
     emulating, in JavaScript (在JavaScript中模仿), 14-18  
     implementation for book (本书中使用的Interface实现), 18

importance of (其重要性), 11  
 in Java (Java中的), 13  
 patterns relying on (依赖于接口的模式), 23  
 in PHP (PHP中的), 13-14  
 role of, in information hiding (在信息隐藏中的角色), 26  
 intrinsic data (内在数据), 179  
 intrinsic state (内在状态), 180-181  
 introspection (内省), 9  
 invoker class (调用者类), 232  
 invoking objects, in command pattern (调用者, 命令模式中的), 227-230  
 IS-A relationships (IS-A (为一) 关系), 128  
 isHighLatency method (isHighLatency方法), 104  
 \_isInitialized method (\_isInitialized方法), 211-212  
 isOffline method (isOffline方法), 104

**J****JavaScript**

design patterns (设计模式), 9-10  
 emulating interfaces in (在其中模仿接口), 14-18  
 encapsulation in (其中的封装), 40  
 event listeners and (事件监听器与), 142  
 features of (其特性), 3-9  
 flexibility of (其灵活性), 3-6, 210  
 functions in (其中的函数), 6-8  
 inheritance in (其中的继承), 9  
 lack of interface support in (其中缺乏对接口的支持), 12  
 as loosely typed language (作为一种弱类型语言), 6  
 objects in (其中的对象), 8-9

**JavaScript libraries (JavaScript库)**

chainable (支持方法链式调用的), 86-88  
 common features of (其常见特性), 86  
 facades and (门面与), 141-142  
 using adapter pattern with (使用适配器模式), 150-152

**JavaScript objects, as wrappers for HTML elements**

(JavaScript对象, 作为HTML元素的包装器), 136-139

**L**

large projects, use of interfaces in (大型项目, 在其中使用接口), 20  
 lazy loading (惰性加载), 75-78, 82, 223  
 leaf elements (叶元素), 127-128, 131  
 libraries (库) 参见JavaScript libraries

ListBuilder class (ListBuilder类), 174-176  
 logging commands (用命令日志记录命令), 239-241  
 command pattern and (命令模式与), 235-242  
 for crash recovery (用于崩溃恢复), 242  
 for implementing undo on nonreversible actions (用于实现不可逆操作的取消), 239-242  
 loosely coupled code (松散耦合的代码), 234  
 loosely coupled modules (松散耦合的模块), 39  
 loosely coupled objects (松散耦合的对象), 41, 107, 139  
 loosely typed variables (弱类型变量), 6

**M**

maintenance, flyweight pattern and (维护, 享元模式与), 194  
 manager, creating to store extrinsic data (管理器, 为保存外在数据而创建), 193  
 memoizing, 101  
 method profiling, using decorator pattern (方法性能分析, 使用装饰者模式), 173-176  
**methods (方法)**  
 adding behavior after (在方法后添加行为), 164  
 adding behavior before (在方法前添加行为), 165  
 adding new (添加新方法), 167-169  
 chaining (链式调用), 83-90  
 mutator (赋值器), 29-30  
 private (私用方法), 25, 37, 39, 204  
 privileged (特权方法), 34, 110  
 public (公用方法), 34  
 replacing (替换), 166-167  
 static (静态的), 35-37, 75  
 参见各种具体方法

**mixin classes (掺元类), 50-51, 59-62**

**modular user interfaces, using command pattern (模块化用户界面, 使用命令模式), 230-235**  
**mutable objects (易变对象), 8-9**  
**mutator methods (赋值器方法), 29-30**

**N**

**namespaces, singletons and (命名空间, 单体与), 65-68, 81**  
**naming conventions, emulating private members using (命名规范, 用以模仿私用成员), 30-32**  
**nested functions (嵌套函数), 32-33, 36**  
**new keyword (new关键字), 42, 99**  
**new operator (new运算符), 42, 57**

nonreversible actions, implementing undo with (不可逆操作, 实现其取消), 239-242

## O

object factories (对象工厂), 38  
 object literals, singletons and (对象字面量, 单体与), 66, 72  
 objects (对象)  
     adding functionality to, with decorator pattern (用装饰者模式为其增添功能), 159-163  
     in chain of responsibility pattern (职责链模式中的), 249-251  
     cloned (克隆的), 48-49  
     converting to flyweights (转化为享元), 185  
     encapsulation of multiple into one large (将多个封装为一个大的), 99  
     fully exposed (门户大开型), 27-30  
     intrinsic vs. extrinsic state (内在状态与外在状态的比较), 180-181  
     invoking in command patter (在命令模式中调用), 227-230  
     loosely coupled (松散耦合的), 41, 107, 139  
     mutability of (其易变性), 8-9  
     observable (可观察的), 217, 223  
     patterns for (用于对象的模式), 26-39  
     prototype (原型), 45-49, 57-58  
     receiver (接收者), 227-230, 245, 262  
     reducing required number of (削减所需对象的数量), 179  
     request (请求), 251-254  
     sender (发送者), 245  
     static members (静态成员), 35-37  
     using naming conventions (使用命名规范), 30-32  
     wrapper (包装器), 149  
     参见各种具体类型

observable objects (可观察对象), 217, 223  
 observed role (被观察的角色), 215-218  
 observer pattern (观察者模式)  
     animation (example) (动画 (示例)), 221  
     benefits of (之利), 223  
     building observer API (构建观察者API), 218-220  
     delivery methods in (其中的投送方法), 216, 219  
     drawbacks of (之弊), 223  
     event listeners (事件监听器), 222  
     implementation of (其实现), 216-218  
     introduction to (介绍), 215

publishers in (其中的发布者), 215-218  
 for request queue (用于请求队列), 114  
 subscribe method (subscribe方法), 219-220  
 subscribers in (其中的订阅者), 215-218  
 unsubscribe method (unsubscribe方法), 220  
 uses of (应用), 220, 223  
 observer role (观察者角色), 215-218  
 optimization (优化)  
     with chain of responsibility pattern (使用职责链模式), 258-261  
     with flyweight pattern (使用享元模式), 179-192  
     with proxy pattern (使用代理模式), 213

## P

page-specific code, singleton as wrapper for (专用于特定网页的代码, 单体用作其包装器), 68-70  
 parentheses, empty (空括号), 36  
 patterns (模式), 9-10  
     adapter (适配器模式), 149-158  
     bridge (桥接模式), 109-123  
     chain of responsibility (职责链模式), 245-262  
     closure (闭包), 33-35  
     command (命令模式), 23, 225-244  
     composite (组合模式), 23, 125-140  
     constants (常量), 37-38  
     for creating objects (用于创建对象的), 26-39  
     decorator (装饰者模式), 23, 159-177  
     facade (门面模式), 141-148  
     factory (工厂模式), 23, 38, 93-108  
     flyweight (享元模式), 179-195  
     fully exposed object (门户大开型对象), 27, 29-30  
     observer (观察者模式), 215-223  
     private members using naming convention (用命名规范区别私用成员), 30-32  
     proxy (代理模式), 197-214  
     relying on interfaces (对接口的依赖), 23  
     singleton (单体模式), 38, 65-82  
     static methods and attributes (静态方法和属性), 35-37  
     performance issues (性能问题), 12, 204  
     PHP, interfaces in (PHP, 其中的接口), 13-14  
     pooling technique (对象池技术), 193  
     primitive types (原始类型), 37  
     private attributes (私用属性), 25  
     scope and (作用域与), 32-33  
     through closures (用闭包实现), 33-35

private code, bridge between public code and (私用的代码, 它与公开的代码之间的桥接元素), 110

private keyword (private关键字), 26

private members (私用成员)

- emulating, using naming conventions (用命名规范模仿), 30-32
- singleton with (带私用成员的单体), 70-75
- syntax for (其语法), 75
- through closures (用闭包创建), 33-35

private methods (私用方法), 25, 37, 39, 204

privileged functions (特权函数), 110

privileged methods (特权方法), 34, 110

programming styles (编程风格), 3

protection proxies (保护代理), 200

prototypal inheritance (原型式继承), 41, 45-49

- vs. classical (与类式继承的比较), 49

edit-in-place field using (使用原型式继承的就地编辑域), 55-58

use of (应用), 62

prototype attribute (prototype属性), 46-48

prototype chaining (原型链接), 42-43, 46-48

prototype method (在prototype中定义方法), 137

prototype objects (原型对象), 45-49, 57-58

proxy objects/pattern (代理对象/模式)

- access control to real subject by (用以控制对本体的访问), 197-200
- benefits of (之利), 213
- vs. decorator pattern (与装饰者模式的比较), 201
- directory lookup (example) (目录查找(示例)), 206-210
- drawbacks of (之弊), 213-214
- function of (功能), 197
- general, for virtual proxy (用于虚拟代理的通用模式), 210-213
- introduction to (介绍), 197
- page statistics (example) (网页统计(示例)), 201-204
- protection (保护代理), 200
- remote (远程代理), 200-204, 213
- structure of (其结构), 197-201
- use of (应用), 201
- virtual (虚拟代理), 199-201, 206-213
- for wrapping web services (用于包装Web服务), 201-206

PS2 slot (PS2插口), 150

PS2-to-USB adapter (PS2-to-USB适配器), 150

public code, bridge between private code and (公开的代码,

它与私用的代码之间的桥接元素), 110

public members (公用成员), 37, 75

public methods (公用方法), 34, 204

publisher-subscriber pattern (发布者-订阅者模式) 参见 observer pattern

publishers (发布者), 215-218

pull delivery environment (“拉”投送环境), 216

push delivery environment (“推”投送环境), 216

**Q**

queuing system (队列化系统), 111-122

**R**

real subjects (本体), 197-200, 212-214

receiver classes (接收者类), 230

receiving objects (接收者), 227-230, 245, 262

refactoring (重构), 262

reflection (反射), 9

reliability issues, with remote proxy (可靠性问题, 远程代理的), 213

remote proxies (远程代理), 200

- benefits of (之利), 213
- drawbacks of (之弊), 213
- performance issues with (其性能问题), 204
- use of (应用), 201
- for wrapping web service (用于包装Web服务), 201-204

renderResults method (renderResults方法), 22

request handling, chain of responsibility model and (请求的处理, 职责链模式与), 261

request method (request方法), 102-103

request objects (请求对象), 251-254

request queue (请求队列), 111-122

ResultFormatter class (ResultFormatter类), 21-22

reusability (可重用性), 11, 39, 50-51

RSS reader, using factory pattern (RSS阅读器, 使用工厂模式), 104-107

**S**

save function (save函数), 128, 130-131

scope (作用域), 32-33, 39

Sellsian approach (Sells方法), 216-217

sender objects (发送者对象), 245

serialize method (serialize方法), 50

setup costs, combining using factory pattern (设置开销, 通

- 过工厂模式节约), 99  
 shortcut icons (快捷方式图标), 141  
 SimpleHandler class (SimpleHandler类), 101  
 singleton pattern (单体模式), 38, 66  
     basic structure of (其基本结构), 65-66  
     benefits of (之利), 81  
     branching (分支), 78-81  
     drawbacks of (之弊) 82  
     introduction to (介绍), 65  
     lazy loading (惰性加载), 75-78, 82  
     namespacing (划分命名空间), 66-68  
     with private members (含私用成员的), 70-75  
     uses of (应用), 65, 81  
     as wrapper for page-specific code (用作专用于特定网页的代码的包装器), 68-70  
 static attributes (静态属性) 35-37  
 static methods (静态方法), 35-37, 75  
 strict type checking (严格的类型检查), 12, 18  
 subclasses (子类), 35, 212-213  
 subscribe method (subscribe方法), 219-220  
 subscribers (订阅者), 215-218  
 successors (下家), 249  
 superclasses (超类), 35, 44-45
- T**
- test-driven development (TDD) (测试驱动的开发), 216  
 testing, interfaces and (接口与测试), 12  
 this keyword (this关键字), 27, 34, 36-37, 42, 72, 233  
 tooltip objects, with flyweight pattern (工具提示对象, 使用享元模式), 186-190  
 type checking (类型检查), 12, 18, 20
- U**
- underscore notation (下划线表示法), 32, 70-71, 74  
 undo buttons (取消按钮), 237-239
- undo method (undo方法), 235, 237-242  
 UndoButton class (UndoButton类), 239  
 UndoDecorator class (UndoDecorator类), 239  
 unsubscribe method (unsubscribe方法), 220  
 USB system (USB系统), 150  
 user interfaces, with command pattern (用户界面, 使用命令模式), 225-226, 230-235
- V**
- validate function (validate函数), 127-128  
 var keyword (var关键字), 36, 67, 75  
 variables (变量)  
     closures for (把闭包用于), 33-35  
     constants (常量), 37-38  
     singleton (单体), 66  
 virtual proxies (虚拟代理), 199-200  
     benefits of (之利), 213  
     directory lookup (example) (目录查找(示例)), 206-210  
     drawbacks of (之弊), 213  
     general pattern for creating (用于创建虚拟代理的通用模式), 210-213  
     use of (应用), 201
- W**
- web services (Web服务)  
     defining interface for (为其定义接口), 203-204  
     proxy pattern for (代理模式用于), 201-206  
 webmail API, using adapter pattern with (Web邮件API, 使用适配器模式), 152-158
- X**
- XHR objects (XHR对象), 99-101  
 XHR requests (XHR请求), 208, 213  
 XMLHttpRequest object (XMLHttpRequest对象), 100, 156

# 计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

**Java** 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

**.Net** 技术精品资料下载汇总: **ASP.NET** 篇

**.Net** 技术精品资料下载汇总: **C#语言** 篇

**.Net** 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++** 编程语言学习资料尽收眼底 电子书+视频教程

**Visual C++(VC/MFC)** 学习电子书及开发工具下载

**Perl/CGI** 脚本语言编程学习资源下载地址大全

**Python** 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

**UML** 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

**Linux** 系统管理员必备参考资料下载汇总

**Linux shell**、内核及系统编程精品资料下载汇总

**UNIX** 操作系统精品学习资料<电子书+视频>分类总汇

**FreeBSD/OpenBSD/NetBSD** 精品学习资源索引 含书籍+视频

**Solaris/OpenSolaris** 电子书、视频等精华资料下载索引

“本书道前人所未道，引导你从编写代码进化为设计代码。书中绝大部分示例代码都来自YUI等实战项目，并进行了深入剖析。强烈推荐。”

——Nicholas C. Zakas，著名JavaScript专家，Yahoo前端工程师，畅销书《JavaScript高级程序设计》作者

“毫不夸张地说，这是我有生以来读到的最好的一本JavaScript图书。作者讲授了大量独门的专家经验。”

——Mostafa Farghaly，埃及程序员

# Pro JavaScript Design Patterns JavaScript设计模式

Web应用取代桌面程序的时代已经到来！作为Web前端的核心技术，JavaScript的重要性不言而喻，它有望成为下一代统治性程序语言。但由于业界长期的误解和滥用，也有不少人仍然对此半信半疑。那么，JavaScript到底能否当此大任呢？

本书中，Google和Yahoo公司的两位资深Web专家对此给出了掷地有声的肯定回答。作者针对常见的开发任务，从YUI等实战代码中取材，提供了专家级的解决方案，不仅透彻剖析了JavaScript中的面向对象编程，而且深入探讨了如何用JavaScript实现以前只在服务器端应用的设计模式，如何根据实际场景选择恰当的设计模式，开发出高质量的企业级代码。本书充分证明：JavaScript不仅毫不逊色于其他高级语言，已经是一种成熟且强大的面向对象语言，而且还拥有Java和C++等语言不具备的面向未来的特性，因此更加灵活、更富于表现力。

无论是前端工程师，还是服务器端程序员，通过本书都将使自己的JavaScript功力提升到前所未有的高度。



**Ross Harms** 资深Web程序员，有10多年编程经验。现任Yahoo前端工程师。他是开源图片博客软件Birch的开发者。Blog地址为<http://techfoolery.com>。



**Dustin Diaz** 资深Web程序员，现任Google用户界面工程师。新一代JavaScript框架DED|Chain（兼具jQuery和YUI的优势）的开发者。他还是一位中长跑健将，800米跑曾经在全国排名第13。拥有西班牙语学士学位。个人网站<http://dustindiaz.com>。

图灵Web开发图书阅读路线图



Apress®

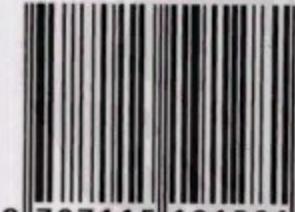
本书相关信息请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)88593802

反馈/投稿/推荐信箱：[contact@turingbook.com](mailto:contact@turingbook.com)



ISBN 978-7-115-19128-1



9 787115 191281 >

ISBN 978-7-115-19128-1/TP

定价：45.00 元

上架建议 计算机/网络开发/程序设计

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