

SPA

设计与架构

理解单页面Web应用

SPA Design and Architecture:
Understanding Single-page Web Applications

[美] Emmit A. Scott 著
卢俊祥 译



Web开发

Web 应用发展的下一个热点是单页面 Web 应用程序，其将原生桌面应用的流畅体验带到了浏览器。如果你打算从传统 Web 应用跨越到 SPA 却又无从下手，那么这本书正是为你准备的。

本书讲述 SPA 应用程序构建所需的设计与开发技术。书中首先介绍 SPA 模型，并阐述 SPA 标准构建方式。随着内容的展开，作者通过具体的 SPA 构建知识点引导你前进，涵盖 MV* 框架、单元测试、路由、布局管理、数据访问、发布 / 订阅模式以及客户端任务自动化等内容。书中示例丰富易懂，并可结合各种第三方库或框架来创建。

本书内容：

- 模块化JavaScript实践
- 理解MV*框架
- 布局管理
- 客户端任务自动化
- SPA应用程序测试

本书针对 Web 开发人员，读者需掌握 JavaScript 基础知识。

Emmit A. Scott 是一位高级软件工程师及架构师，在大规模 Web 应用项目构建方面积累了相当多的经验。

本书主页为 manning.com/books/spa-design-and-architecture，并提供 PDF、ePub 和 Kindle 格式的电子版。



博文视点Broadview



@博文视点Broadview



MANNING



策划编辑：张春雨
责任编辑：李云静
封面设计：吴海燕

“涉及复杂主题，并拆解为易于消化的部分。”

——Burke Holland,
本书序作者, Telerik

“非常棒的热门开发技术资源。”

——Bruno Sonnino,
Revolution 软件

“清晰易懂、多角度、结构合理，
阐述现代 SPA 应用程序的内涵，高度
推荐！”

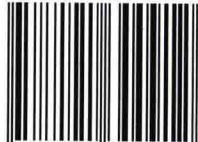
——Alain Couniot,
STIB-MIVB

“代码示例详细、丰富并极具实用性，
为知识内容提供了实践环境。”

——John Shea,
埃迪柯特学院

上架建议：前端 / Web开发

ISBN 978-7-121-30091-2



9 787121 300912 >

定价：79.00元

SPA 设计与架构

理解单页面Web应用

SPA Design and Architecture:
Understanding Single-page Web Applications

[美] Emmit A. Scott 著
卢俊祥 译



电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

SPA 开发技术的运用是当今 Web 开发领域的热门趋势,但真正全面掌握该技术的开发者并不多。本书详尽阐述单页面 Web 应用(SPA)开发技术,从 SPA 构建基础入手,通过 MV*、模块化编程、路由、模块间通信、服务器端交互等概念的阐述,全面介绍 SPA 的设计与架构,帮助读者正确掌握 SPA 开发的各方面知识要素。同时,书中还讨论了 SPA 的单元测试及客户端任务自动化,覆盖了从开发到部署的一系列任务,让读者在阅读完本书之后能够打下扎实的 SPA 开发基础。

本书的重点是帮助读者正确、全面地掌握 SPA 开发概念,这些概念都是通用的。但为了让内容更全面、具体,本书将通过 Knockout、Backbone.js 及 AngularJS 这三种不同风格的 MV* 框架来进行比较性讨论,这是本书的一大特色。同时在涉及具体 MV* 框架知识点时,书中会提供相应介绍。书中示例丰富具体,并提供完整源代码下载。

本书适合前端及对 SPA 技术感兴趣的开发者阅读。读者只需掌握 JavaScript、HTML 和 CSS 基本知识,就可以阅读本书。

Original English Language edition published by Manning Publications, USA. Copyright © 2016 by Manning Publications. Simplified Chinese-language edition copyright © 2016 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由 Manning Publications 授予电子工业出版社。未经许可,不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字:01-2016-1162

图书在版编目(CIP)数据

SPA 设计与架构:理解单页面 Web 应用 / (美)埃米顿·A.斯科特(Emmit A. Scott)著;卢俊祥译.
北京:电子工业出版社,2016.11

书名原文:SPA Design and Architecture: Understanding Single-page Web Applications
ISBN 978-7-121-30091-2

I. ①S… II. ①埃… ②卢… III. ①网页制作工具—程序设计 IV. ①TP393.092

中国版本图书馆CIP数据核字(2016)第247063号

策划编辑:张春雨

责任编辑:李云静

印 刷:北京天宇星印刷厂

装 订:北京天宇星印刷厂

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×980 1/16 印张:19 字数:375 千字

版 次:2016 年 11 月第 1 版

印 次:2016 年 11 月第 1 次印刷

定 价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819 faq@phei.com.cn。

译者序

当第一次使用 Gmail 时，我被它那流畅的原生桌面般体验迷倒，赞叹天底下居然有如此令人惊艳的 Web 应用。之后，SPA 的概念逐渐盛行，越来越多关于 SPA 的介绍、实践分享进入开发者的眼帘。然而，在现实开发世界里，我发现很多开发者对 SPA 开发技术的整体概念是模糊的，往往以为只需靠 Ajax 技术，就能很好地实现 SPA。

在我大致浏览了本书的内容后，立刻感受到这是一本非常不错的 SPA 开发书籍。事实上，这本书在 Amazon 网站中获得了非常不错的读者评价。在翻译过程中，我自己也强化了不少 SPA 开发概念，并受益匪浅。

Web 前端开发技术可谓是当今变化最为频繁的软件开发技术，新的开发理念、新的框架层出不穷，同时 ES 6 在语法上带来了诸多变化，这些都迫使我们不断去适应新趋势的发展。但本书着力于 SPA 开发技术的基本原理。掌握了这些基础知识，就能够做到相当程度的以不变应万变，这也是本书吸引我的地方。

岁月如梭，能够沉浸在技术创造的乐趣中是一件让我无限期待的事情。编写出优美的代码，构建出极致的应用，是每个热爱创造的开发者的共同追求目标。真心期待本书能够带给你不一样的收获。

同时，林长瑞、吴桐、朱建宝、周荣华、吴胜华、叶铭辉、李禧强、姚建峰、郑秀玲亦不同程度地参与了本书的翻译工作。

感谢我的妻子和娃，你们给了我很大的支持，小宝贝还给我带来了许许多多的乐趣。同时还要感谢本书的策划编辑张春雨，在你的鼓励下，我的翻译过程充满愉悦。

卢俊祥

2016年10月

本书献给我的三位可爱宝贝：Ana Carolina、David 和 Sofia。

感谢你们所给予的全部微笑、拥抱和无私的爱。

你们永远是爸爸的甜心。

序

1991 年的时候，Tim Berners-Lee 推出了全球首个网站，这个网站运行在一个他命名为“WorldWideWeb”的程序上。两年后，他发布了 WorldWideWeb 的源代码，世界从此发生了天翻地覆的变化。迄今仍可在 info.cern.ch 浏览人类历史上的第一个 Web 页面。

自 1991 年起，Web 得到了空前的发展。在它 24 岁的时候¹，其仍是 IT 世界里使用最广泛的技术。Web 以某种形式运行于各种操作系统、硬件平台及绝大部分的移动设备之上。让这一切成为可能的软件就是万能的 Web 浏览器。

传统上，Web 浏览器是简单的中间人角色。其从服务器端获取数据，展示数据，再将数据发回服务器端，然后又获取更多的数据并展示。而今天的 Web 浏览器，虽仍坚守初心，但其复杂程度已远非当年可预见。

当年简陋的浏览器已经发展成为各式各样应用程序的运行时成熟环境。无须安装，就能随处访问、运行这些应用程序。这就是开发者的“屠龙技”。部署四处运行且即时更新的代码库——这种魅力实在是难以抗拒。还没有其他任何一种技术能够做到如此得意。

站在 Web 平台成功之巅的是无处不在的 JavaScript——一门 10 天内发明出来的语言，它大概是目前世界上使用最广泛的编程语言了。开发者们已经接纳了

¹ 本书著于 2015 年。——译者注

JavaScript，它帮我们打开了崭新应用类型的大门，之前做梦都想不到这一切会发生在 Web 浏览器中。

这些新型应用程序，我们通常称之为单页面应用程序（SPA），几乎完全在浏览器中运行，其引入了一套全新的规则、模式及问题。Web 的广泛吸引力带来了层出不穷的 JavaScript 和 CSS 框架；框架如此之多，以至于要做出合适的挑选犹如大海捞针。

亲爱的读者，这就是本书如此重要的原因。

在过去的 4 年里，我作为一名开发者在 Telerik 工作，我积极倡导对 Kendo UI JavaScript 库的关注。我看到过太多的 JavaScript 框架起起灭灭。当某个流行框架的炒作达到临界状态时，下一个“大事件”¹就随之而来，只留下那些在所谓“时髦”框架之上实际构建解决方案的开发者们仍在收拾残局。这让我总想知道它何时能够消停下来，这样我们就能够专注于“正确之道”，以构建新一代的富客户端应用程序。

残酷的现实却是：做任何事情都不存在所谓的“正确之道”。唯一的办法就是为你的项目和技能栈而战。这是让你更具生产力并最终成功的唯一姿势。

为了在 SPA 开发世界里杀出一条路，理解 SPA 概念之下的基本原理是很有必要的。掌握一门框架还远远不够，因为这样的话，最终仍会让你感到贫瘠和匮乏。深度理解成功 SPA 构筑之法的核心理念，使你得以从容决策，在借助 JavaScript 框架完成了 80% 功能之后，懂得如何构建剩下的 20%。

本书就是你的指导手册——不管你是专家还是新人。在读它的时候，我发现自己在恶补过往不求甚解的基础，并对之前感觉甚好而实际上只是部分理解（甚至理解错误）的术语有了新认识。这些认识和解释埋藏在理论结合实践的字里行间，并在讨论 SPA 框架的时候教你如何构建 SPA、正面处理现实世界需求。

那些我通常持怀疑态度的书籍，都试图解释跟 SPA 一样大的概念，但是这部书籍的亮点却在于——另辟蹊径设法将复杂主题化解为易于理解与消化的部分。

我毫无保留地倾力推荐这本书——每一页，每一句。

Burke Holland

Telerik 开发者关系主管

1 比如发布各种眼花缭乱的、破坏兼容性的新特性。——译者注

前言

我经历过的许多项目，都需要花费一年甚至更多的时间来持续构建。当然，之后还有各种更新和意想不到的事情在等着我们处理。由于这些类型的项目耗时长久，而在这期间，技术的发展突飞猛进；因此，在我准备开始下一个项目的时候，却发现不得不重新评估我的技术栈，因为事物的变化早已今非昔比。

当我和团队在准备最近一次的 SPA 项目时，我萌生了编写这本书的想法。我的主管允许我们探研项目需要的“最佳”技术栈。因此，我们开始评估各种解决方案并创建小型概念验证应用程序。

结束评估工作之后，我对在海量信息中筛选出方向感到心有余悸。对于那些构建 SPA 应用的新手而言，我的艰辛历程同样也是他们要面对的困难。

因此我打算写一本书，不仅会总结构建 SPA 所需知识点，还将介绍构建 SPA 相关的一些第三方库或框架。此外，我希望这本书的内容简单直接并容易消化，同时还能够提供足够的技术细节，争取让读者在读完内容之后，具备实际构建 SPA 应用的能力。

感谢你跟我一起出发。我希望你最终能够发现这本书是一部不可或缺的 SPA 开发指南。

致谢

这是我写的第一本书，因此你可以想象到个中的艰辛。然而我非常幸运，有三个朋友始终在我身后给予写作上的大力相助：Dan Maharry——我的项目编辑；Joel Kotarski——我的技术编辑；还有 Andrew Gibson——我的技术校对。他们是了不起的团队，非常感谢他们！

非常感谢文字编辑 Sharon Wilkey 的细致，还要感谢 Jean-François Morin 在 Java/Spring 方面的技术校对工作。

同时也要感谢 Manning 的其他工作人员，他们让这本书得以顺利出版：Marjan Bace、Michael Stephens、Bert Bates、Maureen Spencer、Kevin Sullivan、Mary Piergies、Candace Gillhoolley、Rebecca Rinehart、Ana Romac、Toni Bowers 和 Linda Recktenwald，以及许多付出汗水帮助我实现目标的无名英雄。

特别的感谢送给 Burke Holland，他撰写了本书序。我是 Burke 的忠实“粉丝”，他能为本书写序让我倍感荣幸。

感谢那些编写过程中的内容审核者：Alain Couniot、Anirudh Prabhu、Bruno Sonnino、David Schmitz、Fernando Monteiro Kobayashi、Johan Pretorius、John Shea、Maqbool Patel、Philippe Vialatte、Rajesh Pillai、Shrinivas Parashar、Trevor Saunders、Viorel Moisei 以及 Yogesh Poojari。

同时感谢我的家人：我的妻子 Rosalba、我的女儿 Ana Carolina 和 Sof、我的儿子 David，以及我的妈妈 Lucy。言语已难以表达我的感激之情——编写过程中你们赋予我如此多的关爱、支持和鼓励。

我的朋友和同事，无论我们是否在一起，我都要感谢你们的鼓励和支持。

关于本书

本书引导你如何创建单页面应用程序。不仅介绍 SPA 构建所需的框架和技术，同时还介绍单元测试、客户端开发与构建任务的自动化。

由于 SPA 构建的过程细节决定着技术栈的选择，因此本书会使用当今主流的 JavaScript 框架来比较多种方式。为什么会存在这么多框架呢？原因之一就是应用构建没有标准方式。通过比较不同框架，可以更好地决定哪种方案更适合你的项目。

本书中的每章都包含了一个完整可行的应用示例。我尽量保持内容的有趣和简单。我并不喜欢那种用一页页源代码填充了半本书的大型示例，这让人读起来太费力了。因此，我决定为每章创建独立的项目。与此同时，尽可能保持每个示例短小精悍，紧扣当前章节的概念主题并不失趣味性。

路线图

第 1 部分：基础知识

- 第 1 章介绍 SPA 整体概念。一开始你将了解到重要概念及 SPA 应用与传统 Web 应用间的差异。我会给出一个清晰简明的定义，并简要说明 SPA 的各个部分是如何结合起来的。
- 第 2 章会继续深入一层，介绍 JavaScript 框架的一个“流派”——MV* 框架，

同时介绍它们在单页面应用程序创建中的作用。本章还会探讨框架之间的共性以及差异性。章末示例使用三种 MV* 框架来分别创建，让你在实践中感受不同的构建风格。

- 在第 3 章中，你将了解模块化编程。这里会通过示例直观了解应该在 SPA 中使用模块的原因。我们还将分解模块模式的语法，并一步步解释它。本章最后介绍模块加载及 AMD 模块的内容。

第 2 部分：核心概念

- 第 4 章快速介绍客户端路由的知识点。你将了解路由的运行机制，以及各种框架如何处理路由。本章也会介绍客户端路由影响应用程序状态的方式。
- 第 5 章介绍 SPA 的布局设计与视图合成。一开始先了解简单设计，然后借助复杂路由逐步深入到复杂设计。此外，我们还将涉及高级的布局主题，如嵌套视图与并列视图。
- 第 6 章讨论模块间通信。如果创建的模块不能相互通信，那还有什么意义呢？！在本章中，你不仅会看到模块间通信的不同方式，也将了解模块化应用设计之道。
- 第 7 章将阐述 SPA 环境中的服务器端处理。尽管客户端仍是我们关注的焦点，但也应该了解 SPA 中的服务器端通信，以及如何处理服务器端调用的返回结果。本章会讨论通过回调函数与 Promise 方式处理返回结果。你还将看到 MV* 框架如何帮助我们完成这些任务。本章最后将简短介绍 REST 及 SPA 应用程序如何调用 RESTful 服务。
- 第 8 章是一个关于 JavaScript 应用程序单元测试的概览，特别针对 SPA 应用。如果你从未接触过客户端单元测试，也不用太过担心。在这里我们将放缓脚步、降低难度，一步步了解相关的基础知识。
- 最后是第 9 章，在这里会讨论客户端任务自动化在开发和构建过程中带给我们的帮助。同时从整体上了解针对各种场景的最常用任务类型，并在本章最后的示例项目中实践这些任务。

附录是章节的补充。附录 A 解析第 2 章示例项目（三个版本）的完整源代码。附录 B 和附录 C 对第 7 章做补充。附录 B 的内容为 XMLHttpRequest API 概览，附录 C 是该章示例项目服务器端调用部分的总结。尽管第 7 章的内容组织也考虑到让你能够使用自己的服务器端语言，但附录 C 仍额外包含了一个 Spring MVC 代码指南，这些代码包含在下载源文件里。附录 D 是关于 Node.js 和 Gulp.js 安装的简单指南，这些内容是实践第 9 章代码所需的。

阅读对象

本书假设你至少具备 JavaScript、HTML 和 CSS 方面的基本知识。如果你有一定的 Web 开发经验，那也是很有帮助的，但这不是必需的。然而，这本书的目标人群是那些在 SPA 方面欠缺经验甚至根本没经验的开发者，以及在构建 SPA 过程中未使用过本书所涵盖技术的开发者。

代码约定与下载

程序清单中的源代码，以及在正文中为了区别于普通文字的代码，都以等宽字体方式呈现。程序清单中还包含代码注解，以突出重点概念。

本书示例源代码可以通过出版社网站下载：<https://www.manning.com/books/spa-design-and-architecture>。

软硬件要求

如果你使用最新的 Mac OS X 或 Windows，以及现代浏览器（如 Firefox、Safari 或 Chrome），应该可以正常运行书中示例。对于特定软件要求，可以在相关章节或附录中找到答案。

由于大多数示例都动态获取 HTML 文件，如果你在本地（而非服务器）运行书中示例，可能需要在浏览器中设定某些权限。请参考下载源文件里的项目 `readme.txt` 文档以获取具体信息。

如果你想运行第 7 章示例，则需要用到 Web 服务器及服务器端语言。每位开发者都有自己的技术栈，因此你可以决定你自己的技术路线。我使用的是 Java 和 Spring MVC，并在书中提供了一个对应的简短设置指南。如果你使用不同的软件，书中也从概念上描述了服务器端调用及相关对象，以便你能够通过你自己的技术栈重建示例应用。

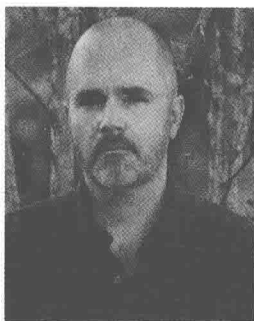
作者在线

购买本书的读者可以免费访问 Manning 出版社维护的专用论坛，在论坛里读者可以评论本书、提出技术问题并得到作者和其他开发者的帮助。要访问论坛并订阅信息请访问 <https://www.manning.com/books/spa-design-and-architecture>。该页面提供了注册后如何访问论坛的指引、各种帮助信息以及论坛行为准则。

Manning 以尽责的态度提供一个读者间、读者与作者间互动的空间。Manning 无法承诺作者的参与程度，其对论坛的贡献基于自愿而免费原则。我们建议你尽量向作者提一些富有挑战性的问题，以保持作者的热情！

只要书籍得以出版，你就可以通过出版社网站访问作者在线论坛以及早期的讨论归档内容。

关于作者



Emmitt A. Scott 是一名有 17 年 Web 应用构建经验的高级软件工程师和架构师。他为教育、银行和通信领域开发过大型应用程序。他的爱好包括阅读（特别喜欢 Jim Butcher 的小说）、吉他（想当年他可是一位摇滚乐手）以及尽可能多陪陪孩子。

译者简介

卢俊祥

译者，书迷；关注 Web 技术趋势，热衷 App 开发、Web 开发、数据分析、架构设计以及各类编程语言；陈氏太极拳五十六式爱好者；佛禅人生，缘散缘聚。

微博：@2gua

个人网站：<http://www.2gua.info/>

知乎专栏：<https://zhuanlan.zhihu.com/guagua/>

关于封面图画

本书封面图画为“1700 年年轻土耳其贵族服饰”。这幅图出自 Thomas Jefferys 的《古今各国服饰集》(*A Collection of the Dresses of Different Nations, Ancient and Modern*)——1757 至 1772 年间出版于伦敦，扉页标明这些画是手工上色的铜版雕刻印刷品，用阿拉伯树胶加固。Thomas Jefferys (1719—1771)，被称为“乔治三世的地理学家”。他是一位英国制图大师，乃当时顶尖的地图制作大师。他为官家镌刻及印制地图，制作了大量商业的地图和地图集，尤其是北美地域的作品。Thomas Jefferys 作为一名地图制作大师，其调查及绘制之地的服饰也引起了他的兴趣，而成果就出色地体现在了这四卷册中。

向往远方而游历，在 18 世纪后期还是新景象，当时这样的卷册极受欢迎，其向远游者和“神游旅行者”介绍了不同国家的居民。Jefferys 卷册里丰富的画面生动地讲述了大约 200 年前世界各国的独特魅力。自那时起着装发生了变化，同时那个年代丰富的地域多样性也逐渐褪去。现在要分辨出不同大陆的居民已非常困难。也许应该尽量乐观地看待这种现象，我们失去文化及视觉多样性的同时，却换来了多样化的个人生活、更加多样有趣的知识和科技生活。

正当计算机书籍内容的同质化现象日益突出的今天，Manning 借助 Jeffreys 的图画将昨日重现，参考两个世纪前地域生活的丰富多样性来设计图书封面，谨表达对计算机领域开创精神的赞誉。

目录

第 1 部分 基础知识	1
1 单页面应用程序介绍	3
1.1 SPA 简述	4
1.1.1 无须刷新浏览器	7
1.1.2 表现逻辑位于客户端	7
1.1.3 服务器端事务处理	7
1.2 更进一步	8
1.2.1 以 Shell 页面开始	8
1.2.2 从传统页面到视图	9
1.2.3 视图的产生	10
1.2.4 实现无刷新的视图切换	11
1.2.5 贯穿动态更新过程的流畅性	12
1.3 SPA 应用相较传统 Web 应用的优势	12
1.4 温故知新	13
1.5 优秀 SPA 应用的构成	15
1.5.1 组织项目	15

1.5.2	创建可维护的松耦合 UI	17
1.5.3	使用 JavaScript 模块	18
1.5.4	执行 SPA 导航	19
1.5.5	创建视图组成与布局	19
1.5.6	模块通信	20
1.5.7	与服务器端通信	20
1.5.8	执行单元测试	20
1.5.9	客户端自动化技术	20
1.6	小结	21
2	MV* 框架介绍	22
2.1	MV* 概念	24
2.1.1	传统 UI 设计模式	25
2.1.2	MV* 和浏览器环境	27
2.2	MV* 基础概念	28
2.2.1	框架	29
2.2.2	我们的 MV* 项目	30
2.2.3	模型	32
2.2.4	绑定	36
2.2.5	模板	40
2.2.6	视图	44
2.3	为什么要用 MV* 框架	44
2.3.1	关注分离	45
2.3.2	简化日常任务	46
2.3.3	提升生产率	47
2.3.4	标准化	47
2.3.5	可扩展性	48
2.4	框架选择	48
2.5	挑战环节	50
2.6	小结	50
3	JavaScript 模块化	52
3.1	模块概念	53
3.1.1	模块模式概念	53

3.1.2 模块结构	54
3.1.3 揭示模式	55
3.2 模块化编程的意义	56
3.2.1 避免命名冲突	56
3.2.2 保护代码完整性	65
3.2.3 隐藏复杂性	67
3.2.4 降低代码改变带来的冲击	68
3.2.5 代码组织	68
3.2.6 模块模式的不足	69
3.3 模块模式剖析	69
3.3.1 可访问性控制	69
3.3.2 创建公有 API	70
3.3.3 允许全局导入	73
3.3.4 创建模块的命名空间	73
3.4 模块加载及依赖管理	74
3.4.1 脚本加载器	74
3.4.2 异步模块定义——AMD	75
3.4.3 通过 RequireJS 实践 AMD	76
3.5 挑战环节	81
3.6 小结	81

第 2 部分 核心概念 83

4 单页面导航 85

4.1 客户端路由器概念	86
4.1.1 传统导航	86
4.1.2 SPA 导航	86
4.2 路由及其配置	88
4.2.1 路由语法	90
4.2.2 路由配置项	90
4.2.3 路由参数	91
4.2.4 缺省路由	93
4.3 客户端路由器的工作机制	93
4.3.1 片段标识符方式	94

4.3.2 HTML5 历史 API 方式	95
4.3.3 使用 HTML5 历史 API 方式	97
4.4 综合实作：实现 SPA 路由	98
4.4.1 教员列表（缺省路由）.....	99
4.4.2 主要联系人路由	101
4.4.3 教员授课时间（参数化路由）.....	102
4.5 挑战环节	104
4.6 小结	105
5 视图合成与布局.....	106
5.1 项目介绍	107
5.2 布局设计概念	108
5.2.1 视图	108
5.2.2 Region.....	109
5.2.3 视图合成	110
5.2.4 嵌套视图	111
5.2.5 路由	112
5.3 高级合成与布局的可选方案	113
5.3.1 优点	113
5.3.2 缺点	114
5.4 设计应用程序	114
5.4.1 设计基本布局	115
5.4.2 设计基本内容	117
5.4.3 在复杂设计中应用视图管理	122
5.4.4 通过自身状态创建嵌套视图	125
5.5 挑战环节	127
5.6 小结	128
6 模块间交互.....	129
6.1 模块概念回顾	131
6.1.1 用模块封装代码	131
6.1.2 API 提供对内部功能的访问控制	133
6.1.3 SRP——以单一目的作为设计出发点	134
6.1.4 代码重用——控制项目规模	135

6.2 模块间交互方式	136
6.2.1 通过依赖进行模块间交互	136
6.2.2 依赖方式的优缺点	138
6.2.3 通过发布 / 订阅模式进行模块间交互	138
6.2.4 发布 / 订阅模式优缺点	141
6.3 示例项目细节	142
6.3.1 搜索功能	144
6.3.2 显示产品信息	150
6.4 挑战环节	155
6.5 小结	155

7 与服务器端通信

156

7.1 示例项目新要求	157
7.2 与服务器端通信综述	158
7.2.1 选择数据类型	158
7.2.2 HTTP 请求方法	159
7.2.3 数据转换	160
7.3 使用 MV* 框架	161
7.3.1 请求生成	162
7.3.2 通过回调函数处理结果	165
7.3.3 通过 Promise 处理结果	166
7.3.4 Promise 错误处理	170
7.4 RESTful Web 服务调用	172
7.4.1 什么是 REST	172
7.4.2 REST 原则	172
7.4.3 MV* 框架的 RESTful 支持	174
7.5 示例项目细节	174
7.5.1 配置 REST 调用	174
7.5.2 添加产品到购物车	177
7.5.3 查看购物车	179
7.5.4 修改购物车	181
7.5.5 从购物车中移除产品	183
7.6 挑战环节	184
7.7 小结	184

8	单元测试	186
8.1	示例项目说明	187
8.2	什么是单元测试	187
8.2.1	单元测试的好处	188
8.2.2	构建更好的单元测试	189
8.3	传统的单元测试	192
8.3.1	QUnit 起步	193
8.3.2	创建第一个单元测试	196
8.3.3	测试由 MV* 对象创建的代码	200
8.3.4	测试对 DOM 所做的改变	205
8.3.5	混合使用其他测试框架	206
8.4	挑战环节	208
8.5	小结	208
9	客户端任务自动化	209
9.1	Task Runner 的常见用途	210
9.1.1	即时刷新浏览器	210
9.1.2	自动化 JavaScript 和 CSS 的预处理过程	211
9.1.3	自动化 Linter 代码分析	211
9.1.4	持续单元测试	211
9.1.5	文件串接	212
9.1.6	代码压缩	212
9.1.7	持续集成	212
9.2	Task Runner 选择	212
9.3	本章示例项目	213
9.3.1	Gulp.js 介绍	214
9.3.2	创建第一个任务	215
9.3.3	创建代码分析任务	216
9.3.4	创建浏览器刷新任务	218
9.3.5	自动化单元测试	220
9.3.6	创建构建过程	222
9.4	挑战环节	227
9.5	小结	227

A	员工通讯录示例说明	229
B	XMLHttpRequest API.....	259
C	第 7 章内容的服务器端设置与总结	266
D	安装 Node.js 与 Gulp.js.....	277

第1部分

基础知识

在开发第一个单页面 Web 应用程序（Single-page Web Application, SPA）之前，本部分将介绍一些相关的基础概念。

在第 1 章，我们通过非常清晰的专业描述来讨论什么是 SPA。搞清楚这种架构涉及哪些内容及为什么选择这种架构而非传统 Web 应用程序架构是很重要的。

在单页面应用开发中，程序代码的整洁性及可维护性很关键。第 2 章比较各种 JavaScript 框架的不同风格，并介绍对这些框架产生深远影响的三种架构模式：MVC、MVP 以及 MVVM。同时还将讨论即使是相同应用程序，也需因应框架的不同实现风格而做出变化（采取不同实现方式）的必要性。

第 3 章快速介绍模块模式（Module Pattern），以及它如何改变 JavaScript 代码组织的思维方式。通过该模式，能够像往常一样创建函数和变量，但却是在某种结构的舒适范围内，该结构模仿其他语言的经典封装方式。在本章你会发现，模块化编程对一个成功的 SPA 应用而言是至关重要的。

单页面应用程序介绍

本章内容

- 单页面应用程序（SPA）的定义
- SPA 基本元素概览
- SPA 相对于传统 Web 应用程序的优势

Web 开发者始终有一种追求，希望开发出的 Web 应用具有原生桌面应用那样的界面效果。事实上也存在着几种仿原生体验的开发技术，诸如 IFrame、Java Applet 以及 Adobe Flash、Microsoft Silverlight 等，它们也都获得了不同程度的成功。这些技术虽然实现不同，但它们至少都有一个共同目标：将桌面应用的强大能力带到跨平台、瘦客户端的 Web 浏览器环境中。单页面 Web 应用程序，或简称 SPA，亦以此为目标，但它不需要额外的浏览器插件，也不需要学习一门新的语言。仅仅借助 JavaScript、HTML 和层叠样式表（CSS）技术就能实现原生体验效果的想法，实在是吸引人，但 SPA 蕴含着怎样的魔力？这种想法打哪儿来？

镜头定格到本世纪初。AJAX 技术的逐步流行催生了一种全新的 Web 页面设计理念。其起源于一个有趣而又难以理解的技术——微软 IE 浏览器里的 ActiveX 控件，ActiveX 控件用于异步发送和接收数据。当各主要浏览器厂商将 ActiveX 控件具备

的功能以 *XMLHttpRequest (XHR) API* 的方式予以正式采纳时，最初的简陋最终带来了一场革新。

着手使用 JavaScript、HTML 和 CSS 整合该 API 的开发者取得了显著成效。这些技术的融合最后演进成了著名的 *AJAX* 技术（或称异步 JavaScript 和 XML）。AJAX 的不唐突式数据请求，结合 JavaScript 动态更新文档对象模型（Document Object Model, DOM）以及 CSS 实时改变页面样式的能力，使得 AJAX 成为现代 Web 开发中的前沿技术。

借力 AJAX 的成功推进，SPA 概念通过把 AJAX 处理技术扩展到页面层级，将 Web 开发技术提升到了一个新的水平。此外，创建 SPA 应用时常用的模式及实践，可以整体提升应用设计、代码维护和开发时间等方面的效率。而理解 SPA 架构，对成功创建单页面应用程序而言意义非凡。

与大多数时新解决方案一样，单页面应用程序设计也有各种招式。当今专家们的不同观点，加上众多相互竞争的第三方库和框架，对选择正确的 SPA 项目解决方案造成了不少挑战。懂得越多，就越有把握找到满足开发目标的正确实现，因此，我将首先对 SPA 及其优势进行清晰阐述。贯穿全书，你将通过一种被称为 MV* 框架的 JavaScript 框架风格，来实践 SPA 开发的方方面面。

MV* 并非一切

本书讨论的 SPA 仅限于使用 MV* 框架（你将在第 2 章了解更多相关概念）。开宗明义地指明这一点很重要，因为创建 SPA 还有其他方式，如使用 React (<https://facebook.github.io/react>) 或者 Web 组件 (Web Component, W3C 规范，组件式 Web 开发技术的一系列标准)。

1.1 SPA 简述

在 SPA 应用里，整个应用作为单个 Web 页面运行。在这种方式下，应用的表现层从服务器端脱离出来，并在浏览器端管理。为了更形象地了解 SPA 思想，我们用几张图示来说明。

首先来看一下非 SPA 式的 Web 应用程序。图 1.1 展示了一个大型 Web 应用程序，其使用了传统的服务器端设计理念。

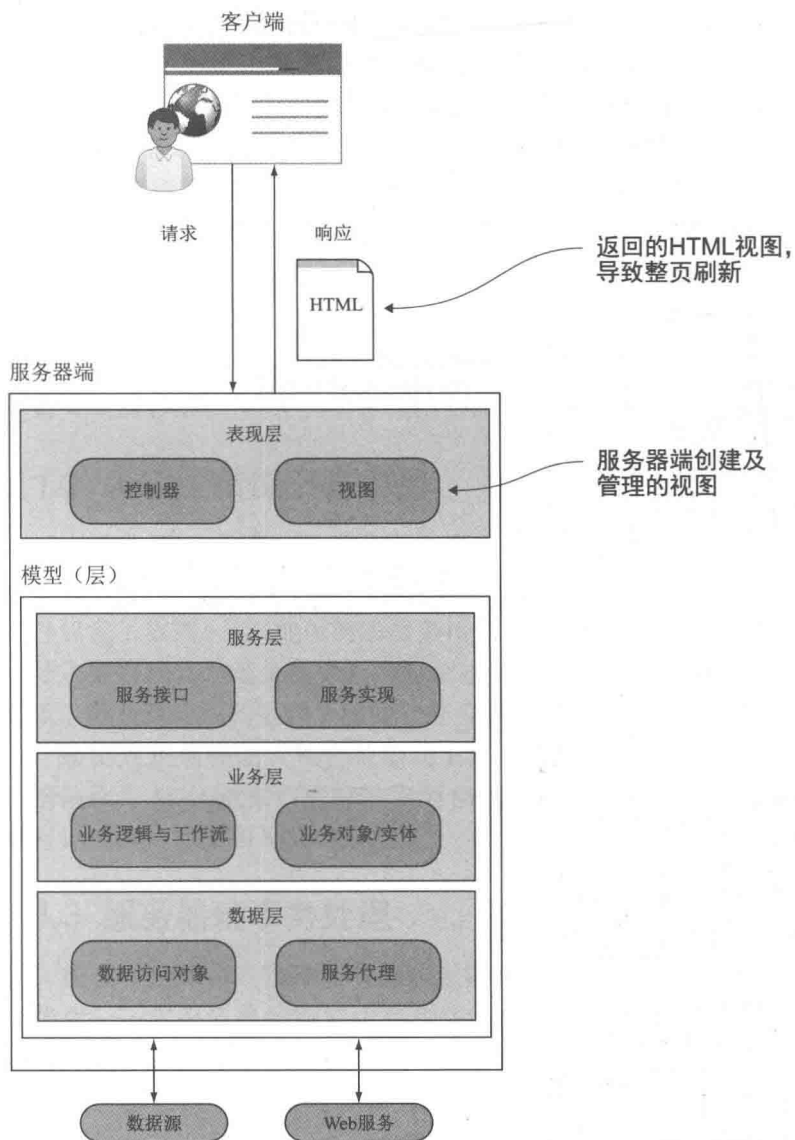


图 1.1 在传统 Web 应用程序中, 每个新视图 (HTML 页面) 都在服务器端构建

在图示的设计中, 每个新视图 (HTML 页面) 请求都会导致对服务器端的双向访问。当客户端需要新的数据时, 则会向服务器端发送请求。在服务器端, 请求由表现层的某个控制器对象拦截。然后该控制器通过服务层与模型层交互, 服务层决定完成模型层任务所需的组件。通过数据访问对象 (DAO) 或服务代理获取数据之后, 所有必要的数据库更新都将由业务层的业务逻辑产生。

控制传回到表现层, 在这里选择合适的视图。展示逻辑规定新获取数据在选中

视图中如何展示。通常情况下，结果视图是一个包含占位符的源文件，数据（及其他可能的渲染指令）将插入到占位符中。每当控制器将请求路由至视图时，该文件表现得就像某种类型的模板，以让视图设置好占位符的数据。

数据与视图整合好之后，视图返回给浏览器。然后浏览器接收新的 HTML 页面，并通过界面刷新，将包含请求数据的新视图展示给用户。

图 1.2 演示了 SPA 应用的设计风格。请注意表现层和事务处理的变化。

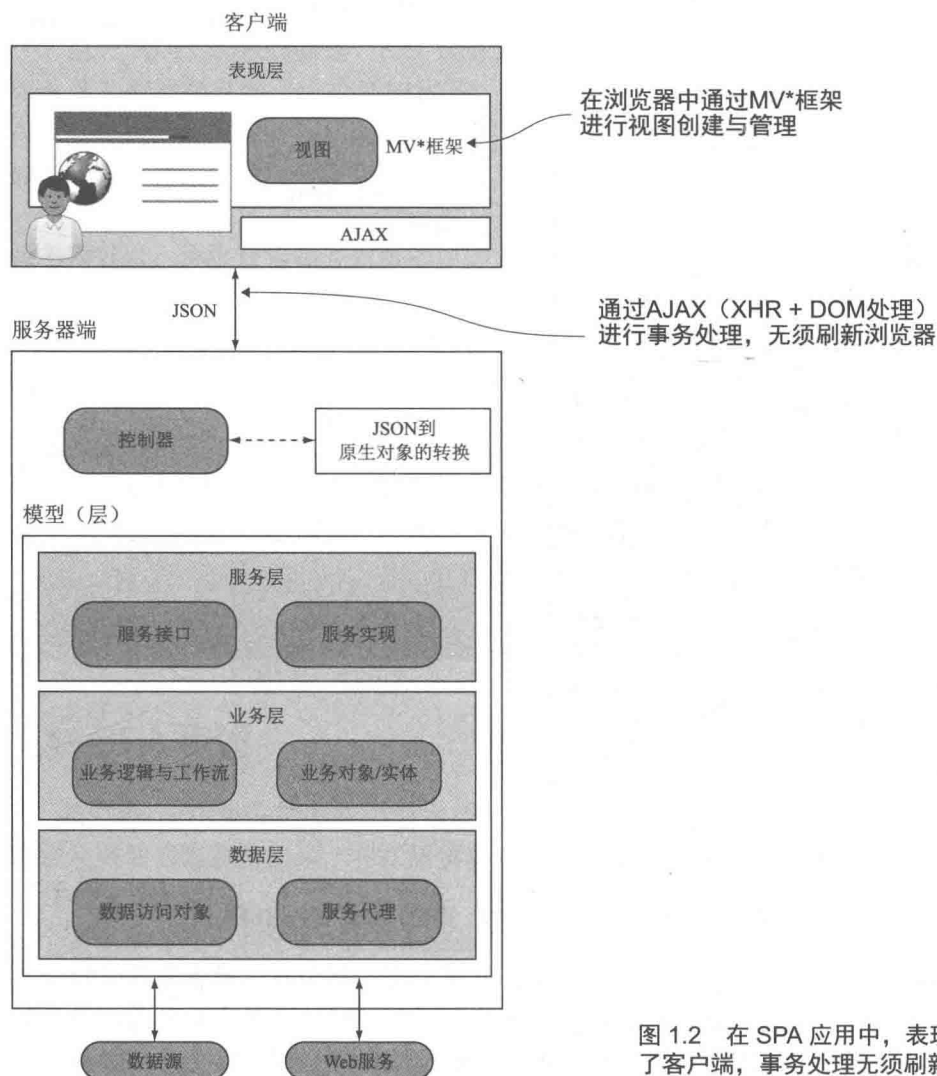


图 1.2 在 SPA 应用中，表现层移到了客户端，事务处理无须刷新浏览器

将视图的创建与管理从服务器端解耦出来，并移到 UI 层。从架构角度而言，这种改变赋予 SPA 应用一个有趣的优势。除非在服务器端需要处理部分渲染，否则

服务器端将不会再收到数据呈现方面的请求。

SPA 整体设计与传统 Web 应用设计相似。而 SPA 关键的不同点在于：没有整页刷新、表现逻辑位于客户端，以及取决于数据渲染的偏好，服务器端的事务处理可以只涉及数据。

1.1.1 无须刷新浏览器

在 SPA 应用中，视图并非完整的 HTML 页面。它们仅仅是构成屏幕可视区域的 DOM 的一部分。初始页面加载之后，所有创建和显示视图所需的内容将下载并准备就绪。如果后续需要新的视图，视图将在浏览器本地生成，并通过 JavaScript 动态关联到 DOM。SPA 应用无须刷新浏览器。

1.1.2 表现逻辑位于客户端

由于在 SPA 应用中表现逻辑主要集中在客户端，因此，整合 HTML 与数据的任务也就从服务器端移到了浏览器端。如同传统 Web 应用中服务器端的 HTML 源文件包含了数据（及其他可能的渲染指令）待插入的占位符，客户端模板是客户端（通过插入数据或其他渲染指令）产生新视图的基本手段，但它不是完整页面的 HTML 模板，其只针对视图呈现页面的一部分。

路由到正确视图、整合数据与 HTML 模板、管理视图生命周期等重任通常委托给被称为 MV* 框架（有时被称为 SPA 框架）的第三方 JavaScript 文件处理。第 2 章将详细阐述模板和 MV* 框架。

1.1.3 服务器端事务处理

在 SPA 应用中，渲染服务器端返回数据的方式有好几种，其中包括服务器端部分渲染——在服务器端响应中包含了已整合数据的 HTML 代码片段。本书聚焦另一种方式——渲染在客户端完成，而在业务处理时发送与接收操作只涉及数据。这种方式总是通过 XHR API 异步达成。我们将采用的数据交换格式是典型的 JSON（JavaScript Object Notation），尽管这不是必需的。此外，虽说我们采用了客户端渲染的方式，但服务器端在 SPA 应用中仍扮演着相当重要的角色，第 7 章将对服务器端的角色作用进行详细介绍。

即使你已经使用了某个服务器端设计模式如 MVC（模型 - 视图 - 控制器，Model-View-Controller）来分离视图、数据以及逻辑，重新配置你的 MVC 框架以满足 SPA 应用的需要也是一件相对容易的事情。因此，像 ASP.NET MVC 或 Spring MVC 这样的服务器端框架仍能与 SPA 应用相得益彰。

1.2 更进一步

你已经大致了解了 SPA 整体概念，现在进一步剖析它。我们接下来的讨论已移至浏览器端的表现层机制。由于后续章节会提供更为详细的内容，因此这里只做总体介绍。

1.2.1 以 Shell 页面开始

单页面应用程序的“单页面”（single-page）指的是初始 HTML 页面，或被称为 Shell（外壳页面）。这个 HTML 页面加载且仅加载一次，其充当应用程序其余部分的起始点。在 SPA 应用中，这是唯一全页面加载的时机。应用后续部分的加载将动态并独立于 Shell 页面进行，无须全页面加载，不让用户感受到页面的刷新。

典型地，Shell 页面在结构上保持最小化，并通常包含一个空 `<div>` 标签，该标签容纳应用程序其余内容（如图 1.3 所示）。你可以将 Shell 页面文件看作母舰，而这个空 `<div>` 标签作为初始容器，就相当于停泊甲板。

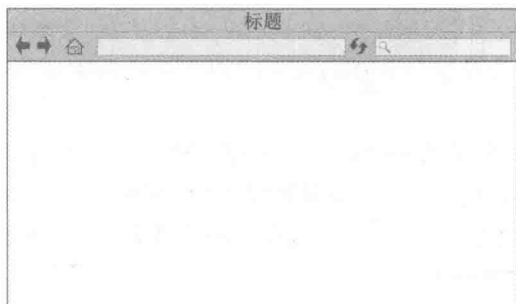
Shell 页面代码具有一些传统 Web 页面的初始基本元素，如 `<head>` 和 `<body>`。清单 1.1 演示了一个基本的 Shell 页面代码。

清单 1.1 SPA 的 Shell 页面示例

```
<!DOCTYPE html>
<html>
<head>
  <title>Shell Example</title>
  <link rel="stylesheet"
        type="text/css"
        href="app/css/default.css">
</head>
<body>
  <div id="container"></div>
</body>
</html>
```

加载应用程序的样式表

初始 `<div>` 容器标签



Shell 页面刚开始空空如也

图 1.3 HTML Shell 页面是开始结构。其当前尚无内容，只有一个空的 `<div>` 标签

如果应用程序的可视化区域划分为几部分，则初始 `<div>` 容器标签可以包含子容器。子容器通常被称为 **Region**（区域）¹，因为它们用于从视觉上将屏幕划分为几个逻辑区块（如图 1.4 所示）。

Region 有助于将可视区域划分为可管理的几块内容。**Region** 的 `<div>` 容器就是通知 MV* 框架插入动态内容的地方。然而，还需要认识到本书以外的框架有可能采用其他范式。例如 **React**，它使用 DOM 修补（*patching*）的方式，而非替换特定 **Region**。



图 1.4 Shell 的子容器被称为 **Region**。**Region** 的内容由视图提供

1.2.2 从传统页面到视图

SPA 应用程序的“页面”其实不是页面，至少不是传统概念上的页面。当用户进行导航操作时，屏幕所呈现貌似页面的部分实际上是应用程序内容的独立部分，称为视图。第 2 章将详细阐述视图。现在只需要知道视图是应用程序的一部分，它是终端用户所见并与之交互的部分。

想象出一般 Web 页面与 SPA 视图间的差异是困难的。为了帮助你形象地了解这些差异，请参考图 1.5 所示，其展示了一个由两个 Web 页面组成的简单 Web 站点。如你所见，该传统站点的所有 Web 页面涵盖了完整的 HTML 结构，包括 `<head>` 和 `<body>` 标签。

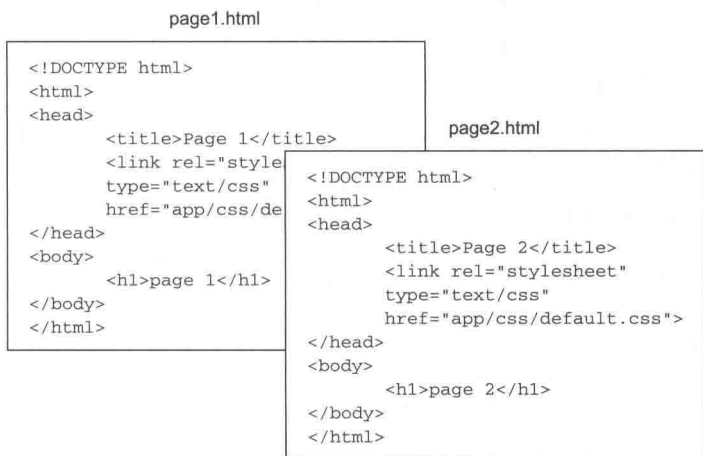


图 1.5 在传统站点的设计中，每个 HTML 文件都是一个完整的 HTML 页面，涵盖了完整的 HTML 结构

1 为了更突出 **Region** 的概念性，本书保留 **Region** 英文描述。——译者注

图 1.6 展示了一个相同内容的 SPA 站点。SPA 的“页面”只是 HTML 代码片段。在 SPA 应用中，屏幕可视区域的内容发生改变的場景，就如同传统站点中页面发生变化那样。

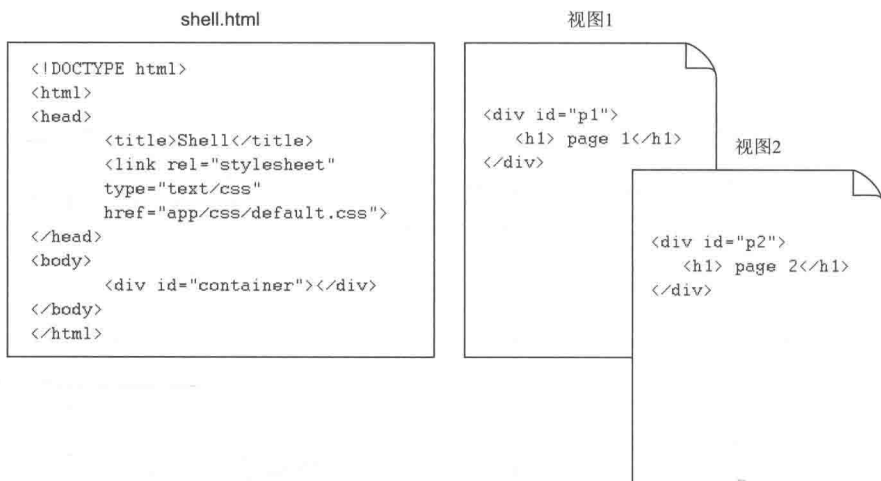


图 1.6 在 SPA 应用设计中，一个完整的 HTML 文件包含占位符，占位符对应存储在视图文件中的 HTML 代码片段

当 SPA 应用程序启动时，MV* 框架会插入视图 1。当用户导航到新“页面”时，框架就会将视图 1 切换到视图 2。第 4 章将详细描述 SPA 导航。

1.2.3 视图的产生

如果应用程序的某部分（或视图）并非初始 Shell 页面的一部分，那么它们要如何才能成为应用程序的一部分呢？如前面提及，通常情况下，作为用户导航的结果，SPA 各个部分按需展示。其中每个部分的 HTML 骨骼构造被称为模板(template)，模板包含数据占位符。开发者通过 JavaScript 的第三方库或框架（这些框架通常被称为 MV* 框架）来绑定数据与模板（可以是一个或多个模板）。图 1.7 展示了最终视图中的数据绑定结果。所有不在 Shell 页面中的屏幕内容都放入单独视图中。



图 1.7 视图是数据与一个或多个模板的绑定结合体

整个视图都与 DOM 关联,根据实际情况,视图要么直接位于初始 `<div>` 容器中,要么位于某个 Region (初始 `<div>` 容器的子容器,如果存在的话)中,如图 1.8 所示。

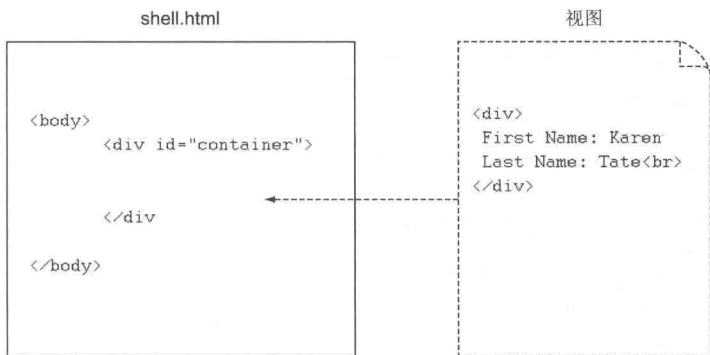


图 1.8 视图动态关联 DOM。通常,作为用户导航的结果,视图位于初始 `<div>` 容器或某个 Region (初始 `<div>` 容器的子容器)中

1.2.4 实现无刷新的视图切换

所有的视图切换都不用刷新 Shell 页面。因此,与传统 Web 应用中每次导航请求都会得到一个新的静态页面不同,SPA 可以显示新内容而不会让用户感受到中断。对于屏幕的特定部分,一个视图的内容仅仅只是被另一个视图的内容所替代。图 1.9 描述了当用户导航时页面本身的改变情况。导航时无须重新加载页面是单页面应用程序的关键特性,其给了用户原生应用般的体验。

MV* 库/框架

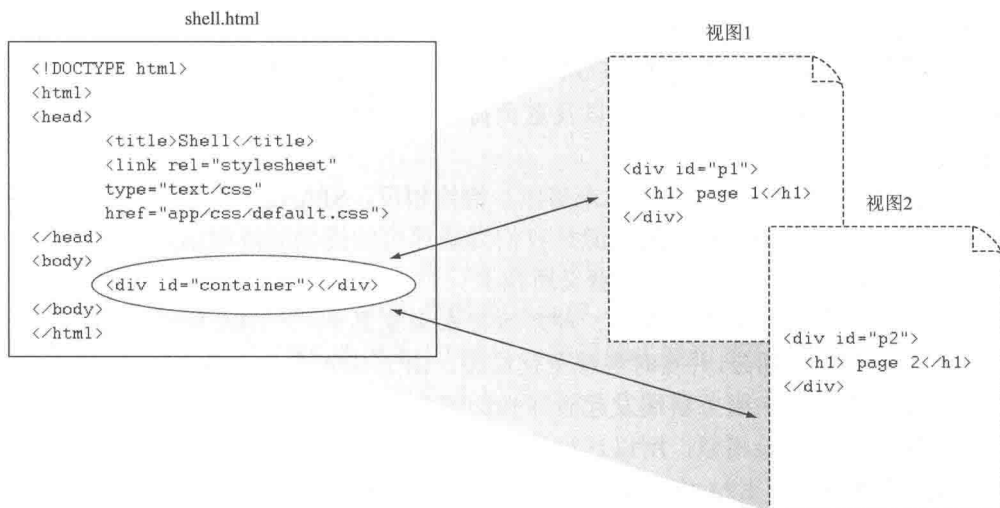


图 1.9 针对屏幕给定区域,SPA 视图通过 DOM 操作进行无缝切换,让用户获得更佳的原生应用体验

在 SPA 应用中，当导航发生时的一件有趣的事情是，于用户而言页面看起来发生了变化，而 URL 却未有变化，甚至后退按钮仍可以用来将用户引导到之前的“页面”。

请记住，在客户端创建及管理视图的任务是由 MV* 框架负责的。在第 2 章，我们将仔细分析任务的各个环节以获得一个更加清晰的理解。

1.2.5 贯穿动态更新过程的流畅性

SPA 的另一个重要方面是异步获取服务器端数据并动态插入到应用。因此，不仅导航时不会重新加载页面，而且在请求及获取服务器端数据时也不会重新加载页面。这同样是一个原生应用的体验效果。AJAX 技术使得这一切成为可能。本章开始时讨论到 Web 开发技术的历史演进，以及 AJAX 在 SPA 应用开发中举足轻重的角色定位。所以，在这里不再对 AJAX 进行赘述。

前面我们花了较多篇幅解释页面或视图在导航期间如何动态切换。服务器端或缓存里的业务数据也能够以相同方式添加或移除。数据获取在程序后台默默进行，并可以伴着其他数据请求并行发生。获取数据之后，数据与 HTML 模板进行绑定，视图实时更新。正确更新页面却让用户察觉不到丝毫闪烁的绝活儿，赋予应用程序一定的流畅性与平滑度，而这恰恰是传统 Web 应用程序无法提供的。第 7 章将涉及更为详细的数据访问相关内容。

1.3 SPA应用相较传统Web应用的优势

Web 浏览器仍是分发软件产品的重要途径，因为它“瘦”、无处不在，而且还是一个标准化环境，终端用户都有浏览器。同时，Web 浏览器对软件更新来说很重要，因为软件更新在服务器端进行，用户不用担心安装过程。不幸的是，各种冲突、整页刷新、每次请求的内容重复，以及重负荷的事务处理都在侵蚀着浏览器传递数据功能的优势。

然而基于 Web 的交互方式还远未落伍。恰恰相反，SPA 技术正处于用户体验变革的最前沿。单页面应用概念的起源是我们渴望尽可能提供给终端用户一个最佳体验。以下是采用单页面应用架构的意义所在：

- 桌面应用程序般的呈现效果，却又运行在浏览器中——SPA 应用能够动态重绘屏幕的某个部分，并实时展现变化结果。由于 SPA 预先下载 Web 页面结构，因此就不需要向服务器端发起破坏性的请求。这种体验与用户通过原生桌面应用获得的体验相似，所以这种体验“感觉”会更自然。而比桌面应用程序更具优势的是，SPA 应用运行在浏览器中，它兼顾了两者的优点：原生应用的体验效果，却又基于浏览器环境。

- 表现层解耦——如前所述，UI 的呈现及行为处理代码在客户端而非服务器端。该特性使得服务器端与客户端尽可能解耦。解耦的好处是服务器端与客户端可以分别独立管理和更新。
- 更快而轻量级的事务处理负荷——服务器端的事务处理变得更加轻量 and 快速，因为应用初始分发之后，客户端与服务器端的交互只有数据的发送与接收。传统 Web 应用有响应下一个页面内容的额外开销。由于所有页面都重新渲染，因此传统 Web 应用返回的内容还包括 HTML 标记。异步、纯数据的事务处理使得 SPA 架构的运行速度非常快。
- 更少的用户等待时间——在今天以 Web 为中心的世界里，更少的页面加载等待时间，也就意味着用户更愿意在网站多做停留并更可能成为回头客。由于 SPA 预先加载 Shell 页面及少量的支持文件，然后在用户导航时进行动态构建，因此应用启动速度感觉起来就非常快。如前面提到的，屏幕渲染更加流畅、平滑，事务处理更加轻量、快速，这些特性都使得用户等待时间大大降低。性能方面的考量并非无足轻重，涉及在线交易时，性能就相当于金钱。*Web Performance Today*¹ 上发表的沃尔玛（Walmart）研究成果指出了每 100 毫秒的性能提升，增收就多 1%，这对沃尔玛来讲非常可观。
- 更简单的代码维护——软件开发者总在寻找开发与管理代码基的更好方式。传统 Web 应用程序有点像狂野的西部，HTML、JavaScript 以及 CSS 可以交织出代码维护的梦魇。服务器端代码混入 HTML 代码（想想看 ASP 和 JSP 程序吧）的实现方式让你捧着一个烫手的山芋。随着你看到后续章节，MV* 框架（如本书提及的框架）将帮助我们吧代码划分为不同的关注层面。JavaScript 代码有其所属位置——与 HTML 代码分开并放在不同的单元。借助第三方库与框架（例如 Knockout、Backbone.js 与 AngularJS），可以单独管理屏幕某块区域的 HTML 结构及该结构对应的数据，同时客户端与服务器端的耦合程度也大大降低。

1.4 温故知新

跟创建传统 Web 应用一样，我们使用相同的技术来开发单页面 Web 应用程序：HTML、CSS 和 JavaScript。不需要浏览器插件，也没有什么神奇的 SPA 专属语言需要掌握。HTML 和 CSS 继续是 UI 结构和布局的主要构建工具，JavaScript 则仍然是交互及 UI 逻辑处理的基石（如图 1.10 所示）。

¹ www.webperformancetoday.com/2012/02/28/4-awesome-slides-showing-how-page-speed-correlates-to-business-metrics-at-walmart-com

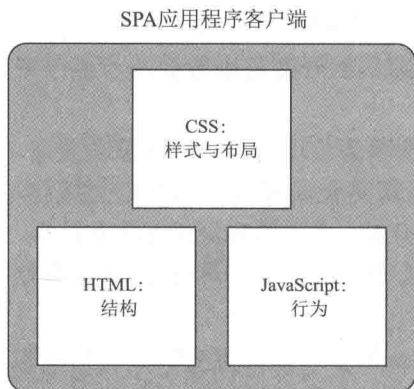


图 1.10 CSS、HTML 及 JavaScript 是单页面应用程序的构建技术栈。不需要掌握新的语言，也不需要安装浏览器插件

使用 SPA 架构之后，对用户来说区别就在于应用程序的体验不一样了。导航时的体验更接近原生桌面应用程序，感觉更平滑、更舒适。而对于开发者而言，不同点则是在单个 HTML 页面中创建应用程序功能，你需要重新检视常规的 Web 开发方式。

如前面所述，SPA 应用内容划分为数个独立区块或视图。因此开发者不用再创建一个完整页面，通常这些页面重复包含了一些诸如页头或主菜单之类的通用元素。在 SPA 程序中，即使是通用区块也是视图。你得放弃以往逐个页面考虑布局的思路，并着手根据屏幕可用区域的视图位置来思考问题。事实证明，一旦你理解，这种思考问题的方式并不难。全局性布局区域如主菜单，从用户体验角度来说是固定保留的。屏幕的公共区域如中心内容区，用户进行导航操作时，应用程序重用它们来切换各种视图（以及所有 Region）。

然而对于终端用户，SPA 应用可以看起来跟传统 Web 应用一模一样。如图 1.11 所示，SPA 应用程序有页头、侧边栏，或者任何其他经典的 Web 页面元素。

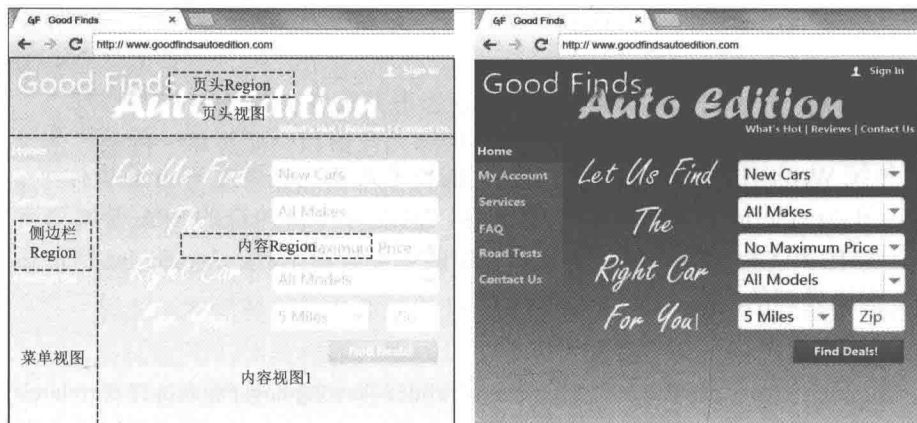


图 1.11 使用 Region 放置 SPA 应用视图，使得 SPA 应用与传统 Web 页面看起来一样

在 JavaScript 侧，可以像往常一样编码，但有一个较大区别。由于要处理无刷新的单页面，因此，针对变量与函数的简单全局作用域则无法满足现实所需。我们得将代码划分为可行的数个单元，并在被称为模块（Module）的函数中安置这些代码，模块具有属于自己的作用域。这种处理方式让你绕开了不得不在全局命名空间中创建所有变量和函数的局限。

SPA 应用中客户端与服务器端通信的方式是 AJAX。尽管“AJAX”的术语暗示到 XML，但绝大多数用到 AJAX 技术的现代 SPA 应用却是使用 JSON 作为首选数据交换格式。JSON 于 SPA 应用而言是理想的格式，因为它轻量而简洁，语法也很适合用于描述对象结构。而 AJAX 对大部分开发者来说并不是什么新鲜事儿，即使是传统 Web 应用程序，通常情况下或多或少也会用到一些 AJAX 技术。

我们的整体设计坚持这样的原则：保持所有 SPA 应用代码的易维护性，各个关注层面相互解耦。但不要担心会产生任何额外的复杂度。一旦掌握了模块模式的独特语法，你的开发生涯将变得更轻松。本书后面会详细介绍模块化编程，并在各个示例中用到各种模块设计模式。因此，别被吓着了——本书将大量涉及模块模式的内容，它会成为你的第二习性的！

1.5 优秀SPA应用的构成

如果你在阅读本书之前调查过单页面应用程序，就有可能在选择上感到不知所措。如你目前了解到的，SPA 应用采用的开发技术不止一种，而是多种技术协作，最终构建出产品。能够采用的库和框架，与该选哪种方式的争论一样多。因此必须承认，试图找出不仅能够良好协作，而且还能满足项目需要及团队喜好的技术栈，真是一件难事。

好消息是，有一个疯狂的方法。如果把单页面应用程序概念看作一个整体，可以将其划分为一系列类型，这些类型适合任何风格的选择方案。

1.5.1 组织项目

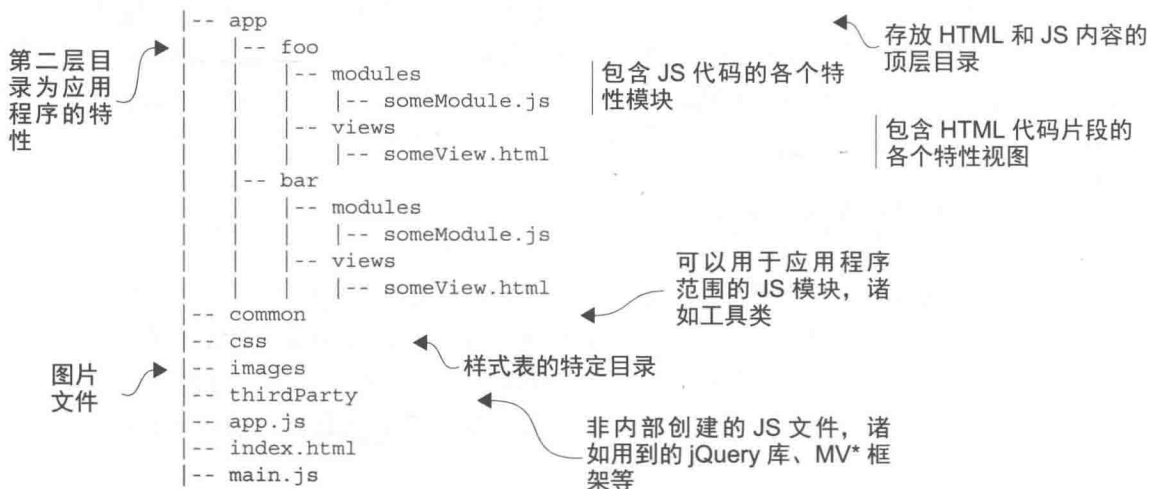
构建组织良好的项目并不复杂，但需要多做思考且不能想当然。幸运的是，目录结构并没有固定标准。一般的经验法则是，选择任何能够为你所用的目录结构方式。现存着一些根据特性（Feature）和功能性（Functionality）¹来组织文件的通用结构方式。

¹ 根据本书上下文判断，特性（Feature）和功能性（Functionality）在本书中各有所指，“特性”着重表示业务逻辑划分，如样例内容中提到的foo、bar，而“功能性”侧重表示架构功能，如模块、视图等。——译者注

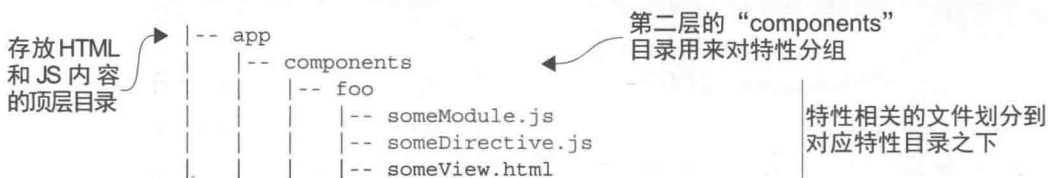
根据特性来分组同类文件有点像编译语言的代码组织方式,比如 Java 包的组织。这种方式干净利索,防止了特性的交叉引用,同时直观地将项目文件结构中特定特性相关的文件划分出来。以下清单描述了使用这种方式组织客户端代码的方法。

AngularJS 风格指南¹建议了一个依据特性组织目录结构的改进版²,其支持清单 1.2 的简化版本,并在每个特性目录之下取消了命名的功能性目录³。该指南的博客内容值得一读,基于应用规模和复杂度,AngularJS 版还会有一些变化。AngularJS 版的结构要点如清单 1.3 所示,其中移除了特性中的各种文件类型的边界。指南认为简化版本仍根据特性来分组内容,但更具可读性,并且为 AngularJS 工具创建了更符合标准的结构。

清单 1.2 目录结构样例(根据特性划分)



清单 1.3 “依据特性”组织目录结构的简化版



1 <http://blog.angularjs.org/2014/02/an-angularjs-style-guide-and-best.html>或<http://angularjs.blogspot.co.uk/2014/02/an-angularjs-style-guide-and-best.html>

2 为了描述方便,我们姑且称其为“AngularJS版”。——译者注

3 这里的功能性目录,指modules、views这样的具体功能目录。——译者注

作为一个选项，你和开发团队可能会选择依据功能性进行项目组织的方式（如清单 1.4 所示），这完全没问题。添加进 SPA 应用的各种库和框架大多不会对目录结构有什么强制规定。选择什么样的项目组织方式实际上取决于个人和团队偏好。若是选择了按功能性来组织目录，在功能性目录下包含特性子目录仍是好主意。否则，在每个功能性目录之下，最终会堆积大量不相干的文件。对于小型应用这不算什么问题，但对于大型应用来说则会制造出大量“垃圾箱”。

清单 1.4 目录结构样例（根据功能性划分）



前面清单所示的方式都非常基础，只是为了给你一个感性认识。应用规模、架构选择以及个人偏好都会对目录类型与命名产生影响。在实践中，*modules* 目录名称可能变为 *js* 或 *scripts*，而 *views* 则可能换成了 *templates*。甚至所选的框架类型也可能影响目录结构的创建方式。例如，创建一个 AngularJS 项目，也许还需要诸如 *controllers*、*directives*、*services* 之类的目录结构。

只要你选择了一种项目组织方式，有了一个约定的文件结构并坚持该组织模型，就能大大增加项目成功的机会。

1.5.2 创建可维护的松耦合 UI

拥有简洁而组织良好的 JavaScript 代码，就在构建可扩展、可维护 SPA 应用的正确道路上向前迈出了一步。对代码进行分层使得 JavaScript 与 HTML 尽可能松耦合则是另一大步。代码分层仍允许 HTML 与 JavaScript 之间相互交互，但能够避免代码的直接引用。

这些分离的层次关系是如何构建的？让我们进入 MV* 模式。分离数据、逻辑及 UI 视图的模式早已有之。其中最值得关注的是模型 - 视图 - 控制器 (Model-View-Controller, MVC)、模型 - 视图 - 表示器 (Model-View-Presenter, MVP) 以及模型 - 视图 - 视图模型 (Model-View-ViewModel, MVVM) 三种模式。近年来，这些模式以 JavaScript 库和框架的形式出现，以将其概念应用到 Web 应用程序的前端领域。这些模式实现的基本思想是，在开发者个人的逻辑之外，以框架或库来管理

JavaScript 与 HTML 之间的关系。MV* 库和框架允许开发者设计 UI, 以让业务数据(模型)与生成的 HTML “页面”(视图, 用户与之交互)之间可以相互通信, 但代码却分开管理。MV* 术语中的最后一个组件 Controller、ViewModel 或 Presenter, 则是 MV* 个中一切机制的协调者。

如图 1.12 所示, 将视图、逻辑和数据分离开, 是一种设计单页面应用程序的有效手段。

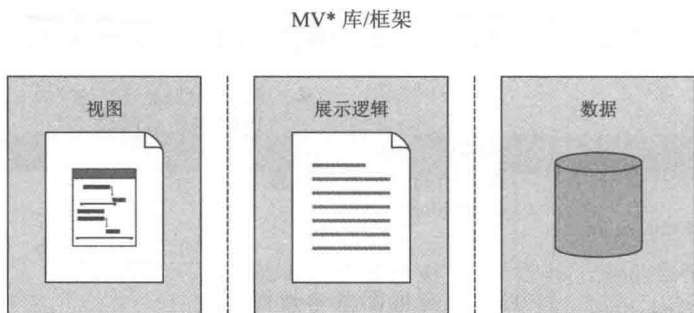


图 1.12 根据用途保持表现层独立, 使得设计者与开发者能够并行工作。同时还有利于开发者更有效地测试、维护及部署代码

对 SPA 应用进行这样的分层有以下优势:

- 设计者与开发者可以更有效地合作。当视图中移除了逻辑时, 两种角色就可以并行开展工作, 不需要依赖某一方的进度。
- 分离视图与逻辑层还能够帮助开发者创建更干净的单元测试, 因为他们只需关注非可视化方面的特性。
- 分层有利于维护和部署。独立的代码更容易更改, 不会影响到应用程序的其他部分。

如果这时候对 MV* 概念还不太理解也是正常现象, 要掌握这些概念确实有点难度。第 2 章会完整覆盖 MV* 的相关内容。

1.5.3 使用 JavaScript 模块

在 SPA 应用开发过程中, 如何让 JavaScript 代码在页面中和谐共处很重要。我们可以将应用功能放置到模块中, 以达成此目标。模块是分组不同功能部分的一种方式, 其隐藏某些内容的同时公开其他内容。ECMAScript 6 已原生支持模块。同时, 各种模式如模块模式已应运而生, 其可以作为回调来使用。

在传统的 Web 应用程序中, 每当页面重新加载时, 页面都需要重新组织。所有之前创建的 JavaScript 对象都得以清除, 同时新页面所需对象再一次被重建。这不仅为新页面释放内容, 而且避免页面函数及变量名与其他页面的函数及变量发生冲

突。单页面应用程序不会出现这种情况。单页面意味着在用户请求新视图时不用从头再来。模块能够帮助你解决个中矛盾¹。

模块能够限制代码的作用域。在每个模块中定义的变量及函数都有其所属的局部作用域（如图 1.13 所示）。

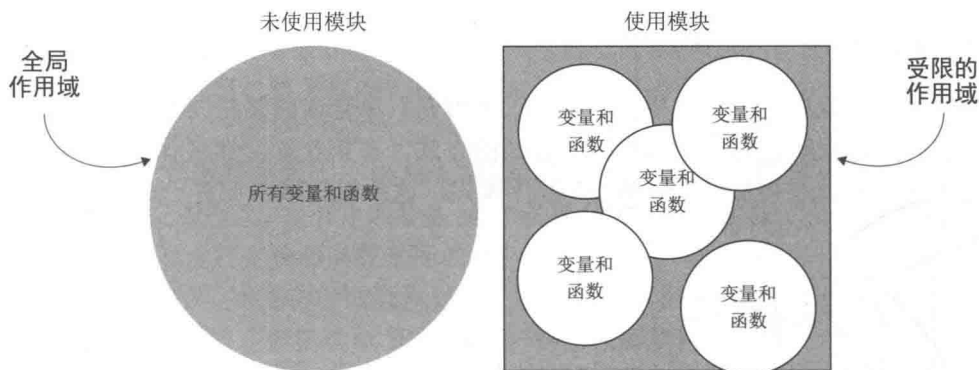


图 1.13 利用模块模式限制变量和函数作用域为模块自身。避开全局作用域相关的各种陷阱

模块模式结合其他相关技术管理模块及其依赖的可行方式，使得程序员能够借助单页面架构方法来设计大型、健壮的 Web 应用程序。

本书大量涉及 JavaScript 模块化编程方面的主题，第 3 章将介绍模块编程。本书还会阐述脚本加载器，脚本加载器能够用来管理模块及其依赖。学习完本书，相信今后你会依赖模块模式来编写应用。

1.5.4 执行 SPA 导航

第 4 章将带你深入了解客户端路由选择。为了让用户掌握其导航位置，单页面应用程序通常会在设计中融入路由选择（Routing）的设计思路：借助 MV* 框架或第三方库的代码实现，将 URL 风格的路径与功能关联起来。路径通常看起来像相对 URL，其充当用户导航时到达特定视图的触发因素。路由器可以动态更新浏览器 URL，并允许用户使用前进与后退按钮。这进一步强化了当屏幕某部分变化时会到达新位置的设计理念。

1.5.5 创建视图组成与布局

在单页面应用程序中，UI 由视图而非新页面构成。内容 Region 的创建以及视

¹ 指函数与变量的冲突问题。——译者注

图在这些 Region 中的位置，决定了应用程序的布局。客户端路由用于连接这些点¹。所有的这些要素有机结合起来就影响了应用程序的可用性和美感。

第 5 章将讨论在 SPA 应用中如何处理视图合成与布局，涉及从简单到复杂的设计思路。

1.5.6 模块通信

模块封装了我们的逻辑，并提供独立工作单元。尽管这能够帮助开发者解耦及私有化代码，但我们仍需有一种模块间通信的方式。在第 6 章，你将看到基本的模块通信方式，其间还将接触到 *pub/sub* 设计模式，这种设计模式允许一个模块广播消息给其他模块。

1.5.7 与服务器端通信

本书开篇介绍了自 XMLHttpRequest API 引入以来，Web 开发技术的演进历史，并由此定义了什么是单页面应用程序。作为技术发展的集大成者，且以 XMLHttpRequest API 为核心的 AJAX，是 SPA 应用构建技术的核心所在。异步获取数据及重绘屏幕各部分区域的能力，是 SPA 应用架构的主要构成。毕竟，在 SPA 应用中导航时，我们像变戏法般使得屏幕不知不觉中变得流畅与轻巧起来；若是缺失了获取数据的能力，戏法如何变得出来呢？

第 7 章聚焦于通过 MV* 框架与服务器端通信。我们将了解这些框架如何抽取大量样板代码用于请求发起及结果处理。同时还将了解 *Promise* 和被称为 RESTful 服务的 Web 服务方式。

1.5.8 执行单元测试

要开发出成功的单页面应用程序，另一个重要而容易忽视的部分是 JavaScript 代码测试。我们通常会测试后端代码片段，而不幸的是，对待 JavaScript 单元测试我们就没这么上心了。当前已经涌现出了不少优秀的单元测试库。第 8 章将通过 QUnit 框架来介绍基本的 JavaScript 单元测试方法。

1.5.9 客户端自动化技术

第 9 章将介绍客户端自动化技术，其不仅可以用于 SPA 应用的构建过程，还可以自动化常见的开发任务。

¹ 指在不同视图间导航。——译者注

1.6 小结

我们来快速回顾一下目前学到的 SPA 概念：

- SPA 是一种 Web 开发方法，整个应用功能都存在于单个页面中。
- 在 SPA 应用中，应用加载之后就不会再有整页刷新。相反，展示逻辑预先加载，并有赖于内容 Region 中的视图切换来展示内容。
- SPA 客户端与服务器端实行异步通信。常用的数据通信格式为 JSON 文本格式。
- MV* 框架提供机制，让 SPA 应用绑定服务器端请求数据与视图（用户所见并与之交互）。本书并未覆盖 MV* 的替代方案，如 React 或 Web 组件技术。
- 与依赖全局变量和函数不同的是，SPA 中的 JavaScript 代码通过模块来组织。模块提供了状态和 / 或数据封装。模块还有助于代码解耦及维护。
- SPA 的优势还包括类桌面应用的呈现效果、解耦的表现层、更快速轻量的负荷、更少的用户等待时间以及更好的代码维护性等。

MV*框架介绍

本章内容

- UI 设计模式概览
- 浏览器端的 MV* 介绍
- 核心 MV* 概念
- MV* 库 / 框架的优势
- 选择框架时的注意事项

在项目演进过程中保持代码优雅，大概是 Web 开发者面对的最困难任务。随着项目规模和复杂度的不断增加，任务将更加艰巨。而如果能够以某种方式塑造代码，使得项目的故障排除、维护以及改进变得更容易而非更困难，则是一件非常了不起的事情。即使对传统 Web 项目而言也是这样的情况。

在 SPA 应用中，基于功能隔离代码的意义已不仅仅是最佳实践，它还是成功实施单页面应用程序的关键所在，而重中之重是实现关注分离。在代码结构中实现关注分离意味着需基于代码职责全力分离各部分代码。

我们可以将整个 SPA 的服务器端和客户端划分为许多层。在浏览器端，可以以一种基本的方式——首先通过 HTML、CSS 和 JavaScript 这三种主要语言的角色，

来着手创建关注分离：

- **HTML**——应用程序的脚手架。其主要关注提供内容占位符的元素，规划出 UI 结构，并提供用户可与之交互的控件。
- **CSS**——样式表描述 UI 的设计，负责外观与格式。
- **JavaScript**——简而言之，该层代码负责应用表现层逻辑。该层通常实现 Web 应用的动态特性，提供其余两层之上的行为与编程控制。

开发者都接触过这三种语言并理解各自角色。即便如此，由于它们可以轻易混合使用，所以常常导致代码很快变成了“意大利面条”（如图 2.1 所示）。这将使得项目的管理非常困难。

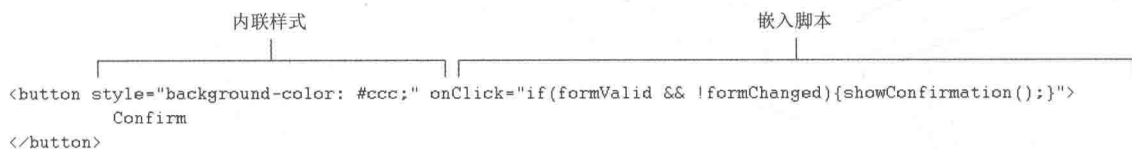


图 2.1 随意交织 JavaScript、HTML 以及 CSS 代码使得项目难以管理

但是，仍然能够将代码组织成解耦方式，且保持各层之间交互不受影响。每部分代码都可以基于其服务目标进行划分。

此外，这种划分可以扩展开来以包含呈现给用户的应用数据。数据及事件可以通过映射分配给 UI，而不用直接在业务逻辑中进行分配。UI 和数据的变化都能够观察到，允许 UI 和数据保持同步，并赋予处理逻辑一种正确的交互方式。因此，不仅代码自身可以基于其职责进行分离，而且 UI 的呈现与展示的数据也能够相互分离。这就实现了更高层次的分离效果（如图 2.2 所示）。

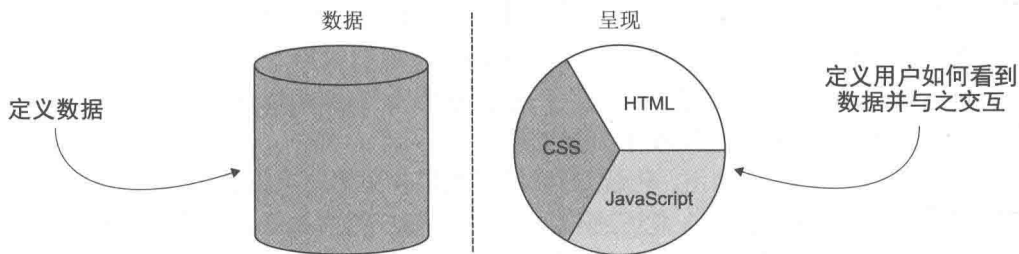


图 2.2 UI 中呈现的应用数据可以分离出来

尽管完全可以另辟蹊径来管理所有分离层，但没什么人愿意在这上面浪费太多开发时间。谢天谢地，有非常多的库和框架很适合此类管理任务。一旦在应用中使用了这些库和框架，它们就将发挥出核心作用，通过管理逻辑、数据与 UI 之间的关系，来创建成功的关注分离。在不同程度上，这些库和框架还提供了大量其他特性，以帮助开发者构建 SPA 应用。

本章讲述 JavaScript MV* 内容，简要讨论其起于传统 UI 设计模式的演进过程，并阐述这些框架的作用。在这里还会涉及一些 MV* 基础概念，同时分别通过三种 MV* 框架的例子，针对同一个目标来介绍不同实现方式。如第 1 章所述，MV* 并不代表一切¹，但本书重点是 MV* 风格的框架。

为了进一步阐述 MV* 中同一概念的实现尽管不尽相同，但确实具有普遍性，我们将使用不同框架来分别创建一个合理、实用的在线通讯录小项目。正文中的示例代码经过了缩减，因此不要一头陷进代码堆里却丢了概念（三个版本的完整代码在附录 A 中提供，同时也能够在线下载）。

本章结尾提供了一个选择合适框架时的参考列表。由于各个 MV* 框架解决问题的方式不尽相同，因此最终都得考虑清楚哪个框架更合适，但不存在绝对的答案。一旦理解了 MV* 的作用及其根本模式，你就能做出最合适的选择。毕竟，没有人比自己更了解自己的情况：那些影响项目、最终用户、预算、时间线以及开发资源的因素。

2.1 MV*概念

如本章引言所介绍的那样，MV* 术语表示基于浏览器的一系列框架，用于构建应用程序的关注分离。这些框架立足于传统 UI 设计模式，但在整个实现过程中，其遵循某种模式的程度是变化的。

在 MV* 概念中，M 代表模型（Model），V 代表视图（View）。2.2 节会进一步覆盖这些概念，但在这里我们先简要浏览一下：

模型——典型的模型包含了数据、业务逻辑以及验证逻辑。从概念上讲，其代表诸如某个客户或某条支付之类的东西。模型从不关心数据的展示。

视图——视图即用户所见以及交互的界面，是模型数据的可视化呈现。有赖于框架其他部分（这些部分负责用户交互的更新和响应），其可以是一个简单结构；或者同样有赖于 MV* 实现，其也可以包含逻辑。

读完 2.1.1 节你就会知道，传统 UI 设计模式包含了第三个组成部分²，该部分帮助管理模型与视图间的关系，以及模型、视图与用户间的关系。尽管大多数 MVC/MVVM 结构的现代 Web UI 设计框架都包含了模型与视图的一些概念，但第三个组成部分在名称及履行职责上却不尽相同。因此，人们通常用通配符 * 来泛指这个组成部分。

2.1.2 节会介绍浏览器端 MV* 的更多内容。但首先让我们来看看传统 UI 设计模

1 比如，还有 React、Web 组件的方式。——译者注

2 即 MV* 的 *——译者注

式，它是 MV* 的基石。跟着历史的脚印将有助于我们更好地理解 MV* 的工作方式。

2.1.1 传统 UI 设计模式

使用架构模式来分离数据、逻辑以及输出结果展示是一个由来已久的概念。这些设计模式的中心思想就是基于应用各层的职责类型进行分离，使得应用代码更容易设计、开发和管理。

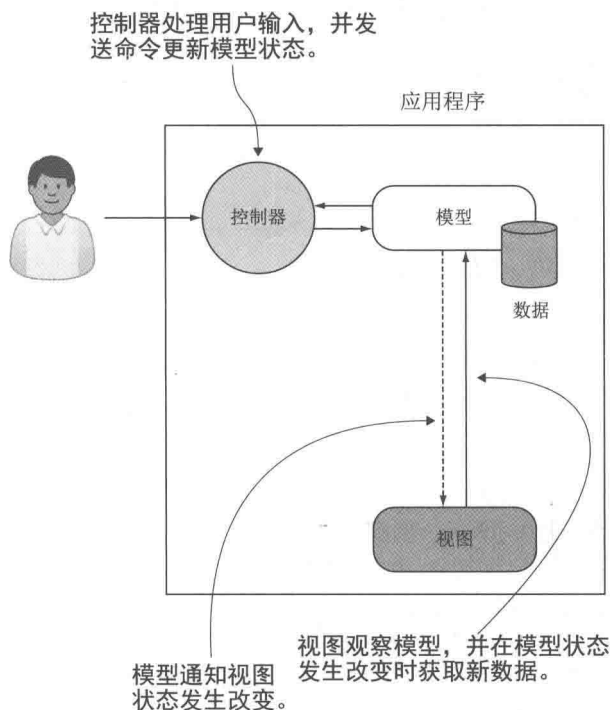


图 2.3 在图形用户界面开发领域，MVC 设计模式已流行许久

设计模式。MVC 模式包含模型、视图和控制器（如图 2.3 所示）：

- 控制器——控制器是应用程序的入口点，接受来自 UI 控件的信号。它还包含了处理用户输入的逻辑，以及基于接收到的输入，发送命令给模型以更新模型状态的处理逻辑。

与控制器的交互引发一个事件链，最终产生视图更新。在此模式中，视图能够捕获模型变化，并在观察到模型改变时进行视图更新。

2. 模型 - 视图 - 表示器

1996 年，一家叫塔利根特（Taligent）的 IBM 子公司提出了一个 MVC 变体结

本节详述对客户端最具影响力的三种设计模式：模型 - 视图 - 控制器（MVC）、模型 - 视图 - 表示器（MVP）以及模型 - 视图 - 视图模型（MVVM）。介绍完这些内容，在接下来的 2.2.2 节将阐述它们如何与当今浏览器 MV* 框架关联在一起。

1. 模型 - 视图 - 控制器

模型 - 视图 - 控制器（MVC）是历史最悠久的，用于分离数据、逻辑和展示的模式之一。MVC 由 Trygve Reenskaug 提出，之后于上世纪 70 代在 Smalltalk 编程语言中得以实现。

时至今日，MVC 仍是图形用户界面设计的强大工具。自问世以来，MVC 及其变体发展成了所有软件开发类型的常见

构——模型 - 视图 - 表示器，简称为 MVP。该模式的出发点是进一步解耦模型与 MVC 其他两个组件的关系。在 MVP 里，类似控制器的对象与视图一起表示用户界面或呈现（Presentation），模型则继续表示数据管理。如图 2.4 所提及，MVP 里没有充当守护者角色的控制器。每个视图都由一个被称为表示器的组件来支持：

- 表示器——表示器包含视图的展示逻辑。视图通过将职责委托给表示器，其仅仅用于响应用户交互。表示器直接访问模型以获取任何更新，并将数据更新回传给视图。在这种方式下，表示器在模型和视图之间扮演了中间人的角色。

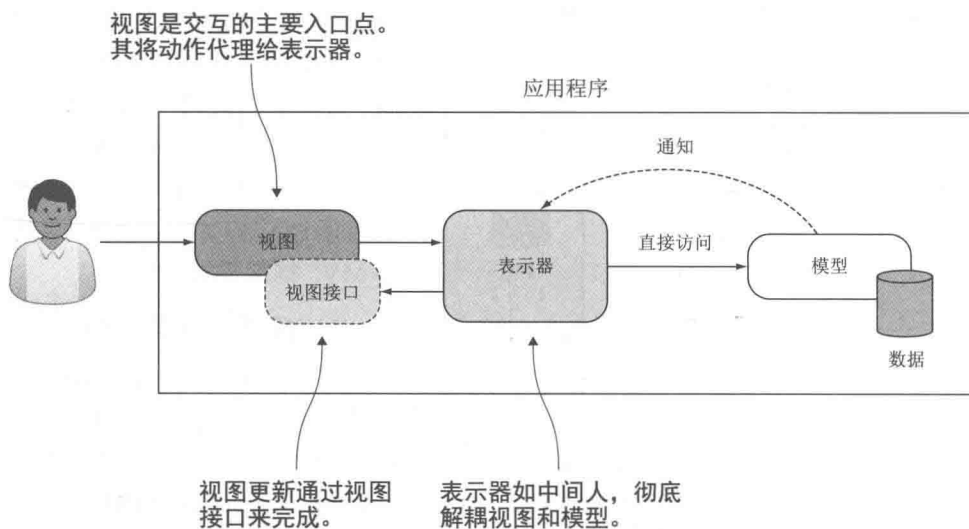


图 2.4 MVP 是一个 MVC 变体。在该模式之下，视图是入口点，但其逻辑则在表示器

表示器负责保持视图与模型的更新。在视图与模型中间建立一个对象，有利于视图和模型更聚焦于各自职责。

3. 模型 - 视图 - 视图模型

模型 - 视图 - 视图模型（MVVM）由 John Gossman 于 2005 年提出，在使用微软 Windows Presentation Foundation（WPF）技术来创建用户界面时，其作为一种简化及标准化创建过程的手段。它是另一个应运而生的设计模式，让 UI 代码更直观、更易维护，同时仍保持组件分离。

如同 MVP 一样，MVVM 的视图也是入口点，而且 MVVM 的模型也有一个对象位于模型与视图之间（如图 2.5 所示）。在此模式中，第三个组成部分被称为视图模型：

- 视图模型——视图模型是视图的模型或展示代码，此外，其还是模型与视图之间的中间人。所有定义及管理视图的代码都包含在视图模型中。其包含了

数据 Property¹ 及展示逻辑。在模型中，每个需要在视图里得以反映的数据点，都映射到视图模型的对应 Property 上。就像 MVP 的表示器，每个视图都由一个视图模型所支持。视图模型能够掌握视图与模型的变化，并保持两者同步。

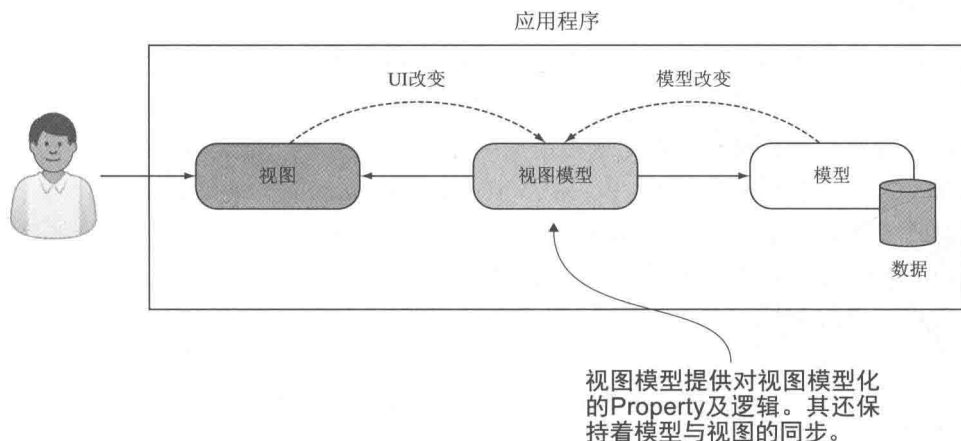


图 2.5 MVVM 中，视图模型能够掌握视图与模型的变化，并保持两者同步

现在我们了解了一点传统 UI 设计模式的知识，这有助于理解浏览器端的 MV* 方式。接下来，我们要开始讨论浏览器端的 MV* 了。

2.1.2 MV* 和浏览器环境

就像服务器端应用或原生桌面应用的代码一样，浏览器端代码也能从好的架构设计模式中受益，这也是近年涌现出来的大量框架所追求的目标。

大多数框架都在某种程度上遵从 MVC、MVP 或 MVVM。然而浏览器环境却大不相同，我们得一股脑地面对三种语言（JavaScript、HTML 与 CSS）。因此，很难用某种设计模式来完全匹配浏览器端 MV* 框架。在大多数情况下，试图将这些框架分门别类是徒劳的。设计模式应该是可扩展的，而非僵化指令。

我们首先讲述 MV* 术语概念的一个理由是，其通常很难明确框架中的第三个组成部分。MV* 术语是一个折中的表述，以此来绕开框架到底像哪种模式的无休止争论。

传统模式的第三个组成部分的概念是松散的。每种模式都有某种形式的数据模型，无论是 POJO（普通 JavaScript 对象，Plain Old JavaScript Object）形式，还是某些取决于实现的模型结构。每种模式也都有视图的概念。而模式中的第三个组成

¹ 为了区分 Attribute 与 Property，书中的 Property 一律保持英文表述，而“属性”特指 Attribute。——译者注

部分则更难以捉摸，框架可能会采用明确的控制器、表示器或者视图模型，或采用某种混合方式，甚至可能就根本没有此组成部分！

Backbone.js 组合应用库 Marionette.js 的作者 Derick Bailey，在其在线博客上发表了一篇极具争论性的文章 *Backbone.js Is Not an MVC Framework*（《Backbone.js 并不是一个 MVC 框架》）：

最后，试图将 Backbone 塞进某个不合适的模式语言中是个歪主意。结果就是毫无意义的争论和夸大，以及啰啰唆唆的本篇博文，因为没有人能够就哪个俗套的模式名称更适合达成一致。依我看来，Backbone 并不属于 MVC，同样也不属于 MVP、MVVM（如 Knockout.js）或任何其他什么众人皆知的特定名头。它从 MV* 一族里提取了各种风味，并创建一个非常灵活的工具库，使得开发者可以用它来构建酷炫网站。因此，我看还是把所谓的 MVC/MVP/MVVM 统统扔出窗外，只需说 Backbone 是一种 MV* 框架就好。

——原链接：<http://lostechies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework/>

许多人持相同的观点，认为试图用传统设计模式来一一匹配现今的 MV* 是徒劳的。当 AngularJS 团队给出相似结论的时候，框架有效性优于框架分类的理念获得了更多的支持。AngularJS 团队的 Igor Minar 在博客中明确指出，开发者会无休止地争论一个特定 MV* 框架的分类。他继续表示 AngularJS 刚开始更像 MVC，但随着时间的推移它又变得更像 MVVM。实际上，AngularJS 却更像两者的结合。在该篇博客中，他还提出了 MVW 这一术语（这当然不是真的）：

我希望看到开发者构建经过良好设计并遵从关注分离的强大应用，而非看到他们浪费时间争论什么 MV* 废话。因此，我特此声明 AngularJS 是 MVW 框架——Model-View-Whatever：Whatever 代表任何对你胃口的东西。

——原链接：<https://plus.google.com/+IgorMinar/posts/DRUAKzmXjNV>

大多数 MV* 框架实现只是松散地以原始传统设计模式为基础进行设计，了解了这一点，就能认识到框架属于哪种模式并不重要。

2.2 MV*基础概念

现在，我们弄明白了 MV* 是什么，接下来回顾应用实现中几个频繁出现的基础概念。在每个概念的示例里，你将很快发现即使框架间的语法和方式有所不同，但核心理念是一样的。在开始之前，先花点时间来快速回顾一下本节所涉及的概念：

- 模型——模型代表应用数据。其包含了访问及管理数据所需的 Property、处理逻辑与验证。模型常常还包含了业务逻辑。
- 视图——视图是用户所见并与之交互的部分，模型在这里得以可视化呈现。在某些 MV* 实现中，视图还可能包含展示逻辑。
- 模板——模板是视图的可复用构件块，用来处理视图中的动态内容。其包含数据的占位符以及其他用于模板内容渲染的指令。在 SPA 应用中通常会用到一个或多个模板来创建视图。
- 绑定——该术语描述模型数据与模板元素的关联处理。某些 MV* 实现中还提供了其他类型绑定，如事件与 CSS 样式间的绑定。

图 2.6 展示了一个全局视图，用来描述上述概念在 SPA 应用中的相互关系。这些概念可能是 SPA 应用构建过程所需的最基本要求了。其他的一些特性，诸如路由（第 4 章内容），也很常见（甚至必须），但未必得以普遍提供。不必担心，本书后续内容将覆盖大量其他的相关概念。当下我们的起步只需要打好坚实的基础。

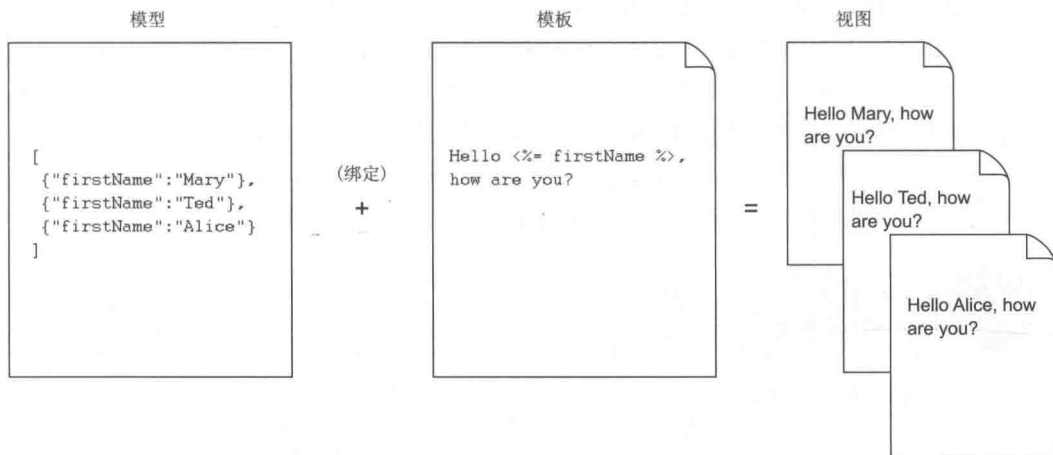


图 2.6 模型数据与可复用的模板相结合（绑定），以创建构成 SPA UI 的视图

2.2.1 框架

由于我们使用三个框架来阐述本节内容，因此会按一定顺序展开相应内容。每个框架采纳的方式都跟基本 MV* 概念稍有不同。而通过对不同方式的了解，最终将大大拓宽你的视野。先来了解一下这三个框架。



URL : <http://backbonejs.org>

描述：如前所述，Backbone.js 并不完全遵循传统设计模式，但从某种角度来看可以视为介于 MVC 和 MVP 之间。Backbone.js 是代码驱动的。模型与视图通过扩展 Backbone.js 对象，以 JavaScript 编程方式创建。通过扩展核心对象，应用就自动继承了内建的大量功能。Backbone.js 还提供了其他的开箱即用特性，使得日常任务变得更容易。然而 Backbone.js 并未提供 SPA 应用构建过程所需的所有功能，因此需要借助其他库或框架来填补某些空白。



URL : <http://knockoutjs.com>

描述：Knockout 也许并不完全遵从原始 MVVM 定义，但还是比较接近的。在该框架中，模型代表任何的数据源，而非由框架规定的显式对象结构。视图及模板通过纯 HTML 代码实现。视图模型映射模型数据到 UI 元素，并把以 JavaScript 编程方式创建的行为提供给视图，但大部分的其他功能，则通过声明方式（在 HTML 中添加定制属性）来实现。Knockout 主要关注绑定处理过程的清晰与简单程度。Knockout 的实现思路使得其小巧而更聚焦，这也驱使你去看一下别的 SPA 应用采纳的其他框架和库。



URL : <https://angularjs.org>

描述：AngularJS 不露主角地称自己为“超级英雄般的 JavaScript MVW 框架”。其创作团队把它设计成了一个一站式框架，几乎能够满足 SPA 应用构建过程中的各种需求。AngularJS 混搭了传统模式和其他流行框架中的各种概念，最终使得其开箱即用特性的平衡度达到较高层次。使用 AngularJS 开发应用时，一部分开发工作靠 JavaScript 编程方式完成，另一部分则通过定制 HTML 属性以声明方式完成。

2.2.2 我们的 MV* 项目

为了举例说明前面所列的基础概念，我们将创建一个简单的员工在线通讯录。我们考虑使用 2.2.1 节列出的三种框架来分别创建。本书的后续章节将涉及更多高级主题，诸如路由和服务器端事务处理。现在，我们只需关注所建项目的基础部分。

尽管这个例子有点儿刻意而为，但仍然覆盖了基本的 CRUD 操作。就入门角度来看这还是有足够挑战度的。

下面来看看这个例子要实现的功能：

- 创建一个简单 SPA 应用，用于员工信息输入。
- 构建一个易用的 UI 界面，满足员工姓名、头衔、邮箱地址以及电话号码的输入要求。
- 跟踪条目列表的每个条目，整个屏幕分为两部分，左边是具体条目内容，右边是通讯录条目列表。
- 左边条目内容区域有两个按钮：一个是添加新条目按钮，另一个是清除表单按钮。
- 右边列表的每个条目旁边有一个按钮，用来从列表中移除条目。
- 每个条目字段旁边都有指示器，用来表示输入内容是否满足该字段要求（在用户输入时，每个指示器的内容会随之变化）。

现在我们梳理清楚了示例目标，接下来看一下应用最终效果图（如图 2.7 所示）。

三个 MV* 框架开发出来的应用都具有相同的外观和行为。

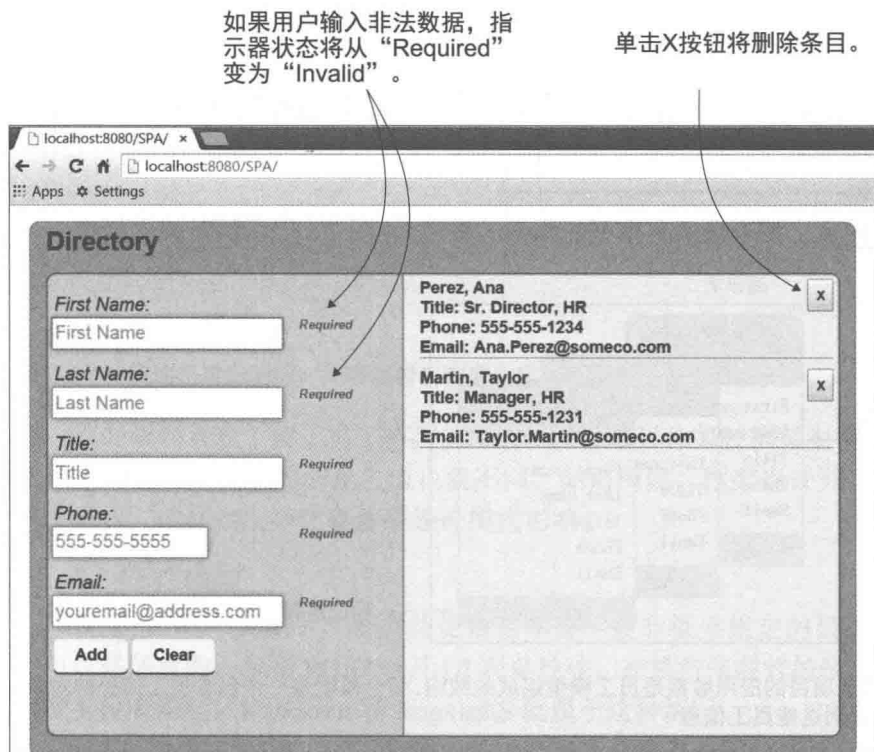


图 2.7 在线通讯录截图。用户在左边输入信息，合法条目在右边列表中呈现

讨论完 MV* 概念，我们会着手开发这个示例项目。同时，本书还将讨论不同的框架设计哲学对编码类型产生的影响。尽管阐述过程只会用到一部分代码，但每个 MV* 版本的完整代码都包含在附录 A 中，并提供下载。

2.2.3 模型

从模式讨论中我们得知模型常常包含了业务逻辑与验证。其也代表了应用数据。然而，模型所包含数据不应该是无关信息的大杂烩。之所以称之为模型，是因为它是一个对应用逻辑而言很重要的、现实存在的实体。

假设你在构建一个酒店在线预订系统，模型就可能包含酒店、房间、代理商、顾客、预订、便利设施、注意事项、发票、收据以及付款等内容。而教师使用的 Web 应用该是什么样子的呢？数据至少要包含学校、老师、学生、课程、分数等级。每个模型都代表现实中的某个对象。因此，系统越大越复杂，模型类型也就越多。

我们来看看员工在线通讯录需要什么模型。记住模型对照现实世界的事物，其包含的不仅是数据，还有行为。在这里，我们将建立通讯录列表模型。为了突出概念而非代码，我们尽量确保功能简单。每个模型包含内容如下：

- 名字
- 姓氏
- 头衔
- 邮箱地址
- 电话号码

通讯录里的员工列表是所有模型的集合（数组）。图 2.8 展示的是概念模型。

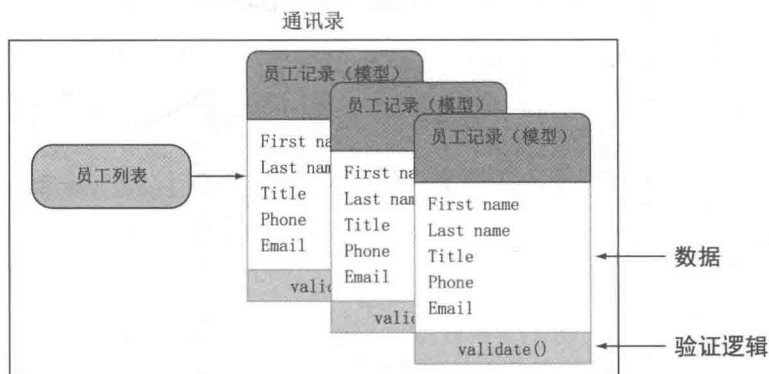


图 2.8 在线通讯录项目的应用数据是员工模型组成的数组。每个模型是一个包含员工信息的对象，我们将在页面上看到这些员工信息

图 2.9 展示了模型添加进集合后的可视化效果。记住，每个列表模型是数组中的某个对象。为了感受实际模型，本图叠加展示了添加到列表中的每个模型数据快照。

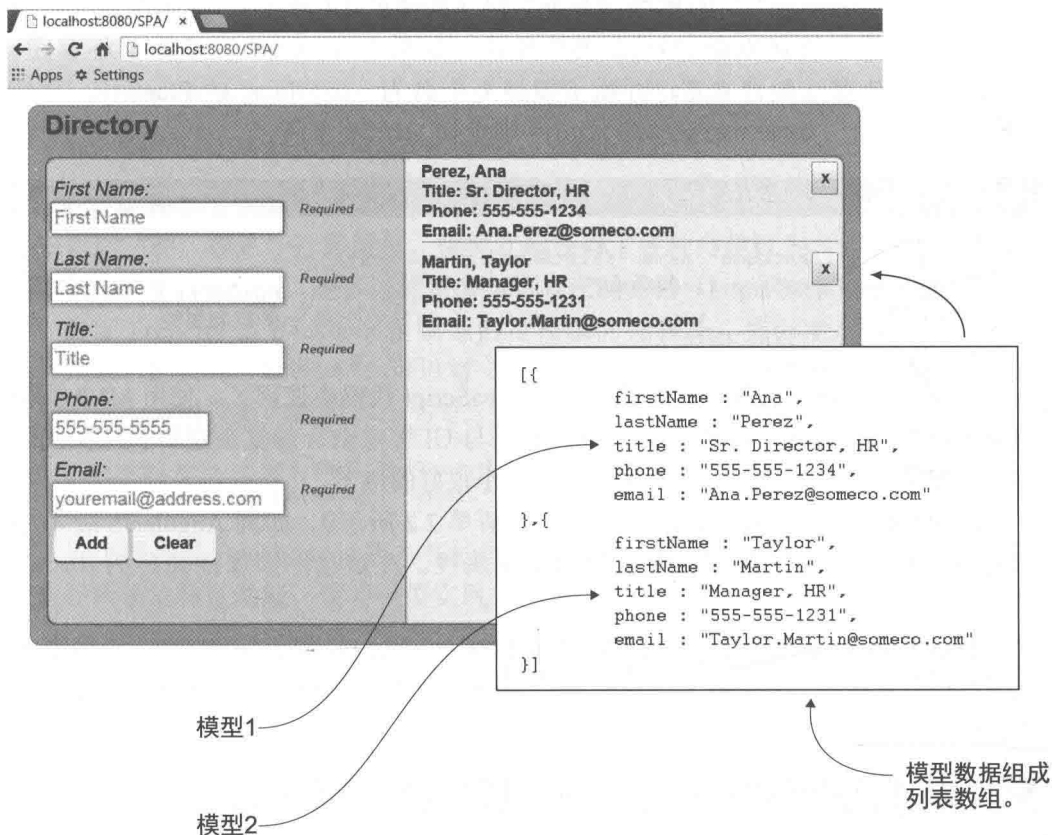


图 2.9 在线通讯录示例图。你可以看到列表中添加了员工模型的两个实例

现在我们直观了解了员工模型的法，接下来讨论每种框架中模型的定义方法。使用不同框架，模型的创建之道也就不同。如前所述，框架也许并不完全契合传统设计模式，但其实现必然受传统设计模式影响。

1. 隐式模型

在某些 MV* 实现中，模型就是数据本身，而非框架规定的显式结构，其数据来源可以是任意的，包括 POJO 以及 UI 表单控件。在模型是隐式的情况下，用什么数据来源并没有限制。Knockout 和 AngularJS 就属于这种情况。

例如，在虚构的员工通讯录中，员工模型的数据来自用户所填条目表单。因此，

不像创建 JavaScript 对象或者从服务器端获取 JSON 格式数据，当创建通讯录列表的每个条目时，我们需要直接从表单的 INPUT 输入字段中获取数据。

AngularJS 为此提供了一种简捷方式。如果要从 INPUT 字段中获取数据，可以添加一个 `ng-model` 定制属性到每个字段（如清单 2.1 所示）。该属性声明模型数据来自放置了该属性的 HTML 表单元素。如果 `formEntry` 模型尚不存在，该属性就会施魔法般建立它，并赋予模型一个名为 `firstName` 的 Property。之后 `ng-model` 会把 `formEntry.firstName` 绑定到 INPUT 字段。

清单 2.1 AngularJS 模型

```
<input id="firstName" name="firstName" type="text"
ng-model="formEntry.firstName"
required
placeholder="First Name"/>
```

ng-model 是 AngularJS 的定制属性

一旦建立好内在关联，模型就可以在 JavaScript 代码中使用了。使用 MV* 框架的众多好处之一就是，应用逻辑之外，数据与 UI 绑定结合的复杂逻辑和样板代码将由框架帮你打理。`ng-model` 属性就是一个很好的例子。

Knockout 的模型也是隐式声明的（如清单 2.2 所示）。如同 AngularJS 版本中的处理，我们为每个 INPUT 字段添加定制属性。在这里，该属性被称为 `data-bind`。

通过 Knockout（真正的 MVVM 风格），`data-bind` 属性将 INPUT 字段绑定到视图模型对应的 Property 上。反过来，JavaScript 代码也就能够通过视图模型访问该字段。

清单 2.2 Knockout 模型

```
<input id="firstName" name="firstName" type="text"
data-bind="hasFocus: isFocused,
value: entry.firstName,
valueUpdate: 'afterkeydown'"
placeholder="First Name" />
```

data-bind 是 Knockout 的定制属性

将在代码中定义的、视图模型中对应的 Property

通过 AngularJS 和 Knockout，模型的数据源可以是任意的。由于我们依托条目表单，因此条目表单就是我们的数据源。在 AngularJS 和 Knockout 这两个例子中，并没有显式定义模型对象。相反，这两个框架都提供了一个定制属性，我们可以将其添加到 HTML，以将条目表单作为模型的数据源。现在，我们准备来看看 Backbone.js 中如何创建模型，Backbone.js 的模型是在代码中显式定义的。

2. 显式模型

在框架规定需要显式声明模型的 MV* 实现中，模型作为 JavaScript 对象来创建。

Backbone.js 是此种类型的代表。Backbone.js 模型除了数据还可以带有逻辑,如验证、缺省数据和定制函数。此外,Backbone.js 模型还可以继承大量功能。只需通过扩展框架自身的模型来创建我们的模型,就自动获取了各种各样的基本功能,甚至不需要写代码。

直接继承大量功能的能力,使得此类框架的模型创建能力变得异常强大和灵活。例如,在 Backbone.js 中创建一个简单模型只需一行代码:

```
var EmployeeRecord = Backbone.Model.extend({});
```

为了使用 Backbone 对象,我们创建了一个它的新实例,并能够调用其提供的开箱即用功能。在这一行声明里,模型立即获得了各种内建行为,诸如验证、执行 RESTful 服务的函数等。更多功能可参考官方在线文档 (<http://backbonejs.org>)。

分配 Property 给 Backbone.js 模型的过程也同样容易。要创建 firstName 新属性并设置它的值为 “Emmit”,可以将 {firstName : “Emmit”} 传给对象的构造函数,或者使用模型内建的 set 方法:

```
var employee = new EmployeeRecord({});
employee.set({firstName : "Emmit"});
```

以下清单是 Backbone.js 版本在线通讯录的员工模型 (如清单 2.3 所示)。通讯录程序所需的验证功能,使得模型实现代码相当啰唆。

清单 2.3 Backbone.js 模型

```
var validators = {
  "name": [ {
    expr: /\S/,
    message: "Required"
  } ],
  "phone": [ {
    expr: /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/ ,
    message: "Invalid"
  } ],
  "email": [ {
    expr: /^[a-z0-9!#$%&'*+\/=?^_`{|}~]+(?:\.[a-z0-9!#$%&'*+\/=?^_`{|}~]+)*@
      (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9]
      (?:[a-z0-9-]*[a-z0-9])?$/i,
    message: "Invalid"
  } ]
};
```

定义模型 Property 中数据的验证规则


```
function validateField(value, key) {
  var rules = validators["name"].concat(validators[key] || []);
```

记录具体错误

```
var broken =
_.find(rules,function(rule) {return !rule.expr.test(value)});

return broken ? {"attr":key,"error":broken.message} : null;
}

var EmployeeRecord = Backbone.Model.extend({
  validate: function(attrs) {
    var validated = _.mapObject(attrs, validateField);
    var attrsInError = _.compact(_.values(validated));
    return attrsInError.length ? attrsInError : null;
  },
  sync: function(method, model, options) {
    options.success();
  }
});
```



创建模型

Backbone.js 在底层为我们做了大量工作。我们通过扩展 Backbone.js 的基本对象，继承了 Backbone.js 模型的强大能力。但框架把数据验证的工作留给了开发者。Backbone.js 还提供了钩子函数 `validate(attrs, options)`，你可以在其中补充自己需要的代码。

2.2.4 绑定

绑定（Binding）是你使用 MV* 框架时应该要理解的另一个概念。其在讨论 UI 开发时会频繁涉及。其直接意思是把两个东西绑/连接在一起。在我们讨论诸如 .NET 桌面程序或 MV* Web 开发时，该术语指的是将视图的 UI 元素与代码相关部分联系起来（比如某个模型的数据）。

然而绑定内容并非仅限于数据。不同的库和框架支持不同的绑定类型。样式、属性以及如 `click` 这样的事件等都可以绑定到 UI。依赖于所选框架，绑定类型也不尽相同。在本节你将看到一些绑定方式的示例代码。

在应用程序中如何准确声明绑定？MV* 框架简化了代码与 UI 元素间的绑定操作。要理解如何声明绑定，需先了解绑定语法。

1. 绑定语法

绑定语法有两个特性：

- 表达式（Expression），用来包装/限制绑定条目的特定字符。
- HTML 属性（Attribute，AngularJS 中称为指令——Directive，Knockout 中称为绑定——Binding）。

有了这两个特性，绑定语法就可以自由混入模板的 HTML 代码。表 2.1 列出了一些流行库/框架所用的绑定语法例子。这不是一份完整列表，但能让你了解个大概。

还请记住，表 2.1 使用简单的文本绑定来阐述语法风格。如前所述，除了数据，诸如事件和 CSS 样式，都可以为绑定所支持。通过各个库 / 框架的官方文档，可以了解绑定支持的完整内容和进一步的使用说明。

表 2.1 属性或表达式的绑定形式。AngularJS 在一定程度上同时支持这两种风格

框架 / 库	类型	示例
Knockout http://knockout.com	属性	<code>data-bind="text: firstName"</code>
AngularJS (类型 1) https://angularjs.org	属性	<code>ng-bind="firstName"</code>
AngularJS (类型 2)	表达式	<code>{{ firstName }}</code>
Mustache http://mustache.github.io	表达式	<code>{{ firstName }}</code>
Handlebars http://handlebarsjs.com	表达式	<code>{{ firstName }}</code>
Underscore.js (缺省) http://underscorejs.org	表达式	<code><%= firstName %></code>

看过框架或库的文档，了解了绑定语法之后，下一步就是理解绑定的定向数据流了。

2. 定向绑定

将代码相关部分绑定到视图的可视元素的方式可以是双向、单向或单次绑定。绑定的关系类型也是通过 MV* 框架来创建的。

3. 双向绑定

双向 / 两路绑定，在绑定连接建立后，改变任何一端的状态都会导致对端随之更新。这确保了两端同步。在 Web 应用程序中，双向绑定与 UI 控件关联，例如表单元素，其支持用户输入。

Knockout 是一个用以阐述双向绑定概念的极佳 JS 库。别忘了绑定是 Knockout 的主要设计目标。如前所述，在 Knockout 中创建一个绑定只需简单地在 HTML 中输入定制属性 `data-bind`。`data-bind` 属性告知 Knockout 在 UI 里的某个元素准备绑定到视图模型中的某个 **Property** 上。在下面的例子里，我们将把 INPUT 控件的值绑定到名为 `firstName` 的视图模型 **Property** 上：

```
<input data-bind="value: firstName" />
```

在双向绑定关系的另一路，通过将该 **Property** 的数据包装到 Knockout 观察对象中，我们通知 Knockout 希望其能够观察到该 **Property** 的更新（还记得观察者模式的

观察值吗？)。

```
var myViewModelObject = {  
    firstName : ko.observable("Emmit")  
};
```

恰是这寥寥数行代码构建出的双向绑定，我们得以确保关联的两端自动同步。

AngularJS 的绑定处理甚至更简单。在前面讨论模型时，我们已经接触到了 AngularJS 双向绑定的例子。可以通过添加 `ng-model` 属性到 `INPUT` 标签来实现绑定：

```
<input ng-model="firstName" />
```

在 JavaScript 代码端，有一个 `$scope` 对象而非视图模型。作用域大致处于视图与 JavaScript 代码之间，并让我们能够访问模型。

AngularJS 框架的神奇之处就是能够自动化诸多双向设置。首先，`$scope` 对象自动监控模型的改变；其次，甚至不必创建 `$scope` 对象，只要有需要，AngularJS 就会为你自动创建好；再者，可以在代码中通过 `$scope` 对象引用 `Property`：

```
$scope.firstName
```

仅此而已。现在 `INPUT` 字段和 `Property` 就实现了双向绑定。任何一端发生改变都会影响另一端。非常容易，不是吗？

我们已经了解了两种框架的双向绑定方式。即便如此，两者的概念还是保持一致的。接下来快速浏览一下单向绑定。

4. 单向绑定

在单向 / 单路绑定中，源状态的改变会影响目标状态，但反之则不然。这种类型的绑定通常跟那些不需要用户输入的 HTML 元素有关，诸如 `DIV` 或 `SPAN` 之类的标签。对于这些类型的元素，我们一般关心其文本而非值。在 JavaScript 代码端的数据访问方式则跟双向绑定一样，但在模板中需特别为单向文本绑定选择对应属性。

在 Knockout 中，将 `value` 改为 `text`：

```
<span data-bind="text: firstName"></span>
```

在 AngularJS 中，属性由 `ng-model` 改为 `ng-bind`：

```
<span ng-bind="firstName"></span>
```

我们再次发现，纵然是不同的 MV* 框架，其绑定概念却没什么不同。

提示 如果需要, Knockout 提供了额外的方式来支持用户输入的单向绑定。这时候只需从视图模型 Property 中移除观察值“包装器”, 如 `firstName: "Emmit"`。

此时容易产生困惑, 为何要自寻烦恼使用单向绑定? 为什么不总是使用双向绑定呢? 嗯, 通常如自动处理、双向绑定此类神奇的功能, 都是要付出代价的。双向绑定需要稍多一点的开销。但也不用太敏感和回避双向绑定。对绝大多数的视图而言, 这种开销微不足道。但如果应用程序中有大量的双向绑定, 你就应该采取措施以节省开销。

如果视图需要接受用户输入, 而且数据与视图间始终需要保持同步, 则应该选择双向绑定的方式。如果 UI 元素是只读的, 则选择单向绑定方式。单向绑定在模型改变时会同步更新视图, 但不会监控视图端, 因为元素是只读的。

5. 单次绑定

单次绑定是单路绑定的一种类型, 但绑定作用只发生一次。更新状态始终不会被观察到。不管源状态还是目标状态发生改变, 都不会有联动的后续状态更新。

在单次绑定中, 当模板和数据一旦结合并以视图方式渲染出来, 此次绑定就结束了。如果新的更新需要应用到视图, 则整个处理会重新再来。之前的视图将摧毁, 新数据绑定到同一模板以重新生成视图。

本节未涉及 Backbone.js。Backbone.js 渲染模板的典型方式是单次绑定 (虽然某些 Backbone 兼容库和插件提供了其他两种绑定方式)。对于 AngularJS 和 Knockout, 在绑定建立之后, 它们即可重复使用。Backbone.js 的总体思路是: 需要新数据时, 视图 (与绑定一道) 被摧毁并重建。我们将在 2.2.5 节涉及更多的模板相关内容。

注意 Backbone.js 并不具备内建的模板 / 绑定能力, 但允许开发者选择外部库。其支持的缺省模板库是 Underscore.js。

绑定概念概括起来如表 2.2 所示。

表 2.2 绑定类型包括双向绑定、单向绑定和单次绑定

绑定类型	行为
双向绑定	双向的——始终保持数据与视图的同步
单向绑定	单向或单路——改变源状态将影响目标状态, 但反之不然
单次绑定	单路——由模型到视图, 在渲染期间只发生一次

2.2.5 节将通过在线员工通讯录实践绑定与模板。

2.2.5 模板

模板 (Template) 是 HTML 片段, 其作为视图如何渲染的方式。渲染方式可以额外包含多种绑定及其他指令, 决定模板及其模型数据如何处理。顾名思义, 模板是可重用的。

一个视图由一个或多个模板创建, 而复杂视图通常有多个渲染模板同时呈现。无论是内建或借助外部库, MV* 框架中将模板和模型数据结合起来的那部分, 通常被称为模板引擎 (Template Engine)。图 2.10 阐述了数据 (来自员工通讯录表单, 我们的模型) 与模板的结合, 之后将效果呈现给用户。

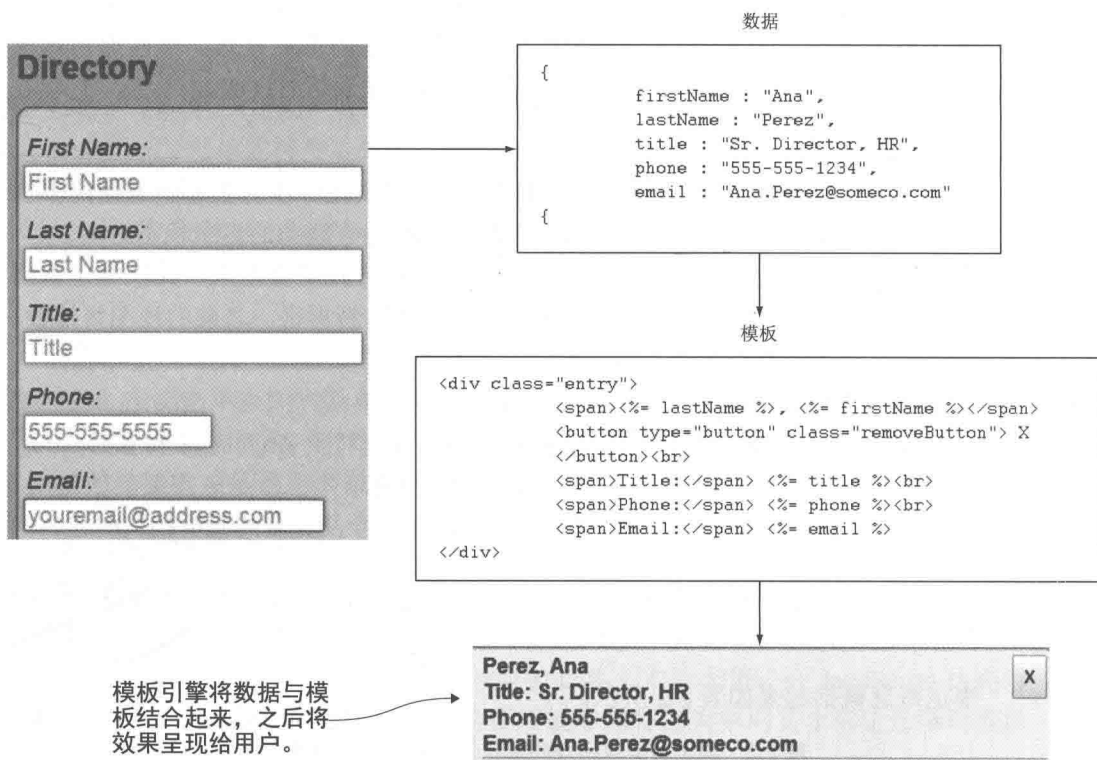


图 2.10 整个渲染模板, 由模板引擎创建

现在看到绑定时你应该能够鉴别它们了, 因此接下来看一些模板及其绑定的实际样例。我们突出显示了绑定内容, 因此可以容易地识别出模板哪部分是绑定、哪部分只是 HTML。

1. 模板概括

所有模板都有一个特性是它们代表了视图的某部分。对开发者而言, 这意味

着除了绑定语法，视图只是 HTML 代码。也就是说如果团队里有 Web 设计师，视图也能设计师所构造。

清单 2.4 展示了 Knockout 模板。注意其中有定制属性，而除此之外其他内容都是普通 HTML 代码。

清单 2.4 Knockout 模板

```
<li class="entry">
  <button type="button" class="remove-entry"
    data-bind="click: removeEntry">&#9587;</button>

  <span data-bind="text: $data.lastName"></span>,
  <span data-bind="text: $data.firstName"></span>
  <br />

  <span>Title:</span>
  <span data-bind="text: $data.title"></span>
  <br />

  <span>Phone:</span>
  <span data-bind="text: $data.phone"></span>
  <br />

  <span>Email:</span>
  <span data-bind="text: $data.email"></span>
</li>
```

click 绑定用
来移除条目

data 绑定用来将
表单字段联系到
视图模型的数据

还记得讨论绑定语法时，AngularJS 既可以使用表达式也可以使用属性吗？在我们的在线通讯录程序里，使用表达式仅是为了演示其用法（如清单 2.5 所示）。如果你更喜欢属性风格，则可以使用 `ng-bind` 替代。

清单 2.5 AngularJS 模板

```
<li class="entry" ng-repeat="entry in entries">

  <button type="button" class="remove-entry"
    ng-click="removeEntry(entry)">&#9587;</button>

  {{entry.lastName}}, {{entry.firstName}}<br />

  Title: {{entry.title}}<br />
  Phone: {{entry.phone}}<br />
  Email: {{entry.email}}

</li>
```

调用一个函
数，传入当
前条目对象

为条目列表中的每个
条目生成模板

Backbone.js 不局限于特定模板语法，因为它并没有内建模板库。Backbone.js 允许我们选择需要的模板引擎。在在线通讯录例子里，我们将采用 Backbone.js 缺省的

模板库 Underscore.js（如清单 2.6 所示）。Underscore.js 表达式的缺省定界符是 `<%= %>`，但定界符并未做限定（包括 `{{ }}`）。

清单 2.6 Backbone.js 模板，使用 Underscore.js 模板库

```
<button type="button" class="remove-entry">&#9587;</button>
<%= lastName %>, <%= firstName %><br />
Title: <%= title %><br />
Phone: <%= phone %><br />
Email: <%= email %>
```

我们使用缺省的 Underscore.js 作为模板引擎，但也可以用另一种模板引擎替代它

这些示例很好地介绍了模板的用法以及各种绑定风格。但我们尚未讨论如何触发处理过程。

2. 模板渲染

一旦应用启动，AngularJS 的模板渲染就会自动触发。AngularJS 搜索 DOM 中的定制属性，包括绑定模板的定制属性。

对于其他框架，模板渲染的方式也不难，但可能需要稍加明确。例如 Knockout，需要在 JavaScript 代码中为每个视图模型添加一行代码以激活绑定：

```
knockout.applyBindings(myViewModel, $("#someElement")[0])
```

Knockout 调用特定函数 `applyBindings`，其用视图模型提供的模型数据渲染模板。第一个参数是视图模型自身。第二个参数是 DOM 中的位置，该位置是你希望 Knockout 为所给视图模型查找绑定的起点。第二个参数是可选的，但出于效率考虑应该添加它，以将绑定处理限制在特定父元素上。

提示 `$("#someElement")[0]` 是 jQuery 用法，用来访问选择器引用的底层 DOM 对象，因为 jQuery 并不知道有多少元素会匹配给定选择器。我们也可以使用 JavaScript 的 `document.getElementById("someElement")` 方法作为第二个参数。

对于 Backbone.js，渲染模板更多是手动过程。Backbone.js 提供 `template` 和 `render` 作为所选外部模板引擎的钩子。要将模板渲染到屏幕上，我们必须通过编译和渲染处理来运行它。下一节里我们会看到全貌视图，但现在仍聚焦于模板渲染，如清单 2.7 所示。

清单 2.7 Backbone.js 模板编译与渲染

```
template : _.template(templateHTML),
render : function() {
    var modelAsJSON = this.model.toJSON();
```

转换模型数据为 JSON 字符串

编译模板到可重用函数中

结合数据与
HTML

```
}  
var renderedHTML = this.template(modelDataAsJSON);  
this.$el.html(renderedHTML);  
return this;  
}
```

用待渲染的 HTML
替代元素内容

相较于其他两种框架，前面的 Backbone.js 代码看起来比较冗长。但这换来了允许选择任何模板引擎的能力。

现在我们了解了模板是什么以及如何创建它们。接下来的问题也许是我们应该在哪儿保存这些模板代码。

3. 模板的存放位置

模板可以包含在 SPA 初始下载中（内嵌），或者作为外部的局部代码（或片段）按需下载。

4. 内嵌模板

如果模板使用了表达式风格绑定语法，且并非按需下载，则需要将模板放置在 `<script>` 标签中。通过 `<script>` 标签避免在渲染处理发生前暴露绑定代码给用户。浏览器不会尝试显示 `<script>` 标签中的代码。

要阻止浏览器试图将脚本作为 JavaScript 代码来执行，需要给 `<script>` 标签添加其他多用途互联网邮件扩展 (Multipurpose Internet Mail Extensions, MIME) 类型，而非 `text/javascript` 或 `application/javascript`，如清单 2.8 所示。

清单 2.8 内嵌模板

```
<script type="text/template" id="myTemplate">  
  Hello, <%= firstName %>, how are you?  
</script>
```

将模板包装在 `<script>` 标签中，隐藏模板代码。非 JavaScript MIME 类型不会被作为 JavaScript 代码对待。

如果内嵌模板使用属性方式的绑定语法，则无须做特别处理，此时不需要 `<script>` 标签。同时，由于属性不会显示，用户也不可能在视图渲染之前意外看到绑定代码。

5. 局部模板

如果按需下载模板，则不需要 `<script>` 标签（即使使用了表达式风格）。模板引擎可以直接使用动态获取的模板，这就完全没 `<script>` 标签什么事了。

如前所述，这些按需获取的模板有时作为局部代码或片段来引用。它们并非应用程序加载的初始 HTML 文档的一部分。相反，它们是运行时从服务器端直接加载的代码片段。

现在我们了解了模型、绑定和模板，最后还需了解它们如何集成到视图。下一节将阐述 MV* 框架中的视图概念。

2.2.6 视图

如讨论模板时所述, 像 Knockout 和 AngularJS 这类的框架在模板中使用声明式绑定, 通常采用在 HTML 元素上添加特定属性的方式。在这些框架里, 模板和视图差不多是同一类东西。因此, 当使用这些框架构建视图时, 需要考虑采取内嵌模板还是按需下载的方式。这更像一个设计问题。

在像 Backbone.js 这类代码驱动类型的框架中, 则采取编程方式创建视图。清单 2.9 演示了在线通讯录应用程序的 Backbone 版本视图创建方式。此时先别担心还不太理解其中的代码含义。附录 A 包含了完整的代码清单, 随着你对本书内容的掌握, 可以参考它。

清单 2.9 Backbone.js 的视图创建

```
var Employee = Backbone.View.extend({
  tagName: "li",
  template: _.template(templateHTML),
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  },
  events: {
    "click .remove-entry": "removeEntry"
  },
  removeEntry: function() {
    this.model.destroy();
    this.remove();
  }
});
```

定义元素类型

编译模板

扩展内建 Backbone.js 视图对象

渲染视图, 并将其附加到 DOM

定义单击事件行为

每当视图移除时执行清理操作

Backbone.js 允许我们为视图定义特质 (trait), 诸如其 CSS 类名及其所具有的元素类型。另外, 你还可以按自己的方式自主地定义视图生命周期中的关键阶段, 如视图的渲染和移除阶段。

视图的讨论充实了 MV* 基本概念的内容。理解 MV* 相关概念是非常有益的, 但 MV* 能带给我们什么好处呢? 下一节会告诉你, 使用 MV* 框架可以让 SPA 开发者过得更开心, 代码也会更清晰。

2.3 为什么要用MV*框架

在项目中决定引进任何外部软件都不是一件小事。毕竟你引入了一个依赖。然而话说回来, 如果引入的好处大于代价, 那就是值得的决定。本节阐述使用 MV* 框架的几个关键好处。

2.3.1 关注分离

如前所述, MV* 框架提供一种手段, 基于底层设计模式, 将 JavaScript 对象划分为不同角色。代码的每部分都可以各司其职。

代码的每部分都可以专注于各自职责——这个关注分离的首要概念有利于设计针对特定目的的对象。模型可以专注于数据; 视图可以专注于数据展现; 而诸如控制器、表示器以及视图模型等组成部分则可以解耦模型与视图, 并维持两者间的互相信通。越是将对象聚焦于单一目的, 编码、测试及上线后的更改也就变得越容易。

MV* 框架通过其特定机制, 要求我们遵循特定方式编写代码以达成松耦合目的, 这样就降低了意大利面条式代码产生的可能性。其次, 通过移除内嵌的 JavaScript 和 CSS 代码, 能够尽可能保持 HTML 代码的干净。同时避免了 JavaScript 与 DOM 元素间的紧耦合。

图 2.11 展示了一个 AngularJS 典型样例, 其演示 MV* 如何让我们的代码保持更清晰。右边的 `` 标签反映了左边 INPUT 字段的输入情况。

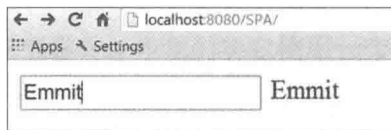


图 2.11 在这个示例中, 当用户在 INPUT 字段中输入内容时, `` 标签将动态更新

我们来创建图 2.11 的示例, 首先看看意大利面条式代码风格, 之后是 AngularJS 方式。清单 2.10 中的 HTML 与 JavaScript 代码是紧耦合方式, 紧耦合意味着从一个函数 / 组件到另一个函数 / 组件的直接引用或调用。也即会把各种结构合为一体。

意大利面条式代码可以运行, 但其难以阅读且导致后续的修改比较痛苦。如果整个单页面应用程序都按这种方式来写, 可以想象维护工作有多么困难。

清单 2.10 紧耦合的 HTML 与 JavaScript 代码

```
<html>
<body>
  <input id="name"
    onKeyUp="document.getElementById('output').innerHTML
      = document.getElementById('name').value">
  <span id="output"></span>
</body>
</html>
```

千万别这么做!

警告 清单 2.10 中的代码能够在 INPUT 字段输入时动态更新 `` 标签的内容, 但这种方式难以阅读和管理。

现在来看看 AngularJS 的实现示例（如清单 2.11 所示）。其抽象出了构建功能所需的大量样板代码。

清单 2.11 AngularJS 的实现示例

```
<html>
<body ng-app>
  <input ng-model="name">
  <span>{{name}}</span>
  <script src="js/thirdParty/angular.min.js"></script>
</body>
</html>
```

AngularJS 框架将 INPUT 字段与
 标签绑定在一起，移除了
HTML 中的 JavaScript 代码

信不信由你，仅需要这么点代码就够了。当然，这只是一个用于演示的小例子而已。尽管使用了像 AngularJS 这样强大的框架，但复杂应用仍具有复杂逻辑。而使用框架的所有理由就是，可以集中精力编写业务逻辑，告别低水平的旧方式。

2.3.2 简化日常任务

MV* 框架还能简化开发者的某些常规任务。举个例子：通过 HTML 标记把列表数据重复打印到屏幕上。这是一项乏味的任务，但却要花费大量代码来完成它。而且，当处理此任务时，我们发现总是一遍遍重复编写相同的代码。

来看看我们的员工通讯录。其中一个功能是能够将某个员工数据添加为列表条目。如果不得不手工编写所有代码，就会发现自己得在一个 JavaScript 循环中为每个条目创建 DOM 元素。实现的代码肯定不漂亮，而且代码极有可能无法为其他任务重用。

MV* 框架将此类脏活从任务中剥离出来。清单 2.12 与清单 2.13 是使用 Knockout 添加列表条目的实现。清单 2.12 描述 HTML 端的处理。出于简洁性考虑，这段代码并不完整，只是添加员工列表条目功能的一部分。附录 A 有相应的完整实现。

清单 2.12 员工列表的 HTML 代码

```
<ul class="entry-list" id="entryList"
  data-bind="foreach: entries">
</ul>
```

使用一个空的 UL 与 foreach 绑定
来迭代构建列表

注意上述 HTML 代码中并不存在任何 JavaScript 代码。清单 2.13 展示了该模板部分对应的视图模型支持实现。

清单 2.13 员工列表的 JavaScript 代码

```
self.entries = ko.observableArray();
self.addEntry = function(e) {
    var newEntry = {
        firstName : self.entry.firstName(),
        lastName : self.entry.lastName(),
        title : self.entry.title(),
        phone : self.entry.phone(),
        email : self.entry.email()
    };
    self.entries.push(newEntry);
};
```

只要用户单击了添加按钮，就会创建一个新的条目

特定 Knockout 数组，保持数组数据与 HTML 同步

将条目添加进数组，之后 Knockout 继续处理其他任务

通过几个巧妙的属性和极简代码，就能完成列表需求。就像这样把脏活、累活从日常任务中移开，极大地简化了程序员的工作。

2.3.3 提升生产率

从开发者的角度来看，能够将时间和精力放在业务逻辑处理上，就意味着生产率提升。当我们决定使用一个外部库或框架时，也就意味着我们正在卸下代码中的维护重负。我们同时在这些框架覆盖的领域应用框架发明者们的技术积累。当然，你也可以自己写一个同样的项目，但却需要付出巨大的努力才能达到已有 MV* 框架实现的水准。

对于大多数的库 / 框架，如果遇到问题，你还能在网上获得社区提供的海量知识。绝大部分的 MV* 框架作者也都有 bug 报告机制。也就是说，会定期对修复代码进行测试和发布，不需要你花大量时间在问题解决上。

如果每件任务都要自己处理，则除了负责自己的业务逻辑，还得为外部库 / 框架提供的所有结构性代码做额外的 bug 修复与测试。

2.3.4 标准化

如本书一再强调的那样，编写一个稳健而清晰、可扩展的 Web 应用程序已经很困难了。而这些困难在单页面应用中还可能堆叠在一块。因此，最不该出现的情况就是，每个团队成员的代码风格都大相径庭。

你希望能够读懂队友的代码，就好像这些代码是你自己写的一样。不然，在腾出时间修改某些代码之前，你将不断浪费时间破译某些“异形”风格的代码。即使是自己独立开发的，一个统一的代码标准也将在回头更改代码之时给你提供帮助。

MV* 库和框架在使用上有必须遵循的明确约定。这将强制要求开发者以一个更加正式、标准的方式来编写应用程序。

2.3.5 可扩展性

如前所述, MV* 框架内在强化了关注分离的概念。反过来, 这也使得项目更具有可扩展性, 因为松耦合的对象往往稍加调整就能够在其他对象中重用。

在 MV* 中, 对象还可以完全交换出来以替换成新功能, 同时不会在项目中产生巨大的连锁反应。这使得项目成长方式更优雅, 因为代码改变的负面影响要小得多。

我们已经了解了一些 MV* 核心特性, 但哪种风格的 MV* 框架看起来更容易或更难使用呢? 开发者必须做出判断。一些开发者不喜欢诸如 MVVM 中绑定与 HTML 页面任意混合的声明式风格, 但其他一些人则青睐该风格不用过多样板代码就能够创建视图。

虽然最终不得不就框架选择做出决定, 但 2.4 节将提供一些帮助决策的参考意见。

2.4 框架选择

决定在项目中引入 MV* 库或框架之后, 将有许多库或框架可供选择。即使你对风格有特殊偏好, 相应的选择依旧繁多。这里列出一些本书编写时流行的客户端 MV* 框架, 以供参考:

- AngularJS (<https://angularjs.org>)
- Agility.js (<http://agilityjs.com>)
- Backbone.js (<http://backbonejs.org>)
- CanJS (<http://canjs.com>)
- Choco (<https://github.com/ahe/choco>)
- Dojo Toolkit (<http://dojotoolkit.org>)
- Ember.js (<http://emberjs.com>)
- Ext JS (www.sencha.com/products/extjs)
- Jamal (<https://github.com/adcloud/jamal>)
- JavaScriptMVC (<http://javascriptmvc.com>)
- Kendo UI (www.telerik.com/kendo-ui)
- Knockout (<http://knockoutjs.com>)
- Spine (<http://spinejs.com>)

如你所见, 我们确实有不少选择。这些都是库或框架。如果想选择一个并非无所不包的别的框架来处理其他特性 (如路由和视图管理), 面临的选择几乎一样多。短短几年中, 我们可以选择的库 / 框架从相对较少发展到今天让你眼花缭乱的程度。

要评选出框架孰优孰劣是件毫无意义的事情。各种观点实在是见仁见智。而且，由于各种框架特性风格都不尽相同，就很难将它们放在一个层面上进行比较。尽管如此，我们还是可以归纳出一些在选择库 / 框架时该记住的考虑点：

- 是点菜方式还是一站式购物风格——这完全是个人想法，你想要一个无所不包的框架？还是更需要一个尽可能小、只关注某些核心功能的框架？两者都有争论。有些人的考虑重点并不是为缺失特性选择其他库，其更担心依赖过多及失败潜因。而这样的话，你就得熟悉各种不同工具，光靠一种工具显然行不通。另一些人则相反：使用无所不包的解决方案，“把鸡蛋放进一个篮子里”。但如果对这个框架的支持一旦停止，寻找替代方案的工作将比采用单一功能支持库的“点菜”方式来得困难得多。诸如 Knockout 和 Backbone.js 这类精练型框架，非常适合“小即优”的方案，但也意味着编写 SPA 应用程序时还得寻求其他的补充功能。像 Ember.js、kendo UI 及 AngularJS 这类框架则添加了各种功能，却又隐藏了大量底层细节，这让那些希望拥有更多控制权的开发者“不太感冒”。
- 许可与支持——预算是现实的。对于项目：你有钱购买框架吗？还是说需要的产品是免费的？公司要求购买商业产品吗？公司要求为项目涉及的所有软件购买一定程度的支持吗？项目是关键任务吗？要求 bug 修复及功能更迭的周期最小？
- 编程风格偏好——Knockout 和 Kendo UI 归属 MVVM 阵营。其他的，包括 Backbone.js 和 Ember.js，更接近 MVC 和 / 或 MVP。AngularJS 有点像 MVVM，但仍保留了一些类 MVC 特性。它们都可以用于创建大型、稳健的应用。在实践了几种不同风格之后，你的选择取决于个人对框架风格的偏好。
- 学习曲线——对某些人而言这点可能不那么重要，因为只要给你足够的时间就可以掌握任何框架。但也有某些人学习起来绝对要比别人吃力得多。此外，项目进度紧张的情况下你也可能没有太多时间。
- bug 与修复率——所有软件都有一定数量的 bug，现实使然。但可以把软件历史上高危 bug 的比例作为选择决策的依据之一。同时，bug 修复的速度有多快？如果大量重要 bug 在好长一段时间后还未得到处理，那估计就要亮红灯了。
- 文档——库或框架的文档好不好？到目前为止的更新情况？某些 MV* 框架提供免费的在线视频及交互式训练。同时还要考虑代码示例是否匹配 API 文档。
- 成熟度——我们可以通过成熟度判断框架的好坏。如果框架相当成熟，则我们可以对其可预见的未来充满憧憬。如果框架是新创建的，就很可能仍在经历“成长的烦恼”。若是应用并不重要，则采用此类框架也许是可以接受的。

然而，若是框架变化无常，那就几乎不可能用它来创建关键应用。

- 社区——这方面有时候容易忽视，但如果你计划将第三方软件纳入到依赖中来，而该软件伴有一个庞大社区则是好事情。人多力量大。开发者迟早会遇到文档中未涉及的问题，通过在线论坛和博客获取帮助可以应一时之需。
- 灵活度——对于常规任务，如创建对象及列表、发送 / 接收 / 处理请求等，灵活度有多大？其是否通过严格的约束条件限制程序行为（或者是否与你的程序发生冲突）？它与其他库或框架的兼容性如何？
- 概念性验证（proof of concept, POC）——一旦将选择范围缩小到少数几个目标，就该为每个选择项进行概念性验证，以在实践中获得初步感知。在实际项目中总是会遇到意外情况，并迫使开发者寻求解决方法，这是开发者的本能。但通过实施一个简单的、至少带有基本 CRUD 功能的概念性验证，将能够辅助决策。最好 CRUD 操作再包含一个对象列表，这样同时还可以获取集合管理复杂度方面的感知。

如你所见，在选择一个 MV* 框架的时候有许多考虑因素。但我们现在熟悉了 MV* 传统设计模式及基本核心概念，同时还了解了 MV* 库 / 框架设计上的一些差异。综合这些及以上考虑点，我们就能够在时机成熟时做出更明智的选择。

2.5 挑战环节

现在进入挑战环节，看看你对本章的掌握程度，检查一下自己能否通过单向绑定创建简单视图。假定你所在的当地图书馆想要着手提供在线电子书籍，并已向社区寻求技术支持。图书馆已将其第一批图书转换成了电子格式，但需要 Web 开发者搭建一个电子书网站。请选择本章涉及的三个 MV* 框架中的任意一个（或者别的，只要你喜欢），创建一个包含书籍列表的视图。视图应该遵循以下版式：

- 头部——页面头部应该包括图书馆名称、地址及电话号码。同时还应该显示用户登录名。对用户而言，创建一个 JavaScript 变量并使用用户名称作为它的值。之后创建一个简单的绑定，在视图的头部显示该值。
- 页面体——在 JavaScript 代码中，创建一个书籍对象的列表（书籍名称、作者以及简要描述）。在页面体里，选择一个迭代打印列表中每本书籍的绑定。

2.6 小结

理解以下内容将帮助我们不断前行。我们来回顾一下：

- 传统设计模式对 MV* 库 / 框架(MVC: 模型 - 视图 - 控制器, MVP: 模型 - 视图 - 表示器, MVVM: 模型 - 视图 - 视图模型)有着深远影响。

- 模型代表了数据。在 MVVM 模式中，对象主要是数据。在另两种模式中，模型还包含了其他种类的逻辑，如数据管理逻辑。
- 视图代表了应用中用户所见并与之交互的部分。
- MV* 中的第三个组成部分：控制器、表示器或者视图模型，一定程度上承担着中间对象的角色，保持模型与视图的解耦，但模型与视图间有交互联系。
- 每个模式必须能适应其环境。MV* 库 / 框架作者必须灵活运用各种传统模式，以创建能在浏览器装置中工作的解决方案。
- 了解了模型、绑定、模板以及视图等基本 MV* 概念。
- MV* 框架选型时应该牢记在心的各种考虑点：是点菜方式还是一站式购物风格，以及许可与支持、编程风格偏好、学习曲线、bug 与修复率、文档、成熟度、社区支持以及灵活度，等等。
- 当框架选型范围缩小至两三种的时候，对每种框架进行概念性验证，为后续在项目中具体应用事先获得初步感知。

JavaScript模块化

本章内容

- 模块的定义
- 为什么要在 SPA 应用中使用模块
- 回顾 JavaScript 作用域及对象创建
- 模块语法与机制分析
- 模块加载与 AMD 模块介绍

JavaScript 是一门强大而灵活的语言，其已经成为交互式 Web 应用程序开发的事实标准。即便如 Amazon、Google 和 Walmart 这样的大公司也都在使用它。JavaScript 很容易掌握，也很容易在 Web 页面中使用。因此，一门自由而易用的语言流行起来也就不足为奇了。

然而，语言的友好度是一把双刃剑。在任何如 JavaScript 这样的动态语言中，稍不小心就很容易把代码搞得一团糟。在 Google 技术讲座的一个报告中，著名的开发者、架构师、作者及演讲者 Douglas Crockford 指出 JavaScript “拥有一些非常棒的编程语言设计理念”（请参看：<http://googlecode.blogspot.com/2009/03/doug-crockford-javascript-good-parts.html>），可他还补充道：JavaScript 还“拥有一些非常糟糕的理念”。

幸运的是，遵循最佳实践及验证过的设计模式，就能让你始终把握项目前进的方向。在现代 Web 应用开发中有一种模式特别有用——模块模式。模块提供了一种代码构建的优雅方式。其还能够避开非常多的问题，没有了它，SPA 应用程序构建过程将问题不断。本章将深入介绍模块相关内容，并告诉你为什么要在 SPA 应用构建中使用模块模式。

3.1 模块概念

通常，模块（Module）是指某个更大结构的一部分或组件。然而，模块术语依据不同上下文，甚至在软件开发范畴内，都可以有不同含义。有时我们会听到人们在一般意义层面讨论模块。比如，他们可能说“支付模块”或者“旅行计划模块”。这时候模块意指整体特性是完全没什么问题的。而在我们特指 JavaScript 代码模块的时候，模块就代表一个函数——一个通过模块模式创建的特定函数。

基本模块模式

```
var moduleName =  
(function() {  
    // 私有变量、函数  
    return {  
        // 公有函数  
    };  
})();
```

一个全局变量，作为模块的命名空间

模块内部代码

返回一个带有公有函数的对象字面量，该对象作为模块的公有 API 提供者

别担心看不懂这里的陌生语法。3.3 节会进一步详尽阐述。现在，继续模块基本编程的学习之旅，先来回顾一些概念。

3.1.1 模块模式概念

暂停一下前进的脚步，来看一些模块模式的概念：

- **命名空间**——命名空间是一种为一组相关成员提供具体作用域的方式。尽管命名空间当前并非 JavaScript 语言的一部分，但仍可以通过为更大作用域内的任何变量（如全局变量）分配模块函数来达到同样效果。
- **匿名函数表达式**——函数表达式是表达式的一部分，且不以 `function` 关键字开头。如果函数表达式未命名，则被称为匿名函数表达式。模块体包含在一个匿名函数表达式当中。
- **对象字面量**——JavaScript 提供了一种创建对象的快捷方式，被称为字面量标记法，其通过花括号声明一个对象，它的 Property 用键值对来表示。对象字面量会在 3.3.2 节进一步解释。

- 闭包——通常，当函数执行完毕，在其中创建的任何局部变量的生命周期也就宣告结束。闭包则是一个例外情况，当函数包含了外层作用域的变量引用时，例外就发生了。在 JavaScript 中，每个函数都有自己的作用域 Property——其引用外层作用域变量。每个函数也都有一个外层作用域，即使外层作用域恰好就是全局作用域（所以从技术角度来讲，所有函数都是闭包）。闭包对于我们的讨论内容是非常重要的，因为即使模块模式的外层函数立即执行完毕，只要模块仍在使用，外层函数返回语句所引用作用域链之上的任何对象或值，都无法被垃圾回收。

3.1.2 模块结构

模块结构巧妙地使用一个函数作为封装其逻辑的容器。这是可能的，因为在模块中局部声明变量及函数可以避免模块外部直接访问它们。模块内部功能的访问可以通过返回语句暴露的内容来控制（如图 3.1 所示）。

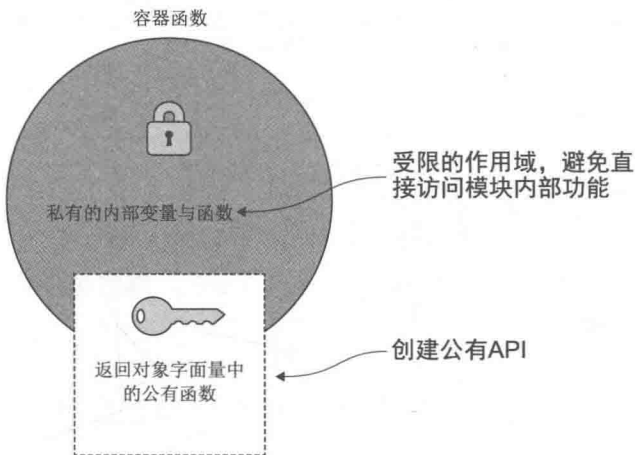


图 3.1 在模块中，外层函数封装模块功能，模块之外不能直接访问模块内部的变量和函数。通过返回的对象字面量中的函数来控制访问

为了弄明白如何使用这样一个新奇构造，我们来看一个简单函数，并使用模块模式重写它。以下函数相加两个数值并返回结果：

```
var num1 = 2;  
function addTwoNumbers(num2) {  
    return num1 + num2;  
}
```

```
alert( addTwoNumbers(2) );
```

弹出结果“4”

清单 3.1 展示了相同功能，现在用重用模块的方式重写它。请注意模块模式抽象出了模块的内部功能。函数调用者不必关心模块内部实现，其只需知道调用公有接口的 `addTwoNumbers()` 函数，就能弹出正确结果。3.3 节会完整介绍实现机制。

清单 3.1 以模块模式重写两值相加函数

```
var numberModule = (function() {
  var num1 = 2;

  function addNumbersInternally(num2) {
    return num1 + num2;
  }

  return {
    addTwoNumbers : function(num2) {
      alert(addNumbersInternally(num2));
    }
  };
})();

numberModule.addTwoNumbers(2);
```

私有变量
num1 包含了
初始状态

私有函数
addNumbersInternally()
负责数值相加操作

通过调用
addNumbersInternally(),
弹出计算结果

就像这样调用模块的
公有函数

3.1.3 揭示模式

模块模式的吸引力之一就是在模块内部功能与公开功能之间划定了清晰的界限。有一种频繁用到并能够让划分更加清晰的改良版模块模式——揭示模式 (revealing module pattern)。

揭示模式由 Christian Heilmann 提出，这个稍加改良的版本使得公有接口的使用更加清晰。其实现思路是：将任何 API 所需代码移到内部，将公有函数作为纯粹指向内部代码的指针，并只暴露该公有函数。清单 3.2 实现了与清单 3.1 相同的功能，但用揭示模式重写了。

清单 3.2 揭示模式

```
var numberModule = (function() {
  var num1 = 2;

  function addTwoNumbers(num2) {
    alert(num1 + num2);
  }

  return {
    addTwoNumbers : addTwoNumbers
  };
})();

numberModule.addTwoNumbers(2);
```

公有 addTwoNumbers() 函数是一个指向私有 addTwoNumbers() 函数的指针

使用模块的方式不变

你可以看到，公有 API 非常清晰，而且更具可读性。我们将在本章后续示例中都使用揭示模式。现在我们对模块有了基本了解了，接下来讨论开发者为什么应该多考虑模块化编程。

3.2 模块化编程的意义

模块模式创建于 2003 年，并通过 Douglas Crockford 的演讲得以普及。模块结构遵循以下原则：

- 保持私有代码部分只能在模块中使用。
- 创建公有 API，以控制模块功能的访问。

我们也可以分配单个命名空间给模块及其关联子模块，以减轻全局命名空间的污染。可是，为什么这些工作如此重要呢？初次遇上模块模式的语法，你可能不会喜欢它，甚至会感到有点恐怖。然而一旦理解了这种独特语法，我们就能充分理解为什么要在 SPA 应用中使用模块模式了。

3.2.1 避免命名冲突

变量及函数命名冲突甚至在小型应用中都会发生。大型应用中出现的概率将成倍放大，因为所有的 JavaScript 变量和函数都塞到全局命名空间中了。回到 Douglas Crockford 的演讲内容，他提到“迄今为止，JavaScript 语言最糟糕的设计就是全局变量”，他特别指出全局代码中的命名冲突问题。当所有的对象都位于全局命名空间中时，它们就共享同一个作用域。而在同一个作用域中用相同名称声明变量或函数，并不会产生错误，最后声明的变量或函数将覆盖前面的声明。这将导致不可预期的结果，并很难被发现。我们来看看下面的例子。

假设你有一家初创公司，叫 *Simpler Times Gourmet*，你打算搭建一个 SPA 应用来销售你的美食（第 4 章将讨论视图创建，因此当下我们先使用一些 `<div>` 标签，以将重点放在代码上）。首次下单有慷慨的 25% 折扣——你希望通过这样的措施来吸引新顾客。你也想回报老客户；但让所有人都享受 25% 的折扣，就无法实现首年的赢利目标。因此，现在你只打算向新顾客提供折扣。

刚开始一切都很顺利。老客户进来将看到一条简单的欢迎消息，只有新顾客会看到 25% 的折扣消息。为了确保回头客户看到欢迎消息，我们采取硬编码方式。图 3.2 演示了界面效果。



图 3.2 老客户只会看到一条简单的欢迎消息，看不到折扣消息

清单 3.3 和清单 3.4 则是相关代码。此时的设计未使用模块模式。DOM 准备就绪时，我们可以调用 `getWelcomeMessage()`（定义在 `welcomeMessage.js` 中），以返回合适的问候消息。之后问候消息通过 jQuery 的 `html()` 函数插入到内容 `<div>` 标签里，这样，顾客就能看到问候消息了，如以下清单所示。

清单 3.3 index.html

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="css/default.css">
</head>
<body>
  <div class="siteMain" id="container">
    <div id="header" class="header">The Simpler Times Gourmet</div>
    <div id="content" class="content"></div>
  </div>
  <script src="js/thirdParty/jquery.min.js"></script>
  <script src="js/welcomeMessage.js"></script>
  <script>
    $(document).ready(function() {
      $("#content").html(getWelcomeMessage())
    });
  </script>
</body>
</html>
```

`getWelcomeMessage()`
返回相应的问候消息

请特别关注清单 3.4 中 `getStatus()` 的结果方式。`getStatus()` 是否返回 `existing` 决定了显示什么样的消息。

注意 jQuery 的 `.ready()` 函数保证其中定义的函数在 DOM 准备就绪后才可执行。

清单 3.4 welcomeMessage.js

```
var customerLoggedIn = true;
var customerName = "Emmitt";
var isNewCustomer = false;
```

```
function getStatus(){
    if(customerLoggedIn){
        if(isNewCustomer){
            return "new";
        } else {
            return "existing";
        }
    } else {
        return "unknown";
    }
}
```

getStatus() 检查顾客是新顾客还是老顾客

```
function getWelcomeMessage(){
    if(getStatus() !== "unknown" ) {
        if(getStatus() === "existing" ) {
            return "Hi again " + customerName
            + ", glad to see you back!";
        } else {
            return "Welcome " + customerName
            + " - 25% off entire purchase!";
        }
    } else {
        return "Sign up for some great gourmet deals!";
    }
}
```

getWelcomeMessage() 有赖于 getStatus() 的结果决定返回什么样的消息

目前一切都正常运作。我们接着添加一个 JavaScript 文件来显示购物车的当前状态消息，这看起来应该没什么大问题。但当我们添加接下来的代码之后，功能测试失败了。购物车的状态消息没错，但无关乎顾客状态，就算是老客户也能够意外收到 25% 的折扣消息。图 3.3 展示了新文件添加之后的欢迎页面。



图 3.3 每个人都意外收到了一个给力折扣的消息

当对新文件进行独立的单元测试时，看上去好像一切正常。而将其添加至站点时为何会出问题呢？如清单 3.5 所示，问题的根源在于产生了命名冲突。开发者无意间包含了一个名为 getStatus() 的函数。由于最初的 getStatus() 与新的 getStatus() 都定义在全局命名空间中，那么就造成了命名冲突。

清单 3.5 shoppingCartStatus.js

```
var cartActiveItems = [];  
  
function getStatus(){  
    if(cartActiveItems.length > 0){  
        return "pending";  
    } else {  
        return "empty";  
    }  
}  
  
function getStatusMessage(){  
    if(getStatus() === "empty" ) {  
        return "Cart is empty";  
    } else {  
        return "Cart (" + cartActiveItems.length + " items)";  
    }  
}
```

添加 getStatus() 函数导致了
命名冲突

不幸的是,我们无法在浏览器中看到错误信息。由于新文件在最后包含进来(如清单 3.6 所示),新的 getStatus() 函数覆盖了前面的一个(如图 3.4 所示)。就好像第一个 getStatus() 函数从未存在过一样。

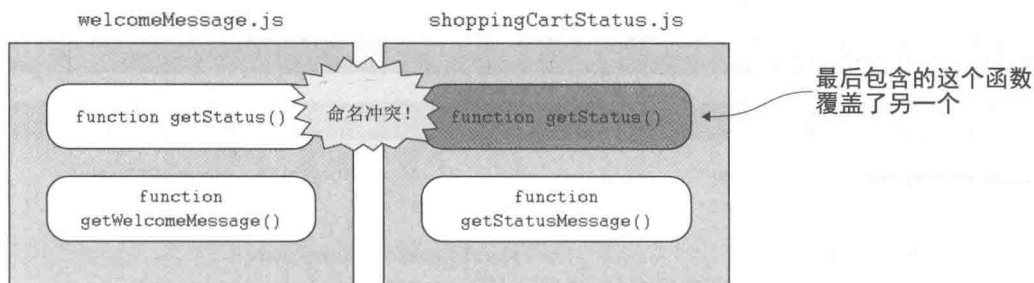


图 3.4 不用模块限制函数作用域,把函数一股脑地放进全局作用域,将很容易导致命名冲突

现在 getWelcomeMessage() 使用错误的 getStatus() 函数来创建问候消息。在这个例子中,我们感兴趣的事情是:新文件中的 getStatus() 函数本来跟欢迎消息没有任何关系,但仍对结果产生重大影响。

清单 3.6 index.html

```
<!DOCTYPE html>  
<html>  
<head>  
<link rel="stylesheet" href="css/default.css">  
</head>
```

```

<body>
  <div class="siteMain" id="container">
    <div class="header" id="header">
      <div id="banner" class="banner">
        The Simpler Times Gourmet
      </div>
      <div id="cart" class="cart"></div>
    </div>
    <div id="content" class="content"></div>
  </div>
  <script src="js/thirdParty/jquery.min.js"></script>
  <script src="js/welcomeMessage.js"></script>
  <script src="js/shoppingCartStatus.js"></script>
  <script>
    $(document).ready(function() {
      $("#content").html(getWelcomeMessage());
      $("#cart").html(getStatusMessage());
    });
  </script>
</body>
</html>

```

标头现在包含了显示购物车状态的区域

第二个getStatus()覆盖了第一个

getWelcomeMessage()使用错误的getStatus()创建错误的消息

现在来考虑一下如何纠正这个错误。我们将使用模块来重写该应用程序。前面已尽可能让每个函数的逻辑都尽可能接近其原意，因此重写过程很容易接续下去。清单 3.7 展示了新的 index 页面。

清单 3.7 index.html

```

<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="css/default.css">
</head>
<body>
<div class="siteMain" id="container">
  <div class="header" id="header">
    <div id="banner" class="banner">
      The Simpler Times Gourmet
    </div>
    <div id="cart" class="cart"></div>
  </div>
  <div id="content" class="content"></div>
</div>
<script src="js/thirdParty/jquery.min.js"></script>
<script src="js/STGourmet.js"></script>
<script src="js/customer.js"></script>
<script src="js/welcomeMessage.js"></script>
<script src="js/shoppingCart.js"></script>
<script>
  STGourmet.init();
</script>
</body>
</html>

```

增加了两个新文件进来

STGourmet.init()是整个应用程序的单一起点

你会立即观察到新变化，页面代码中添加了一些新的 JavaScript 模块。首先，原来调用 jQuery 来更新 <div> 内容的方式，已经替换为模块调用方式（第一处模块调用）。应用程序现在通过单独调用 STGourmet 模块的唯一公有函数 init() 来启动。

其次，除了 jQuery，代码中共包含了 4 个 JavaScript 文件。我们将用模块方式重写原有文件中的代码。同时，这里也出现了一些新模块。请注意加载顺序，如果使用标准的 <script> 标签加载文件，模块必须按顺序包含进代码中，以便任何后续需要的变量或函数事先得以加载。STGourmet 是主要的模块，因此第一个加载。customer 的函数在后续会用到，因此第二个加载。

第 9 章会讲述合并及压缩代码以换取更佳性能。3.4 节涉及调用第三方库 RequireJS 来异步加载模块。现阶段我们关注标准的文件包含方式，它是同步加载的，因此，加载顺序非常重要。

注意 当模块通过 <script> 标签加载时，其加载顺序非常重要！模块应按现实所需顺序加载。

清单 3.8 展示了我们的第一个模块。我们定义了 STGourmet，并将其作为整个应用程序的单一全局变量。这样就构造出了单一的命名空间，整个应用程序都在其之下。再次说明，别担心目前的模块模式语法，3.3 节会具体讨论。让我们继续关注如何通过模块来解决命名冲突问题。

清单 3.8 STGourmet.js

```
var STGourmet = (function($) {  
    function init() {  
        $(document).ready(function() {  
            STGourmet.shoppingCart.displayStatus();  
            STGourmet.welcomeMessage.showGreeting();  
        });  
    }  
    return {  
        init : init  
    };  
})(jQuery);
```

STGourmet 成为应用程序的命名空间

STGourmet 的两个子模块：shoppingCart 和 welcomeMessage

任何 return 中的内容都是公有的，因此 init() 是我们唯一的公有函数

jQuery 在尾部导入模块，并在顶部设其别名为 "\$"

由于只有在返回语句中声明的函数才是公有的，因此，模块的 init() 是其唯一公有函数。在 index.html 中，我们通过调用 STGourmet.init() 启动应用程序。接着调用 shoppingCart 子模块的 displayStatus() 函数和 welcomeMessage 子模块的 showGreeting() 函数。

清单 3.9 包含了 `customer` 模块。由于三个私有的硬编码变量是购物车或问候功能的一部分，因此，相关代码就归到其所属的 `customer` 模块。

清单 3.9 `customer.js`

```
STGourmet.customer = (function() {

    var customerLoggedIn = true;
    var customerName = "Emmit";
    var isNewCustomer = false;

    function isLoggedIn() {
        return customerLoggedIn;
    }

    function getName() {
        return customerName;
    }

    function isNew() {
        return isNewCustomer;
    }

    return {
        isLoggedIn : isLoggedIn,
        getName : getName,
        isNew : isNew
    };

})();
```

将 `.customer` 添加给 `STGourmet` 以创建新的子模块

不在返回语句中的代码都是私有的，模块外部无法访问

任何定义在返回语句中的内容都是公有的

子模块

我们做的唯一一件事情就是在这个文件中创建了一个子模块。这是使用模块模式创建应用程序子模块的普遍做法。这种方式允许我们将代码分割进模块，而代码仍属于相同命名空间的一部分。

请注意我们并未用 `var` 声明子模块。这并非意外。实际上，我们通过圆点记法将一个 **Property** 添加给了之前定义的 `STGourmet` 函数对象。通过将 `.customer` 添加给 `STGourmet` 对象，我们就创建了名为 `customer` 的子模块（如图 3.5 所示）。



图 3.5 通过圆点记法而非 `var` 来声明一个子模块。我们要做的事情就是添加一个 **customer** **Property**，该 **Property** 本身包含一个模块

清单 3.10 包含 welcomeMessage 子模块。如你所见，其包含的逻辑跟前面是一样的，只是把代码包装进了模块模式。

清单 3.10 welcomeMessage.js

```
STGourmet.welcomeMessage = (function() {
    function getStatus(){
        if(STGourmet.customer.isLoggedIn()){
            if(STGourmet.customer.isNew()){
                return "new";
            } else {
                return "existing";
            }
        } else {
            return "unknown";
        }
    }

    function getWelcomeMessage() {
        if(getStatus() !== "unknown" ) {
            if(getStatus() === "existing" ) {
                return "Hi again " + STGourmet.customer.getName()
                + ", glad to see you back!";
            } else {
                return "Welcome "
                + STGourmet.customer.getName()
                + " - 25% off entire purchase!";
            }
        } else {
            return "Sign up for some great gourmet deals!";
        }
    }

    function showGreeting(){
        $("#content").html(getWelcomeMessage());
    }

    return {
        showGreeting : showGreeting
    };
})();
```

← welcomeMessage
子模块

最后，清单 3.11 包含了 shoppingCart 子模块。

清单 3.11 shoppingCartStatus.js

```
STGourmet.shoppingCart = (function() {
    var cartActiveItems = [];

    function getStatus(){
        if(cartActiveItems.length > 0){
```

← shoppingCart
子模块

```
        return "pending";
    } else {
        return "empty";
    }
}

function getStatusMessage(){
    if(STGourmet.customer.isLoggedIn()
    && getStatus() === "empty" ) {
        return "Cart is empty";
    } else {
        return "Cart ("
        + cartActiveItems.length
        + " items)";
    }
}

function displayStatus(){
    $("#cart").html(getStatusMessage());
}

return {
    displayStatus : displayStatus
};

})();
```

现在，我们所有的代码都按模块模式组织好了，并避免了命名冲突。来看看此时应用程序的模样（如图 3.6 所示）。



图 3.6 通过采用模块模式，我们的欢迎消息又正确显示了

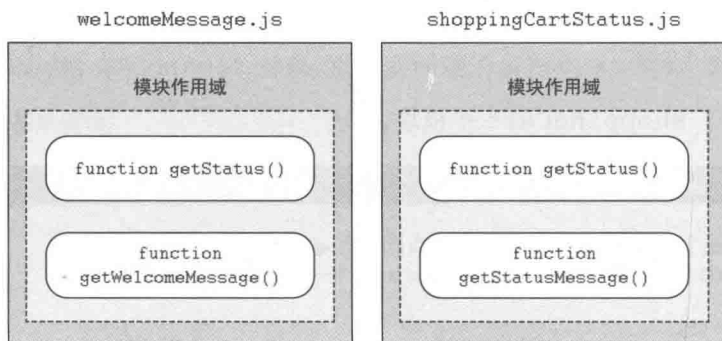


图 3.7 现在的两个 `getStatus()` 函数存在于各自模块中，命名冲突问题解决了

采用模块模式，我们可以很容易地编写避免命名冲突的代码。现在两个 `getStatus()` 函数可以融洽相处了，各自功能满足预期要求（如图 3.7 所示）。

如你所见，模块模式能够让你按自己意愿命名变量或函数，而不用担心不同模块代码间会产生命名冲突问题。当你的项目规模和复杂度不断增加时，这个细节上的小小变化将发挥出关键作用。

3.2.2 保护代码完整性

在某些语言中，访问某部分应用程序代码可以通过诸如 `public` 或 `private` 这样的访问修饰符来控制。而在 JavaScript 中 `private` 是保留关键字，在本书写作时尚未开放使用，因此无法明确声明某个对象属性为私有的。但如前所述，我们仍然可以通过模块模式限定变量和函数的访问权限。这是可能的，因为在某个函数里声明的变量和函数，意味着它们的作用域已限制为容器函数。这种限制访问某部分模块代码的能力将阻止其他代码直接改变其内部状态，维系模块内部正常运作，并避免模块数据遭到肆意修改导致违背其预期目的。

让我们来看一个基本的计数器示例。这是一个简单的函数，每次调用它值都会加 1，其不存在什么陷阱。

```
var count = 0;

function incrementCount(){
    ++count;
}

function printCount(){
    console.log("Count incremented: " + count + " times");
}

function displayNewCount(){
    incrementCount();
    printCount();
}
```

正确增加了 count 变量的值，
并打印 “Count incremented:
1 times”

当前的代码运行正常。但却没有任何手段来阻止开发者直接修改 `count` 变量的值——尽管对于维系 `incrementCount()` 正常工作而言，避免直接修改 `count` 变量是非常重要的。我们假设程序员想要修改 `displayNewCount()`，在 `count` 为 1 时打印信息以 *time* 结尾，否则以 *times* 结尾。如果允许函数修改 `count` 变量值，看看会发生什么：

```
var count = 0;

function incrementCount(){
    ++count;
}

function printCount(){
```

```

    if(count === 1) {
        count = count + " time";
    } else {
        count = count + " times";
    }
    console.log("Count incremented: " + count);
}

function displayNewCount(){
    incrementCount();
    printCount();
}

```

变量 count 变成了字符串，这是非法的

第一次会打印 “Count incremented: 1 time”，但后续则打印 “Count incremented: NaN times”

通过允许直接操作 incrementCount() 函数所用数据，printCount() 函数导致 count 的值变成了 NaN（非数字），这在业务逻辑中是错误的。

现在用模块模式来重写代码。在清单 3.12 中，我们只允许 incrementCount() 的消费者通过计数器模块公有接口访问当前 count。这种方式有效地阻止了直接操作变量 count。

清单 3.12 通过模块化设计重写代码

```

var Counter = (function() {
    var count = 0;

    function incrementCount() {
        ++count;
    }

    function getCount() {
        return count;
    }

    return {
        incrementCount : incrementCount,
        getCount : getCount
    };
})();

```

count 变量是私有的，不再能够直接访问了

由模块的公有 API 控制 count 变量的访问

```

Counter.displayUtil = (function() {

    function printCount() {
        var count = Counter.getCount();
        if(count === 1) {
            count = count + " time";
        } else {
            count = count + " times";
        }
        console.log("Count incremented: " + count);
    }

    function displayNewCount() {

```

printCount() 函数必须通过主模块的 getCount() 才能获取 count

```
        Counter.incrementCount();  
        printCount();  
    }  
  
    return {  
        printCount : printCount,  
        displayNewCount : displayNewCount  
    };  
  
})();
```

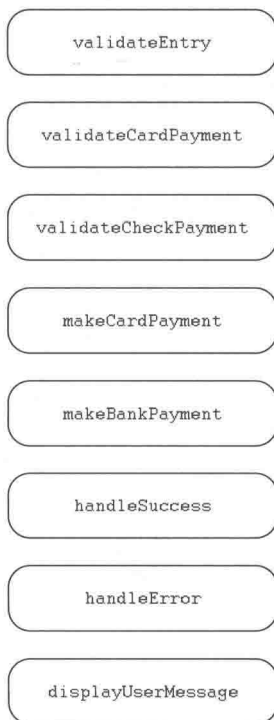
通过公有 API 来调用
displayNewCount() 以
正确增加值, 同时通过
公有 API 打印信息

正确编写安全的代码难度很大。如果你无法阻止代码内部逻辑的错误使用, 编写安全代码的希望就愈发渺茫。模块模式提供了一种管理内部代码访问性的方式。

3.2.3 隐藏复杂性

当讨论隐藏编程复杂度时, 并非说我们想讨论如何保守秘密或者如何增加安全性。我们要讨论的是以下两种方式的差异性: 通过大量全局函数实现具体功能的复杂逻辑, 以及将复杂逻辑放至内部并只通过公有接口暴露开发者所需功能。后者减少了混乱, 并使得函数调用更加清晰起来, 以正确使用应用功能。图 3.8 阐述了带有公有 API 的模块, 能够让其他开发者更容易理解我们的代码。

哪一个函数
是入口点?



VS.



图 3.8 模块模式让其他开发者更清晰地使用我们的代码

通过隐藏模块的复杂性，就能够让团队成员轻松（通过公有 API）使用模块，而不需要了解其内部代码的个中逻辑。

3.2.4 降低代码改变带来的冲击

在模块内部组织代码逻辑也就意味着其他程序员只需针对公有接口编程。只要公有 API 行为保持不变，内部代码也就可以随意调整，且不会强制应用程序其他部分（指外部代码）跟着调整。减少系统其他部分的变化，能够降低代码维护成本。

作为模块开发者，可以更加自由地修改代码。如果 API 协议保持不变，因疏忽而引入无关代码的可能性将大大降低。

3.2.5 代码组织

让一切事物时刻保持井井有条，能够让我们生活更轻松、更有效率。编程过程也是这个道理。除非你使用 ECMAScript 6 版本的 JavaScript，否则无法正式声明类、模块或子模块的代码。尽管如此，将一组变量和函数定义为功能性单元的一部分，则无疑是个好做法。

模块模式授予你一种方式，可以将代码从全局命名空间搬出来，并以一种更具意义的方法来组织这些代码。你可以站在整体功能的高度来着手考虑代码，而非把功能一个个独立开来考虑（如图 3.9 所示）。

最后，将代码重构进一个组织良好的单元，会在代码重用、维护及升级等方面带来更高效率。

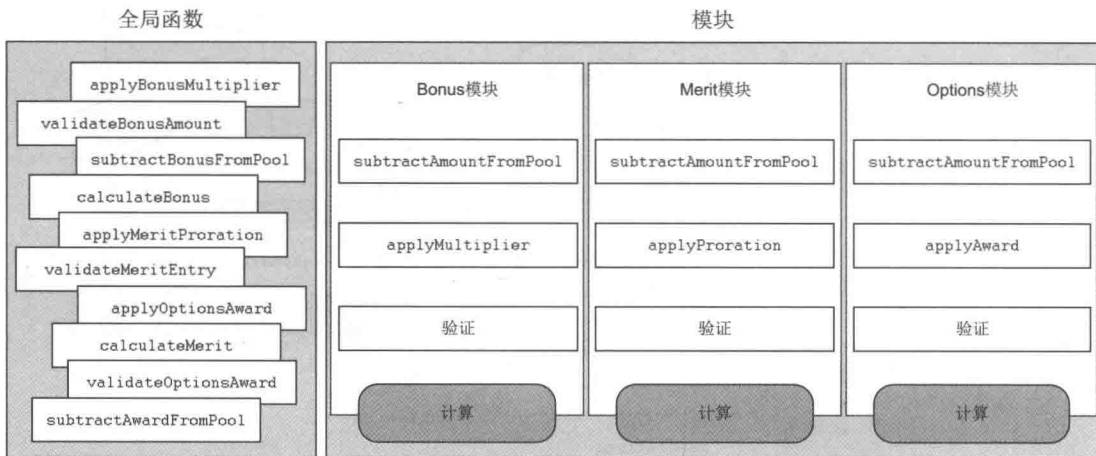


图 3.9 模块模式能够帮助我们将代码组织进功能性单元，而非全局命名空间里的一个个独立函数

3.2.6 模块模式的不足

很不幸，创建复杂代码的过程没有“银弹”。对于现代 Web 应用程序的开发而言更是如此。除了模块模式提供的种种好处，你也需要知道它的一些限制：

- **测试**——有些人抱怨无法对模块中的私有函数进行单元测试。另一些人则认为通过模块公有 API 进行单元测试更为有效。个中存在着争议。尽管前者将无法对模块中的私有函数进行单元测试视作缺点，但后者却认为如果要测试私有函数，就应该考虑是否将这些函数公有化。争论点就在于单元测试是否应该设计成对对象接口的测试，而非对对象内部私有代码进行测试。
- **扩展对象**——JavaScript 程序员习惯于仅需添加操作就能随意扩展对象，这也证明了 JavaScript 强大而灵活的特性。但如前面所见，模块外部是无法访问模块内部对象及非公有 API 部分的。因此，添加一个新 Property 或方法到模块内部的私有对象上，并不能让它立即生效。但话又说回来，这算不算缺点全靠你如何看待它。前面提到，这同时能够保护模块核心功能的完整性。

3.3 模块模式剖析

迄今为止，我们已经掌握了大量模块模式相关知识，并知道如何使用它。但我们尚未深入理解其结构。理解其语法为什么长成这样，有助于在模式模块的应用过程中更加接纳它。本节将深入剖析模块模式，以了解它的运作机理。

先来回顾一下最初的样板结构：

```
var moduleName = (function() {  
    return {  
    };  
})();
```

这是模块模式的基本骨架。

3.3.1 可访问性控制

JavaScript 只有两种作用域类型：局部和全局。在一个函数里声明的变量或函数属于局部作用域（私有），在任何函数之外声明的变量及函数则属于全局作用域（公有）。由于我们无法明确标识变量或函数是公有的还是私有的，因此，只能依靠它们的作用域来实现访问限制。这真是小特性解决了大问题。

JavaScript 构建私有性的唯一方式就是在函数中声明局部变量和函数。图 3.10 突出显示了模块模式的匿名函数，其将模块功能内部化。

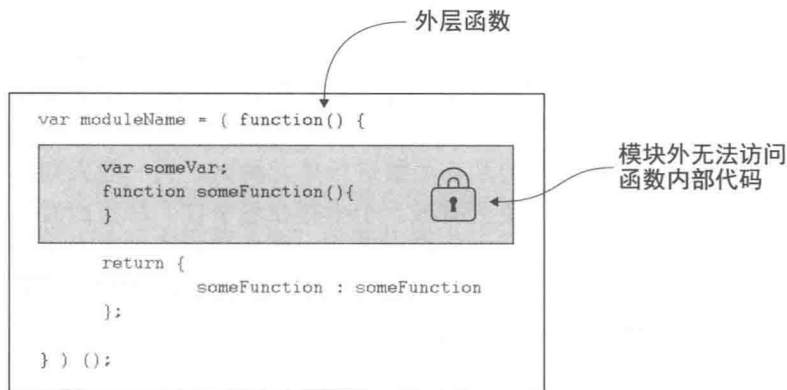


图 3.10 模块模式中的外层函数为其内部的变量和函数创建了一个局部作用域。这使得模块能够构建内部代码的私有性

你可以回忆一下前面章节讨论的模块模式内容，代码内部化（在函数内部构建应用逻辑）的能力是实现模块模式目标的根本要素，这些目标包括：避免命名冲突、保护代码完整性以及模块功能访问管理等。

3.3.2 创建公有 API

我们通过结合某些技术来创建模块的公有 API。这些技术采用有点独特的语法结构，但却能完美结合并达成效果。

1. 返回对象字面量

不同于诸如 `true` 或 `3` 这样的简单值，模块模式中将返回一个对象（如图 3.11 所示）。

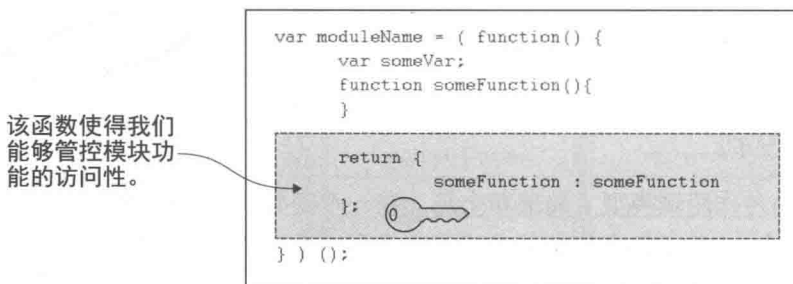


图 3.11 返回一个对象字面量。其中的函数可以访问模块内部的变量和函数。这使得调用代码可以管控模块功能的访问性

返回的对象可以是任意类型的变量与函数。但函数应该是那些允许你公开行为的东西。这也就是我们通常看到返回对象中只带有函数的原因。

对象字面量非常适合用来返回对象，因为它的语法提供了一个不错的手段，不需要借助 `new` 关键字，并以一种层次化的方式来定义对象。如早先讨论模块模式时所述，暴露的函数仅仅不过是指向模块内部函数的指针而已。

对象字面量回顾

我们来回顾一下对象字面量。在 JavaScript 中，我们可以通过 `new` 关键字、`Object.create()` 函数（ECMAScript 5）创建一个对象，或者使用字面量记法（也称为对象初始化器——`object initializer`）。字面量记法使用花括号定义对象，对象的 `Property` 与值以名称 - 值对的方式组织，用冒号分隔。我们还需要在每个名称 - 值对的最后加上逗号（除了最后一个名称 - 值对）。值可以包含变量、函数，或者其他对象。这是一个示例：

```
var employee = {                                ← 开始声明对象
  firstName : "Bob",                            声明变量
  lastName : "Jones",
  deptInfo : {
    dept : "Accounting",
    bldg : "C Building",
    floor : "1st floor"                        嵌套对象
  },
  getFullName : function(){
    return this.firstName + " "
    + this.lastName;
  },
  getDeptInfo : function(){
    return this.deptInfo.dept + ", "
    + this.deptInfo.bldg + ", "
    + this.deptInfo.floor;                    声明函数
  }
}                                               ← 结束对象声明
console.log(employee.getFullName()
+ ": " + employee.getDeptInfo());           打印 "Bob Jones: Accounting,
                                           C Building, 1st floor"
```

2. 让函数立即返回

通常在函数定义完毕，直到其他代码调用它，它才会返回。在模块模式中，我们需要作为模块命名空间的变量指向返回的对象字面量，而非函数本身。这可以通过在代码结构中添加一组尾部括号来实现（如图 3.12 所示）。

如果没有尾部括号，整个函数就将分配给变量，而非期望中的返回对象。而加了尾部括号，匿名函数将立即调用，并返回对象字面量给变量。这通常被称为立即调用函数表达式（`immediately invoked function expression`, `IIFE`）。

```

var moduleName = ( function() {
    var someVar;
    function someFunction(){
    }

    return {
        someFunction : someFunction
    };
} ) ();

```

这将引发立即调用。

图 3.12 尾部括号引发模块模式的匿名函数立即调用，并返回对象字面量

3. 闭包构造

为了构成模式的整个技术栈能够正常工作，所有返回对象字面量中引用的私有变量或函数都不能回收。否则，就可能导致在使用 API 时导致错误。当对象字面量分配给 `moduleName` 变量时，这种特殊情况就发生了。此时对象字面量函数已提供使用，因此它们不能进行垃圾回收。由于对象字面量中的函数同时引用了内部私有对象，所以这些私有对象就也不能进行回收。如前面定义闭包时所述，这种特殊情况是可能存在的，因为所有函数都具有引用了外层词法作用域的作用域¹。这就是闭包，能够在 IIFE 完成执行后仍维系着内部功能的生存期（如图 3.13 所示）。

```

var moduleName = ( function() {
    var someVar;

    function someFunction(){
    }

    return {
        someFunction : someFunction
    };
} ) ();

```

闭包使得模块的外层作用域中被引用的对象不会进行垃圾回收。

图 3.13 闭包能够让其引用的任何变量或函数（指 IIFE 中的变量或函数）存活下来，即使在 IIFE 执行之后

在闭包中，由于继续持有模块内部函数的引用，在外层函数执行结束时，模块内部函数仍可以安全工作，不会成为 `undefined`。

¹ 假设这里的 `someFunction` 包含了外层作用域的变量引用（如 `someVar`），此时该变量不能回收——这应该是这句话想表达的意思。感觉原著在这里对闭包阐述得过于含糊。——译者注

3.3.3 允许全局导入

尾部括号还同时提供了一种方式，能够通过参数形式将你声明的内容传进匿名函数。在模块术语中，将扩展对象带进内部供内部使用的过程被称作导入（Import）。

使用这个技巧将全局变量导入模块是通用实践。这不仅使得代码更易读，而且还能够加速解释器的变量解释过程。最后，该技巧允许我们在函数作用域范围给全局变量设置别名。比如说 jQuery：

```
var moduleName = (function($) {  
    function init() {  
        $("#div-name").html("Hello World");  
    }  
    return {  
        init : init  
    };  
})(jQuery);
```

通过函数参数，给 jQuery 设置别名 \$

导入 jQuery

\$ 是大多数开发者喜欢的 jQuery 引用方式。其比键入 *jQuery* 容易多了。但许多库也喜欢在其代码中使用 \$。由于我们在模块中局部指定了 jQuery 的别名，因此这里的 \$ 就不会跟其他库中的 \$ 发生冲突。

3.3.4 创建模块的命名空间

模块模式的最后部分是建立其命名空间。命名空间提供了调用模块公有 API 的方式，并能够帮助我们创建子模块。

JavaScript 的函数可以声明或分配为一个表达式。模块模式的外层函数是一个 IIFE。分配立即调用匿名函数给一个变量，不仅获得一个指向返回对象字面量的指针，还创建了模块名称。同时也定义了模块命名空间，子模块可以依附其上（如图 3.14 所示）。

创建模块的命名空间

```
var moduleName = { function() {  
    var someVar;  
    function someFunction(){  
    }  
    return {  
        someFunction : someFunction  
    };  
} } ();
```

图 3.14 该次分配创建了模块的命名空间

提示 IIFE 的外层括号不是必需的，因为 IIFE 是一个函数表达式。但这能够形象地建立一个模块边界（如图 3.15 所示）。

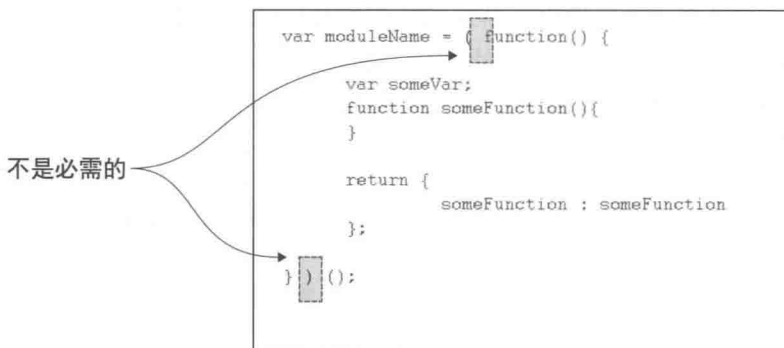


图 3.15 外层括号不是必需的

模块模式的内容到这里就讲解完了。如你所见，模式的各个部分都有其存在目的的。整个模块模式最终让我们的代码更加清晰，目标更加明确，并能够满足项目规模不断扩大的需要。

3.4 模块加载及依赖管理

在大多数的浏览器中，模块文件用到的 `<script>` 标签会产生阻塞。脚本加载时应用程序将暂停。因此，模块文件数越多，等待应用加载的时间就越长，而用户感受到的延迟现象也就越严重。同时，大量的 HTTP 请求也将使网络不堪重负。

为了缓解该问题，我们可以将模块合并到尽可能少的文件当中去，之后优化最终文件。第 9 章将介绍这两种技术及相关工具。但是，尽管我们能够从这两种技术中受益，但本质上仍需要 `<script>` 标签同步处理。为了解决这个问题，我们可以尝试寻求第三方库以异步加载模块。

3.4.1 脚本加载器

能够绕过 `<script>` 标签的阻塞状态将极大缩减应用加载时间。通过 `<script>` 标签的 `defer` 和 `async` 属性，HTML 为 JavaScript 代码的加载和执行引入了原生的非阻塞支持。`defer` 属性表明脚本在页面完成解析后才执行。`async` 属性则只要脚本可用即异步执行。使用 `<script>` 标签时，由开发者来确保脚本的正确顺序，以便依赖在需要的时候可用。另一个替代方式是使用 AMD 脚本加载库。

AMD 脚本加载器通过处理低级别样板代码来管理异步下载过程。其还允许指定模块所需的依赖给函数。如果依赖模块不存在，则框架将获取它们并确保在执

行之前下载就绪。有不少脚本加载器可供选择——比如 LABjs、HeadJS、curl.js 以及 RequireJS 等。每个脚本加载器都有少许不同，但都是用于解决加载及管理问题的。同样值得一提的是，虽然在撰写本书时尚未批准，但异步加载脚本特性已成为 CommonJS 风格脚本加载器的一项提案。

然而世上没有完美之物。异步加载脚本提升了加载速度却引入了另一个问题：无法预知的资源可用性。脚本异步加载就意味着无法精确知道哪个脚本将首先加载。在所有依赖准备就绪之前，某个文件就已下载并开始执行——完全有可能出现这种情况。如果脚本因为依赖尚未加载而失败，那么，再快的加载时间也是空谈。值得庆幸的是，大多数脚本加载库同样意识到了这个问题。

脚本加载器推迟脚本执行，直到模块所需的文件和所有依赖加载完毕。大多数脚本加载器也能缓存模块，因此无论请求有多少次，模块都只需加载一次。

为了阐述一些基本的加载及管理概念，本书需要引用一个库。我们选择了 RequireJS，因为它是目前使用最广泛的脚本加载库。了解 RequireJS 还给了我们一个好机会，能够了解一个与之前讨论的传统模块模式稍有不同的流行模块格式——即异步模块定义（Asynchronous Module Definition, AMD）API。下一节将定义 AMD 并通过 RequireJS 掌握脚本加载的基本概念。

3.4.2 异步模块定义——AMD

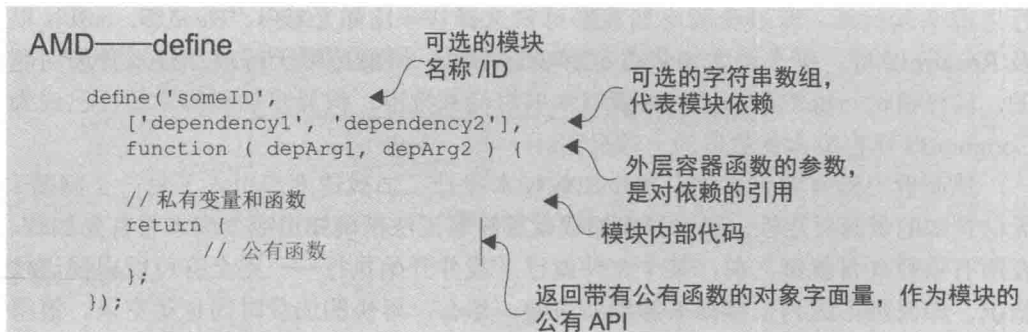
AMD 起初作为大型项目 CommonJS 的模块格式草案。CommonJS 不仅试图解决 JavaScript 语言缺失标准模块定义的问题，而且还希望以一种格式应对各种各样的环境——包括服务器端。但在 CommonJS 开发组里，并未就模块规范达成共识（至本书撰写时），于是 AMD 就移到了自己的规范组——AMD JS 规范组（<https://github.com/amdjs>）。

在 Web 浏览器中，AMD 已经很大程度上得以采用。该规范定义了一个标准模块格式，同时还定义了异步加载模块及其依赖的方式。AMD 规范定义了两种结构：define 和 require。在使用 RequireJS 开发样例之前，我们先来讨论这两种结构。

1. define

使用 define 声明一个模块，就像之前使用模块模式那样。其跟模块模式的相似之处还有：AMD 模块也允许导入对象，可以通过匹配函数参数来访问导入对象；AMD 模块体也嵌套在一个外层容器函数中；另一个相似点是 AMD 模块体也包含了可通过返回语句暴露的私有功能。

这时候你可能会问，既然之前已经有了通过模块模式创建模块的完美方式，为什么需要 AMD？主要的原因是 AMD 代表了一个模块定义的正式规范，且是 JavaScript 语言缺失的。但更具体地说，它是定义异步加载模块及其依赖的一个规范。

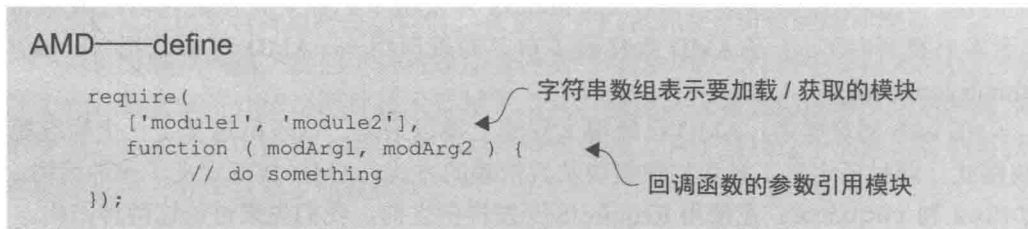


提示 尽管规范允许有一个模块 ID，但通常会省略。如果省略了 ID，脚本加载器会内部生成一个 ID。反过来，内部通过生成的 ID 对模块进行管理。未命名的 AMD 模块更便于移植，允许你无须修改代码即移动模块到新路径下。

注意命名空间并未像模块模式中那样定义。使用 AMD 和脚本加载器的另一个好处是命名空间已不再需要了。模块由库来管理。同时也不必担心依赖的顺序，它们将在模块函数执行时加载并就绪。

2. require

require 语法用来异步加载 / 获取指定模块。在此结构中，定义加载模块的方式与通过 define 语法声明依赖的方式相同。当请求模块获取到并准备就绪时执行回调函数。



require 结构是指令，不是模块声明。因而模块定义通常是一个个物理文件，require 应在需要之时使用。require 可以单独使用，也可以在模块中使用。

3.4.3 通过 RequireJS 实践 AMD

实践有助于理解概念。因此，我们来看一个早先的样例，并将模块转换成 AMD 风格的模块。此时将使用 RequireJS 作为模块加载库，以实现 AMD。

在开始该例子之前，让我们先看看 RequireJS 的概念。尽管我们聚焦于模块加

载器概念，但仍需要了解一些 RequireJS 的基础知识来完成该例子。

- data-main——一切皆有起源。如 Java 或 C# 语言的“main”方法，RequireJS 同样有一个入口。当在 SPA 应用中添加一个包含 RequireJS 库的 `<script>` 标签时，也需要为其添加一个 data-main 属性。data-main 属性是加载了 RequireJS 的应用的入口。通常情况下，主 JavaScript 文件包含 RequireJS 配置及 require 指令以执行初始 AMD 模块。
- requirejs.config()——该函数创建 RequireJS 配置项。其唯一参数是一个对象字面量，该对象字面量包含了所有配置项及其值。
- baseUrl——相对于 Web 应用程序根目录的路径。配置对象中的任何其他路径都相对于该路径。
- path——我们得把模块的位置告知 RequireJS。path 映射模块名称（你所构建）到 Web 服务器的路径上，相对于 baseUrl（以及应用程序根目录）。其可以是指向模块源文件的全路径（除去扩展），也可以是部分路径。如果使用部分目录路径，则至文件的剩余路径必须包含在对象作为依赖出现的任何地方。

还有许多配置选项，但我们只关心 RequireJS 基础概念，以将重点放在学习 AMD 上，而非专门阐述 RequireJS。如果你想进一步了解 RequireJS 具体配置项，可以参考 <http://requirejs.org> 上的官方文档。

现在我们了解了一些 RequireJS 基础概念，接下来就把例子中的模块转换成 AMD 模块。我们使用的样例取自 3.2.2 节，因为它很小，并且很容易进行前后版本的比较。

首先，我们需要从 <http://requirejs.org> 下载 RequireJS 库。图 3.16 圈出了正确的下载项。

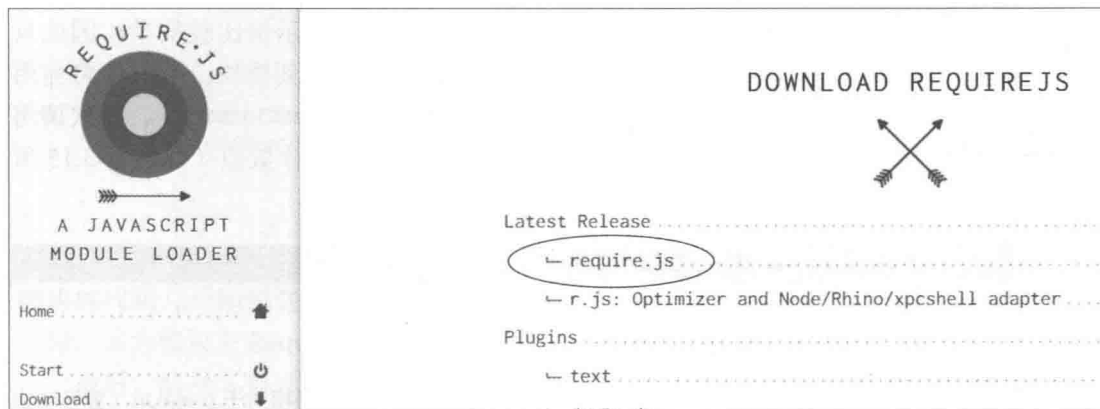
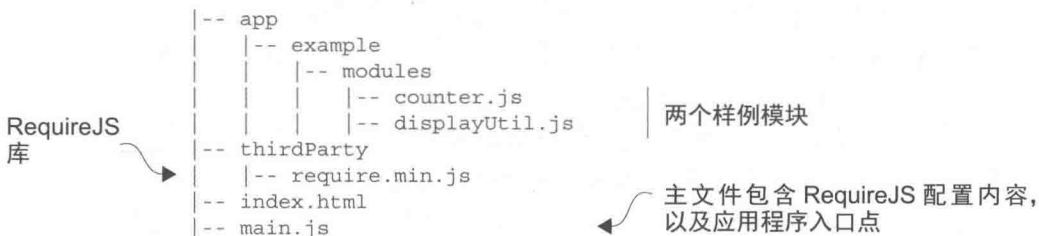


图 3.16 RequireJS 没有需下载的依赖项，只需下载 require.js 文件

整理好所需文件，样例目录现在的样子如清单 3.13 所示。

清单 3.13 AMD 样例目录结构



在浏览器中打开缺省 URL，就能立即看到 index.html 页面，因为服务器已将其设置为欢迎页面。我们唯一要做的就是在该页面中添加一个 RequireJS 库的包含。在 <script> 标签，通过 data-main 属性让 RequireJS 知道主文件为 main.js，其存在于 Web 应用程序根目录下（如清单 3.14 所示）。

清单 3.14 index.html

```

<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="css/default.css">
</head>
<body>
  <!-- starting point defined in data-main -->
  <script data-main="main.js"
        src="thirdParty/require.min.js">
  </script>
</body>
</html>
  
```

将应用程序主文件的位置告知 RequireJS

RequireJS 库

接下来，我们需要添加一些基本配置到 main.js。这个示例比较简单，因此只需配置 baseUrl 和 paths Property，以便 RequireJS 能够找到模块。我们还将另用 require 指令两次调用 displayUtil 模块的 displayNewCount()。两次调用 displayNewCount() 函数将让你明白计数器会相应增加计数值（如清单 3.15 所示）。

清单 3.15 main.js

```

requirejs.config({
  baseUrl : "app/example",
  paths : {
    counter : "modules/counter",
    util : "modules/displayUtil"
  }
});
  
```

baseUrl 相对 Web 应用程序根目录

路径相对于 baseUrl（冒号左边是模块名称）

```
require([ "util" ],
function(utilModule) {
    // 第一次增加计数器
    utilModule.displayNewCount();
    // 第二次增加计数器
    utilModule.displayNewCount();
}
);
```

使用模块名称将资源声明为依赖

函数参数接受对应依赖导出的对象

在回调函数内部，两次调用 displayUtil 来增加计数器

来看看清单 3.15 中 `require` 的调用代码，通过包含来自 `paths` 选项中的模块名称，我们通知 `RequireJS` 要在回调函数中使用 `displayUtil` 模块。依赖列表中的每个模块名称字符串都将有一个相应的函数参数。在回调函数中，我们通过该参数来引用模块。

清单 3.16 展示了 `displayUtil` 模块代码。由于这是一个模块声明，因此使用 `define` 语法结构。

清单 3.16 `displayUtil.js`

```
define(["counter"], function(counterModule) {
    function printCount(){
        var count = counterModule.getCount();
        if(count === 1) {
            count = count + " time";
        } else {
            count = count + " times";
        }
        console.log("Count incremented: " + count);
    }

    function displayNewCount(){
        counterModule.incrementCount();
        this.printCount();
    }

    return {
        printCount : printCount,
        displayNewCount : displayNewCount
    };
});
```

counter 模块是一个依赖

修改代码，以通过参数引用 counter 模块

在 `displayUtil` 模块中，我们唯一的依赖是 `counter` 模块。`displayUtil` 模块体代码几乎跟原先非 AMD 版本一一对应。请牢记我们不需要为模块分配命名空间，因为模块由 `RequireJS` 内部来管理。

最后，清单 3.17 展示了 `counter` 模块。由于它没有依赖，因此可以省去此部分的结构代码。既然没有依赖，外层函数也就没有参数。

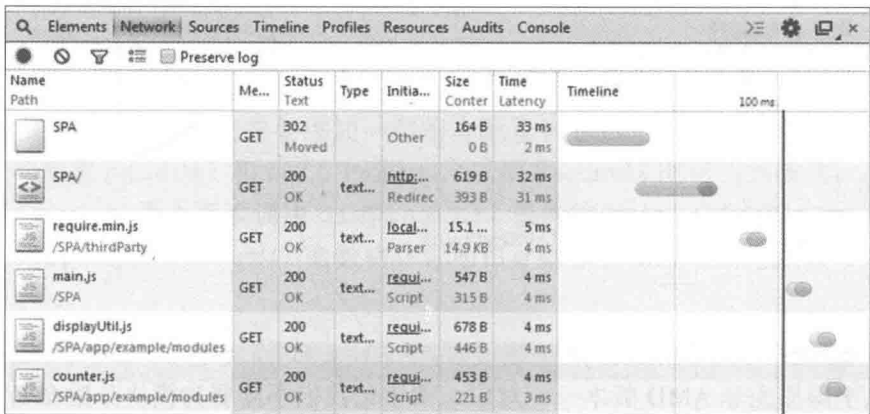
清单 3.17 counter.js

```
define(function() {  
    var count = 0;  
  
    function incrementCount(){  
        ++count;  
    }  
  
    function getCount(){  
        return count;  
    }  
  
    return {  
        incrementCount : incrementCount,  
        getCount : getCount  
    };  
});
```

依赖列表部分的代码结构可以省略；
模块外层函数没有参数

随着所有文件准备就绪，我们可以启动服务器并导航到缺省 URL。应用将在 `require` 指令完成后启动。图 3.17 展示了浏览器的网络控制台信息。现在我们可以看到模块加载器的实际情况了。

通过网络控制台的输出，我们可以发现模块加载器会自动加载所有请求模块。需要我们完成的工作就是为 `require` 或 `define` 声明添加依赖。由于每个依赖都通过相应函数参数传进模块，因此我们就能访问依赖模块的公有 API。此外，由于 RequireJS 自动管理模块，因此我们就不需要任何命名空间。这有力体现了 AMD 模块及实现了 AMD 规范的模块加载器的强大能力。



Name Path	Me...	Status Text	Type	Initia...	Size Conter	Time Latency	Timeline
SPA	GET	302 Moved	Other		164 B 0 B	33 ms 2 ms	
SPA/	GET	200 OK	text...	http...	619 B 393 B	32 ms 31 ms	
require.min.js /SPA/thirdParty	GET	200 OK	text...	local...	15.1 ... 14.9 KB	5 ms 4 ms	
main.js /SPA	GET	200 OK	text...	requi...	547 B 315 B	4 ms 4 ms	
displayUtil.js /SPA/app/example/modules	GET	200 OK	text...	requi...	678 B 446 B	4 ms 4 ms	
counter.js /SPA/app/example/modules	GET	200 OK	text...	requi...	453 B 221 B	4 ms 3 ms	

图 3.17 模块加载器正确地下载并管理我们的 AMD 模块及依赖。main.js 中的 `require` 指令声明依赖，以引导 RequireJS 获取 `displayUtil` 模块。`displayUtil` 模块则有一个 `counter` 模块的依赖，模块加载器会动态加载 `counter` 模块

3.5 挑战环节

现在来挑战一下自己，看看你在本章中都学到了哪些技法。在诸如 Firefox 或 Chrome 这样的浏览器中，你能够通过 %c 来给控制台打印消息添加样式，如：

```
console.log("%c" + errorMessage, "color: red");
```

试试看能否将其转换为一个可复用的日志模块。暴露的函数能根据不同日志模式（debug、info、warning 和 error）打印出不同颜色。在内部创建变量，分别代表不同的模式颜色：debug 用黑色，info 用绿色，warning 用橙色，而 error 用红色。还需要定义内部功能，通过所用日志模式的对应颜色打印信息。

3.6 小结

本章介绍了模块的概念及模块模式用途：

- 设置私有代码，只能够在模块中使用。
- 提供公有 API，通过控制内部代码的访问权限，隐藏复杂性，并保护组件代码的完整性。
- 避免将所有的 JavaScript 变量和函数都塞到全局命名空间，防止命名冲突的发生。
- 降低代码变更造成的影响。
- 将应用程序的关注点划分为更合理的可管理单元。

世上并无完美之物。本章还提到了模块模式的缺点：

- 公有 API 的单元测试受到限制。
- 内部对象的扩展并不容易。

最后，我们讨论了一个更为正式的模块规范——AMD，了解了此时已不再需要命名空间，同时还学习了异步获取模块及其依赖的方式。

在下一章，我们将讨论关注分离的另一种实现途径：通过 MV* 库来分离 HTML 与 JavaScript 代码。

第2部分

核心概念

本书第2部分讲述 SPA 应用各个部分的协同运作，同时还会介绍 SPA 应用的测试以及客户端任务自动化。

第4章介绍客户端路由。我们将了解路由器语法，并对几种路由器风格进行比较。在此过程中，还将讨论路由器的工作原理以及它们对应用程序状态的影响。

第5章介绍 SPA 应用的设计与布局概念。首先接触一个简单的设计，之后创建带有更为复杂布局的 SPA 应用。本章还涉及如何将高级路由设计与视图管理结合进 SPA 应用。

第6章我们将了解模块间通信方式，包括模块化设计概念的介绍。在这里，会学到模块间通信的几种方式。同时还将讨论各种方式的利弊。

第7章讲述客户端与服务器端通信，以及 MV* 框架是如何简化这一过程的。首先会介绍 HTTP 事务的基础概念，接下来涉及一些更高级的主题，如 Promise 及 RESTful 服务。

通过第8章的学习，我们将大致了解 SPA 应用程序的单元测试。在这里，会了解到一些基本的单元测试概念，并体验不同框架（本书所涉及）风格的测试过程。

第9章是本书的最后一章，我们将看到客户端自动化在开发阶段及构建过程的创建中所扮演的角色。本章内容是高级主题，因此在这里会一步步讨论过程的各个环节。当本章内容结束时，我们不仅能够掌握如何创建客户端构建过程，还能掌握如何让开发过程更高效。

单页面导航

本章内容

- 客户端路由介绍
- 路由及其配置
- 路由器工作原理
- 路由参数用法

本书第 1 部分覆盖了视图及其如何创建的内容。然而第 1 部分并未谈到客户端路由。客户端路由用来从一个视图导航到下一个，但这也只是其功能的一部分。它还跟应用程序不同状态间的转换有关。如无导航情况下修改当前视图，或者完全不涉及 UI 变化的其他活动。

本章探讨通过客户端路由在 SPA 应用中实现用户导航。在本章中，不仅会讨论客户端路由器的概念，还将了解其底层机理。

讨论过程使用伪代码阐述，尽量做到与框架无关。一旦掌握了相关概念，我们将利用所学到的知识，借助 AngularJS 来创建一个简单的路由样例。该样例为某所大学的某个系部创建一个网站。尽管例子很基础，但却充分阐述了最常见的客户端路由器基础概念，并避免牵扯太多与框架相关的细节。我们将在整个讨论中都使用这个大学网站例子，这将为我们的伪代码和讨论赋予更多实战的味道。

4.1 客户端路由器概念

首先把导航作为一个整体放到我们讨论的上下文里来。一开始我们将了解传统 Web 应用程序导航中的请求和响应模型。

4.1.1 传统导航

在传统 Web 应用程序中，导航是以整个 Web 页面为单位进行的。当在浏览器地址栏输入一个新的 URL 地址时，通常情况下，页面请求从浏览器发往服务器，服务器响应并返回一个完整的 HTML 页面（如图 4.1 所示）。



图 4.1 传统 Web 页面的导航，整个页面发回浏览器，之后刷新显示该新页面

其实返回的是请求页面的 HTML 文档。浏览器收到页面 HTML 文档之后，将获取该文档引用的其他所有源文件，如 CSS 与 JavaScript 文件。HTML 文档引用的其他资源如图片，也将在浏览器解析文档的 HTML 代码及遇到相应标签时得以下载。要显示新内容，浏览器就执行一次完整的刷新动作。

4.1.2 SPA 导航

如前所述，SPA 的 DOM 元素通常是作为 SPA index 页面中的 Shell 的起点。这就是全部所需。SPA 模块及 MV* 框架，包括支持库，都跟 index 页面一同下载，或者使用了 AMD 脚本加载库的话则进行异步加载。SPA 应用也有能力异步获取数据、任何远程模板（局部）以及尚未包括的其他资源，并按需动态渲染视图。

当用户导航时，视图无缝呈现。连同异步获取的数据，整个应用展示过程更加平滑，给人一种类似原生应用的效果，极大提升了用户体验。以往那种旧页面被清除，然后下载显示新页面所带来的粗糙体验已不复存在。一旦初始页面加载了，SPA 的各种动作都不需要刷新它。

然而，在 SPA 应用加载之后用户需要有一种方式来访问应用的其他内容。由于 SPA 仍属基于 Web 的应用程序，因此，用户会希望使用地址栏和导航按钮来进行导航。但在只有一个页面且不刷新浏览器的情况下如何进行 SPA 导航呢？

事实证明，这不仅可能，也很容易。让单页面环境中的导航成为可能的法宝就是客户端路由器（或简称路由器）。

但请记住当我们讨论导航时，说的是在用户导航时管理 SPA 视图、数据以及业务事务的状态。路由器承担着浏览器导航控制的职责，并依此管理应用程序状态，其允许开发者直接将 URL 的改变映射到客户端功能上(如图 4.2 所示)。我们会在 4.3 节具体讨论其实现。



图 4.2 SPA 应用的客户端路由器负责导航控制，允许 SPA 显示新视图而非整个页面

通过这种方式，与服务器端的往返已不再是必需的。路由器借助几种侦测浏览器位置发生改变的方法，来决定何时需要变化状态，比如监听特定事件。只要 URL 发生改变，路由器就会试着使用其配置里的某个配置项来匹配新 URL 的某部分。

在剖析典型的路由器配置项之前，先来看一下图 4.3，其提供了 SPA 导航过程的概览，其中还突出显示了客户端路由器。请注意路由器从未与服务器端发生交互。所有的路由都在浏览器端完成。

如你所见，当路由器匹配上路径（path）配置项与浏览器实际的 URL 时，其能够判断应该触发何种应用程序状态的改变：当前视图的数据有改变吗？业务流程跟该路由是否有关？路由是否导致当前视图的变化？

现在我们对基本路由过程有了些印象，接下来了解路由设置。下一节介绍路由器的配置及其典型内容的定义。

1. 用户通过教员ID “manderson” 请求授课时间视图

`http://someuniv.edu/#/officehrs/manderson`

2. 路由器尝试用路由器配置中的某条路由对 URL 进行匹配

路由器

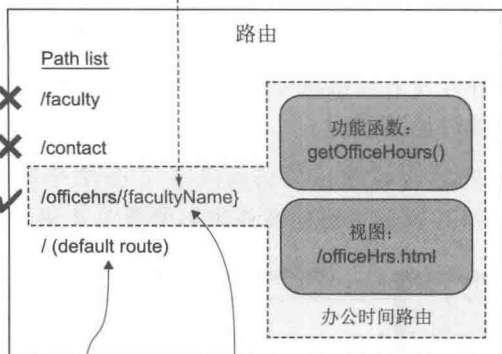
```
getOfficeHours(){
  ...
}
```

3. 根据匹配上的路由或未匹配上的缺省路由运行相应代码

4. 为 Mary Anderson 博士显示视图



视图



当匹配不上时

路由能够包含参数（通过特殊符号，具体看框架约定）

该示例产生了视图变化，但路由并不一定会导致 UI 更新

图 4.3 SPA 导航过程概览及路由器角色

4.2 路由及其配置

无论使用什么路由器，都必须事先进行一定的配置。必须通过路由器配置项，来制定路由器在用户导航时如何响应的策略。

每一个路由器配置项被称为一个路由。于开发阶段就在路由器配置中设置好路由，通过一个个路由，来表示 SPA 应用中的各个逻辑流程。典型地，我们将路由设计为应用程序的布局。虽然路由器配置因不同路由器而不尽相同，但仍有一些典型的配置术语：

- 名称 (Name) —— 有些路由器会给路由分配名称，而有些路由器则将路径作为路由标识。
- 动词 (Verb) —— 路由有时候定义为匹配 HTTP 动词的功能名称，例如 `get()` 或 `put()`，但也不总是这样，有些路由器使用更加通用的名称，如 `on()`、`when()` 或 `route()`。
- 路径 (Path) —— 路径是 URL 的一部分。路径用于配置路由器，以在 URL 与路由 / 路由处理程序之间建立链接。这样允许路由器找出要执行哪组动作。只要浏览器的 URL 发生改变，路由器就会用配置文件中路径列表的所有路由

路径跟新 URL 进行比较，以找出匹配项。如果找到匹配项，就执行该路由。路由路径必须是 URL 的合法部分。虽然通常情况下路径是简单的文本串，但有些路由器也允许使用正则表达式。

- 功能（Functionality）——可能执行的相关代码，如控制器或回调函数。对路由做任何事情，都需要执行某些相关代码。

有些（并非所有的）路由器能够通过配置来定义视图。还记得路由事关应用状态的改变，而非产生视图更新。我们在这里提到视图定义是因为某些路由器而言视图也是其配置项之一：

- 视图——在大多数情况下，如果路由器允许视图作为路由配置的一部分，则该配置项八成是到 HTML 某部分的路径。你可能还记得第 2 章中提到的，在开发阶段分离出来且仅包含针对某特定视图的元素的文件。当视图配置为路由的一部分时，路由器通常会处理它的显示，并给出访问该视图（或视图的某个代理对象，如视图模型）的功能。

图 4.4 展示了一个某些路由器配置中的普通路由，正如我们本章中用到的。由于每个路由器都不尽相同，本图并无法完美涵盖所有路由器。然而，可以将其作为一个高度概括的通用路由配置。请记住，SPA 应用程序开发完成后，最终很可能在路由器配置中会存在大量路由。

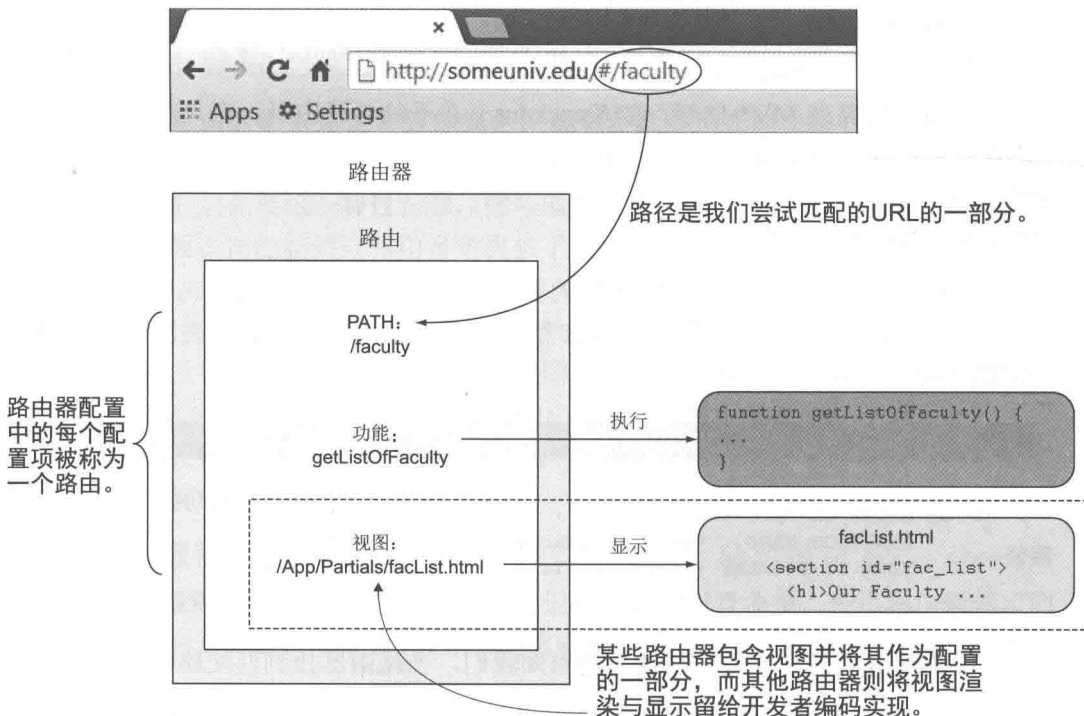


图 4.4 当路由路径匹配上浏览器 URL 的一部分时，路由器配置项充当下一步动作的指令

路由器可能还提供了其他更高级的特性。各种路由器情况不一。可以查阅所用路由器实现的官方文档，以获取完整的可用配置项列表。

4.2.1 路由语法

从语法上来看，路由代码因不同路由器而各有不同。在某些路由器实现中，路由和路由处理的配置是分离的。而其他的路由器实现则在同一处进行两者的配置。然而各家路由器都有一个共同点——执行路由。下面来看一些语法样例。

表 4.1 不是一份详尽列表，但它为你展示了几个例子。请记得通过所用路由器的文档获取准确语法。

表 4.1 客户端路由器语法示例

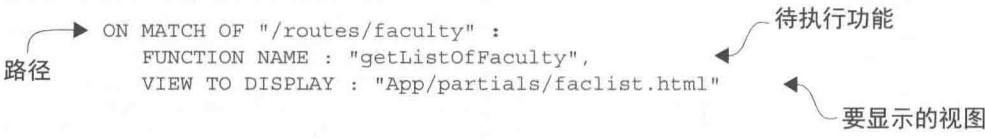
框架 / 库	路径示例
Sammy.js http://sammyjs.org	<code>get('#/routes/faculty', function() {...})</code>
Kendo UI http://www.telerik.com/kendo-ui	<code>route("/routes/faculty", function() {...})</code>
AngularJS https://angularjs.org	<code>when("/routes/faculty", { ... })</code>
Backbone.js (两个步骤) http://backbonejs.org	<code>1.routes: {"/routes/faculty": "facultyRoute"} 2.on("route:facultyRoute", function () {...})</code>

还请记住某些 MV* 框架，如 Knockout，并不包含路由器。在这种情况下，我们需要考虑外部路由器库，比如 Sammy.js。

4.2.2 路由配置项

现在来了解一个完整路由配置项的真面目。我们再次使用伪代码来描述真实内容——本章最后完成的项目。清单 4.1 描述了一个基本路由，能够在其中的配置里指定视图。

清单 4.1 带有视图功能的路由器（伪代码）

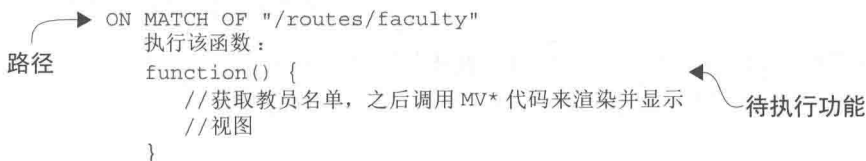


路由配置项几乎就是一句话。其告知我们，当路由器找到匹配路径的 URL 时，

该如何动作。这也是一条可遵循的简单规则：匹配模式、运行代码，并展示结果。

接下来仍是同样的路由，但路由器让开发者决定视图渲染和显示的实现。在清单 4.2 中，路由器只提供匹配路由和某个控制器 / 回调函数的设置。

清单 4.2 路由器将视图处理的实现留给了我们（伪代码）



```
ON MATCH OF "/routes/faculty"
  执行该函数：
  function() {
    // 获取教员名单，之后调用 MV* 代码来渲染并显示
    // 视图
  }
```

路径

待执行功能

当使用如 Backbone.js 这样的代码驱动型框架时，这种类型的路由器会提供极大的方便。

提示 不管使用何种类型的路由器，都请尽量将路由配置文件用于路由本身。这是一个更好的实践，以让路由的回调函数使用应用模块来执行所有业务逻辑，而非在路由中混入无关代码。

现在，我们了解了路由基本概念。大多数路由器通过将路由路径转为动态方式，提供了一种极大扩展路由能力的途径。下一节将阐述路由参数主题及其使用方式。

4.2.3 路由参数

大多数路由器都有路由参数的概念。路由参数是指在路由路径中定义的变量。此特性允许我们为 URL 添加变量，在后续路由执行时会计算变量。

为什么我们需要这个特性？嗯，你可能想使用同一功能和同一视图，但根据不同情况又需要不同的结果。路由参数提供了这样的灵活度。这是一个任你发挥的强大特性。

我们来讨论一下大学网站示例中显示教员授课时间的相关路由概念。在此场景中，路由以及视图的实现是相同的。但视图需根据所选教员名字显示不同信息。要达成此目的，得在路由路径中传入教员 ID。这是使用路由参数的极佳例子。

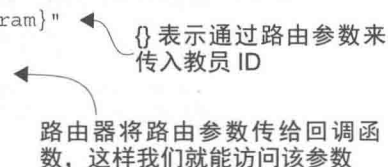
要让路由参数正常工作，还需要做两件事情：相对 URL 中包含需要传递的文本内容，接收端有带参数的路由路径。

1. 配置带参数的路由路径

为了告知路由器路由中包含参数，我们得在路由配置项中，使用路由器规定的特定字符来定义参数。清单 4.3 展示了带有路由参数的授课时间路由。

清单 4.3 带有参数的授课时间路由（伪代码）

```
ON MATCH OF "/routes/officehrs/{facultyNameParam}"
  执行该函数：
  function(facultyNameParam) {
    // 使用 facultyNameParam 参数传进的 ID，获取
    //  教师的授课时间
  }
```

 {} 表示通过路由参数来传入教员 ID
路由器将路由参数传给回调函数，这样我们就能访问该参数

每种路由器都有其自己的路由参数语法。表 4.2 包含了几种路由参数语法样例的比较。我们使用与表 4.1 相同的路由器。每种路由器恰好都使用相同的路由参数语法。

表 4.2 路由参数语法样例

框架 / 库	路径示例
Sammy.js http://sammyjs.org	:facultyNameParam
Kendo UI http://www.telerik.com/kendo-ui	:facultyNameParam
AngularJS https://angularjs.org	:facultyNameParam
Backbone.js（两个步骤） http://backbonejs.org	:facultyNameParam

在表 4.2 列出的框架 / 库中，全部使用冒号来表示一个路由参数。但不存在统一的标准。其他路由器可能使用不同的约定。此外，你选择的路由器还可能提供了更高级的参数选项，如正则表达式。具体请参看所选路由器的相关文档。

2. 带有传入文本信息的相对 URL

如果使用了路由参数，路由器可以通过路径识别 URL 哪部分是参数、哪部分应该逐字匹配。比如以下程序清单展示了一个链接，其匹配清单 4.3 中的路由。该 URL 完全匹配路径（除了参数部分）。

```
<a href="#/routes/officehrs/manderson">
  Dr. Mary Anderson
</a>
```

如果这是到授课时间路由的链接，则链接 URL 的最后部分代表教员 ID。每位教员在我们的虚构大学站点上可以拥有这样一条链接：使用同一路由模式但却以不同 ID 作为 URL 尾部。

3. 多个参数

大多数（并非全部）路由器允许同时使用多个参数。假设针对一个教员存在多条链接，我们也许会希望进一步参数化该路由：

```
/routes/officehrs/{facultyNameParam}/{dayOfTheWeek}
```

除了定义特定路由，还可以包含一个缺省路由。

4.2.4 缺省路由

讨论完路由及其配置，我们还需要了解一下缺省路由。这种类型的路由适用于路由未指定或路由非法等各种情况。当用户输入的 URL 在路由器配置中未找到匹配项时，如果没有缺省路由，应用就会无所适从。而一旦存在一条缺省路由，在这种情况下就能立即重定向到该特定路由。清单 4.4 展示了路由匹配项未找到时，如何重定向所有 URL 到教员路由。

清单 4.4 缺省路由（伪代码）

```
DEFAULT ROUTE: [  
  REDIRECT TO "/routes/faculty"  
]
```

其他情况表示缺省情况。当 URL 未匹配到其他所有路径时，重定向回教员路由。

缺省路由意味着如果存在匹配路由项，则使用该路由；否则，重定向到教员路由。缺省路由还是一个便利机制，用于当用户在浏览器中输入站点的 base URL 时指定具体动作，同时不需要包括特定路径。

下一节解释导航发生时路由器会做些什么。这是一个高度概括的内容，但刚好够我们理解路由的幕后机制。

4.3 客户端路由器的工作机制

客户端路由器的部分职责是允许用户如同传统 Web 应用中那样使用地址栏和浏览器导航按钮。许多客户端路由器为了满足该需求至少都提供了以下特性：

- 通过路由定义的路径来匹配 URL 模式。
- 当匹配成功时允许应用程序执行代码。
- 当路由触发时允许指定需显示的具体视图。
- 允许通过路由路径传入参数。
- 允许用户使用标准的浏览器导航方法来进行 SPA 应用导航。

这些特性是提供最低限度 SPA 导航功能所必需的。但请记住，不存在所有客户

端路由器必须遵循的保障标准。这些只是我们会遇到的最常见选项。MV* 框架（或独立的路由器库）的文档会列出它的所有特性说明。

总结完绝大多数路由器提供的基本特性后，让我们揭开面纱看看单页面设置中路由器到底是如何提供导航功能的。

路由器通过两种方式之一来进行导航：通过 URL 的片段标识符（fragment identifier），或者通过 HTML5 历史 API（History API）。这两种方式都能够让路由器提供非服务器端的导航，但方式上存在些许差异。由于 HTML5 历史 API 比较新且老浏览器不支持它，因此我们首先采用更为传统的片段标识符方式来阐述客户端路由导航。

4.3.1 片段标识符方式

SPA 路由器提供导航功能的传统方式是使用片段标识符。如图 4.5 所示，片段标识符是任意的文本字符串并以 # 号为前缀。这个 URL 的可选部分引用当前文档的某部分，并非对新文档的引用。

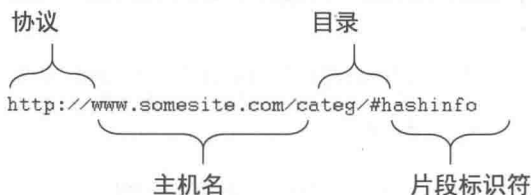


图 4.5 片段标识符

浏览器对待片段标识符的方式不同于对待 URL 其余部分的方式。当新的片段标识符添加到 URL 中时，浏览器不会尝试连接服务器端。但是添加的结果会在浏览器历史中新增一个条目。

这很重要，因为所有浏览器历史中的条目，即使是那些由片段标识符产生的条目，也能通过正常方式如地址栏、导航按钮导航到。要观察实际的情况，可以打开任一站点，如 `www.manning.com`。然后在浏览器控制台执行下列命令：

```
window.location.hash = "hello";
```

你会看到，这将导致 `hello` 添加为 URL 的片段标识符。执行后的 URL 看起来像这样：

```
http://www.manning.com/#hello
```

该动作也将在浏览器历史中添加一个新条目。现在你可以在片段标识符和原始 URL 之间来回导航试试看。

利用浏览器的 location 对象

location 对象包含了一个 API，允许我们访问浏览器的 URL 信息。在 SPA 应用程序中，路由器利用 location 对象以编程方式访问当前 URL，包括片段标识符。路由器以此方式，并通过 window 的 onhashchange 事件来监听 URL 片段标识符部分的改变（如果浏览器支持 onhashchange 事件的话——否则轮询 hash¹ 的变化）。

当改变发生时，将用新的 hash 串来跟路由器配置中各路由的所有路径进行比较。如果匹配上了，路由器就执行具体程序，之后显示来自匹配路由的视图。

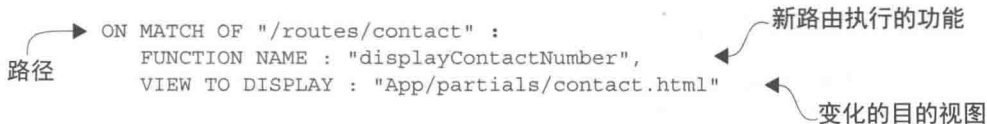
举例来说，假设站点标头有一条到虚构系部的主要联系人页面的链接。链接的代码指向一个片段标识符 URL：

```
<a href="#routes/contact">Contact Us</a>
```

当单击该链接时，浏览器的片段标识符将由初始值变为 #/routes/contact。

由于路由器实时监听片段标识符的变化，因此，新的 hash 就能够监测到。根据监听结果，路由器搜索配置中的所有路由，看是否有路径匹配 /routes/contact。如果找到匹配项，清单 4.5 中的路由就会执行。

清单 4.5 主要联系人路由（伪代码）



现在我们应该对客户端路由基础知识掌握得比较好了。如本节开头提到了，路由器可以通过两种方式来控制应用程序的状态。我们了解了片段标识符方式，下一节来看看较新的 HTML5 历史 API 的方式。

4.3.2 HTML5 历史 API 方式

上一节我们学习了通过片段标识符方式来改变 URL 的 hash 信息，路由器能够添加新的导航条目到浏览器历史中。每次 hash 变化都会在历史栈里添加一条新条目。添加完条目之后，用户就可以在无须触发页面刷新的情况下在 hash 间来回导航。但在使用片段标识符方式时，开发者必须围绕 hash 符号（#）来创建路径。

HTML5 历史 API 的新方式则与此不同。路由器可以利用 HTML5 提供的新功能与浏览器历史交互，且无须依赖片段标识符。同时，由于该方式不为老式浏览器所支持，因此，大部分路由器都会自动优雅地使用片段标识符作为回退方案。

1 hash symbol，也就是#符号。比如前面的#hello，hello代表当前页面中的某个位置，#hello也就是片段标识符。本章中提到的hash串、hash信息等，其实也就是指#hello这样的内容。——译者注

1. pushState 与 replaceState 方法

路由器可以使用历史对象 API 中的这两个新方法：

- `pushState()` —— 允许我们添加新的历史条目。
- `replaceState()` —— 允许我们用新条目替代已有历史条目。

这些新方法允许直接访问浏览器历史而不需要依赖片段标识符。下面简略介绍一下 HTML5 历史 API 方法。

通过 `history.pushState()` 或 `history.replaceState()`，路由器可以直接修改浏览器历史记录栈。这两个方法还使得路由器的处理方式更加“优雅”，用更自然的 URL 片段取代了 `hash`。它们都具有三个参数：

- 状态对象 (State object) —— 可选的与历史条目相关的 JavaScript 对象。
- 标题 (Title) —— 表示历史条目的新标题（然而本书写作时尚未得到大多数浏览器的支持）。
- URL —— 将在浏览器地址栏显示的 URL。

现在试一试 `pushState()` 方法，以了解它们是如何工作的。打开任一站点，如 `www.manning.com`。然后在浏览器控制台输入以下命令：

```
history.pushState({myObject: "hi"}, "A Title", "newURL.html");
```

该命令导致 URL 变成了：

```
http://www.manning.com/newURL.html
```

与此同时，这条命令还在浏览器历史中添加了一条新条目。现在我们可以试着在新 URL 及原有 URL 间来回导航。同时还能看到通过 `pushState()` 添加的 URL 并未触发浏览器刷新，也未包含 `#` 号。

为了观察状态对象是否添加了，我们可以在控制台输入 `history.state`，在输出结果中可以看到返回的 `myObject` 内容。

2. popstate 事件

最后，路由器给出了一种方式可以监听历史栈的变化——`window.popstate` 事件。只要用户在历史条目间导航，浏览器就会触发该事件。

我们可以在控制台试验一下该事件。使用 `pushState()` 方法来添加一些历史条目，然后在控制台执行以下代码：

```
console.log("popstate event fired");  
});
```

接下来，在通过 `pushState()` 添加的 URL 间来回导航。我们将看到控制台打印出了以下的日志内容：


```
popstate event fired
```

现在我们了解了新的 HTML5 历史 API 方式的路由，接下来看看如何修改我们的代码以使用它。

4.3.3 使用 HTML5 历史 API 方式

大多数路由器都支持 HTML5 历史 API 的路由导航方式。告知路由器你想用哪种方式通常就如同设置一条配置项那样简单。然而，除了模式切换还需要做出一些其他调整，本节将讨论这些内容。

我们开始来改用 HTML5 历史 API 方式。在许多路由器中，只需要将某个 Boolean 值由 false 改为 true。

1. HTML5 模式

为了将我们的示例由片段标识符方式改为 HTML5 历史 API 方式，需要调整路由器设置中的相应配置项，以换用 HTML5 历史 API 方式。例如在 AngularJS 中，我们会这么做：

```
html5Mode(true);
```

在 Backbone.js 中则会这么做：

```
Backbone.history.start({pushState: true});
```

我们再次看到了基于特定框架的示例。请查阅你所用路由器的文档，以了解确切语法。

2. 基准链接

现在已告知路由器使用 HTML5 历史 API 方式，还需要在 index 页 <head> 头部设置基准链接 (BASE HREF)：

```
<head>
<base href="/SPA/">
</head>
```

要让 HTML5 历史 API 正常工作，BASE HREF 必须与部署应用 *base URL* 的根路径一致，否则在路由中获取视图时将报 “Error 404 not found” 错误。

提示 只有不打算在链接 / 代码中包含完整路径时，才需要 base URL。

在该示例中，/SPA/ 是 *base URL* 中的根路径。因此需要将其作为 BASE HREF。市面上存在大量不同的服务器，应用部署方式也各不相同。但只要 BASE HREF 设置得当，视图就能够正常显示。

3. 服务器端调整

最后, 要完成 HTML5 历史 API 的使用配置, 还需要对服务器端进行调整, 以便其总能为根路径返回内容。例如, 如果有完善的服务器端路由配置, 那它始终能够为客户端返回正确资源。

要注意一点, 如果用户使用标签或进行页面刷新, 浏览器将对同一内容发出请求。一个可能的解决方案就是在服务器端设置重定向, 内部重定向到相同的 URL。

4. 移除 hash

如果路由器支持 HTML5 历史 API 方式, 我们现在就能从视图中的链接里移除 hash 字符 (# 号)。例如, 在主要联系人信息的链接中, `<a>` 标签可以重写如下:

```
<a href="/routes/contact">Contact Us</a>
```

当我们单击该链接时, 将在浏览器地址栏看到一条普通的 URL。没有 hash 字符了! 现在我们理解了客户端路由的基本概念, 接下来是卷起袖管写点代码的时候了。

4.4 综合实作: 实现SPA路由

本节通过伪代码来讨论及阐述概念, 并用 AngularJS 创建一个可运行的实际例子。在此例子中, 我们继续聚焦于虚拟大学的功能实现。假定我们是大学 IT 职员, 负责为大学某系部搭建一个网站。

我们需要三个视图: 一个教员名字可供选择的登录视图, 一个显示所选教员的授课时间的视图, 还有一个普通的系部联系方式视图。

用户可以通过单击登录视图教员列表中的教员名字, 或者单击标头的导航链接进行导航。同时, 由于我们包含了一个 SPA 导航组件, 因此用户也能够通过浏览器地址栏和导航按钮进行导航。

我们在这里摘录了联系最紧密的代码。如果你打算试一试本章样例的完整代码, 可以下载它。图 4.6 展示了该样例的最终模样。

现在开始创建样例的第一个路由——教员列表路由。这是缺省路由。你是否还记得早前说过的, 缺省路由在当前 URL 无法匹

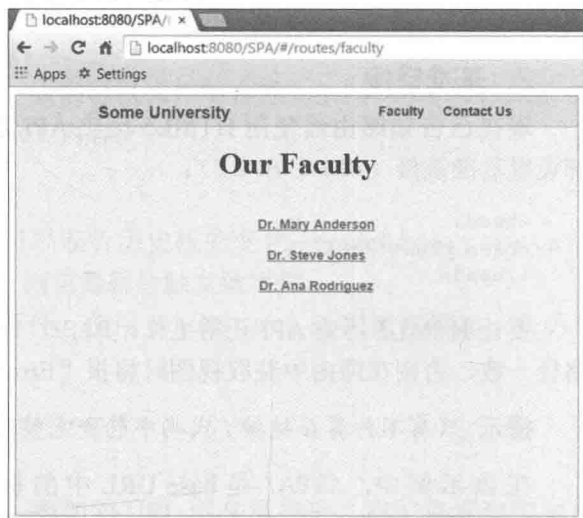


图 4.6 在本章样例中, 我们将为虚拟大学的某个系部创建一个基本网站

配置文件中任一路径的情况下使用？由于站点的 base URL 没有匹配项，因此缺省情况下应该总是执行教员列表路由。标头处有一条直达 base URL 的链接，因此也能通过该链接访问本路由。

4.4.1 教员列表（缺省路由）

在讨论路由时我们已见过教员列表的伪代码，现在来实现真正的代码。

要使用 AngularJS 的路由器，需要通过 `$routeProvider` 来配置路由。可以使用 `$routeProvider` 的 `when()` 方法来配置普通路由，并用 `otherwise()` 来配置缺省路由。虽然这是 AngularJS 的用法，但无须太担心。如果你使用不同的框架或路由库，则可以使用不同语法来添加同样的配置。

现在来看看路由本身的含义，我们使用文本串 `/routes/faculty` 来表示路由路径（如清单 4.6 所示）。

清单 4.6 教员列表路由

```
$routeProvider.when("/routes/faculty", {  
    templateUrl : "App/components/simplerouter/partials/facList.html",  
    controller : "facultyController"  
})
```

如前所述，我们希望缺省显示教员列表。清单 4.7 展示了我们的缺省路由。

清单 4.7 缺省路由

```
.otherwise({  
    redirectTo: "/routes/faculty"  
})
```

随着缺省路由的配置完成，在浏览器中打开应用 URL 时将立即重定向到教员列表。

1. 路由执行的功能

在 AngularJS 中，可以指定某个注册控制器的名字作为要执行的功能函数名称。对于其他库或框架，则可能使用其他类型对象或回调函数名称。在这个例子中，只要路由器找到包含 `/routes/faculty` 的 URL，`facultyController` 控制器中的代码就会执行。

教员列表控制器提供在视图中显示的教员列表（如清单 4.8 所示）。我们的样例控制器文件中定义了三个控制器，教员列表控制器是其中的一个，该控制器文件还包含了 `routeControllers` 对象的定义：

```
var routerControllers = angular.module(  
    "RouterApp.controllers", []);
```

要让视图能够使用教员列表，教员列表需要添加到 AngularJS 的 `$scope` 中。如第 2 章所述，`$scope` 是内建对象，其在视图与数据间充当“中间人”角色（有点像视图模型）。

清单 4.8 路由功能

```
routerControllers.controller(
    "facultyController", function($scope) {
        $scope.deptFaculty = [ {
            hrefVal : "manderson",
            displayText : "Dr. Mary Anderson"
        }, {
            hrefVal: "sjones",
            displayText : "Dr. Steve Jones"
        }, {
            hrefVal: "arodriguez",
            displayText : "Dr. Ana Rodriguez"
        } ];
    });
```

定义 facultyController

为教员授课时间添加数据到 \$scope，以便能够在视图中使用这些数据

供视图使用的数组是一个教员对象列表。每个对象有两个 Property：`hrefVal` 和 `displayText`。`hrefVal` Property 用于创建链接 URL。`displayText` Property 包含了教员名字。我们将教员名字作为链接的显示文本。

注意 使用 AngularJS 时一个常用的实践就是将各种类型的业务逻辑放到一个 AngularJS 服务中，但我们在这里并未遵循这条规则，因为我们希望保持示例尽可能简单。

2. 教员列表视图

最后，清单 4.9 展示了视图代码。当路由关联的状态发生改变时，就会显示该视图。该视图将包含清单 4.8 中的数据。请注意，视图模板通过 `ng-repeat` 绑定来使用 `$scope` 对象的 Property，以迭代创建链接列表。

清单 4.9 路由相关视图

```
<section id="fac_list">
  <h1>Our Faculty</h1>
  <section id="faculty">
    <ul>
      <li ng-repeat="faculty in deptFaculty">
        <a href="#routes/officehrs/{{faculty.hrefVal}}">
          {{faculty.displayText}}
        </a>
      </li>
    </ul>
  </section>
</section>
```

`ng-repeat` 用来为 `deptFaculty` 数组中的每个对象生成 `<a>` 标签。`ng-repeat` 绑定通知框架遍历列表并为每个列表条目创建 `` 元素结构。其工作方式类似于 JavaScript 的 `for...in` 循环。

代码执行时, `facList.html` 中的视图将显示。视图模板将渲染每个数组项。清单 4.10 展示了模板渲染之后, DOM 中第一个数组项的情况。现在每个 `<a>` 标签都有相应链接和显示文本。

清单 4.10 渲染模板的 DOM 结构

```
<li ng-repeat="faculty in deptFaculty" class="ng-scope">
  <a href="#/routes/officehrs/manderson" class="ng-binding">
    Dr. Mary Anderson
  </a>
</li>
```

渲染的 URL

渲染的文本

提示 在 AngularJS 中, 我们通过特定指令 `ng-view` 来标识视图渲染部分。可以在 SPA 应用的 Shell 页面任何位置上使用该指令, `$route` 服务将自动发现它并在那儿放置对应视图。由于这是 AngularJS 特定语法, 因此不同 MV* 框架的语法方式也各有不同。

4.4.2 主要联系人路由

站点导航标头上还有一条到虚构系部主要联系人视图的链接。该链接代码指向一个片段标识符 URL:

```
<a href="#routes/contact">Contact Us</a>
```

当单击该链接时, 浏览器片段标识符由 `#/routes/faculty` 变为 `#/routes/contact` (如图 4.7 所示)。

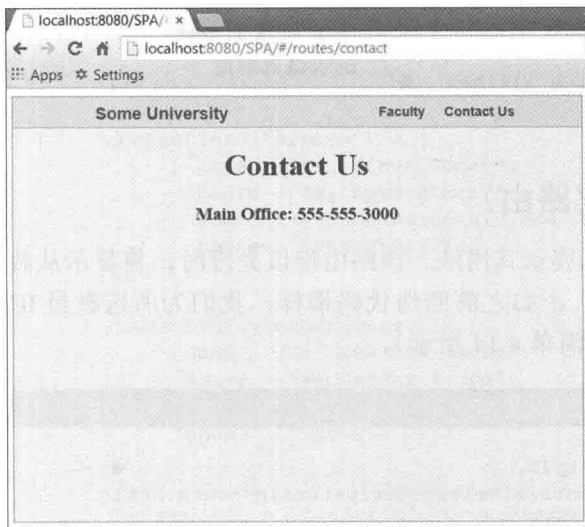


图 4.7 单击“Contact Us”链接, 会在浏览器 URL 中产生一个新的片段标识符

由于路由器实时监听片段标识符的变化，因此能够侦测到新的 hash。之后路由器搜索所有的路由配置项，以找出 `/routes/contact` 的匹配路径。找到后，清单 4.11 中的路由就会执行。

清单 4.11 主要联系人路由

```

    .when("/routes/contact", {
      templateUrl : "App/components/simplerouter/partials/contact.html",
      controller : "contactController"
    })
  
```

路径 → 要执行的功能 ← 要显示的视图

路由相应的控制器显得有点不自然，但它通过提供新的视图，便于我们演示导航操作（参见清单 4.12）。再次说明，所有真实的业务逻辑都应该放在另一个组件中，如 AngularJS 服务。

清单 4.12 路由相应的功能

```

routerControllers.controller(
  "contactController", function($scope) {
    $scope.mainOffice = "555-555-3000";
  });
  
```

← 定义 contactController
← 在视图中显示的 Property

`contactController` 中只有一个 Property：办公室联系电话。清单 4.13 展示了路由对应的视图。其通过表达式绑定展示了该电话号码。

清单 4.13 路由对应的视图

```

<section id="contact_us">
  <h1>Contact Us</h1>

  <h3>Main Office: {{ mainOffice }}</h3>
</section>
  
```

← 主要联系人电话的表达式绑定

4.4.3 教员授课时间（参数化路由）

我们来创建一个参数化路由，以展示其用法。该路由得以处理时，将显示从教员列表视图选择的教员的授课时间。正如之前的伪代码那样，我们为所选教员 ID 先来创建一个带有占位符的路由（如清单 4.14 所示）。

清单 4.14 带有参数的教员授课时间路由

```

    .when("/routes/officehrs/:facultyID", {
      templateUrl : "App/components/simplerouter/partials/hours.html",
      controller : "hoursController"
    })
  
```

← 路由参数传入的是某位教员的 ID

不同类型的路由器其路由参数的语法也不同。具体请参考你选择的路由器文档。本例中使用 `facultyID` 作为变量名。

1. 带有传递文本的相对 URL

如果观察链接源代码，会发现每条链接都指向同一路由，不同之处只在于 URL 的最后部分。下面是第一个教员的链接：

```
<a href="#/routes/officehrs/manderson" class="ng-binding">
  Dr. Mary Anderson
</a>
```

链接 URL 的最后部分表示教员 ID。它是动态绑定的。由于我们的教员列表中有三个教员，因此每位教员的 URL 都包含不同的 ID。

2. 控制器

要能够使用路由参数传进来的信息，每个框架或库都会提供相关的代码访问方式。AngularJS 提供了相应的 `$routeParams` 对象。清单 4.15 展示了控制器对该变量的访问方式。

清单 4.15 教员授课时间控制器

```
routerControllers.controller("hoursController",
    function($scope, $routeParams) {
        var contactInfo = {};

        contactInfo["manderson"] = {
            name : "Dr. Mary Anderson",
            hours : "Tuesday 12-2pm",
            email : "manderson@someuniv.edu",
            phone : "555-555-1111"
        };

        contactInfo["sjones"] = {
            name : "Dr. Steve Jones",
            hours : "By Appointment",
            email : "sjones@someuniv.edu",
            phone : "555-555-1112"
        };

        contactInfo["arodriguez"] = {
            name : "Dr. Ana Rodriguez",
            hours : "Wednesday 1-3pm",
            email : "arodriguez@someuniv.edu",
            phone : "555-555-1113"
        };

        $scope.info = contactInfo[$routeParams.facultyID]
    })
```

对路由参数进行访问的对象

定义 hoursController

定义授课时间与联系信息

路由参数用于找出正确的联系人信息

在本示例中，传递进来的路由参数将匹配三条联系人信息中的一条。路由参数（教员 ID）作为 `contactInfo` 对象的 `Property` 名称，返回相应的教员信息。之后就可以将返回信息填充到 `$scope` 变量，以让视图显示信息。清单 4.16 提供了路由对应的视图。

清单 4.16 授课时间视图

```
<section id="office_hours">
  <h1>Office Hours</h1>

  <p class="hrs_faculty_name">{{ info.name }}</p>
  <p>{{ info.hours }}</p>
  <p>{{ info.email }}</p>
  <p>{{ info.phone }}</p>

</section>
```

通过表达式绑定显示
info 对象的 Property

现在来看看本示例的最终效果。当用户单击教员列表视图中的第一条链接时，教员 ID `manderson` 将通过路由参数传入控制器，展示效果如图 4.8 所示。

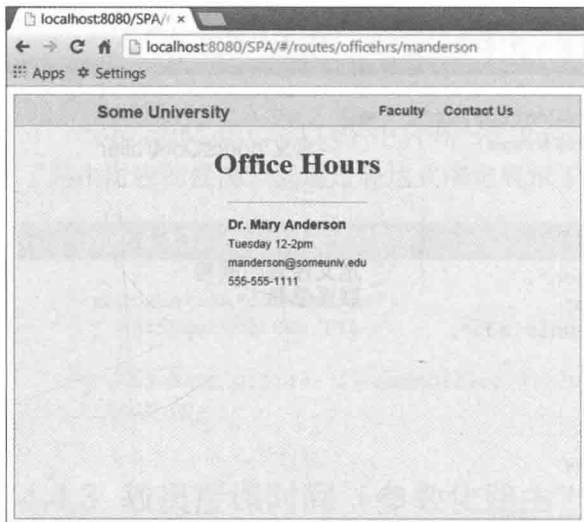


图 4.8 通过路由参数传入 `manderson`，最终显示相应的授课时间

我们了解了路由功能，掌握了路由器用法，可随时回头复习本章内容。

4.5 挑战环节

现在来测试一下你对本章知识的掌握情况：创建简单的图片视图。找几张图片，

并通过客户端路由来显示它们。你可以使用不同的路由并为每张图片生成视图。如果你打算让挑战更有趣一些，可以试着使用一个路由并把图片名称作为路由参数。之后，使用路由参数中传入的图片名称，通过 MV* 框架与动态路由参数来选择图片。

4.6 小结

本章确实有非常多的内容需要消化。现在来看看你都学到了哪些内容：

- 路由器是允许你为给定应用 URL 指定所需状态的库 / 框架。
- 路由在路由器配置中设置。
- SPA 应用路由在浏览器发生，不需要服务器端请求。
- 路由器通过两种方式之一来构建客户端路由——片段标识符方式或 HTML5 历史 API 方式。
- HTML5 历史 API 方式通常需要在路由器配置中显式声明以使用它。其还需要对代码做些改变，包括服务器端代码的改变。
- 通过在路由路径中结合使用路由参数，同一路由可以用来显示不同的结果。路由参数是定义在路由路径中的变量，其允许信息通过 URL 传递。

5 视图合成与布局

本章内容

- 介绍布局设计
- 视图合成的步骤
- 如何设计复杂路由
- 如何处理嵌套与并列视图

到目前为止我们学习了一些单页面应用程序构建的基础概念。在这个学习旅程中，我们了解了如何模块化代码、如何利用 MV* 框架的强大威力创建用户所见并与之交互的视图。同时，还探索了路由器在应用中所发挥的至关重要作用：其不仅允许我们在 SPA 应用中使用浏览器原生的导航特性，还提供了一种配置 URL 关联功能和视图的方式。

然而之前的几章都是专注特定概念，而非 SPA 应用全景。我们的 SPA 开发工具箱里到目前已经积攒了一些工具，接下来聚焦并处理 SPA 应用的整个设计过程。设计一个成功的 SPA 应用有点像学讲一门新语言：掌握了基础——词汇与语法——但是，不代表你掌握了开启流利口语的钥匙。

在本章中，我们将开启横贯整个 SPA 应用设计的旅程，从可视化布局开始，直到将需求转换为一个现实可用的应用。我们会学到用视图取代页面来设计布局，并设计路由以串起这些布局，之后用代码将各部分整合在一起。尽管我们已经理解了 SPA 应用的底层机理，但通过更加聚焦于整个过程，有助于做到举一反三。在此过程中，我们还将了解如何处理更贴近现实的复杂场景，比如多视图之下的复杂布局及路由。

5.1 项目介绍

本章的项目是为一家虚拟医疗公司创建在线工具。假定这个 SPA 应用用于销售代表跟踪他们的客户订单。由于本章的焦点是整个设计过程，因此该项目的布局会比前几章的例子更复杂一些。应用需要为某条给定路由显示几种类型的信息，诸如客户数据、订单历史以及客户的账单地址与发货地址。

随着设计过程的深入，后续会涉及更多的项目具体细节。现在先来看看项目最终的效果（如图 5.1 所示）。这是项目构建各个阶段都完成后的总览图。

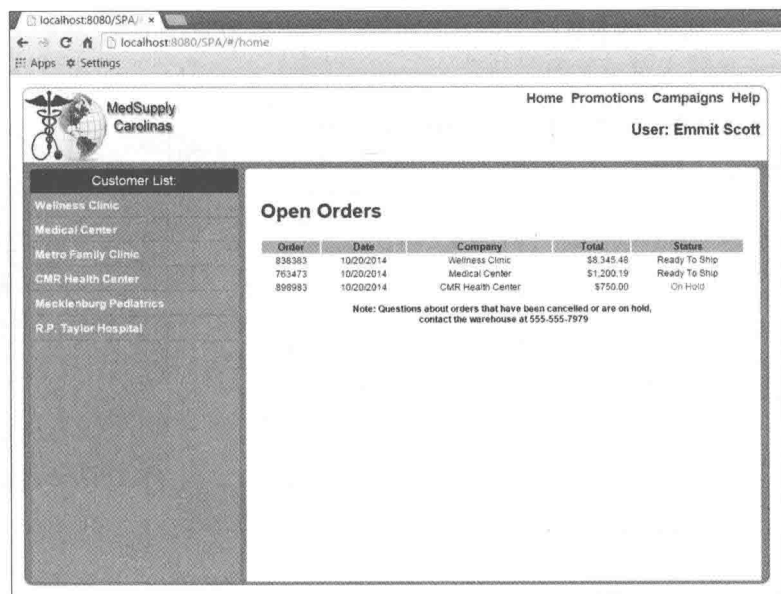


图 5.1 示例项目跟踪虚拟医疗公司的订单状态，其包含一个比之前例子更加复杂多变的布局

然而在开始之前，先来讨论一些布局设计的基本概念。首先我们将确保已掌握概念和新概念间的融会贯通。

5.2 布局设计概念

到目前为止，我们讨论的 SPA 概念仍只框定在一个狭小范围内。而现在我们将从整体角度审视 SPA 应用全景。在此过程中会揭示一些必须掌握的新细节，以便能够开始本章项目的实作。

5.2.1 视图

在设计 SPA 应用视图的时候，我们创建 SPA 应用完整拼图的每个小部分。每个部分都为用户提供特定效果，不论其仅仅是为了显示数据，还是为了提供用户输入。一般而言，视图设计有两个层面：基本层面——设计视图本身；在更广泛的层面上，则更关心视图如何适应整体架构。

从视图内部来看，我们的设计工作集中于诸如显示数据、添加 JavaScript 交互功能等任务。如前所述，MV* 框架能够帮助我们分离各部分应用代码，但又能保持应用的整体协同。HTML 元素与绑定功能相结合构成了视图设计的基础。通过 CSS 将样式应用到元素上能够进一步丰富我们的视图设计（如图 5.2 所示）。

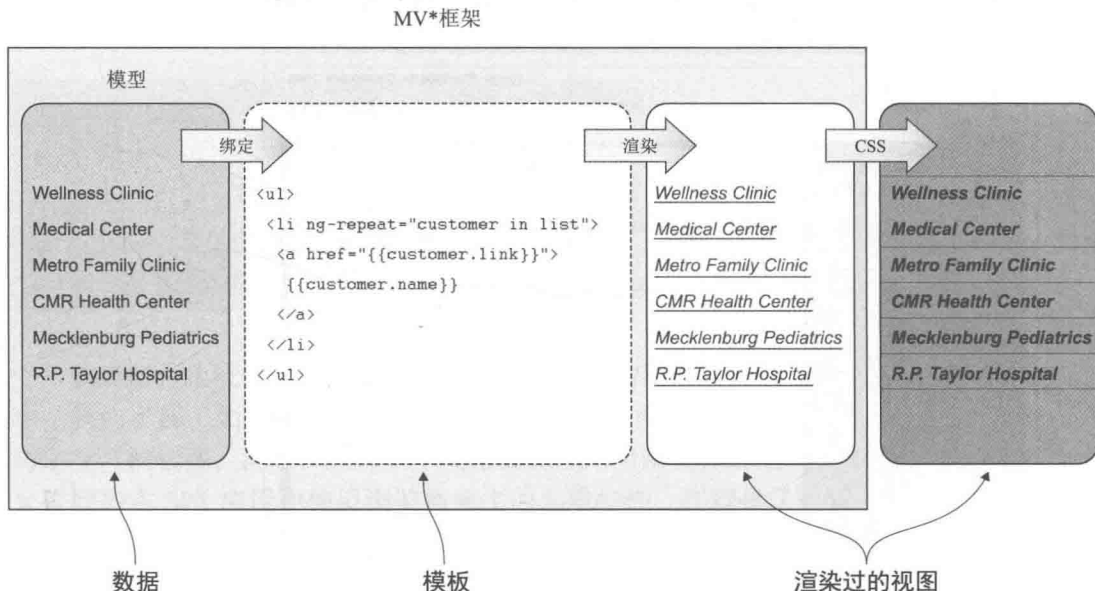


图 5.2 模板的 HTML 代码构建了初始结构，但由 CSS 修饰其外观

当涉及更大的场景（布局）时，我们不得不在功能上思考每个视图与其他视图

间的定位。为了在屏幕特定位置定位一个渲染视图，我们将求助于被称为 *Region* 的构造形式。

5.2.2 Region

第 1 章简短介绍过 *Region* 的概念。尽管特定的视图引擎对该术语有其自己的观点，但第 1 章提到的 *Region* 是指屏幕上设计为包含一个或多个视图的某个区域。如果你使用了 HTML5，*Region* 可以定义成使用语义元素（如图 5.3 所示）；如果未使用 HTML5，则可以使用诸如 `<div>` 这样的元素。这些类型的元素都是合适的，因为它们对用户不可见，但又能够用来在 UI 中定义实际空间。

要定义 *Region* 的大小及其与别的 *Region* 间的美学关系，可以使用层叠样式表（CSS）。在图 5.4 中，我们可以看到一个给 *Region* 应用样式以完成布局构建的场景。

该场景通过给 *Region* 分配一个简单的 `float` Property，并排定位了两个 *Region*。对于简单的 2×2 布局，我们可以浮动一个 *Region* 到左边、另一个到右边。放置好 *Region*，就可以来确定它们要包含的视图了。

还可以分配其他的 CSS Property 给 *Region*，以进一步增强布局设计的效果，诸如 `width`、`height`、`padding`、`borders`、`margins` 以及 `background-color` 等。任何应用到此类所选元素的 Property 都能达成某种特定效果。

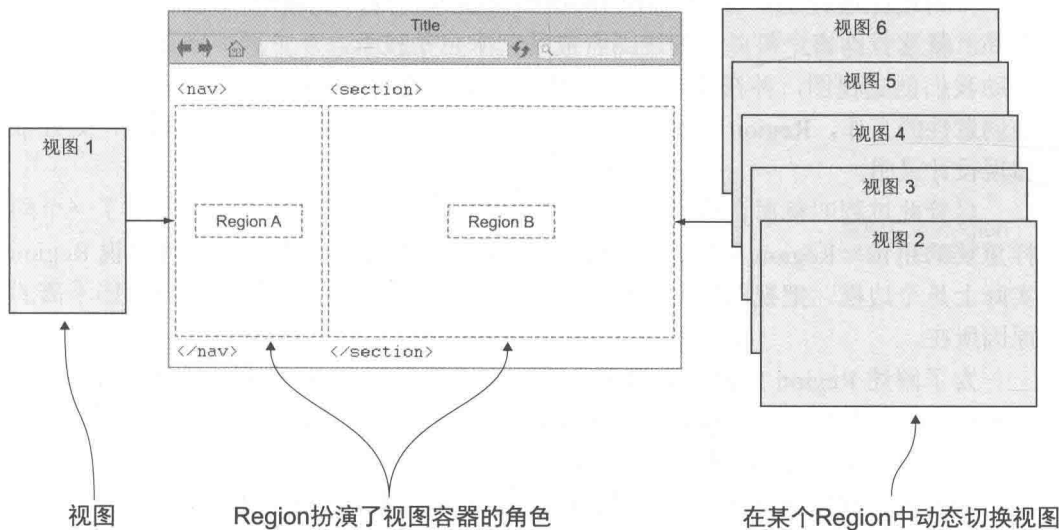


图 5.3 *Region* 提供了 UI 中的实际空间来显示视图。在一个 *Region* 中，视图可以修改及动态切换

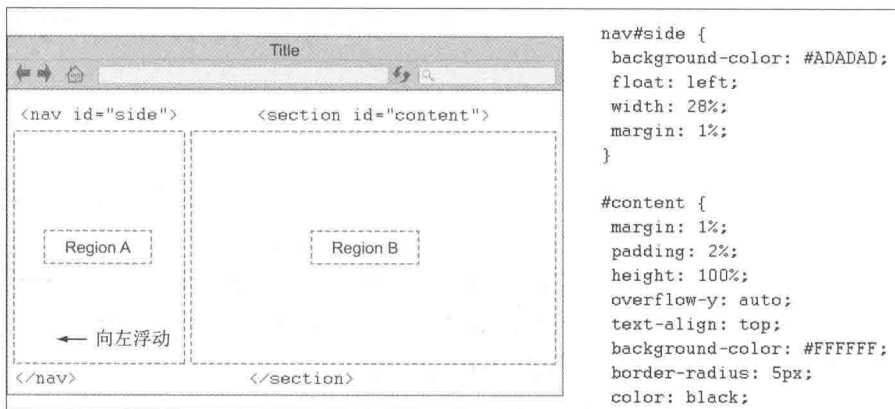


图 5.4 CSS 用来定义布局中 Region 的实际属性，并定义某个 Region 与 UI 中其他 Region 的关系

敲定 Region 的大小与形状，以及在这些 Region 中布置视图以实现具体布局的过程被称为视图合成（view composition）。

5.2.3 视图合成

作为设计过程的一部分，我们的视图布置或合成，都按照一定方式来组织 UI 布局。就这点而言，可以认为视图合成是艺术也是技术。一方面，通过这种技术来帮助我们创建视图，并在特定路由执行时，视图得以在 UI 中展示。另一方面，这是创意性的工作，Region 怎么放、一个 / 一系列视图在 Region 中怎么布置，处处都体现设计灵感。

尽管此过程叫视图合成，但如前所述，Region 在布局创建过程中扮演了一个同样重要的角色。Region 里放置了 MV* 框架要渲染的视图。鉴于此，可以说 Region 实际上是个边框，把视图框定在布局中。这就是视图合成中视图与布局形影不离的原因所在。

为了阐述 Region 如何影响视图合成，来看看图 5.5。这里面的视图与图 5.3 中的相同。通过重新排列 Region，显示的是相同的信息，但布局却完全不同了。相应地，这直接影响到 SPA 应用的外观。

Region 与视图如何布置完全是你的主观决定，得根据你的项目目标和设计喜好量体裁衣。

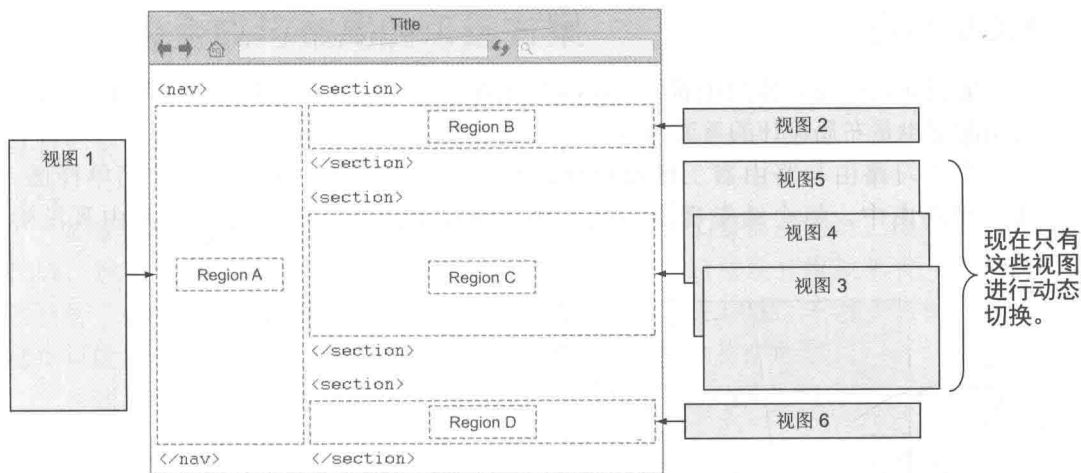


图 5.5 Region 与视图如何排列会影响视图合成及最终布局

5.2.4 嵌套视图

要记住的一个很重要的事情就是 Region 的使用并不局限于 SPA 应用的 Shell。Region 也可以用于视图以嵌套其他视图（如图 5.6 所示）。

嵌套视图可能大大增加设计复杂度，但有时候考虑到所构建功能的性质，这是必需的。我们还可以配置应用程序路由，以便应用状态发生改变时设计正确地反映在 UI 中。

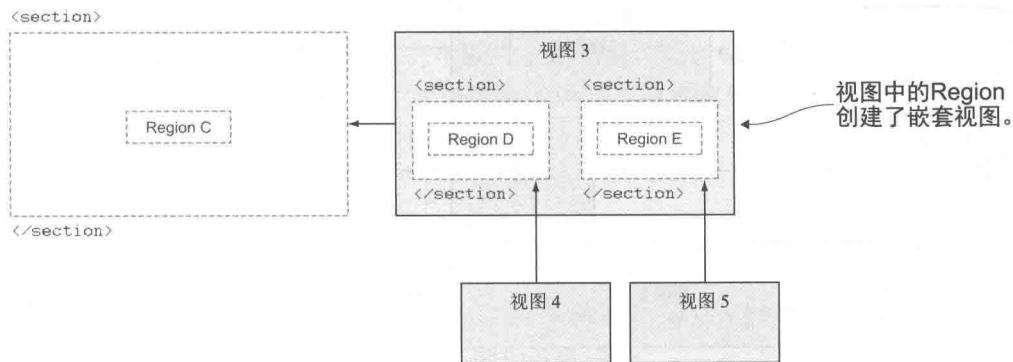


图 5.6 Region 还可应用于视图，以满足需要在视图中嵌套一个或多个视图的场景

5.2.5 路由

如第4章所述，路由配置方式影响应用状态。应用状态包括了UI状态。因此，路由配置也是布局设计的重要考量点。

在学习路由与路由器工作原理时，我们只涉及图5.7所示那样的简单样例。在这种路由中，每个结果视图占据了整个目标Region。设计这样的路由真没啥意思。

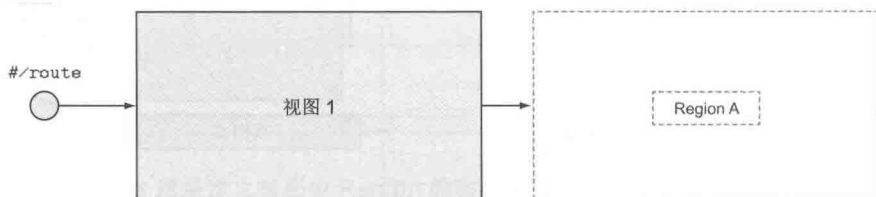


图 5.7 简单路由示例

而就像我们刚才学到的，目标Region可以放置在屏幕的任意多个地方。这会引发某些有趣的设计。例如，我们可以有多个彼此相邻的Region，并为某条给定路由显示多个视图（如图5.8所示）。

当设计的路由会导致复杂的Region配置(或视图配置)时，事情将变得难以管理。对于复杂布局，如果我们不使用一个内置了稳健的路由及视图管理方案的一站式框架，就可能需要专门针对所用MV*框架，来考虑选择某种应用状态管理库。下一节的内容将就采用视图管理库进行利弊分析，同时提供一些可选参考。

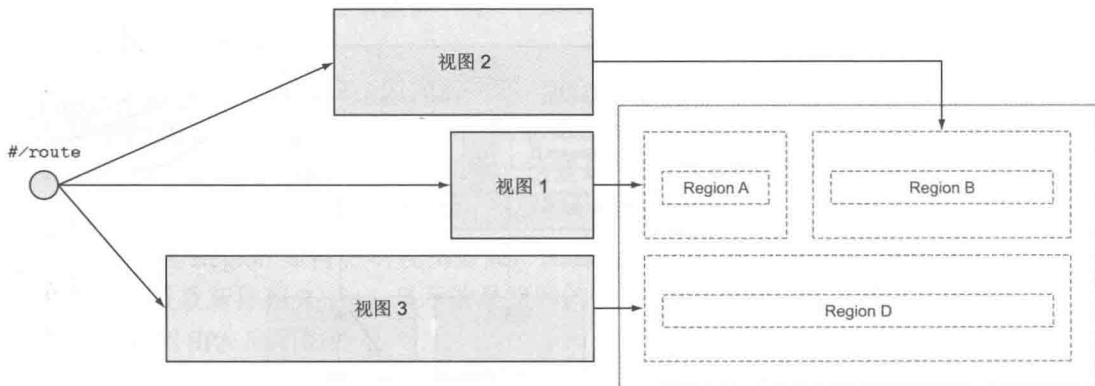


图 5.8 带有复杂视图的路由示例

5.3 高级合成与布局的可选方案

为何要考虑采用内建一系列优良特性的框架？这是有原因的。最直接的原因就是我们曾提到其具备处理复杂布局的能力，特别是那些处理多视图、嵌套视图的能力。

如果所用 MV* 框架不具备某个具体内建功能，我们最终就得自己编写该功能代码。然而，现有框架很多都发展自旧有 MV* 框架。这些现有框架都有一个共同的目标：处理某些复杂任务——诸如高级视图合成和复杂路由，在减少样板代码方面尽可能让开发者来得简单。仅此一条就是考虑外部帮助的充足理由。

考虑这些外部框架的另一个理由就是其可能带给我们额外的功能，比如事件、消息以及方便布局创建的内置对象等。

最后，在复杂设计中，我们不得不处理视图状态。请记住，显示一个视图（或某些视图）时，我们也需与 MV* 框架进行交互。我们不仅仅依赖框架来进行数据处理以及视图交互，还用它来管理视图生命周期。

回忆一下我们评估各种风格的 MV* 框架时的情形，每种风格的绑定及渲染过程都不同。比如 Backbone.js，其通常的做法是销毁之前的视图，再重新通过携带新数据的模板来创建它。而对于 Knockout 这样的 MVVM 风格的框架，绑定只处理一次，视图将保持活动状态。我们仅仅在需要之时跟视图模型进行交互。

每种框架都有需处理的生命周期具体任务，诸如视图何时渲染、展示、隐藏和（如果可以的话）销毁。某些库 / 框架要么为开发者代劳了这些任务，要么提供了开发者使用的钩子。

如前所述，绝大部分的大型端到端框架都具备处理以上大多数或全部关注点的能力。如果你所用框架的内建能力中缺失上述功能，可以试一试这些库 / 框架¹。但首先得考虑一下各种因素。未经过合适的考虑就不应该贸然添加依赖项。因此，在采取行动之前，先来考虑一下各种利弊。

5.3.1 优点

采用库 / 框架的优点如下：

- 如果某个库 / 框架扩展或增强了一个特定的 MV* 框架，则其将围绕着该 MV* 框架的微妙之处进行恰当的创造性设计。
- 这些库 / 框架由经验丰富的开发者开发，他们精通某个 MV* 框架，并理解复杂任务管理的困难和陷阱。
- 我们基本不用担心需要亲自拼凑各种其他的库和框架。

¹ 指前面提到的大型端到端框架。——译者注

5.3.2 缺点

采用库 / 框架的缺点如下：

- 你得忍受开发者的 bug 修复与升级。
- 由于导入的外部库 / 框架不属于我们的代码，且不是核心 MV* 框架的一部分，因此程序错误调试就可能更困难。
- 作者有可能出于任何原因放弃库 / 框架。这样，你依赖的库 / 框架就过时了。

如果需要更稳健的 SPA 应用方案系列特性，表 5.1 提供了一些可以考虑的选项。这些框架凭借其自身实力，都演进得很强大，其提供的特性远不止视图 / 布局管理。表 5.1 并非详尽清单，但提供了本书写作时的一些可参考选项。

表 5.1 内置高级合成与布局特性的框架

框架	针对更复杂合成与布局的选项
Knockout	Durandal (http://durandaljs.com)
Backbone.js	Marionette.js (http://marionettejs.com) Geppetto (https://github.com/ModelN/backbone.geppetto) Chaplin (http://chaplinjs.org) Vertebrae (https://github.com/hautelook/vertebrae) LayoutManager (https://github.com/tbranyen/backbone.layoutmanager) Thorax (https://github.com/walmartlabs/thorax)
AngularJS	AngularUI (http://angular-ui.github.io), AngularJS 的一部分，但作为单独下载项
Kendo UI	内建
Ember.js	内建

如之前选择 MV* 框架时那样，我们可以对这些框架应用相同的挑选标准，考虑的因素包括学习曲线、bug 与修复率、文档、成熟度以及社区支持等。

现在，我们已经了解了基本设计概念，并思考了创建复杂布局的可能软件方案，接下来创建项目。由于我们使用 AngularJS 来阐述，因此会用到 AngularUI 组件。如表 5.1 所述，AngularUI 是整个 AngularJS 软件的一部分，但其必须单独下载。同时它还包括在本章源代码中，可在线下载。

5.4 设计应用程序

先来看看我们需要设计的内容：一个供医疗公司使用的在线报表工具。该工具帮助公司销售代表跟踪订单状态、查阅订单历史，并查阅客户信息。假设我们需要以下功能：

- 提供一个可选择列表，内容为销售代表名下的客户。
- 缺省显示所有的开放订单。
- 当选择某个客户时，显示公司信息、联系方式以及订单历史。
- 初始隐藏客户账单及发货信息，但在需要时使其可见。此外，分配给客户其专属的 URL，以便使用者能够快速到达之前的位置。
- 在标头包含到达公司当前活动、促销及应用帮助的链接。

5.4.1 设计基本布局

首要的任务是必须确定应用程序的基本结构，这跟设计普通 Web 应用程序的思路一样。这个结构通常被称作基本布局（base layout）。应用程序是主从结构吗？或者添加一个导航侧边栏会更好些？需要页脚吗？存在着各种可行的选择。

我们将在项目中使用典型的“三部分式”基本布局。在标头放置链接，导航条里的客户列表位于左下方，主内容区域里的路由显示结果则放在右下方。整个结构的创建分为两个步骤。

首先，将屏幕划分成一个标头区和一个主内容区，如图 5.9 所示。

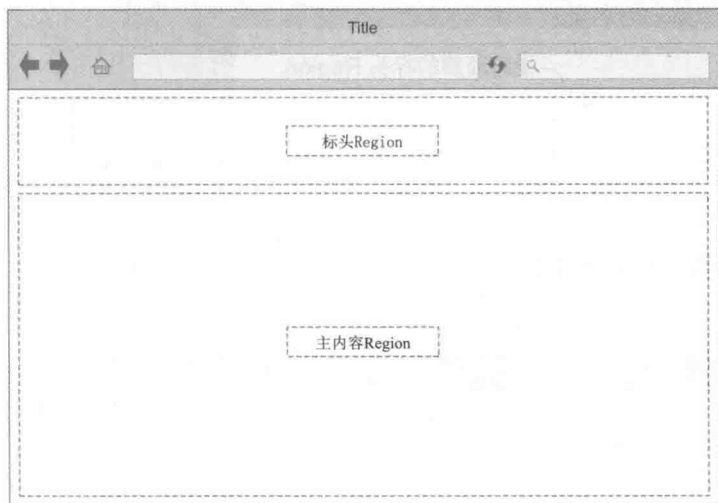


图 5.9 基本布局，首先设计顶部 Region 和底部 Region

接下来，将主内容 Region 划分为两部分：左边的导航 Region 和右边用于显示内容的内容 Region（如图 5.10 所示）。

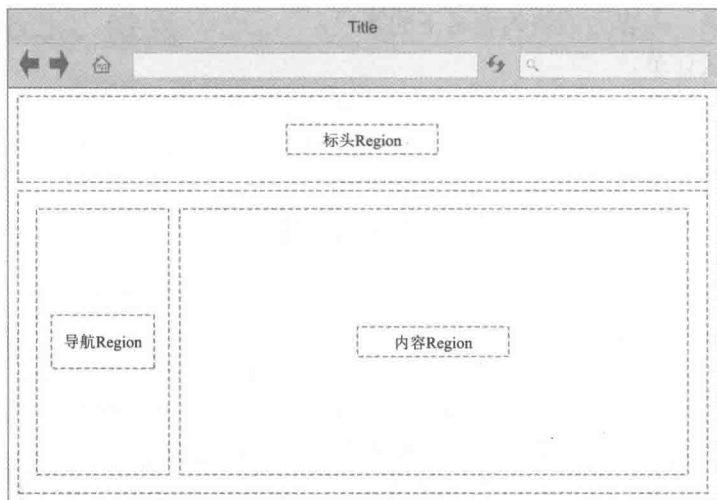


图 5.10 最后完成导航 Region 和内容 Region 的设计

现在，我们的基本布局源代码如清单 5.1 所示。

清单 5.1 基本布局

```
<main>
  <header></header>
  <nav id="side"></nav>
  <section id="content"></section>
</main>
```

横跨顶部的标头 Region

底部的主内容 Region，分割成
导航 Region 和内容 Region

我们会使用 CSS 来给基本布局添加样式。一开始，先给标头添加些高度以及白色背景（如清单 5.2 所示）。

清单 5.2 标头 Region 的 CSS 代码

```
header {
  height: 15%;
  background-color: #FFFFFF;
}
```

我们也要给侧边栏导航 Region 及内容 Region 添加样式，浮动导航 Region 到左边以让这两个 Region 排排坐。同时通过分配部分宽度给导航 Region 来定义这两个 Region 的比例关系（如清单 5.3 所示）。

清单 5.3 导航 Region 与内容 Region 的 CSS 代码

```
nav#side {  
    background-color: #ADADAD;  
    float: left;  
    width: 28%;  
    height: 100%;  
    margin: 1%;  
}  
#content {  
    margin: 1%;  
    padding: 2%;  
    height: 100%;  
    overflow-y: auto;  
    text-align: top;  
    background-color: #FFFFFF;  
    border-radius: 5px;  
}
```

← 导航 Region 浮动到左边，同时设置其宽度以定义这两个 Region 间的关系

基本布局样式处理完毕的效果如图 5.11 所示。

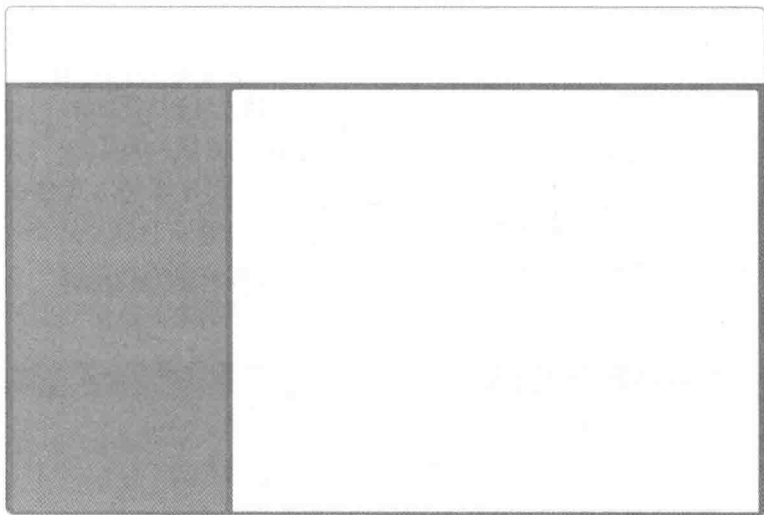


图 5.11 附带样式的基本布局

完成了应用程序的基本布局，就可以着手添加其内容了。

5.4.2 设计基本内容

现在我们可以为 Region 添加内容了。接下来设计视图及缺省路由。由于标头及导航视图是固定的（只有它们的内容会变化），我们就先来设计这两个结构。

1. 标头

标头视图很简单。其有一个商标、几条链接以及登录用户名。由于它是一个固定视图，因此就不需要路由支持。该视图永远不会发生与其他视图的交换。综上，只需包含它就能够将其显示在屏幕上。对于 AngularJS，可以通过添加一个属性到标头 Region 的 `<div>` 来实现所需功能。然而请注意，选择不同的 MV* 框架有可能需要用不同的方法来包含一个视图。

```
<div
  id="header"
  ng-include src="'App/components/customerorders/partials/header.html'">
</div>
```

清单 5.4 为标头视图源代码。

清单 5.4 标头视图

```
<section ng-controller="navController">
  <section id="logo">
    
  </section>
  <nav id="top">
    <ul>
      <li><a ui-sref="home">Home</a></li>
      <li><a ui-sref="promos">Promotions</a></li>
      <li><a ui-sref="campaigns">Campaigns</a></li>
      <li><a ui-sref="help">Help</a></li>
    </ul>
    <p id="userName">User: {{user}}</p>
  </nav>
</section>
```

ui-sref 属性由视图管理库提供，用于执行路由

用户名通过数据绑定方式添加

Home 链接返回缺省内容，其他链接处理其对应需求。视图渲染及添加样式之后，效果如图 5.12 所示。



图 5.12 应用样式之后的标头布局

随着标头处理的完成，接下来设计导航视图。在这个应用里，导航视图中的链接由公司当前客户列表构成。

2. 导航

客户列表视图中的客户列表依然简单。它也是固定的，因此也不需要路由。用一条包含语句，就能加载应用。

```
<div
id="navigation"
ng-include src="'App/components/customerorders/partials/customerList.html'" >
</div>
```

为了创建客户列表中每位客户的链接，需要使用一个绑定来迭代列表。这跟第2章模板与绑定内容中提到的做法一样。我们还需要使用刚才提到的、视图管理组件中的特定链接属性(ui-sref)，在单击某条链接时执行 customerInfo 路由（如清单 5.5 所示）。

清单 5.5 导航视图

```
<div id="listheader">Customer List:</div>
<div id="navButton" ng-controller="customerListController">
  <ul>
    <li ng-repeat="customer in customerList">
      <a ui-sref=
        "customerInfo({ customerID:customer.custNum })">
        {{customer.name}}
      </a>
    </li>
  </ul>
</div>
```

对于列表中的每个客户，创建一条到 customerInfo 路由的链接。当路由执行时，传递客户号作为路由参数。

请注意，当使用核心 AngularJS 路由器时，相应的视图管理器允许我们传递路由参数。在本示例中，customerID 是变量名，customer.custNum 表示在 ng-repeat 属性从模板获取信息后，分配给参数的数据。

随着视图渲染到屏幕，我们可以使用 CSS 将无序列表转换成一组可单击面板的效果。清单 5.6 展示了部分样式设置（完整代码可以通过下载获取）。

清单 5.6 导航样式设置

```
#navButton {
  width: 100%;
  padding: 0 0 1em 0;
  margin-bottom: 1em;
  background-color: #ADADAD;
  color: white;
}
```

背景颜色设置

```
#navButton li {
  border-bottom: 1px solid #979797;
  margin: 0;
}
```

细微的底部边框衬托面板

```
#navButton li a:hover {
  background-color: #D9D8D8;
  color: #fff;
}
```

为 <a> 标签添加 hover 选择器，实现反转效果

导航视图渲染之后，缺省内容的效果如图 5.13 所示。

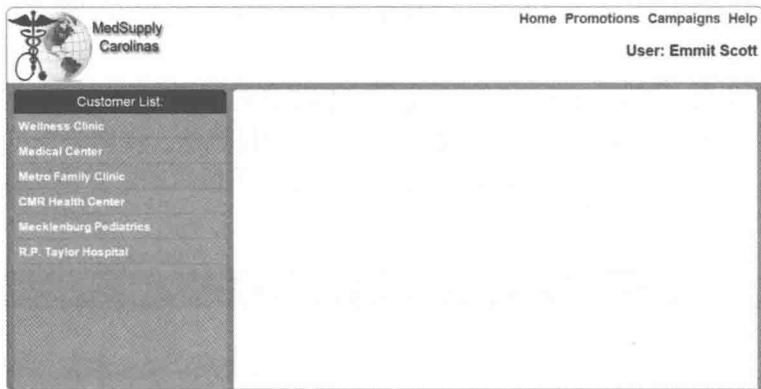


图 5.13 标头及导航视图都渲染后的布局效果

随着固定内容部分的完成，接下来关注动态部分。

3. 缺省路由

根据需求，默认情况下，用户进入网站时应该看到开放订单列表。因此，需要创建一条缺省路由。该路由的视图结果是用户刚进入站点或单击 Home 链接时看到的内容。

清单 5.7 展示了缺省路由的设置。如前所述，我们使用 AngularUI 路由器。其语法与核心 AngularJS 所提供的稍有不同，但并非完全陌生。该路由器更加强大，且特意围绕状态而非 URL 来打造。同时它还支持多视图，包括嵌套及并列视图。

请记住 AngularUI 路由器的语法有些特殊。但别太担心。对支持高级状态管理的大多数框架而言，基础概念都是通用的。

清单 5.7 缺省路由配置

```

状态名 → .state("home", {
    url: "/home",
    templateUrl: "App/components/customerorders/partials/openOrders.html",
    controller: "openOrdersController"
})
                                ← 定义缺省路由为 home
.otherwise("/home");
  
```

从该模板代码可以看到，缺省展示所有开放订单的需求，随着这里的路由完成配置，也就实现了。关键字 `otherwise` 代表缺省路由。

为了指定 `content Region` 作为本条路由所渲染视图的容器，需要在 `Region` 的 `<div>` 标签中应用 `ui-view` 属性。


```
<div id="content" ui-view></div>
```

不管使用哪一种视图管理解决方案，都得标识出视图所在的 Region。根据你使用的视图管理器查看相应文档，以了解更多信息。

在开放订单视图中，我们再次使用了 repeat 绑定指令来通知 AngularJS 重复生成开放订单表的每行数据（如清单 5.8 所示）。

清单 5.8 开放订单视图的行数据模板

```
<tr ng-repeat="rorder in recentOrders">
  <td class="orderData">{{rorder.orderNumber}}</td>
  <td class="orderData">{{rorder.date}}</td>
  <td class="orderData">{{rorder.name}}</td>
  <td class="orderTotal">{{rorder.total}}</td>
  <td class="orderData"
    ng-class="rorder.status == 'On Hold'
    ? 'orderOnHold' : '' rorder.status}</td>
</tr>
```

重复每条订单

绑定类，以针对
“On Hold” 状态应
用不同类

随着缺省内容的填入，来看看目前的效果。效果跟图 5.1 是一样的，但还是再次在这里感受一下其全貌吧（如图 5.14 所示）。该图展示了当前我们完成的工作。

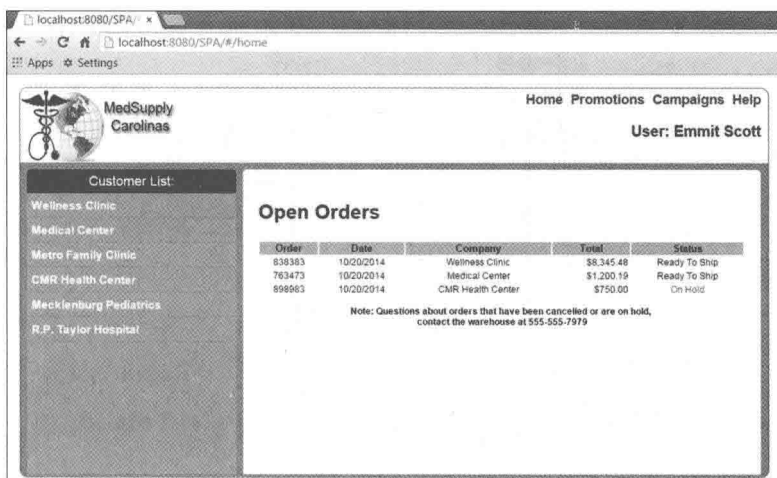


图 5.14 缺省路由设置好以后的布局情况，内容 Region 中展示了开放订单

随着缺省内容的完成，我们将到达本应用的最后一站：显示所选客户的相关信息。在这里，视图合成变得更复杂了，而我们也会因此尝到视图管理组件带来的好处。

5.4.3 在复杂设计中应用视图管理

直到现在,我们的应用程序还没有任何普通路由器与 MV* 框架无法处理的功能。然而,接下来在选择导航视图中的某个客户时,需要在内容 Region 中同时显示多个视图。

此时我们的功能设计包括一个主客户视图,其需要在客户信息路由中配置。然而,个中复杂度在于这个新视图将包含一些它自己的 Region,用来放置客户联系信息及订单历史。反过来说,这些 Region 还包括它们自己的视图。我们将会在与主视图相同的路由中配置它们。我们可以假设这三个视图都很复杂,以至于需要单独开发与管理。

本章早些时候还提到,Region 的位置并不局限于 Shell 本身。为了设计出满足需求的最合适布局,在视图使用 Region 是我们可以考虑的又一种视图合成策略。图 5.15 展示了此时我们的 Region 与视图布置。

有了这么一副行动指南图,接下来看看所选视图管理组件的路由及视图合成扩展能力是如何为我们所用的。

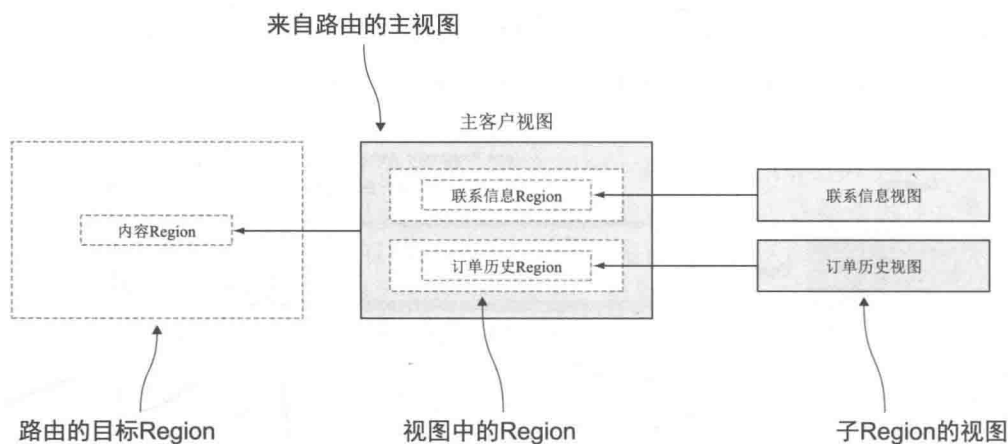


图 5.15 为了构造客户信息功能,需要在主视图自己内部放置额外 Region,以包含相关却又独立的联系信息及订单历史视图

下面开始构建路由。在了解如何配置此类路由之后,再要理解剩余代码的含意就变得更简单了。清单 5.9 展示了多状态路由的配置。

清单 5.9 多状态下的客户信息路由

```

    .state("customerInfo", {
      url: "/customerInfo/:customerID",
      views: {

        // 主客户视图
        "": {
          templateUrl: "App/components/customerorders/partials/
            customerMain.html",
          controller: "customerController"
        },

        // 联系信息与客户历史相关的内部视图
        "contact@customerInfo": {
          templateUrl: "App/components/customerorders/partials/
            customerContact.html",
          controller: "customerContactController"
        },

        "history@customerInfo": {
          templateUrl: "App/components/customerorders/
            partials/customerHistory.html",
          controller: "customerHistoryController"
        }
      }
    })

```

状态名

带有参数定义的路由路径

联系信息与客户历史相关的内部视图

再次地，如果使用不同视图管理器，语法也就跟这里的不一样，但概念是相同或类似的。由于该路由的复杂度，因此我们需要对其多加解释，通过对所见的 AngularUI 路由语法进行分析，以确保真正理解它。

1. 路由分析

此类型路由具备的内容与普通路由相同（路径、视图以及功能函数）——但在数量上是其三倍。该类型路由的配置说明如下：

- 此状态的缺省视图无须命名。
- 每个视图的绝对名称都包含了对应 Region 的 ID 以及状态名，并通过 @ 串起来（一会儿我们就会看到主视图的源代码）。

我们还能看到视图列表中的每个视图都可以有其自己的功能函数。这是可选的，但很方便，因为对此类型路由而言，好处在于拥有视图相关的功能，同时又能够独立开发此功能。此外，受益于视图管理器的扩展功能，我们能够与每个内部视图及其功能函数共享路由参数。

2. 路由的主视图 / 外部视图

我们已看过路由代码，接下来看看它的主视图代码（如清单 5.10 所示）。这

个视图很重要，因为它包含了其自身信息的绑定，以及其他两个视图待渲染的 Region。

清单 5.10 来自 customerInfo 路由的主视图

```
<h1 ng-bind="custInfo.name"></h1>
```

通过数据绑定来显示
所选客户名称

```
<h4>Customer #
```

```
  <span ng-bind="custInfo.custNum"></span>
</h4>
```

通过另一个数据绑定来显示
所选客户的客户号

```
<section ui-view="contact"></section>
```

```
<section ui-view="history"></section>
```

子 Region 的名称必须对应
路由配置中的视图条目

两个内部视图的代码也是典型的实现。它们通过绑定来显示客户相关的数据。前面提到的路由参数是共享的，内部视图也能够用它来定位正确的数据条目。可以在完整的在线源代码中找到对应实现。

现在展示一下应用程序效果，看看我们做了啥（如图 5.16 所示）。

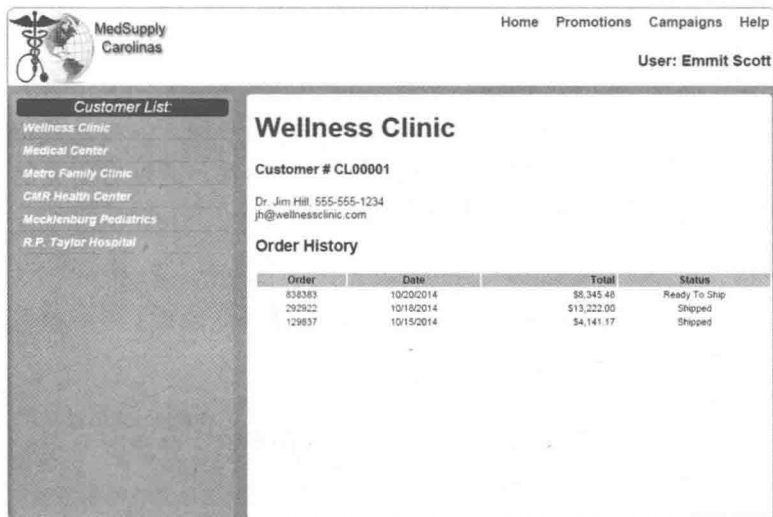


图 5.16 三个视图在内容 Region 中的展示效果

屏幕截图展示了三个应用了样式的渲染视图。无法告知你这三个渲染视图的确切位置，但它们确实是划分成每一部分来设计的。虽然视图是分隔出来开发与管理的，但内容呈现于用户而言却是宛如无缝的一个整体页面。再来看屏幕截图，但这次我们给出了客户信息路由中每个视图的位置（如图 5.17 所示）。

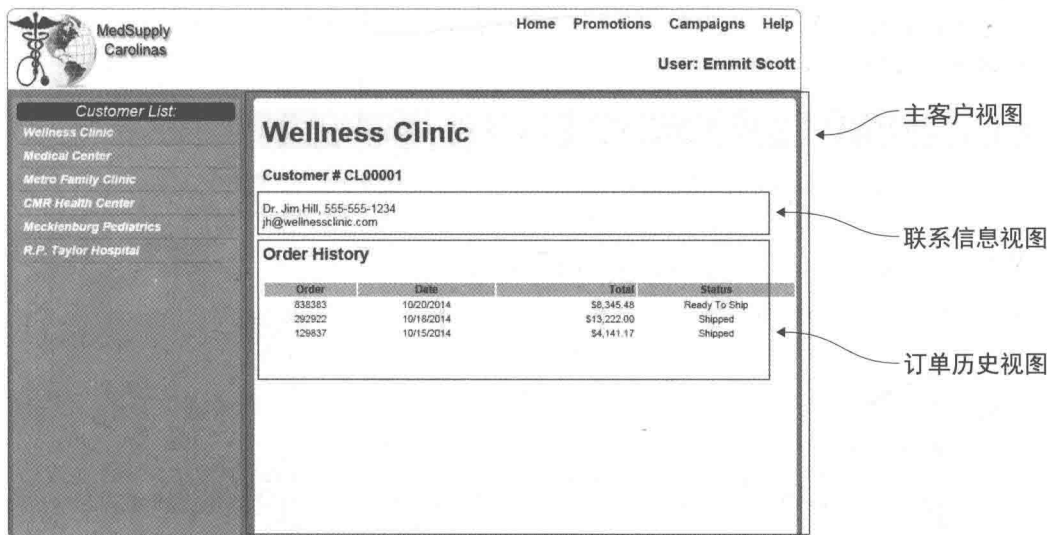


图 5.17 突出显示客户信息路由中的视图

该项目还有一个尚未讨论的重要内容——使用嵌套视图。

5.4.4 通过自身状态创建嵌套视图

为了满足查询账单及发货信息的需求，这两个功能都有其自己的状态，我们将再次依赖视图管理软件。清单 5.11 展示了这些视图的路由。

清单 5.11 账单及发货信息的路由

```
.state("customerInfo.shipping", {
  url: "/shipping",
  templateUrl: "App/components/customerorders/partials/billShipInfo.html",
  controller: "customerShippingController"
})

.state("customerInfo.billing", {
  url: "/billing",
  templateUrl: "App/components/customerorders/partials/billShipInfo.html",
  controller: "customerBillingController"
});
```

子路由通过点号表示法来定义
(parentRoute.childRoute)

请注意子路由通过配置文件中的另一条路由结合点号表示法定义。在这里，我们还定义了客户信息路由作为父路由。

现在我们需要了解如何构造链接以到达这些子路由。其中会发生一点奇妙的事情。为了到达这些子路由，我们也在链接中使用了点号表示法。因此，现在来把它们（指

这些链接) 加到之前的主客户视图, 并放置到添加的元素中。清单 5.12 展示了主客户视图的改进代码。

清单 5.12 添加了账单及发货信息的主客户视图

```
<h1 ng-bind="custInfo.name"></h1>

<h4>Customer #
  <span ng-bind="custInfo.custNum"></span>
</h4>

<section ui-view="contact"></section>

<section ui-view="history"></section>

<nav class="customerInfoNav">
  <a ui-sref=".shipping">View Shipping Info</a>
  <a ui-sref=".billing">View Billing Info</a>
</nav>

<section ui-view></section>
```

通过点号表示法定义到嵌套视图的链接

显示账单及发货视图用的空 Region

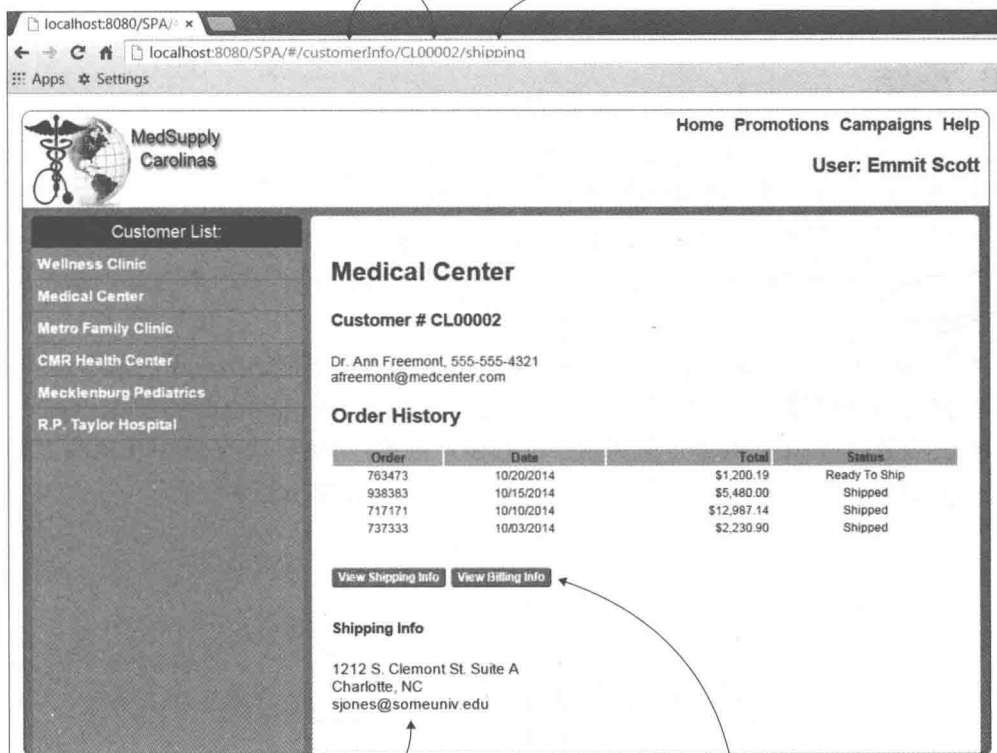
再一次, 你所用视图管理器的语法可能跟这里不尽相同, 但我们将从总体上讨论这里的添加过程, 以便大家都能够理解中心意思。我们为客户信息视图添加了 `<nav>` 元素 (导航), 以显示账单或发货信息。添加的其他元素是一个空 `<section>` 元素, 用来展示来自路由的结果视图。

另一个需要注意的事情是, 出于理解代码的目的, 我们通过 AngularUI 路由器, 将未命名 Region 作为路由的“杂项视图”容器使用。联系信息及订单历史是具名 Region, 因此对应视图将直接分配到同名 Region。当路由的视图找不到名称匹配的 Region 时, 其将关联到第一个未命名 Region。这是 AngularUI 的特殊约定, 还请注意这一点以避免困惑。

现在还剩最后一步了, 我们将给链接添加一些 CSS 属性, 让它们看起来像个按钮, 通过单击该按钮我们能够看到所需内容。图 5.18 展示了按钮风格的链接, 以及路由执行后其中一个子视图的内容。

根据需要, 用户可以单击相应的按钮来展示账单或发货信息, 之后单击浏览器的后退按钮返回到之前的 SPA 应用状态。

客户信息路由及参数 嵌套视图的路由



发货信息视图

添加了样式的链接

图 5.18 嵌套的发货视图，其有自己的 URL

5.5 挑战环节

检验你对本章掌握程度的挑战环节开始了。假设需要你为本地一家园林公司开发一个网站。该公司希望为其每个服务提供一个视图：草坪维护、景观美化、灌木修剪及住宅外部装饰等。该公司还需要一个产品视图以展示其建造的定制平台、喷泉及游泳池。你可以简化每个视图的具体细节，比如用一条描述内容来替代：“landscaping view”，或通过一些模拟内容让它看起来更像一回事。

将屏幕划分成三个 Region，一个用于标头，一个用于导航，另一个用于显示内容。每个服务都拥有自己的视图和关联的导航链接。导航 Region 中也有一个到产品视图

的导航链接，但产品视图又细分为自己的产品标头和产品细节 Region。产品标头应该放置一些链接，这些链接通到每个产品的视图，每个产品的视图显示在产品细节 Region 中。

5.6 小结

本章的内容很多，但你走过来了！我们来做个快速回顾：

- 在 SPA 应用中，我们通过视图合成过程来创建布局。
- 使用语义元素定义 Region，Region 作为屏幕实际空间的占位符。通过 CSS 来设置元素样式和位置。这些措施会影响视图的呈现方式，并直接影响布局设计。
- 首先，一系列 Region 构建在基本布局之上，基本布局描绘出应用的基本结构。
- 当布局设计过程中需要多视图和 / 或嵌套视图时，Region 也能够应用于视图。
- 那些生成视图及 Region 复杂组合的路由是很难管理的。具备路由及视图管理强健能力的框架 / 库，则能够从应用状态的视角抽象出复杂度。

6 模块间交互

本章内容

- 模块概念回顾
- 模块间交互
- 使用模块依赖
- 使用发布 / 订阅模式

第 3 章阐述了模块化编程的大量知识。其中最重要的思想是，通过模块模式来内化逻辑复杂性并提供功能对应公有 API。这是 JavaScript 的封装实现方式。

如你所见，模块化的代码能够将应用逻辑组织成单一目标的更小单元，其更容易管理及修改。毫无疑问这带来了更好的可重用性。模块还有助于保持数据完整性、代码组织以及避免命名冲突。毕竟我们为无刷新的单一页面编写代码，如果不按这种设计方式编写代码，纯粹依赖全局变量和函数的话，代码将很快变得难以管理（如图 6.1 所示）。

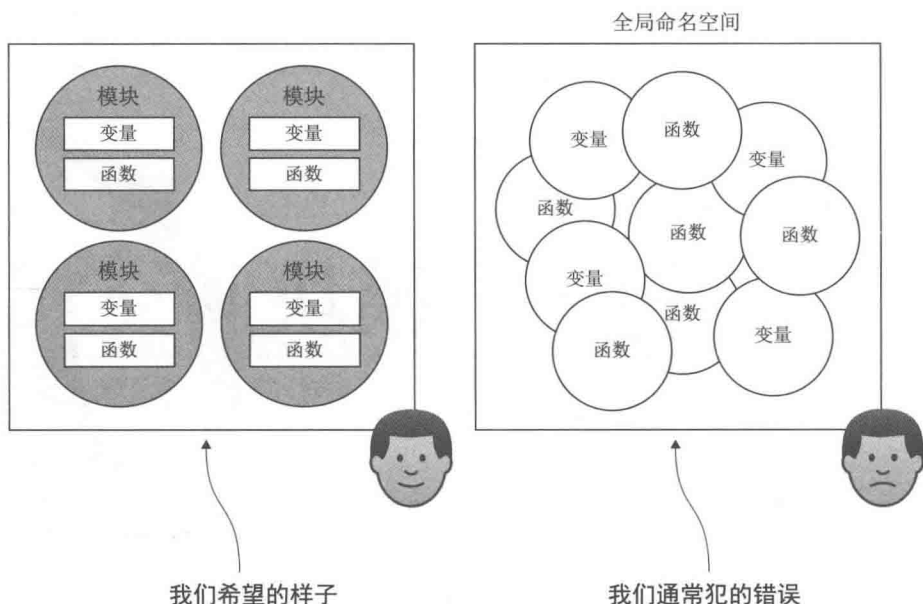


图 6.1 随着项目规模的增长, 如果将变量和函数都放进全局命名空间, 代码会变得越发难以控制

尽管模块本身是模块化编程的核心, 但要用它们创建成功的 SPA 应用, 还需要更深入掌握其工作机制。我们也还需理解它们的交互实现: 一个模块如何调用另一个模块的功能? 也许还需要收到另一个模块的响应。

本章继续讨论模块, 但这次通过我们设计模块交互的方式来打造 SPA 应用架构, 并在此背景下展开模块的讨论。本章首先对模块结构的概念做一个总体回顾, 但主要关注模块间交互过程的设计。

注意 由于模块是基于多个模式风格的 (诸如传统模式、揭示模式、AMD 以及 AngularJS 风格模块等), 我们将尽量保持所讨论概念的共通性。而在其他章, 将通过具体例子包含需强调的内容, 这些例子都可以在线下载。

在这里假设我们要为你的朋友创建一个在线商店, 用于销售二手电视游戏。我们不必太过关注购物车的复杂度, 而要把重点放在商店产品的搜索功能上。虽然该应用比较简单, 但能够让我们无须编写大量代码, 同时又可以创建多个模块, 并设计它们之间的交互方式。

如前面章节的方式, 项目具体细节放到后面讨论。尽管这个项目的界面很简单, 但我们将确保模块实现足够有趣。在讨论模块间交互方法之前, 有必要先来回顾一下模块化编程的几个基础概念。

6.1 模块概念回顾

现在来总体回顾一下模块的基础概念。我们将以这些概念作为基准，来讨论余下的内容。

6.1.1 用模块封装代码

由于在本书写作时 JavaScript 规范尚无内建语法用于创建针对代码封装的模块或类，因此对代码的封装通过模块模式来模拟实现¹。

注意 下一个 JavaScript 版本——ECMAScript（也被称为 *Harmony* 或 *ES.next*），在语言中添加了模块的正式支持。

模块对于 JavaScript 而言是特殊的构造函数。该类型函数通常被称为立即调用函数表达式（immediately invoked function expression, IIFE）。

立即调用函数表达式

在此提醒，模块模式的外层函数通常被称为立即调用函数表达式（或 IIFE），因为开发者将它写成函数表达式（不以 `function` 关键字打头），而非函数声明，同时其后跟一对圆括号以实现立即调用。

IIFE 的语法看起来像这样：

```
var x = (function() {  
    // 一些处理逻辑  
})();
```

若要掌握更多函数表达式的内容及其与函数声明的区别，可参考以下这个不错的资源：<http://javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions>。

清单 6.1 是一个典型模块模式的用法。其创建一个模块对某个产品的价格进行打折。产品价格通过 `calculate` 函数传入，并返回新的折扣价格。

清单 6.1 典型模块模式

```
var pricingSvcMod = (function() {  
    var discountRate = 40;  
    function calculate(amt) {  
        if(isNaN(amt)){  
            return 1;  
        }  
    }  
})();
```

模块内部使用的代码

全局变量作为模块名称（或带有子模块的命名空间）

¹ 当然，ECMAScript 6现在已经发布了，相应的特性支持业已具备。——译者注

```

    }else{
        return ((discountRate / 100) * amt).toFixed(2);
    }

};

return {
    applyDiscount : function(param) {
        return calculate(param);
    }
};

})();

```

返回带有公有函数的对象字面量，该对象字面量作为模块的公有 API

注意 对于 AMD/CommonJS 模块，不需要赋值给全局变量。

第 3 章还介绍了流行的改良版模块模式——揭示模式。清单 6.2 展示了该风格用法。

清单 6.2 揭示模式

```

var pricingSvcMod = (function() {

    var discountRate = 40;

    function calculate(amt) {
        if(isNaN(amt)){
            return 1;
        }else{
            return ((discountRate / 100) * amt).toFixed(2);
        }
    };

    return {
        applyDiscount : calculate
    };

})();

```

简化的返回对象简化了 API

在这个版本中，仅有返回的对象字面量代码与清单 6.1 版本不同。在这里，公有函数只是内部代码的指针。这使得 API 更清爽，也更易读。

这两个版本的模式设计使得模块能够作为功能的包装器使用（如图 6.2 所示）。

模块模式的外层函数给模块代码提供了保护屏障。这多亏了外层函数的受限作用域。

注意 作用域（广义上）指应用的某部分到相同应用另一部分的可达性。

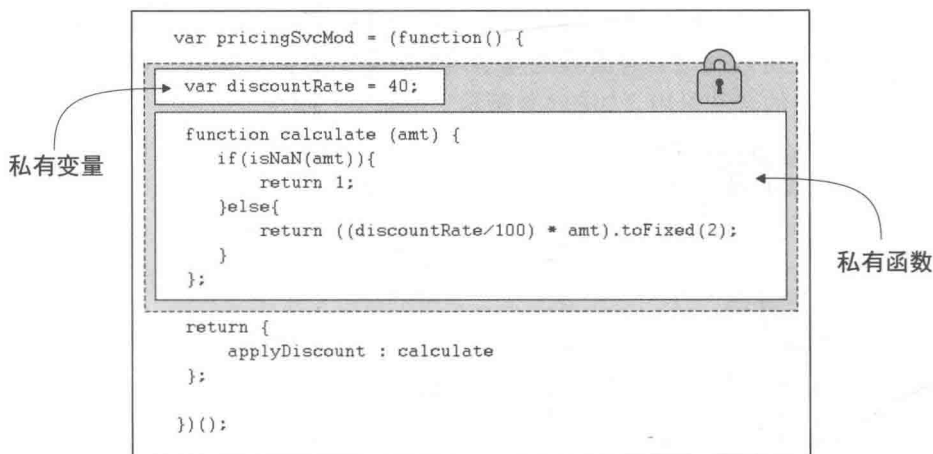


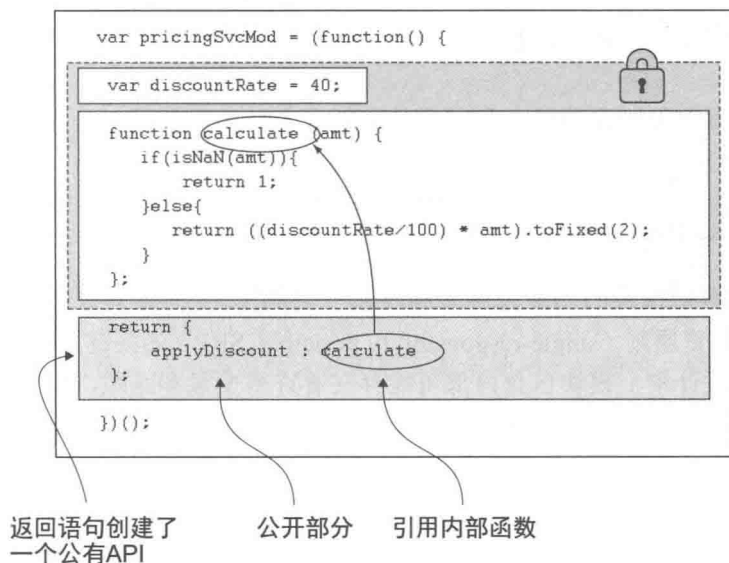
图 6.2 模块为内部代码提供了保护屏障。模块内部定义的变量与函数都是私有的

这种巧妙的函数设计同时还能够避免变量及函数污染全局命名空间，因为它们现在都是模块外层函数的局部定义。

6.1.2 API 提供对内部功能的访问控制

模块模式的另一个优良特性是创建应用编程接口（API）。API 就像模块间通信的契约。API 定义了公开可用的部分。其他模块可以通过 API 对该模块内部代码进行有限及受控的访问。

API 通过模块返回语句来组织（如图 6.3 所示）。这种方式在模块内部功能与外部世界之间搭建了一座桥梁。



在图 6.3 中，我们返回了一个通过对象字面量语法定义的对象。在这个返回对象中，在冒号左边的所有命名对象都是公开的。右边对象则引用模块内部代码。

图 6.3 API 中的每个公开函数（冒号左边）都有一个对模块内部私有对象的相关引用（冒号右边）

对象返回之后，其被分配给一个外部变量。通过该变量可以从外部对模块功能进行访问。其他模块可以发送信息给该变量引用的对象。只要该变量一直存在，其就能始终拥有对对象的合法引用（如图 6.4 所示）。

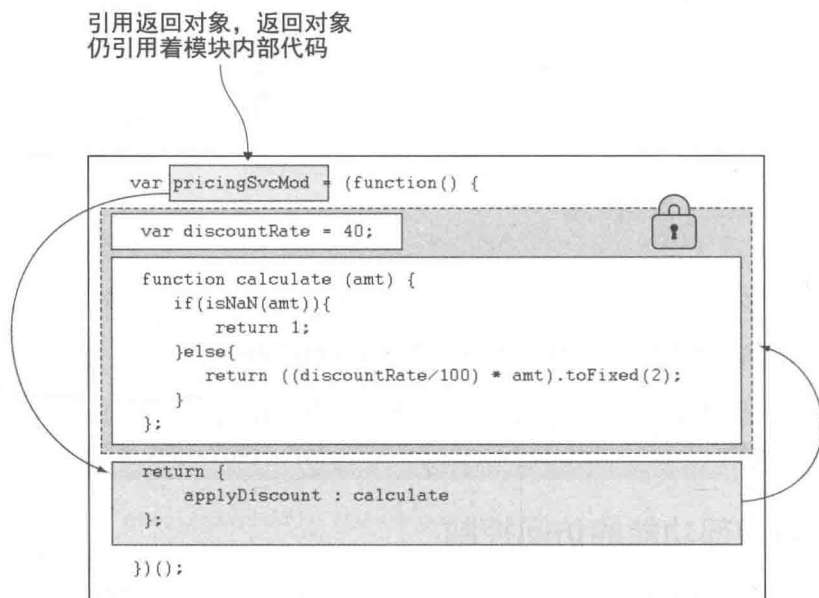


图 6.4 所分配的外部变量能够用来间接引用内部对象

为封装代码提供 API 支持使得我们不仅能够访问模块内部代码，还可以通过 API 定制交互行为。我们可以针对外层代码为 API 的公有函数设定有意义的名称；同时，针对内部代码为私有函数设置别的对应意义的名称。此外，还可以决定哪些内部逻辑应该隐藏或开放。

请记住我们并非出于保密而隐藏功能，我们只是限制了 API 中开放的部分刚好能满足其他模块成功调用它。

6.1.3 SRP——以单一目的作为设计出发点

设计模块时，我们尽量把模块的功能范围限制在单一目的上。“一个对象一个用途”是软件设计中单一职责原则（single-responsibility principle, SRP）的关键所在。可以将此思想扩展到模块设计中：模块自身内部可能存在着许多变量和函数，但所有这一切都应该服务于模块的单一整体职责（勿夹杂着出于其他目的的功能用途）。如果坚持用 SRP 设计模块，模块就好比机器上的齿轮。每个齿轮都有一个特定用途，但与其他模块间能够协调工作。

随着应用变得越来越复杂，模块复杂度通常也会随之增长。然而模块的好处是，如果其代码在某些方面开始失控（如代码规模变大、出现了其他功能用途），我们总可以将其重构和分割到一个或多个其他模块中。

举个例子，假定电视游戏应用开始作为一个单独的模块来设计，但其功能增长最终超出了最初设计。这时候最佳策略是重构这些代码，将其分割到单一职责的更小模块中去（如图 6.5 所示）。重构大型、多用途的模块，将有助于我们维系应用代码的 SRP 本色。

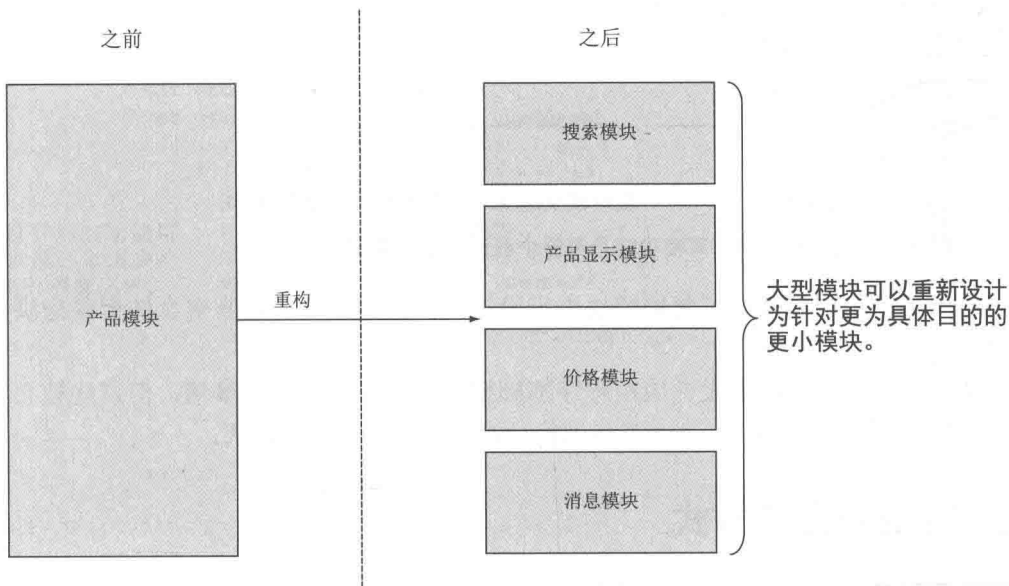


图 6.5 模块可以拥有多个函数，但理想情况下应该只有单一的整体目的

6.1.4 代码重用——控制项目规模

重构过程有时候会出现另一好处，有可能发现可在当前或是将来阶段重用的功能。可复用组件意味着项目规模不断扩大时却能够做到更少的开发量，因为共享模块消除了多处重复代码的需要。

在本章当前的 SPA 样例程序中，从搜索结果中选择了某个产品之后只显示具体游戏价格。然而假设哪天你的朋友要求新增一些特性，诸如添加购物车或产品列表，这些功能需要显示游戏的折扣价格（如图 6.6 所示）。

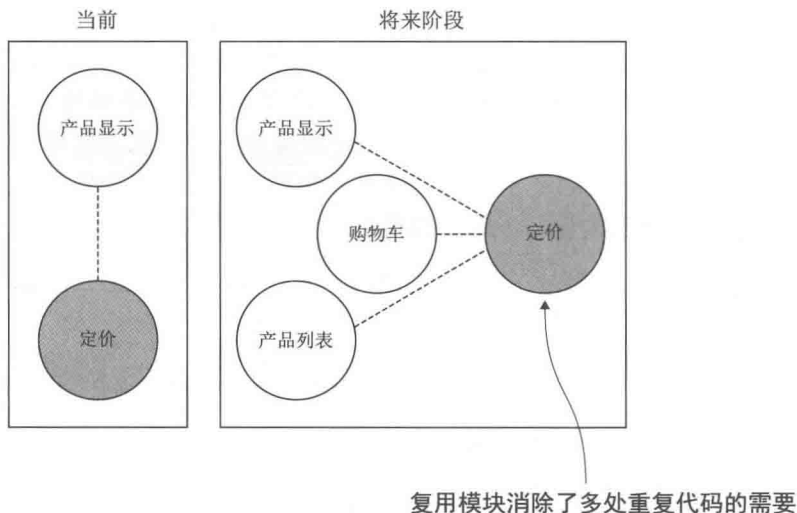


图 6.6 模块可以在其他模块、其他功能乃至整个应用程序中重用

由于我们的应用采用可复用模块进行设计，因此可以用比通常更少的调整来满足这些类型的新增需求。

能够用模块化风格来设计应用程序基础构造是一件非常棒的事情，但这些独立单元如何实现彼此交互呢？下一节我们将讨论几种基本的交互方式。

6.2 模块间交互方式

模块交互有两种方式：直接通过模块 API——这种方式创建了一个直接依赖；或者通过事件方式。我们详细讨论过模块 API 的细节，因此本节更多关注如何使用模块交互的依赖来影响应用架构。本节还将展示事件的解耦效果——更具体地说，使用一种被称为发布 / 订阅模式的事件聚合模式。

6.2.1 通过依赖进行模块间交互

尽管各种模块模式的语法风格看起来都比较独特，但模块始终只是个函数。因此，我们可以通过其参数传入些什么。将另一个模块作为参数传入，也是可采用的一种模块交互方式。这种交互是直接方式，因为一个模块直接访问了另一个模块的 API。当第一个模块通过直接调用第二个模块的 API 来与其（第二个模块）交互时，第二个模块被称为第一个模块的依赖。

每种模块风格都提供了一种将其他模块声明为依赖的方式。尽管语法迥异，但各种类型的依赖都针对一个共同的目的：允许模块访问另一个模块的 API，以便能

够与之交互。如果使用传统模块模式，举个例子，我们在模块尾部括号内声明依赖，并通过参数（依赖）列表来访问这些依赖。

为了方便起见，我们用之前的价格模块来阐述。为了获取每个所选游戏的折扣价格，需要将价格模块作为依赖添加到产品显示模块（如清单 6.3 所示）。同时，也需要将产品数据模块作为依赖添加到产品显示模块，以访问产品数据。

清单 6.3 传统模块模式的依赖

```
var productDisplaySvcMod = (function(productData, pricingSvc) {
    function getDetailsById(id) {
        var gameFound = null;
        var gameList = productData.usedGames;
        for ( var i = 0; i < gameList.length; i++) {
            if (gameList[i].productId === id) {
                gameFound = {
                    name : gameList[i].name,
                    productId : gameList[i].productId,
                    summary : gameList[i].summary,
                    url : gameList[i].url,
                    price : pricingSvc.applyDiscount(gameList[i].price)
                };
            }
        }

        return gameFound;
    };

    return {
        getDetails : getDetailsById
    };
})(productDataMod, pricingSvcMod);
```

合并后的功能可以通过本模块的 API 暴露出来

通过参数列表访问

模块现在能够访问依赖的 API

在尾部括号中声明依赖

模块声明为另一个模块的依赖之后，就可以访问其（依赖）API 了。依赖的 API 确保让你如愿访问其功能，并确保传入满足调用的任何所需信息。记住，我们并未直接访问依赖模块的内部函数或对象。

注意 尽管将某个模块添加为依赖是模块交互的直接方式，但我们仍是通过依赖模块的 API 来与其交互。

通过依赖进行交互对许多场景来说是个非常不错的选择，但它不是万能的。该方式有优点也有缺点。

6.2.2 依赖方式的优缺点

这里列出了一些模块间直接交互方式的优缺点。但别将这些内容作为是否选择这种方式的依据。在模块化编程中避免使用依赖是不现实的。把这里的内容作为何时使用依赖方式的帮助指南就好。

优点：

- 不调用中间对象；模块能够直接调用另一个模块的 API。
- 直接交互对于调试来说有时显得更容易。
- 通过模块的依赖列表，很容易查看源代码并找出为了特定功能或特性而将哪些模块组合在一起。

缺点：

- 随着依赖模块的引入，也引入了一定程度的耦合。耦合指的是代码一部分与另一部分直接关联。当模块发生了耦合时，就降低了代码修改的灵活度。
- 依赖列表有可能变得老长，在跟踪依赖关系时，有时候这会带来一些麻烦。
- 当模块与其依赖交互时，个中是一对一关系。此类型的交互涉及面较狭窄，只有一个接收对象，与下一节所描述的方式截然不同，下一节的方式可以有多个接收对象。

模块交互的其他方式是通过事件。下一节会重点介绍一种流行的事件聚合模式——发布 / 订阅模式（publish/subscribe 或 pub/sub）。其中我们将了解发布 / 订阅模式的概念、总体应用方法及其在 SPA 应用中的一些优缺点。

6.2.3 通过发布 / 订阅模式进行模块间交互

在我们讨论 MV* 框架时，无论是 DOM 交互还是对象间交互，事件都在现代应用中得以广泛应用。可以把整个浏览器环境看成是事件驱动的。事件提供了一种实现松耦合的自然方式，因为接收方可以决定是否监听事件，也可以决定响应方式。

这些年出现了一些与事件有关的设计模式。本节关注发布 / 订阅模式（publish/subscribe 或 pub/sub）。发布 / 订阅模式是一种用于不同模块间的常见模式。发布 / 订阅模式基于经典的观察者模式（observer pattern）。

在观察者模式中，某个对象被直接观察（Observable——可观察对象），其他多个对象（Observer——观察者）可以选择观察它，如图 6.7 所示。只要 Observable 状态发生了改变，其就发送一个通知（通常通过事件），以便 Observer 能够做出相应的响应。

与经典的观察者模式的区别在于，发布 / 订阅模式通常由一个中间服务代表另一个对象发布（发送或广播）通知。其他对象可以决定监听与否。

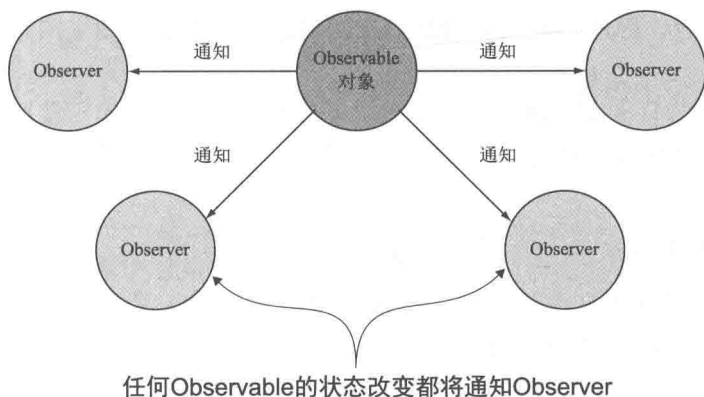


图 6.7 在观察者模式中，只要 Observable 发生状态改变，每个 Observer 都会收到通知

当两个没有联系的模块需要交互，或者某个应用范围的消息需要广播出去而发布者又不关心接收者收到消息后的行为时，这种类型的中介代理是间接实现模块间交互的理想方式。

1. 主题 (Topic)

尽管不是必需的，但绝大多数发布 / 订阅模式实现的通知都是基于主题 (Topic) 的。主题 (或 AngularJS 中的事件名称) 是一个简单的名称，其用于表示一个特定通知。如果另一个对象想要监听它，这个对象就订阅该主题。当一条主题消息发布了，该消息的中介代理就分发通知给所有的主题订阅者。

在我们的应用案例里，你已经创建了一个模块，其唯一目的就是使用 AngularJS 内建发布 / 订阅机制来广播系统级消息。该模块使用发布 / 订阅模式来发布一条带有 userMessage 主题的消息。

如图 6.8 所示，该主题只有一个订阅者：一个控制器，其用于显示用户提醒视图。由于该控制器是

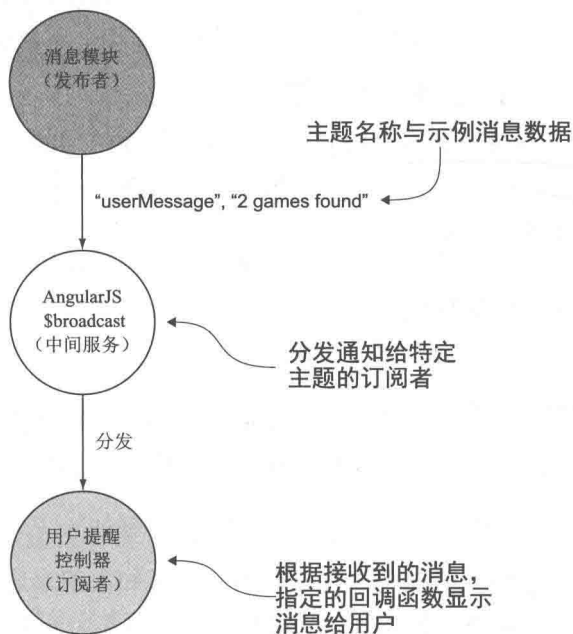


图 6.8 消息模块使用发布 / 订阅模式，通过中间服务 (中介代理) 来发布消息给所有的订阅者

userMessage 主题的订阅者，因此其在接收到新消息的情况下将更新视图里的文本信息。

通常情况下，主题通知与订阅都通过发布 / 订阅模式具体实现所提供的语法风格来编程实现。如果决定在应用程序中使用发布 / 订阅方式，就得考虑一些发布 / 订阅模式的实现工具。

2. 发布 / 订阅模式的实现库

对于 SPA 应用程序，消息代理功能实现要么内建在 MV* 框架中，要么得从众多的发布 / 订阅模式的 JavaScript 实现库中挑一个。表 6.1 列出了本书写作时可用的几个发布 / 订阅模式实现库。

表 6.1 几个可用的发布 / 订阅模式实现库

发布 / 订阅模式实现库	URL
AmplifyJS	http://amplifyjs.com
PubSubJS	https://github.com/mroderick/PubSubJS
Radio.js	http://radio.uxder.com
Arbiter.js	http://arbiterjs.com

对照应用程序依赖处理的做法，使用前面提及的通用衡量标准来审视所有可选方案：学习曲线、bug 及修复率，文档、成熟度与社区支持等。

3. 基本通知类型

最基本的通知类型不包含任何需传递的数据。其只不过是所发布消息的主题名称。为了阐述发布 / 订阅模式中的基本通知类型，我们将采用工具无关的伪代码来讨论。

首先讨论如何发布主题消息。为了在模块 A 中发布一条消息，我们需包含类似下面的这行代码：

```
pSub.publish("hello_world_topic");
```

这很简单。接下来在模块 B 中订阅该主题也一样简单。我们包含需要监听的消息的主题名称，以及监听到消息时需产生的动作：

```
pSub.subscribe("hello_world_topic", functionToCallWhenHeard);
```

通常情况下，我们使用基本通知类型来通知所有订阅者发生了某事。之后，每个订阅者根据监听到的消息，能够以完全不同的方式做出反应。

然而我们经常需要在发布主题中包含数据。这同样能够通过发布 / 订阅模式来轻松实现。

4. 带有数据的通知类型

除了基本通知类型，大多数发布 / 订阅模式的中介代理能够在发布消息的同时捎带上数据。紧接着，该主题的每个订阅者借助中介代理获取该数据并将数据传递给回调函数。要发布带有数据的消息，我们在模块 A 中使用类似以下这行代码：

```
pSub.publish("hello_world_topic", dataObjectToSend);
```

在模块 B 中，订阅的代码行跟基本通知类型里的是一样的。消息代理传递数据给订阅中的回调函数：

```
pSub.subscribe("hello_world_topic", functionToCallWhenHeard);
```

接收数据时的唯一不同点在于回调函数本身。在这里，我们需要函数签名带有一个参数，用来表示通过订阅传入的数据：

```
function functionToCallWhenHeard( paramForDataPassed ) { ... }
```

对于大多数消息代理而言，任何合法的 JavaScript 对象或值都可以跟通知一起传递。

5. 退订

对于绝大多数发布 / 订阅模式实现库来说，另一个常用功能是退订功能。由于订阅是基于主题的，因此不再需要对主题做出反应时，订阅者能够调用消息代理的退订函数。再次地，绝大多数发布 / 订阅模式实现库处理起这件事情来超级简单：

```
pSub.unsubscribe("hello_world_topic");
```

也许还有一些功能可供使用，诸如设置主题优先级，但这属进阶内容且跟各种发布 / 订阅模式实现库的具体实现相关。此外，到目前为止提到的功能是最低限度的，但并不能保证你所用的发布 / 订阅模式实现库对其提供了支持。消息代理的文档会列出可用特性。

6.2.4 发布 / 订阅模式优缺点

如前所述，发布 / 订阅模式用于解耦代码模块。它是一个强大而灵活的工具，但并非没有缺点。以下给出了 SPA 应用中发布 / 订阅模式的主要优缺点。

优点：

- 该模式并不管理直接依赖，而是通过将通知发给消息代理提升了模块的松耦合程度。
- 与使用 API 一样，发布 / 订阅模式容易实现。
- 通知能够立即广播给多个订阅者。
- 应用不同部分可以对是否观察所发布消息做出选择。

缺点：

- 如果该模式未内建到 MV* 框架中，消息代理实现库本身就是一个额外的依赖，其必须单独管理。
- 通知流只有一个方向。没有确认或响应回送给发布者（尽管可以创建响应主题来实现双向作用类型）。
- 主题是简单的文本串。必须依赖命名约定来确保主题传递到正确的接收者。
- 调试时很难跟踪贯穿系统的消息流。
- 在通知发布之前，或者尚未对主题进行监听时，我们的代码必须确保订阅者的有效性，而且它要能够监听发布者。

现在我们已经理解了模块概念及模块交互方式，接下来我们要讨论本章示例项目的重点内容。为了保持跟本书其他章节的一致性，在这里仍使用 AngularJS。

6.3 示例项目细节

如本章开始时所述，你的朋友开了一家销售二手电视游戏的小公司，我们的项目将为他创建一个简单的在线商店。在这里，我们只创建这个 SPA 应用的产品搜索部分，因为这已足够帮助我们理解本章讨论的模块间交互方式。

在之前的几章中，我们比较关注示例应用的一两个功能点，以聚焦于当时的概念。但现在我们的应用代码会更复杂一些了。在这里，应用功能划分为以下几点：

- 搜索
- 产品显示
- 定价
- 消息
- 用户提醒

当我们理解了一些重点代码后，就会对每个模块使用的模块间交互方法有所理解，进而掌握应用的通信方式。然而在讲述代码之前，我们得先来看一下这个项目的目标。

首先来观察一下图 6.9，其展示了应用完成时的总体效果。有了第一印象之后，这里列出了示例应用应该具备的功能：

- 客户可以通过部分或完整的游戏标题进行搜索。
- 搜索成功后展示一个结果列表，其中包含了游戏缩略图及游戏标题。
- 除了搜索结果，每次搜索的结果数量信息也会显示在应用底部位置。
- 当选择某个游戏时，将把用户带到产品视图，产品视图展示每个具体游戏的细节信息。
- 二手游戏的折扣价格基于当前零售价格再统一进行 40% 的折扣（简单起见）。

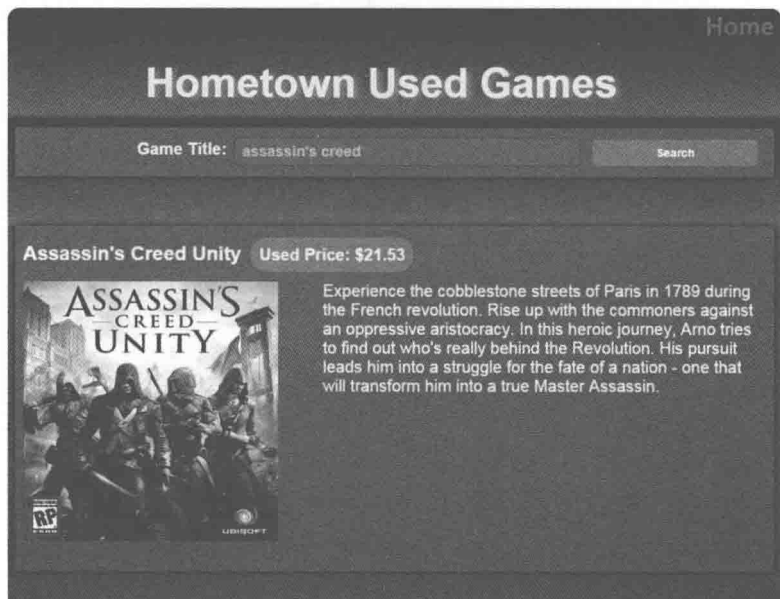


图 6.9 示例项目是一个销售二手电视游戏的在线商店

由于之前章节已经介绍了路由及视图的知识，我们只在模块设置时会涉及它们。就像其他示例那样，完整源代码可以在线下载。

本章示例项目使用 AngularJS，因此还需要简要了解 AngularJS 中的模块及依赖。尽管我们的讨论将尽可能做到工具无关，但仍需要掌握一点必要的 AngularJS 知识以跟上代码的脚步。

AngularJS 模块及依赖快速介绍

在 AngularJS 应用程序中，可以通过调用框架的 `module()` 函数来创建模块，提供给该函数一个模块名称，以及一个可选的依赖列表：

```
angular.module("moduleName", ["dependency1", "dependency2"])
```

现在比较一下它和传统模块模式的差异：

```
var moduleName = (function(depParam1, depParam2) {  
  })(dependency1, dependency2)
```

如果我们没有任何依赖，则提供空列表即可：

```
angular.module("moduleName", [])
```

创建好模块之后，就可以根据代码功能在模块中创建具体的 AngularJS 组件。AngularJS 提供了以下开箱即用的组件：过滤器、指令、控制器、value、constant、服务、工厂和提供者等。这些 AngularJS 组件的细节内容已超越本书范畴，但你可以通过在线文档了解它们：<https://angularjs.org>。

除了 AngularJS 内建指令, 我们还使用控制器、value 及工厂。value 组件用来保存存根数据, 因为该组件是存储应用级数据的理想工具。工厂组件最像传统模块模式 (从用途角度来看), 因此我们将主要使用它来构建基本功能。控制器组件如前所述, 扮演应用代码和 UI 之间的桥梁角色。

为了创建模块的组件, 需要在模块声明中添加一个组件函数, 诸如 `factory()` 或 `controller()`。也可以在声明中包括其他组件的名称, AngularJS 将把它们注入到我们创建的组件中去。这种方式被称作依赖注入 (dependency injection, DI)。

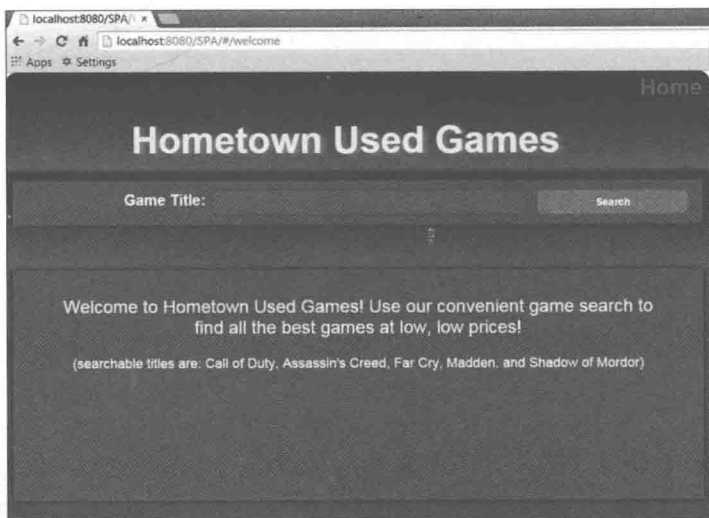
为了通知 AngularJS 注入另一个组件到我们创建的组件, 需要添加另一个组件到函数参数列表中。所注入组件的名称复制为文本串, 以将所注入依赖的具体实现名称从代码绑定的指定引用中解耦出来:

```
angular.module("moduleName", [])  
  .factory("componentName",  
    ["otherComponent", function(otherComponent) {...}]  
  )
```

所注入组件可以是在其他模块中创建的任意组件 (如果当前模块依赖列表中包含了其他模块)、AngularJS 自身提供的任意开箱即用的组件, 或其他第三方组件。

现在, 我们最低限度理解了 AngularJS 的模块、组件以及依赖, 接下来编写本章 SPA 样例程序代码。

6.3.1 搜索功能



当应用加载时, 用户将看到欢迎消息, 同时我们还提供了游戏搜索功能。标头视图和搜索视图是固定的, 因此它们在主内容伴着每次搜索结果而改变时将保持不变。搜索标题时, 单击搜索按钮, 以调用路由显示搜索结果 (如图 6.10 所示)。

图 6.10 用户一进入网站就会看到欢迎信息, 并能够立即搜索游戏

图 6.11 大致描述了整个搜索流程，请关注相关处模块间交互的类型。

1. 用户搜索使用“of”一词（其作为参数传递给搜索控制器）

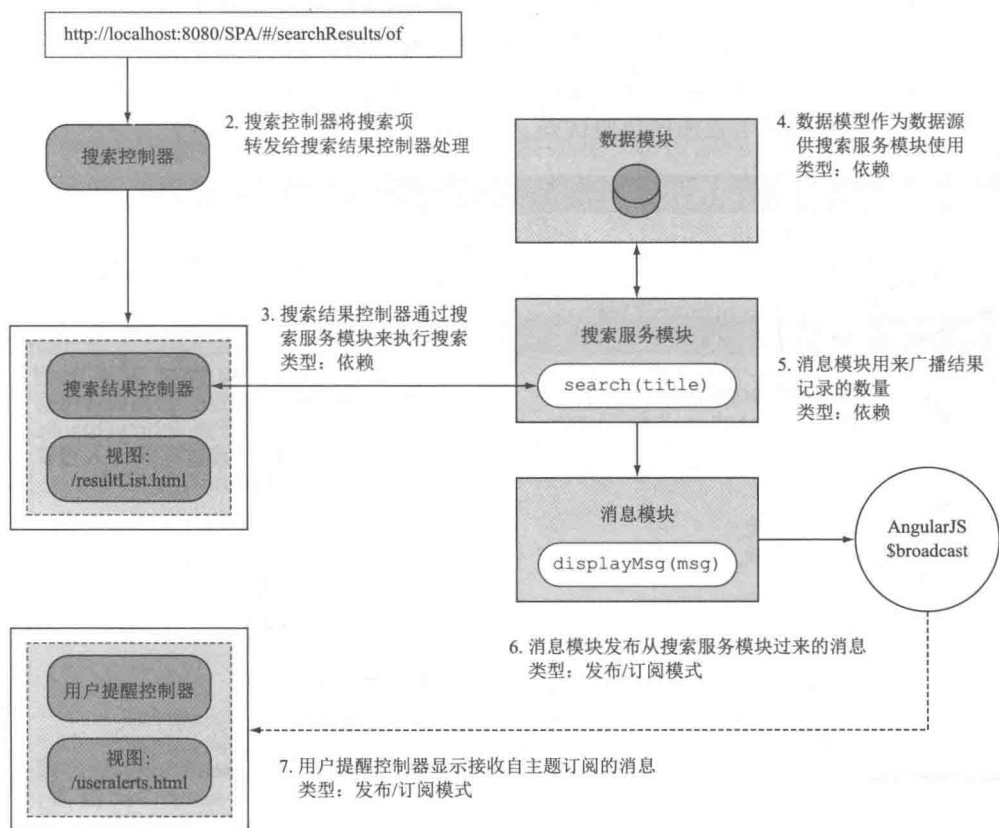


图 6.11 用户搜索游戏标题流程概览

所有的搜索都是关键字搜索，因此所有标题中包含搜索项的游戏都会添加到结果列表中并得以显示。

以上是用户进行搜索时的整体流程说明。此部分说明就告一段落了，接下来介绍各种模块交互类型之下的代码。

1. 搜索控制器模块

搜索控制器 (`search.controllers`) 模块包含两个控制器组件：一个用于处理搜索本身，另一个用于显示搜索结果。它还将搜索服务 (`search.services`)

模块间交互类型：依赖方式

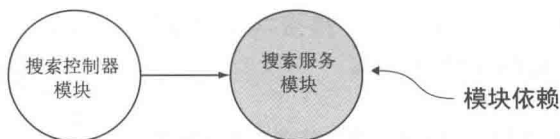


图 6.12 搜索控制器模块使用搜索服务模块来执行搜索。搜索服务模块是搜索控制器模块唯一的依赖

模块作为其唯一的依赖（如图 6.12 所示）。

搜索执行后，搜索结果控制器使用 `search.services` 模块中的一个组件来搜索用户输入的搜索项。如果找到结果，`search.services` 模块中的搜索组件就返回游戏对象列表给结果控制器，结果控制器将结果传递给它的视图以便显示结果。

清单 6.4 展示了 `search.controllers` 模块的代码。请记住在 AngularJS 中，应在模块声明的方括号中添加模块级依赖。

清单 6.4 搜索控制器模块

```
angular.module("search.controllers", ["search.services"])
    .controller("searchController",
        ["$scope", "$state",
            function($scope, $state){
                $scope.search = function() {
                    $state.go("searchResults",
                        { gameTitle:$scope.game.name } );
                };
            }
        ])
    .controller("searchResultsController",
        ["$scope", "$stateParams", "searchSvc",
            function($scope, $stateParams, searchSvc){
                $scope.results =
                    searchSvc.search($stateParams.gameTitle);
            }
        ]);
```

创建模块，将 `search.services` 模块作为依赖

为搜索创建控制器组件

在 UI 中，搜索按钮绑定到 `$scope.search`。该函数会将当前状态过渡到 `searchResults` 状态，此过渡过程需传入搜索项参数（`gameTitle`）

`searchResultsController` 控制器用于 `searchResults` 状态

注入 `search.services` 模块的 `searchSvc` 组件，将其用于数据搜索

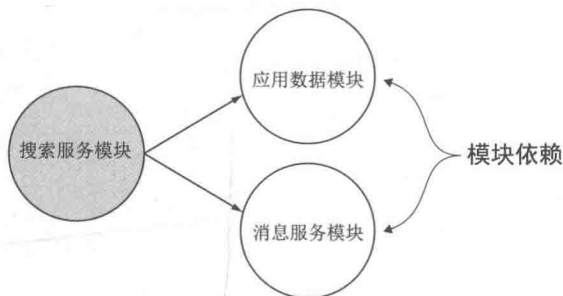
我们现在了解了搜索相关的控制器，接下来看看作为依赖添加的 `search.services` 模块的代码。该模块至关重要。

2. 搜索服务模块

探索服务 (`search.services`) 模块有两个依赖：一个用来访问数据，另一个用来广播搜索结果数量（如图 6.13 所示）。

图 6.13 搜索服务模块将应用数据模块作为数据源，并通过消息服务模块来广播搜索结果数量

模块间交互类型：依赖方式



我们将搜索功能拆开来分析以方便理解。先来看看依赖列表：

```
angular.module("search.services", ["data.appData",
    "messaging.services"])
```

我们注意到 `search.services` 模块也有其依赖：

- `data.appData`——该模块包含游戏清单。
- `messaging.services`——该模块创建一条搜索结果数量的消息，并通过发布 / 订阅模式广播该消息。

清单 6.5 提供了搜索服务模块的完整代码。代码比较长，但大部分代码都比较普通，通过任意的游戏标题来匹配搜索项。

清单 6.5 搜索服务模块

```
angular.module("search.services",
    ["data.appData", "messaging.services"])

.factory("searchSvc", [ "productData", "messageSvc",
    function(productData, messageSvc) {

function searchByTitle(title) {

    if(!(title && (typeof title.trim == "function"))){
        return [];
    }

    if(!(productData && productData.usedGames
        && (typeof productData.usedGames.filter == "function"))){
        return [];
    }

    var loweredTitle = title.trim().toLowerCase();

    var gamesFound =
        productData.usedGames.filter(function(game) {
            return game.name.toLowerCase().indexOf(loweredTitle) > -1;
        });

    // 搜索结果数量消息
    messageSvc.displayMsg(
        createResultsMsg(gamesFound.length)
    );

    return gamesFound;
}

function createResultsMsg(resultSize) {
```

模块 API 引
用的函数

创建模块，有两个依赖

创建工厂组件，并注入
两个组件

访问从 `app.appData`
模块获取的数据

访问 `messaging.services`
模块中 `messageSvc` 组件
的 API

```
var quantifier = resultSize > 0 ? resultSize : "No";
var noun = resultSize === 1 ? "game" : "games";
var terminator = resultSize > 0 ? "!" : ",";

return quantifier + " " + noun
+ " found" + terminator;
}

return {
  search : searchByTitle
};
} 1);
```

← 创建一个公有 API

在这段代码中，我们通过 AngularJS 注入 `data.appData` 模块中的 `productData`，以及 `messaging.services` 模块中的 `messageSvc`。`productData` 和 `messageSvc` 是各自模块中的主要功能。

对搜索项的每个匹配结果，其信息都返回给调用者（搜索结果控制器）。之后将结果显示给用户。图 6.14 展示了搜索成功后的搜索结果视图。在这里，用户在搜索中使用“of”一词。这将匹配“Call of Duty Advanced”和“Middle Earth: Shadow of Mordor”的游戏标题。

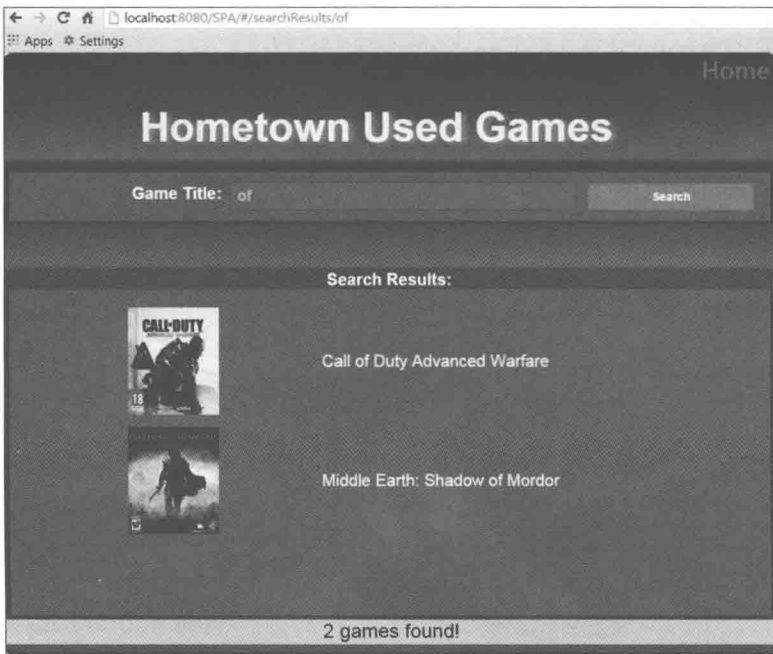


图 6.14 显示搜索结果，以及一条搜索结果数量的简短提醒。每个搜索结果都是一条链接，用于显示结果项的具体信息

另外，搜索结果记录数呈现在屏幕底端。在 `searchByTitle` 函数返回之前，我们通过 `messaging.services` 模块的 `messageSvc` 组件来创建搜索结果数量消息，并将消息广播给应用程序的其他部分，如图 6.14 所显示的黄色消息。消息模块通过发布 / 订阅模式来广播消息。

3. 消息模块

消息模块没有模块级的依赖，因此不会与其他模块发生直接交互。然而它还是要通过发布 / 订阅模式跟用户提醒模块进行间接交互（如图 6.15 所示）。

模块间交互类型：发布/订阅模式

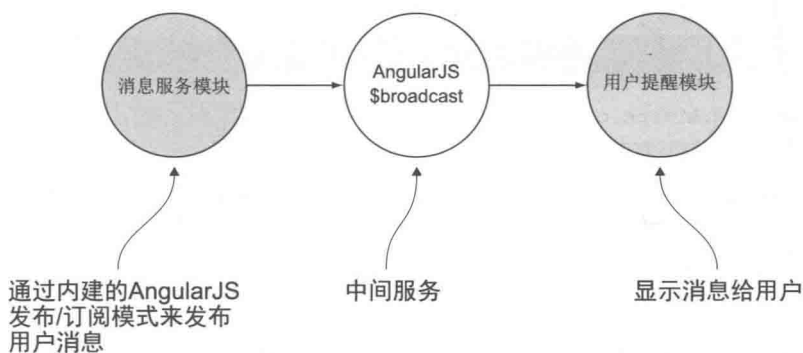


图 6.15 消息服务是通用工具，只负责广播其获取到的消息。在这里，唯一的监听模块是用户提醒模块

消息模块的代码很简单。它有一个 `messageSvc` 组件，通过 AngularJS 内建的发布 / 订阅模式实现来广播传递给它的所有消息，如程序清单 6.6 所示。

清单 6.6 消息服务模块

```

angular.module("messaging.services", [])

    .factory("messageSvc", [ "$rootScope", function($rootScope) {
        function displayMsg(msg) {
            $rootScope.$broadcast("userMessage", msg)
        };
        return {
            displayMsg : displayMsg
        };
    } ] );
  
```

创建工厂组件，通知 AngularJS 注入 \$rootScope（最顶层的作用域对象）

创建模块并命名，没有依赖

通过 \$broadcast 来发布 userMessage 主题，以及所有消息文本

在提供的 API 中通过 displayMsg 函数来暴露功能

像消息广播这样一个通用的系统级功能，如果没有更好的方式，那么使用发布

/ 订阅模式是合适的。同时, 以这种标准模式来构建模块使得我们能够将消息模块作为通用消息处理系统, 在需要广播消息的任意地方复用。

随着搜索结果的显示及结果数量的广播, 我们剩下的模块将接收消息并将消息显示为用户提醒。

4. 用户提醒模块

用户提醒 (user.alerts) 模块同样没有模块级依赖。但如上一小节内容所述, 其与消息服务模块间接交互 (如图 6.15 所示)。

在这个模块中, 我们有一个控制器组件, 以便我们能够在用户提醒视图中显示通过订阅获取的任意信息 (如清单 6.7 所示)。

清单 6.7 用户提醒模块

```
angular.module("userAlerts.controllers", [])  
  .controller("userAlertsController",  
    ["$scope", "$rootScope", "$timeout",  
      function($scope, $rootScope, $timeout){  
        $rootScope.$on("userMessage", function(e, msg){  
          $scope.msg = msg;  
          $timeout(function() {  
            $scope.msg = null;  
          }, 2000);  
        });  
      }  
    ]  
  );
```

创建模块并命名, 没有依赖

通过 \$on 订阅 userMessage 事件

通知 AngularJS 注入几个需要用到的开箱即用组件

显示消息 (通过 \$scope/viewmodel), 并在 2 秒后移除消息

至此, 我们已经了解了搜索功能中模块间直接与间接交互的方式。接下来讨论用户从搜索结果列表中选择一条记录时的处理。

6.3.2 显示产品信息

当用户单击某条搜索结果时, 状态的改变允许应用程序显示所选游戏的详细信息。图 6.16 展示了整个处理过程。

从搜索结果列表选择某个游戏之后, 游戏 ID 作为一个参数发送给产品显示控制器。

1. 用户从搜索结果视图中选择 “Middle Earth: Shadow of Mordor”

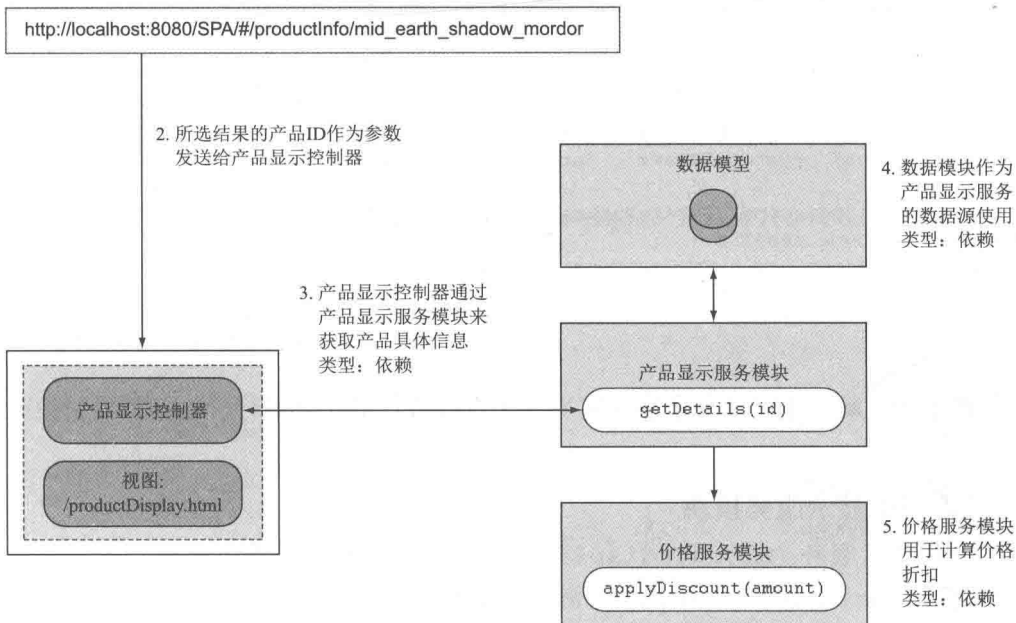


图 6.16 产品展示流程概览

1. 产品展示控制器模块

产品展示控制器 (productdisplay.controllers) 模块只有一个模块级依赖：产品展示服务模块 (如图 6.17 所示)。

模块间交互类型：依赖

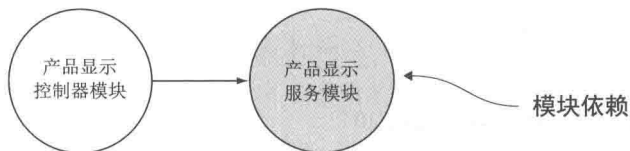


图 6.17 产品展示服务模块通过所选游戏 ID 找出正确的产品信息

如清单 6.8 所示，我们通过 productdisplay.services 模块的 productDisplaySvc 组件，使用参数中的产品 ID 来查找所选游戏的具体信息。之后把游戏具体信息分配给 \$scope (视图模型)，用于显示该信息。

清单 6.8 产品显示控制器

```
angular.module("productdisplay.controllers",
  ["productdisplay.services"])

.controller("productDisplayController",
  ["$scope", "$stateParams", "productDisplaySvc",
  function($scope, $stateParams, productDisplaySvc){
    $scope.results =
      productDisplaySvc.getDetails($stateParams.productId);
  }
]);
```

创建模块及其依赖

通知 AngularJS 注入来自别的模块的 productDisplaySvc 组件

使用 productDisplaySvc 组件来获取所选游戏的具体信息

接下来了解一下产品显示服务模块为了查找游戏信息及进行价格打折需要做些什么。

2. 产品显示服务模块

产品显示服务 (productdisplay.services) 模块有两个依赖：一个用来访问数据，另一个用来计算所选二手游戏的 40% 价格折扣（如图 6.18 所示）。

模块间交互类型：依赖

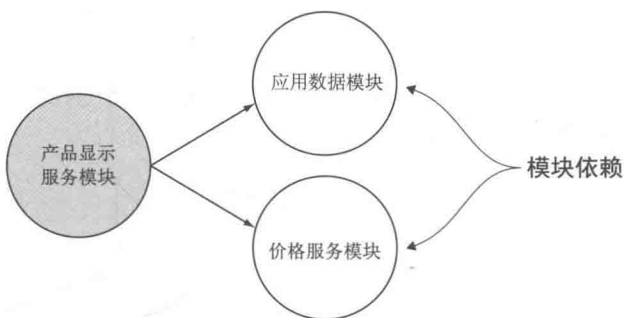


图 6.18 使用数据以及价格模块组件来计算所选游戏的价格折扣

该模块中有一些典型的 JavaScript 代码，用来遍历游戏列表并找出 ID 匹配内容（如清单 6.9 所示）。

清单 6.9 产品显示服务模块

```
angular.module("productdisplay.services",
  ["data.appData", "pricing.services"])

.factory("productDisplaySvc", [ "productData", "pricingSvc",
  function(productData, pricingSvc) {
    function getDetailsById(id) {
```

创建新模块，有两个依赖

通知 AngularJS 注入来自依赖模块的数据及价格服务组件

访问游戏清单

```

    if (!(productData && productData.usedGames)){
        return null;
    }

    if (!(typeof productData.usedGames.filter == "function")) {
        return null;
    }

    var gamesFound = productData.usedGames.filter(function(game) {
        return game.productId == id;
    });

    if (!gamesFound.length) {
        return null;
    }

    var gameFound = gamesFound[0];

    return {
        name : gameFound.name,
        productId : gameFound.productId,
        summary : gameFound.summary,
        url : gameFound.url,
        price : pricingSvc.applyDiscount (gameFound.price)
    };
}

return {
    getDetails : getDetailsById
};
});

```

调用价格组件 API 的
applyDiscount 函数

创建 API

了解了产品显示服务模块，我们就到达了该示例项目的最后一站。接下来看看折扣计算，并将折扣结果返回给调用模块。

3. 价格服务模块

价格服务 (pricing.services) 模块没有依赖，也很简单。它从外部模块获取金额并乘以折扣率，如清单 6.10 所示。

清单 6.10 价格服务模块

```

angular.module("pricing.services", [])

.factory("pricingSvc", function() {

    var discountRate = 40;
    var discount = discountRate / 100;

```

创建模块，不带依赖

```
function isNumber(n) {  
    return !isNaN(parseFloat(n)) && isFinite(n);  
}  
  
function calculate(amt) {  
    if(!isNumber(amt)){  
        return 1;  
    } else{  
        return (discount * amt).toFixed(2);  
    }  
}  
  
return {  
    applyDiscount : calculate  
};  
});
```

返回计算后的折扣价格

通过 API 公开折扣计算函数

通过 ID 找到正确的游戏, 且打完折之后, 一个新的游戏对象就会返回给控制器, 然后信息传递给视图并显示出来。图 6.19 展示了选择产品之后的效果。

结果视图是该项目中模块间交互设计的重点, 它同时包含了依赖方式与发布 / 订阅方式。

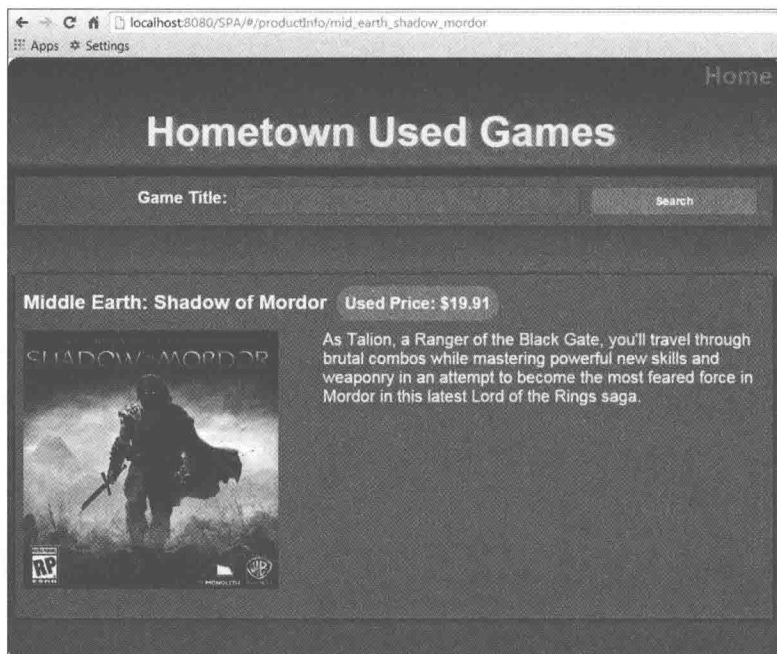


图 6.19 找到匹配游戏并打完折之后的结果视图

6.4 挑战环节

又到了挑战环节，现在来检验一下本章的学习情况。创建一个电影标题搜索功能，当用户在一个输入文本字段中输入电影标题时，将匹配搜索标题的电影展示到一个列表中。使用一个视图，顶端是输入文本字段，输入文本字段下面是记录结果的无序列表。通过综合使用依赖方式与发布 / 订阅方式来创建该功能。使用你喜欢的 MV* 框架，绑定一个 Key-up 事件到发布输入字段内容的函数上（通过每次按键触发发布动作）。创建一个模块，用来监听主题消息并执行搜索。该模块还应该使用发布 / 订阅模式来发布结果。应用根据收听到的结果，借助 MV* 框架填充无序结果列表。

6.5 小结

我们在本章中学到了模块间交互的大量内容：

- 模块是应用程序基础结构中至关重要的组成部分，其提供了封装能力，并且是代码重用的有效手段。
- 一个良好设计的模块可以由许多函数组成，但应该只有单一的整体目标。
- 尽管模块的内部代码是隐藏的，但其 API 为内部功能提供了一个集中而可控制的访问点。
- 模块间交互存在两种主要方式：通过 API 直接交互或通过事件间接交互。本章采用一种被称为发布 / 订阅模式的事件聚合模式。
- 不存在选择模块间交互类型的固定法则。一般而言，功能相关的模块间可考虑依赖方式。通常使用到的模块，诸如工具模块，也可以作为一个直接依赖。发布 / 订阅模式最适合用于功能不相关模块以及应用级通知。
- 依赖方式下的交互涉及面较狭窄，但该方式允许直接访问另一个模块的 API。
- 发布 / 订阅模式下的交互涉及面很广，允许主题的所有订阅者同时获取通知。

7 与服务器端通信

本章内容

- SPA 应用环境中的服务器端角色
- MV* 框架与服务器端的通信方式
- 通过回调函数和 Promise 处理结果
- 调用 RESTful 服务

在第 1 章中，我们了解了 XMLHttpRequest (XHR) 及 AJAX 技术的普及最终推动了 SPA 应用的出现。自从浏览器支持 XHR 以来——由最初的 COM 组件形式到之后的原生支持——开发者可以利用它来异步加载应用的脚手架内容及其数据，且无须刷新页面。从此，构建 Web 页面的新方式如雨后春笋般不断出现。

在之前的章节我们都是关注创建 SPA 应用程序本身。在这些场景中，我们通过 XHR 来动态获取用于构建视图的模板，但却仅限于本地存储的数据。在本章，我们将向前迈出重要一步，把数据源从本地移到服务器端，并学习如何从 SPA 应用程序中远程访问这些数据。

我们首先会跳过细节来简要了解一下 SPA 客户端与服务器端的通信过程。在掌握全貌之后，将继续介绍客户端的具体实现。

在客户端，我们聚焦于 MV* 框架如何简化与服务器端的通信。MV* 框架通过功能扩展集为持续增强 XMLHttpRequest API 提供了内建支持。但由于各种框架都利用 XHR，因此可以提炼出它们的一些共性处理方式。

SPA 应用客户端与服务器端的最佳通信方式？

通常情况下，SPA 应用客户端与服务器端的最佳通信方式是使用 MV* 框架提供的对象——如果框架支持服务器端通信的话。因为这些对象是作为框架内部运作结构的一部分而专门创建的，它们提供了现成的请求及响应方法来与框架的其余部分协同工作。针对自己的具体需求，我们需要通过配置或采取某种扩展方式来定制这些对象。基本上我们不需要在所选框架中采用额外的库。

万一框架没有对服务器端通信功能提供内建支持，我们的选择也不少：通过 XMLHttpRequest 对象本身提供的低级方法来直接实现与服务器端的交互、使用通用工具库（如 jQuery），或者考虑一个组件少但功能又更专业的库（如 AmplifyJS，<http://amplifyjs.com>）。

掌握了与服务器端通信的基础知识之后，我们把目光移到 XHR 结果处理上。先引入传统回调函数的实现方式，用该函数来说明调用成功或失败时的处理。紧接着会介绍 Promise 用法。Promise 正快速成为大量现代 MV* 框架处理 XHR 结果的首选方式。然而更重要的是，Promise 是 ECMAScript 6 实现的一部分。其通常被认为是一种比简单的回调函数更简洁优雅的异步处理方式。

本章将简要介绍 SPA 调用 RESTful 服务的方式。REST 是针对 Web 站点与 Web 服务的一种架构风格，这些年来日趋流行——乃至许多的 MV* 框架给予其内建支持，有些甚至将 RESTful 风格作为缺省实现。

我们不会涉及 RESTful 服务设计的细节，因为服务器端话题超出本书范围。我们将从设计意义的角度来讨论 REST 是什么，并讨论一些 MV* 框架处理 REST 的方式。

在本章的示例中，我们将继续之前章节所用的二手电视游戏商店项目，为其添加购物车功能。购物车是购物 / 在线服务网站的标配功能，同时还是阐述服务器端通信的理想操练场。我们后续会深入购物车功能的细节。话虽如此，但首先需要讨论示例项目的一些新要求。

7.1 示例项目新要求

不像前面的章节，为了本章代码能够运行，我们需要一个服务器软件。由于大多数 MV* 框架，包括我们用到的（AngularJS），都是服务器无关的，因此我们可以

挑选任何服务器软件。我们还可以使用自己喜欢的服务器端语言。因此，不论你喜欢 JavaScript、PHP、Python、Ruby、.NET、Java 或是别的什么服务器端开发语言，都是完全没问题的。

但无论你选择哪个服务器 / 语言组合，都有两个硬性要求：

- 支持 RESTful 服务，因为我们的例子要使用 REST。
- JSON 支持，其要么是内建支持，要么通过插件提供支持。

示例所用的服务器端代码使用 Spring MVC（版本 4，它是一个基于 Java 的 MVC 框架）开发。但即使你不熟悉 Java 或 Spring，也不用担心。在本章的讨论中，我们只在概念上涉及服务器端代码。附录 C 中有服务器端代码的配置指南。如果你更喜欢别的什么服务器端技术栈，该附录一开始提供了一个服务器端对象及任务的总结，以便你能够相应地构造出自己的服务器端代码。完整的项目源代码可以在线下载。

现在，我们了解了一些项目背景，接下来看看 SPA 应用程序如何才能与服务器端通信。

7.2 与服务器端通信综述

对于各种类型的 Web 应用程序，尽管与服务器端通信的许多概念是一致的，但下面几节将就单页面应用程序环境中的一些基本概念进行介绍。同时也会突出 MV* 框架特定的通信方式。

7.2.1 选择数据类型

为了 SPA 应用客户端与服务器端间能够通信，两端都需要有能够对上话的共同语言。首要任务就是决定发送与接收数据的类型。为了便于描述，我们在本章后面部分将使用一个购物车示例。

当用户与购物车交互时——不论是添加条目、修改条目数量，还是查看购物车当前内容——我们都发送并接收 JSON 格式的文本。JSON 是 SPA 应用程序与服务器端通信时的常用格式（尽管数据类型可以从纯文本到 XML、文件的各种格式）。

尽管使用 JSON 格式文本作为通用数据交换格式，但它只不过是系统原生对象的表述形式。要让文本能为我所用，两端都需要对其进行转换。我们稍后会讨论它。为了确保原生对象的正常转换，两端还必须尽责确保在调用中使用约定的 JSON 格式。

调用服务器端时，请求可以包括合适的互联网媒体类型（Internet Media Type, MIME）相关信息，因为资源可用于各种语言及媒体类型。之后服务器端可以响应

其认为最合适的某个版本的请求资源。这叫作内容协商 (Content Negotiation)。对于本项目, 我们只对 JSON 感兴趣。为了表达我们的这个想法, 可以为数据交换显式声明一种 `application/json` 的互联网媒体类型。

互联网媒体类型

互联网媒体类型 (Internet Media Type, MIME) 是一种标准方式, 用来识别在两个系统间进行交换的数据。其被许多互联网协议所使用, 包括 HTTP。互联网媒体类型格式为类型/子类型, 在本章中, 我们使用 `application/json` 媒体类型: 其类型为 `application`, 子类型为 `json`。

如果有需要, 还可以通过分号添加可选参数。例如, 为了指定 `text` 媒体类型, 其子类型为 `html`, 同时字符编码为 `UTF-8`, 可以这么表述: `text/html; charset=UTF-8`。

使用 HTTP 头信息 (HTTP Header) 指定互联网媒体类型, HTTP 头信息是传输中发送的字段, 用来提供请求、响应或消息体所包含内容的相关信息。Content-type 头信息告知其他系统在请求和响应中期待包含哪些内容。也可以在请求中指定 Accept 头信息, 让服务器端知道应返回的合适媒体类型。

选择数据类型之后, 就需采用合适的请求方法以确保调用成功。下一节介绍 SPA 应用的常见请求方法。

7.2.2 HTTP 请求方法

当客户端发起请求时, 其可以通过指定请求方法 (Request Method) 来表明希望服务器端执行的动作类型。然而, 为了确保请求成功, 服务器端调用代码必须支持请求中指定的 HTTP 请求方法。如果不支持, 服务器端就可能响应 405 “Method Not Allowed” 状态码。

由于 HTTP 请求方法描述了请求中所表述资源的动作, 因此它通常被称为调用动词 (Verb)。不修改资源的请求方法, 比如 GET, 被认为是安全的。任何请求方法, 不管对其执行多少次调用, 最终的结果都相同, 则认为其是等幂的。再如, 当用户想要修改购物车中特定条目的数量时, 我们使用 PUT。由于 PUT 是等幂的, 可以连续 10 次通知服务器端我们想要两份 “Madden NFL”, 而 10 次之后, 我们的购物车中仍只有两份。

表 7.1 定义了我们购物车示例中用到的常见 HTTP 请求方法。尽管不是一个完整列表, 但差不多代表了单页面应用程序中用到的最常见请求方法。

表 7.1 SPA 常用 HTTP 方法

方法	描述	示例	安全性	等幂性
GET	典型地, GET 用来获取数据	查看购物车	是	是
POST	该方法基本用于创建资源或添加条目到资源	添加条目到购物车	否	否
PUT	典型地, PUT 用于诸如修改或创建动作, 修改存在的资源, 并在资源不存在时可以选择添加它	修改购物车中条目的数量	否	是
DELETE	用来移除资源	从购物车中移除条目	否	是

HTTP 协议中指定了其他请求方法。完整列表请参考 http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods。

本节最后一部分内容是逻辑数据与 JSON 格式数据间的互换。

7.2.3 数据转换

确定数据类型之后, 客户端与服务器端都必须配置为发送和接受该特定类型。对于本章的购物车, 我们只使用 JSON, 因此两端的代码都应该能够对 JSON 格式数据进行编解码。

在客户端, 转换 JavaScript 对象为 JSON 格式数据的功能可能已经内建到 MV* 框架中了。如果是这样, 则该功能就很可能是缺省的, 当使用该框架发起服务器端请求时转换就会自动发生。万一自动转换功能并未内建到所用框架中, 该框架也可能提供一个定制类型的转换工具。对于 JavaScript 对象的转换, 可以使用 JavaScript 原生命令 `JSON.stringify()` :

```
var cartJSONText = JSON.stringify(cartJSObj);
```

在服务器端, 通过内建在服务器端语言中的 JSON 解析器或第三方库, JSON 格式文本转换为服务器端语言的原生对象。如同 HTTP 方法, 服务器端执行转换的具体方法也会有所不同。

为了阐述与服务器端通信的整个过程, 我们再次利用修改购物车的示例。假设用户增加了购物车里某个条目的数量。出于验证与处理此处修改的需要, 我们将把修改后的购物车发给服务器端。图 7.1 描绘了两端发生的转换过程。

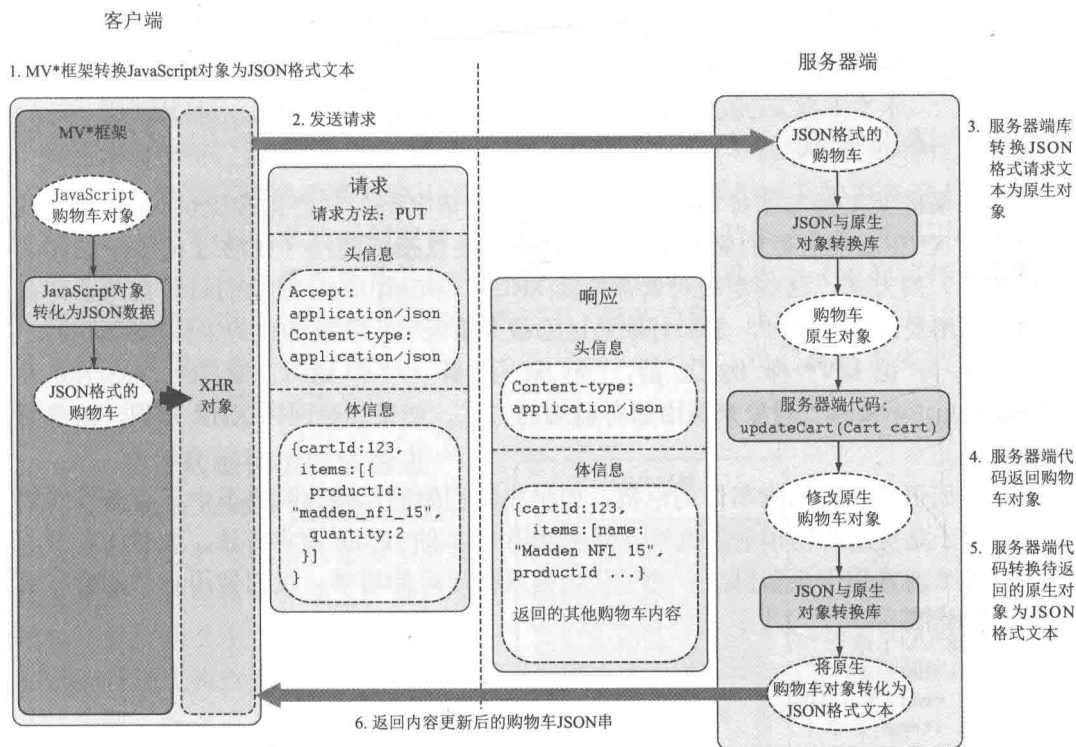


图 7.1 JavaScript 对象转换为 JSON，并在请求发起时添加到请求体。服务器端响应时，通过响应体，将修改后的购物车以 JSON 格式发回给客户端

修改函数调用之后，MV* 框架将 JavaScript 购物车对象转化为 JSON 格式文本。接下来，MV* 框架传递数据给 XMLHttpRequest API。之后 JSON 数据通过请求体发给服务器端。

当客户端收到响应之后，返回的文本将再次转换。这次它将转换回 JavaScript 原生对象。通常情况下，MV* 框架会为你自动处理转换过程。万一框架未自动处理，我们可以使用 JavaScript 原生命令 `JSON.parse()`：

```
var cartJSObj = JSON.parse(returnedCartJSONText);
```

现在我们讨论了与服务器端通信的整体情况，接下来回到客户端继续讨论 MV* 框架如何帮助我们简化此通信过程。

7.3 使用MV*框架

MV* 框架的一个伟大之处就在于通过抽象出大量相关样板代码来简化复杂任

务。在与服务器端通信时也是如此。本节内容具体包括发起请求与处理响应。在讨论中，我们将阐述一些化繁为简的处理方式。

7.3.1 请求生成

如果框架支持与服务器端通信，那么一般情况下，它会直接提供 XHR 对象或提供对 XHR 功能进行抽象的专有对象。不管是直接或间接（如通过 jQuery 这样的外部库）的方式，这些定制对象都扮演 XMLHttpRequest 对象的封装器角色。在请求调用及结果处理时，定制对象通过隐藏大量乏味、重复的任务来体现其优势。

在讨论 MV* 样例之前，先来了解一下通过原生 JavaScript 及 XMLHttpRequest 对象来调用服务器端的方式。如果你想回顾 XHR 知识，请参阅附录 B。

我们仍将使用购物车作为示例。如早先我们做的，修改购物车中已有条目的数量。由于是更新购物车条目数量，因此使用 PUT 的 HTTP 请求方法。如前面章节所述，PUT 通常用于更新修改。为了让过程来得尽可能简单，我们使用一个购物车数据的简化版：

```
var cartObj = {  
  cartId : 123,  
  items : [ {  
    productId : "madden_nfl_15",  
    quantity : 2  
  } ]  
};
```

清单 7.1 展示了直接通过 XMLHttpRequest 对象，以原生 JavaScript 方式来修改购物车。

清单 7.1 直接通过 PUT 与 XMLHttpRequest 来修改购物车

```
var cartJSON = JSON.stringify(cartObj);  
var xhrObj = new XMLHttpRequest();  
xhrObj.open("PUT", "/SPA/controllers/shopping/carts", true);  
xhrObj.setRequestHeader("Content-Type", "application/json");  
xhrObj.setRequestHeader("Accept", "application/json");  
xhrObj.send(cartJSON);
```

将 JavaScript 对象转换为 JSON 格式文本

创建 XMLHttpRequest 对象新实例

定义调用 Property

设置 Content-Type

发送请求

声明接受的数据类型

在这里，我们尚未处理结果，将其留待下一节处理。即便如此，我们还是涉及了一些底层细节。我们不得不手工设置 Content-Type 以及其他所需的头信息（如 Accept）。此外，还得手工转换 JavaScript 购物车对象为 JSON 格式文本。

通常情况下，如果 MV* 框架提供了开箱即用的与服务器端通信的特性，我们就很可能借助以下两种类型对象中的一个来生成请求：模型或某种类型的工具 / 服务对象。如果框架要求我们创建显式定义的数据模型，则很可能通过调用模型自身的功能来执行服务器端操作。如果框架并没有显式数据模型（框架将任何数据源都视作隐式模型），就可能借助框架的工具 / 服务对象。比如 AngularJS，其提供了与服务器端通信的两个服务：\$http 和 \$resource。我们将在本实例中使用 \$resource，稍后就会看到它。

1. 通过数据模型生成请求

在一些框架中（比如 Backbone.js），我们通过扩展框架内建模型对象来显式定义数据模型。通过扩展框架模型，就自动继承了大量功能，包括在远程资源上执行完整的 CRUD（创建、读取、修改以及删除）操作（如图 7.2 所示）。

如果我们需要自定义调用，也不用太过担心。大多数框架能够让开发者重写并定制自己的开箱即用行为。

清单 7.2 扩展 Backbone.Model 对象以定义购物车，并传进相应的属性名称作为购物车 ID。其中还针对所有服务器端请求定义了基本 URL。整个过程我们只需做一次模型定义。

模型定义好了之后，就可以在任何需要使用它的场景创建其新实例。之后所有购物车新实例都会继承与服务器端通信时所需的全部功能。

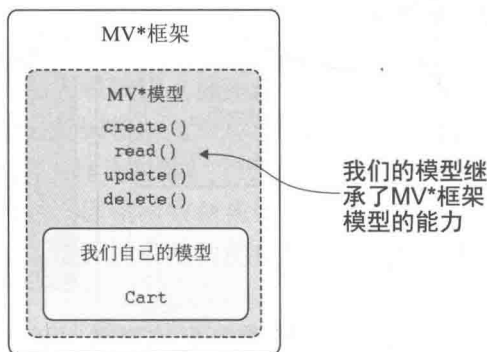


图 7.2 在 MV* 框架中，扩展自框架的模型自动继承了父对象的能力，诸如生成服务器端请求

清单 7.2 Backbone.js 版本的购物车修改功能

```
var Cart = Backbone.Model.extend({
  idAttribute: 'cartId',
  urlRoot: 'controllers/shopping/carts/',
});

var cartInstance = new Cart(cartObj);
cartInstance.save();
```

创建一个模型新实例

定义模型，其带有一个 URL 和一个 ID

调用继承来的 save() 函数以发起请求

Backbone.js 代码的确减少了复杂度。其在幕后也帮我们完成了一些任务。首先，

其假设我们处理的是 JSON（除非我们告知它别的格式），并自动将传入构造函数的对象转换为 JSON 格式文本。另外，其自动设置 Content-type 和 Accept 头信息为 JSON。最后，它能够根据请求对象是否具有一个 ID 来决定使用 PUT 还是 POST。同样，这些特性可以定制或重写。

2. 通过数据源对象生成请求

MV* 框架生成请求的其他方式还包括借助独立的数据源对象来生成请求。当诸如 AngularJS 这样的框架允许我们使用任何数据源充当数据模型时，通常就采用这种方式。无须扩展父对象，特性继承也就无从谈起。在这种情况下，框架提供了一个数据源对象，可以把我们的模型传递给它，以生成请求调用（如图 7.3 所示）。

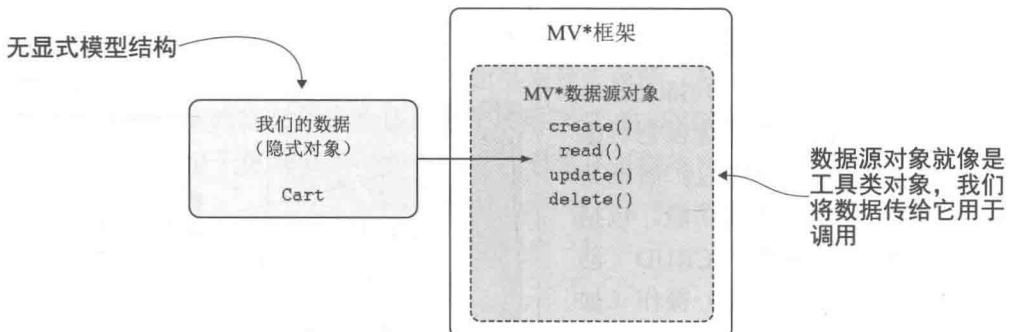


图 7.3 框架提供了与服务器端通信的功能，但并不支持模型扩展，相反，其更可能提供一个数据源对象

下面来看看通过数据源对象生成请求的例子。清单 7.3 使用 AngularJS 的 \$resource 对象来更新购物车。前面说过 \$resource 是 AngularJS 的一种服务，用来与服务器端通信。它针对请求生成及服务器端响应处理提供了大量特性。在本章的项目里，我们将深入 \$resource 的使用以理解项目。现在，我们先来了解这种风格的 MV* 代码，作为与原生 JavaScript 调用服务器端方式的比较。

清单 7.3 AngularJS 版本的购物车修改功能

```
var CartDataSrc =
  $resource(
    "controllers/shopping/carts", null,
    {updateCart : {method : "PUT"}}
  );
CartDataSrc.updateCart(cartObj);
```

定义一个 URL 用于调用，没有 URL 参数 (null)

通过内建的 \$resource 对象来创建数据源实例

除了 update(), AngularJS 的 \$resource 对象具备所有 CRUD 操作；但我们可以配置 update()

使用我们添加的 updateCart() 方法

虽然没有扩展任何对象，但整个概念与第一个 MV* 例子相同。我们可以借助 MV* 框架来生成请求。就像 Backbone.js, AngularJS 在幕后设置了合适的头信息，将 JavaScript 对象转换成 JSON，并使用我们定义的 HTTP 方法。然而如你所见，AngularJS 并不内置包含修改购物车的功能（PUT）。但通过定制数据源对象来添加更新功能是非常简单的。

MV* 框架提供的另一个特性是能够简化服务器端调用结果的处理过程。某些框架通过回调函数来处理该项任务，而另一些框架则依赖 Promise 来处理。Promise 在 MV* 框架中正变得越来越流行，但首先得确保我们理解使用异步请求的回调。

7.3.2 通过回调函数处理结果

当我们处理异步任务时（诸如调用服务器端以修改购物车），在等待服务器端响应时往往不希望应用就停在那儿。有时候我们需要继续处理其他任务，而等待响应可以放在后台执行。因此，我们通过把回调函数传入修改函数来处理修改任务执行完的结果，而非等待修改函数执行完毕后返回值。之所以能够这么做，是因为函数可以被传递。这就允许任意函数将其他函数作为参数。

当回调函数作为参数传进另一个函数时，回调函数就像该函数的延展。回调函数能够获取控制并在控制获取点继续执行。这种回调函数的使用方式被称为后续传递风格（Continuation-Passing Style, CPS）。

接下来我们来看一下使用后续传递风格的编程方式来处理服务器端调用结果。由于 Backbone.js 支持回调函数方式，因此使用它来阐述。让我们给前一版本的购物车修改功能添加些处理。图 7.4 是总体描述。

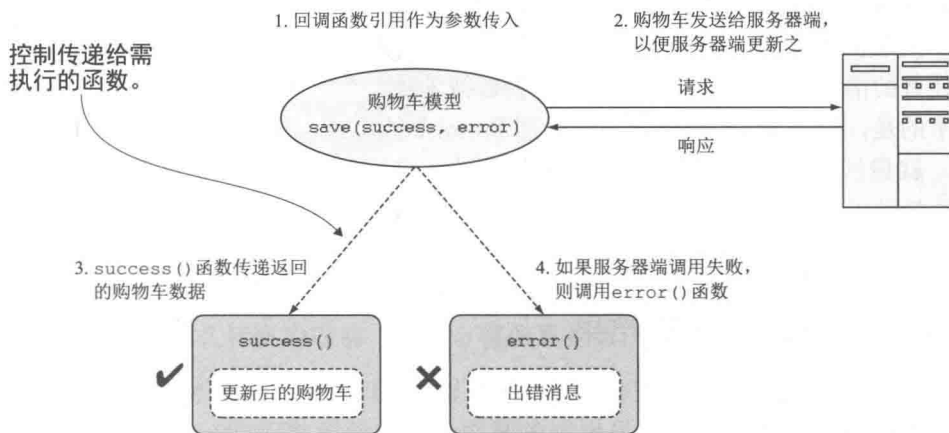


图 7.4 通过回调函数，在服务器端调用处理完成后，控制从 save() 函数传递到 success() 函数或 error() 函数

如果调用成功, `save()` 函数就通过 `XHR` 对象调用 `success()` 函数, 并把返回的购物车数据传递给 `success()` 函数。如果服务器端调用失败, `save()` 函数就调用 `error()`, 传递失败的具体信息给 `error()` 函数。无论哪一种情况, 处理都会从模型的 `save()` 函数继续传递到其中某个回调函数。

现在来看看代码。我们发起跟之前 `Backbone.js` 示例相同的请求, 但这里需处理调用结果 (如清单 7.4 所示)。

清单 7.4 通过回调函数处理购物车更新过程

创建新的
购物车实例

```
var cartInstance = new Cart(cartObj);

cartInstance.save(null, {
  success : function(updatedCart, response) {
    console.log("Cart ID: " + updatedCart.id);
  },
  error : function(cartUnchanged, response) {
    console.log("Error: " + response.statusText);
  }
});
```

保存之前, 没有要改变的模型属性 (null)

定义 `save()` 中的回调函数

这段代码不仅更易读, 而且还能够传递配置对象给 `save()` 函数。在该对象中可以定义 `success` 与 `error` 回调函数, 以及 `save()` 支持的其他所需配置项。`Backbone.js` 还能自动传递服务器端调用结果给我们定义的回调函数。

在成功调用后, 我们可以访问更新后的购物车对象以及服务器端响应。如果调用失败了, 则可以通过响应获取失败原因。此外, 如果需要用到底层调用功能, `save()` 方法同时也返回了一个 `jQuery` 的 `jqXHR` 对象, 它是一个 `XHR` 封装器。关于 `jqXHR` 的更多内容, 请参阅: <http://api.jquery.com/jQuery.ajax/#jqXHR>。

对于处理简单结果, 回调函数方式很容易使用, 也很合适。但在调用完成后要执行多任务的情况下, 后续传递风格有时会显得笨拙。

庆幸的是, 许多 `MV*` 框架倾向于返回 `Promise` 的方式, 而非依赖后续传递风格的回调。就像回调函数, `Promise` 也是非阻塞的: 应用程序不用停下来等待调用结束。这种特性很适合用于异步处理。在下一节里会看到, 对于结果处理的复杂需求, `Promise` 能够提供额外的 `Property` 与行为, 使得我们的开发更轻松。

7.3.3 通过 `Promise` 处理结果

`Promise` 是一个对象, 代表仍未结束的过程的结果。如果 `MV*` 框架支持 `Promise` 特性, 则 `MV*` 框架执行异步服务器端调用的函数将返回一个 `Promise`, 该 `Promise` 作为最终调用结果的代理。通过 `Promise` 就可以规划复杂的结果处理安排。为了掌握 `Promise` 用法, 首先必须理解其在调用前后的内部状态。

1. Promise 状态

使用 Promise 的一个好消息是，其只存在于以下三种状态中的一种：

- 成功（Fulfilled）——这是过程处理成功的 Promise 状态。Promise 中包含的值信息为运行处理结果。对于购物车修改过程，则是服务器端返回的修改之后的购物车内容。
- 失败（Rejected）——这是过程处理失败的 Promise 状态。Promise 中包含了失败原因（通常是一个 Error 对象）。
- 待决（Pending）——这是过程处理完成之前的 Promise 初始状态。在这个状态中，Promise 既不是成功也不是失败的状态。

这三种状态是互斥的最终状态。在 Promise 状态成功或失败之后，则认为过程处理完毕且不能转换成其他状态。图 7.5 使用购物车来描述 Promise 的这三种状态。

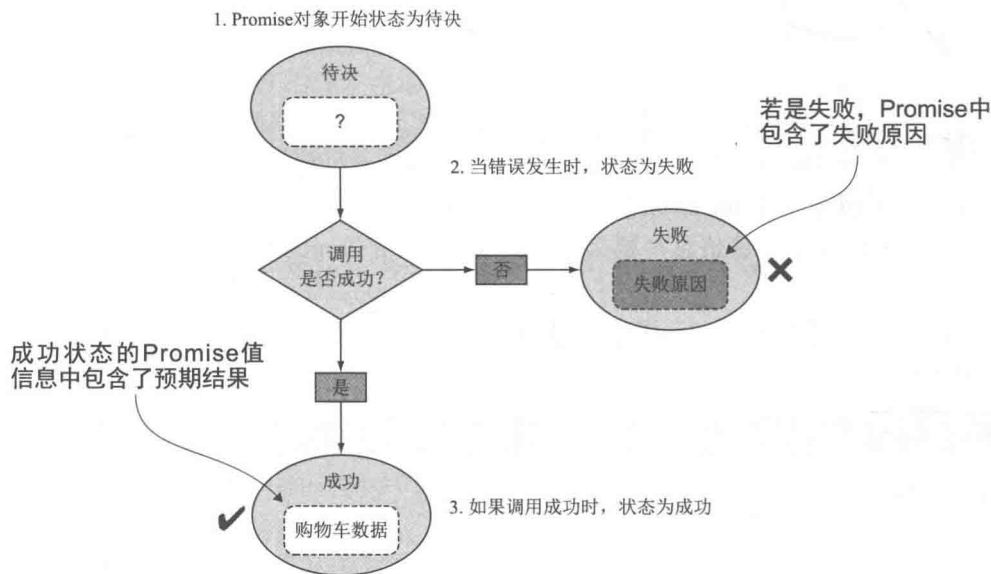


图 7.5 一个 Promise 对象有三种互斥状态：待决、成功、失败

分配给 Promise 的变量在等待函数返回时不会保持空引用。相反，Promise 在待决状态中会立即返回一个完整对象，其带有一个未决的值信息。当处理完成时，Promise 状态要么转变为成功，Promise 值信息包含了调用结果；要么转变为失败状态，Promise 包含失败原因。

2. 访问处理结果

我们尚未讨论 Promise 返回之后如何访问处理结果。Promise API 提供了几个有用的方法，但使用最多的一个是 `then()` 方法。

`then()` 方法注册回调函数让 `Promise` 返回处理结果。在这里定义的函数称作 *Reaction*。第一个 *Reaction* 函数代表 `Promise` 成功的情况。第二个是可选的并代表 `Promise` 失败的情况：

```
promise.then(  
  function (value) {  
    // 处理成功值信息的 Reaction  
  },  
  function (reason) {  
    // 可选，处理失败原因的 Reaction  
  }  
);
```

因为失败的 *Reaction* 是可选的，因此可以通过便捷方式使用 `then()` 方法：

```
promise.then(function (value) {  
  // 处理成功值信息，忽略失败情况  
});
```

对于 *Reaction* 函数，需要记住：不管用哪种代码格式，这两个函数只有一个会执行——永远不可能两个都执行——要么执行这个，要么执行另一个。就这一点来看，这多少有点类似于一个 `try/catch` 块。还要注意 *Reaction* 函数的参数是 `Promise` 返回结果（要么是成功后的值信息，要么是失败原因）。当 `Promise` 返回了结果时，你就能处理它们了。

现在实际看一下 `then()` 函数。清单 7.5 修改购物车，并用 `Promise` 取代回调函数来处理结果。

清单 7.5 通过 `Promise` 来处理购物车更新结果

```
更新函数  
返回一个  
Promise    CartDataSrc.updateCart(cartObj).$promise  
           .then(  
             function(updatedCart) {  
               console.log("Cart ID: " + updatedCart.id);  
             },  
             function(errorMsg) {  
               console.log("Error: " + errorMsg);  
             }  
           );
```

通过 `Promise` 的
`then()` 函数来访问结果

AngularJS 的 `$resource` 方法内建了 `Promise` 返回。如本例所示，跟之前一样，可以输出调用结果到控制台——但这次我们打算使用返回的 `Promise` 对象来取代将控制转移到回调函数。`then()` 方法传递成功结果或失败原因到相应的 *Reaction* 函数。

`Promise` 的另一个好处是可以链式集中调用多个 `then()` 方法（如果服务器端调用之后需要执行多个任务的话）。

3. Promise 链

通常过程运行之后，还希望做一些处理。而我们可能需要这些处理能够按序发生，以确保某个事件成功之后才发生下一个事件。这不仅可能而且还很容易用 Promise 实现（参见清单 7.6）。

注意 jQuery 的 Promise 实现不支持本节描述的每个场景。具体请参考 <https://blog.domenic.me/youre-missing-the-point-of-promises>。

到目前为止的购物车更新功能能够在控制台打印出结果。在真实应用中，我们可能希望在服务器端调用结束之后执行下列任务：

- (1) 重新计算购物车总数，应用必要的折扣。
- (2) 根据结果修改视图。
- (3) 重新利用消息服务来更新用户消息——调用成功。

此外，我们还希望这些任务顺序执行，并只有在每个任务成功执行之后，下一个任务才能够开始。在沿途发生问题时，这将确保用户不会被错误告知一切顺利。

清单 7.6 通过 Promise 规范控制流程

```
$resource 返回一个 Promise
var promise = CartDataSrc.updateCart(cartObj).$promise

promise.then(function( updatedCart ) {
    return shoppingCartSvc
        .calculateTotalCartCosts(updatedCart);
})
.then(function(recalculatedCart) {
    replaceCartInView(recalculatedCart);
})
.then(function() {
    messageSvc.displayMsg("Cart updated!");
})
["catch"](function(errorResult) {
    messageSvc.displayError(errorResult);
});
```

返回重新计算后的购物车给接下来的 then() 使用

显示重新计算后的购物车

显示一条用户消息

处理沿途发生的任何错误

这段代码将正确运行，因为每个 then() 函数都返回一个 Promise。如果前面的 then() 的 Reaction 返回一个 Promise，则该 Promise 的值在随后的 Promise 中使用，会传递给接下来的 then()。如果 Reaction 返回简单值，这个值就成为随后 Promise 中的值。这样的方式允许我们将 Promise 以直接而清晰的方式链接在一起。

通过几行代码，将多任务依序链接在一起的能力非常神奇。但 Promise 链的作用还不止这些。Promise 链的另一个神奇之处是，在链上能够有多个异步过程。

4. 依序链接多个异步过程

有时在我们需要依序运行多个任务的时候，其中可能不止一个异步任务。由于不知道异步过程何时会结束，因此试图将其与其他任务一起放入序列中确实是一种挑战。然而，通过 **Promise** 来实现的话，事情就会变得很容易。因为每个 `then()` 在下一个 `then()` 执行之前已得以执行，整个链是顺序执行的。即使多个异步过程存在于链中也是这个理儿。

出于描述的需要，假设服务器端 API 需要在随后的 GET 调用中使用购物车更新功能返回的购物车 ID，以在屏幕上正确显示购物车。清单 7.7 描述了如何通过 **Promise** 实现这个需求。

清单 7.7 依序执行多个服务调用

```
var promise = CartDataSrc.updateCart(cartObj).$promise

promise.then(function( cartReturned ) {
    return shoppingCartSvc
        .getCartById(cartReturned.cartId);
})
.then(function(fetchedCart) {
    return shoppingCartSvc
        .calculateTotalCartCosts(fetchedCart);
})
.then(function(recalculatedCart) {
    replaceCartInView(recalculatedCart);
})
.then(function() {
    messageSvc.displayMsg(userMsg);
})
["catch"](function(errorResult) {
    messageSvc.displayError(errorResult);
});
```

将更新功能返回的购物车用于下一个服务调用

将所获取的购物车用于重新计算

在这条链上，更新功能第一个执行。在其返回之后，触发下一个服务调用。由于来自 `$resource` 的 GET 调用已创建了一个 **Promise**，因此该 **Promise** 的值将在随后的 **Promise** 中使用，并传递给接下来的 `then()`。

在 **Promise** 讨论结束之前，快速浏览一下错误处理。我们已经在几个例子中看到了错误处理，但尚未涉及细节。

7.3.4 Promise 错误处理

我们可以用两种方式处理失败的 **Promise**。早先我们看到了第一种方式。该方式使用 **Promise** 的 `then()` 方法的第二个 **Reaction** 函数。第二个 **Reaction** 函数在调用过程失败时触发：

```
promise.then(  
  function (value) {  
  },  
  function (reason) {  
    // 失败处理逻辑  
  }  
);
```

第二种方式是在 Promise 链的最后添加名为 `catch()` 的错误处理方法：

```
.catch(function (errorResult) {  
  // 失败处理逻辑  
});
```

一些浏览器存在 `catch()` 方法名冲突，因为它是 JavaScript 语言的既存称谓。作为替代，你可以使用这种语法：

```
["catch"](function(errorResult) {  
  // 失败处理逻辑  
});
```

提示 将 `.catch()` 写成 `["catch"]` 看起来很奇怪，但能规避不支持 ECMAScript 5 的旧式浏览器的潜在问题。如果使用该语法，如例子中所见，注意它的前面没有点号。

我们已经看到了购物车调用中使用了第二种方式。其使用消息服务来记录错误信息并广播消息给用户。

```
["catch"](function(errorResult) {  
  messageSvc.displayError(errorResult);  
});
```

对于任何一种错误处理方式，失败状态都会依链传递下去，直到第一个可用的错误处理程序。对 `catch()` 方式而言这种行为看起来很明显。而使用可选 `Reaction` 函数进行错误处理的方式则看起来不太明显，但确实也是这么回事。如果失败在链上某处发生，并且在链上遇到任何一种错误处理方法（即便错误处理程序在数个 `then()` 之后），则错误处理程序将触发，同时抛出的错误会传给它。

如购物车示例中所描述的那样，Promise 很强大且易于使用（只要你掌握它）。除了这里涉及的 Promise 特性，框架及库有时候会添加更多的 Promise 特性。具体细节可参考其对应的文档。

尽管项目需要支持老式浏览器，但仍能够借助 MV* 框架来使用 Promise（只要框架或第三方库支持 Promise 特性）。以下列出了本书写作时流行的 Promise 第三方库：

- bluebird——<https://github.com/petkaantonov/bluebird>
- Q——<https://github.com/krisKowal/q>
- RSVP.js——<https://github.com/tildeio/rsvp.js>
- when——<https://github.com/cujojs/when>
- WinJS——<http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx>

所有这些都是 Promise 库，除此之外，它们还遵从当前最主要的 Promise 标准——*Promise/A+*。其也是原生 JavaScript 的 Promise 所遵从的标准。如果想深入阅读 *Promise/A+* 规范，可以看看这个不错的资源：<https://github.com/promises-aplus/promises-spec>。

补充一点：jQuery 也实现了其版本的 Promise，但本书写作时其尚未遵从 *Promise/A+*。jQuery 的 Promise 特性通过 Deferred 对象实现。若是对此感兴趣，可以了解一下 jQuery 官方文档：<http://api.jquery.com/category/deferred-object>。

JavaScript 的 ECMAScript 6 版本也已实现 Promise。甚至在规范完成之前，许多现代浏览器也已有限支持 Promise。

现在，离实现本章示例还差一步——RESTful 服务调用。

7.4 RESTful Web服务调用

本节阐述 SPA 调用 RESTful Web 服务。RESTful Web 服务在当下 SPA 项目中得以普遍应用。

7.4.1 什么是 REST

REST 可以表述为表述性状态转移 (Representational State Transfer)，其不是协议，甚至也不是规范，它是针对分布式超媒体系统的一种架构风格。其受到广泛欢迎，大量 MV* 框架不仅对它提供了内建支持，而且默认支持这种风格。

在 RESTful 服务中，API 定义了用于表述资源及驾驭应用状态的媒体类型。API 中的 URL 和 HTTP 方法为给出的媒体类型定义处理规则。HTTP 方法描绘做了什么，URL 唯一标识动作影响的资源。通过描述 REST 指导原则集可以准确定义 REST。

7.4.2 REST 原则

本节阐述 RESTful Web 服务调用时最常用到的一些 REST 原则。这些内容也能够让你更好地理解 REST 概念。

1. 一切皆是资源

REST 的一个基本概念就是一切皆资源。资源用类别表示，且从概念上映射到一个实体或实体集。资源可以是文档、图像或代表诸如一个人的对象。资源的概念也可以扩展为服务，比如当天天气，或者本章例子中的购物车。

2. 每个资源都需要有一个唯一标识

RESTful 服务中的每个资源都应该用一个唯一 URL 来标识它。这通常需要创建并分配一个唯一 ID 给资源。得确保 URL 中使用的任何 ID 不会危及应用程序的安全性或完整性。常见的安全方式是分配随机生成的 ID 给所有私人的或保密的资源。为了确保只有预期用户使用这些 ID，服务器端代码需确保请求者是该资源的认证用户，且具备在该资源上执行动作的相应权限。

3. REST 强调组件间接口的一致性

我们已经了解过 HTTP 方法如何作为 Web 服务调用的动词。资源标识符及 HTTP 方法用于提供资源访问的统一方式。表 7.2 描述了示例项目中的一些例子。

表 7.2 REST 中的 URL 唯一标识一个资源，HTTP 方法描述了在资源上执行的动作

REST
URL : /shopping/carts/CART_ID_452 方法 : GET 目的 : 获取购物车
URL : /shopping/carts/CART_ID_452/products/cod_adv_war 方法 : POST 目的 : 添加条目到购物车
URL : /shopping/carts/CART_ID_452 方法 : PUT 目的 : 修改购物车内容
URL : / shopping/carts/CART_ID_452/products/cod_adv_war 方法 : DELETE 目的 : 从购物车中移除指定产品的所有实例

很重要的一点是，所采用的 URL 风格并不是 REST 的一部分，尽管有时候会在一些 REST 文章中看到 *RESTful URL* 这样的字眼。

4. 交互是无状态的

请求间的应用程序会话状态应在 SPA 客户端保持，服务器端不该保存客户端上下文。SPA 发出给服务器端的每次请求应该传送所有需要的信息，以完成本次请求及允许 SPA 应用过渡到新状态。

再次说明一下，我们仅仅涉及 REST 的一点皮毛。关于 REST 及其架构的更多信息，请参见 http://en.wikipedia.org/wiki/Representational_state_transfer。

7.4.3 MV* 框架的 RESTful 支持

要理解及应用 REST 颇有一番难度。幸运的是，像 Backbone.js 和 AngularJS 这样的 MV* 框架提供了很好的 REST 默认支持。比如使用 Backbone.js 来更新购物车，Backbone.js 会自动添加模型的 ID 到 URL，以便 URL 唯一识别请求中的资源。而像 AngularJS 这样没有显式模型的框架，则允许我们通过 URL 模板中的路径变量来创建 RESTful URL。我们马上就会看到路径变量，在讨论 \$resource 对象的使用时会涉及它。

MV* 框架同时能够帮助我们调用 RESTful 服务，使得发送正确的 HTTP 请求方法来得更容易。通常情况下框架要么封装好 GET、POST、PUT 和 DELETE 相关的功能，要么允许我们通过配置轻松生成这些功能。

现在，我们对 RESTful 服务及其指导原则有了一个大致的了解。最后我们来操练示例项目。在该项目中，会直接看到 Promise 如何与 REST 亲密配合以管理购物车。

7.5 示例项目细节

继续前一章的二手电视游戏项目，在这里要添加购物车功能。按照惯例，仍将使用 AngularJS 作为我们的 MV* 框架。AngularJS 内建支持 Promise 和 RESTful 服务调用功能。如本章开头处提到的，对于应用程序服务器端，在这里只涉及其概念。

由于大量服务器端语言及框架可以取代本章使用的技术栈，所以附录 C 包含了针对每次调用，服务器端需要执行的任务小结。以这种方式，你就可以如愿地使用不同技术栈来创建服务器端代码。完整代码同样可以通过在线下载获得。接下来，首先来设置数据源。

7.5.1 配置 REST 调用

在本章早些时候，我们了解了 AngularJS 的 \$resource 对象。它使得 RESTful 服务调用来得更简单。同时它的所有方法都返回 Promise。我们将在示例项目中使用它来处理每次服务器端调用。尽管试图保持讨论内容独立于框架，但还是得花些时间来进一步了解 \$resource 对象的工作方式。乍一接触 \$resource 对象可能会被它吓住。然而在了解了其工作方式之后，就会发现它还是很容易使用的。在稍加介绍 \$resource 之后，我们就来讨论如何在 SPA 购物车服务示例项目中配置它。

使用 AngularJS \$resource 创建 URL

就像其他的一些 MV* 框架一样, AngularJS 通过 \$resource 对象内建支持 RESTful Web 服务调用。该对象针对底层 XMLHttpRequest 对象添加了许多语法糖, 以隐藏原本需要我们自己编写的大量样板代码。

\$resource 对象的主要目的就是让 RESTful 服务的调用更加简单。REST 原则之一就是有一个一致方式来表示资源。确定了 URL 风格之后, \$resource 工厂将帮助我们创建符合这种风格的 URL。

\$resource 工厂使得我们能够定义模板, 用来为各种 REST 调用创建资源 URL。要使用 \$resource, 可以传递一个 URL、可选的一些缺省参数以及一个可选的动作集合给其构造器:

```
$resource(DEFAULT URL, DEFAULT URL PARAMS, OPTIONAL ACTIONS)
```

我们的缺省 URL 为:

```
"controllers/shopping/carts"
```

如果未重写, 则使用该缺省 URL。但在本章示例项目里定义的定制函数将用自己的 URL 重写它。每个定制动作可以有自己的 URL。为了在 RESTful Web 服务所需结构中构建 URL, 可以使用 URL 路径参数。如同路由, 在 URL 中的某个字符串开头处使用冒号来指明参数。以下是我们配置里的 URL 示例, 其包括 URL 路径参数:

```
"controllers/shopping/carts/:cartId/products/:productId"
```

下一个参数是可选参数列表, 就像一个数据映射。其告知 \$resource 对象在一个或多个调用中, 可能会用到可选参数列表。该列表组织成键值对形式。左边是 URL 中参数的名称, 右边是该参数的值。@ 符号告诉 \$resource 对象该值是一个数据 Property 名称, 而不仅仅是一个字符串。随着这一表述, AngularJS 将扫描传递进来的数据对象以查找该名称 (指左边的 URL 参数名称) 对应的 Property, Property 值将在 URL 路径中使用。

```
{
  cartId : "@cartId",
  productId : "@productId"
}
```

例如, 假设我们传递一个 myCart 对象给调用, URL 参数 cartId 的值将取自 myCart.cartId。URL 参数 productId 的值将取自 myCart.productId。

将 \$resource 作为一个 REST URL 模板的好处是, 我们可以获取内置的一系列 REST 调用, 这些 REST 调用已用下列 HTTP 方法预先设置好了:

get() —— GET

query() —— GET (用于列表; 该动作的返回对象缺省为数组)

save() —— POST

delete() —— DELETE

remove() —— DELETE (等同于 delete() , 在 delete() 动作无法在某些浏览器中正常工作的情况下使用)

如果想要定制调用, 可以传进可选的命名函数集(在 AngularJS 中称为动作)。可以通过这些动作来创建完整的定制调用, 或者重写某个内置函数。例如, 为了创建一个定制动作 updateCart() , 可以将以下代码包含到我们的动作集中:

```
updateCart : {  
  method : "PUT",  
  url : "controllers/shopping/carts/:cartId"  
}
```

一旦配置好了 \$resource 对象, 通过它生成的所有调用都会自动返回一个 Promise。前面我们已经了解了如何通过 Promise 来处理调用结果。

在本章的示例项目中, 我们在购物车服务内部使用 \$resource 对象, 因为在数据返回给控制器之前, 需要做一些额外处理。对于简单数据的返回, 我们也许会考虑在另一个 AngularJS 对象 (比如工厂) 中包装 \$resource 对象, 并将其 (指另一个 AngularJS 对象) 直接包含到控制器中。

要了解有关 \$resource 对象的完整信息, 请访问 AngularJS 官方网站: [https://docs.angularjs.org/api/ngResource/service/\\$resource](https://docs.angularjs.org/api/ngResource/service/$resource)。

现在我们了解了 \$resource 对象的基本概念, 接下来看看购物车项目所用 \$resource 实例的完整代码 (如清单 7.8 所示)。这段代码展示了将在购物车服务内部生成的调用类型。

清单 7.8 REST 调用配置

```
var Cart = $resource("controllers/shopping/carts", {  
  cartId : "@cartId",  
  productId : "@productId"  
}, {  
  // 购物车方法  
  getCart : {  
    method : "GET",  
    url : "controllers/shopping/carts/:cartId"  
  },  
  updateCart : {  
    method : "PUT",  
    url : "controllers/shopping/carts/:cartId"  
  },  
  // 产品项级的方法  
  addProductItem : {  
    method : "POST",
```

分配所创建的 \$resource 给一个变量

定义缺省参数

定义动作


```
url : "controllers/shopping/carts/:cartId/products/:productId"
},
removeAllProductItems : {
  method : "DELETE",
  url : "controllers/shopping/carts/:cartId/products/:productId"
},
});
```

有了 `getCart()`，就可以随时获取购物车内容。可以通过 `addProductItem()` 来添加新产品到购物车，或使用 `removeAllProductItems()` 移除所给产品类型的所有内容。还能够使用 `updateCart()` 修改整个购物车。

提示 尽管在本示例项目中没有实现安全功能，但通常情况下，服务器端代码都要对生成的每次调用进行安全性和数据完整性的验证。

由于我们在前面章节就已着手开发本示例项目，因此在这里只会关注服务器端调用的相关代码，以及如何处理调用结果。首先给购物车添加新产品。

7.5.2 添加产品到购物车

根据早前选择的 URL 格式，我们在 RESTful 服务调用中包含购物车 ID 和产品 ID，以添加产品项到购物车（如表 7.3 所示）。如果产品已经在购物车中存在，则递增数量。

表 7.3 添加产品到购物车的 RESTful 调用

方法	URL	HTTP 方法	请求	响应
<code>Cart.addProductItem()</code>	<code>/shopping/carts/CART_ID_89/products/cod_adv_war</code>	POST	产品	购物车

我们修改产品显示页面，以包含一个新的按钮，其将触发添加产品项到购物车的调用。当单击按钮时，其调用购物车 `$resource` 配置中的 `addItem()` 动作。清单 7.9 修改视图以包含这个新按钮。

清单 7.9 修改产品显示视图

```
angular.module("data.appData", [])

<section class="product_info">
  <h2>
    {{results.name}}

    <span class="price">
      Used Price:
```

```

    {{results.discountPrice | currency:"$":0}}
  </span>

  <button id="product_info_add_btn"
    ng-click="addToCart('{{results.productId}}')">
    Add to Cart
  </button>
</h2>

<section id="product_info_img_container">
  
</section>

<section id="product_info_summary">
  {{results.summary}}
</section>
</section>

```

传进找到的游戏的产品 ID

图 7.6 展示了视图修改完成后的效果。

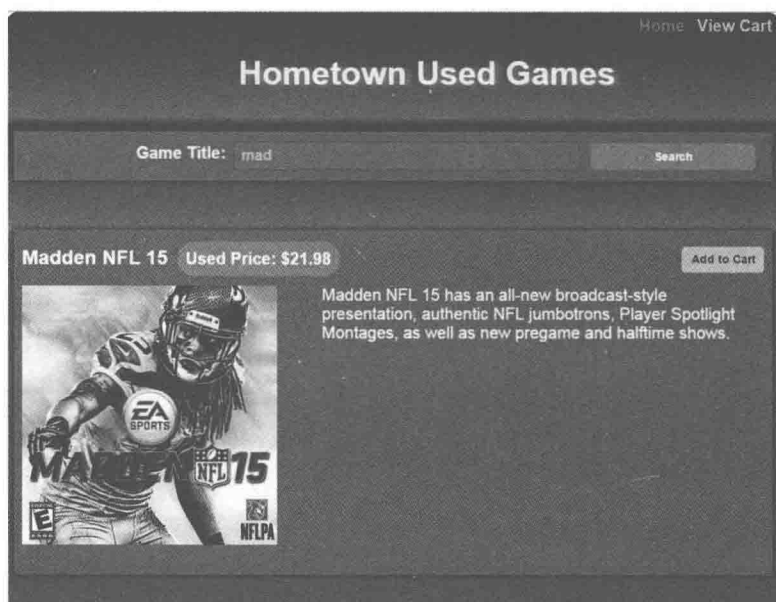


图 7.6 产品显示页面现在有了一个添加产品项到购物车的按钮

通过一个 `ng-click` 绑定，我们将按钮单击绑定到控制器中 `$scope`（模型视图）上的 `addToCart()` 函数，以处理这个新的用户动作。相应地，该函数调用购物车服务的 `addToCart()` 函数（如清单 7.10 所示）。提醒一下，AngularJS 的 `$stateParams` 对象允许我们访问执行路由的参数。

清单 7.10 应用数据持有购物车 ID

```
$scope.addToCart = function() {
    shoppingCartSvc.addToCart($stateParams.productId);
};
```

通过购物车服务来
添加产品项

函数调用发生之后，购物车服务中的 `addToCart()` 函数生成服务器端 RESTful 调用以处理业务逻辑（如清单 7.11 所示）。

清单 7.11 生成 `addProductItem()` 调用的函数

```
function addToCart(productId) {
    return Cart.addProductItem({
        cartId : createorGetExistingCart(),
        productId : productId
    }).$promise.then(function(cartReturned) {

        messageSvc.displayMsg("Item added to cart!");

        console.log("Item added successfully to cart ID "
            + cartReturned.cartId);
    })

    ["catch"](function(error) {
        messageSvc.displayError(error);
    });
}
```

传进 cart ID 和 product
ID 作为调用参数

Promise 链——显
示用户信息

处理任意错误

产品 ID 和购物车 ID 映射给早前 `$resource` 配置中 `addProductItem()` 定制动作的默认参数。用户添加产品项到购物车之后，需要一个新视图以显示购物车内容。因此，得为应用添加一个全新视图。

7.5.3 查看购物车

在此调用中，我们使用用户登录欢迎页面时在本地生成的购物车 ID。通过购物车 ID 可以获取购物车的当前状态。表 7.4 列出了调用 Property。

表 7.4 RESTful 调用，获取购物车以在视图中显示其内容

方法	URL	HTTP 方法	请求	响应
<code>Cart.getCart()</code>	<code>/shopping/carts/CART_ID_89</code>	GET	空	购物车

为了能够在任何地方查看购物车，我们在标头处添加一个新链接。单击该链接将执行 `viewCart` 路由，该路由将把我们带到购物车视图：

```
<a id="viewCartLink" ui-sref="viewCart">View Cart</a>
```

当购物车视图对应的控制器得以调用时，控制器首先会发起 GET 调用以从服务器端获取购物车数据。我们先来看看控制器发起的这个调用，之后再了解购物车服务。

在调用发起的控制器中，`getCart()` 函数返回 `$resource` 调用所生成的 `Promise`。如我们讨论 `Promise` 时提到的，这里引用的 `Promise` 在调用完成之前还是待决的：

```
var promise = shoppingCartSvc.getCart();
handleResponse(promise, null);
```

在该购物车控制器中，我们还将这个 `Promise` 传递给了一个 JavaScript 通用函数，该函数将处理 `Promise` 的返回。`Promise` 的一个好处是它们可以像任何其他 JavaScript 对象一样传递出去。在每次调用中，无论是获取购物车、修改购物车还是移除产品项，我们都以这个同样的方式来处理 `Promise`（如清单 7.12 所示）。

清单 7.12 处理所有购物车 `Promise` 的通用函数

```
function handleResponse(promise, userMsg){  
  
    promise.then(function( cartReturned ) { B  
        return shoppingCartSvc  
        .calculateTotalCartCosts( cartReturned );  
    })  
  
    .then(function( recalculatedCart ) {  
        replaceCartInView( recalculatedCart );  
    })  
  
    .then(function( recalculatedCart ) {  
        messageSvc.displayMsg( userMsg );  
    })  
  
    ["catch"](function( errorResult ) {  
        messageSvc.displayError( errorResult );  
    });  
};
```

传入 `Promise` 和 可选的用户消息

重新计算购物车

修改视图

显示用户消息

处理所有错误

在购物车服务中，得益于 MV* 框架中内建 REST 支持的魔法，获取购物车的调用变成了一行程序。在这里，我们传递一个带有购物车 ID 的对象作为调用负载。`$resource` 将扫描该对象以找到匹配 URL 参数 `cartId` 的 `Property` 名称。由于购物车 ID 已存储在客户端的 `cartData` 对象中，因此，URL 中要用到该 ID 的时候就可以使用 `cartData` 对象：

```
function getCart() {  
    return Cart.getCart({cartId : cartData.cartId})  
    .$.promise;  
}
```

别忘了 `Cart` 是分配给所创建 `$resource` 对象的变量名。当 `Cart.getCart()` 调用完成时，`Promise` 就返回给控制器用于之前所见的处理。如果调用完成时，`Promise` 链上的所有 `Promise` 都成功了，那么视图将显示购物车中的当前所有产品项。同时还会显示每个产品项的原始价格、二手价格以及省了多少钱。在购物车顶部显示的是所有产品项的总数及二手价格总和（如图 7.7 所示）。

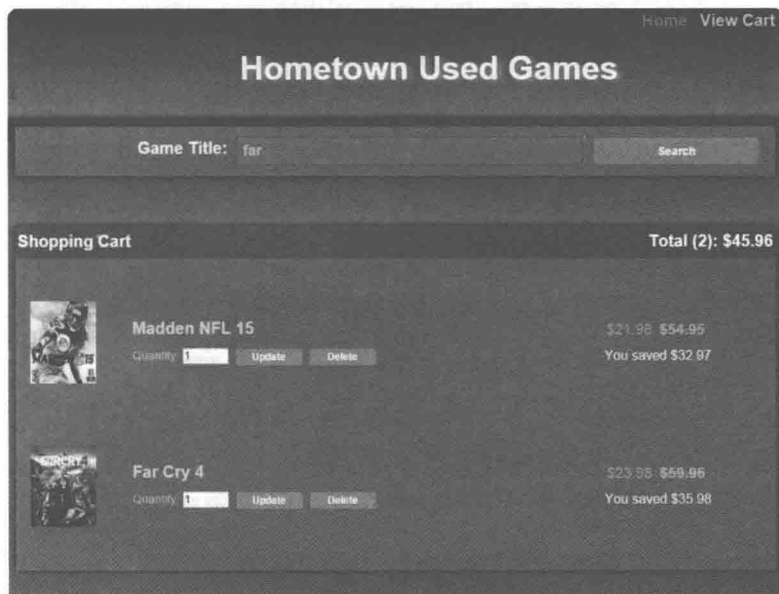


图 7.7 购物车视图允许用户修改购物车内容

购物车返回之后，用户就可以通过 UI 界面进行产品项的修改或删除操作。

7.5.4 修改购物车

修改购物车时，不是只发送新的产品项；我们要发送并接收整个购物车。RESTful URL 标识正在修改的购物车，以及请求体有修改的购物车数据。图 7.5 列出了调用 `Property`。

表 7.5 RESTful 调用，使用用户输入的新内容修改购物车

方法	URL	HTTP 方法	请求	响应
<code>Cart.updateCart()</code>	<code>/shopping /carts/CART_ID_89</code>	PUT	购物车	购物车

对于每个修改项，我们都提供了一个输入控件，让用户输入产品项的新数量。同时每个修改项还有一个按钮可以修改整个购物车。每个修改按钮以同样方式修改整个购物车。在每个修改项边上放上一个修改按钮只是为了操作上的便利。

```

<span class="cartQuantityLabel">Quantity: </span>

<input type="text" ng-model="game.quantity" size="4">

<button class="cartItemButton" ng-click="updateQuantity()">
    Update
</button>

```

updateQuantity() 函数不需要参数,因为它总是传递整个购物车。在控制器中,依靠购物车服务来发起修改请求,并传递返回的 Promise 给通用 Promise 处理函数(如清单 7.13 所示)。

清单 7.13 修改购物车的控制器代码

```

$scope.updateQuantity = function() {
    var uCart =
        shoppingCartSvc.createCartForUpdate($scope.cart);
    var promise = shoppingCartSvc.updateCart(uCart);
    handleResponse(promise, "Cart updated!");
};

```

发出修改调用, 返回 Promise

为请求体创建请求对象

传递 Promise 和用户消息给通用处理函数

在购物车服务中,使用一个 JavaScript 函数来创建发送给服务器端的购物车对象。为了让请求来得更简洁,在接下来的清单 7.14 中我们只包含 ID 和可更新 Property。

清单 7.14 创建要修改的请求对象

```

function createCartForUpdate(cartFromView) {

    var cart = {
        cartId : cartFromView.cartId,
        totalCount : cartFromView.totalCount,
        items : new Array()
    };

    angular.forEach(cartFromView.items, function(item) {
        var pItem = {
            productId : item.productId,
            quantity : item.quantity
        };
        cart.items.push(pItem);
    });

    return cart;
};

```

创建 items 数组

迭代购物车产品项, 并将其添加到请求中

当请求对象准备就绪时,就可以发起修改请求。再次得益于 MV* 框架对 REST 的支持,只需要一行代码即可:

```
function updateCart(cart) {  
    return Cart.updateCart(cart).$promise;  
};
```

如同其他调用，修改调用会返回 Promise 给控制器，以便 Promise 链可以处理结果。

最后我们讨论从购物车中移除产品项。下一节解释如何移除购物车特定产品类型的所有数量。

7.5.5 从购物车中移除产品

为了移除购物车中的某个产品，显然应该选择 DELETE 的 HTTP 方法。我们需要确保标识了购物车与产品，如添加产品时的处理。表 7.6 列出了调用 Property。

表 7.6 RESTful 调用，移除购物车某个产品的所有数量

方法	URL	HTTP 方法	请求	响应
Cart.removeAllProductItems()	/shopping/carts/CART_ID_89/ products/cod_adv_war	DELETE	空	购物车

除了让用户能够修改数量，每个产品项边上的删除按钮能够完全移除产品项。在这个视图中，我们绑定按钮单击到控制器上的 removeItem() 函数。该函数调用传递删除产品的 ID。

```
<button class="cartItemButton"  
ng-click="removeItem(game.productId)">Delete</button>
```

在控制器中，如同获取或修改购物车，我们发起调用，并传递返回的 Promise 给通用 Promise 处理函数（如清单 7.15 所示）。

清单 7.15 删除购物车的控制器代码

```
$$scope.removeItem = function(productId) {  
    var promise =  
        shoppingCartSvc.removeAllProductItems(productId);  
  
    handleResponse(promise, "Cart removed!");  
};
```

传递产品 ID，
获取返回的
Promise

传递 Promise 及用户
消息给通用处理函数

最后，我们来看看购物车服务中调用发起的代码（如清单 7.16 所示）。从购物车数据对象获取的购物车 ID 映射到 cartId URL 参数，传递进来的产品 ID 映射到 productId 参数。

清单 7.16 删除购物车的控制器代码

```
function deleteItem(productId) {  
  return Cart.removeItem(  
  
    {  
      cartId : cartData.cartId,  
      productId : productId  
    }  
  
  )$.promise;  
};
```

传递购物车 ID 和产品 ID 给调用

返回 Promise 给控制器

别忘了如果打算在自己的环境中创建项目，则可以参考附录 C 所补充的服务器端内容，其一开始就列出了服务器端对象及任务的总结信息。附录 C 包括这些内容是考虑到你有可能采用与本书不同的开发栈。同样，本章完整的样例代码可以在线下载。

7.6 挑战环节

又到了挑战环节，现在测试一下你对本章的掌握程度吧！在前一章的挑战环节中，你创建了一个电影搜索功能。用户在输入字段输入搜索内容，之后显示完全或部分匹配这些内容的电影标题。一个 Key-up 事件绑定到发布输入字段内容的函数上（通过每次按键触发发布动作）。还有一个搜索模块订阅主题并执行相应的搜索。同时，搜索模块使用发布 / 订阅模式来发布结果，结果显示在输入字段下方的无序列表中。

通过把基础数据和搜索逻辑放到服务器端来扩展该练习。我们仍维持一个客户端模块，用来监听要发布的输入字段内容。相应地，客户端模块每次监听到主题时都会触发服务器端调用。在服务器端，可以使用你中意的任何开发栈。服务器端调用采用 RESTful 服务调用的方式。通过 RESTful URL 和合适的 HTTP 方法来处理请求。使用 Promise 来处理服务器端调用结果。以上过程都成功之后，发布搜索结果。在控制台输出过程中产生的所有错误。

7.7 小结

本章涉及了不少内容，我们来回顾一下：

- 对于单页面应用程序而言，服务器端仍是非常重要的一环，其提供诸如安全、验证以及访问后台数据的标准 API 等特性。
- 客户端与服务器端之间有一个数据类型协议。同时，HTTP 方法对于调用成

功是至关重要的。

- 在请求和响应过程中，两端的原生对象转换为一致的数据格式。
- 支持服务器端通信的 MV* 框架通常都关注路由处理，诸如提供内建的标准请求类型及数据转换处理。
- MV* 框架支持服务器端通信的方式通常有两种，一种是扩展父模块，另一种是借助数据源对象。
- 调用结果的处理，要么使用后续传递风格的回调方式，要么使用框架或库提供的 Promise 方式。
- Promise 表示待决异步处理的结果，其起始时是待决状态，但在调用完成时最终会转换为完成或失败状态。
- Promise 有几个方法，但最常用的是 `then()`。该方法允许我们注册两个函数（它们被称为 *Reaction*）来处理成功或失败的 Promise。
- Promise 成功状态对应的 Reaction 允许我们访问处理结果数据。Promise 失败状态对应的 Reaction 包含了失败原因（通常情况下是一个 `Error` 对象）。
- Reaction 可以链接在一起，以控制一组处理流，即使是其他异步调用也在该链上。
- Promise 错误可以通过失败 Reaction 处理，也可以通过 `catch()` 方法处理。
- REST 可以表述为表述性状态转移（Representational State Transfer），它是开发 Web 服务的一种架构风格。
- REST 的一个基本概念就是一切皆资源，每个资源都应该用唯一 URL 来表示它。
- HTTP 方法描述了资源的动作。最常用的四个方法分别是 GET、POST、PUT 和 DELETE。

8 单元测试

本章内容

- 单元测试介绍
- 使用 JavaScript 测试框架
- 对 MV* 对象进行单元测试
- 测试驱动开发 (Test-Driven Development, TDD) 的单元测试介绍¹

开发一个软件，无论使用何种平台或语言，我们都希望尽力确保最终产品的质量。在本书中，我们已经看到了一些创建健壮而可维护 SPA 应用的方式，如将代码划分进模块、使用强大的 MV* 框架等。SPA 应用编程的另一个重要方面是测试。于产品最终交付而言，测试代码的重要性一点都不亚于业务代码本身。代码测试的效果远不止发现 bug。当看到代码按既定设想运行并能很好地处理异常时，这将帮助你写出更好的代码，同时增强你的开发信心。

尽管可以对应用程序进行各种各样的测试，但本章聚焦于单元测试。本章接下来的内容将覆盖单元测试概念、创建更有助于单元测试的代码，以及单元测试的好

¹ 原文是 Behavior-Driven Development (BDD)，但从本章内容来看，应该是 TDD，BDD 应该是作者的笔误。——译者注

处等内容。虽然单元测试的某些方面有赖于个人认知，但我们可以聚焦于它的最常用特性。

我们将讨论限定在通过 MV* 框架和模块化代码来构建单元测试上。在讨论完一些单元测试基础概念之后，将通过一个叫 QUnit 的框架来编写 SPA 应用的单元测试代码。QUnit 很强大，但却易于使用，且学习曲线比较平缓。

本章的示例项目非常简单，这将有助于我们聚焦单元测试任务。我们会使用 Backbone.js、Knockout 和 AngularJS 来构建项目，以阐述各种 MV* 框架风格之下的测试。

8.1 示例项目说明

对于本章示例项目，我们将创建一个基本的小费计算器。在后续编写单元测试代码时，会介绍这个项目的各个部分。首先，我们来看看项目的最终全貌（如图 8.1 所示）。

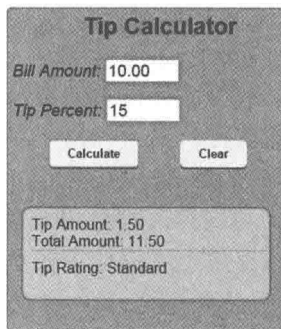


图 8.1 示例项目计算小费总数以及支付总数。同时评估小费等级

在随后的进展中请记住这张图。按照惯例，本章完整代码（这次包含了测试代码）可以在线下载。

8.2 什么是单元测试

从广义上来说，单元测试指在应用程序的最小可测部分上执行测试。此类测试是在开发阶段由开发人员执行的低层次测试。此外，不管测试主体是 MV* 对象还是一个普通模块，单元测试通常都会对代码行为方式进行断言。

单元测试还可以通过它与其他类型测试的关系来表述。按照诸如范围、时间及规模的度量，通常用一个金字塔结构来描述不同测试类型。由于单元测试的关注点和范围较狭窄、能够快速执行并易于管理，而且能够快速反馈结果，因此其处于金

金字塔底层（如图 8.2 所示）。

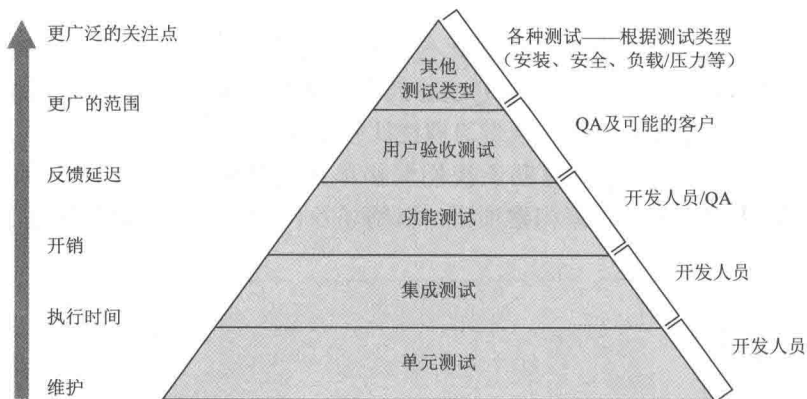


图 8.2 单元测试是在开发阶段创建的低层次针对性测试，能够快速执行并快速反馈结果

现在，我们了解了单元测试的基本概念，接下来讨论为什么要重视单元测试。

8.2.1 单元测试的好处

单元测试的好处远不止找出代码中的 bug。在实践中，我们可能会在其他更高层的测试中找到更多的 bug。单元测试的目标是设计出更好的软件。以下列出了单元测试的一些好处：

- 引领我们向更好的软件设计目标迈进——创建易于单元测试的代码有利于强化软件组件的“高内聚低耦合”的实践思想。这能够从整体上实现应用的更优设计。
- 有助于早期发现问题——一个合格的单元测试，应该是写得好而且确实有效，能证明所写代码按预期工作。越早侦测到代码 bug，就能够越早且花费更小开销纠正问题。
- 给你更多的信心做出改变——有时可能对做出改变犹豫不决，因为我们不想打破已在运行的某些功能。此时单元测试可以增强我们的信心。一个好的单元测试应该能够重复一致的结果。在我们重构的时候，同样的单元测试如果在改变发生之前是成功的，那么在改变发生之后也应该是成功的。
- 提供了代码运行的极好例子——尽管我们仍会考虑为应用编写文档，但单元测试提供了一个便利的方式，让其他人明白你的代码是如何运行的。编写良好的单元测试为那些希望快速深入代码的开发者描述了正确用法。单元测试还能够作为团队新成员的代码指引。

介绍完单元测试的好处，下面深入理解一下单元测试的编写原则。

8.2.2 构建更好的单元测试

尽管单元测试不存在官方规则手册，但我们可以遵从一些通用的最佳实践，让单元测试成为好的单元测试。

1. 每个单元测试都专注于特定业务相关概念

理想状态下，每个单元测试都应该专注于代码中的特定概念。如果单元测试范围太广，或者关注点超出了特定业务需求，那么就很难找准代码中未按设计要求正常工作的特定范围。

例如，不要试图启动一个单元测试让其自动通过用户操作——诸如表单填写或按钮单击之类的。尽管 SPA 应用中的处理经常由用户驱动的事件来触发，但 UI 层级的测试在范围上比单元测试要广泛得多，因为 UI 测试通常一次会在应用程序的各个层面生成多个活动。单元测试更应该测试低层次代码部分。

如果在代码中使用了第三方软件，你也可能忍不住想测试一下它是否像宣传中那样有效。但单元测试的重心应该只放在我们的逻辑代码上。如果第三方软件有 bug，这只是单元测试中的附带事件，应该排除在单元测试范围之外。

现在考虑一下我们的小费计算器示例。图 8.3 提供了一个概览，说明了示例项目中单元测试的范围。

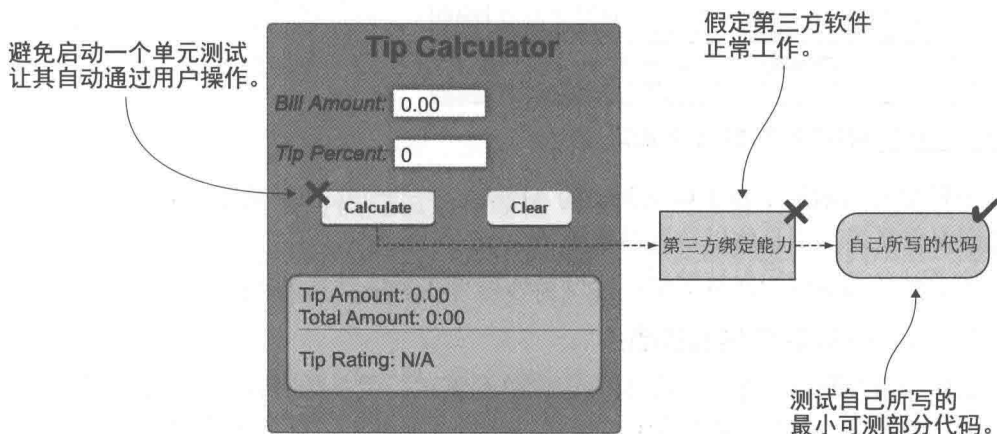


图 8.3 单元测试关注最底层的应用逻辑

创建可测代码的另一个更容易方式是避开模糊而不明确的 API。这些 API 通常没有得到很好的设计。如果你试图测试它们，就会吃到苦头。

2. 避免创建模糊且没有针对性的 API

写得很差的 API 不仅使得应用程序变得糟糕，而且让单元测试变得异常困难。

如果无法便捷测试业务逻辑的各个部分，则应该好好考虑一下是否需要重构代码了。

例如，小费计算器在计算按钮单击时，一次显示好几个值。我们应该要能够测试每个独立计算的逻辑。由于这个应用很简单，因此在一个函数中进行所有计算是件相当容易的事情（如图 8.4 所示）。

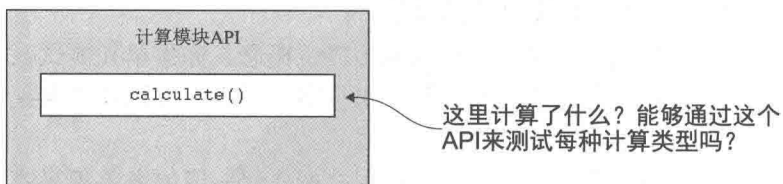


图 8.4 在这个设计里，所有的逻辑都隐藏在这个模糊的 API 里

现在我们来重构一点点代码，并在 API 中为业务逻辑的每个关注点创建其自己的函数。图 8.5 分离了原先揉成一团的任务。

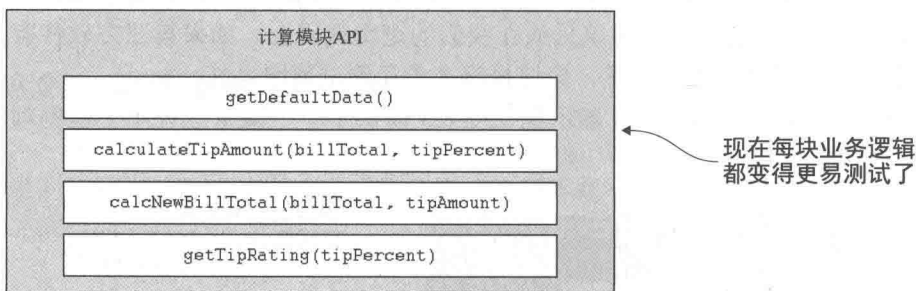


图 8.5 小而具体的函数更适合单元测试

当开发人员创建了易于单元测试的 API 时，一个让人愉快的附带效果是代码往往也变得更易读、更具体以及更易维护了。

除了对自己的代码保持敏感，还可以构造单元测试自身以让其变得更好。

3. 没有必须遵循的测试顺序

我们经常发现得为某一功能编写大量单元测试。大多数测试工具允许将测试划分为组（或套件）。我们应当避免包括那些在测试套件中依赖特定顺序工作的测试。单元测试在测试套件中的位置无关紧要。如果我们为 SPA 应用创建了一系列良好的单元测试，就应当能够以任意顺序运行它们，且实现一致的相同结果。例如在小费计算器的测试中，小费总数计算的测试应当生成同一结果，不管它是第一个运行还是最后一个运行，或者夹在所有其他测试中间的某处运行。

在小费计算器例子中，我们的测试套件可能首先包括小费总数计算，其次包括新账单总数计算，之后再包括小费等级评估。但如果出于某些考虑需要重新安排它

们的顺序，那么每个测试结果应该与它们之前顺序的测试结果相同。

4. 单元测试应当自依赖

好的单元测试应该能够重复一致的结果。确保此效果的唯一可行途径就是让测试独立于上下文，且不依赖外部依赖，以确保正确工作。

首先，单元测试应该能够重复，即使其脱离原先编写的环境运行在另一个环境里。比如小费计算器的单元测试，不管是运行在本地开发环境还是在专门机器上测试，结果都应该相同。

其次，单元测试不该依赖任何外部系统，甚至其他的测试。在这个简单示例项目中我们没有使用服务器端代码，但不妨假设一下我们会用到。假设小费等级数据的获取不通过硬编码实现，而是通过数据库获取。如果我们要创建一个单元测试以验证具体小费将生成对应等级，就不该在测试每次运行的时候都发起一个服务器端调用。请记住好的单元测试必须快速而可靠，有一致的结果。外部系统会影响测试，因为每次调用的响应是不同的，使得测试执行时间变得不可预期。依赖外部系统还会造成测试易受外部问题破坏，诸如网络问题或服务器宕机。此外，远程系统的任何改变，如数据改变或配置改变，都会导致不一致的结果。为了避免依赖外部系统，我们可以用 Mock 取代外部系统。Mock 是带有预编程期待的对象，可以用在单元测试中代替真实系统。例如，可以将 Mock 作为实时 Web 服务调用的替代品。其可以表现为系统调用，且能做出测试所需的响应。

最后，如果测试的主功能需要用到附加功能，可以通过 Stub 临时替代其他功能以保持主功能的独立性。Stub 是带有预定行为的临时替代品。其允许我们将单元测试聚焦在主功能上，且不用受其他代码行为的困扰。

5. 每个测试的关注点应该清晰易懂

好的单元测试还可以仅通过查看测试标题及注释就能快速明白其用意。随着时间的推移，开发者容易忘记代码的来龙去脉，因此扫一眼即能够明白单元测试的用途是很有帮助的。

下面看一个本章项目的例子。就小费总数计算中典型的单元测试，表 8.1 列出了测试套件及每个测试的标题，以及相应的注释。通过这些信息，不管是业务分析师，还是 QA 测试者、别的开发人员或业务负责人，测试目的都能够让人一看就懂。

表 8.1 单元测试的标题与注释清晰易懂

测试套件标题	测试标题	测试注释
小费计算套件	计算小费总数	10 美元按 15% 的小费率，将生成 1.50 美元小费

讨论完单元测试几个方面的内容，让我们来实践一下。在讨论 TDD 测试之前，

先通过一些传统单元测试概念来巩固基础。

8.3 传统的单元测试

单元测试的方式有好几种。最典型的方式是先写代码，然后围绕这些代码创建单元测试（如图 8.6 所示）。这种方式也相对简单并容易掌握。

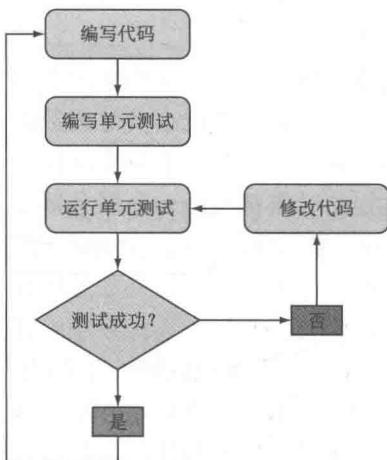


图 8.6 在典型的单元测试中，在代码创建后编写测试

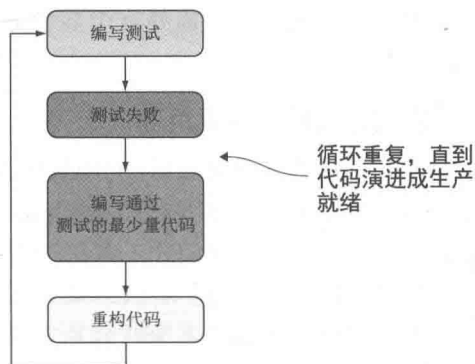
传统方式只是设计单元测试方法中的一种。一些开发者比较倾向于这种方式，因为他们更喜欢先写代码，之后再创建测试以确保设计是正确的。由于单元测试的目标是创建更好的软件，其他不同意见则认为传统的单元测试方式太过脱离了设计过程。出于比较的目的，我们会通过测试驱动开发（Test-Driven Development, TDD）的单元测试来与传统方式进行比较。

创建的测试，不管采用 TDD 还是别的方式，都定义了测试代码的行为。在 TDD 方式下，通过前置单元测试，将单元测试与设计过程更多地联系在一起。由于我们依赖编写良好的测试来辅助代码设计，因此就不得不预先深入理解需求。

刚开始的测试会失败，因为尚未编写逻辑代码。下一步是创建让测试通过所需的最少量代码。这将为新特性或增量开发一个基线。接下来，对代码不断进行重构，代码逐渐由最初的最小实现演进成良好设计、生产就绪的代码。每次重构之后，测试都要重新运行。这个过程能让你时刻做到胸有成竹，每次改进都不会造成不经意的破坏，也不会构建出不满足测试点的代码。图 8.7 阐述了这个循环过程。

由于传统单元测试比较简单直接，因此我们使用它为本章示例项目创建一些基本的单元测试。

为了对 SPA 应用进行单元测试，还需要选择一个 JavaScript 测试框架。着实有太多的选择，因此在后续章节我们也会讨论一些其他选择。但是，我们的初次选择是历久弥坚且最容易使用的 QUnit 框架。



8.3.1 QUnit 起步

首先，下载 QUnit：<http://qunitjs.com>。我们同时需要 JavaScript 及 CSS 文件。接下来，准备测试环境。我们需在项目目录中创建一个目录结构以容纳测试脚本。

图 8.7 在 TDD 中，在逻辑代码创建之前编写测试

1. 创建测试目录

如项目本身那样，测试脚本的目录结构也看个人口味。常见方式是设置一个跟应用结构类似的目录结构。测试目录通常跟应用源代码分隔开来。

为了能够使用多种 MV* 框架来进行测试，得用 Backbone.js、Knockout 与 AngularJS 来创建本章示例。图 8.8 展示了示例项目 Backbone.js 版本的目录结构，让你对此有个感性认识。在 Knockout 和 AngularJS 版本中也遵从同样的结构。该结构也可以应用到你使用的其他 MV* 框架上。

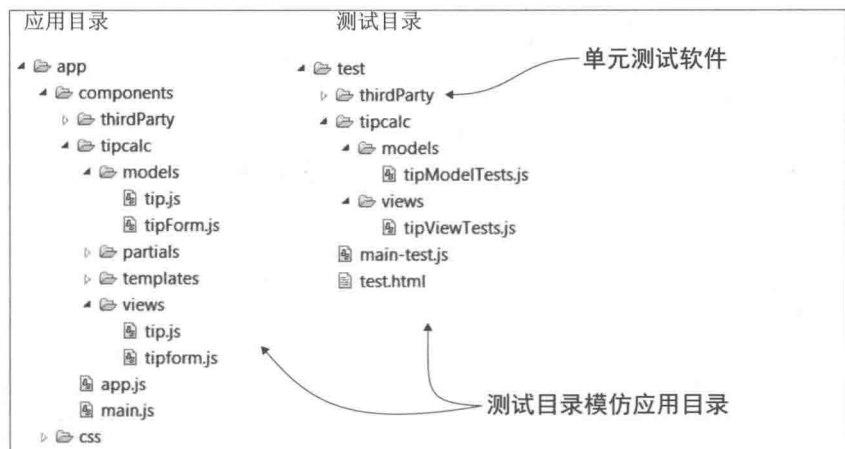


图 8.8 测试目录跟应用目录在结构上相近，但通常与应用代码分隔开来

创建好目录之后，需要有个显示测试输出的地儿。我们得创建一个新的 HTML 页面来做这件事。

2. 定义测试结果页面

在准备好测试目录之后，QUnit 还需要一个 HTML 页面来显示单元测试的结果。对我们的这个项目，创建一个 `test.html` 页面。可以直接在浏览器中打开该页面链接来查看它。实际上，我们并不希望应用程序提供测试页面链接。

清单 8.1 展示了这个测试结果页面的结构。在现实世界的解决方案里，我们更倾向于使用一个构建工具来收集所需内容。由于在下一章才会覆盖构建过程，因此我们直接在 HTML 页面中包含所有需要的文件（以及 AngularJS 相关文件）。在这里，将包含对测试脚本及源文件的引用。

清单 8.1 test.html——AngularJS 版本

```
<!DOCTYPE html>
<html>
<head>

<title>QUnit tests</title>

<link rel="stylesheet"
href="../../test/thirdParty/qunit/QUnit_Styles.css">

</head>
<body>

<h1 id="qunit-header">QUnit Test Suite</h1>
<h2 id="qunit-banner"></h2>
<div id="qunit-testrunner-toolbar"></div>
<h2 id="qunit-userAgent"></h2>
<ol id="qunit-tests"></ol>
<div id="qunit"></div>
<div id="qunit-fixture"></div>

<script src="../../app.js">
</script>
<script src="../../tipcalc/services/calculateSvc.js">
</script>
<script src="../../tipcalc/controllers/tipCalcCtrl.js">
</script>

<script src="../../thirdParty/qunit/QUnit_1_17_1.js">
</script>
<script src="../../thirdParty/angular.min.js">
```

包含 QUnit CSS 样式文件

QUnit 使用的 HTML 元素

包含应用文件

包含 QUnit 及 AngularJS

```

</script>

<script src="../services/calculateSvc_tests.js">
</script>
<script src="../controllers/tipCalcCtrl_tests.js">
</script>

</body>
</html>

```

包含测试脚本

Knockout 与 Backbone.js 版本使用 AMD 风格模块及 RequireJS。使用 RequireJS 时, HTML 页面跟这个页面几乎一致, 除了用一个引用 RequireJS 的 `<script>` 标签来替代其他相应的 `<script>` 标签 (如清单 8.2 所示)。该 `<script>` 标签中的 `data-main` 属性定义了主应用程序文件的位置。`main-test.js` 文件包含了到框架、应用源代码文件以及测试脚本位置的引用。

清单 8.2 test.html——Knockout and Backbone.js 版本

```

<!DOCTYPE html>
<html>
<head>
<title>QUnit tests</title>
<link rel="stylesheet"
href="../../test/thirdParty/qunit/QUnit_Styles.css">
</head>
<body>
  <h1 id="qunit-header">QUnit Test Suite</h1>
  <h2 id="qunit-banner"></h2>
  <div id="qunit-testrunner-toolbar"></div>
  <h2 id="qunit-userAgent"></h2>
  <ol id="qunit-tests"></ol>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>

  <script
    data-main="../../test/main-test.js"
    src="../../thirdParty/require.js">
  </script>
</body>
</html>

```

包含 RequireJS、指定配置文件位置

同时, 在使用 AMD 风格模块的时候, 单元测试的定义方式跟其他模块一样, 通过 `define()` 来定义。套件本身启动于 `require()` 内部, 跟项目一样。清单 8.3 展示了 Knockout 版本的 `require()` 定义。Backbone.js 版本也类似处理并遵循相同方式。如果需要参考完整代码, 可以在线下载。

在这里, 我们为每个模块创建了 `addTests()` 定制 API。`addTests()` 只负责执行 QUnit 测试函数, 其添加测试到 QUnit 框架要运行的测试列表。我们马上就能看到相关例子。

清单 8.3 main-test.js——Knockout 版本，使用 AMD 风格模块与 RequireJS

```
require(["QUnit",
  "utilTests/tipCalculatorUtilTests",
  "viewModelTests/tipViewModelTests" ],

function($,
  QUnit,
  tipCalculatorUtilTests,
  tipViewModelTests) {

  tipCalculatorUtilTests.addTests();
  tipViewModelTests.addTests();

  QUnit.load();
  QUnit.start();
}
);
```

确保依赖得以加载

显式添加 QUnit 要运行的所有测试

手动加载 / 启动 QUnit

注意 需要在 RequireJS 设置文件中将 QUnit 的 `autoStart` 设置为 `false`，这样 QUnit 的 `load()` 和 `start()` 函数在测试添加之后才能手动执行。

环境设置完毕，就可以开始编写测试脚本了。在下一节中，我们将讨论 QUnit 的断言及如何通过它们来创建单元测试。

8.3.2 创建第一个单元测试

示例项目的三个版本都执行两个计算：一个是计算小费总数，另一个计算新账单总数。此外，各个版本的逻辑中都对输入的小费百分比进行了诸如“Standard”或“Great!”这样的评级。我们尽量让各个版本的项目来得简单，尽管其间存在一些差异，而这些差异是由于采用不同框架而造成的。例如 AngularJS 版本项目，业务逻辑是在 AngularJS 模块的服务组件中；而对于 Knockout 版本，业务逻辑放在辅助 AMD 模块里；Backbone.js 版本中的业务逻辑则放在表单输入视图使用的模型中。

说了这么多，让我们卷起袖子开干吧！要创建我们的第一个单元测试，还需要理解 QUnit 中的断言。

1. 制造断言

我们将创建的单元测试，其背后的基本思想是要断言某些逻辑是真。当开始一个 QUnit 测试时，每个断言得以执行。QUnit 通过验证其断言来让各个单元测试成功还是失败。断言可以设计成任何你希望的样子。比如，其可以简单如验证每个特定对象是否存在，或者验证每个操作的结果。

QUnit 提供了各种类型的断言。完整列表可参考：<http://api.qunitjs.com/category/assert>。对于我们的单元测试，只用到一种类型的断言：

- `strictEqual()` ——该断言比较第一个和第二个参数是否相等。其第三个

参数用于注释。其使用严格相等操作符 ===。

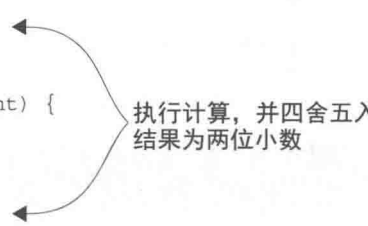
我们用 Knockout 版本中的计算功能来创建第一个测试。在该版本中，业务逻辑在一个标准的 AMD 模块里。并无特别设置需要我们处理。对于 Backbone.js 版本，计算逻辑则存于模型，而 AngularJS 版本存于服务。稍晚一些我们将讨论 MV* 对象测试。

2. 编写单元测试

我们的第一个单元测试将测试小费总数与新账单总数计算的输出。为测试脚本添加一些上下文代码，如清单 8.4 展示。请记住完整的项目源代码可以在线下载。

清单 8.4 计算小费总数和新账单总数的逻辑

```
function calcTipAmount(billTotal, tipPercent) {  
    var tipAmount =  
        Number(billTotal) * (Number(tipPercent) / 100);  
  
    return tipAmount.toFixed(2);  
}  
  
function calcNewBillTotal(billTotal, tipAmount) {  
    var newBillTotal  
    = Number(billTotal) + Number(tipAmount);  
  
    return newBillTotal.toFixed(2);  
}
```

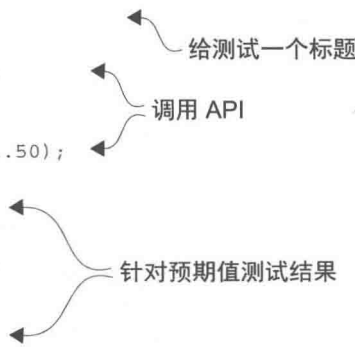


执行计算，并四舍五入
结果为两位小数

现在来编写单元测试，QUnit 使用的语法很简单。调用一个 test() 函数，给出测试标题及包含断言的函数，如清单 8.5 所示。

清单 8.5 用于验证 roundTipPercent() 函数的单元测试

```
test("Tip Calculations", function() {  
    var tipAmount  
    = tipCalculator.calcTipAmount(10.00, 15);  
  
    var newBillTotal  
    = tipCalculator.calcNewBillTotal(10.00, 1.50);  
  
    strictEqual(tipAmount,  
        "1.50",  
        "When the tip is 15% and the bill is $10,  
        the tip amount should be $1.50");  
  
    strictEqual(newBillTotal,  
        "11.50",  
        "When the bill is is $10 and the tip amount is $1.50,  
        the new bill total should be $11.50");  
});
```



给测试一个标题

调用 API

针对预期值测试结果

当测试执行时，应该能够在早先设置的测试 HTML 页面中看到测试报告。缺省地，每个测试不会显示测试注释。图 8.9 中点开了测试标题，以显示本次测试通过的断言。

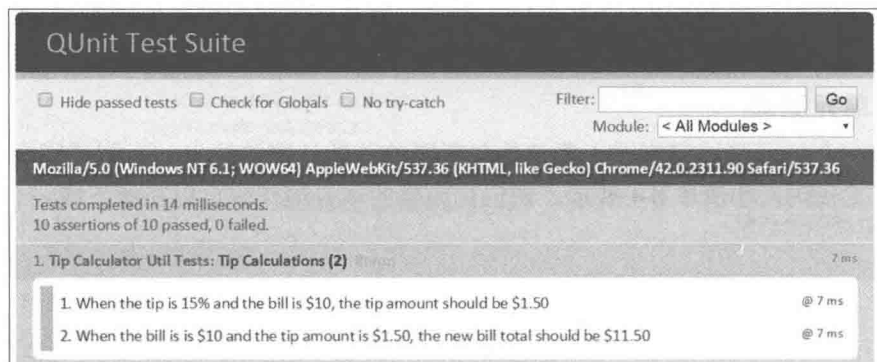


图 8.9 第一个单元测试运行完毕的 QUnit 测试报告

如果有一个测试失败了，报告的样子将如图 8.10 所示。QUnit 显示期望和结果各是什么，以及两者间的差异，还有执行过程的栈跟踪。

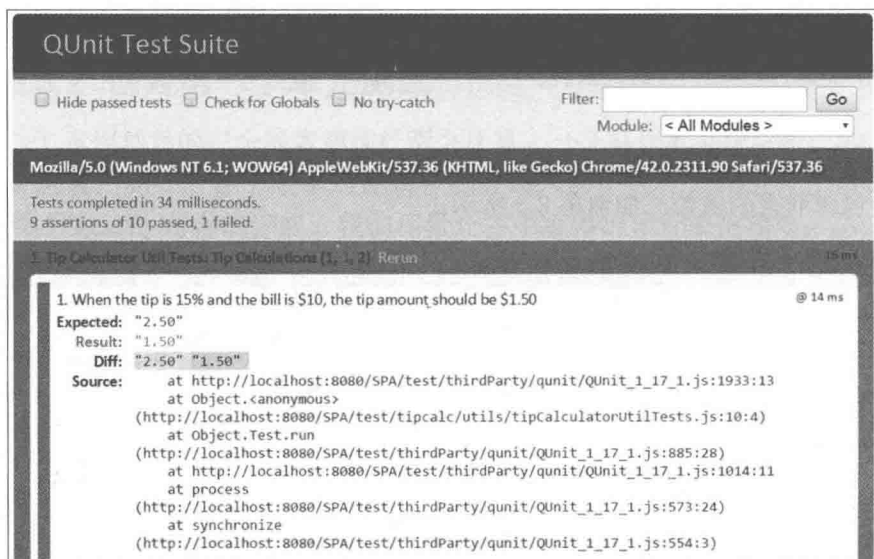


图 8.10 测试失败的 QUnit 测试报告

创建一个单元测试如此简单，接下来为余下的功能编写测试。然而，首先让我们将这些测试组到一个标题下。

3. 对测试进行分组

JUnit 的另一个优良特性是能够通过其 `module()` 方法来分组测试。在这里，我们将所有计算功能的测试组到一个标题下。这在后续为应用程序其他部分添加测试时将给我们提供便利。在 JUnit 中，将测试组进模块只需简单的一行代码：

```
module("Tip Calculator Util Tests");
```

清单 8.6 展示了小费计算功能模块的完整测试组合。

清单 8.6 在一个模块中组合计算功能的各个测试

```
module("Tip Calculator Util Tests");
```

测试计算功能

```
test("Tip Calculations", function() {  
    var tipAmount = tipCalculator.calcTipAmount(10.00, 15);  
    var newBillTotal = tipCalculator.calcNewBillTotal(10.00, 1.50);  
  
    strictEqual(tipAmount,  
        "1.50",  
        "When the tip is 15% and the bill is $10,  
        the tip amount should be $1.50");  
  
    strictEqual(newBillTotal,  
        "11.50", "When the bill is is $10 and the tip amount is $1.50,  
        the new bill total should be $11.50");  
});
```

定义模块

```
test("Tip Ratings", function() {  
    var rating = tipCalculator.getTipRating(5);  
    strictEqual(rating,  
        "So so",  
        "When the tip amount is below $15, the rating should be So So");  
  
    rating = tipCalculator.getTipRating(15);  
    strictEqual(rating,  
        "Standard",  
        "When the tip amount equals $15, the rating should be Standard");  
  
    rating = tipCalculator.getTipRating(20);  
    strictEqual(rating,  
        "Great!",  
        "When the tip amount is between $15 and $20 (inclusive),  
        the rating should be Great!");  
  
    rating = tipCalculator.getTipRating(50);  
    strictEqual(rating,  
        "Super!",  
        "When the tip amount is between $20 and $50 (inclusive),  
        the rating should be Super!");  
  
    rating = tipCalculator.getTipRating(60);  
    strictEqual(rating,  
        "WOW!",
```

测试小费等
级逻辑

```
"When the tip amount is greater than $50,  
the rating should be Wow!");  
});
```

图 8.11 展示了测试报告当前的样子，所有的测试都归到一个测试模块中。请记住每个测试最初都是折叠显示的，但可以展开来，如之前看到的那样，这样可以看到注释。

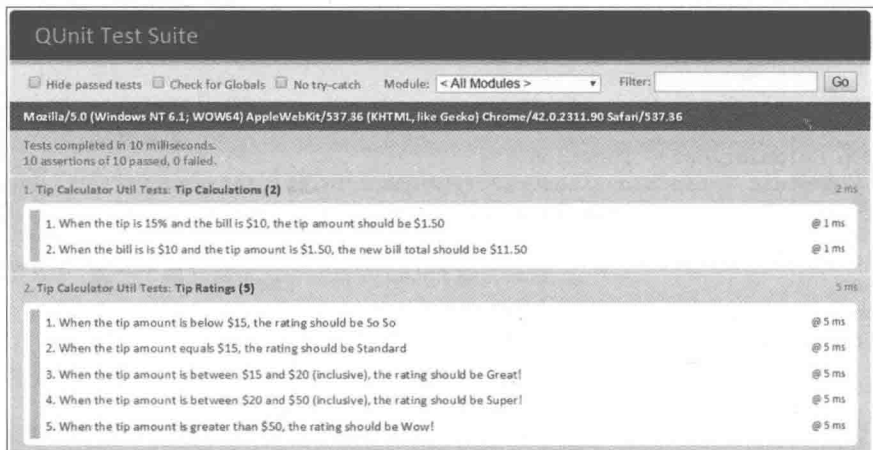


图 8.11 从 QUnit 测试报告中可以看到，测试组进了一个测试模块

MV* 对象中的业务相关代码也可以通过类似方式来测试。唯一问题是可能需要处理的大小大小设置项。

8.3.3 测试由 MV* 对象创建的代码

至于单元测试的编制之法，很不幸，不存在通吃一切的魔法公式。由于每种框架 / 库都是不同的，因此框架 / 库中的每个对象类型的用途也就不尽相同，这让测试运行起来的设置项数量也是变化的。然而设置通常以最低程度为标准。

在对 MV* 对象进行单元测试时也可以遵循一些简单的经验法则：

- 不要在测试之间共享对象——避免在测试之间共享 MV* 对象。开发人员的思维习惯于寻求对象重用并避免重复性代码。但在单元测试情况下，不要共享对象。
- 测试遵从 DIY 原则——如同处理生产代码，避免在每个测试里重复相同代码。应提取重复代码到单独函数中。针对需在每次测试之前运行的任何代码，许多测试框架也会提供一种设置方式。
- 参阅 MV* 框架文档，获取测试相关的特定指令——对于像 Backbone.js

或 Knockout 这样的框架来说, 仅需创建所用 MV* 对象的新实例。而像 AngularJS 这样的框架, 则需要考虑某些特定设置任务, 以能够正确测试组件。待会儿测试小费计算器的 AngularJS 版本时就会看到具体做法。

记住这里讲的, 我们开始对一些 MV* 对象进行单元测试。此处不会覆盖所有 MV* 对象。本示例项目各个版本的源代码, 以及对应完整的测试套件, 都提供了在线下载。然而, 我们要借助几个 MV* 对象来掌握对其进行单元测试的概念。

开始先来看看小费计算器 Backbone.js 版本的业务逻辑。我们要测试的计算类型与 Knockout 版本功能模块中的一样。但这次的逻辑代码放在 Backbone.js 的模型中, 如清单 8.7 所示。

清单 8.7 对 Backbone.js 模型中的业务逻辑进行单元测试

```
module("Tip Form Model Tests", {
  beforeEach: function() {
    this.tipFormModel = TipForm;
  }
});

test("Tip Calculations", function() {
  this.tipFormModel.calcTipAmount(10.00, 15);

  strictEqual(this.tipFormModel.get("tipAmount"),
    1.50,
    "When the tip is 15% and the bill is $10,
    the tip amount should be $1.50");

  this.tipFormModel.calcNewBillTotal(10.00, 1.50);

  strictEqual(this.tipFormModel.get("newBillTotal"),
    11.50, "When the bill is is $10 and the tip amount is
    $1.50, the new bill total should be $11.50");
});
```

TipForm 模块返回
一个模型新实例

使用该模型
执行计算

在该代码中, 请注意 beforeEach() 函数。JUnit 针对任何预测设置提供了一个可选的 beforeEach() 函数, 针对拆卸/清理工作提供了一个 afterEach() 函数。随着在 beforeEach() 函数中新实例的创建, 就可以开始测试该模型的逻辑了。

继续使用该模型并创建小费等级逻辑的测试, 如清单 8.8 所示。

清单 8.8 Backbone.js 模型中小费等级逻辑的单元测试

```
test("Tip Ratings", function() {
  this.tipFormModel.applyTipRating(5);
  strictEqual(this.tipFormModel.get("tipRating"),
    "So so",
    "When the tip amount is below $15, the rating should be So So");
});
```

改变每次断言
的等级值

```
this.tipFormModel.applyTipRating(15);
strictEqual(this.tipFormModel.get("tipRating"),
"Standard",
"When the tip amount equals $15, the rating should be Standard");

this.tipFormModel.applyTipRating(20);
strictEqual(this.tipFormModel.get("tipRating"),
"Great!",
"When the tip amount is between $15 and $20 (inclusive),
the rating should be Great!");

this.tipFormModel.applyTipRating(50);
strictEqual(this.tipFormModel.get("tipRating"),
"Super!", "When the tip amount is between $20 and $50 (inclusive),
the rating should be Super!");

this.tipFormModel.applyTipRating(60);
strictEqual(this.tipFormModel.get("tipRating"),
"WOW!", "When the tip amount is greater than $50,
the rating should be Wow!");
});
```

由于我们可以将测试组进 QUnit 模块里，因此 QUnit 模块分别提供一个针对模型中小费等级验证逻辑的测试，以及一个针对小费计算的测试。图 8.12 展示了此刻的测试套件执行情况。

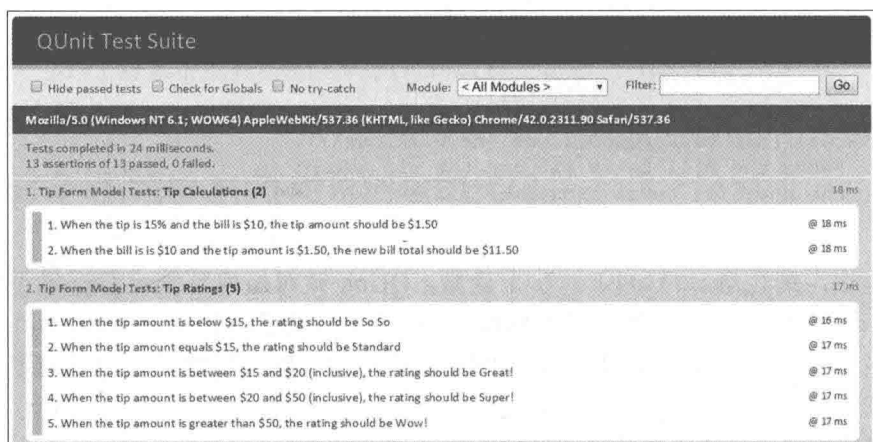


图 8.12 针对 Backbone.js 的 QUnit 单元测试

现在接着来看看 AngularJS 对象的单元测试。AngularJS MV* 框架文档中有一个设置测试脚本具体步骤的例子。在本示例中，我们要对 AngularJS 版本的计算功能进行单元测试。在 Backbone.js 版本里，计算功能放在一个 AMD 模块的模型里。而在 AngularJS 版本中，计算功能则放在一个 AngularJS 风格模块的服务组件里。

当应用运行时，AngularJS 的依赖注入机制会自动递上所需对象的实例。但由于在应用程序外部运行测试，因此必须手动调用依赖注入。存在几种手动注入 AngularJS 对象的方式，这里列出其中一种：

```
var injector = angular.injector([ 'ng', 'tipcalculator' ]);
```

我们需要为注入器传进作为依赖的 AngularJS 核心模块 ng 以及应用名称。然后在设置中可以使用注入器获取计算服务的实例：

```
beforeEach: function() {  
    this.CalcTestSvc = injector.get("calculateSvc");  
}
```

现在我们来查看一下计算服务的单元测试完整代码（如清单 8.9 所示）。

清单 8.9 AngularJS 服务的单元测试

```
var injector = angular.injector([ 'ng', 'tipcalculator' ]);  
  
module("Tip Service Tests", {  
    beforeEach: function() {  
        this.CalcTestSvc = injector.get("calculateSvc");  
    }  
});  
  
test("Tip Calculations", function() {  
    var tipAmount =  
        this.CalcTestSvc.calcTipAmount(10.00, 15);  
  
    var newBillTotal =  
        this.CalcTestSvc.calcNewBillTotal(10.00, 1.50);  
  
    strictEqual(tipAmount,  
        "1.50",  
        "When the tip is 15% and the bill is $10,  
        the tip amount should be $1.50");  
  
    strictEqual(newBillTotal,  
        "11.50",  
        "When the bill is is $10 and the tip amount is  
        $1.50, the new bill total should be $11.50");  
});  
  
test("Tip Ratings", function() {  
    var rating = this.CalcTestSvc.getTipRating(5);
```

获取注入器的实例

获取计算服务的实例

定义测试脚本

```

    strictEqual(rating,
    "So so",
    "When the tip amount is below $15,
    the rating should be So So");

    rating = this.CalcTestSvc.getTipRating(15);
    strictEqual(rating,
    "Standard",
    "When the tip amount equals $15,
    the rating should be Standard");

    rating = this.CalcTestSvc.getTipRating(20);
    strictEqual(rating,
    "Great!",
    "When the tip amount is between $15 and $20 (inclusive),
    the rating should be Great!");

    rating = this.CalcTestSvc.getTipRating(50);
    strictEqual(rating,
    "Super!",
    "When the tip amount is between $20 and $50 (inclusive),
    the rating should be Super!");

    rating = this.CalcTestSvc.getTipRating(60);
    strictEqual(rating,
    "WOW!", "When the tip amount is greater than $50,
    the rating should be Wow!");
  });

```

图 8.13 展示了测试运行结果，其说明了 AngularJS 注入器设置正常。

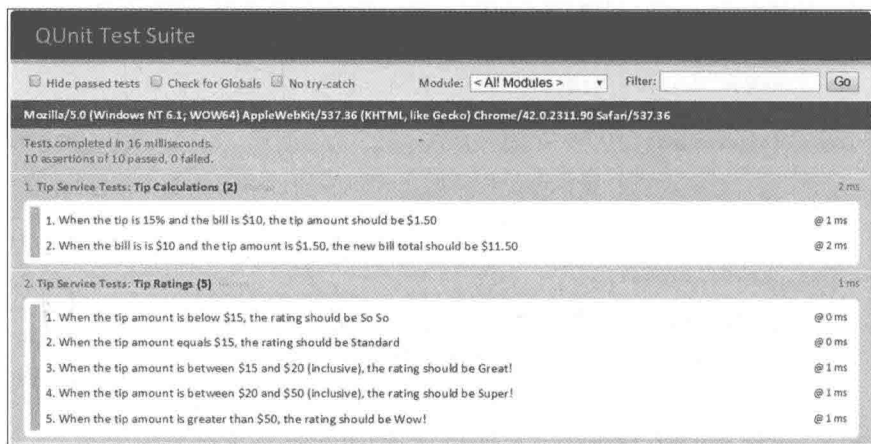


图 8.13 QUnit 测试报告展示了 AngularJS 服务的测试运行情况

通过 injector, 就可以在需要进行单元测试的任何 AngularJS 对象上运行测试。

8.3.4 测试对 DOM 所做的改变

有时还可能测试 SPA 应用中所做的 DOM 修改。在小费计算器示例项目中，我们在 UI 中通过视图的 `render()` 函数来映射模型状态。为了测试计算器的输出准确传到 DOM，QUnit 提供了一个 ID 为 `qunit-fixture` 的 DIV 元素。该元素被称为夹具 (Fixture)，可以用于添加 DOM 相关输出。在每次测试结束后 QUnit 自动清除该夹具元素，移除添加的任何内容。清单 8.10 展示了在视图上进行测试的设置与拆卸。

清单 8.10 在 Backbone.js 视图上执行夹具测试所需的设置 / 拆卸

```
module("Tip View Tests", {  
  beforeEach: function() {  
    $( "#qunit-fixture" ).append( "<section></section>" );  
  }  
});
```

手动添加元素；视图
将渲染其输出

通过在 UI 中执行单元测试，可以检查 DOM 元素、CSS 类或某一文本的存在。清单 8.11 展示了一个相关例子。

清单 8.11 通过 Backbone.js 视图对小费总数输出进行单元测试

```
test("Tip Display", function() {  
  TipForm.setBillTotal("10");  
  TipForm.roundTipPercent("15");  
  this.tipView = new TipView({  
    model : TipForm  
  });  
  this.tipView.render();  
  
  strictEqual(  
    $("section").children("p").eq(0).text(),  
    "Tip Amount: 1.50",  
    "When the tip is 15% for a $10 bill,  
    'Tip Amount: 1.50' should display"  
  );  
  
  strictEqual(  
    $("section").children("p").eq(1).text(),  
    "Total Amount: 11.50",  
    "When the tip is 15% for a $10 bill,  
    'Total Amount: 11.50' should display"  
  );  
  
  strictEqual(  
    $("section").children("p").eq(2).text(),
```

为视图模型的输出
设置种子值

使用 jQuery 从视图
选择我们期望的内
容作为输出

针对期望内容
测试输出

```
"Tip Rating: Standard",  
"When the tip is 15% for a $10 bill,  
'Tip Rating: Standard' should display"  
);  
});
```

我们已经覆盖了 QUnit 的大量内容。QUnit 是一个卓越的测试框架，甚至用整章的内容都无法说尽 QUnit 的各种功能。但即使你拥有了如 QUnit 这样强大的 JavaScript 测试框架，也可能仍想（或需要）针对特定任务使用别的框架。

8.3.5 混合使用其他测试框架

有时候并不是所有的测试都必须用一个 JavaScript 测试框架来覆盖，尽管用一个框架来测试往往也是可以的。常用的方式是使用一个测试工具框架来增强你的首要测试框架。为了有个直观理解，我们假设要保存用户输入的小费和账单总数以进行统计。这有点儿扯远了，但就当为了举例这么说吧。我们再次使用 Backbone.js 版本来阐述。

在本章开始我们了解了不要在单元测试中包含实时服务器端调用。那在这种情况下我们能够做什么呢？其实，JavaScript 框架 / 插件可以让我们很容易地模拟服务器端调用，且无须修改代码。一个叫 Sinon.js 的测试软件能够为我们提供帮助。

1. 使用 Sinon.js 来辅助 QUnit

本节的目标不是覆盖 Sinon.js 的方方面面，如同 QUnit，Sinon.js 很强大且具有大量优良特性。本节旨在说明使用其他测试框架扩充你的单元测试武器库其实并不难。我们将用 Sinon.js 来辅助 QUnit，以模拟服务器端调用这个我们假设出来的功能。下载该框架 (<http://sinonjs.org>) 并将其添加到代码结构中。可以在测试目录的 QUnit 目录旁边添加它（如图 8.14 所示）。

我们不用装模作样地创建服务器端代码，因为在这里只不过是阐述如何伪造服务器端调用。这将省去服务器端设置及代码编写的工作。相反地，想象一下我们已经有了一个运行中的服务器端调用，其用来保存小费计算器应用中的用户输入。为了模拟一个服务器端调用，需首先请求 Sinon.js 创建一个伪造的服务器端（如清单 8.12 所示）。



图 8.14 添加第二种 JavaScript 测试框架之后的测试目录

清单 8.12 伪造一个服务器端以模拟服务器端请求

```
module("Tip Form Model Tests", {  
  beforeEach: function() {  
    this.tipFormModel = TipForm;
```

```

    this.server = sinon.fakeServer.create();
  },
  afterEach: function() {
    this.server.restore();
  }
});

```

创建伪造的服务器端

执行清理并还原 XHR 构造器

可以添加该代码到我们的单元测试设置中。伪造的服务器端创建好之后，就可以通知 Sinon.js 你希望其如何做出响应。请记住我们不是测试该伪造的服务器端的输出，因为我们交由单元测试场景来决定伪造的服务器端如何响应。我们测试与服务器端调用相关联的业务逻辑。鉴于不同类型的服务器端响应，这些业务逻辑可以是判断调用是否已正确创建或发起，或者是应用程序的行为方式。清单 8.13 展示了模拟的服务器端，以及使用其响应的几个简单单元测试。

清单 8.13 在 Backbone.js 模型的单元测试中使用模拟服务器端

```

test("Save Request", function() {
  this.server.respondWith(
    [200,
      {"Content-Type": "text/html"},
      '{"save_status": "saved"}'
    ]
  );

  this.tipFormModel.save();

  this.server.respond();

  equal(this.tipFormModel.urlRoot,
    "/mockrequest",
    "When the model is saved,
    the URL should be /mockrequest");

  equal(this.server.requests.length, 1,
    "When the model is saved,
    the request length should be 1");

  equal(this.server.requests[0].requestBody,
    JSON.stringify(this.tipFormModel.attributes),
    "When the model is saved,
    the request body should equal the model attributes");
});

```

希望伪造的服务器端如何响应的状态

保存模型数据

向伪造的服务器端请求响应

确保 save() 正确地构造了 URL

测试重复调用

测试请求

Sinon.js 的众多特性使其变得强大而完备。如你所见，它也可以成为像 QUnit 这样的其他测试框架的完美搭档。由于现在在讨论其他测试框架，因此接下来会涉及你可能会感兴趣的其他一些选择。同时也包含了 QUnit，以便比较。

2. JavaScript 单元测试框架选择

尽管不是一份详尽的清单，但表 8.2 还是给出了几个流行的 JavaScript 测试框架，

它们都提供了单元测试特性。

表 8.2 提供了单元测试特性的流行 JavaScript 测试框架

名称	URL	评价
QUnit	http://qunitjs.com	成熟的框架，大量本书未涉及的特性，容易设置与使用
Mocha	http://mochajs.org	Mocha 具备大量特性，但需要你自己选择断言库，诸如 Unit.js（ http://unitjs.com ）或 Chai.js（ http://chaijs.com ）
Buster.js	http://busterjs.org	尽管在本书写作时仍是 beta 版，但该框架包含了大量先进特性
Jasmine	http://jasmine.github.io	另一个易于使用的框架，但其关注行为驱动开发（Behavior-Driven Development, BDD）

别惧怕探索其他框架（即便你已经有了一个喜欢的框架）。其他框架具有辅助你的首选框架的独特魅力。

8.4 挑战环节

又到了检验学习成果的挑战环节。请创建一个简单的调查类 SPA 应用程序，对某个特定主题，调查用户最喜爱的内容。主题可以是最佳影片、最佳食品等诸如此类的内容。输出应根据每次调查提交持续统计结果，显示每个调查项的排名情况。其可以是简单应用，也可以做得复杂些，一切按你所愿。之后，使用本章讨论的 JavaScript 单元测试框架，或是你自己喜欢的框架，为应用程序创建一个单元测试套件。

8.5 小结

本章快速介绍了客户端单元测试相关内容，我们来回顾一下：

- 在代码的各个部分执行单元测试。被测代码可能是一个完整的模块，但往往是独立函数。
- 单元测试提供了代码运作方式的示例。
- 单元测试应聚焦于软件行为与目的相关联的独立概念。
- 单元测试不该有任何特定顺序要求，应该自依赖，还应该易于理解。
- 单元测试的创建方式之一——在软件编写完毕后再创建测试。
- 在 TDD 中，测试则在逻辑代码编写前创建。

客户端任务自动化

本章内容

- 理解 Task Runner 概念及其意义
- 处理 Task Runner 任务
- 创建客户端构建过程

在软件开发中，经常会发现自己在整个开发周期里不断重复某些任务。这些任务包括针对给定语言执行特定开发步骤、运行测试以及创建构建过程等。为了实现这些任务的自动化，许多基于任务的自动化工具（或 *Task Runner*）纷纷涌现。其中，像 *make* 工具，已经存在了很长一段时间。其他的，如本章示例中用到的工具，相比之下还相当年轻。

Task Runner 工具形形色色。有些运行于特定平台，而其他则跨平台。有些针对特定构建语言，而其他的则是跨语言工具。许多这样的工具在传统上更关注构建过程。

在现代 Web 应用程序中，如我们的 SPA 应用程序，JavaScript 专业人员需要的不仅仅是一个构建工具。当代开发人员面临的现实相当繁杂，同时要求 *Task Runner* 能够自动化面向开发的大量任务。

新一代的 *Task Runner* 应运而生，同时其提倡将客户端开发置于重要位置，并

视客户端任务为“一等公民”。本章覆盖这些面向客户端的 Task Runner，以及通常由这些工具自动化的任务类型。我们会分别讨论开发阶段如何使用它们，以及如何使用它们来创建客户端构建过程。

对于本章示例项目，将使用基于 JavaScript 的 Task Runner——Gulp.js。为了拿一些代码来演示，我们将利用第 6 章创建的 SPA 示例应用。出于方便起见，Gulp.js 文件及第 6 章源代码已一并提供在本章可下载资源里了。

还有其他的 Task Runner 可供选择。本章后面会涉及其中的一些。然而，在进入工具选择之前，先来理解 Task Runner 能为我们做些什么。

9.1 Task Runner的常见用途

使用 Task Runner 时，要创建一套明确可重复的指令（或任务），用于描述有待自动化的动作的类型。如前所述，Task Runner 可以用来自动化构建过程，但其同时还能用于开发阶段。图 9.1 提供了我们可能执行的常见任务，覆盖开发阶段和客户端构建过程的创建。还可能有其他的一些或多或少的任务，这有赖于我们的需要，但这张图给了我们一个指引。

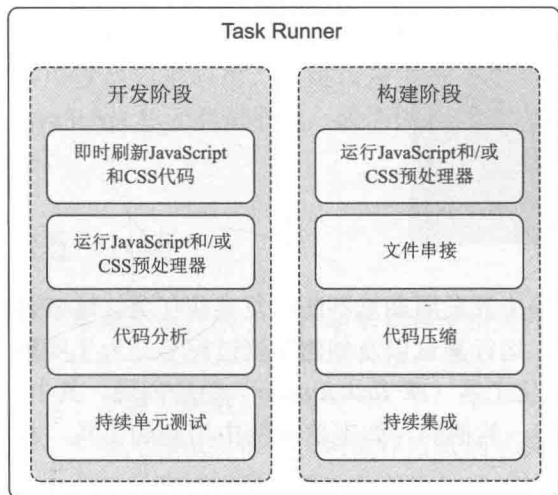


图 9.1 基于 JavaScript 的 Task Runner 的常见任务

后面在创建本章示例项目的时候，会实践其中的一些任务。现在来拆解一下这些任务并讨论其各自含义。

9.1.1 即时刷新浏览器

当创建一个 Web 应用程序（如 SPA 应用程序）时，你会发现一个无时无刻不

在做的特定开发任务——刷新浏览器。靠浏览器自身是无法获悉 CSS 或 JavaScript 文件何时会发生改变的。每次修改了文件，浏览器的显示都无法做到同步。为了让更改生效，就不得不重新加载页面。

为了自动化这个过程，诸如 LiveReload (<http://livereload.com>) 和 Browsersync (www.browsersync.io) 这样的工具可以用于在文件改变时自动更新浏览器。可以想象一下不用中断手头工作去手动加载浏览器能够节省多少时间。这些工具可以独立使用，但从 Task Runner 中调用它们使得我们能够在触发浏览器重新加载之前运行任意数量的任务。

9.1.2 自动化 JavaScript 和 CSS 的预处理过程

预处理器 (Preprocessor) 是用来创建已有语言新版本或定制版本的程序，其通过扩展或改变原始语法以包含新特性。此类型的编译预处理器执行过程有时候被称为 *transpiling*、*transcompiling* 或 *source-to-source compiling*¹。

Sass (<http://sass-lang.com>) 和 Less.js (<http://lesscss.org>) 是 CSS 流行预处理器的代表。它们能够让我们通过诸如变量、mix-in 以及内嵌规则等特性扩展 CSS。CoffeeScript (<http://coffeescript.org>) 和 LiveScript (<http://livescript.net>) 则是独立编程语言，通过预处理过程编译成 JavaScript 代码。

如果我们使用的语言需要一个预处理器，则可以通过 Task Runner 来自自动化预处理过程。

9.1.3 自动化 Linter 代码分析

JavaScript 与 CSS 的代码分析工具 (或 *Linter*) 可以检查代码错误及其他问题。这类工具也可以用于确保代码遵循一套标准的编码实践。对于 JavaScript, JSHint (<http://jshint.com>) 和 JavaScript Lint (www.javascriptlint.com) 是常见选择。而 CSS Lint (<http://csslint.net>) 则是 CSS 领域的流行工具。如同预处理器，在开发过程中使用工具进行代码分析的手工过程，也可以通过 Task Runner 来实现自动化。这不仅节省了时间，而且在编码时能够侦测到问题。

9.1.4 持续单元测试

对于某些开发风格，如 TDD，需要对代码进行持续单元测试。单元测试的一个麻烦就是不管在任何时候，只要想运行测试 (组)，那就不得不停下手头的工作。Test Runner 可以配制成观察文件改变并自动触发单元测试。此外，在任务中使用无头 Test Runner 还能够让我们无须来回切换浏览器就可以看到测试结果。

¹ 通俗而浅显地来讲就是从一种语言编译为另一种语言。——译者注

注意 术语无头 (*Headless*) 指的是无须借助图形化用户界面 (比如通过命令行方式) 就能够访问程序输出内容。

9.1.5 文件串接

术语串接 (*Concatenate*) 的意思是联合起来。当把众多文件联合进尽可能少的文件中, 就节省了带宽, 减少了网络流量, 并提升了用户体验, 让用户感觉应用程序加载变快了。

当创建 SPA 构建过程时, 一个常见的 Task Runner 执行指导步骤就是串接 JavaScript 文件到有意义的更少量文件中。这可能就意味着所有文件都联合进了单一文件或少量文件中。也可以按此方式处理 CSS 文件。

9.1.6 代码压缩

另一个在前端构建过程中执行的 Task Runner 步骤是压缩应用程序源代码。从源代码文件中移除应用程序运行时不需要的所有字符的过程被称为代码压缩 (*Minification*)。这些字符包括空格、换行符以及注释。通过改变源代码来减少变量及函数名称的长度 (有时候甚至减少到一个字母), 压缩工具也可以用于生成紧凑版本的代码文件。

9.1.7 持续集成

持续集成 (*Continuous integration*), CI, 是一种软件开发实践, 依托项目组成员全天频繁检入代码。CI 同时使用自动化构建过程至少每天一次地为应用程序执行代码构建。

大量开发团队使用集中式代码仓库以及集中式的 CI 服务器, 如 Jenkins (<http://jenkins-ci.org>), 来建立持续构建过程。使用如 Jenkins 这样的产品, 构建及相关测试可以在预定时间间隔上运行, 或者只要有新代码检入即执行。尽管 CI 实现细节超越了本书范围, 但我们的面向客户端构建及测试任务也能够为 CI 服务器所调用。此时 Task Runner 也同时需要在 CI 机器上安装。

了解完 Task Runner 的一些常见作用, 我们来讨论如何选择合适的 Task Runner 工具。

9.2 Task Runner选择

如本章开头所述, 存在一堆的 Task Runner 可供选择。哪怕将讨论范围缩小到只针对基于 JavaScript 的工具, 也有不断出现的各种选择。很不幸, 在做出选择时

没有明显的获胜者。最终还是要看个人口味。然而在选择时可以考虑一些常见差异。本书使用 Grunt.js 和 Gulp.js 来阐述，因为它们代表了相互竞争的 Task Runner 架构，同时也可以说是最流行的基于 JavaScript 的 Task Runner。

- 任务创建——一个要考虑的因素是，你喜欢通过配置还是喜欢以编程方式通过函数调用来描述任务？这种差异是醒目的，而 Task Runner 间的其他差异性可能相对细微。Grunt.js 是重度依赖配置来实现任务创建的 Task Runner 的代表，而像 Gulp.js 这样的 Task Runner，则通过代码创建任务。
- 处理过程是采用临时文件还是管道——选择 Task Runner 时的另一个差异点考虑是处理数据的方式。在某些 Task Runner 中，如 Grunt.js，创建临时文件用于中间处理过程。与此相反，像 Gulp.js 这样的工具则使用 I/O 流。这些流可以管道化在一起以编排任务流，而不需要依赖临时文件。流通常被认为是一个更快的处理任务的方式。
- 插件数量——并非所有但大部分的 Task Runner 允许通过插件方式扩展其基本功能。插件是附加模块，其扩展某个 Task Runner 的内建功能。插件多并不一定意味着这个工具就更好，但在创建 Task Runner 脚本时就有更多选择。在本书编写时，Grunt.js 的插件数量几乎是 Gulp.js 的三倍。

为了对某个具体的 Task Runner 有一个感性认识，我们应当花些时间好好浏览一下它们的网站。你可以通过网站获取最新信息，并下载工具进行验证。表 9.1 列出了一些至本书编写时基于 JavaScript 的 Task Runner 或构建框架。

表 9.1 一些基于 JavaScript 的 Task Runner 或构建工具

工具	URL
Grunt.js	http://gruntjs.com
Gulp.js	http://gulpjs.com
RequireJS Optimizer (r.js)	http://requirejs.org/docs/optimization.html
Mimosa	http://mimosa.io
Broccoli	https://github.com/broccolijs/broccoli
Brunch	http://brunch.io

了解过了 Task Runner 及其在开发与构建过程中的常见用途，我们来创建一些任务。接下来要实践本章的示例项目。然而，我们会保持一个平缓的节奏，以让你有时间熟悉所用 Task Runner 的语法及使用方法。

9.3 本章示例项目

不像之前章节所创建的 SPA 应用程序，在这个示例项目里我们将使用一个基

于 JavaScript 的 Task Runner 来阐述浏览器即时刷新和测试自动化。我们还将创建一个客户端构建脚本。如前所述，我们使用 Gulp.js 来构建该项目。Gulp.js 极易安装，且其通过编程来创建任务的方式也很容易理解。

Gulp.js 需要依托 Node.js 来运行。因此事先要安装 Node.js 与 Gulp.js。请参考附录 D 以了解具体的安装知识。在那里，所有所用插件的安装命令都已列出。

9.3.1 Gulp.js 介绍

如早先所述，Gulp.js 是一个 Task Runner。其主要功能是任务自动化。内建在 Gulp.js 中的基本功能共有四个，随着讨论的展开你会了解到更多的细节：

- `gulp.src`——指定要通过管道传送给插件处理的文件。
- `gulp.dest`——管道化过程的输出（写入）。
- `gulp.task`——创建任务。
- `gulp.watch`——监控文件，以在改变发生时能够做出反应（如调用一个任务）。

这是这些方法的高度概括。尽管在本章能够学到它们的用法，但也能通过 Gulp.js 的 API 文档来掌握其具体功能：<https://github.com/gulpjs/gulp/blob/master/docs/API.md>。

对于未在核心 API 中覆盖到的其他功能，就需要尝试到 Gulp.js 的插件库里寻找答案（<http://gulpjs.com/plugins>）。在接下来的一节中，我们将使用插件来执行本章早前介绍的某些开发及构建任务。

在 Gulp.js 中，任务通过函数调用而非配置来定义，当使用 Gulp.js 处理应用程序源文件时，数据通过 Node.js 的 I/O 流来处理。这些流可以连接起来（或管道化）以形成一个任务管道。任务中每个操作的输出可以连接到另一个操作的输入上。这使得通过流方式将多个过程形成一个菊花链以到达最终输出变得容易（如图 9.2 所示）。

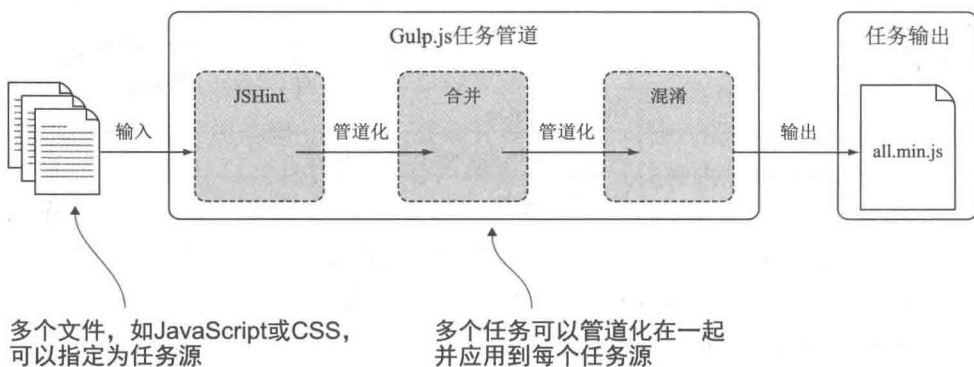


图 9.2 Gulp.js 能够将数据流管道化在一起进行处理

介绍完 Gulp.js，下面来创建我们的第一个任务。

9.3.2 创建第一个任务

要开始一个新任务，需要创建一个包含任务定义的文件。Gulp.js 将在运行命令的目录中自动查找一个名为 `gulpfile.js` 的文件。因此，先在本地项目目录中创建一个新的 `gulpfile.js` 空文件。可以使用任意目录。只需确保安装插件或执行任务时你在这个目录下。

接下来，添加一个任务到该 `gulpfile.js` 文件。在创建任何构建相关的任务之前，我们将创建一个简单的任务来浅尝一下 Gulp.js。Gulp.js 中的任务使用 JavaScript 编写并遵循以下语法：

```
gulp.task("task-name", [optionalArrayOfDependentTasks], function() {  
  // 任务操作  
});
```

对于我们的第一个任务，我们只输出经典的“Hello world”到控制台。没有任何依赖，因此这时候的 `gulpfile.js` 文件看起来像以下这样（参见清单 9.1）。

清单 9.1 通过任务输出“Hello world”

```
var gulp = require("gulp");  
  
gulp.task("say-hello", function() {  
  console.log("Hello world");  
});
```

← 导入 Gulp.js

← say-hello 是任务名称

请注意顶部的 `require()` 调用。由于运行时是 Node.js，且其使用 CommonJS 模块系统，因此这就是包含任何扩展模块引用的方式。这也包括 Gulp.js 自己。

Gulp 任务通过命令行调用。打开你的命令行工具，进到 `gulpfile.js` 文件所在目录。在其中执行 `gulp` 命令：

```
C:\chpt9\AngularJS-proj> gulp say-hello
```

输出结果类似以下所示：

```
[00:00:39] Using gulpfile C:\chpt9\AngularJS-proj\gulpfile.js  
[00:00:39] Starting 'say-hello'...  
Hello world  
[00:00:39] Finished 'say-hello' after 141 μs
```

注意 `μs` 符号表示 *microsecond*（微秒）。

接下来，我们来看看如何在 Gulp.js 中指定任务依赖。为了告知 Gulp.js 一个或多个任务应该在某个任务执行前运行，我们添加每个依赖任务的名称到一个任务名

称数组中。该数组是一个任务的第二个参数（可选）：

```
gulp.task("task-name", ["task-b", "task-c", "task-d"], function() { ... });
```

为了测试输出，需要创建第二个任务。这次，我们将输出“How are you?”到控制台，但在任务中指定之前的任务必须先执行（如清单 9.2 所示）。

清单 9.2 指定依赖任务

```
var gulp = require("gulp");
```

```
gulp.task("say-hello", function() {  
  console.log("Hello world");  
});
```

```
gulp.task("how-are-you", ["say-hello"], function() {  
  console.log("How are you?");  
});
```

现在是 how-are-you 的依赖

say-hello 任务仍能单独运行。但如果运行 how-are-you 任务，两个任务都会执行。首先是运行 say-hello，因为它是依赖，之后才运行 how-are-you 任务。新任务运行后的输出看起来类似以下所示：

```
C:\chpt9\AngularJS-proj> gulp how-are-you  
[00:13:37] Using gulpfile C:\chpt9\AngularJS-proj\gulpfile.js  
[00:13:37] Starting 'say-hello'...  
Hello world  
[00:13:37] Finished 'say-hello' after 144 μs  
[00:13:37] Starting 'how-are-you'...  
How are you?  
[00:13:37] Finished 'how-are-you' after 58 μs
```

现在我们初步尝试了一些简单的任务，接下来创建一个更有价值的任务。我们将从简单入手。

9.3.3 创建代码分析任务

代码分析，或 linting，是通过一个程序来分析代码中的错误或其他问题的过程。表 9.2 列出了我们将在本次任务中使用的插件，以及安装命令。可以使用 @latest 选项来获取最新版本。

表 9.2 JSHint 的 Gulp.js 插件，用于 JavaScript 代码分析

插件	安装
gulp-jshint www.npmjs.com/package/gulp-jshint	<code>npm install gulp-jshint@latest --save-dev</code>

为了使用该插件，需要先包含它。添加该调用到 `gulpfile.js` 文件的顶部：

```
var jshint = require("gulp-jshint");
```

现在可以在 `gulpfile.js` 文件中创建该任务了（如清单 9.3 所示）。在本示例中将任务命名为 `lint`，但你可以换成其他名称。

清单 9.3 使用 gulp-jshint 插件运行 Linter 任务

```
gulp.task("lint", function() {  
    return gulp.src(["App/components/**/*.js",  
        "!./App/components/thirdParty/**"])  
        .pipe(jshint())  
        .pipe(jshint.reporter('default'))  
    });
```

指定使用哪些文件

应用插件

输出报告

这段程序使用了 `gulp.src()` 来指定在分析任务中应该使用哪些文件。`gulp.src()` 函数使用一个字符串或一个数组。在这个字符串或数组中，可以使用完整文件路径或通配符（glob）模式来表述要包含或排除的文件与目录。`pipe()` 函数用于在处理过程中连接 I/O 流。

注意 通配符模式是使用模式和通配字符来表述文件或目录的一种方式。

Gulp.js 通过 `node-glob` 支持通配符模式。`node-glob` 站点上有其支持模式的完备入门内容：<https://github.com/isaacs/node-glob>。

使用通配符模式定义源文件为 `App/components` 目录或其所有子目录（`**`）中扩展名为 `.js` 的所有文件（`*.js`）。同时在这里还通过惊叹号（`!`）排除了第三方软件目录下的所有文件及目录。

你也许还注意到了 `jshint.reporter('default')` 调用。对于 `jshint` 插件，我们得通过其 `reporter()` 函数来打印分析结果到控制台。我们使用该插件缺省的结果报告器，但 `jshint` 插件也允许使用外部报告器。

随着任务定义完毕，就可以打开命令行，并输入以下命令：

```
C:\chpt9\AngularJS-proj> gulp lint
```

可以看到类似以下所示的任务运行输出内容：

```
C:\chpt9\AngularJS-proj> gulp lint
```

```
[10:09:38] Using gulpfile C:\chpt9\AngularJS-proj\gulpfile.js  
[10:09:38] Starting 'lint'...
```

```
C:\chpt9\AngularJS-proj\AppData\components\messaging\services\messageSvc.js:  
line 7, col 6, Unnecessary semicolon.
```

```
1 error
```

```
[10:09:38] Finished 'lint' after 112 ms
```

任务运行了所有指定文件，分析并查找代码问题。在这里，`Lint` 在某个函数的末尾找到了一个并不需要的额外分号。除了这个问题，`jshint` 插件还告知我们问题发生的文件及对应行列号。

现在来看一个非常酷的神奇功能，其将在开发期间给我们提供极大帮助——在编码期间如何让浏览器自动刷新。

9.3.4 创建浏览器刷新任务

在开发阶段，一件恼人却又无法避免的事情就是，修改代码之后要重新加载浏览器以观察最新效果。这个矛盾可以通过诸如 `LiveReload` 与 `Browsersync` 这样的程序来缓解。本示例使用 `Browsersync`（如表 9.3 所示），因为它很容易设置。`Browsersync` 不是 `Gulp.js` 插件；其是一个完全独立的产品，可以单独使用，也可以集成到你喜欢的 `Task Runner` 上。

对于本示例项目，我们将把 `Browsersync` 集成到 `Gulp.js` 上。同时也让刚才创建的代码分析任务成为浏览器刷新任务的依赖。该项目演示了如何使用 `Task Runner` 在浏览器重新加载发生之前对应用源代码文件进行必要的自动化处理。

表 9.3 浏览器自动刷新用到的 `Browsersync`

工具	安装
<code>Browsersync</code> www.browsersync.io	<code>npm install browser-sync@latest --save-dev</code>

使用 `Browsersync` 的好处是不需要浏览器添加任何特定插件或扩展。同时还能保持多个浏览器和多个设备同步，都在同一时间触发动作！我们不会讨论 `Browsersync` 的所有功能，你可以访问其站点观看介绍视频。

现在来创建任务。首先使用 `require()` 来导入 `Browsersync`，如之前导入 `Lint` 插件时那样：

```
var browserSync = require("browser-sync");
```

如前所述，我们可能希望在刷新前执行某些任务。为了将 `Browsersync` 与 `Task Runner` 集成在一起，我们将其包含进任务中，并指定一些需要在其之前执行的依赖任务：

```
gulp.task("reload", ["lint"], browserSync.reload);
```

`Browsersync` 内建了监控文件变化的能力，但既然它与 `Gulp.js` 集成在一起，因此在 `CSS` 与 `JavaScript` 文件发生变化时，我们还是使用 `gulp.watch` 来执行 `reload` 任务。

```
gulp.watch(["./App/css/*.css", "App/components/**/*.js"], ["reload"]);
```

gulp.watch 的第一个参数是字符串或是包含文件路径的数组、需用到的通配符。第二个参数是在观察到任意文件改变时应该调用的任务名称。在这里，如果任意 CSS 或 JavaScript 文件发生了改变，接着就会执行 reload 任务。

Browsersync 可以代理本地虚拟主机服务器，或启动自己的迷你服务器。在这里，我们并未使用服务器，因此将让 Browsersync 使用它自己的服务器，并通过该服务器来查看并刷新 SPA 页面。我们要做的就是通知 Browsersync 应用程序的基本路径在哪里：

```
browserSync({
  server: {
    baseDir: "./"
  }
});
```

现在来合成我们的任务（如清单 9.4 所示）。

清单 9.4 通过一个任务来自动刷新页面

```

reload 任务 → gulp.task("reload", ["lint"], browserSync.reload);

gulp.task("file-watch", ["lint"], function () {
  browserSync({
    server: {
      baseDir: "./"
    }
  });

  gulp.watch(
    ["/App/css/*.css", "App/components/**/*.js"],
    ["reload"]
  );
});
```

观察文件改变的任务

Browsersync 服务器设置

要监控的文件

文件改变发生时调用的任务

由于 reload 本身是一个任务，因此可以在其他任务中重用它。也可以用它来手动加载浏览器。写代码的时候需要在文件改变时自动刷新浏览器，因此请从命令行运行 file-watch 任务。运行情况类似以下所示：

```

C:\chpt9\AngularJS-proj> gulp file-watch
[11:27:07] Using gulpfile C:\chpt9\AngularJS-proj\gulpfile.js
[11:27:07] Starting 'lint'...
C:\chpt9\AngularJS-proj\App\components\messaging\services\messageSvc.js:
line 7, col 6, Unnecessary semicolon.

1 error
[11:27:08] Finished 'lint' after 105 ms

[11:27:08] Starting 'file-watch'...
[11:27:08] Finished 'file-watch' after 133 ms
[BS] Access URLs:
```

```
-----  
Local: http://localhost:3000  
External: http://192.168.1.68:3000  
-----  
UI: http://localhost:3001  
UI External: http://192.168.1.68:3001  
-----  
[BS] Serving files from: ./
```

因为我们有一个代码分析任务作为依赖，因此它首先执行，之后浏览器启动。我们没有指定 **Browsersync** 要自动打开的浏览器，因此它将打开系统的缺省浏览器。接下来，如果对列表中的任一文件进行修改，就会马上看到浏览器中发生的变化。同时在命令行还能看到以下类似的信息：

```
[11:33:35] Starting 'lint'...  
C:\chpt9\AngularJS-proj\App\components\messaging  
line 7, col 6, Unnecessary semicolon.  
  
1 error  
[11:33:35] Finished 'lint' after 76 ms  
  
[11:33:35] Starting 'reload'...  
[BS] Reloading Browsers...  
[11:33:35] Finished 'reload' after 1.54 ms
```

再次地，依赖任务先运行。同时也能看到浏览器刷新的环节。

现在，我们了解了代码变化时浏览器如何自动刷新的内容，接下来要讨论的另一个任务是开发阶段常见的。

9.3.5 自动化单元测试

如浏览器自动刷新任务那样，**Task Runner** 自动处理重复性任务节省了我们的时间与精力。现在讨论如何设计一个任务，为所写代码进行自动化单元测试。由于我们在第 8 章已经使用了 **QUnit**，因此本章依旧使用它。不过不打紧，可以使用你喜欢的任何 **JavaScript** 测试框架。

在第 8 章 **SPA** 应用的源文件处理代码中，有一个 **AngularJS** 模块用于计算二手电视游戏的折扣价格。现在先假设尚未编写此部分代码，因此请先移除该函数的代码：

```
function calculate(amt) {  
  
}
```

该函数接收一个总数，并用模块提供的一个标准折扣率乘以它。我们也已经为该函数创建了单元测试。相关代码可以从本章下载源代码的 `/test` 目录中获得。

针对自动化测试，需要一种方式，以显示 QUnit 测试结果。通常需要在浏览器打开测试结果的 HTML 页面，但我们并不希望在编码的时候也这样，我们只需要看到结果就成。

我们只需要一个能够运行测试、在命令行报告测试结果的程序，而不需要一个图形化用户界面。那么，我们将使用一个叫 `node-qunit-phantomjs` 的工具（如表 9.4 所示）。其在内部使用一个叫 PhantomJS 的无头浏览器。

表 9.4 在 PhantomJS 中运行 QUnit 的工具

工具	安装
<code>node-qunit-phantomjs</code> https://github.com/jonkemp/node-qunit-phantomjs	<code>npm install node-qunit-phantomjs@latest</code> <code>--save-dev</code>

该特定工具能够从命令行独立运行，但我们将把它包装进一个任务，并设置对 JavaScript 源文件的观察。这类似于前一节对 Browsersync 的处理。在正常情况下，我们还需要在 `gulpfile.js` 文件里包含该工具并将其作为一个依赖。

```
var qunitp = require("node-qunit-phantomjs");
```

这个工具很容易使用。传给它 HTML 页面（在浏览器中看 QUnit 测试结果的页面）的名称即可：

```
qunitp("./test/test.html");
```

现在来拼接我们的任务（如清单 9.5 所示）。

清单 9.5 文件改变时自动运行单元测试任务

```
gulp.task("unit-test", function() {  
  qunitp("./test/test.html");  
});  
gulp.task("watch-js", function() {  
  gulp.watch("App/components/**/*.js", ["unit-test"]);  
});
```

在命令行显示测试结果

当 JavaScript 文件发生改变时运行单元测试任务

我们先来手动运行一下 `unit-test` 任务。它失败了，因为你还没有实现函数逻辑。失败信息跟浏览器方式下看到的一样冗长。部分结果如下所示：

```
Testing ../../test/test.html  
Took 4 ms to run 1 tests. 0 passed, 1 failed.
```

```
Test failed: Pricing Service Tests: Pricing Calculations:  
Failed assertion: When the regular price is $10.00,  
the discounted price should be $4.00, expected: 4.00,  
but was: undefined
```

现在开始执行观察任务。当 watch-js 任务开始时，可以看到类似以下的命令行输出信息：

```
C:\chpt9\AngularJS-proj> gulp watch-js
[15:08:40] Using gulpfile C:\chpt9\AngularJS-proj\gulpfile.js
[15:08:40] Starting 'watch-js'...
[15:08:40] Finished 'watch-js' after 25 ms
```

该任务不会做什么事情，除非我们改变了其观察的文件内容。在该单元测试中，我们提供 10 美元作为电视游戏的价格。标准折扣率是 40%，因此正确的结果应该是 4 美元。若是此时我们为该函数添加上了能产生正确计算结果的代码块，则可以看看会发生什么。命令行此时的输出结果如下所示：

```
[15:17:19] Starting 'unit-test'...
[15:17:19] Finished 'unit-test' after 5.67 ms
Testing ..\..\test\test.html
Took 12 ms to run 1 tests. 1 passed, 0 failed.
```

我们已经设置了对所有 JavaScript 文件的观察，因此当我们编写的代码和单元测试越来越多的时候，测试结果也将随之输出到控制台。

了解完开发阶段 Task Runner 的使用，接下来用 Task Runner 创建客户端构建过程。每个项目的要求各不相同，因此我们在构建方面可能要面对或多或少的任务。在这里将通过几个常见构建任务来阐述客户端构建过程。

9.3.6 创建构建过程

在前面的内容中，我们了解了如何进行自动化测试。构建过程也可能会涉及自动化测试任务，因此此时需确保构建过程会包含测试任务。但还可能包括哪些任务呢？常见的构建任务也包括优化代码。我们首先来介绍优化 JavaScript 文件。

1. 优化 JavaScript 文件

JavaScript 代码文件的常用优化步骤包括串接以及压缩代码文件。我们使用 gulp-concat 插件来串接代码，使用 gulp-uglify 插件来压缩代码文件。同时还需要一个专门为 AngularJS 提供的 gulp-ng-annotate 插件，让 AngularJS 代码在压缩后还能正常工作。

注意 如果压缩任务配制成函数参数短名称方式，这会阻碍 AngularJS 依赖注入过程的正常工作。gulp-ng-annotate 插件使得所有应用组件得以正确注解，以防止该问题出现。

对于构建过程的输出，我们将添加一个新的 dist 目录（作为发布目录）到项目目录下。我们还使用了 del 包在每次新构建发生之前清理 dist 目录。表 9.5 列出了新

的插件 / 工具。

表 9.5 处理 JavaScript 以及清理构建发布目录的插件

插件 / 工具	安装
gulp-concat www.npmjs.com/package/gulp-concat	npm install gulp-concat@latest --save-dev
gulp-uglify www.npmjs.com/package/gulp-uglify	npm install gulp-uglify@latest --save-dev
gulp-ng-annotate www.npmjs.com/package/gulp-ng-annotate	npm install gulp-ng-annotate@latest --save-dev
del www.npmjs.com/package/del	npm install del@latest --save-dev

清单 9.6 展示了 `gulpfile.js` 文件中构建相关的任务。其有一个清理 `dist` 目录的任务，以及一个优化 JavaScript 源文件的任务。

清单 9.6 添加 JavaScript 优化任务

```
var gulp = require("gulp");
var del = require("del");
var concat = require("gulp-concat");
var uglify = require("gulp-uglify");
var ngAnnotate = require("gulp-ng-annotate");
```

```
gulp.task("clean", function(done) {
  console.log("cleaning dist dir");
  del(["dist/**/*"], done);
});
```

从 dist 目录中移除所有文件和目录

```
gulp.task("scripts", ["clean"], function() {
  console.log("processing scripts");

  return gulp.src(["App/components/**/*.js",
    "!./App/components/thirdParty/**"])
    .pipe(concat("all.min.js"))
    .pipe(ngAnnotate())
    .pipe(uglify())
    .pipe(gulp.dest("./dist/"));
});
```

设定串接文件名称

JavaScript 文件处理任务

指定已处理文件的存放目录

任务的输出是 `concat()` 函数中定义的 `all.min.js`。我们通过 `gulp.dest()` 函数指定 `dist` 目录作为任务输出的目标目录。

除了 `scripts` 任务，`gulpfile.js` 文件中还有另一个 `clean` 任务。我们通过 `clean` 任务在每次构建过程重新开始之前清理 `dist` 目录下的所有文件和目录。该任务使用事先安装好的 `del` 包。为了告知 `Gulp.js` 要在 `scripts` 任务之前运行 `clean`

任务，我们将 clean 任务声明为 scripts 任务的依赖。

2. 优化 CSS

到目前为止，我们只通过 Gulp.js 来优化 JavaScript 源代码。为了优化样式表，还得再次利用 npm 安装一个 gulp-minify-css 插件（如表 9.6 所示）。

表 9.6 压缩 CSS 的插件

插件	安装
gulp-minify-css www.npmjs.com/package/gulp-minify-css	npm install gulp-minify-css@latest --save-dev

清单 9.7 展示了向 gulpfile.js 文件构建脚本添加新任务。

清单 9.7 添加优化 CSS 的任务

```
var minifyCss = require("gulp-minify-css");

gulp.task("css", ["clean"], function() {
  console.log("processing css");

  return gulp.src("./App/css/*.css")
    .pipe(concat("styles.min.css"))
    .pipe(minifyCss())
    .pipe(gulp.dest("./dist/"));
});
```

← 串接并压缩所有样式表为 styles.min.css 文件

上述的这些代码负责优化脚本以及 CSS。当为构建过程进行优化操作时，还可以考虑优化 SPA 应用图像的任务。

3. 优化图像

对于图像，我们可以使用 gulp-imagemin 插件（如表 9.7 所示）。

表 9.7 减少图片大小的插件

插件	安装
gulp-imagemin www.npmjs.com/package/gulp-imagemin	npm install gulp-imagemin@latest --save-dev

清单 9.8 展示了为部署过程优化 SPA 应用图片文件的任务。

清单 9.8 添加缩小图片尺寸的任务

```
var imagemin = require("gulp-imagemin");

gulp.task("images", ["clean", "html"], function() {
  console.log("processing images");
```



```
return gulp.src("App/images/*.png")
    .pipe(imagemin())
    .pipe(gulp.dest("./dist/App/images"));

});
```

优化 .png 文件

构建过程至此已优化了源代码文件及图像，并将它们移到 `dist` 目录。要完成一个构建，还需要让整个应用程序达到可发布状态。我们要创建一个迁移剩余 SPA 应用文件的任务。

4. 迁移剩下的 SPA 应用文件

并非所有文件都需要优化，但它们仍是应用程序的一部分。这些文件包括了 HTML 文件及所有第三方代码库。它们都需要迁移到相同的 `dist` 目录中，跟那些优化过的代码和图片一道，用于发布。针对此任务，我们并不需要额外插件。可以通过 `gulp.src` 与 `gulp.dest` 来移动这些文件（如清单 9.9 所示）。

清单 9.9 添加移动剩余文件的任务

```
gulp.task("html", ["clean"], function(done) {
    console.log("copying over HTML");

    return gulp.src(["App/components/**/*.html",
        "./App/components/thirdParty/**"], {base: "./"})
        .pipe(gulp.dest("./dist/"));

});
```

移动所有 HTML 及第三方代码

处理完剩余文件，如果能够用 `Gulp.js` 来自动创建一个新版本的 `index.html`，其能够引用最新的优化文件，那该多好啊？嗯哼，是有专门做这事儿的插件！

5. 动态修改文件引用

有几个脚本替代插件可以通过引用优化版本来动态替换原有脚本和 CSS 的引用。对于我们的项目，将使用 `gulp-html-replace` 插件（如表 9.8 所示）。

表 9.8 该插件修改 `index.html` 中的引用，映射到新的构建文件上

插件	安装
<code>gulp-html-replace</code> www.npmjs.com/package/gulp-html-replace	<code>npm install gulp-html-replace@latest</code> <code>--save-dev</code>

这个插件的好处是，可以通过 HTML 风格的注释，在进行替换的源代码文件中注解位置。其通过开始及结束位置的注释来标识该区域为替换块，使用 `build:xxx` 这样的语法来指示新的替换块的开始。这种方式下，你就可以非常自然地标识任意数量的引用作为待替换目标。

作为开始，我们打开本地项目目录中的 `index.html` 文件，在替换位置周围添加

注释块。清单 9.10 展示了注解 CSS 替换块的情况，这里只展示了相关部分的代码，完整代码可下载获取。

清单 9.10 为 gulp-html-replace 插件注解 CSS 替换块

```
<!-- build:css -->
```

```
<link rel="stylesheet" href="App/css/default.css">
```

```
<!-- endbuild -->
```

在这里用优化过的引用替换所有的 CSS 原有引用

可以对 JavaScript 文件做同样的处理（如清单 9.11 所示）。

清单 9.11 为 gulp-html-replace 插件注解 JavaScript 替换块

```
<!-- build:js -->
```

```
<script src="App/components/app.js"></script>
```

```
<script src="App/components/data/appdata.js"></script>
```

```
<script src="App/components/useralerts/controllers/useralertsCtrl.js">
```

```
</script>
```

```
<script src="App/components/search/controllers/searchCtrl.js"></script>
```

```
<script
```

```
  src="App/components/productdisplay/controllers/productDisplayCtrl.js">
```

```
</script>
```

```
<script src="App/components/pricing/services/pricingSvc.js"></script>
```

```
<script src="App/components/messaging/services/messageSvc.js"></script>
```

```
<script src="App/components/search/services/searchSvc.js"></script>
```

```
<script src="App/components/productdisplay/services/productDisplaySvc.js">
```

```
</script>
```

```
<!-- endbuild -->
```

在这里用优化过的引用替换所有的 JavaScript 原有引用

标识了替换块，就可以创建 Gulp.js 任务来动态修改 index.html 文件，并将其移到 dist 目录下。你可以把这个作为 build 任务（如清单 9.12 所示）。

清单 9.12 修改 index.html 中文件引用的任务

```
gulp.task("build", ["clean", "scripts", "css", "images"], function() {  
  console.log("updating index");
```

```
  return gulp.src("index.html")
```

```
    .pipe(htmlreplace({
```

```
      "css": "styles.min.css",
```

```
      "js": "all.min.js"
```

```
    }));
```

```
  .pipe(gulp.dest("./dist/"));
```

```
});
```

替换掉所有注释块里的原有引用，换以新文件的引用

如果在任务运行之后检查 dist 目录下的 index.html 文件，将发现 CSS 引用已替换成了对优化过的单个 CSS 文件的引用：

```
<link rel="stylesheet" href="styles.min.css">
```

所有 JavaScript 原有引用也一样，都替换成了对优化过的单一 JavaScript 文件的引用：

```
<script src="all.min.js"></script>
```

最后，我们可以定义一个缺省任务。但这不是必需的。如果你打算创建缺省任务，需告知 Gulp.js 当 gulp 命令不带任务名称执行时应该怎么做。在这里，我们的缺省任务是触发构建过程。

```
gulp.task("default", ["build"]);
```

值得一提的是，在创建自己的构建过程时，涉及的任务可能与本章内容不尽相同。每个项目都有其自己的需求，每个开发团队也有其自己的选择。最终都得考虑什么是最适合你的！

9.4 挑战环节

现在进入章节挑战环节，看看你的功力到底提升了多少！挑一个本书的示例项目，或者你自己的也成，然后创建几个任务。确保其中的一些任务定义成依赖其他的任务。同时创造性地尝试一些插件。

9.5 小结

通过对本章的学习，我们介绍了 Task Runner 及客户端任务自动化。现在来快速回顾一下：

- Task Runner 是使重复性任务自动化的工具，不用它你就得手动执行这些任务。
- 在客户端开发阶段可以使用 Task Runner 来执行诸如浏览器即时刷新、代码分析、CSS/JavaScript 预处理、自动化测试等任务。
- Task Runner 同时也可用于构建过程的创建。构建过程是一系列的一致而重复步骤，为应用发布到生产环境做准备。
- 在一个基于 Web 的应用程序中，例如 SPA 应用程序，构建过程通常还包括 CSS/JavaScript 预处理、文件串接以及代码压缩等过程。
- 像 Gulp.js 或 Grunt.js 这样的 Task Runner 工具可以使用插件，这将大大增加该工具的任务执行范围。
- 存在大量的 Task Runner，能够以不同方式执行相同类型的任务。对 Task Runner 做出选择通常要看个人喜好及工具所处项目 / 环境的类型。

员工通讯录示例说明

本章内容

- Backbone.js 版本示例说明
- Knockout 版本示例说明
- AngularJS 版本示例说明

第 2 章讨论了员工通讯录应用的各种代码示例。通过学习，我们对 MV* 库 / 框架在 SPA 应用创建过程中所发挥的作用有了一个更好的理解。

这里将解释各个 MV* 框架版本的通讯录完整代码。如果你打算实践其中一个版本，那么这里的内容将帮助你事先理解整个应用。再来复习一遍我们的目标：

- 创建一个简单 SPA 应用，用于员工信息输入。
- 构建一个易用的 UI 界面，满足员工姓名、头衔、邮箱地址以及电话号码的输入要求。
- 跟踪条目列表的每个条目，整个屏幕分为两部分，左边是具体条目内容，右边是通讯录条目列表。
- 左边条目内容区域有两个按钮：一个是添加新条目按钮，另一个是清除表单按钮。

- 右边列表的每个条目旁边有一个按钮，用来从列表中移除条目。
- 每个条目字段旁边都有指示器，用来表示输入内容是否满足该字段要求（在用户输入时，每个指示器的内容会随之变化）。

在本书的其余部分，我们还掌握了一些更高阶的主题，如路由及服务器端事务处理。由于初次涉足 SPA 及 MV* 世界，我们暂先避开路由器概念以让事情保持简单。同时，在内存中保存员工数据。

回顾完目标，再来看看第 2 章首次见到的应用图示，如图 A.1 所示，这是最终产品图示。不同 MV* 框架开发出来的应用都具有这个相同的外观和行为。

这种设计虽然简单，但却涉及了视图、模板、绑定及模型等概念，同时还包括一些框架特定组件。该例子的一个好处就是，虽然我们并未与服务器端通信，但仍能在通讯录列表中进行 CRUD 操作。

如果用户输入非法数据，指示器状态将从“Required”变为“Invalid”。

单击X按钮将删除条目。

The screenshot shows a web application titled "Directory". On the left, there are five input fields: "First Name", "Last Name", "Title", "Phone", and "Email". Each field has a "Required" indicator to its right. Below these fields are "Add" and "Clear" buttons. On the right, there is a list of entries. Each entry is displayed with its details and a small "X" button for deletion. The entries shown are:

- Perez, Ana
Title: Sr. Director, HR
Phone: 555-555-1234
Email: Ana.Perez@someco.com
- Martin, Taylor
Title: Manager, HR
Phone: 555-555-1231
Email: Taylor.Martin@someco.com

The browser's address bar shows "localhost:8080/SPA/" and the page title is "Directory".

图 A.1 在线通讯录截图。用户在左边输入信息，合法条目在右边列表中呈现

A.1 CSS

我们将针对三个 MV* 版本都使用同一个 CSS 文件。清单 A.1 是缺省样式表。其只是进行必要的基本设置，如标头和主要区块的字体和样式。

清单 A.1 default.css

```
html, body {
    font-family: arial;
}

main {
    display: block; /* for IE */
    width: 800px;
    border: 1px solid #909092;
    background-color: #B7B8BA;
    border-radius: 10px;
    margin: 15px;
    padding: 5px 15px 15px 15px;
    box-sizing: border-box;
}

main > header {
    font-size: 25px;
    font-weight: bold;
    margin-bottom: 15px;
}
```

清单 A.2 定义了条目表单及条目集合的样式，在用户添加通讯录条目时会显示效果。

清单 A.2 entries.css

```
.entries {
    background-color: #D9D9D9;
    border: 1px solid #6D6D6D;
    border-radius: 10px;
    overflow: hidden;
    box-shadow: inset 0 0 3px 0 #6B6B6B;
    box-sizing: border-box;
}

.entries * {
    box-sizing: inherit;
}

.entries form {
    float: left;
    width: 45%;
```

```

padding: 5px 0 0 5px;
font-size: 18px;
}

.entries p {
margin: 12px 0;
}

.entries label {
font-style: italic;
line-height: 1.4em;
}

.entries input {
padding: 2px;
box-shadow: 0 0 8px rgba(0, 0, 0, 0.3);
border: 1px solid #999;
line-height: inherit;
font-size: inherit;
}

.entries .error-message {
float: right;
padding: 1px;
margin-right: 15px;
color: #DC3E5E;
font-size: 12px;
font-weight: bold;
}

.entries button {
cursor: pointer;
background-color: #EFEFEF;
}

.entries button::-moz-focus-inner {
padding: 0;
}

.entries form button {
box-shadow: 0px 2px 0px rgba(0, 0, 0, 0.5);
font-size: 16px;
border-radius: 5px;
padding: 7px 20px;
font-weight: bold;
border: none;
}

.entries .entry-list {
margin: 0 0 0 45%;
width: 55%;
height: 458px;
border-left: 1px solid #9F9F9F;

```



```
background-color: #EFEFEF;
box-shadow: inset 0 0 3px 0 #6B6B6B;
padding: 5px 0 0 0;
overflow-y: auto;
}

.entries .entry-list li {
  list-style-type: none;
  margin: 5px 15px 15px 15px;
  border-bottom: 1px solid #9F9F9F;
  font-weight: bold;
}

.entries .entry-list .remove-entry {
  float: right;
  border: 1px outset #D9D9D9;
  padding: 5px;
}
```

由于这是个小应用程序，因此我们在屏幕上为其设定了固定大小。这样也让我们在了解 MV* 概念的过程中能够聚焦某块区域的内容。

A.2 Backbone.js示例

先来看看 Backbone.js 示例代码。请记住 Backbone.js 是类 MVC/MVP 的框架。它的功能比 Knockout 丰富些，但远不及 AngularJS 那么丰富。然而其拥有开放的设计，允许我们通过代码或第三方插件扩展它以增加特性。

A.2.1 下载依赖

兵马未动粮草先行，我们得先下载所需工具。我们将用到几个第三方库 / 框架来扩展 Backbone.js 版示例应用。在这里不会深入覆盖所有的第三方库 / 框架，但会讨论如何使用它们（如表 A.1 所示）。

表 A.1 Backbone.js 版示例应用所需依赖

框架 / 库	URL	注释
Backbone.js	http://backbonejs.org	可以选择压缩（紧凑）版或开发（可读性好、格式良好）版。两者都能正常工作。开发版有助于理解 Backbone.js 的源代码。对于部署，请总是使用压缩版
jQuery	http://jquery.com	伟大的整体方案工具库！DOM 操作和事件处理的神器
Underscore.js	http://underscorejs.org	另一个工具库。与各种编程辅助器一起加载。将把它作为模板引擎使用

续表

框架 / 库	URL	注释
Handlebars.js	http://handlebarsjs.com	Handlebars 是另一个模板库。在本书编写之际，Handlebars 是 Backbone.js 使用上的要求，即使你不用它
RequireJS	http://requirejs.org	我们在第 3 章已经介绍过 RequireJS。现在，只需知道它用于异步 JavaScript 模块加载和依赖管理。通过它，可以使用 AMD 规范（也在第 3 章讨论过）创建代码。还可以使用两个 RequireJS 插件：domReady.js——确保在 JavaScript 代码作用于 DOM 之前 DOM 已准备就绪；text.js——异步下载模板。这两个插件都可以在 http://requirejs.org/docs/download.html 找到

A.2.2 目录结构

我们尽量保持应用目录结构在三个版本中的一致性。但在需要之处使用框架特定目录。图 A.2 展示了 Backbone.js 版本的应用目录结构。

该结构设计仍然遵循按特性设计的方式，在第 1 章我们曾介绍过。这里将特性作为目录（*components* 目录之下）。*components* 目录之下的特性目录可以随着应用的开发进展而不断增加。但在这里的特性目录中，其结构描绘了一个比较典型的 Backbone.js 布局。



图 A.2 Backbone.js 版本的应用目录结构

A.2.3 Shell 页面

在一个典型的 SPA 风格中,清单 A.3 所示的 index.html 欢迎页面几乎没什么内容。其主要的功能就是引用样式表、用 `<div>` 标签包含应用、通过 RequireJS 入口启动应用程序。

清单 A.3 index.html 文件

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="app/css/default.css">
  <link rel="stylesheet" href="app/css/entries.css">
  <!-- [if IE]>
    <script>
      document.createElement("main");
    </script>
  <![endif]-->
</head>
<body>
  <main />
  <script
    data-main="app/components/main.js"
    src="app/components/thirdParty/require.js">
  </script>
</body>
</html>
```

SPA 应用程序用到的样式表

IE 的 HTML5 修复

容纳应用程序的其余部分

RequireJS 配置文件

引用 RequireJS

RequireJS 动态地异步加载应用程序需要的所有依赖文件,包括模块和模板。同时它还确保这些文件在调用之前准备就绪。在这种方式下,不用担心脚本顺序及 `<script>` 标签的同步、阻塞特性。此时也不用对 RequireJS 感到不适应,其只是将 JavaScript 代码封装进优雅而独立的模块中。

A.2.4 main.js

本文件包含了 RequireJS 的配置项,如清单 A.4 所示。在其中我们定义了 base URL、RequireJS 能够下载到各个文件的对应路径,并为每个路径分配别名。

清单 A.4 main.js——RequireJS 配置

```
"use strict";
requirejs.config({
  baseUrl: "app/components",
  paths: {
    // 第三方库
    jquery: "thirdParty/jquery.min",
    domReady: "thirdParty/domReady",
    text: "thirdParty/text",
    backbone: "thirdParty/backbone-min",
    underscore: "thirdParty/underscore-min",
```

为我们的 SPA 应用程序定义一个 base URL

相对 base URL 的文件路径及其别名

```
// 应用程序
collections: "directory/collections",
models: "directory/models",
views: "directory/views",
templates: "directory/templates",
partials: "directory/partials"
}
});

require([ "app" ], function(app) {
    app.init();
});
```

SPA 应用程序的启动点，
app.js 作为依赖项

最后一行是 SPA 应用程序的启动点。RequireJS 确保在应用程序的 `init()` 函数调用之前会先加载 `app.js`。

A.2.5 app.js

`app.js` 文件如清单 A.5 所示，用 SPA 的初始 HTML 内容文件来加载用户的初始可视区域。`app.js` 文件不一定非得叫 `app.js`，这是对初始代码文件的典型命名约定。各种的应用准备工作及设置都在这个文件里面进行。

清单 A.5 app.js

```
"use strict";
define([ "jquery", "collections/entries",
        "views/entrylist", "views/directory",
        "text!partials/directoryContent.html",
        "domReady" ],
```

依赖名称匹配 main.js 中的
文件别名

用 text.js 插
件来下载非
JavaScript 代
码的文件

```
function($, Entries, EntryList, Directory,
        directoryHTML, domReady) {
```

针对各个依赖的参数
名称

确保 DOM 在开
始之前准备就
绪

```
function init() {
    domReady(function() {
        var entries = new Entries({});

        $("main").html(directoryHTML);

        var container = $("main .entries");

        new Directory(
            { el: container,
              collection: entries
            }
        );

        new EntryList(
            { el: container.find(".entry-list"),
              collection: entries
            }
        );
    });
});
```

使用 jQuery 来插入 HTML
片段（局部的）

```

    }

    return {
      init: init
    };
  }
};

```

RequireJS 开始先下载列表中的所有依赖。依赖都就位之后，就调用应用程序的 `init()` 函数。在这里，我们使用 jQuery 来插入初始 HTML 内容。当 DOM 准备好之后，就创建一个 Backbone.js 主视图的实例。

A.2.6 directoryContent.html

如清单 A.6 所示，我们的 HTML 文件看起来很干净。Backbone.js 使得我们无须内嵌 JavaScript 代码即可与其交互。

清单 A.6 directoryContent.html

```
<section class="entries">
```

```
  <form name="entryForm">
```

```
    <p>
```

```
      <label for="firstName">First Name:<br/>
```

```
        <input name="firstName" id="firstName"
```

```
          placeholder="First Name"/>
```

```
        <span class="error-message"></span>
```

```
      </label>
```

```
    </p>
```

```
    <p>
```

```
      <label for="lastName">Last Name:<br />
```

```
        <input name="lastName" id="lastName"
```

```
          placeholder="Last Name"/>
```

```
        <span class="error-message"></span>
```

```
      </label>
```

```
    </p>
```

```
    <p>
```

```
      <label for="title">Title:<br />
```

```
        <input name="title" id="title"
```

```
          placeholder="Title"/>
```

```
        <span class="error-message"></span>
```

```
      </label>
```

```
    </p>
```

```
    <p>
```

```
      <label for="phone">Phone:<br/>
```

```
        <input name="phone" id="phone"
```

```
          placeholder="555-555-5555"/>
```

```
        <span class="error-message"></span>
```

```
      </label>
```

```
    </p>
```

```
    <p>
```

```
      <label for="email">Email:<br />
```

应用程序的输入表单

SPAN 包含了每个输入的有效性检查标识

```

        <input name="email" id="email"
        placeholder="youremail@address.com"/>
        <span class="error-message"></span>
      </label>
    </p>
    <p>
      <button type="submit">Add</button>
      <button type="reset">Clear</button>
    </p>
  </form>

  <ul class="entry-list"></ul>
</section>

```

清除表单的按钮

添加条目的按钮

员工集合的条目列表

到目前为止一切进展顺利。HTML 文件看起来很不错，而且还干净，并未夹杂着 JavaScript 代码，体现了良好的可读性。在下一节中，我们将看到该内容对应的 Backbone.js 视图长什么样子。

Backbone.js 对象

在 Backbone.js 版本的 JavaScript 代码中，需要关注一点：通过 Backbone.js 创建的对象，其扩展了内建的 Backbone.js 对象。我们会首先定义原型，之后，在需要使用该原型类型的对象时，使用 `new` 关键字创建一个该原型的新实例。我们的视图扩展了 `Backbone.View`、列表扩展了 `Backbone.Collection`、模型扩展了 `Backbone.Model`。通过这么做，我们可以继承大量的内建特性。要了解完整的特性与示例，请参考 <http://backbonejs.org>。

A.2.7 directory.js 视图

也许你还记得讨论 Backbone.js 视图时的内容，视图以代码形式创建。我们通过一个模板及一个模型来创建可视内容（如清单 A.7 所示）。还请注意开始于下画线“_”的函数调用是 `Underscore.js` 的函数。该框架使得某些任务，特别是那些数组相关的操作，来得更简单。

清单 A.7 directory.js 视图

```

"use strict";
define([ "backbone", "models/employeeRecord" ],
  function(Backbone, EmployeeRecord) {

    var Directory = Backbone.View.extend({
      events : {
        "keyup :input" : "handleInputKeyup",
        "submit" : "handleClickAdd",
        "reset" : "handleClickReset"
      },
      initialize : function() {
        this.scheduleReset();
      },

```

视图事件

初始化时重置

```

handleInputKeyup : function() {
    this.buildAndValidateModel();
},
handleClickReset : function() {
    this.scheduleReset();
},
handleClickAdd : function(e) {
    e.preventDefault();

    var employee = this.buildAndValidateModel();

    this.collection.create(employee, {
        wait : true
    });

    if (!_contains(this.collection.models, employee))
    {
        return;
    }

    this.$("form").trigger("reset");

    scheduleReset : function() {

        this.$(":text:visible:first").focus();

        setTimeout(this.buildAndValidateModel.bind(this), 0);
    },
    buildAndValidateModel : function() {

        var fields = this.$("form").serializeArray();

        function compose(obj, field) {
            obj[field.name] = field.value;
            return obj;
        }

        var attrs = _.reduce(fields, compose, {});

        var model = new EmployeeRecord(attrs);

        this.$(".error-message").text("");

        if (model.isValid()) {
            return model;
        }

        .each(model.validationError,
            this.displayValidationMessage.bind(this)
        );

        return model;
    },

```

填充一个模型并验证
每次按键释放

获取一个新
模型

在集合中添加新模型,
除非存在错误

触发 reset 事件

重置表单, 并将焦点
放在第一个字段

reduce() 所需的回调
函数, 通过表单字段
数组来创建一个对象

通过 reduce() 构建模
型所需的属性对象

如果模型有效, 则返
回给集合

如果模型非法, 传递
每条错误给一个显示
函数

如果不成功, 则返回

添加表单字段到数组

新的模型实例

```

displayValidationMessage : function(err) {
    var selector =
        "[name='" + err.attr + "']+error-message";

    this.$(selector).text(err.error);
}

});

return Directory;
});

```

显示验证错误的函数

相比较我们用到的另外两个 MV* 框架, Backbone.js 少了不少魔法特性。就像清单 A.7 所示, 所有的工作都聚焦于在用户输入时管理表单的状态。

A.2.8 entrylist.js 视图

我们通过条目列表视图来与条目列表交互 (如清单 A.8 所示)。在这里我们声明了一个 Backbone.js 的 collection 来管理条目模型的集合。可以把 collection 当作一个管理模型内部数组的代理。

清单 A.8 entrylist.js 视图

```

"use strict";
define(["backbone", "views/employee"],
    function(Backbone, Employee) {

    var EntryList = Backbone.View.extend({
        initialize: function() {
            this.listenTo(this.collection, "add",
                this.renderEntry);
        },
        renderEntry: function(model, collection, options) {
            if(options.add) {

                var employee =
                    new Employee({ model: model });

                this.$el.append(employee.render().el);
            }
        }
    });

    return EntryList;
});

```

当添加了一个模型时, 调用 renderEntry()

新视图实例, 传进了模型

渲染视图

在条目列表的视图中, 可以监听 Backbone.js 的 collection, 只要添加了新模型, 即做出反应。在这里, 我们渲染添加的每个模型。

A.2.9 entries.js 集合

这个文件相当简单，如清单 A.9 所示。对于我们这个简单的示例应用，只需要定义一个集合，用它来存放 EmployeeRecord 模型。

清单 A.9 entries.js 集合

```
"use strict";
define([ "backbone", "models/employeeRecord" ],
  function(Backbone, EmployeeRecord) {

    var Entries = Backbone.Collection.extend({
      model: EmployeeRecord
    });

    return Entries;
  }
);
```

← 存放 EmployeeRecord
类型模型的集合

确保已包含了模型，并将其作为依赖。在前一节中，我们看到只要表单数据是有效的，对应模型就会添加到集合里。

A.2.10 employee.js 视图

我们通过 RequireJS 的 text 插件来动态获取并缓存员工模板，该模板包含了模型数据的占位符（如清单 A.10 所示）。由于在这个视图中处理模板，因此我们还要将 Underscore.js 作为模板引擎。

清单 A.10 employee.js 视图

```
"use strict";
define([ "underscore", "backbone",
  "text!templates/entrytemplate.html" ],

  function(_, Backbone, templateHTML) {

    var Employee = Backbone.View.extend({
      tagName: "li",
      template: _.template(templateHTML),
      render: function() {
        this.$el.html(this.template(this.model.toJSON()));
        return this;
      },
      events: {
        "click .remove-entry": "removeEntry"
      },
      removeEntry: function() {
        // 对应模型
        this.model.destroy();
        // 本视图
        this.remove();
      }
    });
```

← 使用的元素类型

编译 / 渲染模板

← 单击事件，调用
removeEntry

← 执行 DOM 和事
件的清理动作

```

});

return Employee;
}
);

```

在这个视图中，没有任何数据验证，那是对应模型的工作，视图只关乎用户所见及用户与 UI 间的交互。这里只涉及员工条目，该列表视图关注整个员工列表的 UI。在这儿，我们只处理模型数据获取并把它传递给模板。

其中还有一个事件，在用户单击“X”按钮时调用 `removeEntry` 函数。Backbone.js 在这里施展了点魔法。`destroy` 事件由集合观察，该事件会移除对模型的引用。为了完成清理工作，我们也调用 `remove()` 来移除该视图中所有相关的 DOM 元素，以及所有与该视图关联的事件绑定。

A.2.11 employeeRecord.js 模型

清单 A.11 展示了视图之下的员工模型。别被一大堆代码吓住了。它是有点儿啰唆，但只是模型需要的验证工作。

清单 A.11 employeeRecord.js 模型

```

"use strict";
define([ "jquery", "backbone" ], function($, Backbone) {

```

```

    var validators = {
      "required": [ {
        expr: /\S/,
        message: "Required"
      } ],
      "phone": [ {
        expr: /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/ ,
        message: "Invalid"
      } ],
      "email": [ {
        expr: /^[a-z0-9!#$%&'*\+\/=?^_`{|}~.-]+(?:\.[a-z0-9!#$%&'*\+\/=?^_`{|}~.-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?$/i,
        message: "Invalid"
      } ]
    };

```

← 用于验证的正则表达式

```

    function validateField(value, key) {

      var rules = validators["required"]
        .concat(validators[key] || []);

      var broken = _.find(rules, function(rule) {
        return !rule.expr.test(value);
      });

```

← 验证规则列表

← 如果存在，则找出第一条违反的规则

```

    }
  };

  return broken ?
  { "attr": key, "error": broken.message } : null;
}

var EmployeeRecord = Backbone.Model.extend({
  validate: function(attrs) {

    var validated = _.mapObject(attrs, validateField);

    var attrsInError = _.compact(_.values(validated));

    return attrsInError.length ? attrsInError : null;

  },
  sync: function(method, model, options) {

    options.success();

  }
});

return EmployeeRecord;
});

```

针对违反的规则，返回一条描述，或者返回 null

验证每个字段

返回错误数组或 null

由于未实现与服务器端的同步，因此直接调用成功

编制错误列表，扣去所有空值

显然，该模型中主要的事件是验证处理。Backbone.js 可以采取各种各样的验证方式。如果想在模型之外使用错误，唯一需要我们做的就是通过模型的 `validate` 方法返回这些错误。

A.2.12 entrytemplate.html

最后来看看员工数据展示模板，如清单 A.12 所示。该文件只包含内嵌模板标记的 HTML 代码片段。这些 HTML 代码片段也被称为 *Partial* 或 *Fragment*。

清单 A.12 entrytemplate.html

```

<button type="button" class="remove-entry">
  &#9587;
</button>

```

```

<%= lastName %>, <%= firstName %><br />

```

```

Title: <%= title %><br />

```

```

Phone: <%= phone %><br />

```

```

Email: <%= email %>

```

<%= %> 是 Underscore.js 语法，用来标识模型数据的插入位置

还记得第 2 章的内容吗？<%= %> 标识模型数据的插入位置。当模板引擎“编译”模板时，其只是把文本字符串传入一个可重用的函数以生成结果视图。当视图渲染

时，模型的 Property 值以其最终呈现状态插入视图。请记住 Backbone.js 对于模板引擎是开放的。我们在这儿使用 Underscore.js，但也可以使用其他模板引擎，另一个流行的选择是 Handlebars。

A.3 Knockout示例

第二个 POC（概念验证，Proof of Concept）使用 Knockout 创建。Knockout 遵循 MVVM。尽管 Knockout 擅长处理数据绑定，但我们仍得挑选其他框架/库（如 Durandal）来补充其他功能。Knockout 着力于其强项：绑定。对于那些不喜欢“黑匣子”式解决方案而更喜欢菜单点菜方式的开发者来说，Knockout 是一个完美选择。对于此 POC，我们会包含一些 Backbone.js 版本中也用到的附加辅助库。

A.3.1 下载依赖

表 A.2 列出了 Knockout 版本所需的依赖项。如果你已经在 Backbone.js 版本中下载过，那就不需要重复下载了，可以再次使用它。

表 A.2 Knockout 版示例应用所需依赖

框架 / 库	URL	注释
Knockout	http://knockoutjs.com	如同 Backbone.js，可以选择压缩（紧凑）版或开发（可读性好、格式良好）版。请记住，开发版只用于理解源代码和调试。对于部署，请总是使用压缩版
Knockout 验证插件	https://github.com/Knockout-Contrib/Knockout-Validation	由于 Knockout 并未提供任何内建的验证功能，因此我们要么自己写代码要么使用插件。如果你高兴，可以创建自己的代码。但对于这个示例，我们使用一个流行的插件，其易于使用，且着重于验证视图模型的数据
jQuery	http://jquery.com	伟大的整体方案工具库！DOM 操作和事件处理的神器
RequireJS	http://requirejs.org	我们在第 3 章讨论了 RequireJS。现在，只需知道它用于异步 JavaScript 模块加载和依赖管理。通过它，可以使用 AMD 规范（也在第 3 章讨论过）创建代码。还可以使用两个 RequireJS 插件：domReady.js——确保在 JavaScript 代码作用于 DOM 之前 DOM 已准备就绪；text.js——异步下载模板。这两个插件都可以在 http://requirejs.org/docs/download.html 找到

A.3.2 目录结构

如同 Backbone.js 示例那样，我们尽量保持应用目录结构在三个版本中的一致性，但在需要之处使用框架特定目录（如图 A.3 所示）。

该结构设计再次遵循按特性设计的方式，在第1章曾介绍过。这里仍将特性作为目录。其他的结构代表了比较典型的 Knockout 布局。



图 A.3 Knockout 版本的应用目录结构

A.3.3 Shell 页面

在一个典型的 SPA 风格中，如清单 A.13 所示的 index.html 欢迎页面几乎没什么内容。其主要的功能就是引用样式表，用 `<div>` 标签包含应用，通过 RequireJS 入口启动应用程序。

清单 A.13 index.html 文件

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="app/css/default.css">
  <link rel="stylesheet" href="app/css/entries.css">
  <!-- [if IE]>
  <script>
    document.createElement("main");
  </script>
  <![endif]-->
</head>
<body>
  <main />
  <script
    data-main="app/components/main.js"
    src="app/components/thirdParty/require.js">
  </script>
</body>
</html>
```

SPA 应用程序用
到的样式表

IE 的 HTML5
修复

RequireJS
配置文件

引用 RequireJS

如同 Backbone.js 那样，这里也使用 RequireJS 管理依赖。

A.3.4 main.js

本文件提供 RequireJS 配置项，如清单 A.14 所示。在其中我们定义了 base URL、RequireJS 能够下载到各个文件的对应路径，并为每个路径分配别名。

清单 A.14 main.js——RequireJS 配置

```
"use strict";
requirejs.config({
  baseUrl: "app/components",
  paths: {
    // 第三方库
    jquery: "thirdParty/jquery.min",
    domReady: "thirdParty/domReady",
    text: "thirdParty/text",
    knockout: "thirdParty/knockout.min",
    knockout_validation: "thirdParty/knockout-validation.min",

    // 应用程序
    viewmodels: "directory/viewmodels",
    templates: "directory/templates",
    partials: "directory/partials",
  },
  shim: {
    "knockout_validation": {
      "deps": ["knockout"]
    }
  }
});

require([ "app" ], function(app) {
  app.init();
});
```

SPA 应用程序的 base URL

相对 base URL 的文件路径及其别名

声明插件，需加载 Knockout

SPA 应用程序的启动点及依赖项

shim 块内容为 RequireJS 专用，以帮助非 AMD 模块能够与 AMD 模块和谐共处。其还能用于定义其他加载 Property，比如依赖。最后一行是 SPA 应用程序的启动点。RequireJS 确保 app.js 在 init() 函数调用之前加载。

A.3.5 app.js

app.js 文件如清单 A.15 所示，用 SPA 的初始 HTML 内容文件来加载用户的初始可视区域。app.js 文件不一定非得叫 app.js，但这是对初始代码文件的典型命名约定。各种的应用准备工作及设置都在这个文件里面进行。

清单 A.15 app.js

```
"use strict";
define([ "jquery",
  "text!partials/directoryContent.html",
  "viewmodels/directory", "domReady" ],
```

text.js 用于下载非 JavaScript 文件

```
function($, directoryHTML, directoryViewModel, domReady) {

    function init() {
        $("main").html(directoryHTML);

        domReady(function() {
            directoryViewModel.init();
        });
    }

    return {
        init: init
    };
}

);
```

使用 jQuery 来插入
下载的 HTML 片段
(局部的)

RequireJS 开始先下载列表中的所有依赖。依赖都就位之后，就调用应用程序的 `init()` 函数。在这里，我们使用 jQuery 来插入初始 HTML 内容。当 DOM 准备好之后，就在视图模型的 JavaScript 模型内部调用 `init()` 函数。

由于我们正在创建单例方式的 RequireJS 模块（只有一个实例），因此创建一个不会反复调用的函数，作为将调用应用到视图模型绑定的理想场所。该函数不用非得叫 `init()`，只是我们这么称呼它。

请记住，针对所有给定的 DOM 元素，我们希望只应用一次绑定。不像 Backbone.js，其模板根据每个新数据集而重建，视图模型通常持久保持。让调用反复应用到绑定将导致内存泄漏，因为我们通过每次调用复制了事件处理。

提示 每个子树插入到 DOM 只调用一次 `applyBindings()` 或 `applyBindingsWithValidation()`，将避免重复事件处理。

A.3.6 directoryContent.html

我们的 HTML 文件，如清单 A.16 所示，并未混杂任何 JavaScript 代码。但你也能够看到，其比 Backbone.js 版的文件稍微冗长了些。我们的生活处处存在着权衡：在 Knockout 中，JavaScript 可能显得不是那么重要，而添加定制 Knockout 属性（其被称为声明式绑定——declarative bindings）使得 HTML 的位置显得更突出。

一些开发者喜欢干净的代码，如果代价仅仅是多了一点冗长 HTML 的话；而另一些人则不喜欢往 HTML 中添加框架提供的属性。这都是个人主观意愿。但话又说回来，这也说明了像这样做一些概念验证工作为什么是一个好主意，这样能够让你通过一些首选方案获得实践经验，并帮助你判断哪个最适合自己。

清单 A.16 directoryContent.html

```
<header>Directory</header>

<section class="entries" id="directoryContent">
  <form name="entryForm">
```

应用程序输入表单

```

<p>
  <label for="firstName">First Name:<br/>

    <input id="firstName" type="text"
      name="firstName" placeholder="First Name"
      data-bind="hasFocus: isFocused,
        value: entry.firstName, valueUpdate: 'afterkeydown'" />

    <span class="error-message"
      data-bind="validationMessage: entry.firstName">
    </span>
  </label>
</p>
<p>
  <label for="firstName">Last Name:<br/>
    <input id="firstName" type="text"
      name="firstName" placeholder="Last Name"
      data-bind="value: entry.lastName,
        valueUpdate: 'afterkeydown'" />

    <span class="error-message"
      data-bind="validationMessage: entry.lastName">
    </span>
  </label>
</p>
<p>
  <label for="title">title:<br/>
    <input id="title" type="text"
      name="title" placeholder="Title"
      data-bind="value: entry.title,
        valueUpdate: 'afterkeydown'" />

    <span class="error-message"
      data-bind="validationMessage: entry.title">
    </span>
  </label>
</p>
<p>
  <label for="title">Phone:<br/>
    <input id="phone" type="text"
      name="phone" placeholder="555-555-5555"
      data-bind="value: entry.phone,
        valueUpdate: 'afterkeydown'" />

    <span class="error-message"
      data-bind="validationMessage: entry.phone">
    </span>
  </label>
</p>
<p>
  <label for="email">Phone:<br/>
    <input id="email" type="text"
      name="email" placeholder="youremail@address.com"
      data-bind="value: entry.email, valueUpdate: 'afterkeydown'" />
    <span class="error-message"

```

绑定视图模型数据和表单字段

绑定验证插件的错误消息


```

        data-bind="validationMessage: entry.email">
    </span>
</label>
</p>
<p>
    <button id="add"
        data-bind="click: addEntry,
            enable: isValidForm" >Add</button>

    <button id="clear"
        data-bind="click: clearForm">Clear</button>

</p>
</form>

<ul class="entry-list" id="entryList"
    data-bind="foreach: entries">
</ul>
</section>

```

添加新条目到列表

清除表单

HTML 代码看起来会比其实际来得更冗长一点，因为代码要适配书本宽度。你在实践相关源代码的时候，可以随时按自己喜欢的方式调整代码格式。

A.3.7 directory.js

在该概念验证中，包含了视图模型的模块，其代码量是最多的（如清单 A.17 所示）。并非所有的视图模型都像这样。一些视图模型只处理数据，因此可能比较小；而另一些带有大量逻辑，可能就比较大。

清单 A.17 directory.js

```

define([ "jquery", "knockout",
    "text!templates/entrytemplate.html", "domReady",
    "knockout_validation" ],

function($, ko, entryHTML, domReady, validation) {

    var emptyString = "";
    var astMsg = "Required";
    var invMsg = "Invalid";

    var emailRegX = /^[a-z0-9!#$%&'*/=\?^_`{|}~]+(?:
        :\. [a-z0-9!#$%&'*/=\?^_`{|}~]+)*@(?:[a-z0-9](?:
        [a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*
        [a-z0-9])?\$/i;

    var phoneRegX = /(?:\d{3}|\(\d{3}\))([-\./])\d{3}\1\d{4}/;

    var directoryViewModel= function() {
        var self = this;
        self.entry = {
            firstName : ko.observable().extend({
                required : {

```

验证用的正则表达式

视图模型观察对象使用验证插件扩展器

视图模型起点

```

        params : true,
        message : astMsg
    }
}),
lastName : ko.observable().extend({
    required : {
        params : true,
        message : astMsg
    }
}),
title : ko.observable().extend({
    required : {
        params : true,
        message : astMsg
    }
}),
phone : ko.observable().extend({
    customRegEx : {
        regX : phoneRegX,
        blankMsg : astMsg,
        invalidMsg : invMsg
    }
}),
email : ko.observable().extend({
    customRegEx : {
        regX : emailRegX,
        blankMsg : astMsg,
        invalidMsg : invMsg
    }
}),
});

```

为邮箱地址与电话号
码定制扩展器

```
self.entries = ko.observableArray();
```

```
self.isFocused = ko.observable(false);
```

```
self.addEntry = function(e) {
```

```

    var newEntry = {
        firstName : self.entry.firstName(),
        lastName : self.entry.lastName(),
        title : self.entry.title(),
        phone : self.entry.phone(),
        email : self.entry.email()
    };

```

```
self.entries.push(newEntry);
```

```
self.clearForm();
```

```
self.isFocused(true);
```

```
};
```

```

self.isValidForm = ko.computed(function() {
    if (self.entry.firstName.isValid()
        && self.entry.lastName.isValid()
        && self.entry.title.isValid()

```

条目列表的观察对象
数组

添加新条目到
列表中去

用于验证的 computed
观察对象

```

        && self.entry.phone.isValid()
        && self.entry.email.isValid()) {
            return true;
        } else {
            return false;
        }
    }, this);

    self.removeEntry = function(entry) {
        self.entries.remove(entry);
    };

    self.clearForm = function() {
        self.entry.firstName(emptyString);
        self.entry.lastName(emptyString);
        self.entry.title(emptyString);
        self.entry.phone(emptyString);
        self.entry.email(emptyString);
    };
}

function getValidationConfig() {
    return {
        insertMessages : false
    };
}

function createCustomValidationRule() {
    ko.validation.rules["customRegEx"] = {
        validator : function(userInput, ruleObj) {
            if (!userInput || userInput.length == 0) {
                this.message = ruleObj.blankMsg;
                return false;
            }
            if (!ruleObj.regX.test(userInput)) {
                this.message = ruleObj.invalidMsg;
                return false;
            }
            return true;
        },
    };
}

ko.validation.registerExtenders();

function init() {
    $("#entryList").html(entryHTML);

    createCustomValidationRule();

    domReady(function() {
        ko.applyBindingsWithValidation(
            directoryViewModel,
            $("#directoryContent")[0],

```

移除一个条目

清除表单条目

使得验证插件不会自动添加验证文本

配置定制的扩展器

注册扩展器

添加模板

通过扩展器添加定制验证

应用绑定

```
        getValidationConfig());  
    });  
}  
;  
  
return {  
    init : init  
};  
  
}  
);
```

请记住在 MVVM 中，观察对象包装器围绕 POJO（普通 JavaScript 对象）Property 建立其与表单间的魔术联系。视图模型代码的整个上半部分是绑定表单输入字段，以便我们能够与之交互。

观察对象看起来稍稍有点陌生，只因为需要一些附加信息以通知验证插件如何验证 Property。required 选项是插件提供的验证类型。电话号码与邮箱地址的验证信息供定制验证器使用，定制验证器是为这个特定类型的 UI 设计（指电话号码与邮箱地址）而创建的。插件提供了一些针对电话号码和邮箱地址的选项，但无法刚好满足我们的需要。幸运的是，该插件非常灵活，我们可以创建自己的验证程序。

还有一件事没提到——一个被称为 computed 观察对象（计算观察对象）的 Knockout 观察对象类型。它并不是 MVVM 特有的，而是 Knockout 特有的（尽管其他 MVVM 框架也可能包含它）。这是一个魔幻的术语，意味着值由编程方式得到，而非直接赋值。这里的代码实现决定观察对象获取的值是 true 还是 false。

提示 如果观察对象的值需要通过编程方式来决定，则使用 computed 观察对象比简单的赋值更合适。

实现了部分视图模型及验证功能之后，我们就拥有了一些能够处理列表条目添加和移除的绑定。值得关注的是，绑定的声明式部分（HTML 中的属性）承担着遍历数组值列表并确保 UI 反映出其当前状态的重任。我们不必手动编写这部分功能。

借助双向同步实现，Knockout 通过观察对象数组跟踪一切变化，并相应更新 DOM 和视图模型。我们要做的就是添加与删除条目，其他的就由 Knockout 来负责。很优雅，嗯哼？

创建定制验证规则的代码是专门针对验证插件的。对于如何定义自己的验证器，我们遵循网站的业务规则。我们通知插件，如果字段为空，则非法，并使用空白提示消息（提示 Required）；如果字段不为空但仍非法，则使用非法提示消息（提示 Invalid）。

验证器是可靠的。我们在 SPA 应用程序中可以使用大量其他特性。打开依赖的网站链接，以了解更多内容。

模块的最后一部分内容是 init() 函数。请记住，其代码只运行一次。这使其

成为一切设置工作的理想地。在这里，我们使用 jQuery 来添加模板到 DOM，并应用我们的绑定。通常情况下，调用的 Knockout 函数是 `applyBindings()`，但使用了验证插件，因此需要调用 `applyBindingsWithValidation()`。我们传入三个参数：视图模型自身、应用绑定的 DOM 节点，以及针对验证插件的配置。

A.3.8 entrytemplate.html

最后一部分程序如清单 A.18 所示，是员工数据的展示模板。这个文件只包含内嵌模板标记的 HTML 代码片段。这些 HTML 代码片段也被称为 *Partial* 或 *Fragment*。

清单 A.18 entrytemplate.html

```
<li class="entry">
  <button type="button" class="remove-entry"
    data-bind="click: removeEntry">
    &#9587;
  </button>

  <span data-bind="text: $data.lastName"></span>,
  <span data-bind="text: $data.firstName"></span><br />

  <span>Title:</span>
  <span data-bind="text: $data.title"></span><br />

  <span>Phone:</span>
  <span data-bind="text: $data.phone"></span> <br />

  <span>Email:</span>
  <span data-bind="text: $data.email"></span>
</li>
```



移除条目

如同 Backbone.js 模板，这个文件很简单。MVVM 的一个明显不同是使用声明式绑定（特定属性）。请记住 Backbone.js 使用占位符来标识模板引擎在哪儿插入数据及创建新视图。在 MVVM 中，模板和视图是同一个东西。定制 Knockout 属性通过视图模型的 Property 连接 DOM 内容。Knockout 在后台处理数据同步。

A.4 AngularJS示例

AngularJS 中你会立即注意到的一件事情是文件数量的减少。一个原因是该框架包含了所有开箱即用功能。

A.4.1 下载依赖

我们不会在 AngularJS 中使用 RequireJS。AngularJS 有其自己专用的模块系统。

所以，为了保持“AngularJS 风格”，我们将使用 AngularJS 模块（如表 A.3 所示）。这也进一步降低了依赖的数量。

表 A.3 AngularJS 版示例应用所需依赖

框架 / 库	URL	注释
AngularJS	https://angularjs.org	由于 AngularJS 是一体化框架，因此下载 AngularJS 就涵盖了大多数你需要的 MV* 功能
jQuery	http://jquery.com	伟大的整体方案工具库！DOM 操作和事件处理的神器

注意 AngularJS 自带一个叫 *jqLite* 的 jQuery 子集。如果没找到 jQuery，AngularJS 就会选择其作为回退版本，否则将使用完整的 jQuery。

A.4.2 目录结构

图 A.4 展示了 AngularJS 版本目录。尽管使用 AngularJS，但我们仍可以使用“按特性风格”的目录结构。

A.4.3 Shell 页面

再次地，我们的 index.html 页面（如清单 A.19 所示）很简单。然而请注意，AngularJS 版本中需要使用 ng-app 指令定义整个 SPA 应用程序的开始点。请记住，指令是特殊的 HTML 属性，其告知 AngularJS 你想做什么。看看 AngularJS 文档中的描述：

使用指令来自动引导 AngularJS 应用程序。ngApp 指令指派应用程序根元素，其通常放在页面根元素边上——例如 <body> 或 <html> 标签。

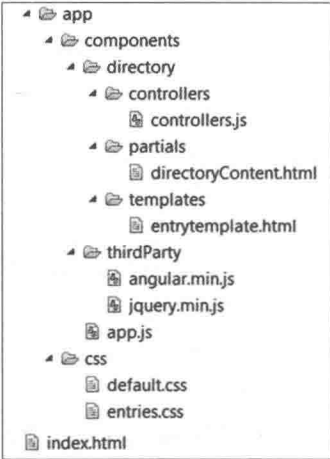


图 A.4 AngularJS 版本的应用目录结构

清单 A.19 index.html

```
<html ng-app="DirectoryApp">
<head>
<link rel="stylesheet" href="app/css/default.css">
<link rel="stylesheet" href="app/css/entries.css">
</head>
<body>
```

定义应用程序的起始点

```

<main id="directoryShell"
  class="container"
  ng-include
  src="'app/components/directory/partials/directoryContent.html'">
</main>

<script src="app/components/thirdParty/jquery.min.js"></script>
<script src="app/components/thirdParty/angular.min.js"></script>
<script src="app/components/app.js"></script>
<script src="app/components/directory/controllers/controllers.js"></
script>

</body>
</html>

```

动态获取文件的指令

我们还使用了一个 AngularJS 指令来告知 AngularJS 为 Shell 页面动态获取 SPA 内容。在这里要注意 AngularJS 只管理其自己的 JavaScript 模块。如果我们有其他第三方库要纳入进来，则需要使用标准的 `<script>` 标签。

A.4.4 app.js

我们的 `app.js` 文件，如清单 A.20 所示，极其简单。似乎有 100 万种不同的 AngularJS 代码构建方式。确定一条你自己的方式。尽管 AngularJS 是一个神奇的框架，但它在这点上很灵活。

清单 A.20 app.js

```

angular.module("DirectoryApp", [
  "DirectoryApp.controllers"
]);

```

只有一个依赖

如前所述，有许多种配置 AngularJS 项目的方式。对于大型项目，在不同源代码文件中存放各种模块。这使得开发与维护来得更容易。在第 9 章中，我们讨论了构建过程，压缩和串接文件到尽可能少的文件中去。对于这个简单示例应用，我们不需要搞得这么复杂。

A.4.5 directoryContent.html

就像 Knockout 的内容，AngularJS 版的代码也比较冗长，如清单 A.21 所示。我们使用大量内建 AngularJS 指令来配置 UI 行为。

清单 A.21 directoryContent.html

```

<header>Directory</header>

<section class="entries"
  ng-controller="DirectoryController">

```

控制器上下文

ng-model 绑定输入
字段和模型

显示必填字段
的相关错误

针对非法输入的
不同验证信息

```
<form name="entryForm">

  <p>
    <label for="firstName">First Name:<br/>
      <input id="firstName" name="firstName" type="text"
        ng-model="formEntry.firstName"
        required placeholder="First Name"/>

      <span class="error-message"
        ng-show="entryForm.firstName.$error.required">
        Required
      </span>
    </label>
  </p>

  <p>
    <label for="lastName">Last Name:<br />
      <input name="lastName" id="lastName" type="text"
        ng-model="formEntry.lastName"
        required placeholder="Last Name"/>

      <span class="error-message"
        ng-show="entryForm.lastName.$error.required">
        Required
      </span>
    </label>
  </p>

  <p>
    <label for="title">Title:<br />
      <input name="title" id="title" type="text"
        ng-model="formEntry.title"
        required placeholder="Title"/>
      <span class="error-message"
        ng-show="entryForm.title.$error.required">
        Required
      </span>
    </label>
  </p>

  <p>
    <label for="phone">Phone:<br/>
      <input name="phone" id="phone"
        ng-pattern="phoneRegX"
        type="text"
        ng-model="formEntry.phone"
        required placeholder="555-555-5555"/>

      <span class="error-message"
        ng-show="entryForm.phone.$error.required">
        Required
      </span>

      <span class="error-message"
        ng-show="entryForm.phone.$error.pattern">
        Invalid
      </span>
    </label>
  </p>
</form>
```



```

</p>
<p>
  <label for="email">Email:<br />
    <input name="email" id="email" type="email"
      ng-model="formEntry.email"
      required placeholder="youremail@address.com" />

    <span class="error-message"
      ng-show="entryForm.email.$error.required">
      Required
    </span>

    <span class="error-message"
      ng-show="entryForm.email.$error.email">
      Invalid
    </span>
  </label>
</p>
<p>
  <button id="add"
    ng-click="addEntry(formEntry)"
    ng-disabled="entryForm.$invalid">
    Add
  </button>

  <button id="clear"
    ng-click="clearForm()">
    Clear
  </button>
</p>
</form>

<ul class="entry-list"
  ng-include
  src="'app/components/directory/templates/entrytemplate.html'">
</ul>
</section>

```

针对非法输入的不同验证信息

如果所有表单条目都合法，则按钮可用，用于添加条目

清除表单

再次地，HTML 代码看起来会比其实际来得更冗长一点，因为代码要适配书本宽度。在这里使用指令来处理 Backbone.js 与 Knockout 版本中的同样任务。ng-model 用于值绑定，ng-click 用于按钮单击。对于验证，只需要添加 required 关键字，告知 AngularJS：如果字段为空，则模型非法。

错误与一个特定模型绑定关联在一起，因此我们可以在 标签中通过 ng-show 指令与 \$error 的结合使用来展示 / 隐藏它。\$error 后面的关键字是验证类型，用来触发 ng-show。在这里，当值为空时，带有 required 的 标签将显示，而带有 invalid 的 标签在用户输入的值不符合匹配模式时显示。

A.4.6 entrytemplate.html

清单 A.22 展示了模板代码。如同其他两个 MV* 框架的版本那样，该文件是一个 HTML 代码片段（也被称为 *Partial* 或 *Fragment*）。

清单 A.22 entrytemplate.html

```

<li class="entry" ng-repeat="entry in entries">
    <button type="button" class="remove-entry"
        ng-click="removeEntry(entry)">
        单击时移除条目
    </button>

    {{entry.lastName}}, {{entry.firstName}}<br />
    Title: {{entry.title}}<br />
    Phone: {{entry.phone}}<br />
    Email: {{entry.email}}
</li>

```

遍历条目列表的绑定

{{}} 指示模型数据插入位置

就像 Backbone.js 那样，占位符标识框架插入数据的位置。然而不必过多考虑在代码中编写模板指令，这是另一回事，AngularJS 会处理。

最后，清单 A.23 展示了应用的控制器代码。

清单 A.23 controllers.js

```

angular.module("DirectoryApp.controllers", [])
.controller("DirectoryController", function($scope) {

    $scope.phoneRegX
    = /(?:\d{3}|\(\d{3}\))([-\./\s])\d{3}\d{4}/;

    $scope.entries = [];

    $scope.addEntry = function(entryToAdd) {
        $scope.errorMsg = "";
        $scope.entries.push(angular.copy(entryToAdd));
        $scope.clearForm();
    };

    $scope.removeEntry = function(entryToRemove) {
        var index = $scope.entries.indexOf(entryToRemove);
        $scope.entries.splice(index, 1);
    };

    $scope.clearForm = function() {
        $scope.entryForm.$setPristine();
        $scope.formEntry = '';
    };

});

```

电话号码正则表达式（邮箱地址的是内建的）

单击添加按钮时添加条目

清除表单，回到初始的“清洁”状态，清除模型

请记住，在 AngularJS 中，\$scope 类似于视图模型。

A.5 小结

本附录里的示例代码展示了如何构建概念验证，同时在涉及模型到视图转化时，还体现出了不同框架在设计哲学上的差异性会改变代码侧重点：是倾向于多写 HTML 代码还是 JavaScript 代码？

XMLHttpRequest API

本章内容

- XMLHttpRequest API 知识回顾
- XHR 数据源模块的创建

在本附录中，我们会回顾服务器端调用的低级 API。在真实世界的应用程序里，我们更愿意依赖 MV* 框架来处理 XHR 调用（如果框架具备内建能力的话）。或者，即使框架不具备 XHR 调用能力，我们也会考虑像 jQuery 这样的辅助库。这些框架和库通常抽象出大量样板代码，只提供给开发者简单、易于使用的方法。尽管如此，至少大致掌握一些底层知识总是好的。这也是我们继续探究 JavaScript 的原因，以让你能够理解基本的 XHR 运作机制。为了保持简单性，我们在这里不会使用 RESTful 调用。

B.1 使用XMLHttpRequest对象

当你听到一些开发者提到发出一个 AJAX 调用时，他们通常说的是发出一个 XHR 调用（XHR 是 *XMLHttpRequest* 的缩写）。AJAX（或 *Ajax*）是 *Asynchronous JavaScript and XML* 的缩写，通常指的是借助 JavaScript 使用一个 XHR 调用结果来

动态更新 Web 页面。本节只关注 AJAX 的 XMLHttpRequest 部分的内容。

如第 1 章所述, XMLHttpRequest 的功能最早由微软的开发者实现。最终, 其演进成了所有主流浏览器的标准 API 实现。该标准允许你使用简单的 JavaScript 代码创建一个 XMLHttpRequest 对象实例, 该对象的行为在绝大多数现代浏览器中都相同。

我们先来创建该对象的实例。只要用户使用支持 XHR API 的浏览器, 那么创建该对象只需这么一行代码:

```
var xhrObj = new XMLHttpRequest();
```

如果需要在支持微软的 IE 7 之前的版本, 还应该在一个 XMLHttpRequest 对象判断中包装对象创建的代码; 如果不支持 XMLHttpRequest 对象, 则需要使用一个 ActiveX 对象。微软推荐的方式如下:

```
function createXHRObject() {  
    if (window.XMLHttpRequest) {  
        return new XMLHttpRequest();  
    }  
    else {  
        return new ActiveXObject("Microsoft.XMLHTTP");  
    }  
}
```

随着该对象实例的创建, 现在就可以向服务器端发出一个请求了。

B.2 发起请求

创建一个 XHR 对象实例之后, 就可以使用其事件、方法以及 Property 来定制我们的服务器端调用了。以下列表展示了本节涉及的 XHR 概念:

- onreadystatechange——该事件在调用状态改变时触发。可以指派自己的函数来处理该事件。通常用该事件来观察调用是否完成。如果完成了, 接下来就可以检查调用是成功还是失败, 并做出相应的反应。
- open——该方法为调用指派请求方法以及 URL。存在着各种请求方法, 但并非所有的请求方法都可以得到服务器端所采用技术的支持。在第 7 章中, 我们只使用了 GET、POST、PUT 和 DELETE。
- setRequestHeader——这个方法用来指定请求的头部, 如发送内容的类型以及在响应中接收内容的类型等。
- send——该方法触发请求。对于如 GET 这样不需要请求体的请求, 可以传进 null 或使用该函数的重载版本 (不带参数, 这是首选方式)。否则, 通过

该方法的参数传进请求体的数据。

- `readyState`——这是个 `Property`，在 `onreadystatechange` 事件触发时我们将检查它，以知晓调用处于何种状态。该 `Property` 有五个基本值：0 (`UNSENT`)、1 (`OPENED`)、2 (`HEADERS_RECEIVED`)、3 (`LOADING`) 和 4 (`DONE`)。在大多数情况下，只需要检查 4 的值以确保调用完成。
- `status`——调用完成时，可以通过检查该 `Property` 来观察调用是成功还是失败了。状态码为 400 段或 500 段就表示失败。在 www.w3.org/Protocols/rfc2616/rfc2616-sec10.html 可以找到完整的状态码列表。
- `responseText`——该 `Property` 包含了字符串型的响应体。对于无错误的 JSON 调用，JSON 文本将包含在这里。对于错误情况，由服务器端代码来决定创建什么内容到响应体。
- `responseXML`——如果响应能够解析为 XML 或 HTML，则该 `Property` 包含了一个 `Document` 对象。否则，该 `Property` 为空。

这里只覆盖了基本概念，完整的 `XMLHttpRequest` API 列表可以在 www.w3.org/TR/XMLHttpRequest 找到。

B.2.1 使用 URL 参数

当通过带有简单负载或根本就没有负载的请求来获取数据时，GET 方法是好选择。此种类型的请求，不发送请求体，而是在需要发送信息时，采取在 URL 中添加 URL 参数的方式。在我们的第一个示例中（如清单 B.1 所示），将使用一个简单的 GET 请求，其带有一个 URL 参数（购物车 ID），并通过该参数来获取购物车内容。

在第 7 章的购物车应用程序中，我们通过 AngularJS 绑定返回数据到视图。而在这里，我们将通过控制台输出方式来更好地理解每个请求的情况。

清单 B.1 使用 GET 和 URL 参数进行 XHR 调用

```
function getCart(){
  var xhrObj = createXHRObject();
  xhrObj.onreadystatechange = function () {
    if (xhrObj.readyState == 4) {
      var response = "\nreadyState: "
        + xhrObj.readyState
        + "\nstatus: " + xhrObj.status
        + "\nstatusText: " + xhrObj.statusText
        + "\nresponseText: " + xhrObj.responseText;

      if (xhrObj.status == 200) {
        response = "Success:" + response;
      }
    }
  }
}
```

创建 XHR 对象

对于 OK 状态 (200)，为控制台信息添加“Success”前缀，否则添加“Error”状态

只在状态变为 4（完成）的时候才进行处理

```
    } else {  
        response = "Error:" + response;  
    }  
  
    console.log(response);  
  
    }  
  
};  
  
xhrObj.open("GET",  
    "/SPA/controllers/shopping/getCart?cartId=123",  
    true);  
  
xhrObj.setRequestHeader("Accept", "application/json");  
  
xhrObj.send();  
}
```

发起请求

定义 GET 方法的请求，包含一个 URL 参数 cartId

告知服务器端，响应格式为 JSON

请求完成后，以下信息会输出到控制台：

```
Success:  
readyState: 4  
status: 200  
statusText: OK
```

在这些信息中，readyState 为 4 表示完成，调用状态为 200/OK。我们还能够在浏览器开发者工具的“Network”选项卡中确认请求 URL 及请求头信息（如图 B.1 所示）。



图 B.1 成功发起获取购物车内容的请求。通过 GET 方法，传递 cartId 为“123”的 URL 参数

在浏览器开发者工具中，还能在控制台看到服务器端的输出（如图 B.2 所示）。

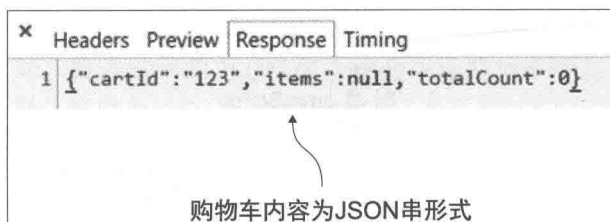


图 B.2 服务器端响应购物车的 JSON 格式文本。我们可以看到购物车还没有记录

在“Network”选项卡中观察响应，可以看到购物车中还没有记录。

为了添加一个游戏到购物车，只需要用到 `cartId` 和游戏 ID。我们可以继续使用 URL 参数，但这时发送更复杂一些的请求负载可能是一个更好的选择。要发送更复杂的负载，则需要依赖请求体。

B.2.2 使用请求体

当添加新记录到购物车时，我们更倾向于在请求中发送一个对象。对于请求而言，我们将转换 JavaScript 请求对象为一个 JSON 串，并通过请求体传输这个 JSON 串。清单 B.2 展示了该技术。

清单 B.2 使用 POST 和请求体进行 XHR 调用

```
function updateCart() {
    var xhrObj = new XMLHttpRequest();
    xhrObj.onreadystatechange = function () {
        if (xhrObj.readyState == 4) {
            var response = "\nreadyState: "
                + xhrObj.readyState
                + "\nstatus: " + xhrObj.status
                + "\nstatusText: " + xhrObj.statusText;
            + "\nresponseText: " + xhrObj.responseText;

            // 200 is OK
            if (xhrObj.status == 200) {
                response = "Success:" + response;
            } else {
                response = "Error:" + response;
            }

            console.log(response);
        }
    };
    xhrObj.open("POST",
        "/SPA/controllers/shopping/addToCart",
        true);
    xhrObj.setRequestHeader("Accept", "application/json");
}
```

创建 XHR 对象

只在状态变为 4 (完成) 的时候才进行处理

对于 OK 状态 (200)，为控制台信息添加“Success”前缀，否则添加“Error”状态

定义 POST 方法的请求

告知服务器端，响应格式为 JSON

告知服务器
端接收请求
中的 JSON
格式

```
xhrObj.setRequestHeader("Content-Type", "application/json");

var cartRequestObj = {
    cartId : "123",
    itemNum : "madden_nfl_15"
};

var cartRequestString = JSON.stringify(cartRequestObj);

xhrObj.send(cartRequestString);
}
```

创建 JavaScript
请求对象

转换 JavaScript
对象为 JSON 格
式文本

通过请求体传递
JSON 文本

注意 如果非得支持老式浏览器，JSON.js 可以用作 JSON 对象的辅助操作工具。

观察这些代码，你会发现这一次我们使用 POST 请求方法，以利用请求体。之后使用 JavaScript 方法 `JSON.stringify()` 来将 JavaScript 对象转换成 JSON 格式文本。接下来，传递该文本串进 XHR 对象的 `send()` 方法，通过请求体发送负载（数据）。

另一件要注意的事情是，由于这一次我们发送 JSON 串，因此就需要指定不同的请求头信息。打开浏览器开发者工具的“Network”选项卡，来看一下现在的请求情况（如图 B.3 所示）。

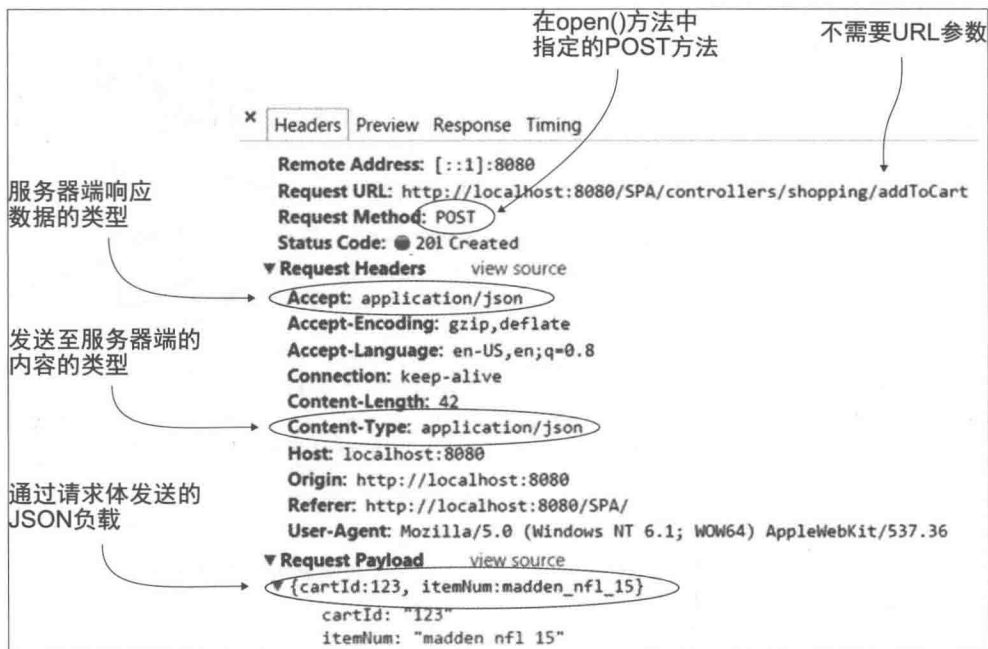


图 B.3 以 JSON 格式文本方式，通过 POST 方法来发送复杂请求对象到 SPA 应用程序的服务器端

在服务器端编写代码，以实现当添加或更新购物车时将整个购物车返回给客户端。检查返回的 JSON 串，你会发现购物车中有了一条 ID 为 `madden_nfl_15` 的记录（见图 B.4）。

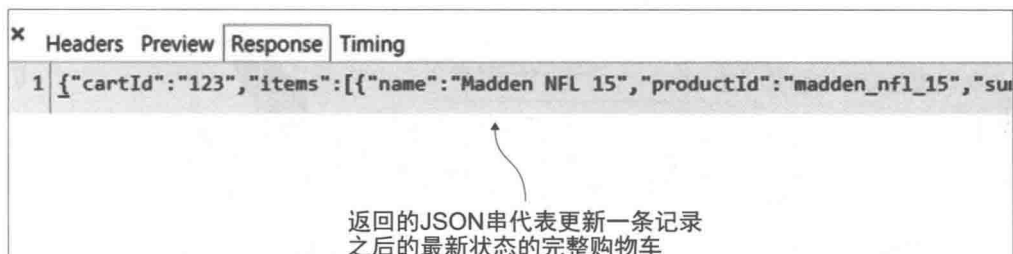


图 B.4 新记录发送到服务器端之后，购物车中就有了一个游戏

第7章内容的服务器端 设置与总结

本章内容

- 处理服务器端对象和任务
- 为第 7 章示例项目设置 Spring MVC
- 在第 7 章示例项目中使用 Spring 注解

如第 7 章所述，你可以为该章示例项目选择任意的服务器端技术。如果你愿意在本项目中使用 Spring MVC，这里会提供 Spring MVC 设置的有关细节。同时也涉及 Spring 注解的使用。完整代码可以在线下载。

如果你采用完全不同的技术栈，本附录也总结了每次调用执行的任务，以便更容易将任务转换成其他技术实现。请记住本项目使用 RESTful 服务，因此所用的后端技术必须直接或通过第三方库 / 插件间接支持 REST。

C.1 服务器端对象

本示例项目只用到两个数据对象：一个 ShoppingCart 对象和一个 Game 对象。每个购物车包含一个游戏集合（见图 C.1）。Game 对象是独立的，能够为 ShoppingCart 对象单独使用。

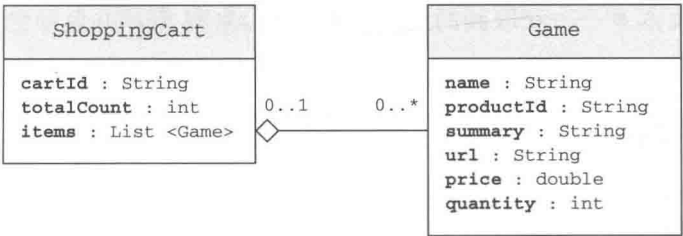


图 C.1 购物车有一个条目列表，其容纳不定数量的游戏

我们还在服务器端使用一个对象来容纳错误发生时的信息（如图 C.2 所示）。message 字段是显示给用户的错误信息。服务器端除了要处理所有服务器端的异常，还会将异常通过 exceptionText 字段返回给客户端 UI。

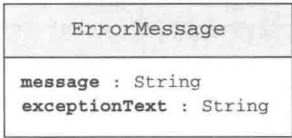


图 C.2 ErrorMessage 对象用于传递错误回 UI

现在来小结一下服务器端调用。我们在讨论 Spring MVC 之前做这件事，有利于更好理解底层机制（不管你用何种后端开发技术）。

C.2 服务器端调用小结

本节概括了一个服务调用时服务器端执行的任务。我们将描述在请求和响应中要用到的 URL 格式以及所有对象，并会针对各种情况提供一个关于服务器端代码处理任务的简短小结。

C.2.1 查看购物车

该调用会在用户单击查看购物车链接时触发。花括号 {} 表示一个包含所请求 cartId 的路径变量。表 C.1 包含了其 Property。

表 C.1 通过路径变量获取所需购物车的调用

URL	HTTP 方法	请求	响应
/shopping/carts/{cartId}	GET	空	购物车

任务列表：

- 从内存中获取购物车——如果在静态散列中找到一个匹配所请求 cartId 的购物车，代码就会返回该匹配的购物车。如果未找到匹配结果，新的购物车就会创建并添加到购物车散列中。

- 返回购物车的 JSON 文本串——获取到的购物车转换回 JSON 串，并返回给客户端。

C.2.2 给购物车添加条目

当添加了一条产品条目时，我们通过 POST 的 HTTP 请求，传递购物车 ID 和产品条目 ID。表 C.2 涵盖了调用 Property。

表 C.2 添加产品条目到购物车的调用

URL	HTTP 方法	请求	响应
/shopping/carts/{cartId}/products/{productId}	POST	产品	购物车

任务列表：

- 添加条目——首先要检查从产品显示视图中添加的产品在购物车里是否已经存在。如果存在，则已有产品数量加 1，否则就从库存中获取所需游戏，并创建一个新的 Game 对象。总条目数量同时也做相应更新。
- 在内存更新购物车——更新内存里的购物车，已反映出购物车新状态。
- 返回购物车的 JSON 文本串——获取到的购物车，其携带更新过的产品列表与产品数量，将其转换回 JSON 串，并返回给客户端。

C.2.3 更新购物车

本调用修改一个已有产品。任何修改都将提交整个购物车至服务器端。表 C.3 涵盖了本调用的 Property。

表 C.3 通过路径变量更新相应购物车的调用

URL	HTTP 方法	请求	响应
/shopping/carts/{cartId}	PUT	购物车	购物车

任务列表：

- 修改数量——整个购物车在修改完成后发送。由于一个简版的购物车通过购物车视图的 \$scope 生成，因此任何遗漏的信息会通过服务器端的静态库存列表补充生成。总条目数量也会更新。
- 在内存更新购物车——内存中的购物车也会更新，以反映出购物车的最新状态。
- 返回购物车的 JSON 文本串——获取到的购物车，其携带更新过的产品列表与产品数量，将其转换回 JSON 串，并返回给客户端。

C.2.4 删除条目

当删除一条条目时，在 URL 中标识购物车与条目。不需要在请求体中发送任何对象，因为要删除的资源完全可以只通过 URL 标识出来。表 C.4 涵盖了该调用 Property。

表 C.4 通过路径变量删除相应条目的调用

URL	HTTP 方法	请求	响应
/shopping/carts/{cartId}/products/{productId}	DELETE	空	购物车

任务列表：

- 从内存获取购物车——通过 URL 的信息从内存中获取购物车。
- 删除条目——在该调用中，我们迭代购物车里的条目，直到找出一条匹配 URL 中 productId 的条目。找到匹配项之后，就移除该条目。
- 修改购物车里的总数——条目移除后，条目数量要重新计算。
- 返回购物车的 JSON 文本串——获取到的购物车，其携带更新过的产品列表与产品数量，将其转换回 JSON 串，并返回给客户端。

第 7 章只聚焦购物车处理，以阐述个中概念。在该示例项目中，我们还发出了一些其他调用。通过产品搜索视图，用户可以搜索游戏。如果搜索到游戏，通过产品显示视图里的按钮，游戏会添加进购物车。

C.2.5 搜索产品

在产品搜索视图中，用户输入搜索项。搜索项是一个简单字符串，表示游戏的完整或部分名称（如表 C.5 所示）。

表 C.5 搜索游戏的调用

URL	HTTP 方法	请求	响应
/games/search/{srchTerm}	GET	空	Game 对象列表

任务列表：

- 搜索——源自 URL 路径的搜索项与库存中的所有游戏进行比较和匹配（至少是部分匹配）。匹配数量的范围可以是 0 到 n 。该调用的返回结果是一个 Game 对象列表。
- 返回列表的 JSON 文本串——将所获取游戏列表的 JSON 串返回给客户端。

C.2.6 显示产品

搜索结果列表呈现之后，用户可以单击标题以展示游戏具体信息（如表 C.6 所示）。

表 C.6 显示所选游戏信息的调用

URL	HTTP 方法	请求	响应
/games/id/{productId}	GET	空	游戏

任务列表：

- 通过 ID 获取游戏——在本次调用中，产品 ID 传进了 URL。逻辑代码使用该 ID 从库存中找到匹配游戏。
- 返回购物车的 JSON 文本串——将所获取游戏的 JSON 串返回给客户端。

C.3 示例项目

本节覆盖示例项目用到的服务器端技术，并介绍 Spring MVC 设置。同时还会阐述 Spring 注解的使用。

C.3.1 准备工作

可以用你已经在用或喜欢的技术栈来替代这里的指定技术选型。下述列表是本书所用的技术栈：

- *Apache Tomcat*——Tomcat 是可以自由使用的服务器，其支持 Java EE¹。Tomcat 版本 7 支持 Servlet 3.0 规范，其符合我们的服务器端应用程序需要（如果是更高版本也适合）。我们可以从 <http://tomcat.apache.org> 下载 Tomcat。如果你喜欢不同的 Java Web 服务器，请尽管使用它，只要其支持 Servlet 3.0 规范。
- *Apache Maven 3.2.1* 或更高版本——Apache Maven 是一个软件管理工具。其可以从 <http://maven.apache.org> 获取。

下述依赖通过 Maven 管理：

- *Spring 4.1.6* (Core + MVC) ——Spring (<https://spring.io>) 是一个基于 Java 的应用框架和控制反转容器。<http://repo.spring.io/release/org/springframework/spring> 上有一系列可下载选项。
- *Jackson*——Jackson 是一个快速的轻量级 JSON 解析器。其缺省支持 Spring MVC。本示例项目使用 2.5.3 的版本。可以从 <http://wiki.fasterxml.com/JacksonDownload> 下载它。
- *Commons Logging* (Spring 需要它) ——参见 <http://commons.apache.org/proper/commons-logging> 以获取更多细节。可从 <http://commons.apache.org/>

1 Tomcat只支持Java EE规范的Web Profile部分，但其足以满足本书示例项目的需求。诸如GlassFish、JBoss、WebLogic与WebSphere等服务器则支持整个Java EE规范。

proper/commons-logging/download_logging.cgi 下载。本示例使用 1.2 版本。

- *Commons Lang*（可选项）——Lang 是辅助工具套件，对于字符串管理特别有用。本书使用 3.4 版本，但版本 3 及以上就可以了：<http://commons.apache.org/proper/commons-lang>。Lang 不是强制选项。在这里出于便利性而使用它，你也可以使用 Java 原生功能作为替代。

同时还要注意我们使用了 Java 8。对于列表中所列的工具，如果你使用的是早期版本，还请确保其适配我们使用的 Java 及 Tomcat 版本。

在应用程序运行起来之前，还需要配置 Spring MVC。幸运的是，配置工作极其简单，因为我们不使用数据库或任何外部 Web 服务。

C.3.2 Spring MVC 设置

首先在你的 IDE 中创建一个动态 Web 项目。如果你使用 Eclipse，导入项目后即准备就绪。如果使用其他的 IDE，则可以在所用 IDE 中创建一个新的动态 Web 项目，并从压缩文件中复制代码。在第 7 章的下载源文件中可以获取你需要的一切，包括 pom.xml。请参见 readme.txt 文件获取更多的安装说明。

工作空间（workspace）设置完毕之后，就可以开始我们的配置工作了。在 Spring 2.5.6 及更高版本中，可以使用 XML 或 Java 类来配置。本示例项目使用 Java 配置方式。这纯粹是一个设计选择，请尽管使用 XML 配置方式，只要你喜欢（甚至混合两种方式）。从 Servlet 3.0 规范开始，甚至就不再需要 web.xml 文件了，因此该文件不是必需选项。只在满足 Maven 插件需要时才涉及 web.xml 文件。然而请记住，容器仍需支持 web.xml。具体请查阅所用服务器的文档。

在我们的示例项目中，我们将遵循确保配置代码量最小化的方式。为了让 Web 应用程序不需 web.xml 文件就能运行起来，需要一个实现了 `WebApplicationInitializer` 接口的类。该接口在 Spring 3.1 或更高版本中提供。清单 C.1 展示了实现该接口的类代码。该接口放在哪个包里并不重要，应用程序能够自动侦测到它。

清单 C.1 WebAppInitializer.java

```
public class WebAppInitializer implements WebApplicationInitializer {  
  
    @Override  
    public void onStartup(ServletContext container)  
        throws ServletException {  
  
        AnnotationConfigWebApplicationContext ctx  
            = new AnnotationConfigWebApplicationContext();
```

Web 应用程序上下文及注解支持



```

        ctx.register(WebMvcConfig.class);
        ctx.setServletContext(container);
        Dynamic dynamic
            = container.addServlet("dispatcher",
                new DispatcherServlet(ctx));
        dynamic.addMapping("/controllers/*");
        dynamic.setLoadOnStartup(1);
    }
}

```

设置应用程序的 Servlet 上下文

注册 MVC 配置类

注册 dispatcher Servlet

用 dispatcher 映射所有以 /controllers/（在根路径之后）开头的请求

来自客户端的所有请求都将通过 Spring MVC 的 dispatcher Servlet。为了使用 Java 配置方式、不借助 web.xml 文件，需要实现 `WebApplicationInitializer` 接口并提供 `onStartup` 方法的实现。Spring 会赋予该方法正确的 Servlet 上下文。我们在 `onStartup` 方法中定义那些在 Servlet 容器启动阶段执行的各种引导程序。同时，还需要让 Spring 知道哪个类将处理我们的 MVC 配置参数。清单 C.1 定义了 `WebMvcConfig` 作为我们的 MVC 配置类。清单 C.2 展示了 `WebMvcConfig` 代码。

清单 C.2 WebMvcConfig.java

```

@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
    { "com.gamestore.controllers", "com.gamestore.dataservices" })
public class WebMvcConfig {}

```

指明这是一个配置类

让 Spring MVC 生效

让组件扫描这些包

由于我们定义了 `WebMvcConfig` 作为 MVC 配置类，因此像使用 `@Configuration` 注解那样来装饰 `WebMvcConfig`。 `WebMvcConfig` 类本身没有类体。真正的魔法源是 `@EnableWebMvc`。该注解设置 Spring MVC 默认值。如果你需要重写任何缺省值，可以在配置类中实现 `WebMvcConfigurer`，并重写其各种方法。作为选择，Spring 提供了 `WebMvcConfigurerAdapter` 作为一个便利方式以避免不得不实现 `WebMvcConfigurer` 所有的 16 个方法。于我们而言，只需要一个基本的 Spring MVC 配置，于是我们使用 `WebMvcConfigurerAdapter`。

最后，我们通过 `@ComponentScan` 注解来告知 Spring 在哪儿查找应用程序中使用的其他组件。我们在 `controllers` 包里有控制器组件，另有一个简单的服务组件在 `dataservices` 包中。在下一节我们会了解这两块内容。

图 C.3 展示了我们 IDE 的工作空间情况，其具备完整的项目配置，并准备就绪。

C.3.3 项目中用到的注解

我们已经讨论了 Spring MVC 设置中用到的注解。本节内容覆盖控制器与服务中使用的其他注解。为了实际展示注解，我们将拿购物车控制器来做例子。首先描述类级别的注解，之后逐渐延伸到方法级别的注解。

1. 控制器使用的类级别注解

我们的购物车控制器类由两个注解来装饰：`@RestController` 和 `RequestMapping` (如清单 C.3 所示)。在 Spring MVC 中，可以结合使用多个注解：

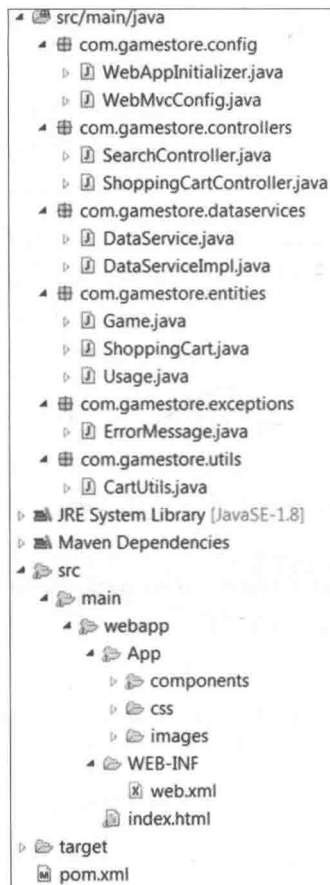


图 C.3 完整的项目工作空间

清单 C.3 类级别注解

```
@RestController
@RequestMapping("/shopping")
public class ShoppingCartController {
```

在底层应用 `@Controller` 与 `@ResponseBody`

所有的以 `"/shopping"` 打头（在根路径之后）的 URL 都会路由到该控制器

`@RestController` 是一个便利注解，其等同于添加 `@Controller` 与 `@ResponseBody` 到类。

`@Controller` 注解让 Spring MVC 知道该类是一个控制器且具备处理请求的方法。有了这个注解之后，Spring MVC 将扫描请求映射注解的方法，我们马上就会讲到该方法。

当我们希望方法的返回对象绑定到调用的响应体时，就要使用到 `@ResponseBody`。这个注解可以应用在类级别或方法级别。当应用在类级别时，控

制器中的任何方法都遵循这一行为¹，因此其也可以在方法级别省略。当购物车对象返回时，该对象将通过响应体返回给客户端 UI。此外，由于 Jackson 库在 classpath 中，Jackson 将自动转换购物车对象为 JSON 格式文本。

@RequestMapping 注解映射进来的请求到特定类和 / 或方法。其可以装饰类、方法或两者。在我们的购物车示例项目中，其在两个地方都定义了。这在被映射的请求中构建了一种层级结构。定义在类级别的 @RequestMapping 映射整个类到以 /shopping 串打头（根路径之后）的任何 URL。

控制器别处的 @RequestMapping 注解，用特定方法映射请求 URL 中 /shopping 之后的其他部分。我们会在下一节讨论这些内容。

2. 方法级别注解

要实际理解方法级别的注解，需借助购物车更新功能。更新功能涵盖了在所有方法中用到的注解（如清单 C.4 所示）。

清单 C.4 方法级别注解

```
@RequestMapping(  
    value = "/carts/{cartId}",  
    method = RequestMethod.PUT)  
public ResponseEntity<ShoppingCart> updateCart(  
    @RequestBody ShoppingCart cart,  
    @PathVariable String cartId,  
    HttpServletRequest request) throws Exception {  
  
    authenticateRequest(request, cartId);  
  
    ShoppingCart newCart = dataService.updateQuantities(cart);  
  
    return new ResponseEntity<ShoppingCart>(newCart, HttpStatus.OK);  
}
```

绑定请求体到 cart 参数

映射 URL 和请求类型到该方法

映射路径变量 cartId 到 cartId 方法参数

为了装饰方法自身，我们再次用到了 @RequestMapping 注解。它的值是用来映射请求到特定方法的 URL 的一部分。由于已经使用部分 URL 在类级别定义了注解，因此该方法对应的完整路径就是 /shopping/carts/{cartId}。

你还会注意到，在 UI 侧可以定义 URL 路径参数。URL 路径参数由花括号 {} 装饰，并对应到由 @PathVariable 注解装饰的方法参数上。最后，在 @RequestMapping 中，我们定义 HTTP 请求方法为 PUT 方法，因为我们要修改购物车。

@RequestMapping 和 @PathVariable 允许我们方便地通过路径变量和 HTTP 请求方法类型来定义 URL 路径中的资源。如第 7 章所述，URL 路径中的唯

1 指方法的返回对象绑定到调用的响应体。——译者注

一资源标识与 HTTP 请求方法结合起来创建 RESTful 请求。

在这里用到的另一个方法级别注解是 `@RequestBody`。它绑定由其装饰的方法参数到请求体中的发送对象上。在这个购物车修改功能中，整个购物车就是请求体中的发送对象。`@RequestBody` 注解绑定购物车对象到 `ShoppingCart` 参数。由于我们使用了 Jackson 库，Spring 会自动转换请求体的 JSON 格式文本为原生的 Java `ShoppingCart` 对象。

我们总是希望调用一直是成功的，但如果出现错误，则可以通过注解将异常抛进一个有意义的响应中。

3. 异常处理注解

REST 架构的另一个方面是定义 HTTP 状态以让客户端知晓调用结果。在错误发生时这显得特别重要。在 Spring MVC 中，我们也一样可以通过注解，将 Java 异常与特定响应关联起来。这里有个控制器的样例，其抛出任何 `UnauthorizedAccessException` 异常并将其转到合适响应中（见清单 C.5）。

清单 C.5 异常处理注解

```
@ExceptionHandler(UnauthorizedAccessException.class)
@ResponseStatus(value = HttpStatus.UNAUTHORIZED)
public ErrorMessage handleUnauthorizedException
    (UnauthorizedAccessException e) {
    return new ErrorMessage(
        "Unable to complete user request.",
        e.getMessage());
}
```

要处理的异常

要返回的 HTTP 状态

返回定制 ErrorMessage 类的新实例

`@ExceptionHandler` 注解可用于指定针对特定异常应该返回什么内容。在这个购物车示例里，我们正在使用一个替代登录动作的简单伪认证方法。我们对请求中传入的购物车 ID 与会话中存储的 ID 进行了比较，以便阐述异常处理注解的用法。如果请求未通过检查，购物车控制器中的相关方法将抛出 `UnauthorizedAccessException` 异常。这个方法会自动得以调用。其返回一个定制对象，定制对象中包括一条人性化消息及抛出的异常。我们通过 `@ResponseStatus` 注解在这里定义了 401/Unauthorized（未授权）状态。

最后讨论的两个注解，用于请求 Spring 将数据服务组件注入购物车控制器。

4. 服务与自动装配

在第 7 章里，我们将模拟数据从客户端移到了服务器端。将其放进一个名为 `DataServiceImpl` 的类里。我们将这个类当作数据存储、服务以及数据访问对象（DAO）的混合体。通常情况下，服务为业务层代码创建统一 API，而业务层代码

反过来通过 DAO 和其他服务来访问应用程序数据。由于我们并未使用数据库或其他 Web 服务，因此为简单起见，这个类将承担多重角色。

由于搜索与显示游戏相关的其他控制器中也用到了该类，因此，我们并不直接在购物车控制器中创建其新实例。相反地，我们让 Spring 来管理它，并注入服务到使用它的任何控制器中。

该类除了提供方法用来搜索、访问静态库存数据以及购物车，本身并不处理其他功能。如果你想分析源代码，则可以下载它。我们要突出的是这里使用到了注解。

要定义该类为一个注入服务，需用 @Service 注解装饰它。在 Spring 执行组件扫描时，Spring 将能够自动监测到该类：

```
@Service
public class DataServiceImpl implements DataService {
```

要使用服务，可以在任何注入目标类中用 @Autowired 注解装饰一个 setter 方法或实例变量。请注意，对于自动装配，我们使用类的接口。在购物车示例中，我们定义了一个 DataService 接口类型的实例变量，并用 @Autowired 注解装饰它：

```
@Autowired
private DataService dataService;
```

当 Spring 找到该注解时，Spring 会自动分配该服务的一个实例给这个实例变量，以提供使用。

该示例项目仅触及 Spring MVC 的一点皮毛。如果你想掌握更多有关 Spring MVC 的知识，可以从 <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html> 上的友好介绍开始。

安装Node.js与Gulp.js

本章内容

- 安装 Node.js
- 安装 Gulp.js

Gulp.js 是一个运行于 Node.js 环境中并基于 JavaScript 的 Task Runner。本附录指导你安装 Node.js 和 Gulp.js。

D.1 安装Node.js

除了 Gulp.js，第 9 章还有另一个基本要求。Gulp.js 运行在 Node.js 之上，因此，还需要安装 Node.js。可通过 <https://nodejs.org/download> 下载 Node.js。在 Node.js 官网，也能够找到对应各种操作系统的 Node.js 安装程序。Node.js 安装完毕之后，就可以通过在命令行输入 `node -v` 来验证安装结果¹：

```
C:\> node -v
V0.12.2
```

¹ 截至本书翻译到此处时（2016-6-30 20:23:33），Node.js 的版本为：v6.2.2。——译者注

这告知我们安装了什么版本的 Node.js，并做出适当的响应，确认安装成功。Node.js 安装好了之后，接下来安装 Gulp.js。

D.2 安装Gulp.js

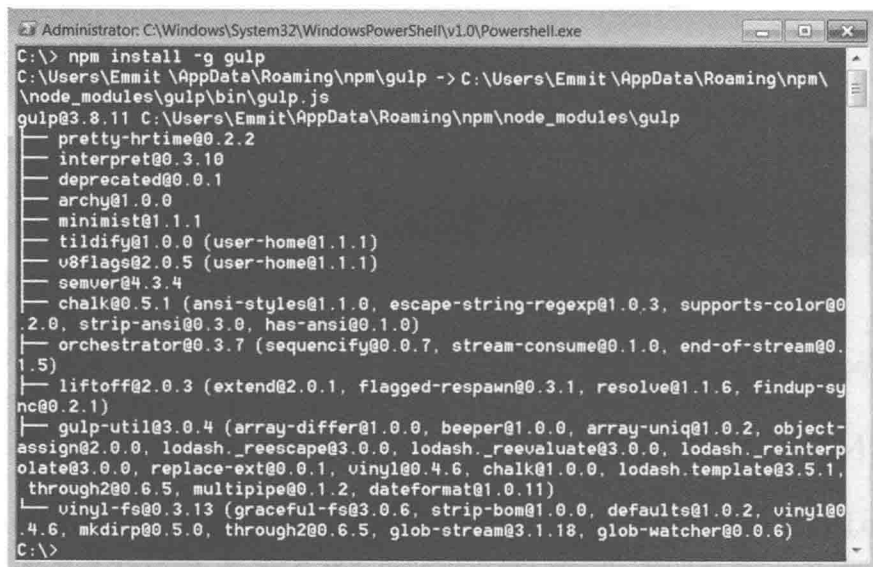
要开始使用 Gulp.js，首先需要安装它。有了 Node.js，事情就变得简单起来。我们可以借助安装 Node.js（最新版）之后得到的包管理器（npm），并只需一条简单的命令行命令即可下载并安装 Gulp.js。

在 npm 中，我们安装的东西被称为包（Package）。包可以局部安装或全局安装（或两者）。我们用 -g 选项来全局安装 Gulp.js，以便我们能够在任何目录中运行 Gulp.js。在命令行，运行：`npm install -g gulp`。

注意 如果使用 OS X 或 Linux 存储系统，在安装包时，建议你避开 `sudo`。

参见：<https://docs.npmjs.com/getting-started/fixing-npm-permissions>。

如果安装成功，应该能够看到类似图 D.1 所示的控制台输出。



```
Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\PowerShell.exe
C:\> npm install -g gulp
C:\Users\Emmit\AppData\Roaming\npm\gulp -> C:\Users\Emmit\AppData\Roaming\npm\
\node_modules\gulp\bin\gulp.js
gulp@3.8.11 C:\Users\Emmit\AppData\Roaming\npm\node_modules\gulp
├── pretty-hrtime@0.2.2
├── interpret@0.3.10
├── deprecated@0.0.1
├── archy@1.0.0
├── minimist@1.1.1
├── tildify@1.0.0 (user-home@1.1.1)
├── v8flags@2.0.5 (user-home@1.1.1)
├── semver@4.3.4
├── chalk@0.5.1 (ansi-styles@1.1.0, escape-string-regexp@1.0.3, supports-color@
├── 2.0, strip-ansi@0.3.0, has-ansi@0.1.0)
├── orchestrator@0.3.7 (sequencify@0.0.7, stream-consume@0.1.0, end-of-stream@0.
├── 1.5)
├── liftoff@2.0.3 (extend@2.0.1, flagged-respawn@0.3.1, resolve@1.1.6, findup-sy
├── nc@0.2.1)
├── gulp-util@3.0.4 (array-differ@1.0.0, beeper@1.0.0, array-uniq@1.0.2, object-
├── assign@2.0.0, lodash._reescape@3.0.0, lodash._reevaluate@3.0.0, lodash._reinterp
├── olate@3.0.0, replace-ext@0.0.1, vinyl@0.4.6, chalk@1.0.0, lodash.template@3.5.1,
├── through2@0.6.5, multipipe@0.1.2, dateformat@1.0.11)
├── vinyl-fs@0.3.13 (graceful-fs@3.0.6, strip-bom@1.0.0, defaults@1.0.2, vinyl@0
├── 4.6, mkdirp@0.5.0, through2@0.6.5, glob-stream@3.1.18, glob-watcher@0.0.6)
C:\>
```

图 D.1 安装 Gulp.js

如处理 Node.js 那样，可以通过 -v 命令：`gulp -v` 来检查版本号，并确认安装结果。由于可以在任何目录运行 Gulp.js 命令，因此请切换至示例项目目录。在这里，我们可以安装项目特定需要的任何包。

注意 为了包含 Node.js 项目的相关元数据、构建依赖项及其版本的有关信息，我们需要创建一个 `package.json` 文件。npm 包管理器可以在这个文件中使用版本字段来确保正确安装了包版本。此外，如果该文件已经存在，在使用 `npm install` 命令时若是带上 `--save` 或 `--save-dev` 选项，安装的包信息将添加到依赖项列表中。`--save` 选项可用于指定运行时依赖（应用程序使用，如 MV* 框架），`--save-dev` 选项用于开发环境依赖。尽管 `package.json` 文件中的依赖项列表会在使用上述选项进行包安装时自动更新，但我们仍需要创建初始的 `package.json` 文件。可以手工或通过 `npm init` 命令来创建。如果需要一个创建指导，`npm init` 命令将通过问你几个问题来一步步向导式创建 `package.json` 文件。可以随意编辑生成的 `package.json` 文件。创建过程完成后，就创建好了一个初始的 `package.json` 文件，其带有你指定的缺省值。

博文视点诚邀精锐作者加盟

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博文通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巅。

英雄帖

江湖风云起,代有才人出。

IT群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享IT心得

• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身定制写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



@博文视点Broadview



微信公众号 博文视点Broadview

