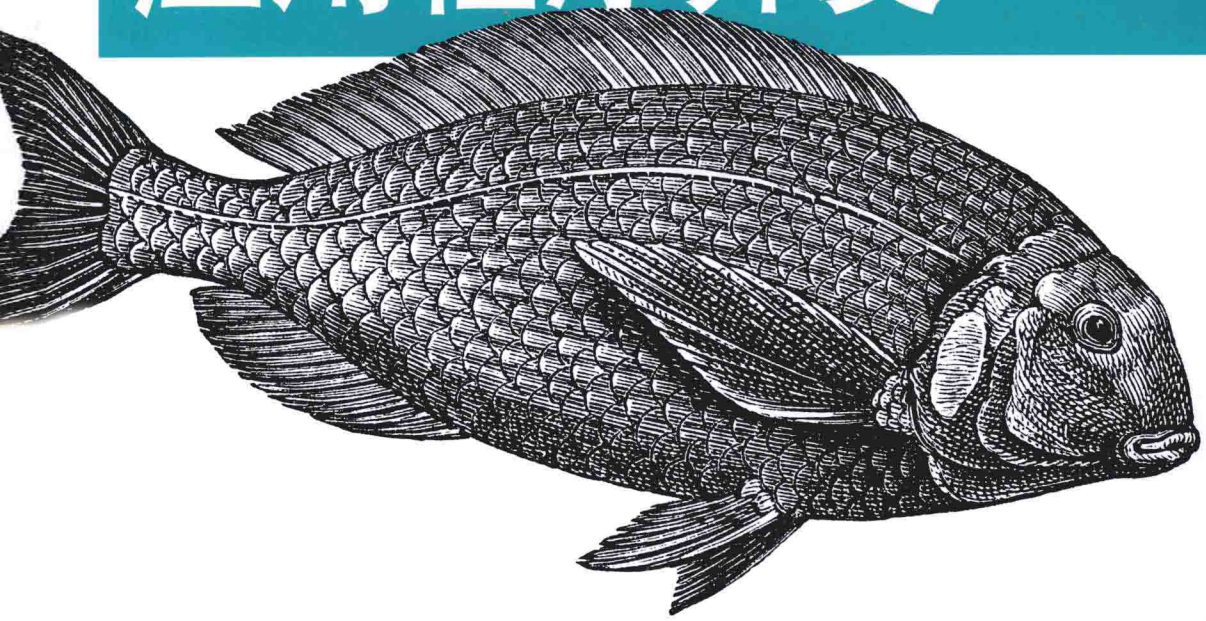
Developing Backbone.js Applications

构建卓越的JavaScript应用程序

Backbone.js 应用程序开发



[美] *Addy Osmani* 著

徐涛 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

Backbone.js应用程序开发

如果你想使用单页应用程序（SPA）模型创建前端站点，本书向你展示了如何使用Backbone.js完成这类工作。你将学会使用Backbone的模型-视图-控制器（MVC）架构，来创建结构化的JavaScript应用程序。

本书先从了解MVC、SPA和Backbone的基本知识开始，然后着手构建示例应用程序——一个简单的Todo列表应用程序、RESTful风格的图书应用程序以及使用Backbone和RequireJS的模块化应用程序。本书的作者是谷歌Chrome团队的工程师Addy Osmani，他还演示了框架的高级应用。

- 了解Backbone.js如何给客户端带来MVC方面的好处；
- 编写易于阅读的、结构化和易扩展的代码；
- 使用Backbone.Marionette和Thorax扩展框架；
- 解决使用Backbone.js时会遇到的常见问题；
- 使用AMD和RequireJS将代码进行模块化组织；
- 使用Backbone.Paginator插件为Collections数据分页；
- 使用样板代码引导新的Backbone.js应用程序；
- 使用jQuery Mobile，并解决两者之间的路由问题；
- 使用Jasmine、QUnit和SinonJS对Backbone应用进行单元测试。

Addy Osmani，谷歌Chrome团队的开发工程师，对JavaScript应用程序架构有着强烈的爱好。他创建了一些比较流行的项目，如TodoMVC，并对Yeoman、Modernizr和jQuery等其他开源项目也有重要贡献。Addy Osmani是一位高产的博主（*http://addyosmani.com/blog*），他也是O'Reilly出版的《JavaScript设计模式》一书的作者。

"编写Web应用程序是一个复杂的过程，但Osmani却能用Backbone.js将这些片段分解成简单、可管理的部分。本书为读者提供了成功设计、部署和测试复杂Web应用程序的基础和结构。"

——Samuel Clay
NewsBlur创始人

"要编写可扩展、可维护、富数据的Web应用程序，你需要使用Backbone.js。"

——Marc Friedman
美国康普公司
高级资深软件工程师

O'REILLY®
oreilly.com.cn

封面设计：Randy Comer，张健

O'Reilly Media, Inc.授权人民邮电出版社出版

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

分类建议：程序设计/JavaScript
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-35664-2
定价:55.00 元

O'REILLY®

Backbone.js应用程序开发

[美] Addy Osmani 著
徐 涛 译

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

Backbone.js应用程序开发 / （美） 奥萨姆（Osmani,A.）著 ; 徐涛译. -- 北京 : 人民邮电出版社, 2014.9
ISBN 978-7-115-35664-2

Ⅰ. ①B… Ⅱ. ①奥… ②徐… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第129857号

版 权 声 明

◆ 著 [美] Addy Osmani
译 徐 涛
责任编辑 陈冀康
责任印制 彭志环 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：787×1000 1/16
印张：21
字数：393 千字 2014 年 9 月第 1 版
印数：1 - 3 000 册 2014 年 9 月河北第 1 次印刷

著作权合同登记号 图字：01-2013-1021 号

定价：55.00 元

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

内容提要

Backbone.js 提供了一套 Web 开发的框架，为复杂的 JavaScript 应用程序提供了一个 MVC 结构。

本书详细介绍了如何使用 Backbone.js 完成 Web 应用开发。全书从了解 MVC、SPA 和 Backbone.js 的基本知识开始，然后着手构建 3 个示例应用程序。本书还介绍了 Backbone 和 Grunt-BBB、jQuery Mobile 等开发工具的配合使用，以及 Jasmine、QUnit 和 SinonJS 等测试解决方案。

本书的作者是知名的 JavaScript 专家、谷歌 Chrome 团队的工程师 Addy Osmani。本书适合于 JavaScript 程序员、Web 开发人员，尤其是想要学习和使用 Backbone.js 的读者阅读参考。

作者简介

Addy Osmani 是谷歌 Chrome 团队的开发工程师，对 JavaScript 应用程序架构有着强烈的爱好。

作为 Yeoman 团队中的一员，他创建了像 TodoMVC 这样流行的项目，并对 Modernizr 和 jQuery 等其他开放源代码项目也做出很大贡献。作为一位高产的博主（http://addyosmani.com/blog），Addy 的文章经常出现在《JavaScript Weekly》、《Smashing Magazine》及很多其他出版物上。

译者简介

徐涛（网名：汤姆大叔；微博：@TomXuTao），微软最有价值专家（MVP）、项目经理、软件架构师，擅长大型互联网产品的架构与设计，崇尚敏捷开发模式，熟悉设计模式、前端技术、以及各种开源产品，曾获 MCP、MCSE、MCDBA、MCTS、MCITP、MCPD、PMP 认证。《JavaScript 编程精解》、《JavaScript 启示录》译者，博客地址：Http://www.cnblogs.com/TomXu。

前言

不久以前，富数据的 Web 应用程序（data-rich web application）还是一个矛盾。如今，这些应用程序无处不在，因此我们需要知道如何构建它们。

通常，Web 应用程序将大量数据操作工作留给服务器，服务器以完整页面加载的方式将 HTML 推入到浏览器中。客户端 JavaScript 的使用仅限于改善用户体验。现在，这种关系已经被反转过来了——客户端应用程序从服务器获取原始数据，并可以随时随地根据需要在浏览器中进行渲染。

思考一下 Ajax 购物车，当添加一个商品到购物车时，不需要刷新页面。最初，jQuery 是这种范式的首选库。它的特性是让 Ajax 发出请求，然后更新页面上的文本等。然而，使用 jQuery 的这种模式透露出我们在客户端有隐式模型数据。

与服务器对话的客户端代码的兴起（却是很适合的）意味着客户端复杂性的增加。在客户端构建优良架构不再被置后考虑，而是变得非常必要——我们不能只编写一些 jQuery 代码，并期望它的规模能够随着应用程序的增长而扩大。我们很可能最终获得很多与业务逻辑纠缠在一起的杂乱的 UI 回调，并注定要被继承我们代码的可怜人所丢弃。

值得庆幸的是，现在有越来越多有助于改善代码结构和可维护性的 JavaScript 库，使我们能够更方便、轻松地构建功能强大的界面。Backbone.js 已经迅速成为最受欢迎的、用于解决这些问题的开源解决方案。在本书中，我将带领大家深入地了解它的用法。

我们将从基本概念开始，然后进行练习，并学习如何构建组织清晰且易维护的应用程序。如果你是一位希望编写更容易阅读、有组织性和可扩展性的代码的开发人员，本书可以帮你做到。

提高开发者的教育水准对我来说很重要，这也是这本书采用知识共享署名-非商业性使用-相同方式共享 3.0 Unported 许可协议出版的原因。这意味着大家可以购买或免费获得本书，也可以帮助进一步改进它。随时欢迎大家修正现有的材料，我希望我们能够共同为社区提供有用的最新资源。

我衷心感谢 Jeremy Ashkenas 和 DocumentCloud 创建了 Backbone.js，以及社区里的一些成员帮助我让这个项目更加完美。

译者注：

Jeremy Ashkenas：https://github.com/jashkenas

DocumentCloud：https://www.documentcloud.org/

社区贡献者：https://github.com/addyosmani/backbone-fundamentals/graphs/contributors

目标读者

本书面向初学者以及希望学习如何更好地组织客户端代码的中级开发人员。读者需要理解 JavaScript 基本原理才能好好利用它；但我只在必要时对这些概念进行基本的描述。

致谢名单

没有社区其他开发人员和其他写作者为本书投入的时间和精力，就不可能有本作品的问世。我要衷心感谢：

Marc Friedman https://github.com/dcmaf

Derick Bailey https://github.com/derickbailey

Ryan Eastridge https://github.com/eastridge

Jack Franklin https://github.com/jackfranklin

David Amend https://github.com/raDiesle

Mike Ball https://github.com/mdb

Uģis Ozols https://github.com/ugisozols

Bjorn Ekengren https://github.com/Ekengren

也要感谢其他优秀的贡献者，是他们让这个项目变成可能。

其他读物

我假定你在JavaScript方面的水平已经超越初级水平了，因此，我跳过了某些主题，比如对象字面量。如果想要深入了解该语言，我很乐意向你推荐以下书目：

《JavaScript 编程精解》

《JavaScript 权威指南》，作者：David Flanagan (O'Reilly)

《Effective JavaScript》，作者：David Herman (Pearson)

《JavaScript 语言精粹》，作者：Douglas Crockford (O'Reilly)

《JavaScript 面向对象编程指南》，作者：Stoyan Stefanov (Packt Publishing)

本书约定

本书使用下列排版约定：

斜体（*Italic*）

表示专业词汇、链接（URL）、文件名和文件扩展名。

等宽字体（Constant width）

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

等宽粗体（Constant width bold）

表示应该由用户按照字面引入的命令或其他文本。

等宽斜体（Constant width italic）

表示应该由用户替换或取决于上下文的值。


这个图标表示提示、建议或一般说明。


这个图标表示警告或提醒。

代码示例

这本书是为了帮助你顺利完成工作而写的。你可以在程序和文档中使用本书的代码。只要不是大规模复制本书中的代码，就不需要联系我们获取许可。例如，使用本书中的几段代码写一个程序不需要许可。出售和分发 O’Reilly 书中用例的光盘（CD-ROM）是需要许可的。通过引用本书用例和代码来回答问题不需要许可，把本书中大量的用例代码并入到你的产品文档中则需要许可。

我们希望但不强求注明信息来源。一条信息来源通常包括标题、作者、出版者和国际标准书号（ISBN）。例如：“*Developing Backbone.js Applications* byAdnan Osmani (O’Reilly). Copyright 2013 Addy Osmani, 978-1-449-32825-2.”。

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

Safari 在线图书

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

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

Safari 在线图书为组织、政府机构和个人提供了一系列的产品组合和定价计划。用户可以在一个来自各个出版社的完全可搜索的数据库中访问成千上万的书籍、培训视频和正式出版前的手稿，这些出版社包括：O’Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。欲获得有关 Safari 在线图书的更多信息，请在线访问我们。

联系我们

如果你对本书有意见和问题，请联系出版社：

美国：

O’Reilly Media，Inc.

1005 Gravenstein Highway North

Sebastopol，CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们已将本书做成一个网页，我们在那里列出了勘误表、示例和额外信息。你可以打开 http://oreil.ly/dev_backbone_js_apps 访问这个页面。

如需对本书发表评论或提出技术问题，请发送电子邮件至：bookquestions@oreilly.com。

欲获得有关本书、课程、会议及新闻的更多信息，请访问我们的网站：http://www.oreilly.com。

你还可以：

在 Facebook 上找到我们：http://facebook.com/oreilly

在 Twitter 上追踪我们：http://twitter.com/oreillymedia

在 YouTube 上关注我们：http://www.youtube.com/oreillymedia

致谢

我要感谢技术校对人员，他们的出色工作帮助我改进了这本书。他们的知识、精力和热情让本书成为一个更好的学习资源，他们会继续成为我创作的灵感源泉。感谢：

Derick 和 Marc	再次感谢
Jeremy Ashkenas	https://github.com/jashkenas
Samuel Clay	https://github.com/samuelclay
Mat Scales	http://github.com/wibblymat
Alex Graul	https://github.com/alexgraul
Dusan Gledovic	https://github.com/g6scheme
Sindre Sorhus	https://github.com/sindresorhus

我还要感谢我亲爱的家人在我写这本书的时候所表现出的耐心和支持，以及本书的优秀编辑 Mary Treseler。

目录

第1章

概　述

建筑大师 Frank Lloyd Wright 曾经说过：“你虽然当不成建筑师，但你仍然可以随心所欲地打开门窗，让阳光照进来。”在这本书中，我将在如何改进 Web 应用程序的结构方面给出一些启示，以便大家将来能够编写出更易维护、可读的应用程序。

所有架构的目的都是建立一个完美框架——对我们来说，是编写经久不衰的代码，让我们以及在我们之后维护代码的开发人员感到满意。我们都希望架构简单而美观。

现代 JavaScript 框架和库可以把结构性和组织性融入到项目中，从一开始就建立起一个易维护的基础模式。开发人员历尽艰辛、反复试验，解决我们现在或将来可能会遇到的回调混乱难题，才成功创建了这些框架和库。

仅使用 jQuery 来开发应用程序时，缺少的是构造和组织代码的方式。创建结束 jQuery 选择器和回调函数混乱状态的 JavaScript 应用程序很容易，只需努力保持 UI 的 HTML、JavaScript 中的逻辑和对 API 的数据调用之间的数据同步即可。

如果没有清理混乱的方法，很可能要把一组独立的插件和库串起来，以弥补功能缺失，或者重新开始构建，而且必须自己进行维护。Backbone 可以解决这个问题，它提供了一种整洁地组织代码的方式，并把任务分离成可识别、且易于维护的片段。

在本书中，我和其他几位经验丰富的作者将向大家展示如何使用流行的 JavaScript 库的 Backbone.js 1.0 版本，来改进 Web 应用程序的结构。

1.1 什么是 MVC

很多现代 JavaScript 框架为开发人员提供使用 MVC（Model-View-Controller，模型-视图-控制器）模式的变体来轻松组织代码的方法。MVC 将应用程序中的关注点分为以下 3 个部分。

- 模型（Model）代表了应用程序中的特定领域知识和数据。可以将它想象为一个可以建模的数据类型，如一个用户、一张图片或备忘录。模型在其状态改变时可以通知其观察者。
- 视图（View）通常用于在应用程序中构成用户界面（如标记和模板），但不是必须的。它们观察着模型，但并不与它们直接通信。
- 控制器（Controller）用于处理输入（单击或用户操作）以及更新模型。

因此，在 MVC 应用程序中，用户的输入受控制器的支配，而控制器更新模型。视图观察模型，并在模型发生更改时更新用户界面。

然而，JavaScript MVC 框架并不总是严格遵循这种模式。一些解决方案（包括 Backbone.js）将控制器的任务合并到了视图，而其他的方法也混入到额外的组件中。

为此，我们称这种框架遵循了 MV*模式，也就是说，我们可能会有模型和视图，但唯独控制器可能不存在，其他组件可能会发挥控制器的作用。

1.2 什么是 Backbone.js

Backbone.js（见图 1-1）是一种轻量级的 JavaScript 库，用于将结构添加到客户端代码。它可以让应用程序的关注点管理和解耦工作变得很容易。从长远来看，它可以使代码更易于维护。

开发人员通常使用像 Backbone.js 这样的库来创建单页面应用程序（SPA）。SPA 是一种将页面加载到浏览器，然后不需要从服务器刷新整个页面，就可以在客户端完成数据交互的 Web 应用程序。

Backbone 非常成熟和流行，活跃的开发者社区和基于它的大量可用插件和扩展都离不开它。它已经被 Disqus、沃尔玛、SoundCloud 和 LinkedIn 等公司用于开发重要的应用程序。

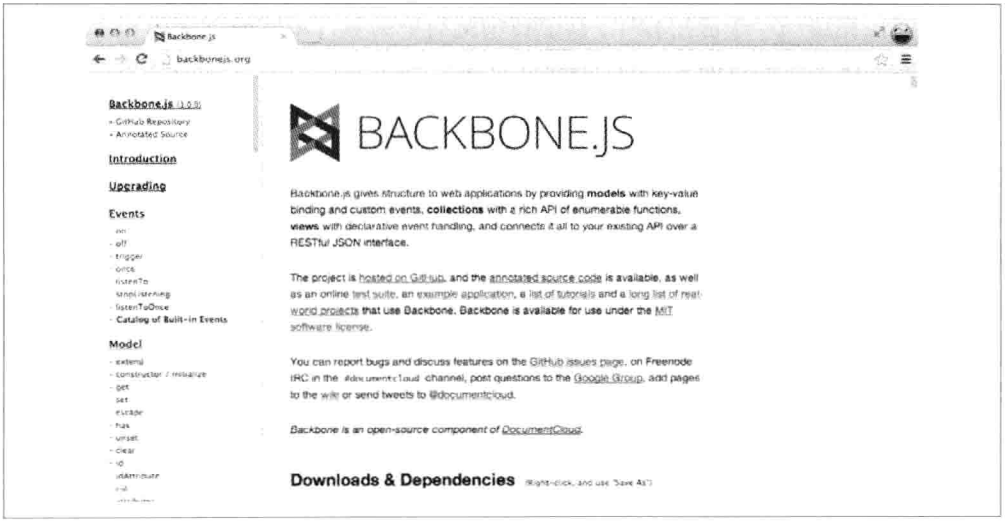


图 1-1　Backbone.js 主页

Backbone 关注于向用户提供查询和操作数据的有用方法，而不是重建 JavaScript 对象模型。它是一个库，而不是框架，从嵌入式小部件到大规模应用，它都具有很好的可伸缩性和兼容性。

由于 Backbone 很小，还有一些用户将它下载到手机中使用，或者用于改善很慢的链接速度。Backbone 的所有源码可以在短短几小时内就阅读并理解完。

1.3　何时需要 JavaScript MVC 框架

使用 JavaScript 构建一个单页面应用程序时，不管它是否包括一个复杂的用户界面，还是只是试图减少新视图所需的 HTTP 请求数量，我们可能会发现自己其实已经创建了 MV*框架中的许多部分。

刚开始时，编写能够提供一些特殊方法来避免“意大利面条”式代码的应用程序框架非常困难。但是，如果说编写一个与 Backbone 一样强大的代码也同样繁琐，那就会是一个非常错误的观念。

MV*框架在构建应用程序方面还有更深入的内容，而不只是捆绑 DOM 操作库、模板和路由。成熟的 MV*框架通常不仅包括我们找到的自己编写的部分，还包括之后所找到的某些问题的解决方案。这是一个节省时间的事物，我们不应低估它的价值。

所以，你可能会在哪里需要 MV*框架，在哪里不需要？

如果我们正在编写一个应用程序，其中在浏览器中会有许多的视图渲染和数据操作工作，那么这时我们可能会发现 JavaScript MV*框架很有用处。这类应用程序包括 Gmail、NewsBlur 和 LinkedIn 移动应用程序等。

这些类型的应用程序通常下载一个完整的加载，其中包含所有的脚本和样式表，以及用户需要处理常见事情并在幕后执行大量附加行为的 HTML 标记。例如，无需向服务器发送一个新的页面请求，就能轻松将阅读 E-mail 文档切换到编写 E-mail 文档。

然而，如果你正在构建的应用程序仍然依赖于服务器完成大部分的页面/视图渲染工作，而且你仅仅使用一些 JavaScript 或 jQuery 来让一切更具有互动性，那么 MV* 框架可能就大材小用了。当然，也有一些复杂的 Web 应用程序，其视图的分部渲染可以与单页面应用程序进行有效的结合，但在其他方面，我们最好坚持使用简单的设置。

软件（框架）开发的成熟并不取决于一个框架已经存在了多久；而取决于这个框架有多坚固，更重要的是，它是否能够很好地发挥它的作用。在解决常见问题方面，它是否变得更有效呢？在开发人员构建更大、更复杂的应用程序时，它是否在继续改进呢？

1.4 为何考虑 Backbone.js

Backbone 提供了一组数据结构（模型、集合）和用户界面（视图、URL），当使用 JavaScript 构建动态应用程序时，它们非常有用。这不是固执己见，它意味着我们可以根据需要自由、灵活地构建 Web 应用程序的最佳体验。可以使用它提供的开箱即用的特定架构或者对它进行扩展，以满足需求。

该库并不关注小部件或替换我们构建对象的方式，它只提供用于操作和查询应用程序数据的工具。它也没有指定特定的模板引擎；我们可以自由使用 Underscore.js（它的依赖项之一）提供的微模板，视图可以绑定到所选择的模板解决方案构建的 HTML 上。

当我们看到大量使用 Backbone 构建的应用程序后，可以知道它具有良好的可扩展性，这点很明显。Backbone 也能够很好地与其他库兼容，这意味着可以在使用 AngularJS 编写的应用程序中嵌入 Backbone 小部件，可以将它和 TypeScript 一起使用，或仅使用单个类（如模型）作为简单应用程序的数据支持者。

使用 Backbone 构建应用程序没有性能上的缺点。它避免了运行循环、双向绑定以及持续查询数据结构时进行更新，它让事情尽可能的简单化。即便如此，你希望背

道而驰吗？当然，你可以在它上面执行这样的操作。Backbone 是不会阻止你的。

在一个有众多插件和扩展作者加入的充满活力的社区里，如果想获得一些Backbone缺乏的功能，可能会在里面找到与之相关的互补项目。此外，Backbone 提供了源代码的读写文档，任何人都有机会轻松了解其幕后的工作。

经过两年半的不断改进，Backbone 库变得成熟起来，它将继续为构建更好的 Web 应用程序提供一种极简主义的解决方案。我经常使用它，希望大家像我一样，把它当作自己的另外一个工具。

1.5 设定预期目标

本书的目的是创建一个权威和集中式信息存储库，来帮助那些使用 Backbone 开发现实世界应用程序的开发人员。如果你在本书中发现你认为可以改进或扩展的部分或主题，请在本书的 GitHub 网站上提交问题（或者 Pull Request 更好）。它不会花费你很长时间，而且这样做会帮助其他开发人员避免遇到这类问题。

译者注：

GitHub 网站：https://github.com/addyosmani/backbone-fundamentals

本书的主题包括 MVC 理论，以及如何使用 Backbone 的模型、视图、集合和路由来构建应用程序。我还将带领大家了解一些高级主题，如使用 Backbone.js 和 AMD（通过 RequireJS）进行模块化开发，嵌套视图等常见问题的解决方案，以及如何使用 Backbone 和 jQuery Mobile 解决路由问题等。

下面是每章要学习的内容。

第 2 章 基本概念

追溯 MVC 设计模式的历史，并介绍 Backbone.js 和其他 JavaScript 框架是如何使用实现它的。

第 3 章 Backbone 基础

涵盖了 Backbone.js 的主要功能，以及大家需要知道的技术和技巧，以便能够有效地使用它。

第 4 章 练习 1：Todos——第一个 Backbone.js 应用程序

带你一步步了解简单客户端 Todo 列表应用程序的开发。

第 5 章 练习 2：Book Library——第一个 RESTful 风格的 Backbone.js 应用程序

了解图书馆应用程序的开发，该程序使用 REST API 将其模型持久化到服务器上。

第 6 章 Backbone 扩展

描述了两个扩展框架 Backbone.Marionette 和 Thorax，并向 Backbone.js 添加了用于开发大型应用程序的特性。

第 7 章 常见问题和解决方案

回顾使用 Backbone.js 时可能遇到的常见问题，以及解决这些问题的方法。

第 8 章 模块化开发

了解如何使用 AMD 模块和 RequireJS 进行模块化开发。

第 9 章 练习 3：第一个模块化的 Backbone/RequireJS 应用程序

了解在 RequireJS 的帮助下，重写在练习 1 中创建的应用程序，使其更加模块化。

第 10 章 对 Backbone.js 请求和集合进行分页

了解如何使用 Backbone.Paginator 插件对集合进行数据分页。

第 11 章 Backbone Boilerplate 和 Grunt-BBB

介绍强大的工具，通过使用样板代码来开始一个新 Backbone.js 应用程序。

第 12 章 Backbone 和 jQuery Mobile

解决 Backbone 和 jQuery Mobile 一起使用时出现的问题。

第 13 章 Jasmine

介绍如何利用 Jasmine 测试框架对 Backbone 代码进行单元测试。

第 14 章 QUnit

讨论如何使用 QUnit 进行单元测试。

第 15 章 SinonJS

讨论如何使用 SinonJS 对 Backbone 应用程序进行单元测试。

第 16 章 结论

结束 Backbone.js 开发旅程。

附录 A 延伸学习

重新回到设计模式的讨论，对比 MVC 模式与 MVP（模式-视图-表示器）模式，并了解 Backbone.js 如何与两者相关，还包括了解如何从零开始编写像 Backbone 这样的库，以及其他一些主题。

附录 B 资源

提供 Backbone 相关的额外资源引用。

第 2 章

基本概念

设计模式是常见开发问题中经验证的解决方案，它可以帮助我们改善应用程序的组织和结构。通过使用设计模式，我们能够从重复解决类似问题的高级开发人员的集体经验中受益。

历史上，开发人员在创建桌面和服务器级应用程序时，有大量的设计模式可以依赖，但这种模式在前几年才应用到客户端开发上。

在本章中，我们将要探索 MVC（Model-View-Controller）设计模式的发展，让我们首先来了解一下 Backbone.js 如何让我们能够将这种模式应用到客户端开发。

2.1 MVC

MVC 是一种架构设计模式，鼓励通过关注点分离改进应用程序的组织。它强制将业务数据（模型）从用户界面（视图）处隔离，第三个组件（控制器）仍管理逻辑、用户输入和模型与视图的协调。该模式最初由 Trygve Reenskaug 在写作《Smalltalk-80》（1979 年）时提出，它最初被称为 Model-View-Controller-Editor。“四人组”在 1994 年出版的《设计模式：可复用面向对象软件的基础》对 MVC 进行了深入描述。这本书对 MVC 的推广和使用起重要作用。

2.1.1 Smalltalk-80 MVC

重要的是，我们要了解最初的 MVC 模式要解决的问题，因为它自诞生以来已经发生了很大的变化。早在 20 世纪 70 年代的时候，图形用户界面非常稀少，一种被称为分离表现的方法开始用于在域对象和表示对象之间做明确区分。域对象用于在现实世界模拟概念（如照片、一个人），表示对象用于渲染到用户的屏幕。

MVC 的 Smalltalk-80 将这个概念进一步升华，目标是从用户界面分离出应用程序逻辑。其想法是：解耦应用程序的这部分内容，实现在应用程序中对其他接口进行模型重用。Smalltalk-80 的 MVC 架构有一些值得注意的有趣地方。

- 领域元素被称为模型，对用户界面（视图和控制器）一无所知。
- 视图和控制器负责表示层，但不只是单个视图和控制器——每个显示在屏幕上的元素都需要一个视图控制器对，所以它们之间没有真正的分离。
- 在这个视图控制器对中，控制器的作用是处理用户的输入（如按键和单击事件）并做一些合理的事情。
- 当模型改变时，观察者模式用于更新视图。

在开发人员得知观察者模式（现在通常实现为发布/订阅系统）在几十年前就作为 MVC 架构的一部分之后，他们有时会感到很惊讶。在 Smalltalk-80 的 MVC 中，视图和控制器都观察模型：当模型发生变化时，视图会有反应。一个简单的示例是关于股票市场数据的应用程序：让应用程序显示实时信息，模型中的任何数据修改都应使视图立即刷新。

Martin Fowler 的关于 MVC 起源的文章写得很棒（http://martinfowler.com/eaaDev/uiArchs.html），所以如果你有兴趣对 Smalltalk-80 MVC 的历史信息进行进一步了解，推荐大家阅读他的作品。

2.1.2 MVC 应用于 Web

Web 在很大程度上依赖于 HTTP 协议，它是无状态的，这意味着在浏览器和服务器之间没有一直开放的连接，每个请求都要在两者之间实例化一个新通信信道。一旦请求发起人（如浏览器）获得一个响应，连接就会关闭。与很多使用 MVC 原始思想构建的操作系统相比，这将催生出一个完全不同的上下文。MVC 的实现必须符合 Web 上下文。

在服务器端 Web 应用程序框架中，试图将 MVC 应用于 Web 上下文的一个示例就是 Ruby on Rails 框架，如图 2-1 所示。

该框架的核心是我们预料到的 3 个 MVC 组件：模型、视图、控制器。

- 模型表示应用程序中的数据，通常用于管理与特定数据库表交互的规则。通常一个表对应一个模型，应用程序中的大量业务逻辑都位于这些模型内。

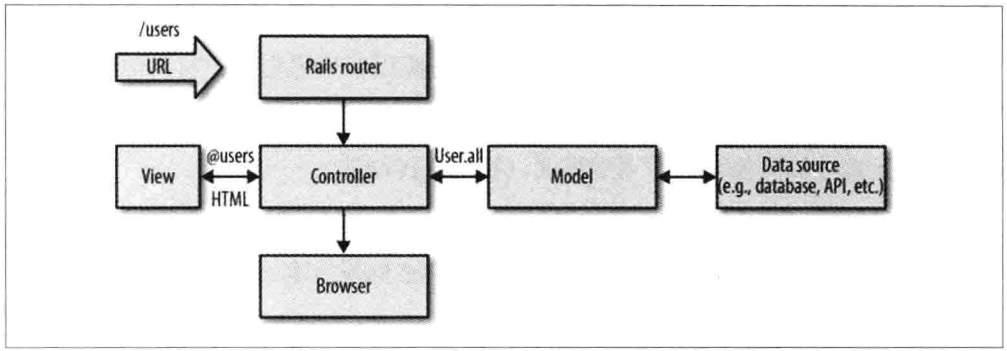


图 2-1　Ruby on Rails 框架

- 视图表示用户界面，通常采用 HTML 的形式被发送到浏览器，用于将应用程序数据展示给任何从应用程序发出请求的组件。
- 控制器在模型和视图之间充当黏合剂。它们的职责是处理来自浏览器的请求，通知模型提供数据，然后将数据提供给视图，以便将它们渲染在浏览器上。

虽然在 Rails 中有像 MVC 一样清晰的分离关注点，但它实际上是使用另一个被称为 Model2 的模式，因为 Rails 不在模型中通知视图，并且控制器只是将模型数据直接传递给视图。

这就是说，即便是从 URL 接收请求的服务器端工作流，将产生的 HTML 页面作为响应，并从界面中分离业务逻辑也是有很多好处的。以相同的方式，在服务器端框架中将 UI 从数据库记录中分离出来是非常有用的，同样，对于在 JavaScript 中从数据模型处分离 UI 也是有用的（稍后我们将了解更多相关内容）。

MVC 的其他服务器端实现（比如 PHP Zend 框架）也实现了前端控制器设计模式。这种模式在单一入口点后面放置一个 MVC 堆栈。这种单一入口点意味着所有 HTTP 请求（例如 http://www.example.com、http://www.example.com/whicheverpage/ 等）不依赖 URI，由服务器配置将其路由到相同的处理程序。

当前端控制器接收 HTTP 请求时，它会分析请求，并决定调用哪个类（控制器）和哪个方法（动作）。选定的控制器 action 接管并与相应的模型进行交互来完成请求。控制器接收从模型返回的数据，加载一个适当的视图，然后将模型数据注入到其中，并将响应返回到浏览器。

例如，假设我们在 www.example.com 上有个博客程序，我们想编辑一篇文章（id=43），然后请求 http://www.example.com/article/edit/43。

在服务器端，前端控制器将分析 URL 并调用 article 控制器（对应于 URI 的*/article/*部分）以及其编辑动作（对应于 URI 的*/edit/*部分），在该动作内会对文章模型及其 Articles::getEntry(43)方法（43 对应 URI 末尾的*/43*）进行调用，并将从数据库中返回需要编辑的博客文章数据。article 控制器将加载（*article/edit*）视图，包括注入文章数据的逻辑，以便将数据放置到一个合适的表单中，用于编辑内容、标题以及其他（元）数据。最后，生成的 HTML 响应将被返回到浏览器。

可以想象，在表单中单击“保存”按钮后，发送的 POST 请求也需要一个类似的流程。POST 动作的 URI 看起来类似*/article/save/43*。该请求将通过相同的控制器，但是这次 save 动作将被调用（由于是*/save/* URI），文章模型使用 Articles::saveEntry(43)将编辑后的文章保存到数据库，浏览器将被重定向到*/article/edit/43* 的 URI，以进行进一步的编辑。

最后，如果用户请求 *http://www.example.com/*，前端控制器将调用默认的控制器和动作（例如，index 控制器及它的 index 动作）。在 index 动作内，将对文章模型的 Articles::getLastEntries(10)方法进行调用，该方法将返回最新的 10 篇博客文章。控制器将加载 blog/index 视图，该视图内含有列出博客文章的基本逻辑。

图 2-2 所示显示了服务器端 MVC 中的典型 HTTP 请求/响应生命周期。

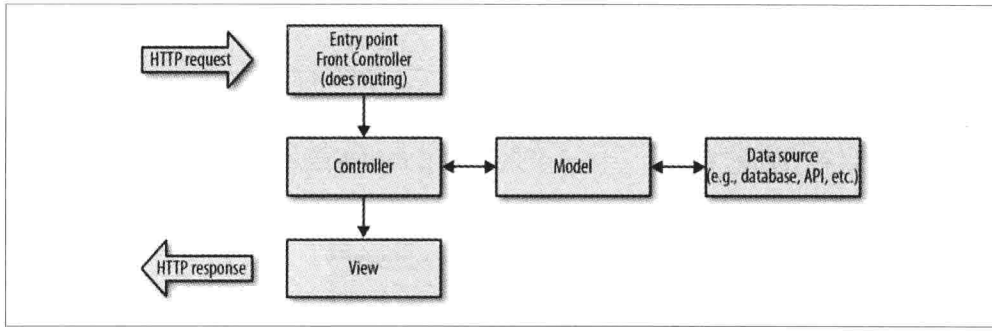


图 2-2　服务器端 MVC 中的 HTTP 请求/响应生命周期

服务器接收到一个 HTTP 请求，并通过一个单一入口点对它进行路由。在该入口点，前端控制器分析请求，并调用适当控制器的动作。这个过程称为路由（*routing*）。该动作模型要求返回并（或者）保存提交的数据。模型与数据源（例如数据库或 API）进行通信。一旦模型完成工作，就会把数据返回到控制器，然后控制器加载合适的视图，视图再利用其提供的数据执行表示层逻辑（循环文章，并输出标题、内容等）。最后，HTTP 响应返回到浏览器。

2.1.3 客户端 MVC 和单页面应用程序

多项研究已经证实，延迟改进对网站和应用程序的使用以及用户参与都有积极的影响。这与传统的 Web 应用程序开发方法不同，传统方法以服务器为中心，需要重新加载一个完整的页面才能完成从一个页面到下一个页面的过程。即使有大量缓存，浏览器仍然需要解析 CSS、JavaScript、HTML，并把界面渲染到屏幕上。

除了导致大量的重复内容返回给用户以外，这种方法还影响了延迟和用户体验的整体响应能力。在过去几年，改进感知延迟的一个趋势是创建单页面应用程序（SPA）——在初始页面加载后，应用程序能够处理后续导航和数据请求，而不需要重新加载页面。

当用户导航到一个新视图时，应用程序使用 XHR（XMLHttpRequest）请求视图所需的额外内容，其通常与服务器端 REST API 或终端进行通信。Ajax（异步 JavaScript 和 XML 的简称），与服务器进行异步通信，以便数据可以在后端进行传输并处理，可以让用户能够在没有交互作用的情况下使用页面的其他部分。这可以改进可用性和响应性。

从一个视图移到另一个视图时，SPA 还可以利用浏览器的特性，如 HistoryAPI，来更新显示在地址栏里的地址。这些 URL 还可以让用户制作书签并分享特定的应用程序状态，而不需要导航到全新的页面。

典型的 SPA 由代表逻辑实体的接口小块组成，它们都有自己的 UI、业务逻辑和数据。一个很好的示例是 Web 商城应用程序中的购物车，可以向其中添加商品。这个购物车可以在页面的右上角，以一个盒子的形式呈现给用户（见图 2-3）。

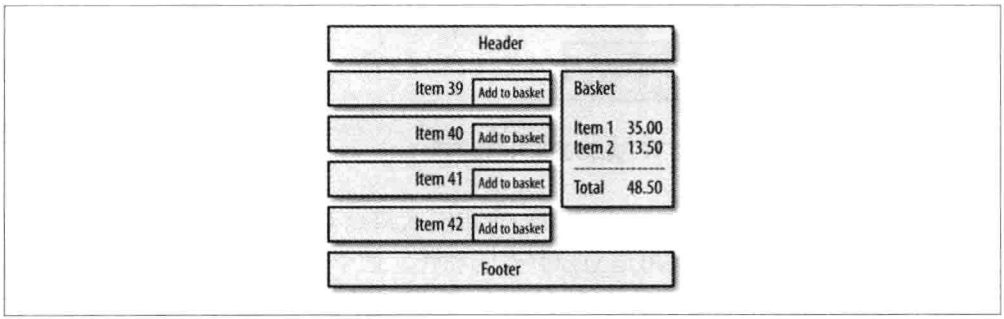


图 2-3 单页面应用程序中的购物车形式

购物车及其数据以 HTML 的方式呈现。HTML 中的数据及其相关的视图随时间变化而变化。我们曾经使用过 jQuery（或类似的 DOM 操作库）和一堆 Ajax 调用与

回调函数来保持两者同步，但这通常会导致生成结构欠佳或难以维护的代码，并会经常产生 Bug，甚至不可避免。

对快速、复杂、响应式的 Ajax Web 应用程序的需求，导致需要在客户端复制大量的这种逻辑，这极大地增加了驻留在客户端代码的大小和复杂性。最终，这将导致我们需要在客户端实现 MVC（或类似架构），以便更好地组织代码，并在应用程序的生命周期内更容易维护和扩展它。

经过不断发展和反复试验，JavaScript 开发人员已经强化了传统 MVC 模式的功能，同时也开发了多个使用 MVC 的 JavaScript 框架，如 Backbone.js。

2.1.4 客户端 MVC：Backbone 风格

让我们首先来看看，Backbone.js 是如何利用示例中的 Todo 应用程序，将 MVC 的好处带给客户端开发的。在接下来的内容探索 Backbone 的特性时，我们会将这个示例作为构建基础，但现在我们只把重点放在核心组件与 MVC 的关系上。

我们的示例将需要一个 div 元素，以便可以附加一系列的 todo 项；还将需要一个 HTML 模板，该模板包含 todo 项标题和标记完成的复选框。通过以下 HTML 提供：

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
</head>
<body>
  <div id="todo">
  </div>
  <script type="text/template" id="item-template">
    <div>
      <input id="todo_complete" type="checkbox" <%= completed ?
       'checked="checked"' : '' %>>
      <%- title %>
    </div>
  </script>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>
  <script src="demo.js"></script>
</body>
</html>
```

在 todo 应用程序（demo.js）中，Backbone 模型的实例用于保存每个 todo 项的数据：

```
// Define a Todo model
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Instantiate the Todo model with a title, allowing completed attribute
// to default to false
var myTodo = new Todo({
  title: 'Check attributes property of the logged models in the console.'
});
```

todo 模型扩展了 Backbone.Model，简单地为两个数据属性定义了默认值。你会发现在接下来的章节里，Backbone 模型提供了更多的功能，但是这个简单的模型表明了一件最重要的事——模型首先是一个数据容器。

每个 Todo 实例都将通过 TodoView 在页面上进行渲染：

```
var TodoView = Backbone.View.extend({

  tagName:  'li',

  // Cache the template function for a single item.
  todoTpl: _.template( $('#item-template').html() ),

  events: {
    'dblclick label': 'edit',
    'keypress .edit': 'updateOnEnter',
    'blur .edit':   'close'
  },

  // Called when the view is first created
  initialize: function () {
    this.$el = $('#todo');
    // Later we'll look at:
    // this.listenTo(someCollection, 'all', this.render);
    // but you can actually run this example right now by
    // calling TodoView.render();
  },

  // Rerender the titles of the todo item.
  render: function() {
    this.$el.html( this.todoTpl( this.model.toJSON() ) );
    // $el here is a reference to the jQuery element
    // associated with the view, todoTpl is a reference
    // to an Underscore template and toJSON() returns an
    // object containing the model's attributes
    // Altogether, the statement is replacing the HTML of
    // a DOM element with the result of instantiating a
```

```
    // template with the model's attributes.
    this.input = this.$('.edit');
    return this;
  },

  edit: function() {
    // executed when todo label is double-clicked
  },

  close: function() {
    // executed when todo loses focus
  },

  updateOnEnter: function( e ) {
    // executed on each keypress when in todo edit mode,
    // but we'll wait for enter to get in action
  }
});

// create a view for a todo
var todoView = new TodoView({model: myTodo});
```

我们通过扩展 Backbone.View 定义 TodoView，并用一个关联的模型对它进行实例化。在示例中，render()方法使用模板来构建 todo 项的 HTML 内容，该 HTML 内容被放置在 li 元素里面。每一次调用 render()，都会使用当前模型的数据来替换 li 元素中的内容。因此，视图实例使用相关模型的属性来渲染 DOM 元素中的内容。稍后我们将会看到视图如何将其 render()方法绑定到模型更改事件，以便在模型发生变化时视图能够重新渲染。

到目前为止，我们已经看到，Backbone.Model 实现了 MVC 的模型，Backbone.View 实现了视图。然而，正如我们前面所指出的，在控制器方面，Backbone 偏离传统的 MVC，没有 Backbone.Controller。

相反，控制器的责任在视图内得到了处理。回想一下，控制器响应请求并执行适当的操作，这可能会导致模型修改和视图更新。在 SPA 中有事件，而不是传统意义上的请求。事件可以是传统的浏览器 DOM 事件（如单击）或内部应用程序事件（如模型变更）。

在 TodoView 中，events 属性承担控制器配置的角色，定义了在视图 DOM 元素内发生的事件如何被路由到视图中所定义的事件处理方法上。

而在这个示例中，事件帮助我们把 Backbone 关联至 MVC 模式，我们将看到事件在 SPA 应用程序中担任了更大的角色。Backbone.Event 是 Backbone 的一个基础组件，它混入到了 Backbone.Model 和 Backbone.View 中，为它们提供了丰富的事件管理能力。注意，传统的视图角色（Smalltalk-80-style）由模板担任，而不是

Backbone.View。

这就完成了与 Backbone.js 的第一次接触。本书的其余部分将探讨该框架的很多其他特性。在继续之前，让我们看看 JavaScript MV *框架的常见功能。

2.1.5 实现规范

SPA 通过一个正常的 HTTP 请求和响应加载到浏览器。该页面可能仅仅是一个 HTML 文件（就像前面的例子），也可能是一个由服务器端 MVC 实现所构建的视图。

SPA 一旦加载，客户端路由就会截获 URL 并调用客户端逻辑，而不是发送新的请求到服务器。图 2-4 显示了由 Backbone 实现的客户端 MVC 的典型请求处理。

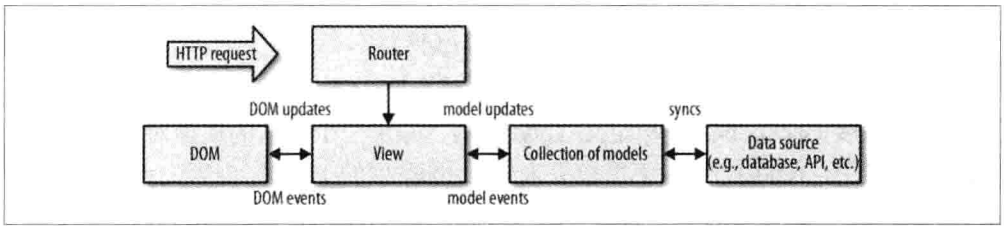


图 2-4 Backbone 处理请求的方法

URL 路由、DOM 事件（如鼠标单击）和模型事件（如属性更改）都在视图中触发处理逻辑。这些处理程序更新 DOM 和模型，并且也可能会触发其他事件。模型与数据源进行同步，可能会涉及与后端服务器的通信。

1. 模型

- 在不同的框架中，模型的内置功能也不同；然而，它们一般都支持属性验证，属性（attributes）表示模型的属性（properties），比如模型标识符。
- 在实际应用程序中使用模型时，我们通常也需要一种持久化模型。持久化允许我们在知道所保存模型的最新状态下，对其进行编辑和更新，例如，在 Web 浏览器的 localStorage 数据源中保存，或者与数据库进行同步。
- 一个模型可能会有多个视图在观察它的变化。通过观察，每当模型更新时，视图就可以获得通知，使得视图能够确保将屏幕上显示的内容与模型中的数据保持同步。根据你的需求，你可以创建一个显示所有模型属性的视图，也可以创建多个单独的视图来显示不同的属性。最重要的是，该模型不会关注这些视图的组织方式，只是在必要时通过框架的事件系统宣布它的数据进行了更新。

- 现代 MVC/MV*框架所提供的一种将模型分组在一起的方式是不常见的。在 Backbone 中，这些分组被称为集合。在分组中管理模型，可以让我们在组内模型发生变化时，根据来自组的通知来编写应用程序逻辑，避免了手动观察单个模型实例的麻烦。我们将在本书后面部分看到这个操作。集合还可以在多个模型上执行任何聚合操作。

2. 视图

- 用户与视图进行交互，通常意味着读取或修改模型数据。例如，在 Todo 应用程序中，Todo 模型的查看是在所有任务列表的用户界面上发生的。在列表内部，每个 todo 都渲染自己的标题并完成复选框。模型的修改是用户通过选择特定 todo 的标题后在编辑视图中完成的。

- 在视图中定义 render()方法，它负责使用 JavaScript 模板引擎（由 Underscore.js 提供）渲染模型的内容，并更新视图的内容，这些内容可以由 this.el 进行引用。

- render()回调函数作为模型的构造函数进行添加，以便模型改变时触发视图更新。

- 你可能想知道用户交互在什么地方发挥作用。当用户单击视图中的 todo 元素时，视图没有责任知道下一步要做什么。控制器做这个决定。在 Backbone 中，我们通过在 todo 元素上添加事件监听器来完成该操作，todo 元素将会把单击事件委托给事件处理程序。

3. 模板

在支持 MVC/MV*的 JavaScript 框架的上下文中，值得仔细了解是的 JavaScript 模板以及它与视图的关系。

通过字符串连接，手动在内存中创建大量的 HTML 标记，一直被认为是糟糕的实践（计算起来成本很高）。使用这种技术的开发人员经常发现要自己遍历数据，并把它封装在嵌套的 div 中，然后使用 document.write 等过时的技术将模板注入到 DOM 中。这种方法通常意味着，在标准标记内保持脚本标记很快就变得难以阅读和维护，特别是在构建大型应用程序时。

JavaScript 模板库（比如 Mustache 或 Handlebars.js）通常将视图模板定义为包含模板变量的 HTML 标记。这些模板可以保存在自定义类的<script>标记外部或内部（例如文本/模板）。变量通过变量语法分隔（例如，Underscore 的<%= title %>、Handlebars

的{{title}}）。

JavaScript 模板库通常以多种格式接收数据，包括 JSON——始终是一个字符串的序列化格式。数据填充模板的繁重工作通常由框架本身完成。这有很多好处，尤其是当我们选择在外部存储模板时，使得应用程序能够按需动态加载模板。

让我们来比较一下 HTML 模板的两个例子：一个是使用流行的 Handlebars.js 库，另一个是使用 Underscore.js 的微模板。

Handlebars.js

```
<div class="view">
  <input class="toggle" type="checkbox" {{#if completed}} "checked" {{/if}}>
  <label>{{title}}</label>
  <button class="destroy"></button>
</div>
<input class="edit" value="{{title}}">
```

Underscore.js 微模板

```
<div class="view">
  <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
  <label><%- title %></label>
  <button class="destroy"></button>
</div>
<input class="edit" value="<%= title %>">
```

值得一提的是，在经典 Web 开发中，独立视图之间的导航需要使用页面刷新，但在 JavaScript 单页面应用程序中，一旦数据通过 Ajax 从服务器获取，它就可以在同一个页面中动态地渲染新视图。由于它不能自动更新 URL，因此导航的作用会转到路由，协助管理应用程序状态（例如，允许用户给它们导航到的特定视图加书签）。由于路由既不是 MVC 的一部分，也不在每个 MVC 框架中，因此这一节不会做更详细的介绍。

4. 控制器

在 Todo 应用程序中，控制器将负责处理用户在特定 Todo 的编辑视图中所做的更改，以便在用户完成编辑后更新特定的 Todo 模型。

大多数 JavaScript MVC 框架的控制器与传统的 MVC 模式所解释的相悖，其原因各有不同，但在我看来，JavaScript MVC 框架的作者最初看到的可能是 MVC 的客户端解释（例如 Ruby on Rails），我们意识到这种方法没有在客户端进行 1:1 的转换，

所以重新解释一下 MVC 中的 C，以解决它们的状态管理问题。这是一种巧妙的方法，但这使开发人员在第一次接触 MVC 时，很难理解经典 MVC 模式以及控制器在其他 JavaScript 框架中的角色。

Backbone.js 有控制器吗？ Backbone 的视图通常包含控制器逻辑，而路由用于帮助管理应用程序状态。但根据经典 MVC 的解释，两者都不是真正的控制器。

在这点上，与在官方文档或博客文章中可能会提到的相反，Backbone 不是真正的 MVC 框架，可以将它作为一个以自己方式进行架构的 MV*家族的一员。当然，这没有错，但重要的是要通过讨论 MVC 来区分经典 MVC 和 MV*，以便有助于自己完成 Backbone 项目。

2.2 MVC 能带给我们什么

总之，MVC 模式可以帮助我们保持应用程序逻辑与用户界面的分离，使得变更和维护更加容易。由于这种逻辑分离，我们可以更清楚地知道，在哪里需要对数据、接口或业务逻辑进行修改，以及应该编写哪些单元测试。

2.2.1 深究 MVC

现在，大家可能已经对 MVC 模式提供的内容有了一个基本的了解，但为了满足大家的好奇心，我们将进一步探讨 MVC。

“四人组”没有将 MVC 视为设计模式，而是将它视为用于构建用户界面的一系列类。在他们看来，这实际上是其他 3 种经典设计模式的一种变体：观察者（Observer）（发布/订阅）、策略模式（Strategy）和组合模式（Composite）。根据 MVC 在框架中的实现方式，它也可以使用 Factory 和 Decorator 模式。如果大家想进一步了解这些模式，可以阅读我的另外一本书《JavaScript 设计模式初阶》，我在里面介绍了一部分这些模式。

正如我们已经讨论的，模型表示应用程序数据，而视图处理用户在屏幕上展示的内容。因此，MVC 依靠发布/订阅进行一些核心通信（奇怪的是在很多关于 MVC 模式的文章中都没有涉及）。当模型发生变化时，它发布到应用程序的其余部分表示其已经更新了，然后订阅者（一般是控制器）更新相应的视图。这种关系的观察者-视图（observer-viewer）特性，可以促进多个视图附加到同一个模型上。

对于有兴趣了解更多有关 MVC 解耦特性的开发人员（仍取决于实现）来说，这种模式的其中一个目标是帮助定义主题和其观察者之间的一对多关系。当主题发生变

化时，它的观察者获得更新。视图和控制器之间有一种稍微不同的关系。控制器为视图对不同用户输入做出响应提供了便利，它是一个 Strategy 模式的例子。

2.2.2 总结

回顾了经典 MVC 模式后，大家现在应该了解它是如何使开发者在应用程序中干净地分离关注点的，也应该意识到 JavaScript MVC 框架在 MVC 解释方面可能有所不同，以及它们如何分享原始模式的一些基本概念。

在查看一个新的 JavaScript MVC/MV*框架时，请记住，它对于我们退一步思考它如何被选择用于获得模型、视图、控制器或者其他替代方法是很有用处的，因为这可以更好地帮助我们了解如何使用框架。

2.2.3 延伸阅读

如果你有兴趣学习更多关于 Backbone.js 使用的 MVC 变体，请阅读附录 A 中的“MVP”小节。

2.3 基本概况

2.3.1 Backbone.js

- 包含模型、视图、集合、路由核心组件，执行自己的 MV*风格。
- 在视图和模型之间，支持事件驱动的通信。正如我们将看到的，在任何模型的属性上添加事件监听器、针对视图中的变化为开发者提供细粒度控制，这是相对简单的。
- 支持通过手动事件或单独的键值观察（KVO）库进行数据绑定。
- 为立即可用的 RESTful 接口提供支持，因此模型可以很容易地绑定至后端。
- 具有广泛的事件系统，很容易为 Backbone 添加发布/订阅支持。
- 使用有些开发人员更喜欢的 new 关键字实例化原型。
- 不需要了解模板框架，但 Underscore 的微模板在默认情况下是可用的。
- 对构建应用程序提供清晰而灵活的约定用法。Backbone 不强迫使用所有的组件，可以只使用那些需要的组件。

2.3.2 使用案例

1. Disqus

Disqus 选择 Backbone.js 来帮助最新版本的评论小部件（见图 2-5）的发布。鉴于 Backbone 占用空间小且易于扩展这一特点，对于分布式 Web 应用，它被认为是正确的选择。

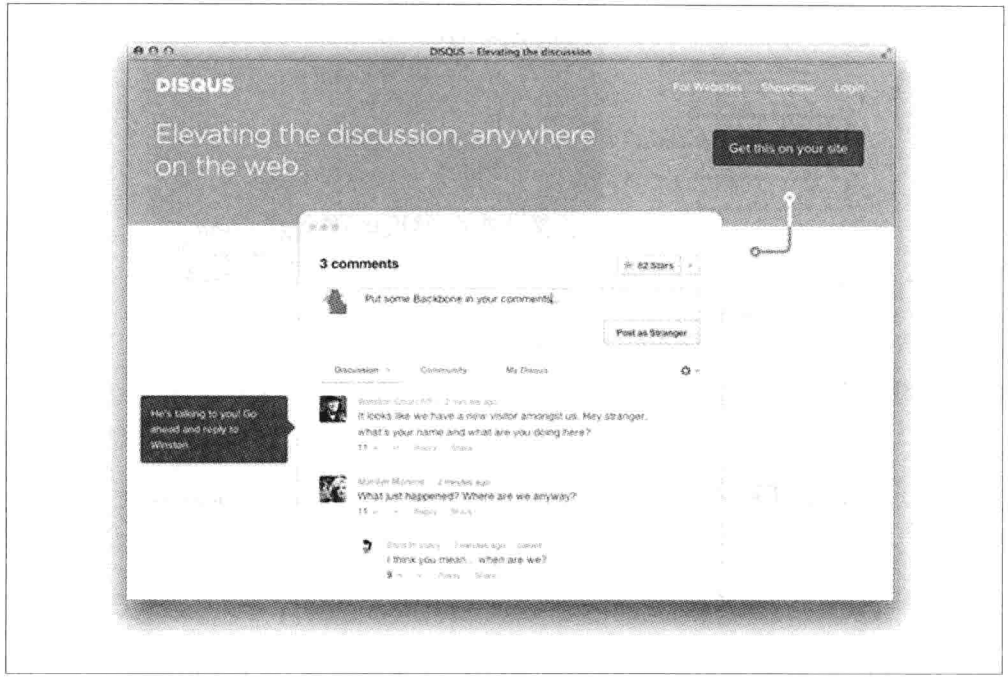


图 2-5 Disqus 评论小部件

2. Khan Academy

旨在提供一个在任何地方都可以给任何人提供免费世界级教育的 Web 应用程序，Khan 使用 Backbone 来保持其前端代码被模块化和组织化（见图 2-6）。

3. MetaLab

MetaLab 创建了 Flow——一个使用 Backbone 的团队任务管理应用程序(见图 2-7)。其工作平台使用 Backbone 创建任务视图、活动、账户、标签等。

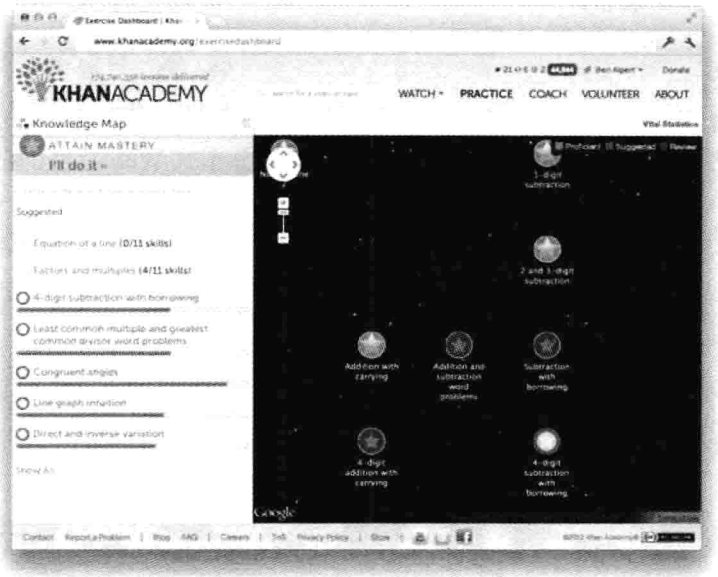


图 2-6　Khan Academy 知识地图

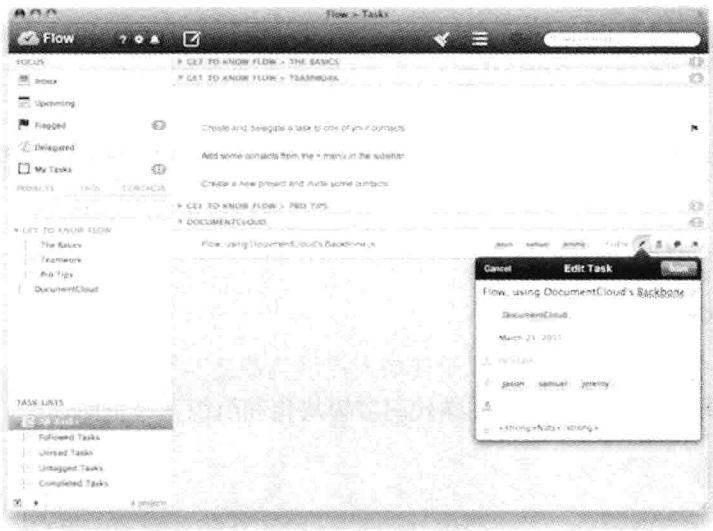


图 2-7　在线任务管理应用

4．Walmart Mobile

沃尔玛选择 Backbone 来帮助其移动 Web 应用程序（见图 2-8）的开发，在开发过

程中创建两个新的扩展框架：Thorax 和 Lumbar。我们稍后将讨论这两种扩展框架。

图 2-8　Walmart Mobile

5．Airbnb

Airbnb（见图 2-9）使用 Backbone 开发了其移动 Web 应用，目前在很多产品中都使用了 Backbone。

图 2-9　Airbnb 主页

6. Code School

Code School 课程挑战应用（见图 2-10）的构建从一开始就使用了 Backbone，利用其路由、集合、模型以及复杂事件处理的所有特性。

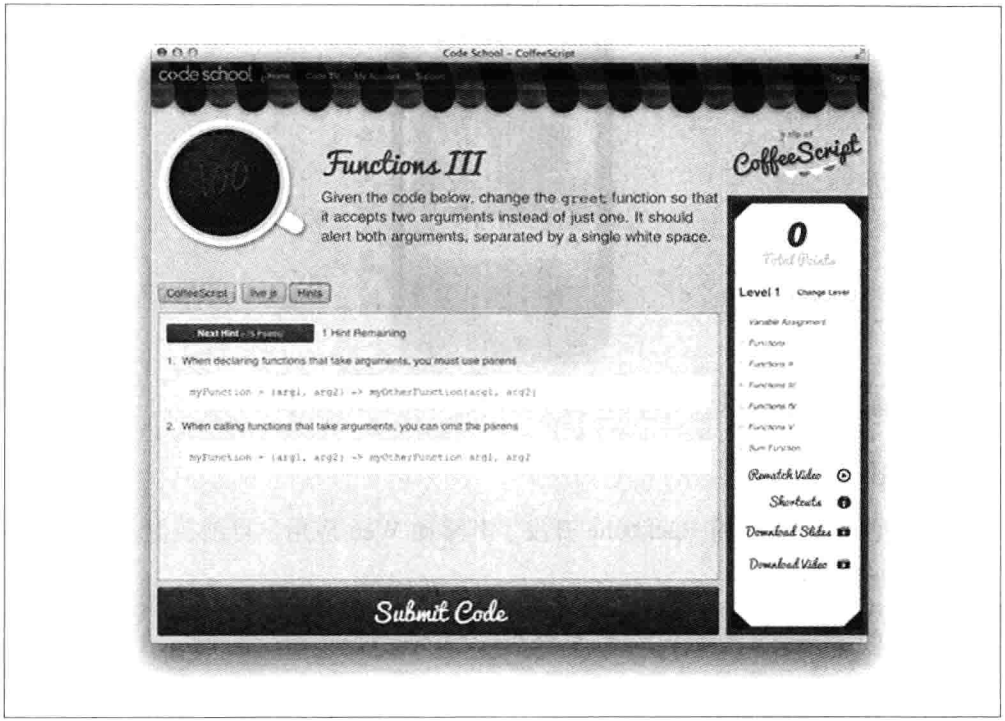


图 2-10　Code School 学习环境

第3章

Backbone基础

在本章中，你将会学习Backbone的基本知识：模型（model）、视图（view）、集合（collection）、事件（event）以及路由（router）。这并不是官方文档的简单复制，而是帮助你在使用Backbone构建应用程序之前去理解Backbone背后的核心概念。

3.1 准备开始

开始深入了解代码示例之前，让我们定义一下样板代码，用于指定Backbone所需的依赖文件。该样板代码在很多情况下都可以通过简单地修改代码而重复利用，从而让你很轻松地运行示例代码。

可以将下面的代码粘贴到文本编辑器里，使用任何示例中的JavaScript代码替换<script>标签中的注释行。

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
</script>
<script src="http://documentcloud.github.com/underscore/underscore-min.js">
</script>
<script src="http://documentcloud.github.com/backbone/backbone-min.js">
</script>
<script>
```

```
  // Your code goes here
</script>
</body>
</html>
```

然后可以保存该文件，并在所选择的浏览器（如 Chrome 或 Firefox）中运行该文件。或者，如果你喜欢使用在线代码编辑器，也可以在 **jsFiddle**（http://jsfiddle.net）和 **jsBin**（http://jsbin.com）上运行该样板代码。

大多数代码示例也可以直接在浏览器开发者工具中的控制台上运行，前提是你已经加载了 HTML 样板页面，以便 Backbone 和其依赖文件都可用。

对于Chrome浏览器，你可以通过右上角的Chrome菜单打开开发人员工具(DevTools)：选择“工具”(Tools)→“开发人员工具”(Developer Tools)命令，或者在 Windows/Linux 上使用 Ctrl+Shift+I 快捷键（或者 Mac 上的 Mac+Alt+I 键），如图 3-1 所示。

图 3-1　Chrome 开发工具控制台

接下来，切换到 Console（控制台）选项卡，在这里我们可以通过回车键，输入和运行任意 JavaScript 代码，还可以让控制台作为一个多行代码编辑器，在 Windows 上使用 Shift+Enter 快捷键（或在 Mac 上使用 Ctrl+Enter 键），将代码从一行的结尾切换到新一行的开头。

3.2　模型（Model）

Backbone 模型包含应用程序里的数据以及与数据相关的逻辑。例如，可以使用一个模型来表示一个待处理项(todo item)，它包含了像 title(todo 的内容)和 completed(todo 的状态）这样的属性。

我们可以通过扩展 Backbone.Model 来创建该模型，代码如下：

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

// We can then create our own concrete instance of a (Todo) model
// with no values at all:
var todo1 = new Todo();
// Following logs: {}
console.log(JSON.stringify(todo1));

// or with some arbitrary data:
var todo2 = new Todo({
  title: 'Check the attributes of both model instances in the console.',
  completed: true
});

// Following logs: {"title":"Check the attributes of both model
// instances in the console.","completed":true}
console.log(JSON.stringify(todo2));
```

3.2.1 初始化

创建一个新模型实例的时候，initialize()方法会被调用。该方法是可选的，不过，看看下面的代码，就知道为什么使用它是一个很好的实践了。

```
var Todo = Backbone.Model.extend({
  initialize: function(){
      console.log('This model has been initialized.');
  }
});

var myTodo = new Todo();
// Logs: This model has been initialized.
```

3.2.2 默认值

很多时候，我们想让模型拥有一组默认值（例如，一个所有数据都不是由用户提供的场景）。可以通过使用模型里的 defaults 属性来设置默认值。

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Now we can create our concrete instance of the model
// with default values as follows:
var todo1 = new Todo();

// Following logs: {"title":"","completed":false}
console.log(JSON.stringify(todo1));
```

```
// Or we could instantiate it with some of the attributes (e.g., with
// custom title):
var todo2 = new Todo({
  title: 'Check attributes of the logged models in the console.'
});

// Following logs: {"title":"Check attributes of the logged models
// in the console.","completed":false}
console.log(JSON.stringify(todo2));

// Or override all of the default attributes:
var todo3 = new Todo({
  title: 'This todo is done, so take no action on this one.',
  completed: true
});

// Following logs: {"title":"This todo is done, so take no action on
// this one.","completed":true}
console.log(JSON.stringify(todo3));
```

3.2.3 赋值与取值

1. Model.get()

Model.get() 用于访问模型的属性。

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

var todo1 = new Todo();
console.log(todo1.get('title')); // empty string
console.log(todo1.get('completed')); // false

var todo2 = new Todo({
  title: "Retrieved with model's get() method.",
  completed: true
});
console.log(todo2.get('title')); // Retrieved with model's get() method.
console.log(todo2.get('completed')); // true
```

如果我们需要读取或者复制一个模型的所有数据属性，可以使用该模型的 toJSON() 方法。该方法将所有属性的副本作为一个对象进行返回（不是 JSON 字符串，尽管名字是 toJSON）。当给 JSON.stringify()传递一个带有 toJSON()方法的对象时，JSON.stringify()处理的是该对象执行 toJSON()以后的返回值，而不是原始对象。上一节的示例在调用 JSON.stringify()记录模型实例的时候就充分利用了该特性。

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

var todo1 = new Todo();
var todo1Attributes = todo1.toJSON();
// Following logs: {"title":"","completed":false}
console.log(todo1Attributes);

var todo2 = new Todo({
  title: "Try these examples and check results in console.",
  completed: true
});

// logs: {"title":"Try these examples and check results in console.",
// "completed":true}
console.log(todo2.toJSON());
```

2. Model.set()

Model.set()是在模型上设置一个包含一个或多个属性的 Hash 散列，当其中的任意属性更改模型的状态时，就会在上面触发 change 事件。每个属性的 change 事件也可以在模型上进行触发和绑定（如 change:name、change:age）。

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Setting the value of attributes via instantiation
var myTodo = new Todo({
  title: "Set through instantiation."
});
console.log('Todo title: ' + myTodo.get('title'));
// Todo title: Set through instantiation.
console.log('Completed: ' + myTodo.get('completed'));
// Completed: false

// Set single attribute value at a time through Model.set():
myTodo.set("title", "Title attribute set through Model.set().");
console.log('Todo title: ' + myTodo.get('title'));
// Todo title: Title attribute set through Model.set().
console.log('Completed: ' + myTodo.get('completed'));
// Completed: false
```

```
// Set map of attributes through Model.set():
myTodo.set({
  title: "Both attributes set through Model.set().",
  completed: true
});
console.log('Todo title: ' + myTodo.get('title'));
// Todo title: Both attributes set through Model.set().
console.log('Completed: ' + myTodo.get('completed'));
// Completed: true
```

3. 直接访问

模型暴露了一个.attributes 属性，描述了包含该模型状态的一个 hash 散列。通常和服务器返回数据的 JSON 对象的形式一样，但也可以采用其他形式。

如果通过模型上的.attributes 属性设置值，可以绕过该模型上绑定的触发器。

修改成{silent:true}并不会触发 change:attr 事件，而是会像石沉大海一样。

```
var Person = new Backbone.Model();
Person.set({name: 'Jeremy'}, {silent: true});

console.log(!Person.hasChanged(0));
// true
console.log(!Person.hasChanged(''));
// true
```

记住，在可能的情况下，使用 Model.set()或像前面那样直接实例化是最佳做法。

3.2.4 监听模型变化

模型改变的时候，如果我们想收到一个通知，可以通过监听该模型上的 change 事件来实现。方便添加监听器的地方就是在 initialize()函数中，代码如下：

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  },
  initialize: function(){
    console.log('This model has been initialized.');
    this.on('change', function(){
        console.log('- Values for this model have changed.');
    });
  }
});
var myTodo = new Todo();

myTodo.set('title', 'The listener is triggered whenever an attribute
// value changes.');
```

```
console.log('Title has changed: ' + myTodo.get('title'));


myTodo.set('completed', true);
console.log('Completed has changed: ' + myTodo.get('completed'));

myTodo.set({
  title: 'Changing more than one attribute at the same time only triggers
// the listener once.',
  completed: true
});

// Above logs:
// This model has been initialized.
// - Values for this model have changed.
// Title has changed: The listener is triggered when an attribute value changes.
// - Values for this model have changed.
// Completed has changed: true
// - Values for this model have changed.
```

在 Backbone 模型中，我们也可以监听单个属性的改变。在下面的示例中，当特定属性（Todo 模型中的 title 属性）改变时，我们记录了相应的消息。

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  },

  initialize: function(){
    console.log('This model has been initialized.');
    this.on('change:title', function(){
        console.log('Title value for this model has changed.');
    });
  },

  setTitle: function(newTitle){
    this.set({ title: newTitle });
  }
});

var myTodo = new Todo();

// Both of the following changes trigger the listener:
myTodo.set('title', 'Check what\'s logged.');
myTodo.setTitle('Go fishing on Sunday.');
// But this change type is not observed, so no listener is triggered:
myTodo.set('completed', true);
console.log('Todo set as completed: ' + myTodo.get('completed'));

// Above logs:
// This model has been initialized.
```

```
// Title value for this model has changed.
// Title value for this model has changed.
// Todo set as completed: true
```

3.2.5 验证

Backbone 支持通过 model.validate()进行模型验证，允许在设置属性值之前对属性值进行检查。默认情况下，通过调用 save()方法或带有{validate:true}参数的 set()方法持久化模型时，验证就会触发。

```
var Person = new Backbone.Model({name: 'Jeremy'});

// Validate the model name
Person.validate = function(attrs) {
  if (!attrs.name) {
    return 'I need your name';
  }
};

// Change the name
Person.set({name: 'Samuel'});
console.log(Person.get('name'));
// 'Samuel'

// Remove the name attribute, force validation
Person.unset('name', {validate: true});
// false
```

模型中也可以使用 unset()方法，通过删除模型内部的属性 hash 散列，从而达到删除该属性的目的。

验证函数可以是非常简单的，也可以根据需要编写成复杂的。如果提供的属性都是有效的，.validate()不会返回任何值；相反，如果参数是无效的，就会返回一个错误值。

如果有错误值返回，则：

- model 会触发 invalid 事件，同时会将.validate()的返回值赋值给 validationError 属性。
- .save()不会继续执行，同时 model 上的属性不会在服务器上修改。

下面是一个更完整的验证例子：

```
var Todo = Backbone.Model.extend({
  defaults: {
    completed: false
  },
```

```
  validate: function(attribs){
    if(attribs.title === undefined){
        return "Remember to set a title for your todo.";
    }
  },

  initialize: function(){
    console.log('This model has been initialized.');
    this.on("invalid", function(model, error){
        console.log(error);
    });
  }
});

var myTodo = new Todo();
myTodo.set('completed', true, {validate: true});
// logs: Remember to set a title for your todo.
console.log('completed: ' + myTodo.get('completed')); // completed: false
```

传递给 validate 函数的 attributes 对象表示的是在执行 set()或者 save()以后该模型的结果，该对象有别于 model 当前的属性及模型操作时所传递的参数。因为它是通过复制来创建的，它不会改变函数输入的任何 Number、String 或 Boolean 属性，但有可能会改变嵌套对象里这些类型的属性。

可以访问 http://jsfiddle.net/2NdDY/7/查看这样的例子（由@fivetanley 提供）。

3.3 视图（View）

Backbone 中的视图不会保护应用程序中的 HTML 代码，它们包含着模型数据里的展示逻辑，用于展示给用户。视图使用 JavaScript 模板完成这一功能（如 Underscore 微模板、Mustache、jQuerytmpl 等）。视图的 render()方法可以被绑定在模型的 change()事件上，不需要重新刷新整个页面，就可以使视图即时反映模型的变化。

3.3.1 创建视图

创建新视图相对简单，和创建模型类似。简单扩展一下 Backbone.View 就可以创建一个新视图。上一小节我们介绍了如下的简单例子 TodoView；现在，让我们来仔细看看它是如何工作的。

```
var TodoView = Backbone.View.extend({

  tagName:  'li',

  // Cache the template function for a single item.
  todoTpl: _.template( "An example template" ),
```

```
    events: {
      'dblclick label': 'edit',
      'keypress .edit': 'updateOnEnter',
      'blur .edit':   'close'
    },

    // Rerender the titles of the todo item.
    render: function() {
      this.$el.html( this.todoTpl( this.model.toJSON() ) );
      this.input = this.$('.edit');
      return this;
    },

    edit: function() {
      // executed when todo label is double-clicked
    },

    close: function() {
      // executed when todo loses focus
    },

    updateOnEnter: function( e ) {
      // executed on each keypress when in todo edit mode,
      // but we'll wait for enter to get in action
    }
  });

  var todoView = new TodoView();

  // log reference to a DOM element that corresponds to the view instance
  console.log(todoView.el); // logs <li></li>
```

3.3.2 el 是什么

View 的一个核心属性是 el（上例中最后一条语句输出的值）。那么，el 是什么呢？它是如何定义的呢？

从根本上来说，el 是 DOM 元素的一个引用，所有视图都必须有一个 el。视图可以使用 el 构成它的元素内容，然后将所有的内容一次性插入到 DOM 里，使页面渲染更快，因为浏览器执行了最小次数的重排和重绘。

有两种方式可以将 DOM 元素与视图相关联：为视图创建一个新元素，随后将它添加到 DOM 里；用页面已经存在的元素给视图做一个引用。

如果想对一个视图创建一个新的元素，可以在视图上对下面的属性做任意组合的设置：tagName、id 和 className。Backbone 就会创建一个新的元素，新创建的元素引用将在 el 属性上可用。tagName 如果不指定值的话，默认为 div。

在前面的示例中，tagName 设置的是 li，所以结果是创建了一个 li 元素。下面的示例将创建一个带有 id 属性和 class 属性的 ul 元素：

```
var TodosView = Backbone.View.extend({
  tagName: 'ul', // required, but defaults to 'div' if not set
  className: 'container', // optional, you can assign multiple classes to
// this property like so: 'container homepage'
  id: 'todos', // optional
});

var todosView = new TodosView();
console.log(todosView.el); // logs <ul id="todos" class="container"></ul>
```

上述代码创建了下面的 DOM 元素，但它并没有附加到视图的 DOM 元素上。

```
<ul id="todos" class="container"></ul>
```

如果元素在页面上已经存在了，可以为 el 设置一个 CSS 选择器来匹配该元素。

```
el: '#footer'
```

或者在创建视图的时候，将 el 设置在现有的元素上：

```
var todosView = new TodosView({el: $('#footer')});
```

定义一个视图的时候，如果想在运行的时候指定相关的值，可以将 options、el、tagName、id 和 className 定义为函数。

1. $el 与$()

视图中的逻辑经常需要在 el 元素和其嵌套元素上调用 jQuery 或 Zepto 的函数。通过定义$el 属性和$()函数，Backbone 将使这一切变得很简单。视图的.$el 属性等价于$(view.el)，.$(selector)则等价于$(view.el).find(selector)。在 TodosView 例子的 render 方法中，我们看到了 this.$ el 用于设置元素的 HTML，而 this.$()则用于查找 class 为 edit 的子元素。

2. setElement

如果想将现有的 Backbone 视图应用于另外一个不同的 DOM 元素，可以使用 setElement。重写 this.el 需要同时更改 DOM 引用和在新元素上重新绑定事件（并且从旧元素上解绑事件）。

setElement 将创建一个缓存$el 引用，并且将事件从旧元素委托给新元素。

```
// We create two DOM elements representing buttons
// which could easily be containers or something else
var button1 = $('<button></button>');
var button2 = $('<button></button>');
```

```
// Define a new view
var View = Backbone.View.extend({
      events: {
        click: function(e) {
          console.log(view.el === e.target);
        }
      }
    });

// Create a new instance of the view, applying it
// to button1
var view = new View({el: button1});

// Apply the view to button2 using setElement
view.setElement(button2);

button1.trigger('click');
button2.trigger('click'); // returns true
```

el 属性表示的是视图即将渲染展示的页码标记。为了让视图能在页面上渲染，需要将视图作为新元素添加到页面上，或者将视图附加到一个页面现有的元素上。

```
// We can also provide raw markup to setElement
// as follows (just to demonstrate it can be done):
var view = new Backbone.View;
view.setElement('<p><a><b>test</b></a></p>');
view.$('a b').html(); // outputs "test"
```

3. 理解 render()

render()是渲染模板逻辑中的一个可选函数。在本书的例子中，我们将使用 Underscore 微模板，但请记住，我们也可以使用其他模板框架。我们的示例将参考以下 HTML 标记：

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
</head>
<body>
  <div id="todo">
  </div>
  <script type="text/template" id="item-template">
    <div>
      <input id="todo_complete" type="checkbox" <%= completed ?
       'checked="checked"' : '' %>>
      <%= title %>
    </div>
  </script>
  <script src="underscore-min.js"></script>
```

```
    <script src="backbone-min.js"></script>
    <script src="jquery-min.js"></script>
    <script src="example.js"></script>
  </body>
  </html>
```

Underscore 里的_.template 方法将 JavaScript 模板编译成可以渲染页面的可执行函数。在 TodoView 里，当该试图创建的时候，将 id 为 item-template 的模板传给了_.template 方法进行编译，并赋值给 todoTpl 属性。

render()方法使用该模板，通过使用 toJSON()将视图相关的模型属性编码后，传递给该模板。该模板使用模型（model）的 title 和 completed 属性，执行完视图里包含的表达式后，返回了最终的 HTML 标记。然后使用$el 属性，将这些标记设置为 el DOM 元素的 HTML 内容。

短短几行代码，填充了模板，并且提供一个具有完整数据集的 HTML 标记。

常见的 Backbone 约定是在 render 的底部返回 this，它很有用，原因是：

- 使得该试图可以在其他父视图里很容易得到重用；
- 创建一批元素，但并不为它们单独渲染和绘制，而是一次性填充整组元素。

让我们尝试实现第二条。简单 ListView 里的 render 方法，并没有为每个 item 条目都使用 itemView，可以用如下代码编写：

```
var ListView = Backbone.View.extend({
  render: function(){
    this.$el.html(this.model.toJSON());
  }
});
```

它足够简单。现在假设我们决定使用 ItemView 为我们的列表进行完善和增强，ItemView 可以编写成如下这样：

```
var ItemView = Backbone.View.extend({
  events: {},
  render: function(){
    this.$el.html(this.model.toJSON());
    return this;
  }
});
```

注意 render 底部 return this 的用法。这种常见模式允许我们将该视图作为子视图进行重用，也可以使用它在渲染之前进行预渲染。此时需要如下修改 ListView 的 render 方法：

```
var ListView = Backbone.View.extend({
  render: function(){

    // Assume our model exposes the items we will
    // display in our list
    var items = this.model.get('items');

    // Loop through each of our items using the Underscore
    // _.each iterator
    _.each(items, function(item){

      // Create a new instance of the ItemView, passing
      // it a specific model item
      var itemView = new ItemView({ model: item });
      // The itemView's DOM element is appended after it
      // has been rendered. Here, the 'return this' is helpful
      // as the itemView renders its model. Later, we ask for
      // its output ("el")
      this.$el.append( itemView.render().el );
    }, this);
  }
});
```

4. Events 哈希

Backbone 的 events 哈希允许我们为 el 相关的自定义选择器或者直接为 el 本身（如果没有提供选择器）添加事件监听器。事件采用的是'eventName selector'：'callbackFunction'这样的 key-value 对的形式，并且支持大量的 DOM 事件类型，包括 click、submit、mouseover、dblclick 等。

```
// A sample view
var TodoView = Backbone.View.extend({
  tagName:  'li',

  // with an events hash containing DOM events
  // specific to an item:
  events: {
    'click .toggle': 'toggleCompleted',
    'dblclick label': 'edit',
    'click .destroy': 'clear',
    'blur .edit': 'close'
  },
```

不是特别明显的是，Backbone 使用 jQuery 的.delegate()时，还进行了进一步延展，以便 this 总是指向带有回调函数（callback）的视图对象（view object）。唯一需要记住的是：提供给 events 属性的任何字符串回调，在当前视图的作用域里都需要对应一个同名的函数。

声明委托式的 jQuery event，也就意味着不必担心一个元素是否已经在 DOM 上渲

染。通常在使用 jQuery 的过程中，我们要担心的是：绑定事件时所绑定的元素在 DOM 里是否存在。

在 TodoView 示例中，当用户双击 el 元素里的 labal 元素时，edit 回调进行了调用；而样式为 edit 元素上在进行按键的时候，updateOnEnter 回调也会被调用；edit 元素失去焦点的时候，则会调用 close 回调。这些任何一个回调函数都可以使用 this 引用 TodoView 对象。

注意，也可以使用_.bind(this.viewEvent, this)自行绑定方法，这和每个事件里的 key-value 对一样有效。如下代码在模型改变时使用_.bind 重新渲染了视图：

```
var TodoView = Backbone.View.extend({
  initialize: function() {
    this.model.bind('change', _.bind(this.render, this));
  }
});
```

.bind 一次只能绑定一个方法，但它支持柯里化（Currying）。鉴于柯里化可以返回回弹函数（bound function），我们可以在匿名函数上使用.bind。

3.4 集合（Collection）

集合是模型的组合，可以通过扩展 Backbone.Collection 来创建集合。

通常情况下，创建集合的时候也要定义一个属性，指定该集合所包含的模型（model）类型，同时还包含任何所需的实例属性。

在下面的示例中，我们创建了一个包含 Todo 模型的 TodoCollection 集合：

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo
});

var myTodo = new Todo({title:'Read the whole book', id: 2});

// pass array of models on collection instantiation
var todos = new TodosCollection([myTodo]);
console.log("Collection size: " + todos.length); // Collection size: 1
```

3.4.1 添加和移除模型

在前面的示例中，集合在实例化过程中使用包含模型的数组来进行填充。集合创建以后，可以通过使用 add()和 remove()方法添加或移除模型。

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
});

var a = new Todo({ title: 'Go to Jamaica.'}),
    b = new Todo({ title: 'Go to China.'}),
    c = new Todo({ title: 'Go to Disneyland.'});

var todos = new TodosCollection([a,b]);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 2

todos.add(c);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 3

todos.remove([a,b]);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 1

todos.remove(c);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 0
```

注意，add()和 remove()方法都接受传入单个模型和模型列表。

另外也要注意，在集合上使用 add()时，传入{merge: true}将导致新模型合并到现有的模型上，而不是忽略掉。

```
var items = new Backbone.Collection;
items.add([{ id : 1, name: "Dog" , age: 3}, { id : 2, name: "cat" , age: 2}]);
items.add([{ id : 1, name: "Bear" }], {merge: true });
items.add([{ id : 2, name: "lion" }]); // merge: false

console.log(JSON.stringify(items.toJSON()));
// [{"id":1,"name":"Bear","age":3},{"id":2,"name":"cat","age":2}]
```

3.4.2 检索模型

有几种不同的方式可以从集合里检索一个模型，最简单的方式是使用集合的.get()方法，接受一个 id 作为参数，代码如下：

```
var myTodo = new Todo({title:'Read the whole book', id: 2});

// pass array of models on collection instantiation
var todos = new TodosCollection([myTodo]);

var todo2 = todos.get(2);

// Models, as objects, are passed by reference
console.log(todo2 === myTodo); // true
```

在客户端-服务器端（client-server）应用程序中，集合包含从服务器端获取的模型。任何时候在客户端和服务器端之间进行数据交换，都需要一种方式来定义唯一的标识，用来表示模型。在 Backbone 里，可以使用 id、cid 和 idAttribute 属性达到这一目的。

在 Backbone 里，每个模型都有一个 id，它是唯一标识符，可能是整数，也可能是字符串（例如，UUID）。模型也有一个 cid（客户端 ID），在模型创建的时候由 Backbone 自动生成。两种标识符都可以从集合里检索模型。

id 和 cid 之间的主要区别是：cid 是由 Backbone 生成的，如果我们没有真正的 id，这对我们就非常有益。使用 cid 的情况有可能是模型还没保存到服务器上，或者我们不想在数据库里保存模型。

idAttribute 是从服务器端所返回模型的标识属性（即数据库中的 id）。该属性告诉 Backbone 服务器上的那个数据字段用于填充 id 属性（想象这是一个映射器）。默认情况下，它假定为 id，但是我们可以根据需要进行自定义。例如，如果服务器在模型上设置的唯一属性叫 userId，那么在模型定义的时候，就需要将 idAttribute 设置为 userId。

模型 idAttribute 的值应该由服务器在保存模型时设置。在此之后，我们不需要手动设置它，除非需要进行进一步的控制。

如果模型实例有值，Backbone.Collection 在内部会包含一个可以用 id 属性遍历的模型数组。调用 collection.get(id)时，该数组会检查与模型 id 相一致的模型实例是否存在。

```
// extends the previous example

var todoCid = todos.get(todo2.cid);

// As mentioned in previous example,
// models are passed by reference
console.log(todoCid === myTodo); // true
```

3.4.3 事件监听

因为集合表示的是一组 item，所以我们可以监听 add 和 remove 事件，从集合添加或移除模型的时候会触发这些事件。示例如下：

```
var TodosCollection = new Backbone.Collection();

TodosCollection.on("add", function(todo) {
  console.log("I should " + todo.get("title") + ". Have I done it before? "
  + (todo.get("completed") ? 'Yeah!': 'No.' ));
});

TodosCollection.add([
  { title: 'go to Jamaica', completed: false },
  { title: 'go to China', completed: false },
  { title: 'go to Disneyland', completed: true }
]);
// The above logs:
// I should go to Jamaica. Have I done it before? No.
// I should go to China. Have I done it before? No.
// I should go to Disneyland. Have I done it before? Yeah!
```

此外，在集合中任意模型的属性上，都可以绑定 change 事件，用于监听这些模型属性的变化。

```
var TodosCollection = new Backbone.Collection();

// log a message if a model in the collection changes
TodosCollection.on("change:title", function(model) {
    console.log("Changed my mind! I should " + model.get('title'));
});

TodosCollection.add([
  { title: 'go to Jamaica.', completed: false, id: 3 },
]);

var myTodo = TodosCollection.get(3);

myTodo.set('title', 'go fishing');
// Logs: Changed my mind! I should go fishing
```

jQuery 风格的事件映射形式 obj.on({click: action})也可以使用。这比在.on 上使用 3 个单独的调用要清晰，而且在视图里使用事件哈希的对齐方式也更好：

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var myTodo = new Todo();
```

```
myTodo.set({title: 'Buy some cookies', completed: true});

myTodo.on({
   'change:title' : titleChanged,
   'change:completed' : stateChanged
});

function titleChanged(){
  console.log('The title was changed!');
}

function stateChanged(){
  console.log('The state was changed!');
}
myTodo.set({title: 'Get the groceries'});
// The title was changed!
```

Backbone 事件还支持 once()方法，它可以确保在通知到达以后，回调只执行一次。它类似于 Node 的 once 或者 jQuery 的 one。当我们想“下一次发生时，调用这个”时，这样做就非常有用。

```
// Define an object with two counters
var TodoCounter = { counterA: 0, counterB: 0 };
// Mix in Backbone Events
_.extend(TodoCounter, Backbone.Events);


// Increment counterA, triggering an event
var incrA = function(){
  TodoCounter.counterA += 1;
  TodoCounter.trigger('event');
};

// Increment counterB
var incrB = function(){
  TodoCounter.counterB += 1;
};

// Use once rather than having to explicitly unbind
// our event listener
TodoCounter.once('event', incrA);
TodoCounter.once('event', incrB);

// Trigger the event once again
TodoCounter.trigger('event');

// Check our output
console.log(TodoCounter.counterA === 1); // true
console.log(TodoCounter.counterB === 1); // true
```

counterA 和 counterB 应该只增加一次。

3.4.4 重置和刷新集合

比起一个一个单独添加或删除模型，我们更想要一次性更新整个集合。Collection.set()接收一个模型数组参数，并执行更新集合所必要的添加、删除和更新操作。

```
var TodosCollection = new Backbone.Collection();

TodosCollection.add([
    { id: 1, title: 'go to Jamaica.', completed: false },
    { id: 2, title: 'go to China.', completed: false },
    { id: 3, title: 'go to Disneyland.', completed: true }
]);
// we can listen for add/change/remove events
TodosCollection.on("add", function(model) {
  console.log("Added " + model.get('title'));
});

TodosCollection.on("remove", function(model) {
  console.log("Removed " + model.get('title'));
});

TodosCollection.on("change:completed", function(model) {
  console.log("Completed " + model.get('title'));
});

TodosCollection.set([
    { id: 1, title: 'go to Jamaica.', completed: true },
    { id: 2, title: 'go to China.', completed: false },
    { id: 4, title: 'go to Disney World.', completed: false }
]);

// Above logs:
// Removed go to Disneyland.
// Completed go to Jamaica.
// Added go to Disney World.
```

如果需要简单地替换整个集合的内容，可以如下这样使用 Collection.reset()：

```
var TodosCollection = new Backbone.Collection();

// we can listen for reset events
TodosCollection.on("reset", function() {
  console.log("Collection reset.");
});

TodosCollection.add([
  { title: 'go to Jamaica.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Disneyland.', completed: true }
]);

console.log('Collection size: ' + TodosCollection.length); // Collection size: 3
```

```
TodosCollection.reset([
  { title: 'go to Cuba.', completed: false }
]);
// Above logs 'Collection reset.'

console.log('Collection size: ' + TodosCollection.length); // Collection size: 1
```

另一个有用的技巧是，通过使用不带参数的 reset 方法来完全清空一个集合。当我们想清空当前页面的记录，然后动态加载新页面的记录时，使用该技巧非常方便。

```
myCollection.reset();
```

注意，使用 Collection.reset()时不会触发任何 add 和 remove 事件，而是会如前面的例子所示，触发 reset 事件。我们使用它的原因是要在极端情况下使用这种超级优化的渲染方式，因为单独的触发事件太浪费资源。

另外还要注意的是，为了方便起见，监听 reset 事件时可以从 options.previousModels 访问 reset 之前的模型列表。

```
var Todo = new Backbone.Model();
var Todos = new Backbone.Collection([Todo])
.on('reset', function(Todos, options) {
  console.log(options.previousModels);
  console.log([Todo]);
  console.log(options.previousModels[0] === Todo); // true
});
Todos.reset([]);
```

集合的 update()方法可以用于对模型集进行智能更新（也可以作为一个选项去获取）。这种方法可以指定一组模型列表用于智能更新集合。新更新列表中的模型，在集合中如果不存在，就添加它；如果存在，就合并其属性。在集合中有、而在更新列表中没有的模型将会被删除。

```
var theBeatles = new Collection(['john', 'paul', 'george', 'ringo']);

theBeatles.update(['john', 'paul', 'george', 'pete']);

// Fires a `remove` event for 'ringo', and an `add` event for 'pete'.
// Updates any of john, paul, and george's attributes that may have
// changed over the years.
```

3.4.5 Underscore 实用函数

Backbone 充分利用了其强依赖库 Underscore，使得很多实用函数都可以直接在集合上使用。

（1）forEach：迭代集合

```
var Todos = new Backbone.Collection();
```

```
Todos.add([
  { title: 'go to Belgium.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Austria.', completed: true }
]);
// iterate over models in the collection
Todos.forEach(function(model){
  console.log(model.get('title'));
});
// Above logs:
// go to Belgium.
// go to China.
// go to Austria.
```

（2）sortBy()：通过特定的属性对集合进行排序

```
// sort collection
var sortedByAlphabet = Todos.sortBy(function (todo) {
    return todo.get("title").toLowerCase();
});

console.log("- Now sorted: ");

sortedByAlphabet.forEach(function(model){
  console.log(model.get('title'));
});
// Above logs:
// go to Austria.
// go to Belgium.
// go to China.
```

（3）map()：通过转换函数映射列表里的每个项，重新生成一个新集合

```
var count = 1;
console.log(Todos.map(function(model){
  return count++ + ". " + model.get('title');
}));
// Above logs:
//1. go to Belgium.
//2. go to China.
//3. go to Austria.
```

（4）min()/max()：获取特定属性为最小/最大值的 model 项

```
Todos.max(function(model){
  return model.id;
}).id;

Todos.min(function(model){
  return model.id;
}).id;
```

（5）pluck()：获取特定属性的集合

```
var captions = Todos.pluck('caption');
// returns list of captions
```

（6）filter()：过滤集合

通过一组模型 ID 形成的数组进行过滤。

```
var Todos = Backbone.Collection.extend({
  model: Todo,
  filterById: function(ids){
    return this.models.filter(
      function(c) {
        return _.contains(ids, c.id);
      })
  }
});
```

（7）indexOf()：返回集合中特定索引位置的模型

```
var People = new Backbone.Collection;

People.comparator = function(a, b) {
  return a.get('name') < b.get('name') ? -1 : 1;
};

var tom = new Backbone.Model({name: 'Tom'});
var rob = new Backbone.Model({name: 'Rob'});
var tim = new Backbone.Model({name: 'Tim'});

People.add(tom);
People.add(rob);
People.add(tim);

console.log(People.indexOf(rob) === 0); // true
console.log(People.indexOf(tim) === 1); // true
console.log(People.indexOf(tom) === 2); // true
```

（8）any()：通过迭代器测试集合中是否存在特定的模型

```
Todos.any(function(model){
  return model.id === 100;
});

// or
Todos.some(function(model){
  return model.id === 100;
});
```

（9）size()：返回集合的大小

```
Todos.size();

// equivalent to
Todos.length;
```

（10）isEmpty()：判断集合是否为空

```
var isEmpty = Todos.isEmpty();
```

（11）groupBy()：通过模型的属性将集合进行分组

```
var Todos = new Backbone.Collection();

Todos.add([
  { title: 'go to Belgium.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Austria.', completed: true }
]);

// create groups of completed and incomplete models
var byCompleted = Todos.groupBy('completed');
var completed = new Backbone.Collection(byCompleted[true]);
console.log(completed.pluck('title'));
// logs: ["go to Austria."]
```

另外，Underscore 的几个其他操作，在模型上作为方法也是可以用的。

（12）pick()：过滤出模型特定属性的属性值

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var todo = new Todo({title: 'go to Austria.'});
console.log(todo.pick('title'));
// logs {title: "go to Austria"}
```

（13）omit()：过滤出除模型特定属性以外的属性值

```
var todo = new Todo({title: 'go to Austria.'});
console.log(todo.omit('title'));
// logs {completed: false}
```

（14）keys()与 values()：获取一个对象的所有属性名称/属性值

```
var todo = new Todo({title: 'go to Austria.'});
console.log(todo.keys());
// logs: ["title", "completed"]

console.log(todo.values());
//logs: ["go to Austria.", false]
```

（15）pairs()：把一个对象转变为[key, value]形式的数组

```
var todo = new Todo({title: 'go to Austria.'});
```

```
var pairs = todo.pairs();

console.log(pairs[0]);
// logs: ["title", "go to Austria."]
console.log(pairs[1]);
// logs: ["completed", false]
```

（16）invert()：将一个对象的键（keys）和值（values）互换后创建一个新对象

```
var todo = new Todo({title: 'go to Austria.'});
console.log(todo.invert());

// logs: {go to Austria.: "title", false: "completed"}
```

可以在官方文档（http://documentcloud.github.io/underscore/）中找到 Underscore 的完整功能清单。

3.4.6 链式 API

说到这些实用方法，Backbone 里的另外一个语法糖是它支持 Underscore 的 chain()方法。链式是面向对象语言的一个常见用法，链就是通过一条语句在同一个对象上进行一组方法的调用。当 Backbone 在让 Underscore 的数组操作行为可以在集合上使用时，并不能直接进行链式操作，因为它们返回的是数组，而不是原有集合。

幸运的是，Underscore 的 chain()方法可以让我们在集合上进行链式调用。

chain()方法返回一个对象，并且 Underscore 的所有数组操作方法都附加到该对象上。在链的结尾通过调用 value()方法返回数组结果。如果你之前没有见过链式 API，它看起来就像这样：

```
var collection = new Backbone.Collection([
  { name: 'Tim', age: 5 },
  { name: 'Ida', age: 26 },
  { name: 'Rob', age: 55 }
]);

var filteredNames = collection.chain()
// start chain, returns wrapper around collection's models
  .filter(function(item) { return item.get('age') > 10; })
  // returns wrapped array excluding Tim
  .map(function(item) { return item.get('name'); })
  // returns wrapped array containing remaining names
  .value(); // terminates the chain and returns the resulting array

console.log(filteredNames); // logs: ['Ida', 'Rob']
```

Backbone的一些特定方法返回this，也就意味着它们也可以链式操作：

```
var collection = new Backbone.Collection();
```

```
collection
    .add({ name: 'John', age: 23 })
    .add({ name: 'Harry', age: 33 })
    .add({ name: 'Steve', age: 41 });

var names = collection.pluck('name');

console.log(names); // logs: ['John', 'Harry', 'Steve']
```

3.5 RESTful 持久化

到目前为止，我们所有的示例数据都是在浏览器上创建的。对于大多数单页面应用程序，模型是驻留在服务器上的数据集。在数据访问方面，Backbone 极大地简化了代码，通过在集合和模型上使用简单的 API，来执行 RESTful 与服务器同步。

3.5.1 从服务器上获取模型

Collections.fetch()通过发送 HTTP GET 请求到 URL 上(集合的 url 属性或 url 函数)，从服务器获取 JSON 数组形式的模型数据集。一旦数据接收，Backbone 将执行 set()函数来更新集合。

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  url: '/todos'
});

var todos = new TodosCollection();
todos.fetch(); // sends HTTP GET to /todos
```

3.5.2 保存模型到服务器

Backbone 可以从服务器上一次性获取整个集合，但更新模型却是通过单独调用 save()方法来实现的。当在服务器获取的模型上调用 save()方法时，Backbone 通过在集合的 URL 上附加一个 id 来构造一个新 URL，然后发送 HTTP PUT 请求到服务器上。如果模型是在浏览器上创建的新实例（没有 id），则 HTTP POST 请求会发送到集合的 URL 上。Collections.create()可以用于创建一个新模型，并将其添加到集合里，然后通过一个单独的方法调用，再将其发送到服务器上。

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  url: '/todos'
});

var todos = new TodosCollection();
todos.fetch();

var todo2 = todos.get(2);
todo2.set('title', 'go fishing');
todo2.save(); // sends HTTP PUT to /todos/2

todos.create({title: 'Try out code samples'});
// sends HTTP POST to /todos and adds to collection
```

正如前面章节提到的，调用 save()方法的时候，将自动调用 validate()方法，并且一旦验证失败，就会触发一个 invalid 事件。

3.5.3 从服务器删除模型

可以通过调用 destroy()方法从集合和服务器中删除一个存在的模型。与 Collection.remove()不同，remove()方法只是从集合中删除一个模型，而 Model.destroy()还会向集合的 URL 发送一个 HTTP DELETE 请求。

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  url: '/todos'
});

var todos = new TodosCollection();
todos.fetch();

var todo2 = todos.get(2);
todo2.destroy(); // sends HTTP DELETE to /todos/2 and removes from collection
```

如果模型是新建的，在该模型上调用 destroy()会返回 false。

```
var Todo = new Backbone.Model();
console.log(Todo.destroy());
// false
```

3.5.4 选项

每一个 RESTful API 方法都接受各种各样的可选项参数。最重要的是，所有的方法都接受成功（success）和错误（error）回调，用于自定义处理服务器响应。

在 Model.save()上指定{patch: true}选项，将使用 HTTP PATCH 请求，仅仅发送已经改变的属性（比如局部更新）到服务器上，而不是发送整个模型——也就是 model.save(attrs, {patch: true})。

```
// Save partial using PATCH
model.clear().set({id: 1, a: 1, b: 2, c: 3, d: 4});
model.save();
model.save({b: 2, d: 4}, {patch: true});
console.log(this.syncArgs.method);
// 'patch'
```

同样，给 Collection.fetch()传递{reset: true}选项，将返回使用 reset()更新的集合，而不是 set()。

欲获得 Backbone 支持选项的完整描述，可以查看 Backbone.js 文档。

3.6 事件（Event）

事件（Events）是一个基本的控制反转（IOC）。将事件触发后所要执行的函数注册成事件处理句柄，一旦特殊事件发生，就触发该函数，而不是让事件触发前的函数通过名称去调用该触发函数。

需要知道如何调用另外一部分反转程序的这部分应用程序，称为核心组件。该组件可以让你的业务逻辑不必知道用户界面是如何工作的，这就是 Backbone 事件系统的最强大之处。

掌握事件是让 Backbone 开发最有效的一种最快捷方式，那么就让我们来仔细看看 Backbone 的事件模型。

Backbone.Events 混入到了其他 Backbone 类，其中包括：

- Backbone；
- Backbone.Model；

- Backbone.Collection；
- Backbone.Router；
- Backbone.History；
- Backbone.View。

注意，Backbone.Events 也混入到了 Backbone 对象中。鉴于 Backbone 是一个全局都可以访问的对象，它可以作为一个简单的事件总线：

```
Backbone.on('event', function() {
    console.log('Handled Backbone event');
});
```

3.6.1 on()、off()和 trigger()

Backbone.Events 可以让任何对象具有绑定和触发事件的能力。我们可以很容易地将 Backbone.Events 混入到任意对象上，并且在绑定回调处理程序之前不需要定义任何事件。例如：

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

// Add a custom event
ourObject.on('dance', function(msg){
  console.log('We triggered ' + msg);
});

// Trigger the custom event
ourObject.trigger('dance', 'our event');
```

如果熟悉 jQuery 的自定义事件或者发布/订阅的概念，会知道 Backbone.Events 提供了类似的系统，其中 on 类似于 subscribe，trigger 类似于 publish。

on 会在对象上绑定一个回调函数，就像上例中绑定的 dance 函数。只要事件被触发，该回调函数就会被调用。

如果在页面上有很多事件，Backbone.js 官方文档建议，使用冒号对事件进行命名空间设定。例如：

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);
```

```
function dancing (msg) { console.log("We started " + msg); }
// Add namespaced custom events
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);

// Trigger the custom events
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");

// This one triggers nothing as no listener listens for it
ourObject.trigger("dance", "break dancing. Yeah!");
```

如果想对一个对象上发生的所有事件都进行通知（例如，在一个单独的地方显示所有事件），特殊事件 all 可以满足这一需求。all 事件可以如下这样使用：

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We started " + msg); }

ourObject.on("all", function(eventName){
  console.log("The name of the event passed was " + eventName);
});

// This time each event will be caught with a catch 'all' event listener
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");
ourObject.trigger("dance", "break dancing. Yeah!");
```

off 可以删除之前绑定在对象上的回调函数。回顾对发布/订阅（publish/subscribe）的比较，可以将 off 想象为自定义事件中的退订（unsubscribe）。

要删除上例中在 ourObject 上绑定的 dance 事件，我们只需这样做：

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We " + msg); }

// Add namespaced custom events
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);

// Trigger the custom events. Each will be caught and acted upon.
ourObject.trigger("dance:tap", "started tap dancing. Yeah!");
ourObject.trigger("dance:break", "started break dancing. Yeah!");

// Removes event bound to the object
```

```
ourObject.off("dance:tap");
// Trigger the custom events again, but one is logged.
ourObject.trigger("dance:tap", "stopped tap dancing.");
// won't be logged as it's not listened for
ourObject.trigger("dance:break", "break dancing. Yeah!");
```

要删除所有的回调函数，可以在 off()方法上传入之前在对象上绑定的事件名称。如果想删除特定的回调函数，则可以将回调函数作为第二个参数进行传入：

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We are dancing. " + msg); }
function jumping (msg) { console.log("We are jumping. " + msg); }

// Add two listeners to the same event
ourObject.on("move", dancing);
ourObject.on("move", jumping);

// Trigger the events. Both listeners are called.
ourObject.trigger("move", "Yeah!");

// Removes specified listener
ourObject.off("move", dancing);

// Trigger the events again. One listener left.
ourObject.trigger("move", "Yeah, jump, jump!");
```

最后，正如我们在前面的例子中所看到的，trigger 为指定事件（或用空格分隔的一组事件列表）触发回调函数，例如：

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function doAction (msg) { console.log("We are " + msg); }

// Add event listeners
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
ourObject.on("skip", doAction);

// Single event
ourObject.trigger("dance", 'just dancing.');

// Multiple events
ourObject.trigger("dance jump skip", 'very tired from so much action.');
```

trigger 可以为回调函数传递多个参数：

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function doAction (action, duration) {
  console.log("We are " + action + ' for ' + duration );
}

// Add event listeners
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
ourObject.on("skip", doAction);

// Passing multiple arguments to single event
ourObject.trigger("dance", 'dancing', "5 minutes");

// Passing multiple arguments to multiple events
ourObject.trigger("dance jump skip", 'on fire', "15 minutes");
```

3.6.2 listenTo()和 stopListening()

on()和 off()在观察对象上直接添加或移除回调函数，而 listenTo()则是让一个对象监听另外一个对象上的事件，允许该对象跟踪所监听的对象上的事件。之后可以调用 stopListening()来停止监听这些事件：

```
var a = _.extend({}, Backbone.Events);
var b = _.extend({}, Backbone.Events);
var c = _.extend({}, Backbone.Events);

// add listeners to A for events on B and C
a.listenTo(b, 'anything', function(event){
    console.log("anything happened"); });
a.listenTo(c, 'everything', function(event){
    console.log("everything happened"); });

// trigger an event
b.trigger('anything'); // logs: anything happened

// stop listening
a.stopListening();

// A does not receive these events
b.trigger('anything');
c.trigger('everything');
```

stopListening()也可以有选择地停止监听具体的事件、模型或回调处理程序。

使用 on()和 off()的时候，如果同时移除视图和对应模型的话，一般是没有问题的，

但如果所要移除的视图在模型上注册了事件监听，而我们没有删除模型或者移除该视图的事件处理程序，那么移除视图的时候就会引起错误。由于该模型有一个对视图的回调函数的引用，JavaScript 垃圾回收器无法从内存中删除视图。这被称为宿主视图（ghost view），又因为在整个应用程序声明周期内，模型通常会比相应的视图存活得久，所以这是一个常见的内存泄露。关于这方面内容的详细介绍和解决方案，请查看 Derick Bailey 的精彩文章（网址：http://bit.ly/ZN0Sci）。

事实上，对象上的每个 on()方法调用，在垃圾回收器回收的时候，都需要一个 off()方法调用。listenTo()却不一样，它允许视图绑定多个模型事件监听，而只需要调用一次 stopListening()就可以将这些绑定全部解除。

View.remove()的默认实现里调用了一次 stopListening()，以确保视图在销毁之前将所有通过 listenTo()绑定的监听器进行解除绑定。

```
var view = new Backbone.View();
var b = _.extend({}, Backbone.Events);

view.listenTo(b, 'all', function(){ console.log(true); });
b.trigger ('anything'); // logs: true

view.listenTo(b, 'all', function(){ console.log(false); });
view.remove(); // stopListening() implicitly called
b.trigger('anything');
// does not log anything
```

3.6.3 事件与视图

在一个视图中，有两种类型的事件可以监听：DOM 事件和使用事件 API（Event API）注册的事件。理解视图如何绑定这些事件和所调用回调函数中的上下文之间的区别非常重要。

可以使用视图的 events 属性或 jQuery.on()来绑定 DOM 事件。使用 events 属性绑定的事件，回调里的 this 指向的是视图对象，而直接使用 jQuery 绑定的所有事件，this 都会被 jQuery 设置为该 DOM 元素。jQuery 会为所有 DOM 元素的回调传入一个 event 对象，查看 Backbone 文档里的 delegateEvents()可以获取进一步的详细信息。

本节描述的使用事件 API 注册的事件绑定，如果在观察对象上使用 on()绑定事件，可以将上下文参数作为第三个参数进行传入；如果使用 listenTo()绑定事件，回调里的 this 指向的是监听器自身。给事件 API 所绑定的事件回调传入的参数，取决于事件的类型。详细信息请查看 Backbone 文档里的 Events 目录。

下面的示例说明了这些差异：

```
<div id="todo">
    <input type='checkbox' />
</div>

var View = Backbone.View.extend({

    el: '#todo',


    // bind to DOM event using events property
    events: {
        'click [type="checkbox"]': 'clicked',
    },

    initialize: function () {
        // bind to DOM event using jQuery
        this.$el.click(this.jqueryClicked);

        // bind to API event
        this.on('apiEvent', this.callback);
    },

    // 'this' is view
    clicked: function(event) {
        console.log("events handler for " + this.el.outerHTML);
        this.trigger('apiEvent', event.type);
    },

    // 'this' is handling DOM element
    jqueryClicked: function(event) {
        console.log("jQuery handler for " + this.outerHTML);
    },

    callback: function(eventType) {
        console.log("event type was " + eventType);
    }

});

var view = new View();
```

3.7 路由（Router）

在 Backbone 中，路由用于将 URL 地址（真实地址或哈希片段）和应用程序各部分连接在一起。程序里的任何部分，如果想做成书签标记、共享或者支持按钮回退的话，都需要一个单独的 URL 地址。

如下是使用哈希标记的路由示例：

```
http://example.com/#about
http://example.com/#search/seasonal-horns/page2
```

一个应用程序通常至少有一个路由将 URL 路由映射到一个函数上，用于判断用户访问这个路由时会发生什么。这种关系定义如下：

```
'route' : 'mappedFunction'
```

让我们通过扩展 Backbone.Router 来定义我们的第一个路由。为了配合本向导，我们继续假设要创建一个复杂的 todo 应用程序（类似个人管理/计划），它需要一个复杂的路由 TodoRouter。

注意下面代码示例中的行内注释，也是我们本节要讲的内容。

```
var TodoRouter = Backbone.Router.extend({
    /* define the route and function maps for this router */
    routes: {
        "about" : "showAbout",
        /* Sample usage: http://example.com/#about */

        "todo/:id" : "getTodo",
        /* This is an example of using a ":param" variable, which allows us to
        match any of the components between two URL slashes */
        /* Sample usage: http://example.com/#todo/5 */

        "search/:query" : "searchTodos",
        /* We can also define multiple routes that are bound to the same map
        function, in this case searchTodos(). Note below how we're optionally
        passing in a reference to a page number if one is supplied */
        /* Sample usage: http://example.com/#search/job */

        "search/:query/p:page" : "searchTodos",
        /* As we can see, URLs may contain as many ":param"s as we wish */
        /* Sample usage: http://example.com/#search/job/p1 */

        "todos/:id/download/*documentPath" : "downloadDocument",
        /* This is an example of using a *splat. Splats are able to match
        any number of URL components and can be combined with ":param"s*/
        /* Sample usage: http://example.com/#todos/5/download/todos.doc */

        /* If you wish to use splats for anything beyond default routing,
        it's probably a good idea to leave them at the end of a URL;
        otherwise, you may need to apply regular expression parsing
        on your fragment */

        "*other": "defaultRoute"
        /* This is a default route that also uses a *splat. Consider the
        default route a wildcard for URLs that are either not matched or where
        the user has incorrectly typed in a route path manually */
        /* Sample usage: http://example.com/# <anything> */,

        "optional(/:item)": "optionalItem",
```

```
        "named/optional/(y:z)": "namedOptionalItem"
        /* Router URLs also support optional parts via parentheses, without
           having to use a regex.  */
    },

    showAbout: function(){
    },

    getTodo: function(id){

        /*
        Note that the id matched in the above route will be passed to this
        function */
        console.log("You are trying to reach todo " + id);
    },

    searchTodos: function(query, page){
        var page_number = page || 1;
        console.log("Page number: " + page_number + " of the results for todos
        containing the word: " + query);
    },

    downloadDocument: function(id, path){
    },

    defaultRoute: function(other){
        console.log('Invalid. You attempted to reach:' + other);
    }
});

/* Now that we have a router setup, we need to instantiate it */

var myTodoRouter = new TodoRouter();
```

Backbone 通过 window.history.pushState 为 HTML5 的 pushState 提供选择性支持，这将允许我们定义像 http://backbonejs.org/just/an/example 这样的路由。当用户的浏览器不支持 pushState 时，它将自动接手并支持该功能。注意，如果可以在服务器上也支持 pushState，那就更好了，虽然有点儿难实现。

大家可能想知道使用路由的时候是否需要限制路由数量，Andrew de Andrade 指出，Backbone 的创建者 DocumentCloud 通常在绝大部分应用程序里只使用一个路由。在我们自己的项目里，路由可能不需要超过一到两个。大多数应用程序可以通过单一的路由来让程序保持组织性，而不会显得臃肿。

接下来，我们需要初始化 Backbone.history，因为它在应用程序里负责处理 hashchange 事件。它会自动处理已经定义过的路由，并且在这些路由被访问的时候自动触发相应的回调。

Backbone.history.start()将简单告诉 Backbone，可以正式开始监控所有的 hashchange 事件了，示例如下：

```
var TodoRouter = Backbone.Router.extend({
  /* define the route and function maps for this router */
  routes: {
    "about" : "showAbout",
    "search/:query" : "searchTodos",
    "search/:query/p:page" : "searchTodos"
  },

  showAbout: function(){},

  searchTodos: function(query, page){
    var page_number = page || 1;
    console.log("Page number: " + page_number + " of the results for todos
    containing the word: " + query);
  }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to and check console:
// http://localhost/#search/job/p3   logs: Page number: 3 of the results for
// todos containing the word: job
// http://localhost/#search/job      logs: Page number: 1 of the results for
// todos containing the word: job
// etc.
```

要运行上面的示例，需要创建本地开发环境以及测试项目，我们将在第 4 章讲述这些内容。

如果想在特定的点上通过更新 URL 来反映应用程序状态，可以使用路由的.navigate()方法。默认情况下，它只更新 URL 片段，而不触发 hashchange 事件：

```
// Let's imagine we would like a specific fragment
// (edit) once a user opens a single todo
var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... other routes
  },

  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit');
```

```
    // updates the fragment for us, but doesn't trigger the route
  },

  editTodo: function(id) {
    console.log("Edit todo opened.");
  }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to: http://localhost/#todo/4
//
// URL is updated to: http://localhost/#todo/4/edit
// but editTodo() function is not invoked even though location we end up
// is mapped to it.
//
// logs: View todo requested.
```

当然，在更新 URL 片段的时候，也可以通过传入 trigger:true 选项让 Router.navigate() 触发 hashchange 事件。

这种用法不太好，推荐前面示例中的做法：应用程序如果需要转换到特定的状态，为它创建可以书签化的 URL 地址即可。

```
var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... other routes
  },

  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit', {trigger: true});
// updates the fragment and triggers the route as well
  },

  editTodo: function(id) {
    console.log("Edit todo opened.");
  }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to: http://localhost/#todo/4
//
// URL is updated to: http://localhost/#todo/4/edit
// and this time editTodo() function is invoked.
```

```
//
// logs:
// View todo requested.
// Edit todo opened.
```

除了 Backbone.history 之外，路由改变也会触发 route 事件。

```
Backbone.history.on('route', onRoute);

// Trigger 'route' event on router instance."
router.on('route', function(name, args) {
  console.log(name === 'routeEvent');
});

location.replace('http://example.com#route-event/x');
Backbone.history.checkUrl();
```

3.8 Backbone 同步 API

前面我们讨论了 Backbone 如何通过模型上的 fetch()、save()、destroy()方法和集合上的 fetch()、create()方法来支持 RESTful 持久化。现在，让我们来仔细看看 Backbone 的 sync 方法，它是这些持久化操作的基础。

Backbone.sync 方法是 Backbone.js 不可或缺的一部分。它就像 jQuery 的$ ajax()方法，所以其 HTTP 参数也是基于 jQuery 的 API 来组织的。由于一些老的服务器可能不支持 JSON 格式的请求和 HTTP PUT 以及 DELETE 操作，我们可以通过对 Backbone 的两个配置参数进行设置来模仿这些请求。如下代码是它们的默认值配置：

```
Backbone.emulateHTTP = false;
// set to true if server cannot handle HTTP PUT or HTTP DELETE
Backbone.emulateJSON = false;
// set to true if server cannot handle application/json requests
```

如果服务器不支持扩展的 HTTP 方法，则 Backbone.emulateHTTP 选项需要设置为 true。如果服务器不能解释 JSON 的 MIME 类型，则 Backbone.emulateJSON 选项也需要设置为 true。

```
// Create a new library collection
var Library = Backbone.Collection.extend({
    url : function() { return '/library'; }
});

// Define attributes for our model
var attrs = {
    title  : "The Tempest",
    author : "Bill Shakespeare",
    length : 123
```

```
};

// Create a new library instance
var library = new Library;

// Create a new instance of a model within our collection
library.create(attrs, {wait: false});

// Update with just emulateHTTP
library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
  emulateHTTP: true
});

// Check the ajaxSettings being used for our request
console.log(this.ajaxSettings.url === '/library/2-the-tempest');
// true
console.log(this.ajaxSettings.type === 'POST'); // true
console.log(this.ajaxSettings.contentType === 'application/json');
// true

// Parse the data for the request to confirm it is as expected
var data = JSON.parse(this.ajaxSettings.data);
console.log(data.id === '2-the-tempest');  // true
console.log(data.author === 'Tim Shakespeare'); // true
console.log(data.length === 123); // true
```

类似的，我们也可以使用 emulateJSON 来更新：

```
library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
  emulateJSON: true
});

console.log(this.ajaxSettings.url === '/library/2-the-tempest'); // true
console.log(this.ajaxSettings.type === 'PUT'); // true
console.log(this.ajaxSettings.contentType ===
'application/x-www-form-urlencoded'); // true

var data = JSON.parse(this.ajaxSettings.data.model);
console.log(data.id === '2-the-tempest');
console.log(data.author ==='Tim Shakespeare');
console.log(data.length === 123);
```

每次 Backbone 尝试读取、保存、删除模型的时候，Backbone.sync 都要调用一次。它使用 jQuery 或 Zepto 的 ajax()实现来发送这些 RESTful 请求，但如果有需要，可以重载这些实现。

我们可以在全局上重载 Backbone.sync 的 sync 方法，也可以在 Backbone 集合或者单个模型上添加一个 sync 函数，以细粒度级别进行重载。

由于所有的持久化操作都是由 Backbone.sync 函数负责的，所以可以通过定义一个同名函数简单覆盖 Backbone.sync，来替换该持久化层：

```
Backbone.sync = function(method, model, options) {
};
```

下面的 methodMap 由标准的 sync 实现使用，用于将方法参数映射到 HTTP 操作上，并说明每个方法参数所需的行为类型：

```
var methodMap = {
  'create': 'POST',
  'update': 'PUT',
  'patch':  'PATCH',
  'delete': 'DELETE',
  'read':   'GET'
};
```

如果我们要仅仅用一个 HTTP 方法计算同步时的调用次数来替换 sync 标准实现，我们可以这么做：

```
var id_counter = 1;
Backbone.sync = function(method, model) {
  console.log("I've been passed " + method + " with " + JSON.stringify(model));
  if(method === 'create'){ model.set('id', id_counter++); }
};
```

注意，我们为新创建的模型指定了唯一 id。

我们需要覆盖 Backbone.sync 方法，以支持其他持久化后端。内置的方法是根据 RESTful JSON API 的一个特定形式来实现的——最初从 Ruby on Rails 应用程序抽象出来，这些应用程序用同样的方式、使用像 PUT 这样的 HTTP 方法。

调用 sync 方法的时候使用 3 个参数：

- method：create、update、patch、delete、read 中的一种。
- model：Backbone 模型对象。
- options：可能包括 success 和 error 方法。

我们可以像下面的模式这样实现一个新的 sync 方法：

```
Backbone.sync = function(method, model, options) {

  function success(result) {
    // Handle successful results from MyAPI
    if (options.success) {
      options.success(result);
    }
  }

  function error(result) {
    // Handle error results from MyAPI
```

```
      if (options.error) {
        options.error(result);
      }
    }

    options || (options = {});

    switch (method) {
      case 'create':
        return MyAPI.create(model, success, error);

      case 'update':
        return MyAPI.update(model, success, error);

      case 'patch':
        return MyAPI.patch(model, success, error);

      case 'delete':
        return MyAPI.destroy(model, success, error);

      case 'read':
        if (model.attributes[model.idAttribute]) {
          return MyAPI.find(model, success, error);
        } else {
          return MyAPI.findAll(model, success, error);
        }
    }
  };
```

该模式将 API 调用委托到一个新的对象（MyAPI），这可能是一个支持事件的 Backbone 风格的类。这样就可以安全地单独进行测试，并且除了 Backbone，也可以与其他库一起用。

这里有很多 sync 的实现。下面的示例在 GitHub 上都可以访问到：

- *Backbone* localStorage：持久化到浏览器的 localStorage 上。
- *Backbone offline*：支持离线工作。
- *Backbone Redis*：使用 Redis 的 key-value 进行存储。
- *backbone-parse*：将 Backbone 和 Parse.com 集成。
- backbone-websql：在 WebSQL 里保存数据。
- Backbone Caching Sync：为其他 sync 实现使用 localStorage 作为缓存。

3.9 依赖文件

Backbone.js 官方文档状态：Backbone 只有一个硬依赖，Underscore.js（>= 1.4.3）或者是 Lo-Dash。通过 Backbone.Router 和使用 Backbone.View 进行 DOM 操作，包括使用 json2.js 以及 jQuery（>= 1.7.0）或者 Zepto 来支持 RESTful 持久化和 history 功能。

这意味着，如果要使用卓越的模型进行操作，可能需要引用一个像 jQuery 或者 Zepto 这样的 DOM 操作库。Underscore 主要用于 Backbone 的实用方法（Backbone 严重依赖它），而 json2.js 用于为旧浏览器提供 JSON 支持（如果使用 Backbone.sync 的话）。

3.10 总结

在本章中，我向大家介绍了使用 Backbone 构建应用程序所需的组件：模型（model）、视图（view）、集合（collection）和路由（router）。我们探讨了 Backbone 使用的事件（Events）混入，以增强所有具有发布/订阅功能的组件，并了解它是如何在任意对象上使用的。最后，我们了解了 Backbone 如何使用 Underscore.js 和 jQuery/Zepto API，为 Backbone 集合和模型添加操作和持久化功能。

Backbone 有很多功能和选项已经超越我们在这里所介绍的功能，而且它在不断发展，所以一定要访问官方文档，了解更多相关细节和最新的功能。在下一章，大家将要实践起来，简单实现你的第一个 Backbone 应用程序。

第 4 章

练习 1：Todos——第一个 Backbone.js 应用程序

至此，我们已经掌握了 Backbone 的相关基础，让我们来编写第一个 Backbone.js 应用程序。我们将构建 TodoMVC.com 上展示的 Backbone Todo List 应用程序。构建 todo 列表是学习 Backbone 规定的一种很好的方式（见图 4-1）。这是一个相对简单的应用程序，但围绕着绑定、持久化模型数据、路由、模板渲染这些技术调整，也提供了一个阐明 Backbone 核心功能的机会。

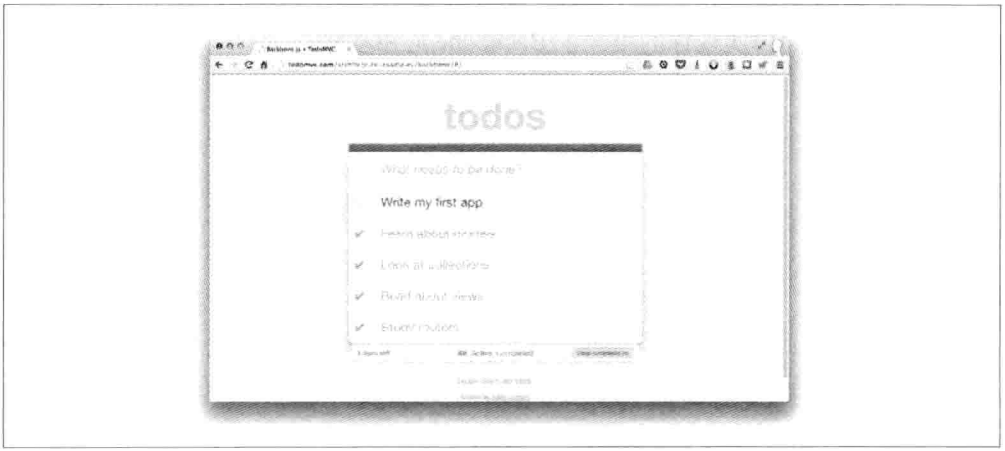


图 4-1　todo 列表——要编写的第一个 Backbone.js 应用程序

让我们站在比较高的角度考虑一下应用程序的架构。我们需要如下功能：

- 描述单个 todo 项的 Todo 模型；
- 存储和持久化 todo 项的 TodoList 集合；
- 创建 todo 项；
- 展示 todo 列表；
- 编辑现有的 todo 项；
- 标记一个 todo 项已完成状态（completed）；
- 删除 todo 项；
- 过滤所有已完成（或未完成）todo 列表。

从本质上讲，这些功能是典型的 CRUD（创建、读取、更新、删除）方法。让我们开始！

4.1 静态 HTML

我们将所有的 HTML 都放在一个名为 index.html 的文件里。

4.1.1 HTML 头部和 Script 脚本

首先，我们要设置页面标题和应用程序的基本依赖文件：jQuery、Underscore、Backbone.js 和 Backbone localStorage adapter（地址：http://bit.ly/16dX4op）。

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Backbone.js • TodoMVC</title>
  <link rel="stylesheet" href="assets/base.css">
</head>
<body>
  <script type="text/template" id="item-template"></script>
  <script type="text/template" id="stats-template"></script>
  <script src="js/lib/jquery.min.js"></script>
  <script src="js/lib/underscore-min.js"></script>
  <script src="js/lib/backbone-min.js"></script>
  <script src="js/lib/backbone.localStorage.js"></script>
  <script src="js/models/todo.js"></script>
  <script src="js/collections/todos.js"></script>
  <script src="js/views/todos.js"></script>
```

```
    <script src="js/views/app.js"></script>
    <script src="js/routers/router.js"></script>
    <script src="js/app.js"></script>
  </body>
  </html>
```

注意，除了上述的依赖性文件外，页面也加载了应用程序相关的特定文件。这些文件被组织到了不同的文件夹，表示不同的应用程序职责：models、views、collections 和 routers。app.js 是整个应用程序的核心初始化代码。

要继续操作，需要创建 index.html 文件里展示的目录结构：

（1）将 index.html 放在顶级目录。

（2）从各自的官方站点下载 jQuery、Underscore、Backbone 和 Backbone localStorage，然后放到 js/iib 目录。

（3）创建 js/models、js/collections、js/views 和 js/routers 目录。

另外还需要将 **bass.css**（http://bit.ly/YePkgQ）和 **bg.png**（http://bit.ly/11YarU3）文件放置于 assets 目录中。请记住，可以在 TodoMVC.com 上查看应用程序的最终 demo 版本。

在该教程里，我们将创建应用程序的 JavaScript 文件。不必担心页面里的两个 text/template script 元素——很快我们就会替掉它！

4.1.2 应用程序 HTML

现在让我们来填充 index.html 的 body 元素，我们需要一个创建 todo 项的输入框 <input>、一个列出 todo 列表的<ul id="todo-list" />，以及稍后可以添加统计信息和链接操作（如，清空已完成的 todo 项）的底部 footer。在<body>标签里的 script 脚本元素前面，立即把这些 HTML 标记添加上：

```
<section id="todoapp">
  <header id="header">
    <h1>todos</h1>
    <input id="new-todo" placeholder="What needs to be done?" autofocus>
  </header>
  <section id="main">
    <input id="toggle-all" type="checkbox">
    <label for="toggle-all">Mark all as complete</label>
    <ul id="todo-list"></ul>
  </section>
  <footer id="footer"></footer>
</section>
<div id="info">
```

```
    <p>Double-click to edit a todo</p>
    <p>Written by <a href="https://github.com/addyosmani">Addy Osmani</a></p>
    <p>Part of <a href="http://todomvc.com">TodoMVC</a></p>
  </div>
```

4.1.3 模板

要完成 index.html，我们还需要添加模板，用于将模型数据注入到模板里，动态生成 HTML。在页面里包含模板的一种方式是使用自定义<script>标签。浏览器不会执行这些标签，而只会将其作为纯文本。Underscore 微模板可以访问这些模板，并将其渲染成 HTML 片段。

我们开始填充#item-template，用于显示单个的 todo 项。

```
<!-- index.html -->

<script type="text/template" id="item-template">
  <div class="view">
    <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
    <label><%- title %></label>
    <button class="destroy"></button>
  </div>
  <input class="edit" value="<%- title %>">
</script>
```

上面例子中的模板标签，如“<%=”和“<%–”，是 Underscore.js 的特定标签，在 Underscore 站点上有文档描述。我们可以在自己的应用程序中选择其他的模板库，比如 Mustache 或 Handlebars。可以使用任何一种你喜欢的方法，Backbone 不会有影响。

我们还需要定义#stats-template，用于将其填充到底部 footer。

```
<!-- index.html -->

<script type="text/template" id="stats-template">
  <span id="todo-count"><strong><%= remaining %></strong>
  <%= remaining === 1 ? 'item' : 'items' %> left</span>
  <ul id="filters">
    <li>
      <a class="selected" href="#/">All</a>
    </li>
    <li>
      <a href="#/active">Active</a>
    </li>
    <li>
      <a href="#/completed">Completed</a>
    </li>
  </ul>
  <% if (completed) { %>
  <button id="clear-completed">Clear completed (<%= completed %>)</button>
  <% } %>
</script>
```

#stats-template 用于显示还未完成的 todo 项的数量，它包含一系列的超链接，这些链接在我们实现 router 的时候用于执行操作；还包含一个按钮，用于清除所有已完成的 todo 项。

现在所有需要的 HTML 都已经有了，那就从最基本的 Todo 模型开始来实现我们的应用程序。

4.2 Todo 模型

Todo 模型非常简单。首先，一个 todo 项有两个属性：用于保持 todo 项标题的 title 和标识其是否完成的 completed。这些属性作为默认值传入，如下所示：

```
// js/models/todo.js

var app = app || {};

// Todo Model
// ----------
// Our basic **Todo** model has 'title', 'order', and 'completed' attributes.

app.Todo = Backbone.Model.extend({

  // Default attributes ensure that each todo created has 'title' and
  // 'completed' keys.
  defaults: {
    title: '',
    completed: false
  },

  // Toggle the 'completed' state of this todo item.
  toggle: function() {
    this.save({
      completed: !this.get('completed')
    });
  }

});
```

其次，Todo 模型有一个 toggle()方法，通过该方法可以设置并保持一个 todo 项的完成状态。

4.3 Todo 集合

接下来，TodoList 集合用于组织我们的模型。该集合使用 localStorage 适配器来覆盖 Backbone 的默认 sync()操作，用于将 todo 项保存到 HTML5 的 localStorage 里。

通过使用 localStorage，可以在页面请求间保存它们。

```
// js/collections/todos.js

var app = app || {};

// Todo Collection
// ---------------

// The collection of todos is backed by *localStorage* instead of a remote
// server.
var TodoList = Backbone.Collection.extend({

  // Reference to this collection's model.
  model: app.Todo,

  // Save all of the todo items under the `"todos-backbone"` namespace.
  // Note that you will need to have the Backbone localStorage plug-in
  // loaded inside your page in order for this to work. If testing
  // in the console without this present, comment out the next line
  // to avoid running into an exception.
  localStorage: new Backbone.LocalStorage('todos-backbone'),

  // Filter down the list of all todo items that are finished.
  completed: function() {
    return this.filter(function( todo ) {
      return todo.get('completed');
    });
  },

  // Filter down the list to only todo items that are still not finished.
  remaining: function() {
  // apply allowsus to define the context of this within our function scope
    return this.without.apply( this, this.completed() );
  },

  // We keep the Todos in sequential order, despite being saved by unordered
  // GUID in the database. This generates the next order number for new items.
  nextOrder: function() {
    if ( !this.length ) {
      return 1;
    }
    return this.last().get('order') + 1;
  },

  // Todos are sorted by their original insertion order.
  comparator: function( todo ) {
    return todo.get('order');
  }
});

// Create our global collection of **Todos**.
app.Todos = new TodoList();
```

集合的 completed()和 remaining()方法分别返回一个已完成 todo 项和未完成 todo 项的数组。

nextOrder()方法实现了一个序列产生器；同时 comparator()方法通过其 todo 项的插入顺序（order）对集合进行排序。

this.filter、this.without 和 this.last 是混入到 Backbone.Collection 的 Underscore 方法，这样读者可以找到它们的更多相关资料。

4.4 应用程序视图（AppView）

让我们来检查定义在视图里的应用程序核心逻辑。每个视图都支持相应的功能，如即时点击编辑（edit-in-place），因此也都包含一堆逻辑。为了帮助组织这些逻辑，我们使用元素控制器模式。元素控制器模式包含两个视图：一个控制 todo 集合，另一个处理单个的 todo 项。

在我们的例子中，AppView 将负责处理新 todo 项的创建以及初始 todo 列表的渲染。TodoView 实例将和单个独立的 todo 记录相关联。Todo 实例负责编辑、更新和销毁相关的 todo 项。

为了让一切保持简明扼要，我们不会在本教程中实现应用程序的所有功能，而只介绍有助于大家入门的功能。即便如此，我们也会在 AppView 里覆盖很多内容，所以我们将讨论内容分成了两个部分。

```
// js/views/app.js

var app = app || {};

// The Application
// ---------------

// Our overall **AppView** is the top-level piece of UI.
app.AppView = Backbone.View.extend({

  // Instead of generating a new element, bind to the existing skeleton of
  // the app already present in the HTML.
  el: '#todoapp',
  // Our template for the line of statistics at the bottom of the app.
  statsTemplate: _.template( $('#stats-template').html() ),

  // At initialization we bind to the relevant events on the `Todos`
  // collection, when items are added or changed.
```

```
    initialize: function() {
      this.allCheckbox = this.$('#toggle-all')[0];
      this.$input = this.$('#new-todo');
      this.$footer = this.$('#footer');
      this.$main = this.$('#main');

      this.listenTo(app.Todos, 'add', this.addOne);
      this.listenTo(app.Todos, 'reset', this.addAll);
    },

    // Add a single todo item to the list by creating a view for it, and
    // appending its element to the `<ul>`.
    addOne: function( todo ) {
      var view = new app.TodoView({ model: todo });
      $('#todo-list').append( view.render().el );
    },

    // Add all items in the **Todos** collection at once.
    addAll: function() {
      this.$('#todo-list').html('');
      app.Todos.each(this.addOne, this);
    }

});
```

在 AppView 的初始版本里，有几个值得注意的功能，其中包括 statsTemplate、初始化时隐式调用的 initialize 方法，以及其他几个 view 特定的方法。

el 属性保存 id 为 todoapp 的 DOM 元素的一个引用。在我们的应用程序示例中，el 指向 index.html 里的<section id="todoapp" />元素。

使用 Underscore 微模板的_.template 调用，从#stats-template 模板构建了一个 statsTemplate 对象。稍后在渲染视图的时候，我们会使用到该模板。

现在，让我们看看 initialize 函数。首先，它使用 jQuery 将元素缓存到局部变量上（回想一下，this.$()查找的是 this.$el 上相关的元素）。然后，在 Todo 集合上绑定了两个事件：add 和 reset。更新和删除操作，我们是委托给 TodoView 视图进行处理的，所以不需要担心这两个操作。add 和 reset 的逻辑如下：

- 当 add 事件触发时，addOne()方法被调用，并传入新的模型。addOne()创建一个 TodoView 实例，并进行渲染，然后将渲染结果附加到 todo 列表里。
- 当 reset 事件触发时（从 localStorage 加载 todo 数据，批量更新集合时），addAll()方法被调用，并对当前集合的所有 todo 项进行迭代，然后触发每个 todo 项的 addOne()方法。

注意，我们可以使用 addAll()方法里的 this 来表示 AppView 视图，因为 listenTo()方法在创建绑定的时候，隐式地将回调的上下文设置成了该视图。

现在，继续添加一些逻辑来完成 AppView：

```
// js/views/app.js

var app = app || {};

// The Application
// ---------------

// Our overall **AppView** is the top-level piece of UI.
app.AppView = Backbone.View.extend({

  // Instead of generating a new element, bind to the existing skeleton of
  // the app already present in the HTML.
  el: '#todoapp',

  // Our template for the line of statistics at the bottom of the app.
  statsTemplate: _.template( $('#stats-template').html() ),

  // New
  // Delegated events for creating new items, and clearing completed ones.
  events: {
    'keypress #new-todo': 'createOnEnter',
    'click #clear-completed': 'clearCompleted',
    'click #toggle-all': 'toggleAllComplete'
  },

  // At initialization we bind to the relevant events on the `Todos`
  // collection, when items are added or changed. Kick things off by
  // loading any preexisting todos that might be saved in *localStorage*.
  initialize: function() {
    this.allCheckbox = this.$('#toggle-all')[0];
    this.$input = this.$('#new-todo');
    this.$footer = this.$('#footer');
    this.$main = this.$('#main');

    this.listenTo(app.Todos, 'add', this.addOne);
    this.listenTo(app.Todos, 'reset', this.addAll);

    // New
    this.listenTo(app.Todos, 'change:completed', this.filterOne);
    this.listenTo(app.Todos,'filter', this.filterAll);
    this.listenTo(app.Todos, 'all', this.render);

    app.Todos.fetch();
  },

  // New
  // Rerendering the app just means refreshing the statistics -- the rest
```

```
// of the app doesn't change.
render: function() {
  var completed = app.Todos.completed().length;
  var remaining = app.Todos.remaining().length;

  if ( app.Todos.length ) {
    this.$main.show();
    this.$footer.show();

    this.$footer.html(this.statsTemplate({
      completed: completed,
      remaining: remaining
    }));

    this.$('#filters li a')
      .removeClass('selected')
      .filter('[href="#/' + ( app.TodoFilter || '' ) + '"]')
      .addClass('selected');
  } else {
    this.$main.hide();
    this.$footer.hide();
  }

  this.allCheckbox.checked = !remaining;
},

// Add a single todo item to the list by creating a view for it, and
// appending its element to the `<ul>`.
addOne: function( todo ) {
  var view = new app.TodoView({ model: todo });
  $('#todo-list').append( view.render().el );
},

// Add all items in the **Todos** collection at once.
addAll: function() {
  this.$('#todo-list').html('');
  app.Todos.each(this.addOne, this);
},

// New
filterOne : function (todo) {
  todo.trigger('visible');
},
// New
filterAll : function () {
  app.Todos.each(this.filterOne, this);
},


// New
// Generate the attributes for a new todo item.
newAttributes: function() {
  return {
```

```
        title: this.$input.val().trim(),
        order: app.Todos.nextOrder(),
        completed: false
      };
    },

    // New
    // If you hit return in the main input field, create new Todo model,
    // persisting it to localStorage.
    createOnEnter: function( event ) {
      if ( event.which !== ENTER_KEY || !this.$input.val().trim() ) {
        return;
      }

      app.Todos.create( this.newAttributes() );
      this.$input.val('');
    },

    // New
    // Clear all completed todo items, destroying their models.
    clearCompleted: function() {
      _.invoke(app.Todos.completed(), 'destroy');
      return false;
    },

    // New
    toggleAllComplete: function() {
      var completed = this.allCheckbox.checked;

      app.Todos.each(function( todo ) {
        todo.save({
          'completed': completed
        });
      });
    }
  });
```

我们已经为创建 todo 项、编辑 todo 项、基于完成的状态过滤 todo 项功能添加了逻辑。

我们已经为 DOM 事件定义了一个包含声明式回调的 events 哈希。它将这些事件绑定到了如下方法。

1. createOnEnter()

用户在<input/>元素上按回车键的时候，创建一个新的 Todo 模型，并将它保存到 localStorage 中，并且重置<input/>元素，以便创建下一个 todo 项。Todo 模型通过 newAttributes()进行填充，该方法返回了一个由新 todo 项的 title、order、completed 组成的对象字面量。注意，因为回调是通过 events 哈希绑定的，所以 this 指向的是视图，而不是 DOM 元素。

2. clearCompleted()

当用户选中 clear-completed 复选框（该复选框由#stats-template 填充在 footer 里）的时候，删除 todo 列表里所有标记为已完成的 todo 项。

3. toggleAllComplete()

通过选中 toggle-all 复选框，允许用户将 todo 列表中的所有 todo 项都标记为已完成。

4. initialize()

为其他几个额外的事件绑定了回调：

- 在 Todo 集合上的 change:completed 事件绑定一个 filterOne()回调，该绑定监听集合中任意一个模型的 completed 标记，受影响的 todo 会传入给该回调，该回调会在该模型上触发一个 visible 自定义事件。
- 为 filter 事件绑定一个 filterAll()回调，工作方式有点像 addOne()和 addAll()。它的职责是：通过调用 filterOne()，让 UI 页面上符合选择条件（all、completed、remaining）的 todo 项变成可见的。
- 使用特殊的 all 事件将 Todos 集合上触发的任何事件绑定至视图的渲染方法（稍后将做讨论）。

initialize()方法用于从 localStorage 获取之前保存的 todo 项。

以下这些情况在 render()方法里发生。

（1）#main 和#footer 部分的显示和隐藏，取决于集合里是否包含 todo 项。

（2）footer 页面的 HTML 代码填充，是由 statsTemplate 以及已完成 todo 项的数字 completed 和未完成的数字 remaining 产生的。

（3）footer 的 HTML 里包含了一系列过滤链接。app.TodoFilter 的值将由我们的路由来设置，用于将所选样式类应用于与目前所选择过滤条件对应的链接条目上，其结果是将条件式 CSS 样式应用于该过滤条件。

（4）allCheckbox 的更新取决于是否还有剩余的 todo 项。

4.5 独立的待办项视图（TodoView）

现在让我们来看看 TodoView 视图。该视图将负责单独的 todo 记录，以确保 todo

更新的时候 view 也进行相应的更新。要启用该功能，需要在视图上添加事件监听器，在 todo 的 HTML 显示上监听事件。

```
// js/views/todos.js

var app = app || {};

// Todo Item View
// --------------

// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName: 'li',

  // Cache the template function for a single item.
  template: _.template( $('#item-template').html() ),

  // The DOM events specific to an item.
  events: {
    'dblclick label': 'edit',
    'keypress .edit': 'updateOnEnter',
    'blur .edit': 'close'
  },

  // The TodoView listens for changes to its model, rerendering. Since there's
  // a one-to-one correspondence between a **Todo** and a **TodoView** in this
  // app, we set a direct reference on the model for convenience.
  initialize: function() {
    this.listenTo(this.model, 'change', this.render);
  },

  // Rerenders the titles of the todo item.
  render: function() {
    this.$el.html( this.template( this.model.toJSON() ) );
    this.$input = this.$('.edit');
    return this;
  },

  // Switch this view into `"editing"` mode, displaying the input field.
  edit: function() {
    this.$el.addClass('editing');
    this.$input.focus();
  },

  // Close the `"editing"` mode, saving changes to the todo.
  close: function() {
    var value = this.$input.val().trim();

    if ( value ) {
      this.model.save({ title: value });
```

```
    }

    this.$el.removeClass('editing');
  },

  // If you hit `enter`, we're through editing the item.
  updateOnEnter: function( e ) {
    if ( e.which === ENTER_KEY ) {
      this.close();
    }
  }
});
```

在 initialize()构造函数里，我们设置了一个监听器，用于监听 Todo 模型的 change 事件。其结果是：todo 更新的时候，应用程序会重新渲染视图，并且直观地反映其变化。注意，Todo 模型是通过 AppView 在 arguments 参数里传递进来的，并且在 this.model 上自动可用。

在 render()方法中，使用 Underscore.js 渲染#item-template，该模板之前通过 Underscore 的_.template()方法被编译到 this.template 里。返回的 HTML 片段将替换视图元素（基于 tagName 属性隐式创建的 li 元素）的内容。换句话说，渲染后的模板现在是在 this.el 下，并且可以附加到用户界面的 todo 列表里。将实例化模板里的 input 元素缓存到 this.input 后，render()方法结束。

events 哈希包含以下 3 个回调。

1. edit()

用户双击 todo 列表中的一个现有 todo 项时，edit()将当前项的查看模式切换成编辑模式，以便用户可以修改当前 todo 项的 title 属性。

2. updateOnEnter()

检查用户是否按了回车键，然后执行 close()函数。

3. close()

将<input/>元素字段的文本值的前后空格进行过滤，确保如果不输入任何其他文本则不进行处理（例如，‘’）。如果提供的值有效，则将该值保持至当前 Todo 模型，并且通过移除相应的 CSS 类关闭编辑模型。

4.6 程序启动

至此，我们已经有了两个视图了：AppView 和 TodoView。前者需要在页面加载的

时候进行实例化，它的代码才会被执行。我们可以通过 jQuery 的 ready()函数实现这个目的，在 DOM 加载时，该函数会执行一个函数。

```
// js/app.js

var app = app || {};
var ENTER_KEY = 13;

$(function() {

  // Kick things off by creating the **App**.
  new app.AppView();

});
```

4.7 实战操作

让我们暂停一下，以确保到目前为止完成的工作都会像预期的那样有效。

在浏览器中打开 file://*path*/index.html 文件，并监控控制台，除了我们还没有创建的 router.js 文件外，我们不应该看到任何 JavaScript 错误。因为我们还没有创建任何 todo 记录，所以 todo 列表应该是空的。另外，在用户界面的功能完全可用之前，我们还需要继续做一些额外的工作。

不过，在 JavaScript 控制台已经可以测试一些东西了。

在控制台添加一个新的 todo 项：“window.app.Todos.create({ title: 'My first Todo items'});”，然后按回车键（见图 4-2）。

图 4-2 通过 JavaScript 控制台添加一个新 todo 项

如果一切运行正常，我们刚刚新建的 todo 项就会保存在 Todo 集合里。新创建的 todo 项也会保存到 localStorage 里，并且在页面上刷新显示。

window.app.Todos.create()执行集合方法，Collection.create(attributes, [options])用于初始化一个新的模型项，该模型的类型是在集合中定义的模型类型——本例是 app.Todo：

```
// from our js/collections/todos.js

var TodoList = Backbone.Collection.extend({

    model: app.Todo // the model type used by collection.create() to
// instantiate new model in the collection
    ...
)};
```

在控制台运行如下代码，并检查结果：

```
var secondTodo = window.app.Todos.create({ title: 'My second Todo item'});
secondTodo instanceof app.Todo // returns true
```

现在刷新页面，应该就能看到我们的劳动成果了。

由于添加的 todo 项已经保存在 localStorage 里了，所以仍应该显示在列表上。同时，我们也应该能通过输入标题然后按回车键来创建一个新 todo 项（见图 4-3）。

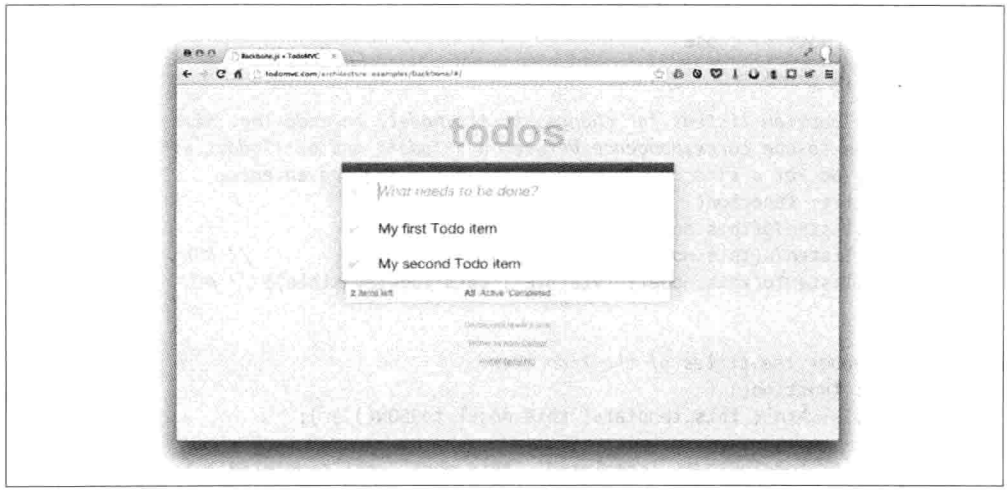


图 4-3　添加的新 todo 项

好极了！我们取得了很大的进步，但如何将 todo 项标记为已完成以及删除 todo 项呢？

4.8 标记完成或删除 todo 项

接下来的部分将讲述标记完成和删除 todo 项。这两种操作都特定于单个 todo 项目，所以我们需要将此功能添加到 TodoView 视图中。通过添加 togglecompleted()和 clear()方法，以及在 events 哈希里添加对应的事件和回调来实现上述两种操作。

```
// js/views/todos.js

var app = app || {};

// Todo Item View
// --------------

// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName: 'li',
  // Cache the template function for a single item.
  template: _.template( $('#item-template').html() ),

  // The DOM events specific to an item.
  events: {
    'click .toggle': 'togglecompleted', // NEW
    'dblclick label': 'edit',
    'click .destroy': 'clear',          // NEW
    'keypress .edit': 'updateOnEnter',
    'blur .edit': 'close'
  },

  // The TodoView listens for changes to its model, rerendering. Since there's
  // a one-to-one correspondence between a **Todo** and a **TodoView** in this
  // app, we set a direct reference on the model for convenience.
  initialize: function() {
    this.listenTo(this.model, 'change', this.render);
    this.listenTo(this.model, 'destroy', this.remove);        // NEW
    this.listenTo(this.model, 'visible', this.toggleVisible); // NEW
  },

  // Rerender the titles of the todo item.
  render: function() {
    this.$el.html( this.template( this.model.toJSON() ) );

    this.$el.toggleClass( 'completed', this.model.get('completed') ); // NEW
    this.toggleVisible();                                             // NEW

    this.$input = this.$('.edit');
    return this;
  },

  // NEW - Toggles visibility of item
```

```
    toggleVisible : function () {
      this.$el.toggleClass( 'hidden',  this.isHidden());
    },

    // NEW - Determines if item should be hidden
    isHidden : function () {
      var isCompleted = this.model.get('completed');
      return ( // hidden cases only
        (!isCompleted && app.TodoFilter === 'completed')
        || (isCompleted && app.TodoFilter === 'active')
      );
    },

    // NEW - Toggle the `"completed"` state of the model.
    togglecompleted: function() {
      this.model.toggle();
    },
    // Switch this view into `"editing"` mode, displaying the input field.
    edit: function() {
      this.$el.addClass('editing');
      this.$input.focus();
    },

    // Close the `"editing"` mode, saving changes to the todo.
    close: function() {
      var value = this.$input.val().trim();

      if ( value ) {
        this.model.save({ title: value });
      } else {
        this.clear(); // NEW
      }

      this.$el.removeClass('editing');
    },

    // If you hit `enter`, we're through editing the item.
    updateOnEnter: function( e ) {
      if ( e.which === ENTER_KEY ) {
        this.close();
      }
    },

    // NEW - Remove the item, destroy the model from
    // *localStorage* and delete its view.
    clear: function() {
      this.model.destroy();
    }
  });
```

这部分的关键是我们添加的两个事件处理程序，一个是 todo 复选框上的 togglecompleted 事件，另一个是<button class="destroy" />按钮上的 click 事件。

让我们来看看选中 todo 项的 checkbox 复选框时发生的事件：

（1）togglecompleted()函数被调用，该函数会调用 Todo 模型上的 toggle()方法。

（2）toggle()切换所选 todo 项的完成状态，并调用 Todo 模型上的 save()方法。

（3）保存操作在模型上产生了一个 change 事件，该事件绑定在 TodoView 的 render()方法上。在 render()里，我们添加了一行语句，用于根据模型的 completed 状态在该元素上切换 completed 样式。如果一个 todo 已经完成，相关的 CSS 样式就会改变 title 文本的颜色，并给文本加上删除线。

（4）保存操作也会在模型上产生一个 change:completed 事件，该事件由 AppView 的 filterOne()方法负责处理。回顾一下 AppView，我们看到 filterOne()会在模型上触发一个 visible 事件。该事件结合路由和集合的过滤功能一起使用，以便只显示符合 completed 过滤条件的 todo 项。在更新的 TodoView 里，我们将模型的 visible 事件绑定到了 toggleVisible()方法。该方法使用新加的 isHidden()方法判断 todo 项是否应该可见，并进行相应的更新。

现在，让我们来看看单击 todo 项的 destroy 按钮时会发生什么事情：

（1）clear()方法被调用，该方法会调用 Todo 模型上的 destroy()方法。

（2）从 localStorage 删除 todo 项，并触发 destroy 事件。

（3）在更新的 TodoView 里，我们将 destroy 事件绑定到了视图继承的 remove()方法上。该方法删除视图，并自动将相关联的元素从 DOM 中删除。由于我们使用了 listenTo()将视图的监听者绑定到该模型上，remove()方法也会将这些监听回调从模型上进行解绑，以确保不发生内存泄漏。

（4）destroy()还会将模型从集合中删除，并在集合上触发一个 remove 事件。

（5）由于 AppView 的 render()方法绑定了集合的所有事件，所以视图会重新被渲染，并且底部的统计信息也会被更新。

这就是上述两个功能的所有介绍了。

如果要看这两个功能的示例代码，请查阅完整代码（网址：http://bit.ly/11eV9ir）。

4.9 Todo 路由

最后，我们转向路由，路由将允许我们很容易地过滤列表中的活动项以及已经完成的项（见图 4-4）。我们将支持如下路由：

```
#/ (all - default)
#/active
#/completed
```

图 4-4　已完成 todo 项的过滤列表

路由改变时，todo 列表将在模型层级上进行过滤，并且就像刚才描述的那样，所选择的底部过滤链接的样式会被切换。一个模型被更新时，active 过滤条件的结果也会相应地更新（例如，如果过滤条件是 active，同时一个模型项又被选中，则该模型项会隐藏）。active 过滤在重新加载的时候进行持久化操作。

```
// js/routers/router.js

// Todo Router
// ----------

var Workspace = Backbone.Router.extend({
  routes:{
    '*filter': 'setFilter'
  },

  setFilter: function( param ) {
    // Set the current filter to be used

    // Trigger a collection filter event, causing hiding/unhiding
    // of Todo view items
    window.app.Todos.trigger('filter');
  }
});

app.TodoRouter = new Workspace();
Backbone.history.start();
```

我们使用*splat 设置了一个默认路由，用于将 URL 中“#/”后面的字符串传递给 setFilter()方法。setFilter()方法用于将 window.app.TodoFilter 设置到该字符串。

在 window.app.Todos.trigger('filter')这一行可以看到，一旦 filter 过滤被设置，我们只是在 Todo 集合上触发了 filter 事件，以判断哪些 todo 项是可见的、哪些是隐藏的。回想一下，AppView 的 filterAll()方法是绑定到集合的 filter 事件上的，并且集合上的任何事件都将导致 AppView 重新渲染。

最后，我们创建一个路由实例，然后调用 Backbone.history.start()，在页面加载时路由到初始 URL 上。

4.10 总结

至此，我们已经构建了第一个完整的 Backbone.js 应用程序。大家可以访问 TodoMVC 网站，在线查看完整程序的最新版本，并且源代码随时可用。

在第 8 章里，我们将学习如何使用 RequireJS 进一步模块化该应用程序，以及如何将我们的持久层切换到后端数据库，最后再使用几个不同的测试框架对该应用程序进行单元测试。

第5章

练习2：Book Library——第一个RESTful风格的Backbone.js应用程序

第一个应用程序让我们很好地尝试了如何编写 Backbone.js 应用程序，但大多数真实的应用程序都要与后端进行通信。让我们通过另外一个例子来巩固我们所学的知识，但这次我们将为应用程序创建一个 RESTful API 用于通信。

在该练习中，我们将使用 Backbone 构建一个用于管理电子书的图书馆应用程序。每本书我们将存储标题（title）、作者（author）、发布日期（releaseDate）和一些关键字（keywords）；还会存储每本书的一幅封面。

5.1 程序建立

首先，需要为我们的项目创建文件夹结构。为保证前端和后端独立，我们在项目根目录下为客户端项目创建一个 site 文件夹。在该文件夹里，我们将创建 css、img 以及 js 目录。

和上一个示例相同，我们将按功能分隔 JavaScript 文件，所以在 js 目录里创建了名为 lib、models、collections、views 的文件夹。目录层次结构如下：

```
site/
    css/
    img/
    js/
        collections/
        lib/
        models/
        views/
```

下载 Backbone、Underscore 和 jQuery 库，然后将其复制到 js/lib 文件夹中。我们还需要为图书封面准备一个占位图。保存这张图片（见图 5-1）到 site/img 文件夹中。

图 5-1　Eloquent JavaScript——为图书封面准备的占位图

像之前一样，我们需要在 site/index.html 文件里加载所有的依赖文件：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8"/>
        <title>Backbone.js Library</title>
        <link rel="stylesheet" href="css/screen.css">
    </head>
    <body>
        <script src="js/lib/jquery.min.js"></script>
        <script src="js/lib/underscore-min.js"></script>
        <script src="js/lib/backbone-min.js"></script>
        <script src="js/models/book.js"></script>
        <script src="js/collections/library.js"></script>
        <script src="js/views/book.js"></script>
        <script src="js/views/library.js"></script>
        <script src="js/app.js"></script>
    </body>
</html>
```

我们还应该添加 HTML 用户界面。我们需要一个表单，用于添加新书，所以应在 body 元素里添加如下代码：

```
<div id="books">
    <form id="addBook" action="#">
        <div>
            <label for="coverImage">CoverImage: </label>
            <input id="coverImage" type="file" />
            <label for="title">Title: </label><input id="title" type="text" />
            <label for="author">Author: </label><input id="author" type="text" />
            <label for="releaseDate">Release date: </label>
            <input id="releaseDate" type="text" />
            <label for="keywords">Keywords: </label>
            <input id="keywords" type="text" />
            <button id="add">Add</button>
        </div>
```

```
    </form>
</div>
```

我们还需要一个模板，用于显示每一本书，它应该放在<script>标签前面：

```
<script id="bookTemplate" type="text/template">
    <img src="<%= coverImage %>"/>
    <ul>
        <li><%= title %></li>
        <li><%= author %></li>
        <li><%= releaseDate %></li>
        <li><%= keywords %></li>
    </ul>

    <button class="delete">Delete</button>
</script>
```

要想看看有数据的时候页面是什么样子的，可以在 div 元素的 books 里手动添加一些书的信息。

```
<div class="bookContainer">
    <img src="img/placeholder.png"/>
    <ul>
        <li>Title</li>
        <li>Author</li>
        <li>Release Date</li>
        <li>Keywords</li>
    </ul>

    <button class="delete">Delete</button>
</div>
```

在浏览器中打开该文件时，显示页面应该和图 5-2 类似。

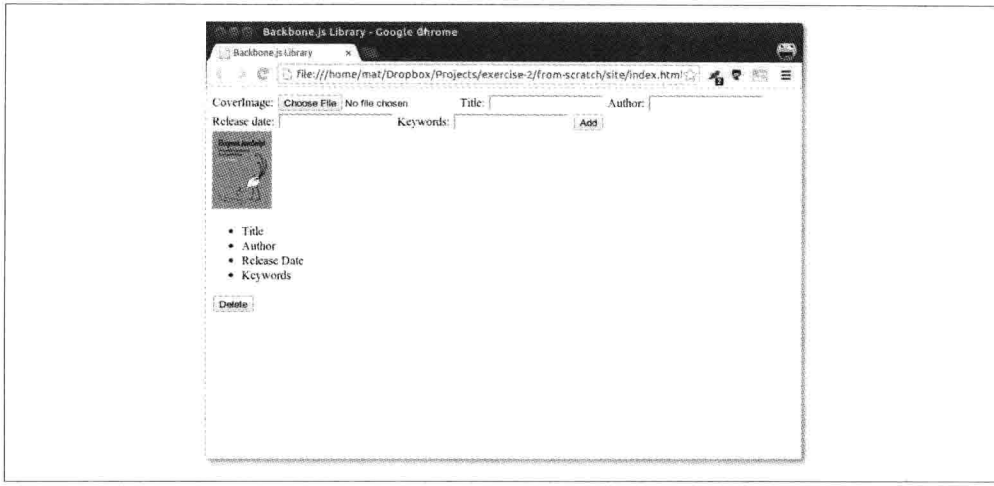


图 5-2　程序初始布局

这个页面布局不太好看。虽然这不是 CSS 教程，但是我们依然需要做一些格式化。在 site/css 文件里创建一个名为 screen.css 的文件：

```
body {
    background-color: #eee;
}

.bookContainer {
    outline: 1px solid #aaa;
    width: 350px;
    height: 130px;
    background-color: #fff;
    float: left;
    margin: 5px;
}

.bookContainer img {
    float: left;
    margin: 10px;
}

.bookContainer ul {
    list-style-type: none;
    margin-bottom: 0;
}

.bookContainer button {
    float: right;
    margin: 10px;
}

#addBook label {
    width: 100px;
    margin-right: 10px;
    text-align: right;
    line-height: 25px;
}

#addBook label, #addBook input {
    display: block;
    margin-bottom: 10px;
    float: left;
}

#addBook label[for="title"], #addBook label[for="releaseDate"] {
    clear: both;
}

#addBook button {
    display: block;
    margin: 5px 20px 10px 10px;
    float: right;
```

```
    clear: both;
}

#addBook div {
    width: 550px;
}

#addBook div:after {
    content: "";
    display: block;
    height: 0;
    visibility: hidden;
    clear: both;
    font-size: 0;
    line-height: 0;
}
```

现在，就像在图 5-3 里看到的，效果好了一些。

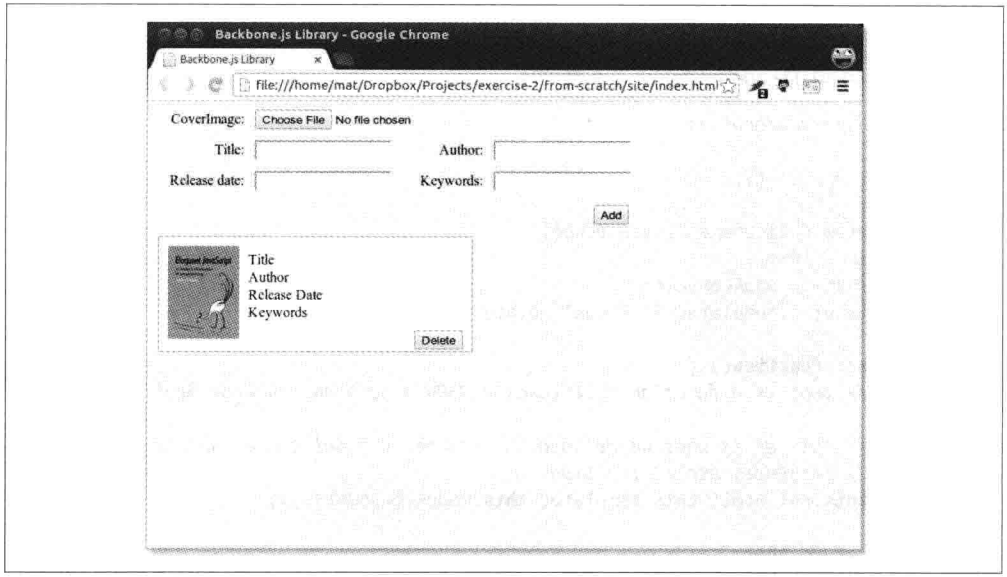


图 5-3　应用程序改进后的用户界面

这就是我们最终想要的结果，但不是多本图书显示的样子。要查看多本书放一起显示的样子，可以将 bookContainer 这个 div 多复制几次。至此，我们已经可以正式开始开发真正的应用程序了。

首先，需要为每本书创建一个模型，以及一个保存图书列表的集合。这些都比较简单，只需要为模型定义一些默认值即可：

```
// site/js/models/book.js

var app = app || {};

app.Book = Backbone.Model.extend({
    defaults: {
        coverImage: 'img/placeholder.png',
        title: 'No title',
        author: 'Unknown',
        releaseDate: 'Unknown',
        keywords: 'None'
    }
});
// site/js/collections/library.js

var app = app || {};

app.Library = Backbone.Collection.extend({
    model: app.Book
});
```

然后，要显示图书，我们需要一个视图：

```
// site/js/views/book.js

var app = app || {};

app.BookView = Backbone.View.extend({
    tagName: 'div',
    className: 'bookContainer',
    template: _.template( $('#bookTemplate').html() ),

    render: function() {
        // tmpl is a function that takes a JSON object and returns html

        // this.el is what we defined in tagName. use $el to get access
        // to jQuery html() function
        this.$el.html( this.template( this.model.toJSON() ));

        return this;
    }
});
```

我们还需要一个视图，用于显示图书列表：

```
// site/js/views/library.js

var app = app || {};

app.LibraryView = Backbone.View.extend({
    el: '#books',

    initialize: function( initialBooks ) {
```

```
        this.collection = new app.Library( initialBooks );
        this.render();
    },

    // render library by rendering each book in its collection
    render: function() {
        this.collection.each(function( item ) {
            this.renderBook( item );
        }, this );
    },

    // render a book by creating a BookView and appending the
    // element it renders to the library's element
    renderBook: function( item ) {
        var bookView = new app.BookView({
            model: item
        });
        this.$el.append( bookView.render().el );
    }
});
```

注意，在 initialize 函数中，我们接收一个数据数组，将其传给了 app.Library 构造器。我们使用该方式将一些样本数据填充至集合，以便可以看到程序是否正常运行。最后，我们定义一个带有样本数据的程序入口：

```
// site/js/app.js

var app = app || {};

$(function() {
    var books = [
        { title: 'JavaScript: The Good Parts', author: 'Douglas Crockford',
          releaseDate: '2008', keywords: 'JavaScript Programming' },
        { title: 'The Little Book on CoffeeScript', author: 'Alex MacCaw',
          releaseDate: '2012', keywords: 'CoffeeScript Programming' },
        { title: 'Scala for the Impatient', author: 'Cay S. Horstmann',
          releaseDate: '2012', keywords: 'Scala Programming' },
        { title: 'American Psycho', author: 'Bret Easton Ellis',
          releaseDate: '1991', keywords: 'Novel Splatter' },
        { title: 'Eloquent JavaScript', author: 'Marijn Haverbeke',
          releaseDate: '2011', keywords: 'JavaScript Programming' }
    ];

    new app.LibraryView( books );
});
```

我们的应用程序仅仅将样本数据传给 app.LibraryView 创建的一个新实例。由于 LibraryView 里的 initialize()构造函数调用视图的 render()方法，所以图书馆所有的图书都将显示出来。因为我们将整个入口程序作为回调传递给了 jQuery（代码里的 $别名），所以 DOM 加载完毕后，该回调函数就会执行。

在浏览器中访问 index.html，应该会显示类似图 5-4 这样的页面。

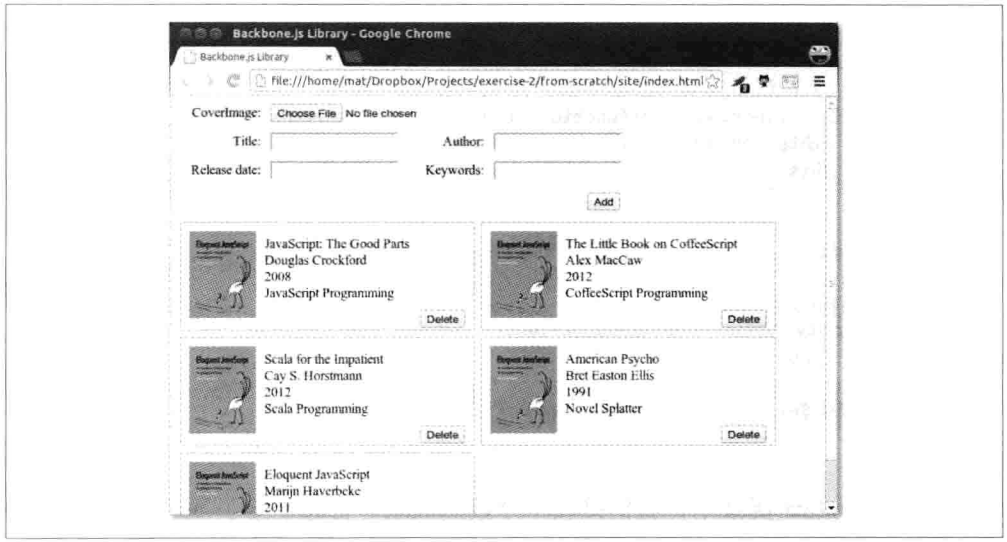


图 5-4　使用样本数据填充 Backbone 应用程序

虽然它并没有完成任何值得关注的事情，但这是一个完整的 Backbone 应用程序。

5.2　界面布局

现在，我们将在顶部还没用的表单上以及每本书的 Delete 按钮上，添加一些功能。

5.2.1　添加模型

用户单击 Add 按钮时，我们要收集表单中的数据，并用它创建一个新模型。我们需要在 LibraryView 里为单击事件添加一个事件处理程序：

```
events:{
    'click #add':'addBook'
},

addBook: function( e ) {
    e.preventDefault();

    var formData = {};

    $( '#addBook div' ).children( 'input' ).each( function( i, el ) {
        if( $( el ).val() != '' ){
            formData[ el.id ] = $( el ).val();
        }
    });
```

```
        this.collection.add( new app.Book( formData ) );
    },
```

我们收集表单里的所有 input 元素的值，并使用 jQuery 的 each 进行迭代。由于表单里的 id 元素和 Book 模型里的键值一样，我们可以简单地直接将数据存储在 formData 对象中。接着，用该数据创建新的 book，然后添加到集合里。我们忽略了无值的字段，以便可以使用默认值。

Backbone 给该事件处理函数传递了一个事件对象参数。这种情况对我们非常有利，因为我们不希望表单被提交以及页面重新加载。在 addbook 函数里添加一个 event 的 preventDefault 调用，将实现这一功能。

然后，添加新模型以后，只需要让视图重新渲染一下即可（见图 5-5）。要实现这一功能，我们在 LibraryView 的 initialize 函数里添加如下代码：

```
this.listenTo( this.collection, 'add', this.renderBook );
```

此时，应该可以让应用程序来个华丽变身了。

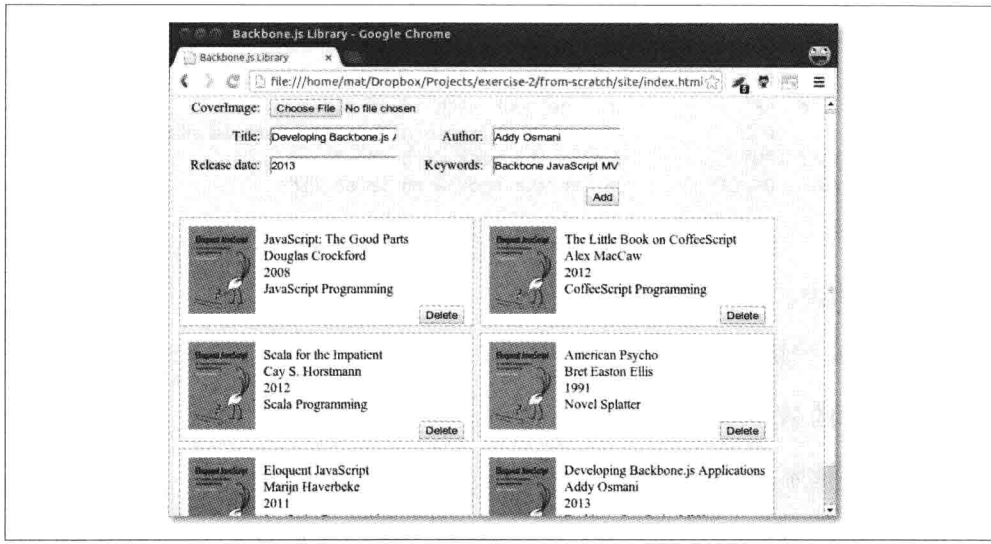


图 5-5　一旦新模型添加到集合中，视图就会渲染

大家可能会注意到，上面封面图标的 file 输入框不能用，这就留给读者作为一次练习吧。

5.2.2　删除模型

接下来，我们需要实现 Delete 按钮的功能。在 BookView 里设置该事件处理程序：

```
events: {
    'click .delete': 'deleteBook'
},

deleteBook: function() {
    // Delete model
    this.model.destroy();

    // Delete view
    this.remove();
},
```

至此，在图书馆程序里，我们应该可以添加和删除图书了。

5.3 创建后端系统

现在，我们要先绕一个迂回，设置一个带有 REST API（应用程序编程接口）的服务器。因为本书是关于 JavaScript 的，所以我们将使用 JavaScript，利用 Node.js 创建该服务器。如果你倾向于使用其他语言来创建 REST 服务器，需要遵守如下 API：

```
url             HTTP Method  Operation
/api/books      GET          Get an array of all books
/api/books/:id  GET          Get the book with id of :id
/api/books      POST         Add new book, return the book with id attribute added
/api/books/:id  PUT          Update the book with id of :id
/api/books/:id  DELETE       Delete the book with id of :id
```

本节大纲如下：

- 安装 Node.js、npm、MongoDB；
- 安装 Node 模块；
- 创建简单的 Web 服务器；
- 连接到数据库；
- 创建 REST API。

5.3.1 安装 Node.js、npm、MongoDB

从 Nodejs.org 下载并安装 Node.js，Node 包管理器（npm）也将被安装。

从 mongodb.org 下载并安装 MongoDB。官方网站有详细的安装指南（http://docs.mongodb.org/manual/installation/）。

5.3.2 安装 Node 模块

在项目的根目录下，创建一个名为 package.json 的文件。文件内容如下：

```
{
    "name": "backbone-library",
    "version": "0.0.1",
    "description": "A simple library application using Backbone",
    "dependencies": {
        "express": "~3.1.0",
        "path": "~0.4.9",
        "mongoose": "~3.5.5"
    }
}
```

除此之外，该文件告诉 npm 我们项目的依赖项。在命令行里，切换到项目的根目录，输入：

```
npm install
```

应该可以看到 npm 会获取我们在 package.json 文件中所列的依赖项，然后将其保存在名为 node_modules 的文件中。

项目文件夹的结构如下：

```
node_modules/
  .bin/
  express/
  mongoose/
  path/
site/
  css/
  img/
  js/
  index.html
package.json
```

5.3.3 创建简单的 Web 服务器

在项目的根目录中，创建一个名为 server.js 的文件，并且包含如下代码：

```
// Module dependencies.
var application_root = __dirname,
    express = require( 'express' ), //Web framework
    path = require( 'path' ), //Utilities for dealing with file paths
    mongoose = require( 'mongoose' ); //MongoDB integration

//Create server
var app = express();

// Configure server
```

```
app.configure( function() {
    //parses request body and populates request.body
    app.use( express.bodyParser() );

    //checks request.body for HTTP method overrides
    app.use( express.methodOverride() );

    //perform route lookup based on URL and HTTP method
    app.use( app.router );

    //Where to serve static content
    app.use( express.static( path.join( application_root, 'site') ) );

    //Show all errors in development
    app.use( express.errorHandler({ dumpExceptions: true, showStack: true }));
});
//Start server
var port = 4711;
app.listen( port, function() {
    console.log( 'Express server listening on port %d in %s mode',
    port, app.settings.env );
});
```

我们首先加载当前项目所需的模块：创建 HTTP 服务器用的 Express、处理文件用的 Path 和连接数据库用的 mongoose。然后，我们创建一个 Express 服务器，并使用匿名函数进行配置。这是一个非常标准的配置，我们的应用程序实际上并不需要 methodOverride 部分，这部分用于直接从表单上发送 PUT 和 DELETE HTTP 请求，这是因为通常表单只支持 GET 和 POST。最后，通过运行 listen 函数启动服务器。本例使用的端口号是 4711——可以在系统里使用任何可用的端口。我使用 4711，是因为其他任何程序好像都无法使用它。至此，就可以运行我们的第一个服务器了：

```
node server.js
```

如果在浏览器中打开 http://localhost:4711，看到的页面应该如图 5-6 所示。

我们现在在服务器上运行程序，而不是直接在文件上。做得好！现在我们可以开始定义服务器响应的路由（URL）了。这就是我们的 REST API。我们通过在 app 后面加上一个 HTTP 动词 get、put、post 和 delete 来定义路由，这些动词分别对应 create、read、update 和 delete 操作。让我们回到 server.js 来定义一个简单的路由：

```
// Routes
app.get( '/api', function( request, response ) {
    response.send( 'Library API is running' );
});
```

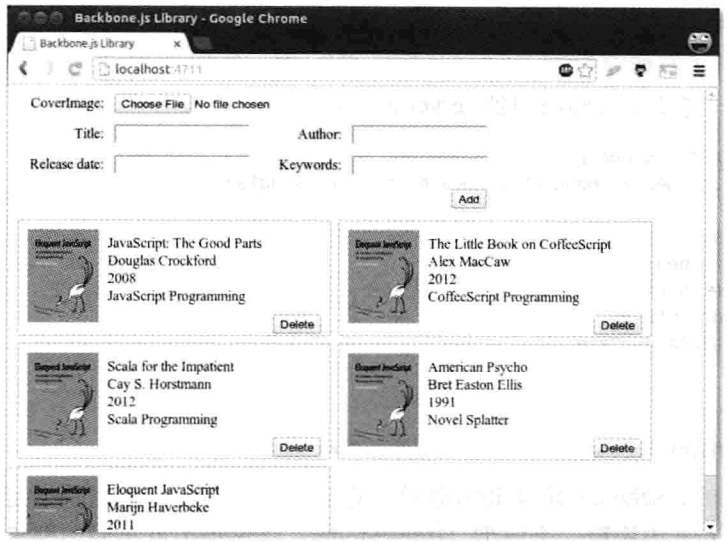


图 5-6　通过 Express 服务的 Backbone 应用程序

get 函数接收一个 URL 作为第一个参数，接收一个函数作为第二个参数。该函数调用时会接收 request 对象和 response 对象。现在，我们可以重新启动 Node，然后访问我们指定的 URL 了，页面显示如图 5-7 所示。

图 5-7　RESTful API 返回的初始响应

5.3.4 连接到数据库

现在，由于我们要将数据存储在 MongoDB 中，所以需要定义一个数据库架构（schema）。将如下代码添加到 server.js 中：

```
//Connect to database
mongoose.connect( 'mongodb://localhost/library_database' );

//Schemas
var Book = new mongoose.Schema({
    title: String,
    author: String,
    releaseDate: Date
});

//Models
var BookModel = mongoose.model( 'Book', Book );
```

就像你看到的，schema 定义非常简单。它可以更复杂，但我们就先定义成这样。还要从 Mongo 中提取一个模型（BookModel），该模型就是我们即将要用的。接下来，我们为 REST API 定义了一个 GET 操作，用于返回所有的图书：

```
//Get a list of all books
app.get( '/api/books', function( request, response ) {
    return BookModel.find( function( err, books ) {
        if( !err ) {
            return response.send( books );
        } else {
            return console.log( err );
        }
    });
});
```

BookModel 的 find 函数定义是这样的：function find（conditions, fields, options, callback），但由于我们想让函数返回所有图书，所以只需要 callback 参数。该 callback 调用时接收两个参数，一个是错误对象，一个是查找对象的数组。如果程序执行没有错误，就使用 response 的 send 函数将对象的数组返回到客户端；否则在控制台上记录该错误日志。

为了测试 API，我们需要在 JavaScript 控制台输入一点东西。重新启动 Node，在浏览器中访问 localhost:4711。打开 JavaScript 控制台。如果使用的是 Chrome，访问查看（View）→开发（Developer）→JavaScript 控制台（Console）。如果使用的是 Firefox，安装 Firebug，并访问查看（View）→Firebug。其他大多数浏览器通常也会有类似的控制台。在控制台中输入如下代码：

```
jQuery.get( '/api/books/', function( data, textStatus, jqXHR ) {
    console.log( 'Get response:' );
    console.dir( data );
```

```
    console.log( textStatus );
    console.dir( jqXHR );
});
```

按回车键，应该可以看到类似图 5-8 所示的结果。

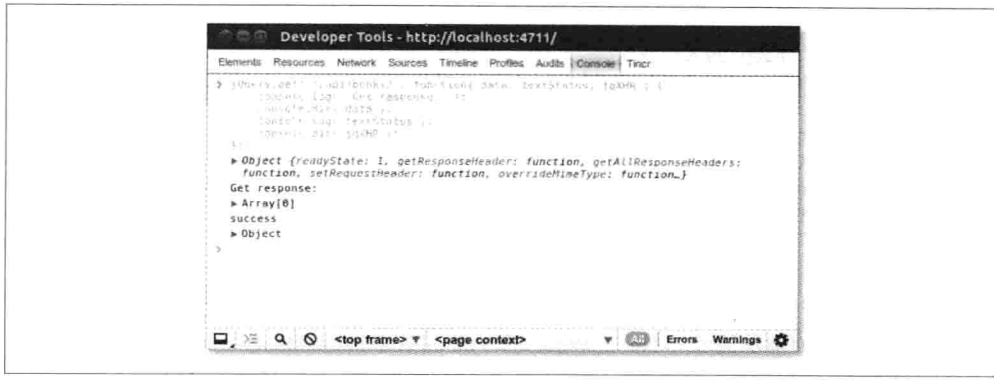


图 5-8　使用 jQuery 调用 REST API

这里，我们使用了 jQuery 来调用 REST API，因为它已经在页面上进行加载了。很显然，返回的数组是空的，因为我们还没有在数据库里保存任何东西。让我们继续在 server.js 里创建 POST 路由，用于添加新图书，代码如下：

```
//Insert a new book
app.post( '/api/books', function( request, response ) {
    var book = new BookModel({
        title: request.body.title,
        author: request.body.author,
        releaseDate: request.body.releaseDate
    });
    book.save( function( err ) {
        if( !err ) {
            return console.log( 'created' );
        } else {
            return console.log( err );
        }
    });
    return response.send( book );
});
```

首先，创建一个新的 BookModel，传递一个包含 title、author 和 releaseDate 属性的对象。该对象的数据是从 request.body 中收集的，这意味着，任何人在 API 上调用该操作，都需要提供一个包含 title、author 和 releaseDate 属性的 JSON 对象。实际上，因为我们没有对这些属性进行强制性设置，所以调用者可以省略部分或全部属性。

然后，在 BookModel 上调用 save 函数，传入一个和上例中 get 方法一样的 callback 回调。最后，返回已经保存的 BookModel。我们之所以返回 BookModel，而不是成

功或类似的字符串，是因为 BookModel 保存的时候会从 MondoDB 中获取一个_id 属性，该_id 属于使得客户端可以更新或删除指定的图书。再来试试，重新启动 Node，返回到控制台，并输入如下代码：

```
jQuery.post( '/api/books', {
    'title': 'JavaScript the good parts',
    'author': 'Douglas Crockford',
    'releaseDate': new Date( 2008, 4, 1 ).getTime()
}, function(data, textStatus, jqXHR) {
    console.log( 'Post response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});
```

然后输入：

```
jQuery.get( '/api/books/', function( data, textStatus, jqXHR ) {
    console.log( 'Get response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});
```

现在，应该可以从服务器得到包含一个元素的数组了。大家可能想知道这一行代码：

```
'releaseDate': new Date(2008, 4, 1).getTime()
```

MongoDB 希望时间是 UNIX 时间格式(从 UTC 时间 1970 年 1 月 1 日开始计算毫秒)，所以在 post 之前，我们必须将日期进行转换。不过，转换得到的对象包含一个 JavaScript 的 Date 对象。另外，也需要注意一下返回对象上的_id 属性，如图 5-9 所示。

图 5-9　返回 BookModel 的数据结构

让我们在 server.js 中继续创建一个 GET 请求，用于查询单本图书：

```
//Get a single book by id
app.get( '/api/books/:id', function( request, response ) {
    return BookModel.findById( request.params.id, function( err, book ) {
        if( !err ) {
            return response.send( book );
        } else {
            return console.log( err );
        }
    });
});
```

这里我们使用冒号表示法(:id)告诉 Express：路由的这部分是动态的。在 BookModel 上，我们还使用了 findById 函数，用于得到单个结果。重新启动 Node，使用如下 URL 地址，通过添加之前获得的 id，可以得到该书的单个信息：

```
jQuery.get( '/api/books/4f95a8cb1baa9b8a1b000006',
  function( data, textStatus, jqXHR ) {
    console.log( 'Get response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});
```

接下来，我们创建一个 PUT（更新）函数：

```
//Update a book
app.put( '/api/books/:id', function( request, response ) {
    console.log( 'Updating book ' + request.body.title );
    return BookModel.findById( request.params.id, function( err, book ) {
        book.title = request.body.title;
        book.author = request.body.author;
        book.releaseDate = request.body.releaseDate;

        return book.save( function( err ) {
            if( !err ) {
                console.log( 'book updated' );
            } else {
                console.log( err );
            }
            return response.send( book );
        });
    });
});
```

该示例比上个示例稍微长点，但也是非常简单的：通过 id 查找一本书，更新其属性并保存，然后将其发回到客户端。

为了验证这一点，我们需要使用更常用的 jQuery ajax 函数。同样，在这些示例中，我们需要将 id 属性换成与数据库中相匹配项的值：

```
jQuery.ajax({
    url: '/api/books/4f95a8cb1baa9b8a1b000006',
    type: 'PUT',
    data: {
        'title': 'JavaScript The good parts',
        'author': 'The Legendary Douglas Crockford',
        'releaseDate': new Date( 2008, 4, 1 ).getTime()
    },
    success: function( data, textStatus, jqXHR ) {
        console.log( 'Post response:' );
        console.dir( data );
        console.log( textStatus );
        console.dir( jqXHR );
    }
});
```

最后，创建删除（delete）路由：

```
//Delete a book
app.delete( '/api/books/:id', function( request, response ) {
    console.log( 'Deleting book with id: ' + request.params.id );
    return BookModel.findById( request.params.id, function( err, book ) {
        return book.remove( function( err ) {
            if( !err ) {
                console.log( 'Book removed' );
                return response.send( '' );
            } else {
                console.log( err );
            }
        });
    });
});
```

测试一下：

```
jQuery.ajax({
    url: '/api/books/4f95a5251baa9b8a1b000001',
    type: 'DELETE',
    success: function( data, textStatus, jqXHR ) {
        console.log( 'Post response:' );
        console.dir( data );
        console.log( textStatus );
        console.dir( jqXHR );
    }
});
```

至此，我们的 REST API 已经完成了——支持所有的 4 个 HTTP 动词。下一步是什么？对，直到现在，我们还遗漏了本书的关键字（keywords）部分。由于一本书可以有多个关键字，所以这会更复杂一点儿，我们不想将这些关键字作为一个单一字符串，而是要将其作为一个字符串数组。为此，我们需要另外一个架构（schema）。在 Book 架构代码前面添加一个 keywords 架构：

```
//Schemas
var Keywords = new mongoose.Schema({
    keyword: String
});
```

在现有的架构（schema）上添加一个子架构（subschema），我们使用方括号表示，代码如下：

```
var Book = new mongoose.Schema({
    title: String,
    author: String,
    releaseDate: Date,
    keywords: [ Keywords ]                          // NEW
});
```

还要更新 POST 和 PUT：

```
//Insert a new book
app.post( '/api/books', function( request, response ) {
    var book = new BookModel({
        title: request.body.title,
        author: request.body.author,
        releaseDate: request.body.releaseDate,
        keywords: request.body.keywords       // NEW
    });
    book.save( function( err ) {

        if( !err ) {
            return console.log( 'created' );
        } else {
            return console.log( err );
        }
    });
    return response.send( book );
});

//Update a book
app.put( '/api/books/:id', function( request, response ) {
    console.log( 'Updating book ' + request.body.title );
    return BookModel.findById( request.params.id, function( err, book ) {
        book.title = request.body.title;
        book.author = request.body.author;
        book.releaseDate = request.body.releaseDate;
        book.keywords = request.body.keywords; // NEW

        return book.save( function( err ) {
            if( !err ) {
                console.log( 'book updated' );
            } else {
                console.log( err );
            }
            return response.send( book );
        });
```

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

这就是我们所需的所有代码了。现在，可以在控制台上试一试了：

```
jQuery.post( '/api/books', {
    'title': 'Secrets of the JavaScript Ninja',
    'author': 'John Resig',
    'releaseDate': new Date( 2008, 3, 12 ).getTime(),
    'keywords':[
        { 'keyword': 'JavaScript' },
        { 'keyword': 'Reference' }
    ]
}, function( data, textStatus, jqXHR ) {
    console.log( 'Post response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});
```

至此，我们已经有一个功能齐全的 REST 服务器用于供前端调用了。

5.4 和服务器通信

本节我们将介绍通过 REST API 将 Backbone 应用程序连接到服务器上。

正如我在第 3 章所提到的，将 collection.url 设置为 API 的 URL，可以通过 collection.fetch()从服务器查询模型。让我们来更新集合实现该功能：

```
var app = app || {};

app.Library = Backbone.Collection.extend({
    model: app.Book,
    url: '/api/books'     // NEW
});
```

Backbone.sync 的默认实现结果，是假设 API 像下面这样：

```
url             HTTP Method  Operation
/api/books      GET          Get an array of all books
/api/books/:id  GET          Get the book with id of :id
/api/books      POST         Add new book, return book with id attribute added
/api/books/:id  PUT          Update the book with id of :id
/api/books/:id  DELETE       Delete the book with id of :id
```

为了让应用程序在页面加载的时候从服务器获取图书模型，我们需要更新 LibraryView。Backbone 文档建议在服务器端生成页面时就插入所有模型数据，而不是在页面加载后从客户端获取它们。因为本章是要给大家展示更完整的服务器交互，所以我们将忽略该推荐内容。找到 LibraryView 定义文件，更新 initialize 函数，

代码如下：

```
initialize: function() {
    this.collection = new app.Library();
    this.collection.fetch({reset: true}); // NEW
    this.render();

    this.listenTo( this.collection, 'add', this.renderBook );
    this.listenTo( this.collection, 'reset', this.render ); // NEW
},
```

现在，我们可以使用 this.collection.fetch()从数据库填充数据到我们的 Library 上了，initialize()函数不再传入样例数据作为参数，也不需要在 app.Library 的构造函数上传入任何东西了。现在可以从 site/js/app.js 里删除该样例数据了，减少到只剩一个创建 LibraryView 的语句：

```
// site/js/app.js

var app = app || {};

$(function() {
    new app.LibraryView();
});
```

我们还需要在 reset 事件上增加了一个监听器，因为模型数据是在页面渲染以后异步获取的。获取完毕后，由于 reset: true 选项要求，Backbone 触发了 reset 事件，我们的监听器就重新渲染了视图。如果重新加载页面，应该会看到服务器上存储的所有图书，都像图 5-10 一样显示在页面上。

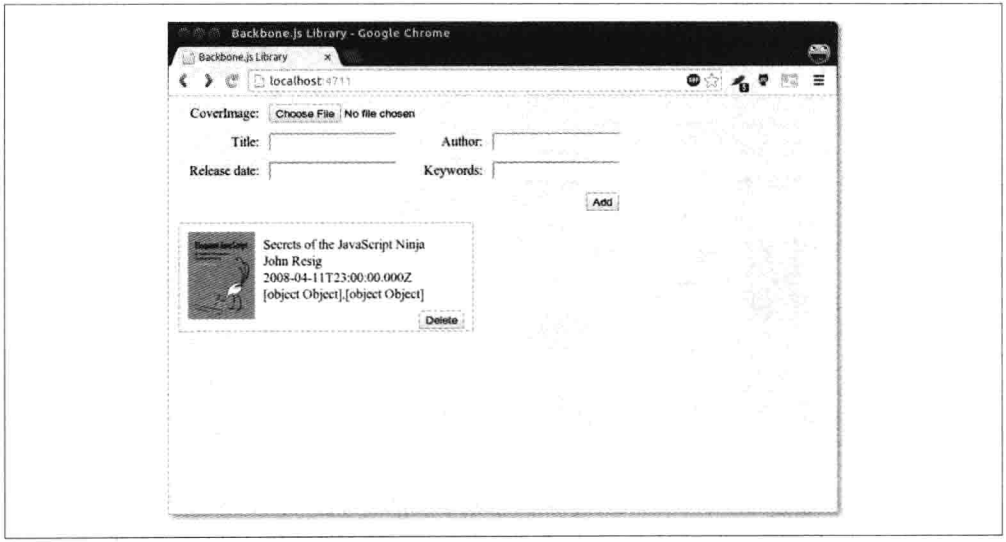


图 5-10　重新加载页面时显示服务器上保存的图书

就像你所看到的，日期和关键字看上去有点怪异。日期从服务器上获取后，转换为JavaScript Date 对象，应用到 underscore 模板时，使用 toString()函数进行显示。在JavaScript 中，toString()并没有很好地支持日期格式化，因此我们将使用 dateFormat（jQuery 插件）来修复这个问题。下载该插件（http://github.com/phstc/jquery-dateFormat），并将其放置到 site/js/lib 文件夹中。更新图书模板，以便日期像下面这样显示：

```
<li><%= $.format.date( new Date( releaseDate ), 'MMMM yyyy' ) %></li>
```

为插件添加一个 script 元素：

```
<script src="js/lib/jquery-dateFormat-1.0.js"></script>
```

现在页面上的日期显示得应该好点了。那么关键字怎么显示呢？因为查询的关键字在一个数组里，我们需要执行一些代码，将这些关键字生成由多个关键字空格隔开的一个字符串。要实现这一点，我们可以在模板标签里省略等号（=）字符，这将允许我们执行代码，而不显示任何信息：

```
<li><% _.each( keywords, function( keyobj ) {%>
    <%= keyobj.keyword %><% } ); %></li>
```

这里，我使用了 Underscore 的 each 函数遍历 keywords 数组，然后输出每个关键字。注意，我是使用 Underscore 微模板语法来显示关键字的。显示的时候，它会在每个关键字之间用空格隔开。

此时再重新加载页面，将会有视觉改进，正如在图 5-11 中所看到的。

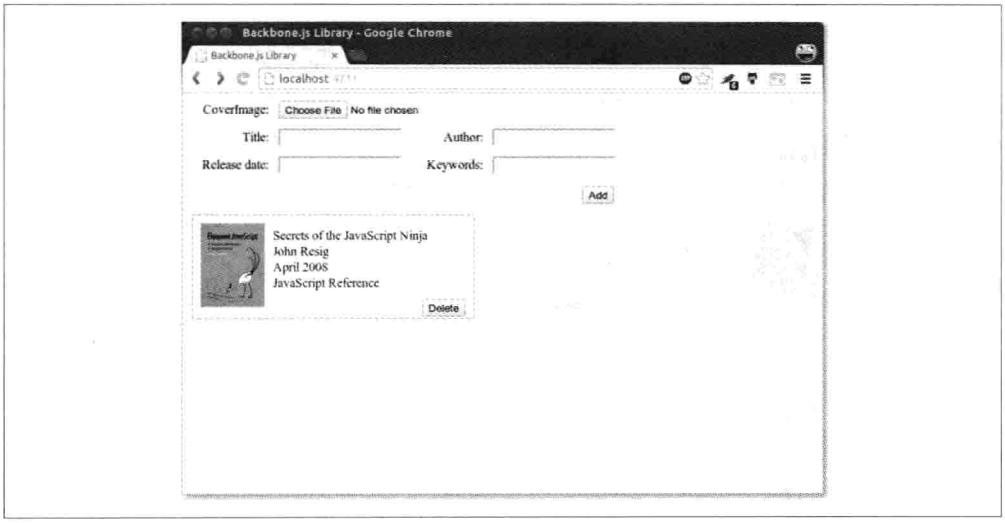


图 5-11　改进后的日期格式化

接着，删除一本书，然后重新加载页面，我们发现删除的书又回来了！这是为什么呢？这是因为从服务器获取图书模型数据（BookModel）时，它们有一个_id 属性（注意下画线），但 Backbone 期望获得的是 id 属性（没有下画线）。由于没有 id 属性，Backbone 将该模型作为新模型看待，删除一个新模型并不需要任何同步。

要解决这个问题，我们可以使用 Backbone.Model 的 parse 函数。parse 函数允许在数据传递给模型构造函数之前对服务器响应进行修改。在 BookModel 上添加一个 parse 函数：

```
parse: function( response ) {
    response.id = response._id;
    return response;
}
```

简单地将_id 的值复制到所需要的 id 属性上，重新加载页面，单击 Delete 按钮的时候，就会看到模型数据从服务器上真正地删除了。

让 Backbone 将_id 作为唯一标识符，另一个更简单的方法是将模型的 idAttribute 属性设置为_id。

如果现在使用表单添加一本新书，将会与 delete 操作发生同样的事情——添加的模型没有保存到服务器上，这是因为 Backbone.Collection.add 没有自动同步，但这很容易解决。在 views/library.js 的 LibraryView 里，将如下一行代码：

```
this.collection.add( new Book( formData ) );
```

修改为：

```
this.collection.create( formData );
```

现在新创建的图书就会保存到服务器上了。实际上，我们输入的日期可能不是一个日期，而服务器期望获得的是 UNIX 时间戳格式（从 1970 年 1 月 1 日后的毫秒数）。同样，我们输入的任何关键字也不会保存到服务器上，因为服务器期望获得的是一个带有 keyword 属性的数组。

先解决日期问题，我们并不是真的希望用户手动输入一个特定格式的日期，所以我们将使用 jQuery UI 的标准 datepicker。继续创建一个包含 datepicker 的自定义 jQuery UI（http://jqueryui.com/download/）。将 css 主题添加到 site/css/，并且将 JavaScript 文件添加到 site/js/lib 中。在 index . html 里进行链接：

```
<link rel="stylesheet" href="css/cupertino/jquery-ui-1.10.0.custom.css">
```

（cupertino 是我在下载 jQuery UI 时选择的风格名称。）

该 JavaScript 文件必须在 jQuery 加载以后再进行加载。

```
<script src="js/lib/jquery.min.js"></script>
<script src="js/lib/jquery-ui-1.10.0.custom.min.js"></script>
```

然后，在 app.js 中，在 releaseDate 字段上绑定 datepicker：

```
var app = app || {};

$(function() {
    $( '#releaseDate' ).datepicker();
    new app.LibraryView();
});
```

现在，单击 releaseDate 字段的时候，应该可以选择一个日期了，如图 5-12 所示。

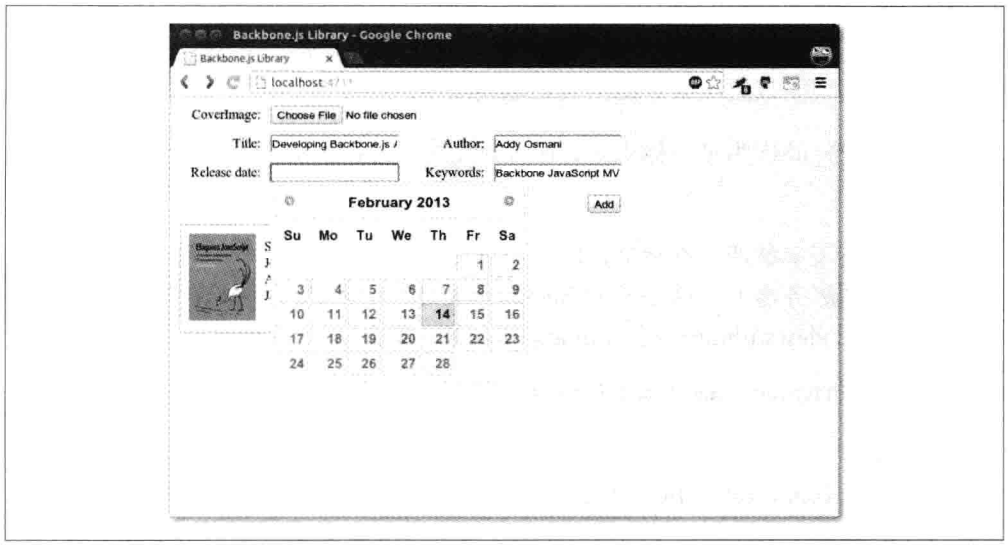


图 5-12　releaseDate 字段上的日期选择

最后，必须确保表单输入能够正确地转化为我们的存储格式。在 LibraryView 里修改 addBook 函数：

```
addBook: function( e ) {
    e.preventDefault();

    var formData = {};

    $( '#addBook div' ).children( 'input' ).each( function( i, el ) {
        if( $( el ).val() != '' )
        {
            if( el.id === 'keywords' ) {
                formData[ el.id ] = [];
                _.each( $( el ).val().split( ' ' ), function( keyword ) {
                    formData[ el.id ].push({ 'keyword': keyword });
```

```
                });
            } else if( el.id === 'releaseDate' ) {
                formData[ el.id ] = $( '#releaseDate' ).datepicker( 'getDate' ).getTime();
            } else {
                formData[ el.id ] = $( el ).val();
            }
        }
        // Clear input field value
        $( el ).val('');
    });

    this.collection.create( formData );
},
```

我们的修改在表单字段上增加了两个检查。首先，判断当前元素是否为 keywords 字段，如果是，使用空格将关键字进行分离，并为这些关键字对象创建一个数组。

然后，判断当前元素是否为 releaseDate 字段，如果是，调用 datePicker("getDate")返回一个 Date 对象，然后使用 Date 对象上的 getTime 函数获取以毫秒为单位的时间。

至此，应该可以添加带有发布日期和关键字的新书了，如图 5-13 所示。

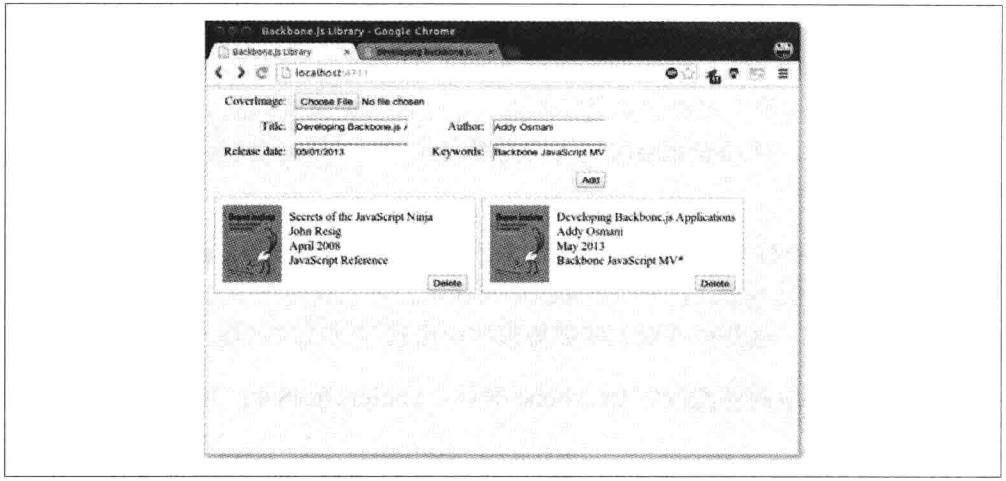


图 5-13　添加的新书显示发布日期和关键字

5.5　总结

在本章，我们通过使用 REST API 将应用程序绑定到服务器上，从而进行应用程序持久化。我们还了解了一些在数据序列化和反序列化中可能发生的问题以及解决方案。最后，我们还学习了 dateFormat 和 datepicker 这两个 jQuery 插件，以及如何在 Underscore 模板里做一些更高级的东西。读者可以从我的 GitHub 页面（http://bit.ly/12C56nH）上获得代码。

第 6 章

Backbone 扩展

Backbone 灵活、简单，并且功能强大。然而，你会发现，你正构建的程序所需要的复杂性远比开箱即用的功能复杂得多。另外还有一些它无法直接解决的问题，它的目标之一是实现极简。

以视图为例，其默认的 render 方法调用的时候，不会做任何事情，而且不会产生任何真实的结果，尽管大多数的实现都使用它来生成视图所管理的 HTML。同时，模型和集合没有用于处理嵌套层次结构的内置方式——如果需要该功能，需要自己编写它或使用插件。

在这些情况下，有许多现成的 Backbone 插件，可以为大规模的 Backbone 应用程序提供更高级的解决方案。在 Backbone 的 wiki 上，可以找到一个相当完整的可用插件以及框架列表。这些插件对大多数规模的应用程序的成功构建已经足够用了。

本章，我们将了解两种流行的 Backbone 插件：MarionetteJS 和 Thorax。

6.1 MarionetteJS（Backbone.Marionette）

作者：Derick Bailey、Addy Osmani

正如我们所看到的，Backbone 为 JavaScript 应用程序提供了一套非常好的构建模块。它为我们提供了核心结构，满足我们构建中小型应用、组织 jQuery DOM 事件、创建支持移动设备的单页面程序以及大型企业的需要。但 Backbone 并不是一个完整的框架，它是一套构建模块，将大部分的应用程序设计、架构和可扩展性工作留给开发人员，包括内存管理、视图管理等。

MarionetteJS 也称为 Backbone.Marionette。在 Backbone 自身提供的功能之上，它提供了很多优秀应用程序开发人员所需的特性。它是一个复合应用程序库，旨在简化大型应用程序的构建。它提供了一组常见的设计和实现模型，这些设计和模型是该库的创建者 Derick Bailey（http://lostechies.com/derickbailey/）和其他很多贡献者在构建 Backbone 应用时发现的。

Marionette 的主要优点包括：

- 允许我们使用模块化的、事件驱动的架构，扩大应用程序的规模；
- 提供合理的默认值，如，用于视图渲染的 Underscore 模板；
- 对于应用程序的特定需求，可以很轻松地修改；
- 使用专门的视图类型，减少视图引用；
- 在应用程序上创建一个模块化的架构，然后将模块附加到该架构上；
- 使用 region 和 layout，允许在运行时组合程序的 UI；
- 在视觉区域内，提供嵌套的视图和布局；
- 包括内置的内存管理，以及视图、区域、布局中无用东西的消除；
- 使用 EventBinder 提供了内置的事件清理功能；
- 使用 EventAggregator 合并事件驱动的架构；
- 提供一个灵活、即需即用的架构，让我们能够根据需要进行选择。

Marionette 也遵循了 Backbone 的构建思想，它提供一套组件，这些组件可以单独使用，也可以一起使用，以便为开发人员带来卓越的优势。但它比 Backbone 的组件结构更进了一步，并且提供了一个包含众多组件和构件块的应用程序层。

Marionette 的组件能够提供各种各样的特性，但它们会组合起来，来创建复合应用程序层，这样既能减少 boilerplate 代码，又能提供一个迫切需要的应用程序结构。其核心组件包括：各种特定的视图类型，将 boilerplate 排除在常见的 Backbone.Model 和 Backbone.Collection 场景的渲染之外；一个应用程序对象和模块架构，扩展子应用程序、特性和文件的应用程序；命令模式、事件聚合器和请求/响应机制的整合；以及更多的对象类型，可以以无数种方式被扩展，用于创建特定需求的应用程序架构。

尽管 Marionette 提供了大量的构造函数，但不必因为想使用其中一部分而全部都使用。就 Backbone 本身而言，我们可以随时选择想要使用的那些特性，这使我们能够轻松使用其他 Backbone 框架和插件。这也意味着，使用 Marionette 时不需要做全部的迁移改造。

6.1.1 Boilerplate 渲染代码

思考一下通常使用 Backbone 和 Underscore 模板进行视图渲染的代码。一般需要一个模板进行渲染，它可以直接放在 DOM 里，并且需要用 JavaScript 定义一个视图，该视图使用模板，并将模型里的数据填充到里面。

```
<script type="text/html" id="my-view-template">
  <div class="row">
    <label>First Name:</label>
    <span><%= firstName %></span>
  </div>
  <div class="row">
    <label>Last Name:</label>
    <span><%= lastName %></span>
  </div>
  <div class="row">
    <label>Email:</label>
    <span><%= email %></span>
  </div>
</script>

var MyView = Backbone.View.extend({
  template: $('#my-view-template').html(),

  render: function(){

    // compile the Underscore.js template
    var compiledTemplate = _.template(this.template);

    // render the template with the model data
    var data = this.model.toJSON();
    var html = compiledTemplate(data);

    // populate the view with the rendered html
    this.$el.html(html);
  }
});
```

一旦这些都设置好了，就需要创建一个视图实例，并将模型传入到里面。然后，要显示该视图，可以将视图的 el 附加到 DOM 里。

```
var Derick = new Person({
  firstName: 'Derick',
  lastName: 'Bailey',
  email: 'derickbailey@example.com'
```

```
});

var myView = new MyView({
  model: Derick
})

myView.render();

$('#content').html(myView.el)
```

这是 Backbone 标准的定义、构建、渲染和显示视图的设置，也就是我们所说的 boilerplate code——代码在具有相同功能的每个项目和每个实现里都重复。它很快就会变得乏味和重复。

进入 Marionette 的 ItemView——一种用于减少视图定义的 boilerplate 的简单方式。

6.1.2 使用 Marionette.ItemView 减少 Boilerplate

所有 Marionette 的视图类型（Marionette.View 除外）都包含了一个内置的 render 方法，用于处理核心的渲染逻辑。我们可以利用这点改变 MyView 实例来继承它，而不是继承 Backbone.View。不必为视图提供自己的 render 方法，我们可以让 Marionette 来渲染它。我们仍然使用相同的 Underscore.js 模板和渲染机制，但其实现却隐藏在幕后。因此，我们可以减少该视图所需的代码数量。

```
var MyView = Marionette.ItemView.extend({
  template: '#my-view-template'
});
```

就是这样——它可以让我们得到与前面例子的实现一模一样的效果。仅仅是将 Backbone.View.extend 替换成 Marionette.ItemView.extend，然后去除 render 方法。我们仍然可以使用模型创建视图实例，然后在视图实例上调用 render 方法，和之前一样的方式将视图显示在 DOM 里。但是视图定义已经减少到只配置模板的一行代码了。

6.1.3 内存管理

除了为定义视图减少代码以外，Marionette 还在所有的视图中包含一些高级内存管理功能，让清理视图实例和事件处理程序的工作变得简单。

思考一下如下的视图实现：

```
var ZombieView = Backbone.View.extend({
  template: '#my-view-template',
  initialize: function(){

    // bind the model change to rerender this view
```

```
    this.model.on('change', this.render, this);

  },

  render: function(){

    // This alert is going to demonstrate a problem
    alert('We`re rendering the view');

  }
});
```

如果我们为该视图创建两个实例，并且使用相同的变量名，接着改变模型中的一个值，那么会弹出多少次警告框？

```
var Person = Backbone.Model.extend({
  defaults: {
    "firstName": "Jeremy",
    "lastName": "Ashkenas",
    "email":    "jeremy@example.com"
  }
});

var Derick = new Person({
  firstName: 'Derick',
  lastName: 'Bailey',
  email: 'derick@example.com'
});


// create the first view instance
var zombieView = new ZombieView({
  model: Derick
});

// create a second view instance, reusing
// the same variable name to store it
zombieView = new ZombieView({
  model: Derick
});

Derick.set('email', 'derickbailey@example.com');
```

因为在这两个实例上，我们使用了同一个 zombieView 变量，第二个对象创建的时候，第一个视图的实例将会立即脱离作用域。这会让 JavaScript 垃圾回收器来清理，这意味着第一个视图实例不再存活，并且不再响应模型的 change 事件。

但是当我们运行这段代码时，最终的警告框却出现两次！

该问题是绑定视图的 initialize 方法里的模型事件导致的。每当我们将 this.render 作

为回调方法传递给模型的 on 事件绑定时，模型本身和视图实例之间会产生一个直接的引用。因为现在模型是持有视图实例的引用，将新的视图实例赋值给 zombieView 变量时，不会导致原有视图离开作用域。该模型依然会有该视图的引用，因此，该视图仍在作用域内。

鉴于原来的视图仍然在作用域内，第二个视图实例也在作用域内，所以在数据库上修改数据，将导致两个实例都会响应。

虽然修复这个很简单，只需在视图完成工作需要关闭时调用 stopListening 即可。为此，给视图添加一个 close 方法。

```
var ZombieView = Backbone.View.extend({
  template: '#my-view-template',

  initialize: function(){
    // bind the model change to rerender this view
    this.listenTo(this.model, 'change', this.render);
  },

  close: function(){
    // unbind the events that this view is listening to
    this.stopListening();
  },

  render: function(){

    // This alert is going to demonstrate a problem
    alert('We`re rendering the view');

  }
});
```

然后，不再需要第一个实例的时候，调用其 close 方法，此时只会有一个视图实例存活。更多关于 listenTo 和 stopListening 函数的信息，请查看第 3 章和 Derick 的文章“Managing Events As Relationships, Not Just References”（http://lostechies.com/derickbailey/2013/02/06/managing-events-as-relationships-not-just-references/）。

```
var Jeremy = new Person({
  firstName: 'Jeremy',
  lastName: 'Ashkenas',
  email: 'jeremy@example.com'
});

// create the first view instance
var zombieView = new ZombieView({
  model: Person
})
```

```
zombieView.close(); // double-tap the zombie

// create a second view instance, reusing
// the same variable name to store it
zombieView = new ZombieView({
  model: Person
})

Person.set('email', 'jeremyashkenas@example.com');
```

现在，上述代码运行的时候只会弹出一次警告框。

与其手动删除这些事件处理程序，不如让 Marionette 帮我们实现这一点。

```
var ZombieView = Marionette.ItemView.extend({
  template: '#my-view-template',

  initialize: function(){

    // bind the model change to rerender this view
    this.listenTo(this.model, 'change', this.render);

  },

  render: function(){

    // This alert is going to demonstrate a problem
    alert('We`re rendering the view');

  }
});
```

注意，本例我们使用了 listenTo 方法。该方法来自于 Backbone.Events，可以在所有混入 Backbone.Events 的对象上使用——包括绝大多数 Marionette 对象。listenTo 方法类似于 on 方法，唯一的区别在于 on 方法接收一个对象作为第一个参数用于触发事件。

Marionette 的视图还提供了一个 close 事件，在该事件里，使用 listenTo 的事件绑定都会被自动删除。这意味着，我们不需要再直接定义一个 close 方法，而且我们知道，如果使用 listenTo 方法，事件就会被删除，并且视图也不再会变成僵尸。

但是，在真实的应用程序里，我们如何在视图上自动调用 close 方法？何时何地去调用？进入 Marionette.Region 部分——管理单个视图生命周期的对象。

6.1.4 区域管理

视图创建以后，通常需要将其放在 DOM 里以便进行显示。通常，我们使用 jQuery 选择器，然后设置结果对象的 html()来实现：

```
var Joe = new Person({
  firstName: 'Joe',
  lastName: 'Bob',
  email: 'joebob@example.com'
});

var myView = new MyView({
  model: Joe
})

myView.render();

// show the view in the DOM
$('#content').html(myView.el)
```

这种方式，也是 boilerplate code。我们不必手动调用 render 和选择 DOM 元素来显示视图。此外，这段代码并不适合关闭之前的任何视图实例，这些实例可能已经附加到我们要填充的 DOM 元素上，并且我们已经看到了僵尸视图的危险性了。

要解决这些问题，Marionette 提供了一个 Region 对象——管理单个视图生命周期的对象，这些视图显示在特定的 DOM 元素里。

```
// create a region instance, telling it which DOM element to manage
var myRegion = new Marionette.Region({
  el: '#content'
});

// show a view in the region
var view1 = new MyView({ /* ... */ });
myRegion.show(view1);

// somewhere else in the code,
// show a different view
var view2 = new MyView({ /* ... */ });
myRegion.show(view2);
```

这里有几个注意事项。首先，在 region 实例里，我们通过指定 el 来告诉 region 哪个 DOM 元素需要管理。其次，在视图上不再调用 render 方法。最后，我们也不在视图上调用 close 方法，虽然它已经被调用了。

当使用一个区域（Region）来管理视图的生命周期，并将其显示在 DOM 中时，该区域对象本身会处理这些问题。当我们向 region 对象的 show 方法传递一个视图实例时，该 region 对象会调用视图上的 render 方法，然后把视图的 el 结果填充至 DOM 元素。

下一次，我们在调用 region 对象的 show 方法时，该对象会记得当前已经有一个视图在显示了，所以会在其视图上调用 close 方法，将其从 DOM 中删除，然后继续

对新传入的视图进行渲染并显示。

因为 region 帮我们处理调用 close 方法，并且我们在视图实例上使用 listenTo 事件绑定器，所以不需要再担心应用程序里的僵尸视图了。

Region 并不仅仅局限于 Marionette 视图，任何有效的 Backbone.View 都可以通过 Marionette.Region 来管理。如果该视图恰好有一个 close 方法，视图关闭的时候，该 close 方法就会被调用；如果没有，就会调用 Backbone.View 内置的 remove 方法。

6.1.5 Marionette Todo 应用程序

了解了 Marionette 的高级概念以后，现在我们来重构第一个练习中所创建的 Todo 应用程序。大家可以在 Derick 的 TodoMVC 项目的 fork 上找到该应用程序的完整代码。

我们最终实现的效果和功能与原有应用程序是一致的，如图 6-1 所示。

图 6-1 正在构建中的 Marionette Todo 应用程序

首先，定义一个应用程序对象表示基本的 TodoMVC 应用程序。代码将包含初始化代码以及定义应用程序的默认布局区域。

1. TodoMVC.js

```
var TodoMVC = new Marionette.Application();

TodoMVC.addRegions({
  header : '#header',
  main   : '#main',
  footer : '#footer'
});

TodoMVC.on('initialize:after', function(){
  Backbone.history.start();
});
```

区域（Region）用于管理显示在特定元素内的内容，TodoMVC 对象上的 addRegions 方法只是创建区域对象的一个快捷方式。我们为每个要管理的区域提供一个 jQuery 选择器（如，#header、#main 和#footer），然后告诉该区域对象，在该区域内显示各种各样的 Backbone 视图。

初始化应用程序对象以后，通过调用 Backbone.history.start()将其路由到初始的 URL 地址上。

接下来定义我们的布局（layouts）。一个 layout 是一种直接继承 Marionette.ItemView 的特殊类型的视图，这意味着其目的是渲染一个单独的模板，并且该模板可能有（也可能没有）相关联的模型（或 item）。

布局和 ItemView 之间的一个主要不同点是：布局包含了 region 区域。定义布局时，我们不仅提供了模板，也提供了模板所包含的 region 区域。布局渲染以后，使用之前定义的 region，我们在布局内可以显示其他的视图。

在 TodoMVC 布局模块中，我们为如下部分定义布局：

- Header：在这里，我们可以创建新的 todo 项。
- Footer：在这里，我们可以统计有多少 todo 项还未完成或已经完成。

这里使用了之前定义在 AppView 和 TodoView 里的一些视图逻辑。

注意，Marionette 模块（例如下面的代码）提供了一个简单的模块系统，用于在 Marionette 里创建私有属性和封装属性。当然，这些不是必须使用的，但在 6.1.7 小节里，我们将使用 RequireJS + AMD（异步模块定义）提供另外一种实现。

2. TodoMVC.Layout.js

```
TodoMVC.module('Layout', function(Layout, App, Backbone, Marionette, $, _){

  // Layout Header View
  // ------------------

  Layout.Header = Marionette.ItemView.extend({
    template : '#template-header',

    // UI bindings create cached attributes that
    // point to jQuery selected objects
    ui : {
      input : '#new-todo'
    },

    events : {
```

```
    'keypress #new-todo':   'onInputKeypress'
  },

  onInputKeypress : function(evt) {
    var ENTER_KEY = 13;
    var todoText = this.ui.input.val().trim();

    if ( evt.which === ENTER_KEY && todoText ) {
      this.collection.create({
        title : todoText
      });
      this.ui.input.val('');
    }
  }
});

// Layout Footer View
// ------------------


Layout.Footer = Marionette.Layout.extend({
  template : '#template-footer',

  // UI bindings create cached attributes that
  // point to jQuery selected objects
  ui : {
    count   : '#todo-count strong',
    filters : '#filters a'
  },

  events : {
    'click #clear-completed' : 'onClearClick'
  },

  initialize : function() {
    this.listenTo(App.vent, 'todoList:filter', this.updateFilterSelection);
    this.listenTo(this.collection, 'all', this.updateCount);
  },

  onRender : function() {
    this.updateCount();
  },

  updateCount : function() {
    var count = this.collection.getActive().length;
    this.ui.count.html(count);

    if (count === 0) {
      this.$el.parent().hide();
    } else {
      this.$el.parent().show();
    }
  },
```

```
    updateFilterSelection : function(filter) {
      this.ui.filters
        .removeClass('selected')
        .filter('[href="#' + filter + '"]')
        .addClass('selected');
    },

    onClearClick : function() {
      var completed = this.collection.getCompleted();
      completed.forEach(function destroy(todo) {
        todo.destroy();
      });
    }
  });

});
```

接下来，我们来解决应用程序路由和工作流问题，例如，控制页面中布局（layout）的显示或隐藏。

回想一下，Backbone 是如何在路由里触发方法的，如下所示的代码是我们在第一个练习中的初始路由代码：

```
var Workspace = Backbone.Router.extend({
  routes:{
    '*filter': 'setFilter'
  },

  setFilter: function( param ) {
    // Set the current filter to be used
    if (param){ param = param.trim()
    }
    app.TodoFilter = param ||";

    // Trigger a collection filter event, causing hiding/unhiding
    // of Todo view items
    app.Todos.trigger('filter');
  }
});
```

Marionette 使用 AppRouter 的概念来简化路由，这减少了用于处理路由事件的 boilerplate，并且允许配置路由，以便在对象上直接调用方法。通过替换原有路由里的“'*filter': 'setFilter'”，我们使用 appRoutes 来配置 AppRouter，并在控制器（controller）上调用一个方法。

TodoList 控制器在下面的代码中也有，它用于处理最初在 AppView 和 TodoView 里定义的一些剩余的显示逻辑，即使已经使用了一些非常可读的布局。

3. TodoMVC.TodoList.js

```
TodoMVC.module('TodoList', function(TodoList, App, Backbone, Marionette, $, _){

  // TodoList Router
  // ---------------
  //
  // Handle routes to show the active versus complete todo items

  TodoList.Router = Marionette.AppRouter.extend({
    appRoutes : {
      '*filter': 'filterItems'
    }
  });

  // TodoList Controller (Mediator)
  // ------------------------------
  //
  // Control the workflow and logic that exists at the application
  // level, above the implementation detail of views and models

  TodoList.Controller = function(){
    this.todoList = new App.Todos.TodoList();
  };

  _.extend(TodoList.Controller.prototype, {

    // Start the app by showing the appropriate views
    // and fetching the list of todo items, if there are any
    start: function(){
      this.showHeader(this.todoList);
      this.showFooter(this.todoList);
      this.showTodoList(this.todoList);

      this.todoList.fetch();
    },
    showHeader: function(todoList){
      var header = new App.Layout.Header({
        collection: todoList
      });
      App.header.show(header);
    },

    showFooter: function(todoList){
      var footer = new App.Layout.Footer({
        collection: todoList
      });
      App.footer.show(footer);
    },

    showTodoList: function(todoList){
      App.main.show(new TodoList.Views.ListView({
        collection : todoList
```

```
      }));
    },

    // Set the filter to show complete or all items
    filterItems: function(filter){
      App.vent.trigger('todoList:filter', filter.trim() || '');
    }
  });

  // TodoList Initializer
  // --------------------
  //
  // Get the TodoList up and running by initializing the mediator
  // when the application is started, pulling in all of the
  // existing todo items and displaying them.

  TodoList.addInitializer(function(){

    var controller = new TodoList.Controller();
    new TodoList.Router({
      controller: controller
    });

    controller.start();

  });

});
```

4. Controllers

请注意，在本例的特定应用程序里，并没有给控制器添加大量的整体工作流程。通常来说，Marionette 的路由原则是，在应用程序实现之后进行定义。在很多情况下，我们看到开发人员滥用 Backbone 的路由系统，他们将其作为整个应用程序工作流和逻辑的唯一控制器。

这将不可避免地导致将所有可能的代码组合在一起放到路由方法里——视图创建、模型加载、协调应用程序的不同部分等。Derick 等开发人员认为这违反了单一职责原则（http://bit.ly/15y3IpT）或 SRP 以及关注点分离。

Backbone 的路由（router）和历史（history）已经可以处理浏览器的特定行为了——“前进”和“后退”按钮。Marionette 的观点是：它应该仅限于此，让别的地方的导航去执行这些代码。这将使我们无论是在有还是没有路由的情况下都可以使用应用程序。我们可以通过按钮单击、应用程序事件处理程序或者路由来调用控制器的 show 方法，并且无论我们怎么调用该方法，最终都会以相同的应用程序状态结束。

Derick 详细描写了他对这个主题的见解，大家可以在他的博客（LosTechies）上阅读更多详细内容：

- **The Responsibilities Of The Various Pieces Of Backbone.js**（http://lostechies.com/derickbailey/2011/12/27/the-responsibilities-of-the-various-pieces-of-backbone-js/）；
- **Reducing Backbone Routers To Nothing More Than Configuration**（http://loste chies.com/derickbailey/2012/01/02/reducing-backbone-routers-to-nothing-more-than-configuration/）；
- **3 Stages Of A Backbone Application's Startup**（http://lostechies.com/ derickbailey/2012/02/06/3-stages-of-a-backbone-applications-startup/）。

5. 组合视图

下一个任务是为 TodoMVC 应用程序中的单独 todo 项和列表定义视图，为此，我们利用 Marionette 的组合视图。CompositeView 的理念是，它代表了一个主干和分支（或节点）的可视化复合或层级结构。

想象一下，这些视图是一个父子关系模型的层级，默认情况下是递归操作。同样的组合视图类型将用于渲染其所处理集合中的单个 todo 项。对于非递归层次结构，我们通过定义一个 itemView 属性，重载该 todo 项的视图。

todo 列表中的单个待办项视图，我们将其定义为 ItemView；其次，列表视图是一个组合视图，在该视图里，我们重载了 itemView 设置，并告知在集合中为每个单独的待办项使用 ItemView。

6. TodoMVC.TodoList.Views.js

```
TodoMVC.module('TodoList.Views', function
              (Views, App, Backbone, Marionette, $, _){

  // Todo List Item View
  // -------------------
  //
  // Display an individual todo item, and respond to changes
  // that are made to the item, including marking completed.

  Views.ItemView = Marionette.ItemView.extend({
      tagName : 'li',
      template : '#template-todoItemView',

      ui : {
```

```
    edit : '.edit'
  },

  events : {
    'click .destroy' : 'destroy',
    'dblclick label' : 'onEditClick',
    'keypress .edit' : 'onEditKeypress',
    'click .toggle'  : 'toggle'
  },

  initialize : function() {
    this.listenTo(this.model, 'change', this.render);
  },

  onRender : function() {
    this.$el.removeClass('active completed');
    if (this.model.get('completed')) this.$el.addClass('completed');
    else this.$el.addClass('active');
  },

  destroy : function() {
    this.model.destroy();
  },

  toggle  : function() {
    this.model.toggle().save();
  },

  onEditClick : function() {
    this.$el.addClass('editing');
    this.ui.edit.focus();
  },

  onEditKeypress : function(evt) {
    var ENTER_KEY = 13;
    var todoText = this.ui.edit.val().trim();

    if ( evt.which === ENTER_KEY && todoText ) {
      this.model.set('title', todoText).save();
      this.$el.removeClass('editing');
    }
  }
});
// Item List View
// --------------
//
// Controls the rendering of the list of items, including the
// filtering of active versus completed items for display.

Views.ListView = Marionette.CompositeView.extend({
  template : '#template-todoListCompositeView',
  itemView : Views.ItemView,
  itemViewContainer : '#todo-list',
```

```
    ui : {
      toggle : '#toggle-all'
    },

    events : {
      'click #toggle-all' : 'onToggleAllClick'
    },

    initialize : function() {
      this.listenTo(this.collection, 'all', this.update);
    },

    onRender : function() {
      this.update();
    },

    update : function() {
      function reduceCompleted(left, right)
      { return left && right.get('completed'); }
      var allCompleted = this.collection.reduce(reduceCompleted,true);
      this.ui.toggle.prop('checked', allCompleted);

      if (this.collection.length === 0) {
        this.$el.parent().hide();
      } else {
        this.$el.parent().show();
      }
    },

    onToggleAllClick : function(evt) {
      var isChecked = evt.currentTarget.checked;
      this.collection.each(function(todo){
        todo.save({'completed': isChecked});
      });
    }
  });

  // Application Event Handlers
  // --------------------------
  //
  // Handler for filtering the list of items by showing and
  // hiding through the use of various CSS classes

  App.vent.on('todoList:filter',function(filter) {
    filter = filter || 'all';
    $('#todoapp').attr('class', 'filter-' + filter);
  });

});
```

在最后的代码块中，大家可能还会注意到一个使用 vent 的事件处理程序。这是一个事件聚合器，用于处理 TodoList 控制器里的 filterItem 触发器。

最后，定义表示 todo 项的模型和集合。这些在语义上和第一个练习中定义的模型和集合没有很大的区别，只是进行重写，以便更能适应 Derick 倾向的代码风格。

7. Todos.js

```
TodoMVC.module('Todos', function(Todos, App, Backbone, Marionette, $, _){

  // Todo Model
  // ----------

  Todos.Todo = Backbone.Model.extend({
    localStorage: new Backbone.LocalStorage('todos-backbone'),

    defaults: {
      title     : '',
      completed : false,
      created   : 0
    },

    initialize : function() {
      if (this.isNew()) this.set('created', Date.now());
    },

    toggle  : function() {
      return this.set('completed', !this.isCompleted());
    },

    isCompleted: function() {
      return this.get('completed');
    }
  });

  // Todo Collection
  // ---------------

  Todos.TodoList = Backbone.Collection.extend({
    model: Todos.Todo,
    localStorage: new Backbone.LocalStorage('todos-backbone'),

    getCompleted: function() {
      return this.filter(this._isCompleted);
    },

    getActive: function() {
      return this.reject(this._isCompleted);
    },

    comparator: function( todo ) {
      return todo.get('created');
    },
```

```
    _isCompleted: function(todo){
      return todo.isCompleted();
    }
  });

});
```

调用主应用程序对象上的 start 以后，我们终于搞定 index 文件里的所有内容了。初始化代码如下：

```
$(function(){
  // Start the TodoMVC app (defined in js/TodoMVC.js)
  TodoMVC.start();
});
```

就这么多东西了！

6.1.6 Todo 应用程序的 Marionette 实现更具可维护性吗？

Derick 认为可维护性的实质是模块化，通过使用模块避免关注点混合在一起，从而分离责任（单一职责原则和关注点分离）。然而，为了提取、抽象或把概念分成其最简单的部分，很难抽象成单独的模块。

而 SRP 告诉我们的完全相反——我们需要了解事物变化的上下文。在该系统中，哪些部分总是要在一起改变？哪些部分可以单独改变？不了解这个，我们就无法知道哪些部分应该拆分成单独的组件和模块，哪些部分应该放在同一个模块或对象中。

Derick 组织应用程序分解成多个模块是通过在每个层次上进行概念分解的。一个高层次的模块是一个高层次的关注点——职责的聚合。每个职责分解成一个富有表现力的 API 集，这个 API 集是通过低层次的模块实现的（这被称为依赖倒置原则）。这些模块通过一个中介者进行协调，通常是指模块里的一个控制器。

Derick 的文件组织方式也直接影响了可维护性。他撰写了一些关于保持理智应用程序文件夹结构的重要性的博文，推荐阅读：

- **JavaScript File & Folder Structures: Just Pick One**（http://lostechies.com/derickbailey/2012/02/02/javascript-file-folder-structures-just-pick-one/）；
- **How to organize and structure the files and folders of HiloJS**（http://hilojs.codeplex.com/discussions/362875#post869640）。

6.1.7 Marionette 与灵活性

Marionette 是一个灵活的框架，就像 Backbone 本身。它提供了各种各样的工具，在

Backbone 上帮助我们创建和组织一个应用程序架构，但就像 Backbone 本身，它不会为了使用其中一个而要求我们必须使用所有的模块。

通过比较 3 个为了进行比较而创建的不同 TodoMVC 实现，你会发现 Marionette 的灵活性和多功能性很容易理解：

简单版本，作者：Jarrod Overson

https://github.com/jsoverson/todomvc/tree/master/labs/architecture-examples/backbone_marionette

这个版本的 TodoMVC 展示了 Marionette 各种视图类型、应用程序对象和事件聚合器的使用。对象创建以后直接添加到全局命名空间，非常简单。这是一个很好的例子，可以帮助我们理解如何用 Marionette 来增强现有代码而无需重写与 Marionette 相关的所有东西。

RequireJS 版本，作者：Jarrod Overson

https://github.com/jsoverson/todomvc/tree/master/labs/dependency-examples/backbone_marionette_require

Marionette 和 RequireJS 一起使用有助于创建一个模块化的应用程序架构——可伸缩 JavaScript 应用程序中一个非常重要的概念。RequireJS 提供了一套有极大利用价值的强大的工具集，可以让 Marionette 变得更加灵活。

Marionette 模块化版本，作者：Derick Bailey

https://github.com/derickbailey/todomvc/tree/marionette/labs/architecture-examples/backbone_marionette/js

RequireJS 并不是创建模块化应用程序架构的唯一方式。对于那些希望在模块和名称空间中构建应用程序的人来说，Marionette 提供了一个内置的模块和命名空间结构。该示例应用程序实现了一个简单版本的 Todo 应用程序，并且使用一个将所有内容聚焦在一起的应用程序控制器（中介者/工作流对象），将其重写成一个名称空间化的应用程序架构。

在 Backbone 应用程序如何进行架构方面，Marionette 无疑也提供了自己的建议。基于这些建议，模块、视图类型、事件聚合器、应用程序对象等，可以用于创建更强大、更灵活的架构。

但是，如你所见，Marionette 也不完全是一个死板的、“我行我素”的框架。它提供了许多应用基础的元素，用于混入或匹配到其他架构风格，如 AMD 或命名空间化，或者我们可以通过减少渲染视图的样板代码来增强现有的项目。

这种灵活性让 Marionette 为你和你的项目创造了提供更大价值的机会，因为它允许你根据应用程序的需要来扩展 Marionette 的使用。

6.1.8 更多特性

即便我们对 ItemView 和 Region 对象都进行了研究，Marionette 的上述特性仅仅是冰山的一角。Marionette 还有更多的功能、更多的特性，以及更大的灵活性和可定制性，这些都可以用于各种对象。另外，Marionette 还提供了一堆组件，每个组件都有其自己的内置行为集、定制化和扩展点，甚至更多。

要了解更多关于 Marionette 的组件、提供的特性，以及如何使用它们的信息，请登录 http://marionettejs.com 查阅 Marionette 官方文档、wiki 链接、源代码链接、项目核心贡献者以及更多内容。

6.2 Thorax

作者：Ryan Eastridge、Addy Osmani

Backbone 的部分吸引力是它提供了程序结构，而通常又不死板，特别是当它涉及视图的时候。Thorax 做了一个武断的决定，它决定使用 Handlebars 作为模板解决方案。Marionette 里的一些模式，在 Thorax 中同样也有。Marionette 将大多数模式暴露成 JavaScript API，而在 Thorax 里则经常作为模板助手（template helper）。本节假设读者对 Handlebars 有一定的了解。

Ryan Eastridge 和 Kevin Decker 开发 Thorax 是为了创建沃尔玛的移动 Web 应用。本节的内容将限制于 Thorax 实现的的模板特性和模式，不管是否采用 Thorax，都可以在应用程序中利用这些特性。要了解更多关于 Thorax 的其他功能实现的信息并下载示例项目，请访问 Thorax 官方网站（http://thoraxjs.org/）。

6.2.1 Hello World

在 Backbone 里创建新视图时，传入的 options 和视图中已经存在的默认选项合并在一起，并且通过 this.options 访问，以便以后引用。

Thorax.View 和 Backbone.View 不同，它没有 options 对象。传递到构造函数的所有参数都会成为视图的属性，进而在模块中可用：

```
var view = new Thorax.View({
    greeting: 'Hello',
    template: Handlebars.compile('{{greeting}} World!')
});
view.appendTo('body');
```

本节的绝大多数例子都会指定一个 template 属性。在大型项目中（包括 Thorax 网站提供的示例项目）将会使用 name 属性代替，并且项目中会有一个同名的模板文件自动分配给该视图。

如果在视图上设置了 model，那么该 model 值上的属性在模板上也将可用：

```
var view = new Thorax.View({
    model: new Thorax.Model({key: 'value'}),
    template: Handlebars.compile('{{key}}')
});
```

6.2.2 嵌入子视图

视图助手允许我们在视图中嵌入其他的视图。子视图可以指定为该视图的属性：

```
var parent = new Thorax.View({
    child: new Thorax.View(...),
    template: Handlebars.compile('{{view child}}')
});
```

或者子视图在初始化的时候进行命名，可以传入任何我们想选择的名字。在本例中，子视图必须使用 extend 进行提前创建，并且设置一个 name 属性：

```
var ChildView = Thorax.View.extend({
    name: 'child',
    template: ...
});

var parent = new Thorax.View({
    template: Handlebars.compile('{{view "child" key="value"}}')
});
```

视图助手也可以当成模板块助手，在本例这种情况下，该模板块将赋值给子视图的 template 属性：

```
{{#view child}}
    child will have this block
    set as its template property
{{/view}}
```

与 Backbone.View 实例拥有 DOM（el）不同，Handlebars 是基于字符串的。因为我们是混合使用视图，嵌入的视图通过占位符机制进行工作，本例中是视图助手将助手传递过来的视图添加到 children 哈希里，然后注入到模板中的 HTML 占位符，例如：

```
<div data-view-placeholder-cid="view2"></div>
```

然后，一旦父视图呈现，在 DOM 中搜索所有创建的占位符，就使用子视图的 el 进行替换：

```
this.$el.find('[data-view-placeholder-cid]').forEach(function(el) {
    var cid = el.getAttribute('data-view-placeholder-cid'),
        view = this.children[cid];
    view.render();
    $(el).replaceWith(view.el);
}, this);
```

6.2.3 视图助手

Thorax 中最实用的构造函数之一是 Handlebars.registerViewHelper（不要和 Handlebars.registerHelper 混淆）。该方法将注册一个块助手，该块助手将创建并嵌入一个 HelperView 实例，该实例的 template 属性被设置为捕获的块。在模板中，HelperView 实例和普通子视图的实例是不同的，HelperView 实例的上下文是模板里父视图的上下文。和其他子视图一样，HelperView 实例会有一个关联到父视图的 parent 属性。Thorax 里的很多内置助手（包括集合助手）都是以这种方式创建的。

举一个简单的例子，每次在定义父视图上触发事件时，on 助手会重新渲染已经生成的 HelperView 实例：

```
Handlebars.registerViewHelper('on', function(eventName, helperView) {
    helperView.parent.on(eventName, function() {
        helperView.render();
    });
});
```

使用示例是：每次单击按钮的时候，用一个计数器记录。这个例子使用了 Thorax 的 button 助手，在按钮被单击的时候，该助手只是让按钮调用一个方法：

```
{{#on "incremented"}}{{i}}{/on}}
{{#button trigger="incremented"}}Add{{/button}}
```

对应的视图类如下：

```
new Thorax.View({
    events: {
        incremented: function() {
            ++this.i;
        }
    },
    initialize: function() {
        this.i = 0;
    },
    template: ...
});
```

6.2.4 集合助手

集合助手创建并嵌入一个 CollectionView 实例，为集合中的每个项创建一个视图，并且项在集合中被添加、删除或修改时，更新该视图。该助手最简单的用法如下：

```
{{#collection kittens}}
  <li>{{name}}</li>
{{/collection}}
```

对应的视图如下：

```
new Thorax.View({
  kittens: new Thorax.Collection(...),
  template: ...
});
```

在每个项的视图创建时，本例中的模板块会赋值给其 template 属性，并且其上下文将成为给定模型的 attributes 属性。这个助手接受可选项，如任意的 HTML 属性，用于指定所包含的集合类型 tag 标签，或者以下列表中的任意一个：

- **item-template**：显示单个模型的模板。如果指定了一个模板块，它将成为 item-template。
- **item-view**：每个项的视图创建时所使用的视图类。
- **empty-template**：集合为空时要显示内容的模板。如果指定了 inverse/else 块，它将成为 empty-template。
- **empty-view**：集合为空时要显示的视图。

可选项和模板块可以一起组合使用。下面的例子将创建一个 KittenView 类，将其 template 属性设置为下面的模板块，用于集合中每一个 kitten 的展示：

```
{{#collection kittens item-view="KittenView" tag="ul"}}
  <li>{{name}}</li>
{{else}}
    <li>No kittens!</li>
  {{/collection}}
```

请注意，每个视图可以使用多个集合，并且集合可以进行嵌套。当模型中包含集合，集合中又包含模型时，这就很有用：

```
{{#collection kittens}}
  <h2>{{name}}</h2>
  <p>Kills:</p>
  {{#collection miceKilled tag="ul"}}
    <li>{{name}}</li>
  {{/collection}}
{{/collection}}
```

6.2.5 自定义 HTML Data 属性

Thorax 使用了大量的自定义 HTML Data 属性进行操作。一些属性在 Thorax 上下文中行得通，而有一些属性在 Backbone 项目里用于编写其他功能或者一般调试的时候非常有用。要在非 Thorax 项目里的视图上添加 HTML Data 属性，需要在视图基类上重载 setElement 方法：

```
MyApplication.View = Backbone.View.extend({
  setElement: function() {
      var response = Backbone.View.prototype.setElement.apply(this, arguments);
      this.name && this.$el.attr('data-view-name', this.name);
      this.$el.attr('data-view-cid', this.cid);
      this.collection && this.$el.attr('data-collection-cid',
      this.collection.cid);
      this.model && this.$el.attr('data-model-cid', this.model.cid);
      return response;
  }
});
```

为了让应用程序在复查代码的时候更一目了然，可以使用函数来扩展 jQuery / Zepto，查找与给定元素最接近的视图、模型或集合。为此，必须通过重载_configure 方法，在视图基类里保存每个已创建视图的引用：

```
MyApplication.View = Backbone.View.extend({
    _configure: function() {
        Backbone.View.prototype._configure.apply(this, arguments);
        Thorax._viewsIndexedByCid[this.cid] = this;
    },
    dispose: function() {
        Backbone.View.prototype.dispose.apply(this, arguments);
        delete Thorax._viewsIndexedByCid[this.cid];
    }
});
```

然后扩展 jQuery/Zepto：

```
$.fn.view = function() {
    var el = $(this).closest('[data-view-cid]');
    return el && Thorax._viewsIndexedByCid[el.attr('data-view-cid')];
};

$.fn.model = function(view) {
    var $this = $(this),
        modelElement = $this.closest('[data-model-cid]'),
        modelCid = modelElement && modelElement.attr('[data-model-cid]');
    if (modelCid) {
        var view = $this.view();
        return view && view.model;
    }
    return false;
};
```

特定 DOM 事件发生时，可以使用$(element).model()，而无需在应用程序中将引用随机保存至模型来进行查找。在 Thorax 里，结合集合助手在集合里为每个模型都生成一个视图类（带有一个 model 属性）特别有用。如下是一个示例模板：

```
{{#collection kittens tag="ul"}}
  <li>{{name}}</li>
{{/collection}}
```

对应的视图类如下：

```
Thorax.View.extend({
  events: {
    'click li': function(event) {
      var kitten = $(event.target).model();
      console.log('Clicked on ' + kitten.get('name'));
    }
  },
  kittens: new Thorax.Collection(...),
  template: ...
});
```

Backbone 应用程序中一个常见的反模式是为一个单独的视图类设置样式名称。考虑使用 data-view-name 属性作为 CSS 选择器，进行 CSS 样式保存，可以多次使用：

```
[data-view-name="child"] {

}
```

6.2.6 Thorax 资源

没有 Todo 应用程序，Backbone 教程就不会完整。可以找到 TodoMVC 的 Thorax 实现（http://todomvc.com/labs/architecture-examples/thorax/）的示例，除了使用单独的 Handlebars 模板组成的简单例子外：

```
{{#collection todos tag="ul"}}
  <li{{#if done}} class="done"{{/if}}>
    <input type="checkbox" name="done"{{#if done}} checked="checked"{{/if}}>
    <span>{{item}}</span>
  </li>
{{/collection}}
<form>
  <input type="text">
  <input type="submit" value="Add">
</form>
```

对应的 JavaScript 代码如下：

```
var todosView = Thorax.View({
    todos: new Thorax.Collection(),
    events: {
        'change input[type="checkbox"]': function(event) {
```

```
            var target = $(event.target);
            target.model().set({done: !!target.attr('checked')});
        },
        'submit form': function(event) {
            event.preventDefault();
            var input = this.$('input[type="text"]');
            this.todos.add({item: input.val()});
            input.val('');
        }
    },
    template: '...'
});
todosView.appendTo('body');
```

要想查看 Thorax 在大型可伸缩站点的实战，可以用 Android 或 iOS 设备访问 walmart.com。欲获得完整的资源列表，请访问 Thorax 官方网站（http://thoraxjs.org/）。

6.3 总结

Backbone 是构建现代客户端应用程序的热门选择，但一些项目需要更多开箱即用的决策。在 Backbone 开发方面，Thorax 提供了 Rails 风格的开发体验，解决了很多决策问题。Thorax 回应了“Backbone 项目应该是什么样子”、“应该使用什么样的目录结构”、“如何针对每个目标平台，将客户端应用程序构建成可部署的单元”这样的问题。Thorax 可能并不适合于所有人，但它却为寻求构建更复杂应用程序的开发人员提供了一些很不错的语法糖。

第 7 章

常见问题和解决方案

本章我们将回顾一些开发人员开始在相对重要的项目上使用 Backbone.js 时经常遇到的问题，以及目前可能的解决方案。

我们最常遇到的问题可能是如何利用视图做更多的事情。如果大家有兴趣探索如何使用嵌套视图，或者学习视图清理和继承，本章将会覆盖到。

7.1 使用嵌套视图

7.1.1 问题

在 Backbone.js 里，渲染和附加嵌套视图（或子视图）的最好方式是什么？

7.1.2 解决方案 1

由于页面是由嵌套元素组合而成的，并且 Backbone 的视图在页面中也对应着相应的元素，所以使用嵌套视图来管理元素的层级是一种很直观的方式。

组合视图的最好方式是简单使用如下代码：

```
this.$('.someContainer').append(innerView.el);
```

其仅依赖于 jQuery。我们可以像下面这样，在真实的示例中使用：

```
...
initialize : function () {
    //...
},
render : function () {

    this.$el.empty();
```

```
        this.innerView1 = new Subview({options});
        this.innerView2 = new Subview({options});

        this.$('.inner-view-container')
            .append(this.innerView1.el)
            .append(this.innerView2.el);
    }
```

7.1.3 解决方案 2

初学者有时可能也会尝试使用 setElement 来解决这个问题。不过要记住，使用这种方式极易自找麻烦。如果解决方案 1 可行，要避免使用这种方式：

```
    // Where we have previously defined a View, SubView
    // in a parent View we could do:

    ...
    initialize : function () {

        this.innerView1 = new Subview({options});
        this.innerView2 = new Subview({options});
    },

    render : function () {

        this.$el.html(this.template());

        this.innerView1.setElement('.some-element1').render();
        this.innerView2.setElement('.some-element2').render();
    }
```

这里我们在父视图的 initialize()方法里创建了子视图，并且在父视图的 render()方法里进行了渲染。子视图管理的元素已经存在于父视图的模板里了，View.setElement()方法用于将每个子视图相关的元素进行重新赋值。

setElement()改变了视图中的元素，其中包括重新委托事件处理程序，其事件处理程序从旧元素上移除，并绑定到新元素上。注意，setElement()返回的是视图，允许我们调用 render()。

这种方式可用，并且也有一些积极的方面：附加的时候，不需要担心 DOM 元素顺序的维护问题，视图是提前初始化的，并且 render()方法一次不需要承担太多的职责。

这种方式的缺点是不能设置子视图的 tagName 属性，并且需要重新委托事件。解决方案 1 并不受该问题的影响。

7.1.4 解决方案 3

另外一个可能的解决方案，可以这样编写：

```
var OuterView = Backbone.View.extend({
    initialize: function() {
        this.inner = new InnerView();
    },

    render: function() {
        this.$el.html(template); // or this.$el.empty() if you have no template
        this.$el.append(this.inner.$el);
        this.inner.render();
    }
});

var InnerView = Backbone.View.extend({
    render: function() {
        this.$el.html(template);
        this.delegateEvents();
    }
});
```

这种方式能解决一些特定的设计决策：

- 附加子元素的顺序。
- OuterView 中不包含 InnerView(s)要设置的 HTML 元素，这就意味着仍然可以在 InnerView 里设置 tagName。
- render()是在 InnerView 的元素放置到 DOM 以后调用的。如果在页面上，InnerView 的 render()方法调用是根据其他元素的大小进行自适应大小设置的话，这就非常有用处。这是一个常见的现象。

要注意，因为子视图里的元素已经被替换了，所以 InnerView 需要调用 View.delegateEvents()重新将事件处理程序绑定到新 DOM 上。

7.1.5 解决方案 4

这是一个更好的解决方案，很整洁，但有潜在的性能影响，代码如下：

```
var OuterView = Backbone.View.extend({
    initialize: function() {
        this.render();
    },
    render: function() {
        this.$el.html(template); // or this.$el.empty() if you have no template
        this.inner = new InnerView();
        this.$el.append(this.inner.$el);
```

```
    }
});

var InnerView = Backbone.View.extend({
    initialize: function() {
        this.render();
    },

    render: function() {
        this.$el.html(template);
    }
});
```

在一个模板里的特定位置，如果有多个视图需要嵌套，应该为子视图创建一个通过客户端 ID（cid）进行索引的哈希。在模板中，使用一个自定义的 HTML 属性 data-view-cid 为每个嵌入的视图创建占位符元素。模板一旦进行渲染，其输出附加到父视图的$el 上，每个占位符都能被查询，并且被替换成子视图的 el。

一个包含单个子视图的示例实现代码如下：

```
var OuterView = Backbone.View.extend({
    initialize: function() {
        this.children = {};
        this.child = new Backbone.View();
        this.children[this.child.cid] = this.child;
    },

    render: function() {
        this.$el.html('<div data-view-cid="' + this.child.cid + '"></div>');
        _.each(this.children, function(view, cid) {
            this.$('[data-view-cid="' + cid + '"]').replaceWith(view.el);
        }, this);
    }
};
```

在这里，cid 的包含是非常有用的，因为它让视图通过其实例（而不是属性）进行引用，从而将模型和视图进行分离。要求所有的视图都在其模型上对应一个属性是很正常的，但如果有递归视图或重复视图（常见情况），则不能简单地通过属性要求视图实现，除非指定额外的属性用于区分重复。使用 cids 解决了这个问题，因为它允许和视图有直接引用。

一般来说，大多数开发人员的选择是解决方案 1 或方案 4，因为：

- 大多数视图都已经在 render()方法里依赖于 DOM；
- OuterView 重新渲染时，视图不需要重新初始化，因为重新初始化有可能导致内存泄漏以及现有绑定上的问题。

Backbone 的扩展 Marionette 和 Thorax 为嵌套视图和集合渲染提供了逻辑，其集合中的每个条目都有一个相关联的视图。Marionette 在 JavaScript 中提供了 API，Thorax 是通过 Handlebars 模板助手提供 API 的。

感谢 Lukas 和 Ian Taylor 提供的这些技巧。

Lukas：http://stackoverflow.com/users/299189/lukas

Ian Taylor：http://stackoverflow.com/users/154765/ian-storm-taylor

7.2 在嵌套视图中管理模型

7.2.1 问题

在嵌套视图中，管理模型的最好方式是什么？

7.2.2 解决方案

在嵌套设置中，为了找到相关模型的属性，模型需要提前了解彼此，了解一些 Backbone 并没有进行隐式处理的东西。

其中一种方式是，确保每个子模型都有一个 parent 属性。通过采用这种方式，可以遍历嵌套并首先到达父模型，然后到达我们所知道的任何兄弟模型。因此，假设有模型 modelA、modelB、modelC：

```
// When initializing modelA, I would suggest setting a link to the parent
// model when doing this, like this:

ModelA = Backbone.Model.extend({

    initialize: function(){
        this.modelB = new modelB();
        this.modelB.parent = this;
        this.modelC = new modelC();
        this.modelC.parent = this;
    }
}
```

上述代码使我们能够通过 this.parent 在任意子模型上找到父模型。

至此，我们已经讨论了多个通过 Backbone 构建嵌套视图的方式。为了简单起见，我们假设要在如下代码 ViewA 的 initialize()方法里创建一个新的子视图 ViewB。ViewB 可以找到 ViewA 模型，并且可以在任意嵌套模型上进行事件监听。

查看代码里的行内注释，详细了解每一步是如何实现的：

```
// Define View A
ViewA = Backbone.View.extend({

    initialize: function(){
       // Create an instance of View B
       this.viewB = new ViewB();

       // Create a reference back to this (parent) view
       this.viewB.parentView = this;

       // Append ViewB to ViewA
       $(this.el).append(this.viewB.el);
    }
});

// Define View B
ViewB = Backbone.View.extend({

    //...,

    initialize: function(){
        // Listen for changes to the nested models in our parent ViewA
        this.listenTo(this.model.parent.modelB, "change", this.render);
        this.listenTo(this.model.parent.modelC, "change", this.render);

        // We can also call any method on our parent view if it is defined
        // $(this.parentView.el).shake();
    }

});

// Create an instance of ViewA with ModelA
// viewA will create its own instance of ViewB
// from inside the initialize() method
var viewA = new ViewA({ model: ModelA });
```

7.3 在子视图中渲染父视图

7.3.1 问题

如何在子视图中渲染父视图？

7.3.2 解决方案

有这样一个场景：一个视图包含另一个视图，如相册程序包含一个大型模型，我们可能会发现需要在子视图中渲染或者重新渲染父视图。其实，解决这个问题非常简单，其中最简单的解决方案就是使用“this.parentView.render();”。

另外，如果控制反转是可用的话，使用事件也能提供同样有效的解决方案。

比如我们希望在一个特定事件发生的时候开始渲染视图。对于这个示例，我们可以调用 somethingHappened 事件。父视图可以在子视图上绑定通知，以便知道事件何时发生。然后，它可以自我渲染。

父视图代码如下：

```
// Parent initialize
this.listenTo(this.childView, 'somethingHappened', this.render);

// Parent removal
this.stopListening(this.childView, 'somethingHappened');
```

子视图代码如下：

```
// After the event has occurred
this.trigger('somethingHappened');
```

子视图将触发一个somethingHappened事件，并且父视图的render函数会被调用。

感谢 Tal Bereznitskey 提供的这些技巧（http://stackoverflow.com/users/269666/tal-bereznitskey）。

7.4 消除视图层级结构

7.4.1 问题

当应用程序设置有多个父视图和子视图时，一旦不需要某个视图时，我们可能想要删除视图相关的 DOM 元素，以及将事件处理程序从子元素上进行解绑。

7.4.2 解决方案

上一个问题的解决方案应该足以解决这种问题，但如果大家想看一个处理子视图的更明确的例子，可以查看如下代码：

```
Backbone.View.prototype.close = function() {
    if (this.onClose) {
        this.onClose();
    }
    this.remove();
};

NewView = Backbone.View.extend({
    initialize: function() {
       this.childViews = [];
    },
```

```
    renderChildren: function(item) {
        var itemView = new NewChildView({ model: item });
        $(this.el).prepend(itemView.render());
        this.childViews.push(itemView);
    },
    onClose: function() {
      _(this.childViews).each(function(view) {
        view.close();
      });
    }
});

NewChildView = Backbone.View.extend({
    tagName: 'li',
    render: function() {
    }
});
```

这里为视图实现一个 close()方法，用于当视图不再需要或者需要重置时消除该视图。

在大多数情况下，删除视图不影响任何相关的模型。例如，如果我们在编写一个博客程序，并且要删除一个评论视图，可能应用程序的另外一个视图也对该评论进行了显示，重新设置集合可能也会影响到这些视图。

感谢 dira 提供的这些技巧（http://stackoverflow.com/users/906136/dira）。

大家可能对第 6 章的 Marionette 复合视图也会感兴趣。

7.5 渲染视图层级结构

7.5.1 问题

假如有一个集合，集合中的每个项本身也可能是一个集合，可以渲染集合中的每个项，也可以渲染本身是集合的项。大家可能遇到的问题是：如何才能渲染具有层级的数据结构的 HTML。

7.5.2 解决方案

解决该问题的最简单方式是使用像 Derick Bailey 的 Backbone.Marionette 这样的框架。该框架包含了一个叫作 CompositeView 类型的视图。

CompositeView 的基础理念是使用同样的视图渲染模型和集合。可以是模板渲染单个的模型，对于该模型中的集合，它也可以对其中的每个项进行视图渲染。默认情况下，它使用相同的 CompositeView 类型，该 CompositeView 类型是为渲染集合中每个项而定义的。我们要做的仅仅是通过 initialize 方法告诉视图实例集合在什么地方，然后就能得到递归层次的渲染。

这里有一个在线的实战示例，地址为：http://jsfiddle.net/derickbailey/AdWjU/。

也可以从这里获取 Marionette 的源代码和文档，地址为：https://github.com/marionettejs/backbone.marionette。

7.6 使用嵌套模型或嵌套集合

7.6.1 问题

Backbone 在其框架内并不包括支持嵌套模型或嵌套集合，为了能够在客户端使用好的模式对结构化数据进行建模，我们应该怎么做？

7.6.2 解决方案

正如我们所见，使用 Backbone 创建表示一组模型的集合是很常见的。但是，根据当前编写应用程序的类型，在模型里嵌套集合也是很常见的。

例如，一个建筑模型，其包含很多的房间模型应该放在一个房间集合里。

可以为每个建筑物暴露一个 this.rooms 集合，并且在建筑物对象打开时延迟加载房间。

```
var Building = Backbone.Model.extend({

    initialize: function(){
        this.rooms = new Rooms;
        this.rooms.url = '/building/' + this.id + '/rooms';
        this.rooms.on("reset", this.updateCounts);
    },

    // ...

});

// Create a new building model
var townHall = new Building;

// once opened, lazy-load the rooms
townHall.rooms.fetch({reset: true});
```

也有许多 Backbone 的插件可以帮助解决嵌套数据结构问题，如 Backbone Relational。对于 Backbone 的模型，该插件可以处理一对一、一对多和多对一的关系，并且还有一些很优秀的文档（http://backbonerelational.org/）。

7.7 更好的模型属性验证

7.7.1 问题

就像我们在本书前面所了解的，模型上的 validate 方法是通过 set（validate 选项是 set 时）和 save 进行调用的。通过给这些方法传值，从而将需要更新的模型属性传递给 validate 方法。

默认情况下，当我们自定义一个 validate 方法时，不管模型的哪个属性被修改了，每次 Backbone 都会将该模型的所有属性传递给该方法。

这就意味着，判断哪些字段被修改或者经过验证，而同时无需关注其他没有修改的字段，是一个很大的挑战。

7.7.2 解决方案

为了更好地说明这个问题，我们来看一个典型的注册表单场景：

- 使用 blur 事件进行字段验证；
- 验证一个字段时，无需关注其他的模型属性（其他表单数据）是否有效。

这里有一个场景：假设在表单里，我们在 first name、last name 和 email 输入框触发焦点，不输入任何东西然后离开焦点。每个输入框旁边都应该显示一个“This field is required”消息。

HTML 代码如下：

```
<!doctype html>
<html>
<head>
  <meta charset=utf-8>
  <title>Form Validation - Model#validate</title>
  <script src='http://code.jquery.com/jquery.js'></script>
  <script src='http://underscorejs.org/underscore.js'></script>
  <script src='http://backbonejs.org/backbone.js'></script>
</head>
<body>
  <form>
    <label>First Name</label>
```

```
    <input name='firstname'>
    <span data-msg='firstname'></span>
    <br>
    <label>Last Name</label>
    <input name='lastname'>
    <span data-msg='lastname'></span>
    <br>
    <label>Email</label>
    <input name='email'>
    <span data-msg='email'></span>
  </form>
</body>
</html>
```

使用该表单，可以利用当前 Backbone 的 validate 方法编写基本验证，实现代码类似如下：

```
validate: function(attrs) {

    if(!attrs.firstname) return 'first name is empty';
    if(!attrs.lastname) return 'last name is empty';
    if(!attrs.email) return 'email is empty';

}
```

可惜这种方式在每次离开其他焦点的时候，都只会触发 firstname 错误消息，并且该错误消息只会出现在 firstname 输入框旁边。

潜在的解决方案是验证所有字段，并返回所有的错误消息：

```
validate: function(attrs) {
  var errors = {};

  if (!attrs.firstname) errors.firstname = 'first name is empty';
  if (!attrs.lastname) errors.lastname = 'last name is empty';
  if (!attrs.email) errors.email = 'email is empty';

  if (!_.isEmpty(errors)) return errors;
}
```

可以调整一下解决方案，在表单里为每个输入框定义一个字段模型（Field），通过传参的形式来使用，代码如下：

```
$(function($) {

  var User = Backbone.Model.extend({
    validate: function(attrs) {
      var errors = this.errors = {};

      if (!attrs.firstname) errors.firstname = 'firstname is required';
      if (!attrs.lastname) errors.lastname = 'lastname is required';
      if (!attrs.email) errors.email = 'email is required';
```

```
      if (!_.isEmpty(errors)) return errors;
    }
  });

  var Field = Backbone.View.extend({
    events: {blur: 'validate'},
    initialize: function() {
      this.name = this.$el.attr('name');
      this.$msg = $('[data-msg=' + this.name + ']');
    },
    validate: function() {
      this.model.set(this.name, this.$el.val(), {validate:true});
      this.$msg.text(this.model.errors[this.name] || '');
    }
  });

  var user = new User;

  $('input').each(function() {
    new Field({el: this, model: user});
  });

});
```

上述代码能够使用，为每个属性进行单独的检查，并对对应的失去焦点字段设置错误消息。@braddunbar 对上述例子创建的一个示例，可以在此进行访问：http://jsbin.com/afetez/2/edit。

遗憾的是，该方案每次都要验证所有的字段，尽管我们只想在仅仅改变内容的字段上显示错误消息。如果有多个客户端验证方法，每次验证每个属性的时候，我们可能不想调用所有的验证方法，所以该解决方案可能并不适合所有的人。

7.7.3 Backbone.validateAll

上述场景更好的潜在替代方案是使用@gfranko 的 Backbone.validateAll 插件（https://github.com/gfranko/Backbone.validateAll），用于验证特定的模型属性（或表单字段）而无需关注其他模型属性（或表单字段）的验证。

下面是针对当前场景，使用该插件构建的部分 User 模型以及 validate 方法，代码如下：

```
// Create a new User Model
var User = Backbone.Model.extend({

    // RegEx Patterns
    patterns: {

        specialCharacters: '[^a-zA-Z 0-9]+',
```

```
      digits: '[0-9]',

      email: '^[a-zA-Z0-9._-]+@[a-zA-Z0-9][a-zA-Z0-9.-]*[.]{1}[a-zA-Z]{2,6}$'
  },

// Validators
  validators: {

      minLength: function(value, minLength) {
        return value.length >= minLength;

      },

      maxLength: function(value, maxLength) {
        return value.length <= maxLength;

      },

      isEmail: function(value) {
        return User.prototype.validators.pattern(value,
        User.prototype.patterns.email);

      },

      hasSpecialCharacter: function(value) {
        return User.prototype.validators.pattern(value,
        User.prototype.patterns.specialCharacters);

      },
     ...

// We can determine which properties are getting validated by
// checking to see if properties are equal to null

    validate: function(attrs) {

      var errors = this.errors = {};

      if(attrs.firstname != null) {
          if (!attrs.firstname) {
              errors.firstname = 'firstname is required';
              console.log('first name isEmpty validation called');
          }

          else if(!this.validators.minLength(attrs.firstname, 2))
            errors.firstname = 'firstname is too short';
          else if(!this.validators.maxLength(attrs.firstname, 15))
            errors.firstname = 'firstname is too large';
          else if(this.validators.hasSpecialCharacter(attrs.firstname))
            errors.firstname = 'firstname cannot contain special characters';
      }

      if(attrs.lastname != null) {
```

```
        if (!attrs.lastname) {
            errors.lastname = 'lastname is required';
            console.log('last name isEmpty validation called');
        }

        else if(!this.validators.minLength(attrs.lastname, 2))
          errors.lastname = 'lastname is too short';
        else if(!this.validators.maxLength(attrs.lastname, 15))
          errors.lastname = 'lastname is too large';
        else if(this.validators.hasSpecialCharacter(attrs.lastname))
          errors.lastname = 'lastname cannot contain special characters';

    }
```

这允许我们在 validate 方法里编写逻辑来判断当前哪些表单字段正在被设置/验证，并且忽略哪些未进行设置的模型属性。

使用该插件同样也非常简单：可以简单地定义一个新模型实例，然后通过 validateAll 选项使用该插件的默认行为在模型上设置数据：

```
var user = new User();
user.set({ 'firstname': 'Greg' }, {validate: true, validateAll: false});
```

这样就行了。Backbone.validateAll 没有覆盖 Backbone 默认的验证逻辑，所以，如果不管我们是否关注字段验证的性能，该插件都能够完全用于相应的场景。然而，本节的两种方案都是可以正常使用的。性能比较可参考网址：http://jsperf.com/backbone-validateall。

7.7.4 Backbone.Validation

就像我们看到的，Backbone 的 validate 方法默认是返回 undefined，要让模型验证到位，需要使用自定义验证逻辑覆盖该返回值。开发人员在实现验证逻辑中使用嵌套 if/else 的时候经常碰到这个问题，事情越来越复杂时，它将变得不可维护。

另外一个很有用的 Backbone 插件是 Backbone.Validation（https://github.com/thedersen/backbone.validation）。它通过提供一个可扩展的方式在模型上定义验证规则，并重载幕后的 validate 方法，以解决这个问题。

该插件其中一个非常有用的方法是通过 preValidate 方法进行（伪）即时验证。用户按下键盘在输入框输入数据但还没有改变模型属性值时，该方法用于检查该值是否有效。可以通过调用 preValidate 方法传入要验证的属性名称和属性值，对模型上的任何属性进行验证。

```
// If the value of the attribute is invalid, a truthy error message is returned
// if not, it returns a falsy value

var errorMsg = user.preValidate('firstname', 'Greg');
```

7.7.5 特定表单验证类

也就是说，验证问题的最优解决方案不一定适合我们的模型属性验证。相反，我们可以为特定表单的验证编写特殊的函数来实现，此外，还有很多优秀的 JavaScript 表单验证库能够有助于解决这个问题。

如果想把验证依附在模型上，可以把它编写成类函数：

```
User.validate = function(formElement) {
  //...
};
```

欲获得关于 Backbone 验证插件的更多信息，请访问 Backbone wiki 文档（https://github.com/documentcloud/backbone/wiki/Extensions,-Plugins,-Resources#model）。

7.8 避免多个 Backbone 版本的冲突

7.8.1 问题

在编写 Backbone 实例时，同一个页面可能要使用多个版本的 Backbone。怎样做才能不引起冲突？

7.8.2 解决方案

像大多数客户端项目一样，Backbone 的代码包装在一个即时调用的函数表达式里：

```
(function(){
  // Backbone.js
}).call(this);
```

要解决冲突问题，需在配置阶段进行一些设置。Backbone 名称空间被创建、同一页面上多个版本的 Backbone 是通过 noConflict 模式进行支持的：

```
var root = this;
var previousBackbone = root.Backbone;

Backbone.noConflict = function() {
  root.Backbone = previousBackbone;
  return this;
};
```

可以像下面这样，通过调用 noConflict 在同一个页面上使用多个版本的 Backbone：

```
var Backbone19 = Backbone.noConflict();
// Backbone19 refers to the most recently loaded version,
// and `window.Backbone` will be restored to the previously
// loaded version
```

7.9 构建层级模型和层级视图

7.9.1 问题

使用 Backbone 如何进行继承？在相似模型或视图上如何进行代码共享？如何调用已经被重载的方法？

7.9.2 解决方案

对于继承，受 goog.inherits（Google 在 Closure 库的实现）的启发，Backbone 在内部使用一个 inherits 函数。这主要是一个用于正确设置原型链的函数。

```
var inherits = function(parent, protoProps, staticProps) {
    ...
```

这里唯一的差别是，Backbone 的 API 接受实例和静态方法这两个对象。

根据这一点，鉴于继承目的，Backbone 所有的对象都包含一个 extend 方法，代码如下：

```
Model.extend = Collection.extend = Router.extend = View.extend = extend;
```

使用 Backbone 的大多数开发工作都是围绕继承这些对象来做的，其用于模拟经典的面向对象实现。

上述内容和 ECMAScript 5 的 Object.create 不太一样，因为 Object.create 实际上是将属性（方法和值）从一个对象复制到另一个对象上。由于这并不足以 Backbone 的继承和类模型，extend 执行了如下步骤：

（1）检查实例方法是否有构造函数属性。如果有，类的构造函数就会使用；否则，父类的构造函数就会使用（如 Backbone.Model）。

（2）调用 Underscore 的 extend 方法，将父类的方法添加到新的子类上。

（3）无构造函数的函数，其 prototype 属性设置为父对象的原型，this 的新实例设置为子对象的 prototype 属性。

（4）再次调用 Underscore 的 extend 方法，将静态方法和实例方法添加到子类上。

（5）给子对象原型的构造函数和__super__属性赋值。

（6）该模式也在 CoffeeScript 的类上进行使用，所以 Backbone 类也兼容 CoffeeScript 类。

extend 可以用在更多地方，喜欢使用混入（mixin）的开发人员希望可以在任何自定义对象上定义功能，然后可以逐字将对象上的所有方法和属性复制并粘贴到一个 Backbone 对象上：

例如：

```
var MyMixin = {
  foo: 'bar',
  sayFoo: function(){alert(this.foo);}
};

var MyView = Backbone.View.extend({
 // ...
});

_.extend(MyView.prototype, MyMixin);

var myView = new MyView();
myView.sayFoo(); //=> 'bar'
```

我们可以更进一步扩展，将其应用于视图继承。下面的示例描述了如何用一个视图扩展另外一个视图：

```
var Panel = Backbone.View.extend({
});

var PanelAdvanced = Panel.extend({
});
```

7.9.3 调用重载方法

然而，如果在 Panel 上有一个 initialize() 方法，而在 PanelAdvanced 中也有一个 initialize() 方法，Panel 上的 initialize() 方法就不会被调用，所以需要显式调用 Panel 上的 initialize() 方法：

```
var Panel = Backbone.View.extend({
  initialize: function(options){
    console.log('Panel initialized');
    this.foo = 'bar';
  }
});

var PanelAdvanced = Panel.extend({
  initialize: function(options){
    Panel.prototype.initialize.call(this, [options]);
    console.log('PanelAdvanced initialized');
    console.log(this.foo); // Log: bar
  }
});
```

```
// We can also inherit PanelAdvaned if needed
var PanelAdvancedExtra = PanelAdvanced.extend({
  initialize: function(options){
    PanelAdvanced.prototype.initialize.call(this, [options]);
    console.log('PanelAdvancedExtra initialized');
  }
});

new Panel();
new PanelAdvanced();
new PanelAdvancedExtra();
```

这不是最优的解决方案，因为如果有很多视图需要继承 Panel，就需要在所有视图的 initialize()方法里都调用 Panel 的 initialize()方法。

要注意的是，如果 Panel 现在没有 initialize()方法，但我们选择在将来添加它，那么我们将来就得将所有继承它的类都翻一遍，以确保它们都调用了 Panel 的 initialize()方法。

所以，下面是定义 Panel 的另外一种方式，这样继承视图就不需要再调用 Panel 的 initialize()方法了：

```
var Panel = function (options) {
  // put all of Panel's initialization code here
  console.log('Panel initialized');
  this.foo = 'bar';

  Backbone.View.apply(this, [options]);
};

_.extend(Panel.prototype, Backbone.View.prototype, {
  // put all of Panel's methods here. For example:
  sayHi: function () {
    console.log('hello from Panel');
  }
});

Panel.extend = Backbone.View.extend;

// other classes then inherit from Panel like this:
var PanelAdvanced = Panel.extend({
  initialize: function (options) {
    console.log('PanelAdvanced initialized');
    console.log(this.foo);
  }
});

var panelAdvanced = new PanelAdvanced();
// Logs: Panel initialized, PanelAdvanced initialized, bar
panelAdvanced.sayHi(); // Logs: hello from Panel
```

使用得当的话，Underscore 的 extend 方法可以为我们编写冗余代码节省大量的时间和精力。

感谢 Alex Young（http://dailyjs.com/）、Derick Bailey（http://stackoverflow.com/users/93448/derick-bailey）和 JohnnyO（http://stackoverflow.com/users/188740/johnnyo）为这些技巧提供的提示。

7.9.4 Backbone-Super

Lukas Olson 使用 John Resig 的继承脚本在其编写的 Backbone-Super 插件里，为 Backbone.Model 添加了一个_super 方法。该方法不再使用 Backbone.js 文档里的 Backbone.Model.prototype.set.call 进行调用，而是使用_super 调用进行替代。

Backbone-Super 插件：https://github.com/lukasolson/Backbone-Super。

John Resig 的继承脚本：http://ejohn.org/blog/simple-javascript-inheritance/。

```
// This is how we normally do it
var OldFashionedNote = Backbone.Model.extend({
  set: function(attributes, options) {
    // Call parent's method
    Backbone.Model.prototype.set.call(this, attributes, options);
    // some custom code here
    // ...
  }
});
```

引用该插件后，使用如下语法可以做同样的事情：

```
// This is how we can do it after using the Backbone-super plug-in
var Note = Backbone.Model.extend({
  set: function(attributes, options) {
    // Call parent's method
    this._super(attributes, options);
    // some custom code here
    // ...
  }
});
```

7.10 事件聚合器和中介者

7.10.1 问题

如何通过一个单一对象引导多个事件源？

7.10.2 解决方案

使用事件聚合器。通常，开发者认为应该使用中介者来解决这个问题，所以让我们研究一下什么是聚合器，什么又是中介者，以及它们之间有什么不同。

设计模式通常只是在语义和目的上进行区分——也就是说，语言是用来描述模式的不同之处，超越于特定模式的实现。它常常归结为正方形、矩形以及多边形之间的区别。使用上述 3 个模式可以创建相同的最终结果，即便仍然能满足正方形的约束条件，或者可以使用多边形来创建一个无限大且更复杂的东西。

就中介者和事件聚合器模式而言，由于其实现的相似之处，有时候看起来模式是可互换的。然而，这些模式的语义和目的却非常不同。即便其实现都使用一些相同的核心结构，我认为两者之间也会有一个很明显的区别。我还认为，由于其不同性，在通信过程中它们不应该进行互换或颠三倒四。

7.10.3 事件聚合器

根据 Martin Fowler 的说法，事件聚合器的核心理念是通过单一对象引导多个事件源，以便其他对象可以订阅事件，而不必知道每个事件源。

1. Backbone 的事件聚合器

Backbone.js 最简单的事件聚合器是直接内置在 Backbone 对象上的。

```
var View1 = Backbone.View.extend({
  // ...

  events: {
    "click .foo": "doIt"
  },

  doIt: function(){
    // trigger an event through the event aggregator
    Backbone.trigger("some:event");
  }
});

var View2 = Backbone.View.extend({
  // ...

  initialize: function(){
    // subscribe to the event aggregator's event
    Backbone.on("some:event", this.doStuff, this);
  },

  doStuff: function(){
    // ...
```

```
  }
})
```

在本例中，第一个视图在其 DOM 元素被单击的时候触发一个事件。事件是通过 Backbone 内置的事件聚合器（Backbone 对象本身）触发的。当然，我们很容易在 Backbone 里创建自己的事件聚合器，但我们需要记住一点，那就是保持代码简单。

2. jQuery 的事件聚合器

大家知道 jQuery 有一个内置的事件聚合器吗？jQuery 不称它为事件聚合器，但它就在那里，它作用于 DOM 事件。碰巧它看起来也很像 Backbone's 的事件聚合器：

```
$("#mainArticle").on("click", function(e){

  // handle click event on any element underneath our #mainArticle element

});
```

上述代码设置了一个事件处理函数，等待未知数量的事件源去触发单击事件，并且允许任意数量的监听者附加到这些事件发布者的事件上。jQuery 仅仅将事件聚合器作用在 DOM 上。

7.10.4 中介者

中介者用于表示一个对象在多个对象之间进行协调交互（逻辑和行为）。基于其他对象的行为（或不作为）和输入，它决定何时调用哪个对象。

Backbone 没有像其他 MV*框架一样有构建中介者的思想，但这并不意味着我们不能用一行代码去编写它：

```
var mediator = {};
```

当然，这只是 JavaScript 中的一个对象字面量。再次强调，我们这里谈论的日语义。中介者的目的是控制对象之间的工作流程，一个对象字面量足以能够解释它，我们真的不需要其他内容来解释。

```
var orgChart = {

  addNewEmployee: function(){

    // getEmployeeDetail provides a view that users interact with
    var employeeDetail = this.getEmployeeDetail();

    // when the employee detail is complete, the mediator (the 'orgchart' object)
    // decides what should happen next
    employeeDetail.on("complete", function(employee){
```

```
        // set up additional objects that have additional events, which are used
        // by the mediator to do additional things
        var managerSelector = this.selectManager(employee);
        managerSelector.on("save", function(employee){
          employee.save();
        });

      });
    },

    // ...
  }
```

该示例展示了一个中介者对象的基本实现，它使用了基于 Backbone 的对象，可以触发和订阅事件。过去我经常将这种类型的对象作为工作流对象，但事实上，它是一个中介者。它是在其他很多对象之间处理工作流程的一个对象，聚合工作流程的职责落到单个对象上，其结果是一个易于理解和维护的工作流。

7.10.5 相似性与差异性

毫无疑问，我在这里已经展示了事件聚合器和中介者之间的相似性。相似性归结为两个主要点：事件和第三方对象。虽然最好的情况下，这些差异是很小的，但当我们深入到模式的目的时，可以看到其实现是非常不同的，模式的本质变得更加明显。

1. 事件

在前面的示例中，事件聚合器和中介者都使用了事件。很显然，事件聚合器负责处理事件；毕竟它的名字在那呢。中介者只使用事件，是因为它使得处理 Backbone 变得更容易。没有什么能说明中介者必须和事件一起构建。通过将中介者引用到子对象上，可以使用回调方法或者任何其他的方式都可以创建一个中介者。

区别是：为什么这两种模式都使用事件。作为一种模式，事件聚合器是用于处理事件的；而中介者只是因为方便才使用事件的。

2. 第三方对象

事件聚合器和中介者的设计，都是通过使用第三方对象来便于处理事情。对于事件发布者和事件订阅者来说，事件聚合器本身就是一个第三方对象。它是作为一个事件经过的中央集线器。中介者对于其他对象来说也是一个第三方对象。所以，区别是什么？为什么不将事件聚合器叫成中介者？答案主要是根据应用程序逻辑和工作流是在哪里编写的。

对于事件聚合器来说，第三方对象只是促使事件从未知数量的事件源传递到未知数

量的事件处理程序上。所有需要的工作流程和业务逻辑都是直接在用于触发事件和处理事件的对象里编写的。

对于中介者来说，业务逻辑和工作流聚合在中介者自身里。中介者基于其了解的因素，决定什么时候一个对象需要调用其方法或更新其属性。它封装了工作流程和过程，协调多个对象来产生所需的系统行为。参与这个工作流的每个对象都知道如何执行自己的任务。是它的中介者，通过在单个对象之上的角度进行决策，来告诉对象什么时候去执行任务。

事件聚合器促进的是一个“触发即丢弃”的沟通模式。对象触发事件时并不关心是否有任何订阅者，它只是触发事件并继续。中介者虽然可能是使用事件做决策，但它绝对不会触发事件和丢弃事件。中介者注重一系列已知的输入或活动，以便对一系列已知的参与者（对象）促进和协调额外的行为。

7.10.6 关系：何时用，用哪个

理解事件聚合器和中介者的相似性和差异性，从语义上说非常重要。同样重要的是要理解何时该用哪个模式。模式的基本语义和目的可以解决何时的问题，但使用模式的实际体验将帮助我们理解更多微妙点以及必须要做的有细微差别的决策。

1．事件聚合器的使用

一般来说，当有太多的对象需要直接监听的时候，或者对象是完全无关的时候，要使用事件聚合器。

当两个对象已经有直接关系了，比如，父视图和子视图——此时事件聚合器可能不会有什么用处。让子视图触发事件，然后父视图处理事件。在 Backbone 的集合和模型中，这是最常见的情况，所有的模型事件通过其父集合进行冒泡。集合通常使用模型事件来修改自身的状态或者其他模型。处理集合里已选中的项就是一个很好的例子。

jQuery 的 on 方法作为事件聚合器是监听众多对象的一个很好的例子。如果有 10、20 或 200 个 DOM 元素可以触发 click 事件，那么在所有元素上单独设置监听器可能是个不明智的想法，这可能会迅速恶化应用程序性能和用户体验。而使用 jQuery 的 on 方法可以让我们能够聚集所有的事件，并将 10、20 或 200 个事件处理程序开销减少到 1 个。

对象间如果是间接的关系，使用事件聚合器也是很好的时机。在 Backbone 应用程序中，多个视图对象之间没有直接的关系，但需要通信是很常见的。例如，一个菜

单系统可能有一个视图用于处理菜单项单击，但是我们不希望菜单直接绑定到单击菜单项才显示的具体内容和信息的视图上。从长远来看，将内容和菜单耦合在一起会让代码变得很难维护。我们可以使用事件聚合器触发 menu:click:foo 事件，然后让一个 foo 对象处理该单击事件，并在屏幕上显示内容。

2. 中介者的使用

当两个或两个以上的对象之间有间接的工作关系，并且这些对象的业务逻辑或者工作流需要决定交互和协调时，中介者是最好的选择。

在 7.10.4 小节里的 orgChart 示例展示的向导界面就是一个很好的例子，有多个视图促进向导的整个工作流程。与其让它们之间互相引用紧密耦合在一起，不如通过引入一个中介者对它们进行解耦，并更加显式地模拟其工作流程。

中介者提取工作流的实现细节，并在更高层面上创建一个更自然的抽象，向我们更迅速地展示工作流是什么。在工作流中，我们不再需要深入每个视图的细节来了解工作流到底是什么。

7.10.7 事件聚合器与中介器一起使用

事件聚合器和和中介者之间的关键区别，以及为什么这些模式名称不能互换，最好可以让一个范例来说明如何让它们在一起使用。事件聚合器的菜单示例也是介绍中介者的最好例子。

在整个应用程序里，单击一个菜单项可能引发一系列的变化，其中一些变化将独立于其他变化，在这里使用事件聚合器是有意义的；另外一些变化可能是内部彼此相关，也许可以使用一个中介者来促成这些变化。接着，可以对中介者进行设置，以便可以监听事件聚合器。它可以运行自身逻辑和过程来促进和协调众多彼此相关、但和原有事件源无关的对象。

```
var MenuItem = Backbone.View.extend({

  events: {
    "click .thatThing": "clickedIt"
  },

  clickedIt: function(e){
    e.preventDefault();

    // assume this triggers "menu:click:foo"
    Backbone.trigger("menu:click:" + this.model.get("name"));
  }
```

```
});

// ... somewhere else in the app

var MyWorkflow = function(){
  Backbone.on("menu:click:foo", this.doStuff, this);
};

MyWorkflow.prototype.doStuff = function(){
  // instantiate multiple objects here.
  // set up event handlers for those objects.
  // coordinate all of the objects into a meaningful workflow.
};
```

在该示例中，当拥有正确模型的菜单项被单击时，menu:click:foo 事件被触发。MyWorkflow 对象的实例（假设已经实例化了）将处理这个特定的事件并协调所有它知道的对象，来创建所需的用户体验和工作流程。

事件聚合器和中介者的结合可以在代码或应用程序本身都创造更有意义的体验。通过事件聚合器，在菜单和工作流程上，现在有了清晰的分离；通过中介者，我们仍能保持工作流程清晰并且可维护。

7.10.8 模式语言：语义

在所有这些讨论中，最重要的一点是：语义。通过对模式进行命名，只有在沟通各方用同样的方式理解语言，描述沟通的目的和语义才是可行并且有效的。

如果我说“苹果”，那我是在说什么？是在谈论一个水果？还是在谈论一个科技公司与消费者产品？就像 Sharon Cichelli 所说：“在我们学会如何用语言以外的东西进行沟通以前，语义仍然很重要。”

第 8 章

模块化开发

当我们说应用程序是模块化程序时，通常是指它由一组高度解耦的模块组成，并且每个模块都有截然不同的功能。正如你可能知道的，松散耦合的应用程序尽可能地消除依赖，可以提高其可维护性。当有效地实现模块化时，很容易看到系统一部分改变时可能会影响的另一部分。

与一些传统的编程语言不同，JavaScript 的当前迭代（ECMA-262）并没有为开发人员提供这样整洁、有组织的代码模块。

开发人员只能结合<script>标签或脚本加载器，依靠模块或对象字面量模式的变体来实现模块化开发。大量使用这些变体，模块脚本使用命名空间在 DOM 里串联在一起，虽然其命名空间是由一个单一全局对象所描述的，但它仍然可能有名称冲突。除非手动或通过第三方工具进行管理，否则还没有清晰的方式来处理依赖管理。

这些问题的原始解决方案可以通过 ES6（http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts）（官方 JavaScript 规范的下一个版本）的模块提议（http://wiki.ecmascript.org/doku.php?id=harmony:modules）来解决，好在 JavaScript 模块化编程从未如此简单过，你现在就可以开始。

本书下面的部分，我们将了解在应用程序里，如何使用 AMD（异步模块定义）和 RequireJS 将代码单元整洁地包装为可管理的模块。同时也会涉及另外一种叫作 Lumbar 的方法，它使用路由来确定模块何时被加载。

8.1 使用 RequireJS 和 AMD 组织模型

部分内容由 Jack Franklin（https://github.com/jackfranklin）提供。

RequireJS 是一个流行的脚本加载，由开发人员 James Burke 编写，它一直致力于塑造我们刚刚所讨论的 AMD 模块格式。此外，RequireJS 可以帮我们加载多个脚本文件、定义有依赖或者没有依赖的模块、加载像文本文件这样的非脚本依赖。

8.1.1 多个脚本文件的可维护性问题

大家可能会认为使用 RequireJS 没有什么好处，因为可以通过使用多个<script>标记很简单地加载 JavaScript 文件。然而，这样做有很多缺点，包括增加 HTTP 开销。

浏览器每次加载一个用<script>引用的文件，就会发送一个 HTTP 请求来加载文件的内容。对每个要加载的文件都要使用一个新的 HTTP 请求，这可能导致以下问题。

- 浏览器限制了可用的并发请求，所以加载多个文件的时候经常很慢，因为同一时间只能加载固定数量的文件。这个数字取决于用户设置和浏览器，但通常是 4～8 个。在进行 Backbone 应用程序开发时，将应用程序划分成多个 JS 文件是很好的做法，因为这样容易快速突破这一限制。可以将代码压缩到一个文件里作为构建过程的一部分，以缩减文件的数量，但这解决不了下一个问题。
- 脚本被同步加载，这意味着在加载脚本的时候，浏览器不能继续进行页面渲染。

RequireJS 要做的是脚本的异步加载。这意味着必须微调我们的代码——对于 RequireJS 的一小段代码，我们不能移除<script>元素——但它带来的好处是非常值得我们这样做的：

- 异步加载脚本意味着其加载过程是异步非阻塞方式。脚本加载的时候，浏览器可以继续渲染页面的剩余部分，以加快页面的初始加载时间。
- 加载模块的时候可以更智能，可以进行更多的控制，以确保其依赖模块都能按照正确的顺序进行加载。

8.1.2 需要更好的依赖管理

依赖管理是一个很具挑战性的课题，尤其是在浏览器中编写 JavaScript 代码时。依赖管理的最接近情况，默认情况下只是确保<script>标签能够按顺序显示，比如，一段代码，其依赖于另外一个文件的代码，该代码必须在依赖文件加载之后才能加载。这并非一种好的方式。正如已经讨论过的，这种方式加载多个文件时对性能有不利影响；以特定顺序加载多个文件是非常不稳定的。

在程序需要的时候才加载代码是 RequireJS 的专长。与其在页面初始加载的时候加载所有的 JavaScript 代码，不如在代码需要时才进行模块的动态加载，以避免在用

户第一次访问应用程序时加载所有的代码，从而加快页面的初始加载时间。

此时，思考一下 Gmail 的 Web 客户端。当用户第一次访问时加载页面，谷歌可以简单地将部件进行隐藏，如聊天模块，一直到用户想用它时才显示（通过单击扩展）。通过动态依赖加载，谷歌可以在用户单击时才加载聊天模块，而不是在页面第一次初始化时强迫所有用户都加载它。这可以改进性能和加载时间，在构建大型应用程序时绝对有用。随着应用程序代码库的增长，这就显得尤为重要。

这里要注意的重点是，虽然在开发应用程序时不用脚本加载器也是可以的，但如果使用像 RequireJS 这样的工具，则会有显著的好处。

8.1.3 异步模块定义（AMD）

RequireJS 实现了 AMD 规范（https://github.com/amdjs/amdjs-api/wiki/AMD），它定义了一个用于编写模块代码和管理依赖的方法。RequireJS 官方网站也有章节记录了实现 AMD 背后的原因（http://requirejs.org/docs/whyamd.html）：

AMD 格式来自于模块化格式需求，其优于如今的“为隐式依赖手动按序编写一堆<script>标签”，并且易于在浏览器中直接使用。其良好的调试特性，不需要特定的服务器工具。

8.1.4 使用 RequireJS 编写 AMD 模块

正如之前所讨论的，AMD 格式的整体目标是提供一个当前开发人员可用的 JavaScript 模块化开发的解决方案。使用脚本加载器时需要知道的两个关键概念是定义模块用的 define()方法和加载依赖模块用的 require()方法。define()使用如下签名定义命名或匿名模块：

```
define(
    module_id /*optional*/,
    [dependencies] /*optional*/,
    definition function /*function for instantiating the module or object*/
);
```

从行内注释可以看到，module_id 是一个可选的参数，通常只有在非 AMD 格式工具使用时才定义（边界情况也可能有用到）。当该参数不提供时，我们称该模块为匿名模块（anonymous）。使用匿名模块时，RequireJS 要使用匿名模块的文件路径作为模块 id(module id)，因此应该在 define()调用时省略模块 id，以应用 DRY（Don't Repeat Yourself，不要重复自己）格言。

dependencies 参数是一个数组，代表该模块所依赖的其他模块，第三个参数可以是用于实例化该模块的可执行函数，也可以是一个对象。

可以像下面这样，使用 define()定义一个简洁的模块（兼容 RequireJS）：

```
// A module ID has been omitted here to make the module anonymous

define(['foo', 'bar'],
    // module definition function
    // dependencies (foo and bar) are mapped to function parameters
    function ( foo, bar ) {
        // return a value that defines the module export
        // (i.e the functionality we want to expose for consumption)

        // create your module here
        var myModule = {
            doStuff:function(){
                console.log('Yay! Stuff');
            }
        }

        return myModule;
});
```

RequireJS 非常智能，可以为脚本文件自动推断.js 扩展名。因此，指定依赖模块时通常省略这个扩展名。

这里还有一个 define()的语法糖版本可用（http://requirejs.org/docs/whyamd.html#sugar），它允许我们使用 require()将依赖项作为局部变量。使用 Node 的开发人员可能会很熟悉它，它可以很容易地添加或删除依赖。下面是上述例子使用替代语法后的版本：

```
// A module ID has been omitted here to make the module anonymous

define(function(require){
        // module definition function
    // dependencies (foo and bar) are defined as local vars
    var foo = require('foo'),
        bar = require('bar');

    // return a value that defines the module export
    // (i.e., the functionality we want to expose for consumption)

    // create your module here
    var myModule = {
        doStuff:function(){
            console.log('Yay! Stuff');
        }
    }

    return myModule;
});
```

require()方法通常用于在顶级 JavaScript 文件或者是需要动态获取依赖的模块中加载代码。如下代码是它的示例用法：

```
// Consider 'foo' and 'bar' are two external modules
// In this example, the 'exports' from the two modules loaded are passed as
// function arguments to the callback (foo and bar)
// so that they can similarly be accessed

require( ['foo', 'bar'], function ( foo, bar ) {
    // rest of your code here
    foo.doSomething();
});
```

我的帖子“编写模块化 JS”（http://addyosmani.com/writing-modular-js/）详细涵盖了 AMD 的规范。模块的定义和使用，稍后在本书查看使用 RequireJS 编写的更复杂示例时会提到。

8.1.5 RequireJS 入门

使用 RequireJS 和 Backbone 之前，首先要建立一个非常基本的 RequireJS 项目来展示 RequireJS 是如何使用的。要做的第一件事是下载 RequireJS（http://requirejs.org/）。在 HTML 中加载 RequireJS 时，还需要告诉它网站的 JavaScript 主文件在哪（通常叫类似 app. js 这样的名称，是应用程序的主入口点）。通过在<script>标签里添加一个 data-main 属性即可：

```
<script data-main="app.js" src="lib/require.js"></script>
```

此时，RequireJS 将自动加载 app.js。

1. RequireJS 配置

在 data-main 属性上定义的 JavaScript 主文件里，可以配置 RequireJS 要加载的剩余脚本。可以通过调用 require.config 并传入一个对象来达到这一目的：

```
require.config({
    // your configuration key/values here
    baseUrl: "app",
    // generally the same directory as the script used in a data-main attribute
    // for the top level script
    paths: {},
    // set up custom paths to libraries, or paths to RequireJS plug-ins
    shim: {}, // used for setting up all Shims (see below for more detail)
});
```

配置 RequireJS 的主要原因是添加 Shims，我们接下来会讨论。要查看其他的可用配置选项，建议访问 RequireJS 的官方文档（http://requirejs.org/docs/api.html#config）。

2. RequireJS 里的 Shims

理想情况下，RequireJS 使用的每个库都支持 AMD——也就是说，其使用 define 方法将库定义为模块。然而，一些库，包括 Backbone 和它的一个依赖项 Underscore 都不支持。还好，RequireJS 有一种方法能够解决这个问题。

为了说明这一点，首先让我们映射 Underscore，然后继续映射 Backbone。Shims 可以很简单地进行实现：

```
require.config({
    shim: {
        'lib/underscore': {
            exports: '_'
        }
    }
});
```

注意，当为 RequireJS 指定这些库的路径时，应该省略脚本名称后的.js 扩展名。

这里最重要的一行是："exports: '_'"。该行告诉 RequireJS，"lib/underscore.js" 里的脚本创建的是一个叫作_的全局对象，而不是定义一个模块。此时，当我们需要让 Underscore 作为一个依赖项时，RequireJS 就会知道要给我们这个_全局变量，就好像它已经被脚本定义成一个模块了。我们也可以为 Backbone 做一下映射：

```
require.config({
    shim: {
        'lib/underscore': {
          exports: '_'
        },
        'lib/backbone': {
            deps: ['lib/underscore', 'jquery'],
            exports: 'Backbone'
        }
    }
});
```

再说一下，该配置 RequireJS 返回一个 Backbone 导出的 Backbone 全局变量，但这次，我们需要注意 Backbone 的依赖也被定义了。这意味着，不管下面的代码何时运行，首先要确保依赖文件先运行，然后传入全局的 Backbone 对象到回调函数上：

```
require( 'lib/backbone', function( Backbone ) {...} );
```

不需要对每个库都这样做，只需要处理不支持 AMD 的即可。例如，jQuery 是支持 AMD 的，比如 jQuery 1.7。

欲阅读更多关于 RequireJS 的常用用法，RequireJS API 文档（http://requirejs.org/docs/api.html）非常全面且易懂。

3. 自定义路径

输入 lib/backbone 这样的长路径读起来冗长又乏味。RequireJS 允许我们在配置对象里设置自定义路径。下面的代码，每当引用 underscore 时，RequireJS 都会查找 lib/underscore.js 文件：

```
require.config({
    paths: {
        'underscore': 'lib/underscore'
    }
});
```

当然，它也可以和映射（shim）一起结合使用：

```
require.config({
    paths: {
        'underscore': 'lib/underscore'
    },
    shim: {
        'underscore': {
          exports: '_'
        }
    }
});
```

只是应确保在映射设置里也要使用自定义的路径。此时，可以像下面这样使用映射后的 underscore，但仍然使用自定义路径：

```
require( ['underscore'], function(_) {
// code here
});
```

8.1.6 Require.js/Backbone 示例

至此，我们已经知道了如何定义 AMD 模块，让我们来看看如何包装像视图和集合这样的组件，以便在应用程序任何需要它们的地方，将其作为依赖项进行简单的加载。简单来说，一个 Backbone 模型可能只需要依赖 Backbone 和 Underscore.js。这些都是依赖项，所以在定义新模型的时候，我们可以定义这些依赖项。注意，下面的一些例子，假设我们像前面章节所讨论的那样，已经对 RequireJS 进行了配置来映射 Backbone 和 Underscore。

例如，下面的代码展示的是如何使用 AMD 定义一个模型：

```
define(['underscore', 'backbone'], function(_, Backbone) {
  var myModel = Backbone.Model.extend({

    // Default attributes
    defaults: {
```

```
      content: 'hello world',
    },

    // A dummy initialization method
    initialize: function() {
    },

    clear: function() {
      this.destroy();
      this.view.remove();
    }

  });
  return myModel;
});
```

注意我们如何将 Underscore.js 实例的别名设置为别名_，以及将 Backbone 的别名仅仅设置为 Backbone，以便将非 AMD 代码转换成该模块格式。对于可能需要 jQuery 等其他依赖的视图，我们可以像下面这样做：

```
define([
  'jquery',
  'underscore',
  'backbone',
  'collections/mycollection',
  'views/myview'
  ], function($, _, Backbone, myCollection, myView){

  var AppView = Backbone.View.extend({
  ...
```

再次设置美元符号（$）的别名，使用 AMD 可以非常容易地封装应用程序的任何部分。

这样做使我们易于按照自己的意愿组织 Backbone 应用程序。建议将模块单独存放在文件夹里——例如，模型、集合和视图等都有自己单独的文件夹。RequireJS 不关注所使用的文件夹结构；只要在使用 require 的时候指定正确的路径，它就能够顺畅地读取文件。

作为本章节的一部分内容，我使用 RequireJS 创建了一个非常简单的 Backbone 应用程序，大家可以在 GitHub（https://github.com/javascript-playground/backbone-require-example）上找到。这是一个商店管理人员用的库存程序。管理人员可以根据价格添加新项目或者过滤项目，除此之外没有其他功能。正因为它是如此的简单，所以很容易把精力完全集中在 RequireJS 实现部分，而不用处理复杂的 JavaScript 和 Backbone 逻辑。

该应用程序的基础是 Item 模型，它描述了库存中的单个项目。它的实现非常简单：

```
define( ["lib/backbone"], function ( Backbone ) {
  var Item = Backbone.Model.extend({
    defaults: {
      price: 35,
      photo: "http://www.placedog.com/100/100"
    }
  });
  return Item;
});
```

将单独的模型、集合、视图或相似对象转成兼容 AMD、RequireJS 的对象，通常是很简单的。通常需要的就是第一行，调用 define，并确保返回已定义的对象——本例中是 Item 模型。

现在让我们来为单独的 Item 建立一个视图：

```
define( ["lib/backbone"], function ( Backbone ) {
  var ItemView = Backbone.View.extend({
    tagName: "div",
    className: "item-wrap",
    template: _.template($("#itemTemplate").html()),

    render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      return this;
    }
  });
  return ItemView;
});
```

该视图实际上并不依赖其将要使用的模型，所以唯一的依赖是 Backbone。除此之外，它就是一个常规的 Backbone 视图。除了返回对象以及使用 define 以便 RequireJS 可以使用以外，这里无任何特殊之处。现在来创建一个集合视图，用于显示项目列表。这次我们需要引用 Item 模型，所以我们将其作为一个依赖进行添加：

```
define(["lib/backbone", "models/item"], function(Backbone, Item) {
  var Cart = Backbone.Collection.extend({
    model: Item,
    initialize: function() {
      this.on("add", this.updateSet, this);
    },
    updateSet: function() {
      items = this.models;
    }
  });
  return Cart;
});
```

我们将该集合命名为 Cart，因为它包含一组项目。Item 模型作为第二个依赖，在回调函数上定义第二个参数，我们可以将变量 Item 绑定在 Item 模型上。现在，可以在集合实现中使用变量 Item 了。

最后，让我们来看看这个集合的视图（该文件在应用程序中更大，但我移除了一些，以便更容易查看）：

```
define(["lib/backbone", "models/item", "views/itemview"],
  function(Backbone, Item, ItemView) {
  var ItemCollectionView = Backbone.View.extend({
    el: '#yourcart',
    initialize: function(collection) {
      this.collection = collection;
      this.render();
      this.collection.on("reset", this.render, this);
    },
    render: function() {
      this.$el.html("");
      this.collection.each(function(item) {
        this.renderItem(item);
      }, this);
    },
    renderItem: function(item) {
      var itemView = new ItemView({model: item});
      this.$el.append(itemView.render().el);
    },
    // more methods here removed
  });
  return ItemCollectionView;
});
```

如果了解了这个常见模式，一切就轻而易举了。通过 RequireJS 定义每个对象（模型、视图、集合、路由或其他对象），然后指定它们作为依赖，提供给需要它们的其他对象。大家仍然可以在 GitHub 上找到整个应用程序。

如果你想看看别人是怎么做的，Pete Hawkins 的 Backbone Stack 存储库（https://github.com/phawk/Backbone-Stack）是一个使用 RequireJS 构建 Backbone 应用程序的很好的例子。关于如何使用 Backbone 和 Require，Greg Franko 也写了一个概述（http://gregfranko.com/blog/using-backbone-dot-js-with-require-dot-js/），Jeremy Kahn 的帖子也提到他的方法很巧妙。要看完整的示例程序，TodoMVC 应用程序的 Backbone/Require 版本是一个很好的开始（https://github.com/addyosmani/todomvc/tree/gh-pages/dependency-examples/backbone_require）。

8.1.7 使用 RequireJS 和 Text 插件将模板保持在外部

不管是使用 Underscore、Mustache、Handlebars，还是其他的基于文本的模板格式，将模板移动到外部文件实际上是很简单的。让我们看看如何使用 RequireJS 来做到这个。

RequireJS 有一个特殊插件叫 text.js，用于加载依赖的文本文件。要使用 text 插件，需遵循如下步骤：

（1）下载插件，并将其放置在与应用程序的主 js 相同的目录或者合适的子目录。

（2）接下来，在 RequireJS 配置选项里引用 text.js 插件。下面的代码片段中，我们假设在执行代码之前，RequireJS 已经在页面中引用了。

```
require.config( {
    paths: {
        'text': 'libs/require/text',
    },
    baseUrl: 'app'
} );
```

（3）当 text!前缀用于依赖项时，RequireJS 将自动加载该 text 插件，并将依赖文件转换为一个文本资源。实战的典型示例，可能像下面这样：

```
require(['js/app', 'text!templates/mainView.html'],
    function( app, mainView ) {
        // the contents of the mainView file will be
        // loaded into mainView for usage.
    }
);
```

（4）最后，我们可以将加载的文本资源作为模板来用。我们可以使用带有特定标识符的脚本来存放 HTML 模板。

使用 Underscore.js 的微模板（和 jQuery），通常是下面这样的：

- HTML

```
<script type="text/template" id="mainViewTemplate">
    <% _.each( person, function( person_item ){ %>
        <li><%= person_item.get('name') %></li>
    <% }); %>
</script>
```

- JS

```
var compiled_template = _.template( $('#mainViewTemplate').html() );
```

使用 RequireJS 和 text 插件很简单，只要保存相同的模板到外部文本文件（比如，mainView.html），并做如下操作即可：

```
require(['js/app', 'text!templates/mainView.html'],
    function(app, mainView){
        var compiled_template = _.template( mainView );
    }
);
```

就是这样！现在可以将模板应用到 Backbone 的视图上了，代码类似如下：

```
collection.someview.$el.html( compiled_template
                              ( { results: collection.models } ) );
```

所有的模板解决方案都有自己的自定义方法来处理模板编译工作，但是如果理解了前面的内容，使用任何其他的解决方案取代 Underscore 的微模板都是相对简单的。

8.1.8 使用 RequireJS 优化生产环境中的 Backbone 应用

应用程序一旦编写完成，接下来就要将其部署到生产环境。大多数重要的应用程序都可能会包含几个脚本文件，所以需要进行优化以及最小化，以减少用户访问页面时下载脚本的数量。

RequireJS 项目的一个命令行优化工具 r.js 可以帮助解决这个事情。该工具提供了许多功能，其中包括：

- 优化浏览器加载，并保留动态加载模块功能，使用外部工具将“特定”的脚本连接在一起，并进行压缩，比如 UglifyJS（默认使用）或者 Google 的 Closure 编译器。
- 通过使用@import 导入内联 CSS 文件、去除注释等，来优化 CSS 和样式表。
- 提供在 Node 和 Rhino（稍后提供更多信息）都能够运行 AMD 项目的能力。

如果想把所有的依赖文件都放在一个单独的文件里，r.js 也能帮我们实现。RequireJS 确实支持延迟加载，而应用程序可能确实很小，将 HTTP 请求减少到一个脚本文件也是可行的。

你可能会发现我在第一点里使用了“特定”这个词。RequireJS 优化器只能连接在 require 和 define 调用里（可能已经使用）作为字符字面量定义的模块化脚本。优化器文档（http://requirejs.org/docs/optimization.html）对此做出了澄清，这意味着，像下面这样定义的 Backbone 模块是可以的：

```
define(['jquery', 'backbone', 'underscore', 'collections/sample', 'views/test'],
    function($, Backbone, _, Sample, Test){
        //...
    });
```

但是，下面这样的动态依赖代码将被忽略：

```
var models = someCondition ? ['models/ab', 'models/ac'] :
['models/ba', 'models/bc'];
define(['jquery', 'backbone', 'underscore'].concat(models),
    function($, Backbone, _, firstModel, secondModel){
        //...
    });
```

设计原本就是这样的，是因为它要确保即使优化以后，也能够进行动态依赖/模块加载。

尽管 RequireJS 优化器在 Node 和 Java 环境下都能正常使用，但强烈建议在 Node 上运行，因为执行速度明显更快。

要使用 r.js，应从 RequireJS 下载页面或者通过 NPM 获取到安装文件。要使用 r.js 开始构建项目，需要创建一个新的构建文件。

假设应用程序的代码和外部依赖文件是在 app/libs 目录下，build.js 构建文件像这样，很简单：

```
({
  baseUrl: 'app',
  out: 'dist/main.js',
```

前面的路径是相对于我们项目的 baseUrl，本例中，使用 app 文件夹能够行得通。out 参数告知 r.js，我们想将所有的文件都连接到 dist/目录的一个 main.js 文件中。注意，这里我们需要添加.js 扩展名。前面我们看到，通过文件名引用模块时不需要使用.js 扩展名，但本例中确实需要。

或者，也可以指定 dir 目录，以确保 app 目录中的内容能够复制到该目录中。例如：

```
({
  baseUrl: 'app',
  dir: 'release',
  out: 'dist/main.js'
```

其他额外的选项，比如 modules 和 appDir 也都可以指定，不过它们不兼容 out，但如果你想使用这些选项，我们就来简要讨论一下。

modules 是一个数组，其中可以显式指定需要优化的模块名称。

```
modules: [
    {
        name: 'app',
        exclude: [
            // If you prefer not to include certain
            // libs exclude them here
        ]
    }
```

当指定 appDir 时，baseUrl 是相对于这个参数的。如果没有指定 appDir，则 baseUrl 只是相对于 build.js 文件。

```
appDir: './',
```

回到构建文件，main 参数用于指定我们的主模块；这里使用 include，是因为我们打算用 Almond（https://github.com/jrburke/almond），它是一个 RequireJS 模块的分拆式加载器，能够让我们不必动态加载模块。

```
include: ['libs/almond', 'main'],
wrap: true,
```

include 是另外一个数组，用于指定在构建中需要包含的模块。当指定了 main，r.js 将查看所有 main 依赖的模块，然后将其包含进来。Wrap 将包装 RequireJS 要放到闭包的模块，以便我们的输出可以包含在全局环境里。

```
paths: { backbone: 'libs/backbone', underscore: 'libs/underscore',
      jquery: 'libs/jquery', text: 'libs/text' } })
```

build.js 文件的剩余部分是普通的 paths 配置对象。运行如下代码，可以将项目编译到目标文件里：

```
node r.js -o build.js
```

上述代码应该将编译后的项目放到 dist/main.js 文件中。

构建文件通常存放在项目的 scripts 或 js 目录中。然后，根据文档，该文件可以存放在任何我们想放的地方，但需要相应地修改构建文件里的内容。

只要将 UglifyJS/Closure 工具设置正确了，仅仅几个按键，r.js 就应该能够轻松优化整个 Backbone 项目。

如果想了解更多关于构建文件的内容，James Burke 有一个包括所有可选项且含有大量注释的示例文件（https://github.com/jrburke/r.js/blob/master/build/example.build.js）。

8.2 总结

JavaScript 中的依赖管理很有挑战性。一旦你要考虑模块化，如果将应用程序分成几个文件，还需要考虑每个文件的依赖模块，并确保能够按照正常的顺序进行加载。不使用强命名空间约定，自定义对象很容易污染全局命名空间。AMD（和 RequireJS）可以简化这个过程，提供的语法糖用于定义可重用的模块以及其依赖项，并且不会污染全局命名空间。尽管 AMD 不一定适合所有的人，但它也能协助处理代码结构，整洁地将模型、视图和集合进行模块化，从而形成页面中的区域。一定要评估 AMD 风格是否适合你，如果适合，你能得到不少好处。

第 9 章

练习 3：第一个模块化的 Backbone /RequireJS 应用程序

本章，我们将看看第一个实用的 Backbone 和 RequireJS 项目——如何构建模块化的 Todo 应用程序。和第 4 章的练习 1 类似，应用程序允许我们添加新 todo 项、编辑新 todo 项以及清理标记为已完成的 todo 项。有关更高级的应用，请阅读第 12 章。

可以在存储库的 practicals/modular-todo-app 目录里找到完整的源代码（感谢 Thomas Davis 和 Jerome Gravel-Niquet）。另外，我的 TodoMVC 项目（https://github.com/addyosmani/todomvc）也复制了一份，AMD 和非 AMD 版本都包含了。

9.1 概述

编写模块化的 Backbone 应用程序是一个很简单的过程。然而，如果选择使用 AMD 作为模块化格式的话，需要注意一些关键性的概念差异：

- 由于 AMD 不是 JavaScript 和浏览器的原生特性，因此必须使用脚本加载器（如 RequireJS 或 curl.js），以便支持使用 AMD 模块格式来定义组件和模块。我们已经介绍了，使用 AMD 有许多优势，RequireJS 同样也是。
- 模型、视图、控制器和路由需要使用 AMD 格式进行封装。这样允许 Backbone 应用程序的每个组件都能用同样的方式整洁地管理依赖（例如，视图需要的集合），也包括 AMD 允许的非 Backbone 模块。

- 非 Backbone 组件/模块（实用助手或应用程序助手）也可以使用 AMD 进行封装。我鼓励大家尝试使用这种方式开发这些模块，因为这样，Backbone 代码就可以独立进行使用和测试，增加了其可重用性。

介绍完基础知识，让我们来看看应用程序的开发。作为参考，应用程序的结构如下：

```
index.html
...js/
    main.js
    .../models
            todo.js
    .../views
            app.js
            todos.js
    .../collections
            todos.js
    .../templates
            stats.html
            todos.html
    ../libs
        .../backbone
        .../jquery
        .../underscore
        .../require
                require.js
                text.js
...css/
```

9.2 HTML 代码

应用程序的 HTML 代码相对简单，由 3 个主要部分构成：用于输入新 todo 项的输入部分（create-todo）；显示现有项目的列表部分（todo-list），编辑工作也在这里进行；最后还有多少 todo 项需要完成的统计部分（todo-stats）。

```
<div id="todoapp">

      <div class="content">

        <div id="create-todo">
          <input id="new-todo" placeholder="What needs to be done?"
           type="text" />
          <span class="ui-tooltip-top">Press Enter to save this task</span>
        </div>

        <div id="todos">
          <ul id="todo-list"></ul>
        </div>
```

```
        <div id="todo-stats"></div>
      </div>

</div>
```

该教程剩余部分的重点将放在实用的JavaScript方面。

9.3 配置选项

如果已经阅读过前面的 AMD 章节，大家可能已经注意到，显式定义 Backbone 模块（视图、集合或其他模块）的每个依赖可能有点儿乏味。但是，这个是可以改进的。

为了简化引用应用程序中模块可能使用的通用路径，我们使用 RequireJS 配置对象，其通常作为一个顶级脚本文件进行定义。配置对象有很多有用的功能，最有用的是“名称映射”。名称映射是一个基础的键/值（key/value）对，其中键（key）定义的是要用的别名，而值（value）则表示的是真实路径。

在下面的示例代码中（main.js），你可以看到一些常见的名称映射的典型例子，包括 backbone、underscore、jquery 以及其他我们选择的内容，RequireJS 的 text 插件用于协助加载像模板这样的文本资源。

```
require.config({
  baseUrl:'../',
  paths: {
    jquery: 'libs/jquery/jquery-min',
    underscore: 'libs/underscore/underscore-min',
    backbone: 'libs/backbone/backbone-optamd3-min',
    text: 'libs/require/text'
  }
});

require(['views/app'], function(AppView){
  var app_view = new AppView;
});
```

main.js 文件底部的 require()就在这里，以便可以加载和实例化应用程序的主要视图（views/app.js）。在一个项目里，我们会经常看到 require()和配置对象保存在顶级脚本文件里。

除了提供名称映射外，配置对象也可以用于定义附加属性，如 waitSeconds（脚本加载之前要等待的秒数）和 locale（如果要为自定义语言加载 i18n 包）。baseUrl 仅仅是用于模块查找的路径。

欲获得更多有关配置对象的信息，请查看 RequireJS 文档（http://requirejs.org/docs/api.html#config）。

9.4 模块化模型、视图、集合

在深入了解 Backbone 组件的 AMD 包装版之前，先来回顾一下一个非 AMD 视图的示例。下面的视图，监听其模型（todo 项）的改变，并在用户改变模型时将视图进行重新渲染。

```
var TodoView = Backbone.View.extend({

    //... is a list tag.
    tagName:  'li',

    // Cache the template function for a single item.
    template: _.template($('#item-template').html()),

    // The DOM events specific to an item.
    events: {
      'click .check'              : 'toggleDone',
      'dblclick div.todo-content' : 'edit',
      'click span.todo-destroy'   : 'clear',
      'keypress .todo-input'      : 'updateOnEnter'
    },

    // The TodoView listens for changes to its model, rerendering. Since there's
    // a one-to-one correspondence between a **Todo** and a **TodoView** in this
    // app, we set a direct reference on the model for convenience.
    initialize: function() {
      this.model.on('change', this.render, this);
      this.model.view = this;
    },
    ...
```

注意，我们使用模板的惯用做法是引用一个带有 id（或其他选择器）的 script 脚本，并获取它的值。当然，这需要被访问的模板隐式定义在 HTML 代码里。下面的代码是我们刚刚引用模板的嵌入式版本：

```
<script type="text/template" id="item-template">
      <div class="todo <%= done ? 'done' : '' %>">
        <div class="display">
          <input class="check" type="checkbox" <%= done ?
           'checked="checked"' : '' %> />
          <div class="todo-content"></div>
          <span class="todo-destroy"></span>
        </div>
        <div class="edit">
          <input class="todo-input" type="text" value="" />
```

```
        </div>
      </div>
</script>
```

尽管模板本身并没有什么问题，一旦需要使用多个模板开发大型应用程序时，所有模板包含在 HTML 代码里进行页面加载，将会使得加载速度变得难以管理，并对性能造成负面影响。我们马上来解决这个问题。

现在，让我们看看视图的 AMD 版本（views/todo.js）。正如前面所讨论的，模块的包装是使用 AMD 的 define()进行的，其允许我们指定视图需要的依赖。使用映射过的路径来简化常见依赖的引用，以及依赖项实例自我映射到我们可以访问的本地变量（例如，jquery 映射到$上）。

```
define([
  'jquery',
  'underscore',
  'backbone',
  'text!templates/todos.html'
  ], function($, _, Backbone, todosTemplate){
  var TodoView = Backbone.View.extend({

    //... is a list tag.
    tagName:  'li',

    // Cache the template function for a single item.
    template: _.template(todosTemplate),

    // The DOM events specific to an item.
    events: {
      'click .check'              : 'toggleDone',
      'dblclick div.todo-content' : 'edit',
      'click span.todo-destroy'   : 'clear',
      'keypress .todo-input'      : 'updateOnEnter'
    },

    // The TodoView listens for changes to its model, rerendering. Since there's
    // a one-to-one correspondence between a **Todo** and a **TodoView** in this
    // app, we set a direct reference on the model for convenience.
    initialize: function() {
      this.model.on('change', this.render, this);
      this.model.view = this;
    },

    // Rerender the contents of the todo item.
    render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      this.setContent();
      return this;
    },
```

```
    // Use `jQuery.text` to set the contents of the todo item.
    setContent: function() {
      var content = this.model.get('content');
      this.$('.todo-content').text(content);
      this.input = this.$('.todo-input');
      this.input.on('blur', this.close);
      this.input.val(content);
    },
    ...
```

从维护的角度来看，在当前视图的版本里，除了模板的处理方式不同以外，没有什么逻辑上的不同。

使用 RequireJS 的 text 插件（标记为 text 的依赖），实际上可以存储之前外部文件模板的所有内容（templates/todos.html）。

```
<div class="todo <%= done ? 'done' : '' %>">
    <div class="display">
      <input class="check" type="checkbox" <%= done ?
       'checked="checked"' : '' %> />
      <div class="todo-content"></div>
      <span class="todo-destroy"></span>
    </div>
    <div class="edit">
      <input class="todo-input" type="text" value="" />
    </div>
</div>
```

我们不再需要关心模板的 id，因为我们可以将其内容映射到一个局部变量上（本例中是 todosTemplate）。然后，简单地将其传入到 Underscore.js 的_.template()函数，用同样的方式获取该模板脚本的值。

接下来，让我们看看如何将模型定义为依赖项，以便填充到集合。下面的代码（models/todo.js）是一个兼容 AMD 的模型模块，它有两个默认值：表示 todo 项内容的 content 属性和一个能让我们触发该项是否已经完成的布尔属性 done。

```
define(['underscore', 'backbone'], function(_, Backbone) {
  var TodoModel = Backbone.Model.extend({

    // Default attributes for the todo.
    defaults: {
      // Ensure that each todo created has `content`.
      content: 'empty todo...',
      done: false
    },

    initialize: function() {
    },
```

```
    // Toggle the `done` state of this todo item.
    toggle: function() {
      this.save({done: !this.get('done')});
    },

    // Remove this Todo from *localStorage* and delete its view.
    clear: function() {
      this.destroy();
      this.view.remove();
    }

  });
  return TodoModel;
});
```

其他类型的依赖可以很容易将该模型模块映射到一个本地变量上（本例中是Todo），因此它可以在 TodosCollection 里作为模型进行引用。该集合，（collections/todos.js）还支持一个简单的 done()过滤器，用于过滤已经完成的 todo 项；以及一个 remaining()过滤器，用于过滤还未完成的 todo 项。

```
define([
  'underscore',
  'backbone',
  'libs/backbone/localstorage',
  'models/todo'
  ], function(_, Backbone, Store, Todo){

    var TodosCollection = Backbone.Collection.extend({

    // Reference to this collection's model.
    model: Todo,

    // Save all of the todo items under the `todos` namespace.
    localStorage: new Store('todos'),

    // Filter down the list of all todo items that are finished.
    done: function() {
      return this.filter(function(todo){ return todo.get('done'); });
    },

    // Filter down the list to only todo items that are still not finished.
    remaining: function() {
      return this.without.apply(this, this.done());
    },
    ...
```

除了允许用户从视图上添加新的 todo 项外（然后将其作为模型插入到集合中），还希望也能显示多少项已经完成了，又剩下多少。为此，前面集合中已经定义的过滤器可以为我们提供这些信息，让我们在主应用程序视图（views/app.js）中使用它们。

```
define([
  'jquery',
  'underscore',
  'backbone',
  'collections/todos',
  'views/todo',
  'text!templates/stats.html'
  ], function($, _, Backbone, Todos, TodoView, statsTemplate){

  var AppView = Backbone.View.extend({

    // Instead of generating a new element, bind to the existing skeleton of
    // the app already present in the HTML.
    el: $('#todoapp'),

    // Our template for the line of statistics at the bottom of the app.
    statsTemplate: _.template(statsTemplate),

    // ...events, initialize() etc. can be seen in the complete file

    // Rerendering the app just means refreshing the statistics-the rest
    // of the app doesn't change.
    render: function() {
      var done = Todos.done().length;
      this.$('#todo-stats').html(this.statsTemplate({
        total:      Todos.length,
        done:       Todos.done().length,
        remaining:  Todos.remaining().length
      }));
    },
    ...
```

在此，我们为该项目映射第二个模板（templates/stats.html）到 statsTemplate 上，用于显示整个 done 和 remaining 状态。只是简单地将 Todos 集合的长度(Todos.length——目前为止所创建的 todo 项）、已完成项的长度（Todos.done().length）、剩余未完成项的长度（Todos.remaining().length）一起传递给模板即可。

下面的代码是 statsTemplate 的内容。不是太复杂，但它为了显示特定的状态而使用了三目表达式，来判断应该显示一项还是两项。

```
<% if (total) { %>
        <span class="todo-count">
          <span class="number"><%= remaining %></span>
          <span class="word"><%= remaining == 1 ? 'item' : 'items' %>
          </span> left.
        </span>
      <% } %>
      <% if (done) { %>
        <span class="todo-clear">
          <a href="#">
```

```
            Clear <span class="number-done"><%= done %></span>
            completed <span class="word-done"><%= done == 1 ?
            'item' : 'items' %></span>
          </a>
        </span>
      <% } %>
```

Todo 应用的其余来源主要由用来处理用户事件和应用程序事件的代码组成，但都已经囊括了 AMD 的大多数核心概念。

要了解所有的代码是怎么连贯在一起的，可以随时通过复制存储库来获取代码，或者在线浏览（https://github.com/addyosmani/backbone-fundamentals/tree/master/ practicals/modular-todo-app）进行了解。希望对你有所帮助。

9.5 基于路由的模块加载

本节将讨论 Kevin Decker 在 Lumbar（http://walmartlabs.github.com/lumbar）中实现的一种基于路由的模块加载方法。和 RequireJS 一样，Lumbar 也是一个模块化的构建系统，但其加载路由的实现模式可以用于任何构建系统。

Lumbar 构建工具的详情本书就不做讨论了。使用加载器和构建系统、基于 Lumbar 的完整项目，请访问 Thorax（http://thoraxjs.org/），该项目为包括 Lumbar 在内的各种环境提供了示例项目。

9.5.1 基于 JSON 的模块配置

RequireJS 将每个依赖定义成一个文件，而 Lumbar 是在一个集中的 JSON 配置文件里为每个模块定义一个文件列表，并为每个所定义的模块生成一个单独的 JavaScript 文件。Lumbar 要求每个模块（除了基础模块）都要定义一个路由（router）和一个路由线路列表（routes）。示例文件可能如下所示：

```
{
   "modules": {
       "base": {
           "scripts": [
               "js/lib/underscore.js",
               "js/lib/backbone.js",
               "etc"

           ]
       },
       "pages": {
           "scripts": [
               "js/routers/pages.js",
               "js/views/pages/index.js",
```

```
                "etc"
            ],
            "routes": {
                "": "index",
                "contact": "contact"
            }
        }
    }
}
```

一个模块中定义的每个 JavaScript 文件在包含 name 和 routes 的作用域内都有一个 module 对象。在 js/routers/pages.js 里，可以像下面这样，为页面模块定义一个 Backbone 路由：

```
new (Backbone.Router.extend({
    routes: module.routes,
    index: function() {},
    contact: function() {}
}));
```

9.5.2 模块加载器

Backbone.Router 一个很少使用的特性是它可以为同一组路由线路（routes）创建多个路由（routers）。Lumbar 利用该特性创建了一个路由监听应用程序的所有路由路线。当一个路由路线匹配时，主路由检测所需要的模块是否已经加载。如果已经加载，主路由器不采取任何行动，该模块定义的路由将负责继续处理该路由线路。如果所需的模块尚未加载，则加载它，此后 Backbone.history.loadUrl 被调用。该调用将重新加载当前路由线路，以便让主路由不采取进一步行动，并提示新加载模块里定义的路由进行响应。

下面提供的是一个示例实现。config 对象需要包含前面提到的 JSON 配置示例文件里的数据，loader 对象需要实现 isLoaded 和 loadModule 方法。注意，Lumbar 提供了所有这些实现，这些例子将会帮助大家创建自己的实现。

```
// Create an object that will be used as the prototype
// for our master router
var handlers = {
    routes: {}
};

_.each(config.modules, function(module, moduleName) {
    if (module.routes) {
        // Generate a loading callback for the module
        var callbackName = "loader_" moduleName;
        handlers[callbackName] = function() {
            if (loader.isLoaded(moduleName)) {
                // Do nothing if the module is loaded
                return;
```

```
            } else {
                //the module needs to be loaded
                loader.loadModule(moduleName, function() {
                    // Module is loaded, reloading the route
                    // will trigger callback in the module's
                    // router
                    Backbone.history.loadUrl();
                });
            }
        };
        // Each route in the module should trigger the
        // loading callback
        _.each(module.routes, function(methodName, route) {
            handlers.routes[route] = callbackName;
        });
    }
});

// Create the master router
new (Backbone.Router.extend(handlers));
```

9.5.3 使用 NodeJS 处理 pushState

window.history.pushState 支持（Backbone 路由不需要哈希支持）要求服务器知道 Backbone 应用程序需要处理什么 URL，是因为用户可以通过输入任何路由（或者导航到一个 pushStateURL 后进行刷新）来进入应用程序。

在单独的位置定义所有路由的另一个优势是服务器可以加载同一个 JSON 配置文件（前面提供的），然后监听每个路由。使用 Node.js 和 Express 实现的一个示例如下所示：

```
var fs = require('fs'),
    _ = require('underscore'),
    express = require('express'),
    server = express(),
    config = JSON.parse(fs.readFileSync('path/to/config.json'));

_.each(config.modules, function(module, moduleName) {
    if (module.routes) {
        _.each(module.routes, function(methodName, route) {
            server.get(route, function(req, res) {
                  res.sendFile('public/index.html');
            });
        });
    }
});
```

上述实现假设 Backbone 应用程序支持 index.html 访问。只要指定一个 root 选项，Backbone.History 对象就可以处理其余的路由逻辑。在网站根部使用的简单应用程

序示例配置，看起来可能像这样：

```
Backbone.history || (Backbone.history = new Backbone.History());
Backbone.history.start({
  pushState: true,
  root: '/'
});
```

9.6 另外一种依赖管理方式

对于烦琐的视图，DocumentCloud 有一个自制的资产包装器叫作 Jammit（https://github .com/documentcloud/jammit），它可以很容易地与 Underscore.js 进行集成，并且也可用于依赖管理。

Jammit 希望通过.jst 文件的形式，让 JavaScript 模板（JST）可以与任何所使用的 ERB 模板一起使用。Jammit 将模板封装到一个可以将模板渲染成字符的全局 JST 对象。让 Jammit 识别模板是很简单的——在 assets.yml 里添加类似“views/**/*.jst”这样的条目进去即可。

提供 Jammit 依赖管理功能，简单编写一个 assets.yml 文件即可，在文件里，可以按顺序列出所需要的依赖，或者使用通用目录组合（例如，//.js、templates/.js 以及特定文件）。

使用 Jammit 的模板可以从传入的集合中获取渲染后的数据：

```
this.$el.html(JST.myTemplate({ collection: this.collection }));
```

第 10 章

对 Backbone.js 请求和集合进行分页

分页是一个无处不在的问题，我们经常要在 Web 项目上自己解决这个问题——尤其是使用 API 服务以及调用它的 JavaScript 客户端进行编程时。由于大多数人都认为分页相对容易，这通常也是一个未确定的问题。然而，并不总是这样，因为分页往往会变得比最初看起来更加棘手。

在深入探索 Backbone 应用程序数据分页的解决方案之前，先来定义一下我们所认为的分页到底有哪些内容。

分页是一个控制系统，它允许用户通过搜索结果页面进行浏览或者继续浏览任何类型的数据。搜索结果是一个规范化的例子，但是如今在新闻网站、博客以及论坛上发现的分页，通常都是以上一页（Previous）和下一页（Next）链接的形式存在的。完整的分页系统可以提供细粒度的控制，以便可以导航到具体的页面，给用户更多的权力来查找他们想查看的东西。

在页面上限制分页的一些视觉控件也不是什么问题——像 Facebook、Pinterest 和 Twitter 这样的网站已经证明很多无限分页的语义也是很有用的。当然，无限分页就是当我们预先（appear to prefetch）从随后的页面获取内容，然后添加到用户的当前页面时，让人感到体验是无限的。

分页非常特定于上下文，并取决于要显示的内容。在谷歌搜索结果中，分页之所以重要，是因为谷歌要在第一页或者第二页上提供最相关的搜索结果。在那之后，我们可能在选择（或随机选择）访问页面方面更具有选择性。这和我们想通过连续页面显示记录的情况不太一样，例如新闻文章或者博客帖子。

分页几乎肯定是与内容和上下文相关的，但 Faruk Ates 曾指出（https://gist.

github.com/mislav/622561），良好的分页原则应用是不关注内容和上下文的。由于使用 Backbone 后的所有东西都可扩展，我们可以编写自己的分页来解决这些特定内容类型的分页问题。也就是说，我们在这上面可能要花很长时间，有时我们可能只想用一个有效而可靠的解决方案。

关于这个话题，我们要仔细介绍一下我和其他一群贡献者（贡献者列表：https://github.com/addyosmani/backbone.paginator/contributors）为 Backbone.js 编写的各种分页组件。如果你的应用程序需要对 Backbone 集合进行分页，也许就能派上用场。这些组件是一个名为 Backbone.Paginator（http://github.com/addyosmani/backbone.paginator）扩展的一部分。

10.1 Backbone.Paginator

在客户端处理数据时，我们最有可能遇到 3 种类型的分页。

1. 请求服务层（API）

例如，查询包含术语 Paul 的结果——如果有 5000 个结果可用，每页只显示 20 个（并显示可以导航的 250 个结果页面）。

这个问题实际上有很大的信息量，如维护其他 URL 参数的持久性（比如 sort、query、order 等条件），这些参数是可以根据用户界面上的用户搜索配置进行改变的。也需要一个整洁的方式将分页放到视图里，这样就可以轻松地在页面之间进行导航（例如，第一页、最后一页、上一页、下一页、1、2、3），并在每个页面上管理显示的特定数量的搜索结果，等等。

2. 对返回数据进行进一步的客户端分页

例如，已经返回了一个包含 100 个结果的 JSON 响应，我们只能在浏览器的可导航 UI 中显示 20 个结果，而不是给用户显示全部 100 个结果。

类似于请求问题，客户端分页有它自身的挑战，例如再次导航（下一页、上一页、1、2、3）、排序、顺序、每个页面上轮流显示相应的结果等。

3. 无限结果

像 Facebook 这样的服务，数字分页是被替换成了加载更多（Load More）或查看更多（View More）这样的按钮。触发这些按钮，通常会获取下一个页面的 *N* 个结果，然后将其附加到当前页面，而不是完全替换之前加载的结果集。

一个请求分页页面只是将结果内容附加到其视图里，而不是在每次获取的时候都进行替换，这种页面实际上是一个无限的分页页面。

现在让我们来看看我们所说的内容到底是什么。

Backbone.Paginator，如图 10-1 所示，是一个使用 Backbone.js 对数据集合进行分页的组件集。它的目标是提供两种解决方案，用于协助服务器分页请求（比如请求 API），以及对单个加载数据进行分页——这里我们可以将希望进一步集合的 N 个结果分成 M 个页面。

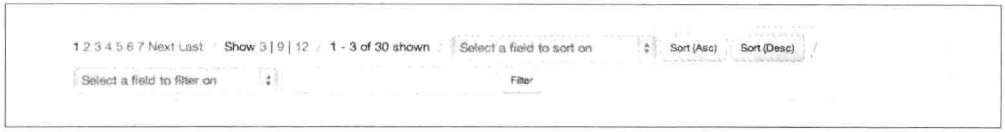


图 10-1　Backbone.Paginator 演示如何在项目里进行分页显示

Backbone.Paginator 支持两种主要分页组件：

- **Backbone.Paginator.requestPager**：对客户端和服务器端 API 之间的请求进行分页。
- **Backbone.Paginator.clientPager**：对于服务器返回的数据，如果想在 UI 上进一步分页的这种情况进行分页（例如，返回 60 个结果，分成 3 页，每页显示 20 个）。

要想查看使用这些组件构建项目的示例，下面是官方演示的链接，这些演示使用了 Netflix API，以便用户可以看到这些组件和真实数据源一起使用的结果。

- Backbone.Paginator.requestPager()：http://addyosmani.github.com/backbone.paginator/examples/netflix-request-paging/index.html。
- Backbone.Paginator.clientPager()：http://addyosmani.github.com/backbone.paginator/examples/netflix-client-paging/index.html。
- 无限分页（Backbone.Paginator.requestPager()）：http://addyosmani.github.com/backbone.paginator/examples/netflix-infinite-paging/index.html。
- Diacritic 插件：http://addyosmani.github.com/backbone.paginator/examples/google-diacritic/index.html。

10.2　Paginator.requestPager

在本节，我们将通过使用 requestPager（见图 10-2）过一遍操作。如果使用自身就

能分页的 API 服务，就可以使用这个组件。该组件允许用户通过客户端控制 API 请求的分页设置（例如，导航到下一页、上一页、第 N 页）。

该思想是：分页、搜索以及数据过滤，都可以在 Backbone 应用程序中完成，而不需要重新加载页面。

图 10-2　使用 requestPager 组件从 Netflix API 上请求分页结果

（1）创建新分页集合。

首先，像下面这样，使用 Backbone.Paginator.requestPager()定义一个新的分页集合：

```
var PaginatedCollection = Backbone.Paginator.requestPager.extend({
```

（2）像往常一样为集合设置模型。

在集合中，像往常一样指定集合中使用的模型，该集合提供的 URL（或基础 URL）用于提供数据（例如 Netflix API）。

```
model: model,
```

（3）配置基础 URL 和请求类型。

需要设定一个基础 URL。默认情况下请求的类型是 GET，数据类型设置为 jsonp 是为了能够跨域请求。

```
paginator_core: {
  // the type of the request (GET by default)
  type: 'GET',

  // the type of reply (jsonp by default)
  dataType: 'jsonp',

  // the URL (or base URL) for the service
  // if you want to have a more dynamic URL, you can make this
  // a function that returns a string
  url: 'http://odata.netflix.com/Catalog/People(49446)/TitlesActedIn?'
},
```

如果数据类型使用的不是 jsonp，请在 server_api 配置中移除自定义 callback 参数。

(4) 配置如何显示结果。

我们需要告诉该库每个页面需要显示多少项、当前页是第几页、页面范围是什么等。

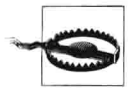
```
paginator_ui: {
  // the lowest page index your API allows to be accessed
  firstPage: 0,

  // which page should the paginator start from
  // (also, the actual page the paginator is on)
  currentPage: 0,

  // how many items per page should be shown
  perPage: 3,

  // a default number of total pages to query in case the API or
  // service you are using does not support providing the total
  // number of pages for us.
  // 10 as a default in case your service doesn't return the total
  totalPages: 10
},
```

(5) 配置发往服务器的参数。

大多数情况下，基础 URL 是不够用的，所以可以向服务器传递更多的参数。注意如何使用函数而不是硬编码值，也可以引用 paginator_ui 中指定的值。

```
server_api: {
  // the query field in the request
  '$filter': '',

  // number of items to return per request/page
  '$top': function() { return this.perPage },

  // how many results the request should skip ahead to
  // customize as needed. For the Netflix API, skipping ahead based on
```

```
    // page * number of results per page was necessary.
    '$skip': function() { return this.currentPage * this.perPage },
    // field to sort by
    '$orderby': 'ReleaseYear',

    // what format would you like to request results in?
    '$format': 'json',

    // custom parameters
    '$inlinecount': 'allpages',
    '$callback': 'callback'
},
```



如果使用$callback，要确保在 paginator_core 配置中使用 jsonp 作为数据类型。

（6）配置 Collection.parse()。

最后一件要做的事就是配置集合的 parse()方法。我们要确保返回 JSON 响应的正确部分，其包含即将要填充集合的数据。下面的示例中是 response.d.results（以 Netflix API 为例）。

```
        parse: function (response) {
            // Be sure to change this based on how your results
            // are structured (e.g., d.results is Netflix-specific)
            var tags = response.d.results;
            //Normally this.totalPages would equal response.d.__count
            //but as this particular NetFlix request only returns a
            //total count of items for the search, we divide.
            this.totalPages = Math.ceil(response.d.__count / this.perPage);
            return tags;
        }
    });

});
```

大家可能还会注意到，我们将 this.totalPages 设置成了 API 返回的页面总数。该数字允许我们为当前或下一个请求定义页面的最大数，这样可以清晰地将其显示在 UI 上。它也能影响是否要单击事件，比如，判断“下一页”按钮是否应处理请求。

为了方便，可以在视图中使用下面的方法与 requestPager 进行交互：

- **Collection.goTo(*n, options*)**：跳转到指定的页面。
- **Collection.nextPage(*options*)**：跳转到下一页。
- **Collection.prevPage(*options*)**：跳转到上一页。

- **Collection.howManyPer(*n*)**：设置每页显示多少记录。

requestPager 集合中的方法.goTo()、.nextPage()、以及 prevPage()都是 Backbone 原始方法 Collection.fetch()的扩展，因此，它们都可以采取相同的选项对象作为参数。

option 对象在服务器响应以后，可以使用 success 和 error 参数传递函数并执行。

```
Collection.goTo(n, {
  success: function( collection, response ) {
    // called if server request success
  },
  error: function( collection, response ) {
    // called if server request fail
  }
});
```

要管理回调，也可以使用这些方法返回的 jqXHR。

```
Collection
  .requestNextPage()
  .done(function( data, textStatus, jqXHR ) {
    // called if server request success
  })
  .fail(function( data, textStatus, jqXHR ) {
    // called if server request fail
  })
  .always(function( data, textStatus, jqXHR ) {
    // do something after server request is complete
  });
});
```

如果要向当前集合添加新模型，而不是替换收集的内容，可以将{update: true, remove: false}作为 option 选项传递给这些方法。

```
Collection.prevPage({ update: true, remove: false });
```

10.3 Paginator.clientPager

clientPager（见图 10-3）用于将服务器 API 返回的数据进行进一步分页。比如从服务器请求到 100 个记录，并且想分成 5 页，在客户端层面上每页包含 20 个记录——clientPager 可以很容易做到这点。

如果更愿意在单个加载中使用 clientPager 进行分页，可以在用户获取下一页数据时避免额外的网络请求。由于把所有的结果都一次请求到了，它只是在数据范围内进行切换并呈现给用户。

图 10-3　使用 clientPager 组件将 Netflix API 返回的数据进一步分页

（1）创建带有模型和 URL 的分页集合。

在 requestPager 里，首先创建一个带有模型的 Backbone.Paginator.clientPager 分页集合：

```
var PaginatedCollection = Backbone.Paginator.clientPager.extend({

    model: model,
```

（2）配置基础 URL 和请求类型。

需要设定一个基础 URL。默认情况下请求的类型是 GET，数据类型设置为 jsonp 是为了能够跨域请求。

```
paginator_core: {
  // the type of the request (GET by default)
  type: 'GET',

  // the type of reply (jsonp by default)
  dataType: 'jsonp',

  // the URL (or base URL) for the service
  url: 'http://odata.netflix.com/v2/Catalog/Titles?&'
},
```

（3）配置如何显示结果。

我们需要告诉该库每个页面需要显示多少项、当前页是第几页、页面范围是什么等。

```
paginator_ui: {
  // the lowest page index your API allows to be accessed
  firstPage: 1,

  // which page should the paginator start from
  // (also, the actual page the paginator is on)
  currentPage: 1,

  // how many items per page should be shown
  perPage: 3,

  // a default number of total pages to query in case the API or
  // service you are using does not support providing the total
  // number of pages for us.
  // 10 as a default in case your service doesn't return the total
  totalPages: 10,

  // The total number of pages to be shown as a pagination
  // list is calculated by (pagesInRange * 2) + 1.
  pagesInRange: 4
},
```

(4) 配置发往服务器的参数。

大多数情况下，基础 URL 是不够用的，所以可以向服务器传递更多的参数。注意如何使用函数而不是硬编码值，也可以引用 paginator_ui 中指定的值。

```
server_api: {
  // the query field in the request
  '$filter': 'substringof(\'america\',Name)',

  // number of items to return per request/page
  '$top': function() { return this.perPage },

  // how many results the request should skip ahead to
  // customize as needed. For the Netflix API, skipping ahead based on
  // page * number of results per page was necessary.
  '$skip': function() { return this.currentPage * this.perPage },

  // field to sort by
  '$orderby': 'ReleaseYear',

  // what format would you like to request results in?
  '$format': 'json',

  // custom parameters
  '$inlinecount': 'allpages',
  '$callback': 'callback'
},
```

(5) 配置 Collection.parse()。

最后就是 parse()方法了，它在本例中并不是服务器上的页面总数，而是 UI 界面中

分页数据自己的页面总数。

```
parse: function (response) {
        var tags = response.d.results;
        return tags;
    }

});
```

10.3.1 便利方法

正如前面提到的，视图可以利用大量便利的方法导航 UI 分页数据。clientPager 包括如下便利方法：

- **Collection.goTo(*n, options*)**：跳转到指定的页面。
- **Collection.prevPage(*options*)**：跳转到上一页。
- **Collection.nextPage(*options*)**：跳转到下一页。
- **Collection.howManyPer(*n*)**：设置每页显示多少记录。
- **Collection.setSort(*sortBy, sortDirection*)**：在当前视图上更新排序。如果要对数字（即使是作为字符存储）进行排序，它可以自动探测到，并准确做事。
- **Collection.setFilter(*filterFields, filterWords*)**：过滤当前视图。该过滤支持多个单词，并且无需特定的顺序，这样就基本上有了全文搜索能力。同样，将模型上的一个属性，或者将多个属性组成一个数组传递进入，也可以得到过滤结果。最后的选择是传递给它一个包含单个比较方法和多个规则的对象。目前，只有 Levenshtein 方法是可用的。Levenshtein 距离就是两个字符串之间的区别，实际上就是将一个词转变成另外一个词的最低变化次数。

goTo()、**prevPage()**以及 **nextPage()**函数不需要 options 参数，因为它们都将同步执行。然而，如果指定了 success 参数，success 回调在函数返回之前就会被调用，例如：

```
nextPage(); // this works just fine!
nextPage({success: function() { }}); // this will call the success function
```

options 参数的存在是为了在 requestPaginator 和 clientPaginator 之间保持（一些）接口的统一，以便它们在 Backbone.Views 中可以互换使用。

```
this.collection.setFilter(
  {'Name': {cmp_method: 'levenshtein', max_distance: 7}}
  , "American P" // Note the switched 'r' and 'e', and the 'P' from 'Pie'
);
```

还要注意的是，Levenshtein 插件应该通过 useLevenshteinPlugin 变量加载并启用。最后还有重要的一点是：执行 Levenshtein 比较将返回两个字符串的距离(distance)，它不会允许我们搜索冗长的文字。两个字符串之间的距离意思是通过添加、删除、左移或右移字符操作来让两个字符相等。这就意味着比较“Something”和“This is a test that could show something”，返回的结果是 32，大于“Something”和“ABCDEFG(9).”的比较结果。我们只能在短文本（标题、名称等）上使用 Levenshtein。

- **Collection.doFakeFilter(*filterFields, filterWords*)**：在 Collection.setFilter 上模拟调用以后返回对应模型的数量。

- **Collection.setFieldFilter(*rules*)**：根据传入参数的规则（rules）过滤每个模型的每个值。假设有一个带有出版时间和出版作者的图书集合，可以只过滤出在 1999 年到 2003 年之间出版的图书，然后再添加另外一个规则有，只过滤出作者姓名以 A 开头的图书。可用的规则有：function、required、min、max、range、minLength、maxLength、rangeLength、one of、equalTo、containsAllOf、pattern。传入空规则将会删除所有已经应用的 FieldFilter。

```
my_collection.setFieldFilter([
  {field: 'release_year', type: 'range', value:
  {min: '1999', max: '2003'}},
  {field: 'author', type: 'pattern', value: new RegExp('A*', 'igm')}
]);

//Rules:
//
//var my_var = 'green';
//
//{field: 'color', type: 'equalTo', value: my_var}
//{field: 'color', type: 'function', value: function(field_value){
  return field_value == my_var; } }
//{field: 'color', type: 'required'}
//{field: 'number_of_colors', type: 'min', value: '2'}
//{field: 'number_of_colors', type: 'max', value: '4'}
//{field: 'number_of_colors', type: 'range', value: {min: '2', max: '4'} }
//{field: 'color_name', type: 'minLength', value: '4'}
//{field: 'color_name', type: 'maxLength', value: '6'}
//{field: 'color_name', type: 'rangeLength', value: {min: '4', max: '6'}}
//{field: 'color_name', type: 'oneOf', value: ['green', 'yellow']}
//{field: 'color_name', type: 'pattern', value: new RegExp('gre*', 'ig')}
//{field: 'color_name', type: 'containsAllOf', value:
  ['green', 'yellow', 'blue']}
```

Collection.doFakeFieldFilter(*rules*)：在 Collection.setFieldFilter 上模拟调用以后返回对应模型的数量。

10.3.2 实现备注

在视图里可以使用如下变量，表示分页的实际状态。

- ***totalUnfilteredRecords***：记录的数量，包括以任何方式进行过滤的所有记录（仅在 clientPager 上可用）。
- ***totalRecords***：记录的数量。
- ***currentPage***：分页所处的当前实际页面。
- ***perPage***：分页里每页要显示的记录数量。
- ***totalPages***：总页数。
- ***startRecord***：当前页面第一条记录的位置——例如，2000 条记录中从第 41 条到第 50 条（仅在 clientPager 上可用）。
- ***endRecord***：当前页面最后一条记录的位置——例如，2000 条记录中从第 41 条到第 50 条（仅在 clientPager 上可用）。
- ***pagesInRange***：当前页面前后两侧每侧要显示页面的数量。所以，如果 pagesInRange 是 3，并且当前页是 13，会得到这些结果：10、11、12、13（当前页）、14、15、16。

```
<!-- sample template for pagination UI -->
<script type="text/html" id="tmpServerPagination">

  <div class="row-fluid">

    <div class="pagination span8">
      <ul>
        <% _.each (pageSet, function (p) { %>
        <% if (currentPage == p) { %>
          <li class="active"><span><%= p %></span></li>
        <% } else { %>
          <li><a href="#" class="page"><%= p %></a></li>
        <% } %>
        <% }); %>
      </ul>
    </div>

    <div class="pagination span4">
      <ul>
        <% if (currentPage > firstPage) { %>
          <li><a href="#" class="serverprevious">Previous</a></li>
        <% }else{ %>
          <li><span>Previous</span></li>
```

```
        <% }%>
        <% if (currentPage < totalPages) { %>
          <li><a href="#" class="servernext">Next</a></li>
        <% } else { %>
          <li><span>Next</span></li>
        <% } %>
        <% if (firstPage != currentPage) { %>
          <li><a href="#" class="serverfirst">First</a></li>
        <% } else { %>
          <li><span>First</span></li>
        <% } %>
        <% if (totalPages != currentPage) { %>
          <li><a href="#" class="serverlast">Last</a></li>
        <% } else { %>
          <li><span>Last</span></li>
        <% } %>
      </ul>
    </div>

  </div>

  <span class="cell serverhowmany"> Show <a href="#"
    class="selected">18</a> | <a href="#" class="">9</a> |
    <a href="#" class="">12</a> per page
  </span>

  <span class="divider">/</span>

  <span class="cell first records">
    Page: <span class="label"><%= currentPage %></span> of
    <span class="label"><%= totalPages %></span> shown
  </span>

</script>
```

10.3.3 插件

Diacritic.js 是 Backbone.Paginator 的一个插件，它可以将标注字符（diacritic characters）（´、˝、°、~等）替换成最接近匹配的字符，如图 10-4 所示。该插件对过滤特别有用。

要启用该插件，将 this.useDiacriticsPlugin 设置为 true，示例如下：

```
// Default values used when sorting and/or filtering.
initialize: function(){
  this.useDiacriticsPlugin = true; // use diacritics plug-in if available
...
```

图 10-4　Diacritics 插件用于和 clientPager 一起正确渲染特殊字符

10.3.4　引导

默认情况下，clientPager 和 requestPager 都会向服务器发送初始化请求，以便填充其内部分页数据。为避免这个额外的请求，大家可能会发现，它将有利于从 DOM 中已经存在的数据中引导 Backbone.Paginator 实例。Backbone.Paginator.clientPager 中的配置代码示例如下。

```
// Extend the Backbone.Paginator.clientPager with your own configuration options
var MyClientPager =  Backbone.Paginator.clientPager.extend({paginator_ui: {}});
// Create an instance of your class and populate with the models of your
// entire collection
var aClientPager = new MyClientPager([{id: 1, title: 'foo'},
{id: 2, title: 'bar'}]);
// Invoke the bootstrap function
aClientPager.bootstrap();
```

如果我们打算使用 clientPager，是不需要在配置里指定 paginator_core 对象的（因为我们已经应该在配置文件里配置了 clientPager）。Backbone.Paginator.requestPager 中的配置代码示例如下：

```
// Extend the Backbone.Paginator.requestPager with your own configuration options
var MyRequestPager =  Backbone.Paginator.requestPager.extend({paginator_ui: {}});
// Create an instance of your class with the first page of data
var aRequestPager = new MyRequestPager([{id: 1, title: 'foo'},
{id: 2, title: 'bar'}]);
// Invoke the bootstrap function and configure requestPager with 'totalRecords'
aRequestPager.bootstrap({totalRecords: 50});
```

clientPager 和 requestPager 的 bootstrap 函数都将接受一个 options 参数，该参数将被 Backbone.Paginator 实例进行扩展，而 totalRecords 属性将由 clientPager 进行隐式设置。

欲获得更多关于Backbone引导的信息，请访问Rico Sta Cruz的网站（http://ricostacruz.com/backbone-patterns/#bootstrapping_data）。

10.3.5 样式化

当然，我们可以按照自己的想法，随意定制分页的整体外观和感官。默认情况下，所有示例程序的链接、按钮以及下拉菜单的样式都是使用 Twitter Bootstrap 来处理的。

可以使用CSS类格式化记录数、过滤条件、排序以及更多内容，如图10-5所示。

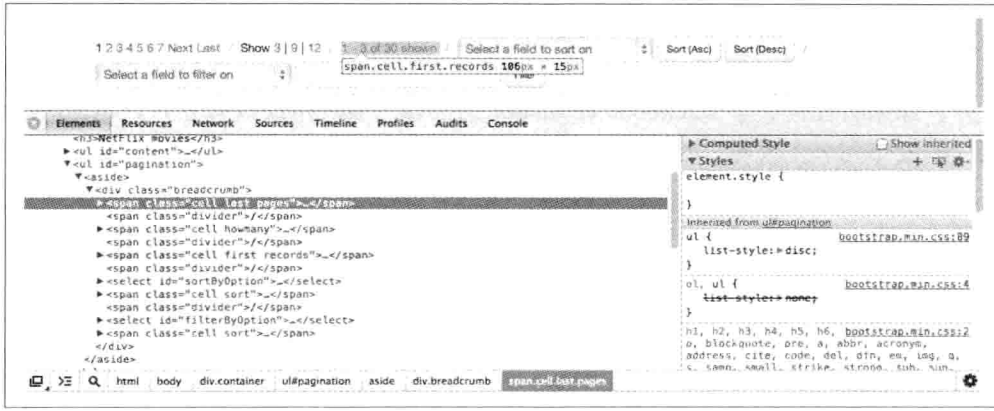


图10-5 使用Chrome DevTool控制台提供的CSS样式属性查看功能检查分页内容

CSS类也可以格式化更细粒度的元素（如带有导航控件>分页数字），例如.page、.page selected样式，如图10-6所示。

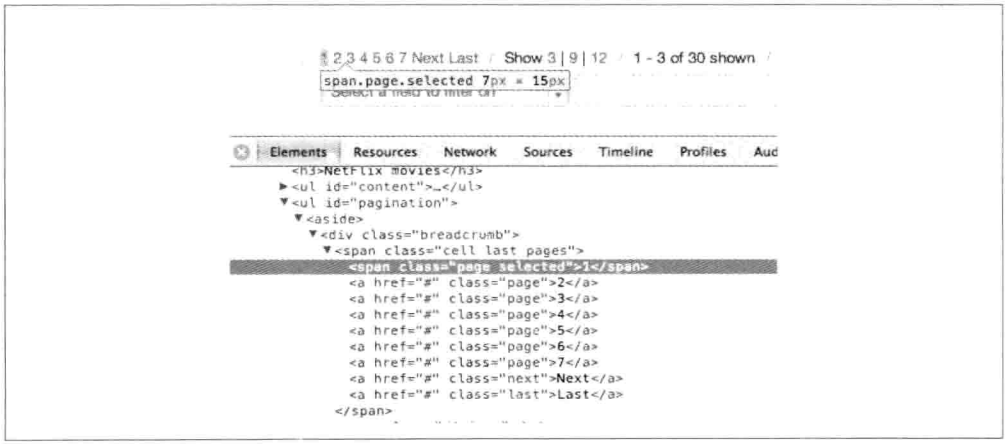


图10-6 演示如何使用Twitter Bootstrap导航控件格式化分页内容

有很多的灵活性用于样式，控制模板时也一样，分页内容可以根据需要在视觉上显示得简单或者复杂。

10.4 总结

尽管使用 Backbone 集合时可以编写自己的自定义分页类，但 Backbone.Paginator 已经为我们做很多工作了。

Backbone.Paginator 是高度可配置的，在处理来自数据库或 API 的数据集合时，我们不必编写自己的分页代码。使用该插件可以将大列表数据转变成更易于管理、易于导航、可分页的列表。

另外，如果有任何关于 Backbone.Paginator 的问题（或想帮助改善它），请随时在项目的发布列表上发帖，地址是：https://github.com/addyosmani/backbone.paginator。

第 11 章

Backbone Boilerplate 和 Grunt-BBB

Boilerplates 为我们进行项目构建提供了一个起点。它们是使用最少代码让功能运行起来的一个构建基础。在编写新的 Backbone 应用程序时，一个新的模型通常只需要几行代码就可以运行。

然而，那可能是不够的，因为我们需要集合来组合这些模型，用视图渲染这些模型，或者为集合数据的特定视图添加一个路由，以便可以书签化。如果开始着手进行一个全新的项目，我们可能还需要一个构建过程，并在这里构建应用程序的优化版本，以便可以发布至生产环境。

这就是样板方案（boilerplate solution）有用的地方。不必为 Backbone 应用程序的每个部分都手动编写初始化代码，样板方案可以为我们做这件事，并且也提供了构建过程。

Backbone Boilerplate（或 BB：https://github.com/tbranyen/backbone-boilerplate/）是用于构建 Backbone.js 应用程序的一个非常好的最佳实践和工具集，是由 Backbone 的贡献者 Tim Branyen 所创建的。他把大量使用 Backbone 构建应用程序时遇到的陷阱、误区和常见的任务，根据其经验，精心制作成了 BB 插件。

Grunt-BBB 或 Boilerplate Build Buddy（https://github.com/backbone-boilerplate/grunt-bbb）是 BB 的配合工具，它提供了基架、文件监视和构建能力。与 BB 一起使用，它能够为快速开始开发新 Backbone 应用程序提供一个良好的基础。如图 11-1 所示是正在命令行中运行的 Grunt-BBB 工具。

图 11-1　Grunt-BBB 创建工具正在命令行中运行

开箱即用、BB 和 Grunt-BBB 提供了如下功能：

- 提供 Backbone、Lo-Dash（Underscore.js 的替代）、jQuery 以及 HTML5 Boilerplate 基础文件。
- 提供 Boilerplate 和基架支持，允许我们用最少的时间为模块、集合等编写代码。
- 提供构建工具，用于模板预编译，将库文件、程序代码以及样式表进行连接并最小化。
- 提供一个轻量级的 Node.js Web 服务器。

Lo-Dash：https://github.com/bestiejs/lodash。

HTML5 Boilerplate：http://html5boilerplate.com/。

构建工具步骤备注：

- 模板预编译：使用像 Underscore 微模板（或 Handlebars.js）这样的模板库，通常需要包含 3 个步骤：（1）读取原始模板，（2）将其编译成 JavaScript 函数，（3）让你希望的数据一起运行编译过的模板。预编译消除了第二步，将其过程从运行时移动到构建过程中。
- 连接是将多个资产（本例中是 script 文件）连接在一起并绑定到几个文件（或一个文件）中的过程，以此来减少 HTTP 请求的次数。

- 最小化（压缩）就是从代码中消除不必要的字符（例如空格、换行、注释），并将其进行压缩，以减少发送脚本的文件大小。

11.1 准备开始

要开始项目，需要先安装 Grunt-BBB，其中包括 Backbone Boilerplate 以及所有可能的第三方依赖，比如 Grunt 构建工具。

可以通过 npm 运行如下命名安装 Grunt-BBB：

```
npm install -g bbb
```

就执行这些，应该可以安装好了。

下面是使用 Grunt-BBB 的一个典型的工作流程，稍后我们将使用：

（1）初始化一个新项目（bbb init）。

（2）添加新模块和模板（bbb init:module）。

（3）使用内置服务器预览变更（bbb server）。

（4）运行构建工具（bbb build）。

（5）使用 r.js 链接 JavaScript、编译模板、构建应用程序，并最小化 CSS 和 JavaScript（bbb release）。

11.2 创建新项目

让我们为新项目创建一个新的目录，运行 bbb init 即可。一批项目子目录和文件就创建出来了，如下所示：

```
$ bbb init
Running "init" task
This task will create one or more files in the current directory, based on the
environment and the answers to a few questions. Note that answering "?" to any
question will show question-specific help and answering "none" to most questions
will leave its value blank.

"bbb" template notes:
This tool will help you install, configure, build, and maintain your Backbone
Boilerplate project.
Writing app/app.js...OK
Writing app/config.js...OK
```

```
Writing app/main.js...OK
Writing app/router.js...OK
Writing app/styles/index.css...OK
Writing favicon.ico...OK
Writing grunt.js...OK
Writing index.html...OK
Writing package.json...OK
Writing readme.md...OK
Writing test/jasmine/index.html...OK
Writing test/jasmine/spec/example.js...OK
Writing test/jasmine/vendor/jasmine-html.js...OK
Writing test/jasmine/vendor/jasmine.css...OK
Writing test/jasmine/vendor/jasmine.js...OK
Writing test/jasmine/vendor/jasmine_favicon.png...OK
Writing test/jasmine/vendor/MIT.LICENSE...OK
Writing test/qunit/index.html...OK
Writing test/qunit/tests/example.js...OK
Writing test/qunit/vendor/qunit.css...OK
Writing test/qunit/vendor/qunit.js...OK
Writing vendor/h5bp/css/main.css...OK
Writing vendor/h5bp/css/normalize.css...OK
Writing vendor/jam/backbone/backbone.js...OK
Writing vendor/jam/backbone/package.json...OK
Writing vendor/jam/backbone.layoutmanager/backbone.layoutmanager.js...OK
Writing vendor/jam/backbone.layoutmanager/package.json...OK
Writing vendor/jam/jquery/jquery.js...OK
Writing vendor/jam/jquery/package.json...OK
Writing vendor/jam/lodash/lodash.js...OK
Writing vendor/jam/lodash/lodash.min.js...OK
Writing vendor/jam/lodash/lodash.underscore.min.js...OK
Writing vendor/jam/lodash/package.json...OK
Writing vendor/jam/require.config.js...OK
Writing vendor/jam/require.js...OK
Writing vendor/js/libs/almond.js...OK
Writing vendor/js/libs/require.js...OK

Initialized from template "bbb".

Done, without errors.
```

让我们来看看都生成了什么内容。

11.2.1 index.html

除了页面底部的 RequireJS 引用之外，这是一个非常标准的精简 HTML5 样板基础。

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <meta name="viewport" content="width=device-width,initial-scale=1">

  <title>Backbone Boilerplate</title>
```

```
    <!-- Application styles. -->
    <!--(if target dummy)><!-->
    <link rel="stylesheet" href="/app/styles/index.css">
    <!--<!(endif)-->
  </head>
  <body>
    <!-- Application container. -->
    <main role="main" id="main"></main>

    <!-- Application source. -->
    <!--(if target dummy)><!-->
    <script data-main="/app/config" src="/vendor/js/libs/require.js"></script>
    <!--<!(endif)-->

  </body>
  </html>
```

RequireJS（AMD 模块和脚本加载器）将在应用程序中对模块进行协助管理。第 10 章已经讲过了，但让我们再回顾一下，在样板文件里的这个特殊代码块：

```
<script data-main="/app/config" src="/vendor/js/libs/require.js"></script>
```

data-main 属性是用来告诉 RequireJS，在加载自身之后，开始加载 app/config.js（一个配置对象）。可能你会注意到，这里省略了.js 扩展名，是因为 RequireJS 可以自动进行添加。但是，如果非要包含扩展名，RequireJS 也照样能处理。下面来看一下被引用的 config 文件。

11.2.2 config.js

RequireJS 配置对象允许我们为常用的依赖项（如 jQuery）指定别名和路径，定义引导属性（如应用程序基础 URL），并对没有原生支持 AMD 的库进行映射。config 文件在 Backbone Boilerplate 中的配置类似如下：

```
// Set the require.js configuration for your application.
require.config({

  // Initialize the application with the main application file and the JamJS
  // generated configuration file.
  deps: ["../vendor/jam/require.config", "main"],

  paths: {
    // Put paths here.
  },

  shim: {
    // Put shims here.
  }

});
```

上述配置定义的第一个选项是 deps: ["../vendor/jam/require.config", "main"]。它告诉 RequireJS 加载额外的 RequireJS 配置文件以及 main.js 文件，这是应用程序的入口点。

大家可能会注意到，我们并没有为 main 指定任何路径信息。RequireJS 需要使用 index.html 里的 data-main 属性推断出默认的 baseUrl。换句话说，baseUrl 就是 app/，任何需要加载的脚本路径都将相对于这个位置。如果要使用不同的位置，也可以使用 baseUrl 这个选项来覆盖该默认值。

第二块是paths，我们可以使用它为经常引用的依赖项指定相对于baseUrl的路径（以及依赖路径/依赖别名）。

在这之后是 shim，RequireJS 配置的一个很重要的部分，允许我们加载不兼容 AMD 格式的 JS 库。这里的基本理念是：不用要求所有的 JS 库都支持 AMD，shim 负责帮我们解决这部分最难的工作。

回到deps，require.config文件的内容如下：

```
var jam = {
    "packages": [
        {
            "name": "backbone",
            "location": "../vendor/jam/backbone",
            "main": "backbone.js"
        },
        {
            "name": "backbone.layoutmanager",
            "location": "../vendor/jam/backbone.layoutmanager",
            "main": "backbone.layoutmanager.js"
        },
        {
            "name": "jquery",
            "location": "../vendor/jam/jquery",
            "main": "jquery.js"
        },
        {
            "name": "lodash",
            "location": "../vendor/jam/lodash",
            "main": "./lodash.js"
        }
    ],
    "version": "0.2.11",
    "shim": {
        "backbone": {
            "deps": [
                "jquery",
                "lodash"
            ],
```

```
            "exports": "Backbone"
        },
        "backbone.layoutmanager": {
            "deps": [
                "jquery",
                "backbone",
                "lodash"
            ],
            "exports": "Backbone.LayoutManager"
        }
    }
};
```

jam 对象用于支持 Jam 的配置。Jam（http://jamjs.org/）是一个前端工作的包管理器，用于帮助安装、升级和配置项目所使用的依赖项。它是 Backbone Boilerplate 包管理器目前的选择。

在 packages 数组中，大量的依赖项被指定包含进来，比如：Backbone、Backbone.LayoutManager 插件、jQuery 以及 Lo-Dash。

Backbone.LayoutManager （https://github.com/tbranyen/backbone.layoutmanager），是 Backbone 的一个插件，提供组装布局和视图的基础。

使用 Jam 添加的额外的包，都会有一个相应的条目添加进来。

11.2.3 main.js

接下来是 main.js，它定义了应用程序的入口点。使用一个全局的 require()方法来加载一个数组，该数组包含了任何其他脚本需要的模块，如应用程序 app.js 以及主路由 router.js。请注意，绝大多数时候，使用 require()只是为了启动一个应用程序，并为所有其他目的调用一个类似方法，名为 define()。

依赖项数组之后定义的函数是一个回调函数，在脚本加载之前不会被触发。注意，为了方便起见，我们使用了局部别名将 app 和 router 引用到 app 和 Router 上。

```
require([
  // Application.
  "app",

  // Main Router.
  "router"
],

function(app, Router) {

  // Define your master router on the application namespace and trigger all
  // navigation from this instance.
```

```
    app.router = new Router();

    // Trigger the initial route and enable HTML5 History API support, set the
    // root folder to '/' by default.  Change in app.js.
    Backbone.history.start({ pushState: true, root: app.root });

    // All navigation that is relative should be passed through the navigate
    // method, to be processed by the router. If the link has a `data-bypass`
    // attribute, bypass the delegation completely.
    $(document).on("click", "a[href]:not([data-bypass])", function(evt) {
      // Get the absolute anchor href.
      var href = { prop: $(this).prop("href"), attr: $(this).attr("href") };
      // Get the absolute root.
      var root = location.protocol + "//" + location.host + app.root;

      // Ensure the root is part of the anchor href, meaning it's relative.
      if (href.prop.slice(0, root.length) === root) {
        // Stop the default event to ensure the link will not cause a page
        // refresh.
        evt.preventDefault();

        // `Backbone.history.navigate` is sufficient for all Routers and will
        // trigger the correct events. The Router's internal `navigate` method
        // calls this anyways.  The fragment is sliced from the root.
        Backbone.history.navigate(href.attr, true);
      }
    });

});
```

在代码中，Backbone Boilerplate 包含了支持 HTML5 History API 的初始化路由这样的样板代码，并处理了其他的导航场景，所以不需要再自己动手了。

11.2.4 app.js

现在来看看 app.js 模块。通常，在非 Backbone Boilerplate 应用程序里，app.js 文件可能只包含核心逻辑或需要启动应用程序的模块引用。

然而，在本例中，该文件用于定义模板和布局配置选项以及操作布局的实用程序。对于新手来说，可能需要理解很多代码，但对于基础的应用程序来说，你可能不需要修改太多代码。相反，你可能会更关心稍后将要讨论的应用程序模块。

```
define([
  "backbone.layoutmanager"
], function() {

  // Provide a global location to place configuration settings and module
  // creation.
  var app = {
    // The root path to run the application.
    root: "/"
```

```
};

// Localize or create a new JavaScript Template object.
var JST = window.JST = window.JST || {};

// Configure LayoutManager with Backbone Boilerplate defaults.
Backbone.LayoutManager.configure({
  // Allow LayoutManager to augment Backbone.View.prototype.
  manage: true,

  prefix: "app/templates/",

  fetch: function(path) {
    // Concatenate the file extension.
    path = path + ".html";

    // If cached, use the compiled template.
    if (JST[path]) {
      return JST[path];
    }

    // Put fetch into `async-mode`.
    var done = this.async();

    // Seek out the template asynchronously.
    $.get(app.root + path, function(contents) {
      done(JST[path] = _.template(contents));
    });
  }
});

// Mix Backbone.Events, modules, and layout management into the app object.
return _.extend(app, {
  // Create a custom object with a nested Views object.
  module: function(additionalProps) {
    return _.extend({ Views: {} }, additionalProps);
  },

  // Helper for using layouts.
  useLayout: function(name, options) {
    // Enable variable arity by allowing the first argument to be the options
    // object and omitting the name argument.
    if (_.isObject(name)) {
      options = name;
    }

    // Ensure options is an object.
    options = options || {};

    // If a name property was specified use that as the template.
    if (_.isString(name)) {
      options.template = name;
    }
```

```
        // Create a new Layout with options.
        var layout = new Backbone.Layout(_.extend({
          el: "#main"
        }, options));

        // Cache the refererence.
        return this.layout = layout;
      }
    }, Backbone.Events);

});
```

JST 表示 JavaScript 模板，并且通常只指在构建过程中已经（或将要）预编译的模板。运行 bbb release 或者 bbb debug 时，Underscore/Lo-dash 模板会被预编译，以避免在浏览器运行时需要编译。

11.2.5 创建 Backbone 样板模块

不要与仅仅是一个 AMD 模块相混淆，一个 Backbone 样板模块是如下模块的脚本组合：

- Model。
- Collection。
- Views（可选）。

使用 grunt-bbb 可以很容易地创建一个新样板模块，再次使用 init：

```
# Create a new module
$ bbb init:module

# Grunt prompt
Please answer the following:
[?] Module Name foo
[?] Do you need to make any changes to the above before continuing? (y/N)

Writing app/modules/foo.js...OK
Writing app/styles/foo.styl...OK
Writing app/templates/foo.html...OK

Initialized from template "module".

Done, without errors.
```

上述代码将生成一个下面这样的模块（foo.js）：

```
// Foo module
define([
  // Application.
```

```
    "app"
  ],

  // Map dependencies from above array.
  function(app) {

    // Create a new module.
    var Foo = app.module();

    // Default Model.
    Foo.Model = Backbone.Model.extend({

    });

    // Default Collection.
    Foo.Collection = Backbone.Collection.extend({
      model: Foo.Model
    });

    // Default View.
    Foo.Views.Layout = Backbone.Layout.extend({
      template: "foo"
    });

    // Return the module for AMD compliance.
    return Foo;

  });
```

请注意样板代码是如何编写模型、集合以及视图的。

另外，我们也可以引用像 Backbone localStorage 或 Offline adapters 这样的插件。在上述代码中，引用一个插件的清晰方式可以像下面这样：

```
// Foo module
define([
  // Application.
  "app",
  // Plug-ins
  'plugins/backbone-localstorage'
],

// Map dependencies from above array.
function(app) {

  // Create a new module.
  var Foo = app.module();

  // Default Model.
  Foo.Model = Backbone.Model.extend({
    // Save all of the items under the `"foo"` namespace.
    localStorage: new Store('foo-backbone'),
```

```
  });

  // Default Collection.
  Foo.Collection = Backbone.Collection.extend({
    model: Foo.Model
  });

  // Default View.
  Foo.Views.Layout = Backbone.Layout.extend({
    template: "foo"
  });

  // Return the module for AMD compliance.
  return Foo;

});
```

11.2.6 router.js

最后，来看看用于处理导航的应用程序路由。Backbone Boilerplate 为我们生成的默认路由包括很全的默认值，并且可以很方便地进行扩展。

```
define([
  // Application.
  "app"
],

function(app) {

  // Defining the application router, you can attach subrouters here.
  var Router = Backbone.Router.extend({
    routes: {
      "": "index"
    },

    index: function() {

    }
  });

  return Router;
});
```

然而，如果在页面加载时（例如，当用户单击了默认路由）想要执行一些特定模块的逻辑，可以将一个模块作为其中一个依赖，并且可以使用 Backbone LayoutManager 将视图附加到布局上，示例如下：

```
define([
  // Application.
  'app',

  // Modules
```

```
    'modules/foo'
  ],

  function(app, Foo) {

    // Defining the application router, you can attach subrouters here.
    var Router = Backbone.Router.extend({
      routes: {
        '': 'index'
      },

      index: function() {
              // Create a new Collection
              var collection = new Foo.Collection();

              // Use and configure a 'main' layout
              app.useLayout('main').setViews({
                      // Attach the bar View into the content View
                      '.bar': new Foo.Views.Bar({
                              collection: collection
                      })
               }).render();
      }
    });

    // Fetch data (e.g., from localStorage)
    collection.fetch();

    return Router;

  });
```

11.3 其他有用的工具和项目

使用 Backbone 开发时，通常需要为应用程序编写很多不同的类和文件。通过为需要的文件生成基本的样本代码，像 Grunt-BBB 这样的基架工具可以帮助我们自动处理这些工作。

11.3.1 Yeoman

如果你喜欢 Grunt-BBB，又想找一个工具帮助自己处理更广泛的开发流程，我很乐意为大家推荐我创建的一个工具——Yeoman（http://yeoman.io/），如图 11-2 所示。

Yeoman 是一个由一组工具集和最佳实践组成的工作流，用于帮助我们进行更有效的开发。它是由基架工具 yo（见图 11-2）、构建工具 Grunt（http://gruntjs.com/）、客户端包管理器 Bower（http://bower.io/）（见图 11-3）3 个部分组成的。

图 11-2　使用 Yeoman 'yo'基架工具搭建一个新的 Backbone 应用程序

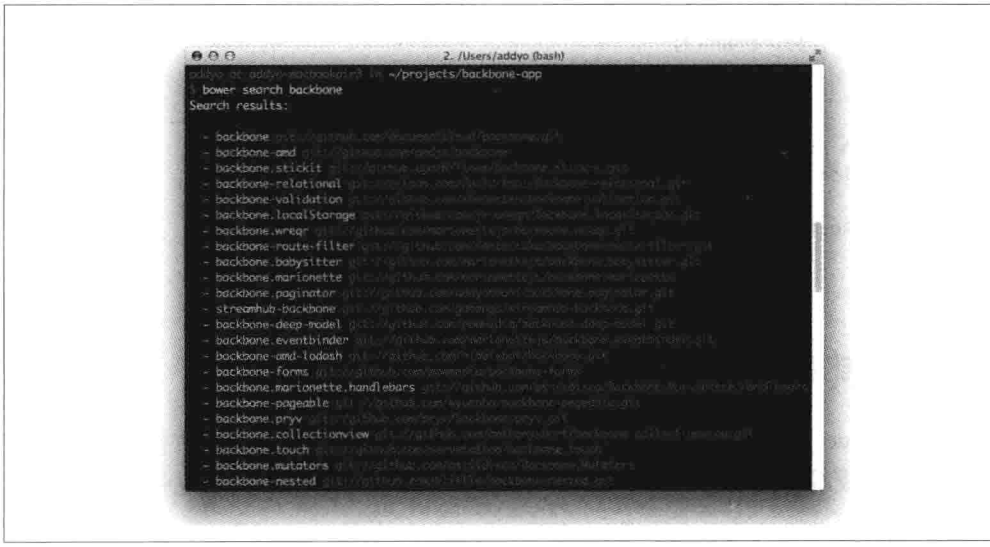


图 11-3　通过 Bower 包管理器查询到的 Backbone 插件及扩展列表

Grunt-BBB 主要为了开启 Backbone 新项目，而 Yeoman 则允许我们使用 Backbone（或其他框架）基架应用；允许通过命令行获取 Backbone 插件；以及编译 CoffeeScript、Sass 或其他抽象代码，而无需额外操作。

大家可能也会对 Brunch（http://brunch.io/）感兴趣，它是一个类似的项目，利用 skeleton boilerplates 生成新的应用程序。

11.3.2 Backbone DevTools

使用 Backbone 构建应用程序时，有一些额外的工具可用于日常工作的调试。

例如，Backbone DevTools 是一个 Chrome DevTools 扩展，它允许检查事件、同步、视图 DOM 绑定以及被实例化的对象（见图 11-4）。

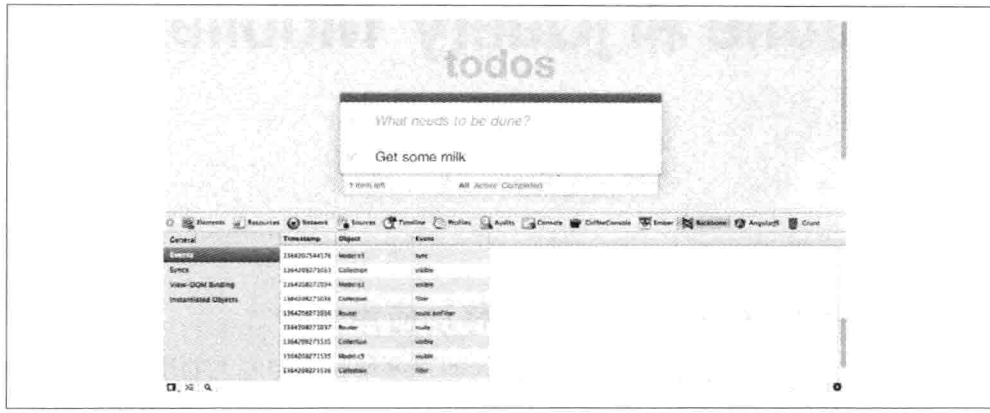


图 11-4　使用 Backbone DevTools 扩展调试本书前面创建的 Todo 应用程序

元素面板（Elements）上显示了一个非常有用的视图层次结构。同样，在检查 DOM 元素时，通过访问$view，可以将最靠近的视图暴露在控制台上。

在撰写本文时，该项目可以在 GitHub 上访问（https://github.com/spect88/backbone-devtools）。

11.4　总结

在本章，我们回顾了 Backbone Boilerplate，并了解了如何使用 bbb 工具来支撑我们的应用程序。

如果想学习更多关于如何利用该项目组织应用程序的知识，BBB 包含了一些内置的样板示例程序，大家可以很轻松地查看。

这些样板项目包括一个样板教程项目（bbb init:tutorial）和一个 TodoMVC 项目的实现（bbb init:todomvc）。推荐查看这些项目，因为它们会让我们更全面地了解 Backbone Boilerplate 以及其模板如何与一个 Web 应用程序的整体设置相适应。

欲获得更多关于 Grunt-BBB 的信息，请查看官方项目的存储库（https://github.com/backbone-boilerplate/grunt-bbb）。如果有兴趣阅读更多信息，还可以访问相关的 slidedeck（http://bit.ly/10b73lU）。

第 12 章

Backbone 和 jQuery Mobile

12.1 使用 jQuery Mobile 进行移动应用开发

移动 Web 的前景是巨大的，它正在以惊人的速度增长。与庞大的增长相关的还有其设备和浏览器的多样性。因此，让应用程序跨平台并支持移动设备非常重要，并且很具有挑战性。创建原生移动应用是昂贵的，非常浪费时间，并且通常还需要多种编程语言经验（如 Objective C、C#、Java 和 JavaScript）来支持多个运行时环境。

HTML、CSS 和 JavaScript 可以让我们构建单个应用程序，并在常见的运行环境（浏览器）上运行。这种方式支持大量的移动设备，如平板电脑、智能手机、笔记本以及传统的个人计算机。

这个具有挑战性的任务不仅仅是使不同的分辨率与文本和图片进行适配，也要在不同的操作系统上提供与原生应用一样的用户体验。像 jQueryUI 一样，jQuery Mobile（或 jQMobile）是一个基于 jQuery 的用户界面框架，适用于所有流行的手机、平板电脑、电子阅读器和桌面平台。它是建立在可访问性和通用访问的思想上的。

该框架的主要思想是能让所有的人不需要编程语言知识，也不需要编写复杂的、特定于设备的 CSS，仅使用 HTML 就能够创建移动应用。出于这个原因，jQMobile 遵循了如下两个主要原则，这是我们首先要理解的，以便将该框架集成至 Backbone：渐进增强（progressive enhancement）和响应式 Web 设计（responsive web design）。

12.1.1 jQMobile 渐进部件增强原则

jQuery Mobile 使用 HTML5 标记驱动的定义和配置，遵循了渐进增强[1]和响应式 Web 设计[2]的原则。

jQuery Mobile 中的页面包含一个带有 data-role="page"属性的元素。在这个页面容器内，任何有效的 HTML 标记都可以使用，但对于典型的 jQM 页面来说，它的直接子元素都是带有 data-role="header"、data-role="content"以及 data-role="footer"的 div 元素。页面的基本要求是要有一个支持导航系统的页面包装器，其余的都是可选的。

一个初始 HTML 页面，看起来可能像下面这样：

```
<!DOCTYPE html>
<html>
<head>
    <title>Page Title</title>

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <link rel="stylesheet"
     href="http://code.jquery.com/mobile/1.3.0/jquery.mobile-1.3.0.min.css" />
    <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
    <script src="http://code.jquery.com/mobile/1.3.0/jquery.mobile-1.3.0.min.js">
    </script>
</head>
<body>

<div data-role="page">
  <div data-role="header">
    <h1>Page Title</h1>
  </div>
  <div data-role="content">
     <p>Page content goes here.</p>
     <form>
       <label for="slider-1">Slider with tooltip:</label>
       <input type="range" name="slider-1" id="slider-1" min="0"
        max="100" value="50"
        data-popup-enabled="true">
     </form>
  </div>
  <div data-role="footer">
     <h4>Page Footer</h4>
```

1. 渐进增强通过分层的方式使用 Web 平台的特性，允许访问页面的基本内容和功能，即便 JavaScript 是关闭的或用户用的不是现代浏览器。增强的经验是提供给那些开启 JavaScript 或者拥有最新、最强悍浏览器的用户的。
2. 响应式 Web 设计（RWD）是一种设计页面的方式，为不同的浏览环境进行布局适配，以便提供一个更优的视觉体验。CSS 媒体查询通常是用来实现这一目标的。

```
    </div>
  </div>
  </body>
  </html>
```

jQuery Mobile 使用其渐进部件增强 API（Progressive Widget Enhancement API）将已经编写的 HTML 定义转换为可呈现的 HTML 和 CSS。它还可以执行使用配置设置、属性设置、运行时特定设置进行设置的 JavaScript 代码。运行结果如图 12-1 所示。

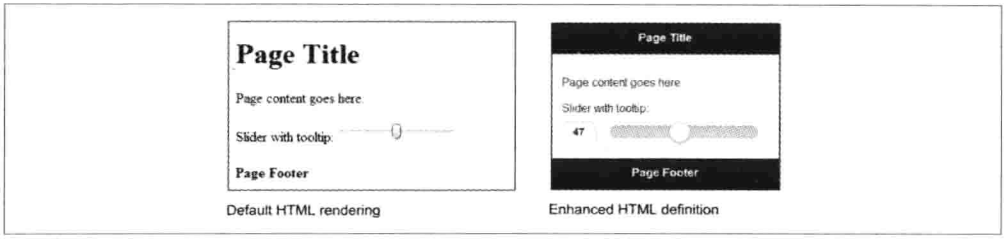


图 12-1　默认 HTML 和 jQuery Mobile 增强版 HTML 的界面比较

这就意味着，每当 HTML 内容增加或改变时，它就需要使用 jQuery Mobile 的渐进部件增强 API 进行处理。

12.1.2　理解 jQuery Mobile 导航

jQuery Mobile 导航系统通过自动拦截标准链接和表单提交并将其转化为 AJAX 请求，控制着应用程序的生命周期。一旦链接被单击或表单被提交，导航系统将自动拦截该事件，并使用链接的 href 值或表单的 action 值发起 AJAX 请求，而不是重新加载页面。

当页面文档请求时，jQuery Mobile 搜索所有含有 data-role="page"属性的文档，解析其内容，并将代码插入到原始页面的 DOM 中。一旦新页面准备好了，jQuery Mobile 的 JavaScript 就触发一个隐藏上一个页面并显示新页面的过渡（transition）。

接下来，新进入页面上的所有小部件都应用所有的样式和行为进行增强，而页面的其余部分则都被丢弃，所以任何脚本、样式表或其他信息都将不会被包括。

通过多页模板功能(multipage templating feature)，在同一个带有<body>标签的 HTML 文件里，可以通过定义拥有 data-role="page"或 data-role="dialog"属性、且拥有 id 属性（在前面加#用于连接功能）的 div 进行页面添加，想添加多少就可以添加多少：

```
<html>
  <head>...</head>
  <body>
  ...
  <div data-role="page" id="firstpage">
    ...
   <div data-role="content">
```

```
      <a href="#secondpage">go to secondpage</a>
    </div>
  </div>
  <div data-role="page" id="secondpage">
    ...
    <div data-role="content" >
      <a href="#firstdialog" data-rel="dialog" >open a page as a dialog</a>
    </div>
  </div>
  <div data-role="dialog" id="firstdialog">
    ...
    <div data-role="content">
      <a href="#firstpage">leave dialog and go to first page</a>
    </div>
  </div>
</body>
</html>
```

例如，要想导航到 secondpage 页面，并在上面使用一个 fade-transition 过渡来显示一个对话框，就要在锚点标记上添加 data-rel="dialog"、datatransition="fade"以及 href="index.html#secondpage"属性。

大致说来，jQuery Mobile 有自己的事件周期，jQuery Mobile 是一个微型 MVC 框架，通过与生俱来的 HTML 配置，包括了渐进部件增强、预读取、缓存以及与多页面模板等功能。通常来说，Backbone.js 开发人员不需要了解其内部的事件工作流程，但是需要知道如何应用基于 HTML 的配置，这些配置将在事件阶段采取行动。12.4.2 小节将详细讲述关于在细粒度 JavaScript 变化需要应用时，要如何处理这些特殊的场景。

关于 jQuery Mobile 的进一步介绍和解释，请访问：

- http://view.jquerymobile.com/1.3.0/docs/intro/。
- http://view.jquerymobile.com/1.3.0/docs/widgets/pages/。
- http://view.jquerymobile.com/1.3.0/docs/intro/rwd.php。

12.2 Backbone 应用的基础设置（用于 jQuery Mobile）

通常，开发人员使用 jQuery Mobile 和 MV*框架构建应用程序，遇到的第一个主要障碍就是两个框架都想处理应用程序导航。

要将 Backbone 和 jQuery Mobile 进行结合，首先需要禁用 jQuery Mobile 的导航系统和渐进增强；接着在 Backbone 的应用程序生命周期内，使用 jQM 的自定义 API 进行应用配置以及增强组件。

图 12-2 中的移动应用是基于 TodoMVC（Backbone-Require.js）示例中的现有代码来构建的，在第 8 章已有讨论，为支持 jQuery Mobile 进行了增强。

图 12-2　使用 jQuery Mobile 构建的 TodoMVC 应用

该实现使用了 Grunt-BBB 以及 Handlebars.js。如图 12-3 所示即为将要提供的额外的移动应用实用工具，可以很轻松地进行组合和扩展（见第 6 章和第 11 章）。

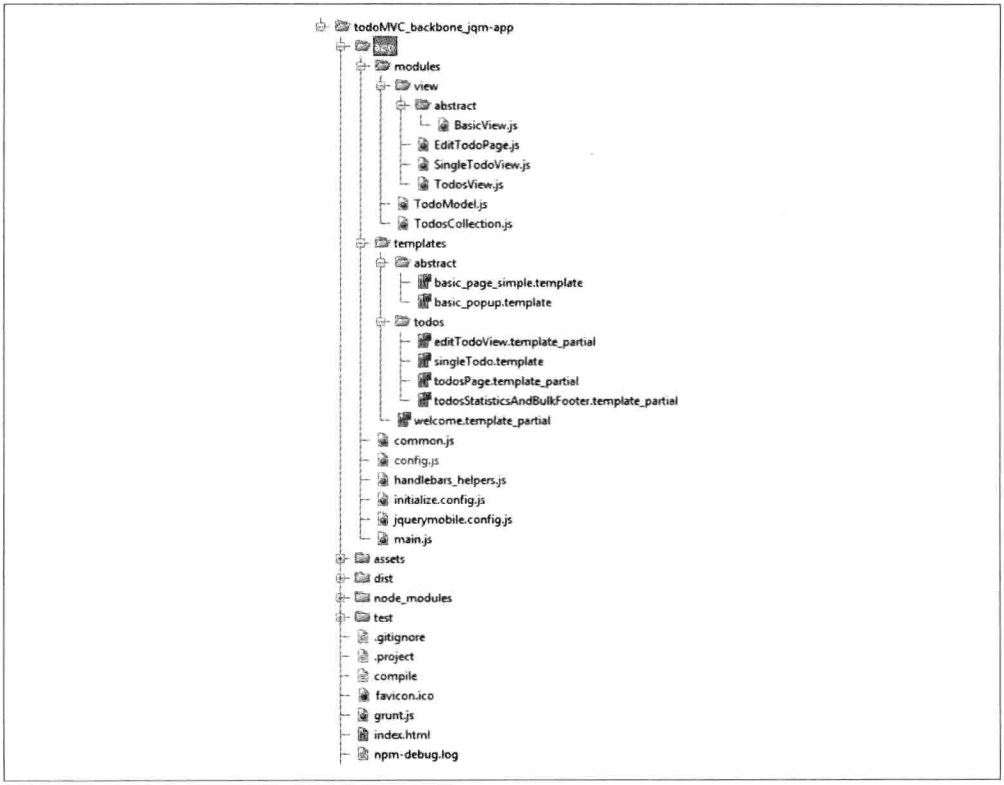


图 12-3　使用 jQueryMobile 和 Backbone 构建 TodoMVC 应用的工作区

Require.js 加载文件的顺序如下：

（1）jQuery。

（2）Underscore/Lo-Dash。

（3）handlebars.compiled。

（4）TodoRouter（初始化特定视图）。

（5）jQueryMobile。

（6）jQueryMobile CustomInitConfig。

（7）Backbone 路由的初始化。

在项目目录下打开控制台，然后运行 Grunt-Backbone 的命令 grunt handlebars 或 grunt watch 时，所有的模板都会被编译，并且编译到 dist/debug/handlebars_packaged 目录。要启动应用程序，就要运行 grunt server 命令。

从 Backbone 路由进行重定向时，如下文件进行了实例化：

- BasicView.js 和 basic_page_simple.template：BasicView 用于负责 Handlebars 的多页模板处理。它的 render 实现调用了 jQuery Mobile API（$.mobile.changePage）来处理页面导航和渐进部件增强。
- 将视图和局部模板进行结合——例如 EditTodoPage.js 和 editTodoView.template_partial：index.html 的 head 部分需要加载 jquerymobile.css 以及 base.css，用于 Todo-MVC 应用的所有页面，并且还要加载 index.css，用于项目特定的自定义样式。

```
<html>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <meta name="viewport" content="width=device-width,initial-scale=1">

    <title>TodoMVC Jquery Mobile</title>

<!-- widget and responsive design styles -->
    <link rel="stylesheet" href="/assets/css/jquerymobile.css">
<!-- used by all TodoMVC apps -->
    <link rel="stylesheet" href="/assets/css/base.css">
<!-- custom css -->
```

```
    <link rel="stylesheet" href="/assets/css/index.css">
</head>

<body>
    <script data-main="/app/config" src="/assets/js/libs/require.js"></script>
</body>
</html>
```

12.3 Backbone 和 jQueryMobile 的工作流程

通过将 jQuery Mobile 框架的路由和导航功能委托给 Backbone，我们可以通过程序结构的清晰分离获益，稍后在桌面 Web 页面、平板电脑和移动应用程序之间可以很容易地进行逻辑共享。

现在需要面对 Backbone 和 jQuery Mobile 处理请求的不同方式。Backbone.Router 提供了一个显式的方式来定义自定义导航路由，而 jQuery Mobile 则在同一个文档里使用 URL 哈希片段来引用单独的页面或视图。

之前已经提出了一些想法来解决这个问题，其中包括手动修补 Backbone 和 jQuery Mobile。下面演示的解决方案不仅可以简化处理 jQuery Mobile 组件初始化事件周期，而且可以使用现有的 Backbone Router 来处理。

将导航控制从 jQuery Mobile 适配到 Backbone 上，首先需要在 mobileinit 事件上应用一些特定设置，该事件在框架加载以后触发，以便让 Backbone 路由决定要加载哪个页面。

jquerymobile.config.js 这个配置将使 jQM 把导航控制委派给 Backbone，并且还将可以手动触发部件的创建：

```
$(document).bind("mobileinit", function(){

// Disable jQM routing and component creation events
   // disable hash-routing
   $.mobile.hashListeningEnabled = false;
   // disable anchor-control
   $.mobile.linkBindingEnabled = false;
   // can cause calling object creation twice and back button issues are solved
   $.mobile.ajaxEnabled = false;
   // Otherwise after mobileinit, it tries to load a landing page
   $.mobile.autoInitializePage = false;
   // we want to handle caching and cleaning the DOM ourselves
   $.mobile.page.prototype.options.domCache = false;

// consider due to compatibility issues
   // not supported by all browsers
```

```
    $.mobile.pushStateEnabled = false;
    // Solves phonegap issues with the back-button
    $.mobile.phonegapNavigationEnabled = true;
    //no native datepicker will conflict with the jQM component
    $.mobile.page.prototype.options.degradeInputs.date = true;
});
```

新工作流程的行为和用法解释如下，按功能进行分组：

（1）路由到具体视图页面。

（2）移动页面模板的管理。

（3）DOM 管理。

（4）$.mobile.changePage。

接下来的讨论中，如图 12-4 所示，图中的步骤①至⑪描述了这个新的移动应用工作流程。

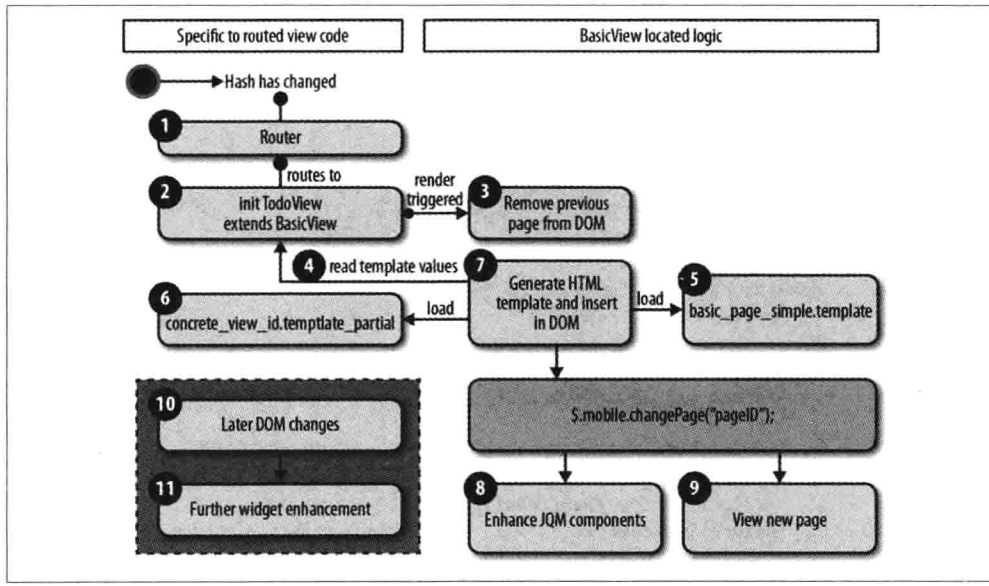


图 12-4　TodoMVC 的工作流程（使用 Backbone 和 jQueryMobile）

12.3.1　路由到具体视图页面，继承于 BasicView

当哈希 URL 更改时（例如，链接单击），配置信息会阻止 jQM 触发其事件，而 Backbone 的路由会监听哈希变化，并决定要请求哪个视图。

经验表明，对于移动页面，为 jQM 组件（如基础页面、弹出窗口、对话框）创建

基础原型以及使用 jQuery 验证插件是一个很好的实践，这将使得在运行时对特定于设备的视图进行交互以及适配通用策略变得更加容易，也将有助于增加语法、以及支持对 JavaScript 和 Backbone 的原型链继承。

通过创建 BasicView 超类，可以使所有继承它的视图页面共享同一个处理 jQM 的方式、同一个模板引擎的相同用法以及特定视图的处理方法。

使用 Grunt/Yeoman 进行构建时，语义模板由 Handlebar.js 编译，并且 AMD 模板被编译到一个单独的文件中。通过将所有的页面定义合并到一个单一文件应用中，可以使它具有离线能力，这对于移动应用程序非常重要。

12.3.2 移动页面模板的管理

在一个具体的视图页面里，可以对静态值和函数进行属性重写，以便返回 BasicView 超类的动态值。这些值最后将由 BasicView 进行处理，并在 Handlebars 的帮助下构建 jQuery Mobile 页面的 HTML。

额外的动态模板参数（如 Backbone 模型信息）将从特定的视图中获取，并与 BasicView 中的参数进行合并。

一个具体的视图看起来可能是这样的（EditTodoPage.js）：

```
define([
    "backbone", "modules/view/abstract/BasicView"],
    function (Backbone, BasicView) {
        return BasicView.extend({
            id : "editTodoView",
            getHeaderTitle : function () {
                return "Edit Todo";
            },
            getSpecificTemplateValues : function () {
                return this.model.toJSON();
            },
            events : function () {
                // merged events of BasicView, to add an older fix for
                // back button functionality
                return _.extend({
                    'click #saveDescription' : 'saveDescription'
                }, this.constructor.__super__.events);
            },
            saveDescription : function (clickEvent) {
                this.model.save({
                    title : $("#todoDescription", this.el).val()
                });
                return true;
            }
        });
    });
```

默认情况下，BasicView 使用 basic_page_simple.template 作为 Handlebars 的模板。如果需要使用自定义模板，或者想要引入一个使用替代模板的新超级抽象视图，则需覆盖 getTemplateID 函数：

```
getTemplateID : function(){
  return "custom_page_template";
}
```

按照约定，id 属性将被视为 jQM 页面的 id，对应模板文件的文件名也需要作为 basic_page_simple.template 名字的一部分。对于 EditTodoPage 视图这个例子来说，文件的名称应该是 editTodoPage.template_partial。

每个具体的页面都会被作为其中的一部分，插入到 datarole="content"元素里，templatePartialPageID 参数存放在该元素里。

稍后，在 EditTodoPage 中，getHeaderTitle 函数的结果将替换抽象模板（basic_page_simple.template）中的 headerTitle。

```
<div data-role="header">
        {{whatis "Specific loaded Handlebars parameters:"}}
        {{whatis this}}
        <h2>{{headerTitle}}</h2>
        <a id="backButton" href="href="javascript:history.go(-1);"
         data-icon="star" data-rel="back" >back</a>
    </div>
    <div data-role="content">
        {{whatis "Template page trying to load:"}}
        {{whatis templatePartialPageID}}
        {{> templatePartialPageID}}
    </div>
    <div data-role="footer">
        {{footerContent}}
</div>
```

Handlebars 的 whatis 视图助手用于对参数进行简单的记录。

getSpecificTemplateValues 返回的其他所有额外参数将插入到具体的 editTodoPage.template_partial 模板中。

因为 footerContent 使用很少，所以它的内容将由 getSpecificTemplateValues 返回。

对于 EditTodoPage 视图，所有的模型信息都是从具体的局部页面中返回的，title 也是在具体的局部页面中进行使用的：

```
<div data-role="fieldcontain">
    <label for="todoDescription">Todo Description</label>
    <input type="text" name="todoDescription" id="todoDescription"
     value="{{title}}" />
</div>
    <a id="saveDescription" href="#" data-role="button" data-mini="true">Save</a>
```

render 触发时，basic_page_simple.template 和 editTodoView.template_partial 模板将被加载，EditTodoPage 和 BasicView 中的参数将合并在一起，由 Handlebars 生成结果，如下：

```
<div data-role="header">
    <h2>Edit Todo</h2>
    <a id="backButton" href="href="javascript:history.go(-1);"
     data-icon="star" data-rel="back" >back</a>
</div>
<div data-role="content">
  <div data-role="fieldcontain">
   <label for="todoDescription">Todo Description</label>
   <input type="text" name="todoDescription" id="todoDescription"
    value="Cooking" />
  </div>
  <a id="saveDescription" href="#" data-role="button" data-mini="true">Save
  </a>
</div>
<div data-role="footer">
    Footer
</div>
```

接下来的小节将解释模板参数是如何由 BasicView 类收集的，以及 HTML 定义是如何加载的。

12.3.3 DOM 管理与$.mobile.changePage

render 执行时（下面例子源码中的第 29 行），BasicView 首先通过删除上一个页面（第 70 行）来清理 DOM。从 DOM 中删除元素不能使用$.remove，但可以使用 $previousEl.detach()，因为分离（detach）不会删除该元素上附加的事件和数据。

这很重要，因为 jQuery Mobile 仍然需要这些信息（例如，切换到另一个页面时需要触发的过渡效果）。记住，DOM 数据和事件稍后还应该被清除，以避免出现可能的性能问题。

除了 cleanupPossiblePageDuplicationInDOM 函数使用的 DOM 清理策略外，其他策略也是可行的。有和当前页面有相同 id 的旧页面，且在之前已经被请求了，则可以从 DOM 中删除，这也是防止 DOM 重复的一个策略。根据最能满足应用程序需要的内容，它也可能是一个使用缓存机制进行交换的简单[1]问题。

[1] 译者注：直译是“给根香蕉就能解决的问题”。源自于训练有素的猴子可以从事低级别工作，给它根香蕉就能把活干了。

接下来，BasicView 从具体的视图实现中收集所有模板参数，并将请求页面的 HTML 插入到 body 内。图 12-4 中的步骤 4、5、6 和 7 是用于做这件事情的（代码清单，第 23～51 行之间）。

此外，data-role 将在 jQuery Mobile 页面上进行设置，常用的属性值有：page、dialog 或 popup。

从 BasicView.js 中可以看到（第 74 行开始），goBackInHistory 函数包含一个处理回退按钮行为的手动实现。在某些情况下，jQuery Mobile 的后退按钮导航功能在老版本和禁用导航系统的情况下是不能使用的。

```
1 define([
2     "lodash",
3     "backbone",
4     "handlebars",
5     "handlebars_helpers"
6 ],
7
8 function (_, Backbone, Handlebars) {
9     var BasicView = Backbone.View.extend({
10         initialize: function () {
11             _.bindAll();
12             this.render();
13         },
14         events: {
15             "click #backButton": "goBackInHistory"
16         },
17         role: "page",
18         attributes: function () {
19             return {
20                 "data-role": this.role
21             };
22         },
23         getHeaderTitle: function () {
24             return this.getSpecificTemplateValues().headerTitle;
25         },
26         getTemplateID: function () {
27             return "basic_page_simple";
28         },
29         render: function () {
30             this.cleanupPossiblePageDuplicationInDOM();
31             $(this.el).html(this.getBasicPageTemplateResult());
32             this.addPageToDOMAndRenderJQM();
33             this.enhanceJQMComponentsAPI();
34         },
35 // Generate HTML using the Handlebars templates
36         getTemplateResult: function (templateDefinitionID, templateValues) {
```

```
37              return window.JST[templateDefinitionID](templateValues);
38          },
39  // Collect all template parameters and merge them
40          getBasicPageTemplateResult: function () {
41              var templateValues = {
42                  templatePartialPageID: this.id,
43                  headerTitle: this.getHeaderTitle()
44              };
45              var specific = this.getSpecificTemplateValues();
46              $.extend(templateValues, this.getSpecificTemplateValues());
47              return this.getTemplateResult(this.getTemplateID(),
                templateValues);
48          },
49          getRequestedPageTemplateResult: function () {
50              this.getBasicPageTemplateResult();
51          },
52          enhanceJQMComponentsAPI: function () {
53  // changePage
54              $.mobile.changePage("#" + this.id, {
55                  changeHash: false,
56                  role: this.role
57              });
58          },
59  // Add page to DOM
60          addPageToDOMAndRenderJQM: function () {
61              $("body").append($(this.el));
62              $("#" + this.id).page();
63          },
64  // Cleanup DOM strategy
65          cleanupPossiblePageDuplicationInDOM: function () {
66          // Can also be moved to the event "pagehide": or "onPageHide"
67              var $previousEl = $("#" + this.id);
68              var alreadyInDom = $previousEl.length >= 0;
69              if (alreadyInDom) {
70                  $previousEl.detach();
71              }
72          },
73  // Strategy to always support back button with disabled navigation
74          goBackInHistory: function (clickEvent) {
75              history.go(-1);
76              return false;
77          }
78      });
79
80      return BasicView;
81  });
```

在将动态 HTML 添加到 DOM 之后，$.mobile.changePage 必须在步骤 8 进行应用（第 54 行代码）。

这是最重要的 API 调用，因为它触发了当前页面的 jQuery Mobile 组件的创建。

接下来，网页将在第 9 步显示给用户（见图 12-5）。

```
<a data-mini="true" data-role="button" href="#" id="saveDescription"
 data-corners="true"
 data-shadow="true" data-iconshadow="true" data-wrapperels="span" data-theme="c"
 class="ui-btn ui-shadow ui-btn-corner-all ui-mini ui-btn-up-c">
    <span class="ui-btn-inner">
         <span class="ui-btn-text">Save</span>
     </span>
</a>
```

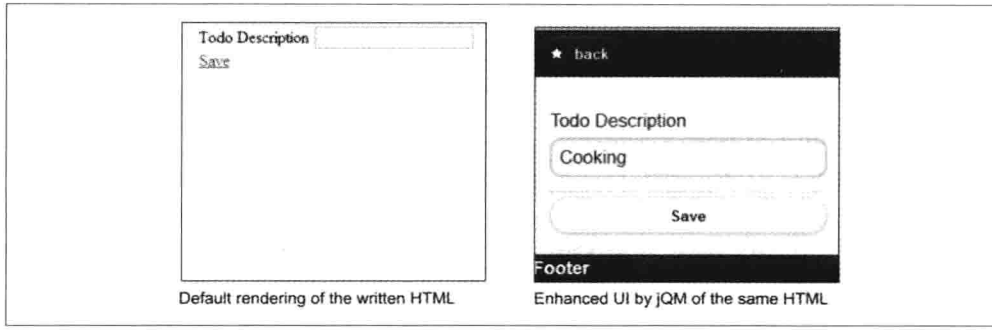


图 12-5　手写 HTML 页面和 jQuery Mobile 页面（增强的 todo 描述页面）的外观

用户界面增强是在第 52 行的 enhanceJQMComponentsAPI 函数中实现的：

```
$.mobile.changePage("#" + this.id, {
                      changeHash: false,
                      role: this.role
                    });
```

要保留哈希路由控制，必须把 changeHash 设置为 false，并提供适当的 role 参数，以保证正确的页面外观。最后，changePage 将使用其定义的过渡把新页面显示给用户。

对于基本场景，建议一个页面有一个视图，在需要部件增强的时候，总是通过调用 $.mobile.changePage 来渲染完整的页面。

将新添加的 HTML 片段处理成层级格式并添加到 DOM 中，需要应用先进的技术来确保移动组件的正确外观。在 UI 元素上创建局部 HTML 代码或更新值时，也需要非常小心。下一节将解释如何处理这些情况。

12.4　在 Backbone 上应用 jQM 高级技术

12.4.1　动态 DOM 脚本

先前描述的解决方案通过调用$.mobile.changePage('pageID')解决了使用 Backbone 处

理路由的问题，此外，也保证了 HTML 页面被 jQuery Mobile 进行了完全增强。

使用 jQuery Mobile 的第二个棘手部分是动态操纵特定的 DOM 内容（例如，使用 Ajax 加载内容以后）。建议大家使用本节的这种技术，除非有证据表明会引起大的性能问题。

当前版本（1.3）的 jQM 提供了 3 种方式，在官方的 API、论坛、博客上进行文档展示和解释。

```
$(pageId).trigger(pagecreate)
```

创建 header 以及 footer 内容的 HTML 标记

```
$(anyElement).trigger(create)
```

创建该元素以及所有子元素的 HTML 标记：

- $(myListElement).listview(refresh)
- $([type=radio]).checkboxradio()
- $([type=text]).textinput()
- $([type=button]).button()
- $([data-role=navbar]).navbar()
- $([type=range]).slider()
- $(select).selectmenu()

jQM 的每个组件都提供了可调用的插件方法，用于更新特定 UI 元素的状态。

有时候，从头开始创建一个组件可能会看到这样的错误：“Cannot call methods on ListView prior to initialization.”。在组件初始化之后、标记增强之前，通过以下方式调用，就可以避免这个问题：

```
$('#mylist').listview().listview('refresh')
```

要查看进一步脚本化 jQM 页面的更多细节和增强功能，阅读它的 API 并经常关注它的发布说明：

- jQuery Mobile: Page Scripting（http://jquerymobile.com/test/ docs/pages/page-

scripting.html）。

- jQuery Mobile: Document Ready vs. Page Events（http://bit.ly/ZMzkix）。
- StackOverflow: Markup Enhancement of Dynamically Added Content（http://bit.ly/XTNfHa）。

如果考虑使用模型绑定插件，需要想出一种自动化的机制来丰富单一组件。

现在，了解了有关动态 DOM 脚本的更多内容，你会知道它也许不能完全再现一个组件的创建（如 Listview），因为它需要较长的时间去加载并减少事件委托的复杂性。我们应该使用组件特定的插件，这些插件将只更新 HTML 和 CSS 需要更新的部分。

对于 ListView，将需要调用如下函数来更新添加、编辑或删除后的列表：

```
$('#mylist').listview()
```

你需要提供一个可以检测组件类型的方法，用于判断哪个插件方法需要被调用。jQuery Mobile Angular.js 适配器提供了一个这样的策略以及解决方案。

在 GitHub 上（http://bit.ly/YrZ2wu）可以查看到一个使用 jQuery Mobile 进行模型绑定的示例。

12.4.2 拦截 jQuery Mobile 事件

特殊情况下，需要在一个已触发的 jQuery Mobile 事件上做出一些行动，可以像下面这样做：

```
$('#myPage').live('pagebeforecreate', function(event){
    console.log(page was inserted into the DOM');

    // run your own enhancement scripting here...
    // prevent the page plug-in from making its manipulations
    return false;
});

$('#myPage').live('pagecreate', function(event){
    console.log('page was enhanced by jQM');
});
```

在这样的场景中，知道什么时候触发 jQuery Mobile 事件非常重要。图 12-6 描述了事件的周期（page A 离开、page B 载入）。

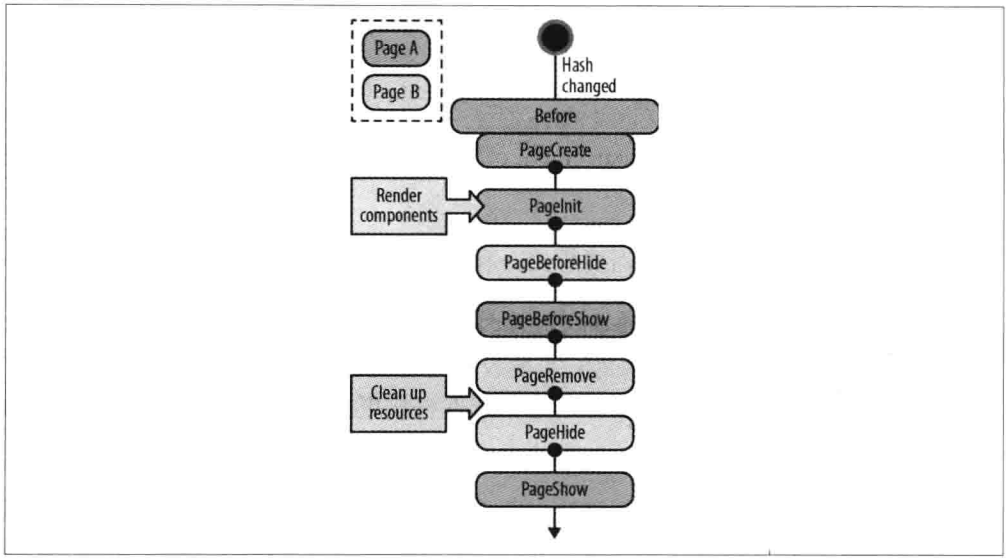


图 12-6　jQuery Mobile 事件周期

另外一种拦截方式是 jQuery Mobile Router 项目，可以用它替代 Backbone 路由。在 jQM Router 项目的帮助下，会有一种强大的方法来拦截和路由各种 jQM 事件。它是 jQuery Mobile 的一个扩展，可以独立使用。

要知道，jQM Router 会漏掉 Backbone.Router 的一些特性，并且和 jQuery Mobile 框架紧密耦合。鉴于这些原因，我们没有在 TodoMVC 应用程序中使用它。如果要使用它，就要考虑使用 Backbone.js 自定义构建来排除路由代码。相对于 17.1KB 的最大压缩大小，可以节省大约 25%的空间。

访问 Backbone 自定义构建工具，网址为 http://gregfranko.com/backbone/customBuild/。

12.4.3　性能

性能是与移动设备有关的一个很重要的话题。jQuery Mobile 提供了各种用于创建性能日志的工具，可以让我们全面了解花在路由逻辑、组件增强以及视觉效果上的实际时间。

根据不同的设备，过渡所花费的时间可占用多达 90%的加载时间。要禁用所有的过渡，可以在配置代码块中，通过$.mobile.changePage()将 transition 设置为 none：

```
$(document).bind("mobileinit", function(){
…
// Otherwise, depending on takes up to 90% of loadtime
  $.mobile.defaultPageTransition = "none";
```

```
    $.mobile.defaultDialogTransition = "none";
      });
    })
```

或者，可以考虑为特定设备添加特定设置，例如：

```
$(document).bind("mobileinit", function(){

  var iosDevice =((navigator.userAgent.match(/iPhone/i))
  || (navigator.userAgent.match(/iPod/i))) ? true : false;

  $.extend(  $.mobile , {
    slideText :  (iosDevice) ? "slide" : "none",
    slideUpText :  (iosDevice) ? "slideup" : "none",
    defaultPageTransition:(iosDevice) ? "slide" : "none",
    defaultDialogTransition:(iosDevice) ? "slide" : "none"
  });
```

同时，考虑为已经增强的 jQuery Mobile 页面做预缓存。

在性能上，jQuery Mobile API 的每个新版本都会进行增强。建议查看最新更新的 API 来确定最适合的使用动态脚本的最优缓存策略。

欲获得关于性能的更多信息，请阅读以下内容：

- jQuery Mobile profiling tools（https://github.com/jquery/jquery-mobile/tree/master/tools）。
- Device-specific jQuery Mobile configurations（http://backbonefu.com/2012/01/jquery-mobile- and-backbone-js-the-ugly/）。
- jQuery Mobile debugging tools（http://bit.ly/17UMba2）。
- jQuery Mobile precaching functionalities（http://jquerymobile.com/demos/1.2.0/docs/pages/page-cache.html）。

12.4.4 智能的多平台支持管理

如今，公司通常都已经有自己的 Web 页面了，管理层决定要给客户提供额外的移动应用。Web 页面代码和移动应用代码变得相互独立，网页内容或功能的改变所需要的时间比单独 Web 页面的改变要长得多。

目前的趋势是提供越来越多的移动平台和维度，相应的工作量也在增加。最终，为每种设备都创建一种用户体验并不总是可行的。但是，不管浏览器和平台是什么样的，其内容对所有用户都是可用的。在设计阶段，必须坚持牢记这一原则。

Responsive (http://www.lukew.com/ff/entry.asp?933) 和 mobile-first (http://www.abooka-part.com/ products/mobile-first) 方式可以应对这些挑战。

本章提出的移动应用架构解决了很多繁重的实际需求，是因为它支持响应式布局，甚至支持不能进行媒体查询处理的浏览器。作为一个 UI 框架，jQM 不同于 jQuery UI 之处可能不是很明显。jQuery Mobile 使用小部件厂，应用不仅仅局限于移动环境。

要使 jQuery Mobile 和 Backbone 支持多个平台的浏览器，增加时间和精力就可以拥有：

(1) 理想情况下，一个代码项目，使用不同的 CSS 适应不同的设备。

(2) 相同的代码项目，根据每个设备类型，在运行时改变不同的 HTML 模板和超类。

(3) 相同的代码项目，响应 API 设计和 jQuery Mobile 的大多数小部件将被重用。对于桌面浏览器，一些组件将由另一个部件框架添加（如 jQueryUI 或 Twitter Bootstrap），也就是由 HTML 模板控制。

(4) 相同的代码项目，在运行时，jQuery Mobile 将完全由另一个小部件框架（如 jQueryUI 或 Twitter Bootstrap）所取代。超类、配置以及具体的 Backbone.View 代码片段也需要被替换。

(5) 不同的代码项目，但重用常见的模块。

(6) 一个完全独立的桌面应用程序代码项目。原因可能是使用了完全不同的编程语言和/或者不同的框架，缺乏响应式设计知识或污染遗留。

理想的解决方案是——只用移动框架构建美观的桌面应用程序——这听起来挺疯狂，但却是可行的。

如果在桌面浏览器上浏览 jQuery Mobile API 页面，它完全不像一个移动应用程序（见图 12-7）。

同样，对于 jQuery Mobile 设计的例子，jQuery Mobile 打算添加更多的用户界面体验（见图 12-8）。

手风琴（accordion）、时间选择器（datepicker）、滑动条（slider）——桌面 UI 上的一切内容，都是重用 jQM 在移动设备上给用户提供的内容。举例来说，在组件上添加属性 data-mini="true"，将会把笨拙的移动组件从桌面浏览器上移除。

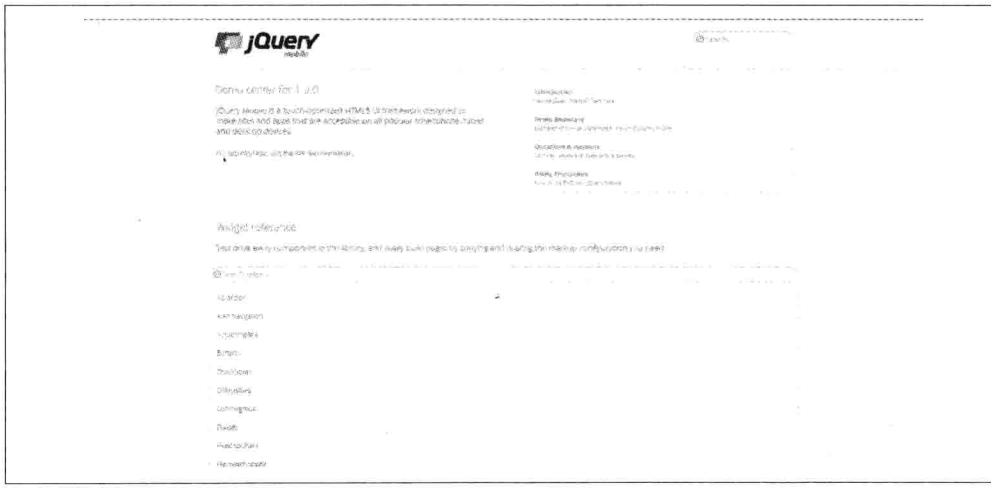


图 12-7　桌面浏览 jQuery Mobile API 和 Docs 页面程序

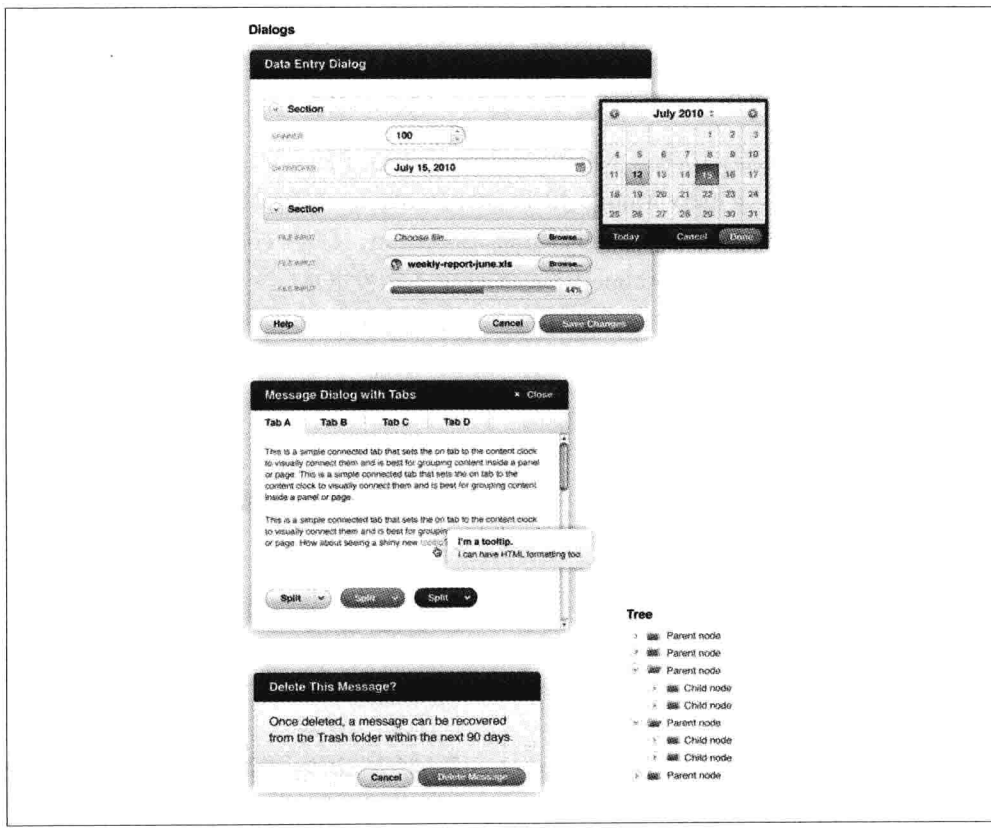


图 12-8　jQuery Mobile 在桌面环境的设计示例

对于桌面应用程序，请访问 jQuery Mobile 网站（http://jquerymobile.com/demos/1.2.0/docs/forms/forms-all-mini.html）查看更多关于迷你小部件的信息。

由于一些媒体查询，桌面 UI 可以使用空格、扩展组件块优化，并提供不同的布局，同时还仍可使用 jQM 作为组件框架。

这么做的好处是不需要单独引入另外的部件框架（例如，jQuery UI）来使用这些特性。由于 ThemeRoller，组件看起来和我们想象的差不多，并且应用程序用户可以得到一个低分辨率的 jQM UI 以及类似 jQM 的 UI 的一切。

重点是要记住，如果还没有使用麻烦的条件语句基于屏幕分辨率（使用 matchMedia.js 等）来进行 script/style 加载的话，还有更简单的方法可以处理跨平台的组件主题。至少 jQuery Mobile 的响应式设计 API（自 1.3.0 版本添加）总是合理的，因为它对移动和桌面都适用。总之，可以使用 jQuery Mobile 组件给用户一个典型的桌面外观，并且他们不会感觉到差别。

更多关于 jQuery Mobile 响应式设计的内容，请查看 http://view.jquerymobile.com/1.3.0/docs/intro/rwd.php。

同样，对于桌面浏览器，如果有 jQuery Mobile 应用程序的 CSS 样式和配置的限制，就会最大限度地减少 jQuery Mobile 和 Twitter Bootstrap 的额外使用。这种情况下，桌面浏览器请求该页面，并且加载 Twitter Bootstrap，移动 TodoMVC 应用会需要使用条件判断，以确保在 Backbone.View 实现中不触发 jQM 部件渐进增强插件 API（12.4.1 小节有叙述）。因此，正如在前面小节中解释的，在加载完整的页面后，我们推荐只使用一次$.mobile.changePage 来触发部件增强功能。

图 12-9 展示了这种部件混合使用的示例。

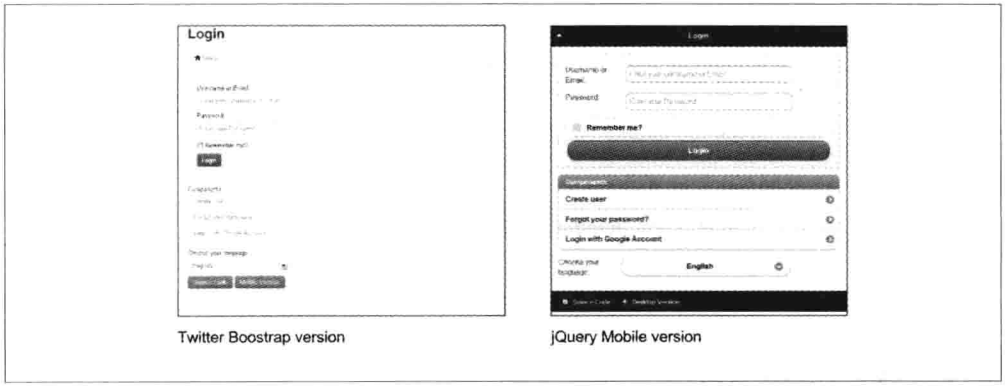


图 12-9　应用引擎样板、桌面、移动的外观

虽然它使用 Python 编程语言对这种服务器端技术进行模板化处理，页面加载时触发渐进增强（triggering progressive enhancement）这一原则和$mobile.changePage 是一样的。

正如你可以看到的，JavaScript 甚至 CSS 都保持不变。在代码实现中，唯一特定于设备的条件判断和差异是选择导入合适的框架，条件判断和差异位于 HTML 模板中：

```
...
 {% if is_mobile %}
    <link rel="stylesheet" href="/mobile/jquery.mobile-1.1.0.min.css" />
    {% else %}
      <link rel="apple-touch-icon" href="/apple-touch-icon.png" />
      <link rel="stylesheet" href="/css/style.css" />
      <link rel="stylesheet" href="/css/bootstrap.min.css">
      <link rel="stylesheet" href="/css/bootstrap-responsive.min.css">
    {% endif %}
      <link rel="stylesheet" href="/css/main.css" />

    {% block mediaCSS %}{% endblock %}
...
 {% if is_mobile %}
      <script src="/mobile/jquery.mobile-1.1.0.min.js"></script>
    {% else %}
      <script src="/js/libs/bootstrap.min.js"></script>
    {% endif %}
...
```

第 13 章

Jasmine

单元测试（unit testing）的一个定义是一个处理过程，该过程包括将应用程序中的代码处理成最小的可测试单元，将其从其余部分的代码库中进行隔离，并确定其行为与预期完全一致。

一个经过良好测试的应用程序，它的每个函数应该有自己单独的单元测试，对不同条件的处理进行测试。在功能确认完整之前，所有的测试都必须通过。这允许开发人员可以在确认这些修改不会导致出错的情况下，进行代码单元及其依赖项的修改。

单元测试的一个基本例子是开发人员断言，给加法函数传入特定多个值，能返回正确的结果。例如，和本书更相关的一个例子，我们可能希望断言，在列表里添加一个新的待办项时，结果将会给特定的 Todos 集合添加一个模型。

当构建现代 Web 应用程序时，将自动化单元测试作为开发过程的一部分，通常被认为是最佳做法。接下来的几章，我们将讲述用于测试 Backbone.js 应用程序的 3 个不同解决方案：Jasmine、QUnit 以及 SinonJS。

13.1 行为驱动开发

本章，我们将了解一下，如何使用来自 Pivotal Labs 的流行 JavaScript 测试框架 Jasmine 来对 Backbone 应用程序进行单元测试。

Jasmine 将自己描述为：用于测试 JavaScript 代码的行为驱动开发（BDD）框架。在了解该框架如何工作之前，先准确了解一下 BDD 是非常有用的。

BDD 是第二代测试方法，首次由 BDD 的作者 Dan North 提出，Dan North 试图测试软件的行为。BDD 之所以被认为是第二代，是因为它融合了领域驱动设计（DDD）和精益软件开发的创意精髓。BDD 通过在早期的敏捷过程中回答很多特别令人困惑的问题，来帮助团队交付高质量的软件。这些问题包括那些与文档和测试有关的问题。

如果你读过关于 BDD 的书，你会发现它很可能已经被描述成压式（outside-in）和拉式（pull-based）。其原因是它借用了精益制造的拉式功能，有效确保了正确的软件解决方案是按照如下方式编写的：

- 专注于期望的系统输出；
- 确保这些输出是完成的。

BDD 认为，通常一个项目会有多个干系人，而不是一个单用户系统。这些不同的团体将会以不同的方式影响软件开发，对系统的理解有不同的意见。正因为如此，了解谁会为软件带来价值并带来什么样的价值是非常重要的。

最后，BDD 依赖于自动化。一旦定义了质量预期，团队将会在正常构建的解决方案功能上进行检查，并和他们的期望结果进行比较。为了有效地推动这一过程，必须要使用自动化。BDD 严重依赖规范测试的自动化，而 Jasmine 就是可以协助自动化的一个工具。

BDD 既有助于开发人员，也有助于非技术干系人：

- 更好地理解和描述要解决问题的模型；
- 使用非开发人员可以读懂的语言来解释测试用例；
- 专注于减少业务部门领域语言和编写技术代码之间的转换。

这就意味着，开发人员应该能够将 Jasmine 单元测试展示给项目干系人看（在一个高的层面上，多亏使用了共同的词汇），使得干系人能够在概念上理解该代码。

开发人员实现 BDD，通常是和另外一个称为测试驱动开发（TDD）的测试模式一起的。TDD 背后的主要思想是使用如下开发过程：

（1）编写单元测试，描述希望代码所支持的功能；

（2）查看这些失败的测试（因为此时还没有编写代码支持这些功能）；

（3）编写代码，以便通过测试；

（4）整理、重复并进行重构。

在本章，我们将使用 BDD（和 TDD）为 Backbone 应用程序编写单元测试。

我看过很多开发人员是在编码之后再编写单元测试来验证自己的代码行为。虽然这也可以，但是请注意，它可能是有缺陷的，比如你只测试了当前代码所支持的功能，而没有测试所需要的行为，从而无法完全解决这个问题。

13.2 suite、spec 以及 spie

使用 Jasmine 时，需要编写 suite 和规范（spec）。suite 基本上用于描述场景，而 spec 则用于描述在这些场景中能够做什么。

每个 spec 都是一个 JavaScript 函数，其描述为：调用一个 it()，并使用一个字符串描述和一个函数作为参数。该描述应该描述该特定单元代码应该展示的行为，记住 BDD 是非常有意义的。一个基本的 spec 示例如下：

```
it('should be incrementing in value', function(){
    var counter = 0;
    counter++;
});
```

在 spec 内部，如果没有设置行为的预期，它并不是特别有用。可以使用 expect()函数和期望匹配器（https://github.com/pivotal/jasmine/wiki/Matchers）在 spec 里定义行为预期——例如，toEqual()、toBeTruthy()、toContain()。修改后的代码使用了期望匹配器，示例如下：

```
it('should be incrementing in value', function(){
    var counter = 0;
    counter++;
    expect(counter).toEqual(1);
});
```

上述代码将我们的行为期望设置为 counter 等于 1。注意，最后一行是多么容易阅读（不需要其他任何解释就能理解）。

使用 Jasmine 的 describe()函数，再传入一个字符串描述和一个函数（就像为 it()定义的函数一样），就可以将 spec 进行分组，从而形成 suite。suite 的名称/描述通常是所要测试的组件或模块。

当对所运行的 spec 进行结果输出时，Jasmine 将使用描述作为分组的名称。包含样例 spec 的一个简单 suite，看起来像下面这样：

```
describe('Stats', function(){
    it('can increment a number', function(){
        ...
    });

    it('can subtract a number', function(){
        ...
    });
});
```

suite 还共享了函数作用域，所以 describe 块里还可以声明 spec 能够访问的变量和函数：

```
describe('Stats', function(){
    var counter = 1;

    it('can increment a number', function(){
        // the counter was = 1
        counter = counter + 1;
        expect(counter).toEqual(2);
    });

    it('can subtract a number', function(){
        // the counter was = 2
        counter = counter - 1;
        expect(counter).toEqual(1);
    });
});
```

suite 是按照描述的顺序来执行的。如果想将程序特定部分的测试结果作为第一部分报告的话，知道它就非常有用了。

Jasmine 还支持 spie——一种在单元测试中进行模拟（mock）、监视（spy）、行为作假（fake）的方法。spie 替换要监视的函数，允许我们模拟想要模拟的行为（例如，不使用实际的功能实现进行测试）。

在下面的例子中，监视虚拟 Todo 函数上的 setComplete 方法来测试传入的参数是否和预期的一致。

```
var Todo = function(){
};

Todo.prototype.setComplete = function (arg){
    return arg;
}
```

```
describe('a simple spy', function(){
    it('should spy on an instance method of a Todo', function(){
        var myTodo = new Todo();
        spyOn(myTodo, 'setComplete');
        myTodo.setComplete('foo bar');

        expect(myTodo.setComplete).toHaveBeenCalledWith('foo bar');

        var myTodo2 = new Todo();
        spyOn(myTodo2, 'setComplete');

        expect(myTodo2.setComplete).not.toHaveBeenCalled();

    });
});
```

你可能更想将 spie 使用在应用程序的异步行为测试上，例如 AJAX 请求。Jasmine 支持如下测试：

- 使用 spie 编写可以模拟 AJAX 请求的测试。这既能让我们测试带有初始化 AJAX 请求的代码，也能让我们测试 AJAX 请求结束以后才执行的代码，还可以对服务器上的响应进行模拟/作假。这种测试类型的好处是它没有真正调用服务器的速度。能够模拟任何服务器响应也有很大的好处。
- 支持不依赖 spie 的异步测试。

第一种测试类型在本例中展示了如何模拟一个 AJAX 请求，并且也验证了：该请求既调用了正确的 URL，也执行了提供的一个回调。

```
it('the callback should be executed on success', function () {

    // `andCallFake()` calls a passed function when a spy
    // has been called
    spyOn($, 'ajax').andCallFake(function(options) {
        options.success();
    });

    // Create a new spy
    var callback = jasmine.createSpy();

    // Execute the spy callback if the
    // request for Todo 15 is successful
    getTodo(15, callback);

    // Verify that the URL of the most recent call
    // matches our expected Todo item.
    expect($.ajax.mostRecentCall.args[0]['url']).toEqual('/todos/15');

    // `expect(x).toHaveBeenCalled()` will pass if `x` is a
```

```
    // spy and was called.
    expect(callback).toHaveBeenCalled();
});
function getTodo(id, callback) {
    $.ajax({
        type: 'GET',
        url: '/todos/'' + id,
        dataType: 'json',
        success: callback
    });
}
```

所有这些功能都是特定 spy 的匹配器，在 Jasmine 的 wiki 上都有记录。

对于测试的第二种类型（异步测试），我们可以使用 Jasmine 支持的其他三个方法对前面的测试进行进一步操作：

- **waits(*timeout*)**：运行下一个代码块之前的等候时间。
- **waitsFor(function, optional message, optional timeout)**：暂停 spec，直到其他一些工作已经完成。在移动到下一个代码块之前，Jasmine 会一直等候，直到提供的函数返回 true。
- **runs(*function*)**：代码块的运行就像直接被调用一样。由于它的存在，我们就可以测试异步流程了。

```
it('should make an actual AJAX request to a server', function () {

    // Create a new spy
    var callback = jasmine.createSpy();

    // Execute the spy callback if the
    // request for Todo 16 is successful
    getTodo(16, callback);

    // Pause the spec until the callback count is
    // greater than 0
    waitsFor(function() {
        return callback.callCount > 0;
    });

    // Once the wait is complete, our runs() block
    // will check to ensure our spy callback has been
    // called
    runs(function() {
        expect(callback).toHaveBeenCalled();

    });
});
```

```
function getTodo(id, callback) {
    $.ajax({
        type: 'GET',
        url: 'todos.json',
        dataType: 'json',
        success: callback
    });
}
```

在单元测试里对 Web 服务器使用真正的请求会大幅降低测试的速递（有很多因素，包括服务器延迟）——记住这一点是很有用的。因为它还引入了一个可以（而且应该）在单元测试中最小化的外部依赖项，在 Web 服务器请求上，强烈建议使用 spie 来删除该依赖。

13.3 beforeEach()和 afterEach()

Jasmine 还支持在运行每个测试之前（beforeEach()）或之后（afterEach()）运行指定的代码，这对于执行一致条件的测试来说非常有用（比如每个 spec 之前都要重置变量）。在下面的例子中，beforeEach()用于创建一个新的 Todo 示例模型，spec 可以用于测试其属性。

```
beforeEach(function(){
   this.todo = new Backbone.Model({
      text: 'Buy some more groceries',
      done: false
   });
});

it('should contain a text value if not the default value', function(){
   expect(this.todo.get('text')).toEqual('Buy some more groceries');
});
```

测试嵌套中的每个 describe()都可以拥有自己的 beforeEach()和 afterEach()方法，用于支持包括 setup 和 teardown 在内的特定 suite 中的方法。

可以将 beforeEach()和 afterEach()一起使用来编写测试，用于验证 Backbone 路由正确地触发我们所导航的 URL。可以先从 index 验证：

```
describe('Todo routes', function(){

   beforeEach(function(){

        // Create a new router
        this.router = new App.TodoRouter();

        // Create a new spy
```

```
        this.routerSpy = jasmine.spy();

        // Begin monitoring hashchange events
        try{
            Backbone.history.start({
                silent:true,
                pushState: true
            });
        }catch(e){
           // ...
        }

        // Navigate to a URL
        this.router.navigate('/js/spec/SpecRunner.html');
    });

    afterEach(function(){

        // Navigate back to the URL
        this.router.navigate('/js/spec/SpecRunner.html');

        // Disable Backbone.history temporarily.
        // Note that this is not really useful in real apps but is
        // good for testing routers
        Backbone.history.stop();
    });

    it('should call the index route correctly', function(){
        this.router.bind('route:index', this.routerSpy, this);
        this.router.navigate('', {trigger: true});

        // If everything in our beforeEach() and afterEach()
        // calls has been correctly executed, the following
        // should now pass.
        expect(this.routerSpy).toHaveBeenCalledOnce();
        expect(this.routerSpy).toHaveBeenCalledWith();
    });

});
```

能让上述测试通过的实际 TodoRouter，示例如下：

```
var App = App || {};
App.TodoRouter = Backbone.Router.extend({
    routes:{
        '': 'index'
    },
    index: function(){
        //...
    }
});
```

13.4 共享作用域

想象一下，我们有一个 suite 套件，希望检查是否存在一个新的 todo 项实例。可以通过像下面这样复制 spec 来实现这一目的：

```
    // Spec
    it("Should be defined when we create it", function(){
        // A Todo item we are testing
        var todo = new Todo("Get the milk", "Tuesday");
        expect(todo).toBeDefined();
    });

    it("Should have the correct title", function(){
        // Where we introduce code duplication
        var todo = new Todo("Get the milk", "Tuesday");
        expect(todo.title).toBe("Get the milk");
    });

});
```

正如你所看到的，我们所说的重复定义应该被重构成整洁的代码。可以使用 Jasmine 的 suite（共享的）函数作用域来达到这一目的。

同一个 suite 里的所有 spec 都共享相同的函数作用域，也就是说，suite 自身里定义的变量对该 suite 里所有的 spec 都可用。通过将 todo 对象的创建代码移动到公共作用域内，可以解决我们的重复问题：

```
describe("Todo tests", function(){

    // The instance of Todo, the object we wish to test
    // is now in the shared functional scope
    var todo = new Todo("Get the milk", "Tuesday");

    // Spec
    it("should be correctly defined", function(){
        expect(todo).toBeDefined();
    });

    it("should have the correct title", function(){
        expect(todo.title).toBe("Get the milk");
    });

});
```

在前一节中你可能已经注意到，在 beforeEach()的调用作用域内，我们初始化定义了 this.todo，然后就能继续在 afterEach()中使用这个引用。

这也是共享函数作用域的作用，对所有的代码块（包括 runs()）来说，允许这样的声明是常见的。

在共享作用域外部声明的变量（在局部作用域内 var todo =…）是不会被共享的。

13.5 准备开始

既然已经了解了一些基本知识，让我们下载 Jasmine 并且准备所有设置来编写测试。

可以从官方发布页面（https://github.com/pivotal/jasmine/downloads）下载一个独立的 Jasmine 版本。

除 Jasmine 之外，还需要一个名为 SpecRunner.html 的文件。可以从 GitHub 上下载，或者从 Jasmine 存储库下载完整的 Jasmine 版本，或者从 Jasmine 主版本上用 Git 克隆一个版本。

- SpecRunner.html：https://github.com/gruntjs/grunt-contrib-jasmine/blob/master/test/fixtures/pivotal/SpecRunner.html。
- Jasmine 完整版：https://github.com/pivotal/jasmine/zipball/master。
- Git 克隆：https://github.com/pivotal/jasmine。

让我们来看一下 SpecRunner.html.jst。

首先，它包括报告所需要的 Jasmine 和必要的 CSS：

```
<link rel="stylesheet" type="text/css"
 href="lib/jasmine-<%= jasmineVersion %>/jasmine.css">
<script src="lib/jasmine-<%= jasmineVersion %>/jasmine.js"></script>
<script src="lib/jasmine-<%= jasmineVersion %>/jasmine-html.js"></script>
<script src="lib/jasmine-<%= jasmineVersion %>/boot.js"></script>
```

接下来是要被测试的源码：

```
<!-- include source files here... -->
<script src="src/Player.js"></script>
<script src="src/Song.js"></script>
```

最后包含了一些样例测试：

```
<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
```

上述 SpecRunner 的这段代码可以用于运行实际的测试。

鉴于我们不会修改这段代码，我们就跳过它了。然而，我鼓励大家阅读 PlayerSpec.js 和 SpecHelper.js 来了解一下它。对于了解如何设置最小化的测试，这些是很有用的基本示例。

- PlayerSpec.js：https://github.com/pivotal/jasmine/blob/master/lib/jasmine-core/example/spec/PlayerSpec.js。
- SpecHelper.js：https://github.com/pivotal/jasmine/blob/master/lib/jasmine-core/example/spec/SpecHelper.js。

还请注意，出于介绍的目的，本章的一些示例将用于测试 Backbone.js 自身，只是为了让大家感受一下 Jasmine 是如何工作的。通常不需要对框架编写测试来确保其能按预期工作。

13.6 TDD 与 Backbone

使用 Backbone 开发应用程序时，可能不仅要测试核心模块，还要测试模型、视图、集合和路由。以 TDD 的方式进行测试，让我们看一下用于测试的 spec，对流行的 BackboneTodo 应用程序的组件进行测试。

13.7 模型

Backbone 模型的复杂性差异很大，其取决于应用程序所要达到的目标。在接下来的示例中，我们将测试模型的默认值、属性、状态变化以及验证规则。

首先，我们使用 describe()用于模型测试的 suite 定义：

```
describe('Tests for Todo', function() {
```

模型的属性应该具有默认值，这有助于确保实例被创建时，如果没有为任何属性设置值，可以使用默认值（如，一个空字符串）进行替代。这里的思想是不用任何意想不到的行为，允许应用程序和模型进行交互。

在下面的 spec 中，我们创建一个新的待办项，不传入任何属性，然后查找 text 属性的值是什么。因为没有设置值，所以期待返回一个默认值"（空字符串）。

```
it('Can be created with default values for its attributes.', function() {
    var todo = new Todo();
    expect(todo.get('text')).toBe('');
});
```

如果在编写模型之前测试这个 spec，将会得到预期的失败结果。要让这个 spec 通过测试，需要做的是给属性 text 设置一个默认值。在 Todo 模型里，可以像下面的示例这样，将这个默认值和其他一些有用的默认值（稍后将用到）设置好：

```
window.Todo = Backbone.Model.extend({

    defaults: {
      text: '',
      done:  false,
      order: 0
    }
```

其次，模型里包含验证逻辑，以确保应用程序里的用户输入或其他模块输入都有效，这是很常见的。

Todo 应用程序可能希望验证所提供的文本输入，以防止文本里包含粗鲁词语。同样，如果使用布尔值来保存待办项的 done 状态，也需要验证传入的值是否为真假值（true/false），而不是任意字符串。

下面的 spec 中利用了验证失败时 model.validate()就会触发一个 invalid 事件这一事实。这允许我们测试，一旦提供的输入无效，验证是否也能正常失败。

使用 Jasmine 内置的 createSpy()方法创建一个 errorCallback 监听，以便允许我们可以像下面的示例这样监听无效事件：

```
it('Can contain custom validation rules, and will trigger an invalid event on
failed validation.', function() {

    var errorCallback = jasmine.createSpy('-invalid event callback-');

    var todo = new Todo();

    todo.on('invalid', errorCallback);

    // What would you need to set on the todo properties to
    // cause validation to fail?

    todo.set({done:'a non-boolean value'});

    var errorArgs = errorCallback.mostRecentCall.args;

    expect(errorArgs).toBeDefined();
    expect(errorArgs[0]).toBe(todo);
    expect(errorArgs[1]).toBe('Todo.done must be a boolean value.');
});
```

让上述失败测试来支持验证的代码是相对简单的。在我们的模型中，我们覆盖了 validate()方法（Backbone 文档中推荐的），用于检查以确保模型中有一个 done 属

性，并且在接收传值时要确保它是一个有效的布尔值。

```
validate: function(attrs) {
    if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
        return 'Todo.done must be a boolean value.';
    }
}
```

如果要查看模型的最终代码，如下即是：

```
window.Todo = Backbone.Model.extend({

    defaults: {
      text: '',
      done:  false,
      order: 0
    },

    initialize: function() {
        this.set({text: this.get('text')}, {silent: true});
    },

    validate: function(attrs) {
        if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
            return 'Todo.done must be a boolean value.';
        }
    },

    toggle: function() {
        this.save({done: !this.get('done')});
    }

});
```

13.8 集合

现在需要为 Todo 模型集合（TodoList）来定义测试 spec 了。集合负责一定数量的列表任务，包括管理排序和过滤。

下面是我们在使用集合编写 spec 时会想到的一些具体的 spec：

- 新 Todo 模型作为对象或数组时，确保其可以添加进来。
- 属性测试，以确保集合的 URL 等基础属性是我们所期望的值。
- 有意添加 done:true 状态的模型，然后检查集合的项已完成多少、还剩余多少。

在本节中，我们将介绍前两个，第三个作为进行扩展的练习。

编写新 Todo 模型作为对象或数组时，确保其可以添加进来，这样的测试比较简单。首先，初始化一个新 TodoList 集合，并确保其长度（所包含 Todo 模型的数量）为 0。接下来，将新待办项同时作为对象和数组添加进来，在每个阶段检查集合上的 length 属性，确保整个计数是我们所期望的：

```
describe('Tests for TodoList', function() {

    it('Can add Model instances as objects and arrays.', function() {
        var todos = new TodoList();

        expect(todos.length).toBe(0);

        todos.add({ text: 'Clean the kitchen' });

        // how many todos have been added so far?
        expect(todos.length).toBe(1);

        todos.add([
            { text: 'Do the laundry', done: true },
            { text: 'Go to the gym'}
        ]);

        // how many are there in total now?
        expect(todos.length).toBe(3);
    });
...
```

类似于模型属性，集合属性的测试也是非常简单的。这里我们有一个规范，它确保了集合的 URL（服务器上集合位置的 URL 引用）是我们所期望的：

```
it('Can have a url property to define the basic url structure for all contained
        models.', function() {
        var todos = new TodoList();

        // what has been specified as the url base in our model?
        expect(todos.url).toBe('/todos/');
});
```

第三个 spec（作为练习进行编写）值要注意的是，集合实现里有一些过滤方法用于判断有多少 todo 项已经完成，多少还未完成，操作方法是调用 done()和 remaining()方法。编写 spec，可以先创建一个新集合，接着添加一个 done 属性为 true 状态的 todo 项，然后再添加两个 done 属性为 false 状态的 todo 项。测试使用 done()和 remaining()方法返回对象的长度，来告诉我们应用程序的状态管理是否正常工作，或者还需要一个小的调整。

TodoList 集合的最终实现如下所示：

```
window.TodoList = Backbone.Collection.extend({
```

```
    model: Todo,

    url: '/todos/',

    done: function() {
        return this.filter(function(todo) { return todo.get('done'); });
    },

    remaining: function() {
        return this.without.apply(this, this.done());
    },

    nextOrder: function() {
        if (!this.length) {
            return 1;
        }

        return this.last().get('order') + 1;
    },

    comparator: function(todo) {
        return todo.get('order');
    }

});
```

13.9 视图

在进入 Backbone 视图测试之前，让我们简要了解一个 jQuery 插件，它可以协助我们编写 Jasmine spec。

我们知道，Todo 应用程序将使用 jQuery 进行 DOM 操作，一个有用的 jQuery 插件叫 jasmine-jquery（https://github.com/velesin/jasmine-jquery），可以帮助我们简化视图渲染的 BDD 测试。

该插件提供了许多额外的 Jasmine 匹配器，来帮助测试 jQuery 包装的 spec，例如：

- **toBe(jQuerySelector)**：例如，expect($(' ')).toBe('div#some-id')。
- **toBeChecked()**：例如，expect($('<input type="checkbox" checked="checked"/>')).toBeChecked()。
- **toBeSelected()**：例如，expect($('<option selected="selected"></option>')).toBeSelected()。

还有其他很多匹配器。该插件所支持的匹配器完整列表可以在项目主页上找到。与标准的 Jasmine 匹配器类似，可以使用.not 前缀（例如，expect(x).not.toBe(y)）将列

出的这些自定义匹配器进行反转，知道这些非常有用：

```
expect($('<div>I am an example</div>')).not.toHaveText(/other/)
```

jasmine-jquery 还包括一个固定模块（fixture），在测试中，可以用于加载任意我们希望的 HTML 内容，包括外部文件中的 HTML，如 some.fixture.html：

```
<div id="sample-fixture">some HTML content</div>
```

接着，在实际的测试中，用如下方式加载它：

```
loadFixtures('some.fixture.html')
$('some-fixture').myTestedPlugin();
expect($('#some-fixture')).to<the rest of your matcher would go here>
```

jasmine-jquery 插件默认从 spec/javascripts/fixtures 目录中加载 fixtures。如果希望配置该路径，可以通过初始化设置jasmine.getFixtures().fixturesPath = 'your custom path' 来实现。

最后，jasmine-jquery 还包括支持在 jQuery 事件上进行监听（spy），而不需要任何额外的工作。可以使用 spyOnEvent()和 assert(eventName).toHaveBeenTriggered (selector)函数来做这个事情。例如：

```
spyOnEvent($('#el'), 'click');
$('#el').click();
expect('click').toHaveBeenTriggeredOn($('#el'));
```

下面我们将了解编写 Backbone 视图 spec 的 3 个维度：初始设置、视图渲染以及模板化。后两个都是最常见的测试，但是很快我们就会看到，在视图中为 spec 编写初始化操作也是可以带来好处的。

1. 初始设置

最基本的，Backbone 视图的 spec 应该验证其能够正确绑定到特定的 DOM 元素上，并且支持有效的数据模型。其原因是这些 spec 可以识别后续测试更加复杂的问题。同时，鉴于整体值已提供，编写这些内容是相当简单的。

为帮助确保 spec 有一致的测试设置，使用 beforeEach()在 DOM 上添加一个空 (#todoList)，并使用一个空的 Todo 模型初始化一个 TodoView 的实例。afterEach() 用来删除前面的#todoList 以及前面视图的实例。

```
describe('Tests for TodoView', function() {

    beforeEach(function() {
        $('body').append('<ul id="todoList"></ul>');
        this.todoView = new TodoView({ model: new Todo() });
    });
```

```
    afterEach(function() {
        this.todoView.remove();
        $('#todoList').remove();
    });

...
```

第一个编写的有用 spec 是一个检查器，用于检查我们创建的 TodoView 使用了正确的 tagName（元素或类名）。该测试的目的是确保视图创建以后能够正确绑定到一个 DOM 元素上。

Backbone 视图初始化时，通常会创建空的 DOM 元素，但是，这些元素并没有附加到可见的 DOM 上，以便让它们在构建时不受渲染性能的影响。

```
it('Should be tied to a DOM element when created, based off the property
    provided.', function() {
    //what html element tag name represents this view?
    expect(todoView.el.tagName.toLowerCase()).toBe('li');
});
```

再提一下，如果还没有开始编写 TodoView，我们会看到失败的 spec。幸好，解决这个问题很简单，创建一个带有特定 tagName 的 Backbone.View 即可。

```
var todoView = Backbone.View.extend({
    tagName:  'li'
});
```

如果要使用 className 替代 tagName 进行相反的测试，可以利用 jasmine-jquery 的 toHaveClass()匹配器：

```
it('Should have a class of "todos"'), function(){
   expect(this.view.$el).toHaveClass('todos');
});
```

toHaveClass()匹配器在 jQuery 对象上进行操作。如果没有使用该插件，将会抛出一个异常。如果不使用 jasmine-jquery，也可以通过访问 el.className 来测试 className。

大家可能已经注意到，在 beforeEach()里，给视图传入的是一个初始化的 Todo 模型（虽然未满）。视图应该由提供数据的模型实例来支持。这对于视图的功能是非常重要的，我们可以编写一个 spec 来确保模型一定要被定义（使用 toBeDefined()匹配器），然后测试模型的属性，以确保默认值存在，并且也都是我们所期望的。

```
it('Is backed by a model instance, which provides the data.', function() {

    expect(todoView.model).toBeDefined();

    // what's the value for Todo.get('done') here?
    expect(todoView.model.get('done')).toBe(false); // or toBeFalsy()
});
```

2. 视图渲染

接下来要来了解一下如何为视图渲染编写 spec。具体地说，我们想要测试 TodoView 元素是否可以按预期进行渲染。

在小型应用程序中，新 BDD 开发人员可能认为视图渲染的视觉确认可以取代视图的单元测试。事实是，当处理可能大量增加视图的应用程序时，从一开始就自动化这个过程是行得通的。除了屏幕上显示的视觉需要验证外，渲染方面的测试也需要验证（稍后很快就会看到）。

我们将通过编写两个 spec 开始测试视图。第一个 spec 将检查视图的 render()方法是否能正确返回该视图实例。对于链式操作来说，这是有必要的。第二个 spec 将检查 HTML 的生成是否和 TodoView 相关模型实例的属性所期望的一致。

不像先前我们讲述的部分 spec，本节将更多地使用 beforeEach()，用于演示如何使用嵌套 suite，以及确保 spec 有一致的条件。在第一个例子中，我们将简单地创建一个示例模型（基于 Todo）并使用它实例化一个 TodoView。

```
describe('TodoView', function() {

  beforeEach(function() {
    this.model = new Backbone.Model({
      text: 'My Todo',
      order: 1,
      done: false
    });
    this.view = new TodoView({model:this.model});
  });

  describe('Rendering', function() {

    it('returns the view object', function() {
      expect(this.view.render()).toEqual(this.view);
    });

    it('produces the correct HTML', function() {
      this.view.render();

      // let's use jasmine-jquery's toContain() to avoid
      // testing for the complete content of a todo's markup
      expect(this.view.el.innerHTML)
        .toContain('<label class="todo-content">My Todo</label>');
    });

  });

});
```

当运行这些 spec 时，只有第二个（produces the correct HTML）失败了。第一个 spec（returns the view object）用于测试 render()返回的 TodoView 实例，它通过了，是因为这是 Backbone 的默认行为，而且我们还没有使用自定义的版本来重载 render()方法。

为了保持可读性，本节的所有模板示例都将使用如下最小化的 todo 视图模板，因为扩展它相对简单。如果需要，请参考这个示例：

```
<div class="todo <%= done ? 'done' : '' %>">
        <div class="display">
          <input class="check" type="checkbox" <%= done ?
           'checked="checked"' : '' %> />
          <label class="todo-content"><%= text %></label>
          <span class="todo-destroy"></span>
        </div>
        <div class="edit">
          <input class="todo-input" type="text" value="<%= content %>" />
        </div>
</div>
```

第二个 spec 失败，抛出了如下信息：

```
Expected '' to contain '<label class="todo-content">My Todo</label>'.
```

其原因是 render()的默认行为并不创建任何标记。让我们编写一个 render()替代来修复这个问题：

```
render: function() {
  var template = '<label class="todo-content">+++PLACEHOLDER+++</label>';
  var output = template
    .replace('+++PLACEHOLDER+++', this.model.get('text'));
  this.$el.html(output);
  return this;
}
```

上述代码指定了一个内联字符串模板，模板中的+++PLACEHOLDER+++是要替换的字段，用于替换相关模型对应的值。在方法里，我们还返回了 TodoView 实例，所以第一个 spec 依然能通过。

讨论单元测试的时候不提到 fixtures 是不可能的。fixtures 通常包含测试数据（如 HTML），在单元测试需要时进行加载（可能是本地数据，也可能从外部文件获取的数据）。到目前为止，我们是基于视图的 el 属性来建立 jQuery 期望的。这种方法适用于一些测试，然而，在一些实例中，可能有必要将标记渲染到文档。处理这些 spec 的最优方式是通过使用 fixture（jasmine-jquery 给我们带来的另一个功能）来实现。

使用 fixture 重写上一个 spec，示例如下：

```
describe('TodoView', function() {

  beforeEach(function() {
    ...
    setFixtures('<ul class="todos"></ul>');
  });

  ...

  describe('Template', function() {

    beforeEach(function() {
      $('.todos').append(this.view.render().el);
    });

    it('has the correct text content', function() {
      expect($('.todos').find('.todo-content'))
        .toHaveText('My Todo');
    });

  });

});
```

我们在该 spec 中所做的是：将渲染后的 todo 项追加到 fixture 中，然后对这个 fixture 设置预期。当一个视图设置在 DOM 中已经存在的元素上时，这是理想的。我们会提供 fixture 并测试 el 属性，在视图初始化时准确地找到预期的元素。

3. 使用模板系统进行渲染（模板化）

当用户将一个 todo 项标记为已完成（done）时，我们可能希望提供一个视觉反馈（如，在文本上画一条线）来区分项目还有哪些未完成。可以通过给 todo 项添加一个 CSS 类来实现这一效果。先编写一个测试：

```
describe('When a todo is done', function() {

  beforeEach(function() {
    this.model.set({done: true}, {silent: true});
    $('.todos').append(this.view.render().el);
  });
  it('has a done class', function() {
    expect($('.todos .todo-content:first-child'))
      .toHaveClass('done');
  });

});
```

测试将会失败，并显示如下信息：

```
Expected '<label class="todo-content">My Todo</label>' to have class 'done'.
```

可以在现有的 render()方法里解决该问题，如下所示：

```
render: function() {
  var template = '<label class="todo-content">' +
    '<%= text %></label>';
  var output = template
    .replace('<%= text %>', this.model.get('text'));
  this.$el.html(output);
  if (this.model.get('done')) {
    this.$('.todo-content').addClass('done');
  }
  return this;
}
```

然而，这样做很快就会变得笨拙。由于模板中复杂性和逻辑的增加，因此对此做相关测试非常有挑战性。我们可以通过利用现代模板库来缓解这个过程，其中很多模板库已经被证实可以很好地处理像 Jasmine 这样的测试解决方案。

JavaScript 模板系统——如 Handlebars、Mustache 以及 Underscore 自身的微模板——在模板字符串中都支持条件逻辑。这实际上意味着，根据需要，我们可以在行内添加可执行的 if、else 以及三目表达式，以允许我们建立更强大的模板。

在我们的示例中，我们将使用 Underscore.js 里的微模板，因为没有额外的文件引入，使用微模板不需要费太多精力，就可以很容易地修改现有的 spec。

假设我们的模板是使用 id 为 myTemplate 的<script>标记来定义的：

```
<script type="text/template" id="myTemplate">
    <div class="todo <%= done ? 'done' : '' %>">
            <div class="display">
              <input class="check" type="checkbox"
              <%= done ? 'checked="checked"' : '' %> />
              <label class="todo-content"><%= text %></label>
              <span class="todo-destroy"></span>
            </div>
            <div class="edit">
              <input class="todo-input" type="text" value="<%= content %>" />
            </div>
    </div>
</script>
```

可以使用 Underscore 微模板将 TodoView 修改成如下代码：

```
var TodoView = Backbone.View.extend({

  tagName: 'li',
  template: _.template($('#myTemplate').html()),

  initialize: function(options) {
    // ...
```

```
    },

    render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      return this;
    },

    ...

  });
```

所以，这里都发生了什么？首先在自定义类型（如，type=text/template）的<script>标记里定义我们的模板。因为这不是一个任何浏览器都能够明白的脚本类型，所以只是简单忽略它；然而，通过 id 属性来引用该脚本，可以让模板与页面的其他部分隔离开来。

在视图中，使用 Underscore 的_.template()方法将我们的模板编译到一个函数中，以便稍后可以很容易将模型数据传进去。在 this.model.toJSON()代码行，我们只是简单地将模型的属性进行 JSON 字符串化，然后传给 template 方法，创建的 HTML 块现在可以附加到 DOM 上了。

注意，理想情况下，所有模板的逻辑应该存在于 spec 之外，要么是单独的模板文件，要么是通过<script>标记在 SpecRunner 里进行嵌入。这样通常更易于维护。

如果使用的都是很小的模板，并且不使用这种方式，还有一个有用的技巧，可以在 Jasmine 共享作用域内为每次测试进行自动化创建模板或扩展模板。

通过在 spec 文件夹内创建一个新目录（比如，templates），并包含一个含有如下内容的 SpecRunner.html 文件，我们可以手动添加自定义属性来表示我们希望使用的小模板：

```
beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
    todo: '<label class="todo-content">' +
            '<%= text %>' +
          '</label>'
  });
});
```

要完成这一关，简单地更新现有的 spec，在初始化 TodoView 时引用该模板：

```
describe('TodoView', function() {

  beforeEach(function() {
    ...
    this.view = new TodoView({
      model: this.model,
      template: this.templates.todo
    });
  });
```

```
    ...
});
```

使用这种方式，现有的 spec 测试依然会通过，让我们在使用 done 状态的 todo 项上使用一些附加条件逻辑，从而自由地调整模板：

```
beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
    todo: '<label class="todo-content <%= done ? 'done' : '' %>"' +
            '<%= text %>' +
          '</label>'
  });
});
```

这个 spec 没有任何问题，还是会通过测试。然而，正如前面提到的，只有在使用较小且高度动态的模板时，最后一种方式才可能行得通。

13.10 练习

作为练习，我建议看一下 practicals\jasminekoans 中的 Jasmine Koans，尝试修正一些故意失败的测试（而测试的这些功能又是必须有的）。这是了解 Jasmine spec 和 suite 工作方式的一种很好的方式，过一遍这些示例（不回头查看）也可以将学到的 Backbone 技能应用于测试。

13.11 延伸阅读

- Testing Backbone Apps with SinonJS（作者：James Newbery）

http://tinnedfruit.com/2011/04/26/testing-backbone-apps-with-jasmine-sinon-3.html

- Jasmine Backbone.js Revisited（作者：Chris Strom）

http://japhr.blogspot.com/2011/11/jasmine-backbonejs-revisited.html

- Phantom.js and Backbone.js (和 require.js)（作者：Chris Strom）

http://japhr.blogspot.com/2011/12/phantomjs-and-backbonejs-and-requirejs.html

13.12 总结

本章介绍了如何为 Backbone.js 模型、集合、视图编写 Jasmine 测试。测试路由有时也是可以的，一些开发者认为可以使用第三方工具（如 Selenium）进行优化。

第 14 章

QUnit

QUnit 是一个由 jQuery 团队成员 Jorn Zaefferer（http://bassistance.de/）编写的强大的 JavaScript 测试套件，被很多大型的开源项目（如 jQuery 和 Backbone.js）用于测试其代码。它既可以在浏览器中测试标准的 JavaScript 代码，也可以在服务器上测试代码（支持的环境包括 Rhino、V8 和 SpiderMonkey）。这使得它成为一个能够测试大量用例的强大解决方案。

很多 Backbone.js 贡献者认为，如果不需要马上用 Jasmine 和 BDD 进行项目测试，则 QUnit 是一个更好的基础性测试框架。在本章稍后，我们将会看到，QUnit 也可以结合第三方解决方案（如 SinonJS）产生一个更强大的测试解决方案，用于支持监听（spy）和模拟（mock），甚至有人说它优于 Jasmine。

我建议先比较一下这两个框架，然后选择一个我们觉得最好用的解决方案。

14.1 准备开始

幸好，QUnit 的开始设置是一个非常简单的过程，用不了 5 分钟。

首先，建立一个由 3 个文件构建的测试环境：

- 用于显示测试结果的 HTML 结构；
- 组成测试框架的 qunit.js；
- 样式化测试结果的 qunit.css。

后两个文件可以在 QUnit 网站（http://qunitjs.com/）进行下载。

如果愿意，可以使用托管版本的 QUnit 源文件来达到测试的目的。URL 托管地址可以在如下地址找到：https://github.com/jquery/qunit/。

兼容 QUnit 标记的样例 HTML 如下：

```
<!DOCTYPE html>
<html>
<head>
    <title>QUnit Test Suite</title>

     <link rel="stylesheet" href="qunit.css">
     <script src="qunit.js"></script>

     <!-- Your application -->
     <script src="app.js"></script>

     <!-- Your tests -->
     <script src="tests.js"></script>
</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">test markup, hidden.</ol>
</body>
</html>
```

让我们浏览一下 QUnit 所用到的带有 ID 的元素。当 QUnit 运行时：

- qunit-header 显示测试套件（suite）的名称。
- 如果测试失败，qunit-banner 显示为红色；如果测试通过，则显示为绿色。
- qunit-testrunner-toolbar 包含配置显示测试的附加选项。
- qunit-userAgent 显示 navigator.userAgent 属性。
- qunit-tests 是显示测试结果的一个容器。

正确运行 QUnit 时，上述测试运行器看起来如图 14-1 所示。

每个测试名称后面的（a,b,c）中的这些数字对应着：a）失败的断言数目，b）通过的断言数目，c）断言总数目。单击测试名称可以展开，可以显示该测试用例的所有断言。绿色的断言表示已经成功通过（见图 14-2）。

然而，如果测试失败，则测试结果会高亮显示（顶部的 QUnit 横幅会显示为红色，见图 14-3）。

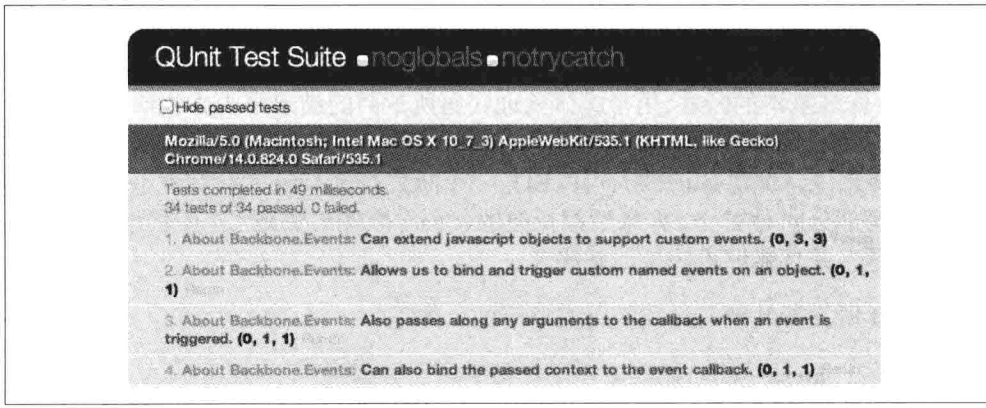


图 14-1　QUnit 测试运行器在浏览器中执行 Backbone 单元测试

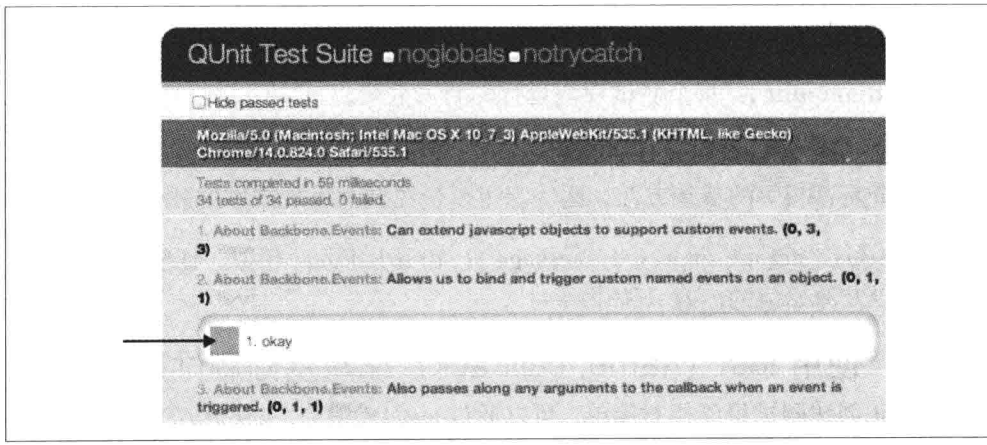


图 14-2　通过测试的断言显示为绿色标记

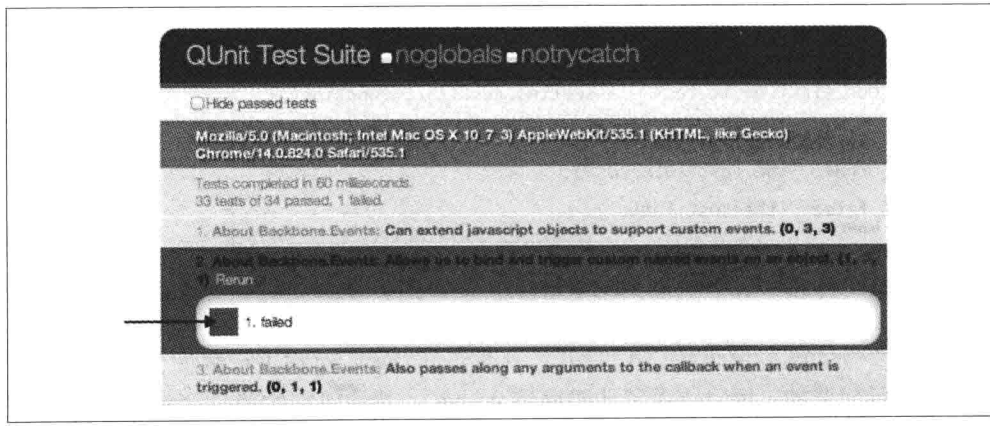


图 14-3　失败的测试将高亮显示为红色

14.2 断言

QUnit支持许多基本的断言，用于测试验证代码所返回的结果是否为我们所希望的。如果断言失败，就知道有Bug存在。和Jasmine类似，QUnit可以轻松地用于测试回归。具体来说，发现Bug的人可以编写一个断言测试Bug的存在性，编写补丁，然后将这两部分同时提交。如果后续更改的代码导致测试失败了，我们就会知道出了什么问题，并且能够更轻松地解决它。

一些QUnit所支持的断言是我们首先要看的，如下：

- **ok (state, message)**：如果第一个参数为真，则通过。
- **equal (actual, expected, message)**：使用强制类型进行比较的简单断言。
- **notEqual (actual, expected, message)**：和equal()相反。
- **expect(amount)**：每个测试期望运行的断言数量。
- **strictEqual(actual, expected, message)**：提供了一个比equal()更严格的比较，是检查是否相等的首选方法，因为它可以避免偶然发现微妙的硬性Bug。
- **deepEqual(actual, expected, message)**：和strictEqual相似，比较给定对象、数组以及原始值的内容（使用===）。

14.2.1 使用test（name, callback）编写基础测试用例

使用QUnit创建测试用例相对简单，可以通过test()来编写。test()构造了一个测试，其第一个参数是用于显示的测试名称，第二个参数是一个包含所有断言的callback函数，该函数在QUnit运行的时候就会被调用。

```
var myString = 'Hello Backbone.js';

test( 'Our first QUnit test - asserting results', function(){

    // ok( boolean, message )
    ok( true, 'the test succeeds');

    ok( false, 'the test fails');

    // equal( actualValue, expectedValue, message )
    equal( myString, 'Hello Backbone.js', 'Expected value: Hello Backbone.js!');
});
```

这里我们要做的就是使用特定的值定义一个变量，然后对它进行测试，以确保它的值是我们所预期的。我们使用了比较断言equal()，其希望它的第一个参数是测试中的值，第二个参数则是所预期的值。我们也使用ok()，它允许我们可以轻易地对函

数或变量进行布尔转换的测试。

在我们的测试用例中，我们可以给 test()传入一个预期值，来测试我们期望运行的断言数量。形式是：test(name, [expected], test)；或者通过在 test 函数的最顶部手动设置期望值，类似：expect(1)。建议大家养成总是定义期望断言数这样的习惯。稍后有更多关于这方面的内容。

14.2.2 比较函数的实际输出和期望输出

由于测试简单的静态变量非常容易，我们可以对实际的功能进行进一步测试。在接下来的例子中，我们使用 equal()和 notEqual()测试一个字符串反转函数的输出，以确保该输出正确：

```
function reverseString( str ){
    return str.split('').reverse().join('');
}
test( 'reverseString()', function() {
    expect( 5 );
    equal( reverseString('hello'), 'olleh', 'The value expected was olleh' );
    equal( reverseString('foobar'), 'raboof', 'The value expected was raboof' );
    equal( reverseString('world'), 'dlrow', 'The value expected was dlrow' );
    notEqual( reverseString('world'), 'dlroo', 'The value was expected to not
    be dlroo' );
    equal( reverseString('bubble'), 'double', 'The value expected was elbbub' );
})
```

在 QUnit 测试运行器（HTML 测试页面加载的时候可以看到）上运行这些测试，我们会发现前 4 个断言通过了，而最后一个没有通过。原因是对“double”的测试失败了，因为它是故意写错的。在自己的项目中，如果断言正确，但测试不通过，可能就发现了一个 Bug！

14.3 为断言添加结构

如果所有的断言都放在一个测试用例里，断言很快就会变得难以维护，不过幸运的是，QUnit 支持构建更清晰的断言块。可以使用 module()方法来达到这一目的，它可以很轻松地让我们对测试用例进行分组。典型的分组方法是将一个特定方法的多个测试作为同一个组（module）的一部分。

14.3.1 QUnit 基本模块

```
module( 'Module One' );
test( 'first test', function() {} );
test( 'another test', function() {} );

module( 'Module Two' );
```

```
test( 'second test', function() {} );
test( 'another test', function() {} );

module( 'Module Three' );
test( 'third test', function() {} );
test( 'another test', function() {} );
```

通过给模块引入 setup()和 teardown()回调，可以进行进一步操作。setup()是在所有测试运行之前运行，而 teardown()是在所有测试运行之后运行。

14.3.2 使用 setup()和 teardown()

```
module( 'Module One', {
    setup: function() {
        // run before
    },
    teardown: function() {
        // run after
    }
});

test('first test', function() {
    // run the first test
});
```

这些回调函数可以用来定义(或清理)任何在测试用例中我们想初始化使用的组件。我们很快就能看到，这是定义项目中的视图、集合、模型以及路由实例的理想之地，以便可以在多个测试中进行引用。

14.3.3 使用 setup()和 teardown()用于初始化和清理工作

```
// Define a simple model and collection modeling a store and
// list of stores

var Store = Backbone.Model.extend({});

var StoreList = Backbone.Collection.extend({
    model: Store,
    comparator: function( Store ) { return Store.get('name') }
});

// Define a group for our tests
module( 'StoreList sanity check', {
    setup: function() {
        this.list = new StoreList;
        this.list.add(new Store({ name: 'Costcutter' }));
        this.list.add(new Store({ name: 'Target' }));
        this.list.add(new Store({ name: 'Walmart' }));
        this.list.add(new Store({ name: 'Barnes & Noble' }));
    },
    teardown: function() {
        window.errors = null;
```

```
    }
});

// Test the order of items added
test( 'test ordering', function() {
    expect( 1 );
    var expected = ['Barnes & Noble', 'Costcutter', 'Target', 'Walmart'];
    var actual = this.list.pluck('name');
    deepEqual( actual, expected, 'is maintained by comparator' );
});
```

这里，在 setup()里创建一个商店列表并保存。teardown()回调仅用于简单地清除可能会在 window 作用域上保存的列表错误，不需要做其他任何工作。

14.4 断言示例

在进一步之前，让我们来看更多的例子，以便了解编写测试用例时如何正确使用 QUnit 的各种断言。

1. equal

比较断言。如果 actual == expected 为真，则表示通过。

```
test( 'equal', 2, function() {
  var actual = 6 - 5;
  equal( actual, true,  'passes as 1 == true' );
  equal( actual, 1,     'passes as 1 == 1' );
});
```

2. notEqual

比较断言。如果 actual != expected 为真，则表示通过。

```
test( 'notEqual', 2, function() {
  var actual = 6 - 5;
  notEqual( actual, false, 'passes as 1 != false' );
  notEqual( actual, 0,     'passes as 1 != 0' );
});
```

3. strictEqual

比较断言。如果 actual === expected 为真，则表示通过。

```
test( 'strictEqual', 2, function() {
  var actual = 6 - 5;
  strictEqual( actual, true,  'fails as 1 !== true' );
  strictEqual( actual, 1,     'passes as 1 === 1' );
});
```

4. notStrictEqual

比较断言。如果 actual !== expected 为真，则表示通过。

```
test('notStrictEqual', 2, function() {
  var actual = 6 - 5;
  notStrictEqual( actual, true,  'passes as 1 !== true' );
  notStrictEqual( actual, 1,     'fails as 1 === 1' );
});
```

5. deepEqual

递归比较断言。和 strictEqual()不一样，它可以对对象、数组和原始值进行断言。

```
test('deepEqual', 4, function() {
  var actual = {q: 'foo', t: 'bar'};
  var el =  $('div');
  var children = $('div').children();

  equal( actual, {q: 'foo', t: 'bar'},   'fails - objects are not equal
   using equal()' );
  deepEqual( actual, {q: 'foo', t: 'bar'},
  'passes - objects are equal' );
  equal( el, children, 'fails - jQuery objects are not the same' );
  deepEqual(el, children, 'fails - objects not equivalent' );

});
```

6. notDeepEqual

比较断言。返回 deepEqual 的相反面。

```
test('notDeepEqual', 2, function() {
  var actual = {q: 'foo', t: 'bar'};
  notEqual( actual, {q: 'foo', t: 'bar'},   'passes - objects are not equal' );
  notDeepEqual( actual, {q: 'foo', t: 'bar'},   'fails - objects are
  equivalent' );
});
```

7. raises

测试回调函数是否抛出任何异常的断言。

```
test('raises', 1, function() {
  raises(function() {
    throw new Error( 'Oh no! It`s an error!' );
  }, 'passes - an error was thrown inside our callback');
});
```

14.5 Fixtures

有时候，我们需要编写修改 DOM 的测试用例。管理测试操作中的这些清理工作是

非常痛苦的，不过多亏了 QUnit 有解决这个问题的解决方案。Qunit 以#qunit-fixture 的形式来解决：

```
<!DOCTYPE html>
<html>
<head>
    <title>QUnit Test</title>
    <link rel="stylesheet" href="qunit.css">
    <script src="qunit.js"></script>
    <script src="app.js"></script>
    <script src="tests.js"></script>
</head>
<body>
    <h1 id="qunit-header">QUnit Test</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-fixture"></div>
</body>
</html>
```

我们可以选择在 fixture 里放入静态标记，也可以选择将所需要的 DOM 元素插入或附加进去。在每个测试结束后，QUnit 会自动将 fixture 的 innerHTML 重置为原来的值。如果使用 jQuery，知道 QUnit 可以检查其可用性并且使用它的$(el). html() 是非常有用的，而且也可以清除所有的 jQuery 事件处理程序。

现在让我们通过使用 fixtures 来完成一个更完整的例子。大多数人都会在 jQuery 中做的一件事是——使用通常用于定义菜单标记、表格以及很多其他组件。在以特定方式操作给定的列表之前，可以使用 jQuery 插件，它可以帮助测试该插件最后的（操作）输出是否为我们所期望的。

为了完成下一个示例，我们将使用 Ben Alman 的$.enumerate()插件，它可以通过其索引对每个 item 项进行预处理，也可以选择性地允许我们来决定该列表中第一项的数字是什么。使用该插件的代码片段可以在如下找到，后面紧接着是其生成的输出结果：

```
$.fn.enumerate = function( start ) {
      if ( typeof start !== 'undefined' ) {
        // Since `start` value was provided, enumerate and return
        // the initial jQuery object to allow chaining.

        return this.each(function(i){
          $(this).prepend( '<b>' + ( i + start ) + '</b> ' );
        });

      } else {
```

```
        // Since no `start` value was provided, function as a
        // getter, returning the appropriate value from the first
        // selected element.

        var val = this.eq( 0 ).children( 'b' ).eq( 0 ).text();
        return Number( val );
      }
    };

/*
    <ul>
      <li>1. hello</li>
      <li>2. world</li>
      <li>3. i</li>
      <li>4. am</li>
      <li>5. foo</li>
    </ul>
*/
```

现在让我们为插件编写一些测试。首先，在 qunit-fixture 元素内，定义一个包含一些示例 item 的列表标记：

```
<div id="qunit-fixture">
    <ul>
      <li>hello</li>
      <li>world</li>
      <li>i</li>
      <li>am</li>
      <li>foo</li>
    </ul>
 </div>
```

接着，需要想想应该测试什么内容。$.enumerate()支持几个不同的用例，其中包括：

- 不传入任何参数：`$(el).enumerate()`。
- 传入参数 0：`$(el).enumerate(0)`。
- 传入参数 1：`$(el).enumerate(1)`。

每个列表项 item 的文本值是 n.item-text 的形式，仅仅需要对预期的输出进行测试即可，我们可以使用$(el).eq(index).text()直接访问它的内容（更多信息请参考 http://api.jquery.com/eq/）。

最后，下面是我们的测试用例：

```
module('jQuery#enumerate');

test( 'No arguments passed', 5, function() {
  var items = $('#qunit-fixture li').enumerate(); // 0
  equal( items.eq(0).text(), '0. hello', 'first item should have index 0' );
```

```
  equal( items.eq(1).text(), '1. world', 'second item should have index 1' );
  equal( items.eq(2).text(), '2. i', 'third item should have index 2' );
  equal( items.eq(3).text(), '3. am', 'fourth item should have index 3' );
  equal( items.eq(4).text(), '4. foo', 'fifth item should have index 4' );
});

test( '0 passed as an argument', 5, function() {
  var items = $('#qunit-fixture li').enumerate( 0 );
  equal( items.eq(0).text(), '0. hello', 'first item should have index 0' );
  equal( items.eq(1).text(), '1. world', 'second item should have index 1' );
  equal( items.eq(2).text(), '2. i', 'third item should have index 2' );
  equal( items.eq(3).text(), '3. am', 'fourth item should have index 3' );
  equal( items.eq(4).text(), '4. foo', 'fifth item should have index 4' );
});

test( '1 passed as an argument', 3, function() {
  var items = $('#qunit-fixture li').enumerate( 1 );
  equal( items.eq(0).text(), '1. hello', 'first item should have index 1' );
  equal( items.eq(1).text(), '2. world', 'second item should have index 2' );
  equal( items.eq(2).text(), '3. i', 'third item should have index 3' );
  equal( items.eq(3).text(), '4. am', 'fourth item should have index 4' );
  equal( items.eq(4).text(), '5. foo', 'fifth item should have index 5' );
});
```

14.6 异步代码

正如 Jasmine 一样，使用 QUnit 运行异步测试所需要做的工作还是相当少的。也就是说，需要异步回调函数的测试（比如昂贵的处理过程、Ajax 请求等）是怎样的？处理异步代码时，下一个测试用例的运行不是让 QUnit 来控制的，而是告诉 QUnit 需要停止运行，直到异步代码 OK 了才可以继续运行。

记住：运行异步代码时，如果没有任何特殊注意事项，可能导致不正确的断言在其他测试中出现，所以要确保异步代码是正确的。

为异步代码编写 QUnit 测试，可以通过 start()和 stop()方法来实现，它们以编程方式在这样的测试中设置开始点和结束点。下面是一个简单的例子：

```
test('An async test', function(){
   stop();
   expect( 1 );
   $.ajax({
        url: '/test',
        dataType: 'json',
        success: function( data ){
            deepEqual(data, {
               topic: 'hello',
               message: 'hi there!''
            });
```

```
            ok(true, 'Asynchronous test passed!');
            start();
        }
    });
});
```

上述代码中的 jQuery ajax()请求是用于连接到 test 资源上，并断言返回的数据是正确的。这里使用了 deepEqual()，是因为它允许我们比较不同的数据类型（例如，对象、数组），并确保返回的数据正是我们所希望的。我们知道 Ajax 请求是异步的，所以我们首先调用了 stop()，然后代码发送请求，最后，在回调快结束的时候，通知 QUnit 继续运行其他的测试。

与其使用 stop()，不如简单排除它，并使用 asyncTest()代替 test()来进行异步代码的测试。当在套件（suite）中混合处理异步和同步测试时，可以提高其可读性。而这个设置适应于很多用例，但不保证$.ajax()请求里的回调会被调用。把这个因素考虑到我们的测试中，我们可以再次使用 expect()来定义在测试中我们期望看到多少个断言。这是一个健康保障，因为它能够保证，如果测试结束时断言数量不足，我们就知道出问题了，然后就可以修复它。

第 15 章

SinonJS

和使用 Jasmine BDD 框架测试 Backbone.js 应用章节类似，我们已经可以利用所学的知识，为 Todo 应用程序编写一些 QUnit 测试了。

然而，在开始之前，你可能已经注意到 QUnit 不支持测试监听（spie）。测试监听是一种函数，用于记录其调用的参数、异常和返回值。它们通常用来测试回调函数，以及所要测试的函数在应用程序中是如何使用的。在测试框架中，监听（spie）通常是匿名函数或已存在函数的包装。

15.1 SinonJS 概述

为了能在 QUnit 中支持监听测试，我们将利用一个名为 SinonJS 的模拟框架（http://sinonjs.org/），它由 Christian Johansen 所创建。我们还将使用 SinonJS-QUnit 适配器（http://sinonjs.org/qunit/），该适配器对 QUnit 提供了无缝集成（即最小化设置）。SinonJS 是一个完全与测试框架无关的框架，应该很容易和其他测试框架一起使用，所以它非常符合我们的需要。

该框架支持 3 种功能，我们将在程序单元测试中利用它们：

- 匿名监听（Anonymous spies）；
- 在现有方法上监听（Spying on existing methods）；
- 丰富的检测接口（A rich inspection interface）。

15.1.1 基础 spy

使用 this.spy()不带任何参数，可以创建一个匿名 spy。这与 jasmine.createSpy()类似。在下面的示例中，可以观察 SinonJS 的 spy 基本用法：

```
test('should call all subscribers for a message exactly once', function () {
    var message = getUniqueString();
    var spy = this.spy();

    PubSub.subscribe( message, spy );
    PubSub.publishSync( message, 'Hello World' );

    ok( spy.calledOnce, 'the subscriber was called once' );
});
```

15.1.2 在现有函数上监听

在下面的示例中，我们也可以使用 this.spy()在现有函数（如 jQuery 的$ajax）上进行监听。当在现有函数上进行监听时，函数的行为不变，但我们得到对其进行调用的数据，出于测试目的，这是非常有用的。

```
test( 'should inspect the jQuery.getJSON usage of jQuery.ajax', function () {
    this.spy( jQuery, 'ajax' );

    jQuery.getJSON( '/todos/completed' );

    ok( jQuery.ajax.calledOnce );
    equals( jQuery.ajax.getCall(0).args[0].url, '/todos/completed' );
    equals( jQuery.ajax.getCall(0).args[0].dataType, 'json' );
});
```

15.1.3 检测接口

SinonJS 配有丰富的检测接口，以允许我们测试一个 spy 是否使用特定的参数进行了调用，判断是否调用了特定的次数，并测试比较参数的值。在 SinonJS.org 网站上，可以找到关于检测接口所支持的完整列表，但让我们看看一些例子，这些例子展示了一些最常用的检测接口。

(1) 匹配参数：测试一个 spy 是否使用一组特定参数进行了调用。

```
test( 'Should call a subscriber with standard matching': function () {
    var spy = sinon.spy();

    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', { id: 45 } );
    assertTrue( spy.calledWith( { id: 45 } ) );
});
```

(2) 严格参数匹配：测试一个 spy 至少使用一次特定参数进行了调用，并且没有其他参数。

```
test( 'Should call a subscriber with strict matching': function () {
    var spy = sinon.spy();

    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', 'many', 'arguments' );
    PubSub.publishSync( 'message', 12, 34 );

    // This passes
    assertTrue( spy.calledWith('many') );

    // This however, fails
    assertTrue( spy.calledWithExactly( 'many' ) );
});
```

(3) 测试调用顺序：测试一个 spy 是否在另外一个 spy 之前或之后进行了调用。

```
test( 'Should call a subscriber and maintain call order': function () {
    var a = sinon.spy();
    var b = sinon.spy();

    PubSub.subscribe( 'message', a );
    PubSub.subscribe( 'event', b );

    PubSub.publishSync( 'message', { id: 45 } );
    PubSub.publishSync( 'event', [1, 2, 3] );

    assertTrue( a.calledBefore(b) );
    assertTrue( b.calledAfter(a) );
});
```

(4) 匹配执行计数：测试一个 spy 是否调用了特定次数。

```
test( 'Should call a subscriber and check call counts', function () {
    var message = getUniqueString();
    var spy = this.spy();

    PubSub.subscribe( message, spy );
    PubSub.publishSync( message, 'some payload' );


    // Passes if spy was called once and only once.
    ok( spy.calledOnce ); // calledTwice and calledThrice are also supported

    // The number of recorded calls.
    equal( spy.callCount, 1 );

    // Directly checking the arguments of the call
    equals( spy.getCall(0).args[0], message );
});
```

15.2 stub 与 mock

SinonJS 还支持另外两个强大的功能：stub 和 mock。stub 和 mock 都实现了 spy 接口的所有功能，而且还附加了一些功能。

15.2.1 stub

stub 允许我们将特定方法的任何现有行为替换成其他的东西。stub 对于模拟异常特别有用，并且当代码的特定依赖还没有编写时，它经常用于编写测试用例。

让我们简要提炼一下 Backbone Todo 应用程序，其中包含一个 Todo 模型和一个 TodoList 集合。对于本场景，我们想孤立 TodoList 集合，伪造一个 Todo 模型来测试添加新模型的行为是怎么样的。

我们可以假装模型还没有开始编写，仅仅是为了演示 stub 是如何被执行的。只包含模型引用的集合，示例如下：

```
var TodoList = Backbone.Collection.extend({
    model: Todo
});

// Let's assume our instance of this collection is
this.todoList;
```

假设集合自己正在实例化新模型，为了测试，我们有必要 stub 该模型的构造函数。可以像如下代码一样，创建一个简单的 stub：

```
this.todoStub = sinon.stub( window, 'Todo' );
```

上述代码在 window 对象上创建了一个 Todo 方法的 stub。当 stub 一个持久化对象时，必须将其恢复到原来的状态。我们可以像下面这样，在 teardown()里做这个工作：

```
this.todoStub.restore();
```

在这之后，我们需要修改构造函数的返回，这里可以有效使用 Backbone.Model 的构造函数。虽然它不是一个 Todo 模型，但仍然可以为我们提供一个真实的 Backbone 模型。

```
setup: function() {
    this.model = new Backbone.Model({
      id: 2,
      title: 'Hello world'
    });
    this.todoStub.returns( this.model );
});
```

这里的期望是确保这段代码可以让我们的 TodoList 总是可以实例化一个 stub 过的 Todo 模型，但因为集合中引用的模型已经存在了，我们需要如下步骤来重置集合的模型属性：

```
this.todoList.model = Todo;
```

这样做的结果是，当 TodoList 集合实例化新的 Todo 模型时，它将返回我们期望的 Backbone 模型实例，这将允许我们可以像下面这样，为新的模型字面量编写测试：

```
module( 'Should function when instantiated with model literals', {

  setup:function() {

    this.todoStub = sinon.stub(window, 'Todo');
    this.model = new Backbone.Model({
      id: 2,
      title: 'Hello world'
    });

    this.todoStub.returns(this.model);
    this.todos = new TodoList();

    // Let's reset the relationship to use a stub
    this.todos.model = Todo;

    // add a model
    this.todos.add({
      id: 2,
      title: 'Hello world'
    });
  },

  teardown: function() {
    this.todoStub.restore();
  }

});

test('should add a model', function() {
    equal( this.todos.length, 1 );
});

test('should find a model by id', function() {
    equal( this.todos.get(5).get('id'), 5 );
  });
});
```

15.2.2 mock

mock 实际上和 stub 一样，但是它们模拟了完整的 API，并且有一些有关如何使用

它们的内置预期。mock 和 spy 的区别是，使用 mock 时的预期是预先定义的，如果测试结果不满足，测试就会失败。

下面是一个基于 PubSubJS 的 mock 用法示例。在这里，使用 clearTodo()方法作为回调函数，并使用 mock 来验证它的行为。

```
test('should call all subscribers when exceptions', function () {
    var myAPI = { clearTodo: function () {} };

    var spy = this.spy();
    var mock = this.mock( myAPI );
    mock.expects( 'clearTodo' ).once().throws();

    PubSub.subscribe( 'message', myAPI.clearTodo );
    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', undefined );

    mock.verify();
    ok( spy.calledOnce );
});
```

15.3 练习

现在可以开始为 Todo 应用程序编写测试了，这次测试将按组件进行分离（例如，模型、集合等）。有必要注意一下测试的名称、测试的逻辑以及最重要的断言，因为这将启发我们如何将所学知识应用到完整的应用程序中。

为了更好地理解这一节内容，推荐查看 practicals/qunit-koans 文件夹里的 QUnit Koans——它是从 Backbone.js Jasmine Koans 转到 QUnit 的一个入口点。

如果你还没有机会尝试使用 Koans 工具包，你就可以把它们当成练习来完成。Koans 工具包是一组使用特定测试框架的单元测试，既演示了应用程序的测试如何编写，也留下了一些未完成的测试。

15.3.1 模型

对于模型，我们至少要测试如下内容：

- 可以创建带有预期默认的新实例。
- 属性可以正确地进行设置和检索。
- 改变状态时，在需要时能够正确地触发自定义事件。

- 验证规则能够正确执行。

```
module( 'About Backbone.Model');

test('Can be created with default values for its attributes.', function() {
    expect( 3 );

    var todo = new Todo();
    equal( todo.get('text'), '' );
    equal( todo.get('done'), false );
    equal( todo.get('order'), 0 );
});

test('Will set attributes on the model instance when created.', function() {
    expect( 1 );

    var todo = new Todo( { text: 'Get oil change for car.' } );
    equal( todo.get('text'), 'Get oil change for car.' );

});

test('Will call a custom initialize function on the model instance when
    created.', function() {
    expect( 1 );

    var toot = new Todo
      ({ text: 'Stop monkeys from throwing their own crap!' });
    equal( toot.get('text'),
      'Stop monkeys from throwing their own rainbows!' );
});

test('Fires a custom event when the state changes.', function() {
    expect( 1 );

    var spy = this.spy();
    var todo = new Todo();

    todo.on( 'change', spy );
    // Change the model state
    todo.set( { text: 'new text' } );

    ok( spy.calledOnce, 'A change event callback was correctly triggered' );
});


test('Can contain custom validation rules, and will trigger an invalid
    event on failed validation.', function() {
    expect( 3 );

    var errorCallback = this.spy();
    var todo = new Todo();

    todo.on('invalid', errorCallback);
```

```
    // Change the model state in such a way that validation will fail
    todo.set( { done: 'not a boolean' } );

    ok( errorCallback.called, 'A failed validation correctly triggered an
    error' );
    notEqual( errorCallback.getCall(0), undefined );
    equal( errorCallback.getCall(0).args[1], 'Todo.done must be a boolean
    value.' );

});
```

15.3.2 集合

对于集合，我们要测试如下内容：

- 集合有一个 Todo 模型。
- 使用 localStorage 做同步。
- done()、remaining()以及 clear()能够按预期工作。
- Todo 项的排序能够正常数字化。

```
describe('Test Collection', function() {

  beforeEach(function() {

    // Define new todos
    this.todoOne = new Todo;
    this.todoTwo = new Todo({
      title: "Buy some milk"
    });

    // Create a new collection of todos for testing
    return this.todos = new TodoList([this.todoOne, this.todoTwo]);
  });

  it('Has the Todo model', function() {
    return expect(this.todos.model).toBe(Todo);
  });

  it('Uses localStorage', function() {
    return expect(this.todos.localStorage).toEqual(new Store
    ('todos-backbone'));
  });

  describe('done', function() {
    return it('returns an array of the todos that are done', function() {
      this.todoTwo.done = true;
      return expect(this.todos.done()).toEqual([this.todoTwo]);
    });
  });
```

```
  describe('remaining', function() {
    return it('returns an array of the todos that are not done', function() {
      this.todoTwo.done = true;
      return expect(this.todos.remaining()).toEqual([this.todoOne]);
    });
  });

  describe('clear', function() {
    return it('destroys the current todo from localStorage', function() {
      expect(this.todos.models).toEqual([this.todoOne, this.todoTwo]);
      this.todos.clear(this.todoOne);
      return expect(this.todos.models).toEqual([this.todoTwo]);
    });
  });

  return describe('Order sets the order on todos ascending numerically',
    function() {
    it('defaults to one when there arent any items in the collection',
    function() {
      this.emptyTodos = new TodoApp.Collections.TodoList;
      return expect(this.emptyTodos.order()).toEqual(0);
    });

    return it('Increments the order by one each time', function() {
      expect(this.todos.order(this.todoOne)).toEqual(1);
      return expect(this.todos.order(this.todoTwo)).toEqual(2);
    });
  });

});
```

15.3.3 视图

对于视图，我们要确保如下内容：

- 视图创建时可以正确地绑定到 DOM 元素上。
- 视图可以进行渲染，渲染之后，视图的 DOM 显示应该是可见的。
- 支持将视图方法链接到 DOM 元素上。

我们还可以更深入一些，在模型中测试用户是否有正确视图结果的交互，便于将这些需要修改的视图进行正确更新。

```
module( 'About Backbone.View', {
    setup: function() {
        $('body').append('<ul id="todoList"></ul>');
        this.todoView = new TodoView({ model: new Todo() });
    },
    teardown: function() {
```

```
        this.todoView.remove();
        $('#todoList').remove();
    }
});

test('Should be tied to a DOM element when created, based off the property
provided.', function() {
    expect( 1 );
    equal( this.todoView.el.tagName.toLowerCase(), 'li' );
});

test('Is backed by a model instance, which provides the data.', function() {
    expect( 2 );
    notEqual( this.todoView.model, undefined );
    equal( this.todoView.model.get('done'), false );
});

test('Can render, after which the DOM representation of the view will be
visible.', function() {
   this.todoView.render();

    // Append the DOM representation of the view to ul#todoList
    $('ul#todoList').append(this.todoView.el);

    // Check the number of li items rendered to the list
    equal($('#todoList').find('li').length, 1);
});

asyncTest('Can wire up view methods to DOM elements.', function() {
    expect( 2 );
    var viewElt;

    $('#todoList').append( this.todoView.render().el );

    setTimeout(function() {
        viewElt = $('#todoList li input.check').filter(':first');

        equal(viewElt.length > 0, true);

        // Ensure QUnit knows we can continue
        start();
    }, 1000, 'Expected DOM Elt to exist');
    // Trigger the view to toggle the 'done' status on an item or items
    $('#todoList li input.check').click();

    // Check the done status for the model is true
    equal( this.todoView.model.get('done'), true );
});
```

15.3.4 App

App 对于应用程序中放置的启动代码编写测试，也是非常有用的。对于下面的模块，进行 setup 初始化，并附加一个 TodoApp 视图，从视图的局部示例是否被正确定义，

到应用程序的交互是否能够在本地集合实例中做出相应的改变，我们可以测试任何东西。

```
module( 'About Backbone Applications' , {
    setup: function() {
        Backbone.localStorageDB = new Store('testTodos');
        $('#qunit-fixture').append('<div id="app"></div>');
        this.App = new TodoApp({ appendTo: $('#app') });
    },

    teardown: function() {
        this.App.todos.reset();
        $('#app').remove();
    }
});

test('Should bootstrap the application by initializing the Collection.',
function() {
    expect( 2 );

    // The todos collection should not be undefined
    notEqual( this.App.todos, undefined );

    // The initial length of our todos should however be zero
    equal( this.App.todos.length, 0 );
});

test( 'Should bind Collection events to View creation.' , function() {

      // Set the value of a brand new todo within the input box
      $('#new-todo').val( 'Buy some milk' );

      // Trigger the enter (return) key to be pressed inside #new-todo
      // causing the new item to be added to the todos collection
      $('#new-todo').trigger(new $.Event( 'keypress', { keyCode: 13 } ));

      // The length of our collection should now be 1
      equal( this.App.todos.length, 1 );
 });
```

15.4 延伸阅读与资源

这就是关于使用 QUnit 和 SinonJS 测试应用程序的内容。我鼓励大家尝试使用 QUnit Backbone.js Koans（https://github.com/addyosmani/backbone-koans-qunit），并看看是否可以扩展一些例子。要进一步阅读，可以考虑下面的这些额外资源：

- **Test-Driven JavaScript Development (book)**

http://tddjs.com/

- **SinonJS/QUnit adapter**

http://sinonjs.org/qunit/

- **Using Sinon.JS with QUnit**

http://cjohansen.no/en/javascript/using_sinon_js_with_qunit

- **Automating JavaScript Testing with QUnit**

http://msdn.microsoft.com/en-us/magazine/gg749824.aspx

- **Unit Testing with QUnit**

http://benalman.com/talks/unit-testing-qunit.html

- **Another QUnit/Backbone.js demo project**

https://github.com/jc00ke/qunit-backbone

- **SinonJS helpers for Backbone**

https://supportbee.com/devblog/2012/02/10/helpers-for-testing-backbone-js-apps-using-jasmine-and-sinon-js/

第 16 章

结　论

希望大家已经发现本书对 Backbone.js 的介绍是有价值的。希望你学到的知识是：仅使用 DOM 操作库（如 jQuery）来构建偏 JavaScript 的应用程序当然是一个可行的技巧，但没有任何正式的程序结构，很难构建卓越的应用程序。成堆地嵌套 jQuery 回调和 DOM 元素是不太可能做成具有规模的应用程序的，并且随着应用程序的增长，它会变得非常难以维护。

Backbone.js 的优势是它的简单。如果你开始研究 Backbone.js 源码的话，会发现用很少的代码提供其功能和灵活性是显而易见的。用 Jeremy Ashkenas 的话说："Backbone 的核心前提是一直尝试和发现用最小集合的基础数据结构（模型和集合）和用户界面（视图和 URL）。这在使用 JavaScript 构建 Web 应用程序时非常有用。" Backbone.js 就是帮助你提高应用程序的结构，更好地分离关注点。没有什么事情比这更重要。

Backbone 提供了用键/值对绑定的模型、事件，拥有很丰富的 API 枚举方法的集合，能进行事件处理的声明式视图，以及一种可以通过 RESTful JSON 接口将现有 API 链接到客户端程序上的方式。使用这些功能可以将数据抽象到理智的模型，将 DOM 操作抽象到视图，仅仅使用事件就可以将它们绑定在一起。

任何一个在 JavaScript 应用程序上开发过一段时间的开发人员，如果他重视架构和可维护性，最终都将创建一个类似的解决方案。选择使用它或其他类似的东西取决于你自己——通常是一个将一组不会在一起工作的库混合在一起使用的过程。在抽象组织代码或测试代码时，可以将 jQuery BBQ 用于历史管理，且使用 Handlebars 用于模板管理。

与 Backbone 相比，Backbone 拥有源码文档，还有一个用户和黑客都参与的繁荣社

区，而且像 Stack Overflow 这样的站点上每天还有大量的问题被解答。与其做重复工作，不如使用一个基于整个社区集体知识和经验的解决方案，从中获取构建应用程序的优势。

除了有助于为应用程序提供健全的结构外，Backbone 还是高度可扩展的，支持更多所需的自定义架构（超越于 Backbone 的默认架构）。很明显，去年发布了大量的扩展和插件，其中一些我们已经谈到了（如 MarionetteJS 和 Thorax）。

最近一段时间，我们使用 Backbone.js 创建了很多复杂的 Web 应用程序，从 LinkedIn 移动应用到 NewsBlur 这样的流行 RSS 阅读器（http://newsblur.com/），以及社交评价插件 Disqus（http://disqus.com/）。这个简单但又理智抽象的小框架协助我们创建了新一代富 Web 应用程序，并且我和我的合作者希望随着时间的推移，它也可以帮助到你。

如果不知道是否值得在项目上使用 Backbone，那就要问问自己，你构建的程序有多复杂，值不值得用。你能突破你的能力限制来组织你的代码吗？编写的应用程序，在不需要请求服务器刷新页面的情况下，UI 是否会有规律地变化？你想从关注点分离中受益吗？如果是，像 Backbone 这样的解决方案也许能够帮助到你。

谷歌的 Gmail 经常作为一个构建良好的单页面程序示例被提到。如果你用过它，可能已经注意到，它请求了一个很大的初始代码块，包括了大部分用户需要的 JavaScript、CSS 和 HTML，在这之后，所有额外的需求都是在后台发生的。不需要重新渲染整个页面，Gmail 就可以很容易地从收件箱切换到垃圾邮件文件夹。像 Backbone 这样的库，可以帮助 Web 开发人员得到这样的体验。

也就是说，要构建的应用，如果不值得为它学习相关的库，那么 Backbone 就不能提供帮助了。如果应用程序或网站仍然需要使用服务器做繁重的建设，并且给浏览器提供整个页面的话，我们可能会发现，使用简单的 JavaScript 或 jQuery 获得简单效果或进行交互操作反而更简单。花时间评估一下什么样的情况适合用 Backbone，为每一个项目做出正确的选择。

Backbone 既不难学也不难用，学习使用它构建应用程序所花的时间和精力是值得的。虽然阅读本书可以让你基本上能了解整个库，但是最好的学习方式仍然是尝试构建一下自己的真实应用程序。希望你能发现，最终的产品是更整洁、更有组织、更易于维护的代码。

然后，希望大家在前往 Backbone 世界的旅途中一切顺利，并赠送给大家美国作家 Henry Miller 的一句话：“一个人的目的地绝不仅仅是一个地点，而是一种观察世界的新眼光。”

附录 A

延伸学习

A.1 简单的 JavaScript MVC 实现

全面讨论 Backbone 的实现超出了本书的范围，然而，我们可以实现一个简单的 MVC 库（取名为 Cranium.js）来说明像 Backbone 这样的框架式是如何实现 MVC 模式的。

像 Backbone 一样，对于继承和模板处理，我们将依靠 Underscore。

A.1.1 事件系统

JavaScript MVC 实现的核心是基于发布订阅模式的事件系统（对象），它能够让 MVC 组件以优雅、解耦的方式进行通信。订阅者监听感兴趣的特定事件，并且在发布者广播这些事件的时候做出响应。

事件是混合到视图和模型组件中的，以便这些组件的实例发布感兴趣的事件。

```
// cranium.js - Cranium.Events

var Cranium = Cranium || {};

// Set DOM selection utility
var $ = document.querySelector.bind(document) || this.jQuery || this.Zepto;

// Mix in to any object in order to provide it with custom events.
var Events = Cranium.Events = {
  // Keeps list of events and associated listeners
  channels: {},

  // Counter
  eventNumber: 0,
```

```
  // Announce events and passes data to the listeners;
  trigger: function (events, data) {
    for (var topic in Cranium.Events.channels){
      if (Cranium.Events.channels.hasOwnProperty(topic)) {
        if (topic.split("-")[0] == events){
          Cranium.Events.channels[topic](data) !== false ||
          delete Cranium.Events.channels[topic];
        }
      }
    }
  },
  // Registers an event type and its listener
  on: function (events, callback) {
    Cranium.Events.channels[events + --Cranium.Events.eventNumber] = callback;
  },
  // Unregisters an event type and its listener
  off: function(topic) {
    delete Cranium.Events.channels[topic];
  }
};
```

事件系统将实现如下功能：

- 视图通知用户交互（如单击或者表单输入）的订阅者，然后更新/重新渲染其展示，等等。
- 模型数据改变时，通知订阅者更新自身（例如，更新视图用于展示准确/更新后的数据），等等。

A.1.2 模型

模型为应用程序管理（领域特定的）数据。它们既不关心用户界面，也不关心表示层，而只是表示应用程序所需要的结构化数据。当模型变化（比如更新）时，它通常会通知其观察者（订阅者）自己发生了一个变化，以便它们可以做出相应的反应。

让我们看一个简单模型的实现：

```
// cranium.js - Cranium.Model

// Attributes represents data, model's properties.
// These are to be passed at Model instantiation.
// Also we are creating id for each Model instance
// so that it can identify itself (e.g., on chage
// announcements)
var Model = Cranium.Model = function (attributes) {
    this.id = _.uniqueId('model');
    this.attributes = attributes || {};
};
```

```
// Getter (accessor) method;
// returns named data item
Cranium.Model.prototype.get = function(attrName) {
    return this.attributes[attrName];
};

// Setter (mutator) method;
// Set/mix in into model mapped data (e.g.{name: "John"})
// and publishes the change event
Cranium.Model.prototype.set = function(attrs){
    if (_.isObject(attrs)) {
      _.extend(this.attributes, attrs);
      this.change(this.attributes);
    }
    return this;
};

// Returns clone of the Models data object
// (used for view template rendering)
Cranium.Model.prototype.toJSON = function(options) {
    return _.clone(this.attributes);
};

// Helper function that announces changes to the Model
// and passes the new data
Cranium.Model.prototype.change = function(attrs){
    this.trigger(this.id + 'update', attrs);
};

// Mix in Event system
_.extend(Cranium.Model.prototype, Cranium.Events);
```

A.1.3 视图

视图是模型的可视化表示，展示的是当前状态的筛选视图。一个视图通常要观察一个模型，并且在模型更改时得到通知，以允许视图相应地对自己进行更新。设计模式通常认为视图是一个“哑巴”，是由于它们对应用程序中的模型和控制器的了解有限。

让我们使用一个简单的 JavaScript 例子进一步探索视图：

```
// DOM View
var View = Cranium.View = function (options) {
  // Mix in options object (e.g., extending functionality)
  _.extend(this, options);
  this.id = _.uniqueId('view');
};
// Mix in Event system
_.extend(Cranium.View.prototype, Cranium.Events);
```

A.1.4 控制器

控制器是模型和视图之间的一个中介者，通常负责完成以下两项任务：

- 当模型改变时，负责更新视图；
- 用户操作视图时，负责更新模型。

```
// cranium.js - Cranium.Controller

// Controller tying together a model and view
var Controller = Cranium.Controller = function(options){
  // Mix in options object (e.g extending functionality)
  _.extend(this, options);
  this.id = _.uniqueId('controller');
  var parts, selector, eventType;

  // Parses Events object passed during the definition of the
  // controller and maps it to the defined method to handle it;
  if(this.events){
    _.each(this.events, function(method, eventName){
      parts = eventName.split('.');
      selector = parts[0];
      eventType = parts[1];
      $(selector)['on' + eventType] = this[method];
    }.bind(this));
  }
};
```

A.1.5 实际用法

下面是初级用户使用的 HTML 模板：

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
</head>
<body>
<div id="todo">
</div>
  <script type="text/template" class="todo-template">
    <div>
      <input id="todo_complete" type="checkbox" <%= completed %>>
      <%= title %>
    </div>
  </script>
  <script src="underscore-min.js"></script>
  <script src="cranium.js"></script>
  <script src="example.js"></script>

</body>
</html>
```

Cranium.js 的用法：

```
// example.js - usage of Cranium MVC

// And todo instance
var todo1 = new Cranium.Model({
    title: "",
    completed: ""
});

console.log("First todo title - nothing set: " + todo1.get('title'));
todo1.set({title: "Do something"});
console.log("Its changed now: " + todo1.get('title'));
''
// View instance
var todoView = new Cranium.View({
  // DOM element selector
  el: '#todo',

  // Todo template; Underscore templating used
  template: _.template($('.todo-template').innerHTML),

  init: function (model) {
    this.render( model.toJSON() );

    this.on(model.id + 'update', this.render.bind(this));
  },
  render: function (data) {
    console.log("View about to render.");
    $(this.el).innerHTML = this.template( data );
  }
});

var todoController = new Cranium.Controller({
  // Specify the model to update
  model: todo1,

  // and the view to observe this model
  view:  todoView,

  events: {
    "#todo.click" : "toggleComplete"
  },

  // Initialize everything
  initialize: function () {
    this.view.init(this.model);
    return this;
  },
  // Toggles the value of the todo in the Model
  toggleComplete: function () {
```

```
        var completed = todoController.model.get('completed');
        console.log("Todo old 'completed' value?", completed);
        todoController.model.set({ completed: (!completed) ? 'checked': '' });
        console.log("Todo new 'completed' value?",
        todoController.model.get('completed'));
        return this;
      }
    });


    // Let's start things off
    todoController.initialize();

    todo1.set({ title: "Due to this change Model will notify View and
    it will rerender"});
```

Backbone.js 第一个版本的作者——Samuel Clay 评价过 Cranium.js：“不出所料，它看起来完全就像 Backbone 的开始。视图是哑的，所以它们很少有样板及设置。模型负责模型的属性和变更通知。”

我希望大家能发现，这个实现能有助于解释如何才能从头开始编写像 Backbone 这样的库，但更多的是，它能鼓励大家利用成熟的现有解决方案做出选择，而不必深究它们到底能做什么。

A.2 MVP

模型-视图-表示器（MVP）是 MVC 设计模式的一种派生，主要用于改进表示逻辑。它起源于 20 世纪 90 年代一个名叫 Taligent 的公司，当时该公司正在致力于 C++ CommonPoint 环境的模型工作。尽管 MVC 和 MVP 的目标都是跨越多组件的关注点分离，但还是有一些基本的差异。

为了进行总结，我们将关注最适合 Web 架构的 MVP 版本。

MVP 中的 P 代表表示器。它是一个为了视图而包含用户界面业务逻辑的组件。与 MVC 不同，它将视图中的调用委托给表示器，从而从视图中解耦出来，并且通过一个接口进行通信。这将允许在单元测试中做任何有用的事情，比如模拟视图。

MVP 最常见的实现是被动视图（总而言之是一种哑巴视图），几乎不包含逻辑。MVP 的模型和 MVC 的模型几乎一样，包括处理应用程序数据的方式。表示器扮演了一个中介者的角色，在视图和模型之间进行沟通；但是，视图和模型是彼此隔离的。表示器将模型有效绑定到视图，该责任在 MVC 中由控制器负责。表示器是 MVP 模型的核心，大家也能猜到，它包含了视图背后的表示逻辑。

表示器收到视图请求时，执行用户请求所需要的任何工作，并将处理结果返回给用户。在这方面，表示器检索数据，操作数据，然后判断数据在视图中应该如何显示。在一些实现中，表示器还与保存数据（模型）的服务层进行交互。模型用于触发事件，但向模型订阅事件却是表示器的职责，以便其自身可以更新视图。在这种被动的架构中，没有直接数据绑定的概念。视图暴露设置器，表示器可以用它来设置数据。

从 MVC 中进行演进，这种变化的好处是它能够提高应用程序的可测试性，并在视图和模型之间提供一个清晰的分离。这并不是没有代价的，由于在模式中缺乏数据绑定的支持，这就意味着不得不经常单独关注这个事情。

尽管被动视图的一个常见实现是让视图来实现接口，但还有其他的变种，其中包括使用事件将视图从表示器中解耦出来。因为 JavaScript 中没有接口，我们越来越多地将表示器作为一个协议而不是显式接口进行使用。在技术上，它依然是一个 API，从这个角度来看，将它称之为接口也是很公平的。

MVP 还有一个变异版本叫监视控制器（Supervising Controller），它和 MVC 以及 MVVM——模型（M）、视图（V）、视图模型（VM）——模式比较接近，因为它直接从视图中提供了模型绑定的功能。键/值对观察（KVO）插件（如 DerickBailey 的 Backbone.ModelBinding 插件）给 Backbone 引入了监视控制器这个理念。

A.3 MVP 还是 MVC

MVP 一般最常在企业级应用上使用，在这些应用上，有必要尽可能多地重用表示器逻辑。应用程序如果有非常复杂的视图和大量的用户交互，可能会发现 MVC 其实并不适合这里，因为要解决这个问题，可能意味着要严重依赖多个控制器。在 MVP 里，所有的复杂逻辑都可以封装在一个表示器内，这样可以大大简化维护工作。

由于 MVP 的视图是通过一个接口来定义的，而该接口从技术上来说，它是系统和视图（除了表示器）之间的唯一联络点。MVP 模式还允许开发人员直接为应用程序编写表示逻辑，而无需等待设计师设计的布局和图片。

根据不同的实现，MVP 的自动化单元测试可能比 MVC 更容易。经常说这个观点的原因是表示器可以作为一个用户界面的完整模拟对象，所以他可以独立于其他组件进行单元测试。根据我的经验，这实际上要取决于你实现 MVP 的语言（这对一个 JavaScript 项目的 MVP 实现来说有很大的不同，比如 ASP.NET）。

在结束的时候，MVC 可能适用于 MVP 方面，大家可能有潜在的担忧，因为它们

之间的区别主要是语义上的。只要能将模型、视图、控制器（或表示器）进行清晰的分离关注点，不管选择哪一个模式，都能获得相同的好处。

A.4 MVC、MVP 以及 Backbone.js

很少有 JavaScript 框架宣称自己能实现 MVC 或 MVP 的经典模式，因为很多 JavaScript 开发人员认为 MVC 和 MVP 不是互斥的（我们实际上可以看到，在 ASP.NET 或 GWT 这样的 Web 框架上才更可能看到 MVP 的严格实现）。这是因为，在应用程序里可能有一些额外的表示器或视图逻辑，但仍认为这是一种 MVC 风格。

Backbone 的贡献者 Irene Ros（http://ireneros.com/）接受了这种思维方式，因为当她将 Backbone 的视图分离到自己单独的组件中时，她需要一种方式将这些组件组装起来。它可以是一个控制器路由（如，Backbone.Router），或者获取数据响应后的一个回调。

据说，一些开发人员觉得 Backbone.js 更适合 MVP 的描述，而不是 MVC。他们的观点是：

- 与控制器相比，MVP 中的表示器能更好地描述 Backbone.View（视图模板和所绑定视图的数据之间的层）；
- 模型适合 Backbone.Model（和经典的 MVC 模型没什么不同）；
- 视图最好地描述了模板（比如，Handlebars/Mustache 标记模板）。

针对上述观点的一个回应可能是，视图也可以仅仅只是一个视图（根据 MVC），因为 Backbone 足够灵活，可以让视图用于多种目的。MVC 中的 V 和 MVP 中的 P 的工作，都可以由 Backbone.View 来完成的，因为它们都能实现两个目的：两者都渲染原子组件，并且组装其他视图渲染过的组件。

我们还看到，在 Backbone 里，控制器的职责由 Backbone.View 和 Backbone.Router 两者一起共享。在下面的例子中，我们可以看到它们真正负责的方面。

在这里，BackboneTodoView 使用了观察者模式，在视图模型的 this.model.on ('change',...)上订阅了更改事件。在 render()方法里还处理了模型操作，但是与其他实现不同，用户交互也是在视图中进行处理的（参见 events）。

```
// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName:  'li',

  // Pass the contents of the todo template through a templating
  // function, cache it for a single todo
  template: _.template( $('#item-template').html() ),

  // The DOM events specific to an item.
  events: {
    'click .toggle':  'togglecompleted'
  },

  // The TodoView listens for changes to its model, rerendering. Since there's
  // a one-to-one correspondence between a **Todo** and a **TodoView** in this
  // app, we set a direct reference on the model for convenience.
  initialize: function() {
    this.model.on( 'change', this.render, this );
    this.model.on( 'destroy', this.remove, this );
  },

  // Rerender the titles of the todo item.
  render: function() {
    this.$el.html( this.template( this.model.toJSON() ) );
    return this;
  },

  // Toggle the `"completed"` state of the model.
  togglecompleted: function() {
    this.model.toggle();
  },
});
```

另一种（完全不同的）观点是，Backbone 更像我们前面介绍的 smalltalk - 80 MVC（http://martinfowler.com/eaaDev/uiArchs.html#ModelViewController）。

MarionetteJS 的作者 Derick Bailey（http://lostechies.com/derickbailey/2011/12/23/backbone-js-is- not-an-mvc-framework/）曾经写到：最好不要将 Backbone 强迫套进任何一个特定设计模式。设计模式应被看作是灵活的应用指南，用于指导如何将应用程序结构化，而在这方面，Backbone 不管是对 MVC 还是 MVP 都不完美。它从多个架构模式中借鉴了一些最好的概念，并创建了一个很好用的灵活框架。称它为 Backbone 方式、MV *或者任何其他有助于体现其应用程序架构风格的名字。

然而，需要理解这些概念的起源在哪里，以及为什么会有这些概念。所以希望我对 MVC 和 MVP 所做的解释能对你有所帮助。大多数结构化的 JavaScript 框架都会在经典模式上有意或偶然地适配自己的模式，但重要的是，它们能帮助我们开发有组

织的、整洁的、易于维护的应用程序。

A.5 命名空间

在学习如何使用 Backbone 时，大家会发现教程中一个重要的且经常被忽视的领域：命名空间（namespacing）。如果在 JavaScript 中你已经有命名空间的经验了，下面部分将在如何具体应用你所知道的 Backbone 概念上提供一些建议。但是，对初学者来说，我也将覆盖解释一下，以确保每个人都在同一水平。

A.5.1 什么是命名空间

使用命名空间是一种可以避免与全局命名空间中的其他对象或变量产生冲突的方法。使用命名空间可以减少代码遭受同一页面使用相同变量的其他脚本破坏的潜在可能性。作为全局命名空间的一个好公民，你必须尽量减少代码被其他开发人员的脚本破坏的可能性。

对于命名空间，JavaScript 并没有像其他语言一样有真正的内置支持，但是有闭包，可以用来实现一个类似效果。

本节我们将看看如何对模型、视图、路由以及其他组件使用命名空间的例子。要看的模式如下：

- 单一全局变量（single global variables）；
- 对象字面量（object literals）；
- 嵌套命名空间（nested namespacing）。

1. 单一全局变量

JavaScript 中使用命名空间的一个流行模式是：选择一个全局变量作为引用的主对象。下面是一个简单实现，这里我们返回了一个包含函数和属性的对象：

```
var myApplication = (function(){
    function(){
      // ...
    },
    return {
      // ...
    }
})();
```

大家之前可能见过这种技术。Backbone 风格的例子看起来可能像这样：

```
var myViews = (function(){
    return {
        TodoView: Backbone.View.extend({ .. }),
        TodosView: Backbone.View.extend({ .. }),
        AboutView: Backbone.View.extend({ .. });
        //etc.
    };
})();
```

这里我们可以返回一组视图，但同样的技术也可以返回一个包含模型、视图和路由的单个对象，这取决于如何结构化应用程序。虽然这种方法适用于特定情况，但单一全局变量模式面临的最大挑战是：要确保在同一页面上，没有人像你一样使用同名的单一全局变量。

该问题的其中一个解决方案，如 Peter Michaux 所讲，可以使用命名空间前缀。这是一个简单的概念，但其思想是选择一个常见的前缀名称（本例为 myapplication_），然后在前缀后面定义任何方法、变量或其他对象。

```
var myApplication_todoView = Backbone.View.extend({}),
    myApplication_todosView = Backbone.View.extend({});
```

从在全局作用域内降低特定变量重名的几率这个角度来说，这是有效的，但记住唯一命名的对象具有相同的效果。另外，这种模式最大的问题是：随着应用程序的增长，会导致产生大量的全局对象。

关于单一全局变量模式更多关于 Peter 的观点，请阅读他的精品文章，地址如下：http://michaux.ca/articles/javascript-namespacing。

有几个关于单一全局变量模式的变体。但是，了解多个变体后，我觉得在 Backbone 上使用加前缀的方法是最好的。

2. 对象字面量

对象字面量的优点是不会污染全局命名空间，并且逻辑上能协助组织代码和参数。如果想创建简单易读的结构，以便可以扩展为支持深层嵌套的话，使用对象字面量是非常有用的。与单一全局变量不同，对象字面量也经常考虑测试同名变量的存在，这有助于减少重名的机会。

下面的例子演示了两种方式，可以用于判断命名空间在定义之前是否已经存在。我通常使用第二种方式。

```
/* Doesn't check for existence of myApplication */
var myApplication = {};

/*
Does check for existence. If already defined, we use that instance.
Option 1:   if(!myApplication) myApplication = {};
Option 2:   var myApplication = myApplication || {};
We can then populate our object literal to support models, views, and collections
(or any data, really):
*/

var myApplication = {
    models : {},
    views : {
        pages : {}
    },
    collections : {}
};
```

也可以选择在现有的命名空间上直接添加属性（比如，下面示例中的视图）：

```
var myTodosViews = myTodosViews || {};
myTodosViews.todoView = Backbone.View.extend({});
myTodosViews.todosView = Backbone.View.extend({});
```

这种模式的好处是：用一种方式就可以轻易地封装所有的模型、视图、路由等，以便将它们清晰地分离，并为扩展代码提供一个坚实的基础。

这种模式有许多好处。将应用程序的默认配置解耦到一个单独的区域，以便可以很容易地修改，而不需要为了修改它而搜索整个代码库，这往往是一个好办法。下面这个例子，假设一个对象字面量保存了应用程序的配置：

```
var myConfig = {
  language: 'english',
  defaults: {
    enableDelegation: true,
    maxTodos: 40
  },
  theme: {
    skin: 'a',
    toolbars: {
      index: 'ui-navigation-toolbar',
      pages: 'ui-custom-toolbar'
    }
  }
}
```

请注意，对象字面量和 JSON 数据集其实只有一些微小语法上的差异。如果出于任何原因，我们希望使用 JSON 来存储配置（例如，向后端发送请求时的简单数据保存），可以自由选择。

更多关于对象文本模式的信息，建议阅读 Rebecca Murphey 的优秀文章，地址：http://rmurphey.com/blog/2009/10/15/using-objects-to-organize-your-code。

3. 嵌套命名空间

对象字面量的一个扩展是嵌套命名空间。它是另外一种常用的模式，它有较低的冲突风险，即便有顶级命名空间已经存在这样的事实，它也不可能嵌套同样的子命名空间。例如，Yahoo 的 YUI 广泛使用了嵌套对象命名空间模式：

```
YAHOO.util.Dom.getElementsByClassName('test');
```

DocumentCloud（Backbone 的创建者）甚至在其主要应用程序中使用了嵌套命名空间模式。使用 Backbone 的嵌套命名空间的简单示例，看起来像下面这样：

```
var todoApp =  todoApp || {};

// perform similar check for nested children
todoApp.routers = todoApp.routers || {};
todoApp.model = todoApp.model || {};
todoApp.model.special = todoApp.model.special || {};

// routers
todoApp.routers.Workspace   = Backbone.Router.extend({});
todoApp.routers.TodoSearch = Backbone.Router.extend({});

// models
todoApp.model.Todo   = Backbone.Model.extend({});
todoApp.model.Notes = Backbone.Model.extend({});

// special models
todoApp.model.special.Admin = Backbone.Model.extend({});
```

对于给 Backbone 应用进行命名空间设置来说，这是可读的、组织清晰的和相对安全的方式。唯一真正需要注意的是，它需要浏览器的 JavaScript 引擎首先找到 todoApp 对象，然后再向下找，一直找到要调用的函数。而且，像 Juriy Zaytsev 这样的开发人员，对单一全局变量方式和嵌套方式进行了并发和性能测试，发现两者之间的差别是可以忽略的。

A.5.2 DocumentCloud 用的是什么

如果大家有什么疑惑，下面是 DocumentCloud（还记得创建 Backbone 的这群家伙么？）的原始工作区在必要时所使用的命名空间。这种方法是行得通的，因为公司的文档（以及注释和文档列表）是嵌入在第三方新站点上的。

```
// Provide top-level namespaces for our javascript.
(function() {
  window.dc = {};
```

```
    dc.controllers = {};
    dc.model = {};
    dc.app = {};
    dc.ui = {};
})();
```

正如你所看到的，DocumentCloud 选择在 window 上声明一个顶级命名空间 dc（应用的简称），随后为控制器、模型、用户界面以及其他应用程序部分所定义的嵌套名称空间。

A.5.3 推荐

前面的名称空间模式中，在编写 Backbone 应用程序时，我倾向于选择使用对象字面量模式的嵌套对象命名空间。

对一些小型应用程序，单一全局变量也许能胜任工作。然而，需要命名空间和深层命名空间的大型代码库则需要一个具有可读性和可伸缩性的简洁方案。我认为这种模式实现了这些目标，而且对大多数 Backbone 开发来说是一个很好的选择。

A.6 Backbone 依赖项详解

下面的小节讲述了 Backbone 是如何使用 jQuery / Zepto 以及 Underscore.js 的。

A.6.1 DOM 操作

尽管大多数开发人员不需要，但 Backbone 确实支持设置一个自定义的 DOM 库，以替换这些选项。从源代码中看：

```
// For Backbone's purposes, jQuery, Zepto, Ender, or My Library (kidding) owns
// the `$` variable.
 Backbone.$ = root.jQuery || root.Zepto || root.ender || root.$;
```

因此，设置“Backbone.$ = myLibrary;”将允许我们使用任何自定义 DOM 操作库来代替默认的 jQuery。

A.6.2 实用工具

在 Backbone 幕后，从对象扩展到事件绑定，都大量使用了 Underscore.js。由于整个库都被包括进去了，我们可以自由使用在集合上都可以使用的很多实用工具，比如过滤_.filter()、排序_.sortBy()、映射_.map()等。

源代码如下：

```
// Underscore methods that we want to implement on the Collection.
```

```
// 90% of the core usefulness of Backbone Collections is actually implemented
// right here:
var methods = ['forEach', 'each', 'map', 'collect', 'reduce', 'foldl',
'inject', 'reduceRight', 'foldr', 'find', 'detect', 'filter', 'select',
'reject', 'every', 'all', 'some', 'any', 'include', 'contains', 'invoke',
'max', 'min', 'toArray', 'size', 'first', 'head', 'take', 'initial', 'rest',
'tail', 'drop', 'last', 'without', 'indexOf', 'shuffle', 'lastIndexOf',
'isEmpty', 'chain'];

// Mix in each Underscore method as a proxy to `Collection#models`.
_.each(methods, function(method) {
    Collection.prototype[method] = function() {
        var args = slice.call(arguments);
        args.unshift(this.models);
        return _[method].apply(_, args);
    };
});
```

Underscore 所支持的方法的完整列表，请访问官方文档，地址：http://backbonejs.org/#Collection-Underscore-Methods。

A.6.3 RESTful 持久化

在 Backbone 中，我们可以使用 fetch、save 以及 destroy 方法，将模型和集合同步到服务器上。所有这些方法都将委托给 Backbone.sync 函数，实际上是包装了 jQuery / Zepto 的$.ajax 函数，调用 GET、POST 和 DELETE，以分别响应 Backbone 模型上的持久化方法。

Backbone.sync 源代码如下：

```
var methodMap = {
  'create': 'POST',
  'update': 'PUT',
  'patch':  'PATCH',
  'delete': 'DELETE',
  'read':   'GET'
};

Backbone.sync = function(method, model, options) {
    var type = methodMap[method];
    // ... Followed by lots of Backbone.js configuration, then..

    // Make the request, allowing the user to override any Ajax options.
    var xhr = options.xhr = Backbone.ajax(_.extend(params, options));
    model.trigger('request', model, xhr, options);
    return xhr;
```

A.6.4 路由

Backbone.History.start 的调用，依赖于 jQuery / Zepto 在 window 对象上绑定的

popState 或 hashchange 事件监听器。

Backbone.history.start 源代码如下：

```
// Depending on whether we're using pushState or hashes, and whether
// 'onhashchange' is supported, determine how we check the URL state.
if (this._hasPushState) {
    Backbone.$(window)
        .on('popstate', this.checkUrl);
} else if (this._wantsHashChange && ('onhashchange' in window) && !oldIE) {
    Backbone.$(window)
        .on('hashchange', this.checkUrl);
} else if (this._wantsHashChange) {
    this._checkUrlInterval = setInterval(this.checkUrl, this.interval);
}
...
```

类似地，Backbone.History.stop 使用 DOM 操作库给这些事件监听器解除绑定。

A.7 Backbone 与其他库和框架

Backbone只是构建应用程序众多解决方案中的一个，我们并不认为Backbone就是最终选择。Backbone帮助本书的很多贡献者都构建过简单的和复杂的Web应用程序，我希望它也一样可以帮到你。“Backbone是否比××好？”这个问题的答案，通常与正在构建的应用程序类型有很大关系。

AngularJS 和 Ember.js 是 Backbone 的有效替代选择，但是它们和 Backbone 不同，它们更固执己见。对于一些项目来说，它们可能有用；对其他项目来说，也许不是。要记住的重要一点是，没有任何一个库或框架是可以适用于所有用例的完美解决方案，因此重要的是了解所使用的工具，并从项目角度来决定使用哪一个。

选择合适的工具做正确的工作，这就是为什么我建议花时间做一个小的调查，要考虑生产率、易用性、可测试性、社区以及文档。如果你正在寻找更多的框架间的具体对比，请阅读：

- Journey Through the JavaScript MVC Jungle

http://coding.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/

- Rich JavaScript Applications——The Seven Frameworks

http://blog.stevensanderson.com/2012/08/01/rich-javascript-applications-the-seven-frameworks-throne-of-js-2012/

Backbone.js、AngularJS 以及 Ember 幕后的作者在 Quora、StackOverflow 等地方，也都讨论了他们方案的一些优点和缺点：

- Jeremy Ashkenas on why Backbone

http://backbonejs.org/#FAQ-why-backbone

- Tom Dale on Ember.js versus AngularJS

http://www.quora.com/Ember-js/Which-one-of-angular-js-and-ember-js-is-the-better-choice/answer/Tom-Dale

- Brian Ford and Jeremy Ashkenas on Backbone versus Angular (discussion)

http://www.reddit.com/r/javascript/comments/17h22w/an_introduction_to_angular_for_backbone_developers/

你选择的解决方案，可能需要支持构建重要的特性，并且最终用于在未来几年维护应用程序，所以请考虑如下内容：

（1）这个库/框架到底能做什么？

花时间阅读框架的源代码，并查看官方的功能列表，看看它们是否符合你的要求。可能会有项目需要修改或扩展底层的源代码，所以要确保，如果发生这种情况，你已经在代码上做过调研。该框架已经被用于生产环境了吗？是否有开发人员用它真正构建和部署过大型应用程序？是否可以公开访问？Backbone有很棒的案例组合（SoundCloud、LinkedIn、Walmart），这并不是所有的框架都能做到的。Ember用于很多大型应用程序，其中包括ZenDesk的新版本。AngularJS一直被用来创建PS3的YouTube应用程序，以及其他许多地方。重要的是要知道，框架不仅能在生产环境中使用，而且也能够让人看到真实的代码，从而得知可以用它构建哪些应用。

（2）该框架成熟吗？

我通常建议开发者不要简单地选择一个框架使用。新项目在发布之时通常会有很多异议，在生产环境级的应用上选择使用它们时，一定要特别小心。我不希望冒险让项目被封闭、经历长时间的重构或有其他破坏性的改变，而需要在一个框架成熟之前进行更仔细的计划。成熟的项目也往往有更详细的文档可用，要么是它们的官方文档，要么是社区驱动的文档。

（3）该框架是很灵活还是很固执？

要了解自己框架的风格，因为存在具有各种风格的大量框架可用。固执的框架将我

们锁定在特定的方式（它们的）做事。它们有意限制，但较少强调开发人员，必须要明白它们各自的工作原理。你是否真正在用该框架？

不使用框架编写一个小程序，然后尝试使用框架重构代码，以确认它是否容易使用。尽可能多地研究和阅读代码，这将会影响我们的决定。这与使用该框架编写实际的代码一样重要，以确保能够适应它强制执行的概念。

（4）该框架是否有全面的文档？

虽然演示应用程序可以用于参考，但我们总是发现自己需要查询官方框架的文档来找出它所支持的 API，如何用它创建常见的任务或组件，以及有哪些值得注意的地方。任何合格的框架都应该都一套详细的文档，这将有助于指导开发人员使用该框架。没有文档，我们会发现自己严重依赖 IRC 频道、小组或自我研究，这些方式是可行的，但与提前提供大量文档相比，它们往往过于耗时。

（5）框架的总大小是多少？是否支持最小化、压缩以及模块化构建？

该框架的依赖项是什么？框架往往只列出基础库自身的文件大小，而不包括依赖项的大小。这可能意味着，选择一个库，刚开始看起来很小，但是如果依赖 jQuery 和其他库的话，相对就会比较大。

（6）你是否查看了框架相关的社区？

是否有一个活跃的社区，以便在你遇到问题的时候，项目贡献者和用户能够帮你解决问题？是否有足够的开发人员在使用这个框架，并且有现成的参考应用、教程甚至视频，以便可以让你从中学到关于它的更多知识。

附录 B

资　源

B.1　书籍和课程

- **Prosthetics and Orthotics**

https://leanpub.com/building-backbone-plugins

- **PeepCode: Backbone.js Basics**

https://peepcode.com/products/backbone-js

- **Prosthetics and Orthotics——再推荐一下**

https://leanpub.com/building-backbone-plugins

- **CodeSchool: Anatomy of Backbone**

http://www.codeschool.com/courses/anatomy-of-backbonejs

- **Recipes with Backbone**

http://recipeswithbackbone.com/

- **Backbone Patterns**

http://ricostacruz.com/backbone-patterns/

- **Backbone on Rails**

https://learn.thoughtbot.com/products/1-backbone-js-on-rails

- **Derick Bailey's Resources for Learning Backbone**

http://lostechies.com/derickbailey

- **Learn Backbone.js Completely**

http://javascriptissexy.com/learn-backbone-js-completely/

- **Backbone.js on Rails**

https://learn.thoughtbot.com/products/1-backbone-js-on-rails

B.2 扩展/库

- **MarionetteJS**

http://marionettejs.com/

- **Backbone Layout Manager**

https://github.com/tbranyen/backbone.layoutmanager

- **AuraJS**

https://github.com/aurajs/aura

- **Thorax**

http://thoraxjs.org/

- **Lumbar**

http://walmartlabs.github.com/lumbar

- **Backbone Boilerplate**

https://github.com/tbranyen/backbone-boilerplate

- **Backbone Forms**

https://github.com/powmedia/backbone-forms

- **Backbone-Nested**

http://afeld.github.com/backbone-nested/

- **Backbone.Validation**

http://github.com/thedersen/backbone.validation

- **Backbone.Offline**

https://github.com/ask11/backbone-offline

- **Backbone-relational**

https://github.com/PaulUithol/Backbone-relational

- **Backgrid**

https://github.com/wyuenho/backgrid

- **Backbone.ModelBinder**

https://github.com/theironcook/Backbone.ModelBinder

- **Backbone Relational——处理模型的关系**

https://github.com/PaulUithol/Backbone-relational

- **Backbone CouchDB**

https://github.com/janmonschke/backbone-couchdb

- **Backbone.Validation——受 HTML5 启发的验证**

https://github.com/thedersen/backbone.validation

封面介绍

本书封面上的动物是一条澳大拉西亚鲷鱼（Pagrus auratus），主要出现于印尼海岸、中国大陆、菲律宾、中国台湾、澳大利亚、新西兰和日本。根据不同地区的发现，这条鱼有许多名字，但是无论在哪里，这都是一条珍贵的食用鱼。

澳大拉西亚鲷鱼在近岸产卵，并生活在岩石地区和珊瑚礁。在产卵时，它们将变成金属绿色，并在鱼鳞上积累大量的高浓酸。它们的成长模式在不同的区域有不同的变化，但是它们都能存活 40 年。