JavaScript Web Applications

面向jQuery开发者的客户端应用开发指南

基于MVC的 JavaScript Web 富应用开发

Alex MacCaw 著

李晶 张散集 译

O'REILLY®

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

如今 Web 应用程序的开发已经越来越向传统应用软件开发靠拢了，Web 和应用之间的界限也进一步模糊。传统编程语言中的设计模式、MVC、应用架构等理论也在慢慢地融入 Web 前端开发。这本书所涵盖的知识点非常全面，从 MVC 的基本理论到网络协议、从模块解耦到异步编程模型、从 HTML5/CSS3 到 NodeJS、从软件测试到部署调试，对于很多前端工程师来说，这些知识正是突破自己的瓶颈所亟需的。

这本书将专注于讲述如何构建“优雅又不失高水准”（state of the art）的 JavaScript 应用，包括软件架构、模板引擎、框架和库、同服务器的消息通信等内容。书中同样提供了大量的示例代码，可以帮助你更深入地理解很多重要的概念。除此之外，作者在 MVC 和架构方面的很多观点都很有启发性，即使你不是一名 JavaScript 程序员，读完本书后也会受益匪浅。

本书适合从事 JavaScript 开发，寻求进阶的前端开发人员、Web 架构师阅读。

978-1-449-30351-8 JavaScript Web Applications © 2011 by O’Reilly Media, Inc.
Simplified Chinese edition, jointly published by O’Reilly Media, Inc. and Publishing House of Electronics Industry, 2011.
Authorized translation of the English edition, 2011 O’Reilly Media Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O’Reilly Media, Inc.授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2011-7104

图书在版编目（CIP）数据

基于 MVC 的 JavaScript Web 富应用开发 /（美）麦卡劳（MacCaw,A.）著；李晶，张散集译. —北京：电子工业出版社，2012.5
书名原文：JavaScript Web Applications
ISBN 978-7-121-10956-0

Ⅰ. ①基… Ⅱ. ①麦… ②李… ③张… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 061858 号

策划编辑：张春雨
责任编辑：付　睿
封面设计：Karen Montgomery　张　健
印　　刷：北京丰源印刷厂
装　　订：三河市鹏成印业有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：19.25　字数：462 千字
印　　次：2012 年 5 月第 1 次印刷
定　　价：59.00 元

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

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

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始,O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来,而我们关注真正重要的技术趋势——通过放大那些"细微的信号"来刺激社会对新科技的应用。作为技术社区中活跃的参与者,O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的"动物书";创建第一个商业网站(GNN);组织了影响深远的开放源代码峰会,以至于开源软件运动以此命名;创立了Make杂志,从而成为DIY革命的主要先锋;公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖,共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择,O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版,在线服务或者面授课程,每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

"O'Reilly Radar博客有口皆碑。"

——Wired

"O'Reilly凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。"

——Business 2.0

"O'Reilly Conference是聚集关键思想领袖的绝对典范。"

——CRN

"一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。"

——Irish Times

"Tim是位特立独行的商人,他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了:'如果你在路上遇到岔路口,走小路(岔路)。'回顾过去Tim似乎每一次都选择了小路,而且有几次都是一闪即逝的机会,尽管大路也不错。"

——Linux Journal

译者序

从第一眼看到封面上这只憨憨的猫头鹰开始，就深深地喜欢上了这本 *JavaScript Web Applications*，读了简介和目录之后就已经不能自拔了。这几年鲜有深入讲架构级 web app 的好书，这让这本 *JavaScript Web Applications* 更加难得，作为 O'Reilly 第一本专注于纯高端 JavaScript 架构思想的书，凡是有一点“架构情节”的工程师都不应当错过。

如今 Web 应用程序的开发已经越来越向传统应用软件开发靠拢了，Web 和应用之间的界限也进一步模糊。传统编程语言中的设计模式、MVC、应用架构等理论也在慢慢地融入 Web 前端开发。随着服务器端 JavaScript 和移动终端的兴起，作为一名前端工程师，也深知自己正处在一个深刻变革的年代，面对眼花缭乱的新概念和新技术更应当把握本质、认清方向，勇于创新和实践，而这本 *JavaScript Web Applications* 的出现更是一阵及时雨，为我们工作中遇到的很多难题提供了解决方案和最佳实践。同时，这本书所涵盖的知识点非常全面，从 MVC 的基本理论到网络协议、从模块解耦到异步编程模型、从 HTML5/CSS3 到 NodeJS、从软件测试到部署调试，对于很多前端工程师来说，这些知识正是突破自己的瓶颈所亟需的。

这本书将专注于讲述如何构建“优雅又不失高水准”（state of the art）的 JavaScript 应用，包括软件架构、模板引擎、框架和库、同服务器的消息通信等内容。书中同样提供了大量的示例代码，可以帮助你更深入地理解很多重要的概念。除此之外，作者在 MVC 和架构方面的很多观点都很有启发性，即使你不是一名 JavaScript 程序员，读完本书后也会受益匪浅。

本书作者 Alex MacCaw 是一名 Ruby/JavaScript 程序员，是 Spine 框架的开发者。在翻译本书的过程中，我深深体会到他作为一名优秀工程师所具备的扎实的计算机专业功底和让人敬佩的开源精神。尽管这本书包含大量的专业术语，但作者文笔轻松流畅，即使直接读原文也丝毫不会感到枯燥，所以我们在翻译过程中也是非常小心，生怕丢掉这种

轻松流畅的阅读感觉，尽力为大家原汁原味地呈现本书。当然由于专业知识所限，翻译过程难免疏漏，还希望各位高手批评指正。

最后，我要感谢博文视点的张春雨在译书过程中给予我们的帮助和信任。感谢我的好友王保平（玉伯）对很多关键的技术性问题提出的宝贵意见，还要感谢可爱的同事杨振楠（栋寒）、杨翰文（地极）、李燕青（霸先）、车思慧（灵玉）、陈良（舒克）的细心校对，他们给译文提了很多中肯的建议。当然，最最需要感谢的是家中的“领导”，已经记不得多少次赶译稿加班太晚，得到的不是你的抱怨，而是你的鼓励，这让我至今备感温暖。

李晶（拔赤），张散集（一舟）

2011 年 12 月 北京

前言

1995 年随着 Netscape 浏览器的发布，JavaScript 也作为它的组成部分进入到公众的视野，之后 JavaScript 的发展道路尽管充满坎坷但成长飞速，如今得益于高性能的 JIT（just in time）解析引擎，（在浏览器端）JavaScript 已经无孔不入了。仅仅在 5 年以前，开发者还在使用 Ajax 写一些短小的代码或热衷于实现一些类似“黄色渐褪技术”的网页特效；而现在，复杂的 JavaScript 应用已经可以写上成百上千行的代码了。

就在去年，互联网出现了一股追捧 JavaScript 应用的浪潮，很多人开始着迷于给 Web 应用加入很多桌面软件的交互元素，增强 Web 应用的用户体验，这种趋势犹如星星之火迅速蔓延至整个互联网。在过去，在浏览器性能不佳的情况下，用户在进行 Web 应用时每次交互都要刷新页面，而且页面加载很慢。而如今 JavaScript 引擎已经变得异常强大，我们可以将很多交互行为植入客户端，这样交互的响应就会非常及时，增强体验。

当然获得提升的不仅仅是 JavaScript 引擎的性能。尽管 CSS3 和 HTML5 规范现在仍在修订之中，也已经有很多现代浏览器广泛支持这些新特性了，比如 Safari、Chrome 和 Firefox，IE9 也在一定程度上支持这些新特性。利用这些特性可以花更少的时间做出更棒的视觉效果，而且不用花精力做图片的切割和拼合来模拟视觉效果。现在浏览器的升级也很快，对 HTML5 和 CSS3 的支持也一天比一天好。但你还是要定义一个浏览器测试基准（你的应用所支持的最低标准的客户端软件和版本），基于此才能更加合理地选择所需的技术。

将应用的重心从服务器迁移到客户端并不轻松，这和构建服务器应用的方法完全不一样。你需要想清楚架构、模板、与服务器端的通信、框架等，这些正是本书所涵盖的内容。我将手把手教你如何构建“优雅又不失高水准”的 JavaScript 应用。

本书的目标读者

本书不是为 JavaScript 初学者所写，如果你对 JavaScript 这门语言缺乏基本的了解和认识，我建议你先阅读一些更基础的书，比如 Douglas Crockford 著的 *JavaScript: The Good Parts*（*http://oreilly.com/catalog/9780596517748*）（O'Reilly）。本书更适合有一些 JavaScript 开发经验的开发者，比如使用 jQuery 类库的开发者，或者当你希望构建更复杂、更高级的 JavaScript 应用时，本书也是适合你的。此外，本书的很多章节，特别是附录，对于有经验的 JavaScript 开发者来说也是非常有帮助的。

本书的内容组织

第 1 章

本章从 JavaScript 的发展历程开始，介绍了 JavaScript 的发展现状和对互联网的巨大影响。然后轻描淡写地介绍了 MVC 的基本概念，随后又讲解了 JavaScript 的构造函数、原型继承及如何使用 JavaScript 创建一个类库。

第 2 章

本章主要介绍了浏览器的事件机制，包括事件机制的发展历史，API 设计和事件模型的行为和实现。然后讲解了如何基于 jQuery 绑定事件监听、使用代理，以及创建自定义事件。最后使用发布 / 订阅模式实现了“DOM 无关”事件。

第 3 章

本章讲解了如何在你的应用中使用 MVC 模型，包括加载和操作远程数据。我们将会提到为什么在构建 ORM 类库的时候使用 MVC 和命名空间是如此之重要，以及如何使用 ORM 类库来管理模型数据。接下来讲解了如何使用 JSONP 和跨域 Ajax 来加载远程数据。最后介绍了如何通过使用 HTML5 本地存储和将本地存储提交至 RESTful 服务器，来实现模型数据的持久化。

第 4 章

本章演示了如何使用控制器模式在客户端保持一个状态。我们将讨论如何将逻辑封装成模块、阻止全局命名空间的污染，然后介绍如何使用视图来进一步简化控制器的结构，以及怎样在视图中实现 DOM 事件监听。本章的最后将会讨论路由选择，包括使用 URL 中的 hash 片段，使用新的 HTML5 History API 等技术，以及确保解释两种方法的利弊。

第 5 章

本章介绍了视图和 JavaScript 模板，给出了多种动态渲染视图的方式，以及很多模

板类库和存储模板的方式（使用行内形式存储模板、使用 script 标签，以及远程加载）。接下来，你会接触到数据绑定的一些内容，包括使模型控制器、视图与模型数据、视图数据动态同步连接。

第 6 章

本章详细介绍了使用 CommonJS 模块系统来做 JavaScript 的依赖管理。开始会介绍 CommonJS 背后的历史和思想，接下来会讲解如何在浏览器端使用 CommonJS 模块，包括介绍一些模块加载器类库，比如 Yabble 和 RequireJS。然后，我们讨论了如何自动在服务器端包装模块，从而提高性能、节省时间。本章的最后会介绍 CommonJS 的一些替代方案，比如 Sprockets 和 LABjs。

第 7 章

这里将会讲到 HTML5 带给我们的一些好处：文件操作 API。本章将会涵盖文件操作 API 的浏览器支持情况、多文件上传、拖曳上传文件及使用剪切板事件。接下来会介绍使用二进制大文件和文件切割来读文件，同时将读取的结果在浏览器中输出。然后讲解使用 XHR（XMLHttpRequest）Level 2 规范来实现在后台上传文件，最后向大家展示一个使用 jQuery Ajax API 实现文件上传进度指示的例子。

第 8 章

本章主要关注实时应用和 WebSocket 技术的一些令人兴奋的发展趋势。首先介绍实时应用的发展历史及各种实现技术的浏览器兼容性情况。然后更详细地介绍 WebSocket 和基于它的更高级的实现，包括浏览器兼容性和 JavaScript API。接下来展示一个使用 WebSocket 实现的简单的 RPC 服务，看一下如何在客户端和服务器端之间建立连接。然后介绍 Socket.IO 和如何搭建实时架构，最后介绍用户体验方面的一些考量。

第 9 章

本章主要讲解测试和调试的内容，这些内容是 JavaScript 网络应用开发过程中的关键环节。我们的话题将围绕跨浏览器测试的主题进行展开，介绍浏览器基准的选择、单元测试和测试类库，比如 QUnit 和 Jasmine。接下来，介绍自动化测试和持续集成服务器，比如 Selenium。然后讲解调试相关的内容，研究了 Firefox 和 WebKit 网络监测器、主控台，以及使用 JavaScript 调试器。

第 10 章

本章介绍了另外一个非常重要却又极易被忽略的内容——JavaScript 网络应用的部署。我们主要考虑性能方面，以及如何使用缓存、代码压缩、gzip 压缩及其他减少应用初始化加载时间的技术。最后简单讲解了如何使用 CDN 服务器来让我们的工作事半功倍，以及如何使用浏览器内置的策略来提升你站点的性能。

第 11 章

接下来的 3 章主要介绍了一些流行的 JavaScript 类库，这些类库常用来做 JavaScript 应用开发。Spine 是一个轻量级的 MVC-compliant 类库，这个类库使用了本书中讲到的很多概念。本章将会为你介绍类库的核心部分：类、事件、模型和控制器。最后本章用一个管理应用的例子来展示本章所讲到的知识点。

第 12 章

Backbone 是一个非常流行的类库，使用这个类库可以非常高效地构建 JavaScript 应用，本章主要介绍这个类库。本章会涵盖 Backbone 的核心观念和类，比如模型、集合、控制器和视图等。接下来会介绍使用 RESTful JSON 请求从服务器同步获取模型数据，以及如何在服务器端响应 Backbone。最后我们给出一个待办事项列表应用的例子，来向大家展示如何使用这个类库。

第 13 章

本章主要介绍了 JavaScriptMVC 类库，这是一个流行的基于 jQuery 的框架，用来构建 JavaScript 网络应用。在本章中你将会学到 JavaScriptMVC 的一些基础知识，比如类、模型和控制器，同时还包含客户端的模板及渲染视图。本章的最后会给出一个实际的 CRUD 列表的例子，给读者展示使用 JavaScriptMVC 创建抽象的、可重用的、节省内存的组件是多么的简单。

附录 A

附录 A 中是对 jQuery 的简要介绍，如果你想温习类库内容，则这部分内容对你会非常有帮助。本书中大部分示例代码都是基于 jQuery 的，首先熟悉 jQuery 是很重要的。这一部分会讲到大部分核心的 API，比如 DOM 操作、DOM 查询和遍历，以及事件绑定、触发和事件代理。接下来会讲解 jQuery 的 Ajax API，包括 POST、GET 和 JSON 请求。随后将介绍 jQuery 扩展，如何使用 jQuery 来封装一个插件，让你的代码更具通用性。最后展示了一个实际的例子：创建一个 Growl jQuery 插件。

附录 B

附录 B 的内容主要是讲解 Less，Less 是 CSS 的超集，它使用变量、混合、操作符和优雅的规则扩展了 CSS 本身的语法。利用这些规则可以极大地减少你所写的 CSS 代码量，特别是使用 CSS3 效果更佳。附录 B 包含 Less 的主要的增强的语法，以及如何使用命令行工具和 JavaScript 类库来将 Less 文件编译成 CSS。

附录 C

附录 C 主要讲解了 CSS3。首先介绍了一些 CSS3 的背景知识、浏览器厂商的前缀，然后开始介绍 CSS3 的主要内容，从主要附件到规格说明。这里介绍的 CSS 特性

主要包括：圆角、`rgba`颜色、阴影、渐变、动画和变换。附录的最后讨论了使用Modernizr实现的优雅降级，并展示了一个实际的使用box-sizing规范的例子。

本书的约定

本书使用下列排版约定：

斜体 *Italic*

用于表示新术语、URL、电子邮件地址、文件名、文件扩展名和事件。

等宽字体 `Constant width`

用来表示计算机代码片段，包括命令、数组、元素、语句、操作符、变量、属性、关键字、函数、类型、类、命名空间、方法、形参、实参、值、对象、事件处理程序、XML标签、HTML标签、宏指令、文件内容及命令行的输出等。

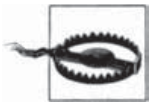
等宽加粗字体 **`Constant width bold`**

用来表示命令或者其他用户输入的文本。

等宽斜体 *`Constant width italic`*

用来表示可被替换的字符或文本，这些字符在合适的场景和特定的条件下会被替换成其他的值。

这个图标表示一种提示、建议或一般的消息提醒。



这个图标表示一种警告。

中文版书中切口处的“▭”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引所列页码为原英文版页码。

附加文件

本书的附加文件都存放在GitHub上（*https://github.com/maccman/book-assets*），可以直接在GitHub上查看，也可以下载压缩包（*https://github.com/maccman/book-assets/zipball/master*）。所有这些示例代码都以章节为单位存放，都已经包含了各自所需的类库，本书中用到的大多数示例代码同样在单独的文件中。

在本书中凡是引用这些附加文件的地方，都会以这种形式表述：*assets/chapter_number/name*。

代码约定

本书中我们以 `assert()` 和 `assertEqual()` 函数来展示变量的值或者函数调用的结果。`assert()` 是一种快捷表述方式，用来表示一个特定的变量是真值。这在自动化测试中是一种非常常见的模式。`assert()` 可以接收两个参数：一个值和一个可选的消息。如果运行结果不是真值，这个函数将抛出一个异常：

```
var assert = function(value, msg) {
  if ( !value )
    throw(msg || (value + " does not equal true"));
};
```

`assertEqual()` 是表示一个值等于另外一个值的另一种表述。它和 `assert()` 类似，但接收两个值。如果这两个值不相等，则这个断言失败：

```
var assertEqual = function(val1, val2, msg) {
  if (val1 !== val2)
    throw(msg || (val1 + " does not equal " + val2));
};
```

这两个函数非常简单，正如你在示例代码中所看到的。如果断言失败，你会在浏览器的控制台中看到一个错误消息：

```
assert( true );

// 和 assertEqual() 等价
assert( false === false );

assertEqual( 1, 1 );
```

我们可以从代码中看出，对象比较会失败，除非两个对象是指向同一块内存的引用。解决办法是深比较，在 *assets/ch00/deep_equality.html* 这个例子中可以看到完整的代码。

jQuery 示例代码

本书的大部分示例代码都是基于 jQuery（*http://jquery.com*）的，jQuery 是现在最流行的 JavaScript 类库，它对事件、DOM 遍历、DOM 操作和 Ajax 都做了封装。这里我选用 jQuery 是出于几个原因的考虑，最主要的原因是 jQuery 可以让代码变得非常简洁，而且当下大部分人对 jQuery 都非常熟悉，一看即懂。

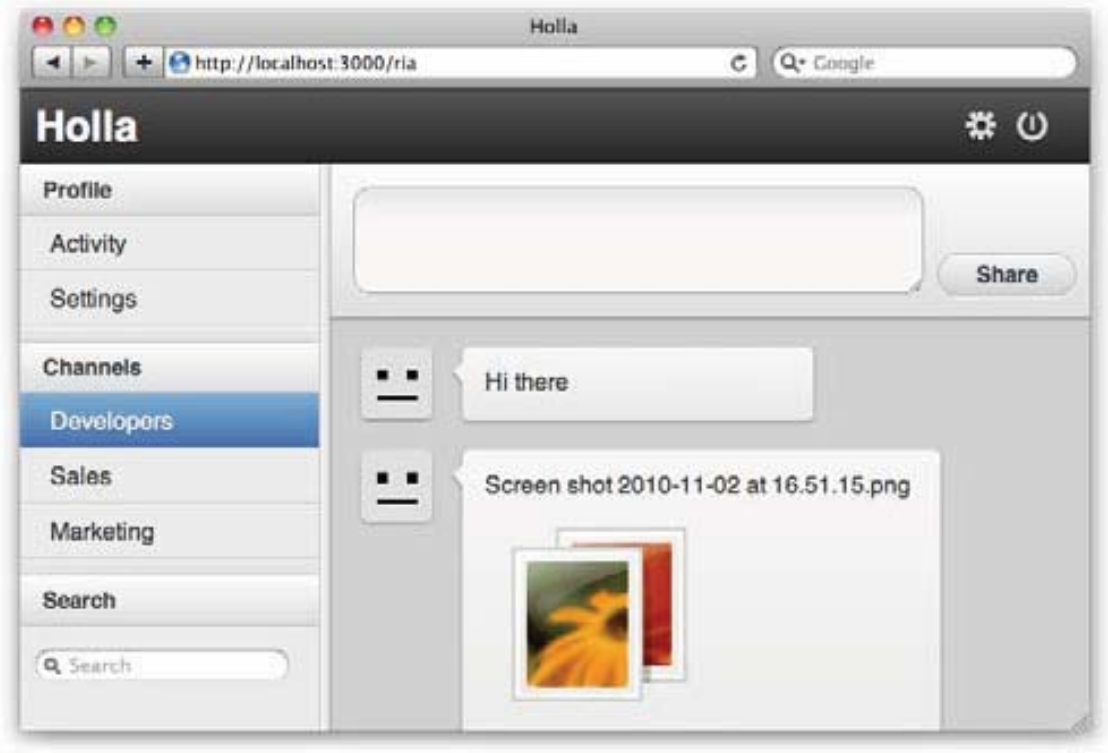
如果你没有使用过 jQuery，我强烈推荐你首先看一下 jQuery 的文档。它的 API 非常不错，为 DOM 提供了一组非常棒的抽象的接口。可以在附录 A 中查阅到简短的 jQuery 入门内容。

Holla

Holla（*http://github.com/maccman/holla*）贯穿本书始终，它是一个 JS 群聊应用。Holla 是一个非常不错的示例应用，因为它和本书中大多数章节和内容都有交集。除了正文章节中对 Holla 的讲述之外，Holla 为我们展示了：

- 使用 CSS3 和 HTML5 来构建美观的界面。
- 拖曳上传文件。
- 使用 Sprockets 和 Less 来编写代码。
- 使用 WebSocket 将数据发送给客户端。
- 创建带有状态的 JavaScript 应用。

可以从 Holla 的 GitHub 的代码库（*http://github.com/maccman/holla*）中将代码复制下来，研读一下它的代码。本书中用到的很多例子都来自 Holla 的源代码，Holla 的界面如图 P-1。



图P-1：Holla聊天应用程序运行界面

作者附言

本书是我在环游世界的时候完成的，这花费了我一年的时间。这一年我经历了很多地方，这本书一部分是我在非洲时编写的，那里没有电也没有网络，还有一部分是在日本古朴幽静的寺院中和凝霞漫烂的樱花树下完成的，还有一些内容是在遥远美丽的哥伦比亚岛屿上完成的。我非常享受这段时光，希望我的这种美妙的体验能通过我的文字传达给每一位读者。

这里我要特别感谢一些人。感谢 Stuart Eccles、Tim Malbon、Ben Griffins 和 Sean O'Halpin，是他们给了我这个机会，让我重新找寻到埋藏在心底的激情。同样要感谢 James Adam、Paul Battley 和 Jonah Fox，他们是我值得尊敬的导师，谆谆之言让我获益良多。

同样要感谢出版社的编辑们，他们严格的审校保证了本书的质量：Henrik Joreteg、Justin Meyer、Lea Verou、Addy Osmani、Alex Barbara、Max Williams 和 Julio Cesar Ody。

当然最需要感谢的是我的父母，他们的默默支持是我坚实的后盾。

Safari® Books Online

Safari 在线图书是一个数字图书馆，读者可以在这个图书馆里自选图书，在这里可以搜索到超过 7500 本技术相关的书籍创作和视频，在这里可以迅速找到你想要的内容。

订阅之后，你就可以阅读在线图书馆的任意图书的任意章节和任意视频。你还可以将图书下载到手机里。在纸质书籍出版前就可以抢先阅读，甚至可以抢先阅读作者手稿，并实时给作者反馈。同时还可以复制粘贴实例代码、组织你的收藏内容、下载章节、将关键段落加入书签、创建笔记、打印出来，你既可以节省时间又可以提升阅读效率。

O'Reilly 已经将本书（英文版）上传至 Safari 在线图书馆里了。如果想在线阅读本书和其他相关内容，请免费登录 *http://my.safaribooksonline.com*。

联系我们

对于本书的评论或问题请联系出版商：

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

我们为本书制作了一个 Web 页面，页面中包含了简介、样章，以及其他信息。可以从这里访问这个页面：

http://www.oreilly.com/catalog/9781449303518
http://www.oreilly.com.cn

如果要留言或者提交关于本书的技术问题的反馈，请发邮件至：

bookquestions@oreilly.com

本书的更多信息、资源、参考文献和新闻，请登录出版社官网：*http://www.oreilly.com* 或者 *http://www.oreilly.com.cn/*。

Facebook：*http://facebook.com/oreilly*

Twitter：*http://twitter.com/oreillymedia*

YouTube：*http://www.youtube.com/oreillymedia*

目录

第1章 1

MVC和类

最初

JavaScript 程序开发已经和最初我们想象中的模样有了天壤之别，也很少有人能记起从 JavaScript 诞生之初的 Netscape 浏览器到如今异常强大的解析引擎——比如 Google 的 V8——的进化历程。JavaScript 到 ECMAScript 的标准化道路也充满坎坷。然而对于 JavaScript 的发明者来说，做梦也不会想到 JavaScript 会有今天这么强大。

尽管 JavaScript 已然非常成功和流行，但仍然被大多数人所误解。只有少数人知道 JavaScript 是一种强大的、动态的面向对象编程语言。JavaScript 中诸如原型继承、模块和命名空间等高级特性依然会让很多人感到吃惊。那么，为什么这门语言会如此被误解？

一个原因是早期的 JavaScript 实现非常糟糕，有很多 bug；另一个原因是因为其名字带有“*Java*”前缀，让人以为它和 Java 有关系。实际上，它和 Java 是完全不同的两种语言。然而，在我看来，真正的原因在于大多数开发者接触和使用 JavaScript 的方式。对于其他语言来说，比如 Python 和 Ruby，开发者必须要坚持阅读技术文档、视频教程和学习指南。但是直到现在，使用 JavaScript 开发程序也不用这样，开发者的需求往往是给现有代码添加一个表单验证、弹出框或图片轮播控件，而且工期也很紧。因此他们直接去网上找一段能用的代码就可以了，而不必花时间去学习和理解这门语言。很多人就是这样开始接触 JavaScript 的，并堂而皇之地把 JavaScript 技能写入他们的简历。

现在，JavaScript 引擎和浏览器已经变得非常强大，使用 JavaScript 来构建庞大的应用已经屡见不鲜，而且越来越流行。像 Gmail 和 Google Maps 之类的产品给我们带来了 Web 应用全新的体验，开发者们顿时趋之若鹜。公司开始雇用全职的 JavaScript 程序员，JavaScript 也早已不再是只能完成表单验证的“不入流的脚本语言”了。现在凭借其自身独特的优势，JavaScript 已经成为一门独立的、潜力无穷的编程语言。 2

这种趋势说明 JavaScript 应用会如雨后春笋一般遍地开花。不幸的是，可能是因为 JavaScript 糟糕的过去，很多 JavaScript 应用的架构是非常脆弱的。某些原因是，当使用 JavaScript 开发应用时，那些经典的设计模式和最佳实践被抛在了脑后。开发者往往忽略架构模型，比如 MVC 模型，而常将应用中的 HTML 和 JavaScript 混杂在一起，看着像一个大杂烩。

本书不会教给你 JavaScript 是一门什么样的语言，你可以阅读其他书籍来学习使用 JavaScript，比如 Douglas Crockford 的 *JavaScript: The Good Parts*（*http://goo.gl/JDoIT*）（O'Reilly）。但是，本书将会向你展示如何搭建复杂的 JavaScript 应用，教你创造不可思议的网络用户体验。

增加结构

构建大型的 JavaScript 应用的秘诀是“不要”构建大型 JavaScript 应用。相反，你应当把你的应用解耦成一系列相互平等且独立的部分。开发者常犯的错误是创建应用时使用了很多互相依赖的部分，用了很多 JavaScript 文件，并在 HTML 页面中用大量的 script 标签引入这些文件。这类应用非常难于维护和扩展，因此无论如何都应当避免这种情况的发生。

开始构建你的应用的时候，花点精力来做应用的架构，会为最终结果带来意想不到的改观。不管你之前怎么看待 JavaScript，从现在开始将它当做一门面向对象的编程语言来对待。如果你使用 Python 和 Ruby 这样的编程语言来开发应用，你同样会使用类、继承、对象和设计模式等。对于构建服务器端应用来说，体系结构是非常重要的，那么为什么不在客户端应用中采用这些东西呢？

本书提倡使用 MVC 模式，这是一种久经考验的搭建应用的方式，可以确保应用的可维护性和可扩展性。MVC 模式完全适用于 JavaScript 应用。

什么是 MVC

MVC 是一种设计模式，它将应用划分为 3 个部分：数据（模型）、展现层（视图）和用户交互层（控制器）。换句话说，一个事件的发生是这样的过程：

1. 用户和应用产生交互。
2. 控制器的事件处理器被触发。
3. 控制器从模型中请求数据，并将其交给视图。
4. 视图将数据呈现给用户。

现在来看一个真实的例子，图 1-1 展示了在 Holla 中如何发送新的聊天消息。

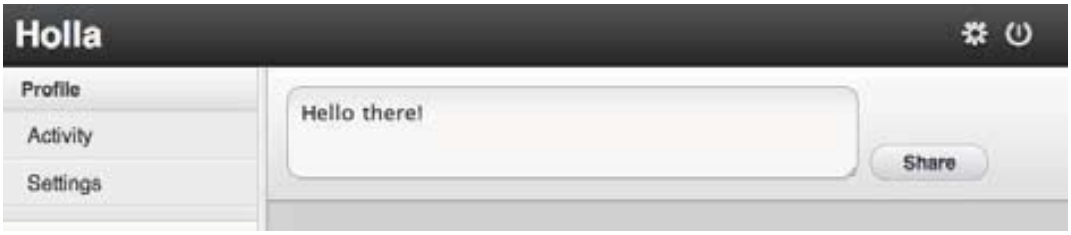


图1-1：从Holla中发送一个新的聊天消息

1. 用户提交一个新的聊天消息。
2. 控制器的事件处理器被触发。
3. 控制器创建了一个新的聊天模型（Chat Model）记录。
4. 然后控制器更新视图。
5. 用户在聊天窗口看到新的聊天消息。

我们不用类库或框架就可以实现这种 MVC 架构模式。关键是要将 MVC 的每部分按照职责进行划分，将代码清晰地分割为若干部分，并保持良好的解耦。这样可以对每个部分进行独立开发、测试和维护。

下面来详细讲解 MVC 中的各个组成部分。

模型

模型用来存放应用的所有数据对象。比如，可能有一个 User 模型，用以存放用户列表、它们的属性及所有与模型有关的逻辑。

模型不必知晓视图和控制器的细节，模型只需包含数据及直接和这些数据相关的逻辑。任何事件处理代码、视图模板，以及那些和模型无关的逻辑都应当隔离在模型之外。将模型和视图的代码混在一起，是违反 MVC 架构原则的。模型是最应该从你的应用中解耦出来的部分。

当控制器从服务器抓取数据或创建新的记录时，它就将数据包装成模型实例。也就是说，我们的数据是面向对象的（object oriented），任何定义在这个数据模型上的函数或逻辑都可以直接被调用。

因此，不要这样做 ：

4

```
var user = users["foo"];
destroyUser(user);
```

而要这样做：

```
var user = User.find("foo");
user.destroy();
```

第 1 段代码没有命名空间的概念，并且不是面向对象的。如果在应用中定义了另一个 `destroyUser()` 函数的话，两个函数就会产生冲突。我们应当确保全局变量和函数的个数尽可能少。在第 2 段代码中，`destroy()` 函数是存放在命名空间 User 的实例中的，User 中存放了所有的记录。当然这只是理想状况，因为我们控制了全局变量的个数，更好地避免了潜在的冲突，这种代码更加清晰，而且非常容易做继承，类似 `destroy()` 的这种函数就不用在每个模型中都定义一遍了。

在第 3 章中我们会更深入地讲解模型，其中包含从服务器下载数据及创建对象关系映射（ORM）。

视图

视图层是呈现给用户的，用户与之产生交互。在 JavaScript 应用中，视图大都是由 HTML、CSS 和 JavaScript 模板组成的。除了模板中简单的条件语句之外，视图不应当包含任何其他逻辑。

实际上，和模型类似，视图也应当从应用的其他部分中解耦出来。视图不必知晓模型和控制器中的细节，它们是相互独立的。将逻辑混入视图之中是编程的大忌。

这并不是说 MVC 不允许包含视觉呈现相关的逻辑，只要这部分逻辑没有定义在视图之内即可。我们将视觉呈现逻辑归类为“视图助手”（*helper*）：和视图有关的独立的小型工具函数。

来看下面的例子，其在视图中包含了逻辑，这是一个反例，平时不应当这样做：

```
// template.html
<div>
  <script>
    function formatDate(date) {
      /* ... */
    };
  </script>
  ${ formatDate(this.date) }
</div>
```

在这段代码中，我们把 `formatDate()` 函数直接插入视图中，这违反了 MVC 的原则，结果导致标签看上去像大杂烩一样不可维护。可以将视觉呈现逻辑剥离出来放入视图助手中，正如下面的代码就避免了这个问题，可以让这个应用的结构满足 MVC。

```
// helper.js
var helper = {};
helper.formatDate = function(){ /* ... */ };

// template.html
<div>
  ${ helper.formatDate(this.date) }
</div>
```

此外，所有视觉呈现逻辑都包含在 helper 变量中，这是一个命名空间，可以防止冲突并保持代码清晰、可扩展。

不要太在意视图和模板的细节，我们会在第 5 章中有详细讲述。本小节的目的只是简单介绍视图和 MVC 架构模式之间的联系。

控制器

控制器是模型和视图之间的纽带。控制器从视图获得事件和输入，对它们（很可能包含模型）进行处理，并相应地更新视图。当页面加载时，控制器会给视图添加事件监听，比如监听表单提交或按钮点击。然后，当用户和你的应用产生交互时，控制器中的事件触发器就开始工作了。

不用使用类库和框架也能实现控制器，下面这个例子就是使用简单的 jQuery 代码来实现的：

```
var Controller = {};

// 使用匿名函数来封装一个作用域
(Controller.users = function($){

  var nameClick = function(){
    /* ... */
  };

  // 在页面加载时绑定事件监听
  $(function(){
    $("#view .name").click(nameClick);
  });

})(jQuery);
```

我们创建了 users 控制器，这个控制器是放在 Controller 变量下的命名空间。然后，我们使用了一个匿名函数封装了一个作用域，以避免对全局作用域造成污染。当页面加载时，程序给视图元素绑定了 *click* 事件的监听。

正如你所看到的，控制器并不依赖类库或框架。然而，为了构建需要的一个完整的 MVC 框架，我们需要将模型从视图中抽离出来。控制器和状态的详细内容会在第 4 章详细讲解。

6 向模块化进军，创建类

在讲解 MVC 的本质之前，我们首先给大家补习一下基础知识，比如 JavaScript 的类和事件。只有打下一个坚实的基础，才能更好地学习、理解更高级的概念。

对于静态的类来说，JavaScript 对象直接量就已经够用了，但它对使用继承和实例来创建经典的类往往更有帮助。有必要强调一下：JavaScript 是基于原型的编程语言，并没有包含内置类的实现。但通过 JavaScript 可以轻易地模拟出经典的类。

JavaScript 中的类口碑并不太好，因为“不够 JavaScript”而饱受批评。jQuery 并没有涉及太多架构方法和继承模式，这让 JavaScript 开发者确信自己不必考虑太多架构性的东西，甚至觉得类的用处不大或干脆禁用类。实际上，类是另一种有用的工具，作为一名实用主义者，我相信类在 JavaScript 中的重要性丝毫不亚于它在其他现代编程语言中的重要性。

JavaScript 中并没有真正的类，但 JavaScript 中有构造函数和 new 运算符。构造函数用来给实例对象初始化属性和值。任何 JavaScript 函数都可以用做构造函数，构造函数必须使用 new 运算符作为前缀来创建新的实例。

new 运算符改变了函数的执行上下文，同时改变了 return 语句的行为。实际上，使用 new 和构造函数很类似于传统的实现了类的语言：

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

// 实例化一个 Person
var alice = new Person('alice');

// 检查这个实例
assert( alice instanceof Person );
```

构造函数的命名通常使用驼峰命名法，首字母大写，以此和普通的函数区分开来，这是一种习惯用法。记住这一点非常重要，因为你不会希望用省略 new 前缀的方式来调用构造函数。

```
// 不要这么做！
Person('bob'); //=> undefined
```

这个函数只会返回 undefined，并且执行上下文是 window（全局）对象，你无意间创建了一个全局变量 name。调用构造函数时不要丢掉 new 关键字。

当使用 new 关键字来调用构造函数时，执行上下文从全局对象（window）变成一个空的上下文，这个上下文代表了新生成的实例。因此，this 关键字指向当前创建的实例。尽管理解起来有些绕，实际上其他语言内置类机制的实现也是如此。

默认情况下，如果你的构造函数中没有返回任何内容，就会返回 this——当前的上下文。要不然就返回任意非原始类型的值。比如，我们可以返回一个用以新建一个新类的函数，第一步要做的是创建自己的类模拟库：

```
var Class = function(){
  var klass = function(){
    this.init.apply(this, arguments);
  };
  klass.prototype.init = function(){};
  return klass;
};

var Person = new Class;

Person.prototype.init = function(){
  // 基于 Person 的实例做初始化
};

// 用法：
var person = new Person;
```

令人费解的是，由于 JavaScript 2（*http://www.mozilla.org/js/language/js20-1999-02-18/classes.html*）规范从未被实现过，class 一直都是保留字。最常见的做法是将变量名 class 改为 _class 或 klass。

给类添加函数

在 JavaScript 中，在构造函数中给类添加函数和给对象添加属性是一模一样的：

```
Person.find = function(id){ /*...*/ };

var person = Person.find(1);
```

要想给构造函数添加实例函数，则需要用到构造函数的 prototype：

```
Person.prototype.breath = function(){ /*...*/ };

var person = new Person;
```

```
person.breath();
```

一种常用的模式是给类的 prototype 起一个别名 fn，写起来也更简单：

```
Person.fn = Person.prototype;

Person.fn.run = function(){ /*...*/ };
```

实际上这种模式在 jQuery 的插件开发中是很常见的，将函数添加至 jQuery.fn 中也就相当于添加至 jQuery 的原型中。

8 给“类”库添加方法

现在，我们的“类”库（class library）[译注1] 包含了生成一个实例并初始化这个实例的功能，给类添加属性和给构造函数添加属性是一样的。

直接给类设置属性和设置其静态成员是等价的：

```
var Person = new Class;

// 直接给类添加静态方法
Person.find = function(id){ /* ... */ };

// 这样我们可以直接调用它们
var person = Person.find(1);
```

给类的原型设置的属性在类的实例中也是可用的：

```
var Person = new Class;

// 在原型中定义函数
Person.prototype.save = function(){ /* ... */ };

// 这样就可以在实例中调用它们
var person = new Person;
person.save();
```

但在我看来这种语法有些绕，不切实际且不够简洁，很难一眼就分辨出类的静态属性和实例的属性。因此我们采用另外一种不同的方法来给类添加属性，这里用到了两个函数 extend() 和 include()：

```
var Class = function () {
  var klass = function () {
    this.init.apply(this, arguments);
```

译注1：原文中此处和小标题中都是class library，意思是“实现了类机制的类库”，直译为“类库”，有时library也译为类库，但两者的含义是不一样的，读者应能自行分辨。

```
  };

  klass.prototype.init = function () {};

  // 定义 prototype 的别名
  klass.fn = klass.prototype;

  // 定义类的别名
  klass.fn.parent = klass;

  // 给类添加属性
  klass.extend = function (obj) {
    var extended = obj.extended;
    for (var i in obj) {
      klass[i] = obj[i];
    }
    if (extended) extended(klass)
  };

  // 给实例添加属性
  klass.include = function (obj) {
    var included = obj.included;
    for (var i in obj) {
      klass.fn[i] = obj[i];
    }
    if (included) included(klass)
  };

  return klass;
};
```


这段代码是“类”库的增强版，我们使用 extend() 函数来生成一个类，这个函数的参数是一个对象。通过迭代将对象的属性直接复制到类上：

```
var Person = new Class;

Person.extend({
  find:    function(id) { /* ... */ },
  exists: functions(id) { /* ... */ }
});

var person = Person.find(1);
```

include() 函数的工作原理也是一样的，只不过不是将属性复制至类中，而是复制至类的原型中。换句话说，这里的属性是类实例的属性，而不是类的静态属性。

```
var Person = new Class;

Person.include({
```

```
  save:    function(id) { /* ... */ },
  destroy: functions(id) { /* ... */ }
});

var person = new Person;
person.save();
```

同样地，这里的实现支持 extended 和 included 回调。将属性传入对象后就会触发这两个回调：

```
Person.extend({
  extended: function(klass) {
    console.log(klass, " was extended!");
  }
});
```

如果你基于 Ruby 实现过类，会感觉它的写法与此很相近。这种写法之美在于它已经可以支持模块了。模块是可重用的代码段，用这种方法可以实现各种继承，用来在类之间共享通用的属性。

```
var ORMModule = {
  save: function(){
    // 共享的函数
  }
};

var Person = new Class;
var Asset  = new Class;

Person.include(ORMModule);
Asset.include(ORMModule);
```


基于原型的类继承

我们之前已经提到过 prototype 属性很多次了，但还没有正儿八经地解释过它。现在我们来详细讲解什么是原型，以及如何用它来实现类的继承。

JavaScript 是基于原型的编程语言，原型用来区别类和实例，这里提到一个概念：原型对象（*prototypical object*）。原型是一个“模板”对象，它上面的属性被用做初始化一个新对象。任何对象都可以作为另一个对象的原型对象，以此来共享属性。实际上，可以将其理解为某种形式的继承。

当你读取一个对象的属性时，JavaScript 首先会在本地对象中查找这个属性，如果没有找到，JavaScript 开始在对象的原型中查找，若仍未找到还会继续查找原型的原型，直到查找到 Object.prototype。如果找到这个属性，则返回这个值，否则返回 undefined。

换句话说，如果你给 Array.prototype 添加了属性，那么所有的 JavaScript 数组都具有了这些属性。

为了让子类继承父类的属性，首先需要定义一个构造函数。然后，你需要将父类的新实例赋值给构造函数的原型。代码如下：

```
var Animal = function(){};

Animal.prototype.breath = function(){
  console.log('breath');
};

var Dog = function(){};

// Dog 继承了 Animal
Dog.prototype = new Animal;

Dog.prototype.wag = function(){
  console.log('wag tail');
};
```

现在我们来检查一下继承是否生效了：

```
var dog = new Dog;
dog.wag();
dog.breath(); // 继承的属性
```


给“类”库添加继承

现在来给我们自定义的“类”库添加继承，我们通过传入一个可选的父类来创建新类：

```
var Class = function(parent){
  var klass = function(){
    this.init.apply(this, arguments);
  };

  // 改变 klass 的原型
  if (parent) {
    var subclass = function() { };
    subclass.prototype = parent.prototype;
    klass.prototype = new subclass;
  };

  klass.prototype.init = function(){};

  // 定义别名
  klass.fn = klass.prototype;
```

```
    klass.fn.parent = klass;
    klass._super = klass.__proto__;

    /* include/extend 相关的代码…… */

    return klass;
  };
```

如果将 parent 传入 Class 构造函数，那么所有的子类则必然共享同一个原型。这种创建临时匿名函数的小技巧避免了在继承类的时候创建实例，这里暗示了只有实例的属性才会被继承，而非类的属性。设置对象的 __proto__; 属性并不是所有浏览器都支持，类似 Super.js（*http://github.com/maccman/super.js*）的类库则通过属性复制的方式来解决这个问题，而非通过固有的动态继承的方式来实现。

现在，我们可以通过给 Class 传入父类来实现简单的继承：

```
var Animal = new Class;

Animal.include({
  breath: function(){
    console.log('breath');
  }
});

var Cat = new Class(Animal)

// 用法
var tommy = new Cat;
tommy.breath();
```

12

函数调用

在 JavaScript 中，函数和其他东西一样都是对象。然而，和其他对象不同的是，函数是可调用的。函数内上下文，如 this 的取值，取决于调用它的位置和方法。

除了使用方括号调用函数之外，还有其他两种方法可以调用函数：apply() 和 call()。两者的区别在于传入函数的参数的形式。

apply() 函数有两个参数：第 1 个参数是上下文，第 2 个参数是参数组成的数组。如果上下文是 null，则使用全局对象代替。例如：

```
function.apply(this, [1, 2, 3])
```

call() 函数的行为和 apply() 函数的并无不同，只是使用方法不一样。call() 的第 1 个参数是上下文，后续是实际传入的参数序列。换句话说，这里使用多参数——而不是类

似 apply() 的数组——来将参数传入函数。

```
function.call(this, 1, 2, 3);
```

为什么要更换上下文？这的确是一个问题，因为其他编程语言不允许手动更换上下文也没什么不好。JavaScript 中允许更换上下文是为了共享状态，尤其是在事件回调中。（依个人所见，这是语言设计中的一个错误，因为这会对初学者造成一些困扰，并引入一些 bug。但亡羊补牢为时已晚，你需要花精力来弄清楚它们是如何工作的。）

jQuery 在其 API 的实现中就利用了 apply() 和 call() 来更改上下文，比如在事件处理程序中或者使用 each() 来做迭代时。起初这很让人费解，一旦你理解了就会发现它非常有用：

```
$('.clicky').click(function(){
  // 'this' 指向当前节点
  $(this).hide();
});

$('p').each(function(){
  // 'this' 指向本次迭代
  $(this).remove();
});
```

为了访问原始上下文，可以将 this 的值存入一个局部变量中，这是一种常见的模式，比如：

```
var clicky = {
  wasClicked: function(){
    /* ... */
  },

  addListeners: function(){
    var self = this;
    $('.clicky').click(function(){
      self.wasClicked()
    });
  }
};

clicky.addListeners();
```


然而，我们可以使用 apply 来将这段代码变得更干净一些，通过将回调包装在另外一个匿名函数中，来保持原始的上下文：

```
var proxy = function(func, thisObject){
  return(function(){
    return func.apply(thisObject, arguments);
  });
```

```
};

var clicky = {
  wasClicked: function(){
    /* ... */
  },

  addListeners: function(){
    var self = this;
    $('.clicky').click(proxy(this.wasClicked, this));
  }
};
```

因此在上面的例子中，我们在点击事件的回调中指定了要使用的上下文；jQuery 中调用这个函数所用的上下文就可以忽略了。实际上，jQuery 也包含了实现这个功能的 API，你或许已经猜到了，就是 jQuery.proxy()：

```
$('.clicky').click($.proxy(function(){ /* ... */ }, this));
```

使用 apply() 和 call() 还有其他很有用的原因，比如“委托”。我们可以将一个调用委托给另一个调用，甚至可以修改传入的参数：

```
var App {
  log: function(){
    if (typeof console == "undefined") return;

    // 将参数转换为合适的数组
    var args = jQuery.makeArray(arguments);

    // 插入一个新的参数
    args.unshift("(App)");

    // 委托给 console
    console.log.apply(console, args);
  }
};
```

14 在这个例子中首先构建了一个参数数组，然后将我们自己的参数添加进去，最后将这个调用委托给了 console.log()。你可能对 arguments 变量不熟悉，它是当前调用的作用域内解释器内置的用来保存参数的数组。但它并不是真正的数组，比如它是不可变的，因此我们需要通过 jquery.makeArray() 将其转换为可用的数组。

控制“类”库的作用域

上文提到的 proxy 函数是一个非常有用的模式，我们应当将其添加至我们的“类”库中。我们在类和实例中都添加 proxy 函数，这样就可以在事件处理程序之外处理函数的时候和下面这段代码所示的场景中保持类的作用域：

```
var Class = function(parent){
  var klass = function(){
    this.init.apply(this, arguments);
  };
  klass.prototype.init = function(){};
  klass.fn = klass.prototype;

  // 添加一个 proxy 函数
  klass.proxy = function(func){
    var self = this;
    return(function(){
      return func.apply(self, arguments);
    });
  }

  // 在实例中也添加这个函数
  klass.fn.proxy = klass.proxy;

  return klass;
};
```

现在我们可以使用 proxy() 函数来包装函数，以确保它们在正确的作用域中被调用：

```
var Button = new Class;

Button.include({
  init: function(element){
    this.element = jQuery(element);

    // 代理了这个 click 函数
    this.element.click(this.proxy(this.click));
  },

  click: function(){ /* ... */ }
});
```

如果我们没有使用 proxy 将 click() 的回调包装起来，它就会基于上下文 this.element 来调用，而不是 Button，这会造成各种问题。在新版本的 JavaScript——ECMAScript 5 (ES5)中同样加入了 bind() 函数用以控制调用的作用域。bind() 是基于函数进行调用的，用来确保函数是在指定的 this 值所在的上下文中调用的。例如：

```
Button.include({
  init: function(element){
    this.element = jQuery(element);

    // 绑定这个 click 函数
    this.element.click(this.click.bind(this));
  },

  click: function(){ /* ... */ }
});
```

这个例子和我们的 proxy() 函数是等价的，它能确保 click() 函数基于正确的上下文进行调用。但老版本的浏览器不支持 bind()，幸运的是，如果需要则可以手动实现它。对于老版本的浏览器来说，手动实现的 bind() 兼容性也不错，可直接扩展相关对象的原型，这样就可以像今天在 ES5 中使用 bind() 那样在任意浏览器中调用它。例如，下面就是一段实现了 bind() 函数的代码：

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function (obj) {
    var slice = [].slice,
        args = slice.call(arguments, 1),
        self = this,
        nop = function () {},
        bound = function () {
          return self.apply( this instanceof nop ? this : (obj || {}),
                             args.concat(slice.call(arguments)));
        };

    nop.prototype = self.prototype;

    bound.prototype = new nop();

    return bound;
  };
}
```

如果浏览器原生不支持 bind()，我们仅须重写 Function 的原型。现代浏览器则可以继续使用内置的实现。对于数组来说这种“打补丁”[译注2]式的做法非常有用，因为在新版本的 JavaScript 中，数组增加了很多新的特性。我个人推荐使用 es5-shim（*https://github.com/kriskowal/es5-shim*）项目，因为它涵盖了 ES5 中新增的尽可能多的特性。

译注2： 原文为Shimming，意指“补偿”。

添加私有函数

迄今为止，我们为“类”库添加的属性都是“公开的”，可以被随时修改。现在我们来探究一下如何给“类”添加私有属性。

很多开发者都习惯在私有属性之前冠以下画线前缀（_）。尽管本质上这并不是私有属性，但至少能一眼看出它们就是私有属性，因此它是私有 API 的组成部分。我尽可能地不考虑这种情况，因为它看上去实在太丑陋了。

JavaScript 的确支持不可变属性，然而在主流浏览器中并未实现，我们还没办法直接利用这个特性。相反，我们可以利用 JavaScript 匿名函数来创建私有作用域，这些私有作用域只能在内部访问：

```
var Person = function(){};

(function(){

  var findById = function(){ /* ... */ };

  Person.find = function(id){
    if (typeof id == "integer")
      return findById(id);
  };

})();
```

我们将类的属性都包装进一个匿名函数中，然后创建了局部变量（findById），这些局部变量只能在当前作用域中被访问到。Person 变量是在全局作用域中定义的，因此可以在任何地方都能访问到。

定义变量的时候不要丢掉 var 运算符，因为如果丢掉 var 就会创建全局变量。如果你需要定义全局变量，在全局作用域中定义它或者定义为 window 的属性：

```
(function(exports){
  var foo = "bar";

  // 将变量暴露出去
  exports.foo = foo;
})(window);

assertEqual(foo, "bar");
```

“类”库

为了便于理解本书中的很多概念，最好先理解“类”的一些基础理论，但实际上开发人员往往是直接就去使用一个“类”库。jQuery 本身并不支持类，但通过插件的方式可以轻易引入类的支持，比如 HJS（*http://plugins.jquery.com/project/HJS*）。HJS 允许你通过给 `$.Class.create` 传入一组属性来定义类：

17
```
var Person = $.Class.create({
  // 构造函数
  initialize: function(name) {
    this.name = name;
  }
});
```

可以在创建类的时候传入父类作为参数，这样就实现了类的继承：

```
var Student = $.Class.create(Person, {
  price: function() { /* ... */ }
});

var alex = new Student("Alex");
alex.pay();
```

可以直接给类挂载属性：

```
Person.find = function(id){ /* ... */ };
```

HJS 的 API 中同样包含一些工具函数，比如 `clone()` 和 `equal()`：

```
var alex = new Student("Alex");
var bill = alex.clone();

assert( alex.equal(bill) );
```

HJS 并不是我们的唯一选择，Spine（*http://maccman.github.com/spine*）同样实现了类，通过直接在页面中引入 spine.js（*http://maccman.github.com/spine/spine.js*）来使用它：

```
<script src="http://maccman.github.com/spine/spine.js"> </script>
<script>
  var Person = Spine.Class.create();

  Person.extend({
    find: function() { /* ... */ }
  });

  Person.include({
    init: function(atts){
      this.attributes = atts || {};
```

```
    }
  });

  var person = Person.init();
</script>
```

Spine“类”库的 API 和我们本章所构建的“类”库 API 非常类似。使用 `extend()` 来添加类属性并使用 `include()` 来添加实例属性。通过给 `Spine.Class` 实例传入父类来实现继承。

如果你不想把视野局限于 jQuery 的话，那就多关注一下 Prototype（*http://prototypejs.org/*），它包含很多不错的 API（*http://prototypejs.org/learn/class-inheritance*），并且是其他很多类库的灵感来源。

jQuery 的作者 John Resig 在他的博客中写过一篇文章，专门讲解如何实现经典的类继承（*http://goo.gl/09l0V*），这篇文章也值得一读，尤其是当你想挖掘 JavaScript 原型系统背后的真相的时候。

18

第2章 19

事件和监听

事件是 JavaScript 应用程序的核心，是所有内容的驱动，它决定了在应用程序产生用户交互的起始时刻。然而在 JavaScript 诞生之初“事件”的实现并不标准，甚至非常丑陋。在之后的浏览器大战中网景和微软分道扬镳，它们各自实现的事件模型互不兼容。尽管后来 W3C 对此做了标准化，但 IE 仍然坚持使用与 W3C 不兼容的事件模型，直到最新发布的 IE9 才遵循标准。

幸运的是，有很多诸如 jQuery 和 Prototype 的类库很好地处理了兼容性问题，对外提供了统一的 API 来实现事件。但是了解事件的机制仍然是非常重要的，因此这里首先讲解 W3C 中的事件模型，然后展示各种流行类库的一些实例。

监听事件

绑定事件监听的函数叫做 `addEventListener()`，它有 3 个参数：`type`（比如 *click*）、`listener`（比如 callback）及 `useCapture`（后续会讲到 `useCapture`）。使用前两个参数可以给一个 DOM 元素绑定一个函数，当特定的事件（比如点击）被触发时执行这个函数：

```
var button = document.getElementById("createButton");

button.addEventListener("click", function(){ /* ... */ }, false);
```

可以使用 `removeEventListener()` 来移除事件监听，参数和传入 `addEventListener()` 的一样。如果监听的函数是匿名函数，没有任何引用指向它，在不销毁这个元素的前提下，这个监听是无法被移除的：

```
var div = document.getElementById("div");

var listener = function(event) { /* ... */ };
```

```
div.addEventListener("click", listener, false);
div.removeEventListener("click", listener, false);
```

20 带入 listener 函数的第 1 个参数是 event 对象，通过 event 对象可以得到事件的相关信息，比如时间戳、坐标和事件宿主元素（target）。它同样包含很多方法来停止事件冒泡和阻止事件的默认行为。

不同的浏览器对事件类型的支持也不尽相同，但所有现代浏览器都支持这些事件：

- *click*
- *dblclick*
- *mousemove*
- *mouseover*
- *mouseout*
- *focus*
- *blur*
- *change*（表单输入框特有）
- *submit*（表单特有）

可以从 PPK 的文章中（*http://goo.gl/l7Zq*k）查看怪异模式支持的事件类型。

事件顺序

在进一步讨论之前，很有必要介绍一下事件顺序。如果一个节点和它的一个父节点都绑定了相同事件类型的回调，当事件触发时哪个回调会先执行？尽管网景和微软的处理方式不一致，也不要太过担心。

Netscape 4 支持事件捕捉（*capturing*），从顶层的父节点开始触发事件，从外到内传播。

微软则支持事件冒泡（*bubbling*），从最内层的节点开始触发事件，逐级冒泡直到顶层节点，从内向外传播。

我认为事件冒泡看起来更合理一些，这也是我们日常开发所用的事件模型。W3C 对此做了让步，将对这两种事件模型的支持都加入标准规范之中。根据 W3C 模型，事件首先被目标元素所捕捉，然后向上冒泡。

你可以自行选择要注册的事件处理程序的调用类型，捕捉或冒泡，通过给 `addEventListener()` 传入第 3 个参数 `useCapture` 来设置。如果 `addEventListener()` 的最后一个参数是 `true`，事件处理程序以捕捉模式触发；如果是 `false`，事件处理程序以冒泡模式触发：

```
// 给最后一个参数传入 false，来设置事件冒泡
button.addEventListener("click", function(){ /* ... */ }, false);
```

大多数情况下我们都在使用冒泡模式，如果对此不太确定，可以给 addEventListener() 的最后一个参数传入 false。

取消事件

当事件冒泡时，可以通过 stopPropagation() 函数来终止冒泡，这个函数是 event 对象中的方法。比如这段代码，任何父节点的事件回调都不会触发：

```
button.addEventListener("click", function(e){
  e.stopPropagation();
  /* ... */
}, false);
```

此外，一些类库比如 jQuery 还支持 stopImmediatePropagation() 函数，用来阻止后续所有的事件触发——哪怕这些事件是注册在同一个节点元素上的也不例外。

浏览器同样给事件赋予了默认行为。比如，当你点击一个链接时，浏览器的默认行为是载入新页面，当点击一个复选框时，浏览器会将其选中（或取消选中）。在事件传播阶段（之后）会触发这些默认行为，在任何一个事件处理程序中都可以阻止默认行为。可以通过调用 event 对象的 preventDefault() 函数来阻止默认行为，同样也可以通过在回调中返回 false 来实现同样的效果：

```
bform.addEventListener("submit", function(e){
   /* ... */
   return confirm("Are you super sure?");
}, false);
```

如果调用 confirm() 返回 false（用户点击了对话框的取消按钮），这个事件回调函数就返回 false，这样就会取消事件，阻止表单的提交。

事件对象

和上面提到的函数 stopPropagation() 和 preventDefault() 一样，event 对象还包含很多有用的属性。W3C 规范中包含的大部分属性都列在下面，更多信息请参照完整的标准规范（*http://www.w3.org/TR/DOM-Level-2-Events/*）。

事件类型：

bubbles

布尔值，表示事件是否通过 DOM 以冒泡形式触发。

22 事件发生时，反映当前环境信息的属性：

`button`

表示（如果有）鼠标所按下的按钮。

`ctrlKey`

布尔值，表示 Ctrl 键是否按下。

`altKey`

布尔值，表示 Alt 键是否按下。

`shiftKey`

布尔值，表示 Shift 键是否按下。

`metaKey`

布尔值，表示 Meta 键[译注 1] 是否按下。

表示键盘事件的属性：

`isChar`

布尔值，表示当前按下的键是否表示一个字符。

`charCode`

表示当前按键的 unicode 值（仅对 *keypress* 事件有效）。

`keyCode`

表示非字符按键的 unicode 值。

`which`

表示当前按键的 unicode 值，不管当前按键是否表示一个字符。

事件发生时的环境参数：

pageX，*pageY*

事件发生时相对于页面（如 viewport 区域）的坐标。

screenX，*screenY*

事件发生时相对于屏幕的坐标。

译注1： Meta键是以前MIT计算机键盘上的一个特殊键，一般的电脑键盘没有这个键，类似Ctrl和Alt的功能。

和事件相关的元素：

`currentTarget`
: 事件冒泡阶段所在的当前 DOM 元素。

`target,originalTarget`
: 原始的 DOM 元素。

`relatedTarget`
: 其他和事件相关的 DOM 元素（如果有的话）。

不同的浏览器对这些属性的兼容性也不同，尤其是那些不兼容 W3C 的浏览器。幸运的是，诸如 jQuery 和 Prototype 这些类库为我们解决了这些兼容性问题。

事件库

很多时候我们仅仅是将 JavaScript 类库用于事件管理，毕竟手动处理众多浏览器的差异性吃力不讨好。现在我为大家介绍如何使用 jQuery 的 API 来做事件管理，当然使用其他的类库也是不错的选择，比如 Prototype（*http://www.prototypejs.org/*）、MooTools（*http://mootools.net/*）和 YUI（*http://developer.yahoo.com/yui*）。可以参照更多更深入的文档来获取它们各自的 API 信息。

jQuery 的 API 提供了 `bind()` 函数用来跨浏览器绑定事件监听。在一个 jQuery 实例上调用此函数，传入事件名称和回调函数：

```
jQuery("#element").bind(eventName, handler);
```

比如，给一个元素注册点击事件：

```
jQuery("#element").bind("click", function(event) {
  // ...
});
```

jQuery 提供了一些常用事件的快捷方法，比如 *click*、*submit* 和 *mouseover*。看这段代码：

```
$("#myDiv").click(function(){
  // ...
});
```

需要注意的是，使用这个方法之前要确保 DOM 元素是存在的，这一点很重要。例如，应当在页面载入完成后绑定事件，因此需要绑定 window 的 *load* 事件，然后添加监听：

```
jQuery(window).bind("load", function() {
  $("#signinForm").submit(checkForm);
});
```

然而，还有一个比监听 window 的 *load* 事件更好的方法，即 *DOMContentLoaded*。当 DOM 构建完成时触发这个事件，这时图片和样式表可能还未加载完毕。这也就是说这个事件一定会在用户和页面产生交互之前触发。

并不是所有的浏览器都支持 *DOMContentLoaded*，因此 jQuery 将它融入了 ready() 函数，这个函数是兼容各个浏览器的：

```
jQuery.ready(function($)){
  $("#myForm"). bind("submit", function(){ /*...*/});
});
```

实际上，可以不用 ready() 函数而直接将回调函数写入 jQuery 对象。

```
jQuery(function($){
  // 当页面内容可用时调用
});
```

24 切换上下文

关于事件有一点经常让人感到迷惑，那就是调用事件回调函数时上下文的切换。当使用浏览器内置的 addEventListener() 时，上下文从局部变量切换为目标 HTML 元素：

```
new function(){
  this.appName = "wem";

  document.body.addEventListener("click", function(e){
    // 上下文发生改变，因此 appName 是 undefined
    alert(this.appName);
  }, false);
};
```

要想保持原有的上下文，需要将回调函数包装进一个匿名函数，然后定义一个引用指向它。我们在第 1 章已经提到这种模式，即使用代理函数来保持当前的上下文。这在 jQuery 中也是一种很常用的模式，包括一个 proxy() 函数，只需将指定的上下文传入函数即可：

```
$("signinForm").submit($.proxy(function(){ /* ... */ }, this));
```

委托事件

从事件冒泡时开始就发生了事件委托，我们可以直接给父元素绑定事件监听，用来检测在其子元素内发生的事件。这也是类似 SproutCore（*http://www.sproutcore.com/*）这种框架所使用的技术，用来减少应用中的事件监听的数量：

```
// 在 ul 列表上做了事件委托
list.addEventListener("click", function(e){
  if (e.currentTarget.tagName == "li") {
    /* ... */
    return false;
  }
}, false);
```

jQuery 的处理方式更妙，只需给 `delegate()` 函数传入子元素的选择器、事件类型和回调函数即可。如果使用事件绑定的话，就会给每一个 `li` 元素都绑定 *click* 事件，然而使用 `delegate()` 方法就能减少这种事件监听的数量，改善代码性能：

```
// 不要这样做，这样会给每个 li 元素都添加事件监听（非常浪费）
$("ul li").click(function(){ /* ... */ });

// 这样只会添加一个事件监听
$("ul").delegate("li", "click", /* ... */);
```

使用事件委托的另一个好处是，所有为元素动态添加的子元素都具有事件监听。因此，在上面的例子中，在页面载入完成后添加的 `li` 节点同样可以触发点击事件的回调。 25

自定义事件

除了浏览器内置的事件之外，我们也可以触发和绑定自定义事件。实际上，这是架构库的一个好方法——也是 jQuery 的大多数插件所使用的模式。大多数浏览器厂商均未实现 W3C 标准中的自定义事件，可以使用诸如 jQuery 或 Prototype 的类库来使用这个特性。

jQuery 中可以使用 `trigger()` 函数来触发自定义事件。可以通过命名空间的形式来管理事件名称，命名空间中的单词用点号分隔[译注 2]，比如：

```
// 绑定自定义事件
$(".class").bind("refresh.widget",function(){});

// 触发自定义事件
$(".class").trigger("refresh.widget");
```

通过给 `trigger()` 传入一个额外的参数来给事件处理程序传入数据。数据会以附加参数的形式带入回调：

```
$(".class").bind("frob.widget", function(event, dataNumber){
  console.log(dataNumber);
});
```

译注2： 用点号分隔只是一种约定，并无特殊含义，点号在jQuery中比较常用，而在YUI中自定义事件名称通常使用冒号分隔，比如ddm:start。

```
$(".class").trigger("frob.widget", 5);
```

和内置事件一样，自定义事件同样会沿着 DOM 树做冒泡。

自定义事件和 jQuery 插件

jQuery 插件的实现深受自定义事件机制的影响，同样，自定义事件也是处理与 DOM 产生交互的代码逻辑片段之间耦合的很好的架构方法。如果你对 jQuery 插件不甚了解，请移步附录 B，附录 B 中更深入地讲解了 jQuery。

当你想给你的应用添加一个功能片段时，或许经常纠结于是否应当将这个片段抽离为一个插件。自定义事件的思路可以帮你做这种解耦，并逐渐形成一个可复用的库。

比如，我们来看一个简单的 jQuery 插件——选项卡。我们让 `ul` 列表来响应点击事件。当用户点击一个列表项，给这个列表项添加一个名为 *active* 的类，同时将其他列表项中的 *active* 类移除：

```
<ul id="tabs">
  <li data-tab="users">Users</li>
  <li data-tab="groups">Groups</li>
</ul>

<div id="tabsContent">
  <div data-tab="users"> ... </div>
  <div data-tab="groups"> ... </div>
</div>
```


另外，id 为 `tabsContent` 的 div 用来存放每个选项卡对应的实际内容。根据当前激活的选项卡，来对应地给 div 的子节点添加或删除 *active* 类。实际的显示和隐藏选项卡和内容都由 CSS 来控制，我们的插件仅仅处理 *active* 类：

```
jQuery.fn.tabs = function(control){
  var element = $(this);
  control = $(control);

  element.find("li").bind("click", function(){
    // 从列表项中添加或删除 active 类
    element.find("li").removeClass("active");
    $(this).addClass("active");

    // 给 tabContent 添加或删除 active 类
    var tabName = $(this).attr("data-tab");
    control.find(">[data-tab]").removeClass("active");
    control.find(">[data-tab='" + tabName + "']").addClass("active");
  });
```

```
  // 激活第 1 个选项卡
  element.find("li:first").addClass("active");

  // 返回 this 以启用链式调用
  return this;
};
```

插件位于 jQuery 的 prototype 里，因此可以基于 jQuery 实例来调用：

```
$("ul#tabs").tabs("#tabContent");
```

现在看上去插件有什么问题吗？没错，我们给所有的列表项都添加了 *click* 事件回调，这是第 1 个错误。我们可以使用上文提到的 delegate() 来优化代码。同样，点击事件回调的实现很臃肿，很难一眼看出发生了什么。除此之外，如果另一个开发者想要扩展这个插件，他很可能会将其重写。

我们来看下如何使用自定义事件让代码变得更整洁。在点击选项卡时触发一个 *change.tabs* 事件，并绑定若干回调方法来适当修改 *active* 类：

```
jQuery.fn.tabs = function (control) {
  var element = $(this);
  control = $(control);

  element.delegate("li", "click", function () {
    // 遍历选项卡名称
    var tabName = $(this).attr("data-tab");

    // 在点击选项卡时触发自定义事件
    element.trigger("change.tabs", tabName);
  });

  // 绑定到自定义事件
  element.bind("change.tabs", function (e, tabName) {
    element.find("li").removeClass("active");
    element.find(">[data-tab='" + tabName + "']").addClass("active");
  });

  element.bind("change.tabs", function (e, tabName) {
    control.find(">[data-tab]").removeClass("active");
    control.find(">[data-tab='" + tabName + "']").addClass("active");
  });

  // 激活第 1 个选项卡
  var firstName = element.find("li:first").attr("data-tab");
  element.trigger("change.tabs", firstName);

  return this;
};
```

我们看到使用自定义事件回调可以让代码更加整洁。这也意味着选项卡状态切换回调彼此分离，这也让插件代码更具扩展性。比如我们可以在程序中直接更改选项卡的状态，只需触发被观察列表的 *change.tabs* 事件即可：

```
$("#tabs").trigger("change.tabs", "users");
```

同样，我们可以将切换选项卡的动作和窗口的 hash 做关联，这样就可以使用浏览器的后退按钮了：

```
$("#tabs").bind("change.tabs", function(e, tabName){
  window.location.hash = tabName;
});

$(window).bind("hashchange", function(){
  var tabName = window.location.hash.slice(1);
  $("#tabs").trigger("change.tabs", tabName);
});
```

自定义事件的运用实际上给其他开发者很大的空间来扩展我们的工作成果。

DOM 无关事件

基于事件的编程非常强大，因为它能让你的应用架构充分解耦，让功能变得更加内聚且具有更好的可维护性。事件本质上是和 DOM 无关的，因此你可以很容易地开发出一个事件驱动的库。这种模式称为发布 / 订阅（*http://en.wikipedia.org/wiki/Publish/subscribe*），这是一个很有用的模式。

发布 / 订阅（Pub/Sub）是一种消息模式，它有两个参与者：发布者和订阅者。发布者向某个信道（channel）发布一条消息，订阅者绑定这个信道，当有消息发布至信道时就会接收到一个通知。最重要的一点是，发布者和订阅者是完全解耦的，彼此并不知晓对方的存在。两者仅仅共享一个信道名称。

发布者和订阅者的解耦可以让你的应用易于扩展，而不必引入额外的交叉依赖和耦合，从而提高了应用的可维护性，添加额外功能也非常容易。

那么，应当如何在应用中使用发布 / 订阅（Pub/Sub）模式呢？你只需记录回调和事件名称的对应关系及调用它们的方法。看一下这个例子，这段代码中实现了 PubSub 对象，用它可以添加并触发事件监听：

```
var PubSub = {
  subscribe: function(ev, callback) {
    // 创建 _callbacks 对象，除非它已经存在了
    var calls = this._callbacks || (this._callbacks = {});
```

```
    // 针对给定的事件 key 创建一个数组，除非这个数组已经存在
    // 然后将回调函数追加到这个数组中
    (this._callbacks[ev] || (this._callbacks[ev] = [])).push(callback);
    return this;
  },

  publish: function() {
    // 将 arguments 对象转换为真正的数组
    var args = Array.prototype.slice.call(arguments, 0);

    // 拿出第 1 个参数，即事件名称
    var ev = args.shift();

    // 如果不存在 _callbacks 对象，则返回
    // 或者如果不包含给定事件对应的数组
    var list, calls, i, l;
    if (!(calls = this._callbacks)) return this;
    if (!(list = this._callbacks[ev])) return this;

    // 触发回调
    for (i = 0, l = list.length; i < l; i++)
      list[i].apply(this, args);
    return this;
  }
};

// 使用方法
PubSub.subscribe("wem", function(){
  alert("Wem!");
});

PubSub.publish("wem");
```

你可以使用命名空间的方式来管理事件名称，比如使用冒号分隔符（:）。

```
PubSub.subscribe("user:create", function(){ /* ... */ });
```

如果你在使用 jQuery，可以关注下 Ben Alman（*http://benalman.com*）写的一个更容易 29
的库（*http://gist.github.com/799721/c119783954e1b10551c4afef53b2c04fefcb7465*）。这个库非常简单，用不了一页纸：

```
/*!
 * jQuery Tiny Pub/Sub - v0.3 - 11/4/2010
 * http://benalman.com/
 *
 * Copyright (c) 2010 "Cowboy" Ben Alman
 * Dual licensed under the MIT and GPL licenses.
 * http://benalman.com/about/license/
```

```
 */

(function($){
  var o = $({});

  $.subscribe = function() {
    o.bind.apply( o, arguments );
  };

  $.unsubscribe = function() {
    o.unbind.apply( o, arguments );
  };

  $.publish = function() {
    o.trigger.apply( o, arguments );
  };
})(jQuery);
```

这里的 API 和 jQuery 的 `bind()` 及 `trigger()` 函数的参数一致。唯一的区别就是这两个函数直接保存在 `jQuery` 对象中，且名叫 `publish()` 和 `subscribe()`：

```
$.subscribe( "/some/topic", function( event, a, b, c ) {
  console.log( event.type, a + b + c );
});

$.publish( "/some/topic", "a", "b", "c" );
```

上面我们将 Pub/Sub 用于全局事件，也能很容易地将其用于局部事件。现在我们用上面提到的 PubSub 对象来创建一个对象的局部事件：

```
var Asset = {};

// 添加 PubSub
jQuery.extend(Asset, PubSub);

// 现在就可以用 publish/subscribe 函数了
Asset.subscribe("create", function(){
  // ...
});
```

30 我们使用了 jQuery 的 `extend()` 来将 PubSub 的属性复制至 `Asset` 对象。这样的话，所有对 `publish()` 和 `subscribe()` 的调用均是针对 `Asset` 的局部调用。这在很多场景中非常有用，包括对象关系映射（ORM）中的事件、状态机及当 Ajax 请求结束时的回调等场景。

Web Development

基于MVC的JavaScript Web富应用开发

开发JavaScript富应用程序，将桌面程序的用户体验带入Web应用程序中，不是一件简单的事情，因为需要将服务器端的复杂度移植到客户端。本书将教你如何构建优雅又不失高水准的应用程序，包括架构、模板、框架和服务器端的通信及很多其他的方面。

本书提供大量的实例代码，这些实例代码能帮助你更好地理解书中提到的很多概念。学会构建JavaScript应用也会提升你的编码质量，改善你的产品的用户体验。

- 使用模型-视图-控制器（MVC）模式，学习如何管理应用程序中的模块依赖。
- 介绍模板系统和数据绑定。
- 学习如何加载远程数据、Ajax以及跨域请求。
- 使用WebSocket和Node.js创建实时应用。
- 使用拖曳文件的方式来实现文件上传进度条。
- 学习使用主流的框架和库，包括jQuery、Spine和Backbone。
- 编写测试代码，并使用控制台来调试你的程序。
- 学习部署应用的最佳实践，比如缓存控制和代码压缩

"强烈向读者推荐此书，它将会教你如何构建先进的富应用程序。书中给出的很多优秀的工具和最佳实践都是很多程序员和工程师在工作中亟需的。我已经记不起向多少人推荐过这本书了。"

——Addy Osmani
美国在线JavaScript工程师

Alex MacCaw是一名Ruby/JavaScript程序员，在开源社区中很有名望，是Spine框架的作者，开发过Taskforce，Socialmod等大型开源项目，同时活跃在纽约、旧金山和柏林的各大Ruby/Rails会议。

O'REILLY®
oreilly.com.cn

图书分类：Web开发/JavaScript

策划编辑：张春雨
责任编辑：付睿

Broadview®
www.phei.com.cn

O'Reilly Media, Inc.授权电子工业出版社出版

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 in the mainland of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-121-10956-0
9 787121 109560 >

定价：59.00元