

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 第一本专注于讲纯高端 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此书。

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，心里已经觉得很满足。

李晶（拔赤），张散集（一舟）

2011 年 12 月 北京

目录

前言	xv
第1章 MVC和类	1
之初	1
增加结构	2
什么是 MVC	2
模型	3
视图	4
控制器	5
向模块化进军，创建类	6
给类添加函数	7
给“类”库添加方法	8
基于原型的类继承	10
给“类”库添加继承	11
函数调用	12
控制“类”库的作用域	15
添加私有函数	17
“类”库	18
第2章 事件和监听	21
监听事件	21
事件顺序	22

取消事件	23
事件对象	23
事件库	25
切换上下文	26
委托事件	26
自定义事件	27
自定义事件和 jQuery 插件	28
DOM 无关事件	30
第3章 模型和数据	33
MVC 和命名空间	33
构建对象关系映射 (ORM)	34
原型继承	35
添加 ORM 属性	36
持久化记录	37
增加 ID 支持	39
寻址引用	40
装载数据	41
直接嵌套数据	42
通过 Ajax 载入数据	42
JSONP	46
跨域请求的安全性	46
向 ORM 中添加记录	47
本地存储数据	47
给 ORM 添加本地存储	49
将新记录提交给服务器	51
第4章 控制器和状态	53
模块模式	54
全局导入	54
全局导出	54
添加少量上下文	55
抽象出库	56

文档加载完成后载入控制器	58
访问视图	59
委托事件	61
状态机	63
路由选择	65
使用 URL 中的 hash	65
检测 hash 的变化	66
抓取 Ajax	67
使用 HTML5 History API	68
第5章 视图和模板	71
动态渲染视图	71
模板	73
模板 Helpers	75
模板存储	75
绑定	77
模型中的事件绑定	78
第6章 依赖管理	81
CommonJS	82
模块的声明	83
模块和浏览器	83
模块加载器	84
Yabble	84
RequireJS	85
包装模块	87
模块的按需加载	88
LABjs	89
无交互行为内容的闪烁 (FUBC)	89
第7章 使用文件	91
浏览器支持	91
获取文件信息	92
文件输入	92

拖拽	93
拖拽	94
释放拖拽	96
撤销默认的 Drag/Drop	97
复制和粘贴	97
复制	98
粘贴	99
读文件	100
二进制大文件和文件切割	101
自定义浏览器按钮	102
上传文件	102
Ajax 进度条	104
jQuery 拖拽上传	106
创建拖拽目标区域	106
上传文件	107
第8章 实时Web	109
实时 Web 的发展历史	109
WebSocket	110
Node.js 和 Socket.IO	114
实时架构	116
感知速度	117
第9章 测试和调试	119
单元测试	121
断言	121
QUnit	122
Jasmine	126
驱动	128
无界面的测试	131
Zombie	132
Ichabod	134
分布式测试	135

提供支持	136
调试工具	136
Web Inspector	136
Firebug	138
控制台	139
控制台函数	140
使用 JavaScript 调试器	141
分析网络请求	143
Profile 和函数运行时间	144
第10章 部署	147
性能	147
缓存	148
源码压缩 (Minification)	150
Gzip 压缩	151
使用 CDN	152
审查工具	153
外部资源	154
第11章 Spine类库	155
设置	156
类	156
实例化	156
类扩展	157
上下文	158
事件	159
模型	160
获取记录	161
模型事件	162
校验	163
持久化	163
控制器	165
代理	166

元素	167
委托事件	167
控制器事件	168
全局事件	168
渲染模式	169
元素模式	169
构建联系人管理应用	171
联系人模型	172
侧边栏控制器	173
联系人控制器	175
应用程序控制器	178
第12章 Backbone类库	181
模型	182
模型和属性	182
集合	184
控制集合的内部顺序	185
视图	185
渲染视图	186
委托事件	187
绑定和上下文	187
控制器	188
与服务器的同步	190
填充集合	192
服务器端	192
自定义行为	193
构建 To-Do 列表应用	195
第13章 JavaScriptMVC类库	203
设置	204
Class	204
实例化	205
调用基类的方法	205

代理	205
静态继承	206
自省	206
一个模型的例子	207
模型	207
属性和可观察	208
扩展模型	210
Setter	210
Defaults	211
辅助方法	211
服务封装	212
类型转换	215
CRUD 事件	216
在视图中使用客户端模板	216
基本用法	217
jQuery 修改器	217
用 Script 标签加载	217
\$.View 和子模板	218
延时对象	218
打包、预加载和性能	219
\$.Controller : jQuery 插件工厂	220
概览	222
控制器实例化	222
事件绑定	223
模板动作	224
大综合：一个抽象的 CRUD 列表	225
附录A jQuery基础	227
附录B CSS扩展	239
附录C CSS3参考	245
索引	267

作者介绍	281
译者介绍	281
封面图片介绍	282

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，用户使用 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，增强了体验。

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，本章主要就是介绍这个类库。本章会涵盖 Backbone 的核心理念和类，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的话，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，因此首先熟悉 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。

在本书中凡是有引用到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（revolves to true）。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，

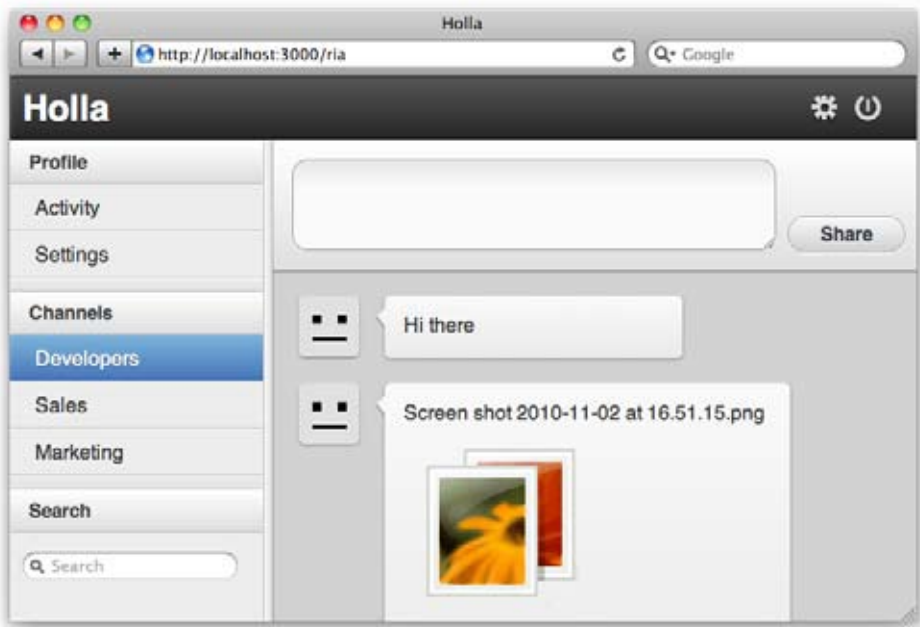
为 DOM 提供了一组非常棒的抽象的接口。可以在附录 A 中查阅到简短的 jQuery 入门内容。

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 Joretteg、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，在这里可以快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，请免费注册 <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>*

MVC和类

之初

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。

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。

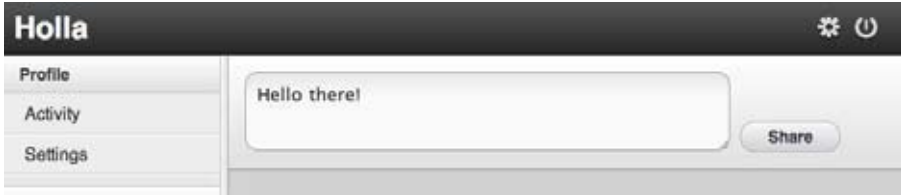


图1-1：从Holla中发送一个新的聊天消息

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。

我们可以不用类库或框架就实现这种 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、他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：

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

而要做到：

```
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*)：和视图有关的小型工具函数。

来看下面的例子，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，对它们进行处理（很可能包含模型），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点击事件的监听。

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 开发者确信他们也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` 和构造函数和传统的实现了类的语言中的使用方法是类似的：

```
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/languageaage/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）^{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);
```

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。——译者注

```

};

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;
};

```

9

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);

```

10

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，若还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
```

◀ 11

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，取决调用它的位置和方法。

除了使用方括号可以调用函数之外，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();
```

13

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 也包含了实现了这个功能的 API，你或许已经猜到了 `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，它是解释器内置的当前调用的作用域内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：

◀ 15

```

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，如果需要的话可以手动实现它。对于老版本的浏览器来说，手动实现的 `bind()` 兼容性也不错，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()`，我们仅仅重写了 `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”^{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。

注2： 原文为 *Shimming*，意指“补偿”。——译者注

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划线前缀（_）。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 并不是我们的惟一选择，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

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，他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/l7Zqk>) 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^{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。

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

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

23

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目：

```
// 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，25 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。

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^{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);
});
```

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`。——译者注

```
$(".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>
```

26

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;
};

```

27

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，彼此并不知道对方的存在。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>) 写的一个更容易的库 (<http://gist.github.com/799721/c119783954e1b10551c4afe53b2c04fefcb7465>)。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。惟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。

模型和数据

将应用重心从后台迁往前台的一个挑战是数据管理。传统方式是通过页面请求从数据库获取数据，用户和页面中的结果进行直接交互。然而在复杂的 JavaScript 应用中做数据管理是非常困难的。前端并没有请求 / 响应模型，也没办法访问服务器端的变量，甚者，远程取回的数据只是临时的保存在客户端。

尽管这种（从后台迁往前台的）转变依然在激烈争论之中，但这还是有一些好处的。比如，客户端数据存储速度非常快，几乎是瞬间读取的，因为数据是直接从内存中获得的。这会让你的应用接口变得与众不同，所有交互操作都会瞬间得到响应，这极大地提高了产品的用户体验。

如何在客户端架构数据存储呢？这需要很多思考。这就像要走出一个充满诱惑和陷阱的迷宫，对于程序员新手来说困难重重，尤其是对于那些想要把自己的应用做大的开发者来说更是如此。本章我们会介绍如何最好地达成这种转变，我也会给出一些推荐的模式和最佳实践。

MVC 和命名空间

要确保应用中的视图、状态和数据彼此清晰分离，这样才能让架构更加整洁有序且更加健壮。引入 MVC 模式，数据管理则归入模型（MVC 中的 M）。模型应当从视图和控制器中解耦出来。与数据操作和行为相关的逻辑都应当放入模型中，通过命名空间进行管理。

在 JavaScript 中，我们通过给对象添加属性来管理一个命名空间，这个命名空间可以是函数，也可以是变量，比如：

```
var User = {
```

```
    records: [ /* ... */ ]
};
```

32 User 的数组数据就在命名空间 `User.records` 中。和 `user` 相关的函数也可以放入 `User` 模型的命名空间里。比如，我们使用 `fetchRemote()` 函数来从服务器端获取 `user` 的数据：

```
var User = {
  records: [],
  fetchRemote: function(){ /* ... */ }
};
```

将模型的属性保存至命名空间中的做法可以确保不会产生冲突，这也是符合 MVC 原则的，同时也能避免你的代码变成一堆函数和回调混杂在一起的大杂烩。

我们可以对命名空间做一点改进，将那些在真实 `user` 对象上的和 `user` 实例相关的函数也都添加进去。假设 `user` 记录包含一个 `destory()` 函数，它是和具体的 `user` 相关的，因此这个函数应当基于 `User` 实例进行调用：

```
var user = new User;
user.destroy()
```

为了做到这一点，我们应当将 `User` 写成一个类，而不是一个简单对象：

```
var User = function(atts){
  this.attributes = atts || {};
};

User.prototype.destroy = function(){
  /* ... */
};
```

对于那些和具体的 `user` 不相关的函数和变量，则可以直接定义在 `User` 对象中：

```
User.fetchRemote = function(){
  /* ... */
};
```

关于命名空间的更多内容，请参阅 Peter Michaux 的博客，其中有一个关于命名空间专题的文章 (<http://goo.gl/FOnPL>)，非常不错。

构建对象关系映射（ORM）

对象关系映射（Object-relational mapper，简称 ORM）是在除 JavaScript 以外的编程语言中常见的一种数据结构。然而在 JavaScript 应用中，对象关系映射也是一种非常有用的技术，它可以用来做数据管理及用做模型。比如使用 ORM 你可以将模型和远程服务捆绑在一起，任何模型实例的改变都会在后台发起一个 Ajax 请求到服务器端。或者你可

以将模型实例和 HTML 元素绑定在一起，任何对实例的更改都会界面中反映出来。这些例子会在之后详细讨论，现在让我们来创建一个自定义 ORM。

本质上讲，ORM 是一个包装了一些数据的对象层。以往 ORM 常用于抽象 SQL 数据库，但在这里 ORM 只是用于抽象 JavaScript 数据类型。这个额外的层有一个好处，我们可以通过给它添加自定义的函数和属性来增强基础数据的功能。比如添加数据的合法性验证、监听、数据持久化及服务器端的回调处理等，这样会增加代码的重用率。

原型继承

这里使用 `Object.create()` 来构造我们的 ORM，这和在第 1 章中提到的基于类的例子有一点不同。这里使用基于原型（prototype-based）的继承，而没有用到构造函数和 `new` 关键字。

`Object.create()` 只有一个参数即原型对象，它返回一个新对象，这个新对象的原型就是传入的参数。换句话说，传入一个对象，返回一个继承了这个对象的新对象。

`Object.create()` 最近才添加进了 ECMAScript 第 5 版规范，因此在有些浏览器中并未实现，比如 IE。但这并不是什么大问题，如果需要的话，我们可以很容易地模拟出这个函数：

```
if (typeof Object.create !== "function")
  Object.create = function(o) {
    function F() {}
    F.prototype = o;
    return new F();
  };
```

这段示例代码摘自 Douglas Crockford 的文章《原型继承》(<http://goo.gl/OLQhp>)。如果想深入挖掘 JavaScript 原型和继承的机制可以参阅这篇文章。

现在来创建 `Model` 对象，`Model` 对象将用于创建新模型和实例：

```
var Model = {
  inherited: function() {},
  created: function() {},

  prototype: {
    init: function() {}
  },

  create: function(){
    var object = Object.create(this);
    object.parent = this;
```

```

    object.prototype = object.fn = Object.create(this.prototype);

    object.created();
    this.inherited(object);
    return object;
},

init: function(){
    var instance = Object.create(this.prototype);
    instance.parent = this;
    instance.init.apply(instance, arguments);
    return instance;
}
};

```

如果你不熟悉 `Object.create()`，这段代码读起来会很晦涩，所以我把它分解来讲。`create()` 函数返回一个新对象，这个对象继承自 `Model` 对象，我们使用它来创建新模型。`init()` 函数返回一个新对象，它继承自 `Model.prototype`——如 `Model` 对象的一个实例：

```

var Asset = Model.create();
var User = Model.create();

var user = User.init();

```

添加 ORM 属性

现在如果给 `Model` 对象添加属性，对于继承的模型来说，这些新增属性都是可访问的：

```

// 添加对象属性
jQuery.extend(Model, {
    find: function(){}
});

// 添加实例属性
jQuery.extend(Model.prototype, {
    init: function(atts) {
        if (atts) this.load(atts);
    },

    load: function(attributes){
        for(var name in attributes)
            this[name] = attributes[name];
    }
});

```

`jQuery.extend()` 只是代替 `for` 循环手动复制属性的一种快捷方式，这和我们在 `load()` 函数中的做法差不多。现在，我们的对象和实例属性都传播到了单独的模型里：

```

assertEqual( typeof Asset.find, "function" );

```

实际上我们会增加很多属性，因此还需将 `extend()` 和 `include()` 添加至 `Model` 对象中：

```
var Model = {
  /* .....代码片段.....*/

  extend: function(o){
    var extended = o.extended;
    jQuery.extend(this, o);
    if (extended) extended(this);
  },

  include: function(o){
    var included = o.included;
    jQuery.extend(this.prototype, o);
    if (included) included(this);
  }
};

// 添加对象属性
Model.extend({
  find: function(){}
});

// 添加实例属性
Model.include({
  init: function(atts) { /* ... */ },
  load: function(attributes){ /* ... */ }
});
```

35

现在我们可以创建新的资源并设置一些属性：

```
var asset = Asset.init({name: "foo.png"});
```

持久化记录

我们需要一种保持记录持久化的方法，即将引用保存至新创建的实例中以便任何时候都能访问它。我们通过在 `Model` 中使用 `records` 对象来实现。当我们保存一个实例的时候，就将其添加进这个对象中；当删除实例时，和将它从对象中删除：

```
// 用来保存资源的对象
Model.records = {};

Model.include({
  newRecord: true,

  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this;
  }
});
```

```

    },

    destroy: function(){
        delete this.parent.records[this.id];
    }
});

```

如何更新一个已存在的实例呢？简单——只需更新对象引用即可：

```

Model.include({
    update: function(){
        this.parent.records[this.id] = this;
    }
});

```

现在创建一个快捷函数来保存实例，这样就不用每次都检查这个实例是否已经保存过或是否需要新创建实例了：

```

// 将对象存入 hash 记录中，保持一个引用指向它
Model.include({
    save: function(){
        this.newRecord ? this.create() : this.update();
    }
});

```

36

如何实现一个用来根据 ID 查找资源的 find() 函数呢？

```

Model.extend({
    // 通过 ID 查找，找不到则抛出异常
    find: function(id){
        return this.records[id] || throw("Unknown record");
    }
});

```

现在我们已经成功地创建了一个基本的 ORM，我们来运行一下：

```

var asset = Asset.init();
asset.name = "same, same";
asset.id = 1;
asset.save();

var asset2 = Asset.init();
asset2.name = "but different";
asset2.id = 2;
asset2.save();

assertEqual( Asset.find(1).name, "same, same" );

asset2.destroy();

```

增加 ID 支持

此时，每次我们保存一条记录都必须手动指定一个 ID。这实在是糟透，但幸运的是，我们可以加入自动化处理。首先，我们需要一个方法来自动生成 ID，可以使用全局统一标识（Globally Unique Identifier，简称 GUID）生成器来做这一步。从技术的角度讲，出于 API 的原因，JavaScript 无法友好正式地生成 128 位的 GUID，它只能生成伪随机数。生成真正随机的 GUID 是一个众所周知的难题，操作系统使用 MAC 地址、鼠标位置、BIOS 的校验和、测量电信号的噪声或者检测放射性衰变来计算 GUID，甚至用上了熔岩灯^{注 1}。虽然 JavaScript 中内置的 `Math.random()` 方法尽管产生的是伪随机数，但也足够用了。

Robert Kieffer 写了一个简单明了的 GUID 生成器，它使用 `Math.random()` 来产生一个伪随机数的 GUID (<http://goo.gl/0b0hu>)，代码也很简单：

```
Math.guid = function(){  
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {  
    var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);  
    return v.toString(16);  
  }).toUpperCase();  
};
```

37

现在我们有了解生成 GUID 的方法，可以很容易将它集成到 ORM 里，剩下的只需修改 `create()` 函数了：

```
Model.extend({  
  create: function(){  
    if ( !this.id ) this.id = Math.guid();  
    this.newRecord = false;  
    this.parent.records[this.id] = this;  
  }  
});
```

这样任何新创建的记录都包含一个随机的 GUID 作为它们的 ID：

```
var asset = Asset.init();  
asset.save();  
  
asset.id //=> "54E52592-313E-4F8B-869B-58D61F00DC74"
```

注 1：熔岩灯又称为蜡灯、水母灯。名字源于其内不定形状蜡滴的缓慢流动，让人联想到熔岩的流动。熔岩灯有多种形状和颜色，多用于装饰，更多信息请参照 <http://zh.wikipedia.org/wiki/熔岩灯>。——译者注

寻址引用

如果你足够细心的话，就会发现我们的 ORM 中存在一个与引用相关的 bug。当保存或通过 `find()` 查找记录时，所返回的实例并没有复制一份，因此对任何属性的修改都会影响原始资源。这的确是一个问题，因为我们只想当调用 `update()` 方法的时候才会修改资源：

```
var asset = new Asset({name: "foo"});
asset.save();

// 正确传入资源
assertEqual( Asset.find(asset.id).name, "foo" );

// 更改属性，而没有调用 update()
asset.name = "wem";

// 天那，出问题了，因为 asset 的名字被修改为 “wem”
assertEqual( Asset.find(asset.id).name, "foo" );
```

我们需要修复这个问题，在执行 `find()` 操作的时候新创建一个对象，同样在创建或更新记录的时候需要复制对象：

```
Asset.extend({
  find: function(id){
    var record = this.records[id];
    if ( !record ) throw("Unknown record");
    return record.dup();
  }
});

Asset.include({
  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this.dup();
  },

  update: function(){
    this.parent.records[this.id] = this.dup();
  },

  dup: function(){
    return jQuery.extend(true, {}, this);
  }
});
```

这里存在另外一个问题，`Model.records` 是被所有模型所共享的对象：

```
assertEqual( Asset.records, Person.records );
```

非常不幸，这在合并所有的记录时会有副作用：

```
var asset = Asset.init();
asset.save();

assert( asset in Person.records );
```

解决办法是在创建新的模型时设置一个新的 records 对象。Model.create() 是创建新对象的回调，因此我们可以设置任意描述这个模型的对象：

```
Model.extend({
  created: function(){
    this.records = {};
  }
});
```

装载数据

除非你的 Web 应用完全限制在浏览器中运行，否则应用总是需要从服务器端下载一些数据的。一般地，应用启动时装载的往往是数据的子集，更多的数据则是在交互发生时装载的。根据应用的类型和数据量的大小，页面初始化时可能会将所有的数据一次性加载进来。这是理想状况，这样的话用户就不必再等待额外的数据载入了。但对多数应用来说这种做法并不具可行性，因为数据量太大以至于这些数据没办法在浏览器内存里老老实实呆着。

数据的预加载是一个重要的手段，这能让应用的体验更流畅、速度更快，用户的等待时间也尽可能压缩到很短。但是，在预加载的数据之中可能存在用不到的数据，你需要提前通知用户可能需要哪种类型的数据（或者需要某种方法来判断应用是否依然在线）。

如果你想展示一个可翻页的列表，为什么不预加载下一页的数据，这样在翻页的时候就会更流畅？或者更进一步，展示一个长长的列表时，当滚动到列表所在的位置时就自动加载并插入数据（无限滚动模式）？当用户更少感觉到延迟时，体验就会越好。

◀ 39

当你在取数据的时候，要保证 UI 没有被阻塞。这时需要展示一些加载指示标志，但要确保用户界面依然是可用的。实际工作中很少遇到需要阻塞 UI 的场景。

数据可以直接嵌套显示在初始页面中，或者通过 Ajax 或 JSONP 的方式使用单独的 HTTP 请求加载数据。就我个人而言，我推荐使用后两种技术，因为直接在初始页面中嵌套数据会增加页面体积，而并行加载会更快一些。AJAX 和 JSON 同样允许你将 HTML 页面缓存住，而不是每次渲染都会动态发起请求。

直接嵌套数据

我真的不推荐使用这种方式，原因在上一段落中已经给出了。但在某些特定场景下它还是很有用的，尤其是当下载的数据量很少的时候。而且这种技术的开发成本很低，很易实现。

你只需要直接将 JSON 对象渲染到页面中，比如使用 Rails 上的 Ruby 可以这样来实现：

```
<script type="text/javascript">
  var User = {};
  User.records = <%= raw @users.to_json %>;
</script>
```

我们使用 ERB 标签（<%%>）来输出表示用户数据的 JSON，`raw`方法是防止对 JSON 进行转义处理。当页面渲染输出时，生成的 HTML 结果看起来像这样：

```
<script type="text/javascript">
  var User = {};
  User.records = [{"first_name": "Alex"}];
</script>
```

JavaScript 可以直接处理 JSON 数据，因为 JSON 数据格式本质上就是一个 JavaScript 对象。

通过 Ajax 载入数据

当说到后台请求时，你最先想到的载入远程数据的方法可能就是 Ajax 了，理由很简单：这是一个被充分证明可行而且支持所有现代浏览器的方法。这不是说 Ajax 就没有缺点，历史中的 Ajax 并不标准，因此其 API 也不统一，而且出于安全性的考虑，浏览器默认禁止 Ajax 跨域读取数据。

40 如果你想快速学习 Ajax 和 XMLHttpRequest 类，请阅读 Mozilla 开发者手册 "Getting Started" (http://developer.mozilla.org/en/Ajax/Getting_Started)。在大多数情况下，我们都会使用类库比如 jQuery 来处理 Ajax，因为这些类库可抽象出一套不错的 Ajax API，而且处理了浏览器的兼容性问题。因此这里我们简单讲解一下 jQuery 的 API，这里不会对原生 XMLHttpRequest 类做过多介绍。

jQuery 的 Ajax API 包含一个底层函数 `jQuery.ajax()`，以及一些高级方法，高级方法是对 `jQuery.ajax()` 的抽象，目的是减少编程时要写的代码。`jQuery.ajax()` 接收一个哈希表^{注2}作为参数，哈希表中存放的是配置信息，包括请求参数、content type、回调函数及其他信息。调用这个函数时，jQuery 就会在后台发起一个异步请求。

注2： 哈希表就是键值对的集合，这里就是指一个对象，关于哈希表请参照 http://en.wikipedia.org/wiki/Hash_table。——译者注

url :

请求的 url，默认是当前页面的 url。

success :

当请求成功时触发的回调函数，从服务器返回的任何数据都会以参数形式带入回调函数。

contentType :

设置请求的 Content-Type 头字段。如果请求包含数据，默认值为 application/x-www-form-urlencoded，大多数情况下都是使用这个值。

data :

是发送给服务器的数据，如果它不是字符串，jQuery 会将其做序列化处理并进行 URL 编码。

type :

HTTP 的发送方式：GET、POST 或 DELETE。默认为 GET。

dataType :

希望从服务器返回的数据类型。jQuery 需要知道这个设置以便知道如何处理返回结果。如果没有指定 dataType，jQuery 将会根据 MIME 的类型来猜测响应数据的类型，可选的值包括：

text :

纯文本响应，不需要做额外处理。

script :

jQuery 将响应结果看做 JavaScript 代码并执行它。

json :

jQuery 将响应结果看做 JSON，并将其转换为对象。

jsonp :

处理 JSONP 请求，后续会讨论到。

我们来看个例子，这是一个简单的 Ajax 请求，获取到服务器的数据后将其 alert 出来：

```
jQuery.ajax({  
  url: "/ajax/endpoint",  
  type: "GET",  
  success: function(data) {  
    alert(data);  
  }  
});
```

然而这些选项看起来有点臃肿。幸运的是，jQuery 提供了一些快捷方式，`jQuery.get()` 的参数就很简单，URL、可选的数据和回调函数：

```
jQuery.get("/ajax/endpoint", function(data){
    $(".ajaxResult").text(data);
});
```

或者，如果想要通过 GET 请求发送少量的查询参数的话，可以这样：

```
jQuery.get("/ajax/endpoint", {foo: "bar"}, function(data){
    /* ... */
});
```

如果希望服务器端返回的是 JSON 数据，则需要调用 `jQuerygetJSON()` 来代替，它会自动将请求的 `dataType` 设置为 “json”：

```
jQuery.getJSON("/json/endpoint", function(json){
    /* ... */
});
```

与之类似，jQuery 中还实现了 `jQuery.post()` 函数，同样可以传入 URL、数据和回调函数等参数：

```
jQuery.post("/users", {first_name: "Alex"}, function(result){
    /* Ajax POST 成功 */
});
```

如果想使用其他的 HTTP 方法，比如 DELETE、HEAD 和 OPTIONS，则必须使用底层的 `jQuery.ajax()` 函数。

这里仅仅是对 jQuery 的 Ajax API 的简单介绍，更多详细信息请参阅完整的文档：<http://api.jquery.com/category/ajax>。

Ajax 的一个限制是同源策略 (*same origin policy*)，它要求所有的请求必须来自同一个域名、子域名，并且地址的端口也应当一致。主要原因是出于安全考虑：因为当一个 Ajax 请求被发送，所有的请求都会附带主域的 cookie 信息一起发送。也就是说，对于远程服务来讲，请求如果是来自于登录后的用户，若没有同源策略的限制，攻击者就有可能获取你的 Gmail 里的邮件、得到你的 Facebook 状态或者你 Twitter 中的好友，这是一个非常严重的安全性问题。

但是，尽管出于安全问题的考虑而提出了同源策略，这也对那些的确需要跨域获取合法数据的开发者造成了一些不方便。诸如 Adobe Flash 和 Java 之类的技术已经着手解决了这个问题，通常是使用跨域策略文件。现在 Ajax 也在迎头赶上，提出了名为 CORS (cross-origin resource sharing) 的标准规范 (<http://www.w3.org/TR/access-control>)。

CORS 打破同源策略的限制，赋予了前端代码访问可信的远程服务的权限。主流的浏览器都很好地支持这个规范，除非使用 IE6，基本上可以很好地使用它。

支持 CORS 的浏览器：

- IE ≥ 8 (需要安装 caveat)。
- Firefox ≥ 3 。
- Safari: 完全支持。
- Chrome: 完全支持。
- Opera: 不支持。

CORS 的使用非常简单。如果想将你的服务器添加为受信任的数据源，只需在 HTTP 协议的响应头里加几行：

```
Access-Control-Allow-Origin: example.com
Access-Control-Request-Method: GET,POST
```

这两个头字段会对来自 *example.com* 的跨域 GET 和 POST 请求做验证。多个值之间用逗号分隔，就像上面提到的 GET、POST 值一样。如果要添加多个域名，将域名列在 Access-Control-Allow-Origin 头字段之中，每两个域名之间用逗号分隔。如果允许来自任意域的访问请求，则需要在源中加入通配符 (*)。

对于有些浏览器来说，比如 Safari，它会首先发起一个 OPTIONS 请求以检查服务器是否允许跨域的请求。另一方面，Firefox 则会直接发起跨域请求，但当服务器没有配置 CORS 头字段时会抛出一个安全异常。要注意一下这种浏览器行为的不同。

你甚至可以使用 Access-Control-Request-Headers 头字段来认证自定义的请求头：

```
Access-Control-Request-Headers: Authorization
```

这也意味着客户端可以在 Ajax 请求中添加自定义头，比如使用开放认证 (OAuth)^{注 3} 对请求进行签名：

```
var req = new XMLHttpRequest();
req.open("POST", "/endpoint", true);
req.setRequestHeader("Authorization", oauth_signature);
```

不幸的是，尽管 CORS 是可以正常工作在 IE8 及更高版本的 IE 中的，微软还是选择另辟蹊径，不兼容规范且对 W3C 工作组制定的标准视而不见 (<http://goo.gl/sLc2R>)。微软使用了一个自己的对象 XDomainRequest (<http://msdn.microsoft.com/en-us/library/>

注 3： OAuth(开放认证)是允许从桌面和网络应用以简单标准的方法进行安全 API 认证的开放协议。——译者注

cc288060%28VS.85%29.aspx)，用来代替 XMLHttpRequest 进行跨域通信。它的接口和 XMLHttpRequest 的非常相像，它包含一系列约束和限制 (<http://goo.gl/b3H2N>)，比如只支持 GET 和 POST 方法，不支持验证和自定义字段，最后最让人费解的是，只支持“Content-Type:text/plain”类型的请求。如果你满足了这些限制条件，就可以在 IE 中使用正确的 Access-Control 头字段来实现 CORS 了。

43

JSONP

JSONP (JSON with padding) (<http://goo.gl/cqDT6>) 很早之前就被标准化了，甚至在 CORS 之前。这是另一种从远程服务器端抓取数据的方式。原理是通过创建一个 script 标签，所辖的外部文件包含一段 JSON 数据，数据是由服务器所返回的，作为参数包装在一个函数调用中。script 标签获取脚本文件并不受跨域的限制，所有浏览器都支持这种技术。

来看这个例子，这里有一个 script 标签指向一个远程服务：

```
<script src="http://example.com/data.json"> </script>
```

所请求的文件 *data.json* 中包含一个 JSON 对象，这个对象包装在一个函数调用中：

```
jsonCallback({"data": "foo"})
```

这时我们定义一个全局函数。当加载脚本后，这个函数就会被调用：

```
window.jsonCallback = function(result){  
    // 处理返回结果的相关逻辑  
}
```

可以看到这个过程有点小复杂，幸运的是，jQuery 将其包装成了简洁的 API：

```
jQuery.getJSON("http://example.com/data.json?callback=?", function(result){  
    // 处理返回结果的相关逻辑  
});
```

jQuery 将上面 URL 中最后的问号替换为一个由它创建的随机命名的临时函数。服务器会获取这个 callback 参数，使用这个名字作为回调函数名称返回给客户端。

跨域请求的安全性

如果你的服务器打开了允许来自任何域名的跨域请求或 JSONP 的支持，你不得不考虑安全问题。跨域策略往往会阻止攻击者调用 Twitter 的 API 及获取你的私人数据，CORS 和 JSONP 则改变了这个情况。通过普通的 Ajax 请求，也会把本地 cookie 传给服务器，这样就可以利用 Twitter 的 API 来模拟登录，任何潜在的攻击者都可以完全控制你的账户。

因此何时何地都不要忘记：安全第一。

考虑到这一点，如果你无法控制那些拥有你的 API 访问权限的域名的话，这时使用 CORS/JSONP 时有一些要点需要尤为注意：

- 不要暴露任何敏感信息，比如电子邮件地址。
- 禁用任何操作，比如 Twitter 的“follow”操作。

44

或者还有另外一种减少安全隐患的方法，列出可以访问你的数据的域名白名单，或只使用开放认证（OAuth）的身份识别验证。

向 ORM 中添加记录

向 ORM 中添加数据非常简单。我们只需从服务器抓取数据并更新模型的记录即可。现在给 Model 对象增加 populate() 函数，它会对任何给定的值做遍历、创建实例并更新 records 对象：

```
Model.extend({
  populate: function(values){
    // 重置 model 和 records
    this.records = {};

    for (var i=0, il = values.length; i < il; i++) {
      var record = this.init(values[i]);
      record.newRecord = false;
      this.records[record.id] = record;
    }
  }
});
```

现在我们可以使用 Model.populate() 函数，传入请求的返回数据：

```
jQuery.getJSON("/assets", function(result){
  Asset.populate(result);
});
```

这样任何从服务器返回的记录都会同步到我们的 ORM 中了。

本地存储数据

在过去本地数据存储一直都是瓶颈。惟一可用的方法就是使用 cookies 和类似 Adobe Flash 的插件。Cookies 的 API 非常陈旧，不能存储很大的数据量，并且每次请求都会将 cookies 带回服务器，增加了不必要的带宽。那么对于 Flash 来说，这种插件要尽可能少用。

幸运的是，HTML5 加入了对本地存储的支持，而且主流浏览器都已经实现了本地存储。和 cookies 不同，这些数据妥妥地存放在客户端，且不会发给服务器。而且可存储的数据量非常巨大，不同的浏览器的存储上限也有所不同（甚至不同的版本之间也不同，下面会提到），但至少都能为每个域名提供 5MB 的存储空间：

45

- IE ≥ 8 。
- Firefox ≥ 3.5 。
- Safari ≥ 4 。
- Chrome ≥ 4 。
- Opera ≥ 10.6 。

HTML5 本地存储的规范源自 HTML5 Web Storage specification (<http://www.w3.org/TR/webstorage>)，其中包含两类：*local storage* 和 *session storage*。浏览器关闭后 local storage 数据也能够保持，而 session storage 数据则不存在了。浏览器端所存储的数据是以域名分隔开的，某个域中的脚本存储的数据只能被这个域读取。

你可以使用 `localStorage` 和 `sessionStorage` 对象来分别访问并操作 local storage 和 session storage。设置属性的操作和 JavaScript 中的对象操作一样，这两个对象的设置属性操作一模一样：

```
// 设置一个值
localStorage["someData"] = "wem";
```

WebStorage API 还提供了一些新的特性：

```
// 存储数据的个数
var itemsStored = localStorage.length;

// 设置一个项（是一种 hash 写法）
localStorage.setItem("someData", "wem");

// 得到一个已经存储的项，如果不存在则返回 null
localStorage.getItem("someData"); //=> "wem";

// 删除一个项，如果不存在则返回 null
localStorage.removeItem("someData");

// 清空本地存储
localStorage.clear();
```

数据均存储为字符串，所以如果你想保存的数据是对象或数字，则必须自己做类型转换，如果使用 JSON 的话，则需要将对象先做序列化处理再保存它们，从本地存储中读取 JSON 时也需要将它转换为对象：

```

var object = {some: "object"};

// 序列化并保存一个对象
localStorage.setItem("seriData", JSON.stringify(object));

// 读取并将 JSON 转换为对象
var result = JSON.parse(localStorage.getItem("seriData"));

```

如果你存储的数据超出了上限（通常是每个域名 5MB），再保存额外数据时则会抛出 QUOTA-EXCEEDED-ERR 异常。

给 ORM 添加本地存储

46

现在我们来给 ORM 添加本地存储的支持，这样页面刷新后这些数据也能保存下来。需要首先将记录序列化为 JSON 字符串，然后使用 `localStorage` 对象存储它们。当前的问题是，此时序列化后的对象看起来是这样：

```

var json = JSON.stringify(Asset.init({name: "foo"}));
json //=> '{"parent":{"parent":{"prototype":{}},"records":[],"name":"foo"}'

```

所以需要覆盖我们的模型的序列化 JSON 的方法。首先需要判断哪些属性需要序列化，我们给 `Model` 对象增加一个 `attributes` 数组，它的每个模型都可以用来定位它们的属性：

```

Model.extend({
  created: function(){
    this.records = {};
    this.attributes = [];
  }
});

Asset.attributes = ["name", "ext"];

```

因为每个模型包含不同的属性，因此也不会共享相同的数组引用，`attributes` 属性不会直接设置在 `Model` 中。相反地，当新创建一个模型时我们创建一个新数组，这和 `records` 对象的做法类似。

现在我们来创建 `attributes()` 函数，用以返回包含属性到值的映射的对象：

```

Model.include({
  attributes: function(){
    var result = {};
    for(var i in this.parent.attributes) {
      var attr = this.parent.attributes[i];
      result[attr] = this[attr];
    }
    result.id = this.id;
  }
});

```

```

        return result;
    }
});

```

现在就可以为每个模型设置一个属性数组了：

```
Asset.attributes = ["name", "ext"];
```

并且这个 `attributes()` 函数的返回包含正确属性的对象：

```
var asset = Asset.init({name: "document", ext: ".txt"});
asset.attributes(); //=> {name: "document", ext: ".txt"};
```

覆盖 `JSON.stringify()` 所需要的也仅仅是模型实例中的 `toJSON()` 方法。JSON 库使用这个函数来查找需要序列化的对象，而不是这样序列化 `records` 对象：

47 

```

Model.include({
  toJSON: function(){
    return(this.attributes());
  }
});

```

让我们再来试一下序列化记录的操作，这次结果中的 JSON 包含了正确的属性：

```
var json = JSON.stringify(Asset.records);
json //="{"7B2A9E8D...":{"name":"document","ext":".txt","id":"7B2A9E8D..."}"}"
```

现在的 JSON 序列化操作就变得非常优雅，而给模型添加本地存储的功能则显得有些琐碎。我们给 `Model` 增加了两个函数：`saveLocal()` 和 `loadLocal()`。当保存数据的时候，`Model.records` 对象将转换为数组、做序列化并发送给 `localStorage`：

```

var Model.LocalStorage = {
  saveLocal: function(name){
    // 将记录转换为数组
    var result = [];
    for (var i in this.records)
      result.push(this.records[i])

    localStorage[name] = JSON.stringify(result);
  },

  loadLocal: function(name){
    var result = JSON.parse(localStorage[name]);
    this.populate(result);
  }
};

Asset.extend(Model.LocalStorage);

```

页面加载时从本地存储中读取数据，页面关闭时将数据保存至本地存储中，这的确是个好主意。对于读者来说，应当养成这种处理数据的习惯。

将新记录提交给服务器

在之前的章节中，我们讲到如何使用 jQuery 的 `post()` 函数向服务器发送数据。这个函数有 3 个参数，URL、请求数据和回调函数：

```
jQuery.post("/users", {first_name: "Alex"}, function(result){  
    /* Ajax POST 成功 */  
});
```

现在我们有 `attributes()` 函数，将新记录发送至服务器就显得非常简单了，只需将记录的属性 POST 到服务器即可：

```
jQuery.post("/assets", asset.attributes(), function(result){  
    /* Ajax POST 成功 */  
});
```

如果遵守 REST 的约定，我们希望在创建记录时使用 HTTP POST 请求，在更新记录时使用 PUT 请求。这就需要给 Model 实例增加两个方法——`createRemote()` 和 `updateRemote()`——使用这两个方法可以给服务器发送正确的 HTTP 请求类型：

◀ 48

```
Model.include({  
    createRemote: function(url, callback){  
        $.post(url, this.attributes(), callback);  
    },  
  
    updateRemote: function(url, callback){  
        $.ajax({  
            url:    url,  
            data:   this.attributes(),  
            success: callback,  
            type:   "PUT"  
        });  
    }  
});
```

现在如果在 Asset 实例上调用 `createRemote()`，它的属性就会 POST 给服务器：

```
// 用法：  
Asset.init({name: "jason.txt"}).createRemote("/assets");
```


控制器和状态

从以往的开发经验来看，我们通常是用服务器的 session cookies 来管理状态。因此当用户页面跳转之后，上一个页面的状态就丢失了，只有 cookies 保存了下来。然而 JavaScript 应用往往被限制在单页面，这也意味着可以将状态保存在客户端的内存中。

将状态保存在客户端其中一个主要好处是带来更快速的界面响应。用户和页面产生交互时可以立即得到反馈，而不必花好几秒钟时间等待下一个页面的加载。速度的改善极大地提升了用户体验，这让很多 JavaScript 应用的体验更加愉悦。

但将状态保存在客户端也存在诸多挑战。状态保存在哪里？在本地变量中？或者保存在 DOM 里？这也是大多数开发者困惑之处，状态保存的确是让人很纠结的一件事，正确的保存状态非常重要，处理好状态的保存才能做到更加自如地控制应用的状态。

首先，应当避免将状态或数据保存在 DOM 中，因为根据滑坡理论^{注1}，这会导致程序逻辑变得更加错综复杂且混乱不堪。在我们的例子中使用了 MVC 架构来搭建应用，状态都是保存在应用的控制器里的。

到底什么是控制器？你可以将控制器理解为应用中视图和模型之间的纽带。只有控制器知道视图和模型的存在并将它们连接在一起。当加载页面的时候，控制器将事件处理程序绑定在视图里，并适时地处理回调，以及和模型必要的对接。

创建控制器并不需要依赖任何类库，尽管类库看起来很有用。控制器是模块化的且非常独立，了解这一点非常重要。理想状况下不应该定义任何全局变量，而应当定义完全解耦的功能组件。模块模式是处理组件解耦的非常好的方法。

注1：滑坡理论 (slippery slope) 也称为滑坡谬误，是一种逻辑谬论，即不合理地使用连串的因果关系，将“可能性”转换为“必然性”，以达到某种意欲之结论。——译者注

模块模式

模块模式是用来封装逻辑并避免全局命名空间污染的好方法。使用匿名函数可以做到这一点，匿名函数也是 JavaScript 中被证明最优秀的特性之一。通常是创建一个匿名函数并立即执行它。在匿名函数里的逻辑都在闭包里运行，为应用中的变量提供了局部作用域和私有的运行环境：

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

在执行这个匿名函数之前，我们用一对括号 () 将它包起来。这样才能让 JavaScript 解释器正确地将这段代码解析为一个语句。

全局导入

定义在模块里的变量都是局部变量，因此在全局命名空间中是无法访问它们的。然而应用的全局变量仍都是可用的，从模块的内部可以很容易地访问并操纵它们。开发者往往很难一眼看出哪个全局变量被模块使用了，尤其是当模块数量很多的时候。

另外，隐式的全局变量会让程序变得更慢，因为 JavaScript 解释器不得不遍历作用域链。局部变量的读取会更快、更高效。

非常幸运，模块为我们提供了一种简单的方法来解决这些问题。将全局对象作为参数传入匿名函数，可以将它们导入我们的代码中，这种实现方法比隐式的全局对象更加简洁而且速度更快：

```
(function($){  
    /* ... */  
})(jQuery);
```

在这个例子中，我们将全局变量 jQuery 导入我们的模块里，并将其重命名为 \$。它清晰地表明这个模块中所用到的全局变量是什么，并且对这个全局对象的读取速度很快。实际上，这是一种最佳实践 (<http://goo.gl/w1yy1I>)，用这种模式可以安全地使用 jQuery 的别名 \$，而且你的代码不会和其他的类库产生冲突。

全局导出

我们可以使用类似的技术来实现全局对象的导出。理想状况下应当使用尽可能少的全局变量，但总会有某些特殊场景用到全局变量。可以将页面的 window 导入我们的模块，直接给它定义属性，通过这种方式可以暴露全局变量：

```
(function($, exports){

    exports.Foo = "wem";

})(jQuery, window);

assertEqual( Foo, "wem" );
```

这里我们使用的变量名叫 `exports`，用它来暴露全局变量，这样代码看起来更干净易读，可以直接看出模块创建了哪些全局变量。

添加少量上下文

使用局部上下文是一种架构模块很有用的方法，特别是当需要给事件注册回调函数时。实际情况是，模块中的上下文都是全局的，`this` 就是 `window`：

```
(function(){
    assertEqual( this, window );
})();
```

如果想自定义作用域的上下文，则需要将函数添加至一个对象中，比如：

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

    mod.contextFunction = function(){
        assertEqual( this, mod );
    };

    mod.contextFunction();
})();
```

在 `contextFunction()` 中的上下文不是全局的，而是 `mod` 对象。这时使用 `this` 就不必担心会创建全局变量了。为了展示如何在实际中更好地使用这种技术，我们对例子做进一步修改：

```
(function($){

    var mod = {};

    mod.load = function(func){
        $($.proxy(func, this));
    };

    mod.load(function(){
        this.view = $("#view");
    });
```

```

mod.assetsClick = function(e){
    // 处理点击
};

mod.load(function(){
    this.view.find(".assets").click(
        $.proxy(this.assetsClick, this)
    );
});

})(jQuery);

```

这里新建了 `load()` 函数来处理回调，当页面加载后执行它。注意，我们使用了 `jQuery.proxy()` 来确保回调是基于正确的上下文执行的。

然后，当页面加载时，我们给一个元素添加了点击事件监听，这里使用了局部函数 `assetsClick()` 作为回调函数。创建控制器不会比这更复杂。控制器内所有的状态都是局部的且整洁地封装进模块里，这一点非常重要。

抽象出库

现在我们将它抽象成库，这样就可以在别的模块和控制器中重用它了。我们包含了已有的 `load()` 函数，并添加了新的函数，比如 `proxy()` 和 `include()`：

```

(function($, exports){
    var mod = function(includes){
        if (includes) this.include(includes);
    };
    mod.fn = mod.prototype;

    mod.fn.proxy = function(func){
        return $.proxy(func, this);
    };

    mod.fn.load = function(func){
        $(this.proxy(func));
    };

    mod.fn.include = function(ob){
        $.extend(this, ob);
    };

    exports.Controller = mod;
})(jQuery, window);

```

`proxy()` 保证了函数在局部上下文中执行，对于事件回调来说是非常有用的模式。`include()` 函数只是给控制器添加属性、保存类型功能的快捷方式。

代码中将控制器挂载到 `exports` 对象中，对外暴露为全局的 `Controller` 变量，在模型中我们使用构造函数来实现 `Controller` 对象。来看一个简单的例子，这个例子是根据鼠标是否移过元素来给它添加和删除一个元素的类的：

```
(function($, Controller){

    var mod = new Controller;

    mod.toggleClass = function(e){
        this.view.toggleClass("over", e.data);
    };

    mod.load(function(){
        this.view = $("#view");
        this.view.mouseover(this.proxy(this.toggleClass), true);
        this.view.mouseout(this.proxy(this.toggleClass), false);
    });

})(jQuery, Controller);
```

当页面加载时创建了 `view` 变量，并绑定了一些事件监听。当鼠标移过元素就调用 `toggleClass()`，改变元素的类。可以在本书附加文件中查阅完整的例子：`assets/ch04/modules.html`。

当然，使用除了本地变量之外的上下文意味着使用 `this` 的时候可能写更多的代码。然而，这种技术带来了更优的代码重用机制包括实现对象拼合（`mixin`）。比如我们可以通过给它的 `prototype` 设置属性，将函数添加至每个 `Controller` 实例中：

```
Controller.fn.unload = function(func){
    jQuery(window).bind("unload", this.proxy(func));
};
```

或者可以使用之前定义的 `include()` 函数来扩展出一个独立的控制器，只需给它传入一个对象即可：

```
var mod = new Controller;
mod.include(StateMachine);
```

这段示例代码中的 `StateMachine` 对象可以被其他的模型重复使用，从而避免了复制粘贴代码和代码的自我复制，这样的代码才会更简洁。

文档加载完成后载入控制器

根据目前情况，控制器的一部分在生成 DOM 之前就载入了，另一部分则在页面文档载入完成后触发的回调里。这让人很困惑，因为控制器逻辑在多个阶段执行，这就牵扯到很多文档加载完成的回调函数。

这个问题是可以解决的，就是在 DOM 生成之后统一载入控制器。我非常推荐这种方式，因为这种做法可以让你不必考虑加载页面时 DOM 元素的渲染处于什么阶段。

54 ➤ 现在利用这一技巧来简化我们的类库，这样我们的控制器就更简洁了。Controller 类并不一定非要是构造函数，因为这里并不需要在生成子控制器（subcontroller）时传入上下文：

```
// 使用全局对象作为上下文，而不是 window 对象
// 用来创建全局对象
var exports = this;

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

  mod.create = function(includes){
    var result = function(){
      this.init.apply(this, arguments);
    };

    result.fn = result.prototype;
    result.fn.init = function(){};

    result.proxy = function(func){ return $.proxy(func, this); };
    result.fn.proxy = result.proxy;

    result.include = function(ob){ $.extend(this.fn, ob); };
    result.extend = function(ob){ $.extend(this, ob); };
    if (includes) result.include(includes)

    return result;
  };

  exports.Controller = mod;
})(jQuery);
```

现在我们可以使用新的 Controller.create() 函数来创建控制器，传入一个包含实例属性的对象直接量。注意这里整个控制器都被包装在 jQuery(function() { /* ... */ }) 中。这是 jQuery.ready() 的另一种写法，这句话的意思就是在页面 DOM 节点构建完成后才执行初始化的动作加载控制器。

```

jQuery(function($){
    var ToggleView = Controller.create({
        init: function(view){
            this.view = $(view);
            this.view.mouseover(this.proxy(this.toggleClass), true);
            this.view.mouseout(this.proxy(this.toggleClass), false);
        },

        this.toggleClass: function(e){
            this.view.toggleClass("over", e.data);
        }
    });

    // 实例化控制器，调用 init()
    new ToggleView("#view");
});

```

我们还做了另一个重要的更改，就是根据实例化的情况来将视图元素传入控制器，而不是将元素直接写死在代码中。这一步提炼很重要，因为将代码抽离出来，我们就可以将控制器重用于不同的元素，同时保持代码最短。

55

访问视图

一种常见的模式是一个视图对应一个控制器。视图包含一个 ID，因此可以很容易地传入控制器。然后在视图之中的元素则使用 class 而不是 ID，所以和其他视图中的元素不会产生冲突。这种模式为一种通用实践提供了良好的架构，但用法可以很灵活。

讲到这里，本章中所提到的访问视图的方法无非是使用 jQuery() 选择器，将指向视图的本地引用存储在控制器。后续对视图中的元素的查找则被视图的引用限制住了范围，从而提高了查找速度：

```

// ...
init: function(view){
    this.view = $(view);
    this.form = this.view.find("form");
}

```

但是，这的确意味着控制器中会塞满很多选择器，需要不断地查找 DOM。我们可以在控制器中开辟一个空间专门存放选择器到变量的映射表，用这种方法可以让代码更简洁，比如：

```

elements: {
    "form.searchForm": "searchForm",
    "form input[type=text]": "searchInput"
}

```

这样可以确保变量 `this.seachForm` 和 `this.searchInput` 在控制器实例化时就被创建了，并与 DOM 元素相对应。它们是普通的 jQuery 对象，因此可以像平常那样操纵它，设置事件回调和读取属性。

让我们在控制器中实现对这个 `elements` 映射表的支持，遍历所有的选择器并设置局部变量。这个过程在我们的 `init()` 函数中进行，当控制器实例化时调用 `init()` 函数：

```
var exports = this;

jQuery(function ($) {
  exports.SearchView = Controller.create({
    // 选择器到局部变量名的映射
    elements: {
      "input[type=search]": "searchInput",
      "form": "searchForm"
    },

    // 实例化时调用
    init: function (element) {
      this.el = $(element);
      this.refreshElements();
      this.searchForm.submit(this.proxy(this.search));
    },

    search: function () {
      console.log("Searching:", this.searchInput.val());
    },

    // 私有

    $: function (selector) {
      // 需要一个 `el` 属性，同时传入选择器
      return $(selector, this.el);
    },

    // 设置本地变量
    refreshElements: function () {
      for (var key in this.elements) {
        this[this.elements[key]] = this.$(key);
      }
    }
  });

  new SearchView("#users");
});
```

`refreshElements()` 希望每个控制器都包含当前的元素属性 `el`，`el` 由选择器决定。一旦调用了 `refreshElements()`，就会设置控制器的 `this.searchForm` 和 `this.searchInput`

属性，随后就可以进行绑定事件等 DOM 操作了。

可以在本书附加文件中查阅完整的例子：[assets/ch04/views.html](#)。

委托事件

同样地，我们可以将绑定的事件都移除，并通过一个 `events` 对象来代理，这个 `events` 对象包含事件类型和选择器到回调函数的映射。这和 `elements` 对象非常类似，格式是这样的：

```
events: {events: {  
  "submit form": "submit"  
}}
```

让我们继续，将它添加至我们的 `SearchView` 控制器，和 `refreshElements()` 类似，我们使用 `delegateEvents()` 函数，当控制器实例化的时候调用它。这会解析控制器的 `events` 对象，将它绑定至事件回调。在这个 `SearchView` 例子中，我们希望视图的 `<form>` 提交的时候调用函数 `search()`：

```
var exports = this;  
jQuery(function($){  
  exports.SearchView = Controller.create({  
    // 所有事件名称、选择器和回调的映射  
    events: {  
      "submit form": "search"  
    },  
  
    init: function(){  
      // ...  
      this.delegateEvents();  
    },  
  
    search: function(e){ /* ... */ },  
  
    // 私有  
  
    // 根据第 1 个空格来分割  
    eventSplitter: /^(\w+)\s*(.*)$/,  
  
    delegateEvents: function(){  
      for (var key in this.events) {  
        var methodName = this.events[key];  
        var method      = this.proxy(this[methodName]);  
  
        var match        = key.match(this.eventSplitter);  
        var eventName    = match[1], selector = match[2];
```

57

```

        if (selector === '') {
            this.el.bind(eventName, method);
        } else {
            this.el.delegate(selector, eventName, method);
        }
    }
}
});

```

注意，这里在 `delegateEvents()` 中使用到了 `delegate()` 函数，包括 `bind()` 函数。如果没有提供事件选择器，这个事件将会直接放在 `el` 上。否则就将这个事件“委托”(<http://goo.gl/k8SIW>)给 `el`，在匹配选择器的子元素中触发了这个类型的事件时调用它。这种委托机制有一个优势，就是在含有大量事件监听的场景中，不必给每个元素都添加监听，因为当事件冒泡的时候会动态地触发相应的事件回调。

我们可以将这个增强版的控制器添加至之前的 `Controller` 库，这样在每个控制器中都能重用它们了。这个是最终的示例代码，你可以在这里找到所有的控制器库代码：`assets/ch04/finished_controller.html`：

```

var exports = this;

jQuery(function($){
    exports.SearchView = Controller.create({
        elements: {
            "input[type=search]": "searchInput",
            "form": "searchForm"
        },

        events: {
            "submit form": "search"
        },

        init: function(){ /* ... */ },

        search: function(){
            alert("Searching: " + this.searchInput.val());
            return false;
        },
    });

    new SearchView({el: "#users"});
});

```

状态机

状态机——之前我们称之为“有限状态机”（*Finite State Machines*, FSM）^{注2}——是编写 UI 程序的利器。使用状态机可以轻松地管理很多控制器，根据需要显示和隐藏视图。那么，到底什么是状态机？本质上讲状态机由两部分组成：状态和转换器。它只有一个活动状态，但也包含很多非活动状态（passive state）。当活动状态之间相互切换时就会调用状态转换器。

状态机如何工作呢？考虑这样一个场景，应用中存在一些视图，它们的显示是相互独立的，比如一个视图用来显示联系人，另一个视图用来编辑联系人。这两个视图一定是互斥的关系，其中一个显示时另一个一定是隐藏的。这个场景就非常适合引入状态机，因为它能确保每个时刻只有一种视图是激活的。的确，如果我们想添加一些新视图，比如一个承载设置操作的视图，用状态机来处理这种场景绰绰有余。

我们来对实际的例子做进一步修改，来看一看实现一个状态机的思路是怎样的。这个例子非常简单，没有实现多个转换器类型，但足以满足我们的需要。首先，我们使用 jQuery 的事件 API 创建一个 Events 对象（参照第 2 章），给它添加绑定和触发状态机的事件的能力：

```
var Events = {
  bind: function(){
    if ( !this.o ) this.o = $({});
    this.o.bind.apply(this.o, arguments);
  },

  trigger: function(){
    if ( !this.o ) this.o = $({});
    this.o.trigger.apply(this.o, arguments);
  }
};
```

59

这里的 Events 对象本质上是扩展了 jQuery 现有的 DOM 外部事件的支持，这样我们就可以将它应用到我们的库中。现在我们来创建 StateMachine 类，它包含一个主要的函数 add()：

```
var StateMachine = function(){};
StateMachine.fn = StateMachine.prototype;

// 添加事件绑定或触发行为
$.extend(StateMachine.fn, Events);
```

注 2：有限状态机，又称有限状态自动机，简称状态机，是表示有限个状态及在这些状态之间的转移和动作等行为的数学模型。——译者注

```

StateMachine.fn.add = function(controller){
    this.bind("change", function(e, current){
        if (controller == current)
            controller.activate();
        else
            controller.deactivate();
    });

    controller.active = $.proxy(function(){
        this.trigger("change", controller);
    }, this);
};

```

这个状态机的 `add()` 函数将传入的控制器添加至状态列表，并创建一个 `active()` 函数。当调用 `active()` 的时候，控制器的状态就转换为激活状态。对于激活状态的控制器，状态机将基于它调用 `activate()`，对于其他的控制器，状态机则会调用 `deactivate()`。这里给出两个例子，通过例子可以看出这两类控制器是如何工作的，我们首先将控制器添加至状态机中，然后激活其中一个控制器：

```

var con1 = {
    activate: function(){ /* ... */ },
    deactivate: function(){ /* ... */ }
};

var con2 = {
    activate: function(){ /* ... */ },
    deactivate: function(){ /* ... */ }
};

// 创建一个新的状态机，并添加状态
var sm = new StateMachine;
sm.add(con1);
sm.add(con2);

// 激活第 1 个状态
con1.active();

```

状态机的 `add()` 函数给 `change` 事件创建了一个回调，根据需要调用 `activate()` 或 `deactivate()` 函数。尽管状态机给我们提供了 `active()` 函数，我们同样可以通过手动触发 `change` 事件来改变状态：

60  `sm.trigger("change", con2);`

在控制器 `activate()` 函数的内部，我们可以创建并显示它的视图，添加并显示元素。与此类似，在 `deactivate()` 函数内部我们则将元素销毁来隐藏视图。可以通过 CSS 的类来隐藏和显示视图，这种方法非常不错，即给元素添加名为 `.active` 的类来显示视图，将它移除就可以隐藏视图：

```

var con1 = {
  activate: function(){
    $("#con1").addClass("active");
  },
  deactivate: function(){
    $("#con1").removeClass("active");
  }
};

var con2 = {
  activate: function(){
    $("#con2").addClass("active");
  },
  deactivate: function(){
    $("#con2").removeClass("active");
  }
};

```

然后在样式表中保证视图包含 `.active` 类，否则不包含 `.active` 类：

```

#con1, #con2 { display: none; }
#con1.active, #con2.active { display: block; }

```

在这里可以看到完整的示例代码：assets/ch04/state_machine.html。

路由选择

我们的应用是以单个页面的形式运行的，也就是说 URL 不会改变。这对于用户来说是一个问题，因为用户习惯通过惟一的 URL 来获取 Web 中的资源。此外，用户在浏览网页的时候还经常使用前进和后退按钮。

为了解决这个问题，我们需要将应用的状态反映在 URL 中，建立状态和 URL 的某种对应关系，当应用的状态发生改变时，URL 也随之改变。反之亦然，当 URL 改变时，应用的状态也会发生改变。在页面的初始化加载过程中，我们需要检查 URL 并设置应用的初始状态。

使用 URL 中的 hash

但是，定位本页面所用的 URL（基于 URL）是不能更改的，如果改变则会引起页面的刷新，这是我们要避免发生的。幸好我们有一些解决办法。操作 URL 的一种传统办法是改变它的 hash。hash 不会发送给服务器，因此更改 hash 不会造成页面的刷新。比如，这个 URL 是 Twitter 的页面，它的 hash 值就是 `#!/maccman`：

```
http://twitter.com/#!/maccman
```

可以通过 `location` 对象来读取或修改页面 `hash`：

```
// 设置 hash
window.location.hash = "foo";
assertEqual( window.location.hash , "#foo" );

// 去掉 “#”
var hashValue = window.location.hash.slice(1);
assertEqual( hashValue, "foo" );
```

如果 URL 中没有 `hash`，`location.hash` 则返回空字符串。否则，`location.hash` 和 URL 的 `hash` 部分相等，带有 `#` 前缀。

太过频繁地设置 `hash` 也会影响性能，特别是在移动终端里的浏览器中。因此，如果你对 `hash` 的改动太频繁，就要注意限制这种改动，否则在移动终端里可能会造成页面的频繁滚动。

检测 hash 的变化

以往检测 `hash` 变化的方法是通过轮询的计时器来监听，这种方法非常原始。现在情形有所改观，现代浏览器都支持 `hashchange` 事件。这是一个 `window` 的事件，如果想检测 `hash` 的改变就需要绑定这个监听：

```
window.addEventListener("hashchange", function(){ /* ... */ }, false);
```

使用 jQuery 的代码：

```
$(window).bind("hashchange", function(event){
    // hash 发生改变，更改状态
});
```

当触发 `hashchange` 事件时，我们需要确定应用的当前状态。这个事件的浏览器兼容性非常不错，主流浏览器的最新版本都支持这个事件：

- IE ≥ 8 。
- Firefox ≥ 3.6 。
- Chrome。
- Safari ≥ 5 。
- Opera ≥ 10.6 。

但老的浏览器并不支持这个事件，jQuery 为我们提供了一个有用的插件（<http://goo.gl/Sf4IP>）可以给老版本的浏览器添加 `hashchange` 事件的支持。

62 > 不必太在意页面初始加载时是否触发了这个事件，只要关心 `hash` 发生改变时触发这个事

件即可。如果你的应用中使用了 hash 来处理状态的路由，则需要在页面加载时手动触发这个事件：

```
jQuery(function(){
    var hashValue = location.hash.slice(1);
    if (hashValue)
        $(window).trigger("hashchange");
});
```

抓取 Ajax

由于很多搜索引擎爬虫程序无法运行 JavaScript，因此它们也无法得到动态创建的内容。当然页面的 hash 路由也不会起作用。在爬虫程序的眼中，它们看上去都是相同的 URL，因为 hash 字段从来不会发送给服务器。

如果我们想让纯粹的 JavaScript 应用程序在搜索引擎（比如 Google）中也能运行的话，则必须解决这个显而易见的问题。工程师想到了一个办法，就是创建内容的镜像^{注3}。将这个特殊的静态 HTML 内容的快照发送给爬虫程序，而正常的浏览器则继续使用动态生成内容的方法来展现应用。这增加了工程师的工作量，而且要做很多额外工作，比如浏览器嗅探，通常不推荐在应用中添加浏览器嗅探。幸运的是，Google 对引擎做了改进，它提出了“Ajax 抓取规则”(<http://goo.gl/rhNr9>)。

我们再来看一下我的 Twitter 用户信息页的地址（注意 # 后面的感叹号）：

```
http://twitter.com/#!/macman
```

对于 Google 的爬虫程序来说，看到这个感叹号就知道了当前页面是遵从“Ajax 抓取规则”的。这时爬虫程序就会将这个 URL 转换为下面这种形式，这样当然就不会包含 hash：

```
http://twitter.com/?_escaped_fragment_=/macman
```

这里的 hash 替换成了 URL 中的 `_escaped_fragment_` 参数，在“规则”中称之为“丑陋的 URL”(*ugly URL*)。用户是接触不到这个地址的，爬虫程序则会从这个“丑陋的 URL”抓取内容。这样 hash 片段就可以转换为 URL 参数，服务器就可以精确地定位到要抓取的资源位置，在这里的例子中就是我的 Twitter 页面。

不管“丑陋的 URL”包含多么纯净的 HTML 或文本片段，服务器都可以根据它来定位其资源位置，用这种规则就可以实现资源的索引。因为 Twitter 实现了静态页面的版本，因此只需将爬虫抓取的 URL 重定向到对应的静态页面地址即可：

注3：原文是 parallel universe 即平行的宇宙，意思是说要把由 JavaScript 创建的内容再拼成静态的 html 内容。——译者注

```
curl -v http://twitter.com/?_escaped_fragment_=maccman
302 redirected to http://twitter.com/maccman
```

63 由于 Twitter 使用了暂时重定向（302）而非永久重定向（301），因此在搜索结果中所展示的页面 URL 还是带有 hash 地址的 URL，即由 JavaScript 动态生成内容的页面地址（<http://twitter.com/#!/maccman>）。如果你的站点没有实现静态内容的版本，则在 URL 中带有请求参数 `_escaped_fragment_` 时输出静态的 HTML 或文本片段即可。

一旦给你的站点增加了对“Ajax 抓取规则”的支持，可以使用“Fetch as Googlebot tool”（<http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=158587>）来检查你的工作是否生效。如果你的站点不支持“Ajax 抓取规则”，虽然通过浏览器访问站点是可用的，但很可能无法在搜索引擎中正确展示你的站点带索引的页面的内容。但长期来看，诸如 Google 这种搜索引擎应当给爬虫程序增加 JavaScript 的支持，这样本小节所述的内容就不需要了。

使用 HTML5 History API

History API 是 HTML5 规范的组成部分，利用它可以实现将当前地址替换为任意 URL。你也可以控制是否将新的 URL 添加至浏览器的历史记录中，从而根据需要来控制浏览器的“后退”按钮。和设置地址的 hash 类似，关键是页面不会重新加载，页面状态也会一直保持下来。

支持 History API 的浏览器有：

- Firefox >= 4.0。
- Safari >= 5.0。
- Chrome >= 7.0。
- IE: 不支持。
- Opera >= 11.5。

这个 API 非常简单，主要是 `history.pushState()` 函数。它包含 3 个参数，数据对象、标题和新 URL：

```
// 数据对象可以是任意的，当 popstate 事件触发时传入
var dataObject = {
  createdAt: '2011-10-10',
  author: 'donnamoss'
};

var url = '/posts/new-url';
history.pushState(dataObject, document.title, url);
```

这 3 个参数都是可选的，但我们用它们来控制将哪些内容添加至浏览器的历史堆栈中：

data 对象

这个对象可以是任意的，可以指定你想要的任意自定义对象。当触发 *popstate* 事件的时候随之传入回调（稍后会讲解到这个事件）。

title 参数

大多数浏览器都将其忽略了，但根据规范的描述，这个参数是用来设置浏览器历史记录中的新页面标题的。

64

url 参数

这是一个字符串，表示用来替换当前浏览器地址的 URL。如果是相对地址，则新地址同样使用当前地址的域名、端口和协议。当然也可以传入绝对 URL，但由于安全原因，绝对 URL 也做了限制，必须使用和当前地址相同的域名。

在 JavaScript 应用中使用新 History API，是为了让每个 URL 都和真实的 HTML 对应起来。尽管调用 `history.pushState()` 时浏览器不会请求新的 URL，但当页面刷新的时候就会发起请求了。换句话说，你所传入 API 的每个 URL 都是真实存在的，就像上文提到的对 hash 的设置一样，不能处理完页面的内容片段就万事大吉了，还要对历史记录或者 URL 做相应的修改。

如果你的站点已经实现了展现静态 HTML 的版本的话，这些问题都不是难题，但如果你的应用是纯 JavaScript 应用时，URL 的控制就是一个不得不考虑的问题了。一种解决办法就是忽略 URL 控制，不幸的是这会造成 404（页面没有找到）的问题，因此每个 URL 需要返回一个成功的响应。一种改进办法是在服务器端做相应的检查，来确保 URL 和请求的资源都是可用的。

History API 包含了一些其他特性。`history.replaceState()` 的作用几乎和 `history.pushState()` 一模一样，但它不会给历史记录堆栈添加新东西。可以通过 `history.back()` 和 `history.forward()` 函数来在浏览器历史记录中做跳转。

之前我们提到 *popstate* 事件，这个事件是在页面加载后或 `history.pushState()` 方法调用时触发的。在下面的例子中，`event` 对象包含了 `state` 属性，这个属性就是传递给 `history.pushState()` 的数据对象：

```
window.addEventListener("popstate", function(event){
    if (event.state) {
        // 调用了 history.pushState()
    }
});
```

2 ➤ 你可以对这个事件绑定监听，确保你的应用状态保持和 URL 的一一对应。如果你使用了 jQuery，只需将这个事件当成标准事件来对待即可。因此，为了访问状态对象，你需要访问原始事件：

```
$(window).bind("popstate", function(event){  
    event = event.originalEvent;  
    if (event.state) {  
        // 调用了 history.pushState()  
    }  
});
```

视图和模板

视图是应用的接口，它们为用户提供视觉呈现并与用户产生交互。在这里的场景中，视图是无逻辑的 HTML 片段，由应用的控制器来管理，视图处理事件回调及内嵌数据。可以通过直接将一些逻辑包含进视图中来打破 MVC 的抽象，这看起来蛮有诱惑。但不要沉迷于这种诱惑！不然会导致毫无语义的面条式代码^{注1}。

对于工程师来说，将应用的重心从服务器端迁往客户端的过程中，视图的迁移是最繁重的工作。以往，服务器端的应用往往嵌套很多 HTML 片段及新创建的页面。然而 JavaScript 应用中的视图则有很大不同。

首先，你需要将视图所需的所有数据都迁往客户端，因为你用不着再去访问服务器端变量了。最常用的方法就是用 Ajax，Ajax 返回一个 JSON 对象，然后由应用的模型载入它。你不必在服务器端对 HTML 做预渲染操作，而是将渲染界面的操作全放在客户端。这使得你的客户端应用不依赖于服务器来渲染视图，这种用户界面给人感觉爽快无比。

然后你将数据载入视图之中，不是通过 JavaScript 动态创建 DOM 元素的形式生成视图而是使用模板。本章将会着重讲解这两个主题。

动态渲染视图

通过 JavaScript 程序创建视图有多种方式，其中一种方式是使用 `document.createElement()` 创建 DOM 元素，设置它们的内容并将它们追加至页面中。当需要重绘视图时，只需将视图清空并重复前面的过程：

注1：面条式代码（spaghetti code）是软件工程中反模式的一种，是指一个代码的控制结构复杂、混乱而难以理解，尤其是用了很多 GOTO、异常或其他组织混乱的分支结构，更多内容请参照：<http://zh.wikipedia.org/wiki/面条式代码>。——译者注

```

var views = document.getElementById("views");
views.innerHTML = ""; // 将元素内容清空

var container = document.createElement("div");
container.id = "user";

var name = document.createElement("span");
name.innerHTML = data.name;

container.appendChild(name);
views.appendChild(container);

```

66

或者，使用 jQuery 的 API 来完成相同的操作，代码更加简洁：

```

$("#views").empty();

var container = $("

待渲染的视图的内容不多的时候推荐用这种方法，可能只需创建几个元素即可。将视图元素放在控制器或者状态里，其实是应用的 MVC 架构的一种妥协方法。



除了这种直接将内容拼接进 DOM 节点中的做法，我更推荐将静态 HTML 包含在页面中，在必要的时候显示或隐藏。这会让控制器中所有和视图相关的代码量降到最少，你也可以根据需要更新元素的内容。



举个例子，我们创建一个 HTML 片段用做视图展示：



```

<div id="views">
 <div class="groups"> ... </div>
 <div class="user">

 </div>
</div>

```



现在，我们可以使用 jQuery 选择器来更新视图，控制多个元素的显示或隐藏：



```

$("#views div").hide();

var container = $("#views .user");
container.find("span").text(data.name);
container.show();

```



使用这种方式来生成元素更加可取，因为这让视图和控制器彼此尽可能地分离开来。



72 | 第5章 视图和模板


```

模板

如果你经常在服务器端做套页面的工作，你对模板会很熟悉。已有很多模板类库——你应当根据你所用的 DOM 类库来选择模板类库。然而，它们的大多数都具有相似的语法，接下来会有讨论。

JavaScript 模板的核心概念是，将包含模板变量的 HTML 片段和 JavaScript 对象做合并，把模板变量替换为对象中的属性值。总的来说，JavaScript 中的模板类库的实现原理和其他语言没什么两样，比如 PHP 的 Smarty，Ruby 的 ERB，以及 Python 的字符串格式化。

67

在本书的例子中，我们使用 jQuery.tmpl (<http://api.jquery.com/category/plugins/templates/>) 库作为模板引擎。如果你不想用 jQuery 或者想用其他不同的模板类库，这些示例代码会非常有帮助。大多数模板类库的语法即使不完全相同也非常相近，如果你想从中挑选一个不错的模板类库，可以关注下 Mustache (<http://mustache.github.com/>)，它包含很多语言的实现版本，包括 JavaScript。

jQuery.tmpl 是由微软开发的，是在 John Resig^{注2} 的原始工作 (<http://goo.gl/sFh6c>) 的基础上做的模板插件。这个库目前仍在维护之中，并在 jQuery 官网 (<http://api.jquery.com/jquery.tmpl>) 上有完整的文档。这个库有一个主要的函数 jQuery.tmpl()，可以给它传入一个模板和一些数据，函数会返回渲染好的元素节点，可以将渲染的结果追加至页面里。如果数据是数组的话，对于数组中的每个数据项都会生成渲染好的模板，否则，将只会渲染一个模板：

```
var object = {
  url: "http://example.com",
  getName: function(){ return "Trevor"; }
};

var template = '<li><a href="${url}">${getName()}</a></li>';

var element = jQuery.tmpl(template, object);
// 得到的结果： <li><a href="http://example.com">Trevor</a></li>

$("body").append(element);
```

这里你可以看到我们使用 `${}` 语法来书写插见的变量。不管括号中的变量名是什么，都会根据传入 jQuery.tmpl() 的对象来计算得出要填充的文本，不考虑它是一种属性还是一个函数。

然而模板的功能要比这种纯粹的插值替换强大很多。很多模板库都具有一些高级功能，

注2： John Resig 是 jQuery 的作者。——译者注

诸如条件流（conditional flow）和迭代。你可以通过使用 if 和 else 语句来实现条件流，就像用纯粹的 JavaScript 写出的代码一样。惟一和 JavaScript 语法不同的地方是，这里需将关键字用双括号括起来，以便模板引擎能正确识别它们：

```
{{if url}}
  ${url}
{{/if}}
```

如果指定的属性值不是 false、0、null、""、NaN 或 undefined 就会执行 if 代码块。正如你所看到的，这个代码块在 {{/if}} 之处结束，所以不要忘掉这一点。当数组——比如说聊天消息——是空的时候显示某条消息，这时就能用到这种通用的模式了：

```
{{if messages.length}}
  <!-- Display messages... -->
{{else}}
  <p>Sorry, there are no messages</p>
{{/if}}
```

遍历是所有模板类库都提供的基础功能。使用模板类库的 {{each}} 关键字可以遍历任何 JavaScript 类型，包括 Object 和 Array。如果将对象传入 {{each}}，它将会基于对象的属性做遍历。同样地，将数组传入 {{each}}，它也会基于数组中的每个索引进行遍历。

在每个片段的内部，你可以使用 \$value 变量来访问当前正被遍历的值。显示这个值和上面的嵌入代码的示例一模一样，它用到了 \${\$value}。看一下这个对象：

```
var object = {
  foo: "bar",
  messages: ["Hi there", "Foo bar"]
};
```

然后使用接下来的模板来遍历这个 message 数组，显示每条消息。此外，数组元素的索引也可以使用 \$index 变量来输出：

```
<ul>
  {{each messages}}
    <li>${$index + 1}: <em>${$value}</em></li>
  {{/each}}
</ul>
```

正如代码中所示，jQuery.tmpl 模板 API 非常简单直接。如前所述，大多数可选的模板类库 API 都大同小异，当然也有一些模板提供高级特性，比如 lambda 表达式^{注3}、局部模板^{注4}和注释。

注3： lambda 表达式也叫做入演算，更多内容请参照 http://zh.wikipedia.org/wiki/λ_演算。——译者注

注4： 局部模板就是模板中包含的模板片段。——译者注

模板 Helpers

有时在视图内部使用“通用 helper 函数”（generic helper function）是非常好用的，比如格式化一个日期或数字。然而我们仍要保持良好的 MVC 架构的思路，而不是直接在视图中任意添加感兴趣的函数。比如，让我们来替换一段纯文本中的 `<a></a>` 标签里的链接。通常我们会像下面这段代码这样做，但这是一种错误的方法：

```
<div>
  ${ this.data.replace(
    /(http|https|ftp):\/\/[\/?=&.\-;#~%-]+(?:[\/\s?&.\-;#~%]="-]*>))/g,
    'a target="_blank" href="$1">$1</a> ' ) }
</div>
```

我们应当将它抽象出来，并用命名空间进行管理，而不是直接将函数掺杂进视图中，这样才能保持逻辑和视图之间的解耦。在这个例子中，我们来创建一个单独的 *helper.js* 文件，这个文件包含应用所有的工具函数，比如 `autoLink()` 函数。这样就可以通过这个 helper 来整理我们的视图：

```
// helper.js
var helper = {};
helper.autoLink = function(data){
  var re = /(http|https|ftp):\/\/[\/?=&.\-;#~%-]+(?:[\/\s?&.\-;#~%]="-]*>))/g;
  return(data.replace(re, 'a target="_blank" href="$1">$1</a> '));
};

// template.html
<div>
  ${ helper.autoLink(this.data) }
</div>
```

这里还有一个额外的好处，`autoLink()` 函数是通用的，在应用的任何地方都可以重用它。

模板存储

说到模板存储，有这样一些内容需要考虑：

- 在 JavaScript 中以行内形式存储。
- 在自定义 script 标签里以行内形式存储。
- 远程加载。
- 在 HTML 中以行内形式存储。

其中一些非常适合在 MVC 架构中使用。我个人推荐使用第 2 种方式，即在自定义 script 标签里以行内形式存储模板，原因接下来会给出。

你可以将模板保存在 JavaScript 文件中，但并不推荐这样做，因为这需要将视图的代码放入一个控制器中，这违背了 MVC 架构的原则。

你可以根据需要来通过发送 Ajax 请求动态地加载模板。这种方法的好处是初始化页面的体积非常小，缺点是在模板加载时 UI 的渲染变得很慢。使用 JavaScript 来构建应用的主要原因就是速度问题，所以加载远程资源时就要非常小心，如果因此降低了 UI 的渲染速度则显得得不偿失了。

你还可以将模板保存在页面的 HTML 中，以行内的形式保存。这种方法的好处是它不存在 UI 加载缓慢的问题，这就避免了通过 Ajax 加载模板造成的用户体验上的损失。源代码也更加清晰，模板就在它们将要展示的位置以行内的形式存放。这种方法的缺点也很显而易见，它增加了初始页面的体积。坦白讲，这种体积增加造成的性能损失微不足道，特别是当服务器开启了压缩和缓存的情况下，就更不值得一提了。

70 我推荐使用自定义 script 标签来存放模板，通过 JavaScript 的 ID 来获取模板的引用。这种存放模板的方法非常方便，特别是当你想在很多地方使用它们时。自定义 script 标签还有一个好处就是，浏览器不必对它们进行渲染，而仅将它们解析为内容文本。

如果在页面中以行内形式定义了模板，你可以使用 `jQuery.fn.tmpl(data)` 方法来调用，也就是说在 jQuery 元素上调用 `tmpl()` 方法：

```
<script type="text/x-jquery-tmpl" id="someTemplate">
  <span>${getName()}</span>
</script>

<script>
  var data = {
    getName: function(){ return "Bob" }
  };
  var element = $("#someTemplate").tmpl(data);
  element.appendTo($("#body"));
</script>
```

在程序的后台，`jQuery.tmpl` 会确保模板一旦生成后，解析后的模板就能被缓存住。这将提高程序的运行效率，因为再次使用模板的时候不必第 2 次解析它们。注意，我们首先根据内容生成元素然后将它追加到页面中，这种方式性能比操作已经追加至页面的元素更好，这是一种最佳实践。

尽管你还是渲染了所有在页面中以行内形式保存的模板，这并不意味着服务器端也要遵从这种结构。尽可能地将每个模板放入一个单独的文件中，然后在页面请求的时候拼接进文档中。在第 6 章将要讲到的一些“依赖管理工具”，会帮你完成这个工作，比如 RequireJS。

绑定

从绑定开始，你开始感觉到客户端渲染视图的切实好处。本质上讲，绑定将视图元素和 JavaScript 对象（通常是模型）挂接在一起。当 JavaScript 对象发生改变时，视图会根据新修改后的对象做适时更新。换句话说，一旦你将视图和模型绑定在一起，则当应用的模型更新时视图也会自动渲染。

绑定是一件“大事”。它意味着当记录发生改变时你的控制器不必处理视图的更新，因为这些更新都是在后台自动完成的。使用绑定来架构你的应用同样为构建实时应用打下了基础。在第 8 章会对实时应用做进一步阐述。

因此，为了将 JavaScript 对象和视图绑定在一起，我们需要设置一个回调函数，当对象的属性发生改变时发送一个更新视图的通知。问题是 JavaScript 并没有提供原生方法支持。JavaScript 中也不像 Ruby 和 Python 一样提供了 `method_missing` 功能，并且也不可能使用 JavaScript 的 `getter` 和 `setter` (<http://goo.gl/9haXh>) 来模拟这种行为。但要知道 JavaScript 是一门极其灵活的动态语言，我们可以直接写 `change` 回调：

71

```
var addChange = function(ob){
  ob.change = function(callback){
    if (callback) {
      if ( !this._change ) this._change = [];
      this._change.push(callback);
    } else {
      if ( !this._change ) return;
      for (var i=0; i < this._change.length; i++)
        this._change[i].apply(this);
    }
  };
};
```

这里的 `addChange()` 函数将 `change()` 函数添加至传入的任意对象中。这里的 `change()` 函数的运行和 jQuery 中的 `change` 事件是一模一样的。你可以通过调用 `change()` 来添加回调，或者通过调用 `change()` 来触发事件，不用带入任何参数。来看一个实际的例子：

```
var object = {};
object.name = "Foo";

addChange(object);
object.change(function(){
  console.log("Changed!", this);
  // 这里可以添加更新视图的代码
});

object.change();
```

```
object.name = "Bar";
object.change();
```

因此可以看到我们给这个对象添加了 `change()` 回调，这样就可以执行绑定和触发 *change* 事件了。

模型中的事件绑定

现在我们把刚才绑定的例子更进一步应用于模型里了。当模型记录创建、更新或销毁时，都会触发 *change* 事件，并重新渲染视图。在下面的例子中，我们创建了一个基础的 `User` 类，新建了事件绑定和触发，最后监听了 *change* 事件，当触发 *change* 事件时重新渲染视图：

```
<script>
  var User = function(name){
    this.name = name;
  };
  User.records = []

  User.bind = function(ev, callback) {
    var calls = this._callbacks || (this._callbacks = {});
    (this._callbacks[ev] || (this._callbacks[ev] = [])).push(callback);
  };

  User.trigger = function(ev) {
    var list, calls, i, l;
    if (!(calls = this._callbacks)) return this;
    if (!(list = this._callbacks[ev])) return this;
    jQuery.each(list, function(){ this() })
  };

  User.create = function(name){
    this.records.push(new this(name));
    this.trigger("change")
  };

  jQuery(function($){
    User.bind("change", function(){
      var template = $("#userTpl").tmpl(User.records);

      $("#users").empty();
      $("#users").append(template);
    });
  });
</script>

<script id="userTpl" type="text/x-jquery-tmpl">
```

```
<li>${name}</li>  
</script>
```

```
<ul id="users">  
</ul>
```

现在无论何时修改 `User` 的记录，`User` 的模型的 `change` 事件都会被触发，调用我们模板的回调函数并重绘用户列表。这很有帮助，因为我们只需关注创建和更新用户的记录，而不必担心视图的更新，视图的更新是自动的。比如，我们创建一个新的 `User`：

```
User.create("Sam Seaborn");
```

这时 `User` 的 `change` 事件会被调用，我们的模板会重新渲染，自动将新添加的 `User` 更新至视图里。你可以在这里看到完整的模型绑定的示例代码：<assets/ch05/model.html>。

依赖管理

JavaScript 中缺失了很多计算机高级编程语言所应有的功能特性，其中一个重要的特性就是依赖管理和模块系统。和其他编程语言不同，对于很多学习 JavaScript 的人来说，并不会过多关注命名空间和模块化等这些传统的重要知识点。实际上，诸如 jQuery 之类的流行类库亦不会增强应用的架构性。开发者需要自己解决“应用架构”的问题。我见到过太多太多面条式 JavaScript 代码了，这类代码的缩进排版近乎疯狂，包含数不清的匿名函数的嵌套，下面的代码是不是似曾相识？

```
function() {  
  function() {  
    function() {  
      function() {  
  
      }  
    }  
  }  
}
```

构建巨型应用当然需要引入模块化和命名空间，但还不得不考虑另外一个问题——内置的依赖管理系统。很长一段时间内，我们以为只用 script 标签就足够了，毕竟页面中使用的 JavaScript 代码并不涉及太多的维护和扩展。但当你开始写复杂的 JavaScript 应用的时候，就必须引入依赖管理系统了。手动维护页面的中 script 标签之间的依赖关系根本不可行。代码会变得混乱不堪，看起来像这样：

```
<script src="jquery.js" type="text/javascript" charset="utf-8"></script>  
<script src="jquery.ui.js" type="text/javascript" charset="utf-8"></script>  
<script src="application.utils.js" type="text/javascript" charset="utf-8"></script>  
<script src="application.js" type="text/javascript" charset="utf-8"></script>  
<script src="models/asset.js" type="text/javascript" charset="utf-8"></script>  
<script src="models/activity.js" type="text/javascript" charset="utf-8"></script>
```

```
<script src="states/loading.js" type="text/javascript" charset="utf-8"></script>
<script src="states/search.js" type="text/javascript" charset="utf-8"></script>
<!-- ... -->
```

74 依赖管理系统除了能解决实际的编程复杂度和可维护性的问题，还能解决性能方面的问题。浏览器需要针对每个 JavaScript 文件都发起一个 HTTP 请求，尽管可以将这些请求放入异步队列，但大量的 HTTP 连接总会造成性能的下降，每个连接都包含额外的 HTTP 头信息、Cookie，并都要做 TCP 的三次握手^{注1}。当你的应用是基于 SSL^{注2} 提供服务的话，情况会更加糟糕。

CommonJS

当大家开始关注如何将 JavaScript 应用于服务器端时，引入了很多解决依赖管理问题的建议方法。SpiderMonkey (<http://www.mozilla.org/js/spidermonkey/>) 和 Rhino (<http://www.mozilla.org/rhino/>) 提供了 `load()` 函数，但并没有很好的解决命名空间的问题。Node.js (<http://nodejs.org/>) 提供了 `require()` 函数，用来加载外部资源文件，Node.js 自有的模块系统也使用这种方式来管理。但代码的可移植性并不好，所以当你想在 Node.js 中运行 Rhino 的代码时会不会出问题呢？

为了让代码更具可移植性，则亟需引入一个标准解决方案，让所有的 JavaScript 都能遵照这个标准来实现统一的模块管理系统，这样 JavaScript 代码库就可以运行在所有的环境中了。Kevini Dangoor 按照这个思路提出了 CommonJS 规范。它在一篇博客 (<http://goo.gl/voRNb>) 中首次提出一个公开的标准，这个标准适用于 JavaScript 解释器和开发者，他对此做了一些说明：

JavaScript 需要一个标准的方法来解决加载外部模块和用命名空间的方式谨慎管理模块的问题。命名空间的问题很容易解决。但并没有标准的编程模式来解决（一次性）加载模块的问题。

这不是一个技术问题，而是需要我们一起共同讨论并达成一致——给出构建大型应用的通用方法，群策群力才能让这个标准影响范围更广、看起来更酷。

随着邮件列表 (<http://groups.google.com/group/commonjs>) 的建立，CommonJS (<http://www.commonjs.org/>) 诞生了，随后越来越多的重要组织相继加入。它的发展非常快，很快成为了 JavaScript 模块写法的事实标准。它包含很多标准，包括 IO 接口、底层的套接字流 (Socket stream)，以及单元测试。

注 1：如果服务器开启 Connection:keep-alive 的话，是可以一定程度上减少 TCP 三次握手的次数的。——译者注

注 2：安全套接层 (Secure Sockets Layer, SSL) 是网景公司 (Netscape) 在推出 Web 浏览器首版的同时提出的协议。SSL 采用公开密钥技术，保证两个应用间通信的保密性和可靠性，使客户与服务器应用之间的通信不被攻击者窃听，更多内容参照 <http://zh.wikipedia.org/wiki/传输层安全>。——译者注

模块的声明

声明 CommonJS 模块非常简单、直接。命名空间的概念也被融入在内。模块被分隔为不同的文件，通过给 `exports` 对象添加内容来对外暴露模块的变量和方法，`exports` 变量是在解释器中定义好的：

```
// maths.js
exports.per = function(value, total) {
  return( (value / total) * 100 );
};

// application.js
var Maths = require("./maths");
assertEqual( Maths.per(50, 100), 50 );
```

75

要想使用在模块中定义的函数，只需 `require()` 这个文件即可，同时将运行结果保存在本地变量中。在上面的例子中，*maths.js* 中暴露的方法都存在于 `Maths` 变量中，可以看到模块就是命名空间，并且这种代码可以在所有遵循 CommonJS 规范的 JavaScript 解释器中运行，比如 Narwhal (<http://narwhaljs.org/>) 和 Node.js。

模块和浏览器

那么，如何在客户端 JS 中也实现 CommonJS 呢？因为很多开发者发现在客户端实现的模块加载器和服务器端有所不同，也就是说如果遵照现行版本的 CommonJS 规范，模块都是以异步的方式加载的。在服务器端可以完美地实现 CommonJS 规范，但在浏览器端实现 CommonJS 就不那么容易了，因为它需要阻塞 UI 并适时地执行刚加载的 script 脚本（在客户端则需要避免这种情况的出现）。CommonJS 团队为此提出了一个规范：“模块转换格式” (<http://goo.gl/Vd1vD>)。这种转换格式将 CommonJS 的模块包装在一个回调函数中，以便更好地处理客户端的异步加载。^{注 3}

我们对上一个例子做一些改进。将它包装进一个“转换格式”里，以启用异步加载，这样就可以完美支持浏览器了：

```
// maths.js
require.define("maths", function(require, exports){

  exports.per = function(value, total) {
```

注 3：在客户端里，为了处理模块依赖关系不得不将模块主逻辑包含在某个回调函数中，加载模块的过程实际是“保存回调”的过程，最后统一处理这些回调函数的执行顺序。作为模块加载器鼻祖的 YUI Loader 便是遵循这种逻辑实现的，并在 YUI3 中形成了其独具特色的 `use()` 和 `add()` 模块化编程风格。为了便于理解客户端模块加载器的基本原理，可以参照译者实现的一个小型的类库 Sandbox.js (<https://github.com/jayli/sandbox>)，文档中详细讲解了模块加载器的基本原理。——译者注

```

        return( (value / total) * 100 );
    });

});

// application.js
require.define("application", function(require, exports){

    var per = require("./maths").per;
    assertEqual( per(50, 100), 50 );

}), [".maths"]); // 给出它的依赖 (maths.js)

```

这样就可以通过在浏览器中引入模块加载器来管理并执行我们的模块了。这为我们管理代码带来很多便利，我们可以将代码分隔成模块组件，这是做良好的应用架构设计的秘诀之一，同时我们还获得了依赖管理的支持，包括独立的作用域和命名空间。没错，相同的模块可以运行在浏览器、服务器、桌面应用，以及任何支持 CommonJS 的环境中。换句话说，现在就可以在客户端和服务端之间共享代码了！

76 模块加载器

为了在客户端使用 CommonJS 模块，我们需要引入模块加载器类库。当然，可选的库还有很多，每个类库都各有其优缺点。这里我会讲到多数流行的类库，你可以根据需要自行选择合适的类库。^{注4}

CommonJS 模块格式目前还在修订之中，还有很多提议没有处理。很遗憾，就目前来看，并未有真正完美的官方的模块加载约定，缺少统一的标准会让情况变得更复杂。目前最主要、应用最广泛的两个模块实现是 Transport C(<http://goo.gl/iaksA>)和 Transport D(<http://goo.gl/An9qK>)。如果采用了接下来要讲的任意一个模块加载器类库，必须确保你实现模块的代码格式遵循所采用加载器所支持的格式规范。幸运的是，很多模块加载器的包装工具^{注5}可以兼容，而且会在各自的文档中给出所支持的包装写法。

Yabble

Yabble (<http://github.com/jbrantly/yabble>) 是一款优秀的轻量级的模块加载器。你可以配置 Yabble 来支持通过 XHR 加载模块或者使用 script 标签来加载模块。通过 XHR 来抓取模块的优势是你不必再用转换格式把模块多包装一层了。然而，这种做法的缺点是——

注4： 这里向大家强烈推荐模块加载器 SeaJS (<http://seajs.com/>)，SeaJS 开创性地在客户端实现了同步 `require()`，使得在客户端也能遵循 CommonJS 标准规范进行编码开发，当然，SeaJS 作为客户端的一个类库，仍然存在着无法回避的问题，比如伪阻塞。因此使用 SeaJS 做开发时需要遵守一些额外的约定 (<http://seajs.com/docs/zh-cn/rules.html>)。——译者注

注5： 原文是 wrapping tools，指的是包裹你的模块逻辑代码的最外层的添加模块的语法。——译者注

必须用 `eval()` 来执行模块代码，调试起来很不方便。另外还会遇到跨域的问题，尤其是当使用了 CDN^{注6} 的时候。理想情况是，应当使用 XHR 来实现一些“快餐式”^{注7} 的开发，而不是严谨规范的开发：

```
<script src="https://github.com/jbrantly/yabble/raw/master/lib/yabble.js"> </script>
<script>
  require.setModuleRoot("javascripts");

  // 如果模块通过转换格式做了包装
  // 那么我们就可以使用 script 标签
  require.useScriptTags();

  require.ensure(["application"], function(require) {
    // 应用加载完毕
  });
</script>
```

这个例子中，程序会抓取包装后的 `application` 模块，然后加载它的依赖 `utils.js`，之后才开始运行这个模块。我们可以使用 `require()` 函数来加载模块：

```
<script>
  require.ensure(["application", "utils"], function(require) {
    var utils = require("utils");
    assertEqual( utils.per( 50, 200 ), 25 );
  });
</script>
```

尽管 `utils` 被引用了两次，一次被内联的 `require.ensure()` 函数引用，另一次被 `application` 模块所引用，我们的脚本却非常聪明，可以只加载它一次。但必须确保你的模块所需的所有依赖都加上了转换格式。

◀ 77

RequireJS

RequireJS (<http://requirejs.org>) 是 Yabble 的一个不错的替代品，它是现在最流行的加载器之一。RequireJS 对模块加载的看法略有不同，它遵循“异步模块定义” (Asynchronous Module Definition, 简称 AMD, 见 <http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition>) 格式。主要的不同之处在于 AMD 的 API 是即时计算依赖关系，而不是延迟计算。实际上，RequireJS 完全和 CommonJS 的模块相互兼容，

注 6：CDN 的全称是 Content Delivery Network，即内容分发网络。其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输得更快、更稳定，比如雅虎所用的 `cn.ying.com` 和淘宝使用的 `a.tbcdn.cn`，都属于 CDN。——译者注

注 7：原文是 quick and dirty，意思是“迅速而又随性的”，专指用快速且肮脏的手法来实现一些功能，虽然性能和可维护性都很糟糕，但的确能很有效地实现功能。——译者注

只是包装转换的写法格式不同。^{注8}

为了加载 JavaScript 文件，只需将它们的路径传入 `require()` 函数即可，并指定一个回调函数，当依赖都加载完成后执行这个回调函数：

```
<script>
  require(["lib/application", "lib/utls"], function(application, utls) {
    // 加载完成！
  });
</script>
```

在这个例子中可以看出，`application` 和 `utls` 模块是以回调参数的形式传入的，而不必使用 `require()` 函数来获取它们。

你能引用的不光是模块，RequireJS 同样支持原始的 JavaScript 类库以依赖的形式载入，尤其是 jQuery 和 Dojo。其他的类库可以正常工作，但它们不会以参数形式正确地传入所需的回调函数中。然而，任何以依赖形式载入的类库都需要使用模块格式写法：

```
require(["lib/jquery.js"], function($) {
  // jQuery 加载完毕
  $("#el").show();
});
```

传入 `require()` 的路径是相对于当前文件或模块的路径，除非路径的前缀是 `/`。出于最优化的考虑，RequireJS 推荐你将初始脚本加载器放入一个单独的文件中，这个迷你的类库甚至提供了一种快捷的引入方式：`data-main` 属性：

```
<script data-main="lib/application" src="lib/require.js"></script>
```

设置 `script` 标签的 `data-main` 属性是告知 RequireJS 直接调用 `require()`，将这个属性值以参数传入 `require()` 中，在这个例子中，它将会调用 `lib/application.js` 脚本，它还会加载我们应用所需的其他脚本：

```
// lib/application.js
require(["jquery", "models/asset", "models/user"], function($, Asset, User) {
  //...
});
```

我们刚刚讲了加载模块的方式，那么如何定义模块呢？上文已经提到，RequireJS 使用一种完全不同的语法来定义模块——不是使用 `require.define()`，而是直接使用 `define()` 函数。因为模块都在不同的文件中，因此无须使用显式的命名。第一个参数是它的依赖，是一个字符串组成的数组，随后的参数是一个回调函数，回调函数中包含实际的模块逻辑。

注8：关于 AMD 规范到底是一种“模块书写格式”（Module Authoring Format）还是一种“转换格式”（Transport Format）一直存在争议。——译者注

辑，和 RequireJS 的 `require()` 函数类似，依赖的内容以参数的形式传给回调函数：

```
define(["underscore", "./utils"], function(_, Utils) {
  return({
    size: 10
  })
});
```

默认情况下是没有 `exports` 变量的。如果要从模块中暴露一些变量，只需将数据从函数中返回即可。RequireJS 的模块有一个优势，就是它们已经是被包装好的，因此你不必担心为了兼容浏览器还要再去写转换格式。然而，需要注意的是，这个 API 并不兼容 CommonJS 的模块，即不能写一个同时运行在 Node.js 和浏览器中的模块代码。其实是可以做一些简单的 hack 来让 CommonJS 的模块代码运行在客户端，RequireJS 为 CommonJS 提供了一个“容错层”——只需将现有的模块代码外部用 `define()` 函数包装一层即可：

```
define(function(require, exports) {
  var mod = require("./relative/name");

  exports.value = "exposed";
});
```

回调函数的参数必须和这段示例代码中所示的一模一样，用 `require` 和 `exports`。现在你的模块就可以照常使用这些变量了，而不用作任何改动。

包装模块

现在我们有了管理模块依赖和命名空间的方法，但仍然有一个问题自始至终都没有解决——每个文件都独占一个 HTTP 请求。我们依赖的所有模块都从远程加载，尽管是以异步的形式，还是会造成很严重的性能问题——将应用的启动时间推后。

对于需要异步加载的模块，我们手动将模块包装成一种转换格式，这种做法看起来也很冗余。可以在服务器端将小文件合并为一个文件输出，这种做法一举两得。这样浏览器只需发起一个 HTTP 请求来抓取一个资源文件，就能将所有的模块都载入进来，显然这种做法更高效。使用打包工具也是一种明智的做法，这样就不必随意、无组织地打包模块了，而是静态分析这些文件，然后递归地计算它们的依赖关系。打包工具同样会将不符合要求的模块包装成转换格式，这样就不必手动输入代码了。

除了在服务器端合并代码，很多模块打包工具也支持代码的压缩（minify），以进一步减少请求的体积。实际上，一些工具——比如 rack-modulr (<http://goo.gl/0YcFK>) 和 Transporter (<http://goo.gl/Exkpm>) ——已经整合进了 Web 服务器，当首次处理某个请求

◀ 79

时会自动处理模块操作。^{注9}

例如,这里是一段简单的 Rack (<http://rack.rubyforge.org>) CommonJS 模块服务示例代码,它使用 rack-modulr 来实现:

```
require "rack/modulr"

use Rack::Modulr, :source => "lib", :hosted_at => "/lib"
run Rack::Directory.new("public")
```

可以使用 rackup 命令来启动服务。所有 CommonJS 模块都包含在 lib 文件夹中,它们会根据依赖关系自动合并成一个文件并包装进一个转换回调中。我们的脚本加载器根据需要发起对模块的请求,将它们载入到页面中:

```
>> curl "http://localhost:9292/lib/application.js"
require.define("maths"....
```

如果你不熟悉 Ruby 的话,还有很多其他的选择。FlyScript (<http://www.flyscript.org/>) 是用 PHP 编写的一个 CommonJS 模块包装器,Transporter 是基于 JSGI (<http://jackjs.org/>) 服务器的转换器,Stitch (<http://github.com/sstephenson/stitch>) 则整合了 Node.js 服务器。

模块的按需加载

你可能不想用模块化的方式来写代码,或许因为你现有的代码和库改成用模块化的方法来管理需要作太多改动。幸运的是,还有很多其他的替代方案,比如 Sprockets (<http://getsprockets.org>)。Sprockets 给你的 JavaScript 代码添加了同步 require() 支持。以 `//=` 形式书写的注释指令都会被 Sprockets 作预处理。比如,`//=require` 指令通知 Sprockets 来检查类库的加载路径,加载它并以行内形式包含进来:

```
//= require <jquery>
//= require "./states"
```

在这个例子中, `jquery.js` 是 Sprockets 所需的加载地址, `states.js` 是以相对路径的形式给出的。Sprockets 非常聪明,它会保证只加载库文件一次,自动忽略之后的加载需求。相比 CommonJS 模块, Sprockets 支持缓存和压缩 (minify)。在开发过程中,你的服务器会根据页面的需要动态解析并合并文件。当站点的访问非常频繁时, JavaScript 文件

注9: 随着应用体积越来越大,考虑到可维护性。因为需要将模块很细地颗粒化,新的文件会拆分成很多个,而考虑到性能,又不得不将文件合并载入。两者是一对矛盾,所以模块依赖管理既要考虑到代码结构的整洁、清晰,又要兼顾性能最优化。文中提到的方法就是在努力寻求两者之间的最佳平衡。雅虎和淘宝都已经实现了基于 CDN 的静态文件合并 (combo) 输出功能。使用 YUI3 或 KISSY 的加载器来计算合并输出的 url,是解决这一对矛盾的一种尝试。——译者注

可以被预合并，这样就可以作为静态文件来处理，从而提高了性能。

尽管 Sprockets 是一个基于命令行的工具，但也有一些工具集成了 Rack 和 Rails，比如 rack-sprockets (<http://github.com/kelredd/rack-sprockets>)，甚至还有一些 PHP 的实现 (<http://goo.gl/0HvwT>)。Sprockets（包括所有的模块包装器）的中心思想是，所有的 JavaScript 文件都需要预处理，不管是在服务器端用程序作处理，还是使用命令行工具作处理。

LABjs

LABjs (<http://www.labjs.com>) 是最简单的模块依赖管理器^{注 10}。它不需要任何服务器支持和 CommonJS 模块支持。使用 LABjs 载入你的脚本代码，减少了页面加载过程中的资源阻塞，这是一种极其简单且极为有效的性能优化的方法。默认状况下，LABjs 会尽可能快速地以并行方式加载脚本。然而，如果代码之间有依赖关系，也可以非常简单地指定它们的执行顺序：

```
<script>
  $LAB
    .script('/js/json2.js')
    .script('/js/jquery.js').wait()
    .script('/js/jquery-ui.js')
    .script('/js/vapor.js');
</script>
```

在这个例子中，所有的脚本加载都是并行的，但 LABjs 会确保 *jquery.js* 在 *jquery-ui.js* 和 *vapor.js* 之前加载并执行。它的 API 非常简单且简洁明了，如果了解 LABjs 更多的高级特性，比如 LABjs 如何支持行内脚本，可以查看它的文档 (<http://labjs.com/documentation.php>)。

无交互行为内容的闪烁 (FUBC)

使用这些加载器来加载页面时，有一点需要尤为注意——用户可能会看到页面闪了一下，出现一部分没有交互行为的内容快速闪过 (FUBC)，比如在 JavaScript 执行之前会有一部分无样式的页面原始内容闪烁一下。如果你不依赖 JavaScript 来修改初始页面的样式，问题其实并不严重。但如果依赖 JavaScript 来操作样式，则需要将样式提取出来放入初始化 CSS 之中，比如隐藏一些元素或展示一个加载指示器，提示页面正在加载中。

注 10：作者这里将 LABjs 归类为“依赖管理器”有些勉强，LABjs 给自己的定位是“脚本加载器”，主要是为了解决加载脚本时的性能问题。——译者注

使用文件

以往，文件的访问和操作都是基于桌面应用程序，在 Web 中操作文件必须借助第三方插件技术，比如 Adobe Flash。然而随着 HTML5 的普及和推广，基于 Web 应用的文件操作不再变得遥不可及，HTML5 赋予开发者更多权限来处理文件，这进一步打破了桌面应用和 Web 应用之间的界限。在现代浏览器中，用户可以直接将文件拖拽进页面中，粘贴格式化数据，或是实时查看上传文件的进度。

浏览器支持

不是所有的浏览器都支持新的 HTML5 文件 API，但现在已经有足够多的浏览器都实现了这一特性，你完全可以将文件操作的功能添加至你的应用中：

- Firefox ≥ 3.6
- Safari ≥ 6.0
- Chrome ≥ 7.0
- IE: no support
- Opera ≥ 11.1

由于 IE 还不支持这一特性，你需要采用渐进增强的理念来处理文件操作。给用户提供传统的文件上传功能，同时允许在支持文件操作的浏览器中使用文件拖拽功能。特性检测也很简单，只需检查相关的对象是否存在即可：

```
if (window.File && window.FileReader && window.FileList) {  
    // 支持文件操作 API  
}
```

获取文件信息

HTML5 的文件操作有一定的安全限制，主要的限制是只有被用户选中的文件才能被访问。将文件拖拽至浏览器中、选择要输入的文件或将文件粘贴至 Web 应用中，这些操作当然可以满足这一安全限制。尽管已经有人实现了“基于 JavaScript 的文件系统”(<http://goo.gl/CbDnt>)，但这里的访问都是基于沙箱的。显然，允许 JavaScript 任意无限制的操作文件会带来很严重的安全风险。

在 HTML5 中使用 File 对象来表示文件，它有三个属性：

name

文件名，这是一个只读字符串

size

文件大小，这是一个只读的整数

type

文件的 MIME 类型，是一个只读字符串，如果类型没有指定就是空字符串

出于安全的原因，文件的路径是无法得到的。

可以通过 FileList 对象来获取多个文件，可以把它当作 File 对象组成的数组来看待。

文件输入

从 Web 诞生之日起就包含了文件输入的功能，用户可以使用这个传统的功能来上传文件。HTML5 增强了这一功能，修正了很多缺陷。对于开发者来说，长期以来很让人头疼的是多文件上传。过去开发者只能堆积很多文件输入框或者依赖插件（如 Flash）来实现多文件上传。HTML5 提供了 multiple 属性来支持多文件。给文件输入框指定 multiple 属性，告知浏览器可以使用它来选择多个文件。旧版本的浏览器如果不支持 HTML5 则会忽略这个属性：

```
<input type="file" multiple>
```

但 UI 并不完美，用户需要按住 Shift 键来多选文件。有时你需要为这个效果增加一些提示，比如，Facebook 发现 85% 的用户上传照片的时候，每次只上传一张 (<http://goo.gl/8yWuy>)。通过在上传过程中给出如何上传多个照片的提示，如图 7-1 所示，这个比例从 85% 降到了 40%。



图7-1: Fackbook的多文件上传

对于开发者来说还有一个棘手的问题，他们不知道当前选中了哪些文件。往往我们需要添加一些诸如验证选中文件合法性的功能，确保所选文件的类型和大小都是正确的。HTML5 提供了接口可以获得选中文件的信息，这用到了 `files` 属性。

`files` 属性是只读的，它返回一个 `FileList`，可以用它对文件作遍历，逐一对它们作验证，并将验证结果通知给用户。

```
var input = $("input[type=file]");

input.change(function(){
    var files = this.files;

    for (var i=0; i < files.length; i++)
        assert( files[i].type.match(/image.*/) )
});
```

83

访问选中的文件不会有任何限制，比如你可以读取文件内容，以及显示上传预览。此外你还可以使用 Ajax 在后台执行上传，这样就不会像传统上传文件那样阻塞 UI。在后台上传文件还可以实时显示进度条。在接下来的章节中会详细讨论这些内容。

拖拽

早在 1999 年，微软的 IE5 就“设计”并实现了最原始的拖拽，自那时起后续的 IE 版本都支持拖拽。HTML5 规范刚刚增加了拖拽的内容，现在 Safari、Firefox 和 Chrome 也都模仿 IE 的实现提供了拖拽支持。然而，坦白地讲，HTML5 的规范并不明晰 (http://www.quirksmode.org/blog/archives/2009/09/the_html5_drag.html)，需要重新整理。

关于拖拽的事件至少有七个：*dragstart*、*drag*、*dragover*、*dragenter*、*dragleave*、*drop* 和 *dragend*。接下来会对每个事件作详细讲解。

即使你的浏览器不支持 HTML5 的文件 API，但你仍可继续使用拖拽 API。当前的浏览器支持情况如下：

84

- Firefox >= 3.5
- Safari >= 3.2
- Chrome >= 7.0
- IE >= 6.0
- Opera: no support

拖拽

拖拽^{注1}的实现非常简单。可以将元素的 *draggable* 属性设置为 *true* 来启用元素的拖拽。

```
<div id="dragme" draggable="true">Drag me!</div>
```

现在给可拖拽的元素关联一些数据。我们可以监听 *dragstart* 事件并调用事件的 *setData()* 函数：

```
var element = $("#dragme");

element.bind("dragstart", function(event){
    // 我们不想使用 jQuery 的抽象方法
    event = event.originalEvent;

    event.dataTransfer.effectAllowed = "move";
    event.dataTransfer.setData("text/plain", $(this).text());
    event.dataTransfer.setData("text/html", $(this).html());
    event.dataTransfer.setDragImage("/images/drag.png", -10, -10);
});
```

jQuery 提供了一个抽象的事件^{注2}，它不包含我们需要的 *dataTransfer* 对象。但为了方便起见，这个抽象对象提供了 *originalEvent* 属性，我们可以通过它来访问拖拽（*drag/drop*）API。

刚才提到，事件包含 *dataTransfer* 对象，其中包含拖拽和释放拖拽所需的方法。通过 *setData()* 函数可以设置“互联网媒体类型”（*MIMEType*）和一个字符串数据。在这个

注1： Drag 和 Drop 在不引起歧义时统一译为“拖拽”，其实这个词组是由 Drag（拖拽）和 Drop（释放拖拽）组成，本小节标题是 Dragging，特指区别于“释放拖拽”的“拖拽”。——译者注

注2： 为了处理底层 API 的差异性，通常将底层对象作封装，对外提供统一的接口，这是一种常见的编程模式，被称为“门面模式”（*facade*），jQuery 和 YUI3 的事件都是包装后的“门面”，作者此处提到的抽象事件就是指这个“门面”。——译者注

例子中，我们给 *drag* 事件设置了 *text* 和 *text/html* 数据。当元素释放拖拽，就会触发 *drop* 事件，这时可以读取这个数据。与之类似，如果元素拖拽到浏览器的外部，其他的应用也可以根据它们支持的文件类型来处理释放拖拽的数据。

当拖拽文本时应使用 *text/plain* 类型。推荐将应用的反馈类型总是设置为此类型，包括应用的默认类型及释放拖拽的目标不支持其他格式时。拖拽的链接包含两种格式：*text/plain* 和 *text/uri-list*。通过将每个链接合并为一个新行来拖拽多个链接：

```
// 拖拽链接
event.dataTransfer.setData("text/uri-list", "http://example.com");
event.dataTransfer.setData("text/plain", "http://example.com");

// 拖拽多个链接
event.dataTransfer.setData("text/uri-list", "http://example.com\nhttp://google.com");
event.dataTransfer.setData("text/plain", "http://example.com\nhttp://google.com");
```

85

setDragImage() 函数是可选的，用它可以设置拖拽操作过程中跟随鼠标移动的图片，它的参数是图片源地址和 *x/y* 坐标，这个坐标是相对于鼠标位置的。如果没有提供，则会将拖拽的元素拷贝一份并显示为半透明。除了 *setDragImage()* 之外还可以使用 *addElement(element,x,y)*，它使用给定的元素来更新被拖拽的元素。换句话说，你可以为拖拽操作的过程自定义要显示的元素。

同样，可以让用户拖拽浏览器之外的文件，只需设置 *DownloadURL* 类型即可。你可以将 URL 指定为文件路径，浏览器随后会将它下载下来。Gmail 实现了类似功能，特效非常出众，可以让用户将邮件附件从浏览器中拖拽至桌面。

坏消息是，只有 Chrome 支持这个特性，且这个特性还在修订之中，但不影响使用。在不久的将来其他的浏览器也会陆续加入对这个特性的支持。*DownloadURL* 值的格式是由冒号 (:) 分隔的文件列表信息：媒体类型 (MIME)、名称和地址。

```
$("#preview").bind("dragstart", function(e){
    e.originalEvent.dataTransfer.setData("DownloadURL", [
        "application/octet-stream",    // MIME 类型
        "File.exe",                    // 文件名
        "http://example.com/file.png"  // 文件地址
    ].join(":"));
});
```

在本书的附加文档资源中可以查看到完整的 HTML5 拖拽 API 的例子：*assets/ch07/drag.html*。

释放拖拽

拖拽 API 允许你监听 *drop* 事件，它可以对释放拖拽的文件和其他元素做出响应。现在就可以开始领略拖拽 API 的美妙之处了。要想只触发 *drop* 事件，你需要撤销两个默认事件 *dragover* 和 *dragenter*。例如，用下面的方式来撤销这两个事件：

```
var element = $("#dropzone");

element.bind("dragenter", function(e){
    // 撤销事件
    e.stopPropagation();
    e.preventDefault();
});

element.bind("dragover", function(e){
    // 设置鼠标
    e.originalEvent.dataTransfer.dropEffect = "copy";

    // 撤销事件
    e.stopPropagation();
    e.preventDefault();
});
```

86

你可以在 *dragover* 事件中通过设置 *dropEffect* 来设置鼠标样式，正如上面代码所示。可以通过监听 *dragenter* 和 *dragleave* 事件，以及添加 / 删除目标元素的 class 来增强拖拽功能的交互性，当用户拖拽的文件进入目标区域后，可以在释放拖拽时给他一个提示。

只有当我们撤销了 *dragenter* 和 *dragover* 事件，才能开始监听 *drop* 事件。当拖拽的元素或者文件在目标区域释放时才会触发 *drop* 事件。*drop* 事件的 *dataTransfer* 对象有一个 *files* 属性，它返回拖拽的所有文件的 *FileList*：

```
element.bind("drop", function(event){
    // 撤销事件
    event.stopPropagation();
    event.preventDefault();

    event = event.originalEvent;

    // 访问拖拽的文件
    var files = event.dataTransfer.files;

    for (var i=0; i < files.length; i++)
        alert("Dropped " + files[i].name);
});
```

可以使用 *dataTransfer.getData()* 函数来获取文件的数据，将支持的格式作为参数传入。如果那个格式不可用，函数会返回 *undefined*。

```
var text = event.dataTransfer.getData("Text");
```

`dataTransfer` 对象拥有一个只读的 `types` 属性，它返回一个包含了由设置在 *dragstart* 事件上的媒体类型格式组成的 `DOMStringList`（本质上是数组），此外，如果拖拽了其他的文件，其中一个类型是字符串“Files”。

```
var dt = event.dataTransfer
for (var i=0; i < dt.types.length; i++)
    console.log( dt.types[i], dt.getData(dt.types[i]) );
```

可以在这里看到完整的释放拖拽的示例代码：[assets/ch07/drop.html](#)。

撤销默认的 Drag/Drop

默认情况下，将一个文件拖拽到 web 页面中会让浏览器重定向到这个文件。我们当然不希望应用的 URL 地址发生改变，因此需要阻止将文件拖拽进应用的非目标位置时的页面跳转。这很容易实现，只需撤销 `body` 的 *dragover* 事件即可。

```
$("#body").bind("dragover", function(e){
    e.stopPropagation();
    e.preventDefault();
    return false;
});
```

87

复制和粘贴

除了将拖拽功能和桌面进行整合，有些浏览器还支持复制和粘贴。API 还没有被标准化，而且没有纳入 HTML5 规范，因此你需要针对不同版本的浏览器决定是否实现复制和粘贴的功能。

有意思的是，IE 再一次成为了先行者，从 IE5.0 时代就开始支持复制和粘贴了。WebKit 仿效了微软的 API 并对它作了一定的增强，而且和拖拽的 API 作了整合。两者几乎一模一样，只是使用了不同的对象：复制粘贴使用了 `clipboardData`，拖拽使用了 `dataTransfer`。

Firefox 不支持复制和粘贴，至少目前不支持，尽管它有一个专用的 API 可以访问剪切板。WebKit (Safari/Chrome) 对复制和粘贴支持良好，我认为 W3C 最终会将剪切板 API 纳入标准规范。浏览器的支持情况如下：

- Safari >= 6.0
- Chrome (only pasting)
- Firefox: no support
- IE >= 5.0 (different API)

复制

和复制相关的事件有两个，和剪切相关的事件也有两个：

- *beforecopy*
- *copy*
- *beforecut*
- *cut*

从事件的名称就可以看到，*beforecopy* 和 *beforecut* 是在剪切板操作之前触发的，这样就可以根据需求选择是否撤销复制和剪切的操作。当用户复制了一些选中的文本，这时触发了 *copy* 事件，使用 `clipboardData` 对象就可以设置自定义的剪切板数据。和 `dataTransfer` 对象类似，`clipboardData` 包含一个 `setData()` 方法，参数是媒体格式和字符串值。如果想调用这个函数，你需要撤销原始的 *copy* 事件，阻止默认的浏览器行为（复制操作）。

IE 将 `clipboardData` 对象设置在 `window` 对象上，而不是事件上。你需要检查事件中是否存在这个对象，如果不存在则再检查一下是否在 `window` 对象中。

Firefox 实际上可以触发 *copy* 事件，但它不允许你访问 `clipboardData` 对象。Chrome 允许你访问这个对象，但它会忽略你在这个对象上设置的任何数据。

```
88 > $("textarea").bind("copy", function(event){
    event.stopPropagation();
    event.preventDefault();

    var cd = event.originalEvent.clipboardData;

    // IE
    if ( !cd ) cd = window.clipboardData;

    // Firefox
    if ( !cd ) return;

    cd.setData("text/plain", $(this).text());
});
```

根据目前浏览器更新换代的速度，复制 / 粘贴的 API 会很快被标准化。如果你想为应用增加复制 / 粘贴支持，需要考虑一下实际情况再作决定。

粘贴

和粘贴相关的事件有两个，*beforepaste* 和 *paste*。当用户开始粘贴操作但数据还未被粘贴时触发 *paste* 事件。同样，不同的浏览器的实现也不一样。Chrome 里只有当没有任何元素被选中时才会触发这个事件。IE 和 Safari 都需要有被选中的元素存在。

粘贴的 API 和 *drop* 事件的很像。事件中包含 `clipboardData` 属性，可以通过这个对象的 `getData()` 方法来获取要粘贴的数据，当然也需要给定媒体格式。不幸的是，根据我的测试结果来看，`types` 属性总是 `null`，所以无法看到剪切板中的数据支持哪种类型。除非你撤销了这个事件，否则粘贴的操作会照常执行，数据会被粘贴至获得焦点的元素：

```
$("#textarea").bind("paste", function(event){
    event.stopPropagation();
    event.preventDefault();

    event = event.originalEvent;

    var cd = event.clipboardData;

    // IE
    if ( !cd ) cd = window.clipboardData;

    // Firefox
    if ( !cd ) return;

    $("#result").text(cd.getData("text/plain"));

    // Safari 中的事件支持文件的粘贴
    var files = cd.files;
});
```

WebKit 的“午夜版”(<http://nightly.webkit.org/>) 允许你访问 `clipboardData` 的 `files` 属性，可以在应用中添加文件粘贴功能。如果这个功能被整理进标准规范中，我希望其他浏览器厂商也及时地实现这个规范。

89

那么，是否有必要考虑浏览器兼容性情况呢？没错，实际上没有其他更好的变通方法可以实现复制 / 粘贴。比如 Cappuccino (<http://cappuccino.org/>) 选择绕过 *oncopy* 事件族，通过监听键盘事件来实现。当监听到组合键 `Ctrl+v` 被按下时，它会让一个隐藏的输入框获得焦点，这个输入框可以得到粘贴的数据。它兼容每个浏览器，但显然只支持键盘事件，通过菜单执行的粘贴操作就无法监听到了。

读文件

当你获得 `File` 的引用时,可以用它来实例化一个 `FileReader` 对象,将文件内容读入内存。文件的读取是异步的,你需要给 `FileReader` 实例提供一个回调函数,当读文件时触发这个回调。

`FileReader` 中包含四个方法来读取文件数据,可根据所需的文件格式来选择合适的方法。

`readAsBinaryString(Blob|File)` :

以二进制字符串形式返回这个文件 / 二进制大对象 (`Blob`) 的数据。每个字节使用 0 到 255 之间的整数来表示。

`readAsDataURL(Blob|File)` :

以 URL (http://en.wikipedia.org/wiki/Data_URI_scheme) 编码的形式返回这个文件 / 二进制大对象的数据,比如可以把它用作图片的 `src` 属性的值。

`readAsText(Blob|File, encoding='UTF-8')` :

以字符串形式返回文件 / 二进制大对象的数据,默认情况下,字符串以 UTF-8 格式编码。

`readAsArrayBuffer(Blob|File)` :

以 `ArrayBuffer` 对象的形式返回文件 / 二进制大对象的数据,多数浏览器都未实现这个方法。

`FileReader` 实例包含很多事件,当其中一个读文件的函数被调用时就触发这些事件。你需要关注的事件包括 :

onerror :

当发生错误时调用

onprogress :

当数据在读取过程中周期性地调用

onload :

当数据可用时调用

90 要想使用 `FileReader`,你需要生成一个实例,添加事件监听,使用其中一个读文件的方法。

`onload` 事件包含 `result` 属性,以正确的格式表示读取的数据 :

```
var reader = new FileReader();
reader.onload = function(e) {
    var data = e.target.result;
```

```
};
reader.readAsDataURL(file);
```

比如，我们可以将上例中的 `data` 属性用作图片源，显示某张图片文件的缩略图：

```
var preview = $("img#preview")

// 检查文件是否是图片类型
// 再确保它的体积不要太大，否则会造成浏览器的问题
if (file.type.match(/image.*/) &&
    file.size < 50000000) {

    var reader = new FileReader();
    reader.onload = function(e) {
        var data = e.target.result;
        preview.attr("src", data);
    };
    reader.readAsDataURL(file);
}
```

二进制大文件和文件切割

有时最好将文件的一个片段读入内存，而不是整个文件。HTML5 中的文件 API 提供了一个非常方便的 `slice()` 函数。其第一个参数是读文件的起始字节位置，第二个参数是以字节为单位的偏移量(或片段长度)。它返回 **Blob** 对象，我们可以使用支持 **File** 对象(比如 **FileReader**) 的方法对它做操作。比如，可以用这种方法将文件读入缓存里：

```
var bufferSize = 1024;
var pos = 0;

var onload = function(e){
    console.log("Read: ", e.target.result);
};

var onerror = function(e){
    console.log("Error!", e);
};

while (pos < file.size) {
    var blob = file.slice(pos, bufferSize);

    var reader = new FileReader();
    reader.onload = onload;
    reader.onerror = onerror;
    reader.readAsText(blob);
    pos += bufferSize;
}
```

91

正如你所看到的，可以使用 **FileReader** 实例一次，之后你需要生成一个新的实例。

你可以在这里看到完整的示例代码：`assets/ch07/slices.html`。有一点需要特别注意，如果沙箱^{注3}是本地的，那么就无法读文件，换句话说，如果直接从硬盘读取 `slices.html`，而不是通过域名来访问它，读文件操作会失败并触发一个 `onerror` 事件。

自定义浏览器按钮

在开发项目的过程中，经常需要用程序打开文件预览窗口。换句话说，我们往往希望点击一个带自定义样式的“浏览”或“附件”按钮来即刻打开浏览文件对话框，这样就不必忍受浏览器自带的（丑陋无比的）传统样式按钮了。但从安全角度来看，这更像是一种“奇技淫巧”。文件输入框并不提供打开浏览文件对话框的函数，而且 Firefox 的实现更诡异，当点击文件输入框时甚至无法触发自定义的 `click` 事件。

当前的解决办法听起来像是一种 hack 技术，但的确很好用。当将鼠标移动到按钮上方时，在相同的位置放置一个透明的文件输入框，尺寸和按钮一样。透明的文件输入框可以获得任何点击事件，并打开一个浏览文件的对话框。

在本书附加文件的 `assets/ch07` 文件夹里，你可以找到 `jquery.browse.js`，这个文件是基于 jQuery 的一个插件来实现这种功能的。调用 jQuery 实例的 `browseElement()` 函数来创建一个自定义的浏览按钮。这个函数将返回一个文件输入框，可以给它绑定 `change` 事件监听，检测用户何时选中了一些文件。

```
var input = $("#attach").browseElement();

input.change(function(){
    var files = $(this).attr("files");
});
```

这个插件兼容各个浏览器，使用起来非常方便！

上传文件

XMLHttpRequest Level 2 规范（<http://www.w3.org/TR/XMLHttpRequest2/>）赋予了 Ajax 上传文件的能力。在此之前，文件上传的用户体验简直糟糕透了。一旦选择上传一个文件则页面就会重新加载，用户必须等待文件上传完毕，没有任何上传进度提示让用户体验很不好，更谈不上易用性了。幸运的是，XHR 2 解决了这个问题。它允许我们在后台上传文件，甚至提供了上传事件，这样就可以实时地监控文件的上传进度了。主流浏览器都支持这种做法：

注3： 一个 html 文件的打开方式有很多种，可以直接将这个文件拖入浏览器来访问，也可以放入虚拟主机中通过 http 协议来访问，这里所说的“沙箱”特指打开这个文件的方式。——译者注

- Safari >= 5.0
- Firefox >= 4.0
- Chrome >= 7.0
- IE: 不支持
- Opera: 不支持

可以使用现有的 XMLHttpRequest API 来完成文件上传，使用 `send()` 函数即可，或者使用 `FormData` 实例。`FormData` 实例用一种非常简单的接口表示表单的内容。可以直接通过抓取一个表单来创建 `FormData`，或者在实例化对象时传入已经存在的 `form` 元素：

```
var formData = new FormData($("#form")[0]);

// 可以添加字符串
formData.append("stringKey", "stringData");

// 甚至可以添加文件对象
formData.append("fileKey", file);
```

当 `formData` 中的数据准备完成，就可以使用 XMLHttpRequest 将数据 POST 到服务器了。如果你在使用 jQuery 来处理 Ajax 请求，则需要将 `processData` 选项置为 `false`，这样 jQuery 就不会尝试去对数据作序列化处理了。不要设置 `Content-Type` 头，因为浏览器会自动将请求头设置为 `multipart/form-data`，同时设置的还有多个字段的边界：

```
jQuery.ajax({
  data: formData,
  processData: false,
  url: "http://example.com",
  type: "POST"
})
```

除了 `FormData` 之外还有另一种方法，将文件直接传给 XHR 对象的 `send()` 函数：

```
var req = new XMLHttpRequest();
req.open("POST", "http://example.com", true);
req.send(file);
```

此外，使用 jQuery 的 Ajax API 可以这样实现：

```
$.ajax({
  url: "http://example.com",
  type: "POST",
  success: function(){ /* ... */ },
  processData: false,
  data: file
});
```

这种方式和传统的 `multipart/form-data` 方式有很小的不同，可以忽略。通常我们会将文件信息（比如文件名）也上传上去。但这个例子中并不是这样，这里上传的只是纯文件内容。要想将文件的信息也一并传入，可以设置自定义的头，比如 `X-File-Name`。服务器可以读取这些头字段，并正确处理这些信息：

```
$.ajax({
  url: "http://example.com",
  type: "POST",
  success: function(){ /* ... */ },
  processData: false,
  contentType: "multipart/form-data",

  beforeSend: function(xhr, settings){
    xhr.setRequestHeader("Cache-Control", "no-cache");
    xhr.setRequestHeader("X-File-Name", file.fileName);
    xhr.setRequestHeader("X-File-Size", file.fileSize);
  },

  data: file
});
```

不幸的是，很多服务器处理上传文件时都遇到不小麻烦。相比于 `multipart` 或从参数中读取 URL 编码的数据来说，纯文件数据更难处理。使用这种方法就需要你自己对请求作解析。正是因为这个原因，我推荐使用 `FormData` 对象，以 `multipart/form-data` 格式发送序列化后的上传数据。在 `assets/ch07` 文件夹中可以找到 `jquery.upload.js`，它是一个 jQuery 插件，它将文件上传的功能封装成了一个简单的接口：`$.upload(url, file)`。

Ajax 进度条

XHR 2 规范加入了对 `progress` 事件的支持，下载和上传都支持这个事件。使用它就可以实现实时的文件上传进度条，可以让用户时刻跟踪上传进度。

想要监听下载请求的 `progress` 事件，只需直接给 XHR 实例绑定事件即可：

```
var req = new XMLHttpRequest();

req.addEventListener("progress", updateProgress, false);
req.addEventListener("load", transferComplete, false);
req.open();
```

对于上传 `progress` 事件来说，则需要给 XHR 实例的 `upload` 属性添加监听：

```
var req = new XMLHttpRequest();

req.upload.addEventListener("progress", updateProgress, false);
```

```
req.upload.addEventListener("load", transferComplete, false);
req.open();
```

load 事件只有在请求完成时才会触发，但是会在服务器终结响应之前触发。我们可以将它添加至 jQuery，因为 XHR 对象和设置都被传给了 *beforeSend* 回调。来看一个完整的例子，包括自定义的头：

94

```
$.ajax({
  url: "http://example.com",
  type: "POST",
  success: function(){ /* ... */ },
  processData: false,
  dataType: "multipart/form-data",

  beforeSend: function(xhr, settings){
    var upload = xhr.upload;

    if (settings.progress)
      upload.addEventListener("progress", settings.progress, false);

    if (settings.load)
      upload.addEventListener("load", settings.load, false);

    var fd = new FormData;

    for (var key in settings.data)
      fd.append(key, settings.data[key]);

    settings.data = fd;
  },

  data: file
});
```

progress 事件包含上传的 *position*（已经上传的字节数）和 *total*（上传的总字节数）。你可以使用这两个属性来计算进度的百分比：

```
var progress = function(event){
  var percentage = Math.round((event.position / event.total) * 100);
  // 设置进度条
}
```

实际上这个事件还包含一个时间戳，因此如果你记录了开始上传的时间，可以计算出离上传完成还需多长时间（ETA）：

```
var startStamp = new Date();
var progress = function(e){
  var lapsed = startStamp - e.timeStamp;
```

```
    var eta = lapsed * e.total / e.position - lapsed;
  };
```

如果上传文件体积很小时，这个估计完成时间会非常不准（当然也很快）。在我看来，如果文件体积很大，上传时间超过大约四分钟，则最好给出 ETA。多数情况下只需进度条就够了，它足以清晰地告知用户大概还需要多久能上传完毕。

95 jQuery 拖拽上传

现在，让我们把学到的东西应用到实践中，来实现一个可拖拽上传文件的功能。我们需要几个库：*jquery.js* 用来作底层库，*jquery.ui.js* 用来构建进度条，*jquery.drop.js* 用来提供抽象的拖拽 API，以及 *jquery.upload.js* 用来作 Ajax 上传。我们所有的逻辑都将放在 `jQuery.ready()` 中，因此程序会在 DOM 树构建完成后运行：

```
//= require <jquery>
//= require <jquery.ui>
//= require <jquery.drop>
//= require <jquery.upload>

jQuery.ready(function($){
  /* ... */
});
```

创建拖拽目标区域

我们想把文件拖拽到 `#drop` 元素上，首先要把它转换为释放拖拽的区域。这就需要绑定 `drop` 事件，撤销事件并遍历释放拖拽的文件列表，然后将它们传入 `uploadFile()` 函数：

```
var view = $("#drop");
view.dropArea();

view.bind("drop", function(e){
  e.stopPropagation();
  e.preventDefault();

  var files = e.originalEvent.dataTransfer.files;
  for ( var i = 0; i < files.length; i++)
    uploadFile(files[i]);

  return false;
});
```

上传文件

现在来看 `uploadFile()` 函数——“让我们见证奇迹发生的时刻！” 我们使用 *jquery.upload.js* 中的 `$.upload()` 函数来发送 Ajax 上传请求到服务器。然后监听请求的上传进度事件并更新 jQuery UI 进度条。当上传完成时，我们即刻通知用户上传已经完成，并将元素删除。

```
var uploadFile = function(file){
    var element = $("

◀ 96



怎么样，简单吧！可以在这里查看到完整的示例代码：assets/ch07/dragdro-pupload.html



---



jQuery拖拽上传 | 107


```


实时Web

为什么实时 Web 这么重要？我们生活在一个实时（real-time）的世界中，因此 Web 的最终最自然的状态也应当是实时的。用户需要实时的沟通、数据和搜索。我们对互联网信息实时性的要求也越来越高，如果信息或消息延时几分钟后才更新，简直让人无法忍受。现在很多大公司（如 Google、Facebook 和 Twitter）已经开始关注实时 Web，并提供了实时性服务。实时 Web 将是未来最热门的话题之一。

实时 Web 的发展历史

传统的 Web 是基于 HTTP 的请求 / 响应模型的^{注 1}：客户端请求一个新页面，服务器将内容发送到客户端，客户端再请求另外一个页面时又要重新发送请求。后来有人提出了 Ajax，Ajax 使得页面的体验更加“动态”，可以在后台发起到服务器的请求。但是，如果服务器有更多数据需要推送到客户端，在页面加载完成后是无法实现直接将数据从服务器发送给客户端的。实时数据无法被“推送”给客户端。

为了解决这个问题，有人提出了很多解决方案。最简单（暴力）的方案是用轮询：每隔一段时间都会向服务器请求新数据。这让用户感觉应用是实时的。实际上这会造成延时和性能问题，因为服务器每秒中都要处理大量的链接请求，每次请求都会有 TCP 三次握手并附带 HTTP 的头信息。尽管现在很多应用仍在使用轮询，但这并不是最理想的解决方案。

后来随着 Comet 技术的提出，又出现了很多更高级的解决方案。这些技术方案包括永久

注 1： HTTP 协议是 Web 的基石，HTTP 都是短连接，客户端向服务器发送请求，服务器需要做出响应，请求加响应就构成一次完整的 HTTP 连接的过程，响应完成后连接就“断掉”了，所以对于服务器来说，信息推送到客户端都是“被动的”，理论上任何信息从服务器发送到客户端都必须由客户端先发起请求，这就是文中所说的请求 / 响应模型。——译者注

帧 (forever frame)、XHR 流 (xhr-multipart)、htmlfile, 以及长轮询。长轮询是指, 客户端发起一个到服务器的 XHR 连接, 这个连接永不关闭, 对客户端来说连接始终是挂起状态。当服务器有新数据时, 就会及时地将响应发送给客户端, 接着再将连接关闭。然后重复整个过程, 通过这种方式就实现了“服务器推”(server push)。

98 Comet 技术是非标准的 hack 技术, 正因为此, 浏览器端的兼容性就成了问题。首先, 性能问题无法解决, 向服务器发起的每个连接都带有完整的 HTTP 头信息, 如果你的应用需要很低的延时, 这将是一个棘手的问题。当然不是说 Comet 本身有问题, 因为还没有其他替代方案前 Comet 是我们的惟一选择。

浏览器插件 (如 Flash) 和 Java 同样被用于实现服务器推。它们可以基于 TCP 直接和服务器建立 socket 连接, 这种连接非常适合将实时数据推给客户端。问题是并不是所有的浏览器都安装了这些插件, 而且它们常常被防火墙拦截, 特别是在公司网络中。

现在 HTML5 规范为我们准备了一个替代方案。但这个规范稍微有些超前, 很多浏览器都还不支持, 特别是 IE, 对于现在很多开发者来说帮助不大, 鉴于大部分浏览器还未实现 HTML5 的 WebSocket, 现行最好的办法仍然是使用 Comet。

WebSocket

WebSocket (<http://dev.w3.org/html5/websockets>) 是 HTML5 规范 (<http://www.w3.org/TR/html5>) 的一部分, 提供了基于 TCP 的双向的、全双工^{注2}的 socket 连接。这意味着服务器可以直接将数据推送给客户端, 而不需要开发者求助于长轮询或插件来实现, 这是一个很大的进步。尽管有一些浏览器实现了 WebSocket, 但由于一些安全问题没有解决, 因此协议 (<http://goo.gl/F7lvW>) 仍然在修订之中。然而这不会阻碍我们的脚步, 这些安全问题属于技术性问题, 会很快被修复, WebSocket 很快就会成为最终规范。与此同时, 对于那些不支持 WebSocket 的浏览器, 可以降级使用笨方法来实现, 比如 Comet 或轮询。

和之前的服务器推的技术相比, WebSocket 有着巨大的优势, 因为 WebSocket 是全双工的, 而不是基于 HTTP 的, 一旦建立链接就不会断掉。Comet 所面对的现实问题就是 HTTP 的体积太大, 每个请求都带有完整的 HTTP 头信息。而且包含很多没有用的 TCP 握手, 因为 HTTP 是比 TCP 更高层次的网络协议。

使用 WebSocket 时, 一旦服务器和客户端之间完成握手, 信息就可以畅通无阻地随意往来于两端, 而不用附加那些无用的 HTTP 头信息。这极大地降低了带宽的占用, 提高了性能。因为连接一直处于活动状态, 服务器一旦有新数据要更新时就可以立即发送给客

注 2: 全双工是指在发送数据的同时也能够接收数据, 两者同步进行。这就像我们平时打电话一样, 说话的同时也能够听到对方的声音。全双工的好处在于延迟小, 速度快。——译者注

户端（不需要客户端先请求，服务器再响应了）。另外，连接是双工的，因此客户端同样可以发送数据给服务器，当然也不需要附带多余的 HTTP 头。

下面这段话是出自 Google 的 Ian Hickson，HTML5 规范小组负责人，它是这样描述 WebSocket 的：

将千字节的数据降为 2 字节……并将延时从 150 毫秒降为 50 毫秒，这种优化跨越了不止一个量级，实际上仅这两点优化就足以让 Google 确信 WebSocket 会给产品带来非一般的用户体验。

现在来看一下都有哪些浏览器支持 WebSocket：

99

- Chrome ≥ 4
- Safari ≥ 5
- iOS ≥ 4.2
- Firefox $\geq 4^*$
- Opera $\geq 11^*$

尽管 Firefox 和 Opera 也都实现了 WebSocket，但考虑到 WebSocket 仍然存在安全隐患，默认并没有启用它。但这不是什么大问题，或许本书出版时 WebSocket 的安全问题就已经解决了。同时你也可以在那些对 WebSocket 支持不好的浏览器中作降级处理，使用诸如 Comet 和 Flash 的笨方法。IE 至今还未支持 WebSocket，可能在 IE9 之前的版本中都不会支持。

检测浏览器是否支持 WebSocket 也非常简单、直接：

```
var supported = ("WebSocket" in window);
if (supported) alert("WebSockets are supported");
```

长远来看，浏览器的 WebSocket API 非常清晰且合乎逻辑。可以使用 WebSocket 类来实例化一个新的套接字（socket），这需要传入服务器的端地址，在这个例子中是 ws://example.com：

```
var socket = new WebSocket("ws://example.com");
```

然后我们需要给这个套接字添加事件监听：

```
// 建立连接
socket.onopen = function(){ /* ... */ }

// 通过连接发送了一些新数据
socket.onmessage = function(data){ /* ... */ }
```

```
// 关闭连接
socket.onclose = function(){ /* ... */ }
```

当服务器发送一些数据时，就会触发 *onmessage* 事件，同样，客户端也可以调用 *send()* 函数将数据传回服务器。很明显，我们应当在连接建立且触发了 *onopen* 事件之后调用它：

```
socket.onmessage = function(msg){
    console.log("New data - ", msg);
};

socket.onopen = function(){
    socket.send("Why, hello there").
};
```

发送和接收的消息只支持字符串格式。但在字符串和 JSON 数据之间可以很轻松地相互转换，这样就可以创建你自己的协议：

100

```
var rpc = {
    test: function(arg1, arg2) { /* ... */ }
};

socket.onmessage = function(data){
    // 解析 JSON
    var msg = JSON.parse(data);

    // 调用 RPC 函数
    rpc[msg.method].apply(rpc, msg.args);
};
```

这段代码中，我们创建了一个远程过程调用（remote procedure call，简称 RPC）脚本，服务器可以发送一些简单的 JSON 来调用客户端的函数，就像下面这行代码：

```
{"method": "test", "args": [1, 2]}
```

注意，这里的调用是限制在 *rpc* 对象里的。这样做的原因主要是出于安全考虑，如果允许在客户端执行任意 JavaScript 代码，黑客就会利用这个漏洞。可以调用 *close()* 函数来关闭这个连接：

```
var socket = new WebSocket("ws://localhost:8000/server");
```

你肯定注意到了我们在实例化一个 *WebSocket* 的时候使用了 *WebSocket* 特有的协议前缀 *ws://*，而不是 *http://*。*WebSocket* 同样支持加密的连接，这需要使用以 *wss://* 为协议前缀的 TLS^{注3}。默认情况下 *WebSocket* 使用 80 端口建立非加密的连接，使用 443 端口建立加密的连接。你可以通过给 URL 带上自定义端口来覆盖默认配置。要记住，并不是所有的端口都可以被客户端使用，一些非常规的端口很容易被防火墙拦截。

注3： TLS 是“传输层的安全加密”，更多信息请查看 <http://zh.wikipedia.org/zh/传输层安全>。——译者注

说到现在，你或许会想，“我还不能在项目中使用 WebSocket，因为标准还未成型，而且 IE 不支持 WebSocket”。这样的想法并没有错，幸运的是，我们有解决方案。Websocket-js (<https://github.com/gimite/web-socket-js>) 是一个基于 Adobe Flash 实现的 WebSocket。用这个库就可以在不支持 WebSocket 的浏览器中做优雅降级。毕竟几乎所有的浏览器都安装了 Flash 插件。基于 Flash 实现的 Socket API 和 HTML5 标准规范完全一样，因此当 WebSocket 的浏览器兼容性更好的时候，只需简单地将库移除即可，而不必对代码做任何修改。

尽管客户端的 API 非常简洁、直接，但在服务器端情况就不同了。WebSocket 协议包含两个互不兼容的草案协议：草案 75 (<http://goo.gl/cgSjp>) 和草案 76 (<http://goo.gl/2u78y>)。服务器需要通过检测客户端使用的连接握手类型来判断使用哪个草案协议。

WebSocket 首先向服务器发起一个 HTTP “升级” (upgrade) 请求。如果你的服务器支持 WebSocket，则会执行 WebSocket 握手并初始化一个连接。“升级”请求中包含了原始域(请求所发出的域名)的信息。客户端可以和任意域名建立 WebSocket 连接，只有服务器才会决定哪些客户端可以和它建立连接，常用做法是将允许连接的域名做成白名单。

在 WebSocket 的设计之初，设计者们希望只要初始连接使用了常用的端口和 HTTP 头字段，就可以和防火墙和代理软件和谐相处。然而理想是丰满的，现实是骨感的。有些代理软件对 WebSocket 的“升级”请求的头信息做了修改，打破了协议规则。事实上，协议草案的最近一次更新（版本 76）也无意中打破了对反向代理和网关的兼容性。为了更好地更成功地使用 WebSocket，这里给出一些步骤：

101

- 使用安全的 WebSocket 连接 (wss)。代理软件不会对加密的连接胡乱篡改，此外你所发送的数据都是加密后的，不容易被他人窃取。
- 在 WebSocket 服务器前面使用 TCP 负载均衡器，而不要使用 HTTP 负载均衡器，除非某个 HTTP 负载均衡器大肆宣扬自己支持 WebSocket。
- 不要假设浏览器支持 WebSocket，虽然浏览器支持 WebSocket 只是时间问题。诚然，如果连接无法快速建立，则迅速优雅降级使用 Comet 和轮询的方式来处理。

那么，如何选择服务器端的解决方案呢？幸运的是，在很多语言中都实现了对 WebSocket 的支持，比如 Ruby、Python 和 Java。要再次确认每个实现是否支持最新的 76 版协议草案，因为这个协议是被大多数客户端所支持的。

- Node.js
 - node-Websocket-server (<http://github.com/miksago/node-websocket-server>)
 - Socket.IO (<http://socket.io>)

- Ruby
 - EventMachine (<http://github.com/igrigorik/em-websocket>)
 - Cramp (<https://github.com/lifo/cramp>)
 - Sunshowers (<http://rainbows.rubyforge.org/sunshowers/>)
- Python
 - Twisted (<http://github.com/rlotun/twWebSocket>)
 - Apache module (<http://code.google.com/p/pywebsocket>)
- PHP
 - php-WebSocket (<http://github.com/nicokaiser/php-websocket>)
- Java
 - Jetty (<http://www.eclipse.org/jetty>)
- Google Go
 - native (<http://code.google.com/p/go>)

102 Node.js 和 Socket.IO

在上面的名单中，Node.js (<http://nodejs.org>) 是一名新成员^{注4}，也是当下最受关注的新技术。Node.js 是基于事件驱动的 JavaScript 服务器，采用了 Google 的 V8 引擎 (<http://code.google.com/p/v8>)。正因为此，NodeJS 速度非常快，也可以解决服务器高并发连接数的资源消耗问题，和 WebSocket 服务器一样。

Socket.IO (<http://socket.io/>) 是一个 Node.js 库，实现了 WebSocket。最让人感兴趣的不止于此，来看一段官网上的宣传文字：

Socket.IO 的目标是在每个浏览器和移动设备中构建实时 app，这缩小了多种传输机制之间的差异。

如果环境支持 WebSocket，那么 Socket.IO 就会尝试使用 WebSocket，若有必要也会降级使用其他的传输方式。这里列出了所支持的传输方式，非常全面，因此 WebSocket.IO 可以做到更好的浏览器兼容：

- WebSocket
- Adobe Flash Socket
- ActiveX HTMLFile (IE)
- 基于 multipart 编码发送 XHR (XHR with multipart encoding)
- 基于长轮询的 XHR

注4：Oreilly 出版的一本小册子专门介绍 Node.js——《什么是 Node》(What is Node)，已经有中译本，请参考：<http://jayli.github.com/whatisnode/>。——译者注

- JSONP 轮询（用于跨域的场景）

Socket.IO 的浏览器支持非常全面。“服务器推”的实现是众所周知的难题，但 Socket.IO 团队为你解决了这些烦恼，Socket.IO 保证了它能兼容大多数浏览器，浏览器支持情况如下：

- Safari ≥ 4
- Chrome ≥ 5
- IE ≥ 6
- iOS
- Firefox ≥ 3
- Opera ≥ 10.61

尽管在服务器端实现的 Socket.IO 最初是基于 Node.js 的，现在也有用其他语言实现的版本了，比如 Ruby (Rack) (<http://github.com/markjee/Socket.IQ-rack>), Python (Tornado) (<https://github.com/MrJoes/tornadio>), Java (<http://code.google.com/p/socketio-java>) 和 Google Go (<http://github.com/madari/go-socket.io>)。

来看一下它的 API，写法非常简单、直接，客户端的 API 和 WebSocket 的 API 看起来很像：

```
var socket = new io.Socket();

socket.on("connect", function(){
    socket.send('hi!');
});

socket.on("message", function(data){
    alert(data);
});

socket.on("disconnect", function(){});
```

103

在后台 Socket.IO 会选择使用最佳的传输方式。正如在 *readme* 文件中所描述的，“你可以使用 Socket.IO 在任何地方构建实时 app”。

如果你想寻求比 Socket.IO 更高级的解决方案，可以关注一下 Juggernaut (<http://github.com/maccman/juggernaut>)，它就是基于 Socket.IO 实现的。Juggernaut 包含一个信道接口 (channel interface)：客户端可以订阅信道监听，服务器端可以向信道发布消息，即所谓的订阅 / 发布 (<http://en.wikipedia.org/wiki/PubSub>) 模式。这个库可以针对不同的客户端和实现环境作灵活扩展，比如基于 TLS 等。

如果你需要虚拟主机中的解决方案，可以参考 Pusher (<http://pusherapp.com/>)。Pusher

可以让你从繁杂的服务器管理事务中抽身出来，使你能将注意力集中在有意义的部分：web 应用的开发。客户端的实现非常简单，只需将 JavaScript 文件引入页面中并订阅信道监听即可。当有消息发布的时候，仅仅是发送一个 HTTP 请求到 REST API (<http://pusherapp.com/docs>)。

实时架构

将数据从服务器推送给客户端的理论看起来有点纸上谈兵，如何将理论和 JavaScript 应用的开发实践相结合呢？如果你的应用正确地划分出了模型，那么应用实时架构将会非常简单。接下来我们给出在应用中构建实时架构的每个步骤，这里大量用到了订阅 / 发布模式。首先需要了解的是将更新通知到客户端的整个过程。

实时架构是基于事件驱动的 (event-driven)。事件往往是由用户交互触发的：用户修改了数据记录，事件就会传播给系统，直到数据推送给已经建立连接的客户端并更新数据。要想为你的应用构建实时架构，则需要考虑两件事：

- 哪个模型需要是实时的？
- 当模型实例发生改变时，需要通知哪些用户？

实际情况往往是当模型发生改变时，你希望给所有建立连接的客户端发送通知。这种情况更多发生在网站首页需要实时提供活动的数据源的场景中，比如，每个客户端都能看到相同的信息。然而更多的应用场景是，要想针对不同的用户群发送不同的数据源，你可以根据不同类型的数据源有针对性地给用户推送更新。

103 ▶ 我们来看一个聊天室的场景：

1. 用户在聊天室中发送了一个新消息。
2. 客户端向服务器发送一条 Ajax 请求，并创建一条 Chat 记录。
3. 在 Chat 模型上触发了“保存”的回调，调用我们的方法来更新客户端数据。
4. 查找聊天室中所有和这个 Chat 记录有关的用户，我们需要给这些用户发送更新通知。
5. 用一条更新来描述发生了什么事情 (创建 Chat 记录)，将这个更新推送给相关的用户。

这个过程细节和你选用的服务器环境有关，然而，如果你使用 Rails, Holla (<http://github.com/maccman/holla>) 是一个非常不错的例子。当创建了 Message 记录时，JuggernautObserver 会更新相关的客户端。

现在就引入了另外一个问题：如何向特定用户发送通知？最佳方法是使用发布 / 订阅模式：客户端订阅某个特定的信道，服务器向这个信道发布消息。每个用户订阅惟一的信道，信道包含一个 ID，可能是用户在数据库中存放的 ID。然后，服务器只需向这个惟一的

信道发布消息即可，这样就可以做到将通知发送给特定的用户。

例如，某个用户可以订阅下面这个信道：

```
/observer/0765F0ED-96E6-476D-B82D-8EBDA33F4EC4
```

这里的随机字符串是当前登录用户惟一的标识。要想将通知发送给这个特定用户，服务器只需向同一个信道发布消息即可。

你可能很想知道发布 / 订阅模式在信息传输过程（WebSocket 或 Comet）中是怎样工作的。幸运的是，已经有很多可用的解决方案，比如 Juggernaut 和 Pusher，之前都有提到过。发布 / 订阅是最常见的抽象，处于 WebSocket 的最高层，不管你选用什么服务或库，它们的 API 都非常相似。

一旦服务器将通知推送给客户端，你将体会到 MVC 架构带来的美感。让我们回过头来看刚才的聊天室的例子。发送给客户端的通知格式看起来像这样：

```
{
  "klass": "Chat",
  "type": "create",
  "id": "3",
  "record": {"body": "New chat"}
}
```

它包含一个被更改的模型、更新类型和其他相关属性。使用它就可以让客户端在本地创建新的 Chat 记录。由于客户端的模型已经绑定了 UI，用户界面会根据新的聊天记录自动更新。

最让人吃惊之处在于这个过程并不和特定的 Chat 模型相关，如果我们想创建另一个实时模型，只需添加另外一个服务器观察者，确保服务器更新时客户端会随之更新即可。现在我们的后台和客户端模型绑定在一起。任何后台模型的更改都会自动传播给相关的客户端，并更新 UI。使用这种架构搭建的应用就是真正的实时应用。一个用户和应用产生的任何交互即刻被广播给其他的用户。

◀ 105

感知速度

速度是 UI 设计最重要也是最易忽略的问题，速度对用户体验（UX）的影响非常大，并直接影响网站的收益。很多大公司一直都在研究、调查速度和网站收益之间的关系：

Amazon

页面加载时间每增加 100 毫秒，就会造成 1% 的销售损失（来源：Greg Linden, Amazon）。

Google

页面加载时间每增加 500 毫秒，就会造成 20% 的流量损失（来源：Marrissa Mayer, Google）。

Yahoo!

页面加载时间每增加 400 毫秒，在页面加载完成之前就点击“后退”按钮的人会增加 5-9%（来源：Nicole Sullivan, Yahoo!）。

“感知速度”（perceived speed）和真实的速度同等重要，因为感知速度关系到用户的感官体验。因此，关键是要让用户“感觉”到你的应用很快，尽管实际的速度可能并不快，而这正是 JavaScript 应用带给我们的最大好处：尽管某一时刻在后台会有很多请求不会及时响应，但 UI 不会被阻塞。

让我们再次回过头来讨论一下刚才聊天室的场景。用户发送了新的消息，触发了一个 Ajax 请求。我们可以等待这个请求在网络中走一个来回之后，将响应结果更新到聊天记录中。然而，从发起请求的时刻开始，到获得响应并更新至聊天记录，会有几秒钟的延时。这会令应用看起来很慢，肯定会造成用户体验上的损失。

既然如此，为什么不直接在本地创建一个新记录呢？只需将消息立即添加至聊天记录中即可。用户会感知到这个消息被立即发送出去了，他们不知道（甚至不关心）这个消息是否被分发给聊天室中的所有人。只有这种清澈、流畅的产品体验，才会让用户倍感愉悦。

除了交互设计的小窍门之外，Web 应用中最耗时的部分是新数据的加载。最明智的做法是在用户请求数据之前预测用户的行为并预加载数据，这一点非常重要。预加载的数据被缓存在内存中，如果随后用户需要这个数据，就不必再发起到服务器的请求了。应用在启动伊始就应当预加载常用的数据。应用加载时的略微延时或许可忍，加载完成后糟糕的交互体验断不可忍。

106 ▶ 当用户和你的应用产生交互时，你需要适时给用户一些反馈，通常使用一些可视化的进度指示来给出反馈。用行业术语来讲就是“期望管理”（expectation management）——要让用户知道当前项目的状态和估计完成时间。“期望管理”同样适用于用户体验领域，适时地给用户一些反馈，告知用户发生了什么事情，会让用户更有耐心等待程序的运行。当用户等待新数据的加载时最好给出信息提示或一张旋转的小图片。如果在上传文件，则给出上传进度条及估计完成时间。这些都属于感知速度的范畴，可有效地提升产品的用户体验。

测试和调试

从某种程度上说，每个开发者在开发过程中都需要测试。起码手动执行代码也是一种测试。但本章将要讲的是 JavaScript 自动化测试——编写特定的断言并自动地执行它们。自动化测试不会消除程序中的 bug，但的确是一种必要的手段，用来减少程序出现瑕疵的几率并避免代码库出现较旧的 bug。测试有很多类型，而且网上已经有很多相关的学习资料。关于测试的基本原理和概念性的东西本章不再赘述，这里只关注基于 JavaScript 的测试以及和其他语言的区别。

很多人认为 JavaScript 的测试是一个鸡肋，因此多数 JavaScript 开发者没有为他们的程序写测试代码。在我看来这种认识的主要原因是，JavaScript 的自动化测试非常困难，且不具备可伸缩性^{注1}。我们以 jQuery 为例，jQuery 库中包含一百多个测试单元和大约十种不同的测试分组，以模拟在不同环境中的运行场景。每次测试都要运行所有的测试用例。jQuery 支持的浏览器包括：

- Safari: 3.2, 4, 5, 午夜版
- Chrome: 8, 9, 10, 11
- Internet Explorer: 6, 7, 8, 9
- Firefox: 2, 3, 3.5, 3.6, 午夜版
- Opera: 9.6, 10, 11

因此，jQuery 总共需要支持五大浏览器的二十多个版本，每个测试单元都要在所有浏览器中测试通过。你会发现测试量已经开始呈指数级增长，我们甚至还没考虑到各个平台

注1： 相比传统的软件开发，Web 应用的需求变化更加频繁，因此测试用例也在不断变化，这对自动化测试来说是一个挑战。另外，由于多数 JavaScript 功能的模块化和 API 的设计并不完善，编程模式和约定也不统一，导致某个功能的测试用例很难被复用。甚至，测试代码会随着需求的增加呈指数级的增长，这也就是作者所说的“不具备可伸缩性”。——译者注

的情况^{注2}！因此这种测试根本不具备可伸缩性。

显然，jQuery 是特例，只是为了说明问题到底有多糟。你的程序所兼容的浏览器往往不超过 jQuery 支持浏览器的二分之一，所以你也不需要这么多测试用例。然而，你必须针对你的应用所兼容的浏览器作充分测试。

108 在接下来的讨论之前，有必要先看一下各大浏览器的市场占有率，因为这个统计会最终决定前端开发工程师做开发所依赖的浏览器类型和版本。但这个占有率的数据变化非常快，今天的统计数据可能明天就会过时，但大致的趋势是不会变的。

浏览器的市场占有率的统计结果往往取决于数据采样的维度。不同的国家统计结果也不一样，这和每个国家互联网用户的专业程度有关。比如，下面的统计数据来自于 Statcounter.com，是 2011 年欧洲的数据：

- Safari: 4%
- Chrome: 15%
- IE: 36%
- Firefox: 37%
- Opera: 2%

IE 的占有率在不断减少，而 Firefox 和 Chrome 的占有率在不断攀升，这种趋势会一直持续下去。IE6 早已是明日黄花、风光不再了，占有率也跌至个位数。现在也只有为企业或政府部门开发网站时会考虑那些上了年纪的用户，除此之外基本不必再担心如何去兼容这些古老的浏览器了^{注3}。

古人云，“世界上有三种谎言：谎话，弥天大谎，还有统计数字”^{注4}。这句话用在浏览器的统计上再合适不过了。比如我的博客的流量统计显示 IE 占 5%，低于国家的平均值。换句话说，你所看到的统计数据是与你的网站的用户群有关系的。如果你的站点是科技类或喜欢炫一些花哨技术的站点，那么使用 Firefox 和 Chrome 的用户比例会非常高，而更多的主流网站的统计数据则能更好地反映国家的平均水平。首先考虑你的站点主要面向哪些人提供服务，然后决定要支持哪些浏览器，而不要太迷信统计数据。然而，根据经验来看，推荐大家主要测试这些浏览器：^{注5}

注2：指各种操作系统、PC、Mac 机和各种移动终端。——译者注

注3：根据淘宝网的统计数据，国内 IE6 的占有率依然非常高，所以作者所说的情况并不适用于中国大陆。——译者注

注4：这句谚语据说最初出自英国前首相本杰明·迪斯雷利（Benjamin Disraeli）之口，不过它的广泛流传却是因为马克·吐温的引用，意思是说统计数字不靠谱。——译者注

注5：雅虎最早提出了浏览器分级支持（GBS），即根据功能需求的权重来将浏览器支持划分为多个级别，并且非常详细、系统地定义了浏览器测试基准和操作系统支持标准，请参照 <http://yuilibrary.com/yui/docs/tutorials/gbs/>，而作者在这里提到的主流浏览器更多适用于 Web App，并非所有的 Web 页面。——译者注

- IE 8, 9
- Firefox 3.6^{注6}
- Safari 5
- Chrome 11

如果你不清楚你的网站访问来源的统计数据，也不知道用户用了哪些浏览器访问你的站点，就需猜测网站用户的受教育情况。不管通过什么方式，只要确定下来你的网站需要兼容的浏览器类型和版本，接下来就是写自动化测试，来确保你的应用能在这些浏览器中都正常运行。

单元测试

手工测试更像集成测试，从更高层次上保证应用的正常运行。单元测试则是更低层次的测试，确保特定的后台代码片段能正常运行。单元测试更多的是为了发现浏览器兼容性 bug，但这些 bug 的解决相对容易，因为被测试的代码片段往往很短。

单元测试的另一个优势是为自动化测试铺平道路。在本章后续的小节中会有深入讨论。将很多单元测试整合起来就可以做到连续的集成测试了——每次代码有更新时都重新执行一遍所有的单元测试。这要比对应用做手动回归测试省时省力得多，并可确保每一处代码的小改动都不会影响到应用中其他的功能。

现在有很多 JavaScript 单元测试类库，每种库都各有优、缺点。接下来会介绍一些主流的测试类库，但使用这些类库之前要理解测试的基本原理。

断言

断言是测试的核心^{注7}，它们决定了哪些测试会通过、哪些会失败。断言是一些表述代码期望执行结果的语句。如果断言不正确，则测试失败，你就会知道代码出了问题。

比如，这里有一个简单的 `assets()` 函数，本书中还有很多其他示例代码会用到它：

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

注6：撰写本书时的 Firefox 版本还是 3.6，从 Firefox4 之后版本升级非常快。最重要的不同是 Firefox 3.6 遵循 ECMAScript3，而 4 及以后的版本则遵循 ECMAScript5，因此这里更推荐使用 Firefox 4+。——译者注

注7：很多初学者将断言和测试用例混为一谈，断言是用来检查测试用例中的条件的，理论上断言是测试用例的子集。——译者注

它得到一个值和一个可选的字符串消息。如果值不是 `true`，那么断言失败：

```
// 这些断言均失败
assert( false );
assert( "" );
assert( 0 );
```

在 JavaScript 中，如在希望使用布尔值的地方使用了 `undefined`、`0` 和 `null`，则这些值都会被转换为 `false`。换句话说，下面这个 `assert` 可以对 `null` 做检查：

```
// 如果语句是 null，则断言失败
assert( User.first() );
```

类型转换多少会影响你的测试，因此有必要在类型转换之前首先检查值的类型，以避免很多怪异的代码引起各种奇怪的问题 (<http://goo.gl/M209w>)。

110 断言类库不仅限于正确性检查。大多数类库还包含完整的数组匹配，以及通过比较原始对象来检查数字的大小。这些库中至少都实现了 `assertEqual()` 函数，用以比较两个值：

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

// 断言通过
assertEqual("one", "one");
```

接下来要讲的测试类库都实现了一组断言，每个库定义断言的 API 都有所不同。

QUnit

QUnit (<http://docs.jquery.com/Qunit>) 是目前最流行且维护良好的测试类库，这个库最初是用来测试 jQuery 用的。那么，如何用 QUnit 建立测试环境呢？第一步是将项目文件下载到本地，然后创建静态测试执行页面：

```
<!DOCTYPE html>
<html>
<head>
  <title>QUnit Test Suite</title>
  <link rel="stylesheet" href="qunit/qunit.css" type="text/css" media="screen">
  <script type="text/javascript" src="qunit/qunit.js"></script>
  <!-- include tests here... -->
</head>
<body>
  <h1 id="qunit-header">QUnit Test Suite</h1>
  <h2 id="qunit-banner"></h2>
  <div id="qunit-testrunner-toolbar"></div>
```

```

    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests"></ol>
    <div id="qunit-fixture">test markup</div>
</body>
</html>

```

要想创建断言，你需要将它们放入测试用例中。比如，现在我们来创建一个测试单元，来测试在第3章里介绍的 ORM：

```

test("load()", function(){
    var Asset = Model.setup();

    var a = Asset.init();
    a.load({
        local: true,
        name: "test.pdf"
    });

    ok(a.local, "Load sets properties");
    equals(a.name, "test.pdf", "load() sets properties (2)");

    var b = Asset.init({
        name: "test2.pdf"
    });

    equals(b.name, "test2.pdf", "Calls load() on instantiation");
});

```

111

这里调用了 `test()` 来创建一个新测试用例，并给它设置了名字和测试回调（主逻辑都放在回调里）。在回调函数中我们使用了若干种断言：`ok()` 断言用来判断它的第一个参数是否是 `true`，`equals()` 用来比较它的前两个参数。所有的断言都可以带最后一个字符串参数，表示断言的名字，以便清楚地看出哪些断言通过了哪些失败。

现在把测试代码放入页面中并刷新这个页面，运行结果如图 9-1 所示：



图9-1：QUnit测试结果

运行结果非常清楚，非常不错！我们只需一眼就能看出哪些测试通过了，哪些没有通过，要做的只是每次测试的刷新页面。现在就可以将它在所有的浏览器中打开、运行，来检查应用的兼容性，确保应用在所有浏览器中都能正常运行。

使用 `module()` 函数可以将测试分离开，它包含一个名字和一个配置项。现在对上一个例子做进一步整理，给 `module()` 传入 `setup` 选项，它是一个回调函数，这个模块中的每个测试执行的时候都会调用它。在这个例子中，我们所有的测试都需要 `Asset`，所以我们在 `setup` 中创建它：

```
module("Model test", {
  setup: function(){
    this.Asset = Model.setup();
  }
});

test("load()", function(){
  var a = this.Asset.init();
  a.load({
    local: true,
    name: "test.pdf"
  });

  ok(a.local, "Load sets properties");
  equals(a.name, "test.pdf", "load() sets properties (2)");

  var b = this.Asset.init({
    name: "test2.pdf"
  });

  equals(b.name, "test2.pdf", "Calls load() on instantiation");
});
```

代码变得更清晰了一些，这在为它增加更多测试代码时非常有帮助。`module()` 还可以接收 `teardown` 选项，它是一个回调函数，模块中的每个测试执行完毕后会调用它。现在来给这个测试单元增加另一个测试：

```
test("attributes()", function(){
  this.Asset.attributes = ["name"];

  var a = this.Asset.init();
  a.name = "test.pdf";
  a.id = 1;

  equals(a.attributes(), {
    name: "test.pdf",
    id: 1
  });
});
```

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

如果执行这段代码，你会发现测试失败了，正如图 9-2 中所示。

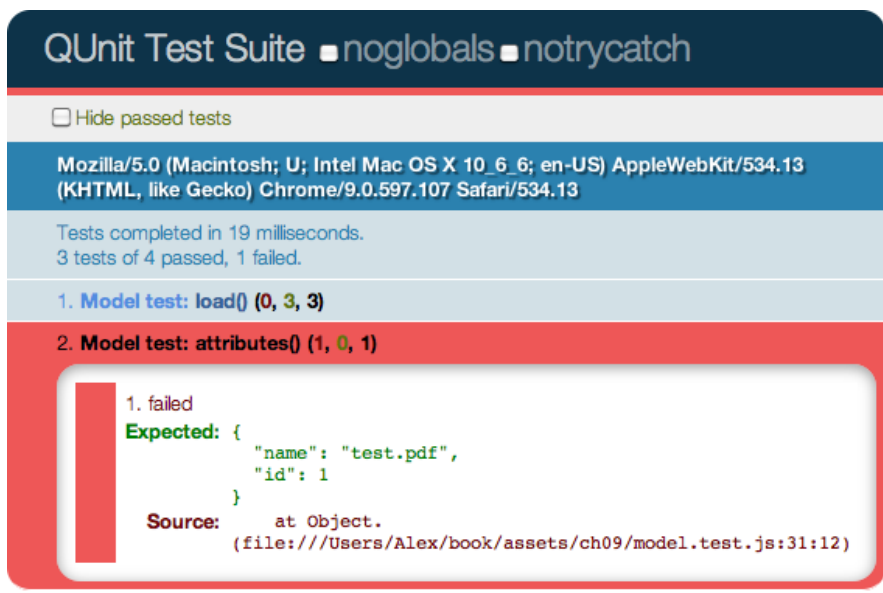


图9-2: QUnit出错提示

这是因为 `equals()` 函数使用了 `==` 比较运算符，它无法比较数组和对象。相反，我们应当使用 `same()` 函数，它会执行更深度的比较，现在测试单元就都通过了：

```
test("attributes()", function(){
    this.Asset.attributes = ["name"];

    var a = this.Asset.init();
    a.name = "test.pdf";
    a.id = 1;

    same(a.attributes(), {
        name: "test.pdf",
        id: 1
    });
});
```

QUnit 还包含很多其他的断言类型，比如 `notEqual()` 和 `raises()`。可以从这里看到这些断言的完整的使用方法：`assets/ch09/qunit/model.test.js`，在 QUnit 的文档 (http://docs.jquery.com/QUnit#API_documentation) 中也可以查阅到。

Jasmine (<http://pivotal.github.com/jasmine/>) 是另一个非常流行的测试类库（只是我的个人观点）。和 QUnit 不同，Jasmine 定义了用以描述应用中特定对象的行为的测试片段（spec）。实际上这些片段和单元测试非常类似，只不过换了一种表述方式。

Jasmine 的优势是它不依赖任何其他第三方库，甚至不依赖 DOM。也就是说它可以在所有的 JavaScript 环境中运行，比如可以在装有 Node.js 的服务器中运行。

和 QUnit 一样，我们需要创建一个静态 HTML 页面来载入测试代码，执行测试代码可以得到运行结果：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Jasmine Test Runner</title>
  <link rel="stylesheet" type="text/css" href="lib/jasmine.css">
  <script type="text/javascript" src="lib/jasmine.js"></script>
  <script type="text/javascript" src="lib/jasmine-html.js"></script>

  <!-- 引入源文件 ... -->
  <!-- 引入测试文件 ... -->
</head>
<body>

  <script type="text/javascript">
    jasmine.getEnv().addReporter(new jasmine.TrivialReporter());
    jasmine.getEnv().execute();
  </script>

</body>
</html>
```

114

现在来看如何用 Jasmine 来写测试代码，现在来对第 3 章介绍的 ORM 库作进一步测试：

```
describe("Model", function(){
  var Asset;

  beforeEach(function(){
    Asset = Model.setup();
    Asset.attributes = ["name"];
  });

  it("can create records", function(){
    var asset = Asset.create({name: "test.pdf"});
    expect(Asset.first()).toEqual(asset);
  });
});
```

```

it("can update records", function(){
  var asset = Asset.create({name: "test.pdf"});

  expect(Asset.first().name).toEqual("test.pdf");

  asset.name = "wem.pdf";
  asset.save();

  expect(Asset.first().name).toEqual("wem.pdf");
});

it("can destroy records", function(){
  var asset = Asset.create({name: "test.pdf"});

  expect(Asset.first()).toEqual(asset);

  asset.destroy();

  expect(Asset.first()).toBeFalsy();
});
});

```

这里使用 `describe()` 将测试代码划分为不同的分组（测试单元），每个分组都带有名字和一个匿名函数。在上面这个例子中，我们使用 `beforeEach()` 来为每段测试代码的运行创建设置程序，在每段测试代码运行之前会触发设置程序。Jasmine 同样包含“扫尾函数” `afterEach()`，在每段测试代码运行之后调用扫尾函数。我们在 `beforeEach()` 函数之外定义了变量 `Asset`，它对于每个测试单元来说是局部变量，在每段测试代码中都可以访问到它。

每段测试代码都是以 `it()` 函数开始，`it()` 函数带有名字和一个匿名回调函数，函数中包含我们所需的断言。通过给 `expect()` 函数传入期望得到的值来创建断言，包括调用一个判断匹配的方法，这些方法包括：

115

`expect(x).toEqual(y)`

比较对象或原始值 `x` 和 `y` 是否相等，如果相等则通过测试

`expect(x).toBe(y)`

比较对象或原始值是否相等，如果它们是同一个对象则通过测试

`expect(x).toMatch(pattern)`

比较 `x` 是否和字符串或者正则表达式相匹配，如果匹配则通过测试

`expect(x).toBeNull()`

如果 `x` 是 `null` 则通过测试

`expect(x).toBeTruthy()`

如果 x 的值为真值则通过测试

`expect(x).toBeFalsy()`

如果 x 的值为假值则通过测试

`expect(x).toContain(y)`

如果数组或字符串 x 包含 y 则通过测试

`expect(fn).toThrow(e)`

如果函数 fn 在执行时抛出了异常 e 则通过测试

Jasmine 还包含其他很多匹配方法，甚至可以写一些自定义的匹配方法（<http://goo.gl/ddH89>）。

图 9-3 是运行 Jasmine 测试代码时的结果，运行的就是上面这段代码。

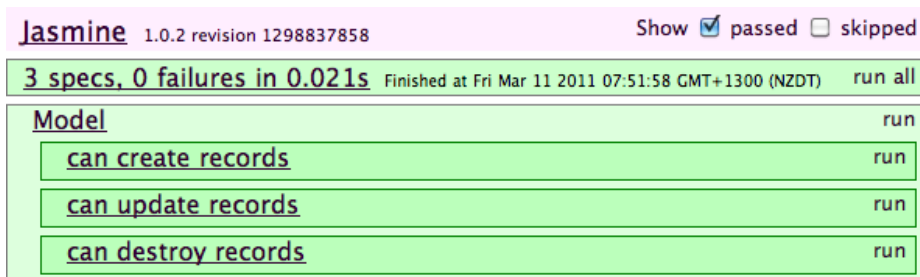


图9-3：Jasmine测试结果

驱动

尽管使用测试框架可以做到一定程度的自动化测试，但在各式各样的浏览器中进行测试依然是个问题。每次测试时都要开发者手动在五个浏览器中执行刷新，这种做法显然很低效。

为了解决这个问题，有人就开发出了驱动。这里所说的驱动实际上是一个守护进程^{注8}，它整合了不同的浏览器，可以自动运行 JavaScript 测试代码，测试不通过时会给出提示。^{注9}

注8：守护进程是一个操作系统级的概念，特指为了运行某些进程而开启的服务。——译者注

注9：前端开发自动化测试是现在比较热门的话题，国内也有很多前端团队开始了这方面的研究和尝试，可以参考来自淘宝的“前端测试实践”（<http://goo.gl/koPLJ>）以及来自新浪的“多浏览器集成的JavaScript单元测试”（<http://goo.gl/2CnHe>），两种实践思想和作者在本节提到的原则是一致的，即将不同的浏览器环境作集成，提供服务，以便展开下一步的自动化测试。——译者注

要想将驱动安装在每个开发者的机器上需要很多工作要做，所以很多公司都有一个单独的持续集成服务器，利用 post-commit 的 hook 功能^{注 10}来自动运行 JavaScript 测试代码，确保每次提交的代码都是正确无误的。

Watir (<http://watir.com/>) (Watir 的发音和 water 一样) 是一个基于 Ruby 的驱动类库，整合了 Chrome、Firefox、Safari 和 IE (IE 的版本和平台有关)。安装之后，可以通过给 Watir 发送 Ruby 指令来启动浏览器，而且可以像真实用户一样完成点击链接和填写表单等行为。在这个过程中，你可以运行一些测试用例和断言以判断程序运行是否满足期望：

```
# FireWatir drives Firefox
require "firewatir"

browser = Watir::Browser.new
browser.goto("http://bit.ly/watir-example")

browser.text_field(:name => "entry.0.single").set "Watir"
browser.button(:name => "logon").click
```

由于浏览器的安装受到操作系统的限制，如果你想测试 IE，你的持续集成服务器就需要安装特定版本的 Windows 操作系统，同样，如果你想测试 Safari，则需要一台安装 Mac OS X 的服务器。

另外一个非常流行的浏览器驱动工具是 Selenium (<http://seleniumhq.org/>)。这个库提供了一种域脚本语言 (domain scripting language, 简称 DSL)，用这种脚本语言可以为多种编程语言编写测试代码，比如 C#、Java、Groovy、Perl、PHP、Python 和 Ruby。Selenium 可以在本地运行。它往往是以后台服务的形式运行于持续集成服务器中，启动的方法和通过提交代码来启动测试一样，也会在测试不通过时给出提示。Selenium 的优势在于它支持很多编程语言，同时提供了一个 Firefox 插件 Selenium IDE (<http://seleniumhq.org/projects/ide/>)，这个插件可以记录浏览器的行为并可回放，这极大方便了开发者的自测。

如图 9-4 所示，我们使用 Selenium IDE 工具来记录链接的点击、填写表单和表单提交。当记录了一组行为时，你可以使用绿色的 *play* 按钮来回放这个过程。这个工具可以对记录下的动作和行为进行仿真模拟，这让测试变得更加轻松且高效。

注 10：为了方便管理员控制代码提交的过程，代码版本管理工具 SVN 提供了很多事件，*post-commit* 是其中一个事件，即“事务提交完毕，新的修订版被创建”。同时 SVN 还提供了 hook (钩子) 机制，即当特定的事件发生时，相应的 hook 会被调用，这样就可以做到某种程度的“自动化”。——译者注

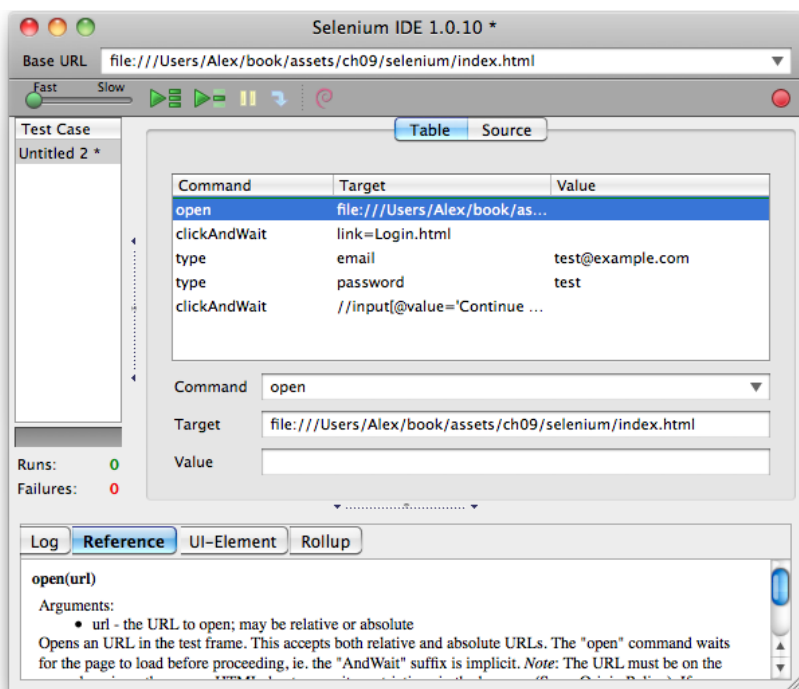


图9-4：使用Selenium记录的指令

我们可以将测试用例的记录导出为任何格式，如图 9-5 所示。

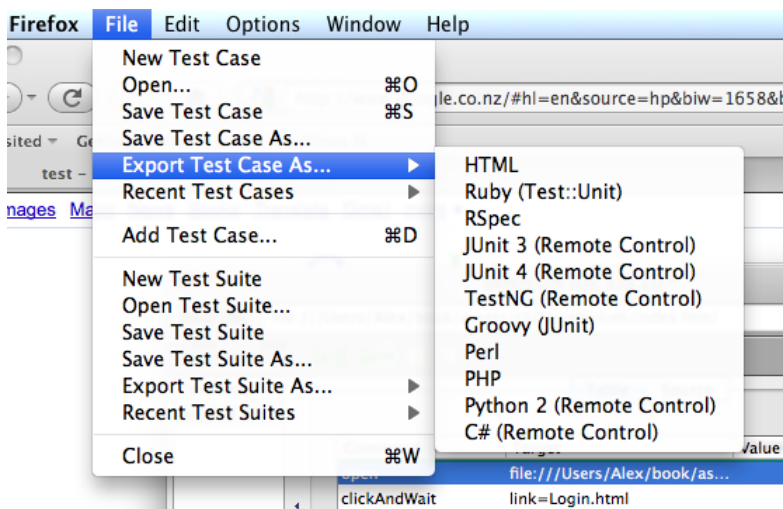


图9-5：将Selenium测试用例导出为各种格式

来看一个例子，这里的一段测试用例就被导出为 Ruby Test::Unit 用例。正如你所看到的，Selenium IDE 为我们生成了所有相关的驱动方法，极大地降低了直接在页面中手写代码的成本：

```
class SeleniumTest < Test::Unit::TestCase
  def setup
    @selenium = Selenium::Client::Driver.new \
      :host => "localhost",
      :port => 4444,
      :browser => "*chrome",
      :url => "http://example.com/index.html",
      :timeout_in_second => 60

    @selenium.start_new_browser_session
  end

  def test_selenium
    @selenium.open "http://example.com/index.html"
    @selenium.click "link=Login.html"
    @selenium.wait_for_page_to_load "30000"
    @selenium.type "email", "test@example.com"
    @selenium.type "password", "test"
    @selenium.click "//input[@value='Continue →']"
    @selenium.wait_for_page_to_load "30000"
  end
end
```

118

现在我们可以基于 @selenium 对象做断言，比如检查一个文本片段是否存在：

```
def test_selenium
  # ...
  assert @selenium.is_text_present("elvis")
end
```

可以访问 Selenium 的官网来获取更多帮助信息，或者参考这个视频教程：<http://seleniumhq.org/movies/intro.mov>。

无界面的测试

现在越来越多的人开始在服务器端（比如基于 Node.js 或 Rhino）编写 JavaScript 程序，这时就需要在脱离浏览器环境的命令行中运行你的测试代码。这种做法的优势是命令行环境速度快而且易于安装，同时不用涉及多浏览器及持续集成服务器环境。它的不足也很明显，就是测试代码无法在真实环境中运行。

119

这听起来不像是太严重的问题，因为你会发现你写的大多数 JavaScript 代码都是应用逻辑。

辑，是不依赖于浏览器的。而 jQuery 更多的是处理 DOM 和 Event 的浏览器兼容性问题，不会太深入业务逻辑层面。对于小型的应用来说，都会有一些固定的环境专门用于代码部署及一些高级的跨浏览器集成测试（不管是手动还是自动化），所以没有必要一定到真实的线上环境中执行测试。

Envjs (<http://www.envjs.com/>) 是 John Resig (jQuery 的作者) 开发的一个类库，这个类库在 Rhino 环境中实现了浏览器 DOM API，Rhino 是 Mozilla 实现的 JavaScript 引擎。你可以使用 Rhino 和 env.js 在命令行下执行 JavaScript 测试。

Zombie

Zombie.js (<http://zombie.labnotes.org/>) 是一个无界面的 JavaScript 类库，专门为 Node.js 设计，充分利用了它的高性能和异步特性。主要特点是速度快，花在等待测试执行上的时间越少，用在实现新功能和修复 bug 的时间就越多。

如果你的应用使用了很多客户端 JavaScript，那么加载、渲染和运行 JavaScript 的时间会很久。而如果使用纯粹的 Google V8 JavaScript 引擎则会让你的测试运行速度倍增。

尽管你的测试用例和客户端 JavaScript 都运行在相同的引擎之上，但 Zombine 还利用了 V8 的另一个特性——上下文，这个特性让它们彼此隔离^{注 11}，因此它们也不会共用同一个全局变量 / 全局上下文。这和 Chrome 中每个标签都各自独占一个进程是类似的。

上下文带来的另一个好处是使得多个测试代码可以并行运行，每个测试代码都有自己的 **Browser** 对象。一个用来检查 DOM 内容，另一个用来等待页面请求的回调，从而减少整个测试单元的运行时间。你需要使用一个异步测试框架，比如 Vows.js (<http://vowsjs.org/>)，这是一个非常优秀的框架，它可以识别出哪些测试需要并行执行，哪些测试需要串行执行。

Zombine.js 提供了一个 *Browser* 对象，它和真实的 web 浏览器中的 BOM 非常类似：它包含页面的状态 (cookie、历史记录和 web 存储) 以及提供了访问当前窗口的接口 (包括加载完成的 DOM 文档)。另外，它还提供了操作当前窗口的方法，可以模拟真实的用户和页面产生交互 (访问页面、填写表单、弹出对话框等)，还可以监控窗口的内容 (使用 XPath 或 CSS 选择器)。

119 ▶ 举个例子，来填写用户名和密码、提交表单并测试标题元素的内容：

```
// 填写邮件、密码并提交表单
browser.
```

注 11：“它们彼此隔离”是说基于业务逻辑的测试代码和处理浏览器兼容性的底层代码之间彼此隔离，因为开发者最不希望因为引入了测试代码而影响到原有业务逻辑的运行。——译者注

```
fill("email", "zombie@underworld.dead").
fill("password", "eat-the-living").
pressButton("Sign Me Up!", function(err, browser) {
  // Form submitted, new page loaded.
  assert.equal(browser.text("title"), "Welcome to Brains Depot");
});
```

这个例子是不完整的。显然你需要首先引入 `Zombie.js` 类库，创建新的 `Browser`，而且页面加载完成之后才能开始执行交互代码。同样，你需要注意这里的 `err` 参数。

和 web 浏览器一样，`Zombie.js` 本身就是异步的：你的代码不会阻塞页面的加载、事件的触发或定时器的超时。^{注 12} 相反，你可以注册事件监听（比如 `loaded` 和 `error`），或传入回调。

根据约定，当你将一个回调传入 `Zombie` 时，调用这个回调会有几种方式。如果执行成功，就会（给回调）传入 `null` 或其他值，多数情况下是传入 `Browser` 对象的引用。如果执行不成功，则传入一个 `Error` 对象。所以可以通过检查第一个参数的值来判断你的请求是否成功了，并判断是不是还有其他你感兴趣的参数。

对于 `Node.js` 来说这是一个通常的约定，很多库也实现了这个基本约定，包括刚才提到的 `Vows.js` 测试框架。`Vows.js` 同样使用了回调，这个回调的参数是 `error` 或 `null`，如果参数是 `null`，则会给这个测试用例带入第二个参数。

下面这段示例代码就是使用 `Zombie.js` 和 `Vows.js` 实现的一个测试用例。它访问了一个 web 页面并使用选择器 `“.brains”` 来查找元素（期望这个元素不存在）：

```
var zombie = require("zombie");

vows.describe("Zombie lunch").addBatch({
  "visiting home page": {
    topic: function() {
      var browser = new zombie.Browser;
      browser.cookies("localhost").update("session_id=5678");
      browser.visit("http://localhost:3003/", this.callback);
    },
    "should find no brains": function(browser) {
      assert.isEmpty(browser.css(".brains"));
    }
  }
});
```

使用 `Zombie.js` 还可以做很多事情。比如，你可以在测试代码运行结束后将浏览器状态

注 12：这里的“异步”指的是 JavaScript 的异步编程模型，而不是浏览器的渲染机制，众所周知浏览器的渲染是单线程的，直接在页面中引入的 JavaScript 文件是会阻塞页面渲染的。——译者注

保存下来 (cookie、历史记录、web 存储等), 以便其他的测试代码也能使用这些状态 (例如使用旧的 session 来保持登录状态, 这样运行新的测试代码就不用再去登录了)。

121 同样你也可以触发 DOM 事件, 比如模拟一次鼠标点击, 或者对提示框做出响应。你可以查看请求和响应的历史记录, 和 WebKit (开发者工具) 监视器中的“资源”选项卡类似。尽管 Zombie 是运行在 Node.js 之上的, 它也可以向其他服务器发起 HTTP 请求, 因此完全可以用它来测试你的 Ruby 或 Python 应用。

Ichabod

Ichabod (<http://github.com/maccman/ichabod>) 是另一个用于无界面测试的类库, 这个名字很富想象力。^{注 13} 如果你需要一个简洁、高效的测试运行环境, 强烈推荐你使用 Ichabod。

除了能模拟 DOM 和解析引擎之外, Ichabod 的优势还在于它使用了 Webkit 解析引擎, 这也是 Safari 和 Chrome 浏览器所使用的解析引擎。但它的缺点是只能运行在 OS X 中, 因为它需要 MacRuby 和 OS X WebView API 的支持。

Ichabod 的安装很简单, 首先需要安装 MacRuby, 可以直接从项目官网下载、安装 (<http://www.macruby.org/>), 也可以通过 rvm (<http://rvm.beginrescueend.com/interpreters>) 安装。下面, 就来安装 Ichabod gem :

```
$ macgem install ichabod
```

Ichabod 目前可以运行 Jasmine 和 QUnit 的测试代码, 后续还会增加更多对测试类库的支持。运行测试代码也很简单, 只需将待测试的地址带入 ichabod 命令即可 :

```
$ ichabod --jasmine http://path/to/jasmine/specs.html
$ ichabod --qunit http://path/to/qunit/tests.html
```

待测试页面不一定非得是外网地址, 也可以是本地文件路径 :

```
$ ichabod --jasmine ./tests/index.html
...
Finished in 0.393 seconds
1 test, 5 assertions, 0 failures
```

Ichabod 会把待测试的文件载入进来, 并在一个无界面的 WebKit 中运行它们, 这一切都可以在命令行中完成。

注 13 : Ichabod 是迪斯尼的动画电影 *The Adventures of Ichabod and Mr. Toad* (伊老师与小蟾蜍历险记) 中的主人翁, 这部电影 1949 年上映, 是迪斯尼第一部颇具影响力的长篇动画片。本片荣获当年金球奖最佳色彩片奖, 也是该奖首次颁发给动画电影。——译者注

分布式测试

跨浏览器测试的一个解决方案是利用外包的专用服务器集群，这些集群都安装有不同的浏览器。这正是 TestSwarm (<http://swarm.jquery.org/>) 的做法：

TestSwarm 的目标是简化在多个浏览器中执行繁琐、耗时的 JavaScript 测试用例的工作。它为你的 JavaScript 项目提供了一个持续集成的工作流，并带有必要的测试工具。

TestSwarm 并不是通过向浏览器集成一些插件和扩展来实现，这种做法非常低端，而是选择了另外一种思路。浏览器在 TestSwarm 的终端里运行，并自动执行推送给它们的测试。它们可以部署在任意机器、任意操作系统中，TestSwarm 会自动将得到的测试地址传递给一个新打开的浏览器。

122

这种方法听起来很简单，但仍然需要对持续集成服务器做大量的部署工作，这个过程也非常痛苦和麻烦。要解决的主要问题是确保很多浏览器都能正确地连接至 TestSwarm。实际上可以将这个工作外包给更专业的社区和公司来做，你只需告诉它们你想要的测试平台即可。图 9-6 就是 TestSwarm 以表格形式返回测试在各浏览器中执行测试用例的结果。

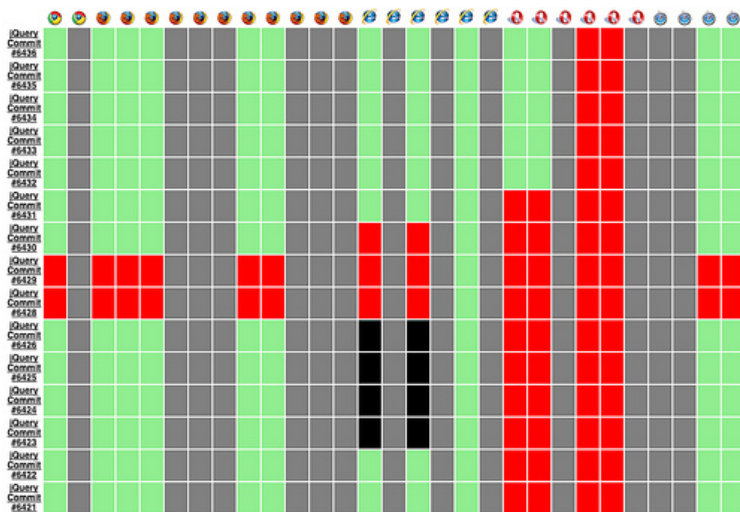


图9-6：TestSwarm执行测试用例的结果

可以选一些公司，比如 Sauce Labs (<http://saucelabs.com>) 提供的成熟的服务，这些公司大多是将浏览器运行于云端^{注 14} 你只需运行一个 Selenium 测试驱动，剩下的工作都交

注 14：请参照 <http://zh.wikipedia.org/zh-cn/> 云计算。——译者注

给服务器去做，包括在不同的平台启动不同的浏览器来运行你的测试代码，以及确保所有的场景下都能通过测试。

提供支持

不管你的测试是多么严格，你的应用一定还会存在 bug。你最好接受这个现实，要知道你上线的应用总会有 bug，用户也或多或少会遇到这些 bug 和错误。所以应当提供一种反馈机制，让用户可以很容易地将错误报告提交给你，同时你也应当做到有效地响应这些提交上来的 bug 报告。

123

调试工具

在过去，前端开发工程师最常用的开发和调试工具大概就是 `alert()` 了。现在大多数主流浏览器都包含了强大的元素查看器和调试器，这极大地提高了前端开发工程师的工作效率，降低了调试的复杂度。接下来会讲解两个主要的调试工具。浏览器中的调试工具都大同小异，学会使用一个，其他的也能很快上手。

Web Inspector

Web Inspector 是 Safari 和 Google Chrome 浏览器自带的调试工具。Safari 和 Chrome 两者的调试工具的界面稍有差异，但功能都是一样的。

在 Safari 中，需要在“Preferences...” / “Advanced”面板中勾选“Show Develop menu in menu bar”，如图 9-7 所示。

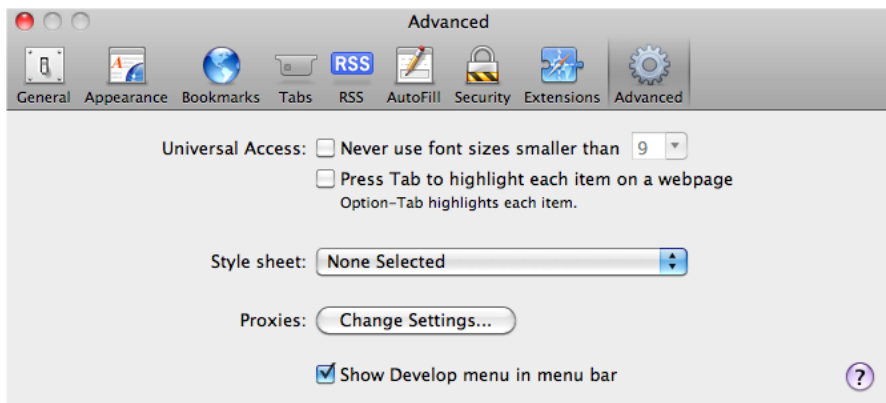


图9-7：启用Safari中的调试工具

Chrome 的开发者工具则在视图工具栏里，可以通过菜单来选择激活调试工具。在两个浏览器中都可以通过单击右键后选择“查看元素”来启用调试工具。

如图 9-8 所示，Web Inspector 是一个非常有用的工具，使用它可以监控 HTML 元素、编辑样式、调试 JavaScript 等。目前这种调试器已经是前端开发工程师调试页面和 JavaScript 程序不可或缺的工具了。

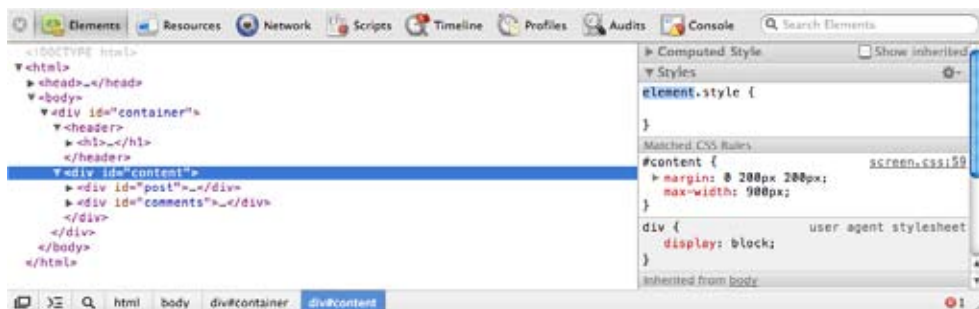


图9-8：使用Safari的Web Inspector抓取DOM元素

下面着重介绍一下调试工具的特性，但这里仍只是概要地介绍调试工具的组成部分：

Elements

监控 HTML 元素，编辑样式

Resources

页面的源文件和所需的资源

Network

HTTP 请求

Scripts

JavaScript 文件和调试器

Timeline

浏览器渲染过程的详细预览

Audits

代码和内存统计

Console

执行 JavaScript，并可查看结果

Firebug

Firefox 没有自带 JavaScript 调试工具，但它有一个非常优秀的扩展插件：Firebug (<http://getfirebug.com/>)，如图 9-9 所示。

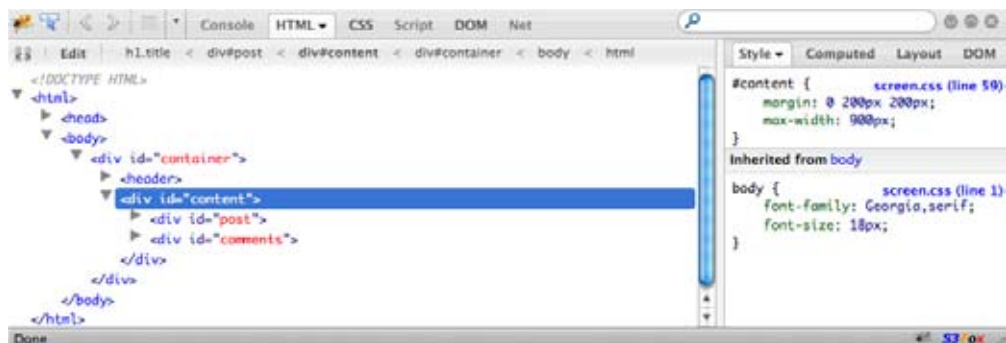


图9-9：使用Firebug抓取DOM元素

125 你可以发现，尽管 Firebug 操作面板中各个选项卡的名称和 Web Inspector 不一样，但他们的功能大都是类似的：

Console

执行 JavaScript 代码并查看结果

HTML

查看 DOM 元素，编辑样式

CSS

查看并编辑页面的 CSS

Script

JavaScript 文件和调试器

DOM

查看全局变量

Net

HTTP 请求

Firebug 的开发团队还开发了一个不依赖于 Firefox 的 Firebug——Firebug Lite (<http://>

`getfirebug.com/firebuglite`)^{注 15}。这个版本包含 Firebug 的大部分功能,甚至外观和交互都和 Firebug 一样,而且兼容主流的浏览器。尤其是在 IE 中,使用 Firebug Lite 进行调试会非常方便(甚至比 IE 内置的调试工具还好用)。Firebug Lite 不需要安装,只需在页面中引入一个 script 标签即可:

```
<script type="text/javascript" src="https://getfirebug.com/firebug-lite.js"></script>
```

除此之外你还可以在 Firebug Lite (<http://getfirebug.com/firebuglite#Stable>) 官网将它加入到书签里。

控制台

我们可以使用控制台来轻松地执行 JavaScript 代码并检查页面的全局变量。控制台的一个主要优势是你可以使用 `console.log()` 函数直接向它输出 log。调用是异步的,可以带多个参数,并且不用将参数转化为字符串:

```
console.log("test");
console.log(1, 2, {3: "three"});
```

还有另外一类的 log 提示。可以使用 `console.warn()` 和 `console.error()` 来提高 log 的级别,利用这两个函数可以在产生真正的程序错误之前就给出提示:

```
console.warn("a diabolical warning");
console.error("something broke!");
```

```
try {
  // 可能会出问题的代码逻辑
} catch(e) {
  console.error("App error!", e);
}
```

126

同样,使用一个代理函数也可以对 log 做命名空间的管理:

```
var App = {trace: true};
App.log = function(){
  if (!this.trace) return;
  if (typeof console == "undefined") return;
  var slice = Array.prototype.slice;
  var args = slice.call(arguments, 0);
  args.unshift("(App)");
  console.log.apply(console, args);
};
```

注 15: 在 IE 中使用 Firebug Lite, 不支持 JavaScript 断点调试。——译者注

这里的 `App.log()` 函数给它的参数都加上了字符串 “App” 前缀，然后调用了 `console.log()`。

当使用控制台输出 `log` 的时候，变量 `console` 有可能没有定义，不要忘了这一点。在那些不支持控制台的浏览器中——比如 IE 或没有安装 Firebug 的 Firefox——就没有定义 `console` 对象，这时使用 `console.log()` 就会出错。这也是推荐使用代理函数 (`App.log()`) 来为应用输出 `log` 的原因^{注 16}。

可以使用 `console.trace()` 来向控制台输出运行时脚本的当前堆栈。如果你想查看当前函数的被调用关系，这个方法非常有用，因为它可以追踪到程序中堆栈里所有的调用：

```
// 输出调用堆栈
console.trace();
```

应用程序的错误都会显示在控制台中，并且除非浏览器的 JIT 编译器^{注 17} 对函数调用做了优化，否则控制台会显示出完整的堆栈调用。

控制台函数

控制台通常提供很多快捷的工具函数，方便我们的调试。比如 `$0` 到 `$4` 变量存放的是用 Web Inspector 或 Firebug 选取的当前和前三个节点。相信我没错，这四个变量实在是太有用了，尤其是当你想访问多个被操作的元素时：

```
// $0 是当前选中的元素
$0.style.color = "green";

// 也可以使用 jQuery
jQuery($0).css({background: "black"});
```

可以给 `$()` 传入元素的 ID 来得到特定的元素。其实 `$()` 是 `document.getElementById()` 的缩写。jQuery、Prototype 或其他类库也都重写了 `$`：

```
$("user").addEventListener("click", function(){ /* ... */});
```

128 `$$()` 函数返回了匹配某个 CSS 选择器的一组元素组成的数组，这和 `document.querySelectorAll()` 类似。同样，如果你使用 Prototype 或 Mootools 这些类库，要注意这些类库也重写了这个函数：

注 16：很多类库都实现了输出 `log` 的代理函数，比如在 YUI3 中就实现了 `Y.log()`，甚至实现了一个轻型的控制台小组件，可以在不支持控制台的浏览器中正常地查看 `log`，请参照 <http://yuilib.com/yui/docs/console/console-global.html>。——译者注

注 17：JIT 是 Just-In-Time 的缩写，JIT 编译器的工作主要是对源码做优化，目的是提高程序执行效率，其中一个步骤就是减少函数的调用层级，更多信息请参照 http://en.wikipedia.org/wiki/Just-in-time_compilation。——译者注

```
// 使用 .users 来选取一组元素
var users = $$(".users");
users.forEach(function(){ /* ... */ });
```

`$x()` 函数返回了匹配某个 XPath 表达式的一组元素组成的数组：

```
// 选择所有的 form
var checkboxes = $x("/html/body//form");
```

`clear()` 函数用来将控制台里的 log 清空：

```
clear();
```

`dir()` 输出了对象中的所有属性：

```
dir({one: 1});
```

`inspect()` 的参数可以是元素、数据库或存储区域，并会自动跳转到调试工具的对应面板以显示相关的信息：

```
inspect($("user"));
```

`keys()` 返回由对象中所有的属性的名字组成的数组：

```
// 返回 ["two"]
keys({two: 2});
```

`values()` 返回由对象属性值组成的数组，用法和 `keys()` 类似：

```
// 返回 [2]
values({two: 2});
```

使用 JavaScript 调试器

JavaScript 调试器是做 JavaScript 开发必不可少的工具。它提供了完整的调试功能，可以设置断点、查看表达式、监控变量，以及处理和运行时相关的一切事务。

设置断点非常简单，只需在脚本中想设断点的位置添加 `debugger` 语句，程序执行到 `debugger` 时就会停住：

```
var test = function(){
  // ...
  debugger
};
```

同样地，你还可以在调试器中的脚本选项卡面板里找到要设断点的位置，然后点击左侧的行号，如图 9-10 所示的例子。

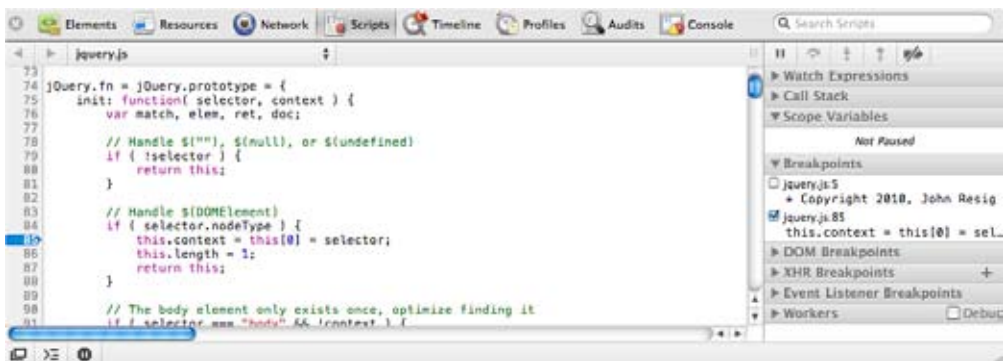


图9-10：在Safari的Web Inspector中设置断点

第二个方法更好一些，因为你不必冒在生产环境的代码中添加 debugger 的风险了。当JavaScript 执行到设置断点的行时就会停止执行，你就可以监控当前的调用堆栈了，如图 9-11 所示。

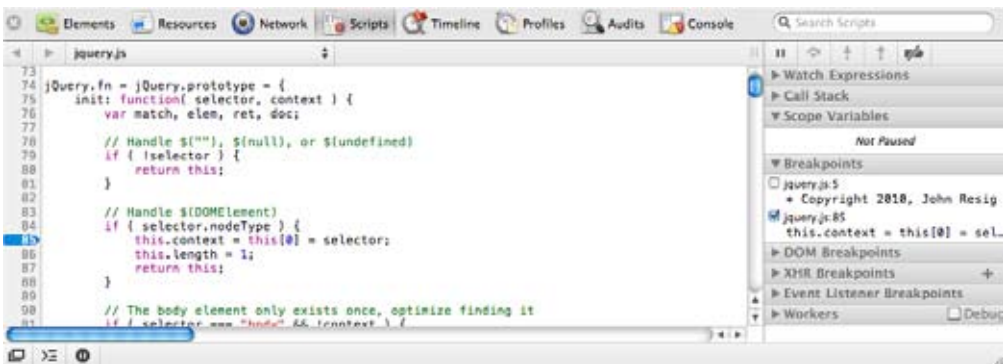


图9-11：在Safari的Web Inspector中调试断点

在脚本面板的右侧可以看到完整的调用堆栈，包括局部和全局变量，以及其他相关的调试信息。可以将鼠标移动到每个变量之上来查看其当前的值。控制台当前所在的上下文和断点之处的上下文一样，这样你就可以直接操作变量和执行函数了。

你可以继续执行代码，进入函数调用，跳出函数调用，并在调试工具条的右侧查看当前的调用堆栈。调试工具条上的每个选项卡都带有一个图标，可以将鼠标移动到这些图标上，这时会弹出一个黄色的小气泡提示，说明这个选项卡的功能。

页面刷新之后断点依然存在，这一点不要忘记。如果你想移除一个断点，也只需再次点

击设置了断点的行号，或者在断点列表中将它们的复选框去掉勾选。JavaScript 调试器是代替 `console.log()` 的强大工具，它能帮助你深入应用程序的内部调试每一行代码。

分析网络请求

如图 9-12 所示，调试工具中的网络监控面板显示了本页面发起的所有 HTTP 请求，包括每个请求耗费的时间及何时完成。^{注 18}

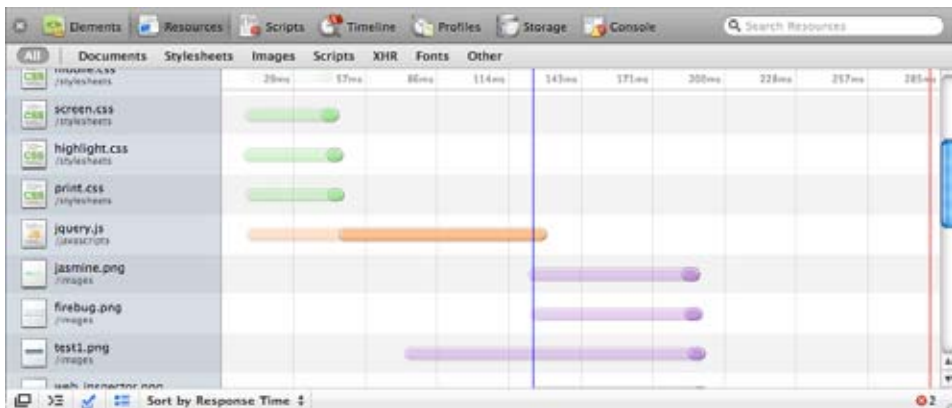


图9-12：使用Web Inspector分析网络请求

你可以看到初始请求的延时是用半透明的颜色表示的。当开始接收数据时，时间轴的颜色就变成不透明。在上图所示的例子中，jQuery 的文件体积比样式表更大，因此尽管初始请求的延时和样式差不多，但下载所耗费的时间更长。

如果你没有在 `script` 标签中使用 `async` 或 `defer` 属性（参照第 10 章），则会注意到 JavaScript 文件的加载是串行的而不是并行的。只有当前一个脚本加载完成并执行之后才会发起下一个脚本请求。^{注 19} 其他所有的资源都是并行加载的。

网络时间轴中的竖线表示了页面加载的状态。蓝色的线表示 `DOMContentLoaded` 事件的触发时间，换句话说就是 DOM 加载完成的时间。红色的线表示页面的 `load` 事件触发的时间，当页面中所有的图片和资源文件都下载完成时页面才算加载完成，这时触发 `load` 事件。^{注 20}

注 18：网络面板只能监控页面发出的 HTTP 请求，如果页面中包含 Flash，则 Flash 发起的请求是无法在网络面板中看到的，只能用更底层的抓包工具，比如 WireShark。——译者注

注 19：作者这里的表述有错误，浏览器是可以并行加载外部脚本文件的，只是脚本一定要按照顺序执行。——译者注

注 20：还有一个非常重要的时间线“页面首次渲染时间”，通常是绿色的线，这条线在 Firebug 和 Web Inspector 中看不到，可以使用 HttpWatch (<http://www.httpwatch.com/>) 来查看，这条线的重要性丝毫不亚于蓝线和红线。——译者注

在网络面板中还可以打开每个 HTTP 请求的头信息，这样就可以非常方便地查看到这些请求是否被正确地缓存，如图 9-13 所示。

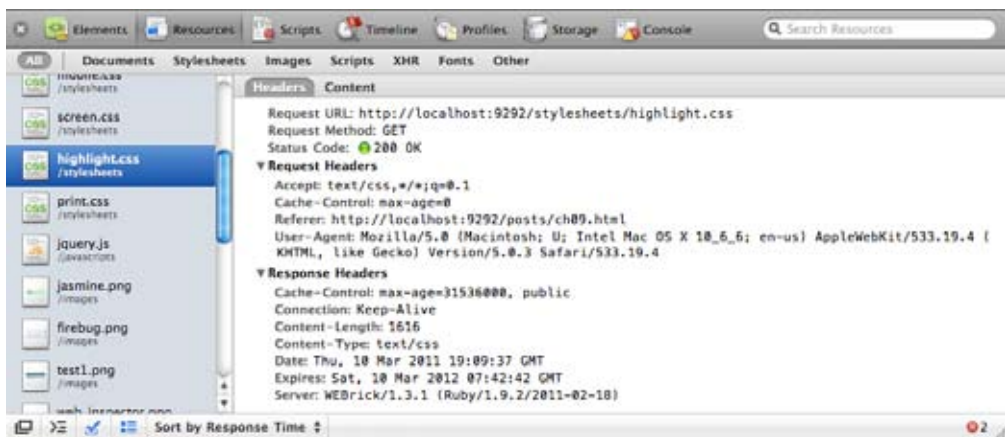


图9-13：查看HTTP请求的头信息，包括请求头和响应头

Profile 和函数运行时间

如果你所构建的是一个大型 JavaScript 应用，你需要特别关注性能，特别是当你的应用是运行在移动终端时。Web Inspector 和 Firebug 都包含了检查程序运行效率和时间的工具，它们可以帮助你更精确地把控程序的性能。

Profile^{注 21} 代码很简单，只要在你想统计的代码段两端加上 `console.profile()` 和 `console.profileEnd()` 即可：

```
console.profile();
// ...
console.profileEnd();
```

当调用到 `profileEnd()` 时，控制台就会创建一个报表，将期间所有的函数调用都统计出来，包括每次调用花费的时间及调用次数，如图 9-14 所示。

注 21：这里的 Profile 指的是调试工具提供的一个功能，用来统计代码中函数执行的次数和时间，并给出报表。——译者注

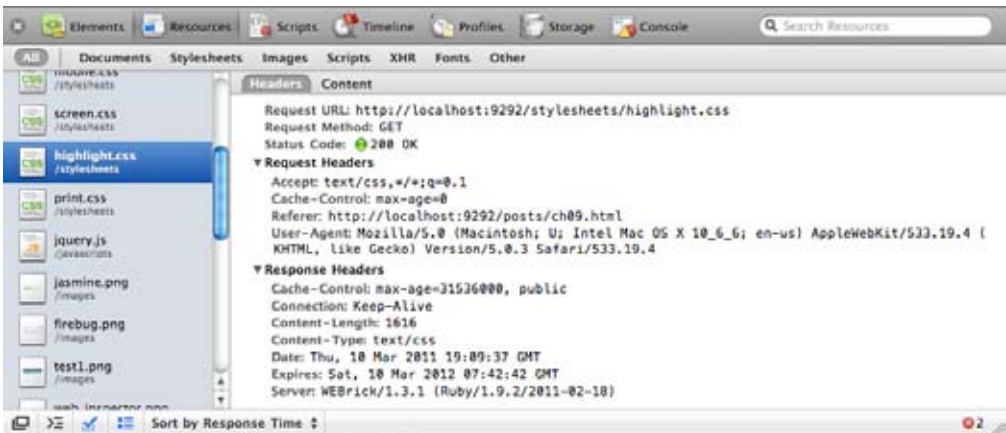


图9-14：使用Web Inspector统计出函数执行情况的报表

同样，你也可以使用调试器 Profile 的 *record* 特性，它的功能和直接嵌入 console 语句是一样的。通过查看哪些函数被调用了及哪些函数耗费了更长的时间，可以发现代码中的性能瓶颈。

◀ 131

你也可以使用 Profile 的快照 (*snapshot*) 功能生成页面当前的堆 (heap) ^{注 22} 的快照，如图 9-15 所示。这里显示了当前使用了多少对象，占用了多少内存。这是查找内存泄漏的好方法，因为你可以看到哪些对象被无意间存储在内存中，应当被回收而未被回收。

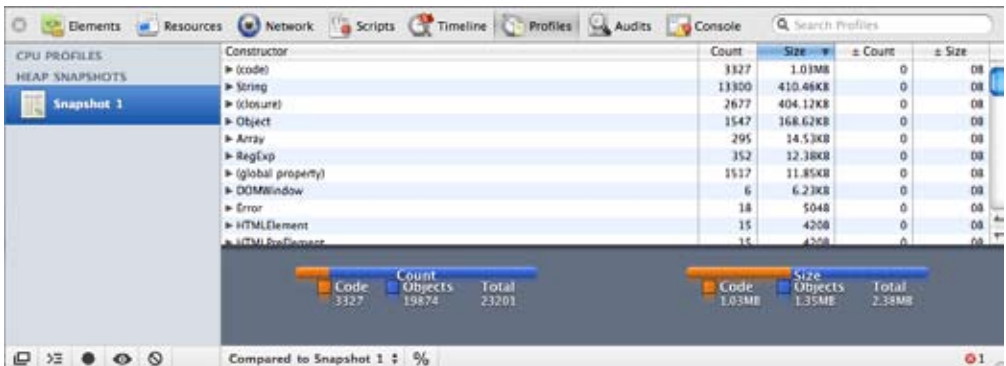


图9-15：使用Web Inspector查看堆的快照

注 22：和其他编程语言一样，JavaScript 运行时的内存也划分为堆 (heap) 和栈 (stack)。栈是用来存储局部变量的原始值和“引用”（可以将引用理解为一个内存地址）的，而堆则是存放“引用值”（引用指向的内容）的，这里的快照查看的是堆的内容，而不是栈的内容，主要是因为和堆相比栈的内存占用很小。更多内容请参照 <http://goo.gl/lbECT>。——译者注

使用控制台工具函数同样可以查看代码的执行时间。API 和 Profile 类似，只需给要统计时间的代码前后加上 `console.time(name)` 和 `console.timeEnd(name)` 即可。你可以直接在脚本代码中加入这两句：

```
console.time("timeName");  
// ...  
console.timeEnd("timeName");
```

当执行到 `timeEnd()` 时，它们之间的代码执行时间就会以毫秒为单位发送给控制台，以 `log` 的形式输出。使用控制台的时间统计 API，可以将性能测试也加入到你的测试代码中，以保证你的代码不会出现性能瓶颈，从而从整体上保证应用的良好用户体验。

部署

事实上，合理地部署你的 Web 应用程序和开发它一样重要；对于那些使用 Facebook 的真实用户来说，如果加载速度提升不上去，即便是开发新版本的 Facebook 也无太大意义。用户希望你的站点在提供服务的时候，能尽量的可靠和快速。部署 JavaScript 和 HTML 文件听起来很简单——毕竟它们只是资源文件——但它们的数量相当的大。在构建 Web 应用程序时，这一部分是最容易被忽略的。

幸运的是，已经有一些久经考验的技术，适合所有 JavaScript 应用程序，事实上它们也适合任何类型的静态资源文件。如能遵守以下的建议，你将找到自己的方法来加速你的 web 应用。

性能

提高性能，最简单的也是最显著的方法就是：减少 HTTP 请求的数量。每一个 HTTP 请求除了有 TCP 开销以外，还包含了大量的头信息。保持最小的独立连接数可以保证用户的页面加载速度更快。这显然涉及到了服务器需要传输的数据量的问题。让页面和其资源文件保持较小的体积将减少网络用时——对任何互联网上的应用而言，这才是真正的瓶颈。

将多个脚本文件合并成一个脚本文件，或将多个 CSS 合并成一个样式表，能减少页面渲染所需的 HTTP 连接的数量。可以在部署或运行时这样做。如果是后者，务必保证生成的文件在生产环境中可以被缓存。

使用 CSS Sprites 技术合并多张小图为一张大图，然后使用 CSS 的 `background-image` 和 `background-position` 属性在页面中显示对应的图片。只需要设定图片要显示的尺寸和背景位置的偏移坐标。

避免重定向也是减少 HTTP 请求的数量的方法。你也许认为这很少见，其实 URL 结尾缺少斜线 (/) 是一个最常见的重定向场景，而这个斜线不应当被丢掉。例如，当前访问 `http://facebook.com` 时，会被重定向到 `http://facebook.com/`。如果使用了 Apache，可以使用 Alias 或 `mod_rewrite` 来修正这个问题。

理解浏览器如何下载资源也很重要。为了加速页面渲染，现代浏览器并行下载所需的资源。但是，在所有的样式表和脚本下载完成之前，页面是不会开始渲染的。有些浏览器更是变本加厉，在处理任何 JavaScript 文件时，都会阻塞其他资源的下载。

尽管如此，大多数脚本需要访问 DOM，并且增加一些诸如事件句柄之类的东西，它们会在页面加载完成后执行。换言之，浏览器没有必要在一切都下载完成之前限制页面的渲染，因为这样做降低了性能。通过设置脚本的 `defer` 属性可以解决这个问题——告诉浏览器该脚本不会在页面加载完成之前操作 DOM：

```
<script src="foo.js" type="text/javascript" charset="utf-8" defer></script>
```

`defer` 属性设置为“`defer`”的脚本将和其他资源一起并行下载，它们不会阻塞页面的渲染。HTML5 还引入了一个新的脚本下载和执行的模式，称作 `async`。通过设置 `async` 属性，脚本将在完成下载后等待合适的时机执行代码。这意味着有可能（很有可能）异步不会按照它们在页面中出现的顺序执行代码，这又可能导致脚本执行如有依赖顺序时出错。如果脚本没有依赖关系，`async` 则是很有用的。例如 Google Analytics 默认利用了该特性：

```
<script src="http://www.google-analytics.com/ga.js" async></script>
```

缓存

如果没有缓存，互联网早就在网络流量拥堵下崩溃了。缓存就是将最近请求的资源存储到本地，以便接下来的请求能从磁盘中使用这些资源，而不用再次去下载。明确地告诉浏览器什么是可以被缓存的是很重要的。有些浏览器（如 Chrome）会使用自己默认的缓存策略，但也不能就此完全依赖它。

针对静态资源，可通过添加一个表示“在很远的将来才过期”的 `Expires` 头，让缓存“永不”失效。这将保证浏览器只会下载该资源一次。所有的静态资源文件都应该这样设置，包括脚本、样式表和图片。

```
Expires: Thu, 20 March 2015 00:00:00 GMT
```

建议相对于当前日期设置一个表示“很远的将来”的失效日期。如上例告诉浏览器该缓存在 2015 年 3 月 20 日之前不会过期。如果使用了 Apache，用 `ExpiresDefault` 可以方

便地设置一个相对的失效日期：

```
ExpiresDefault "access plus 5 years"
```

但是如果在那个时间之前让资源过期怎么办呢？一种很有用的技术就是——在引用资源文件的 URL 查询参数中添加文件的修改时间（或 mtime）。例如，Rails 默认采用这种方式。这样一来，任何时候文件被修改，资源文件的 URL 都会改变，也即清除了缓存。

```
<link rel="stylesheet" href="master.css?1296085785" type="text/css">
```

HTTP 1.1 引入了一类新的头，Cache-Control。它带给开发者更高级的缓存，同时还弥补了 Expires 的不足。Cache-Control 的控制头信息有很多选项，可用逗号隔开：

```
Cache-Control: max-age=3600, must-revalidate
```

查看全部的选项，请访问规范（<http://www.ietf.org/rfc/rfc2616.txt>）。现将其中很可能会经常用到的选项列举如下：

max-age

以秒为单位，指定资源的最大有效时间。和 Expires 不一样的是，该指令是相对于该请求的时间。

public

标记资源是可被缓存的。默认情况下，通过 SSL 或使用 HTTP 认证后访问的资源，缓存是关闭的。

no-store

完全关闭缓存，动态内容才会更多地使用这个选项。

must-revalidate

告诉缓存它们必须遵循任何你给定的信息，并基于这些信息来判断资源的新旧程度。在某些条件下，HTTP 允许缓存针对它们自己的规则使用过期的资源。通过指定该选项，可告诉缓存要严格按照你的规则来决策。

给提供服务的资源增加 Last-Modified 头信息也有助于缓存。浏览器在对该资源后续的请求中，就能指定 If-Modified-Since 头信息，这是一个时间戳。如果该资源在最后一次访问之后未被修改，服务器只返回 304 状态码（未修改）。浏览器仍然可以请求，但服务器却不一定在响应中返回该资源的内容，这可以节省网络时间和带宽：

```
# Request
GET /example.gif HTTP/1.1
Host:www.example.com
If-Modified-Since:Thu, 29 Apr 2010 12:09:05 GMT
```

```
# Response
HTTP/1.1 200 OK
Date: Thu, 20 March 2009 00:00:00 GMT
Server: Apache/1.3.3 (Unix)
Cache-Control: max-age=3600, must-revalidate
Expires: Fri, 30 Oct 1998 14:19:41 GMT
Last-Modified: Mon, 17 March 2009 00:00:00 GMT
Content-Length: 1040
Content-Type: text/html
```

Last-Modified 的替代方式是 ETag。比较 Etag 就像比较两个文件的哈希值——如果 ETag 值不一致，缓存就过期了，必须重新验证。它的工作原理和 Last-Modified 头信息一样。服务器将用 ETag 头信息将该值附加到资源响应中，客户端将用 If-None-Match 头信息检查：

```
# Request
GET /example.gif HTTP/1.1
Host:www.example.com
If-Modified-Since:Thu, 29 Apr 2010 12:09:05 GMT
If-None-Match:" 48ef9-14a1-4855efe32ba40"

# Response
HTTP/1.1 304 Not Modified
```

ETag 通常是使用服务器指定的一些属性来构建的——也就是说，两个独立的服务器对同样的资源生成的 ETag 可能不一样——随着服务器集群越来越普遍，这成为一个现实的问题。个人而言，我建议坚持使用 *Last-Modified* 并且关闭 ETag。

源码压缩（Minification）

JavaScript 源码压缩是从脚本文件中删除不必要的字符，它不改变功能。删除的字符包括空白、换行和注释。更好的压缩工具应该能够翻译 JavaScript，因此能安全地缩短变量和函数的名字，这样就进一步减少了字符。文件越小越好，因为在网络上传输的数据越少越好。

不仅仅可以对 JavaScript 文件进行压缩，样式表和 HTML 文件也可以被压缩。特别是样式表，通常包含了大量冗余的空白。压缩最好能在部署时再完成，因为开发时你不希望调试任何压缩过的代码。如果在生产环境中有一个错误，首先应该尝试在开发环境中复现——你会发现这样更容易调试错误。

源码压缩带来额外的好处是让代码晦涩难读。尽管这种代码总能被一些有动机的人通过某种方式还原出来，但对大部分的观察者来说这是一个障碍。

当前有很多源码压缩的工具，但我建议选择一个带有 JavaScript 引擎、能够翻译代码的工具。YUI Compressor (<http://developer.yahoo.com/yui/compressor>) 是我的最爱，因为它有很好的维护和支持。它是由 Yahoo! 工程师 Julien Lecomte 开发的，它的目标是——通过智能优化源代码，比 JSMIn (<http://www.crockford.com/javascript/jsmin.html>) 更好地缩减 JavaScript 文件大小。^{注1} 假设有如下函数：

```
function per(value, total) {
    return( (value / total) * 100 );
}
```

除了移除空白，YUI Compressor 会缩短局部变量来节省更多的字符：

```
function per(b,a){return((b/a)*100)};
```

YUI Compressor 实际上解析了 JavaScript 代码，它通常会替换变量——不会引入代码错误。但也不总是如此，偶尔它也不能正确揣摩你代码的本意，只能保持原样。最常见的原因是代码中使用了 `eval()` 或 `with()` 语句。如果检测到你使用了任何一种语句，它就不再应用变量替换了。另外，`eval()` 和 `with()` 都会导致性能问题——浏览器的 JIT 编译器有和压缩器一样的问题。我个人的建议是不要使用任何这类语句。

使用 YUI Compressor 最简单的方法就是下载其二进制文件 (<http://yuilibrary.com/downloads/#yuicompressor>)，它依赖 Java，在命令行中执行：

```
java -jar yuicompressor-x.y.z.jar foo.js | foo.min.js
```

尽管如此，也可以在部署时通过编程来实现。如果使用了类似 Sprockets (<http://getsprockets.org>) 或 Less (<http://lesscss.org>) 这样的库，它们便可以提供这类功能。否则，YUI Compressor 也提供了一些库的接口，例如 Sam Stephenson 的 Ruby-YUI-Compressor gem (<https://github.com/sstephenson/ruby-yui-compressor>) 或 Jammit 库 (<http://documentcloud.github.com/jammit>)。

Gzip 压缩

在 Web 上 Gzip 是最流行并且支持最广泛的压缩方式。它是由 GNU 项目组开发的，在 HTTP/1.1 中开始对其支持。Web 客户端通过在发送请求时增加 `Accept-Encoding` 头信息来标识自己支持的压缩方式：

```
Accept-Encoding: gzip, deflate
```

注 1：对于带有中文的 JavaScript 文件来说，仅做源码压缩是不够的，还需要对文件做 Unicode 转码，以保证压缩后的文件不会因为编码问题引入 bug。此外，由于每个项目中的 JS 文件往往比较多，也需要一些批量压缩工具来提高工作效率，这里推荐一个工具 TPACKER：<http://github.com/jayli/TPacker>。——译者注

如果 web 服务器看到该头信息, 并且支持列出的压缩方式, 它将会对响应结果进行压缩, 并通过 `Content-Encoding` 头信息标识其压缩方式:

`Content-Encoding: gzip`

然后浏览器才能正确地解码, 得到解码后的响应。显然, 压缩数据可以减少网络传送时间, 但这并没大范围地得以实现。Gzip 通常能减少响应 70% 的体积, 巨大的体积缩减极大地加速了网站的加载速度。

服务器通常已经配置了哪些文件类型应该被压缩。一条不错的经验就是压缩任何文本类型的响应, 例如 HTML、JSON、JavaScript 和样式表。如果文件已经被压缩, 例如图片和 PDF, 则不应该再用 Gzip 压缩了, 因为体积不会再减小了。

配置 Gzip 依赖于使用的 web 服务器, 但在 Apache 2.x 或更高的版本中, 你还需要 `mod_deflate` 模块 (http://httpd.apache.org/docs/2.0/mod/mod_deflate.html)。对于其他的服务器, 请查看各自的文档。

使用 CDN

内容分发网络 (或叫 CDN) 为你的站点提供静态资源内容服务, 以减少它们的加载时间。用户和 web 服务器之间的距离对加载时间有直接的影响。CDN 将你的内容部署在跨越多个地理位置的服务器上, 故当用户发起一个请求时, 可从就近的服务器得到响应资源 (理想情况是在同一个国家中)。Yahoo! 已经发现 CDN 可以减少终端用户 20% 或更多的响应时间 (<http://developer.yahoo.com/performance/rules.html#cdn>)。

根据你能承担的费用, 当前有很多提供 CDN 服务的公司, 如 Akamai Technologies (<http://www.akamai.com>)、Limelight Networks (<http://www.limelightnetworks.com>)、EdgeCast (<http://www.edgecast.com>) 和 Level 3 Communications (<http://www.level3.com>)。Amazon Web Services (<http://aws.amazon.com>) 最近发布了一个还能支付得起的服务, 叫作 Cloud Front (<http://aws.amazon.com/cloudfront>), 它和 S3 服务 (<http://aws.amazon.com/s3>) 紧密地捆绑在一起, 对初创的公司而言也许是个不错的选择。

Google 为很多流行的开源 JavaScript 库提供了一个免费的 CDN 和加载架构, 包括 jQuery 和 jQueryUI。使用 Google 的 CDN 的其中一个优点就是很多其他网站都在使用它, 这会让你要引用的 JavaScript 文件有更多的几率停留在用户浏览器的缓存之中。

首先, 检出可用的库的列表 (<http://code.google.com/apis/libraries/devguide.html>)。假如想要包含 jQuery 库 (<http://code.google.com/apis/libraries/devguide.html#jquery>), 既可以使用 Google 的 JavaScript 加载器, 也可以更简单地使用 script 标签:

```
<!-- minimized version of the jQuery library -->
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min.js" ></script>

<!-- minimized version of the jQuery UI library -->
<script src="//ajax.googleapis.com/ajax/libs/jqueryui/1.8.6/jquery-ui.min.js" >
</script>
```

你会发现在上面的例子中我们没有指定请求的协议，而只使用//。这个鲜为人知的技巧使得获取脚本文件时，可以使用和宿主页面一样的协议。换言之，如果页面通过安全的HTTPS加载，该脚本文件同样会使用HTTPS，从而避免所有的安全警告。没有协议的相对URL是合法的，与RTF规范兼容。更重要的是，它得到了全盘的支持。见鬼了！协议 - 相对URL甚至在IE 3.0中也能工作。

审查工具

139

有一些工具非常好，如果你想查看你的网站性能详情，它们能给你眼前一亮的惊喜，比如YSlow (<http://developer.yahoo.com/yslow>)。它是Firebug的扩展，由于Firebug是Firefox的扩展。因此所有这些软件安装完YSlow才能使用。安装好后，就可以用它来审查web页面了。该扩展通过一系列的检查来运行，包括缓存、源码压缩、Gzip压缩和CDN等。它将给出网站的评分等级，这依赖于检查过程中的耗费。然后，给出如何提高分数的建议。

Google Chrome和Safari也有审查工具，但是它们是内置在浏览器中的。如图10-1所示，在Chrome中只要简单地找到Web Inspector的Audits面板并点击Run就行了。能看到这些真的很棒：你的网站有哪些是可以改进的，进而如何提高它的性能。



图10-1：使用Web Inspector来监控网页性能

外部资源

Yahoo! 和 Google 都用了大量的研究、调查来分析 web 性能。很明显，提高网站的渲染速度是它们最感兴趣的，这对它们自己的服务和客户浏览互联网时的用户体验都有好处。事实上，Google 现在考虑其 Pagerank 算法的速度，它用来帮助判断网站的搜索查询等级。两个公司在改善性能上都有优秀的资源，可以在 Google (<http://code.google.com/speed/page-speed/docs/payload.html>) 和 Yahoo! (<http://code.google.com/speed/page-speed/docs/payload.html>) 的开发者网站上找到这些资源。

Spine类库

Spine (<http://maccman.github.com/spine>) 是一个轻量级 JavaScript 应用程序开发库，涉及到本书中的很多概念，如 MVC、事件和类。当说到轻量级，指的是体积轻便——整个库大约仅有 500 行的 JavaScript 代码，经过缩减和压缩后的代码大约只有 2K。但千万不要由此认为 Spine 类库很弱。Spine 将会让你在保证代码干净整洁和耦合松散的前提下，构建功能全面、丰富的 JavaScript 应用程序。

在写这本书的时候，我创建了 Spine，因为当时找不到一个客户端 MVC 框架，能满足我的需求。Spine 尝试恪守本书中建议的最佳实践，而事实上本书中的例子 Holla 就是用 Spine 打造的。

和基于 Widget 库，的如 Cappuccino (<http://cappuccino.org>) 和 SproutCore (<http://sproutcore.com>) 等不一样，Spine 本身不会决定如何向用户呈现数据。它的重点在于灵活和简单。Spine 只提供骨架，接下来按照你自己喜欢的方法处理——开发真正了不起的应用。

Spine 包含了一个可以继承的类库 `Spine.Class`、一个事件模块 `Spine.Events`、一个 ORM——`Spine.Model`，以及一个控制器类 `Spine.Controller`。你自己可以使用其他任何类库，像模板支持或 DOM 库等，用你最熟悉的就可以了。不仅如此，Spine 包含了对 jQuery 和 Zepto.js 库特别的支持，它们是对 Spine 的非常好的补充。

目前 Spine 的不足就是缺乏文档。但既然现在还是该库的初级阶段，文档肯定会得以改善的。到目前为止，本章应该是个不错的介绍，示例应用程序将提供进一步的解释。

设置

Spine 的设置很简单，从 Spine 项目代码仓库 (<http://github.com/maccman/spine>) 下载文件，引用到页面中即可（Spine 没有其他的依赖关系）：

```
<script src="spine.js" type="text/javascript" charset="utf-8"></script>
```

Spine 的命名空间完全在 `Spine` 变量之后，故它和其他变量都不会冲突。因此，可以安全地包含 jQuery、Zepto 或 Prototype 等类库，而不会产生任何兼容性问题。

类

Spine 中几乎每一个对象都封装在类里面。但是，Spine 的类是使用 `Object.create()` 和纯原型继承的方式来构造的（在第三章中有相应介绍），它和大多数类抽象的构造方式是不一样的。

为了创建一个新的类，调用 `Spine.Class.create([instanceProperties, classProperties])`，传入一个可选的由实例和类属性组成的对象：

```
var User = Spine.Class.create({  
  name: "Caroline"  
});
```

如上例所示，`User` 的实例现在有一个默认的属性 `name`。背后之意——`create()` 创建一个新对象，设置它的 `prototype` 为 `Spine.Class`——也就是说，`User` 继承自它。如果想要创建子类，只要简单地在父类上调用 `create()` 即可：

```
var Friend = User.create();
```

`Friend` 现在是 `User` 的子类，将继承它所有的属性：

```
assertEqual( Friend.prototype.name, "Caroline" );
```

实例化

因为使用了纯原型对象，并且用继承代替了构造函数，所以不能使用 `new` 关键词来生成实例。相反，Spine 使用了 `init()` 函数：

```
var user = User.init();  
assertEqual( user.name, "Caroline" );  
  
user.name = "Trish";  
assertEqual( user.name, "Trish" );
```

任何传递给 `init()` 的参数都会传入实例的初始化函数 `init()` :

```
var User = Spine.Class.create({
  init: function(name){
    this.name = name;
  }
});

var user = User.init("Martina");
assertEqual( user.name, "Martina" );
```

类扩展

就像在创建时设置类和实例的属性一样，你可以使用 `include()` 和 `extend()` 传入一个对象直接量：

```
User.include({
  // Instance properties
});

User.extend({
  // Class properties
});
```

`include()` 和 `extend()` 为模块铺平了道路，它们是可复用的代码片段，可以多次被包含：

```
var ORM = {
  extended: function(){
    // 当继承的时候调用这里的逻辑
    // this === User
  },
  find: function(){ /* ... */ },
  first: function(){ /* ... */ }
};

User.extend( ORM );
```

当模块被包含或扩展时你会接收到一个回调。在上面的例子中，当调用 `User.extend()` 函数时，`extended` 函数会被调用，上下文是 `User`。同样，如果模块有 `included` 属性，当模块被包含到一个类中时它也会被调用。

因为使用了基于原型的继承，任何添加到类中的属性都能够在运行时被动态反射到子类中：

```
var Friend = User.create();

User.include({
  email: "info@eribium.org"
```

```
});

assertEqual( Friend.init().email, "info@eribium.org" );
```

在子类中属性可以被覆盖而不影响父类。但修改子类中的对象，如数组，则会影响整个继承树。如果你想让一个对象惟一派生自一个类或实例，需要在类或实例的首次初始化时创建它。可以在 `created()` 函数中做这个事情，在类首次设置或实例化时，Spine 会调用它：

144

```
// 我们想让一组记录来表示这个类
var User = Spine.Class.create({
  // 实例化时调用
  init: function(){
    this.attributes = {};
  }
},{
  // 创建类的时候调用
  created: function(){
    this.records = [];
  }
});
```

上下文

更改上下文在 JavaScript 编程中是非常普遍的，所以 `Spine.Class` 包含了一些工具方法来控制作用域。为了使问题清晰，来看一个例子：

```
var Controller = Spine.Class.create({
  init: function(){
    // 绑定事件监听
    $("#destroy").click(this.destroy);
  },

  destroy: function(){
    // 这个析构函数的调用使用了错误的上下文
    // 因此使用 this 会有问题
    // 下面这个断言执行失败
    assertEqual( this, Controller.fn );
  }
});
```

在上面的例子中，当事件发生时，`destroy()` 函数会被调用，上下文是元素 `#destroy`，而不是 `Controller`。为了正确处理这种情况，可以将代理上下文强制指定为你所需要的那个。Spine 提供了 `proxy()` 函数来做到这件事情：

```
var Controller = Spine.Class.create({
  init: function(){
```

```

        $("#destroy").click(this.proxy(this.destroy));
    },

    destroy: function(){ }
});

```

如果发现要多次地使用代理，你需要重写程序，使其总是包含这个代理。Spine 包含了一个 `proxyAll()` 函数来应对这种情况：

```

var Controller = Spine.Class.create({
  init: function(){
    this.proxyAll("destroy", "render")
    $("#destroy").click(this.destroy);
  },

  // 现在调用函数时总是会使用正确的上下文
  destroy: function(){ },
  render: function(){ }
});

```

145

`proxyAll()` 可以接收多个函数名称，但被调用时它会重写这些函数，从而使用当前的作用域代理这些函数。这就保证了 `destroy()` 或 `render()` 总是在局部作用域下执行代码。

事件

事件在 Spine 中很关键，在内部它们也经常使用。Spine 的事件功能包含在 `Spine.Events` 模块中，在任何需要的地方都可以包含它。例如，给一个 Spine 中的类添加事件支持：

```

var User = Spine.Class.create();
User.extend(Spine.Events);

```

`Spine.Events` 提供了三个处理事件的函数：

- `bind(eventName, callback)`
- `trigger(eventName, [*data])`
- `unbind(eventName, [callback])`

如果你用过 jQuery 的事件 API，这看起来很眼熟。例如，给我们的 `User` 类绑定和触发事件：

```

User.bind("create", function(){ /* ... */ });
User.trigger("create");

```

将多个事件绑定到一个回调函数上，只要使用空格隔开即可：

```

User.bind("create update", function(){ /* ... */ });

```

trigger() 接收一个事件名称，并传递可选的参数给事件的回调函数：

```
User.bind("countChange", function(count){
    // 调用 trigger 方法时将 `count` 传入
    assertEquals(count, 5);
});

User.trigger("countChange", 5);
```

最常见的使用 Spine 事件的方式是和数据绑定在一起，用视图在应用程序的模型上挂接钩子。在后面“构建联系人管理”小节中将详细地对其进行介绍。

模型

如果看过一眼 Spine 的源代码 (<https://github.com/maccman/spine/blob/master/spine.js>)，就会注意到其大量的篇幅是在处理模型，换言之——模型是任何 MVC 应用程序的核心部分。模型处理应用程序数据的排序和操作，通过提供完整的 ORM Spine 来简化它。

可以通过生成一个新的模型来使用 Spine.Model.setup(name, attrs)，而不是使用 create() 函数，它已经被保留了。可以给 setup() 传递模型名称和一个属性数组：

```
// 创建任务模型
var Task = Spine.Model.setup("Task", ["name", "done"]);
```

使用 include() 和 extend() 来添加实例和类属性：

```
Task.extend({
    // 返回所有完成的任务
    done: function(){ /* ... */ }
});

Task.include({
    // 默认名字
    name: "Empty...",
    done: false,

    toggle: function(){
        this.done = !this.done;
    }
});
```

当实例化一条记录时，可以传递一个包含初始属性的可选对象：

```
var task = Task.init({name: "Walk the dog"});
assertEquals( task.name, "Walk the dog" );
```

设置和获取属性就像普通对象的设置和获取属性一样。另外，`attributes()` 函数返回一个包含记录所有属性的对象直接量：

```
var task = Task.init();
task.name = "Read the paper";
assertEqual( task.attributes(), {name: "Read the paper"} );
```

保存新的或已存在的记录只要简单地调用 `save()` 函数。保存记录时，如果其不存在就会生成一个 ID，然后记录就会持久化存储到本地内存中：

```
var task = Task.init({name: "Finish book"});
task.save();
task.id //=> "44E1DB33-2455-4728-AEA2-ECBD724B5E7B"
```

只要传递进去记录的 ID，就可以通过使用模型的 `find()` 函数来获取记录：

```
var task = Task.find("44E1DB33-2455-4728-AEA2-ECBD724B5E7B");
assertEqual( task.name, "Finish book" );
```

如果对应的 ID 没有记录存在，将会产生一个异常。可以使用 `exists()` 函数来检查一条记录是否存在，而不用担心产生异常：

◀ 147

```
var taskExists = Task.exists("44E1DB33-2455-4728-AEA2-ECBD724B5E7B");
assert( taskExists );
```

使用 `destroy()` 函数可以从本地缓存中删除一条记录：

```
var task = Task.create({name: "Thanks for all the fish"});

assert( task.exists() );
task.destroy();
assertEqual( task.exists(), false );
```

获取记录

使用 ID 来获取记录仅仅是其中一种方式。通常，它在遍历所有的记录或返回要过滤的子集时非常有用。通过使用 `all()`、`select()` 和 `each()` 等函数，Spine 可以让你轻松做到这些事情：

```
// 返回所有任务
Task.all(); //=> [Object]

// 通过带回 done 属性的求反值来返回所有的任务
var pending = Task.select(function(task){ return !task.done });

// 每个任务都调用一个回调
Task.each(function(task){ /* ... */ });
```

另外，Spine 还提供了一些帮助函数，可通过属性找到记录：

```
// 通过给定的属性值找到第一个任务
Task.findByAttribute(name, value);    //=> Object

// 通过给定的属性值找到所有的任务
Task.findAllByAttribute(name, value); //=> [Object]
```

模型事件

可以在模型上绑定事件，当记录改变时触发回调：

```
Task.bind("save", function(record){
  console.log(record.name, "was saved!");
});
```

如果涉及到一条记录，它会被传递到事件回调函数中。还可以为每条记录在模型上绑定一个监听来处理全局的回调，或者也可以为特定的记录绑定事件：

```
Task.first().bind("save", function(){
  console.log(this.name, "was saved!");
});

Task.first().updateAttributes({name: "Tea with the Queen"});
```

148 ▶ 很显然，你可以使用 `trigger()` 来创建自定义的事件，但下面这些事件已经默认存在了：

save

记录被保存（创建或更新）

update

记录被更新

create

记录被创建

destroy

记录被销毁

change

发生任一以上事件，记录被创建、更新或销毁

refresh

所有记录被删除和替换

error

校验无效

在创建应用程序时会发现模型事件非常关键，特别是在模型和视图绑定时。

校验

校验是最简单的方法实现的——通过覆盖模型实例的 `validate()` 函数。只要记录被保存, `validate()` 函数就会被调用。只要 `validate()` 返回一些值, 就说明校验失败了。否则, 记录将被顺利地继续保存, 永久驻留在本地内存中：

```
Task.include({
  validate: function(){
    if ( !this.name ) return "Name required";
  }
});
```

如果校验失败, 应该返回一个字符串, 这个字符串用来解释异常原因。可以将此消息通知给用户, 告诉他们什么地方出错了, 以及如何改正：

```
Task.bind("error", function(record, msg){
  // 最简单的报错
  alert("Task didn't save: " + msg);
});
```

只要校验出错, 模型错误事件就会被调用。回调函数传递进来的是出错的记录和指示错误的消息。

持久化

149

Spine 的记录总是持久化存储在内存中的, 但也有可能将其存储到后台, 例如 HTML5 的 Local Storage (本地存储) 或 Ajax。

使用本地存储非常简单。只要包含 *spine.model.local.js* 这个 JavaScript 文件即可, 然后用 `Spine.Model.Local` 来扩展你的模型：

```
// 使用本地存储来保存
Task.extend(Spine.Model.Local);
Task.fetch();
```

记录不会从浏览器的本地存储中自动获取, 因此需要调用 `fetch()`, 用已存在的数据来填充模型。通常, 在应用程序全部初始化后会做这件事情。一旦模型填充了新的数据, 就会触发 *refresh* 事件：

```
Task.bind("refresh", function(){
  // 新任务！
  renderTemplate(Task.all());
});
```

类似地，也可以使用 Ajax 来实现持久化存储。只要包含 *spine.model.ajax.js* 这个 JavaScript 即可，然后用 *Spine.Model.Ajax* 来扩展你的模型：

```
// 保存至服务器
Task.extend(Spine.Model.Ajax);
```

默认情况下，Spine 会检测模型名称，并使用一些通用的转换方法来生成一个 URL。因此，如上例所示，Task 模型的 URL 将会是“/tasks”。在类中提供你自己的 URL 属性可以覆盖默认的行为：

```
// 添加自定义 URL
Task.extend({
  url: "/tasks"
});

// 从服务器获取新任务
Task.fetch();
```

一旦调用了 *Task.fetch()*，Spine 会发起一个 Ajax 的 GET 请求到 /tasks，它期望得到一个 JSON 的响应，里面应该是一个包含一些 task（任务）的数组。如果服务器成功返回响应，这些记录将会被加载进来，并触发 *refresh* 事件。

只要你创建、更新或销毁一个记录，Spine 就会发起 Ajax 请求到服务器来同步两边的数据。我们的程序期望服务器提供 REST 服务，这样可以做到前、后端的无缝衔接，不过也可以覆盖这一请求，以便自定义你的请求类型。Spine 期望存在这些对应终端：

```
read    →  GET    /collection
create  →  POST   /collection
update  →  PUT    /collection/id
destroy →  DELETE /collection/id
```

150 客户端创建了一条记录以后，Spine 将会发送一个 HTTP POST 请求到服务器，提交以 JSON 形式表示的记录。例如，创建一个名称为“Bug eggs”的 Task，将会发送到服务器的请求如下：

```
POST /tasks HTTP/1.0
Host: localhost:3000
Origin: http://localhost:3000
Content-Length: 66
Content-Type: application/json

{"id": "44E1DB33-2455-4728-AEA2-ECBD724B5E7B", "name": "Buy eggs"}
```

同样地，销毁一条记录将会发送一个 DELETE 请求到服务器，更新一条记录将会发送一个 PUT 请求。对于 PUT 和 DELETE 请求，记录的 ID 的值是包含在 URL 中的：

```
PUT /tasks/44E1DB33-2455-4728-AEA2-ECBD724B5E7B HTTP/1.0
Host: localhost:3000
Origin: http://localhost:3000
Content-Length: 71
Content-Type: application/json
```

```
{"id": "44E1DB33-2455-4728-AEA2-ECBD724B5E7B", "name": "Buy more eggs"}
```

Spine 的 Ajax 同步和其他大多数库的有一个区别——它首先将记录保存到客户端，然后才发送到服务器，因此客户端不用等待响应。这意味着客户端完全和服务器解耦了——也就是说，为了实现功能并不一定需要有服务器的存在。

和服务器解耦有三个优势。首先，接口速度很快，而且非阻塞，因此用户与应用程序交互不用等待。其次，简化了代码——记录可能会显示在程序界面中，但是在服务器的响应到达客户端之前内容不可编辑，这就省去了相应逻辑的设计。最后，如果真的需要添加离线支持，可以很容易地做到。

服务端校验是怎么回事呢？Spine 假设你将在客户端进行所有必要的校验。服务器只在发生一个异常（代码有问题）时返回一个错误，而这个异常只能在异常环境中发生。

当服务器返回一个失败的响应时，模型将触发一个 *ajaxError* 事件，包含该记录、一个 XMLHttpRequest 对象、Ajax 设置属性和应当抛出的异常对象：

```
Task.bind("ajaxError", function(record, xhr, settings, error){
  // Invalid response
});
```

控制器

151

控制器是 Spine 最后一个组件，它将应用程序剩下的部分粘合在一起。控制器通常就是添加事件句柄到 DOM 元素、模型、渲染模板，并保持视图和模型的同步。创建一个 Spine 控制器，需要调用 `create()` 来派生出 `Spine.Controller` 的子类：

```
jQuery(function(){
  window.Tasks = Spine.Controller.create({
    // 控制器属性
  });
});
```

建议在页面的其他部分都加载进来之后再加载控制器，从而无须处理各种不同页面状态的问题。在 Spine 所有的例子中，会发现每个控制器都是在 `jQuery()` 调用的里面。这保

证了控制器是在文档加载完成以后才被创建的。

在 Spine 中，控制器的名字采用驼峰命名法，用英文单词的复数形式表示，这里的英文单词指代相关的模型。大多数控制器只有实例属性，仅在实例化后才可被使用。实例化控制器就像实例化任何类一样，需调用 `init()`：

```
var tasks = Tasks.init();
```

控制器总是会 and DOM 元素相关联，通过 `el` 属性来访问元素。可以在实例化时传递进去，这是可选的；否则，控制器自身生成一个默认的 `div` 元素：

```
var tasks = Tasks.init({el: $("#tasks")});  
assertEqual( tasks.el.attr("id"), "tasks" );
```

该元素在内部可以用来添加模板和渲染视图：

```
window.Tasks = Spine.Controller.create({  
  init: function(){  
    this.el.html("Some rendered text");  
  }  
});  
  
var tasks = Tasks.init();  
$("body").append(tasks.el);
```

事实上，任何传递给 `init()` 函数的参数都会在控制器上设置属性。例如：

```
var tasks = Tasks.init({item: Task.first()});  
assertEqual( Task.first(), tasks.item );
```

152 代理

在前面的例子中你会发现所有事件的回调函数都用 `this.proxy()` 包起来，以保证在正确的上下文中执行。由于这是一个很常用的模式，Spine 提供了一个快捷方式：`proxied`。在控制器上只要简单地添加一个 `proxied` 属性，对应的值是包含函数名的数组，这些函数都将会在控制器的上下文中执行：

```
// 和使用 proxyAll 等价  
var Tasks = Spine.Controller.create({  
  proxied: ["render", "addAll"],  
  
  render: function(){ /* ... */ },  
  addAll: function(){ /* ... */  
});
```

现在可以将类似 `render()` 这些回调函数传递给事件监听了，而不用担心它们的执行上下文。这些函数总是会在正确的上下文中得到执行。

元素

在应用程序内部以本地属性的方式来访问元素是非常有用的。Spine 提供了一个快捷方式：`elements`。只要在控制器上添加 `elements` 属性，它的值是一个对象，包含选择器的名字。在下面的例子中，`this.input` 就是使用 `form input[type=text]` 选择得到的元素。所有的选择器都是在控制器的 `el` 元素之下执行选取的，而并非整个页面：

```
// input 实例变量
var Tasks = Spine.Controller.create({
  elements: {
    "form input[type=text]": "input"
  },

  init: function(){
    // this.input 指向表单的输入框
    console.log( this.input.val() );
  }
});
```

但请记住，如果替换了控制器的 `el` 元素，你需要调用 `refreshElements()` 来刷新所有的元素引用。

委托事件

Spine 的事件属性提供了一种简单的批量添加事件监听的方法。该场景背后的工作原理是，Spine 利用了事件冒泡，所以只要在控制器的元素（`el`）上绑定一个事件监听。像事件属性一样，所有的事件委托的作用域也是 `el`。

事件属性的形式是：`{"eventName selector": "callback"}`。选择器是可选的，如果没有提供事件就会直接绑定在元素 `el` 上。否则，事件就被委托（[http://api.jquery.com/](http://api.jquery.com/delegate)`delegate`）：当在一个匹配选择器的子元素上发生该类型的事件时就会触发它。这是动态的，所以元素 `el` 里面的内容是否有变化并不重要：

153

```
var Tasks = Spine.Controller.create({
  events: {
    "keydown form input[type=text]": "keydown"
  },

  keydown: function(e){ /* ... */ }
});
```

在上面的例子中，匹配选择器的 `input` 元素在收到 `keydown` 事件后，控制器的 `keydown` 回调函数就会执行。Spine 保证了该函数在正确的上下文中执行，所以在这里无须担心，也不用代理事件回调函数。

事件对象将传递到回调函数中，它非常有用，因为在本例中我们需要知道哪个键被按下了。另外，上面讨论到的元素可以从事件的 `target` 属性中获取。

控制器事件

类似事件委托，Spine 的控制器支持自定义事件。默认情况下，控制器扩展自 `Spine.Events`，这意味着它们具有必要的事件功能，如 `bind()` 和 `trigger()`。利用这一点可以保证各个控制器之间的解耦，或者良好地组织控制器的内部代码：

```
var Sidebar = Spine.Controller.create({
  events: {
    "click [data-name]": this.click
  },

  init: function(){
    this.bind("change", this.change);
  },

  change: function(name){ /* ... */ },

  click: function(e){
    this.trigger("change", $(e.target).attr("data-name"));
  }

  // ...
});

var sidebar = Sidebar.init({el: $("#sidebar")});
sidebar.bind("change", function(name){
  console.log("Sidebar changed:", name);
})
```

154 在上面的例子中，其他的控制器也能绑定 `Sidebar` 的 `change` 事件或触发它。就像在第二章展示的一样，自定义事件就算在外部从不被使用，但在程序的内部组织中极为常用。

全局事件

Spine 可以让你在全局层面绑定和触发事件。这是一种发布和订阅的形式，即便控制器之间并不知晓对方的存在，也能让它们相互通信，这保证了它们互相之间的解耦。这是通过一个全局对象 `Spine.App` 来实现的，在它上面什么都可以绑定，也可以触发事件：

```
var Sidebar = Spine.Controller.create({
  proxied: ["change"],

  init: function(){
```

```

    this.App.bind("change", this.change);
  },

  change: function(name){ /* ... */ }
});

```

Spine 的控制器将 `Spine.App` 的别名简称为 `this.App`，以简化输入。在上面的例子中可以看到，`Sidebar` 控制器绑定了全局的 `change` 事件。然后，其他控制器或脚本就可以触发该事件，将必要的数据传递进来：

```
Spine.App.trigger("change", "messages");
```

渲染模式

既然已经介绍了控制器中所有主要的可用选项，那下面就来看一下典型的用例。

渲染模式是将模型和视图绑定在一起的非常有用的方法。控制器实例化以后，添加一个事件监听到对应的模型，在模型刷新或变化时调用回调函数。该回调函数将更新元素 `el`，而通常就是使用渲染模板来替换 `el` 中的内容：

```

var Tasks = Spine.Controller.create({
  init: function(){
    Task.bind("refresh change", this.proxy(this.render));
  },

  template: function(items){
    return($("#tasksTemplate").tmpl(items));
  },

  render: function(){
    this.el.html(this.template(Task.all()));
  }
});

```

这个绑定数据的方法很简单，但比较生硬，只要有一条记录发生变化，它就会更新所有的元素。在小而不复杂的列表应用中，这是没有问题的，但随后会发现我们需要对独立的元素有更多的控制，例如需要对其中的项目添加事件处理句柄。就这样元素模式应运而生。

◀ 155

元素模式

本质上元素模式和渲染模式提供的功能是一样的，但有了更多的控制。它由两个控制器构成：一个用来控制多个项目的一个集合，另一个用来处理单个的项目。让我们深入代码来一窥究竟：

```

var TasksItem = Spine.Controller.create({
  // 将点击事件委托给本地事件处理程序
  events: {
    "click": "click"
  },

  // 保证函数的执行拥有正确的上下文
  proxied: ["render", "remove"],

  // 将事件绑定至记录中
  init: function(){
    this.item.bind("update", this.render);
    this.item.bind("destroy", this.remove);
  },

  // 渲染一个元素
  render: function(item){
    if (item) this.item = item;

    this.el.html(this.template(this.item));
    return this;
  },

  // 使用模板, 这里使用了 jQuery.tmpl.js
  template: function(items){
    return($("#tasksTemplate").tmpl(items));
  },

  // 当销毁一个元素时调用
  remove: function(){
    this.el.remove();
  },

  // 我们很好的控制了事件, 同时也很轻松的就可以访问记录
  click: function(){ /* ... */ }
});

var Tasks = Spine.Controller.create({
  proxied: ["addAll", "addOne"],
  init: function(){
    Task.bind("refresh", this.addAll);
    Task.bind("create", this.addOne);
  },

  addOne: function(item){
    var task = TasksItem.init({item: item});
    this.el.append(task.render().el);
  },

  addAll: function(){

```

```
        Task.each(this.addOne);
    }
});
```

在上面的例子中，`Tasks` 承担的责任是当记录初始创建时将其添加进去，`TasksItem` 承担的责任是处理更新、销毁事件，并且在必要时渲染记录。虽然它比之前的渲染模式更加复杂，但也有一些优势。

首先，它更高效了——任何单个元素变化时，该列表不需要重绘。更进一步说，我们对单独的项目有了更多的控制。像上例中的点击回调一样，我们能够在单个项目的层面来设置事件处理句柄及处理渲染。

构建联系人管理应用

下面就让我们使用对 `Spine` 的 API 掌握的知识做一些实际的应用，比如构建一个联系人管理应用。我们希望提供给用户读取、创建、更新和删除联系人的功能，当然也包括搜索。

图 11-1 展示了完成的结果，从中已经可以看到这个想法所创建出来的东西。

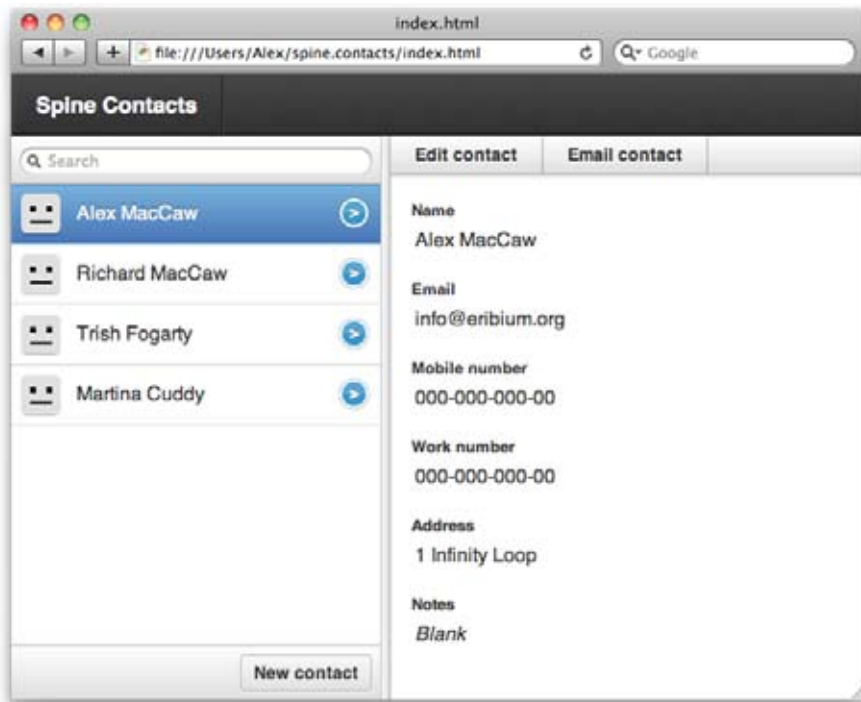


图11-1：Spine应用程序例子中的联系人列表

该联系人管理应用是 Spine 开源类库所提供的例子。你可以跟随下面的学习向导，也可以从项目的代码仓库（<http://github.com/maccman/spine.contacts>）中下载全部的代码。

参照图 11-1，联系人管理应用主要有两个单元：侧边栏和联系人视图。二者分别构成了两个独立的控制器：Sidebar 和 Contacts。该应用只有一个模型：Contact。在展开讨论每个单独的组件之前，先来看看初始化页面的结构：

```
<div id="sidebar">
  <ul class="items">
  </ul>

  <footer>
    <button>New contact</button>
  </footer>
</div>

<div class="vdivide"></div>
<div id="contacts">
  <div class="show">
    <ul class="options">
      <li class="optEdit">Edit contact</li>
      <li class="optEmail">Email contact</li>
    </ul>
    <div class="content"></div>
  </div>

  <div class="edit">
    <ul class="options">
      <li class="optSave default">Save contact</li>
      <li class="optDestroy">Delete contact</li>
    </ul>
    <div class="content"></div>
  </div>
</div>
```

结构中有两个独立的单元，一个是 #sidebar 的 div，另一个是 #contacts 的 div。我们的应用将会用联系人的姓名填充 .items 的列表，并选中一个当前的联系人，在 #contacts 中显示它的相关信息。同时，监听 .optEmail 和 .optSave 上面的点击事件，根据需要切换显示或编辑的状态。最后，监听 .optDestroy 上面的点击事件，销毁当前联系人并选中其他的联系人。

联系人模型

我们只用极少量代码就完成了联系人模型。Contact 有三个属性：first_name、last_name 和 email。同时还提供了一个方便的函数，用来获得全名，它在模板中很有用：

```
// 创建一个模型
var Contact = Spine.Model.setup("Contact", ["first_name", "last_name", "email"]);

// 页面重新载入后也能保持原有的模型
Contact.extend(Spine.Model.Local);

// 添加一些实例方法
Contact.include({
  fullName: function(){
    if ( !this.first_name && !this.last_name ) return;
    return(this.first_name + " " + this.last_name);
  }
});
```

请注意，`Spine.Model.Local` 扩展了该模型。这保证了这些记录能保存到浏览器的本地存储中，使得下次页面加载后数据仍然是可用的。

侧边栏控制器

现在来看一下 `Sidebar` 控制器，它承担的责任是在列表中显示联系人并跟踪当前选中的是哪个联系人。一旦联系人有变化，`Sidebar` 控制器必须完成自我更新来反映这些变化。另外，侧边栏有一个“New contact”按钮，它将监听点击事件，处理创建空联系人的任务。

下面就是该控制器所实现的全部功能。首先这里的代码就是最主要的部分——如果对 `Spine` 还不熟悉——它包含了丰富的注释，所以仔细研究以后还是应该能理解的：

```
jQuery(function($){

  window.Sidebar = Spine.Controller.create({
    // 创建实例变量
    // this.items ==> <ul></ul>
    elements: {
      ".items": "items"
    },

    // 绑定事件委托
    events: {
      "click button": "create"
    },
    // 确保函数调用使用了正确的作用域
    // 正如在事件回调中使用它们一样
    proxied: ["render"],

    // 渲染模板
    template: function(items){
      return($("#contactsTemplate").tmpl(items));
    },
```

```

init: function(){
    this.list = Spine.List.init({
        el: this.items,
        template: this.template
    });

    // 当列表的当前项发生改变时，显示这个联系人
    this.list.bind("change", this.proxy(function(item){
        this.App.trigger("show:contact", item);
    }));

    // 当当前的联系人发生改变时，比如创建了新的联系人
    // 更改这个列表的当前选中项
    this.App.bind("show:contact edit:contact", this.list.change);

    // 当联系人发生改变时就渲染联系人
    Contact.bind("refresh change", this.render);
},

render: function(){
    var items = Contact.all();
    this.list.render(items);
},

// 当‘创建’按钮被点击时调用
create: function(){
    var item = Contact.create();
    this.App.trigger("edit:contact", item);
}
});

});

```

你会发现该控制器的 `init()` 函数使用了一个叫 `Spine.List` 的类，之前还没有对它进行解释。`Spine.List` 是一个实用工具控制器，生成记录列表时它非常有用。另外，`Spine.List` 将会跟踪当前选中的项目，并在用户选中一个不同的项目时用一个 `change` 事件通知事件的监听者。

当联系人有改变或刷新时该列表会全部刷新。这让该例子简单、漂亮，但如果将来它暴露了某些性能问题，我们还需要对此做进一步改进。

在 `template()` 中的 `#contactsTemplate` 是一个 `script` 元素，它包含了单个列表项目的联系人模板：

160

```

<script type="text/x-jquery-tmpl" id="contactsTemplate">
    <li class="item">

```

```

    {{if fullName()}}
    <span>${fullName()}</span>
    {{else}}
    <span>No Name</span>
    {{/if}}
  </li>
</script>

```

我们将 jQuery.tmpl (<http://api.jquery.com/jquery.tmpl/>) 用作模板，如果读过第五章，你可能已经很熟悉了。Spine.List 就是使用这段代码所示的模板来渲染每一条记录，如果某条记录是当前选中的，则会在此记录对应的 上增加一个名为 current 的 class 值。

联系人控制器

Sidebar 控制器负责显示联系人列表，允许用户选中单个联系人。但如何显示当前选中的联系人呢？这就是 Contacts 控制器的来由：

```

jQuery(function($){

  window.Contacts = Spine.Controller.create({
    // 填充内部元素的属性
    elements: {
      ".show": "showEl",
      ".show .content": "showContent",
      ".edit": "editEl"
    },

    proxied: ["render", "show"],

    init: function(){
      // 内部视图显示联系人
      this.show();

      // 当联系人发生改变时重新渲染视图
      Contact.bind("change", this.render);

      // 绑定至全局事件
      this.App.bind("show:contact", this.show);
    },

    change: function(item){
      this.current = item;
      this.render();
    },

    render: function(){
      this.showContent.html($("#contactTemplate").tmpl(this.current));
    },
  });

```

```

        show: function(item){
            if (item && item.model) this.change(item);

            this.showEl.show();
            this.editEl.hide();
        }
    });

```

一旦侧边栏中有一个新的联系人被选中了，全局事件 `show:contact` 就会被触发。在 `Contacts` 中绑定该事件，它会执行 `show()` 函数，得到并解析新选中的联系人。然后渲染 `showContent` 的 `div`，用当前选中的记录来代替。

可以看到，我们引用了 `#contactTemplate` 模板，它用来显示当前联系人。接下来，将该模板添加到页面中：

```

<script type="text/x-jquery-tmpl" id="contactTemplate">
    <label>
        <span>Name</span>
        ${first_name} ${last_name}
    </label>

    <label>
        <span>Email</span>
        {{if email}}
            ${email}
        {{else}}
            <div class="empty">Blank</div>
        {{/if}}
    </label>
</script>

```

我们现在已经有显示联系人的功能了，但编辑和销毁的功能呢？为了做到这一点，让我们重写一下 `Contacts` 控制代码。主要区别就是应用程序在两种状态下的切换，当点击 `.optEdit` 和 `.optSave` 元素时显示和编辑。同时添加一个增加联系人的模板：`#editContactTemplate`。当保存记录时，读取正在编辑的表单输入，并且更新记录的属性：

```

jQuery(function($){

    window.Contacts = Spine.Controller.create({
        // 填充内部元素属性
        elements: {
            ".show": "showEl",
            ".edit": "editEl",
            ".show .content": "showContent",
            ".edit .content": "editContent"
        },

```

```

// 委托事件
events: {
  "click .optEdit": "edit",
  "click .optDestroy": "destroy",
  "click .optSave": "save"
},

proxied: ["render", "show", "edit"],

init: function(){
  this.show();
  Contact.bind("change", this.render);
  this.App.bind("show:contact", this.show);
  this.App.bind("edit:contact", this.edit);
},

change: function(item){
  this.current = item;
  this.render();
},

render: function(){
  this.showContent.html($("#contactTemplate").tpl(this.current));
  this.editContent.html($("#editContactTemplate").tpl(this.current));
},

show: function(item){
  if (item && item.model) this.change(item);

  this.showEl.show();
  this.editEl.hide();
},

// 当点击‘编辑’按钮时调用
edit: function(item){
  if (item && item.model) this.change(item);

  this.showEl.hide();
  this.editEl.show();
},

// 当点击‘删除’按钮时调用
destroy: function(){
  this.current.destroy();
},

// 当点击‘保存’按钮时调用
save: function(){
  var atts = this.editEl.serializeForm();

```

```

        this.current.updateAttributes(atts);
        this.show();
    }
});

});

```

如上面提到的，我们正在使用一个名叫 `#editContactTemplate` 的新模板。我们需要将其添加到页面中，这样才能引用成功。本质上来说，`#editContactTemplate` 和 `#contactTemplate` 非常相似，不同的是它使用 `input` 元素来显示记录数据：

163 

```

<script type="text/x-jquery-tmpl" id="editContactTemplate">
  <label>
    <span>First name</span>
    <input type="text" name="first_name" value="${first_name}" autofocus>
  </label>

  <label>
    <span>Last name</span>
    <input type="text" name="last_name" value="${last_name}">
  </label>

  <label>
    <span>Email</span>
    <input type="text" name="email" value="${email}">
  </label>
</script>

```

应用程序控制器

就这样，我们有了两个控制器 `Sidebar` 和 `Contacts`——来处理选中、显示和编辑 `Contact` 记录。到目前为止，最后需要的是一个 `App` 控制器，用来实例化每个控制器，传递给它们所需要的页面元素：

```

jQuery(function($){
  window.App = Spine.Controller.create({
    el: $("body"),

    elements: {
      "#sidebar": "sidebarEl",
      "#contacts": "contactsEl"
    },

    init: function(){
      this.sidebar = Sidebar.init({el: this.sidebarEl});
      this.contact = Contacts.init({el: this.contactsEl});
    }
  });
});

```

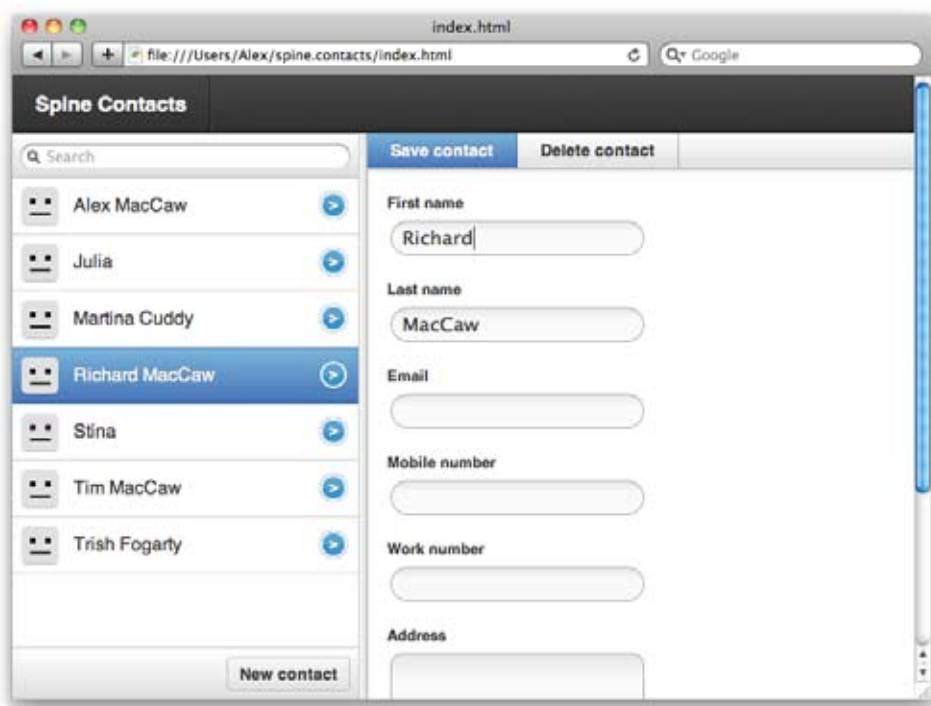
```

        // 从本地存储中获取联系人
        Contact.fetch();
    }
    }).init();
});

```

请注意，我们在 App 控制器创建后立即调用了它的 `init()` 函数。同时，在 `Contact` 模型上也调用了 `fetch()` 函数，从本地存储中获取所有联系人信息。

所以，这就是联系人的全部功能和代码。两个主要的控制器（`Sidebar` 和 `Contacts`），一个模型（`Contact`），两个视图。想要查看最终的产品，从仓库中检出代码（<http://github.com/maccman/spine.contacts>），对照图 11-2。



164

图11-2：Spine应用程序例子中的编辑联系人

Backbone类库

Backbone 是构建 JavaScript 应用程序的一个优秀的类库。它的优美之处在于其简洁。这是一个轻量类库，它覆盖了所有基础的功能，同时提供了最大的灵活性。与本书的其他内容一样，MVC 是我们讨论的重点，这也是贯穿 Backbone 核心的模式。该类库提供了模型、控制器和视图——构建应用程序的骨架。

Backbone 和其他框架，如 SproutCore 或者 Cappuccino 有什么不同呢？这么说吧，主要的不同就是 Backbone 轻量的特质。SproutCore 和 Cappuccino 提供了厚重的 UI widget 和巨大的核心库，并且它们决定了你的 HTML 结构。在打包压缩以后，这两个库的大小都是数百 KB，加载到浏览器中时，JavaScript、CSS 和图片有好几兆字节。对比而言，Backbone 只有 4KB，只单纯地提供了模型、事件、集合、视图、控制器和持久化等核心的概念。

Backbone 惟一的依赖就是 underscore.js (<http://documentcloud.github.com/underscore/>)，它是一个类库，由一些非常有用的工具和通用目的的 JavaScript 函数组成。Underscore 提供了 60 多个函数来处理数组操作、函数绑定、JavaScript 模板和深度的相等检测。如果你经常处理大量的数组，使用 Underscore 的 API 真是太值了。除了 Underscore，你也可以用 jQuery 或者 Zepto.js 来提升 Backbone 处理视图的能力。

Backbone 的文档虽然很完备，但刚开始阅读时还是有点茫然。本章的目的就是，通过对类库深入和实际的介绍帮你调整这种状态。刚开始的几节是对 Backbone 组件的概述，接下来将介绍一个实际的应用程序。如果你想马上看到 Backbone 可以直接跳过这些内容。

模型

166 让我们从 MVC 最关键的组件——模型开始。模型是保存应用程序数据的地方。你可以把模型想像为对应用程序原始数据的精心设计的抽象，并且添加了一些工具函数和事件。你可以在 `Backbone.Model` 上调用 `extend()` 函数来创建 Backbone 模型：

```
var User = Backbone.Model.extend({
  initialize: function() {
    // ...
  }
});
```

`extend()` 的第一个参数是一个对象，它成为了模型实例的属性。第二个参数是可选的类属性的哈希。通过多次调用 `extend()` 可以生成模型的子类，它们将继承父亲所有的类和实例的属性：

```
var User = Backbone.Model.extend({
  // 实例属性
  instanceProperty: "foo"
}, {
  // 类属性
  classProperty: "bar"
});

assertEqual( User.instanceProperty, "foo" );
assertEqual( User.prototype.classProperty, "bar" );
```

当模型实例化时，它的 `initialize()` 函数可以接受任意实例参数。背后的工作原理是，Backbone 模型本身是构造函数，所以可以使用 `new` 关键词来生成一个新的实例：

```
var User = Backbone.Model.extend({
  initialize: function(name) {
    this.set({name: name});
  }
});

var user = new User("Leo McGarry");
assertEqual( user.get("name"), "Leo McGarry");
```

模型和属性

使用 `set()` 和 `get()` 函数来设置和获取实例里的属性：

```
var user = new User();
user.set({name: "Donna Moss"})
```

```

assertEqual( user.get("name"), "Donna Moss" );
assertEqual( user.attributes, {name: "Donna Moss"} );

```

`set(attrs, [options])` 需要一个哈希形式表示的属性对象以便应用到实例上，`get(attr)` 只需要一个字符串参数——属性的名字——返回它的值。该实例使用本地一个哈希（名为 `attributes` 的对象）来保存和跟踪它的属性。不能直接操作该属性对象；而是使用 `get()` 和 `set()` 函数对其进行操作，确保调用相应的校验和事件。

可以使用 `validate()` 函数来校验一个实例的属性。默认情况下，此函数是没有定义的，◀ 167 但可以覆盖它来自定义校验逻辑：

```

var User = Backbone.Model.extend({
  validate: function(atts){
    if (!atts.email || atts.email.length < 3) {
      return "email must be at least 3 chars";
    }
  }
});

```

如果该模型和属性是合法的，`validate()` 函数不需要返回任何东西；如果属性不合法，返回值既可以是一个描述错误的字符串，也可以是一个 `Error` 对象。如果校验失败，`set()` 和 `save()` 函数将不再继续，而是触发一个 `error` 事件。程序可以绑定 `error` 事件，以保证当校验失败时得到通知：

```

var user = new User;

user.bind("error", function(model, error) {
  // 错误处理
});

user.set({email: "ga"});

// 或者给一个特定的集合添加一个错误处理程序
user.set({"email": "ga"}, {error: function(model, error){
  // ...
}});

```

使用哈希名为 `default` 的对象来指定默认属性。在创建一个模型实例时，任何没有指定值的属性都会被设置成默认值：

```

var Chat = Backbone.Model.extend({
  defaults: {
    from: "anonymous"
  }
});

assertEqual( (new Chat).get("from"), "anonymous" );

```

集合

在 Backbone 中，模型实例的数据存放在多个集合中。为什么在模型之间要使用独立的集合，其原因在这还不能立刻明显地看出来，但在实际场景中这种做法是很普遍的。例如，如果重新创建 Twitter，就至少需要两个集合，Followers 和 Followees，两者都由 User 的实例来填充数据。虽然两个集合都是由同一个模型来填充数据，但每个包含了不同 User 的实例的数组；因此它们是独立的集合。

针对模型，可以通过扩展 Backbone.Collection 来创建一个集合：

```
168 > var Users = Backbone.Collection.extend({  
    model: User  
});
```

在上面的例子中，可以看到我们覆盖了 model 属性来指定与集合相关联的模型——这里就是 User 模型。虽然这并不是绝对必需的步骤，但是为该集合设置一个默认模型指向在需要时能派上用场。通常，集合会包含单个模型类型的一个实例，而不是不同模型的多个实例。

在创建一个集合时，可以选择传递一个模型数组。比如 Backbone 的模型，如果定义了一个初始化实例的函数，在初始化时就会调用它：

```
var users = new Users([ {name: "Toby Ziegler"}, {name: "Josh Lyman"} ]);
```

另一种方法是使用 add() 函数给集合添加模型：

```
var users = new Users;  
  
// 添加一个单独的模型  
users.add({name: "Donna Moss"});  
  
// 或者添加模型组成的数组  
users.add([ {name: "Josiah Bartlet"}, {name: "Charlie Young"} ]);
```

在为集合添加一个模型时，会触发 add 事件：

```
users.bind("add", function(user) {  
    alert("Ahoy " + user.get("name") + "!");  
});
```

同样地，可以使用 remove() 函数从集合中删除一个模型，这时会触发 remove 事件：

```
users.bind("remove", function(user) {  
    alert("Adios " + user.get("name") + "!");  
});
```

```
users.remove( users.models[0] );
```

获取指定的模型很简单。如果有模型的 ID，可以使用控制器的 `get()` 函数：

```
var user = users.get("some-guid");
```

如果没有模型 ID，还可以使用 `cid` 获取模型——当创建一个新模型时由 Backbone 自动创建的客户 ID：

```
var user = users.getByCid("c-some-cid");
```

除了 `add` 和 `remove` 事件，当一个集合中的模型被修改后，还会触发 `change` 事件：

```
var user = new User({name: "Adam Buxton"});
```

```
var users = new Backbone.Collection;  
users.bind("change", function(rec){  
    // 改变了一个记录  
});  
users.add(user);
```

```
user.set({name: "Joe Cornish"});
```

169

控制集合的内部顺序

一个集合的内部元素顺序可以通过 `comparator()` 函数来控制，该函数的返回值代表你希望集合内部元素按何种规则排序：

```
var Users = Backbone.Collection.extend({  
    comparator: function(user){  
        return user.get("name");  
    }  
});
```

返回值既可以是一个字符串也可以是一个数值，以便将集合的元素按此规则排列（与 JavaScript 的常规排序不一样）。在上面的例子中，保证了 `Users` 集合内的元素是以 `name` 的字母顺序存储的。排序将在后台自动完成，但若要强化执行重新排序，可以调用 `sort()` 函数。

视图

Backbone 视图并不是模板本身，却是一些控制类，它们处理模型的表现。这有些让人迷惑，因为很多 MVC 的实现将视图看做多块 HTML 或模板，它们在控制器中处理事件和渲染。但不一样的是，在 Backbone 中，视图“代表了一个 UI 逻辑块，负责一个简单的 DOM 的内容”。

与模型和集合一样，视图也扩展自 Backbone 的现存类——这里就是 `Backbone.View`：

```
var UserView = Backbone.View.extend({
  initialize: function(){ /* ... */ },
  render: function(){ /* ... */ }
});
```

不管视图有没有被插入到页面中，每个视图的实例都知道当前的 DOM 的元素，抑或就是 `this.el`。el 是从视图的 `tagName`、`className` 或 `id` 等属性值中创建的元素。如果没有指定这些值，`el` 是一个空的 `div`：

```
var UserView = Backbone.View.extend({
  tagName: "span",
  className: "users"
});

assertEqual( (new UserView).el.className, "users" );
```

170 如果希望将视图绑定到页面中已存在的元素上，只要直接指定 `el` 即可。很明显，应该在页面加载完成以后再设置该视图；否则，可能找不到需要的元素：

```
var UserView = Backbone.View.extend({
  el: $(".users")
});
```

也可以在实例化一个视图时传递 `el`，它是可选的，用 `tagName`、`className` 或 `id` 等属性来标记：

```
new UserView({id: "followers"});
```

渲染视图

每个视图有一个 `render()` 函数，默认情况下里面没有任何操作（空函数）。一旦视图需要重绘，应该调用此函数。对不同的视图应该用不同功能的函数来覆盖该函数，以处理模板渲染，并使用新的 HTML 来更新 `el`：

```
var TodoView = Backbone.View.extend({
  template: _.template($("#todo-template").html()),

  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  }
});
```

Backbone 本身并不知道你是如何渲染视图的。你可以自己生成元素也可以使用模板类库。建议使用后者，因为通常这种方法更干净——让 HTML 保持在 JavaScript 程序之外。既然 Backbone 依赖的 `underscore.js` 加载到页面中了，那么就可以使用 `_.template()`

(<http://documentcloud.github.com/underscore/#template>)，这是一个生成模板非常方便的实用工具。

在前面的代码中，你会发现我们使用了一个叫做 `this.model` 的本地属性。它实际指向一个模型的实例，在实例化时传递到视图中。模型的 `toJSON()` 函数实际上返回模型未加工的原始属性，可以在模板中使用：

```
new TodoView({model: new Todo});
```

委托事件

通过委托，Backbone 的视图提供了一种添加事件到 `el` 的简单快捷的方法。如下代码就是在视图上通过 `events` 哈希对象设置了事件和对应回调：

```
var TodoView = Backbone.View.extend({
  events: {
    "change input[type=checkbox]" : "toggleDone",
    "click .destroy"           : "clear",
  },

  toggleDone: function(e) { /* ... */},
  clear: function(e) { /* ... */}
});
```

171

`events` 对象为 `{"eventType selector": "callback"}` 这样的格式。`selector` 是可选的，如果不提供，事件会直接绑定在 `el` 上；如果提供了，事件就会被委托 (<http://api.jquery.com/delegate/>)，即事件动态绑定在与 `selector` 匹配的 `el` 子元素上。委托利用了事件冒泡，也就意味着事件可以一直触发而不管 `el` 中的内容是否已经改变。

`callback` 是个字符串，它代表当前视图上的一个函数的实例。当视图的事件回调被触发时，在当前视图的上下文中就会调用它们，而不是在当前的目标元素下或者 `window` 的上下文中调用。因此，可以在任何回调中直接访问 `this.el` 和 `this.model`，这样非常方便，在上述例子的 `toggleDone()` 和 `clear()` 函数中就是这样做的。

绑定和上下文

实际情况中视图的 `render()` 函数是如何调用的？事实上每当视图的模型变化时，就会触发 `change` 事件，然后调用该函数。这就意味着应用程序的视图及 HTML 和与之对应的模型数据是同步（绑定）的：

```
var TodoView = Backbone.View.extend({
  initialize: function() {
    _.bindAll(this, 'render', 'close');
```

```

        this.model.bind('change', this.render);
    },

    close: function(){ /* ... */
    });

```

需要小心的是在回调函数中的上下文已经改变。Underscore 提供了一个很有用的函数来处理这个事情：`_.bindAll(context, *functionNames)` (<http://documentcloud.github.com/underscore/#bindAll>)。它将函数名字（字符串形式）和一个上下文绑定在一起。`_.bindAll()` 保证了所有给定的函数总是在指定的上下文中被调用。这在事件回调中尤其有用，因为回调函数的上下文总是在变化。在上述的例子中，`render()` 和 `close()` 函数总是能在 `TodoView` 的实例上下文中执行。

模型销毁的配合工作很简单。视图只需要绑定模型的 `delete` 事件，当此事件被触发时删除 `el` 即可：

```

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

  remove: function(){
    $(this.el).remove();
  }
});

```

172

请注意，渲染 Backbone 的视图时也可以不使用模型或事件回调。在 `initialize()` 函数中调用 `render()` 函数就可以在第一次实例化时渲染视图。但是，我们已经介绍过模型和视图的集成，对于视图来说它才是典型的案例——这种绑定的能力是 Backbone 最有用和强大的特性之一。

控制器

Backbone 的控制器将应用程序的状态和 URL 的哈希片段关联在一起，使 URL 地址可分享，也可作为书签使用。本质上，控制器由一些路由和函数构成，当导航到这些路由时那些函数就会被调用。

路由是一个哈希对象——其键由路径、参数和“隔板”构成——其值设置为与之相关的函数：

```

routes: {
  "help":
    // Matches:
    "help", // #help

```

```

    "search/:query":      "search", // #search/kiwis
    "search/:query/p:page": "search" // #search/kiwis/p7
    "file/*path":         "file"     // #file/any/path.txt
  }

```

上例中可以看到参数是以“:”开始的，紧接着是参数名。当路由被调用时，其所有参数都会传递到它的动作^{注1}中。用“*”指定的“隔板”基本上是一个通配符，匹配所有的内容。和参数一样，“隔板”将和匹配的值一起传递到路由的动作中。

路由是以在哈希对象中定义顺序的倒序进行解析的。换言之，最通用的“匹配所有”的路由应该位于 routes 哈希对象的尾部。

通常，扩展 Backbone.Controllers，给它传递一个包含初始化属性的对象，可以创建控制器：

```

var PageController = Backbone.Controller.extend({
  routes: {
    "":                "index",
    "help":            "help",    // #help
    "search/:query":   "search",  // #search/kiwis
    "search/:query/p:page": "search" // #search/kiwis/p7
  },

  index: function(){ /* ... */ },

  help: function() {
    // ...
  },

  search: function(query, page) {
    // ...
  }
});

```

173

在上例中，当用户被导航到 <http://example.com#search/coconut> 时，不管是手动输入的还是单击“后退”按钮，都将调用 search() 函数，并传递进一个 query 变量，值为“coconut”。

如果希望你的应用程序既适用于 Ajax Crawling 规范 (<http://goo.gl/rv00>)，也能被搜索引擎索引（第 4 章有讨论），所有的路由前缀必须是“!/”，如以下代码所示：

```

var PageController = Backbone.Controller.extend({
  routes: {
    "!/page/:title": "page", // #!/page/foo-title
  }
  // ...
});

```

注 1： 函数——译者注

同时, 根据规范你还需要对服务端做一些修改。如果要想让路由功能更丰富, 比如要保证某些参数是整数, 可以直接给 `route()` 函数传递一个正则表达式:

```
var PageController = Backbone.Controller.extend({
  initialize: function(){
    this.route(/pages\/(\d+)/, 'id', function(pageId){
      // ...
    });
  }
});
```

总之, 路由将 URL 的哈希片段的变化和控制器联系起来, 但如何在起点设置该片段呢? 无须手动设置 `window.location.hash`, Backbone 提供了一个便捷的方法——`saveLocation(fragment)` 函数:

```
Backbone.history.saveLocation("/page/" + this.model.id);
```

当调用 `saveLocation()` 函数并更新了 URL 的哈希片段时, 并不会调用任何控制器。这意味着, 比如你可以在视图的 `initialize()` 函数中放心地调用 `saveLocation()` 函数而不会有任何控制器介入。

在内部, 如果浏览器支持, Backbone 将监听浏览器的 `onhashchange` 事件, 否则它会使用 `iframe` 和定时器来实现一个变通的方案。但是, 首先需要开启 Backbone 的 `history` 来支持这个功能:

```
Backbone.history.start();
```

在开启 Backbone 的 `history` 之前要保证页面已经加载完, 并且保证视图、模型和集合都是可用的。就目前来说, Backbone 还不支持 HTML5 中的 `pushState()` 和 `replaceState()` 等 `history` API。因为它们在服务端需要特殊的处理, 并且 IE 也还不支持。在这些问题解决后 Backbone 也许会增加对它们的支持。目前所有的路由功能都是通过 URL 的哈希片段来实现的。

与服务器的同步

默认情况下, 只要保存模型, Backbone 就会用 Ajax 请求通知服务器, 它可以使用 jQuery 或者 Zepto.js 等类库。当创建、更新或删除一个模型之前, Backbone 通过调用 `Backbone.sync()` 函数来实现这个功能。Backbone 发起 REST 形式^{注2}的 JSON 请求到服务器, 如果成功, 就将更新客户端的模型。

注2: 表征状态转移 (英文: Representational State Transfer, 简称 REST), 更多内容请阅读 <http://zh.wikipedia.org/zh-cn/REST>。——译者注

如要利用这个特性，需要在模型中定义一个名为 `url` 的属性并赋值，并且要求服务器处理请求符合 REST 形式。Backbone 将处理剩下的所有任务：

```
var User = Backbone.Model.extend({
  url: '/users'
});
```

`url` 属性既可以是一个字符串也可以是一个返回字符串的函数。路径既可以是相对的也可以是绝对的，但必须返回模型端点。

Backbone 将创建、读取、更新和删除（CRUD）等动作映射到以下方法：

```
create → POST   /collection
read   → GET    /collection[id]
update → PUT    /collection/id
delete → DELETE /collection/id
```

例如，如果你在创建一个 `User` 实例，Backbone 会发送一个 POST 请求到 `/users`。同样地，更新一个 `User` 实例，会发送一个 PUT 请求到 `/users/id` 这个端点，这里的 `id` 是模型的惟一标识。Backbone 期望服务器响应 POST、PUT 和 GET 请求时返回一个 JSON 形式的哈希对象，而且它应该包含用来更新实例的一些属性。

如果要调用模型的 `save([attrs], [options])` 函数将模型保存到服务器上，可以随意传递一个由属性和请求项组成的哈希对象。如果模型有了一个 `id`，首先假设该模型在服务器上已经存在，`save()` 发送的是一个 PUT（更新）请求。否则，`save()` 发送的是一个 POST（创建）请求：

```
var user = new User();
user.set({name: "Bernard"});

user.save(null, {success: function(){
  // 保存用户成功
}});
```

所有的 `save()` 调用都是异步的，但可以通过设置 `success` 和 `failure` 两个选项来监听 Ajax 请求的回调。事实上，如果 Backbone 使用了 jQuery，传递给 `save()` 的所有选项都将传递给 `$.ajax()`。换言之，在保存模型时可以使用所有 jQuery 的 Ajax 选项（<http://goo.gl/mCb2>），比如 `timeout`。

◀ 175

如果服务器返回错误，`save` 操作就会失败，并在模型上触发一个 `error` 事件。反之，如果成功，模型将使用服务器的响应直接更新数据：

```
var user = new User();

user.bind("error", function(e){
```

```
// 服务器返回了一个错误
});

user.save({email: "Invalid email"});
```

可以使用 `fetch()` 函数来刷新模型，该函数将会（通过一个 GET 请求）从服务器请求模型的属性。如果远程模型所表示的数据和当前模型属性数据不一致，将触发一个 *change* 事件：

```
var user = Users.get(1);
user.fetch();
```

填充集合

到目前为止，已经介绍了模型的创建和更新，但第一次如何从服务器上获取模型数据呢？这就是 Backbone 的集合应运而生的原因，它们用来请求远程的模型并保存在本地。和模型类似，必须给集合指定一个 `url` 属性来设置它的端点。如果没提供 `url`，Backbone 将转而使用与之相关联的模型的 `url`：

```
var Followers = Backbone.Collection.extend({
  model: User,
  url: "/followers"
});

Followers.fetch();
```

集合的 `fetch()` 函数将发送一个 GET 请求到服务器——在这个例子中，发送到 `/followers`——来获取远程模型。当模型数据从服务器返回后，该集合会刷新，并触发一个 *refresh* 事件。

可以使用 `refresh()` 函数来手动刷新集合，传入一个模型对象的数组即可。在第一次设置页面的时候使用这种方法非常方便。和页面加载后发送的另外一个 GET 请求不一样，你可以通过 `refresh()` 传入一个内联的 JSON 对象，而预先填充数据到集合中。以下代码是一个使用 Rails 的例子：

```
<script type="text/javascript">
  Followers.refresh(<%= @users.to_json %>);
</script>
```

176 服务器端

如上面提到的，为了与 Backbone 无缝整合，服务器需要实现一些 REST 形式的端点：

```
create → POST   /collection
read   → GET    /collection
```

```
read    → GET    /collection/id
update  → PUT    /collection/id
delete  → DELETE /collection/id
```

Backbone 在发送模型之间先将其序列化为 JSON 格式。User 模型看上去像这样：

```
{"name": "Yasmine"}
```

请注意，上面的数据并没有使用当前模型作为前缀，这让 Rails 的开发者有些犯难。下面就来介绍 Rails 与 Backbone 整合的规范，如果您没有使用该框架，请直接跳过下面的章节。

在 CRUD 方法中，应该使用简单的、无前缀的参数。例如以下代码的 Rails 控制器的 update 方法就能正常工作：

```
def update
  user = User.find(params[:id])
  user.update_attributes!(params)
  render :json => user
end
```

很显然，应该使用 attr_accessible 方法通过白名单属性 (<http://guides.rubyonrails.org/security.html#mass-assignment>) 来保护模型免受恶意输入的侵扰，不过这些超出了本书的范围。除了 destroy 以外，控制器的每一个方法都必须返回表示该记录的一个 JSON 对象。

将属性序列化为 JSON 格式也是个问题，默认情况下，Rails 会将模型名称作为前缀，就像这样：

```
{"user": {"name": "Daniela"}}
```

不幸的是，Backbone 不能正确地解析该对象。你需要保证 Rails 序列化时在 JSON 中不包含模型名称，可以用新建一个初始化文件来解决：

```
# config/initializers/json.rb
ActiveRecord::Base.include_root_in_json = false
```

自定义行为

在 Backbone 每次试图读取或保存模型到服务器时都会调用 Backbone.sync() 函数。你可以覆盖此函数来改变它默认的行为(发送 Ajax 请求),以便使用一种不同的持久化策略,如 WebSockets、XML 传输流或本地存储。例如,下面的函数只是用来记录一些调用参数的日志,其他什么都没有做,用它来覆盖 Backbone.sync() :

◀ 177

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

可以看到，`Backbone.sync()` 函数需要传递 `method`、`model` 和 `options` 等对象，如下所示：

`method`

即 CRUD 方法（create、read、update 或 delete）。

`model`

需要保存的模型（或需要读取的集合）。

`options`

请求的可选项，包括成功和失败的回调函数。

Backbone 希望你做的惟一一件事情就是调用 `options.success()`，或者调用 `options.error()`。

每个模型或集合覆盖各自 `sync()` 函数而不是在全局处理也是可行的：

```
Todo.prototype.sync = function(method, model, options){ /* ... */ };
```

在本地存储适配器里（<https://github.com/jeromegn/Backbone.localStorage>）有一个很好的自定义 `Backbone.sync()` 函数的例子。包含该适配器，并且在相关的模型或集合中设置 `localStorage` 选项后，Backbone 就能使用 HTML5 的 `localStorage`，而不是后台的服务器。在下面的例子中可以看到，`Backbone.sync()` 对 `store` 对象的 CRUD 操作，并且对于不同的方法，最后用相应的模型调用了 `options.success()` 函数：

```
// 将所有待完成的项都保存至本地存储的命名空间 "todos" 中
Todos.prototype.localStorage = new Store("todos");

// 重写 Backbone.sync(), 这样就可以给模型或者集合的本地存储属性添加委托
// 本地存储属性应当是一个 Store 的实例
Backbone.sync = function(method, model, options) {

  var resp;
  var store = model.localStorage || model.collection.localStorage;

  switch (method) {
    case "read":  resp = model.id ? store.find(model) : store.findAll(); break;
    case "create": resp = store.create(model);                               break;
    case "update": resp = store.update(model);                               break;
    case "delete": resp = store.destroy(model);                             break;
  }
}
```

```

    if (resp) {
      options.success(resp);
    } else {
      options.error("Record not found");
    }
  }
};

```

构建 To-Do 列表应用

让我们用所学到的关于 Backbone 的知识做一次练习，编写一个简单的 to-do 列表应用程序。我们希望用户能够对这些 to-do 进行增、删、改和查，在页面刷新后仍然能保持数据。可以用下面例子中的代码来构建这个应用程序，或者查看 *assets/ch12/todos* 目录下已经完成的应用程序。

页面的初始结构如下；我们加载了 CSS 文件、一些 JavaScript 库以及包含在 *todos.js* 文件中的 Backbone 应用代码：

```

<html>
<head>
  <link href="todos.css" media="all" rel="stylesheet" type="text/css"/>
  <script src="lib/json2.js"></script>
  <script src="lib/jquery.js"></script>
  <script src="lib/jquery.templ.js"></script>
  <script src="lib/underscore.js"></script>
  <script src="lib/backbone.js"></script>
  <script src="lib/backbone.localStorage.js"></script>
  <script src="todos.js"></script>
</head>

<body>
  <div id="todoapp">
    <div class="title">
      <h1>Todos</h1>
    </div>

    <div class="content">
      <div id="create-todo">
        <input id="new-todo" placeholder="What needs to be done?" type="text" />
      </div>

      <div id="todos">
        <ul id="todo-list"></ul>
      </div>
    </div>
  </div>
</body>
</html>

```

页面结构很简单，只包含了：一个文本输入框（`#new-todo`），用来创建新的 to-do；一个列表（`#todo-list`），用来显示已存在的 to-do。

接下来看看 `todos.js` 脚本，这里面是 Backbone 应用的核心代码。我们把所有的东西都用 `jQuery()` 封装起来，保证了在页面加载完成以后才运行代码：

179 

```
// todos.js
jQuery(function($){
  // 应用程序在此 ...
})
```

下面创建基本的 `Todo` 模型，它有 `content` 和 `done` 属性。同时我们提供了 `toggle()` 辅助函数来翻转模型的 `done` 属性值：

```
window.Todo = Backbone.Model.extend({
  defaults: {
    done: false
  },

  toggle: function() {
    this.save({done: !this.get("done")});
  }
});
```

我们在 `window` 对象上设置 `Todo` 模型以便它能被全局访问。同时，使用这种模式，就能很清楚地看到一个脚本定义了哪些全局变量——只要看一眼脚本中有哪些地方引用了 `window` 对象即可。

下一步，设置 `TodoList` 集合，这也是保存 `Todo` 模型数组的地方：

```
window.TodoList = Backbone.Collection.extend({
  model: Todo,

  // 将所有 to-do 项都保存至 "todos" 命名空间下
  localStorage: new Store("todos"),

  // 从 to-do 项列表中过滤掉对所有已经完成任务
  done: function() {
    return this.filter(function(todo){ return todo.get('done'); });
  },

  remaining: function() {
    return this.without.apply(this, this.done());
  }
});

// 创建全局集合
window.Todos = new TodoList;
```

我们使用的是 Backbone 的本地存储解决方案 (*backbone.localstorage.js*)，它要求必须在每个需要保存数据的集合或模型中设置一个 `localStorage` 属性。在 `ToDoList` 中另外两个函数 `done()` 和 `remaining()` 用于过滤集合，返回已经或还没有完成的 to-do 模型数据。因为只可能有一个 `ToDoList`，我们将其实例化为一个全局变量：`window.Todos`。

下一步，用来显示单独的 to-do 的视图，叫做 `ToDoView`。它绑定了 `ToDo` 模型的 `change` 事件，当事件被触发时重新渲染视图：

```
window.ToDoView = Backbone.View.extend({  
  
    // 视图是一个 li 标签  
    tagName: "li",  
  
    // 对于每个单独的项都缓存一个模板函数  
    template: $("#item-template").template(),  
  
    // 给视图函数委托事件  
    events: {  
        "change .check"      : "toggleDone",  
        "dblclick .todo-content" : "edit",  
        "click .todo-destroy"  : "destroy",  
        "keypress .todo-input"  : "updateOnEnter",  
        "blur .todo-input"     : "close"  
    },  
  
    initialize: function() {  
        // 确保在正确的作用域调用函数  
        _.bindAll(this, 'render', 'close', 'remove');  
  
        // 监听模型的修改  
        this.model.bind('change', this.render);  
        this.model.bind('destroy', this.remove);  
    },  
  
    render: function() {  
        // 使用存储的模板来更新 el  
        var element = jQuery.tmpl(this.template, this.model.toJSON());  
        $(this.el).html(element);  
        return this;  
    },  
  
    // 当复选框被选中时，就将模型切换为已完成状态  
    toggleDone: function() {  
        this.model.toggle();  
    },  
  
    // 将当前视图切换为 " 编辑 " 模式，显示 input 输入框  
    edit: function() {
```

```

    $(this.el).addClass("editing");
    this.input.focus();
  },

  // 关闭 " 编辑 " 模式, 将更改保存至 to-do 列表中
  close: function(e) {
    this.model.save({content: this.input.val()});
    $(this.el).removeClass("editing");
  },

  // 如果按下了回车键, 则结束编辑状态
  // 调用 close(), 触发 input 的 blur 事件
  updateOnEnter: function(e) {
    if (e.keyCode == 13) e.target.blur();
  },

  // 当模型被销毁时也删除元素
  remove: function() {
    $(this.el).remove();
  },

  // 当点击 ".todo-destroy" 时销毁模型
  destroy: function() {
    this.model.destroy();
  }
});

```

181

可以看到我们将许多事件委托给管理更新、完成和删除 to-do 的视图。例如, 每当复选框状态改变时, `toggleDone()` 会被调用, 并切换模型的 `done` 属性值。这个结果将触发模型的 `change` 事件, 致使视图重新渲染。

我们使用了 `jQuery.templ` (<http://api.jquery.com/category/plugins/templates/>) 来处理 HTML 模板, 每当渲染时视图就用重新生成的模板替换 `el` 中的内容。该模板引用 ID 为 `#item-template` 的元素, 不过我们还未定义此元素。将其放到 `index.html` 页面的 body 标签里:

```

<script type="text/template" id="item-template">
  <div class="todo {{if done}}done{{/if}}">
    <div class="display" title="Double click to edit...">
      <input class="check" type="checkbox" {{if done}}checked="checked"{{/if}} />
      <div class="todo-content">${content}</div>
      <span class="todo-destroy"></span>
    </div>
    <div class="edit">
      <input class="todo-input" type="text" value="${content}" />
    </div>
  </div>
</script>

```

本书第 5 章比较深入地介绍了 jQuery.templ，如果你读过，那么模板语法对你而言应该相当地熟悉了。实际上，我们是在互相操作 `#todo-content` 和 `#todo-input` 元素里的内容。另外，要保证复选框的“选择”状态是正确的。

`TodoView` 一定程度上是自包含的——只要在实例化的时候提供给它一个模型，并且将 `el` 属性指定为 to-do 列表。这就是 `AppView` 的基本工作，它保证初始的 to-do 列表是用实例化的 `TodoView` 实例填充的。`AppView` 的另一个角色就是当用户点击 Return 按钮时根据 `#new-todo` 的输入内容创建新 Todo 记录：

```
// 我们的整个 AppView 是顶层的 UI 片段
window.AppView = Backbone.View.extend({

  // 给现有的 App 骨架绑定已有的 HTML，而不是创建新元素
  el: $("#todoapp"),
  events: {
    "keypress #new-todo": "createOnEnter",
    "click .todo-clear a": "clearCompleted"
  },

  // 在初始化时，我们将相关的事件绑定给了 Todos 集合
  // 当添加或者修改集合中的元素时触发这些事件
  // 通过载入可能存在本地存储中的记录来给出初始数据
  initialize: function() {
    _.bindAll(this, 'addOne', 'addAll', 'render');

    this.input = this.$("#new-todo");

    Todos.bind('add', this.addOne);
    Todos.bind('refresh', this.addAll);

    Todos.fetch();
  },

  // 创建一个视图并将这个元素绑定到 `<ul>` 中，
  // 以此为列表添加一个单独的 to-do 项
  addOne: function(todo) {
    var view = new TodoView({model: todo});
    this.$("#todo-list").append(view.render().el);
  },

  // 一次性将所有项都添加至 Todos 集合中
  addAll: function() {
    Todos.each(this.addOne);
  },

  // 如果在主输入框域中敲了回车键，则创建一个新的 Todo 模型
  createOnEnter: function(e) {
    if (e.keyCode !== 13) return;
```

```

    var value = this.input.val();
    if ( !value ) return;

    Todos.create({content: value});
    this.input.val('');
  },

  clearCompleted: function() {
    _.each(Todos.done(), function(todo){ todo.destroy(); });
    return false;
  }
});

// 最后, 创建一个 App
window.App = new AppView;

```

当页面首次加载后, Todos 集合将填充数据, 然后触发 *refresh* 事件调用。这将调用 *addAll()* 来获取所有的 Todo 模型, 生成 *TodoView* 视图, 并将它们添加到 *#todo-list* 中。另外, 当有新的 Todo 模型添加到 Todos 中时, Todos 触发 *add* 事件, 调用 *addOne()* 并将一个新的 *TodoView* 添加到 *#todo-list* 的列表中。也就是说, 初始化填充和 Todo 的创建是由 *AppView* 来处理的, 而它们的更新和销毁由单独的 *TodoView* 视图来处理。

让我们马上刷新页面看看刚完成的工作的结果如何。尽管会有一些 bug 和排版的问题, 看到的结果应该和图 12-1 类似。

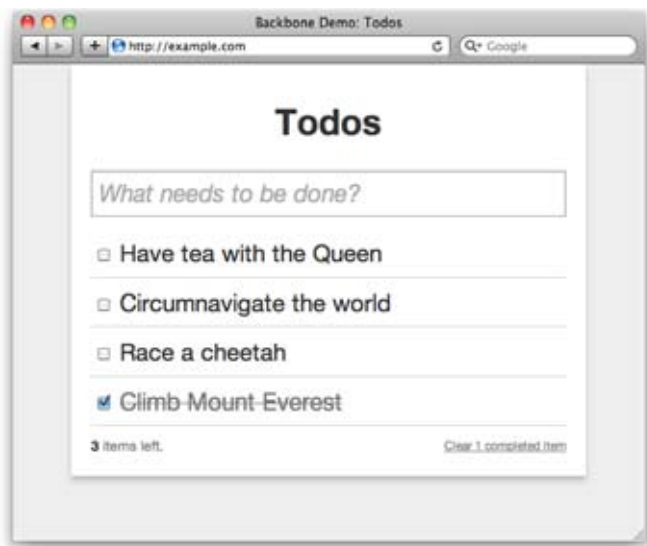


图12-1: 完整的Backbone Todo应用程序

我们已经实现了包含添加、检查、更新和删除功能的“to-do”应用,而且只使用较少的代码。由于使用了 Backbone 适配器来处理本地存储,“to-do”在页面加载中做到了持久化。这个例子应该能让你充分理解 Backbone 的用武之地何在,以及如何用它来创建你的应用程序。

你可以在随书文件中找到完整的应用程序,位于 *assets/ch12/todos* 目录下。

JavaScriptMVC类库

Justin Meyer (<http://jupiterjs.com/pages/justin-meyer>), JavaScriptMVC 的作者, 贡献了本章内容。

JavaScriptMVC (JMVC) 是一个基于 jQuery 的开源 JavaScript 框架。它基本上就是一个完整的前端开发框架, 将一些实用的测试工具、依赖关系管理、文档和许多非常有用的 jQuery 插件都打包在一起了。

同时, JavaScriptMVC 的每一部分都可以独立使用, 而不依赖于其他部分, 这使得该类库非常轻量。它的类、模型、视图和控制器算在一起经缩减和压缩后只有 7KB 大小, 当各部分独立使用时就更小了。JavaScriptMVC 的独立性能让应用开始时很小, 遇到 Web 上非常复杂的应用程序时也能扩展而从容应对。

本章内容涵盖 JavaScriptMVC 的 `$.Class`、`$.Model`、`$.View` 和 `$.Controller`。下面依次来介绍这些组件:

`$.Class`

基于 JavaScript 的类系统。

`$.Model`

传统的模型层。

`$.View`

客户端模板系统。

`$.Controller`

jQuery 的 widget 工厂。

JavaScriptMVC 的命名习惯与传统的模型 - 视图 - 控制器 (<http://goo.gl/488Hp>) 设计模式有轻微的不同。例如, `$.Controller` 用来创建传统的视图控件, 比如分页按钮和列表, 也用来创建在传统的视图和模型之间进行协同的传统的控制器。

186 设置

JavaScriptMVC 可以作为单个下载来使用, 包含了整个框架的所有内容。既然本章内容只涉及 MVC 部分, 请使用编译下载工具 (<http://javascriptmvc.com/builder.html>), 选中控制器、模型和视图的 EJS 模板, 然后单击“下载”按钮。

下载文件中包含了缩减版以及连同 jQuery 和你选中的插件一起的非缩减版。在页面中加载以下脚本标签:

```
<script type='text/javascript' src='jquery-1.6.1.js'></script>
<script type='text/javascript' src='jquerymx-1.0.custom.js'></script>
```

Class

JMVC 的控制器和模型继承自它的辅助类: `$.Class`。创建一个类, 可以调用 `$.Class (NAME, [classProperties, ] instanceProperties)`:

```
$.Class("Animal",{
  breathe : function(){
    console.log('breathe');
  }
});
```

在上面的例子中, `Animal` 的实例有一个 `breathe()` 方法。我们创建一个新的 `Animal` 实例就可以在其之上调用 `breathe()` 方法:

```
var man = new Animal();
man.breathe();
```

如果想创建一个子类, 只要调用基类构造函数, 传递子类名和属性即可:

```
Animal("Dog",{
  wag : function(){
    console.log('wag');
  }
})

var dog = new Dog;
dog.wag();
dog.breathe();
```

实例化

当创建一个类的新实例时，会调用类的 `init` 方法，并将参数传递给构造函数：

```
$.Class('Person',{
  init : function(name){
    this.name = name;
  },

  speak : function(){
    return "I am " + this.name + ".";
  }
});

var payal = new Person("Payal");
assertEqual( payal.speak() , 'I am Payal.' );
```

187

调用基类的方法

使用 `this._super` 来调用基类的方法。下面的代码比 `person` 提供更有“风度”的问候语：

```
Person("ClassyPerson", {
  speak : function(){
    return "Salutations, " + this._super();
  }
});

var fancypants = new ClassyPerson("Mr. Fancy");
assertEquals( fancypants.speak() , 'Salutations, I am Mr. Fancy.'
```

代理

`Class` 的 `callback` 方法返回一个函数，这个函数里面“`this`”指向了适当的上下文，类似 `jQuery` 的 `$.proxy` (<http://api.jquery.com/jQuery.proxy/>)。下面的代码创建了一个 `Clicky` 类，来统计它被点击了多少次：

```
$.Class("Clicky",{
  init : function(){
    this.clickCount = 0;
  },

  clicked: function(){
    this.clickCount++;
  },

  listen: function(el){
    el.click( this.callback('clicked') );
  }
});
```

```

    }
  })

  var clicky = new Clicky();
  clicky.listen( $('#foo') );
  clicky.listen( $('#bar') );

```

静态继承

Class 能够定义静态属性和方法的继承。下面的代码允许调用 `Person.findOne(ID, success(person))` 从服务器获取一个 `person` 实例。成功后调用回调函数，传入 `Person` 的一个实例对象，它拥有 `speak` 方法：

```

$.Class("Person",{
  findOne : function(id, success){
    $.get('/person/'+id, function(attrs){
      success( new Person( attrs ) );
    },'json')
  }
},{
  init : function(attrs){
    $.extend(this, attrs)
  },
  speak : function(){
    return "I am "+this.name+ ".";
  }
})

Person.findOne(5, function(person){
  assertEquals( person.speak(), "I am Payal." );
})

```

自省

Class 提供命名空间，而且通过 Class 可以访问类名和命名空间对象的名字：

```

$.Class("Jupiter.Person");

Jupiter.Person.shortName; //-> 'Person'
Jupiter.Person.fullName;  //-> 'Jupiter.Person'
Jupiter.Person.namespace; //-> Jupiter

var person = new Jupiter.Person();

person.Class.shortName; //-> 'Person'

```

一个模型的例子

全部综合起来，我们可以开发一个最基本的 ORM 风格的模型层。只要继承自 `Model`，从 REST 形式的 Web 服务器请求数据，得到数据后包装在继承自 `Model` 的实例中：

```
$.Class("Model",{
  findOne : function(id, success){
    $.get('/' + this.fullName.toLowerCase() + '/' + id,
      this.callback(function(attrs){
        success( new this( attrs ) );
      })
    }, 'json')
  }
},{
  init : function(attrs){
    $.extend(this, attrs)
  }
})

Model("Person",{
  speak : function(){
    return "I am "+this.name+ ".";
  }
});

Person.findOne(5, function(person){
  alert( person.speak() );
});

Model("Task");

Task.findOne(7,function(task){
  alert(task.name);
});
```

189

上例的工作原理与 JavaScriptMVC 的模型层类似。

模型

JavaScriptMVC 的模型和它的插件提供了很多组织模型数据的工具，如校验、关联、列表等等。但核心功能集中在服务封装、类型转换和事件上。

属性和可观察

对一个模型层来说，最重要的就是获取和设置模型上数据属性的能力，以及在一个模型实例上监听各种变化。这就是观察者模式，它处于 MVC 应用的中心地带——视图监听模型中的变化。

所幸的是用了 JavaScriptMVC 就很容易让所有的数据可观察。分页是一个很好的例子。在页面上存在多个分页控件是很常见的。例如，一个用来控制“Next”和“Previous”按钮；另一个用来控制当前正在显示的详情（例如，“Showing items 1-20”）。所有分页控件都需要以下这些数据：

offset

显示的第一项的索引值。

limit

显示选项的数量。

count

选项的总数量。

可以这样用 JavaScriptMVC 的 \$.Model 来组织模型：

```
190 > var paginate = new $.Model({  
    offset: 0,  
    limit: 20,  
    count: 200  
});
```

paginate 变量现在就变成了可观察的。可以将这个变量传递给分页控件供其读取、写入以及监听属性的变化。你可以用常规的方法或使用 model.attr(NAME) 方法来读取属性：

```
assertEqual( paginate.offset, 0 );  
assertEqual( paginate.attr('limit') , 20 );
```

如果单击了“Next”按钮，我们需要将 offset 加 1，可以使用 model.attr(NAME, VALUE)。如下代码将 offset 移动到下一页：

```
paginate.attr('offset', 20);
```

当一个控件改变了 paginate 的状态时，应该通知其他的控件。你可以用 model.bind(ATTR, success(ev, newVal)) 来绑定一个特定属性的变化，并更新该控件：

```
paginate.bind('offset', function(ev, newVal){  
    $('#details').text( 'Showing items ' + (newVal + 1 ) + '-' + this.count )  
})
```

只要绑定 “updated.attr” 事件，就可以监听任何属性的变化：

```
paginate.bind('updated.attr', function(ev, newVal){
    $('#details').text( 'Showing items ' + (newVal+1) + '-' + this.count )
})
```

下面的代码是一个 next-previous 的 jQuery 插件，它接受 paginate 的数据：

```
$.fn.nextPrev = function(paginate){
    this.delegate('.next','click', function(){
        var nextOffset = paginate.offset + paginate.limit;
        if( nextOffset < paginate.count){
            paginate.attr('offset', nextOffset );
        }
    });

    this.delegate('.prev','click', function(){
        var nextOffset = paginate.offset-paginate.limit;
        if( 0 < paginate.offset ){
            paginate.attr('offset', Math.max(0, nextOffset) );
        }
    });

    var self = this;
    paginate.bind('updated.attr', function(){
        var next = self.find('.next'),
            prev = self.find('.prev');
        if( this.offset == 0 ){
            prev.removeClass('enabled');
        } else {
            prev.removeClass('disabled');
        }
        if( this.offset > this.count - this.limit ){
            next.removeClass('enabled');
        } else {
            next.removeClass('disabled');
        }
    });
};
```

191

这个插件还有一些小问题。首先，如果控件从页面上删除了，它不会从 paginate 上自己解除绑定。我们在讨论控制器时来解决这个问题。

其次，逻辑上插件内部应该防止出现负的 offset 或 offset 超出总数的现象。这个逻辑应该在模型中处理。为了修复这个问题，我们需要创建一个分页类，在那里可以添加额外的约束来限制 limit、offset 和 count 的值。

扩展模型

JavaScriptMVC 的模型继承自 `$.Class`。因此，你创建的模型类是从 `$.Model(NAME, [STATIC, ], PROTOTYPE)` 上继承的：

```
$.Model('Paginate',{
  staticProperty: 'foo'
},{
  prototypeProperty: 'bar'
})
```

还有一些方法来让 `Paginate` 模型更加有用。下面要讨论的是，通过添加 `setter` 方法，就可以限制设置 `count` 和 `offset` 的值。

Setter

`setter` 方法是模型的原型方法，它们的命名规则为 `setName`。当 `val` 存在，并且被传递给 `model.attr(NAME, val)` 时，`setter` 方法就会被调用。同时，还传递 `success` 和 `error` 等回调函数。通常，该方法应该返回模型实例被设置的值，或者调用 `error` 函数输出一个错误消息。`success` 函数是在异步调用 `setter` 时使用的。

`Paginate` 模型用 `setter` 来避免设置无效的总数和偏移量。例如，必须保证数值不能为负数：

```
$.Model('Paginate',{
  setCount : function(newCount, success, error){
    return newCount < 0 ? 0 : newCount;
  },

  setOffset : function(newOffset, success, error){
    return newOffset < 0 ? 0 :
    Math.min(newOffset, !isNaN(this.count - 1) ? this.count : Infinity )
  }
});
```

192 现在，该 `nextPrev` 插件能够比较随意地设置偏移量：

```
this.delegate('.next','click', function(){
  paginate.attr('offset', paginate.offset+paginate.limit);
});

this.delegate('.prev','click', function(){
  paginate.attr('offset', paginate.offset-paginate.limit );
});
```

Defaults

我们可以设置 `Paginate` 实例的静态属性 `defaults` 来为它赋予初始的默认值。在新建一个 `paginate` 实例时，如果没有提供初始值就会使用这些默认值：

```
$.Model('Paginate',{
  defaults : {
    count: Infinity,
    offset: 0,
    limit: 100
  }
},{
  setCount : function(newCount, success, error){ ... },
  setOffset : function(newOffset, success, error){ ... }
});

var paginate = new Paginate({count: 500});
assertEqual(paginate.limit, 100);
assertEqual(paginate.count, 500);
```

给 `Paginate` 模型添加辅助方法，可以把“Next”和“Previous”的实现变得更加简单。

辅助方法

辅助方法是一些可以协助设置或获取模型实例上有用的数据的原型方法。下面的代码是一个完成的 `Paginate` 模型，包含了 `next` 和 `prev` 等方法，用来实现页面的前进和回退。同时提供了 `canNext` 和 `canPrev` 方法来判断页面是否能够移动到下一页：

```
$.Model('Paginate',{
  defaults : {
    count: Infinity,
    offset: 0,
    limit: 100
  }
},{
  setCount : function( newCount ){
    return Math.max(0, newCount );
  },
  setOffset : function( newOffset ){
    return Math.max( 0 , Math.min(newOffset, this.count ) )
  },
  next : function(){
    this.attr('offset', this.offset+this.limit);
  },
  prev : function(){
    this.attr('offset', this.offset - this.limit )
  },

```

```

    canNext : function(){
        return this.offset > this.count - this.limit
    },
    canPrev : function(){
        return this.offset > 0
    }
})

```

因此，我们的 jQuery widget 越来越精致了：

```

$.fn.nextPrev = function(paginate){
    this.delegate('.next','click', function(){
        paginate.attr('offset', paginate.offset+paginate.limit);
    })
    this.delegate('.prev','click', function(){
        paginate.attr('offset', paginate.offset-paginate.limit );
    });
    var self = this;
    paginate.bind('updated.attr', function(){
        self.find('.prev')[paginate.canPrev() ? 'addClass' : 'removeClass']('enabled')
        self.find('.next')[paginate.canNext() ? 'addClass' : 'removeClass']('enabled');
    })
};

```

服务封装

刚才我们已看到 `$.Model` 在客户端状态建模方面是多么有用。但是，对多数应用来说，重要的数据都放在服务器而非客户端上。客户端需要在服务器上创建、获取、更新和删除（CRUD）数据。维护好客户端和服务上数据的一致性是一件相当不容易的事情；而 `$.Model` 能简化该问题。

`$.Model` 非常灵活。它可以与任何类型的服务和数据类型协同工作。本书只讨论 `$.Model` 与当下最常用和流行的服务和数据类型的工作原理：表述性状态转移（REST）和 JSON。

REST 服务使用 URL 和 HTTP 的谓词 POST、GET、PUT 和 DELETE 来分别创建、获取、更新和删除数据。例如，一个允许你 CRUD 任务的 task 服务看上去会是这个样子：

```

create    → POST    /tasks
read all  → GET     /tasks
read      → GET     /tasks/2
update    → PUT     /tasks/2
delete    → DELETE  /tasks/2

```

使用下面的代码就能在服务器上完成数据的创建、获取、更新和删除：

```
$.Model("Task",{
  create : "POST /tasks.json",
  findOne : "GET /tasks/{id}.json",
  findAll : "GET /tasks.json",
  update : "PUT /tasks/{id}.json",
  destroy : "DELETE /tasks/{id}.json"
},{ });
```

接下来进一步说明怎样使用 Task 模型完成 CRUD 的操作。

创建任务

```
new Task({ name: 'do the dishes' }).save(
  success( task, data ),
  error( jqXHR )
) //=> taskDeferred
```

为了在服务器上创建一个模型的实例，首先使用 `new Model(attributes)` 创建一个实例，然后调用 `save()`。这个函数会检查是否该 task 已经有 ID 了，这个例子中是没有的，所以 `save()` 发起了一个 `create` 请求并传递该 task 的一些属性。它接受两个参数：

success

当 `save()` 请求成功后被回调的函数。调用此函数时会传入 task 实例和服务器返回的数据 `data`。

error

当请求发生错误时被回调的函数。调用此函数时会传入 jQuery 包装过的 XHR 对象。

`save()` 返回一个封装了被创建的 task 对象的延时对象。

获取任务

```
Task.findOne(params,
  success( task ),
  error( jqXHR )
) //=> taskDeferred
```

从服务器获取一个 task 需要三个参数：

params

传递给服务器的数据，通常是一个 ID，如 `{id: 2}`。

success

当请求成功后被回调的函数。调用此函数时会传入 task 实例。

error

当请求发生错误时被回调的函数。

findOne() 返回一个封装了 task 对象的延时对象。

195 获取多个任务

```
Task.findAll(params,  
  success( tasks ),  
  error( jqXHR )  
) //=> tasksDeferred
```

从服务器获取一个包含多个 task 的数组也需要 3 个参数：

params

传递给服务器的数据，通常是一个空对象 ({}) 或过滤器 {limit: 20, offset: 100}。

success

当请求成功后被回调的函数。调用此函数时会传入包含多个 task 实例的数组。

error

当请求发生错误时被回调的函数。

findAll() 返回一个延时对象，其中封装了一个包含多个 task 对象的数组。

更新任务

```
task.attr('name','take out recycling');  
task.save(  
  success( task, data ),  
  error( jqXHR )  
) //=> taskDeferred
```

为了更新服务器数据，首先需要使用模型实例的 attr 来改变属性，然后调用 save()。save() 的参数和返回值都与上面创建任务时一样。

销毁任务

```
task.destroy(  
  success( task, data ),  
  error( jqXHR )  
) //=> taskDeferred
```

从服务器上删除一个 task 需要 2 个参数：

success

当请求成功后被回调的函数。调用此函数时传入该 task 实例和服务器返回的数据 data。

error

当请求发生错误时被回调的函数。

与 save() 类似，destroy() 的返回值是一个封装了被销毁的 task 对象的延时对象。Task 模型基本上已经能为我们提供合格的服务了。

类型转换

196

你是否发现了服务器响应的 createdAt 的值，如 1303173531164 这样的数字，我们应该如何处理？这个数字实际上代表了 20011 年 4 月 18 日。如果不是从 task.createdAt 中直接得到这个数字，而是返回一个 JavaScript 日期对象，如 new Date(1303173531164)，这样将更加有用。当然，你也能用一个 setCreatedAt setter 来做这些事情，但如果有很多日期类型需要转换，很快就将产生大量重复性的工作。

为了让这些工作变得更简单，\$.Model 允许定义一个属性的类型，以及定义一个这些类型的转换函数。在静态的 attributes 对象上设置属性的类型，在静态的 convert 对象上设置转换方法，代码如下：

```
$.Model('Task',{
  attributes : {
    createdAt : 'date'
  },

  convert : {
    date : function(date){
      return typeof date == 'number' ? new Date(date) : date;
    }
  }
},{});
```

现在 Task 将 createdAt 转换成 Date 类型的数据。如果要枚举每个 task 对象的年份，代码如下：

```
Task.findAll({}, function(tasks){
  $.each(tasks, function(){
    console.log( "Year = "+this.createdAt.fullYear() )
  })
});
```

CRUD 事件

当模型的实例在被创建、更新或销毁时，模型将发布（触发）一些事件。你既可以在全局监听这些事件也可以在单个的模型实例上监听。监听创建、更新或销毁事件可以使用 `MODEL.bind(EVENT, callback(ev, instance))`。

我们想知道一个 `task` 何时被创建，因为之后才能将它添加到页面上。在添加以后，我们会监听该 `task` 的所有更新以便能正确地显示其名字。看看下面是如何实现的：

```
Task.bind('created', function(ev, task){
  var el = $('<li>').html(todo.name);
  el.appendTo($('#todos'));

  task.bind('updated', function(){
    el.html(this.name);
  }).bind('destroyed', function(){
    el.remove();
  })
});
```

197

在视图中使用客户端模板

JavaScriptMVC 的视图实质上就是客户端模板，它接收数据然后返回一个字符串。通常情况，返回的就是将要被插到 DOM 中的 HTML 字符串。

`$.View` 是一个模板接口，使用模板来降低复杂度。它提供如下支持：

- 方便、一致的语法。
- 从 HTML 元素或外部文件中加载模板。
- 同步或异步方式加载模板。
- 模板预加载。
- 缓存已处理的模板。
- 产品构建时批量处理模板。
- `$.Deferred`（延时对象）的支持。

JavaScriptMVC 打包了 4 种不同的模板引擎：

- EJS
- JAML
- Micro
- Tmpl

下面的例子用了 EJS 模板，但技术上各个模板引擎是通用的（只是在语法上有一些小区别）。

基本用法

在使用视图时，你总是会将模板渲染的结果插到页面中。jQuery.View 覆盖了 jQuery 的修改器，使用起来像下面这样方便：

```
$("#foo").html('mytemplate.ejs',{message: 'hello world'})
```

上面代码的工作流程如下。

1. 加载文件 *mytemplate.ejs* 中的模板。看起来如下：

```
<h2><%= message %></h2>
```

2. 应用 {message: 'hello world'} 来渲染，结果如下：

```
<h2>hello world</h2>
```

3. 将结果插到元素 *foo* 中以后，就变成：

```
<div id='foo'><h2>hello world</h2></div>
```

jQuery 修改器

198

以下 jQuery 修改器方法可以和模板一起协同使用：

```
$('#bar').after('temp.ejs',{});
$('#bar').append('temp.ejs',{});
$('#bar').before('temp.ejs',{});
$('#bar').html('temp.ejs',{});
$('#bar').prepend('temp.ejs',{});
$('#bar').replaceWith('temp.ejs',{});
$('#bar').text('temp.ejs',{});
```

用 Script 标签加载

视图可以从 script 标签或文件中加载。要从一个 script 标签加载视图，首先应创建一个 script 标签，并设置其 type 属性为模板的属性（text/ejs），用一个 id 来标识该模板：

```
<script type='text/ejs' id='recipesEJS'>
<% for(var i=0; i < recipes.length; i++){ %>
  <li><%=recipes[i].name %></li>
<%} %>
</script>
```

用该模板来渲染，例如：

```
$("#foo").html('recipesEJS', recipeData)
```

请注意，我们传递了想要渲染元素的 id。

\$.View 和子模板

有时我们只需要渲染的字符串，这种情况下，可以直接使用 `$.View(TEMPLATE, data)`。给 `$.View` 传递模板的路径和要渲染的数据：

```
var html = $.View("template/items.ejs", items );
```

其实，最常见的案例是子模板。通常，将单个项目模板和项目列表模板 (*items.ejs*) 隔离开。例如，用 `template/items.ejs` 为每个项目的渲染一个 `<>`，而使用 `template/item.ejs` 渲染每个项目内部的内容：

```
<% for( var i = 0; i < this.length; i++){ %>
  <li>
    <%= $.View("template/item.ejs", this[i]);
  </li>
< % } %>
```

`this` 指代的是传递给模板的数据。在 `template/items.ejs` 的示例代码中，`this` 是包含多个项目的数组。在 `template/item.ejs` 里，`this` 是包含单个项目的数组。

199 延时对象

当前使用 Ajax 请求数据并用模板来渲染结果是非常普遍的现象。使用前面 `$.Model` 一节中提到的 Task 模型，可以像下面这样来渲染 `#tasks`：

```
Task.findAll({}, function(tasks){
  $('#tasks').html("views/tasks.ejs" , tasks )
})
```

`$.View` 支持延时 (<http://api.jquery.com/category/deferred-object/>)，使用强大、简洁、高性能的语法。如果在传递给 `$.View` 的渲染数据中或 jQuery 的修改器中能找到一个延时对象，`$.View` 将异步加载模板，并等待所有延时结束，模板加载完成以后才开始渲染。

模型方法 `findAll`、`findOne`、`save` 和 `destroy` 等都返回延时对象。因此，我们可以用一行代码来重写 `task` 列表的渲染：

```
$('#tasks').html("views/tasks.ejs" , Task.findAll() )
```

有多个延时对象时也可以这样写：

```
$('#app').html("views/app.ejs" , {  
  tasks: Task.findAll(),  
  users: User.findAll()  
})
```

打包、预加载和性能

默认情况下，`$.View` 是同步加载模板的，这样做是因为它认为你：

- 把模板放置在 `script` 标签之内。
- 模板和 JavaScript 代码打包编译在一起。
- 预加载模板。

JavaScriptMVC 不推荐将模板放在 `script` 标签之内。`script` 标签模板在不同的 JavaScript 应用之间很难复用。而且，如果你的 app 不是立即就需要这些模板，它还会降低加载的性能。

JavaScriptMVC 推荐初始化打包时使用应用的 JavaScript 的模板，然后预加载后续要用到的模板。

StealJS、JavaScriptMVC 的编译系统可以处理和打包模板，并将其添加到一个缩减后的生产环境编译包里面。只须使用 `steal.views(PATH, ...)` 就成指定模板：

```
steal.views('tasks.ejs', 'task.ejs');
```

接下来，当 `$.View` 查找该模板时，它就能使用这份缓存的模板，而无须额外的 Ajax 请求。

对于不是立即需要使用的模板，可以使用 `jQuery.get` 来预加载并缓存起来。很简单，只要提供模板的 URL，以及一个叫做“view”的 `dataType`（最好在页面加载完成后的短时间内做这件事）：

```
$(window).load(function(){  
  setTimeout(function(){  
    $.get('users.ejs',function(){},'view');  
    $.get('user.ejs',function(){},'view');  
  },500)  
})
```

◀ 200

\$.Controller : jQuery 插件工厂

JavaScriptMVC 的控制器包含了很多东西。它们是一个 jQuery 插件工厂^{注1}，可以作为传统的视图使用，创建分页 widget 和栅格化控件；也可以作为传统的控制器使用，初始化控制器并绑定到模型上。大多数情况下，控制器是一种组织应用程序代码非常棒的方法。

控制器提供了一些非常易用的特性，如：

- 创建 jQuery 插件
- 自动绑定
- 默认选项值
- 自动判定

但是控制器最重要的特性并不是很明显。下面的代码用于创建一个类似 tooltip 的 widget，它会一直显示，在文档被点击时关闭：

```
$.fn.tooltip = function(){
    var el = this[0];

    $(document).click(function(ev){
        if (ev.target !== el)
            $(el).remove();
    });

    $(el).show();
    return this;
};
```

如何使用呢？直接将该元素添加到要显示的页面上，然后调用 tooltip：

```
$("<div class='tooltip'>Some Info</div>")
    .appendTo(document.body)
    .tooltip()
```

但上述代码是有问题的。你能指出问题的所在吗？提示一下：如果你的页面生命周期很长，有很多这样的 tooltip 被创建出来时会怎样呢？

201 问题就在于内存泄漏！每个 tooltip 元素及其子元素，都会永远保留在内存中。这是因为 click 事件处理句柄并没有从 document 中移除，它形成的闭包仍然在引用着该元素。

这是一个非常容易犯的错误。jQuery 对从页面上移除的元素本来会自动解除所有事件

注1：“插件工厂”英文为“Plug-in Factory”，实际上这个概念用在这里并不严谨，插件（Plug-ins）和组件（Widget）是有区别的，插件是可以安装亦可以卸载的，而组件更多的是创建出来但并不需要销毁它，所以插件包含“构造”和“析构”的概念，而组件只需“构造”，在 YUI 中插件和组件的概念是如此区分的。——译者注

处理句柄的绑定，而开发者一般不需要关心解绑的问题。但在该案例中，我们是在该 widget 的外面——document 中——所以并没有解除事件的绑定。

但在模型 - 视图 - 控制器的结构下，控制器监听视图，然后视图监听模型。我们经常是在 widget 的外面来监听事件。例如，在 \$.Model 一节中介绍的 nextPrev 这个 widget，是在 paginate 模型中监听更新的：

```
paginate.bind('updated.attr', function(){
    self.find('.prev')[this.canPrev() ? 'addClass' : 'removeClass']('enabled')
    self.find('.next')[this.canNext() ? 'addClass' : 'removeClass']('enabled');
})
```

但是它并没有从 paginate 中解绑事件！忘记移除事件处理句柄是一个潜在的错误根源。尽管如此，tooltip 和 nextPrev 都没有导致错误，它们只不过在悄悄地吞噬应用程序的性能。所幸的是，\$.Controller 让这些工作更简单、更有条理。tooltip 的代码可以这样写：

```
$.Controller('Tooltip',{
    init: function(){
        this.element.show()
    },
    "{document} click": function(el, ev){
        if(ev.target !== this.element[0]){
            this.element.remove()
        }
    }
})
```

当点击文档时，该元素从 DOM 中移除，\$.Controller 将自动解绑 document 的 click 事件处理句柄。

\$.Controller 对 nextPrev 也做了同样的事情，将其绑定到 Paginate 模型：

```
$.Controller('Nextprev',{
    ".next click" : function(){
        var paginate = this.options.paginate;
        paginate.attr('offset', paginate.offset+paginate.limit);
    },
    ".prev click" : function(){
        var paginate = this.options.paginate;
        paginate.attr('offset', paginate.offset-paginate.limit );
    },
    "{paginate} updated.attr" : function(ev, paginate){
        this.find('.prev')[paginate.canPrev() ? 'addClass' : 'removeClass']('enabled')
        this.find('.next')[paginate.canNext() ? 'addClass' : 'removeClass']('enabled');
    }
})
```

```
// 创建一个 nextprev 控制器
$('#pagebuttons').nextprev({ paginate: new Paginate() })
```

如果元素 `#pagebuttons` 从页面上移除了，`NextPrev` 控制器实例会自动将其从 `Paginate` 模型上解除绑定。

到目前为止对于 `error-free` 的代码，你的胃口刚好被适当地刺激了一下。接下来就详细地介绍 `$.Controller` 的工作原理。

概览

`$.Controller` 继承自 `$.Class`。要创建一个控制器类，调用 `$.Controller(NAME, classProperties, instanceProperties)` 函数，传入控制器的名字，静态方法和实例方法即可。如下的代码可作为创建一个可复用的列表 widget 的开始：

```
$.Controller("List", {
  defaults : {}
},{
  init : function(){ },
  "li click" : function(){ }
})
```

创建一个控制器类以后，它还用类似的名字创建了一个 jQuery 辅助方法。该辅助方法主要用于在页面的元素之上创建新的控制器实例。方法名就是控制器名字的下画线形式，每段都用下画线替换。例如，`$.Controller('App.Foo.Bar')` 的辅助函数就是 `$(el).app_foo_bar()`。

控制器实例化

要创建一个控制器实例，可以调用 `new Controller(element, options)`，传入一个 HTML 元素或者由 jQuery 包装的元素，以及另一个可选的对象来配置控制器。例如：

```
new List($('ul#tasks'), {model : Task});
```

也可以使用 jQuery 的辅助方法在 `#tasks` 元素上来创建一个 `List` 控制器实例：

```
$('#ul#tasks').list({model : Task})
```

当控制器创建以后，它将调用控制器的原型方法 `init`，并传入以下参数：

`this.element`

它是用 jQuery 包装的 HTML 元素。

`this.options`

它是传递给控制器的可选配置项与类本身的 `defaults` 对象合并以后的产物。

下面的代码是改进版的 List 控制器，从模型请求数据，并使用可选的模板渲染到列表中： 203

```
$.Controller("List", {
  defaults : {
    template: "items.ejs"
  }
}, {
  init : function(){
    this.element.html( this.options.template, this.options.model.findAll() );
  },
  "li click" : function(){ }
});
```

到此为止，我们可以提供一个模板来配置 List 控制器。挺灵活的吧！

```
$('#tasks').list({model: Task, template: "tasks.ejs"});
$('#users').list({model: User, template: "users.ejs"})
```

如果不提供模板，List 控制器会用默认的 *items.ejs*。

事件绑定

在 `$.Controller` 的介绍中提到过，它最强大的特性就是事件处理句柄的绑定和解绑的能力。

当创建了一个控制器之后，它会查找动作方法。动作方法看起来像事件处理句柄——例如，`"li click"`。这些动作会用到 `jQuery.bind` 或 `jQuery.delegate`。当从页面上移除和控制器关联的元素或者调用 `destroy` 时，控制器将会自己销毁，解除这些事件，防止内存泄漏。

下面给出一些动作及其所监听事件的例子：

`"li click"`

控制器元素内部的 `li` 之上或它内部的元素上发生点击事件。

`"mousemove"`

控制器元素内部发生鼠标移动事件。

`"{window} click"`

`window` 内部发生的点击事件。

回调动作函数的时候将传入一个用 jQuery 包装的元素或发生事件的对象, 以及事件本身。
例如：

```
"li click": function( el, ev ) {
    assertEquals(el[0].nodeName, "li" )
    assertEquals(ev.type, "click")
}
```

204

模板动作

\$.Controller 支持模板动作。模板动作可以绑定到其他对象上, 自定义事件类型或者自定义选择器。

控制器会替换动作的某些部分, 比如控制器选项中或在 window 中带一个值的 {OPTION}。

下面是一个菜单的骨架, 可以自定义针对不同的事件显示子菜单：

```
$.Controller("Menu",{
    "li {openEvent}" : function(){
        // 显示子孙
    }
});

// 创建一个菜单, 当点击子孙节点时显示出来
$("#clickMenu").menu({openEvent: 'click'});

// 创建一个菜单, 当鼠标划过子孙节点时显示出来
$("#hoverMenu").menu({openEvent: 'mouseenter'});
```

我们可以进一步增强该菜单的功能, 甚至允许自定义 menu 元素的便签：

```
$.Controller("Menu",{
    defaults : {menuTag : "li"}
},{
    "{menuTag} {openEvent}" : function(){
        // 显示子孙
    }
});

$("#divMenu").menu({menuTag : "div"})
```

模板动作让你能在控制器元素之外实现与元素或对象的绑定。例如, 在 \$.Model 一节中介绍过的 Task 模型会在一个新的 Task 创建之时产生一个 “created” 事件。可以让该列表 widget 监听 task 的创建, 然后自动将它们添加到列表中：

```
$.Controller("List", {
    defaults : {
```

```

        template: "items.ejs"
    },
    {
        init : function(){
            this.element.html( this.options.template, this.options.model.findAll() );
        },
        "{Task} created" : function(Task, ev, newTask){
            this.element.append(this.options.template, [newTask])
        }
    }
})

```

"{Task} created" 动作函数会被回调，并传入该 Task 模型、created 事件和刚创建的 Task。函数使用模板来渲染 task 的列表（这里只有一个），并将生成的 HTML 添加到该元素中。

◀ 205

但是如果能将 List 改进为能与任意模型协调工作就更好了。不必去硬编码实现 tasks，我们让控制器以可选项的形式接受一个模型：

```

$.Controller("List", {
    defaults : {
        template: "items.ejs",
        model: null
    }
}, {
    init : function(){
        this.element.html( this.options.template, this.options.model.findAll() );
    },
    "{model} created" : function(Model, ev, newItem){
        this.element.append(this.options.template, [newItem])
    }
});

// 创建一个任务列表
$('#tasks').list({model: Task, template: "tasks.ejs"});

```

大综合：一个抽象的 CRUD 列表

现在我们将进一步改进该列表，使得不仅当项目创建后能自动添加进列表，而且能够自动更新并在项目销毁后移除它们。为此，首先我们需要监听 updated 和 destroyed 事件：

```

"{model} updated" : function(Model, ev, updatedItem){
    // 找到有更新的项并更新 LI
},
"{model} destroyed" : function(Model, ev, destroyedItem){
    // 找到待删除的项，并将 LI 移除
}

```

你会发现一个问题。我们需要以某种方式去找到表示特定模型实例的元素。为此，我们需要标记该元素，说明它是属于该模型实例的。所幸的是，`$.Model` 和 `$.View` 都做了处理，使得标记元素属于某个实例以及如何找到这些元素都非常简单。

要在 EJS 视图中用某个模型实例标记元素，只需将模型实例写到元素上。`tasks.ejs` 的代码可能如下：

```
<% for(var i =0 ; i < this.length; i++){ %>
  <% var task = this[i]; %>
  <li <%= task %> > <%= task.name %> </li>
<% } %>
```

`tasks.ejs` 迭代 `task` 列表，对于每个 `task`，它创建一个 `li` 元素并将其内容设置为 `task` 的 `name`。但同时，它也使用了 `<li><%= task %></li>` 将该 `task` 添加到元素的 jQuery 数据中。

206 为了在后面通过给定的模型实例得到元素，可以调用 `modelInstance.elements([CONTEXT])`，它将返回 jQuery 包装的表示该模型实例的元素。

整合起来，列表就出来了：

```
$.Controller("List", {
  defaults : {
    template: "items.ejs",
    model: null
  }
},{
  init : function(){
    this.element.html( this.options.template, this.options.model.findAll() );
  },
  "{model} created" : function(Model, ev, newItem){
    this.element.append(this.options.template, [newItem])
  },
  "{model} updated" : function(Model, ev, updatedItem){
    updatedItem.elements(this.element)
      .replaceWith(this.options.template, [updatedItem])
  },
  "{model} destroyed" : function(Model, ev, destroyedItem){
    destroyedItem.elements(this.element)
      .remove()
  }
});

// 创建一组任务列表
$('#tasks').list({model: Task, template: "tasks.ejs"});
```

使用 JavaScriptMVC 来创建抽象、可复用、内存安全的 widget 真是简单得要命，不是吗？

jQuery基础

现在已经有很多类库让开发者能够更加容易地处理 DOM，但流行度和赞誉很少有能与 jQuery 比肩的。理由：jQuery 的 API 非常优秀，轻量，基于命名空间，不会和你使用的任何东西冲突。而且，jQuery 非常容易扩展；已经有人开发出了一整套 plug-in (<http://plugins.jquery.com/>)，从 JavaScript 校验到进度条统统都有。

jQuery 的命名空间是在 jQuery 这个变量下面的，它的别名是一个美元符号“\$”。和其他类库如 Prototype (<http://prototypejs.org/>) 不同的是，jQuery 没有扩展任何 JavaScript 原生的对象，最大程度上避免和其他类库的冲突。

关于 jQuery 需要理解的另一个重点是选择器。如果你熟悉 CSS，对于选择器你应该也不陌生。所有 jQuery 的实例方法都运行在选择器之上，故除了在元素上迭代遍历，还可以就用选择器来获取它们。在 jQuery 选择器上调用的函数都会在每个被选中的元素上执行。

下面通过一个例子来说明 jQuery 的这些特点。在所有类名为 foo 的元素上增加一个值为 selected 的类名。第一段代码完全使用 JavaScript，第二段使用 JQuery：

```
// 纯 JavaScript 代码
var elements = document.getElementsByClassName("foo");
for (var i=0; i < elements.length; i++) {
    elements[i].className += " selected";
}

// jQuery 代码
$(".foo").addClass("selected");
```

因此，从上例中你可以看到 jQuery 的选择器 API 极大地降低了代码量。我们再来深入看一下这些选择器。就如同在 CSS 中会使用哈希 (“#”) 来通过 ID 选择元素一样，你可以在 jQuery 中这么做：

```
// 通过 ID(wem) 选择元素
var element = document.getElementById("wem");
var element = $("#wem");

// 通过 class(bar) 选择所有元素
var elements = document.getElementsByClassName("bar");
var elements = $(".bar");

// 通过标签(p) 选择所有元素
var elements = document.getElementsByTagName("p");
var elements = $("p");
```

就像在 CSS 中一样，你还可以用复合选择器来选择更具体的元素：

```
// 选择 .bar 中的 .foo
var foo = $(".bar .foo");
```

甚至使用属性选取一个元素：

```
var username = $("input[name='username']");
```

或者，选择第一个匹配的元素：

```
var example = $(".wem:first");
```

每当我们在选择器上调用一个函数，在所有选中的元素上都会起作用：

```
// 给所有的 .foo 元素添加一个名为 'bar' 的 class
$(".foo").addClass("bar");
```

前面提到过，所有 jQuery 的函数都是基于命名空间的，所以如果在一个 DOM 元素上直接调用它们就会失败：

```
// 这段代码会失败
var element = document.getElementById("wem");
element.addClass("bar");
```

相反，如果想用 jQuery 的 API，首先要用 jQuery 实例将该元素包装起来：

```
var element = document.getElementById("wem");
$(element).addClass("bar");
```

DOM 遍历

要是你已经选择了一些元素，jQuery 提供了许多方法（<http://api.jquery.com/category/traversing/>）来在选择器中找到与之相关的其他元素：

```
var wem = $("#wem");
```

```
// 找到匹配的子节点
wem.find(".test");

// 选择父节点
wem.parent();

// 或者，得到一组父节点，通过传入一个指定的选择器来匹配父节点
wem.parents(".optionalSelector");

// 选择（第一个元素的）直接的子节点
wem.children();
```

或者，也可以在选择器内部遍历这些元素：

```
var wem = $("#wem");

// 通过指定索引位置 0 来返回元素
wem.eq( 0 );

// 返回第一个元素，即 $.fn.eq(0)
wem.first();

// 通过传入一个选择器来过滤得到所匹配的元素
wem.filter(".foo");

// 通过传入一个匹配函数来过滤所需要的元素
wem.filter(function(){
    // this, 即当前的元素
    return $(this).hasClass(".foo");
});

// 返回元素的匹配给定选择器的子孙节点
wem.has(".selected");
```

jQuery 也有一些迭代器，`map()` 和 `each()`，它们接受一个回调：

```
var wem = $("#wem");

// 将选中的每个元素都传入一个函数
// 根据返回值来构建一个新数组
wem.map(function(element, index){
    assertEquals(this, element);

    return this.id;
});

// 使用一个回调函数来遍历每个选中的元素，和 `for` 循环等价
wem.each(function(index, element){
    assertEquals(this, element);
```

```

    /* ... */
  });

```

同时还可以为选择器手动添加元素：

```

// 将所有的 p 元素添加到给定的选择器上
var wem = $("#wem");
wem.add( $("p") );

```

210

DOM 操作

然而，jQuery 不仅仅是选择器；它还有一个强大的 API 与 DOM 接口并操作 DOM。除了选择器，jQuery 的构造器接受 HTML 标签输入，并生成新的元素：

```

var element = $("p");
element.addClass("bar")
element.text("Some content");

```

在 DOM 中添加新元素也非常简单——只要使用 jQuery 的 `append()` 或 `prepend()` 函数即可。出于性能方面的考虑，最好在新元素添加到 DOM 之前对它们做一些操作：

```

// 追加一个元素
var newDiv = $("<div />");
$("body").append(newDiv);

// 将元素作为第一个子节点添加到母节点中
$(".container").prepend($("<hr />"));

```

或者，在另一个元素之前 / 之后插入一个元素：

```

// 在元素之后插入节点
$(".container").after( $("<p />") );

// 在元素之前插入节点
$(".container").before( $("<p />") );

```

删除元素也很简单：

```

// 删除元素
$("#wem").remove();

```

那改变元素的属性呢？jQuery 也提供同样的支持。例如，使用 `addClass()` 函数为元素添加类名：

```

$("#foo").addClass("bar");

// 移除一个 class
$("#foo").removeClass("bar");

```

```
// 判断元素是否包含这个 class
var hasBar = $("#foo").hasClass("bar");
```

设置和获取 CSS 样式也同样相当简单。css() 函数既可以作为 getter 也可以作为 setter，这个取决于传递给它参数类型：

```
var getColor = $(".foo").css("color");

// 设置颜色样式
$(".foo").css("color", "#000");

// 或通过传入对象来对多个样式设置值
$(".foo").css({color: "#000", backgroundColor: "#FFF"});
```

jQuery 还提供一些最常用的变更样式的快捷方法：

211

```
// 隐藏元素
$(".bar").hide();

// 显示元素
$(".bar").show();
```

或者，如果希望逐渐改变不透明度，可以如下操作：

```
$(".foo").fadeOut();
$(".foo").fadeIn();
```

jQuery 的 getter 和 setter 函数不仅限于 CSS。例如，可以使用 html() 函数来获取和设置元素的内容：

```
// 得到这个选择器中第一个元素的 HTML 代码
var getHTML = $("#bar").html();

// 为所选择的元素设置 HTML 代码
$("#bar").html("<p>Hi</p>");
```

text() 函数也是一样道理，只是它的参数内容会被转义：

```
var getText = $("#bar").text();

$("#bar").text("Plain text contents");
```

最后，删除一个元素的所有子元素，只要使用 empty() 函数：

```
$("#bar").empty();
```

事件

曾经有一段时期，在浏览器中处理事件十分混乱，导致 API 各不相同，缺少一致性。jQuery 为我们解决了这个问题，提供了一个统一的 API 消除了各种浏览器之间的差异。下面简要介绍 jQuery 的事件处理，但想要参考更多的信息，请看第 2 章以及 jQuery 的官方文档 (<http://api.jquery.com/category/events/>)。

要添加事件处理句柄，可以使用 `bind()` 函数同时传入事件类型和回调函数作为参数：

```
$(".clicky").bind("click", function(event){
    // 发生点击事件时执行这里的代码
});
```

jQuery 为常用的事件提供了一些快捷方法，除了调用 `bind()` 函数，还可以这样做：

```
$(".clicky").click(function(){ /* ... */ });
```

还有一个事件你可能经常会用到：`document.ready`。此事件在页面加载过程中触发，这时 DOM 已完成但有一些元素如图片等尚未加载完。jQuery 提供了一个优雅的方式——直接给 jQuery 对象传递一个函数即可：

212

```
jQuery(function($){
    // document.ready 时执行这里的代码
});
```

经常让 jQuery 新手迷惑的地方就是回调中上下文的改变。例如，在上面的例子中，回调的上下文变成指向该元素，此处就是 `$(".clicky")`：

```
$(".clicky").click(function(){
    // 'this' 指向事件的目标对象
    assert( $(this).hasClass(".clicky") );
});
```

如果在回调中使用 `this` 就会导致这种上下文变化的问题。习惯用法就是将上下文保存到一个局部变量中，常常命名为 `self`：

```
var self = this;
$(".clicky").click(function(){
    self.clickedClick();
});
```

另一种方法就是使用 `jQuery.proxy()` 代理函数将回调包装起来，如下：

```
$(".clicky").click($.proxy(function(){
    // 上下文不会改变
}, this));
```

想了解事件委托和上下文更多的信息，请参考第 2 章。

Ajax

Ajax 或 XMLHttpRequest，在各个浏览器之间的实现也是差别很大的。jQuery 也将它进一步抽象，消除了这些差别，为我们提供了一个漂亮的 API (<http://api.jquery.com/category/ajax>)。在第 3 章中详细介绍了 jQuery 的 Ajax 接口 API，这里再简单概述。

jQuery 有一个底层的函数叫 `ajax()`，同时提供了几个高层的抽象。`ajax()` 函数接受一个哈希对象来设置一些选项，如端点 `url`，请求的类型 `type`，还有成功后的 `success` 回调函数：

```
$.ajax({
  url: "http://example.com",
  type: "GET",
  success: function(){ /* ... */ }
});
```

尽管如此，jQuery 提供的快捷方法使得 API 调用更加简洁：

```
$.get("http://example.com", function(){ /* on success */ })
$.post("http://example.com", {some: "data"});
```

jQuery 的 `dataType` 选项告诉 jQuery 如何处理 Ajax 响应。如果不指定此选项，默认情况下，jQuery 通过响应头信息做一个智能的推测。如果你知道响应的类型，最好能明确地设置：

```
// 请求一个 JSON
$.ajax({
  url: "http://example.com/endpoint.json",
  type: "GET",
  dataType: "json",
  success: function(json){ /* ... */ }
});
```

213

jQuery 还提供请求常用数据类型的快捷方法，如 `getJSON()`，就等同于上面的 `ajax()` 函数调用：

```
$.getJSON("http://example.com/endpoint.json", function(json){ /* .. */ });
```

想更深入地分析理解 jQuery 的 Ajax API 中的选项，请参考第 3 章以及其官方的文档。

做个好市民

jQuery 为自己能做个 web 好市民而感到非常自豪；例如，它完全基于命名空间，没有污染全局作用域。但其他类库也经常使用 jQuery 对象的别名 \$，如 Prototype (<http://www.prototypejs.org/>)。因此，为了避免这种冲突，应该使用 jQuery 的 noConflict 模式，来改变它的别名并释放 \$：

```
var $J = jQuery.noConflict();

assertEqual( $, undefined );
```

当你在写 jQuery 扩展时，应该假设 jQuery 的 noConflict 模式是打开的，也就是说 \$ 并不指代 jQuery 对象。然而，在实际情况中，\$ 是非常有用的快捷变量，因此最佳实践是保证它是个局部变量：

```
(function($){

    // $ 是局部变量
    $(".foo").addClass("wem");

})(jQuery);
```

为了简化代码，jQuery 在 *document.ready* 事件中将其本身作为参数传递进来：

```
jQuery(function($){
    // 当页面加载完成后执行这里的代码
    assertEqual( $, jQuery );
});
```

扩展

再也没有比扩展 jQuery 更简单的事情了。如果要添加类函数，只要直接在 jQuery 对象上创建函数即可：

```
214 jQuery.myExt = function(arg1){ /*...*/ };

// 这样就可以使用它了
$.myExt("anyArgs");
```

如果要添加函数实例，并且在元素的选择器上有效，只要在 *jQuery.fn* 对象上创建函数即可，它其实就是 *jQuery.prototype* 的别名。最佳实践是在扩展的最后返回当前上下文（也就是说，*this*），这样就能链式调用了：

```
jQuery.fn.wemExt = function(arg1){
    $(this).html("Bar");
```

```

    return this;
};

$("#element").wemExt(1).addClass("foo");

```

另一个最佳实践就是将扩展用模块的模式 (<http://goo.gl/czZm>) 来封装, 这可以避免作用域泄漏和变量冲突。将扩展包装在一个匿名函数中, 所有的变量都是局部的:

```

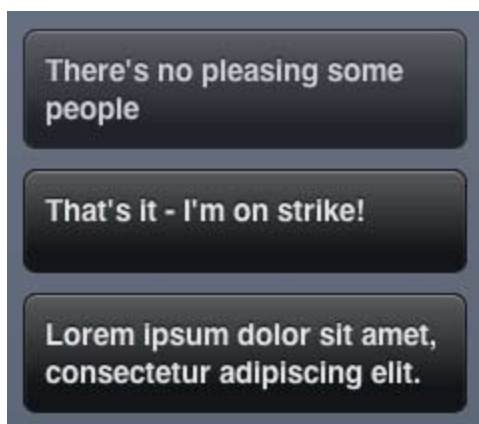
(function($){
    // 本地上下文在此处
    var replaceLinks = function(){
        var re = /((http|https|ftp):\/\/[^\w?=&.\\/-;#~%- ]+(?![\w\s?&.\\/;#~%"]=-]*>))/g;
        $(this).html(
            $(this).html().replace(re, '<a target="_blank" href="$1">$1</a> ');
        );
    };

    $.fn.autolink = function() {
        return this.each(replaceLinks);
    };
})(jQuery);

```

构建一个 Growl 的 jQuery 插件

将我们学到的 jQuery 知识应用于实际, 创建一个 Growl 库试试。对于不熟悉 Growl 的人来说, 先介绍一下: Growl 是 Mac OS X 的通知类库 (<http://growl.info/>), 应用程序可以用它在桌面上显示消息而不会让用户感到很冒失。接下来, 我们将来模拟该 OS X 类库, 在页面中用 JavaScript 来显示消息, 如图 A-1 所示。



图A-1: Growl消息举例

首先创建一个 `#container` 的 `div`，我们所有的消息元素都是它的后代。如你所见，我们引入了 `jQuery` 和 `jQuery UI` 类库——后面将使用后者来添加一些效果。当页面加载完，添加名为 `container` 的 `div`：

```
//= require <jquery>
//= require <jquery.ui>

(function($){
  var container = $("

下面是插件的逻辑部分。每当有一个新消息，我们就给 container 元素添加一个 div。给消息添加一个 drop 效果，一段时间以后，淡出，并且将它删除——就像 OS X 里 Growl 的行为一样：



```
$.growl = function(body){
 // 创建这个 Growl 元素
 var msg = $("

以上就是所需的全部 JavaScript 代码。乍看上去还是有些丑陋，可以用 CSS3 修饰一下。我们想让 #container 绝对定位在页面的右下角：

236 | 附录A jQuery基础


```


```

```
#growl {
  position: absolute;
  bottom: 10px;
  right: 20px;
  overflow: hidden;
}
```

接下来给消息元素添加一些样式。我很喜欢 Growl 的 HUD 主题，试着模拟一下。使用 `rgba` 让背景有一点点透明，然后添加一个嵌入状的 `box-shadow`，让元素呈现一种高光的效果：

```
#growl .msg {
  width: 200px;
  min-height: 30px;
  padding: 10px;
  margin-bottom: 10px;

  border: 1px solid #171717;
  color: #E4E4E4;
  text-shadow: 0 -1px 1px #0A131A;
  font-weight: bold;
  font-size: 15px;

  background: #141517;
  background: -webkit-gradient(
    linear, left top, left bottom,
    from(rgba(255, 255, 255, 0.3)),
    color-stop(0.8, rgba(255, 255, 255, 0))),
    rgba(0, 0, 0, 0.8);

  -webkit-box-shadow: inset 0 1px 1px #8E8E8E;
  -moz-box-shadow: inset 0 1px 1px #8E8E8E;
  box-shadow: inset 0 1px 1px #8E8E8E;

  -webkit-border-radius: 7px;
  -moz-border-radius: 7px;
  border-radius: 7px;
}
```

上面就是全部的内容。可以看到创建 jQuery 插件是多么地简单。还有一些其他的例子，请在这里 [assets/appA/growl.html](#) 查看完整版源代码。

CSS扩展

正如 Less (<http://lesscss.org/>) 的作者 Alexis Sellier 所说, “Less 是一种建立在 CSS 的语法之上的动态样式表语言”。Less 是 CSS 的一个超集, 它扩展出了变量、混合、操作符和嵌套规则。

Less 的伟大就在于它确实能减少需要书写的 CSS 代码的数量——特别是遇到 CSS3 一些具有厂商特殊性的规则时。你可以将 Less 文件编译为纯 CSS。

换言之, 原来需要这样写的代码:

```
.panel {  
  background: #CCC;  
  background: -webkit-gradient(linear, left top, left bottom, from(#FFF), to(#CCC));  
  background: -moz-linear-gradient(top, #FFF, #CCC);  
}
```

现在可以这样写:

```
.panel {  
  .vbg-gradient(#FFF, #CCC);  
}
```

变量

如果你正在重复使用一些颜色和规则属性, 使用 Less 的变量可以将它们合并到一个地方, 需要时全局改动即可而不用查找与替换了!

设置变量很简单:

```
@panel-color: #CCC;
```

然后，就可以在样式规则中使用：

```
header {  
  color: @panel-color;  
}
```

218

混合

Less 的混合行为和 C 的宏有很多相似之处。基本上定义了一个混合（mixin）后，它可以接受可选的参数，像这样：

```
.vbg-gradient(@fc: #FFF, @tc: #CCC) {  
  background: @fc;  
  background: -webkit-gradient(linear, left top, left bottom, from(@fc), to(@tc));  
  background: -moz-linear-gradient(top, @fc, @tc);  
  background: linear-gradient(top, @fc, @tc);  
}
```

上面的例子接受两个参数，fc 和 tc，并且默认值分别为 #FFF 和 #CCC，然后把它们插到类的内容里面。可以把这个过程想象为定义一个变量，但这里只对整个类有效。

CSS3 还没有完成标准化，因此浏览器厂商通常指定它们自己的前缀，如 `-webkit` 和 `-moz`。从某方面来说这是好事，因为我们马上可以开始使用这些特性；但这种语法却显得很累赘，定义样式时为不同的浏览器你需要书写两倍或三倍数量的代码。

你可能猜到了，Less 的确能够削减需要书写的代码量——只要将各厂商特有的样式变为一个混合。

下面给出一些可能有用的混合例子：

```
/* 圆角 */  
.border-radius(@r: 3px) {  
  -moz-border-radius: @r;  
  -webkit-border-radius: @r;  
  border-radius: @r;  
}  
  
/* 阴影 */  
.box-shadow (@h: 0px, @v: 0px, @b: 4px, @c: #333) {  
  -moz-box-shadow: @h @v @b @c;  
  -webkit-box-shadow: @h @v @b @c;  
  box-shadow: @h @v @b @c;  
}
```

嵌套规则

获取元素时不用再指定很长的选择器名字了，转而使用嵌套选择器。完整的选择器会在后台生成，但嵌套规则使得样式表更加干净和易读：

```
button {  
  .border-radius(3px);  
  .box-shadow(0, 1px, 1px, #FFF);  
  .vbg-gradient(#F9F9F9, #E3E3E3);  
  
  :active {  
    .vbg-gradient(#E3E3E3, #F9F9F9);  
  }  
}
```

219

提醒一点，我不建议使用超过两层的嵌套，因为如果不注意很可能会被滥用，结果你的样式表将变成有史以来最糟糕的东西。

包含其他样式表

如果你正在计划拆分样式表（我强烈建议这样做），使用 `@import` 在当前文件中包含其他样式表。Less 将获取这些文件并合并插入，避免客户端发起单独的 HTTP 请求而改善性能。

这个方法也经常和混合一起使用。假设有一个 CSS3 混合文件，可以这样导入：

```
@import "utils";
```

颜色

Less 有一个尚未配备文档的很新的特性，但是很有用，值得在此说明。Less 可以使用各种函数来调节颜色：

```
background: saturate(#319, 10%);  
background: desaturate(#319, 10%);  
background: darken(#319, 10%);  
background: lighten(#319, 10%)
```

很多设计使用相同的颜色，但它们会使用不同的色调。事实上和变量组合使用，可以很快得到不同主题。

如何使用 Less

把 Less 代码编译成 CSS 有很多种方法。

命令行

安装 Less gem，然后调用 `lessc` 命令：

```
gem install less
lessc style.less
```

220 Rack

如果你正在使用基于 Rack 的框架，如 Rails 3，有一个更简单的方案：`rack-less gem`。只要你的文件中包含相应的 gem 即可：

```
gem "rack-less"
```

然后，将中间层注入到 `application.rb`：

```
require "rack/less"
config.middleware.use "Rack::Less"
```

所有在 `/app/stylesheets` 目录下的 Less 样式表都会被自动编译。你甚至可以通过 `production.rb` 配置文件设置 `rack-less` 以后，缓存和压缩编译的结果：

```
Rack::Less.configure do |config|
  config.cache      = true
  config.compress   = :yui
end
```

JavaScript

对 Ruby 库的开发似乎已经慢下来了，但所幸的是有一个更新的选择：`Less.js` (<http://github.com/cloudhead/less.js>)，它是 JavaScript 版本的 Less。你可以在页面中指定 Less 样式表，然后包含 `less.js`，它将自动编译这些 Less 样式表：

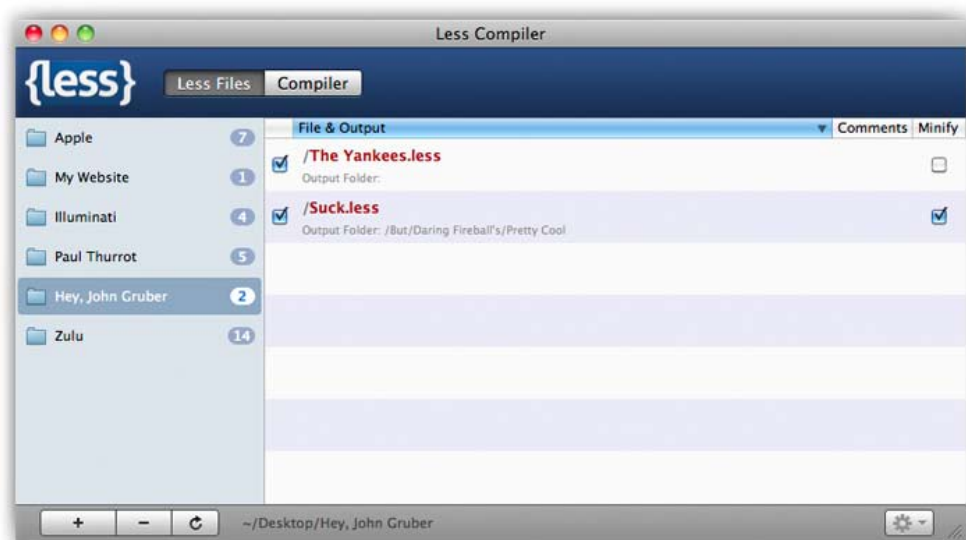
```
<link rel="stylesheet/less" href="main.less" type="text/css">
<script src="less.js" type="text/javascript"></script>
```

`Less.js` 比 Ruby 版的要快 40 倍。尽管如此，你可能还是想预编译 Less 样式表以避免客户端的性能问题。如果你安装了 `Node.js`，可以用命令行来编译：

```
node bin/lessc style.less
```

Less.app

Max OS X 应用程序 (<http://incident57.com/less/>) 让 Less 更好用了。它在后台使用了 Less.js，可以指定“监控”某些文件夹——也就是说，当 Less 样式表保存时会自动被编译成 CSS 格式，如图 B-1 所示。



221

图B-1：用Less.app自动编译Less文件

CSS3参考

在 CSS2.1 之下要做出优美的用户接口比较难，因为通常需要用到很多额外的标签、图片和 JavaScript 等。CSS3 试图解决这些问题，它提供了很多有用的属性和选择器来帮助你创建炫酷的用户接口。

在设计 Web 应用程序时，我常常跳过 Photoshop，而直接投向 CSS3 和 HTML5 的怀抱。鉴于这些强大的技术，再设计静态的 PSD 原型感觉有点多余了。客户也更欣赏这种方式，因为他们能够和产品的 HTML 原型进行交互，用户体验方面得到了更好的感受。

与此同时我也听到你的哭喊，“旧版本的浏览器怎么办呢？”使用 CSS3 的黄金时间还没有到来吗？好吧，问题的答案就是优雅降级。旧版本的浏览器会忽略 CSS3 样式，回退到标准的样式。例如，在 Chrome 中，用户将看到应用程序全部的光晕、渐变等所有的效果，但在 IE7 中，它就不会那么漂亮，只是提供所有的基本功能。

对于 IE6，我主张一同放弃支持吧！Facebook、Amazon 和 Google 都开始放弃支持，少量的 IE6 用户不用那么努力支持也是可行的。Web 在发展，旧的技术应该被抛弃。

主流浏览器有 IE、Firefox、Chrome 和 Safari。Chrome 和 Safari 有相同的渲染引擎叫 WebKit，但它们的 JavaScript 引擎是不一样的。两个浏览器之间只有一些细微的差别——它们使用不同的图形库——在 Chrome 中的修复会推进到 WebKit，反之亦然。

在本书撰写之时微软发布了 IE9。希望很快它能被大众所接纳，因为对比以前的版本它有一个很大的进步，甚至包含了对 CSS3 的很多支持。

作为一名前端工程师，当下是一个难以置信的令人兴奋的时刻，你应该考虑立即使用这些新的技术。

前缀

浏览器厂商以前就一直在实施 CSS3，但它还未成为真正的标准。为此，当有一些 CSS3 样式语法还在波动时，它们提供了针对浏览器的前缀。例如，CSS3 渐变样式在 Firefox 和 Safari 中是不同的。Firefox 使用 `-moz-linear-gradient`，而 Safari (WebKit) 使用 `-webkit-gradient`，这两种语法都使用厂商类型的前缀。可以使用的不同前缀如下：

- Chrome : `-webkit-`
- Safari : `-webkit-`
- Firefox : `-moz-`
- IE : `-ms-`
- Opera : `-o-`

因此，应该先用有厂商前缀的指定样式，紧接着使用无前缀的。这样可以保证当浏览器移除了前缀，使用标准 CSS3 规范时，你写的样式仍然有效：

```
#prefix-example {
  -moz-box-shadow: 0 3px 5px #FFF;
  -webkit-box-shadow: 0 3px 5px #FFF;
  box-shadow: 0 3px 5px #FFF;
}
```

颜色

CSS3 提供了几种指定颜色的新方法，包括 Alpha 透明度。创建透明颜色的老方法一般是使用 `1px × 1px` 的背景图片，但现在你可以将它抛在脑后了。

`rgb` 的样式可以指定颜色的红、绿和蓝字段——三原色——而不是传统的十六进制数值。使用 Safari Web Inspector 很容易在两种方法之间转换——只要在 Styles 选项中点击一个颜色即可。

下面的例子等同于十六进制的 `#FFF`——也就是白色：

```
#rgb-example {
  //      rgb(red, green, blue);
  color: rgb(255, 255, 255);
}
```

也可以使用 `hsl` 样式，它代表了色度、饱和度和亮度。

HSL 有三个值：

色度

色轮的度数。0（或 360）代表红色，120 代表绿色，240 代表蓝色，介于这 3 个值之间的数值代表了不同的颜色。

饱和度

百分比数值。100% 完全显示该颜色。

亮度

百分比数值。0% 代表暗（黑色），100% 代表亮（白色），50% 就是平均值。

为 rgb 或 hsl 增加 Alpha 透明度也很简单——只要使用 rgba 或 hsla 即可。Alpha 透明度是一个 0 介于（透明）到 1（不透明）之间的数值。

```
#alpha-example {
  background: hsla(324, 100%, 50%, .5);
  border: 1em solid rgba(0, 0, 0, .3);
  color: rgba(255, 255, 255, .8);
}
```

浏览器支持情况如下。

- Firefox：全部支持
- Chrome：全部支持
- Opera：全部支持
- Safari：全部支持
- IE：全部支持

圆角

在 CSS2.1 中做圆角是件吃力的工作，常常需要很多额外的标签、多张图片，甚至要用到 JavaScript。

现在就简单多了——只需要应用 border-radius 样式即可。与 padding 和 margin 样式用法一样，可以为不同的角分别指定不同的角半径，或者用两个值来分别指定水平和垂直的角半径，或者用一个值来指定所有的角半径。如果指定的半径值足够大，甚至能够创建一个圆形：

```
border-radius: 20px;
```

```
// 水平, 垂直
```

```
border-radius: 20px 20px;

// 左上, 右上, 右下, 左下
border-radius: 20px 20px 20px 20px;
```

浏览器支持情况如下。

226

- Firefox : 全部支持
- Chrome : 全部支持
- Safari : 使用 -webkit-
- IE >= 9.0 : 全部支持
- Opera : 全部支持

下拉阴影

在 CSS3 之前, 很多人不曾被下拉阴影困扰过, 因为它非常难做。不过, CSS3 提供了 `box-shadow` 样式, 使用它实现起来就轻而易举了。但是也不要用过火了, 那样只会让界面看上去很扎眼; 而且下拉阴影也非常消耗性能。

`box-shadow` 接受一些选项: 水平偏移量、垂直偏移量、模糊半径、可选的延伸距离和颜色。如果提供 `inset` 选项, 该阴影会在元素内部绘制; 否则, 默认情况是在元素外部绘制的。并且你可以指定多个阴影, 只要使用逗号分隔即可, 如以下代码所示:

```
// 水平偏移量, 垂直偏移量, 模糊直径, 颜色
box-shadow: 10px 5px 15px #000;

// 内阴影
box-shadow: 10px 5px 15px #000 inset;

// 水平偏移量, 垂直偏移量, 模糊直径, 延展距离, 颜色
box-shadow: 10px 5px 15px 15px #000;

// 多阴影
box-shadow: 0 1px 1px #FFF inset, 5px 5px 10px #000;
```

设计师通常在他们的作品中指定一个光源, 让界面看起来更加有型和生动。使用 `box-shadow` 就很容易实现——只要指定 `1px` 白色嵌入阴影即可。这个例子中, 光源处于页面的上方, 为了保持样式的一致性。代码如下:

```
#shadow-example {
  -moz-box-shadow: 0 1px 1px #FFF inset;
  -webkit-box-shadow: 0 1px 1px #FFF inset;
  box-shadow: 0 1px 1px #FFF inset;
}
```

浏览器支持情况如下。

- Firefox : 全部支持
- Chrome : 使用 -webkit-
- Safari : 使用 -webkit-
- IE >= 9.0 : 全部支持
- Opera : 全部支持

文本阴影

在 CSS3 之前，实现文本阴影的惟一方法就是用图片替换——一种不入流的解决方案。CSS3 使用 `text-shadow` 样式为文本增加阴影。只要传入水平偏移量、垂直偏移量、可选的模糊半径和颜色即可：

```
// 水平偏移量，垂直偏移量，颜色
text-shadow: 1px 1px #FFF;
```

```
// 水平偏移量，垂直偏移量，模糊直径，颜色
text-shadow: 1px 1px .3em rgba(255, 255, 255, .8);
```

文本阴影和 `box-shadow` 不一样，因为前者并没有延伸距离或嵌入阴影。但你还是可以通过设置文本阴影的位置为 `inset` 或 `outset` 来欺骗用户的眼睛。如果阴影的垂直偏移量为负值，并且在文本之上，那么它看起来就是嵌入的。相应的，如果阴影在文本之下，它看起来是外折的。

浏览器支持情况如下。

- Firefox : 全部支持
- Chrome : 全部支持
- Safari : 全部支持
- IE : 不支持
- Opera : 全部支持。

渐变

以前，渐变是通过背景图片的重复平铺来实现的。这意味着它们必须有固定的宽度或高度，为此需要打开图片编辑器来修改。

CSS3 增加了对线性和径向性渐变的支持，这是它最有用的特性之一。还有一些 CSS 函数你可以用来生成渐变，可以在任何需要用到颜色的地方使用它们。

对于线性渐变，只要给 `linear-gradient` 函数传递一个你希望的颜色值列表即可：

```
linear-gradient(#CCC, #DDD, white)
```

默认情况下，渐变是垂直方向的；不过也可以传递一个位置参数来改变方向：

```
// 水平渐变
linear-gradient(left, #CCC, #DDD, #FFF);

// 或者给定一个倾斜角度
linear-gradient(-45deg, #CCC, #FFF)
```

228 如果需要更多地控制渐变开始过渡的位置，可以使用颜色停止值。只要指定颜色值的同时指定一个百分比或像素值即可：

```
linear-gradient(white, #DDD 20%, black)
```

还可以从透明色开始过渡或过渡到透明色：

```
radial-gradient( rgba(255, 255, 255, .8), transparent )
```

Safari 当前还有一个差异较大的语法。不过，它马上也将皈依标准，但现在的用法如下：

```
// -webkit-gradient(<type>, <point> [, <radius>]?, <point> [, <radius>]?
//[, <stop>*]);
-webkit-gradient(linear, left top, left bottom,
from(#FFF), color-stop(10%, #DDD), to(#CCC));
```

虽然大多数主流浏览支持 CSS 渐变标准，但都使用厂商前缀语法格式。

- Firefox：使用 `-moz-`
- Chrome：使用 `-webkit-`
- Safari：以其他实现方式
- IE >= 10：使用 `-ms-`
- Opera >= 11.1：使用 `-o-`

所以，一个能跨浏览器工作的渐变样式看上去是这样的：

```
#gradient-example {
  /* Fallback */
  background: #FFF;
  /* Chrome < 10, Safari < 5.1 */
  background: -webkit-gradient(linear, left top, left bottom, from(#FFF), to(#CCC));
  /* Chrome >= 10, Safari >= 5.1 */
  background: -webkit-linear-gradient(#FFF, #CCC);
  /* Firefox >= 3.6 */
  background: -moz-linear-gradient(#FFF, #CCC);
  /* Opera >= 11.1 */
```

```
background: -o-linear-gradient(#FFF, #CCC);
/* IE >= 10 */
background: -ms-linear-gradient(#FFF, #CCC);
/* The standard */
background: linear-gradient(#FFF, #CCC);
}
```

哎，上面这样的代码真是难以一口气写下来！所幸的是，像 Less 和 Sass 这样的项目为我们减轻了痛苦，在后面会详细阐述。

多重背景

在 CSS3 中能够指定多个阴影，当然你也可以指定多重背景。以前，想要设置多张背景图片，就需要创建很多个元素——也就是说，需要过多额外的标签。CSS3 提供以逗号分隔的背景样式，最大限度地减少了标签使用总量：

◀ 229

```
background: url(snowflakes.png) top repeat-x,
            url(chimney.png) bottom no-repeat,
            -moz-linear-gradient(white, #CCC),
            #CCC;
```

浏览器支持情况如下。

- Firefox：全部支持
- Chrome：全部支持
- Safari：全部支持
- IE >= 9.0：全部支持
- Opera：全部支持

选择器

CSS3 提供一系列新的目标元素选择器：

:first-child

选择器选中的第一个元素。

:last-child

选择器选中的最后一个元素。

:only-child

选中的元素是其父元素的惟一子元素。

:target

选中当前 URL 的哈希中的目标元素

:checked

选中复选框已被勾选的元素

下面将详细阐述几个选择器。

nth-child 选择器

使用 `:nth-child` 可以修改第 n 个子元素的样式。例如,下面的选择器选中第 $3n$ 个子元素:

```
#example:nth-child( 3n ) { /* ... */ }
```

可以用它来选中偶数或奇数的子元素:

```
/* 偶数子节点 */
#example:nth-child( 2n ) { /* ... */ }
#example:nth-child( even ) { /* ... */ }
```

```
/* 奇数子节点 */
#example:nth-child( 2n+1 ) { /* ... */ }
#example:nth-child( odd ) { /* ... */ }
```

还可以反转这个选择器:

```
/* 最后一个子节点 */
#example:nth-last-child( 1 )
```

事实上, `:first-child` 等同于 `:nth-child(1)`, `:last-child` 等同于 `:nth-last-child(1)`。

直接后代选择器

使用大于号 “>” 可以限制选择器只选中直接后代子元素:

```
/* 找到直系子节点中的元素 */
#example > div { }
```

否定选择器

使用 `:not` 否定选择器,给它传递一个简单的选择器即可。当前,它还不支持 `p:not(h1 + p)` 这样复杂的选择器:

```
/* 找到直系子孙节点,但不能包含名为 "current" 的 class */
#example > *:not(.current) {
  display: none
}
```

浏览器支持情况如下。

- Firefox : 全部支持。
- Chrome : 全部支持。
- Safari : 全部支持。
- IE >= 9.0 : 全部支持。
- Opera : 全部支持。

过渡

CSS3 增加了对过渡的支持，当样式变化时可以创建简单的动画，不过你需要传递持续的时间长度、对应的属性和可选的动画类型，等等。持续时间的单位可以是秒（s）或毫秒（ms）：

```
/* 持续时间, 属性, 动画类型 (可选) */
transition: 1.5s opacity ease-out

/* 多个动画 */
transition: 2s opacity , .5s height ease-in
transition: .5s height , .5s .5s width
```

第一个例子中，当 opacity 变化时（就是说，一个样式以内联的方式应用时），原来的样式值和新值之间用动画过渡。

◀ 231

定时器函数有好几种，如下：

- linear
- ease-in
- ease-out
- ease-in-out

或者，可以使用三次贝塞尔曲线自定义一个定时器，用它来描述动画的速率。下面的语句实现的是一个弹跳动画：

```
#transition-example {
  position: absolute;
  /* cubic-bezier(x1, y1, x2, y2) */
  transition: 5s left cubic-bezier(0.0, 0.35, .5, 1.3);
}
```

在 Safari 和 Chrome 中，一旦过渡完成，在目标元素上还会发送一个 *WebKitTransitionEvent* 事件。在 Firefox 中，事件名为 *transitionend*。不过，对于使用 CSS3 还有几个忠告：对

播放的控制能力很有限，不是所有的样式值都支持过渡。对于简单的动画来说过渡非常有用，甚至有些浏览器（如 Safari）有硬件加速的支持：

```
#transition-example {
  width: 50px;
  height: 50px;
  background: red;
  -webkit-transition: 2s background ease-in-out;
  -moz-transition: 2s background ease-in-out;
  -o-transition: 2s background ease-in-out;
  transition: 2s background ease-in-out;
}

#transition-example:hover {
  background: blue;
}
```

由于这样或那样的原因，对于渐变的过渡只能在 Alpha 透明度变化时起作用。而且，有些值也不能过渡，如 `height:0` 到 `height:auto`。

浏览器支持情况如下。

- Firefox：使用 `-moz-`
- Chrome：使用 `-webkit-`
- Safari：使用 `-webkit-`
- IE >= 10.0：使用 `-ms-`
- Opera：使用 `-o-`

232 图片边框

使用 `border-image` 可以用图片作为一个元素的边框。第一个参数指定图片的 URL；紧接着的几个参数描述图片是如何切片使用的；最后的参数指定伸缩属性值，它们描述对于边部和中部的切片的比例关系和铺设方式。伸缩属性值有 3 种：`round`、`repeat` 和 `stretch`：

```
border-image: url(border.png) 14 14 14 14 round round;

border-image: url(border.png) 14 14 14 14 stretch stretch;
```

浏览器支持情况如下。

- Firefox：使用 `-moz-`
- Chrome：使用 `-webkit-`
- Safari：使用 `-webkit-`

- IE : 不支持
- Opera : 使用 -o-

盒子尺寸

有没有碰到过希望设置一个元素的宽度为 100%，但是它仍然有内边距或外边距的应用？使用传统的盒模型时，CSS 使用父元素的宽度计算百分比宽度，然后再加上外边距和内边距。换言之，100% 宽度的元素加上其内边距、外边距和边框总是会溢出的。

因此，将 `box-sizing` 设置为 `border-box`——而不是原来的默认值 `content-box`——你可以改变测量尺寸的方式，将边框、外边距、内边距和内容都考虑进去了：

```
.border-box {  
  -webkit-box-sizing: border-box;  
  -moz-box-sizing: border-box;  
  box-sizing: border-box;  
}
```

主流的浏览器基本都支持这种用法，除非你打算支持 IE8 以前的版本，那么可以放心使用这个属性。

变形

CSS3 提供了基本的 2D 变形，能使元素平移、旋转、按比例缩放和倾斜。例如，将一个元素逆时针旋转 30 度：

```
transform: rotate( -30deg );
```

将一个元素环绕 x 和 y 轴倾斜指定的角度：

```
transform: skew( 30deg , -10deg );
```

使用 `translateX` 或 `translateY` 可以将一个元素的位置沿 x 或 y 轴平移：

```
translateX(30px);  
translateY(500px);
```

使用 `scale` 可以将一个元素的尺寸放大或缩小。默认情况下元素的 `scale` 值为 1：

```
transform: scale(1.2);
```

可以连着书写多个变形：

```
transform: rotate(30deg) skewX(30deg);
```

浏览器支持情况如下。

- Firefox : 使用 `-moz-`
- Chrome : 使用 `-webkit-`
- Safari : 使用 `-webkit-`
- IE >= 9 : 使用 `-ms-`
- Opera : 使用 `-o-`

弹性盒模型

CSS3 引入了弹性盒模型的概念，这是一种显示内容的全新方式。因为它将一些在 GUI 框架（如 Adobe Flex）中已经应用了一段时间的新特性引入到 CSS 中了，所以这个确实有用。过去，如果要使一个列表水平排列，就要用浮动。而弹性盒模型可以实现更多效果和功能。看一看如下代码：

```
.hbox {
  display: -webkit-box;
  -webkit-box-orient: horizontal;
  -webkit-box-align: stretch;
  -webkit-box-pack: left;

  display: -moz-box;
  -moz-box-orient: horizontal;
  -moz-box-align: stretch;
  -moz-box-pack: left;
}

.vbox {
  display: -webkit-box;
  -webkit-box-orient: vertical;
  -webkit-box-align: stretch;

  display: -moz-box;
  -moz-box-orient: vertical;
  -moz-box-align: stretch;
}
```

234

上面将 `display` 设置为 `-webkit-box` 或者 `-moz-box`，然后设置子元素布局的方向。默认情况下，所有子元素都将自动扩充为父元素一样的大小。但通过设置 `box-flex` 属性却可以修改此默认行为。

如果设置 `box-flex` 为 0，就指定了该元素不允许扩充；相反，如果设置为 1 或者更大的数值，该元素将会自动扩充可利用的内容空间。例如，对于一个侧边栏也许应该设置其 `flex` 为 0，

而对于一个主内容区也许应该设置 flex 为 1 :

```
#sidebar {
  -webkit-box-flex: 0;
  -moz-box-flex: 0;
  box-flex: 0;

  width: 200px;
}

#content {
  -webkit-box-flex: 1;
  -moz-box-flex: 1;
  box-flex: 1;
}
```

浏览器支持情况如下。

- Firefox : 使用 -moz-
- Chrome : 使用 -webkit-
- Safari : 使用 -webkit-
- IE >= 10 : 使用 -ms-
- Opera : 不支持

字体

@font-face 让我们能够在网页上使用自定义字体来显示文本，这样就不再受制于用户安装的几种有限的系统字体了。

它所支持的字体格式有 TrueType 和 OpenType。字体遵循一些域名策略限制——字体文件和页面必须来自同一个域名。

按照如下方法使用 @font-face，命名 font-family 并指定字体文件的 URL 路径：

```
@font-face {
  font-family: "Bitstream Vera Serif Bold";
  src: url("/fonts/VeraSeBd.ttf");
}
```

然后，就像使用其他字体一样使用它即可：

```
#font-example {
  font-family: "Bitstream Vera Serif Bold";
}
```

字体文件将异步下载，当下载完后才会被应用。这就意味着用户首先看到是系统里的默认字体，直到自定义的字体下载完成。因此，最好同时指定一种本地可用的字体作为后备方案。

浏览器支持情况如下。

- Firefox : 全部支持
- Chrome : 全部支持
- Safari : 全部支持
- IE >= 9 : 全部支持
- Opera : 全部支持

优雅降级

如果正确地书写 CSS，那么你的应用程序将能够优雅降级，即在不支持 CSS3 的浏览器中也能够实现功能——只是没那么漂亮。

优雅降级的关键是浏览器忽略那些它们无法理解的设置，如 CSS 属性、值和选择器等。CSS 属性通常是一个覆盖另一个，如果在同一个规则中定义两次属性，第一次的属性值将被第二次的值覆盖。应该将符合 CSS 2.1 规范的属性放在前面。如以下代码所示，如果支持 `rgba`，它将覆盖前面设置的 `white`：

```
#example-gd {
    background: white;
    background: rgba(255, 255, 255, .75);
}
```

如何处理厂商前缀？可以应用同样的规则。提供包含这些厂商前缀的各个版本即可——浏览器会应用它们能够理解的那个规则。并且在最后应该放上不带前缀的版本，因为浏览器的 CSS3 支持标准化以后，取消了前缀，也能够应用该样式：

```
#example-gd {
    background: #FFF;
    background: -webkit-gradient(linear, left top, left bottom, from(#FFF), to(#CCC));
    background: -webkit-linear-gradient(#FFF, #CCC);
    background: -moz-linear-gradient(#FFF, #CCC);
    background: linear-gradient(#FFF, #CCC);
}
```

Modernizr

Modernizr (<http://www.modernizr.com/>) 用于检测各种 CSS3 属性的支持情况，这样在样式表中能确定特定的浏览器行为：

```
.multiplebgs div p {
    /* 支持多重背景的浏览器的样式 */
}
.no-multiplebgs div p {
    /* 优雅降级适应那些不支持多重背景的浏览器 */
}
```

Modernizr 检测支持的特性有：

- @font-face
- rgba()
- hsla()
- border-image:
- border-radius:
- box-shadow:
- text-shadow:
- 多重背景
- 弹性盒模型
- CSS 动画
- CSS 渐变
- CSS 2D 变形
- CSS 过渡

如需查看全部的功能列表或者下载 Modernizr，可以访问其项目网站 (<http://www.modernizr.com/>)。

Modernizr 用起来非常简单——只要引用该 JavaScript 文件，并且在 <html> 标签上添加样式 class 为 no-js 即可：

```
<script src="/javascripts/modernizr.js"></script>
<html class="no-js">
```

然后 Modernizr 会自动给 <html> 标签添加一些样式 class，可以在选择器中用此标签来实现针对特定浏览器行为的样式：

```
/* 不支持当弹性盒子模型时，降级为使用传统布局 */
.no-flexbox #content {
```

```
float: left;
}
```

237 Google Chrome Frame

Google Chrome Frame (GCF) 是一个 IE 插件，用来让 IE 的渲染引擎切换到 Google Chrome 的渲染引擎 Chromium (<http://www.chromium.org/>)。

一旦安装了此插件，就可以使用页面头部的一个 meta 标签来开启 GCF：

```
<meta http-equiv="X-UA-Compatible" content="chrome=1">
```

或者，在服务器响应头中添加该设置：

```
X-UA-Compatible: chrome=1
```

以上就是为页面开启 GCF 渲染所需要的所有设置。但是，GCF 还有一些特性，例如当用户运行 IE（而没有安装 GCF）时自动提示用户来安装。提示信息可以在页面中以浮动层形式展现，在 GCF 安装完毕后能自动刷新页面而不需要重启浏览器。

第一步，引入 GCF 的 JavaScript 文件：

```
<script src="http://ajax.googleapis.com/ajax/libs/chrome-frame/1/CFInstall.min.js">
```

然后，在页面的 load 事件处理句柄中或在页面底部调用 CFInstall：

```
<script>
  jQuery(function(){
    CFInstall.check({
      mode: "overlay",
    });
  });
</script>
```

CFInstall 需要几个参数：

mode

可以是 inline、overlay 或 popup。

destination

安装后的目标地址，通常是当前页面。

node

包含安装提示的元素的 ID。

一旦 GCF 已经安装，浏览器的 User-Agent 头信息中会添加“chrome”字符串。GCF 非常聪明，它使用 IE 的网络协议栈来操作 URL 请求。这就保证了在使用 GCF 时，URL 请求有相同的 cookie、history 和 SSL 状态等，这样基本上保留了用户已有的会话状态（session）。

更多的信息，请查看相关文档（<http://goo.gl/L5II>）。

创建布局

现在将前面所有学过的知识，应用一下，来创建一个简单的布局，命名为 Holla。

首先，创建基本的页面标签。一个头部和两栏——一个固定宽度的侧边栏 sidebar 和一个主内容容器 main：

```
<body>
  <header id="title">
    <h1>Holla</h1>
  </header>

  <div id="content">
    <div class="sidebar"></div>
    <div class="main"></div>
  </div>
</body>
```

然后，添加基本的 reset 和 body 样式：

```
body, html {
  margin: 0;
  padding: 0;
}

body {
  font-family: Helvetica, Arial, "MS Trebuchet", sans-serif;
  font-size: 16px;
  color: #363636;
  background: #D2D2D2;
  line-height: 1.2em;
}
```

接下来是 h 标签样式：

```
h1, h2 {
  font-weight: bold;
  text-shadow: 0 1px 1px #ffffff;
}
```

```

h1 {
    font-size: 21pt;
    color: #404040;
}

h2 {
    font-size: 24pt;
    color: #404040;
    margin: 1em 0 0.7em 0;
}

h3 {
    font-size: 15px;
    color: #404040;
    text-shadow: 0 1px 1px #ffffff;
}

```

239 ➤ 现在，为布局定义头部样式。我们使用了 CSS3 背景渐变，但如果浏览器不支持，其默认的后备颜色是一个十六进制数值：

```

#title {
    border-bottom: 1px solid #535353;
    overflow: hidden;
    height: 50px;
    line-height: 50px;

    background: #575859;
    background: -webkit-gradient(linear, left top, left bottom,
    from(#575859), to(#272425));
    background: -webkit-linear-gradient(top, #575859, #272425);
    background: -moz-linear-gradient(top, #575859, #272425);
    background: linear-gradient(top, #575859, #272425);
}

#title h1 {
    color: #ffffff;
    text-shadow: 0 1px 1px #000000;
    margin: 0 10px;
}

```

现在，如果在浏览器中查看，应用程序的头部是深色的，如图 C-1 所示。



图C-1：目前为止这个CSS应用程序显示了一个背景渐变的头部

下面创建一个 `#content` 的 div，它是应用程序的主体。我们希望它沿 x 和 y 方向填满页面，故将其绝对定位。它的直接子元素水平对齐，所以将它的显示方式设置为弹性盒子类型：

240

```
#content {
    overflow: hidden;

    /*
     正文的 div 将覆盖整个页面
     但会为头部区域留出空间
    */
    position: absolute;
    left: 0;
    right: 0;
    top: 50px;
    bottom: 0;

    /* 子元素水平对齐 */
    display: -webkit-box;
    -webkit-box-orient: horizontal;
    -webkit-box-align: stretch;
    -webkit-box-pack: left;

    display: -moz-box;
    -moz-box-orient: horizontal;
    -moz-box-align: stretch;
    -moz-box-pack: left;
}
```

接下来, 创建左边的一栏, 命名为 `.sidebar`。它有固定的宽度, 所以设置其 `box-flex` 为 0:

```
#content .sidebar {
  background: #EDEDDE;
  width: 200px;

  /* 此栏有固定宽度 */
  -webkit-box-flex: 0;
  -moz-box-flex: 0;
  box-flex: 0;
}
```

在 `.sidebar` 内部创建一个菜单项目的列表。每个菜单用一个 `h3` 分隔, 叫做菜单头。如你所见, 这里用到很多 CSS3, 由于应用了厂商前缀, 有很多代码是重复的。如果使用 Less 的混合则代码可以更简洁:

```
#content .sidebar ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

#content .sidebar ul li {
  display: block;
  padding: 10px 10px 7px 20px;
  border-bottom: 1px solid #cdcdee;
  cursor: pointer;

  -moz-box-shadow: 0 1px 1px #fcfcfc;
  -webkit-box-shadow: 0 1px 1px #fcfcfc;
  box-shadow: 0 1px 1px #fcfcfc;
}

#content .sidebar ul li.active {
  color: #ffffff;
  text-shadow: 0 1px 1px #46677f;

  -webkit-box-shadow: none;
  -moz-box-shadow: none;

  background: #7bb5db;
  background: -webkit-gradient(linear, left top, left bottom,
    from(#7bb5db), to(#4775b8));
  background: -webkit-linear-gradient(top, #7bb5db, #4775b8);
  background: -moz-linear-gradient(top, #7bb5db, #4775b8);
  background: linear-gradient(top, #7bb5db, #4775b8);
}
```

在 HTML 中标签中添加一些示例菜单：

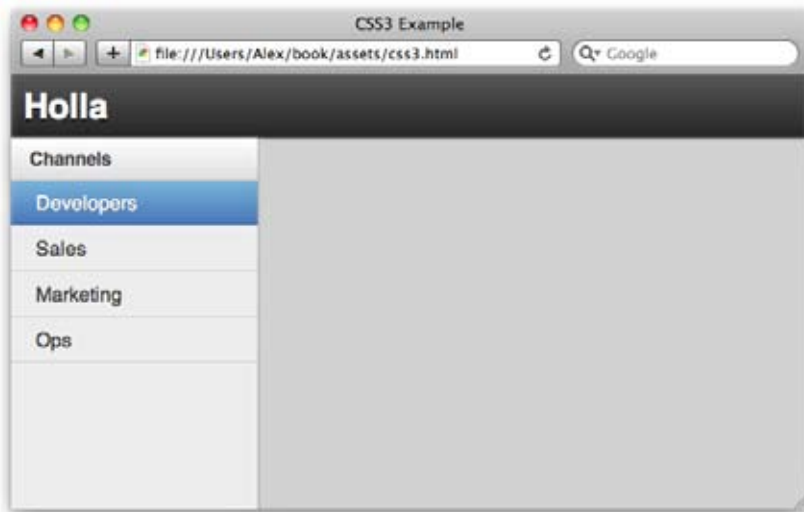
```
<div class="sidebar">
  <h3>Channels</h3>
  <ul>
    <li class="active">Developers</li>
    <li>Sales</li>
    <li>Marketing</li>
    <li>Ops</li>
  </ul>
</div>
```

最后还要给 .main 这个 div 添加一些 CSS，它在页面右侧拉伸填满：

```
#content .main {
  -moz-box-shadow: inset 0 1px 3px #7f7f7f;
  -webkit-box-shadow: inset 0 1px 3px #7f7f7f;
  box-shadow: inset 0 1px 3px #7f7f7f;

  /* 我们希望 .main 可以尽可能地伸展开 */
  -webkit-box-flex: 1;
  -moz-box-flex: 1;
  box-flex: 1;
}
```

请再查看一下浏览器，如图 C-2 所示，到目前为止我们得到了一个应用程序的基本布局，它还可以进一步扩展。



图C-2：应用程序的基本布局

242 前面提到过，由于我们不得不使用厂商前缀，CSS3 的语法很冗长，但使用 Less 的混合可以让代码更干净。例如：

```
#content .sidebar h3 {  
  .vbg-gradient(#FFF, #DEDFE0);  
  .box-shadow(0, -5px, 10px, #E4E4E4);  
}
```

更多信息请查看附录 B，并阅读 Holla 的样式表，它是一个很好的样例。

Symbols

\$ shortcut, jQuery, 50
\$\$() function, 126
\$() function, 126
\$x() function, 127
() braces, anonymous functions, 50
_ (underscore), prefixing private properties, 16
_.bindAll(), 171

A

Access-Control-Allow-Origin header, 42
Access-Control-Request-Headers header, 42
addEventListener(), 19, 24
addressing references, 37
Adobe Flash, 44
afterEach() function, 114
Ajax
 crawling, 62
 jQuery, 212
 loading data with, 39–42
 progress, 93
 syncing and Spine, 150
ajax() function, 212
ajaxError event, 150
alpha transparency, 225
AMD format, 77
Apache web server, relative expiration date, 134
APIs
 History API, 63
 HTML5 file APIs, 81
 jQuery.tmpl templating API, 68
 jQuery's selectors API, 207

 pushState() and replaceState() history API, 173
 Underscore's AP, 165
 WebSocket API, 99
 XMLHttpRequest API, 92
App.log() function, 126
append() function, 210
apply(), 12
arguments variable, 14
assert libraries, 110
assert() function, 109
assertEqual() function, 110
assertion types, 112
assertions, 109
async attribute, 134
attributes
 async attribute, 134
 Backbone library models, 166
 defer attribute, 134
 files in HTML5, 82
 JavaScriptMVC library models, 189–191
 multiple attribute, 82
 originalEvent attribute, 84
 returning, 46
 validating an instance's attributes, 167
 whitelisting, 176
auditors, deployment, 138
autoLink() function, 69

B

Backbone library, 165–183
 collections, 167
 controllers, 172
 models, 165
 syncing with the server, 174–177

†：索引所列页码为英文版页码，请参照用“□”表示的原书页码。

- to-do lists, 178–183
- views, 169
- Backbone.sync() function, 174, 176
- backgrounds, CSS3, 228
- beforecopy event, 87
- beforecut event, 87
- beforeEach() function, 114
- beforepaste event, 88
- bind() function, 15, 23, 57, 211
- binding, about, 70
- blobs and slices, 90
- border images in CSS3, 232
- border-radius style, 225
- box model, flexible box model: CSS3, 233
- box sizing, CSS3, 232
- box-shadow style, 226
- breakpoints, 127
- browse buttons, 91
- browsers
 - CommonJS modules, 75
 - copy and paste, 87
 - CORS, 41
 - CSS3, 223–242
 - DOMContentLoaded event, 23
 - drag and drop, 83
 - event types, 20
 - files, 81
 - Firebug add-on, 124
 - graceful degradation, 223
 - hashchange events, 61
 - local storage, 44
 - performance, 134
 - testing JavaScript, 107
 - uploading files, 91
- bubbles event, 21
- bubbling, Spine, 152

C

- Cache-Control, 135
- caching, deployment, 134
- call(), 12
- callbacks
 - ajax() function, 212
 - Backbone, 177
 - binding to model events, 147
 - click() callback, 15
 - context changes in event callbacks, 171
 - event callbacks, 151
 - extended and included callbacks, 9

- hash of, 170
- jQuery has some iterators,, 209
- Model.created() callback, 38
- registering to events, 51
- Spine libraries, 143
- canceling
 - default drag/drop, 85
 - events, 21
- Cappuccino, 89, 165
- CDNs (content delivery networks), 138
- change event, 59, 71, 159, 171, 175
- change() function, 71
- chat room, 103
- Chrome, Web Inspector, 123
- class, as a reversed keyword, 7
- classes
 - adding functions to, 7
 - adding inheritance to class libraries, 11
 - adding methods to class libraries, 8
 - class inheritance using prototype, 10
 - controlling scope in class libraries, 14
 - JavaScriptMVC library, 186–189
 - calling base methods, 187
 - instantiation, 186
 - introspection, 188
 - model example, 188
 - proxies, 187
 - static inheritance, 187
 - MVC pattern, 6
 - Spine libraries, 142–145
 - Spine.List class, 159
 - WebSockets, 99
 - XMLHttpRequest class, 40
- click event, 91
- click() callback, 15
- client-side templates, JavaScriptMVC library, 197–200
 - \$.View and subtemplates, 198
 - basic use, 197
 - deferreds, 199
 - jQuery modifiers, 198
 - loading from a script tag, 198
 - packaging, preloading and performance, 199
- clipboardData object, 87
- clipboardData property, 88
- collections
 - Backbone library, 167
 - populating: Backbone library, 175

- colors
 - CSS extensions, 219
 - CSS3, 224
- Comet techniques, 97
- CommonJS
 - about, 74
 - module loaders, 76
- CommonJS initiative, 74
- comparator() function, 169
- compression, Gzip, 137
- concatenation, 78
- console, debugging, 125
- console.error(), 125
- console.log() function, 125
- console.profile() and console.profileEnd(), 129
- console.time(name) and console.timeEnd(name), 131
- console.trace(), 126
- console.warn(), 125
- constructor functions, 6
- contacts manager in Spine libraries, 156–163
 - App controller, 163
 - contact model, 157
 - Contacts controller, 160
 - Sidebar controller, 158
- content delivery networks (CDNs), 138
- context, 51–57
 - abstracting into a library, 52
 - changing, 12, 24, 171, 212
 - delegating events, 56
- controllers, 49–64
 - about, 5
 - abstracting into a library, 52
 - accessing views, 55
 - adding context, 51–57
 - App controller in contacts manager in Spine libraries, 163
 - Backbone library, 172
 - contacts controller in contacts manager in Spine libraries, 160
 - delegating events, 56
 - JavaScriptMVC library, 200–205
 - event binding, 203
 - instantiation, 202
 - overview, 202
 - templated actions, 204
 - loading controllers after the document, 53
 - module pattern, 50
 - routing, 60–64
 - Ajax crawling, 62
 - detecting hash changes, 61
 - HTML5 History API, 63
 - URL's hash, 60
 - Spine libraries, 150–156
 - state machines, 58
- cookies, 44
- copy and paste, 87–89
- copy event, 87
- corners, rounding in CSS3, 225
- CORS (cross-origin resource sharing), 41
- crawling, Ajax, 62
- create() function, 37, 142, 146, 150
- created() function, 143
- cross-browser event listeners, 23
- cross-domain requests, security, 43
- CRUD events, JavaScriptMVC library models, 196
- CRUD list example, 205
- CSS extensions, 217–220
 - colors, 219
 - including other stylesheets, 219
 - Less code, 219
 - mixins, 218
 - nested rules, 218
 - variables, 217
- CSS gradient standard, 228
- CSS sprites, 133
- css() function, 210
- CSS3, 223–242
 - border images, 232
 - box sizing, 232
 - colors, 224
 - creating a layout, 238
 - CSS3, 234
 - drop shadows, 226
 - flexible box model, 233
 - fonts, 234
 - graceful degradation, 235
 - gradients, 227
 - mixins, 218
 - multiple backgrounds, 228
 - prefixes, 224
 - rounding corners, 225
 - selectors, 229
 - text shadows, 227
 - transformations, 232
 - transitions, 230

cubic bezier curve, 231

D

data

- loading, 38–43
 - including data inline, 39
- JSONP, 43
- security with cross-domain requests, 43
 - with Ajax, 39–42
- models, 3
- storing locally, 44

dataTransfer object, 84, 86

dataTransfer.getData() function, 86

dataType option, 212

deactivate() function, 59

debugging, 122–131

- (see also testing)
- console, 125
- inspectors, 122
- network requests, 128
- profile and timing, 129–131
- using the debugger, 127

declaring CommonJS modules, 74

defaults

- browsers and default actions to events, 21
- caching, 134
- constructor function return context, 7
- controllers extending, 153
- JavaScriptMVC library models, 192

defer attribute, 134

deferreds, JavaScriptMVC library client-side templates, 199

define() function, 78

degradation, graceful degradation: CSS3, 235

delegate() function, 24, 57

delegating

- Backbone library events, 170
- events, 24, 56

dependency management, 73–80

- CommonJS, 74
- FUBCs, 80
- module alternative, 79
- module loaders, 76
- wrapping up modules, 78

deployment, 133–139

- auditors, 138
- caching, 134
- CDNs, 138
- Gzip compression, 137

minification, 136

performance, 133

resources, 139

descendants, CSS3, 230

describe() function, 114

destroy() function, 4, 144, 147

dir() function, 127

direct descendants, CSS3, 230

distributed testing, 121

document.createElement(), 65

document.ready event, 211, 213

DOM elements

controllers, 151

creating, 65

DOM, jQuery, 208–211

domains

- same origin policy, 41
- whitelist of, 43

DOMContentLoaded event, 23, 129

DownloadURL type, 85

drag and drop

files, 83–86

jQuery drag-and-drop uploader, 95–96

dragenter event, 85

dragover event, 85

dragstart event, 84

drivers, testing, 115

drop areas, 95

drop event, 95

drop shadows, CSS3, 226

dropping files, 85

dynamically rendering views, 65

E

el property, 151

element pattern, 154

elements

- associated with events, 22
- DOM elements, 65
- mapping, 55
- Spine libraries' controllers, 152
- tooltip element, 201

empty() function, 211

Envjs, 119

equals() function, 112

error event, 148, 167

ETags, 136

event bubbling, 20

event callbacks

- context changes, 171
- proxying, 151
- event capturing, 20
- events, 19–30
 - ajaxError event, 150
 - Backbone library, 170
 - canceling, 21
 - change event, 59, 71, 159, 171, 175
 - click event, 91
 - context change, 24
 - controllers and event listeners, 5
 - copying and pasting events, 87
 - custom events, 25–27
 - delegating, 24, 56, 152
 - document.ready event, 211, 213
 - DOMContentLoaded event, 129
 - drag and drop, 83
 - dragenter event, 85
 - dragover event, 85
 - dragstart event, 84
 - drop event, 95
 - error event, 148, 167
 - event binding: JavaScriptMVC library
 - controllers, 203
 - event libraries, 23
 - event object, 21
 - global events in Spine libraries' controllers, 153
 - hash of, 170
 - hashchange events, 61
 - jQuery, 211
 - keydown event, 152
 - listening to, 19
 - load event, 93
 - non-DOM events, 27–30
 - onhashchange event, 173
 - onopen event, 99
 - ordering, 20
 - pasting events, 88
 - popstate events, 64
 - progress event, 93, 94
 - real-time architecture, 103
 - refresh event, 175
 - registering callbacks to, 51
 - show:contact event, 160
 - Spine libraries, 145, 147
 - Spine libraries' controller events, 153
- exists() function, 146
- expect() function, 115
- expectation management, 105
- Expires header, 134
- export, global export, 50
- extend() function, 8, 143, 146, 166
- extending
 - CSS, 217–220
 - jQuery, 213
- extending models, JavaScriptMVC library, 191
- extensions, wrapping, 214

F

- fetch() function, 175
- fetchRemote() function, 32
- FileReader object, 89
- files, 81–96
 - browser support, 81
 - copy and paste, 87–89
 - custom browse buttons, 91
 - drag and drop, 83–86
 - getting information about, 81
 - inputs, 82
 - jQuery drag-and-drop uploader, 95–96
 - reading, 89
 - uploading, 91–94
- find() function, 36, 146
- find() operation, 37
- Finite State Machines (FSMs), 58
- Firebug, 124, 138
- Firebug Lite, 125
- Firefox
 - Firebug, 124
 - Firebug and YSlow, 138
 - Selenium IDE, 116
- firewalls, WebSockets, 100
- flexible box model, CSS3, 233
- FlyScript, 79
- Followers and Followees collections, 167
- FormData instance, 92
- FSMs (Finite State Machines), 58
- FUBCs (flashes of unbehaved content), 80
- functions
 - \$\$() function, 126
 - \$() function, 126
 - \$x() function, 127
 - activate() function, 59
 - add() function, 168
 - addClass() function, 210
 - addEventListener(), 19

- adding private functions, 16
- adding to classes, 7
- afterEach() function, 114
- ajax() function, 212
- anonymous functions, 16, 50
- App.log() function, 126
- append() function, 210
- assert() function, 109
- assertEqual() function, 110
- autoLink() function, 69
- Backbone.sync() function, 174, 176
- beforeEach() function, 114
- bind() function, 15, 23, 57, 211
- change() function, 71
- clear() function, 127
- comparator() function, 169
- console.log() function, 125
- constructor functions, 6
- create() function, 37, 142, 146, 150
- created() function, 143
- css() function, 210
- dataTransfer.getData() function, 86
- deactivate() function, 59
- define() function, 78
- delegate() function, 24, 57
- describe() function, 114
- destroy() function, 4, 144, 147
- dir() function, 127
- empty() function, 211
- equals() function, 112
- exists() function, 146
- expect() function, 115
- extend() function, 8, 143, 146, 166
- fetch() function, 175
- fetchRemote() function, 32
- find() function, 36, 146
- function invocation, 12
- generic helper functions inside the view, 68
- get() function, 166
- getData() function, 88
- getter and setter functions, 211
- global variables and functions, 4
- history.back() and history.forward() functions, 64
- history.pushState() function, 63
- include() function, 8, 52, 143, 146
- init() function, 34, 55, 142, 151
- initialize() instance function, 166, 168
- inspect() function, 127
- it() function, 114
- jQuery.ajax() function, 40
- jQuery.tmpl() function, 67
- keys() function, 127
- load() function, 52, 74
- Math.random() function, 36
- module() function, 111
- namespacing, 31
- Object.create() function, 33, 142
- post() function, 47
- prepend() function, 210
- proxy() function, 14, 24, 52, 144
- proxyAll() function, 144
- ready() function, 23
- refresh() function, 175
- remove() function, 168
- removeEventListener(), 19
- render() function, 171
- require() function, 74–76
- reventDefault() function, 21
- same() function, 112
- save() function, 146, 174, 194
- search() function, 173
- send() function, 92, 99
- set() function, 166
- setData() function, 84
- setDragImage() function, 85
- slice() function, 90
- stopImmediatePropagation() function, 21
- stopPropagation() function, 21
- text() function, 211
- timing functions, 231
- toJSON() function, 170
- trigger() function, 25
- update() function, 37
- uploadFile() function, 95
- validate() function, 167
- values() function, 127

G

- GCF (Google Chrome Frame), 237
- get() function, 166
- getData() function, 88
- getJSON(), 212
- getter and setter functions, 211
- global export, 50
- global import, 50
- global variables and functions, 4
- Google

- Ajax Crawling specification, 62
- GCF, 237
- Pagerank algorithm, 139
- V8 JavaScript engine, 119
- Google Analytics, 134
- graceful degradation
 - about, 223
 - CSS3, 235
- gradients, CSS3, 227
- Growl jQuery plug-in, 214
- GUID generators, 36
- Gzip compression, 137

H

- hash
 - default, 167
 - detecting hash changes, 61
 - events and callbacks, 170
 - routing and URL's hash, 60
- headless testing, 118–121
 - Ichabod library, 121
 - Zombie.js, 119
- helpers
 - defined, 4
 - JavaScriptMVC library models, 192
 - templates, 68
- history
 - JavaScript, xi, 1
 - real-time Web, 97
- History API, 63
- history.back() and history.forward() functions, 64
- history.pushState() function, 63
- HJS plug-in, 16
- Holla, xvii, 104
- HTML
 - prerendering, 65
 - templating, 181
- HTML5
 - drag and drop, 83
 - History API, 63
 - HTML5 file APIs, 81
 - local storage, 45
 - WebSockets, 98
- HTTP requests, performance, 133

I

- Ichabod library, 121

- ID support, 36
- If-Modified-Since header, 135
- If-None-Match header, 136
- images, border images in CSS3, 232
- immutable properties, 16
- import, global import, 50
- include() function, 8, 52, 143, 146
- including data inline, 39
- inheritance
 - adding inheritance to class libraries, 11
 - class inheritance using prototype, 10
 - classes, 17
 - prototypal inheritance, 33
 - static inheritance: JavaScriptMVC library classes, 187
- init method, 202
- init() function, 34, 55, 142, 151
- initialize() instance function, 166, 168
- inputs, files, 82
- inspect() function, 127
- inspectors, 122
- instantiation
 - JavaScriptMVC library classes, 186
 - JavaScriptMVC library controllers, 202
 - Spine libraries' classes, 142
- interfaces (see views)
- interpolating variables, 67
- introspection, JavaScriptMVC library classes, 188
- invocation, function invocation, 12

J

- Jasmine, 113–115
- JavaScript
 - history, xi, 1
 - Less code, 220
 - minification, 136
- JavaScriptMVC library, 185–206
 - abstract CRUD list, 205
 - classes, 186–189
 - calling base methods, 187
 - instantiation, 186
 - introspection, 188
 - model example, 188
 - proxies, 187
 - static inheritance, 187
 - client-side templates, 197–200
 - \$.View and subtemplates, 198
 - basic use, 197

- deferreds, 199
 - jQuery modifiers, 198
 - loading from a script tag, 198
 - packaging, preloading and performance, 199
 - controllers, 200–205
 - event binding, 203
 - instantiation, 202
 - overview, 202
 - templated actions, 204
 - models, 189–196
 - attributes and observables, 189–191
 - CRUD events, 196
 - defaults, 192
 - extending, 191
 - helper methods, 192
 - service encapsulation, 193–195
 - setters, 191
 - type conversion, 196
 - setup, 186
 - jQuery, 207–216
 - \$ shortcut, 50
 - about, 207
 - Ajax, 212
 - apply() and call(), 12
 - being a good web citizen, 213
 - DOM manipulation, 209
 - DOM traversal, 208
 - drag-and-drop uploader, 95–96
 - events, 211
 - extensions, 213
 - Growl plug-in, 214
 - plug-ins and custom events, 25
 - jQuery() selector, 55
 - jQuery.ajax() function, 40
 - jQuery.bind, 203
 - jquery.browse.js plug-in, 91
 - jQuery.delegate, 203
 - jQuery.extend(), 34
 - jQuery.fn object, 213
 - jQuery.get, 200
 - jquery.js, 79
 - jQuery.prototype, 213
 - jQuery.proxy(), 52, 212
 - jQuery.tmpl, 181
 - jQuery.tmpl library, 67
 - jquery.upload.js plug-in, 93
 - JSMin, 136
 - JSON objects, 39
 - JSONP (JSON with padding), 43
- ## K
- keyboard events, 22
 - keydown event, 152
 - keys() function, 127
 - keywords
 - class as a reserved keyword, 7
 - new keyword and calling constructor functions, 6
 - klass, 7
- ## L
- LABjs, 80
 - Last-Modified header, 135
 - layouts, creating in CSS3, 238
 - Less code, CSS extensions, 219
 - Less.app, 220
 - libraries
 - adding inheritance to class libraries, 11
 - assert libraries, 110
 - Backbone library, 165–183
 - CDN, 138
 - class libraries, 8, 16
 - controlling scope in class libraries, 14
 - Envjs, 119
 - event libraries, 23
 - Ichabod library, 121
 - JavaScriptMVC library, 185–206
 - jQuery, 207–216
 - Selenium, 116
 - Spine libraries, 17, 141–163
 - templating libraries, 66
 - underscore.js library, 165
 - linear and radial gradients, 227
 - listening to events, 19
 - literals, JavaScript object literals, 6
 - load event, 93
 - load() function, 52, 74
 - loading
 - controllers after the document, 53
 - data
 - including data inline, 39
 - JSONP, 43
 - security with cross-domain requests, 43
 - with Ajax, 39
 - local storage
 - about, 44

- adapter, 177
- adding to ORMs, 46
- logging level, 125

M

- MacRuby, 121
- manual testing, 109
- matchers, 115
- Math.random() function, 36
- messages, sending and receiving, 99
- minification
 - about, 78
 - deployment, 136
- mixins, CSS extensions, 218
- model ID, 168
- model property, 168
- Model.created() callback, 38
- models, 31–48
 - about, 3
 - adding ID support, 36
 - adding local storage ORM, 46
 - addressing references, 37
 - Backbone library, 165
 - binding, 71
 - contact model in contacts manager in Spine
 - libraries, 157
 - flexible box model: CSS3, 233
 - JavaScriptMVC library, 189–196
 - attributes and observables, 189–191
 - CRUD events, 196
 - defaults, 192
 - extending models, 191
 - helper methods, 192
 - service encapsulation, 193–195
 - setters, 191
 - type conversion, 196
 - JavaScriptMVC library classes example, 188
 - loading data, 38–43
 - including data inline, 39
 - JSONP, 43
 - security with cross-domain requests, 43
 - with Ajax, 39–42
 - MVC pattern and namespacing, 31
 - ORMs, 32–36
 - populating ORMs, 44
 - Spine libraries, 145–150
 - storing data locally, 44
 - submitting new records to the server, 47

- Modernizr, 236
- modified time (mtime), 135
- modularity, MVC pattern and classes, 6
- module pattern, 50
- module transport format, 75
- module() function, 111
- modules
 - alternatives to, 79
 - browser, 75
 - declaring with CommonJS, 74
 - module loaders, 76
 - wrapping up, 78
- mod_deflate, 138
- multiple attribute, 82
- Mustache, 67
- MVC pattern, 1–17
 - about, 2
 - adding inheritance to class libraries, 11
 - adding methods to class libraries, 8
 - adding private functions, 16
 - adding structure to applications, 2
 - class functions, 7
 - class inheritance using prototype, 10
 - class libraries, 16
 - classes, 6
 - controlling scope in class libraries, 14
 - function invocation, 12
 - JavaScript history, 1
 - namespacing, 31

N

- namespacing
 - CommonJS initiative, 74
 - custom events, 25
 - importance of, 73
 - introspection, 188
 - MVC pattern, 31
 - namespace pollution, 50
 - variable definitions and global namespaces, 50
- negating selectors in CSS3, 230
- nested rules, CSS extensions, 218
- network requests, 128
- new operator, 6
- next and prev methods, 192
- Node.js, 101, 220
- non-DOM events, 27–30
- nth-child, CSS3, 229

O

- object-oriented languages, JavaScript as, 2
- Object-relational mappers (see ORMs)
- Object.create() function, 33, 142
- objects
 - clipboardData object, 87
 - dataTransfer object, 84, 86
 - event object, 21, 153
 - FileReader object, 89
 - JavaScript object literals, 6
 - jQuery.fn object, 213
 - JSON objects, 39
 - localStorage and sessionStorage objects, 45
 - properties and namespacing, 31
 - prototypical objects, 10
- onhashchange event, 173
- onmessage, 99
- onopen event, 99
- operators
 - new operator, 6
 - var operator, 16
- options.error() callback, 177
- options.success() callback, 177
- order, events, 20
- originalEvent attribute, 84
- ORMs (Object-relational mappers), 32–36
 - adding local storage, 46
 - adding ORM properties, 34
 - persisting records, 35
 - populating, 44
 - prototypal inheritance, 33

P

- packaging, JavaScriptMVC library client-side
 - templates, 199
- Pagerank algorithm, 139
- pagination controls, 189
- pasting, 88
- performance
 - deployment, 133
 - JavaScriptMVC library client-side
 - templates, 199
 - perceived speed, 105
- persistence
 - records: ORMs, 35
 - Spine libraries' models, 148
- plug-ins
 - Adobe Flash, 44

- Growl jQuery plug-in, 214
- HJS plug-in, 16
- jQuery plug-ins and custom events, 25
- jquery.browse.js plug-in, 91
- jQuery.tmpl library, 67
- jquery.upload.js plug-in, 93
- Selenium, 116
- popstate events, 64
- populating
 - collections: Backbone library, 175
 - ORMs, 44
- post() function, 47
- prefixes, CSS3, 224
- preloading
 - data, 38
 - intelligent preloading, 105
- prepend() function, 210
- prerendering HTML, 65
- private functions, 16
- profile, debugging, 129–131
- progress event
 - about, 94
 - Ajax, 93
- properties
 - adding ORM properties, 34
 - adding to classes, 8
 - bubbles event, 22
 - classes, 17
 - clipboardData property, 88
 - defining classes, 16
 - el property, 151
 - event object, 21
 - events property, 152
 - immutable properties, 16
 - keyboard events, 22
 - model property, 168
 - prototypical objects, 10
 - Spine libraries, 143
 - url instance property, 174
- prototypal inheritance
 - class inheritance using prototype, 10
 - ORMs, 33
- proxied shortcut, 151
- proxies
 - JavaScriptMVC library classes, 187
 - Spine libraries' controllers, 151
 - WebSockets, 100
- proxy() function, 14, 24, 52, 144
- proxyAll() function, 144

- Publish/Subscribe pattern, 27
- PubSub
 - global events, 153
 - pattern, 104
- Pusher, 103
- pushState() and replaceState() history API, 173

Q

- QUnit, 110

R

- Rack, Less code, 220
- rack-modulr, 79
- rackup command, 79
- raw method, 39
- reading files, 89
- ready() function, 23
- real-time Web, 97–105
 - architecture, 103
 - history, 97
 - perceived speed, 105
 - WebSockets, 98–103
- record feature, 131
- records
 - persisting records, 35
 - retrieving in Spine libraries's models, 147
 - submitting new records to the server, 47
- references, addressing, 37
- refresh event, 175
- refresh() function, 175
- refreshElements(), 152
- registering callbacks to events, 51
- relative expiration date, 134
- remove() function, 168
- removeEventListener(), 19
- render pattern, 154
- render() function, 171
- rendering
 - Backbone library views, 170
 - dynamically rendering views, 65
 - templates inline, 70
- requests
 - network requests, 128
 - same origin policy, 41
 - security with cross-domain requests, 43
- require() function, 74–76
- RequireJS, 77

- resources, deployment, 139
- return statements, new operator, 6
- reventDefault() function, 21
- rgb style, 224
- Rhino, 74
- rounding corners, CSS3, 225
- routes, 172
- routing, 60–64
 - Ajax crawling, 62
 - detecting hash changes, 61
 - HTML5 History API, 63
 - URL's hash, 60
- RPC script, 100

S

- Safari, Web Inspector, 123
- same origin policy, 41
- same() function, 112
- Sauce Labs, 122
- save() function, 146, 174, 194
- saveLocation(), 173
- scope
 - class libraries, 14
 - context, 51
- script tags
 - about, 73
 - templates, 199
- scripts, performance, 134
- search() function, 173
- security, cross-domain requests, 43
- selectors
 - CSS3, 229
 - jQuery, 207
- Selenium, 116
- self local variable, 212
- send() function, 92, 99
- server push, 98
- server-side applications, views, 65
- server-side validation, 150
- servers
 - submitting new records to, 47
 - syncing with: Backbone library, 174–177
- session storage, 45
- set() function, 166
- setData() function, 84
- setDragImage() function, 85
- setters, JavaScriptMVC library models, 191
- shadows
 - drop shadows in CSS3, 226

- text shadows in CSS3, 227
- shortcuts, event types, 23
- show:contact event, 160
- Sidebar controller in contacts manager in Spine
 - libraries, 158
- slice() function, 90
- slices and blobs, 90
- Socket.IO, 102
- speed, real-time Web, 105
- SpiderMonkey, 74
- Spine libraries, 141–163
 - class implementation, 17
 - classes, 142–145
 - context, 144
 - extending, 143
 - instantiation, 142
 - contacts manager, 156–163
 - App controller, 163
 - contact model, 157
 - Contacts controller, 160
 - Sidebar controller, 158
 - controllers, 150–156
 - controller events, 153
 - delegating events, 152
 - element pattern, 154
 - elements, 152
 - global events, 153
 - proxying, 151
 - render pattern, 154
 - events, 145
 - models, 145–150
 - events, 147
 - fetching records, 147
 - persistence, 148
 - validation, 148
 - setup, 141
- Spine.App, 153
- Spine.Events, 153
- Spine.List class, 159
- splats, 172
- Sprockets, 79
- SproutCore, 165
- state machines, controllers, 58
- state, storing and controllers, 49
- static inheritance, JavaScriptMVC library
 - classes, 187
- Stitch, 79
- stopImmediatePropagation() function, 21
- stopPropagation() function, 21

- storage
 - local storage, 44, 46
 - templates, 69
- submitting new records to the server, 47
- syncing with the server: Backbone library, 174–177

T

- tags, loading from script tags: JavaScriptMVC
 - library client-side templates, 198
 - (see also script tags)
- Task.fetch(), 149
- teardown function, 114
- teardown option, 112
- templated actions, JavaScriptMVC library
 - controllers, 204
- templates, 66–70
 - about, 66
 - helpers, 68
 - HTML, 181
 - prototypical objects as, 10
 - script tags, 199
 - storage, 69
 - templating interface, 197
 - views, 4
- testing, 107–122
 - (see also debugging)
 - about, 107
 - distributed testing, 121
 - drivers, 115
 - headless testing, 118–121
 - Ichabod library, 121
 - Zombie.js, 119
 - support, 122
 - unit testing, 109–115
 - assertions, 109
 - Jasmine, 113–115
 - QUnit, 110
- TestSwarm, 121
- text shadows, CSS3, 227
- text() function, 211
- this
 - context change, 212
 - function invocation, 12
- this.App, 154
- this.input, 152
- this.proxy(), 151
- timeEnd(), 131
- timing

- debugging, 129–131
 - functions, 231
- to-do lists, Backbone library, 178–183
- toggleClass(), 53
- toJSON() function, 170
- tooltip, 200
- transformations, CSS3, 232
- transitionend, 231
- transitions, CSS3, 230
- trigger() function, 25
- Twitter, Ajax crawling, 62
- type conversion
 - JavaScriptMVC library models, 196
 - testing, 109

U

- UIs, FSMs, 58
- underscore (_), prefixing private properties, 16
- underscore.js library, 165
- unit testing, 109–115
 - assertions, 109
 - Jasmine, 113–115
 - QUnit, 110
- update() function, 37
- uploaders, jQuery drag-and-drop uploader, 95–96
- uploadFile() function, 95
- uploading files, 91–94
- url instance property, 174
- URLs, routing and URL's hash, 60
- User.extend(), 143
- UX (user experience), 105

V

- V8 JavaScript engine, 119
- validate() function, 167
- validation, Spine libraries' models, 148
- values() function, 127
- var operator, 16
- variables
 - arguments variable, 14
 - CSS extensions, 217
 - global variables and functions, 4
 - interpolating, 67
 - namespacing, 31
 - variable definitions and global namespaces, 50

- views
 - about, 4
 - Backbone library, 169
 - controllers, 55
 - dynamically rendering, 65
 - JavaScriptMVC library client-side templates, 197–200
- Vows.js, 119

W

- Watir, 116
- Web Inspector, 123
- web page for this book, xviii
- Web-socket-js, 100
- WebKit, 87
- WebKitTransitionEvent, 231
- WebSockets, 98–103
- whitelist
 - of domains, 43
 - whitelisting attributes, 176
- wildcards, 172
- wrapping
 - extensions, 214
 - modules, 78

X

- XDomainRequest, 42
- XMLHttpRequest API, 92
- XMLHttpRequest class, 40

Y

- Yabble, 76
- YSlow, 138
- YUI Compressor, 136

Z

- Zepto.js, 165
- Zombie.js, 119

作者介绍

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，还开发过 Taskforce, Socialmod 等大型开源项目，活跃在纽约、旧金山和柏林的各大 Ruby/Rails 会议。除了作为一名工程师，他还喜欢带着他的尼康 D90 和冲浪板环游世界。

译者介绍

李晶，花名拔赤，淘宝前端工程师，具有多年前端开发经验，在团队协作、框架开发等方面有深入研究，曾经参与淘宝彩票、保险和淘宝首页等项目开发。热爱分享，喜欢折腾。《JavaScript 权威指南（第六版）》译者。

张散集，花名一舟，淘宝前端工程师。主要从事技术管理，负责淘宝网（北京）—新业务技术—前端团队，热爱前端新技术的推广与应用。《JavaScript 权威指南（第六版）》译者。

封面图片介绍

本书封面的动物是长耳鸮（也叫长耳猫头鹰，或夜猫子）。

长耳鸮身材苗条修长，羽毛呈灰褐色，是一种林地猫头鹰。它的显著特点就是头上有一对高耸的耳朵。因为其耳朵很有特点，所以我们才给这种鸟起了一个外号叫“猫头鹰”。

长耳鸮生活于北美、欧洲、亚洲以及北非的开阔丛林、灌木林和森林之中。它们主要猎食小型啮齿动物，比如田鼠和老鼠，它们的耳朵非常灵敏，能够精准地定位猎物的位置，即使在黑夜中也能捕猎。它们往返于开阔地之间寻找猎物，飞行技巧高超，飞行时非常轻盈且平稳，很多人经常将它和巨型飞蛾相比较。

长耳鸮不会自己筑巢。相反，它们会使用被喜鹊、乌鸦、鹰以及松鼠遗弃的巢穴。在猎食季节（通常是在四月中到六月初），它们彼此分散且有各自的领地，而在寒冷的冬季，它们则会聚集在一起，彼此依靠以相互取暖，通常是 7~50 只长耳鸮在一起栖息。