

Mastering Web Application Development with AngularJS

化繁为简，化简为零

精通AngularJS

AngularJS开发轻量化的、优雅的单页面Web应用

[美] Pawel Kozlowski 著
Peter Bacon Darwin
李路 王永强 马海波 译



华中科技大学出版社
<http://www.hustp.com>

提供各种书籍pdf下载，如有需要，请联系 QQ: 461573687

PDF制作说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ: 461573687, 或者 QQ: 2404062482。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

备用QQ:2404062482

PDF电子书说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

精通AngularJS

Mastering Web Application Development with AngularJS

[美] Pawel Kozlowski 著
Peter Bacon Darwin

李 路 王永强 马海波 译

华中科技大学出版社

内 容 介 绍

AngularJS 是一套基于 JavaScript 的前端开发框架,用于开发当下流行的数据驱动的单页面 web 应用。本书深入浅出地讲解了 AngularJS 的开发概念和原理,并通过丰富的开发实例向读者展示了构建复杂应用的完整过程,包括学习使用 AngularJS 特有的基于 DOM 的模板系统、实现复杂的后端通信、创建漂亮的表单、制作导航、使用依赖注入系统、提高 web 应用的安全性、使用 Jasmine 开展单元测试,等等。

图书在版编目(CIP)数据

精通 AngularJS / (美)科兹洛夫斯基(Kozlowski, P.), (美)达尔文(Darwin, P. B.) 著; 李路, 王永强, 马海波译. — 武汉: 华中科技大学出版社, 2014. 9

ISBN 978-7-5680-0396-4

I. ①精… II. ①科… ②达… ③李… ④王… ⑤马… III. ①JAVASCRIPT 语言-程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2014)第 212635 号

Original English language edition copyright © 2013 by Packt Publishing.

The Chinese translation edition copyright © 2014 by Huazhong University of Science and Technology Press in arrangement with Packt Publishing.

湖北省版权局著作权合同登记 图字:17-2014-281 号

书 名 精通 AngularJS
作 者 [美] Pawel Kozlowski, Peter Bacon Darwin
译 者 李路, 王永强, 马海波

策划编辑 徐定翔
责任编辑 陈元玉
责任校对 刘 竣
责任监印 周治超

出版发行 华中科技大学出版社(中国·武汉)
武昌喻家山(邮编 430070 电话 027-81321915)

录 排 武汉金睿泰广告有限公司
印 刷 湖北新华印务有限公司
开 本 787mm×960mm 1/16
印 张 23
字 数 448 千字
版 次 2014 年 10 月第 1 版第 1 次印刷
定 价 79.00 元

本书若有印装质量问题,请向出版社营销中心调换
全国免费服务热线 400-6679-118 竭诚为您服务
版权所有 侵权必究

致谢

Acknowledgments

Pawel Kozlowski：回想过去的几个月，我不敢相信，自己竟能有幸与如此之多出色的人一同为本书工作，没有你们的汗水和帮助，这段话就不可能出现在这里，谢谢你们。

首先，我想对属于 **Google** 的 **AngularJS** 团队全体成员说声谢谢，对于这个令人惊异的框架来说，你们是理想的梦幻团队，请继续你们出色的工作！特别要感谢 **Brad Green**、**Miško Hevery**、**Igor Minar** 和 **Vojta Jína**。**Brad**，感谢你为 **Peter** 和我介绍出版商，并鼓励我们撰写本书；**Miško**，感谢你审校本书，并容忍我们提出关于 **AngularJS** 那些天真的问题；**Igor**，你恒久的支持与从不间断的有用提示，让本书变好了很多很多。和你们这些家伙一同工作，实在是一件非常快乐的事。

我还要感谢整个 **AngularJS** 社区，特别是和我在邮件列表及其他论坛里交流过的人们。我没办法点出所有人的名字，但你们提出的那些深刻的问题是本书伟大的灵感之源。**AngularJS** 欣欣向荣背后的支持社区，是它如此出色的另一个理由。

感谢所有 **Packt** 出版社的人：**Rukhsana Khambatta**、**Dayan Hyames**，以及 **Arshad Sopariwala**，你们让整个编写和出版过程非常顺畅，感谢你们。

我要感谢 **Amadeus** 的同事们，在那里我学会了如何成为一名客户端程序员。首先要感谢的是我的领导，**Bertrand Laporte** 和 **Bruno Chabrier**。**Bertrand**，感谢你带我进入客户端开发的世界，并鼓励我撰写本书；**Bruno**，感谢你让我兼职工作以专注于本项目，谢谢你们两位的慷慨大度。也感谢 **Julian Descottes** 与 **Corinne Krych**，你们审校了本书的早期草稿，并提供了非常有价值的反馈。

非常感谢 **Peter** 同意与我共同撰写此书。**Peter**，我享受与你共同在这个项目中度过的每一分钟，做梦都不会想到有更好的合作者了。

最后也是最重要的，我想感谢我美丽的未婚妻 **Ania**，没有你无条件的支持与耐心，我甚至都不会想到开始与本书相关的工作。

Peter Bacon Darwin：我想感谢 Google 的团队，他们为我们带来了 AngularJS，特别是以下几位，Miško Hevery、Igor Minar、Brad Green、Vojta Jína，他们持续不断地给予我们灵感。我的合作者，Pawel，是本书背后的推动力量。他构思了本书的结构，撰写了大部分的内容，而且是个非常出色的同事。我们从 AngularJS 社区受益良多，尤其是从 AngularUI 项目那里获益良多。最后，如果没有我太太 Kelyn，还有孩子们（Lily 和 Zachary）的爱与支持，我想我无法完成本书。

审校者简介

About the Reviewers

Stephane Bisson 是 IT 咨询公司 ThoughtWorks 的程序员，住在加拿大多伦多。他也为医疗、金融和制造行业开发富 web 应用。

Miško Hevery 曾在 Google 担任敏捷教练，负责培训自动测试，帮助 Google 提高 web 应用的发布质量。在之前，他曾为 Adobe、Sun、Intel 和 Xerox 工作，精通 Java、JavaScript、Flex、ActionScript 应用。他积极参与开源活动，是好几个开源项目的作者，其中最著名的就是 AngularJS。

Lee Howard 是西北区健康教育中心（Northwest AHEC）的首席系统分析师。他为 Northwest AHEC 开发了一系列 web 应用。他开发了 CreditTrakr（针对 iOS 设备的移动应用），让内科医生得以追踪他们的 CME 积分。

作者简介

About the Authors

Pawel Kozlowski 有 15 年以上的 web 开发经验，接触过多种 web 技术、语言 and 平台。他精通客户端和服务端编程，并始终在寻找能提高生产率的工具和流程。

Pawel 是自由和开源软件的坚定支持者，他在 AngularJS 社区中非常活跃，为 AngularJS 项目贡献了大量的代码。他也对 Angular UI(AngularJS 的配套框架) 有所贡献，并为 AngularJS 提供了 Twitter Bootstrap 相关的指令 (directives)。

不写代码时，Pawel 常在会议与活动上为 AngularJS 布道。

Peter Bacon Darwin 已经积累了 20 年以上的编程经验，他在 .NET 发布之前，就已经用它工作了。他还对 IronRuby 的开发有所贡献。他曾在 Avanade 和 IMGROUP 做 IT 顾问。退休后，自由开发和照顾孩子占用了他大部分的时光。

Peter 是 AngularJS 社区的名人。他最近加入了 Google 的 AngularJS 团队。他还是 Angular UI 项目的创始成员。他曾在 Devvxx UK 和其他活动中发表过有关 AngularJS 的演讲，并开办过 AngularJS 的培训课程。他希望帮助企业更好地利用 AngularJS。

序

Preface

AngularJS 是新兴的 JavaScript MVC 框架，它为业界带来了重大的变化，包括对模板化 (templating) 的创新实现，以及数据的双向绑定 (two-way data binding)，这些特性使得它强大而易用。大多数开发者认为，与其他框架相比，AngularJS 明显缩减了项目所需的代码量。

AngularJS 是软件中的杰作，它强调测试与代码质量，在 JavaScript 生态圈内堪为表率，它的优质与创新，催生了周围活跃的程序员社区，这反过来又让 AngularJS 的人气更加旺盛。

随着 AngularJS 的愈发流行，人们开始在复杂的项目中运用它。很快，官方文档和在线示例已不足以解决遇到的难题。这时，像钻研任何技术一样，需要走得更深，去探索 AngularJS 的风格 (idioms) 和模式 (patterns)，以及社区所积累的最佳实践经验。

这也是写作本书的目的，不仅描述 AngularJS 如何工作，还会讲述如何轻松编写大型的 AngularJS web 应用，以及回答社区讨论的很多问题。

一言以蔽之，本书是由 AngularJS 应用的开发者写给同行的，以探讨他们会遇到的真实问题，你将从本书中学到：

- 运用 AngularJS 服务 (services) 和指令 (directives) 构建完善、健壮的应用。

目录

Table of Contents

序	1
第 1 章 Angular 之禅	7
1.1 邂逅 AngularJS	7
熟悉框架	8
参与 AngularJS 项目	8
社区	8
在线学习资源	9
库和扩展	9
工具	9
Batarang	10
Plunker与jsFiddle	10
IDE扩展和插件	10
1.2 AngularJS 速成	10
Hello World——AngularJS 示例	10
双向数据绑定	12
AngularJS 中的 MVC 模式	12
鸟瞰	13
深入作用域	15
视图	21
模块与依赖注入	26
模块	26
协作对象	27
注册服务	29
模块的生命周期	33
模块依赖	35
1.3 AngularJS 和其他框架	38
jQuery 与 AngularJS	39
苹果与橙子	40
窥视未来	41
1.4 总结	41
第 2 章 构建与测试	43

2.1 介绍示例应用	44
熟悉问题领域	44
技术栈	45
持久化存储	46
MongoLab	46
服务器端环境	47
第三方JavaScript库	48
Bootstrap CSS	48
2.2 构建系统	48
构建系统准则	49
自动化所有事情	49
尽早报错, 清晰报错	49
不同的工作流, 不同的命令	50
构建脚本同样是代码	50
工具	50
Grunt.js	51
测试库与工具	51
Jasmine	51
Karma runner	52
2.3 组织文件和目录	52
根目录	52
进入源代码目录	54
AngularJS的特定文件	54
轻装上路	56
深入测试目录	57
文件命名约定	57
2.4 AngularJS 模块和文件	57
一个文件, 一个模块	58
模块内部	59
注册provider的不同语法	59
声明配置和运行块的语法	61
2.5 自动化测试	62
单元测试	63
剖析Jasmine测试	64
测试AngularJS对象	65
测试服务	65
测试控制器	67
Mock对象和异步代码测试	68
端对端测试	70
日常工作流	71
Karma runner的提示与技巧	72

执行测试子集	73
调试	73
2.6 小结	74
第3章 与后端服务器通信	75
3.1 使用 \$http 进行 XHR 和 JSONP 请求	75
熟悉数据模型和 MongoLab URLs	76
\$http API 快速导览	76
配置对象说明	77
转换请求数据	78
处理 HTTP 响应	79
转换响应数据	79
处理同源政策约束	79
利用 JSONP 克服同源政策约束	80
JSONP 的限制	81
利用 CORS 克服同源政策约束	81
服务器端代理	83
3.2 promise API 与 \$q	84
工作中的 promise 和 \$q 服务	85
学习 \$q 服务的基础知识	85
promise 是第一类 JavaScript 对象	87
聚合回调	88
注册回调和承诺的生命周期	88
异步动作的链式调用	89
关于 \$q 的其他	91
AngularJS 中的 \$q 集成	93
3.3 promise API 与 \$http	94
3.4 与 RESTful 端点通信	95
\$resource 服务	95
构造级与实例级方法	97
\$resource 创建异步方法	100
\$resource 服务的限制	101
使用 \$http 自定义 REST 适配器	101
3.5 使用 \$http 的高级特性	104
截取响应	104
3.6 测试与 \$http 交互的代码	106
3.7 小结	108
第4章 显示与格式化数据	109
4.1 引用指令	109
4.2 显示表达式的求值结果	110

插值指令	110
利用 ngBind 渲染模型值	111
AngularJS 表达式中的 HTML 内容	111
4.3 条件化显示	112
根据条件包含内容块	114
4.4 用 ngRepeat 指令渲染集合	114
熟悉 ngRepeat 指令	115
特殊变量	115
迭代对象的属性	116
ngRepeat 模式	117
列表和细节	117
改动表格、行和类	119
4.5 DOM 事件处理器	120
4.6 基于 DOM 的模板	121
习惯烦琐的语法	121
ngRepeat 和多个 DOM 元素	122
不能在运行时修改的元素和属性	123
自定义 HTML 元素与 IE 的老版本	124
4.7 使用过滤器处理模型变换	124
内置过滤器	125
格式化过滤器	125
数组变换过滤器	125
编写自定义过滤器——分页示例	131
从 JavaScript 代码中访问过滤器	133
过滤器做什么与不做什么	134
过滤器与DOM操作	135
过滤器中代价高昂的数据变换	136
不稳定的过滤器	136
4.8 摘要	138
第 5 章 创建高级表单	139
5.1 AngularJS 表单与传统表单的比较	139
介绍 ngModel 指令	141
5.2 创建用户信息表单	142
5.3 理解输入指令	143
添加所需验证	143
使用基于文本的输入 (text、textarea、e-mail、URL、number)	143
使用 checkbox 输入	144
使用 radio 输入	145
使用 select 输入	145
提供简单的字符串 options	145

利用ngOptions指令提供动态options	146
select指令与空的options	148
理解select和对象判等	149
选择多个options	150
运用传统的 HTML hidden input 字段	150
嵌入来自服务器的值	150
提交传统的HTML表单	151
5.4 详解 ngModel 数据绑定	151
理解 ngModelController	151
在模型与视图之间转换值	152
追踪值是否变化	152
跟踪input字段有效性	153
5.5 校验 AngularJS 表单	153
理解 ngFormController	153
运用name属性将表单附加到作用域上	154
为用户信息表单增加动态行为	154
显示验证错误	155
让保存按钮无效	156
使原生浏览器校验无效	157
5.6 在其他表单中嵌套表单	157
将子表单作为可重用组件	157
5.7 重复子表单	158
验证重复输入	159
5.8 处理传统的 HTML 表单提交	161
直接向服务器提交表单	161
处理表单提交事件	161
使用ngSubmit处理表单提交	162
使用ngClick处理表单提交	162
5.9 重置用户信息表单	162
5.10 摘要	164
第 6 章 导航	165
6.1 单页 Web 应用的 URL	166
HTML5 之前的 Hashbang URL	166
HTML5 和 history API	167
6.2 使用 \$location 服务	168
理解 \$location 服务 API 与 URL 的关系	169
哈希、页面内导航和 \$anchorScroll	170
配置 HTML5 方式的 URL	171
客户端	171
服务端	171

使用 \$location 导航	172
根据路由构建页面	173
路由映射URL	174
定义路由时指定控制器	174
导航的不足	175
6.3 使用 AngularJS 自带的路由服务	175
基础路由定义	175
显示匹配的路由内容	176
匹配灵活的路由	177
定义默认路由	178
访问路由参数	178
多个控制器重用局部模板	178
路由改变时避免 UI 抖动	179
取消路由更新	181
6.4 \$route 服务的局限	182
一个路由只对应页面中的一个区域	183
使用ng-include处理多个UI区域	183
不支持嵌套路由	184
6.5 路由相关的模式及技巧	185
处理链接	185
创建可点击的链接	186
兼容HTML5及hashbang模式	186
链接外部页面	187
组织路由定义	187
将路由定义分离到多个模块	188
减少路由定义的重复代码	188
6.6 总结	189
第7章 安全	191
7.1 提供服务端认证和授权	192
处理未授权的访问	192
提供服务端验证 API	192
7.2 保护局部模板	193
7.3 阻止恶意攻击	194
防止 cookie 监听、中间人攻击	194
防止跨站脚本攻击	195
确保AngularJS 表达式内HTML 内容的安全性	195
允许不安全的HTML 绑定	196
净化HTML	196
防止 JSON 注入攻击	197
防止跨站请求伪造	198

7.4	客户端安全	198
	创建 security 服务	199
	显示登录表单	200
	创建安全的菜单及工具栏	201
	隐藏菜单项	201
	创建登录工具栏	202
7.5	支持客户端认证	203
	处理认证失败	203
	拦截响应	204
	HTTP响应拦截器	204
	创建 securityInterceptor 服务	205
	创建 securityRetryQueue 服务	207
	通知安全服务	208
7.6	防止导航到安全受限路由	208
	使用路由 resolve 函数	209
	创建授权服务	210
7.7	总结	212
第 8 章	创建自定义指令	213
8.1	什么是 AngularJS 指令	214
	理解内置指令	214
	在 HTML 标签中使用指令	215
8.2	指令的编译生命周期	215
8.3	为指令编写单元测试	217
8.4	定义指令	218
8.5	使用指令修改按钮样式	219
	编写一个按钮指令	220
8.6	理解 AngularJS 的组件指令	222
	编写一个分页指令	222
	为分页指令编写单元测试代码	223
	在指令中使用 HTML 模板	224
	从父作用域中隔离指令	225
	使用@插入属性	226
	使用=绑定数据	227
	使用&提供一个回调表达式	227
	实现分页组件	228
	为指令添加分页跳转回调	229
8.7	创建一个自定义验证指令	230
	需要其他指令的控制器	231
	可选的依赖控制器	231
	查找祖先元素的控制器	232

使用 ngModelController	232
编写自定义验证指令的单元测试	233
实现自定义验证指令	235
8.8 创建一个异步模型验证器	235
模拟用户服务	236
为异步验证编写测试代码	237
实现异步验证指令	238
8.9 包装 jQueryUI datepicker 指令	239
为包装组件指令编写测试代码	240
实现 jQuery datepicker 指令	242
8.10 小结	243
第 9 章 创建高级指令	245
9.1 使用嵌入	245
在指令中使用嵌入	245
在独立作用域指令中使用嵌入	246
创建一个使用嵌入的提示指令	246
理解指令定义中的 replace 属性	247
理解指令定义中的 transclude 属性	248
使用 ng-transclude 插入嵌入元素	248
理解嵌入作用域	248
9.2 创建和使用嵌入函数	250
使用 \$compile 服务创建一个嵌入函数	251
在嵌入时克隆原始元素	251
在指令中访问嵌入函数	252
通过编译函数中的 transcludeFn 来获取嵌入函数	252
通过 \$transclude 在指令控制器中获取嵌入函数	253
使用嵌入创建一个 if 指令	253
在指令中使用 priority 属性	255
9.3 理解指令控制器	256
为指令控制器注入特殊依赖	257
创建一个基于控制器的分页指令	258
理解指令控制器和链接函数的区别	258
注入依赖	259
编译过程	259
获取其他控制器	260
获取嵌入函数	261
创建一个手风琴指令套件	261
在手风琴组件中使用指令控制器	262
实现 accordion 指令	263
实现 accordion-group 指令	263

9.4 控制编译过程.....	265
创建一个 field 指令.....	265
在指令中使用terminal属性.....	267
使用 \$interpolate 服务.....	268
绑定验证信息.....	269
动态加载模板.....	269
设置 field 指令的模板.....	270
9.5 小结.....	271
第 10 章 创建为全球用户服务的 AngularJS 应用	273
10.1 使用本地化的符号和设置.....	274
配置本地化设置模块.....	274
使用已有的本地化设置.....	275
本地化设置和AngularJS过滤器.....	275
10.2 处理翻译.....	277
翻译 AngularJS 模板中的字符串.....	277
使用过滤器.....	278
使用指令.....	279
翻译 JavaScript 代码中的字符串.....	280
10.3 范式、秘诀和技巧.....	282
按照设定的地区初始化应用.....	282
将地区标识作为URL一部分带来的问题.....	283
切换地区.....	284
针对日期、数字和货币的自定义过滤器.....	285
10.4 小结.....	287
第 11 章 开发健壮的 AngularJS 应用	289
11.1 理解 AngularJS 的内部运作机制.....	290
AngularJS 不是基于字符串的模板引擎.....	290
响应DOM事件更新模型.....	291
将模型变化传播给DOM.....	291
同步DOM和模型变化.....	292
Scope.\$apply——打开AngularJS世界的钥匙.....	293
深入\$digest循环.....	295
整合.....	300
11.2 性能优化——设置期望值、测量、调节、并重复.....	301
11.3 AngularJS 应用的性能优化.....	303
优化 CPU 使用率.....	303
加速\$digest循环.....	303
尽可能少进入\$digest循环.....	310
限制每个\$digest循环的执行轮数.....	312

优化内存占用	312
尽可能避免深度监视	312
注意监视表达式的大小	314
ng-repeat 指令	314
ng-repeat指令中对集合的监视	314
瞬间绑定大量监视	315
11.4 小结	315
第 12 章 打包和部署 AngularJS Web 应用	317
12.1 提升网络相关的性能	318
压缩静态资源	318
AngularJS如何判断依赖关系	318
编写会被安全压缩的JavaScript代码	319
数组风格依赖注入的缺陷	322
模板预加载	323
使用<script>指令预加载模板	324
填充\$templateCache服务	325
组合使用不同的预加载技术	327
12.2 优化首页	327
避免显示未经处理的模板	328
使用ng-cloak指令隐藏DOM元素	328
使用ng-bind指令隐藏表达式	329
引入 AngularJS 和应用脚本文件	330
引用脚本文件	330
AngularJS和异步模块定义	331
12.3 浏览器支持	333
在 Internet Explorer 中使用	333
12.4 小结	334
索引	337

序

Preface

AngularJS 是新兴的 JavaScript MVC 框架，它为业界带来了重大的变化，包括对模板化 (templating) 的创新实现，以及数据的双向绑定 (two-way data binding)，这些特性使得它强大而易用。大多数开发者认为，与其他框架相比，AngularJS 明显缩减了项目所需的代码量。

AngularJS 是软件中的杰作，它强调测试与代码质量，在 JavaScript 生态圈内堪为表率，它的优质与创新，催生了周围活跃的程序员社区，这反过来又让 AngularJS 的人气更加旺盛。

随着 AngularJS 的愈发流行，人们开始在复杂的项目中运用它。很快，官方文档和在线示例已不足以解决遇到的难题。这时，像钻研任何技术一样，需要走得更深，去探索 AngularJS 的风格 (idioms) 和模式 (patterns)，以及社区所积累的最佳实践经验。

这也是写作本书的目的，不仅描述 AngularJS 如何工作，还会讲述如何轻松编写大型的 AngularJS web 应用，以及回答社区讨论的很多问题。

一言以蔽之，本书是由 AngularJS 应用的开发者写给同行的，以探讨他们会遇到的真实问题，你将从本书中学到：

- 运用 AngularJS 服务 (services) 和指令 (directives) 构建完善、健壮的应用。

- 内置功能不足时，对AngularJS进行扩展，包括指令（directives）、服务（services）和过滤器（filter）。
- 创建高质量的AngularJS项目，包括代码组织、构建（build）、测试和性能优化。

本书涵盖范围

What this book covers

第1章简要介绍 AngularJS 项目，大致勾画出了它的理念、主要概念及基本元素。

第2章通过引入应用示例，开始真正解决实际问题，并探讨系统构建与测试的实践方法。

第3章讲解如何从后端获取数据提供给 AngularJS 的 UI 组件，并开始介绍 promise API。

第4章介绍如何在 UI 中渲染后端数据。本章探讨 AngularJS 与 UI 渲染相关的指令（directives），以及用于格式化数据的过滤器（filters）。

第5章介绍如何操纵表单数据、处理不同类型的表单输入。本章列举了 AngularJS 支持的表单输入类型，并深入探讨了表单验证。

第6章介绍如何将单独页面组织成方便用户浏览的应用。本章解释了 URL 在单页面（single-page）web 应用中扮演的角色，并让读者熟悉 AngularJS 中管理 URL 和导航的关键服务（services）。

第7章介绍如何防止对 AngularJS 编写的单页面 web 应用的攻击。本章涵盖了验证与授权背后的概念与技巧。

第8章介绍 AngularJS 中最激动人心的部分之一——指令。本章将带领读者探索指令的结构，同时也将展现测试驱动的益处。

第9章在上一章的基础上介绍更多高级主题。本章有很多真实的指令示例，并力图描述幕后的复杂技术。

第10章介绍了如何翻译模板，以及管理本地化设置。

第11章介绍 web 应用的性能表现。本章揭开了 AngularJS 的面纱，帮助读者理解其内部机制，以避免常见的性能陷阱。

第 12 章将完成的应用发布到生产（production）环境。本章还介绍了如何定制你的应用入口页面。

阅读本书的准备

What you need for this book

只需要浏览器和文本编辑器（或者你最喜欢的 IDE），就可以运行本书中的 AngularJS 示例，不过，如果要充分利用本书，我们推荐安装 `node.js` (<http://nodejs.org>) 和它的 `npm` 包管理系统，然后在 `npm` 包管理系统中安装以下模块：

- Grunt (<http://gruntjs.com/>);
- Karma runner (<http://karma-runner.github.io>)。

此外，与后端交互的示例，会用到基于云平台的 MongoDB 数据库（MongoLab），所以，还将需要网络连接来运行很多例子。

本书的目标读者

Who this book is for

本书对于想在真实项目中运用 AngularJS 的开发者们尤为有用，事先对 AngularJS 有一点了解，也会很有帮助，哪怕只是几个简单的例子。当然，假定你有 HTML、CSS 和 JavaScript 的使用经验。

约定

Conventions

本书会使用几种字体来辨识不同的信息，下面是例子和解释。

文本中嵌入代码片断：可以用 `include` 指令来包含其他上下文。


一个代码块则看起来是这样：

```
angular.module('filterCustomization', [])
  .config(function ($provide) {
    var customFormats = {
      'fr-ca': {
        'fullDate': 'y'
      }
    }
  });
```

当希望特别强调某代码片段时，会使用粗体：

```
<head>
<meta charset="utf-8">
<script src="/lib/angular/angular.js"></script>
<script src="/lib/angular/angular-locale_<%= locale %>.js"></script>
<base href="/<%= locale %>/ ">
```

新的术语和重要的单词也会使用粗体。你在屏幕上看到的单词，比如在菜单或对话框中，我们会这样强调：“点击 **Next** 按钮会让你移动到下一屏”。

[ 警告或重要提示会出现在这样的框中。]

[ 提示和技巧则是这样。]

读者反馈 Reader feedback

我们时刻欢迎读者的反馈，以便了解你对本书的看法，这可以帮助我们今后的工作做得更好。

反馈的方法很简单，只要发邮件至 feedback@packtpub.com 就可以，记得在标题中提及书名。

如果你是某个领域的专家，并对编写书籍感兴趣，欢迎来 www.packtpub.com/authors 查看我们的作者指南。

读者服务 Customer support

现在你就是 Packt 图书的尊贵客户了，我们有一系列的支持来保障您的购物物有所值。

下载示例代码

Downloading the example code

你可以在 <http://www.packtpub.com> 下载全部在我们网站上所购买的 Packt 图书的示例代码。如果你是在其他地方购买的，请访问 <http://www.packtpub.com/support> 并注册，我们会直接将代码文件发邮件给你。

勘误

Errata

尽管我们已尽一切可能保证图书内容的正确，但仍有可能发生错误。如果你在我们的任何一本图书中发现了错误，无论是文字还是代码，非常欢迎你提交给我们。如此一来，你可以帮助其他读者更好地理解本书，并让我们在后续的版本中进行改进。提交的方法是访问 <http://www.packtpub.com/submit-errata>，选择你的图书，点击 **errata submission form** 链接，然后输入你的勘误细节。一旦我们确认你的勘误，就会将它上传到我们的网站上，或者任何已知的勘误表上。你可以在 <http://www.packtpub.com/support> 访问现有的勘误表。

关于盗版

Piracy

网络盗版是个对所有媒体来说都棘手的问题。在 Packt，我们非常严肃地保护我们的版权与许可。如果你在网络上发现了我们作品的任何形式的非法拷贝，请及时告知我们相关网址，以便于我们采取后续措施。

请通过 copyright@packtpub.com 告知我们你发现的盗版相关信息，我们将对你保护我们作家权利的行为深表敬意。

问题

Questions

你可以发邮件至 questions@packtpub.com 咨询你遇到的任何关于这本书的问题，我们会尽力让你满意。

第 1 章

Angular 之禅

Angular Zen

本章简要介绍 AngularJS。首先,是谁在推动这个项目,在哪里寻找源码和文档,如何寻求社区的帮助,等等。

然后,介绍 AngularJS 框架,包括核心概念和编程模式。为了让学习更轻松,我们会呈现大量代码示例。

AngularJS 是一个独一无二的框架,它无疑将在未来的 web 开发中占有重要地位。因此,本章最后会解释 AngularJS 的独特之处、与其他框架的比较及演进方向。

本章涵盖如下主题。

- 运用AngularJS编写Hello World 应用,了解去哪里寻找AngularJS的源代码、文档及社区。
- 熟悉AngularJS应用的基本元素:模板(template)、指令(directive)、作用域(scope)和控制器(controller)。
- 认识AngularJS依赖注入(dependency injection)系统的精妙之处。
- 理解AngularJS与其他库和框架的区别(特别是jQuery),以及它为何独树一帜。

1.1 邂逅 AngularJS Meet AngularJS

AngularJS 是采用 JavaScript 语言编写的客户端(client-side) MVC 框架,帮助开发者编写现代化的单页面(single-page)应用。它尤其适合编写有大量 CRUD(增删改查)操作的,具有 Ajax 风格的富客户端应用。

熟悉框架

Getting familiar with the framework

AngularJS 是近来不断涌现的客户端 MVC 框架中的新成员，它的出众要归功于创新的模板（templating）系统、友好的开发过程和可靠的工程实践。下面先来看看模板系统的独特之处。

- 使用HTML作为模板语言。
- 无须对DOM进行显式刷新，因为AngularJS可以通过用户动作、浏览器事件、模型（model）变化来决定在何时刷新何类模板。
- 有趣的和可扩展的组件子系统能够教会浏览器如何解释自定义的HTML标签及属性。

AngularJS 最显而易见的部分，也许就是上述的模板系统，但是，AngularJS 并不只是简单地包装一些与开发单页面 web 应用相关的功能与服务。

实际上，AngularJS 还隐藏着不少珠玑，如依赖注入（dependency injection, DI）及可测试性（testability）。通过 DI，可以用短小精炼且仔细测试过的服务（services）来组建 web 应用。另外，AngularJS 框架的设计和周边的工具群，在开发应用的每个阶段都鼓励良好的测试实践。

参与 AngularJS 项目

Finding your way in the project

在各种客户端 MVC 框架中，AngularJS 属于后起之秀。2012 年 6 月，AngularJS 才正式发布 1.0 版。但实际上在 2009 年，Miško Hevery（一位 Google 工程师）就启动了此项目（当时作为个人项目），之后，AngularJS 取得很大成功，于是获得 Google 公司官方支持并为此组建了一支全职团队。

AngularJS 是发布在 GitHub (<https://github.com/angular/angular.js>) 上的开源项目，版权属于 Google 公司，遵循 MIT 许可证（the MIT license）。

社区

如果缺乏支持，那么项目就无法存续。所幸 AngularJS 拥有出色的社区，下面就是一些可以讨论框架设计或寻求帮助的渠道：

- angular@googlegroups.com邮件列表（Google group）。

- **Google+ 社区** (<https://plus.google.com/u/0/communities/11536882070087033075>)。
- **#angularjs IRC 频道**。
- <http://stackoverflow.com> 上的 **[angularjs]** 标签。

AngularJS 团队成员和社区保持着密切的联系，其博客地址是：<http://blog.angularjs.org>。此外，他们还在 **Google + (+AngularJS)** 及 **Twitter (@angularjs)** 上活跃。世界上很多地方都有社区自发组织的聚会，如果你家附近有，千万不要错过！

在线学习资源

AngularJS 主页 (<http://www.angularjs.org>) 上包罗万象：概念上的综览、教程、开发者指南、API 参考等。AngularJS 所有版本的源代码都可以在 <http://code.angularjs.org> 上下载。

寻找代码例子的人们会欣喜地发现，AngularJS 的文档中有大量代码片段。此外，还可以在 <http://builtwith.angularjs.org> 看到一系列使用 AngularJS 构建的应用。甚至在 YouTube 上还有 AngularJS 频道 (<http://www.youtube.com/user/angularjs>)，记录了以往的重大事件以及有用的视频教程。

库和扩展

Libraries and extensions

尽管 AngularJS 的核心部分已经包括大量功能，但活跃的社区几乎每天都在提交新的扩展，在网站 <http://ngmodules.org> 上可以找到其中的大部分。

工具

Tools

AngularJS 依赖于 HTML 和 JavaScript——web 开发的两位老朋友。所以，我们可以继续用最喜欢的编辑器、IDE、浏览器扩展等开发 AngularJS 应用。此外，AngularJS 社区也在不断为现存的 HTML/JavaScript 工具库添砖加瓦。

Batarang

Batarang 是一个 Chrome 开发工具类插件，可用于检视基于 AngularJS 的 web 应用，方便查看并可视化应用的运行时特征。本书会运用 Batarang 查看应用内部的信息。Batarang 可以像任何 Chrome 扩展一样，通过 Chrome 在线商店 (*AngularJS Batarang*) 安装。

Plunker与jsFiddle

Plunker (<http://plnkr.co>) 和 jsFiddle (<http://jsfiddle.net>) 让分享代码片段 (JavaScript、CSS 与 HTML) 变得容易。尽管这些工具不仅限于为 AngularJS 所用，但它们依然在短时间内被 AngularJS 社区吸纳，用于共享短小的代码示例，以及重现 bug 等。这里需要特别提及 Plunker，因为它由 AngularJS 所编写，在社区中相当流行。

IDE扩展和插件

我们都有自己中意的 IDE 或编辑器，幸运的是，几个人气非常高的 IDE 都有对应的 AngularJS 扩展或插件，包括 Sublime Text 2 (<https://github.com/angular-ui/AngularJS-sublime-package>)、Jet Brains' products (<http://plugins.jetbrains.com/plugin?pr=idea&pluginId=6971>) 等。

1.2 AngularJS 速成 AngularJS crash course

了解如何找到源代码和文档后，就应该实际编写代码来看看 AngularJS 如何工作了。本节涵盖 AngularJS 的模板、模块化(modularity)及依赖注入(dependency injection)等基本元素，从而为后续章节打下良好的基础。

Hello World——AngularJS 示例 Hello World — the AngularJS example

我们将通过经典的 “Hello, World!” 与 AngularJS 初次见面：

```
<html>
<head>
  <script src="http://ajax.googleapis.com/ajax/libs/
angularjs/1.0.7/angular.js"></script>
</head>
```

```
<body ng-app ng-init="name = 'World'">
  <h1>Hello, {{name}}!</h1>
</body>
</html>
```

首先，要包含 AngularJS 库。这很简单，因为它就是一个包装好的 JavaScript 文件。



AngularJS库相当小：最小化（minified）并用gzip压缩后只有30KB。AngularJS库最小化但没有压缩的版本则是80KB，它不依赖其他库。

本书中的简短示例使用部署在Google的内容分发网络（content delivery network, CDN）上的对开发者友好的非最小化版本。所有版本的AngularJS源代码都可以在<http://code.angularjs.org>下载。

仅包含 AngularJS 库是不能运行示例的，还需要引导（bootstrap）它。最简单的方法是使用 AngularJS 自定义的 `ng-app` HTML 属性。

仔细观察 `<body>` 标签，能看到另一个自定义的 HTML 属性：`ng-init`。在模板渲染前，可用 `ng-init` 初始化模型（model）`name`，并通过 `{{name}}` 表达式来传递它的值。

通过这个非常简单的例子，AngularJS 模板系统表现出了一些重要特征。

- 使用自定义的HTML标签与属性，为静态HTML文档添加动态行为。
- 使用双花括号作为输出模型值的表达式的分隔符（`{{expression}}`）。

在 AngularJS 中，所有能够被框架理解和解释的特殊 HTML 标签和属性，统称为指令（directives）。

双向数据绑定

AngularJS 渲染模板简单直接，这在创建动态 web 应用时相当便利。为了展现它的威力，让我们为“Hello World”增加一个输入栏（input field），代码如下：

```
<body ng-app ng-init="name = 'World'">
  Say hello to: <input type="text" ng-model="name">
  <h1>Hello, {{name}}!</h1>
</body>
```

除了新增加的 ng-model 属性外，这个 <input>HTML 标签看上去没有什么特别。但当我们开始向 <input> 输入文本时，奇迹发生了。在每一次击键后，页面都被重绘，以显示你输入的名字！我们无须编写代码去刷新模板，也无须调用框架的 API 去更新模型。AngularJS 智能地检测到了模型的变化，并随之更新了页面 DOM 元素。

大部分传统的模板系统，对模板的渲染是个线性单向的过程：模型（或变量）与模板混合在一起产生结果的标记集合。任何对模型的变化都需要通过模板的重新计算。但 AngularJS 有所不同，任何用户引发的视图（view）的改变，都会映射在模型上，继而任何模型的变化，也会立刻传播到整个模板。

AngularJS 中的 MVC 模式

The MVC pattern in AngularJS

当前大部分 web 应用都基于知名的模型 - 视图 - 控制器（model-view-controller, MVC）模式的某形态。但是准确来说，MVC 并不是编程模式，而是更高级的架构（architect）模式，因此相当抽象。更糟糕的是，当前有很多 MVC 模式的变种和派生（如目前最流行的 MVP 和 MVVM 模式），开发者们都在按照自己对 MVC 的理解在设计架构。这导致了同样的名字（MVC）用于描述不同的架构和编码方式。Martin Fowler 在他关于 GUI 架构的精彩文章（<http://martinfowler.com/eaaDev/uiArchs.html>）中总结了这一点：

“就拿 MVC 来举例吧。它经常被认为是一种模式，但我认为把它当成模式并不十分有用，因为它包括了不少不同的想法。不同的人在不同的地方对 MVC 有不同的理解，但这些理解统一都称为‘MVC’。如果这还不能让你感到困惑，那么想想越传越走样的传话游戏（chinese whispers）吧，这就是对 MVC 的误解引发的结果。”

在 MVC 家族中，AngularJS 采用了注重实效的方式，他们宣布自己的框架基于 MVW(model-view-whatever) 模式。口说无凭，让我们实际看一下代码的运行吧。

鸟瞰

至今为止的“Hello World”示例，没有明显的分层策略：数据初始化、逻辑和视图都混在了同一文件中。然而，在真实的应用里，我们需要仔细考虑将职责分别指派给每个抽象层。幸运的是，AngularJS 为构建复杂应用提供了架构级的结构体。

 为了简洁起见，本书后续的例子将省略 AngularJS 的初始化代码（代码库包含 ng-app 等）。

让我们看一下略微修改过的“Hello World”示例。

```
<div ng-controller="HelloCtrl">
  Say hello to: <input type="text" ng-model="name"><br>
  <h1>Hello, {{name}}!</h1>
</div>
```

移除了 ng-init 属性后，取而代之的是新指令 ng-controller 及对应的 JavaScript 函数 HelloCtrl。该函数接受神秘的参数 \$scope，代码如下：

```
var HelloCtrl = function ($scope) {
  $scope.name = 'World';
}
```

下面将解释 \$scope 和作用域 (scope) 的概念。

1. 作用域

AngularJS 中的 \$scope 对象是模板的域模型 (domain model)，也称为作用域实例 (instance)。通过为其属性赋值，可以传递数据给模板渲染。

作用域 (scope) 可以加入与模板相关的数据和提供相关的功能。例如，为作用域实例定义方法，以封装 UI 交互逻辑供模板使用。

例如，为 `name` 变量创建读取器（getter）函数，代码如下：

```
var HelloCtrl = function ($scope) {
    $scope.getName = function() {
        return $scope.name;
    };
}
```

然后在模板中调用它：

```
<h1>Hello, {{getName()}}!</h1>
```

`$scope` 对象可以精准地控制领域模型（domain model）的哪些数据和操作在视图（view）上是有效的。从概念上来说，AngularJS 的作用域（scopes）与 MVVM 模式的视图模型（view model）非常相似。


2. 控制器

控制器（controller）的主要职责是初始化作用域实例。在实践中，初始化逻辑又可以再分为：

- 提供模型的初始值。
- 增加UI相关的行为（函数）以扩展 `$scope` 对象。

控制器实际上就是 JavaScript 函数。它们不需要扩展任何 AngularJS 特定的类，也不需要调用任何特定的 AngularJS API，就可以正常工作。

[




请注意，在设定模型初始值时，控制器和 `ng-init` 指令做同样的工作。控制器可以让你在 JavaScript 代码中表达初始化逻辑，而不是污染 HTML 模板。

]

3. 模型

AngularJS 的模型（model）实际上就是普通的 JavaScript 对象。与控制器类似，不需要特别地去扩展任何 AngularJS 的底层类，也不用去构造（construct）模型对象。

可以在模型层中使用任何当前存在的纯 JavaScript 的类或对象。模型可拥有的属性也不仅限于原始值（primitive values），任何有效的 JavaScript 对象或数组都是可以的。只需要将模型简单地指派给 `$scope`，AngularJS 就可确认它的存在。

 AngularJS没有侵略性，它让模型对象远离框架特定的代码。

深入作用域

每个 `$scope` 都是 `Scope` 类的实例。`Scope` 类拥有很多方法，用于控制作用域的生命周期、提供事件传播（**event-propagation**）功能，以及支持模板的渲染等。

1. 作用域层级


下面再看之前的 `HelloCtrl` 示例：

```
var HelloCtrl = function ($scope) {
    $scope.name = 'World';
}
```

`HelloCtrl` 看起来像一般的 JavaScript 构造函数（**constructor function**），只有 `$scope` 参数令人疑惑，它是从何而来的呢？

答案是，`ng-controller` 指令会调用 `scope` 对象的 `$new()` 方法创建新的作用域 `$scope`。看上去需要至少一个作用域实例才能创建新的作用域。确实如此，**AngularJS** 拥有 `$rootScope`，它是其他所有作用域的父作用域，将在新应用启动时自动创建。

`ng-controller` 指令是作用域创建（**scope-creating**）指令。当在 **DOM** 树中遇到作用域创建指令时，**AngularJS** 都会创建 `Scope` 类的新实例 `$scope`。新创建的作用域实例 `$scope` 会拥有 `$parent` 属性，并指向它的父作用域。在 **DOM** 树中，会有很多这样的指令创建出很多作用域。

 众多作用域形成了以 `$rootScope` 为根的树结构。鉴于 **DOM** 树驱动了作用域的创建，作用域树模仿 **DOM** 树的结构就不奇怪了。

某些指令会创建子作用域，也许你会奇怪为什么要搞得这么复杂。为了说明原因，我们来看使用 `ng-repeat` 重复器（**repeater**）指令的例子。

控制器如下：

```
var WorldCtrl = function ($scope) {
  $scope.population = 7000;
  $scope.countries = [
    {name: 'France', population: 63.1},
    {name: 'United Kingdom', population: 61.8},
  ];
};
```

HTML 标记片断如下：

```
<ul ng-controller="WorldCtrl">
  <li ng-repeat="country in countries">
    {{country.name}} has population of{{country.population}}
  </li>
  <hr>
  World's population: {{population}} millions
</ul>
```

ng-repeat 指令为国家集合中每个国家创建新的 DOM 元素。指令的语法很容易理解，对应每项创建新变量 **country**，并暴露给 **\$scope** 以在视图中渲染。

但这里有个问题，对应每个 **country**，都有个新变量要暴露给 **\$scope**，而又不能覆盖之前变量的值。AngularJS 为了解决此问题，给集合中每个元素创建新的作用域。这些作用域会组成类似 DOM 树的层级（**hierarchy**）结构，运用 Chrome 扩展 **Batarang**，可以直观地看到如图 1-1 所示截图。

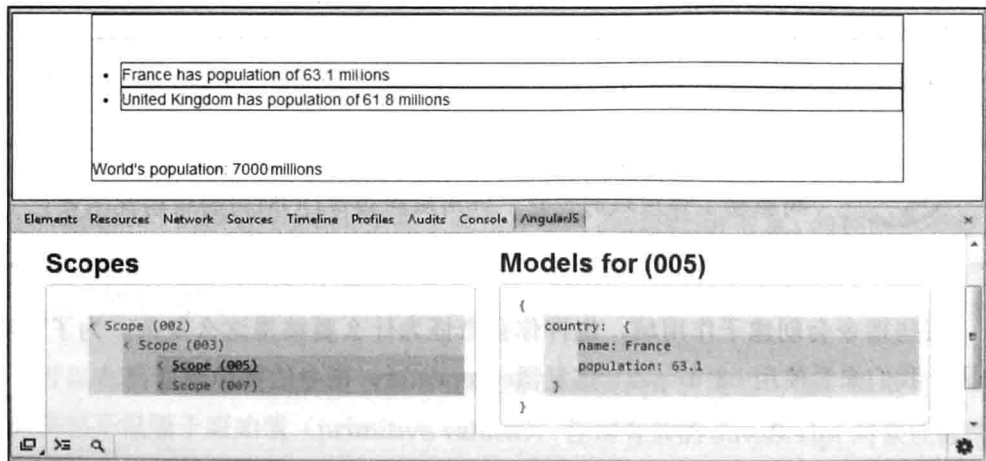


图1-1

就像截图中看到的，每个作用域（边界被长方形所标记）有自己的整套模型值（**model values**）。在不同的作用域中，定义同名的变量，不会造成命名冲突（不同的 **DOM** 元素指向不同的作用域，并使用各自作用域中的变量渲染模板），这相当于集合中每个项目都有自己的命名空间（**namespace**）。在前面的例子里，每个 `` 元素都有自己的作用域，在里边分别定义自己的 `country` 变量。

2. 作用域层级和继承

作用域中定义的属性对所有子作用域是可见的，只要子作用域没有定义同名的属性。在实践中这十分有用，再也不需要为了让某属性在作用域层级（**hierarchy**）中可见，而一遍一遍地重复定义它。

在之前示例的基础上，我们想显示出居住在给定国家的人口占世界人口的百分比。为了满足此需求，在 `WorldCtrl` 管理的作用域中定义 `worldsPercentage` 函数，代码如下：

```
$scope.worldsPercentage = function(countryPopulation){
    return (countryPopulation / $scope.population)*100;
}
```

之后，在 `ng-repeat` 指令创建的作用域中调用此函数：

```
<li ng-repeat="country in countries">
    {{country.name}} has population of{{country.population}},
    {{worldsPercentage(country.population)}} % of the World's
    population
</li>
```

AngularJS 中的作用域继承（**inheritance**）和 JavaScript 中的原型继承（**prototypical inheritance**）遵循同样的规则（沿继承树向上查找属性，直至找到为止）。

3. 在作用域层级中继承的风险

进行读操作（**read access**）时，作用域层级的继承关系直观易懂。然而，进行写操作（**write access**）时情况就有些复杂了。

在父作用域中定义变量，而在子作用域中不管它，我们看看会发生什么，代码如下：

```
var HelloCtrl = function ($scope) {
}
```

视图代码则如下：

```
<body ng-app ng-init="name='World'">
<h1>Hello, {{name}}</h1>
<div ng-controller="HelloCtrl">
  Say hello to: <input type="text" ng-model="name">
  <h2>Hello, {{name}}!</h2>
</div>
</body>
```

运行这段代码会发现，`name` 变量被定义在了最顶层的作用域中。然而，它在整个应用中到处可见。这说明，变量会沿着作用域层级向下继承。换句话说，定义在父作用域中的变量，在子作用域中也可以访问。

现在，在 `<input>` 输入框中开始打字，我们看看会发生什么，如图 1-2 所示。

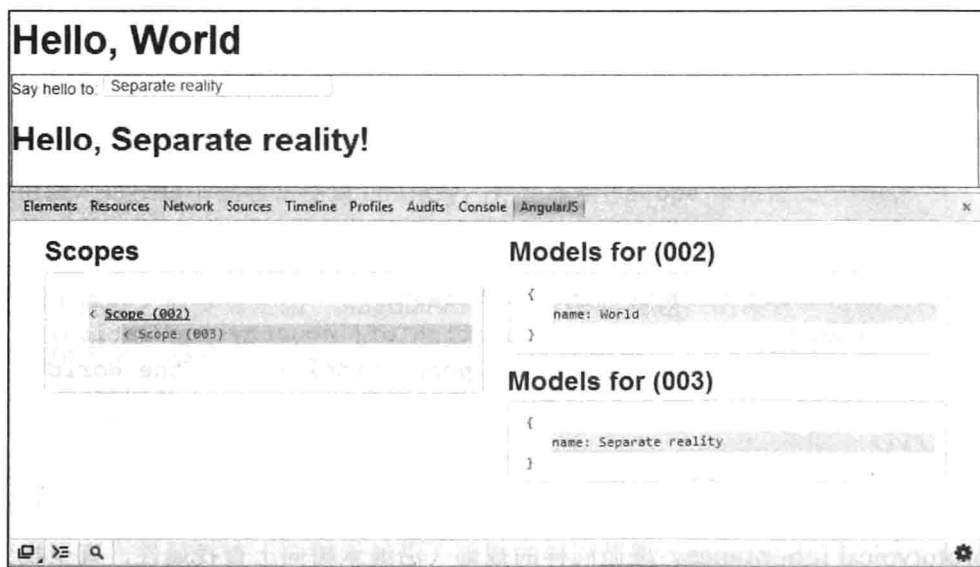


图1-2

你也许会感到惊讶，`HelloCtrl` 控制器生成的作用域中创建出了新的 `name` 变量，而不是去修改 `$rootScope` 设定的 `name` 变量的值。但是，如果你了解作用域之间是原型继承（**prototypical inheritance**）关系，就不会那么惊讶了。**JavaScript** 对象原型继承的所有规则都适用于作用域之间的原型继承，毕竟，作用域也是 **JavaScript** 对象。

有几种方法可以从子作用域中影响定义在父作用域中的属性。首先，可以利用 `$parent` 属性明确地引用父作用域。修改后的模板如下所示：

```
<input type="text" ng-model="$parent.name">
```

在本例中，这确实能解决问题，但要认识到，这种解决方案相当不可靠。问题出在 `ng-model` 指令所用的表达式对 DOM 结构做了武断的假设。如果在 `<input>` 标签上的某处插入了另外的作用域创建指令，那么 `$parent` 就会指向完全不同的作用域。



尽量避免使用 `$parent` 属性，因为它在 AngularJS 表达式和模板创建的 DOM 结构间建立了强关联。对 HTML 结构轻微的改变就可能毁掉整个应用。

另一种解决方案是将变量绑定（binding）为某对象的属性，而不是直接绑定为作用域的属性。代码如下：

```
<body ng-app ng-init="thing = {name : 'World'}">
<h1>Hello, {{thing.name}}</h1>
<div ng-controller="HelloCtrl">
  Say hello to: <input type="text" ng-model="thing.name">
  <h2>Hello, {{thing.name}}!</h2>
</div>
</body>
```

这是更好的解决方案，它没有对 DOM 树结构做任何假设。



避免直接绑定变量给作用域属性。对象属性（已暴露给作用域）的双向数据绑定（two-way data binding）是更好的方案。

经验之谈，记得在提供给 `ng-model` 指令的表达式中要有点（比如，`ng-model="thing.name"`）。

4. 作用域层级与事件系统

作用域层级，可以看成传送事件的列车。**AngularJS** 允许跨越作用域层级，传播带有信息的命名事件（**named events**）。事件可以从任何作用域开始分发（**dispatch**），然后向上分发（**\$emit**）或向下广播（**\$broadcast**），如图 1-3 所示。

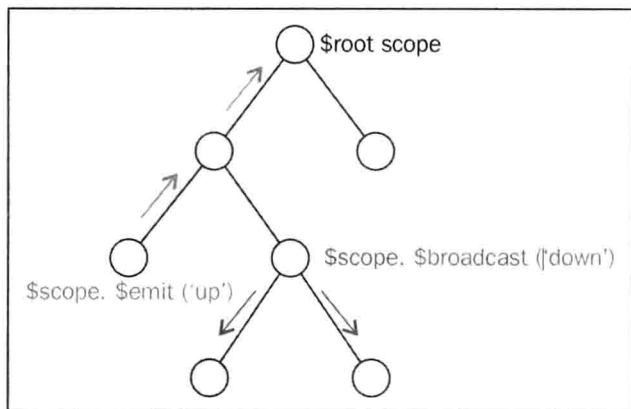


图1-3

AngularJS 的核心服务（**services**）与指令通过这趟事件列车来发送信号，通知应用状态的重要变化。例如，可以监听 **\$locationChangeSuccess** 事件（由 **\$rootScope** 广播），当地址（浏览器 **URL**）变化时我们会收到通知，代码如下：


```
$scope.$on('$locationChangeSuccess', function(event, newUrl, oldUrl){

    //在这里对地址(location)的变化做出反映
    //例如，根据新的Url更新面包屑(breadcrumbs)导航


});
```

每个作用域实例都有 **\$on** 方法，用于注册（**register**）作用域事件（**scope-event**）的处理器（**handler**）。调用处理器函数时，被分发的（**dispatched**）**event** 对象会作为第一个参数传入，后续参数则会依据事件负载的信息和事件类型（**event-type**）而定。

与 **DOM** 事件一样，**event** 对象有 **preventDefault()** 和 **stopPropagation()** 方法。**stopPropagation()** 会阻止事件通过作用域层级冒泡（**bubbling up**），即它仅在事件向上分发（**\$emit**）时有效。

 尽管AngularJS的事件系统模仿DOM的事件系统，但它们的传播机制却完全独立，没有共通之处。

对于某些问题来说，虽然跨越作用域层级的事件传播（特别是全局的、异步的、关于状态变迁的通知）是非常优雅的解决方案，但还是应该谨慎使用它们。通常可以依赖双向数据绑定，更便捷地解决问题。在 AngularJS 框架中，仅有三个事件（`$includeContentRequested`、`$includeContentLoaded`、`$viewContentLoaded`）可以被向上分发（**emitted**），七个事件（`$locationChangeStart`、`$locationChangeSuccess`、`$routeUpdate`、`$routeChangeStart`、`$routeChangeSuccess`、`$routeChangeError`、`$destroy`）可以被向下广播（**broadcasted**）。如你所见，使用作用域事件是相当保守的，我们应该在发送自定义事件前考虑其他选项（大部分情况下是双向数据绑定）。

 不要在AngularJS中试图模仿DOM的基于事件的编程模型。大部分情况下，最好使用双向数据绑定。

5. 作用域生命周期

作用域提供独立的命名空间，避免变量的名称冲突。因此，小型、组织良好的作用域，能节约内存的使用。作用域在不需要后会被销毁，在其中暴露的（**exposed**）模型和函数都会被垃圾回收。

作用域通常会依赖作用域创建指令而创建和销毁，也可以调用 `Scope` 类上的 `$new()` 和 `$destroy()` 方法，手动创建和销毁作用域。

视图

我们已经列举了不少 AngularJS 模板的例子，它没有独立的模板语言，却相当强大。使用 **HTML** 作为模板语法的 AngularJS，不但允许我们扩充 **HTML** 的词汇库，而且无须手动干预刷新页面上的任何部分。

实际上, AngularJS 与 HTML 和 DOM 有着更加紧密的联系, 因为 AngularJS 依靠浏览器去解析模板文本 (就像浏览器对任何 HTML 文档所做的)。浏览器将文本转换成 DOM 树之后, AngularJS 参与进来, 开始遍历 (traverse) 解析好的 DOM 结构。当遇到指令时, AngularJS 就执行相关逻辑, 将指令转换成页面的动态部分。



因为 AngularJS 依靠浏览器去解析模板, 所以要保证模板是有效的 HTML。尤其要小心闭合好 HTML 标签 (未闭合好的标签不会产生任何错误信息, 但会让视图不能正常渲染)。AngularJS 在有效的 DOM 树下才能好好工作。

AngularJS 扩展了 HTML 的词汇库 (增加新的属性或 HTML 元素, 并告知浏览器如何解释它们), 这类似于创造了一门基于 HTML 的领域特定语言 (domain-specific language, DSL), 并指导浏览器如何理解它。因此, 常听到 AngularJS “教浏览器新把戏” 的说法。

声明式模板视图——命令式控制器逻辑

AngularJS 提供了很多便利的指令, 接下来的章节会触及其中的大部分指令。然而, 重要的不是这些指令的语法或功能, 而是隐藏在表象之下的关于如何构建 UI 组件的 AngularJS 哲学。

AngularJS 提倡声明式 (declarative) UI 结构。在实践中, 这意味着模板专注于描述所需要的效果, 而非其实现。听上去有些困惑? 那让我们举例来说明吧!

我们希望创建一个让用户可以输入短消息的表单, 然后点击按钮来发送它。这里有些特别的用户体验 (user-experience, UX) 需求, 短消息字数要限制在 100 字以内, 一旦超过此限制, “Send” (发送) 按钮就要被禁用掉。当用户输入消息时, 他应该知道还可以输入多少字。一旦剩余字数小于 10, 就标示它的数字应该改变显示样式, 以警告用户。另外, 它还应该可以清除用户已经输入的消息。完成的表单如图 1-4 所示。

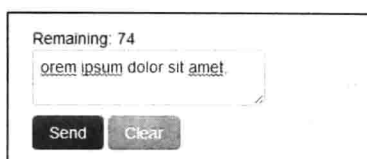


图1-4

上面的需求是相当标准的文本表单，没有多少挑战性。尽管如此，也需要很多 UI 元素协同工作，例如，正确管理按钮的 disabled 状态，采用适当样式显示剩余可输入的准确字数等。初次尝试的代码如下：

```
<div class="container" ng-controller="TextAreaWithLimitCtrl">
  <div class="row">
    <textarea ng-model="message">{{message}}</textarea>
  </div>
  <div class="row">
    <button ng-click="send()">Send</button>
    <button ng-click="clear()">Clear</button>
  </div>
</div>
```

我们用上面的代码作为基础。首先，要显示剩余字数，这很容易，代码如下：

```
<span>Remaining: {{remaining()}}</span>
```

remaining() 函数定义在 TextAreaWithLimitCtrl 控制器创建的作用域下，代码如下^[1]：

```
$scope.remaining = function () {
  return MAX_LEN - $scope.message.length;
};
```

其次，在短消息超出字数限制时，禁用“Send”按钮。这可以用 ng-disabled 指令搞定，代码如下：

```
<button ng-disabled="!hasValidLength()"...>Send</button>
```

[1] 译者注：在实践中，建议在这里判断 \$scope.message 是否已定义，如果没有定义，则应直接返回 MAX_LEN。

我们看到了重复出现的模式：仅需修改模板的很少一部分，就可以操纵 UI，用模型状态（本例中是消息的长度）得出想要的结果（剩余字数的显示、按钮的禁用等）。令人激动的是，不用在 JavaScript 代码中保存任何对 DOM 元素的引用，也不用显式地操纵 DOM 元素。我们只是专注于模型的变化，而让 AngularJS 去做辛苦的工作。所有需要做的，只是以指令的形式去提供一些提示而已。

回来继续看上面的例子。下一步是，当可以输入的文字数量已经不多时，改变剩余字数的显示样式。这里可以看到更多的命令式（declarative）UI 示例，代码如下：

```
<span ng-class="{ 'text-warning' : shouldWarn() }">  
  Remaining: {{remaining()}}  

```

```
</span>
```

shouldWarn() 方法的实现代码如下：

```
$scope.shouldWarn = function () {  
  return $scope.remaining() < WARN_THRESHOLD;  
};
```

模型的改变驱动了 CSS 类的改变，而且，在代码的任何地方都没有显式的 DOM 操纵逻辑。UI 基于声明式的表达“愿望”，而重新绘制。ng-class 指令所表述的是这样：“每次当用户输入快要超出字数限制的时候，应该看到警告， 元素应该有 text-warning CSS 类。”这和另一种表述方式截然不同：“当输入新的字符促使字符数量超出限制时，找到一个 元素，并为此元素添加 text-warning CSS 类。”

以上两个表述虽然听起来差别不大，但是，实际上声明式（declarative）和命令式（imperative）是针锋相对的。命令式编程风格专注于描述通向理想结果的步骤，而声明式编程风格专注于描述结果本身，步骤则主要由支持的框架来处理。这就好像说：“亲爱的 AngularJS，当模型在特定状态时，我想要 UI 看起来是这样。去吧，决定何时及怎样重绘 UI 吧！”

通常来说，声明式编程风格更具表现力，它让开发者从严格的底层实现中解放出来，写出简洁易读的代码。但是，为了让声明式编程风格工作，需要能够正确解释高级命令的机制。一方面，程序要依赖这些机制，就要放弃某种程度的底层控制。另一方面，如果采用命令式编程风格，就可以控制一切调整每个细节。但是，“掌控一切”的代价，就是要编写大量底层的、重复的代码。

熟悉 SQL 语言的人会发现这听起来似曾相识（SQL 就是很有表现力的用于查询特定数据的声明式语言）。简单地描述想要的结果（要获取的数据），然后让（关系型）数据库决定怎样获得指定数据。大部分情况下，此过程工作良好，为我们获取想要的数据库。但有些时候，为了优化性能，需要提供额外的线索（索引、查询线索等），或者亲自控制一些细节。

AngularJS 的指令用声明表达出了需要的效果，所以不用再去按部就班地修改 DOM 元素（就像那些 jQuery 应用）。AngularJS 提倡在模板中使用声明式编程风格，而在 JavaScript 代码中（控制器和业务逻辑）使用命令式编程风格。有了 AngularJS 的支持，很少需要步骤详细的底层 DOM 操纵（唯一的例外是实现指令的代码）。



重要规则：永远不要在 AngularJS 的控制器中操纵 DOM 元素。在控制器中获取对 DOM 元素的引用、操纵元素的属性，预示着用命令式编程风格来构建 UI——这背离了 AngularJS 之道。

由 AngularJS 指令编写的声明式的 UI 模板，能够快速描述复杂的交互性 UI。AngularJS 会接管何时及如何操纵 DOM 树的所有底层决定。大部分情况下，AngularJS 做“正确的事”，并如期望的那样，及时更新 UI。但是，理解 AngularJS 的内部工作机制依然很重要，这样可以在 AngularJS 需要时，为它提供适当的线索。再次运用 SQL 的比喻，大部分情况下我们不需要担心查询规划器（query planner）所做的工作，但当遇到性能问题时，最好能了解查询规划器是如何做出决定的，以便于我们为它提供更多的线索。同样的道理也适用于 AngularJS 所管理的 UI：想要更有效地使用模板和指令，就要了解更多的内部机制。

模块与依赖注入

Modules and dependency injection

机警的读者也许已经注意到，之前给出的例子都使用了全局构造函数来定义控制器（**controller**）。但是，全局状态是危险的，它会伤害应用的结构，让代码难以维护、测试和阅读。**AngularJS** 不建议使用全局状态；相反，它拥有全套 API，用来定义模块（**module**）及在模块中注册对象。

模块

让我们看看如何将丑陋的、全局定义的控制器的模块化（**modular**），目前代码如下：

```
var HelloCtrl = function ($scope) {  
    $scope.name = 'World';  
}
```

模块化后代码如下：


```
angular.module('hello', [])  
    .controller('HelloCtrl', function($scope){  
        $scope.name = 'World';  
    });
```

AngularJS 为自己定义了全局命名空间（**namespace**）**angular**，它提供多种功能及不少便利函数，**module** 就是其中之一。**module** 为 **AngularJS** 管理的对象（控制器、服务等）扮演容器的角色。正如很快将看到的，模块化远不只是命名空间与代码组织，还有很多东西要学。

定义新的模块，需要传入名字，作为调用 **module** 的第一个参数，而第二个参数则表达此模块依赖哪些其他模块（上例中的模块不依赖其他模块）。


angular.module 的调用会返回新创建模块的实例，然后就开始定义新的控制器，这相当容易，只要用下面的参数调用 **controller** 函数即可：

- 控制器的名字（字符串类型）。
- 控制器的构造函数。

 全局定义的控制器构造函数，只在简单代码示例和快速制作原型时才有用，永远不要在复杂的真实应用中使用全局定义的控制器。

模块已经定义好了，现在要告知 AngularJS 它的存在，这只要为 `ng-app` 属性赋值即可：

```
<body ng-app="hello">
```

 忘记为 `ng-app` 属性指定模块的名字，是经常会发生的错误，很让人头疼，它会导致控制器未定义的错误说明。

协作对象

如你所见，AngularJS 用模块组织对象。在模块上可以注册的对象，不仅限于 AngularJS 对象（控制器、过滤器等），还包括开发者自定义的任何对象。

模块模式在组织代码方面非常有用，但是，AngularJS 走得更远一些。除了在模块中注册对象外，它还可以声明这些对象的相互依赖关系。

1. 依赖注入

如前所述，`$scope` 对象被神秘地注入了控制器中。实际上，这是因为控制器声明了它需要 `$scope`（这里不需要给出 `$scope` 的初始化信息，无论是新创建的，还是重复利用的），所以 AngularJS 才会创建并注入它。这套依赖管理系统可以这样总结：“为了正常工作，我需要一个依赖（协作对象）：我不知道它从哪儿来，也不知道它如何创建。我只知道我需要它，所以请为我提供它。”

AngularJS 拥有内建的依赖注入（`dependency injection, DI`）引擎，职责如下：

- 理解对象对其协作对象的需求。
- 找到所需的协作对象。
- 连接协作对象，以形成功能完备的应用。

能够声明依赖关系非常有用:对象无须担心协作对象的生命周期。更有趣的是,可以随意交换协作对象,然后只要替换服务(service)就能创建不同的应用,这是能够进行有效单元测试的关键。

2. 依赖注入的好处

为认识到依赖注入的威力,我们列举一个负责推送并找回信息的通知服务。为了让情况复杂些,我们还要引入一个归档(archiving)服务。它们将通力合作,当通知数量超过限制时,让最旧的通知进入归档。有些麻烦的是,我们想要在不同的应用中使用不同的归档服务。有时,只需把旧信息打印在浏览器的控制台(console)上;另外一些时候,则需要借助XHR(XMLHttpRequest)调用将过期的通知送到服务器上。

通知服务的代码如下:

```
var NotificationsService = function () {
    this.MAX_LEN = 10;
    this.notificationsArchive = new NotificationsArchive();
    this.notifications = [];
};

NotificationsService.prototype.push = function (notification) {

    var newLen, notificationToArchive;

    newLen = this.notifications.unshift(notification);
    if (newLen > this.MAX_LEN) {
        notificationToArchive = this.notifications.pop();
        this.notificationsArchive.archive(notificationToArchive);
    }
};

NotificationsService.prototype.getCurrent = function () {
    return this.notifications;
};
```

因为使用 `new` 关键字初始化归档服务 (`NotificationsArchive`)，上面的代码与归档服务的某个特定实现紧密耦合了，这很不理想。实际上，两个类要遵守的合约，就只有 `archive` 方法（接受一条通知信息，并将其归档）。


能够交换协作者的机能，对可测试性 (`testability`) 来说极其重要。如果没有将真实实现替换为伪造实现 (`doubles/mocks`) 的能力，很难想象可以孤立地测试某对象。在本章接下来的内容中，我们将看到如何重构这些紧密耦合的对象，使之成为灵活且可测试的一组互相协作的服务。在此过程里，我们将领略 AngularJS 依赖注入的优势。

注册服务

AngularJS 只连接其认识的对象。因此，接入依赖注入机制的第一步，是将对象注册在模块 (`module`) 上。我们不直接注册对象的实例，而是将对象创建的方案抛给依赖注入系统，然后 AngularJS 解释这些方案以初始化对象，并在之后连接它们，最后成为可运行的应用。

AngularJS 的 `$provide` 服务可以注册不同的对象创建方案。之后 `$injector` 服务会解释这些方案，生成完备而可用的对象实例（已经解决好所有的依赖关系）。

`$injector` 服务创建的对象称为服务 (`service`)。在整个应用的生命中，每种方案 AngularJS 仅解释一次，也就是说，每个对象仅有一个实例。

 `$injector` 创建的对象都是单件 (`singleton`)，每个运行中的应用，只拥有一个给定服务的实例。

归根结底，AngularJS 模块保存对象实例，而我们控制这些对象的创建。

1. 值

让 AngularJS 管理对象最容易的方法是，注册已初始化好的对象，代码如下：

```
var myMod = angular.module('myMod', []);
myMod.value('notificationsArchive', new NotificationsArchive());
```

AngularJS 依赖注入机制管理的服务都需要独立的名字（如上例中的 `notificationsArhive`），然后就是创建新实例的方案。

值（**value**）对象不太有趣，通过以上方法注册的对象不能依赖于其他对象。这对于 `NotificationsArchive` 实例来说不成问题，它没有任何依赖。但实际中，这种注册方法只对非常简单的对象有效（内建（**built-in**）对象或对象字面量（**literal**））。

2. 服务

`NotificationsService` 依赖于归档服务，所以它不能注册为值对象。我们可以使用 `service` 方法注册构造函数，代码如下：

```
myMod.service('notificationsService', NotificationsService);
```

`NotificationsService` 构造函数实现如下：

```
var NotificationsService = function (notificationsArchive) {  
  
    this.notificationsArchive = notificationsArchive;  
  
};
```

通过依赖注入，在 `NotificationsService` 构造函数中可以消除 `new` 关键词。此服务已经不依赖于其他对象的初始化，可以接受任何归档服务，这让应用变得非常灵活！



服务（**service**）是蕴涵着很多意义的词汇之一。在 AngularJS 中，服务可能是注册构造函数的方法（如上例），也可能是任何依赖注入系统创建和管理的单件（**singleton**）对象，不管它们的注册方法为何（这也是大部分人在 AngularJS 模块的上下文中提到服务时所想表达的意思）。

在实际中，`service` 方法并不经常用到，但它在注册已存在的构造函数时，也许能带来方便，这样 AngularJS 就能够管理那些构造函数创建的对象了。

3. Factory

另一条注册对象创建方案的途径是使用 `factory` 方法。任何能够创建对象的函数都可被注册，因此它相比 `service` 而言更加灵活，示例代码如下：

```
myMod.factory('notificationsService', function(notificationsArchive) {

  var MAX_LEN = 10;
  var notifications = [];

  return {
    push: function (notification) {
      var notificationToArchive;
      var newLen = notifications.unshift(notification);

      //push方法现在可以依靠闭包(closure)作用域了
      if (newLen > MAX_LEN) {
        notificationToArchive = this.notifications.pop();
        notificationsArchive.archive(notificationToArchive);
      }
    },
    // NotificationsService的其他方法
  };
});
```

`factory` 方法返回的对象可以是任何有效的 JavaScript 对象，当然也包括 `function` 对象。

`factory` 是让对象加入依赖注入系统最常用的方法，它十分灵活，又能实现复杂的对象创建逻辑。它是普通的函数，所以我们可以利用词法作用域（lexical scope）来模拟“私有（private）”变量，这对隐藏服务的实现细节很有帮助。在上面的例子里，服务 `notificationToArchive` 配置参数 `MAX_LEN` 和内部状态 `notifications` 都是“私有”的^[2]。

4. 常量

`NotificationsService` 越来越好，它已经和协作对象解耦，并隐藏了私有状态。不幸的是，还有常量 `MAX_LEN` 这个写死的配置。`AngularJS` 考虑到了这点，常量可以注册在模块的级别上，且可以像其他协作对象一样被注入。

[2] 译者注：`factory` 返回的是闭包，熟悉函数式编程的朋友会明了其优势所在。

理想状态下，NotificationsService 应该这样获取配置信息：

```
myMod.factory('notificationsService',

function (notificationsArchive, MAX_LEN) {
    ...
    //创建逻辑没有变化
});
```

在 NotificationsService 外，配置信息被注册在模块上，代码如下：

```
myMod.constant('MAX_LEN', 10);
```

当创建不同应用使用的共通服务（服务的客户希望定制它）时，常量十分有用。但要注意，注册常量唯一的缺点是，当服务声明对某常量依赖时，必须提供那个常量的值。有时比较好的实践是有默认的配置值，然后让客户按需变化。

5. Provider

以上描述过的所有注册方法，都是最通用方法 `provider` 的特殊类型。下面是用 `provider` 注册 `notificationsService` 的例子：

```
myMod.provider('notificationsService', function () {

    var config = {
        maxLen : 10
    };
    var notifications = [];

    return {
        setMaxLen : function(maxLen) {
            config.maxLen = maxLen || config.maxLen;
        },

        $get : function(notificationsArchive) {
            return {
                push:function (notification) {
                    ...
                    if (newLen >config.maxLen) {
                        ...
                    }
                }
            }
        }
    };
});
```

```

        },
        //其他方法在这里
    };
};
});
});

```

首先, `provider` 是一个函数, 它返回包含 `$get` 属性的对象。该属性是一个工厂函数, 它被调用时会返回 `service` 的实例。可以认为, `provider` 就是 `$get` 属性为工厂方法的对象。

其次, `provider` 函数返回的对象可以有其他方法和属性, 在 `$get` 方法被调用前, 还能设置配置信息。实际上, 我们能够设置 `maxLen`, 但这不是必须的。甚至可以实现更复杂的配置逻辑, 比如采用配置方法 `setMaxLen`。

模块的生命周期

如前所述, **AngularJS** 支持多种对象创建方案, `provider` 是其中的通用方法, 它在创建对象实例前可以对其进行配置。为了支持 `provider`, **AngularJS** 将模块的生命周期分为两个阶段, 如下。

- 配置阶段: 收集对象创建方案, 并进行配置。
- 运行阶段: 执行所有初始化后的逻辑。

1. 配置阶段

配置 **Provider** 只能在配置阶段。确实, 在对象创建好之后, 再去修改创建方案也没什么意义。**Provider** 的配置代码如下:

```

myMod.config(function(notificationsServiceProvider) {
    notificationsServiceProvider.setMaxLen(5);
});

```

这里重要的是, 对 `notificationsServiceProvider` 对象的依赖, 它有 `provider` 的后缀, 表明它是即将执行的对象创建方案, 配置阶段允许我们对它进行最后的调整。

2. 运行阶段


运行阶段在应用启动时安排后面要执行的工作。你可能会将运行阶段联想为其他编程语言中的 `main` 方法。但是，最大的不同是，**AngularJS** 模块可能有多个配置和运行块（`block`）。在此意义上，应用没有单一的入口（运行时的应用其实就是一组在协作的对象）。

为了说明运行阶段的作用，我们假设要显示应用的已运行时间给用户。为了实现目标，我们将应用的已运行时间设定为 `$rootScope` 实例的属性，代码如下：

```
angular.module('upTimeApp', []).run(function($rootScope){
    $rootScope.appStarted = new Date();
});
```

然后在模板中取回它，代码如下：

```
Application started at: {{appStarted}}
```



在上面的例子中，我们直接为 `$rootScope` 设定了属性。不要忘记，`$rootScope` 实例是全局变量，因此它也有全局状态的种种弊端。`$rootScope` 实例应该只定义需要被很多模板访问的属性，且尽量少。

3. 不同的阶段与不同的注册方法

让我们总结创建对象的不同方法，以及这些方法对应的模块生命阶段，如表 1-1 所示。

表1-1

	对象种类	可以在配置阶段注入	可以在运行阶段注入
Constant	常量值	Yes	Yes
Variable	变量值	—	Yes
Service	构造函数创建的新对象	—	Yes
Factory	工厂函数返回的新对象	—	Yes
Provider	<code>\$get</code> 工厂函数创建的新对象	Yes	—

模块依赖

AngularJS 不仅在管理对象依赖上表现出众，还考虑了模块间的依赖。我们很容易就能将相关服务组织进一个模块，进而创建（可重用的）服务库。

例如，通知服务和归档服务，可以迁移到它们自己的模块中去（分别命名为 `notifications` 和 `archive`），代码如下：

```
angular.module('application', ['notifications', 'archive'])
```

这样，每个服务（或者一组相关服务）都被组织进可重用的模块，继而最顶层（应用级）模块声明对应用所需全部模块的依赖。

不只是最顶层模块依赖其他模块，其实每个模块都能表述其对子模块的依赖。这样，模块也形成了层级结构。于是，在 AngularJS 的模块中，存在两组不同而又相关的层级关系：模块层级和服务层级（服务也存在对其他服务、值和常量的依赖）。

AngularJS 的模块互相依赖，每个模块包含一些服务，服务之间也相互依赖，这产生了几十个有趣的问题。

- 定义在一个模块中的服务可以依赖于其他模块中的服务吗？
- 定义在子模块中的服务可以依赖于父模块中的服务吗？还是只能依赖于其他子模块的服务？
- 存在只有特定模块可见的私有服务吗？
- 在不同的模块中可以定义同名服务吗？

1. 服务的跨模块可见性

定义在子模块中的服务，如期望那样，可以注入父模块的服务，代码如下：

```
angular.module('app', ['engines'])

.factory('car', function ($log, dieselEngine) {
  return {
    start: function() {
      $log.info('Starting' + dieselEngine.type);
    };
  };
});
```

```

    }
  });

  angular.module('engines', [])
    .factory('dieselEngine', function () {
      return {
        type: 'diesel'
      };
    });

```

car 服务在 app 模块中定义，app 模块依赖于 engines 模块，后者定义了 dieselEngine 服务，因此 car 可以注入 dieselEngine。

令人惊讶的是，兄弟模块的服务也互相可见。我们将 car 服务迁移到独立模块中，然后修改模块依赖关系，让 app 同时依赖于 engines 和 cars 两个模块。

```

angular.module('app', ['engines', 'cars'])

angular.module('cars', [])
  .factory('car', function ($log, dieselEngine) {
    return {
      start: function() {
        $log.info('Starting ', + dieselEngine.type);
      }
    };
  });

angular.module('engines', [])
  .factory('dieselEngine', function () {
    return {
      type: 'diesel'
    };
  });

```

在上面的例子里，car 也注入了 dieselEngine。



定义在某模块中的服务，对其他模块可见。换句话说，模块的层级不影响服务对其他模块的透明度。AngularJS在引导应用启动时，它在不同模块中定义的所有服务都被导入应用，相当于全局命名空间。


AngularJS 将所有模块中的所有服务混入了应用级别的单一命名空间,在这里,对应每个名字只存在唯一的服务。我们可以利用这点,在依赖某模块的同时去覆盖 (override) 此模块提供的服务,比如,在 cars 模块中重定义 dieselEngine 服务,代码如下:

```
angular.module('app', ['engines', 'cars'])
  .controller('AppCtrl', function ($scope, car) {
    car.start();
  });


angular.module('cars', [])
  .factory('car', function ($log, dieselEngine) {
    return {
      start: function() {
        $log.info('Starting' + dieselEngine.type);
      }
    }
  })

  .factory('dieselEngine', function () {
    return {
      type: 'custom diesel'
    };
  });
```

car 服务被定义于同一模块的 dieselEngine 服务注入了,也就是说,car 模块的 dieselEngine 覆盖了 engines 模块的 dieselEngine。

 AngularJS 应用中的服务是不能重名的,父模块中的服务会覆盖子模块中的同名服务。

在当前版本的 AngularJS 中,模块中的服务对其他所有模块都可见,尚没有方法限制服务仅在指定模块内可见。

 至本书写作时为止,AngularJS 还不支持模块内的私有服务。

2. 为什么要用模块

AngularJS 将来自全部模块的所有服务都混入单一的应用级命名空间，这让人颇为惊讶，你可能会因此质疑使用模块的理由。毕竟，所有的服务都被装进了同一个大背包里，那为什么还要费劲地将它们再分为模块呢？

AngularJS 模块有助于在应用中组织多个 JavaScript 文件。将应用分解成模块有很多方法，第 2 章（构建与测试）中的很大一部分将会讨论这些策略及其优缺点。而且，短小、功能专一的模块在测试时会加载一组方便识别的服务，这也有助于降低单元测试的难度。同样，在第 2 章（构建与测试）中有更多的细节。

1.3 AngularJS 和其他框架 AngularJS and the rest of the world

为下一个项目选择完美的框架并不容易。有些框架更适合特定类型的应用，而团队经验、个人偏好及其他很多因素也会左右最后的选择。

不可避免地，AngularJS 会被拿来与其他流行的 JavaScript MV* 框架进行比较。不同的比较可能会得出不同的结果，不同的视点也会引起很多激烈的争论。因此，与其严密地一项项比较功能，我们更愿意提炼 AngularJS 与其他框架的不同之处。



如果你想看看用 AngularJS 编写的代码与用其他框架编写的代码的差异，就查看 TodoMVC (<http://addyosmani.github.com/todomvc>) 吧，此项目运用不同的 JavaScript MV* 框架实现同样的示例应用（TODO 列表）。这是个独一无二的机会，可以比较不同框架的架构实现、语法、大小及可读性。

很多特性让 AngularJS 鹤立鸡群，我们已经看到它实现 UI 模板的方法相当新颖，要点回顾如下。

- 自动刷新与双向数据绑定，让开发者从显式触发 UI 重绘的枯燥工作中解放出来。
- 以 HTML 语法为基础的动态 DOM，充当模板语言。更重要的是，还可以扩展现存的 HTML 词汇库（通过创建新的指令），继而用新的基于 HTML 的 DSL 构建 UI。

- 声明式UI，十分精炼并富有表达力。
- UI模板化机制没有给JavaScript代码添加任何限制（例如，模型和控制器也可以使用普通的、旧的JavaScript创建，而无须调用任何AngularJS API）。

AngularJS 突破性地为 JavaScript 世界带来了可靠的测试实践。框架自身被彻底测试过（言行一致），但是，对可测试性的追求不只如此，框架及其周围的生态圈都是抱持可测试的思想去构建的。

- 依赖注入引擎使可测试性成为可能，因为可以用短小的、仔细测试过的服务组成整个应用。
- AngularJS文档中的大部分代码示例拥有对应测试，这是AngularJS编写的代码具备良好可测试性的最好证据！
- AngularJS团队创建了卓越的JavaScript测试运行器Testacular (spectacular test runner, 优秀的测试运行器)，它将总体测试流程转变成了愉快的体验，有效、快速和可靠。测试并不容易，所以需要能够帮得上忙，而不是阻碍我们的工具。

最重要的是，AngularJS 让 web 开发再次变得有趣！它接管了如此之多的底层细节，以至于应用代码变得极度精炼。经常能听到，用 AngularJS 重写应用后，总代码量缩减到了原来的五分之一或更少。当然，具体情况要视项目及团队而定，但 AngularJS 让我们前进得更快，仅用一眨眼的工夫，就能收获成果。

jQuery 与 AngularJS

jQuery and AngularJS

AngularJS 和 jQuery 的关系，值得特别提及。首先，AngularJS 内嵌（作为源码的一部分）jQuery 的一个简化版本，jqLite。jqLite 是 jQuery 功能的微小子集，只专注于操纵 DOM 的例程（routine）。

[



内嵌jqLite后，AngularJS不用再依赖其他外部库。

]

但是，AngularJS 是 JavaScript 社区的良好“公民”，它可以与 jQuery 携手“共事”。一旦探测到 jQuery，AngularJS 就会用 jQuery 的 DOM 操作功能取代 jqLite。



要想在 AngularJS 项目中使用 jQuery，就得在 AngularJS 脚本前包含它。

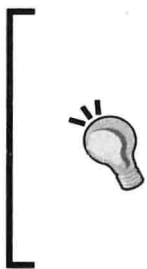
如果要尝试使用 jQuery UI 套件中的 UI 组件，事情就会变得更复杂。有些组件虽能正常工作，但大多数时候会有些小摩擦。这是因为两个库的指导哲学完全不同，以至于我们不能期待无缝结合。第 8 章（创建自定义指令）详述了如何集成与创建能正常工作的 UI 组件。

苹果与橙子

jQuery 与 AngularJS 虽然可以合作，但对它们进行比较是一件棘手的事。首先，jQuery 的诞生是为了简化 DOM 操作，它专注于文档遍历、事件处理、动画及 Ajax 交互。

而 AngularJS 是完整的框架，用于试图处理现代 web 2.0 应用开发的每一方面。

最重要的是，AngularJS 采用了完全不同的方法来构建 UI，由模型的变化来驱动模板更新（声明式）。使用 jQuery，经常要编写以 DOM 操作为中心的代码，当项目逐步成熟时（更大、互动也变得复杂）就显得难以管理。



AngularJS 围绕模型，jQuery 围绕 DOM，它们是彻底不同的开发范型。老练的 jQuery 开发者在转向 AngularJS 时，可能会陷入 jQuery 思考模式的误区。这不但会抑制 AngularJS 的力量，还会导致编写与其冲突的代码。这就是在 AngularJS 的学习中我们不推荐依赖 jQuery 的原因（尽量不被以前的习惯诱惑用 AngularJS 的方式去解决问题）。

AngularJS 为了现代 web 应用的开发而诞生，它尽力让浏览器成为更理想的开发平台。

窥视未来

A sneak peek into the future

AngularJS 在 web 开发的很多方面有所创新，它也许会影响我们在未来的浏览器中编写代码的方式。至本书写作时为止，有两套与 AngularJS 类似的规范正在制定当中。

`Object.observe` (<http://wiki.ecmascript.org/doku.php?id=harmony:observe>) 规范力图为浏览器内建追踪 JavaScript 对象变化的能力。AngularJS 比较对象的状态 (dirty checking) 来触发 UI 重绘。如果浏览器原生就有探测对象变化的机制，则会显著提升包括 AngularJS 在内的许多 JavaScript MVC 框架的性能。事实上，AngularJS 团队已经就此规范进行了一些实验，并观察到了 20%~30% 的性能提升。

web 组件规范 (<http://w3c.github.io/webcomponents/explainer/>) 试图定义一批具有丰富视觉效果的组件 (它们不可能用 CSS 单独实现)，且容易组合与重用 (现有的脚本库还不能支持)。

这并不是一个容易实现的目标，但 AngularJS 的指令已经揭示出，定义封装良好的可重用组件还是有可能的。

AngularJS 不只是创新型框架 (以今天的标准来看)，它还将影响未来的 web 开发生态。AngularJS 团队与上述规范的作者密切合作，所以很有可能，AngularJS 催生的许多创意会进入未来浏览器的内核。我们可以期待，今天在学习和使用 AngularJS 上的投资，在未来能够得到回报。

1.4 总结

Summary

本章介绍了很多内容。首先，我们从熟悉 AngularJS 项目及其团队开始，学习了如何寻找项目的源代码和文档，编写第一个应用 “Hello World”。令人惊喜的是，AngularJS 非常轻量化，也很容易入门。

其次，本章的大部分内容讲解如何使 AngularJS 的控制器、作用域和视图协同工作，这为之后的学习打下了坚实的基础。也有一大部分内容介绍了在模块内创建服务，以及运用依赖注入连接这些服务的方法。

最后，我们将 AngularJS 与其他 JavaScript 框架进行了比较，阐述了它的独特之处。我们希望说服你，花时间学习 AngularJS 是值得的。

视图、控制器和服务是任何 AngularJS 应用的基本，所以，进一步理解这些主题是很有必要的。幸运的是，我们已经学会如何创建服务层和视图层，为构建真实项目做好了准备。第 2 章将为一个复杂应用架设基本结构，从代码组织开始进入构建与测试的主题。

第 2 章

构建与测试

Building and Testing

第 1 章介绍了 AngularJS 框架，涵盖了所属项目、维护者及基本使用场景等内容。现在，我们已经准备好构建完整精致的 web 应用。本书的余下部分将围绕一个示例应用展开，以讲解如何在真实项目中使用 AngularJS。

从本章开始，我们将构建一款支持 SCRUM 敏捷软件开发方法的简单项目管理工具。此示例应用将展现 AngularJS 的 API 与行话术语，以及后端通信、组织导航、安全、国际化等典型应用情景。本章还将介绍此示例应用所在的问题领域及其使用的技术栈。

在启动每个项目时，总会就文件组织策略、构建系统、基本工作流这几方面做些起始决策。示例应用也一样，本章也会讨论构建系统与项目布局相关的主题。

AngularJS 及其生态圈都提倡自动化测试这一可靠的工程实践。我们坚定地相信，除非是那些最小型的项目，任何项目都必须进行自动化测试。因此，本章的最后部分将完全贡献给测试：不同的测试类型、机制、工作流，最佳实践及如何使用工具。

本章中，我们将了解以下内容。

- 贯穿本书的示例应用，它所在的问题领域及使用的技术栈。
- 为 AngularJS web 应用推荐构建系统、关联工具与工作流。
- 组织文件、目录与模块的建议。
- 自动化测试实践、不同类型的测试及其在项目中的地位。你还将熟悉 AngularJS web 应用常选用的测试库和工具。

2.1 介绍示例应用

Introducing the sample application

本节将给出示例应用的更多细节，它将作为我们学习的素材贯穿本书。



本应用的源代码公开在Github上: <https://github.com/angular-app/angular-app>。这个资源库包含完整的源代码、详细的安装指导及整个项目历史。

熟悉问题领域

Getting familiar with the problem domain

为了展示 AngularJS 最有优势的适用环境，我们将构建一款项目管理工具，它为遵循 SCRUM 方法的团队而设计。



AngularJS是一个在构造CRUD类的应用时表现得非常出色的框架，此类应用通常由很多动态表单、列表和表格的页面组成。

很多读者也许已经对 SCRUM 有了一定的认识，它是运转项目的流行的敏捷方法。关于 SCRUM，不熟悉的人也无须担忧，因为它的内容很容易掌握。有很多精彩的书籍与文章，可以让你深入了解 SCRUM。但是，也许只需要看看维基百科上的相关介绍就能有基本的理解，[http://en.wikipedia.org/wiki/Scrum_\(development\)](http://en.wikipedia.org/wiki/Scrum_(development))。

示例应用的目标是辅助团队管理 SCRUM 产物：项目及其工作列表、冲刺及其工作列表、任务、进度图，等等。该示例应用还具有完整的管理模块，用于管理用户和他们所在的项目。



示例应用不会去努力遵循所有完整的和严格的SCRUM原则。一旦需要，我们就采取一些捷径，以便更清晰地描绘AngularJS的使用方式，而不是花费力气去构建真正的SCRUM项目管理工具。

完成该示例应用时应如图 2-1 所示。

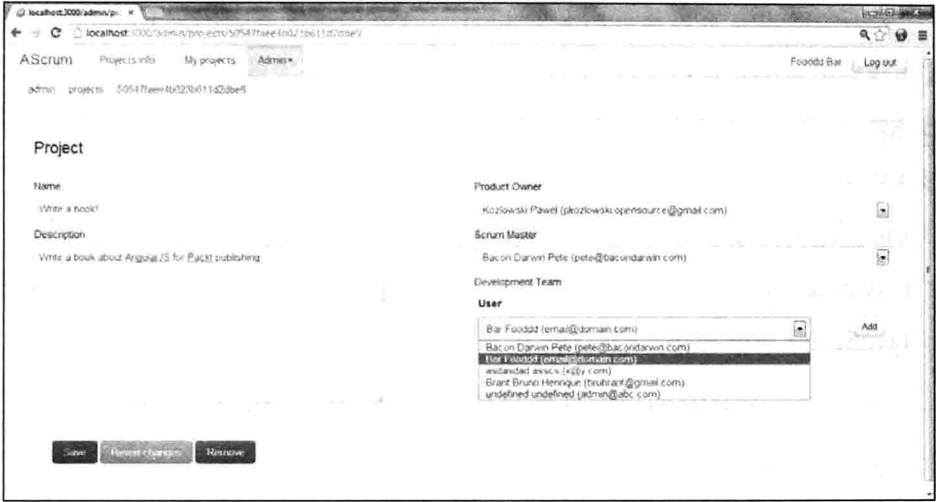


图2-1

通过图 2-1，我们能立刻意识到这是个相当典型的 **CRUD web 应用**。

- 获取、显示和编辑持久化存储（**persistent store**）中的数据。
- 鉴权（**authentication**）和授权（**authorization**）。
- 复杂的导航模式及所支持的 UI 元素，如菜单、面包屑（**breadcrumbs**）等。

技术栈

Technical stack

本书专注于讲解 **AngularJS**，但对于构建任何稍具规模的 **web 应用**，我们需要的不仅仅是一个运行在浏览器中的 **JavaScript** 库。至少，还需要持久化存储和后端（**back-end**）。本书不会指定任何技术栈，我们相信各位会使用各种不同的后端、服务器端框架和持久化存储与 **AngularJS** 协作。尽管如此，我们仍需做出选择，以描绘 **AngularJS** 如何适用于大场景。

首先，也是最重要的，我们需要尽可能地使用对 **JavaScript** 友好的技术。虽然有很多出色的后端、平台、持久化存储可供选择，但是，我更愿意使用 **JavaScript** 开发者熟悉的技术。而且，我们也会使用在 **JavaScript** 生态系统中被视为主流的技术。



本书中描述的所有技术栈元素和工具皆基于JavaScript，属于可免费使用的开源项目。

持久化存储

Persistence store

数据存储有很多可选项，最近的 NoSQL 运动更是为我们增加了不少替代方案。为了本书的讲解目的，我们会使用基于文档的 MongoDB 数据库，因为它更适合基于 JavaScript 的环境。

- 文档使用JSON风格的数据格式存储（二进制JSON，简称BSON）。
- 可以用JavaScript及熟悉的JSON语法查询和操作数据。
- 通过REST端点提供和使用JSON格式的数据。

理解本书中的示例，并不需要 MongoDB 的相关知识，而且，我们也会在讨论代码片段时解释那些诡秘的部分。这并不是说 MongoDB 或其他基于文档的 NoSQL 数据库，对于 AngularJS 应用是更好的选择。AngularJS 并不关心后端和存储方案。

MongoLab

MongoDB 相当容易入门，很多 JavaScript 开发者在这个基于文档的数据库上工作时，都会感到亲切。为了让学习更简单，示例应用将依赖 MongoDB 的线上寄存版本（hosted version）。所以，运行本书中的示例，不需要安装任何软件。

虽然有不少支持 MongoDB 的云端寄存服务提供商，但我们发现，MongoLab (<https://mongolab.com>) 可靠，容易入门，并且为数据量在 0.5 GB 以下的数据库提供免费寄存。这个空间对本书示例应用来说，已经足够用了。

MongoLab 有一个不容忽视的优点，即它通过设计良好的 REST 接口暴露所寄存的数据库。我们可通过此接口来呈现 AngularJS 如何与 REST 端点进行交流。

使用 MongoLab 需要线上注册，这很轻松，只需要填写一张简短的电子表单。搞定之后，就能立刻开始访问寄存的 MongoDB 数据库、输出 JSON 数据的 REST 接口，以及在线管理控制台。

服务器端环境

AngularJS 应用通过 REST 接口，可以直接访问寄存在 MongoDB 上的数据库。对于非常简单的项目来说，浏览器直接和 MongoDB 通信可能是比较好的方案。但是，现实中的公开项目则不能这样做，MongoLab 并没有内建足够的安全机制。

在实践中，AngularJS 所编写的大部分 UI 组件将与某类后端通信以获取数据。中间件往往提供安全服务（认证）及查证访问许可（授权）。示例应用也需要后端支持。所以，我们选择了基于 JavaScript 的解决方案：node.js。

node.js 在所有流行的操作系统下都有对应的二进制包，可以从 <http://nodejs.org> 获取。需要下载和安装 node.js 运行时环境，以在本地机器上运行本书示例。



介绍node.js超出了本书范围。幸运的是，运行本书示例，只需要基本的关于node.js及其包管理器（node.js package manager - npm）的相关知识。熟悉node.js的开发者会很容易理解在示例应用的服务器端发生的事情，但想理解本书中的AngularJS示例，熟悉node.js的知识并非必要条件。

除了 node.js 自身，我们将使用 node.js 库来构建示例应用的服务器端组件。

- **Express** (<http://expressjs.com>)，服务器端web应用框架，提供路由、静态资源及数据服务。
- **Passport** (<http://passportjs.org>)，node.js的安全中间件。
- **Restler** (<https://github.com/danwrong/restler>)，node.js的HTTP客户端库。

对于有效学习 AngularJS 来说，熟悉 node.js 及上述库也许会有帮助，但并不是必要的。一方面，你也许想深入理解本书中的服务器端例子，因而去研究探索 node.js 及上述库。另一方面，如果你的工作需要使用不同的后端，也可以放心忽略大部分 node.js 的相关细节。

第三方JavaScript库

AngularJS 能够单独编写相当复杂的应用程序，但它并不打算重新实现流行的库。

示例应用试图将对外部第三方库的依赖减至最小，以便我们专注于 AngularJS 的使用。尽管如此，很多项目会用到其他有名的库，如 jQuery、underscore.js 等。为了呈现 AngularJS 如何与其他类库协同工作，我们将在项目中引入最新的 jQuery 相容版本。

Bootstrap CSS

AngularJS 不强迫或不指定任何特定的 CSS 用法，所以任何人都可以自由地按需调整他们应用的样式。为了让 SCRUM 应用看起来更好，我们会使用流行的 Twitter Bootstrap (<http://twitter.github.com/bootstrap/>) CSS。

示例应用会包括 Bootstrap 的 LESS 模板，以描绘构建链 (build chain) 如何包含进 LESS 的编译。

2.2 构建系统 Build system

曾几何时，JavaScript 一度被视为玩具语言，但这已过去很久了。最近几年，JavaScript 已经成为主流语言，每天都有复杂的大型应用程序被编写出来，部署上线。复杂度的提升意味着需要编写更多的代码，这段时间我们经常会看到有上百万行 JavaScript 代码的项目。

在 HTML 文档中包含单一 JavaScript 文件已经不再实际，我们需要构建系统。在被部署到生产服务器之前，JavaScript 和 CSS 文件会经受许多检查和转换，这类转换的例子如下：

- JavaScript 源代码必须运用 `jslint` (<http://www.jshint.com>)、`jshint` (<http://www.jshint.com>) 等工具检查是否符合编码标准。
- 测试套件应尽可能频繁地执行，至少必须成为每一次构建的组成部分。因此，测试工具和测试过程需要紧密集成进构建系统。
- 也许需要生成某些文件（比如从 LESS 模板生成的 CSS 文件）。
- 文件需要被合并 (merged) 在一起，并最小化 (minified) 以优化在浏览器中下载及运行的性能。

除了上述列出的步骤，通常在应用被部署到生产服务器之前，还有其他任务必须执行，如文件要拷贝到最终目的地、文档更新等。在稍具规模的应用中，常有许多费力的任务，我们应尽量尝试将之自动化。

构建系统准则

Build system principles

开发者热爱编写代码，不过在现实中，我们只会在文本编辑器前花费一部分时间。除去那些有趣的部分（设计、和同伴交流、编程及修复 bug），还有大量让人疲惫的任务需要频繁执行——有时相当频繁。构建应用就属于此类任务，它必须一次次地反复运行，所以我们应让这个过程尽量快而且尽量轻松，下面将介绍设计构建系统的一些准则。

自动化所有事情

本书作者很不擅长一遍遍地手动执行重复性的任务。我们迟缓，会犯错误，而且，很容易被日常构建任务搞得心情烦躁。如果你也像我一样，就很容易理解为什么我们坚信应尽可能将构建过程的每一步自动化。

计算机很擅长做重复性的工作，它们从不犯错、不需要休息、不会分心、也不会和被指派执行重复任务的时候抱怨。投资在自动化构建过程中会很快获得回报，因为我们将在今后的每一天都省下大量时间！

尽早报错，清晰报错

典型的构建过程由不同的步骤组成。其中一些步骤大部分情况下会执行正常，另一些步骤则不时失败。那些可能失败的任务，应当在构建过程的早期被执行，当探测出异常情况时，应尽早停止构建。这保证了更花时间的步骤在基本步骤执行正常前不会被执行。

在付出代价后，才认识到这条规则有多重要。开始时，示例应用的构建系统在 jshint 之前执行自动化测试，等待所有自动化测试执行之后，发现无效的 JavaScript 中断了构建过程。解决之道就是将 jshint 任务移动到构建管道的起始段上，这样才能尽可能早地中断构建。

当构建中断时，打印出清楚的错误信息非常重要。一旦有了清楚的错误信息，就能够在几秒内确定构建失败的根源。没有什么事情比在项目部署上线时盯着暗号般的错误信息猜测真相更糟糕的了。

不同的工作流，不同的命令

作为开发者，我们被赋予不同的任务。某天我们也许忙于增加新代码（和测试一起），然后下一天又会花时间集成新开发的特性。为了应对这样的现实情况，构建系统应该为不同类型的工作流提供不同的命令。根据我们的经验，构建系统至少应有如下三项不同的构建任务。

- 快速运行与检测代码正确性的命令：在JavaScript项目中，这也许意味着运行 `jslinc/jshint` 并执行单元测试（**unit tests**）。因为会被频繁地执行，所以需要此命令飞速运行。这种构建任务非常有利于进行测试驱动开发（**test driven development, TDD**）。
- 为测试而部署完整应用的命令：在执行完此构建任务后，我们应当在浏览器中运行应用。为了完成此目标，这里需要进行更多处理，如生成CSS文件等。这种构建任务专注于UI相关开发的工作流。
- 向生产环境部署的构建任务应该运行上述提到的所有检测，并为最终部署做准备，如合并与最小化文件、运行集成测试等。

构建脚本同样是代码

构建脚本是项目产物的一部分，我们应该同其他产物一样平等对待它们。构建脚本也需要被阅读、理解和维护，就像任何其他源代码一样。编写糟糕的构建脚本会显著拖慢进展，而且会产生让人沮丧的调试争论。

工具 Tools

前面勾勒出的构建系统设计准则，可以应用在任何项目和任何工具集上。我们选择的工具链是与操作系统（OS）无关的，所以你可以在任何流行的操作系统上运行示例应用。



在为示例项目选择最好的工具之后，我们意识到不同的项目会使用不同的工具。尽管如此，本书其他部分和本章中推荐的工具依然重要，它们应该很容易适配其他构建系统。



Grunt.js

示例 **SCRUM** 应用的构建系统使用 **Grunt** (<http://gruntjs.com>)。grunt.js 的宣传语如下：

“基于任务的命令行构建工具，为 JavaScript 项目服务。”

对我们来说，重要的是 grunt.js 构建脚本采用 JavaScript 语言编写，并在 node.js 平台上运行。这是非常好的消息，意味着我们能用同样的平台与同样的编程语言来构建和运行示例应用。

Grunt 属于基于任务的构建工具这一分类，比如 Gradle (Groovy 编写的脚本)，或者 Rake (Ruby 编写的脚本)，所以熟悉那些工具的人应该感到很亲切。

测试库与工具

AngularJS 使用、提倡和鼓励自动化测试实践。AngularJS 团队非常严肃地看待可测试性，他们确保采用 AngularJS 编写的代码容易测试。不仅如此，AngularJS 团队还编写代码或扩展工具，以让测试在实践中变得更轻松。

Jasmine

AngularJS 的测试代码是用 Jasmine (<http://pivotal.github.com/jasmine/>) 编写的。Jasmine 是用于测试 JavaScript 代码的框架，源于行为驱动测试 (behavior driven development, BDD) 运动。BDD 影响了 Jasmine 的语法。

所有 AngularJS 原生文档中的例子都使用 Jasmine 语法，所以该测试框架对示例应用而言是很自然的选择。更进一步，AngularJS 已经开发了各种 mock 对象以及 Jasmine 扩展，以提供集成更好、讲求实效的日常测试体验。

Karma runner

Karma runner (<http://karma-runner.github.io>) 用于执行 JavaScript 测试。作为纯 JavaScript, 基于 node.js 的解决方案, Karma runner 是为了取代另一个流行的测试运行器 JS TestDriver 而生的。

Karma runner 能够分发源代码和测试代码到浏览器的运行实例上 (或者在需要时开启新浏览器), 触发测试的执行, 收集各测试的输出结果, 并汇报最终结果。它使用真实浏览器执行测试, 这在 JavaScript 世界中很了不起, 因为我们能够同时在几个不同的浏览器下运行测试, 以保证代码能正确工作于任何地方。



Karma runner 在稳定性和速度上表现非凡, 在 AngularJS 项目中, 它作为持续集成构建的一部分来执行测试。每次构建中, Karma runner 在多个浏览器中执行约 2000 个单元的测试, 在 20 秒内, 它总共执行约 14000 个测试。这些数字让我们增加了对 Karma runner 作为工具、AngularJS 作为框架这一组合的信心。

测试在 AngularJS 中是个中心主题, 所以我们将在本章稍靠后的部分中深入探索 Jasmine 测试和 Karma runner。

2.3 组织文件和目录 Organizing files and folders

到目前为止, 关于构造示例 SCRUM 应用所需的技术栈和工具, 我们已经做了一些不错的决定。现在, 需要回答更严峻的问题——如何用有意义的目录结构组织不同的文件。

对于任何项目, 有多种组织文件的方法。有时我们没有选择, 因为所用的工具和框架已经预先指定了文件布局。但是, grunt.js 和 AngularJS 没有强迫我们接受任何特定的目录结构, 所以我们可以做出自己的选择。

根目录 Root folders

在设计目录布局时, 我们希望得出更容易导航代码库的目录结构, 同时又将构建的复杂度控制在合理的级别上。

引领我们走向高级目录结构设计的基本假设如下。

- 应用的源代码和伴随测试应该分清楚，以保证构建系统容易维护，因为源代码和测试通常有两套不同的构建任务。
- 任何外部库的第三方代码，应该与内部的代码库隔离。外部库与内部库的变化不在同一步调上，而我们想让外部库的随时升级变得更容易。将代码与外部库混合在一起，会让升级更困难，也更花时间。
- 构建的相关脚本应该安置在它们自己的专属目录中，而不是散布在整个代码库里。
- 构建结果应该被输出到单独目录下，构建输出的内容和结构应该匹配生产环境部署的需求——容易将构建输出的结果部署到最终目的地上。

综合以上假设，我们将在项目中给出如下顶级目录：

- src：包含应用源代码。
- test：包含相伴的自动化测试。
- vendor：包含第三方依赖库。
- build：包含构建脚本。
- dist：包含构建结果，准备部署到目标环境中去。

最终，视觉化后的顶级目录结构如图 2-2 所示。










Name	Type	Size
 build	File folder	
 dist	File folder	
 src	File folder	
 test	File folder	
 vendor	File folder	
 .gitignore	Text Document	1 KB
 grunt.js	JScript Script File	4 KB
 LICENSE	File	2 KB
 package.json	JSON File	1 KB

图2-2

除了以上描述过的目录,我们注意到还有些顶级文件,以下是这些文件的说明:

- `gitignore`: 与git版本控制系统相关,包括指明哪些文件不应进入git资源池的规则。
- `LICENSE`: MIT版权协议。
- `Gruntfile.js`: `grunt.js` 构建的起点。
- `package.json`: `node.js`应用的包描述。

进入源代码目录

Inside the source folder

有了前面的基础,现在准备好进入 `src` 目录结构了。首先让我们看看它视觉化后的样子,如图 2-3 所示。






Name	Type	Size
 <code>app</code>	File folder	
 <code>assets</code>	File folder	
 <code>common</code>	File folder	
 <code>less</code>	File folder	
 <code>index.html</code>	Chrome HTML Do...	1 KB

图2-3

`index.html` 文件是示例应用的起点,可以看到四个目录,其中,两个目录用于保存 AngularJS 特定的代码 (`app` 和 `common`),另外两个目录则包含与 AngularJS 无关的表现层产物 (`assets` 和 `less`)。

`assets` 和 `less` 目录并不神秘,前者保存图像和图标,后者保存 LESS 变量。请注意 Twitter Bootstrap CSS 所用的 LESS 模板是位于 `vendor` 目录中的。这里我们只保存变量的值。

AngularJS的特定文件

AngularJS 应用由两类文件组成,即脚本和 HTML 模板。任何有规模的项目都会生成这两类文件,所以我们需要了解组织这些文件的方法。理想情况下,我们把相关的文件组织在一起,而把不相关的文件隔离出来。可令人头痛的是,文件会以多种不同的形式彼此产生关联,而我们仅有一棵目录树来表现这些关联性。

通常解决此类问题的策略包括按照特性组织文件（**package by feature**）、按照架构层组织文件（**package by layer**）或依靠文件类型。这里将采用混合的方案。

- 大部分应用按照特性组织文件，功能相关的脚本和子模板（**partials**）在一起。这样的安排非常有利于在应用的垂直切面上工作，因为被修改的所有文件同属于一组。
- 封装基础组件（持久化存储访问、本地化、通用指令等）的文件应组织在一起，理由是这类基础脚本不像功能性代码那样频繁变化。在典型的应用生命周期中，一些基础技术组件很早就被写好，关注点很快会随着应用的成熟转移到功能性代码中去。因此，通用的、基础级别的文件最好按照架构层来组织。

对示例应用的目录结构采用上面推荐的方案，如图 2-4 所示。

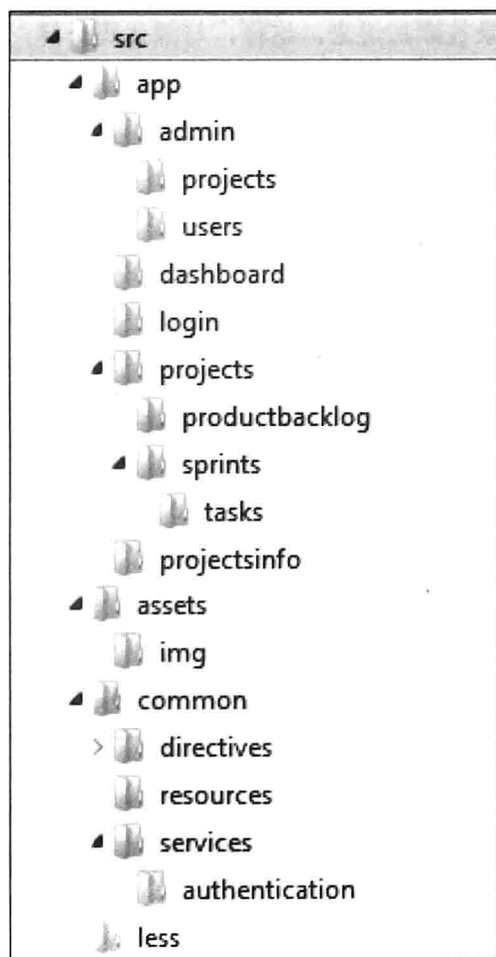


图2-4



这里描述的目录结构不同于AngularJS seed项目 (<https://github.com/angular/angular-seed>) 推荐的目录结构。后者的目录布局对于简单的项目来说很不错, 但AngularJS社区一致认同更大的项目按照特性来组织文件更好。

轻装上路

观察前面给出的目录结构, 我们注意到一些包含功能性代码的目录有很深的层级, 那些目录实际上紧密匹配应用自身的分级导航。这是很可取的方法, 因为只要看看应用的 UI, 就能很快了解相应的代码可能会存放在哪里。

好的办法是, 用非常简单的目录结构来启动项目, 然后逐步向最终的路径布局接近。例如, 示例应用一开始在 **admin** 下没有任何子目录, 而是将所有管理 **SCRUM** 项目和用户的功能都放在了同一路径下。随着代码库的不断演变, 文件会越来越多, 新的子目录被添加进来, 这样, 与源代码一样, 整个目录结构也会在几次迭代中被重构和优化。

1. 共同演变的控制器与子模板

常可以看到按照文件类型组织目录的项目, 在 AngularJS 上下文中, JavaScript 脚本和模板常分开存放在不同的目录结构中, 这种分离虽然听上去不错, 但实践中模板和对应的控制器常常同时演变。这就是为什么在示例 **SCRUM** 应用中, 模板和控制器被放在了一起。每个功能特性都有它自己的目录, 模板和控制器都安放在同一个目录里。

下面是用户管理功能相关目录中的内容, 如图 2-5 所示。


Name	Date modified	Type	Size
 admin-users.js	2012-11-20 17:59	JScript Script File	1 KB
 admin-users-edit.js	2012-11-27 21:25	JScript Script File	3 KB
 users-edit.tpl.html	2012-11-26 19:29	Chrome HTML Document	2 KB
 users-list.tpl.html	2012-11-20 17:59	Chrome HTML Document	1 KB

图2-5

深入测试目录

开展自动化测试是为了判断应用是否正常运行，因此，测试代码与功能代码紧密关联就不意外了，test 目录几乎完全模仿 src/app 目录，如图 2-6 所示。

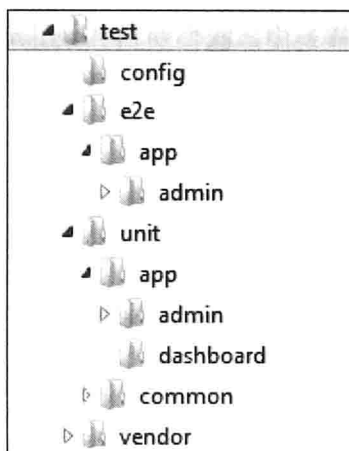


图2-6

从图 2-5 中可以看到，测试目录的内容几乎是对代码目录的镜像。所有的测试库位于专属的 vendor 目录中，Karma runner 的测试配置也有它自己的目录 (config)。

文件命名约定

File-naming conventions

约定文件命名，对于在代码库中轻松导航很重要。本书使用 AngularJS 社区遵循的一组约定，如下：

- 所有的JavaScript文件使用标准的.js扩展名。
- 子模板的后缀是.tpl.html，这样就能轻易地从其他HTML文件中分辨它们。
- 测试文件和它测试的目标文件命名一致，后缀依靠测试类型，如单元测试会以.spec.js结尾。

2.4 AngularJS 模块和文件

AngularJS modules and files

我们已经组织好应用的文件目录，现在可以看看单独文件的内容了。这里我

们将专注于 JavaScript 文件及其内容，以及与 AngularJS 模块（module）的关联。

一个文件，一个模块 One file, one module

在第 1 章（Angular 之禅）中，我们明白 AngularJS 的模块相互依赖，这意味着在 AngularJS 应用中需要同时处理目录层级和模块层级。现在将进一步挖掘这些主题，来获得关于如何组织 AngularJS 模块及其内容的注重实效的推荐方案。

基本上，我们能采取三种方法关联单个文件和 AngularJS 模块。

- 允许一个 JavaScript 文件中存在多个 AngularJS 模块。
- 让 AngularJS 模块跨越多个 JavaScript 文件。
- 在每个 JavaScript 文件中都定义且仅定义一个 AngularJS 模块。

在单个文件中定义多个模块，并非十分有害，但可能会导致单个文件中的代码行数过多。这让我们很难在代码库中发现所需的特定模块，因为不但要在文件系统中找到文件，还要在文件中找到模块。所以，在单个文件中定义多个模块，也许对非常简单的项目有效，但在较大的代码库中，表现并不乐观。

让一个模块跨越多个文件，或许是应该避免的。当单个模块的代码分布在多个文件中时，就要考虑那些文件的加载顺序：模块需要定义在注册 provider 之前。此外，这样的模块也会变得越来越大，越来越难以维护。在单元测试中，我们希望加载和测试尽量小的代码片段，因此，大模块也不利于单元测试。

在上述三种方法里，在每个文件中仅定义一个 AngularJS 模块是最明智的选择。



坚持一个文件等于一个 AngularJS 模块的原则，这样就可以维护相对小、目标单一的文件和模块。而且，无须关心文件的加载顺序，也可以在单元测试下加载不同的模块。

模块内部

Inside a module

模块一旦被定义，就可以用来注册对象创建方案。提醒一下，这些方案在 AngularJS 术语中称为 **providers**。每个 **provider** 在执行时，都会产生一个运行时服务实例(run-time service instance)。尽管服务创建方案可以有多种形式(factory、service、providers 和变量)，但最后的结果总是一样的——配置过的服务实例。

注册provider的不同语法

要注册新的 **provider**，首先要获得模块的实例。这并不困难，对 `angular.module` 方法的调用会返回它。

可以保存返回模块实例的引用，然后重用它注册多个 **provider**。比如，在模块上注册两个负责管理项目的控制器，可以这样写：

```
var adminProjects = angular.module('admin-projects', []);

adminProjects.controller('ProjectsListCtrl', function($scope) {
    //这里是控制器的代码
});

adminProjects.controller('ProjectsEditCtrl', function($scope) {
    //这里是控制器的代码
});
```

这个方案当然有效，但它有个缺点：为了能在一个模块上声明多个 **provider**，我们被迫声明了一个中间变量（这里是 `adminProjects`）。更糟糕的是，如果我们不特别小心的话（封装模块声明进入闭包、创建命名空间等），中间变量可能会进入全局命名空间。理想情况下，最好能够不依靠中间变量，而以某种方法取回已经声明过模块的实例，我们可以如下这样做：

```
angular.module('admin-projects', []);

angular.module('admin-projects').controller('ProjectsListCtrl',
function($scope) {
    //这里是控制器的代码
});
```

```
angular.module('admin-projects').controller('ProjectsEditCtrl',
function($scope) {
    //这里是控制器的代码
});
```

虽然成功地消除了额外的变量，但代码并没有变好。你注意到 `angular.module('admin-projects')` 在到处重复了吗？所有的代码重复都是可怕的，而且，如果某天想给模块换个名称，那么我们会深受打击。除此之外，声明新模块和取回已存在模块的语法会导致错误，得到非常让人困惑的结果。例如，比较 `angular.module('myModule')` 和 `angular.module('myModule', [])`，很容易忽视它们的差别，不是吗？



最好避免使用 `angular.module('myModule')` 构造函数取回 AngularJS 模块，语法冗长，而且会导致代码重复。更糟糕的是，定义模块会很容易和获取已存在模块的实例相混淆。

幸运的是，有更好的方案解决上述所有问题，先看下面代码：

```
angular.module('admin-projects', [])

.controller('ProjectsListCtrl', function($scope) {
    //这里是控制器的代码
})

.controller('ProjectsEditCtrl', function($scope) {
    //这里是控制器的代码
});
```

我们知道，可以连锁调用对控制器的注册逻辑。每次对 `controller` 的调用会返回模块的实例，用于继续调用方法。其他注册 **provider** 的方法（`factory`、`service`、`value` 等）也都会返回模块实例，便于采用同样的模式在其上注册多个不同的 **provider**。

在示例 **SCRUM** 应用中，我们将用上面描述的链式（**chaining**）语法注册所有的 **provider**。这种注册 **provider** 的方法消除了创建多余变量（潜在的全局）的风险，避免了代码重复。如果在代码格式上下点工夫，这种方式还非常具有可读性。

声明配置和运行块的语法

在第 1 章（Angular 之禅）中，启动 AngularJS 应用的过程被分成了两个不同的阶段，即配置阶段和运行阶段。每个模块可以有多个配置和运行块，这里并没有限制必须使用唯一的块。

注册将在配置阶段运行的函数，AngularJS 支持两种不同的方式。我们已经看到，可以指定一个配置函数，作为 `angular.module` 函数的第三个参数：

```
angular.module('admin-projects', [], function() {
    //这里是配置逻辑
});
```

这种方式允许我们注册一个且仅一个配置块，采用这种方式，模块声明会变得相当冗长（特别是模块对其他模块有依赖时）。另一种可替代的方式是使用 `angular.config` 方法：

```
angular.module('admin-projects', [])
    .config(function() {
        //配置块1
    })

    .config(function() {
        //配置块2
    });
```

如你所见，此替代方式可以注册多个配置块，这样有利于将大的配置函数（尤其是有很多依赖的）分割成更小、目标也更专一的函数。小而紧凑的函数更容易阅读、维护及测试。

SCRUM 示例应用将使用后者的形式来注册配置函数与运行函数，因为这样更有可读性。

下载示例代码



可以从<http://www.packtpub.com>下载所有从账号直接在线购买过的Packt图书的示例源代码，如果你在别的地方购买了本书，可以访问<http://www.packtpub.com/support>以注册并要求我们直接将源代码以电子邮件的形式发给你。

2.5 自动化测试

Automated testing

软件开发很难，作为开发者，我们需要在客户需求和最后期限的压力之间走钢丝，与其他团队成员协同工作，同时还要精通大量软件开发工具和编程语言。随着软件复杂度的提高，bug 的数量也不断增加。人类会犯很多错误，低级的编程错误只是其中的一种，安装、配置和集成问题也会污染任何有规模的项目。

项目里很容易出现不同类型的 bug，因此我们需要有效的工具和技术与之“战斗”。当前的软件开发者越来越认为，只有通过严苛的自动化测试，才能保证按时交付有质量的产品。

敏捷开发方法（极限编程，XP）首次使自动化测试实践流行开来，但是，几年前崭新的革命性方法（当然一定程度上有争议）现在已经成为被广泛接受的标准实践。近些年来，已经没有任何借口不进行广泛的自动化测试了。

单元测试是我们与程序 bug 战斗的第一道防线，这些测试就像其名字所暗示的那样，专注于小的代码单元，通常是单个类或密切协作的对象集群。单元测试是属于开发者的工具，可帮助我们确保获取到正确的底层结构，让函数输出预期的结果。除了单纯的一次性代码验证，单元测试还带来了许多其他好处。

- 可帮助我们在开发过程中及早发现问题，这时调试和修正错误的成本最小。
- 编写良好的测试套件可以在每次构建时执行，以提供持续保障。这让我们得以安心，充满自信地修改代码。
- 理论上来说，没有比文本编辑器加测试运行器更好和更轻量化的开发环境。无须构建、部署和在应用中点击，就能够迅速确定代码改动的正确性。

- 运用测试驱动开发（TDD）编写适合于测试的代码，能够帮助我们设计类及其接口。测试就像另一个调用代码的客户，可帮助我们设计出灵活、解耦的类与接口。
- 最后但也很重要，测试能够起到文档的作用。通常我们可以打开测试文件，查看如何使用一套可接受的参数来调用给定的方法。而且，这套文档是可执行且一直保持最新的。

单元测试有很大帮助，但它不能解决所有可能的问题，特别是它探测不到所有配置和集成问题。要想彻底让自动化测试覆盖应用，则需要执行高等级的端对端（集成）测试。集成测试的目的就是实际演练组装好的应用，确保所有独立单元能够很好地在一起工作，以组成功能完整的应用。

编写和维护自动化测试的技能像其他技能一样，需要学习和磨炼。在测试旅程的开始，你也许会觉得测试实践因为拖慢了进度而不值得实施，但随着对测试技巧的精通，你会赞叹对代码的严苛测试实际上会为你省下很多时间。



有句名言说，在没有版本控制系统（VCS）的情况下编写代码，相当于在没有降落伞的情况下跳伞。时至今日，几乎没有人会在没有VCS的情况下做项目。同样的类比也适用于自动化测试，我们可以说：“在没有自动化测试的情况下编写代码，就像在没有安全绳索的情况下攀登，或者在没有降落伞的情况下跳伞。”你可以这样尝试，但结果几乎可以确定是灾难性的。

AngularJS 团队完全认同自动化测试的重要性，他们在框架上工作时严格遵守测试流程。了不起的是，为了让应用更容易测试，AngularJS 拥有整套工具和库。

单元测试 Unit tests

AngularJS 采用 Jasmine 作为它的测试框架：AngularJS 框架自身的单元测试代码即为 Jasmine 所编写，文档中的所有例子也使用 Jasmine 语法。此外，AngularJS 还扩展了原生 Jasmine 库，为其添加了几个让测试更容易的组件。

剖析Jasmine测试

Jasmine 称自己为“测试 JavaScript 代码的行为驱动（behavior-driven）开发框架”。行为驱动测试代码读起来像英语句子。下面看看对标准 JavaScript 类的简单测试，以了解实践中它是如何工作的：

```
describe('hello World test', function () {  
  
    var greeter;  
    beforeEach(function () {  
        greeter = new Greeter();  
    });  
  
    it('should say hello to theWorld',function({  
        expect(greeter.say('World')).toEqual('Hello, World!');  
    });  
});
```

在上面的测试中有几个结构体需要解释：

- `describe`函数用于描述应用的一个特性，它的作用是成为单个测试的聚合容器。如果你熟悉其他测试框架，就会想到测试套件的`describe`区块（`block`），它们能够在内部互相嵌套（`nested`）。
- 测试自身位于`it`函数内部，这里的考虑是在一个测试中测试某特性的某个特定方面。每个测试都有名字和主体，通常测试主体的起始部分调用被测对象的方法，而后面的部分用于核实结果是否符合预期。
- 包含在`beforeEach`块中的代码会在每次单个测试运行前执行，这里是安放任何需要在每次测试前执行的初始化代码的好地方。
- 最后要提到的是`expect`和`toEqual`函数，我们可以运用这两个结构体去比较预期与实际结果。Jasmine和很多其他流行的测试框架一样，拥有丰富的`matcher`集，如`toBeTruthy`、`toBeDefined`、`toContain`，这些只是很少的几个例子。

测试AngularJS对象

测试 AngularJS 对象和测试任何其他合格的 JavaScript 类并没有太多不同。唯一的不同是 AngularJS 的依赖注入系统。为了学习编写能利用依赖注入机制的测试代码，我们将专注于测试服务（services）和控制器（controller）。



所有与测试相关的扩展和mock对象都分离出来放在名为 angular-mocks.js 的 AngularJS 文件中，别忘记在测试运行器中包含此脚本。同时，也记得不要在应用的部署版本中包含它。

测试服务

为注册在 AngularJS 模块中的对象编写测试并不难，但需要进行初始化设置。更进一步地，我们要确保适当的 AngularJS 模块被成功初始化，整个依赖注入机制运转正常。幸运的是，AngularJS 提供了一套方法，让 Jasmine 测试和依赖注入系统能够非常好地协同工作。

在第 1 章（Angular 之禅）中，我们介绍了 notificationsArchive，现在要为其编写简单的测试代码，以看看如何测试 AngularJS 的服务。为了起到提醒作用，给出服务本身的代码如下：

```
angular.module('archive', [])
  .factory('notificationsArchive', function () {

    var archivedNotifications = [];
    return {
      archive:function (notification) {
        archivedNotifications.push(notification);
      },
      getArchived:function () {
        return archivedNotifications;
      }
    };
  });
```

如下是对应的测试代码：

```
describe('notifications archive tests', function () {

  var notificationsArchive;
  beforeEach(module('archive'));
```

```

beforeEach(inject(function (_notificationsArchive_) {
    notificationsArchive = _notificationsArchive_;
}));

it('should give access to the archived items', function () {
    var notification = {msg: 'Old message.'};
    notificationsArchive.archive(notification);

    expect(notificationsArchive.getArchived())
        .toContain(notification);
});
});

```

你应该能认出 **Jasmine** 测试中熟悉的结构体，外加一些新的函数调用 `module` 和 `inject`。

在 **Jasmine** 测试中，`module` 函数用于指出要准备好来自给定模块的服务。这种方法的角色与 `ng-app` 指令类似，它指出 **AngularJS** `$injector` 应该为给定模块（以及所有依赖模块）所创建。



不要把测试中使用的 `module` 函数和 `angular.module` 方法混淆起来，它们虽然名字一样，但角色截然不同。`angular.module` 方法用于声明新的模块，而 `module` 函数则允许我们指定测试中用到的特定模块。

在现实中，有可能在一个测试中有多次对 `module` 函数的调用。这时所有来自指定模块的服务（`services`）、值（`values`）与常量（`constants`）都在 `$injector` 中有效。

`inject` 函数只负责一件事，就是将服务注入测试。

最让人疑惑的，也许是 `inject` 函数调用中出现的神秘下画线：

```

var notificationsArchive;
beforeEach(inject(function (_notificationsArchive_) {
    notificationsArchive = _notificationsArchive_;
}));

```

实际上，`$injector` 将在检查函数的参数以获取依赖时，去掉成对的首尾下画线。这是个有用的小技巧，因为我们能节省出没有下画线的变量名给测试自己使用。

测试控制器

对控制器的测试和对服务的测试遵循相似的模式。让我们看看来自示例应用的 `ProjectsEditCtrl` 控制器的片段，此控制器负责在应用的管理部分中编辑项目。这里将测试控制器中负责增加和移除项目团队成员的方法：

```
angular.module('admin-projects', [])
  .controller('ProjectsEditCtrl', function($scope, project) {

    $scope.project = project;

    $scope.removeTeamMember = function(teamMember) {
      var idx = $scope.project.teamMembers.indexOf(teamMember);
      if(idx >= 0) {
        $scope.project.teamMembers.splice(idx, 1);
      }
    };

    //控制器的其他方法
  });
```

上面给出的控制器逻辑并不复杂，可以将精力集中在测试代码上，如下：

```
describe('ProjectsEditCtrl tests', function () {

  var $scope;
  beforeEach(module('admin-projects'));
  beforeEach(inject(function ($rootScope) {
    $scope = $rootScope.$new();
  }));

  it('should remove an existing team member', inject(function(
    $controller) {

    var teamMember = {};
    $controller('ProjectsEditCtrl', {
      $scope: $scope,
      project: {
        teamMembers: [teamMember]
      }
    });

    //核实初始设置
    expect($scope.project.teamMembers).toEqual([teamMember]);
```

```

//执行并核实结果
$scope.removeTeamMember(teamMember);
expect($scope.project.teamMembers).toEqual([]);
    }));
});

```

要测试的 `removeTeamMember` 方法由 `ProjectsEditCtrl` 控制器定义在 `$scope` 上。为了有效测试此方法，需要创建新的 `scope`、新的 `ProjectsEditCtrl` 控制器的实例，并将它们连接起来。从本质上说，我们需要手动做 `ng-controller` 指令的工作。

让我们再次把视线转向 `beforeEach` 部分，这里有些有趣的事情。首先，我们访问了 `$rootScope` 服务，并创建了新的 `$scope` 实例 (`$scope.$new()`)。我们这样做是为了模拟运行中应用的情况，在那里，`ng-controller` 指令创建新的 `scope`。

我们用 `$controller` 服务创建控制器实例（请注意 `inject` 函数在 `it` 级别和在 `beforeEach` 部分都能发挥作用）。



调用 `$controller` 服务指定控制器的构造函数参数，如上所述十分容易。这里依赖注入的优势得以体现——我们提供了伪造的 `scope` 和测试数据，去完全孤立地测试控制器的实现。



Mock对象和异步代码测试

我们看到 **AngularJS** 很好地集成了依赖注入系统和 **Jasmine** 测试框架，但是，还有更多关于可测试性的好消息，**AngularJS** 提供了一些出色的 `mock` 对象。

异步编程在 **JavaScript** 中非常普遍，不幸的是，异步代码不易测试。异步事件不是很好预测，能以任意顺序触发，而且往往在一段不可知的时间后触发。好消息是，**AngularJS** 团队提供了出色的 `mock` 对象，让测试异步代码非常轻松，更重要的是，它们还是快速的和可预测的。如何能做到这一点？让我们使用 `$timeout` 服务的代码的测试例子，先看看被测试的代码：

```

angular.module('async', [])
  .factory('asyncGreeter', function ($timeout, $log) {
    return {

```

```

say:function(name, timeout) {
  $timeout(function(){
    $log.info('Hello, ' + name + '!');
  })
}
};
});

```



\$timeout服务是JavaScript `setTimeout` 函数的代替，使用\$timeout延时执行动作更合适，因为它与 AngularJS的HTML编译器紧密结合，会在到时后触发 DOM 刷新，从而使得代码更容易测试。

测试代码如下：

```

describe('Async Greeter test', function () {

  var asyncGreeter, $timeout, $log;
  beforeEach(module('async'));
  beforeEach(inject(function(_asyncGreeter_, _$timeout_, _$log_){
    asyncGreeter = _asyncGreeter_;
    $timeout = _$timeout_;
    $log = _$log_;
  }));

  it('should greet the async World', function () {
    asyncGreeter.say('World', 999999999999999999);
    //
    $timeout.flush();
    expect($log.info.logs).toContain(['Hello, World!']);
  });
});

```

上述代码大部分容易理解，但有两个非常有趣的地方值得注意。首先，我们看到对 `$timeout` 这个 mock 对象调用了 `$timeout.flush()` 方法，这模拟了一次异步事件的触发，而且有趣的是，我们不需要等待时间到期，也不需要外部事件的帮助，就可以完全控制何时触发这个事件。注意，这里我们设定了非常长的过期时间，但测试还是立即执行，这是因为没有依赖 JavaScript 的 `setTimeout`，而用 `$timeout` mock 对象来模拟异步环境下的行为。



为异步服务实现的可预测的mock对象，是AngularJS测试可以运行如此之快的原因之一。

在很多平台上经常会有难以测试的基础性全局服务，日志记录和异常处理代码就是这样的让测试困扰的例子。幸运的是，AngularJS 有所补救——它能够提供和相应 mock 对象一并记录这些基础关系的服务。你也许已经注意到上面的测试例子中用到了名为 `$log` 的 mock 对象，它模仿 `$log` 服务，收集并存储所有的日志记录以备后用。使用 mock 对象，无须调用真正的平台服务——特别是当它们会影响性能，或者有副作用的时候（比如，想象 `$log` 服务可能通过 HTTP 发送日志到后台服务器，而在执行测试时进行网络通信是一个非常糟糕的主意）。

端对端测试 End-to-end tests

AngularJS 引入了自有的端对端测试解决方案 Scenario Runner，它通过在真正的浏览器中执行动作来进行集成测试，这些动作（填充表单、点击按钮和链接等）会自动执行并确认 UI 响应（改变页面、显示合适的信息等），因而显著减少了对传统测试（通常由质量保障团队手工执行）的需求。

AngularJS 的解决方案与其他现存工具（比如非常流行的 Selenium）功能相似，但它与框架结合得更紧密，特别是：

- Scenario Runner 会注意到异步事件（特别是XHR调用），它能暂停测试的执行，直到异步事件完成为止。在实践中，这意味着测试代码不需要用到任何显式的等待指令。任何曾经使用传统工具测试过重Ajax的web应用的人，都会非常感谢这个特性。
- 可以利用已存在于AngularJS模板中的绑定信息来选择DOM元素，以便进行进一步操作和检查。从本质上来说，能够基于模型（model）来找到它们绑定的DOM元素和表单输入，这意味着不再需要嵌入表层HTML属性（通常是ID或CSS类），也不用依靠不稳定的XPath表达式来让测试框架找到DOM元素。

- **Scenario Runner**使用的语法，让寻找DOM元素、与它们互动、指定它们的属性等事情变得相当容易，有完整的领域驱动语言（DSL）来完成对输入框和重复列表等组件的搜索和匹配。

不幸的是，随着时间的流逝，**Scenario Runner** 自身实现的限制不断增多，因此到本书写作时为止，**Scenario Runner** 并没有得到积极的维护。现在的替代方案是基于 **Selenium** 集成的 **Protractor**，可以在 **GitHub** 上查询它的进展：<https://github.com/angular/protractor>。



我们不推荐在新项目中使用**Scenario Runner**，它的维护情况很糟糕，可以选择**Protractor**作为替代方案。

日常工作流

自动化测试需要严苛的应用才有效，测试要尽可能地频繁运行，失败的测试也要尽快地修复。失败的测试应该被当成失败的构建，而修复失败的构建应该是团队的首要任务。

我们经常在 **JavaScript** 代码和 **UI** 模板之间来回切换，当专注于纯 **JavaScript** 代码和其他 **AngularJS** 产物（比如过滤器或指令）时，频繁运行单元测试就很重要了。实际上，由于 **Karma runner** 运行 **AngularJS** 测试时非常高速，甚至可以在每次文件保存时都运行所有的单元测试！

本书作者在编写示例 **SCRUM** 应用时，将 **Karma runner** 设置成监控所有文件的变化情况（包括源代码和测试代码），每次保存文件时，都会执行整套测试以提供即刻健康检查。在这样的设置下，反馈循环变得非常短，我们就能掌握几秒前编写的代码是否如预期般工作。如果一切正常，则可以迅速前进；如果测试失败，就知道这是由于最后的代码变化所导致的。当测试运行如此频繁时，那么可以向漫长而枯燥的调试过程说再见了。

运行测试无须太多努力，营造舒适的测试环境非常重要，这样自动化测试才可能尽量频繁地执行。如果运行测试需要好几个手动的枯燥步骤，那么我们会逃避测试。

现代工具有可能帮你做极端高效的设置，比如，图 2-7 是撰写本书示例时的开发环境截图，你能看到源代码和测试代码被并列打开，所以很容易就可以在它们之间互相切换。Karma runner 用于监控文件系统，在每次文件保存操作时都执行测试，并立即提供反馈信息（在下面的面板上）。



图2-7



这里描述的测试环境实际上还是尽量轻量化的开发环境，我们用所有的时间专注于在IDE内编写代码，没有必要切换环境以进行构建、部署，或者在浏览器内点击来验证代码工作情况这样的行为。

Karma runner的提示与技巧

测试驱动开发（TDD）实践显著减少了漫长调试过程发生的次数。编写代码测试时，首先进行小的改动，然后频繁运行测试，这样导致测试失败的代码通常会仅就在几次击键之前。但是，如果不借助调试器，那么我们依然会不理解发生了什么，尽管尽了最大努力。在这样的时刻，我们要能够快速孤立出失败的测试，并专注于它。

执行测试子集

Karma runner 装载的 Jasmine 版本拥有非常有用的扩展功能，可以快速隔离出一套测试子集来运行：

- 为测试或测试套件加字母x做前缀（xit、xdescribe），将在下次运行时禁用此测试或测试套件。
- 为测试套件加字母d做前缀（ddescribe）将只运行此测试套件，而忽略其他测试套件。
- 为测试加字母i做前缀（iit）将只运行此测试，而忽略其他测试和测试套件。

这些小前缀在实践中极其有用，用法如下：

```
describe('tips & tricks', function () {

  xdescribe('none of the tests here will execute', function () {

    it('won't execute - spec level', function () {
    });

    xit('won't execute - test level', function () {
    });
  });

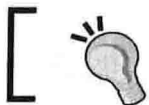
  describe('suite with one test selected', function () {

    iit('will execute only this test', function () {
    });

    ('will be executed only after removing iit', function () {
    });
  });
});
```

调试

缩小失败测试的范围，只是成功的一半。我们要理解其真相，就要经常参与其调试的过程。但是，怎样调试 Karma runner 执行的测试代码呢？这其实很容易——只要在测试或生成代码中加入 debugger 语句就足够了。



使用debugger语句时，不要忘记在你最喜欢的浏览器中打开开发者工具，否则浏览器不会暂停让你调试。

有时仅在控制台上打印一些变量的值也许更容易，AngularJS 的 mock 对象对这种情况有相应的便利方法 `angular.mock.dump(object)`，实际上 `dump` 方法也暴露给了全局（`window`）作用域，所以也可简写为 `dump(object)`。

2.6 小结 Summary

本章为编写真实的 AngularJS 应用做好准备，一开始就浏览了示例应用的细节——它所在的问题领域以及技术栈。本书余下部分有很多示例应用（用 SCRUM 方法管理项目）中的例子，我们将用基于 JavaScript 的（或至少对 JavaScript 友好的）全开源技术栈来编写此应用——在线寄存的 MongoDB 作为持久化存储，`node.js` 作为服务器端方案，很显然，还有 AngularJS 作为前端框架。

我们花了大量时间讨论构建相关的问题，即完美构建系统背后的哲学及构建的实际实现。我们选择基于 JavaScript（由 `node.js` 支持）的 `grunt.js` 来编写示例构建脚本。研究构建，不可避免地会引导我们去研究示例项目中文件和目录的组织关系。我们选择可以与 AngularJS 模块系统很好协作的目录结构，在仔细衡量几种不同的声明 AngularJS 模块和 `provider` 的方式后，我们选用了合适的语法，它避免混淆，不污染全局空间，也将构建系统的复杂度抑制在了合理的范围内。

自动化测试的益处，近年来已被熟知与理解。AngularJS 很强调用它编写代码的可测试性，我们看到 AngularJS 提供了完整的测试工具（`Karma runner`）、与现有测试库的结合（`Jasmine`），以及依赖注入系统的支持，以便于独立地测试单个对象。我们看到如何在实践中测试 AngularJS 产物，对于服务和控制器的测试也已不是秘密。有了这些解决方案，就很容易组建起非常高效的轻量化测试 workflow。

至此，我们已经学习了 AngularJS 的基础知识，了解了如何构架有规模的应用。现在将在此基础上增加具体的、真实的元素。因为任何 CRUD 应用都要与来自持久化存储的数据打交道，第 3 章将介绍 AngularJS 与多样的后端交流的方式。

第 3 章

与后端服务器通信

Communicating with a Back-end Server

web 应用经常要与持久化存储通信，以获取和操作数据，对本质上就是数据编辑的 CRUD 类应用尤为如此。

AngularJS 能够通过 **XMLHttpRequest** (XHR) 和 **JSONP** 请求与各种后端交流，拥有通用的 `$http` 服务以进行 XHR 和 JSONP 调用，以及专门面向 RESTful 后端接口的 `$resource` 服务。

在本章中，将研究与后端通信的 API 与技术，特别是以下内容：

- 使用 `$http` 服务进行基本的 XHR 调用，以及进行相应的测试。
- 利用 AngularJS 的 `$q` 服务提供的 promise API 进行有效的异步请求。
- 使用 `$resource` 工厂轻松与 RESTful 后端进行沟通。
- 根据后端需求构建自定义的与 `$resource` 类似的 API。

3.1 使用 `$http` 进行 XHR 和 JSONP 请求 Making XHR and JSONP requests with `$http`

`$http` 服务是进行 XHR 和 JSONP 调用的基础通用 API。我们很快将看到它被精心制作以便于使用。但是，在深入介绍 `$http` API 的细节之前，需要对示例 SCRUM 应用的数据模型有更多的了解，这样示例才会更有意义。

熟悉数据模型和 MongoLab URLs

Getting familiar with the data model and MongoLab URLs

示例 SCRUM 应用的数据模型相当简单，可以用图 3-1 描绘。

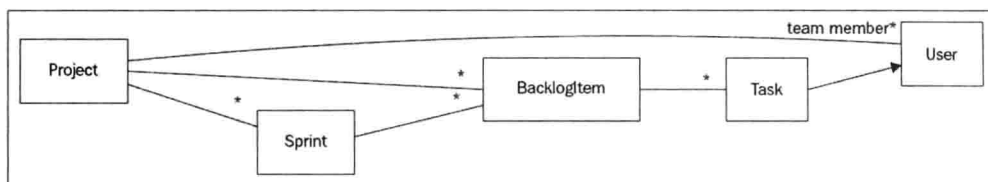


图3-1

这里有五个不同的 MongoDB 集合，用于保存与用户、项目有关的数据。所有数据都可以通过 MongoLab 提供的 RESTful 接口访问，这些 REST API 则通过定义好的 URL 来调用：

```
https://api.mongolab.com/api/1/databases/[DB name]/collections/
[collection name]/[item id]?apiKey=[secret key]
```



所有访问MongoLab上的数据库的REST API调用，都要包括名为apiKey的请求参数，这个参数用于鉴权，且对于每个账号是唯一的。要查阅MongoLab提供的REST API的完整描述，可以访问 <https://support.mongolab.com/entries/20433053-rest-api-for-mongodb>。

\$http API 快速导览

\$http APIs quick tour

利用 \$http 服务进行 XHR 和 JSONP 调用直截了当，比如下面的例子通过 GET 请求获取 JSON 内容：

```
var futureResponse = $http.get('data.json');
futureResponse.success(function(data, status, headers, config) {
    $scope.data = data;
});
futureResponse.error(function (data, status, headers, config) {
    throw new Error('Something went wrong...');
});
```

首先我们看到了 XHR GET 请求的调用方法，当然也有对其他 XHR 请求的：

- GET:\$http.get(url, config)。
- POST:\$http.post(url, data, config)。
- PUT:\$http.put(url, data, config)。
- DELETE:\$http.delete(url, config)。
- HEAD:\$http.head。

也可以用 \$http.jsonp(url, config) 触发 JSONP 请求。

\$http 接受的参数根据所用的 HTTP 方法稍有不同，对于在请求体中携带数据的调用（POST 和 PUT），方法签名如下：

- url: 调用目标URL。
- data: 请求体中送出的数据。
- config: 包含额外配置信息的JavaScript配置对象，对请求和响应都有影响。

对于其他在请求体中没有数据的方法（GET、DELETE、HEAD、JSONP），签名更简单，被缩减到只有两个参数：url 和 config。

我们可以在调用 \$http 方法返回的对象上，注册成功与失败的回调(callback)。

配置对象说明

JavaScript 配置对象保存着很多可选项，以影响请求、响应及传送的数据。配置对象中较重要的属性如下。

- method: 所用的HTTP方法。
- url: 请求的目标URL。
- params: URL的参数。
- headers: 额外的请求头。
- timeout: XHR请求终止前的超时时间（单位是毫秒）。
- cache: XHR GET请求的缓存开关。
- transformRequest、transformResponse: 在与后端交换数据前或交换后，对数据进行处理的数据变换函数。

你也许会为 `method` 和 `url` 出现在配置选项中感到惊讶，因为它们本身就已经是 `$http` 方法的参数了。但实际上，`$http` 有更通用的形式：

```
$http(configObject);
```

这种通用形式在 **AngularJS** 没有提供“捷径”方法的时候很有用（如 **PATCH** 或 **OPTIONS** 请求）。一般而言，使用捷径方法会得到更准确和更容易阅读的代码，所以我们推荐尽可能地使用捷径方法代替通用形式。

转换请求数据

`$http.post` 和 `$http.put` 方法接受任何 **JavaScript** 对象（或字符串）值作为它们的 `data` 参数。如果 `data` 是 **JavaScript** 对象，则 `data` 会被默认转换为 **JSON** 字符串。



默认 `data` 到 **JSON** 的转换机制，忽视任何以 `$` 开头的属性。一般来说，在 **AngularJS** 中，以 `$` 开头的属性意味着“私有（private）”。这也许会给某些后端带来麻烦，因为我们需要发送以 `$` 开头的数据（如 **MongoDB**）。解决方法是手动转换数据（如使用 `JSON.stringify` 方法）。

来看看实际中的数据转换，向 **MongoLab** 提交一个 **POST** 请求以创建新用户：

```
var userToAdd = {
  name: 'AngularJS Superhero',
  email: 'superhero@angularjs.org'
};

$http.post('https://api.mongolab.com/api/1/databases/ascrum/
  collections/users',
  userToAdd, {
    params: {
      apiKey: '4fb51e55e4b02e56a67b0b66'
    }
  });
```

这个例子也说明了怎样为 **URL** 添加 **HTTP** 查询参数（这里是 `apiKey`）。

处理HTTP响应

请求可能成功或失败,AngularJS 提供两种方法以注册对应这两种结果的回调: `success` 和 `error`。它们都接受 `callback` 函数,此函数会调用如下参数:

- `data`: 实际的响应数据。
- `status`: 响应的HTTP状态。
- `headers`: 访问HTTP响应头信息的函数。
- `config`: 请求触发时提供的配置对象。



AngularJS会对状态在200~299范围内的HTTP响应调用 `success`回调,状态不在此范围内的响应会触发 `error`回调,重定向响应(HTTP状态码3xx)则会被浏览器自动跟随。



成功和失败回调都是可选的,如果没有注册任何回调,则响应会安静地忽略掉。

转换响应数据

和转换请求数据一样, `$http` 服务会试图将响应中包含的 `JSON` 字符串转换为 `JavaScript` 对象,这种转换发生在成功或失败回调之前,默认的转换行为是可定制的。



在当前版本的AngularJS中, `$http`服务试图会对任何看起来像JSON的响应(以`{`或`[`开始,以`}`或`]`结束)都执行从JSON字符串到JavaScript对象的转换。



处理同源政策约束

Dealing with same-origin policy restrictions

web 浏览器强制实行同源安全政策(same-origin security policy),此政策仅对与目标资源来自同源(协议(protocol)、主机(host)和端口(port)的结合)的 XHR 互动授权,并对与“外部”资源的互动加以限制。

作为 web 开发者，需要在安全性和多个源聚合数据的功能需求之间取得平衡。实际上，从第三方服务抓取数据，并将其呈现在 web 应用中，往往是很值得做的。不幸的是，XHR 请求不能轻易到达源域之外的服务器，除非使用些小诡计。

有些技术可以访问外部服务器提供的数据：JSON with paddings (JSONP)、Cross-origin resource sharing (CORS) 也许是其中在现代 web 开发中最流行的。本节会展示 AngularJS 如何帮助我们在实践中运用这些技术。

利用JSONP克服同源政策约束

利用 JSONP，可以超越同源政策约束来获取数据。它的实现，有赖于浏览器能够自由地通过 `<script>` 标签从外部服务器获取 JavaScript。

JSONP 调用不触发 XHR 请求，取而代之的是生成一个 `<script>` 标签，其源 (source) 指向外部资源。当产生的 `<script>` 标签出现在 DOM 中时，浏览器履行它的职责去请求外部服务器，而服务器会在 web 应用的上下文中以函数调用填补响应 (JSONP 命名中的 “padding”)。

下面通过例子看看 JSONP 请求和响应在实践中是如何工作的。首先需要发起 JSONP 请求：

```
$http
  .jsonp('http://angularjs.org/greet.php?callback=JSON_CALLBACK', {
    params:{
      name:'World'
    }
  }).success(function (data) {
    $scope.greeting = data;
  });
```

一旦调用 `$http.jsonp` 方法，AngularJS 就会动态创建新的 `<script>` DOM 元素，代码如下：

```
<script type="text/javascript" src="http://angularjs.org/greet.
php?callback=angular.callbacks._k&name=World"></script>
```

当这段 `script` 标签被加进 DOM 时，浏览器就会请求 `src` 属性中指定的 URL，返回的响应体会遵循如下模式：

```
angular.callbacks._k({"name":"World","salutation":"Hello","greeting": "Hello World!"});
```

JSONP 响应看起来像标准的 JavaScript 函数调用，而实际上它也确实如此。AngularJS 在幕后生成 `angular.callbacks._k` 函数，该函数调用时会触发成功回调。提供给 `$http.jsonp` 函数调用的 URL 必须包含 `JSON_CALLBACK` 请求参数，AngularJS 会将此字符串转换为动态产生的函数名。



AngularJS 产生的 JSONP 回调名称会拥有如 `angular.callbacks._[variable]` 这样的形式，请确保后端能够接受包含点号的回调名称。

JSONP 的限制

对于同源政策约束，JSONP 是很聪明有效的解决办法，但它还是有几个限制。首先，只能用 JSONP 技术提交 GET HTTP 请求；其次，错误处理也相当麻烦，因为浏览器不会通过 `<script>` 标签暴露 HTTP 响应状态。在实践中，这意味着难以报告 HTTP 状态错误，并调用错误回调。

JSONP 也给 web 应用带来了一些潜在安全问题。除了众所周知的 XSS 攻击，最大的问题可能是服务器在 JSONP 响应中能够生成起决定性作用的 JavaScript 代码，这些代码会被浏览器加载，并在用户会话的上下文中执行。不怀好意的服务器会执行那些应避而远之的脚本，这会造成不同程度的伤害，从单纯破坏页面到盗取敏感数据。所以，应该认真选择 JSONP 请求的目标服务，并只使用可信任的服务器。

利用 CORS 克服同源政策约束

CORS (cross-origin resource sharing) 是一套 W3C 标准，致力于标准、可靠、安全地解决上述 JSONP 所面对的问题。CORS 标准基于 `XMLHttpRequest` 对象，采用清晰和可控的方式进行跨域 AJAX 请求。

CORS 的指导思想是浏览器和外界服务器需要协同（通过发送适当的请求和响应头）进行有条件的跨域请求。外界服务器需要依此配置，浏览器则必须发送恰当的请求和请求头，并翻译服务器响应以成功完成跨域请求。



外界服务器需要进行适当配置以加入CORS会话，关于如何配置服务器以接受HTTP CORS的更多信息可以参阅<http://www.html5rocks.com/en/tutorials/cors/>。这里主要专注于通信过程中浏览器所扮演的角色。

CORS 请求大致上可以分为“简单”和“非简单”两类。GET、POST 和 HEAD 请求被认为属于“简单”类（仅当发送的头信息在允许范围内时），使用其他 HTTP 动词或允许范围外的请求头会使浏览器强行提交“非简单”类的 CORS 请求。



大部分现代浏览器都自带对CORS通信的支持。但是，IE 8 和IE 9只能通过非标准的XDomainRequest对象提供CORS支持，因为这个IE特定的XDomainRequest对象实现的限制，AngularJS对它不提供支持。因此，\$http服务在IE 8和IE 9下不支持CORS请求。

对于“非简单”类的请求，浏览器会强制要求在提交主要请求前先发送探测用的（preflight）OPTIONS 请求，并等待服务器的批准。这经常会产生疑惑，因为观察 HTTP 通信流会看到很多神秘的 OPTIONS 请求。可以通过从浏览器直接调用 MongoLab 的 REST API 来看到这些请求，比如，观察删除用户时的 HTTP 通信：

```
$http.delete('https://api.mongolab.com/api/1/databases/ascrum/
collections/users/' + userId,
{
  params:{
    apiKey:'4fb51e55e4b02e56a67b0b66'
  }
});
```

能看到两个以同样的 URL 为目标的请求 (**OPTIONS** 和 **DELETE**), 如图 3-2 所示。

Elements	Resources	Network	Sources	Timeline	Profiles	Audits	Console	AngularJS
Name	Method	Status	Type	Initiator	Size	Time		
userid?apiKey=4fb51e55e4b02e56a67b0b66	OPTIONS	200	text/html	angular.js:9002	656 B	697 ms		
userid?apiKey=4fb51e55e4b02e56a67b0b66	DELETE	200	application/j...	Other	346 B	438 ms		

图3-2

MongoLab 服务器的响应包含下面的头信息, 以促成最后的 **DELETE** 请求, 如图 3-3 所示。

▼ Response Headers	view source
Access-Control-Allow-Credentials: true	
Access-Control-Allow-Headers: accept, origin, x-requested-with, content-type	
Access-Control-Allow-Methods: DELETE	
Access-Control-Allow-Methods: OPTIONS	
Access-Control-Allow-Methods: PUT	
Access-Control-Allow-Methods: GET	
Access-Control-Allow-Origin: *	
Access-Control-Max-Age: 1728000	
Allow: PUT	
Allow: OPTIONS	
Allow: DELETE	
Allow: GET	

图3-3

MongoLab 服务器进行了很好的配置, 以发送合适的头信息来响应 **CORS** 请求。如果服务器没有进行合适的配置, 那么 **OPTIONS** 请求就会失败, 目标请求也不会被执行。



不要对 **OPTIONS** 请求感到惊讶, 这只是 **CORS** 工作中的握手机制。失败的 **OPTIONS** 请求在大部分情况下意味着服务器没有进行良好配置。

服务器端代理

JSONP 并不是进行跨域请求的理想技术。**CORS** 标准虽好, 但它依然需要服务器端的额外配置, 以及支持此标准的浏览器。

如果你不能使用 **CORS** 和 **JSONP** 技术, 还有一种避免跨域请求问题的方法, 那就是为外界服务器配置本地服务器做代理。应用正确的服务器配置, 可以通过自己的服务器代理跨域请求, 这样浏览器只会将服务器作为目标。这种技术在所有的浏览器上都有效, 也不需要事先用 **OPTIONS** 请求试探。而且, 我们不用冒任何安全风险。这种方法的唯一缺点就是, 我们依旧需要对服务器进行配置。



本书描述的示例SCRUM应用依靠配置好的node.js服务器，它代理所有对MongoLab REST API的调用。

3.2 promise API 与 \$q The promise API with \$q

JavaScript 程序员很习惯异步（asynchronous）编程模型，无论是浏览器还是 node.js 执行环境，都充斥着异步事件：XHR 响应、DOM 事件、IO 与延时处理，它们可能在任何时刻以随机顺序被触发。即使我们有经验成功应付这些执行环境下的异步状况，异步编程还是很让人困惑，尤其是同步化（synchronizing）多个异步事件的时候。

在同步（synchronous）的世界里，链式方法调用（在一种方法执行的结果上调用另一种方法）和异常处理（使用 try/catch）都直截了当。在异步的世界里，我们不能简单地链接函数调用，而要依靠回调。回调在仅处理一个异步事件时工作得很好，但一旦要协调多个异步事件时，事情就开始变得复杂了，这时特别难的是异常状况处理。

为了让异步编程更容易，Promise API 在最近被几个流行的 JavaScript 库所采用，它背后的概念并不算新颖，早在 20 世纪 70 年代末期就已被提出，不过直到最近，这些概念才为主流 JavaScript 编程所吸收。



Promise API 的主要思想是，为异步世界带来我们在同步编程世界中所享有的那些便利之处，比如链式方法调用和错误处理。

AngularJS 拥有非常轻量化的 Promise API 实现——\$q 服务。为数不少的 AngularJS 服务（主要是 \$http，也有 \$timeout 和其他服务）相当依赖 promise 风格的 API，所以，为了有效运用这些服务，需要熟悉 \$q。



\$q 服务的灵感来自于 Kris Kowal 的 Q Promise API 库（<https://github.com/kris/kowal/q>）。你也许想要检出（checkout）这个库，以获得更好的对 promise 概念的理解，并用 AngularJS 的轻量化实现和这个功能完整的 Promise API 库进行比较。

工作中的 promise 和 \$q 服务

Working with promises and the \$q service

为了学习 \$q 服务提供的精简 API，我们来看看现实中的例子，以证明除了 XHR 调用外，Promise API 还可以应用在任何异步事件上。

学习\$q服务的基础知识

假设我们想打电话叫比萨，结果不外乎两种，一种是比萨按时送达，另一种是接到告知我们订单有问题的电话。预订比萨只需要打个电话，但外送（订单实现）却需要时间，它是异步的。

为了感受 Promise API，我们用 \$q 服务来将比萨订单与它的成功递送模型化。首先，我们将定义一位顾客，他的行为包括享用比萨，或者在订单没有送达时感到失望，代码如下：

```
var Person = function (name, $log) {

  this.eat = function (food) {
    $log.info(name + " is eating delicious " + food);
  };

  this.beHungry = function (reason) {
    $log.warn(name + " is hungry because: " + reason);
  };
};
```

上面定义的 Person 构造函数可用于生产包含 eat 和 beHungry 方法的对象，我们将用这些方法分别作为成功与失败的回调。

现在，我们用 Jasmine 测试比萨下单和完成过程的代码，如下：

```
it('should illustrate basic usage of $q', function () {

  var pizzaOrderFulfillment = $q.defer();
  var pizzaDelivered = pizzaOrderFulfillment.promise;

  pizzaDelivered.then(pawel.eat, pawel.beHungry);


  pizzaOrderFulfillment.resolve('Margherita');
  $rootScope.$digest();

  expect($log.info.logs).toContain(['Pawel is eating
    deliciousMargherita']);
});
```

这个单元测试在一开始调用 `$q.defer()` 方法以得到一个延迟 (deferred) 对象, 从概念上来说, 它反映了未来将完成的任务 (或失败), 此对象有两项职责:


- 保存一个承诺 (promise) 对象 (在 `promise` 属性上), 这是被延迟的任务未来结果 (成功或失败) 的占位符。
- 提供方法以使此未来的任务完成 (履行, `resolve`) 或失败 (拒绝, `reject`)。

在 **Promise API** 中, 通常有两个角色: 一个是控制未来任务的执行 (在延迟对象上调用方法), 另一个则依赖于未来任务的执行结果 (保存承诺结果)。

 延迟对象 (deferred object) 反映未来将要完成或失败的任务, 承诺对象 (promise object) 则是此任务在未来执行结果的占位符。

控制未来任务的实体 (在例子里是餐馆) 提供承诺对象 (`pizzaOrderFulfillment.promise`) 给对任务结果感兴趣的其他实体。在上面的例子中, **Pawel** 对外送订单感兴趣, 并通过在承诺对象上注册回调来表达他的兴趣。为了注册回调, 需要使用 `then(successCallBack, errorCallBack)` 方法, 它接受回调函数。这些回调函数在被调用时, 会接收未来的任务结果 (成功回调) 或失败原因 (错误回调) 作为参数。错误回调是可选且可被省略的, 如果它被省略了, 当未来任务失败时, 失败也会被安静地忽视。

想通知未来任务使之完成, 就要在延迟对象上调用 `resolve` 方法, 传递给 `resolve` 方法的参数将会提供给注册过的成功回调。在成功回调之后, 未来任务完成, 承诺得到履行。类似地, 对 `reject` 方法的调用会触发错误回调和对承诺的拒绝 (promise rejection)。

 在上面的测试示例中, 调用了神秘的 `$rootScope.$digest()` 方法。在 **AngularJS** 中, 承诺被履行 (或被拒绝) 的结果作为 `$digest` 周期的一部分来传播。你可以参考第11章 (开发健壮的 **AngularJS** 应用) 来了解更多 **AngularJS** 的内幕及 `$digest` 周期。

promise是第一类JavaScript对象

乍看上去好像 promise API 增加了不必要的复杂度，但是，我们还有更多的例子来欣赏 promise 的威力。首先，我们要了解 promise 是第一类 JavaScript 对象，它可以作为函数调用的参数与返回值，这让我们可以很轻易地将异步操作封装为服务。比如设想简化后的餐馆服务，代码如下：

```
var Restaurant = function ($q, $rootScope) {

    var currentOrder;

    this.takeOrder = function (orderedItems) {
        currentOrder = {
            deferred:$q.defer(),
            items:orderedItems
        };
        return currentOrder.deferred.promise;
    };

    this.deliverOrder =function() {
        currentOrder.deferred.resolve(currentOrder.items);
        $rootScope.$digest();
    };

    this.problemWithOrder = function(reason) {
        currentOrder.deferred.reject(reason);
        $rootScope.$digest();
    };
};
```

现在餐馆服务封装好了异步任务，它仅在 takeOrder 方法中返回承诺对象。此对象随后被餐馆的顾客用于保存承诺的结果，而且此对象将在结果有效后获得通知。

下面看看新制作的 API 在实践中如何工作，让我们编写代码来描绘拒绝承诺时调用错误回调的情况：

```
it('should illustrate promise rejection', function () {

    pizzaPit = new Restaurant($q, $rootScope);
    var pizzaDelivered = pizzaPit.takeOrder('Capricciosa');
```

```

    pizzaDelivered.then(pawel.eat, pawel.beHungry);

    pizzaPit.problemWithOrder('no Capricciosa, only Margherita
    left');
    expect($log.warn.logs).toContain(['Pawel is hungry because: no
    Capricciosa, only Margherita left']);
  });

```

聚合回调

可以在同一个承诺对象上注册多个回调，让我们设想本书的两位作者都预订了比萨，因此都对订单的递送感兴趣，代码如下：

```

it('should allow callbacks aggregation', function () {

  var pizzaDelivered = pizzaPit.takeOrder('Margherita');
  pizzaDelivered.then(pawel.eat, pawel.beHungry);
  pizzaDelivered.then(pete.eat, pete.beHungry);

  pizzaPit.deliverOrder();
  expect($log.info.logs).toContain(['Pawel is eating delicious
  Margherita']);
  expect($log.info.logs).toContain(['Peter is eating delicious
  Margherita']);
});

```

上述注册了多个成功回调，而且它们都在履行承诺的时候被调用了。类似地，拒绝承诺时也会调用所有注册的错误回调。

注册回调和承诺的生命周期

承诺一旦被履行或拒绝，就不能再改变它的状态，它只有一次机会提供承诺结果。换句话说，就是不可能做到：

- 履行已被拒绝的承诺。
- 履行已被履行过的承诺以得到不同的结果。
- 拒绝已被履行的承诺。
- 拒绝已被拒绝的承诺，提供不同的拒绝理由。

这些规则相当直观，例如，比萨如果已经成功送达（也许还被吃掉了），那么再去通知回调订单有问题，就显然没有意义。



任何在承诺履行后再注册的回调将会与起始的回调有同样的结果。

异步动作的链式调用

聚集回调很不错。但 Promise API 真正的强大之处在于，它能够在异步世界中模仿同步函数调用。

继续比萨示例，设想这次朋友招待我们吃比萨，主人将预订比萨，并当它送达时切好上桌。这是一串异步事件：首先比萨要被送达，然后才能准备上桌。这等于在我们能够吃饭之前，有两个承诺要被履行：餐馆承诺送餐，主人承诺送到的比萨会被切好上桌。下面看看模型化这种状况的代码：

```
it('should illustrate successful promise chaining', function () {

  var slice = function(pizza) {
    return "sliced "+pizza;
  };

  pizzaPit.takeOrder('Margherita').then(slice).then(pawel.eat);

  pizzaPit.deliverOrder();
  expect($log.info.logs).toContain(['Pawel is eating delicious sliced
    Margherita']);});
```

在上面的例子中，我们看到了一个承诺链（对 then 方法的调用），这种结构与同步风格的代码十分相似：

```
pawel.eat(slice(pizzaPit));
```



仅当 then 方法返回新的承诺对象时，承诺链才能成立。这个返回的承诺对象将以回调的返回结果为参数来履行承诺。

更让人印象深刻的是，现在处理错误变得非常简单。下面看看失败传播到承诺所负责的人时的情况：

```
it('should illustrate promise rejection in chain', function () {

  pizzaPit.takeOrder('Capricciosa').then(slice).then(pawel.eat,
```

```

        pawel.beHungry);

    pizzaPit.problemWithOrder('no Capricciosa, only Margherita
        left');
    expect($log.warn.logs).toContain(['Pawel is hungry because: no
        Capricciosa, only Margherita left']);
  });

```

餐馆对承诺的拒绝传播给了对最终结果感兴趣的人，这恰恰是同步世界里异常处理的工作机制：抛出的异常会冒泡给第一个捕获（**catch**）块。

在 **Promise API** 中，错误回调与捕获块起同样的作用，而且相当标准——有好几个可选项去处理异常状况，比如可以：

- 恢复（从捕获块返回值）。
- 宣布传播失败（再次抛出异常）。

利用 **Promise API** 可以轻易模拟捕获块中的恢复，比如，假设主人将会在想点的比萨无法提供时，设法预订另一个比萨：

```

it('should illustrate recovery from promise rejection', function () {

    var retry = function(reason) {
        return pizzaPit.takeOrder('Margherita').then(slice);
    };

    pizzaPit.takeOrder('Capricciosa')
        .then(slice, retry)
        .then(pawel.eat, pawel.beHungry);

    pizzaPit.problemWithOrder('no Capricciosa, only Margherita left');
    pizzaPit.deliverOrder();

    expect($log.info.logs).toContain(['Pawel is eating delicious sliced
        Margherita']);
  });

```

我们能从错误回调中返回新的承诺对象，它将成为结算链的一部分，最终消费者甚至不会注意到有些东西出错。这是一种非常强大的模式，能运用于任何失败后重试的场景。我们将在第7章（安全）中使用这种方法实现安全检查。

其他场景下，比如不可能恢复的时候，我们应该考虑重新抛出异常。这种情况下唯一的选择就是触发另外一个错误，\$q 服务有一种对应的专用方法（\$q.reject）：

```
it('should illustrate explicit rejection in chain', function () {

  var explain = function(reason) {
    return $q.reject('ordered pizza not available');
  };

  pizzaPit.takeOrder('Capricciosa')
  .then(slice, explain)
  .then(pawel.eat, pawel.beHungry);

  pizzaPit.problemWithOrder('no Capricciosa, only Margherita
    left');

  expect($log.warn.logs).toContain(['Pawel is hungry because:
    ordered pizza not available']);
});
```

\$q.reject 方法相当于异步世界中的抛出异常，它返回新的被拒绝的承诺对象，并利用参数指定拒绝的理由。

关于\$q的其他

\$q 服务有两种额外的方法很有用：\$q.all 和 \$q.when。

1. 聚合承诺

\$q.all 方法能够同时启动多个异步任务，并在所有任务都完成后获得通知，它有效地从多个异步动作聚合承诺，并返回单一的、组合后的承诺作为接合点。

为了说明 \$q.all 方法的用途，让我们来看看从多个餐馆预定食物的示例，希望所有订单都送达后再开饭：

```
it('should illustrate promise aggregation', function () {

  var ordersDelivered = $q.all([
    pizzaPit.takeOrder('Pepperoni'),
    saladBar.takeOrder('Fresh salad')
  ])
```

```

    });

    ordersDelivered.then(pawel.eat);

    pizzaPit.deliverOrder();
    saladBar.deliverOrder();

    expect($log.info.logs).toContain(['Pawel is eating delicious
        Pepperoni, Fresh salad']);
    });

```

一方面, `$q.all` 方法接收包含承诺对象的数组作为参数, 并返回聚合后的承诺, 仅当所有单个承诺都履行后, 此聚合承诺才会被履行。另一方面, 一旦某单个动作失败, 整个聚合承诺就会被拒绝:

```

it('should illustrate promise aggregation when one of the promises
    fail', function () {

    var ordersDelivered = $q.all([
        pizzaPit.takeOrder('Pepperoni'),
        saladBar.takeOrder('Fresh salad')
    ]);

    ordersDelivered.then(pawel.eat, pawel.beHungry);

    pizzaPit.deliverOrder();
    saladBar.problemWithOrder('no fresh lettuce');
    expect($log.warn.logs).toContain(['Pawel is hungry because: no fresh
        lettuce']);
    });

```

基于同样的理由, 聚合承诺也被拒绝了。

2. 将值包装为承诺

有时候我们会遇到这样的状况, 同样的 API, 混用从异步动作与同步动作得来的结果, 这种情况下将所有的结果都当成异步的来处理, 往往会更容易些。



`$q.when`方法能够将JavaScript对象包装成承诺对象。

继续看“比萨与沙拉”的例子，设想沙拉已经准备完毕（同步动作），但比萨却需要预订和递送（异步动作），尽管如此，我们依然想将两种食物一并奉上。下面是描绘如何使用 `$q.when` 和 `$q.all` 方法优雅地实现此目标的示例：

```
it('should illustrate promise aggregation with $q.when', function () {

  var ordersDelivered = $q.all([
    pizzaPit.takeOrder('Pepperoni'),
    $q.when('home made salad')
  ]);

  ordersDelivered.then(pawel.eat, pawel.beHungry);

  pizzaPit.deliverOrder();
  expect($log.info.logs).toContain(['Pawel is eating delicious
    Pepperoni,home made salad']);
});
```

`$q.when` 方法返回的承诺对象，其履行值为调用 `when` 方法时提供的参数。

AngularJS 中的 \$q 集成

\$q integration in AngularJS

`$q` 服务不仅是相当胜任的（而且轻量化的）`Promise API` 实现，而且与 `AngularJS` 的渲染体系结合得相当紧密。


首先，承诺可以直接暴露于作用域之下，而且承诺一旦得到履行，就会被自动渲染。这允许我们将承诺视为模型的值，示例如下：

```
<h1>Hello, {{name}}!</h1>
```

控制器中的代码如下：

```
$scope.name = $timeout(function () {
  return "World";
}, 2000);
```

这段有名的文本 "Hello, World!" 将会在 2 秒后被渲染，无须任何程序员的手动干预。

 `$timeout`服务返回承诺，它将以`timeout`回调所返回的值来履行。


这种模式也许很方便，但它导致了不是很易读的代码。当我们得知从函数调用返回的承诺不会被自动渲染时，事情甚至会变得更让人迷惑，模板标记如下：

```
<h1>Hello, {{getName()}}!</h1>
```

控制器中的代码如下：

```
$scope.getName = function () {
    return $timeout(function () {
        return "World";
    }, 2000);
};
```

这段代码不会在模板上输出期望的文本。


 我们不建议为了履行值的自动渲染，而在 `$scope` 内直接暴露承诺。这样的方案相当令人迷惑，特别是考虑到从函数调用返回的承诺的不稳定行为的时候。

3.3 promise API 与 \$http The promise API with \$http

现在已介绍了承诺 API，现在 `$http` 方法调用返回的响应对象也不那么神秘了。回忆本章开头的简单示例，你会想起 `$http` 调用会返回一个对象，我们能在其上注册成功和失败回调。实际上，这个对象是一个完整的承诺对象，它还有两种额外的便利方法，即 `success` 和 `error`。像任何承诺一样，`$http` 调用返回的承诺对象为 `then` 方法，这让我们能以如下形式覆盖回调注册：

```
var responsePromise = $http.get('data.json');
responsePromise.then(function (response) {
    $scope.data = response.data;
}, function (response) {
    throw new Error('Something went wrong...');
});
```

`$http` 服务返回的承诺以响应对象为参数履行，此对象包括以下属性：`data`、`status`、`headers` 和 `config`。

 对 `$http` 服务方法调用返回的承诺，有两种附加方法（`success` 和 `error`）用于方便地注册回调。

由于 `$http` 服务从它的方法调用返回承诺对象，因此我们能在与后端交互的时候充分利用 **Promise API**。聚集回调、链接请求及在异步世界中也受惠于精巧的错误处理。

3.4 与 RESTful 端点通信


Communicating with RESTful endpoints

对于通过网络提供服务来说，表述性状态转移（**representational state transfer, REST**）是很流行的架构方案。`$http` 提供的界面已经能够轻易与任何基于 **AngularJS** 的 web 应用的 **RESTful** 端点交互。不过，更进一步，**AngularJS** 为之提供了专用的 `$resource` 服务，这让此类交互变得更加容易。

`$resource` 服务

The `$resource` service

RESTful 端点通常提供 **CRUD** 操作，它们可以通过在一组类似的 **URL** 上调用不同的 **HTTP** 方法来访问。与这样的端点交互的代码通常很简单直接，不过写起来却很单调乏味。`$resource` 服务允许我们消除重复的代码，并让我们在更高的抽象级别上操作，用对象（资源，**resources**）的形式去思考数据操纵，用方法调用来代替低级别的 **HTTP** 调用。

 `$resource` 服务被分发在单独的文件中（`angular-resource.js`），属于专用模块（`ngResource`）。为了利用 `$resource` 服务，我们需要包含 `angular-resource.js` 文件，并在应用模块中声明对 `ngResource` 模块的依赖。

为了查看用 `$resource` 服务与 **RESTful** 端点交互有多么容易，我们将在 **MongoLab** 提供的 **RESTful** 服务之上构建抽象层，比如用户列表：

```
angular.module('resource', ['ngResource'])

.factory('Users', function ($resource) {

return
  $resource('https://api.mongolab.com/api/1/databases/ascrum/
    collections/users/:id', {
```

```

    apiKey: '4fb51e55e4b02e56a67b0b66',
    id: '@_id.$oid'
  });
});

```

一开始我们就为 `Users` 构造函数注册了生成方案（工厂），但请注意，无须为这个构造函数编写任何代码，`$resource` 服务会为我们准备好实现的代码。

`$resource` 服务会采用一组方法用于与 **RESTful** 端点进行交互，例如，想要查询持久化存储中的所有用户，只要编写如下简单代码：

```

.controller('ResourceCtrl', function($scope, Users){
  $scope.users = Users.query();
});

```

调用 `Users.query()` 方法时，`$resource` 生成的代码将发起 `$http` 调用。当响应准备好时，收到的 **JSON** 字符串将会被转换为 **JavaScript** 数组，其中每个元素都为 `User` 类型。

 对 `$resource` 服务的调用会返回新生成的构造函数，它会被加入下列方法以和 **RESTful** 端点进行交互：`query`、`get`、`save` 和 `delete`。

要生成功能完善的资源（**resource**），**AngularJS** 并不需要多少信息。下面看看 `$resource` 方法的参数，以了解必要的输入和可定制化的可选项：

```

$resresource('https://api.mongolab.com/api/1/databases/ascrum/
collections/users/:id', {
  apiKey: '4fb51e55e4b02e56a67b0b66',
  id: '@_id.$oid'
});

```

第一个参数是 **URL** 或 **URL 模式**，后者能包含有名字的占位符，命名以冒号开始。我们可以只指定一种 **URL 模式**，这意味着所有的 **HTTP** 动词都会使用非常相似的 **URL**。

 如果端口号被包括在后端的 **URL** 中，那么在为 `$resource` 调用提供 **URL 模式** 时，端口号需要被转义（**escaped**，例如，`http://example.com\\:3000/api`）。这是因为在 `$resource` 的 **URL 模式** 中，冒号有着特殊的含义。

`$resource` 函数的第二个参数，能定制每次请求所发送的默认参数，请注意这里的“参数”既可以是 URL 模板中的定位符，也可以是作为查询字符串的标准请求参数。

AngularJS 首先会尝试去为 URL 模板“填洞”，然后再将剩下的参数加入 URL 查询字符串中。

默认的参数既可以为静态参数（在工厂中指定），也可以是从资源对象中获得的动态参数，后者的值以 `@` 字符为前缀。

构造级与实例级方法

`$resource` 服务自动生成两套方法。其中一套方法会在构造级（类级）上针对给定资源生成，这些方法的目的是操作资源的列表，或者在没有任何资源实例被创建时提供所需的功能。另一套方法会对特定资源的某个实例有效，此类方法负责与单个资源（数据存储中的单条记录）交互。

1. 构造级方法

`$resource` 产生的构造函数对不同的 HTTP 动词有相应的方法。

- `Users.query(params, successcb, errorcb)`：发布 HTTP GET 请求，期望 JSON 响应返回数组，用于取得条目的列表。
- `Users.get(params, successcb, errorcb)`：发布 HTTP GET 请求，期望 JSON 响应返回单个对象，用于取得单个条目。
- `Users.save(params, payloadData, successcb, errorcb)`：发布 HTTP POST 请求，请求体从载荷（payload）中产生。
- `Users.delete(params, successcb, errorcb)`（别名：`Users.remove`）：发布 HTTP DELETE 请求。

对上面列出的所有方法，`successcb` 和 `errorcb` 分别表示成功与失败的回调函数。`params` 参数允许我们为每个动作指定参数，它们将成为 URL 的一部分，或者作为查询字符串中的参数。最后，`payloadData` 参数让我们在合适的情况（POST 和 PUT 请求）下指定 HTTP 请求体。

2. 实例级方法

`$resource` 服务不仅生成构造函数，而且会添加原型（实例）级别的方法，它们与类级的那些对应方法等效，但操作的对象变成了单个实例。例如，单个用户既可以通过下面的调用删除：

```
Users.delete({}, user);
```

也可以在用户的实例上调用方法：


```
user.$delete();
```

实例级方法非常方便，我们可以精准地控制资源。下面看另一个保存新用户的例子：

```
var user = new Users({
  name: 'Superhero'
});
user.$save();
```


也可以使用类级的`save`方法重写代码：

```
var user = {
  name: 'Superhero'
};
Users.save(user);
```

 `$resource` 工厂同时生成类级和实例级方法，后者以 `$` 字符为前缀。两类方法等效，所以何者更为便利，取决于实际情况。

3. 自定义方法

`$resource` 工厂默认生成的方法对典型用例来说已相当够用，但如果后端的某些操作使用不同的 HTTP 动词（如 `PUT` 或 `PATCH`），那么在资源级别上添加自定义方法也相当容易。

 默认情况下，`$resource` 工厂不生成任何对应 HTTP `PUT` 请求的方法，如果后端将任何操作映射给 HTTP `PUT` 请求，就不得不手工添加这些方法。

例如, **MongoLab REST API** 使用 **HTTP POST** 方法来创建新的条目。但当更新现有条目时, 则必须使用 **PUT** 方法。让我们看看如何自定义 **update** 方法 (类级的 **update** 和实例级的 **\$update**):

```
.factory('Users', function ($resource) {
  return $resource('https://api.mongolab.com/api/1/databases/
    ascrum/collections/users/:id', {
      apiKey: '4fb51e55e4b02e56a67b0b66',
      id: '@_id.$oid'
    }, {
      update: {method: 'PUT'}
    });
});
```

如你所见, 定义新方法很简单, 只要为 **\$resource** 工厂函数提供第三个参数即可。此参数必须是以下形式的对象:

```
action: {method:?, params:?, isArray:?, headers:??}
```

action 键是生成的新方法名称, 该方法会发起 **method** 指定的 **HTTP** 请求, **params** 用于保存此动作的默认参数, **isArray** 则指定从后端返回的数据是表示列表 (数组) 还是单个对象, 也可以自定义 **HTTP** 头信息。

\$resource 服务仅能与返回 **JavaScript** 数组和对象的后端服务协同工作, 单个值 (原始类型) 不被支持。返回列表的方法 (被 **isArray** 标记) 必须返回 **JavaScript** 数组, 而数组一旦被包装进 **JavaScript** 对象中, 就不能如期工作了。

4. 为资源对象添加行为

\$resource 工厂生成构造函数, 与任何其他 **JavaScript** 构造函数一样, 可以用 **new** 关键字创建新的资源实例。我们也能扩展构造函数的原型, 好为资源对象增加新行为。设想我们在 **user** 级别拥有新的方法, 根据姓和名输出用户全名。以下是实现此功能推荐的方式:

```
.factory('Users', function ($resource)
  var Users = $resource('https://api.mongolab.com/api/1/databases/
    ascrum/collections/users/:id', {
      apiKey: '4fb51e55e4b02e56a67b0b66',
      id: '@_id.$oid'
```

```

    }, {
      update: {method: 'PUT'}
    });

    Users.prototype.getFullName = function() {
      return this.firstName + ' ' + this.lastName;
    };

    return Users;
  })

```

也有可能在类（构造）级别增加新方法。由于在 JavaScript 中的函数是第一类对象，因此可以在构造函数上定义新方法。这样就能“手工”添加自定义的方法，而不是依赖于 AngularJS 的自动化方法生成。如果在某种资源方法中需要一些非标准逻辑，这也许会有用。例如，在发起 PUT（update）请求时，MongoLab 的 REST API 需要从载荷（payload）移除对象的标示符。

\$resource 创建异步方法

下面再看看 query 方法的示例：

```
$scope.users = Users.query();
```

生成资源的行为是同步化风格的（没有使用任何回调或承诺），这让人印象深刻。其实 query 方法的调用是异步的，只是 AngularJS 运用了一个小诡计让它看起来是同步的而已。

真相是 AngularJS 会立即从对 Users.query() 的调用中返回空数组作为结果，然后当异步调用成功、真实数据从服务器抵达时，会用这些数据更新此数组。AngularJS 在开始时只是简单地保存对返回数组的引用，并在有了数据之后填充这个数组。这个小诡计在 AngularJS 中能够有效工作，因为当数据到达时，返回数组的内容会改变，对应模板也会自动进行刷新。

尽管如此，也请不要误会，\$resource 调用的确是异步的。这经常引起困惑，比如你也许会试图编写如下代码（或用其他方式访问初始数组）：

```
$scope.users = Users.query();
console.log($scope.users.length);
```

这是不会得到期望结果的！

幸运的是，还是可以在 \$resource 工厂生成的方法中使用回调，并覆盖前面提到的代码以让它的行为异步化：

```
Users.query(function(users) {
  $scope.users = users;
  console.log($scope.users.length);
});
```



`$resource` 工厂生成的方法是异步的，尽管 AngularJS 用了—
个聪明的诡计，让语法看起来是在处理同步的方法。

`$resource` 服务的限制

`$resource` 工厂是便利的服务，让我们不需花多少时间就能实际开始与 RESTful 后端的交流。但是，问题在于 `$resource` 是个通用服务，并没有针对任何特定的后端需求进行定制。因此，它会对我们选择的后端做出一些可能不符合实际情况的假设。

让 `$resource` 工厂与后端及 web 应用一起工作，是很棒的一件事。在很多用例中 `$resource` 都够用，但在更复杂的应用中，使用低级别的 `$http` 服务常常会更好。

使用 `$http` 自定义 REST 适配器 Custom REST adapters with `$http`

`$resource` 工厂非常方便，但如果想突破它的限制，那么基于 `$http` 服务创建自定义的类似 `$resource` 的工厂也相当容易。花时间编写自定义的资源工厂代码，就能完全控制关于 URL 和数据处理前后的所有事情。此外，我们也不需要再包含 `angular-resource.js` 文件，因此能为页面的总大小节省一些字节数。

以下是自定义工厂的简化示例，它类似于 `$resource` 工厂，专门用于访问 MongoLab RESTful API。熟悉 Promise API，是理解此实现的关键：

```
angular.module('mongolabResource', [])

.factory('mongolabResource', function ($http, MONGOLAB_CONFIG) {

  return function (collectionName) {

    //基础配置
    var collectionUrl =
      'https://api.mongolab.com/api/1/databases/' +
      MONGOLAB_CONFIG.DB_NAME +
```

```

        '/collections/' + collectionName;

    var defaultParams = {apiKey:MONGOLAB_CONFIG.API_KEY};

    //功能性方法
    var getId = function (data) {
        return data._id.$oid;
    };

    //新资源的构造函数
    var Resource = function (data) {
        angular.extend(this, data);
    };

    Resource.query = function (params) {
        return $http.get(collectionUrl, {
            params:angular.extend({q:JSON.stringify({} || params)},
            defaultParams)
        }).then(function (response) {
            var result = [];
            angular.forEach(response.data, function (value, key){
                result[key] = new Resource(value);
            });
            return result;
        });
    };

    Resource.save = function (data) {
        return $http.post(collectionUrl, data, {params:defaultParams})
            .then(function (response) {
                return new Resource(data);
            });
    };

    Resource.prototype.$save = function (data) {
    return Resource.save(this);
    };

    Resource.remove = function (data) {
        return $http.delete(collectionUrl + '', defaultParams)
            .then(function (response) {
    return new Resource(data);
            });
    };

    Resource.prototype.$remove = function (data) {

```



```

return Resource.remove(this);
};

//这里是其他CRUD方法

//快捷方法
Resource.prototype.$id = function () {
    return getId(this);
};

return Resource;
};
});

```

示例代码从定义新的模块（`mongolabResource`）和接受配置对象（`MONGOLAB_CONFIG`）的工厂（`mongolabResource`）开始，它们现在看起来已不再陌生。基于提供的配置对象，能准备出资源将用到的 URL，在这里我们完全控制了 URL 的创建和使用。

然后，声明 `Resource` 构造函数，这样就能从现有的数据中创建新的资源对象了。紧随其后的是几种方法的定义：`query`、`save` 和 `remove`。这些方法被定义在构造（类）级上，但实例级方法也会被声明（在适当的地方）以遵循原生 `$resource` 实现的惯例。提供实例级方法很容易，只要委托（`delegating`）给对应类级方法：

```

Resource.prototype.$save = function (data) {
    return Resource.save(this);
};

```

承诺链的用法是实现自定义资源的决定性组成部分。`then` 方法通常会在 `$http` 调用返回的承诺对象上调用，得到的结果承诺对象会返回给客户，在此对象上注册的成功回调会用于注册后处理（`post-processing`）逻辑。例如，在 `query` 方法的实现中，我们对服务器返回的原始 JSON 进行了后处理，并创建了资源实例。这样，我们再次完整地控制了响应数据的解析过程。

下面看看新的资源工厂如何使用吧！

```

angular.module('customResourceDemo', ['mongolabResource'])
    .constant('MONGOLAB_CONFIG', {
        DB_NAME: 'ascrum',
        API_KEY: '4fb51e55e4b02e56a67b0b66'
    })

```

```

    .factory('Users', function (mongolabResource) {
        return mongolabResource('users');
    })

    .controller('CustomResourceCtrl', function ($scope, Users,
        Projects) {

        Users.query().then(function (users) {
            $scope.users = users;
        });
    });

```

自定义资源的用法与原来的 `$resource` 并没有太多不一样。首先，我们要声明对自定义资源工厂所在模块（`mongolabResource`）的依赖；然后，我们以常量的形式提供所需的配置选项，一旦初始化设置完成，就能定义实际的资源，它们由自定义工厂创建，创建过程只要调用 `mongolabResource` 工厂函数，并传递 MongoDB 的列表名字作为参数就可以了。

在应用运行时，新定义的资源构造函数（这里是 `Users`）能够和任何其他依赖一样被注入（`injected`），之后它将用于调用类级或实例级方法，后者的示例如下：

```

$scope.addSuperhero = function () {
    new Users({name: 'Superhero'}).$save();
};

```

转换到自定义的基于 `$http` 的资源工厂，最大的优点是我们能享用到 Promise API 的全部好处。

3.5 使用 `$http` 的高级特性

Using advanced features of `$http`

`$http` 服务极其灵活与强大，这种力量来自于简洁灵活的 API 与 Promise API 的应用。在本节中，我们将学习如何使用 `$http` 服务中更高级的特性。

截取响应

Intercepting responses

AngularJS 内置的 `$http` 服务允许我们注册拦截器（`interceptors`），它将在每个请求前后执行，这样的拦截器在我们需要对大量（可能是全部）请求进行特殊处理时非常有用。

作为起始示例，设想我们要重试失败的请求。要达成此目的，我们可以定义一个拦截器用于检查响应状态码，并在探测到 **HTTP Service Unavailable** (503, 服务无效) 状态码时重试此请求，代码大略如下：

```
angular.module('httpInterceptors', [])

.config(function($httpProvider){
  $httpProvider.responseInterceptors.push('retryInterceptor');
})

.factory('retryInterceptor', function ($injector, $q) {

  return function(responsePromise) {
    return responsePromise.then(null, function(errResponse) {
      if (errResponse.status === 503) {
        return $injector.get('$http')(errResponse.config);
      } else {
        return $q.reject(errResponse);
      }
    });
  };
});
```

拦截器是一个函数，它接受原始请求的承诺作为参数，并返回另一个将履行成为拦截结果的承诺。在这里会检查 `errResponse.status` 以确定其是否处于需要恢复的错误条件下，如果确实如此，则返回来自全新 `$http` 调用的承诺，且此调用和原始请求有一样的配置对象。如果拦截到不能处理的错误，则简单地将错误传播出去就行 (`$q.reject` 方法)。

AngularJS 的拦截器利用了 `promise API`，这就是它们如此强大的原因。在上面的例子中，我们重试 **HTTP** 请求的方法对使用 `$http` 服务的客户完全透明。在第 7 章（安全）中会有使用响应拦截器来提供精巧安全机制的完整示例。


注册新的拦截器很容易，只要将对于新拦截器（工厂创建的 **AngularJS** 服务）的引用加入 `$httpProvider` 所维护的拦截器数组中就可。请注意，我们使用 `provider` 来注册新的拦截器，而它们仅在配置块内有效。

3.6 测试与 \$http 交互的代码

Testing code that interacts with \$http

因为网络延迟与数据的变化，测试调用外部 HTTP 服务的代码很让人头疼。我们想要测试运行得足够快，并且可预测。幸运的是，AngularJS 提供了模拟 HTTP 响应的卓越的仿制品（mocks）。

在 AngularJS 中，\$http 服务依赖于另一个低级服务——\$httpBackend，我们可以将其想象成对 XMLHttpRequest 对象的粗浅包装，掩盖了浏览器的不兼容，并允许 JSONP 请求。

 应用代码永远不应该直接调用 \$httpBackend，因为 \$http 服务提供的抽象好很多。但是，拥有一个分开的 \$httpBackend 服务意味着我们能在测试期间将它替换为 mock。

为了了解 \$httpBackend mock 在单元测试中的应用，我们来看看针对示例控制器的测试代码，此控制器通过 \$http 服务触发了 GET 请求：

```
.controller('UsersCtrl', function ($scope, $http) {

    $scope.queryUsers = function () {
    $http.get('http://localhost:3000/databases/ascrum/collections/
    users')

        .success(function (data, status, headers, config) {
            $scope.users = data;
        }).error(function (data, status, headers, config) {
            throw new Error('Something went wrong...');
        });
    };
});
```

为了测试控制器代码，可以编写如下程序：

```
describe('$http basic', function () {

    var $http, $httpBackend, $scope, ctrl;
    beforeEach(module('test-with-http-backend'));
    beforeEach(inject(function (_$http_, _$httpBackend_) {
        $http = _$http_;
        $httpBackend = _$httpBackend_;
```

```

    }));
    beforeEach(inject(function (_$rootScope_, _$controller_) {
        $scope = _$rootScope_.$new();
        ctrl = _$controller_('UsersCtrl', {
            $scope : $scope
        });
    }));

    it('should return all users', function () {

        //设置期待请求和响应
        $httpBackend
        .whenGET('http://localhost:3000/databases/ascrum/collections/users').
        respond([{name: 'Pawel'}, {name: 'Peter'}]);

        //调用代码以测试
        $scope.queryUsers();

        //模拟响应
        $httpBackend.flush();

        //核实结果
        expect($scope.users.length).toEqual(2);
    });

    afterEach(function() {
        $httpBackend.verifyNoOutstandingExpectation();
        $httpBackend.verifyNoOutstandingRequest();
    });
});

```

首先,我们了解到 `$httpBackend mock` 允许指定期望的请求 (`whenGET(...)`), 并准备虚假的响应 (`respond(...)`)。对于每个 HTTP 动词, 都有一整族的 `whenXXX` 方法, 这些方法的签名十分灵活, 甚至允许我们指定正则表达式作为 URL。

为了模仿从后端正常返回的数据, 可以使用 `respond` 方法。这种方法也同样可能有 `mock` 响应头。

`$httpBackend mock` 最大的优点就是, 它允许我们精准地控制响应及其时机。运用 `flush()` 方法, 我们不再依靠异步事件的怜悯, 而可以在任意指定时刻模拟后端传来的 HTTP 响应。使用 `$httpBackend mock` 的单元测试能够同步运行, 尽管 `$http` 服务在设计上是异步的, 这保证了单元测试能够以预期方式快速执行。

`verifyNoOutstandingExpectation` 方法可以确定所有调用都如期执行（`$http` 方法被调用，响应也被刷新），同时 `verifyNoOutstandingRequest` 调用可以确定测试代码没有触发任何意外的 XHR 调用。使用这两种方法，就能确定测试代码已执行，并且只执行所有预期之中的方法。

3.7 小结 Summary

本章概述了与后端通信、接收或操纵数据所采用的不同的方法。首先，从查看 `$http` 的 API 开始，它是在 AngularJS 中发送 XHR 和 JSONP 请求的基础服务。然后，熟悉了 `$http` 服务的基本 API，而且还仔细剖析了处理跨域请求的不同方案。

大多数 AngularJS 异步服务都依靠 Promise API 来提供优雅的界面。`$http` 服务就非常依赖承诺，所以我们不得不考察 AngularJS 中承诺的实现方法。我们了解到 `$q` 服务提供了通用的 Promise API，并与渲染机制紧密结合。对它的理解，也让我们完全理解了 `$http` 方法调用的返回值。

AngularJS 能够轻易与 RESTful 端点进行通信，专用的 `$resource` 工厂让编写与 RESTful 后端交互的代码非常轻松。尽管 `$resource` 工厂十分方便，但它也非常通用，因此不一定能够满足你所有的需要。基于 `$http` API，我们应该勇于创建自定义的、类似 `$resource` 的工厂。

在本章快到结尾时，我们提及了响应拦截器，它属于 `$http` API 的高级用法。

最后，像任何 JavaScript 代码一样，运用 `$http` API 的方法应该被彻底测试，我们看到 AngularJS 提供了出色的 mock 对象，以便更容易地编写与后端交流的单元测试代码。

带着所有加载到客户端并在 JavaScript 中有效的数据，我们准备开始深入研究在 AngularJS 中渲染这些数据的多种方法。第 4 章将专注于模板、指令（directive）和渲染。

第 4 章

显示与格式化数据

Displaying and Formatting Data

我们已经学习了如何从后端获取数据并加载入浏览器，现在可以来看看 AngularJS 如何显示和操作这些数据。本章首先会对模板中可用的 AngularJS 内置指令（**directives**）做一概览，讲解完这些基本知识后，再介绍这些指令在实践中的应用及典型模式。此外，本章还将介绍 AngularJS 过滤器（**filters**）。

在本章中你将学到：

- AngularJS指令的命名惯例。
- 根据条件显示和隐藏标记（**markup**）块。
- **repeater**（**ng-repeat**）指令的应用模式与陷阱。
- 注册DOM事件处理器（**handlers**）以便用户能够与应用交互。
- AngularJS中基于DOM模板语言的限制，以及可能的解决之道。
- 过滤器（**filters**）：它们的用途和用法示例，内置的过滤器以及如何创建和测试自定义过滤器。

4.1 引用指令

Referencing directives

在学习不同的 AngularJS 内置指令之前，要强调一件重要的事情，那就是我们能用多种命名惯例在 HTML 标记中引用指令。

在 AngularJS 的文档中，所有指令的名字以驼峰（camel-case）命名法被索引收录（比如，ngModel）。然而，在模板中，则需要使用蛇形（snake-case）命名法（ng-model），即以冒号分隔（ng:model）或者下画线分隔（ng_model）的形式。而且，对指令的每个引用都能以 x 或 data 作为前缀。



每个指令都可以被若干不同（但却等效）的名字所引用，以 ngModel 为例，我们能在模板中写出下面任何一个：ng-model、ng:model、ng_model、x-ng-model、x-ng:model、x-ng_model、data-ng-model、data-ng:model、data-ng_model、x:ng-model、data_ng-model，等等。

对于保证 HTML 文档遵循 HTML5 规范，并通过 HTML5 检验测试来说，data 前缀非常便利，除此而外，可以选择任何让你感到满意的命名方案，它们都是等效的。



本书中的所有示例，都使用最简短的蛇形命名法（如 ng-model），我们认为这样的语法简明易懂。

4.2 显示表达式的求值结果 Displaying results of expression evaluation

AngularJS 提供各种渲染数据的方法，可为用户显示一致的模型内容，其中有些值得指出的微妙之处。

插值指令 The interpolation directive

插值指令（interpolation directive）是显示模型数据时最基础的指令，它接受用花括号定界的表达式：

```
<span>{{expression}}</span>
```

该指令会简单地对 expression 求值，并在屏幕上渲染结果。

AngularJS 使用的定界符是可配置的，如果你计划混用 AngularJS 与服务器的其他模板语言，那么也许需要修改定界符的默认值。配置非常简单，仅需在 \$interpolateProvider 上设置属性：


```
myModule.config(function($interpolateProvider) {
  $interpolateProvider.startSymbol('[[');
  $interpolateProvider.endSymbol(']]');
});
```

这里我们将默认的 `{{}}` 修改为 `[[]]`，于是就可以在模板中写：

```
[[expression]]
```

利用 ngBind 渲染模型值

Rendering model values with ngBind

插值指令拥有名为 `ng-bind` 的等效指令，它可以被用作 **HTML** 属性：

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

花括号形式虽然更易用，但是，有时候使用 `ng-bind` 指令更方便，它通常可用于在 **AngularJS** 处理表达式之前（页面初次加载）将其隐藏，这样做的好处是可以有效地预防 **UI** 闪烁，以提供更好的用户体验。关于初始页面加载的优化，将在第 12 章“打包和部署 **AngularJS** Web 应用”中介绍更多的细节。

AngularJS 表达式中的 HTML 内容

HTML content in AngularJS expressions

默认情况下，**AngularJS** 会对插值指令求值表达式（模型）中的任何 **HTML** 标记都进行转义，例如以下模型：

```
$scope.msg = 'Hello, <b>World</b>!';
```

以及标记片段：

```
<p>{{msg}}</p>
```

渲染过程会对 `` 标签进行转义，它们会显示为纯文本而非标记：

```
<p>Hello, &lt;b>World&lt;/b>!</p>
```

插值指令会对模型中的任意 **HTML** 内容进行转义，这是为了防止 **HTML** 注入攻击（**injection attacks**）。

如果因为某个理由，包含 **HTML** 标记的模型要被浏览器求值和渲染，那么可以用 `ng-bind-html-unsafe` 指令来关掉默认的 **HTML** 标签转义：

```
<p ng-bind-html-unsafe="msg"></p>
```


使用 `ng-bind-html-unsafe` 指令，将得到含有 `` 标签的，能被浏览器正常解释的 HTML 片段。

运用 `ng-bind-html-unsafe` 指令时，应该极度小心，它应被限制在你完全信任并能控制的表达式上，否则恶意用户也许会在你的页面上随心所欲地注入任何 HTML。

AngularJS 还有一个指令——`ng-bind-html`，它能够选择性地净化（`sanitize`）指定的 HTML 标签，同时允许其他标签被浏览器所解释，它的用法与前面不那么安全的对应指令类似：


```
<p ng-bind-html="msg"></p>
```

从转义的角度来看，`ng-bind-html` 指令是 `ng-bind-html-unsafe` 指令（允许所有的 HTML 标签）和插值（`interpolation`）指令（不允许任何 HTML 标签）的折中。

 `ng-bind-html`指令在分离的模块（`ngSanitize`）中，需要包含额外的源文件：`angular-sanitize.js`

如果想用 `ng-bind-html`，别忘记对 `ngSanitize` 模块声明依赖：

```
angular.module('expressionsEscaping', ['ngSanitize'])
  .controller('ExpressionsEscapingCtrl', function ($scope) {
    $scope.msg = 'Hello, <b>World</b>!';
  });
```

 除非你正与遗留的陈旧系统（CMS，发送HTML的后端等）一同工作，其他情况下应该尽量避免在模型里出现标记。这类标记不能包括AngularJS指令，并需要`ng-bind-html-unsafe`或`ng-bind-html`指令来获得想要的结果。

4.3 条件化显示

Conditional display

基于某些条件，显示和隐藏部分 DOM 是很普遍的需求，而 AngularJS 对此拥有四套不同的指令集（`ng-show/ng-hide`、`ng-switch-*`、`ng-if`、`ng-include`）。

`ng-show/ng-hide` 指令集用于隐藏（利用 CSS 显示规则）DOM 树的一部分，它基于对表达式的求值结果：

```
<div ng-show="showSecret">Secret</div>
```

上边的代码可以用 `ng-hide` 重写如下：

```
<div ng-hide="!showSecret">Secret</div>
```



`ng-show/ng-hide` 指令通过简单地应用 `style="display: none;"` 来隐藏 DOM 元素，这些元素并不从 DOM 树中移除。

如果试图从物理上根据条件增加和移除 DOM 节点，则 `ng-switch` 指令集使用会很方便：

```
<div ng-switch on="showSecret">
  <div ng-switch-when="true">Secret</div>
  <div ng-switch-default>Won't show you my secrets!</div>
</div>
```

`ng-switch` 指令与 JavaScript 的 `switch` 语句相当接近，在一个 `ng-switch` 中可以存在好几个 `ng-switch-when`。




`ng-show/ng-hide` 与 `ng-switch` 指令的主要区别是它们对待 DOM 元素的方式。`ng-switch` 指令会在 DOM 树上增加/移除 DOM 元素，而 `ng-show/ng-hide` 只是简单地使用 `style="display: none;"` 来隐藏元素。`ng-switch` 指令创建了新的作用域。

尽管 `ng-show/ng-hide` 指令容易使用，但在用于大量 DOM 节点上时可能会导致性能下降。如果你的性能问题与 DOM 树的大小相关，那么应该倾向于使用更冗长一些的 `ng-switch` 指令集了。

`ng-switch` 指令集的问题是，对于简单用例，语法也会变得十分冗长。幸运的是，AngularJS 还有一套指令集：`ng-if`。它与 `ng-switch` 指令行为类似（同样从 DOM 树中增加/移除元素），但语法却非常简单：

```
<div ng-if="showSecret">Secret</div>
```


 `ng-if`指令仅在最近的AngularJS版本中可用：1.1.x 或 1.2.x。

根据条件包含内容块


Including blocks of content conditionally

尽管不像 `if/else` 语句那样直接，但 `ng-include` 指令也可以根据条件显示动态的、AngularJS 支持的标记块。此指令有个非常棒的属性，它能根据表达式的求值结果，有条件地加载和显示子模板（partials），这让我们很容易创建高度动态化的页面。例如，根据用户的角色（role）来包含不同的用户编辑表单。在如下代码片段中，我们为管理员角色的用户加载了特定子模板：

```
<div ng-include="user.admin && 'edit.admin.html' || 'edit.user.html'">
</div>
```

 `ng-include`指令为其包含的每个子模板都创建了新的作用域。

此外，`ng-include` 也可用于将小些的标记片段组合成最终页面。

 `ng-include`指令接受表达式作为参数，所以，如果打算使用指向给定子模板的定值，就需要传递单引号括起来的字符串，例如，`<div ng-include="'header. tpl.html'">`

4.4 用 `ngRepeat` 指令渲染集合

Rendering collections with the `ngRepeat` directive

`ng-repeat` 指令也许是最常用和最强大的指令，它对集合的项进行迭代，为集合中每个条目创立新的 DOM 元素，而且，此指令并不止于对集合的初次渲染，它还对数据源进行持续监视，以在有任何变化发生时，重新渲染模板。



重复器 (repeater) 是经过高度优化的, 以便数据结构对 DOM 树的影响最小化。

在内部, ng-repeat 也许会选择移动 DOM 节点 (从数组中移动元素)、删除 DOM 节点 (从数组中删除元素), 以及插入新的节点 (从数组中新增元素)。无论重复器选择何种策略, 重要的是要认识到, 这不是只运行一次的单纯 for 循环。ng-repeat 指令的行为, 更像数据的观察者 (observer), 试图将集合中的条目映射 (map) 到对应的 DOM 节点之上, 而此过程将永久持续下去。

熟悉 ngRepeat 指令

Getting familiar with the ngRepeat directive

基本用例和语法非常简单, 如下:

```
<table class="table table-bordered">
  <tr ng-repeat="user in users">
    <td>{{user.name}}</td>
    <td>{{user.email}}</td>
  </tr>
</table>
```

users 数组定义在一个作用域上, 并包含典型的用户对象, 如 name、email 等属性。ng-repeat 指令将迭代此用户集合, 并为每个条目创建 <tr> DOM 元素。



ng-repeat 指令为其迭代的集合中的每个元素都创建新的作用域。

特殊变量

Special variables

AngularJS 重复器在为每个独立条目创建作用域时, 都会声明一组特殊变量, 它们可用于确定此元素在集合中的位置。

- \$index: 指代元素在集合中的索引数字 (从 0 开始)。
- \$first、\$middle、\$last: 这些变量会根据元素的位置获得对应的布尔值。

在实践中利用上述变量非常便利，例如示例 SCRUM 应用，可以依赖 `$last` 变量在面包屑（breadcrumb）元素中渲染导航链接，如图 4-1 所示。对于路径的最后（选定）部分，没有必要渲染链接，而路径的其他部分则都需要 `<a>` 元素。

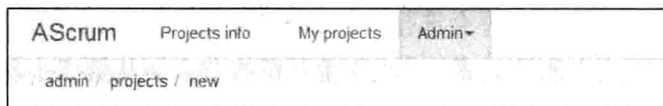


图4-1

可用如下代码模型化此 UI：

```
<li ng-repeat="breadcrumb in breadcrumbs.getAll()">
  <span class="divider">/</span>
  <ng-switch on="$last">
    <span ng-switch-when="true">{{breadcrumb.name}}</span>
    <span ng-switch-default>
      <a href="{{breadcrumb.path}}">{{breadcrumb.name}}</a>
    </span>
  </ng-switch>
</li>
```


迭代对象的属性

Iterating over an object's properties

通常，`ng-repeat` 指令用于显示来自 JavaScript 数组的条目，此外，它还可用于迭代对象的属性，这时语法会有些不同，如下：

```
<li ng-repeat="(name, value) in user">
  Property {{$index}} with {{name}} has value {{value}}
</li>
```

在上面的例子中，可以将用户对象的所有属性作为无序列表来显示。请注意，我们必须为属性名和属性值指定变量名，并用括号来注解（name, value）。

 `ng-repeat` 指令将会在输出结果前对属性按照其名字的字母顺序排序，此行为不能改变，所以没有办法在给对象使用 `ng-repeat` 时控制迭代的顺序。

`$index` 变量依然可用于获取给定属性在已排序好的全属性列表中的位置。

尽管可以迭代对象的属性，但仍然有些限制，其中最主要的就是我们不能控制迭代的顺序。



如果你想改变迭代对象属性的顺序，则可以在控制器中将这些属性进行排序，并将结果存入数组。

ngRepeat 模式

ngRepeat patterns

本节会介绍普通的展示模式，以及利用 AngularJS 实现它们的方法，尤其是关于列表的一些细节，以及如何改变列表中元素的类。

列表和细节

这是一个很常见的用例。一个列表，它的项在被点击时可以展开以显示更多的细节。此模式有两种变化：仅展开一个元素，同时展开多个元素。图 4-2 是描绘此 UI 设计的截图。

Name	e-mail
Pawel	pawel@domain.com
Pawel details go here...	
Peter	peter@domain.com

图4-2

1. 显示单行细节

用如下代码展开单个元素：

```
<table class="table table-bordered" ng-
  controller="ListAndOneDetailCtrl">

  <tbody ng-repeat="user in users" ng-click="selectUser(user)" ng-
    switch on="isSelected(user)">
    <tr>
      <td>{{user.name}}</td>
      <td>{{user.email}}</td>
    </tr>
```

```

<tr ng-switch-when="true">
  <td colspan="2">{{user.desc}}</td>
</tr>
</tbody>
</table>

```

在上面的例子中，仅当选中特定用户时，才会渲染包含此用户细节信息的额外的行，这个过程依靠 `selectUser` 和 `isSelected` 函数实现：

```

.controller('ListAndOneDetailCtrl', function ($scope, users) {
  $scope.users = users;

  $scope.selectUser = function (user) {
    $scope.selectedUser = user;
  };

  $scope.isSelected = function (user) {
    return $scope.selectedUser === user;
  };
})

```

这两个函数利用同一作用域（定义在整个表格的 DOM 元素上）来存储指向列表中激活项的指针（`selectedUser`）。

2. 显示多行细节

我们要改变策略来显示多行细节。这一次，对细节的选定状态应该被存储在所有单个元素的级别上。希望你还记得，`ng-repeat` 指令为其迭代的集合中的每个元素都创建新的作用域，我们将利用此新作用域来存储每个项的“选定”状态：

```

<table class="table table-bordered">
  <tbody ng-repeat="user in users" ng-controller="UserCtrl"
    ng-click="toggleSelected()" ng-switch on="isSelected()">
    <tr>
      <td>{{user.name}}</td>
      <td>{{user.email}}</td>
    </tr>
    <tr ng-switch-when="true">
      <td colspan="2">{{user.desc}}</td>
    </tr>
  </tbody>
</table>

```


这个例子很有意思，因为我们在每个项上都应用了 `ng-controller` 指令，此控制器会在作用域上定义函数和变量，以控制被选定的状态：

```
.controller('UserCtrl', function ($scope) {

    $scope.toggleSelected = function () {
        $scope.selected = !$scope.selected;
    };

    $scope.isSelected = function () {
        return $scope.selected;
    };

});
```

在 `ng-repeat` 指令作用的 DOM 元素上指定控制器，意味着了解此事的意义相当重要，即由此控制器来管理重复器生成的新作用域。在实践中，这意味着我们可以用专属的控制器来管理集合中的单个项。这是很强大的模式，因为能够很干净地封装单项特定的变量和行为（函数）。

改动表格、行和类

列表上经常会添加斑马纹（`zebra-striping`）来增加可读性，AngularJS 由一对指令（`ngClassEven` 和 `ngClassOdd`）来处理此任务：

```
<tr ng-repeat="user in users"
    ng-class-even="'light-gray'" ng-class-odd="'dark-gray'">
    ...
</tr>
```

`ngClassEven` 和 `ngClassOdd` 指令只是更通用的 `ngClass` 指令的特例。`ngClass` 是一个可用于多种情况下的多面手，可以重写上面的例子以显示其威力：

```
<tr ng-repeat="user in users"

    ng-class="{ 'dark-gray' : !$index%2, 'light-gray' : $index%2 }">
```

这里，`ngClass` 指令接受对象作为参数。该对象的键是类名，而值是条件表达式。键指定的类会根据值对应表达式求值的结果而添加或删除。



ng-class指令也能接受字符串或数组类型的参数，它们都包括由一个CSS类组成的列表（字符串参数用逗号分隔），以添加到指定元素上。

4.5 DOM 事件处理器 DOM event handlers

UI 只有在用户能与它进行交互时才有用（不管是通过鼠标、键盘，还是触摸事件）。好消息是，在 AngularJS 中注册事件处理器非常简单，下面是对点击事件做出反应的例子：

```
<button ng-click="clicked()">Click me!</button>
```

clicked() 表达式在当前的 \$scope 上求值，所以它能轻易调用定义在此作用域上的任何方法。在这里不只受限于简单的函数调用，也能使用更复杂的表达式，包括能够接受参数的表达式：

```
<input ng-model="name">
<button ng-click="hello(name)">Say hello!</button>
```



AngularJS的新手经常试着注册如下的事件处理器：ng-click="{{clicked()}}"或ng-click="sayHello({{name}})"。这样在属性值中使用插值（interpolation）表达式是不必要的和不正确的，AngularJS会在处理DOM时对插值表达式进行解析和求值，求值的结果才会成为真正注册的事件处理器。

AngularJS 对不同的事件有内置支持，由如下指令实现：

- 点击事件：ngClick和ngDbClick。
- 鼠标事件：ngMousedown、ngMouseup、ngMouseenter、ngMouseleave、ngMousemove和ngMouseover。
- 键盘事件：ngKeydown、ngKeyup和ngKeypress。
- 输入变动事件（ngChange）：ngChange指令与ngModel配合工作，让我们用户在用户输入引起模型变动时做出反应。

上述的 DOM 事件处理器都能够接收特别的参数 `$event`，以表示原始的 DOM 事件，这让我们可以访问事件的底层属性、阻止默认发生的动作、停止事件传播等。作为例子，下面看看如何读取被点击元素的点击位置：

```
<li ng-repeat="item in items" ng-click="logPosition(item, $event)">
  {{item}}
</li>
```

`logPosition` 函数定义在作用域上，代码如下：

```
$scope.readPosition = function (item, $event) {
  console.log(item + ' was clicked at: ' + $event.clientX + ',' +
    $event.clientY);
};
```



暴露给事件处理器的 `$event` 特殊变量，不应滥用于 DOM 操作中。想想在第 1 章中，我们谈到 AngularJS 中所有声明式的 UI 和 DOM 操作都应限制在指令上，这就是为什么 `$event` 参数大部分情况下都用在指令代码中的原因。

4.6 基于 DOM 的模板

Working effectively with DOM-based templates

采用 DOM 和 HTML 语法的模板系统并不常见，但实践中采用这样的方案却让人感到惊喜。习惯其他基于字符串模板引擎的人也许需要时间调整，但在写过一些基于 DOM 模板之后终会适应。下面是两个值得注意的地方。

习惯烦琐的语法

Living with verbose syntax

首先，一些构造函数的语法也许有点啰唆，像 `ng-switch` 族指令，某些时候需要多打些字。下面看一个基于列表中的项数量显示不同信息的例子：

```
<div ng-switch on="items.length>0">
  <span ng-switch-when="true">
    There are {{items.length}} items in the list.
  </span>
```

```

    <span ng-switch-when="false">
      There are no items in the list.
    </span>
  </div>

```

当然，可以将准备信息的逻辑部分移动到控制器中，以避免模板中出现 `switch` 语句。但在视图层中有这类简单的条件逻辑也并不为过。

幸运的是，在最新版本的 **AngularJS** 中有内置的 `ng-if` 指令，当你不需要完整的 `if/else` 表达式时，它是非常便利的工具。

ngRepeat 和多个 DOM 元素

ngRepeat and multiple DOM elements

更严重的问题是，`ng-repeat` 重复器在它采用最简化的形式时，仅仅知道如何重复单个元素（与它的子元素），这意味着 `ng-repeat` 不能管理一组相邻元素。

为了说明这点，让我们设想有一个列表，其中的项拥有名字和描述。现在希望用表格展示此列表，其中名字和描述被渲染为分离的行（`<tr>`）。问题是，我们要加上 `<tbody>` 标签以给 `ng-repeat` 指令找到一个挂靠的地方，代码如下：

```

<table>
  <tbody ng-repeat="item in items">
    <tr>
      <td>{{item.name}}</td>
    </tr>
    <tr>
      <td>{{item.description}}</td>
    </tr>
  </tbody>
</table>

```

看上去，`ng-repeat` 指令需要一个容器元素，它迫使我们创建确定的 **HTML** 结构。这是否会造成问题，取决于你有多严格地遵循 **web** 设计师描绘的标记结构蓝图。

在前面的例子里，幸好有合适的 **HTML** 容器存在（`<tbody>`），但有时候并没有这样可以让 `ng-repeat` 挂靠的 **HTML** 元素。下面看看 **HTML** 输出的例子：

```

<ul>
  <!-- we would like to put a repeater here -->
  <li><strong>{{item.name}}</strong></li>

```

```
<li>{{item.description}}</li>
<!-- and end it here -->
</ul>
```

新版的 AngularJS (1.2.x) 会扩展 `ngRepeat` 指令的基础语法, 为迭代 DOM 元素提供更多选择, 可以这样写:

```
<ul>
  <li ng-repeat-start="item in items">
    <strong>{{item.name}}</strong>
  </li>
  <li ng-repeat-end>{{item.description}}</li>
</ul>
```

使用 `ng-repeat-start` 和 `ng-repeat-end` 属性, 可以迭代一组相邻的元素。

不能在运行时修改的元素和属性

Elements and attributes that can't be modified at runtime

由于 AngularJS 的目标是浏览器中的 DOM 树, 所以它仅能进行浏览器允许的操作。在某些罕见的例子中, 有些浏览器在元素被插进 DOM 树后, 不允许对它们及其属性做任何变动。

为了观察这些限制在实际中的影响, 让我们设想一个常见的用例, 动态指定 `input` 元素的 `type` 属性。很多用户会尝试 (而且失败了!) 编写这样的代码:

```
<input type="{{myinput.type}}" ng-model="myobject[myinput.model]">
```

麻烦的是, 有些浏览器 (是的, 你猜对了, 就是 IE!) 不能修改刚创建好的 `input` 的 `type` 属性。这些浏览器认为 `{{myinput.type}}` (还未求值) 是一种输入类型, 由于不认识此类型, 它会被翻译成 `type="text"`。

有几种方案可以解决这个问题。但是, 在讨论它们之前, 我们要学习更多关于 AngularJS 自定义指令的知识。第 9 章 (创建自定义指令) 提供了其中一种解决方案, 另一种简单的方案则是利用内置的 `ng-include` 指令为不同的输入类型封装静态模板:

```
<ng-include src="'input'+myinput.type+'.html'"></ng-include>
```

被包含的片断用静态字符串指定输入的类型。



使用此技术时，注意作用域的问题，因为ng-include 创建了新作用域。

自定义 HTML 元素与 IE 的老版本

Custom HTML elements and older versions of IE

最后，我们要指出，自定义 HTML 元素和属性并不被 IE8 及以下版本支持。必须采取额外的措施才能在 IE8 和 IE7 中发挥 AngularJS 指令的全部优势，这些细节将在第 12 章（打包和部署 AngularJS Web 应用）中设置真实的部署场景进行描述。

4.7 使用过滤器处理模型变换

Handling model transformations with filters

AngularJS 表达式可能相当复杂，且包含函数调用，这些函数有不同的用途，但主要是模型变换和格式化。为了满足这些用例，AngularJS 表达式支持名为过滤器（filters）的特殊格式化（变换）函数：

```
{{user.signedUp | date:'yyyy-MM-dd'}}
```

此例中，date 过滤器用于格式化用户的注册日期。

过滤器是全局的命名函数，在视图中使用管道（|）符调用，接收用冒号（:）分隔的参数。实际上，我们可以用定义在作用域上的 formatDate 函数重写示例代码：

```
{{formatDate(user.signedUp, 'yyyy-MM-dd')}}
```

过滤器的优点是双重的：它们不需要在作用域上注册函数（所以在每个模板上都有效），而且，与规整的函数调用相比，它们通常有更简洁的语法。

上面的例子揭示了过滤器能够参数化（换句话说，参数能够传递给过滤器函数）：我们指定了日期格式，并将其作为参数传给了 date 过滤器。

几个过滤器可以混合（连锁）成一个变换管道，比如格式化字符串，我们可以限制它的长度为 80 个字符，同时转换所有字母为小写：

```
{{myLongString | limitTo:80 | lowercase}}
```

内置过滤器

Working with built-in filters

AngularJS 绑定了几个过滤器作为核心库的一部分，我们可以将它们大致分为两组，即格式化过滤器和数组变换过滤器。

格式化过滤器

下面是内置的格式化过滤器列表，很容易辨认它们的用途及使用场景。

- **currency**: 用两个小数位和一个货币符号来格式化数字。
- **date**: 根据指定的数据格式来格式化日期，模型包含的日期可表达为 **Date** 对象或字符串（这时字符串会在格式化前被解析为 **Date** 对象）。
- **number**: 用参数指定的小数位数量格式化输入。
- **lowercase** 与 **uppercase**: 就像名字提示的那样，这些过滤器用于格式化字符串为小写或大写形式。
- **json**: 此过滤器主要用于调试，它能打印漂亮的 **JavaScript** 对象，典型用法为:

```
<pre>{{someObject | json}}</pre>
```

。

数组变换过滤器

AngularJS 默认提供三种操作数组的过滤器。

- **limitTo**: 它将数组收缩到参数指定的长度，可以从集合的头或尾开始保留其中的元素（如果是尾部，则参数必须为负数）。
- **Filter**: 提供通用的过滤功能，它非常灵活，支持很多可以精确从集合中选择元素的选项。
- **orderBy**: 此排序过滤器根据给定的条件对数组中的元素进行排序。



这里列出的过滤器只对数组有效（除了`limitTo`，它也能用于字符串），当用于非数组的其他对象时，这些过滤器没有任何作用，只会单纯地返回原对象。

数组相关的过滤器经常与`ng-repeat`指令一起使用以渲染出过滤后的结果。在下面几节中，我们将构建一个完整的可以被排序、过滤和分页的表格示例。此例围绕示例应用的SCRUM待办事项列表构建，并展示如何混合应用过滤器和重复器指令。

1. 使用“filter”过滤器进行过滤

首先，我们要知道AngularJS有个名为`filter`的过滤器。这个名字有点不幸，因为“`filter`”可能会指代任何通用过滤器（变换函数），也可能会指代这个名为“`filter`”的特殊过滤器。

“`filter`”过滤器是一个通用的过滤函数，用于从数组中选择子集（或合并一些被排除在外的元素）。有一些参数格式可以提供给此过滤器，以支持元素选择过程。最简化的情况下，可以提供一个字符串，集合中的所有元素的所有字段都会检查是否含有此字符串作为子串。

作为例子，让我们基于搜索条件去过滤产品待办列表（product backlog list）。用户将面对一个输入框，可在其中键入搜索条件。结果列表应该只包括那些在某字段中提供的子字符串的元素，完成后的UI截图如图4-3所示。


Search for <input type="text" value="rev"/>				
Name	Description	Priority	Estimation	Done?
Review draft	Re-read and review the 1st draft	4	5	false
Incorporate reviewers remarks	Go over and reviewers remarks	6	3	false

图4-3

假定数据模型有如下属性：name、desc、priority、estimation 和 done。我们可以为前面讨论的 UI 编写这样的模板，代码如下：

```
<div class="well">
  <label>
    Search for:<input type="text" ng-model="criteria">
  </label>
</div>
<table class="table table-bordered">
  <thead>
    <th>Name</th>
    <th>Description</th>
    ...
  </thead>
  <tbody>
    <tr ng-repeat="backlogItem in backlog | filter:criteria">
      <td>{{backlogItem.name}}</td>
      <td>{{backlogItem.desc}}</td>
      ...
    </tr>
  </tbody>
</table>
```

像看到的那样，根据用户的输入增加过滤器非常容易，只需要将输入框的值作为参数传递给过滤器就可，AngularJS 的自动数据绑定和刷新机制会处理好其他一切事情。

[ 也可以用前缀!操作符来对匹配条件取反。]

在前面的例子中，我们用子字符串的匹配来搜索源对象的所有属性。如果想要更精准的属性匹配控制，则可以为过滤器提供一个对象参数。这个对象的行为像一个示例查询，比如，我们想要匹配 name 属性，且只包含还没有完成的条目：

```
ng-repeat="item in backlog | filter:{name: criteria, done: false}"
```

在此代码片段中，参数指定了对象必须匹配的所有属性，我们可以说这些由单个属性表达的条件是以 AND（与）逻辑操作符混合起来的。

AngularJS 还额外提供了匹配全部的属性名：\$。使用此通配符（wildcard）作为属性名，就可以混合使用 AND 和 OR（或）逻辑操作符。比如，我们想用字符串匹配源对象所有属性的搜索，只计入未完成的条目。这时过滤表达式可以重写如下：

```
ng-repeat="item in backlog | filter:{$: criteria, done: false}"
```

搜索条件的混合，也许会变得非常复杂，以至于无法再通过对象的语法表达。这时也可以将函数作为参数传递给过滤器（称为 predicate 函数）。此函数会作用于源列表的每一个单独元素，结果数组中仅包含过滤函数返回 true 的那些元素。设想个稍勉强的例子，想要查看已完成的且要多于 20 单位工效（units of effort）的待办事项，使用过滤器函数很容易编写代码，如下：

```
$scope.doneAndBigEffort = function (backlogItem) {
  return backlogItem.done && backlogItem.estimation > 20;
};
```

也很容易使用：

```
ng-repeat="item in backlog | filter:doneAndBigEffort"
```

对过滤结果计数

想要显示列表中条目的数量，通常可以用 {{myArray.length}} 表达式轻松做到。但是，当要显示经过过滤器过滤后的数组长度时，事情会变得有些复杂。一种不很成熟的解决方案是从重复器中复制过滤器到计数表达式中，比如：

```
<tr ng-repeat="item in backlog | filter:{$: criteria, done: false}">
```

我们尝试创建摘要行：

```
Total: {{(backlog | filter:{$: criteria, done: false}).length}}
```

这样做有几个缺点，不仅有了代码重复，而且同一过滤器要在两个不同的地方被执行好几次，从性能角度来说并不理想。

为了解决这些问题，可以创建一个中间变量（`filteredBacklog`），用于负责保存过滤后的数组：

```
ng-repeat="item in filteredBacklog = (backlog | filter:{$: criteria,
  done: false})"
```

然后对过滤结果计数，这只要显示存储数组的长度就可：

```
Total: {{filteredBacklog.length}}
```

对过滤结果计数的这种模式并不是很直观，然而，它允许将过滤逻辑放在一个地方。

另一种解决方法是，将整个过滤逻辑都放进控制器中，然后仅暴露过滤结果给作用域。此方法有个优点：过滤代码存放于控制器中，因此很容易进行单元测试。使用此方法，你需要学习如何在 **JavaScript** 中访问过滤器，这在本章稍后会提及。

2. 用orderBy过滤器排序

表格化的数据常可以被用户自由排序，点击单列的头部通常会将此列对应字段作为排序标准，再次点击时则会反转排序顺序。本节将用 **AngularJS** 实现此常见模式。

此工作的首选工具是 `orderBy` 过滤器，一旦完成，保存待办事项列表的示例表格就会拥有带图标的完整排序功能，截图如图 4-4 所示。

Name ▲	Description	Priority	Estimation	Done?
Incorporate reviewers remarks	Go over and reviewers remarks	6	3	false
Prepare outline	Prepare book outline with estimated page count	2	2	true
Prepare samples	Think of code samples	3	5	true
Review draft	Re-read and review the 1st draft	4	6	false
Total 4				

图4-4

`orderBy` 过滤器很容易使用，所以我们不会花很多时间在理论介绍上，会立刻深入代码示例。首先要让排序工作，然后再加上排序指示器。下面是参与排序的相关标记部分：

```
<thead>
  <th ng-click="sort('name')">Name</th>
  <th ng-click="sort('desc')">Description</th>
  ...
</thead>
<tbody>
  <tr ng-repeat="item in filteredBacklog = (backlog |
    filter:criteria | orderBy:sortField:reverse)">
    <td>{{item.name}}</td>
    <td>{{item.desc}}</td>
    ...
  </tr>
</tbody>
```

实际的排序由 `orderBy` 过滤器接管，在这个例子中它接受两个参数。


- `sortField`：用于排序依据的属性名。
- 排序顺序（`reverse`）：该参数指示排序数组是否应反转顺序。

`sort` 函数由表格头部的点击触发，负责选择排序依据的字段，以及反转排序的方向。下面是相关的控制器代码：

```
$scope.sortField = undefined;
$scope.reverse = false;

$scope.sort = function (fieldName) {
  if ($scope.sortField === fieldName) {
    $scope.reverse = !$scope.reverse;
  } else {
    $scope.sortField = fieldName;
    $scope.reverse = false;
  }
};
```

上面构建了排序示例，因此，现在待办列表能够进行过滤和排序了。运用 AngularJS，混合所有这些过滤器来创建交互式表格，容易让人惊喜。

 orderBy过滤器被故意放在filter过滤器之后。理由是性能：相比过滤，排序要付出更大代价，因此，在越小的数据集上运用排序算法越好。

排序可以工作了，现在我们只需要增加图标来指示排序字段，以及指示是升序还是降序。再次，ng-class 指令将大显身手，下面是对应 name 列的视觉指示器示例：

```
<th ng-click="sort('name')">Name
< i ng-class="{ 'icon-chevron-up': isSortUp('name'), 'icon-
  chevron-down': isSortDown('name') } "></i>
</th>
```

isSortUp 和 isSortDown 函数非常简单，代码如下：

```
$scope.isSortUp = function (fieldName) {
  return $scope.sortField === fieldName && !$scope.reverse;
};

$scope.isSortDown = function (fieldName) {
  return $scope.sortField === fieldName && $scope.reverse;
};
```

当然，有很多显示排序指示器的方法。上面的方法只是尽量让 CSS 类远离 JavaScript 代码，这样，只要调整模板就可以轻易改变表现样式。

编写自定义过滤器——分页示例

Writing custom filters –a pagination example

到目前为止，我们成功做到了使用支持排序和过滤的动态表格显示待办条目。下一种用于大数据集的 UI 模式则是分页。

AngularJS 并未提供任何能帮助我们基于起始和结束位置精确选择数组子集的过滤器。为了支持分页，我们要创建新过滤器，这是个熟悉如何编写自定义过滤器的好机会。

让我们认识一下新过滤器——`pagination` 的界面吧！首先，写出标记的梗概：

```
<tr ng-repeat="item in filteredBacklog = (backlog |
  pagination:pageNo:pageSize)">
  <td>{{item.name}}</td>
  ...
</tr>
```

新的 `pagination` 过滤器要接受两个参数：将显示的页（它的索引）以及此页的大小（每页的条目数量）。

下面是非常基础的对此过滤器的实现（为了专注于过滤器编写机制，错误处理被有意忽略了）。

```
angular.module('arrayFilters', [])

.filter('pagination', function(){

  return function(inputArray, selectedPage, pageSize) {
    var start = selectedPage*pageSize;
    return inputArray.slice(start, start + pageSize);
  };
});
```

过滤器和其他提供者（`provider`）一样，需要在模块（`module`）的实例上注册。`filter` 方法会以过滤器名称和一个工厂函数作为参数调用，后者将创建新过滤器的一个实例，已注册的工厂函数必须返回实际的过滤器对象。

`pagination` 过滤函数的第一个参数代表将被过滤的输入项，后续参数则声明用来支持过滤器的可选项。

过滤器非常容易进行单元测试，它们在提供的输入参数上工作，当工作完成时也没有任何副作用。下面是自定义的 `pagination` 过滤器的测试示例：

```
describe('pagination filter', function () {

  var paginationFilter;
  beforeEach(module('arrayFilters'));
  beforeEach(inject(function (_paginationFilter_) {
    paginationFilter = _paginationFilter_;
  }));

  it('should return a slice of the input array', function () {
```

```

    var input = [1, 2, 3, 4, 5, 6];

    expect(paginationFilter(input, 0, 2)).toEqual([1, 2]);
    expect(paginationFilter(input, 2, 2)).toEqual([5, 6]);
  });

  it('should return empty array for out-of bounds', function () {

    var input = [1, 2];
    expect(paginationFilter(input, 2, 2)).toEqual([]);
  });
});

```

测试过滤器如同测试函数一样简单，大部分时候都非常直接。刚才展现的测试示例的结构应该很容易理解，因为没什么新东西。唯一需要解释的就是，从 JavaScript 代码中访问过滤器实例的方法。

从 JavaScript 代码中访问过滤器

Accessing filters from the JavaScript code

过滤器通常在标记中调用（使用表达式中的管道符），但也可以从 JavaScript 代码（控制器、服务、其他过滤器等）中访问过滤器实例。这样，我们就能混合现有的过滤器以提供新功能。

过滤器能够被注入由 AngularJS 的依赖注入（dependency injection）系统管理的任何对象中。有两种不同方法说明对过滤器的依赖关系：

- `$filter` 服务。
- 以 `Filter` 为后缀的过滤器名称。

`$filter` 服务是一个查找函数，它能根据名字获得过滤器的实例。为了查看它在实践中的应用，让我们写一个行为类似于 `limitTo` 的过滤器，它能修剪（trim）长字符串，而且这个自定义版本还会在字符串被修剪后为其增加“...”后缀。以下为相关代码：

```

angular.module('trimFilter', [])
  .filter('trim', function($filter){

    var limitToFilter = $filter('limitTo');

    return function(input, limit) {
      if (input.length > limit) {

```

```

        return limitToFilter(input, limit-3) + '...';
    }
    return input;
};
});

```

`$filter('limitTo')` 函数调用能让我们根据过滤器的名字获取其实例。

虽然上面的方法行得通，但还有一种替代方法更容易读写：

```

.filter('trim', function(limitToFilter){

    return function(input, limit) {
        if (input.length > limit) {
            return limitToFilter(input, limit-3) + '...';
        }
        return input;
    };
});

```

第二个例子中，通过名称 `[filter name]Filter` 声明了依赖关系，其中 `[filter name]` 是我们想获取的过滤器名字。



利用 `$filter` 服务访问过滤器实例会导致古怪的语法，这是我们认为 `Filter` 后缀形式更好用的原因。`$filter` 服务的用武之地是在同一个地方获取多个过滤器实例，或者基于变量获取过滤器实例，如 `$filter(filterName)`。

过滤器做什么与不做什么 Filters dos and don'ts

过滤器提供了优美简洁的语法，非常方便对模板中的数据进行格式化和转化。但是，过滤器只是完成特定任务的工具，它和其他工具一样，一旦滥用就会造成损害。本节介绍了应避免使用过滤器的情况，此时另外一些替代方案往往是更好的选择。

过滤器与DOM操作

从过滤器的执行结果返回 HTML 标记，也许有点诱惑力。实际上，AngularJS 正好包含一个完成此工作的过滤器：linky（在分离出的 ngSanitize 模块中）。

但实践证明，输出 HTML 的过滤器并不是一个好主意。

主要的问题是，当渲染这类过滤器的输出时，我们需要运用此前介绍过的绑定指令——ngBindUnsafeHtml 或 ngBindHtml。这不仅让绑定语法更冗长（与简单的 {{expression}} 相比），还会让 web 页面受到 HTML 注入攻击的潜在威胁。

为了查看输出 HTML 过滤器的一些问题，下面来检查简单的 highlight 过滤器：

```
angular.module('highlight', [])

.filter('highlight', function() {

  function(input, search) {
    if (search) {
      return input.replace(new RegExp(search, 'gi'),
        '<strong>$&</strong>');
    } else {
      return input;
    }
  };

});
```

你能立刻看到此过滤器包含硬编码进去的 HTML 标记，这导致我们不能使用插值指令，而要写这样的模板：

```
<input ng-model="search">
<span ng-bind-html="phrase | highlight:search"></span>
```

过滤器输出的 HTML 标记不能包含任何 AngularJS 指令，因为它们不会被求值。

大多数时候，自定义指令能够以更优雅的方式解决此问题，而不带来潜在的安全危机。指令将在第 9 章（创建自定义指令）和第 10 章（创建为全球用户服务的 AngularJS 应用）中介绍。

过滤器中代价高昂的数据变换

在模板中应用的过滤器，成为 AngularJS 表达式的一部分，因此会被频繁求值。实际上，这样的过滤器函数会在每个消化周期中被调用多次。我们将围绕 `uppercase` 过滤器创建一个日志包装器（logging wrapper），代码如下：

```
angular.module('filtersPerf', [])
  .filter('logUppercase', function(uppercaseFilter){
    return function(input) {
      console.log('Calling uppercase on: '+input);
      return uppercaseFilter(input);
    };
  });
```

在标记中使用这个新定义的过滤器：

```
<input ng-model="name"> {{name | logUppercase}}
```

我们将看到，在每次击键时至少记录一次（通常两次）日志语句。这个实验应该能说服你，过滤器被执行得非常频繁，所以它们最好能执行得快一些。



当看到过滤器在同一行中被调用多次时，不要感到惊讶，这就是 AngularJS 的脏检查（dirty checking）在工作。努力编写轻量化、执行快速的过滤器吧！

不稳定的过滤器

过滤器会被调用多次，如果输入不变，过滤器的返回值也不变，我们就称这样的函数是随参数稳定的。

如果过滤器不具备这一属性，那么事情很快会失去控制。为了查看不稳定过滤器造成的灾难性后果，让我们写一个充满恶意的 `random` 过滤器，它从输入数组中随机选择一个元素（不稳定的）：

```
angular.module('filtersStability', [])
  .filter('random', function () {
    return function (inputArray) {
      var idx = Math.floor(Math.random() * inputArray.length);
      return inputArray[idx];
    };
  });
```

给定一个包含不同项的数组，它存储在作用域上的 `items` 变量中。在模板上应用 `random` 过滤器如下：

```
{{items | random}}
```

如果一切正常，那么上述代码在执行时应该打印出随机的值。但实际在浏览器的控制台上，我们发现记录的是一个错误信息：

```
Uncaught Error: 10 $digest() iterations reached. Aborting!
```

此错误意味着一个表达式在每次求值时都产生了不同的结果。**AngularJS** 观察到一种不断变化的模型，并重新对表达式求值，希望它能够稳定化。在重复此过程 10 次后，消化（`digest`）过程被放弃了，打印出最后的结果，并在控制台上记录此错误。第 11 章（开发健壮的 **AngularJS** 应用）将对这些主题做更深入的探讨，并解释 **AngularJS** 内部的工作机制，以更容易理解此类错误产生的原因。

这种状况的解决方案是，在控制器中计算随机值，并在渲染模板之前：

```
.controller('RandomCtrl', function ($scope) {

    $scope.items = new Array(1000);
    for (var i=0; i<$scope.items.length; i++) {
        $scope.items[i] = i;
    }

    $scope.randomValue = Math.floor(Math.random() * $scope.items.length);
});
```

这样，随机值会在模板处理过程前计算，我们就能安全地使用 `{{randomValue}}` 表达式输出准备好的值。

对于同样的输入，如果函数会产生不同的结果，那么使用过滤器就不是一个好的选择。从控制器中调用这个函数，让 **AngularJS** 渲染预先计算好的值。

4.8 摘要

Summary

本章带领我们学习了一组模式，用于显示模型中包含的数据。

我们先从快速梳理指令的命名惯例开始，然后概览 **AngularJS** 的内置指令，特别是 `ng-repeat`，因为它非常强大，而且是最常用的指令之一。

AngularJS 的基于 **DOM** 的声明式模板，大部分时间都工作得很好，但有时也会有限制。确认这些状况很重要，我们需要准备好在必要时对标记做微妙的改变。

过滤器为 **UI** 特定的模型格式化提供了非常便利的语法。我们知道 **AngularJS** 默认提供了几种有用的过滤器，而且当有特殊需要时，也很容易创建自定义过滤器。过滤函数不应被滥用，我们仔细观察了应该考虑替代结构的一些场景。

本章涵盖的指令——过滤器和显示模式都专注于显示数据。但是，在数据被显示之前，用户应该能使用多种输入元素输入它们。第 5 章将深入 **AngularJS** 对表单元素和数据输入问题的介绍。

第 5 章

创建高级表单

Creating Advanced Forms

AngularJS 构建于标准 HTML 表单和输入元素之上，这意味着你能继续使用标准 HTML 设计工具，通过已经熟悉的 HTML 元素来创建 UI。

至目前为止，我们已经为 SCRUM 应用创建了一些基本表单，其中的输入元素绑定了模型数据，以及保存删除等的按钮。AngularJS 会肩负起将元素 - 模型绑定 (element-model binding) 和事件处理器绑定 (event-handler binding) 连接在一起的责任。

在本章中，我们将介绍 AngularJS 表单的工作细节，并为应用中的表单加上验证 (validation) 功能，以及动态的用户交互。

本章将会涵盖以下内容。

- 模型数据绑定及输入指令 (input directives)。
- 表单验证。
- 嵌套 (nested) 与重复的表单。
- 表单提交。
- 表单重置。

5.1 AngularJS 表单与传统表单的比较

Comparing traditional forms with AngularJS forms

在动手改进应用的表单之前，应该先理解 AngularJS 的表单如何工作。在本节中，我们会解释标准 HTML 输入元素和 AngularJS 输入指令 (input directive) 之间的区别，将展示 AngularJS 是如何修改和扩展 HTML 输入元素的行为的，以及 AngularJS 如何管理这些输入元素和模型的同步更新。

在标准 HTML 表单中，输入元素的值会在表单提交时送往服务器，如图 5-1 所示。

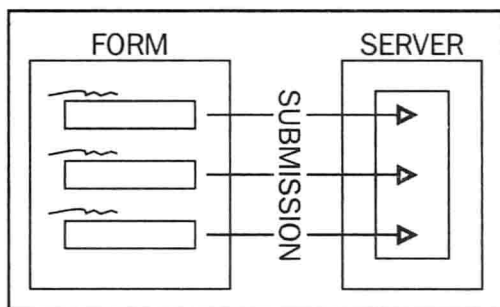


图5-1

用输入元素保存你提交的值，麻烦在于你会被用户输入的值卡住。例如，日期输入域允许用户输入遵循预定义格式的字符串，如“12 March 2013”。但在你的代码中，也许值是一个 JavaScript Date 对象会更好。通常，编写这类转换代码是枯燥乏味和错误层出的。

AngularJS 为模型与视图解耦（decouple），让输入指令关心如何显示输入值的问题，让 AngularJS 去关心如何根据输入值的变化更新模型。这让我们可以自由地利用模型工作，比如通过控制器，而不用担心数据是如何被输入和显示的，如图 5-2 所示。

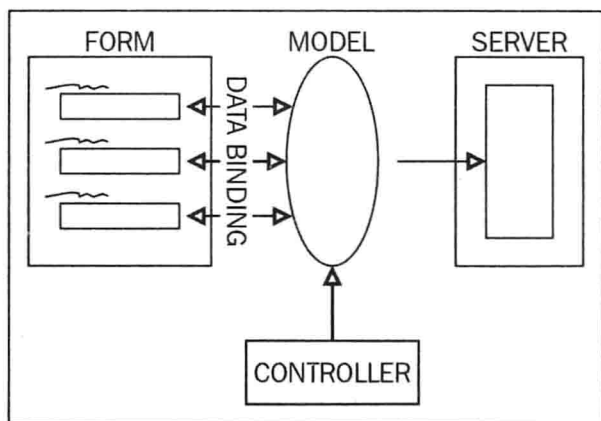


图5-2

为了达到这种分离效果，AngularJS 利用 form 和 input 指令来验证指令和控制器，以增强 HTML 表单效果。这些指令和控制器覆盖了 HTML 表单的内置行为，但是，对于漫不经心的观察者来说，AngularJS 的表单看起来很像标准的 HTML 表单。

首先，ngModel 指令定义了输入是如何与模型绑定的。

介绍 ngModel 指令

Introducing the ngModel directive

我们已经看到，AngularJS 是如何在 scope 对象的字段和页面的 HTML 元素之间建立数据绑定的。我们可以用花括号 {} 或用 ngBind 这样的指令设置数据绑定。但是，这样的绑定只是单向的。当为输入指令绑定值的时候，则使用 ngModel，代码如下：

```
<div>Hello <span ng-bind="name"/></div>
<div>Hello <input ng-model="name"/></div>
```

参见 <http://bit.ly/Zm55zM>。

在第一个 div 中，AngularJS 将当前作用域下的 scope.name 绑定给了 span 的文本。这种绑定关系是单向的：如果 scope.name 的值变化了，那么 span 的文本也会随之变化；但如果改变 span 的文本，那么 scope.name 的值不会随之改变。

在第二个 div 中，AngularJS 将当前作用域下的 scope.name 绑定给了 input 元素的值。这里的数据绑定就是双向的，因为如果在输入框中修改输入值，那么 scope.name 模型的值会立刻被更新，继而此更新也会在 span 的单向绑定中可见，如图 5-3 所示。

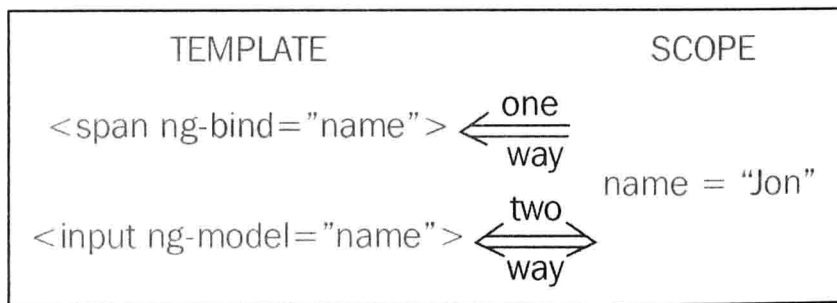


图5-3



为什么由不同的指令来指定与输入的绑定关系？ngBind指令仅将其表达式的值绑定（单向）到元素的文本上，而有了ngModel，数据绑定成为双向的，输入值的变化会反射到模型上。

在模型和输入指令之间进行同步数据绑定的时候，AngularJS 还允许指令变换和验证 ngModel 指令的值。这会在 ngModelController 一节中介绍。

5.2 创建用户信息表单

Creating a User Information Form

本节中将介绍示例 SCRUM 应用中的简单用户信息表单。贯穿本章，我们将渐进式地为此表单增加功能，以展示 AngularJS 表单的威力。基本表单如下：

```
<h1>User Info</h1>
<label>E-mail</label>
<input type="email" ng-model="user.email">

<label>Last name</label>
<input type="text" ng-model="user.lastName">

<label>First name</label>
<input type="text" ng-model="user.firstName">

<label>Website</label>
<input type="url" ng-model="user.website">

<label>Description</label>
<textarea ng-model="user.description"></textarea>

<label>Password</label>
<input type="password" ng-model="user.password">

<label>Password (repeat)</label>
<input type="password" ng-model="repeatPassword">

<label>Roles</label>
<label class="checkbox">
<input type="checkbox" ng-model="user.admin"> Is Administrator
</label>

<pre ng-bind="user | json"></pre>
```

参见 <http://bit.ly/10ZomqS>。

看起来我们只是有了一个标准 HTML 输入的列表，但这些实际上是 AngularJS 的输入指令。每个 input 都被赋予一个 ngModel 指令，由该指令定义 input 元素的值绑定给当前作用域上的什么对象。此例中，每个 input 都被绑定给当前作用域下的 user 对象上的一个字段。在控制器中可以如下记录此模型字段的值：

```
$log($scope.user.firstName);
```


请注意，我们并没有使用 `form` 元素，也没有为任何 `input` 元素设置 `name` 或 `id` 属性。对于没有验证的简单表单，AngularJS 会确保输入元素的值和模型中的值保持同步。然后就可以在控制器中自由应用用户模型，而无需担心如何在视图中表现它们。



我们也为用户模型的JSON呈现绑定了 `pre` 元素，这样就能看到AngularJS对模型同步的数据。

5.3 理解输入指令

Understanding the input directives

在本节中，我们会描述 AngularJS 提供的输入指令。对于熟悉 HTML 表单的人来说，运用输入指令非常自然，因为 AngularJS 就构建在 HTML 之上。

在表单中，可以运用所有的标准 HTML 输入类型。输入指令与 `ngModel` 协同工作，可提供诸如验证或模型绑定这样的附加功能。AngularJS `input` 指令用于检查 `type` 属性以辨认何种功能要被加到输入元素上。

添加所需验证

Adding the required validation

所有基本输入指令都支持 `required`（或 `ngRequired`）属性。为输入元素添加这一属性，AngularJS 就知道，`ngModel` 的值不能是 `null`、`undefined` 或 `""`（空字符串）。下面关于字段校验的段落会介绍更多细节。

使用基于文本的输入（`text`、`textarea`、`e-mail`、`URL`、`number`）

Using text-based inputs (`text`, `textarea`, `e-mail`, `URL`, `number`)

基本输入指令，`type="text"` 或 `textarea`，接受任何字符串作为值。当改变输入的文本时，模型会立即更新为对应值。

其他基于文本的输入指令，如 `e-mail`、`URL` 或 `number`，有类似的行为，但当这些指令在输入框中的值仅匹配对应的正则表达式时，才会允许模型更新。如果要在 `e-mail` 输入中键入内容，除非输入框中的文本包含有效的 `e-mail` 字符串，那么模型中的 `e-mail` 字段会一直为空。这意味着，模型将永远不会包含无效的 `e-mail` 地址，这是模型与视图解耦的一大好处。

除了以上验证外，所有基于文本的指令都允许指定文本的最小和最大长度，以及指定文本必须匹配的正则表达式。可通过 `ngMinLength`、`ngMaxLength`、`ngPattern` 指令实现，代码如下：

```
<input type="password" ng-model="user.password"
      ng-minlength="3" ng-maxlength="10"
      ng-pattern="/^.*(?:=.*\d) (?:=.*[a-zA-Z]).*$/">
```

参见 <http://bit.ly/153L87Q>。

这里 `user.password` 模型字段的长度必须在 3 到 10 个字符之间（含 3 或 10），而且必须匹配指定的正则表达式，以保证它包含至少一个字母和一个数字。



请注意，内置的验证特性不能阻止用户输入无效的字符串，输入指令只在字符串无效时清除模型上的字段。

使用 checkbox 输入 Using checkbox inputs

`checkbox` 表示布尔输入。在表单中，输入指令为 `ngModel` 指定的模型字段赋 `true` 或 `false`。你可以在用户信息表单的“是否管理员”字段中查看到。

```
<input type="checkbox" ng-model="user.admin">
```

当选中 `checkbox` 时，`user.admin` 的值设为 `true`，否则设为 `false`。相反，如果 `user.admin` 的值设为 `true`，那么 `checkbox` 也会被勾选。

你也可以为 `true` 和 `false` 值指定不同的用在模型中的字符串，比如，可以在 `role` 字段中使用 `admin` 和 `basic`。

```
<input type="checkbox" ng-model="user.role" ng-true-value="admin" ng-
      false-value="basic">
```

参见 <http://bit.ly/Yidt37>。

此时，`user.role` 模型会根据 `checkbox` 是否被勾选选择是包含 `admin` 还是包含 `basic`。

使用 radio 输入

Using radio inputs

radio 按钮可为字段提供一组固定的选择。AngularJs 的实现非常简单：只要绑定一组内的所有 radio 按钮给同一模型字段，标准的 HTML value 属性就用于指定当 radio 被选中时赋予模型字段的值：

```
<label><input type="radio" ng-model="user.sex" value="male">
  Male</label>
<label><input type="radio" ng-model="user.sex" value="female">
  Female</label>
```

参见 <http://bit.ly/14hYNsN>。

使用 select 输入

Using select inputs

select 输入指令允许创建下拉（drop-down）列表，用户可从中选取一个或多个条目。AngularJS 可以为下拉列表中的条目指定静态 options，或者指定作用域上的数组。

提供简单的字符串options

如果要从静态的 options 列表中进行选择，则可以单纯地将它们作为 select 元素下的 option 元素提供出来：

```
<select ng-model="sex">
  <option value="m" ng-selected="sex=='m'">Male</option>
  <option value="f" ng-selected="sex=='f'">Female</option>
</select>
```

注意，因为 value 属性只能接收字符串，所以它绑定的值也只能是一个字符串。



如果想绑定非字符串值，或者想要从数据中动态地创建 options 列表，则可以使用 ngOptions 选项。



利用ngOptions指令提供动态options

为了动态地定义含有复杂 options 列表的 select 指令，AngularJS 提供了额外的语法。如果想绑定 select 指令的值给一个对象，而不是一个简单的字符串，那么要用 ngOptions。此属性接受理解表达式（comprehension expression），它定义了会被显示的 options。此类表达式的形式如图 5-4 所示。

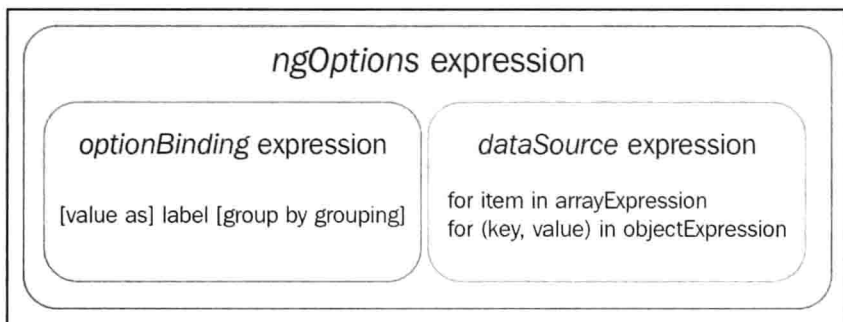


图5-4

dataSource expression 描述了将要显示的 options 的信息来源，即数组中的元素或对象的属性。对 dataSource expression 中的每一个条目，都会生成一个 option。

optionBinding 表达式描述了可以从每个数据源条目中解析出什么内容，以及这个条目如何与 option 绑定。

1. ngOptions的典型示例

在解释怎样定义这些理解表达式之前，先来看一些典型的例子。

使用数组数据源

使用 user.email 作为 label 来选择用户对象：

```
ng-options="user.email for user in users"
```

使用计算过的 label（此函数定义在作用域上）来选择用户对象：

```
ng-options="getFullName(user) for user in users"
```

选择用户的 e-mail 而非完整的用户对象，将用户的全名作为 label：

```
ng-options="user.email as getFullName(user) for user in users"
```

利用性别将列表进行分组来选择用户对象：

```
ng-options="getFullName(user) group by user.sex for user in users"
```

参见 <http://bit.ly/1157jqa>。

使用对象数据源

我们提供两个将国家名字与代码相关联的对象，代码如下：

```
$scope.countriesByCode = {
  'AF' : 'AFGHANISTAN',
  'AX' : 'ÅLAND ISLANDS',
  ...
};

$scope.countriesByName = {
  'AFGHANISTAN' : 'AF',
  'ÅLAND ISLANDS' : 'AX',
  ...
};
```

通过国家名字选择国家代码，并按照国家代码排序：

```
ng-options="code as name for (code, name) in countriesByCode"
```

按照国家名字排序：

```
ng-options="code as name for (name, code) in countriesByName"
```

参见 <http://bit.ly/153LKdE>。

现在，我们已经查看了一些能够展示这些表达式的完整规则的例子。

2. 理解dataSource表达式


如果数据源是数组，那么 `arrayExpression` 会用于此数组。这个指令会迭代数组中的每一个条目，同时将数组中的当前条目赋值给 `value` 变量。



此时，`select` 列表的 `options` 显示顺序与数组中条目的顺序相同。



如果数据源是对象，那么 `objectExpression` 会用于此对象。这个指令迭代对象的每一个属性，并将属性的值赋给 `value` 变量，将属性的键赋给 `key` 变量。

 此时, select列表显示的options会按照对应key的字母顺序来排序。

3. 理解optionBinding表达式


optionBinding 表达式定义了如何获取每个 option 的 label 和值, 以及如何将由 dataSource 表达式提供的条目生成的 options 分组。此表达式可以利用所有的 AngularJS 表达式语法, 包括过滤器。通用语法如下:

```
value as label group by grouping
```

如果没能提供 value 表达式, 则在 option 被选中时, 其来源的数据条目自身会被用做赋给模型的值。如果提供了分组表达式, 则直接使用组名作为赋给模型的值。

select指令与空的options

当绑定的模型值没有匹配任何 option 列表中的值时, select 指令会在 options 列表的顶端显示一个空的 option。


 当模型没有匹配任何options时, 总会显示空的option。如果用户手动选择了空的option, 则模型值会被设为null, 而不是undefined。

可以通过为 select 元素添加一个 option 子元素来定义空的 option, 它的值是一个空字符串, 代码如下:

```
<select ng-model="..." ng-options="...">
  cc<option value="">-- No Selection --</option>
</select>
```

参见 <http://bit.ly/ZeNpZX>。

这样, 就定义好了显示 label 为 No Selection 的空 option。

 如果定义了自己的空option, 那么它会一直被显示在options列表中, 并能够被用户选择。

如果没有在 select 指令的声明中定义空 option, 则指令会生成它自己的空 option。



指令生成的空 **option**，仅在模型没有匹配列表中的任何条目时才会显示。所以，用户无法手动将 **select** 的值设为 **null/undefined**。

要隐藏空 **option**，可以定义自己的空 **option**，并将其样式设为 `display: none`。

```
<option style="display:none" value=""></option>
```

参见 <http://bit.ly/ZeNpZX>。

此时 **select** 指令会使用我们定义的空 **option**，但浏览器不会显示它。现在，如果模型不匹配任何 **options**，则 **select** 指令会为空且无效，虽然列表中并没有显示出空的 **option**。

理解select和对象判等

select 指令借助对象判等操作符 (`===`) 将对象的值与 **option** 的值进行匹配。这意味着如果 **option** 的值是对象而非简单值（比如数字和字符串），那么你必须使用指向实际 **option** 值的引用来与模型值比较，否则 **select** 指令会认为对象不同而无法匹配。

可以在控制器中通过对象数组设置 **options** 和被选中的条目：

```
app.controller('MainCtrl', function($scope) {
  $scope.sourceList = [
    {'id': '10005', 'name': "Anne"},
    {'id': '10006', 'name': "Brian"},
    {'id': '10007', 'name': "Charlie"}
  ];
  $scope.selectedItemExact = $scope.sourceList[0];
  $scope.selectedItemSimilar = {'id': '10005', 'name': "Anne"};
});
```

这里，`selectedItemExact` 实际上引用了 `sourceList` 的第一个条目。而 `selectedItemSimilar` 是不同的对象，尽管字段是相同的：


```
<select
  ng-model="selectedItemExact"
  ng-options="item.name for item in sourceList">
</select>
<select
  ng-model="selectedItemSimilar"
  ng-options="item.name for item in sourceList">
</select>
```

参见 <http://bit.ly/Zrachk>。

这里创建了两个绑定给这些值的 `select` 指令。绑定给 `selectedItemSimilar` 的并不会有 `option` 被选中。因此，应当一直绑定 `select` 的值为 `ng-options` 数组中的条目。也许不得不搜索数组来获得合适的 `option`。

选择多个options

如果想选择多个条目，只要为 `select` 指令应用 `multiple` 属性即可。绑定到此指令的 `ngModel` 就成为数组，包括指向每个被选择的 `option` 值的引用。

 AngularJS提供`ngMultiple`指令，该指令接收一个表达式，以决定是否允许多选。当前，`select`指令并不监控此变化（是否接收多选），所以`ngMultiple`指令的应用有限。

运用传统的 HTML hidden input 字段

Working with traditional HTML hidden input fields

在 AngularJS 中，我们在作用域上存储所有模型数据，且几乎没有对 `hidden input` 字段的需求。因此，AngularJS 没有 `hidden input` 指令。一般在两种情况下需要 `hidden input` 字段：嵌入来自服务器的值，提交传统的 HTML 表单。

嵌入来自服务器的值

如果使用服务器端的模板引擎来创建 HTML，并通过模板从服务器传递数据给 AngularJS，那么只要在 HTML 中加入由服务器生成的 `ng-init` 指令就行，此指令会在作用域上添加值：

```
<form ng-init=>user.hash='13513516'>>
```

这里，服务器发送的 HTML 包括一个 `form` 元素，其中包含 `ng-init` 指令，该指令会在 `form` 的作用域上初始化 `user` 和 `hash`。

提交传统的HTML表单

传统上, 也许需要向服务器提交不在模板上的值, 也就是非可见的输入组件, 这通常通过为表单增加 `hidden` 字段来解决。在 `AngularJS` 中, 工作的模型和表单是解耦的, 所以并不需要这些 `hidden` 字段。只需要单纯地将这些值添加到作用域上, 然后使用 `$http` 服务模拟表单提交就可以了, 详见第 3 章 (与后端服务器通信) 介绍的内容。

5.4 详解 ngModel 数据绑定

Looking inside ngModel data binding

我们已经看到, `ngModel` 创建了与 `input` 字段值之间的绑定关系的模型。在本节中, 我们将深入查看此指令是如何工作的, 以及它提供的其他东西。

理解 ngModelController

Understanding ngModelController

每个 `ngModel` 指令都会创建 `ngModelController` 的一个实例, 该控制器对 `input` 元素上的所有指令都有效, 如图 5-5 所示。

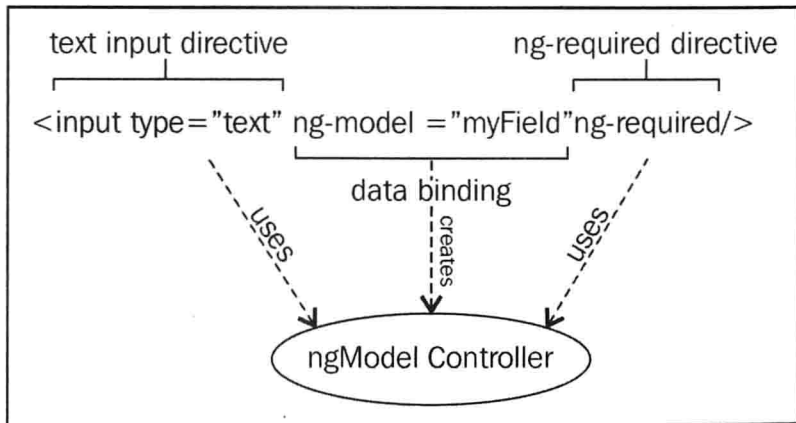


图5-5

`ngModelController` 负责管理存储在模型 (由 `ngModel` 指定) 中的值与 `input` 元素显示值之前的数据绑定。

`ngModelController` 也追踪视图上的值是否有效, 以及它是否已经被 `input` 元素修改。

在模型与视图之间转换值

当每次更新数据绑定的时候，ngModelController 都会应用它的转换管道。此管道由两个数组组成：\$formatters 处理从模型到视图的转换，\$parsers 则处理从视图到模型的转换。input 元素上的每个指令都可以添加它们自己的 formatters 和 parsers 到此管道，以定制数据绑定的行为，如图 5-6 所示。

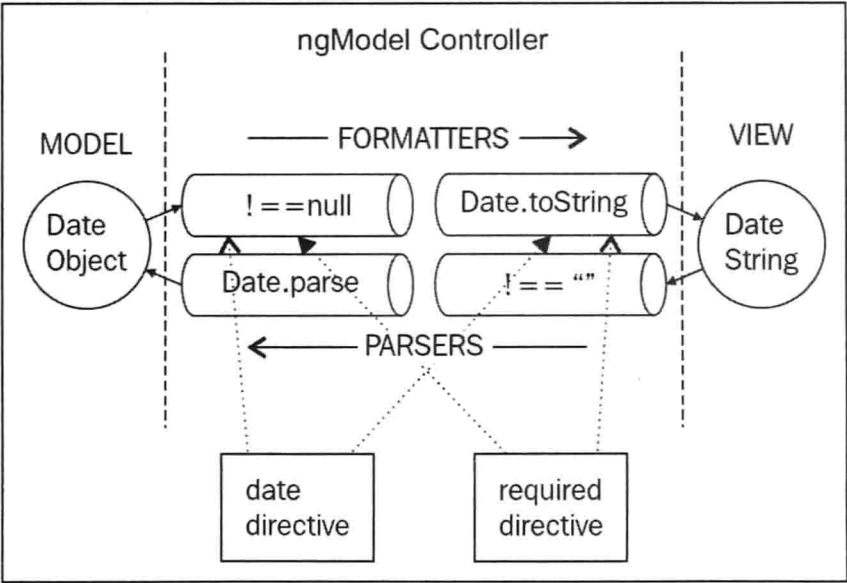


图5-6

这里有两个指令被添加进了转换管道，date 指令用于解析和格式化数据，ng-required 指令则负责检查值是否缺失。

追踪值是否变化

伴随模型与视图之间值的转换，ngModelController 追踪在初始化之后的值是否有变化及是否有效。

当首次初始化时，ngModelController 标记值为纯净的 (pristine)，也就是还没有被修改过。它通过输入元素上的 ng-pristine CSS 类来标示。当视图改变时，比如通过在输入框中键入值，值就会被标记为脏的 (dirty)，此时会用 ng-dirty CSS 类来替换 ng-pristine CSS 类。

通过为这些类提供 CSS 样式，就能基于用户是否输入或修改数据来改变输入元素的外观：

```
.ng-pristine { border: solid black 1px; }
.ng-dirty { border: solid black 3px; }
```

这里，在用户改变了输入值后，可以让元素的边变得更粗一些。

跟踪input字段有效性

input 元素上的指令也能将值的 valid 或 invalid 状态通知给 ngModelController，这通常通过加入转换管道的挂钩（hooking）来完成，它是检查值而不是转换。ngModelController 跟踪值是否有效，并相应地在其上应用 ng-valid 或 ng-invalid CSS 类。基于这些类，可以提供更进一步的样式来改变元素外观：

```
.ng-valid.ng-dirty { border: solid green 3px; }
.ng-invalid.ng-dirty { border: solid red 3px; }
```

这里，我们混合使用 pristine 和 invalid 以确保只有被用户改变过输入值的字段得到修饰：invalid 时是红色实线边框，valid 时则是绿色实线边框。

在下一节（校验 AngularJS 表单）中，我们将会看到如何与 pristine、dirty、valid 和 invalid 的概念用编程的方式一起工作。

5.5 校验 AngularJS 表单

Validating AngularJS forms

在本节中，我们将介绍如何使用验证指令，以及它们如何与 ngFormController 协同工作，以提供功能完善的验证框架。

理解 ngFormController

Understanding ngFormController

每个 form（或 ngForm）指令都创建 ngFormController 的一个实例。ngFormController 对象管理表单的 valid 或 invalid、pristine 或 dirty 状态，重点是它与 ngModelController 协同工作以跟踪表单中的每个 ngModel 字段。

当 ngModelController 被创建时，它将自己注册给其父元素列表中最近的 ngFormController。这样，ngFormController 就了解了它应该追踪哪些输入指令，它可以检查这些字段是否为 valid/invalid 或 pristine/dirty，并相应地将表单设置为 valid/invalid 或 pristine/dirty 状态。

运用name属性将表单附加到作用域上

可以给表单命名，以使 `ngFormController` 出现在本地作用域上。表单中任何有名字的输入元素也会有它们的 `ngModelController` 对象，其作为属性会被附加到 `ngFormController` 对象。

表 5-1 展示了包含控制器的作用域是如何与表单中的每个元素发生联系的。

表5-1

HTML	Scope	Controller
<code><form name="form1"></code>	<code>model1, model2, ...</code>	
	<code>form1 : {</code>	<code>ngFormController</code>
	<code> \$valid, \$invalid,</code>	
	<code> \$pristine, \$dirty,</code>	
	<code> ...</code>	
	<code> field1: {</code>	
<code> <input</code>	<code> \$valid, \$invalid,</code>	
<code> name="field1"</code>	<code> \$pristine,</code>	<code>ngModelController</code>
<code> ng-model="model1"</code>	<code> \$dirty, ...</code>	
<code> /></code>	<code> },</code>	
<code> <input</code>	<code> field2: {</code>	
<code> name="field2"</code>	<code> \$valid, \$invalid,</code>	
<code> ng-model="model2"</code>	<code> \$pristine,</code>	
<code> /></code>	<code> \$dirty, ...</code>	<code>ngModelController</code>
<code></form></code>	<code> } },</code>	

为用户信息表单增加动态行为
Adding dynamic behavior to the user information form

虽然表单允许根据用户在字段中输入的值来修改输入元素的外观，但是，为了响应更好的用户体验，我们还想要展示和隐藏验证信息，并根据表单字段的状态，相应改变表单中按钮的状态。

作用域上的 `ngFormController` 和 `ngModelController` 对象可以用编程的方式与表单状态协作，使用像 `$invalid` 和 `$dirty` 这样的值来改变用户可见信息与可操作的组件。

显示验证错误

当某些输入无效的时候，能够为输入和整个表单显示错误信息，模板如下：

```
<form name="userInfoForm">
  <div class=»control-group«
    ng-class="getCssClasses(userInfoForm.email)">

    <label>E-mail</label>
    <input type="email" ng-model="user.email"
      name="email" required>

    <span ng-show=»showError(userInfoForm.email, 'email')" ...>
      You must enter a valid email
    </span>

    <span ng-show="showError(userInfoForm.email, 'required')" ...>
      This field is required
    </span>
  </div>
  ...
</form>
```

控制器如下：

```
app.controller('MainCtrl', function($scope) {
  $scope.getCssClasses = function(ngModelController) {
    return {
      error: ngModelController.$invalid && ngModelController.$dirty,
      success: ngModelController.$valid && ngModelController.$dirty
    };
  };
  $scope.showError = function(ngModelController, error) {
    return ngModelController.$error[error];
  };
});
```

参见 <http://bit.ly/XwLUFZ>。

上例展示了用户表单中的 e-mail 输入。因为使用 Twitter Bootstrap CSS 来修饰表单，所以有 control-group 和 inline-help CSS 类。同时，也在控制器中创建了两个帮助函数。

`ng-class` 指令将更新包含 `label`、输入和帮助文本的 `div` 上的 `CSS` 类。此指令用于调用 `getCssClasses()` 方法，并传送对象和错误名作为参数。



对象参数实际上是 `ngModelController`，它被暴露给了 `ngFormController`。反过来，后者也被暴露在了 `scope.userInfoForm.email` 作用域上。

`getCssClasses()` 方法返回一个对象，此对象定义了应该添加哪些 `CSS` 类。每个对象的键都是某个 `CSS` 类的名字，而每个对象的值当 `CSS` 类被添加时为 `true`。此时，当模型是 `dirty` 和 `invalid` 时，`getCssClasses()` 会返回 `error`；而当模型是 `dirty` 和 `valid` 时，则会返回 `success`。

让保存按钮无效

当表单的状态不应该被保存时，则可以让保存按钮无效。

```
<form name="userInfoForm">
  ...
  <button ng-disabled="!canSave()">Save</button>
</form>
```

在视图中，我们添加了有 `ngDisabled` 指令的保存按钮，此指令在当其表达式求值为真的时候会令按钮无效，上例中就是对调用 `canSave()` 方法的结果求反。在当前作用域上我们提供了 `canSave()` 方法，在主要控制器中代码如下：

```
app.controller('MainCtrl', function($scope) {
  $scope.canSave = function() {
    return $scope.userInfoForm.$dirty &&
      $scope.userInfoForm.$valid;
  };
});
```

参见 <http://bit.ly/123zIhw>。

`canSave()` 方法用于检查 `userInfoForm` 是否有 `$dirty` 和 `$valid` 标记，如果有，则说明表单可以被保存。

使原生浏览器校验无效

Disabling native browser validation

浏览器通常会在表单提交的时候校验表单中的输入值。比如，你在 input 框上有一个 required 属性，如果在此字段没有值的时候试图提交表单，那么浏览器就会抱怨，而这与 AngularJS 无关。

由于是通过 AngularJS 的指令和控制器提供所有校验，所以不再希望浏览器尝试自己的原生验证。为了关掉它，可以对表单元素应用 HTML5 的 novalidate 属性。

```
<form name="novalidateForm" novalidate>
```

参见 <http://bit.ly/1l10hS4>。

这个名为 novalidateForm 的表单和 novalidate 属性将会告知浏览器，不要试图在此表单的任何输入上尝试验证。

5.6 在其他表单中嵌套表单

Nesting forms in other forms

不像标准 HTML 表单，AngularJS 表单可以相互嵌套 (nested)。因为在 form 标签中再加入 form 标签是无效的 HTML，所以 AngularJS 为嵌套表单提供了 ngForm 指令。



每个提供了名字的表单都会被添加入它们的父表单，如果没有父表单，则会直接被加到作用域上。

将子表单作为可重用组件

Using subforms as reusable components

嵌套表单的行为像复合字段，它基于包含的字段暴露自己的验证信息，这样的表单可作为子表单复用在更大的容器表单中。下面将两个输入框归为一组，以创建密码和密码确认组件：

```
<script type="text/ng-template" id="password-form">
  <ng-form name="passwordForm">
    <div ng-show="user.password != user.password2">
      Passwords do not match
    </div>
    <label>Password</label>
    <input ng-model="user.password" type="password" required>
```

```

        <label>Confirm Password</label>
        <input ng-model="user.password2" type="password" required>
    </ng-form>
</script>

<form name="form1" novalidate>
    <legend>User Form</legend>
    <label>Name</label>
    <input ng-model="user.name" required>
    <ng-include src="'password-form'"></ng-include>
</form>

```

参见 <http://bit.ly/10QWwyu>。

我们在子模板中定义了子表单，此例它存放在同文件的 `script` 块中，当然也可以放在另外的文件里。接下来，容器表单、`form1`，使用 `ngInclude` 指令包含子表单。

子表单拥有自己的验证状态及相关的 CSS 类，而且，请注意因为子表单有 `name` 属性，所以它会以容器表单的所有物形式（property）出现。

5.7 重复子表单

Repeating subforms

有时，基于模型中的数据，在表单中有些字段需要一定次数的重复，当想要在单个表单中显示一对多（one-to-many）的数据关系时，这种情况相当普遍。

在 SCRUM 应用中，允许用户在个人资料中拥有 0 个到多个网站 URL，可以使用 `ngRepeat` 指令来设置它：

```

<form ng-controller="MainCtrl">
    <h1>User Info</h1>
    <label>Websites</label>
    <div ng-repeat="website in user.websites">
        <input type="url" ng-model="website.url">
        <button ng-click="remove($index)">X</button>
    </div>
    <button ng-click="add()">Add Website</button>
</form>

```


控制器初始化模型，并提供帮助函数 `remove()` 和 `add()`，代码如下：

```
app.controller('MainCtrl', function($scope) {
  $scope.user = {
    websites: [
      {url: 'http://www.bloggs.com'},
      {url: 'http://www.jo-b.com'}
    ]
  };
  $scope.remove = function(index) {
    $scope.user.websites.splice(index, 1);
  };
  $scope.add = function() {
    $scope.user.websites.push({ url: ''});
  };
});
```

参见 <http://bit.ly/XHLEWQ>。

在模板中，使用 `ngRepeat` 指令迭代用户个人资料中的网站列表，重复块中的每个输入指令都和 `user.websites` 模型中对应的 `website.url` 进行了数据绑定。帮助函数只负责向数组中添加和删除条目，而 **AngularJS** 数据绑定完成剩下的工作。



也许另一种方案听上去不错：网站数组中每个网站项都是一个简单的包含URL的字符串。遗憾的是，这种方案并不能工作得很好，在JavaScript中，字符串按值传递，所以 `ngRepeat` 区块中的字符串并非是对数组中字符串的引用，也就是说，修改输入框中的值时，数组并不会更新。

验证重复输入

Validating repeated inputs

当为重复字段进行验证时，该方案将会遇到问题。我们需要表单中的每个输入都有独一无二的名字，以访问该字段的验证相关接口，`$valid`、`$invalid`、`$pristine`、`$dirty` 等。不幸的是，**AngularJS** 并不允许你为 `input` 指令动态生成 `name` 属性，后者必须是一个指定的字符串。

使用嵌套表单来解决此问题，每个表单都在当前作用域下暴露自己。所以，如果在每个包含重复输入指令的重复区块中都放置嵌套表单，就可以在作用域下访问字段的验证相关接口。

模板如下：

```
<form novalidate ng-controller="MainCtrl" name="userForm">
  <label>Websites</label>
  <div ng-show="userForm.$invalid">The User Form is invalid.</div>
  <div ng-repeat="website in user.websites" ng-form="websiteForm">
    <input type="url" name="website"
      ng-model="website.url" required>
    <button ng-click="remove($index)">X</button>
    <span ng-show="showError(websiteForm.website, 'url')">
      Please must enter a valid url </span>
    <span ng-show="showError(websiteForm.website, 'required')">
      This field is required</span>
  </div>
  <button ng-click="addWebsite()">Add Website</button>
</form>
```

控制器如下：

```
app.controller('MainCtrl', function($scope) {
  $scope.showError = function(ngModelController, error) {
    return ngModelController.$error[error];
  };
  $scope.user = {
    websites: [
      {url: 'http://www.bloggs.com'},
      {url: 'http://www.jo-b.com'}
    ]
  };
});
```

参见 <http://bit.ly/14i1sTp>。

这里我们在 div 上应用 ngForm 指令创建嵌套表单，对于作用域上 websites 数组中的每个网站，都会重复调用该表单。每个嵌套表单名为 websiteForm，该表单中的每个输入名为 website。这意味着对 ngRepeat 作用域中的每个 website，我们都能访问其对应的 ngModel 的验证接口。

可以利用这点在输入无效时显示错误信息，两个 ng-show 指令将会在 showError 函数返回 true 时显示它们的错误信息。showError 函数用于检查传入的 ngModelController，以查看 \$error 字段是否有对应的验证信息。我们为这个函数传送了 websiteForm.website，因为它就是对应 website 输入框的 ngModelController。

在 ngForm 之外,我们不能在作用域上引用 websiteForm (ngFormController) 对象或 websiteForm.website (ngModelController) 对象,因为它们更大的作用域上并不存在。然而,可以访问包含的 userForm (ngFormController) 对象。这个表单的验证,是基于它的所有子输入和子表单的验证的。如果其中一个 websiteForm 是无效的,则 userForm 也无效。当 userForms.\$invalid 为 true 时,表单顶部的 div 将会显示总体错误信息。

5.8 处理传统的 HTML 表单提交

Handling traditional HTML form submission

本节将介绍 AngularJS 是如何处理表单提交的。对于 AngularJS 最擅长的单页 (single page) AJAX 应用,不应像传统 web 应用那样,试图遵循直接向服务器提交的流程。但是,有时候应用的确需要类似的支持。这里我们将展示不同的提交实现场景,在你希望将表单数据直接提交给服务器的时候。

直接向服务器提交表单

Submitting forms directly to the server

在 AngularJS 应用中,如果为表单包含了 action 属性,表单就会向 action 所定义的 URL 进行正常提交,代码如下:

```
<form method="get" action="http://www.google.com/search">
  <input name="q">
</form>
```

参见 <http://bit.ly/115cQgq>。



注意, Plnkr 的预览会阻塞对 Google 的重定向。




处理表单提交事件

Handling form submission events

如果没有包含 action 属性,则 AngularJS 假定我们想要在客户端处理提交事件,通过在作用域上调用一个函数实现。此时,AngularJS 将阻止表单对服务器的直接提交。

可以利用 button 上的 ngClick 指令,或 form 上的 ngSubmit 指令,来触发该客户端函数。

 不应在同一表单中既使用 `ngSubmit` 指令，又使用 `ngClick` 指令，因为这两个指令浏览器都会触发，继而造成双重提交。


使用 `ngSubmit` 处理表单提交

为了在表单中应用 `ngSubmit`，需要提供一个表达式，它会在表单被提交时进行求值。表单提交将会在用户对输入其中之一按下 **Enter** 键或点击相应按钮时发生：

```
<form ng-submit="showAlert(q)">
  <input ng-model="q">
</form>
```

参见 <http://bit.ly/ZQBLYj>。

这里在输入框按下 **Enter** 键时会调用 `showAlert` 方法。

 `ngSubmit` 应只用于仅有一个输入和不多于一个按钮的表单，比如例子中的搜索表单。

使用 `ngClick` 处理表单提交

要在 `button` 或 `input[type=submit]` 上应用 `ngClick`，则要提供一个表达式，它将在按钮被点击时求值，代码如下：

```
<form>
  <input ng-model="q">
  <button ng-click="showAlert(q)">Search</button>
</form>
```

参见 <http://bit.ly/1530vLS>。

这里点击按钮或在输入字段上按下 **Enter** 键，会调用 `showAlert` 方法。

5.9 重置用户信息表单

Resetting the User Info form

在用户信息表单中，想要取消做出的改变并重设表单到它的初始状态，可以通过保存原始模型的一份拷贝来实现，这样就能抹消掉用户做出的任何变化。

模板如下：

```
<form name="userInfoForm">
  ...
  <button ng-click="revert()" ng-disabled="!canRevert()">Revert
  Changes</button>
</form>
```

控制器如下：

```
app.controller('MainCtrl', function($scope) {
  ...
  $scope.user = {
    ...
  };
  $scope.passwordRepeat = $scope.user.password;

  var original = angular.copy($scope.user);

  $scope.revert = function() {
    $scope.user = angular.copy(original);
    $scope.passwordRepeat = $scope.user.password;
    $scope.userInfoForm.$setPristine();
  };

  $scope.canRevert = function() {
    return !angular.equals($scope.user, original);
  };

  $scope.canSave = function() {
    return $scope.userInfoForm.$valid &&
      !angular.equals($scope.user, original);
  };
});
```

参见 <http://bit.ly/17vHLWX>。

这样就有了可以将模型回溯到原始状态的按钮，点击该按钮将会调用作用域上的 `revert()` 函数。该按钮将在 `canRevert()` 返回 `false` 时无效。

在控制器中可以看到，我们用 `angular.copy()` 制作了模型的拷贝，并将它赋给了一个本地变量。`revert()` 方法将这个原始模型拷贝回工作中的 `user` 模型，并将表单设置回纯净状态，这样所有的 `ng-dirty` CSS 类就都被移除了。

5.10 摘要

Summary

本章介绍了 **AngularJS** 如何扩展标准的 **HTML** 表单控制，以提供更强大灵活的用户输入系统。通过 `ngModel`、变化追踪机制、对输入的校验（使用校验指令）及 `ngFormController` 对象，成功做到了模型与视图的分离。

第 6 章将介绍如何完善管理应用的导航。我们将看到 **AngularJS** 如何支持深度链接，以将 **URL** 直接映射到应用的切面（**aspect**）上，以及如何利用 `ngView` 根据当前的 **URL** 自动为用户显示相关的内容。

第 6 章

导航

Organizing Navigation

第 5 章已经学习了如何从后端获取数据、使用 AngularJS 表单组件进行修改，并运用不同的指令将数据显示出来。本章介绍如何将各个页面组织成方便用户浏览的应用。

对用户来说，精心设计且容易记住的 URL（统一资源定位符）非常重要。用户可以使用熟悉的浏览器功能快速地浏览不同的页面。AngularJS 包含一些相关的服务和指令，让单页 web 应用也能用上 web 1.0 常用的 URL 功能，主要有：

- URL 路径可以指向单页 web 应用中的某一功能，之后可以被收藏、分享（例如，放在邮件内或者即时消息）。
- 浏览器的前进后退按钮可以正常使用，用户可以在单页 web 应用中的不同页面间跳转。
- 支持 HTML5 history API 的浏览器，URL 可以显示为直观且容易记忆的模式。
- 不管是支持 HTML5 history API 的浏览器还是旧版本的浏览器，AngularJS 都能提供跨浏览器的 URL 支持。

AngularJS 采用成熟的机制来处理 URL。本章将学习以下内容。

- 单页 web 应用中 URL 的使用。
- AngularJS 实现 URL 的方式：\$location 和 \$anchorScroll 服务。

- 在客户端使用\$route 服务（及其\$routeProvider）和ngView指令来组织导航。
- 在AngularJS单页web应用中使用 URL 过程的一些常用方法和技巧。

6.1 单页 Web 应用的 URL

URLs in single-page web applications

互联网初期，页面间的导航比较简单，只需要在浏览器地址栏输入 URL 即可，因为一个 URL 只用来指定服务器上一个单一的物理资源（如文件）。当页面加载后，就可以点击链接跳转到其他资源，或者使用前进后退按钮来跳转到已访问的页面。

动态生成的页面破坏了这种简单的导航方式，即使是相同的 URL，应用内部逻辑的变化也可以显示不同的页面内容。前进后退按钮也在这种情况下受到影响，它们的使用变得有点不可预测，很多网站不再建议用户使用前进后退按钮（而是建议用户使用导航链接）。

单页 web 应用并没有改善这种情况，反而更糟糕！在那些最前卫、重度依赖 Ajax 的应用中，地址栏上往往只能看到一个 URL（加载应用的首页）。之后的 HTTP 交互都是通过 XHR 对象实现的，而浏览器地址栏不会有任何变化。这时，前进后退按钮完全不起作用了，点击后可能会跳转到其他网站，而不是当前应用的其他页面。书签或复制粘贴浏览器地址栏中的链接也没有用。收藏后的链接也只是显示应用的起始页。

但浏览器的前进后退按钮和收藏地址功能还是非常有用的。用户希望在单页 web 应用中也能用上这些功能！有了 AngularJS，我们就能够像以前一样处理 URL！

HTML5 之前的 Hashbang URL

Hashbang URLs in the pre-HTML5 era

重度依赖 Ajax 的 web 应用可以通过一个技巧来支持原来的 URL。

这个技巧是通过修改 URL 地址中 # 符号之后的部分，而不会触发当前页面重新加载。这部分 URL 称为 URL 片段。通过修改 URL 片段，可以将新的元素加入浏览器的 history 堆栈（window.history 对象）。而前进后退按钮正好依赖浏览器的 history 堆栈。只要我们能保证 history 记录正确，这些导航按钮就能正常使用。

我们先来看看 CRUD 类型应用中常见的一些 URL。通常情况下，我们需要一个指向项目列表的 URL、一个编辑项目的表单和一个新项目的表单。以（来自简单的 SCRUM 应用）用户管理为例，大概会有以下这些不同的 URL。

- /admin/users/list: 这个 URL 将显示用户列表。
- /admin/users/new: 这个 URL 将显示新增用户表单。
- /admin/users/[userId]: 这个 URL 将显示编辑用户表单，用户 ID 等于 [userId]。

使用 # 技巧改造后的 URL 就变成这样：

- http://myhost.com/#/admin/users/list。
- http://myhost.com/#/admin/users/new。
- http://myhost.com/#/admin/users/[userId]。

指向单页 web 应用内部功能的 URL 部分一般称为“hashbang URL”。

使用这样的 URL 格式，可以修改浏览器地址栏的 URL 而不需要刷新页面。浏览器会选取 URL 的差异部分（# 符号后的 URL 片段）来供前进后退按钮使用。同时，它不会发起任何请求到服务器。

仅修改 URL 格式还是不够的，还需要 JavaScript 逻辑来监控 URL 片段的变化，并相应地更新客户端的状态。

HTML5 和 history API

HTML5 and the history API

用户都喜欢简单易记且能够收藏的 URL，而刚刚介绍的 hashbang 技巧却让 URL 变得又长又怪。幸运的是，HTML5 规范的 history API 可以很好地解决这个问题。

[



现在大多数浏览器都支持 history API。只有 IE 是从 10.0 版本才开始支持的，在这之前的版本只能使用 hashbang URL。

简单来说，使用 history API，可以模拟访问网页而不是真的向服务器发送请求。使用新的 history.pushState 方法，可以把这些完整而又直观的 URL 推送到浏览器的 history 堆栈。这个 history API 也采用内置的机制来监控 history 堆栈的变化。只要监听 window.onpopstate 事件就可以修改应用的状态。

使用 HTML5 history API，又可以在单页 web 应用中使用直观的 URL（不需要用到 # 技巧），用户也可以收藏 URL、使用前进后退按钮等。上一示例中的 URL 就会变成这样：

- `http://myhost.com/admin/users/list。`
- `http://myhost.com/admin/users/new。`
- `http://myhost.com/admin/users/[userId]。`

这些 URL 就像指向服务器上真实资源的“标准”URL。看起来不错，这就是我们想要的格式。但是，如果这些 URL 直接在浏览器中输入，浏览器还是会向服务器发送请求。

[



这种情况下，为了正常使用HTML5方式，服务器要进行正确的配置：始终返回应用的首页。在后面的章节中将详细讲解相应的服务器配置。

]

6.2 使用 \$location 服务 Using the \$location service

AngularJS 的 \$location 服务为 URL（及其行为）提供了一个抽象层。让开发者可以使用统一的 API，而不必关心用户所使用的浏览器和 hashbang 与 HTML5 方式间的区别。\$location 服务内部会进行很多相关的处理，包括：

- 提供方便简洁的 API 来获取 URL 中的各个部分（协议、主机、端口、路径、查询参数，等等）。

- 能以编程方式修改当前 URL 的不同部分，并即时更新浏览器地址栏。
- 允许监控当前 URL 的各个部分并做出响应。
- 截获用户在页面链接上所进行的操作（例如点击<a>标记），同时保存到浏览器历史中。

理解 \$location 服务 API 与 URL 的关系
Understanding the \$location service API and URLs

在开始使用 \$location 服务之前，我们需要熟悉它的 API。\$location 服务是与 URL 相关联的，可以采用最简单的方式来了解它：看看 URL 的不同部分是如何映射到 API 方法的。

先来看一个指向用户列表的 URL。为了保证示例完整，我们将使用一个带有所有部分的 URL，包括：路径、查询参数和片段。这个 URL 的基本部分看起来是这样的：

```
/admin/users/list?active=true#bottom
```

现在开始解析这个 URL：在后台管理面板、列表所有活跃的用户，并将屏幕滚动到最下方。它只是我们感兴趣的 URL 中的一部分，实际的 URL 还包含协议、主机等。根据使用方式（hashbang 或者 HTML5）的不同，URL 的完整格式看起来也会不一样。可以通过下方的 URL 来查看相同的 URL 在不同的方式下是怎样的。

在 HTML5 方式下显示为：

```
http://myhost.com/myapp/admin/users/list?active=true#bottom。
```

在 hahbang 方式下则显示有点怪：

```
http://myhost.com/myapp#/admin/users/list?active=true#bottom。
```

无论采用哪种方式，\$location 服务的 API 都会将差异屏蔽掉，提供统一的 API。表 6-1 显示了 API 中的部分方法。

表6-1

方法	对应上述示例，将返回
\$location.url()	/admin/users/list?active=true#bottom
\$location.path()	/admin/users/list
\$location.search()	{active: true}
\$location.hash()	Bottom

这些方法都是 jQuery getter 风格。换句话说，它们都可以用来设置和获取 URL 部分的值。例如，可以使用 `$location.hash` 来获取 URL 片段的值，也可以通过引入参数来设置它的值：`$location.hash('top')`。

`$location` 服务还提供了其他方法（没有在表 6-1 中列出）来访问 URL 的任何部分：协议（`protocol()`）、主机（`host()`）、端口（`port()`）和 URL 绝对地址（`baseUrl()`），这种方法是只读的，不可修改。

哈希、页面内导航和 `$anchorScroll` Hashes, navigation within a page, and `$anchorScroll`

一般情况下，# 符号之后的 URL 是用来定位的。现在被单页 web 应用用来导航，就会产生冲突。但有时候我们确实需要浏览器滚动到页内指定的位置。现在来看看冲突所在：在 hashbang 方式，URL 将包含两个 # 符号，例如，

```
http://myhost.com/myapp#/admin/users/list?active=true#bottom
```

浏览器根本不知道应该使用第二个哈希（#bottom）来进行页面内定位。需要 AngularJS 提供一些帮助才行，这就是 `$anchorScroll` 服务的用武之地。

默认情况下，`$anchorScroll` 服务会监控 URL 片段。一旦检测到有需要用于页内定位的哈希，它就会滚动到相应的位置。这个过程在 HTML5 方式（URL 中只有一个哈希）和 hashbang 方式（URL 中有两个哈希）都能顺利运行。总之，`$anchorScroll` 服务会做好之前浏览器所做的工作，同时将 hashbang 方式也考虑进去。

当需要手动控制 `$anchorScroll` 服务的滚动效果时，可以通过调用 `$anchorScrollProvider` 服务的 `disableAutoScrolling()` 方法来禁用 URL 片段监测，代码如下：

```
angular.module('myModule', [])
  .config(function ($anchorScrollProvider) {
    $anchorScrollProvider.disableAutoScrolling();
  });
```

通过这样的设置，可以随时调用 `$anchorScroll()` 函数来触发滚动。

配置 HTML5 方式的 URL

Configuring the HTML5 mode for URLs

AngularJS 的 URL 会默认使用 hashbang 方式，想要使用直观的 HTML5 方式，则需要修改 AngularJS 的默认配置，以及让服务端也支持可收藏的 URL 格式。

客户端

在 AngularJS 中，很容易修改 URL HTML5 模式，只需要调用 `$locationProvider` 的 `html5Mode()` 方法，代码如下：

```
angular.module('location', [])
  .config(function ($locationProvider) {
    $locationProvider.html5Mode(true);
  })
```

服务端

为了让 HTML5 模式正常使用，还需要服务端的支持，也就是说，当浏览器请求访问深层（deep-linking）链接时，服务端要能重定向到单页 web 应用的起始页面（包含 `ng-app` 指令）。

我们通过一个例子来理解为什么需要这样的重定向：用户收藏的 URL（HTML5 模式）指向某个项目的待办事项列表，如下：

```
http://host.com/projects/50547faee4b023b611d2dbe9/productbacklog
```

浏览器认为这是一个正常的 URL，并向服务端发送请求。但这个 URL 只在单页 web 页面的客户端有效，服务器根本没有 `/projects/50547faee4b023b611d2dbe9/productbacklog` 这样的静态页面，也无法动态生成，只能重定向到起始页面，让客户端重新加载 AngularJS 应用。之后 `$location` 服务会获取浏览器地址栏的 URL，这样客户端就可以进行了处理。

本书不深入讨论如何配置不同的服务器，只探讨一些基本的规则：大概有三种 URL 类型需要 web 服务器进行处理。

- 指向图片、CSS 文件、AngularJS 局部文件（Partials）等静态资源的 URL。
- 获取后端数据或者修改数据的请求（例如 RESTful API）URL。
- 代表应用功能且以 HTML5 模式显示的 URL（用户收藏或者直接在浏览器输入），服务器需要跳转到应用的起始页面。

将所有 HTML5 模式的链接一一列出会相当麻烦，最好是使用常用的地址前缀来代表静态资源和修改数据的操作。这也是本书示例 SCRUM 应用所采用的方法。所有以 /static 开头的 URL 代表静态资源，以 /databases 开头的则代表修改后端数据，其余的 URL 将定向到 SCRUM 应用的起始页面（index.html）。

使用 \$location 导航

Handcrafting navigation using the \$location service

现在我们已经了解了 \$location 服务的 API 及其配置，可以将这些知识付诸实践了。下面将使用 ng-include 指令和 \$location 服务来进行简单的导航。

既然是单页 web 应用中最简单的导航，也应该提供一些工具方便开发者使用，我们至少需要做这些工作：

- 以统一且易于维护的方式定义路由（routes）。
- URL 变化时更新应用。
- 用户进行导航（点击链接，使用前进后退按钮等）时更新 URL。



在本书接下来的内容中，术语路由（routing）表示一个用来响应 URL 变化并同步应用状态的地方。一个路由匹配到一个 URL 之后，就会更新应用的状态。

根据路由构建页面

查看示例代码之前，我们应该知道，典型的 web 应用包括像页头、页尾这样的固定内容，也包括随着用户操作而改变的动态内容。考虑到这点，我们以固定内容和动态内容分离的方式组织标记（markup）和控制器（controller）。回到上一个 URL 示例，应用的 HTML 结构如下：

```
<body ng-controller="NavigationCtrl">
<div class="navbar">
  <div class="navbar-inner">
    <ul class="nav">
      <li><a href="#/admin/users/list">List users</a></li>
      <li><a href="#/admin/users/new">New user</a></li>
    </ul>
  </div>
</div>
<div class="container-fluid" ng-include="selectedRoute.templateUrl">
  <!-- Route-dependent content goes here -->
</div>
</body>
```

上一个例子的动态内容使用带 `ng-include` 指令的 `<div>` 标记来表示，并指向一个动态 URL：`selectedRoute.templateUrl`。看看这样的表达式是如何在 URL 改变时更新内容的？

先看 JavaScript 部分的 `NavigationCtrl`，可以这样来定义 `routes` 变量：

```
.controller('NavigationCtrl', function ($scope, $location) {

  var routes = {
    '/admin/users/list': {templateUrl: 'tpls/users/list.html'},
    '/admin/users/new': {templateUrl: 'tpls/users/new.html'},
    '/admin/users/edit': {templateUrl: 'tpls/users/edit.html'}
  };
  var defaultRoute = routes['/admin/users/list'];
  ...
});
```

`routes` 对象给出了应用的基础结构，它将所有可能的局部模板和 URL 对应起来。通过这个路由定义，可以看出应用是由哪些页面组成的，每一个路由将使用哪些局部模板。

路由映射URL

有了这样一个简单的路由结构还不够，还需要使用当前 URL 来同步激活路由。通过监测当前 URL 的 `path()` 组件就可以实现：

```
$scope.$watch(function () {  
    return $location.path();  
}, function (newPath) {  
    $scope.selectedRoute = routes[newPath] || defaultRoute;  
});
```

这样，`$location.path()` 组件每次变化都会去查找路由，只要找到相应的路由就会直接使用，否则使用默认路由。

定义路由时指定控制器

当某条路由成功匹配时，就会加载相应的局部模板，`ng-include` 指令会将它显示在页面上。之前提到的 `ng-include` 会创建一个新的 `scope`，一般情况下，我们要为这个 `scope` 提供一些数据，例如，用户列表、需要编辑的用户信息等，以供新加载的 `partial` 使用。

AngularJS 使用控制器来接管数据以及行为。我们需要为每个局部模板都定义一个控制器，最直接的方法就是在每一个局部模板的根元素上使用 `ng-controller` 指令。例如，用于修改用户信息的局部模板如下：

```
<div ng-controller="EditUserCtrl">  
    <h1>Edit user</h1>  
    ...  
</div>
```

这种方法的缺点是，不同的控制器不能重用相同的局部模板。有些情况下，使用相同的局部模板会很方便，只需修改局部模板背后的数据和行为。最常见的例子是编辑表单，我们想使用相同的表单来新增或修改项目，这时表单是完全一样的，只是数据会有变化。

导航的不足

这个特殊的例子不是很健壮，也不完整，在实际项目中不会这样使用。这里只为了展示 `$location` 服务一些有趣的特征。

首先，`$location` 服务的 API 不只对浏览器原生 API 进行了良好的封装，还兼容了不同的 URL 模式（hashbang 以及 HTML5 history）。

其次，我们能体会到 AngularJS 及其服务带来的便捷，那些之前尝试使用原生 JavaScript 来编写类似导航系统的开发者则更能体会到这一点。不过，有些地方还是需要改进，这次我们不基于 `$location` 服务，而会讨论 AngularJS 自带的方案，`$route` 服务，来解决这些问题。

6.3 使用 AngularJS 自带的路由服务

Using built-in AngularJS routing services

AngularJS 框架内置 `$route` 来处理单页 web 应用的路由转换，它不只包含了我们试图使用 `$location` 服务来实现的所有功能，还有一些特别有用的工具，下面一步步来熟悉这些内容。



从1.2版本开始，AngularJS将路由系统分离到单独的文件（`angular-route.js`）中，全名为`ngRoute`模块。当使用AngularJS的最新版本时，请记得包含`angular-route.js`文件，并声明依赖于`ngRoute`模块。

基础路由定义

Basic routes definition


在深入学习之前，请忘掉之前的简单例子，改为使用 `$route` 服务的语法来定义路由。

AngularJS 可以通过 `$routeProvider` 来定义应用的配置，用法大致与 `$location` 类似：


```
angular.module('routing_basics', [])
  .config(function($routeProvider) {
    $routeProvider
      .when('/admin/users/list', {templateUrl: 'tpls/users/list.html'})
      .when('/admin/users/new', {templateUrl: 'tpls/users/new.html'})
      .when('admin/users/:id', {templateUrl: 'tpls/users/edit.html'})

      .otherwise({redirectTo: '/admin/users/list'});
  })
```

`$routeProvider` 服务提供流畅风格的 API，允许以链接的方式定义新的路由（when）以及定义默认路由（otherwise）。

 初始化之后，不能再通过修改配置来增加（或者删除）路由，因为 AngularJS 提供了注入（injected）机制，所以只能在应用初始化阶段修改配置。

在上一个例子中，虽然每个路由只有 `templateUrl` 属性，但其实 `$routeProvider` 服务提供的语法非常丰富。

 路由的内容甚至可以直接使用 `template` 属性，虽然这样的语法也被支持，但它非常不灵活（也不利于维护），所以很少使用。

显示匹配的路由内容

当一个 URL 匹配了路由时，将使用 `ng-view` 指令显示路由的内容（由 `templateUrl` 或者 `template` 定义）。之前基于 `$location` 的版本是这样的：

```
<div class="container-fluid" ng-include="selectedRoute.templateUrl">
  <!-- Route-dependent content goes here -->
</div>
```

使用 `ng-view` 重写如下：

```
<div class="container-fluid" ng-view>
  <!-- Route-dependent content goes here -->
</div>
```

如同你所看到的，我们只是简单地将 `ng-include` 指令替换为 `ng-view` 指令，并且不再需要属性值，因为 `ng-view` 指令可以通过当前匹配的路由找到要显示的内容。

匹配灵活的路由

Matching flexible routes

在上一个例子中，我们使用了非常简单的路由匹配算法，在 URL 不能使用变量。也许不能称为算法，只是简单地查询对象的属性，找到相应的 URL 路径！由于算法简单，我们会将用户 ID 放在 URL 查询参数：

```
/admin/users/edit?user={{user.id}}
```

将用户 ID 嵌入 URL 内部，看起来更直观：

```
/admin/users/edit/{{user.id}}
```

有了 AngularJS 路由可以很容易实现，URL 中任何以冒号（:）开头的字符串都会作为通配符。将用户 ID 作为嵌入 URL，代码如下：

```
.when('/admin/users/:userid', {templateUrl: 'tpls/users/edit.html'})
```

这个路由定义将匹配任何可以替换 `:userid` 通配符的 URL，例如，

```
/users/edit/1234
```

```
/users/edit/dcc9ef31db5fc
```

不过，URL 中没有 `:userid` 部分，或者 `:userid` 只有斜杠（/）不会被匹配。



某些情况下，匹配的参数可能要包含斜杠，这时可以使用类似的语法：`*id`。例如，使用星号语法去匹配包含斜杠的路径：`/wiki/pages/*page`。整个路由匹配语法在 AngularJS 1.2 版本会进一步扩展。

定义默认路由

通过调用 `otherwise` 方法，可以设置默认路由。默认路由只能有一个，所以不需要指定任何 URL。



通常，默认路由会定义 `redirectTo` 属性，跳转到某个已定义的路由。

没有指定路径的 URL 或者路径无效的 URL（没有匹配的路由）都会匹配到默认路由，触发相应的路由更新事件。

访问路由参数

URL 路由定义可以包含变量作为参数。当一个路由匹配到 URL 时，使用 `$routeParams` 服务就能轻易获取这些参数值。实际上，`$routeParams` 服务也是一个简单的 JavaScript 对象（哈希），哈希键是路由参数名称，哈希值是从匹配的 URL 解析出来的字符串。

`$routeParams` 作为一个服务，也可以被 AngularJS 依赖反转系统注入任何对象。我们来看一个控制器（`EditUserCtrl`）更新用户数据（`/admin/users/:userid`）的例子：

```
.controller('EditUserCtrl', function($scope, $routeParams, Users){
    $scope.user = Users.get({id: $routeParams.userid});
    ...
})
```

`$routeParams` 服务可以兼容 URL 路径中或者查询参数（`?` 符号之后）中的参数。以上代码同时适用于 `/admin/users/1234/edit` 以及 `/admin/users/edit?userid=1234` 两种 URL。

多个控制器重用局部模板

Reusing partials with different controllers

本书之前讨论的方法都是在局部模板使用 `ng-controller` 指令来初始化局部作用域（`scope`）。其实，AngularJS 路由系统还允许在路由级别上定义控制器。将控制器从局部模板抽离，可以降低耦合度。

编辑用户数据页面常规的做法是这样：

```
div ng-controller="EditUserCtrl">
  <h1>Edit user</h1>
  . . .
</div>
```

删除 `ng-controller` 指令的代码如下：

```
<div>
  <h1>Edit user</h1>
  . . .
</div>
```

相应的路由定义如下：

```
.when('/admin/users/:userid', {
  templateUrl: 'tpls/users/edit.html'
  controller: 'EditUserCtrl'})
```

将控制器声明移到路由定义中，相同的控制器就可以用于不同的局部模板。更有用的是，多个控制器能重用相同的局部模板，在某些场景下非常灵活。举个典型的例子：一个局部模板包含编辑项目的表单，通常情况下，我们想让新增以及修改项目的表单输入项保持一致，但是行为上会有些许差别（例如，新增时执行保存操作，修改已有项目时则只是更新操作）。

路由改变时避免 UI 抖动

Avoiding UI flickering on route changes

当应用切换页面时，会将新页面的元素显示出来，同时获取相应的实体数据。渲染新页面有两种方式。

- 尽早显示新页面元素（尽管数据还没有准备好），从后端获取到数据后，再重新渲染显示 UI。
- 等待所有后端数据都获取到后再显示新页面元素。

AngularJS 默认使用第一种方式。对于同时定义了 `templateUrl` 以及控制器属性的路由，AngularJS 匹配路由之后不会等待控制器获取数据，而直接渲染局部模板的内容；当数据就绪（并绑定到作用域）之后会再次渲染局部模板，此时，用户可能会注意到难看的抖动效果。因为在短时间内，相同的局部模板被渲染了两次，第一次没有数据，第二次才绑定了数据。


对于第二种方式，AngularJS 路由系统也提供了支持：等待数据就绪后，才会更新路由（以及重新渲染 UI）。我们可以在路由定义时指定 `resolve` 属性，将路由控制器所有依赖的异步数据都罗列出来。AngularJS 会确保所有依赖条件都满足后才进行路由更新及初始化控制器。

为了展示 `resolve` 属性的基本用法，我们来重写“编辑用户”的路由定义：

```
.when('/admin/users/:userid', {
  templateUrl: 'tpls/users/edit.html'
  controller: 'EditUserCtrl',
  resolve: {
    user: function($route, Users) {
      return Users.getById($route.current.params.userid);
    }
  }
})
```

`resolve` 属性是一个对象，哈希键定义的新变量将会被注入路由控制器，这些变量的值由一个函数提供。函数参数也可以由 AngularJS 依赖反转系统注入。在这里，`$route` 及 `Users` 服务会被注入，用于获取用户数据。

`resolve` 函数可以返回简单的 JavaScript 值、对象或者 `promise`。当返回的是 `promise` 时，AngularJS 会等待 `promise` 处理完成后才更新路由。如果有多个 `resolve` 函数返回 `promise`，那么 AngularJS 路由系统会确保所有的 `promise` 都已完成。

 路由定义中的 `resolve` 函数可以返回 `promise`。所有的 `promise` 处理成功才会更新路由。

所有定义在 `resolve` 属性的变量都处理完毕，它们将被注入路由控制器：

```
.controller('EditUserCtrl', function($scope, user){
    $scope.user = user;
    ...
})
```

这种模式非常强大、实用，它允许我们定义本地路由变量，然后注入路由控制器。在 **SCRUM** 示例应用中，将运用它结合不同的 `user` 变量（新建或者从后端获取）来重用相同的控制器，代码如下：

```
$routeProvider.when('/admin/users/new', {
    templateUrl:'admin/users/users-edit.tpl.html',
    controller:'UsersEditCtrl',
    resolve:{
        user: function (Users) {
            return new Users();
        }
    }
});

$routeProvider.when('/admin/users/:userId', {
    templateUrl:'admin/users/users-edit.tpl.html',
    controller:'UsersEditCtrl',
    resolve:{
        user: function ($route, Users) {
            return Users.getById($route.current.params.userId);
        }
    }
});
```

在路由的 `resolve` 属性中定义本地变量，并注入路由控制器，极大地提高了控制器的可测试性。

取消路由更新


Preventing route changes

在某些情况下，我们需要取消路由更新，例如，以下路由用于编辑用户数据。


```
/users/edit/:userid
```

当用户不存在时，我们需要确定如何处理。起码不能跳转到根本不存在的用户编辑页面。

路由定义的 `resolve` 属性支持终止页面跳转。只要 `resolve` 键值中的 `promise` 被拒绝（`rejected`），AngularJS 就取消页面跳转，保持 UI 不变。

 在路由定义的 `resolve` 返回结果中，只要有一个 `promise` 被拒绝，路由变更的操作就会被取消，UI 也不会进行更新。


需要留意的是，如果取消路由更新，则浏览器地址栏的 URL 不会被恢复。例如，访问 `/users/list` 会显示用户列表，列表中所有的用户记录都采用链接指向相应的编辑表单（`/users/edit/:userid`），点击链接将会更新浏览器地址栏（变为类似 `/users/edit/1234` 的地址），但并不能保证我们能修改用户的信息（可能刚被删除或者没有足够权限等）。如果路由导航被取消，虽然 UI 会保持 `/users/list` 路由的内容，但浏览器地址栏不会恢复之前的 URL，仍然会显示 `/users/edit/1234`。

 如果路由被取消，则浏览器地址栏和 UI 显示的内容可能会不同步。

6.4 \$route 服务的局限

Limitations of the \$route service

虽然 `$route` 服务有着良好的设计，在很多应用中表现出色，但它自身也有短板。本节将列出这些不足内容，读者可以稍加留意，必要时修改应用的设计，或者自定义开发路由服务。

 现在已经有一个由开发者社区维护的开源项目：`ui-router`，它为 AngularJS 应用提供更为强大的路由系统，包括嵌套路由以及支持页面多区域路由（`multiple rectangles`）。原书出版之时，`ui-router` 仍在完善阶段，读者可以关注：<https://github.com/angular-ui/ui-router>。

一个路由只对应页面中的一个区域

One route corresponds to one rectangle on the screen

我们知道，`ng-view` 指令表示 DOM 元素的内容要被路由定义的内容替换（根据 `template` 或者 `templateUrl` 属性），也就是说，在 `$route` 服务中，一个路由只能影响 UI 中的一块内容。

事实上，当路由变化时，经常需要更新页面上的多个区域。例如，多个路由可能会共用菜单，截图如图 6-1 所示。

当用户导航到管理员页面时，则需要保持 ADMIN 菜单。当前 AngularJS 版本实现这种导航方式的唯一方法是使用多个 `ng-include` 指令，具体会在下一节中介绍。

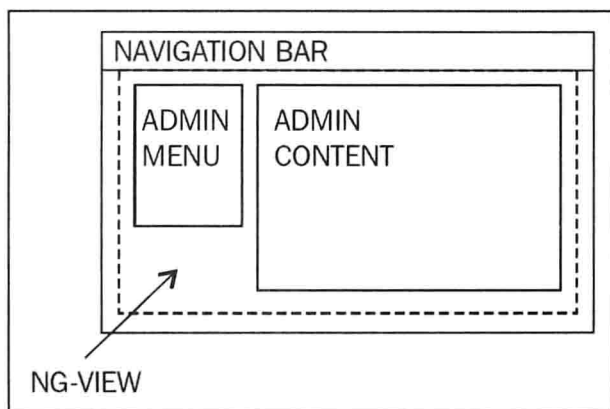


图6-1

使用ng-include处理多个UI区域

路由定义对象是普通的 JavaScript 对象，也可以包含自定义的属性。虽然这些属性不会被 AngularJS 处理，但会保留下来，以便我们进一步处理。了解这一点，可以在路由中自定义 `menuUrl` 及 `contentUrl` 属性：

```
$routeProvider.when('/admin/users/new', {  
    templateUrl: 'admin/admin.tpl.html',
```

```

    templateUrl: 'admin/users/users-edit.tpl.html',
    menuUrl: 'admin/menu.tpl.html',
    controller: 'UsersEditCtrl',
    ...
  });

```

之后，还要将 `templateUrl` 属性指向新的局部模板，它再将 `menu` 以及 `content` 作为子模板，代码如下：

```

<div>
  <div ng-include='$route.current.contentUrl'>
    <!--menu goes here -->
  </div>
  <div ng-include='$route.current.menuUrl'>
    <!--content goes here -->
  </div>
</div>

```

这种变通的方法虽然能达到我们想要的视觉效果，但是每次更新路由菜单的 DOM 元素都会被重新渲染，这些操作都是没有必要的。如果局部模板比较简单且不需要从后端获取数据，那么问题不大。但是，一旦局部模板要进行耗资源的处理，就得谨慎应对。

不支持嵌套路由

No nested routes support

路由系统的另一个缺点是不支持嵌套路由，只能有一个 `ng-view` 指令；或者说，我们不能在局部模板中使用 `ng-view` 指令，这在路由原生就具有层次结构的大型应用可能会是个问题。SCRUM 示例应用也有一些这样的路由：

- `/projects`：列出所有项目。
- `/projects/[project id]/sprints`：列出某个项目所有的冲刺（sprint）记录。
- `/projects/[project id]/sprints/[sprint id]/tasks`：列出某个项目中某个冲刺的所有任务。

这种导航方案直接就能看出它的层次结构，如图 6-2 所示。

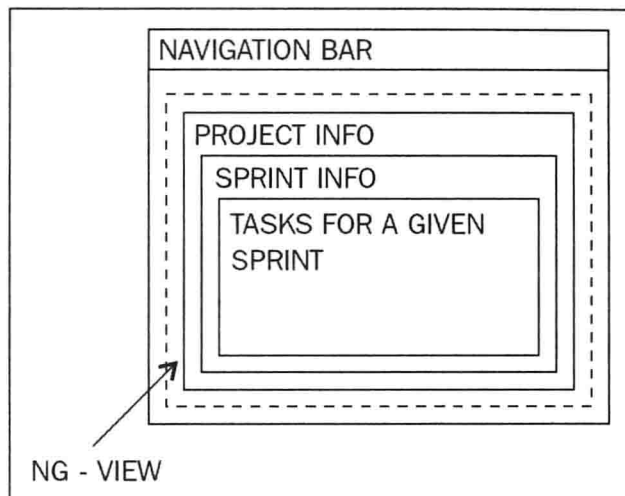


图6-2

在相同的项目不同的冲刺（sprint）间切换任务时，要保持这个项目信息的数据以及 UI 不变。但是，由于受到“一个路由对应一个区域”的限制，项目相应的 UI 以及数据都会随着整个页面动态内容的刷新而被重新渲染。也就是说，每次路由更新，即使项目不变，也要重新获取项目的信息。

除了可以使用 `ng-include` 指令来嵌套 UI 元素，但数据获取的操作还是无法避免。唯一能做的就是优化后端（或者加上缓存），让数据获取更快一些。

6.5 路由相关的模式及技巧

Routing-specific patterns, tips, and tricks

本章简要介绍了 AngularJS 与单页 web 导航相关的 API，接下来学习几个实用的例子以及最佳实践。

处理链接

Handling links

HTML 链接标记（`a`）是创建导航的主要工具，AngularJS 会为这些链接进行特殊的处理，包括以下内容。

创建可点击的链接

当链接的 `href` 属性没有被指定时，AngularJS 会防止链接跳转。只要使用一个 `a` 标记和 `ng-click` 指令就可以创建可点击的链接。例如，点击以下代码所示的链接就可以调用 `scope` 的函数：

```
<a ng-click='showFAQ()'>Frequently Asked Questions</a>
```

直接使用 `a` 标记而忽略默认的导航操作会非常方便，因为主流 CSS 框架都使用 `a` 标记来渲染不同风格的按钮外观。例如，Twitter Bootstrap CSS 框架就是使用 `a` 标记来渲染菜单样式的。

为 `a` 标记指定行为而不使用 `href` 属性，你可能想知道哪一种方法更适合用来创建导航链接。以下是直接使用 `href` 属性的方法：

```
<a href="/admin/users/list">List users</a>
```

另一种方法如下：

```
<a ng-click="listUsers()">List users</a>
```

`listUsers` 方法会定义在 `scope`：

```
$scope.listUsers = function() {
    $location.path("/admin/users/list");
};
```

实际上，这两种方法还是存在一些细微的差别。使用 `href` 属性创建的链接对用户来说要友好一些，用户可以通过右击链接，并选择在浏览器新的标签页（或者窗口）打开。一般情况下，尽量使用 `href` 属性来创建导航链接，或者使用 AngularJS 的 `ng-href` 属性来创建动态 URL：

```
<a ng-href="/admin/users/{{user.$id()}}">Edit user</a>
```

兼容HTML5及hashbang模式

AngularJS 应用的 `$location` 服务可以配置使用 `hashbang` 或者 `HTML5` 两种模式，不管采用哪种模式，都需要在配置块中预先指定，链接也要根据所选模式创建。

选择 **hashbang** 模式，创建链接的代码如下（请注意 # 符号）：

```
<a ng-href="#/admin/users/{{user.$id()}}">Edit user</a>
```

而 **HTML5** 模式的链接会稍微简单些：

```
<a ng-href="/admin/users/{{user.$id()}}">Edit user</a>
```

正如上一个例子所看到的，所有的链接都需要根据 `$locationProvider` 服务设置的模式进行创建。问题是，每个普通的应用都会包含很多链接，如果我们决定要修改 `$location` 的配置，就要找出应用的所有链接，添加（或者删除）# 符号。



所以，请尽早决定 URL 的模式，避免后期在整个应用中检查和修改所有链接。

链接外部页面

AngularJS 假设所有使用 `a` 标记创建的链接都是指向应用的内部功能，点击后直接更新应用的状态，而不会重新加载页面。大多数情况下，这正是我们想要的行为。但有时我们想让链接指向某个资源，则浏览器可以进行下载。在 **HTML5** 模式下，单单通过 **URL** 地址区分不了究竟是应该更新应用的状态还是下载外部资源。可以使用 `target` 属性来解决这个问题：

```
<a href="/static/form.pdf" target="_self">Download</a>
```

组织路由定义

Organizing route definitions

一个大型的 **web** 应用通常会包含非常多的路由。即使 `$routeProvider` 服务提供了风格非常直观的 **API**，路由定义（特别是使用了 `resolve` 属性）还是会显得非常冗长。这些定义合并在一起，组成一个包含几百行代码的大型 **JavaScript** 文件，维护起来就相当困难，每个开发者都要频繁修改这个文件，单单处理版本控制的冲突问题就够呛了。

将路由定义分离到多个模块

其实，AngularJS 并没有要求我们将所有的路由定义到一个文件！可以采用第 2 章（构建与测试）提到的方法：每个功能都能依赖于模块，也可以将路由移到应用相关的模块。

AngularJS 的模块可以调用 `config` 函数，通过注入 `$routeProvider` 服务来定义路由。例如，SCRUM 示例应用的 `administration` 模块包含两个子模块：一个用于管理用户，另一个用于管理项目。每个子模块都可以定义路由，代码如下：

```
angular.module('admin-users', [])
.config(function ($routeProvider) {
  $routeProvider.when('/admin/users', {...});
  $routeProvider.when('/admin/users/new', {...});
  $routeProvider.when('/admin/users/:userId', {...});
});

angular.module('admin-projects', [])
.config(function ($routeProvider) {
  $routeProvider.when('/admin/users', {...});
  $routeProvider.when('/admin/users/new', {...});
  $routeProvider.when('/admin/users/:userId', {...});
});

angular.module('admin', ['admin-projects', 'admin-users']);
```

采用这种方法，可以将路由定义分离到多个模块，以避免维护一个包含所有路由定义的大文件。

减少路由定义的重复代码

正如上一节所述，路由定义在使用了 `resolve` 属性后会变得更加冗长。通过仔细观察会发现，这些定义的代码块和配置都包含了大量重复的内容。这种情况在增删改查（CRUD）一类的应用中特别明显，每个功能都有相同的模式。

可以编写自定义的 `provider`，对 `$routeProvider` 服务器进行封装，以减少代码重复。这个 `provider` 提供更高层次的 API。

按照这个思路自定义编写的 **provider**，在不同的应用间差异会很大，所以我们不介绍具体的代码编写步骤，只是大致了解这种方法如何大幅减少 **SCRUM** 示例应用中路由定义的代码量。具体的源代码可以在 **Github** 找到。以下代码展示了自定义 **API** 的使用方法：

```
angular.module('admin-users', ['services.crud'])
.config(function (crudRouteProvider) {
  crudRouteProvider.routesFor('Users')
    .whenList({
      users: function(Users) { return Users.all(); }
    })
    .whenNew({
      user: function(Users) { return new Users(); }
    })
    .whenEdit({
      user: function ($route, Users) {
        return Users.getById($route.current.params.itemId);
      }
    });
});
```

这就是 **CRUD** 应用一个功能所需要的所有路由定义！

6.6 总结

Summary

经过良好设计的应用导航链接非常重要，它代表了应用的整体结构，可以让用户快捷地访问应用，让开发者自身也能更好地组织代码。

在本章，我们看到了运用 **AngularJS** 构建的应用。在链接以及导航方面，相对于 **web 1.0** 时代，能提供更良好的用户体验。这意味着，用户又可以使用浏览器的前进后退按钮进行导航。更进一步，浏览器地址栏又可以显示直观的、可收藏的 **URL**。

`$route` 服务（及其 **provider** - `$routeProvider`）在路由更新时，允许页面（一个区域）进行更新。这个自带的 `$route` 服务可以应付大多数应用的需要，如果“一个路由对应一个区域”无法满足，则可以考虑选择另一种由社区驱动的解决方案。如果你有足够的信心，运用之前我们介绍的 `$location` 服务，也可以考虑开发自己的路由系统，`$route` 服务也是基于 `$location` 底层 API 构建的。

本章结尾的几个章节，我们介绍了一些模式、技巧以及大型应用在使用路由时经常遇到问题的解决方案。我们希望，这些例子也能适用你的应用。特别建议你將路由定义移到各个相关的模块，以避免维护一个包含所有路由定义的大文件。

路由和导航相关的内容还要考虑安全性，应用中不是所有内容都允许公开访问，我们要限制用户只能使用一部分路由。第7章将介绍安全路由及一些与安全相关的细节。

第 7 章

安全

Securing Your Application

web 应用需要限制未授权用户访问敏感数据或者进行非法操作。虽然我们可以保障服务器的安全性，但由于浏览器端的代码很容易被篡改，所以，所有与服务器交互的数据都需要进行检查。本章的第一部分将介绍如何保证服务端及客户端的安全性：

- 禁止未授权用户访问数据。
- 加密链接防止网络监听。
- 防止跨站脚本攻击（XSS）及跨站伪造请求攻击（XSRF）。
- 阻止 JSON 注入攻击。


虽然服务端会负责安全检查，但还是要提供用户体验良好的客户端页面，只显用户有权限访问的功能；对于关键业务，要提供简洁的认证流程。本章第二部分将介绍 AngularJS 是如何做到这几点的：

- 相对于传统无状态（stateless）基于服务端的应用，要确保有状态（stateful）富客户端应用的安全性，有哪些不同之处。
- 服务端通过拦截 HTTP 响应来处理认证错误。
- 限制部分路由（route）访问。

7.1 提供服务端认证和授权

Providing server-side authentication and authorization

对于 B/S 结构的应用来说，只有服务端的数据才是安全的。我们不能依赖客户端的代码来阻止敏感信息的访问，服务端也不能信赖客户端已经校验过的数据。

 这在 JavaScript 应用中更为重要，因为它的源代码能被直接看到，只需稍加修改就能执行一些恶意的操作。

在实际 web 应用中，服务端需要达到适当的安全级别。对于演示应用，服务端已经有简单的安全措施：使用 ExpressJS 的插件 Passport 来实现身份验证和授权。用户登录后，经过验证的 userID 会存储在加密过的 session cookie 中并传递给浏览器。每次发送请求时，cookie 也会被传递回服务端进行身份验证。

处理未授权的访问

Handling unauthorized access

当用户访问某个 URL 时，服务端会检查这个 URL 是否需要验证用户、用户是否有足够的权限。在应用中，当客户端尝试以下操作时，服务器会返回 401 未认证错误：

- 未认证用户发起的任何 non-GET 请求（例如 POST、PUT 或者 DELETE）。
- 非管理员用户对用户或者项目数据库集合发起的 non-GET 请求。

通过这种方式，可以阻止未授权数据（JSON 请求）的访问，也可以防止其他资源，如 HTML 或者图片的非法访问。

提供服务端验证 API

Providing a server-side authentication API

客户端可以通过服务端的 HTTP API 来验证用户。

- POST/login: 通过 POST 请求的 username 和 password 参数来验证用户。

- POST /logout: 通过删除已认证的 cookie 信息来实现用户退出。
- GET /current-user: 获取当前用户信息。

这些接口足够演示应用是如何处理验证和授权的。



商业应用对安全性的要求更为复杂，或者你可以考虑使用第三方的认证方案，如 OAuth2（详见<http://oauth.net/>）。

7.2 保护局部模板

Securing partial templates

有些情况下，当不希望用户访问 AngularJS 路由时，由于权限不足获取不到数据，却仍能看到局部模板（HTML）的内容，特别是模板包含了敏感信息。

在这种情况下，服务端要认证访问这些局部模板的请求。首先，应用启动时不能再预加载这些局部模板；然后，当用户请求访问这些局部模板时，由服务端检查权限，对于未授权的用户，会返回 HTTP 401 错误。



当需要服务端对每个局部模板请求进行身份验证时，浏览器（或任何代理）不能缓存局部模板。也就是说，服务端返回这些局部模板时，要指定以下 HTTP header:

```
Cache-Control : no-cache, no-store, must-revalidate
Pragma       : no-cache
Expires      : 0
```

AngularJS 会缓存所有使用 `$templateCache` 服务获取的模板。如果要保护这些模板，就不能在用户验证通过前进行缓存。用户登录前，我们必须从 `$templateCache` 服务中删除这些模板，以避免用户无意中看到这些内容。

可以从 `$templateCache` 服务中选择性地或者完全地删除模板。例如，当用户导航到其他页面或者退出时，则删除当前受保护的模板。这种方式不利于管理，最安全的办法可能是重新加载页面，也就是说，当用户退出时，重新刷新浏览器。



通过刷新浏览器来重载应用有一个额外的好处：完全清除 AngularJS 服务缓存的所有数据。

7.3 阻止恶意攻击

Stopping malicious attacks

服务端与浏览器之间必须有一种相互信任的机制来允许合法用户的正常访问。但是，许多攻击手法也正是利用了这种机制。只要服务端提供支持，AngularJS 就可以提供安全防护。

防止 cookie 监听、中间人攻击

Preventing cookie snooping (man-in-the-middle attacks)

当通过 HTTP 在客户端和服务端之间传递数据时，第三方有可能会监听到敏感信息，甚至获取 cookie 进而伪装成你与服务端进行通信。这种方式通常称为“中间人攻击”，详见 http://en.wikipedia.org/wiki/Man-in-the-middle_attack。防止这种攻击的最简单方法就是使用 HTTPS，而不是 HTTP 协议。



任何在客户端与服务端之间传递敏感数据的应用，都应该使用 HTTPS 来保证这些数据都是加密过的。

使用 HTTPS 加密链接，可以防止敏感数据在传输过程中被获取，防止未授权用户获取已验证的 cookie 来劫持我们的会话（session）。

演示应用向 MongoLab DB 发送的请求都使用了 HTTPS 传输。为了完全防止监听，还要确保客户端与服务端的交互也是使用 HTTPS；也就是说，服务端要设置为只接收 HTTPS 请求，客户端也只发送 HTTPS 请求。

要在服务端实现这一点，取决于后端技术，这超过了本书探讨的范畴。如果是 Node.js，则可以使用 https 模块，代码如下：

```
var https = require('https');
var privateKey =
  fs.readFileSync('cert/privatekey.pem').toString();
var certificate =
  fs.readFileSync('cert/certificate.pem').toString();
var credentials = {key: privateKey, cert: certificate};
var secureServer = https.createServer(credentials, app);
secureServer.listen(config.server.securePort);
```

在客户端，只要确保连接服务端的 URL 没有指定 HTTP 协议。最简单的方法是，所有 URL 都不要使用协议，代码如下：

```
angular.module('app').constant('MONGOLAB_CONFIG', {
  baseUrl: '/databases/',
  dbName: 'ascrum'
});
```

另外，还应该限制已认证的 cookie 只能用于 HTTPS 请求。当服务端创建 cookie 时，只要设置 httpOnly 以及 secure 参数为 true。

防止跨站脚本攻击

Preventing cross-site scripting attacks

跨站脚本攻击（XSS）是指攻击者注入客户端脚本到某些用户能访问到的页面。如果注入脚本是向服务端发起请求，那么就会非常危险，因为服务端会认为这是由已认证用户发起的合法请求。

XSS 攻击有很多种方法。最常见的是，用户提供的内容没有经过正确转义来防止恶意 HTML 代码就被显示出来。下一节将介绍客户端如何实现这一点，除此之外，还应该确保任何用户提供的内容在传递给客户端之前，都是经过服务端处理的。

确保AngularJS 表达式内HTML 内容的安全性

通过 ng-bind 指令或者模板内插（{{ 花括号 }} 内的文本）显示的内容，AngularJS 都会将 HTML 符号进行转义。例如，使用以下模型：

```
$scope.msg = 'Hello, <b>World</b>!';
```

相应的标记片段如下：

```
<p ng-bind="msg"></p>
```

渲染过程会将 `` 标记进行转义，以纯文本的方式显示，如下：

```
<p>Hello, &lt;b>World&lt;/b>!</p>
```

这种方法能有效防止 XSS 攻击。如果确实要显示 HTML 标记的文本，则可以使用 `ng-bind-html-unsafe` 指令，或者加载 `ngSanitize` 模块净化(sanitize)文本，然后使用 `ng-bind-html` 指令。

允许不安全的HTML 绑定

以下绑定将渲染 `` 标记，浏览器会以 HTML 的方式进行解释：


```
<p ng-bind-html-unsafe="msg"></p>
```

净化HTML

AngularJS 还有一个指令，即 `ng-bind-html`，可以选择性地净化某些 HTML 标记，而允许其他标记被浏览器解释，用法类似 `unsafe` 指令：

```
<p ng-bind-html="msg"></p>
```

相对于转义，`ng-bind-html` 指令介于 `ng-bind-html-unsafe`（允许所有 HTML 标记）以及 `ng-bind`（不允许任何 HTML 标记）之间。当允许某些 HTML 标记由用户输入时，它是很好的选择。


 净化器（sanitization）有一个安全HTML 标记的白名单，而`<script>`、`<style>`标记还有一些以URL 为值的属性，如 `href`、`src`及`usemap`就会被过滤。

`ng-bind-html` 指令存放在独立的模块（`ngSanitize`）中，要求引入额外源文件：`angular-sanitize.js`。当需要使用 `ng-bind-html` 指令，必须声明依赖于 `ngSanitize` 模块：

```
angular.module('expressionsEscaping', ['ngSanitize'])
  .controller('ExpressionsEscapingCtrl', function ($scope) {
    $scope.msg = 'Hello, <b>World</b>!';
  });
```

ng-bind-html 指令所使用的 \$sanitize 服务也存放于 ngSanitize 模块。这个服务是一个以字符串作为参数的函数，调用时会返回净化后的字符串，代码如下：

```
var safeDescription = $sanitize(description);
```



除非你面对的是一个遗留的系统（如CMS，由后端渲染HTML等），否则都要避免在模型中使用标记。这些标记不能包含AngularJS 指令，并使用ng-bind-html-unsafe或ng-bind-html 指令来显示。

防止 JSON 注入攻击

Preventing the JSON injection vulnerability

有一个 JSON 注入漏洞可能会被第三方网站利用，只要服务端返回的是 JSON 数组，就能获取到这些重要数据。这是通过在网页中以 script 方式加载 JSON，然后执行即可实现，详见 <http://haacked.com/archive/2008/11/20/anatomy-of-a-subtle-json-vulnerability.aspx>。

\$http 服务已经内置解决方案来防止这种情况发生。为了防止浏览器执行从服务端返回的 JSON，服务器要为所有 JSON 添加 "]]}',\n" 前缀字符串，变成非法的 JavaScript 对象浏览器就无法执行。只要这个前缀字符串出现在 JSON 响应中，\$http 服务就会自动过滤。例如，返回以下数据的 JSON：

```
['a','b','c']
```

这就会存在 JSON 注入漏洞。如果服务端返回

```
]]}',
['a','b','c']
```

就不是合法的JavaScript，无法被浏览器执行，也就不存在漏洞了。只要有这个前缀，\$http服务就自动过滤掉。

防止跨站请求伪造

Preventing cross-site request forgery

用户登录后，服务端会信任并允许他执行操作，而其他网站会伪装成用户进行恶意操作。这就是所谓的跨站请求伪造（XSRF）。如果在访问恶意网站之前，你已经登录过安全网站，恶意网站就可能会向你的安全网站发送请求，而安全网站仍会认为是你在操作。

这类攻击通常会将 `` 标记的 `src` 指向目标网站，当用户登录了目标网站后，不小心访问到恶意网页，浏览器就会尝试加载图片，而实际上是向目标网站发送了请求。

```

```

解决方案是，服务端向浏览器提供加密字符串，且只能被 JavaScript 处理，而 `src` 属性是无法获取的。任何向服务端发送的请求都要在 `header` 中包含这个加密字符串才能通过校验。

`$http` 服务已经支持这种方案来防止此类攻击。服务端接收到第一个 GET 请求时，需要在 `session cookie` 中设置一个名为 `XSRF-TOKEN` 的 `token`，这个 `token` 在整个连接过程中必须是唯一的。

客户端的 `$http` 服务将从 `cookie` 获取到 `token`，随后所有的 HTTP 请求都会在 `header` 中带上该 `token`（命名为 `X-XSRF-TOKEN`）。服务端必须校验每一个请求的 `token`，一旦 `token` 无效就阻止访问。AngularJS 团队建议将认证 `cookie` 及 `salt` 的摘要值作为 `token`，以增强安全性。

7.4 客户端安全

Adding client-side security support

接下来将探讨运行在浏览器中的 AngularJS 应用如何提供安全保障，以及在认证授权时如何提供一致的用户体验。

实际上，AngularJS 并未提供处理身份认证和授权的功能。在示例应用中，我们开发了指令和提供了一些服务，用于模板以及控制器显示安全相关的信息、处理认证失败、管理用户登录以及退出。

图 7-1 包含了两个用户界面：login-toolbar 及 login-form。两者都依赖于 security 服务。login-form 的 security 服务使用 \$dialog 服务来创建模式对话框（modal dialog box）。

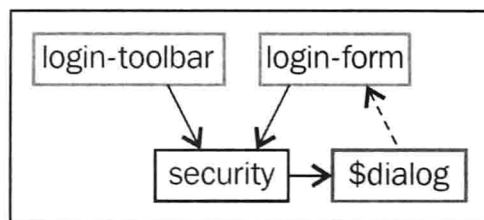


图7-1

创建 security 服务

Creating a security service

security 服务是一个组件，可为应用提供一些 API 管理用户登录退出，以及获取当前用户的信息。可以将这个服务注入控制器和指令，模板即可访问以下属性和方法。

- `currentUser`: 如果用户已登录，则此属性将包含当前用户的信息。
- `getLoginReason()`: 此方法返回本地化（localized）消息，表明需要登录的原因，例如，当前用户权限不足。
- `showLogin()`: 此方法将在用户点击登录按钮、服务端返回 HTTP 401 未授权错误时显示登录表单。
- `login(email, password)`: 此方法向服务端发送认证信息。当用户提交登录表单时需要调用，如果登录成功则会关闭表单，否则重试（再次发送请求）。
- `logout(redirectTo)`: 此方法用于用户退出，页面进行重定向。当用户点击退出按钮时调用。
- `cancelLogin(redirectTo)`: 此方法用于取消登录，所有未认证的请求将被取消，应用会跳转到其他页面。当用户关闭或取消登录表单时调用。

显示登录表单 Showing a login form

登录表单用来指引用户登录到应用（见图 7-2），它由模板（security/login/form.tpl.html）及控制器（LoginFormController）组成。当需要认证或者认证已经通过时，security 服务会打开（showLogin()）或关闭（login() 或 cancelLogin()）表单。

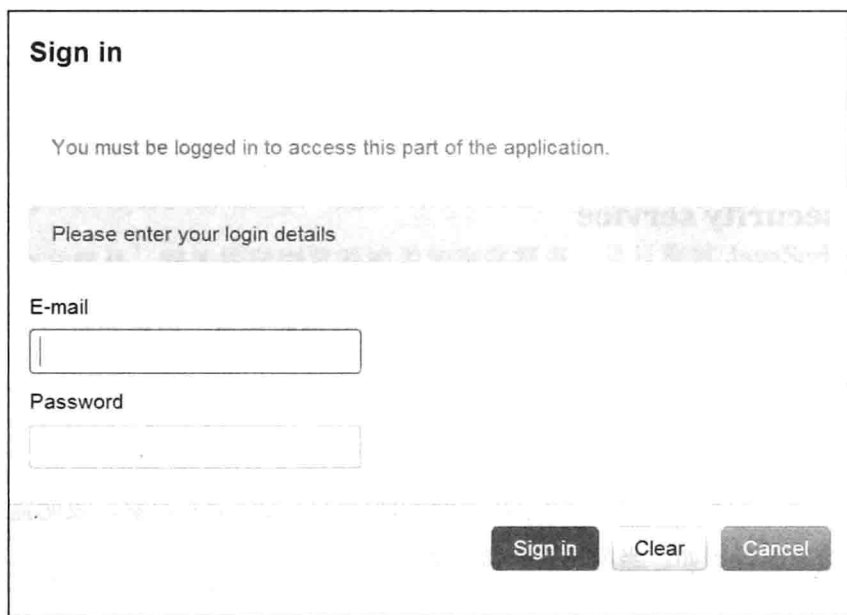


图7-2

AngularUI bootstrap 项目的 \$dialog 服务可以用于显示表单，只需为登录表单指定好模板和控制器即可。

\$dialog 服务的详细信息可参考 <http://angular-ui.github.io/bootstrap/#/dialog>。

security 服务需要两个辅助函数：openLoginDialog() 及 closeLoginDialog()。

```
var loginDialog = null;
function openLoginDialog() {
  if ( !loginDialog ) {
    loginDialog = $dialog.dialog();
    loginDialog.open(
      'security/login/form.tpl.html',
```

```

        'LoginFormController')
        .then(onLoginDialogClose);
    }
}
function closeLoginDialog(success) {
    if (loginDialog) {
        loginDialog.close(success);
        loginDialog = null;
    }
}
}

```

要打开模式对话框，只需调用 `openLoginDialog()` 并传入登录表单模板的 URL——`security/login/form.tpl.html`，以及控制器的名称——`loginFormController`。当调用 `closeLoginDialog()` 来关闭窗口时，之前返回的 `promise` 对象将被进一步处理。关闭窗口时，`onLoginDialogClose` 方法会被调用，并根据登录成功与否做一些收尾工作。

这里的模板和控制器并无特别之处，只提供一个简单的表单录入 email 地址和密码，再与 `security` 服务关联。登录按钮由 `security.login()` 负责处理，取消按钮则由 `security.cancelLogin()` 负责处理。

创建安全的菜单及工具栏

Creating security-aware menus and toolbars

为了达到良好的用户体验，用户无权限访问的功能都应该隐藏。隐藏功能并不是为了提高安全性，而仅仅是为了避免用户点击访问功能时发现权限不足的情况。常见做法是，在菜单以及工具栏选择性地显示功能。可以创建 `currentUser` 服务，以方便编写用户相应的认证状态的模板。

隐藏菜单项

应该只显示当前用户有权限访问的菜单项，如图 7-3 所示。

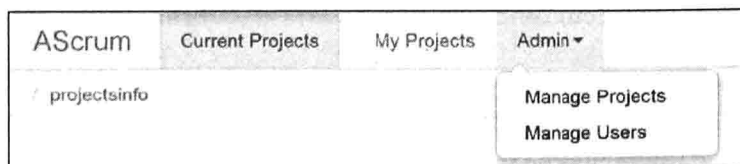


图7-3

导航菜单的元素使用 `ng-show` 指令检查当前用户是否拥有相关授权，根据检查结果决定显示或隐藏菜单。

```
<ul class="nav" ng-show="isAuthenticated()">
  <li ...><a href="/projects">My Projects</a></li>
  <li ... ng-show="isAdmin()">
    <a ... >Admin ...</a>
    <ul ...>
      <li><a ... >Projects</a></li>
      <li><a ... >Users</a></li>
    </ul>
  </li>
</ul>
```

如果用户未认证，则可以看到整个导航目录都是隐藏的。除此之外，如果用户不是管理员，则 **admin** 菜单项也是隐藏的。

创建登录工具栏

可以创建一个可复用的 `login-toolbar` 组件指令，用户登录前显示一个登录按钮，登录后则显示当前用户的昵称和一个退出按钮，如图 7-4 所示。



图7-4

以下是这个指令的模板。它会调用 `currentUser` 及 `security` 的方法，显示用户的信息，隐藏或显示登录和退出按钮。

```
<ul class="nav pull-right">

  <li class="divider-vertical"></li>

  <li ng-show="isAuthenticated()">
    <a href="#">{{currentUser.firstName}} {{currentUser.lastName}}</a>
  </li>

  <li ng-show="isAuthenticated()">
    <form class="navbar-form">
      <button class="btn" ng-click="logout()">Log out</button>
    </form>
  </li>
</ul>
```

```

</li>

<li ng-hide="isAuthenticated()">
  <form class="navbar-form">
    <button class="btn" ng-click="login()">Log in</button>
  </form>
</li>

</ul>


```

7.5 支持客户端认证

Supporting authentication and authorization on the client

保证富客户端应用（如 **AngularJS** 示例应用）的安全性，与传统的、基于服务端的应用还是有明显的不同，这取决于我们验证用户的时机和方式。

基于服务端的应用是无状态的（**stateless**），每一次交互都是向服务端发送新的请求。服务器对每次请求进行身份验证，验证失败则跳转到登录页面。

 传统的、基于服务端的web应用，只需简单地将浏览器跳转到登录页面，成功登录后再跳转回之前请求的页面。

富客户端应用交互时不会发送整个页面的请求。它们会维护自身的状态并且只向服务端发送及接收数据。服务端不会知道应用的当前状态，这种情况下，很难实现传统的登录页面跳转。客户端需要序列化（**serialize**）所有的当前状态，并发送给服务端。成功登录之后，服务端再将状态返回给客户端，客户端再继续之后的操作。

处理认证失败

Handling authorization failures

当服务端拒绝处理未认证的请求时，返回 **HTTP 401** 未认证错误，我们需提示用户登录，之后重新发送请求。这个请求可能是客户端复杂流程中的一环，且无法通过 **URL** 识别。此时，如果重定向到登录页面，那么就中断当前的操作，登录后用户可能又得重新操作一遍。

相反，在服务端将未认证的请求响应返回给调用者之前将进行拦截，暂停当前应用的业务逻辑流程，先进行认证，之后再通过重新发送失败的请求，恢复流程，如图 7-5 所示。

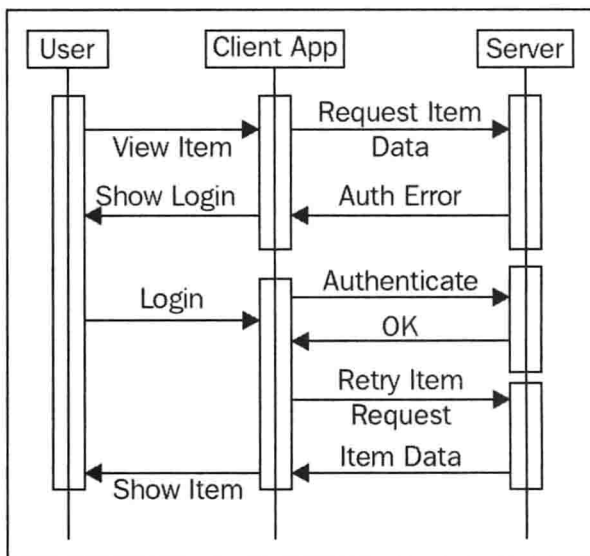


图7-5

拦截响应

Intercepting responses

还记得 `$http` 请求会返回 `promise` 吧，它在服务端返回响应后进行处理。`promise` 的强大之处在于，它可以被串联，对返回的响应数据进行转换，甚至返回一个完全不同的 `deferred` 对象。正如第 3 章（与后端服务器通信）所讲，AngularJS 允许我们创建拦截器，在服务端响应返回到最初调用者之前进行处理。

HTTP响应拦截器

响应拦截器是一个简单的函数，接收服务端响应的 `promise` 对象作为参数，最后同样会返回一个 `promise` 对象。每一个 `$http` 请求响应的 `promise` 对象都将被传递到一个个的拦截器，并在返回给最初调用者之前进行处理。

通常情况下，拦截器函数将调用 `promise` 对象的 `then()` 方法串联各个处理器（**handlers**），最后创建一个新的 `promise` 并返回。可以通过这些处理器来读取和修改 `response` 对象，如以下代码中的 `header` 及 `data`：

```
function myInterceptor(promise) {
    return promise.then(function(response) {
```

```

    if ( response.headers()['content-type'] == "text/plain") {
      response.data = $sanitize(response.data);
    };
    return response;
  });
}

```

这个拦截器会检查 response 对象的 content-type。如果值为 text/plain, 则会净化 response 对象的数据, 然后返回 response 对象。

创建 securityInterceptor 服务

Creating a securityInterceptor service

可以创建一个 securityInterceptor 服务, 与服务端响应的 promise 对象配合工作。在拦截器中将检查响应的 promise 对象是否包含被拒绝的 401 未认证错误。如果是, 则创建一个新的 promise 重试之前的请求, 以代替最初的 promise 返回给调用者。



这种做法源自于 Witold Szczerba 的一篇优秀博客: <http://www.espeo.pl/2012/02/26/authentication-in-angularjs-application>。

可以创建 securityInterceptor 作为服务, 加入 \$http 服务的 responseInterceptors 数组:

```

.config(['$httpProvider', function($httpProvider) {
  $httpProvider.responseInterceptors.push(
    'securityInterceptor');
}]);

```



请注意这里是通过名称来指定 securityInterceptor 服务, 而不是对象本身, 因为它所依赖的服务在 config 代码块中还不存在。

securityInterceptor 是以服务的方式实现的, 所以可以为它注入其他服务。



但是不能直接注入 \$http 到拦截器, 否则会导致循环依赖; 而是注入 \$injector 服务, 再通过它获取 \$http 服务。

```

.factory('securityInterceptor',
  ['$injector', 'securityRetryQueue',

```

```
function($injector, queue) {
return function(promise) {
    var $http = $injector.get('$http');
    return promise.then(null, function(response) {
        if(response.status === 401) {
            promise = queue.pushRetryFn('unauthorized-server',
                function() {return $http(response.config); }
            );
        }
        return promise;
    });
};
})
```

拦截器会监测包含 401 状态码的错误响应。它通过处理器（handler）调用 `then()` 的第二个参数实现。第一个参数 `null` 表示不对成功处理的 `promise` 对象进行拦截。

当一个请求返回 401 错误时，拦截器会在 `securityRetryQueue` 服务添加一条记录，后面再详细说明。这个服务会在登录成功后处理这条队列记录，并重试失败的请求。

值得注意的是，`promise` 处理器可以返回 `value` 或 `promise` 对象：

- 返回 `value`，这个 `value` 会直接传递给下一个处理器。
- 返回 `promise` 对象，下一个处理器会等待此 `promise` 对象已经处理（或者被拒绝）才会触发。

在以上示例中，当最初响应返回 401 未认证错误时，则会返回一个新的 `promise` 用于 `securityRetryQueue` 服务进行重试。如果服务端返回成功，则这个 `promise` 将被处理；而如果 `securityRetryQueue` 被取消或响应返回另一个错误，则 `promise` 会被拒绝。

当最初调用者耐心等待服务端返回响应时，可以弹出登录窗口，提示用户登录，最后重新发送队列中的请求。当最初调用者接收到成功的响应时，就会像一直处于已认证状态一样，继续执行下一步操作。

创建 securityRetryQueue 服务

Creating the securityRetryQueue service


当用户认证成功后, securityRetryQueue 服务会存储所有需要重试的项目。一般来说, 会是一个列表, 使用 pushRetryFn() 添加函数 (项目被重试时会调用)。通过调用 retryAll() 或 cancelAll() 来处理项目。以下是 retryAll() 方法:

```
retryAll: function() {
  while(retryQueue.length) {
    retryQueue.shift().retry();
  }
}
```

队列中所有的项目都必须包含两种方法: retry() 及 cancel()。使用 pushRetryFn() 方法很容易创建这些对象, 代码如下:

```
pushRetryFn: function(reason, retryFn) {
  var deferredRetry = $q.defer();
  var retryItem = {
    reason: reason,
    retry: function() {
      $q.when(retryFn()).then(function(value) {
        deferredRetry.resolve(value);
      }, function(value) {
        deferredRetry.reject(value);
      });
    },
    cancel: function() {
      deferredRetry.reject();
    }
  };
  service.push(retryItem);
  return deferredRetry.promise;
}
```

这个函数返回一个 promise, 用于重试提供的 retryFn 函数。当队列中的项目重试完毕时, 这个重试的 promise 对象也将被处理或者被拒绝。

 我们需要为重试的 promise 创建新的 deferred 对象, 因为它会被 retry() 或 cancel() 触发, 而不是通过服务端响应。

通知安全服务

剩下的工作是当新的项目添加进来时，要如何来通知 security 服务。我们的实现是，为 securityRetryQueue 增加新的方法 onItemAdded()，每次向队列添加项目时都会被调用：

```
push: function(retryItem) {
    retryQueue.push(retryItem);
    service.onItemAdded();
}
```

security 服务会重载这种方法，任何时候认证失败都能做出反应：

```
securityRetryQueue.onItemAdded = function() {
    if (securityRetryQueue.hasMore() ) {
        service.showLogin();
    }
};
```

以上代码可以在 security 服务找到，service 变量即为 security 服务本身。

7.6 防止导航到安全受限路由 Preventing navigation to secure routes

使用客户端代码来防止受限路由（security routes）的访问是不安全的。唯一安全的方式是，确保用户不能导航到应用的未授权页面，一旦访问就重载页面，服务端也会拒绝访问 URL。不过这种方式并不理想，富客户端的优势无法显示出来。



以重载页面的方式来保障富客户端应用的安全虽然不够理想，但是，如果应用有明确的权限区域，就会相当有用。例如，应用包含两个子应用，每个子应用都有独立的授权限制，则可以使用不同的 URL，由服务端确保用户拥有足够的权限再加载子应用。

实际上，既然我们能够决定哪些数据显示给用户，那么通过重定向的方式来阻止用户访问就显得不那么重要了。当路由改变时，客户端可以直接停止导航到未授权的路由。



这并不是安全措施，只是一种帮助用户在要求授权时进行正确认证的方式。

使用路由 `resolve` 函数

Using route resolve functions

每一个通过 `$routeProvider` 服务定义的路由都包含一系列的路由 `resolve` 函数。这些函数都会返回一个 `promise` 对象，成功导航到路由之前，这些对象必须被成功处理（**resolved**），否则取消导航。

进行身份验证最简单的方法是，提供一个路由 `resolve` 函数只在当前用户拥有足够权限时才处理成功：

```
$routeProvider.when('/admin/users', {
  resolve: [security, function requireAdminUser(security) {
    var promise = service.requestCurrentUser();
    return promise.then(function(currentUser) {
      if ( !currentUser.isAdmin() ) {
        return $q.reject();
      }
      return currentUser;
    });
  }]);
```

在这里，我们从 `security` 服务获取当前用户作为 **promise**，如果用户不是管理员，则 **promise** 会被拒绝。唯一的不足是，用户无法再进行登录认证。整个路由会被中断。

这与处理服务端 **HTTP 401** 认证错误的情况类似，也可以在导航到一个路由之前重试失败的认证请求。只要添加重试项目到 `securityRetryQueue` 服务，只要路由 `resolve` 失败，在用户登录之后就会再一次尝试处理。

```
function requireAdminUser(security, securityRetryQueue) {
  var promise = security.requestCurrentUser();
  return promise.then(function(currentUser) {
    if ( !currentUser.isAdmin() ) {
      return securityRetryQueue.pushRetryFn(
        'unauthorized-client',
```

```

        requireAdminUser);
    }
  });
}

```

现在，只要非管理员的用户想要访问某个带有 `resolve` 函数的路由，就会向 `securityRetryQueue` 服务添加一个新的项目，并触发 `security` 服务显示登录表单，用户就能以管理员身份登录。一旦登录成功，`requireAdminUser` 方法就被重试，若成功则会更新路由。

创建授权服务

Creating the authorization service

为了配合这些路由的 `resolve` 方法，要创建一个名为 `authorization` 的服务，提供检查当前用户是否具备指定权限。



在复杂一些的应用中，可以创建一个服务用于配置一系列的角色和权限，以满足应用的安全要求。

这个服务在示例应用中是非常简单的，只包含两种方法，即 `requireAuthenticatedUser()` 及 `requireAdminUser()`。

- `requireAuthenticatedUser()`：此方法返回一个 `promise`，只有在用户已经登录的情况下才会处理成功。
- `requireAdminUser()`：此方法返回一个 `promise`，只有在管理员已经登录的情况下才会处理成功。

由于这些方法存放于服务，所以在配置 `$routeProvider` 时并不可用，只能通过数组中的函数调用。

```

['securityAuthorization', function(securityAuthorization) {
  return securityAuthorization.requireAdminUser();
}]

```

为了避免产生重复代码，可以将这些数据放在一个 **provider** 中，代码如下：

```
.provider('securityAuthorization', {
  require Admin User: [
    'securityAuthorization',
    function(securityAuthorization) {
      return securityAuthorization.requireAdminUser();
    }
  ],

  $get: [
    'security',
    'securityRetryQueue',
    function(security, queue) {
      var service = {
        requireAdminUser: function() {
        },
      };
      return service;
    }
  ]
});
```

实际的认证服务在于 `$get` 属性。可以将路由 `resolve` 辅助方法，如 `requireAdminUser()`，作为 **provider** 的方法。当配置路由的时候，只需注入这个 **provider**，调用以下方法：

```
config([
  'securityAuthorizationProvider',
  function (securityAuthorizationProvider) {
    $routeProvider.when('/admin/users', {
      resolve: securityAuthorizationProvider.requireAdminUser
    });
  }
])
```

现在，路由都会在导航前检查，用户也有机会进行进一步登录认证了。

7.7 总结 Summary

首先，通过本章，我们看到了富客户端的一些安全问题，并与传统的、基于服务端的 web 应用进行比较。虽然安全检查的工作都在服务端进行，但仍然需要客户端与服务端配合才能防止恶意攻击。其次，也通过实现一些服务和指令来提高应用的安全性，了解 AngularJS 基于 promise 的 \$http 服务如何拦截未授权的请求响应，用户无须中断工作或者重启业务逻辑就可以进一步登录。最后，使用路由的 resolve 函数，在用户导航到受限区域之前检查授权。

下面将介绍如何通过开发新的指令来扩展浏览器的能力，这将允许我们以声明式的语句来开发用户界面组件。

第 8 章

创建自定义指令

Building Your Own Directives

在使用一段时间的控制器和 AngularJS 内置的指令之后，必然会进入一个新的境界，你会通过创建自己的自定义指令来教会浏览器一些新的招数。你想要开发自定义指令的原因大概如下。

- 需要像jQuery那样直接操作DOM。
- 想重构部分应用程序以删除那些重复代码。
- 想创建一些针对非开发者（如设计师）使用的新HTML标签。

本章将介绍如何开发自定义 AngularJS 指令。这些指令将会出现在应用的不同地方，并且发挥各自不同的作用。本章的主要内容有：

- 如何定义一个指令。
- 最常见的自定义指令样例，以及如何编写对应代码。
- 相互依赖的指令。
- 如何对指令进行单元测试。

8.1 什么是 AngularJS 指令

What are AngularJS directives?

指令应该是 AngularJS 最强大的功能。它们是黏合程序逻辑与 HTML DOM 的胶水。图 8-1 展示了指令如何将 AngularJS 的各层架构黏合在一起。

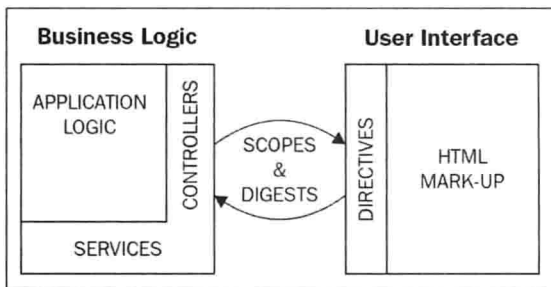


图8-1

指令通过扩展和自定义浏览器处理 HTML 的行为，从而使应用开发者或设计师可以将精力集中在应用的业务逻辑或视觉风格上。而且，指令使用的是声明式风格，而不是低级的通过编程与 DOM 进行交互。这使得开发效率更高，代码更易于维护，最重要的是开发变得更加有趣。

AngularJS 指令为 HTML 应用的标签增添了新的意义和行为。在指令内部，你可能会看到一些低级的和丑陋的代码来操作 DOM，这些代码一般都使用 jQuery 或 AngularJS 的 jqLite 编写。



如果在 AngularJS 之前加载 jQuery，那么 AngularJS 会将 jQuery 作为 DOM 操作类库。否则，AngularJS 则会假定你没有使用 jQuery，它就会实现自己的简化版 jQuery，也就是常说的 jqLite。

指令的职责是修改 DOM 结构，并将作用域和 DOM 连接起来。这说明指令既要操作 DOM，将作用域内的数据绑定到 DOM 节点上，又要为 DOM 绑定事件调用作用域内的对应方法。


理解内置指令

Understanding the built-in directives

AngularJS 框架提供了一组内置指令。内置指令明显包括那些自定义的 HTML 元素和属性，如 `ng-include`、`ng-controller` 和 `ng-click`，同时也包括标准的 HTML 元素，如 `script`、`a`、`select` 和 `input`。这些指令都是 AngularJS

核心框架默认提供的。

很赞的一点是，定义内置指令所使用的 API 与自定义指令完全一样。内置指令与开发者自定义的指令之间并无特殊区别。所以，阅读 AngularJS 中有关指令的源代码，是学习如何开发自定义指令的绝好办法。

 AngularJS 内置的指令请见 AngularJS 工程的 `src/ng/directive` 目录 (<https://github.com/angular/angular.js/tree/master/src/ng/directive/>)。

在 HTML 标签中使用指令

Using directives in the HTML markup

指令可以以 HTML 元素、属性、注释或 CSS 类等几种形式呈现。另外，任何指令都能以多种格式识别。

下面的代码展示了在 HTML 标签中使用指令的一些示例（下面的示例并非全都恰当，具体情况根据指令使用方法而定）：


```
<my-directive></my-directive>
<input my-directive>
<!-- directive: my-directive-->
<input class="my-directive">
```

在 JavaScript 中定义和调用指令时，指令名称有规范要求，必须使用驼峰式命名，如 `myDirective`。


8.2 指令的编译生命周期

Following the directive compilation life-cycle

当 AngularJS 编译一个 HTML 模板时，它会遍历浏览器提供的 DOM 树，尝试参照已注册的指令集来匹配每个元素、属性、注释及 CSS 类。每当匹配一个指令时，AngularJS 就会调用该指令的编译函数，该函数会返回一个链接函数。AngularJS 会收集所有的链接函数。


 编译工作是在作用域创建之前执行的，所以在编译函数中没有任何可用的作用域数据。

一旦所有指令都被编译完成，AngularJS 就会创建作用域，然后通过调用每个指令对应的链接函数将指令和作用域连接起来。



在连接阶段，作用域已经被附加到指令上，所以链接函数可以将作用域和DOM绑定起来。

编译阶段通常做一些优化工作。有可能指令的几乎所有工作都会在链接函数中完成（除了一些高级任务，如访问嵌入函数）。而对于重复指令（如 `ng-repeat` 内部），指令的编译函数只会被调用一次，但链接函数在每次迭代时都会被调用，每次迭代对应的数据也会随之变化。



在第9章（创建高级指令）中会详细讲解嵌入函数。

表 8-1 展示了 AngularJS 的编译器在匹配指令时所调用的编译函数。可以看到模板中每个指令所对应的编译函数只被调用了一次。


表8-1

模板	编译步骤
<code><ul my-dir></code>	myDir 编译函数
<code><li ng-repeat="obj in objs" my-dir></code>	ngRepeat 编译函数
<code></code>	myDir 编译函数
<code></code>	

表 8-2 展示了模板被转译为最终的 HTML 时所调用的链接函数。可以看到在重复指令的每一次迭代中都会调用 `myDir` 的链接函数。

表8-2

HTML	链接步骤
<code><ul my-dir></code>	myDir 链接函数
<code><!-- ng-repeat="obj in objs" --></code>	ngRepeat 链接函数
<code><li my-dir></code>	myDir 链接函数
<code><li my-dir></code>	myDir 链接函数
<code><li my-dir></code>	myDir 链接函数
<code></code>	



如果指令中有一些不依赖于作用域数据的复杂功能，那么这些功能应该放在编译函数中，这样该功能就只会被调用一次。

8.3 为指令编写单元测试

Writing unit tests for directives

指令中既可以包含初级的 DOM 操作，也可以包含复杂的功能。这使得指令容易出错且难以调试。因此，相较于应用中的其他部分，最应该对指令进行全方位的测试。

以往为指令编写单元测试代码很麻烦，但 AngularJS 提供了一些出色的特性以便让这项工作尽量简单轻松，最终你也会从测试中收获好处，你的指令是可靠且可维护的。



AngularJS 对内置的指令做了全方位的测试。这些测试代码放在 AngularJS 工程的 `test/ng/directive` 目录中 (<https://github.com/angular/angular.js/tree/master/test/ng/directive/>)。

测试指令的基本思路如下：

- 加载包含指令的模块 (module)。
- 编译一段包含指令的标签字符串以获取链接函数。
- 执行链接函数将其链接至 `$rootScope`。
- 检查页面元素是否具备了你所期望的属性。

下面是一段常见的指令单元测试的主体结构代码：

```
describe('myDir directive', function () {
  var element, scope;

  beforeEach(module('myDirModule'));


  beforeEach(inject(function ($compile, $rootScope) {
    var linkingFn = $compile('<my-dir></my-dir>');
    scope = $rootScope;
    element = linkingFn(scope);
  }));

  it('has some properties', function() {
    expect(element.someMethod()).toBe(XXX);
  });

  it('does something to the scope', function() {
    expect(scope.someField).toBe(XXX);
  });

  ...
});
```

将包含指令的模块加载到单元测试中，然后使用 `$compile` 和 `$rootScope` 函数创建一个包含指令的元素。保存一份对 `element` 和 `scope` 的引用以保证它们在后续是整个测试代码中都可用。

 根据代码编写的测试类型，可能需要在每个 `it` 分句中编译不同的元素。这种情况下还应该保存一份对 `$compile` 函数的引用。

最后，通过使用 `jQuery/jqLite` 方法与元素交互以及修改作用域数据来检查指令是否如愿执行。

如果指令中使用了 `$watch`、`$observe` 或 `$q`，那么在检查预期结果之前需要触发一下 `$digest`。示例如下：

```
it("updates the scope via a $watch", function() {
    scope.someField = 'something';
    scope.$digest();
    expect(scope.someOtherField).toBe('something');
});
```

按照测试驱动开发（TDD）的理念，在本章的剩余部分我们会通过指令的单元测试来讲解对应的自定义指令。

8.4 定义指令

Defining a directive

每个指令都必须在模块（module）上注册。定义指令的方法是在模块上调用 `directive()` 方法，调用方法时需要传递指令的标准名称（canonical name）和一个返回指令定义（directive definition）的工厂函数（factory function）。

```
angular.module('app', []).directive('myDir', function() {
    return myDirectiveDefinition;
});
```

工厂函数可以注入服务以便指令使用。

指令定义是一个对象，它包含的字段告知编译器该指令要做什么。其中有些字段是声明式的（如 `replace: true`，这个字段告知编译器使用模板替换原有的元素）。而有些字段则是命令式的（如 `link: function(...)`，这个字段为编译器提供了链接函数）。

表 8-3 中展示了指令定义中可以使用的所有字段：

表8-3

字段	描述
name	指令名称
restrict	指令可以使用哪种标签形式
priority	提示编译器该指令执行的顺序（优先级）
terminal	编译器是否在该指令之后继续编译其他指令
link	定义将指令与作用域连接起来的链接函数
template	用于生成指令标签的字符串
templateUrl	指向指令模板的 URL 地址
replace	是否使用模板内容替换指令的现有元素
transclude	是否为指令模板和编译函数提供指令元素中的内容
scope	是为指令创建一个新的子作用域，还是创建一个独立作用域
controller	一个作为指令控制器的函数
require	设置要注入当前指令链接函数中的其他指令的控制器
compile	定义编译函数，编译函数会操作原始 DOM，而且会在没有提供 link 设置的情况下创建链接函数

你编写的大多数自定义指令可能只需要其中一部分字段。在接下来的章节中，我们会展示 SCRUM 应用中使用的各种自定义指令。针对每个指令，我们都会看看它所对应的指令定义。

8.5 使用指令修改按钮样式

Styling buttons with directives

我们在示例应用中使用的是 Bootstrap CSS 库。这个 CSS 库设定了按钮的标签规范和 CSS 类名以保证按钮的样式美观。例如，一个典型的按钮使用如下标签格式：

```
<button type="submit"
      class="btn btn-primary btn-large">Click Me!</button>
```

记住这些类名既费时又容易出错。我们可以定义一个指令，让按钮的使用变得既简单又实用。

编写一个按钮指令

Writing a button directive

示例应用中的所有按钮都应该使用 Bootstrap CSS 库的按钮样式。之前我们不得不给每个按钮都追加 `class="btn"`，现在可以定义一个按钮指令来帮我们做这件事。该指令的单元测试代码如下：

```
describe('button directive', function () {
  var $compile, $rootScope;
  beforeEach(module('directives.button'));
  beforeEach(inject(function(_$compile_, _$rootScope_) {
    $compile = _$compile_;
    $rootScope = _$rootScope_;
  }));

  it('adds a "btn" class to the button element', function() {
    var element = $compile('<button></button>')($rootScope);
    expect(element.hasClass('btn')).toBe(true);
  });
});
```

在单元测试代码中，加载了对应模块，创建了按钮元素，并检查按钮是否具备正确的 CSS 类。



一定要记得注入函数会忽略围绕在参数名称两边的下划线（如 `_$compile_`）。这样就可以使用正确的名称将注入的服务拷贝至变量中（如 `$compile = _$compile_`），以便我们稍后使用。

更进一步，带有 `type="submit"` 属性的按钮应该自动使用主按钮样式，如果能通过一个专门的属性来设置按钮的尺寸就更好了。这样页面标签就很简单了：

```
<button type="submit" size="large">Submit</button>
```

应该创建对应的单元测试，代码如下：

```
it('adds size classes correctly', function() {
  var element = $compile('<button size=" large" ></button>')
($rootScope);
  expect(element.hasClass('btn-large')).toBe(true);
});

it('adds primary class to submit buttons', function() {
  var element = $compile('<button type=" submit" ></button>')
($rootScope);
  expect(element.hasClass('btn-primary')).toBe(true);
});
```

现在来看这个指令应该如何实现，代码如下：

```
myModule.directive('button', function() {
  return {
    restrict: 'E',
    compile: function(element, attributes) {
      element.addClass('btn');
      if ( attributes.type === 'submit' ) {
        element.addClass('btn-primary');
      }
      if ( attributes.size ) {
        element.addClass('btn-' + attributes.size);
      }
    }
  };
});
```



注意，我们假定已经有一个定义好的myModule模块。

我们的指令名称是 'button'，它被限制只能以元素形式出现（restrict: 'E'）。这表示该指令会在 AngularJS 编译器遇到任何按钮(button)元素时被执行。实际上，我们是给标准的 HTML button 元素追加了自定义行为。

该指令的剩余部分就是一个编译函数。该函数会在编译器匹配到按钮指令时被调用。编译函数传递了一个名为 element 的参数。这个参数是一个 jQuery（或 jqLite）对象，引用了指令所对应的 DOM 元素，在本例中就是一个按钮元素。

在编译函数中，我们只是简单地根据元素的属性值给元素追加 CSS 类。可以使用注入的属性（attributes）参数来获取元素的属性值。

在这个指令中，可以在编译函数中执行完所有的操作，完全没有使用链接函数，这是因为我们对元素的修改完全没有依赖绑定在元素上的作用域数据。也可以将这些功能全都挪到链接函数中，这样，如果按钮出现在 ng-repeat 循环中，那么每次迭代都会调用 addClass() 方法。



有关循环指令中链接函数的执行次数问题，请见“8.2 指令的编译生命周期”一节。

将指令功能放在编译函数中，只会被调用一次，而 `ng-repeat` 指令只会简单地复制按钮。如果你对 DOM 执行了非常复杂的操作，那么编译函数的这种机制就显得非常有价值，尤其是在循环次数比较多的情况下。

8.6 理解 AngularJS 的组件指令

Understanding AngularJS widget directives

指令最强大的功能之一就是能让你创建自己的领域特定标签。换句话说，你能够创建自定义的元素和属性，然后在你的应用所限定的特定领域内，为相应的 HTML 标签赋予新的语义和行为。

例如，类似于标准的 HTML 标签，你可以创建一个 `<user>` 元素来显示用户信息，或者创建一个 `<g-map>` 元素来与 Google 地图交互。这种创造的可能性是无穷无尽的，随之而来的好处是你的标签完全匹配你的开发领域。^[1]

编写一个分页指令

Writing a pagination directive

在 SCRUM 应用中，我们经常会笨拙地将大量的任务列表或待办事项一股脑地放在一页中显示。可以使用分页来将长长的列表分割成一组更易于管理的页面。在网页中使用分页模块是一种很通用的做法。

Bootstrap CSS 库提供了一个简洁美观的分页组件，如图 8-2 所示。



图8-2

接下来会将这个分页模块编写成一个可复用的组件指令，这样在使用时就无须考虑分页组件的具体细节。分页指令的标签代码将会是如下这个样子：

```
<pagination num-pages="tasks.pageCount"
             current-page="tasks.currentPage">
</pagination>
```

[1] 译者注：你的指令标签完全匹配开发场景和需求。

为分页指令编写单元测试代码

Writing tests for the pagination directive

针对该组件的测试需要考虑所有可能的变化, 这些变化可能来自 `$scope` 函数, 也可能来自用户点击分页链接。这个测试意义非凡, 下面是测试代码的重点选段:

```
describe('pagination directive', function () {
  var $scope, element, lis;
  beforeEach(module('directives'));
  beforeEach(inject(function($compile, $rootScope) {
    $scope = $rootScope;
    $scope.numPages = 5;
    $scope.currentPage = 3;
    element = $compile('<pagination num-pages="numPages" current-
page="currentPage"></pagination>')($scope);
    $scope.$digest();
    lis = function() { return element.find('li'); };
  }));

  it('has the number of the page as text in each page item',
function() {
  for(var i=1; i<=$scope.numPages;i++) {
    expect(lis().eq(i).text()).toEqual(''+i);
  }
});

  it('sets the current-page to be active', function() {
    var currentPageItem = lis().eq($scope.currentPage);
    expect(currentPageItem.hasClass('active')).toBe(true);
  });

  ...

  it('disables "next" if current-page is num-pages', function() {
    $scope.currentPage = 5;
    $scope.$digest();
    var nextPageItem = lis().eq(-1);
    expect(nextPageItem.hasClass('disabled')).toBe(true);
  });

  it('changes currentPage if a page link is clicked', function() {
    var page2 = lis().eq(2).find('a').eq(0);
    page2.click();
    $scope.$digest();
  });
});
```

```

        expect($scope.currentPage).toBe(2);
    });

    ...

    it('does not change the current page on "next" click if already at
last page', function() {
        var next = lis().eq(-1).find('a');
        $scope.currentPage = 5;
        $scope.$digest();
        next.click();
        $scope.$digest();
        expect($scope.currentPage).toBe(5);
    });

    it('changes the number of items when numPages changes', function() {
        $scope.numPages = 8;
        $scope.$digest();
        expect(lis().length).toBe(10);
        expect(lis().eq(0).text()).toBe('Previous');
        expect(lis().eq(-1).text()).toBe('Next');
    });
});

```

在指令中使用 HTML 模板

Using an HTML template in a directive

分页组件需要我们生成一些 HTML 标签来替换指令标签。最简单的办法就是为指令使用模板。分页组件的模板代码如下：

```

<div class="pagination"><ul>
  <li ng-class="{disabled: noPrevious()}">
    <a ng-click="selectPrevious()">Previous</a>
  </li>
  <li ng-repeat="page in pages"
    ng-class="{active: isActive(page)}">
    <a ng-click="selectPage(page)">{{page}}</a>
  </li>
  <li ng-class="{disabled: noNext()}">
    <a ng-click="selectNext()">Next</a>
  </li>
</ul></div>

```

该模板使用了一个名为 `pages` 的数组和一些辅助函数，如 `selectPage()` 和 `noNext()`。这些数组和辅助函数是分页组件在内部实现时需要的。它们要被放置在一个作用域内以便模板可以接收到这些内容，但它们不应该在某个具体分页组件的作用域内出现。要做到这点，可以要求编译函数为模板创建一个独立的作用域。

从父作用域中隔离指令

Isolating our directive from its parent scope

我们无法知晓在指令的具体使用场景中作用域会包含什么内容。因此，提供带有良好定义的开发接口的指令是一种很好的做法。这样可以保证指令不依赖于具体使用场景的作用域，也不会被作用域的任何属性影响。

针对在指令和模板中使用的作用域(scope)，有三个设置选项。具体定义如下。

- 复用组件具体使用位置所在的作用域。这是默认设置，对应的定义语句是 `scope: false`。
- 创建一个子作用域，该作用域原型继承自组件具体使用位置所在的作用域。对应的定义语句是 `scope: true`。
- 创建一个独立的作用域，该作用域没有原型继承，所以与其父作用域完全隔离。设置的方法是给 `scope` 属性传递一个对象：`scope: { ... }`。

要将组件模板与应用的其他部分完全解耦，这样它们之间就不存在数据泄露的风险。所以会使用独立作用域，如图 8-3 所示。

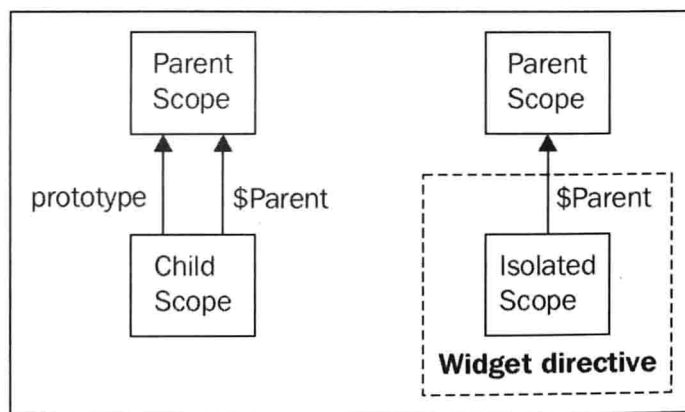


图8-3



虽然独立作用域与其父作用域不存在原型继承关系，但仍然可以通过`$parent`属性来访问父作用域。我们不建议使用该属性，因为这样做实际上是在破坏指令与周围环境的隔离关系。

现在的指令作用域从父作用域完全隔离出来了，所以需要明确指定父作用域与独立作用域之间的数据映射关系。通过在指令元素的属性上使用 AngularJS 表达式来实现这一点。在分页组件中，`num-pages` 和 `current-page` 属性担当此任。

可以通过监视来使属性表达式与模板作用域中的属性保持同步。这种同步可以通过手动设置来实现，也可以要求 AngularJS 来实现。定义元素属性与独立作用域之间的关系，共有 3 种接口：插入（@）、数据绑定（=）和表达式（&）。在指令定义中会以名 - 值对的方式来定义这些接口：

```
scope: {
  isolated1: '@attribute1',
  isolated2: '=attribute2',
  isolated3: '&attribute3'
}
```

此处为独立作用域定义了 3 个字段，AngularJS 会将具体指令元素上的特定属性映射到对应的字段上。



如果在定义字段值时省略了属性名称，那么就表示该映射属性与对应的独立作用域字段名称完全一致：

```
scope: { isolated1: '@' }
```

上句中的属性名称就是`isolated1`。

使用@插入属性

@ 符号表示 AngularJS 会将特定属性的值插入作用域，当模板属性值发生变化时，也会同步更新独立作用域中的对应属性。插入是用双花括号（`{{ }}`）包裹父作用域^[2]中的属性值，生成一个字符串来插入。

[2] 译者注：组件使用场景所在的作用域。



一种常见的误解是，以为插入的对象还是原先那个对象。而实际上插入永远返回字符串。所以，如果你有一个对象，假设就是一个具有userName字段的user对象，那么插入之后user对象就会转换为一个字符串，你无法再访问这个字符串的userName属性。

属性插入与手动 \$observe 属性效果等同：

```
attrs.$observe('attribute1', function(value) {
  isolatedScope.isolated1 = value;
});
attrs.$$observers['attribute1'].$scope = parentScope;
```

使用=绑定数据

等号(=)表示 AngularJS 会保持特定属性表达式与独立作用域属性值双向同步。这是一种双向数据绑定，允许对象和值在组件内外直接映射。



由于这种接口支持双向数据绑定，所以DOM属性中的表达式应该是可以赋值的（就是引用作用域上的字段或对象），而不是随意计算出来的表达式。

使用等号的双向数据绑定有点类似于手动设置两种 \$watch 方法：

```
var parentGet = $parse(attrs['attribute2']);
var parentSet = parentGet.assign;
parentScope.$watch(parentGet, function(value) {
  isolatedScope.isolated2 = value;
});
isolatedScope.$watch('isolated2', function(value) {
  parentSet(parentScope, value);
});
```

当然，AngularJS 中真正的实现要比上述代码复杂得多，从而可以保证两个作用域之间的稳定性。

使用&提供一个回调表达式

& 符号表示属性中设置的表达式会被当成作用域中的一个函数，当属性被调用时，该表达式函数就会执行。这个接口可以用来创建组件的回调函数。

这种绑定方式等同于 `$parse` 属性中的表达式，而且在独立作用域中暴露了表达式函数：

```
parentGet = $parse(attrs['attribute3']);
scope.isolated3 = function(locals) {
    return parentGet(parentScope, locals);
};
```

实现分页组件 Implementing the widget

分页组件的指令定义如下：

```
myModule.directive('pagination', function() {
    return {
        restrict: 'E',
        scope: {
            numPages: '=',
            currentPage: '='
        },
        template: ...,
        replace: true,
```

指令被限制以元素形式出现。组件使用独立作用域，包含 `numPages` 和 `currentPage` 两个字段，分别绑定在 `num-pages` 和 `current-page` 属性上。指令元素会被之前章节展示的模板直接替换：

```
link: function(scope) {
    scope.$watch('numPages', function(value) {
        scope.pages = [];
        for(var i=1;i<=value;i++) { scope.pages.push(i); }
        if ( scope.currentPage > value ) {
            scope.selectPage(value);
        }
    });
    ...

    scope.isActive = function(page) {
        return scope.currentPage === page;
    };

    scope.selectPage = function(page) {
        if ( ! scope.isActive(page) ) {
            scope.currentPage = page;
        }
    }
}
```

```

    };

    ...

    scope.selectNext = function() {
        if ( !scope.noNext() ) {
            scope.selectPage(scope.currentPage+1);
        }
    };
}

```

链接函数在 `numPages` 属性上设置了一个监视 (`$watch`)，并根据 `numPages` 的属性值来创建分页数组。另外，还为独立作用域增添了各种在指令模板中使用的辅助函数。

为指令添加分页跳转回调

Adding a `selectPage` callback to the directive

为组件提供一个回调函数或表达式，在分页跳转时执行，这是非常有用的特性。为了实现这点，可以为指令设计一个新属性，然后使用 `&` 将其映射到独立作用域上：

```

<pagination
  num-pages="tasks.pageCount"
  current-page="tasks.currentPage"
  on-select-page="selectPage (page)" >
</pagination>

```

上面的代码表示，只要发生分页跳转，指令就会调用 `selectPage (page)` 函数，并为其传递新的分页序号作为参数。下面是针对该新特性的单元测试：

```

it('executes the onSelectPage expression when the current page
changes', inject(function($compile, $rootScope) {
    $rootScope.selectPageHandler =
        jasmine.createSpy('selectPageHandler');
    element = $compile(
        '<pagination num-pages="numPages" ' +
            ' current-page="currentPage" ' +
            ' on-select-page=" selectPageHandler (page)" >' +
        '</pagination>')($rootScope);
    $rootScope.$digest();
    var page2 = element.find('li').eq(2).find('a').eq(0);
    page2.click();
    $rootScope.$digest();
    expect($rootScope.selectPageHandler).toHaveBeenCalled();
}));

```


在测试代码中，我们创建了一个替身（spy）来处理回调。在点击新分页时，it 函数会调用该替身方法。

为了实现指令的分页跳转回调，我们在独立作用域的定义中追加一个字段：

```
scope: {
  ...,
  onSelectPage: '&'
},
```

现在，独立作用域中就有了一个 onSelectPage() 函数。当这个函数被调用时，它就会执行 on-select-page 属性上传递的表达式。现在修改独立作用域上的 selectPage() 函数（指的是链接函数中的 selectPage 方法），使其在执行时调用 onSelectPage() 函数：

```
scope.selectPage = function(page) {
  if ( ! scope.isActive(page) ) {
    scope.currentPage = page;
    scope.onSelectPage({ page: page });
  }
};
```

 注意，我们将page变量封装成了一个对象传递给了回调表达式。这个变量在回调表达式执行时会作为参数发挥作用，就好像父作用域中的属性值一样。

8.7 创建一个自定义验证指令

Creating a custom validation directive

在 SCRUM 应用中有一个编辑用户表单。在那个表单里我们需要用户提供一个密码。由于密码输入框是做了遮蔽处理的（输入值均显示为点号或星号），用户无法看到输入内容，所以有必要再加一个密码输入框来进行密码确认。

需要检查用户的输入密码和确认密码是否完全一致。所以，我们将创建一个自定义的验证指令，该指令可以应用到 input 元素上，以检查某个 input 元素的模型值与另一个 input 的值是否匹配。实际使用时的标签代码如下：

```
<form name="passwordForm">
  <input type="password" name="password" ng-model="user.password">
  <input type="password" name="confirmPassword" ng-
model="confirmPassword" validate-equals="user.password">
</form>
```


这个自定义的模型验证器指令必须与 `ngModelController` 集成，以便为用户提供一致的验证体验。

可以通过给 `form` 和 `input` 设置 `name` 来将 `ngModelController` 暴露给作用域。这样，就可以访问控制器中的模型验证方法了。自定义的验证指令会检查 `confirmPassword` 和 `user.password` 模型值，如果一致，则会将 `confirmPassword` 元素设置为验证通过。

需要其他指令的控制器 Requiring a directive controller


验证指令需要访问 `ngModelController`，`ngModelController` 是 `ng-model` 指令的控制器。可以在指令定义的 `require` 字段中指定需要的控制器。这个字段接受字符串或字符串数组。每个字符串必须是所需控制器对应的指令名称。

当所需的指令被找到之后，该指令的控制器会被注入自定义指令的链接函数中，作为第 4 个参数：

```
require: 'ngModel',
link: function(scope, element, attrs, ngModelController) { ... }
```

如果需要多个控制器，那么第 4 个参数将会是一个由所需控制器按顺序（`require` 字段中的排列顺序）组成的数组。

[



如果当前元素不包含 `require` 字段设定的指令，那么编译函数会抛出一个错误。这是一种好办法，可以保证我们所依赖的其他指令都是存在的。


]

可选的依赖控制器

`require` 字段中的控制器可以设置为可选，方式是在指令名称前加一个 `'?'`，如 `require: '?ngModel'`。这样，如果对应的指令尚不存在，那么第 4 个参数将会是 `null`（不再报错）。如果自定义指令依赖多个外部控制器，那么注入的控制器数组中对应的值将会是 `null`。

查找祖先元素的控制器

如果允许所需要的指令控制器存在于当前元素上，或者也可以存在于当前元素的任意祖先元素上，那么可以在指令名称前加一个 '^'，如 `require: '^ngModel'`。这样，编译器就会从包含当前指令的元素开始向上查找，并返回第一个匹配的控制器。



你可以组合使用可选前缀和祖先前缀，这样，这个指令就可使可选的也存在于祖先元素中。如 `require: '^?form'`，这样就可以查找 `form` 指令的控制器，即使 `ng-model` 指令没有将自己注册到 `form` 元素上，也是可以的。

使用 `ngModelController`
Working with `ngModelController`

一旦拿到了 `ngModelController`，就可以使用它的 API 来给 `input` 元素设定验证方法。这是这类指令的通用做法，这种模式相当简洁明了。`ngModelController` 暴露了如表 8-4 所示的函数和属性以供我们使用。

表8-4

名称	描述
<code>\$parsers</code>	<code>input</code> 的值发生变化时会被依次调用的函数管线（pipeline）
<code>\$formatters</code>	模型发生变化时会被依次调用的函数管线
<code>\$setValidity(validationErrorKey, isValid)</code>	用于设置模型验证结果的函数，参数中设定了验证错误时的标识
<code>\$valid</code>	如果验证通过，则值为 <code>True</code>
<code>\$error</code>	包含验证错误信息的对象

`$parsers` 和 `$formatters` 中的函数都是接受一个输入值并返回一个输出值，类似于这样，`function(value) { return value; }`。它们接受的输入值都是管线中前一个函数的输出值。这些函数中放置验证逻辑，并会调用 `$setValidity()` 方法。^[3]

[3] 译者注：管线（pipeline）是指把若干个命令串起来，前面命令的输出成为后面命令的输入，如此完成一个流式计算。详细解释请见[http://en.wikipedia.org/wiki/Pipeline_\(software\)](http://en.wikipedia.org/wiki/Pipeline_(software))。

编写自定义验证指令的单元测试

Writing custom validation directive tests

测试验证指令的方式是编译一个表单，该表单中包含一个使用 `ng-model` 和验证指令的 `input`。示例代码如下：

```
<form name="testForm">
  <input name="testInput"
    ng-model="model.value"
    validate-equals="model.compareTo">
</form>
```

验证指令是 `input` 元素上的一个属性。该属性的值是一个表达式，与模型上对应的值相关联。验证指令会对该模型值与 `input` 中的输入值进行对比。

可以使用 `ng-model` 指令来给 `input` 绑定模型。这样，就创建了 `ngModelController`，该控制器会被暴露给 `$scope.testForm.testInput`，而 `input` 的模型值则会被暴露给 `$scope.value`。

然后，对 `input` 的值和模型值进行修改，检查 `ngModelController` 是否相应变成了 `$valid` 或 `$error`。

在测试中，可以对模型（`model`）和 `ngModelController` 分别保持一个引用。

```
describe('validateEquals directive', function() {
  var $scope, modelCtrl, modelValue;

  beforeEach(inject(function($compile, $rootScope) {
    ...
    modelValue = $scope.model = {};
    modelCtrl = $scope.testForm.testInput;
    ...
  }));

  ...
  describe('model value changes', function() {
    it('should be invalid if the model changes', function() {
      modelValue.testValue = 'different';
      $scope.$digest();
      expect(modelCtrl.$valid).toBeFalsy();
      expect(modelCtrl.$viewValue).toBe(undefined);
    });
    it('should be invalid if the reference model changes', function() {
      modelValue.compareTo = 'different';
```

```

    $scope.$digest();
    expect(modelCtrl.$valid).toBeFalsy();
    expect(modelCtrl.$viewValue).toBe(undefined);
  });
  it('should be valid if the modelValue changes to be the same as
the reference', function() {
    modelValue.compareTo = 'different';
    $scope.$digest();
    expect(modelCtrl.$valid).toBeFalsy();

    modelValue.testValue = 'different';
    $scope.$digest();
    expect(modelCtrl.$valid).toBeTruthy();
    expect(modelCtrl.$viewValue).toBe('different');
  });
});

```

在上面的代码中，我们对作用域数据做了修改，既修改了 **input** 元素本身的模型值（`modelValue.testValue`），又修改了用于对比的模型值（`modelValue.compareTo`）。之后对 **input**（`modelCtrl`）的验证结果进行了测试。在测试代码中，调用了 `$digest()` 方法以确保模型的修改同步到了 **input** 元素上。

```

describe('input value changes', function() {
  it('should be invalid if the input value changes', function() {
    modelCtrl.$setViewValue('different');
    expect(modelCtrl.$valid).toBeFalsy();
    expect(modelValue.testValue).toBe(undefined);
  });

  it('should be valid if the input value changes to be the same as
the reference', function() {
    modelValue.compareTo = 'different';
    $scope.$digest();
    expect(modelCtrl.$valid).toBeFalsy();

    modelCtrl.$setViewValue('different');
    expect(modelCtrl.$viewValue).toBe('different');
    expect(modelCtrl.$valid).toBeTruthy();
  });
});
});

```

上面的测试代码中调用了 `$setViewValue()` 方法修改 **input** 的输入值，模拟了用户在实际操作中的输入或粘贴动作。

实现自定义验证指令

Implementing a custom validation directive

现在，测试代码已经就绪，下面开始实现指令的具体功能：

```
myModule.directive('validateEquals', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ngModelCtrl) {
      function validateEqual(myValue) {
        var valid = (myValue === scope.$eval(attrs.validateEquals));
        ngModelCtrl.$setValidity('equal', valid);
        return valid ? myValue : undefined;
      }

      ngModelCtrl.$parsers.push(validateEqual);
      ngModelCtrl.$formatters.push(validateEqual);

      scope.$watch(attrs.validateEquals, function() {
        ngModelCtrl.$setViewValue(ngModelCtrl.$viewValue);
      });
    }
  };
});
```

我们创建了一个名为 `validateEqual(value)` 的函数，用于比较传入的值与属性表达式指定的值是否相等。我们将该函数压入 `$parsers` 和 `$formatters` 管线，这样，每次模型值或输入值发生变化时都会调用该验证函数。

在这个验证指令中，还要特别注意用于对比参考的模型发生变化的情况。为了考虑这种情况，我们从链接函数的 `attrs` 参数中获取了属性表达式 (`validateEquals`)，为其设置了一个监视器。当该模型发生变化时，可以通过故意调用 `$setViewValue()` 来手工触发 `$parsers` 管线。这样就确保了任何对相关模型的修改都会触发执行所有可能有关的 `$parse`。

8.8 创建一个异步模型验证器

Creating an asynchronous model validator

有些验证器只有与远程服务（比如与数据库交互）交互才能完成验证。这种情况下，远程服务返回的响应会是异步的。这不仅给模型验证增加了复杂度，也使得对应的功能测试变得更麻烦。

在管理员用户表单中，需要检查用户输入的 e-mail 地址是否已被占用。我们会创建一个 `uniqueEmail` 指令，该指令会通过后端服务来检查 e-mail 地址是否被占用：

```
<input ng-model="user.email" unique-email>
```

模拟用户服务

Mocking up the Users service

可以使用 `Users` 服务来查询数据库，以检查 e-mail 地址是否被占用。在测试中，需要模拟该服务中的 `query()` 方法。



在这个测试中，最简单的办法是创建一个测试模型，然后在其中模拟一个 `Users` 服务，而不是注入 `Users` 服务后伪造 `query()` 方法，这是因为 `Users` 服务本身还依赖于其他服务和常量。

```
angular.module('mock.Users', []).factory('Users', function() {
  var Users = { };
  Users.query = function(query, response) {
    Users.respondWith = function(emails) {
      response(emails);
      Users.respondWith = undefined;
    };
  };
  return Users;
});
```

`query()` 方法创建的 `Users.respondWith()` 方法会调用传递给 `query()` 的 `response` 回调。这样就可以在测试中模拟一个针对查询结果的响应。



在 `Users.query()` 被调用之前，以及在使用 `Users.respondWith()` 之后，都可将 `Users.respondWith` 方法设置为 `undefined`。

之后在测试中加载该模型：

```
beforeEach(module('mock.users'));
```

这样，原先的 `Users` 服务就会被模拟服务覆写。

为异步验证编写测试代码

Writing tests for asynchronous validation

异步验证的测试代码和前面的自定义验证指令的测试代码类似：

```
beforeEach(inject(function($compile, $rootScope, _Users_) {
    Users = _Users_;
    spyOn(Users, 'query').andCallThrough();
    ...
}));
```

我们伪装了 `Users.query()` 方法，而且想要它接通模拟函数以便使用 `Users.respondWith()` 来模拟最终响应。

下面展示了几个典型的单元测试：

```
it('should call Users.query when the view changes', function() {
    testInput.$setViewValue('different');
    expect(Users.query).toHaveBeenCalled();
});
```

```
it('should set model to invalid if the Users.query response
contains users', function() {
    testInput.$setViewValue('different');
    Users.respondWith(['someUser']);
    expect(testInput.$valid).toBe(false);
});
```

```
it('should set model to valid if the Users.query response
contains no users', function() {
    testInput.$setViewValue('different');
    Users.respondWith([]);
    expect(testInput.$valid).toBe(true);
});
```

在第一个单元测试中，我们检查了 `Users.query()` 是否被调用。除此之外，由于 `Users.query()` 一直在跟踪响应回调的结果，所以我们可以使用 `Users.respondWith()` 来模拟一个服务器端的响应结果。

还有一点，我们要进行专门测试，如果用户输入的值与模型中提供的值完全一致，则不需要查询服务器端。例如，如果是在修改一个用户而不是创建新用户，那么用户的 e-mail 地址肯定已经存在于服务器端的数据库中，但这个地址是可以通过验证的。

```
it('should not call Users.query if the view changes to be the same as
the original model', function() {
    $scope.model.testValue = 'admin@abc.com';
```

```

$scope.$digest();
testInput.$setViewValue('admin@abc.com');
expect(Users.query).not.toHaveBeenCalled();
testInput.$setViewValue('other@abc.com');
expect(Users.query).toHaveBeenCalled();

querySpy.reset();
testInput.$setViewValue('admin@abc.com');
expect(Users.query).not.toHaveBeenCalled();
$scope.model.testValue = 'other@abc.com';
$scope.$digest();
testInput.$setViewValue('admin@abc.com');
expect(Users.query).toHaveBeenCalled();
});

```

我们设置了模型中的值；然后检查 `User.query()` 只有在 `input` 中的值与模型的初始值不同时才会被调用。还使用 `User.query.reset()` 来重置伪造函数(spy)，以使 `User.query()` 回到未被调用过的状态。

实现异步验证指令

Implementing the asynchronous validation directive

实现异步验证指令的代码结构与之前的验证指令类似。我们需要 `ngModel` 的控制器，还需要在链接函数中使用 `$parsers` 和 `$formatters`，代码如下：

```

myModule.directive('uniqueEmail', ["Users", function (Users) {
  return {
    require: 'ngModel',
    link: function (scope, element, attrs, ngModelCtrl) {
      var original;
      ngModelCtrl.$formatters.unshift(function (modelValue) {
        original = modelValue;
        return modelValue;
      });

      ngModelCtrl.$parsers.push(function (viewValue) {
        if (viewValue && viewValue !== original) {
          Users.query({email:viewValue}, function (users) {
            if (users.length === 0) {
              ngModelCtrl.$setValidity('uniqueEmail', true);
            } else {

```



```

        ngModelCtrl.$setValidity('uniqueEmail', false);
    }
    });
    return viewValue;
}
});
}
};
}));

```

我们只在 `$parser` 方法中向服务器发送了检查请求，也就是在用户修改了 `input` 值的时候做了检查。如果 `input` 的值通过模型做了修改，则认为应用的业务逻辑保证了这是一个合法的 `e-mail` 地址。例如，加载一个用户的信息进行修改，那么该用户的 `e-mail` 地址即使在数据库中已存在，也可以通过验证。

通常在验证函数中，如果验证非法，则返回 `undefined`。这样，可以阻止模型去更新未通过验证的值。但在这个指令中，在返回的那一刻，验证函数并不知道验证结果（异步的）。所以总是返回 `input` 的值，之后让响应回调来设置最终的验证结果。

我们在 `$formatters` 管线中追加了一个函数，用来跟踪模型中设置的默认 `input` 值。这样就可以避免在用户输入与原始值一样的内容时向服务器发送请求，但此时它也会错误地将 `e-mail` 地址设置为验证未通过。

8.9 包装 jQueryUI datepicker 指令

Wrapping the jQueryUI datepicker directive


有时会用到第三方组件，它们极其复杂，短期内针对它们专门写一个纯 AngularJS 版本有些不划算。此时可以将这类组件包装成 AngularJS 指令以加快开发进度，但必须注意这两个类库之间的交互问题。

接下来看看如何将 jQueryUI datepicker 组件包装成一个 datepicker `input` 指令。该组件暴露了如表 8-5 所示的 API 接口，我们会在 AngularJS 中使用它们，其中的 `element` 指的是初始组件所用元素的 jQuery 包装器。

表8-5

方 法	描 述
<code>element.datepicker(options)</code>	使用参数中的配置创建一个新的组件，并将其绑定在元素上
<code>element.datepicker("setDate", date)</code>	设置日期
<code>element.datepicker("getDate")</code>	获取日期
<code>element.datepicker("destroy")</code>	销毁并删除组件

我们要在用户选择一个新日期时获得通知。创建组件函数中的 `options` 参数可以设置一个 `onSelect` 回调，该函数会在用户选择日期时被调用。



为了简单起见，我们设定 `datepicker` 指令在模型中只接受 JavaScript 日期对象。

包装 JQuery `input` 组件指令的模式还是与创建验证指令的模式一样。你需要引入 `ngModel`，并且要在 `$parsers` 和 `$formatters` 管线中加入函数以便在模型和视图之间转换数据。

和之前一样，我们要在模型发生变化时将数据同步到组件中显示，在组件值发生变化时将数据同步到模型中。所以，我们覆写 `ngModel.$render()` 函数来更新组件值。该函数会在 `$formatters` 中的所有函数成功执行之后被调用。而为了取出组件数据，我们使用 `onSelect` 回调，在其中调用 `ngModel.$setViewValue()` 以更新视图数据进而触发 `$parsers` 管线函数，如图 8-4 所示。

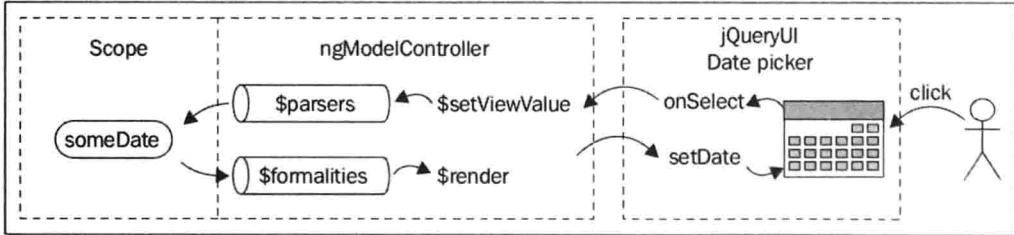


图8-4

为包装组件指令编写测试代码

Writing tests for directives that wrap libraries

在正规单元测试中，我们应该模仿一个 JQueryUI `datepicker` 组件，并暴露组件接口。但在本例中，我们将采用一种更加简单实用的方法，即直接在测试中生成一个真正的 `datepicker` 组件。


这种方法的好处是，不必在测试中逐个暴露（定义）所需的组件接口。通过直接调用组件的实际接口方法，以及检查用户界面是否正确更新，就可以确定指令是否正常运行。这种方法的不足就是组件中的 DOM 操作会影响测试的运行速度，而且必须要有一种专门方法来与组件交互以确保组件运行正常。

在本例中，jQueryUI datepicker 组件暴露的另外一种接口方法可以模拟用户选择日期：

```
$.datepicker._selectDate(element);
```

那么，我们创建一个名为 `selectDate()` 的辅助函数，用它来模拟用户选择日期的行为：

```
var selectDate = function(element, date) {
  element.datepicker('setDate', date);
  $.datepicker._selectDate(element);
};
```

 有时这种模拟很难实现，如果是这样，那么应该考虑在测试中仿造整个组件。

下面的测试代码用到了组件的 API 方法及前面的这个辅助函数：

```
describe('simple use on input element', function() {
  var aDate, element;
  beforeEach(function() {
    aDate = new Date(2010, 12, 1);
    element = $compile(
      "<input date-picker ng-model='x'/>"
    )($rootScope);
  });
  it('should get the date from the model', function() {
    $rootScope.x = aDate;
    $rootScope.$digest();
    expect(element.datepicker('getDate')).toEqual(aDate);
  });
  it('should put the date in the model', function() {
    $rootScope.$digest();
    selectDate(element, aDate);
    expect($rootScope.x).toEqual(aDate);
  });
});
```

在测试中，我们检查了模型值与组件值之间的双向同步。注意，我们在 `selectDate()` 方法之后没有调用 `$digest()`，这是因为在用户操作之后触发 `digest` 是指令本身的职责。



针对该指令的各种不同使用场景，还有很多单元测试。这些单元测试代码请见示例代码包。



实现 jQuery datepicker 指令 Implementing the jQuery datepicker directive

该指令的实现再一次利用了 `ngModelController` 提供的功能。我们向 `$formatters` 管线中加入一个函数以确保模型是一个日期 (`Date`) 对象，以及在 `options` 中加入 `onSelect` 回调。另外，还覆写了 `$render` 函数以在模型变化时更新组件。

```
myModule.directive('datePicker', function () {
  return {
    require: 'ngModel',
    link: function (scope, element, attrs, ngModelCtrl) {
      ngModelCtrl.$formatters.push(function (date) {
        if ( angular.isDefined(date) &&
            date !== null &&
            !angular.isDate(date) ) {
          throw new Error('ng-Model value must be a Date object');
        }
        return date;
      });

      var updateModel = function () {
        scope.$apply(function () {
          var date = element.datepicker("getDate");
          element.datepicker("setDate", element.val());
          ngModelCtrl.$setViewValue(date);
        });
      };

      var onSelectHandler = function (userHandler) {
        if ( userHandler ) {
          return function (value, picker) {
            updateModel();
            return userHandler(value, picker);
          };
        }
      };
    }
  };
});
```

```

    } else {
      return updateModel;
    }
  };

```

`onSelectHandler()` 函数调用了 `updateModel()` 函数，该函数通过 `$setViewValue()` 向 `$parsers` 管线中传入了一个新的日期值。

```

var setUpDatePicker = function () {
  var options = scope.$eval(attrs.datePicker) || {};
  options.onSelect = onSelectHandler(options.onSelect);
  element.bind('change', updateModel);
  element.datepicker('destroy');
  element.datepicker(options);
  ngModelCtrl.$render();
};

ngModelCtrl.$render = function () {
  element.datepicker("setDate", ngModelCtrl.$viewValue);
};

scope.$watch(attrs.datePicker, setUpDatePicker, true);
}
};
});

```

8.10 小结

Summary

在本章中，我们介绍了各种定义、测试及实现指令的方式，学习了如何集成 `ngModel` 来实现验证指令、如何编写一个封装良好可复用的组件，以及如何将第三方组件包装成 **AngularJS** 指令。本章自始至终都在强调测试的重要性并付诸实践，学习了 **AngularJS** 指令测试的基本策略。

第 9 章将深入探究 **AngularJS** 指令，学习一些有关指令的高级功能，如嵌入、编译自定义模板等。

第 9 章

创建高级指令

Building Advanced Directives

第 8 章介绍了如何开发和测试自定义指令。本章将学习一些在开发 AngularJS 指令时会用到的高级指令。本章主要内容包括以下几点。

- 理解嵌入：尤其是理解嵌入函数和嵌入作用域的用法。
- 定义自己的指令控制器来实现指令之间的相互协作，理解指令控制器与链接函数的区别。
- 终止并接管指令编译过程：动态加载自定义模板，以及使用 `$compile` 和 `$interpolate` 服务。

9.1 使用嵌入 Using transclusion

当将元素从 DOM 中的一个地方移动到另外一个地方时，必须说明与其关联的作用域如何变化。

简单直接的办法就是为该元素关联其新位置定义的新作用域。但这样可能会影响应用逻辑，因为元素无法再访问其原始作用域中的数据。

我们真正需要的是“嫁鸡随鸡，嫁狗随狗”。这种移动元素时将其关联作用域一起移动的做法就称为嵌入。我们会在本章的“理解嵌入作用域”一节中介绍如何移动作用域。现在先看几个例子。

在指令中使用嵌入 Using transclusion in directives

当一个指令的原始标签内容被新元素替换，但同时又需要在新元素的某些地方用到原始内容时，就必须使用嵌入。

例如，`ng-repeat` 指令会嵌入和克隆它的原始元素，它会按照列表迭代次数复制出多个嵌入元素的副本。每一个副本都关联了一个新作用域，该作用域是原始元素作用域的子作用域，如图 9-1 所示。

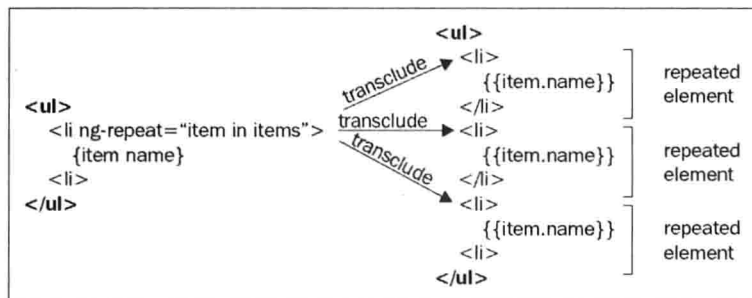



图9-1

在独立作用域指令中使用嵌入 Transcluding into an isolated scope directive

`ng-repeat` 指令比较特殊，它先克隆了自己的副本，然后才进行嵌入。嵌入最常见的用法是创建一个有模板组件的指令，然后在模板中的某个位置插入原始元素的内容。

创建一个使用嵌入的提示指令 Creating an alert directive that uses transclusion

这类使用模板组件的指令，最简单的例子就是提示（alert）指令。

[ 提示是指展示一些信息以告知用户当前应用的状态，截图如图 9-2 所示。]

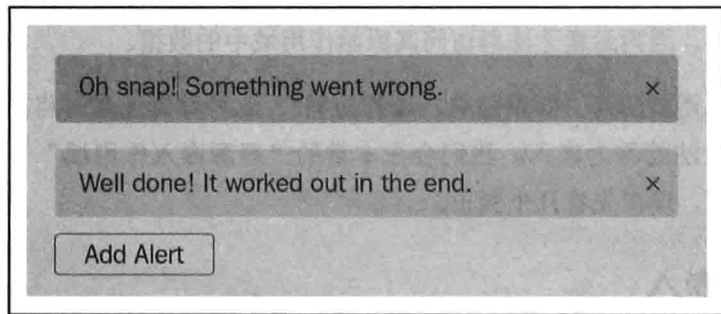


图9-2

alert 元素的内容包含在 alert 指令中显示的信息。这种情况就要将 alert 嵌入指令模板中。要显示一组提示信息，可以使用 ng-repeat 指令：

```
<alert type="alert.type" close="closeAlert($index)"
      ng-repeat="alert in alerts">
  {{alert.msg}}
</alert>
```

代码中的 close 属性包含一个表达式，该表达式会在用户点击关闭 alert 时执行。实现该指令非常简单，代码如下：

```
myModule.directive('alert', function () {
  return {
    restrict: 'E',
    replace: true,
    transclude: true,
    template:
      '<div class="alert alert-{{type}}">' +
        '<button type="button" class="close"' +
          'ng-click="close()">&times;' +
        '</button>' +
        '<div ng-transclude></div>' +
      '</div>',
    scope: { type: '=', close: '&' }
  };
});
```

理解指令定义中的replace属性

replace 属性会告诉编译器使用 template 字段所提供的模板来替换原始的指令元素。如果只提供 template 但没有定义 replace，那么编译器就会将模板追加到指令元素中。

[



当我们要求编译器使用模板替换指令元素时，编译器同时会将原始元素上的所有属性复制到模板元素上。

]

理解指令定义中的transclude属性

transclude 属性的值要么是 true，要么是 'element'。这个属性告诉编译器取出原始 <alert> 元素的内容，保证其在嵌入模板时可用。

- 使用 transclude: true 表示指令元素的内容（子元素）会被嵌入。alert 指令就是这种嵌入模式，我们使用模板替换了指令元素，然后嵌入原始元素内容。
- 使用 transclude: 'element' 表示整个元素会被嵌入，包括那些尚未被编译的属性指令。ng-repeat 指令就是这种模式。

使用ng-transclude插入嵌入元素

ng-transclude 指令会得到嵌入元素，然后将嵌入元素追加到模板元素中 ng-transclude 所在的位置。这是嵌入最简单、最常见的用法。

理解嵌入作用域

Understanding the scope of transclusion

所有经过 AngularJS 编译的 DOM 元素都会有一个与其关联的作用域。大多数情况下，DOM 元素的关联作用域都不是该元素直接定义的，而是从它们的祖先元素那里继承的。在指令定义时指定 scope 属性则会创建一个新的作用域。

[



只有少数核心指令定义了新的作用域，具体包括 ng-controller、ng-repeat、ng-include、ng-view 和 ng-switch。这些指令全部创建了从父作用域原型继承而来的子作用域。

]

我们在第 8 章中学习了如何创建具有独立作用域的组件指令，独立作用域可以确保组件内外的作用域不会相互影响。这意味着组件模板中的表达式无法访问父作用域中的值。这点非常有用，因为我们不希望父作用域中的属性和模板内容之间相互影响。



指令元素的原始内容，将会被插入模板中，它需要与指令的原始作用域关联起来，而且该作用域不是独立作用域。通过嵌入原始元素，就可以为模板元素提供一个正确的作用域。

示例中的 `alert` 指令是一个组件，使用独立的作用域。注意前面小节中 `alert` 指令创建的作用域内容。在 `alert` 指令尚未被编译之前，DOM 和作用域看起来会是这样：

```
<!--定义 $rootScope -->
<div ng-app ng-init=" type='success'" >
  <!--绑定 $rootScope -->
  <div>{{type}}</div>
  <!--绑定 $rootScope -->
  <alert type="'info'" ...>Look at {{type}}</alert>
</div>
```

`<div>{{type}}</div>` 元素没有在自己上面直接定义作用域。相反，它悄悄地绑定到了 `$rootScope` 上，因为它是 `ng-app` 元素的子元素。`ng-app` 元素定义了 `$rootScope`，所以 `{{type}}` 表达式的值是 `'success'`。

在 `aelrt` 元素上有一个属性：`type="'info'"`。该属性被映射到模板元素作用域的 `type` 属性上。一旦 `alert` 指令被编译，`alert` 元素就会被它的模板替换，编译后 DOM 和作用域如下：

```
<!--定义 $rootScope -->
<div ng-app ng-init="type='success'">
  <!--绑定 $rootScope -->
  <div>{{type}}</div>
  <!--定义独立作用域 -->
  <div class="alert-{{type}}" ...>
    <!--绑定独立作用域 -->
    <button>...</button>
    <div ng-transclude>
      <!--定义新的嵌入作用域 -->
      <span>Look at {{type}}</span>
    </div>
  </div>
</div>
```

在模板中，`class="alert-{{type}}"` 属性绑定到了独立作用域上，所以最终会被解析为 `class="alert-info"`。

相比之下，`<alert>` 元素中的嵌入内容，也就是 `Look at {{type}}`，现在被绑定到了一个新的嵌入作用域上。如果只是简单地将这句代码写在模板中，那么它所关联的作用域则会从 `$rootScope` 悄然变为独立作用域，而其中的 `{{type}}` 会被解析为 `'info'`。这并不是我们想要的结果。

相反，新的嵌入作用域是一个继承自 `$rootScope` 的子作用域。这表示 `span` 元素会被正确解析为 `Look at success`。图 9-3 展示了这些作用域在 `Batarang` 中的层级结构。

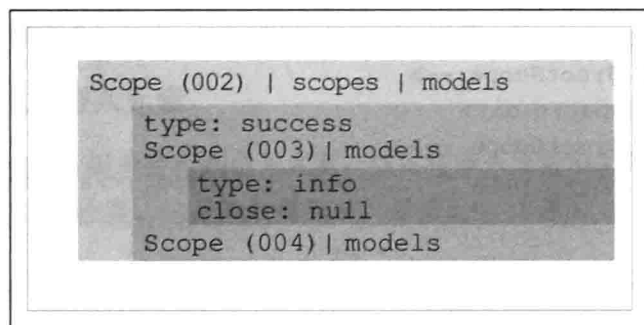


图9-3

在图 9-3 中，`Scope (002)` 是 `$rootScope`，其中包含 `type="success"`。`Scope (003)` 是一个关联 `alert` 模板的独立作用域，与 `$rootScope` 没有继承关系。`Scope (004)` 是一个继承自 `$rootScope` 的嵌入作用域，所以在该作用域中 `type` 的值是 `success`。

当元素被嵌入模板中某个位置时，它们会带着之前绑定在它们身上的原始作用域一起迁徙。更准确的说法是，嵌入元素会绑定一个新的作用域，而该作用域原型继承自该元素当初诞生地所在的作用域。

9.2 创建和使用嵌入函数

Creating and working with transclusion functions

在 `AngularJS` 中也可以使用嵌入函数（`transclusion functions`）来实现嵌入。嵌入函数实际上就是通过调用 `$compile` 服务创建的链接函数。

当一个指令要求嵌入时，AngularJS 就会从 DOM 中提取对应的嵌入元素，然后对它进行编译。下面的代码展示了在 `transclude:true` 时 AngularJS 的简要处理过程：

```
var elementsToTransclude = directiveElement.contents();
directiveElement.html('');
var transcludeFunction = $compile(elementsToTransclude);
```

第一行代码获取到了请求嵌入的指令元素的内容。第二行代码清空了指令元素。第三行代码编译了嵌入内容以生成一个嵌入函数，该函数会返回给指令以供其使用。

使用 `$compile` 服务创建一个嵌入函数


Creating a transclusion function with the `$compile` service

AngularJS 的编译器对外以 `$compile` 服务的形式暴露出来。该服务就是 AngularJS 用来编译应用的编译函数。使用该服务很简单，只需在调用时传递一个 DOM 节点列表（或者一个可被解析为 DOM 节点的字符串）作为参数即可。

```
var linkingFn = $compile(
  '<div some-directive>Some {{ "interpolated" }} values</div>');
```

调用 `$compile` 服务会返回一个链接函数。调用该链接函数，传递一个作用域作为参数，则会生成一个编译之后绑定在指定作用域上的 DOM 元素：

```
var compiledElement = linkingFn(someScope);
```

[ 嵌入函数实际上只是链接函数的一个特殊实例。]

在嵌入时克隆原始元素

如果在调用链接函数时传递一个回调函数，那么链接函数不会像上节那样返回编译后的元素，而是返回该元素的一个克隆副本。同时，回调函数也会被传入的克隆元素作为参数进行调用。

```
var clone = linkingFn(scope, function callback(clone) {
  element.append(clone);
});
```



如果你想对原始元素的子元素制作副本，这个特性就很有用，`ng-repeat`指令就做了这样的处理。

在指令中访问嵌入函数

Accessing transclusion functions in directives

编译器会将嵌入函数传回给指令。有两个地方可以访问嵌入函数：编译函数中和指令控制器中（本章后面有一节专门详细讲解了有关指令控制器的内容）。

```
myModule.directive('myDirective', function() {
  return {
    transclude: true,
    compile: function(element, attrs, transcludeFn) { ... };
    controller: function($scope, $transclude) { ... },
  };
});
```

上面的代码中，我们声明了指令应该使用嵌入（`transclude: true`）。可以通过 `transcludeFn` 参数来访问编译函数中的嵌入函数，也可以通过 `$transclude` 参数访问指令控制器中的嵌入函数。

通过编译函数中的 `transcludeFn` 来获取嵌入函数

可以通过指令编译函数的第三个参数来获取嵌入函数。在编译阶段，因为作用域尚未确定，所以嵌入函数并没有绑定任何作用域。因此，当调用嵌入函数时，需要将作用域作为第一个参数传递给它。

在链接函数中作用域已经确定，因此，一般情况下，嵌入函数都会在链接函数中被调用。

```
compile: function(element, attrs, transcludeFn) {
  return function postLink(scope, element, attrs, controller) {
    var newScope = scope.$parent.$new();
    element.find('p').first().append(transcludeFn(newScope));
  };
}
```

我们将嵌入元素插入了指令元素中的第一个 `<p>` 元素中。当调用嵌入函数（代码中的 `transcludeFn`）时，我们将嵌入元素绑定到了一个作用域上。本例创建了一个新作用域，该作用域是指令父作用域的子作用域，也就是指令作用域的兄弟作用域。


当指令使用独立作用域时，像上面代码中那样给嵌入函数传递作用域是必需的。因为此时传递给链接函数的作用域是指令的独立作用域，该作用域并没有继承父作用域的属性，而嵌入元素恰恰需要这些属性，所以需要给嵌入函数专门传递作用域。

通过 `$transclude` 在指令控制器中获取嵌入函数

可以通过向指令控制器注入 `$transclude` 来获取嵌入函数。在这种模式下，`$transclude` 是一个已经预先绑定了新作用域（新作用域是嵌入元素父作用域的子作用域）的函数，所以调用时无需提供作用域。

```
controller: function($scope, $element, $transclude) {
  $element.find('p').first().append($transclude());
}
```

在控制器中，再一次将嵌入元素插入了指令元素中的第一个 `<p>` 中。

 使用 `$transclude` 时，预先绑定的作用域是原型继承自嵌入元素父作用域的子作用域。

使用嵌入创建一个 if 指令

Creating an if directive that uses transclusion

接下来看一个真正使用嵌入函数而没有依赖 `ng-transclude` 的简单指令的例子。在 **AngularJS 1.0** 中同时提供了 `ng-show` 和 `ng-switch` 指令用于改变元素的可视状态，`ng-show` 在隐藏元素时不会将其从 **DOM** 中删除，而 `ng-switch` 指令针对一些简单情形则显得太麻烦。

如果只是想在不需要某个元素时将其从 **DOM** 中移除，那么可以创建一个 `if` 指令。该指令的用法与 `ng-show` 类似，代码如下：

```
<body ng-init="model= {show: true, count: 0}">
  <button ng-click="model.show = !model.show">
    Toggle Div
  </button>
```

```

<div if="model.show" ng-init="model.count=model.count+1">
  Shown {{model.count}} times
</div>
</body>

```

如上所示，每次点击按钮，`model.show` 的值就会在 `true` 和 `false` 之间切换。为了展示 DOM 元素会在每次切换时被删除和被重新插入，我们每次都增加 `model.count` 的值以示区别。

在单元测试中，要测试 DOM 元素是否确实被正确地删除和插入，代码如下：

```

it('creates or removes the element as the if condition changes',
function () {
  element = $compile(
    '<div><div if=" someVar" ></div></div>')(scope);
  scope.$apply('someVar = true');
  expect(element.children().length).toBe(1);
  scope.$apply('someVar = false');
  expect(element.children().length).toBe(0);
  scope.$apply('someVar = true');
  expect(element.children().length).toBe(1);
});

```

在测试代码中，我们检测了容器元素（包含 `if` 指令元素的 `div`）的子元素个数，看它是否根据模型值的变化同步变为 0 或 1。



注意，使用一个 `div` 将包含 `if` 指令的元素包裹起来，因为指令会使用 `jqLite.after()` 方法将元素插入 DOM 中，该方法需要目标元素有一个父元素。

我们来看看如何实现这个指令，代码如下：

```

myModule.directive('if', function () {
  return {
    transclude: 'element',
    priority: 500,
    compile: function (element, attr, transclude) {
      return function postLink(scope, element, attr) {
        var childElement, childScope;

        scope.$watch(attr['if'], function (newValue) {
          if (childElement) {

```




```

        childElement.remove();
        childScope.$destroy();
        childElement = undefined;
        childScope = undefined;
    }
    if (newValue) {
        childScope = scope.$new();
        childElement = transclude(childScope, function(clone) {
            element.after(clone);
        });
    }
});
...

```

`if` 指令嵌入了整个元素 (`transclude: 'element'`)。我们提供了一个编译函数, 在编译函数中可以获取嵌入函数, 该编译函数最终会返回一个监视 (`$watch`) `if` 属性表达式的链接函数。

 我们是使用 `$watch` 而不是使用 `$observe` 来执行监视, 是因为 `if` 的属性值不仅可能是字符串, 还有可能是表达式。

当表达式的值发生变化时, 如果指令对应的作用域和子元素都已存在, 就要对它们进行清理, 这样可以保证不产生内存泄露。如果表达式的值解析为 `true`, 那么就创建一个新的子作用域, 然后使用嵌入函数 (新创建的作用域作为参数) 来克隆一个嵌入元素的新副本, 最后将克隆元素插入到指令元素的后面。

在指令中使用 `priority` 属性

所有指令都有优先级, 默认值为 `0`, 前面例子中的 `alert` 指令就是默认优先级。在每个元素上, **AngularJS** 会先按照优先级从高到低依次编译指令。可以在指令定义对象上使用 `priority` 属性来设定优先级。

如果一个指令设定为 `transclude: 'element'`, 那么编译器只会嵌入那些指令优先级比当前指令优先级低的属性, 换句话说, 只会嵌入那些尚未被处理过的指令。

ng-repeat 指令设定了 transclude: 'element', 而且优先级为 1000 (priority: 1000), 所以基本上出现在 ng-repeat 元素上的所有属性都会被嵌入新的克隆元素副本上。



我们为 if 指令设定的优先级是 500, 比 ng-repeat 的优先级低。这意味着如果将 if 指令与 ng-repeat 指令放在同一个元素上, 那么 if 指令监视的表达式所关联的作用域就是 ng-repeat 每次迭代生成的作用域。

在这个 if 指令中, 嵌入特性可以帮助我们控制指令元素的内容, 绑定正确的作用域, 并按条件将元素插入 DOM 中。

接下来将改变方向, 去看看如何为指令提供定制的控制。

9.3 理解指令控制器 Understanding directive controllers

AngularJS 中的控制器是一个附属于 DOM 元素的对象, 主要用于初始化工作及为元素所在的作用域添加行为。



前面已经介绍过很多由 ng-controller 实例化而来的应用控制器。这些控制器不会和 DOM 直接交互, 而是只处理与当前作用域的交互。

指令控制器是控制器的一种特殊形式, 它由某个指令定义而来, 每当指令在 DOM 中出现时, 该控制器就会实例化一次。指令控制器的职责是负责初始化工作并为指令 (而不是作用域) 提供交互行为。

可以使用指令定义对象中的 controller 对象来定义一个指令控制器。controller 属性的值可以是一个字符串, 该字符串是已经在模块上定义好的控制器的名称, 代码如下:

```
myModule.directive('myDirective', function() {
  return {
    controller: 'MyDirectiveController'
  };
});
myModule.controller('MyDirectiveController', function($scope) {
  ...
});
```

`controller` 属性也可以是一个用来实例化控制器的构造函数，代码如下：

```
myModule.directive('myDirective', function() {
  return {
    controller: function($scope, ...) { ... }
  };
});
```



如果在模块上单独定义指令控制器（上面第一段代码），那么可以脱离指令单独对该控制器进行测试。但这同时意味着该控制器暴露给了整个应用，可以通过注入器随意使用，所以需要注意该控制器的名称不能与应用中其他模块的控制器名称冲突。

而在指令定义中以匿名函数的方式定义指令控制器（上面第二段代码），这样会使得脱离指令对控制器单独进行测试变得极其困难，但同时这种方式保证了该控制器是指令私有的。

为指令控制器注入特殊依赖

Injecting special dependencies into directive controllers

和控制器一样，AngularJS 为指令控制器注入了依赖关系。所有控制器都被注入了 `$scope`，你还可以定义其他需要被注入的服务，如 `$timeout` 或 `$rootScope`。除了上面这些，指令控制器还可以注入如下三个特殊的服务。

- `$element`：对指令DOM元素的引用，是一个被jQuery包裹的对象。
- `$attrs`：出现在指令DOM元素上的属性列表。
- `$transclude`：绑定在当前作用域上的嵌入函数，即前面小节中所讲的嵌入函数。

创建一个基于控制器的分页指令

Creating a controller-based pagination directive

指令控制器和链接函数在功能上有很大的重叠。我们经常使用控制器来替代链接函数。接下来将编写一个分页指令（第8章（使用链接函数）实现过一个分页指令），这次我们使用指令控制器来替代链接函数，代码如下：

```
myModule.directive('pagination', function() {
  return {
    restrict: 'E',
    scope: { numPages: '=', currentPage: '=', onSelectPage: '&' },
    templateUrl: 'template/pagination.html',
    replace: true,
    controller: ['$scope', '$element', '$attrs',
      function($scope, $element, $attrs) {
        $scope.$watch('numPages', function(value) {
          $scope.pages = [];
          for(var i=1; i<=value; i++) {
            $scope.pages.push(i);
          }
          if ( $scope.currentPage > value ) {
            $scope.selectPage(value);
          }
        });
        $scope.noPrevious = function() {
          return $scope.currentPage === 1;
        };
        ...
      }
    ]
  };
});
```

在这个简单的例子中，使用链接函数和使用控制器的唯一区别是，链接函数被传入了 `scope`、`element`、`attrs` 和 `controller` 这几个参数，而指令控制器必须使用依赖注入来提供 `$scope`、`$element` 和 `$attrs` 服务。

理解指令控制器和链接函数的区别

Understanding the difference between directive controllers and link functions

当在选择使用链接函数或指令控制器时，理解它们之间的区别很有帮助。

注入依赖

首先，正如上一节所见，指令控制器必须使用依赖注入来指定它所需要的服务，如 `$scope`、`$element` 和 `$attrs`。而链接函数则永远被传入 4 个相同的参数：`scope`、`element`、`attrs` 和 `controller`，定义链接函数时所用的参数名称对此毫无影响。

编译过程

指令控制器和链接函数是在编译过程中的不同时间被调用的。假设在 DOM 元素上使用了一些指令，具体结构如图 9-4 所示。

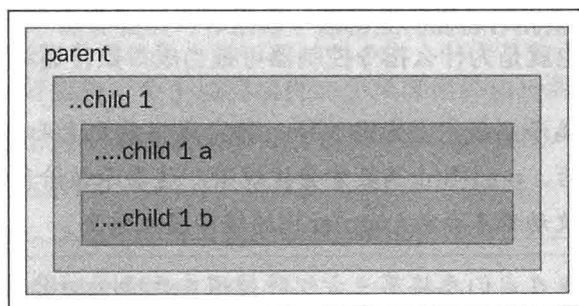


图9-4

指令控制器和链接函数会按如下顺序被调用：

- `parent (controller);`
- `parent (pre-link);`
 - `child 1 (controller),`
 - `child 1 (pre-link),`
 - `child 1 a (controller),`
 - `child 1 a (pre-link),`
 - `child 1 a (post-link),`
 - `child 1 b (controller),`
 - `child 1 b (pre-link),`
 - `child 1 b (post-link),`
 - `child 1 (post-link),`
- `parent (post-link)。`

如果一个元素上包含多个指令，那么编译顺序如下：

- 按需创建作用域；
- 实例化每个指令的指令控制器；
- 调用每个指令的预链接函数（pre-link function）；
- 所有子元素完成链接；
- 调用每个指令的链接函数（post-link function）。

这意味着当一个指令控制器实例化时，该指令的指令元素和子元素都尚未被完全链接。但当链接函数（pre 或 post）被调用时，该元素的所有指令控制器均已完成实例化。这也就是为什么指令控制器可被当成参数传递给链接函数的原因。



在编译函数完全完成编译，并完成当前元素和子元素的链接之后，post-link函数才会被调用。这表示该阶段对DOM的任何改动都不会被AngularJS的编译器注意到。

这点在我们要将第三方组件接通在元素上时非常有用，比如在元素上初始一个jQuery插件，这种插件会对DOM做一些修改，而这些修改会让AngularJS编译器感到困惑。在post-link函数中执行这些操作则可避免这个问题。

获取其他控制器

链接函数的第4个参数包含了指令所需要（require 属性指定）的所有指令控制器。我们来看看链接函数是如何获取 ngModelController 的，代码如下：

```
myModule.directive('validateEquals', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ngModelCtrl) {
      ...
    };
  });
});
```

validateEquals 指令需要使用 ngModel 的指令控制器，然后该控制器通过 ngModelCtrl 参数传递给了链接函数。

与此相反，指令控制器本身无法获取被注入当前指令中的其他指令控制器。

获取嵌入函数

我们在有关嵌入函数的小节中已经介绍过，指令控制器可以被注入一个 `$transclusion` 函数，该嵌入函数已经绑定了正确的作用域。

链接函数只能通过编译函数的闭包来访问嵌入函数，而该嵌入函数尚未绑定作用域。

创建一个手风琴指令套件

Creating an accordion directive suite

指令控制器为指令元素追加行为，同时它也可以被其他指令引入（在链接函数中通过参数引入）。这让我们可以创建一组相互沟通协作的指令套件。

本节将学习如何实现一个手风琴组件。手风琴组件由一组头部可点击的可折叠内容块组成。点击其中某块的头部可以展开对应的内容区，同时其他块的内容区折叠，如图 9-5 所示。

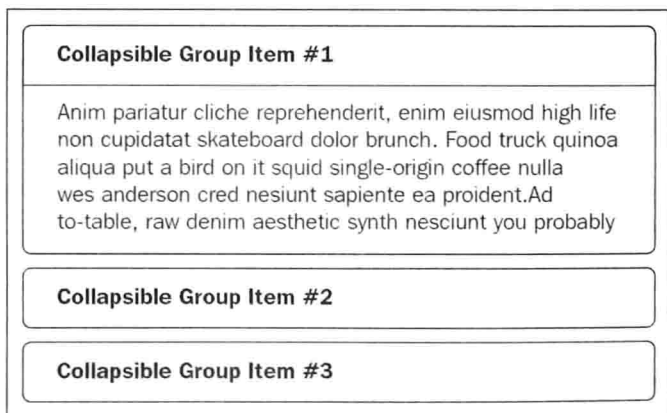


图9-5

使用手风琴组件的 HTML 代码如下：

```
<accordion>
  <accordion-group heading="Heading 1">
    Group 1 <strong>Body</strong>
  </accordion-group>
  <accordion-group heading="Heading 2">
    Group 2 <strong>Body</strong>
  </accordion-group>
</accordion>
```

如上代码所示，我们创建了两个新的元素指令：作为手风琴容器的 `accordion`，以及定义每个内容区块的 `accordion-group`。

在手风琴组件中使用指令控制器

为了让手风琴中的各个模块相互沟通，`accordion` 指令定义了一个名为 `AccordionController` 的指令控制器。每一个 `accordion-group` 指令都需要引入这个控制器。

`AccordionController` 指令控制器会暴露两种方法：`addGroup` 和 `closeOthers`。`accordion-group` 指令会使用 `addGroup` 来将它自己注册为手风琴的一部分，然后在自己展开时使用 `closeOthers` 来告知其他 `accordion-groups` 折叠起来。

测试指令控制器和测试应用控制器非常相似，控制器的测试内容具体请见第 2 章。下面是针对 `closeOthers` 方法的测试：

```
describe('closeOthers', function() {
  var group1, group2, group3;
  beforeEach(function() {
    ctrl.addGroup(group1 = $scope.$new());
    ctrl.addGroup(group2 = $scope.$new());
    ctrl.addGroup(group3 = $scope.$new());
    group1.isOpen = group2.isOpen = group3.isOpen = true;
  });
  it('closes all groups other than the one passed', function() {
    ctrl.closeOthers(group2);
    expect(group1.isOpen).toBe(false);
    expect(group2.isOpen).toBe(true);
    expect(group3.isOpen).toBe(false);
  });
});
```

我们在测试中添加了 3 个内容区块，然后将它们的 `isOpen` 全部设置为 `true`。在调用 `closeOthers(group2)` 之后，检测 `group1` 和 `group3` 的 `isOpen` 是否已被设置为 `false`。

下面展示了 `AccordionController` 的实现代码：

```
myModule.controller('AccordionController', ['$scope', '$attrs',
function ($scope, $attrs) {
  this.groups = [];
  this.closeOthers = function(openGroup) {
    angular.forEach(this.groups, function (group) {
      if ( group !== openGroup ) {
        group.isOpen = false;
      }
    });
  };
}
```



```

    });
    this.addGroup = function(groupScope) {
        var that = this;
        this.groups.push(groupScope);
        groupScope.$on('$destroy', function (event) {
            that.removeGroup(groupScope);
        });
    };
    this.removeGroup = function(group) {
        var index = this.groups.indexOf(group);
        if ( index !== -1 ) {
            this.groups.splice(this.groups.indexOf(group), 1);
        }
    };
    });
});

```

注意，我们在手风琴区块（**accordion-group**）对应的作用域被销毁时自动从数组中删除了区块。这点非常重要，因为手风琴的内容区块（**groups**）可能是在运行时由 **ng-repeat** 指令生成的，这些生成的元素和对应的作用域可能会被删除。如果一直保持对这些内容区块作用域的引用，那它们就无法被垃圾回收机制回收。

实现accordion指令

accordion 指令的实现只是定义了 **AccordionController** 作为它的指令控制器，然后在链接函数中为指令元素追加了名为 **accordion** 的 **CSS** 类，代码如下：

```

myModule.directive('accordion', function () {
    return {
        restrict: 'E',
        controller: 'AccordionController',
        link: function(scope, element, attrs) {
            element.addClass('accordion');
        }
    };
});

```

实现accordion-group指令

每一个手风琴内容区块都会由一个 **accordionGroup** 指令来定义。每个区块都由一个标题超链接和一个内容区块组成。该指令的模板如下：

```

<div class="accordion-group">
  <div class="accordion-heading" >
    <a class="accordion-toggle"

```

```

        ng-click="isOpen=!isOpen">{{heading}}</a>
    </div>
    <div class="accordion-body" ng-show="isOpen">
        <div class="accordion-inner" ng-transclude></div>
    </div>
</div>

```

我们将原始指令元素的子元素嵌入了模板中。模板中引用了当前作用域中的 `isOpen` 和 `heading`。想要完全控制这些模型值，那么 `accordion-group` 指令就要拥有独立作用域。

在测试中，我们会设置一个 `accordion` 和几个 `accordion-group` 指令，然后检查手风琴是否能正确地展开和折叠。单元测试的代码示例如下：

```

describe('accordion-group', function () {
    var scope, element, groups;
    beforeEach(inject(function($rootScope, $compile) {
        scope = $rootScope;
        var tpl =
            "<accordion>" +
            "<accordion-group heading='title 1'>Content 1</accordion-group>" +
            "<accordion-group heading='title 2'>Content 2</accordion-group>" +
            "</accordion>";
        $compile(tpl)(scope);
        scope.$digest();
        groups = element.find('.accordion-group');
    });
    ...
    it('should change selected element on click', function () {
        groups.eq(0).find('a').click();
        expect(findGroupBody(0).scope().isOpen).toBe(true);
        groups.eq(1).find('a').click();
        expect(groups.eq(0).scope().isOpen).toBe(false);
        expect(groups.eq(1).scope().isOpen).toBe(true);
    });
    ...
});

```

首先在测试中触发某个区块的点击事件，然后检查其他区块的作用域中的 `isOpen` 是否为 `false`。

`accordion-group` 指令的实现非常简单，如下：

```

myModule.directive('accordionGroup', function() {
    return {
        require: '^accordion',

```

```

restrict: 'E',
transclude: true,
replace: true,
templateUrl: 'template/accordion/accordion-group.html',
scope: { heading: '@' },
link: function(scope, element, attrs, accordionCtrl) {
    accordionCtrl.addGroup(scope);
    scope.isOpen = false;
    scope.$watch('isOpen', function(value) {
        if ( value ) {
            accordionCtrl.closeOthers(scope);
        }
    });
}
});
});

```

从以上代码中可以看出，`accordion-group` 指令需要 `accordion` 指令的指令控制器，`accordion` 在 DOM 结构上是 `accordion-group` 指令的祖先。`accordion` 的指令控制器作为第 4 个参数，命名为 `accordionCtrl`，传递给了链接函数。`accordion-group` 指令使用 `accordionCtrl` 的 `addGroup()` 函数将自己注册在手风琴上，然后在自己展开时调用 `closeOthers()` 来处理其他区块。

9.4 控制编译过程

Taking control of the compilation process

有些情况下，需要对 AngularJS 的编译过程与如何连接元素及其子元素有更多的控制。也许我们是想动态地加载指令模板，或者想对元素嵌入指令模板有更多的控制。这些情况下我们可以终止编译过程，然后进行自定义修改，之后手动编译指令元素及其子元素。

创建一个 field 指令

Creating a field directive

在开发含有大量表单的应用时，很快就会发现为每个表单条目都写了大量的 HTML 代码，这其中大量的重复代码和冗余代码。

例如，针对每个表单条目都会有一个 `input` 元素和一个 `label` 元素，这些元素都会被各种 `div` 或 `span` 元素包裹起来。在这些表单元素上，我们要提供一组属性，`ng-model`、`name`、`id` 和 `for`，这些属性通常都很相似甚至完全相同。当然，还有各种 CSS 类。另外，还要在 `input` 值未通过验证时显示验证提示信息。

于是就出现了这样的满篇重复的 HTML：

```
<div class="control-group" ng-class="{ 'error' : form.email.$invalid &&
form.email.$dirty, 'success' : form.email.$valid && form.
email.$dirty}">
  <label for="email">E-mail</label>
  <div class="controls">
    <input type="email" id="email" name="email" ng-model="user.email"
required>
    <span ng-show="form.email.$error['required'] && form.email.dirty"
class="help-inline">Email is required</span>
    <span ng-show=" form.email.$error['email'] && form.email.dirty"
class="help-inline">Please enter a valid email</span>
  </div>
</div>
```

我们可以创建一个 `field` 指令来消除大部分无用的重复。这个指令会插入合适的模板，其中包含带有 `label` 标签的 `input` 输入框。下面是使用 `field` 指令的示例代码：

```
<field type="email" ng-model="user.email" required >
  <label>Email</label>
  <validator key="required">$fieldLabel is required</validator>
  <validator key="email">Please enter a valid email</validator>
</field>
```

现在，我们只需要简单地以属性的形式为 `field` 指令提供 `ng-model`、`type` 和针对 `input` 的验证指令即可。然后将 `label` 和验证信息作为 `field` 元素的子元素。请注意，在验证信息中还可以使用 `$fieldLabel` 属性。`field` 指令会将该属性添加到验证信息所属的作用域上。

使用 **AngularJS** 内置的指令 API 很难实现 `field` 指令，存在如下几个问题。

- 不同于以往我们为整个指令提供一个固定的模板，现在需要根据表单元素的不同类型来插入不同的模板。无法通过在指令定义中使用 `template`（或 `templateUrl`）属性来实现这点。
- 在 `ng-model` 指令编译之前，我们需要为 `input` 生成唯一的 `name` 和 `id` 属性，并将该属性与 `label` 元素的 `for` 属性关联起来。
- 想在表单值有错误时从 `field` 指令的子元素中获取验证信息并用于模板中。

field指令的指令定义对象如下：

```
restrict: 'E',
priority: 100,
terminal: true,
compile: function(element, attrs) {
  ...
  var validationMgs = getValidationValidationMessages(element);
  var labelContent = getLabelContent(element);


  element.html('');

  return function postLink(scope, element, attrs) {
    var template = attrs.template || 'input.html';
    loadTemplate(template).then(function(templateElement) {
      ...
    });
  };
}
```

在field指令中我们终止了编译过程，将指令的优先级设为100，以保证该指令会在ng-model指令之前运行。在编译函数中，我们通过getValidationValidationMessages方法获取了验证信息，通过getLabelContent方法获取了label信息。在找到这些信息之后，就清空元素的内容，以便有一个干净的元素来加载模板。compile函数最终返回了一个postLink函数，该函数会为指令加载合适的模板。

在指令中使用terminal属性

如果一个指令设定为terminal: true，那么它的编译器就会停止，不会再编译该指令元素的子元素，也不会编译该指令元素上的优先级低于该指令的其他指令。

 即使你终止了一个指令，但该指令的指令控制器、编译函数和链接函数均已执行。

一旦终止了指令的编译过程，就可以对指令元素及其子元素进行修改，但同时也必须负责设定新作用域，正确地嵌入内容，以及更进一步地负责编译可能包含指令的子元素。

大多数时候，也就是在模板中使用 `{{}}` 时，AngularJS 会负责将字符串插值置入表达式中。但在 `field` 指令中，我们需要编程实现字符串插值。使用 `$interpolate` 服务可以实现这点。

使用 `$interpolate` 服务

Using the `$interpolate` Service

`getLabelContent` 方法只是简单地将指令元素中 `label` 的 HTML 内容拷贝到了模板的 `label` 中：

```
function getLabelContent(element) {
  var label = element.find('label');
  return label[0] && label.html();
}
```

但对于验证信息，我们准备在验证失败时使用 `ng-repeat` 来显示仅与当前错误有关的验证信息。所以，需要将验证信息以 `$validationMessages` 属性的形式存储在与模板关联的作用域中。这些验证信息中可能包含插值字符串，所以在编译时我们会对它们做插值处理：

```
function getValidationMessageMap(element) {
  var messageFns = {};
  var validators = element.find('validator');
  angular.forEach(validators, function(validator) {
    validator = angular.element(validator);
    messageFns[validator.attr('key')] =
      $interpolate(validator.text());
  });
  return messageFns;
}
```

针对每个 `<validator>` 元素，我们都使用 `$interpolate` 服务创建一个插值处理函数，该插值处理函数以 `validator` 的文本内容为参数（最终返回以插值替换过的字符串）。可以将 `validator` 的 `key` 属性和对应的插值处理函数作为键值对添加到一个 `map` 对象中。这个 `map` 对象最终会被设置为模板作用域中的 `$validationMessages`。

在 AngularJS 中，`$interpolate` 服务被用来处理包含 `{{}}` 的字符串。如果给这个服务传入一个含有 `{{}}` 的字符串，它就会创建一个可接受作用域的插值处理函数，并最终返回以插值替换的字符串。

```
var getFullName = $interpolate('{{first}}{{last}}');
var scope = { first:'Pete',last:'Bacon Darwin' };
var fullName = getFullName(scope);
```


此处，首先创建了一个名为 `getFullName` 的插值处理函数，它处理的插值字符串是 `{{first}}` `{{last}}`；然后，传入一个 `scope` 对象来调用它，最终 `fullName` 返回的结果是 `'Pete Bacon Darwin'`。

绑定验证信息

在 `field` 指令模板中显示验证的错误信息，可以使用以下 **HTML** 代码：

```
<span class="help-inline" ng-repeat="error in $fieldErrors">
  {{ $validationMessages[error] (this) }}
</span>
```

先对 `$fieldErrors` 中的所有 `error` 键进行迭代，然后通过 `error` 键调用对应的插值处理函数来绑定最终的验证信息。

 我们必须为插值处理函数提供一个作用域对象。在模板中可以使用 `this`，`this` 表示当前作用域。如果不传递作用域，则会导致程序异常，调试起来也很费劲。

`$fieldErrors` 属性中包含了当前 `field` 验证错误所对应的 **key**（一个数组）。该属性通过 `loadTemplate()` 成功回调中创建的监视来保持实时更新。

动态加载模板

Loading templates dynamically

`loadTemplate` 函数会加载指定的模板，然后将模板转换成一个经过 `jqLite` 或 `jQuery` 包裹的 **DOM** 元素，代码如下：

```
function loadTemplate(template) {
  return $http.get(template, {cache:$templateCache})
    .then(function(response) {
      return angular.element(response.data);
    }, function(response) {
      throw new Error('Template not found: ' + template);
    });
}
```

该函数是异步的，所以它返回一个 **promise** 对象来处理模板元素。正如在指令中使用的 `templateUrl` 和 `ng-include` 一样，我们在模板加载成功之后使用 `$templateCache` 来缓存它。

设置 field 指令的模板

Setting up the field template

在 field 指令的链接函数中，我们以指令元素上的 **template** 属性值为参数（如果没有设置 **template** 属性则使用 'input.html'）调用 loadTemplate 函数。

```
loadTemplate(template).then(function(templateElement) {
```

一旦把 **promise** 中的事务搞定，那么 field 指令就可以如期工作了。

```
    var childScope = scope.$new();
    childScope.$validationMessages = angular.copy(validationMsgs);
    childScope.$fieldId = attrs.ngModel.replace('.', '_').toLowerCase()
    + '_' + childScope.$id;
    childScope.$fieldLabel = labelContent;

    childScope.$watch('$field.$dirty && $field.$error',
function(errorList) {
    childScope.$fieldErrors = [];
    angular.forEach(errorList, function(invalid, key) {
        if ( invalid ) {
            childScope.$fieldErrors.push(key);
        }
    });
}, true);
```

首先，我们创建了一个新的子作用域，并为它添加了一些有用的属性，如 \$validationMessages、\$fieldId、\$fieldLabel 和 \$fieldErrors。

```
    var inputElement = findInputElement(templateElement);
    angular.forEach(attrs.$attr, function (original, normalized) {
        var value = element.attr(original);
        inputElement.attr(original, value);
    });
    inputElement.attr('name', childScope.$fieldId);
    inputElement.attr('id', childScope.$fieldId);
```

然后，将指令元素上的所有属性全部拷贝到模板中的 input 元素上，并为其添加经过唯一性计算生成的 name 属性和 id 属性。

```
    var labelElement = templateElement.find('label');
    labelElement.attr('for', childScope.$fieldId);
    labelElement.html(labelContent);
```


其次, 将 `labelContent` 拷贝到模板中的 `label` 元素上, 并为它设置 `for` 属性。

```
element.append(templateElement);  
$compile(templateElement)(childScope);  
childScope.$field = inputElement.controller('ngModel');  
});
```

最后, 将 `templateElement` 插入原始的 `field` 元素中, 并使用 `$compile` 服务来编译整个元素, 将其与新的 `childScope` 连接起来。一旦元素完成连接, 就可以获取 `ngModelController` 并将其放在 `$field` 属性中以供模板使用。

9.5 小结

Summary

在本章中, 我们学习了一些开发指令的高级特性。当实现 `alert` 指令时, 我们学习了当创建组件时是如何通过 `ng-transclude` 来实现嵌入的。`accordion` 指令套件阐明了如何使用指令控制器来实现各个指令之间的沟通协作。我们还在 `field` 指令中学习了通过终止当前编译进程来完全控制指令的编译过程, 以及使用 `$compile` 服务来手动编译元素内容。

第 10 章

创建为全球用户服务的 AngularJS 应用

Building AngularJS Web Applications for an International Audience

如今我们生活在地球村中，任何一个接入因特网的人都可以访问你的 web 应用。你的项目在启动时可能只提供了一种语言，但随着网站在全球互联网中的逐渐流行，你可能会为你的用户提供各种语言版本的内容，以匹配他们的阅读需求，或者你也可能由于客户要求或法律强制要求必须提供多个本地化版本。无论什么原因，总之，国际化问题是许多 web 开发人员要面对的现实。

国际化（internationalization，简称 i18n）和本地化（localization，简称 l10n）包含很多方面的内容。但在本章中，我们只关注 AngularJS 应用中的国际化问题和解决方案。本章主要包括如下内容：

- 根据用户的偏好设置来配置日期、数字和货币的显示格式以及处理其他与地域有关的设置。
- 将网站内容翻译成多种语言（不仅包括 AngularJS 模板中的内容，还包括 JavaScript 中处理的内容）。

本章的最后部分将会展示一些对创建国际化 AngularJS 应用非常有用的模式、秘诀和技巧。

10.1 使用本地化的符号和设置

Using locale-specific symbols and settings

AngularJS 框架中包含一套用于本地化设置的模块。本节将展示配置本地化设置模块所需的步骤，并介绍各项设置的意义和各地区可用的常量。

配置本地化设置模块

Configuring locale-specific modules

如果你仔细查看 AngularJS 框架的代码，就会发现在每个发行版本中都包含一个名为 `i18n` 的文件夹。在这个文件夹中是一组 `js` 组件，文件命名按照这种模式：`angular-locale_[locale name].js`，其中 `[locale name]` 对应一个具体的地区名称（一个由语言代码和国家代码组合的名称）。例如，一个针对加拿大地区法语的本地化配置文件，则它的文件名是 `angular-locale_fr-ca.js`。



AngularJS 发布有超过 280 个针对不同语言和国家的本地化配置文件。但这些文件并不作为 AngularJS 项目的一部分。这些本地化设置其实是定期从 closure 库 (<http://closure-library.googlecode.com/>) 中提取而来的。^[1]

AngularJS 默认会将 `i18n` 设置为美国英语环境 (`en-us`)。如果你在开发一个针对其他地区的应用，那就需要引入一个对应的本地化文件并声明对本地化模块的依赖。例如，要配置一个网站使用加拿大法语，则需要按照如下所示的格式组织 `script` 文件：

```
<!doctype html>
<html ng-app="locale">
<head>
<meta charset="utf-8">
<script src="lib/angular/angular.js"></script>
<script src="lib/angular/angular-locale_fr-ca.js"></script>
<script src="locale.js"></script>
</head>
<body ng-controller="LocaleCtrl">
...
</body>
```

[1] 译者注：angular-1.2.16版本中共包含245个本地化设置文件。

其中的 `locale.js` 文件应该包含一个模块定义, 该模块依赖于 `ngLocale` 模块:

```
angular.module('locale', ['ngLocale'])
```

使用已有的本地化设置

Making use of available locale settings

AngularJS 发布的每个本地化文件都包含一个名为 `ngLocale` 的模块。`ngLocale` 模块只暴露一个服务: `$locale`。`$locale` 服务所提供的公开的有文档说明的 API 只包括一个 `$locale.id`, 用于获取当前所使用的地区标识。

实际上, `$locale` 服务暴露了很多有关地区标识的信息。顺着它所暴露的常量寻找, 我们能发现一个存储日期时间格式的常量 (`$locale.DATETIME_FORMATS`), 以及能找到数字格式 (`$locale.NUMBER_FORMATS`)。这两个常量都是 JavaScript 对象, 其中又包含用于格式化日期、时间、数字和货币的各种设置。例如, 可以通过 `$locale.DATETIME_FORMATS.MONTH` 找到所有月份的名称。

本地化设置和AngularJS过滤器

能获取所有本地化设置, 可以给我们编写自定义指令和给过滤器带来极大便利。AngularJS 已经在内置的过滤器中很好地利用了这些设置。

1. 日期过滤器

日期 (`date`) 过滤器会根据设定的格式转换日期的显示方式。日期格式既可以是一种精确定义较为通用的格式 (如 `mm/dd/yy hh:mm a`), 也可以是一种完全本地化的格式。除了精确定义日期中每一个字符这种方式, 还可以使用预先定义好的命名日期格式。AngularJS 能识别如下这些预定义的命名日期格式: `medium`、`short`、`fullDate`、`longDate`、`mediumDate`、`shortDate`、`mediumTime`、`shortTime`。

举一个例子, 假设使用 `{{now | date:'fullDate'}}` 这个表达式 (其中的 `now` 会被解析为 `new Date()`), 那么根据不同的地区会输出不同的结果:

- `Tuesday, April 9, 2013 in en-us` (英语美国);
- `mardi 9 avril 2013 in fr-fr` (法语法国)。

2. 货币 (currency) 过滤器

货币过滤器可以将数字格式化为各种货币形式。默认情况下, AngularJS 会使用当前地区的货币符号。例如, `{{100 | currency}}` 表达式会根据配置的不同地区显示出不同的结果:

- **\$100.00 in en-us** (英语美国);
- **100.00 € in fr-fr** (法语法国)。

通常, 我们是将带有货币符号的一串数字看做价格, 所以货币过滤器的默认行为可能会让用户感到迷惑。如果在用户改变本地化设置之后简单地将 **100.00 €** 直接改为 **100.00 \$**, 那就更加匪夷所思了。^[2]

除非你只用一种货币符号, 否则, 建议在使用货币过滤器时永远记得指定一个货币符号:

```
{{100 | currency:'€'}}
```

还存在一个更复杂的问题, 虽然货币过滤器允许我们设定货币符号, 但却无法让我们设定货币符号或小数分隔符的位置 (依然会使用当前地区的默认设置)。这样, 前面列举的例子就会被解析成 **€ 100.00**, 这种格式在欧元中并不常用。

如果货币过滤器的行为正好符合你的需求, 那就尽其所用。但如果有时它无法满足需求, 那就应该准备开发自定义的过滤器。

3. 数字过滤器

数字过滤器的作用符合我们的期望, 它会按照本地化设置中的千进制和小数分隔符来格式化数字。例如,

```
{{1000.5 | number}}
```

会被解析为:

- **1,000.5 in en-us** (英语美国);
- **1000,5 € in fr-fr** (法语法国)。

[2] 译者注: 假设英语网页上有一个价格为\$100的商品, 将本地化地区配置为中国后 (无论是在应用中配置或用户修改本地化设置), 货币过滤器默认会直接将\$100变为¥100, 这明显是不合理的。

10.2 处理翻译

Handling translations

按照本地化设置格式化日期和数字只是本地化课题中的很小一部分。通常人们所说的本地化，第一时间想到的肯定是翻译。

在 AngularJS 中至少有两个地方可以找到需要翻译的内容：模板和 JavaScript 代码中的字符串。

在本章接下来的内容中，假设已经有了一份针对各地区翻译好的字符串列表，且它已经按照某种格式（如 JSON）组织好了备用。该 JSON 对象的键是字符串片段的逻辑名称（如 `crud.user.remove.success`），而值是某个指定地区所对应的实际翻译后的字符串。例如，针对 `en-us` 地区，该 JSON 会是这个样子：

```
{
  'crud.user.remove.success': 'A user was removed successfully.',
  'crud.user.remove.error': 'There was a problem removing a
  user.'
  . . .
}
```

同样的 JSON 结构，针对 `pl-pl`（波兰语 - 波兰）地区，则其内容如下：

```
{
  'crud.user.remove.success': 'Użytkownik został usunięty.',
  'crud.user.remove.error': 'Wystąpił błąd podczas usuwania
  użytkownika.'
  . . .
}
```

翻译 AngularJS 模板中的字符串

Handling translated strings used in AngularJS templates

通常，需要翻译的绝大多数字符串都在 AngularJS 模板中。下面以简单的 "Hello, World!" 为例：

```
<span>Hello, {{name}}!</span>
```

要让这个模板翻译成不同的语言，则需要找到一种方法来将 `Hello` 替换成目标语言对应的字符串。此处，可以使用几种不同的技术来实现这点，每种方法都各有长短，具体讨论请见下节。

使用过滤器

假设翻译字符串所在的 JSON 结构如下：

```
{
  'greetings.hello': 'Hello'
  . . .
}
```

可以设想开发一个翻译过滤器（叫它 `i18n` 吧），使用方法如下：

```
<span>{{'greetings.hello' | i18n}}, {{name}}!</span>
```

开发一些简易版本的 `i18n` 过滤器不是难事，关键实现代码如下：

```
angular.module('i18nfilter', ['i18nmessages'])
  .filter('i18n', function (i18nmessages) {
    return function (input) {
      if (!angular.isString(input)) {
        return input;
      }
      return i18nmessages[input] || '?' + input + '?';
    };
  });
```

`i18n` 过滤器依赖于一组翻译语料库（`i18nmessages`）。可以在一个单独的模块中声明这个语料库，示例如下：

```
angular.module('i18nmessages', [])
  .value('i18nmessages', {
    'greetings.hello': 'Hello'
  });
```

此处展示的 `i18n` 过滤器非常简单，还有很大扩展和完善的空间：通过带有缓存的 `$http` 服务加载翻译语料库，实时切换地区设置等。目前这个简单的 `i18n` 过滤器基本可用，我们可以等到它遇到性能问题时，再对它进行完善。

在这种方法中，我们将静态的 **"Hello"** 变成一个过滤器可识别的表达式（`{{'greetings.hello' | i18n}}`），这样就在 **AngularJS** 应用中多加入了一个表达式。第 11 章将介绍在 **AngularJS** 框架中有大量需要监视和解析的表达式。为每一个单独的字符串都添加一个新的监视表达式是一种极大的浪费，可能会将应用拖慢到无法接受的地步。



虽然过滤器的翻译方法简单灵活，但对性能有不利影响。如果只是一个简单页面上有少数几个字符串需要翻译，那么这种性能影响尚可接受。对含有大量翻译字符串的复杂页面，性能问题就变成一个极大的瓶颈。

使用指令

为了弥补基于过滤器的翻译方法在性能上的不足，可以将目光转向指令。使用指令翻译的语法示例如下：

```
<span><i18n key='greetings.hello'></i18n>, {{name}}!</span>
```

使用指令方法，不需要给 AngularJS 添加新的监视表达式，这样就消除了基于过滤器翻译所存在的性能问题。但是，使用指令翻译有它自己的问题。

首先，语法冗余不够优雅。我们可以尝试用其他指令方式来替换（如使用属性指令），但不管使用何种指令方式，模板都会因此变得难以阅读和修改。不过还有一个更严重的问题：指令翻译方法在某些特定场景下无法使用。我们来看一个带有 placeholder 属性的 input 元素：

```
<input ng-model='name' placeholder='Provide name here'>
```

我们无法使用指令翻译方法来翻译 "Provide name here"，因为 AngularJS 中无法解析 HTML 属性值中的指令。

如上所述，基于指令的翻译方法无法覆盖所有翻译需求，所以需要我们探索更好的解决方案。

1. 在构建时翻译模板

本章中要讲解的最后一种翻译方法是，将翻译工作迁移到构建系统。该方法的基本思路是在构建时处理所有模板，生成一系列对应各种语言的模板，以便浏览器下载使用。在这种方法中，提供给 AngularJS 的模板是固定的，不需要在客户端做任何语言翻译的处理工作。

在构建时，翻译模板所需要的具体技术依赖于应用所使用的构建系统。在基于 Grunt.js 的构建系统中，可以使用 Grunt 的功能来生成固定模板。假设有如下名为 hello.tpl.html 的模板：

```
<div>
<h3>Hello, {{name}}!</h3>
<input ng-model='name' placeholder='Provide name here'>
</div>
```

可以将其转换成 Grunt.js 模板，以便在构建时使用：

```
<div>
<h3><%= greeting.hello %>, {{name}}!</h3>
<input ng-model='name' placeholder='<%= input.name %>'>
</div>
```

然后，基于应用所支持的地区和语言，Grunt.js 的构建系统就会生成各种翻译后的模板，并将其保存在按照地区名称命名的文件夹中。示例如下：

```
/en-us/hello.tpl.html
/fr-ca/hello.tpl.html
/pl-pl/hello.tpl.html
```

配置一套完整的构建时翻译系统是比较烦琐复杂的，但通常只要在项目开始时花费一些精力，配置好之后就受益无穷了。具体的益处就是避免本地化设置可能带来的各种性能问题，而且可以翻译模板中的任意字符串。

翻译 JavaScript 代码中的字符串

Handling translated strings used in the JavaScript code

除了翻译 AngularJS 模板中的字符串之外，有时还需要翻译 JavaScript 中的文字。我们可能需要在 JS 中显示本地化的错误信息，或者提醒信息等。不论何种原因，都要考虑如何翻译 JavaScript 中的字符串。

AngularJS 本身并没有为这种需求提供任何工具或帮助，所以需要卷起袖子，自己埋头写一个简单的服务，这个服务就是我们得心应手的翻译工具。在编写代码之前，先构思一个具体的使用场景。假设的场景是，在数据存储中成功删除一条记录，或者在数据存储中没有找到我们需要的数据等的情况下，显示一条本地化的提示信息。

如果想要支持多种语言，那么在 JavaScript 代码中就不能将消息文字写成硬编码，而是要通过一种方法以 **key** 为索引获取本地化和参数化的信息，具体示例如下：

```
localizedMessages.get('crud.user.remove.success', {id: 1234})
```

在 en-us 环境中，上面的代码返回的信息会是 "A user with id '1234' was removed successfully"，在 zh-cn 环境中则会是“ID 为 ‘1234’ 的用户已被成功删除”。

只写出一个通过 **key** 和当前地区来查找信息的服务是没有实际价值的。现在面临的唯一难点是，如何处理本地化字符串中的参数。幸运的是，我们可以学习 **AngularJS**，再次请出 `$interpolate` 这个法宝，也就是 **AngularJS** 用来处理模板中插值指令的服务。这样，本地化信息就可以按如下格式定义：

```
"A user with id '{{id}}' was removed successfully."
```

在 **SCRUM** 示例应用中，就有基于上面所描述的想法实现的一个完整的本地化服务。该服务的核心实现代码如下：

```
angular.module('localizedMessages', [])
.factory('localizedMessages', function ($interpolate, i18nmessages) {

    var handleNotFound = function (msg, msgKey) {
        return msg || '?' + msgKey + '?';
    };

    return {
        get : function (msgKey, interpolateParams) {
            var msg = i18nmessages[msgKey];
            if (msg) {
                return $interpolate(msg)(interpolateParams);
            } else {
                return handleNotFound(msg, msgKey);
            }
        }
    };
});
```

10.3 范式、秘诀和技巧

Patterns, tips, and tricks

本章的最后部分主要讲解与国际化和本地化有关的一些开发范式。首先会讲解对 web 应用进行本地化处理及切换地区的方法。然后学习一些处理运行状态中 web 应用本地化问题的范式：覆写默认的本地化格式设置，以及按照用户选择的地区处理用户输入值。

按照设定的地区初始化应用

Initializing applications for a given locale

在本章开头介绍过，AngularJS 的每个发行版本都包含有一组本地化设置文件。每个地区对应这样一个文件，每个文件中都有 ngLocale 模块。如果想使用本地化设置，那么需要在应用中声明对 ngLocale 模块的依赖。下面的代码展示了如何针对一个特定地区初始化 AngularJS 应用：

```
<head>
<meta charset="utf-8">
<script src="lib/angular/angular.js"></script>
<script src="lib/angular/angular-locale_fr-ca.js"></script>
<script src="locale.js"></script>
</head>
```

如果应用只是为某一个特定地区提供服务，那么这种方法就很好。但实际上，我们通常是想根据用户的偏好设置来初始化本地设置。用户的偏好设置有多种来源：

- 用户浏览器的设置；
- HTTP请求的头部信息（例如，Accept-Language）；
- URL地址或请求参数；
- 服务器端设置（用户资料、地理位置等）。

从上面介绍的内容中可以清晰地看出，从服务器端获取地区信息更有价值（信息更完备）。这就是为什么应该提倡在服务器端处理应用的初始页面。

为了演示实际中的本地化处理过程，我们来看看 SCRUM 示例应用中所采取的方案。该方案定位最终目标地区基于以下几个因素：

- 地区设置作为URL地址中的一部分，会触发应用的引导程序；
- 请求头部的Accept-Language信息；
- 一组本地化设置文件。

该方案中用户可以使用如下两种不同的 URL 来访问应用：

- `http://host.com/fr-ca/admin/users/list`:该地址表示用户设定了一个期望的地区 (`fr-ca`)。我们应该在本地化设置文件中查找请求所需要的文件, 如果恰好不支持用户所请求的地区, 那应该将用户重定向到另外一个URL上去。例如, 在不支持`fr-ca`的情况下, 可以将用户重定向到默认地区 (`en-us`): `http://host.com/en-us/admin/users/list`。
- `http://host.com/admin/users/list`:该地址中没有指定具体地区。我们可以根据HTTP请求头部的`Accept-Language`信息猜测出用户大致的地区, 然后将其重定向到对应的URL地址。很明显, 此处也需要对请求地区和已支持的地区列表做交叉检查 (和第一条中一样)。



在实际应用中, 你可能会有更加复杂的算法, 根据各种不同来源的信息合并计算出最精确的地区, 并提供可靠的默认方案。这种根据请求信息定位用户所在地区的方案, 其精确程度取决于应用的需求。

一旦用户所在的地区确定, 就可以给浏览器返回一个页面用于引导整个应用。初始页面 (`index.html` 或类似页面) 需要用服务器端动态生成。**SCRUM** 示例应用中的初始页面 `index.html` 如下:

```
<head>
<meta charset="utf-8">
<script src="lib/angular/angular.js"></script>
<script src="lib/angular/angular-locale_<%= locale %>.js"></script>
. . .
</head>
```

此处的目标地区 `locale` 是通过定位算法得出的一个地区标识, 这个初始页面模板在发送给浏览器之前会先经过服务器端处理 (解析 `locale` 变量)。

将地区标识作为URL一部分带来的问题

一旦给应用的 URL 地址添加一个新的部分, 就需要注意由此带来的两个新问题: 重新配置路由和下载模板。

通常, URL 中新加入的路径元素都应该被当做路由的一部分。因此我们完全可以去应用中重新修改所有路由配置, 例如将 `/admin/users/list` 改为 `/:locale/admin/users/list`。

这种方法能保证正常工作，但这种改法非常乏味，而且在所有路由前面加 `/:locale` 容易导致错误。所以我们换一种方法，在初始页面的 HTML 中定义一个 `base` 标签，将其指向特定地区所在的文件夹：

```
<head>
<meta charset="utf-8">
<script src="/lib/angular/angular.js"></script>
<script src="/lib/angular/angular-locale_<%= locale %>.js"></script>
<base href="/<%= locale %>/">
. . .
</head>
```

AngularJS 的 `$location` 服务（也就是路由系统）可以识别 `<base>` 标签，它会按照 `base` 的 `href` 属性中的地址来处理所有路由。

使用指向地区文件夹的 `base` 标签的另一个好处就是，可以使用相对路径来下载 AngularJS 模板。如果采用构建时翻译系统处理本地化，那么就会在各地区文件夹中生成很多翻译后的模板。此时正确配置的 `<base>` 标签可以保证 AngularJS 下载到用户所在地区的模板。



在页面上使用 `<base>` 标签，意味着针对所有与地区无关的静态内容都需要使用绝对地址。

在现实的发布场景中，模板极有可能是一开始就被预先取回到本地的，而不是需要时才去下载。因此，从模板加载方面来看定义 `<base>` 标签并没有实际意义。对发布场景的详细介绍请见第 12 章。

切换地区 Switching locales

在目前的 AngularJS 中，应用程序所依赖的所有模块必须在应用被引导之前在依赖关系中被一一声明。ngLocale 模块也不例外，这表示它需要被加载到浏览器中，然后在 AngularJS 应用启动之前声明对它的依赖关系。因此，我们需要在应用启动之前选择出目标地址对应的 ngLocale 模块。如果不重新初始化 AngularJS 应用，那么这个选定的 ngLocale 模块也无法被替换。



在当前版本的AngularJS中，地区模块需要在应用初始化之前预先选定。而设置好之后的地区也无法在运行时实时切换，要先实现切换，就需要选定另一个ngLocale模块之后重新初始化应用。

明白了当前 AngularJS 模块系统的工作方式之后，切换地区的最好做法就是，通过将用户重定向到一个新的目标地区的 URL，从而触发应用重新初始化。重定向意味着应用的当前状态信息都会丢失，幸运的是，AngularJS 的深度链接（deep link）特性可以在切换地区之后将用户带到他之前浏览的位置。

为了通过重定向切换地区，我们需要先准备好一个新的 URL 地址。使用 \$location 服务的 API 可以很简单地得到新 URL。例如，根据当前的 URL：
http://host.com/en-us/admin/users/list，得到新的目标 URL：http://host.com/fr-ca/admin/users/lis。

下面编写一个 switchLocaleUrl 函数来处理此事：

```
$scope.switchLocaleUrl = function(targetLocale) {
    return '/' + targetLocale + '/' + $location.url();
};
```

上面的 switchLocale Url 函数使用当前的应用地址和指定的 targetLocale 计算出一个新地址。这个函数可以用在标准的 <a> 标签上，以实现用户切换地区的效果：

```
<a ng-href="switchLocaleUrl('fr-ca')" target="_self">Français</a>
```

针对日期、数字和货币的自定义过滤器

Custom formatting for dates, numbers, and currencies

AngularJS 的一些过滤器默认就可以处理与本地化相关的内容。如在本章开头介绍的日期（date）过滤器，可以根据设定的命名格式以本地化方式处理日期（{{now | date:'fullDate'}}）。fullDate 所对应的实际格式，是在本地化设置文件中定义的，具体是以 \$locale.DATETIME_FORMATS.fullDate 为键名。例如，针对 fr-ca 地区 fullDate 所对应的具体格式为：EEEE d MMMM y。

这些默认的格式通常情况下就是针对某种语言和国家所期望的样子，但有时我们可能想针对某些特定地区对格式稍做修改。给过滤器创建一个修饰器（decorator）可以实现这点。



在 AngularJS 中可以给任何已有的服务或过滤器创建修饰器。修饰器将某个服务包装起来，然后在上面修饰（扩展）一些新功能。AngularJS 修饰器是修饰模式的绝佳例子。有关修饰模式请见 <http://zh.wikipedia.org/wiki/>。

例如，要针对 fr-ca 地区修改 fullDate 的格式，就可以为日期过滤器包装一个修饰器：

```
angular.module('filterCustomization', [])
  .config(function ($provide) {
    var customFormats = {
      'fr-ca': {
        'fullDate': 'y'
      }
    };

    $provide.decorator('dateFilter', function ($delegate, $locale) {
      return function (input, format) {
        return $delegate(input, customFormats[$locale.id][format]
          || format);
      };
    });
  })
```

在修饰器中我们发挥了 AngularJS 依赖注入系统的优势。通过定义一个新的修饰器（`$provide.decorator()`），就可以将某个已有的服务包入自定义服务中，同时仍可以访问原始的服务（`$delegate`）。上面的代码中除了新出现的 `$provide` 服务，其他代码应该很好理解。在一个哈希对象（`customFormats`）中检索目标地区对应的覆写日期格式，如果覆写格式存在，就用它来调用日期过滤器；否则，就使用日期过滤器提供的原始格式。

修饰器直接替换了日期过滤器，因为我们在注册过滤器时使用了与原始日期过滤器一样的名字。这种方法的好处是不用改动应用程序的代码就能直接使用自定义的过滤器。

我们还可以进一步完善这个修饰器，应该为没有指定目标地区的情况也提供一个默认的日期格式。



还有一种简单粗暴的办法，那就是直接修改对应的本地化设置文件（就是AngularJS发行版本中包含的本地化文件）。这种办法的缺点就是在对AngularJS进行升级之后需要记得去修改这个文件。

10.4 小结

Summary

本章主要讲解了 AngularJS 应用中的国际化和本地化问题，了解了 AngularJS 提供的 `ngLocale` 模块，该模块中为日期、货币和数字做了一些本地化设置。这些设置是从 `closure` 库中派生而来的，并对 AngularJS 的一些内置过滤器的行为有影响。

本地化最主要的工作是翻译模板和 JavaScript 代码中的文本信息。我们探究了各种翻译 AngularJS 模板的方法：基于过滤器、基于指令以及构建时翻译。基于过滤器的翻译方法看起来不错，但对大多数项目来说它可能会带来无法承受的性能影响。基于指令的翻译方法解决了性能问题，但用在实际项目中不够灵活。在一番探究之后，我们推荐使用构建时翻译系统来翻译 AngularJS 模板，模板会在构建时翻译好以备浏览器直接使用。

本章我们还总结了一些和 `i18n/l10n` 有关的开发范式，研究了应用初始化和切换地区的问题。发现当前版本的 AngularJS 需要在应用启动之前就设定好地区，所以，我们应该在服务器端处理地区信息，然后动态地生成应用的引导页面。在目前版本的 AngularJS 中，运行时实时切换地区是不可行的，所以最好的办法就是将用户重定向到引导页面以实现地区切换。

本章还介绍了其他开发范式，用以扩展自定义的日期、数字和货币格式来显示信息。虽然 AngularJS 提供的默认本地化设置已经相当合理，但有时我们可能需要对这些设置稍做修改。使用修饰器来改装已有的过滤器，是自定义数据输出的好办法。

目前，我们已经学习了如何开发一个功能完备且国际化的 AngularJS web 应用。第 11 章将关注如何保证应用的健壮可靠。说的更具体一些就是，我们将会寻找和定位每一个可能潜在的与性能有关的问题。

第 11 章

开发健壮的 AngularJS 应用

Writing Robust AngularJS Web Applications

web 应用的性能是一种非功能性需求，需要我们在功能任务和紧急状态之间取得平衡。很明显，我们无法忽略性能问题，即使应用看上去完美无缺且具备了所有需要的功能，但如果运行不够稳定，人们还是会拒绝使用。

影响整个 web 应用性能表现的因素有很多方面：网络情况、DOM 树的大小、CSS 规则的数量和复杂度、JavaScript 的逻辑和算法、数据结构，等等。不要推脱说性能问题完全取决于用户所使用的浏览器和用户的主观感受！有些性能问题是与具体技术无关且普遍存在的。而有些性能问题则是 AngularJS 特定的问题，本章将专注于 AngularJS 特有的问题。

为了理解 AngularJS 的性能特征，我们需要更好地理解它的内部组织结构。所以本章一开始就对 AngularJS 的核心架构做了深入的探究，紧接着介绍了一些影响性能的开发模式，探讨了各种可用的方案及其优缺点。

阅读本章之后你将有如下收获。

- 了解隐藏在引擎盖之下 AngularJS 渲染引擎是如何工作的。熟悉 AngularJS 的内部结构以便更好地理解它的性能特性。
- 理解 AngularJS 应用在理论上的各种性能限制，能够快速判断出你的工程是否触到了性能边界。

- 能够分析和定位出 AngularJS 应用中的 CPU 杀手和内存消耗大户。在你动手写代码之前应该先熟识各种性能敏感的开发模式，同时学会使用性能监控工具来发现已有代码中的性能问题。
- 理解在大批量数据下使用 `ng-repeat` 指令对性能的影响。

11.1 理解 AngularJS 的内部运作机制

Understanding the inner workings of AngularJS

想要理解 AngularJS 应用的性能特征，需要掀开框架的盖子看看里面。熟悉 AngularJS 内部结构的运作机制，就能通过分析应用场景和具体代码，找出影响应用性能的最主要问题。

AngularJS 不是基于字符串的模板引擎

It is not a string-based template engine

查看 AngularJS 的简单示例代码，就会有一个基本印象：AngularJS 又是一个客户端模板引擎。没错，来看看下面的代码：

```
Hello, {{name}}!
```

AngularJS 的代码与其他一些常规的模板系统（比如 Mustache，<http://mustache.github.io/>）看上去并无二致。只有在添加了 `ng-model` 指令之后，它们之间的区别才变得更明显：

```
<input ng-model="name">  
Hello, {{name}}!
```

只需要如上所示的代码，DOM 就会根据用户的输入实时更新，无需开发者做任何干预。刚开始见识双向数据绑定的效果会觉得很神奇。毋庸置疑，AngularJS 使用了非常可靠的算法为 DOM 树赋予了生命！在本章的后续部分我们会剖析这些算法，仔细研究 DOM 的变化如何传播给模型，模型的变化又如何触发 DOM 重绘。

响应DOM事件更新模型

AngularJS 通过不同指令注册的 DOM 事件监视器来将 DOM 树的变化传播给模型。事件监视器中的代码通过修改 `$scope` 暴露的变量来更新模型。

我们可以模拟编写一个与 `ng-model` 指令效果相同的简单指令（我们称它为 `simple-model`），用来阐明更新模型的技术要点，代码如下：

```
angular.module('internals', [])
  .directive('simpleModel', function ($parse) {
    return function (scope, element, attrs) {

      var modelGetter = $parse(attrs.simpleModel);
      var modelSetter = modelGetter.assign;

      element.bind('input', function(){
        var value = element.val();
        modelSetter(scope, value);
      });
    };
  });
```

`simple-model` 指令的核心思路是元素的 `input` 事件函数兼容着输入框的变化，随时根据用户输入的值更新模型。

为了设置真实的模型值，我们使用了 `$parse` 服务。该服务既用来根据作用域计算 AngularJS 表达式的值，又可以在某个作用域上设置值。当传入一个表达式作为参数来调用 `$parse` 服务时，它会返回一个 `getter` 函数。如果传入的表达式可被赋值，那么这个 `getter` 函数就会带有一个 `assign` 属性（属性值是对应的 `setter` 函数）。

将模型变化传播给DOM

也可以使用 `$parse` 服务编写一个简单的 `ng-bind` 指令，该指令可以将模型值渲染为 DOM 节点中的文字，代码如下：

```
.directive('simpleBind', function ($parse) {
  return function (scope, element, attrs) {

    var modelGetter = $parse(attrs.simpleBind);
    element.text(modelGetter(scope));
  }
});
```

此处的 `simple-bind` 指令获取了一个表达式（由 DOM 元素上的 `simpleBind` 属性提供），然后套上一个作用域进行计算，最后将求得的值更新为 DOM 元素中的文字。

同步DOM和模型变化

上面的两个指令编写好之后，就可以试着在 HTML 代码中使用它们。我们期望这两个新的指令能像 `ng-model` 和 `ng-bind` 那样工作，代码如下：

```
<div ng-init='name = "World"'>
  <input simple-model='name'>
  <span simple-bind='name'></span>
</div>
```

不幸的是，运行上面的代码得不到我们期望的效果！初始渲染的效果是正确的，但 `<input>` 的变化并不会触发 `` 的同步更新。

肯定少写了什么，快速检查 `simple-bind` 指令就会发现这个指令只做了初始模型的渲染，并没有监视模型的变化，也就不会对模型变化做出任何反应。可以在作用域实例上使用 `$watch` 方法来修复这个问题：

```
.directive('simpleBind', function ($parse) {
  return function (scope, element, attrs) {

    var modelGetter = $parse(attrs.simpleBind);
    scope.$watch(modelGetter, function(newVal, oldVal){
      element.text(modelGetter(scope));
    });
  }
});
```

`$watch` 方法可以让我们监视模型的变化，每当有变化时就执行一个函数做出响应。`$watch` 方法的特征很明显，代码如下：

```
scope.$watch(watchExpression, modelChangeCallback)
```

`watchExpression` 既可以是一个函数也可以是一个 AngularJS 表达式（用于指定要监控的模型值）。`modelChangeCallback` 是一个回调函数，会在每次 `watchExpression` 的值发生变化时被调用。`modelChangeCallback` 本身可以接受两个参数，分别是 `watchExpression` 变化前后的值。

在熟悉了 `$watch` 的作用之后，就会发现 `simple-model` 指令也用得上它，可以监控 `input` 的输入值，在发生变化时更新模型中对应的值：

```
.directive('simpleModel', function ($parse) {
  return function (scope, element, attrs) {

    var modelGetter = $parse(attrs.simpleModel);
    var modelSetter = modelGetter.assign;

    //从Model更新DOM
    scope.$watch(modelGetter, function(newVal, oldVal){
      element.val(newVal);
    });

    //从DOM更新Model
    element.bind('input', function () {
      modelSetter(scope, element.val());
    });
  });
});
```

在上面的这些修改中，我们为指令设置了监视来监控模型的变化。上述代码看起来很完善，理论上它的表现应该很完美。不幸的是，这些代码在实际使用时还是无法正常工作。

事实证明我们忽略了一个很基本的问题：**AngularJS** 该在何时以何种形式监视模型的变化。我们要理解 **AngularJS** 在哪种情况下才会开始解析监视表达式并检查模型变化。

Scope.\$apply——打开AngularJS世界的钥匙

当 **AngularJS** 首次向公众发布之后，就有许多关于它的模型变化监控算法的“阴谋论”。其中最被津津乐道的一种是，怀疑 **AngularJS** 使用了某种轮询机制。这种机制可能每隔一小段时间就去检查模型值的变化，如果发现变化，就重绘 DOM。所有这些猜测都是错误的。



AngularJS没有使用任何形式的轮询算法来定期检查模型变化。



AngularJS 的模型变化监控机制背后的思路其实是“善后”（observe at the end of the day），因为引发模型变化的情况是有限（数得出来）的。这些情况包括：

- DOM事件（例如，用户修改了input 的值，然后点击一个按钮，调用一个 JavaScript函数或执行其他操作）；
- XHR响应触发回调；
- 浏览器的地址变化；
- 计时器（setTimeout、setInterval）触发回调。

的确，如果上面的任何一种情况都未发生（用户与页面没有发生任何交互，没有任何 XHR 响应，没有计时器结束），那么监控模型的变化是没有意义的。此时页面上什么事情也没发生，模型也没有变化，重绘 DOM 也就毫无必要。

AngularJS 只会在被明确告知的情况下才会启动它的模型监控机制。为了让这种复杂的机制运作起来，需要在 scope 对象上执行 \$apply 方法。

回到之前的 simple-model 指令，可以在 input 值每次发生变化后执行 \$apply 方法（这样每次按键的变化都会传播给模型。这也是 ng-model 指令的默认行为）。

```
.directive('simpleModel', function ($parse) {
  return function (scope, element, attrs) {

    var modelGetter = $parse(attrs.simpleModel);
    var modelSetter = modelGetter.assign;

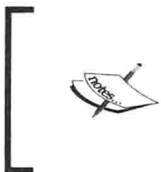
    //从Model更新DOM
    scope.$watch(modelGetter, function(newVal, oldVal){
      element.val(newVal);
    });

    //从DOM更新Model
    element.bind('input', function () {
      scope.$apply(function () {
        modelSetter(scope, element.val());
      });
    });
  };
});
```


或者，也可以修改默认的策略，只在用户光标离开输入框的时候才传播模型变化：

```
//从DOM更新Model
element.bind('blur', function () {
    scope.$apply(function () {
        modelSetter(scope, element.val());
    });
});
```

不论选择哪种策略，最重要的一点是模型变化监控机制需要被明确告知在何时启动。使用 AngularJS 的内置指令时，我们完全无须调用 `$apply` 方法，就可以看到很多“奇迹”，这一点让人很惊讶。但实际上，内置指令的实现代码中调用了 `$apply` 方法。也就是说，标准的指令和服务已经帮我们处理好了模型变化的监控工作（`$http`、`$timeout`、`$location` 等）。



当在一个作用域上调用 `$apply` 方法后，AngularJS 就会启动模型变化监控机制。在网络通信、DOM 事件、JavaScript 计时器或浏览器地址发生变化之后，AngularJS 标准的服务和指令内就会调用 `$apply` 方法。

深入 \$digest 循环

在 AngularJS 的术语中，把检测模型变化的过程称为 `$digest` 循环。这个名字来源于 `Scope` 实例上的 `$digest` 方法。这种方法被作为 `$apply` 中的重要一步来调用，它会检测注册在所有作用域上的所有监视对象。

为什么 AngularJS 中要有 `$digest` 循环呢？它又是如何判断模型发生变化的？`$digest` 循环的存在是为了解决下面的两个问题。

- 判定模型的哪些部分发生了变化，以及 DOM 中的哪些属性应该被更新。这一步的目的是让检测模型变化的过程对开发者保持尽可能的简单。在开发时，我们只需要修改属性值，之后 AngularJS 指令会自动找出网页中哪些部分应该被重绘。
- 减少不必要的重绘以提升应用性能，避免 UI 界面闪烁。为了实现这点，AngularJS 会将 DOM 重绘尽可能推迟到最后一刻，此时模型趋于稳定（所有模型值都已完成运算，时刻准备驱动 UI 重绘）。

要理解 AngularJS 如何实现这种效果，首先要明白 web 浏览器只有一个 UI 线程。浏览器中当然还有其他线程（如负责网络相关操作的线程），但只有一个线程用于渲染 DOM 元素、监视 DOM 事件，以及执行 JavaScript 代码。浏览器不停地在 JavaScript 执行环境和 DOM 渲染环境之间切换。

AngularJS 确保在将控制权交还给 DOM 渲染环境之前，所有的模型值都已完成运算且已“稳定”。这种方法保证了 UI 一次性完成重绘，而不会为了响应某个单独的模型变化而不停地重绘。这保证了更快的执行速度（因为运行环境的切换很少）和更好的视觉效果（所有重绘一次完成）。每个单独的模型属性变化都触发一次 UI 重绘会让界面变得很慢，而且会出现明显的闪烁。

1. 解剖\$watch

AngularJS 使用脏检查（dirty checking）机制来判定某个模型值是否真正发生了变化。脏检查的工作机制是将之前保存的模型值和能导致模型变化的事件（DOM 事件、XHR 事件等）发生后计算的新模型值做对比。

再来看一下，注册一个新的模型监视基本语法如下：

```
$scope.$watch(watchExpression, modelChangeCallback)
```

当作用域上添加一个新的 \$watch 时，AngularJS 会运算 watchExpression 表达式，然后在内部将运算所得的值存储起来。紧接着进入 \$digest 循环，watchExpression 会被再次运算，运算所得的新值会和之前保存的值进行对比。modelChangeCallback 只会在新值与旧值不同时才会执行。这个新值也会被保存起来以备下一次对比使用，整个过程可以一直这样持续下去。

作为开发者，我们对自己手动注册的监视（watches）很清楚。但我们还要明白任何指令（AngularJS 的内置核心指令和任何第三方指令）都可以设置自己的监视。任何插值表达式（{{expression}}）也都会在作用域上注册一个新的监视。

2. 模型的稳定性

如果模型上的任何一个监视器都检测不到任何变化，AngularJS 就认为该模型是稳定的（此时就可以进行 UI 渲染工作了）。只要一个监视器的一个变化，就足以使整个 \$digest 循环变“脏”（dirty），迫使 AngularJS 进入新一轮循环。此时使用了“一颗老鼠屎坏了一锅汤”的原则，当然这么做是有充足理由的。

AngularJS 会持续执行 `$digest` 循环，反复运算所有作用域上的所有监视，直到没有发现任何变化为止。连续几轮 `$digest` 循环是必要的，因为模型监视回调会有一些副作用。如果只是简单地设置一个回调，当监视的模型值变化时执行，那就可能改变我们已经运算过且认为是稳定的那些模型。

我们来考虑一个简单的表单例子，该表单有两个字段 `Start` 和 `End`。在这个表单中结束时间（`end`）应该永远晚于开始时间（`start`）。

```
<div>
<form>
  Start date: <input ng-model="startDate">
  End date: <input ng-model="endDate">
</form>
</div>
```

为了保证在模型中 `endDate` 永远晚于 `startDate`，可以注册一个监视，代码如下：

```
function oneDayAhead(dateToIncrement) {
  return dateToIncrement.setDate(dateToIncrement.getDate() + 1);
};

$scope.$watch('startDate', function (newValue) {
  if (newValue <= $scope.startDate) {
    $scope.endDate = oneDayAhead($scope.startDate);
  }
});
```

在上面的监视中，我们让两个模型值之间产生一个依赖关系，其中一个模型值的变化（`startDate`）可以触发另一个的变化。这个例子就能说明模型变化回调的副作用，一个模型变化时可能会让另一个已被认为“稳定”的模型值也发生变化。

对脏检查算法的深入理解，会让你明白一个 `watchExpression` 在每次 `$digest` 循环中都会被至少运算两次。可以通过创建下面的代码来验证这一点：

```
<input ng-model='name'>
{{getName()}}
```

`getName()` 是定义在作用域上的一个函数：

```
$scope.getName = function() {
  console.log('dirty-checking');
  return $scope.name;
}
```

如果运行上面的代码，注意观察控制台的输出，就会发现 `<input>` 的每次变化都会输出两条日志。



任何一个 `$digest` 循环至少都会运行一次，一般情况下会运行两次。这意味着每一个被监视的表达式在每个 `$digest` 循环中都会被运算两次（在浏览器离开 JavaScript 环境转向 UI 渲染之前）。

3. 不稳定的模型

有些情况下，`$digest` 执行两次循环也不足以确定模型的稳定性。更糟糕的是，有可能会永远无法确定模型稳定性的情况！我们来看下面这个例子：

```
<span>Random value: {{random()}}</span>
```

`random()` 函数的定义如下：

```
$scope.random = Math.random;
```

监视表达式等于 `Math.random()`，这样会（极大可能）导致每次 `$digest` 循环运算所得的值都不一样。这表示每次检查的结果都是“脏”的，需要启动新一轮检查。这会导致一遍又一遍地循环，直到 AngularJS 认为这个模型是“不稳定”的，然后强制跳出 `$digest` 循环。



AngularJS 默认最多会执行 10 次循环，之后就会声明该模型是不稳定的，然后中断 `$digest` 循环。

在中断 `$digest` 之后，AngularJS 会抛出一个错误（使用 `$exceptionHandler` 服务来处理，该服务用来在控制台记录错误）。抛出的错误中会包含 5 个最新的不稳定的监视信息（其中含有这些表达式的新值和旧值）。大多数情况下只会有一个不稳定的监视模型，所以很容易找到罪魁祸首。

在 `$digest` 循环被中止之后，JavaScript 线程就会离开“AngularJS 世界”，此时，没有什么能够阻挡浏览器对渲染的向往。之后，用户就会看到使用 `$digest` 循环最后一次运行所得到的模型值渲染的页面。



即使 \$digest 循环运行超过 10 次上限，页面还是会被渲染。在你观察控制台日志之前，这种错误很难定位，所以它可能会被忽略一段时间。即便如此，还是应该追踪并解决与不稳定模型有关的问题。

1. \$digest 循环和作用域的层级

每次 \$digest 循环都会从 \$rootScope 开始，重新计算所有作用域上的所有监视表达式。这一眼看上去有点违反直觉，有些人可能会说只要重新计算指定作用域及其子作用域上的表达式就足矣。不幸的是，这样做会导致 UI 和模型不同步。原因就是子作用域上的变化有可能会影响父作用域上的变量。请看下面的例子，含有两个作用域（一个是 \$rootScope，另一个是 ng-controller 指令创建的）：

```
<body ng-app ng-init='user = {name: "Superhero"}'>
  Name in parent: {{user.name}}
  <div ng-controller="ChildCtrl">
    Name in child: {{user.name}}
    <input ng-model='user.name'>
  </div>
</body>
```

模型（user.name）的变化是被子作用域（ng-controller）触发的，但是这个变化也改变了 \$rootScope 上对应的属性。

这种情况迫使 AngularJS 从 \$rootScope 开始逐级往下（使用深度优先遍历）运算所有的监视表达式。如果 AngularJS 只是运算某个作用域上（加上它的子作用域）的监视表达式，那么，一旦模型有变化，应用中就存在模型和实际显示不同步的风险，比如我们讨论的这个例子，Name in parent: {{user.name}} 中的插值表达式就不会被运算，显示的结果也就不同步。



在每次 \$digest 循环中，AngularJS 都需要运算所有作用域上的所有监视表达式。这种过程从 \$rootScope 开始，然后以深度优先顺序遍历所有子作用域。

整合

总结之前学习的 AngularJS 的内部运作机制，通过讲解一个 input 元素如何将模型变化传播给 DOM 的例子来说明：

```
<input ng-model='name'>
{{name}}
```

上面的代码产生的结果是在 `$scope` 上注册了两个监视器，每一个监视器都在监视模型中的 `name` 变量。在初始加载之后，页面上没有发生任何变化，浏览器守株待兔，时刻等待任何事件的发生。当用户开始在输入框中输入值时，整个机制才会运作起来：

- DOM中的input事件被触发。浏览器进入JavaScript执行环境。
- 由input指令注册的DOM事件处理函数被执行。该处理函数会更新模型值，然后在作用域实例上调用`scope.$apply`方法。
- JavaScript执行进入“AngularJS世界”，`$digest`循环启动。第一次`$digest`循环中发现有一个监视表达式是“脏”的（`{{name}}`插值表达式注册的监视），此时需要再来一次循环。
- 一旦检查出一个模型变化，就会触发一个`$watch`的回调函数。该回调函数会更新插值表达式所在位置的`text`属性。
- 第二次`$digest`循环重新运算了所有监视表达式，但这次没有发现任何变化。AngularJS宣布该模型“稳定”，然后退出`$digest`循环。
- JavaScript执行上下文处理其他任何非AngularJS代码。大多数情况下没有这类代码，之后浏览器退出JavaScript执行环境。
- UI线程切换到DOM渲染环境，浏览器重绘那个`text`属性发生变化的DOM节点。
- 在重绘完成之后，浏览器回到守株待兔的状态。

如上所示，在 DOM 和模型之间传播变化需要一系列严密的步骤。

11.2 性能优化——设置期望值、测量、调节、并重复 Performance tuning – set expectations, measure, tune, and repeat

性能优化需要一套严格的方法，具体如下。

- 要明确定义有关性能表现的期望值，包括可测量的性能指标以及测量所需的环境。
- 对系统目前的性能进行测试，并与期望值进行交叉检查。
- 如果系统的性能表现未达到期望值，那么就要找出性能瓶颈所在，并修复这些问题。在修复完问题之后，需要对整个系统的性能表现再做一次测量，以证明修复工作确实将系统性能提升到了期望水平，或者发现新的瓶颈。

在进入正题之前，我们要专门强调很重要的两点。第一，要明确设置期望值以及测试和衡量性能表现所处的环境。第二，性能优化本身不是目标，它只是一个将系统性能提升到期望水准的一个过程。

AngularJS 和其他任何设计优良的库一样，它的构造基于一系列边界条件，AngularJS 之父 Miško Hevery 对此有很好的描述 (<http://stackoverflow.com/a/9693933/1418796>)：

看起来 AngularJS 好像很慢，因为脏检查是很低效的。但我们需要用真实数据说话，而不是理论上的争论，那就先来定义一些约束条件。

人类的感知定义：

慢的标准——50 毫秒内发生的事情人类是察觉不到的，所以这个时间长度可以被认为是“瞬间”。

数量限制——你不能在一个页面上为人们展示超过 2000 个小块的信息。比这还复杂的页面绝对是糟糕透顶的 UI，人们根本无法处理这种情况。

所以本质的问题就是：AngularJS 能否在 50 毫秒内处理完 2000 个对比计算，即使运行环境是那些老旧浏览器？也就是每个对比计算有 25 微秒的时间。我相信在如今的老旧浏览器中这都不是问题。

警告：对比计算需要尽量简单，以便在 25 微秒内完成。不幸的是，我们能很容易向 angular 中添加一些很慢的对比运算，所以，当你不知道在做什么而胡搞一气的时候，就很容易创建一个很慢的应用。但我们希望通过提供一个性能检测模块让你心中有数，这个模块会显示出那些比较慢的对比计算。

上面所展示的有关 AngularJS 应用在实际和理论上的边界限制条件，以及设置潜藏指令以进行性能优化，对我们有很好的启示。

幸运的是，Miško Hevery 提到的“性能检测模块”已经有了，称为 Batarang，是一个 Chrome 扩展程序。图 11-1 所示的截图展示了 Batarang 扩展运行的效果。另外，Batarang 还可以显示作用域上注册的监视表达式，并显示如下信息：

- 每个监视表达式的执行时间；
- 每个监视表达式在整个 \$digest 循环的执行时间中所占的实际执行时间。

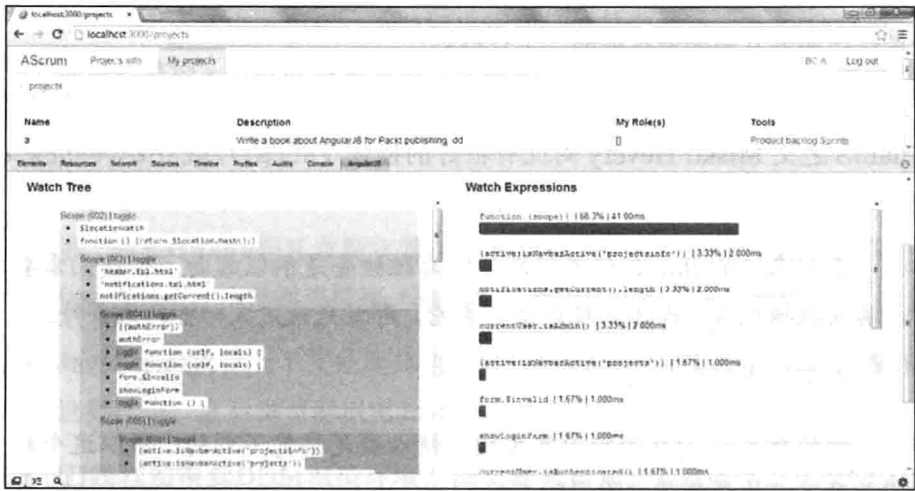


图11-1

Batarang 可以让我们轻松地找到执行最慢的监视表达式，并且可以在性能优化过程中测量优化效果。

11.3 AngularJS 应用的性能优化

Performance tuning of AngularJS applications

我们都希望应用很“快”，但是“快”对于不同的人可能有不同的意思。这就是为什么我们需要专注在那些影响整个应用性能的可测量的问题上。本章我们将专注于浏览器内的性能问题，而把网络相关的问题放在一边（网络相关问题将在第 12 章讨论）。如果把性能问题集中在浏览器内，那么需要关注 CPU 使用率和内存消耗量。

优化 CPU 使用率

Optimizing CPU utilization

除开应用逻辑，还有一些 AngularJS 特有的操作会耗尽 CPU 资源。`$digest` 循环值得我们特别关注。需要确保单个 `$digest` 循环运行得够快，还需要认真组织代码以保证 AngularJS 尽可能少地进入 `$digest` 循环。

加速 `$digest` 循环

需要努力保证 `$digest` 循环的执行时间少于 50 毫秒，这样人的肉眼就不会注意到它的运行。这点很重要，因为如果 `$digest` 循环快到让人不注意，那么应用就会被认为是对用户操作（DOM 事件）快速响应的。

可以在两个主要方向上努力，以便保证我们在“50 毫秒 / 每次 `$digest` 循环”的雷达监控下安全飞行：

- 让每个监视表达式运行得更快；
- 限制每个 `$digest` 循环需要运算的监视表达式数量。

1. 保持监视器简单快速

下面回忆监视表达式的语法：

```
$scope.$watch(watchExpression, modelChangeCallback)
```

你应该还记得每个单独的监视器都由两部分组成：`watchExpression` 用来比较模型值的变化，`modelChangeCallback` 在模型发生变化时被调用。`watchExpression` 远比 `modelChangeCallback` 执行得更加频繁，所以我们特别关注 `watchExpression`。



一个 `watchExpression` 在一次 `$digest` 循环中至少会被执行一次（通常是两次）。如果引入一个需要大量运行时间的监视表达式，就会拖慢整个 AngularJS 应用。所以，我们要特别关注监视表达式，避免引入极其耗时的表达式。

原则上，监视器的表达式部分应该尽量保持简单，运行要尽可能地快。下面介绍几种应该避免的错误模式以保证监视表达式的运行速度。

首先，应该尽量减少监视表达式中的耗时计算。通常在模板中使用简单的表达式，这种情况下监视表达式运行很快，但在两种情况下我们很容易引入“不简单”的表达式。

你应该特别关注那些执行函数的表达式，例如，

```
{{myComplexComputation()}}
```

这样的语法会导致 `myComplexComputation()` 成为监视表达式的一部分！在表达式中使用函数的另一个非显而易见的结果就是，我们可能不小心在函数中留下一些记录日志的语句。事实证明 `console.log` 会明显拖慢监视器的运行速度。看如下两个函数：

```
$scope.getName = function () {
    return $scope.name;
};

$scope.getNameLog = function () {
    console.log('getting name');
    return $scope.name;
};
```

使用如下的标签代码，然后观察 Batarang 的 `performance` 标签中的信息：

```
<span>{{getName()}}</span>
<span>{{getNameLog()}}</span>
```

对比的结果极其夸张，如图 11-2 所示。

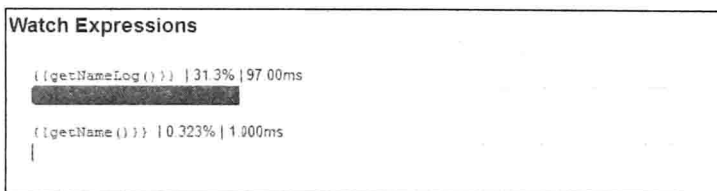


图11-2



谨记一定要在生产代码中删除所有 `console.log` 调用，你可以使用 `jshint` 等工具帮助你处理这个任务。永远不要在性能优化中使用日志记录语句，因为这些语句会严重影响测量结果。

过滤器是另一个我们会不知不觉引入的耗时计算。请看下面的例子：

```
{{myModel | myComplexFilter}}
```

过滤器只不过是使用特殊语法引入 AngularJS 表达式的一种函数。由于它是表达式的一部分，所以在一次 `$digest` 循环中，它也会被至少执行一次（通常是两次）。如果过滤器中的逻辑很复杂，就会拖慢整个 `$digest` 循环。过滤器的性能问题极其棘手，因为它们可以声明对服务的依赖，从而引入极其耗时的服务方法。



每个 AngularJS 表达式中的过滤器在每次 `$digest` 循环中至少都会被执行一次（通常是两次）。所以在过滤器方法中对逻辑进行监视可以保证过滤器不拖慢应用。

避免在监视表达式中访问 DOM

有时你可能禁不住要在 `watchExpression` 中访问 DOM 属性。这种方法有两个问题。

首先，访问 DOM 属性很慢很慢。这个问题是因为 DOM 属性是在被访问时实时计算实时更新的。只需要监视一个 DOM 属性，就足以使表达式运算变得极其耗时，从而拖慢整个 `$digest` 循环。例如，在监视表达式中获取元素的位置和尺寸属性，这会强迫浏览器重新计算元素的位置和大小，这比访问 JavaScript 属性慢好几个数量级。



任何DOM操作都比较慢，而计算DOM属性则格外地慢。最本质的问题是DOM是用C++实现的。在JS中调用一个C++方法是非常耗时的。所以任何对DOM的访问都比访问JavaScript对象慢好几个数量级。

第二个问题是理念问题。整个 AngularJS 哲学基于一条原则：真理的源头是模型。也就是说，模型驱动了声明式 UI。但是，如果在表达式中访问 DOM 属性，就让事情本末倒置了！突然 DOM 开始驱动模型后模型又返回来驱动 UI。我们将自己引入了一个相互循环依赖的绝境。



有时可能不得不监视DOM属性的变化，尤其在将第三方JavaScript组件集成到AngularJS应用中时。要注意这种做法会对\$digest循环的性能造成极大的影响。所以尽量避免在表达式中监视DOM属性变化，或者至少在引入这种监视器之后再测量性能表现。

2. 限制监视器的数量

如果你对现有的监视做过调优，清除了它们的所有性能瓶颈，但应用还需要更大的性能提升，那么就需要采取一些激进措施了。

删除不必要的监视

AngularJS 的双向数据绑定特性强大而易用，你可能不小心就会陷入任意滥用的境地，在那些静态值能应付的场合也使用数据绑定。

第 10 章讨论过一个使用数据绑定来实现翻译的例子。翻译信息很少改变（甚至永远都不会变）。给每个需要翻译的字符串使用 AngularJS 表达式，会导致在每次 \$digest 循环中需要进行大量的运算。



每当我们向模板中追加一个插值表达式时，试着评估它是否能受益于双向数据绑定。或许模板中的这个值还可以通过服务器端生成。

用心琢磨 UI 界面

每个注册在作用域上的监视表达式都代表页面上的一个“运转部件”。如果我们删除了所有不必要的数据绑定（不必要的数据绑定，就是上一节所说的那些数据不会发生变化的东西），就只剩下一个“每个页面 2000 个运转部件”的理论限制。但实际情况比理论复杂得多，所以我们要将每个单独监视器的速度也考虑进去。

2000 是一个很好的指标。但 2000 是一个合理的数字吗？这取决于你问谁，以及你所开发的应用的类型，但是我们应该好好想想我们的用户。他们能一次处理 2000 个“运转部件”吗？我们能否为他们提供最佳的用户体验？应用的 UI 界面是否应该按照提供必要数据和交互控制的原则重新评估？用户界面设计原则之一的“可见性原则”如是说（http://en.wikipedia.org/wiki/Principles_of_user_interface_design）：

设计应该展示与任务相关的所有必需的选项和材料，不要展示与主题无关的或多余的信息来干扰用户。好的设计绝不会使用不必要的信息来淹没或迷惑用户。

精简 UI 既可以提升 \$digest 循环的速度，又可以给用户提供更好的用户体验。这条路值得探索！

不监视隐藏元素

AngularJS 提供两个指令 ng-show 和 ng-hide，以供在需要按条件显示 DOM 的场景下使用。我们在第 4 章介绍过这两个指令并不会将隐藏元素从 DOM 中删除，而是会通过设置 style 属性（display: none）将元素隐藏起来。所以，本质就是这些“隐藏”元素仍然存在于 DOM 中，由这些元素（或者其子元素）注册的监视器依然会在每次 \$digest 循环中被执行。

我们来看看下面的代码以及对应 Batarang 中的性能表现：

```
<input ng-model='name'>
<div ng-show="false">
  <span>{{getNameLog()}}</span>
</div>
```

这里的 `getNameLog()` 函数只是简单地返回一个名词，在函数中使用了 `console.log` 来模拟一个耗时操作。如果我们开始在输入框中打字，就能看到 `getNameLog()` 表达式在每次击键时都会被调用。而实际上这个表达式的值在屏幕上根本看不到，如图 11-3 所示。

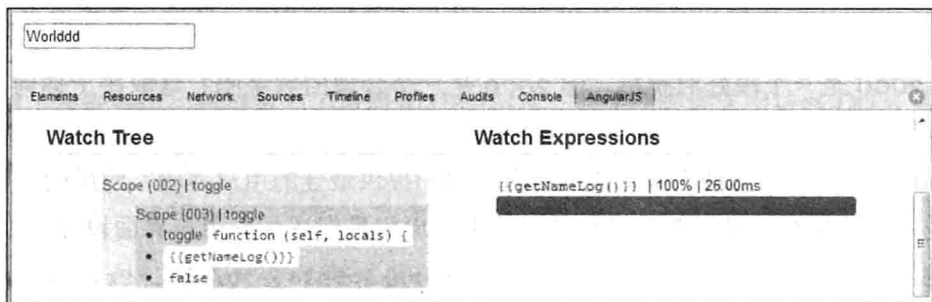


图11-3



注意，那些耗时的监视表达式可能在屏幕上没有任何对应的视觉展示。如果页面中隐藏的部分拖慢了应用的速度，那么可以考虑使用 `ng-switch` 系列指令。这类指令会将需要隐藏的元素从 DOM 树中完全删除。

在受影响作用域明确的情况下使用 `Scope.$digest` 替代 `Scope.$apply`

在本章的开头，我们介绍了 AngularJS 在执行一次 `$digest` 循环时需要遍历整个应用的所有作用域。这样做的目的是应对一个作用域的变化可能触发其父作用域同时变化的情况。但有时我们明确知晓某个改变会对那个作用域造成影响，此时就可以利用这一优势。

如果明确知道那个作用域会受到模型改变的影响，那么可以在最顶层的受影响作用域上调用 `scope.$digest` 方法，而不是调用 `scope.$apply`。`scope.$digest` 会在指定的作用域子集上运行 `$digest` 循环。与此对应，只有在这个作用域（及其子作用域）上注册的监视器才会检查模型的变化。这种方法可以极大地减少运行时监视表达式的数量，提升 `$digest` 循环的速度。

列举一个限定受影响作用域的具体例子，来看看自动完成的（autocomplete）指令 `typeahead`，效果如图 11-4 所示。

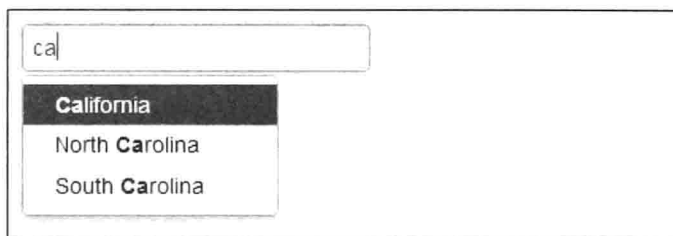


图11-4

在这个指令中，显示自动完成建议的浮层有它自己的作用域，用于持有所有的建议条目及对当前选中条目的引用（California）。用户可以通过键盘（上方向键或下方向键）在建议列表中更改自己的选择。当检测到一个键盘事件时，需要调用 `scope.$apply()`，之后 AngularJS 的双向数据绑定机制介入，对应的条目被高亮选中。但是，键盘导航只会影响浮层元素对应的作用域，所以每次击键都运行整个应用中所有作用域上的所有监视器明显是极大的浪费。我们可以更机智一些，只在浮层作用域上调用 `$digest()` 方法。

删除不用的监视器

你可能会遇到一种情况，那就是注册的监视器只需要在应用运行的某段时间内发挥作用。AngularJS 可以注销那些已经废弃的监视器，语法如下：

```
var watchUnregisterFn = $scope.$watch('name', function (newValue,
oldValue) {
    console.log( "Watching 'name' variable" );
    ...
});

//稍后，当不再需要该监视器时
```

```
watchUnregisterFn();
```

如上所示，`scope.$watch` 方法会返回一个函数，该函数即可用来在不需要监视时注销之前注册的监视器。

尽可能少进入\$digest循环

通常，\$digest 是由 AngularJS 内置的指令或服务触发的，所以我们无需手动控制。但我们还是应该注意触发 \$digest 循环的具体情况。AngularJS 指令和服务一般监听 4 种事件，以便触发调用 scope.\$apply() 方法。

- 导航事件：用户点击超链接，使用前进或后退按钮等情况。
- 网络事件：所有对\$http服务（以及\$resource服务）的调用在得到响应（不论成功或失败）后都会触发\$digest循环。
- DOM事件：当某个具体DOM事件发生时，与DOM事件对应的AngularJS 指令（ng-click、ng-mouseover等）就会被调用。
- JavaScript计时器：在JS计时器完成后，由setTimeout封装而来的 \$timeout服务就会触发一次\$digest循环。

如上所述，\$digest 循环会因为各种事件而经常运行并执行所有监视表达式，尤其是有很多 DOM 事件可以启动循环。对此我们没有太多办法，但是有一些技巧可以帮助我们减少 AngularJS 进入 \$digest 循环的频次。

首先，可以试着减少网络事件调用的次数，可以重新组织后端 API，将多个用户操作所需的结果合并在一个 XHR 响应中返回。当然这种方法不一定总能实现，但如果你对后端有完全的控制权，则可以考虑试试这种方法。这种方法不仅能限制网络调用的次数，同时也能减少 AngularJS 进入 \$digest 循环的次数。

其次，要特别注意计时器的使用，尤其是对 \$timeout 服务的使用。默认情况下，\$timer 服务会在计时器完成之后调用 scope.\$apply，如果不够细心，则可能会造成无法预料的后果。我们通过一个显示当前时间的简单 clock 指令来看看可能会出现的问题：

```
.directive('clock', function ($timeout, dateFilter) {
  return {
    restrict: 'E',
    link: function (scope, element, attrs) {

      function update() {
        // 获取当前时间并格式化，然后将其更新到DOM文本中
      }
    }
  }
})
```



```

        element.text(dateFilter(new Date(), 'hh:mm:ss'));
        //1秒钟后重复该操作
        $timeout(update, 1000);
    }

    update();
};
}))

```

这样一个指令，使用 `<clock></clock>` 标签初始，就会在每秒触发一次 `$digest` 循环。这就是为什么我们要向 `$timeout` 服务传递第 3 个可选参数，这个参数用于设定在计时服务中是否调用 `scope.$apply`。所以，编写 `update()` 函数的更好写法如下（该简单指令不需要数据双向同步）：

```

function update() {
    element.text(dateFilter(new Date(), 'hh:mm:ss'));
    $timeout(update, 1000, false);
}

```



可以通过在注册计时器时传递第 3 个参数（`false`）来避免触发 `$digest` 循环。

最后，可能在注册大量特定的 DOM 事件处理函数后，从而触发极大数量的 `$digest` 循环，尤其是与鼠标移动相关的事件（触发频繁）。例如，一个声明式的、AngularJS 特色的指令，当鼠标移动到元素上时修改元素的 `class`：

```

<div ng-class='{active: isActive}' ng-mouseenter ='isActive=true' ng-
mouseleave='isActive=false'>Some content</div>

```

当上面的代码运作时，每当鼠标滑过该元素，就会触发一次 `$digest` 循环，重新计算该元素的 CSS 类名。这种写法在少量使用（在少数 DOM 元素上偶尔使用）时不会产生多少问题，但如果在大批量元素上使用这种方法就会拖垮应用。如果你开始注意到有与鼠标事件有关的性能问题，那么可以考虑编写自己的自定义指令，在其中你就能根据 DOM 事件和模型变化直接进行 DOM 修改。

限制每个\$digest循环的执行轮数

如果模型很难进入稳定状态，那么要执行多轮 \$digest 循环。结果就会导致每轮循环中所有监视表达式都会被重新运算一次。通常，一个 \$digest 循环需要执行两次，但是，如果模型永远无法稳定，那么这个数字就会迅速上涨到最大限制（默认 10 次）。

认真想想要让一个新注册的监视表达式进入稳定状态需要哪些必要条件。高度不稳定的表达式不应该被引入监视表达式，而应该在 \$digest 循环之外对其单独计算。

优化内存占用 Optimizing memory consumption

AngularJS 使用脏检查机制来判定某个模型是否发生了变化，以及是否对比采取对应动作（更新 DOM 属性，修改其他模型值等）。高效的模型值对比算法对脏检查机制来说至关重要。

尽可能避免深度监视

AngularJS 默认使用一致性比较（针对对象）或者值比较（针对基本类型）来检测模型变化。这类比较快速直接，但有些情况下我们要对对象的属性逐个对比（称为深度比较）。

下面以一个典型的拥有多个不同属性的 User 对象为例：

```
$scope.user = {
  firstName: 'AngularJS',
  lastName: 'Superhero',
  age: 4,
  superpowers: 'unlimited',
  //更多其他属性
};
```

假设要在某个模型变量上持有 user 的全名（full name），当用户的姓（first name）或名（last name）发生变化时自动更新全名，那么应该在 \$scope 上注册一个新的监视器：

```
$scope.$watch('user', function (changedUser) {
  $scope.fullName =
    changedUser.firstName + ' ' + changedUser.lastName;
}, true);
```

可以给 `$watch` 方法提供第三个参数 (`true`)，以声明 AngularJS 应该使用深度比较。这样，AngularJS 就会使用 `angular.equal` 函数对整个对象进行深度比较（对对象的每个属性逐个比较）。在深度监视模式中对比运算过程会比较慢。不仅如此，这种模式下 AngularJS 需要拷贝和保存整个对象以供将来的对比计算。从内存占用角度来看这是很明显的浪费，我们不停地拷贝和存储对象的所有属性，但实际上我们只对其中的一部分感兴趣。

幸好有很多方案可以替代深度监视，如果我们只关心对象属性中的一部分，那么这些方法也很简单易用。比如，可以监视全名的计算函数：

```
$scope.$watch(function(scope) {
    return scope.user.firstName + ' ' + scope.user.lastName;
}, function (newFullName) {
    $scope.fullName = newFullName;
});
```

这种方法的优势是，内存中只会存储全名的运算结果以备下次对比。缺点就是，即使用户属性没有发生变化，全名运算函数在每次 `$digest` 循环中都会被执行。使用这种技术可以减少内存使用，但代价是拖累了 CPU 的运算周期。不过，可以通过在模板的 AngularJS 表达式中引入一个函数来实现同样的效果：

```
{{fullName()}}
```

`fullName()` 函数的定义如下：

```
$scope.fullName = function () {
    return $scope.user.firstName + ' ' + $scope.user.lastName;
};
```



深度监视在性能上有双重缺陷（既耗内存又耗CPU）。AngularJS 不仅要在内存中存储对象的拷贝以供下次对比使用，而且实际的对比检查过程也很慢。如果你只关心对象中的一部分属性，那么可以试试上面介绍的两种替代方案。

性能优化是一种微妙的平衡。在上面的这些例子中，我们可以看到用 CPU 的消耗换取内存空间。你在应用中所采取的最佳解决方案取决于具体的性能瓶颈问题。

注意监视表达式的大小

除了避免使用深度监视模式，还应该思考 AngularJS 监视和比较的真实值是什么。这个问题不是很明显，只有在处理模板时，这个监视表达式的问题才会露出端倪。

看看下面的例子，一长段文字中间有一个 AngularJS 表达式：

```
<p>This is very long text that refers to one {{variable}}
defined on a scope. This text can be really, really long and
occupy a lot of space in memory. It is so long since... </p>
```

你会期望 AngularJS 这样处理这个模板：创建一个表达式为 variable 的监视器。但 AngularJS 并不会这么做，<p> 标签中的整段文字会被当做表达式。这表示这个长文本会在内存中被存储和拷贝。可以使用 标签将需要数据绑定的部分文字围起来，这样就可以减小表达式的长度：

```
<p>This is very long text that refers to one <span ng-
bind='variable'></span> defined on a scope. This text can be
really, really long and occupy a lot of space in memory. It is
so long since...
</p>
```

这样一个简单的修改，就会让 AngularJS 注册一个只包含 variable 变量的监视表达式。

ng-repeat 指令

The ng-repeat directive

如果组织一场比赛，那么 ng-repeat 指令应该可以获得最有用的指令的金牌。它拥有极其强大的功能，而它的语法却简单优雅。不过，ng-repeat 指令同时也是极其耗费性能的指令之一。有两个原因导致如此：首先，它需要在每轮 \$digest 循环时对一批元素集合进行检查。其次，当检测到变化之后，对应的 DOM 元素需要重绘，由此可能导致一系列的 DOM 操作。

ng-repeat指令中对集合的监视

ng-repeat 指令需要对它所迭代的集合进行监视。为了正常运转，ng-repeat 指令需要能识别出添加到集合中的元素，并能在集合内移动，或者整个删除元素集合。为了实现这点，该指令在每次 \$digest 循环中都执行一种精妙算法。我们不对该算法进行深究，但它的性能表现取决于集合的大小。

瞬间绑定大量监视

当在一个极其庞大的数据集合上使用 `ng-repeat` 指令时，可以导致瞬间在作用域上注册大量监视器。

这里做一个简单计算。假设有一个简单的 5 列表格，每一列至少有一个数据绑定。这表示表格的每一行都会有 5 个数据绑定。假设 AngularJS 的理论上限是 2000 个数据绑定，那么这个表格最大限度能容纳 400 行数据。当然，如果表格列数大于 5 列（或者每列的数据绑定大于一个），那就会导致相应地缩减表格的行数。



超过 500 行的数据对 `ng-repeat` 指令来说就不太适用了。具体的数据量限制取决于绑定数据的数量，但我们不能期望有成千个元素的页面有多好的性能表现。

不幸的是，针对使用大数据量集合的 `ng-repeat` 指令，我们其实做不了什么。我们应该在将数据集合丢给 `ng-repeat` 之前，尽量对其进行预过滤或修正处理。所有类似技巧都值得一试：过滤、分页等。

如果开发需求确实需要在页面上一次性显示数以千计的数据，那么 AngularJS 内置的 `ng-repeat` 指令或许不是你的最佳选择。在这种情况下，你应该开发自己的自定义指令。你的自定义指令不要为每个元素创建双向数据绑定，而只是根据数据集合渲染出对应的 DOM 元素。这样可以避免创建大量的数据绑定。

11.4 小结 Summary

本章对 AngularJS 的内部结构进行了深入介绍。我们需要对 AngularJS 的内部运作机制有一个清晰的了解，以便理解它的性能特征和理论上的限制。

所有性能相关的提升都必须从严谨的测量体系开始，以便定位和理解已存在的瓶颈问题。没有过硬测试数据的性能优化就像是黑夜里乱打枪。幸运的是，我们有一个极其优秀的 Chrome 扩展——Batarang，它能检测应用的运行数据。

应该特别注意 AngularJS 中 `$digest` 循环的执行时间，因为它会影响到用户对整个应用速度的感知。如果 `$digest` 循环的执行时间超过 50~100 ms，那么用户就会感觉到应用的反应有些迟钝。这就是为什么我们在本章花了很多时间来讨论 `$digest` 循环的运转原理以及如何提升 `$digest` 的执行速度。

`$digest` 循环的执行速度与监视器的数量以及监视器的执行时间成正比。可以通过减少监视器的数量或加快监视器的运行速度来提升 `$digest` 循环的性能。也可以尽量少进入 `$digest` 循环。

内存消耗是整个性能优化课题中另一个需要检测的内容。在 AngularJS 代码中，宝贵的内存会被深度监视快速耗尽。我们应该尽可能避免使用深度监视，不仅要避免手动创建 (`scope.$watch`)，也要确保不会无意间在模板中创建。

从 CPU 使用和内存消耗两方面来看，`ng-repeat` 指令都对性能有明显影响。只要在数百条数据集上使用 `ng-repeat` 指令，就会轻松地造出一个性能危险地带。为了解决这个问题，你需要使用预过滤器来使数据集保持尽可能小。如果你确实需要显示数百个元素，那么应该考虑编写自己的自定义指令，这样做性能优化的效果更好。

第 12 章将讲解如何把一个经过性能优化的应用部署到生成环境，同时也会讨论与网络相关的性能问题。

第 12 章

打包和部署 AngularJS Web 应用

Packaging and Deploying AngularJS Web Applications

经过辛苦的编码、测试和性能优化之后，终于到了正式发布应用的时候了。但是，别急，在应用闪亮面世之前，还需要处理一些小细节。

首先，要保证应用合理地使用网络资源。这可以通过限制 HTTP 请求的数量以及控制每个单独请求的数据量大小来实现。预加载和压缩静态资源是节省网络流量的两种常用技术，我们会在本章学习如何在 AngularJS 应用环境中使用这些技术。

web 应用的首页决定了用户的第一印象和期望。如果初次体验不好，那么用户可能会感到气馁，并且不再使用我们为之付出心血的应用。所以，优化应用的首页极其重要。本章将讨论各种调整优化方法，以保证首页用户体验良好。

本章的最后一节将讨论 AngularJS 对不同浏览器的支持情况，并重点讨论对 Internet Explorer 的支持。

本章将学习如下内容。

- 如何压缩和合并JavaScript代码，以及如何预加载局部模板，以优化下载静态资源时的网络利用率。
- 如何优化首页。
- AngularJS支持哪些浏览器，以及要让它Internet Explorer中正常运行需要做哪些必要的处理。

12.1 提升网络相关的性能

Improving network-related performance

作为 web 开发者，我们应该为用户提供带有友好用户界面的 web 应用程序。即使在应用正式启动之前，用户也应该获得优雅的体验。因此，可以通过减少每个请求中的下载数据量和限制 HTTP 请求的数量来减少 HTTP 流量，以缩短应用的加载时间。

压缩静态资源

Minifying static resources

缩减网页加载时间的方法之一，就是减少 web 服务器和浏览器之间传输的字节数量。如今压缩 JavaScript、CSS 和 HTML 已经是 web 开发中的一个标准环节。压缩可以减少需要下载到浏览器端的数据量，此外，还会使网页源代码难以阅读，并对那些不怀好意的目光增加一点防护措施。

可以继续安全地使用那些你喜欢的 CSS 和 HTML 压缩技术，而“刑侦队”含有 AngularJS 的 JavaScript，在使用常见压缩工具处理之前有一些注意事项。

AngularJS如何判断依赖关系

在整个框架中，AngularJS 都严重依赖“依赖注入”模式。在 AngularJS 代码中，可以很容易地通过注册服务来表明依赖关系，并将依赖的服务注入代码中。

AngularJS 可以判断出一个指定函数的依赖关系，然后通过 `$injector` 服务获取这些依赖服务，之后将这些作为参数提供给函数。AngularJS 使用了一种出奇简单的技术来判断函数的依赖关系。我们以 SCRUM 示例应用中的控制器为例来看看这种技术：

```
angular.module('projects', ['resources.projects'])
  .controller('ProjectsViewCtrl', function ($scope, $location) {
    //控制器的实现代码
  });
```

`ProjectsViewCtrl` 函数的参数声明该函数依赖于两个服务：`$scope` 和 `$location`。AngularJS 会将函数定义转换为字符串，然后使用正则表达式来匹配依赖函数的名称，从而解析出依赖关系。可以通过一个小例子来看看这种技术的原理：


```
var ctrlFn = function($scope, $location) {};
console.log(ctrlFn.toString());
```

执行上面的代码，就会在控制台上将整个函数体以字符串形式记录下来：

```
"function ($scope, $location) {}"
```

在将函数体转换为一个字符串之后，就可以通过简单的正则表达式匹配来解析出参数的名称，进而找出依赖服务的名称。

现在，我们来思考对上例中的函数定义执行 JavaScript 压缩会有什么效果。虽然每个压缩工具的具体功能稍有不同，但大多数压缩工具都会对函数的参数重新命名，所以压缩后的代码看起来是这样的：

```
angular.module('projects', ['resources.projects'])
  .controller('ProjectViewCtrl', function (e, f) {
    //压缩后的代码重命名了参数名称
  });
```

很明显，AngularJS 无法在压缩代码中找出依赖关系，因为 e 和 f 并不是有效的服务名称。

编写会被安全压缩的JavaScript代码

标准的 JavaScript 压缩处理会破坏 AngularJS 用于判断依赖关系所需的参数信息。由于压缩后的代码会丢掉参数信息，所以需要使用其他方法提供依赖关系提示。


AngularJS 针对压缩代码提供了几种不同的方法来声明依赖关系，我强烈建议你使用数组风格的参数定义，代码如下：

```
angular.module('projects', ['resources.projects'])
  .controller('ProjectViewCtrl', ['$scope', '$location', function
($scope, $location) {
    //控制器的实现代码
  }]);
```

压缩后的代码如下：

```
angular.module('projects', ['resources.projects'])
  .controller('ProjectViewCtrl', ['$scope', '$location',
function (e, f) {
    //控制器的实现代码
  }]);
```

即使 `function` 的参数名称被修改了，压缩工具也不会修改数组中的元素，这样 **AngularJS** 就有足够信息找出指定函数的依赖关系。

 如果想让代码被安全压缩，那么就使用数组替换掉函数（至少有一个依赖）原先的参数定义，该数组中应包含依赖服务对应的函数名称，最后一个数组元素应该是当前函数本身。

数组风格的依赖注入语法有时看起来会觉得有点奇怪，所以，我们来快速浏览示例 **SCRUM** 应用中的几个具体例子。接下来的小节将展示几种最常用的使用范例。

为示例 **SCRUM** 应用编写的代码中包含有数组风格的依赖注入语法。你可以去 [GitHub](#) 上参阅具体的代码样例，了解这种语法在不同情况下的各种用法。

模块

模块上的 `config` 和 `run` 函数可以注入依赖。下面的代码展示了如何使用安全压缩的方法来定义配置：

```
angular.module('app')
  .config(['$routeProvider', '$locationProvider', function
($routeProvider, $locationProvider) {
  $locationProvider.html5Mode(true);
  $routeProvider.otherwise({redirectTo: '/projectsinfo'});
}]);
```

`run` 函数的安全写法如下：

```
angular.module('app')
  .run(['security', function(security) {
    security.requestCurrentUser();
  }]);
```

provider

AngularJS 提供了几种注册 **provider** 的方法（也就是对象实例化的方法）。其中，工厂函数应该是创建单例对象最常用的方法，代码如下：

```
angular.module('services.breadcrumbs', [])
  .factory('breadcrumbs', ['$rootScope', '$location',
    function($rootScope, $location){
      . . .
    }]);
```

在服务上定义修饰器（**decorators**）可以给已有代码扩展更多功能。定义修饰器的语法不是很直观，尤其是使用数组风格的依赖注入语法时。为了让你逐渐习惯这种语法，下面展示了使用数组风格语法进行依赖注入的修饰器：

```
angular.module('services.exceptionHandler')
  .config(['$provide', function($provide) {
    $provide.decorator('$exceptionHandler', ['$delegate',
      'exceptionHandlerFactory', function ($delegate,
      exceptionHandlerFactory) {
        return exceptionHandlerFactory($delegate);
      }]);
  }]);
```

指令

使用安全压缩方法定义指令的语法和定义其他 **provider** 没什么区别。例如，第 9 章中的 **field** 指令安全定义如下：

```
.directive('field', ['$compile', '$http', '$templateCache',
  '$interpolate', function($compile, $http, $templateCache,
  $interpolate) {
  . . .
  return {
    restrict: 'E',
    priority: 100,
    terminal: true
    . . .
  };
}]);
```

定义指令控制器时要稍微注意，因为刚开始看到它的语法可能会觉得有点奇怪。下面展示了 accordion 指令（详见第 9 章）的控制器定义：

```
.directive('accordion', function () {
  return {
    restrict: 'E',
    controller: ['$scope', function ($scope) {
      // 这个数组记录着手风琴组的信息
      this.groups = [];

      . . .
    }],
    link: function(scope, element, attrs) {
      element.addClass('accordion');
      . . .
    }
  };
});
})
```

数组风格依赖注入的缺陷

数组风格的依赖注入可以让 AngularJS 代码被安全压缩，但伴随的是代码重复。确实，我们需要把每个函数的参数都写两遍：函数定义中写一遍，数组中再写一遍。代码重复是个大问题，尤其是在重构已有代码时更加棘手。

在这种重复写法中，很容易忘记添加一个新参数或在数组中遗漏新参数，也很容易把参数顺序搞错，代码如下：

```
angular.module('app')
  .config(['$locationProvider', '$routeProvider', function
($routeProvider, $locationProvider){
  $locationProvider.html5Mode(true);
  $routeProvider.otherwise({redirectTo: '/projectsinfo'});
}]);
```

因此，编写数组风格的依赖注入要非常细心，因为由参数问题导致的错误很难定位。^[1]

[1] 译者注：只有开发者自己细心一些，暂时没有办法解决这个问题。



可以尝试使用编译工具，让编译工具在编译时处理你的代码，自动添加依赖注入声明，这样就可以减轻你使用数组风格依赖注入的痛苦。编写这类编译工具也不是一件小事（需要对JavaScript代码做详细分析），所以目前还未被广泛使用。如果你的编译系统是基于Grunt.js的，则可以尝试用ngmin (<https://github.com/btford/ngmin>) 执行一个Grunt.js任务 (grunt-ngmin)。

模板预加载 Preloading templates

使用 AngularJS 之后，就有很多机会将 HTML 代码拆分成微小的、可复用的局部模板。AngularJS 允许我们将每个局部模板存储为单独文件，然后在需要时下载。使用大量局部模板的单独小文件，从代码组织的角度来看是对开发有好处的，但可能对性能存在负面影响。

来看一个典型的路由定义，代码如下：

```
angular.module('routing_basics', [])
  .config(function($routeProvider) {
    $routeProvider
      .when('/admin/users/list', {templateUrl: 'tpls/users/list.html'})
      . . .
  })
```

如果将路由模板保存为一个单独文件 (tpls/users/list.html)，那么 AngularJS 在跳转到该路由之前必须先下载这个文件。通过网络下载模板文件，很明显会导致在 UI 渲染之前必须额外等待一段时间。在网络通信上花费这个额外的时间，就无法为用户提供快速响应的 UI 界面。

路由 (route) 定义只是这种局部模板问题的其中一个例子而已。ng-include 指令也可以引用模板，自定义指令定义的 templateUrl 属性也可以。

模板预加载技术可以大幅减少网络通信，提升 UI 界面的响应速度。AngularJS 提供了两种不同的方法用于在初次使用模板之前对它们进行预加载：<script> 指令和 \$templateCache 服务。

AngularJS 非常智能,它能缓存所有已经请求过的模板(使用 `$templateCache` 服务)。简而言之,AngularJS 会在通过网络获取模板之前,先检查 `$templateCache` 服务已经缓存的内容。这样,同一个模板永远不会通过网络请求两次。所以,使用 `$templateCache` 服务,就可以确保所有局部模板在应用启动之后全部加载就绪,永远无需再通过网络下载。

使用<script>指令预加载模板

AngularJS 提供了一个非常好用的 `<script>` 标签指令,它可以用来将单独的模板预加载到 `$templateCache` 服务中。通常,我们会在应用的初始页面(`index.html` 或类似页面)中嵌入这些预加载模板。以之前讨论过的路由定义为例,可以按照如下格式嵌入模板:

```
<script type="text/ng-template" id="tpls/users/list.html">

  <table class="table table-bordered table-condensed">
    <thead>
      <tr>
        <th>E-mail</th>
        <th>Last name</th>
        <th>First name</th>
      </tr>
    </thead>
    <tbody>
      ...
    </tbody>
  </table>

</script>
```

要预加载一个局部模板,需要将其内容放在 `<script>` 标签中,并将其 `type` 属性设置为 AngularJS 特有的类型: `text/ng-template`。而局部模板的 URL 则必须以 `id` 属性值的方式来设定。



包含局部模板的 `<script>` 标签需要放置在含有 `ng-app` 指令的 DOM 元素之中作为其子元素。如果将 `<script>` 标签放置在 AngularJS 维护的 DOM 树之外(不作为含有 `ng-app` 指令的 DOM 元素的子元素),那 AngularJS 就无法识别它,结果就是该模板不会被预加载。在该模板被初次使用时,AngularJS 仍会尝试通过网络去下载该模板。

在 `index.html`（或类似文件）文件中维护所有嵌入的局部模板会非常麻烦。在开发过程中，还是更倾向于将每个模板都存为一个单独的文件。这样，最终的 `index.html` 文件应该是在构建过程中生成的。

填充 `$templateCache` 服务

使用 `<script>` 指令预加载局部模板对于少量的模板来说是很好的，但针对稍大一些的项目就会失控。这种方法最主要的问题是要将生成的 `index.html` 纳入构建流程中。而 `index.html` 文件中通常会包含大量的手写代码，所以这样会存在问题。将自动生成的代码和手写的代码混在一个文件中很不理想，最好是将每个需要预加载的模板都放在单独的文件中。使用 `<script>` 指令的另一个问题是在单元测试环境中该指令无法预加载模板。幸好有补救这两个问题的方案。

可以在应用启动时，预先填充 `$templateCache` 的内容。每条 `$templateCache` 服务内容都按照这样的格式：模板 URL 作为键名，模板内容（转换为 JavaScript 字符串）作为值。

例如，有一个模板包含如下内容：

```
<div class='hello'>Hello, {{world}}!</div>
```

该模板的 URL 是 `tpls/hello.tpl.html`，可以按照如下格式将其填充到 `$templateCache` 服务中：

```
angular.module('app', []).run(function($templateCache){
  $templateCache.put('tpls/hello.tpl.html',
    '<divclass=\'hello\'>Hello, {{world}}!</div>');
});
```

“手动”为 `$templateCache` 添加模板还是非常麻烦的，因为要小心处理转义引号（如上面代码中的 `class` 属性中的引号）。为了解决这个问题，我们又一次使用一个构建时任务，该任务会迭代查找工程中所有的模板，然后生成填充 `$templateCache` 所需的 JavaScript 代码。示例 **SCRUM** 应用就是使用这种方法——有一个专门的构建任务，用于生成一个带有 `run` 函数的模块（模块名称为 `templates`），该模块负责向 `$templateCache` 中填充内容。之后 `templates` 模块会和其他模块一起被设置为应用主模块的依赖模块，如下所示：

```
angular.module('app', ['login', 'dashboard', 'projects',
  'admin', services.breadcrumbs, 'services.i18nNotifications',
  'services.httpRequestTracker', 'directives.crud'
  'templates']);
```



向 `$templateCache` 中预加载局部模板在 AngularJS 社区中是很通用的做法，所以有一个专门的 Grunt.js 任务用于自动处理该流程： `grunt-html2js` (<https://github.com/karlgoldstein/grunt-html2js>)。这个 Grunt.js 任务的灵感就是从本书中的 SCRUM 应用的构建过程中来的。

在实际使用中，示例 SCRUM 应用的构建系统会为每个模板创建单独的 AngularJS 模块。在这种情况下，模板的 URL 就被作为模块的名称：

```
angular.module("header.tpl.html", [])
  .run(["$templateCache", function($templateCache) {
    $templateCache.put("header.tpl.html",
      "<div class=\"navbar\" ng-controller=\"HeaderCtrl\">" +
      . . .
      "</div>");
  }]);

angular.module("login/form.tpl.html", [])
  .run(["$templateCache", function($templateCache) {
    $templateCache.put("login/form.tpl.html",
      "<div modal=\"showLoginForm\" close=\"cancelLogin()\">" +
      . . .
      "</div>" +
      "");
  }]);
```

随后，所有单独的模板模块（template modules）都会被设置为 `templates` 模块的依赖模块，这样应用程序就只需依赖 `templates` 一个模块，从而引入所有模板，具体如下：

```
angular.module('templates', ['header.tpl.html', 'login/form.
tpl.html', ...]);
```


每个局部模板都对应一个模块，对单元测试来说是很便利的，可以精确控制在当前测试中 `$templateCache` 需要加载哪些模板。这样就可以保证每个单元测试都尽可能独立。

组合使用不同的预加载技术

和其他性能优化技术一样，模板预加载也是一种平衡的艺术。我们将模板放进了 `$templateCache` 服务中，从而减少了网络通信，提升了 UI 的响应速度，其代价则是增加了内存消耗。`$templateCache` 服务中的每一条内容都会增加内存消耗，其消耗量和模板大小成正比。所以，在庞大的应用（是指该应用中包含数以百计的布局模板）中这种方法存在隐患，尤其是针对那些包含多个不同逻辑区块的应用。根据具体应用和其使用方式的不同，就有可能出现用户根本无法访问应用中的大多数页面，这样，预加载许多用户用不到的模板完全是在做无用功。

幸运的是，我们在预加载模板时不必采取全有或全无的逻辑。可以组合使用不同的加载策略，例如，在应用初始加载时预加载最常用的模板，然后在需要时再加载其他模板。还可以在应用启动之后，在后台预取模板后将其放入 `$templateCache` 服务中。比如，当用户进入应用的某个特定模块之后，再预加载该模块所需的所有模板。

12.2 优化首页 Optimizing the landing page

优化首页的性能对任何 web 应用来说都是至关重要的。因为首页是用户看到的第一个页面，是应用的第一印象。

对于单页 web 应用来说，加载并“正确”显示首页稍微有点棘手。单页 web 应用通常会有大量的网络通信，在 JavaScript 驱动的渲染引擎工作之前需要下载很多脚本文件。本节将介绍一些 AngularJS 特有的技巧，用于提升用户对首页加载和显示的感知体验。

避免显示未经处理的模板

Avoid displaying templates in their unprocessed form

AngularJS 驱动的 web 页面在模板被处理和渲染之前，需要下载 AngularJS 代码和应用的业务代码。这意味着用户可能在加载瞬间看到 AngularJS 模板原始的未经处理的格式。以经典的“Hello World!”为例，假设下载 JavaScript 代码花费了很长时间，那么在 JavaScript 代码下载和解析完成之前，用户就会看到如图 12-1 所示的内容。

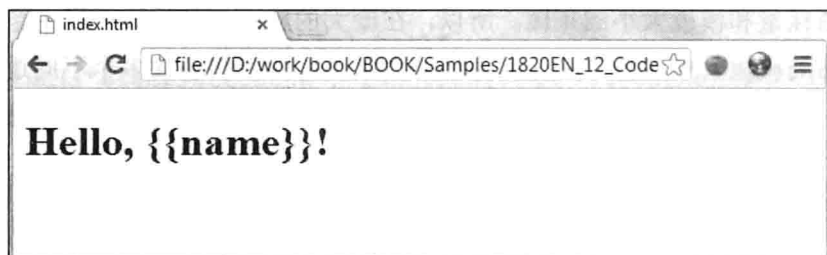


图12-1

在所有 JS 脚本下载完成之后，AngularJS 应用就会启动，图 12-1 中所示的插值表达式({{name}}) 也会被替换为作用域中对应的值。这个过程不够理想：一方面，用户可能会因为看到页面上显示的奇怪表达式而感到惊奇；另一方面，在表达式被替换为真实的插入值时，UI 界面也会闪烁。AngularJS 提供了 ng-cloak 和 ng-bind 两个指令用于帮助解决这类 UI 问题。

使用ng-cloak指令隐藏DOM元素

ng-cloak 指令可以隐藏 DOM 树中的任意节点，直到 AngularJS 为处理整个页面做好准备之后，才会将 DOM 元素显示出来。你可以隐藏 DOM 树中的任意元素，只需将 ng-cloak 指令放入对应 DOM 元素即可：

```
<div ng-controller="HelloCtrl" ng-cloak>
  <h1>Hello, {{name}}!</h1>
</div>
```

如果应用的首页是由很多动态部分组成的，那么可以将 ng-cloak 指令放置在 <body> 元素上，从而隐藏整个页面的内容。如果是另一种情况，首页是由静态内容和动态内容组合而成的，那么将 ng-cloak 指令放置在动态部分的元素上效果更好，因为这样用户在 AngularJS 应用加载和启动的同时就可以看到静态内容。

`ng-cloak` 指令使用 CSS 规则来隐藏那些包含动态内容的元素，然后在 AngularJS 准备就绪之后再将这些元素显示出来，之后再编译 DOM 树。隐藏 DOM 元素所对应的 CSS 规则（匹配 `ng-cloak` 属性），代码如下：

```
[ng\:cloak], [ng-cloak], [data-ng-cloak], [x-ng-cloak], .ng-cloak, .x-ng-cloak {
  display: none;
}
```

AngularJS 会动态创建如上所示的 CSS 规则，所以你无需手动定义。

当 AngularJS 加载完成并启动之后，它会逐层编译整个 DOM 树并连接所有指令。`ng-cloak` 指令既是一个常规的 HTML 属性，又是一个指令，当它执行完毕之后就会删除 DOM 元素上的 `ng-cloak` 属性。这样，匹配 `ng-cloak` 属性的 CSS 规则就无法应用到该 DOM 元素上，于是该元素就会显示出来。

使用 `ng-bind` 指令隐藏表达式

要在 AngularJS 准备就绪之前隐藏大块的 DOM 元素，`ng-cloak` 指令是一个不错的选择。`ng-cloak` 指令解决了 UI 界面闪烁的问题，但它并没有解决另外一个问题，用户在动态脚本完全解析之前无法看到任何内容。

理想情况下，应该在服务器端预渲染首页，然后在 AngularJS 准备就绪时将首页与动态 JavaScript 结合起来。有很多人都在尝试各种不同的服务器端渲染技术，但截至写作本书时，没有任何一种技术官方支持 AngularJS 的服务器端渲染。因此，如果应用首页主要是由静态内容构成的，那么可以使用 `ng-bind` 指令来替换插值表达式。

假设代码中主要是静态 HTML 代码，并夹杂有少许的插值指令，示例代码如下：

```
<div ng-controller="HelloCtrl">
  Hello, {{name}}!
</div>
```


可以使用 `ng-bind` 指令等价替换 `{{name}}` 表达式：

```
<div ng-controller="HelloCtrl">
  Hello, <span ng-bind="name"></span>!
</div>
```

使用 `ng-bind` 的好处就是那些使用花括号的 **AngularJS** 表达式不会在 **HTML** 中出现，这样，这些表达式也就永远不会以未经处理的面貌显示给用户。浏览器无法识别自定义属性 (`ng-bind`)，所以就会直接忽略它们，而 **AngularJS** 则会处理这些属性。使用该技术时甚至还可以通过原始的 **HTML** 代码为表达式提供默认值：

```
<div ng-controller="HelloCtrl">
  Hello, <span ng-bind="name">Your name</span>!
</div>
```

在 **AngularJS** 处理 `ng-bind` 指令之前，浏览器会显示出该默认值 (`Your name`)。

 此处所介绍的使用 `ng-bind` 指令实现隐藏表达式的技术，只应该在应用的首页中使用。在随后的页面中可以安全地使用插值表达式，因为 **AngularJS** 已经加载完成并准备就绪。


引入 **AngularJS** 和应用脚本文件

Including **AngularJS** and application scripts

这一小节是专门讲解如何优化首页的，如果不介绍脚本加载，那么内容就显得不够完整。如果你注意到异步脚本加载器日益流行的趋势，那么这个问题就更重要。

引用脚本文件

通常认为将 `<script>` 标签放在首页的最底部是个很好的做法。但这种方法和其他方法一样，应该按需评估恰当使用。将 `<script>` 标签放在页面底部，**JavaScript** 文件的下载和解析不会阻塞其他内容的下载，也不会阻塞 **HTML** 的渲染。这种方法适合于那些有大量静态内容而只有极少量 **JavaScript** 代码的页面。而对于那些高度动态的、由 **JavaScript** 驱动的 web 应用，这种方法则值得商榷。

 如果应用首页包含大量静态内容，只有少量动态内容，那么就将 `<script>` 标签放在页面最后，然后使用 `ng-bind` 指令以便在页面加载和处理过程中隐藏数据绑定。否则，就将 `<script>` 标签挪到 `<head>` 标签中，然后使用 `ng-cloak` 指令。

AngularJS和异步模块定义

异步模块定义 (asynchronous module definition, AMD), 是由 `Require.js` 和类似的 `JavaScript` 库推行的一种规范, 其目标是为异步加载的可复用 `JavaScript` 模块制定一套通用规则。AMD 模块可以按需加载, 并能实现各模块之间的相互依赖。另外, AMD 模块定义还可以被离线构建脚本用来合并特定应用功能所需的所有模块。

AngularJS 有它自己的模块系统。AngularJS 和 AMD 定义都使用了同一个单词 “module”, 但同一个词所指代的事物完全不同, 具体如下。

- AngularJS 模块 (modules) 用于定义不同的 `JavaScript` 类 (classes) 和对象 (objects) 在运行时应该如何组合在一起。AngularJS 模块不会执行任何脚本加载操作。相反, AngularJS 期望在应用启动之前所有的模块已经全部加载到浏览器中。
- AMD 模块主要关心脚本加载问题。AMD 定义允许将应用拆分成若干个更小的文件, 然后在需要时仅异步加载你所需的那部分文件。

AngularJS 模块和 AMD 模块解决的是完全不同的问题, 因此不要混淆。

在 AngularJS 的当前版本中 (1.1.x), 所有组建应用的脚本文件需要在应用启动之前全部加载到浏览器中。在这种情况下, 异步按需加载的模块完全没有用武之地。在 AngularJS 应用初始化之后, 根本就不可能再注册任何新的模块或 providers (服务、控制器、指令等)。



在当前版本的 AngularJS 中, 异步加载 AMD 模块只能用于在应用初始化之前加载 `JavaScript` 库, 或者提前加载应用程序代码。需要时才从服务器端异步加载应用程序代码是暂时不支持的。


AngularJS 模块缺少按需加载的功能, 这听起来好像是一个很大的缺陷, 但再说一次, 我们应该以需度物。AMD 模块主要用于解决如下几个问题。

- 异步加载脚本文件, 这样浏览器不会被阻塞, 可以同时并行加载若干其他资源文件。

- 根据用户使用应用的情景，按需加载JavaScript代码。这点有效地消除了提前加载应用程序代码的需求。
- 可以使用模块定义来指定模块之间的依赖关系，还可以设定生产环境中打包所需的模块。

可以在应用启动之前使用 AMD 模块来加载 AngularJS 库文件、所有第三方依赖文件及应用程序代码。这样做的好处是可以异步加载所有的脚本文件，避免多个 `<script>` 标签的阻塞。这可能会给应用性能带来积极影响（也可能不会），具体影响取决于加载文件的数量。

在使用 AMD 模块之后就不能再使用 `ng-app` 指令来启动 AngularJS 应用了。这是因为使用 `ng-app` 时 AngularJS 会在 `document-ready` 事件开始处理 DOM 树。此时异步模块可能还没有加载到浏览器中，而 AngularJS 则会尝试在所需 JavaScript 文件加载完成之前开始启动应用。为了解决这个问题，AngularJS 提供了一种手动的 JavaScript 启动方法：`angular.bootstrap`。

 如果决定在 AngularJS 应用中使用 AMD 模块，则需要删除 `ng-app` 指令，转而使用 `angular.bootstrap` 方法。使用这种方法就可以控制 AngularJS 启动应用的时机。

还应该看看按需加载脚本的问题。按需加载这种优化技巧是用更多的网络通信来换取更少的初始加载、更快的启动时间和更小的内存消耗。按需加载的假设前提是组建应用的 JavaScript 文件“足够大”，需要花费很长的网络通信时间。但通常情况下，使用 AngularJS 编写的代码相比使用其他替代框架编写的代码要少很多。这么少的代码，再经过压缩和 `gzipped`，最终的那点数据量对于网络下载和浏览器存储来说都是小菜一碟。

AMD 模块的创始人意识到异步加载太多小文件会产生大量的 XHR 请求。激增的网络通信所导致的性能损失可能会很严重，所以最好停止异步加载所有脚本文件，对它们进行适当合并。这也就是为什么会有一些 AMD 工具可以分析模块之间的依赖关系，然后为生成环境打包文件。精心设计的构建脚本也可以这样合并文件。在 AngularJS 应用中合并文件尤其简单，因为 AngularJS 模块被设计为每个单独模块都可以以任意顺序加载到浏览器中。



考虑到当前版本的AngularJS不支持异步按需加载代码，而异步加载器的配置比较烦琐，性能方面尚存争议，所以建议不要在AngularJS 1.1.x应用中使用异步加载器（如Require.js及类似库）。

12.3 浏览器支持 Supported browsers

AngularJS 的源代码一直在一个持续集成（continuous integration, CI）服务器中被严格地测试。框架的每一次代码变动都会触发一套全面的单元测试。任何代码都无法单独成为框架代码库的一部分，它必须带有自己对应的单元测试。这种严苛的单元测试策略保证了框架的稳定性和长期平滑的更新。

在写作本书的时候，AngularJS 的持续集成服务器在不同的浏览器中执行超过 2000 个单独的测试。测试的浏览器包括：最新版的 Chrome、Firefox、Safari、Opera 及 Internet Explorer 8、9、10。单元测试的数量和用于测试的浏览器数量让我们对框架的成熟度和稳定性有了更深的认识。AngularJS 保证能在上述所列的浏览器中正常工作。它很有可能还可以在那些未在此列出的现代浏览器（脑海中闪现出移动端浏览器）中运行。

在 Internet Explorer 中使用 Working with Internet Explorer

在支持 Internet Explorer 时会有一些特殊的注意事项，这应该不意外吧。针对 IE9 和 IE10，AngularJS 是开箱即用的，但针对 IE8 有一些特别注意事项。

通常我们使用 `ng-app="application_name"` 指令来启动 AngularJS 应用。不幸的是，在 IE8 中光这样还不行，还需要追加另一个属性：`id="ng-app"`。

如果不进行专门的设置，IE8 是无法识别自定义 HTML 标签的。例如，无法使用下面的标签来引入模板，除非我们先教会 IE8 识别 `<ng-include>` 标签：

```
<ng-include="'myInclude.tpl.html'"></ng-include>
```


要教会 IE8，需要先创建自定义 DOM 元素，代码如下：

```
<head>
  <!--[if lte IE 8]>
    <script>
      document.createElement('ng-include');
```

```

        document.createElement('ng-view');
        . . .
    <![endif]-->
</head>

```


 自定义元素应该在页面<head>部分的脚本中创建。这样可以确保在解析器看到新元素之前先运行代码创建它。

当然，还可以通过使用属性版的 `ng-include` 指令来直接绕过自定义 HTML 标签的问题，代码如下：

```
<div ng-include="'myInclude.tpl.html'"></div>
```

对 IE7 的支持存在很多问题。如果要在 IE7 中使用 AngularJS，则需要加上 IE8 所需的所有额外设置。但 IE7 并不在集成构建和测试的名单中，所以无法保证 IE7 能支持所有的 AngularJS 功能。除此之外，IE7 缺少很多现代浏览器普遍支持的 API。最好的例子就是 IE7 中没有实现 JSON API，所以需要引入一个补丁 (<http://bestiejs.github.io/json3/>)。

AngularJS 完全不支持 IE6。

 AngularJS 的文档中有一小节专门介绍了针对 IE8 和 IE7 需要哪些额外设置，文档地址：<http://docs.angularjs.org/guide/ie>。

12.4 小结 Summary

本章围绕部署相关问题做了详细介绍。首先，讨论了各种网络相关的优化技巧。一般来说，可以通过减少通信数据大小和限制请求数量来缩减浏览器花费在网络通信上的时间。

压缩 HTML、CSS 和 JavaScript 文件是一种常用的减少浏览器下载数据量的优化技巧。AngularJS 的依赖注入机制迫使我们必须使用安全压缩的方式来编写 JavaScript 代码。在具体实践中，我们应该在所有有依赖声明的函数中使用数组风格的依赖定义。

可以合并或预加载局部模板，从而限制浏览器发送 XHR 请求的数量。本章学习了两种不同的模板预加载技术：使用 `<script>` 指令和使用 `$templateCache` 服务。

AngularJS 是一个在浏览器中创建动态页面的框架，因此它严重依赖于 JavaScript。客户端生成 HTML 只能在所有所需 JavaScript 库文件加载完成且准备就绪之后进行。这种模式的不良作用就是用户可能会在页面加载和处理的过程中看到一个半成品的显示效果。为了去除这种不好的 UI 效果，AngularJS 提供了两个指令：`ng-cloak` 和 `ng-bind`。`ng-cloak` 指令可以隐藏大块的 DOM 树，针对那些高度动态没有多少静态内容显示的首页来说非常适用。与此相反，`ng-bind` 指令则更适用于包含很多静态内容的页面。

在优化首页的用户体验时，还应该考虑各种页面元素的加载顺序。将 JavaScript 文件的引用放在页面最底部通常被认为是最佳实践，但这种做法针对高度动态的由 JavaScript 驱动的 web 应用则没有太大实用价值。如果首页大部分是静态的，包含大量的 HTML 代码，那么将 `<script>` 标签放在页面最底部是一种好办法。对于那些高度动态的页面，在 `<head>` 标签中加载脚本往往效果更好，尤其是配合使用 `ng-cloak` 指令。

本章我们简要讨论了有关异步脚本加载器的问题。如今 AMD 模式大行其道，但它的价值在 AngularJS 应用中被削弱了。悲惨的现实是，截止写作本书时，AngularJS 无法很好地与异步脚本加载器配合使用，尤其针对加载应用程序代码更是无解。考虑到极其有限的好处和极其烦琐的配置，建议在 AngularJS 应用中不要使用 AMD 模块。

如果要将应用发布给公众，那么应该非常关心浏览器支持情况。令人欣慰的是，AngularJS 在所有主流浏览器中都能完美运行。不过 Internet Explorer 需要一些额外的设置，在本章的最后讲解了这些额外的设置。

索引

Index

Symbols

- \$interpolate service** 281
- \$anchorScroll**
 - about 165, 170
 - hashes 170
 - navigation 170
- \$anchorScrollProvider service** 170
- \$apply method** 295
- \$attrs** 257
- \$compile service**
 - transclusion function, creating 251
- \$dialog service** 199
- \$digest loop**
 - about 295-299, 304, 305, 310, 311
 - speeding up 303
 - turns, limiting 312
- \$element** 257
- \$error** 232
- \$exceptionHandler service** 298
- \$fieldErrors property** 269
- \$filter service** 133
- \$formatters pipeline** 232, 239
- \$get (factory) method** 33
- \$get property** 211
- \$http**
 - advanced features 104
 - and Promise API 94
 - code, testing 106, 107
 - REST adapters with 101-104
- \$http APIs** 76, 77
- \$httpBackend mock** 106, 107
- \$httpBackend service** 106
- \$http.jsonp function** 81
- \$http.post method** 78
- \$http.put method** 78
- \$http request** 204
- \$http service**
 - about 75, 197, 205
 - advanced features 104
- \$index variable** 115, 116
- \$injector service** 318
- \$interpolateProvider** 110
- \$interpolate service**
 - using 268
- \$locale.id variable** 275
- \$locationChangeSuccess event** 20
- \$location.hash()** 170
- \$location.path() component** 174
- \$locationProvider service** 187
- \$location service**
 - about 165, 170, 186, 284
 - used, for handcrafting navigation 172
 - using 168, 169
- \$location service API** 169
- \$on method** 20
- \$parsers** 232
- \$parse service** 291
- \$provide service** 286
- \$q.all method** 91, 93
- \$q.defer() method** 86
- \$q integration**
 - in AngularJS 93
- \$q service**
 - about 85
 - basics 85, 86
 - with Promise API 84
- \$q.when method** 92
- \$resource service**
 - about 95, 96
 - limitations 101
- \$rootScope.\$digest() method** 86

- \$routeParams service** 178
- \$routeProvider service provider** 176, 209
- \$route service**
 - about 175
 - limitations 182-185
 - multiple UI rectangles, handling
 - with ng-include 183, 184
- \$scope instance** 27
- \$scope object** 14
- \$setValidity(validationErrorKey, isValid)** 232
- \$templateCache content** 325
- \$templateCache service**
 - about 193, 194, 324, 327
 - filling 325, 326
- \$timeout service** 310
- \$timer service** 310
- \$transclude** 253, 257
- \$valid** 232
- \$watch** 293, 296
- \$watch functions** 227
- \$watch method** 292, 313
- \$watch property** 229
- <script> directives**
 - using, to preload templates 324, 325

A

- Accept-Language request header** 282
- accordion**
 - directive controller, using 262, 263
 - directive, implementing 263
- accordion directive suite**
 - creating 261
- accordion-group directive**
 - implementing 263-265
- action chaining**
 - asynchronous 89-91
- addGroup() function** 265
- alert directive**
 - creating 246, 247
- AMD**
 - and AngularJS 331, 332
- angular.callbacks._k function** 81
- AngularJS**
 - \$digest loop 295-299

- \$q integration 93
- \$watch 296
 - about 7, 8, 165, 318
 - and AMD 331
 - and application scripts 330
 - and jQuery 39, 40
 - applications, performance tuning 303
 - community 9
 - crash course 10
 - DOM, synchronizing 292, 293
 - example 10, 11
 - inner workings 290
 - libraries and extensions 9
 - model changes, propagating to DOM 291
 - model, stability 296-298
 - models, unstable 298
 - model, synchronizing 292, 293
 - model, updating 291
 - modules 26, 27
 - MVC pattern 12
 - Online learning resources 9
 - project 8
 - Scope.\$apply 293, 294, 295
 - scopes, hierarchy 299
 - tools 9

AngularJS directives

- about 214
- built-in directives 214
- compilation life-cycle 215, 216
- URL 215
- using, in HTML mark up 215

AngularJS expressions

- HTML content 111, 112
- HTML content, securing 195
- HTML, sanitizing 196
- unsafe HTML bindings, allowing 196

AngularJS forms

- about 139, 140
- validating 153

AngularJS objects

- testing 65

AngularJS seed project

- URL 56

AngularJS templates

- translated strings, handling 277

AngularJS widget directives

- about 222
- HTML template, using 224
- pagination directive, tests writing 223
- pagination directive, writing 222

angular.module function 26**application**

- sample application 44, 45
- securing 191

array data sources

- using 146

array-style DI annotations

- pitfalls 322, 323

asynchronous model validator

- creating 235
- implementing 238, 239
- tests, writing 237
- users service, mocking 236

attacks

- preventing 194

attribute (@)

- interpolating 226, 227

attribute (=)

- data binding to 227

authorization 210**authorization service**

- creating 210, 211

automated testing 43, 62, 63**B****Batarang 10, 302****beHungry method 85****bind-html-unsafe directive 112****Birds eye view**

- about 13
- controller 14
- model 14
- scope 13

Bootstrap CSS

- URL 48

build system

- about 48
- principles 49, 50

built-in directives, AngularJS

- directives 214

button directive

- writing 220-222

C**cache property 77****call-back expression**

- providing, in attribute (&) 227, 228

callback function 79**callbacks**

- aggregation 88
- registration 88

cancelLogin(redirectTo) method 199**cancel() method 207****canSave() method 156****checkbox inputs**

- using 144

clickable links

- creating 186

clicked() expression 120**click events 120****client-side authentication**

- supporting 203

client-side authorization

- failures, handling 203
- responses, intercepting 204
- securityInterceptor service, creating 205, 206
- SecurityRetryQueue service, creating 207
- supporting 203

client-side security support

- adding 198, 199
- login form, showing 200, 201
- menu items, hiding 202
- security-aware menus, creating 201
- security service, creating 199
- toolbars, creating 201

closeLoginDialog() helper 200**compilation process**

- about 265
- field directive, creating 265-267
- field template, setting up 270
- templates, loading dynamically 269
- terminal property in directives, using 267
- validation messages, binding to 269

compilation stage, directives 215, 216**compile field 219****compile function**

- transclusion, getting 252, 253

comprehension expression 146

configuration object primer

- about 77
- cache property 77
- headers property 77
- method property 77
- params property 77
- timeout property 77
- transformRequest property 77
- transformResponse property 77
- url property 77

configuration phase, modules 33**constructor-level**

- and instance-level methods 97, 98
- behavior, adding to resource objects 99, 100
- custom methods 98, 99

constructor-level

- and instance-level methods 97

content delivery network (CDN) 11**continuous integration (CI) server 333****controller based pagination directive**

- creating 258

controller field 219**controller property 256****controllers**

- about 56
- testing 67-70

cookie snooping

- preventing 194, 195

CORS 81-83**CPU utilization**

- \$digest loops, speeding up 303
- DOM access in watch-expression, avoiding 305, 306
- optimizing 303
- unnecessary watches, removing 306
- unused watches, removing 309
- watch expression, syntax 303-305

Cross-origin resource sharing. *See* CORS**cross-site request forgery**

- preventing 198

cross-site scripting attacks

- HTML content in AngularJS expressions, securing 195
- HTML, sanitizing 196
- preventing 195
- unsafe HTML bindings, allowing 196

currency filter 276**currentUser property 199****custom validation directive**

- creating 230, 231
- directive controller 231
- implementing 235
- tests, writing 233, 234

D**Daily workflow 71****data conversion 79****dataSource expression 146, 147****data transformations**

- in filters 136

date filter 275, 286**datepicker input directive 239****debugging 74****declarative template view 22, 24, 25****deep-watching mode 313****deferred object 207****dependency injection (DI)**

- about 8, 27, 28, 259, 318
- benefits 28, 29

describe function 64**details**

- multiple rows, displaying 118, 119
- one row, displaying 117, 118

DI annotations

- pitfalls 322

directive() 218**directive controllers**

- about 256, 257
- accordion directive, implementing 263
- accordion directive suite, creating 261
- accordion-group directive, implementing 263, 264
- and link functions 258
- compilation process 259, 260
- controller based pagination directive, creating 258
- dependency injection 259
- in accordion, using 262, 263
- optional, creating 231
- other controllers, access to 260
- parents, searching 232
- priority property, using 256
- requiring 231

- special dependencies, injecting 257
- transclusion function, access to 261
- transclusion, getting 253
- directives**
 - attribute (&), call-back expression providing 227
 - attribute (=), data binding to 227
 - attribute (@), interpolating 226
 - button directive, writing 220, 221
 - button, styling 219
 - compile field 219
 - controller field 219
 - defining 218, 219, 321, 322
 - isolating, from parent
 - scope 225, 226
 - link field 219
 - name field 219
 - pagination directive definition object 228, 229
 - pagination directive, tests writing 223
 - pagination directive, writing 222
 - priority field 219
 - priority property, using 255
 - referencing 109, 110
 - replace field 219
 - require field 219
 - restrict field 219
 - scope field 219
 - selectPage call-back, adding 229, 230
 - skeleton unit test 217
 - template field 219
 - templateUrl field 219
 - terminal field 219
 - transclude field 219
 - transclusion functions, accessing 252
 - transclusion, using 245, 246
 - unit tests, writing 217, 218
 - using 279
 - using, in HTML mark up 215
- disableAutoScrolling() method** 170
- DOM**
 - access in watch-expression,
 - avoiding 305, 306
 - synchronizing 292
- domain-specific language (DSL)** 22, 71
- DOM-based templates**
 - about 121

- HTML elements, custom 124
- multiple DOM elements 122
- Repeater DOM elements 122
- verbose syntax 121
- DOM Event handlers** 120, 121
- DOM events** 310

E

- empty options**
 - using, with select directive 148
- End to end tests**
 - about 70
 - daily workflow 71
 - debugging 73, 74
 - Karma runner tips and tricks 72
 - tests subset, executing 73
- errorCallback method** 86
- Express**
 - URL 47

F

- factory method** 31
- field directive** 266
- field template**
 - setting up 270
- file naming conventions**
 - test folder 57
- files**
 - about 58
 - AngularJS specific files 54, 55
- filters**
 - about 125-127
 - accessing, from JavaScript code 133, 134
 - and DOM manipulation 135
 - array-transforming filters 125
 - built-in filters 125
 - custom filters 131, 132
 - data transformations, costly 136
 - dos and donts 134
 - filtered results, counting 128, 129
 - formatting 125
 - model transformations, handling 124
 - orderBy filter, sorting with 129-131
 - unstable 136, 137
- first-class JavaScript objects** 87
- flush() method** 107

folders

- root folders 52, 53
- sources folder 54
- test folder 57

fullDate format 285, 286**fullDate** string 285**function** arguments 320**function** method 319**G****getCssClasses()** method 156**GET**/current-user message 193**getLoginReason()** method 199**getName()** expression 297**getNameLog()** expression 308**GitHub**

- URL 8, 44

Google + community

- URL 9

Grunt.js 51**GUI architectures**

- URL 12

H**handcrafting navigation**

- controllers in route partials, defining 174
- controlling, \$location service used 172
- missing bits 175
- pages, structuring around routes 173
- routes, mapping to URLs 174

headers property 77**href** attribute 186, 284**HTML**

- sanitizing 196

HTML5

- and history API 167, 168
- working with 186

HTML5 mode

- client side 171
- configuring, for URLs 171
- server side 171, 172

HTML content

- in AngularJS expressions 111, 112

HTML form submission

- events 161
- handling 161

handling, **ngClick** used 162handling, **ngSubmit** used 162

to server 161

HTML hidden input fields 150, 151**HTML mark up**

- directives, using 215

HTML template

- using, in directives 224

HTTP CORS

- URL 82

HTTP responses

- dealing with 79

https module 195**I****i18n** filter 278**IDE** extensions 10**IE.** *See* Internet Explorer**if** directive

- creating 253, 254

input change event 120**input** directives

- about 143
- checkbox inputs, using 144
- radio inputs, using 145
- required validation, adding 143
- select inputs, using 145
- simple string options, providing 145
- text-based inputs, using 143, 144

Internet Explorer 333, 334**interpolation** directive 110**it** function 64**J****Jasmine**

- about 51
- test, anatomy 64

JavaScript code

- minification-safe, writing 319, 320
- translated strings, handling 280, 281

JavaScript timers 310**jQuery**

- and AngularJS 39, 40

jQueryUI Date Picker

- directive, implementing 242, 243
- tests, writing 240-242

- wrapping 239, 240
- jsFiddle 10
- jshint 305
- jslint
 - URL 48
- JSON API
 - URL 334
- JSON_CALLBACK request parameter 81
- JSON injection vulnerability
 - preventing 197
- JSONP
 - limitations 81
 - same-origin policy restrictions,
 - overcoming 80, 81
- JSONP requests
 - with \$http 75

K

- Karma runner 52
- keyboard events 120

L

- landing page
 - DOM portion hiding, ng-cloak
 - used 328, 329
 - individual expressions, hiding with ng-bind
 - 329, 330
 - optimizing 327
 - templates in unprocessed form, display
 - avoiding 328
- limitTo filter 125
- link field 219
- links
 - handling 185
- loadTemplate function 269
- locales
 - including, as URLs part 283, 284
 - switching 284, 285
- locale-specific modules
 - and AngularJS filters 275
 - configuring 274
 - settings, using 274
 - symbols, using 274
- login(email, password) method 199
- login toolbar
 - creating 202

- login-toolbar widget directive 202
- logout(redirectTo) method 199
- logPosition function 121

M

- man-in-the-middle attack
 - URL 194
- memory consumption
 - optimizing 312
- method property 77
- model
 - changes, propagating to DOM 291
 - DOM events response, updating for 291
 - stability 296, 297
 - synchronizing 292
 - unstable 298
- modelChangeCallback 296
- model transformations
 - handling, with filters 124
- model values
 - rendering, with ngBind 111
- module function 66
- modules
 - about 26, 27, 59, 320
 - advantages 38
 - blocks, configuring 61
 - blocks, declaring 61
 - blocks, running 61
 - depending on other modules 35
 - lifecycle 33
 - providers registering, syntax for 59, 60
 - services, visibility 35, 36, 37
- modules, lifecycle
 - about 33
 - configuration phase 33
 - registration, methods 34
 - run phase 34
- MongoDB
 - collections 76
- MongoLab
 - about 46
 - URL 76
- mouse events 120
- multiple options
 - selecting 150

Mustache

URL 290

MVC pattern

about 12

Birds eye view 13

scopes 15

view 21, 22

N**name field** 219**native browser validation**

disabling 157

navigation events 310**network events** 310**network-related performance**

improving 318

ngBind

model values, rendering with 111

ng-bind directive

about 111, 195

individual expressions,

handling with 329, 330

ng-bind-html directive 112, 196, 197**ng-bind-html-unsafe directive** 112, 196**ngClass** 119**ng-class directive** 156**ngClassEven directive** 119**ngClassOdd directive** 119**ngClick**

using, to handle form submission 162

ng-click directive 186**ng-cloak directive** 329**ng-controller directive** 13, 15, 178**ngFormController**

about 153

name attribute, using 154

ng-hide family 113**ng-include directive** 114, 174**ng-init attribute** 13**ngLocale module** 275, 282, 284**ngModel** 110**ngModelController**

\$error 232

\$formatters 232

\$parsers 232

\$setValidity(validationErrorKey, isValid)
232

\$valid 232

about 151, 232

input field validity, tracking 153

transformation pipeline 152

ng-model directive 291, 294**ngModel directive** 141**ngOptions directive**

array data sources, using 146, 147

dynamic options, providing 146

examples 146

object data sources, using 147

ng-repeat directive 116

about 114-116, 222, 256, 314, 315

bindings, simplifying 315

collection, watching 314

ngRepeat directive 115**ngRepeat patterns** 117**ngSanitize module** 112, 196, 197**ng-show family** 113**ngSubmit**

using, to handle form submission 162

ng-switch directive 113**ng-transclude directive** 248**non-GET request** 192**NotificationsService service** 30**novalidate attribute** 157**novalidateForm attribute** 157**number filter** 276**O****OAuth2**

URL 193

object data sources

using 147

Object.observe 41**objects**

collaborating 27

onLoginDialogClose() method 201**onSelectPage() function** 230**openLoginDialog() helper** 200, 201**optionBinding expression** 146, 148**orderBy filter** 125, 129

sorting with 129-131

P**page**

- about 225
- hashes 170
- navigation 170
- structuring, around routes 173

pagination directive

- tests, writing 223
- writing 222

pagination filter 132**params argument 97****params property 77****partial templates**

- securing 193, 194

Passport

- URL 47

performance tuning 301, 302**plugins 10****Plunker 10****POST /login message 192****POST /logout message 193****priority field 219****Promise API**

- about 84
- and \$http 94
- with \$q 84

promise object 86, 204**promises**

- lifecycle 88

providers 321

- registering, syntax for 59, 60

pushRetryFn() method 207**Q****Q Promise API library**

- URL 84

query() function 236**query() method 236****R****radio inputs**

- using 145

reject method 86**remaining() function 23****removeTeamMember method 68****replace field 219****replace property**

- directive definition 247

Representational State Transfer. *See* REST**request data conversion 78****requireAdminUser() method 210, 211****requireAuthenticatedUser() method 210****require field 219****resolve function 209****resolve method 86, 210****resolve property 180, 188****response interceptors 204****REST 95****REST adapters**

- with \$http 101

REST API 76**Restler**

- URL 47

restrict field 219**retry() method 207****root folders 53****route definitions**

- code duplication 188, 189
- organizing 187
- spreading, among modules 188

routes

- mapping, to URLs 174

routes object 173**routes variable 173****routing services**

- about 175
- changes, preventing 181, 182
- clickable links, creating 186
- default routes, defining 178
- defining 176
- definitions, organizing 187
- definitions, spreading 188
- external pages, linking to 187
- flexible routes, matching 177
- limitations 182
- links, handling 185
- matched route content, displaying 176
- parameter values, accessing 178
- partials, reusing with different controllers 178, 179
- patterns 185
- tips 185

- tricks 185
- UI flickering, avoiding 179-181
- run phase, modules** 34

S

- same-origin security policy**
 - restrictions 79, 80
 - restrictions, with CORS 81-83
 - restrictions, with JSONP 80, 81
- Scenario Runner** 70, 71
- Scope.\$apply** 293, 294
- scope.\$apply()** method
 - unused watches, removing 310
- Scope.\$digest**
 - calling, instead of Scope.\$apply 308, 309
- scope.\$digest method** 308
- scope field** 219
- scopes**
 - about 15
 - and evening system 20
 - hierarchy 15, 16, 20
 - inheritance 17
 - inheritance, through scopes
 - hierarchy 17-19
 - lifecycle 21
- scripts**
 - referencing 330
- SCRUM** 44
- secure routes**
 - access, preventing 208
 - resolve functions, using 209
- securityInterceptor service**
 - creating 205, 206
- SecurityRetryQueue service**
 - about 206, 210
 - creating 207
- security service**
 - about 199, 200
 - notifying 208
- select directive**
 - about 149
 - empty options, using 148
- select inputs**
 - using 145
- selectPage call-back**
 - adding, to directive 229, 230

- selectPage() function** 230
- server-side authentication**
 - API, providing 192
 - providing 192
- server-side authorization**
 - providing 192
 - unauthorized access, handling 192
- server-side environment** 47
- server-side proxies** 83
- services**
 - testing 65, 66
- services, registering**
 - about 29, 30
 - constants 31, 32
 - factory method 31
 - providers 32, 33
- setTimeout function** 310
- showLogin() method** 199
- simple-bind directive** 292
- simple-model directive** 291, 293
- single-page web applications**
 - URLs 166
- sortField property** 130
- sources folder** 54
- special variables** 115, 116
- static resources**
 - minifying 318
- stopPropagation() method** 20
- subforms**
 - repeated inputs, validating 159-161
 - repeating 158, 159
 - using, as reusable components 157, 158
- switchLocale function** 285
- switchLocaleUrl function** 285

T

- table rows classes**
 - altering 119
- template field** 219
- templates module** 325
- templates, preloading**
 - \$templateCache service, filling in 325-327
 - <script> directives used 324
 - about 323
 - techniques, combining 327

- templateUrl field** 219
- templateUrl property** 183, 323
- terminal field** 219
- terminal property**
 - in directives, using 267
- Test Driven Development (TDD)** 218
- tests**
 - subset, executing 73
- Third-party JavaScript libraries** 48
- timeout property** 77
- tools, AngularJS**
 - about 9
 - Batarang 10
 - IDE extensions and plugins 10
 - Plunker and jsFiddle 10
- transclude field** 219
- transcludeFn** 252, 253
- transclusion**
 - alert directive, creating 246, 247
 - if directive, creating 253-255
 - in isolated scope directive 246
 - property, in directive definition 248
 - using 245
 - using, in directives 245, 246
- transclusion function**
 - about 250
 - creating, \$compile service 251
 - in compile function 252, 253
 - in directive controller 253
 - in directives, accessing 252
- transclusion scope** 248, 250
- translated strings**
 - used in JavaScript code, handling 280, 281
- translations**
 - handling 276, 277
 - partials translating, build-time
 - used 279, 280
 - translated strings handling, Angular JS
 - templates used 277
 - translated strings handling,
 - directives used 279
 - translated strings handling, filters used 278
- two-way data binding** 12

U

- UI** 307
- Uniform resource locators.** *See* **URLs**
- unit tests**
 - about 63
 - AngularJS objects, testing 65
 - asynchronous code testing 68, 70
 - controllers, testing 67, 68
 - Jasmine test, anatomy 64
 - services, testing 65, 66
 - writing, for directives 217, 218
- update() function** 311
- updateModel() function** 243
- url property** 77
- URLs**
 - about 165, 169
 - HTML5 mode, configuring 171
 - in pre-HTML5 era 166, 167
 - in single-page web applications 166
 - routes, mapping 174
- user-experience (UX)** 22
- User Info form**
 - dynamic behavior, adding 154
 - native browser validation, disabling 157
 - resetting 162, 163
 - save button, disabling 156
 - validation errors, displaying 155, 156
- User Information Form**
 - creating 142, 143
- user.password model field** 144
- User.query() method** 96
- user.role model** 144
- Users.query() function** 237
- Users.respondWith() function** 236

V

- validateEqual(value)** 235
- validation messages**
 - displaying 269
- verifyNoOutstandingExpectation method** 107
- view** 22

W**watched expressions**

- size 314

watches

- deep watching, avoiding 312, 313
- number, evaluating 306
- unnecessary watches, removing 306
- unused watches, removing 309

watchExpression 292-303**web application** 289**widget**

- implementing 228, 229

worldsPercentage function 17**X****XHRrequests**

- types 77

XMLHttpRequest (XHR) 75**XSRF.** *See* cross-site request forgery**XSRF-TOKEN** 198**XSS.** *See* cross-site request forgery**X-XSRF-TOKEN** 198

精通 AngularJS

AngularJS诞生于Google，已用于开发多款Google产品。它是一套JavaScript前端框架，用于开发当下流行的数据驱动的单页面Web应用。其核心特性是：MVC、模块化、自动双向数据绑定、语义化标签、依赖注入等。

本书深入浅出地讲解了AngularJS的开发概念和原理，并通过丰富的开发实例向读者展示了构建复杂应用的完整过程，包括学习使用AngularJS特有的基于DOM的模板系统，实现复杂的后端通信，创建漂亮的表单，制作导航，使用依赖注入系统，提高Web应用的安全性，使用Jasmine开展单元测试，等等。

读者对象

前端开发工程师、JavaScript程序员、网页设计师、交互设计师

本书能帮助你

- 学习使用AngularJS特有的基于DOM的模板系统，了解其精妙之处。
- 通过各种后端查询和修改数据，熟练掌握Promise API。
- 充分利用双向数据绑定，快速创建复杂的表单。
- 借助HTML5的History API为Web应用添加导航。
- 借助依赖注入系统管理AngularJS模块间的依赖关系。
- 提高Web应用安全性。
- 使用Jasmine的行为驱动测试框架开展单元测试。
- 学习如何使用AngularJS编译器。

策划编辑：李丹立
责任编辑：陈元玉

ISBN 978-7-5681



9 787568 003964 >

定价：79.00元

上架建议：Web开发 / 前端开发

[PACKT]
PUBLISHING

社区实践精华

PDF电子书说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！