AngularJS

开发结构化的Web应用：
代码更少、乐趣更多、生产力更高

用AngularJS开发下一代Web应用

Brad Green & Shyam Seshadri 著
大漠穷秋 译

O'REILLY®

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

内 容 简 介

AngularJS是一款来自Google的前端JS框架，它的核心特性有：MVC、双向数据绑定、指令和语义化标签、模块化工具、依赖注入、HTML模板，以及对常用工具的封装，例如$http、$cookies、$location等。AngularJS框架的体积非常小，但是设计理念和功能却非常强大，值得前端开发者深入学习。

本书对AngularJS框架的核心特性做了全面的介绍，包括常用的开发工具和开发环境。作为国内第一本关于AngularJS的书籍，本书是学习AngularJS的必备入门工具。

版权贸易合同登记号　图字：01-2013-6328

图书在版编目（CIP）数据

用 AngularJS 开发下一代 Web 应用 /（美）格林（Green,B.），

（美）夏德瑞（Seshadri,S.）著；大漠穷秋译. —北京：电子工业出版社，2013.10

ISBN 978-7-121-21574-2

Ⅰ. ①用… Ⅱ. ①格… ②夏… ③大… Ⅲ. ① JAVA 语言—程序设计 IV. ① TP312

中国版本图书馆 CIP 数据核字（2013）第 230425 号

策划编辑：张春雨
责任编辑：徐津平
封面设计：Randy Comer　张健
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：12.75　　字数：280千字
印　　次：2013年10月第1次印刷
定　　价：55.00元

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

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

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

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

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

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

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

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

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

——Linux Journal

译者序

本书是国内第一本关于 AngularJS 框架的书籍。

AngularJS 是一款来自 Google 的前端 JS 框架，该框架已经被应用到了 Google 的多款产品中。这款框架最核心特性有：MVC、模块化、自动化双向数据绑定、语义化标签、依赖注入，等等。

目前，web 应用的规模和复杂度不断提升，各种框架层出不穷，然而从实际的使用效果来看，我们所做的努力依然不够。与各种服务端框架相比，前端框架在设计理念方面仍然存在很大的提升空间。

例如，很多服务端框架都有“依赖注入”的概念，但目前市面上很少有人会把这一概念应用到前端框架中，而 AngularJS 就是第一个吃螃蟹的。

又如，AngularJS 框架自身是通过 TDD（测试驱动）的方式来开发的，从这个角度来看，AngularJS 是敏捷开发的一次成功实践。

再例如，使用模板和控制器的概念，AngularJS 对 DOM 操作进行了彻底的封装，因此，对于业务代码来讲，根本不需要再去关心原始的 DOM 操作，例如设置 CSS 样式、注册事件监听器等。

这种设计还带来了另外一个好处，那就是让单元测试和集成测试成为可能。大家都知道，一般来说，想对前端的 JavaScript 代码进行完善的单元测试是非常困难的，这里面最关键的一个问题就是，很多代码里面会涉及 DOM 操作，因此很多代码脱离浏览器环境是无法运行的，更不用说进行单元测试了！而 AngularJS 就很好地解决了这一问题。

类似这种理念性的变革，在 AngularJS 中比比皆是。正如原书作者所言，AngularJS 并没有发明这些概念，只是把现有的概念应用到了前端框架中。AngularJS 不是一个大而全

的框架，但是它所提出的很多探索性的理念值得所有专注前端的开发者悉心研究。

最后，正如大多数 Google 的产品一样，AngularJS 框架是完全免费开源的，这对于国内的很多开发者来说也是需要首先考虑的优势。

作为一个前端开发的老鸟，译者在翻译本书的过程中也体验到了 Google 牛人们思想上的高瞻远瞩。所以，这里要首先感谢张春雨编辑当初给我推荐了这款框架。同时也必须感谢电子工业出版社各位编辑的辛苦劳动，让本书的中文版得以面世。另外，本书能够成为 AngularJS 框架的第一本中文书籍，对此译者也感到相当自豪，也算是为国内软件工业水平的提升尽了绵薄之力。

当然，由于译者本身的阅历和水平所限，难免存在一些疏漏和错误，欢迎读者不吝指正。

本书所涉及的各种引用资源都可以在 github 的主页上找到：*https://github.com/angular/*。

大漠穷秋

2013 年 6 月 27 日于南京

前言

关于 Angular 的起源，我可以追溯到 2009 年的 Google Feedback 项目。当时，对于项目的开发速度以及如何编写可测试代码的问题，我们已经经受了几个月的折磨。6 个月时，我们开发了差不多 17000 行前端代码。这时候，团队中的一个成员 Misko Hevery 做出了一个大胆的宣言：利用他自己业余时间所开发的一个开源库，他可以在两周之内把目前所有东西重写一遍。

我当时想，两周的时间并不会给我们造成太大的影响，同时我们也接受了 Misko 努力构建一些东西的想法。然而 Misko 最终还是估算错了时间，他用了三个星期。但是，我们所有人还是被他深深地震撼了，更让我们感到震撼的是，他所开发的新应用的代码量从原来的 17000 行压缩到了 1500 行。看起来，Misko 的东西值得深入推广。

Misko 和我决定，围绕他所提倡的理念组建一个团队，这个简单的理念就是：简化对 web 开发者的经验要求。Shyam Seshadri，也就是本书的合著者，后来继续领导 Google Feedback 团队开发了第一款搭载 Angular 的应用。

从那时起，我们在大家的指导下继续开发 Angular。给予我们指导的人有的来自 Google 自己的团队，也有来自全球的数以百计的开源贡献者。数千名开发者在他们的日常工作中依赖 Angular，并且发展成了一个优质的支持者网络。

我们也非常期望能够接受你的指导。

排版约定

在本书中将会使用以下排版和印刷约定。

斜体（*Italic*）
表示新的术语、URL、邮箱地址、文件名以及文件的扩展名。

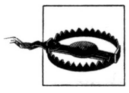
等宽字体（Constant width）
用于代码清单，以及段落中所引用的编程元素，例如变量、函数名称、数据库、数据类型、环境变量名称、语句以及关键字。

等宽加粗字体（**Constant width bold**）
用于显示需要用户输入的文本字面值，例如命令或者文本。

等宽斜体（*Constant width italic*）
用于显示应该被替换的文本，这些文本会被用户所输入的值或根据上下文所决定的值所替换。

书中切口处的“ ”表示原书页码。

这个标记表示一个小提示、建议，或者一般性的注意点。



这个标记表示一个警告或者注意点。

使用实例代码

本书的目标是帮助你完成工作。总地来说，你完全可以将本书中所包含的实例代码用在你的程序和文档中。你没有必要联系我们来获得授权，除非你想对代码做出大规模的重构。举例来说，如果你编写一个程序，里面将会使用几段来自本书中的代码，这种行为并不需要获得授权；而出售或者分发 O’Reilly 书籍中实例代码的 CD-ROM 就需要获得授权；引用本书中的内容或者本书中的实例代码来回答问题不需要授权；而把本书中的大量代码合并到你的产品文档中就需要授权。

我们非常感激，但是并不强制你注明引用内容的出处。注明出处一般包含标题、作者、出版商及 ISBN 信息。例如：“*AngularJS* by Brad Green and Shyam Seshadri (O’Reilly). Copyright 2013 Brad Green and Shyam Seshadri, 978-1-449-34485-6.”

如果你感觉你使用实例代码的方式不属于以上所述的任何方式，可以随时与我们联系，邮箱地址为 *permissions@oreilly.com*。

Safari® Books Online

Safari Books Online 是一个即时的数字图书馆，它会随时以书籍或者视频

的形式发布专业级的技术和商业文章，这些内容都出自全球顶级作者的手笔。

大量的技术专家、软件开发者、web 设计师，以及商业和创意设计师都在使用 Safari Books Online，他们把它作为研究、解决日常问题、学习以及认证培训的首要资源。

Safari Books Online 为各种组织、政府机构及个人提供了一组产品包和计费系统。订阅者可以访问成千上万种书籍、培训视频，以及正式出版的手稿，这些内容来自以下出版社的全文检索数据库：O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。关于 Safari Books Online 的更多信息，请访问我们的在线站点。

联系我们

关于本书的建议和疑问，可以与下面的出版社联系。

美国：

O'Reilly Media，Inc.
1005 Gravenstein Highway North
Sebastopol，CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们将关于本书的勘误表，例子以及其他信息列在本书的网页上，网页地址是：

http://oreilly/angularJS

如果要评论本书或者咨询关于本书的技术问题，请发邮件到：

bookquestions@oreilly.com

想了解关于 O'Reilly 图书、课程、会议和新闻的更多信息，请访问以下网站：

http://www.oreilly.com.cn

http://www.oreilly.com

致谢

我想对 Misko Hevery 致以特别的感谢，他是 Angular 之父，感谢他以非凡的勇气对 web 应用的编写方式提出了不同的想法并把这些想法付诸实现。感谢 Igor Minar，感谢他为 Angular 项目带来的稳定性和结构化，同时还为形成今天这样一个优质的开源社区奠定了基础。感谢 Vojta Jina，他为 Angular 编写了大量的模块，同时还给我们带来了全世界最快的测试引擎。感谢 Naomi Black、John Lindquist 以及 Mathias Matias Niemelä，感谢他们所做的编辑工作。最后，感谢 Angular 社区做出的贡献，感谢他们的反馈，这些反馈告诉了我们如何让 Angular 在构建真实的应用方面做得更好。

第1章

1

AngularJS简介

目前，在创建优质的 web 应用方面，我们已经具备了令人难以置信的能力。但是，在创建这些应用的过程中所引入的复杂性同样令人难以置信。我们 Angular 团队致力于减轻开发人员在开发 AJAX 应用过程中的痛苦。在 Google，我们已经通过艰苦的努力构建了很多大型的 web 应用，例如 Gmail、Maps、Calender 等。我们认为，这些经验可以让每个人获益。

在编写 web 应用的过程中，我们都希望一边编写代码，一边就能够惊奇地看到代码所产生的效果。我们希望编码的过程就像是创作，而不是为了满足 web 浏览器内部的一些奇怪的运行机制。

我们还想要一个开发环境，它可以帮助我们在设计方面做出抉择，让应用从一开始就易于创建和理解；同时，当应用变得很大的时候，还能够持续地做出正确的抉择，让我们的应用易于测试、扩展和维护。

在 AngularJS 框架中，我们已经成功实现了以上目标。我们对所取得的成就感到非常兴奋。这里面很大一部分功劳要归于 Angular 周边的开源社区，它们互相支持，做得非常出色，同时也教会我们很多东西。希望你也能加入我们的社区，把 Angular 发展得更好。

在 GitHub 库中有一些大型的、复杂的例子，以及一些代码片段，你可以查看这些例子、创建分支，或者在我们的 GitHub 页面 *https://github.com/shyamseshadri/angularjs-book* 上尝试运行它们。

一些概念

你将会用到一些核心的概念，这些概念将会贯穿整个应用。并非我们发明了这些概念，而只是从其他开发环境中大量地借用了这些比较成功的习惯用语，然后以适合 HTML、浏览器及其他常见的 web 标准的方式来实现了这些概念。

2 客户端模板

多页面 web 应用会在服务端创建 HTML，把 HTML 和数据装配并混合起来，然后再把生成的页面发送到浏览器中。在某种程度上，大部分单页面应用——也叫做 AJAX 应用——也会做同样的事情。在这一方面，Angular 的处理方式完全不同，在 Angular 中，模板和数据都会发送到浏览器中，然后在客户端进行装配。这样一来，服务器的角色就变成了仅仅为这些模板提供一些静态的资源，然后为这些模板提供所需要的正确数据。

我们来看一个例子，看看在 Angular 中浏览器是如何把数据和模板装配起来的。我们会例行公事地写一个“Hello, World”示例，但是不会把“Hello, World”作为一行完整的字符串打印出来，而会把打招呼用的单词“Hello”构造成一个数据，然后再在后面修改这个单词。

为了实现这一功能，我们把模板创建在 *hello.html* 中：

```
<html ng-app>
<head>
  <script src="angular.js"></script>
  <script src="controllers.js"></script>
</head>
<body>
  <div ng-controller='HelloController'>
    <p>{{greeting.text}}, World</p>
  </div>
</body>
</html>
```

同时我们的应用逻辑写在 *controllers.js* 中：

```
function HelloController($scope) {
  $scope.greeting = { text: 'Hello' };
}
```

使用任意浏览器打开 *hello.html*，然后我们就可以看到如图 1-1 所示的效果：

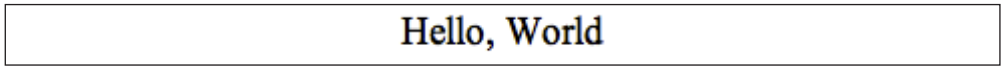


图1-1：Hello, World

与目前广泛使用的一些方法相比，这里有一些非常有趣的东西需要注意：

- HTML 里面没有 class 或者 ID 来标记在哪里添加事件监听器。
- 当 `HelloController` 把 `greeting.text` 设置成单词 *Hello* 的时候，我们没有必要注册任何事件监听器或者编写任何回调函数。
- `HelloController` 只是一个普通的 JavaScript 类，并且不需要继承 Angular 所提供的任何东西。
- `HelloController` 可以获取到它所需要的 `$scope` 对象，而没有必要去创建它。
- 没有必要自己调用 `HelloController` 的构造方法，或者指定何时去调用它。

后面我们还会看到更多不同之处，但是这里已经能够清晰地显示出 Angular 应用与以前类似的应用在结构上存在巨大的差异。

为什么我们会在设计上做出这种选择，Angular 的工作原理又是什么呢？我们来看看 Angular 从其他地方“偷来”的一些很好的理念。

Model View Controller（MVC）

MVC 程序结构是在 19 世纪 70 年代作为 Smalltalk 语言的一部分被引入的。在 Smalltalk 的发展初期，MVC 在每一种桌面开发环境中都变得非常流行，这些环境中的用户界面都相当复杂。无论是 C++、Java 还是 Objective-C，它们都带有一些 MVC 的意味。但是，直到现在，MVC 还依然与 web 开发毫不相干。

MVC 背后的核心理念是：你应该把管理数据的代码（model）、应用逻辑代码（controller），以及向用户展示数据的代码（view）清晰地分离开。

视图会从模型中获取数据，然后展示给用户。当用户通过鼠标点击或者键盘输入与应用进行交互的时候，控制器将会做出响应并修改模型中的数据。最后，模型会通知视图数据已经发生了变更，这样视图就可以刷新其中显示的内容。

在 Angular 应用中，视图就是 Document Object Model（DOM，文档对象模型），控制器就是 JavaScript 类，而模型数据则被存储在对象的属性中。

MVC 之所以非常灵活，原因有很多。首先，对于什么东西应该放在哪里，MVC 给了你一个思想上的模型，所以你不需要每次都来构建这个模型。如果和你一起做项目的人知道你正在使用 MVC 来组织代码，那么他们立刻就能明白你写的东西是什么意思。最重

要的一点是，使用 MVC 模型会让你的应用更加易于扩展、维护和测试。

数据绑定

在 AJAX 型的单页应用普及之前，类似 Rails、PHP 和 JSP 之类的平台都可以帮助我们创建用户界面（UI），它们会把 HTML 字符串和数据混合起来，然后再发送给用户并显示。

4

而 jQuery 之类的库则在客户端继承了这一模型，让我们遵守类似的风格，但是使用 jQuery 可以单独刷新 DOM 中的局部内容，而不是刷新整个页面。在 jQuery 中，我们会把 HTML 模板字符串和数据混合起来，然后把获得的结果插入 DOM 中我们所期望的位置，插入的方式是把结果设置给一个占位符元素的 `innerHtml` 属性。

以上机制都工作得相当不错，但是当你想要把最新的数据插入到 UI 中，或者根据用户输入来修改数据的时候，你就需要做很多极其繁琐的工作来保证数据的状态是正确的，并且 UI 和 JavaScript 属性要同时正确。

但是，如果我们不需要编写代码就能做到以上所有事情会怎么样？如果我们可以仅仅声明 UI 中的某个部分需要映射到某个 JavaScript 属性，然后让它们自己去同步会怎么样？这种编程风格叫做数据绑定。因为它可以和 MVC 很好地结合起来，所以我们把它引入到了 Angular 中。这样一来，当你编写视图和模型的时候，可以节省代码量。在 UI 中，把数据从一个值修改成另一个值的大部分工作会自动进行。

为了在实战中看到这一点，我们来修改第一个例子，让它变成动态的。目前的情况是，`HelloController` 会给模型 `greeting.text` 赋一次值，之后再也不会修改它。为了让它变成动态的，我们来修改这个例子，增加一个文本输入框，它会把 `greeting.text` 的值修改成用户所输入的内容。

下面是新的模板：

```
<html ng-app>
<head>
  <script src="angular.js"></script>
  <script src="controllers.js"></script>
</head>
<body>
  <div ng-controller='HelloController'>
    <input ng-model='greeting.text'>
    <p>{{greeting.text}}, World</p>
  </div>
</body>
</html>
```

控制器 `HelloController` 保持原样不变。

把这个例子加载到浏览器中，效果如图 1-2 所示。

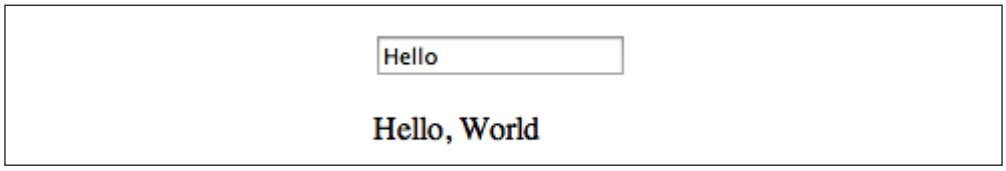


图1-2：greeting应用的默认状态

如果我们把输入框中的 *Hello* 换成 *Hi*，可以看到效果如图 1-3 所示。

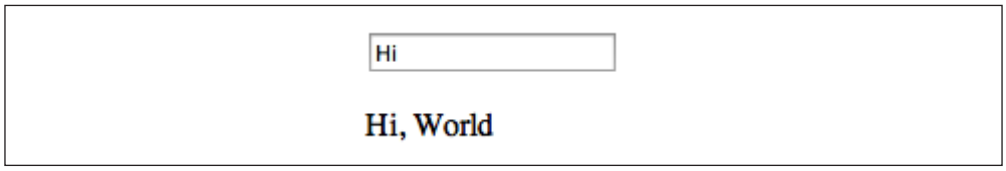


图1-3：输入发生改变之后的Greeting应用

我们没有在输入框上注册任何 change 事件监听器就可以让 UI 自动刷新了。这一机制对于来自服务器的更新同样有效，在我们的控制器中，可以向服务器发起一次请求，获取响应，然后把 `$scope.greeting.text` 设置为服务端所返回的内容。Angular 会自动把输入框和花括号中的文本更新为所获得的新值。

依赖注入

前面曾经提到过这些内容，但是请允许我再重复一下，对于 `HelloController` 还有很多东西不需要我们去编写。例如，进行数据绑定的 `$scope` 对象会被自动传递给我们；我们并不需要调用任何函数去创建这个对象。只要把 `$scope` 对象放在 `HelloController` 的构造函数里面，然后就可以获取它了。

正如后续章节中会看到的，`$scope` 对象并不是我们唯一可以获取的东西。如果我们想把数据绑定到用户浏览器中的 URL 地址上，可以使用 `$location` 对象，只要把 $location 对象放到我们的构造函数中即可，示例如下：

```
function HelloController($scope, $location) {
  $scope.greeting = { text: 'Hello' };
  // 这里可以使用 $location 对象来做一些很酷的事情 ...
}
```

这种神奇效果是通过 Angular 的依赖注入机制实现的。依赖注入让我们遵守这样一种开

发风格：我们的类只是简单地获取它们所需要的东西，而不需要创建那些它们所依赖的东西。

这种风格遵循了一种叫做迪米特法则[1]（Law of Demeter，*http://en.wikipedia.org/wiki/Law_of_Demeter*）的设计模式，也叫做最少知识原则。既然 `HelloController` 的职责是设置 greeting 模型的内部状态，那么用这一法则就意味着，它不应该去操心任何其他东西，例如 `$scope` 是如何创建的，以及到哪里找这个对象等。

这一特性并非仅仅对 Angular 框架所创建的对象有效，你可以以同样的方式编写所有其他代码。

指令

Angular 最强大的功能之一就是，你可以把模板编写成 HTML 的形式。之所以可以做到这一点，是因为我们引入了一款强大的 DOM 转换引擎，可以用它来扩展 HTML 的语法。

6

我们已经在模板里面看到了一些新的属性，这些属性不属于 HTML 规范。例如，我们引入了双花括号用来实现数据绑定；引入了 `ng-controller` 用来指定每个控制器负责监视视图中的哪一部分；引入了 `ng-model`，用来把输入数据绑定到模型中的一部分属性上。我们把这些叫做 HTML 扩展指令。

Angular 内置了很多指令，可以帮助你为你的应用定义视图，我们很快会看到更多相关内容。这些指令可以把常见的视图定义成模板，它们可以设置应用的运行方式，或者用来创建可复用的组件。

同时，你并不会受限于 Angular 内置的指令。你可以编写自己的指令用来扩展 HTML 模板的功能，从而实现你梦想中的任何东西。

实例：购物车

我们来看一个稍微大一点的例子，它将会展示 Angular 的更多特性。想象一下，我们打算构建一款购物应用，需要在应用中的某个地方展示用户的购物车，并且用户能够对其进行编辑。我们直接跳到购物车这个部分：

```
<html ng-app='myApp'>
<head>
  <title>Your Shopping Cart</title>
</head>
```

译注 1：著名的设计原则，核心含义是一个类要尽量少地知道其他类的相关信息。来源于 1987 年荷兰大学的 Demeter 项目。

```
<body ng-controller='CartController'>
  <h1>Your Order</h1>
  <div ng-repeat='item in items'>
    <span>{{item.title}}</span>
    <input ng-model='item.quantity'>
    <span>{{item.price | currency}}</span>
    <span>{{item.price * item.quantity | currency}}</span>
    <button ng-click="remove($index)">Remove</button>
  </div>
  <script src="angular.js"></script>
  <script>
    function CartController($scope) {
      $scope.items = [
        {title: 'Paint pots', quantity: 8, price: 3.95},
        {title: 'Polka dots', quantity: 17, price: 12.95},
        {title: 'Pebbles', quantity: 5, price: 6.95}
      ];

      $scope.remove = function(index) {
        $scope.items.splice(index, 1);
      }
    }
  </script>
  </body>
</html>
```

最终 UI 截屏如图 1-4 所示。

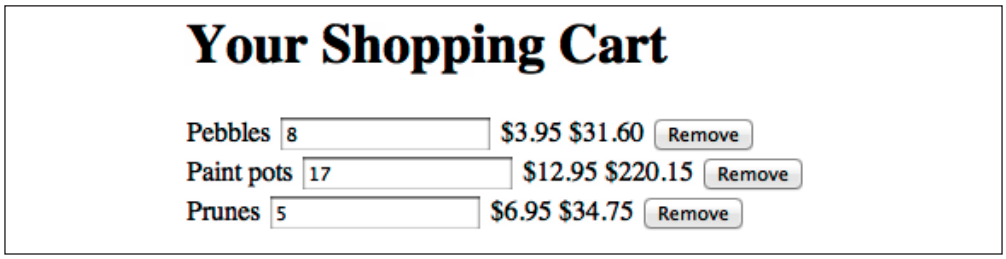


图1-4：购物车UI

下面是对这段代码的一个概要解释，本书后续的内容将会对此做更深入的解析。

我们从头开始：

```
<html ng-app>
```

ng-app 属性将用来告诉 Angular 页面中的哪一部分需要接受它的管理。既然我们把这个属性放在 <html> 标签上，那么就是在告诉 Angular，我们希望它管理整个页面。通常情

况下你只需要这么做就可以了，但是，如果你打算把 Angular 集成到一个现有的应用中，而这个应用使用了其他的方式来管理页面，那么你可能需要把 `ng-app` 属性放到应用中的一个 `<div>` 上。

```
<body ng-controller='CartController'>
```

在 Angular 中，你将会使用一种叫做控制器的 JavaScript 类来管理页面中的区域。在 body 标签中引入一个控制器，就是在声明 `CartController` 将会管理介于 `<body>` 和 `</body>` 之间的所有内容。

```
<div ng-repeat='item in items'>
```

`ng-repeat` 的意思是，对于 *items* 数组中的每一个元素，都把 `<div>` 中的 DOM 结构复制一份（包括 div 自身）。对于 div 的每一份拷贝，都会把一个叫做 *item* 的属性设置给它，这样我们就可以在模板中使用这份拷贝的元素了。如你所见，这样一来就会产生 3 个 `<div>`，其中分别包含了产品的名称、数量、单价、总价，以及一个可以用来完全删除一个项目的按钮。

```
<span>{{item.title}}</span>
```

正如我们在“Hello, World”那个例子中所展示的，通过 `{{ }}` 进行数据绑定让我们可以把变量的值插入到页面的一部分中，同时能够保证它会自动同步。完整的表达式 `{{item.title}}` 将会获取循环中的当前 item，然后把这个 item 的 title 属性值插入到 DOM 中。

```
<input ng-model='item.quantity'>
```

定义 `ng-model` 将会在输入框和 `item.quantity` 的值之间创建数据绑定关系。

8

`<span>` 里面的 `{{ }}` 将会创建一个单向的关系，也就是说“在这里插入一个值”。这正是我们所要达到的效果，但是应用还需要知道用户何时修改了数量，这样它就可以修改总价了。

使用 `ng-model` 我们就可以保持变更与模型同步了。`ng-model` 声明将会把 `item.quantity` 的值插入文本框中，同时，不管什么时候，只要用户输入了新的数值，它就会自动更新 `item.quantity` 的值。

```
<span>{{item.price | currency}}</span>
<span>{{item.price * item.quantity | currency}}</span>
```

我们想让单价和总价能够以美元的格式显示。Angular 带有一种叫做过滤器（filter）的特性，我们可以用它来转换文本的格式，有一个内置过滤器叫做 `currency`（货币），它可以为我们实现美元格式化。在下一章中我们将会看到关于过滤器的更多内容。

```
<button ng-click='remove($index)'>Remove</button>
```

这个按钮可以让用户从他们的购物车中删除项目，点击产品旁边的 *Remove* 按钮即可，因为我们已经设置好了，点击这个按钮将会调用 `remove()` 函数。同时我们还会把 `$index` 传递过去，`$index` 包含了 `ng-repeat` 过程中的循环计数，这样一来我们就知道要删除哪一个项目了。

```
function CartController($scope) {
```

`CartController` 将会负责管理购物车的业务逻辑。在函数的形参中放一个 `$scope` 就可以告诉 Angular：控制器需要一个叫做 `$scope` 的东西。我们可以通过 `$scope` 把数据绑定到 UI 中的元素上。

```
$scope.items = [
  {title: 'Paint pots', quantity: 8, price: 3.95},
  {title: 'Polka dots', quantity: 17, price: 12.95},
  {title: 'Pebbles', quantity: 5, price: 6.95}
];
```

通过定义 `$scope.items`，我们创建了一个虚拟的 hash 型数据，用来表示用户购物车中的项目集合。我们想让这些项目能够对 UI 的数据绑定有效，所以我们要把它们设置到 `$scope` 上。

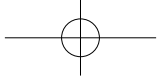
当然，这个例子的真实版本不可能只是在内存里面运行，它还需要与服务端进行交互并正确地把数据持久化。我们将会在后续的章节中讨论这一话题。

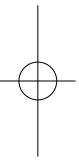
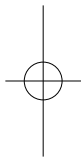
```
$scope.remove = function(index) {
  $scope.items.splice(index, 1);
}
```

在绑定 UI 的时候，我们希望 `remove()` 函数也有效，所以我们也需要把它设置到 `$scope` 上。对于购物车的纯内存版，`remove()` 函数可以只从数组中删除元素。由于 `ng-repeat` 所创建的 `<div>` 列表都是绑定在数据上的，所以当数组中的项目消失时，这个列表将会自动收缩。记住，无论何时，只要用户点击了 Remove 按钮，就会从 UI 中调用 `remove()` 函数。

接下来

我们已经看过了 Angular 最基本的一些风格，以及一些非常简单的例子。在本书后续的内容中，我们将会展示框架所必须提供的各种特性。

第2章

11

AngularJS应用骨架

使用典型的类库时，你可以选择并使用你所喜欢的功能；而对于 Angular 框架来说，你必须把它作为一个完整的套件来使用，框架中的所有东西都包含在里面。在本章中，我们将会覆盖 Angular 的所有基础模块，这样你就可以理解它们是如何被装配到一起的。在后续的章节中，将会对这里涉及的很多模块做出更加详细的解析。

调用Angular

为了使用 Angular，所有应用都必须首先做两件事情：

1. 加载 *angular.js* 库。
2. 使用 `ng-app` 指令告诉 Angular 应该管理 DOM 中的哪一部分。

加载脚本

加载类库的操作非常简单，与其他 JavaScript 类库遵循同样的规则。你可以从 Google 的内容分发网络（CDN）中加载类库[1]，示例如下：

```
<script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.4/angular.min.js">
</script>
```

我们推荐你使用 Google 的 CDN，Google 的服务器非常快，并且可以在多个应用之间缓存脚本库。也就是说，如果你的用户有多款应用使用了 Angular，类库只要下载一次就

译注 1：　鉴国内的特殊情况，请国内开发者不要使用这种方式。

可以了。同时，如果用户访问了其他站点，而该站点使用了 Google 的 CDN 链接来获取 Angular 库，那么用户在访问你的站点时，就不需要再次下载 Angular 库了。

如果你更喜欢使用本地（或者其他任何地方）的主机，当然也可以，只需要在 `src` 属性中指定正确的地址即可。

12

使用ng-app声明Angular的边界

你可以用 `ng-app` 指令告诉 Angular 应该管理页面中的哪一块。如果你正在构建一款纯 Angular 应用，那么你应该把 `ng-app` 指令写在 `<html>` 标签中，示例如下：

```
<html ng-app>
  …
</html>
```

这样就会告诉 Angular 去管理页面上的所有 DOM 元素。

如果你有一款现存的应用，该应用正在用其他一些技术（如 Java 或 Rails）来管理 DOM，那么你可以让 Angular 只管理页面中的一部分，只要在页面中放入一些 `<div>` 之类的元素即可。

```
<html>
  …
  <div ng-app>
      …
  </div>
  …
</html>
```

Model View Controller

在第 1 章中，我们曾经提到过 Angular 支持 Model View Controller 风格的应用设计。尽管设计 Angular 应用具有很大的灵活性，但是以下这些风格你将会经常涉及：

- 用来容纳数据的模型，模型代表应用当前的状态。
- 用来展示数据的一些视图。
- 用来管理模型和视图之间关系的一些控制器。

你将会使用对象的属性来创建数据模型，甚至只用基本数据类型来存储数据。模型属性并没有什么特殊的东西。如果你想要给用户展示一些文本，你可以使用字符串，示例如下：

```
var someText = 'You have started your journey.';
```

你可以编写一个 HTML 页面来创建视图，然后把它和你模型中的数据融合起来。正如我

们前面所看到的，你可以在 DOM 中插入占位符，然后像下面这样设置其中的文本：

```
<p>{{someText}}</p>
```

我们把这叫做双花括号插值语法，因为它可以把新的内容插入到现有的模板中。

控制器就是你所编写的类或者类型，它的作用是告诉 Angular 该模型是由哪些对象或者基本数据构成的，只要把这些对象或者基本数据设置到 $scope 对象上即可，$scope 对象会被传递给控制器：

```
function TextController($scope) {
  $scope.someText = someText;
}
```

把以上所有东西整合起来，我们就获得了如下内容：

```
<html ng-app>
<body ng-controller="TextController">
  <p>{{someText}}</p>

  <script
      src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.1/angular.min.js">
  </script>

  <script>
    function TextController($scope) {
      $scope.someText = 'You have started your journey.';
    }
  </script>
</body>
</html>
```

把以上内容加载到浏览器中，你将会看到：

```
You have started your journey.[2]
```

在很简单的情况下，以上使用基本数据类型的模型工作得很好，但是对于大多数应用来说，你都需要创建模型对象类来容纳你的数据。我们来创建一个 messages 模型对象，然后用它来存储 someText 属性。所以模型不再是以下内容：

```
var someText = 'You have started your journey.';
```

而会这样编写：

```
var messages = {};
messages.someText = 'You have started your journey.';
```

译注 2：　你的旅程已经开始。

```
function TextController($scope) {
  $scope.messages = messages;
}
```

然后在你的模板中这样使用：

```
<p>{{messages.someText}}</p>
```

在稍后讨论 `$scope` 对象的时候我们将会看到，像这样创建模型对象的方式可以避免一些非预期的行为，这些行为可能是由 `$scope` 对象中的原型继承机制所导致的。

在上一个例子中，我们在全局作用域中创建了 `TextController`。虽然对于示例代码来说，这没有什么问题，但是其实定义控制器的正确方式是，把它定义成模块的一部分，控制器可以为应用里面相关的部分提供命名空间机制。修改之后的代码看起来就像下面这样：

14

```
<html ng-app='myApp'>
<body ng-controller='TextController'>
  <p>{{someText.message}}</p>

<script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.1/angular.min.js">
</script>

<script>
  var myAppModule = angular.module('myApp', []);

  myAppModule.controller('TextController',
      function($scope) {
    var someText = {};
    someText.message = 'You have started your journey.';
    $scope.someText = someText;
  });
</script>
</body>
</html>
```

在这个版本中，我们把 `ng-app` 属性设置成了模块的名称 *myApp*。然后我们再调用 angular 对象创建了一个名为 *myApp* 的模块，并且把控制器函数传递给了 myApp 模块的 controller 函数。

稍后我们就会来解释什么是模块以及为什么要使用模块之类的问题。现在你只要记住，把东西从全局命名空间中隔离开是一件非常好的事情，而模块机制可以帮助我们实现这一点。

模板和数据绑定

Angular 应用中的模板只是一些 HTML 片段而已，我们可以从服务器上加载，或者在 `<script>` 标签中定义，处理方式与所有其他静态资源相同。如果你需要 UI 组件，你可以在模板中进行定义，使用标准的 HTML 加上 Angular 指令即可。

模板一旦加载到浏览器之后，Angular 将会把它和数据整合起来，然后再把这些模板展开到整个应用中。在第 1 章中，当我们显示购物车中的物品时，我们已经看到过这种例子：

```
<div ng-repeat="item in items">
  <span>{{item.title}}</span>
  ...
</div>
```

这里，对于 items 数组中的每一个元素，Angular 将会给外层 `<div>` 生成一份拷贝，包括其中的所有内容。

那么，这里的数据是从哪儿来的呢？在购物车实例中，我们只是在代码中的一个数组里定义了它。在你刚开始构建 UI，并且只是想测试一下它的运行效果的时候，这样能工作得很好。但是，大多数应用都会用到服务端的一些持久化的数据。浏览器中的应用将会连接到服务端，请求用户当前加载页面所需要的数据，然后 Angular 再把这些数据和模板融合起来。

基本的运作流程如下。

1. 用户请求应用起始页。
2. 用户的浏览器向服务器发起一次 HTTP 连接，然后加载 `index.html` 页面，这个页面里面包含了模板。
3. Angular 被加载到页面中，等待页面加载完成，然后查找 `ng-app` 指令，用来定义模板边界。
4. Angular 遍历模板，查找指令和绑定关系，这将触发一系列动作：注册监听器、执行一些 DOM 操作、从服务器获取初始化数据。这项工作的最后结果是，应用将会启动起来，并且模板被转换成了 DOM 视图。
5. 连接到服务器去加载需要展示给用户的其他数据。

对于每一个 Angular 应用来说，步骤 1 到步骤 3 都是标准化的，步骤 4 和步骤 5 是可选的。这些步骤可以同步进行也可以异步进行。为了提升性能，对于应用中的第一个视图，你

可以把数据和 HTML 模板一起加载进来，从而避免发起多次请求。

通过 Angular 来构建应用可以将应用中的模板和数据分离开来。这使得这些模板可以缓存起来。在第一次请求之后，只需要把新的数据下载到浏览器中即可。与 JavaScript、图片、CSS 以及其他资源一样，把这些模板缓存起来可以提升应用的性能。

显示文本

使用 `ng-bind` 指令，你可以在 UI 中的任何地方显示并刷新文本。它还有两种等价形式。第一种是使用花括号的形式，我们已经见过了：

```
<p>{{greeting}}</p>
```

还有一种方式就是使用基于属性的指令，叫做 `ng-bind`：

```
<p ng-bind="greeting"></p>
```

对于输出的内容来说两种形式是等价的。如果模型变量 greeting 被设置为“Hi there”，Angular 将会生成以下 HTML：

```
<p>Hi there</p>
```

16 然后浏览器就会显示“Hi there”。

那么，你为什么要使用一种方式替代另一种呢？我们创建双花括号插值语法的目的是为了阅读起来更加自然，并且需要输入的内容更少。虽然两种形式的输出内容相同，但是使用双花括号语法的时候，在 Angular 使用数据替换这些花括号之前，第一个加载的页面，也就是应用中的 *index.html*，其未被渲染好的模板可能会被用户看到。而使用第二种方法的视图就不会遇到这个问题。

造成这种现象的原因是，浏览器需要首先加载 HTML 页面，渲染它，然后 Angular 才有机会把它解释成你期望看到的内容。

好消息是，在大多数模板中你依然可以使用 `{{ }}`。但是，对于 index.html 页面中的数据绑定操作，建议使用 `ng-bind`。这样一来，在数据加载完成之前你的用户就不会看到任何内容。

表单输入

在 Angular 中使用表单元素非常方便。正如我们在前面几个例子中看到的，你可以使用 `ng-model` 属性把元素绑定到你的模型属性上。这一机制对于所有标准的表单元素都可以起作用，例如文本框、单选按钮、复选框，等等。我们可以像下面这样把一个复选框绑

定到一个属性上：

```
<form ng-controller="SomeController">
  <input type="checkbox" ng-model="youCheckedIt">
</form>
```

这样做的含义是：

1. 当用户选中了复选框之后，SomeController 中 $scope 的 youCheckedIt 的属性就会变成 true。而反选复选框会让 youCheckedIt 变为 false。

2. 如果你在 SomeController 中把 $scope.youCheckedIt 设置为 true，那么 UI 中的复选框将会变成选中状态。设置为 false 将会反选复选框。

举例来说，当用户做了某件事情的时候，我们希望程序能真正地做出某种动作。对于输入元素来说，你可以使用 ng-change 属性来指定一个控制器方法，一旦用户修改了输入值，这个方法就会被调用。我们来做一个简单的计算，帮助消费者计算一下需要付多少钱：

```
<form ng-controller="StartUpController">
  Starting: <input ng-change="computeNeeded()"
                   ng-model="funding.startingEstimate">
  Recommendation: {{funding.needed}}
</form>
```

对于这个非常简单的例子来说，我们只要把输出文本框的值设置为用户估价的 10 倍即可。同时，在一开始的时候我们会把文本框的默认值设置为 0：

```
function StartUpController($scope) {
  $scope.funding = { startingEstimate: 0 };

  $scope.computeNeeded = function() {
    $scope.needed = $scope.startingEstimate * 10;
  };
}
```


但是，以上代码所采取的策略有一个潜在的问题，即，只有当用户在文本框中输入值的时候我们才会去计算所需的金额。当用户在这个特定的输入框中输入时，输入框就会正确地刷新。但是，如果还有其他输入框也绑定到模型中的这个属性上，会怎么样呢？如果接收到服务端的数据，导致数据模型进行刷新，又会怎么样呢？

为了能够正确地刷新输入框，而不管它是通过何种途径进行刷新的，我们需要使用 $scope 中的 $watch() 的函数。在本章后续的内容里将会详细讨论这个 watch 函数。最基本的一点就是，你可以调用 $watch() 函数来监视一个表达式，当这个表达式发生变化时就会调用一个回调函数。

在当前这个例子中，我们需要监视 funding.startingEstimate，当它的值发生变化时就调用 computeNeeded()。下面运用这一技术重写 StartUpController：

```
function StartUpController($scope) {
  $scope.funding = { startingEstimate: 0 };

  computeNeeded = function() {
    $scope.funding.needed = $scope.funding.startingEstimate * 10;
  };

  $scope.$watch('funding.startingEstimate', computeNeeded);
}
```

请注意，需要监视的表达式位于引号中。是的，它是一个字符串。这个字符串会被当作 Angular 表达式来执行。表达式可以执行一些简单的操作，并且可以访问 $scope 对象中的属性。在本章后续的内容中我们将会详细解释表达式相关的内容。

我们还可以监视一个函数的返回值，但是监视 funding.startingEstimate 属性是没有用的，因为这个值算出来是 0，也就是它的初始值，而且这个值永远不会发生变化。

然后，无论何时只要 funding.startingEstimate 发生变化，我们的 funding.needed 就会自动刷新，我们可以编写一个更加简单的模板，示例如下：

```
<form ng-controller="StartUpController">
  Starting: <input ng-model="funding.startingEstimate">
  Recommendation: {{funding.needed}}
</form>
```

18

在某些情况下，你不想一有变化就立刻做出动作；而是要进行等待，直到用户告诉你他已经准备好了，例如完成订购或者发出一条确认信息之后。

如果你正在使用表单把输入项组织起来，你可以在 form 自身上使用 ng-submit 指令来指定一个函数，当表单提交的时候可以执行这个函数。我们来扩展前面的例子，实现用户点击一个按钮就可以为他所选的商品计算金额：

```
<form ng-submit="requestFunding()" ng-controller="StartUpController">
  Starting: <input ng-change="computeNeeded()" ng-model="startingEstimate">
  Recommendation: {{needed}}
  <button>Fund my startup!</button>
</form>

function StartUpController($scope) {
  $scope.computeNeeded = function() {
    $scope.needed = $scope.startingEstimate * 10;
  };
```

```
  $scope.requestFunding = function() {
    window.alert("Sorry, please get more customers first.");
  };
}
```

同时，在提交表单的时候，ng-submit 还会自动阻止浏览器执行默认的 POST 操作。

对于处理其他事件的情况，例如提供非表单提交型的交互，Angular 提供了一些事件处理指令，它们类似于浏览器原生的事件属性。对于 onclick，可以使用 ng-click；对于 ondblclick，可以使用 ng-dblclick；等等。

我们可以试着实现一个这样的功能：再次扩展开始的那个计算器，为之添加一个复位按钮，这个按钮将会把输入值重置为 0：

```
<form ng-submit="requestFunding()" ng-controller="StartUpController">
   Starting: <input ng-change="computeNeeded()" ng-model="startingEstimate">
   Recommendation: {{needed}}
   <button>Fund my startup!</button>
   <button ng-click="reset()">Reset</button>
</form>

function StartUpController($scope) {
  $scope.computeNeeded = function() {
    $scope.needed = $scope.startingEstimate * 10;
  };

  $scope.requestFunding = function() {
    window.alert("Sorry, please get more customers first.");
  };

  $scope.reset = function() {
    $scope.startingEstimate = 0;
  };
}
```


浅谈非入侵式JavaScript

在你 JavaScript 开发生涯中的某个时间点上，某人可能会告诉你，你应该编写“非入侵式的 JavaScript”，在 HTML 中使用 click、mousedown 以及其他内联的事件处理器是一种非常糟糕的处理方式。他是对的。

非入侵式 JavaScript 的观点可以从很多角度进行解读，大体是针对以下几个观点：

1. 并不是每个人的浏览器都支持 JavaScript。你需要让每个人都能看到你的全部内容，而且无须在浏览器中执行代码就能够使用你的应用。

2. 有些人会使用一些运行方式非常奇怪的浏览器。比如视觉受损的人会使用屏幕阅读器，一些手机用户无法使用含有 JavaScript 的站点。

3. JavaScript 在不同的平台上运行方式不同。IE 是造成这一问题的罪魁祸首。我们需要根据不同的浏览器编写不同的事件处理代码。

4. 这些事件处理器都会引用全局命名空间中的函数。如果你想把其他类库集成进来，而这些类库又带有相同的函数名称，那么你就会头疼了。

5. 这些事件监听器都会绑定数据结构和行为。这让你的代码更加难以维护、扩展和理解。

在大多数情况下，使用这种风格编写 JavaScript 代码还说得过去。但是，有一件事情不太好，那就是代码的复杂度和可读性。如果把事件处理器定义在需要处理的元素上，你通常需要为这些元素赋 ID 值，获取元素的引用，然后使用回调函数设置事件处理器。你当然可以发明一种结构，只在大家都熟悉的地方去创建这些关联关系，但是大多数应用最后还是会把这些处理器弄得乱七八糟。

在 Angular 中，我们决定重新审视这个问题。

自从前面这 5 个观点被提出以后，世事早已沧海桑田。对于所有人口样本来说，上面的观点 1 已经不再正确。如果你正在运行的浏览器不支持 JavaScript，那么你就被降到了 19 世纪 90 年代那些网站的状态。对于观点 2，最新的屏幕阅读器已经赶上来了。正确地使用 ARIA[3] 语义学标签，你可以制作出非常易于访问的 UI。现在移动电话运行 JavaScript 的水平已经可以和桌面计算机平起平坐。

那么，现在的问题就是：我们是否能够解决观点 3 和观点 4 中的问题，同时还能获得在标签中内联代码技术的可读性和简单性？

正如前面所提到的，对于大多数内联的事件监听器来说，Angular 有一种等价的形式 `ng-eventhandler="expression"`，这里的 `eventhandler` 可以被替换成 `click`、`mousedown`、`change` 等。如果你希望当用户点击一个元素的时候能够收到通知，你可以像下面这样简单地使用 `ng-click` 指令：

20

```
<div ng-click="doSomething()">...</div>
```

你的大脑中是不是正在说“不，不！不好，不好！”？放松，这些指令和原来的事件处理器有以下不同之处：

在所有浏览器中具有相同的行为。Angular 将会帮你屏蔽差异性。

译注 3：　Accessible Rich Internet Applications，易于访问的富互联网应用。

不会在全局命名空间中进行操作。你所指定的表达式只能访问元素控制器作用域范围内的函数和数据。

第二点可能会有点模糊，我们来看一个例子。在一款典型的应用中，你会创建一个导航栏和一个内容区域，当你在导航栏中选择不同的菜单项时，内容区域会发生相应的改变。我们可以编写出一个大概的骨架，示例如下：

```
<div class="navbar" ng-controller="NavController">
…
   <li class="menu-item" ng-click="doSomething()">Something</li>
…
</div>

<div class="contentArea" ng-controller="ContentAreaController">
…
   <div ng-click="doSomething()">...</div>
…
</div>
```

当用户点击的时候，导航栏中的两个`<li>`和内容区域中的`<div>`都会调用`doSomething()`的函数。作为开发者，你可以在控制器代码中写好这些调用所引用的函数。实际上这两个函数可以是同一个函数，也可以是不同的函数：

```
function NavController($scope) {
  $scope.doSomething = doA;
}

function ContentAreaController($scope) {
  $scope.doSomething = doB;
}
```

这里，`doA`和`doB`两个函数可以是同一个函数，也可以是不同的函数，就看你如何定义它们了。

现在还剩下观点 5，事件处理器会绑定数据结构和行为。这种争论是毫无来由的，因为你无法具体指出这一做法会导致什么样的负面作用。但是，这个问题和我们所要考虑的另一个真正棘手的问题非常类似，那就是：界面展示相关的功能会和业务逻辑混合在一起。这样做确实会导致一些副作用，也就是大家在描述 structure/behavior（数据结构 / 行为）问题时所说的内容。

对于你的系统是否会遭遇这种耦合，我们有一种简单的测试方法可以验证：我们能否为应用逻辑编写一个单元测试而不需要 DOM 结构存在？

21

是的，在 Angular 中我们可以编写包含业务逻辑的控制器，而无须引用 DOM。问题不是

出在事件处理器上，而是之前我们编写 JavaScript 的方式有问题。请注意，到目前为止我们所编写的所有控制器中，都没有在任何地方引用 DOM 或者 DOM 事件。所有定位元素和处理事件的工作都是在 Angular 内部完成的。

在编写单元测试的时候这一点很重要。如果你需要 DOM，你就需要在测试用例中首先创建 DOM，这就增加了测试的复杂性。并且，当你的页面发生变化时，还有更多的维护工作要做，你需要为测试用例同步修改 DOM。最后，访问 DOM 的操作非常慢。很慢的测试就意味着很慢的反馈，甚至很慢的交付。而 Angular 的控制器测试就不会有这些问题。

最后。你可以开心地使用 Angular 的声明式事件处理器，同时又能保持简单性和可读性，所以没有必要为违反最佳实践而感到内疚。

列表、表格以及其他迭代型元素

`ng-repeat` 可能是最有用的 Angular 指令了，它可以根据集合中的项目一次创建一组元素的多份拷贝。不管在什么地方，只要你想创建一组事物的列表，你就可以使用这条指令。

比方说，我们正在为老师们编写一套学生花名册系统。我们当然需要从服务器上获取学生信息，但是对于当前这个例子来说，我们还是把模型直接定义在 JavaScript 代码里面：

```
var students = [{name:'Mary Contrary', id:'1'},
                {name:'Jack Sprat', id:'2'},
                {name:'Jill Hill', id:'3'}];

function StudentListController($scope) {
  $scope.students = students;
}
```

为了显示这个学生列表，我们可以编写下面的代码：

```
<ul ng-controller='StudentListController'>
  <li ng-repeat='student in students'>
    <a href='/student/view/{{student.id}}'>{{student.name}}</a>
  </li>
</ul>
```

`ng-repeat` 将会生成标签内部所有 HTML 元素的一份拷贝，包括放指令的标签。经过这样的操作之后，我们将会看到以下结果：

- Mary Contrary
- Jack Sprat
- Jill Hill

根据具体情况分别链接到 */student/view/1*、*/student/view/2* 以及 */student/view/3*。

正如我们前面所看到的，修改学生信息数组将会自动刷新所渲染的列表。如果我们需要向列表中插入一条新的学生信息，可以这样写：

```
var students = [{name:'Mary Contrary', id:'1'},
                {name:'Jack Sprat', id:'2'},
                {name:'Jill Hill', id:'3'}];

function StudentListController($scope) {
  $scope.students = students;

  $scope.insertTom = function () {
    $scope.students.splice(1, 0, {name:'Tom Thumb', id:'4'});
  };
}
```

然后添加一个按钮在模板中调用新增的函数：

```
<ul ng-controller='StudentListController'>
  <li ng-repeat='student in students'>
    <a href='/student/view/{{student.id}}'>{{student.name}}</a>
  </li>
</ul>
<button ng-click="insertTom()">Insert</button>
```

我们现在将会看到：

- Mary Contrary
- Tom Thumb
- Jack Sprat
- Jill Hill

`ng-repeat` 指令可以通过 `$index` 返回当前引用的元素序号；还可以通过 `$first`、`$middle` 及 `$last`，`ng-repeat` 指令返回布尔值，告诉你当前元素是否是集合中的第一个元素、中间的某个元素，或者最后一个元素。

你可能需要使用 `$index` 在表格中显示出行号，那么你只要编写下面这样的模板即可：

```
<table ng-controller='AlbumController'>
  <tr ng-repeat='track in album'>
    <td>{{$index + 1}}</td>
    <td>{{track.name}}</td>
    <td>{{track.duration}}</td>
  </tr>
</table>
```

对应的控制器代码如下：

23
```
var album = [{name:'Southwest Serenade', duration: '2:34'},
             {name:'Northern Light Waltz', duration: '3:21'},
             {name:'Eastern Tango', duration: '17:45'}];

function AlbumController($scope) {
  $scope.album = album;
}
```

然后我们就可以获得以下输出：

```
1   Southwest Serenade        2:34
2   Northern Light Waltz      3:21
3   Eastern Tango             17:45
```

隐藏和显示

对于菜单、上下文敏感的工具以及很多其他情况来说，显示和隐藏元素是一项核心的功能。与 Angular 中其他功能一样，我们通过修改数据模型的方式来驱动 UI 刷新，然后通过指令把变更反应到 UI 上。

这个例子将会使用 `ng-show` 和 `ng-hide`。这两条指令的功能是等价的，但是运行效果正好相反，它们都可以根据你所传递的表达式来显示或者隐藏元素。也就是说，`ng-show` 在表达式为 true 时将会显示元素，为 false 时将会隐藏元素；而 `ng-hide` 则恰好相反。哪一条指令更能表达你的意图，你就使用哪一条。

这两条指令的工作原理是：根据实际情况把元素的样式设置为 `display:block` 来显示元素；设置为 `display:none` 来隐藏元素。我们来看一个虚构的例子，在这个例子中我们将为 Death Ray[4] 构建控制面板。

```
<div ng-controller='DeathrayMenuController'>
  <button ng-click='toggleMenu()'>Toggle Menu</button>
  <ul ng-show='menuState.show'>
    <li ng-click='stun()'>Stun</li>
    <li ng-click='disintegrate()'>Disintegrate</li>
    <li ng-click='erase()'>Erase from history</li>
  </ul>
<div/>

function DeathrayMenuController($scope) {
  $scope.menuState.show = false;

  $scope.toggleMenu = function() {
    $scope.menuState.show = !$scope.menuState.show;
  };

   // Death Ray 的具体功能留给读者自行练习
}
```

译注 4：　死亡射线，一种科幻武器。

CSS类和样式

现在很明显，你可以在你的应用中动态地设置 CSS 类和样式了，只要使用 {{ }} 插值语法把它们进行数据绑定即可。你甚至还可以在模板中构造 CSS 类名的部分匹配方式。例如，如果你想要有条件地禁用某些菜单，你可能会做以下事情来给用户视觉上的提示。

假设有下面这样一段 CSS：

```
.menu-disabled-true {
  color: gray;
}
```

使用以下模板，你就可以把 Death Ray 的 stun（击晕）功能显示成禁用状态：

```
<div ng-controller='DeathrayMenuController'>
  <ul>
    <li class='menu-disabled-{{isDisabled}}' ng-click='stun()'>Stun</li>
    ...
  </ul>
<div/>
```

然后你就可以根据需要，通过控制器来设置 isDisabled 属性了：

```
function DeathrayMenuController($scope) {
  $scope.isDisabled = false;

  $scope.stun = function() {
    // 击晕目标，然后禁用复活按钮
    $scope.isDisabled = 'true';
  };
}
```

stun 菜单项上的 CSS 类将会被设置为 menu-disabled- 加上 $scope.isDisabled 的值。由于 $scope.isDisabled 属性的初始值为 false，所以拼接出来的结果就是 menu-disabled-false。由于这个结果无法匹配到任何 CSS 样式，所以不会产生任何效果。当 $scope.isDisabled 被设置为 true 时，CSS 样式类就变成了 menu-disabled-true，这样就会调用相应的样式让文本变成灰色。

当使用插值的方式绑定内联样式的时候，这一技术同样适用，例如 style="{{some expression}}"。

虽然这种方式具有很大灵活性，但是也有一些不利的地方，那就是在构造 CSS 类名时存在一定程度的间接性。虽然在这个小例子里面很容易理解它，但是当需要同时在模板和 JavaScript 中使用时，它很快就会变得无法维护，进而无法正确地创建 CSS。

正是由于这个原因，Angular 提供了 ng-class 和 ng-style 指令。这两个指令都可以接受一个表达式，表达式执行的结果可能是如下取值之一：

25

- 表示 CSS 类名的字符串，以空格分隔。
- CSS 类名数组。
- CSS 类名到布尔值的映射。

我们来想象一下，你现在希望在应用头部的固定位置向用户显示错误和警告信息。使用 ng-class 指令，你可以这样做：

```
.error {
  background-color: red;
}

.warning {
  background-color: yellow;
}

<div ng-controller='HeaderController'>
  ...
  <div ng-class='{error: isError, warning: isWarning}'>{{messageText}}</div>
  …
  <button ng-click='showError()'>Simulate Error</button>
  <button ng-click='showWarning()'>Simulate Warning</button>
</div>

function HeaderController($scope) {
  $scope.isError = false;
  $scope.isWarning = false;

  $scope.showError = function() {
    $scope.messageText = 'This is an error!';
    $scope.isError = true;
    $scope.isWarning = false;
  };

  $scope.showWarning = function() {
    $scope.messageText = 'Just a warning.  Please carry on.';
    $scope.isWarning = true;
    $scope.isError = false;
  };
}
```

你还可以做一些更炫的事情，例如把表格中被选中的行进行高亮显示。比方说我们正在构建一个餐馆名录，而且想把用户点击的那一行进行高亮显示。

在 CSS 中，我们可以为需要高亮显示的行设置一个样式：

```
.selected {
  background-color: lightgreen;
}
```

在模板中，我们把 ng-class 设置为 {selected: $index==selectedRow}。这样做的效果就是，当模型属性 selectedRow 的值等于 ng-repeat 中的 $index 时，selected 样式就会被设置到那一行上。我们还会设置一个 ng-click 指令，用来告诉控制器用户点击了哪一行：


```
<table ng-controller='RestaurantTableController'>
  <tr ng-repeat='restaurant in directory' ng-lick='selectRestaurant($index)'
      ng-class='{selected: $index==selectedRow}'>
    <td>{{restaurant.name}}</td>
    <td>{{restaurant.cuisine}}</td>
  </tr>
</table>
```

在我们的 JavaScript 中只是创建了一些虚拟的餐馆，然后创建了 selectRow 函数。

```
function RestaurantTableController($scope) {
  $scope.directory = [{name:'The Handsome Heifer', cuisine:'BBQ'},
                      {name:'Green's Green Greens', cuisine:'Salads'},
                      {name:'House of Fine Fish', cuisine:'Seafood'}];

  $scope.selectRestaurant = function(row) {
    $scope.selectedRow = row;
  };
}
```

反思src和href属性

当在 <img> 或者 <a> 标签上进行数据绑定时，在 src 或者 href 属性上简单使用 {{ }} 无法很好地运行。由于浏览器会优先使用并行的方式来加载图片和其他内容，所以 Angular 没有机会拦截到数据绑定请求。<img> 标签最自然的用法是：

```
<img src="/images/cats/{{favoriteCat}}">
```

但其实，这里应该使用 ng-src 指令，你的模板应该写成这样：

```
<img ng-src="/images/cats/{{favoriteCat}}">
```

类似地，对于 <a> 标签，应该使用 ng-href 指令：

```
<a ng-href="/shop/category={{numberOfBalloons}}">some text</a>
```

表达式

在模板中使用表达式是为了以充分的灵活性在模板、业务逻辑和数据之间建立联系，同时又能避免让业务逻辑渗透到模板中。

到现在为止，我们大多数情况下是把基本数据类型的引用作为表达式传递给 Angular 指令的，但是这些表达式实际上还能做更多事情。可以进行简单的数学运算（+、-、/、*、%），进行比较运算（==、!=、>、<、>=、<=），执行布尔逻辑（&&、||、!），以及位运算（\^、&、|）。你还可以调用控制器中 `$scope` 对象上所暴露的函数，引用数组和对象符号（[ ]、{ }、.）。

以下这些都是合法的表达式示例：

```
<div ng-controller='SomeController'>
  <div>{{recompute() / 10}}</div>
  <ul ng-repeat='thing in things'>
    <li ng-class='{highlight: $index % 4 >= threshold($index)}'>
      {{otherFunction($index)}}
    </li>
  </ul>
</div>
```

这里的第一个表达式，`recompute() / 10`，虽然是合法的，但是它也是把业务逻辑放到模板中的一个典型例子，应该避免这种做法。清晰地区分视图和控制器之间的职责可以保证含义明确并且易于测试。

尽管我们可以使用表达式做很多事情，但表达式是使用一个自定义的解析器来执行的（这个解析器是 Angular 的一部分），而不是用 JavaScript 的 `eval()` 函数执行的，所以局限性较大。

表达式是通过 Angular 内置的解析器执行的。在这款解析器中，你找不到循环结构（for、while 等）、流程控制操作符（if-else、throw），以及修改数据的操作符（++、--）。当你需要这些类型的操作符时，请在你的控制器中执行或者通过指令执行。

虽然在很多方面这里的表达式比 JavaScript 更严格，但是它们对 `undefined` 和 `null` 的容错性更好。如果遇到错误，模板只是简单地什么都不显示，而不会抛出一个 `NullPointerException` 错误。这样一来你就可以安全地使用未经初始化的模型值，而一旦它们被填入值之后就会立即出现在 UI 中。

区分UI和控制器的职责

在应用中控制器有三种职责：

- 为应用中的模型设置初始状态。
- 通过 $scope 对象把数据模型和函数暴露给视图（UI 模板）。
- 监视模型其余部分的变化，并采取相应的动作。

对于前两项职责，本章已经列举过很多例子，这里我们来简要地看看第三项。从概念上讲，控制器存在的目的是：当用户与视图进行交互时，它负责提供代码或者逻辑以便执行用户的想法。

为了让控制器保持小巧和可管理状态，我们的建议是，为视图中的每一块功能区域创建一个控制器。也就是说，如果你有一个菜单，就创建一个 MenuController；如果你有一个用来导航的面包屑[5]，那就写一个 BreadcrumbController；等等。

28

你可能已经开始明白了，但是需要明确的是，控制器是绑定在特定的 DOM 片段上的，这些片段就是它们需要负责管理的内容。有两种主要的方法可以把控制器关联到 DOM 节点上，第一种是在模板中通过 ng-controller 属性来声明，另一种是通过路由把它绑定到一个动态加载的 DOM 模板片段上，这个模板叫做视图。

我们将会在本章稍后的内容中来讨论视图和路由。

如果你的 UI 中带有一些非常复杂的区域，你可以创建嵌套的控制器，它们可以通过继承树结构来共享数据模型和函数，这样你就可以保持代码的简单和可维护性。嵌套控制器非常简单，你只要把控制器设置到 DOM 元素内部嵌套的元素上即可，示例如下：

```
<div ng-controller="ParentController">
  <div ng-controller="ChildController">...</div>
</div>
```

虽然我们把这种方式叫做控制器嵌套，但真实的嵌套发生在 $scope 对象上。通过内部的原型继承机制，父控制器对象上的 $scope 会被传递给内部嵌套控制器的 $scope。具体到上面例子就是，ChildController 的 $scope 对象可以访问 ParentController 的 $scope 对象上的所有属性（和函数）。

利用$scope暴露模型数据

利用向控制器传递 $scope 对象的机制，可以把模型数据暴露给视图。在你的应用中可能还有其他数据，但是只有通过 $scope 触及这些数据，Angular 才会把它当成数据模型的一部分。你可以把 $scope 当成一个上下文环境，它让数据模型上的变化变得可以观察。

译注 5：UI 设计中的术语。面包屑导航（Breadcrumb Navigation）的概念来自童话故事“汉赛尔和格莱特”，当汉赛尔和格莱特穿过森林时，不小心迷路了，他们在沿途走过的地方都撒下了面包屑，让这些面包屑来帮助他们找到回家的路。所以，面包屑导航的作用是告诉访问者他们目前在网站中的位置以及如何返回。

对于显式地创建 $scope 属性我们已经看到过很多例子了，例如 $scope.count = 5。还可以间接地通过模板自身创建数据模型。你可以通过以下几种方式来实现这一点。

1. 通过表达式。既然表达式是在控制器的 $scope 环境中执行的，而这个 $scope 与它们所管理的元素有关，那么在表达式中设置属性值就和设置控制器的 $scope 属性值是一样的。也就是说，这样做：

```
<button ng-click='count=3'>Set count to three</button>
```

与下面这种做法的效果相同：

```
<div ng-controller='CountController'>
 <button ng-click='setCount()'>Set count to three</button>
</div>
```

CountController 定义如下：

29

```
function CountController($scope) {
  $scope.setCount = function() {
    $scope.count=3;
  }
}
```

2. 在表单输入项上使用 ng-model。与表达式类似，ng-model 上指定的模型参数同样工作在外层控制器内。唯一的不同点在于，这样会在表单项和指定的模型之间建立双向绑定关系。

使用$watch监控数据模型的变化

在 scope 内置的所有函数中，用得最多的可能就是 $watch 函数了，当你的数据模型中某一部分发生变化时，$watch 函数可以向你发出通知。你可以监控单个对象的属性，也可以监控需要经过计算的结果（函数），实际上只要能够被当作属性访问到，或者可以当作一个 JavaScript 函数被计算出来，就可以被 $watch 函数监控。它的函数签名为

```
$watch(watchFn, watchAction, deepWatch)
```

其中每个参数的详细含义如下。

watchFn

该参数是一个带有 Angular 表达式或者函数的字符串，它会返回被监控的数据模型的当前值。这个表达式将会被执行很多次，所以你要保证它不会产生其他副作用。也就是说，要保证它可以被调用很多次而不会改变状态。基于同样的原因，监控表达式应该很容易被计算出来。如果你使用字符串传递了一个 Angular 表达式，那么

它将会针对调用它的那个作用域中的对象而执行。

`watchAction`

这是一个函数或者表达式，当 `watchFn` 发生变化时会被调用。如果是函数的形式，它将会接收到 `watchFn` 的新旧两个值，以及作用域对象的引用。其函数签名为 `function(newValue, oldValue, scope)`。

`deepWatch`

如果设置为 true，这个可选的布尔型参数将会命令 Angular 去检查被监控对象的每个属性是否发生了变化。如果你想要监控数组中的元素，或者对象上的所有属性，而不只是监控一个简单的值，你就可以使用这个参数。由于 Angular 需要遍历数组或者对象，如果集合比较大，那么运算负担就会比较重。

`$watch` 函数会返回一个函数，当你不再需要接收变更通知时，可以用这个返回的函数注销监控器。

如果我们需要监控一个属性，然后接着注销监控，我们可以使用以下代码：

```
...
var dereg = $scope.$watch('someModel.someProperty', callbackOnChange());
...
dereg();
```

30

我们再回到第 1 章的购物车案例，把它的功能扩充完整。例如，当用户添加到购物车中的商品价值超过 100 美元的时候，我们会给他 10 美元的折扣。我们将会使用下面这种模板：

```
<div ng-controller="CartController">
  <div ng-repeat="item in items">
    <span>{{item.title}}</span>
    <input ng-model="item.quantity">
    <span>{{item.price | currency}}</span>
    <span>{{item.price * item.quantity | currency}}</span>
  </div>
  <div>Total: {{totalCart() | currency}}</div>
  <div>Discount: {{bill.discount | currency}}</div>
  <div>Subtotal: {{subtotal() | currency}}</div>
</div>
```

而 `CartController` 看起来可能像下面这样：

```
function CartController($scope) {
  $scope.bill = {};
```

```
  $scope.items = [
    {title: 'Paint pots', quantity: 8, price: 3.95},
    {title: 'Polka dots', quantity: 17, price: 12.95},
    {title: 'Pebbles', quantity: 5, price: 6.95}
  ];

  $scope.totalCart = function() {
    var total = 0;
    for (var i = 0, len = $scope.items.length; i < len; i++) {
      total = total + $scope.items[i].price * $scope.items[i].quantity;
    }

    return total;
  }

  $scope.subtotal = function() {
    return $scope.totalCart() - $scope.discount;
  };

  function calculateDiscount(newValue, oldValue, scope) {
    $scope.bill.discount = newValue > 100 ? 10 : 0;
  }

  $scope.$watch($scope.totalCart, calculateDiscount);
}
```

31 注意 `CartController` 的底部，我们在 `totalCart()` 的值上面设置了一个监控，用来计算此次购物的总价。只要这个值发生变化，监控器就会调用 `calculateDiscount()`，然后我们就可以把折扣设置为相应的值。如果总价超过 100 美元，我们将会把折扣设置为 10 美元。否则，折扣为 0。

图 2-1 展示了用户将会看到的效果。

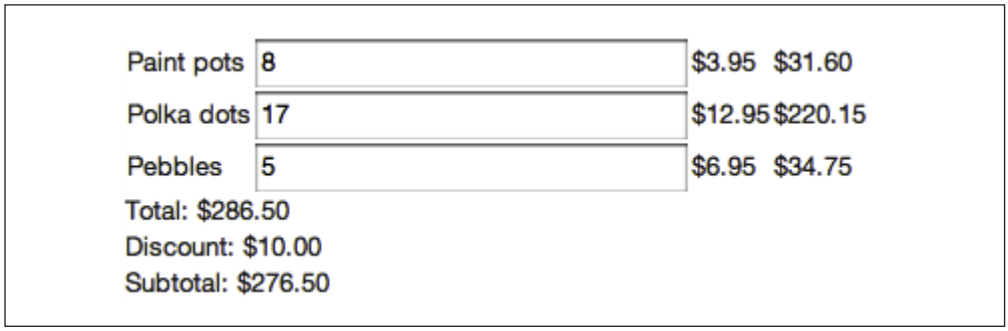


图2-1：带折扣的购物车

watch()中的性能注意事项

前面的例子可以正确地运行，但是却存在潜在的性能问题。如果你在 `totalCart()` 中打一个调试断点，你就会发现在渲染这个页面时，该函数被调用了 6 次。虽然在当前这个应用中它所引起的性能问题并不明显，但是在更加复杂的应用中，运行 6 次就会成为一个问题。

为什么是 6 次呢？其中 3 次我们可以很容易跟踪到，因为在以下每种情况下它都会运行一次：

- 模板 `{{totalCart() | currency}}`
- `subtotal()` 函数
- `$watch()` 函数

然后 Angular 把以上整个过程又重复了一遍，最终就是 6 次。Angular 这样做的目的是，检测模型中的变更已经被完整地进行了传播，并且模型已经被设置好。Angular 的做法是，把所有被监控的属性都拷贝一份，然后把它们和当前的值进行比较，看看是否发生了变化。实际上，Angular 可能会把以上过程运行 10 次，从而确保进行了完整的传播。如果经过 10 次重复之后被监控的属性还在发生变化，Angular 将会报错并退出。如果出现了这种情况，你可能需要解决循环依赖的问题。

虽然你现在对这个问题感到担心，但是当你读完本书之后，这可能就不是个问题了。由于 Angular 需要用 JavaScript 实现数据绑定，所以我们与 TC39 团队一起开发了一个叫做 `Object.observe()`（*http://updates.html5rocks.com/2012/11/Respond-to-change-with-object-observe*）的底层本地化实现。实现了这一点之后，Angular 将会在所有存在此实现的地方自动使用 `Object.observe()`，让你的数据绑定操作就像本地化代码一样快速。

正如你将会在下一章中所看到的，Angular 带有一个很好的 Chrome 调试插件 Batarang，它会自动为你指出哪些数据绑定操作比较昂贵。 32

有几种方法可以解决这个问题。一种方法是监控 items 数组的变化，然后重新计算 `$scope` 属性中的总价、折扣和小计值。

为了实现这一点，我们需要把模板修改一下以便使用这些属性：

```
<div>Total: {{bill.total | currency}}</div>
<div>Discount: {{bill.discount | currency}}</div>
<div>Subtotal: {{bill.subtotal | currency}}</div>
```

然后，在 JavaScript 中，我们将会监控 items 数组，当数组发生任何变化时调用一个函

数来计算总价，示例如下：

```
function CartController($scope) {
  $scope.bill = {};

  $scope.items = [
    {title: 'Paint pots', quantity: 8, price: 3.95},
    {title: 'Polka dots', quantity: 17, price: 12.95},
    {title: 'Pebbles', quantity: 5, price: 6.95}
  ];

  var calculateTotals = function() {
    var total = 0;
    for (var i = 0, len = $scope.items.length; i < len; i++) {
      total = total + $scope.items[i].price * $scope.items[i].quantity;
    }
    $scope.bill.totalCart = total;
    $scope.bill.discount = total > 100 ? 10 : 0;
    $scope.bill.subtotal = total - $scope.bill.discount;
  };

  $scope.$watch('items', calculateTotals, true);
}
```

请注意，上面的代码在调用 `$watch` 函数时把 `items` 写成了一个字符串。这样做是可以的，因为 `$watch` 函数既可以接受一个函数（就像我们前面做的），也可以接受一个字符串。如果把一个字符串传递给了 `$watch` 函数，它将会在被调用的 `$scope` 作用域中当成表达式来执行。

对于你的应用来说，这种策略可能会工作得很好。但是，既然我们在监控 items 数组，Angular 就会制作一份数组的拷贝，用来进行比较操作。对于大型的 items 数组来说，如果每次在 Angular 显示页面时只需要重新计算 bill 属性，那么性能会好很多。通过创建一个带有 `watchFn` 的 `$watch` 函数我们就可以实现这一点，`watchFn` 将会像下面这样来重新计算属性值：

```
$scope.$watch(function() {
  var total = 0;
  for (var i = 0; i < $scope.items.length; i++) {
    total = total + $scope.items[i].price * $scope.items[i].quantity;
  }
  $scope.bill.totalCart = total;
  $scope.bill.discount = total > 100 ? 10 : 0;
  $scope.bill.subtotal = total - $scope.bill.discount;
});
```

33

监控多个东西

如果你想监控多个属性或者对象，并且当其中任何一个发生变化时就去执行一个函数，应该怎么做呢？你有两种基本的选择：

- 监控把这些属性连接起来之后的值。
- 把它们放到一个数组或者对象中，然后给 `deepWatch` 参数传递一个 true 值。

在第一种情况下，如果在你的作用域中存在一个 *things* 对象，它带有两个属性 *a* 和 *b*，当这两个属性发生变化时都需要执行 `callMe()` 函数，你可以同时监控这两个属性，示例如下：

```
$scope.$watch('things.a + things.b', callMe(...));
```

当然，*a* 和 *b* 也可以属于不同的对象，只要需要，这个列表可以无限长。如果列表非常长，你就需要写一个函数来返回连接之后的值，而不是依赖一个表达式来完成这一逻辑。

在第二种情况下，需要监控 `things` 对象上的所有属性，你可以这样做：

```
$scope.$watch('things', callMe(...), true);
```

这里，给第三个参数传递了一个 *true*，请求 Angular 遍历 `things` 的属性，然后当其中任何一个属性发生变化时就调用 `callMe()`。这一机制在数组上的运行方式和在对象上的运行方式相同。

使用Module（模块）组织依赖关系

开发任何一款优秀的应用都会面临一项非常困难的工作，那就是找到一种方式把代码组织在合适的功能范围内。我们已经看过控制器的处理方式，它会提供一块空间，把正确的数据模型和函数暴露给视图模板。但是其他那些用来支撑我们应用的代码应该怎么办呢？有一块最明显的可以放这些代码的地方，那就是控制器中的函数。

这种做法对于小型的应用和例子来说可以工作得很好，我们已经看到过很多这样的例子，但是在真实的应用中，这种做法很快就会使代码变得无法维护。控制器将会变成一个垃圾场，我们要做的所有东西都会倒在里面。它们会非常难以理解，并且非常难以修改。

我们来看模块。它们提供了一种方法，可以用来组织应用中一块功能区域的依赖关系；同时还提供了一种机制，可以自动解析依赖关系（又叫做依赖注入）。一般来说，我们把这些叫做依赖服务，因为它们会负责为应用提供特殊的服务。

34

例如，如果购物站点中的一个控制器需要从服务器上获取一个商品列表，那么我们

就需要某些对象——不妨把它叫做 Items——来处理从服务器端获取商品的工作。进而，Items 对象就需要以某种方式与服务器上的数据库进行交互，可以通过 XHR 或者 WebSocket。

如果在没有模块的情况下来做这件事情，那么代码看起来就像下面这样：

```
function ItemsViewController($scope) {
  // 向服务器发起请求
  …
  // 解析响应并放入 Item 对象
  …
  // 把 Items 数组设置到 $scope 上，这样视图才能够显示它
  ...
}
```

虽然这样做确实可以运行，但是却存在大量潜在的问题。

- 如果其他控制器也需要从服务端获取 Items，那么我们只能把这段代码重写一遍。这会给维护工作造成很大负担，因为如果修改了架构或者其他东西，那么我们就必须在很多地方修改同一段代码。
- 加上其他一些因素，例如服务端认证、解析数据的复杂性等，会使得控制器对象很难划分出一个合理的功能边界，并且代码阅读起来更加困难。
- 为了对这段代码进行单元测试，我们需要真正启动一个服务器，或者对 XMLHttpRequest 进行模拟（monkey patch[6]），从而返回一些假数据。真正运行一个服务器会让测试过程变得很慢，配置服务器很痛苦，而且给测试工作带来不可靠性。模拟（monkey patching）路由的方式可以解决运行速度和不可靠测试的问题，但是这意味着，你必须在两次测试之间去掉被模拟对象上的模拟内容，同时它还会带来额外的复杂度和弱点，因为它会强制给你的数据指定一个“线上[7]”形式（并且当这个形式发生变化时需要同时修改测试代码）。

利用模块和模块内置的依赖注入功能，我们就可以把控制器写得更加简单，示例如下：

```
function ShoppingController($scope, Items) {
  $scope.items = Items.query();
}
```

35

现在你可能会问自己，“不错，这样看起来很酷，但是 Items 这个对象是从哪儿来的呢？”以上代码假设我们已经把 Items 对象定义成了一个服务。

服务都是单例（单个实例）的对象，它们用来执行必要的任务，支撑应用的功能。

译注 6：这里意思是，在运行时拦截请求，不会真正去请求后台，而是用前台 JS 代码来模拟返回的数据。更详细的描述请参见 *http://en.wikipedia.org/wiki/Monkey_patch*。

译注 7：也就是说，针对原来的数据格式，重新定义一种格式，专门用来进行测试工作。

Angular 内置了很多服务，例如 $location 服务，用来和浏览器的地址栏进行交互；$route 服务，用来根据 URL 地址的变化切换视图；还有 $http 服务，用来和服务器进行交互。

你可以并且也应该去创建自己的服务，用它们来实现你的应用所特有的功能。如果需要，服务可以在任何控制器之间进行共享。因此，当你需要在多个控制器之间进行交互和共享状态时，这些服务就是一种很好的机制。Angular 内置的服务以 $ 符号打头，你当然可以给你的服务随意起名字，但是，最好不要以 $ 打头，以免引起命名冲突。

你可以使用模型对象的 API 来定义服务。以下 3 个函数可以用来创建一般的服务，它们的复杂度和功能不同。

函数	定义
provider(name, Object OR constructor())	一个可配置的服务，创建的逻辑比较复杂。如果你传递了一个 Object 作为参数，那么这个 Object 对象必须带有一个名为 $get 的函数，这个函数需要返回服务的名称。否则，Angular 会认为你传递的是一个构造函数，调用构造函数会返回服务实例对象。
factory(name,$get Function())	一个不可配置的服务，创建逻辑比较复杂。你需要指定一个函数，当调用这个函数的时候，会返回服务的实例。你可以把它看成 provider(name, { $get: $getFunction() }) 的形式。
service(name, con structor())	一个不可配置的服务，创建逻辑比较简单。与上面 provider 函数的 constructor 参数类似，Angular 调用它可以创建服务实例。

后面我们会解释 provider() 函数的配置项，我们先来讨论一个针对 Item 的使用 factory() 的例子。我们可以像下面这样来编写服务：

```
// 创建一个模型用来支撑我们的购物视图
var shoppingModule = angular.module('ShoppingModule', []);

// 设置好服务工厂，用来创建我们的 Items 接口，以便访问服务端数据库
shoppingModule.factory('Items', function() {
  var items = {};
  items.query = function() {
    // 在真实的应用中，我们会从服务端拉取这块数据 ...
    return [
      {title: 'Paint pots', description: 'Pots full of paint', price: 3.95},
      {title: 'Polka dots', description: 'Dots with polka', price: 2.95},
      {title: 'Pebbles', description: 'Just little rocks', price: 6.95}
    ];
  };
  return items;
});
```

36 当 Angular 创建 ShoppingController 时，它会把 $scope 对象和刚定义的 Items 服务作为参数传递进去。这一点是通过参数名匹配来实现的，也就是说，Angular 会查看我们的 ShoppingController 类的函数签名，然后就会发现它需要一个 Items 对象。既然我们已经把 Items 定义成了一个服务，那么 Angular 当然知道去哪里找这个服务了。

以字符串的形式查找这些依赖关系的结果是，可以进行注入的那些函数（例如控制器的构造器）的参数是没有顺序的。所以，除了下面这种写法之外：

```
function ShoppingController($scope, Items) {...}
```

你还可以这样写：

```
function ShoppingController(Items, $scope) {...}
```

这种写法同样能够按照我们所期望的方式运行。

为了让这一机制能够和模板配合起来，我们需要把模块名称告诉 ng-app 指令，示例如下：

```
<html ng-app='ShoppingModule'>
```

为了完成这个例子，我们可以把模板的其他部分实现如下：

```
<body ng-controller="ShoppingController">
  <h1>Shop!</h1>
  <table>
      <td>{{item.title}}</td>
      <td>{{item.description}}</td>
      <td>{{item.price | currency}}</td>
    </tr>
  </table>
</div>
```

应用最终完成之后的效果如图 2-2 所示。

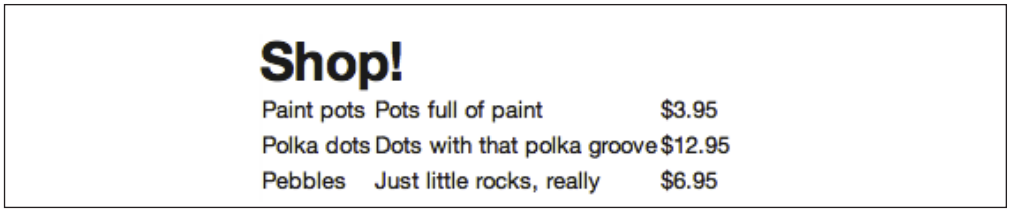


图2-2: 商品

我需要多少个模块呢

服务自身可以互相依赖，类似地，你也可以使用 Module 接口来定义模块之间的依赖关系。

在大多数应用中，创建供所有代码使用的单个模块，并把所有依赖的东西都放入这个模块中，这样就能工作得很好。但是，如果你打算使用第三方包中所提供的服务或者指令，它们一般会带有自己的模块。由于你的应用会依赖这些模块，所以需要在应用模块中定义依赖关系才能引用它们。

例如，如果你引入了（假想的）模块 SnazzyUIWidgets 和 SuperDataSync，那么应用的模块声明看起来可能会像下面这样：

```
var appMod = angular.module('app', ['SnazzyUIWidgets', 'SuperDataSync']);
```

使用过滤器格式化数据

你可以用过滤器来声明应该如何变换数据格式，然后再显示给用户，你只要在模板中使用一个插值变量即可。使用过滤器的语法是：

```
{{ expression | filterName : parameter1 : ...parameterN }}
```

这里的表达式可以是任意的 Angular 表达式，`filertName` 是你需要使用的过滤器名称，过滤器的多个参数之间使用冒号分隔。这些参数自身也可以是任何合法的 Angular 表达式。

Angular 内置了很多过滤器，例如我们在前面见过的 currency：

```
{{12.9 | currency}}
```

以上这段代码将会显示如下结果：

```
$12.90
```

我们把插值变量声明在视图里面（而不是在控制器或者模型里面），因为数字前面的美元符号对人类来说很重要，而对我们用来处理数值的逻辑来说是没有意义的。

Angular 内置的其他过滤器还有 date、number、uppercase 等。

在绑定的过程中，可以用管道符号把过滤器连接起来。例如，对于上面这个例子，我们可以加一个 number 过滤器来把小数点后面的数值格式化掉，number 过滤器会把数值作为参数传递给 round 函数。所以：

```
{{12.9 | currency | number:0 }}
```

将会显示成：

```
$13
```

你不必受限于内置的过滤器，自己编写过滤器也非常简单。例如，如果我们需要为标题

文字创建首字母大写的字符串，可以像下面这样做：

```
var homeModule = angular.module('HomeModule', []);
homeModule.filter('titleCase', function() {
  var titleCaseFilter = function(input) {
    var words = input.split(' ');
    for (var i = 0; i < words.length; i++) {
      words[i] = words[i].charAt(0).toUpperCase() + words[i].slice(1);
    }
    return words.join(' ');
  };
  return titleCaseFilter;
});
```


对应的模板如下：

```
<body ng-app='HomeModule' ng-controller="HomeController">
  <h1>{{pageHeading | titleCase}}</h1>
</body>
```

然后通过控制器把 pageHeading 作为一个模型变量插入进去：

```
function HomeController($scope) {
  $scope.pageHeading = 'behold the majesty of your page title';
}
```

我们会看到如图 2-3 所示的结果。

图2-3: 首字母大写过滤器

使用路由和$location切换视图

虽然从技术上来说 AJAX 应用确实是单页面应用（即在第一个请求时加载一个 HTML 页面，后面只刷新 DOM 中的局部区域），但是很多时候，出于各种原因，我们需要为用户展示或者隐藏一些子页面视图。

我们可以利用 Angular 的 $route 服务来管理这种场景。可以使用路由服务这样定义：对于浏览器所指向的特定 URL，Angular 将会加载并显示一个模板，并实例化一个控制器来为模板提供内容。

在应用中，你可以通过调用 $routeProvider 服务上的函数来创建路由，把需要创建的路由当成一个配置块传给这些函数即可。创建过程类似下面的伪代码：

```
var someModule = angular.module('someModule', [...module dependencies...])
someModule.config(function($routeProvider) {
  $routeProvider.
    when('url', {controller:aController, templateUrl:'/path/to/tempate'}).
    when(...other mappings for your app...).
    …
    otherwise(...what to do if nothing else matches...);
)};
```

以上代码是说，当浏览器中的 URL 变成指定的取值时，Angular 将会加载 */path/to/template* 路径下的模板，然后把这个模板中的根元素关联到 `aController` 上（因为我们在模板中写了 `ng-controller=aController`）。

39

最后一行中的 `otherwise()` 调用可以告诉路由，在没有匹配到任何东西时跳转到哪里。

我们把这一技术投入应用，构建一款 email 应用，这款应用将会轻松超越 Gmail、Hotmail，以及一切 mail，我们把它叫做 AMail。现在，我们从简单入手。我们会有一个首页视图，里面将会展示邮件列表，其中带有日期、标题和发件人。当点击一份邮件时，将会打开一个新视图来展示邮件内容。

如果你想自己尝试运行以上代码，由于浏览器有安全限制，因此需要把代码放到 web 服务器中，而不是仅仅使用 *file://* 的方式来访问。如果你安装了 python，就可以在工作目录中执行 `python –m SimpleHTTPServer 8888` 来启动服务器。

在主模板中，我们会做一些不同的东西。我们不会把所有东西都放在第一个加载的页面中，而是仅仅创建一个布局模板，然后再用这个模板来容纳各种视图。在主视图中，我们将会把多个视图共有的东西都放在里面，例如菜单。在这个例子中，我们仅仅把应用的标题放在里面，然后再用 ng-view 指令来告诉 Angular 把视图显示在哪。

index.html

```
<html ng-app="AMail">
  <head>
    <script src="src/angular.js"></script>
    <script src="src/controllers.js"></script>
  </head>
  <body>
    <h1>A-Mail</h1>
    <div ng-view></div>
  </body>
</html>
```

由于所有视图模板都会被插入刚才所创建的外壳中，所以可以把它们都编写成 HTML 文档片段。对于邮件列表视图而言，我们将会使用 `ng-repeat` 指令来遍历邮件列表，然后

把它们渲染到一个 table 中。

list.html

```
<table>
  <tr>
    <td><strong>Sender</strong></td>
    <td><strong>Subject</strong></td>
    <td><strong>Date</strong></td>
  </tr>
  <tr ng-repeat='message in messages'>
    <td>{{message.sender}}</td>
    <td><a href='#/view/{{message.id}}'>{{message.subject}}</td>
    <td>{{message.date}}</td>
  </tr>
</table>
```

40

注意这里，我们想实现用户点击一个主题就能被导航到相应的邮件中。我们已经在 URL 和 `message.id` 之间进行了数据绑定，所以用户点击 `id=1` 的邮件就会被导航到 */#/view/1*。这种根据 URL 导航的方式也叫做深度连接，我们将会在邮件详细视图控制器中使用，以实现把一份可用的邮件链接到对应的详细视图上。

为了创建这个邮件详情视图，需要创建一个模板，用来展示单个邮件对象上的属性。

detail.html

```
<div><strong>Subject:</strong> {{message.subject}}</div>
<div><strong>Sender:</strong> {{message.sender}}</div>
<div><strong>Date:</strong> {{message.date}}</div>
<div>
    <strong>To:</strong>
    <span ng-repeat='recipient in message.recipients'>{{recipient}} </span>
<div>{{message.message}}</div>
<a href='#/'>Back to message list</a>
```

为了把这些模板关联到对应控制器上，我们将会给 `$routeProvider` 配置一个 URL，`$routeProvider` 将会负责调用控制器和模板。

controllers.js

```
// 为核心的 AMail 服务创建模块
var aMailServices = angular.module('AMail', []);

// 在 URL、模板和控制器之间建立映射关系
function emailRouteConfig($routeProvider) {
  $routeProvider.
  when('/', {
    controller: ListController,
```

```
    templateUrl: 'list.html'
  }).
// 注意，为了创建详情视图，我们在 id 前面加了一个冒号，从而指定了一个参数化的 URL 组件
  when('/view/:id', {
    controller: DetailController,
    templateUrl: 'detail.html'
  }).
  otherwise({
    redirectTo: '/'
  });
}

// 配置我们的路由，以便 AMail 服务能够找到它
aMailServices.config(emailRouteConfig);

// 一些虚拟的邮件
messages = [{
  id: 0, sender: 'jean@somecompany.com', subject: 'Hi there, old friend',
  date: 'Dec 7, 2013 12:32:00', recipients: ['greg@somecompany.com'],
  message: 'Hey, we should get together for lunch sometime and catch up.'
  +'There are many things we should collaborate on this year.'
}, {
  id: 1,  sender: 'maria@somecompany.com',
  subject: 'Where did you leave my laptop?',
  date: 'Dec 7, 2013 8:15:12', recipients: ['greg@somecompany.com'],
  message: 'I thought you were going to put it in my desk drawer.'
  +'But it does not seem to be there.'
}, {
  id: 2, sender: 'bill@somecompany.com', subject: 'Lost python',
  date: 'Dec 6, 2013 20:35:02', recipients: ['greg@somecompany.com'],
  message: "Nobody panic, but my pet python is missing from her cage."
  +'She doesn't move too fast, so just call me if you see her."
}, ];

// 把我们的邮件发布给邮件列表模板
function ListController($scope) {
  $scope.messages = messages;
}

// 从路由信息（从 URL 中解析出来的）中获取邮件 id，然后用它找到正确的邮件对象
function DetailController($scope, $routeParams) {
  $scope.message = messages[$routeParams.id];
}
```


我们已经为应用搭好了基本的框架，这款应用带有很多视图，可以通过修改 URL 的方式来切换视图。这就意味着对用户来说前进和后退按钮能够真正运行起来了。用户可以把应用内部的视图链接加入书签或者通过邮件发送出去，而实际上这里只有唯一一个真实的 HTML 页面。

与服务器交互

真正的应用需要和真实的服务器进行交互，移动应用和新兴的 Chrome 桌面应用可能是

个例外，但是对于此外的所有应用来说，无论你是想把数据持久化到云端，还是需要与其他用户进行实时交互，都需要让应用与服务器进行交互。

为了实现这一点，Angular 提供了一个叫做 `$http` 的服务。它提供了一个可扩展的抽象
42 方法列表，使得与服务器的交互更加容易。它支持 HTTP、JSONP 和 CORS 方式。它还包含了安全性支持，避免 JSON 格式的脆弱性和 XSRF[8]。它让你可以轻松地转换请求和响应数据，甚至还实现了简单的缓存。

例如，我们打算让购物站点从服务器上获取商品信息，而不是从内存假数据获取。如何编写服务端代码已经超越了本书的范畴，所以，我们仅仅来想象一下，比方说我们已经创建了一个服务器，当查询 */products* 路径时，它会以 JSON 格式返回一个商品列表。

返回的响应示例如下：

```
[
  {
    "id": 0,
    "title": "Paint pots",
    "description": "Pots full of paint",
    "price": 3.95
  },
  {
    "id": 1,
    "title": "Polka dots",
    "description": "Dots with that polka groove",
    "price": 12.95
  },
  {
    "id": 2,
    "title": "Pebbles",
    "description": "Just little rocks, really",
    "price": 6.95
  }
  ...etc...
]
```

我们可以像下面这样编写查询代码：

```
function ShoppingController($scope, $http) {
  $http.get('/products').success(function(data, status, headers, config) {
    $scope.items = data;
  });
}
```

然后在模板中这样使用它：

```
<body ng-controller="ShoppingController">
    <h1>Shop!</h1>
```

译注 8：　跨站请求伪造。

```
      <table>
        <tr ng-repeat="item in items">
          <td>{{item.title}}</td>
          <td>{{item.description}}</td>
          <td>{{item.price | currency}}</td>
        </tr>
      </table>
    </div>
  </body>
```


正如我们前面讲过的，从长远来看，让服务来代理与服务器交互的工作对我们有好处，这个服务可以被多个控制器共享。在本书第 5 章中将会讲述这种结构以及 `$http` 服务的各种功能。

使用指令修改DOM

指令扩展了 HTML 语法，同时它也是使用自定义的元素和属性把行为和 DOM 转换关联到一起的方式。通过这些指令，你可以创建可复用的 UI 组件，配置你的应用，并且可以做到你能想象到的几乎所有事情，这些事情都可以在你的 UI 模板中实现。

你可以使用 Angular 内置的指令来编写应用，但是也难免会遇到需要自己编写指令的情况。当你想要以某种方式处理浏览器时间或者修改 DOM，而这些东西又没有内置指令可以支持时，你就会知道是时候深入理解指令体系了。你编写的代码将会位于你所编写的指令中，而不是控制器、服务或应用的其他地方。

与服务一样，你可以通过模块对象的 API 来定义指令，只要调用模块实例的 `directive ()` 函数即可，其中 `directiveFunction` 是一个工厂函数，用来定义指令的特性。

```
var appModule = angular.module('appModule', [...]);
appModule.directive('directiveName', directiveFunction);
```

编写指令工厂函数是一个非常深的领域，我们在本书中专门开辟了一整章的内容来讨论这一话题。为了充分调动你的兴趣，我们先来看一个简单的例子。

HTML5 中带有一个很棒的新属性，叫做 autofocus，它可以把键盘的焦点定位到一个输入元素中。通过它，用户能够立即通过键盘和元素进行交互，而不必先点击元素。这一点很棒，因为它让你可以通过声明的方式来描述需要浏览器做什么，而没有必要编写任何 JavaScript 代码。但是，如果你想把焦点放到一些非交互型元素上，例如超链接或者 `div`，应该怎么做呢？如果你想让这种机制在不支持 HTML5 的浏览器中也能工作，又应该怎么做呢？我们可以通过一个指令来实现。

```
var appModule = angular.module('app', []);
```

```
appModule.directive('ngbkFocus', function() {
  return {
    link: function(scope, element, attrs, controller) {
      element[0].focus();
    }
  };
});
```

44 在上面的代码中，我们返回了指令的配置对象，其中带有特定的 link 函数。link 函数会获取外层 scope 的引用、它所存在的 DOM 元素、传递给指令的一个属性数组，以及 DOM 元素上的控制器——如果有控制器存在话。这里，我们只需要获取元素，然后调用它的 `focus()` 方法即可。

然后我们就可以把它用到例子中，如下。

index.html

```
<html lang='en' ng-app='app'>
...include angular and other scripts...
<body ng-controller="SomeController">
  <button ng-click="clickUnfocused()">
    Not focused
  </button>
  <button ngbk-focus ng-click="clickFocused()" >
    I'm very focused!
  </button>
  <div>{{message.text}}</div>
</body>
</html>
```

controllers.js

```
function SomeController($scope) {
  $scope.message = { text: 'nothing clicked yet' };

  $scope.clickUnfocused = function() {
    $scope.message.text = 'unfocused button clicked';
  };

  $scope.clickFocused = function {
    $scope.message.text = 'focus button clicked';
  }
}
var appModule = angular.module('app', ['directives']);
```

当页面加载完成之后，用户将会看到一个按钮，其标签为“I’m very focused!”，它获得了高亮焦点。按下空格键或者回车键将会导致一次点击动作，并且会调用 `ng-click`，它将会把 div 上的文本设置为‘focus button clicked’。在浏览器中打开这个例子，我们将会看到如图 2-4 所示的结果。

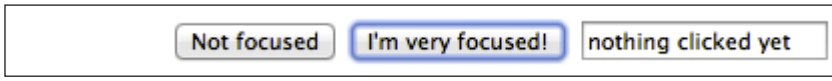


图2-4：焦点指令

校验用户输入

Angular 自动为 <form> 元素增加了一些很好用的特性，使其更适合开发单页面应用。其中一个特性是，Angular 允许你为表单中的输入元素定义一个合法的状态，并且只有当所有元素都是合法状态时才允许提交表单。

例如，创建一个新用户注册表单，我们要求必须输入用户名和邮件地址，但是年龄字段是可选的，在这些字段被提交到服务器之前我们可以校验这几个输入项。把下面的例子加载到浏览器中，然后会显示如图 2-5 所示的内容。

图2-5：表单校验

我们想要确保用户在两个名称字段中都输入了值，并且输入了一个格式合法的邮件地址。另外如果他输入了年龄，那也是合法的。

利用 Angular 的 <form> 扩展以及大量的输入元素，我们可以在模板中实现以上效果，示例如下：

```
<h1>Sign Up</h1>
<form name='addUserForm'>
  <div>First name: <input ng-model='user.first' required></div>
  <div>Last name: <input ng-model='user.last' required></div>
  <div>Email: <input type='email' ng-model='user.email' required></div>
  <div>Age: <input type='number'
                   ng-model='user.age'
                   ng-maxlength='3'
                   ng-minlength='1'></div>
  <div><button>Submit</button></div>
</form>
```

注意，我们使用了 HTML5 的 required 属性以及 email 和 number 两种输入类型，用来校验其中的一些字段。这一机制在 Angular 中工作得很好，而对于那些老式的不支持 HTML5 的浏览器来说，Angular 将会自动使用指令来适配，从而实现相同的功能。

然后就可以添加一个控制器来处理提交功能了，只要修改 form 来引用控制器即可。

46
```
<form name='addUserForm' ng-controller="AddUserController">
```

在控制器中，我们可以通过 `$valid` 属性获取表单的校验状态。当表单中的所有输入项都合法时，Angular 将会把这个属性设置为 true。我们可以使用 `$valid` 属性来做很多很炫的事情，例如当表单没有输入完成时可以禁用 Submit 按钮。

我们可以在 Submit 按钮上添加一个 `ng-disabled` 指令，从而避免提交非法状态的表单：

```
<button ng-disabled='!addUserForm.$valid'>Submit</button>
```

最后，我们想让控制器告诉用户，他已经注册成功了。最终模板如下：

```
<h1>Sign Up</h1>
<form name='addUserForm' ng-controller="AddUserController">
  <div ng-show='message'>{{message}}</div>
  <div>First name: <input name='firstName' ng-model='user.first' required></div>
  <div>Last name: <input ng-model='user.last' required></div>
  <div>Email: <input type='email' ng-model='user.email' required></div>
  <div>Age: <input type='number'
                   ng-model='user.age'
                   ng-maxlength='3'
                   ng-min='1'></div>
  <div><button ng-click='addUser()'
               ng-disabled='!addUserForm.$valid'>Submit</button>
</ng-form>
```

控制器代码如下：

```
function AddUserController($scope) {
  $scope.message = '';

  $scope.addUser = function () {
    // 由读者自己实现：把 user 真正保存到数据库中 ...
    $scope.message = 'Thanks, ' + $scope.user.first + ', we added you!';
  };
}
```

继续前进

在前两章中，我们已经看过了 Angular 框架的几乎所有常用特性。对于讨论过的每一种特性，还有很多额外的细节未涉及。在下一章中，我们将会分析一个典型的开发过程，从而进一步深入学习。