Less Code, More Fun, and Enhanced Productivity with Structured Web Apps

AngularJS

O'REILLY®

Brad Green & Shyam Seshadri

序

本人**懒虫一个**，目前在一家传统软件行业工作，闲来无事，特翻译 O'Reilly《AngularJS》一书，供广大网友阅读。经过本人的通读与赏阅，本书适合 AngularJS 初学者阅读，了解及运用 AngularJS 进行开发，本书中涉及到了 AngularJS 大量的基本知识和核心概念，通过阅读本书后，相信你已经具备基本的 AngularJS 开发经验。

本人尊重原作的著作权，尊重原稿，本译文暂称为《AngularJS 中文》。

原本书中有不少错误，本人以批注的方式提醒并做了改正。翻译过程中难免有疏漏或错译之处，希望广大网友批评指出、提出建议。

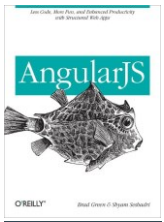
特此，献给广大 IT 网友

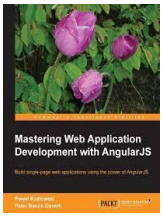
其他资源

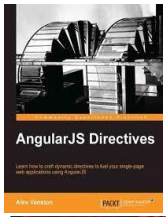
1. 本书源代码：
https://github.com/shyamseshadri/angularjs-book

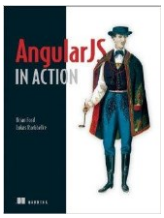
2. Github
https://github.com/jmcunningham/AngularJS-Learning

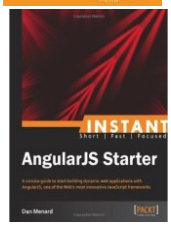
3. 书籍











《AngularJS》就不多介绍了
《Mastering Web Application Development with AngularJS》也是一本不错的书，建议看看，据说有团队在翻译了，期待 ing

目　录

第一章：AngularJS 简介

我们创造基于 WEB 应用程序的惊人能力是令人难以置信，然而构造这些应用程序的复杂性也同样令人难以置信。Angular 团队想减轻使用 AJAX 开发应用程序的痛苦。在 Google，我们曾经经历过构建像 Gmail，Maps，Calendar 和其他大型 WEB 应用程序的沉痛教训。我们认为我们有能力利用这些经验，让每个人受益。

我们希望编写 WEB 应用程序就像我们第一次写几行代码，然后对我们所做的愣住了。我们希望编程的过程就像在创造，而不是尝试着去满足 WEB 浏览器各种奇怪的内部机制。

同时，我们希望有一个环境，能够帮助我们决策，使应用程序容易创建，并从一开始就能够理解，随着项目的变大，能够继续正确的策略使应用程序容易测试、扩展和维护。

我们已经在 Angular 框架中尝试做到了这点。对于我们达到的成就十分兴奋。许多功劳归功于开源社区，Angular 周围的人在互相支持上做了出色的工作，教会了我们很多东西。我们希望你能够加入我们社区，帮助我们了解如何让 Angular 更好。

一些更大和更多相关的例子和代码片段在 GitHub 仓库，你可以查看、做个分支，并在我们的 GitHub 页面上玩弄。

概念

在整个 Angular 应用，你会用到几个核心理念。事实证明，我们没有创造任何东西。相反，我们大量借鉴了其他开发环境中的成功语录，并通过拥抱 HTML，浏览器和其他常见的 WEB 标准的方法实现了它们。

客户端模板

多页面 WEB 应用程序通过组装和拼接服务器上的数据来创造它们的 HTML，然后将完成的页面输送给浏览器。某种程度上，绝大部分单页面应用（也被称为 AJAX 应用）同样做到了。Angular 不同于它的是，传递模板和数据到浏览器，然后在浏览器端组装。服务器的角色变成了只提供模板的静态资源和提供模板所需要的数据。

让我们来看一个例子，在浏览器端如何用 Angular 组装数据和模板。我们举个 Hello world 的例子，而不是输出 Hello world 字符换，然后组织可以改变的“hello”。

因此，在 hello.html 建立了模板：

```
<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>
```

浏览器加载 hello.html 页面，产生如图 1-1 所示

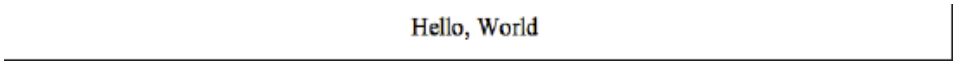


图 1-1 Hello World

和当前使用的绝大多数方法相比，需要注意一些有趣的东西：

- 在 HTML 中没有 classes 或者 IDs 来区分绑定的事件监听器。
- 当 HelloController 设置 greeting.text 为 hello 时，我们并没有注册任何事件监听器和写任何回调函数。
- HelloController 是一个普通的 JavaScript 类，并没有继承来 Angular 提供的任何东西。
- HelloController 获取它需要的$scope 对象，而没有必要创建它。
- 我们没有必要调用 HelloController 的构造函数，或者计算出何时调用他们。

很快，我们将会看到更多的不同，但是有一点很清晰的是，Angular 应用组织结构非常不同于以前类似的应用。

为什么指定这种决策以及 Angular 如何运行的？让我们看看一些 Angular 从别的地方偷来的建议。

模型 视图 控制器(MVC)

在 20 世纪 70 年代引入了 MVC 应用结构作为 Smalltalk 的一部分。从 Smalltalk 开始，MVC 在涉及用户界面的桌面开发环境中变的很流行。无论你使用 C++，java，Object-c，都有有 MVC 的风格。然而，直到最近几年，MVC 才是 WEB 开发的全部。

MVC 背后的核心理念就是，你在你的代码之间明确分离管理数据（模型），应用程序逻辑（控制器），并将数据给用户（视图）。

视图从模型中获取数据展示给用户。当用户通过点击或者输入和应用程序进行交互时，控制器通过改变模型中的数据响应。最终，模型层通知视图层，已经发生改变，一边更新显示。

在 Angular 应用中，视图层就是 DOM，控制器就 JavaScript 类，模型数据存储在对象属性中。

我们认为，MVC 是整齐的几个原因。首先，它更够给你一个内心的模型用来存放什么，因此你不需要重复造轮子。你们项目中的其他人，由于知道使用了 MVC 结构组织代码，因此很容易的理解你所写的。可能最重要的是，带来了巨大的利益，是有应用程序更容易扩展、

维护、和测试。

数据绑定

在 AJAX 之前，单页面应用是常见的，想 Rails，PHP，JSP 等平台通过合并 HTML 字符串和发送给用户展示的数据来创建用户界面。

像 jQuery 库这种扩展到客户端的模式，让我们遵循相似的风格，但由于有更新的能力，单独地 DOM 的部分，而不是更新整个页面。这里，我们合并 HTML 字符串和数据，然后通过元素上设置 innerhtml 将结果插入到我们所想要的 DOM 中。

这一切都运行的相当好，但是当你想将新数据插入到界面，或者改变基于用户输入的数据时，你需要做很多又不是价值不高的工作，来确保同时在界面和 JavaScript 属性中获取的数据是正确的状态。

但是，倘若我们有什么东西把这些工作都为我们做好了，同时不需要写代码？倘若我能仅仅声明界面的某一部分映射到 JavaScript 的属性，让他们自动的同步？这种编程方式叫做数据绑定。我们在 Angular 中包括这种功能，因为当编写视图和模型时，使用 MVC 来消除代码那是非常棒的。移动数据从一个地方到另外一个地方的绝大部分工作是自动发生的。

为了要看到这种效果，让我们做第一个例子，并让他动起来。之前，Hello Controllery 一开始设置了 greeting.text 模型，之后就没有改变过。为了让它‘活起来’，让我们增加一个文本输入框改变示例，随着用户的输入改变 greeting.text 的值。

这里是新的模板：

```
<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 HellControler，可以完全相同。

浏览器加载后，将看到图 1-2 中的截图：

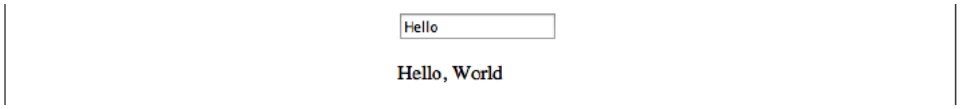


图 1-2 greeting 应用的默认状态。

如果我们用 Hi 取代输入框中的 Hello，将看到图 1-3 中的截图：

图 1-3 Greeting 应用输入框改变后。

永远不需要在输入字段上注册一个改变监听器，我们就有一个可以动态更新的界面。对于服务器来回的变化同样是可以的。在控制器里，我们可以向服务器发起一个请求，获取相应，设置到$scope.greeting.text 中。Angular 可以自动更新输入框和大括号中的值。

依赖注入（DI）

之前我们提及到的，但是值得重复 HellController 中有许多我们没写。例如，$scope 对象把数据绑定自动的传递了给我们。我们没有必要通过调用任何函数来创建它。我们只是要求把它放到 HelloController 构造函数中。

在后续的章节中，我们将会发现，$scope 并不是我们唯一需要的。如果我们需要数据绑定到用户浏览器指定的 URL 地址中，我们需要在构造函数中，添加一个$loaction 对象，就像这样：

```
function HelloController($scope, $location) {
$scope.greeting = { text: 'Hello' };
// use $location for something good here...
}
```

通过 Angular 的依赖注入系统，我们可以得到这种效果。依赖注入允许我们遵循一种开发风格，这种开发风格中，不是创建依赖，我们的类仅仅添加他们需要的。

这个遵循了一个叫**迪米特法则**的设计模式，也被称作**最少知识法则**。由于 HelloController 的任务是建立 greeting 模型的的初始值，这种模式就是说，它不需要担心像$scope 如何创建以及在哪里找到它。

这种特性不仅仅是 Angular 框架为了创建对象而有的。你同样可以写完这些代码的剩余部分。

指令

Angular 最优秀部分之一是你可以把你写的模板当成 HTML。因为在框架的核心层，我们已经包括了一个强大的 DOM 转换引擎，可以让你**扩展 HTML 语法**，因此你才可以这样做。

我们已经在我模板文件中看到了多个新的属性，这些并不是 HTML 规范的一部分。示例中包括两个大括号是用用来数据绑定的，ng-controller 是用来指定哪个控制器来服务哪个视图，ng-model 将一个输入框绑定到模型部分。**我们称这些叫 HTML 扩展指令**。

Angular 带有很多标识符，帮助你为你的应用程序定义视图。很快我们将看到更多。这些标识符可以定义我们常见的视图作为模板。它们可以说明应用程序如何工作的或者创建可重复使用的组件。

同时不局限域 Angular 自带的标识符。你可以写你自己的来扩展 HTML 模板，做任何你想做的事。

购物车示例

让我们来看一个稍微大点的示例，顺便炫耀下 Angular 更多的特性。假想我们将构建一个购物应用。应用中的某个地方，我们需要展示用户的购物车，同时可以编辑它。然我们直接跳到那部分。

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <base/>
        <title>Your Shopping Cart</title>
        <script src="../frm/angular/angular.js"></script>
    </head>
    <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($index)"> Remove </button>
        </div>
        <script>
            function CartController($scope) {
                ////@formatter:off
                $scope.items = [{
                    title : 'Paint pots',
                    quantity : 8,
                    price : 3.95
                }, {
                    title : 'Polka dots',
                    quantity : 17,
                    price : 12.95
                }, {
                    title : 'Pebbles',
                    quantity : 5,
```

批注 [spy1]: 原文有错误，现已改正

```
                price : 6.95
            }];
            ////@formatter:on
            $scope.remove = function(index) {
                $scope.items.splice(index, 1);
            };
        }
    </script>
</body>
</html>
```

下面是一段上述的简介。这本书的剩余部分专门进行了深入的解释。

首先从最上面开始：

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

ng-app 属性告诉 Angular 它应该管理页面的哪一部分。由于我们把它放在 html 元素上，告知 Angular 管理整个页面。这个常常是你想要的，但是如果你正在集成 Angular 和一个已存在的使用其他方式管理页面的应用，那么你可能需要放在应用的 div 上

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

在 Angular 中，用 JavaScript 类管理的页面区域叫做控制器。通过在 body 标签中包含一个控制器，声明的 CartController 将管理 body 标签之间的任何东西。

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

ng-repeat 代表为 items 数组中每个元素拷贝一次这 div 中的 DOM。在 div 每次拷贝中，同时设置了一个叫 item 的属性代表当前的元素，所以我们能够在模板中使用。正如你看到的，每个 div 中都包含了产品名称，数量，单价，总价和一个移除按钮。

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

正如演示的‘Hello World’示例，数据绑定是通过{{ }}把变量的值插入到页面某部分和保持同步。完整的表达式{{item.title}}检索迭代中的当前项，然后将当前项的 titile 属性值插入到 DOM 中。

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

ng-model 定义创建了输入字段和 item.quantity 之间的数据绑定。

span 标签中的{{ }}建立了一个单向联系，在这里插入值。但是应用程序需要知道当用户改变数量时，能够改变总价，这是我们想要的效果。

通过使用 ng-model 我们将与我们的模型保持同步更改。ng-model 申明将 item.quantity 的值插入到输入框中，无论何时用户输入一个新值将自动更新 item.quantity。

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

我们希望单价和总价格式化成美元形式。Angular 带有一个叫过滤器的特性，能够让我们转换文本，有一个叫 currency 的过滤器将为我们做这个美元形式格式化。在下面章节将有

更多的关于过滤介绍。

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

这个允许用户点击产品旁边的Remove按钮从购物车中移除该项。我们已经建立了联系，以便点击这个按钮就可以调用 remove()函数。同时传递了$index，这个包含了 ng-repeat 的迭代顺序，以便知道要移除哪一项。

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

CartController 管理这购物车的逻辑。通过这个我们告知 Angular 控制器需要一个叫$scope 的对象。$scope 就是用来把数据绑定到界面上的元素的。

```
$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$scope.items，我们已经创建一个虚拟数据代表了用户购物车中物品集合。我们希望让这些数据能和界面绑定，因此我们把他们增加到$scope 中。

当然，这样的购物车是不能仅工作在内存中，也需要通知服务器端持久化数据。我们在后面的章节中做这个。

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

我们希望 remove()函数能够绑定到界面上，因此我们也把它增加到$scope 中。对于这个内存中的购物车版本，remove()函数只是从数组中删除了它们。因为通过 ng-repeat 创建的这些 是数据捆绑的，当某项消失时，列表自动收缩。记住，无论何时用户点击移除按钮中的一个，都将从界面上调用 remove()函数。

后续

我们只是看到了 Angular 最基本的用法和一些非常简单的示例。书的剩余部分介绍了框架提供了些什么功能。

第二章：AngularJS 应用剖析

不像典型的库按照你喜欢的筛选函数，Angular 中的一切都是被设计用于合作的套件。在本章中，我们将覆盖 Angular 中所有基本构建块，以便你能够理解他们是如何组合的。这块的很多部分将在后续章节中详细介绍。

调用 Angular

任何应用使用 Angular 必须做两件事：

1. 加载 angular.js
2. 使用 ng-app 告知 Angular 管理哪一部分的 DOM

加载脚本

加载库很简单，像其他 JavaScript 库一样遵循相同的规则。可以从 Google 的 CDN 上加载脚本，就像这样：

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

推荐使用 Google 的 CDN。Google 的服务器是非常快的，脚本是跨应用缓存的。那就是说，如果你的用户有多个使用 Angular 的应用，它只下载一次。同样，如果用户访问过使用 Google Angular 的 CDN 链接，那么当他访问你的站点时没有必要再次下载。

如果你喜欢存储在本地（或者其他地方），你也可以这么做。只是在 src 中指定正确的路径。

用 ng-app 声明 Angular 范围

ng-app 标识符用来告诉 Angular 应该管理页面上的哪一部分。如果你构建的是一整套的 Angular 应用，那么你应该将 ng-app 作为<html>标签的一部分，就像这样：

```
<html ng-app>
…
</html>
```

这个告知 Angular 管理页面中所有的 DOM 元素。

如果你已经有一个现有的应用，这些应用中期望使用像 Java ， Rails 这样的技术来管理 DOM。你可以把它（ng-app）放在某个元素上（像页面上的 元素）告知 Angular 来管理

页面的某一部分。

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

模型 视图 控制器(MVC)

在第一章，我们曾经提到 Angular 支持 MVC 的应用程序设计风格。尽管你在设计 Angular 应用时有很多灵活性，但是你将会发现更多：

- 模型中包含了代表应用当前状态的数据
- 视图显示了这些数据
- 控制器管理这模型和视图的关系

使用对象属性，或者只是原型包含的数据来创建模型。模型层变量没有什么特别的。如果想向用户展示文本，你可以设置成字符串，就想这样：

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

你可以通过写一个模板作为 HTML 页面，创建视图层，然后用模型中的数据合并它。正如我们看到的，你可以在 DOM 中插入一个占位符，设置它的内容，就像这样：

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

我们称这个为**二次解析赋值（doule-curly syntax interpolation）**，因为它把新的内容值插入到一个已存在的模板中。

控制器是一些类或者是你写的类型告知 Angular 哪个对象或者原型通过将他们指定到 $scope 对象传递到控制器填充模型

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

结合起来，就是：

```
<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>
```

浏览器中将展示：

```
You have started your journey.
```

尽管原始风格模型在简单场景中可以运行，对于绝大多数应用，需要创建一个模型对象包含数据。创建一个 messages 模型对象，用它来存储 someText。因此而不是：

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

应该这样：

```
var messages = {};
messages.someText = 'You have started your journey.';
function TextController($scope) {
$scope.messages = messages;
}
```

模板中这样使用：

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

当我们讨论完$scope 对象之后，我们就知道，创建像这样的一个模型对象可以防止在$scope 对象中原型继承引起非预期的行为。

然而我们正在讨论的方法从长远看是拯救了你，在之前的例子中，我们在全局的 scope 中创建了 TextController。虽然这是一个很好的例子，但是正确的方法是创建一个控制器作为某一部分被称作**模块**，模块为引用中相关的部分提供了一个命名空间。更新后的代码如下：

```
<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>
```

在这个模板中，我们告知 ng-app 元素我们的模块名，myApp。然后我们调用了 Angular 对象创建一个名为 myApp 的模块，传递了控制器函数给模块的控制器函数。

We'll get to all the whys and hows of modules in a bit.现在，只要记住，远离全局命名空间是一件好事。模块化这是我们通用的机制。

模板和数据绑定

Angular 应用中的模板只是那些我们从服务器加载的 HTML 文档或者是定义在<script>标签中的一些静态资源。你在模板中定义界面，在界面组件中使用标准的 HTML 加上 Angular 标识符。

一旦在 WEB 浏览器中，Angular 通过合并模板和数据扩展这些模板到整个应用。之前当我们显示购物车中的物品列表时，在第一章看过这样的例子

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

这里，除了 外部，里面的所有内容，数组中的每个元素运行一次。

那么这些数据来自哪里？在我们购物车示例中，我们只是以代码的形式定义了一个数组。当你开始构建 UI 和只是想测试它如何运作这将能够运作的非常好。然而大多数应用会在服务器端使用一些持久化数据的代码。你的应用在浏览器中连接到服务端，为用户请求它需要的任何内容，Angular 将它和模板合并。

基本的启动流程就像这样：

1. 用户请求应用的第一页面。
2. 用户的浏览器发出一个 HTTP 链接到你服务器，加载包含模板的 index.html 页面。
3. Angular 加载到页面，等待页面完全加载完成，然后寻找 ng-app 定义模板的边界。
4. Angular 经过模板寻找标识符和捆绑。这样的结果是监听器和 DOM 操作完成了注册，同时从服务器查询初始化数据。这块工作的最终结果是应用完成了自举（启动完成，计算

机专业用语），就像 DOM 一样将模板转换成是视图。

5. 你连接到服务器按需加载你额外需要展示给用户的数据。

步骤 1 至 3 是每个 Angular 应用标准的步骤。步骤 4 和 5，你可以做选择。这些步骤可以同步或异步发生。为了性能，应用需要展示给用户的一个视图数据可以随 HTML 模板一起加载，可以避免多次请求。

通过使用 Angular 结构化你的应用，应用程序的模板和填充它们的数据是分离开的。这样的好处就是这些模板现在可缓存了。在第一次加载后，只有新数据加载到浏览器。只有 JavaScript，图片，CSS 和其他资源，缓存这些模板可以给应用更好的性能。

显示文本

你可以在界面中使用 ng-bind 标识符显示和更新文本。它有两种等价的形式。之前我们看到过使用两个大括号的形式：

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

另外一个基于属性标识符的叫做 ng-bind:

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

以上两个在输出上是等价的。如果模型变量 greeting 的值是‘Hi there,’，Angular 生成 HTML：

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

浏览器上会显示‘Hi there’。

那么，你为什么会使用一种形式，而不是另外一种。我们创造了双括号插补语法，阅读起来更加自然，需要更少的输入。然而两种形式产生等价的输出，使用双括号语法，在应用的 index.html 每次页面加载上，在 Angular 用数据替换大括号之前，用户是有可能看到未渲染的模板。后面的视图就不会遇到这样的情况。

原因是这样的，浏览器加载 HTML 页面，渲染，最后 Angular 才能按照你的意图解析它。

好消息是你仍可以在大多数模板中使用{{ }}。然而，为了在 index.html 页面做数据绑定，用 ng-bind 代替之。这样一来，直到数据加载用户才能看到内容。

表单输入

在 Angular 中使用表单元素很简单的。正如我们在几个示例中所看到的，你可以使用 ng-model 属性来绑定元素到模型的属性上。这个能够和像输入框，单选按钮，复选框等等所有标准的元素正常工作。我们可以像这样绑定一个复选框到属性：

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

这意味着：

1. 如果选中它，SomeController 中的$scope 一个叫 youCheckedIt 属性值会变成 true。不选中的话 youCheckedIt 就是 false。
2. 如果你在 SomeController 中设置$scope.youCheckedIt 为 true，那么界面上就会选中复选框。相反则不选中。

现在让我们讨论下当用户操作时我们实际上想要做的。对于输入框，用 ng-change 属性来指定一个控制器的方法，当用户输入的值发生变化时将被调用。让我们做一个简单的计算示例，来帮助创业者了解他们需要多少资金支持。

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

对于我们这个单纯的例子，我们只是设置了输出是用户的价值的 10 倍。同时我们设置了一个默认值 0 作为开始：

```
function StartUpController($scope) {
$scope.funding = { startingEstimate: 0 };
$scope.computeNeeded = function() {
$scope.funding.needed = $scope.funding.startingEstimate * 10;
};
}
```

批注 [spy2]: 原文有错误

然而，处理代码的策略上这里有个潜在的问题。问题是，当用户在输入框中输入时，我们只是重新计算了需要的数量。如果在这种场景下当用户输入时，输入框只是做了更新，这看上去运行的很好。但是倘若其他的输入框也绑定到这个模型的属性该怎么办？倘若当来自服务器的数据，是否会更新呢？

我们可以用$scope 的$watch 函数更新这个字段，无论它是如何被更新的。我们将在章后面详细讨论$wath 函数。简单的说就是你可以调用$watch()，传递一个观察的表达式和一个回调函数，当表达式改变时，将调用回调函数。

既然这样，我们希望监控 funding.startingEstimate，每当它改变时就调用 computeNeeded()；然后我们就可以使用这项技术重写 StartUpController：

```
function StartUpController($scope) {
$scope.funding = { startingEstimate: 0 };
computeNeeded = function() {
$scope.funding.needed = $scope.funding.startingEstimate * 10;
```

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

注意 watch 中的表达式是在引号中的。是的，就是字符串。字符串作为 Angular 表达式执行。表达式能做简单的运算和访问$scope 中的属性。我们将在后面的章节中详细介绍表达式。

你也可以监控函数的返回值，但是监控 funding.StartingEstimate 属性是不起作用的，因为执行结果是 0，初始值也是 0，因此没有变化。

那么，因为每次 funding.startingEstimate 变化时，funding.needed 就会自动更新，所以，我们可以写一个更简单的模板，就像这样：

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

有些场景下，你不希望每次变化后都采取行动。相反，你希望等待直到用户告诉你他已经一切就绪。典型的例子就像正在完成付完款或正在发送聊天消息。

如果你有一个管理这多个输入框的表单，那么你可以在表单时上用 ng-submit 标识符指定一个函数，当表单提交时调用这个函数。我们可以扩展我们之前的示例，让用户通过点击按钮请求计算启动资金。

```
<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.");
};
}
```

当尝试提交表单时，ng-sbumit 标识符也有自动的阻止浏览器发送默认的 POST 动作。

为了处理其他事件场景，比如当你希望提供不提交表单的交互，Angular 提供了一些类似浏览器本地事件属性的事件处理标识符。对于 onclick，你可以使用 ng-click。对于 ondbclick 可以使用 ng-dbclick 等等。

我们能通过扩展一个带归零的重置按钮启动计算示例将其完整的模拟出来。

```
<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;
};
}
```

批注 [spy3]: 个人建议将requestFunding()放到<button ng-click='reqeustFunding()'>Fund my startup!</button>中

非侵入性 JavaScript 一些建议

在某些点上，在你的 JavaScript 开发生涯，有人可能告诉你在 HTML 里应该写“非侵入性的 JavaScript”，那就是使用 click，mousedown 和其他内嵌的事件处理，这可能是一个坏想法。他是对的。

非侵入性的 JavaScript 思想已经解释了很多方面，但是这种编码风格的基本原理大致如下：

1. 不是每个人的浏览器都支持 JavaScript。让每个人看到你所有的内容，并使用你的应用，而不需要再浏览器中执行代码。
2. 一些使用浏览的人从事着不同地工作。那些使用屏幕阅读器和移动电话的视障认识就不能使用带 JavaScript 的站点。
3. JavaScript 在不同的平台上运行有差异。IE 通常是罪魁祸首。你需要根据浏览器放入到不同的事件处理代码中。
4. 这些时间处理引用函数在全局命名空间下。当和其他库中的相同函数名集成时，煞是令人头痛。
5. 这些事件处理程序把结构和行为组织在一起。这导致你的代码更加难以维护、扩展和理解。

在大多数情形下，以这种方式书写的 JavaScript 生活的更好，但是有一件事就是代码的复杂度和可读性没有更好。而不是声明你的事件处理程序和他们的元素之前见的动作，通常你给这些元素赋予 ID 属性，拿到这个元素的引用，建立带回调函数的事件处理。你可能会发明一种结构，这种结构中创建了所有已知路径的关联，但是大部分应用最终以处理程序建立杂乱而结束。

批注 [spy4]: 意思是基本上可以维护，等，不存在大规模的困难

在 Angular 中，我们决定重新审视这个问题。

由于这些概念的诞生，世界作出了改变。对于任何有兴趣的人群，第一点已不在是事实。

如果你运行着不支持 JavaScript 的浏览器，那么你将被归类到 20 世纪 90 年代的站点。对于第二点，现代的屏幕阅读这已经赶超了。随着使用 ARIA 语义标签的使用，你可以制作十分丰富的界面并很容易访问。手机上运行 JavaScript 可以和台式机相提并论。

那么现在的问题是我们能够解决问题 3 和 5 同时增加可阅性和内嵌技术的简单化？

正如前面提到的，对于大多数内嵌事件处理，Angular 有一个等价的 ng-eventHandler='expression'形式，在 ng-eventHandler 中可以使用 click，mousedown，change 等等。如果你希望当用户点击一个元素时获取一个通知，简单点你可以使用 ng-click 标识符像这样：

批注 [spy5]:
无障碍互联网应用
Accessible Rich Internet Applications **(ARIA)**

https://developer.mozilla.org/en-US/docs/Accessibility/ARIA

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

你的大脑会说"no，no，no！糟糕，糟糕，太糟糕了！"好消息是，你可以很淡定从容。

这些标识符不同与他们的事件处理程序区别就是：

- 在每个浏览中具有相同的行为。Angular 已经为你充分考虑这些不同。
- 没有在全局命名空间上运行。你指定的表达式只能访问在元素控制器范围内的函数和数据。

最后一点可能听起来有点神秘，因此让我们先看一个示例。在一个典型的应用中，你创建一个导航条和一个随你选择不同的菜单选项而变化的内容区域。我们可以简略的写出骨架程序，就像这样：

```
<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>
```

这里 nabar 中的<li>和内容区域中的 都有一个叫 doSomething()函数，当用户点击它们时就会调用他们。作为开发人员，你建立了在控制器代码中调用的函数。它们可能是相同的函数或者是不同的：

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

这里，doA()和 doB()可能是相同的或是不同的，因为你定义它们。

最后一点，结构和行为组织在一起。这是一个多层次的问题，因为你不能指向具体的负

面结果，但是非常类似于我们心中的实际问题，混合了展示和应用逻辑。当描述的问题被标记为结构和行为时，人们讨论出确实是有些负面影响。

有一个简单的严峻考验，我们可以用它来弄清楚，如果我们的系统遭受这种耦合：我们能够为应用逻辑创建一个单元测试，同时不需要 DOM？

可以的，在 Angular 中我们可以写包含业务逻辑而不带 DOM 引用的控制器。问题不是在在事件处理上，而是在以前我们写 JavaScript 的方式上。注意，到目前为止，我们所写的所有控制器，本书的任何地方，他们都是没有 DOM 引用或 DOM 事件的。你可以很容易的创建没有 DOM 的控制器。所有定位元素和操作事件的工作都是在 Angular 中完成的。

这个问题首先出现在编写单元测试。如果你需要使用 DOM，那么你必须在你的测试设置中创建它，增加了测试的复杂度。还需要更多的维护，因为当你的页面发生改变时，你需要为你的测试而修改 DOM。最终，访问 DOM 是慢的。缓慢的测试意味这缓慢的返回和最终缓慢的发布。Angular 控制器测试就没有这样的问题。

这样就可以了，你可以愉快地使用声明式的事件处理简单的和可读的事物，毫无违反最佳实践。

列表，表格和其他非侵略性元素

ng-repeat 可能是最有用的 Angular 标识符，用它为集合中的每个项一次创建一系列元素的拷贝。你应该在任何你希望创建列表的地方使用它。

比方说，我们为老师编写一个学生花名册应用。我们可能从服务器获取学生数据，但是这个例子中，我们只是在 JavaScript 中定义它做为一个模型。

```
var students = [{name:'Mary Contrary', id:'1'},
{name:'Jack Sprat', id:'2'},
{name:'Jill Hill', id:'3'}];
function StudentListController($scope) {
$scope.students = students;
}
```

为了显示这些学生数据，我们可以像这样做：

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

批注 [spy6]: 修正原文错误

Ng-repeat 将生成 HTML 内所有的标签的拷贝，包括它所持有的标签。有了这个，我们将看到：

- Mary Contrary
- Jack Sprat
- Jill Hill

链接分别是：*/student/view/1*，*/student/view/2* 和 */student/view/3*

正如我们之前看到过的，改变学生数组将自动改变渲染的列表。如果我们做一些像插入一个新学生到列表中操作：

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

模板中增加一个添加按钮：

```
<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>
```

批注 [spy7]: 修正原文错误

现在看到的效果：

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

Insert new Student

ng-repeat 标识符也给出了当前元素索引的引用$index，以及用$first,$middle,$last 代表一些布尔值，比如说集合中元素是第一个位置，在中间或者是最后一个位置。

你可以想象到使用$index 来标识表格中的行号。这里给出一个模板：

```
<table ng-controller='AlbumController'>
```

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

和对应的控制器：

```
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;
}
```

我们将看到如下效果：

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、上下文敏感工具和其他场景，显示和隐藏元素是一个关键特性。在 Angular 中的一切，都是基于模型的改变，进而通过标识符反映这些变化到界面上。

这里，ng-show 和 ng-hide 可以做这样的事。他们提供了等价单相反的功能，显示和隐藏是基于你传递给他们的表达式而决定的。那就是说，当它的表达式是 true 时 ng-show 将显示元素，反之则隐藏。当表达式为 true 时 ng-hide 会隐藏元素，反之则显示。无论使用哪一个都能清晰的表达你的意图。

这些标识符是通过设置元素的样式 display:block 显示和 display:none 隐藏进行工作的。让我们来举个例子：

```
<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 functions left as exercise to reader
}
```

批注 [spy8]: 修正 1.0.8 下显示问题

CSS 类和样式

到现在，很明显你可以在应用中通过使用{{}}解析注解来数据绑定，从而能够动态的设置类和样式。甚至可以组合匹配模板中的部分类名。例如，如果你希望有条件地禁用一些菜单，你可能给用户做一些类似如下的效果：

CSS：

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

你可能在这个模板中显示 stun 函数以便禁用

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

你需要通过控制设置 isDisabled 属性：

```
function DeathrayMenuController($scope) {
$scope.isDisabled = false;
$scope.stun = function() {
// stun the target, then disable menu to allow regeneration
$scope.isDisabled = 'true';
};
}
```

stun 菜单项的 class 将被设置成 menu-disabled-加上$scope.isDisabled 的值。由于初始值是 false，结果就是 menu-disabled-false。由于没有这样的 CSS 规则，因此没有效果。当 $scope.isDisabled 设置成 true 时，CSS 规则变成 menu-disabled-true，将会调用这个规则使文

本成灰色。

当把内联样式和设值结合起来，这种技术运行的相当的好，例如‘style={{ 一些表达式 }}‘。

批注 [spy9]: 内联样式：就是初始定义好的；设置：主要是修改后的样式。

虽然是一种小聪明，但是这种技术在组合类名时有着间接层次的缺点。虽然你能很容易地理解在这个小示例中的，但是它很快变得难以管理，以至于不得不同时阅读模板和 JavaScript 才能真正确地创建 CSS。

由于这样的缘故，Angular 提供了 ng-class 和 ng-style 标识符。它们每一个都需要一个表达式。表达式执行结果可能是下列之一：

- 一个字符串，表示空间隔开的类名
- 一个类名数组
- 一个类名到布尔值的映射

让我们假想下，如果你希望在应用的头部的标准位置向用户显示错误和警告。用 ng-class 标识符，你能做一些类似这样的：

```
.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;
};
}
```

你甚至可以做些更好的，像表格中高亮选中的行。比方说，我们正在构建一个餐馆目录，我们希望高亮用户点击的那一行。

在 CSS 中，我们为高亮的行创建样式：

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

模板中，我们设置 ng-class 的值为{selected:$index==selectedRow}。当模型调用 selectRows 时将匹配 ng-repeat 的$index，进而显示选中的样式。同样我们设置了 ng-click 来通知我们的控制器用户点击了哪一行。

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

在 JavaScript 中，我们只是建立一些虚拟的餐馆和创建了 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;
};
}
```

批注 [spy10]: 修正多个引号问题

src 和 href 属性的建议

当数据绑定到一个<img>或<a>标签时，在 src 或者 href 中使用{{}}的路径不能够正常运行。因为对于其他内容，浏览器是并行加载图片的，Angular 没有机会拦截数据绑定的请求。

虽然<img>直接的语法可能是这样：

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

然而，你应该用 ng-src 属性，并像这样写模板：

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

与之类似，<a>标签，你应该使用 ng-href：

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

表达式

在模板中使用表达式的目的就是为了在模板，应用逻辑和数据之间创建挂钩一样智能，但是同时要阻止应用逻辑进入模板。

直到此时，我们大多数一直使用数据的引用作为表达式传递给 Angular 标识符。但是这些表达式可以做更多的事情。可以做简单的运算（+,-,/,*,%），做比较（==,!=,>,<,>=,<=）,执行布尔逻辑（&&，||，!）和位运算（\^,&,|）。你可以调用控制器中暴露出的函数，可以引用数组和对象符号（[],{},.）。

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>
```

这里第一个表达式是 recompute()/10，虽然合法，把逻辑放到模板中的一个典型例子，应该避免这种情况。让你的视图和控制的职责分离，从而保证他们易执行和易测试。

虽然你可以用表达式做很多事，但是它们是用 Angular 的自定义解析器计算的。他们不是用 JavaScript 的 eval ()执行的，并且是有相当限制的。

相反，它们是用 Angular 的自定义解析器执行的。在这个里面，你不会找到循环结构（for,while 等等），控制流程运算（if-else,throw）或者修改运算符（++，--）。当你需要这些类型的操作时，可以在控制器或者通过标识符来做。

尽管和 JavaScript 相比表达式在很多方面有更多的限制，但是它们对 undefined 和 null 有更好的兼容。而不是抛出 NullPointerException 错误，模板只是简单的不渲染任何内容。这就允许你更安全的使用还未初始化的模型值，一旦它们被赋值，就会立刻显示到界面上。

UI 和控制器分离

控制器在应用中有三个作用：

- 在应用模型中设置初始状态
- 通过$scope 向视图（UI 模板）暴露模型和函数

- 监视模型发生变化的其他部分并做出相应的动作

我们已经看过这章很多例子，我们会意识到一点。然而，控制器概念的目的是提供代码或者逻辑，按照他们想和视图交互的样子执行用户所期望的。

为了保持控制器最小化和可管理，我们的建议是，为视图中的每个功能域创建一个控制器。那就是说，如果有一个菜单，就建立一个 MenuController。如果你有一个导航条，那么就写一个 BreadcrumbController 等等。

你可能正在开始下载图片，但是准确的说，控制器已经绑定到由他们负责的 DOM 上了。关联控制器和 DOM 节点的两个主要方法就是在模板中通过申明一个 ng-controller 属性指定它和通过一个路由用已动态可加载的 DOM 模板片段关联它，这就叫做一个视图。

我们将在这章的后半部分讨论视图和路由。

如果你有复杂的界面场景，那么你可以让你的代码保持简洁、可维护。通过创建内嵌的控制器，通过继承结构这些控制器能够共享模型和函数。嵌套的控制器是简单的。你可以简单地把一个控制器给以 DOM 元素，DOM 元素里面的是另外一个控制器，就像这样：

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

尽管我们通过内嵌控制器实现了这个，但是实际上是在作用域上发生了嵌套。传递给内嵌控制的 $scope 继承了它父控制器的$scope。在这种场景下，这意味着传递给 ChildController 的$scope 可以访问传递给 ParentController 的$scope 的所有属性。

发布 scope 中的模型数据

传递给控制器的$scope 对象是一种用来想视图暴露模型数据的机制。在应用中你可能有其他数据，但是当通过 scope 传递这些属性时，Angular 只考虑模型部分。你可以认为 scope 是一个上下文，用它做出的改变对你的模型是可见的。

我们已经看到很多显示使用 scope 的例子，就像$scope.count=5。同样也有一些间接的方法从模板自身建立模型。你可以用下面的方法来实现：

1. **使用表达式**。由于表达式在和它们关联的元素的控制器域中执行，所以在表达式中设置属性和在控制器中设置属性是一样的。那就是说，可以这做：

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

和下面这样做有相同的效果。

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

控制器定义如下：

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

2. **在表单输入框中使用 ng-model**。和表达式一样，ng-model 指定的模型参数是在控制器作用域中执行的。此外，它还在表单字段状态和指定的模型建立了数据绑定。

用$watch 观察模型变化

也许所有的 scope 功能函数中使用最多的就是$watch 函数，因为当你的模型发生改变时能够通知到呢。你可以监视单个对象属性或者计算结果（或函数），事实上，可以是任何能被访问的属性或者可以计算结果的 JavaScript 函数。这个函数签名：

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

每个参数的详细内容如下：

watchFn

这个参数是一个 Angular 表达式字符串或是一个返回监控的模型的当前值的函数。这个表达式会被执行多次，因此你需要确保它没有副作用。那就是说，没有改变状态下可以调用多次。出于同样的原因，watch 的表达式应该是计算上比较简单的。如果你传递一个字符串方式的 Angular 表达式，它将会在被调用的 scope 上用对象执行。

watchAction

这是一个函数或者表达式，当 watchFn 变化时将调用他们。函数形式而言，它有 watchFn 的新旧值以及一个 scope 引用，函数签名是 function(newValue,oldValue,scope)

deepWatch

这是一个可选的布尔参数，如果设置成 true，Angular 将检查在被监视对象中每个属性的每次变化。如果你希望监视数组中的私有元素或者对象中的属性，而不是仅仅一个简单的值时，你应该使用这个参数。

当你不再希望收到变化通知时，$watch 函数返回一个取消监听的函数。

如果你希望监视一个属性，之后取消注册，我们使用如下形式：

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

让我们回头看下第一章中的购物车场景的完整示例。比如说我们希望当用户购物车中的总值超过 100$时给一个 10$的优惠。我们使用如下模板：

```
<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>
```

CartController 就像这样：

```
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.bill.discount;
};

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

}

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

批注 [spy11]: 修正书中错误。

注意：在 CartController 控制器末尾，我们在 totalCart()上建立一个监视，totalCart()是用

来计算付款总额的。每当这个值发生改变时，watch 函数就会调用 calculateDiscount()，然后我们设置对这个值做适当的折扣。如果总值大于 100$，就设置折扣为 10$，否则折扣为 0；

批注 [spy12]: 修正书中错误

你可以在图 2-1 中看到这个示例是如何给用户看的：

图 2-1 打折的购物车

Watch()性能注意事项

前面的例子虽然能够正确执行，但是有一个潜在的性能问题。虽然不是很明显，但是如果在 totalCart()加一个调试断点，你将会看到它被调用了 6 次之后才渲染页面。虽然在应用中从未关注过它，但是在更加复杂的应用中，运行 6 次可能是一个问题。

为什么是 6 次？我们很容易地能够跟踪到 3 次，因为他们每个只运行一次：

- 模板中的{{totalCart | currency}}
- Subtotal()函数
- $watch()函数

Angular 又运行了它们一次，所以执行了 6 次。Angular 这样做是位了验证传递的模型变化已经完全传递，模型已经不再变化。Angular 是通过对所有的被监视属性做了一份备份，和当前值进行比对，从而观察它们是否发生变化来做这个检查的。事实上，Angular 可能运行高达十次从而确保完全的传播。如果十次迭代后变化仍然出现，那么 Angular 会带一个错误执行结束。如果发生了这种情况，那么你可能遇到了一个循环依赖，你需要修正它。

虽然你可能会担心这样的问题，但是到时候你读完本书后，这就不是问题了。虽然 Angular 在 JavaScript 中实现了数据绑定，但是我们一直致力于 TC39 上一个低层次的叫 Object.observe()本地实现。适当的使用这个，Angular 将会自动使用 Object.observe()随时随地的向你展示本地加速的数据绑定效果。

批注 [spy13]: ECMAScript TC39 http://www.ecma-international.org/memento/TC39.htm

在下一章节你将会看到，Angular 有一个非常棒的 Chrome 调试扩展应用，叫 Batarang，它能够为你自动地高亮那些昂贵的数据绑定。

既然我们了解了这问题，我们可以用几种方法来解决它。一种方法就是在数组中的每个元素上创建$watch 监控变化，这样仅仅是重新计算$scope 属性中的 total，discount 和 subtotal。

为了实现这点，我们将用这几个属性来更新模板：

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

然后在 JavaScript 中，我们将监视数组中的元素，数组发生任何改变都会调用函数重新计算总价，就像这样：

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

注意：$watch 指定了一个字符串的 items。这是可以的，因为$watch 函数既可以指定一个函数（就像我们之前做的）或是一个字符串。如果给$watch 函数传递了一个字符串，那么它将作为表达式在调用它的$scope 作用域中执行。

这种策略可能对你的应用非常有用。然而，由于我们监视着数组中的所有项，Angular 不得不为我们对它做一份拷贝用于对比。对于一个很大的元素列表，如果每次 Angular 执行页面我们只是重新计算 bill 这个属性，那么它可能运行的很好。我们可以给$watch 只传一个用于重新计算属性的 watchFn，就像这样：

```
$scope.$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;
});
```

监视多个事物

倘若你希望监视多个属性或对象，当他们发生变化是执行一个函数改怎么办？你有两种基本的选择：

- 把他们放到一个数组或对象中，同时传递 deepWatch 为 true

- 监控这些属性的连接串

第一种场景，如果你有一个对象，在作用域内有两个属性，希望在发生变化时执行 callMe 函数，你可以通过这样来监视它们：

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

当然，属性 a，b 可能在不同的对象上，那么只要你愿意你可以制造很长的列表。如果列表很长，你可以写一个函数，返回连接串而不是依赖表达式的逻辑。

第二种场景，你可能希望监视 things 对象上所有的属性。这种场景，你可以这样做：

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

这里，第三个参数传递了 true，就是要求 Angular 检查 things 对象的所有属性，只要它们任何一个发生变化就会调用 callMe()函数。这个在数组上运行很好，同样适用对象。

组织模块依赖

在任何高价值的应用中，搞清楚如何组织管理你的代码功能到责任范围通常是一项艰难的工作。我们已经看到了控制器是如何给我们一块地方用于存放暴露正确的数据和函数给视图层模板的代码。但是我们还需要其他代码来支持我们的应用，那该怎么办？能够容纳这个最明显的地方就是控制器中的函数。

这个对于小应用和到目前为止看到的示例很有用，但是在真实应用中很快变得难以管理。控制器可能变成一块垃圾场，里面有任何我们需要的东西。它们很难理解，有可能很难再去更改。

这里开始介绍**模块**。它为应用中的函数域提供了一种组织管理依赖的方法，是一种自动解决依赖的机制（也被称作**依赖注入**）。大体上说，我们称这些为依赖服务，因为它们给应用提供了指定的服务。

例如：如果我们的购物站点里一个控制器需要从服务器端查询一份销售列表，我们想用一些对象，暂时称作 Items，用来处理从服务器端查询列表。Items 对象依次需要一些通过 XHR 或 WebSockets 的方法和服务器上的数据库进行交互。

不用模块的做法类似这样：

```
function ItemsViewController($scope) {
// make request to server
...

// parse response into Item objects
...
// set Items array on $scope so the view can display it
---
}
```

批注 [spy14]: 个人批注：以前的一个前端项目，由于规划的不好，导致一个 JSP 页面中有好几十个<script>标签，最后页面加载的甚是慢，总结一点就是前端没有模块化！

虽然这个的确可以运行，但是有一些潜在的问题。

- 如果其他控制器也需要从服务器获取 Items，现在就不得不重新拷贝一份代码。这将使维护成为一种负担，假如我们重新做了架构或其他变更，我们不得不去更新多个地方的代码。
- 由于其他因素，比如服务器端认证，解析的复杂度等等，所以它很难说清这个控制器对象的职责范围，同时阅读代码也变得更加困难。
- 为了测试这部分代码，实际上我们需要一台运行的服务器，或者能够用动态方法 monkey patch 模拟 XMLHttpRequest 返回模拟数据。不得不运行服务器这使测试工作非常缓慢，这是一个痛苦的设置，通常给测试环节引入了片状问题。这个 monkey patch 路由解决了速度和片状问题，但是意味这你不得不记住未打补丁和已打补丁的对象，在测试用例，和它所带来的额外复杂度，以及强制你指定需要的线上数据格式（只要这个测试变化，就不得不更新测试用例。）

批注 [spy15]: A **monkey patch** is a way to extend or modify the run-time code of dynamic languages without altering the original source code. This process has also been termed **duck punching** and **shaking the bag**

使用模块，和从他们那边获取的依赖注入，我们可以写我们的控制器更加简单，类似这样：

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

你可能要问自己“的确，看上去很酷，但是 Items 从哪里来？”上面的代码假设我们已经定义了 Items 作为一服务。

服务是单例的（单实例）对象，能够执行必要的任务以支持应用的功能。Angular 自带很多服务，像用于和浏览器地址进行交互的$location，用于随 URL 变化切换视图的$route，以及和服务器交互的$http。

你能够和应该创建自己的服务，用于为应用处理所有特殊的任务。服务可以跨应用共享，如果需要他们。因此，当你需要跨控制器交互和分享状态，他们是一个很好的机制。Angular 捆绑的服务都以$开头，因此，虽然你可以按照你喜欢的来命名，但是避免使用用$开头从而避免命名冲突是一个很好的想法。

你可以用模块对象的 API 来定义服务。这里有三种函数用于创建不同层次复杂度和能力的通用服务：

函数	定义
provider(name, Object OR constructor())	一个可配置的、有复杂逻辑的服务。如果你传递了一个对象，那么它应该有一个叫$get 的函数返回这个这个服务的实例。否则的话，Angular 假设你已经产地了一个构造函数，当被调用时，创建这个实例
factory(name, $getFunction())	一个不可配置的、有复杂逻辑的服务。你指定一个函数，当被调用时，返回服务实例。你可认为是 provider(name,{$get:$getFunction()})
service(name, constructor())	一个不可配置的、简单逻辑的服务。有点类似于带构造函数的 provider，Angular 掉用它来创建服务实例。

稍后，我们看下 provider()的配置项，但是让我们讨论之前的 Items 示例使用 factory()的示例。我们可以像这样写服务：

```
// Create a module to support our shopping views
var shoppingModule = angular.module('ShoppingModule', []);

// Set up the service factory to create our Items interface to the
// server-side database shoppingModule.factory('Items',
function() {
  var items = {};
  items.query = function() {
    // In real apps, we'd pull this data from the server... 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;
});
```

批注 [spy16]: 修正错误

当 Angular 创建 ShoppingController 时，它将传递 $scope 和创建我们刚刚定义的 Items 服务。这个是通过参数名匹配来完成的。那就是说，Angular 看到 ShoppingController 类的函数签名，意识到他需要一个 Items 对象。由于我们已经定义了一个 Items 服务，它就直达从这里取值。

查找这些字符串依赖的结果意味着像控制器构造函数注入功能参数是与顺序无关的。因此不是这样：

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

我们可写成这样：

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

正如我们所预料的，它仍然能够运行。

为了能和我们的模板一起运行，我们需要用 ng-app 标识符标注模块名，就像这样：

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

为了完成这个示例，我们实现了模板的剩余部分如下：

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

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

批注 [spy17]: 修正

应用运行的结果如下图 2-2 所示：

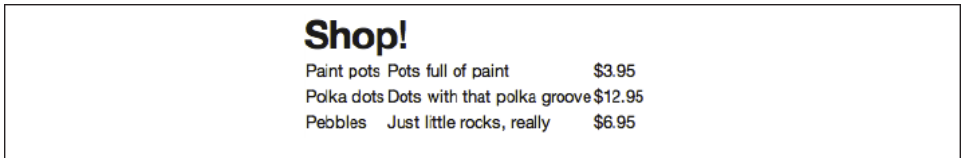


图 2-2 商品列表

需要多少个模块？

由于服务可能有依赖，因此模块 API 能够让你为依赖定义依赖。

在大多数应用中，为你的所有代码创建一个单独的模块，把你所依赖的都放到里面，这将运行的非常好。如果你使用服务或者来自第三方的标识符，它们将带伤自己的模块。由于你的应用依赖它们，因此你将他们作为你应用模块的依赖。

例如，如果你要包括（假设）SnazzyUIWidgets 和 SuperDataSync 模块，你的应用模块申明可能像这样：

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

用过滤器格式话数据

过滤器允许你申明如何将展示给用户的数据转换后插入到你的模板中。过滤器的使用语法是：

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

其中表示是可以是任何 Angular 表达式，filterName 是你想用的过滤器名称，传递给过滤器的参数用冒号隔开。这些参数可以是任何合法的 Angular 表达式。

Angular 自带几个过滤器，像之前看到的 currency。

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

这块代码将显示成如下：
$12.90

我们吧这个申明放在视图层（而不是在控制器或模型），因为$符号在数字前面只有对人有用，对我们处理数字的逻辑没有用。

Angular 自带的其他过滤器，包括 date，number，uppercase 等等。

在绑定时，过滤器可以用额外的管道标识链接起来。例如，我们可以格式话之前的示例，通过 number 过滤器删除小数点后的数字，这个过滤器带一个要舍入的小数点位数作为参数。因此：

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

将显示：
$13

这个并不限制于绑定的过滤器（就是说不限制于框架内嵌的），书写自定义的过滤器也很简单。如果我们希望创建一个过滤器，它能够将标题的首字母大写，例如，我们可能这样做。

```
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;
});
```

对应的模板像这样：

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

通过控制器赋值给模型变量 pageHeading

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

我们将看到如下如图 2-3：

图 2-3 标题大写过滤器

通过 Route 和$location 改变视图

尽管 AJAX 应用在技术上算是单页面应用（因为它们在第一请求时加载 html 页面，然后就是用 DOM 更新那些作用域），但是我们通常有多个子页面视图，适当的时候向用户展示或隐藏。

我们可以使用$route 服务来为我们管理这种场景。Route 可以为一个给定的浏览指向 URL 详细指定 Angular 能够加载和显示一个模板，实例化一个控制器为模板提供上下文。

在应用中可以在$routeProvider 服务上调用函数创建路由作为一个配置块。有点像下面的伪代码：

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

上面的代码的意思是：当浏览器的 URL 变成指定的 URL 时，Angular 将加载/path/to/templete 下的模板，同时用 aController 管理这个模板的根元素（假设我们输入了一个 aController 控制器）。

最后一行的 otherwise()告诉路由如果没有匹配到就走这一段。

让我们来实践一把。我们来构建一个超过 Gmail，Hotmail 等其他应用的 Email 应用。就叫它 A-Mail.现在我们从简单地开始。我们用第一个视图显示带时期，标题和发件人的 email 信息列表。当你点击一个消息时，它将显示消息体内容的新视图。

由于浏览器的安全限制，如果你尝试运行的代码不在应用中，你需要从一个 web 服务器上加载它，而不是使用 file://。如果你已经安装了 python，你可以在工作目录通过执行 python –m SimpleHTTPServer 8888 来提供服务。

对于主模板，我们会做一些不同寻常的东西。不是把所有的东西放到第一个加载的页面，我们只是创建了一个布局模板，我们把所有的视图都放到它里面。我们会把所有的东西持久话到视图，像这里的菜单。在这种情形下，我会只显示一个应用名称的标题。然后用 ng-view 标识符告诉 Angular 我们的视图应展示在哪里。

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>
```

由于我们的视图模板将被插入到刚刚创建的外壳中，所以我们可以把他们写成 HTML 文档片段。对于 email 列表，我们可以用 ng-repeat 来迭代消息列表，然后把它们渲染到表格中。

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>
```

这里注意，我们通过主题上的点击事件将用户导航到一个特定的消息。我们把数据帮 URL 上的 message.id，因此通过点击 id=1 的消息，将引导用户到/#/view/1。我们会用这种 URL 导航，也被称为深度链接，通过消息详细视图控制器，来展示一个特定的消息到详细视图。

为了创建消息详细视图，我们会创建一个模板来显示单个消息对象中的属性值。

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>
```

现在，将用控制器来关联这些模板，我们会配置带 URL 的$routeProvider 来调用我们的控制器和模板。

controllers.js

```
// Create a module for our core AMail services
var aMailServices  = angular.module('AMail', []);

// Set up our mappings between URLs, templates, and controllers
function emailRouteConfig($routeProvider) {
  $routeProvider.when('/', {
    controller: ListController, templateUrl:
    'list.html'
  }).
// Notice  that  for the  detail view,  we  specify a parameterized  URL  component
// by placing  a colon  in front of  the  id
  when('/view/:id', {
    controller: DetailController, templateUrl:
    'detail.html'
  }).otherwise({
redirectTo: '/'
});
}
// Set up our route  so the  AMail service  can find it
aMailServices.config(emailRouteConfig);

// Some  fake  emails
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."
}];
// Publish our messages for the list template
function ListController($scope) {
  $scope.messages = messages;

}
// Get the message id from the route (parsed from the URL) and use it to
// find the right message object.
function DetailController($scope, $routeParams) {
  $scope.message = messages[$routeParams.id];

}
```

批注 [spy18]: 修正，老外写的代码也不咋滴呢，错误还挺多的

我们已经为带多个视图的应用创建了基本结构。通过改变 URL 切换视图。这意味前进和后退按钮只针对用户有效。尽管只有一个 HTML 页面，但是用户能够收藏和链接到应用中的视图。

与服务器交互

好了，够乱搞的了。真正的应用通常需要和服务器进行交互的。移动引用和刚兴起的 Chrome 桌面应用可能是例外，但是对于其他所有而言，如果你想持久化到云端或者和其他用户实时交互，那么，你可能希望你的应用和服务器进行交互。

对于这点，Angular 提供了一个叫$http 的服务。它有一系列的抽象扩展，使和服务器交互变得很容易。他能够支持 HTTP，JSONP，CORS，包括安全措施防止 JSON 漏洞和 XSRF 攻击。它能够让你传输请求与响应数据更简单，甚至可以实现简单的缓存。

比方说，我们想从服务端的购物站点检索商品，而不是从模拟的内存中获取。编写服务端代码超出了本书的范围，因此假想下，我们已经创建了一个服务，当调用/products 服务时会返回一个 JSON 的产品列表。

这里给出一个像这样的相应：

批注 [spy19]: 连续运行卫星定位服务综合系统（Continuous Operational Reference System，缩写为CORS）

```
[
{
  "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...
]
```

我们可能写一个像这样的查询服务：

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

在模板中像这样使用：

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

正如我们之前学习的，从长远来看，把与服务器端交互的工作委派给一个可跨控制器共享的服务是一种不错的选择。我们会在第五章详细介绍这种结构以及$http 的所有功能。

用指令修改 DOM

标识符扩展了 HTML 语法，它是一种用自定义元素和属性关联行为和 DOM 变化的方法。通过它们，你能够创建可重用的 UI 组件，配置我们的应用，以及在 UI 模板中做几乎任何你可以想象的事。

你可以使用 Angular 内嵌的标识符来编写应用，但是你会陷入这样的情形，你需要编写自己的标识符。当你用一种内嵌的标识符还未支持的方式处理浏览器事件或者修改 DOM 时，你就会意识到是应该打破常规。这部分代码应该属于你写的标识符部分，而不是在控制器，服务或者是应用的其他地方。

正如服务一样，你通过模块 API 调用它的 directive()函数来定义标识符，在这个函数中 directiveFunction 是一个定义标识符特性的工厂函数

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

编写标识符工厂函数是一个很深的领域，我们已经在这本书中用单独的一章讲解了它。不过，为了引起你的兴趣，让我们看一个简单的示例。

HTML5 有一个很棒的新属性叫 autofocus，它能够把键盘的焦点聚焦在 input 元素上。你应该使用它，让用户开始第一次和元素交互时通过键盘不需要点击它。这个功能太棒了，因为它能够让你显示地指定浏览器做什么，而不需要写任何 JavaScript 代码。但是倘若你希望它在某个非 input 元素上运行，比如一个链接或某个 div？如果你想让它也能运行在非 HTML5 浏览器上，该怎么做？我们可以用一个标识符来实现：

```
var appModule = angular.module('app', []);
appModule.directive('ngbkFocus', function() {
  return {
    link: function(scope, element, attrs, controller) {
      element[0].focus();

    }
  };
});
```

这里，返回了一个带指定连接函数的标识符对象。这个链接函数有封闭域的引用，它所依赖的 DOM 元素，属性的数组传递给了标识符，以及 DOM 元素上的控制器，如果它存在的话。这里，只需要获取到这个元素，然后调用它的 focus()方法。

那么，我们可以在一个例子中使用它，就像这样：

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', []);
```

批注 [spy20]: 修正

当页面加载完成后，用户会看到高亮的”I’m very focused!”的按钮。点击空格键或者 enter 键会引起一个点击事件，调用 ng-click，将会设置 div 文本内容为”focus button clicked”。在浏览器中运行这个例子，我们会看到类似如果 2-4 效果

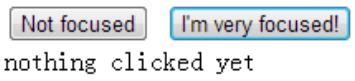


图 2-4 聚焦的标识符

校验用户输入

Angular 自动用几个适合于单页面应用的很好特性改善了<form>元素。特性之一就是，只有当整套元素合法时，Angular 才让你为表单中的输入框申明合法状态以及允许提交表单。

例如，如果我们创建一个注册表单，表单中需要输入一个名称和 email，但是有个可选的年龄字段，在用户提交到服务端之前我们能够校验多个用户属性。加载下面的示例到浏览器，展示的如图 2-5 所示：

图 2-5 表单校验

我们希望确保用户已经在 name 字段中输入文字，以及输入了一个合法的 mail，如果他输入年龄，那么应该是合法的。

我们可以在模板中做这些，使用 Angular 扩展的<form>以及多个 input 元素，就像这样：

```
<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，我们已经使用了来自 HTML5 的 required 属性，email 类型、number 类型的输入框，在一些字段上做我们的校验。这个能和 Angular 很好的运行，在老版本的非 HTML5 浏览器，Angular 会用相同功能的标识符来填充这些。

然后，给表单添加控制器，处理由表单变化的提交请求，来引用这个控制器。

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

在控制器内部，可以通过$valid 属性来访问表单的校验状态。当表单中所有的请求都是合法时，Angular 会把它设置成 true。我们可以使用$valid 属性来做额外的事，比如当表单还未完成时禁用提交按钮。

通过个体提交按钮添加 ng-disbaled，能够阻止非法状态的提交。

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

最后，我们也许希望控制器告知用户已经添加成功了。最终的模板可能像这样：

```
<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：

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

$scope.addUser = function () {
  // TODO for the reader: actually save user to database...
  $scope.message = 'Thanks, ' + $scope.user.first + ', we added you!';
};
}
```

继续前进

在上两章，我们看到了 Angular 框架中所有大部分通用的特性。对每个特性做了讨论，还有许多额外的细节我们还未覆盖到。在下一章，我们将带你，研究下一个典型的开发流程。

第三章：用 Angular 开发

到目前，我们已经对 AngularJS 的组成做了一点点研究。现在我们知道了如何从用户和应用获取数据，如果现实文本，如何做一些复杂的校验，过滤，设置修改 DOM。但是我们如何把他们组织在一起？

在这一章，我们覆盖一下内容：

- 为了快速的开发，如何布局你的 AngularJS 应用
- 在实践中服务器中查看 AngularJS 应用
- 使用 Karma 编写并运行单元测试和场景测试
- 为生成环境编译和压缩 AngularJS 应用
- 使用 Batarang 调试 AngularJS 应用
- 简化你的开发流程（从创建新文件到运行应用和测试）
- 用 RequireJS 集成 Angular 应用，一个依赖管理库

本章目标是给你一个 20000 英尺长的视图，如何尽可能地布局 Angular 应用。我们不会进入实际应用本身。那个是在第四章中，它深入一个简单的应用，使用并展示了多种 AngularJS 特性。

项目组织

我们推荐你的项目中使用 Yeoman，它会创所有必要的文件用来阴道你的 AngularJS 应用。

Yeoman 是一个有多个框架和客户端库构成的强大工具。它通过自动化一些需要引导的日常任务和开发应用提供了一个快速开发环境。我们会在这章中通过一整节介绍如何安装和使用 Yeoman，单在那之前，我们会简单的接触下 Yeoman 命令行，代替手动执行哪些操作。

如果你决定不使用 Yeoman，我们也会详细介绍涉及到的方方面面，因为 Yeoman 在 Window 计算机上确实有些问题，以及从它上构建可能稍微有点挑战。

对于那些没有使用 Yeoman 的，我们将看看一个简单的应用结构（可以在我们的 Github 示例库下找到 chapter3/sample-app 文件夹），它遵循了推荐的结构，同时是由 Yeoman 生成的结构。应用中的文件可以分为以下类别：

JS 源文件

看一下 app/script 文件夹。这里是所有 JS 源码所在的地方。一个主应用文件(app/scripts/app.js)会为应用建立起 Angular 模块和路由分发。

此外，有个一单独的文件---app/scripts/controller，放置这单独的控制器。这些控制器提供了动作和发布数据到那些显示在视图上的作用域。一般，它们和视图是一一对应的。

标识符，控制器和服务同样放在 app/scripts 下，如果它们是正规和复杂的，那么既可以是完整的文件(directives.js,filters.js,services.js)，也可以是单独地。

HTML Angular 模板文件

现在，Yeoman 创建的每个 AngularJS 部分模板都可以放在 app/views 文件夹下。它会为大部分映射 app/scripts/controller 文件夹。

还有一个重要的模板文件，就是入口 app/index.html。这个对管理 AngularJS 源文件有重大作用，也对你为应用创建的任何源文件有重大意义。

如果你创建了一个 JS 文件，确保你把它添加到 index.html 中了，以及也更新了主模块和路由（Yeoman 同样能为你做这些）。

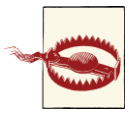
JS 库依赖

Yeoman 向你提供了所有 JS 源码依赖的 app/scripts/vendor 文件。想在应用中使用 Underscore 或者 SocketIO?没问题----添加这些依赖到 vendor 目录（和 index.html 中），然后在应用中开始引用它。

静态资源

最终你创建的是一个 HTML 应用，并且它是一个付出者，你会有 CSS 以及图片依赖，你需要为你的应用提供服务。

App/style 和 app/img 文件夹是处于这个目的。仅仅是在应用中添加你需要的东西，然后开始引用他们（当然，需要正确的相对路径）。

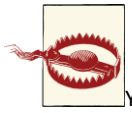

Yeoman 默认没有创建 app/img 路径

单元测试

当谈到 AngularJS 时，测试是超级重要的，并且毫不费力。用测试的观点来看，test/spec 文件夹应该是映射 app/scripts。每个文件应该有一个有单元测试的映射说明文件。种子为每个控制器文件，在 test/spec/controllers 目录下，和原始控制器具有同样的名称。这些是 Jasmine 风格说明，为控制器的每个期望的行为描述了一个规范。

集成测试

AngularJS 带有端到端的测试支持内嵌到库中。以 Jasmine 规范的形式，所有的 E2E 测试，都被保存在 tests/e2e 文件夹下。


Yeoman 默认情况下并没有创建 tests 文件夹


虽然 E2E 测试可能看上去有点想 Jasmin，但是它们不是。它们就是一些函数，通过 Angular Scenario Runner 在未来某一时刻异步的执行。因此不要期望能够做像你在一个正常的 Jasmin 测试（像 console.log 重复打印值）中那样做。

还有一个简单的 HTML 文件在浏览器中可以自己打开，然后手动的运行测试用例。虽然 Yeoman 并不为这些生成存根，但是它们遵循了类似的单元测试风格。

配置文件

有两个配置文件是必须的。第一个是，karma.conf.js，Yeoman 为你生成的，用于运行单元测试。第二个是 karma.e2e.conf.js，它不是 Yeoman 生成的。这个常用于场景测试。在本章的最后，和 RequireJS 集成章节，有一个示例文件。这些配置详细描述了当使用 Karma 运行这些但愿测试时用的依赖和文件。默认情况下，它运行在 Karma 服务器的 9876 端口。

你可能会问：我如何运行应用程序？如何运行单元测试？我甚至如何编写刚才讨论的这些多种多样的片段？

年轻人，不要担心。在这一章，我们会处理建立你的工程以及开发环境，以便这些东西能够迅速移动，一旦我们搅动了一些可怕的代码。写什么样的代码，以及如何组织它们形成最终的应用，这些将会在下面几张中讲述。

工具

AngularJS 只是允许你开发 WEB 页面的工具之一。在这一节，我们会看看多种工具，你会用它们来确保高效快速的开发环境，从集成开发环境到测试运行再到调试。

集成开发环境

让我们先从你如何编辑源代码开始。有一整套转换的 JavaScript 编辑器在那里，既有免费的又有付费的。过去在 JS 方面认为 Emacs 或 Vi 是最佳选择，但是这些东西都已过时了。现在，集成开发环境有语法高亮，自动补全等等，可能值得你使用。那么你应该使用哪个一种呢？

WebStorm，如果你不介意花几块钱的话（但是有 30 天体验），那么最近由 JetBrains 开发的 WebStorm 提供了最全面的 Web 开发平台。它很多特性，之前只能是指定类型语言可用，包括对多个类库和框架的代码补全（在哪个浏览器上，如图 3-1），代码导航，语法，错误高亮以及盒子模型支持。此外，还有一些不错的集成，虽然在 Chrome 中执行但是能够在 IDE 上执行。

图 3-1 WebStorm 中指定浏览器的代码补全

你应该慎重的考虑到 WebStrom 支持 AngularJS 开发的最大理由是它是仅有的支持 AngularJS 插件的 IDE 之一。对于 HTML 模板中的 AngularJS HTML 标签，插件能够给你自动补全支持。此外，它支持最酷的事情之一是内置模板的概念。这些是为公共代码片预置的模板，否则的话你会每次重头开始。因此而不是输入以下内容：

```
directive('$directiveName$', function factory($injectables$) {
var directiveDefinitionObject = {
$directiveAttrs$
  compile: function compile(tElement, tAttrs, transclude) {
    $END$
    return function (scope, element, attrs) {
    }

  }
};
return directiveDefinitionObject;
});
```

在 WebStorm 中你可以输入 ngdc，然后按 tab 键会看到相同的内容。这只是许多代码自动补全插件提供的功能之一。

运行应用程序

现在让我们讨论讨论我们如何获取我们的应用在浏览器中的有效荷载。为了获取到应用如何运作的实际感触，我们需要有一个 web 服务器来提供 HTML 和 JavaScript 代码。我们会探讨两种方式：一种是简单的方法，用 Yeoman 运行应用程序；另外一种并不容易，但是一样的好，不使用 Yeoman。

使用 Yeoman

Yeoman 很容易的让你启动一个 web 服务器，为所有的静态资源和 AngularJS 相关的文件提供服务。只需要执行下面的命令：

```
yeoman server
```

然后他会启动一个服务器，打开你的浏览器，浏览 AngularJS 应用的主页。它甚至能够每当你对源码做些修改时，它会自动刷新浏览器。是不是很酷？

不使用 Yeoman

不使用 Yeoman，你可能需要配置一个 web 服务器来为应用中的所有文件提供服务。如果你不知道一个简单的方法去做，或者不希望在创建 web 服务器上浪费时间，你可以用 Node 中的 ExpressJS（和使用 npm install –g express 一样简单，就可以搭建起来了 ）编写一个简单

web 服务器。它可能看起来像这样：

```
// available at   chapter3/sample-app/web-server.js
  var express = require("express"),
  app         = express(),
  port        = parseInt(process.env.PORT= parseInt(process.env.PORT,  10)  || 8080;

app.configure(function(){
  app.use(express.methodOverride());
  app.use(express.bodyParser());
  app.use(express.static(___dirname + '/'));
  app.use(app.router);
});

app.listen(port);
console.log('Now  serving   the   app at   http://localhost:' + port   + '/app');
```

一旦，你有有了这个文件，你可以用 Node 来运行这个文件，通过执行如下命令：

```
node web-server.js
```

然后，它会启动服务器端的 8080 端口（或者你定义的那个）。
或者，在应用的文件夹用 Python，运行如下命令：

```
Python –m SimpleHTTPServer
```

无论你使用哪种方式，一旦是配置好了服务器，并运行它，导航到如下地址：

```
http://localhost:[port-number]/app/index.html
```

在浏览器中可以看到刚刚创建的应用。注意，你可能需要手动刷新才能看到改变后的效果，这点并不像 Yeoman。

用 Angular 测试

我们之前说过（甚至在本章），这里在说一次：测试是有必要的，AngularJS 使编写正确的单元测试和集成测试变得简单。虽然 AngularJS 和多个测试工具运行的很好，但是我们坚信，Karma 胜过了他们当中的大多数，为你的所有需求提供了最强大，坚实的，出奇地快的测试工具。

Karma

Karma 存在的主要理由就是是你的测试驱动开发（TDD）流程变得简单，迅速和有趣。它使用 NodeJS 和 SocketIO（你不需要知道他们是什么，只管把他们想象成非常棒，很酷的类库）来允许运行你的代码，以极快的速度在多种浏览器中测试。在 https://github.com/vojtajina/karma/ 可以找到更多的内容。

TDD 介绍

测试驱动开发，或者叫 TDD，是一个敏捷方法，通过确保在代码是先前手动编写测试用例，用测试来驱动开发，从而翻转开发生命周期（它不只是作为一种校验工具）。

TDD 的原则很简单的：

- 只有当有失败的测试用例需要用代码来使它通过时才用编写代码。
- 编写极少的代码确保测试通过
- 移除每一步中重复的
- 曾经所有的测试是通过的，但是因为增加了一个功能就会增加一个失败的测试

这些简单的规则，确保了：

- 你的代码逐渐的变大，所写的每行代码有都是有目的的
- 你的代码依然保持这高度模块化，高内聚，高可重用（因为你需要能够测试它们）
- 你提供了一系列全面的测试，以防止未来的破损和缺陷。
- 为了满足未来需求和变化，这些测试扮演这规范，以及文档。

我们在 AngularJS 发现这是真的，以及整个 AngularJS 代码库已经使用 TDD 开发。对于像 JavaScript 这类不需要编译动态语言，我们坚信未来会有一些列良好的单元测试会减轻令人头痛之事。

那么我们是如何知道这个了不起的工具就是 Karma 呢？那好，首先确保你机器上安装了 NodeJS。它自带 NPM（Node Package Manager），NPM 使管理和安装上千个 NodeJS 的类库更加简单。

一旦你已经安装了 NodeJS 和 NPM，那么安装 Karma 和运行一样容易：

```
sudo npm install -g karma
```

之后就可以使用了。通过三个简单的步骤，您已经准备好开始 Karmaing（我只是说说而已，请不要在实际中使用）

设置你的配置文件

如果你使用 Yeoman 来创建应用架构，那么你已经有了一个生成好的 Karma 配置文件，就等着你使用。如果不是，直接跳过，从应用的根目录执行下面的命令：

```
karma init
```

在终端，你将会生成一个虚拟的配置文件（karma.conf.js），可以根据你的喜好进行编辑，

带有一些相当好的标准默认值。你可以直接使用。

启动 Karma 服务器

直接运行如下命令：

```
Karma start [optionalPathToConfigFile]
```

这个将启动端口为 9876 的 Karma 服务器（你可以从之前的步骤中修改 karma.conf.js 文件，来改变这个默认值）。虽然 Karma 应该能自动的打开浏览器并自动捕获，但是它会在控制台打印所有需要捕获另外一个浏览器的指令。如果你懒得做这个，那么只管在另外一个浏览器或设备中输入 http://localhost:9876，然后在多个浏览器中运行测试用例。

虽然 Karma 能够在启动时自动地捕获常规的浏览器（Firefox,Chrome,IE,Opera,甚至 PhantomJS），但是它并不限于这些浏览器。任何你能打开输入 URL 的设备都可能是 Karma 的运行者。如果你打开 iPhone 或者 Android 设备，浏览 http://machinename:9876（首先要提供访问地址），那么你也能在手机设备上运行你的测试用例。

批注 [spy21]: PhantomJS is a headless WebKit scriptable with a JavaScript API. It has fast and native support for various web standards: DOM handling, CSS selector, JSON, Canvas, and SVG.

运行测试用例

执行如下命令：

```
karma run
```

就是那样，你应该在你运行命令行的控制台上得到了输出结果。是不是很简单？

单元测试

AngularJS 是编写单元测试变得容易，以及默认支持支持 Jasmine 风格方式编写测试用（和 Karma 一样）。Jasmine 就是我们我所说的行为驱动开发框架，它允许你编写表示你的代码如何运作的规范。Jasmine 中一个简单的测试看起来像这样：

```
describe("MyController:", function() {

  it("to  work correctly", function() {
    var a = 12;
    var b = a;
  expect(a).toBe(b);
  expect(a).not.toBe(null);
});
```

```
});
```

正如你所看到的，它本身有着高可阅读的格式，因为大多数代码可以用纯英文理解。它也提供了非常多样的、功能强大的匹配符（比如 expect），当然也有 setUp 和 tearDowns 的 xUnit 这样的东西（在每一个单个测试用例中执行前后调用的函数）。

AngularJS 提供了一些非常好的原型以及测试函数，允许你在单元测试中创建服务，控制器和过滤器，以及模拟出 HttpRequests。我们将会在第五章详述。

Karma 可以和开发环境集成起来，是开发更容易，同样在你所写的代码上得到更快的反馈。

和 IDE 集成

对于最新的、最强大的 IED，还没有 Karma 插件（到目前为止），但是实际上你一点不需要他们。你需要做的事就是在 IDE 里增加一个快捷命令执行“karma start”和“karma run”。这点通常可以通过增加一个简单的脚本来执行，或者 shell 命令，这个依赖于你所选的编辑器。当然，每当它完成运行后，你就应看到执行结果。

每次修改后运行测试

这点是所有 TDD 开发人员梦想的：能够运行他们所有的测试用例，每次他们保存，在几毫秒，能够很快的获取到结果。这个在用 AngularJS 和 Karma 很容易做到。结果是，Karma 配置文件（还记得之前的的 karma.confi.js 文件？）有一个看似平淡的”auto Watch”标志。把它设置成 true，每当它监视的文件（代码和测试代码）发生变化时 Karma 就会运行测试用例。如果你从 IDE 中运行 karma star，猜猜会有是什么样？Karma 的运行结果将在 IDE 中可用。你甚至不需要切换控制台或者终端就可以弄清发生了什么。

端到端/集成测试

随着应用的增长（在你意识到它之前，它增长的非常快），手动测试它们是否如预期的运行不只是删掉它。毕竟，每次你增加一个新特新，你不只是验证新的特性是否可用，还需要验证之前的特新仍然运行，确保它们没有错误或回归。如果你开始增加多个浏览器，你可能很容易看到这如何变成组合爆炸。

AngularJS 尝试通过一个场景运行器模拟用户和医用交互，换来缓解这个问题。

场景运行器允许你按照类似 Jasmine 语法描述你的应用。就像之前的单元测试，我们会有一系列的“描述”（未来可能会实现），以及单个描述（用于描述每个单个特性函数）。你可能有一些公共的动作，在每个前或后执行（因为我们称他们为一个测试）。

一个简单的测试，在应用中筛选一系列结果，可能看上上像这样：

```
describe('Search Results', function() {
beforeEach(function() {
  browser().navigateTo('http://localhost:8000/app/index.html');
});
  it('should filter results', function()
  { input('searchBox').enter('jacksparrow');
  element(':button').click();
```

```
  expect(repeater('ul  li').count()).toEqual(10);
  input('filterText').enter('Bees'); expect(repeater('ul
  li').count()).toEqual(1);
});
});
```

有两种方式运行这些测试。无论你用哪种方式运行他们，你必须有个一个 web 服务器，为你的应用提供服务（更多如何去做的信息，请参考之前的章节）。一旦完成，使用下面方法之一：

1. **自动化**：Karma 现在支持运行 Angular 场景测试。通过以下修改创建 Karma 配置文件：
a) 增加 ANGULAR_SCENARIO，ANGULAR_SCENARIO_ADAPTER 到配置文件部分
b) 增加一个代理部分，重定向服务器的请求到测试文件所在的正确文件夹。例如：

```
proxies = {'/': 'http://localhost:8000/test/e2e/'};
```

c) 增加一个 karma 根目录，确保 karma 的源文件不影响你的测试文件，就像这样：

```
urlRoot = '/_karma_/'
```

然后只要记住通过浏览 http://localhost:9876/_karma_捕获你的 Karma 的服务端，你就可以自由的使用 karma 运行你的测试用啦。

2. 手动化：手动方式允许你从 Web 服务器打开一个简单页面，运行（查看）所有的测试。为了做这个，你必须：
a) 创建一个简单那的 runner.html 文件，它包含 Angular 库中的 angular-scenario.js 文件
b) 管理所有按你写的作为场景套件规定的 JS 文件。
c) 启动 Web 服务器，浏览 runner.html 文件

为什么应该使用 Angular 场景运行器，换种说法就是扩展第三方集成，或者端到端测试运行器？使用场景运行器可以获得惊人的好处，包括：

AngularJS 意识

Angular 场景运行器，顾名思义，由 Angular 为 Angular 创造。因此，它就是 AngularJS 意识，知道并理解多种 AngularJS 元素，比如绑定。需要输入文本？检查绑定的值？校验迭代器的状态？所有的可以通过这些通过使用场景运行器来完成。

无需等待

Angular 意识也意味这 Angular 意识到所有的发送到服务端的 XHR，从而能够避免页面加载过程中随机间隔时间等待。场景运行器知道当一个页面加载完成后，从而加载更多决定性的，而不是进行测试。例如，测试超时可能失败，而不是一直等待页面加载。

调试功能

如果你能看到代码，可以深入到 JavaScript，在场景测试进行运行时，当你想要时，可以暂停，继续执行测试。是不是很好？使用 Angular 场景执行器，这一切都是有可能的。

编译

在 JavaScript 世界里编译通常是值压缩代码，通过使用 Google Closure Library，会有一定量的编译工作。但是为什么转换引以为荣的，写的很好的以及很容易理解的代码成不纯无意义的？

一种原因就是把应用做成那样是为了对用户快速响应。另一个主要原因就是为什么客户端应用不再像几年前那样了。让应用越快的上线，就能越快的获取反馈。

这种响应是压缩代码的效果。代码越少，花费的消耗就越少，加载到用户浏览器就越快。这个在移动应用中尤其重要，文件的大小可能会成为瓶颈。

有几种方法，你可以压缩为应用写的 AngularJS 代码，每种方法有不同层次的作用：

基本、简单的优化

这个包括压缩代码中使用的所有变量，但是避免压缩属性。这就是 Closure Compiler 所谓的简单优化。

这种方式可能在文件大小上不会有太大的减少，但是你会获到实质的，最小的开销。

不压缩属性的原因是编译器（Closure 或 Uglifyjs）避免重命名模板中引用的属性。因此，模板可以继续运行，只有本地的变量和参数被重命名了。

用 Google Closure，简单调用：

```
java -jar closure_compiler.jar --compilation_level SIMPLE_OPTIMIZATIONS
--js path/to/file.js
```

高级优化

高级优化需要一些技巧，因为它尝试重命名了很多每个函数都有可能。为了做到这个层次的优化工作，你需要操纵编译器，准确（外部文件）的告诉这些需要重命名的函数，变量，属性。这些通常是模板访问的函数和属性。

编译器使用外部文件，重命名所有。如果完成了。这可以让 JavaScript 的大小大幅度减小。但是，这需要大量的工作，包括每次代码变化都需要更新外部文件。

要记住一件事：当你想压缩代码时，必须声明依赖注入（在控制器中指定$inject 属性）

像这样是不行的：

```
function MyController($scope, $resource) {
// Stuff here
}
```

你应该像下面这样：

```
function MyController($scope, $resource) {
// Same stuff here
}
```

```
MyController.$inject = ['$scope', '$resource'];
or use the module, like so:
    myAppModule.controller('MyController', ['$scope',
                                            '$resource',
                                            function($scope, $resource) {

     // Same stuff here
}]);
```

这是唯一的方式，一旦那些需要混淆或压缩的变量后，AngularJS 需要指定你原始使用的哪个服务或变量。

 总的来说好的做法是，当你开始编译代码时，使用数组方式的注入，能够避免后续的问题。后面让你头疼的事，试图找出为什么$e 变量（一些服务的压缩混淆后的版本）的提供者会突然的丢失，这是不值得。

其他很棒的工具

在这一节，我们将看看其他工具，它们能够帮助你们减轻开发流程，使你们产出更高。这些范围从用 Batarang 调试到实际编码，以及用 Yeoman 开发。

调试

当你使用 JavaScript 时，在浏览器中调试代码将变成第二种选择。你越早的接受，你将受益越多。值得庆幸的是，这些已走过很长的一段路，由于当时还没有 Firebug。现在，无论哪种浏览器，通常使用一些东西进入你的代码，分析你的错误，分析出应用的状态。现在知道了在 Chrome 和 IE 上有开发者工具；Firebug 可以在 firefox 和 chrome 运行。

还有几个额外的属性当你调试应用时，对你有所帮助：

- 当你想调试代码时，总是一只切换到源代码和依赖的非压缩版本。这你可看到友好的变量名，同时看到行号和实际有用的信息以及可调试能力
- 尝试让你源代码放在单个 js 文件中，不是嵌在 HTML 中。
- 断点是有用的！它们允许你检查应用、模型、在指定时间点一切的状态。
- “在所有异常处暂停”是一个非常有用的选项，现在它内嵌到大多数的开发者工具中了。当异常发生时，调试器会高亮异常的那行。

Batarang

当然，我们有 Batarang。Batarang 是一个 Chrome 扩展，它在谷歌 Chrome 浏览器上增加了 AngularJS 知识并内嵌到开发者工具上。一旦安装好（你可重 http://bit.ly/batarangjs 中获取），它增加了另一个叫 AngularJS 的页签到 Chrome 开发者工具面板。

你过去一直想知道 AngularJS 应用状态是什么样的？每个模型，每个作用域，每个变量当前都包含什么？应用的性能如何？如果你还不知道，相信我，你会知道的。当你这样做是，Batarang 会为你服务的。

在 Batarang 中有四个主要有用的部分.

模型选项卡

Batarang 允许你从根元素向下深入到作用域。你可以看到作用域是如何嵌套的，模型是如何附属到上面的（如图 3-2）。你甚至可以在应用中实时修改它们，然后查看相应的变化。这是不是很酷？

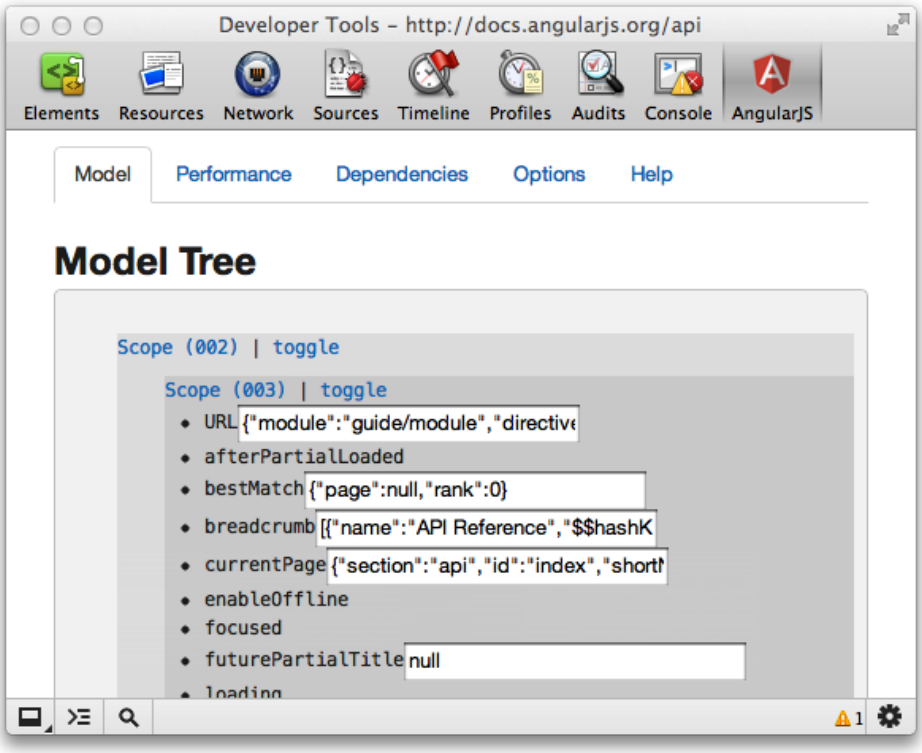


图 3-2 Batarang 中的模型树

性能选项卡

性能选项卡必须单独启用，因为它注入了一些特殊的 JavaScript 代码片段到应用。一旦启用了它，你可以看到多个作用域和模型，然后计算每个作用域中所有监控的表达式的性能（如图 3-3）。随着你使用应用，性能也会更新的，因此它也是实时运行的。

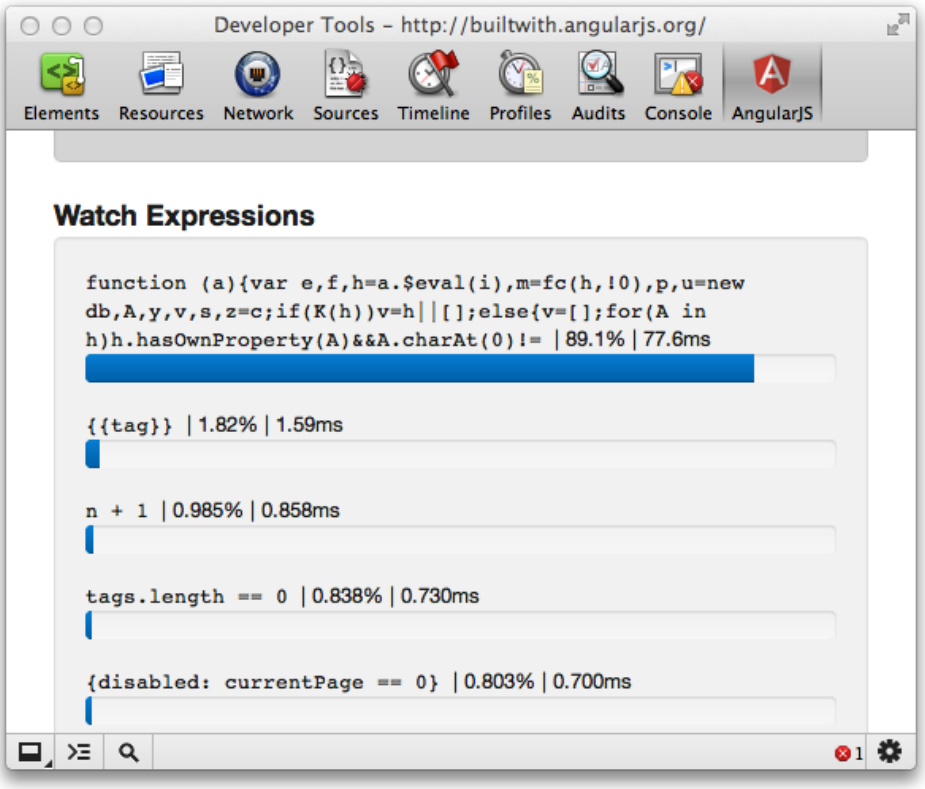


图 3-3 Batarang 中的性能选项卡

服务依赖

对于一个简单的应用，控制器和服务不会有超过一两个的依赖。但是在现实中、大规模应用中，如果没有合适的工具支持，服务依赖管理可能变得一团糟。这里 Batarang 能够为你提供服务，填补了这个空缺，因为它给你提供了一个整齐的，简单的方式是可视化化了服务依赖关系图（如图 3-4）

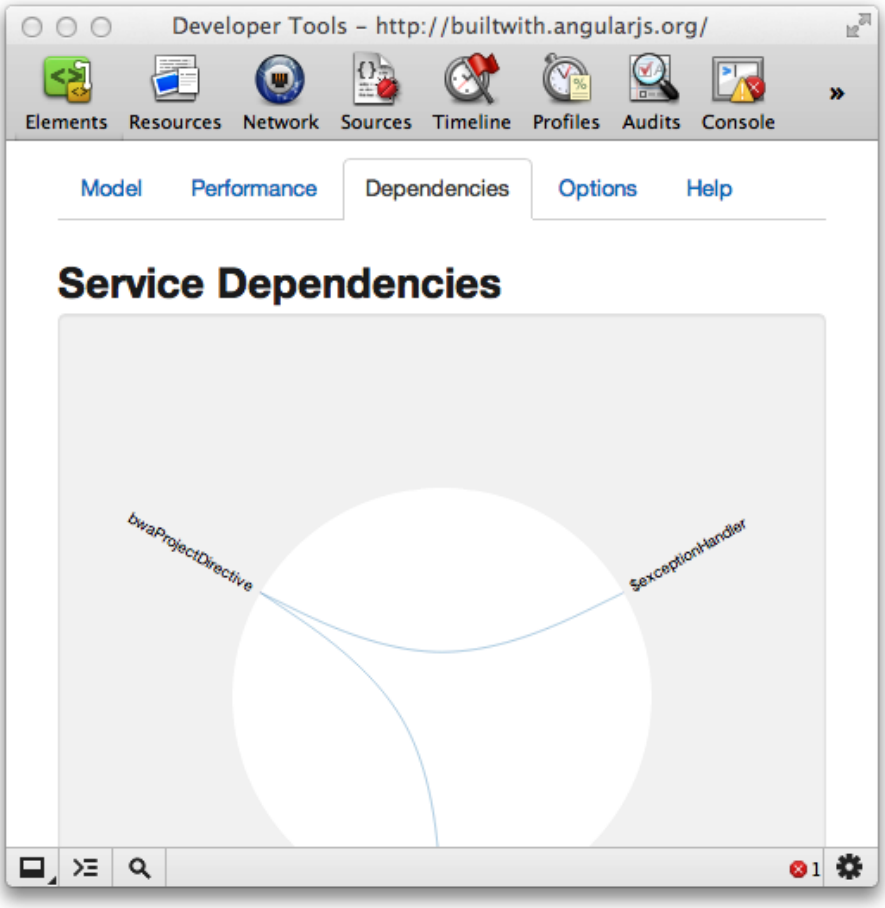


图 3-4 Batarang 中图表展示依赖关系

元素访问和控制台访问

当你深入一个 AngularJS 应用的 HTML 模板，在元素选项卡上的属性面板有一个额外的 AngularJS 属性选项。它允许你查看元素作用域上 banding 的模型。同时想控制台暴露了元素的 scope，因此你能在控制台上通过的$scope 变量访问它。如图 3-5 所示：

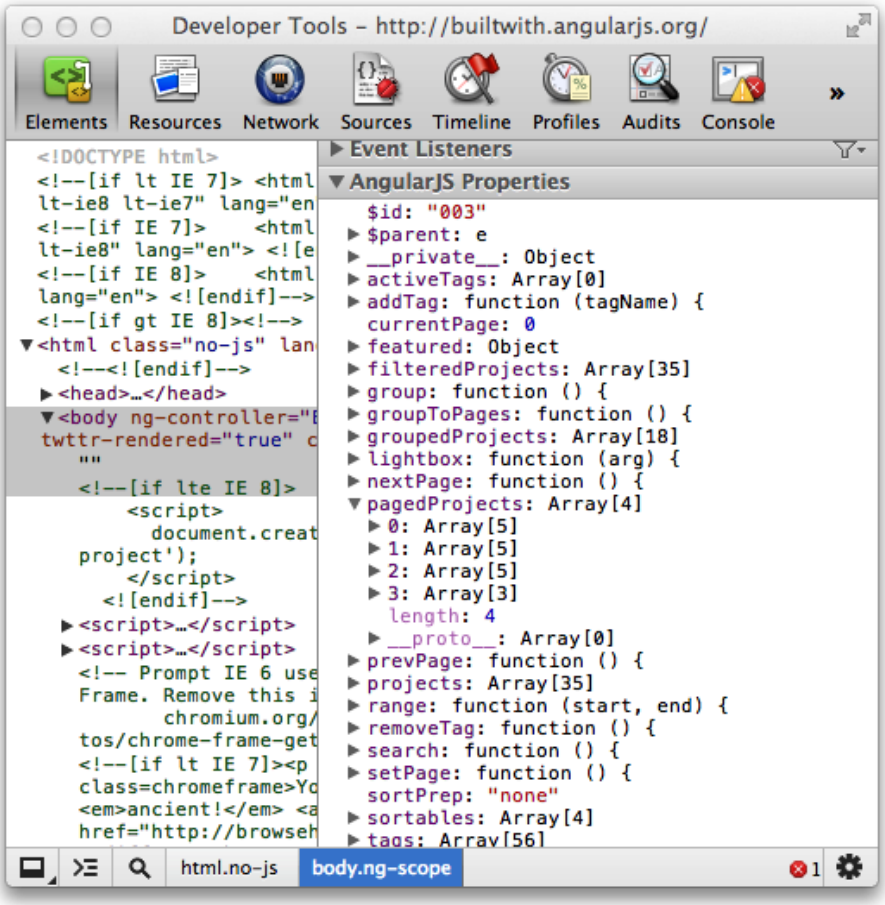


图 3-5 Batarang 中的 AngularJS 属性

Yeoman：优化你的工作流

当开发 Web 应用时，有很多工具如雨后春笋般涌现，以帮助优化您的工作流程。Yeoman，前面的章节接触到的，是一个拥有令人印象深刻的功能集，包括：

- 快速的脚手架
- 内置预览服务器
- 集成包管理
- 强大的构建流程
- 使用 PhantomJS 进行单元测试

它能够和 AngularJS 友好地和可扩展地集成，这是为什么我们强烈推荐 AngularJS 工程使用它的最主要原因。让我们体验下 Yeoman 是怎样让你简化生活方式的：

安装 yeoman

安装 Yeoman 是一个很复杂的过程，但是使用脚本对你是有帮助。
在 Mac/Linux 机器上，运行如下命令：

```
curl –L get.yeoman.io | bash
```

然后跟着它输出的指示获取 Yeoman。
对于 windows 平台，或许你会遇到一些问题，可以访问 https://github.com/yeoman/yeoman/wiki/Manual-Install，根据指示可以帮你扫除障碍。

开始一个新的 AngularJS 项目

正如之前提及的，即使是一个简单 AngularJS 项目也有很多需要做的事，从模板，基本的控制器以及库依赖，到其他任何需要组织的事。你可能会自己动手做，或者用 Yeoman 为你做。

为你的工程创建一个简单的文件夹（Yeoman 会使用文件夹名作为工程名），然后运行：

```
yeoman    init angular
```

这会以本章的‘项目组织‘章节所说的，创建整个详细的组织结构，包括渲染路由的骨架，单元测试等等。

运行 Server

如果你不适用 Yeoman，那么你不得不创建一个 HTTP 服务器，为你的前端代码提供服务。但是使用 Yeoman，你可以用内嵌的服务器，它是预配置好的以及有一些很好的额外好处。你可以使用如下命令启动服务器：

```
yeoman server
```

这个不仅仅启动一个 Web 服务器，为你的代码提供服务，同时会自动地打开你的浏览器，当你对应用做修改时自动的刷新浏览器。

添加新的 Routes,Views,Controllers

给 Angular 添加一个新的路由涉及到几个步骤，包括：

- 在 index.html 中填新的控制器文件

- 给 AngularJS 模块添加正确的路由
- 创建 HTML 模板
- 天剑单元测试

在 Yeoman 所有这些都可以通过如下命令来完成：

```
yeoman init angular:route routeName
```

那么如果你停止运行 yeoman init angular:route home，它会：

- 在 app/scripts/controllers 文件夹下创建一个 home.js 控制器骨架
- 在 test/specs/controllers 文件件下创建一个 home.js 测试描述骨架
- 在 app/views 文件夹增加 home.html 模板
- 在主应用模块（app/scripts/app.js）中管理 home 路由

所有这些都是一个命令来完成！

测试的故事

我们已经看到使用 Karma 是多么的简单地启动和运行测试。最后，只需要两个命令运行你所有的单元测试。

Yeoman 让它更简单（只要你相信它）。每当你用 Yeoman 生成一个文件，它同时会为你创建一个可填充的测试存根。一旦你安装了 Karma，用 Yeoman 运行测试就像运行下面的命令一样简单：

```
yeoman test
```

构建项目

构建应用的生产就绪版本是一件头疼的事，或者至少涉及许多步骤。Yeoman 通过允许你做以下事使这些步骤减轻了许多：

- 把所有的 js 文件整合到一个文件中
- 对文件进行版本化
- 优化图片
- 生产应用缓存清单

所有这些好处都来自一个命令：

```
yeoman buid
```

Yeoman 目前还不支持最小化压缩，但是据开发人员说这个功能不久就会有了。

AngularJS 和 RequireJS 集成

如果你提前做了很多事，那么正确配置你的开发环境将更简单。后期修改开发环境会对大量的文件产生修改。对任何一个大型项目，依赖管理和创建部署包是两项最令人担心的。

在 JavaScript 中，设置开发环境通常是很困难的，因为它涉及到维护 Ant 脚本，构建连接文件的脚本，压缩他们等等。值得庆幸的是，最近一段时间，想 RequireJS 这样的工具已经出现，它允许你定义和管理 JS 依赖，同时把他们放到一个更简单的构建过程。使用这些异步的加载管理工具，它能确保在代码执行前所有的依赖都被执行，专注于开发实际功能从来都没有像这样简单过。

值得庆幸的是，AngularJS 能够以及确实能够和 RequireJS 发挥很好的协作，所以你可以做到两全其美。为了说明这个示例的目的，我们会提供一个我们已经建立的示例，能够很好的运作，并且用一种系统的，易于遵循的方法。

让我们看看工程组织结构（类似于之前描述的架构，但有少量的改动）：

1. **app**：这个目录放置所有显示给用户的所有应用代码。它包括 HTML，JS，CSS，图片以及依赖库。
   a. **/style**：包含所有的 CSS/LESS 文件
   b. **/images**：包含工程中的图片
   c. **/scripts**：主要的 AngularJS 代码目录。这个文件夹也包括我们的引导代码，以及和 RequireJS 集成部分
      i. **/controllers**：AngularJS 控制器目录
      ii. **/directives**：AngularJS 标识符目录
      iii. **/filters**：AngularJS 过滤器目录
      iv. **/services**：AngularJS 服务目录
   d. **/vendor**：项目中依赖的库（比如 Bootstrap，RequireJS，jQuery）
   e. **/views**：视图层中的 HTML 模板以及项目中使用的组件
2. **config**：包含单元测试&场景测试的 Karma 配置
3. **test**：包含应用的单元测试&场景（集成）测试
   a. **/spec**：包含单元测试，映射在应用中的目录的 JS 文件夹结构
   b. **/e2e**：包含端到端的场景规范

我们首先需要做的事是 RequireJS 加载的 main.js 文件（在 app 目录），然后他会触发加载所有的其他依赖。在这个例子中，我们 JS 工程将依赖 jQuery 和 Twitter Bootstrap 加入到我们代码中：

```
// the app/scripts/main.js file, which defines our RequireJS config
require.config({
  paths: {
    angular: 'vendor/angular.min', jquery:
    'vendor/jquery',
    domReady: 'vendor/require/domReady', twitter:
    'vendor/bootstrap',
    angularResource: 'vendor/angular-resource.min',
```

```
  },
  shim: {
    'twitter/js/bootstrap': {
       deps: ['jquery/jquery']
     }, angular: {
      deps: [ 'jquery/jquery',
              'twitter/js/bootstrap'], exports:
      'angular'
    },
    angularResource: { deps:['angular'] }

  }
});
require([
      'app',
      // Note this is not Twitter Bootstrap
      // but our AngularJS bootstrap
      'bootstrap',
      'controllers/mainControllers',
      'services/searchServices',
      'directives/ngbkFocus'
     // Any individual controller, service, directive or filter file
     // that you add will need to be pulled in here.
     // This will have to be maintained by hand.
     ],
     function (angular, app) {
       'use strict';
app.config(['$routeProvider',
   function($routeProvider) {
     // Define your Routes here

   }
]);
}
);
```

然后顶定义一个 app.js 文件。这个定义了我们 AngularJS 应用，同时告诉它所依赖的的所有我们定义的控制器，服务，过滤器以及标识符；我们会看到 RequireJS 依赖列表中提及到一些文件

你可以想象下 RequireJS 依赖表作为一个 JavaScript 阻塞导入语句。那就是说，块中的函数知道所有的依赖都加载完成后才可以执行。

需要注意的是，我们不会单独地告诉 RequireJS 加载的什么是标识符、服务或者过滤器，因为那些不是工程所组织的。为每个控制器、服务、过滤器以及标识符有一个模块，因此它只是定义那些作为我们的依赖。

```
// The app/scripts/app.js file, which defines our AngularJS app
define(['angular', 'angularResource', 'controllers/controllers',
        'services/services', 'filters/filters',
        'directives/directives'], function (angular) {
  return angular.module('MyApp', ['ngResource', 'controllers', 'services',
                                  'filters', 'directives']);
});
```

同样我们也有一个 bootstrap.js 文件，它等待这 DOM 就绪状态（使用 RequireJS 的插件 domReady），然后叫 AngularJS 继续执行，这样非常好。

```
// The app/scripts/bootstrap.js file which tells AngularJS
// to go ahead and bootstrap when the DOM is loaded
define(['angular', 'domReady'], function(angular, domReady) {
   domReady(function() {
      angular.bootstrap(document, ['MyApp']);

   });
});
```

有另外一个好处就是把引导部分从应用中分离出来，我们可能用一个假的或者模拟的 APP 因为测试目的替换了我们 mainApp。例如，如果你们依赖的服务器是不可分离的，那么你可以创建一个 fakeApp，用模拟的数据替代所有的$http 请求，允许你在开发时平滑的切换。

通过这种方式，你可以平滑的切换 fakeBootstrap 和 fakeApp 到你的应用中。

现在，入口页面 index.html（在 app 文件夹下）可能看起来像这样：

```
<!DOCTYPE html>
<html> <!-- Do not add ng-app here as we bootstrap AngularJS manually-->
<head>
   <title>My AngularJS App</title>
   <meta charset="utf-8" />

        <link rel="stylesheet" type="text/css"
        href="styles/bootstrap.min.css">
        <link rel="stylesheet" type="text/css"
        href="styles/bootstrap-responsive.min.css">

   <link rel="stylesheet" type="text/css" href="styles/app.css">
</head>
<body class="home-page" ng-controller="RootController">
<div ng-view ></div>
            <script data-main="scripts/main"
            src="lib/require/require.min.js"></script>

</body>
```

```
</html>
```

现在，我们看下 script/controllers/controllers.js 文件，它和 scripts/directives/directives.js，script/filter/filters.js，script/services/service.js 有很大的相似度：

```
define(['angular'], function(angular) {
  'use strict';
  return angular.module('controllers', []);
});
```

因为使用了 RequireJS 的依赖结构，所有的这些只有在 Angular 依赖加载完成后才执行。

这里的每个文件定义了一个 AngularJS 模块，他们会在独立的控制器、标识符、过滤器和服务添加到定义中才会被使用。

让我们看看一个指定定义（正如第二章中的聚焦标识符）：

```
// File: ngbkFocus.js
define(['directives/directives'], function(directives) {
 directives.directive(ngbkFocus, ['$rootScope', function($rootScope) {
     return { restrict: 'A',
     scope: true,
     link: function(scope, element, attrs) {
       element[0].focus();

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

标识符自身可能没什么东西，但是能够让我们细看到所发生的事。RequireJS 的 shim 配置说了 ngbkFocus.js 文件依赖于“directive/directives.js”文件。然后它使用注入 directives 模块来添加自己的标识符声明。你可以选着有多个标识符，或一个文件就一个。这个完全有你决定。

一个主要注意点：如果你有个控制器需要一个服务（比如说 RootController 依赖于 UserSerivce，然后获取注入的 UserSerivce），那么你不得不确保你也把这个文件定义放到 RequireJS 中，就像这样：

```
define(['controllers/controllers', 'services/userService'],
  function(controllers) {
controllers.controller('RootController', ['$scope', 'UserService',
  function($scope, UserService) {
// Do what's needed
};
}]);
```

```
});
```

那只是简单地，你的整个源码结构是如何构建的。

但是你可能会问，这是如何影响我的测试呢？我们很高兴你能提出这样的问题，因为你即将就会知道答案。

好消息是 Karma 的确支持 RequireJS。只要安装了最新、最强大的 Karma 版本（使用 npm install –g karma）。

一旦你已经完成了，单元测试的 karma 配置也需要做轻微的修改。下面就是我们如何为我们之前定义的工程结构建立可运行的单元测试：

```
// This file is config/karma.conf.js.
// Base path, that will be used to resolve files
// (in this case is the root of the project)
basePath = '../';

// list files/patterns to load in the browser
  files = [ JASMINE,
  JASMINE_ADAPTER, REQUIRE,
  REQUIRE_ADAPTER,
// !! Put all libs in RequireJS 'paths' config here (included: false).
// All these files are files that are needed for the tests to run,
// but Karma is being told explicitly to avoid loading them, as they
// will be loaded by RequireJS when the main module is loaded.
{pattern: 'app/scripts/vendor/**/*.js', included: false},

// all the sources, tests // !! all src and test modules (included: false)
{pattern: 'app/scripts/**/*.js', included: false},
{pattern: 'app/scripts/*.js', included: false},
{pattern: 'test/spec/*.js', included: false},
{pattern: 'test/spec/**/*.js', included: false},

// !! test main require module last
'test/spec/main.js'
];
// list of files to exclude
exclude = [];

// test results reporter to use
// possible values: dots || progress
reporter = 'progress';
// web server port
port = 8989;
// cli runner port
runnerPort = 9898;
```

```
// enable/disable colors in the output (reporters and logs)
colors = true;

// level of logging
logLevel = LOG_INFO;
// enable/disable watching file and executing tests whenever any file changes
autoWatch = true;
// Start these browsers, currently available:
// - Chrome
// - ChromeCanary
// - Firefox
// - Opera
// - Safari
// - PhantomJS
// - IE if you have a windows box
browsers = ['Chrome'];

// Continuous Integration mode
// if true, it captures browsers, runs tests, and exits
singleRun = false;
```

我们使用一个稍微不同的格式来定义我们的依赖（included：false 是非常重要的），我们也血药在 REQUIRE_JS 和其适配器上添加依赖。最终让这些运行起来的是 main.js，它将触发我们的测试。

```
// This file is test/spec/main.js
require.config({
  // !! Karma serves files from '/base'
  // (in this case, it is the root of the project /your-project/app/js)
  baseUrl: '/base/app/scripts', paths: {
      angular: 'vendor/angular/angular.min', jquery:
      'vendor/jquery',
      domReady: 'vendor/require/domReady', twitter:
      'vendor/bootstrap', angularMocks:
      'vendor/angular-mocks',
      angularResource: 'vendor/angular-resource.min', unitTest:
      '../../../base/test/spec'

  },
  // example of using shim, to load non-AMD libraries
  // (such as Backbone, jQuery)
  shim: {
      angular: {
```

```
    exports: 'angular'
  },
  angularResource: { deps:['angular']}, angularMocks: { deps:['angularResource']}
                                                                                    }
                                                                                });
  // Start karma once the dom is ready.
  require([
    'domReady',
    // Each individual test file will have to be added to this list to ensure
    // that it gets run. Again, this will have to be maintained manually.
    'unitTest/controllers/mainControllersSpec',
    'unitTest/directives/ngbkFocusSpec',
    'unitTest/services/userServiceSpec'
  ], function(domReady) {
    domReady(function() {
        window.___karma___.start();

    });
  });
```

因此采用这种配置，我们可以运行如下命令：

```
karma start config/karma.conf.js
```

然后，我们就可以运行测试用例了。

当然，当它涉及到编写单元测试就有少量的修改。他们也需要成为 RequireJS 支持的模块，所以呢让我们来看一个简单的例子：

```
// This is test/spec/directives/ngbkFocus.js
define(['angularMocks', 'directives/directives', 'directives/ngbkFocus'],
    function() {
  describe('ngbkFocus Directive', function() {
    beforeEach(module('directives'));

    // These will be initialized before each spec (each it(), that is),
    // and reused
    var elem;
    beforeEach(inject(function($rootScope, $compile) {
      elem = $compile('<input type="text" ngbk-focus>')($rootScope);
    }));
   it('should have focus immediately', function() {
     expect(elem.hasClass('focus')).toBeTruthy();
   });
  });
  });
```

我们的每个测试会做如下操作：

1. 引入 angularMocks，它会提供 angular，angularResource，以及 angularMocks。
2. 引入高等级的模块（directives 带白哦标识符，controllers 代表控制器等等），然后实际上它是测试的单个文件（loadingIndicator）
3. 如果你测试依赖其他服务或控制器，确保你也定义了 RequireJS 依赖，另外把它告诉了 AngularJS

这种方式可以用于任何测试，你应该善于使用它。

值得庆幸的是，RequireJS 方式一点都不影响我们端到端的测试，因此他们能用目前我们所了解到的方法简单的完成。一个示例配置如下，假设你的应用运行在服务器端 http://localhost:8000/ 上。

```
// base path, that will be used to resolve files
// (in this case is the root of the project
basePath = '../';

// list of files / patterns to load in the browser
  files = [ ANGULAR_SCENARIO,
  ANGULAR_SCENARIO_ADAPTER,
  'test/e2e/*.js'

];

// list of files to exclude
exclude = [];
// test results reporter to use
// possible values: dots || progress
reporter = 'progress';
// web server port
port = 8989;

// cli runner port
runnerPort = 9898;
// enable / disable colors in the output (reporters and logs)
colors = true;
// level of logging
logLevel = LOG_INFO;

// enable / disable watching file and executing tests whenever any file changes
autoWatch = true;

urlRoot = '/_karma_/';
proxies = {
 '/': 'http://localhost:8000/'
```

```
};
// Start these browsers, currently available:
browsers = ['Chrome'];
// Continuous Integration mode
// if true, it capture browsers, run tests and exit
singleRun = false;
```

第四章：分析一个 AngularJS 应用

我们在第二章讨论了一些 AngularJS 的常用特性，然后在第三章深入到开发环境是如何组织的。第四章不是继续深入到特性，而是看一个小的，真实的应用。我们会感受一下我们之前讨论的（举一个玩具的例子）把所有的结合在一起形成一个真正的，可运作的应用。

不是把整个应用放在前面和中间，我们会在一段时间内介绍一部分，然后讨论感兴趣的以及相关的部分，慢慢地直到这章结束会构建整个应用。

应用

GutHub 是一个简单的食谱管理应用，我们设计它既能存储我们超级可口的食谱又能显示 AngularJS 应用的多个部分。这个应用：

- 有两栏布局
- 左边有个导航栏
- 允许你创建新的食谱
- 允许你浏览已存在的食谱列表

  主要的视图在右侧，它根据 URL 的不同而变化，显示食谱列表，或单个食谱的详细内容，或一个可编辑的表单用来添加或编辑已存在的食谱。我们可以看到应用的截图（图 4-1）

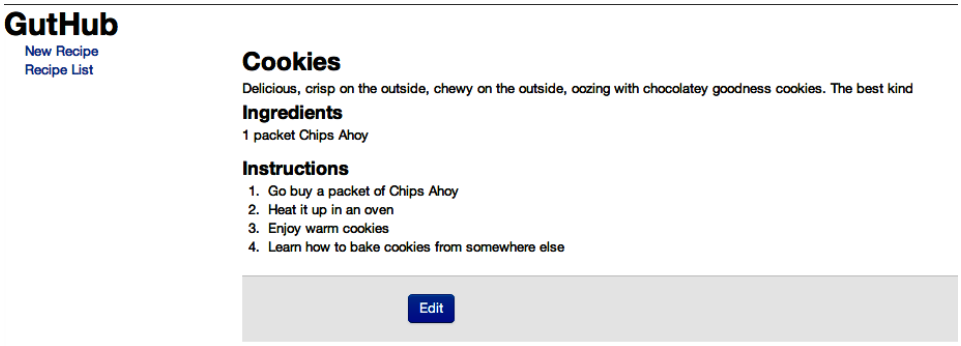


图 4-1 Guthub：一个简单的食谱应用

整个应用在我们 Github 仓库的 chapter4/guthub 下是可用的。

模型、控制器、模板之间的关系

在我们深入到应用之前，让我们用一两段讨论先如何将我们应用的三部分有机地进行合作，以及如何思考他们每个部分。

model（模型）就是真理。只要重复几次那句话。整个应用是由模型驱动的---显示什么样的视图，在视图中显示什么，保存什么，等等所有的一切。因此花点时间思考下你的模型，有什么样对象属性，如何从服务器获取它以及保存它。通过数据绑定，视图将会自动更新，因此关注点应该放在模型上。

controller（控制器）拥有这业务逻辑：如何获取模型，在上面执行什么样的操作，视图从模型中取什么样的信息，以及如何把模型传给你想要的内容。校验、调用服务器端，用正确的数据引导视图，以及它们之间的大部分事情的责任都属于控制器。

最终，***template***（模板）展示了模型如何展示的，以及用户是如何和应用进行交互的。主要限制于以下几点：

- 显示模型
- 定义用户和应用交互的方式（点击事件，输入框等等）
- 为应用指定样式，弄清清楚如何以及何时哪些元素需要展示（显示或隐藏，悬浮等等）
- 过滤以及格式化数据（包括输入和输出）

意识到 Angular 模板未必是 MVC 设计模式的视图部分。相反，视图是模板执行后的编译版本。他是模板和视图的结合。

任何类型的业务逻辑和行为都不应该进入到模板，这个应该严格限制到控制器上。保持模板简洁允许一个合适的关注点分离，同样要确保能够在单元测试下获取到大部分代码。模板会和场景一起测试。

你可能会问，DOM 操作去哪里了？DOM 操作实际上并没有进入控制器或模板。它进入了 AngularJS 标识符(但是有时可以通过服务使用，服务持有 DOM 操作能够避免代码从夫)。同样我们会在 GutHub 应用上详述那种示例。

事不宜迟，让我们深入探索吧。

我们为这个应用会把模型设置的很简单。还有菜谱。他们是整个应用中竟有的模型对象。其他的一切都不需它。

模型

每个食谱都有如下属性：

- 一个 ID，如果他持久化到服务器
- 一个名称
- 一个简短的描述
- 是否有特色的配方
- 一个数组的成分，包括数量，单位，名称

就这么多，超简单。应用中一切都是以及这个简单的模型，这里有个让你参考的简单食谱（和图 4-1 中的一样）：

```
{
"id": "1",
"title": "Cookies",
    "description": "Delicious, crisp on the outside, chewy" + " on the
    outside, oozing with chocolatey goodness " + "cookies. The best kind",
"ingredients": [
```

```
{
    "amount": "1", "amountUnits": "packet",
    "ingredientName": "Chips Ahoy"
  }
],

    "instructions": "1. Go buy a packet of Chips Ahoy\n" + "2. Heat it up
    in an oven\n" +
    "3. Enjoy warm cookies\n" +
    "4. Learn how to bake cookies from somewhere else"
}
```

我们将会看到根据这个简单的模型，如何创建更复杂的 UI 功能。

控制器，指令和服务

现在，我们终于可以将把牙齿深入到这个可口的应用的肉中了。首先，我们会看看标识符和服务代码，以及讨论下这里应该做什么，然将会看看这个应用需要的多个控制器。

批注 [spy22]: 简而言之，就是终于可以深入应用了。

服务

```
// This file is app/scripts/services/services.js
var services = angular.module('guthub.services', ['ngResource']);

services.factory('Recipe', ['$resource',
    function($resource) {
  return $resource('/recipes/:id', {id: '@id'});
}]);

services.factory('MultiRecipeLoader', ['Recipe', '$q',
    function(Recipe, $q) {
  return function() {
    var delay = $q.defer();
    Recipe.query(function(recipes) {
      delay.resolve(recipes);
    }, function() {
      delay.reject('Unable to fetch recipes');
    });
    return delay.promise;
```

```
  };
}]);

services.factory('RecipeLoader', ['Recipe', '$route', '$q',
    function(Recipe, $route, $q) {
  return function() {
    var delay = $q.defer();
    Recipe.get({id: $route.current.params.recipeId}, function(recipe) {
delay.resolve(recipe);
}, function() {
    delay.reject('Unable to fetch recipe ' + $route.current.params.recipeId);
  });
  return delay.promise;
};
}]);
```

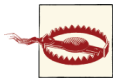
首先让我们看看服务。根据“用模块来组织依赖”的方式，我们构建了服务。这里，我们会再深入一点。

在这个文件，实例化了 3 个 AngularJS 服务。

这个有一个食谱服务，它返回我们调用的 Angular 资源。他们是 RESTfull 资源，指向一个 RESTfull 服务器。Angular 资源封装了底层的$http 服务，因此你可以在代码中直接调用这些对象。

由于只有单独一行代码，return $resource（当然依然与 guthub.services 模块），我们现在在任何一个控制器中可将 recipe 作为一个参数，它会被注入到控制器中。此外，每个 recipe 对象都有如下属性：

- Recipe.get()
- Recipe.save()
- Recipe.query()
- Recipe.remove()
- Recipe.delete()

 如果你想使用 Recipe.delete，和在 IE 中运行，那么你必须像这样使用：Recipe[delete]()。这是因为 ***delete*** 在 IE 中是个关键字。

和之前的方法一样，除了 query 的所有方法都可以和 recipe 协作。query()默认返回一个食谱的数组。

声明资源的代码行，return $resource，也为我们做了一些好的事情。

1. 注意在 URL 中指定的:id 是 RESTfull 的资源。简单的说当你做任何一个查询（例如，Recip.get()），如果你你传递一个 id 字段，那么这个字段的值都就添加到 URL 后面。那就是说，调用 Recipe.get({id:15})会调用/recipe/15。
2. 第二对象怎么办？{id：@id}？正如他们所说的，一行代码有一千中解释，因此让我们举个例子。

比如，我们有一个 recipe 对象，它有这必备的属性已经存储在里面了，包括一个 id。然后，我们通过如下简单的方式想要存储它

```
// Assuming existingRecipeObj has all the necessary fields,
// including id (say 13)
var recipe = new Recipe(existingRecipeObj);
recipe.$save();
```

这会发一个 POST 请求到/recipe/13

@id 告诉它从对象中取这个 id 字段值，然后用它作为 id 的参数。这个增加的便利能够为我们节省好几行代码。

Apps/scripts/services/services.js 里有其他两个服务。他们都是加载器，一个加载一个简单的食谱（RecipeLoader），另外一个是加载所有的食谱（MultiRecipeLoader）。这在我们连接我们的路由时使用。在核心代码上，他们有类似的行为。这些服务流程如下：

1. 创建一个$q 延迟对象（这些是 AngularJS promise，用于连接异步函数）
2. 向服务器发起一个请求
3. 当服务器返回值时，接卸这个延迟对象
4. 返回通过 AngularJS 路由机制的 promise

AngularJS 中的 Promises

Promise 是一个接口，用于处理返回或将被填充的在未来某一个时刻（简单的说，异步操作）。其核心，promise 是一个带 then()函数的对象。

为了展示其优势，让我们举个例子，在例子中我需要查询当用户信息：

```
var currentProfile = null;
var username = 'something';
fetchServerConfig(function(serverConfig) {
  fetchUserProfiles(serverConfig.USER_PROFILES, username,
    function(profiles) {
currentProfile = profiles.currentProfile;
});
});
```

用这中方式有几个问题。

1. 由此产生的代码简直就是一场噩梦，尤其如果你链式多个调用
2. 在回调和调用函数之间的错误有可能丢失，除非在每一步中都处理他们。
3. 你必须把你想用 currentPorfile 做的逻辑压缩到回调函数中，或者直接地，或者通过一个单独的函数。

Promises 解决了这样的问题，在我们知道如何做之前，让我们看看个用 promise 的同样问题。

```
var currentProfile =
  fetchServerConfig().then(function(serverConfig) {
  return fetchUserProfiles(serverConfig.USER_PROFILES, username);
}).then(function(profiles) {
```

```
  return profiles.currentProfile;
}, function(error) {
  // Handle errors in either fetchServerConfig or
  // fetchUserProfiles here
});
```

注意优势：

1. 你可以链式调用，因此你不会陷入噩梦之中
2. 你需要确认链式调用中在下一个函数调用之前，前一个函数调用已经完成了。
3. 每个 then()调用需要两个参数（都是函数类型），第一个是成功时回调函数，第二是失败时的句柄。
4. 一旦链式中出现错误，错误会传递到错误句柄的其他部分。因此，回调中的任何错误最后都是会被处理的的。

你会问 resolve 和 reject 呢？AngularJS 中的 deferred 是一种创建 promises 的方法。调用 resolve 后执行 promise(调用成功时的处理函数)，当 promise 调用出错时就会调用 reject。

当我们链接到路由时，会再次回到这里。

指令

现在我们可以讨论即将在应用中使用的标识符，应用中有两个标识符：

butterbar

这个标识符当路由改变是以及页面一直加载信息时会显示或隐藏。它会深入到路由修改策略机制以及基于页面状态，自动地隐藏和显示标签里面的内容，

focus

focus 标识符是用来确保指定的输入字段（元素）能够聚焦。

让我们看看代码：

```
// This file is app/scripts/directives/directives.js

var directives = angular.module('guthub.directives', []);

directives.directive('butterbar', ['$rootScope',
    function($rootScope) {
  return {
    link: function(scope, element, attrs) {
      element.addClass('hide');

      $rootScope.$on('$routeChangeStart', function() {
        element.removeClass('hide');
      });
```

```
                                                                    }
                                                                };
                                                            }]);

$rootScope.$on('$routeChangeSuccess', function() {
  element.addClass('hide');
});
directives.directive('focus',
    function() {
  return {
    link: function(scope, element, attrs) {
      element[0].focus();

    }
  };
});
```

前面一个标志返回了一个带单属性 link 的 Object 对象。我们会在第六章中深入介绍如何创建自定义的标识符。但是现在，你只需要知道的如下内容：

1. 标识符需要两步处理，在第一步中（编译阶段），所有的标识符绑定到已知的 DOM 元素，然后处理。任何 DOM 操作都发生在编译阶段。这一阶段结束后，就产生了链接函数。
2. 在第二步，链接阶段（以前我们都是用的这阶段），之前产生的 DOM 模板链接到作用域 scope。同时任何监控器或者监听器都按需添加了，结果就是在 scope 和元素之间建立以了绑定。因此涉及到 scope 的任何东西都是在链接阶段发生的。

那么，在我们的指令中发生了什么？让我们看看，好吗？
Butterbar 标识符可以像如下是使用

```
<div butterbar>My loading text...</div>
```

它基本上隐藏了右侧的元素，然后在根作用域下添加两个监视器。每当路由改变开始，它就显示元素（通过改变它的 class 属性），以及每当 route 成功完成是，就再次隐藏 butterbar。

另外一件有趣的是我们如何把$rootScope 注入到标识符中。所有标识符都是直接进入 AngularJS 的依赖注入系统，因此你可以注入你的服务以及你需要的一切。

需要注意的最后一点是处理元素的 API。熟悉 jQuery 的开发人员很高兴地知道它遵循了类似 jquery 的语法（addClass，removeClass）。AngularJS 实现了 jquery 调用的一个子集，因此对于 AngularJS 工程 jquery 是一个可选库。万一在项目中最终使用了 jquery 库，那么你应该知道 AngularJS 使用了它，而不是 jquery 内嵌的 jqlite 实现。

第二个标识符（foucs）更简单。只是调用了当前元素的 foucs()方法。你可以通过在任何输入元素上增加 focus 属性，就像这样：

```
<input type="text" focus></input>
```

当页面加载时，元素会立刻获取到焦点。

控制器

上面已经讲述了标识符和服务，最后我们可以进入控制器，这里我们有五个控制器。所有的这些控制器都是位于单一文件中（app/scripts/controllers/controllers.js），但是我们会依次查看它们。首先让我们看下第一个控制器，它是一个列表展示控制器，在系统中用于展示所有的菜谱列表。

```
app.controller('ListCtrl', ['$scope', 'recipes',
function($scope, recipes) {
$scope.recipes = recipes;
}]);
```

请注意列表控制器中一个非常重要的事情：在构造函数中，并不是去链接服务器和查询菜谱列表。相反，它已经查询到了菜谱。你可能想知道那是如何做的。我们会在这章的路由章节中得到答案，但是它确实是通过之前的 MuliRecipeLoader 服务做的。只要记住这点就行了。

和列表控制一样，其他的控制器自然而然十分类似，但是我们仍然会逐一的看下它们，找出有趣的一些东西：

```
app.controller('ViewCtrl', ['$scope', '$location', 'recipe',
  function($scope, $location, recipe) {
$scope.recipe = recipe;

$scope.edit = function() {
  $location.path('/edit/' + recipe.id);
};
}]);
```

视图控制器中有趣的是编辑功能，它暴露在 scope 作用域上。不是通过显示和隐藏字段或者类似其他的东西，这个控制器依赖于 AngularJS 来做这些繁重的任务（你也应该这么做）。编辑功能简单地改变 URL 来编辑菜谱，你看，AngularJS 做了剩余的东西。瞧！AngularJS 意识到 URL 已经改变，加载了相应的视图（在编辑模式下相同的菜谱）。

下面看看编辑控制器：

```
app.controller('EditCtrl', ['$scope', '$location', 'recipe',
  function($scope, $location, recipe) {
$scope.recipe = recipe;

$scope.save = function() {
  $scope.recipe.$save(function(recipe) {
    $location.path('/view/' + recipe.id);
  });
```

```
};
$scope.remove = function() {
  delete $scope.recipe;
  $location.path('/');
};
}]);
```

编辑控制器在 scope 上暴露出的 save 和 remove 方法有什么新的呢？

作用域上的 save 函数做你所希望做的。它保存了菜谱，一旦保存完成，将用户重定向至相同菜谱名称的视图界面。回调函数在这个场景中非常有用的，一旦你的操作完成用来执行一些动作。

这里我们可以用两张发灰色来保存食谱。一种方法是在所示的代码中，通过执行 $scope.recipe.$save()。这个只有在 recipe 是一个在第一次由 RecipeLoader 返回的资源对象时，才可用。

否则的话，保存 recipe 的方法将是：

```
Recipe.save(recipe);
```

Remove 函数也很简单，它从 scope 中移除了 recipe，将用户重定向到了主页面。注意，虽然做一个额外的调用应该不难，实际上并没有从我们的服务器上移除它。

下面，看下新建控制器：

```
app.controller('NewCtrl', ['$scope', '$location', 'Recipe',
  function($scope, $location, Recipe) {
$scope.recipe = new Recipe({
  ingredients: [ {} ]
});

$scope.save = function() {
  $scope.recipe.$save(function(recipe) {
    $location.path('/view/' + recipe.id);
  });
};
}]);
```

新建控制器几乎和编辑控制一样。实际上，你可以把它们放到一个单独的控制器中作为一个练习。仅有的主要区别是新建控制器创建了一个新的 recipe（它是一个资源，因此它有 save 函数）作为第一步。其他都没有变化。

最后，看下“成分”控制器。这个一个特殊的控制器，但是在我们升入到为什么和如何，先让我们看看：

```
app.controller('IngredientsCtrl', ['$scope', function($scope) {
$scope.addIngredient = function() {
```

```
  var ingredients = $scope.recipe.ingredients;
  ingredients[ingredients.length] = {};
};
$scope.removeIngredient = function(index) {
  $scope.recipe.ingredients.splice(index, 1);
};
}]);
```

到目前为止，我们看到的所有其他控制器都是连到 UI 上的部分视图。但是“成分”控制器是特殊的。它有个一子控制器，用在编辑页面上，用于封装某些不需要再外层（父级）需要的功能。需要注意的是，因为它是一个子控制器，它从父控制器上继承了 scope（这个场景中是编辑、新建控制器）。因此，它能够访问父控制器上的$scope.recipe。

控制器本身没有太有趣或唯一的东西。它只是在食谱的成分数组上加了一个新的成分，或者从食谱的成分列表上移除了一个指定的成分。

至此，我们已完成控制器中最后一个。剩下的仅有 JavaScript 片段就是如何建立路由：

```
// This file is app/scripts/controllers/controllers.js

    var app = angular.module('guthub', ['guthub.directives',
    'guthub.services']);

app.config(['$routeProvider', function($routeProvider) {
      $routeProvider. when('/', {
        controller: 'ListCtrl', resolve: {
          recipes: function(MultiRecipeLoader) {
            return MultiRecipeLoader();

          }
        }, templateUrl:'/views/list.html'
        }).when('/edit/:recipeId', { controller:
        'EditCtrl', resolve: {
          recipe: function(RecipeLoader) {
            return RecipeLoader();

          }
        }, templateUrl:'/views/recipeForm.html'
        }).when('/view/:recipeId', { controller:
        'ViewCtrl', resolve: {
          recipe: function(RecipeLoader) {
            return RecipeLoader();

   }
  }, templateUrl:'/views/viewRecipe.html'
  }).when('/new', { controller: 'NewCtrl', templateUrl:'/views/recipeForm.html'
```

```
}).otherwise({redirectTo:'/'});
}]);
```

正如之前提出的，我们最终达到了使用 resolve 函数这点。之前的代码片段设置了 Guthub AngularJS 模块，以及应用中的路由和模板。

这里联系着我们创建的标识符和服务，然后指定了应用中多种多样的路由。

对于每个路由，我们指定了 URL，备份它（URL）的控制器，待加载的载模板，最后（可选的），是一个 resolve 对象。

Resolve 对象告诉 AngularJS，在显示指定的路由给用户前，这个里的每个 resolve key 都需要满足。对于我们而言，希望加载所有的食谱，或单一的食谱，确保显示页面前我们已从服务器端获取了响应。因此，我们告诉路由提供者，我们已经有食谱列表（食谱），然后告诉它如何取查询。

这节链接到了我们在第一节定义的两个服务，MultiRecipeLoader 和 RecipeLoader。如果 resolve 函数返回一个 AngularJS promise，那么 AngularJS 是足够的聪明去等待在它进行前获取解析后的 promise。那就是说它会在服务器响应前一直处于等待。

然后，传递结果到构造函数，作为其参数（和对象字段的一样的参数名）。

最后，otherwise 函数表示默认重定向 URL，在没有路由匹配的情况下会触发。

你可能主要到编辑控制器和新增控制器路由都是指向相同的 URL 模板，views/recipeForm.html。会发生什么样的结果呢？我们充公了编辑模板。依赖了关联的控制器，在编辑模板中显示了不同的元素。

这部分已经完结，我们现在可以移到模板，这些控制器是如何关联他们，以及如何管理显示给终端用户的内容。

模板

让我们看下最外层的，入口模板，他是一个 index.html。这是我们单页面应用的基础，以及在这个模板上下文中加载其他所有的视图。

```
<!DOCTYPE html>
<html lang="en" ng-app="guthub">
<head>
  <title>GutHub - Create and Share</title>
  <script src="scripts/vendor/angular.min.js"></script>
  <script src="scripts/vendor/angular-resource.min.js"></script>
  <script src="scripts/directives/directives.js"></script>
  <script src="scripts/services/services.js"></script>
<script src="scripts/controllers/controllers.js"></script>
  <link href="styles/bootstrap.css" rel="stylesheet">
  <link href="styles/guthub.css" rel="stylesheet">
```

```
</head>
<body>
  <header>
    <h1>GutHub</h1>
  </header>

  <div butterbar>Loading...</div>

  <div class="container-fluid">
    <div class="row-fluid">
      <div class="span2">
        <!--Sidebar-->
        <div id="focus"><a    href="/#/new">New  Recipe</a></div>
        <div><a href="/#/">Recipe  List</a></div>
      </div>
      <div class="span10">
        <div ng-view></div>
      </div>
    </div>
  </div>
</body>
</html>
```

在上面的模板中有五个有趣的元素需要注意下，你已经在第二章中遇到过他们中的大部分。让我们挨个的复习下他们：

ng-app

我们设置 ng-app 为 Guthub。这个和我们在 angular.module 函数中给出的模块名是一样的。这样 AngularJS 就会知道如何吧两者关联上。

script tag

这里是为应用加载 AngularJS。它必须在所有使用 AngularJS 的 JS 文件加载前必须做的。理想情况下，应该在 body 底部执行。

Butterbar

我们第一次使用自定义的标识符。当我们定义 butterbar 标识符前，希望在一个元素上使用它，以便当路由发生改变以及成功隐藏时能够显示。按需显示高亮元素的内容（这里是非常简单的“Loading…”）。

Link href Values

hrefs 链向单页面应用中的多个页面。请注意，他们是如何使用#来确保页面不重新加载，是相对于当前页面。AngularJS 监控的 URL（只要页面没有重新加载），在需要的时候起到神奇的作用（实际上，我们定义的十分枯燥的路由管理作为路由部分）。

Ng-view

这是最后一块神奇的发生地。在我们的控制器章节，我们定义了路由，作为定义的部分，我们为每个路由定义了 URL，控制器管理路由和模板。当 AngularJS 检测到路由发生变化时，它会加载模板，把控制器绑定到它，然后用模板中的内容替换 ng-view

有一件奇怪的事是这里没有 ng-controller 标签。大多数应用都会有一个 MainController 关联最外层的模板。它最常见的位置是在 body 标签上。这里，我们并没有用，因为整个最外层模板没有 AngularJS 内容需要引用到 scope。

现在让我们看看单个模板以及关联的控制器，从食谱列表模板开始：

```
<!-- File is chapter4/guthub/app/views/list.html -->
<h3>Recipe List</h3>
<ul class="recipes">
<li ng-repeat="recipe in recipes">
<div><a  ng-href="/#/view/{{recipe.id}}">{{recipe.title}}</a></div>
</li>
</ul>
```

批注 [spy23]: fix

实际上，它是一个无聊的模板。这里只有两个有趣的地方。第一个是一个非常标准的 ng-repeat 标签用法，它从 scope 中读取 recipes，然后重复它们。

第二个是 ng-href 标签代替了 href。这里纯粹为了避免在 AngularJS 加载时产生一个坏链接。ng-href 确保没有一个畸形链接展示给用户。无论 URL 是动态还是静态的总是使用这个。

当然，你可能想知道，控制器在哪里？没有定义 ng-controller，也没有定义主控制器。这是路由映射在起作用。如果你记得的话（可能就在前几页），“/”路由重定向到了列表模板，然后列表控制器关联了它。因此当任何引用到的变量等等，都是在列表控制器的作用域内。

现在，我们移动到更多内容的地方：视图形式。

```
<!-- file  is  chapter4/guthub/app/views/viewRecipe.html  -->
<h2>{{recipe.title}}</h2>

<div>{{recipe.description}}</div>
<h3>Ingredients</h3>
<ul  class="unstyled">
  <li    ng-repeat="ingredient in    recipe.ingredients">
    <span>{{ingredient.amount}}</span>
    <span>{{ingredient.amountUnits}}</span>
    <span>{{ingredient.ingredientName}}</span>
  </li>
</ul>

<h3>Instructions</h3>
<div>{{recipe.instructions}}</div>

<form ng-submit="edit()"  class="form-horizontal">
```

```
  <div class="form-actions">
    <button class="btn btn-primary">Edit</button>
  </div>
</form>
```

另外一个优秀的，体积小的，包含模板。我们会提醒你三点事，尽管不是按照他们出现的顺序介绍。

第一点是非常标准的 ng-repeat。食谱又在视图控制器的作用域，在页面展示给用户前它是通过 resolve 函数加载的。这个确保了当用户看到它是，页面不是一个破碎的、未加载完成的状态。

另外一个有趣的用法是 ng-show 和 ng-class 来美化模板。添加 ng-show 标签到 span 标签，用于显示一个星号图标。现在，只有食谱是有详细介绍的食谱时才会展示星号（可以设置 recipe.featured 布尔值）。更完美地，为了确保合适的控件，你将在它上面用一个 ng-hide 标识符使用 ng-show 中相同的 AngularJS 表达式，使用另外一个空的图标。那是一个非常常见的用法，依据给出的条件显示某物或隐藏另外一个。

当食谱是一个有详细介绍的食谱时，用 ng-class 给<h2>标签添加 class。这里增加了一些特殊的高亮效果，确保标题更加突出。

最有一个需要注意的点是表单上的 ng-submit 标识符。标识符代表了当表单提交是，这个域上的 edit 函数将会被调用。表单提交事件发生在任何按钮，不需要准确的函数绑定（这里是，编辑按钮）点击时。同样，AngularJS 足够的聪明弄清楚正在引用的作用域，以及在正确的时间调用正确的方法。

现在，让我们看看最后一个模板（也可能是最复杂的一个），食谱表单模板：

```
<!-- file is chapter4/guthub/app/views/recipeForm.html -->
<h2>Edit Recipe</h2>
<form name="recipeForm" ng-submit="save()" class="form-horizontal">
<div class="control-group">
  <label class="control-label" for="title">Title:</label>
  <div class="controls">
    <input ng-model="recipe.title" class="input-xlarge" id="title" focus>
  </div>
</div>

<div class="control-group">
  <label class="control-label" for="description">Description:</label>
  <div class="controls">
            <textarea ng-model="recipe.description"
            class="input-xlarge"
            id="description"></textarea>

  </div>
</div>

<div class="control-group">
```

```
<label class="control-label" for="ingredients">Ingredients:</label>
<div class="controls">
  <ul id="ingredients" class="unstyled" ng-controller="IngredientsCtrl">
    <li ng-repeat="ingredient in recipe.ingredients">
      <input ng-model="ingredient.amount" class="input-mini">
      <input ng-model="ingredient.amountUnits" class="input-small">
      <input ng-model="ingredient.ingredientName">
            <button type="button" class="btn
            btn-mini"
            ng-click="removeIngredient($index)">
      <i class="icon-minus-sign"></i> Delete </button>
    </li>
          <button type="button" class="btn
          btn-mini"
          ng-click="addIngredient()">
       <i class="icon-plus-sign"></i> Add </button>
  </ul>
</div>
</div>

<div class="control-group">
  <label class="control-label" for="instructions">Instructions:</label>
  <div class="controls">
          <textarea ng-model="recipe.instructions"
          class="input-xxlarge"
          id="instructions"></textarea>

  </div>
</div>
<div class="form-actions">
  <button class="btn btn-primary">Save</button>
  <button type="button"
        ng-click="remove()"
        ng-show="!recipe.id"
class="btn">Delete</button>
</div>
</form>
```

不要头疼。它看上去有很多代码，它是由很多代码。但是如果你认真的深入到内部，其实它并不十分复杂。实际上，它是简单的很多，为了可以编辑食谱，重复的样板代码来展示了可编辑的输入字段：

- ***focus*** 标识符添加到了每个第一个标识符（***title*** 字段）。这个会确保当用户导航到这个页面是，title 字段会自动获取焦点，以便用户可以立刻输入 title。

- ***ng-submit*** 标识符和之前的示例非常类似，因此我们不会深入太多，除了将她保存食谱的状态以及通知编辑过程的结束。
- ***ng-model*** 标识符用于绑定多个输入框和文本域到模型
- 这页面上最有趣的一件事，推荐你花点时间尝试理解的东西是在详细列表部分的 ***ng-controller*** 标签。让我们花几分钟理解下这里发生的。

  我们会看到一系列将被展示的成分，和容器标签关联到 ng-controller。这意味这整个<url>标签被标记到成分控制器。但是这个模板的实际控制器是，“编辑”控制器？事实证明，‘成分’控制器是作为‘编辑’控制器的一个子控制器，因此继承了‘编辑’控制器作用域。这就是为什么它可以访问‘编辑’控制器中的 recipe 对象。

  此外，它增加了 addIngredient()方法，它只有通过 ng-click 触发，仅在<ul>标签作用域内是可访问的。为什么你希望这样做？这是最好的方法来分离你的关注点。当模板中的 99%都不关心它时，为什么‘编辑’控制器有一个 addIngredients()方法？子控制器和嵌套控制器对于如此明确，包含的任务是有意义的，允许你分析业务逻辑到更多可管理模块中。

- 我们想深入的另外一个标志符就是表单校验控制。在 AngularJS 世界中设置表单字段‘required’很简单。简单的添加 required 标签到输入框（就像之前代码中的）。但现在你怎么处理它呢？

处于这个目的，我们跳到保存按钮。注意它上面的 ng-disabled 标识符，它就是说 recipeFOrm.$invalid。这个 recipeForm 是我们申明的表单的名字。AngularJS 添加一些特殊的变量（$valid 和$invalid 仅是两个）那是允许你控制元素表单。AngularJS 看到所有必填元素和更新这些相应的变量。如果食谱标题为空，recipeForm.$invalid 就是 true（$valid 就是 false）我们的保存按钮就会立刻禁用。

我们也可以设置输入框的最大和最小长度，以及一个正则表达式对输入字段进行校验。此外，高级用法是仅当指定的条件满足时能够应用到显示确定的错误信息。让我们用一个小例子来看一看

```
<form name="myForm">
                    User name: <input  type="text"
                    name="userName"
                    ng-model="user.name"
                    ng-minlength="3">
<span  class="error"
      ng-show="myForm.userName.$error.minlength">Too   Short!</span>
</form>
```

在上面的代码中，我们增加了一个必填项，用户至少 3 个字符（使用 ng-minlength 标识符）。现在，在作用域中填充的表单，在这个表单中仅有 userName，每个对象都有一个$error 对象（它会包含上面样的有或没有的错误：required，minlength，maxlength，或者模式），以及$valid 标签用来标识输入框是否合法。

我们可以使用这个有选择性的想用户展示信息，依赖于他所制造的输入错误类型，就好像我们在前面例子中做的。

再返回到我们原先的模板中---Recipe form template，另外一个优秀的 ng-show 用法是在成分重复作用内。新增按钮只有在最后一个成分是才会显示。这是通过调用 ng-show 和使用

指定的重复元素内部的 scope 可访问的$last 变量来实现的。

最后，我们看下最有一个 ng-click，它是被绑定到第二个按钮，用于删除食谱。请注意，只有食谱还未保存时才展示按钮。虽然通常它会更多有意的事，写一个 ng-hide=”recipe.id”，但是有时做更有语义的事，比如说 ng-show=”!recipe.id”。这就是说，如果食谱没有 id 则显示，而不是有 id 时隐藏

测试

我们一直向你展示带控制器的测试，但是你早就知道了它们即将到来，不是吗？在这节，我们会看到你为各个部分写的各种各样的测试，以及你如何编写它们。

单元测试

首先以及最重要的类型测试是单元测试。这些测试是你开发的控制器（标识符，服务）是否是正确的构造、编写以及它们所做的是否你所期望的。

在我们深入单元测试前，让我们看看围绕我们控制器的单元测试的测试工具。

```
describe('Controllers', function() {
var $scope, ctrl;
//you need to indicate your module in a test
beforeEach(module('guthub')); beforeEach(function() {
  this.addMatchers({
    toEqualData: function(expected) {
      return angular.equals(this.actual, expected);

    }
  });
});

describe('ListCtrl', function() {....});
// Other controller describes here as well
});
```

这个测试工具（我们任然用 jasmin 的行为方式编写这些测试）做了几件事：

1. 创建了全局的可访问的作用域以及控制器，因此我们不要担心为每个控制器创建一个新的变量。
2. 初始化应用所使用的模块（这里是 Guthub）
3. 添加一个叫 equalData 的匹配器。这个简单的允许我们在通过$resource 服务或者 RESTful 调用的返回资源对象上（像 recipes）去执行断言。

 记住当我们需要在 ngResource 返回值上做断言时，需要添加 equalData 的匹配器。这是因为 ngResource 返回对象上有额外的方法，会使正常期望相等的调用失败。

正如上面的测试工具，让我们看看你列表控制器的的单元测试：

```
describe('ListCtrl', function() {
var mockBackend, recipe;
// _$httpBackend_ is the same as $httpBackend. Only written this way to
// differentiate between injected variables and local variables
beforeEach(inject(function($rootScope, $controller, _$httpBackend_, Recipe) {
  recipe = Recipe;
  mockBackend = _$httpBackend_;
  $scope = $rootScope.$new();
  ctrl = $controller('ListCtrl', {
    $scope: $scope, recipes: [1, 2, 3]
  });
}));
it('should have list of recipes', function() {
  expect($scope.recipes).toEqual([1, 2, 3]);
});
});
```

记住，列表控制器是最简单的控制器之一。控制器的结构只是展示了食谱列表以及把它保存到作用域。你可以为他写一个测试，但是它看上去有点简单（但是我们还是用了它，因为测试很棒！）。

然而，更有趣的是 ***MultiRecipeLoader*** 服务。这是一个从服务器查询食谱列表，然后作为一个参数传递过去的响应（当通过$route 服务正确地链接时）。

```
describe('MultiRecipeLoader', function() {
var mockBackend, recipe, loader;
// _$httpBackend_ is the same as $httpBackend. Only written this way to
// differentiate between injected variables and local variables.
beforeEach(inject(function(_$httpBackend_, Recipe, MultiRecipeLoader) {
  recipe = Recipe;
  mockBackend = _$httpBackend_;
  loader = MultiRecipeLoader;
}));

it('should load list of recipes', function() {
  mockBackend.expectGET('/recipes').respond([{id: 1}, {id: 2}]);
var recipes;
```

```
var promise = loader();
promise.then(function(rec) {
  recipes    = rec;

}); expect(recipes).toBeUndefined();


mockBackend.flush();

    expect(recipes).toEqualData([{id: 1}, {id: 2}]);
  });
});
// Other controller describes  here as well
```

在测试中，我们通过链接一个模拟的 HttpBackend 测试 MultiRecipeLoader。当测试运行时就会包含 angular-mocks.js 文件。只是把它注入到 beforeEach 方法中，这个方法是为你准备设置期望值的。第二不，更有意义的测试，我们设置了一个从服务器端 GET 调用的期望，它会返回一个简单的对象数组。然后我们用我们自定义的匹配器来确保这就是返回的值。注意，在模拟后端调用 flush()方法，它会通知模拟后端从服务器立刻返回一个响应。你可以用这种机制来测试控制流程，然后看看在服务器返回一个响应前后，应用是如何处理的。

这里跳过‘视图’控制器，因为它和列表控制器极其相似，除了在 scope 上有个 edit()方法。这是很简单的测试，因为你可以注入$location 到你的测试中，然后检查返回值。

让我们来看看‘编辑’控制器，它有两点有趣的，我们应该进行单元测试。Resolve 函数和我们之前看到一个的有点类似，可以用同样的方法测试。然而，我们现在想看看如何测试 save()和 remove()方法。让我们看看这些的测试用例假设我们测试工具来自之前的示例）

```
describe('EditController', function() {
  var mockBackend, location;
  beforeEach(inject(function($rootScope,
                             $controller,
                             _$httpBackend_,
                             $location, Recipe) {
    mockBackend  = _$httpBackend_;
    location = $location;
    $scope = $rootScope.$new();

    ctrl = $controller('EditCtrl', {
      $scope: $scope,
      $location: $location,
      recipe: new Recipe({id: 1, title: 'Recipe'})
    });
  }));
it('should save the recipe', function() {
  mockBackend.expectPOST('/recipes/1',
```

```
                {id: 1, title: 'Recipe'}).respond({id: 2});

  // Set it to something else to ensure it is changed during the test
  location.path('test');

  $scope.save();
  expect(location.path()).toEqual('/test');
mockBackend.flush();
expect(location.path()).toEqual('/view/2');
});
  it('should remove the recipe', function()
  { expect($scope.recipe).toBeTruthy(); location.path('test');

  $scope.remove();

  expect($scope.recipe).toBeUndefined();
  expect(location.path()).toEqual('/');
});
});
```

在第一个测试中，我们测试了 save()函数。特别地，我们确保保存功能首先带着参数向服务器发起一个 POST 请求，然后一旦服务器响应，地址就改变到最新的持久化对象的视图食谱页面。

第二个测试更简单。我简单地检查确保调用作用域上的 remove()移除当前的食谱，然后重定向用户到主页面。这个可以通过注入$location 服务到我们测试用例中简单地在做到，能和它一起运行。

控制器剩余部分的单元测试遵循着类似的模式，因此，我们略过它们。在它们底层，单元测试依赖几件事情：

- 确保控制器（更多的是 scope）在初始结束后达到正确的状态。
- 确认发起了正确的服务器端请求，以及在服务器调用和完成后的期间通过作用域完成了正确的状态（通过在单元测试中使用模拟的后端）。
- 凭借 AngularJS 依赖注入的框架在元素上获取句柄，以及控制器能够协作的对象来确保控制器设置了正确的状态。

场景测试

一旦我们对单元测试很满意，很可能就会可能禁不住的往后靠一下，抽根雪茄，收工。但是 AngularJS 开发人员不会这么做，知道他完成了场景测试。虽然单元测试确认了非常小的 JS 代码片段是可以运作的，我们也希望确保模板加载，关联到正确的控制器上，然后在模板周围点击时做正确的事情。

这正是 AngularJS 中的场景测试为你做的。它允许你：

- 加载应用
- 浏览一个指定的页面

- 任意的点击和输入文本
- 确保正确的事情发生

那么，食谱列表页面的场景测试是如何运作的呢？首先，在我们开始实际测试，需要做一些基础工作。

为了场景测试能够运作，我们需要一个运作的 Web 服务器，用于接收来自 GutHub 应用的请求，以及允许从服务器存储和获取食谱列表。随意地更改内存中的食谱列表（移除食谱 $resource，只是把它转成 JSON 对象存储），或者重新使用和修改前面章节中展示的服务器，或者使用 Yeoman！

一旦我们让服务器运行起来，为应用提供服务，然后我们编写和运行如下测试：

```
describe('GutHub App', function() {
it('should show a list of recipes', function() {
  browser().navigateTo('/index.html');
  // Our Default GutHub recipes list has two recipes
  expect(repeater('.recipes li').count()).toEqual(2);
});
});
```

第五章：与服务器通信

到这里，我们已经看到了大多数的 AngularJS 应用应该如何布局的，不同的 AngularJS 片是如何组合及运作的，以及一些 AngularJS 中的模板是如何运作的。合起来，这些允许你构建一个时尚的，性感的应用，但是这还限制于客户端。早先我们了解了一些在第二章使用 $http 的服务端的通信，但是在这章，我们会深入一点，如何在真实的应用中使用它们。

在这章，我们会讨论 AngularJS 允许你如何和服务器经行通信，它同时提供了在最底层的抽象和最好的包装器。此外，我们会深入 AngularJS 如何用内嵌的缓存机制帮助你加快应用。如果你想开发一个使用 SocketIO 的实时 AngularJS 应用，在第八章有一个示例，可以包装 SocketIO 作为一个标识符，然后使用它，这里不做详细讲述。

使用$http 通信

来自 AJAX 应用的传统方式向服务器发起的请求（使用 XMLHttpRequests）包括在 XMLHttpRequest 对象上获取一个句柄，然后发请求，读取响应，检查错误码，最终处理服务端响应。它就像这样：

```
var xmlhttp = new XMLHttpRequest();
```

```
xmlhttp.onreadystatechange = function() {
 if (xmlhttp.readystate == 4 && xmlhttp.status == 200) {
   var response = xmlhttp.responseText;
 } else if (xmlhttp.status == 400) { // or really anything in the 4 series
   // Handle error gracefully

 }
};
// Setup connection
xmlhttp.open("GET", "http://myserver/api", true);
// Make the request
xmlhttp.send();
```

如此简单，通用，重复的任务，是一个很大的工作量。如果你想再三的做它，你可能会创建包装器或者使用类库。

AngularJS XHR API 所遵循的是通用地被称为 Promise 的接口。由于 XHR 是异步的方法调用，服务器端的响应会在未知的时间和日期到达（大部分希望立刻就收到！）。Promise 接口保证如何处理这些响应，以及允许 Promise 的消费者以一种可以预知的方式使用它们。

假设我们想从服务器查询用户信息。如果 API /api/user 是可用的，接收 id 作为 URL 的参数，然后 XHR 请求使用 Angular 的核心服务 $http，就像下面一样：

```
$http.get('api/user', {params: {id: '5'}
}).success(function(data, status, headers, config) {
 // Do something successful.
}).error(function(data, status, headers, config) {
 // Handle the error
});
```

如果你使用过 jQuery，你应该注意到 AngularJS 和 jQuery 在异步请求交互上是何等类似。

我们在上面的示例中使用的$http.get 方法只是 AngularJS 核心服务$http 提供的许多便捷的方法中的一个。类似的，如果你想使用 AngularJS 发送一个想通过 ULR 参数和 POST 数据的 POST 请求，你可能会像这样做：

```
var postData = {text: 'long blob of text'};
  // The next line gets appended to the URL as params
  // so it would become a post request to /api/user?id=5
  var config = {params: {id: '5'}};
  $http.post('api/user', postData, config
).success(function(data, status, headers, config) {
  // Do something successful
}).error(function(data, status, headers, config) {
  // Handle the error
});
```

这里提供了类似的便捷方法给大部分常用请求类型，包括：

- GET
- HEAD
- POST
- DELETE
- PUT
- JSONP

进一步配置请求

有时，提供的标准请求选项是不够用的。这可能是因为你想要：

- 为请求增加一些认证头部
- 为请求变更如何处理缓存
- 以某种特定的方式，处理发出的请求，或处理收到的响应。

在这样的情形下，你可以通过传递可选的配置对象给 request 进一步配置你的请求。在先前的例子中，我们使用 config 对象来指定可选的 URL 参数。但是，即使我们使用的 GET 和 POST 方法是便捷方法，系统内部的方法调用可能像这样：

```
$http(config)
```

下面是调用该方法的一段简单的伪码模板：

```
$http({
method: string, url: string,
params: object,
data: string or object, headers:
object,
                transformRequest: function transform(data, headersGetter)
                or an array of functions,
                 transformResponse: function transform(data, headersGetter)
                 or an array of functions,
cache: boolean or Cache object, timeout:
number, withCredentials: boolean
});
```

GET，POST 和其他便捷的方法会这是 method 字段，所以你不需要设置。

把 Config 对象作为最有一个参数传递给$http.get,$http.post，因此当使用任何便捷方法时，你仍然可以使用它。

你可以通过传递 config 对象更改生成的请求，设置如下键值：

method

一个 HTTP 请求类型的字符串，比如 GET，POST

url

一个 URL 字符串，表示请求资源的绝对或相对的 URL。

params

一个字符串到字符串对象（准确的键值映射），表示键值将会转换成 URL 参数。例如：

```
[{key1: 'value1', key2:  'value2'}]
```

将转换成：

```
?key1=value1&key2=value2
```

拼接到 URL 之后。如果使用一个对象，不是字符串或数字，对于这样值，这个对象将转换成 JSON 字符串。

data

一个字符串或者对象，作为请求消息数据发送出去

timeout

在请求处理前需要等待的毫秒数

还有几个选项可以配置，我们会在下面的章节中深入讲解。

设置 HTTP 报头

AngularJS 有默认的报头，它应用与所有发出的请求，包括如下：

1. ***Accept: application/json, text/plain, /***
2. ***X-Requested-With: XMLHttpRequest***

如果你想设置任何指定的报头，有两种方式来做。

第一种方式，如果你想应用这些头部信息到每个发出的请求，为 AngularJS 做一些特定的报头。在$httpProvider.defaults.headers 配置对象中有些设置。这一步通常在构建应用的 config 部分完成。因此如果你想为你的 GET 请求启用‘DO NOT TRACK’，同时为所有请求移除了 Requested-with，你可以简单地像这样做：

```
angular.module('MyApp',[]).
config(function($httpProvider) {
  // Remove the default AngularJS X-Request-With header
  delete $httpProvider.default.headers.common['X-Requested-With'];
  // Set DO NOT TRACK for all Get requests
  $httpProvider.default.headers.get['DNT'] = '1';
});
```

如果你想为仅有的几个请求设置报头，但是不是作为默认配置，那么你可以传递 header 作为 config 对象部分给$http 服务，传递相同的自定义的报头给一个 GET 请求作为第二个参数，它也需要 URL 参数：

```
$http.get('api/user', {
// Set the Authorization header. In an actual app, you would get the auth
// token from a service
headers: {'Authorization': 'Basic Qzsda231231'}, params: {id: 5}
}).success(function() { // Handle success });
```

对于一个完整的例子，如何操作应用的中的授权，转至第八章中的 Cheatsheets 示例。

缓存响应

AngularJS 为 HTTP GET 请求提供了一中简单缓存系统。默认对所有请求是禁用的，但是为你的请求启用缓存，你需要做的是：

```
$http.get('http://server/myapi', {
cache: true
}).success(function() { // Handle success });
```

这样就启用缓存，AngularJS 存储来自服务器端的响应。下次相同 URL 的请求，AngularJS 从缓存中返回响应。缓存也是智能的，因此即使你发出了相同 URL 的多个模拟请求，只有一个请求是法相服务器的，响应是用于所有的请求。

然而，从可用性角度来看这是不和谐的，因为用户可能第一次看到的是旧数据，然后新数据突然出现了。例如，一个用户可能将要去点击一个项，然后它在他的操作下可能会改变。

请注意，响应（即使是由缓存提供的），实际上仍然是异步的。换句话说，期望你代码的行为正如它第一次发出的请求的那样

在请求和响应间做转换

通过$http 服务，AngularJS 可以在所有的请求和响应上应用一些基本的转换。这些包括：

请求转换

如果请求中的 config 对象的 data 属性包含一个对象，那么就会把它序列化成 JSON 格式

响应转换

如果检测到 XSRF 前缀，剥离它。如果检测到一个 JSON 响应，那么用 JSON 解析器序列化它。

如果你不想做一些转换，或添加自定义的，那么你可以传递函数作为 config 部分。这些

函数获取到 HTTP request/response 消息体，以及报头和序列化的响应，修改版本。使用 transformRequest,transformResponse 来设置 config 函数，他们使用模块的 config 函数中 $httpProvider 服务来配置。

我们何时使用这些？假设我们有一台切合 jQuery 做事方式的服务器。它希望我们 POST 数据从 key1=val1&key2=val2（字符串）传递过来，而不是{key1:val1,key2:val2}的 JSON 格式。虽然我们可以在每次请求中做这种变换，或添加一个 transformRequest 调用，这只是示例的目的，我们希望添加一个通用的 transformRequest，一边所有发出的请求，这种从 JSON 转成字符串的转换都会发生。下面是我们如何做到这点的：

```
var module = angular.module('myApp');
module.config(function ($httpProvider) {
    $httpProvider.defaults.transformRequest = function(data) {
        // We are using jQuery's param method to convert our
        // JSON data into the string form
         return $.param(data);

    };
});
```

单元测试

到目前为止，我们已经知道你可以用你能想到的一切方式如何使用$http 服务以及配置。但是，如何编写一些单元测试来确保它实际上是可以运行的？

正如我们多次提及的，AngularJS 是牢记测试而设计的，因此，它当然有一个模拟的后端，无论发出什么正确的请求都允许你测试，甚至可以控制如何及何时处理来自单元测试的响应。

让我们探索下，你应该如何测试一个可以向服务器发送请求的控制器，查询一些数据，把以一种特殊的格式它设置到通过视图展示的 scope 中。

NamesListCtrl 是一个非常简单的控制器，它存在的目的是：测试 name API，然后在 scope 上存储所有的 name

```
function NamesListCtrl($scope, $http) {
  $http.get('http://server/names', {params: {filter: 'none'}}).
  success(function(data) {
    $scope.names = data;

});
}
```

我们应该如何测试这个？在单元测试里，我们希望确保：

- NamesListCtrl 能够找到它所有的依赖（然后正确的注入他们）
- 控制器能够在加载时想服务器发起查询 name 的请求。

- 控制器能够正确地保存响应到作用域上的 names 变量

虽然我们可以在测试中构建控制器，然后吧 scope 和模拟的 HTTP 服务注入进去，而不是让我们在生产代码中构造同样方式的测试用例。这是推荐的方式，尽管它看上去有一点复杂。让我们看看：

```
describe('NamesListCtrl', function(){
var scope, ctrl, mockBackend;

// AngularJS is responsible for injecting these in tests
beforeEach(inject(function(_$httpBackend_, $rootScope, $controller) {
  // This is a fake backend, so that you can control the requests
  // and responses from the server
  mockBackend = _$httpBackend_;
// We set an expectation before creating our controller,
  // because this call will get triggered when the controller is created
      mockBackend.expectGET('http://server/names?filter=none').
      respond(['Brad', 'Shyam']);
  scope = $rootScope.$new();

  // Create a controller the same way AngularJS would in production
  ctrl = $controller(PhoneListCtrl, {$scope: scope});
}));

it('should fetch names from server on load', function() {
  // Initially, the request has not returned a response
  expect(scope.names).toBeUndefined();

  // Tell the fake backend to return responses to all current requests
  // that are in flight.
  mockBackend.flush();
  // Now names should be set on the scope
  expect(scope.names).toEqual(['Brad', 'Shyam']);
});
});
```

使用 RESETful 资源

$http 服务提供了一个非常底层的实现，允许你发送 XHR 请求，但是仍然给你了很多控制和灵活性。但是在大部分场景下，我们处理对象以及对象模型是有特定的属性和方法的，比如 person 对象（就有详细）或者一个信用卡对象。

在这样的情形下，如果我们创建一个了 JS 对象，代表了这个对象模型，可能不是很好？如果只是编辑了对象属性，比如说保存或更新，如何让状态持久化到服务器上。

$resource 提供了这样的能力，AngularJS 资源允许我们以描述的方式定义对象模型，从而指定：

- Resource 的服务端 URL
- 这类请求的常见参数类型
- 一些额外的方法（自由的使用 get,save,query,remove,delete）为对象模型封装特定的函数功能和业务逻辑。
- 期望的响应类型（数组或对象）
- 报头

何时使用 Angular Resource

如果服务端有 RESTful 方式的行为，那门你应该使用 Angular Resource。

对于本章中举的信用卡场景：

1. /user/123/card 的 GET 请求，返回用户 123 的信用卡列表
2. /user/123/card/15 的 GET 请求，返回用户 123 的编号为 15 的信用卡
3. /user/123/card POST 中带信用卡信息的 POST 请求，为用户 123 创建一张新信用卡
4. /user/123/card/15 带信用卡信息的 POST 请求，为用户 123 更新编号为 15 的信用卡
5. /user/123/card/15 的 DELETE 请求，为用户 123 删除编号为 15 的信用卡

除了提供对象，允许你按你的需求查询服务器，$resource 也允许你处理返回值，就好像他们是持久化的数据模型，做变更，然后要求持久化。

ngResource 是一个独立的，可选的模块，为了使用它，你需要：

- 包含 angular-resource.js
- 在模块声明中包含 ngResource（例如，angular.module('myModule',['ngResource'])）
- 在需要的地方注入$resource

在我们看看 ngResource 方法如何创建一个资源前，先看看它和$http 创建有什么类似的地方。对于信用卡资源，我们希望能够获取，查询，保存，另外还能够修改。

这里有个实现：

```
myAppModule.factory('CreditCard', ['$http', function($http) {
  var baseUrl = '/user/123/card';
  return {
    get: function(cardId) {
      return $http.get(baseUrl + '/' + cardId);
    },
    save: function(card) {
      var url = card.id ? baseUrl + '/' + card.id : baseUrl;
return $http.post(url, card);
},
  query: function() {
    return $http.get(baseUrl);
  },
  charge: function(card) {
    return $http.post(baseUrl + '/' + card.id, card, {params: {charge: true}});
```

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

相反，你可以简单地创建 Angular 服务来映射应用中的资源，像这样：

```
myAppModule.factory('CreditCard', ['$resource', function($resource) {
  return $resource('/user/:userId/card/:cardId',
        {userId: 123, cardId: '@id'},
        {charge: {method:'POST', params:{charge:true}, isArray:false});
}]);
```

现在，只要我们注入 CreditCard，我们就能获取到 Angular resource 默认给出了几个方法。表 5-1 列出了这些方法，他们有什么样的行为，因此你能够知道服务器能够应该如何配置。

表 5-1 信用卡资源

Resource 函数	方法	URL	返回值
CreditCard.get({id: 11})	GET	/user/123/card/11	Single JSON
CreditCard.save({},card)	POST	/user/123/card with post data ‘card’	Single JSON
CreditCard.save({id: 11},card)	POST	/user/123/card/11 with post data ‘card’	Single JSON
CreditCard.query()	GET	/user/123/card	JSON Array
CreditCard.remove({id: 11})	DELETE	/user/123/card/11	Single JSON
CreditCard.delete({id: 11})	DELETE	/user/123/card/11	Single JSON

这里举个信用卡例子，会是事情变得更加清晰：

```
// Let us assume that the CreditCard service is injected here
// We can retrieve a collection from the server which makes the request
// GET: /user/123/card
var cards = CreditCard.query();
// We can get a single card, and work with it from the callback as well
CreditCard.get({cardId: 456}, function(card) {
// each item is an instance of CreditCard
expect(card instanceof CreditCard).toEqual(true);
card.name = "J. Smith";
// non-GET methods are mapped onto the instances
card.$save();// our custom method is mapped as well.
card.$charge({amount:9.99});
// Makes a POST: /user/123/card/456?amount=9.99&charge=true
// with data {id:456, number:'1234', name:'J. Smith'}
```

```
});
```

在上面的示例中发生了很多事情，我们会依次讲述重要的部分：

声明

申明$resource 和使用正确的参数调用注入的$resource 函数一样简单（到现在你应该知道如何注入了，对吧？）

$resource 函数需要一个必填参数 URL，代表了可以访问的资源，以及两个可选参数：默认参数和额外的你想在资源上配置的操作。

注意 URL 是带参数的（注意：参数，:user 代表了会用 userId 参数来替换它，：cardId 会被 cardId 替换）。如果没有传递参数，那么它会被空字符串替换。

第二个参数是传递默认的参数到每个请求中。在这个示例中，我们给 userId 传递了一个常量 123。carId 参数更加有意思，是”@id”。这表示，如果我使用了一个来自服务器端的返回值对象，那么在它上面调用任何方法（比如$save），那么 cardId 字段会从对象的 id 属性中取值。

第三个参数是其他我们想暴露自定义资源上的方法。这个我们会在下一节深入讨论。

自定义方法

$resource 调用的第三个参数是可选的，可暴露资源上的额外方法 。

在这种情况下，我们指定一个充值方法。通过传递一个对象进行配置，key 就是暴露的方法名称。 配置需要指定请求方法类型（GET,POST 等等）需要传递参数作为请求的一部分（这里是 charge=true），返回结果可能是一个数组或者不是（这里不是）。一旦这里配置完成，只要你想你就可以自由调用 CreditCard.charge()（当然，显示中只要用户充值！）

没有回调机制（除非你真的需要他们）

需要注意的第三件事就是 resource 调用的返回类型。回头看看 CreditCard.query()调用。你会看到我们直接把他们复制到 card 变量，而不是在回调函数中指定 card。发起异步服务器请求，那段代码可以运行吗？

你可能会担心代码是否能够正确运行，但是实际上代码是正确的，能够运行。这里发生就是 AngularJS 指派了一个引用（一个对象或者一个数组，依赖于期望的返回类型），在未来某一时刻服务器返回时，将会填充它。在此期间，对象仍然是空的。

由于 AngularJS 应用最通用的流程就是从服务器查询数据，指派给变量，在模板中显示，这个捷径是非常好的。在控制器中代码中，你所要做的就是发起服务端调用，指派返回值给正确的作用域变量，然后让模板去负责当它返回时渲染它。

如果你在返回值上执行一些业务逻辑，那么这种方法是不会有效的。在这个例子中，你必须依赖返回函数，在 CreditCard.get()使用的那种。

简单的服务端操作

无论你使用简洁的返回类型或者回调函数，关于返回对象有几点你需要注意的。

返回值不是一个简单的旧 JS 对象，实际上是一个 resource 类型对象。这意味这除了服务器返回值，还有一些额外的行为绑定到它上面（这里有$save(),$charge）。这就允许你很容易地执行服务端操作，例如通过查询数据，做一些变更，以及持久化变更到服务端（在任何 CRUD 应用中这是常见的行为）

单元测试 ngResource

ngResource 是一个封装，在底层使用的 AngularJS 的$http。因此，你已经知道如何经行单元测试它。从我们之前看到的$http 单元测试，无需任何修改。你只需要知道由 resource 来发起最终的请求的，通知模拟的$http 服务，其他的一切都应该是一样的。让我们看看上面的代码的一个测试用例：

```
describe('Credit Card Resource', function(){
var scope, ctrl, mockBackend;

beforeEach(inject(function(_$httpBackend_, $rootScope, $controller) {
mockBackend = _$httpBackend_;
scope = $rootScope.$new();
  // Assume that CreditCard resource is used by the controller
  ctrl = $controller(CreditCardCtrl, {$scope: scope});
}));

it('should fetched list of credit cards', function() {
  // Set expectation for CreditCard.query() call
      mockBackend.expectGET('/user/123/card'). respond([{id: '234',
      number: '11112222'}]);

  ctrl.fetchAllCards();

  // Initially, the request has not returned a response
  expect(scope.cards).toBeUndefined();

  // Tell the fake backend to return responses to all current requests
  // that are in flight.
  mockBackend.flush();

  // Now cards should be set on the scope
```

```
  expect(scope.cards).toEqualData([{id: '234', number: '11112222'}]);
});
});
```

这个测试用例极其类似与$http 单元测试，处理几个微小的差别。请注意在我们的期望中，我们使用的是 toEqualData，而不是简单的 equals。这个期望足够的智能能够忽略 ngResource 添加的额外的方法。

$q 和 Promise

到目前为止，我们已经知道 AngularJS 如何实现了它的异步，延迟 API。Promise 计划就是 AngularJS 如何组织它的 API 的基础。在底层，Promise 计划做了如下异步请求规定：

- 异步请求返回一个 promise 而不是返回值
- Promise 有一个 then 函数，它有两个参数，一个函数处理 resolved 或者 success 事件，一个函数处理 rejected 或者 failure 事件。这些函数调用时会带上结果或者拒绝的原因。
- 只要执行结果可用，就可以保证必定会调用两个会回调函数中一个

虽然大部分 deferred/Q 实现采用这种途径，但是 AngularJS 的实现因为下面的原因而有些特别：

- $q 对于 AngularJS 是可见的，因此可以和作用域模型集成。这样返回数据就能快速传递，UI 中的闪烁更新也就更少
- AngularJS 模板也知道$q promise，因此他们可以看成他们自己的返回值而不是 promise，这种 promise 会在返回结果时得到通知
- 更小的作用域，因为 AngularJS 实现了常用异步交互的基本的，最重要的函数功能

你可能会问，为什么做这么疯狂的事？，让我们看看一个标准的问题，你可能在异步调用是遇到的：

```
fetchUser(function(user) {
fetchUserPermissions(user, function(permissions) {
  fetchUserListData(user, permissions, function(list) {
    // Do something with the list of data that you want to display
  });
});
});
```

这就是人们抱怨的使用 JavaScript 时的嵌套噩梦。返回值异步特性和代码的同步需求发生了竞争，导致了多个嵌套函数，使追踪当前的上下文变得更加困难。

此外，还关系到错误处理事宜。处理错误的最好方法是什么？在每一步中处理它？那也会变的混乱。

为了修正这个问题，Promise 计划提供了 then 概念，它会在成功时执行一个函数，另一方面，出错是执行另一个函数，每个都可以用链接起来（链式写法）。因此，带 Promise API

（至少是 AngularJS 的实现）的示例，会平坦很多：

```
var deferred = $q.defer();
var fetchUser = function() {
  // After async calls, call deferred.resolve with the response value
  deferred.resolve(user);

  // In case of error, call
  deferred.reject('Reason for failure');
}

// Similarly, fetchUserPermissions and fetchUserListData are handled
deferred.promise.then(fetchUser)
  .then(fetchUserPermissions)
  .then(fetchUserListData)
  .then(function(list) {
    // Do something with the list of data
}, function(errorReason) {
// Handle error in any of the steps here in a single stop
});
```

整个代码优雅地整齐了，而且提供了链式的作用域，以及一个单一的错误处理。在应用中处理异步请求回调时也可以用相同的代码，只要调用 Angular 的$q 服务。这种机制可以帮我们做一些很酷的事情：比如响应拦截

响应拦截

我们已经讲述了向服务器发起请求，处理响应，优雅地抽象封装响应，以及处理异步调用。但是在真实应用中，你需要为每个向服务器的请求都做一些通用的操作，比如错误处理，认证，像修剪数据这样的其他安全考虑。

在$q API 深刻的理解上，我们可以通过使用响应拦截来处理所有的事。响应拦截允许（或者建议）你在发给应用前拦截响应，应用数据转换，错误处理，以及其他东西，包括数据清洗。

让我们看个示例，拦截响应以及做了一些数据转换。

```
// register the interceptor as a service
myModule.factory('myInterceptor', function($q, notifyService, errorLog) {
 return function(promise) {
   return promise.then(function(response) {
     // Do nothing
     return response;
   }, function(response) {
     // My notify service updates the UI with the error message
```

```
      notifyService(response);
      // Also log it in the console for debug purposes
      errorLog(response);
      return $q.reject(response);
    });

  }
});
```

安全考虑

现在，当运行在 WEB 应用中是，安全是一个巨大的概念，应该牢记在心。当谈到两种常见的攻击方式时，AngularJS 确实提供一些帮助，我们会在下面的章节中讲述。

JSON 漏洞

有一个非常微妙的 JSON 漏洞，被暴露出，当发起一个 GET 请求时获取到 JSON 信息作为数组（尤其是这些信息是敏感的，需要登录凭据或认证才能访问）。

这个漏洞涉及恶意站点使用<script>标签来发送相同信息的请求。因为你仍然是登录的，恶意站点使用你的凭据请求 JSON 信息，然后获取它。

你可能想知道如何做的，因为这些信息仍然在你客户端，服务器有不能处理那些消息。通常返回的 JSON 对象作为一个 script 脚本的来源，可能会导致一个错误，虽然数组是一个例外。

但是这里的漏洞是在这里：在 JavaScript 中，重写或重申明内置的对象是有可能的。在这个漏洞中，重新定义了数组构造器，在这个重新定义过程中，恶意站点可以获取数据上一个句柄，然后发到它自己的服务器上。

有两种方式了可以防止这种漏洞：始终确保敏感信息只作为 POST 请求的响应通过 JSON 发出去；二是返回一个对象或者非法的 JSON 表达式作为结果，然后让客户端调用把它转成实际数据。

AngularJS 允许你使用这两种方法来阻止这种漏洞。在应用中，你可以（应该）选择通过只有 POST 请求才能接收 JSON 信息。

此外，你可以配置服务器前缀：

```
")]}',\n"
```

在所有 JSON 响应之前，因此，一个正常响应：

```
['one','two']
```

应该返回如下：

```
)]}',
['one', 'two']
```

AngularJS 会自动丢弃前缀，然后仅处理 JSON

XSRF

XSRF(Cross-Site Request Forgery，跨站点请求伪造)攻击通常具有如下特征：

- 它们涉及依靠认证或者用户标识的站点
- 它们利用用户长时间登录和认证的站点漏洞
- 它们发起伪造的 HTTP/XHR 请求，通常是有害的

考虑下面的 XSRF 攻击示例：

- 用户 A 登陆了他的银行账户（http://www.exampleblank.com）
- 用户 B 知道了这个，让用户 A 访问用户 B 的主页
- 主页上有个特制的图片链接能够触发 XSRF 攻击

```
<img src="http://www.examplebank.com/xfer?from=UserA&amount=10000&to=UserB">
```

如果用户 A 账户保存认证信息保存在 cookie 中，它还没有过期，然后当用户 A 打开用户 B 的站点，它会触发一个未授权的从用户 A 到用户 B 的交易。

那么 AngularJS 是如何帮助我们阻止这个？它提供了双步机制来阻止 XSRF 漏洞。

在客户端，当执行 XHR 请求时，$http 服务从 cookie 中读取一个叫 XSRF-TOKEN 的令牌，然后把它设置到 HTTP 报头 X-XSRF-TOKEN 中。由于只有从你的作用域才能读取和设置令牌，那门你就可以确认 XHR 是来自你的作用域。

这也需要修改服务端代码，以便在第一次 HTTP GET 请求时能设置一个叫 XSRF-TOKEN 的可读的会话 cookie。后续的服务器请求就能够校验 HTTP 报头中第一次设置的 XSRF 令牌值。当然令牌对每个用户来说必须是唯一的，必须通过服务器端校验（从而防止 JavaScript 伪造令牌）

第六章：指令

使用标识符，你可以通过增加声明语法扩展 HTML 从而做任何你想做的事。通过这个，你可以替换通用的 ,元素和属性，这实际上对应用是有特定意义的。AngularJS 提供了基本的函数功能，但是你可以创建你自己的来做一些特定于应用的事。

首先，我们会复习下标识符 API，以及在 Angular 中如何启动和运行的生命周期。从这开始，我们会用这些知识创建一些标识符类。我们会通过如何为标识符编写单元测试以及如何是它们运行的更快来完成本章任务。

但首先，使用标识符语法的一些注意点

指令和 HTML 校验

纵观本书，我们已经使用了 Angular 内嵌 ***ng-指令名*** 语法的标识符。例如 ng-repeat,ng-view,ng-controller。这里 ng 是 Angular 的命名空间，破折号后面是标识符名称。

虽然我们选择了这种语法容易输入，但是在许多 HTML 校验方案中它是不合法的。为了支持这种语法，Angular 用多种方式包含任何标识符。表 6-1 中的语法，它们都是等价的，能够让你爱好的校验器正常运作：

表 6-1 HTML 校验方案

校验器	格式	示例
None	Namespace-name	Ng-repeat=item in items
XML	Namespace:name	Ng:repeat=item in items
HTML5	data-namespace-name	Data-ng-repeat=item in items
xHTML	x-namespace-name	x-ng-repeat=item in items

因为你可以使用这些当中的任何，Angular 文档列出了驼峰风格的标识符，而不是这些任何选项。例如在 ngRepeat 标下是 ng-repeat。正如你看到的，当定义我们自己的标识符时使用这种命名格式

如果你不适用 HTML 校验器（大多数人不会这样做），到目前为止你所看到的使用***命名空间-指令***的语法，你会感觉很好。

API 预览

创建标识符的一段简单伪码模板：

```
var myModule = angular.module(...);
myModule.directive('namespaceDirectiveName', function factory(injectables) {
 var directiveDefinitionObject = {
   restrict: string, priority: number,
```

```
    template: string, templateUrl:
    string, replace: bool, transclude:
    bool, scope: bool or object,
    controller: function controllerConstructor($scope,
                                               $element,
                                               $attrs,
                                               $transclude),

    require: string,
    link: function postLink(scope, iElement, iAttrs) { ... }, compile:
    function compile(tElement, tAttrs, transclude) {
      return {
        pre: function preLink(scope, iElement, iAttrs, controller) { ... }, post: function
        postLink(scope, iElement, iAttrs, controller) { ... }

      }
    }
  };

  return directiveDefinitionObject;
});
```

有些是互斥的，大部分是可选的，他们所有都值得详细解释的。

表 6-2 提供了你使用的选项的预览。

表 6-2 标识符定义选项

属性	含义
restrict	申明标志符在模板中作为元素，属性，类，注释，或组合，如何使用
Priority	设置模板中相对于其他标识符的执行顺序
Template	制定一个字符串式的内嵌模板。如果你指定了模板是一个 URL 那么是不会使用的
TemplateUrl	指定通过 URL 加载的模板。如果你已经指定了内嵌的模板字符串，那么它是不会使用的
Replace	如果为真，替换当前元素。如果是假或未指定，拼接标识符到当前元素
Transclude	能够让你移动一个标识符的原始子节点到一个新模板的位置
Scope	为这个标识符创建一个新的作用域，而不是继承父作用域
Controller	创建一个控制器通过标识符公开通信 API
Require	当前标识符需要另外一个标识符提供正确的函数功能
Link	通过代码修改目标 DOM 元素的实例，添加事件监听，建立数据绑定
Compile	通过标识符拷贝编程修改 DOM 模板，就行使用 ng-repeat。你的编译函数同样可以返回修改目标元素实例的 link 函数

批注 [spy24]: 可暂时翻译成‘嵌入包含’

让我们深入到细节中。

命名指令

你可以通过模块的标识符函数为标识符命名，就像下面这样：

```
myModule.directive('directiveName', function factory(injectables)
```

尽管你可以吧标识符命名成你喜欢的，但是约定是选择一个前缀标识符来标识你的标识符，防止和工程中外部的标识符命名冲突。

当然，如果你不希望用 ng-前缀，可能和 Angular 内部的标识符起冲突。如果你在 suuperDuper Megacorp 工作，那么你可以选择 super-,superduper-,甚至 superduper-megacorp-，你可能会选择第一个因为它简单易输入。

正如之前指出的，Angular 为标识符使用一个标准化的命名方案，在模板中使用驼峰方式的标识符命名变量，可以在五种不同的校验下运作。例如，如果你使用 super-作为前缀，那么你编写日期控件时，你可以这么命名 superDatePicker。在模板中，你可以使用 super-date-picker，super:date-picker，data-super-date-picker 或其他形式。

指令定义对象

正如之前提到的，标识符定义中的大部分选项是可选的。实际上，没有硬性要求，你可以不适用这些参数，构造有用的标识符。让我们看看这些选项都是做什么的。

restrict

restrict 属性允许为标识符指定声明样式，也就是说，它可以做为元素名，属性，类或注释。你可以使用一个字符来代表表 6-3 中的每个，从而指定一个或多个申明样式

表 6-3 标识符声明用法选项

标志	样式	示例
E	Element	<my-menu title='products'></my-menu>
A	Attribute	
C	Class	
M	Comment	<!—directive: my-men products-->

如果你希望标识符作为元素或者属性，你可以传递 EA 作为 restrict 的字符串。

如果你省略了 restrict 属性，***默认就是 A***，那么标识符只能作为属性。

如果你计划支持 IE8，那么基于 attribute-和 class-的标识符是最佳选择，因为它需要额外的努力才能使新元素正常工作。详细信息查看 Angular 文档。

Priorities

有些场景中，在单个 DOM 上有多个标识符，它们应该以哪种顺序应用到事物上，你可

以使用 priority 属性为应用指定顺序。数值越大就越先运行。如果你不指定，那么默认值是 0。

需要设置优先级的场景是罕见的。一个需要设置优先级的标识符场景是 ng-repeat。当重复元素时，我们希望 Angular 能够在其他标识符应用前生成模板元素的拷贝。如果没有这个，其他标识符会应用到标准模板元素中，然不是重复应用中我们希望的元素。

尽管它不在文档中，你用 priority.在 Angular 源码中搜索下其他几个标识符。对于 ng-repeat，使用的 priority 值是 1000， 因此剩余很多空间给在它下面的优先级。

Templates

当创建组件，widget，控制器等等，angular 允许你在你提供的模板中替换和包装元素中内容。例如，如果你准备在界面上创建一系列的标签视图，那么你希望呈现类似图 6-1 这样的

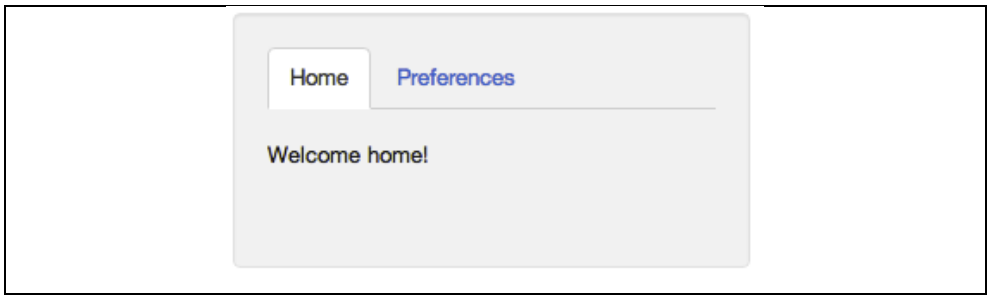


图 6-1 选项卡视图

不是使用一串的 <a>等元素，你可以创建<tab-set><tab>标识符，分别声明了每个页签的结构。那么，你的 HTML 可以做更好的工作来表单模板的用处。结果可能像这样：

```
<tab-set>
<tab title='Home'>
  <p>Welcome  home!</p>
</tab>
<tab title='Preferences'>
  <!-- preferences  UI goes here -->
</tab>
</tabset>
```

你也可以通过<tab><tab-set>上的控制器为 title 和页签内容进行数据绑定。这并不现定于页签，你也可以用这种方式做菜单、手风琴、弹窗，对话框或其他应用需要的。

通过 template 或者 tempateUrl 属性，可以指定替换掉的 DOM 元素。你应该使用 template 来设置模板内容的字符串，templateUrl 指向将被加载的服务器文件。正如你接下来看到的示例，你可以预缓存这些模板，以便减少 GET 请求数，潜在的提高性能。

让我们写个简单的标识符，<hello>元素只是替换了 hi there 。在它里面，我们会设置 restrict 允许元素，设置 template 成我们想展示的。由于默认呢行为是拼接内容到元

素上，设置 replace 成 true，替换原始的模板。

```
var appModule = angular.module('app', []);
appModule.directive('hello', function() {
  return {
    restrict: 'E',
    template: '<div>Hi there</div>', replace: true
  };
});
```

在一个页面中像这样使用：

```
<html lang='en' ng-app='app'>
---
<body>
  <hello></hello>
</body>
---
```

加载到浏览器，我们会看到‘Hi there’。

如果你查看页面源码，你看到的仍然是<hello></hello>。但是如果你检查生成的代码（chrome，右击 hi there，选择‘检查元素’），你会看到：

```
<body>
<div>Hi there</div>
</body>
```

<hello></hello>已经被模板中的 替换。

如果你去掉标识符定义中的 replace：true,，你会看到

```
<hello><div>Hi there</div></hello>.
```

通常，你会使用 templateUrl 代替 template，输入 HTML 到字符串中不是很有意义。Template 属性对非常小的模板有意义。写 templateUrl 是有用的，因为这些模板可以通过设置适当的头部缓存。我们可以重写我们 hello 标识符示例就像这样：

```
var appModule = angular.module('app', []);
appModule.directive('hello', function() {
  return {
    restrict: 'E',
    templateUrl: 'helloTemplate.html', replace: true
  };
```

```
});
```

在 helloTemplate.html 中，你需写入：

```
<div>Hi there</div>
```

如果你使用 chrome 浏览器，同源策略会组织 chrome 加载 file://这里类模板，然后你看到一个错误“Origin null is not allowed by Access-Control-Allow-Origin”。这里你有两种可选方式：

- 通过服务器加载应用
- Chrome 中设置一个标志。你可以通过命令行‘chrome –allow-file-access-from-files’解决

然而，通过 templateUrl 加载文件，会使用户等待直到加载后看到标识符。如果你希望在第一次页面加载时就加载模板，你可以在 script 标签中作为页面的一部分，就像这样：

```
<script type='text/ng-template' id='helloTemplateInline.html'>
<div>Hi there</div>
</script>
```

这里 Id 属性是非常重要的，因为这是 URL 键，Angular 用它来存储模板。你应该在标识符的 templateUrl 中使用 id 来指定插入哪个模板。

这个版本如果没有服务器那么加载的很好，因为没有 XMLHttpRequest 需要查询内容。

最后，你可能通过$http 或其他几机制加载模板，然后直接设置到 Angular 所使用的叫 $templateCache 对象中。我们希望在标识符运行前让这个模板在缓存中是可用的，因此我们会在模块中通过 run 函数调用它。

```
var appModule = angular.module('app', []);
appModule.run(function($templateCache) {
  $templateCache.put('helloTemplateCached.html', '<div>Hi there</div>');
});
appModule.directive('hello', function() {
  return {
    restrict: 'E',
    templateUrl: 'helloTemplateCached.html', replace: true
  };
});
```

你可能会在产品中这样做，因为这种基础可以减少 GET 请求数。你可能会加载所有的模板到单个文件，在新木块中加载它，然后在主应用模块中引用。

Transclusion（嵌入包含）

批注 [spy25]: 嵌入包含（transclusion）通常是指将一份文档以包含（inclusion）的方式置入另一份文档之中以作为参考文献

http://zh.wikipedia.org/wiki/Wikipedia:%E5%B5%8C%E5%85%A5%E5%8C%85%E5%90%AB

除了替换和拼接内容，你也可以通过 transclude 属性移动原始的内容到新模板中。当设置成 true 时，标识符会删除原始的**内容**，但是通过 ng-transclude 标识符使它重新插入到模板中。

采用 transclusion 方式修改示例：

```
appModule.directive('hello', function() {
return {
  template: '<div>Hi there <span ng-transclude></span></div>', transclude:
  true
};
});
```

应用在：

```
<div hello>Bob</div>
```

我们会看到‘Hi there Bob.’

编译和链接函数

虽然插入模板是有用，但是任何标识符真正有意义的工作发生在编译或者链接功能里。

编译和链接功能是 Angular 为应用创建实时视图的后两阶段。让我们看下 Angular 初始化过程的高层次视图，按照次序：

脚本加载

加载 Angular，查找 ng-app 标识符找到应用绑定

编译阶段

在这一阶段，Angular 遍历 DOM 标识模板中所有注册的标志。对于每个标识符，基于标识符规则（template，replace，transclude 等等）改造 DOM，然后如果编译函数存在就调用它。结果是一个编译的 template 函数，它会调用从所有的标志符中搜集的 link 函数，

链接阶段

为了让视图动起来，然后 Angular 为每个标识符运行 link 函数。Link 函数通常在 DOM 或模型上创建监听器。这些监听器让视图和模型始终保持一致。

因此到编译阶段，它处理了转换模板，链接阶段，它处理了修改了视图中的数据。沿着这些思路，标识符中编译功能和链接功能的主要区别就是编译功能转换了模板自身，而链接功能在模型和视图上创建了动态链接。就是在第二阶段，作用域 scopes 被附加到了编译过的 link 功能上，通过数据绑定，标识符变活了。

出于性能原因，这两阶段是分开的。编译功能在编译阶段只执行一次，然而链接功能是

执行多次的，为标识符的每个实例运行一次。例如，在上面的标识符中使用 ng-repeat。你不希望调用 compile，它会引起每个 ng-repeat 迭代上的 DOM 遍历。相反，你只需要编译一次，然后链接。

虽然你当然应该学习编译、链接和每个功能的不同，但是你需要编写的大多数指令是不需要修改模板的；大部分是编写链接功能。

先来看看比较小每个语法。对于编译，我们有：

```
compile: function compile(tElement, tAttrs, transclude) {
return {
pre: function preLink(scope, iElement, iAttrs, controller) { ... },
post: function postLink(scope, iElement, iAttrs, controller) { ... }
}
}
```

对于链接，它是这样：

```
link: function postLink(scope, iElement, iAttrs) { ... }
```

请注意，这里一个不同点就是 link 函数可以访问作用域 scope，但是 compile 函数不能。这是应为在编译阶段，作用域 scope 还未存在。然而，你能够从 compile 函数中返回 link 函数。这些 link 函数确实能够访问到作用域 scope。

请注意，compile 和 link 都可以接收到 DOM 元素的引用，以及这些元素的属性列表。这里不同的是 compile 函数接收到的是 template 元素及其属性，因此获取的是 t 前缀。Link 函数从模板创建的视图实例中接收他们，因此获取的是 i 前缀。

这种区别只有当标识符在其他标识符中，它又拷贝了模板时才有影响。

```
<div ng-repeat='thing in  things'>
  <my-widget  config='thing'></my-widget>
</div>
```

这里，编译函数只调用了一次，但是 link 函数在每次拷贝 my-widget 时，等于 things 中元素的数目。如果 my-widget 需要修改 my-widget 所有（实例）拷贝的公共部分，处于效率原因，做这事的最好地方就是在编译函数中。

你也注意到了 compile 函数接收一个 transclude 函数属性，这里，你有机会编写函数，在简单模板的嵌入修改不满足需求这种场景下可以让程序修改内容。

最后，compile 能同时返回 preLink 和 postLink 函数，然而 link 指定只有 postLink 函数。preLink，正如它命名的，在编译阶段结束后执行，但是在子元素上的标识符链接前。类似的，postLink 在所有子元素标识符链接后运行。这意味着，如果你需要修改 DOM 结构，你应该在 postLink 中做。在 prelink 中做着，会混淆链接过程，引发错误。

作用域

你经常希望从标识符中访问作用域监控模型的值，当它们发生变化时更新 UI，以及当外部事件导致模型值发生变化时通知 Angular。当你正在包装一些非 Angular 组件如 jQuery，closure，其他类库，或实现了简单的 DOM 事件。把执行 Angular 的表达式作为属性传递到标识符中。

当你因为这些原因需要作用域时，你对获取的作用域 scope 有三种选择：

1. 标识符 DOM 元素中**已存在作用域**
2. 创建一个继承封闭的控制器作用域的**新作用域**。这里，你可以读取到结构树作用域的所有值。这个作用域可以和 DOM 元素上的请求同样的类型的作用域其他标志符共享，互相通信。
3. 独立作用域，从父类中不继承任何属性。当你需要隔离这个标识符的操作和父类作用域时，创建可重用的组件可以使用这个选项。

你可以用如下语法创建这些作用域配置：

作用域类型 scope type	语法
已有作用域	scope:false(如果没有指定，这就是默认值)
新作用域	scope:true
独立作用域	scope:{属性名称和绑定风格}

当你创建一个独立作用域是，默认情况下，是不能访问父类作用域的模型。然而，你可以指定，你需要的属性传递到标识符。你可以认为属性名作为参数传递给函数。

注意：虽然独立作用域并没有继承模型属性，但是它们仍然是他们父作用域的子节点。和其他作用域一样，它们有$parent 属性指向父类。

你可以通过标识符属性名的键值对从父类传递指定的属性给独立作用域。这里有三种可行的方式从父作用域传输数据。我们称这些传递数据的方式叫‘绑定策略’。你也可以为这个属性名称指定一个本地别名。

没有别名的语法如下：

```
scope: { attributeName1: 'BINDING_STRATEGY',
         attributeName2: 'BINDING_STRATEGY', ...

}
```

用别名的格式如下：

```
scope: { attributeAlias: 'BINDING_STRATEGY' + 'templateAttributeName',
...
}
```

表 6-4 中通过符号定义绑定策略：

表 6-4 绑定策略

符号	意义
@	传递字符串属性。你可以通过使用改写{{}}属性值从封闭作用域中进行数据绑定
=	数据绑定属性在标识符父作用域的属性中
&	传递一个来自父作用域的函数，稍后调用的

这些是相当抽象的概念，因此让我们看看一个具体例子上的变化来说明它们。比如说，我们想创建一个 expander 标识符，展示一个标题栏，当点击时扩展显示额外的内容。

关闭时如图 6-2

图 6-2 关闭状态的 expander

打开时如图 6-3 所示：

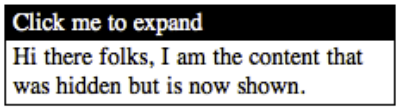


图 6-3 打开状态下的 Expander

代码如下：

```
<div ng-controller='SomeController'>
<expander class='expander' expander-title='title'>
  {{text}}
</expander>
</div>
```

标题（Click me to expand）和内容（Hi there folks…）的值，来自闭合作用域。我们会编写一个像这样的控制器：

```
function SomeController($scope) {
$scope.title = 'Click me to expand';
$scope.text = 'Hi there folks, I am the content
                                    + 'that was hidden but is now shown.';
}
```

然后，我们编写标识符如下：

```
angular.module('expanderModule', [])
.directive('expander', function(){
  return {
    restrict: 'EA', replace: true,
```

```
    transclude: true,
    scope: { title:'=expanderTitle' }, template:
    '<div>' +
        '<div class="title" ng-click="toggle()">{{title}}</div>' +
        '<div class="body" ng-show="showMe" ng-transclude></div>' +
        '</div>',
    link: function(scope, element, attrs) {
      scope.showMe = false;
scope.toggle = function toggle() {
  scope.showMe = !scope.showMe;

}
}
}
});
```

以及样式：

```
.expander {
border: 1px solid black;
width: 250px;
}
.expander > .title {
  background-color: black;
  color: white; padding: .1em .3em;
  cursor: pointer;

}
.expander > .body {
  padding: .1em .3em;
}
```

让我们看一看标识符中每个选项的含义，如表 6-5 所示：

表 6-5 元素的功能

功能名	描述调用标识符为元素或属性，也就是<expander>…</expander>或<div expander> ….
Restrict:EA	
Replace:true	用提供的模板替换原先的元素
Transclude:true	移动原始元素内容到提供的模板中的另外一个地方
Scope:{title:=expanderTitle}	创建一个叫 title 的本地作用域属性，它是用来数据绑定到在 expander-title 属性中声明的 parent-scope 属性。这里，为了方

	便，将 expanderTitle 重名为 title。由于 expanderTitle 在模板中，我们编写作用域 scope：{expanderTitle:'='}来引用它。但是在这个场景中，其他标识符也有一个 title 属性。为了防止引起歧义，为了这里使用它我们只是重名了它。同时注意，这里使用的命名和标识符命名一样使用驼峰式
Template: +….	为标识符提供即将插入的模板。注意，我们使用 ng-click 和 ng-show 来展示或隐藏自身，ng-transclude 来声明原始的内容将何去何从。同时注意，嵌入的内容可以访问父作用域，而不是封闭标识符的作用域
Link:….	建立 showMe 模型，跟踪 expander 的打开和关闭的状态，然后当用户点击 title div 时，调用定义的 toggle 函数

如果我们认为在模板中定义 expander 更有意义，而不是在模型中，我们可以在作用域声明中通过@标识符传递字符串样式的属性，就像这样：

```
scope: { title:'@expanderTitle' },
```

在模板中，我们可以用这个达到相同的效果：

```
<expander class='expander' expander-title='Click me to expand'>
{{text}}
</expander>
```

注意，通过@策略，我们仍然可以通过使用重写把 title 绑定到控制器作用域：

```
<expander class='expander'  expander-title='{{title}}'>
{{text}}
</expander>
```

操作 DOM 元素

传递到标识符的 link 和 compile 函数的参数 iElement 和 tElement 是本地 DOM 元素的包装引用。如果你已经加载了 jQuery 类库，这些就是你曾经使用的 jquery 元素。

如果你不适用 jquery，元素是 Angular 本地包装的叫 jqLite。这个 API 是 jquery 的一个子集，便于我们在 Angular 中创建任何东西。对许多应用，你可以单独使用这些 APPI 做你想做的一切。

如果你需要访问原生的 DOM 元素，你可以通过用对象的第一个元素 element[0]来获取。

你可以在 angular.element()的 Angular 文档中查看到所有支持的 API 列表，你应该使用它来独自创建 jqLite 包装的 DOM 元素。它包括了想 addClass(),bind(),find(),toggleClass()等等。再者，有一些来自 jquery 的大部分有用的核心函数，但是代码量更少。

除了 jquery API，元素也有 Angular 特定的函数。这些都是存在的，无论你是否使用完

整的 jquery 类库。

表 6-6 元素上指定的 Angular 函数

函数	描述
Controller(name)	当你需要和控制器直接进行通信时，这个函数返回绑定在元素上的控制器。如果这个元素上不存在，它会遍历 DOM，然后查找最近的父控制器代替。参数的名字是可选的，用于指定同一元素上其他标识符的名称。如果提供了，它会返回标识符上的控制器，这个名字应该和所有的标志符一样是驼峰式的。也就是说用 ngModel 代替 ng-model。
Injector()	获取当前元素或者父元素的注入器。这个允许你在这些模块中查找模块的依赖
Scope()	返回当前前元素或者最近父元素的作用域
inheritedData()	和 jquery 的 data()函数一样，inheritedData（）以封闭的方式设置以及获取元素上的数据。除了从当期元素获取数据，它会遍历 DOM 查找

举个例子，让我们不用 ng-show 和 ng-click 重新实现下之前的 expander 示例。代码如下：

```
angular.module('expanderModule', [])
.directive('expander', function(){
  return {
    restrict: 'EA', replace: true,
    transclude: true,
    scope: { title:'=expanderTitle' }, template:
    '<div>' +
        '<div class="title">{{title}}</div>' +
'<div class="body closed" ng-transclude></div>' +
'</div>',
link: function(scope, element, attrs) {
  var titleElement = angular.element(element.children().eq(0));
  var bodyElement = angular.element(element.children().eq(1));
titleElement.bind('click', toggle);

function toggle() {
  bodyElement.toggleClass('closed');

}
}
}
});
```

我们从模板中移除了 ng-click 和 ng-show 标识符。然而，当用户点击 expander 标题时，仍然执行预期的操作，我们从 title 元素上创建了一个 jqLite 元素，然后把 toggle 函数绑定到 click 事件上作为它的回调。在 toggle 函数，我们在 expander body 元素上调用 toggleClass()

来添加或移除一觉 closed 的类，我们会设置这个元素 class 设置成 display：noe，就想这样：

```
.closed {
display:  none;
}
```

控制器

当你有嵌套的标识符需要互相通信时，解决方法就是使用控制器。<menu>可以需要知道<menu-item>元素里面的，以便它能够在合适的时候展示和隐藏。同样地，<tab-set>需要知道<tab>元素，<grid-view>需要知道<grid-element>元素。

正如之前看到的，创建一个 API 用于标识符之间通信，你可以用控制器的属性语法声明一个控制器作为标识符的一部分：

```
controller: function controllerConstructor($scope, $element, $attrs, $transclude)
```

控制器函数是靠依赖注入的，因此这里列出的参数，虽然是有用的，但都是可选的，他们可以以任何顺序列出来。它们也只是服务变量的一个子集。

其他标识符可以使用 require 属性语法传递控制器传递给它们。Require 的完整形式：

```
require:  '^?directiveName'
```

表 6-7 有 Require 字符串的说明。

表 6-7 需要控制器的配置项

配置项	说明
directiveName	驼峰式名称，指定了控制器来自哪个标识符。因此，如果<my-menu-item>标识符需要它父类<my-menu>的一个控制器，我们会写 myMenu
^	默认情况下，angular 会从在相同元素上的标识符获取控制器。添加可选的^，说明遍历 DOM 查找标识符。对于<my-menu>示例，我们需要添加这个符号，最终的字符串是\^myMenu
?	如果需要的控制器没有找到，Anguar 会抛出一个异常，告诉你是什么问题。添加?符号说明这个控制器是可选的，如果没有找到就不会抛出异常。虽然这个听上去不可能，但是如果希望<my-menu-item>不使用<my-menu>控制器，我们会为这个最终需要的字符串?\^myMenu 添加这个

举个例子，让我们重写我们的 expander 标识符用于一个叫‘手风琴‘的组件，它能够确保当你打开一个 expander 时，其他的都会自动的关闭，看起来就像这样：

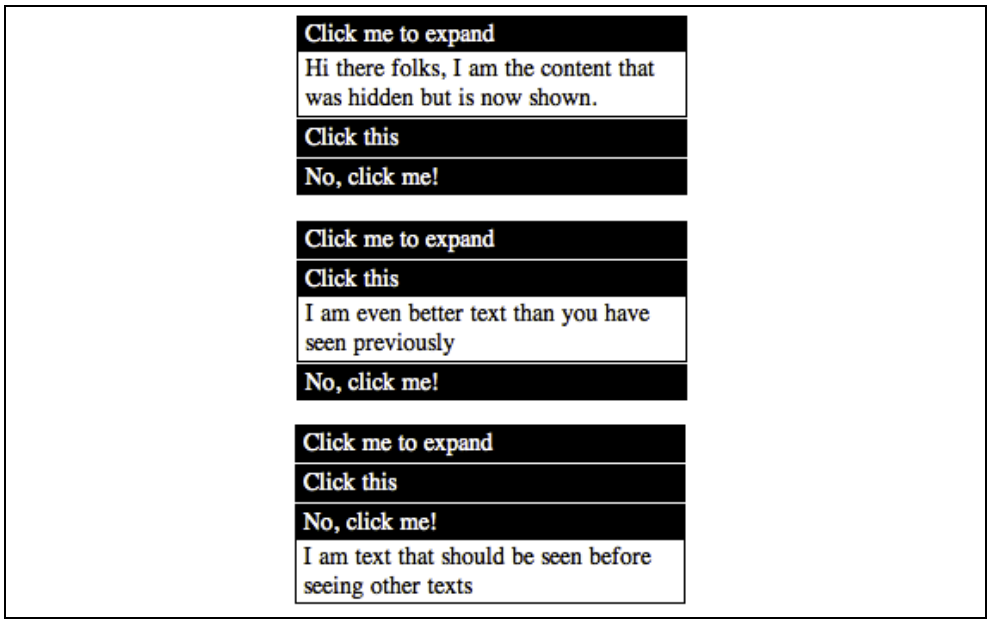


图 6-4 多状态的手风琴组件

首先，让我们写 accordion 标识符，用于协调。这里我们会添加我们的控制器构造函数来处理协调问题：

```
appModule.directive('accordion', function() {
return {
  restrict: 'EA', replace: true,
  transclude: true,
  template: '<div ng-transclude></div>', controller:
  function() {
var expanders = [];
this.gotOpened = function(selectedExpander) {
  angular.forEach(expanders, function(expander) {
    if (selectedExpander != expander) {
      expander.showMe = false;

    }
  });
}
this.addExpander = function(expander) {
  expanders.push(expander);

}
}
}
});
```

这里我们为expander定义了一个addExpander()函数，用于注册它们。也创建gotOpened()函数用于accordion的控制器可以关闭其他expander。

在expander标识符，我们会扩展它，需要来自父元素的accordion的控制器，在合适的时候调用addExpander()以及getOpened()。

```
appModule.directive('expander', function(){
return {
  restrict: 'EA', replace: true,
  transclude: true,
  require: '^?accordion',
  scope: { title:'=expanderTitle' }, template:
  '<div>' +
      '<div class="title" ng-click="toggle()">{{title}}</div>' +
      '<div class="body" ng-show="showMe" ng-transclude></div>' +
      '</div>',
    link: function(scope, element, attrs, accordionController) { scope.showMe
    = false; accordionController.addExpander(scope);
  scope.toggle = function toggle() { scope.showMe
  = !scope.showMe; accordionController.gotOpened(scope);

}
}
}
});
```

注意：accordin标识符内的控制器创建了API，通过它expander可以互相通信。

然后我们编写使用的模板，它会产生图6-4的最终结果：

```
<body ng-controller='SomeController' >
<accordion>
<expander class='expander'
ng-repeat='expander in expanders'
   expander-title='expander.title'>
    {{expander.text}}
  </expander>
</accordion>
</body>
```

当然得使用对应的控制器：

```
function SomeController($scope) {
$scope.expanders = [
  {title: 'Click me to expand',
```

```
    text: 'Hi there folks, I am the content that was hidden but is now shown.'},
  {title: 'Click this',
    text: 'I am even better text than you have seen previously'},
  {title: 'No, click me!',
    text: 'I am text that should be seen before seeing other texts'}

];
}
```

小结

正如我们看到的，标识符可以让我们扩展 HTML 语法以及让许多应用按我所说的声明式运作。标识符使重用变得轻而易举，从配置应用，就像 ng-model,ng-controller，ng-repeat 和 ng-view 来处理模板任务，到以前限制重复使用的组件，如 data-grids,bubble-charts,toop-tips,tabs 等等。

第七章：其他关注点

在这一章，我们会看看 AngularJS 中其他一些有用的特性，到目前为止，本章以及示例都未覆盖所有以及深入。

$location

到目前为止，你已经看到了 AngularJS 中一些有关$location 服务的示例。大部分只是一扫而过----在那儿设置，在这里访问。在本节，我们会深入 AngularJS $location 服务到底能做什么，何时该用，何时不能用。

$location 服务是浏览器中 window.location 的包装。那么为什么不直接使用 windo.location 呢？

不再使用全局状态

Window.location 是全局状态的典型例子（实际上，浏览器中的 window 和 document 对象都是典型的例子）。一旦应用中有了全局状态，它的测试，维护以及运作都是一个麻烦（即使现在不是，长期来看是这样的）。 $location 服务隐藏了这个问题（就是我们所谓的全局状态），允许你在单元测试期间通过注入模拟的参数来测试浏览器的位置信息。

API

Window.location 让你完全访问浏览器地址信息。那就是说，window.location 给你的是一段字符串，然而$location 给你更好的，以一种简洁的方式，jQuery 风格的 set 和 get 方式运作。

AngularJS 集成

如果你使用$location，那么无论你想要什么你应该使用它们。但是像 window.location，你也必须负责通知 AngularJS 这些变化，以及监听改变。

HTML5 集成

$location 服务也是很聪明的，当浏览器中 HTML5 API 可用时就会使用它们。如果不可用，就会退到默认的用法

那么，何时应该使用$location 服务？任何你想改变 URL 的时候（不是被$routes 覆盖的，你应该首先用于基于 URL 的视图） ，以及影响浏览器当前 URL 的变化。

让我们思考一个小示例，在真实的应用中你该如何是要你管$location 服务。考虑一种场景，我们有一个 datepicker，当选择一个日期时，应用导航到一个特定的 URL。让我们看看这个示例应该咋样：

```
// Assume that the datepicker calls $scope.dateSelected with the date
$scope.dateSelected = function(dateTxt) {
  $location.path('/filteredResults?startDate=' + dateTxt);
  // If this were being done in the callback for
  // an external library, like jQuery, then we would have to
```

```
  $scope.$apply();
};
```

使用还是不使用$apply

AngularJS 开发人员对何时该使用$scope.apply()何时不该使用总是有困惑。互联网上的建议和谣言都很多。这一节，我们会把它变得晶莹剔透。

首先，我们尝试使用$apply 的一种简单格式。

Scope.$apply 就像一个延迟的 worker。它被通知做很多事，负责确保更新绑定以及视图反应出这些变化。但不是一直做这个工作，只有当它感觉有足够多的任务去做时它才做它。在其他情况下，它只是点点头，标记下，稍后运行。只有当你得到它的注意或者明确告诉它应该运作，它实际上才运作。AngularJS 以规律化的间隔做这个，但是如果这些调用来自外部（比如 Jquery UI 事件），scope.$apply 只是做个标记，并不做什么。那就是为什么我们必须调用$scope.$apply 来通知它，‘hey！你需要立刻做这个，不需要等待’

这里有四个快速提示，关于何时（和如何）调用$apply。

- ***不要***一直调用它。当 AngularJS 正在愉快地执行（我们称之为$digest）时，再调用 $apply 可能会导致异常。因此，‘宁愿稳妥免致后悔’不是你想使用的途径
- 当需要控制 AngularJS 外部（DOM 事件，像 jqueryUI 控制的额外调用等等）调用 AngularJS 的函数时***应当调用它***。那时，你需要通知 AngularJS 更新自己（模型，视图，等等），$apply 就是做那些的。
- 只要可能，把执行的代码或函数传递给$apply，而不是执行函数，然后调用 $apply()。

例如，执行如下代码：

```
$scope.$apply(function() {
$scope.variable1 = 'some value';
executeSomeAction();
});
```

而不是这样：

```
$scope.variable1 = 'some value';
executeSomeAction();
$scope.$apply();
```

虽然这些都会产生相同的效果，但是在某种程度上是不同的。

首先当调用 excuteSomeAction 时会捕获任何错误，然而会忽略连这种错误。只有你用第一种方式时，才会得到 AngularJS 的错误通知。

- 考虑使用类似 safeApply:

```
$scope.safeApply = function(fn) {
var phase = this.$root.$$phase;
if(phase == '$apply' || phase == '$digest') {
  if(fn && (typeof(fn) === 'function')) {
    fn();

  }
} else {
```

```
    this.$apply(fn);

  }
};
```

你可以模拟这到最外层作用域或者根作用域，然后在任何地方使用$scope.$safeApply 函数。这已经在讨论中，在未来的某个版本中有望添加，这将是默认行为。

$loaction 对象上还有其他的一些可用的方法？表 7-1 为你列出了简单的摘要便于使用。

假如浏览器中的 URL 是 http://www.host.com/base/index.html#!/path?param1=value1#hashValue ，让我们看看 $location 服务会有怎样的行为。

表 7-1 $location 上的函数

Get 函数	值	Set 函数
absUrl()	http://www.host.com/base /index.html#!/path?param1=value1#hashValue	N/A
Hash()	Hash value	Hash('newHash')
Host	www.host.com	N/A
Path	/path	Path('/newpath')
Protocol	http	N/A
Search()	{'a','b'}	Search({'c':'def'})
url()	/path?param1=value1?hashValue	url('newPath?p2=v2')

表 7-1 中的 Set 函数列有一些举例的值指示了它所希望的对象类型。

注意，search()函数有几个操作模式：

- 简单地用 object<String,String>调用 search(searchObj)表示所有的参数以及参数值
- 调用 search(string)将直接在 URL 中设置 URL 参数作为 q=String
- 调用带字符串和 value 的 search(param,value)会在 URL 中设置一个特殊的搜索参数（或调用 null 移除参数）

使用任何一个 set 函数，并不是意味着 windo.location 会立刻发生变化。$location 服务和 Angular 具有同样的生命周期，因此，location 的所有变化会聚集，然后在生命周期的最后一起应用。因此和可以做任何改变，一个接一个的，无需害怕用户看到一个闪烁的和变化的 URL。

HTML5 模式和 Hashbang 模式

通过$locationProvider（它可以被注入，就像 AngularJS 其他参数一样）配置$location 服

务。这个提供者上最有趣的是两个属性：

Html5mode

一个布尔值，决定这$location 服务是否运行在 HTML5 模式下

hashPrefix

一个字符串（实际上是一个单个字符），作为 HashBang URL 的前缀（在 Hashbang 模式下或老版本浏览器在 HTML5 模式下）。默认它是空的，因此 Angular 的 hash 是’’，如果设置 hashPrefix 为!，那么 Angular 就会使用称为 Hashbang URL（!跟在后面）

批注 [spy26]: http://zh.wikipedia.org/wiki/Shebang
"Shebang"或者说"Hashbang"的名字有时也被当做 Ajax 应用程序中的分段标识符，用于浏览器的状态保存；Google 网站站长中心提到，以叹号开头的分段标识符（即...url#!state...）会为 Google 的网页爬虫所索引。

你可能会问，这些模式究竟是什么？好吧，假设你有一个用 Angular 的超棒的站点 www.superawesomewebsite.com。

假如你有一个特殊的路由（有一些参数和一个#），例如/foo?bar=123#baz。

在正常的 Hashbang 模式下（hashprefix 设置成!）,或者在不支持 HTML5 老版本浏览器，你的 URL 可能像这样：

http://www.superawesomewebsite.com/#!/foo?bar=123#baz

虽然在 HTML5 模式下，URL 可能简洁的像这样

http://www.superawesomewebsite.com/foo?bar=123#baz

这两种情况，location.pathname 都是/foo，location.search 都是 bar=123，location.hash 是 baz 即便是这种情形，为什么不想使用 HTML5 模式呢？

Hashbang 可以跨浏览器运行，需要最少的配置。你仅需要设置 hashBang 的前缀（默认是!）并且你可以做的更好。

另一方面，HTML5 模式，通过 HTML5‘历史 API’和浏览器的 URL 进行交互。$location 服务足够的聪明，能够判断出是否支持 HTML5 模式，如果有必要的话回溯到 Hashbang 途径，因此你不需要担心额外的事。但是你必须注意一下内容：

服务端配置

因为 HTML5 链接在应用中看上去像其他的 URL，因此你需要小心应用中服务端路由所有的链接到主页面（最有可能的是 index.html）。例如，如果你的应用是 superswesomewebsite.com 登陆页，应用中有一个/amazing?who=me 的路由，然后浏览器展示了这个 URL 为：http://www.suuperawesomewebsite.com/amazing?who=me+

当通过应用浏览时，这个很正常，因为 HTML5 History API 处理了很多事情。但是如果你尝试直接浏览这个 URL，你的服务器就会认为你疯了，因为在服务端没有这样的已知资源。因此，你不得不确保所有的请求/amazing 会重定向到/index.html#!/amazing。

Angular 将会从那点向前开始介入，处理一些事情。它会检测路径的变化，重定向到已定义的正确 AngularJS 路由。

链接重写

你可以像下面这种方式简单地指定 URL：

```
<a href="/some?foo=bar">link</a>
```

根据你是否使用 HTML5 模式，AngularJS 会做些处理，会分别重定向到/some?foo=bar 或者 index.html#!/some?foo=bar。不需要额外的步骤让你处理。很棒，是不是？

当时下面类型的链接是不会重写的，浏览器会执行页面的重新加载：

a. 链接包含 target 元素：

```
<a href=”/some/link” target=”_self”> link</a>
```

b. 链接到不同域名的绝对路径

```
<a href=http://www.angularjs.org>link</a>
```

虽然前面示例中使用了已存在的基础 URL，但这是不同的，因为它是一个绝对地址。

c. 链接是以一个已定义的不同基准地址开始的

link

相对链接

一定要检查所有的相对链接，图片，脚本等等。你要么指定入口 HTML 文件头部中的基准地址，要么使用绝对地址（以‘/’开始），在任何地方，因为通过文档中初始的绝对 URL，将相对 URL 解析成绝对 URL，这是不同于应用的根目录的。

强烈鼓励运行 Angular 的应用从文档根节点启用 History API，因为它可以处理很多相对链接问题。

AngularJS 模块方法

AngulJS 模块负责定义你的应用是如何启动的。它也声明式的定义了应用的组成部分。让我们看看它是如何实现这个的。

主方法在哪里

如果你来自像 Java 甚至 Python 的编程语言，你可能想知道 AngularJS 的主方法在哪里？你知道，主方法引导一切，并且是第一个执行的东西？主方法在 JavaScript 中定义函数，实例化，把一切都串联气力啊，然后通知你的应用运行？

AngularJS 没有那一套。替代它的是模块的概念。模块允许我们声明式的指定应用依赖，以及如何连接和引导。使用这种方式的原因是多方面的。

1. 它是**声明的**，这就意味这它是以一种更容易编写和理解的方式去书写。就像阅读英文一样。
2. 它是**模块化**的。强制你思考如何定义你的组件、依赖以及使它们更加清晰。
3. 它允许**易于测试**。在你的单元测试中，你可以有选择性的选取模块，避免代码中的不可测试部分。以及在场景测试中，你可以加载额外的模块，它们能够让其他组件协作变得更加容易。

首先，让我们看看你如何使用一个已定义的模块，然后看下我们如何声明一个模块。

比方说我们有一个模块，实际上，叫‘MyAwesomeApp’的模块。在 HTML 页面中，我仅仅是在<html>标签中加入如下内容（或者从技术角度上说，任何一个标签）：

<html ng-app=”MyAwesomeApp”>

ng-app 标识符通知 AngualarJS 使用 MyAwesomeApp 模块来启动应用。

那么，这个模块是如何定义的？那好，我们推荐你已经将 service,directives 和 filter 分离开。那么你的主模块只是声明其他的模块作为一个依赖（就像第四章中做的 RequireJS 示例）。

这样可以更容易的管理你的模块，因为它们是很好的完整代码块。每个模块有一个且只有一个职责。这同样允许你的测试中只加载它们关系的模块，因此减少了初始时需要的模块数量。这样的测试会变得更小，更有重点。

加载和依赖

模块的加载发生在两个不同的阶段，函数反映出了它们。这些就是配置块和运行块（或阶段）：

配置块：

AngularJS 在这阶段挂钩和注册所有的提供者。因为这一阶段，只有提供者和常量能够注入到配置模块。无论是否已初始化的服务（service）都不能注入。

运行块：

运行模块常用于快速启动应用，并且在注入器完成创建后就开始执行。从此时开始，就会阻止进一步的系统配置发生改变，只有实例化的和常量可以注入到运行块。运行块是你找到 AngularJS 中主方法最近的地方。

简便的方法

使用模块能够做什么呢？我们可以实例化控制器，标识符，过滤器以及服务，但是模块类允许你做的更多，正如表 7-2 所示：

API 方法	描述
Config(configFn)	使用这个方法来注册一些当模块加载时需要做的工作
Constant(name,object)	这个发生在第一次，因此你能够声明所有应用范围内的常量，让他们在所有的配置中可用（这列表中的第一个方法），实例化方法（来这里的所有方法，像控制器，服务等等）
Controller(name,constructor)	我们已经看到了它的许多示例；它建立了一个基本可用的控制器
Directive(name,directiveFactory)	正如第六章讨论的，这个允许你在应用中创建标识符
Filter(name,filterFactory)	允许你创建自定义的 AngularJS 过滤器，正如在前面章节中讨论的
Run(initializationFn)	当你需要在注入器建立后执行一些工作，恰好在应用对用户可用前，你可以使用这个方法
Value(name,object)	允许跨应用注入值
Service(name,serviceFactory)	下一章节讲述
Factory(name,factoryFn)	下一章节讲述
Provider(name,providerFn)	下一章节讲述

你可能意识到了，从前面的表中我们少了三个常用 API（Factory，Provider，Service）调用的详细内容。原因是：很容易混淆三者间的用法，因此我们会深入到一个小示例来，更好的说明了何时（以及如何）使用每一个。

Factory

每当我们有一个类或对象，在它们初始化之前需要大量的逻辑和参数时，就可以调用 Factory API。Factory 是一个函数，用于创建特定的值（或对象）。让我们举个例子，greeter

函数需要和 salutation 参数一起初始化：

```
function Greeter(salutation) {
 this.greet = function(name) {
   return  salutation + ' ' + name;
  };
}
```

Greeter factory 应该是这样：

```
myApp.factory('greeter',  function(salut)  {
return  new  Greeter(salut);
});
```

它应该像这样调用：

```
var  myGreeter = greeter('Halo');
```

Service

那 Services 是什么样的呢？好吧，Factory 和 Service 间的区别就是 Factory 调用函数传递进去然后返回结果。Serivce 通过 new 构造函数传递进去，然后返回结果。

因此，前面的 greeter Factory 用 greeter Service 像如下替换：

```
myApp.service('greeter',Greeter);
```

每次调用 greeter，AngualarJS 就会调用 new Greeter()，然后返回结果

Provider

这是它们当中最复杂的（显然，最可配置的）。Provider 结合了 Factory 和 Service，以及在注入系统完全就位之前，抛出配置 Provider 函数的配置信息（那就是说，是在配置块中）。

让我们看看一个使用 Provider 可修改的 greeter Service：

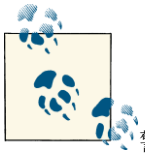
```
myApp.provider('greeter',  function()  {
  var salutation = 'Hello';
  this.setSalutation  = function(s)  {
    salutation = s;

  }

  function  Greeter(a)  {
    this.greet  = function()  {
      return  salutation + ' ' + a;

    }
  }
```

```
this.$get = function(a) {
  return new Greeter(a);
};
});
```

上面允许我们在运行时设置 salutation 参数（例如，基于用户的语言）。

```
var myApp = angular.module(myApp, []).config(function(greeterProvider) {
greeterProvider.setSalutation('Namaste');
});
```

无论何时只要有人调用了 greeter 对象的实例，AngularJS 内部就会调用$get 函数。


警告！

使用 **angular.module('MyApp',[...]);**和 **angular.module('MyApp');**是有轻微的，但是意义不同。

第一种方式，创建了一个 Angular 模块，在方括号中传递了模块依赖。第二种使用已经通过第一次调用定义好的已存在模块。

因此，你应该确保，在整个应用中，下面的代码只用了一次：

```
angular.module('MyApp',[...]);// Or MyModule, if you are modularizing you app
```

如果你不打算把它保存到变量中以及跨应用的引用，那么在剩余部分中应该使用 angular.module('MyApp')以确保你取到了正确的 AngularJS 模块的句柄。通过访问变量定义或者在已定义的模块中添加模块里的一切。

Scope 与$on,$emit,$broadcast 之间的通信

AngularJS 作用域是一种典型的分层和嵌套的结构。有一个主要的$rootScope(也就是，每个 Angular 应用或者 ng-app)，其他的作用域或者继承或者嵌套。常常，你会发现作用域没有共享变量或者没有从其他作用域原型继承。

在这种情况下，作用域间如何进行通信呢？一种方法是在应用的作用域中创建一个单例的 service，通过 service 处理所有内部作用域的通信。

AngularJS 中另外一种方式：通过作用域上的事件进行通信。这有些限制；例如，通常你不能广播一个时间到所有的监控作用域上。你必须有选择性的和父类或子类进行通信。

但是在我们讨论前，如何监听这些事件呢？这里有个示例，在任意星系上的作用域在等待和观察一个叫‘planetDestroyed’的事件。

```
scope.$on('planetDestroyed', function(event, galaxy, planet) {
// Custom event, so what planet was destroyed
scope.alertNearbyPlanets(galaxy, planet);
});
```

事件监听器的额外参数是从哪里来的？让我们看看单个行星是如何和它的父类星系进行通信的。

```
scope.$emit("planetDestroyed", scope.myGalaxy, scope.myPlanet);
```

$emit 的额外参数是以监听器函数的函数参数传递进来的。同样$emit 只能从当前作用域的向上通信，因此，如果他们的星球正在被摧毁，是不会通知星球上的穷人（如果他们有指向自己的作用域）。

同样的，如果银河系想和恒星系进行向下通信，那么它可能像这样进行通信：

```
Scope.$emit('selfDestructSystem', targetSystem);
```

然后，所有的恒星监听到这个事件，能够看到目标星系，然后决定他们是否应该自毁，使用如下命令：

```
scope.$on('selfDestructSystem', function(event, targetSystem) {
if (scope.mySystem === targetSystem) {
    scope.selfDestruct();       // Go Ka-boom!!

}
});
```

当然，由于事件一直向上（或向下）传播，所以，很有必要在某一个特定的等级或作用域时应该‘够了，你不能再通过了!’，或者阻止事件的默认行为。传递给监听器的事件对象有函数来处理上面所有的功能甚至更多，让我们快速浏览下从表 7-3 中得到事件对象的哪些信息。

表 7-3 事件对象属性和方法

时间属性	目的
Event.targetScope	发出或者传播原始事件的作用域
Event.currentScope	当前处理事件的作用域
Event.name	事件名称
Event.stopPropagation()	阻止事件进一步传播的函数（这个只有在$emit 的事件时才可用）
Event.preventDefault()	这个实际上并没有做什么事，只是设置了 defalutPrevented 为 true。由监听器的实现者判断 defaultPrevented 从而采取措施
Event.defalutPrevented	如果 preventDefalut 调用了就是 true

Cookie

不久，你在应用（提供了十分大和复杂的）中就遇到一种情形，你需要在客户端存储一类跨用户会话的状态。你可能记得（或者有过噩梦）通过使用 document.cookie 接口处理简单的文本 cookie。

幸好，许多年过去了，HTML5 API 在已出现的大多数现代浏览器中是可用的。此外，AngularJS 提供了一个非常棒的$cookie 和$cookieStore API 来处理 cookie。这两个服务能够和 HTML5 cookie 很好的协作，当他们可用时使用 HTML5 API，当他们不可用时使用 document.cookies。无论哪种方式，你都使用相同的 API。

让我们先看看$cookie 服务。$cookie 是一个简单的对象。它有键和值。添加一个键和值到对象中，就会添加相应信息到 cookie 中，同时从对象中移除它就会删掉特定的 cookie。就像这么简单。

但是绝大多数时候，你是不会直接在$cookie 级进行操作的。直接在 cookie 层次操作意味这需要维护字符串，自行解析，来回转换数据成对象。对于这些场景，我们有$cookieStore，它提供了一个可编程的方式来书写和移除 cookie。那么搜索控制器使用$cookieStore 来记忆最新 5 个搜索结果应该像这样：

```
function SearchController($scope, $cookieStore) {
$scope.search = function(text) {
  // Do the search here
  ...
  // Get the past results, or initialize an empty array if nothing found
  var pastSearches = $cookieStore.get('myapp.past.searches') || [];
  if (pastSearches.length > 5) {
    pastSearches = pastSearches.splice(0);

  }
  pastSearches.push(text);
  $cookieStore.put('myapp.past.searches', pastSearches);
};
}
```

国际化和本地化

你有可能会听到用户提出两个要求，何时支持应用中显示不同的语言。但是这两者之间有轻微的不同。想象下一个简单的应用，它是一个进入银行预算的门户。每次进入应用，它

会显示且只显示一个：

Greeting! The balance in your account as of 10/25/2012 is $xx,xxx.

很明显，上面的代码是针对于美国观众的。但是倘若我们想让这个应用在英国也可用怎么办（语言本身不需要做修改）？英国人使用不同的日期格式和货币符号，但是你不想让你的代码每当应用程序需要支持一种区域（这里是指 en_US 和 en_UK）时做出改变。从代码逻辑中抽象出日期/时间格式以及货币符号的过程叫做国际化（Internationalization）（或者叫 i18n----i 与 n 之间有 18 个字符）。

倘若让应用支持北印度或者俄罗斯，怎么办？此外，日期格式和货币符号（以及格式），设置 UI 中使用的字符都必须修改。在不同的区域中提供翻译和为抽象的二进制位本地化的字符串的过程称为本地化（Localization）（或者 L10n---用大写的 L 来区分 i 和 l）

在 AngularJS 中能做什么?

AngualarJS 为如下过滤器提供了 i18n 和 L10n：

- Currency
- Date/time
- Number

用 ngPluralize 标识符也提供了多元化的支持（英语，以及 i18n/L10n）。

通过$locale 服务可以操纵和管理所有的多元化支持，它管理着区域规则集。$locale 服务管理着区域标识，它通常由两部分组成：国家编码和语言编码。例如，en_US 和 en_UK，分别表示在 US 和 UK 使用英文。指定国家编码是可选的，只指定一个‘en’也是合法的本地编码。

如何让它们一直运作

让 L10n 和 i18N 在 AngualJS 中运行需要三步骤：

Index.html 修改

AngularJS 需要你为每个独立支持的区域有一个独立的 index.html。你的服务器也需要支持它应该提供那种 index.html，这依赖与用户的语言首选项（当用户改变语言环境时，也可能从客户端触发改变）

创建语言环境规则集

第二步为每个支持的语言创建一个 angular.js 文件，就像 angular_en-US.js，angular_zh-CN.js。这个涉及到在 angular.js 或者 angular.min.js 末尾串联每个特定语言的语言规则（上面的两个语言默认就是 angular-locale_en-US.js，angular-locale_zh-CN.js）。那么，angular_en-US.js 会首先包含 angular.js 的内容，紧跟着 angular-locale_en-US.js 的内容。

管理语言规则集

最后一步，确保本地化后的 index.html 引用到本地化的规则集，而不是原始的 angular.js

文件。因此 index_en-US.html 应该使用 angular-en_US.js 不是 angular.js。

你可能问 UI 字符串呢？AngularJS 目前还没有自己的完全成熟的翻译 API，因此你必须拿出自己的技术和脚本翻译 UI 字符串。这个过程可能是解析 HTML 字符串，然后传递给解析器，为每个语言输出一个 HTML，或者依据你的需求做些更多更复杂的。

常见陷阱

翻译长度

你设计你的 UI，以便在 div 中展示 June 24,1988，花了很大的力气满足它的大小。然后以西班牙语打开 UI。24 de junio de 1988 就不能适合原先同样的空间大小。

当国际化应用时，记住改变语言时，字符串的长度可能发生彻底的变化。应当设计相应的 CSS，通过这个测试多语言的情形（不要忘记，也存在从右到左的语言）

时区

AngularJS date/time 过滤器是从浏览器中获取时区设置的。因此已赖电脑上的时区，不同的人可能看到不同的信息。无论是 JS 还是 AngularJS，内在支持由开发人员用指定的时区显示时间。

清理 HTML 和模块

AngularJS 认真处理其安全性，视图尽可能的确保是大多数攻击最小化。攻击源之一就是围绕不安全的 HTMl 内容注入到 web 页面中，使用这个触发跨站或注入攻击。

考虑一个示例，在作用域上有一个叫 myUnsafeHTMLContent 的变量。OnMouseOver 修改元素的内容为“PWN3D”，如果使用下面的 HTML 内容:

```
$scope.myUnsafeHTMLContent = '<p style="color:blue">an html' +
    '<em onmouseover="this.textContent = 'PWN3D!'">click here</em>' +
'snippet</p>';
```

当在变量中有一些 HTML 内容，并且尝试绑定它时，AngularJS 的默认行为是 AngularJS 忽略里面的内容，直接打印它。因此 HTML 内容会被当做纯文本。

因此：

```
<div ng-bind='myUnsafeHTMLContent'></div>
```

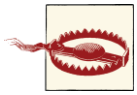
结果就是：

```
<p style="color:blue">an html
<em onmouseover="this.textContent='PWN3D!'">click here</em>
snippet</p>
```

在 Web 页面上就是渲染为文本。

但是倘若你想在 AngularJS 中渲染 myUnsafeHTMLContent 的内容做为 HTML 来呈现。在这种情况下，AngularJS 需要额外的标识符（以及一个服务，$sanitize，来启动）才允许你同时以一种安全和非安全的方式渲染 HTML。

让我们举个安全渲染的例子（正常情况都应该这样！），然后渲染 HTML，在 HTML 中要十分小心的摆脱一切可能的攻击源。你应该在这种情形下使用 ng-bind-html 标识符。



Ng-bind-html,ng-bind-html-safe 以及 linky 过滤器都是在 ngSanitize 模块下。你需要在脚本依赖中包含 angular-sanitize.js(或者.min.js)，然后在它运行前给 ngSanitize 添加模块依赖。

当我们在相同额 myUnsafeHTMLContent 上使用 ng-bind-html 标识符会发生什么呢？

```
<div ng-bind-html="myUnsafeHTMLContent"></div>
```

在这样的情形下输出应该是这样：

```
an html _click here_ snippet
```

需要着重注意的是 style 标签（带 color:blue），在标签的 onmouseover 处理上会被 AngularJS 删除。因为它们被视为是不安全的，因此被丢弃了。

最后，如果你决定你真的想让 myUnsafeHTMLContent 内容渲染出来，或者因为你真的信任内容的来源，或者因为其他原因，那么你可以使用 ng-bind-html-unsafe 标识符：

```
<div ng-bind-html-unsafe="myUnsafeHTMLContent"></div>
```

在这种场景下的输出就像这样：

```
an html _click here_ snippet
```

文本内容的颜色是蓝色（每当样式绑定到 p 标签时），‘click here’确实有一个 onmouseover 注册在上面。因此，你的鼠标移到‘click here’文本上时，输出内容就会变成：

```
an html PWN3D! snippet
```

正如你所看到的，在现实中这是很不安全的，因此当你决定使用 ng-bind-html-unsafe 标识符时必须绝对确认这是你所想要的。某些人可能很容易的读取到用户信息，然后把它发送到服务器。

Linky

Linky 过滤器同样在也 ngSanitize 模块中，基本上允许你添加可渲染的 HTML 内容以及转换成可以在 HTML 中展示的超链接标签。使用上很简单，因此让我们看一个示例：

```
$scope.contents = 'Text with links: http://angularjs.org/ & mailto:us@there.org';
```

如果你使用如下绑定：

```
<div ng-bind-html="contents"></div>
```

这会导致以 HTML 内容的形式打印出来：

```
Text with links: http://angularjs.org/ & mailto:us@there.org
```

现在让我们用 linky 过滤器再看看会发生什么：

```
<div ng-bind-html="contents | linky"></div>
```

Linky 过滤器遍历文本内容，给所有的 URL 和发现的 mailto 链接添加<a>标签，因此提供用户可以交互的 HTML 内容：

```
Text with links: http://angularjs.org/ & us@there.org
```

第八章：捷径和技巧

到目前，我们已经讲述了 Angular 很多不同的部分，包括标识符，服务，控制器，资源等等。但是我们有时只是了解这些还是不够的。有时，我们不需要关系如何运作的，只是想要知道如何用 AngularJS 做一个东西。

在这一章节，我们会为大多数 web 应用中常见的问题提供完整的代码示例（只给出了少量的信息以及解释）。他们是无序的，因此可以跳跃到任何你感兴趣的部分，或者一次遍历他们。随你便，你是老大！

这章包括的示例：

1. 封装 jQuery Datepicker
2. Teams List 应用：过滤器和控制器交互
3. AngularJS 中文件上传
4. 使用 socket.IO
5. 一个简单的分页服务
6. 和服务器协作

封装 jQuery Datepicker

这个示例在我们 GitHub 页面的 chapter8/datepicker 中。

在深入代码之前，我们必须决定我们的组件如何展示和运作。换言之，我们希望在 HTML 中按照如下方式定义我们 datepicker：

```
<input    datepicker ng-model="currentDate"    select="updateMyText(date)"></input>
```

那就是说，通过增加 datepicker 属性修改 input 字段，给它添加一些更多的功能（比如和模型进行数据绑定，当选择一个日期时可以得到通知）。那么，我们如何做到这一点呢？

我们会从用已存在的功能，jQuery UI 的 datepicker，而不是从头构建一个 datepicker。我们只是把它挂到 AngularJS 上，并使用它提供的钩子：

```
angular.module('myApp.directives', [])
.directive('datepicker', function() {
return {
// Enforce the angularJS default of restricting the directive to
// attributes only
restrict: 'A',
// Always use along with an ng-model
require: '?ngModel',
```

```
scope: {
// This method needs to be defined and
// passed in to the directive from the view controller
select: '&' // Bind the select function we refer to the
// right scope
},
link: function(scope, element, attrs, ngModel) {
if (!ngModel) return;
var optionsObj = {};
optionsObj.dateFormat = 'mm/dd/yy';
var updateModel = function(dateTxt) {
scope.$apply(function () {
// Call the internal AngularJS helper to
// update the two-way binding
ngModel.$setViewValue(dateTxt);
});
};
optionsObj.onSelect = function(dateTxt, picker) {
updateModel(dateTxt);
if (scope.select) {
scope.$apply(function() {
scope.select({date: dateTxt});
});
}
};
ngModel.$render = function() {
// Use the AngularJS internal 'binding-specific' variable
element.datepicker('setDate', ngModel.$viewValue || '');
};
element.datepicker(optionsObj);
}
};
});
```

大部分代码是十分简单的，但是让我们过下一些要点：

ng-model

我们将 ng-model 属性传递给标识符的 link 函数。Ng-model（对于标识符来说这是必须额，因为标识符定义中有 require 属性）允许我们定义让属性和对象如何链接到标识符上下文中的 ng-model 行为。这里你需要关注亮点：

ngModel.$setViewValue(dateTxt)

当 AngularJS 外部（这里，是 jQuery UI datepicker 的 onSelect）需要时就会调用。这是通知 AngularJS 它必须更新模型。这通常叫做一个 DOM 事件何时发生。

ngModel.$render

这是 ng-model 的第二部分。这个通知 AngularJS，当模型发生变化时如何更新视图。在我们的示例中，仅仅是把改变后的 datepicker 值传递给 jQuery UI。

绑定下拉框

不是使用属性值和把它计算为 scope 的字符串（在这种情形下，嵌套的函数和对象是不可访问的），我们是使用函数绑定（&作用域绑定）。这会在 select 作用域上创建一个函数，它需要一个参数----一个对象。对象中的每个 key 必须和 HTML 中使用标识符的参数字段必须一致。Key 对应的值作为参数传递给函数。额外的好处就是，把控制器的实现，从必须知道 DOM 或者标识符中分离开来。回调函数仅仅是执行给定参数的行为，不需要知道绑定关系或者更新。

调用下拉框

注意，在 onSelect 函数上，我们传递了一个 optionObj 给 datepicker。jQuery UI 负责调用 onSelect 函数，它通常发生贼 AngularJS 的执行上下文之外。当然，当调用像 onSelect 这类的函数时，AngularJS 是不知道的。这就需要我们来让 AngularJS 知道，它需要采取什么样的措施。我们如何做呢？可以使用 scope.$apply()。

我们仅仅是简单地做了$setViewValue，在 scope.$apply 外调用了 scope.select，然后再调用 scope.apply()。但是在这两步中发生的异常都会丢失。如果发生在 scope.$apply()函数中，那么 AngularJS 会捕获到他们。

示例的剩余部分

为了完成这个示例，让我们看看控制器部分代码，然后可以在 HTML 中运行：

```
var app = angular.module('myApp', ['myApp.directives']);
app.controller('MainCtrl', function($scope) {
  $scope.myText = 'Not Selected';
  $scope.currentDate = '';
  $scope.updateMyText = function(date) {
    $scope.myText = 'Selected';
  };
});
```

相当简单的东西，我们声明了一个控制器，设置了一些作用域变量，然后创建了一个作用域方法（updateMyText），稍后我们会使用它绑定到 datepicker 的 on-select 事件上。HTML 内容是：

```
<!DOCTYPE html>
<html ng-app="myApp">
```

```
<head lang="en">
<meta charset="utf-8">
<title>AngularJS Datepicker</title>
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
</script>
<script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js">
</script>
<script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/
angular.min.js">
</script>
<link rel="stylesheet"
href="http://code.jquery.com/ui/1.9.2/themes/base/jquery-ui.css">
<script src="datepicker.js"></script>
<script src="app.js"></script>
</head>
<body ng-controller="MainCtrl">
<input id="dateField"
datepicker
ng-model="$parent.currentDate"
select="updateMyText(date)"> <br/>
{{myText}} - {{currentDate}}
</body>
</html>
```

注意，这里指定了 select 属性。在作用于上没有叫‘date’的值。但是因为我们在标识符中建立杭虎绑定，AngularJS 现在就回知道，函数会有一个叫 date 的参数。这就是当 datepicker 的 onSelect 调用时我们指定了一个对象。

对于 ng-model，我们指定了$parent.currentDate 而不是 currentDate。为什么呢？因为我们的标识符创建了一个独立的作用域以便我们使用 select 函数绑定。即使我们设置了它，currentDate 也不会被 ng-model 关联。因此我们必须明确告诉 AngularJS，currentDate 它需要的引用不在独立的作用域中，而是在它父作用域上。

那么，当你在浏览器上加载它时，就会看到一个文本框，点击时展示 jQuery UI 的 datepicker。选择时，它会把界面上文本‘No Selected’更新为‘Selected’。输入框中的日期也会被更新。

Teams List 应用：过滤器和控制器交互

在这个示例中，我们会同时处理多个东西，但是有两个卖点：

1. 如何使用过滤器，尤其是以一种简洁的方式，和重复器一起使用？
2. 没有共享继承关系的控制器之间如何交互？

应用本身很简单。有一份数据，来自多个球队的一份团队列表，像篮球队，橄榄球队（国家橄榄球联盟，不是英式足球），曲棍球队。无论队伍有什么特长，每队都有名称，城市，运动项目。

我们想做的就是展示这个列表，同时在左侧展示过滤器，只要你修改了可以马上更新列表。我们打算有两个控制器：一个用来存储数据，另外一个用于和过滤器一起工作。我们打算用 service 来交互 ListCtrl 和 FilterCtrl 之间的过滤变化。

首先让我们看下 service，它将驱动这个应用：

```
angular.module('myApp.services', []).
factory('filterService', function() {
return {
activeFilters: {},
searchText: ''
};
});
```

额，就这样？是的，这里我们所做的就是利用这样一个原理，AngularJS services 是单例（这个以小‘s’开头的单例，在作用域内是单例，但是在全局上是不可见和不可访问的）。当我们声明 filterService 时，我们保证在整个 myApp 应用上只有一个 filterSerivce 实例。

然后，我们最终使用 filterService 作为 filterCtrl 和 listCtrl 之间的通信渠道，因为两者可以绑定到它上面，随着它更新访问资源。这些控制器实际上是写死了的，因为它们没有做任何事除了简单的工作：

```
var app = angular.module('myApp', ['myApp.services']);
app.controller('ListCtrl', function($scope, filterService) {
  $scope.filterService = filterService;
  $scope.teamsList = [{
      id: 1, name: 'Dallas Mavericks', sport: 'Basketball', city:
      'Dallas', featured: true
    }, {
      id: 2, name: 'Dallas Cowboys', sport: 'Football', city:
      'Dallas', featured: false
    }, {
      id: 3, name: 'New York Knicks', sport: 'Basketball', city: 'New
      York', featured: false
    }, {
      id: 4, name: 'Brooklyn Nets', sport: 'Basketball', city: 'New
```

```
      York', featured: false
    }, {
      id: 5, name: 'New York Jets', sport: 'Football', city: 'New
      York', featured: false
    }, {
      id: 6, name: 'New York Giants', sport: 'Football', city: 'New
      York', featured: true
    }, {
      id: 7, name: 'Los Angeles Lakers', sport: 'Basketball', city: 'Los
      Angeles', featured: true
    }, {
      id: 8, name: 'Los Angeles Clippers', sport: 'Basketball', city: 'Los
      Angeles', featured: false
    }, {
      id: 9, name: 'Dallas Stars', sport: 'Hockey', city:
      'Dallas', featured: false
    }, {
      id: 10, name: 'Boston Bruins', sport: 'Hockey', city:
      'Boston', featured: true

    }
  ];
});
app.controller('FilterCtrl', function($scope, filterService) {
$scope.filterService = filterService;
});
```

你可能想知道，哪里复杂了？AngularJS 确实使这个变得容易。我们所要做的就是在模板中把这些组合在一起：

```
<!DOCTYPE html>
<html ng-app="myApp">

<head lang="en">
  <meta charset="utf-8">
  <title>Teams List App</title>
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
  </script>
  <script
    src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
  </script>
  <link rel="stylesheet"
      href="http://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/2.1.1/
       css/bootstrap.min.css">
```

```
  <script
      src="http://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/2.1.1/
      bootstrap.min.js">
  </script>
  <script   src="services.js"></script>
  <script   src="app.js"></script>
</head>
<body>
<div class="row-fluid">
  <div class="span3"   ng-controller="FilterCtrl">
    <form class="form-horizontal">
<div class="controls-row">
  <label   for="searchTextBox"    class="control-label">Search:</label>
  <div class="controls">
          <input type="text" id="searchTextBox"
          ng-model="filterService.searchText">

  </div>
</div>

<div class="controls-row">
  <label   for="sportComboBox" class="control-label">Sport:</label>
  <div class="controls">
    <select   id="sportComboBox"
          ng-model="filterService.activeFilters.sport">
      <option ng-repeat="sport in ['Basketball',    'Hockey', 'Football']">
        {{sport}}
      </option>
</select>
  </div>
</div>
      <div class="controls-row">
        <label   for="cityComboBox" class="control-label">City:</label>
        <div class="controls">
          <select   id="cityComboBox"
                ng-model="filterService.activeFilters.city">
            <option ng-repeat="city in ['Dallas', 'Los   Angeles',
                                        'Boston', 'New York']">

              {{city}}
            </option>
          </select>
        </div>
      </div>
```

```
      <div class="controls-row">
        <label   class="control-label">Featured:</label>
        <div class="controls">
          <input type="checkbox"
                 ng-model="filterService.activeFilters.featured"
                 ng-false-value="" />

        </div>
      </div>
    </form>
  </div>
  <div class="offset1 span8" ng-controller="ListCtrl">
    <table>
      <thead>
      <tr>
        <th>Name</th>
        <th>Sport</th>
        <th>City</th>
        <th>Featured</th>
      </tr>
      </thead>
      <tbody id="teamListTable">
      <tr   ng-repeat="team in teamsList   | filter:filterService.activeFilters |
                    filter:filterService.searchText">
        <td>{{team.name}}</td>
        <td>{{team.sport}}</td>
        <td>{{team.city}}</td>
        <td>{{team.featured}}</td>
      </tr>
      </tbody>
    </table>
  </div>
</div>
</body>
</html>
```

在整个 HTML 模板中有四点值得关注。除此以外的代码，到现在为止，你已经看了数十遍了（甚至这些代码以这样或那样的形式出现过）。让我们依次看下这四点。

搜索框

搜索框仅仅是使用 ng-model 绑定到了 filterService.searchText 字段。属性本身没有什么

值得注意的，但是后面在过滤器上使用它的方式使得这一步很有必要。

组合框

这里有两个下拉框，这里我们只说明第一个。这两个的运行方式都是一样的。他们都是绑定到 filterService.activeFilters.sports 或者 city 属性上（由下拉框决定），它在 filterService 的过滤器对象上建立了 sports（或 city）属性。

复选框

复选框绑定到 filterSerivce.activeFilters.featured 属性上。需要注意的是勾选 featured 时，我们希望只展示那些 featured=true 的团队。当没有勾选时，我们希望展示 featured=true 和 featured=false 的团队。对于这个需求，我们使用 ng-false-value=''标识符，那就是说没有勾选复选框时，featured 过滤器应该被清空。

迭代器

让我们看下 ng-repeat 语句：

```
"team in teamsList    | filter:filterService.activeFilters | filter:filterService.searchText"
```

第一部分总是一样的。有两个过滤器使它变得与众不同。第一个是告诉 AngularJS 使用 filterSerivce.activeFilters 过滤列表。这基本上取过滤器对象的每个属性，然后确保迭代器中的每一项匹配过滤器中响应属性。如果 activeFilters[city]=Dallas，那么只有重复器中每一项 city=Dallas 的才会被选中。如果有多个过滤器，那么需要匹配所有过滤器。

第二个过滤器是一个文本匹配过滤器。它主要选择那些在列表值中出现的 filterService.searchText 值。所以它会匹配 city，name,sports,featured 等属性。

AngularJS 中文件上传

我们常见的另外一个用例就是在 AngularJS 应用中支持文件上传。虽然通过在 HTML 中使用已有的 input type='file'来支持是可行的，但是出于这个示例的目的，我们将扩展已存在的文件上传功能。BlueImp's File Upload 是很好的一个，它使用了 jQuery 和 jQuery UI(或者 Bootstrap)。它们的 API 死简单，同样会使我们的标识符超简单。

那么，让我们从标识符申明部分开始：

```
angular.module('myApp.directives', [])
  .directive('fileupload', function() {
    return { restrict: 'A',
    scope: {
```

```
      done: '&', progress: '&'
    },
    link: function(scope, element, attrs) {
      var optionsObj = {
        dataType: 'json'
      };

      if (scope.done) {
        optionsObj.done = function() {
          scope.$apply(function() {
            scope.done({e: e, data: data});
          });
        };

      }

      if (scope.progress) {
        optionsObj.progress = function(e, data) {
          scope.$apply(function() {
            scope.progress({e: e, data: data});
          });

        }
      }

      // the above could easily be done in a loop, to cover
      // all API's that Fileupload provides

                                                                    }
                                                                  };

element.fileupload(optionsObj);
});
```

我们的 input 标签只是多了如下内容：

Fileupload

这个让 input 标签成为文件上传元素

Data-url

通过 FileUpload 创建来决定将文件上传到哪里去。在我们的示例中，我们假设有个服务端 API /server/uploadFile 来处理它发送过来的数据。

multiple

multiple 属性告诉标识符（fileupload 组件）允许一次选择多个文件。我们从插件中免费获得，不需要写一行额外的代码。再者，这是一个内置插件的好处。

done

当插件完成上传所选的文件时，就会调用 AngularJS 函数。如果我们想要进度条，我们可以增加类似的。这个同样需要为我们的标识符调用指定两个参数。

那么控制器会是什么样子的呢？正如你期望的：

```
var app = angular.module('myApp', ['myApp.directives']);
app.controller('MainCtrl', function($scope) {
  $scope.uploadFinished = function(e, data) {
    console.log('We just finished uploading this baby...');
  };
});
```

使用它们，我们有一个简单，可运行，可重用的文件上传标识符。

使用 Socket.IO

对于现代的 WEB 一个常见的需求是实时 WEB 应用，一旦服务器上的数据更新后，它能够就会更新。之前使用的技术比如轮询已经发现有缺陷，有些时候我们仅仅是希望给客户端打开一个 socket 然后通信。

Socket.IO 是一个优秀的库，它允许你实现这种效果，使用很简单的，基于事件的 API，允许你开发实时 WEB 应用。我们准备开发一个实时，匿名的广播系统（就像 Twiter，没有用户名）允许用户广播一个消息给所有 Socket.IO 的用户，他们能够看到所有的信息。因为不存储任何东西，所以一旦指定的用户是可用的，那么所有的消息都是可见的，但是这足够证明 Socket.IO 可以很好的集成到 AngularJS。

我们马上会封装 Socket.IO 成一个很好用的 AngualrJS 服务，通过这个，我能保证：

- 在 AngularJS 生命周期内，通知和处理 Socket.IO 事件
- 便于集成测试

```
var app = angular.module('myApp', []);
// We define the socket service as a factory so that it
// is instantiated only once, and thus acts as a singleton
// for the scope of the application.
app.factory('socket', function ($rootScope) {
    var socket = io.connect('http://localhost:8080');
    return {
        on: function (eventName, callback) {
            socket.on(eventName, function () {
                var args = arguments;
                $rootScope.$apply(function () {
                    callback.apply(socket, args);
                });
            });
```

```
        },
        emit: function (eventName, data, callback) {
            socket.emit(eventName, data, function () {
                var args = arguments;
                $rootScope.$apply(function () {
                    if (callback) {
                        callback.apply(socket, args);
                    }
                });
            });
        }
    }
});
```

我们仅仅是封装了我们关心的两个功能，他们是 Socket.IO API 中的 on 事件和 emit 事件方法。还有很多方法，可以用类似的方式封装。

我们建一个简单的 index.html，它有一个带发送按钮的输入框以及一个消息列表。在这个示例中，我们没有记录是谁发送的消息以及何时发送的。

```
<!DOCTYPE html>
<html ng-app="myApp">

<head lang="en">
  <meta charset="utf-8">
  <title>Anonymous Broadcaster</title>
  <script src="/socket.io/socket.io.js">
  </script>
  <script
    src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
  </script>
  <script src="app.js"></script>
</head>
<body ng-controller="MainCtrl">
    <input type="text" ng-model="message">
    <button ng-click="broadcast()">Broadcast</button>

    <ul>
      <li ng-repeat="msg in messages">{{msg}}</li>
    </ul>
</body>
</html>
```

让我们看看 MainCtrl（在 app.js 中），这里是我们整合它的地方：

```
function MainCtrl($scope, socket) {
$scope.message = '';
$scope.messages = [];

// When we see a new msg event from the server
socket.on('new:msg', function (message) {
  $scope.messages.push(message);
});
// Tell the server there is a new message
$scope.broadcast = function() {
  socket.emit('broadcast:msg', {message: $scope.message});
  $scope.messages.push($scope.message);
  $scope.message = '';
};
}
```

控制器本身十分简单。它监听这 socket 链接上的时间，只要用户按下广播按钮，服务器就会知道有一个新消息。同时会添加到消息列表，立刻实现给用户。

然后，看下最后的一部分代码，server.js。这是一个 NodeJS 服务器，它知道如何为应用代码提供服务，同时创建了一个 Socket.IO 服务器。

```
var app = require('express')()
, server = require('http').createServer(app) , io = require('socket.io').listen(server);
server.listen(8080);

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});

app.get('/app.js', function(req, res) {
  res.sendfile(__dirname + '/app.js');
});

io.sockets.on('connection', function (socket) {
  socket.emit('new:msg', 'Welcome to AnonBoard');
socket.on('broadcast:msg', function(data) {
  // Tell all the other clients (except self) about the new message
  socket.broadcast.emit('new:msg', data.message);
});
});
```

后续，你能很容易地扩展这个，处理更多的消息以及更复杂的结构，但是这个示例展示了基础功能，在这个功能上实现了客户端和服务器之间的 socket 链接。

整个应用很简单。它没有做任何校验（消息是否为空），但是它有 AngularJS 默认提供的

HTML 清理。不能处理复杂的消息，但是它确实提供了一种将 Socket.IO 集成到 AngularJS 的完整可用的端到端的实现方式，你可以现在构建你的代码。

一个简单的分页服务

对于绝大数 web 应用的一个非常常见的场景就是展示一个列表。时常，我们有更多的数据而不能显示在单个页面上。在这种场景下，我们希望以分页的形式展示数据，同时可以有上下页的功能。由于在整个应用中这是一个通用的需求，因此把它这个功能抽象成一个通用，可重用的分页服务是十分有意义的。

我们的分页服务（一个十分简单的实现）允许用户通知服务如何查询数据，给出一个偏移量和限制，以及每页大小。它会内部处理所有的逻辑，计算出需要查询那一项，下一页是什么，是否有下一页等等。

可以扩展这个服务来在服务内缓存各项，但是这就留给读者做一个练习。所有的示例都是存储在缓存中的 currentPageItems 字段中，如果可用就从这查询它，否则的话调用查询功能函数。

看下服务实现：

```
angular.module('services', []).factory('Paginator', function() {
  // Despite being a factory, the user of the service gets a new
  // Paginator every time he calls the service. This is because
  // we return a function that provides an object when executed

    return function(fetchFunction, pageSize) {
        var paginator = { hasNextVar:
        false, next: function() {
            if (this.hasNextVar) { this.currentOffset +=
            pageSize; this._load();

          }
        },
        _load: function() {
          var self = this;
            fetchFunction(this.currentOffset, pageSize + 1, function(items)
            { self.currentPageItems = items.slice(0, pageSize); self.hasNextVar =
            items.length === pageSize + 1;
          });
        },
        hasNext: function() {
          return this.hasNextVar;
        },
        previous: function() {
            if(this.hasPrevious()) { this.currentOffset -=
            pageSize; this._load();
```

```
        }
      },
      hasPrevious: function() {
        return this.currentOffset !== 0;
      },
      currentPageItems: [], currentOffset:
      0
    };
  // Load the first page
   paginator._load();
  return paginator;
};
});
```

当调用分页服务时，需要两个参数：一个查询函数，和每页大小。查询函数具有如下签名：

fetchFunction(offset, limit, callback)

每当需要查询和展示一个特定页时，就会带上正确的 offset,limit 通过分页调用它。对于内部函数，既可以从很大的数组中切分数据，或者连到服务器，发起调用查询数据。当数据可用时，查询函数就需要带上数据调用 callback 函数。

让我们看下这个描述，为了清晰当有很大的一个数组，有很多项返回给我们时我们该如何使用它。注意这是一个单元测试。由于这种实现方式，我们可以测试任何控制器或者 XHR 请求的服务独立性。

```
describe('Paginator Service', function() {
beforeEach(module('services'));
var paginator;

var items = [1, 2, 3, 4, 5, 6];
var fetchFn = function(offset, limit, callback) {
  callback(items.slice(offset, offset + limit));
};
beforeEach(inject(function(Paginator) {
  paginator = Paginator(fetchFn, 3);
}));
  it('should show items on the first page', function()
  { expect(paginator.currentPageItems).toEqual([1, 2, 3]);
  expect(paginator.hasNext()).toBeTruthy();
  expect(paginator.hasPrevious()).toBeFalsy();
});

  it('should go to the next page', function() { paginator.next();
```

```
  expect(paginator.currentPageItems).toEqual([4, 5, 6]);
  expect(paginator.hasNext()).toBeFalsy();
  expect(paginator.hasPrevious()).toBeTruthy();
});

  it('should go to previous page', function() { paginator.next();
  expect(paginator.currentPageItems).toEqual([4, 5, 6]);
  paginator.previous(); expect(paginator.currentPageItems).toEqual([1, 2,
  3]);
});

  it('should not go next from last page', function() { paginator.next();
  expect(paginator.currentPageItems).toEqual([4, 5, 6]); paginator.next();
  expect(paginator.currentPageItems).toEqual([4, 5, 6]);
});

it('should not go back from first page', function() {
paginator.previous();
  expect(paginator.currentPageItems).toEqual([1, 2, 3]);
});
});
```

分页程序暴露 currentPageItems，它能约束 repeater 上的模板（除非你想展示他们）。HasNext()和 hasPrevious()用于判断何时显示下一页和上一页链接，点击它们时，会分别调用 next()或 previous()。

你如何使用这个，从服务器上查询每页数据？这里有一个可用的控制器，用来从服务器查询搜索结果，每次一页：

```
var app = angular.module('myApp', ['myApp.services']);
app.controller('MainCtrl', ['$scope', '$http', 'Paginator',
  function($scope, $http, Paginator) {
    $scope.query = 'Testing';
    var fetchFunction = function(offset, limit, callback) {
      $http.get('/search',
          {params: {query: $scope.query, offset: offset, limit: limit}})
          .success(callback);

    };

    $scope.searchPaginator = Paginator(fetchFunction, 10);
}]);
```

HTML 页面可以按如下方式使用分页服务：

```
<!DOCTYPE html>
```

```
<html ng-app="myApp">

<head  lang="en">
  <meta  charset="utf-8">
  <title>Pagination Service</title>
  <script
     src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
  </script>
  <script  src="pagination.js"></script>
  <script  src="app.js"></script>
</head>
<body  ng-controller="MainCtrl">
  <input type="text" ng-model="query">
  <ul>
    <li  ng-repeat="item in  searchPaginator.currentPageItems">
      {{item}}
    </li>
  </ul>
  <a href=""
ng-click="searchPaginator.previous()"
     ng-show="searchPaginator.hasPrevious()">&lt;&lt;  Prev</a>
  <a href=""
     ng-click="searchPaginator.next()"
     ng-show="searchPaginator.hasNext()">Next  &gt;&gt;</a>
</body>
</html>
```

与服务端协作以及登陆

我们最后一个示例将覆盖多个场景，大部分或所有都和$http 相关连。根据我们的经验，$http 服务是 AngularJS 中核心服务之一。但是可以扩展它，做一个 WEB 应用常见的需求，包括：

- 有一个公共的错误处理点
- 处理认证和登录跳转
- 和那些不能处理 JSON 的服务器交互
- 通过 JSONP 和外部服务通信（不同域名下）

因此，在这个示例（略做修改的）中，我们会有整个应用的骨架，它能够：

1. 用 butterbar 指令显示所有不可处理的错误（除了 401 错误），只有发生错误时，才会显

示在屏幕上

2. 有一个 ErrorService，用于在指令、视图、和控制器之间的交互
3. 每当服务器返回 401 响应时，就触发一个 event:loginRequired 的时间。然后通过监控整个应用的根控制器处理它
4. 处理那些带当前用户认证报文的请求发送到服务器

我们不会讲述整个应用（路由，模板等等），因为这些都相当简单。我们只会重点突出重要的片段（因此你可以拷贝和粘贴到你的代码库，再开始）。这样可以充分发挥作用。如果你想回顾下 Services 和 Factories 的定义，可以看下第七章。如果你想知道如何和服务器协作，那么可以参考下第五章。

首先来看下 Error Service

```
var servicesModule = angular.module('myApp.services', []);
servicesModule.factory('errorService', function() {
return {
  errorMessage: null, setError:
  function(msg) {
    this.errorMessage = msg;
  },
  clear: function() {
    this.errorMessage = null;

  }
};
});
```

Error Message 指令，实际上是独立于 Error Service，仅仅是找一个警告信息属性，然后绑定上去。如果警告信息可用就会有条件的展示出来。

```
// USAGE: <div alert-bar alertMessage="myMessageVar"></div>
angular.module('myApp.directives', []). directive('alertBar',
['$parse', function($parse) {
  return {
    restrict: 'A',
    template: '<div class="alert alert-error alert-bar"' +
      'ng-show="errorMessage">' +
      '<button type="button" class="close" ng-click="hideAlert()">' +
      'x</button>' +
      '{{errorMessage}}</div>',

    link: function(scope, elem, attrs) {
      var alertMessageAttr = attrs['alertmessage'];
      scope.errorMessage = null;

                                                                    }
                                                                  };
```

```
scope.$watch(alertMessageAttr, function(newVal) {
  scope.errorMessage = newVal;
});
scope.hideAlert = function() {
  scope.errorMessage = null;
  // Also clear the error message on the bound variable.
  // Do this so that if the same error happens again
  // the alert bar will be shown again next time.
  $parse(alertMessageAttr).assign(scope, null);
};
}]);
```

然后我们会郑家警告条到 HTML 模板，就像这样：

```
<div alert-bar alertmessage="errorService.errorMessage"></div>
```

我们需要确保，在添加前面的 HTML 之前，ErrorService 必须以 errorService 保存在控制器的作用域上。那就是说，如果 RootController 是这个有 AlertBar 的控制器，也就是：

```
app.controller('RootController',['$scope', 'ErrorService', function($scope, ErrorService) {
$scope.errorService = ErrorService;
});
```

这就给我们了一个大体上框架来显示和隐藏错误和警告。现在让我们看看通过拦截器如何捕获服务器端抛给我们的各种状态码：

```
servicesModule.config(function ($httpProvider) {
  $httpProvider.responseInterceptors.push('errorHttpInterceptor');
});

// register the interceptor as a service
// intercepts ALL angular ajax HTTP calls
servicesModule.factory('errorHttpInterceptor',
    function ($q, $location, ErrorService, $rootScope) {
  return function (promise) {
    return promise.then(function (response) {
      return response;
    }, function (response) {
      if (response.status === 401) {
        $rootScope.$broadcast('event:loginRequired');
        } else if (response.status >= 400 && response.status < 500)
        { ErrorService.setError('Server was unable to find' +
          ' what you were looking for... Sorry!!');

      }
      return $q.reject(response);
```

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

对于某些地方的控制器而言，现在所需要做的就是监听 loginRequired 事件，然后重定向到登录页（或者做些更复杂的事，如展示一个带登录选项的模态登录框）。

```
$scope.$on('event:loginRequired', function() {
$location.path('/login');
});
```

剩下的请求就需要认证。那就是说所有需要认证的请求都有一个 Authorization 的头信息，它有一个代表当前登录用户的值。由于这个每次都会改变，因此我们不会使用默认的 transformRequests，因为那些配置级别的改变。相反我们会封装$http 服务，创建我们自己的 AuthHttp 服务。

我们同样会有一个 Authentication 服务，负责存储用户的认证信息（只要你希望查询，正常作为登录的一部分）。AuthHttp 服务会引用到 Authentication 服务，然后添加必要的头信息从而认证请求。

```
// This factory is only evaluated once, and authHttp is memorized. That is,
// future requests to authHttp service return the same instance of authHttp
servicesModule.factory('authHttp', function($http, Authentication) {
  var authHttp = {};

  // Append the right header to the request
    var extendHeaders = function(config) { config.headers = config.headers || {};
    config.headers['Authorization'] = Authentication.getTokenType() +
        ' ' + Authentication.getAccessToken();

  };

  // Do this for each $http call
  angular.forEach(['get', 'delete', 'head', 'jsonp'], function(name) {
      authHttp[name] = function(url, config) { config =
      config || {}; extendHeaders(config);
      return $http[name](url, config);
    };
  });
angular.forEach(['post', 'put'], function(name) {
  authHttp[name] = function(url, data, config) {
    config = config || {};
    extendHeaders(config);
    return $http[name](url, data, config);
```

```
  };
});

return  authHttp;
});
```

任何需要认证的请求都是由 authHttp.get()发起的而不是$http.get()。只要认证服务里设置了正确的信息，你的请求都会通畅无阻。对于 Authentication 我们也用一个服务，以便在整个应用中信息是可用的，每当路由切换时每次都去查询一遍。

这几乎覆盖了这个应用中需要的所有片段。你应该只需要拷贝代码，粘贴到应用中，并让它可用。祝你好运！

总结

虽然到了本书的结尾，但是到现在基本覆盖了 AngularJS 中的东西。这本书的目的是提供一个扎实的基础，从中开始自己的探索，以及更善于使用 AngularJS 进行开发。为了达到这种程度，我们覆盖了所有的基本内容（和一些高级主题），同时提供了许多示例，我们可以沿着这条路线一直走过来。

我们做完了吗？不！还有很多东西需要学习，AngualarJS 背后是如何运作的。例如我们没有触及创建复杂的，相互依赖的指令。还有很多内容，三四本书是不够的。但是我们希望这本书给你一些自信，能够首先捕获更多复杂的需求。

我们花费了大量的时间来写这本书，真心希望在互联网上看到一些用 AngularJS 写的令人惊讶的应用