

第一章 AngularJS 简介

我们创造惊人的基于 Web 的应用程序的能力是令人难以置信的，但是创建这些应用程序时所涉及的复杂性也是让人不可思议的。我们的 Angular 团队希望减轻我们在参与开发 AJAX 应用程序时的痛苦。在 Google，我们曾经在构建像 Gmail，Maps，Calendar 以及其他大型 Web 应用程序时经历了最痛苦的教训。我想我们也许能够利用这些经验来帮助其他开发人员。

我们希望在编写 Web 应用程序时感觉更像是第一次我们编写了几行代码然后站在后面惊讶于它所发生的事情。我们希望编码的过程感觉更像是创造而不是试图满足 Web 浏览器的奇怪的内部运行工作。

与此同时，我们还希望我们所面对的工作环境来帮助我们作出设计选择，使应用程序的创建变得很简单并且从一开始就让人们很容易理解它，并且希望伴随着应用程序的不断成长，正确的设计选择会让我们的应用程序易于测试，扩展和维护。

我们试图在 Angular 这个框架中做到这些。我们也为我们已经取得的成果感到非常高兴。这很大程度上归功于 Angular 开源社区中每个成员的出色工作和相互帮助，同时也教会了我们很多东西。我们希望你也加入到我们的社区中来，并帮助我们一起努力让 Angular 变得更好。

你可以在我们的 [Github 主页](#) 的仓库中查看那些较大的或者较复杂的例子和代码片段，你也可以拉取分支以及自行研究这些代码。

目录

- [概念](#)
- [客户端模板](#)
- [模型/视图/控制器\(MVC\)](#)
- [数据绑定](#)
- [依赖注入](#)
- [指令](#)
- [示例：购物车](#)
- [小结](#)

概念

在你将使用的 **Angular** 构建应用的过程中有几个核心的概念。事实上，任何这些概念并不是我们新发明的。相反，我们从其他开发环境借鉴了大量成功的做法(经验)，然后使用包含 **HTML**，浏览器以及许多其他 **Web** 标准的方式实现了它。

客户端模板

多页面的 **Web** 应用程序都是通过装配和连接服务器上数据来创建 **HTML**，然后将构建完成的 **HTML** 页面发送到浏览器中。大多数的单页应用程序--也就是我们所知道的 **AJAX** 应用程序--从某种程度上讲，它的这一点一直做的很好。然而 **Angular** 以不同的方式实现了将模板和数据推送到浏览器中来装配它们。然后服务器角色只是为模板提供静态资源以及为模板适当地提供数据。

让我们来看一个例子，在 **Angular** 中如何在浏览器中组装模板和数据。按照惯例我们使用一个 **Hello, World** 的例子，但是注意这里并不是编写一个 **"Hello, World"** 的单一字符串，而是将问候 **"Hello"** 作为我们稍候可能会改变的数据来构建。

针对这个例子，我们在 `hello.html` 中来创建我们的模板：

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

接下来我们将逻辑编写在 `controllers.js` 中：

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

最后将我们 `hello.html` 载入任意浏览器中，我们将看到如图 1-1 所示的信息：

Hello, World.

图 1-1 Hello World

译注：你也可以自行修改 `controllers.js` 中的数据来查看效果。

与现在我们使用广泛的大多数方法相比，这里有一些有趣的事情需要注意：

- **HTML** 并没有类(class 属性)或者 **IDs** 来标识在哪里添加事件监听器。
- 当 `HelloController` 设置 `greeting.text` 为 `Hello` 时，我们并没有注册任何事件监听器或者编写任何回调函数。
- `HelloController` 只是一个很普通的 **JavaScript** 类，并且它并没有继承任何 **Angular** 所提供的信息。
- `HelloController` 获取了它所需要的 `$scope` 对象，我们无需创建它。
- 我们并没有自己手动的调用 `HelloController` 的构造函数，在这里暂时也不打算弄清楚什么时候调用它。

接下来我们会看到更多与传统开发方式之间的差异，但是现在我们应该清楚：

Angular 应用程序的结构与过去类似的应用程序是完全不同的。

那么为什么我们会做出这些设计选择以及 **Angular** 是如何工作的呢？接下来先让我们来看看 **Angular** 从其他地方借鉴的一些好的思想(概念/经验)。

模型/视图/控制器(MVC)

MVC 应用程序结构是 20 世纪 70 年代作为 **Smalltalk** 的一部分引入的。从 **Smalltalk** 开始，**MVC** 在几乎每一个涉及用户界面的桌面应用程序开发环境中都变得流行起来。无论你是使用 **C++**，**Java**，还是 **Object-C**，都可以找到使用 **MVC** 的场景。然而，直到最近，**MVC** 的思想才开始在国外的 **Web** 开发中应用。

MVC 背后的核心思想是你可以在你的代码中清晰的分离数据管理(模型)，应用程序逻辑(控制器)以及给用户呈现数据(视图)。

视图会从模型中获取数据来显示给用户。当用户通过点击或者输入操作与应用程序进行交互时，控制器就通过修改数据模型来响应用户的操作。最后，被修改的模型会通知视图它已经发生了变化，因此视图可以更新它所显示的信息。

在 **Angular** 应用程序中，视图就是文档对象模型 (**DOM**)，控制器是 **JavaScript** 类，最后模型中的数据便是存储在对象中的属性(属性值)。

JavaScript 并没有类的概念，这里的意思就是用构造函数的方式来处理控制器部分，其他地方所提及的 JavaScript 类的概念读者需要自行甄别。

我们认为 MVC 的灵活性主要主要表现在以下几方面。首先，它给你提供了一个只能的模型用于告诉在哪里存储什么样的数据，因此你不需要每次都重新构造它。当其他人参与到你的项目中合作开发时，便能够即时理解你已经编写好的部分，因为他们会知道你使用了 MVC 结构来组织你的代码。也许最重要的是，它给你提供了一个极大的好处，是你的程序更易于扩展，维护和测试。

译注

1. MVC 是软件工程中的一种软件架构模式 - [MVC](#)。
2. Smalltalk 是一门面向对象的程序设计语言 - [Smalltalk](#)。

数据绑定

之前常见的 AJAX 单页应用程序，像 Rails，PHP 或者 JSP 平台都是在通过在将页面发送给用户显示之前将数据合并到 HTML 字符串中来帮助我们创建用户界面 (UI)。

像 jQuery 这样的库也是将模型扩展到客户端并让我们使用类似的风格，但是它只有单独更新部分 DOM 的能力，而不能更新整个页面。在 AngularJS 中，我们将数据合并到 HTML 模板的字符串中，然后通过在一个占位元素中设置 `innerHTML` 将返回的结果插入到我们的目标 DOM 元素中。

这一切都工作得很好，但是当你希望插入新的数据到用户界面 (UI) 中时，或者基于用户的输入改变数据，你需要做一些相当不平凡的工作以确保你的数据变为正确的状态，无论数据是在用户界面中 (UI) 还是 JavaScript 属性中。

但是如果我们可以不用编写代码就能处理好所有这些工作会怎样呢？如果我们可以只需声明用户界面的哪些部分对应哪些 JavaScript 属性并且让它们自动同步又会怎样呢？这种编程风格被称为数据绑定。由于 MVC 可以在我们编写视图和模型时减少代码，因此我们将它引入到了 Angular 中。大部分将数据从一处迁移到另一处的工作都会自动完成。

下面来看看这一行为，我们继续使用前面的 "Hello World" 的例子，但是我们会让它"动"起来。原来是一旦 `HelloController` 设置了其模型 `greeting.text` 的值，它便不再会改变。首先让我们通过添加一个根据用户输入改变 `greeting.text` 值的文本输入框来改变这个例子，让它能够"动"起来。

下面是新的模板：

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

`HelloController` 控制器可以保持不变。

将它载入到浏览器中，我们将看到如图 1-2 所示屏幕截图：

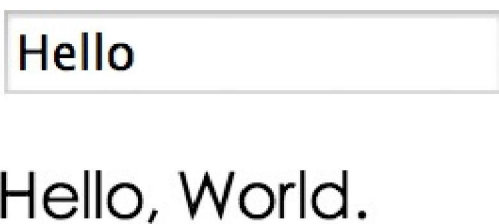


图 1-2 应用程序的默认状态

如果我们使用 'Hi' 文本替换输入框中的 'Hello'，我们将看到如图 1-3 所示截图：

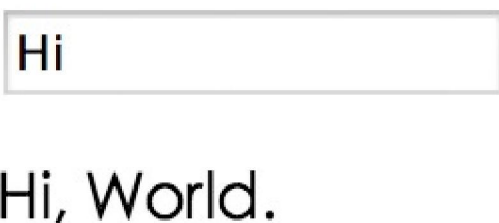


图 1-3 改变文本框值之后的应用程序

我们并没有在输入字段上注册一个改变值的事件监听器，我们有一个将会自动更新的 UI。同样的情况也适用于来自服务器端的改变。在我们的控制器中，我们可以构造一个服务器端的请求，获取响应，然后设置 `$scope.greeting.text` 等于它返回的值。**Angular** 会自动更新文本输入框和双大括号中的 `text` 字段为该返回值。

依赖注入

之前我们提到过，在 `HelloController` 中有很多东西都可以重复，在这里我们并没有编写。例如，`$scope` 对象会将数据绑定自动传递给我们；我们不需要通过调用任何函数来创建它。我们只是通过将它放置在 `HelloController` 的构造器中来请求它。

正如我们将在后面的章节中会看到，`$scope` 并不是我们唯一可以访问的东西。如果我们希望将数据绑定到用户浏览器的 URL 地址中，我们可以通过将数据绑定植入我们控制器的 `$location` 中来访问管理数据。就像这样：

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

这个神奇的效果是通过 **Angular** 的依赖注入系统实现的。依赖注入让我们遵循这种开发风格，而不是创建依赖，我们的类只需要知道它需要什么。

这个效果遵循一个被称为得墨忒耳定律的设计模式，也被称作最少知识原则。由于我们的 `HelloController` 的工作只是设置 `greeting` 模型的初试状态，这种模式会告诉你无需担心其他的事情，例如 `$scope` 是如何创建的，或者在哪里可以找到它。

这个特性并不只是通过 **Angular** 框架创建的对象才有。你最终创建的任何对象和服务也可以以同样的方式注入。

指令

Angular 最好的部分之一就是你可以编写你的模板如同 **HTML** 一样。之所以可以这样做，是因为在这个框架的核心部分我们已经包含了一个强大的 **DOM** 解析引擎，它允许你扩展 **HTML** 的语法。

我们已经在我们的模板中看到了一些不属于 **HTML** 规范的新属性。例如包含在双花括号中的数据绑定，用于指定哪个控制器对应哪部分视图的 `ng-controller`，以及将输入框绑定到模型部分的 `ng-model`。我们称之为 **HTML** 扩展指令。

Angular 自带了许多指令以帮助你定义应用程序的视图。后面我们就会看到更多的指令。这些指令可以定义用来帮助我们定义常见的视图作为模板。它们可以用于声明设置你的应用程序如何工作或者创建可复用的组件。

你并不仅限于使用 **Angular** 自带的指令。你也可以编写你自己的扩展 **HTML** 模板来做你想做的任何事。

示例：购物车

接下来让我们来看一个较大的例子，它展示了更多的 **Angular** 的能力。想象一下，我们要创建一个购物应用程序。在应用程序的某个地方，我们需要展示用户的购物车并允许他编辑。接下来我们直接跳到那部分。

```
<html ng-app>
<head>
<title>Your Shopping Cart</title>
</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($index)">Remove</button>
  </div>
  <script src="angular.js"></script>
  <script>
function CartController($scope){
  $scope.items = [
    {title: 'Paint pots', quantity: 8, price: 3.95},
    {title: 'Polka dots', quantity: 17, price: 12.95},
    {title: 'Pebbles', quantity: 5, price: 6.95}
  ];

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

最终用户界面截屏如图 1-4 所示：



图 1-4 购物车用户界面

下面是关于这里发生了什么的简短参考。本书其余的部分提供了更深入的讲解。

让我们从顶部开始：

```
<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>` 元素中，这就会告诉 **Angular** 我们希望它管理整个页面。这往往也是你所希望的，但是如果你是在现有应用程序中集成 **Angular** 并使用其他方式管理页面，那么你可能希望把它放在应用程序中的某个 `<div>` 元素中。

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

在 **Angular** 中，你用于管理页面某个区域的 **JavaScript** 类被称为控制器。通过在 `<body>` 标签中包含一个控制器，那么说明我的这个 `CartController` 将会管理 `<body>` 和 `</body>` 之间的所有东西。

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

`ng-repeat` 的意思就是给被称为 `items` 数组中的每个元素都复制一次当前 `<div>` 里面的 DOM 结构。在每一个复制的 `div` 副本中，我们都会给当前元素设置一个名为 `item` 的属性，这样我们就可以在模板中使用它。你可以看到，结果返回的三个

`<div>` 中的每一个都包含产品的标题，数量，单价，总价以及一个用于移除当前条目的按钮。

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

正如我们在 "Hello, World" 例子中所示，数据绑定通过 `{{ }}` 让我们将一个变量的值插入到页面某部分中并保持数据同步。完整的表达式 `{{item.title}}` 会以迭代的方式检索当前 `item`，然后将该 `item` 的 `title` 属性的内容插入到 DOM 中。

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

`ng-model` 在输入字段和 `item.quantity` 的值之间定义并创建了数据绑定行为。

`` 中的 `{{ }}` 设置了一个单项关联，它的意思就是"在这里插入一个值"。我们就是想要这个效果，但是应用程序也需要知道用户什么时候改变了商品数量，以便它可以改变选购商品的总价。

因此我们通过使用 `ng-model` 来同步模型中的变化。`ng-model` 声明了将 `item.quantity` 的值插入到文本域中，每当用户输入一个新的值时它也会自动更新 `item.quantity` 的值。

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

我们还希望单价和总价被格式化为美元形式。**Angular** 自带了一个被称为过滤器的特性来让我们转换文本，在这里我们捆绑了一个被称为 `currency` 的过滤器用于给我们处理这里的美元格式操作。在下一章我们将会看到更多的过滤器。

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

这允许用户通过点击产品旁边的 `remove` 按钮来从他们的购物车中移除所选择的商品条目。在这里我们设置它点击这个按钮时调用 `remove()` 函数。我们还给它传递了一个 `$index` 参数，这个参数包含了它在 `ng-repeat` 中的索引值，这样我们就会知道哪一项将会被移除。

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

这个 `CartController` 用于整个管理购物车应用的逻辑。这会告诉 **Angular**，在这里它要给控制器传递一个叫做 `$scope` 的参数。这个 `$scope` 用于帮助我们在用户界面中将数据绑定到元素中。

```
$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 `$scope.items`，我创建一个虚拟数据哈希表[数组]来表示用户的购物车。我们希望它可以用于 UI 中的数据绑定，因此将它添加到 `$scope` 中。

当然，真正的购物车应用不可能只是在内存中工作，它需要访问服务器中正确存储的数据。我们将在后面的章节中讨论这些。

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

我们还希望将 `remove()` 函数绑定在 UI 中使用，因此我们也将它添加到 `$scope` 中。对于这个内存版本的购物车，`remove()` 函数可以即时从数组中删除项目。由于 `<div>` 列表是通过 `ng-repeat` 创建的数据绑定，所以当项目消失时列表会自动收缩。记住，每当用户在 UI 界面上点击一个 **Remove** 按钮时，`remove()` 函数就会被调用。

小结

我们已经看到了 **Angular** 最基本的用法以及一些非常简单的例子。本书后面的部分将专注于介绍这个框架所提供的更多功能。

```
function ShoppingController($scope, $http){

    $http.get('/products').success(function(data, status, headers,
config){

        $scope.items = data;

    });

}
```

然后像这样在模板中使用它：

```
<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>

</body>
```

正如我们之前所学习到的，从长远来看我们将这项工作委托到服务器通信服务上可以跨控制器共享是明智的。我们将在第 5 章来看这个结构和全方位的讨论 `$http` 函数。

使用指令更新 DOM

指令扩展 HTML 语法，也是将行为与 DOM 转换的自定义元素和属性关联起来的方式。通过它们，你可以创建复用的 UI 组件，配置你的应用程序，做任何你能想到在模板中要做的事情。

你可以使用 **Angular** 自带的内置指令编写应用，但是你可能会希望运行你自己所编写的指令的情况。当你希望处理浏览器事件和修改 DOM 时，如果无法通过内置指令支持，你会知道是时候打破指令规则了。你所编写的代码在指令中，不是在控制器中，服务中，也不是应用程序的其他地方。

与服务一样，通过 **module** 对象的 API 调用它的 `directive()` 函数来定义指令，其中 `directiveFunction` 是一个工厂函数用于定义指令的功能(特性)。

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

appModule.directive('directiveName', directiveFunction);
```

编写指令工厂函数是很深奥的，因此在这本书中我们专门顶一个完整的一章。吊吊你的胃口，不过，我们先来看一个简单的例子。

HTML5 中有一个伟大的称为 `autofocus` 的新属性，将键盘的焦点放到一个 `input` 元素。你可以使用它让用户第一时间通过他们的键盘与元素交互而不需要点击。这是很好的，因为它可以让你声明指定你希望浏览器做什么而无需编写任何 **JavaScript**。但是如果你希望将焦点放到一些非 `input` 元素上，像链接或者任何 `div` 上会怎样？如果你希望它也能工作在不支持 HTML5 中会怎样？我们可以使用一个指令做到这一点。

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

```

appModule.directive('ngbkFocus', function(){

    return {

        link: function(scope, elements, attrs, controller){

            elements[0].focus();

        }

    };

});

```

这里，我们返回指令配置对象带有指定的 `link` 函数。这个 `link` 函数获取了一个封闭的作用域引用，作用域中的 **DOM** 元素，传递给指令的任意属性数组，以及 **DOM** 元素的控制器，如果它存在。这里，我们仅仅只需要获取元素并调用它的 `focus()` 方法。

然后我们可以像这样在一个例子中使用它：

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>

```

controller.js

```
function SomeController($scope) {  
  
    $scope.message = { text: 'nothing clicked yet' };  
  
    $scope.clickUnfocused = function() {  
  
        $scope.message.text = 'unfocused button clicked';  
  
    };  
  
    $scope.clickFocused = function {  
  
        $scope.message.text = 'focus button clicked';  
  
    }  
  
}  
  
var appModule = angular.module('app', ['directives']);
```

当载入页面时，用户将看到标记为"I'm very focused!"按钮带有高亮焦点。敲击空格键或者回车键将导致点击并调用 `ng-click`，将设置 `div` 的文本为"focus button clicked"。在浏览器中打开这个页面，我们将看到如图 2-4 所示的东西：



图 2-4 Focus directive

验证用户输入

Angular 带有几个适用于单页应用程序的不错的功能来自动增强 `<form>` 元素。其中之一不错的特性就是 Angular 让你在表单内的 `input` 中声明验证状态，并允许在整组元素通过验证的情况下才提交。

例如，如果我们创建一个登录表单，我们必须输入一个名称和 **email**，但是有一个可选的年龄字段，我们可以在他们提交到服务器之前验证多个用户输入。如下加载这个例子到浏览器中将显示如图 2-5 所示：



Sign Up

First name:

Last name:

Email:

Age:

图 2-5. Form validation

我们还希望确保用户在名称字段输入文本，输入正确形式的 **email** 地址，以及他可以输入一个年龄，它才是有效的。

我们可以在模板中做到这一点，使用 Angular 的 `<form>` 扩展和各个 `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，在某些字段上我们使用了 HTML5 中的 `required` 属性以及 `email` 和 `number` 类型的 `input` 元素来处理我们的验证。这对于 Angular 来说是很好的，在老式的不支持 HTML5 的浏览中，Angular 将使用形式相同职责的指令。

然后我们可以通过改变引用它的形式来添加一个控制器处理表单的提交。

```

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

```

在控制器里面，我们可以通过一个称为 `$valid` 的属性来访问这个表单的验证状态。当所有的表单 `input` 通过验证的时候，Angular 将设置它 (`$valid`) 为 `true`。我们可以使用 `$valid` 属性做一些时髦的事情，比如当表单还没有完成时禁用提交按钮。我们可以防止表单提交进入无效状态，通过给提交按钮添加一个 `ng-disabled`。

```

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

```

最后，我们可能希望控制器告诉用户她已经添加成功了。我们的最终模板看起来像这样：

```

<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></div>

</form>
```

接下来是控制器:

```
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!';

    };

}
```

小结

在前两章中, 我们看到了 **Angular** 中所有最常用的功能(特性). 对每个功能的讨论, 许多额外的细节信息都没有覆盖到. 在下一章, 我们将让你通过研究一个典型的工作流程了解更多的信息.

第二章 **Angular** 应用程序剖析

不像典型的库，你需要挑选你喜欢的功能，在 **Angular** 中所有的东西都被设计成一个用于协作的套件。在本章中我们将涵盖 **Angular** 中所有的基本构建块，这样你就可以理解如何将它们组合在一起。这些块都将在后面的章节中有更详细的讨论。

目录

- [启用 Angular](#)
- [加载脚本](#)
- [使用 ng-app 声明 Angular 的界限](#)
- [模型/视图/控制器](#)
- [模板和数据绑定](#)
- [显示文本](#)
- [表单输入](#)
- [不唐突 JavaScript 的一些话](#)
- [列表, 表格和其他重复的元素](#)
- [隐藏与显示](#)
- [CSS 类和样式](#)
- [src 和 href 属性注意事项](#)
- [表达式](#)
- [分离用户界面\(UI\)和控制器职责](#)
- [使用作用域发布模型数据](#)
- [使用\\$watch 监控模型变化](#)
- [watch\(\)中的性能注意事项](#)
- [使用模块组织依赖](#)
- [使用过滤器格式化数据](#)
- [使用路由和\\$location 更新视图](#)
- [index.html](#)
- [list.html](#)
- [detail.html](#)
- [controllers.js](#)
- [对话服务器](#)
- [使用指令更新 DOM](#)
- [index.html](#)
- [controller.js](#)
- [验证用户输入](#)
- [小结](#)

启用 Angular

任何应用程序都必须做两件事来启用 Angular:

1. 加载 `angular.js` 库
2. 使用 `ng-app` 指令来告诉 Angular 它应该管理哪部分 DOM

加载脚本

加载库很简单, 与加载其他任何 JavaScript 库遵循同样的规则. 你可以从 Google 的内容分发网络(CDN)中载入脚本, 就像这样:

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

推荐使用 Google 的 CDN. Google 的服务器很快, 并且这个脚本是跨应用程序缓存的. 这意味着, 如果你的用户有多个应用程序使用 Angular, 那么他将只需要下载脚本一次. 此外, 如果用户访问过其他使用 Google CDN 连接 Angular 的站点, 那么他在访问你的站点时就不需要再次下载该脚本.

如果你更喜欢本地主机(或者其他方式), 你也可以这样做. 只需要在 `src` 中指定正确的地址.

使用 ng-app 声明 Angular 的界限

原文是 Boundaries, 意思是声明应用程序的作用域, 即 Angular 应用程序的作用范围.

`ng-app` 指令用于让你告诉 Angular 你期望它管理页面的哪部分. 如果你在创建一个完全的 Angular 应用程序, 那么你应该在 `<html>` 标签中包含 `ng-app` 部分, 就像这样:

```
<html ng-app>

...

</html>
```

这会告知 Angular 要管理页面中的所有 DOM 元素.

如果你有一个现有的应用程序，要求使用其他的技术来管理 DOM，例如 **Java** 或者 **Rails**，你可以通过将它放置在页面的一些元素例如 `<div>` 中来告诉 **Angular** 只需要管理页面的一部分即可。

```
<html>

...

  <div ng-app>

    ...

  </div>

...

</html>
```

模型/视图/控制器

在第一章中，我们提到 **Angular** 支持模型/视图/控制器的应用程序设计风格。虽然在设计你的 **Angular** 应用程序时有很大的灵活性，但是总是别有一番风味的：

- 模型包含代表你的应用程序当前状态的数据
- 视图显示数据
- 控制器管理你的模型和视图之间的关系

你需要使用对象属性的方式创建模型，或者只包含原始类型的数据。这里并没有特定的模型变量。如果你希望给用户显示一些文本，你可以使用一个字符串，就像这样：

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

你可以通过编写一个模板作为 **HTML** 页面，并从模型中合并数据的方式来创建视图。正如我们已经看过的，你可以在 **DOM** 中插入一个占位符，然后再像这样设置它的文本：

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

我们调用这个双大括号语法来插入值，它将插入新的内容到一个现有的模版中。

控制器就是你编写用于告诉 **Angular** 哪些对象和原始值构成你的模型的类，通过这些对象或者原始值分配给 `$scope` 对象传递到控制器中。

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

把他们放在一起，我们得到如下代码：

```
<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>
```

将它载入到浏览器中，你就会看到

'You have started you journey'

虽然这个原始风格的模型工作在简单的情况下，然而大多数的应用程序你都希望创建一个模型对象来包裹你的数据。我们将创建一个信息模型对象，并用它来存储我们的 `someText`。因此不是这样的：

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

你应该这样编写:

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

然后在你的模板中这样使用:

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

正如我们后面会看到, 当我们讨论 `$scope` 对象时, 像这样创建一个模型对象将有利于防止从 `$scope` 对象的原型中继承的意外行为.

我们正在讨论的这些方法从长远看来能够帮助你, 在上面的例子中, 我们在全局作用域中创建了 `TextController`. 虽然这是一个很好的例子, 但是正确定义一个控制器的做法应该是将它作为模块的一部分, 它给你的应用程序部分提供了一个命名空间. 更新之后的代码看起来应该是下面这样.

```
<html ng-app="myApp">  
  
  <body ng-controller="TextController">  
  
    <p>{{someText.message}}</p>  
  
    <script  
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.1/angular.min.js">  
  </script>  
  
  <script>  
  
    var myAppModule = angular.module('myApp', []);
```

```
myAppModule.controller('TextController', function($scope){

    var someText = {};

    someText.message = 'You have started your journey';

    $scope.someText = someText;

});

</script>

</body>

</html>
```

在这个版本中，我们声明模块中 `ng-app` 元素的名称为 `myApp`。然后我们调用 Angular 对象创建了一个名为 `myApp` 的模块，然后调用模块的 `controller` 方法并将我们的控制器函数传递给它。

一会儿我们就会知道为什么，以及如何获取所有的模块。但是现在，只需要记住将所有的信息都保存在全局的命名空间中是一件好事，并且这也是我们使用模块的机制。

模板和数据绑定

在 Angular 应用程序中模板只是 HTML 文档，就像我们从服务端载入或者定义在 `<script>` 标签中的任何其他静态资源一样。在你的模板中定义用户界面，可以使用标准的 HTML 加 Angular 指令来定义你所需要的 UI 组件。

一旦进入浏览器中，Angular 就会进入到你的整个应用程序中通过合并模板和数据的方式来解析这些模板。在第一章中我们已经在购物车应用中看过了显示一个项目列表的例子。

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

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

    ...

</div>
```

这里，它只是外层 `<div>` 的一个副本，里面所有的一切，都一一对应 `items` 数组中的每个元素。

那么这些数据从哪里来？在我们的购物车例子中，在我们的代码中我们只将它定义为一个数组。对于你开始创建一个 UI 并希望测试它是如何工作的，这是非常合适的。然而大多数的应用程序，将使用一些服务器上的持久性数据。在浏览器中你的应用程序连接你的服务器，用户在页面上请求他们所需要的一切，然后 Angular 将它[请求的数据]与你的模板合并。

基本的运作流程看起来像这样：

1. 用户请求你的应用程序的第一个页面
2. 用户浏览器发出一个 HTTP 请求连接到你的服务器，然后加载包含模板的 *index.html* 页面
3. Angular 载入到页面中，等到页面完全加载，然后查询定义在模板范围内的 `ng-app`
4. Angular 遍历模板并查询指令和绑定。这将导致注册事件监听器和 DOM 操作，以及从服务器上获取初始数据。这项工作的最终结果是展示应用程序并将模板作为 DOM 转换为视图。
5. 连接到你的服务器加载你需要展示给用户所需的附加数据。

第 1 步至第 3 步是每个 Angular 应用程序的标准。第 4 步和第 5 步对你来说是可选的。这些步骤可以同步或者异步发生。出于性能的考虑，你应用程序所需的数据在第一个视图中[首屏]显示给用户，可以减少并避免重复的请求 HTML 模板。

通过使用 Angular 组织你的应用程序，你可以在你的应用程序中分离模板和数据。这样做的结果是这些模板是可以缓存的。在第一次载入之后，实质上浏览器中只需要请求新的数据了。正如 JavaScript, 图片, CSS 以及其他资源，缓存这些模板可以给你的应用程序提供更好的性能。

显示文本

你可以使用 `ng-bind` 指令在你 UI 的任何地方显示和更新文本。它有两种等价的形式。一种是我们见过的双花括号形式：

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

然后就是一个被称为 `ng-bind` 的基于属性的指令：

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

这两者的输出是等价的。如果模型中的变量 `greeting` 设置为 "Hi, there", Angular 将生成这样的 HTML：

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

浏览器将显示"Hi, There".

那么为什么你会使用上面的另外一种形式？我们创建的双括号插入值的语法读起来更加自然并且只需要更少的输入。虽然两种形式产生相同的输出，但使用双花括号语法，加载你应用程序的第一个页面 `index.html` 时，在 **Angular** 替换花括号中的数据之前，用户可能会看到一个未渲染的模板。随后的视图将不会经历这一点。

原因是浏览器加载 **HTML** 页面，渲染它，直到那时 **Angular** 才可能准备解析它们。

好消息是你仍然可以在大多数模板中使用 `{{ }}`。然而，在你的 `index.html` 页面中绑定数据，应该使用 `ng-bind`。这样，直到数据加载完你的用户将什么也看不到。

表单输入

在 **Angular** 中处理表单元素是很简单的。正如我们见过的几个例子，你可以使用 `ng-model` 属性绑定到你的模型属性元素上。这适用于所有标准的表单元素，例如文本输入框，单选按钮，复选框等等。我们可以像这样绑定一个复选框到一个属性：

```
<form controller="SomeController">

  <input type="checkbox" ng-model="youCheckedIt">

</form>
```

这意味着：

1. 当用户选择复选框，`SomeController` 的 `$scope` 中一个名为 `youCheckedIt` 的属性将变成 `true`。取消选择时使 `youCheckedIt` 变成 `false`。
2. 如果你在 `SomeController` 中设置 `$scope.youCheckedIt` 为 `true`，这个复选框在 **UI** 中会被自动选择。设置它为 `false` 则取消选择。

现在我想说的是我们真正想要的是，当用户做了一些什么事情时作出响应。对于文本输入框元素，你使用 `ng-change` 属性来指定一个控制器方法，那么无论什么时候用户改变输入框的值时，这个控制器方法都应该被调用。让我们做一个简单的计算器来帮助用户自己理解他们需要多少钱才能得到某些东西：

```
<form ng-controller="StartupController">
```



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

对于我们这个简单的例子，让我们只设置输出用户预算十倍的值。我们还将设置一个默认为 0 的值来开始：

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

然而，前面的代码中有一个潜在的策略问题。问题是当用于在文本输入框中输入时我们只是重新计算了所需的金额。如果这个输入框只在用户在这个特定的输入框中输入时更新，这工作得很好。但是如果其他的输入框也在模型中绑定了这个属性会怎样呢？如果它从服务器获取数据来更新又会怎样？

无论这个字段如何更新，我们要使用一个名为 `$watch()` 的 `$scope` 函数 [`$scope` 对象的方法]。我们将在本章的后面详细讨论 `watch` 方法。基本的用法是，可以调用 `$watch()` 并给他传递一个监控表达式和一个用于响应表达式变化的回调函数。

在这种情况下，我们希望监控 `funding.startEstimate` 以及每当它改变时调用 `computeNeeded()`。然后我们使用这个方法重写了 `StartUpController`。

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

```

```

$scope.computeNeeded = function(){

    $scope.funding.needed = $scope.funding.startingEstimate * 10;

};

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

}

```

注意引号中的监控表达式。是的，它是一个字符串。这个字符串是评估某些东西价格的 **Angular** 表达式。表达式可以进行简单的运算和访问 `$scope` 对象的属性。在本章的后面我们会涵盖更多关于表达式的信息。

你也可以监控一个函数返回值，但是它并不会监控 `funding.startingEstimate`，因为它赋值为 0，并且 0[初始值]不再会改变。

然后，由于每当我们的 `funding.startingEstimate` 改变时 `funding.needed` 都会自动更新，我们可以像这样编写一个更简单的模板。

```

<form ng-controller="StartupController">

    Starting: <input ng-model="funding.startingEstimate">

    Recommendation: {{funding.needed}}

</form>

```

在某些情况下，你并不希望每一个改变都发生响应，相反，你希望等到用户来告诉你它准备好了。例如可能完成购买或者发送一个聊天记录。

如果你的表单中有一组输入框，那么你可以在这个表单上使用 `ng-submit` 指令给它指定一个提交表单时的回调函数。我们可以让用户通过点击一个按钮请求帮助他们启动应用的方式来扩展上面的例子：

```

<form ng-submit="requestFunding()" ng-controller="StartupController">

    Starting: <input ng-change="computeNeeded()" ng-
model="funding.startingEstimate">

    Recommendation: {{funding.needed}}

```

```

        <button>Fun my startup</button>

    </form>

    function StartUpController($scope){

        $scope.computeNeeded = function(){

            $scope.funding.needed = $scope.funding.startingEstimate * 10;

        };

        $scope.requestFunding = function(){

            window.alert("Sorry, please get more customers first.");

        };

    }

```

当尝试提交这个表单时, `ng-submit` 指令也会自动阻止浏览器处理其默认的 `POST` 行为.

原文此处有错误, 表单提交的默认行为是 `GET`.

在需要处理其他事件的情况下, 就像当你想要提供交互而不是提交表单一样, **Angular** 提供了类似于浏览器原生事件属性的事件处理指令. 对于 `onclick`, 你应该使用 `ng-click`. 对于 `ondblclick` 你应该使用 `ng-dblclick` 等等.

我们可以尝试最后一次扩展我们的计算器启动应用, 使用一个重置按钮用于将输入框的值重置为 0.

```

<form ng-submit="requestFunding()" ng-controller="StartUpController">

    Starting: <input ng-change="computeNeeded()" ng-
model="funding.StartingEstimate">

    Recommendation: {{funding.needed}}

    <button>Fund my startup!</button>

    <button type="button" ng-click="reset()">Reset</button>

</form>

```

```
function StartUpController($scope){

    $scope.computeNeeded = function(){

        $scope.funding.needed = $scope.funding.startingEstimate * 10;

    };

    $scope.requestFunding = function(){

        window.alert("Sorry, please get more customers first");

    };

    $scope.reset = function(){

        $scope.funding.startingEstimate = 0;

    }

}
```

不唐突 JavaScript 的一些话

在你 JavaScript 开发生涯的某些时刻，有人可能会告诉你，你应该编写"不唐突的 JavaScript"，在你的 HTML 中使用 `click`, `mousedown` 以及其他类似的内联事件处理程序是不好的。那么他是正确。

不唐突的 JavaScript 思想已经有很多解释，但是其编码风格的原理大致如下：

1. 不是每个人的浏览器都支持 JavaScript. 让每个人都能够看到你所有的内容和使用你的应用程序，而不需要在浏览器中执行代码。
2. 有些人使用的浏览器工作方式不同. 视障人员使用的屏幕阅读器和一些手机用户不能使用网站的 JavaScript.
3. JavaScript 在不同的平台工作机制不一样. IE 浏览器通常是罪魁祸首. 你需要根据浏览器的不同而使用不同的事件处理代码。

4. 这些事件处理程序引用全局命名空间中的函数。当你尝试整合其他库中的同名函数时，它会让你头疼。
5. 这些事件处理程序合并了结构和行为。这使你的代码更加难以维护，扩展和理解。

总体来看，当你按照这种风格编写 **JavaScript** 代码，一切都很好。然而有一件事并不是好的，那就是代码的复杂度和可读性。并不是给元素声明事件处理程序不起作用，你通常给这些元素分配了 **ID**，获得这些元素的引用，并给它设置了事件处理的回调函数。你可以发明一个结构只用于清晰的创造它们之间的关联，但大多数应用程序结束于设置在各处的事件处理函数。

在 **Angular** 中，我们决定重新审视这个问题。

在这些概念诞生以来世界就已经改变了。第 1 点，这类有趣的群体已经不再有了。如果你运行的浏览器不支持 **JavaScript**，那么你应该去使用 20 世纪 90 年代创建的网站。至于第 2 点，现代的屏幕阅读器已经跟上来了。随着 **RAIA** 语义标签的正确使用，你可以创造易访问的富 **UI** 应用。现在手机上运行 **JavaScript** 与也能台式机相提并论了。

因此现在的问题是：重新恢复内联技术来解决我们第 3 点和第 4 点的可读性和简洁性的问题吗？

正如前面所提到的，对于大多数的内联事件处理程序，**Angular** 都有一个等价形式的 `ng-eventhandler="expression"` 来替代 `click`, `mousedown`, `change` 等事件处理程序。当用户点击一个元素时，如果你希望得到一个响应，你只需要简单的使用 `ng-click` 这样的指令：

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

你的大脑里可能会说“不，这样并不好”？好消息是你可以放松下来。这些指令不同于它们事件处理程序的前身(标准事件处理程序的原始形式)：

- 在每个浏览器中的行为一致。**Angular** 会给你处理好差异。
- 不会在全局命名空间操作。你所指定的表达式仅仅能够访问元素控制器作用域内的函数和数据。

最后一点听起来可能有点神秘，因此让我们来看一个例子。在一个典型的应用程序中，你会创建一个导航栏和一个随着你从导航栏选择不同菜单而变化的内容区。我们可以这样编写它的框架：

```

<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>

```

这里当用户点击 `navbar` 中的 `` 和 `content` 区中的 `<div>` 时都会调用一个称为 `doSomething()` 的函数。作为开发人员，你设置该函数调用你的控制器中的代码引用。它们可能是相同或者不同的函数：

```

function NavController($scope){

    $scope.doSomething = doA;

}


function ContentAreaController($scope){

    $scope.doSomething = doB;

}

```

这里，`doA()` 和 `doB()` 函数可能时相同或者不同的，取决于你给它们的定义。

现在我们还剩下第 5 点，合并结构和行为。这是一个有争议的话题，因为你不能指出任何负面的结果，但它与我们大脑里所想的合并表现职责和应用程序逻辑的行为非常类似。当人们谈及关于标记结构和行为分离的时候，这当然会有负面的影响。

如果我们的系统面临这种耦合问题时，这里有一个简单的测试可以帮助我们找出来：我们可以给我们的应用程序逻辑创建一个单元测试，而不需要 DOM 的存在。

在 **Angular** 中，是的，我们可以在控制器中只编写包含业务逻辑的代码而不必引用 DOM。在我们之前编写的 **JavaScript** 中，这个问题在事件处理程序中是不存在的。注意，在这里以及在这本书的其他地方，目前我们所编写的控制器中，都没有引用 DOM 和任何 DOM 事件处理程序。你可以很轻松创建出这些不带 DOM 的控制器。所有的元素定位和事件处理程序都发生在 **Angular** 中。

对于这个问题在编写单元测试时，如果你需要 DOM，你在测试中创建它，只会增加测试程序的复杂度。当你的页面发生变化时，你需要在你的测试中改变 DOM，这样只会带来更多的维护工作。最后，访问 DOM 是很慢的，测试缓慢意味着反馈不会及时以及最终解析都是缓慢的。**Angular** 的控制器测试并没有这些问题。

因此你可以很轻松的声明事件处理程序的简单性和可读性，毫无罪恶感的违反最佳实践。

列表，表格和其他重复的元素

最有用可能就是 **Angular** 指令，`ng-repeat` 对于集合中的每一项都创建一次一组元素的一份副本。你应该在你想创建列表问题的任何地方使用它。

比如说我们给老师编写一个学生花名册的应用程序。我们可能从服务器获得学生的数据，但是在这个例子中，我们只在 **JavaScript** 将它定义为一个模型：

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

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

我们可以像下面这样来显示学生列表：

```
<ul ng-controller="">
```

```
<li ng-repeat="student in students">

    <a href="/student/view/{{student.id}}">{{student.name}}</a>

</li>

</ul>
```

`ng-repeat` 将会制作标签内所有 HTML 的副本, 包括标签内的东西. 这样, 我们将看到:

- Mary Contrary
- Jack Sprat
- Jill Hill

分别链接到 `/student/view/1`, `/student/view/2`, `/student/view/3`.

正如我们之前所见, 改变学生数组将会自动改变渲染列表. 如果我们做一些例如插入一个新的学生到列表的事情:

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

function StudentListController($scope){

    $scope.students = students;

    $scope.insertTom = function(){

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

    };

}
```

然后在模板中添加一个按钮来调用:

```
<ul ng-controller="">
```



```

    <li ng-repeat="student in students">

        <a href="/student/view/{{student.id}}">{{student.name}}</a>

    </li>

</ul>

<button ng-click="insertTom()">Insert</button>

```

现在我们可以看到:

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

`ng-repeat` 指令还通过 `$index` 给你提供了当前元素的索引, 如果是集合中第一个元素, 中间的某个元素, 或者是最后一个元素使用 `$first`, `$middle` 和 `$last` 会给你提供一个布尔值.

你可以想象使用 `$index` 来标记表格中的行. 给定一个这样的模板:

```

<table ng-controller="AlbumController">

    <tr ng-repeat="track in album">

        <td>{{ $index + 1 }}</td>

        <td>{{ track.name }}</td>

        <td>{{ track.duration }}</td>

    </tr>

</table>

```

这是控制器:

```

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;  
  
};
```

我们得到如下结果:

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

隐藏与显示

对于菜单, 上下文敏感的工具[原文:*context-sensitive tools*]以及其他许多情况, 显示和隐藏元素是一个关键的特性. 正如在 Angular 中, 我们基于模型的变化触发 UI 的改变, 以及通过指令将改变反映到 UI 中.

这里, `ng-show` 和 `ng-hide` 用于处理这些工作. 它们基于传递给它们的表达式提供显示和隐藏的功能. 即, 当你传递的表达式为 `true` 时 `ng-show` 将显示元素, 当为 `false` 时则隐藏元素. 当表达式为 `true` 时 `ng-hide` 隐藏元素, 为 `false` 时显示元素. 这取决于你使用哪个更能表达的你意图.

这些指令通过适当的设置元素的样式为 `display: block` 来显示元素, 设置样式为 `display: none` 来隐藏元素. 让我们看以个正在构建的 Death Ray 控制板的虚拟的例子:

```
<div ng-controller="DeathrayMenuController">  
  
    <p><button ng-click="toggleMenu()">Toggle Menu</button></p>  
  
    <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

};

```

CSS 类和样式

显而易见，现在你可以在你的应用程序通过使用 `{{ }}` 插值符号绑定数据的方式动态的设置类和样式。甚至你可以在你的应用程序中组成匹配类名。例如，你想根据条件禁用一些菜单，你可以像下面这样从视觉上显示给用户。

有如下 CSS:

```

.menu-disabled-true {

    color: gray;

}

```

你可以使用下面的模板在你的 `DeathRay` 指示 `stun` 函数来禁用某些元素:

```

<div ng-controller="DeatrayMenuController">

    <ul>

        <li class="menu-disabled-{{isDisabled}}" ng-
click="stun()">Stun</li>

        ...

    </ul>

</div>

```

你可以通过控制器适当的设置 `isDisabled` 属性的值:

```
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` 将设置为 `menu-disabled-` 加 `$scope.isDisabled` 的值. 因为它初始化为 `false`, 默认情况下结果为 `menu-disabled-false`. 而此时这里没有与 `CSS` 规则匹配的元素, 则没有效果. 当 `$scope.isDisabled` 设置为 `true` 时, `CSS` 规则将变成 `menu-disabled-true`, 此时则调用规则使文本为灰色.

这种技术也同样适用于嵌入内联样式, 例如 `style="{{some expression}}"`.

虽然想法很好, 但是这里有一个缺点就是它使用了一个水平分割线来组合你的类名. 虽然在这个例子中很容易理解, 但是它可能很快就会变得难以管理, 你必须不断的阅读你的模板和 `JavaScript` 来正确的创建你的 `CSS` 规则.

因此, `Angular` 提供了 `ng-class` 和 `ng-style` 指令. 它们都接受一个表达式. 这个表达式的计算结果可以是下列之一:

- 一个使用空格分割类名的字符串
- 一个类名数组
- 类名到布尔值的映射

让我们想象一下, 你希望在应用程序头部的一个标准位置显示错误和警告给用户. 使用 `ng-class` 指令, 你可以这样做:

```
.error {

    background-color: red;

}

.warning {

    background-color: yellow;
```

```

}

<div ng-controller="HeaderController">

    ...

    <div ng-class="{error: isError, warning:
isWarning}">{{messageText}}</div>

    ...

    <button ng-click="showError()">Simulate Error</button>

    <button ng-click="showWarning()">Simulate Warning</button>

</div>

function HeaderController($scope){

    $scope.isError = false;

    $scope.isWarning = false;

    $scope.showError = function(){

        $scope.messageText = 'This is an error';

        $scope.isError = true;

        $scope.isWarning = false;

    };

    $scope.showWarning = function(){

        $scope.messageText = 'Just a warning. Please carry on';

        $scope.isWarning = true;

        $scope.isError = false;

    };

}

```

你甚至可以做出更漂亮的事情，例如高亮表格中选中的行。比方说，我们要构建一个餐厅目录并且希望高亮用户点击的那行。

在 CSS 中，我们设置一个高亮行的样式：

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

在模版中，我们设置 `ng-class` 为 `{selected: $index==selectedRow}`。当模型中的 `selectedRow` 属性匹配 `ng-repeat` 的 `$index` 时设置 `class` 为 `selected`。我们还设置一个 `ng-click` 来通知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>
```

在我们的 JavaScript 中，我们只设置虚拟的餐厅和创建 `selectRow` 函数：

```
function RestuarantTableController($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;  
    };  
  
}
```

src 和 href 属性注意事项

当数据绑定给一个 `` 或者 `<a>` 标签时，像上面一样在 `src` 或者 `href` 属性中使用 `{{ }}` 处理路径将无法正常工作。因为在浏览器中图片与其他内容是并行加载的，所以 Angular 无法拦截数据绑定的请求。

对于 `` 而言最明显的语法便是：

```

```

相反，你应该使用 `ng-src` 属性并像下面这样编写你的模板：

```

```

同样的道理，对于 `<a>` 标签你应该使用 `ng-href`：

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

表达式

表达式背后的思想是让你巧妙的在你的模板，应用程序逻辑以及数据之间创建钩子而与此同时防止应用程序逻辑偷偷摸摸的进入模版中。

直到现在，我们一直主要是引用原生的数据作为表达式传递给 Angular 指令。但是其实这些表达式可以做更多的事情。你可以处理简单的数学运算(+, -, /, *, %), 进行比较(==, !=, >, <, >=, <=), 执行布尔逻辑运算(&&, ||, !)以及按位运算(^, &, |)。你可以调用暴露在控制器的 `$scope` 对象上的函数，你还可以引用数据和对象表示法([, {}, ...)]。

下面都是有效表达式的例子：

```
<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>
```

```
</ul>

</div>
```

这里的第一个表达式 `recompute() / 10` 是有效的，是在模板中设置逻辑很好的好例子，但是应该避免这种方式。保持视图和控制器之间的职责分离可以确保它们容易理解和测试。

虽然你可以使用表达式做很多事情，它们由 **Angular** 自定义的解释器部分计算。他们并不使用 **JavaScript** 的 `eval()` 执行，`eval()` 有相当多的限制。

相反，它们使用 **Angular** 自带的自定义解释器执行。在里面，你不会看到循环结构 (`for`, `while` 等等)，流程控制语句 (`if-else`, `throw`) 或者改变数据的运算符 (`++`, `--`)。当你需要使用这些类型的运算时，你应该在你的控制器中使用指令进行处理。

尽管表达式在很多方面比 **JavaScript** 更加严格，但它们对 `undefined` 和 `null` 并不是很严格(更宽松)。模板只是简单的渲染一些东西，并不会抛出一个 `NullPointerException` 的错误。这样就允许你安全的使用模型而没有限制，并且只要它们得到数据填充就让它们出现在用户界面中。

分离用户界面(UI)和控制器职责

在你的应用程序中控制器有三个职责：

- 在你的应用程序的模型中设置初试状态.[初始化应用程序]
- 通过 `$scope` 暴露模型和函数到视图中。
- 监控模型的改变并触发行为。

对于第一点第二点在本章的已经看过更多例子。稍候我们会讨论最后一点。然而，控制器其概念上的目的，是提供代码或者执行用户与视图交互愿望的逻辑。

为了保持控制器的小巧和易于管理，我们建议你针对视图的每一个区域创建一个控制器。也就是说，如果你有一个菜单则创建一个 `MenuController`。如果你有一个面包屑导航，则编写一个 `BreadcrumbController`，等等。

你可能开始懂了，但是需要明确的将控制器绑定到一个指定的 **DOM** 块中用于管理它们。有两种主要的方式关联控制器与 **DOM** 节点，一种方式是在模板中指定一个 `ng-controller` 属性，另一种方式是通过 `route`(路由)关联一个动态加载的 **DOM** 模板片段，也称作视图。

我们将在本章的后面再讨论关于视图和路由的信息。

如果你的 UI 中有一个复杂的片段，你可以通过创建嵌套的控制器，通过继承树来共享模型和函数来保持你的代码间接性和可维护性。嵌套控制器很简单，你可以简单的在另一个 DOM 中分配一个控制器到一个 DOM 元素中做到这一点，就像这样：

```
<div ng-controller="ParentController">

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

</div>
```

虽然我们将这个表达为控制器嵌套，实际的嵌套发生在作用域中(`$scope` 对象中)。传递给嵌套控制器的 `$scope` 继承自父控制器的 `$scope` 原型，这意味着传递给 `ChildController` 的 `$scope` 将有权访问传递给 `ParentController` 的 `$scope` 的所有属性。

使用作用域发布模型数据

将 `$scope` 对象传递给我们的控制器便是我们将模型数据暴露给视图的机制。可能你的应用程序中还有其他的数据，但 Angular 中只能够通过 `scope` 访问它可以访问的模型部分的属性。你可以认为 `scope` 就是作为一个上下文环境用于在你的模型中观察变化的。

我们已经看过了很多明确设置作用域的例子，就像 `$scope.count = 5`。也有一些间接的方法在模板内设置其自身的模型。你可以像下面这样做：

1. 通过表达式。由于表达式运行在控制器的作用域关联的元素的上下文中，在表达式中设置属性与在控制器的作用域中设置一个属性一样。

也就是像这样：

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

这样做也有相同的效果：

```
<div ng-controller="CountController">

    <button ng-click="setCount()">Set count to three</button>

</div>
```

`CountController` 定义如下：

```
function CountController($scope){
```

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

1. 在表单的输入框中使用 `ng-model`. 在表达式中, 模型被指定为 `ng-model` 的参数也适用于控制器作用域范围. 此外, 这将在表单字段和你指定的模型之间创建一个双向数据绑定.

使用\$watch 监控模型变化

所有 `scope` 函数中最常用的可能就是`$watch`了, 当你的模型部分发生变化时它会通知你. 你可以监控单个对象属性, 也可以监控计算结果(函数), 几乎所有的事物都可当作一个属性或者一个 `JavaScript` 运算能够被访问. 该函数的签名如下:

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

每个参数的详细信息如下:

watchFn

这个参数是一个 `Angular` 字符串表达式或者是一个返回你所希望监控的模型当前值的函数. 这个表达式会被多次执行, 因此你需要确保它不会有副作用. 也就是说, 它可以被调用多次而不改变状态. 同样的原因, 监控表达式也应该是运算复杂度低的(执行简单的运算). 如果你传递一个字符串的表达式, 它将会对其调用的(执行的表达式)作用域中的有效对象求值.

watchAction

这是 `watchFn` 发生变化时会被调用的函数或者表达式. 在函数形式中, 它接受 `watchFn` 的新值, 旧值以及作用域的引用. 其签名就是 `function(newValue, oldValue, scope)`.

deepWatch

如果设置为 `true`, 这个可选的布尔参数用于告诉 `Angular` 检查所监控的对象中每一个属性的变化. 如果你希望监控数组的个别元素或者对象的属性而不是一个普通的

值, 那么你应该使用它. 由于 **Angular** 需要遍历数组或者对象, 如果集合(数组元素/对象成员)很大, 那么计算的代价会非常高.

当你不再想收到变化通知时, `$watch` 函数将返回一个注销监听器的函数.

如果我们像监控一个属性, 然后在稍后注销它, 我们将使用下面的方式:

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

让我们回顾一下第一章中完整的购物车示例. 比方说, 当用户在他的购物车中添加了超出 100 美元的商品时, 我们希望申请 10 美元的优惠. 我们使用下面的模板:

```
<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>
```

紧接着是 `CartController`, 它看起来像下面这样:

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

```

$scope.items = [

    {title: 'Paint pots', quantity: 8, price: 3.95},

    {title: 'Polka dots', quantity: 17, price: 12.95},

    {title: 'Pebbles', quantity: 5, price: 6.95}

];


$scope.totalCart = function(){

    var total = 0;

    for (var i = 0, len = $scope.items.length; i < len; i++){

        total = total + $scope.items[i].price*
$scope.items[i].quantity;

    }


    return total;

};


$scope.subtotal = function(){

    return $scope.totalCart() - $scope.bill.discount;

};


function calculateDiscount(newValue, oldValue, scope){

    $scope.bill.discount = newValue > 100 ? 10 : 0;

}


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

}

```

注意 `CartController` 的底部，我们给用于计算所购买商品总价的 `totalCart()` 的值设置了一个监控。每当这个值变化时，监控都会调用 `calculateDiscount()`，并且会给 `discount`(优惠项)设置一个适当的值。如果总价为\$100，我们将设置优惠为\$10。否则，优惠就为\$0。

你可以看到这个展示给用户的例子如图 2-1 所示：

Paint pots	8	\$3.95	\$31.60
Polka dots	17	\$12.95	\$220.15
Pebbles	5	\$6.95	\$34.75
Total: \$286.50			
Discount: \$10.00			
Subtotal: \$276.50			

图 2-1 Shopping cart with discount

`watch()`中的性能注意事项

前面例子会正确的执行，但是这里有一个潜在的性能问题。虽然并不明显，如果你在 `totalCart()` 中设置一个调试断点，你会发现在渲染页面时它被调用了 6 次。虽然在这个应用程序中你从来没有注意到它，但是在更多复杂的应用程序中，运行它 6 次可能是一个问题。

为什么是 6 次？其中 3 次我们可以很轻易的跟踪到，因为它分别在下面三个过程中运行一次：

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

然后是 **Angular** 再运行它们一次，因而带给我们 6 次运行。Angular 这样做是为了验证在你的模型中变化是否完全传播出去以及验证你的模型是否稳定。Angular 通过检查一份所监控属性的副本与它们当前值比较来确认它们是否改变。事实上，Angular 也可以运行它多达十次来确保是否完全传播开。如果发生这种情况，你可能需要依赖循环来修复它。

虽然你现在会担心这个问题，但是当你阅读完本书时它可能就不再是问题了。然而 Angular 不得不在 JavaScript 中实现数据绑定，我们一直与 TC39 的人共同努力实现一个底层的原生的 `Object.observe()`。一旦有了它，Angular 将自动使用 `Object.observe()` 随时随地呈现给你一个原生效率的数据绑定。

译注: [TC39](#)

在下一章中你会看到，Angular 有一个很好的 Chrome 调试扩展程序(Chrome 插件)Batarang，它将自动给你突出(高亮)昂贵的数据绑定(从性能的角度而言，表示数据绑定的方式并不是较好的方式)。

译注:

- [Batarang](#) - 这是一个 Angular 调试与性能监控工具。
- [Batarang-Github](#)

现在我们知道了这个问题，这里有一些方法可以解决它。一种方式是在 `items` 数组变化时创建 `$watch` 并且只重新计算 `$scope` 的 `total`, `discount` 和 `subtotal` 属性值。

做到这一点，我们只需要使用这些属性更新模板：

```
<div>Total: {{bill.total | currency}}</div>

<div>Discount: {{bill.discount | currency}}</div>

<div>Subtotal: {{bill.subtotal | currency}}</div>
```

然后，在 JavaScript 中，我们要监控 `items` 数组，以及调用一个函数来计算数组任意改变的总值：

```
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);

}

```

注意这里 `$watch` 指定了一个 `items` 字符串。这可能是因为 `$watch` 函数可以接受一个函数(正如我们之前那样)或者一个字符串。如果传递一个字符串给 `$watch` 函数, 在 `$scope` 调用的作用域中它将被当作一个表达式。

这种策略在你的应用程序中可能工作得很好。然而, 由我监控的是 `items` 数组, **Angular** 将会制作一个副本以供我们进行比较。对于一个较大的 `items` 清单, 如果我们在 **Angular** 每一次计算页面结果时只重新计算 `bill` 属性值, 它可能表现得更好。我们可以通过创建一个 `$watch` 来做到这一点, 它带有只用于重新计算属性的 `watchFn` 函数。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;

});

```

多个监控

如果你想监控多个属性或者对象，并且每当它们发生任何变化时都执行一个函数。你有两个基本的选择：

- 监控属性索引值。
- 把它们放入数组或者对象总并且将传递的 `deepWatch` 设置为 `true`。

译注：原文中两个选项排列顺序颠倒。译文中纠正了顺序并给出对应的信息。

在第一种情况下，如果作用域中有一个对象拥有两个属性 `a` 和 `b`，并且希望在发生变化时执行 `callMe()` 函数，你应该同时监控它们，就像这样：

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

当然，属性 `a` 和 `b` 可能在不同的对象中，只要你喜欢你也可以制作这个列表。如果列表很长，你可能更喜欢编写一个返回索引值的函数而不是依靠一个逻辑表达式。在第二种情况下，你可能希望监控 `things` 对象中的所有属性。在这种情况下，你可以这样做：

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

这里，通过将第三个参数设置为 `true` 来要求 **Angular** 遍历 `things` 对象的属性并在它们发生任何改变时调用 `callMe()`。这同样适用于数组，只是这里是针对一个对象。

使用模块组织依赖

在任何不平凡的应用程序中，在你的代码领域中弄清楚如何组织功能职责通常都是一项艰巨的任务。我们已经看到了控制器是如何到视图模板中给我们提供一个存放暴露正确数据和函数的区域。但是我们在哪里安置支持应用程序的其他代码呢？最明显的方式就是将它们放置在控制器中的函数中。

对于小型应用程序和目前我们所见过的例子,这种方式工作得很好，但是在实际的应用程序中很快变得难以管理。控制器将成为堆积一切以及我们需要做任何事情的垃圾场。它们可能很难理解，也可能很难改变(难以维护)。

引入模块。在你的应用程序功能区，它们提供了一种组织依赖的方式，以及一种自解决依赖的机制(也称为依赖注入[第一章中已经介绍了什么是依赖注入])。一般情况下，我们称之为依赖关系服务，它们给我们的应用程序提供特殊服务。

比如，如果在我们的购物网站中控制器需要从服务器获取一个出售项目列表，我们需要一些对象--让我们称之为 `Items`--注意这里是从服务器获取的项目。反过来，`Items` 对象，需要一些方式通过 `XHR` 或者 `WebSockets` 与服务器上的数据库通信。

不适用模块处理看起来像这样：

```
function ItemsViewController($scope){

    // 向服务器发起请求

    ...

    // 进入 Items 对象解析响应

    ...

    // 在$scope 中设置 Items 数组以便视图可以显示它

}
```

然而这确实能够工作，但是它存在一些潜在的问题。

- 如果一些其他的控制器还需要从服务器获取 `Items`，那我们现在要复制这个代码。这造成了维护的负担，如果我们现在要构造模式或者其他的变化，我们必须在好几个地方更新这个代码。
- 考虑到其他因素，如服务器验证，解析复杂度等等，这也是很难推断控制器对象职责界限的原因，代码也很难阅读。
- 对这段代码进行单元测试，我们需要一台实际运行的服务器或者使用 `XMLHttpRequest` 打补丁返回模拟数据。运行服务器进行测试将导致测试很慢，配置它很痛苦，它通常展示了测试中的碎片。而打补丁的方式解决了速度和碎片问题，但是这意味着你必须记住在测试中清理任何不定对象，这样就带来了额外的复杂度和脆弱性，因为它迫使你指定准确的线上版本的数据格式(每当格式变化时都需要更新测试)。

对于模块和从它们哪里获取的依赖注入，我们就可以编写更简洁的控制器，像这样：

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

现在你可能会问自己，'当然，这看起来很酷，但是这个 `Items` 从哪里来？'。前面的代码假设我们已经定义了作为服务的 `Items`。

服务是一个单独的对象(单例对象)，它执行必要的任务来支持应用程序的功能。

Angular 自带了很多服务，例如 `$location`，用于与浏览器中的地址交互，`$route`，用于基于位置(URL)的变化切换视图，以及 `$http` 用于与服务器通信。

你可以也应该创建你自己的服务去处理应用程序所有的特殊任务。在需要它们时服务可以共享给任何控制器。因此，当你需要跨控制器通信和共享状态时使用它们是一个很好的机制。**Angular** 绑定的服务都以 `$` 开头，所以你也能够命名它们为任何你喜欢的东西，这是一个很好的主意，以避免使用 `$` 开头带来的命名冲突问题。

你可以使用模块对象的 **API** 来定义服务。这里有三个函数用于创建通用服务，它们都有不同层次的复杂性和能力：

```
<table>

  <thead>

    <tr>

      <th>Function</th>

      <th>定义(Defines)</th>

    </tr>
```

```

</thead>

<tbody>

  <tr>

    <td>provider(name, Object/constructor())</td>

    <td>一个可配置的服务，带有复杂的创建逻辑。如果你传递一个对象，它应该有一个名为`$get`的函数，用于返回服务的实例。否则，Angular 会假设你传递了一个构造函数，当调用它时创建实例.</td>

  </tr>

  <tr>

    <td>factory(name, $get Function())</td>

    <td>一个不可配置的服务也带有复杂的创建逻辑。你指定一个函数，当调用时，返回服务实例。你可以认为这和<code>provider(name, { $get: $getFunction()})</code>一样</td>

  </tr>

  <tr>

    <td>service(name, constructor())</td>

    <td>一个不可配置的服务，其创建逻辑简单。就像<code>provider</code>的构造函数选项，Angular 调用它来创建服务实例.</td>

  </tr>

</tbody>

</table>

```

我们稍后再来看 `provider()` 的配置选项，现在我们先来使用 `factory()` 讨论前面的 **Items** 例子。我们可以像这样编写服务：

```

// Create a module to support our shopping views.

var shoppingModule = angular.module('ShoppingModule', []);

// Set up 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;

});

```

当 Angular 创建 `ShoppingController` 时，它会将 `$scope` 和我们刚才定义的新的 `Items` 服务传递进来。这是通过参数名称匹配完成的。也就是说，Angular 会看到我们的 `ShoppingController` 类的函数签名，并通知它(控制器)发现一个 `Items` 对象。由于我们定义 `Items` 为一个服务，它会知道从哪里获取它。

以字符串的形式查询这些依赖结果意味着作为参数注入的函数就像控制器的构造函数一样是顺序无关的。并不是必须这样：

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

我们也可以这样编写：

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

依然和我们所希望的功能一样。

为了在模板中使用它，我们需要告诉 `ng-app` 指令我们的模块名称，就像下面这样：

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

为了完成这个例子，我们可以这样实现模板的其余部分：

```
<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>

</body>
```

应用的返回结果看起来如图 2-2 所示：

Paint pots	Pots full of paint	3.95
Polka dots	Dots with polka	2.95
Pebbles	Just little rocks	6.95

图 2-2 Shop items

我们需要多少模块？

作为服务本身可以有依赖关系，Module API 允许你在的依赖中定义依赖关系。

在大多数应用程序中，创建一个单一的模块将所有的代码放入其中并将所有的依赖也放在里面足以很好的工作。如果你使用来自第三方库的服务或者指令，它们自带有其自身的模块。由于你的应用程序依赖它们，你可以引用它们作为你的应用程序的依赖。

举个例子，如果你要包含(虚构的)模块 `SnazzyUIWidgets` 和 `SuperDataSync`，应用程序的模块声明看起来像这样：

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

使用过滤器格式化数据

过滤器允许你在模板中使用插值方式声明如何转换数据并显示给用户。使用过滤器的语法如下：

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

其中表达式是任意的 **Angular** 表达式，`filterName` 是你想使用的过滤器名称，过滤器的参数使用冒号分割。参数自身也可以是任意有效的 **Angular** 表达式。

Angular 自带了几个过滤器，像我们已经看到的 `currency`：

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

这段代码显示如下：

\$12.9

你不仅限于使用绑定的过滤器(**Angular** 内置的)，你可以简单的编写你自己的过滤器。例如，如果我们想创建一个过滤器来让标题的首字母大写，我们可以像下面这样做：

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

```
        return words.join(' ');
    };

    return titleCaseFilter;
});
```

有一个像这样的模板:

```
<body ng-app="HomeModule" ng-controller="HomeController">

    <h1>{{pageHeading | titleCase}}</h1>

</body>
```

然后通过控制器插入 `pageHeading` 作为一个模型变量:

```
function HomeController($scope){

    $scope.pageHeading = 'behold the majesty of you page title';

}
```

我们会看到如图 2-3 所示的东西:




图 2-3 Title case filter

使用路由和\$location 更新视图

尽管 Ajax 从技术上讲是单页应用程序(理论上它们仅仅在第一次请求时加载 HTML 页面, 然后只需在 DOM 中更新区块), 我们通常会有多个子页面视图用于适当的显示给用户或者隐藏.

我们可以使用 Angular 的 `$route` 服务来给我们管理这个场景. 让你指定路由, 对于浏览器指向给定的 URL, Angular 将加载并显示一个模板, 并且实例化一个控制器给模板提供上下文环境.

通过调用 `$routeProvider` 服务的功能作为配置块来在你的应用程序中创建视图。就像这样的伪代码：

```
var someModule = angular.module('someModule', [... Module
dependencies ...]);

someModule.config(function($routeProvider){

    $routeProvider.

        when('url', {controller: aController, templateUrl:
'/path/to/template'});

        when(...other mappings for your app ...).

        ...

        otherwise(...what to do if nothing else matches...);

});
```

上面的代码表示当浏览器的 URL 变化为指定的 URL 时, Angular 将从 `/path/to/template` 中加载模板, 并使用 `aController` 关联这个模板的根元素(就像我们输入 `ng-controller=aController`).

在最后一行调用 `otherwise()` 用于告诉路由如果没有其他的匹配则跳到哪里.

让我们来使用一下. 我们正在构建一个 email 应用程序将轻松的战胜 Gmail, Hotmail 以及其他的. 我们暂且称它为 A-mail. 现在, 让我们从简单的开始. 我们的首屏中显示一个包括日期, 标题以及发送者的邮件信息列表. 当你点击一个信息, 它应该向将邮件的正文信息显示给你.

由于浏览器的安全限制, 如果你想自己测试这些代码, 你需要在一个 Web 服务器进行而不是使用 `file://`. 如果你安装了 Python, 你可以在你的工作目录通过执行 `python -m SimpleHTTPServer 8888` 来使用这些代码.

对于主模板, 我们会做一点不同的东西. 而不是将所有的东西都放在首屏来加载, 我们只会创建一个用于放置视图的布局模板. 我们会持续在视图中放置视图, 比如菜单. 在这种情况下, 我们只需要显示一个标题包含应用的名称. 然后使用 `ng-view` 指令来告诉 Angular 我们希望视图出现在哪里.

index.html

```
<html ng-app="Amail">
```



```
<head>

  <script src="js/angular.js"></script>

  <script src="js/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容器中，我们可以把它们编写为局部的 HTML 文档. 对于邮件列表，我们将使用 `ng-repeat` 来遍历信息列表并将它们渲染到一个表格中.

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}}</a></td>

    <td>{{message.date}}</td>

  </tr>

</table>
```

注意这里我们打算让用户通过点击主题将他导航到详细信息中。我们将 URL 数据绑定到 `message.id` 上，因此点击一个 `id=1` 的消息将使用户跳转到 `/#/view/1`。我们将通过 `url` 进行导航，也称为深度链接，在详细信息视图的控制器中，让特定的消息对应一个详情视图。

为了创建消息的详情视图，我们将创建一个显示单个 `message` 对象属性的模板。

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>

<div>{{message.message}}</div>

<a href="#/">Back to message list</a>
```

现在，将这些模板与一些控制器关联起来，我们将配置 `$routeProvider` 与 URLs 来调用控制器和模板。

controllers.js

```
//Create a module for our core AMail services

var aMailServices = angular.module('AMail', []);

//Set up our mappings between URLs, tempaltes. 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 AMailservice can find it

aMailServices.config(emailRouteConfig);

//Some take 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 some time 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
i t 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];

    }

```

我们已经创建了一个带有多个视图的应用程序的基本结构。我们通过改变 **URL** 来切换视图。这意味着用户也能够使用前进和后退按钮进行工作。用户可以在我们的应用程序中添加书签和邮件链接，即使只有一个真正的 **HTML** 页面。

对话服务器

好了，闲话少说。实际的应用程序通常与真正的服务器通讯。移动应用和新兴的 **Chrome** 桌面应用程序可能有些例外，但是对于其他的一切，你是否希望它持久保存云端或者与用户实时交互，你可能希望你的应用程序与服务器通信。

对于这一点 **Angular** 提供了一个名为 `$http` 的服务。它有一个抽象的广泛的列表使得它能够很容易与服务器通信。它支持普通的 **HTTP**, **JSONP** 以及 **CORS**。还包括防止 **JSON** 漏洞和 **XSRF** 的安全协议。它让你很容易转换请求和数据响应，甚至还实现了简单的缓存。

比方说，我们希望从服务器检索购物站点的商品而不是我们的内存中模拟。编写服务器的信息超出了本书的范围，因此让我们想象一下我们已经创建了一个服务，当你构造一个 `/product` 查询时，它返回一个 **JSON** 形式的产品列表。

给定一个响应，看起来像这样：

```
[
  {
    "id": 0,
    "title": "Paint pots",
    "description": "Pots full of paint",
    "price": 3.95
  },
  {
    "id": 1,
    "title": "Polka dots",
    "description": "Dots with that polka groove",
    "price": 12.95
  },
]
```

```

{
  "id": 2,
  "title": "Pebbles",
  "description": "Just little rocks, really",
  "price": 6.95
}

... etc ...

]

```

我们可以这样编写查询:

```

function ShoppingController($scope, $http){

  $http.get('/products').success(function(data, status, headers,
config){

    $scope.items = data;

  });

}

```

然后像这样在模板中使用它:

```

<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>

```

</body>

正如我们之前所学习到的，从长远来看我们将这项工作委托到服务器通信服务上可以跨控制器共享是明智的。我们将在第 5 章来看这个结构和全方位的讨论 `$http` 函数。

使用指令更新 DOM

指令扩展 HTML 语法，也是将行为与 DOM 转换的自定义元素和属性关联起来的方式。通过它们，你可以创建复用的 UI 组件，配置你的应用程序，做任何你能想到在模板中要做的事情。

你可以使用 **Angular** 自带的内置指令编写应用，但是你可能会希望运行你自己所编写的指令的情况。当你希望处理浏览器事件和修改 DOM 时，如果无法通过内置指令支持，你会知道是时候打破指令规则了。你所编写的代码在指令中，不是在控制器中，服务中，也不是应用程序的其他地方。

与服务一样，通过 **module** 对象的 API 调用它的 `directive()` 函数来定义指令，其中 `directiveFunction` 是一个工厂函数用于定义指令的功能(特性)。

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

appModule.directive('directiveName', directiveFunction);
```

编写指令工厂函数是很深奥的，因此在这本书中我们专门顶一个完整的一章。吊吊你的胃口，不过，我们先来看一个简单的例子。

HTML5 中有一个伟大的称为 `autofocus` 的新属性，将键盘的焦点放到一个 `input` 元素。你可以使用它让用户第一时间通过他们的键盘与元素交互而不需要点击。这是很好的，因为它可以让你声明指定你希望浏览器做什么而无需编写任何 **JavaScript**。但是如果你希望将焦点放到一些非 `input` 元素上，像链接或者任何 `div` 上会怎样？如果你希望它也能工作在不支持 HTML5 中会怎样？我们可以使用一个指令做到这一点。

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

appModule.directive('ngbkFocus', function(){

    return {
```

```

        link: function(scope, elements, attrs, controller){

            elements[0].focus();

        }

    };

});

```

这里，我们返回指令配置对象带有指定的 `link` 函数。这个 `link` 函数获取了一个封闭的作用域引用，作用域中的 DOM 元素，传递给指令的任意属性数组，以及 DOM 元素的控制器，如果它存在。这里，我们仅仅只需要获取元素并调用它的 `focus()` 方法。

然后我们可以像这样在一个例子中使用它：

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>

```

controller.js

```

function SomeController($scope) {

```



```

$scope.message = { text: 'nothing clicked yet' };

$scope.clickUnfocused = function() {

    $scope.message.text = 'unfocused button clicked';

};

$scope.clickFocused = function {

    $scope.message.text = 'focus button clicked';

}

}

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

```

当载入页面时，用户将看到标记为"I'm very focused!"按钮带有高亮焦点。敲击空格键或者回车键将导致点击并调用 `ng-click`，将设置 `div` 的文本为"focus button clicked"。在浏览器中打开这个页面，我们将看到如图 2-4 所示的东西：



图 2-4 Foucs directive

验证用户输入

Angular 带有几个适用于单页应用程序的不错的功能来自动增强 `<form>` 元素。其中之一不错的特性就是 Angular 让你在表单内的 `input` 中声明验证状态，并允许在整组元素通过验证的情况下才提交。

例如，如果我们创建一个登录表单，我们必须输入一个名称和 **email**，但是有一个可选的年龄字段，我们可以在他们提交到服务器之前验证多个用户输入。如下加载这个例子到浏览器中将显示如图 2-5 所示：

Sign Up

First name:

Last name:

Email:

Age:

图 2-5. Form validation

我们还希望确保用户在名称字段输入文本，输入正确形式的 **email** 地址，以及他可以输入一个年龄，它才是有效的。

我们可以在模板中做到这一点，使用 Angular 的 `<form>` 扩展和各个 `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，在某些字段上我们使用了 HTML5 中的 `required` 属性以及 `email` 和 `number` 类型的 `input` 元素来处理我们的验证。这对于 Angular 来说是很好的，在老式的不支持 HTML5 的浏览中，Angular 将使用形式相同职责的指令。

然后我们可以通过改变引用它的形式来添加一个控制器处理表单的提交。

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

在控制器里面，我们可以通过一个称为 `$valid` 的属性来访问这个表单的验证状态。当所有的表单 `input` 通过验证的时候，Angular 将设置它 (`$valid`) 为 `true`。我们可以使用 `$valid` 属性做一些时髦的事情，比如当表单还没有完成时禁用提交按钮。我们可以防止表单提交进入无效状态，通过给提交按钮添加一个 `ng-disabled`。

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

最后，我们可能希望控制器告诉用户她已经添加成功了。我们的最终模板看起来像这样：

```
<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></div>
```

```
</form>
```

接下来是控制器:

```
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!';

    };

}
```

小结

在前两章中, 我们看到了 **Angular** 中所有最常用的功能(特性). 对每个功能的讨论, 许多额外的细节信息都没有覆盖到. 在下一章, 我们将让你通过研究一个典型的工作流程了解更多的信息.

第三章 AngularJS 开发

现在, 我们已经探究了组成 **AngularJS** 的一些轮子. 我们已经知道用户进入我们的应用程序后如何获取数据, 如何显示文本, 以及如何做一些时髦的验证, 过滤和改变 **DOM**. 但是我们要如何把它们组织在一起呢?

在本章, 我们将讨论以下内容:

- 如何适应快速开发布局 **AngularJS** 应用程序
- 启动服务器查看应用程序行为
- 使用 **Karma** 编写和运行单元测试和场景测试
- 针对产品部署编译和压缩你的 **AngularJS** 应用程序

- 使用 Batarang 调试你的 AngularJS 应用程序
- 简化开发流程(从创建文件到运行应用程序和测试)
- 使用依赖管理库 RequireJS 整合 AngularJS 项目

本章旨在提供一个 20000 英尺的视图以告诉你如何可行的布局你的 AngularJS 应用程序. 我们不会进入实际应用程序本身. 在第 4 章, 深入一个使用和展示了各种各样 AngularJS 特性的示例一用程序.

目录

- [项目组织](#)
- [工具](#)
- [运行你的应用程序](#)
 - [使用 Yeoman](#)
 - [不使用 Yeoman](#)
- [测试 AngularJS](#)
- [单元测试](#)
- [端到端/集成测试](#)
- [编译](#)
- [其他优秀工具](#)
 - [调试](#)
 - [Batarang](#)
- [Yeoman: 优化你的工作流程](#)
 - [安装 Yeoman](#)
 - [启动一个新的 AngularJS 项目](#)
 - [运行服务器](#)
 - [添加新的路由, 视图和控制器](#)
 - [测试的故事](#)
 - [构建项目](#)
- [使用 RequireJS 整合 AngularJS](#)

项目组织

推荐使用 Yeoman 构建你的项目, 将会为你的 AngularJS 应用程序创建所有必要的辅助程序文件.

Yeoman 是一个包含多个框架和客户端库的强大的工具。它通过自动化一些日常任务的引导的方式提供了一个快速的开发环境和增强你的应用程序。本章我们会使用一个完整的小节介绍如何安装和使用 **Yeoman**，但是在那之前，我们先来简单的介绍以下使用 **Yeoman** 命令替代那些手动执行的操作。

我们还详细介绍了涉及到让你决定不使用 **Yeoman** 的情况，因为在 **Windows** 平台的计算机上 **Yeoman** 确实存在一些问题，并且设置它还稍微有一点挑战性。

对于那些不使用 **Yeoman** 的情况，我们将会看看一个示例应用程序结构(可以在 **Github** 示例仓库的 `chapter3/sample-app` 目录中找到 [-链接](#))，下面是推荐的结构，也是使用 **Yeoman** 生成的结构。应用程序的文件可以分为以下类别：

JS 源文件

看看 `app/scripts` 目录。这里是所有 **JS** 源文件所在目录。一个主文件来给你的应用程序设置 **Angular** 模块和路由。

此外，这里还有一个单独的文件夹--`app/scripts/controller`--这里面是各个控制器。控制器提供行为和发布数据到作用域中，然后显示在视图中。通常，它们与视图都是一一对应的。

指令，过滤器和服务也可以在 `app/scripts` 下面找到，不管是否优雅和复杂，作为一个完整的文件(`directives.js`, `filters.js`, `services.js`)或者单个的都行。

Angular HTML 模板文件

现在，使用 **Yeoman** 创建的每一个 **AngularJS** 局部模板都可以在 `app/views` 目录中找到。这是映射到大多数 `app/scripts/controller` 目录中。

还有另外一个重要的 **Angular** 模板文件，就是主要的 `app/index.html`。这用户获取 **AngularJS** 源文件，也是你为应用程序创建的任意源文件。

如果你最终会创建一个新的 **JS** 文件，要确保把它添加到 `index.html` 中，同时还要更新的主模块和路由(**Yeoman** 也会为你做这些)。

JS 库依赖

Yeoman 在 `app/scripts/vendor` 中为你提供了所有的 **JS** 源文件依赖。想在应用程序中使用 [Underscore](#) 和 [SocketIO](#)？没问题--将依赖添加到 `vendor` 目录中(还有你的 `index.html`)，并开始在你的应用程序中引用它。

静态资源

最终你创建了一个 **HTML** 应用程序，它还会考虑到你的应用程序还有作为需要的 **CSS** 和图像依赖。`app/styles` 和 `app/img` 目录就是出于这个目的而产生的。你只需要

添加你需要的东西到目录中并在你的应用程序中引用它们(当然, 要使用正确的绝对路径).

默认情况下 Yeoman 不会创建 `app/img` 路径.

单元测试

测试是非常重要的, 并且当它涉及到 **AngularJS** 时是毫不费力的. 在测试方面 `test/spec` 目录应该映射到你的 `app/scripts` 目录. 每个文件都应该有一个包含它的单元测试的 `spec` 文件映射(镜像). 种子文件会给每个控制器文件创建一个存根文件, 在 `test/spec/controllers` 目录下, 与原来的控制器具有相同的名称. 它们都是 **Jasmine** 风格的规范, 描述了每个控制器预期行为的规范.

集成测试

AngularJS 自带了端对端的测试支持以正确的方式内置到库里面. 你所有的 **Jasmine** 规范形式的 **E2E**(端对端)测试, 都保存在 `tests/e2e` 目录下.

默认情况下 Yeoman 不会创建 `test/` 目录.

虽然 **E2E** 测试可能看起来像 **Jasmine**, 但实际上不是的. 它们的函数是异步执行的, 来未来, 可以通过 **Angular** 场景运行器(**Angular Scenario Runner**)运行. 因此不要指望能够做正常情况下 **Jasmine** 测试所做的事情(像使用 `console.log` 重复输出一个值的情况).

还生成了一个简单的 **HTML** 文件, 可以在浏览器中打开它来手动的运行测试. 然而 Yeoman 不会生成存根文件, 但是它们遵循相似风格的单元测试.

配置文件

这里需要两个配置文件. 第一个是 `karma.conf.js`, 这是 Yeoman 为你生成的用于运行单元测试的. 第二个, 是 Yeoman 不会生成的 `karma.e2e.conf.js`. 这用于运行场景测试. 在本场尾部的继承 **RequireJS** 一节中有一个简单的文件. 这用于配置依赖关系的详情, 同时这个文件用在你使用 **karma** 运行单元测试的时候.

你可能会问: 如何运行我的应用程序? 什么是单元测试? 甚至我该如何编写你们所讨论的这种各样的零件?

别担心, 年轻的蚱蜢, 所有的这些在适当的时间都会解释. 在这一章里面, 我们将处理设置项目和开发环境的问题, 因此一旦我们掺入一些惊人的代码, 那些问题都可以快速的带过. 你所编写的代码以及如何将它们与你最终的惊人的应用程序联系在一起的问题, 我们将在接下来的几章中讨论.

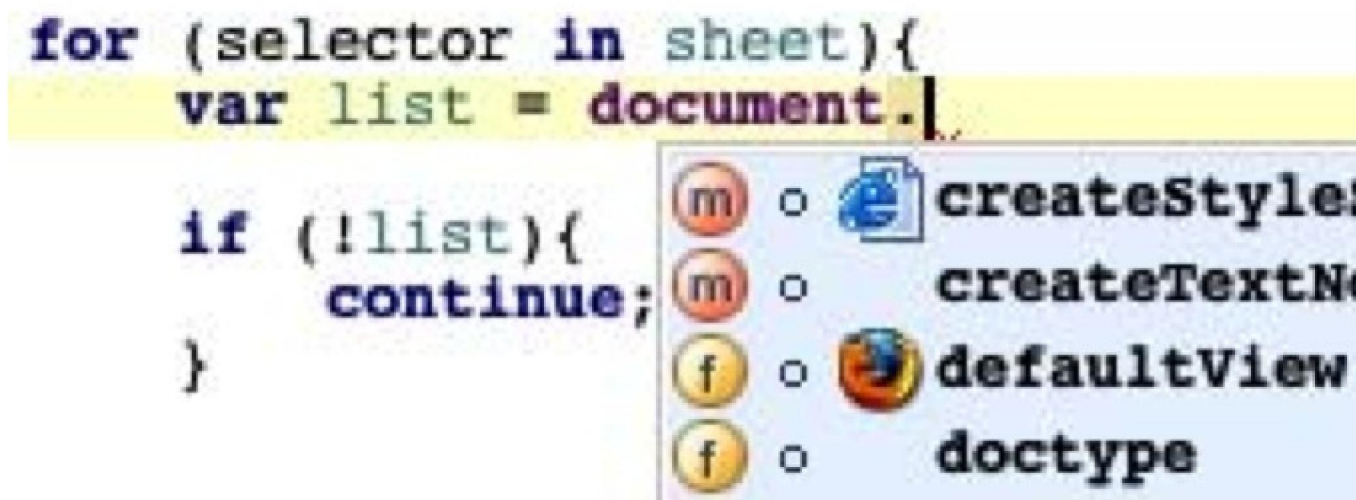
工具

AngularJS 只是你开发实际网页的工具箱的一部分。在这一节，我们将一起开看看一些你用以确保高效和快速开发的不同的工具，从 IDEs 到测试运行器到调试工具。

IDEs

让我们从你如何编写源代码开始。有大量的 JavaScript 编辑器可以选择，有免费的也有付费的。长时间以来的事实证明 Emacs 和 Vi 是开发 JS 的最好选择。现在，各种 IDEs 都自带了语法高亮，自动完成以及其他功能，它给你一个选择的余地，这可能是值得的。那么，应该使用那一个呢？

如果你不介意付几十块钱(尽管它有一个 30 天的免费试用期)，[WebStorm](#) 是个不错的选择，当今时代，WebStorm 由 JetBrains 提供了最广泛的 Web 开发平台。它所具有的特性，在之前只有强类型语言才有，包括代码自动完成(如图 3-1 所示，指定浏览器版本)，代码导航，语法和多无高亮，同时支持多个库和框架的启动即可使用。此外，它还漂亮的集成了在 IDE 中正确的调试 JavaScript 的功能，而且这些都是基于 Chrome 执行的。



最大的你应该考虑使用 WebStorm 开发 AngularJS 原因是它是唯一继承 AngularJS 插件的 IDEs。这个插件支持在你的 HTML 模板中正确的自动完成 AngularJS 的 HTML 标签。这些都是常用的代码片段，否则你每次都要输入拼凑的代码片段。因此不应该像下面这样输入：

```
directive('$directiveName$', function factory($injectables$){  
  
  var directiveDefinitionObject = {
```



```
$directiveAttr$;

compile: function compile(tElement, tAttrs, transclude){

    $END$;

    return function(scope, elements, attrs){

        //...

    }

}

};

return directiveDefinitionObject;

});
```

在 WebStorm 中, 你可以只输入以下内容:

```
ngdc
```

然后按 **Tab** 键获取同样的代码. 这只是大多数代码自动完成插件提供的功能之一.

运行你的应用程序

现在让我们讨论如何运行所有你所做的事情 - 查看应用程序活动起来, 在浏览器中. 真实的感受以下应用程序是如何工作, 我们需要一个服务器来服务于我们的 **HTML** 和 **JavaScript** 代码. 我将探讨两种方式, 一种非常简单的方式是使用 **Yeoman** 运行应用程序, 另外一种不是很容易的不用 **Yeoman** 的方法, 但是同样很好.

使用 Yeoman

Yeoman 让你很简单的使用一个 **Web** 服务器服务你所有的静态资源和相关的 **JavaScript** 文件. 只需要运行以下命令:

```
yeoman server
```

它将启动一个服务器同时也在你的浏览器中打开 **AngularJS** 应用程序的主页. 每当你改变你的源代码时, 它甚至会刷新(自动刷新)浏览器. 很酷不是吗?

不使用 Yeoman

如果不使用 Yeoman, 你可能需要配置一个服务器来服务你所有主目录中的文件. 如果你不知道一个简单的方法做到这一点, 或者不想浪费时间创建你自己的 Web 服务器, 你可以在 Node.js 中使用 ExpressJS 快速的编写一个简单的 Web 服务器 (只要简单的使用 `npm install -g express` 来获取它). 它可能看起来像下面这样:

```
//available at chapter3/sample-app/web-server.js

var express = require('express'),

    app = express(),

    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");
```

一旦你有了这个文件, 你就可以使用 Node 运行这个文件, 通过使用下面的命令:

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

同时它将在 8080 端口启动服务器(或者你自己选择端口).

可选的, 在应用程序文件夹中使用 Python 你应该运行:

```
python -m SimpleHTTPServer
```

无论你是否决定继续，一旦你配置好服务器并运行起来，都将被导航到下面的 URL:

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

然后你就可以在浏览器中查看你刚刚创建的应用程序。注意，你需要手动的刷新浏览器来查看改变，不同于使用 Yeoman.

测试 AngularJS

之前已经说过(甚至在本章的前面)，我们再重新说一次：测试是必不可少的，AngularJS 使编写合理的单元测试和集成测试变得很简单。虽然 AngularJS 使用多个测试运行器运行的很好，但我们坚信 [Karma](#) 胜过大多数你所需要的提供强大，坚实和及其快速的运行器。

Karma

Karma 存在的主要的原因是它让你的测试驱动开发(TDD)流程变得简单，快速和有趣。它使用 NodeJS 和 SocketIO(你不需要知道它们是什么，只需要假设它们是很棒很酷的库)，并允许在多个浏览器中及其快速的运行你的代码和测试。在 <https://github.com/vojtajina/karma/> 中可以找到更多信息。

TDD 简介

测试驱动开发或者 TDD，是一个通过确保在开发生命周期内首先编写测试的敏捷方法，这是在代码实现之前进行的，这就是测试驱动的开发(不只是作为一种验证工具)。

TDD 的原则很简单：

- 代码只需要在一个所需要的代码测试失败时编写。
- 编写最少的代码以确保测试通过。
- 在每一步删除重复代码。
- 一旦所有的测试通过，接下来就是给下一个所需要的功能添加失败测试。

以下是确保这些原则的简单规则：

- 有组织的开发你的代码，每一行代码都要有目的的编写。
- 你的代码仍然是高度模块化，紧密结合和可复用的(你需要能够测试它)
- 提供一系列的全面的测试列表，以防后期的破坏和 Bugs。

- 测试也应该充当规范和文档，以适应后期需要和变化。

在 AngularJS 中我们发现这是真的，同时在整个 AngularJS 代码库中我们都是使用 TDD 来开发。对于像 JavaScript 这样的无需编译的动态语言，我们坚信良好的单元测试可以减轻未来的头痛。

那么，我们如何获取迷人的 Karma 呢？好吧，首先确保在你的机器上安装了 NodeJS。它自带了 NPM(Node 包管理器)，这使得它易于管理和安装成千上万的 NodeJS 可用的库。

一旦你安装了 NodeJS 和 NPM，安装 Karma 只需要简单的运行下面的命令：

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

到这里。你只要简单的三部来开始使用 Karma(我刚才说了，请不要了解它现实上是怎么使用的)。

获取配置文件：

如果你是用 Yeoman 创建应用程序骨架，那么你就已经有一个现成的 Karma 配置文件等你来使用。如果不是，那么继续，并且在你的应用程序目录的根文件夹中执行下面的命令：

```
karma init
```

在你的终端控制器中执行(定位到目录文件夹,然后执行命令)，他会生成一个虚拟的配置文件(karma.conf.js)，你可以根据你的喜好来编辑它，它默认带有一些很好的标准。你可以使用它。

启动 Karma 服务器

只需要运行下面的命令：

```
karma start [optionalPathToConfigFile]
```

这将会在 9876 端口启动 Karma 服务器(这是默认情况，你可以通过编辑在上一步提到的 karma.conf.js 文件来改变)。虽然 Karma 应该打开一个浏览器并自动捕获它，它将在控制台中打印所有其他浏览器中捕获所需要的指令。如果你懒得这样做，只需要在其他浏览器或者设备中浏览 <http://localhost:9876>，并且你最好在多个浏览器中运行测试。

虽然 Karma 可以自动捕获常用的浏览器，在启动时。(FireFox, Chrome, IE, Opera, 甚至是 PhantomJS)，但它不仅限于只是这些浏览器。任何可以浏览一个 URL 的设备都可能可以作为 Karma 运行器。因此如果你打开你的 iPhone 或者 Android 设备上浏览器并浏览 `http://machinename:9876` (只要是可访问的)，你都可能在移动设备上运行同样的测试。

运行测试

执行下面的命令：

```
karma run
```

就是这样。运行这个命令之后，你应该获得正好打印在控制台的结果。很简单，不是吗？

单元测试

AngularJS 是的编写单元测试变得更简单，默认情况下支持编写 [Jasmine](#) 风格的测试(就是 Karma)。Jasmine 就是我们所说的行为驱动开发框架，它允许你编写规范来说明你的代码行为应该如何表现。一个 Jasmine 测试范例看起来可能是这样子的。

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

正如你可以看到，它本身就是非常容易阅读的格式，大部分的代码都可以用简单的英文来阅读理解。它还提供了一个非常多样化和强大的匹配集合(就像 `expect` 从句)，

当然它还有 [xUnit](#) 常用的 `setUp` 和 `tearDowns` (函数在每个独立的测试用例之前或者之后执行)。

AngularJS 提供了一些优雅的原型，和测试函数一样，它允许你有在单元测试中创建服务，控制器和过滤器的权利，以及模拟 `HttpRequests` 输出等等。我们将在第五章讨论这个。

Karma 可以使它很容易的集成到你的开发流程中，以及在你编写的代码中获取即时的反馈。

集成到 IDEs 中

Karma 并没有所有最新版和最好的(greatest)IDEs 使用的插件(已经实现的还没有)，但实际上你并不需要。所有你所需要做的就是你的 IDEs 中添加一个执行" karma start"和" karma run"的快捷命令。这通常可以通过添加一个简单的脚本来执行，或者实际的 shell 命令，依赖于你所选择的编辑器。当然，每次完成运行你都应该看到结果。

在每一个变化上运行测试

这是许多 TDD 开发者的理想国：能够运行在它们所有的测试中，每次它们按下保存，在几毫秒之内迅速的得到返回结果。使用 AngularJS 和 Karma 可以很容易做到这一点。事实证明，Karma 配置文件(记住就是前面的 `karma.conf.js`)有一个看似无害的名为 `autoWatch` 的标志。设置它为 `true` 来告诉 Karma 每次运行你的测试文件(这就是你的源代码和测试代码)都监控它的变化。然后在你的 IDE 中执行" karma start"，猜猜会怎样？Karma 运行结果将可供你的 IDE 使用。你甚至不需要切换控制台或者终端来了解发生了什么。

端到端/集成测试

随着应用程序的发展(或者有这个趋势，事实上很快，之前你甚至已经意识到这一点)，测试它们是否如预期那样工作而不需要手动的裁剪任何功能。毕竟，没一添加新的特性，你不仅要验证新特性的工作，还要验证老特性是否仍然更够正常工作，并且没有 bug 和功能也没有退化。如果你开始添加多个浏览器，你可以很容看出，其实这些可以变成一个组合。

AngularJS 视图通过提供一个 Scenario Runner 来模拟用户与应用程序交互来缓解这种现象。

Scenario Runner 允许你按照类 Jasmine 的语法来描述应用程序。正如之前的单元测试，我们将会有一些的 `describes` (针对这个特性)，同时它还是独立(描述每个单独

功能的特性). 和往常一样, 你可以有一些共同的行为, 对于执行每个规范之前和之后.(我们称之为测试).

来看一个应用程序示例, 他返回过滤器结果列表, 看起来可能像下面这样:

```
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. 不过, 无论使用那种方式运行它们, 你都必须有一个 **Web** 服务器来启动你的应用程序服务(请参见上一节来查看如何做到这一点). 一旦做到这一点, 可以使用下列方法之一:

1. **自动化:** Karma 现在支持运行 **Angular** 情景测试. 创建一个 **Karma** 配置文件然后进行以下改变:
 - a. 添加一个 **ANGULAR_SCENARIO & ANGULAR_SCENARIO_ADAPTER** 到配置的文件部分.
 - b. 添加一个代理服务器将请求定位到正确的测试文件所在目录, 例如:

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

- c. 添加一个 **Karma root**(根目录/基础路径)以确保 **Karma** 的源文件不会干扰你的测试, 像这样:

```
urlRoot = '/karma/';
```

然后只需要记得通过浏览 `http://localhost:9876/_karma_` 来捕捉 Karma 服务器, 你应该使用 Karma 自由的运行你的测试.

2. **手动:** 手动的方法允许你从你的 Web 服务器上打开一个简单的页面运行(和查看)所有的测试.

a. 创建一个简单的 `runner.html` 文件, 这来源于 Angular 库的 `angular-scenario.js` 文件.

b. 所有的 JS 源文件都遵循你所编写的你的场景组件部分的规范.

c. 启动你的 Web 服务器, 浏览 `runner.html` 文件.

为什么你应该使用 Angular 场景运行器, 或者说是外部的第三方库活端对端的测试运行器? 使用场景运行器有令人惊讶的好处, 包括:

AngularJS 意识

Angular 场景运行器, 顾名思义, 它是由 Angular 创建的. 因此, 他就是 Angular aware, 它直到并理解各种各样的 Angular 元素, 类似绑定. 需要输入一些文本? 检查绑定的值? 验证中继器(repeater)状态? 所有的这些都可以通过场景运行器轻松的完成.

无需随机等待

Angular 意识也意味着 Angular 直到所有的 XHR 何时向服务器放出, 从而可以避免页面加载所等待的间隔时间. 场景运行器直到何时加载一个页面, 从而比 Selenium 测试更具确定性, 例如, 超时等待页面加载时任务可能失败.

调试功能

探究 JavaScript, 如果你查看你的代码不是很好; 当你希望暂停和恢复测试时, 所有的这些都运行场景测试吗? 然而所有的这一切通过 Angular 场景运行器都是可行的, 等等.

编译

在 JavaScript 世界里, 编译通常意味着压缩代码, 虽然一些实际的编译可能使用的时 Google 的 Closure 库. 但是为什么你会希望将所有漂亮的, 写的很好, 很容易理解代码变得不可读呢?

原因之一是我们的目标是应用程序更快的响应用户。这是为什么客户端应用程序几年前不想现在腾飞得这么快的主要原因。能够越早获取应用程序并运行，响应得也越早。

这种快速响应是压缩 JS 代码的动机。代码越小，越能有效的减小负载，同时能够更快的将相关文件发送给用户。这在移动应用程序中显得尤为重要，因为其规模成为瓶颈。

这里有集中方法可以压缩你给应用程序所编写的 **AngularJS** 代码，每种方法都具有不同的效果。

基本的和简单的优化

这包括压缩所有在你的代码中使用的变量，但是不会压缩属性。这就是所谓的传递给 **Closure Compiler** 的简单优化。

者不会给你带来多大的文件大小的节省，但是你仍然可以获得一个可观的最小开销。

这项工作的原因是编译器(**Closure** 或者 **UglifyJS**)并不会重命名你从模板中引用的属性。因此，你的模板会继续工作，仅仅重命名局部变量和参数。

对于 **Google Closure**，只需简单的调用下面的命令：

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

高级优化

高级优化有一点棘手，它会试图重名几乎任何东西和每个函数。得到这个级别的优化工作，你将需要通过显示的告诉它哪些函数，变量和属性需要重命名(通过使用一个 **externs** file)。者通常是通过模板访问函数和属性。

编译将会使用这个 **externs** 文件，然后重命名所有的东西。如果处理好，这可能会导致的你的 **JavaScript** 文件大幅度的减小，但是它的确需要相当大的工作像，包括每次改变代码都要更急 **externs** 文件。

要记住一件事：当你想要压缩代码时，你要使用依赖注入的形式(在控制器上指定 **\$inject** 属性)。

下面的代码不会工作

```
function MyController($scope, $resource){
```

```
//Stuff here

}
```

你需要像下面这样做:

```
function MyController($scope, $resource){

    //Some Stuff here

}

MyController.$inject = ['$scope', '$resource'];
```

或者是使用模块, 像这样:

```
myAppModule('MyController', ['$scope', '$resource', function($scope,
$resource){

    //Some stuff here

}]);
```

一旦所有的变量都混淆或者压缩只有, 这是使用 **Angular** 找出那些你最初使用的服务和变量的方式.

每次都是数组的方式注入是比较好的处理方式, 以避免开始编译代码时的错误. 挠头并视图找出为什么提供的 **\$e** 变量丢失了(一些任务的混淆版本压缩了它)是不值得的.

其他优秀工具

在本节, 我们将会看一些其他有助于简化你的开发流程和提高效率的工具. 这包括使用 **Batarang** 调试真实的代码和使用 **Yeoman** 开发.

调试

当你使用 **JavaScript** 工作时, 在浏览器中调试你的代码会成为一个习惯. 你越早接受这个事实, 对你越有好处. 值得庆幸的是, 当过去没有 **Firebug** 时, 这件事已经走过了漫长的路. 现在, 不管选择什么浏览器, 一般来说你都可以介入代码来分析错误和判断应用程序的状态. 只需要去了解 **Chrome** 和 **Internet Explorer** 的开发者工具, 能同时在 **FireFox** 和 **Chrome** 中工作的 **Firebug**.

这里有一些帮助你进一步调试应用程序的技巧提示:

- 永远记住, 当你希望调试引用程序时, 记得切换到非压缩版本的代码和依赖中进行. 你不仅会获得更好(可读)变量名, 也会获得代码行号和实际有用的信息以及调试功能.
- 尽量保持你的源代码为独立的 **JS** 文件, 而不是内联在 **HTML** 中.
- 断点调试是很有用的! 它们允许你检查你的应用程序状态, 模型, 以及给定的时间点上的所有信息.
- "暂停所有异常"是内置在当今大多数开发者工具中的一个非常有用的选项. 当发现一个异常是调试器会终止继续运行并高亮导致异常的代码行.

Batarang

当然, 我们有 Batarang. Batarang 是一个添加 AngularJS 知识的 Chrome 扩展, 它是嵌套在 Google Chrome 中内置开发者工具. 一旦安装(你可以从 <http://bit.ly/batarangjs> 中获取), 它就会在 Chrome 的开发者工具面板中添加一个 AngularJS 选项.

你有没有想过你的 AngularJS 应用程序的当前状态是什么? 当前(视图)包含的每个模型, 作用域和变量是什么? 你的应用程序性能如何? 如果你还没有想过, 相信我, 你会的. 当你需要这么做时, Batarang 会为你服务.

这里有 Batarang 的四个主要的有用的附加功能:

模型选项

Batarang 允许你从根源向下深入探究 `scope`. 然后你可以看到这些 `scopes` 是如何嵌套以及模型是如何与之关联的.(如图 3-2 所示). 你甚至可以实时的改变它们并在应用程序中查看变化的反映. 很酷, 不是吗?

Developer Tools - http://docs.a

Elements Resources Network Sources Timeline Profile

Model Performance Dependencies Op

Model Tree

Scope (002) | toggle

Scope (003) | toggle

- URL {"module":"guide/module","dire
- afterPartialLoaded
- bestMatch {"page":null,"rank":0}
- breadcrumb [{"name":"API Referenc
- currentPage {"section":"api","id":}
- enableOffline
- focused
- futurePartialTitle null
- loading


☐ >≡ 🔍


Figure 3-2 Model tree in Batarang


性能选项


性能选项必须单独启用，它会注入一些特殊的 **JavaScript** 代码到你的应用程序中。一旦你启用它，你就可以看到不同的作用域和模型，并且可以在每个作用域执行所有的性能监控表达式(如图 3-3 所示)。随着你使用应用程序，性能也会得到更新，因此它可以很好的实时工作。


Developer Tools - http://builtwith


Elements

Resources

Network

Sources

Timeline

Profiles

Watch Expressions


```
function (a){var e,f,h=a.$eval(i),m=f
db,A,y,v,s,z=c;if(K(h))v=h||[];else{v
h)h.hasOwnProperty(A)&&A.charAt(0)!=
```


`{{tag}}` | 1.82% | 1.59ms

`n + 1` | 0.985% | 0.858ms

`tags.length == 0` | 0.838% | 0.730ms

`{disabled: currentPage == 0}` | 0.803% |








Figure 3-3. Performance tab in Batarang

服务依赖

对于一个简单的应用程序，不会超过 1-2 个控制器和服务依赖。但是事实上，全面的应用程序，如果没有工具支持，服务依赖管理会成为噩梦。幸好这里有 **Batarang** 可以给你提供服务，填补这个洞，因为它给你提供了一个干净，简单的方法来查看可视化的服务依赖图表(如图 3-4 所示)。



Elements



Resources



Network



Sources



Timeline



Profiler

Model

Performance

Dependencies

Open

Service Dependencies

bwaProjectDirective





Figure 3-4. Charting dependencies in Batarang


元素属性和控制台访问


当你通过 HTML 模板来探究一个 AngularJS 应用程序时，元素选项的属性窗格中现在有一个额外的 AngularJS 属性部分。这允许你检查模型所连接的给定元素的 `scope`。它也会公开这个元素的 `scope` 到控制台中，因此你可以在控制台中通过 `$scope` 变量来访问它。如图 3-5 所示：


Developer Tools - http://builtwith.an


Elements

Resources

Network

Sources

Timeline

Profiler

```
<!DOCTYPE html>
<!--[if lt IE 7]> <html
lt-ie8 lt-ie7" lang="en
<!--[if IE 7]> <html
lt-ie8" lang="en"> <![e
<!--[if IE 8]> <html
lang="en"> <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js" lan
  <!--<![endif]-->
  <head>...</head>
  <body ng-controller="
  twttr-rendered="true" c
    ""
    <!--[if lte IE 8]>
      <script>
        document.creat
        project');
      </script>
    <![endif]-->
    <script>...</script>
    <script>...</script>
    <!-- Prompt IE 6 use
    Frame. Remove this i
    chromium.org/
    tos/chrome-frame-get
    <!--[if lt IE 7]><p
    class=chrome-frame>Yo
    <em>ancient!</em> <a
    href="http://browseh
```

Event Listeners

AngularJS Properties

```
$id: "003"
$parent: e
__private__: Object
activeTags: Array
addTag: function (
  currentPage: 0
featured: Object
filteredProjects:
group: function ()
groupToPages: func
groupedProjects: A
lightbox: function
nextPage: function
pagedProjects: Arr
  0: Array[5]
  1: Array[5]
  2: Array[5]
  3: Array[3]
  length: 4
  __proto__: Array
prevPage: function
projects: Array[35]
range: function (s
removeTag: function
search: function (
setPage: function
  sortPrep: "none"
sortables: Array[4]
tags: Array[56]
```

Figure 3-5. AngularJS properties within Batarang

Yeoman: 优化你的工作流程

相当多的工具如雨后春笋般涌现，以帮助你在开发应用程序时优化工作流程。我们在前面章节所谈及的 **Yeoman** 就是这样一种工具，它拥有令人印象深刻的功能集，包括：

- 轻快的脚手架
- 内置预览服务器
- 集成包管理
- 一流的构建过程
- 使用 **PhantomJS** 进行单元测试

它还很好的集成和扩展了 **AngularJS**，这也是我们为什么强烈推荐任何 **AngularJS** 项目使用它的主要原因之一。让我们通过上面的集中方式使用 **Yeoman** 时你的生活更轻松。

安装 Yeoman

安装 **Yeoman** 是一个相当复杂的过程，但也可以通过一些脚本来帮助你安装。

在 **Mac/Linux** 机器上，运行下面的命令：

```
curl -L get.yeoman.io | bash
```

然后只需按照打印的只是来获取 **Yeoman**。

对于 **Windows** 机器，或者运行它是遇到任何问题，到 <https://github.com/yeoman/yeoman/wiki/Manual-Install> 并按照说明来安装会让你畅通无阻。

启动一个新的 **AngularJS** 项目

正如前面所提到的，甚至一个简单的项目都有许多技术需要处理，从模板到基础控制，再到库依赖，一切事情都需要结构化。你可以手动的做这些工作，或者也可以使用 **Yeoman** 处理它。

只需为你的项目中简单的创建一个目录(**Yeoman** 将把目录名称当作项目名称), 然后运行下面的命令:

```
yeoman init angular
```

这将创建一个本章项目优化部分所详细描述的一个完整的结构, 包括渲染路由的框架, 单元测试等等.

运行服务器

如果你不适用 **Yeoman**, 那么你不得不创建一个 **HTTP** 服务器来服务你的前端代码. 但是如果使用 **Yeoman**, 那么你将获得一个内置的预先配置好的服务器并且还能获得一些额外的好处. 你可以使用下面的命令启动服务器:

```
yeoman server
```

这不单单只启动一个 **Web** 服务器来服务于你的代码, 它还会自动打开你的 **Web** 浏览器并在你改变你的应用程序时刷新你的浏览器.

添加新的路由, 视图和控制器

添加一个新的 **Angular** 路由涉及多个步骤, 其中包括:

- 在 `index.html` 中启用新的控制器 **JS** 文件
- 添加正确的路由到 **AngularJS** 模块中
- 创建 **HTML** 模板
- 添加单元测试

所有的这些在 **Yeoman** 中使用下面的命令可以在一步完成:

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

因此, 如果你运行 `yeoman iniy angular route home` 结束之后它将执行以下操作:

- 在 `app/scripts/controllers` 目录中创建一个 `home.js` 控制器骨架
- 在 `test/specs/controllers` 目录中创建一个 `home.js` 测试规范
- 将 `home.html` 模板添加到 `app/views` 目录中
- 链接主引用模块中的 `home` 路由(在 `app/scripts/app.js`` 文件中)

所有的这些都只需要一条单独的命令!

测试的故事

我们已经看过使用 **Karma** 如何轻松的启动和运行测试. 最终, 运行所有的单元测试只需要两条命令.

Yeoman 使它变得更容易(如果你相信它). 每当你使用 **Yeoman** 生成一个文件, 它都会给你创建一个填充它的测试存根. 一旦你安装了 **Karma**, 使用 **Yeoman** 运行测试只需执行下面的命令即可:

```
yeoman test
```

构建项目

构建一个完备的应用程序可能是痛苦的, 或者至少涉及到需要步骤. **Yeoman** 通过允许你像下面这样做减轻了不少痛苦:

- 连接(合并)所有 **JS** 脚本到一个文件中
- 版本化文件
- 优化图片
- 生成应用程序缓存清单

所有的这些好处都来自于一条命令:

```
yeoman build
```

Yeoman 不支持压缩文件, 但是根据来发者提供的信息, 它很快会到来.

使用 **RequireJS** 整合 **AngularJS**

如果你提单做好更多的事情, 正好会让你的开发环境更简单. 后期修改你的开发环境, 会需要修改更多的文件. 依赖管理和创建包部署是任何规模的项目所忧虑的.

使用 **JavaScript** 设置你的开发环境是相当困难的, 因为它涉及 **Ant** 构建维护, 连接你的文件来构建脚本, 压缩它们等等. 值得庆幸的是, 在不久之前已经出现了像 **RequireJS** 这样的工具, 它允许你定义和管理你的 **JS** 依赖关系, 以及将他们挂到一个简单的构建过程中. 随着这些异步加载管理的工具诞生, 能够确保所有的依赖

文件在执行之前加载好, 重点工作可以放在实际的功能开发, 在此之前从未如此简单过.

值得庆幸的是, **AngularJS** 能够很好的发挥 [RequireJS](#), 因此你可以做到两全其美. 这里有一个目标示例, 我们找到了在一个系统中能够工作的很好而且易于遵循的方式来提供一个样本设置.

让我们一起来看看这个项目的组织(类似前面描述的骨架, 稍微有一点变化):

1. **app**: 这个目录是所有显示给用户的应用程序代码宿主目录. 包括 HTML, JS, CSS, 图片和依赖的库.

a. **/style**: 包含所有的 CSS/Less 文件

b. **/images**: 包含项目的所有图片文件

c. **/script**: 主 AngularJS 代码库. 这个目录也包括我们的引导程序代码, 主要的 RequireJS 集成

i. **/**controllers****: 这里是 AngularJS 控制器

ii. **/**directives****: 这里是 AngularJS 指令

iii. **/**filters****: 这里是 AngularJS 过滤器

iv. **/**services****: 这里是 AngularJS 服务

d. **/vendor**: 我们所依赖的库(Bootstrap, RequireJS, jQuery)

e. **/views**: 视图的 HTML 模板部分和项目所使用的组件

2. **config**: 包含单元测试和场景测试的 Karma 配置

3. **test**: 包含应用程序的单元测试和场景测试(整合的)

a. **/spec**: 包含应用程序的 JS 目录中的单元测试和镜像结构

b. **/e2e**: 包含端到端的场景规范

我们所需做的第一件事情是在 `main.js` 文件(在 **app** 目录)中引入 **RequireJS**, 然后使用它加载所有的其他依赖项. 这里有一个例子, 我们的 JS 项目除了自己的代码还会依赖于 jQuery 和 Twitter 的 Bootstrap.

//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

        }

    ]);

}

);

```

然后我们定义一个 `app.js` 文件. 这个文件定义我们的 **AngularJS** 应用程序, 同时告诉它, 它依赖于我们所定义的所有控制器, 服务, 过滤器和指令. 我们所看到的 **RequireJS** 依赖列表中所提到的只是一点点.

你可以认为 **RequireJS** 依赖列表就是一个 **JavaScript** 的 `import` 语句块. 也就是说, 代码块内的函数直到所有的依赖列表都满足或者加载完成它都不会执行.

另外请注意, 我们不会单独 告诉 **RequireJS**, 载入的执行, 服务或者过滤器是什么, 因为这些并不属于项目的结构. 每个控制器, 服务, 过滤器和指令都是一个模块, 因此只定义这些为我们的依赖就可以了.


```
// 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']);

});
```

我们还有一个 `bootstrap.js` 文件，它要等到 DOM 准备就绪(这里使用的 RequireJS 的插件 `domReady`)，然后告诉 AngularJS 继续执行，这是很好的。

```
// 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']);

    });

});
```

这里将引导从应用程序中分割出来，还有一个有事，即我们可以使用一个伪造的文件潜在的取代我们的 `mainApp` 或者出于测试的目的使用一个 `mockApp`。例如 如果你所依赖的服务器不可开，你只需要创建一个 `fakeApp` 使用伪造的数据来替换所有的 `$http` 请求，以保持你的开发秩序。这样的话，你就可以只悄悄的使用一个 `fakeBootstrap` 和一个 `fakeApp` 到你的应用程序中。

现在，你的 `main.html` 主模板(app 目录中)可能看起来像下面这样：

```
<!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，我们来看看 `js/controllers/controllers.js` 文件，这看起来几乎与 `js/directives/directives.js`, `js/filters/filters.js` 和 `js/services/services.js` 一模一样：

```

define(['angular'], function(angular){

    'use strict';

    return angular.module('controllers', []);

});

```

因为我们使用了 **RequireJS** 依赖的结构，所有的这些都会保证只在 **Angular** 依赖满足并加载完成的情况下才会运行。

每个文件都定义为一个 **Angular** 模块，然后将单个的控制器，指令，过滤器和服务添加到定义中来使用。

让我们来看看一个指定定义(比如第二章的 `focus` 指令)：

```

//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();

            }

        }

    });

});

```

指令自什么很琐碎的, 让我们仔细看看发生了什么. 围绕着文件的 **RequireJS shim** 告诉我们 `ngbkFocus.js` 依赖于在模块中声明的 `directices/directives.js` 文件. 然后它使用注入指令模块将自身指令声明添加进来. 你可以选择多个指令或者一个单一的对应的文件. 这完全由你决定.

一个重点的注意事项: 如果你有一个控制器进入到服务中(也就是说你的 `RootController` 依赖于你的 `UserSevice`, 并且获取 `UserService` 注入), 那么你必须确保将你定义的文件加入 **RequireJS** 依赖中, 就像这样:

```

define(['controllers/controllers', 'services/userService'],
function(controllers){

    controllers.controller('RootController', ['$scope', 'UserService',
function($scope, UserService){

        //Do what's needed

    }]);

});

```

这基本上是你整个源文件目录的结构设置.

但是你会问, 这如何处理我的测试? 我很高兴你会问这个问题, 因为你会得到答案.

有个很好的消息, Karma 支持 RequireJS. 只需安装最新和最好版本的 Karma.(使用 `npm install -g karma`).

一旦你安装好 Karma, Karma 针对单元测试的配置也会响应的改变. 以下是我们如果设置我们的单元测试来运行我们之前定义的项目结构:

```
// 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(包括: **false** 是非常重要的). 我们还添加了 **REQUIRE_JS** 和适配依赖. 最终进行这一系列工作的是 `main.js`, 他会触发我们的测试.

```

// 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();

        });

    });
});

```

由此设置，我们可以运行下面的命令

```

karma start config/karma.conf.js

```

然后我们就可以运行测试了。

当然，当它涉及到编写单元测试就需要稍微的改变一下。它们需要 RequireJS 支持的模块，因此让我们来看一个测试范例：

```

// 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();  
});  
});  
});
```

我们的每个测试将做到以下几点:

1. 拉取 `angularMocks` 获取我们的 `angular`, `angularResource`, 当然还有 `angularMocks`.
2. 拉取高级模块(`directives` 中的指令, `controllers` 中的控制器等等), 然后它实际上测试的是单独的文件(`loadingIndicator`).
3. 如果你的测试愈来愈其他的服务或者控制器, 除了在 `AngularJS` 中告知意外, 要确保也定义在 `RequireJS` 的依赖中.

这种方法可以用于任何测试, 而且你应该这么做.

值得庆幸的是, `RequireJS` 的处理方式并不会影响我们所有的端到端的测试, 因此可以使用我们目前所看到的方式简单的做到这一点. 一个范例配置如下, 假设你的服务其在 `http://localhost:8000` 上运行你的应用程序:

```
// 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 captures browsers, runs tests and exits

singleRun = false;
```

第四章 分析一个 AngularJS 应用程序

在第 2 章中，我们已经讨论了一些 AngularJS 常用的功能，然后在第 3 章讨论了该如何结构化开发应用程序。现在，我们不再继续深单个的技术点，第 4 章将着眼于一个小的，实际的应用程序进行讲解。我们将从一个实际有效的应用程序中感受一下我们之前已经讨论过的(示例)所有的部分。

我们将每次介绍一部分，然后讨论其中有趣和关联的部分，而不是讨论完整应用程序的前端和核心，最后在本章的后面我们会慢慢简历这个完整的应用程序。

目录

- [应用程序](#)
- [模型，控制器和模板之间的关系](#)
- [模型](#)
- [控制器，指令和服务](#)
- [服务](#)
- [指令](#)

- [控制器](#)
- [模板](#)
- [测试](#)
- [单元测试](#)
- [脚本测试](#)

应用程序

Github 是一个简单的食谱管理应用，我们设计它用于存储我们超级美味的食谱，同时展示 **AngularJS** 应用程序的各个不同的部分。这个应用程序包含以下内容：

- 一个两栏的布局
- 在左侧有一个导航栏
- 允许你创建新的食谱
- 允许你浏览现有的食谱列表

主视图在左侧，其变化依赖于 **URL**，或者食谱列表，或者单个食谱的详情，或者可添加新食谱的编辑功能，或者编辑现有的食谱。我们可以在图 **4-1** 中看到这个应用的一个截图：

GutHub

New Recipe
Recipe List

Cookies

Delicious, crisp on the outside,

Ingredients

1 packet Chips Ahoy

Instructions

1. Go buy a packet of Chips Ahoy
2. Heat it up in an oven
3. Enjoy warm cookies
4. Learn how to bake cookies

Ed

Figure 4-1. Github: A simple recipe management application

这个完整的应用程序可以在我们的 Github 中的 [chapter4/github](#) 中得到。

模型，控制器和模板之间的关系

在我们深入应用程序之前，让我们来花一两段文字来讨论以下如何将标题中的者三部分在应用程序中组织在一起工作，同时来思考一下其中的每一部分。

`model`(模型)是真理。只需要重复这句话几次。整个应用程序显示什么，如何显示在视图中，保存什么，等等一切都会受模型的影响。因此你要额外花一些时间来思考你的模型，对象的属性将是什么，以及你打算如何从服务器获取并保存它。视图将通过数据绑定的方式自动更新，所以我们的焦点应该集中在模型上。

`controller` 保存业务逻辑: 如何获取模型, 执行什么样的操作, 视图需要从模型中获取什么样的信息, 以及如何将模型转换为你所想要的. 验证职责, 使用调用服务器, 引导你的视图使用正确的数据, 大多数情况下所有的这些事情都属于控制器来处理. 最后, `template` 代表你的模型将如何显示, 以及用户将如何与你的应用程序交互. 它主要约束以下几点:

- 显示模型
- 定义用户可以与你的应用程序交互的方式(点击, 文本输入等等)
- 应用程序的样式, 并确定何时以及如何显示一些元素(显示或隐藏, `hover` 等等)
- 过滤和格式化数据(包括输入和输出)

要意识到在 **Angular** 中的模型-视图-控制器涉及模式中模板并不是必要的部分. 相关, 视图是模板获取执行被编译后的版本. 它是一个模板和模型的组合.

任何类型的业务逻辑和行为都不应该进入模板中; 这些信息应该被限制在控制器中. 保持模板的简单可以适当的分离关注点, 并且可以确保你只使用单元测试的情况下就能够测试大多数的代码. 而模板必须使用场景测试的方式来测试.

但是, 你可能会问, 在哪里操作 **DOM** 呢? **DOM** 操作并不会真正进入到控制器和模板中. 它会存在于 **Angular** 的指令中(有时候也可以通过服务来处理, 这样可以避免重复的 **DOM** 操作代码). 我们会在我们的 **GutHub** 的示例文件中涵盖一个这样的例子.

废话少说, 让我们来深入探讨一下它们.

模型

对于应用程序我们要保持模型非常简单. 这一有一个菜谱. 在整个完整的应用程序中, 它们是一个唯一的模型. 它是构建一切的基础.

每个菜谱都有下面的属性:

- 一个用于保存到服务器的 ID
- 一个名称
- 一个简短的描述
- 一个烹饪说明
- 是否是一个特色的菜谱
- 一个成份数组, 每个成分的数量, 单位和名称

就是这样. 非常简单. 应用程序的中一切都基于这个简单的模型. 下面是一个让你食用的示例菜谱(如图 4-1 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'
}
```

下面我们将会看到如何基于这个简单的模型构建更复杂的 UI 特性.

控制器, 指令和服务

现在我们终于可以得到这个让我们牙齿都咬到肉里面去的美食应用程序了. 首先, 我们来看看代码中的指令和服务, 以及讨论以下它们都是做什么的, 然后我们我们会看看这个应用程序需要的多个控制器.

服务

```
//this file is app/scripts/services/services.js

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

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

services.factory('MultiRecipeLoader', ['Recipe', '$q', function(Recipe,
q){
    return function(){
        var delay = $.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;

    };

}]);

```

首先让我们来看看我们的服务。在 33 页的"使用模块组织依赖"小节中已经涉及到了服务相关的知识。这里，我们将会更深一点挖掘服务相关的信息。

在这个文件中，我们实例化了三个 AngularJS 服务。

有一个菜谱服务，它返回我们所调用的 Angular Resource。这些是 RESTful 资源，它指向一个 RESTful 服务器。Angular Resource 封装了低层的 \$http 服务，因此你可以在你的代码中只处理对象。

注意单独的那行代码 - `return $resource` - (当然，依赖于 `github.services` 模型)，现在我们可以将 `recipe` 作为参数传递给任意的控制器中，它将会注入到控制器中。此外，每个菜谱对象都内置的有以下几个方法：

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

如果你使用了 `Recipe.delete` 方法，并且希望你的应用程序工作在 IE 中，你应该像这样调用它：`Recipe[delete]()`。这是因为在 IE 中 `delete` 是一个关键字。

对于上面的方法，所有的查询众多都在一个单独的菜谱中进行；默认情况下 `query()` 返回一个菜谱数组。

`return $resource` 这行代码用于声明资源 - 也给了我们一些好东西：

1. 注意: URL 中的 id 是指定的 RESTful 资源. 它基本上是说, 当你进行任何查询时 (`Recipe.get()`), 如果你给它传递一个 id 字段, 那么这个字段的值将被添加至 URL 的尾部.

也就是说, 调用 `Recipe.get{id: 15}` 将会请求 `/recipe/15`.

1. 那第二个对象是什么呢? `{id: @id}` 吗? 是的, 正如他们所说的, 一行代码可能需要一千行解释, 那么让我们举一个简单的例子.

比方说我们有一个 `recipe` 对象, 其中存储了必要的信息, 并且包含一个 `id`.

然后, 我们只需要像下面这样做就可以保存它:

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

这将会触发一个 POST 请求到 `/recipe/13`.

`@id` 用于告诉它, 这里的 id 字段取自它的对象中同时用于作为 id 参数. 这是一个附加的便利操作, 可以节省几行代码.

在 `apps/scripts/services/services.js` 中有两个其他的 service. 它们两个都是加载器 (Loaders); 一个用于加载单独的食谱 (`RecipeLoader`), 另一个用于加载所有的食谱 (`MultiRecipeLoader`). 这在我们连接到我们的路由时使用. 在核心上, 它们两个表现得非常相似. 这两个 service 如下:

1. 创建一个 `$q` 延迟 (deferred) 对象 (它们是 AngularJS 的 promises, 用于链接异步函数).
2. 创建一个服务器调用.
3. 在服务器返回值时 resolve 延迟对象.
4. 通过使用 AngularJS 的路由机制返回 promise.

AngularJS 中的 Promises

一个 promise 就是一个在未来某个时刻处理返回对象或者填充它的接口 (基本上都是异步行为). 从核心上讲, 一个 promise 就是一个带有 `then()` 函数 (方法) 的对象.

让我们使用一个例子来展示它的优势, 假设我们需要获取一个用户的当前配置:

```

var currentProfile = null;

var username = 'something';

fetchServerConfig(function(){

    fetchUserProfiles(serverConfig.USER_PROFILES, username,

        function(profiles){

            currentProfile = profiles.currentProfile;

        });

});

```

对于这种做法这里有一些问题:

1. 对于最后产生的代码, 缩进是一个噩梦, 特别是如果你要链接多个调用时.
2. 在回调和函数之间错误报告的功能有丢失的倾向, 除非你在每一步中处理它们.
3. 对于你想使用 `currentProfile` 做什么, 你必须在内层回调中封装其逻辑, 无论是直接的方式还是使用一个单独分离的函数.

Promises 解决了这些问题. 在我们进入它是如何解决这些问题之前, 先让我们来看看一个使用 **promise** 对同一问题的实现.

```

var currentProfile = fetchServerConfig().then(function(serverConfig){

    return fetchUserProfiles(serverConfig.USER_PROFILES, username);

}).then(function{

    return profiles.currentProfile;

}, function(error){

    // Handle errors in either fetchServerConfig or

    // fetchUserProfile here

});

```

注意其优势:

1. 你可以链接函数调用, 因此你不会产生缩进带来的噩梦.
2. 你可以放心前一个函数调用会在下一个函数调用之前完成.
3. 每一个 `then()` 调用都要两个参数(这两个参数都是函数). 第一个参数是成功的操作的回调函数, 第二个参数是错误处理的函数.
4. 在链接中发生错误的情况下, 错误信息会通过错误处理器传播到应用程序的其他部分. 因此, 任何回调函数的错误都可以在尾部被处理.

你会问, 那什么是 `resolve` 和 `reject` 呢? 是的, `deferred` 在 AngularJS 中是一种创建 `promises` 的方式. 调用 `resolve` 来满足 `promise`(调用成功时的处理函数), 同时调用 `reject` 来处理 `promise` 在调用错误处理器时的事情.

当我们链接到路由时, 我们会再次回到这里.

指令

我们现在可以转移到即将用在我们应用程序的指令上来. 在这个应用程序中将有两个指令:

`butterbar`

这个指令将在路由发生改变并且页面仍然还在加载信息时处理显示和隐藏任务. 它将连接路由变化机制, 基于页面的状态来自动控制显示或者隐藏是否使用哪个标签.

`focus`

这个 `focus` 指令用于确保指定的文本域(或者元素)拥有焦点.

让我们来看一下代码:

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

var directive = 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');
        });

        $routeScope.$on('$routeChangeSuccess', function(){
            element.addClass('hide');
        });
    }
};

}]]);

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

```

上面所述的指令返回一个对象带有一个单一的属性, **link**. 我们将在第六章深入讨论你可以如何创建你自己的指令, 但是现在, 你应该知道下面的所有事情:

1. 指令通过两个步骤处理. 在第一步中(编译阶段), 所有的指令都被附加到一个被查找到的 **DOM** 元素上, 然后处理它. 任何 **DOM** 操作否发生在编译阶段(步骤中). 在这个阶段结束时, 生成一个连接函数.

2. 在第二步中, 连接阶段(我们之前使用的阶段), 生成前面的 DOM 模板并连接到作用域. 同样的, 任何观察者或者监听器都要根据需要添加, 在作用域和元素之前返回一个活动(双向)绑定. 因此, 任何关联到作用域的事情都发生在连接阶段.

那么在我们指令中发生了什么呢? 让我们去看一看, 好吗?

`butterbar` 指令可以像下面这样使用:

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

它基于前面隐藏的元素, 然后添加两个监听器到根作用域中. 当每次一个路由开始改变时, 它就显示该元素(通过改变它的 `class[className]`), 每次路由成功改变并完成时, 它再一次隐藏 `butterbar`.

另一个有趣的事情是注意我们是如何注入 `$rootScope` 到指令中的. 所有的指令都直接挂接到 AngularJS 的依赖注入系统, 因此你可以注入你的服务和其他任何需要的东西到其中.

最后需要注意的是处理元素的 API. 使用 jQuery 的人会很高兴, 因为他直到使用的是一个类似 jQuery 的语法(`addClass`, `removeClass`). AngularJS 实现了一个调用 jQuery 的自己, 因此, 对于任何 AngularJS 项目来说, jQuery 都是一个可选的依赖项. 如果你最终在你的项目中使用完整的 jQuery 库, 你应该直到它使用的是它自己内置的 jQlite 实现.

第二个指令(`focus`)简单得多. 它只是在当前元素上调用 `focus()` 方法. 你可以用过在任何 `input` 元素上添加 `focus` 属性来调用它, 就像这样:

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

当页面加载时, 元素将立即获得焦点.

控制器

随着指令和服务的覆盖, 我们终于可以进入控制器部分了, 我们有五个控制器. 所有的这些控制器都在一个单独的文件中(`app/scripts/controllers/controllers.js`), 但是我们会一个个来了解它们. 让我们来看第一个控制器, 这是一个列表控制器, 负责显示系统中所有的食谱列表.

```
app.controller('ListCtrl', ['$scope', 'recipes', function($scope,
recipes){

    $scope.recipes = recipes;
```

```
});
```

注意列表控制器中最重要的一个事情：在这个控制器中，它并没有连接到服务器和获取食谱。相反，它只是使用已经取得的食谱列表。你可能不知道它是如何工作的。你可能会使用路由一节来回答，因为它有一个我们之前看到 `MultiRecipeLoader`。你只需要在脑海里记住它。

在我们提到的列表控制器下，其他的控制器都与之非常相似，但我们仍然会逐步指出它们有趣的地方：

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

    $scope.recipe = recipe;

    $scope.edit = function(){

        $location.path('/edit/' + recipe.id);

    };

}]);
```

这个视图控制器中有趣的部分是其编辑函数公开在作用域中。而不是显示和隐藏字段或者其他类似的东西，这个控制器依赖于 **AngularJS** 来处理繁重的任务(你应该这么做)！这个编辑函数简单的改变 **URL** 并跳转到编辑食谱的部分，你可以看见，**AngularJS** 并没有处理剩下的工作。**AngularJS** 识别已经改变的 **URL** 并加载响应的视图(这是与我们编辑模式中相同的食谱部分)。来看一看！

接下来，让我们来看看编辑控制器：

```
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('/');

};

}]);

```

那么在这个暴露在作用域中的编辑控制器中新的 `save` 和 `remove` 方法有什么。那么你希望作用域内的 `save` 函数做什么。它保存当前食谱，并且一旦保存好，它就在屏幕中将用户重定向到相同的食谱。回调函数是非常有用的，一旦你完成任务的情况下执行或者处理一些行为。

有两种方式可以在这里保存食谱。一种是如代码所示，通过执行 `$scope.recipe.$save()` 方法。这只是可能，因为 `recipe` 是一个通过开头部分的 `RecipeLoader` 返回的资源对象。

另外，你可以像这样来保存食谱：

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

`remove` 函数也是很简单的，在这里它会从作用域中移除食谱，同时将用户重定向到主菜单页。请注意，它并没有真正的从我们的服务器上删除它，尽管它很再做出额外的调用。

接下来，我们来看看 **New** 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);

        });

    };

}]);

```

New 控制器几乎与 **Edit** 控制器完全一样. 实际上, 你可以结合两个控制器作为一个单一的控制器来做一个练习. 唯一的主要不同是 **New** 控制器会在第一步创建一个新的食谱(这也是一个资源, 因此它也有一个 `save` 函数). 其他的一切都保持不变.

最后, 我们还有一个 **Ingredients** 控制器. 这是一个特殊的控制器, 在我们深入了解它为什么或者如何特殊之前, 先来看一看它:

```

app.controller('Ingredients', ['$scope', function($scope){

    $scope.addIngredients = 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与 UI 视图上的相关部分联系着. 但是这个 **Ingredients** 控制器是特殊的. 它是一个子控制器, 用于在编辑页面封装特定的恭喜而不需要在外层(父级)来处理. 有趣的是要注意, 由于它是一个子控制器, 继承自作用域中的父控制器(在这里就是 **Edit/New** 控制器). 因此, 它可以访问来自父控制器的 `$scope.recipe`.

这个控制器本身并没有什么有趣或者独特的地方. 它只是添加一个新的成份到现有的食谱成份数组中, 或者从食谱的成份列表中删除一个特定的成份.

那么现在, 我们就来完成最后的控制器. 唯一的 JavaScript 代码块展示了如何设置路由:

```
// 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: '/'});

}]);

```

正如我们所承诺的，我们终于到了解析函数使用的地方。前面的代码设置 **Guthub AngularJS** 模块，路由以及参与应用程序的模板。

它挂接到我们已经创建的指令和服务上，然后指定不同的路由指向应用程序的不同地方。

对于每个路由，我们指定了 **URL**，它备份控制器，加载模板，以及最后(可选的)提供了一个 `resolve` 对象。

这个 `resolve` 对象会告诉 **AngularJS**，每个 `resolve` 键需要在确定路由正确时才能显示给用户。对我们来说，我们想要加载所有的食谱或者个别的配方，并确保在显示页面之前服务器要响应我们。因此，我们要告诉路由提供者我们的食谱，然后再告诉他如何来取得它。

这个环节中我们在第一节中定义了两个服务，分别是 `MultiRecipeLoader` 和 `RecipeLoader`。如果 `resolve` 函数返回一个 **AngularJS promise**，然后 **AngularJS** 会智能在获得它之前等待 **promise** 解决问题。这意味着它将会等待到服务器响应。

然后将返回结果传递到构造函数中作为参数(与来自对象字段的参数的名称一起作为参数)。

最后, `otherwise` 函数表示当没有路由匹配时重定向到默认 URL.

你可能会注意到 `Edit` 和 `New` 控制器两个路由通向同一个模板

URL, `views/recipeForm.html`. 这里发生了什么呢? 我们复用了编辑模板. 依赖于相关的控制器, 将不同的元素显示在编辑食谱模板中.

完成这些工作之后, 现在我们可以聚焦到模板部分, 来看看控制器如何挂接到它们之上, 以及如何管理现实给最终用户的内容.

模板

让我们首先来看看最外层的主模板, 这里就是 `index.html`. 这是我们单页应用程序的基础, 同时所有其他的视图也会装在这个模板的上下文中:

```
<!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/controlers/controllers.js"></script>

    <link rel="stylesheet" href="styles/bootstrap.css">

    <link rel="stylesheet" href="styles/guthub.css">

  </head>

  <body>

    <header>

      <h1>Guthub</h1>

    </header>

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

```

<div class="container-fluid">

  <div class="row-fluid">

    <div class="span2">

      <!-- Sidebar -->

      <div class="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>

```

注意前面的模板中有 5 个有趣的元素，其中大部分你在第 2 章中都已经见过了。让我们逐个来看看它们：

`ng-app`

我们设置了 `ng-app` 模块为 `Guthub`。这与我们在 `angular.module` 函数中提供的模块名称相同。这便是 AngularJS 如何知道两个挂接在一起的原因。

`script` 标签

这表示应用程序在哪里加载 AngularJS。这必须在所有使用 AngularJS 的 JS 文件被加载之前完成。理想的情况下，它应该在 `body` 的底部完成(`</body>`之前)。

`Butterbar`

我们第一次使用自定义指令。在我们定义我们的 `butterbar` 指令之前，我们希望将它用于一个元素，以便在路由改变时显示它，在成功的时候隐藏它(`loading...`处理)。需要突出显示这个元素的文本(在这里我们使用了一个非常烦人的"`Loading...`")。

链接的 `href` 值

`href` 用于链接到我们单页应用程序的各个页面。追它们如何使用`#`字符来确保页面不会重载的，并且相对于当前页面。AngularJS 会监控 URL(只要页面没有重载)，然后在需要的时候起到神奇的作用(或者通常，将这个非常烦人的路由管理定义为我们路由的一部分)。

`ng-view`

这是最后一个神奇的杰作。在我们的控制器一节，我们定义了路由。作为定义的一部分，每个路由表示一个 URL，控制器关联路由和一个模板。当 AngularJS 发现一个路由改变时，它就会加载关联的模板，并将控制器添加给它，同时替换 `ng-view` 为该模板的内容。

有一件引人注目的事情是这里缺少 `ng-controller` 标签。大部分应用程序某种程度上都需要一个与外部模板关联的 **MainController**。其最常见的位置是在 `body` 标签上。在这种情况下，我们并没有使用它，因为完整的外部模板没有 AngularJS 内容需要引用到一个作用域。

现在来看看与每个控制器关联的单独的模板，就从"食谱列表"模板开始：

```
<!-- File is chapter4/guthub/app/view/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>
```

是的，它是一个非常无聊(普通)的模板。这里只有两个有趣的点。第一个是非常标准的 `ng-repeat` 标签用法。它会获得作用域内的所有食谱并重复检出它们。

第二个是 `ng-href` 标签的用法而不是 `href` 属性。这是一个在 AngularJS 加载期间纯粹无效的空白链接。`ng-href` 会确保任何时候都不会给用户呈现一个畸形的链接。总是会使用它在任何时候使你的 URLs 都是动态的而不是静态的。

当然，你可能感到奇怪：控制器在哪里？这里没有 `ng-controller` 定义，也确实没有 **Main Controller** 定义。这是路由映射发挥的作用。如果你还记得(或者往前翻几页)，`/` 路由会重定向到列表模板并且带有与之关联的 **ListController**。因此，当引用任何变量或者类似的东西时，它都在 **List Controller** 作用域内部。

现在我们来看一些有更多实质内容的东西：视图形式。

```
<!-- File is chapter4/guthub/app/views/viewRecipe.html -->

<h2>{{recipe.title}}</h2>

<div>{{recipe.decription}}</div>

<h3>Ingredients</h3>

<span ng-show="recipe.ingredients.length == 0">No Ingredients</span>

<ul class="unstyled" ng-hide="recipe.ingredients.length == 0">

  <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>
```

这是另一个不错的，很小的包含模板。我们将提醒你注意三件事，虽然不会按照它们所出现的顺序。

第一个就是非常标准的 `ng-repeat`. 食谱(recipes)再次出现在 View Controller 作用域中, 这是用过在页面现实给用户之前通过 `resolve` 函数加载的. 这确保用户查看它时也面不是一个破碎的, 未加载的状态.

接下来一个有趣的用法是使用 `ng-show` 和 `ng-class`(这里应该是 `ng-hide`)来设置模板的样式. `ng-show` 标签被添加到`<i>`标签上, 这是用来显示一个星号标记的 icon. 现在, 这个星号标记只在食谱是一个特色食谱的时候才显示(例如通过 `recipe.featured` 布尔值来标记). 理想的情况下, 为了确保适当的间距, 你需要使用一个空白的空格图标, 并给这个空格图标绑定 `ng-hide` 指令, 然后同归同样的 AngularJS 表达式 `ng-show` 来显示. 这是一个常见的用法, 显示一个东西并在给定的条件下来隐藏.

`ng-class` 用于添加一个类(CSS 类)给`<h2>`标签(在这种情况下就是"特色")当食谱是一个特色食谱时. 它添加了一些特殊的高亮来使标题更加引人注目.

最后一个需要注意的时表单上的 `ng-submit` 指令. 这个指令规定在表单被提交的情况下调用 `scope` 中的 `edit()` 函数. 当任何没有关联明确函数的按钮被点击时机会提交表单(这种情况下便是 Edit 按钮). 同样, AngularJS 足够智能的在作用域中(从模块,路由,控制器中)在正确的时间里引用和调用正确的方法.

上面这段解释与原书代码有一些差别, 读者自行理解. 原书作者暂未给出解答.

现在我们可以来看看我们最后的模板(可能目前为止最复杂的一个), 食谱表单模板:

```
<!-- file is chapter4/github/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 required>

    </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" ng-
disabled="recipeForm.$invalid">Save</button>

        <button type="button" ng-click="remove()" ng-show="!recipe.id"
class="btn">Delete</button>

    </div>

</form>

```

不要惊慌。它看起来像很多代码，并且它是一个很长的代码，但是如果你认真研究以下它，你会发现它并不是非常复杂。事实上，其中很多都是很简单的，比如重复的显示可编辑输入字段用于编辑食谱的模板：

- `focus` 指令被添加到第一个输入字段上(`title` 输入字段)。这确保当用户导航到这个页面时，标题字段会自动聚焦，并且用户可以立即开始输入标题信息。
- `ng-submit` 指令与前面的例子非常相似，因此我们不会深入讨论它，它只是保存是食谱的状态和编辑过程的结束信号。它会挂接到 **Edit Controller** 中的 `save()` 函数。
- `ng-model` 指令用于将不同的文本输入框和文本域绑定到模型中。

在这个页面更有趣的一方面，并且我们建议你花一点时间来了解它的便是配料列表部分的 `ng-controller` 标签。让我们花一分钟的事件来了解以下这里发生了什么。

我们看到了一个显示配料成份的列表，并且容器标签关联了一个 `ng-controller`。这意味着这个 `` 标签是 **Ingredients Controller** 的作用域。但是这个模板实际的控制器是什么呢，是 **Edit Controller**？事实证明，**Ingredients Controller** 是作为 **Edit Controller** 的子控制器创建的，从而继承了 **Edit Controller** 的作用域。这就是为什么它可以从 **Edit Controller** 访问食谱对象(recipe object)的原因。

此外，它还增加了一个 `addIngredient()` 方法，这是通过处理高亮的 `ng-click` 使用的，那么就只能在 `` 标签作用域内访问。那么为什么你需要这么做呢？因为这是分离你担忧的最好的方式。为什么 **Edit Controller** 需要一个 `addIngredients()` 方法，问 99% 的模板都不会关心它。因为如此精确你的子控制器和嵌套控制器是很不错的，它可以包含任务并循序你分离你的业务逻辑到更多的可维护模块中。

- 另外一个控制器便是我们在这里想要深入讨论的表单验证控制。它很容易在 **AngularJS** 中设置一个特定的表单字段为"必填项"。只需要简单的添加 `required` 标签到输入框上(与前面的代码中的情况一样)。但是现在你要怎么对它。

为此，我们先跳到保存按钮部分。注意它上面有一个 `ng-disabled` 指令，这换言之就是 `recipeForm.$invalid`。这个 `recipeForm` 是我们已经声明的表单名称。**AngularJS** 增加了一些特殊的变量(`$valid` 和 `$invalid` 只是其中两个)允许你控制表单的元素。

AngularJS 会查找到所有的必填元素并更新所对应的特殊变量。因此如果我们的 **Recipe Title** 为空，`recipeForm.$invalid` 就会被这只为 `true` (`$valid` 就为 `false`)，同时我们的保存(Save)按钮就会立刻被禁用。

我们还可以给一个文本输入框设置最大和最小长度(输入长度)，以及一个用于验证一个输入字段的正则表达式模式。另外，这里还有只在满足特定条件时用于显示特定错误消息的高级用法。让我们使用一个很小的分支例子来看看：

```
<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>
```

在前面的这个例子中，我们添加了一要求：用户名至少是三个字符(通过使用 `ng-minlength` 指令)。现在，表单范围内会关心每个命名输入框的填充形式--在这个例子中我们只有一个 `userName`--其中每个输入框都会有一个 `$error` 对象(这里具体的还包括什么样的错误或者没有错误：`required`，`minlength`，`maxlength` 或者模式)和一个 `$valid` 标签来表示输入框本身是否有效。

我们可以利用这个来选择性的将错误信息显示给用户，这根据不同的输入错误类型来显示，正如我们上面的实例所示。

跳回到我们原来的模板中--Recipe 表单模板--在这里的 ingredients repeater 里面还有另外一个很好的 `ng-show` 高亮的用法. 这个 Add Ingredient 按钮只会在最后的一个配料的旁边显示. 着可以通过在一个 repeater 元素范围内调用一个 `ng-show` 并使用特殊的 `$last` 变量来完成.

最后我们还有最后的一个 `ng-click`, 这是附加的第二个按钮, 用于删除该食谱. 注意这个按钮只会在食谱尚未保存的时候显示. 虽然通常它会编写一个更有意义的 `ng-hide="recipe.id"`, 有时候它会使用更有语义的 `ng-show="!recipe.id"`. 也就是说, 如果食谱没有一个 id 的时候显示, 而不是在食谱有一个 id 的时候隐藏.

测试

随着控制器部分, 我们已经推迟向你显示测试部分了, 但你知道它会即将到来, 不是吗? 在这一节, 我们将会涵盖你已经编写部分的代码测试, 以及涉及你要如何编写它们.

单元测试

第一个, 也是非常重要的一种测试是单元测试. 对于控制器(指令和服务)的测试你已经开发和编写的正确的结构, 并且你可能会想到它们会做什么.

在我们深入到各个单元测试之前, 让我们围绕所有我们的控制器单元测试来看看测试装置:

```
describe('Controllers', function() {

    var $scope, ctrl;

    //you need to include your module in a test

    beforeEach(module('github'));

    beforeEach(function() {

        this.addMatchers({

            toEqualData: function(expected) {

                return angular.equals(this.actual, expected);

            }

        });

    });

});
```

```
});

describe('ListCtrl', function() {....});

// Other controller describes here as well

});
```

这个测试装置(我们仍然使用 **Jasmine** 的行为方式来编写这些测试)做了几件事情:

1. 创建一个全局(至少对于这个测试规范是这个目的)可访问的作用域和控制器, 所以我们不用担心每个控制器会创建一个新的变量.
2. 初始化我们应用程序所用的模块(在这里是 **Github**).
3. 添加一个我们称之为 `equalData` 的特殊的匹配器. 这基本上允许我们在资源对象(就像食谱)通过 `$resource` 服务和调用 **RESTful** 来执行断言(测试判断).

记得在任何我们处理在 `ngResource` 上返回对象的断言时添加一个称为 `equalData` 特殊匹配器. 这是因为 `ngResource` 返回对象还有额外的方法在它们失败时默认希望调用 `equal` 方法.

这个装置到此为止, 让我们来看看 **List Controller** 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]);

    });

});

```

记住这个 **List Controller** 只是我们最简单的控制器之一。这个控制器的构造器只是接受一个食谱列表并将它保存到作用域中。你可以编写一个测试给它，但它似乎有一点不合理(我们还是这么做了，因为这个测试很不错)。

相反，更有趣的是 **MulyiRecipeLoader** 服务方面。它负责从服务器上获取食谱列表并将它作为一个参数传递(当通过 `$route` 服务正确的连接时)。

```

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

```

在我们的测试中，我们通过挂接到一个模拟的 `HttpBackend` 来测试 `MultiRecipeLoader`。这来自于测试运行时所包含的 `angular-mocks.js` 文件。只需将它注入到你的 `beforeEach` 方法中就足以让你设置预期目的。接下来，我们进行了一个更有意义的测试，我们期望设置一个服务器的 `GET` 请求来获取 `recipes`，浙江返回一个简单的数组对象。然后使用我们新的自定义的匹配器来确保正确的返回数据。注意在模拟 `backend` 中的 `flush()` 调用，这将告诉模拟 `Backend` 从服务器返回响应。你可以使用这个机制来测试控制流程和查看你的应用程序在服务器返回一个响应之前和之后是如何处理的。

我们将跳过 `View Controller`，因为它除了在作用域中添加一个 `edit()` 方法之外基于与 `List Controller` 一模一样。这是非常简单的测试，你可以在你的测试中注入 `$location` 并检查它的值。

现在让我们跳到 `Edit Controller`，其中有两个有趣的点我们进行单元测试。一个是类似我们之前看到过的 `resolve` 函数，并且可以以同样的方式测试。相反，我们现在想看看我们可以如和测试 `save()` 和 `remove()` 方法。让我们来看看对于它们的测试(假设我们的测试工具来自于前面的例子):

```
describe('EditController', function() {
```

```
var mockBackend, location;

beforeEach(inject($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('/');

    });
});
```

在第一个测试用，我们测试了 `save()` 函数。特别是，我们确保在我们的对象保存时首先创建一个到服务器的 **POST** 请求，然后，一旦服务器响应，地址就改变到新的持久对象的视图食谱页面。

第二个测试更简单。我们进行了简单的检测以确保在作用域中调用 `remove()` 方法的时候移除当前食谱，然后重定向到用户主页。这可以很容易通过注入 `$location` 服务到我们的测试中并使用它。

其余的针对控制器的单元测试遵循非常相似的模式，因此在这里我们跳过它们。在他们的底层中，这些单元测试依赖于一些事情：

- 确保控制器(或者更可能是作用域)在结束初始化时达到正确的状态
- 确认经行正确的服务器调用，以及通过作用域在服务器调用期间和完成后去的正确的状态(通过在单元测试中使用我们的模拟后端服务)
- 利用 **AngularJS** 的依赖注入框架着手处理元素以及控制器对象用于确保控制器会设置正确的状态。

脚本测试

一旦我们对单元测试很满意，我们可能禁不住往后靠一下，抽根雪茄，收工。但是 **AngularJS** 开发者不会这么做，直到他们完成了他们的脚本测试(场景测试)。虽然单元测试确保我们的每一块 **JS** 代码都按照预期工作，我们也要确保模板加载，并正确的挂接到控制器上，以及在模板重点点击做正确的事情。

这正是 **AngularJS** 带给你的脚本测试(场景测试)，它允许你做以下事情：

- 加载你的应用程序
- 浏览一个特定的页面
- 随意的点击周围和输入文本
- 确保发生正确的事情

所以，脚本测试如何在我们的"食谱列表"页面工作？首先，在我们开始实际的测试之前，我们需要做一些基础工作。

对于该脚本测试工作，我们需要一个工作的 **Web** 服务器以准备从 **Github** 应用上接受请求，同时将允许我们从它上面存储和获取一个食谱列表。随意的更改代码以使用内存中的食谱列表(移除 `$resource` 食谱并只是将它转换为一个 **JSON** 对象)，或者复用和修改我们前面章节向你展示的 **Web** 服务器，或者使用 **Yeoman**!

一旦我们有了一个服务器并运行起来，同时服务于我们的应用程序，然后我们就可以编写和运行下面的测试：

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

第五章 与服务器通信

目前，我们已经接触过下面要谈的主题的主要内容，这些内容包括你的 **Angular** 应用如何规划设计、不同的 **AngularJS** 部件如何装配在一起并正常工作以及 **AngularJS** 中的模板代码运行机制的一小部分内容。把它们结合在一起，你就可以搭建一些简洁优雅的 **Web** 应用，但他们的运作主要还是限制在客户端。在前面第二章，我们接触了一点用 `$http` 服务做与服务器端通信的内容，但是在这一章，我们将会深入探讨如何在现实世界的真实应用中使用它(`$http`)。

在这一章，我们将讨论一下 **AngularJS** 如何帮你与服务器端通信，这其中包括在最低抽象等级的层面或者用它提供的优雅的封装器。而且我们将会深入探讨 **AngularJS** 如何用内建缓存机制来帮你加速你的应用。如果你想用 `SocketIO` 开发一个实时的 **Angular** 应用，那么第八章有一个例子，演示了如何把 `SocketIO` 封装成一个指令然后如何使用这个指令，在这一章，我们就不涉及这方面内容了。

目录

- [通过\\$http 进行通行](#)
 - [进一步配置你的请求](#)
 - [设定 HTTP 头信息\(Headers\)](#)
 - [缓存响应数据](#)
 - [对请求\(Request\)和响应\(Response\)的数据所做的转换](#)
- [单元测试](#)
- [使用 RESTful 资源](#)
 - [resource 资源的声明](#)
 - [定制方法](#)
 - [不要使用回调函数机制!\(除非你真的需要它们\)](#)
 - [简化的服务器端操作](#)
 - [对 ngResource 做单元测试](#)
- [\\$q 和预期值\(Promise\)](#)
- [响应拦截处理](#)
- [安全方面的考虑](#)
 - [JSON 的安全脆弱性](#)
 - [跨站请求伪造\(XSRF\)](#)

通过\$http 进行通行

从 **Ajax** 应用(使用 `XMLHttpRequests`)发动一个请求到服务器的传统方式包括：得到一个 `XMLHttpRequest` 对象的引用、发起请求、读取响应、检验错误代码然后最后处理服务器响应。它就是下面这样：

```

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();

```

对于这样一个简单、常用且经常重复的任务，上面这个代码量比较大.如果你想重复性地做这件事,你最终可能会做一个封装或者使用现成的库.

AngularJS XHR(XMLHttpRequest) API 遵循 **Promise** 接口.因为 **XHRs** 是异步方法调用，服务器响应将会在未来一个不定的时间返回(当然希望是越快越好).**Promise** 接口保证了这样的响应将会如何处理,它允许 **Promise** 接口的消费者以一种可预计的方式使用这些响应.

假设我们想从我们的服务器取回用户的信息.如果接口在 **/api/user** 地址可用,并且接受 **id** 作为 **url** 参数，那么我们的 **XHR** 请求就可以像下面这样使用 **Angular** 的核心 **\$http** 服务:

```

$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
});

```

如果你来自 **jQuery** 世界,你可能会注意到: **AngularJS** 和 **jQuery** 处理异步需求的方式很相似.

我们上面例子中使用的 **\$http.get** 方法仅仅是 **AngularJS** 核心服务 **\$http** 提供的众多简便方法之一.类似的，如果你想使用 **AngularJS** 向相同 **URL** 带一些 **POST** 请求数据发起一个 **POST** 请求，你可以像下面这样做:

```

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
});
```

AngularJS 为大多数常用请求类型都提供了类似的简便方法，他们包括：

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

进一步配置你的请求

有时，工具箱提供的标准请求配置还不够,它可能是因为你想做下面这些事情：

- 你可能想为请求添加权限验证的头信息
- 改变请求数据的缓存方式
- 在请求被发送或者响应返回时，对数据以一些方式做一定的转换处理

在上面这样的情况之下,你可以进一步配置请求，通过可选的传递进请求的配置对象.在之前的例子中,我们使用配置对象来标明可选的 **URL** 参数，即便我们哪儿演示的 **GET** 和 **POST** 方法是简便方法。内部的原生方法可能看上面像相面这样：

```
$http(config)
```

下面演示的是一个调用这个方法的伪代码模板：

```
$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 和其它的简便方法已经设置了请求的 `method` 类型,所以不需要再设置这个, `config` 配置对象是传给 `·$http.get·`、`·$http.post·` 方法的最后一个参数,所以当你使用任何简便方法的时候, 你任何能用这个 `config` 配置对象.

你也可以通过传入含有下面这些键的属性集 `config` 对象来改变已有的 `request` 对象

- `method` : 一个表示 `http` 请求类型的字符串, 比如 `GET`,或者 `POST`
- `url` : 一个 `URL` 字符串代表要请求资源的绝对或相对 `URL`
- `params` : 一个对象(准确的说是键值映射)包含字符串到字符串内容, 它代表了将会转换为 `URL` 参数的键值对, 比如下面这样: `[[{key1: 'value1', key2: 'value2'}]` 它将会被转换为: `?key1=value&key2=value2` 这串字符将会加在 `URL` 后面, 如果在 `value` 的位置你用一个对象取代字符串或数字, 那这个对象将会转换为 `JSON` 字符串.
- `data` : 一个字符串或一个对象, 它将会被作为请求消息数据被发送.
- `timeout` : 这是请求被认定为过期之前所要等待的毫秒数.

还有部分另外的选项可以被配置,在下面的章节中, 我们将会深度探索这些选项.

设定 HTTP 头信息(Headers)

AngularJS 有一个默认的头信息,这个头信息将会对所有的发送请求使用,它包含以下信息: 1.Accept: application/json, text/plain, / 2.X-Requested-With:XMLHttpRequest

如果你想设置任何特定的头信息,这儿有两种方法来做这件事:

第一种方法,如果你相对所有的发送请求都使用这些特定头信息,那你需要把特定信息设置为 `Angular` 默认头信息的一部分.可以在 `$httpProvider.defaults.headers` 配置对象里面设置这个,这个步骤通常会在你的 `app` 设置 `config` 部分来做.所以如果你想移除"Requested-With"头信息且对所有的 `GET` 请求启用"DO NOT TRACK"设置,你可以简单地通过以下代码来做:

```
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';
  });
```

如果你只想对某个特定的请求设置头信息,而不是设置默认头信息.那么你可以通过给\$**http** 服务传递包含指定头信息的 **config** 对象来做.相同的定制头信息可以作为第二个参数传递给 **GET** 请求,第一个参数是 **URL** 字符串:

```
$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 });
```

如何在应用中处理权限验证头信息的成熟示例将会在第八章的 **Cheetsheets** 示例部分给出.

缓存响应数据

AngularJS 为 **HTTP GET** 请求提供了一个开箱即用的简单缓存系统.缺省情况下,它对所有的请求都是禁用的,但是如果你想对你的请求启用缓存系统,你可以使用以下代码:

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

这段代码启用了缓存系统,然后 **AngularJS** 将会缓存来自 **Server** 的响应数据.但对相同的 **URL** 的请求第二次发出时,**AngularJS** 将会从缓存里面取出前一次的响应数据作为响应返回.这个缓存系统也很智能,即使你同时对相同 **URL** 发出多个请求,只有一个请求会发向 **Server**,这个请求的响应数据将会反馈给所有(同时发起的)请求。

然而这种做法从可用性的角度看可能是有所冲突的,当一个用户首先看到旧的结果,然后新的结果突然冒出来,比如一个用户可能即将单击一个数据项,而实际上这个数据项后台已经发生了变化.

注意所有响应(即使是从缓存里取出的)本质上仍旧是异步响应.换句话说,期望你的利用缓存响应时的异步代码运行仍旧和他向后台服务器发出请求时的代码运行机制是一样的.

对请求(Request)和响应(Response)的数据所做的转换

AngularJS 对所有 `$http` 服务发起的请求和响应做一些基本的转换,它们包括:

- 请求(Request)转换: 如果请求的 `Cofig` 配置对象的 `data` 属性包含一个对象,将会把这个对象序列化为 JSON 格式.
- 响应(Response)转换: 如果探测到一个 XSRF 头,把它剥离掉.如果响应数据被探测为 JSON 格式,用 JSON 解析器把它反序列化为 JSON 对象.

如果你需要部分系统默认提供的转换,或者想使用你自己的转换,你可以把你的转换函数作为 `Config` 配置对象的一部分传递进去(后面有细述).这些转换函数得到 HTTP 请求和 HTTP 响应的数据主体以及它们的头信息.然后把序列化的修改后版本返回出来.在 `Config` 对象里面配置这些函数需要使用 `·transformRequest·` 键和 `·transformResponse·` 键,这些都可以通过使用 ``$httpProvider·` 服务在模块的 `config` 函数里面配置它.

我们什么时候使用这些哪?让我假设我们有一个服务器,它更习惯于 jQuery 运行的方式.它可能希望我们的 POST 数据以 `key1=val1&key2=val2` 字符串的形式传递,而不是以 `{key1:val1,key2:val2}` 这样的 JSON 格式.这个时候,我们可能相对每个请求做这样的转换,或者单个地增加 `transformRequest` 转换函数,为了达成这个示例这样的目标,我们将要设置一个通用 `transformRequest` 转换函数,以便对所有的发出请求,这个函数都可以把 JSON 格式转换为键值对字符串,下面代码演示了如何做这个工作:

```
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

目前为止，我们已经了解如何使用 `$http` 服务以及如何以可能的方式做你需要的配置。但是如何写一些单元测试来保证这些够真实有效的运行哪？

正如我们曾经三番五次的提到的那样，**AngularJS** 一直以测试为先的原则而设计。所以 **Angular** 有一个模拟服务器后端，在单元测试中，它可以帮你就可以测试你发出的请求是否正确，甚至可以精确控制模拟响应如何得到处理，什么时候得到处理。

让我们探索一下下面这样的单元测试场景：一个控制向服务器发起请求，从服务器得到数据，把它赋给作用域内的模型，然后以具体的模板格式显示出来。

我们的 `NameListCtrl` 控制器是一个非常简单的控制器。它的存在只有一个目的：访问 `names` API 接口，然后把得到数据存储在作用域 `scope` 模型内。

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

怎样对这个控制器做单元测试？在我们的单元测试中，我们必须保证下面这些条件：

- `NamesListCtrl` 能够找到所有的依赖项(并且正确的得到注入的这些依赖)》
- 当控制器加载时尽可能快地立刻发请求从服务器得到 `names` 数据。
- 控制器能够正确地把响应数据存储在作用域 `scope` 的 `names` 变量属性中。

在我们的单元测试中构造一个控制器时，我们给它注入一个 `scope` 作用域和一个伪造的 **HTTP** 服务，在构建测试控制器的方式和生产中构建控制器的方式其实是一样的。这是推荐方法，尽管它看上去上有点复杂。让我看一下具体代码：

```
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 RESTful 资源

·\$http·服务提供一个比较底层的实现来帮你发起 XHR 请求,但是同时也给提供了很强的可控性和弹性.在大多数情况下,我们处理的是对象集或者是封装有一定属性和方法的对象模型,比如带有个人资料的自然人对象或者信用卡对象.

在上面这样的情况下,如果我们自己构建一个 JS 对象来表示这种较复杂对象模型,那做法就有点不够 nice.如果我们仅仅想编辑某个对象的属性、保存或者更新一个对象,那我们如何让这些状态在服务器端持久化.

`$resource` 正好给你提供这种能力.AngularJS `resources` 可以帮助我们以描述的方式来定义对象模型,可以定义一下这些特征:

- `resource` 的服务器端 URL
- 这种请求常用参数的类型

- (你可以免费自动得到 `get`、`save`、`query`、`remove` 和 `delete` 方法),除了那些方法,你可以定义其它的方法,这些方法封装了对象模型的特定功能和业务逻辑(比如信用卡模型的 `charge()` 付费方法)
 - 响应的期望类型(数组或者一个独立对象)
 - 头信息
-

什么时候你可以用 **Angular Resources** 组件?

只有你的服务器端设施是以 **RESTful** 方式提供服务的时候,你才应该用 **Angular resources** 组件.比如信用卡那个案例,我们将用它作为本章这一部分的例子,他将包括以下内容:

1. 给地址 `/user/123/card` 发送一个 **GET** 请求,将会得到用户 **123** 的信用卡列表.
 2. 给地址 `/user/123/card/15` 发送一个 **GET** 请求,将会得到用户 **123** 本人的 ID 为 **15** 的信用卡信息
 3. 给地址 `/user/123/card` 发送一个在 **POST** 请求数据部分包含信用卡信息的 **POST** 请求,将会为用户 **123** 新创建一个信用卡
 4. 给地址 `/user/123/card/15` 发送一个在 **POST** 请求数据部分包含信用卡信息的 **POST** 请求,将会更新用户 **123** 的 ID 为 **5** 的信用卡的信息.
 5. 给地址 `/user/123/card/15` 一个方法为 **DELETE** 类型的请求,将会删除掉用户 **123** 的 ID 为 **5** 的信用卡的数据.
-

除了按照你的要求给你提供一个查询服务器端信息的对象,`$resource` 还可以让你使用返回的数据对象就像他们是持久化数据模型一样,可以修改他们,还可以把你的修改持久化存储下来.

`ngResource` 是一个单独的、可选的模块.要想使用它,你看需要做以下事情:

- 在你的 **HTML** 文件里面引用 `angular-resource.js` 的实际地址
- 在你的模块依赖里面声明对它的依赖(例如,`angular.module('myModule',['ngResource'])`).
- 在需要的地方,注入 `$resource` 这个依赖项.

在我们看怎样用 `ngResource` 方法创建一个 `resource` 资源之前,我们先看一下怎样用基本的 `$http` 服务做类似的事情.比如我们的信用卡 `resource`,我们想能够读取、查询、保存信用卡信息,另外还要能为信用卡还款.

这儿是上述需求一个可能的实现：

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

取代以上方式，你也可以轻松创建一个在你的应用中始终如一的 **Angular** 资源服务，就像下面代码这样：

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

做到现在，你就可以任何时候从 **Angular** 注入器里面请求一个 **CreditCard** 依赖，你就会得到一个 **Angular** 资源，默认情况下，这个资源会提供一些初始的可用方法。表格 5-1 列出了这些初始方法以及他们的运行行为，这样你就可以知道在服务器怎样配置来配合这些方法了。

表格 5-1 一个信用卡 resource Function Method URL Expected Return

CreditCard.get({id: 11})	GET /user/123/card/11	Single JSON	CreditCard.save({}, ccard)	POST /user/123/card with post data “ccard”	Single JSON
CreditCard.save({id: 11}, ccard)	POST /user/123/card/11 with post data “ccard”	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

让我们看一个信用卡 **resource** 使用的代码样例，这样可以让你理解起来觉得更浅显易懂。

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

前面这个样例代码里面发生了很多事情，所以我们将会有一个一个地认真讲解其中的重要部分：

resource 资源的声明

声明你自己的 `$resource` 非常简单，只要调用注入的 `$resource` 函数,并给他传入正确的参数即可。(你现在应该已经知道如何注入依赖,对吧?)

`$resource` 函数只有一个必须参数,就是提供后台资源数据的 URL 地址,另外还有两个可选参数:默认 `request` 参数信息和其它的想在资源上要配置的动作。

请注意：第一个 URL 地址参数中的的变量数据是参数化可配置的(:冒号是参数变量的语法符号,比如 `:userId` 以为这个参数将会被实际的 `userId` 参数变量取代(译者注:写过参数化 SQL 语句的人应该很熟悉),而 `:cardId` 将会被 `cardId` 参数变量的值所取代),如果没有给函数传递这些参数变量,那那个位置将会被空字符取代。

函数的第二个参数则负责提供所有请求的默认参数变量信息.在这个案例中，我们给 `userId` 参数传递一个常量:123,cardId 参数则更有意思,我们给 `cardId` 参数传递了

一个"@id"字符串.这意味着如果我们使用一个从服务器返回的对象而且我们可以调用这个对象的任何方法(比如\$save),那么这个对象的 id 属性将会被取出来赋给 cardId 字段.

函数的第三个参数是一些我们想要暴露的其它方法,这些方法是对定制资源做操作的方法.在下面的章节,我们将会深度讨论这个话题

定制方法

\$resource 函数的第三个参数是可选的,主要用来传递要在 resource 资源上暴露的其它自定义方法。

在这个案例中,我们自定义了一个方法 charge.这个自定义方法可以通过传递一个对象而被配置上.这个对象里有个键值,表明了此方法的暴露名称.这个配置需要顶一个 request 请求的方法类型(GET,POST 等等),以及该请求中需要的参数也要被传递(比如 charge=true),并且声明返回对象是数组还是单个普通对象。这一切到搞定之后,你就可以在有这个业务实际需求的时候,自由地调用

CreditCard.charge()方法.

不要使用回调函数机制!(除非你真的需要它们)

第三个需要注意的事情是资源调用的返回类型.让我们再次关注一下

CreditCard.query() 这个函数.你将会看到不是在回调函数中给 cards 赋值,而是直接把它赋给 card 变量.在异步服务器请求的情况下唉,这样的代码是如何运作的哪?

你担心代码是否正常工作是对的,但是代码实际上是可以正常工作的.这里实际发生的是 AngularJS 赋值了一个引用(是普通对象的还是数组的取决于你期望的返回类型),这个引用将会在未来服务器请求响应返回时被填充.在这期间,这个引用是个空应用.

因为在 AngularJS 应用中的大多数通用过程都是从服务器端取数据,把它赋给一个变量,然后在模版上显示它,而上面这样的简化机制非常优雅.在你的控制器代码中,你所有需要去做的就是发出服务器端请求,把返回值赋给正确的作用域(scope)变量.然后剩下的合适渲染这些数据就由模板系统去操心了.

如果你想对返回值做一些业务逻辑处理,拿着汇总方法就不能奏效了.在这种情况下,你就得依赖回调函数机制了,比如在 Credit.get()调用中使用的那种机制.

简化的服务器端操作

无论你使用资源简化函数机制还是回调函数，关于返回对象都有几点问题需要注意。

返回的对象不是一个普通 JS 对象，实际上，他是“**resource**”类型的对象.这就意味着对象里除了包含服务器返回的数据以外,还有一些附加的行为函数(在这个案例中如**\$save()**和**\$charge** 函数).这样我们就可以很方便的执行服务器端操作,比如取数据、修改数据并把修改在服务器端持久化保存下来(其实也就是一般 **CURD** 应用里面的通用操作).

对 **ngResource** 做单元测试

ngResource 依赖项是一个封装,它以 **Angular** 核心服务 **\$http** 为基础.因此，你可能已经知道如何对它做单元测试了.它和我们看到的对 **\$http** 做单元测试的样例比起来基本没什么真正的变化.你只需要知道最终的服务器端请求应该由 **resource** 发起,告诉模拟 **\$http** 服务关于请求的信息.其他的基本都一样.下面我们来看看如何本节测试前面的代码:

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

这个测试看上去和简单的\$http单元测试非常相似,除了一些细微区别.注意在我们的expect语句里面,取代了简单的"equals"方法,哦我们用的是特殊的"toEqualData"方法.这种expect语句会智能地省略ngResource添加到对象上的附加方法.

\$q和预期值(Promise)

目前为止,我们已经看到了AngularJS是如何实现它的异步延迟API机制.

预期值建议(Promise propoal)是AngularJS构建异步延迟API的底层基础.作为底层机制,预期值建议(Promise propoal)为异步请求做了下面这些事:

- 异步请求返回的是一个预期(promise)而不是一个具体数据值.
- 预期值有一个then函数,这个函数有两个参数,一个参数函数响应"resolved"或者"sucess"事件,另外一个参数函数响应"rejected"或者"failure"事件.这些函数以一个结果参数调用,或者以一个拒绝原因参数调用.
- 确保当结果返回的时候,两个参数函数中有一个将会被调用

大多数的延迟机制和Q(详见\$q API 文档)是以上面这种方法实现的,AngularJS为什么这样实现具体是因为以下原因:

- \$q 对于整个AngularJS是可见的,因此它被集成到作用域数据模型里面.这样返回数据就能快速传递,UI上的闪烁更新也就更少.
- AngularJS模板也能识别\$q预期值,因为预期值可以被当作结果值一样对待,而不是把它仅仅当作结果的预期.这种预期值会在响应返回时被通知提醒.
- 更小的覆盖范围:AngularJS仅仅实现那些基本的、对于公共异步任务的需求来说最重要的延迟函数机制.

你也许会问这样的问题:为什么我们会做如此疯狂激进的实现机制?让我们先看一个在异步函数使用方面的标准问题:

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



```
});
```

上面这个代码就是人们使用 **JavaScript** 时经常抱怨的令人恐惧的深层嵌套缩进椎体的噩梦.返回值异步本质与实际问题的同步需求之间产生矛盾:导致多级函数包含关系,在这种情况下要想准确跟踪里面某句代码的执行上下文环境就很难.

另外,这种情况对错误处理也有很大影响.错误处理的最好方法是什么?在每次都做错误处理?那代码结构就会非常乱.

为了解决上面这些问题,预期值建议(Promise proposal)机制提供了一个 **then** 函数的概念,这个函数会在响应成功返回的时候调用相关的函数去执行,另一方面,当产生错误的时候也会干相同的事,这样整个代码就有嵌套结构变为链式结构.所以之前那个例子用预期值 **API** 机制(至少在 **AngularJS** 中已经被实现的)改造一下,代码结构会平整许多:

```
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
    });
```

那个完全的横椎体代码一下子被优雅地平整了,而且提供了链式的作用域,以及一个单点的错误处理.你在你自己的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.这种机制可以帮我做一些很酷的事情:比如响应拦截.

响应拦截处理

我们的讲解已经覆盖了怎样调用服务器端服务、怎样处理响应、怎样把响应优雅地抽象化封装、怎样处理异步回调.但是在真实世界的 **Web** 应用中，你最终还不得不为每个服务器端请求调用做一些通用的处理操作，比如错误处理、权限认证、以及其它考虑到安全问题的处理操作，比如对响应数据做裁剪处理(译注:有的 **Ajax** 响应为了安全需要，会添加一定约定好的噪声数据).

有着现在已经对**\$q API** 的深入理解,我们目前就可以利用响应拦截器机制来做上面所有提出过的功能.响应拦截器(正如其名)可以在响应数据被应用使用之前拦截他它,并且对它做数据转换处理,比如错误处理以及其它任何处理，包括厨房洗碗槽.(估计是指数据清洗)

让我们看一个代码例子，这个例子中的代码拦截响应，并对响应数据做了轻微的数据转换.

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

// Ensure that the interceptor we created is part of the interceptor chain
$httpProvider.responseInterceptors.push('myInterceptor');
```

安全方面的考虑

目前我们开发 **Web** 应用的时候，安全是一个非常重要的关注点，在我们的考虑维度直中，它必须作为首位被考虑.**AngularJS** 给我们提供了一些帮助，同时也带来了两个安全攻击的角度，下面这一节我们将会讲解这些内容.

JSON 的安全脆弱性

当我们对服务器发送一个请求 JSON 数组数据的 GET 请求时(特别是当这些数据是敏感数据且需要登录验证或读取授权时),就会有一个不易察觉的 JSON 安全漏洞被暴露出来.

当我们使用一个恶意站点时, 站点可能会用<script>标签发起同样的请求而得到相同的信息.因为你仍旧是登录状态, 恶意站点利用了你的验证信息而请求了 JSON 数据, 并且得到了它.

你或许惊奇是如何做到的, 因为信息仍旧在你客户端, 服务器也得不到这个数组信息的引用.并且通常作为请求脚本返回响应 JSO 对象会导致一个执行错误, 虽然数组是个例外.

但是漏洞真正的切入点是: 在 JavaScript 里, 你是可以对内建对象做重定义的.在这个漏洞里面, 数组的构造函数可以被重定义, 通过这种重定义, 恶意站点脚本就可以得到对响应数据的引用, 然后就可以把响应数据发回它自己的服务器喽.

有两种方法可以防止这个漏洞:一是通常要确保敏感数据信息只作为 POST 请求的响应被返回, 二是返回一个不合法的 JSON 表达式,然后客户端用约定好的逻辑把不合法数据转换为可用的真实数据.

AngularJS 中你可以两种方法都用来阻止这个漏洞.在你的应用中, 你可以而且应该选择敏感 JSON 信息只通过 POST 请求来获取.

进一步, 你可以在服务器端给 JSON 响应数据配置一个前缀字符串:

```
"})}}`,\n"
```

那么一个正常 JSON 响应比如:

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

通过前缀字符串设置, 这个 JSON 响应就会变为

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

AngularJS 将会自动的把前缀字符串过滤掉,然后仅仅处理真实 JSON 数据.

跨站请求伪造(XSRF)

跨站请求伪造攻击主要有以下特征:

- 它们影响的站点通常依赖于授权或者用户认证.
- 它们往往利用漏洞站点保存登录或者授权信息这个事实.
- 它们发起以假乱真的 HTTP 或者 XMLHttpRequest 请求来制造副作用, 这种副作用通常是有害的.

考虑依稀下面这个跨站请求伪造攻击的案例:

- 用户 A 登录进他的银行帐号(<http://www.examplebank.com/>)
- 用户 B 意识到这点, 然后诱导用户 A 访问用户 B 的个人主页
- 主页上有一个特殊手工生成的图片连接地址, 这个图片的指向地址将会导致一次跨站请求伪造攻击, 比如如下代码: `<img`

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

如果用户 A 的银行站点把授权信息保存在 cookie 里, 且 Cookie 还没过期. 当用户 A 打开用户 B 的站点时, 就会导致非授权的用户 A 给用户 B 转账行为.

那么 AngularJS 是怎么帮助你防止这种事情发生? 它提供一种双步机制来防止跨站请求伪造攻击.

在客户端, 当发起 XHR 异步请求时, \$http 服务会从一个叫 XSRF-TOKEN 的 cookie 中读取令牌值, 然后把它设置成 X-XSRF-TOKEN 头信息的值, 因为只有你自己域的请求才能读取和设置这个令牌, 你可以保证 XHR 请求只来自你自己的域.

同时, 服务器端代码也需要一点轻微的修改, 以便于你收到你的第一个 HTTP GET 请求时就设置一个可读取的对话 Cookie, 这个对话 Cookie 键叫 XSRF-TOKEN. 后续客户端发往服务器的请求就可以通过对比请求头信息的令牌值和之前第一个请求设置的 Cookie 令牌值来达到验证的目的. 当然, 令牌必须是一个用户一个唯一的令牌值. 这个令牌值必须在服务器端验证(以防止恶意脚本捏造假令牌).

第六章 指令

对于指令, 你可以扩展 HTML 来以添加声明性语法来做任何你喜欢做的事情. 通过这样做, 你可以替换一些特定于你的应用程序的通用的 `<div>s` 和 `s` 元素和属

性的实际意义。它们都带有 **Angular** 提供的基础功能，但是你可以创建特定于应用程序的你自己想做的事情。

首先我们要复习以下指令 **API** 以及它在 **Angular** 启动和运行生命周期里是如何运作的。从那里，我们将使用这些只是来创建一个指令类型。在本将完成时我们将学习到如何编写指令的单元测试和使它们运行得更快。

但是首先，我们来看看一些使用指令的语法说明。

目录

- [指令和 HTML 验证](#)
- [API 预览](#)
 - [为你的指令命名](#)
 - [指令定义对象](#)
 - [编译和链接功能](#)
 - [作用域](#)
 - [操作 DOM 元素](#)
 - [控制器](#)
- [小结](#)

指令和 HTML 验证

在本书中，我们已经使用了 **Angular** 内置指令的 `ng-directive-name` 语法。例如 `ng-repeat`，`ng-view` 和 `ng-controller`。这里，`ng` 部分是 **Angular** 的命名空间，并且 `dash` 之后的部分便是指令的名称。

虽然我们喜欢这个方便输入的语法，但是在大部分的 **HTML** 验证机制中它不是有效的。为了支持这些，**Angular** 指令允许你以几种方式调用任意的指令。以下在表 6-1 中列出的语法，都是等价的并能够让你偏爱的[首选的]验证器正常工作

Table 6-1 HTML Validation Schemes

```
<table>

  <thead>

    <tr>

      <th>Validator</th>
```

```
        <th>Format</th>

        <th>Example</th>

    </tr>
</thead>
<tbody>

    <tr>

        <td>none</td>

        <td>namespace-name</td>

        <td>ng-repeat=<i>item in items</i></td>

    </tr>

    <tr>

        <td>XML</td>

        <td>namespace:name</td>

        <td>ng:repeat=<i>item in items</i></td>

    </tr>

    <tr>

        <td>HTML5</td>

        <td>data-namespace-name</td>

        <td>data-ng-repeat=<i>item in items</i></td>

    </tr>

    <tr>

        <td>xHTML</td>

        <td>x-namespace-name</td>

        <td>x-ng-repeat=<i>item in items</i></td>

    </tr>

</tbody>
```

</table>

由于你可以使用任意的这些形式, [AngularJS 文档](#)中列出了一个驼峰式的指令, 而不是任何这些选项. 例如, 在 `ngRepeat` 标题下你可以找到 `ng-repeat`. 稍后你会看到, 在你定义你自己的指令时你将会使用这种命名格式.

如果你不适用 HTML 验证器(大多数人都不使用), 你可以很好的使用在目前你所见过的例子中的命名空间-指令[namespace-directive]语法

API 预览

下面是一个创建任意指令伪代码模板

```
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;

});

```

有些选项是互相排斥的，它们大多数都是可选的，并且它们都有有价值的详细说明：当你使用每个选项时，表 6-2 提供了一个概述。

Table 6-2 指令定义选项

<table>	
<thead>	
<tr>	
<th>Property</th>	
<th>Purpose</th>	
</tr>	
</thead>	
<tbody>	
<tr>	
<td>restrict</td>	
<td>声明指令可以作为一个元素，属性，类，注释或者任意的组合如何用于模板中</td>	
</tr>	
<tr>	

`<td>priority</td>`

`<td>设置模板中相对于其他元素上指令的执行顺序</td>`

`</tr>`

`<tr>`

`<td>template</td>`

`<td>`指令一个作为字符串的内联模板。如果你指定一个模板 URL 就不要使用这个模板属性。`</td>`

`</tr>`

`<tr>`

`<td>templateUrl</td>`

`<td>`指定通过 URL 加载的模板。如果你指定了字符串的内联模板就不需要使用这个。`</td>`

`</tr>`

`<tr>`

`<td>replace</td>`

`<td>`如果为 **true**，则替换当前元素。如果为 **false** 或者未指定，则将这个指令追加到当前元素上。`</td>`

`</tr>`

`<tr>`

`<td>transclude</td>`

`<td>`让你将一个指令的原始节点移动到心模板位置内。`</td>`

`</tr>`

`<tr>`

`<td>scope</td>`

`<td>`为这个指令创建一个新的作用域而不是继承父作用域。`</td>`

`</tr>`

`<tr>`

<code><td>controller</td></code>	<code><td>为跨指令通信创建一个发布的 API.</td></code>
<code></tr></code>	
<code><tr></code>	<code><td>require</td></code>
<code><td>需要其他指令服务于这个指令来正确的发挥作用.</td></code>	
<code></tr></code>	
<code><tr></code>	<code><td>link</td></code>
<code><td>以编程的方式修改生成的 DOM 元素实例，添加事件监听器，设置数据绑定.</td></code>	
<code></tr></code>	
<code><tr></code>	<code><td>compile</td></code>
<code><td>以编程的方式修改一个指令的 DOM 模板的副本特性，如同使用`ng-repeat`时，你的编译函数也可以返回链接函数来修改生成元素的实例.</td></code>	
<code></tr></code>	
<code></tbody></code>	
<code></table></code>	

下面让我们深入细节来看看.

为你的指令命名

你可以用模块的指令函数为你的指令创建一个名称，如下所示：

```
myModule.directive('directiveName', function
factory(injectables){...});
```

虽然你可以使用任何你喜欢的名字命名你的指令，该符号会选择了一个前缀命名空间标识你的指令，同时避免与可能包含在你的项目中的外部指令冲突。

你当然不希望它们使用一个 `ng-` 前缀，因为这可能与 Angular 自带的指令相冲突。如果你从事于 SuperDuper MegaCorp，你可以选择一个 `super-`，`superduper-`，或者甚至是 `superduper-megacorp-`，虽然你可能选择第一个选项，只是为了方便输入。

正如前面所描述的，Angular 使用一个标准化的指令命名机制，并且试图有效的在模板中使用驼峰式的指令命名方式来确保在 5 个不同的友好的验证器中正常工作。例如，如果你已经选择了 `super-` 作为你的前缀，并且你在编写一个日期选择 (`datepicker`) 组件，你可能将它命名为 `superDatePicker`。在模板中，你可以像这样来使用它： `super-date-picker`，`super:date-picker`，`data-super-date-picker` 或者其他多样的形式。

指令定义对象

正如前面提到的，在指令定义中大多数的选项都是可选的。实际上，这里并没有硬性的要求必须选择哪些选项，并且你可以构造出许多有利于指令的子集参数。让我们来逐步讨论这些选项是做什么的。

restrict

`restrict` 属性允许你指定你的指令声明风格--也就是说，它是否能够用于作为元素名称，属性，类 [`className`]，或者注释。你可以根据表 6-3 来指定一个或多个声明风格，只需要使用一个字符串来表示其中的每一中风格：

Table 6-3 指令声明用法选项

```
<table>

  <thead>

    <tr>

      <th>Character</th>

      <th>Declaration style</th>

      <th>Example</th>

    </tr>

  </thead>

  <tbody>
```

```

<tr>

  <td>E</td>

  <td>element</td>

  <td>&lt;my-menu title=<i>Products</i>&gt;&lt;/my-menu&gt;</td>

</tr>

<tr>

  <td>A</td>

  <td>attribute</td>

  <td>&lt;div my-menu=<i>Products</i>&gt;&lt;/div&gt;</td>

</tr>

<tr>

  <td>C</td>

  <td>class</td>

  <td>&lt;div class=my-menu:<i>Products</i>&gt;&lt;/div&gt;</td>

</tr>

<tr>

  <td>M</td>

  <td>comment</td>

  <td>&lt;!--directive:my-menu Products--&gt;</td>

</tr>

</tbody>

</table>

```

如果你希望你的指令用作一个元素或者一个属性，那么你应该传递 `EA` 作为 `restrict` 字符串。

如果你忽略了 `restrict` 属性，则默认为 `A`，并且你的指令只能用作一个属性(属性指令)。

如果你计划支持 IE8, 那么基于 `attribute-`和 `class-`的指令就是你最好的选择, 因为它需要额外的努力来使新元素正常工作. 可以查看 [Angular](#) 文档来详细了解这一点.

Priorities

在你有多个指令绑定在一个单独的 DOM 元素并要确定它们的应用顺序的情况下, 你可以使用 `priority` 属性来指定应用的顺序. 数值高的首先运行. 如果你没有指定, 则默认的 `priority` 为 0.

很难发生需要设置优先级的情况. 一个需要设置优先级例子是 `ng-repeat` 指令. 当重复元素时, 我们希望 **Angular** 在应用指令之前床在一个模板元素的副本. 如果不这么做, 其他的指令将会应用到标准的模板元素上而不是我们所希望在应用程序中重复我们的元素.

虽然它(`priority`)不在文档中, 但是你可以搜索 **Angular** 资源中少数几个使用 `priority` 的其他指令. 对于 `ng-repeat`, 我们使用优先级值为 1000, 这样就有足够的优先级处理优先处理它.

Templates

当创建组件, 挂件, 控制器一起其他东西时, **Angular** 允许你提供一个模板替换或者包裹元素的内容. 例如, 如果你在视图中创建一组 `tab` 选项卡, 可能会呈现出如图 6-1 所示视图.

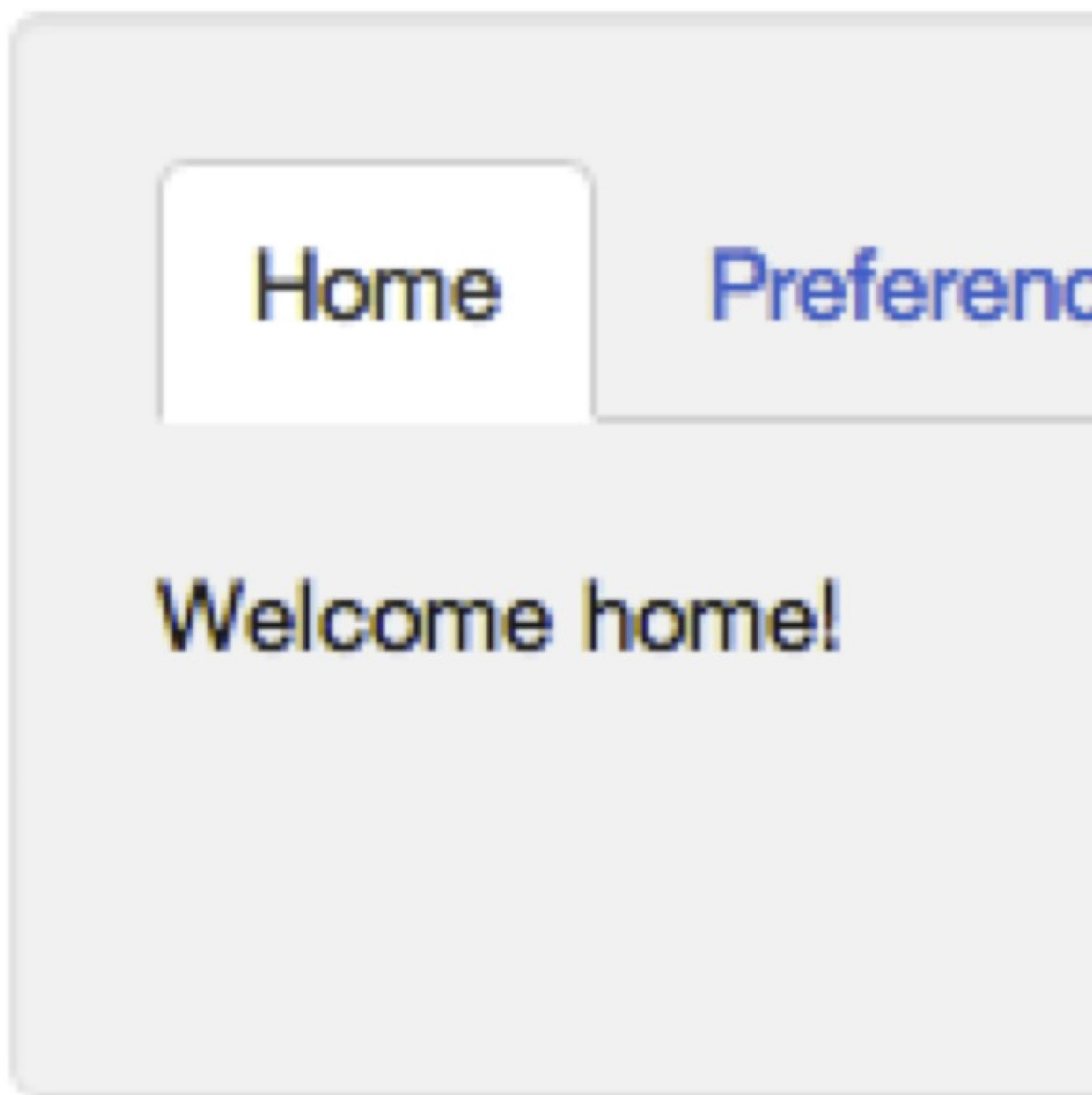


图 6-1 tab 选项卡视图

并不是一堆<div>, 和<a>元素, 你可以创建一个<tab-set>和<tab>指令, 用来声明每个单独的 tab 选项卡的结构. 然后你的 HTML 可以做的更好来表达你的模板意图. 最终结果可能看起来像这样:

```
<tab-set>  
  
  <tab title="Home">
```

```

        <p>Welcome home!</p>

    </tab>

    <tab title="Preferences">

        <!-- preferences UI goes here -->

    </tab>

</tab-set>

```

你还可以给 **title** 绑定一个字符串数据，通过在 **<tab>** 或者 **<tab-set>** 上绑定控制器处理 **tab** 选项内容。它不仅限于用在 **tabs** 上--你还可以用于菜单，手风琴，弹窗，**dialog** 对话框或者其他任何你希望以这种方式实现的地方。

你可以通过 **template** 或者 **templateUrl** 属性来指定替换的 DOM 元素。使用 **template** 通过字符串来设置模板内容，或者使用 **templateUrl** 来从服务器的一个文件上来加载模板。正如你在接下来的例子中会看到，你可以预先缓存这些模板来减少 GET 请求，这有利于提高应用的性能。

让我们来编写一个 **dumb** 指令：一个 **<hello>** 元素，只是用于使用 **<div>Hi there</div>** 来替换自身。在这里，我们将设置 **restrict** 来允许元素和设置 **template** 显示我们所希望的东西。由于默认的行为只将内容追加到元素中，因此我们将设置 **replace** 属性为 **true** 来替换原来的模板：

```

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

appModule.directive('hello', function(){

    return {

        restrict: 'E',

        template: '<div>Hi there</div>',

        replace: true

    };

});

```

在页面中我们可以像这样使用它：

```

<html lang="en" ng-app="app">

```

```
...  
  
<body>  
  
    <hello></hello>  
  
</body>  
  
...
```

将它载入到浏览器中，我们会看到"Hi there".

如果你查看页面的源代码，在页面上你仍然会看到<hello></hello>，但是如果你查看生成的源代码(在 Chrome 中，你可以在"Hi there"上右击然后选择审查元素)，你会看到：

```
<body>  
  
    <div>Hi there</div>  
  
</body>
```

<hello></hello>被模板中的<div>替换了.

如果你从指令定义中移除 `replace: true`，那么你会看到<hello><div>Hi there</div></hello>.

通常你会希望使用 `templateUrl` 而不是 `template`，因为输入 HTML 字符串并不是那么有趣. `template` 属性通常有利于非常小的模板. 使用 `templateUrl` 同样非常有用，可以设置适当的头来使模板可缓存. 我们可以像下面这样重写我们的 `hello`` 指令：

```
var appModule = angular.module('app', []);  
  
appModule.directive('hello', function(){  
  
    return {  
  
        restrict: 'E',  
  
        templateUrl: 'helloTemplate.html',  
  
        replace: true  
  
    };  
  
});
```


在 `helloTemplate.html` 中, 你只需要输入:

```
<div>Hi there</div>
```

如果你使用 **Chrome** 浏览器, 它的"同源策略"会组织 **Chrome** 从 `file://` 中加载这些模板, 并且你会得到一个类似"Origin null is not allowed by Access-Control-Allow-Origin."的错误. 那么在这里, 你有两个选择:

- 通过服务器来加载应用
- 在 **Chrome** 中设置一个标志. 你可以通过在命令行中使用 `chrome --allow-file-access-from-files` 命令来运行 **Chrome** 做到这一点.

这将会通过 `templateUrl` 加载这些文件, 然而, 这会让你的用户要等待到指令加载. 如果你希望在首页加载模板, 你可以在一个 `script` 标签中将它作为这个页面的一部分包含进来, 就像这样:

```
<script type="text/ng-template" id="helloTemplateInline.html">

    <div>Hi there</div>

</script>
```

这里的 `id` 属性很重要, 因为这是 **Angular** 用来存储模板的 **URL** 键. 稍候你将会使用这个 `id` 在指令的 `templateUrl` 中指定要插入的模板.

这个版本能够很好的载入而不需要服务器, 因为没有必要的 `XMLHttpRequest` 来获取内容.

最后, 你可以越过 `$http` 或者以其他机制来加载你自己的模板, 然后将它们直接设置在 **Angular** 中称为 `$templateCache` 的对象上. 我们希望在指令运行之前缓存中的这个模板可用, 因此我们将通过 `module` 上的 `run` 函数来调用它.

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

你可能希望在产品中这么做，仅仅作为一个减少所需的 **GET** 请求数量的技术。你可以运行一个脚本将所有的模板合并到一个单独的文件中，并在一个新的模块中加载它，然后你就可以从你的主应用程序模块中引用它。

Transclusion

除了替换或者追加内容，你还可以通过 `transclude` 属性将原来的内容移到新模板中。当设置为 **true** 时，指令将删除原来的内容，但是在你的模板中通过一个名为 `ng-transclude` 的指令重新插入来使它可用。

我们可以使用 **transclusion** 来改变我们的示例：

```
appModule.directive('hello', function() {  
  
    return {  
  
        template: '<div>Hi there <span ng-transclude></span></div>',  
  
        transclude: true  
  
    };  
  
});
```

像这样来应用它：

```
<div hello>Bob</div>
```

你会看到: "Hi there Bob."

编译和链接功能

虽然插入模板是有用的, 任何指令真正有趣的工作发生在它的 `compile` 和它的 `link` 函数中.

`compile` 和 `link` 函数被指定为 **Angular** 用来创建应用程序实际视图的后两个阶段. 让我们从更高层次来看看 **Angular** 的初始化过程, 按一定的顺序:

Script loads

Angular 加载和查找 `ng-app` 指令来判定应用程序界限.

Compile phase(阶段)

在这个阶段, **Angular** 会遍历 DOM 节点以确定所有注册在模板中的指令. 对于每一个指令, 然后基于指令的规则(`template`, `replace`, `transclude` 等等)转换 DOM, 并且如果它存在就调用 `compile` 函数. 它的返回结果是一个编译过的 `template` 函数, 这将从所有的指令中调用 `link` 函数来收集.

Link phase(阶段)

创建动态的视图, 然后 **Angular** 会对每个指令运行一个 `link` 函数. `link` 函数通常在 DOM 或者模型上创建监听器. 这些监听器用于视图和模型在所有的时间里都保持同步.

因此我们必须在编译阶段处理模板的转换, 同时在链接阶段处理在视图中修改数据. 按照这个思路, 指令中的 `compile` 和 `link` 函数之间主要的区别是 `compile` 函数处理模板自身的转换, 而 `link` 函数处理在模型和视图之间创建一个动态的连接. 作用域挂接到编译过的 `link` 函数正是在这个第二阶段, 并且通过数据绑定将指令变成活动的.

出于性能的考虑, 者两个阶段才分开的. `compile` 函数仅在编译阶段执行一次, 而 `link` 函数会被执行多次, 对每个指令实例. 例如, 让我们说说你上面使用的 `ng-repeat` 指令. 你并不想小勇 `compile`, 这回导致在每次 `ng-repeat` 重复时都产生一个 DOM 遍历的操作. 相反, 你会希望一次编译, 然后链接.

虽然你毫无疑问的应该学习编译和链接之间的不同, 以及每个功能, 你需要编写的大部分的指令都不需要转换模板; 你还会编写大部分的链接函数.

让我们再看看每个语法来比较一下, 我们有:

```
compile: function compile(tElement, tAttrs, transclude) {  
  
    return {
```

```

    pre: function preLink(scope, iElement, iAttrs, controller)
    {...},

    post: function postLink(scope, iElement, iAttrs, controller)
    {...}

  }

}

```

以及链接:

```
link: function postLink(scope, iElement, iAttrs) {...}
```

注意这里有一点不同的是 `link` 函数获得了一个作用域的访问, 而 `compile` 没有. 这是因为在编译阶段期间, 作用域并不存在. 然而你有能力从 `compile` 函数返回 `link` 函数. 这些 `link` 函数能够访问到作用域.

还要注意的是 `compile` 和 `link` 都会获得一个到它们对应的 DOM 元素和这些元素属性[attributes]列表的引用. 这里的一点区别是 `compile` 函数是从模板中获得模板元素和属性, 并且会获取到 `t` 前缀. 而 `link` 函数使用模板创建的视图实例中获得它们的, 它们会获取到 `i` 前缀.

这种区别只存在于当指令位于其他指令中制造模板副本的时候. `ng-repeat` 就是一个很好的例子.

```

<div ng-repeat="thing in things">

  <my-widget config="thing"></my-widget>

</div>

```

这里, `compile` 函数将只被调用一次, 而 `link` 函数在每次复制 `my-widget` 时都会被调用一次--等价于元素在 `things` 中的数量. 因此, 如果 `my-widget` 需要到所有 `my-widget` 副本(实例)中修改一些公共的东西, 为了提升效率, 正确的做法是在 `compile` 函数中处理.

你可能还会注意到 `compile` 函数好哦的了一个 `transclude` 属性函数. 这里, 你还有机会以编写一个函数以编程的方式 `transcludes` 内容, 对于简单的的基于模板不足以 `transclusion` 的情况.

最后, `compile` 可以返回一个 `preLink` 和 `postLink` 函数, 而 `link` 仅仅指向一个 `postLink` 函数. `preLink`, 正如它的名字所暗示的, 它运行在编译阶段之后, 但是会在指令链接到子元素之前. 同样的, `postLink` 会运行在所有的子元素指令被链接之后.

这意味着如果你需要改变 DOM 结构，你将在 `postLink` 中处理。在 `preLink` 中处理将会混淆流程并导致一个错误。

作用域

你会经常希望从指令中访问作用域来监控模型的值并在它们改变时更新 UI，同时在外部的时间造成模型改变时通知 Angular。前者是最常见的，当你从 jQuery, Closure 或者其他库中包裹一些非 Angular 组件或者实现简单的 DOM 事件时。然后将 Angular 表达式作为属性传递到你的指令中来执行。

这也是你期望使用一个作用域的原因之一，你可以获得三种类型的作用域选项：

1. 从指令的 DOM 元素中获得**现有的作用域**。
2. 创建一个**新作用域**，它继承自你闭合的控制器作用域。这里，你见过能够访问树上层作用域中的所有值。这个作用域将会请求这种作用域与你 DOM 元素中其他任意指令共享它并被用于与它们通信。
3. 从它的父层**隔离出来的作用域**不带有模型属性。当你在创建可重用的组件而需要从父作用域中隔离指令操作时，你将会希望使用这个选项。

你可以使用下面的语法来创建这些作用域类型的配置：

```
<table>

  <thead>

    <tr>

      <th>Scope Type</th>

      <th>Syntax</th>

    </tr>

  </thead>

  <tbody>

    <tr>

      <td>existing scope</td>

      <td>scope: false(如果不指定将使用这个默认值)</td>

    </tr>
```

```

        <tr>

            <td>new scope</td>

            <td>scope: true</td>

        </tr>

        <tr>

            <td>isolate scope</td>

            <td>scope: { /* attribute names and binding style */ }</td>

        </tr>

    </tbody>

</table>

```

当你创建一个隔离的作用域时，默认情况下你不需要访问父作用域中模型中的任何东西。然而，你也可以指定你想要的特定属性传递到你的指令中。你可以认为是吧这些属性名作为参数传递给函数的。

注意，虽然隔离的作用域不成为模型属性，但它们仍然是其副作用域的成员。就像所有其他作用域一样，它们都有一个`$parent`属性引用到它们的父级。

你可以通过传递一个指令属性名的映射的方式从父作用域传递特定的属性到隔离的作用域中。这里有三种合适的方式从父作用域中传递数据。我们称这些传递数据不同的方式为“绑定策略”。你也可以可选的指定一个局部别名给属性名称。

以下是没有别名的语法：

```

scope: {

    attributeName1: 'BINDING_STRATEGY',

    attributeName2: 'BINDING_STRATEGY',...

}

```

以下是使用别名的方式：

```

scope: {

    attributeAlias: 'BINDING_STRATEGY' + 'templateAttributeName',...

}

```

```
}
```

绑定策略被定义为表 6-4 中的符号：

表 6-4 绑定策略

Symbol	Meaning
@	<code><td>@</td></code> <code><td></code> 将属性作为字符串传递。你也可以通过在属性值中使用插值符号 <code>{{}}</code> 来从闭合的作用域中绑定数据值。 <code></td></code>
=	<code><td>=</td></code> <code><td></code> 使用你的指令的副作用域中的一个属性绑定数据到属性中。 <code></td></code>
&	<code><td>&</td></code> <code><td></code> 从父作用域中传递到一个函数中，以后调用。 <code></td></code>

```
</table>
```

这些都是相当抽象的概念，因此让我们来看一个具体的例子上的变化来进行说明。
比方说我们希望创建一个 `expander` 指令在标题栏被点击时显示额外的内容。

收缩时它看起来如图 6-2 所示。

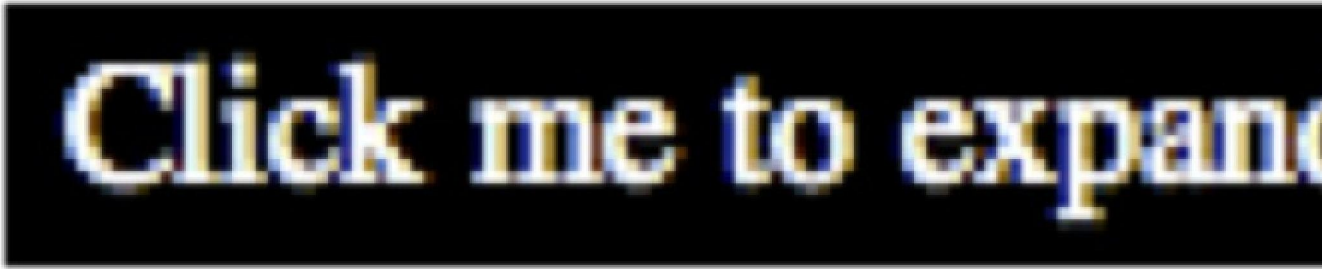


图 6-2 Expander in closed state

展开时它看起来如图 6-3 所示。

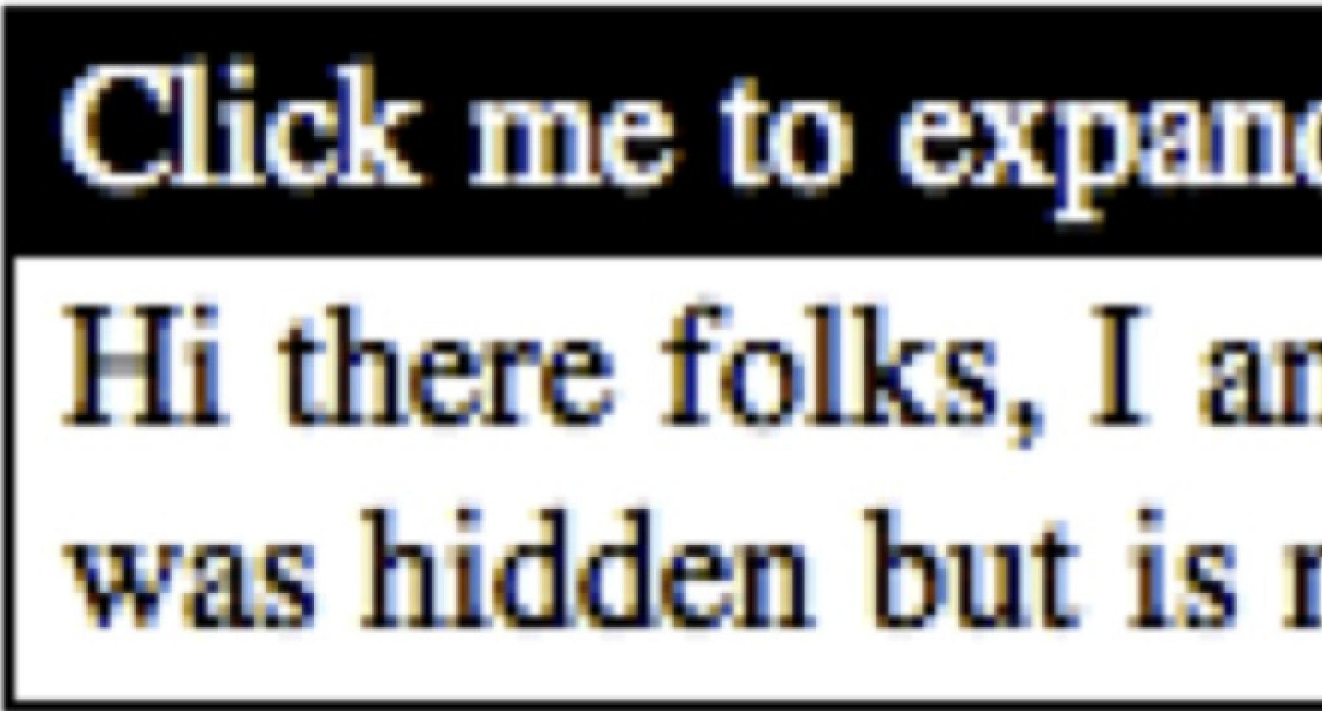


图 6-3 Expander in open state

我们会编写如下代码：


```

<div ng-controller="SomeController">

    <expander class="expander" expander-title="title">

        {{text}}

    </expander>

</div>

```

标题(Clicked me to expand)和文本(Hi there folks...)的值来自于闭合的作用域中. 我们可以像下面这样来设置一个控制器:

```

function SomeController($scope) {

    $scope.title = 'Clicked me to expand';

    $scope.text = 'Hi there folks, I am the content that was hidden but
is now shown.';

}

```

然后我们可以来编写指令:

```

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, attris){

            scope.showMe = false;

            scope.toggle = function toggle(){

                scope.showMe = !scope.showMe;

            }

        }

    }

});

```

然后编写下面的样式:

```

.expander {

    border: 1px solid black;

    width: 250px;

}

.expander > .title {

    background-color: black;

    color: white;

    padding: .1em .3em;

    cursor: pointer;

}

.expander > .body {

    padding: .1em .3em;

}

```

接下来让我们来看看指令中的每个选项是做什么的, 在表 6-5 中.

表 6-5 Functions of elements

<table>

```

<thead>

  <tr>

    <th>FunctionName</th>

    <th>Description</th>

  </tr>

</thead>

<tbody>

  <tr>

    <td>restrict: EA</td>

    <td>一个元素或者属性都可以调用这个指令。也就是说，
\<expander ... \>... \</expander \>与\<div expander ... \>... \</div \>是等价</td>

  </tr>

  <tr>

    <td>replace:true</td>

    <td>使用我们提供的模板替换原始元素</td>

  </tr>

  <tr>

    <td>transclude:true</td>

    <td>将原始元素的内容移动到我们所提供的模板的另外一个位置.</td>

  </tr>

  <tr>

    <td>scope: {title: =expanderTitle}</td>

    <td>创建一个称为`title`的局部作用域，将父作用域的属性数据绑定到声明的
`expanderTitle`属性中。这里，我们重命名 title 为更方便的 expanderTitle。我们可以
编写`scope: { expanderTitle: '='}`，那么在模板中我们就要使用`expanderTitle`了。
但是在其他指令也有一个`title`属性的情况下，在 API 中消除 title 的歧义和只是重命名它
用于在局部使用是有意义的。请注意，这里自定义指令也使用了相同的驼峰式命名方式作为指
令名.</td>

```

```

</tr>

<tr>

<td>template: \<'div'\>+</td>

<td>声明这个指令要插入的模板。注意我们使用了`ng-click`和`ng-show`来
显示和隐藏自身并使用`ng-transclude`声明了原始内容会去哪里。还要注意的
是transcluded的内容能够访问父作用域，而不是指令闭合中的作用域.</td>

</tr>

<tr>

<td>link...</td>

<td>设置`showMe`模型来检测 expander 的展开/关闭状态，同时定义在用于点
击`title`这个div的时候调用定义的`toggle()`函数.</td>

</tr>

</tbody>

</table>

```

如果我们像使用更多有意义的东西来在模板中定义 `expander title` 而不是在模型中，我们还可以使用传递通过在作用域声明中使用@符号传递一个字符串风格的属性，就像下面这样：

```
scope: { title: '@expanderTitle'},
```

在模板中我们就可以实现相同的效果：

```

<expander class="expander" expander-title="Click mr to expand">

  {{text}}

</expander>

```

注意，对于@策略我们仍然可以通过使用插入法将 `title` 数据绑定到我们的控制器作用域中：

```

<expander class="expander" expander-title="{{title}}">

  {{text}}

```

</expander>

操作 DOM 元素

传递给指令的 `link` 和 `compile` 函数的 `iElement` 和 `tElement` 是包裹原生 DOM 元素的引用。如果你已经加载了 `jQuery` 库，你也可以使用你已经习惯使用的 `jQuery` 元素。

如果你没有使用 `jQuery`，你也可以使用 `Angular` 内置的被称为 `jqLite` 的包装器。它提供了一个 `jQuery` 的子集便于我们在 `Angular` 中创建任何东西。对于多数应用程序，你都可以单独使用这些 `API` 做任何你需要做的事情。

如果你需要直接访问原生的 DOM 元素你可以通过使用 `element[0]` 访问对象的第一个元素来获得它。

你可以在 `Angular` 文档的 `angular.element()` 查看它所支持的 `API` 的完整列表--你可以用这个函数创建你自己的 `jqLite` 包装的 DOM 元素。它包含像 `addClass()`、`bind()`、`find()`、`toggleClass()` 等等其他方法。其次，其中大多数有用的核心方法都来自于 `jQuery`，但是它的代码亮更少。

对于其他的 `jQuery` `API`，元素在 `Angular` 中都有指定的函数。这些都是存在的，无论你是否使用完整的 `jQuery` 库。

Table 6-6. Angular specific functions on an element

<table>	
<thead>	
<tr>	
<th>Function</th>	
<th>Description</th>	
</tr>	
</thead>	
<tbody>	
<tr>	
<td>controller(name)</td>	

`<td>`当你需要直接与控制器通信时，这个函数会返回附加到元素上的控制器。如果没有现有的元素，它会遍历 DOM 元素并查找最近的父控制器。`name` 参数是可选的，它是用于指定相同元素上其他指令名称的。如果提供这个参数，它会从相应的指令中返回控制器。这个名字应该与所有指令一样使用一个驼峰式的格式。也就是说，使用 ``ngModule`` 来替换 ``ng-model`` 的形式。`</td>`

`</tr>`

`<tr>`

`<td>injector()</td>`

`<td>`获取当前元素或者它的父元素的注入器。它还允许你访问在这些元素上定义的所依赖的模块。`</td>`

`</tr>`

`<tr>`

`<td>scope()</td>`

`<td>`返回当前元素或者它最近的父元素的作用域。`</td>`

`</tr>`

`<tr>`

`<td>inheritedData()</td>`

`<td>`正如 jQuery 的 ``data()`` 函数，``inheritedData()`` 会在一个封闭的方式中设置和获取数据。此外还能够从当前元素获取数据，它也会遍历 DOM 元素并查找值。`</td>`

`</tr>`

`</tbody>`

`</table>`

这里有一个例子，让我们重新定义之前的 `expander` 例子而不使用 `ng-show` 和 `ng-click`。它看起来像下面这样：

```
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');

            }

        }

    });

```

这里我们从模板中移除了 `ng-click` 和 `ng-show`。相反的时，当用户单击 **expander** 的 **title** 时执行所定义的行为，我们将从 **title** 元素创建一个 `jQuery` 元素，然后它绑定一个 `click` 事件并将 `toggle()` 函数作为它的回调函数。在 `toggle()` 函数中，我们在 **expander** 的 **body** 元素上调用 `toggleClass()` 来添加或者移除一个被称为 `closed` 的 `class`(HTML 类名)，这里我们给元素设置了一个值为 `display: none` 的类，像下面这样：

```
.closed {  
  
    display: none;  
  
}
```

控制器

当你有相互嵌套的指令需要相互通信时，你可以通过控制器做到这一点。比如一个 `<menu>` 可能需要知道它自身内部的 `<menu-item>` 元素它才能适当的显示或者隐藏它们。同样的对于一个 `<tab-set>` 也需要知道它的 `<tab>` 元素，或者一个 `<grid-view>` 要知道它的 `<grid-element>` 元素。

正如前面所展示的，创建一个 **API** 用于在指令之间沟通，你可以使用控制器属性的语法声明一个控制器作为一个指令的一部分：

```
controller: function controllerConstructor($scope, $element, $attrs, $transclude)
```

这个控制器函数就是依赖注入，因此这里列出的参数都是潜在的可用并且全部都是可选的--它们可以按照任意顺序列出。它们也仅仅只是可用服务的一个子集。

其他的指令也可以使用 `require` 属性语法将这个控制器传递给它们。完整的 `require` 的形式看起来像：

```
require: '^?directiveName'
```

关于 `require` 字符串参数的说明可以在表 6-7 中找到。

Table 6-7. Options for required controllers

Option	Usage
directiveName	这个指令驼峰式命名规范应该是来自于控制器。因此如果我们的 <code>ns</code> 指令需要在它自己的控制器，我们需要将它编写为 <code>`myMenu`</code> 。
^	默认情况下， Angular 会从同一元素的命名指令中获取控制器。加入可选的 <code>^</code> 符号来以查找指令。对于 <code>ns</code> 示例，我们需要添加这个符号；最终的字符就是 <code>`^myMenu`</code> 。

Option	Usage
?	如果你所需要的控制器没有找到, Angular 将抛出一个异常信息来告诉你遇到了什么。给字符串就是说这个控制器是可选的并且如果没有找到控制器它不应该抛出一个异常。但是如果我们希望让\不需要使用一个\容器, 我们可以将这个添加给最终所需。

例如, 让我们重写我们的 **expander** 指令用于一组称为"手风琴"的组件, 它可以确保当你打开一个 **expander** 时, 其他的都会自动关闭。它看起来如图 6-4 所示。

Click me to expand

Hi there folks, I am the content that was hidden but is now shown.

Click this

No, click me!

Click me to expand

Click this

I am even better text than you have seen previously

No, click me!

Click me to expand

Click this

No, click me!

I am text that should be seen before seeing other texts

图 6-4. Accordion component in multiple states

首先，让我们编写处理手风琴菜单的 **accordion** 指令。这里我们将添加我们的控制器构造器方法来处理手风琴：

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

这里我们定义了一个 `addExpander()` 函数给 **expanders** 便于调用它来注册自身实例。我们也创建了一个 `gotOpened()` 函数给 **expanders** 便于调用，因而让 **accordion** 的控制器可以知道它能够去关闭任何其他展开的 **expanders**。

在 **expander** 指令自身中，我们将从它的父元素扩展它所需要的 **accordion** 控制器并在适当的时间里调用 `addExpander()` 和 `gotOpened()` 方法。

```
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.toggleOpen(scope);
            }
        }
    });
});

```

注意在手风琴指令的控制器中我们创建了一个 **API**，通过它可以让 **expander** 可以相互通信。

然后我们可以编写一个模板来使用这些指令，最后生成的结果整如图 6-4 所示。

```

<body ng-controller="SomeController">
    <accordion>
        <expander class="expander" ng-repeat="expander in expanders" expander-
title="expander.title">
            {{expander.text}}
        </expander>
    </accordion>
</body>

```

当然接下是对应的控制器：

```

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 should be seen before seeing other texts'}
    ];
}

```

小结

正如我们所看到的，指令允许我们扩展 HTML 的语法并让很多应用程序按照我们声明的意思工作。指令使重用(代码重用/组件复用)变得轻而易举--从使用 `ng-model` 和 `ng-controller` 配置你的应用程序，到处理模板的任务的像 `ng-repeat` 和 `ng-view` 指令，再到前几年被限制的可复用的组件像数据栅格，饼图，工具提示和选项卡等等。

第七章 其他关注点

在这一章中，我们将看一切目前 Angular 所实现的其他有用的特性，但是我们不会涵盖所有的或者深入的章节和例子。

目录

- [\\$location](#)
- [HTML5 模式和 Hashbang 模式](#)
- [AngularJS 模块方法](#)
 - [主方法在哪？](#)
 - [加载和依赖](#)
 - [快捷方法](#)
- [\\$on, \\$emit 和 \\$broadcast 之间的作用域通信](#)
- [Cookies](#)
- [国际化和本地化](#)
 - [在 AngularJS 中我能做什么？](#)
 - [如何获取所有工作？](#)
 - [常见问题](#)
- [净化 HTML 和模块](#)
- [Linky](#)

\$location

到现在为止，你已经看到了不少使用 AngularJS 中的 `$location` 服务的例子。它们大多数都只是短暂的一撇--在这里访问，那里设置。在这一小节，我们将深入研究 AngularJS 中的 `$location` 服务时什么，以及什么时候你应该使用它，什么时候不应该使用它。

`$location` 服务是一个存在于任何浏览器中的 `window.location` 的包装器。那么为什么你应该使用它而不是直接使用 `window.location` 呢？

不再使用全局状态

`window.location` 是一个使用全局状态的很好的例子(实际上, 浏览器中的 `window` 和 `document` 对象都是很好的例子). 一旦你的应用程序中有全局的状态(通常我们都说全局变量), 它的测试, 维护和工作都会变得困难(即使不是现在, 从长远来看它肯定是一个潜在的隐患). `$location` 服务隐藏了这个潜在的隐患(也就是我们所谓的全局状态), 并且允许你通过注入 `mocks` 到你的单元测试中来测试你的浏览器位置信息.

API

`window.location` 让你能够完全访问浏览器位置信息的内容. 也就是说, `window.location` 给你一个字符串而 `$location` 服务给你提供了更好的服务, 它提供了类似于 jQuery 的 `setters` 和 `getters` 让你能够使用它以一个干净的方式工作.

AngularJS 集成

如果你使用 `$location`, 你可以在任何你希望使用的时候使用它. 但是如果直接使用 `window.location`, 在有变化时你必须负责通知给 AngularJS, 并且还要监听这些改变/变化.

HTML5 集成

`$location` 服务会在 HTML5 APIs 在浏览器中可用时智能的识别并使用它们. 如果它们不可用, 它会降级使用默认的用法.

那么什么时候你应该使用 `$location` 服务呢? 任何你想反应 URL 变化的时候(它并不是通过 `$routes` 来覆盖的, 而且你应该主要用于基于 URL 工作的视图中), 以及在浏览器中响应当前 URL 变化的时候使用.

让我们考虑使用一个小例子来看看你应该如何在一个实际的应用程序中使用 `$location` 服务. 想象一下我们有一个 `datepicker`, 并且当我们选择一个日期时, 应用程序导航到某个 URL. 让我们一起来看看它看起来可能是什么样子:

```
// 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();
};
```

用或者不用\$apply?

对于 AngularJS 开发者来说什么时候调用`$scope.$apply()`，什么时候不能调用它是比较混乱的。互联网上的建议和谣言非常猖獗。在本小节我们将让它变得非常清楚。但是首先让我们先尝试以一个简单的形式使用`$apply`。

`Scope.$apply` 就像一个延迟的 **worker**。我们会告诉它有很多工作要做，它负责响应并确保更新绑定和所有变化的视图效果。但并不是所有的时间都只做这项工作，它只会在它觉得有足够的工作要做时才会做。在所有的其他情况下，它只是点点头并标记在稍后处理。它只是在你给它指示时并显示的告诉它处理实际的工作。

AngularJS 只是定期在它的声明周期内做这些，但是如果调用来自于外部(比如说一个 jQuery UI 事件)，`scope.$apply` 只是做一个标记，但并不会做任何事。这就是为什么要调用 `scope.$apply` 来告诉它"嘿!你现在需要做这件事，而不是等待!"。

这里有四个快速的提示告诉你应该什么时候(以及如何)调用`$apply`。

- **不要始终调用它。** 当 AngularJS 发现它将导致一个异常(在其`$digest`周期内，我们调用它)时调用`$apply`。因此"有备无患"并不是你希望使用的方法。
- 当控制器在 AngularJS 外部(DOM 时间，外部回调函数如 jQuery UI 控制器等等)调用 AngularJS 函数时调用它。对于这一点，你希望告诉 AngularJS 来更新它自身(模型，视图等等)，而`$apply`就是做这个的。
- 只要可能，通过传递给`$apply`来执行你的代码或者函数，而不是执行函数，然后调用`$apply()`。例如，执行下面的代码：

```
$scope.$apply(function(){ $scope.variable1 = 'some value';  
excuteSomeAction(); });
```

而不是下面的代码：

```
$scope.variable1 = 'some value';  
excuteSomeAction();  
$scope.$apply();
```

尽管这两种方式将有相同的效果，但是它们的方式明显不同。

第一个会在 `excuteSomeAction` 被调用时将捕获发生的任何错误，而后者则会瞧瞧的忽略此类错误。只有使用第一种方式时你才会从 AngularJS 中获取错误的提示。

- **kaov** 使用类似的 `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); } };
```

你可以在顶层作用域或者根作用域中捕获到它，然后在任何地方使用 `$scope.$safeApply` 函数。一直都在讨论这个，希望在未来的版本中这会称为默认的行为。

是否那些其他的方法也可以在 `$location` 对象中使用呢？表 7-1 包含了一个快速的参考用于让你绑定使用。

让我们来看看 `$location` 服务是如何表现的，如果浏览器中的 URL 是 `http://www.host.com/base/index.html#!/path?param1=value1#hashValue`。

Table 7-1 Functions on the \$location service

Getter Function	Getter Value
<code>absUrl()</code>	http://www.host.com/base/index.html#!/path?param1=value1#hashValue
<code>hash()</code>	<code>hashValue</code>
<code>host()</code>	www.host.com
<code>path()</code>	<code>/path</code>
<code>protocol()</code>	<code>http</code>
<code>search()</code>	<code>{'a':'b'}</code>
<code>url()</code>	<code>/path?param1=value1?hashValue</code>

表 7-1 的 **Setter Function** 一列提供了一个值样本表示 **setter** 函数与其的对象类型。

注意 `search()` setter 函数还有一些操作模式：

- 基于一个 `object<string, string>` 调用 `search(searchObj)` 表示所有的参数和它们的值。
- 调用 `search(string)` 将直接在 URL 上设置 URL 的参数为 `q=String`。

- 使用一个字符串参数和值调用 `search(param, value)` 来设置 URL 中一个特定的搜索参数(或者使用 `null` 调用它来移除参数).

使用任意一个这些 `setter` 函数并不意味着 `window.location` 将立即获得改变. `$location` 服务会在 Angular 生命周期内运行, 所有的位置改变将积累在一起并在周期的后期应用. 所以可以随时作出改变, 一个接一个的, 而不用担心用户会看到一个不断闪烁和不断变更的 URL 的情况.

HTML5 模式和 Hashbang 模式

`$location` 服务可以使用 `$locationProvider` (就像 AngularJS 中的一切一样, 可以注入)来配置. 对它提供两个属性特别有兴趣, 分别是:

html5Mode

一个决定 `$location` 服务是否工作在 HTML5 模式中的布尔值.

hashPrefix

提个字符串值(实际上是一个字符)被用作 Hashbang URLs(在 Hashbang 模式或者旧版浏览器的 HTML 模式中)的前缀. 默认情况下它为空, 所以 Angular 的 hash 就只是". 如果 `hashPrefix` 设置为 '!', 然后 Angular 就会使用我们所称作的 Hashbang URLs(url 紧随 '!' 之后).

你可能会问, 这些模式是什么? 嗯, 假设你有一个超级棒的网站

`www.superawesomewebsite.com` 在使用 AngularJS.

比方说你有一个特定的路由(它有一些参数和一个 hash), 比如 `/foo?bar=123#baz`.

在默认的 Hashbang 模式中(使用 `hashPrefix` 设置为 '!'), 或者不支持 HTML5 模式的旧版浏览器中, 你的 URL 看起来像这样:

```
http://www.superawesomewebsite.com/#!/foo?bar=123#baz
```

然而在 HTML5 模式中, URL 看起来会像这样:

```
http://www.superawesomewebsite.com/foo?bar=123#baz
```

在这两种情况下, `location.path()` 就是 `/foo`, `location.search()` 就是 `bar=123`, `location.hash()` 就是 `baz`. 因此如果是这种情况, 为什么你不希望使用 HTML5 模式呢?

Hashbang 方法能够在所有的浏览器中无缝的工作, 并且只需要最少的配置. 你只需要设置 `hashBang` 前缀(默认情况下为 '!')并且你可以做到更好.

HTML 模式中, 在另一方面, 还可以通过使用 HTML5 的 History API 来访问浏览器的 URL. 而 `$location` 服务能足够智能的判断浏览器是否支持 HTML5 模式, 必要的情况下还可以降级使用 Hashbang 方法, 因此你不需要担心额外的工作. 但是你不能不注意以下事情:

服务端配置

因为 HTML5 的链接看起来像你应用程序的所有其他 URL, 你需要很小心的在服务端将你应用程序的所有链接路由连接到你的主 HTML 页面(最有可能的是 `index.html`). 例如, 如果你的应用是 `superawesomewebsite.com` 的登录页, 并且你的应用中有一个 `/amazing?who=me` 的路由, 然后 URL 在浏览其中显示为

`http://www.superawesomewebsite.com/ amazing?who=me+`.

当你浏览你的应用程序时, 默认情况下表现很好, 因为有 HTML5 History API 介入和负责很多事情. 但是如果你尝试直接浏览这个 URL, 你的服务器会认为你是不是疯了, 因为在服务端它并不知道这个资源. 所以你必须确保所有指向 `/amazing` 的请求被充定向到 `/index.html#!/amazing`.

AngularJS 将会以这种形式来在这一点注意这些事情. 它会检测路径的改变并冲顶像到我们所定义的 AngularJS 路由中.

Link rewriting(链接改写)

你可以很容易的像下面这样指定一个 URL:

```
<a href="/some?foo=bar">link</a>
```

根据你是否使用的 HTML5 模式, AngularJS 会注意分别重定向到 `/some?foo=bar` 或者 `index.html#!/some?foo=bar`. 没有额外的步骤需要你处理. 很棒, 是不是?

但是下面的链接形式像不会被改写, 并且浏览器将在这个页面上执行一个完整的重载:

- a. 链接像下面这样包含一个 `target` 元素

[link](#)

- b. 链接到一个不用域名的绝对路径:

[link](#)

这里时不同的, 因为它是一个绝对的 URL 路径, 而前面的记录会使用现有的基础 URL.

- c. 链接基于一个不同的已经定义好的路径开始时:

[link](#)

Relative Links(相对链接)

一定要检查所有的相对链接(相对路径), 图片, 脚本等等. 你必须在你主 HTML 文件的头部指定基本的参照 URL(), 或者你必须在每一处使用绝对 URLs 路径(以/开头的), 因为相对的 URL 将会使用文档中初试的绝对 URL 被解析为绝对的 URL, 这往往不同于应用程序的根源.

强烈建议从文档根源启用 History API 来运行 Angular 应用程序, 因为它要注意所有相对路径的问题.

AngularJS 模块方法

AngularJS 模块负责定义如何引导你的应用程序。它还声明定义了应用程序片段。接下来让我们一起看看它是如何实现这一点的。

主方法在哪？

如果你来自于 Java, 甚至是 Python 编程语言社区, 你可能会疑惑, AngularJS 中的主方法在哪? 你知道的, 主方法会引导一切, 并且它是首先会个执行的东西? 它会将 JavaScript 函数和实例以及所有的事情联系在一起, 然后再通知你的应用程序去运行?

但是在 AngularJS 中没有。替代的是 Angular 中的模块的概念。模块允许我们声明指定我们应用程序的依赖, 以及应用程序的引导是如何发生的。使用这种方式的原因时多方面的。

1. 首先是**声明**。这意味以这种方式编写代码更容易编写和理解。就像阅读英语一样!
2. 它是**模块化的**。它会迫使你思考如何定义你的组件和依赖, 并让它们很明确。
3. 它还允许**简单测试**。在你的单元测试中, 你可以选择性的拉去模块来测试, 以规避代码中不可以测试的部分。同时在你的场景测试中, 你还可以加载附加的模块, 这样可以结合某些组件一起工作让工作变得更容易。

那么接下来, 先让我们看看你要如何使用一个已经定义好的模块, 然后再来看看我们如何声明一个模块。

比方说我们有一个模块，实际上，我们的应用程序中有一个名为"MyAwesomeApp"的模块。在我的 HTML 中，我可以只添加下面的<html>标签(从技术上讲，也可以是任何其他标签)。

```
<html ng-app="MyAwesomeApp">
```

这里的 `ng-app` 指令会告诉你的 AngularJS 可以使用 `MyAwesomeApp` 模块来引导你的应用程序。

那么，这个模块是如何定义的呢？嗯，首先我们建议你分你的服务，指令和过滤器模块。然后在你的主模块中，你就可以只声明你所依赖的其他模块(与我们在第 4 章中使用 RequireJS 的例子一样)。

这种方式让你管理模块变得更容易，因为它们都是很好的完整的代码块。每个模块有且仅有一个职责。这样就允许你在测试中只载入你所关心的模块，从而减少了初始化这些模块的数量。这样，测试就可以变得更小并且只会关心重点。

加载和依赖

模块的加载发生在两个不同的阶段，并且它们都有对应的函数。它们分别是配置和运行块(阶段)：

配置块

AngularJS 会在这个阶段挂接和注册所有的供应商(提供的模块)。这是因为，只有供应商和常量才能够注入到配置块中。服务能不能被初始化，并不能被注入到这个阶段。

运行块

运行块用于快速启动你的应用程序，并且在注入任务完成创建之后开始执行应用程序。从此刻开始会阻止接下来的系统配置，只有实例和常量可以注入到运行块中。在 AngularJS 中，运行块是最接近你想要寻找的主方法的东西。

快捷方法

那么可以用模块做什么呢？我们可以实例化控制器，指令，过滤器和服务，但是模块类允许你做更多的事情，正如表 7-2 所示：

Table 7-2 模块的快捷方法

API 方法	描述
<code>config(configFn)</code>	模块加载时使用这个方法注册模块需要做的工作。
<code>constant(name, object)</code>	这个首先发生，因此你可以在这里声明所有的常量`app-wide`，和声明列表中的第一个方法)以及方法实例(从这里获取所有的方法，如控制器。
<code>controller(name, constructor)</code>	我们已经看过了很多控制器的例子，它主要用于设置一个控制器。
<code>directive(name, directiveFactory)</code>	正如第 6 章所讨论的，它允许你为应用程序创建指令。
<code>filter(name, filterFactory)</code>	允许你创建命名 AngularJS 过滤器，正如第 6 章所讨论的。
<code>run(initializationFn)</code>	使用这个方法在注入设置完成时处理你要执行的工作，也就是将你的前。
<code>value(name, object)</code>	允许跨应用程序注入值。
<code>service(name, serviceFactory)</code>	下一节中讨论。
<code>factory(name, factoryFn)</code>	下一节中讨论。
<code>provider(name, providerFn)</code>	下一节中讨论。

你可能意识到，在前面的表格中我们省略了三个特定 **API-Factory**，**Provider**，和 **Service** 的详细信息。还有一个原因是：这三者之间的用法很容易混肴，因此我们使用一个简单的例子来更好的说明一下什么时候(以及如何)使用它们每一个。

The Factory

Factory API 可以用来在每当我们有一个类或者对象需要一定逻辑或者参数之前才能初始化的时候调用。一个 **Factory** 就是一个函数，这个函数的职责是创建一个值(或者一个对象)。让我们来看一个例子，**greeter** 函数需要和它的 **salutation** 参数一起初始化：

```
function Greeter(salutation) {  
  this.greet = function(name) {  
    return salutation + ' ' + name;  
  }  
}
```

greeter 工厂方法(它就是一个工厂函数或者说构造函数)看起来就像这样：

```
myApp.factory('greeter', function(salut) {  
  return new Greeter(salut);  
});
```

然后可以像这样调用：

```
var myGreeter = greeter('Halo');
```

The Service

何时服务？嗯，一个 **Factory** 和一个 **Service** 之间的不同就是 **Factory** 方法会调用传递给它的函数并返回一个值。而 **Service** 方法会在传递给它的控制器方法上调用 **"new"** 操作符并返回调用结果。

因此前面的 **greeter** 工厂可以替换为如下所示的 **greeter** 服务：

```
myApp.service('greeter', Greeter);
```

那么我每次访问一个 **greeter** 实例时，**AngularJS** 都会调用 `new Greeter()` 并返回调用结果。

The Provider

这是最复杂的(大部分的都是可配置，很的)一部分。**Provider** 结合了 **Factory** 和 **Service**，同时它会在注入系统完全到位之前抛出 **Provider** 函数能够进行配置的信息(也就是说，它就发生在配置块中)。

让我们来看看使用 **Provider** 修改之后的 **greeter Service**，它看起来可能是下面这样的：

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

这就允许我们在运行时(例如，根据用户选择语言)设置 **salutation** 的值。

```
var myApp = angular.module(myApp, []).config(function(greeterProvider){
  greeterProvider.setSalutation('Namaste');
});
```

每当有人访问 **greeter** 对象实例的时候 **AngularJS** 都会吉利调用 **\$get** 方法。

警告！

这里有一个轻量级的实现，但是它们之间的用法有明显的区别：

```
angular.module('myApp', []);
```

以及

```
angular.module('myApp');
```

这里的不同之处在于第一种方式会创建一个新的 **Angular** 模块，然后它会拉取在方括号([...])中列出的所依赖的模块。第二种方式使用的是现有的模块，它已经在第一次调用用定义好了。

因此你应该确保在完整的应用程序中，下面的代码只使用一次就行了：

```
angular.module('myApp', [...]); // Or MyModule, if you are modularizing your app
```

如果你不打算将它保存为一个变量并且跨应用程序引用它，然后在其他文件中使用 `angular.module(MyApp)` 来确保你获取的是一个正确处理过的 **AngularJS** 模块。模块中的一切都在模块定义中访问变量，或者直接将某些东西加入到模块定义的地方。

\$on, \$emit 和 \$broadcast 之间的作用域通信

AngularJS 中的作用域有一个非常有层次和嵌套分明的结构。其中它们都有一个主要的 `$rootScope` (也就说对应的 **Angular** 应用或者 `ng-app`)，然后其他所有的作用域部分都是继承自这个 `$rootScope` 的，或者说都是嵌套在主作用域下面的。很多时候，你会发现这些作用域不会共享变量或者说都不会从另一个原型继承什么。

那么在这种情况下，如何在作用域之间通信呢？其中一个选择就是在应用程序作用域之中创建一个单例服务，然后通过这个服务处理所有子作用域的通信。

在 **AngularJS** 中还有另外一个选择：通过作用域中的事件处理通信。但是这种方法有一些限制；例如，你并不能广泛的将事件传播到所有监控的作用域中。你必须选择是否与父级作用域或者子作用域通信。

但是在我们讨论这些之前，那么如何监听这些事件呢？这里有一个例子，在我们任意的恒星系统的作用域中等待和监控一个我们称之为 `"planetDestroyed"` 的事件。

```
$scope.$on('planetDestroyed', function(event, galaxy, planet){  
    // Custom event, so what planet was destroyed  
    scope.alertNearbyPlanets(galaxy, planet);  
});
```

或许你会疑惑，传递给事件监听器的这些附加的参数是从哪里来的？那么就让我们来看看一个独立的 `planet` 是如何与它的父级作用域通信的。

```
scope.$emit('planetDestroyed', scope.myGalaxy, scope.myPlanet);
```

`$emit` 的附加参数是以作为监听器函数的函数参数的形式来传递的。并且，`$emit` 只会从它自己当前作用域向上通信，因此，星球上的穷人们(如果它们有自身的作用域)在它们的星球被毁灭之前并不会收到通知。

类似的，如果银河系统希望向下与它的成员通信，也就是恒星系统，那么它们之间的通信可能像下面这样：

```
scope.$emit('selfDestructSystem', targetSystem);
```

然后，所有的恒星系统都可能在目标系统中监听这个事件，并使用下面的命令来决定它们是否应该自毁：

```
scope.$on('selfDestructSystem', function(event, targetSystem){
    if(scope.mySystem === targetSystem){
        scope.selfDestruct(); // Go ka-boom!!
    }
});
```

当然，正如事件向上(或者向下)传播，它都可能必须在同一级或者作用域中说："够了，你不能通过！"，或者阻止事件的默认行为。传递给监听器的事件对象都有函数来处理上面的这些所有事情，或者更多，因此让我们在表 7-3 中看看你可以获取事件对象的哪些信息。

表 7-3 事件对象的属性和方法

事件属性	目的
event.targetScope	发出或者传播原始事件的作用域
event.currentScope	目前正在处理的事件的作用域
event.name	事件名称
event.stopPropagation()	一个防止事件进一步传播(冒泡/捕获)的函数(这只适用于使用`\$emit`发出事件)
event.preventDefault()	这个方法实际上不会做什么事，但是会设置`defaultPrevented`为 true。在采取行动之前它才会检查`defaultPrevented`的值。

事件属性	目的
event.defaultPrevented	如果调用了`preventDefault`则为 true

说明：关于通信这一节译文很粗糙，待斟酌校对。

Cookies

不就之后，在你的应用程序中(假设它足够大并且很复杂)，你需要在客户端通过用户的 **session** 来存储用户会话的某些状态。你可能还记得(或者说还会做噩梦)，通过 `document.cookie` 接口来处理纯文本形式的 **cookies**。

值得庆幸的是，这么多年过去了，并且 **HTML5** 提供的相关 **API** 都能够在现在已经出现的大多数现代浏览器上可用。此外，**AngularJS** 还给你提供了很好的 `$cookie` 和 `$cookieStore` **API** 用来处理 **cookies**。这两个服务都能够很好的发挥 **HTML5 cookies**，当 **HTML5 API** 可用时浏览器会选择使用 **HTML5** 提供的 **API**，如果不可用则默认选择 `document.cookies`。无论那种方式，你都可以选择使用相同的 **API** 来进行工作。

首先让我们来看看 `$cookie` 服务。`$cookie` 是一个简单的对象。它有键(属性)和值。给这个对象添加一个键和对应的值，就会将相关信息添加到 **cookie** 中，反之，从对象中移除键(属性)时就会从 **cookie** 中删除对应的信息。它就是这么简单。但是大多数时候，你都不会希望直接在 `$cookie` 上工作。直接在 `cookies` 上工作意味着你必须自己处理字符串转换操作和解析工作，并且还要从在对象中转换相应的数据。对于这些情况，我们有一个 `$cookieStore` 方法，它提供了一种编写和移除 `cookie` 的方式。因此你可以很方便的使用 `$cookieStore` 来构建一个 **Search** 控制器用户记忆最后五个搜索结果，就像下面这样：

```
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

你可能会听到人们提到这些术语，当它们使用不同的语言支持应用程序时。但是在这两者之间有一些细微的区别。想象一下有一个简单的应用程序作为进入用户银行账单的入口。每当你进入这个应用时，它显示且只显示一个东西：

欢迎！2012 年 10 月 25 日你的账单数据为\$xx,xxx。

现在，明显，上面的代码(信息)目标用户直接定位为美国公民。但是如果我们希望这个应用程序也能够英国(UK)很好的工作(简单的说就是由程序自身根据环境来选择语言)要做什么呢？大不列颠(英国)使用的是不同的日期格式和货币符号，但是你不希望每次你需要应用程序只是一个新的环境是都发生一次变化(比如在 `en_US` 和 `en_UK`)。这里需要抽象的处理输出的日期/事件格式，以及货币符号，都需要从你的代码中来适配国际化的环境(或者 `i18n--18` 表示单词中 `i` 和 `n` 之间的字符数)。

如果我们希望这个应用程序在印度也适用呢？或者俄罗斯？此外还有日期格式和货币符号(和格式)，甚至在 UI 中使用的字符串都需要改变。这种在不同地区转换和本地化分离的二进制字符串的形式就是我们所知道的本地化(或者 `L10n--` 使用大写的 `L` 来区分 `i` 和 `l`)。

在 AngularJS 中我能做什么？

AngularJS 支持下面所列出的 `i18n/L10n`：

- `currency`
- `date/time`
- `number`

对于这些使用 `ngPluralize` 指令也可以多元化支持(对于英语就如同 `i18n/L10n`)。所有的这些多元化的支持都是通过 `$locale` 服务来处理和维护的，用户管理本地特定的规则设置。`$locale` 服务清理本地的 IDs，一般由两部分组成：国家代码和语言代码。例如，`en_US` 和 `en_UK`，分别表示美式英语和英式英语。指定一个国家代码是可选的，并且只指定一个 `"en"` 也是有效的本地代码。

如何获取所有工作？

获取 L10n(本地化)和 i18n(国际化)工作的过程在 AngularJS 中分为三个步骤：

index.html changes

AngularJS 需要你有一个单独的 `index.html` 来处理每个受支持的语言环境。你的服务器也需要知道所提供的 `index.html`，根据用户地区的偏好设置(这也可以通过客户端的变化来触发，当用户改变它的语言环境时)。

创建语言环境规则集

接下来的步骤是针对每个受支持的语言环境创建一个 `angular.js`，就像 `angular_en-US.js` 和 `angular_zh-CN.js`。者涉及到在 `angular.js` 或者 `angular.min.js` 的结束处关联每个特定语言的本地规则(前面两个语言环境的默认文件就是 `angular-locale_en-US.js` 和 `angular-locale_zh-CN.js`)。因此你的 `angular_en-US.js` 首先要包含 `angular.js` 的内容，然后就是 `angular-locale_en-US.js` 的内容。

本地规则集来源

最后一步就是涉及到你必须确保你的本地 `index.html` 引用本地规则集而不是原始的 `angular.js` 文件。因此 `index_en-US.html` 中应该使用 `angular_en-US.js` 而不是 `angular.js`。

常见问题

翻译长度

你设计的 UI 在显示 **June 24, 1988** 时，在 `div` 中尽量控制其大小以适当的正确显示。然后你在西班牙语环境中打开你的 UI，然而 **24 de junio de 1988** 不再适应同一空间...

那么当你国际化你的应用程序时，请记住，你的字符串长度可能发生巨大的变化，从一个语言翻译为另一个语言时。你应该适当的设计你的 **CSS**，并且应该在各个不同的语言环境中进行完整的测试(不要忘记还存在从右到左的语言)。

时区问题

AngularJS 的日期/时间过滤器会直接获取来自浏览器的时区设置。因此它依赖于计算机的时区设置，不同的人可能看到不同的信息。无论时 JS 还是 AngularJS 都有任意的内置支持由开发者指定的显示时间的时区的机制。

净化 HTML 和模块

AngularJS 会很认真对待其安全性，它会尝试尽最大的努力以确保将大多数的攻击转向最小化。一种常见的攻击方式就是注入不安全的 HTML 内容到你的 web 页面中，使用这种方式触发一个跨站攻击或者注入攻击。

考虑有这样一个例子，在作用域中我们有一个称之为 `myUnsafeHTMLContent` 的变量。然后使用利用 HTML，使用 `OnMouseOver` 指令修改元素的内容为 `PWN3D!`，就像下面这样：

```
$scope.myUnsafeHTMLContent = '<p style="color:blue;>an html' +  
  '<em onmouseover="this.textContent=\'PWN3D!\'">click here</em>' +  
  'snippet</p>';
```

在 AngularJS 中其默认行为是：你有一些 HTML 内容存储在一个变量中并且尝试绑定给它，其返回结果是 AngularJS 脱离你的内容并打印它。因此，最终得到的 HTML 内容被视为纯文本内容。

因此：

```
<div ng-bind='myUnsafeHTMLContent'></div>
```

会返回：

```
<p style="color:blue">an html  
<em onmouseover="this.textContent='PWN3D!'">click here</em> snippet</p>
```

最后作为文本渲染在你的 Web 页面上。

但是如果你想将 `myUnsafeHTMLContent` 的内容作为 HTML 呈现在你的 AngularJS 应用程序呢？在这种情况下，AngularJS 提供额外的指令(和用于引导的服务 `$sanitize`)允许你以安全和不安全的方式呈现 HTML。

让我们先来看看使用安全形式的例子(通常也应该如此！)，并且呈现相关 HTML，小心的避免 HTML 最可能受到攻击的部分。在这种情况下你会使用 `ng-bind-html` 指令。

`ng-bind-html`，`ng-bind-html-unsafe` 以及 `linky` 过滤器都在 `ngSanitize` 模块中。因此你的脚本依赖中需要包含 `angular-sanitize.js`(或者 `.min.js`)，然后添加一个 `ngSanitize` 模块依赖，在所有这些工作进行之前。

那么当我么在同样的 `myUnsafeHTMLContent` 中使用 `ng-bind-html` 指令时会发生什么呢？就像这样：

```
<div ng-bind-html="myUnsafeHTMLContent"></div>
```

在这种情况下输出内容就像下面这样：

```
an html _click here_ snippet
```

重要的是要注意这里的样式标记(设置字体颜色为蓝色的样式)，以及标签上的 `onmouseover` 事件处理器都被 AngularJS 移除了。它们被视为不安全的信息，因而被弃用。

最终，如果你决定你确实像呈现 `myUnsafeHTMLContent` 的内容，无论你是真正相信 `myUnsafeHTMLContent` 的内容还是其他原因，那么你可以使用 `ng-bind-html-unsafe` 指令：

```
<div ng-bind-html-unsafe="myUnsafeHTMLContent"></div>
```

那么这种情况下，输出的内容就像下面这样：

```
an html _click here_ snippet
```

此时文本颜色为蓝色(正如附加给 `p` 标签的样式)，并且 `click here` 还有一个注册给它的 `onmouseover` 指令。因此一旦你的鼠标从其他地方滑入 `click here` 这几个文本时，输出就为改变为：

```
an html PWN3D! snippet
```

正如你可以看到的，显示中这是非常不安全的，因此大概你决定使用 `ng-bind-html-unsafe` 指令时你要绝对肯定这是你想要的。因为其他人可能很容易读取用户信息并发送到他/她的服务器中。

Linky

目前 `linky` 过滤器也存在于 `ngSanitize` 模块中，并且基本上允许你将它添加到 HTML 内容中呈现并将现有的 HTML 转为锚点标记的链接。它的用法很简单，让我们来看一个例子：

```
$scope.contents = 'Text with links: http://angularjs.org/ & mailto:us@there.org';
```

现在，如果你使用下面的方式来绑定数据：

```
<div ng-bind-html="contents"></div>
```

这将导致数据会作为 HTML 内容打印在页面中，就像下面这样：

```
Text with links: http://angularjs.org/ & mailto:us@there.org
```

接下来让我们看看如果我们使用 `linky` 过滤器会发生什么:

```
<div ng-bind-html="contents | linky"></div>
```

`linky` 过滤器会通过查找文本内容中的 URL，给其中所有的 URL 格式的文本添加一个 `<a>` 标签和一个 `mailto` 链接，从而最终展现给用户的 HTML 内容就编程下面这样了:

```
Text with links: http://angularjs.org/ & us@there.org
```

第八章 备忘与诀窍

目前为止，之前的章节已经覆盖了 Angular 所有功能结构中的大多数，包括指令、服务、控制器、资源以及其它内容。但是我们知道有时候仅仅阅读是不够的。有时候，我们并不在乎那些功能机制是如何运行的，我们仅仅想知道如何用 AngularJS 去做实现一个具体功能。

在这一章中，我会给出完整的样例代码，并且对这些样例代码仅仅给出少量的信息和解释，这些代码解决的是我们在大多数 Web 应用中碰到的通用问题。这些代码没有具体的先后次序，你尽可以跳到你关心的小节先睹为快或者按着本书内容顺序阅读，选那种阅读方式由你决定。

这一章中我们将要给出代码样例包括以下这些：

1、封装一个 jQuery 日期选择器(DatePicker) 2、团队成员列表应用:过滤器和控制器之间的通信 3、AngularJS 中的文件上传 4、使用 socket.IO 5、一个简单的分页服务. 6、与服务器后端的协作

封装一个 jQuery 日期选择器

这个样例代码文件可以在我们 GitHub 页面的 `chapter8/datepicker` 目录中找到。

在我们开始实际代码之前，我们不得不做出一个设计决策：我们的这个组件的外观显示和交互设计应该是什么样子，假设我们想定义的日期选择器在 HTML 里面使用像以下代码这样：

```
<input datepicker ng-model="currentDate" select="updateMyText(date)" ></input>
```

也就是说我们想修改 `input` 输入域,通过给她添加一个叫 `datepicker` 的属性,来给它添加一些更多的功能(就像这样:它的数据值绑定到一个 `model` 上,当一个日期被选择的时候,输入域能被提醒修改).那么我们如何做到这一点哪?

我们将要复用现存的功能组件:jQueryUI 的 `datepicker`(日期选择器),而不是我们从头自己构建一个日期选择器.我们只需要把它接入到 `AngularJS` 上并且理解它提供的钩子(hooks):

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

```

上面代码中的大多数都非常简单直接,但是我们还是来看一下其中一些较重要的部分。

ng-model

我们可以得到一个 **ng-model** 属性,这个属性的值将会被传入到指令的链接函数中。**ng-model**(这个属性对于这个指令运行是必须的,因为指令定义中的 **require** 属性定义--见上代码)这个属性帮助我们定义属性和绑定到 **ng-model** 上的(js)对象在指令的上下文中的行为机制.这儿有两点你需要注意一下:

```
ngModel.$setViewValue(dateTxt)
```

当 **Angular** 外部某些事件(在这个示例中就是 **jQueryUI** 日期选择器组件中某日期被选定的事件)发生时,上面这条语句会被调用.这样就可以通知 **AngularJS** 更新模型对象.这种语句一般是在某个 **DOM** 事件发生时被调用.

```
ngModel.$render
```

这是 **ng-model** 的另外一部分.这个可以协调 **Angular** 在模型对象发生变化时如何更新视图.在我们的示例中,我们仅仅给 **jQueryUI** 日期选择器传递了发生了改变的日期值.

绑定 select 函数

(结合代码理解本小节内容-译者注)取代使用属性值然后用它计算成作用域对象(**scope**)的一个字符串属性的做法(在这个案例中,嵌套在指令内部的函和对象不是可直接操作的),我们使用了函数方法绑定("&"作用域对象绑定-注意看上面 **scope** 对象定义部分代码-译者注).这就在 **scope** 作用域对象上建立了一个叫 **select** 的新方法.这个方法函数有一个参数,参数是一个对象.这个对象中的每个键必须匹配使用了该指令 **HTML** 元素中的一个确定参数.这个键的值将会作为传递给函数的参数.这个特性添加的优势在于解耦:实现控制器时不需要知道 **DOM** 和指令的相关细节.这种回调函数仅仅根据指定参数执行他的动作,而且不需要知道绑定和刷新的细节.

调用 select 函数

注意我们给 `datepicker` 传递了一个 `optionsObj` 参数,这个参数对象有一个 `onSelect` 函数属性。`jQueryUI` 组件(此处就是指 `datepicker`)负责调用 `onSelect` 方法,这通常在 `AngularJS` 的执行上下文环境之外发生。当然,当像 `onSelect` 这样的函数被调用的时候,`AngularJS` 不会得到通知提示。让 `AngularJS` 知道它需要对数据做一些操作是我们应用程序员的责任。我们如何来完成这个任务?通过使用 `scope.$apply`。

现在我们可以很容易地在 `scope.$apply` 范围之外调用 `setViewValue` 和 `scope.select` 方法,或者仅通过 `scope.$apply` 调用。但是前面那两步(范围之外地)的任何一步发生异常都会静悄悄地被丢弃。但是如果异常是在 `scope.$apply` 函数内部发生,就会被 `AngularJS` 捕捉。

示例代码的其它部分

为了完成这个示例,让我看一下我们的控制器代码,然后让页面正常地跑起来:

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

非常简单的代码。我们声明了一个控制器,设置了一些作用于对象(`$scope`)变量,然后创建了一个方法(`updateMyText`),这个方法后来将会被用来绑定到 `datepicker` 的 `on-select` 事件上。下一步补上 `HTML` 代码:

```
<!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 HTML 元素中的 **select** 属性是如果被声明的.在作用域对象(scope)的范围内没有"date"这个值.但是因为我们已经在指令绑定过程中装配了我们的函数.AngularJS 现在就知道这个函数将会有有一个参数,参数名称将会是"date".这个也就是当 **datepicker** 组件的 **onSelect** 事件绑定函数被调用时我们定义的的那个对象将会传入这个参数.

对于 **ng-model**,我们定义用 **\$parent.currentDate** 取代了 **currentDate**.为什么?因为我们的指令创建了一个隔离的作用域以便于做 **select** 函数绑定这件事.这将使得 **currentDate** 不再被 **ng-model** 所即使我们设定了它.所以我们不得不显式地告诉 AngularJS:需要引用的 **currentDate** 变量不是在隔离作用域内,而是在父作用域内.

做到这一步,我们可以在浏览器内加载这个示例代码,你将会看到一个文本框,当你点击的时候,将会弹出一个 jQueryUI 日期选择器.选定日期后,显示文本"Not Selected"将会被更新为"Selected",选择日期也刷新显示,输入框内的日期也会被更新.

"小组成员列表"应用：数据过滤与控制器之间的通信

在这个示例中,我们将同时处理很多事情,但是其中只有两个新技术点:

- 1、怎样使用数据过滤器--特别是已简洁的方式使用--和重复指令(**ng-repeat**)一起用.
- 2、怎样在没有共同继承关系的控制器之间通信.

这个示例应用本身非常简单.其中只有数据,这些数据是各种体育运动队的成员列表.其中包含的运动有篮球、足球(橄榄球式的不是英式足球那种)和曲棍球.对于这些运动队的成员,我们有他们的姓名、所在城市、运动种类以及所在团队是不是主力团队.

我们想做的是显示这个成员列表,在其左边显示过滤器,当过滤器数据发生变化的时候,成员列表数据做出相应的刷新.我们将要构建两个控制器.一个用来保存列表成员数据,另外一个运行过滤机制.我们将要使用一个服务来为列表控制器和过滤控制器之中的过滤数据变化做通信.

想让我们看看这个服务,它将用来驱动整个应用:

```
angular.module('myApp.services', []).  
  factory('filterService', function() {  
    return {  
      activeFilters: {},  
      searchText: ''  
    };  
  });
```

哇哦,你也许会问:这就是全部?嗯,是的.我们此处所写代码基于这样一个事实:AngularJS 服务是单例模式的(这个以小写 **s** 打头的单例(**singleton**)是作用域 **scope** 内的单例,而不是那种全局可见且可读写的那种.)当你声明了一个过滤服务,我们就授权在整个应用范围内只有一个过滤服务的实例对象.

接下来,我们完成使用过滤服务作为过滤控制器和列表控制器之间的通信渠道的其它代码.两个控制器都可以绑定到它(过滤服务)上,而且两个都可以在它(过滤服务)更新时,读取它的成员属性.这两个控制器实际上都简单得要死,因为在其中除了简单的赋值,基本没做什么别的.

```
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;
});

```

你也想知道:那部分代码会很复杂? **AngularJS** 确实使这个示例整体非常简单,接下来我们所需要做的就是把所有这一期在模版中整合在一起:

```

<!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>
        </form>
    </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>

```

在整个上面这个 **HTML** 模板代码里面,真正需要关注的只有四项.除此以外的旧的东西我们到目前为止可能都看了几十遍了(甚至这些旧代码点都曾经以这样那样的形式出现过).让我们来挨个看一下那四个新代码点:

搜索框

搜索框的值用 `ng-model` 指令绑定到了过滤服务的 `searchText` 域上 (`filterService.searchText`), 这个属性域本身没什么值得注意的. 但是在后面他将被用在过滤器上的方式使得现在这儿这步很关键.

组合框

这儿有两个组合框, 尽管我们只高亮了第一个. 但是这两个工作方式相同. 他们都绑定到过滤服务 (`filterService`) 的激活过滤器域 (`activeFilters`) 的 `sports` 属性或者 `city` 属性 (取决于具体组合框). 这个基本设置了过滤服务 (`filtersService`) 的 `filter` 对象的 `sports` 属性或者 `city` 属性.

复选框

复选框绑定到了过滤服务 (`filterService`) 的激活过滤器域 (`activeFilters`) 的 `featured` 属性上. 这里需要注意的是如果复选框被选定. 我们想显示那些 `featured=true` 的主力团队. 如果复选框没有被选定, 我们想显示 `featured=true` 和 `feature=false` 的两种团队 (也就是全部团队). 为了达到这个效果, 我们用 `ng-false-value=""` 指令来告诉程序当复选框没有选定的时候, `featured` 这个过滤属性将会被清掉.

迭代器

让我们再一次看一下 `ng-repeat` 这条语句:

```
"team in teamsList | filter:filterService.activeFilters |  
filter:filterService.searchText"
```

这条语句的第一部分和之前的一样. 后面的两个新的过滤器则让一切变得不一样了. 第一个过滤器告诉 AngularJS 用 `filterService.activeFilters` 域过滤列表数据. 使用过滤对象的每个属性来过滤数据, 确保迭代器里的每个循环项的属性值与过滤对象的对应属性值相匹配. 所以如果 `activeFilter[city]=Dallas`, 那么迭代器里面只有那些 `city=Dallas` 的被选择出来显示. 如果有多个过滤器对象, 那所有过滤器对象的属性都得匹配.

第二个过滤器是个文本值过滤器. 它基本上只过滤那些只有其属性数据中出现 `filterService.searchText` 文本值即可. 所以这个过滤的属性值会包含所有数据项: `cities`、`team names`、`sports` 和 `featured`.

AngularJS 中的文件上传

另外一个我们即将要看的常用情景示例是在 AngularJS 应用中如何实现文件上传功能.虽然目前支持这个功能可以通过 HTML 标准中的 **file** 类型的 **input** 输入域来做,但是为了达到这个示例的目的,我们将会扩展一个现存的文件上传解决方案.目前这方面的优秀实现之一是 [BlueImp's File Upload](#),它是用 **jQuery** 和 **jQueryUI** 实现(或者 **BootStrap**)的.它的 **API** 相当简单,所以这样使得我们的 **AngularJS** 指令也超级简单.

让我们从令定义的代码开始:

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

```
});
```

这段代码帮助我们以一个非常简单的方式定义了我们这个指令,并且添加了 `onDone` 和 `onProgress` 两个函数钩子.我们使用函数绑定来使 **AngularJS** 调用正确的方法而且使用正确的作用域.

这一切都是通过隔离的作用域定义来完成,它之中有两个函数绑定:一个对应 `progress`,另外一个对应 `done`.我们将在作用域对象 `scope` 上创建一个单参数函数.比如: `scope.done` 以一个对象为参数.这个对象内有两个属性键:`e` 和 `data`.这些都作为参数传递给原始定义的函数.这个函数我们将会在下小节中看到.

下面来让我们看一下我们的 **HTML** 代码来看看我们如何使用函数绑定:

```
<!DOCTYPE html>
<html ng-app="myApp">
  <head lang="en">
    <meta charset="utf-8">
    <title>File Upload with AngularJS</title>
    <!-- Because we are loading jQuery before AngularJS,
         Angular will use the jQuery library instead of
         its own jQueryLite implementation -->
    <script
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
    </script>
    <script
      src="http://raw.github.com/blueimp/jQuery-File-Upload/master/js/vendor/
        jquery.ui.widget.js">
    </script>
    <script
      src="http://raw.github.com/blueimp/jQuery-File-Upload/master/js/
        jquery.iframe-transport.js">
    </script>
    <script
      src="http://raw.github.com/blueimp/jQuery-File-Upload/master/js/
        jquery.fileupload.js">
    </script>
    <script
      src="//ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
    </script>
    <script src="app.js"></script>
  </head>
  <body ng-controller="MainCtrl">
    File Upload:
    <!-- We will define uploadFinished in MainCtrl and attach
```

```
        it to the scope, so that it is available here -->
        <input id="testUpload"
              type="file"
              fileupload
              name="files[]"
              data-url="/server/uploadFile"
              multiple
              done="uploadFinished(e, data)">

    </body>
</html>
```

我们的 `input` 标签仅仅添加了以下附加部分:

`fileupload` 这个使得 `input` 标签成为一个文件上传元素

`data-url` 这个属性被 `FileUpload` 插件用来确定文件上传的服务器端处理 URL. 在我们的示例中, 我们假设在 `/server/uploadFile` URL 上有一个服务器端 API 在监听处理上传的文件数据.

`multiple` 这个 `multiple` 属性告诉指令(以及 `fileupload` 组件)允许它一次性可以选择多个文件. 我们可以通过插件轻松实现此功能, 而不需要多写一行额外代码, 这又是内建插件的一个福利啊.

`done` 这是当插件文件上传结束以后 `AngularJS` 要调用的函数. 如果我们想做, 我们也可以以类似的方式为 `progress` 事件也定义一个函数. 这也指定了我们的指令定义中调用的那两个参数函数.

那控制器看起来会是个什么样子那, 正如你所期望的那样, 它的代码是下面这样:

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

有了上面这些代码, 我们就有了一个简单的、可运行且可复用的文件上传指令.

使用 Socket.IO

目前 Web 开发中一个常见需求就是构建实时 Web 应用, 也就是服务器端数据一更新, 前端浏览器的数据也立即实时刷新. 之前使用的技巧如轮询之类的被发现有缺陷, 有时我们仅仅想建立一个连接前端的套接字(socket)用来通信.

`Socket.IO` 是一个优秀的库.它可以帮我们通过非常简单的基于事件 **API** 构建实时 **Web** 应用.下面我们将要开发一个实时的、匿名的消息广播系统(比如 **Twitter**,不过不需要用户名),这个系统将帮助用户把自己的消息广播给所有的 **Socket.IO** 用户同时还可以看见系统的所有消息.这个系统中没有数据会被持久化存储,所以所有的消息只对那些在线活跃用户是可见的,但是这系统用于说明 **Socket.IO** 如何被优雅地集成进 **AngularJS** 这件事已经绰绰有余.

说干就干,我们来把 **Socket.IO** 封装到一个 **AngularJS** 服务里.这样做,我们就可以保证以下几点:

- **Socket.IO** 的事件会在 **AngularJS** 的生命周期被激发和处理
- 测试集成效果将变得很简单

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

此处我们只封装了我们关注的两个函数,他们分别是 **Socket.IO API** 中的 **on** 事件和 **broadcast** 事件方法.**API** 中还有很多事件方法,我们可以以类似的方式封装他们.

下面我们来看一下简单的 `index.html` 文件源码,将展示一个带发送按钮的文本框和一个消息列表.在这个示例中,我们并不跟踪谁发的消息以及什么时候发的.

```
<!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>

```

下面我们看一下 `MainCtrl` 控制器(这段代码是 `app.js` 中的一部分),在这个控制器中我们将把上面这些整合起来:

```

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 = '';
  };
}

```

这个控制器本身非常简单.它监听套接字连接的事件,而且一旦用户点击广播按钮,就让服务知道有新消息了.并且把新消息添加到消息列表把它直接显示给当前用户.

下面我们来完成最后一部分.这是 **NodeJS Server** 如何支撑前端应用的代码,它相应地用 **Socket.IOAPI** 建立了服务器端.

```

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);
});
});
```

以后你可以轻松地扩展这段代码以支持处理更多的消息和更复杂的结构,尽管如此,这段代码已经打好了基础,在其之上,你可以实现客户端浏览器和服务器端之间的套接字连接.

整个示例应用非常简单.它没有做任何数据验证(不管消息是否为空),但是它包含 AngularJS 默认提供的 HTML 代码清理过滤功能.它没有处理复杂的消息,但是它提供了一个将 Socket.IO 集成进 AngularJS 的完全可用的端到端实现.你可以马上就基于它建立你自己的工作生产代码.

一个简单的分页服务

大多数 Web 应用中一个常用的功能情景是显示一个项目列表.通常,我们有着比一个单个页面合理显示量更大的数据量.这样一个需求场景下,我们想以分页的方式显示这些数据,而且用户还可以在不同的页面之间穿梭.因为这一个在所有 Web 应用之中很常见的场景,所以有理由把这个功能抽取出来封装成一个公共可复用的分页服务.

我们的分页服务(一个非常简单的实现)将帮助该服务的用户在给定的数据偏移量、单页数据量、数据总量条件下取得分页数据.它在内部将处理以下逻辑:某一页要取那些数据,如何下一页存在的情况下,下一页是那页等等必须功能逻辑.

这个服务可以进一步扩展来在服务类缓存数据项,但是这个就作为练习题留给广大读者了.我们的示例全部所需要的就是把当前页数据项 `currentPageItems` 存储在缓存里.在它可用的情况下取出它,就相当于别的取数据函数 `fetch Function` 那一类的东西.

下面我们看一下这个服务的实现:

```
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;
});
});

```

分页服务被调用的时候需要两个参数：一个是 `fetch` 取数据的函数,还有一个就是每页的大小.取数据的函数希望是如下这个函数签名：

```
fetchFunction(offset, limit, callback);
```

一旦这个函数需要取得数来显示一个页面,它就会给以正确的数据偏移量、单页数据大小两个参数而被分页服务调用.在这个函数的内部,它可以或者从一个大的数据数据中做切片或者想服务器端发出请求取回数据.一旦数据可用,取数(`fetch`)函数就需要调用那个作为参数的回调函数.

让我们看一下啊这个函数的设计说明,将要澄清说明我们在有一个包含太多返回数据项的大数组的前提下如何使用这个函数。请注意：这是一个单元测试.由于其实现方式的原因，我们可以在不需要任何控制器和 **XHR** 异步请求的情况下测试这个服务.

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


分页服务暴露其自身的 `currentPageItems` 当前分页数据项这个变量,这样它就可以在模板中被迭代器绑定(或者其它想显示这些数据项的地方).`hasNext()`和 `hasPrevious()`两个函数可已被用来确定是否显示下一页或者上一页这两个链接.而在 `click` 事件上,我们只需要分别调用 `next()`或者 `previous()`这两个函数.

那么在我们需要从服务器端取回每页数据的条件下这个服务应该如何使用哪? 这儿有这么一个控制器:它每显示一页数据都需要从服务器端取回一次搜索结果数据.大概代码如下:

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

使用这个分页服务的 **HTML** 页面代码数据如下:

```
<!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>
```

和服务服务器之间的协作与登录

最后一个案例将要覆盖众多的场景,它们中的全部或者大多数都与`$http`资源有联系. 在我们的经验中,`$http`服务是 **AngularJS** 核心服务之一.同时它可以被扩展来满足 Web 应用的很多常见功能需求,包括:

- 共享一个公共错误处理代码点
- 处理认证和登录之后的重定向
- 与不支持或者支持 **JSON** 通信的服务器协作.
- 通过 **JSONP** 与外部服务(非同域的)之间的通信

所以在这个(轻度设计)的示例中,我们将会有一个成熟 **WebApp** 的骨架,它将会包括如下:

1.在 `butterbar` 指令显示所有不可恢复的错误(不包括验证失败 **HTTP401** 响应),只有异常发生的时候,这个指令才会在屏幕上出现. 2.将会有一个 `ErrorService`,它将会被用来在指令、视图和控制器之间的通信工作. 3.在服务器端响应 **401** 验证失败时激发一个事件(事件 `loginRequired`).它将会被覆盖整个应用的根控制器所处理. 4.处理那些需要带验证头信息的服务器请求,这些请求是特定于当前用户的.

我们不会覆盖整个应用的所有元素(比如路由、模板等等),而且大多数代码是简明易懂的.我们只高亮显示那些与主题关系较密切的代码(便于您把这些代码复制粘帖到您的代码库中并以正确的方式开始编码).这些代码将会是完全功能性代码.如果你想看定义服务或者工厂的代码,请参阅第 7 章.如果你想看如何与服务器端协同合作,可以参考第 5 章.

首先让我看回一下 **Error** 服务的代码:

```
var servicesModule = angular.module('myApp.services', []);
servicesModule.factory('errorService', function() {
  return {
    errorMessage: null,
    setError: function(msg) {
```

```

        this.errorMessage = msg;
    },
    clear: function() {
        this.errorMessage = null;
    }
};
});

```

我们的 `error message` 错误消息指令,它实际上与 **Error** 服务是独立的,它指挥寻找一个弹出框的消息属性,然后把它绑定到模板中.只有错误消息出现的情况下,弹出框才会显示.

```

// 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);
            };
        }
    };
}]);

```

我们添加进 **HTML** 的弹出框代码将如下所示:

```
<div alert-bar alertmessage="errorService.errorMessage"></div>
```

我们需要保证在上面这段 HTML 被新增前, `ErrorService` 必须以 "errorService" 属性名保存在作用域对象范围之内. 也就是说: 如果 `RootController` 是负责拥有 `AlertBar` 指令的控制器, 那么代码应如下:

```
app.controller('RootController',
    ['$scope', 'ErrorService', function($scope, ErrorService) {
        $scope.errorService = ErrorService;
    }]);
```

它给我们一个像样的框架来显示或隐藏错误信息和提示框. 现在让我看看, 如何利用拦截器来处理服务器端可能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);
            });
        };
    });
```

对于某些地方一些控制器来说, 所有需要做的就是注册监听 `loginRequired` 事件, 然后重定向到登录页面(或者做相对更复杂的效果, 比如显示一个登录模态对话框).

```
$scope.$on('event:loginRequired', function() {
    $location.path('/login');
});
```

剩下的就是处理需要认证授权的 Web 请求了. 我们目前只说所有需要认证授权的 Web 请求都有一个 "Authorization" 报头, 这个报头的值对于每一个当前登录用户是特定的. 因为这个报头值每次登录都会改变, 所以我们不能用默认的

`transformRequests`,因为它的数据改变是在 `config` 级.取而代之的是,我们将会封装 `$http` 服务,从而构建我们自己的 `AuthService`.

我们也会有一个认证服务,他负责存储用户的认证信息(在你需要的时候读取它,通常是在登录过程中发生).`AuthHttp` 服务将会访问这个认证服务并通过添加必要的报头来给 **Web** 请求授权.

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

任何需要授权的请求其请求发起函数将会用 `authHttp.get()` 取代 `$http.get()`. 只要 **Authentication** 服务的被设定是正确的信息,你的每次 **Web** 请求调用都会快捷如飞地通过认证.因为我们用一个服务来做授权这个事情,所以其信息对于整个 **Web** 应用来说都是可用的,也就不需要每次路由改变的时候都不得不去读取验证信息.

这已经覆盖了我们在这个 **Web** 应用中需要的所有细节.你可以直接从这儿拷贝代码到你自己的应用代码中,让它为你工作.祝你好运.

总结

当带着我们到这本书末尾的时候，我们几乎接近覆盖了关于 **AngularJS** 的所有内容.写这本书我们的目标就是给大家提供一个坚实的基础,在这个基础之上我们能开始我们的探索并且愉快地使用 **AngularJS** 做开发.我们覆盖了所有的基础知识(和一些高级话题),并且沿途提供了尽可能多的示例.

大功告成，一切都做完了吗？不,我们还需要花大功夫去学习 **AngularJS** 外在功能之下的内在运行机制.比如我们的内容从没有涉及过如何构建复杂且相互依赖的指令.还有那么多的内容没有提及,我们用三本甚至四本书来讲可能都不够.但是我们希望这本书能够给你信心去处理碰到的更复杂的需求.

我们花了大量的时间来写这本书,所以希望能够在 **Internet** 上看到一些用 **AngularJS** 实现的令人惊艳的应用.