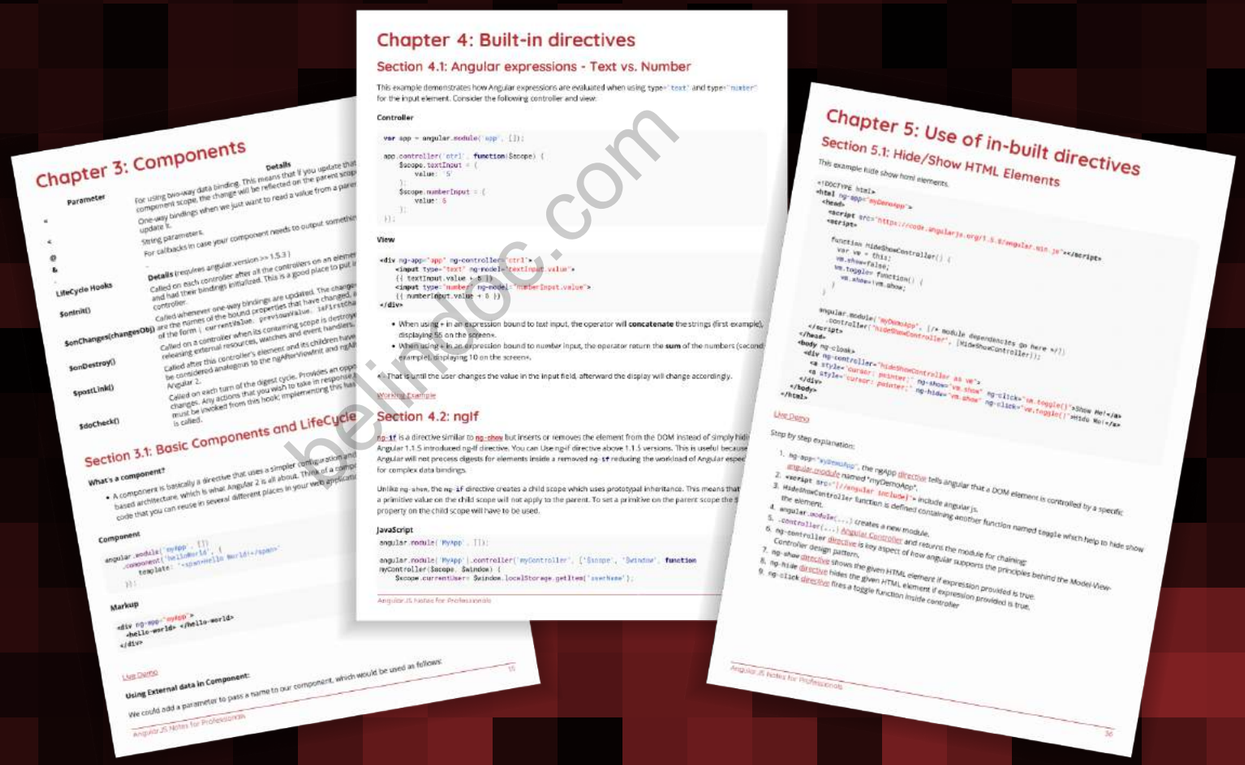
专业人士笔记

AngularJS

专业人士笔记



100多页

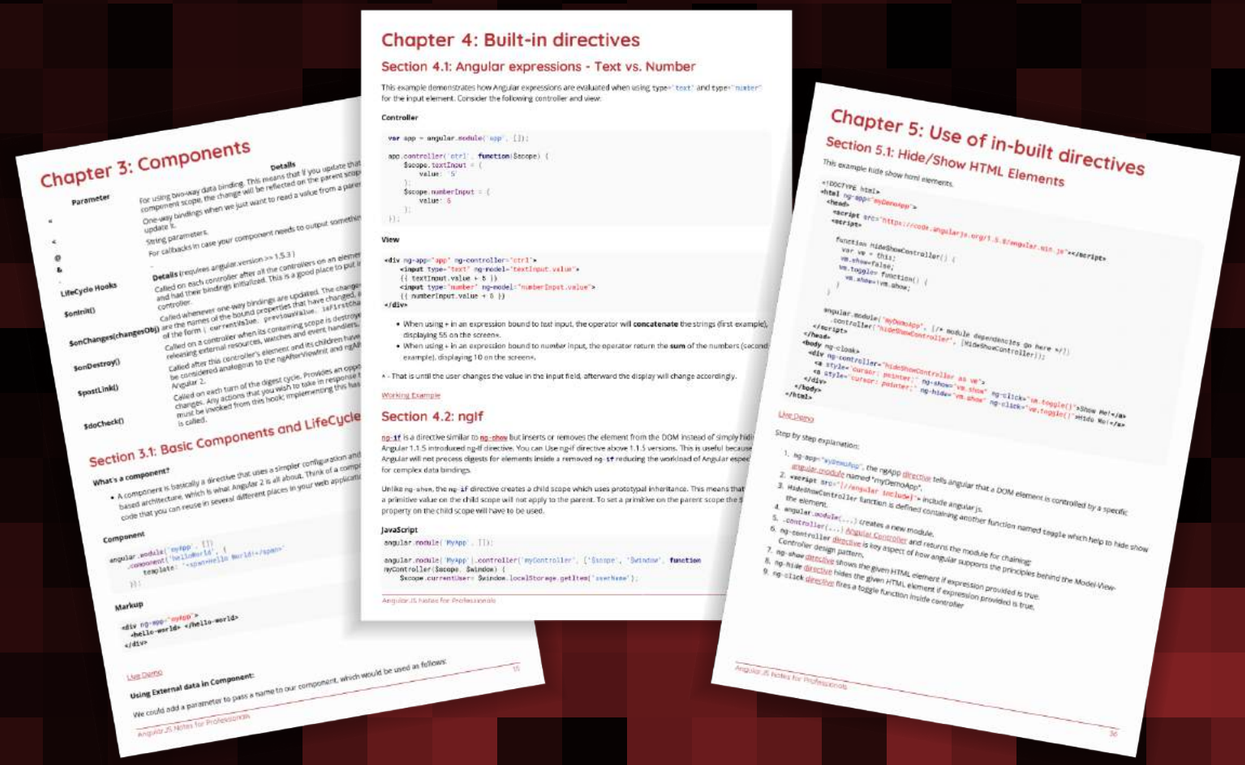
专业提示和技巧

GoalKicker.com
免费编程书籍

免责声明
这是一本非官方的免费书籍，旨在教育用途，
与官方AngularJS组织或公司无关。
所有商标和注册商标均为其各自所有者的财产

AngularJS

Notes for Professionals



100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official AngularJS group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

目录

Contents

beilindoc.com

elibdoc.com

gitbook.com

gitbook.com

allitebooks.com

belindoc.com

关于

欢迎随意免费分享此 PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/AngularJSBook

这本*AngularJS 专业人士笔记*是从Stack Overflow Documentation汇编而成，内容由 Stack Overflow 的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

这是一本非官方的免费书籍，旨在教育用途，与官方 AngularJS 组织或公司及 Stack Overflow 无关。所有商标和注册商标均为其各自公司所有者所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

beijin

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/AngularJSBook

This *AngularJS Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official AngularJS group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：开始使用AngularJS

版本	发布日期
1.6.5	2017-07-03
1.6.4	2017-03-31
1.6.3	2017-03-08
1.6.2	2017-02-07
1.5.11	2017-01-13
1.6.1	2016-12-23
1.5.10	2016-12-15
1.6.0	2016-12-08
1.6.0-rc.2	2016-11-24
1.5.9	2016-11-24
1.6.0-rc.1	2016-11-21
1.6.0-rc.0	2016-10-26
1.2.32	2016-10-11
1.4.13	2016-10-10
1.2.31	2016-10-10
1.5.8	2016-07-22
1.2.30	2016-07-21
1.5.7	2016-06-15
1.4.12	2016-06-15
1.5.6	2016-05-27
1.4.11	2016-05-27
1.5.5	2016-04-18
1.5.4	2016-04-14
1.5.3	2016-03-25
1.5.2	2016-03-19
1.4.10	2016-03-16
1.5.1	2016-03-16
1.5.0	2016-02-05
1.5.0-rc.2	2016-01-28
1.4.9	2016-01-21
1.5.0-rc.1	2016-01-16
1.5.0-rc.0	2015-12-09
1.4.8	2015-11-20
1.5.0-beta.2	*2015-11-18*
1.4.7	2015-09-30
1.3.20	2015-09-30
1.2.29	2015-09-30
1.5.0-beta.1	*2015-09-30*
1.5.0-beta.0	*2015-09-17*
1.4.6	2015-09-17
1.3.19	2015-09-17
1.4.5	2015-08-28

belindoc.com

Chapter 1: Getting started with AngularJS

Version	Release Date
1.6.5	2017-07-03
1.6.4	2017-03-31
1.6.3	2017-03-08
1.6.2	2017-02-07
1.5.11	2017-01-13
1.6.1	2016-12-23
1.5.10	2016-12-15
1.6.0	2016-12-08
1.6.0-rc.2	2016-11-24
1.5.9	2016-11-24
1.6.0-rc.1	2016-11-21
1.6.0-rc.0	2016-10-26
1.2.32	2016-10-11
1.4.13	2016-10-10
1.2.31	2016-10-10
1.5.8	2016-07-22
1.2.30	2016-07-21
1.5.7	2016-06-15
1.4.12	2016-06-15
1.5.6	2016-05-27
1.4.11	2016-05-27
1.5.5	2016-04-18
1.5.4	2016-04-14
1.5.3	2016-03-25
1.5.2	2016-03-19
1.4.10	2016-03-16
1.5.1	2016-03-16
1.5.0	2016-02-05
1.5.0-rc.2	2016-01-28
1.4.9	2016-01-21
1.5.0-rc.1	2016-01-16
1.5.0-rc.0	2015-12-09
1.4.8	2015-11-20
1.5.0-beta.2	2015-11-18
1.4.7	2015-09-30
1.3.20	2015-09-30
1.2.29	2015-09-30
1.5.0-beta.1	2015-09-30
1.5.0-beta.0	2015-09-17
1.4.6	2015-09-17
1.3.19	2015-09-17
1.4.5	2015-08-28

1.3.18	2015-08-19
1.4.4	2015-08-13
1.4.3	2015-07-15
1.3.17	2015-07-07
1.4.2	2015-07-07
1.4.1	2015-06-16
1.3.16	2015-06-06
1.4.0	2015-05-27
1.4.0-rc.2	2015-05-12
1.4.0-rc.1	2015-04-24
1.4.0-rc.0	2015-04-10
1.3.15	2015-03-17
1.4.0-beta.6	*2015-03-17*
1.4.0-beta.5	*2015-02-24*
1.3.14	2015-02-24
1.4.0-beta.4	*2015-02-09*
1.3.13	2015-02-09
1.3.12	2015-02-03
1.4.0-beta.3	*2015-02-03*
1.3.11	2015-01-27
1.4.0-beta.2	*2015-01-27*
1.4.0-beta.1	*2015-01-20*
1.3.10	2015-01-15
1.3.9	2015-01-15
1.4.0-beta.0	*2015-01-14*
1.3.8	2014-12-19
1.2.28	2014-12-16
1.3.7	2014-12-15
1.3.6	2014-12-09
1.3.5	2014-12-02
1.3.4	2014-11-25
1.2.27	2014-11-21
1.3.3	2014-11-18
1.3.2	2014-11-07
1.3.1	2014-10-31
1.3.0	2014-10-14
1.3.0-rc.5	2014-10-09
1.2.26	2014-10-03
1.3.0-rc.4	2014-10-02
1.3.0-rc.3	2014-09-24
1.2.25	2014-09-17
1.3.0-rc.2	2014-09-17
1.2.24	2014-09-10
1.3.0-rc.1	2014-09-10
1.3.0-rc.0	2014-08-30
1.2.23	2014-08-23

belindoc.com

1.3.18	2015-08-19
1.4.4	2015-08-13
1.4.3	2015-07-15
1.3.17	2015-07-07
1.4.2	2015-07-07
1.4.1	2015-06-16
1.3.16	2015-06-06
1.4.0	2015-05-27
1.4.0-rc.2	2015-05-12
1.4.0-rc.1	2015-04-24
1.4.0-rc.0	2015-04-10
1.3.15	2015-03-17
1.4.0-beta.6	2015-03-17
1.4.0-beta.5	2015-02-24
1.3.14	2015-02-24
1.4.0-beta.4	2015-02-09
1.3.13	2015-02-09
1.3.12	2015-02-03
1.4.0-beta.3	2015-02-03
1.3.11	2015-01-27
1.4.0-beta.2	2015-01-27
1.4.0-beta.1	2015-01-20
1.3.10	2015-01-20
1.3.9	2015-01-15
1.4.0-beta.0	2015-01-14
1.3.8	2014-12-19
1.2.28	2014-12-16
1.3.7	2014-12-15
1.3.6	2014-12-09
1.3.5	2014-12-02
1.3.4	2014-11-25
1.2.27	2014-11-21
1.3.3	2014-11-18
1.3.2	2014-11-07
1.3.1	2014-10-31
1.3.0	2014-10-14
1.3.0-rc.5	2014-10-09
1.2.26	2014-10-03
1.3.0-rc.4	2014-10-02
1.3.0-rc.3	2014-09-24
1.2.25	2014-09-17
1.3.0-rc.2	2014-09-17
1.2.24	2014-09-10
1.3.0-rc.1	2014-09-10
1.3.0-rc.0	2014-08-30
1.2.23	2014-08-23

1.3.0-beta.19	*2014-08-23*
1.2.22	2014-08-12
1.3.0-beta.18	*2014-08-12*
1.2.21	2014-07-25
1.3.0-beta.17	*2014-07-25*
1.3.0-beta.16	*2014-07-18*
1.2.20	2014-07-11
1.3.0-beta.15	*2014-07-11*
1.2.19	2014-07-01
1.3.0-beta.14	*2014-07-01*
1.3.0-beta.13	*2014-06-16*
1.3.0-beta.12	*2014-06-14*
1.2.18	2014-06-14
1.3.0-beta.11	*2014-06-06*
1.2.17	2014-06-06
1.3.0-beta.10	*2014-05-24*
1.3.0-beta.9	*2014-05-17*
1.3.0-beta.8	*2014-05-09*
1.3.0-beta.7	*2014-04-26*
1.3.0-beta.6	*2014-04-22*
1.2.16	2014-04-04
1.3.0-beta.5	*2014-04-04*
1.3.0-beta.4	*2014-03-28*
1.2.15	2014-03-22
1.3.0-beta.3	*2014-03-21*
1.3.0-beta.2	*2014-03-15*
1.3.0-beta.1	*2014-03-08*
1.2.14	2014-03-01
1.2.13	2014-02-15
1.2.12	2014-02-08
1.2.11	2014-02-03
1.2.10	2014-01-25
1.2.9	2014-01-15
1.2.8	2014-01-10
1.2.7	2014-01-03
1.2.6	2013-12-20
1.2.5	2013-12-13
1.2.4	2013-12-06
1.2.3	2013-11-27
1.2.2	2013-11-22
1.2.1	2013-11-15
1.2.0	2013-11-08
1.2.0-rc.3	2013-10-14
1.2.0-rc.2	2013-09-04
1.0.8	2013-08-22
1.2.0rc1	2013-08-13

bellindoc.com

1.3.0-beta.19	2014-08-23
1.2.22	2014-08-12
1.3.0-beta.18	2014-08-12
1.2.21	2014-07-25
1.3.0-beta.17	2014-07-25
1.3.0-beta.16	2014-07-18
1.2.20	2014-07-11
1.3.0-beta.15	2014-07-11
1.2.19	2014-07-01
1.3.0-beta.14	2014-07-01
1.3.0-beta.13	2014-06-16
1.3.0-beta.12	2014-06-14
1.2.18	2014-06-14
1.3.0-beta.11	2014-06-06
1.2.17	2014-06-06
1.3.0-beta.10	2014-05-24
1.3.0-beta.9	2014-05-17
1.3.0-beta.8	2014-05-09
1.3.0-beta.7	2014-04-26
1.3.0-beta.6	2014-04-22
1.2.16	2014-04-04
1.3.0-beta.5	2014-04-04
1.3.0-beta.4	2014-03-28
1.2.15	2014-03-22
1.3.0-beta.3	2014-03-21
1.3.0-beta.2	2014-03-15
1.3.0-beta.1	2014-03-08
1.2.14	2014-03-01
1.2.13	2014-02-15
1.2.12	2014-02-08
1.2.11	2014-02-03
1.2.10	2014-01-25
1.2.9	2014-01-15
1.2.8	2014-01-10
1.2.7	2014-01-03
1.2.6	2013-12-20
1.2.5	2013-12-13
1.2.4	2013-12-06
1.2.3	2013-11-27
1.2.2	2013-11-22
1.2.1	2013-11-15
1.2.0	2013-11-08
1.2.0-rc.3	2013-10-14
1.2.0-rc.2	2013-09-04
1.0.8	2013-08-22
1.2.0rc1	2013-08-13

1.0.7	2013-05-22
1.1.5	2013-05-22
1.0.6	2013-04-04
1.1.4	2013-04-04
1.0.5	2013-02-20
1.1.3	2013-02-20
1.0.4	2013-01-23
1.1.2	2013-01-23
1.1.1	2012-11-27
1.0.3	2012-11-27
1.1.0	2012-09-04
1.0.2	2012-09-04
1.0.1	2012-06-25
1.0.0	2012-06-14
v1.0.0rc12	2012-06-12
v1.0.0rc11	2012-06-11
v1.0.0rc10	2012-05-24
v1.0.0rc9	2012-05-15
v1.0.0rc8	2012-05-07
v1.0.0rc7	2012-05-01
v1.0.0rc6	2012-04-21
v1.0.0rc5	2012-04-12
v1.0.0rc4	2012-04-05
v1.0.0rc3	2012-03-30
v1.0.0rc2	2012-03-21
g3-v1.0.0rc1	2012-03-14
g3-v1.0.0-rc2	2012-03-16
1.0.0rc1	2012-03-14
0.10.6	2012-01-17
0.10.5	2011-11-08
0.10.4	2011-10-23
0.10.3	2011-10-14
0.10.2	2011-10-08
0.10.1	2011-09-09
0.10.0	2011-09-02
0.9.19	2011-08-21
0.9.18	2011-07-30
0.9.17	2011-06-30
0.9.16	2011-06-08
0.9.15	2011-04-12
0.9.14	2011-04-01
0.9.13	2011-03-14
0.9.12	2011-03-04
0.9.11	2011-02-09
0.9.10	2011-01-27
0.9.9	2011-01-14

bellindoc.com

1.0.7	2013-05-22
1.1.5	2013-05-22
1.0.6	2013-04-04
1.1.4	2013-04-04
1.0.5	2013-02-20
1.1.3	2013-02-20
1.0.4	2013-01-23
1.1.2	2013-01-23
1.1.1	2012-11-27
1.0.3	2012-11-27
1.1.0	2012-09-04
1.0.2	2012-09-04
1.0.1	2012-06-25
1.0.0	2012-06-14
v1.0.0rc12	2012-06-12
v1.0.0rc11	2012-06-11
v1.0.0rc10	2012-05-24
v1.0.0rc9	2012-05-15
v1.0.0rc8	2012-05-07
v1.0.0rc7	2012-05-01
v1.0.0rc6	2012-04-21
v1.0.0rc5	2012-04-12
v1.0.0rc4	2012-04-05
v1.0.0rc3	2012-03-30
v1.0.0rc2	2012-03-21
g3-v1.0.0rc1	2012-03-14
g3-v1.0.0-rc2	2012-03-16
1.0.0rc1	2012-03-14
0.10.6	2012-01-17
0.10.5	2011-11-08
0.10.4	2011-10-23
0.10.3	2011-10-14
0.10.2	2011-10-08
0.10.1	2011-09-09
0.10.0	2011-09-02
0.9.19	2011-08-21
0.9.18	2011-07-30
0.9.17	2011-06-30
0.9.16	2011-06-08
0.9.15	2011-04-12
0.9.14	2011-04-01
0.9.13	2011-03-14
0.9.12	2011-03-04
0.9.11	2011-02-09
0.9.10	2011-01-27
0.9.9	2011-01-14

0.9.7	2010-12-11
0.9.6	2010-12-07
0.9.5	2010-11-25
0.9.4	2010-11-19
0.9.3	2010-11-11
0.9.2	2010-11-03
0.9.1	2010-10-27
0.9.0	2010-10-21

第1.1节：入门

创建一个新的HTML文件并粘贴以下内容：

```
<!DOCTYPE html>
<html ng-app>
<head>
   <title>你好，Angular</title>
   <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
</head>
<body ng-init="name='World'">
   <label>姓名</label>
   <input ng-model="name" />
   <span>你好，{{ name }}！</span>
   <p ng-bind="name"></p>
</body>
</html>
```

实时演示

当你用浏览器打开该文件时，你会看到一个输入框，后面跟着文本你好，世界！。编辑输入框中的值会实时更新文本，无需刷新整个页面。

说明：

1. 从内容分发网络加载Angular框架。

   ```
   <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
   ```

2. 使用ng-app指令将HTML文档定义为Angular应用程序

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

3. 使用ng-init初始化name变量

   ```
   <body ng-init=" name = 'World' ">
   ```

   注意，ng-init应仅用于演示和测试目的。在构建实际应用程序时，应由控制器初始化数据。

4. 将模型中的数据绑定到HTML控件的视图。使用ng-model将<input>绑定到name属性

   ```
   <input ng-model="name" />
   ```


0.9.7	2010-12-11
0.9.6	2010-12-07
0.9.5	2010-11-25
0.9.4	2010-11-19
0.9.3	2010-11-11
0.9.2	2010-11-03
0.9.1	2010-10-27
0.9.0	2010-10-21

Section 1.1: Getting Started

Create a new HTML file and paste the following content:

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title>Hello, Angular</title>
  <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
</head>
<body ng-init="name='World'">
  <label>Name</label>
  <input ng-model="name" />
  <span>Hello, {{ name }}!</span>
  <p ng-bind="name"></p>
</body>
</html>
```

Live demo

When you open the file with a browser, you will see an input field followed by the text `Hello, World!`. Editing the value in the input will update the text in real-time, without the need to refresh the whole page.

Explanation:

1. Load the Angular framework from a Content Delivery Network.

   ```
   <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
   ```

2. Define the HTML document as an Angular application with the `ng-app` directive

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

3. Initialize the `name` variable using `ng-init`

   ```
   <body ng-init=" name = 'World' ">
   ```

   Note that ng-init should be used for demonstrative and testing purposes only. When building an actual application, controllers should initialize the data.

4. Bind data from the model to the view on HTML controls. Bind an **`<input>`** to the `name` property with `ng-model`

   ```
   <input ng-model="name" />
   ```

5. 使用双大括号{{ }}显示模型内容

```
<span>你好，{{ name }}</span>
```

6. 绑定name属性的另一种方式是使用ng-bind，而不是大括号表达式"{{ }}"

```
<span ng-bind="name"></span>
```

最后三个步骤建立了双向数据绑定。对输入的更改会更新模型，且该更改会反映在视图中。

使用大括号表达式和ng-bind之间存在差异。如果使用大括号表达式，您可能会看到实际的
你好，{{name}} 在页面加载时会显示（在表达式解析之前，即数据加载之前），而如果使用ng-bind，则只有在name解析后才会显示数据。作为替代，可以使用指令ng-cloak来防止大括号表达式在编译前显示。

第1.2节：展示所有常见的Angular构造

下面的示例展示了一个文件中的常见AngularJS构造：

```
<!DOCTYPE html>
<html ng-app="myDemoApp">
  <head>
    <style>.started { background: gold; }</style>
    <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
    <script>
function MyDataService() {
        return {
getWorlds: function getWorlds() {
            return ["这个世界", "另一个世界"];
            }
          };
        }

function DemoController(worldsService) {
        var vm = this;
vm.messages = worldsService.getWorlds().map(function(w) {
          return "你好，" + w + "!";
          });
        }

function startup($rootScope, $window) {
        $window.alert("你好，用户！正在加载世界...");
          $rootScope.hasStarted = true;
        }

angular.module("myDemoApp", [/* 模块依赖项放这里 */])
         .service("worldsService", [MyDataService])
.controller("demoController", ["worldsService", DemoController])
        .config(function() {
console.log('配置应用程序');
         })
         .run(["$rootScope", "$window", startup]);
    </script>
  </head>
  <body ng-class="{ 'started': hasStarted }" ng-cloak>
    <div ng-controller="demoController as vm">
```


5. Display content from the model using double braces `{{ }}`

```
<span>Hello, {{ name }}</span>
```

6. Another way of binding the `name` property is using `ng-bind` instead of handlebars`"{{ }}"`

```
<span ng-bind="name"></span>
```

The last three steps establish the *two way data-binding*. Changes made to the input update the *model*, which is reflected in the *view*.

There is a difference between using handlebars and `ng-bind`. If you use handlebars, you might see the actual `Hello, {{name}}` as the page loads before the expression is resolved (before the data is loaded) whereas if you use `ng-bind`, it will only show the data when the name is resolved. As an alternative the directive `ng-cloak` can be used to prevent handlebars to display before it is compiled.

Section 1.2: Showcasing all common Angular constructs

The following example shows common AngularJS constructs in one file:

```
<!DOCTYPE html>
<html ng-app="myDemoApp">
  <head>
    <style>.started { background: gold; }</style>
    <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
    <script>
      function MyDataService() {
        return {
          getWorlds: function getWorlds() {
            return ["this world", "another world"];
          }
        };
      }

      function DemoController(worldsService) {
        var vm = this;
        vm.messages = worldsService.getWorlds().map(function(w) {
          return "Hello, " + w + "!";
        });
      }

      function startup($rootScope, $window) {
        $window.alert("Hello, user! Loading worlds...");
        $rootScope.hasStarted = true;
      }

      angular.module("myDemoApp", [/* module dependencies go here */])
        .service("worldsService", [MyDataService])
        .controller("demoController", ["worldsService", DemoController])
        .config(function() {
          console.log('configuring application');
        })
        .run(["$rootScope", "$window", startup]);
    </script>
  </head>
  <body ng-class="{ 'started': hasStarted }" ng-cloak>
    <div ng-controller="demoController as vm">
```

```
      <ul>
        <li ng-repeat="msg in vm.messages">{{ msg }}</li>
      </ul>
    </div>
  </body>
</html>
```

文件的每一行解释如下：

实时演示

1. ng-app="myDemoApp"，ngApp指令用于引导应用程序并告诉Angular某个DOM元素由名为"myDemoApp"的特定angular.模块控制；
2. **<script src="angular.min.js">是引导AngularJS库的第一步；**

声明了三个函数（MyDataService、DemoController和startup），它们在下面被使用（并解释）。

3. angular.module(...)当第二个参数为数组时用于创建新模块。该数组用于提供模块依赖列表。在此示例中，我们对`module(...)`函数的结果进行链式调用；
4. .service(...)创建一个Angular服务并返回模块以便链式调用；
5. .controller(...)创建一个Angular控制器并返回模块以便链式调用；
6. .config(...)使用此方法注册模块加载时需要执行的工作。
7. .run(...) 确保代码在启动时运行 并以一个项目数组作为参数。使用此方法注册工作的方法，当注入器完成加载所有模块时应执行该方法。
    - 第一项是让 Angular 知道startup函数需要注入内置的$rootScope服务作为参数；
    - 第二项是让 Angular 知道startup函数需要注入内置的$window服务作为参数；
    - 数组中的*最后*一项，`startup`，是启动时实际运行的函数；
8. ng-class是ngClass 指令，用于设置动态class，在此示例中利用了$rootScope上的hasStarted $rootScope 动态地
9. ng-cloak是一个指令，用于防止未渲染的 Angular HTML 模板（例如 "{{ msg }}"）在 Angular 完全加载应用之前被短暂显示。
10. ng-controller 是一个指令，要求 Angular 实例化一个特定名称的新控制器来协调该部分的 DOM；
11. ng-repeat 是一个指令，用于让 Angular 遍历一个集合并为每个项目克隆一个 DOM 模板；
12. {{ msg }} 展示了插值：即时渲染作用域或控制器的一部分；

第1.3节：作用域的重要性

由于 Angular 使用 HTML 来扩展网页，使用纯 Javascript 来添加逻辑，因此可以轻松地通过ng-app、ng-controller以及一些内置指令如ng-if、ng-repeat等来创建网页。使用新的controllerAs语法，Angular 新手用户可以将函数和数据附加到他们的控制器上，而不是使用$scope。

然而，迟早需要理解这个$scope到底是什么。它会不断出现于


```
      <ul>
        <li ng-repeat="msg in vm.messages">{{ msg }}</li>
      </ul>
    </div>
  </body>
</html>
```

Every line of the file is explained below:

Live Demo

1. `ng-app="myDemoApp"`, the ngApp directive that bootstraps the application and tells angular that a DOM element is controlled by a specific `angular.module` named `"myDemoApp"`;
2. **`<script src="angular.min.js">`** is the first step in bootstrapping the AngularJS library;

Three functions (`MyDataService`, `DemoController`, and `startup`) are declared, which are used (and explained) below.

3. `angular.module(...)` used with an array as the second argument creates a new module. This array is used to supply a list of module dependencies. In this example we chain calls on the result of the `module(...)` function;
4. `.service(...)` creates an Angular Service and returns the module for chaining;
5. `.controller(...)` creates an Angular Controller and returns the module for chaining;
6. `.config(...)` Use this method to register work which needs to be performed on module loading.
7. `.run(...)` makes sure code is run at startup time and takes an array of items as a parameter. Use this method to register work which should be performed when the injector is done loading all modules.
    - the first item is letting Angular know that the `startup` function requires the built-in `$rootScope` service to be injected as an argument;
    - the second item is letting Angular know that the `startup` function requires the built-in `$window` service to be injected as an argument;
    - the *last* item in the array, `startup`, is the actual function to run on startup;
8. `ng-class` is the ngClass directive to set a dynamic `class`, and in this example utilizes `hasStarted` on the `$rootScope` dynamically
9. `ng-cloak` is a directive to prevent the unrendered Angular html template (e.g. "`{{ msg }}`") to be briefly shown before Angular has fully loaded the application.
10. `ng-controller` is the directive that asks Angular to instantiate a new controller of specific name to orchestrate that part of the DOM;
11. `ng-repeat` is the directive to make Angular iterate over a collection and clone a DOM template for each item;
12. `{{ msg }}` showcases interpolation: on-the-spot rendering of a part of the scope or controller;

Section 1.3: The importance of scope

As Angular uses HTML to extend a web page and plain Javascript to add logic, it makes it easy to create a web page using **ng-app**, **ng-controller** and some built-in directives such as **ng-if**, **ng-repeat**, etc. With the new **controllerAs** syntax, newcomers to Angular users can attach functions and data to their controller instead of using `$scope`.

However, sooner or later, it is important to understand what exactly this `$scope` thing is. It will keep showing up in

举例说明，因此理解这一点很重要。

好消息是，这是一个简单而强大的概念。

当你创建以下内容时：

```
<div ng-app="myApp">
  <h1>你好 {{ name }}</h1>
</div>
```

name 变量在哪里？

答案是 Angular 创建了一个 $rootScope 对象。这只是一个普通的 Javascript 对象，因此 name 是 $rootScope 对象上的一个属性：

```
angular.module("myApp", [])
  .run(function($rootScope) {
    $rootScope.name = "World!";
  });
```

就像 Javascript 中的全局作用域一样，通常不建议向全局作用域或

$rootScope 添加项目。

当然，大多数时候，我们会创建一个控制器并将所需功能放入该控制器中。但当
我们创建控制器时，Angular 会自动创建一个 $scope 对象给该控制器。这有时被称为
本地作用域。

所以，创建以下控制器：

```
<div ng-app="myApp">
  <div ng-controller="MyController">
    <h1>你好 {{ name }}</h1>
  </div>
</div>
```

将允许通过$scope参数访问本地作用域。

```
angular.module("myApp", [])
  .controller("MyController", function($scope) {
    $scope.name = "本地先生！";
  });
```

没有$scope参数的控制器可能出于某种原因根本不需要它。但重要的是要意识到，即使使用controllerAs语法，局部作用域依然存在。

由于$scope是一个JavaScript对象，Angular会神奇地将其设置为原型继承自$rootScope。正如你可以想象的，作用域可以形成一个链。例如，你可以在父控制器中创建一个模型，并将其附加到父控制器的作用域上，形式为$scope.model。

然后通过原型链，子控制器可以在本地访问同一个模型，形式为$scope.model。

这些最初都不明显，因为这只是Angular在后台的魔法。但理解$scope是了解Angular工作原理的重要一步。

examples so it is important to have some understanding.

The good news is that it is a simple yet powerful concept.

When you create the following:

```
<div ng-app="myApp">
  <h1>Hello {{ name }}</h1>
</div>
```

Where does **name** live?

The answer is that Angular creates a `$rootScope` object. This is simply a regular Javascript object and so **name** is a property on the `$rootScope` object:

```
angular.module("myApp", [])
  .run(function($rootScope) {
    $rootScope.name = "World!";
  });
```

And just as with global scope in Javascript, it's usually not such a good idea to add items to the global scope or `$rootScope`.

Of course, most of the time, we create a controller and put our required functionality into that controller. But when we create a controller, Angular does it's magic and creates a `$scope` object for that controller. This is sometimes referred to as the **local scope**.

So, creating the following controller:

```
<div ng-app="myApp">
  <div ng-controller="MyController">
    <h1>Hello {{ name }}</h1>
  </div>
</div>
```

would allow the local scope to be accessible via the `$scope` parameter.

```
angular.module("myApp", [])
  .controller("MyController", function($scope) {
    $scope.name = "Mr Local!";
  });
```

A controller without a `$scope` parameter may simply not need it for some reason. But it is important to realize that, **even with controllerAs syntax**, the local scope exists.

As `$scope` is a JavaScript object, Angular magically sets it up to prototypically inherit from `$rootScope`. And as you can imagine, there can be a chain of scopes. For example, you could create a model in a parent controller and attach to it to the parent controller's scope as `$scope.model`.

Then via the prototype chain, a child controller could access that same model locally with `$scope.model`.

None of this is initially evident, as it's just Angular doing its magic in the background. But understanding `$scope` is an important step in getting to know how Angular works.

第1.4节：Angular中的代码压缩

什么是压缩？

这是从源代码中移除所有不必要字符而不改变其功能的过程。

普通语法

如果我们使用普通的Angular语法来编写控制器，那么在压缩文件后，它将破坏我们的功能。

控制器（压缩前）：

```
var app = angular.module('mainApp', []);
app.controller('FirstController', function($scope) {
    $scope.name= 'Hello World !';
});
```

使用压缩工具后，代码将被压缩如下。

```
var app=angular.module("mainApp",[]);app.controller("FirstController",function(e){e.name= 'Hello World !'})
```

这里，压缩移除了代码中不必要的空格和$scope变量。因此，当我们使用这个压缩后的代码时，视图上不会显示任何内容。因为$scope是控制器和视图之间的关键部分，现在被替换成了小写的'e'变量。所以当你运行应用时，会出现未知提供者'e'依赖错误。

有两种方式可以用服务名称信息注释你的代码，这两种方式都是压缩安全的：

内联注释语法

```
var app = angular.module('mainApp', []);
app.controller('FirstController', ['$scope', function($scope) {
    $scope.message = 'Hello World !';
}]);
```

$inject 属性注释语法

```
FirstController.$inject = ['$scope'];
var FirstController = function($scope) {
    $scope.message = 'Hello World !';
}

var app = angular.module('mainApp', []);
app.controller('FirstController', FirstController);
```

压缩后，此代码将变为

```
var
app=angular.module("mainApp",[]);app.controller("FirstController",["$scope",function(a){a.message="Hello World !"}]);
```

这里，angular 会将变量 'a' 视为 $scope，并且它将显示输出为 'Hello World !'。

Section 1.4: Minification in Angular

What is Minification ?

It is the process of removing all unnecessary characters from source code without changing its functionality.

Normal Syntax

If we use normal angular syntax for writing a controller then after minifiying our files it going to break our functionality.

Controller (Before minification) :

```
var app = angular.module('mainApp', []);
app.controller('FirstController', function($scope) {
    $scope.name= 'Hello World !';
});
```

After using minification tool, It will be minified as like below.

```
var app=angular.module("mainApp",[]);app.controller("FirstController",function(e){e.name= 'Hello World !'})
```

Here, minification removed unnecessary spaces and the $scope variable from code. So when we use this minified code then its not going to print anything on view. Because $scope is a crucial part between controller and view, which is now replaced by the small 'e' variable. So when you run the application it is going to give Unknown Provider 'e' dependency error.

There are two ways of annotating your code with service name information which are minification safe:

Inline Annotation Syntax

```
var app = angular.module('mainApp', []);
app.controller('FirstController', ['$scope', function($scope) {
    $scope.message = 'Hello World !';
}]);
```

$inject Property Annotation Syntax

```
FirstController.$inject = ['$scope'];
var FirstController = function($scope) {
    $scope.message = 'Hello World !';
}

var app = angular.module('mainApp', []);
app.controller('FirstController', FirstController);
```

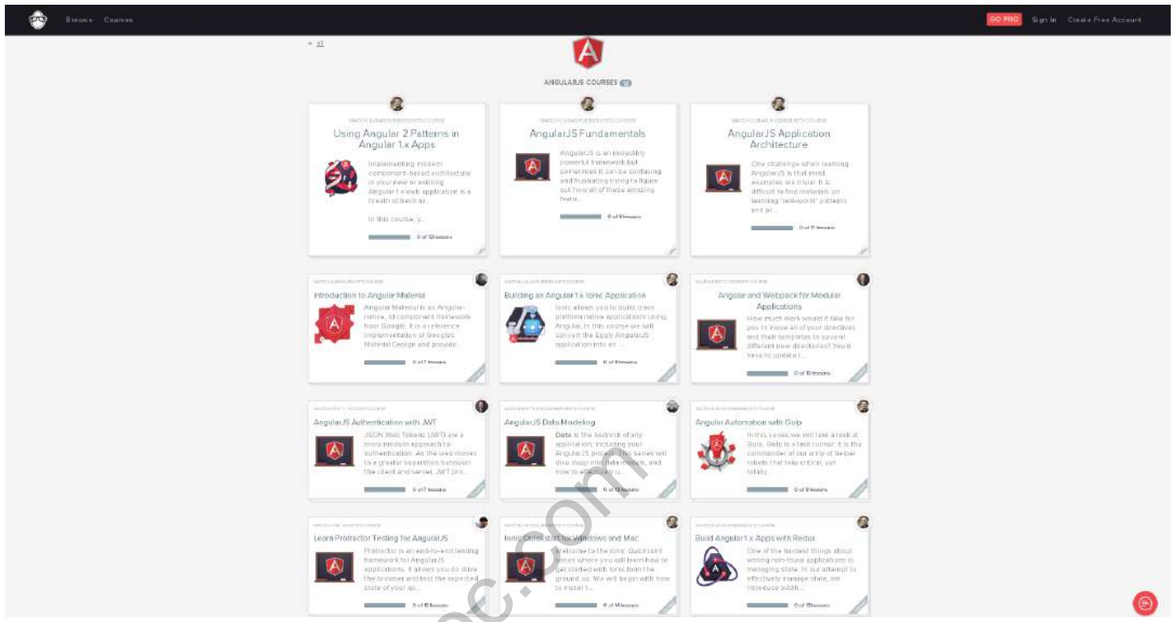
After minification, this code will be

```
var
app=angular.module("mainApp",[]);app.controller("FirstController",["$scope",function(a){a.message="Hello World !"}]);
```

Here, angular will consider variable 'a' to be treated as $scope, and It will display output as 'Hello World !'.

第1.5节：AngularJS入门视频教程

在 egghead.io 上有很多关于 AngularJS 框架的优质视频教程



- https://egghead.io/courses/angularjs-app-from-scratch-getting-started
- https://egghead.io/courses/angularjs-application-architecture
- https://egghead.io/courses/angular-material-introduction
- https://egghead.io/courses/building-an-angular-1-x-ionic-application
- https://egghead.io/courses/angular-and-webpack-for-modular-applications
- https://egghead.io/courses/angularjs-authentication-with-jwt
- https://egghead.io/courses/angularjs-data-modeling
- https://egghead.io/courses/angular-automation-with-gulp
- https://egghead.io/courses/learn-protractor-testing-for-angularjs
- https://egghead.io/courses/ionic-quickstart-for-windows
- https://egghead.io/courses/build-angular-1-x-apps-with-redux
- https://egghead.io/courses/using-angular-2-patterns-in-angular-1-x-apps

第1.6节：最简单的 Angular Hello World

Angular 1 本质上是一个 DOM 编译器。我们可以传入 HTML，无论是作为模板还是普通网页，
然后让它编译成一个应用程序。

我们可以告诉 Angular 将页面的某个区域视为一个表达式，使用{{ }}的 handlebars 风格语法。大括号内的任何内容都会被编译，如下所示：

```
{{ 'Hello' + 'World' }}
```

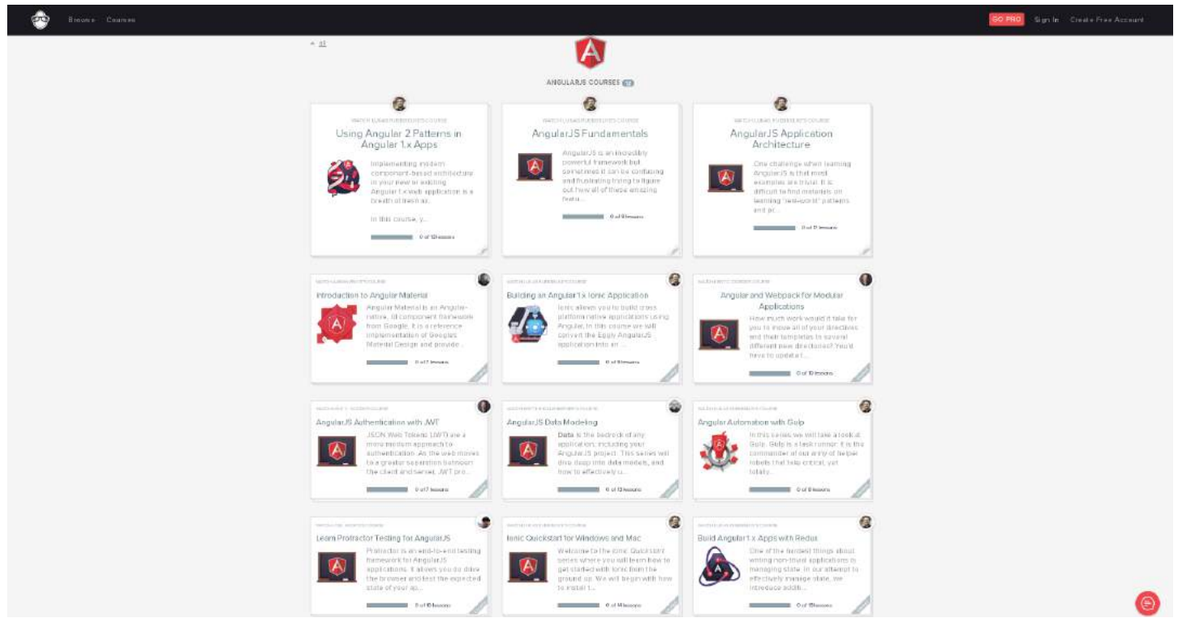
这将输出：

```
HelloWorld
```

ng-app

Section 1.5: AngularJS Getting Started Video Tutorials

There are a lot of good video tutorials for the AngularJS framework on egghead.io



- https://egghead.io/courses/angularjs-app-from-scratch-getting-started
- https://egghead.io/courses/angularjs-application-architecture
- https://egghead.io/courses/angular-material-introduction
- https://egghead.io/courses/building-an-angular-1-x-ionic-application
- https://egghead.io/courses/angular-and-webpack-for-modular-applications
- https://egghead.io/courses/angularjs-authentication-with-jwt
- https://egghead.io/courses/angularjs-data-modeling
- https://egghead.io/courses/angular-automation-with-gulp
- https://egghead.io/courses/learn-protractor-testing-for-angularjs
- https://egghead.io/courses/ionic-quickstart-for-windows
- https://egghead.io/courses/build-angular-1-x-apps-with-redux
- https://egghead.io/courses/using-angular-2-patterns-in-angular-1-x-apps

Section 1.6: The Simplest Possible Angular Hello World

Angular 1 is at heart a DOM compiler. We can pass it HTML, either as a template or just as a regular web page, and then have it compile an app.

We can tell Angular to treat a region of the page as an *expression* using the `{{ }}` handlebars style syntax. Anything between the curly braces will be compiled, like so:

```
{{ 'Hello' + 'World' }}
```

This will output:

```
HelloWorld
```

ng-app

我们通过ng-app *指令*告诉 Angular 应该将 DOM 的哪一部分视为主模板。指令是一个自定义属性或元素，Angular 模板编译器知道如何处理它。现在让我们添加一个 ng-app 指令：

```
<html>
  <head>
    <script src="/angular.js"></script>
  </head>
  <body ng-app>
{{ 'Hello' + 'World' }}
  </body>
</html>
```

我现在已经告诉 body 元素作为根模板。它里面的任何内容都会被编译。

指令

指令是编译器指令。它们扩展了 Angular DOM 编译器的功能。这就是为什么 Angular 的创建者Misko描述 Angular 为：

> “如果网页浏览器是为网络应用而构建的，那会是什么样子。”

我们实际上创建了新的 HTML 属性和元素，并让 Angular 将它们编译成应用程序。 ng-app 是一个简单地开启编译器的指令。其他指令包括：

- ng-click，添加点击处理器，
- ng-hide，有条件地隐藏元素，和
- <form>，为标准 HTML 表单元素添加额外行为。

Angular 自带大约 100 个内置指令，允许你完成大多数常见任务。我们也可以编写自己的指令，这些指令将以与内置指令相同的方式处理。

我们通过一系列指令构建 Angular 应用，并用 HTML 将它们连接起来。

We tell Angular which portion of our DOM to treat as the master template using the ng-app *directive*. A directive is a custom attribute or element that the Angular template compiler knows how to deal with. Let's add an ng-app directive now:

```
<html>
  <head>
    <script src="/angular.js"></script>
  </head>
  <body ng-app>
    {{ 'Hello' + 'World' }}
  </body>
</html>
```

I've now told the body element to be the root template. Anything in it will be compiled.

Directives

Directives are compiler directives. They extend the capabilities of the Angular DOM compiler. This is why **Misko**, the creator of Angular, describes Angular as:

> "What a web browser would have been had it been built for web applications.

We literally create new HTML attributes and elements, and have Angular compile them into an app. ng-app is a directive that simply turns on the compiler. Other directives include:

- ng-click, which adds a click handler,
- ng-hide, which conditionally hides an element, and
- <form>, which adds additional behaviour to a standard HTML form element.

Angular comes with around 100 built-in directives which allow you to accomplish most common tasks. We can also write our own, and these will be treated in the same way as the built in directives.

We build an Angular app out of a series of directives, wired together with HTML.

第2章：模块

第2.1节：模块

模块作为应用程序中不同部分的容器，例如控制器、服务、过滤器、指令等。
模块可以通过Angular的依赖注入机制被其他模块引用。

创建模块：

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

上例中传入的数组[]是app依赖的模块列表，如果没有依赖，则传入空数组，即[]。

将一个模块作为另一个模块的依赖注入：

```
angular.module('app', [
    'app.auth',
    'app.dashboard'
]);
```

引用模块：

```
angular
.module('app');
```

第2.2节：模块

模块是应用程序中各种部分的容器——控制器、服务、过滤器、指令等。

为什么使用模块
大多数应用程序都有一个主方法，用于实例化并连接应用程序的不同部分。
Angular应用程序没有主方法。
但在AngularJS中，声明式过程易于理解，并且可以将代码打包为可重用的模块。
模块可以按任意顺序加载，因为模块会延迟执行。

声明一个模块

```
var app = angular.module('我的应用', []);
// 空数组是 myApp 依赖的模块列表。
// 如果有任何必需的依赖，
// 你可以在模块中添加，比如 ['ngAnimate']

app.controller('我的控制器', function() {

  // 在这里编写你的业务逻辑
});
```

模块加载和依赖

1. 配置块：在提供者和配置阶段执行。

```
angular.module('我的模块', []).
```


Chapter 2: Modules

Section 2.1: Modules

Module serves as a container of different parts of your app such as controllers, services, filters, directives, etc.
Modules can be referenced by other modules through Angular's dependency injection mechanism.

Creating a module:

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

Array [ ] passed in above example is the *list of modules* app depends on, if there are no dependencies then we pass Empty Array i.e. [ ].

Injecting a module as a dependency of another module:

```
angular.module('app', [
    'app.auth',
    'app.dashboard'
]);
```

Referencing a module:

```
angular
    .module('app');
```

Section 2.2: Modules

Module is a container for various parts of your applications - controller, services, filters, directive, etc.

Why to use Modules
Most applications have a main method that instantiates and wires together the different parts of the application.
Angular apps don't have main method.
But in AngularJS the declarative process is easy to understand and one can package code as reusable modules.
Modules can be loaded in any order because modules delay execution.

declare a module

```
var app = angular.module('myApp', []);
// Empty array is list of modules myApp is depends on.
// if there are any required dependancies,
// then you can add in module, Like ['ngAnimate']

app.controller('myController', function() {

  // write your business logic here
});
```

Module Loading and Dependencies

1. Configuration Blocks: get executed during provider and configuration phase.

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

```
config(function(可注入项) {
  // 这里你只能向配置块注入提供者。
});
```

2.运行块：在注入器创建后执行，用于启动应用程序。

```
angular.module('我的模块', []).
run(function(injectables) {
  // 这里你只能向配置块注入实例。
});
```

belindoc.com


```
config(function(injectables) {
  // here you can only inject providers in to config blocks.
});
```

2. Run Blocks: get executed after the injector is created and are used to start the application.

```
angular.module('myModule', []).
run(function(injectables) {
  // here you can only inject instances in to config blocks.
});
```

第3章：组件

参数	详情
=	用于双向数据绑定。这意味着如果你在组件作用域中更新该变量，变化将反映到父作用域中。
<	单向绑定，当我们只想从父作用域读取值而不更新它时使用。
@	字符串参数。
&	当您的组件需要向其父作用域输出内容时的回调。
-	-
生命周期钩子	详情（需要 angular.version >= 1.5.3）
$onInit()	在每个控制器构造完成且其绑定初始化后调用。这是放置控制器初始化代码的好地方。
$onChanges(changesObj)	每当单向绑定更新时调用。 changesObj 是一个哈希，其键是已更改绑定属性的名称，值是形如 { currentValue, previousValue, isFirstChange() } 的对象。
$onDestroy()	当其包含的作用域被销毁时调用控制器。使用此钩子来释放外部资源、监视器和事件处理程序。
$postLink()	在该控制器的元素及其子元素完成链接后调用。此钩子可以被视为类似于Angular 2中的ngAfterViewInit和ngAfterContentInit钩子。
$doCheck()	在每次消化周期中调用。提供检测和响应变化的机会。你希望对检测到的变化采取的任何操作都必须从此钩子中调用；实现此钩子不会影响$onChanges的调用时机。

第3.1节：基本组件和生命周期钩子

什么是组件？

- 组件基本上是一种指令，使用更简单的配置，适合基于组件的架构，这正是Angular 2的核心理念。可以将组件看作一个小部件：一段HTML代码，可以在你的网页应用的多个不同位置重复使用。

组件

```
angular.module('myApp', [])
    .component('helloWorld', {
        template: '<span>Hello World!</span>'
    });
```

标记

```
<div ng-app="myApp">
  <hello-world> </hello-world>
</div>
```

实时演示

在组件中使用外部数据：

我们可以添加一个参数来传递一个名称给我们的组件，使用方法如下：

Chapter 3: Components

Parameter	Details
=	For using two-way data binding. This means that if you update that variable in your component scope, the change will be reflected on the parent scope.
<	One-way bindings when we just want to read a value from a parent scope and not update it.
@	String parameters.
&	For callbacks in case your component needs to output something to its parent scope.
-	-
LifeCycle Hooks	**Details** (requires angular.version >= 1.5.3)
$onInit()	Called on each controller after all the controllers on an element have been constructed and had their bindings initialized. This is a good place to put initialization code for your controller.
$onChanges(changesObj)	Called whenever one-way bindings are updated. The `changesObj` is a hash whose keys are the names of the bound properties that have changed, and the values are an object of the form `{ currentValue, previousValue, isFirstChange() }`.
$onDestroy()	Called on a controller when its containing scope is destroyed. Use this hook for releasing external resources, watches and event handlers.
$postLink()	Called after this controller's element and its children have been linked. This hook can be considered analogous to the ngAfterViewInit and ngAfterContentInit hooks in Angular 2.
$doCheck()	Called on each turn of the digest cycle. Provides an opportunity to detect and act on changes. Any actions that you wish to take in response to the changes that you detect must be invoked from this hook; implementing this has no effect on when $onChanges is called.

Section 3.1: Basic Components and LifeCycle Hooks

What's a component?

- A component is basically a directive that uses a simpler configuration and that is suitable for a component-based architecture, which is what Angular 2 is all about. Think of a component as a widget: A piece of HTML code that you can reuse in several different places in your web application.

Component

```
angular.module('myApp', [])
    .component('helloWorld', {
        template: '<span>Hello World!</span>'
    });
```

Markup

```
<div ng-app="myApp">
  <hello-world> </hello-world>
</div>
```

Live Demo

Using External data in Component:

We could add a parameter to pass a name to our component, which would be used as follows:

```
angular.module("myApp", [])
  .component("helloWorld",{
    template: '<span>Hello {{$ctrl.name}}!</span>',
    bindings: { name: '@' }
  });
```

标记

```
<div ng-app="myApp">
  <hello-world name="'John'" > </hello-world>
</div>
```

实时演示

在组件中使用控制器

让我们来看一下如何给它添加一个控制器。

```
angular.module("myApp", [])
.component("helloWorld",{
template: "Hello {{$ctrl.name}}, 我是 {{$ctrl.myName}}!",
      bindings: { name: '@' },
controller: function(){
        this.myName = 'Alain';
      }
  });
```

标记

```
<div ng-app="myApp">
  <hello-world name="John"> </hello-world>
</div>
```

CodePen 演示

传递给组件的参数在控制器的作用域中可用，正好在 Angular 调用其 $onInit 函数之前。考虑以下示例：

```
angular.module("myApp", [])
.component("helloWorld",{
template: "Hello {{$ctrl.name}}, 我是 {{$ctrl.myName}}!",
      bindings: { name: '@' },
controller: function(){
        this.$onInit = function() {
          this.myName = "Mac" + this.name;
        }
      }
  });
```

在上述模板中，这将渲染为“Hello John，我是 MacJohn!”。

注意，如果未指定 controllerAs，$ctrl 是 Angular 的默认值。

实时演示


```
angular.module("myApp", [])
  .component("helloWorld",{
    template: '<span>Hello {{$ctrl.name}}!</span>',
    bindings: { name: '@' }
  });
```

Markup

```
<div ng-app="myApp">
  <hello-world name="'John'" > </hello-world>
</div>
```

Live Demo

Using Controllers in Components

Let's take a look at how to add a controller to it.

```
angular.module("myApp", [])
  .component("helloWorld",{
      template: "Hello {{$ctrl.name}}, I'm {{$ctrl.myName}}!",
      bindings: { name: '@' },
      controller: function(){
        this.myName = 'Alain';
      }
  });
```

Markup

```
<div ng-app="myApp">
  <hello-world name="John"> </hello-world>
</div>
```

CodePen Demo

Parameters passed to the component are available in the controller's scope just before its `$onInit` function gets called by Angular. Consider this example:

```
angular.module("myApp", [])
  .component("helloWorld",{
      template: "Hello {{$ctrl.name}}, I'm {{$ctrl.myName}}!",
      bindings: { name: '@' },
      controller: function(){
        this.$onInit = function() {
          this.myName = "Mac" + this.name;
        }
      }
  });
```

In the template from above, this would render "Hello John, I'm MacJohn!".

Note that `$ctrl` is the Angular default value for `controllerAs` if one is not specified.

Live Demo

将“require”用作对象

在某些情况下，您可能需要在您的组件内部访问父组件的数据。

这可以通过指定我们的组件需要该父组件来实现，require 会给我们一个对所需组件控制器的引用，然后可以在我们的控制器中使用，如下面的示例所示：

> 请注意，所需的控制器只有在 $onInit 钩子之后才保证已准备好。

```
angular.module("myApp", [])
.component("helloWorld",{
template: "Hello {{$ctrl.name}}, 我是 {{$ctrl.myName}}!",
       bindings: { name: '@' },
require: {
parent: '^parentComponent'
       },
controller: function () {
          // 这里 this.parent 可能尚未初始化

          this.$onInit = function() {
              // 在 $onInit 之后，使用 this.parent 访问所需的控制器
              this.parent.foo();
          }

       }
  });
```

但请记住，这会在子组件和父组件之间创建一种紧耦合关系。

第3.2节：Angular JS中的组件

AngularJS 中的组件可以被视为自定义指令（<html> 是 HTML 指令，类似这样的将是自定义指令 <ANYTHING >）。组件包含视图和控制器。
控制器包含与用户所见视图绑定的业务逻辑。组件与 Angular 指令不同，因为它包含更少的配置。Angular 组件可以这样定义。

```
angular.module("myApp",[]).component("customer", {})
```

组件定义在 Angular 模块上。它包含两个参数，一个是组件的名称，另一个是一个对象，包含键值对，用于定义将使用哪个视图和哪个控制器，如下所示。

```
angular.module("myApp",[]).component("customer", {
    templateUrl : "customer.html", // 你的视图
    controller: customerController, // 你的控制器
    controllerAs: "cust"         // 控制器的别名
})
```

"myApp" 是我们正在构建的应用名称，customer 是我们组件的名称。现在在主 HTML 文件中调用它，只需这样写

```
<customer></customer>
```

现在这个指令将被你指定的视图替换，并且你在控制器中编写的业务逻辑也会生效。

Using “require” as an Object

In some instances you may need to access data from a parent component inside your component.

This can be achieved by specifying that our component requires that parent component, the require will give us reference to the required component controller, which can then be used in our controller as shown in the example below:

> Notice that required controllers are guaranteed to be ready only after the $onInit hook.

```
angular.module("myApp", [])
  .component("helloWorld",{
      template: "Hello {{$ctrl.name}}, I'm {{$ctrl.myName}}!",
      bindings: { name: '@' },
      require: {
        parent: '^parentComponent'
      },
      controller: function () {
        // here this.parent might not be initiated yet

        this.$onInit = function() {
           // after $onInit, use this.parent to access required controller
           this.parent.foo();
        }

      }
  });
```

Keep in mind, though, that this creates a tight coupling between the child and the parent.

Section 3.2: Components In angular JS

The components in angularJS can be visualised as a custom directive (< html > this in an HTML directive, and something like this will be a custom directive < ANYTHING >). A component contains a view and a controller. Controller contains the business logic which is binded with an view , which the user sees. The component differs from a angular directive because it contains less configuration. An angular component can be defined like this.

```
angular.module("myApp",[]).component("customer", {})
```

Components are defined on the angular modules. They contains two arguments, One is the name of the component and second one is a object which contains key value pair, which defines which view and which controller it is going to use like this .

```
angular.module("myApp",[]).component("customer", {
    templateUrl : "customer.html", // your view here
    controller: customerController, //your controller here
    controllerAs: "cust"         //alternate name for your controller
})
```

"myApp" is the name of the app we are building and customer is the name of our component. Now for calling it in main html file we will just put it like this

```
<customer></customer>
```

Now this directive will be replaced by the view you have specified and the business logic you have written in your

控制器。

注意：记住组件（component）以对象作为第二个参数，而指令（directive）以工厂函数作为参数。

belindoc.com

controller.

NOTE : Remember component take a object as second argument while directive take a factory function as argument.

第4章：内置指令

第4.1节：Angular表达式 - 文本与数字

本示例演示了在输入元素使用 type="text"和 type="number"时，Angular表达式如何被求值。请参考以下控制器和视图：

控制器

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

app.controller('ctrl', function($scope) {
     $scope.textInput = {
value: '5'
     };
$scope.numberInput = {
          value: 5
     };
 });
```

查看

```
<div ng-app="app" ng-controller="ctrl">
    <input type="text" ng-model="textInput.value">
   {{ textInput.value + 5 }}
    <input type="number" ng-model="numberInput.value">
   {{ numberInput.value + 5 }}
</div>
```

- 当在绑定到text输入的表达式中使用+时，运算符将**连接**字符串（第一个示例），屏幕上显示55*。
- 当在绑定到number输入的表达式中使用+时，运算符返回数字的**和**（第二个示例），屏幕上显示10*。

* - 直到用户更改输入字段中的值，显示内容才会相应变化。

工作示例

第4.2节：ngIf

ng-if 是一个类似于ng-show的指令，但它是将元素插入或移除DOM，而不仅仅是隐藏元素。Angular 1.1.5引入了ng-If指令。你可以在1.1.5及以上版本使用ng-if指令。这很有用，因为Angular不会对被移除的ng-**if**元素内部进行消化处理，从而减少了Angular的工作量，特别是对于复杂的数据绑定。

与ng-show不同，ng-if指令会创建一个使用原型继承的子作用域。这意味着在子作用域上设置原始值不会影响父作用域。要在父作用域上设置原始值，必须使用子作用域上的$parent属性。

JavaScript

```
angular.module('我的应用', []);

angular.module('我的应用').controller('我的控制器', ['$scope', '$window', function
我的控制器($scope, $window) {
$scope.当前用户= $window.localStorage.getItem('用户名');
```


Chapter 4: Built-in directives

Section 4.1: Angular expressions - Text vs. Number

This example demonstrates how Angular expressions are evaluated when using `type="text"` and `type="number"` for the input element. Consider the following controller and view:

Controller

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

app.controller('ctrl', function($scope) {
    $scope.textInput = {
        value: '5'
    };
    $scope.numberInput = {
        value: 5
    };
});
```

View

```
<div ng-app="app" ng-controller="ctrl">
    <input type="text" ng-model="textInput.value">
    {{ textInput.value + 5 }}
    <input type="number" ng-model="numberInput.value">
    {{ numberInput.value + 5 }}
</div>
```

- When using + in an expression bound to *text* input, the operator will **concatenate** the strings (first example), displaying 55 on the screen*.
- When using + in an expression bound to *number* input, the operator return the **sum** of the numbers (second example), displaying 10 on the screen*.

* - That is until the user changes the value in the input field, afterward the display will change accordingly.

Working Example

Section 4.2: ngIf

ng-if is a directive similar to **ng-show** but inserts or removes the element from the DOM instead of simply hiding it. Angular 1.1.5 introduced ng-If directive. You can Use ng-if directive above 1.1.5 versions. This is useful because Angular will not process digests for elements inside a removed ng-**if** reducing the workload of Angular especially for complex data bindings.

Unlike ng-show, the ng-**if** directive creates a child scope which uses prototypal inheritance. This means that setting a primitive value on the child scope will not apply to the parent. To set a primitive on the parent scope the $parent property on the child scope will have to be used.

JavaScript

```
angular.module('MyApp', []);

angular.module('MyApp').controller('myController', ['$scope', '$window', function
myController($scope, $window) {
    $scope.currentUser= $window.localStorage.getItem('userName');
```

```
}]);
```

查看

```
<div ng-controller="我的控制器">
    <div ng-if="当前用户">
       你好，{{当前用户}}
    </div>
    <div ng-if="!当前用户">
        <a href="/login">登录</a>
        <a href="/register">注册</a>
    </div>
</div>
```

DOM 如果当前用户不为未定义

```
<div ng-controller="我的控制器">
    <div ng-if="当前用户">
       你好，{{当前用户}}
    </div>
    <!-- ng-if: !当前用户 -->
</div>
```

DOM 如果当前用户为未定义

```
<div ng-controller="我的控制器">
    <!-- ng-if: currentUser -->
    <div ng-if="!currentUser">
        <a href="/login">登录</a>
        <a href="/register">注册</a>
    </div>
</div>
```

工作示例

函数 Promise

ngIf 指令也接受函数，逻辑上要求返回 true 或 false。

```
<div ng-if="myFunction()">
    <span>Span 文本</span>
</div>
```

只有当函数返回 true 时，span 文本才会显示。

```
$scope.myFunction = function() {
    var result = false;
    // 用于确定 result 布尔值的代码
    return result;
};
```

作为任何Angular表达式，该函数接受任何类型的变量。

第4.3节：ngCloak

> ngCloak 指令用于防止浏览器在应用加载时短暂显示Angular HTML模板的原始（未编译）形式。- 查看源代码

HTML


```
}]);
```

View

```
<div ng-controller="myController">
    <div ng-if="currentUser">
        Hello, {{currentUser}}
    </div>
    <div ng-if="!currentUser">
        <a href="/login">Log In</a>
        <a href="/register">Register</a>
    </div>
</div>
```

DOM If `currentUser` Is Not Undefined

```
<div ng-controller="myController">
    <div ng-if="currentUser">
        Hello, {{currentUser}}
    </div>
    <!-- ng-if: !currentUser -->
</div>
```

DOM If `currentUser` Is Undefined

```
<div ng-controller="myController">
    <!-- ng-if: currentUser -->
    <div ng-if="!currentUser">
        <a href="/login">Log In</a>
        <a href="/register">Register</a>
    </div>
</div>
```

Working Example

Function Promise

The ngIf directive accepts functions as well, which logically require to return true or false.

```
<div ng-if="myFunction()">
    <span>Span text</span>
</div>
```

The span text will only appear if the function returns true.

```
$scope.myFunction = function() {
    var result = false;
    // Code to determine the boolean value of result
    return result;
};
```

As any Angular expression the function accepts any kind of variables.

Section 4.3: ngCloak

> The `ngCloak` directive is used to prevent the Angular html template from being briefly displayed by the browser in its raw (uncompiled) form while your application is loading. - View source

HTML

GoalKicker.com

```
<div ng-cloak>
  <h1>你好 {{ name }}</h1>
</div>
```

ngCloak 可以应用于body元素，但推荐的用法是在页面的小部分区域应用多个ngCloak指令，以允许浏览器视图的渐进式渲染。

ngCloak 指令没有参数。

另见：防止闪烁

第4.4节：ngRepeat

ng-repeat 是Angular内置指令，允许你遍历数组或对象，并能为集合中的每个项目重复渲染一个元素。

ng-repeat 一个数组

```
<ul>
    <li ng-repeat="item in itemCollection">
      {{item.Name}}
    </li>
</ul>
```

说明：
item = 集合中的单个项目
itemCollection = 你正在迭代的数组

ng-repeat 一个对象

```
<ul>
    <li ng-repeat="(key, value) in myObject">
       {{key}} : {{value}}
    </li>
</ul>
```

说明：
key = 属性名
value = 属性值
myObject = 你正在迭代的对象

通过用户输入过滤你的 ng-repeat

```
<input type="text" ng-model="searchText">
<ul>
    <li ng-repeat="string in stringArray | filter:searchText">
       {{string}}
    </li>
</ul>
```

说明：
searchText = 用户想用来过滤列表的文本
stringArray = 字符串数组，例如 `['string', 'array']`

你也可以通过给过滤输出赋予别名 as 来在其他地方显示或引用过滤后的项目


```
<div ng-cloak>
  <h1>Hello {{ name }}</h1>
</div>
```

ngCloak can be applied to the body element, but the preferred usage is to apply multiple ngCloak directives to small portions of the page to permit progressive rendering of the browser view.

The ngCloak directive has no parameters.

See also: Preventing flickering

Section 4.4: ngRepeat

ng-repeat is a built in directive in Angular which lets you iterate an array or an object and gives you the ability to repeat an element once for each item in the collection.

ng-repeat an array

```
<ul>
    <li ng-repeat="item in itemCollection">
       {{item.Name}}
    </li>
</ul>
```

Where:
item = individual item in the collection
itemCollection = The array you are iterating

ng-repeat an object

```
<ul>
    <li ng-repeat="(key, value) in myObject">
       {{key}} : {{value}}
    </li>
</ul>
```

Where:
key = the property name
value = the value of the property
myObject = the object you are iterating

filter your ng-repeat by user input

```
<input type="text" ng-model="searchText">
<ul>
    <li ng-repeat="string in stringArray | filter:searchText">
       {{string}}
    </li>
</ul>
```

Where:
searchText = the text that the user wants to filter the list by
stringArray = an array of strings, e.g. `['string', 'array']`

You can also display or reference the filtered items elsewhere by assigning the filter output an alias with as

aliasName，示例如下：

```
<input type="text" ng-model="searchText">
<ul>
    <li ng-repeat="string in stringArray | filter:searchText as filteredStrings">
      {{string}}
    </li>
</ul>
<p>共有 {{filteredStrings.length}} 个匹配结果</p>
```

ng-repeat-start 和 ng-repeat-end

要通过定义起点和终点来重复多个 DOM 元素，可以使用 ng-repeat-start 和 ng-repeat-end 指令。

```
<ul>
    <li ng-repeat-start="item in [{a: 1, b: 2}, {a: 3, b:4}]">
       {{item.a}}
    </li>
    <li ng-repeat-end>
       {{item.b}}
    </li>
</ul>
```

输出：

1234始终关闭是很重要的

-
-
-
-

ng-repeat-start 与 ng-repeat-end。

变量

ng-repeat 也会在表达式内部暴露这些变量

变量类型	详情
$index数字	等于当前迭代的索引（$index===0 在第一次迭代时会返回 true
元素，参见 $first）	
$first布尔值	在第一个迭代元素时返回 true
$last	布尔值 在最后一个迭代元素时返回 true
$middle	布尔值，当元素位于 $first 和 $last
$even之间时为真	布尔值，在偶数次迭代时为真（等同于 $index%2===0）
$odd	布尔值，在奇数次迭代时为真（等同于 $index%2===1）

性能考虑

渲染 ngRepeat 可能会变慢，尤其是在使用大型集合时。

如果集合中的对象有标识符属性，您应始终通过 track by 标识符来跟踪，而不是默认的整个对象。如果没有标识符，您可以始终使用内置的 $index。

```
<div ng-repeat="item in itemCollection track by item.id">
<div ng-repeat="item in itemCollection track by $index">
```


aliasName, like so:

```
<input type="text" ng-model="searchText">
<ul>
    <li ng-repeat="string in stringArray | filter:searchText as filteredStrings">
       {{string}}
    </li>
</ul>
<p>There are {{filteredStrings.length}} matching results</p>
```

ng-repeat-start and ng-repeat-end

To repeat multiple DOM elements by defining a start and an end point you can use the ng-repeat-start and ng-repeat-end directives.

```
<ul>
    <li ng-repeat-start="item in [{a: 1, b: 2}, {a: 3, b:4}]">
        {{item.a}}
    </li>
    <li ng-repeat-end>
        {{item.b}}
    </li>
</ul>
```

Output:

- 1
- 2
- 3
- 4

It is important to always close ng-repeat-start with ng-repeat-end.

Variables

ng-repeat also exposes these variables inside the expression

Variable	Type	Details
$index	Number	Equals to the index of the current iteration ($index===0 will evaluate to true at the first iterated element; see $first)
$first	Boolean	Evaluates to true at the first iterated element
$last	Boolean	Evaluates to true at the last iterated element
$middle	Boolean	Evaluates to true if the element is between the $first and $last
$even	Boolean	Evaluates to true at an even numbered iteration (equivalent to $index%2===0)
$odd	Boolean	Evaluates to true at an odd numbered iteration (equivalent to $index%2===1)

Performance considerations

Rendering ngRepeat can become slow, especially when using large collections.

If the objects in the collection have an identifier property, you should always track by the identifier instead of the whole object, which is the default functionality. If no identifier is present, you can always use the built-in $index.

```
<div ng-repeat="item in itemCollection track by item.id">
<div ng-repeat="item in itemCollection track by $index">
```


belindoc.com

ngRepeat 的作用范围

ngRepeat 总是会创建一个孤立的子作用域，因此如果需要访问父作用域，必须小心处理在重复内部。

这里有一个简单的例子，展示了如何从点击事件中设置父作用域的值 ngRepeat。

```
scope val:  {{val}}<br/>
ctrlAs val: {{ctrl.val}}
<ul>
    <li ng-repeat="item in itemCollection">
        <a href="#" ng-click="$parent.val=item.value; ctrl.val=item.value;">
            {{item.label}} {{item.value}}
        </a>
    </li>
</ul>

$scope.val = 0;
this.val = 0;

$scope.itemCollection = [{
    id: 0,
value: 4.99,
    label: '足球'
},
{
id: 1,
value: 6.99,
    label: '棒球'
},
{
id: 2,
value: 9.99,
    label: '篮球'
}];
```

如果只有 val = item.value 在 ng-click 上，它不会更新父作用域中的 val，因为存在隔离的作用域。这就是为什么需要通过 $parent 引用或使用 controllerAs 语法（例如 ng-controller="mainController as ctrl"）来访问父作用域的原因。

嵌套的 ng-repeat

你也可以使用嵌套的 ng-repeat。

```
<div ng-repeat="values in test">
    <div ng-repeat="i in values">
     [{{$parent.$index}},{{$index}}] {{i}}
    </div>
</div>

var app = angular.module("myApp", []);
app.controller("ctrl", function($scope) {
  $scope.test = [
['a', 'b', 'c'],
    ['d', 'e', 'f']
  ];
});
```


Scope of ngRepeat

ngRepeat will always create an isolated child scope so care must be taken if the parent scope needs to be accessed inside the repeat.

Here is a simple example showing how you can set a value in your parent scope from a click event inside of ngRepeat.

```
scope val:  {{val}}<br/>
ctrlAs val: {{ctrl.val}}
<ul>
    <li ng-repeat="item in itemCollection">
        <a href="#" ng-click="$parent.val=item.value; ctrl.val=item.value;">
            {{item.label}} {{item.value}}
        </a>
    </li>
</ul>

$scope.val = 0;
this.val = 0;

$scope.itemCollection = [{
    id: 0,
    value: 4.99,
    label: 'Football'
},
{
    id: 1,
    value: 6.99,
    label: 'Baseball'
},
{
    id: 2,
    value: 9.99,
    label: 'Basketball'
}];
```

If there was only val = item.value at ng-click it won't update the val in the parent scope because of the isolated scope. That's why the parent scope is accessed with $parent reference or with the controllerAs syntax (e.g. ng-controller="mainController as ctrl").

Nested ng-repeat

You can also use nested ng-repeat.

```
<div ng-repeat="values in test">
    <div ng-repeat="i in values">
      [{{$parent.$index}},{{$index}}] {{i}}
    </div>
</div>

var app = angular.module("myApp", []);
app.controller("ctrl", function($scope) {
  $scope.test = [
    ['a', 'b', 'c'],
    ['d', 'e', 'f']
  ];
});
```


belindoc.com

在这里访问子 ng-repeat 内父 ng-repeat 的索引，可以使用$parent.$index。

第4.5节：内置指令速查表

ng-app 设置 AngularJS 区域。

ng-init 设置默认变量值。

ng-bind {{ }} 模板的替代方案。

ng-bind-template 将多个表达式绑定到视图。

ng-non-bindable 表示数据不可绑定。

ng-bind-html 绑定 HTML 元素的内部 HTML 属性。

ng-change 当用户更改输入时，计算指定表达式。

ng-checked 设置复选框。

ng-class 动态设置CSS类。

ng-cloak 在AngularJS接管之前防止显示内容。

ng-click 元素被点击时执行方法或表达式。

ng-controller 将控制器类附加到视图。

ng-disabled 控制表单元素的禁用属性

ng-form 设置表单

ng-href 动态绑定AngularJS变量到href属性。

ng-include 用于获取、编译并包含外部HTML片段到页面中。

ng-if 根据表达式在DOM中移除或重新创建元素。

ng-switch 根据匹配表达式有条件地切换控制。

ng-model 将输入框、选择框、文本区域等元素与模型属性绑定。

ng-readonly 用于设置元素的只读属性。

ng-repeat 用于遍历集合中的每个项目以创建新的模板。

ng-selected 用于设置元素中被选中的选项。

ng-show/ng-hide 根据表达式显示/隐藏元素。

ng-src 动态绑定AngularJS变量到src属性。

ng-submit 绑定Angular表达式到onsubmit事件。

ng-value 绑定Angular表达式到value属性。

ng-required 绑定Angular表达式到onsubmit事件。

Here to access the index of parent ng-repeat inside child ng-repeat, you can use `$parent.$index`.

Section 4.5: Built-In Directives Cheat Sheet

`ng-app` Sets the AngularJS section.

`ng-init` Sets a default variable value.

`ng-bind` Alternative to {{ }} template.

`ng-bind-template` Binds multiple expressions to the view.

`ng-non-bindable` States that the data isn't bindable.

`ng-bind-html` Binds inner HTML property of an HTML element.

`ng-change` Evaluates specified expression when the user changes the input.

`ng-checked` Sets the checkbox.

`ng-class` Sets the css class dynamically.

`ng-cloak` Prevents displaying the content until AngularJS has taken control.

`ng-click` Executes a method or expression when element is clicked.

`ng-controller` Attaches a controller class to the view.

`ng-disabled` Controls the form element's disabled property

`ng-form` Sets a form

`ng-href` Dynamically bind AngularJS variables to the href attribute.

`ng-include` Used to fetch, compile and include an external HTML fragment to your page.

`ng-if` Remove or recreates an element in the DOM depending on an expression

`ng-switch` Conditionally switch control based on matching expression.

`ng-model` Binds an input,select, textarea etc elements with model property.

`ng-readonly` Used to set readonly attribute to an element.

`ng-repeat` Used to loop through each item in a collection to create a new template.

`ng-selected` Used to set selected option in element.

`ng-show/ng-hide` Show/Hide elements based on an expression.

`ng-src` Dynamically bind AngularJS variables to the src attribute.

`ng-submit` Bind angular expressions to onsubmit events.

`ng-value` Bind angular expressions to the value of .

`ng-required` Bind angular expressions to onsubmit events.

ng-style 设置HTML元素的CSS样式。

ng-pattern 为ngModel添加pattern验证器。

ng-maxlength 向 ngModel 添加 maxlength 验证器。

ng-minlength 向 ngModel 添加 minlength 验证器。

ng-classeven 与 ngRepeat 配合使用，仅对偶数行生效。

ng-classodd 与 ngRepeat 配合使用，仅对奇数行生效。

ng-cut 用于指定剪切事件的自定义行为。

ng-copy 用于指定复制事件的自定义行为。

ng-paste 用于指定粘贴事件的自定义行为。

ng-options 用于动态生成元素的列表。

ng-list 用于根据指定分隔符将字符串转换为列表。

ng-open 如果 ngOpen 中的表达式为真，则用于设置元素的 open 属性。

来源（稍作编辑）

第4.6节：ngInclude

ng-include 允许你将页面的一部分控制权委托给特定的控制器。你可能会这样做，因为该组件的复杂性变得如此之大，以至于你想将所有逻辑封装在一个专用的控制器中。

一个例子是：

```
<div ng-include
     src="'/gridview'"
     ng-controller='gridController as gc'>
</div>
```

注意，/gridview 需要由网络服务器作为一个独立且合法的URL提供服务。

另外，注意 src 属性接受一个Angular表达式。例如，这可以是一个变量或函数调用，或者像本例中一样，是一个字符串常量。在这种情况下，你需要确保将源URL用单引号包裹起来，这样它才会被当作字符串常量来计算。这是一个常见的混淆点。

在/gridview 的HTML中，你可以像它包裹整个页面一样引用gridController，例如：

```
<div class="row">
  <button type="button" class="btn btn-default" ng-click="gc.doSomething()"></button>
</div>
```

第4.7节：ng-model-options

ng-model-options 允许更改 ng-model 的默认行为，该指令允许注册在 ng-model 更新时触发的事件，并附加防抖效果。

ng-style Sets CSS style on an HTML element.

ng-pattern Adds the pattern validator to ngModel.

ng-maxlength Adds the maxlength validator to ngModel.

ng-minlength Adds the minlength validator to ngModel.

ng-classeven Works in conjunction with ngRepeat and take effect only on odd (even) rows.

ng-classodd Works in conjunction with ngRepeat and take effect only on odd (even) rows.

ng-cut Used to specify custom behavior on cut event.

ng-copy Used to specify custom behavior on copy event.

ng-paste Used to specify custom behavior on paste event.

ng-options Used to dynamically generate a list of elements for the element.

ng-list Used to convert string into list based on specified delimiter.

ng-open Used to set the open attribute on the element, if the expression inside ngOpen is truthy.

Source (edited a bit)

Section 4.6: ngInclude

ng-include allows you to delegate the control of one part of the page to a specific controller. You may want to do this because the complexity of that component is becoming such that you want to encapsulate all the logic in a dedicated controller.

An example is:

```
<div ng-include
     src="'/gridview'"
     ng-controller='gridController as gc'>
</div>
```

Note that the /gridview will need to be served by the web server as a distinct and legitimate url.

Also, note that the src-attribute accepts an Angular expression. This could be a variable or a function call for example or, like in this example, a string constant. In this case you need to make sure to **wrap the source URL in single quotes**, so it will be evaluated as a string constant. This is a common source of confusion.

Within the /gridview html, you can refer to the gridController as if it were wrapped around the page, eg:

```
<div class="row">
  <button type="button" class="btn btn-default" ng-click="gc.doSomething()"></button>
</div>
```

Section 4.7: ng-model-options

ng-model-options allows to change the default behavior of ng-model, this directive allows to register events that will fire when the ng-model is updated and to attach a debounce effect.

该指令接受一个表达式，该表达式将被求值为一个定义对象或作用域值的引用。

示例：

```
<input type="text" ng-model="myValue" ng-model-options="{'debounce': 500}">
```

上述示例将在 myValue 上附加一个500毫秒的防抖效果，这将导致模型在用户完成输入后500毫秒更新（即当 myValue 完成更新时）。

可用的对象属性

1. updateOn：指定应绑定到输入的事件

   ```
   ng-model-options="{ updateOn: 'blur'}" // 将在失去焦点时更新
   ```

2. 防抖：指定对模型更新延迟若干毫秒

   ```
   ng-model-options="{'debounce': 500}" // 将在半秒后更新模型
   ```

3. allowInvalid: 一个布尔标志，允许模型接受无效值，绕过默认的表单验证，默认情况下这些值将被视为undefined。

4. getterSetter: 一个布尔标志，指示是否将ng-model视为getter/setter函数而非普通模型值。函数将被执行并返回模型值。

   示例：

   ```
   <input type="text" ng-model="myFunc" ng-model-options="{'getterSetter': true}">

   $scope.myFunc = function() {return "value";}
   ```

5. timezone: 如果输入类型为date或 time，定义模型的时区

第4.8节：ngCopy

ngCopy指令指定在复制事件上运行的行为。

阻止用户复制数据

```
<p ng-copy="blockCopy($event)">此段落无法被复制</p>
```

在控制器中

```
$scope.blockCopy = function(event) {
    event.preventDefault();
console.log("复制操作无效");
}
```

第4.9节：ngPaste

ngPaste 指令指定用户粘贴内容时运行的自定义行为

```
<input ng-paste="paste=true" ng-init="paste=false" placeholder='请粘贴'>
已粘贴: {{paste}}
```


This directive accepts an expression that will evaluate to a definition object or a reference to a scope value.

Example:

```
<input type="text" ng-model="myValue" ng-model-options="{'debounce': 500}">
```

The above example will attach a debounce effect of 500 milliseconds on `myValue`, which will cause the model to update 500 ms after the user finished typing over the `input` (that is, when the `myValue` finished updating).

Available object properties

1. `updateOn`: specifies which event should be bound to the input

   ```
   ng-model-options="{ updateOn: 'blur'}" // will update on blur
   ```

2. `debounce`: specifies a delay of some millisecond towards the model update

   ```
   ng-model-options="{'debounce': 500}" // will update the model after 1/2 second
   ```

3. `allowInvalid`: a boolean flag allowing for an invalid value to the model, circumventing default form validation, by default these values would be treated as **`undefined`**.

4. `getterSetter`: a boolean flag indicating if to treat the `ng-model` as a getter/setter function instead of a plain model value. The function will then run and return the model value.

   Example:

   ```
   <input type="text" ng-model="myFunc" ng-model-options="{'getterSetter': true}">

   $scope.myFunc = function() {return "value";}
   ```

5. `timezone`: defines the timezone for the model if the input is of the `date` or `time`. types

Section 4.8: ngCopy

The `ngCopy` directive specifies behavior to be run on a copy event.

Prevent a user from copying data

```
<p ng-copy="blockCopy($event)">This paragraph cannot be copied</p>
```

In the controller

```
$scope.blockCopy = function(event) {
    event.preventDefault();
    console.log("Copying won't work");
}
```

Section 4.9: ngPaste

The `ngPaste` directive specifies custom behavior to run when a user pastes content

```
<input ng-paste="paste=true" ng-init="paste=false" placeholder='paste here'>
pasted: {{paste}}
```

第4.10节：ngClick

ng-click 指令为DOM元素绑定点击事件。

ng-click 指令允许你指定DOM元素被点击时的自定义行为。

当您想要在按钮上附加点击事件并在控制器中处理它们时，这非常有用。

该指令接受一个表达式，事件对象可通过$event访问

HTML

```
<input ng-click="onClick($event)">点击我</input>
```

控制器

```
.controller("ctrl", function($scope) {
    $scope.onClick = function(evt) {
        console.debug("Hello click event: %o ",evt);
    }
})
```

HTML

```
<button ng-click="count = count + 1" ng-init="count=0">
  增加
</button>
<span>
计数: {{count}}
</span>
```

HTML

```
<button ng-click="count()" ng-init="count=0">
  增加
</button>
<span>
计数: {{count}}
</span>
```

控制器

```
...

$scope.count = function(){
    $scope.count = $scope.count + 1;
}
...
```

当按钮被点击时，调用 onClick函数将打印“Hello click event”，后面跟着事件对象。

第4.11节：ngList

ng-list指令用于将文本输入中的分隔字符串转换为字符串数组，反之亦然。

Section 4.10: ngClick

The ng-click directive attaches a click event to a DOM element.

The ng-click directive allows you to specify custom behavior when an element of DOM is clicked.

It is useful when you want to attach click events on buttons and handle them at your controller.

This directive accepts an expression with the events object available as $event

HTML

```
<input ng-click="onClick($event)">Click me</input>
```

Controller

```
.controller("ctrl", function($scope) {
    $scope.onClick = function(evt) {
        console.debug("Hello click event: %o ",evt);
    }
})
```

HTML

```
<button ng-click="count = count + 1" ng-init="count=0">
  Increment
</button>
<span>
  count: {{count}}
</span>
```

HTML

```
<button ng-click="count()" ng-init="count=0">
  Increment
</button>
<span>
  count: {{count}}
</span>
```

Controller

```
...

$scope.count = function(){
    $scope.count = $scope.count + 1;
}
...
```

When the button is clicked, an invocation of the onClick function will print "Hello click event" followed by the event object.

Section 4.11: ngList

The ng-list directive is used to convert a delimited string from a text input to an array of strings or vice versa.

ng-list指令默认使用", "（逗号加空格）作为分隔符。

你可以通过赋值给ng-list一个分隔符来手动设置分隔符，例如ng-list="; "。

在这种情况下，分隔符被设置为分号后跟一个空格。

默认情况下，ng-list有一个属性ng-trim，默认值为true。当ng-trim为false时，会保留分隔符中的空白字符。默认情况下，ng-list不会考虑空白字符，除非你设置ng-trim="false"。

示例：

```
angular.module('test', [])
.控制器('ngListExample', ['$scope', 函数($scope) {
    $scope.列表 = ['angular', '是', '很酷!'];
}]);
```

客户分隔符设置为;。输入框的模型设置为在作用域中创建的数组。

```
<body ng-app="test" ng-controller="ngListExample">
  <input ng-model="list" ng-list="; " ng-trim="false">
</body>
```

输入框将显示内容：angular; is; cool!

第4.12节：ngOptions

ngOptions 是一个指令，用于简化创建HTML下拉框，从数组中选择项目并存储到模型中。ngOptions属性用于通过评估ngOptions理解表达式动态生成<option>元素列表，用于<select>元素，数组或对象作为数据源。

使用ng-options，标记可以简化为仅一个select标签，指令将创建相同的选择框：

```
<select ng-model="selectedFruitNgOptions"
       ng-options="curFruit as curFruit.label for curFruit in fruit">
</select>
```

还有另一种使用ng-repeat创建SELECT选项的方法，但不推荐使用ng-repeat，因为它主要用于通用目的，如forEach循环。而ng-options专门用于创建
SELECT标签选项。

上面使用ng-repeat的示例将是

```
<select ng-model="selectedFruit">
    <option ng-repeat="curFruit in fruit" value="{{curFruit}}">
      {{curFruit.label}}
    </option>
</select>
```

完整示例

让我们详细看看上面的示例以及其中的一些变体。

示例的数据模型：

```
$scope.fruit = [
```


The ng-list directive uses a default delimiter of ", " (comma space).

You can set the delimiter manually by assigning ng-list a delimeter like this ng-list="; ".

In this case the delimiter is set to a semi colon followed by a space.

By default ng-list has an attribute ng-trim which is set to true. ng-trim when false, will respect white space in your delimiter. By default, ng-list does not take white space into account unless you set ng-trim="false".

Example:

```
angular.module('test', [])
  .controller('ngListExample', ['$scope', function($scope) {
    $scope.list = ['angular', 'is', 'cool!'];
}]);
```

A customer delimiter is set to be ;. And the model of the input box is set to the array that was created on the scope.

```
<body ng-app="test" ng-controller="ngListExample">
  <input ng-model="list" ng-list="; " ng-trim="false">
</body>
```

The input box will display with the content: angular; is; cool!

Section 4.12: ngOptions

ngOptions is a directive that simplifies the creation of a html dropdown box for the selection of an item from an array that will be stored in a model. The ngOptions attribute is used to dynamically generate a list of **<option>** elements for the **<select>** element using the array or object obtained by evaluating the ngOptions comprehension expression.

With ng-options the markup can be reduced to just a select tag and the directive will create the same select:

```
<select ng-model="selectedFruitNgOptions"
       ng-options="curFruit as curFruit.label for curFruit in fruit">
</select>
```

There is anther way of creating **SELECT** options using ng-repeat, but it is not recommended to use ng-repeat as it is mostly used for general purpose like, the forEach just to loop. Whereas ng-options is specifically for creating **SELECT** tag options.

Above example using ng-repeat would be

```
<select ng-model="selectedFruit">
    <option ng-repeat="curFruit in fruit" value="{{curFruit}}">
        {{curFruit.label}}
    </option>
</select>
```

FULL EXAMPLE

Lets see the above example in detail also with some variations in it.

Data model for the example:

```
$scope.fruit = [
```

```
    { label: "苹果", value: 4, id: 2 },
    { label: "橙子", value: 2, id: 1 },
    { label: "酸橙", value: 4, id: 4 },
    { label: "柠檬", value: 5, id: 3 }
];
```

```
<!-- 数组中值的标签 -->
<select ng-options="f.label for f in fruit" ng-model="selectedFruit"></select>
```

选择生成的选项标签：

```
<option value="{ label: "Apples", value: 4, id: 2 }"> 苹果 </option>
```

效果：

f.label 将作为 <option> 的标签，value 将包含整个对象。

完整示例

```
<!-- select as label for value in array -->
<select ng-options="f.value as f.label for f in fruit" ng-model="selectedFruit"></select>
```

选择生成的选项标签：

```
<option value="4"> 苹果 </option>
```

效果：

在这种情况下，f.value (4) 将作为值，而标签仍然相同。

完整示例

```
<!-- label group by group for value in array -->
<select ng-options="f.label group by f.value for f in fruit" ng-model="selectedFruit"></select>
```

选择生成的选项标签：

```
<option value="{ label: "Apples", value: 4, id: 2 }"> 苹果 </option>
```

效果：

选项将根据其value进行分组。具有相同value的选项将归为一类

完整示例

```
<!-- label disable when disable for value in array -->
<select ng-options="f.label disable when f.value == 4 for f in fruit" ng-
model="selectedFruit"></select>
```

选择生成的选项标签：

```
<option disabled="" value="{ label: "Apples", value: 4, id: 2 }"> 苹果 </option>
```



```
    { label: "Apples", value: 4, id: 2 },
    { label: "Oranges", value: 2, id: 1 },
    { label: "Limes", value: 4, id: 4 },
    { label: "Lemons", value: 5, id: 3 }
];
```

```
<!-- label for value in array -->
<select ng-options="f.label for f in fruit" ng-model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```

Effects:

`f.label` will be the label of the **`<option>`** and the value will contain the entire object.

FULL EXAMPLE

```
<!-- select as label for value in array -->
<select ng-options="f.value as f.label for f in fruit" ng-model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="4"> Apples </option>
```

Effects:

`f.value` (4) will be the value in this case while the label is still the same.

FULL EXAMPLE

```
<!-- label group by group for value in array -->
<select ng-options="f.label group by f.value for f in fruit" ng-model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```

Effects:

Options will be grouped based on there `value`. Options with same `value` will fall under one category

FULL EXAMPLE

```
<!-- label disable when disable for value in array -->
<select ng-options="f.label disable when f.value == 4 for f in fruit" ng-
model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option disabled="" value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```


belindoc.com

效果：

“苹果”和“青柠”将被禁用（无法选择），因为条件disable when f.value==4。所有value=4的选项都将被禁用

完整示例

```
<!-- label group by group for value in array track by trackexpr -->
<select ng-options="f.value as f.label group by f.value for f in fruit track by f.id" ng-
model="selectedFruit"></select>
```

选择生成的选项标签：

```
<option value="4"> 苹果 </option>
```

效果：

使用trackBy时视觉上没有变化，但Angular将通过id检测变化，而不是通过引用，这通常是更好的解决方案。

完整示例

```
<!-- label for value in array | orderBy:orderexpr track by trackexpr -->
<select ng-options="f.label for f in fruit | orderBy:'id' track by f.id" ng-
model="selectedFruit"></select>
```

选择生成的选项标签：

```
<option disabled="" value="{ label: "Apples", value: 4, id: 2 }"> 苹果 </option>
```

效果：

orderBy 是 AngularJS 的一个标准过滤器，默认按升序排列选项，因此“橙子”在这里会出现在第一位，因为它的 id = 1 。

完整示例

所有带有 <select> 并使用 ng-options 的元素必须绑定 ng-model。

第4.13节：ngSrc

在 src 属性中使用 Angular 标记如 {{hash}} 无法正常工作。浏览器会从包含字面文本 {{hash}} 的 URL 中获取资源，直到 Angular 替换 {{hash}} 内的表达式。 ng-src 指令会覆盖图片标签元素的原始 src 属性，从而解决该问题。

```
<div ng-init="pic = 'pic_angular.jpg'">
    <h1>Angular</h1>
    <img ng-src="{{pic}}">
</div>
```

第4.14节：ngModel

使用 ng-model 可以将变量绑定到任何类型的输入字段。你可以使用双大括号显示该变量

Effects:

"Apples" and "Limes" will be disabled (unable to select) because of the condition `disable when f.value==4`. All options with `value=4` shall be disabled

FULL EXAMPLE

```
<!-- label group by group for value in array track by trackexpr -->
<select ng-options="f.value as f.label group by f.value for f in fruit track by f.id" ng-
model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option value="4"> Apples </option>
```

Effects:

There is not visual change when using `trackBy`, but Angular will detect changes by the `id` instead of by reference which is most always a better solution.

FULL EXAMPLE

```
<!-- label for value in array | orderBy:orderexpr track by trackexpr -->
<select ng-options="f.label for f in fruit | orderBy:'id' track by f.id" ng-
model="selectedFruit"></select>
```

Option tag generated on selection:

```
<option disabled="" value="{ label: "Apples", value: 4, id: 2 }"> Apples </option>
```

Effects:

`orderBy` is a AngularJS standard filter which arranges options in ascending order(by default) so "Oranges" in this will appear 1st since its `id` = 1.

FULL EXAMPLE

All `<select>` with `ng-options` must have `ng-model` attached.

Section 4.13: ngSrc

Using Angular markup like `{{hash}}` in a src attribute doesn't work right. The browser will fetch from the URL with the literal text `{{hash}}` until Angular replaces the expression inside `{{hash}}`. `ng-src` directive overrides the original `src` attribute for the image tag element and solves the problem

```
<div ng-init="pic = 'pic_angular.jpg'">
    <h1>Angular</h1>
    <img ng-src="{{pic}}">
</div>
```

Section 4.14: ngModel

With ng-model you can bind a variable to any type of input field. You can display the variable using double curly

例如 {{myAge}}。

```
<input type="text" ng-model="myName">
<p>{{myName}}</p>
```

当您在输入框中输入或以任何方式更改内容时，段落中的值会立即更新。

在此示例中，ng-model 变量将在您的控制器中作为$scope.myName使用。如果您使用的是 controllerAs语法：

```
<div ng-controller="myCtrl as mc">
    <input type="text" ng-model="mc.myName">
    <p>{{mc.myName}}</p>
</div>
```

您需要通过在 ng-controller
属性中定义的控制器别名前缀来引用控制器的作用域，从而访问 ng-model 变量。这样，您就不需要将$scope注入到控制器中来引用您的 ng-model 变量，该变量将在控制器函数内部作为**this**.myName使用。

第4.15节：ngClass

假设您需要显示用户的状态，并且有多个可能使用的 CSS 类。Angular 使从多个可能的类中选择变得非常简单，您可以指定一个包含条件的对象列表。Angular 能够根据条件的真假使用正确的类。

您的对象应包含键/值对。键是当值（条件）为真时将应用的类名。

```
<style>
.active { background-color: green; color: white; }
    .inactive { background-color: gray; color: white; }
    .adminUser { font-weight: bold; color: yellow; }
    .regularUser { color: white; }
</style>

<span ng-class="{
   active: user.active,
   inactive: !user.active,
   adminUser: user.level === 1,
   regularUser: user.level === 2
}">约翰·史密斯</span>
```

Angular 将检查$scope.user对象以查看active状态和level数字。根据这些变量中的值，Angular 将对应用匹配的样式。

第4.16节：ngDblclick

当你想要将双击事件绑定到 DOM 元素时，ng-dblclick指令非常有用。

该指令接受一个表达式

HTML

```
<input type="number" ng-model="num = num + 1" ng-init="num=0">
```


braces, eg {{myAge}}.

```
<input type="text" ng-model="myName">
<p>{{myName}}</p>
```

As you type in the input field or change it in any way you will see the value in the paragraph update instantly.

The ng-model variable, in this instance, will be available in your controller as $scope.myName. If you are using the controllerAs syntax:

```
<div ng-controller="myCtrl as mc">
    <input type="text" ng-model="mc.myName">
    <p>{{mc.myName}}</p>
</div>
```

You will need to refer to the controller's scope by pre-pending the controller's alias defined in the ng-controller attribute to the ng-model variable. This way you won't need to inject $scope into your controller to reference your ng-model variable, the variable will be available as **this**.myName inside your controller's function.

Section 4.15: ngClass

Let's assume that you need to show the status of a user and you have several possible CSS classes that could be used. Angular makes it very easy to choose from a list of several possible classes which allow you to specify an object list that include conditionals. Angular is able to use the correct class based on the truthiness of the conditionals.

Your object should contain key/value pairs. The key is a class name that will be applied when the value (conditional) evaluates to true.

```
<style>
    .active { background-color: green; color: white; }
    .inactive { background-color: gray; color: white; }
    .adminUser { font-weight: bold; color: yellow; }
    .regularUser { color: white; }
</style>

<span ng-class="{
   active: user.active,
   inactive: !user.active,
   adminUser: user.level === 1,
   regularUser: user.level === 2
}">John Smith</span>
```

Angular will check the $scope.user object to see the active status and the level number. Depending on the values in those variables, Angular will apply the matching style to the ****.

Section 4.16: ngDblclick

The ng-dblclick directive is useful when you want to bind a double-click event into your DOM elements.

This directive accepts an expression

HTML

```
<input type="number" ng-model="num = num + 1" ng-init="num=0">
```

```
<button ng-dblclick="num++">双击我</button>
```

在上述示例中，当按钮被双击时，input中保存的值将会增加。

第4.17节：ngHref

如果 href 属性中包含 Angular 表达式，则使用 ngHref 替代 href 属性。ngHref 指令覆盖 HTML 标签的原始 href 属性，使用 href 属性的标签如 a 标签等。

ngHref 指令确保即使用户在 AngularJS 评估代码之前点击链接，链接也不会失效。

示例 1

```
<div ng-init="linkValue = 'http://stackoverflow.com'">
    <p>前往 <a ng-href="{{linkValue}}">{{linkValue}}</a> ! </p>
</div>
```

示例 2 该示例动态获取输入框中的 href 值并将其作为 href 值加载。

```
<input ng-model="value" />
<a id="link" ng-href="{{value}}">链接</a>
```

示例 3

```
<script>
angular.module('angularDoc', [])
.controller('myController', function($scope) {
  // 设置一些作用域值。
// 这里设置 bootstrap 版本。
$scope.bootstrap_version = '3.3.7';

  // 设置默认布局值
  $scope.layout = 'normal';
});
</script>
<!-- 将其插入到 Angular 代码中 -->
<link rel="stylesheet" ng-href="//maxcdn.bootstrapcdn.com/bootstrap/{{ bootstrap_version
}}/css/bootstrap.min.css">
<link rel="stylesheet" ng-href="layout-{{ layout }}.css">
```

第4.18节：ngPattern

ng-pattern 指令接受一个表达式，该表达式计算为正则表达式模式，并使用该模式验证文本输入。

示例：

假设我们希望当一个<input>元素的值（ng-model）是有效的IP地址时，该元素变为有效。

模板：

```
<input type="text" ng-model="ipAddr" ng-pattern="ipRegex" name="ip" required>
```

控制器：


```
<button ng-dblclick="num++">Double click me</button>
```

In the above example, the value held at the `input` will be incremented when the button is double clicked.

Section 4.17: ngHref

ngHref is used instead of href attribute, if we have a angular expressions inside href value. The ngHref directive overrides the original href attribute of an html tag using href attribute such as tag, tag etc.

The ngHref directive makes sure the link is not broken even if the user clicks the link before AngularJS has evaluated the code.

Example 1

```
<div ng-init="linkValue = 'http://stackoverflow.com'">
    <p>Go to <a ng-href="{{linkValue}}">{{linkValue}}</a>!</p>
</div>
```

Example 2 This example dynamically gets the href value from input box and load it as href value.

```
<input ng-model="value" />
<a id="link" ng-href="{{value}}">link</a>
```

Example 3

```
<script>
angular.module('angularDoc', [])
.controller('myController', function($scope) {
  // Set some scope value.
  // Here we set bootstrap version.
  $scope.bootstrap_version = '3.3.7';

  // Set the default layout value
  $scope.layout = 'normal';
});
</script>
<!-- Insert it into Angular Code -->
<link rel="stylesheet" ng-href="//maxcdn.bootstrapcdn.com/bootstrap/{{ bootstrap_version
}}/css/bootstrap.min.css">
<link rel="stylesheet" ng-href="layout-{{ layout }}.css">
```

Section 4.18: ngPattern

The `ng-pattern` directive accepts an expression that evaluates to a regular expression pattern and uses that pattern to validate a textual input.

Example:

Lets say we want an **<input>** element to become valid when it's value (ng-model) is a valid IP address.

Template:

```
<input type="text" ng-model="ipAddr" ng-pattern="ipRegex" name="ip" required>
```

Controller:

```
$scope.ipRegex =
/\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b/;
```

第4.19节：ngShow 和 ngHide

ng-show 指令根据传入的表达式是否为真来显示或隐藏HTML元素。如果表达式的值为假值，则隐藏；如果为真值，则显示。

ng-hide 指令类似。但如果值为假值，则显示HTML元素；当表达式为真值时，则隐藏它。

可运行的JSBin示例

控制器:

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

angular.module('app')
.controller('ExampleController', ExampleController);

function ExampleController() {

  var vm = this;

  //将用户名绑定到HTML元素
  vm.username = '';

  //一个被占用的用户名
vm.taken_username = 'StackOverflow';

}
```

查看

```
<section ng-controller="ExampleController as main">

    <p>输入密码</p>
    <input ng-model="main.username" type="text">

    <hr>

    <!-- 只要输入的不是 StackOverflow 就会显示 -->
    <!-- 当表达式不等于 StackOverflow 时总为真 -->
    <div style="color:green;" ng-show="main.username != main.taken_username">
     你的用户名可以使用！
    </div>

    <!-- 只有输入 StackOverflow 时才显示 -->
    <!-- 表达式值变为假 -->
    <div style="color:red;" ng-hide="main.username != main.taken_username">
     你的用户名已被占用！
    </div>

    <p>在用户名字段输入 'StackOverflow' 以显示 ngHide 指令。</p>

 </section>
```



```
$scope.ipRegex =
/\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b/;
```

Section 4.19: ngShow and ngHide

The ng-show directive shows or hides the HTML element based on if the expression passed to it is true or false. If the value of the expression is falsy then it will hide. If it is truthy then it will show.

The ng-hide directive is similar. However, if the value is falsy it will show the HTML element. When the expression is truthy it will hide it.

Working JSBin Example

Controller:

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

angular.module('app')
  .controller('ExampleController', ExampleController);

function ExampleController() {

  var vm = this;

  //Binding the username to HTML element
  vm.username = '';

  //A taken username
  vm.taken_username = 'StackOverflow';

}
```

View

```
<section ng-controller="ExampleController as main">

    <p>Enter Password</p>
    <input ng-model="main.username" type="text">

    <hr>

    <!-- Will always show as long as StackOverflow is not typed in -->
    <!-- The expression is always true when it is not StackOverflow -->
    <div style="color:green;" ng-show="main.username != main.taken_username">
      Your username is free to use!
    </div>

    <!-- Will only show when StackOverflow is typed in -->
    <!-- The expression value becomes falsy -->
    <div style="color:red;" ng-hide="main.username != main.taken_username">
      Your username is taken!
    </div>

    <p>Enter 'StackOverflow' in username field to show ngHide directive.</p>

 </section>
```


bellindoc.com

第4.20节：ngRequired

ng-required 指令会在元素上添加或移除 required 验证属性，从而启用或禁用 input 的 require 验证键。

它用于可选地定义 input 元素是否必须有非空值。该指令在设计复杂HTML表单的验证时非常有用。

HTML

```
<input type="checkbox" ng-model="someBooleanValue">
<input type="text" ng-model="username" ng-required="someBooleanValue">
```

第4.21节：ngMouseenter 和 ngMouseleave

ng-mouseenter 和 ng-mouseleave 指令用于在鼠标进入或离开DOM元素时触发事件并应用CSS样式。

ng-mouseenter 指令会在鼠标进入事件时执行表达式（当用户将鼠标指针移入该指令所在的DOM元素时）。

HTML

```
<div ng-mouseenter="applyStyle = true" ng-class="{active': applyStyle}">
```

在上述示例中，当用户将鼠标指向该 div 时，applyStyle 变为 true，进而在 ng-class 上应用 .active CSS类。

ng-mouseleave 指令在鼠标离开事件时运行一个表达式（当用户将鼠标光标移出该指令所在的 DOM 元素时）

HTML

```
<div ng-mouseenter="applyStyle = true" ng-mouseleaver="applyStyle = false" ng-class="{'active':
applyStyle}">
```

重复第一个例子，现在当用户将鼠标指针移出该 div 时，.active 类将被移除。

第4.22节：ngDisabled

该指令用于根据某些现有条件限制输入事件。

ng-disabled 指令接受一个表达式，该表达式应计算为真值或假值。

ng-disabled 用于有条件地在input元素上应用disabled属性。

HTML

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

<button ng-disabled="vm.name.length===0" ng-click="vm.submitMe">提交</button>
```

当vm.name.length===0计算结果为真时，表示输入框长度为0，这将禁用按钮，禁止

Section 4.20: ngRequired

The `ng-required` adds or removes the `required` validation attribute on an element, which in turn will enable and disable the `require` validation key for the `input`.

It is used to optionally define if an `input` element is required to have a non-empty value. The directive is helpful when designing validation on complex HTML forms.

HTML

```
<input type="checkbox" ng-model="someBooleanValue">
<input type="text" ng-model="username" ng-required="someBooleanValue">
```

Section 4.21: ngMouseenter and ngMouseleave

The `ng-mouseenter` and `ng-mouseleave` directives are useful to run events and apply CSS styling when you hover into or out of your DOM elements.

The `ng-mouseenter` directive runs an expression one a mouse enter event (when the user enters his mouse pointer over the DOM element this directive resides in)

HTML

```
<div ng-mouseenter="applyStyle = true" ng-class="{'active': applyStyle}">
```

At the above example, when the user points his mouse over the `div`, `applyStyle` turns to **`true`**, which in turn applies the `.active` CSS class at the `ng-class`.

The `ng-mouseleave` directive runs an expression one a mouse exit event (when the user takes his mouse cursor away from the DOM element this directive resides in)

HTML

```
<div ng-mouseenter="applyStyle = true" ng-mouseleaver="applyStyle = false" ng-class="{'active':
applyStyle}">
```

Reusing the first example, now when the user takes him mouse pointer away from the div, the `.active` class is removed.

Section 4.22: ngDisabled

This directive is useful to limit input events based on certain existing conditions.

The `ng-disabled` directive accepts and expression that should evaluate to either a truthy or a falsy values.

`ng-disabled` is used to conditionally apply the `disabled` attribute on an `input` element.

HTML

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

<button ng-disabled="vm.name.length===0" ng-click="vm.submitMe">Submit</button>
```

`vm.name.length===0` is evaluated to true if the `input`'s length is 0, which is turn disables the button, disallowing the

用户触发 ng-click 的点击事件

第4.23节：ngValue

主要用于 ng-repeat 当使用 ngRepeat 动态生成单选按钮列表时，ngValue 非常有用

```
<script>
   angular.module('valueExample', [])
     .controller('ExampleController', ['$scope', function($scope) {
      $scope.names = ['披萨', '独角兽', '机器人'];
$scope.my = { favorite: '独角兽' };
     }]);
</script>
 <form ng-controller="ExampleController">
   <h2>你最喜欢哪个？</h2>
     <label ng-repeat="name in names" for="{{name}}">
      {{name}}
        <input type="radio"
               ng-model="my.favorite"
               ng-value="name"
               id="{{name}}"
               name="favorite">
     </label>
   <div>你选择了 {{my.favorite}}</div>
 </form>
```

正在运行的 plnkr

belindoc.com

user to fire the click event of ng-click

Section 4.23: ngValue

Mostly used under ng-repeat ngValue is useful when dynamically generating lists of radio buttons using ngRepeat

```
<script>
   angular.module('valueExample', [])
     .controller('ExampleController', ['$scope', function($scope) {
       $scope.names = ['pizza', 'unicorns', 'robots'];
       $scope.my = { favorite: 'unicorns' };
     }]);
</script>
 <form ng-controller="ExampleController">
   <h2>Which is your favorite?</h2>
     <label ng-repeat="name in names" for="{{name}}">
       {{name}}
       <input type="radio"
              ng-model="my.favorite"
              ng-value="name"
              id="{{name}}"
              name="favorite">
     </label>
   <div>You chose {{my.favorite}}</div>
 </form>
```

Working plnkr

第5章：内置指令的使用

第5.1节：隐藏/显示HTML元素

此示例演示隐藏和显示HTML元素。

```
<!DOCTYPE html>
<html ng-app="myDemoApp">
  <head>
    <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
    <script>

function HideShowController() {
        var vm = this;
          vm.show=false;
          vm.toggle= function() {
            vm.show=!vm.show;
          }
        }

angular.module("myDemoApp", [/* 模块依赖项写在这里 */])
        .controller("hideShowController", [HideShowController]);
    </script>
  </head>
  <body ng-cloak>
    <div ng-controller="hideShowController as vm">
      <a style="cursor: pointer;" ng-show="vm.show" ng-click="vm.toggle()">显示我！</a>
      <a style="cursor: pointer;" ng-hide="vm.show" ng-click="vm.toggle()">隐藏我！</a>
    </div>
  </body>
</html>
```

实时演示

逐步说明：

1. ng-app="myDemoApp"，ngApp 指令告诉 Angular 一个 DOM 元素由特定的名为 "myDemoApp" 的 angular.module 控制。
2. <script src="[//angular include]"> 引入 angular js。
3. 定义了 HideShowController 函数，包含另一个名为 toggle 的函数，用于帮助隐藏和显示该元素。
4. angular.module(...) 创建一个新模块。
5. .控制器(...) Angular 控制器 并返回模块以便链式调用；
6. ng-controller 指令 是 Angular 支持模型-视图-控制器设计模式背后原则的关键方面。控制器设计模式。
7. ng-show 指令 如果提供的表达式为真，则显示给定的 HTML 元素。
8. ng-hide 指令 如果提供的表达式为真，则隐藏给定的 HTML 元素。
9. ng-click 指令 触发控制器内的切换函数

Chapter 5: Use of in-built directives

Section 5.1: Hide/Show HTML Elements

This example hide show html elements.

```
<!DOCTYPE html>
<html ng-app="myDemoApp">
  <head>
    <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
    <script>

      function HideShowController() {
        var vm = this;
        vm.show=false;
        vm.toggle= function() {
          vm.show=!vm.show;
        }
      }

      angular.module("myDemoApp", [/* module dependencies go here */])
        .controller("hideShowController", [HideShowController]);
    </script>
  </head>
  <body ng-cloak>
    <div ng-controller="hideShowController as vm">
      <a style="cursor: pointer;" ng-show="vm.show" ng-click="vm.toggle()">Show Me!</a>
      <a style="cursor: pointer;" ng-hide="vm.show" ng-click="vm.toggle()">Hide Me!</a>
    </div>
  </body>
</html>
```

Live Demo

Step by step explanation:

1. `ng-app="myDemoApp"`, the ngApp directive tells angular that a DOM element is controlled by a specific angular.module named "myDemoApp".
2. `<script src="[//angular include]">` include angular js.
3. `HideShowController` function is defined containing another function named `toggle` which help to hide show the element.
4. `angular.module(...)` creates a new module.
5. `.controller(...)` Angular Controller and returns the module for chaining;
6. `ng-controller` directive is key aspect of how angular supports the principles behind the Model-View-Controller design pattern.
7. `ng-show` directive shows the given HTML element if expression provided is true.
8. `ng-hide` directive hides the given HTML element if expression provided is true.
9. `ng-click` directive fires a toggle function inside controller

第6章：自定义指令

参数	详情
作用域	设置指令作用域的属性。它可以设置为 false、true 或隔离作用域：{ @, =, <, & }。
scope: 假值	指令使用父作用域。指令未创建新的作用域。
scope: true	指令以原型方式继承父作用域，作为新的子作用域。如果同一元素上有多个指令请求新的作用域，则它们将共享一个新的作用域。
scope: { @ }	指令作用域属性与DOM属性值的单向绑定。由于绑定的是父作用域中的属性值，因此该值会在指令作用域中变化。
scope: { = }	双向属性绑定，如果指令属性变化，则父作用域中的属性也会变化，反之亦然。
scope: { < }	指令作用域属性与DOM属性表达式的单向绑定。表达式在父作用域中求值。该绑定监视父值的身份，因此父作用域中对象属性的变化不会反映到指令中。指令中对象属性的变化会反映到父作用域，因为两者引用的是同一个对象。
scope: { & }	允许指令将数据传递给父级中要计算的表达式。
compile: 函数	此函数用于在链接函数运行之前对指令模板进行 DOM 转换。它接受 tElement（指令模板）和 tAttr（指令上声明的属性列表）。它无法访问作用域。它可以返回一个函数，该函数将被注册为 post-link 函数，或者返回一个带有 pre 和 post 属性的对象，这些属性将被注册为 pre-link 和 post-link 函数。
link: 函数/对象	link 属性可以配置为函数或对象。它可以接收以下参数：scope（指令作用域）、iElement（应用指令的 DOM 元素）、iAttrs（DOM 元素属性集合）、controller（指令所需的控制器数组）、transcludeFn。它主要用于设置 DOM 监听器、监视模型属性的变化以及更新 DOM。它在模板克隆后执行。如果没有 compile 函数，则独立配置。
pre-link 函数	在任何子链接函数之前执行的链接函数。默认情况下，子指令链接函数在父指令链接函数之前执行，pre-link 函数使父指令先链接。一个使用场景是子指令需要父指令的数据。
post-link 函数	在子元素链接到父元素之后执行的链接函数。通常用于附加事件处理程序和访问子指令，但不应在此设置子指令所需的数据，因为子指令已经被链接。
restrict: 字符串	定义如何从 DOM 内部调用指令。可能的值（假设我们的指令名为 demoDirective）：E - 元素名（<demo-directive></demo-directive>），A - 属性（<div demo-directive> ），C - 匹配类（ ），M - 通过注释（<!-- directive: demo-directive -->）。restrict 属性也支持多个选项，例如 - restrict: "AC" 将指令限制为*属性*或*类*。如果省略，**默认**值为"EA"（元素或属性）。
require: 'demoDirective'	在当前元素上定位demoDirective的控制器，并将其控制器作为链接函数的第四个参数注入。如果未找到则抛出错误。
require: '?demoDirective'	尝试定位demoDirective的控制器，如果未找到则传递null给链接函数。
require: '^demoDirective'	通过搜索元素及其父元素定位demoDirective的控制器。如果未找到则抛出错误。
require: '^^demoDirective'	通过搜索元素的父元素定位demoDirective的控制器。如果未找到则抛出错误。如果未找到。
require: '?^demoDirective'	尝试通过搜索元素及其父元素来定位 demoDirective 的控制器如果未找到，则传递 null 给链接函数。
require: '?^^demoDirective'	尝试通过搜索元素的父元素来定位 demoDirective 的控制器，或者如果未找到，则传递 null 给链接函数。

Chapter 6: Custom Directives

Parameter	Details
scope	Property to set the scope of the directive. It can be set as false, true or as an isolate scope: { @, =, <, & }.
scope: falsy	Directive uses parent scope. No scope created for directive.
scope: true	Directive inherits parent scope prototypically as a new child scope. If there are multiple directives on the same element requesting a new scope, then they will share one new scope.
scope: { @ }	One way binding of a directive scope property to a DOM attribute value. As the attribute value bound in the parent, it will change in the directive scope.
scope: { = }	Bi-directional attribute binding that changes the attribute in the parent if the directive attribute changes and vice-versa.
scope: { < }	One way binding of a directive scope property and a DOM attribute expression. The expression is evaluated in the parent. This watches the identity of the parent value so changes to an object property in the parent won't be reflected in the directive. Changes to an object property in a directive will be reflected in the parent, since both reference the same object
scope: { & }	Allows the directive to pass data to an expression to be evaluated in the parent.
compile: function	This function is used to perform DOM transformation on the directive template before the link function runs. It accepts `tElement` (the directive template) and `tAttr` (list of attributes declared on the directive). It does not have access to the scope. It may return a function that will be registered as a `post-link` function or it may return an object with `pre` and `post` properties with will be registered as the `pre-link` and `post-link` functions.
link: function/object	The link property can be configured as a function or object. It can receive the following arguments: scope(directive scope), iElement(DOM element where directive is applied), iAttrs(collection of DOM element attributes), controller(array of controllers required by directive), transcludeFn. It is mainly used to for setting up DOM listeners, watching model properties for changes, and updating the DOM. It executes after the template is cloned. It is configured independently if there is no compile function.
pre-link function	Link function that executes before any child link functions. By default, child directive link functions execute before parent directive link functions and the pre-link function enables the parent to link first. One use case is if the child requires data from the parent.
post-link function	Link function that executives after child elements are linked to parent. It is commonly used for attaching event handlers and accessing child directives, but data required by the child directive should not be set here because the child directive will have already been linked.
restrict: string	Defines how to call the directive from within the DOM. Possible values (Assuming our directive name is `demoDirective`): `E` - Element name (**`<demo-directive></demo-directive>`**), `A` - Attribute (**`<div demo-directive></div>`**), `C` - Matching class (**`<div class="demo-directive"></div>`**), `M` - By comment (`<!-- directive: demo-directive -->`). The `restrict` property can also support multiple options, for example - `restrict: "AC"` will restrict the directive to *Attribute* OR *Class*. If omitted, the **default** value is `"EA"` (Element or Attribute).
require: 'demoDirective'	Locate demoDirective's controller on the current element and inject its controller as the fourth argument to the linking function. Throw an error if not found.
require: '?demoDirective'	Attempt to locate the demoDirective's controller or pass null to the link fn if not found.
require: '^demoDirective'	Locate the demoDirective's controller by searching the element and its parents. Throw an error if not found.
require: '^^demoDirective'	Locate the demoDirective's controller by searching the element's parents. Throw an error if not found.
require: '?^demoDirective'	Attempt to locate the demoDirective's controller by searching the element and its parents or pass null to the link fn if not found.
require: '?^^demoDirective'	Attempt to locate the demoDirective's controller by searching the element's parents, or pass null to the link fn if not found.

在这里，您将学习 AngularJS 的指令（Directives）功能。下面您将找到关于指令是什么的信息，以及如何使用它们的基础和高级示例。

第6.1节：创建和使用自定义指令

指令是 AngularJS 最强大的功能之一。自定义 AngularJS 指令用于通过创建新的 HTML 元素或自定义属性来扩展 HTML 的功能，以为 HTML 标签提供特定的行为。

directive.js

```
// 如果你还没有创建应用模块，则创建它
var demoApp= angular.module("demoApp", []);

// 如果你已经创建了应用模块，注释掉上面一行并创建应用模块的引用

var demoApp = angular.module("demoApp");


// 使用以下语法创建指令
// 指令用于扩展HTML元素的功能// 你可以将其创建为元素/属性/类// 我们创建了一个名为demoDirective的指令。注意定义指令时采用驼峰命名法，就像ngModel一样

// 只要在HTML中遇到该元素，该指令就会被激活

demoApp.directive('demoDirective', function () {

  // 这将返回一个指令定义对象
  // 指令定义对象是一个简单的JavaScript对象，用于配置指令的行为、模板等

  return {
    // restrict: 'AE'，表示指令是元素/属性指令，
    // “E”表示元素，“A”表示属性，“C”表示类，“M”表示注释。
                                    // 属性将是添加行为时使用最多的主要方式。
    // 如果不指定 restrict 属性，默认值为 "A"
restrict :'AE',

    // scope 属性的值决定了指令内部实际作用域的创建和使用方式。这些值可以是 “false”、“true” 或 “{}”。此设置为指令创建一个隔离作用域。

    // '@' 绑定用于传递字符串。这些字符串支持 {{}} 表达式进行插值。

    // '=' 绑定用于双向模型绑定。父作用域中的模型与指令的隔离作用域中的模型相连接。

    // '&' 绑定用于将方法传入指令的作用域，以便在指令内部调用。

    // 该方法预绑定到指令的父作用域，并支持参数传递。
scope: {
name: "@",  // 即使是由2-3个单词组成的混合词，这里也始终使用小写字母
    },

    // 模板用其文本替换整个元素。
template: "<div>你好 {{name}}!</div>",

    // compile 在应用初始化时调用。AngularJS 在加载html页面时调用一次。

compile: function(element, attributes) {
      element.css("border", "1px solid #cccccc");

                        // linkFunction 与每个元素及其作用域关联，以获取元素特定的数据。
```


Here you will learn about the Directives feature of AngularJS. Below you will find information on what Directives are, as well as Basic and Advanced examples of how to use them.

Section 6.1: Creating and consuming custom directives

Directives are one of the most powerful features of angularjs. Custom angularjs directives are used to extend functionality of html by creating new html elements or custom attributes to provide certain behavior to an html tag.

directive.js

```
// Create the App module if you haven't created it yet
var demoApp= angular.module("demoApp", []);

// If you already have the app module created, comment the above line and create a reference of the app module
var demoApp = angular.module("demoApp");


// Create a directive using the below syntax
// Directives are used to extend the capabilities of html element
// You can either create it as an Element/Attribute/class
// We are creating a directive named demoDirective. Notice it is in CamelCase when we are defining the directive just like ngModel
// This directive will be activated as soon as any this element is encountered in html

demoApp.directive('demoDirective', function () {

  // This returns a directive definition object
  // A directive definition object is a simple JavaScript object used for configuring the directive's behaviour,template..etc
  return {
    // restrict: 'AE', signifies that directive is Element/Attribute directive,
    // "E" is for element, "A" is for attribute, "C" is for class, and "M" is for comment.
    // Attributes are going to be the main ones as far as adding behaviors that get used the most.
    // If you don't specify the restrict property it will default to "A"
    restrict :'AE',

    // The values of scope property decides how the actual scope is created and used inside a directive. These values can be either “false”, “true” or “{}”. This creates an isolate scope for the directive.
    // '@' binding is for passing strings. These strings support {{}} expressions for interpolated values.
    // '=' binding is for two-way model binding. The model in parent scope is linked to the model in the directive's isolated scope.
    // '&' binding is for passing a method into your directive's scope so that it can be called within your directive.
    // The method is pre-bound to the directive's parent scope, and supports arguments.
    scope: {
      name: "@",  // Always use small casing here even if it's a mix of 2-3 words
    },

    // template replaces the complete element with its text.
    template: "<div>Hello {{name}}!</div>",

    // compile is called during application initialization. AngularJS calls it once when html page is loaded.
    compile: function(element, attributes) {
      element.css("border", "1px solid #cccccc");

      // linkFunction is linked with each element with scope to get the element specific data.
```

```
        var linkFunction = function($scope, element, attributes) {
         element.html("名称: <b>"+$scope.name +"</b>");
         element.css("background-color", "#ff00ff");
        };
        return linkFunction;
    }
  };
});
```

该指令随后可以在应用中这样使用 ：

```
<html>

   <head>
      <title>Angular JS 指令</title>
   </head>
   <body>
   <script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
   <script src="directive.js"></script>
    <div ng-app = "demoApp">
        <!-- 注意这里我们使用的是脊柱式命名法（Spinal Casing） -->
        <demo-directive name="World"></demo-directive>

      </div>
   </body>
</html>
```

第6.2节：指令定义对象模板

```
demoApp.directive('demoDirective', function () {
  var directiveDefinitionObject = {
multiElement:
priority:
terminal:
scope: {},
    bindToController: {},
    controller:
controllerAs:
require:
restrict:
templateNamespace:
template:
templateUrl:
transclude:
compile:
link: function(){}
  };
  return directiveDefinitionObject;
});
```

1. multiElement - 设置为 true，指令名称开始和结束之间的任何 DOM 节点都将被收集并归类为指令元素
2. priority - 允许指定在单个 DOM 元素上定义多个指令时应用指令的顺序。数字较高的指令优先编译。single DOM element. Directives with higher numbers are compiled first.
3. terminal - 设置为 true 时，当前优先级将是执行的最后一组指令
4. scope - 设置指令的作用域
5. bind to controller - 将作用域属性直接绑定到指令控制器
6. controller - 控制器构造函数


```
      var linkFunction = function($scope, element, attributes) {
        element.html("Name: <b>"+$scope.name +"</b>");
        element.css("background-color", "#ff00ff");
      };
      return linkFunction;
    }
  };
});
```

This directive can then be used in App as :

```
<html>

   <head>
      <title>Angular JS Directives</title>
   </head>
   <body>
   <script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
   <script src="directive.js"></script>
    <div ng-app = "demoApp">
       <!-- Notice we are using Spinal Casing here -->
       <demo-directive name="World"></demo-directive>

      </div>
   </body>
</html>
```

Section 6.2: Directive Definition Object Template

```
demoApp.directive('demoDirective', function () {
  var directiveDefinitionObject = {
    multiElement:
    priority:
    terminal:
    scope: {},
    bindToController: {},
    controller:
    controllerAs:
    require:
    restrict:
    templateNamespace:
    template:
    templateUrl:
    transclude:
    compile:
    link: function(){}
  };
  return directiveDefinitionObject;
});
```

1. **multiElement** - set to true and any DOM nodes between the start and end of the directive name will be collected and grouped together as directive elements
2. **priority** - allows specification of the order to apply directives when multiple directives are defined on a single DOM element. Directives with higher numbers are compiled first.
3. **terminal** - set to true and the current priority will be the last set of directives to execute
4. **scope** - sets scope of the directive
5. **bind to controller** - binds scope properties directly to directive controller
6. **controller** - controller constructor function

7. require - 需要另一个指令，并将其控制器作为第四个参数注入链接函数
8. controllerAs - 在指令作用域中对控制器的名称引用，允许从指令模板中引用控制器。referenced from the directive template.
9. restrict - 限制指令为元素、属性、类或注释
10. templateNameSpace - 设置指令模板使用的文档类型：html、svg 或 math。默认是 html
11. template - HTML 标记，默认替换指令元素的内容，或包裹该内容如果 transclude 为 true，则为指令元素的内容
12. templateUrl - 异步提供的模板 URL
13. transclude - 提取指令出现的元素内容并使其可用于指令。内容被编译并作为转录函数提供给指令。
14. compile - 用于转换模板DOM的函数
15. link - 仅在未定义 compile 属性时使用。link 函数负责注册 DOM 监听器以及更新DOM。它在模板被克隆后执行。

第6.3节：如何使用指令创建可重用组件

AngularJS指令控制AngularJS应用程序中HTML的渲染。它们可以是HTML元素、属性、类或注释。指令用于操作DOM，附加新的行为到HTML元素，数据绑定等。Angular提供的一些指令示例有ng-model、ng-hide、ng-if。

同样，可以创建自己的自定义指令并使其可重用。创建自定义指令的参考。创建可重用指令的意义在于让你编写的一组指令/组件就像AngularJS通过angular.js提供的一样。这些可重用指令在你拥有一套需要一致行为、外观和感觉的应用程序/应用套件时特别有用。这样一个可重用组件的例子可以是一个简单的工具栏，你可能想在整个应用程序或不同应用程序中使用它们，但希望它们的行为或外观保持一致。

首先，在你的应用文件夹中创建一个名为resuableComponents的文件夹，并创建reusableModuleApp.js文件

reusableModuleApp.js：

```
(function(){

 var reusableModuleApp = angular.module('resuableModuleApp', ['ngSanitize']);

 //请记住，这里的所有依赖项都应注入到计划使用它的应用模块中，或者其脚本应包含在你的主应用中

                                    //我们将注入ng-sanitize

resubaleModuleApp.directive('toolbar', toolbar)

 toolbar.$inject=['$sce'];

 function toolbar($sce){

      return{
restrict :'AE',
        //定义以下隔离作用域实际上为指令提供了一个窗口，以便从使用该指令的应用程序中获取数据。

scope : {
                  value1: '=',
                  value2: '=',
                  },
```


7. **require** - require another directive and inject its controller as the fourth argument to the linking function
8. **controllerAs** - name reference to the controller in the directive scope to allow the controller to be referenced from the directive template.
9. **restrict** - restrict directive to Element, Attribute, Class, or Comment
10. **templateNameSpace** - sets document type used by directive template: html, svg, or math. html is the default
11. **template** - html markup that defaults to replacing the content of the directive's element, or wraps the contents of the directive element if transclude is true
12. **templateUrl** - url provided asynchronously for the template
13. **transclude** - Extract the contents of the element where the directive appears and make it available to the directive. The contents are compiled and provided to the directive as a transclusion function.
14. **compile** - function to transform the template DOM
15. **link** - only used if the compile property is not defined. The link function is responsible for registering DOM listeners as well as updating the DOM. It is executed after the template has been cloned.

Section 6.3: How to create resuable component using directive

AngularJS directives are what controls the rendering of the HTML inside an AngularJS application. They can be an Html element, attribute, class or a comment. Directives are used to manipulate the DOM, attaching new behavior to HTML elements, data binding and many more. Some of examples of directives which angular provides are ng-model, ng-hide, ng-if.

Similarly one can create his own custom directive and make them resuable. For creating Custom directives Reference. The sense behind creating reusable directives is to make a set of directives/components written by you just like angularjs provides us using angular.js . These reusable directives can be particularly very helpful when you have suite of applications/application which requires a consistent behavior, look and feel. An example of such reusable component can be a simple toolbar which you may want to use across your application or different applications but you want them to behave the same or look the same.

Firstly , Make a folder named resuableComponents in your app Folder and make reusableModuleApp.js

reusableModuleApp.js:

```
(function(){

 var reusableModuleApp = angular.module('resuableModuleApp', ['ngSanitize']);

 //Remember whatever dependencies you have in here should be injected in the app module where it is
intended to be used or it's scripts should be included in your main app
                                    //We will be injecting ng-sanitize

 resubaleModuleApp.directive('toolbar', toolbar)

 toolbar.$inject=['$sce'];

 function toolbar($sce){

     return{
     restrict :'AE',
        //Defining below isolate scope actually provides window for the directive to take data from
app that will be using this.
        scope : {
                value1: '=',
                value2: '=',
                },
```

```
        }
template : '<ul>  <li><a ng-click="Add()" href="">{{value1}}</a></li>  <li><a ng-
click="Edit()" href="#">{{value2}}</a></li> </ul> ',
link : function(scope, element, attrs){

            //处理 Add 函数
scope.Add = function(){


            };

            // 处理编辑功能
scope.编辑 = 函数(){

            };
    }
  }

});
```

mainApp.js：

```
(function(){
    var mainApp = angular.module('mainApp', ['reusableModuleApp']); // 在需要使用工具栏组件的应用中注入 reusable
ModuleApp

mainApp.controller('mainAppController', 函数($scope){
      $scope.value1 = "添加";
$scope.value2 = "编辑";

   });

});
```

index.html：

```
<!doctype html>
<html ng-app="mainApp">
<head>
 <title> 演示制作可重用组件

 <body ng-controller="mainAppController">

    <!-- 我们通过 mainApp 控制器向工具栏指令提供数据 -->
    <toolbar value1="value1" value2="value2"></toolbar>

   <!-- 这里需要为两个应用添加依赖的 js 文件 -->
    <script src="js/angular.js"></script>
    <script src="js/angular-sanitize.js"></script>

    <!-- 你的 mainApp.js 应该在此之后添加 --->
     <script src="mainApp.js"></script>

    <!-- 在这里添加你的可重用组件 js 文件 -->
     <script src="resuableComponents/reusableModuleApp.js"></script>

 </body>
</html>
```

指令默认是可重用组件。当你在单独的 Angular 模块中创建指令时，它实际上


```
        }
    template : '<ul>  <li><a ng-click="Add()" href="">{{value1}}</a></li>  <li><a ng-
click="Edit()" href="#">{{value2}}</a></li> </ul> ',
        link : function(scope, element, attrs){

            //Handle's Add function
            scope.Add = function(){


            };

            //Handle's Edit function
            scope.Edit = function(){

            };
    }
  }

});
```

mainApp.js:

```
(function(){
   var mainApp = angular.module('mainApp', ['reusableModuleApp']); //Inject resuableModuleApp in
your application where you want to use toolbar component

   mainApp.controller('mainAppController', function($scope){
      $scope.value1 = "Add";
      $scope.value2 = "Edit";

   });

});
```

index.html:

```
<!doctype html>
<html ng-app="mainApp">
<head>
 <title> Demo Making a reusable component
<head>
 <body ng-controller="mainAppController">

    <!-- We are providing data to toolbar directive using mainApp'controller -->
    <toolbar value1="value1" value2="value2"></toolbar>

   <!-- We need to add the dependent js files on both apps here -->
    <script src="js/angular.js"></script>
    <script src="js/angular-sanitize.js"></script>

    <!-- your mainApp.js should be added afterwards --->
     <script src="mainApp.js"></script>

    <!-- Add your reusable component js files here -->
     <script src="resuableComponents/reusableModuleApp.js"></script>

 </body>
</html>
```

Directive are reusable components by default. When you make directives in separate angular module, It actually

使其可以导出并在不同的 AngularJS 应用中重用。新的指令可以简单地添加到 reusableModuleApp.js 中，reusableModuleApp 可以拥有自己的控制器、服务、指令定义对象（DDO）来定义行为。

第6.4节：基本指令示例

superman-directive.js

```
angular.module('myApp', [])
  .directive('superman', function() {
    return {
      // 限制指令的使用方式
restrict: 'E',
templateUrl: 'superman-template.html',
      controller: function() {
        this.message = "我是超人！"
      },
controllerAs: 'supermanCtrl',
      // 在 Angular 初始化后执行。通常用于
      // 添加事件处理程序和 DOM 操作
link: function(scope, element, attributes) {
        element.on('click', function() {
alert('我是超人！')
        });
      }
    }
  });
```

superman-template.html

```
<h2>{{supermanCtrl.message}}</h2>
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>文档</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.0/angular.js"></script>
  <script src="superman-directive.js"><script/>
</head>
<body>
<div ng-app="myApp">
  <superman></superman>
</div>
</body>
</html>
```

您可以在AngularJS的官方文档中查看有关指令的restrict和link函数的更多信息，链接为 Directives

第6.5节：指令装饰器

有时你可能需要从指令中获得额外的功能。与其重写（复制）指令，不如修改指令的行为。

makes it exportable and reusable across different angularJs applications. New directives can simply be added inside reusableModuleApp.js and reusableModuleApp can have it's own controller, services, DDO object inside directive to define the behavior.

Section 6.4: Basic Directive example

superman-directive.js

```
angular.module('myApp', [])
  .directive('superman', function() {
    return {
      // restricts how the directive can be used
      restrict: 'E',
      templateUrl: 'superman-template.html',
      controller: function() {
        this.message = "I'm superman!"
      },
      controllerAs: 'supermanCtrl',
      // Executed after Angular's initialization. Use commonly
      // for adding event handlers and DOM manipulation
      link: function(scope, element, attributes) {
        element.on('click', function() {
          alert('I am superman!')
        });
      }
    }
  });
```

superman-template.html

```
<h2>{{supermanCtrl.message}}</h2>
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.0/angular.js"></script>
  <script src="superman-directive.js"><script/>
</head>
<body>
<div ng-app="myApp">
  <superman></superman>
</div>
</body>
</html>
```

You can check out more about directive's `restrict` and `link` functions on AngularJS's official documentation on Directives

Section 6.5: Directive decorator

Sometimes you may need additional features from a directive. Instead of rewriting (copy) the directive, you can modify how the directive behaves.

装饰器将在 $inject 阶段执行。

为此，向你的模块提供一个 .config。指令名为 myDirective，所以你需要配置 myDirectiveDirective。（这是 Angular 的约定 [阅读关于提供者的内容]）。

此示例将更改指令的 templateUrl：

```
angular.module('myApp').config(function($provide){
        $provide.decorator('myDirectiveDirective', function($delegate){
              var directive = $delegate[0]; // 这是实际委托的指令
              directive.templateUrl = 'newTemplate.html'; // 你更改指令的模板
          return $delegate;
    })
});
```

此示例在指令元素上添加了一个点击事件，该事件在编译阶段发生。

```
angular.module('myApp').config(function ($provide) {
        $provide.decorator('myDirectiveTwoDirective', function ($delegate) {
            var directive = $delegate[0];
            var link = directive.link; // 这是指令的链接阶段
            directive.compile = function () { // 更改该指令的编译函数
                return function (scope, element, attrs) {
link.apply(this, arguments); // 在链接阶段应用此函数                    element.on('click', functio
n(){ // 当点击指令时添加一个点击事件，记录 hello。

console.log('hello!');
                    });
                };
            };
            return $delegate;
        });

    });
```

类似的方法可以用于提供者（Providers）和服务（Services）。

第6.6节：带模板和隔离作用域的基本指令

创建一个带有隔离作用域的自定义指令，将指令内部的作用域与外部作用域分开，以防止指令意外修改父作用域中的数据，并限制其读取父作用域中的私有数据。

为了创建隔离作用域并且仍允许自定义指令与外部作用域通信，我们可以使用scope选项，描述如何映射指令内部作用域与外部作用域的绑定关系。

实际的绑定是通过附加到指令上的额外属性来完成的。绑定设置通过scope选项和一个包含键值对的对象来定义：

- 一个**键**，对应于指令的隔离作用域属性
- 一个**值**，告诉Angular如何将指令内部作用域绑定到匹配的**属性**

带隔离作用域的指令简单示例：

```
var ProgressBar = function() {
```


The decorator will be executed during $inject phase.

To do so, provde a .config to your module. The directive is called myDirective, so you have to config myDirectiveDirective. (this in an angular convention [read about providers]).

This example will change the templateUrl of the directive:

```
angular.module('myApp').config(function($provide){
        $provide.decorator('myDirectiveDirective', function($delegate){
             var directive = $delegate[0]; // this is the actual delegated, your directive
             directive.templateUrl = 'newTemplate.html'; // you change the directive template
        return $delegate;
    })
});
```

This example add an onClick event to the directive element when clicked, this happens during compile phase.

```
angular.module('myApp').config(function ($provide) {
        $provide.decorator('myDirectiveTwoDirective', function ($delegate) {
            var directive = $delegate[0];
            var link = directive.link; // this is directive link phase
            directive.compile = function () { // change the compile of that directive
                return function (scope, element, attrs) {
                    link.apply(this, arguments); // apply this at the link phase
                    element.on('click', function(){ // when add an onclick that log hello when the
directive is clicked.
                            console.log('hello!');
                    });
                };
            };
            return $delegate;
        });

    });
```

Similar approach can be used for both Providers and Services.

Section 6.6: Basic directive with template and an isolated scope

Creating a custom directive with *isolated scope* will separate the scope **inside** the directive from the **outside** scope, in order to prevent our directive from accidentally change the data in the parent scope and restricting it from reading private data from the parent scope.

To create an isolated scope and still allow our custom directive to communicate with the outside scope, we can use the `scope` option that describe how to **map** the bindings of the directive's inner scope with the outside scope.

The actual bindings are made with extra **attributes** attached to the directive. The binding settings are defined with the `scope` option and an object with key-value pairs:

- A **key**, which is corresponded to our directive's isolated scope property
- A **value**, which tells Angular how do bind the directive inner scope to a matching **attribute**

Simple example of a directive with an isolated scope:

```
var ProgressBar = function() {
```

```
  return {
scope: { // 这是我们定义隔离作用域的方式       current: '=', // 通过
使用 'current' 属性创建一个必需的双向绑定       full: '=?maxValue' // 通过 'max-value' 属性创建一个可选的（注意 '?'）
双向绑定，绑定到指令隔离作用域中的 `full` 属性

    }
template: '<div class="progress-back">' +
               '  <div class="progress-bar"' +
               '       ng-style="{width: getProgress()}">' +
               '  </div>' +
              '</div>',
link: function(scope, el, attrs) {
      if (scope.full === undefined) {
        scope.full = 100;
      }
scope.getProgress = function() {
        return (scope.current / scope.size * 100) + '%';
      }
    }
  }
}

ProgressBar.$inject = [];
angular.module('app').directive('progressBar', ProgressBar);
```

示例：如何使用此指令并将控制器作用域中的数据绑定到指令的内部作用域：

控制器：

```
angular.module('app').controller('myCtrl', function($scope) {
    $scope.currentProgressValue = 39;
$scope.maxProgressBarValue = 50;
});
```

视图：

```
<div ng-controller="myCtrl">
    <progress-bar current="currentProgressValue"></progress-bar>
    <progress-bar current="currentProgressValue" max-value="maxProgressBarValue"></progress-bar>
```

第6.7节：构建可复用组件

指令可以用来构建可复用的组件。以下是一个“用户框”组件的示例：

userBox.js

```
angular.module('simpleDirective', []).directive('userBox', function() {
  return {
scope: {
username: '=username',
        reputation: '=reputation'
    },
templateUrl: '/path/to/app/directives/user-box.html'
  };
});
```

Controller.js


```
  return {
    scope: { // This is how we define an isolated scope
      current: '=', // Create a REQUIRED bidirectional binding by using the 'current' attribute
      full: '=?maxValue' // Create an OPTIONAL (Note the '?'): bidirectional binding using 'max-
value' attribute to the `full` property in our directive isolated scope
    }
    template: '<div class="progress-back">' +
              '  <div class="progress-bar"' +
              '       ng-style="{width: getProgress()}">' +
              '  </div>' +
              '</div>',
    link: function(scope, el, attrs) {
      if (scope.full === undefined) {
        scope.full = 100;
      }
      scope.getProgress = function() {
        return (scope.current / scope.size * 100) + '%';
      }
    }
  }
}

ProgressBar.$inject = [];
angular.module('app').directive('progressBar', ProgressBar);
```

Example how to use this directive and bind data from the controller's scope to the directive's inner scope:

Controller:

```
angular.module('app').controller('myCtrl', function($scope) {
    $scope.currentProgressValue = 39;
    $scope.maxProgressBarValue = 50;
});
```

View:

```
<div ng-controller="myCtrl">
    <progress-bar current="currentProgressValue"></progress-bar>
    <progress-bar current="currentProgressValue" max-value="maxProgressBarValue"></progress-bar>
</div>
```

Section 6.7: Building a reusable component

Directives can be used to build reusable components. Here is an example of a "user box" component:

userBox.js

```
angular.module('simpleDirective', []).directive('userBox', function() {
  return {
    scope: {
        username: '=username',
        reputation: '=reputation'
    },
    templateUrl: '/path/to/app/directives/user-box.html'
  };
});
```

Controller.js

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

myApp.controller('Controller', function($scope) {

$scope.user = "约翰·多伊";
    $scope.rep = 1250;

$scope.user2 = "安德鲁";
    $scope.rep2 = 2850;

});
```

myPage.js

```
<html lang="en" ng-app="myApp">
  <head>
    <script src="/path/to/app/angular.min.js"></script>
    <script src="/path/to/app/js/controllers/Controller.js"></script>
    <script src="/path/to/app/js/directives/userBox.js"></script>
  </head>

  <body>

    <div ng-controller="Controller">
        <user-box username="user" reputation="rep"></user-box>
        <user-box username="user2" reputation="rep2"></user-box>
    </div>

  </body>
</html>
```

user-box.html

```
<div>{{username}}</div>
<div>{{reputation}} 声望</div>
```

结果将是：

```
约翰·多伊
1250声誉
安德鲁
2850声誉
```

第6.8节：指令继承与互操作性

Angular js 指令可以嵌套或实现互操作。

在此示例中，指令 Adir 将其控制器 $scope 暴露给指令 Bdir，因为 Bdir 依赖 Adir。

```
angular.module('我的应用',[]).指令('Adir', 函数 () {
        返回 {
            restrict: 'AE',
            controller: ['$scope', function ($scope) {
                    $scope.logFn = function (val) {
                        console.log(val);
                    }
                }]
            }
```



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

myApp.controller('Controller', function($scope) {

    $scope.user = "John Doe";
    $scope.rep = 1250;

    $scope.user2 = "Andrew";
    $scope.rep2 = 2850;

});
```

myPage.js

```
<html lang="en" ng-app="myApp">
  <head>
    <script src="/path/to/app/angular.min.js"></script>
    <script src="/path/to/app/js/controllers/Controller.js"></script>
    <script src="/path/to/app/js/directives/userBox.js"></script>
  </head>

  <body>

    <div ng-controller="Controller">
        <user-box username="user" reputation="rep"></user-box>
        <user-box username="user2" reputation="rep2"></user-box>
    </div>

  </body>
</html>
```

user-box.html

```
<div>{{username}}</div>
<div>{{reputation}} reputation</div>
```

The result will be:

```
John Doe
1250 reputation
Andrew
2850 reputation
```

Section 6.8: Directive inheritance and interoperability

Angular js directives can be nested or be made interoperable.

In this example, directive Adir exposes to directive Bdir it's controller $scope, since Bdir requires Adir.

```
angular.module('myApp',[]).directive('Adir', function () {
        return {
            restrict: 'AE',
            controller: ['$scope', function ($scope) {
                    $scope.logFn = function (val) {
                        console.log(val);
                    }
                }]
            }
```

```
    })
```

确保设置 require: '^Adir'（查看 Angular 文档，某些版本不需要 ^ 字符）。

```
.directive('Bdir', function () {
        return {
restrict: 'AE',
require: '^Adir', // Bdir 依赖 Adir
           link: function (scope, elem, attr, Parent) {
              // Parent 是 Adir，但可能是所需指令的数组。
elem.on('click', function ($event) {
                      Parent.logFn("Hello!"); // 会在父指令作用域中记录 "Hello!"
                      scope.$apply(); // 应用到父作用域。
                  });
              }
          }
    }]);
```

你可以这样嵌套你的指令：

```
<div a-dir><span b-dir></span></div>
<a-dir><b-dir></b-dir> </a-dir>
```

指令不必在你的 HTML 中嵌套。

belindoc.com


```
    })
```

Make sure to set require: '^Adir' (look at the angular documentation, some versions doesn't require ^ character).

```
.directive('Bdir', function () {
        return {
            restrict: 'AE',
            require: '^Adir', // Bdir require Adir
            link: function (scope, elem, attr, Parent) {
            // Parent is Adir but can be an array of required directives.
                elem.on('click', function ($event) {
                    Parent.logFn("Hello!"); // will log "Hello! at parent dir scope
                    scope.$apply(); // apply to parent scope.
                });
            }
        }
    }]);
```

You can nest your directive in this way:

```
<div a-dir><span b-dir></span></div>
<a-dir><b-dir></b-dir> </a-dir>
```

Is not required that directives are nested in your HTML.

第7章：数据绑定的工作原理

第7.1节：数据绑定示例

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

这个“message”必须绑定到当前元素控制器的作用域（scope）上。

```
$scope.message = "Hello World";
```

在稍后的时间点，即使消息模型被更新，更新后的值也会反映在HTML元素中。当Angular编译模板时，“Hello World”将被附加到当前元素的innerHTML中。Angular维护了一个对所有指令的监视机制。它有一个Digest循环机制，会遍历Watchers数组，如果模型的前一个值发生变化，则更新DOM元素。

作用域不会定期检查其附加对象是否有变化。并非所有附加到作用域的对象都会被监视。作用域以原型方式维护一个$$WatchersArray。只有在调用$digest时，作用域才会遍历这个WatchersArray。

Angular会为每个以下内容向WatchersArray添加一个监视器

1. {{expression}} — 在你的模板中（以及任何包含表达式的地方）或者当我们定义ng-model时。
2. $scope.$watch(‘expression/function’) — 在你的JavaScript中，我们可以直接为作用域对象附加一个监视器观察的角度。

$watch函数接受三个参数：

1.第一个是一个观察者函数，它返回对象，或者我们也可以直接添加一个表达式。

2.第二个是一个监听函数，当对象发生变化时会调用它。所有像DOM变化这样的操作都会在这个函数中实现。

3.第三个是一个可选参数，接受一个布尔值。如果为true，Angular会深度监听对象；如果为false，Angular只会对对象进行引用监听。大致
$watch的实现大致如下

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
   var watcher = {
watchFn: watchFn,
listenerFn: listenerFn || function() { },
       last: initWatchVal  // initWatchVal 通常是未定义的
   };
   this.$$watchers.push(watcher); // 将Watcher对象推入Watchers数组
};
```

Angular中有一个有趣的概念叫做Digest循环。$digest循环是由调用$scope.$digest()触发的。假设你通过ng-click指令在一个处理函数中改变了$scope模型。在

Chapter 7: How data binding works

Section 7.1: Data Binding Example

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

This 'message' has to be attached to the current elements controller's scope.

```
$scope.message = "Hello World";
```

At a later point of time , even if the message model is updated , that updated value is reflected in the HTML element. When angular compiles the template "Hello World" will be attached to the innerHTML of the current world. Angular maintains a Watching mechanism of all the directives atttached to the view. It has a Digest Cycle mechanism where it iterates through the Watchers array, it will update the DOM element if there is a change in the previous value of the model.

There is no periodic checking of Scope whether there is any change in the Objects attached to it. Not all the objects attached to scope are watched . Scope prototypically maintains a **$$WatchersArray** . Scope only iterates through this WatchersArray when $digest is called .

Angular adds a watcher to the WatchersArray for each of these

1. {{expression}}—In your templates (and anywhere else where there’s an expression) or when we define ng-model.
2. $scope.$watch(‘expression/function’)—In your JavaScript we can just attach a scope object for angular to watch.

$watch function takes in three parameters:

1. First one is a watcher function which just returns the object or we can just add an expression.

2. Second one is a listener function which will be called when there is a change in the object. All the things like DOM changes will be implemented in this function.

3. The third being an optional parameter which takes in a boolean . If its true , angular deep watches the object & if its false Angular just does a reference watching on the object. Rough Implementation of $watch looks like this

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
   var watcher = {
       watchFn: watchFn,
       listenerFn: listenerFn || function() { },
       last: initWatchVal  // initWatchVal is typically undefined
   };
   this.$$watchers.push(watcher); // pushing the Watcher Object to Watchers
};
```

There is an interesting thing in Angular called Digest Cycle. The $digest cycle starts as a result of a call to $scope.$digest(). Assume that you change a $scope model in a handler function through the ng-click directive. In

在这种情况下，AngularJS 会自动通过调用 $digest() 触发一个 $digest 循环。除了 ng-click，还有几个内置的指令/服务允许你更改模型（例如 ng-model、$timeout 等）并自动触发 $digest 循环。$digest 的大致实现如下。

```
Scope.prototype.$digest = function() {
      var dirty;
      do {
dirty = this.$$digestOnce();
      } while (dirty);
}
Scope.prototype.$$digestOnce = function() {
   var self = this;
   var newValue, oldValue, dirty;
   _.forEach(this.$$watchers, function(watcher) {
          newValue = watcher.watchFn(self);
oldValue = watcher.last;   // 它只是记住上一次的值以进行脏检查
          if (newValue !== oldValue) { // 引用的脏检查
   // 对对象进行深度检查，应该在这里实现基于值的对象检查代码

watcher.last = newValue;
             watcher.listenerFn(newValue,
                  (oldValue === initWatchVal ? newValue : oldValue),
                   self);
dirty = true;
          }
     });
   return dirty;
 };
```

如果我们使用JavaScript's **setTimeout()**函数来更新作用域模型，Angular无法知道你可能会更改什么。在这种情况下，我们有责任手动调用$apply()，这会触发一个$digest循环。同样，如果你有一个指令设置了DOM事件监听器并在处理函数内更改了一些模型，你需要调用$apply()以确保更改生效。$apply的核心思想是我们可以执行一些不知晓Angular的代码，这些代码仍可能更改作用域上的内容。如果我们用$apply包裹这段代码，它会负责调用$digest()。$apply()的粗略实现。

```
Scope.prototype.$apply = function(expr) {
       try {
         return this.$eval(expr); //在Scope上下文中执行代码
       } finally {
         this.$digest();
       }
};
```


that case AngularJS automatically triggers a $digest cycle by calling $digest().In addition to ng-click, there are several other built-in directives/services that let you change models (e.g. ng-model, $timeout, etc) and automatically trigger a $digest cycle. The rough implementation of $digest looks like this.

```
Scope.prototype.$digest = function() {
      var dirty;
      do {
          dirty = this.$$digestOnce();
      } while (dirty);
}
Scope.prototype.$$digestOnce = function() {
   var self = this;
   var newValue, oldValue, dirty;
   _.forEach(this.$$watchers, function(watcher) {
          newValue = watcher.watchFn(self);
          oldValue = watcher.last;   // It just remembers the last value for dirty checking
          if (newValue !== oldValue) { //Dirty checking of References
   // For Deep checking the object , code of Value
   // based checking of Object should be implemented here
             watcher.last = newValue;
             watcher.listenerFn(newValue,
                  (oldValue === initWatchVal ? newValue : oldValue),
                   self);
          dirty = true;
          }
     });
   return dirty;
 };
```

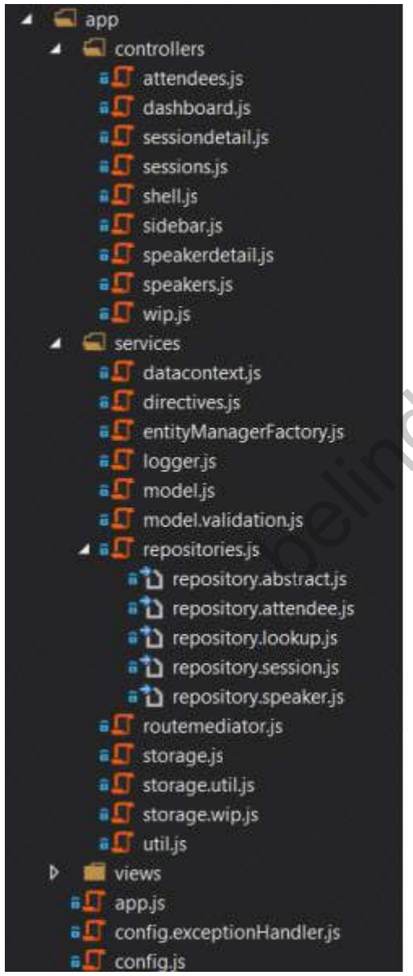
If we use JavaScript's **setTimeout()** function to update a scope model, Angular has no way of knowing what you might change. In this case it's our responsibility to call $apply() manually, which triggers a $digest cycle. Similarly, if you have a directive that sets up a DOM event listener and changes some models inside the handler function, you need to call $apply() to ensure the changes take effect. The big idea of $apply is that we can execute some code that isn't aware of Angular, that code may still change things on the scope. If we wrap that code in $apply , it will take care of calling $digest(). Rough implementation of $apply().

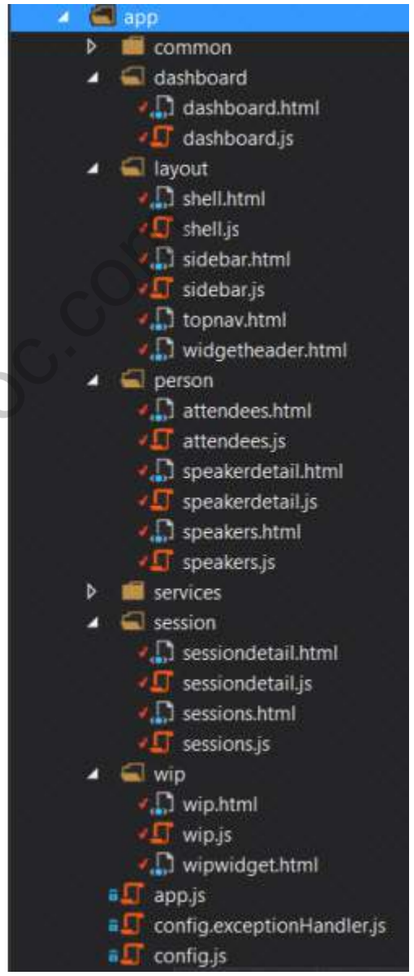
```
Scope.prototype.$apply = function(expr) {
       try {
         return this.$eval(expr); //Evaluating code in the context of Scope
       } finally {
         this.$digest();
       }
};
```

第8章：Angular项目 - 目录结构

第8.1节：目录结构

新手Angular程序员常问的一个问题——“项目结构应该如何设计？”。一个良好的结构有助于实现可扩展的应用开发。开始一个项目时，我们有两个选择，**按类型排序类型**（左）和**按功能排序**（右）。第二种更好，尤其是在大型应用中，项目管理起来会容易得多。





按类型排序（左）

应用程序按文件类型组织。

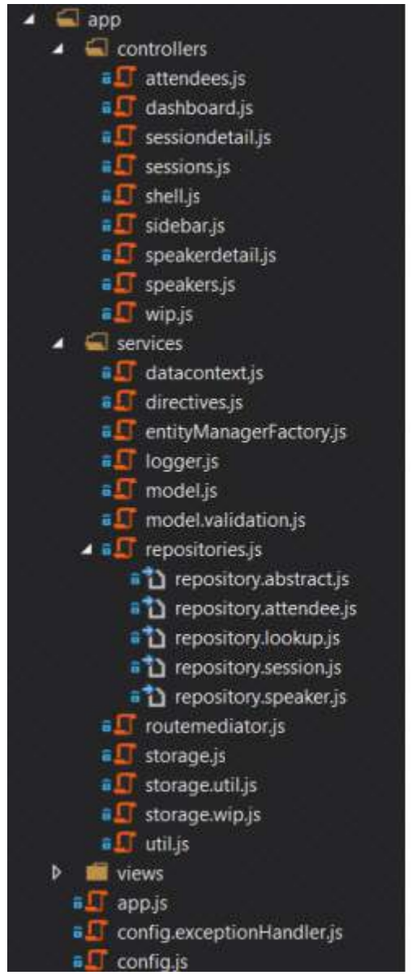
- 优点 - 适合小型应用程序，适合刚开始使用Angular的程序员，并且易于转换为第二种方法。
- 缺点 - 即使是小型应用程序，也开始变得更难找到特定文件。例如，视图和它的控制器位于两个不同的文件夹中。

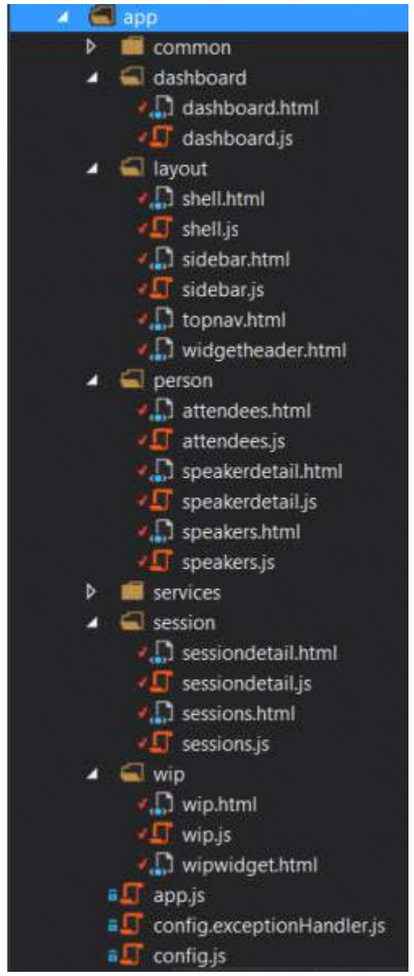
按功能排序（右侧）

Chapter 8: Angular Project - Directory Structure

Section 8.1: Directory Structure

A common question among new Angular programmers - "What should be the structure of the project?". A good structure helps toward a scalable application development. When we start a project we have two choices, **Sort By Type** (left) and **Sort By Feature** (right). The second is better, especially in large applications, the project becomes a lot easier to manage.





Sort By Type (left)

The application is organized by the files' type.

- **Advantage** - Good for small apps, for programmers only starting to use Angular, and is easy to convert to the second method.
- **Disadvantage** - Even for small apps it starts to get more difficult to find a specific file. For instance, a view and it's controller are in two separate folders.

Sort By Feature (right)

建议的组织方法，文件按功能类型排序。

所有布局视图和控制器放在布局文件夹，管理员内容放在管理员文件夹，依此类推。

- 优点 - 查找确定某个功能的代码部分时，所有内容都集中在一个文件夹中。
- 缺点 - 服务有点不同，因为它们“service”许多功能。

你可以在Angular Structure: Refactoring for Growth

上阅读更多相关内容。建议的文件结构结合了上述两种方法：

```
├── app
│   ├── app.js
│   ├── controllers
│   │   ├── home
│   │   │   ├── FirstCtrl.js
│   │   │   └── FirstCtrl.spec.js
│   │   │   └── SecondCtrl.js
│   │   │   └── SecondCtrl.spec.js
│   │   └── about
│   │       └── ThirdCtrl.js
│   │       └── ThirdCtrl.spec.js
│   ├── directives
│   │   ├── home
│   │   │   └── directive1.js
│   │   │   └── directive1.spec.js
│   │   └── about
│   │       ├── directive2.js
│   │       ├── directive2.spec.js
│   │       └── directive3.js
│   │       └── directive3.spec.js
│   ├── filters
│   │   ├── home
│   │   └── about
│   └── services
│       ├── CommonService.js
│       ├── CommonService.spec.js
│       ├── cache
│       │   ├── Cache1.js
│       │   ├── Cache1.spec.js
│       │   └── Cache2.js
│       │   └── Cache2.spec.js
│       └── models
│           ├── Model1.spec.js
│           ├── Model1.js
│           └── Model2.spec.js
```

bellindoc.com

致谢：Angular 风格指南

The suggested organizing method where the filed are sorted by features' type.

All of the layout views and controllers go in the layout folder, the admin content goes in the admin folder, and so on.

- **Advantage** - When looking for a section of code determining a certain feature it's all located in one folder.
- **Disadvantage** - Services are a bit different as they “service” many features.

You can read more about it on Angular Structure: Refactoring for Growth

The suggested file structure combining both of the aforementioned methods:

```
├── app
│   ├── app.js
│   ├── controllers
│   │   ├── home
│   │   │   ├── FirstCtrl.js
│   │   │   └── FirstCtrl.spec.js
│   │   │   └── SecondCtrl.js
│   │   │   └── SecondCtrl.spec.js
│   │   └── about
│   │       └── ThirdCtrl.js
│   │       └── ThirdCtrl.spec.js
│   ├── directives
│   │   ├── home
│   │   │   └── directive1.js
│   │   │   └── directive1.spec.js
│   │   └── about
│   │       ├── directive2.js
│   │       ├── directive2.spec.js
│   │       └── directive3.js
│   │       └── directive3.spec.js
│   ├── filters
│   │   ├── home
│   │   └── about
│   └── services
│       ├── CommonService.js
│       ├── CommonService.spec.js
│       ├── cache
│       │   ├── Cache1.js
│       │   ├── Cache1.spec.js
│       │   └── Cache2.js
│       │   └── Cache2.spec.js
│       └── models
│           ├── Model1.spec.js
│           ├── Model1.js
│           └── Model2.spec.js
```

Credit to: Angular Style Guide

第9章：过滤器

第9.1节：从ng-repeat外部访问过滤后的列表

有时你会想从ng-repeat外部访问过滤器的结果，可能是为了显示
被过滤掉的项目数量。你可以使用as[变量名]语法在ng-repeat中实现这一点。

```
<ul>
  <li ng-repeat="item in vm.listItems | filter:vm.myFilter as filtered">
   {{item.name}}
  </li>
</ul>
<span>显示 {{filtered.length}} / {{vm.listItems.length}}</span>
```

第9.2节：自定义过滤器以移除值

过滤器的一个典型用例是从数组中移除值。在此示例中，我们传入一个数组并移除其中的所有
null值，返回处理后的数组。

```
function removeNulls() {
    return function(list) {
        for (var i = list.length - 1; i >= 0; i--) {
            if (typeof list[i] === 'undefined' ||
                    list[i] === null) {
                list.splice(i, 1);
            }
        }
        return list;
    };
}
```

这将在HTML中使用，如

```
{{listOfItems | removeNulls}}
```

或者在控制器中使用，如

```
listOfItems = removeNullsFilter(listOfItems);
```

第9.3节：用于格式化值的自定义过滤器

过滤器的另一个用例是格式化单个值。在此示例中，我们传入一个值，返回一个合适的布尔真值。

```
function convertToBooleanValue() {
    return function(input) {
        if (typeof input !== 'undefined' &&
input !== null &&
                (input === true || input === 1 || input === '1' || input
                        .toString().toLowerCase() === 'true')) {
            return true;
        }
        return false;
    };
}
```


Chapter 9: Filters

Section 9.1: Accessing a filtered list from outside an ng-repeat

Occasionally you will want to access the result of your filters from outside the `ng-repeat`, perhaps to indicate the number of items that have been filtered out. You can do this using as `[variablename]` syntax on the `ng-repeat`.

```
<ul>
  <li ng-repeat="item in vm.listItems | filter:vm.myFilter as filtered">
    {{item.name}}
  </li>
</ul>
<span>Showing {{filtered.length}} of {{vm.listItems.length}}</span>
```

Section 9.2: Custom filter to remove values

A typical use case for a filter is to remove values from an array. In this example we pass in an array and remove any nulls found in it, returning the array.

```
function removeNulls() {
    return function(list) {
        for (var i = list.length - 1; i >= 0; i--) {
            if (typeof list[i] === 'undefined' ||
                    list[i] === null) {
                list.splice(i, 1);
            }
        }
        return list;
    };
}
```

That would be used in the HTML like

```
{{listOfItems | removeNulls}}
```

or in a controller like

```
listOfItems = removeNullsFilter(listOfItems);
```

Section 9.3: Custom filter to format values

Another use case for filters is to format a single value. In this example, we pass in a value and we are returned an appropriate true boolean value.

```
function convertToBooleanValue() {
    return function(input) {
        if (typeof input !== 'undefined' &&
                input !== null &&
                (input === true || input === 1 || input === '1' || input
                        .toString().toLowerCase() === 'true')) {
            return true;
        }
        return false;
    };
}
```

在 HTML 中的使用方式如下：

```
{{isAvailable | convertToBooleanValue}}
```

或者在控制器中这样使用：

```
var available = convertToBooleanValueFilter(isAvailable);
```

第9.4节：在控制器或服务中使用过滤器

通过注入 `$filter`，可以在控制器、服务、指令甚至其他过滤器中使用 Angular 模块中定义的任何过滤器。

```
angular.module("app")
.service("users", usersService)
  .controller("UsersController", UsersController);

function usersService () {
  this.getAll = function () {
    return [{
id: 1,
      username: "john"
    }, {
id: 2,
      username: "will"
    }, {
id: 3,
      username: "jack"
    }];
  };
}

function UsersController ($filter, users) {
  var orderByFilter = $filter("orderBy");

  this.users = orderByFilter(users.getAll(), "username");
  // 现在用户已按用户名排序：jack、john、will

  this.users = orderByFilter(users.getAll(), "username", true);
  // 现在用户已按用户名逆序排序：will、john、jack
}
```

第9.5节：对子数组执行过滤

此示例旨在演示如何在子数组中执行深度过滤，而无需自定义过滤器。

控制器：

```
(function() {
  "use strict";
  angular
.module('app', [])
    .controller('mainCtrl', mainCtrl);

  function mainCtrl() {
    var vm = this;
```


Which in the HTML would be used like this:

```
{{isAvailable | convertToBooleanValue}}
```

Or in a controller like:

```
var available = convertToBooleanValueFilter(isAvailable);
```

Section 9.4: Using filters in a controller or service

By injecting `$filter`, any defined filter in your Angular module may be used in controllers, services, directives or even other filters.

```
angular.module("app")
  .service("users", usersService)
  .controller("UsersController", UsersController);

function usersService () {
  this.getAll = function () {
    return [{
      id: 1,
      username: "john"
    }, {
      id: 2,
      username: "will"
    }, {
      id: 3,
      username: "jack"
    }];
  };
}

function UsersController ($filter, users) {
  var orderByFilter = $filter("orderBy");

  this.users = orderByFilter(users.getAll(), "username");
  // Now the users are ordered by their usernames: jack, john, will

  this.users = orderByFilter(users.getAll(), "username", true);
  // Now the users are ordered by their usernames, in reverse order: will, john, jack
}
```

Section 9.5: Performing filter in a child array

This example was done in order to demonstrate how you can perform a deep filter in a *child* array without the necessity of a custom filter.

Controller:

```
(function() {
  "use strict";
  angular
    .module('app', [])
    .controller('mainCtrl', mainCtrl);

  function mainCtrl() {
    var vm = this;
```

```
vm.classifications = ["轿车", "轿车", "商用车", "跑车"];
    vm.cars = [
       {
          "name":"car1",
          "classifications":[
             {
                "name":"轿车"
             },
             {
                "name":"轿车"
             }
          ]
       },
       {
          "name":"car2",
          "classifications":[
             {
                "name":"轿车"
             },
             {
                "name":"商用车"
             }
          ]
       },
       {
          "name":"car3",
          "classifications":[
             {
                "name":"跑车"
             },
             {
                "name":"轿车"
             }
          ]
       }
    ];
  }
})();
```

视图：

```
<body ng-app="app" ng-controller="mainCtrl as main">
  按分类筛选汽车：
   <select ng-model="classificationName"
           ng-options="classification for classification in main.classifications"></select>
   <br>
   <ul>
     <li ng-repeat="car in main.cars |
filter: { classifications: { name: classificationName } } track by $index"
                 ng-bind-template="{{car.name}} - {{car.classifications | json}}">
     </li>
   </ul>
</body>
```

查看完整 演示.


```
vm.classifications = ["Saloons", "Sedans", "Commercial vehicle", "Sport car"];
vm.cars = [
   {
      "name":"car1",
      "classifications":[
         {
            "name":"Saloons"
         },
         {
            "name":"Sedans"
         }
      ]
   },
   {
      "name":"car2",
      "classifications":[
         {
            "name":"Saloons"
         },
         {
            "name":"Commercial vehicle"
         }
      ]
   },
   {
      "name":"car3",
      "classifications":[
         {
            "name":"Sport car"
         },
         {
            "name":"Sedans"
         }
      ]
   }
];
  }
})();
```

View:

```
<body ng-app="app" ng-controller="mainCtrl as main">
  Filter car by classification:
  <select ng-model="classificationName"
          ng-options="classification for classification in main.classifications"></select>
  <br>
  <ul>
    <li ng-repeat="car in main.cars |
                  filter: { classifications: { name: classificationName } } track by $index"
                ng-bind-template="{{car.name}} - {{car.classifications | json}}">
    </li>
  </ul>
</body>
```

Check the complete DEMO.

belindoc.com

第10章：自定义过滤器

第10.1节：在控制器、服务或过滤器中使用过滤器

你需要注入$filter：

```
angular
.module('filters', [])
  .filter('percentage', function($filter) {
    return function (input) {
      return $filter('number')(input * 100) + ' %';
    };
  });
```

第10.2节：创建带参数的过滤器

默认情况下，过滤器只有一个参数：它所应用的变量。但你可以向

函数传递更多参数：

```
angular
.module('app', [])
.controller('我的控制器', function($scope) {
    $scope.example = 0.098152;
  })
.filter('百分比', function($filter) {
      return function (input, decimals) {
        return $filter('number')(input * 100, decimals) + ' %';
      };
  });
```

现在，你可以为百分比过滤器指定精度：

```
<span ng-controller="我的控制器">{{ example | 百分比: 2 }}</span>
=> "9.81 %"
```

... 但其他参数是可选的，你仍然可以使用默认过滤器：

```
<span ng-controller="我的控制器">{{ example | 百分比 }}</span>
=> "9.8152 %"
```

第10.3节：简单过滤器示例

过滤器用于格式化表达式的值以显示给用户。它们可以在视图模板、控制器或服务中使用。此示例创建了一个过滤器（addZ），然后在视图中使用它。该过滤器所做的只是将大写字母“Z”添加到字符串末尾。

example.js

```
angular.module('main', [])
.filter('addZ', function() {
            return function(value) {
                return value + "Z";
            }
    })
.controller('MyController', ['$scope', function($scope) {
        $scope.sample = "hello";
```


Chapter 10: Custom filters

Section 10.1: Use a filter in a controller, a service or a filter

You will have to inject `$filter`:

```
angular
  .module('filters', [])
  .filter('percentage', function($filter) {
    return function (input) {
      return $filter('number')(input * 100) + ' %';
    };
  });
```

Section 10.2: Create a filter with parameters

By default, a filter has a single parameter: the variable it is applied on. But you can pass more parameter to the function:

```
angular
  .module('app', [])
  .controller('MyController', function($scope) {
    $scope.example = 0.098152;
  })
  .filter('percentage', function($filter) {
    return function (input, decimals) {
      return $filter('number')(input * 100, decimals) + ' %';
    };
  });
```

Now, you can give a precision to the `percentage` filter:

```
<span ng-controller="MyController">{{ example | percentage: 2 }}</span>
=> "9.81 %"
```

... but other parameters are optional, you can still use the default filter:

```
<span ng-controller="MyController">{{ example | percentage }}</span>
=> "9.8152 %"
```

Section 10.3: Simple filter example

Filters format the value of an expression for display to the user. They can be used in view templates, controllers or services. This example creates a filter (`addZ`) then uses it in a view. All this filter does is add a capital 'Z' to the end of the string.

example.js

```
angular.module('main', [])
    .filter('addZ', function() {
            return function(value) {
                return value + "Z";
            }
     })
    .controller('MyController', ['$scope', function($scope) {
        $scope.sample = "hello";
```


helindoc.com

```
    }])
```

example.html

在视图中，过滤器的使用语法如下：{变量|过滤器}。 在本例中，我们在控制器中定义的变量 sample，正被我们创建的过滤器 addZ过滤。

```
<div ng-controller="MyController">
   <span>{{sample | addZ}}</span>
</div>
```

预期输出

```
helloZ
```

belindoc.com


```
    }])
```

example.html

Inside the view, the filter is applied with the following syntax: `{ variable | filter}`. In this case, the variable we defined in the controller, `sample`, is being filtered by the filter we created, `addZ`.

```
<div ng-controller="MyController">
   <span>{{sample | addZ}}</span>
</div>
```

Expected output

```
helloZ
```

第11章：常量

第11.1节：创建你的第一个常量

```
angular
.module('MyApp', [])
  .constant('VERSION', 1.0);
```

你的常量现在已声明，可以注入到控制器、服务、工厂、提供者，甚至是
config方法中：

```
angular
.module('MyApp')
.controller('FooterController', function(VERSION) {
    this.version = VERSION;
  });
```

```
<footer ng-controller="FooterController as Footer">{{ Footer.version }}</footer>
```

第11.2节：使用场景

这里没有革命性的变化，但Angular常量特别有用，尤其是当你的应用和/或团队开始
增长时……或者如果你只是喜欢写漂亮的代码！

- 重构代码。以事件名称为例。如果你在应用中使用了大量事件，事件名称会分散在各处。当一个新开发者加入你的团队时，他可能会用不同的语法命名事件，……你可以通过将事件名称集中到一个常量中轻松防止这种情况：

```
angular
.module('MyApp')
.constant('事件', {
LOGIN_VALIDATE_FORM: 'login::click-validate',
    LOGIN_FORGOT_PASSWORD: 'login::click-forgot',
    LOGIN_ERROR: 'login::notify-error',
    ...
  });
```

```
angular
.module('MyApp')
.controller('LoginController', function($scope, EVENT) {
    $scope.$on(EVENT.LOGIN_VALIDATE_FORM, function() {
     ...
    });
  })
```

... 现在，您的事件名称可以享受自动补全的便利了！

- 定义配置。 将所有配置集中放置在同一位置：

```
angular
.module('MyApp')
  .constant('CONFIG', {
```


Chapter 11: Constants

Section 11.1: Create your first constant

```
angular
  .module('MyApp', [])
  .constant('VERSION', 1.0);
```

Your constant is now declared and can be injected in a controller, a service, a factory, a provider, and even in a config method:

```
angular
  .module('MyApp')
  .controller('FooterController', function(VERSION) {
    this.version = VERSION;
  });
```

```
<footer ng-controller="FooterController as Footer">{{ Footer.version }}</footer>
```

Section 11.2: Use cases

There is no revolution here, but angular constant can be useful specially when your application and/or team starts to grow ... or if you simply love writing beautiful code!

- **Refactor code.** Example with event's names. If you use a lot of events in your application, you have event's names a little every where. A when a new developer join your team, he names his events with a different syntax, ... You can easily prevent this by grouping your event's names in a constant:

```
angular
  .module('MyApp')
  .constant('EVENTS', {
    LOGIN_VALIDATE_FORM: 'login::click-validate',
    LOGIN_FORGOT_PASSWORD: 'login::click-forgot',
    LOGIN_ERROR: 'login::notify-error',
    ...
  });
```

```
angular
  .module('MyApp')
  .controller('LoginController', function($scope, EVENT) {
    $scope.$on(EVENT.LOGIN_VALIDATE_FORM, function() {
      ...
    });
  })
```

... and now, your event's names can take benefits from autocompletion !

- **Define configuration.** Locate all your configuration in a same place:

```
angular
  .module('MyApp')
  .constant('CONFIG', {
```

```
  BASE_URL: {
  APP: 'http://localhost:3000',
        API: 'http://localhost:3001'
      },
  STORAGE: 'S3',
      ...
    });
```

- 隔离部分。 有时，你可能对某些东西不太自豪.....比如硬编码的值。例如，不要让它们出现在你的主代码中，你可以创建一个 Angular 常量

```
angular
.module('MyApp')
  .constant('HARDCODED', {
    KEY: 'KEY',
  RELATION: 'has_many',
    VAT: 19.6
  });
```

.....并重构成类似

```
$scope.settings = {
  username: Profile.username,
  relation: 'has_many',
vat: 19.6
}
```

变为

```
$scope.settings = {
  username: Profile.username,
  relation: HARDCODED.RELATION,
  vat: HARDCODED.VAT
}
```

belindoc.com


```
    BASE_URL: {
      APP: 'http://localhost:3000',
      API: 'http://localhost:3001'
    },
    STORAGE: 'S3',
    ...
  });
```

- **Isolate parts.** Sometimes, there are some things you are not very proud of ... like hardcoded value for example. Instead of let them in your main code, you can create an angular constant

```
angular
  .module('MyApp')
  .constant('HARDCODED', {
    KEY: 'KEY',
    RELATION: 'has_many',
    VAT: 19.6
  });
```

... and refactor something like

```
$scope.settings = {
  username: Profile.username,
  relation: 'has_many',
  vat: 19.6
}
```

to

```
$scope.settings = {
  username: Profile.username,
  relation: HARDCODED.RELATION,
  vat: HARDCODED.VAT
}
```

第12章：使用 ES6 的自定义过滤器

第12.1节：使用ES6的文件大小过滤器

这里有一个文件大小过滤器，用于描述如何向现有模块添加自定义过滤器：

```
let fileSize=function (size,unit,fixedDigit) {
return size.toFixed(fixedDigit) + ' '+unit;
};

  let fileSizeFilter=function () {
        return function (size) {
            if (isNaN(size))
size = 0;

            if (size < 1024)
                return size + ' 字节';

size /= 1024;

            if (size < 1024)
                return fileSize(size,'KB',2);

size /= 1024;

            if (size < 1024)
                return  fileSize(size,'MB',2);

size /= 1024;

            if (size < 1024)
                return  fileSize(size,'GB',2);

size /= 1024;
            return  fileSize(size,'To',2);
        };
    };
export default fileSizeFilter;
```

该过滤器调用模块：

```
import fileSizeFilter from 'path...';
let myMainModule =
angular.module('mainApp', [])
   .filter('fileSize', fileSizeFilter);
```

调用过滤器的HTML代码：

```
<div ng-app="mainApp">

    <div>
        <input type="text" ng-model="size" />
    </div>
    <div>
        <h3>输出:</h3>
        <p>{{size| 文件大小}}</p>
    </div>
</div>
```


Chapter 12: Custom filters with ES6

Section 12.1: FileSize Filter using ES6

We have here a file Size filter to describe how to add costum filter to an existing module :

```
let fileSize=function (size,unit,fixedDigit) {
return size.toFixed(fixedDigit) + ' '+unit;
};

  let fileSizeFilter=function () {
        return function (size) {
            if (isNaN(size))
                size = 0;

            if (size < 1024)
                return size + ' octets';

            size /= 1024;

            if (size < 1024)
               return fileSize(size,'Ko',2);

            size /= 1024;

            if (size < 1024)
               return  fileSize(size,'Mo',2);

            size /= 1024;

            if (size < 1024)
               return  fileSize(size,'Go',2);

            size /= 1024;
            return  fileSize(size,'To',2);
        };
    };
export default fileSizeFilter;
```

The filter call into the module :

```
import fileSizeFilter from 'path...';
let myMainModule =
    angular.module('mainApp', [])
   .filter('fileSize', fileSizeFilter);
```

The html code where we call the filter :

```
<div ng-app="mainApp">

    <div>
        <input type="text" ng-model="size" />
    </div>
    <div>
        <h3>Output:</h3>
        <p>{{size| Filesize}}</p>
    </div>
</div>
```


belindoc.com

第13章：使用 ngModelController的指令

第13.1节：一个简单的控件：评分

让我们构建一个简单的控件，一个评分小部件，旨在用作：

```
<rating min="0" max="5" nullifier="true" ng-model="data.rating"></rating>
```

暂时不使用花哨的CSS；它将呈现为：

```
0 1 2 3 4 5 x
```

点击数字选择该评分；点击“x”将评分设置为null。

```
app.directive('评分', function() {

    function RatingController() {
        this._ngModel = null;
        this.rating = null;
        this.options = null;
        this.min = typeof this.min === 'number' ? this.min : 1;
        this.max = typeof this.max === 'number' ? this.max : 5;
    }

RatingController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            // 关键点 1
ngModel.$render = this._render.bind(this);
        }
    };

RatingController.prototype._render = function() {
        this.rating = this._ngModel.$viewValue != null ? this._ngModel.$viewValue : -
Number.MAX_VALUE;
    };

RatingController.prototype._calculateOptions = function() {
        if( this.min == null || this.max == null ) {
            this.options = [];
        }
        else {
            this.options = new Array(this.max - this.min + 1);
            for( var i=0; i < this.options.length; i++ ) {
                this.options[i] = this.min + i;
            }
        }
    };

RatingController.prototype.setValue = function(val) {
        this.rating = val;
        // KEY POINT 2
        this._ngModel.$setViewValue(val);
    };

    // KEY POINT 3
```


Chapter 13: Directives using ngModelController

Section 13.1: A simple control: rating

Let us build a simple control, a rating widget, intended to be used as:

```
<rating min="0" max="5" nullifier="true" ng-model="data.rating"></rating>
```

No fancy CSS for now; this would render as:

```
0 1 2 3 4 5 x
```

Clicking on a number selects that rating; and clicking the "x" sets the rating to null.

```
app.directive('rating', function() {

    function RatingController() {
        this._ngModel = null;
        this.rating = null;
        this.options = null;
        this.min = typeof this.min === 'number' ? this.min : 1;
        this.max = typeof this.max === 'number' ? this.max : 5;
    }

    RatingController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            // KEY POINT 1
            ngModel.$render = this._render.bind(this);
        }
    };

    RatingController.prototype._render = function() {
        this.rating = this._ngModel.$viewValue != null ? this._ngModel.$viewValue : -
Number.MAX_VALUE;
    };

    RatingController.prototype._calculateOptions = function() {
        if( this.min == null || this.max == null ) {
            this.options = [];
        }
        else {
            this.options = new Array(this.max - this.min + 1);
            for( var i=0; i < this.options.length; i++ ) {
                this.options[i] = this.min + i;
            }
        }
    };

    RatingController.prototype.setValue = function(val) {
        this.rating = val;
        // KEY POINT 2
        this._ngModel.$setViewValue(val);
    };

    // KEY POINT 3
```

```
    Object.defineProperty(RatingController.prototype, 'min', {
        get: function() {
            return this._min;
        },
        set: function(val) {
            this._min = val;
            this._calculateOptions();
        }
    });

    Object.defineProperty(RatingController.prototype, 'max', {
        get: function() {
            return this._max;
        },
        set: function(val) {
            this._max = val;
            this._calculateOptions();
        }
    });

    return {
restrict: 'E',
        scope: {
            // KEY POINT 3
min: '<?',
            max: '<?',
            nullifier: '<?'
        },
bindToController: true,
        controllerAs: 'ctrl',
        controller: RatingController,
        require: ['rating', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
template:
            '<span ng-repeat="o in ctrl.options" href="#" class="rating-option" ng-
class="{\'rating-option-active\': o <= ctrl.rating}" ng-click="ctrl.setValue(o)">{{ o }}</span>' +
            '<span ng-if="ctrl.nullifier" ng-click="ctrl.setValue(null)" class="rating-
nullifier">&#10006;</span>'
    };
});
```

要点：

1. 实现ngModel.$render以将模型的视图值传递到你的视图中。
2. 每当您觉得视图值应该更新时，调用ngModel.$setViewValue()。
3. 当然，控制器可以参数化；如果使用 Angular 版本 >= 1.5，则使用'<'作用域绑定来传递参数明确指示输入——单向绑定。如果每当参数变化时都必须采取行动，可以使用 JavaScript 属性（参见Object.defineProperty()）来减少一些监听。

注1：为了避免实现过于复杂，评级值被插入到一个数组中——`ctrl.options`。这不是必需的；一种更高效但也更复杂的实现方式可以使用DOM操作来在`min/max`变化时插入/移除评分。

注释2：除`'<'`作用域绑定外，此示例可用于Angular 1.5以下版本。如果您使用的是Angular 1.5及以上版本，建议将其转换为组件，并使用`$onInit()`生命周期钩子来初始化`min`和`max`，而不是在控制器的构造函数中进行初始化。


```
    Object.defineProperty(RatingController.prototype, 'min', {
        get: function() {
            return this._min;
        },
        set: function(val) {
            this._min = val;
            this._calculateOptions();
        }
    });

    Object.defineProperty(RatingController.prototype, 'max', {
        get: function() {
            return this._max;
        },
        set: function(val) {
            this._max = val;
            this._calculateOptions();
        }
    });

    return {
        restrict: 'E',
        scope: {
            // KEY POINT 3
            min: '<?',
            max: '<?',
            nullifier: '<?'
        },
        bindToController: true,
        controllerAs: 'ctrl',
        controller: RatingController,
        require: ['rating', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
        template:
            '<span ng-repeat="o in ctrl.options" href="#" class="rating-option" ng-
class="{\'rating-option-active\': o <= ctrl.rating}" ng-click="ctrl.setValue(o)">{{ o }}</span>' +
            '<span ng-if="ctrl.nullifier" ng-click="ctrl.setValue(null)" class="rating-
nullifier">&#10006;</span>'
    };
});
```

Key points:

1. Implement `ngModel.$render` to transfer the model's *view value* to your view.
2. Call `ngModel.$setViewValue()` whenever you feel the view value should be updated.
3. The control can of course be parameterized; use `'<'` scope bindings for parameters, if in Angular >= 1.5 to clearly indicate input - one way binding. If you have to take action whenever a parameter changes, you can use a JavaScript property (see `Object.defineProperty()`) to save a few watches.

Note 1: In order not to overcomplicate the implementation, the rating values are inserted in an array - the `ctrl.options`. This is not needed; a more efficient, but also more complex, implementation could use DOM manipulation to insert/remove ratings when `min/max` change.

Note 2: With the exception of the `'<'` scope bindings, this example can be used in Angular < 1.5. If you are on Angular >= 1.5, it would be a good idea to transform this to a component and use the `$onInit()` lifecycle hook to initialize `min` and `max`, instead of doing so in the controller's constructor.

还有一个必要的示例：https://jsfiddle.net/h81mgxma/

第13.2节：几个复杂的控件：编辑完整对象

自定义控件不必局限于像基本类型这样简单的内容；它可以编辑更有趣的内容。这里我们展示了两种自定义控件，一种用于编辑人员信息，另一种用于编辑地址。地址控件用于编辑人员的地址。一个使用示例如下：

```
<input-person ng-model="data.thePerson"></input-person>
<input-address ng-model="data.thePerson.address"></input-address>
```

此示例的模型故意设计得很简单：

```
function Person(data) {
data = data || {};
  this.name = data.name;
  this.address = data.address ? new Address(data.address) : null;
}

function Address(data) {
  data = data || {};
  this.street = data.street;
  this.number = data.number;
}
```

地址编辑器：

```
app.directive('inputAddress', function() {

    InputAddressController.$inject = ['$scope'];
    function InputAddressController($scope) {
        this.$scope = $scope;
        this._ngModel = null;
        this.value = null;
        this._unwatch = angular.noop;
    }

InputAddressController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            // KEY POINT 3
ngModel.$render = this._render.bind(this);
        }
    };

InputAddressController.prototype._makeWatch = function() {
        // 关键点 1
        this._unwatch = this.$scope.$watchCollection(
            (function() {
                return this.value;
            }).bind(this),
            (function(newval, oldval) {
                if( newval !== oldval ) { // 跳过初始触发
                    this._ngModel.$setViewValue(newval !== null ? new Address(newval) : null);
                }
            }).bind(this)
        );
    };
```


And a necessary fiddle: https://jsfiddle.net/h81mgxma/

Section 13.2: A couple of complex controls: edit a full object

A custom control does not have to limit itself to trivial things like primitives; it can edit more interesting things. Here we present two types of custom controls, one for editing persons and one for editing addresses. The address control is used to edit the person's address. An example of usage would be:

```
<input-person ng-model="data.thePerson"></input-person>
<input-address ng-model="data.thePerson.address"></input-address>
```

The model for this example is deliberately simplistic:

```
function Person(data) {
  data = data || {};
  this.name = data.name;
  this.address = data.address ? new Address(data.address) : null;
}

function Address(data) {
  data = data || {};
  this.street = data.street;
  this.number = data.number;
}
```

The address editor:

```
app.directive('inputAddress', function() {

    InputAddressController.$inject = ['$scope'];
    function InputAddressController($scope) {
        this.$scope = $scope;
        this._ngModel = null;
        this.value = null;
        this._unwatch = angular.noop;
    }

    InputAddressController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            // KEY POINT 3
            ngModel.$render = this._render.bind(this);
        }
    };

    InputAddressController.prototype._makeWatch = function() {
        // KEY POINT 1
        this._unwatch = this.$scope.$watchCollection(
            (function() {
                return this.value;
            }).bind(this),
            (function(newval, oldval) {
                if( newval !== oldval ) { // skip the initial trigger
                    this._ngModel.$setViewValue(newval !== null ? new Address(newval) : null);
                }
            }).bind(this)
        );
    };
```

```
InputAddressController.prototype._render = function() {
        // 关键点 2
        this._unwatch();
        this.value = this._ngModel.$viewValue ? new Address(this._ngModel.$viewValue) : null;
        this._makeWatch();
    };

    return {
restrict: 'E',
scope: {},
bindToController: true,
        controllerAs: 'ctrl',
        controller: InputAddressController,
        require: ['inputAddress', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
template:
            '<div>' +
                '<label><span>街道：</span><input type="text" ng-model="ctrl.value.street"
/></label>' +
                '<label><span>门牌号：</span><input type="text" ng-model="ctrl.value.number"
/></label>' +
            '</div>'
    };
});
```

要点：

1. 我们正在编辑一个对象；我们不想直接更改父组件传给我们的对象（我们希望我们的模型符合不可变性原则）。因此，我们对正在编辑的对象创建一个浅层监听，并在属性发生变化时使用$setViewValue()更新模型。我们将一个copy传递给父组件。

2. 每当模型从外部发生变化时，我们会复制它并将副本保存到我们的作用域中。不可变性原则再次体现，尽管内部副本不是不可变的，但外部副本很可能是不可变的。此外，我们会重建监听（this_unwatch();this._makeWatch();），以避免触发由模型推送给我们的更改的监听器。（我们只希望监听器对UI中所做的更改触发。）

3. 除了上述几点，我们实现了ngModel.$render()，并调用了ngModel.$setViewValue()，就像处理简单控件一样（参见评分示例）。

人员自定义控件的代码几乎相同。模板使用了<input-address>。在更高级的实现中，我们可以将控制器提取到可重用的模块中。

```
app.directive('inputPerson', function() {

    InputPersonController.$inject = ['$scope'];
    function InputPersonController($scope) {
        this.$scope = $scope;
        this._ngModel = null;
        this.value = null;
        this._unwatch = angular.noop;
    }

InputPersonController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
ngModel.$render = this._render.bind(this);
```



```
    InputAddressController.prototype._render = function() {
        // KEY POINT 2
        this._unwatch();
        this.value = this._ngModel.$viewValue ? new Address(this._ngModel.$viewValue) : null;
        this._makeWatch();
    };

    return {
        restrict: 'E',
        scope: {},
        bindToController: true,
        controllerAs: 'ctrl',
        controller: InputAddressController,
        require: ['inputAddress', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
        template:
            '<div>' +
                '<label><span>Street:</span><input type="text" ng-model="ctrl.value.street"
/></label>' +
                '<label><span>Number:</span><input type="text" ng-model="ctrl.value.number"
/></label>' +
            '</div>'
    };
});
```

Key points:

1. We are editing an object; we do not want to change directly the object given to us from our parent (we want our model to be compatible with the immutability principle). So we create a shallow watch on the object being edited and update the model with `$setViewValue()` whenever a property changes. We pass a *copy* to our parent.
2. Whenever the model changes from the outside, we copy it and save the copy to our scope. Immutability principles again, though the internal copy is not immutable, the external could very well be. Additionally we rebuild the watch (`this._unwatch();this._makeWatch();`), to avoid triggering the watcher for changes pushed to us by the model. (We only want the watch to trigger for changes made in the UI.)
3. Other that the points above, we implement `ngModel.$render()` and call `ngModel.$setViewValue()` as we would for a simple control (see the rating example).

The code for the person custom control is almost identical. The template is using the **`<input-address>`**. In a more advanced implementation we could extract the controllers in a reusable module.

```
app.directive('inputPerson', function() {

    InputPersonController.$inject = ['$scope'];
    function InputPersonController($scope) {
        this.$scope = $scope;
        this._ngModel = null;
        this.value = null;
        this._unwatch = angular.noop;
    }

    InputPersonController.prototype.setNgModel = function(ngModel) {
        this._ngModel = ngModel;

        if( ngModel ) {
            ngModel.$render = this._render.bind(this);
```

```
        }
    };

InputPersonController.prototype._makeWatch = function() {
        this._unwatch = this.$scope.$watchCollection(
            (function() {
                return this.value;
            }).bind(this),
            (function(newval, oldval) {
                if( newval !== oldval ) { // 跳过初始触发
                    this._ngModel.$setViewValue(newval !== null ? new Person(newval) : null);
                }
            }).bind(this)
        );
    };

InputPersonController.prototype._render = function() {
        this._unwatch();
        this.value = this._ngModel.$viewValue ? new Person(this._ngModel.$viewValue) : null;
        this._makeWatch();
    };

    return {
restrict: 'E',
scope: {},
bindToController: true,
        controllerAs: 'ctrl',
        controller: InputPersonController,
        require: ['inputPerson', 'ngModel'],
        link: function(scope, elem, attrs, ctrls) {
            ctrls[0].setNgModel(ctrls[1]);
        },
template:
            '<div>' +
                '<label><span>姓名：</span><input type="text" ng-model="ctrl.value.name" /></label>'
+
                '<input-address ng-model="ctrl.value.address"></input-address>' +
            '</div>'
    };
});
```

注意：这里的对象是有类型的，即它们有合适的构造函数。这不是必须的；模型可以是普通的JSON对象。在这种情况下，只需使用`angular.copy()`代替构造函数。一个额外的好处是控制器对于这两个控件是相同的，并且可以很容易地提取到某个公共模块中。

小提琴：https://jsfiddle.net/3tzyqfko/2/

提取了控制器公共代码的两个 fiddle 版本：https://jsfiddle.net/agj4cp0e/ 和 https://jsfiddle.net/ugb6Lw8b/


```
    }
};

InputPersonController.prototype._makeWatch = function() {
    this._unwatch = this.$scope.$watchCollection(
        (function() {
            return this.value;
        }).bind(this),
        (function(newval, oldval) {
            if( newval !== oldval ) { // skip the initial trigger
                this._ngModel.$setViewValue(newval !== null ? new Person(newval) : null);
            }
        }).bind(this)
    );
};

InputPersonController.prototype._render = function() {
    this._unwatch();
    this.value = this._ngModel.$viewValue ? new Person(this._ngModel.$viewValue) : null;
    this._makeWatch();
};

return {
    restrict: 'E',
    scope: {},
    bindToController: true,
    controllerAs: 'ctrl',
    controller: InputPersonController,
    require: ['inputPerson', 'ngModel'],
    link: function(scope, elem, attrs, ctrls) {
        ctrls[0].setNgModel(ctrls[1]);
    },
    template:
        '<div>' +
            '<label><span>Name:</span><input type="text" ng-model="ctrl.value.name" /></label>'
+
            '<input-address ng-model="ctrl.value.address"></input-address>' +
        '</div>'
};
});
```

Note: Here the objects are typed, i.e. they have proper constructors. This is not obligatory; the model can be plain JSON objects. In this case just use `angular.copy()` instead of the constructors. An added advantage is that the controller becomes identical for the two controls and can easily be extracted into some common module.

The fiddle: https://jsfiddle.net/3tzyqfko/2/

Two versions of the fiddle having extracted the common code of the controllers: https://jsfiddle.net/agj4cp0e/ and https://jsfiddle.net/ugb6Lw8b/

第14章：控制器

第14.1节：你的第一个控制器

控制器是 Angular 中用于保存作用域并处理页面内某些操作的基本结构。每个控制器都与一个 HTML 视图绑定。

下面是一个 Angular 应用的基本模板：

```
<!DOCTYPE html>

<html lang="en" ng-app='MyFirstApp'>
    <head>
        <title>我的第一个应用</title>

        <!-- angular 源代码 -->
        <script src="https://code.angularjs.org/1.5.3/angular.min.js"></script>

        <!-- 你的自定义控制器代码 -->
        <script src="js/controllers.js"></script>
    </head>
    <body>
        <div ng-controller="MyController as mc">
            <h1>{{ mc.title }}</h1>
            <p>{{ mc.description }}</p>
            <button ng-click="mc.clicked()">
              点击我！
            </button>
        </div>
    </body>
</html>
```

这里有几点需要注意：

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

使用ng-app设置应用名称，可以让你在外部Javascript文件中访问该应用，下面会介绍。

```
<script src="js/controllers.js"></script>
```

我们需要一个Javascript文件，在那里定义控制器及其操作/数据。

```
<div ng-controller="MyController as mc">
```

ng 控制器属性为该 DOM 元素及其所有子元素（递归）设置控制器。

你可以通过说 ... as mc 来拥有多个相同的控制器（在本例中为 MyController），我们给这个控制器实例起了一个别名。

```
<h1>{{ mc.title }}</h1>
```

{{ ... }} 表示法是 Angular 表达式。在本例中，这将把 **<h1>** 元素的内部文本设置为 mc.title 的值。

Chapter 14: Controllers

Section 14.1: Your First Controller

A controller is a basic structure used in Angular to preserve scope and handle certain actions within a page. Each controller is coupled with an HTML view.

Below is a basic boilerplate for an Angular app:

```
<!DOCTYPE html>

<html lang="en" ng-app='MyFirstApp'>
    <head>
        <title>My First App</title>

        <!-- angular source -->
        <script src="https://code.angularjs.org/1.5.3/angular.min.js"></script>

        <!-- Your custom controller code -->
        <script src="js/controllers.js"></script>
    </head>
    <body>
        <div ng-controller="MyController as mc">
            <h1>{{ mc.title }}</h1>
            <p>{{ mc.description }}</p>
            <button ng-click="mc.clicked()">
                Click Me!
            </button>
        </div>
    </body>
</html>
```

There are a few things to note here:

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

Setting the app name with `ng-app` lets you access the application in an external Javascript file, which will be covered below.

```
<script src="js/controllers.js"></script>
```

We'll need a Javascript file where you define your controllers and their actions/data.

```
<div ng-controller="MyController as mc">
```

The `ng-controller` attribute sets the controller for that DOM element and all elements that are children (recursively) below it.

You can have multiple of the same controller (in this case, `MyController`) by saying `... as mc`, we're giving this instance of the controller an alias.

```
<h1>{{ mc.title }}</h1>
```

The {{ ... }} notation is an Angular expression. In this case, this will set the inner text of that **<h1>** element to whatever the value of `mc.title` is.

注意： Angular 采用双向数据绑定，这意味着无论你如何更新 `mc.title` 的值，它都会反映在控制器和页面中。

还要注意，Angular 表达式 not 必须引用控制器。Angular 表达式可以简单到 `{{ 1 + 2 }}` 或 `{{ "Hello " + "World" }}`。

```
<button ng-click="mc.clicked()">
```

ng-click 是 Angular 指令，在本例中绑定按钮的点击事件以触发 MyController 实例的 clicked() 函数。

考虑到这些，让我们编写 `MyController` 控制器的实现。根据上面的示例，你需要在 `js/controller.js` 中编写这段代码。

首先，你需要在 Javascript 中实例化 Angular 应用。

```
var app = angular.module("MyFirstApp", []);
```

请注意，我们在这里传递的名称与您在HTML中使用ng-app指令设置的名称相同。

现在我们有了app对象，可以用它来创建控制器。

```
app.controller('MyController', function(){
    var ctrl = this;

ctrl.title = "我的第一个Angular应用";
    ctrl.description = "这是我的第一个Angular应用！";

   ctrl.clicked = function(){
alert("MyController.clicked()");
    };
});
```

注意： 对于我们希望成为控制器实例一部分的任何内容，都使用this关键字。

这就是构建一个简单控制器所需的全部内容。

第14.2节：创建控制器，防止压缩破坏

有几种不同的方法可以保护您的控制器创建不被压缩破坏。

第一种称为内联数组注释。它看起来如下：

```
var app = angular.module('app');
app.controller('sampleController', ['$scope', '$http', function(a, b){
    //逻辑代码
}]);
```

controller 方法的第二个参数可以接受一个依赖数组。正如你所见，我定义了 `$scope` 和 `$http`，它们应该对应于控制器函数的参数，其中 `a` 将是 `$scope`，`b` 将是 `$http`。请注意，数组中的最后一项应该是你的控制器函数。

第二种方式是使用 $inject 属性。它的形式如下：

```
var app = angular.module('app');
app.controller('sampleController', sampleController);
```


Note: Angular employs dual-way data binding, meaning that regardless of how you update the `mc.title` value, it will be reflected in both the controller and the page.

Also note that Angular expressions do *not* have to reference a controller. An Angular expression can be as simple as `{{ 1 + 2 }}` or `{{ "Hello " + "World" }}`.

```
<button ng-click="mc.clicked()">
```

`ng-click` is an Angular directive, in this case binding the click event for the button to trigger the `clicked()` function of the `MyController` instance.

With those things in mind, let's write an implementation of the `MyController` controller. With the example above, you would write this code in `js/controller.js`.

First, you'll need to instantiate the Angular app in your Javascript.

```
var app = angular.module("MyFirstApp", []);
```

Note that the name we pass here is the same as the name you set in your HTML with the `ng-app` directive.

Now that we have the app object, we can use that to create controllers.

```
app.controller('MyController', function(){
    var ctrl = this;

    ctrl.title = "My First Angular App";
    ctrl.description = "This is my first Angular app!";

    ctrl.clicked = function(){
        alert("MyController.clicked()");
    };
});
```

Note: For anything that we want to be a part of the controller instance, we use the **`this`** keyword.

This is all that is required to build a simple controller.

Section 14.2: Creating Controllers, Minification safe

There are a couple different ways to protect your controller creation from minification.

The first is called inline array annotation. It looks like the following:

```
var app = angular.module('app');
app.controller('sampleController', ['$scope', '$http', function(a, b){
    //logic here
}]);
```

The second parameter of the controller method can accept an array of dependencies. As you can see I've defined `$scope` and `$http` which should correspond to the parameters of the controller function in which `a` will be the `$scope`, and `b` would be `$http`. Take note that the last item in the array should be your controller function.

The second option is using the `$inject` property. It looks like the following:

```
var app = angular.module('app');
app.controller('sampleController', sampleController);
```

```
sampleController.$inject = ['$scope', '$http'];
function sampleController(a, b) {
    //逻辑代码
}
```

这与内联数组注释的作用相同，但为偏好不同风格的人提供了另一种选择。

注入依赖的顺序很重要

在使用数组形式注入依赖时，务必确保依赖列表与传递给控制器函数的参数列表相对应。

注意在以下示例中，`$scope` 和 `$http` 的顺序被颠倒了。这会导致代码出现问题。

```
// 故意的错误：注入的依赖顺序颠倒，会导致问题
app.controller('sampleController', ['$scope', '$http',function($http, $scope) {
    $http.get('sample.json');
}]);
```

第14.3节：在Angular JS中使用ControllerAs

在Angular中，$scope 是控制器和视图之间的纽带，帮助实现所有的数据绑定需求。
Controller As是绑定控制器和视图的另一种方式，通常推荐使用。基本上，Angular中有这两种控制器结构（即$scope和Controller As）。

使用Controller As的不同方式有 -

controllerAs 视图语法

```
<div ng-controller="CustomerController as customer">
   {{ customer.name }}
</div>
```

controllerAs 控制器语法

```
function CustomerController() {
    this.name = {};
    this.sendMessage = function() { };
}
```

controllerAs 与 vm

```
function CustomerController() {
    /*jshint validthis: true */
    var vm = this;
vm.name = {};
vm.sendMessage = function() { };
}
```

controllerAs 是 $scope 的语法糖。你仍然可以绑定到视图并访问 $scope 方法。使用controllerAs 是 Angular 核心团队推荐的最佳实践之一。原因有很多，以下是其中几个 -

- $scope 是通过一个中介对象将控制器的成员暴露给视图。通过设置 this.*，我们可以只暴露我们想从控制器到视图暴露的内容。它也遵循


```
sampleController.$inject = ['$scope', '$http'];
function sampleController(a, b) {
    //logic here
}
```

This does the same thing as inline array annotation but provides a different styling for those that prefer one option over the other.

The order of injected dependencies is important

When injecting dependencies using the array form, be sure that the list of the dependencies match its corresponding list of arguments passed to the controller function.

Note that in the following example, `$scope` and `$http` are reversed. This will cause a problem in the code.

```
// Intentional Bug: injected dependencies are reversed which will cause a problem
app.controller('sampleController', ['$scope', '$http',function($http, $scope) {
    $http.get('sample.json');
}]);
```

Section 14.3: Using ControllerAs in Angular JS

In Angular `$scope` is the glue between the Controller and the View that helps with all of our data binding needs. Controller As is another way of binding controller and view and is mostly recommended to use. Basically these are the two controller constructs in Angular (i.e $scope and Controller As).

Different ways of using Controller As are -

controllerAs View Syntax

```
<div ng-controller="CustomerController as customer">
    {{ customer.name }}
</div>
```

controllerAs Controller Syntax

```
function CustomerController() {
    this.name = {};
    this.sendMessage = function() { };
}
```

controllerAs with vm

```
function CustomerController() {
    /*jshint validthis: true */
    var vm = this;
    vm.name = {};
    vm.sendMessage = function() { };
}
```

`controllerAs` is syntactic sugar over `$scope`. You can still bind to the View and still access `$scope` methods. Using `controllerAs`, is one of the best practices suggested by the angular core team. There are many reason for this, few of them are -

- `$scope` is exposing the members from the controller to the view via an intermediary object. By setting `this.*`, we can expose just what we want to expose from the controller to the view. It also follow the

标准的 JavaScript 使用 this 的方式。

- 使用 controllerAs 语法，我们的代码更具可读性，且可以通过父控制器的别名访问父属性，而不必使用 $parent 语法。
- 它促进在视图中绑定到“点状”对象（例如 customer.name 而不是 name），这样更具上下文，更易读，并且避免了没有“点状”时可能出现的任何引用问题。
- 有助于避免在具有嵌套控制器的视图中使用$parent调用。
- 使用controllerAs语法时，为此使用一个捕获变量。选择一个一致的变量名，例如vm，代表视图模型（ViewModel）。因为this关键字是有上下文的，当在控制器内部的函数中使用时，可能会改变其上下文。捕获this的上下文可以避免遇到此问题。

注意：使用controllerAs语法会将当前控制器的引用添加到当前作用域，因此它作为字段可用。

```
<div ng-controller="Controller as vm>...</div>
```

vm作为$scope.vm可用。

第14.4节：创建支持压缩安全的Angular控制器

要创建支持压缩安全的Angular控制器，需要更改controller函数的参数。

在module.controller函数中的第二个参数应传入一个数组，其中最后一个参数是控制器函数，之前的每个参数都是每个注入值的名称。

这与正常的范式不同；正常范式是直接传入带有注入参数的控制器函数。

已知：

```
var app = angular.module('myApp');
```

控制器应如下所示：

```
app.controller('ctrlInject',
    [
        /* 注入的参数 */
        '$Injectable1',
        '$Injectable2',
        /* 控制器函数 */
        function($injectable1Instance, $injectable2Instance) {
            /* 控制器内容 */
        }
    ]
);
```

注意：注入参数的名称不必匹配，但它们将按顺序绑定。

这将被压缩成类似如下内容：

```
var
a=angular.module('myApp');a.controller('ctrlInject',['$Injectable1','$Injectable2',function(b,c){/*
控制器内容 */}]);
```

压缩过程会将每个app实例替换为a，每个$Injectable1Instance实例替换为b，
每个$Injectable2Instance实例替换为c。

standard JavaScript way of using this.

- using `controllerAs` syntax, we have more readable code and the parent property can be accessed using the alias name of the parent controller instead of using the `$parent` syntax.
- It promotes the use of binding to a "dotted" object in the View (e.g. customer.name instead of name), which is more contextual, easier to read, and avoids any reference issues that may occur without "dotting".
- Helps avoid using `$parent` calls in Views with nested controllers.
- Use a capture variable for this when using the `controllerAs` syntax. Choose a consistent variable name such as `vm`, which stands for ViewModel. Because, **`this`** keyword is contextual and when used within a function inside a controller may change its context. Capturing the context of this avoids encountering this problem.

NOTE: using `controllerAs` syntax add to current scope reference to current controller, so it available as field

```
<div ng-controller="Controller as vm>...</div>
```

`vm` is available as `$scope.vm`.

Section 14.4: Creating Minification-Safe Angular Controllers

To create minification-safe angular controllers, you will change the `controller` function parameters.

The second argument in the `module.controller` function should be passed an **array**, where the **last parameter** is the **controller function**, and every parameter before that is the **name** of each injected value.

This is different from the normal paradigm; that takes the **controller function** with the injected arguments.

Given:

```
var app = angular.module('myApp');
```

The controller should look like this:

```
app.controller('ctrlInject',
    [
        /* Injected Parameters */
        '$Injectable1',
        '$Injectable2',
        /* Controller Function */
        function($injectable1Instance, $injectable2Instance) {
            /* Controller Content */
        }
    ]
);
```

Note: *The names of injected parameters are not required to match, but they will be bound in order.*

This will minify to something similar to this:

```
var
a=angular.module('myApp');a.controller('ctrlInject',['$Injectable1','$Injectable2',function(b,c){/*
Controller Content */}]);
```

The minification process will replace every instance of `app` with `a`, every instance of `$Injectable1Instance` with `b`, and every instance of `$Injectable2Instance` with `c`.

第14.5节：创建控制器

```
angular
.module('app')
.controller('SampleController', SampleController)

SampleController.$inject = ['$log', '$scope'];
function SampleController($log, $scope){
    $log.debug('*****SampleController******');

    /* 你的代码写在这里 */
}
```

注意：.$inject确保你的依赖在压缩后不会被打乱。同时，确保它的顺序与命名函数一致。

第14.6节：嵌套控制器

嵌套控制器也会链接$scope。更改嵌套控制器中的$scope变量会更改父控制器中的相同$scope变量。

```
.controller('parentController', function ($scope) {
    $scope.parentVariable = "我是父控制器";
});

.controller('childController', function ($scope) {
    $scope.childVariable = "我是子控制器";

$scope.childFunction = function () {
        $scope.parentVariable = "我正在覆盖你";
    };
});
```

现在让我们尝试同时处理它们，嵌套使用。

```
<body ng-controller="parentController">
    我是哪个控制器？ {{parentVariable}}
    <div ng-controller="childController">
        我是哪个控制器？ {{childVariable}}
        <button ng-click="childFunction()"> 点击我来覆盖！ </button>
    </div>
</body>
```

嵌套控制器可能有其好处，但在这样做时必须牢记一件事。调用 `ngController` 指令会创建控制器的新实例——这通常会导致混淆和意想不到的结果。

Section 14.5: Creating Controllers

```
angular
    .module('app')
    .controller('SampleController', SampleController)

SampleController.$inject = ['$log', '$scope'];
function SampleController($log, $scope){
    $log.debug('*****SampleController******');

    /* Your code below */
}
```

Note: The `.$inject` will make sure your dependencies doesn't get scrambled after minification. Also, make sure it's in order with the named function.

Section 14.6: Nested Controllers

Nesting controllers chains the `$scope` as well. Changing a `$scope` variable in the nested controller changes the same `$scope` variable in the parent controller.

```
.controller('parentController', function ($scope) {
    $scope.parentVariable = "I'm the parent";
});

.controller('childController', function ($scope) {
    $scope.childVariable = "I'm the child";

    $scope.childFunction = function () {
        $scope.parentVariable = "I'm overriding you";
    };
});
```

Now let's try to handle both of them, nested.

```
<body ng-controller="parentController">
    What controller am I? {{parentVariable}}
    <div ng-controller="childController">
        What controller am I? {{childVariable}}
        <button ng-click="childFunction()"> Click me to override! </button>
    </div>
</body>
```

Nesting controllers may have it's benefits, but one thing must be kept in mind when doing so. Calling the `ngController` directive creates a new instance of the controller - which can often create confusion and unexpected results.

第15章：使用ES6的控制器

第15.1节：控制器

如果你熟悉面向对象编程，使用ES6编写AngularJS控制器非常简单：

```
 class exampleContoller{

constructor(service1,service2,...serviceN){
            let ctrl=this;
ctrl.service1=service1;
                ctrl.service2=service2;
              .
                .
                .
                ctrl.service1=service1;
ctrl.controllerName = '示例控制器';
                ctrl.method1(controllerName)

        }

method1(param){
            let ctrl=this;
ctrl.service1.serviceFunction();
              .
                .
                ctrl.scopeName=param;
        }
        .
        .
        .
        .
        methodN(param){
            let ctrl=this;
ctrl.service1.serviceFunction();
              .
                .
        }


    }
exampleContoller.$inject = ['service1','service2',...,'serviceN'];
    export default exampleContoller;
```

belindoc.com

Chapter 15: Controllers with ES6

Section 15.1: Controller

it is very easy to write an angularJS controller with ES6 if your are familiarized with the **Object Oriented Programming** :

```
class exampleContoller{

       constructor(service1,service2,...serviceN){
          let ctrl=this;
              ctrl.service1=service1;
              ctrl.service2=service2;
               .
               .
               .
              ctrl.service1=service1;
              ctrl.controllerName = 'Example Controller';
              ctrl.method1(controllerName)

       }

       method1(param){
          let ctrl=this;
               ctrl.service1.serviceFunction();
              .
              .
              ctrl.scopeName=param;
       }
       .
       .
       .
       .
       methodN(param){
          let ctrl=this;
               ctrl.service1.serviceFunction();
              .
              .
       }


    }
    exampleContoller.$inject = ['service1','service2',...,'serviceN'];
    export default exampleContoller;
```

第16章：控制器中的self或this变量

这是对AngularJS代码中常见模式的解释，通常被认为是最佳实践。

第16.1节：理解self变量的目的

使用“controller as语法”时，在使用ng-controller指令的html中会给控制器起一个别名。

```
<div ng-controller="MainCtrl as main">
</div>
```

然后你可以访问表示我们控制器实例的main变量的属性和方法。例如，访问我们控制器的greeting属性并将其显示在屏幕上：

```
<div ng-controller="MainCtrl as main">
    {{ main.greeting }}
</div>
```

现在，在我们的控制器中，我们需要为控制器实例的greeting属性设置一个值（而不是$scope或其他东西）：

```
angular
.module('ngNjOrg')
.controller('ForgotPasswordController',function ($log) {
  var self = this;

self.greeting = "Hello World";
})
```

为了让HTML正确显示，我们需要在控制器主体内的this上设置greeting属性。我创建了一个名为self的中间变量来保存对this的引用。为什么？请考虑以下代码：

```
angular
.module('ngNjOrg')
.controller('ForgotPasswordController',function ($log) {
  var self = this;

self.greeting = "Hello World";

  function itsLate () {
    this.greeting = "Goodnight";
  }

})
```

在上述代码中，你可能期望当调用方法itsLate时屏幕上的文本会更新，但实际上并没有。JavaScript 使用函数级作用域规则，因此其内部的 "this" 指向与方法体外部的 "this" 不同。然而，如果我们使用self变量，就能得到预期的结果：

```
 angular
.module('ngNjOrg')
```


Chapter 16: The Self Or This Variable In A Controller

This is an explanation of a common pattern and generally considered best practice that you may see in AngularJS code.

Section 16.1: Understanding The Purpose Of The Self Variable

When using "controller as syntax" you would give your controller an alias in the html when using the ng-controller directive.

```
<div ng-controller="MainCtrl as main">
</div>
```

You can then access properties and methods from the ***main*** variable that represents our controller instance. For example, let's access the ***greeting*** property of our controller and display it on the screen:

```
<div ng-controller="MainCtrl as main">
    {{ main.greeting }}
</div>
```

Now, in our controller, we need to set a value to the greeting property of our controller instance (as opposed to $scope or something else):

```
angular
.module('ngNjOrg')
.controller('ForgotPasswordController',function ($log) {
  var self = this;

  self.greeting = "Hello World";
})
```

In order to have the HTML display correctly we needed to set the greeting property on ***this*** inside of our controller body. I am creating an intermediate variable named ***self*** that holds a reference to this. Why? Consider this code:

```
angular
.module('ngNjOrg')
.controller('ForgotPasswordController',function ($log) {
  var self = this;

  self.greeting = "Hello World";

  function itsLate () {
    this.greeting = "Goodnight";
  }

})
```

In this above code you may expect the text on the screen to update when the method *itsLate* is called, but in fact it does not. JavaScript uses function level scoping rules so the "this" inside of itsLate refers to something different that "this" outside of the method body. However, we can get the desired result if we use the ***self*** variable:

```
 angular
.module('ngNjOrg')
```

```
.controller('ForgotPasswordController',function ($log) {
  var self = this;

self.greeting = "Hello World";

  function itsLate () {
    self.greeting = "晚安";
  }

})
```

这就是在控制器中使用“self”变量的好处——你可以在控制器的任何地方访问它，并且总能确定它引用的是你的控制器实例。

belindoc.com


```
.controller('ForgotPasswordController',function ($log) {
  var self = this;

  self.greeting = "Hello World";

  function itsLate () {
    self.greeting = "Goodnight";
  }

})
```

This is the beauty of using a "self" variable in your controllers- you can access this anywhere in your controller and can always be sure that it is referencing your controller instance.

第17章：服务

第17.1节：使用angular.factory创建服务

首先定义服务（在本例中使用工厂模式）：

```
.factory('dataService', function() {
    var dataObject = {};
    var service = {
        // 定义 getter 方法
        get data() {
            return dataObject;
        },
        // 定义 setter 方法
        set data(value) {
dataObject = value || {};
        }
    };
    // 返回"service"对象以暴露 getter/setter 方法
    return service;
})
```

现在你可以使用该服务在控制器之间共享数据：

```
.controller('controllerOne', function(dataService) {
    // 创建一个对 dataService 的本地引用
    this.dataService = dataService;
    // 创建一个用于存储的对象
    var someObject = {
        name: '某对象',
        value: 1
    };
    // 存储对象
    this.dataService.data = someObject;
})

.controller('controllerTwo', function(dataService) {
    // 创建一个对 dataService 的本地引用
    this.dataService = dataService;
    // 这将自动更新共享数据对象的任何更改
    this.objectFromControllerOne = this.dataService.data;
})
```

第17.2节：服务（Service）与工厂（Factory）之间的区别

1) 服务（Services）

服务是一个构造函数，在运行时只调用一次，使用new，就像我们在普通JavaScript中所做的那样，唯一的区别是AngularJS在幕后调用了new。

关于服务，有一个经验法则需要记住

1. 服务是用new调用的构造函数

让我们看一个简单的例子，我们将注册一个使用$http服务来获取学生详情的服务，并在控制器中使用它

Chapter 17: Services

Section 17.1: Creating a service using angular.factory

First define the service (in this case it uses the factory pattern):

```
.factory('dataService', function() {
    var dataObject = {};
    var service = {
        // define the getter method
        get data() {
            return dataObject;
        },
        // define the setter method
        set data(value) {
            dataObject = value || {};
        }
    };
    // return the "service" object to expose the getter/setter
    return service;
})
```

Now you can use the service to share data between controllers:

```
.controller('controllerOne', function(dataService) {
    // create a local reference to the dataService
    this.dataService = dataService;
    // create an object to store
    var someObject = {
        name: 'SomeObject',
        value: 1
    };
    // store the object
    this.dataService.data = someObject;
})

.controller('controllerTwo', function(dataService) {
    // create a local reference to the dataService
    this.dataService = dataService;
    // this will automatically update with any changes to the shared data object
    this.objectFromControllerOne = this.dataService.data;
})
```

Section 17.2: Difference between Service and Factory

1) Services

A service is a `constructor` function that is invoked once at runtime with **`new`**, just like what we would do with plain JavaScript with only difference that `AngularJS` is calling the **`new`** behind the scenes.

There is one thumb rule to remember in case of services

1. Services are constructors which are called with **`new`**

Lets see a simple example where we would register a service which uses `$http` service to fetch student details, and use it in the controller

```
function StudentDetailsService($http) {
  this.getStudentDetails = function getStudentDetails() {
    return $http.get('/details');
  };
}

angular.module('myapp').service('StudentDetailsService', StudentDetailsService);
```

我们只需将此服务注入到控制器中

```
function StudentController(StudentDetailsService) {
  StudentDetailsService.getStudentDetails().then(function (response) {
      // 处理响应
    });
}
angular.module('app').controller('StudentController', StudentController);
```

何时使用？

在任何想使用构造函数的地方使用.service()。它通常用于创建公共API，就像`getStudentDetails()`一样。但如果你不想使用构造函数，而希望使用简单的API模式，那么.service()的灵活性就不大了。

2) 工厂（Factory）

尽管我们可以使用.factory()实现所有通过.service()实现的功能，但这并不意味着.factory()“等同于”.service()。它比.service()更强大、更灵活。

工厂()是一种用于返回值的设计模式。

在工厂模式中有两个经验法则需要记住

1. 工厂返回值
2. 工厂（可以）创建对象（任何对象）

让我们看一些使用`.factory()`可以做的示例

返回对象字面量

让我们看一个示例，工厂使用基本的揭示模块模式返回一个对象

```
function 学生详情服务($http) {
  function 获取学生详情() {
    return $http.get('/details');
  }
  return {
获取学生详情: 获取学生详情
  };
}

angular.module('我的应用').factory('学生详情服务', 学生详情服务);
```

在控制器中的使用

```
function StudentController(StudentDetailsService) {
  StudentDetailsService.getStudentDetails().then(function (response) {
      // 处理响应
    });
```



```
function StudentDetailsService($http) {
  this.getStudentDetails = function getStudentDetails() {
    return $http.get('/details');
  };
}

angular.module('myapp').service('StudentDetailsService', StudentDetailsService);
```

We just inject this service into the controller

```
function StudentController(StudentDetailsService) {
  StudentDetailsService.getStudentDetails().then(function (response) {
      // handle response
    });
}
angular.module('app').controller('StudentController', StudentController);
```

When to use?

Use `.service()` wherever you want to use a constructor. It is usually used to create public API's just like `getStudentDetails()`. But if you don't want to use a constructor and wish to use a simple API pattern instead, then there isn't much flexibility in `.service()`.

2) Factory

Even though we can achieve all the things using `.factory()` which we would, using `.services()`, it doesn't make `.factory()` "same as" `.service()`. It is much more powerful and flexible than `.service()`

A `.factory()` is a design pattern which is used to return a value.

There are two thumb rules to remember in case of factories

1. Factories return values
2. Factories (can) create objects (Any object)

Lets see some examples on what we can do using `.factory()`

Returning Objects Literals

Lets see an example where factory is used to return an object using a basic Revealing module pattern

```
function StudentDetailsService($http) {
  function getStudentDetails() {
    return $http.get('/details');
  }
  return {
    getStudentDetails: getStudentDetails
  };
}

angular.module('myapp').factory('StudentDetailsService', StudentDetailsService);
```

Usage inside a controller

```
function StudentController(StudentDetailsService) {
  StudentDetailsService.getStudentDetails().then(function (response) {
      // handle response
    });
```

```
}
angular.module('app').controller('StudentController', StudentController);
```

返回闭包

什么是闭包？

闭包是指引用了在局部使用但定义在外层作用域中的变量的函数。

下面是一个闭包的示例

```
function closureFunction(name) {
  function innerClosureFunction(age) { // innerClosureFunction() 是内部函数，一个闭包
    // 这里你可以同时操作 'age' 和 'name' 变量
  };
};
```

“奇妙”的部分是它可以访问父作用域中的 `name` 变量。

让我们在 .factory() 中使用上述闭包示例

```
function StudentDetailsService($http) {
  function closureFunction(name) {
  function innerClosureFunction(age) {
    // 这里你可以同时操作 'age' 和 'name' 变量
    };
  };
};

angular.module('我的应用').factory('学生详情服务', 学生详情服务);
```

在控制器中的使用

```
function StudentController(StudentDetailsService) {
  var myClosure = StudentDetailsService('学生姓名'); // 这现在拥有 innerClosureFunction()
  var callMyClosure = myClosure(24); // 这调用 innerClosureFunction()
};

angular.module('app').controller('StudentController', StudentController);
```

创建构造函数/实例

`.service()` 通过调用 `new` 创建构造函数，如上所示。`.factory()` 也可以通过调用 `new` 创建构造函数

让我们看一个如何实现的例子

```
function StudentDetailsService($http) {
  function Student() {
    this.age = function () {
        return '这是我的年龄';
    };
  }
Student.prototype.address = function () {
        return '这是我的地址';
  };
  return Student;
};
```



```
}
angular.module('app').controller('StudentController', StudentController);
```

Returning Closures

What is a closure?

Closures are functions that refer to variables that are used locally, BUT defined in an enclosing scope.

Following is an example of a closure

```
function closureFunction(name) {
  function innerClosureFunction(age) { // innerClosureFunction() is the inner function, a closure
    // Here you can manipulate 'age' AND 'name' variables both
  };
};
```

The *"wonderful"* part is that it can access the `name` which is in the parent scope.

Lets use the above closure example inside `.factory()`

```
function StudentDetailsService($http) {
  function closureFunction(name) {
  function innerClosureFunction(age) {
    // Here you can manipulate 'age' AND 'name' variables
    };
  };
};

angular.module('myapp').factory('StudentDetailsService', StudentDetailsService);
```

Usage inside a controller

```
function StudentController(StudentDetailsService) {
  var myClosure = StudentDetailsService('Student Name'); // This now HAS the innerClosureFunction()
  var callMyClosure = myClosure(24); // This calls the innerClosureFunction()
};

angular.module('app').controller('StudentController', StudentController);
```

Creating Constructors/instances

`.service()` creates constructors with a call to **`new`** as seen above. `.factory()` can also create constructors with a call to **`new`**

Lets see an example on how to achieve this

```
function StudentDetailsService($http) {
  function Student() {
    this.age = function () {
        return 'This is my age';
    };
  }
  Student.prototype.address = function () {
        return 'This is my address';
  };
  return Student;
};
```

```
angular.module('我的应用').factory('学生详情服务', 学生详情服务);
```

在控制器中的使用

```
function StudentController(StudentDetailsService) {
  var newStudent = new StudentDetailsService();

  //现在实例已经创建。可以访问它的属性。

newStudent.age();
 newStudent.address();

};

angular.module('app').controller('StudentController', StudentController);
```

第17.3节：$sce - 在模板中清理和渲染内容及资源

> $sce ("严格上下文转义") 是一个内置的 Angular 服务，用于自动清理模板中的内容和内部资源。

向模板中注入外部资源和原始 HTML时，需要手动使用$sce进行包装。

在此示例中，我们将创建一个简单的 $sce 清理过滤器：`。

演示

```
.filter('sanitizer', ['$sce', [function($sce) {
     return function(content) {
          return $sce.trustAsResourceUrl(content);
      };
}]);
```

模板中的用法

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

    // 清理外部资源
     <iframe ng-src="{{item.youtube_url | sanitizer}}">

// 清理并渲染 HTML
     <div ng-bind-html="{{item.raw_html_content| sanitizer}}"></div>

</div>
```

第17.4节：如何创建服务

```
angular.module("app")
  .service("counterService", function(){

    var service = {
        number: 0
    };

    return service;
```



```
angular.module('myapp').factory('StudentDetailsService', StudentDetailsService);
```

Usage inside a controller

```
function StudentController(StudentDetailsService) {
  var newStudent = new StudentDetailsService();

  //Now the instance has been created. Its properties can be accessed.

 newStudent.age();
 newStudent.address();

};

angular.module('app').controller('StudentController', StudentController);
```

Section 17.3: $sce - sanitize and render content and resources in templates

> $sce ("Strict Contextual Escaping") is a built-in angular service that automatically sanitize content and internal sources in templates.

injecting **external** sources and **raw HTML** into the template requires manual wrapping of$sce.

In this example we'll create a simple $sce sanitation filter :`.

Demo

```
.filter('sanitizer', ['$sce', [function($sce) {
     return function(content) {
          return $sce.trustAsResourceUrl(content);
      };
}]);
```

Usage in template

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

    // Sanitize external sources
    <ifrmae ng-src="{{item.youtube_url | sanitizer}}">

    // Sanitaize and render HTML
    <div ng-bind-html="{{item.raw_html_content| sanitizer}}"></div>

</div>
```

Section 17.4: How to create a Service

```
angular.module("app")
  .service("counterService", function(){

    var service = {
        number: 0
    };

    return service;
```

```
    });
```

第17.5节：如何使用服务

```
angular.module("app")

        // 自定义服务的注入方式与Angular内置服务相同
.controller("step1Controller", ['counterService', '$scope', function(counterService,
$scope) {
counterService.number++;
            // 绑定到对象（按引用），而非值，实现自动同步
$scope.counter = counterService;
        })
```

在使用此控制器的模板中，您可以这样写：

```
// 可编辑
<input ng-model="counter.number" />
```

或者

```
// 只读
<span ng-bind="counter.number"></span>
```

当然，在实际代码中，您会通过控制器上的方法与服务交互，而这些方法又会委托给服务。上面的示例只是每次在模板中使用控制器时简单地递增计数器的值。

AngularJS 中的服务是单例的：

服务是单例对象，每个应用只实例化一次（由 $injector 负责），并且是按需加载的（仅在必要时创建）。

> 单例是一种只允许创建自身唯一实例的类——并且提供对该实例的简单、便捷访问。如这里所述

第17.6节：如何使用

“数组语法”创建带依赖的服务

```
angular.module("app")
.service("counterService", ["fooService", "barService", function(anotherService, barService){

    var service = {
number: 0,
foo: function () {
            return fooService.bazMethod(); // 使用“fooService”
        },
bar: function () {
            return barService.bazMethod(); // 使用“barService”
        }
    };

    return service;
}]);
```



```
    });
```

Section 17.5: How to use a service

```
    angular.module("app")

        // Custom services are injected just like Angular's built-in services
        .controller("step1Controller", ['counterService', '$scope', function(counterService,
$scope) {
            counterService.number++;
            // bind to object (by reference), not to value, for automatic sync
            $scope.counter = counterService;
        })
```

In the template using this controller you'd then write:

```
// editable
<input ng-model="counter.number" />
```

or

```
// read-only
<span ng-bind="counter.number"></span>
```

Of course, in real code you would interact with the service using methods on the controller, which in turn delegate to the service. The example above simply increments the counter value each time the controller is used in a template.

Services in Angularjs are singletons:

Services are singleton objects that are instantiated only once per app (by the $injector) and lazy loaded (created only when necessary).

> A singleton is a class which only allows one instance of itself to be created - and gives simple, easy access to said instance. As stated here

Section 17.6: How to create a Service with dependencies using 'array syntax'

```
angular.module("app")
  .service("counterService", ["fooService", "barService", function(anotherService, barService){

    var service = {
        number: 0,
        foo: function () {
            return fooService.bazMethod(); // Use of 'fooService'
        },
        bar: function () {
            return barService.bazMethod(); // Use of 'barService'
        }
    };

    return service;
  }]);
```

第17.7节：注册服务

创建服务最常见且灵活的方式是使用 angular.module API 的 factory：

```
angular.module('myApp.services', []).factory('githubService', function() {
   var serviceInstance = {};
   // 我们的第一个服务
   return serviceInstance;
});
```

服务工厂函数可以是一个函数，也可以是一个数组，就像我们创建控制器的方式一样：

```
// 通过使用
// 方括号表示法创建工厂
angular.module('myApp.services', [])
.factory('githubService', [function($http) {
}]);
```

要在服务上暴露一个方法，我们可以将其作为属性放在服务对象上。

```
angular.module('myApp.services', [])
    .factory('githubService', function($http) {
        var githubUrl = 'https://api.github.com';
        var runUserRequest = function(username, path) {
        // 返回来自 $http 服务的 Promise
        // 该服务使用 JSONP 调用 Github API
        return $http({
method: 'JSONP',
            url: githubUrl + '/users/' +
            username + '/' +
path + '?callback=JSON_CALLBACK'
        });
    }
// 返回带有单个函数的服务对象
// 事件
return {
events: function(username) {
    return runUserRequest(username, 'events');
    }
};
```

beiliandoc.com

Section 17.7: Registering a Service

The most common and flexible way to create a service uses the angular.module API factory:

```
angular.module('myApp.services', []).factory('githubService', function() {
   var serviceInstance = {};
   // Our first service
   return serviceInstance;
});
```

The service factory function can be either a function or an array, just like the way we create controllers:

```
// Creating the factory through using the
// bracket notation
angular.module('myApp.services', [])
.factory('githubService', [function($http) {
}]);
```

To expose a method on our service, we can place it as an attribute on the service object.

```
angular.module('myApp.services', [])
    .factory('githubService', function($http) {
        var githubUrl = 'https://api.github.com';
        var runUserRequest = function(username, path) {
        // Return the promise from the $http service
        // that calls the Github API using JSONP
        return $http({
            method: 'JSONP',
            url: githubUrl + '/users/' +
            username + '/' +
            path + '?callback=JSON_CALLBACK'
        });
    }
// Return the service object with a single function
// events
return {
    events: function(username) {
    return runUserRequest(username, 'events');
    }
};
```

第18章：区分服务（Service）与工厂（Factory）

第18.1节：工厂与服务的最终区别

定义如下：

服务基本上是构造函数。它们使用'this'关键字。

工厂是简单函数，因此返回一个对象。

底层原理：

工厂内部调用提供者函数。

服务内部调用工厂函数。

讨论：

工厂可以在返回对象字面量之前运行代码。

但同时，服务也可以被编写成返回对象字面量并在返回之前运行代码。
尽管这有悖于服务设计为构造函数的初衷。

事实上，JavaScript 中的构造函数可以返回它们想要的任何内容。

那么哪种更好呢？

服务的构造函数语法更接近 ES6 的类语法，因此迁移会更容易。

总结

总之，提供者、工厂和服务都是提供者。

当你的提供者中只需要一个 $get() 函数时，工厂是提供者的一种特殊情况。它允许你用更少的代码来编写。

当你想返回一个新对象的实例时，服务是工厂的一种特殊情况，同样具有用更少代码编写的好处。

baiinfodoc.com

Chapter 18: Distinguishing Service vs Factory

Section 18.1: Factory VS Service once-and-for-all

By definition:

Services are basically constructor functions. They use 'this' keyword.

Factories are simple functions hence return an object.

Under the hood:

Factories internally calls provider function.

Services internally calls Factory function.

Debate:

Factories can run code before we return our object literal.

But at the same time, Services can also be written to return an object literal and to run code before returning.
Though that is contra productive as services are designed to act as constructor function.

In fact, constructor functions in JavaScript can return whatever they want.

So which one is better?

The constructor syntax of services is more close to class syntax of ES6. So migration will be easy.

Summary

So in summary, provider, factory, and service are all providers.

A factory is a special case of a provider when all you need in your provider is a $get() function. It allows you to write it with less code.

A service is a special case of a factory when you want to return an instance of a new object, with the same benefit of writing less code.

```
mod.provider("myProvider", function() {          provider
    this.$get = function() {                      factory
        return new function() {                   service
            this.getValue = function() {
                return "My Value";
            };
        };
    };
});
```

www.simplygoodcode.com

belindoc.com


```
mod.provider("myProvider", function() {          provider
    this.$get = function() {                      factory
        return new function() {                   service
            this.getValue = function() {
                return "My Value";
            };
        };
    };
});
```

www.simplygoodcode.com

第19章：使用 $q 服务的 Angular 承诺（promises）

第19.1节：使用 $q.when() 将简单值包装成承诺

如果你只需要将值包装成承诺，就不需要像下面这样使用冗长的语法：

```
// 过于冗长
var defer;
defer = $q.defer();
defer.resolve(['one', 'two']);
return defer.promise;
```

在这种情况下，你可以直接写：

```
//更好
return $q.when(['one', 'two']);
```

> **$q.when 及其别名 $q.resolve**
>
> 将一个可能是值或（第三方）then-able（可调用then方法的）promise的对象包装成一个 $q promise。当你处理的对象可能是也可能不是promise，或者promise来自一个不可信的来源时，这非常有用。
>
> — AngularJS $q 服务 API 参考 - $q.when

随着 AngularJS v1.4.1 的发布

你也可以使用一个符合 ES6 标准的别名 `resolve`

```
//与 when 完全相同
return $q.resolve(['one', 'two'])
```

第19.2节：使用 $q 服务的 angular promise

$q 是一个内置服务，帮助执行异步函数并在它们处理完成后使用其返回值（或异常）。

$q与$rootScope.Scope模型观察机制集成，这意味着更快地将
解决或拒绝的结果传播到你的模型中，避免不必要的浏览器重绘，从而防止
界面闪烁。

在我们的示例中，我们调用工厂getMyData，它返回一个promise对象。如果该对象被resolved，则返回一个随机数。如果被rejected，则在2秒后返回带有错误信息的拒绝。

在Angular工厂中

```
function getMyData($timeout, $q) {
  return function() {
    // 模拟异步函数
    var promise = $timeout(function() {
      if(Math.round(Math.random())) {
        return '数据已接收！'
```


Chapter 19: Angular promises with $q service

Section 19.1: Wrap simple value into a promise using $q.when()

If all you need is to wrap the value into a promise, you don't need to use the long syntax like here:

```
//OVERLY VERBOSE
var defer;
defer = $q.defer();
defer.resolve(['one', 'two']);
return defer.promise;
```

In this case you can just write:

```
//BETTER
return $q.when(['one', 'two']);
```

> **$q.when and its alias $q.resolve**
>
> Wraps an object that might be a value or a (3rd party) then-able promise into a $q promise. This is useful when you are dealing with an object that might or might not be a promise, or if the promise comes from a source that can't be trusted.
>
> — AngularJS $q Service API Reference - $q.when

With the release of AngularJS v1.4.1

You can also use an ES6-consistent alias `resolve`

```
//ABSOLUTELY THE SAME AS when
return $q.resolve(['one', 'two'])
```

Section 19.2: Using angular promises with $q service

$q is a built-in service which helps in executing asynchronous functions and using their return values(or exception) when they are finished with processing.

$q is integrated with the $rootScope.Scope model observation mechanism, which means faster propagation of resolution or rejection into your models and avoiding unnecessary browser repaints, which would result in flickering UI.

In our example, we call our factory getMyData, which return a promise object. If the object is resolved, it returns a random number. If it is rejected, it return a rejection with an error message after 2 seconds.

In Angular factory

```
function getMyData($timeout, $q) {
  return function() {
    // simulated async function
    var promise = $timeout(function() {
      if(Math.round(Math.random())) {
        return 'data received!'
```


GoalKicker.com

```
      } else {
        return $q.reject('哦不，出错了！请重试')
      }
    }, 2000);
    return promise;
  }
}
```

在调用中使用 Promise

```
angular.module('app', [])
.factory('getMyData', getMyData)
.run(function(getData) {
  var promise = getData()
    .then(function(string) {
      console.log(string)
    }, function(error) {
      console.error(error)
    })
.finally(function() {
      console.log('Finished at:', new Date())
    })
})
```

要使用 Promise，需要注入 $q 作为依赖。在这里我们在 getMyData 工厂中注入了 $q。

```
var defer = $q.defer();
```

通过调用 $q.defer() 构造一个新的 deferred 实例。deferred 对象只是

一个暴露了 promise 以及用于解决该 promise 的相关方法的对象。它是通过 $q.deferred() 函数构造的，并暴露三个主要方法：resolve()、reject() 和 notify()。

- resolve(value) – 使用该值解决派生的 promise。
- reject(reason) – 使用该原因拒绝派生的 promise。
- notify(value) - 提供关于承诺执行状态的更新。在承诺被解决或拒绝之前，可能会多次调用此方法。

属性

关联的 promise 对象通过 promise 属性访问。promise – {Promise} – 与此 deferred 关联的 promise 对象。

当创建一个 deferred 实例时，会创建一个新的 promise 实例，可以通过调用 deferred.promise 来获取。

promise 对象的目的是允许相关方在延迟任务完成时访问其结果。

Promise 方法 -

- then(successCallback, [errorCallback], [notifyCallback]) – 无论 promise 是何时被解决或拒绝，then 都会在结果可用时异步调用成功或错误回调之一。回调函数接收一个参数：结果或拒绝原因。此外，notify 回调可能会被调用零次或多次，以在 promise 被解决或拒绝之前提供进度指示。


```
      } else {
        return $q.reject('oh no an error! try again')
      }
    }, 2000);
    return promise;
  }
}
```

Using Promises on call

```
angular.module('app', [])
.factory('getMyData', getMyData)
.run(function(getData) {
  var promise = getData()
    .then(function(string) {
      console.log(string)
    }, function(error) {
      console.error(error)
    })
    .finally(function() {
      console.log('Finished at:', new Date())
    })
})
```

To use promises, inject $q as dependency. Here we injected $q in `getMyData` factory.

```
var defer = $q.defer();
```

A new instance of deferred is constructed by calling `$q.defer()`

A deferred object is simply an object that exposes a promise as well as the associated methods for resolving that promise. It is constructed using the `$q.deferred()` function and exposes three main methods: `resolve()`, `reject()`, and `notify()`.

- `resolve(value)` – resolves the derived promise with the value.
- `reject(reason)` – rejects the derived promise with the reason.
- `notify(value)` - provides updates on the status of the promise's execution. This may be called multiple times before the promise is either resolved or rejected.

Properties

The associated promise object is accessed via the promise property. `promise` – {Promise} – promise object associated with this deferred.

A new promise instance is created when a deferred instance is created and can be retrieved by calling `deferred.promise`.

The purpose of the `promise` object is to allow for interested parties to get access to the result of the deferred task when it completes.

Promise Methods -

- `then(successCallback, [errorCallback], [notifyCallback])` – Regardless of when the promise was or will be resolved or rejected, then calls one of the success or error callbacks asynchronously as soon as the result is available. The callbacks are called with a single argument: the result or rejection reason. Additionally, the notify callback may be called zero or more times to provide a progress indication, before the promise is resolved or rejected.

- catch(errorCallback) – 是 promise.then(null, errorCallback) 的简写。fi
- nally(callback, notifyCallback) – 允许你观察 promise 的完成或拒绝，但不会修改最终值。

promise 最强大的功能之一是能够将它们串联起来。这允许数据在链中流动，并在每个步骤中被操作和变更。以下示例演示了这一点：

示例 1：

```
// 创建一个承诺，解析后返回4。
function getNumbers() {

  var promise = $timeout(function() {
    return 4;
  }, 1000);

  return promise;
}

// 解析getNumbers()并链式调用then()，将初始数字从4递减到0，然后输出字符串。

getNumbers()
  .then(function(num) {
      // 4
console.log(num);
      return --num;
  })
.then(function (num) {
      // 3
console.log(num);
      return --num;
  })
.then(function (num) {
      // 2
console.log(num);
      return --num;
  })
.然后(函数 (数字) {
      // 1
console.log(num);
      return --num;
  })
.然后(函数 (数字) {
      // 0
console.log(num);
      返回 '我们完成了！';
  })
.然后(函数 (文本) {
      // "我们完成了！"
      控制台.日志(文本);
  });
```

第19.3节：使用$q构造函数创建承诺

$q构造函数用于从使用回调返回结果的异步API创建承诺。

```
$q(function(resolve, reject) {...})
```


- **catch**(errorCallback) – shorthand for promise.then(null, errorCallback)
- **finally**(callback, notifyCallback) – allows you to observe either the fulfillment or rejection of a promise, but to do so without modifying the final value.

One of the most powerful features of promises is the ability to chain them together. This allows the data to flow through the chain and be manipulated and mutated at each step. This is demonstrated with the following example:

Example 1:

```
// Creates a promise that when resolved, returns 4.
function getNumbers() {

  var promise = $timeout(function() {
    return 4;
  }, 1000);

  return promise;
}

// Resolve getNumbers() and chain subsequent then() calls to decrement
// initial number from 4 to 0 and then output a string.
getNumbers()
  .then(function(num) {
      // 4
      console.log(num);
      return --num;
  })
  .then(function (num) {
      // 3
      console.log(num);
      return --num;
  })
   .then(function (num) {
      // 2
      console.log(num);
      return --num;
  })
  .then(function (num) {
      // 1
      console.log(num);
      return --num;
  })
  .then(function (num) {
      // 0
      console.log(num);
      return 'And we are done!';
  })
  .then(function (text) {
      // "And we are done!"
      console.log(text);
  });
```

Section 19.3: Using the $q constructor to create promises

The $q constructor function is used to create promises from asynchronous APIs that use callbacks to return results.

```
$q(function(resolve, reject) {...})
```

构造函数接收一个函数，该函数被调用时带有两个参数，`resolve`和`reject`，这两个函数用于解决或拒绝承诺。

示例 1：

```
函数 $timeout(fn, 延迟) {
    返回 = $q(函数(resolve, reject) {
         setTimeout(函数() {
              尝试 {
                 让 r = fn();
              解决(r);
              }
              捕获 (e) {
reject(e);
              }
         }, delay);
    };
}
```

上述示例使用WindowTimers.setTimeout API创建了一个Promise。AngularJS框架提供了该函数的更详细版本。有关用法，请参见AngularJS $timeout 服务API参考。

示例2：

```
$scope.divide = function(a, b) {
    return $q(function(resolve, reject) {
      if (b===0) {
        return reject("Cannot devide by 0")
      } else {
        return resolve(a/b);
      }
    });
}
```

上述代码展示了一个返回Promise的除法函数，如果计算不可能，将返回拒绝理由，否则返回结果的Promise。

然后你可以调用并使用.then

```
$scope.divide(7, 2).then(function(result) {
    // 将返回3.5
}, function(err) {
    // 不会执行
})

$scope.divide(2, 0).then(function(result) {
    // 由于除以0导致计算失败，因此不会执行
}, function(err) {
    // 将返回错误字符串。
})
```

第19.4节：避免$q Deferred反模式

> **避免此反模式**
>
> ```
> var myDeferred = $q.defer();
>
> $http(config).then(function(res) {
> ```


The constructor function receives a function that is invoked with two arguments, `resolve` and `reject` that are functions which are used to either resolve or reject the promise.

Example 1:

```
function $timeout(fn, delay) {
    return = $q(function(resolve, reject) {
         setTimeout(function() {
              try {
                 let r = fn();
                 resolve(r);
              }
              catch (e) {
                 reject(e);
              }
         }, delay);
    };
}
```

The above example creates a promise from the WindowTimers.setTimeout API. The AngularJS framework provides a more elaborate version of this function. For usage, see the AngularJS $timeout Service API Reference.

Example 2:

```
$scope.divide = function(a, b) {
    return $q(function(resolve, reject) {
      if (b===0) {
        return reject("Cannot devide by 0")
      } else {
        return resolve(a/b);
      }
    });
}
```

The above code showing a promisified division function, it will return a promise with the result or reject with a reason if the calculation is impossible.

You can then call and use `.then`

```
$scope.divide(7, 2).then(function(result) {
    // will return 3.5
}, function(err) {
    // will not run
})

$scope.divide(2, 0).then(function(result) {
    // will not run as the calculation will fail on a divide by 0
}, function(err) {
    // will return the error string.
})
```

Section 19.4: Avoid the $q Deferred Anti-Pattern

> **Avoid this Anti-Pattern**
>
> ```
> var myDeferred = $q.defer();
>
> $http(config).then(function(res) {
> ```


bellindoc.com

```
myDeferred.resolve(res);
}, function(error) {
myDeferred.reject(error);
});

return myDeferred.promise;
```

没有必要使用$q.defer来制造一个promise，因为$http服务已经返回了一个promise。

```
//INSTEAD
return $http(config);
```

简单地返回由 $http 服务创建的 promise。

第19.5节：使用 $q.all 处理多个 promise

你可以使用 $q.all 函数，在一组 promise 成功解决后调用 .then 方法，并获取它们解决的数据。

示例：

JS:

```
$scope.data = []

 $q.all([
$http.get("data.json"),
$http.get("more-data.json"),
    ]).then(function(responses) {
      $scope.data = responses.map((resp) => resp.data);
    });
```

上述代码对本地 json 文件中的数据运行了两次 $http.get，当两个 get 方法都完成时，它们会解决各自关联的 promise，当数组中的所有 promise 都解决后，.then 方法开始执行，参数 responses 数组中包含了两个 promise 的数据。

然后将数据映射，以便能够显示在模板上，接着我们可以显示

HTML：

```
<ul>
   <li ng-repeat="d in data">
      <ul>
         <li ng-repeat="item in d">{{item.name}}: {{item.occupation}}</li>
      </ul>
   </li>
</ul>
```

JSON：

```
[{
  "name": "alice",
  "occupation": "manager"
}, {
  "name": "bob",
  "occupation": "developer"
}]
```



```
   myDeferred.resolve(res);
}, function(error) {
   myDeferred.reject(error);
});

return myDeferred.promise;
```

There is no need to manufacture a promise with `$q.defer` as the $http service already returns a promise.

```
//INSTEAD
return $http(config);
```

Simply return the promise created by the $http service.

Section 19.5: Using $q.all to handle multiple promises

You can use the `$q.all` function to call a `.then` method after an array of promises has been successfully resolved and fetch the data they resolved with.

Example:

JS:

```
$scope.data = []

$q.all([
   $http.get("data.json"),
   $http.get("more-data.json"),
   ]).then(function(responses) {
     $scope.data = responses.map((resp) => resp.data);
   });
```

The above code runs `$http.get` 2 times for data in local json files, when both `get` method complete they resolve their associated promises, when all the promises in the array are resolved, the `.then` method starts with both promises data inside the `responses` array argument.

The data is then mapped so it could be shown on the template, we can then show

HTML:

```
<ul>
   <li ng-repeat="d in data">
      <ul>
         <li ng-repeat="item in d">{{item.name}}: {{item.occupation}}</li>
      </ul>
   </li>
</ul>
```

JSON:

```
[{
  "name": "alice",
  "occupation": "manager"
}, {
  "name": "bob",
  "occupation": "developer"
}]
```

第19.6节：使用$q.defer延迟操作

我们可以使用$q将操作延迟到未来，同时在当前拥有一个待处理的承诺对象，通过使用 `$q.defer` 我们创建一个将在未来被解决或拒绝的承诺。

该方法不等同于使用$q构造函数，因为我们使用$q.defer来将一个可能返回也可能不返回（或者从未返回过）promise的现有例程转换为promise。

示例：

```
var runAnimation = function(animation, duration) {
    var deferred = $q.defer();
    try {
        ...
        // 运行给定持续时间的动画
deferred.resolve("done");
    } catch (err) {
        // 如果出错，我们希望运行.then的错误处理器
deferred.reject(err);
    }
    return deferred.promise;
}

// 然后
runAnimation.then(function(status) {}, function(error) {})
```

1. 确保你始终返回一个deferred.promise对象，否则调用.then时可能会出错
2. 确保你始终解决或拒绝你的deferred对象，否则.then可能不会执行，且你可能会导致内存泄漏

beliinec.com

Section 19.6: Deferring operations using $q.defer

We can use `$q` to defer operations to the future while having a pending promise object at the present, by using `$q.defer` we create a promise that will either resolve or reject in the future.

This method is not equivalent of using the `$q` constructor, as we use `$q.defer` to promisify an existing routine that may or may not return (or had ever returned) a promise at all.

Example:

```
var runAnimation = function(animation, duration) {
    var deferred = $q.defer();
    try {
        ...
        // run some animation for a given duration
        deferred.resolve("done");
    } catch (err) {
        // in case of error we would want to run the error hander of .then
        deferred.reject(err);
    }
    return deferred.promise;
}

// and then
runAnimation.then(function(status) {}, function(error) {})
```

1. Be sure you always return a the `deferred.promise` object or risk an error when invoking `.then`
2. Make sure you always resolve or reject your deferred object or `.then` may not run and you risk a memory leak

第20章：依赖注入

第20.1节：动态注入

还有一种动态请求组件的选项。你可以使用$injector服务来实现：

```
myModule.controller('myController', ['$injector', function($injector) {
    var myService = $injector.get('myService');
}]);
```

注意：虽然此方法可用于防止可能破坏应用的循环依赖问题，但不建议通过使用它来绕过该问题。循环依赖通常表明你的应用架构存在缺陷，应当加以解决。

第20.2节：在原生JavaScript中动态加载AngularJS服务

你可以使用AngularJS的injector()方法在原生JavaScript中加载AngularJS服务。每个通过angular.element()调用获取的jqLite元素都具有一个injector()方法，可用于获取注入器。

```
var service;
var serviceName = 'myService';

        var ngAppElement = angular.element(document.querySelector('[ng-app],[data-ng-app]') || document);
var injector = ngAppElement.injector();

if(injector && injector.has(serviceNameToInject)) {
    service = injector.get(serviceNameToInject);
}
```

在上述示例中，我们尝试获取包含 AngularJS 应用根的 jqLite 元素（ngAppElement）。为此，我们使用 angular.element() 方法，搜索包含 ng-app 或 data-ng-app 属性的 DOM 元素，如果不存在，则回退到 document 元素。我们使用 ngAppElement 来获取注入器实例（通过 ngAppElement.injector()）。注入器实例用于检查要注入的服务是否存在（通过 injector.has()），然后将服务加载到 service 变量中（通过 injector.get()）。

Chapter 20: Dependency Injection

Section 20.1: Dynamic Injections

There is also an option to dynamically request components. You can do it using the `$injector` service:

```
myModule.controller('myController', ['$injector', function($injector) {
    var myService = $injector.get('myService');
}]);
```

Note: while this method could be used to prevent the circular dependency issue that might break your app, it is not considered best practice to bypass the problem by using it. Circular dependency usually indicates there is a flaw in your application's architecture, and you should address that instead.

Section 20.2: Dynamically load AngularJS service in vanilla JavaScript

You can load AngularJS services in vanilla JavaScript using AngularJS `injector()` method. Every jqLite element retrieved calling `angular.element()` has a method `injector()` that can be used to retrieve the injector.

```
var service;
var serviceName = 'myService';

var ngAppElement = angular.element(document.querySelector('[ng-app],[data-ng-app]') || document);
var injector = ngAppElement.injector();

if(injector && injector.has(serviceNameToInject)) {
    service = injector.get(serviceNameToInject);
}
```

In the above example we try to retrieve the jqLite element containing the root of the AngularJS application (`ngAppElement`). To do that, we use `angular.element()` method, searching for a DOM element containing `ng-app` or `data-ng-app` attribute or, if it does not exists, we fall back to `document` element. We use `ngAppElement` to retrieve injector instance (with `ngAppElement.injector()`). The injector instance is used to check if the service to inject exists (with `injector.has()`) and then to load the service (with `injector.get()`) inside `service` variable.

第21章：事件

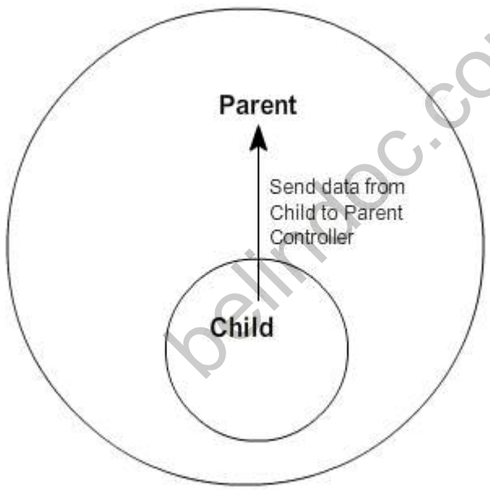
参数	值类型
事件	对象 {name: "eventName", targetScope: Scope, defaultPrevented: false, currentScope: ChildScope}
参数	随事件执行传递的数据

第21.1节：使用Angular事件系统

$scope.$emit

使用$scope.$emit会向上通过作用域层级触发一个事件名称，并通知$scope。事件生命周期从调用$emit的作用域开始。

工作线框图：



$scope.$broadcast

使用$scope.$broadcast会向下触发一个事件到$scope。我们可以使用$scope.$on监听这些事件。

工作线框图：

Chapter 21: Events

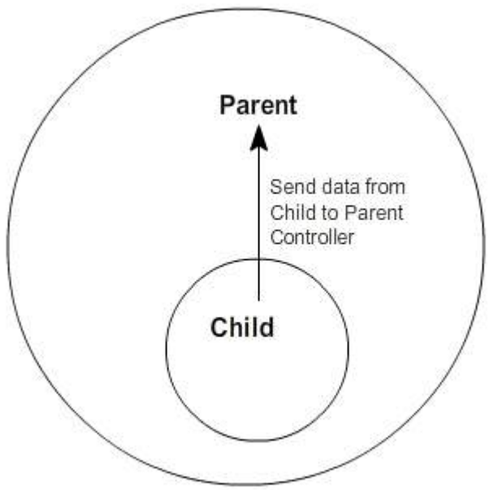
Parameters	Values types
event	Object {name: "eventName", targetScope: Scope, defaultPrevented: false, currentScope: ChildScope}
args	data that has been passed along with event execution

Section 21.1: Using angular event system

$scope.$emit

Using `$scope.$emit` will fire an event name upwards through the scope hierarchy and notify to the `$scope`.The event life cycle starts at the scope on which `$emit` was called.

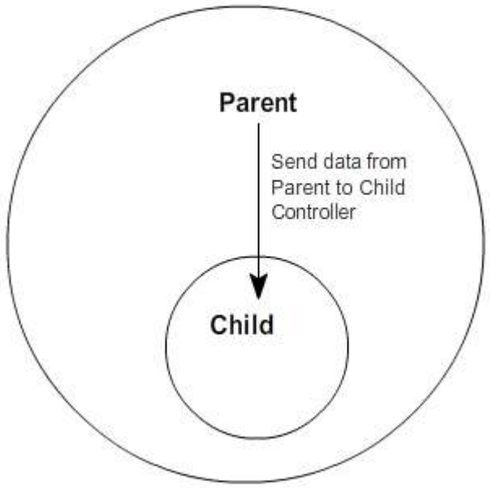
Working wireframe :



$scope.$broadcast

Using `$scope.$broadcast` will fire an event down the `$scope`. We can listen of these events using `$scope.$on`

Working wireframe :



语法：

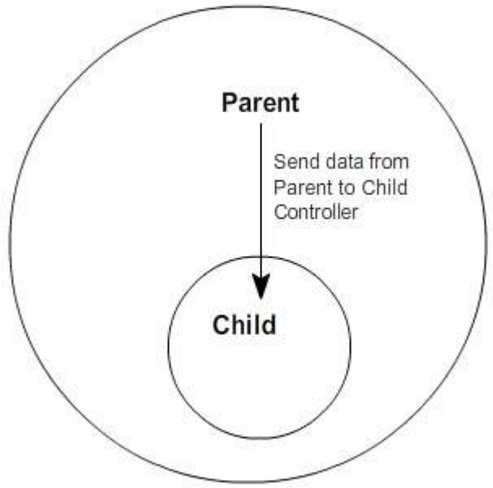
```
// 向上传递事件
$scope.$emit('myCustomEvent', '要发送的数据');

// 向下传递事件
$scope.$broadcast('myCustomEvent', {
  someProp: '某个值'
});

// 在相关的 $scope 中监听事件
$scope.$on('myCustomEvent', function (event, data) {
  console.log(data); // '来自事件的数据'
});
```

除了使用$scope，你也可以使用$rootScope，这样你的事件将在所有控制器中可用，而不管该控制器的作用域如何

清理 AngularJS 中注册的事件

清理已注册事件的原因是，即使控制器已被销毁，已注册事件的处理仍然存在。因此代码肯定会出现意料之外的运行情况。

```
// 向上传递事件
$rootScope.$emit('myEvent', '要发送的数据');

// 监听一个事件
var listenerEventHandler = $rootScope.$on('myEvent', function(){
    //处理代码
});

$scope.$on('$destroy', function() {
    listenerEventHandler();
});
```

Syntax :

```
// firing an event upwards
$scope.$emit('myCustomEvent', 'Data to send');

// firing an event downwards
$scope.$broadcast('myCustomEvent', {
  someProp: 'some value'
});

// listen for the event in the relevant $scope
$scope.$on('myCustomEvent', function (event, data) {
  console.log(data); // 'Data from the event'
});
```

Instead of `$scope` you can use `$rootScope`, in that case your event will be available in all the controllers regardless of that controllers scope

Clean registered event in AngularJS

The reason to clean the registered events because even the controller has been destroyed the handling of registered event are still alive. So the code will run as unexpected for sure.

```
// firing an event upwards
$rootScope.$emit('myEvent', 'Data to send');

// listening an event
var listenerEventHandler = $rootScope.$on('myEvent', function(){
    //handle code
});

$scope.$on('$destroy', function() {
    listenerEventHandler();
});
```

第21.2节：始终在作用域的 $destroy 事件上注销 $rootScope.$on 监听器

如果导航到另一个控制器，$rootScope.$on 监听器将保留在内存中。如果控制器超出作用域，这将导致内存泄漏。

不要

```
angular.module('app').controller('badExampleController', badExample);

badExample.$inject = ['$scope', '$rootScope'];
function badExample($scope, $rootScope) {

    $rootScope.$on('post:created', function postCreated(event, data) {});

}
```

要

```
angular.module('app').controller('goodExampleController', goodExample);

goodExample.$inject = ['$scope', '$rootScope'];
function goodExample($scope, $rootScope) {

    var deregister = $rootScope.$on('post:created', function postCreated(event, data) {});

$scope.$on('$destroy', function destroyScope() {
        deregister();
    });

}
```

第21.3节：用途和意义

这些事件可用于两个或多个控制器之间的通信。

`$emit` 向上通过作用域层级分发事件，而 `$broadcast` 向下分发事件到所有子作用域。这里对此有很好的解释。

控制器之间通信基本上有两种场景：

1. 当控制器具有父子关系时。（在这种场景下我们通常使用 $scope）
2. 当控制器彼此不独立且需要相互通知对方的活动时。（在这种场景下我们可以使用 `$rootScope`）

例如：对于任何电子商务网站，假设我们有 ProductListController（控制产品列表页面当点击任何产品品牌时）和 Cart Controller（管理购物车项目）。现在，当我们点击 添加到购物车按钮时，也必须通知 CartController，以便它能在网站导航栏中反映新的购物车项目数量/详情。这可以通过 $rootScope 实现。

使用 `$scope.$emit`

Section 21.2: Always deregister $rootScope.$on listeners on the scope $destory event

$rootScope.$on listeners will remain in memory if you navigate to another controller. This will create a memory leak if the controller falls out of scope.

Don't

```
angular.module('app').controller('badExampleController', badExample);

badExample.$inject = ['$scope', '$rootScope'];
function badExample($scope, $rootScope) {

    $rootScope.$on('post:created', function postCreated(event, data) {});

}
```

Do

```
angular.module('app').controller('goodExampleController', goodExample);

goodExample.$inject = ['$scope', '$rootScope'];
function goodExample($scope, $rootScope) {

    var deregister = $rootScope.$on('post:created', function postCreated(event, data) {});

    $scope.$on('$destroy', function destroyScope() {
        deregister();
    });

}
```

Section 21.3: Uses and significance

These events can be used to communicate between 2 or more controllers.

`$emit` dispatches an event upwards through the scope hierarchy, while `$broadcast` dispatches an event downwards to all child scopes.This has been beautifully explained here.

There can be basically two types of scenario while communicating among controllers:

1. When controllers have Parent-Child relationship. (we can mostly use `$scope` in such scenarios)
2. When controllers are not independent to each other and are needed to be informed about each others activity. (we can use `$rootScope` in such scenarios)

eg: For any ecommerce website, suppose we have `ProductListController`(which controls the product listing page when any product brand is clicked) and `CartController` (to manage cart items) . Now, when we click on **Add to Cart** button , it has to be informed to `CartController` as well, so that it can reflect new cart item count/details in the navigation bar of the website. This can be achieved using `$rootScope`.

With `$scope.$emit`

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.4/angular.js"></script>
    <script>
var app = angular.module('app', []);

        app.controller("FirstController", function ($scope) {
            $scope.$on('eventName', function (event, args) {
                $scope.message = args.message;
            });
        });

app.controller("SecondController", function ($scope) {
            $scope.handleClick = function (msg) {
$scope.$emit('eventName', {message: msg});
            };
        });

    </script>
  </head>
  <body ng-app="app">
    <div ng-controller="FirstController" style="border:2px ;padding:5px;">
        <h1>父控制器</h1>
        <p>发送消息 : {{message}}</p>
        <br />
        <div ng-controller="SecondController" style="border:2px;padding:5px;">
            <h1>子控制器</h1>
            <input ng-model="msg">
            <button ng-click="handleClick(msg);">Emit</button>
        </div>
    </div>
  </body>
</html>
```

使用$scope.$broadcast:

```
<html>
  <head>
    <title>广播</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.4/angular.js"></script>
    <script>
var app = angular.module('app', []);

        app.controller("FirstController", function ($scope) {
            $scope.handleClick = function (msg) {
$scope.$broadcast('eventName', {message: msg});
            };

        });

app.controller("SecondController", function ($scope) {
            $scope.$on('eventName', function (event, args) {
                $scope.message = args.message;
});
        });

    </script>
  </head>
  <body ng-app="app">
    <div ng-controller="FirstController" style="border:2px solid ; padding:5px;">
        <h1>父控制器</h1>
```



```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.4/angular.js"></script>
    <script>
        var app = angular.module('app', []);

        app.controller("FirstController", function ($scope) {
            $scope.$on('eventName', function (event, args) {
                $scope.message = args.message;
            });
        });

        app.controller("SecondController", function ($scope) {
            $scope.handleClick = function (msg) {
                $scope.$emit('eventName', {message: msg});
            };
        });

    </script>
  </head>
  <body ng-app="app">
    <div ng-controller="FirstController" style="border:2px ;padding:5px;">
        <h1>Parent Controller</h1>
        <p>Emit Message : {{message}}</p>
        <br />
        <div ng-controller="SecondController" style="border:2px;padding:5px;">
            <h1>Child Controller</h1>
            <input ng-model="msg">
            <button ng-click="handleClick(msg);">Emit</button>
        </div>
    </div>
  </body>
</html>
```

With $scope.$broadcast:

```
<html>
  <head>
    <title>Broadcasting</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.4/angular.js"></script>
    <script>
        var app = angular.module('app', []);

        app.controller("FirstController", function ($scope) {
            $scope.handleClick = function (msg) {
                $scope.$broadcast('eventName', {message: msg});
            };

        });

        app.controller("SecondController", function ($scope) {
            $scope.$on('eventName', function (event, args) {
                $scope.message = args.message;
            });
        });

    </script>
  </head>
  <body ng-app="app">
    <div ng-controller="FirstController" style="border:2px solid ; padding:5px;">
        <h1>Parent Controller</h1>
```

```
        <input ng-model="msg">
        <button ng-click="handleClick(msg);">广播</button>
        <br /><br />
        <div ng-controller="SecondController" style="border:2px solid ;padding:5px;">
            <h1>子控制器</h1><p>广播消息
             : {{message}}</p>

    </div>
  </body>
</html>
```

belindoc.com


```
        <input ng-model="msg">
        <button ng-click="handleClick(msg);">Broadcast</button>
        <br /><br />
        <div ng-controller="SecondController" style="border:2px solid ;padding:5px;">
            <h1>Child Controller</h1>
            <p>Broadcast Message : {{message}}</p>
        </div>
    </div>
  </body>
</html>
```

第22章：共享数据

第22.1节：使用 ngStorage 共享数据

首先，在你的 index.html 中引入ngStorage源文件。

注入ngStorage源文件的示例代码如下：

```
<head>
    <title>Angular JS ngStorage</title>
    <script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script src="https://rawgithub.com/gsklee/ngStorage/master/ngStorage.js"></script>
</head>
```

ngStorage为你提供了两种存储方式，分别是：$localStorage和$sessionStorage。你需要引入 ngStorage 并注入这些服务。

假设ng-app="myApp"，那么你可以按如下方式注入ngStorage：

```
var app = angular.module('我的应用', ['ngStorage']);
        app.controller('控制器一', function($localStorage,$sessionStorage) {
        // 一个共享的对象
        var 示例对象 = {
          名称: 'angularjs',
          值: 1
        };
$localStorage.共享的值 = 示例对象;
       $sessionStorage.共享的值 = 示例对象;
    })
.controller('控制器二', function($localStorage,$sessionStorage) {
    console.log('localStorage: '+ $localStorage +'sessionStorage: '+$sessionStorage);
})
```

$localStorage 和 $sessionStorage 在任何控制器中只要注入这些服务就可以全局访问。

你也可以使用 HTML5 的 localStorage 和 sessionStorage。但是，使用 HTML5 localStorage 需要你在使用或保存对象之前进行序列化和反序列化。

例如：

```
var 我的对象 = {
名字: "Nic",
    姓氏: "Raboy",
    网站: "https://www.google.com"
}
//如果你想保存到localStorage，先序列化它
window.localStorage.set("saved", JSON.stringify(myObj));

//unserialize to get object
var myObj = JSON.parse(window.localStorage.get("saved"));
```

第22.2节：使用服务在控制器之间共享数据

我们可以创建一个service来set和get控制器之间的数据，然后将该服务注入到

Chapter 22: Sharing Data

Section 22.1: Using ngStorage to share data

Firstly, include the **ngStorage** source in your index.html.

An example injecting `ngStorage` src would be:

```
<head>
    <title>Angular JS ngStorage</title>
    <script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>
    <script src="https://rawgithub.com/gsklee/ngStorage/master/ngStorage.js"></script>
</head>
```

`ngStorage` gives you 2 storage namely: `$localStorage` and `$sessionStorage`. You need to require ngStorage and Inject the services.

Suppose if `ng-app="myApp"`, then you would be injecting `ngStorage` as following:

```
var app = angular.module('myApp', ['ngStorage']);
        app.controller('controllerOne', function($localStorage,$sessionStorage) {
        // an object to share
        var sampleObject = {
            name: 'angularjs',
            value: 1
        };
       $localStorage.valueToShare = sampleObject;
       $sessionStorage.valueToShare = sampleObject;
    })
.controller('controllerTwo', function($localStorage,$sessionStorage) {
    console.log('localStorage: '+ $localStorage +'sessionStorage: '+$sessionStorage);
})
```

`$localStorage` and `$sessionStorage` is globally accessible through any controllers as long as you inject those services in the controllers.

You can also use the `localStorage` and `sessionStorage` of `HTML5`. However, using `HTML5` `localStorage` would require you to serialize and deserialize your objects before using or saving them.

For example:

```
var myObj = {
    firstname: "Nic",
    lastname: "Raboy",
    website: "https://www.google.com"
}
//if you wanted to save into localStorage, serialize it
window.localStorage.set("saved", JSON.stringify(myObj));

//unserialize to get object
var myObj = JSON.parse(window.localStorage.get("saved"));
```

Section 22.2: Sharing data from one controller to another using service

We can create a `service` to **`set`** and **`get`** the data between the `controllers` and then inject that service in the

我们想要使用它的控制器函数中。

服务：

```
app.service('setGetData', function() {
  var data = '';
getData: function() { return data; },
    setData: function(requestData) { data = requestData; }
});
```

控制器 ：

```
app.controller('myCtrl1', ['setGetData',function(setGetData) {

  // 用于从一个控制器设置数据
  var data = 'Hello World !!';
  setGetData.setData(data);

}]);

app.controller('myCtrl2', ['setGetData',function(setGetData) {

  // 用于从另一个控制器获取数据
  var res = setGetData.getData();
  console.log(res); // Hello World !!

}]);
```

这里，我们可以看到myCtrl1用于设置数据，myCtrl2用于获取数据。所以，我们可以像这样在控制器之间共享数据。

elindoc.com

controller function where we want to use it.

Service :

```
app.service('setGetData', function() {
  var data = '';
    getData: function() { return data; },
    setData: function(requestData) { data = requestData; }
});
```

Controllers :

```
app.controller('myCtrl1', ['setGetData',function(setGetData) {

  // To set the data from the one controller
  var data = 'Hello World !!';
  setGetData.setData(data);

}]);

app.controller('myCtrl2', ['setGetData',function(setGetData) {

  // To get the data from the another controller
  var res = setGetData.getData();
  console.log(res); // Hello World !!

}]);
```

Here, we can see that `myCtrl1` is used for `setting` the data and `myCtrl2` is used for `getting` the data. So, we can share the data from one controller to another contrller like this.

第23章：表单验证

第23.1节：表单和输入状态

Angular 表单和输入具有多种状态，这些状态在验证内容时非常有用

输入状态

状态	描述
`$touched`	字段已被触摸
`$untouched`	字段未被触摸
`$pristine`	字段未被修改
`$dirty`	字段已被修改
`$valid`	字段内容有效
`$invalid`	字段内容无效

上述所有状态都是布尔属性，可以是真或假。

有了这些，向用户显示消息非常容易。

```
<form name="myForm" novalidate>
    <input name="myName" ng-model="myName" required>
    <span ng-show="myForm.myName.$touched && myForm.myName.$invalid">该名称无效</span>
</form>
```

这里，我们使用了ng-show指令，当用户修改了表单但表单无效时，向用户显示消息。

第23.2节：CSS类

Angular还根据表单和输入的状态提供了一些CSS类

类	描述
`ng-touched`	字段已被触摸
`ng-未触碰`	字段未被触碰
`ng-原始`	字段未被修改
`ng-已修改`	字段已被修改
`ng-有效`	字段有效
`ng-无效`	字段无效

您可以使用这些类为您的表单添加样式

```
input.ng-无效 {
  背景-颜色: 深红色;
}
input.ng-有效 {
  背景-颜色: 绿色;
}
```

第23.3节：基本表单验证

Angular的优势之一是客户端表单验证。

Chapter 23: Form Validation

Section 23.1: Form and Input States

Angular Forms and Inputs have various states that are useful when validating content

Input States

State	Description
`$touched`	Field has been touched
`$untouched`	Field has not been touched
`$pristine`	Field has not been modified
`$dirty`	Field has been modified
`$valid`	Field content is valid
`$invalid`	Field content is invalid

All of the above states are boolean properties and can be either true or false.

With these, it is very easy to display messages to a user.

```
<form name="myForm" novalidate>
    <input name="myName" ng-model="myName" required>
    <span ng-show="myForm.myName.$touched && myForm.myName.$invalid">This name is invalid</span>
</form>
```

Here, we are using the `ng-show` directive to display a message to a user if they've modified a form but it's invalid.

Section 23.2: CSS Classes

Angular also provides some CSS classes for forms and inputs depending on their state

Class	Description
`ng-touched`	Field has been touched
`ng-untouched`	Field has not been touched
`ng-pristine`	Field has not been modified
`ng-dirty`	Field has been modified
`ng-valid`	Field is valid
`ng-invalid`	Field is invalid

You can use these classes to add styles to your forms

```
input.ng-invalid {
  background-color: crimson;
}
input.ng-valid {
  background-color: green;
}
```

Section 23.3: Basic Form Validation

One of Angular's strength's is client-side form validation.

bellmdoc.com

处理传统表单输入并使用类似jQuery的查询处理可能既耗时又繁琐
。Angular允许您相对轻松地制作专业的*交互式*表单。

ng-model 指令为输入字段提供双向绑定，通常在表单元素上也会放置 novalidate 属性，以防止浏览器进行原生验证。

因此，一个简单的表单看起来像这样：

```
<form name="form" novalidate>
  <label name="email"> 你的邮箱 </label>
  <input type="email" name="email" ng-model="email" />
</form>
```

为了让 Angular 验证输入，使用的语法与普通的 input 元素完全相同，只是在作用域上绑定变量时需要添加 ng-model 属性。前面的例子展示了邮箱。要验证数字，语法如下：

```
<input type="number" name="postalcode" ng-model="zipcode" />
```

基本表单验证的最后步骤是使用 ng-submit 将表单提交连接到控制器上的提交函数，而不是允许默认的表单提交。这不是强制的，但通常会使用，因为输入变量已经在作用域上，因此可以在提交函数中使用。通常也建议给表单命名。这些更改后的语法如下：

```
<form name="signup_form" ng-submit="submitFunc()" novalidate>
  <label name="email"> 你的邮箱 </label>
  <input type="email" name="email" ng-model="email" />
  <button type="submit"> 注册 </button>
</form>
```

上述代码是可用的，但 Angular 还提供其他功能。

下一步是了解 Angular 使用 **ng-pristine**、**ng-dirty**、**ng-valid** 和 **ng-invalid** 来附加类属性以进行表单处理。在您的 CSS 中使用这些类将允许您为 **valid/invalid** 和 **pristine/dirty** 输入字段设置样式，从而在用户输入表单数据时改变其显示效果。

第23.4节：自定义表单验证

在某些情况下，基本验证是不够的。Angular 支持通过向 `$validators` 对象在 ngModelController 上：

```
angular.module('app', [])
  .directive('myValidator', function() {
    return {
      // 元素必须具有 ng-model 属性
      // 否则 $validators 无法工作
      require: 'ngModel',
      link: function(scope, elm, attrs, ctrl) {
        ctrl.$validators.myValidator = function(modelValue, viewValue) {
            // 使用您的自定义逻辑验证 viewValue
            var valid = (viewValue && viewValue.length > 0) || false;
            return valid;
        };
      }
    };
```


Dealing with traditional form inputs and having to use interrogative jQuery-style processing can be time-consuming and finicky. Angular allows you to produce professional *interactive* forms relatively easily.

The **ng-model** directive provides two-way binding with input fields and usually the **novalidate** attribute is also placed on the form element to prevent the browser from doing native validation.

Thus, a simple form would look like:

```
<form name="form" novalidate>
  <label name="email"> Your email </label>
  <input type="email" name="email" ng-model="email" />
</form>
```

For Angular to validate inputs, use exactly the same syntax as a regular *input* element, except for the addition of the **ng-model** attribute to specify which variable to bind to on the scope. Email is shown in the prior example. To validate a number, the syntax would be:

```
<input type="number" name="postalcode" ng-model="zipcode" />
```

The final steps to basic form validation are connecting to a form submit function on the controller using **ng-submit**, rather than allowing the default form submit to occur. This is not mandatory but it is usually used, as the input variables are already available on the scope and so available to your submit function. It is also usually good practice to give the form a name. These changes would result in the following syntax:

```
<form name="signup_form" ng-submit="submitFunc()" novalidate>
  <label name="email"> Your email </label>
  <input type="email" name="email" ng-model="email" />
  <button type="submit">Signup</button>
</form>
```

This above code is functional but there is other functionality that Angular provides.

The next step is to understand that Angular attaches class attributes using **ng-pristine**, **ng-dirty**, **ng-valid** and **ng-invalid** for form processing. Using these classes in your css will allow you to style **valid/invalid** and **pristine/dirty** input fields and so alter the presentation as the user is entering data into the form.

Section 23.4: Custom Form Validation

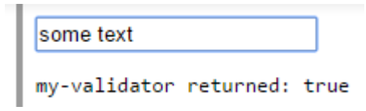
In some cases basic validation is not enough. Angular support custom validation adding validator functions to the `$validators` object on the `ngModelController`:

```
angular.module('app', [])
  .directive('myValidator', function() {
    return {
      // element must have ng-model attribute
      // or $validators does not work
      require: 'ngModel',
      link: function(scope, elm, attrs, ctrl) {
        ctrl.$validators.myValidator = function(modelValue, viewValue) {
            // validate viewValue with your custom logic
            var valid = (viewValue && viewValue.length > 0) || false;
            return valid;
        };
      }
    };
```

验证器被定义为一个需要ngModel的指令，因此要应用验证器，只需将自定义指令添加到输入表单控件中。

```
<form name="form">
  <input type="text"
         ng-model="model"
         name="model"
my-validator>
  <pre ng-bind="'my-validator returned: ' + form.model.$valid"></pre>
</form>
```

而且`my-validator`不必应用于原生表单控件。它可以是任何元素，只要其属性中包含`ng-model`。这在你有一些自定义构建的UI组件时非常有用。



第23.5节：异步验证器

异步验证器允许你使用后台（通过$http）验证表单信息。

当你需要访问服务器存储的信息，而这些信息由于各种原因无法在客户端获取时，就需要这类验证器，例如用户表和其他数据库信息。

要使用异步验证器，你需要访问input的ng-model，并为$asyncValidators属性定义回调函数。

示例：

以下示例检查所提供的名称是否已存在，后端将返回一个状态，如果名称已存在或未提供，则会拒绝该承诺。如果名称不存在，则会返回一个已解决的承诺。

```
ngModel.$asyncValidators.usernameValidate = function (name) {
     if (name) {
          return AuthenticationService.checkIfNameExists(name); // 返回一个promise
     } else {
          return $q.reject("该用户名已被占用！"); // 拒绝的promise
     }
};
```

现在每当输入的ng-model发生变化时，该函数将运行并返回一个带有结果的promise。

第23.6节：ngMessages

ngMessages用于增强视图中显示验证消息的样式。

传统方法

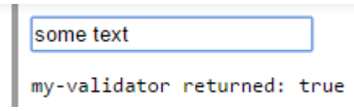
在`ngMessages`出现之前，我们通常使用Angular预定义指令`ng-class`来显示验证消息。这种方法比较杂乱且重复。

现在，通过使用ngMessages我们可以创建自定义消息。

The validator is defined as a directive that require `ngModel`, so to apply the validator just add the custom directive to the input form control.

```
<form name="form">
  <input type="text"
         ng-model="model"
         name="model"
         my-validator>
  <pre ng-bind="'my-validator returned: ' + form.model.$valid"></pre>
</form>
```

And `my-validator` doesn't have to be applied on native form control. It can be any elements, as long as it as `ng-model` in its attributes. This is useful when you have some custom build ui component.



Section 23.5: Async validators

Asynchronous validators allows you to validate form information against your backend (using $http).

These kind of validators are needed when you need to access server stored information you can't have on your client for various reasons, such as the users table and other database information.

To use async validators, you access the `ng-model` of your `input` and define callback functions for the `$asyncValidators` property.

Example:

The following example checks if a provided name already exists, the backend will return a status that will reject the promise if the name already exists or if it wasn't provided. If the name doesn't exist it will return a resolved promise.

```
ngModel.$asyncValidators.usernameValidate = function (name) {
     if (name) {
          return AuthenticationService.checkIfNameExists(name); // returns a promise
     } else {
          return $q.reject("This username is already taken!"); // rejected promise
     }
};
```

Now every time the `ng-model` of the input is changed, this function will run and return a promise with the result.

Section 23.6: ngMessages

`ngMessages` is used to enhanced the style for displaying validation messages in the view.

Traditional approach

Before `ngMessages`, we normally display the validation messages using Angular pre-defined directives `ng-class`.This approach was litter and a `repetitive` task.

Now, by using `ngMessages` we can create our own custom messages.

示例

HTML：

```
<form name="ngMessagesDemo">
  <input name="firstname" type="text" ng-model="firstname" required>
  <div ng-messages="ngMessagesDemo.firstname.$error">
    <div ng-message="required">名字为必填项。</div>
  </div>
</form>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.16/angular.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.16/angular-
messages.min.js"></script>
```

JS:

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

app.controller('mainCtrl', function ($scope) {
    $scope.firstname = "Rohit";
});
```

第23.7节：嵌套表单

有时为了在页面上逻辑分组控件和输入，嵌套表单是很有用的。
但是，HTML5表单不应嵌套。Angular提供了ng-form作为替代。

```
<form name="myForm" noValidate>
  <!-- 嵌套表单可以通过 'myForm.myNestedForm' 引用 -->
  <ng-form name="myNestedForm" noValidate>
    <input name="myInput1" ng-minlength="1" ng-model="input1" required />
    <input name="myInput2" ng-minlength="1" ng-model="input2" required />
  </ng-form>

  <!-- 在此显示嵌套子表单的错误 -->
  <div ng-messages="myForm.myNestedForm.$error">
    <!-- 注意，如果任一输入不满足最小长度，则会显示此信息 -->
    <div ng-message="minlength">长度至少为1</div>
  </div>
</form>

<!-- 表单状态 -->
<p>我的表单中是否有字段被编辑过？ {{myForm.$dirty}}</p>
<p>我的嵌套表单是否有效？ {{myForm.myNestedForm.$valid}}</p>
<p>myInput1 是否有效？ {{myForm.myNestedForm.myInput1.$valid}}</p>
```

表单的每个部分都会影响整体表单的状态。因此，如果其中一个输入项myInput1被编辑过且为$dirty，其所在的表单也会变为$dirty。这个状态会向上级表单传递，因此myNestedForm和myForm都会变为$dirty。

Example

HTML:

```
<form name="ngMessagesDemo">
  <input name="firstname" type="text" ng-model="firstname" required>
  <div ng-messages="ngMessagesDemo.firstname.$error">
    <div ng-message="required">Firstname is required.</div>
  </div>
</form>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.16/angular.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.16/angular-
messages.min.js"></script>
```

JS:

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

app.controller('mainCtrl', function ($scope) {
    $scope.firstname = "Rohit";
});
```

Section 23.7: Nested Forms

Sometimes it is desirable to nest forms for the purpose of grouping controls and inputs logically on the page. However, HTML5 forms should not be nested. Angular supplies `ng-form` instead.

```
<form name="myForm" noValidate>
  <!-- nested form can be referenced via 'myForm.myNestedForm' -->
  <ng-form name="myNestedForm" noValidate>
    <input name="myInput1" ng-minlength="1" ng-model="input1" required />
    <input name="myInput2" ng-minlength="1" ng-model="input2" required />
  </ng-form>

  <!-- show errors for the nested subform here -->
  <div ng-messages="myForm.myNestedForm.$error">
    <!-- note that this will show if either input does not meet the minimum -->
    <div ng-message="minlength">Length is not at least 1</div>
  </div>
</form>

<!-- status of the form -->
<p>Has any field on my form been edited? {{myForm.$dirty}}</p>
<p>Is my nested form valid? {{myForm.myNestedForm.$valid}}</p>
<p>Is myInput1 valid? {{myForm.myNestedForm.myInput1.$valid}}</p>
```

Each part of the form contributes to the overall form's state. Therefore, if one of the inputs `myInput1` has been edited and is `$dirty`, its containing form will also be `$dirty`. This cascades to each containing form, so both `myNestedForm` and `myForm` will be `$dirty`.

第24章：使用 ngRoute 进行路由

第24.1节：基本示例

本示例展示了如何设置一个包含3个路由的小型应用，每个路由都有自己的视图和控制器，使用 controllerAs 语法。

我们在 angular .config 函数中配置路由器

1. 我们将 $routeProvider 注入到 .config
2. 我们在 .when 方法中使用路由定义对象定义路由名称。
3. 我们为 .when 方法提供一个对象，指定我们的 template 或 templateUrl、controller 和 controllerAs

app.js

```
angular.module('myApp', ['ngRoute'])
  .controller('controllerOne', function() {
    this.message = '来自 Controller One 的你好，世界！';
  })
.controller('controllerTwo', function() {
    this.message = '来自 Controller Two 的你好，世界！';
  })
.controller('controllerThree', function() {
    this.message = '来自 Controller Three 的你好，世界！';
  })
.config(function($routeProvider) {
    $routeProvider
.when('/one', {
      templateUrl: 'view-one.html',
      controller: 'controllerOne',
      controllerAs: 'ctrlOne'
    })
.when('/two', {
      templateUrl: 'view-two.html',
      controller: 'controllerTwo',
      controllerAs: 'ctrlTwo'
    })
.when('/three', {
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    // 如果没有其他路由匹配，则重定向到这里
.otherwise({
      redirectTo: '/one'
    });
  });
```

然后在我们的HTML中，我们使用带有<a>元素和 href属性来定义导航，对于路由名称为helloRoute的路由，我们将其写成我的路由

我们还为视图提供了一个容器和指令ng-view来注入我们的路由。

index.html

```
<div ng-app="myApp">
  <nav>
```


Chapter 24: Routing using ngRoute

Section 24.1: Basic example

This example shows setting up a small application with 3 routes, each with it's own view and controller, using the `controllerAs` syntax.

We configure our router at the angular `.config` function

1. We inject `$routeProvider` into `.config`
2. We define our route names at the `.when` method with a route definition object.
3. We supply the `.when` method with an object specifying our `template` or `templateUrl`, `controller` and `controllerAs`

app.js

```
angular.module('myApp', ['ngRoute'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', function() {
    this.message = 'Hello world from Controller Three!';
  })
  .config(function($routeProvider) {
    $routeProvider
    .when('/one', {
      templateUrl: 'view-one.html',
      controller: 'controllerOne',
      controllerAs: 'ctrlOne'
    })
    .when('/two', {
      templateUrl: 'view-two.html',
      controller: 'controllerTwo',
      controllerAs: 'ctrlTwo'
    })
    .when('/three', {
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    // redirect to here if no other routes match
    .otherwise({
      redirectTo: '/one'
    });
  });
```

Then in our HTML we define our navigation using **<a>** elements with `href`, for a route name of `helloRoute` we will route as **<a** `href="#/helloRoute"`**>**`My route`****

We also provide our view with a container and the directive `ng-view` to inject our routes.

index.html

```
<div ng-app="myApp">
  <nav>
```


elindo.com

```
    <!-- 用于切换路由的链接 -->
    <a href="#/one">视图一</a>
    <a href="#/two">视图二</a>
    <a href="#/three">视图三</a>
  </nav>
  <!-- 视图将被注入此处 -->
  <div ng-view></div>
  <!-- 模板可以存放在普通的 HTML 文件中 -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

第24.2节：为单个路由定义自定义行为

为单个路由定义自定义行为的最简单方法相当容易。

在此示例中，我们用它来验证用户身份：

1）routes.js：为任何所需路由创建一个新属性（如 requireAuth）

```
angular.module('yourApp').config(['$routeProvider', function($routeProvider) {
    $routeProvider
.当('/home', {
            templateUrl: 'templates/home.html',
            requireAuth: true
        })
.当('/login', {
            templateUrl: 'templates/login.html',
        })
.否则({
            redirectTo: '/home'
        });
}])
```

2）在一个不绑定到 ng-view 内元素的顶级控制器中（以避免与 angular $routeProvider），检查 newUrl 是否具有 requireAuth 属性并据此采取相应操作

```
angular.module('YourApp').controller('YourController', ['$scope', 'session', '$location',
    function($scope, session, $location) {

$scope.$on('$routeChangeStart', function(angularEvent, newUrl) {

            if (newUrl.requireAuth && !session.user) {
                // 用户未认证
$location.path("/login");
            }

        });
    }
]);
```



```
    <!-- links to switch routes -->
    <a href="#/one">View One</a>
    <a href="#/two">View Two</a>
    <a href="#/three">View Three</a>
  </nav>
  <!-- views will be injected here -->
  <div ng-view></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

Section 24.2: Defining custom behavior for individual routes

The simplest manner of defining custom behavior for individual routes would be fairly easy.

In this example we use it to authenticate a user :

1) `routes.js`: create a new property (like `requireAuth`) for any desired route

```
angular.module('yourApp').config(['$routeProvider', function($routeProvider) {
    $routeProvider
        .when('/home', {
            templateUrl: 'templates/home.html',
            requireAuth: true
        })
        .when('/login', {
            templateUrl: 'templates/login.html',
        })
        .otherwise({
            redirectTo: '/home'
        });
}])
```

2) In a top-tier controller that isn't bound to an element inside the `ng-view` (to avoid conflict with angular `$routeProvider`), check if the `newUrl` has the `requireAuth` property and act accordingly

```
angular.module('YourApp').controller('YourController', ['$scope', 'session', '$location',
    function($scope, session, $location) {

        $scope.$on('$routeChangeStart', function(angularEvent, newUrl) {

            if (newUrl.requireAuth && !session.user) {
                // User isn't authenticated
                $location.path("/login");
            }

        });
    }
]);
```

第24.3节：路由参数示例

此示例扩展了基本示例，通过路由传递参数以便在控制器中使用

为此我们需要：

1. 在路由名称中配置参数的位置和名称
2. 在控制器中注入`$routeParams`服务

app.js

```
angular.module('myApp', ['ngRoute'])
  .controller('controllerOne', function() {
    this.message = '来自 Controller One 的你好，世界！';
  })
.controller('controllerTwo', function() {
    this.message = '来自 Controller Two 的你好，世界！';
  })
.controller('controllerThree', ['$routeParams', function($routeParams) {
    var routeParam = $routeParams.paramName

    if ($routeParams.message) {
        // 如果存在名为'message'的参数，则显示其值作为消息
        this.message = $routeParams.message;
    } else {
        // 如果不存在，则显示默认消息
        this.message = '来自控制器三的问候！';
    }
  }])
.config(function($routeProvider) {
    $routeProvider
.when('/one', {
      templateUrl: 'view-one.html',
      controller: 'controllerOne',
      controllerAs: 'ctrlOne'
    })
.when('/two', {
      templateUrl: 'view-two.html',
      controller: 'controllerTwo',
      controllerAs: 'ctrlTwo'
    })
.when('/three', {
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
.当('/three/:message', { // 我们将通过此路由传递一个名为 'message' 的参数
      templateUrl: 'view-three.html',
controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    // 如果没有其他路由匹配，则重定向到这里
.otherwise({
      redirectTo: '/one'
    });
  });
```

然后，在不对我们的模板做任何更改的情况下，仅添加一个带有自定义消息的新链接，我们就可以在视图中看到新的自定义消息。

Section 24.3: Route parameters example

This example extends the basic example passing parameters in the route in order to use them in the controller

To do so we need to:

1. Configure the parameter position and name in the route name
2. Inject `$routeParams` service in our Controller

app.js

```
angular.module('myApp', ['ngRoute'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', ['$routeParams', function($routeParams) {
    var routeParam = $routeParams.paramName

    if ($routeParams.message) {
        // If a param called 'message' exists, we show it's value as the message
        this.message = $routeParams.message;
    } else {
        // If it doesn't exist, we show a default message
        this.message = 'Hello world from Controller Three!';
    }
  }])
  .config(function($routeProvider) {
    $routeProvider
    .when('/one', {
      templateUrl: 'view-one.html',
      controller: 'controllerOne',
      controllerAs: 'ctrlOne'
    })
    .when('/two', {
      templateUrl: 'view-two.html',
      controller: 'controllerTwo',
      controllerAs: 'ctrlTwo'
    })
    .when('/three', {
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    .when('/three/:message', { // We will pass a param called 'message' with this route
      templateUrl: 'view-three.html',
      controller: 'controllerThree',
      controllerAs: 'ctrlThree'
    })
    // redirect to here if no other routes match
    .otherwise({
      redirectTo: '/one'
    });
  });
```

Then, withoud making any changes in our templates, only adding a new link with custom message, we can see the new custom message in our view.

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- 用于切换路由的链接 -->
    <a href="#/one">视图一</a>
    <a href="#/two">视图二</a>
    <a href="#/three">视图三</a>
    <!-- 新链接，带自定义消息 -->
    <a href="#/three/This-is-a-message">视图三，带有“This-is-a-message”自定义消息</a>
  </nav>
  <!-- 视图将被注入此处 -->
  <div ng-view></div>
  <!-- 模板可以存放在普通的 HTML 文件中 -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

belindoc.com

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- links to switch routes -->
    <a href="#/one">View One</a>
    <a href="#/two">View Two</a>
    <a href="#/three">View Three</a>
    <!-- New link with custom message -->
    <a href="#/three/This-is-a-message">View Three with "This-is-a-message" custom message</a>
  </nav>
  <!-- views will be injected here -->
  <div ng-view></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

第25章：ng-class 指令

第25.1节：三种类型的 ng-class 表达式

Angular 支持在 ng-class 指令中使用三种类型的表达式。

1. 字符串

```
<span ng-class="MyClass">示例文本</span>
```

指定一个求值为字符串的表达式告诉 Angular 将其视为 $scope 变量。Angular 会检查 $scope 并查找名为 "MyClass" 的变量。"MyClass" 中包含的任何文本都将成为应用于该 的实际类名。你可以通过空格分隔来指定多个类。

在你的控制器中，可能有如下定义：

```
$scope.MyClass = "bold-red deleted error";
```

Angular 会求值表达式 MyClass 并找到 $scope 中的定义。它会将三个类 "bold-red"、"deleted" 和 "error" 应用于该 元素。

以这种方式指定类可以让你轻松地在控制器中更改类定义。例如，你可能需要根据其他用户交互或从服务器加载的新数据更改类。此外，如果你有很多表达式需要求值，可以在一个函数中定义最终的类列表并存储在 $scope 变量中。这比试图在 HTML 模板的 ng-class 属性中塞入许多求值要容易得多。

2. 对象

这是使用 ng-class 定义类最常用的方式，因为它可以轻松地让你指定决定使用哪个类的求值。

指定一个包含键值对的对象。键是类名，当值（一个条件）计算为真时将应用该类名。

```
<style>
.red { 颜色: 红色; 字体加粗; }
    .blue { 颜色: 蓝色; }
.green { 颜色: 绿色; }
.highlighted { 背景色: 黄色; 颜色: 黑色; }
</style>

<span ng-class="{ red: ShowRed, blue: ShowBlue, green: ShowGreen, highlighted: IsHighlighted
}">示例文本</span>

<div>红色: <input type="checkbox" ng-model="ShowRed"></div>
<div>绿色: <input type="checkbox" ng-model="ShowGreen"></div>
<div>蓝色: <input type="checkbox" ng-model="ShowBlue"></div>
<div>高亮: <input type="checkbox" ng-model="IsHighlighted"></div>
```

3. 数组

一个求值为数组的表达式允许你使用字符串（见上文#1）和条件对象（见上文#2）的组合。

Chapter 25: ng-class directive

Section 25.1: Three types of ng-class expressions

Angular supports three types of expressions in the `ng-class` directive.

1. String

```
<span ng-class="MyClass">Sample Text</span>
```

Specifying an expression that evaluates to a string tells Angular to treat it as a $scope variable. Angular will check the $scope and look for a variable called "MyClass". Whatever text is contained in "MyClass" will become the actual class name that is applied to this ****. You can specify multiple classes by separating each class with a space.

In your controller, you may have a definition that looks like this:

```
$scope.MyClass = "bold-red deleted error";
```

Angular will evaluate the expression `MyClass` and find the $scope definition. It will apply the three classes "bold-red", "deleted", and "error" to the **** element.

Specifying classes this way lets you easily change the class definitions in your controller. For example, you may need to change the class based on other user interactions or new data that is loaded from the server. Also, if you have a lot of expressions to evaluate, you can do so in a function that defines the final list of classes in a `$scope` variable. This can be easier than trying to squeeze many evaluations into the `ng-class` attribute in your HTML template.

2. Object

This is the most commonly-used way of defining classes using `ng-class` because it easily lets you specify evaluations that determine which class to use.

Specify an object containing key-value pairs. The key is the class name that will be applied if the value (a conditional) evaluates as true.

```
<style>
    .red { color: red; font-weight: bold; }
    .blue { color: blue; }
    .green { color: green; }
    .highlighted { background-color: yellow; color: black; }
</style>

<span ng-class="{ red: ShowRed, blue: ShowBlue, green: ShowGreen, highlighted: IsHighlighted
}">Sample Text</span>

<div>Red: <input type="checkbox" ng-model="ShowRed"></div>
<div>Green: <input type="checkbox" ng-model="ShowGreen"></div>
<div>Blue: <input type="checkbox" ng-model="ShowBlue"></div>
<div>Highlight: <input type="checkbox" ng-model="IsHighlighted"></div>
```

3. Array

An expression that evaluates to an array lets you use a combination of **strings** (see #1 above) and **conditional objects** (#2 above).

```
<style>
.bold { 字体加粗; }
    .strike { 文字装饰: 删除线; }
    .orange { 颜色: 橙色; }
</style>

<p ng-class="[ UserStyle, {orange: warning} ]">两种表达式类型的数组</p>
<input ng-model="UserStyle" placeholder="输入 'bold' 和/或 'strike'"><br>
<label><input type="checkbox" ng-model="warning"> 警告 (应用 "orange" 类)</label>
```

这会创建一个绑定到作用域变量UserStyle的文本输入框，允许用户输入任意类名。
这些类名会随着用户输入动态应用到<p>元素上。此外，用户可以点击绑定到warning作用域变量的复选框。这也会动态应用到<p>元素上。

belindoc.com


```
<style>
    .bold { font-weight: bold; }
    .strike { text-decoration: line-through; }
    .orange { color: orange; }
</style>

<p ng-class="[ UserStyle, {orange: warning} ]">Array of Both Expression Types</p>
<input ng-model="UserStyle" placeholder="Type 'bold' and/or 'strike'"><br>
<label><input type="checkbox" ng-model="warning"> warning (apply "orange" class)</label>
```

This creates a text input field bound to the scope variable `UserStyle` which lets the user type in any class name(s). These will be dynamically applied to the **<p>** element as the user types. Also, the user can click on the checkbox that is data-bound to the `warning` scope variable. This will also be dynamically applied to the **<p>** element.

第26章：ng-repeat

变量	详情
`$index`	*number* 重复元素的迭代偏移量（0..length-1）
`$first`	*boolean* 如果重复元素是迭代器中的第一个，则为true。
`$middle`	boolean 如果重复元素位于迭代器的第一个和最后一个之间，则为true。
`$last`	boolean 如果重复元素是迭代器中的最后一个，则为true。
`$even`	boolean 如果迭代位置$index是偶数，则为true（否则为false）。
`$odd`	boolean 如果迭代器位置$index是奇数则为true（否则为false）。

`ngRepeat` 指令为集合中的每个项目实例化一个模板。集合必须是数组或对象。每个模板实例都有自己的作用域，其中给定的循环变量被设置为当前集合项目，且 `$index` 被设置为项目的索引或键。

第26.1节：ng-repeat-start + ng-repeat-end

AngularJS 1.2 ng-repeat 使用 ng-repeat-start 和 ng-repeat-end 来处理多个元素：

```
// 表格项目
$scope.tableItems = [
    {
row1: '项目 1：第 1 行',
        row2: '项目 1：第 2 行'
    },
    {
row1: '项目 2：第 1 行',
        row2: '项目 2：第 2 行'
    }
];

// 模板
<table>
    <th>
        <td>项目</td>
    </th>
    <tr ng-repeat-start="item in tableItems">
        <td ng-bind="item.row1"></td>
    </tr>
    <tr ng-repeat-end>
        <td ng-bind="item.row2"></td>
    </tr>
</table>
```

输出：

项目
项目 1：第 1 行
项目 1：第 2 行
项目 2：第 1 行
项目 2：第 2 行

第 26.2 节：遍历对象属性

```
<div ng-repeat="(key, value) in myObj"> ... </div>
```


Chapter 26: ng-repeat

Variable	Details
`$index`	*number* iterator offset of the repeated element (0..length-1)
`$first`	*boolean* true if the repeated element is first in the iterator.
`$middle`	*boolean* true if the repeated element is between the first and last in the iterator.
`$last`	*boolean* true if the repeated element is last in the iterator.
`$even`	*boolean* true if the iterator position `$index` is even (otherwise false).
`$odd`	*boolean* true if the iterator position `$index` is odd (otherwise false).

The `ngRepeat` directive instantiates a template once per item from a collection. The collection must be an array or an object. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and `$index` is set to the item index or key.

Section 26.1: ng-repeat-start + ng-repeat-end

AngularJS 1.2 `ng-repeat` handle multiple elements with `ng-repeat-start` and `ng-repeat-end`:

```
// table items
$scope.tableItems = [
    {
        row1: 'Item 1: Row 1',
        row2: 'Item 1: Row 2'
    },
    {
        row1: 'Item 2: Row 1',
        row2: 'Item 2: Row 2'
    }
];

// template
<table>
    <th>
        <td>Items</td>
    </th>
    <tr ng-repeat-start="item in tableItems">
        <td ng-bind="item.row1"></td>
    </tr>
    <tr ng-repeat-end>
        <td ng-bind="item.row2"></td>
    </tr>
</table>
```

Output:

Items
Item 1: Row 1
Item 1: Row 2
Item 2: Row 1
Item 2: Row 2

Section 26.2: Iterating over object properties

```
<div ng-repeat="(key, value) in myObj"> ... </div>
```

例如

```
<div ng-repeat="n in [42, 42, 43, 43]">
  {{n}}
</div>
```

第26.3节：跟踪与重复项

ngRepeat 使用 $watchCollection 来检测集合的变化。当发生变化时，ngRepeat 会对DOM进行相应的更改：

- 当添加一个项目时，会向DOM中添加该模板的新实例。
- 当移除一个项目时，会从DOM中移除其模板实例。
- 当项目重新排序时，其对应的模板也会在DOM中重新排序。

重复项

- 对于可能包含重复值的列表，使用track by。
- 使用track by还能显著加快列表的变化速度。
- 如果在这种情况下不使用track by，会出现错误：[ngRepeat:dupes]

```
$scope.numbers = ['1','1','2','3','4'];

<ul>
  <li ng-repeat="n in numbers track by $index">
    {{n}}
  </li>
</ul>
```

belindoc.com

For example

```
<div ng-repeat="n in [42, 42, 43, 43]">
  {{n}}
</div>
```

Section 26.3: Tracking and Duplicates

`ngRepeat` uses $watchCollection to detect changes in the collection. When a change happens, `ngRepeat` then makes the corresponding changes to the DOM:

- When an item is added, a new instance of the template is added to the DOM.
- When an item is removed, its template instance is removed from the DOM.
- When items are reordered, their respective templates are reordered in the DOM.

Duplicates

- `track by` for any list that may include duplicate values.
- `track by` also speeds up list changes significantly.
- If you don't use `track by` in this case, you get the error: `[ngRepeat:dupes]`

```
$scope.numbers = ['1','1','2','3','4'];

<ul>
  <li ng-repeat="n in numbers track by $index">
    {{n}}
  </li>
</ul>
```

第27章：ng-style

'ngStyle'指令允许你有条件地设置HTML元素的CSS样式。就像在非AngularJS项目中我们可以使用HTML元素的style属性一样，在AngularJS中我们可以使用ng-style根据某些布尔条件应用样式。

第27.1节：ng-style的使用

下面的示例根据"status"参数改变图片的不透明度。

```
<img class="img-responsive"  ng-src="{{imagesrc}}"
   ng-style="{'opacity' : (status == 2) ? 1 : 0.5}">
```

belindoc.com

Chapter 27: ng-style

The 'ngStyle' directive allows you to set CSS style on an HTML element conditionally. Much like how we could use *style* attribute on HTML element in non-AngularJS projects, we can use `ng-style` in angularjs do apply styles based on some boolean condition.

Section 27.1: Use of ng-style

Below example changes the opacity of the image based on the "status" parameter.

```
<img class="img-responsive"  ng-src="{{imagesrc}}"
   ng-style="{'opacity' : (status == 2) ? 1 : 0.5}">
```

第28章：ng-view

ng-view是Angular内置的指令之一，用作切换视图的容器。{info} ngRoute不再是基础angular.js文件的一部分，因此你需要在基础Angular JavaScript文件之后引入angular-route.js文件。我们可以使用$routeProvider的“when”函数来配置路由。首先需要指定路由，然后在第二个参数中提供一个包含templateUrl属性和controller属性的对象。

第28.1节：注册导航

1. 我们在应用程序中注入模块

```
var Registration=angular.module("myApp",["ngRoute"]);
```

2. 现在我们使用来自 "ngRoute" 的 $routeProvider

```
Registration.config(function($routeProvider) {

});
```

3. 最后我们集成路由，定义应用程序中的 "/add" 路由，当应用程序访问 "/add" 时，它会跳转到 regi.htm

```
Registration.config(function($routeProvider) {
    $routeProvider
.when("/add", {
        templateUrl : "regi.htm"
    })
});
```

第28.2节：ng-view

ng-view 是一个与 $route 一起使用的指令，用于在主页面布局中渲染部分视图。在此示例中，Index.html 是我们的主文件，当用户访问 "/" 路由时，templateURL home.html 将在 Index.html 中渲染，位置是在 ng-view 所在处。

```
angular.module('ngApp', ['ngRoute'])

.config(function($routeProvider){
  $routeProvider.when("/",
    {
templateUrl: "home.html",
      controller: "homeCtrl"
    }
  );
});

angular.module('ngApp').controller('homeCtrl',['$scope', function($scope) {
  $scope.welcome= "欢迎来到stackoverflow！";
}]);

//Index.html
<body ng-app="ngApp">
    <div ng-view></div>
</body>

//Home 模板 URL 或 home.html
```

Belindoc.com

Chapter 28: ng-view

ng-view is one of in-build directive that angular uses as a container to switch between views. {info} ngRoute is no longer a part of the base angular.js file, so you'll need to include the angular-route.js file after your the base angular javascript file. We can configure a route by using the “when” function of the $routeProvider. We need to first specify the route, then in a second parameter provide an object with a templateUrl property and a controller property.

Section 28.1: Registration navigation

1. We injecting the module in the application

```
var Registration=angular.module("myApp",["ngRoute"]);
```

2. now we use $routeProvider from "ngRoute"

```
    Registration.config(function($routeProvider) {

});
```

3. finally we integrating the route, we define "/add" routing to the application in case application get "/add" it divert to regi.htm

```
Registration.config(function($routeProvider) {
    $routeProvider
    .when("/add", {
        templateUrl : "regi.htm"
    })
});
```

Section 28.2: ng-view

`ng-view` is a directive used with `$route` to render a partial view in the main page layout. Here in this example, Index.html is our main file and when user lands on "/" route the templateURL home.html will be rendered in Index.html where `ng-view` is mentioned.

```
angular.module('ngApp', ['ngRoute'])

.config(function($routeProvider){
  $routeProvider.when("/",
    {
      templateUrl: "home.html",
      controller: "homeCtrl"
    }
  );
});

angular.module('ngApp').controller('homeCtrl',['$scope', function($scope) {
  $scope.welcome= "Welcome to stackoverflow!";
}]);

//Index.html
<body ng-app="ngApp">
    <div ng-view></div>
</body>

//Home Template URL or home.html
```

```
<div><h2>{{welcome}}</h2></div>
```

belindoc.com


```
<div><h2>{{welcome}}</h2></div>
```

第29章：AngularJS 绑定选项 (`=`, `@`, `&` 等)

第29.1节：绑定可选属性

```
bindings: {
   mandatory: '='
   optional: '=?',
   foo: '=?bar'
}
```

可选属性应标记为问号：`=?` 或 `=?bar`。这是对 (`$compile:nonassign`) 异常的保护。

第29.2节：@ 单向绑定，属性绑定

传入一个字面值（非对象），例如字符串或数字。

子作用域获得自己的值，如果它更新该值，父作用域仍保持自己的旧值（子作用域不能修改父作用域的值）。当父作用域的值改变时，子作用域的值也会改变。所有插值表达式在每次digest调用时都会出现，而不仅仅是在指令创建时。

```
<one-way text="简单文本." <!-- '简单文本.' -->
        simple-value="123" <!-- '123' 注意，实际上是一个字符串对象。 -->
        interpolated-value="{{parentScopeValue}}" <!-- 来自父作用域的某个值。你不能
修改父作用域的值，只能修改子作用域的值。注意，实际上是一个字符串对象。 -->
interpolated-function-value="{{parentScopeFunction()}}" <!-- 执行父作用域的
函数并获取值。 -->

        <!-- 非预期用法。 -->
object-item="{{objectItem}}" <!-- 将对象|日期转换为字符串。结果可能是：
'{"a":5,"b":"text"}'。
function-item="{{parentScopeFunction}}"> <!-- 将是一个空字符串。 -->
</one-way>
```

第29.3节：= 双向绑定

通过引用传递一个值，您希望在两个作用域之间共享该值并从两个作用域中操作它。您不应使用{{...}}进行插值。

```
<two-way text="'简单文本.'" <!-- '简单文本.' -->simple-value="123" <!
        -- 123 注意，现在实际上是一个数字。-->interpolated-value="parentScopeValue"
        <!-- 来自父作用域的某个值。您可以在一个作用域中更改它，并在另一个作用域中获得更新后的值。-->object-item="obje
ctItem" <!-- 来自父作用域的某个对象。您可以在一个作用域中更改
        对象属性，并在另一个作用域中获得更新后的属性。-->          <!-- 意外用法。-->


        interpolated-function-value="parentScopeFunction()" <!-- 会引发错误。-->function-item="incrementInt
        erpolated"> <!-- 通过引用传递函数，您可以在子作用域中使用它。-->

</two-way>
```

通过引用传递函数是个坏主意：为了允许作用域更改函数定义，并且会创建两个不必要的观察者，您需要尽量减少观察者的数量。

Chapter 29: AngularJS bindings options (`=`, `@`, `&` etc.)

Section 29.1: Bind optional attribute

```
bindings: {
   mandatory: '='
   optional: '=?',
   foo: '=?bar'
}
```

Optional attributes should be marked with question mark: `=?` or `=?bar`. It is protection for (`$compile:nonassign`) exception.

Section 29.2: @ one-way binding, attribute binding

Pass in a literal value (not an object), such as a string or number.

Child scope gets his own value, if it updates the value, parent scope has his own old value (child scope can't modify the parens scope value). When parent scope value is changed, child scope value will be changed as well. All interpolations appears every time on digest call, not only on directive creation.

```
<one-way text="Simple text." <!-- 'Simple text.' -->
        simple-value="123" <!-- '123' Note, is actually a string object. -->
        interpolated-value="{{parentScopeValue}}" <!-- Some value from parent scope. You can't
change parent scope value, only child scope value. Note, is actually a string object. -->
        interpolated-function-value="{{parentScopeFunction()}}" <!-- Executes parent scope
function and takes a value. -->

        <!-- Unexpected usage. -->
        object-item="{{objectItem}}" <!-- Converts object|date to string. Result might be:
'{"a":5,"b":"text"}'. -->
        function-item="{{parentScopeFunction}}"> <!-- Will be an empty string. -->
</one-way>
```

Section 29.3: = two-way binding

Passing in a value by reference, you want to share the value between both scopes and manipulate them from both scopes. You should not use {{...}} for interpolation.

```
<two-way text="'Simple text.'" <!-- 'Simple text.' -->
        simple-value="123" <!-- 123 Note, is actually a number now. -->
        interpolated-value="parentScopeValue" <!-- Some value from parent scope. You may change it
in one scope and have updated value in another. -->
        object-item="objectItem" <!-- Some object from parent scope. You may change object
properties in one scope and have updated properties in another. -->

        <!-- Unexpected usage. -->
        interpolated-function-value="parentScopeFunction()" <!-- Will raise an error. -->
        function-item="incrementInterpolated"> <!-- Pass the function by reference and you may use
it in child scope. -->
</two-way>
```

Passing function by reference is a bad idea: to allow scope to change the definition of a function, and two unnecessary watcher will be created, you need to minimize watchers count.

第29.4节：& 函数绑定，表达式绑定

将方法传递给指令。它提供了一种在父作用域上下文中执行表达式的方式。
方法将在父作用域中执行，您可以从子作用域传递一些参数过去。
您不应使用{{...}}进行插值。当您在指令中使用&时，它会生成一个函数，该函数返回针对父作用域计算的表达式的值（这与=不同，=只是传递一个引用）。

```
<expression-binding interpolated-function-value="incrementInterpolated(param)" <!--
interpolatedFunctionValue({param: '嘿'}) 将使用参数调用传入的函数。-->function-item="incrementInterpolated"
                    <!-- functionItem({param: '嘿'})() 将调用传入的函数，但无法设置参数。-->text="'简单文本.'"  <!-- tex
t() == '简单文本.'-->

                    simple-value="123" <!-- simpleValue() == 123 -->
                    interpolated-value="parentScopeValue" <!-- interpolatedValue() == 来自父作用域的某个值
-->
                    object-item="objectItem"> <!-- objectItem() == 来自父作用域的对象项。 -
->
<expression-binding></expression-binding>
```

所有参数将被封装到函数中。

第29.5节：通过简单样本实现的可用绑定

```
angular.component("SampleComponent", {
  bindings: {
     title: '@',
     movies: '<',
     reservation: "=",
     processReservation: "&"
  }
});
```

这里是所有的绑定元素。

@ 表示我们需要一个非常基础的绑定，从父作用域到子作用域，不带任何观察者，
以任何方式都不例外。父作用域中的每次更新都会保留在父作用域中，子作用域中的任何更新
都不会传递给父作用域。

< 表示单向绑定。父作用域中的更新会传播到子作用域，但子作用域中的任何更新都不会应用到父作用域。

= 已经被称为双向绑定。父作用域的每次更新都会应用到子作用域，
每个子作用域的更新也会应用到父作用域。

& 现在用于输出绑定。根据组件文档，它应该用于引用父作用域的方法。无需操作子作用域，只需调用带有更新数据的父方法！

Section 29.4: & function binding, expression binding

Pass a method into a directive. It provides a way to execute an expression in the context of the parent scope.
Method will be executed in the scope of the parent, you may pass some parameters from the child scope there.
You should not use {{...}} for interpolation. When you use & in a directive, it generates a function that returns the value of the expression evaluated against the parent scope (not the same as = where you just pass a reference).

```
<expression-binding interpolated-function-value="incrementInterpolated(param)" <!--
interpolatedFunctionValue({param: 'Hey'}) will call passed function with an argument. -->
                    function-item="incrementInterpolated" <!-- functionItem({param: 'Hey'})() will
call passed function, but with no possibility set up a parameter. -->
                    text="'Simple text.'"  <!-- text() == 'Simple text.'-->
                    simple-value="123" <!-- simpleValue() == 123 -->
                    interpolated-value="parentScopeValue" <!-- interpolatedValue() == Some value
from parent scope. -->
                    object-item="objectItem"> <!-- objectItem() == Object item from parent scope. -
->
</expression-binding>
```

All parameters will be wrapped into functions.

Section 29.5: Available binding through a simple sample

```
angular.component("SampleComponent", {
  bindings: {
     title: '@',
     movies: '<',
     reservation: "=",
     processReservation: "&"
  }
});
```

Here we have all binding elements.

@ indicates that we need a very **basic binding**, from the parent scope to the children scope, without any watcher, in any way. Every update in the parent scope would stay in the parent scope, and any update on the child scope would not be communicated to the parent scope.

< indicates a **one way binding**. Updates in the parent scope would be propagated to the children scope, but any update in the children scope would not be applied to the parent scope.

= is already known as a two-way binding. Every update on the parent scope would be applied on the children ones, and every child update would be applied to the parent scope.

& is now used for an output binding. According to the component documentation, it should be used to reference the parent scope method. Instead of manipulating the children scope, just call the parent method with the updated data!

第30章：提供者

第30.1节：提供者

提供者 可用于配置阶段和运行阶段。

> 提供者模式在语法上定义为实现了$get方法的自定义类型。
>
> 只有当你想暴露一个应用范围内的API，并且该配置必须在应用启动前完成时，才应使用提供者模式。这通常只对可重用服务有意义，因为这些服务的行为可能需要在不同应用间略有差异。

```
angular.module('app',[])
  .provider('endpointProvider', function() {
    var uri = 'n/a';

    this.set = function(value) {
      uri = value;
    };

    this.$get = function() {
      return {
        get: function() {
          return uri;
        }
      };
    };
  })
.config(function(endpointProviderProvider) {
    endpointProviderProvider.set('http://some.rest.endpoint');
  })
.controller('主控制器', function(endpointProvider) {
    var vm = this;
vm.endpoint = endpointProvider.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

如果没有config阶段，结果将是

> endpoint = 不适用

第30.2节：工厂

工厂可在运行阶段使用。

Chapter 30: Providers

Section 30.1: Provider

`Provider` is available both in configuration and run phases.

> The Provider recipe is syntactically defined as a custom type that implements a `$get` method.
>
> You should use the Provider recipe only when you want to expose an API for application-wide configuration that must be made before the application starts. This is usually interesting only for reusable services whose behavior might need to vary slightly between applications.

```
angular.module('app',[])
  .provider('endpointProvider', function() {
    var uri = 'n/a';

    this.set = function(value) {
      uri = value;
    };

    this.$get = function() {
      return {
        get: function() {
          return uri;
        }
      };
    };
  })
  .config(function(endpointProviderProvider) {
    endpointProviderProvider.set('http://some.rest.endpoint');
  })
  .controller('MainCtrl', function(endpointProvider) {
    var vm = this;
    vm.endpoint = endpointProvider.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Without `config` phase result would be

> endpoint = n/a

Section 30.2: Factory

`Factory` is available in run phase.

> 工厂方法通过一个带有零个或多个参数的函数构造一个新的服务（这些参数是对其他服务的依赖）。该函数的返回值即为此方法创建的服务实例。
>
> 工厂可以创建任何类型的服务，无论是原始类型、对象字面量、函数，甚至是自定义类型的实例。

```
angular.module('app',[])
  .factory('endpointFactory', function() {
    return {
      get: function() {
        return 'http://some.rest.endpoint';
      }
    };
  })
.controller('MainCtrl', function(endpointFactory) {
    var vm = this;
vm.endpoint = endpointFactory.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

第30.3节：常量

常量在配置阶段和运行阶段都可用。

```
angular.module('app',[])
.constant('endpoint', 'http://some.rest.endpoint') // 定义
  .config(function(endpoint) {
    // 对端点执行某些操作
    // 在配置阶段和运行阶段均可用
  })
.controller('主控制器', function(endpoint) {         // 注入
    var vm = this;
vm.endpoint = endpoint;                              // 使用
  });
```

```
<body ng-controller="主控制器 as vm">
  <div>endpoint = {{ ::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

第30.4节：服务

服务 在运行阶段可用。

> The Factory recipe constructs a new service using a function with zero or more arguments (these are dependencies on other services). The return value of this function is the service instance created by this recipe.
>
> Factory can create a service of any type, whether it be a primitive, object literal, function, or even an instance of a custom type.

```
angular.module('app',[])
  .factory('endpointFactory', function() {
    return {
      get: function() {
        return 'http://some.rest.endpoint';
      }
    };
  })
  .controller('MainCtrl', function(endpointFactory) {
    var vm = this;
    vm.endpoint = endpointFactory.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Section 30.3: Constant

`Constant` is available both in configuration and run phases.

```
angular.module('app',[])
  .constant('endpoint', 'http://some.rest.endpoint') // define
  .config(function(endpoint) {
    // do something with endpoint
    // available in both config- and run phases
  })
  .controller('MainCtrl', function(endpoint) {       // inject
    var vm = this;
    vm.endpoint = endpoint;                          // usage
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{ ::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Section 30.4: Service

`Service` is available in run phase.

> 服务配方像值（Value）或工厂（Factory）配方一样生成服务，但它是通过使用 new 操作符调用构造函数来实现的。构造函数可以接受零个或多个参数，这些参数表示该类型实例所需的依赖项。

```
angular.module('app',[])
  .service('endpointService', function() {
    this.get = function() {
      return 'http://some.rest.endpoint';
    };
  })
.controller('主控制器', function(endpointService) {
    var vm = this;
vm.endpoint = endpointService.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

第30.5节：值

值在配置阶段和运行阶段均可用。

```
angular.module('app',[])
.value('endpoint', 'http://some.rest.endpoint') // 定义
  .run(function(endpoint) {
    // 使用endpoint
    // 仅在运行阶段可用
  })
.controller('MainCtrl', function(endpoint) {     // 注入
    var vm = this;
vm.endpoint = endpoint;                          // 使用
  });
```

```
<body ng-controller="主控制器 as vm">
  <div>endpoint = {{ ::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

> The Service recipe produces a service just like the Value or Factory recipes, but it does so by *invoking a constructor with the new operator*. The constructor can take zero or more arguments, which represent dependencies needed by the instance of this type.

```
angular.module('app',[])
  .service('endpointService', function() {
    this.get = function() {
      return 'http://some.rest.endpoint';
    };
  })
  .controller('MainCtrl', function(endpointService) {
    var vm = this;
    vm.endpoint = endpointService.get();
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

Section 30.5: Value

`Value` is available both in configuration and run phases.

```
angular.module('app',[])
  .value('endpoint', 'http://some.rest.endpoint') // define
  .run(function(endpoint) {
    // do something with endpoint
    // only available in run phase
  })
  .controller('MainCtrl', function(endpoint) {    // inject
    var vm = this;
    vm.endpoint = endpoint;                       // usage
  });
```

```
<body ng-controller="MainCtrl as vm">
  <div>endpoint = {{ ::vm.endpoint }}</div>
</body>
```

> endpoint = http://some.rest.endpoint

第31章：装饰器

第31.1节：装饰服务、工厂

下面是服务装饰器的示例，覆盖服务返回的null日期。

```
angular.module('app', [])
.config(function($provide) {
$provide.decorator('myService', function($delegate) {
      $delegate.getDate = function() { // 用实际日期对象覆盖
        return new Date();
      };
      return $delegate;
    });
  })
.service('myService', function() {
    this.getDate = function() {
      return null; // 如果不装饰，将返回null
    };
  })
.controller('myController', function(myService) {
    var vm = this;
vm.date = myService.getDate();
  });
```

```
<body ng-controller="myController as vm">
  <div ng-bind="vm.date | date:'fullDate'"></div>
</body>
```

Saturday, August 6, 2016

第31.2节：装饰指令

指令可以像服务一样被装饰，我们可以修改或替换其任何功能。注意，
指令本身在$delegate数组中的位置是0，装饰器中的名称参数必须包含
Directive后缀（区分大小写）。

因此，如果指令名为myDate，可以通过myDateDirective使用$delegate[0]访问。

下面是一个简单的示例，指令显示当前时间。我们将装饰它以每秒更新当前时间。
没有装饰时，它将始终显示相同的时间。

```
<body>
  <my-date></my-date>
</body>
```

```
angular.module('app', [])
.config(function($provide) {
$provide.decorator('myDateDirective', function($delegate, $interval) {
      var directive = $delegate[0]; // 访问指令

directive.compile = function() { // 修改 compile 函数
          return function(scope) {
directive.link.apply(this, arguments);
```


Chapter 31: Decorators

Section 31.1: Decorate service, factory

Below is example of service decorator, overriding `null` date returned by service.

```
angular.module('app', [])
  .config(function($provide) {
    $provide.decorator('myService', function($delegate) {
      $delegate.getDate = function() { // override with actual date object
        return new Date();
      };
      return $delegate;
    });
  })
  .service('myService', function() {
    this.getDate = function() {
      return null; // w/o decoration we'll be returning null
    };
  })
  .controller('myController', function(myService) {
    var vm = this;
    vm.date = myService.getDate();
  });
```

```
<body ng-controller="myController as vm">
  <div ng-bind="vm.date | date:'fullDate'"></div>
</body>
```

Saturday, August 6, 2016

Section 31.2: Decorate directive

Directives can be decorated just like services and we can modify or replace any of it's functionality. Note that directive itself is accessed at position 0 in $delegate array and name parameter in decorator must include `Directive` suffix (case sensitive).

So, if directive is called `myDate`, it can be accessed using `myDateDirective` using `$delegate[0]`.

Below is simple example where directive shows current time. We'll decorate it to update current time in one second intervals. Without decoration it will always show same time.

```
<body>
  <my-date></my-date>
</body>
```

```
angular.module('app', [])
  .config(function($provide) {
    $provide.decorator('myDateDirective', function($delegate, $interval) {
      var directive = $delegate[0]; // access directive

      directive.compile = function() { // modify compile fn
        return function(scope) {
          directive.link.apply(this, arguments);
```

```
$interval(function() {
            scope.date = new Date(); // 每秒更新日期
          }, 1000);
        };
      };

      return $delegate;
    });
  })
.directive('myDate', function() {
    return {
restrict: 'E',
template: '<span>当前时间是 {{ date | date:\'MM:ss\' }}</span>',
      link: function(scope) {
scope.date = new Date(); // 获取当前日期
      }
    };
  });
```

> Current time is 08:33

第31.3节：装饰过滤器

在装饰过滤器时，名称参数必须包含Filter后缀（区分大小写）。如果过滤器名为repeat，装饰器参数为repeatFilter。下面我们将装饰一个自定义过滤器，该过滤器将任意给定的字符串重复 n 次，并将结果反转。你也可以用同样的方式装饰Angular内置的过滤器，尽管不推荐这样做，因为这可能会影响框架的功能。

```
<body>
  <div ng-bind="'i can haz cheeseburger ' | repeat:2"></div>
</body>

angular.module('app', [])
.config(function($provide) {
    $provide.decorator('repeatFilter', function($delegate) {
      return function reverse(input, count) {
// 反转重复的字符串
return ($delegate(input, count)).split('').reverse().join('');
      };
    });
  })
  .filter('repeat', function() {
    return function(input, count) {
      // 重复字符串 n 次
return (input || '').repeat(count || 1);
    };
  });
```

> i can haz cheeseburger i can haz cheeseburger

> regrubeseehc zah nac i regrubeseehc zah nac i


```
          $interval(function() {
            scope.date = new Date(); // update date every second
          }, 1000);
        };
      };

      return $delegate;
    });
  })
.directive('myDate', function() {
  return {
    restrict: 'E',
    template: '<span>Current time is {{ date | date:\'MM:ss\' }}</span>',
    link: function(scope) {
      scope.date = new Date(); // get current date
    }
  };
});
```

> Current time is 08:33

Section 31.3: Decorate filter

When decorating filters, name parameter must include `Filter` suffix (case sensitive). If filter is called `repeat`, decorator parameter is `repeatFilter`. Below we'll decorate custom filter that repeats any given string *n* times so that result is reversed. You can also decorate angular's build-in filters the same way, although not recommended as it can affect the functionality of the framework.

```
<body>
  <div ng-bind="'i can haz cheeseburger ' | repeat:2"></div>
</body>

angular.module('app', [])
  .config(function($provide) {
    $provide.decorator('repeatFilter', function($delegate) {
      return function reverse(input, count) {
        // reverse repeated string
        return ($delegate(input, count)).split('').reverse().join('');
      };
    });
  })
  .filter('repeat', function() {
    return function(input, count) {
      // repeat string n times
      return (input || '').repeat(count || 1);
    };
  });
```

> i can haz cheeseburger i can haz cheeseburger

> regrubeseehc zah nac i regrubeseehc zah nac i

第32章：打印

第32.1节：打印服务

服务：

```
angular.module('core').factory('print_service', ['$rootScope', '$compile', '$http',
'$timeout','$q',
    function($rootScope, $compile, $http, $timeout,$q) {

        var printHtml = function (html) {
            var deferred = $q.defer();
            var hiddenFrame = $('<iframe style="display: none"></iframe>').appendTo('body')[0];

hiddenFrame.contentWindow.printAndRemove = function() {
                hiddenFrame.contentWindow.print();
$(hiddenFrame).remove();
                deferred.resolve();
            };

            var htmlContent =   "<!doctype html>"+
                                "<html>"+
                                    '<head><link rel="stylesheet" type="text/css"
href="/style/css/print.css"/></head>'+
                                    '<body onload="printAndRemove();">' +
html +
                                    '</body>'+
                                "</html>";

            var doc = hiddenFrame.contentWindow.document.open("text/html", "replace");
            doc.write(htmlContent);
doc.close();
            return deferred.promise;
        };

        var openNewWindow = function (html) {
            var newWindow = window.open("debugPrint.html");
            newWindow.addEventListener('load', function(){
                $(newWindow.document.body).html(html);
            }, false);
        };

        var print = function (templateUrl, data) {

$rootScope.isBeingPrinted = true;

            $http.get(templateUrl).success(function(template){
                var printScope = $rootScope.$new()
angular.extend(printScope, data);
                var element = $compile($('<div>' + template + '</div>'))(printScope);
                var waitForRenderAndPrint = function() {
                    if(printScope.$$phase || $http.pendingRequests.length) {
                        $timeout(waitForRenderAndPrint, 1000);
                    } else {
                            // 用 openNewWindow 替换 printHtml 以便调试
printHtml(element.html());
                        printScope.$destroy();
                    }
                };
waitForRenderAndPrint();
```


Chapter 32: Print

Section 32.1: Print Service

Service:

```
angular.module('core').factory('print_service', ['$rootScope', '$compile', '$http',
'$timeout','$q',
    function($rootScope, $compile, $http, $timeout,$q) {

        var printHtml = function (html) {
            var deferred = $q.defer();
            var hiddenFrame = $('<iframe style="display: none"></iframe>').appendTo('body')[0];

            hiddenFrame.contentWindow.printAndRemove = function() {
                hiddenFrame.contentWindow.print();
                $(hiddenFrame).remove();
                deferred.resolve();
            };

            var htmlContent =   "<!doctype html>"+
                                "<html>"+
                                    '<head><link rel="stylesheet" type="text/css"
href="/style/css/print.css"/></head>'+
                                    '<body onload="printAndRemove();">' +
                                        html +
                                    '</body>'+
                                "</html>";

            var doc = hiddenFrame.contentWindow.document.open("text/html", "replace");
            doc.write(htmlContent);
            doc.close();
            return deferred.promise;
        };

        var openNewWindow = function (html) {
            var newWindow = window.open("debugPrint.html");
            newWindow.addEventListener('load', function(){
                $(newWindow.document.body).html(html);
            }, false);
        };

        var print = function (templateUrl, data) {

            $rootScope.isBeingPrinted = true;

            $http.get(templateUrl).success(function(template){
                var printScope = $rootScope.$new()
                angular.extend(printScope, data);
                var element = $compile($('<div>' + template + '</div>'))(printScope);
                var waitForRenderAndPrint = function() {
                    if(printScope.$$phase || $http.pendingRequests.length) {
                        $timeout(waitForRenderAndPrint, 1000);
                    } else {
                        // Replace printHtml with openNewWindow for debugging
                        printHtml(element.html());
                        printScope.$destroy();
                    }
                };
                waitForRenderAndPrint();
```

```
            });
        };

        var printFromScope = function (templateUrl, scope, afterPrint) {
            $rootScope.isBeingPrinted = true;
            $http.get(templateUrl).then(function(response){
                var template = response.data;
                var printScope = scope;
                var element = $compile($('<div>' + template + '</div>'))(printScope);
                var waitForRenderAndPrint = function() {
                    if (printScope.$$phase || $http.pendingRequests.length) {
                        $timeout(waitForRenderAndPrint);
                    } else {
                                // 用 openNewWindow 替换 printHtml 以便调试
printHtml(element.html()).then(function() {
                            $rootScope.isBeingPrinted = false;
                            if (afterPrint) {
afterPrint();
                            }
                        });
                    }
                };
waitForRenderAndPrint();
            });
        };

        return {
print : print,
            printFromScope : printFromScope
        }
    }
]);
```

控制器：

```
var template_url = '/views/print.client.view.html';
print_service.printFromScope(template_url,$scope,function(){
      // 打印完成
});
```



```
            });
        };

        var printFromScope = function (templateUrl, scope, afterPrint) {
            $rootScope.isBeingPrinted = true;
            $http.get(templateUrl).then(function(response){
                var template = response.data;
                var printScope = scope;
                var element = $compile($('<div>' + template + '</div>'))(printScope);
                var waitForRenderAndPrint = function() {
                    if (printScope.$$phase || $http.pendingRequests.length) {
                        $timeout(waitForRenderAndPrint);
                    } else {
                        // Replace printHtml with openNewWindow for debugging
                        printHtml(element.html()).then(function() {
                            $rootScope.isBeingPrinted = false;
                            if (afterPrint) {
                                afterPrint();
                            }
                        });
                    }
                };
                waitForRenderAndPrint();
            });
        };

        return {
            print : print,
            printFromScope : printFromScope
        }
    }
]);
```

Controller :

```
var template_url = '/views/print.client.view.html';
print_service.printFromScope(template_url,$scope,function(){
      // Print Completed
});
```

第33章：ui-router

第33.1节：基本示例

app.js

```
angular.module('myApp', ['ui.router'])
  .controller('controllerOne', function() {
    this.message = '来自控制器一的问候！';
  })
.controller('controllerTwo', function() {
    this.message = '来自 Controller Two 的你好，世界！';
  })
.controller('controllerThree', function() {
    this.message = '来自 Controller Three 的你好，世界！';
  })
.config(function($stateProvider, $urlRouterProvider) {
   $stateProvider
.state('one', {
        url: "/one",
        templateUrl: "view-one.html",
        controller: 'controllerOne',
        controllerAs: 'ctrlOne'
      })
.state('two', {
        url: "/two",
        templateUrl: "view-two.html",
        controller: 'controllerTwo',
        controllerAs: 'ctrlTwo'
      })
.state('three', {
        url: "/three",
        templateUrl: "view-three.html",
        controller: 'controllerThree',
        controllerAs: 'ctrlThree'
      });

$urlRouterProvider.otherwise('/one');
  });
```

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- 用于切换路由的链接 -->
    <a ui-sref="one">视图一</a>
    <a ui-sref="two">视图二</a>
    <a ui-sref="three">视图三</a>
  </nav>
  <!-- 视图将被注入此处 -->
  <div ui-view></div>
  <!-- 模板可以存放在普通的 HTML 文件中 -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>
```


Chapter 33: ui-router

Section 33.1: Basic Example

app.js

```
angular.module('myApp', ['ui.router'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', function() {
    this.message = 'Hello world from Controller Three!';
  })
  .config(function($stateProvider, $urlRouterProvider) {
    $stateProvider
      .state('one', {
        url: "/one",
        templateUrl: "view-one.html",
        controller: 'controllerOne',
        controllerAs: 'ctrlOne'
      })
      .state('two', {
        url: "/two",
        templateUrl: "view-two.html",
        controller: 'controllerTwo',
        controllerAs: 'ctrlTwo'
      })
      .state('three', {
        url: "/three",
        templateUrl: "view-three.html",
        controller: 'controllerThree',
        controllerAs: 'ctrlThree'
      });

      $urlRouterProvider.otherwise('/one');
  });
```

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- links to switch routes -->
    <a ui-sref="one">View One</a>
    <a ui-sref="two">View Two</a>
    <a ui-sref="three">View Three</a>
  </nav>
  <!-- views will be injected here -->
  <div ui-view></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>
```

```
  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```

第33.2节：多视图

app.js

```
angular.module('myApp', ['ui.router'])
  .controller('controllerOne', function() {
    this.message = '来自控制器一的问候！';
  })
.controller('controllerTwo', function() {
    this.message = '来自 Controller Two 的你好，世界！';
  })
.controller('controllerThree', function() {
    this.message = '来自 Controller Three 的你好，世界！';
  })
.controller('controllerFour', function() {
    this.message = '来自控制器四的问候！';
  })
.config(function($stateProvider, $urlRouterProvider) {
   $stateProvider
.state('one', {
        url: "/one",
views: {
          "viewA": {
templateUrl: "view-one.html",
            controller: 'controllerOne',
            controllerAs: 'ctrlOne'
          },
          "viewB": {
templateUrl: "view-two.html",
            controller: 'controllerTwo',
            controllerAs: 'ctrlTwo'
          }
        }
      })
.state('two', {
url: "/two",
views: {
          "viewA": {
templateUrl: "view-three.html",
            controller: 'controllerThree',
            controllerAs: 'ctrlThree'
          },
          "viewB": {
templateUrl: "view-four.html",
            controller: 'controllerFour',
            controllerAs: 'ctrlFour'
          }
        }
      });

$urlRouterProvider.otherwise('/one');
  });
```

index.html

bem...eindoc.com


```
  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>
</div>
```


Section 33.2: Multiple Views

app.js

```
angular.module('myApp', ['ui.router'])
  .controller('controllerOne', function() {
    this.message = 'Hello world from Controller One!';
  })
  .controller('controllerTwo', function() {
    this.message = 'Hello world from Controller Two!';
  })
  .controller('controllerThree', function() {
    this.message = 'Hello world from Controller Three!';
  })
  .controller('controllerFour', function() {
    this.message = 'Hello world from Controller Four!';
  })
  .config(function($stateProvider, $urlRouterProvider) {
    $stateProvider
      .state('one', {
        url: "/one",
        views: {
          "viewA": {
            templateUrl: "view-one.html",
            controller: 'controllerOne',
            controllerAs: 'ctrlOne'
          },
          "viewB": {
            templateUrl: "view-two.html",
            controller: 'controllerTwo',
            controllerAs: 'ctrlTwo'
          }
        }
      })
      .state('two', {
        url: "/two",
        views: {
          "viewA": {
            templateUrl: "view-three.html",
            controller: 'controllerThree',
            controllerAs: 'ctrlThree'
          },
          "viewB": {
            templateUrl: "view-four.html",
            controller: 'controllerFour',
            controllerAs: 'ctrlFour'
          }
        }
      });

    $urlRouterProvider.otherwise('/one');
  });
```

index.html

```
<div ng-app="myApp">
  <nav>
    <!-- 用于切换路由的链接 -->
    <a ui-sref="one">路由一</a>
    <a ui-sref="two">路由二</a>
  </nav>
  <!-- 视图将被注入此处 -->
  <div ui-view="viewA"></div>
  <div ui-view="viewB"></div>
  <!-- 模板可以存在于普通的html文件中 -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-four.html">
    <h1>{{ctrlFour.message}}</h1>
  </script>
</div>
```

第33.3节：使用resolve函数加载数据

app.js

```
angular.module('我的应用', ['ui.router'])
  .service('用户', ['$http', function 用户 ($http) {
    this.getProfile = function (id) {
      return $http.get(...) // 从API加载数据的方法
    };
  }])
.controller('profileCtrl', ['profile', function profileCtrl (profile) {
    // 注入以resolve函数名命名的已解析数据
    // 数据将已经被返回并处理
    this.profile = profile;
  }])
.config(['$stateProvider', '$urlRouterProvider', function ($stateProvider, $urlRouterProvider) {
    $stateProvider
.state('profile', {
        url: "/profile/:userId",
        templateUrl: "profile.html",
        controller: 'profileCtrl',
        controllerAs: 'vm',
resolve: {
profile: ['$stateParams', 'User', function ($stateParams, User) {
              // $stateParams 将包含 URL 中定义的任何参数
              return User.getProfile($stateParams.userId)
                // .then 仅在需要处理返回数据时才使用
              .then(function (data) {
                  return doSomeProcessing(data);
                });
          }]
        }
      }]);
```



```
<div ng-app="myApp">
  <nav>
    <!-- links to switch routes -->
    <a ui-sref="one">Route One</a>
    <a ui-sref="two">Route Two</a>
  </nav>
  <!-- views will be injected here -->
  <div ui-view="viewA"></div>
  <div ui-view="viewB"></div>
  <!-- templates can live in normal html files -->
  <script type="text/ng-template" id="view-one.html">
    <h1>{{ctrlOne.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-two.html">
    <h1>{{ctrlTwo.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-three.html">
    <h1>{{ctrlThree.message}}</h1>
  </script>

  <script type="text/ng-template" id="view-four.html">
    <h1>{{ctrlFour.message}}</h1>
  </script>
</div>
```

Section 33.3: Using resolve functions to load data

app.js

```
angular.module('myApp', ['ui.router'])
  .service('User', ['$http', function User ($http) {
    this.getProfile = function (id) {
      return $http.get(...) // method to load data from API
    };
  }])
  .controller('profileCtrl', ['profile', function profileCtrl (profile) {
    // inject resolved data under the name of the resolve function
    // data will already be returned and processed
    this.profile = profile;
  }])
  .config(['$stateProvider', '$urlRouterProvider', function ($stateProvider, $urlRouterProvider) {
    $stateProvider
      .state('profile', {
        url: "/profile/:userId",
        templateUrl: "profile.html",
        controller: 'profileCtrl',
        controllerAs: 'vm',
        resolve: {
          profile: ['$stateParams', 'User', function ($stateParams, User) {
            // $stateParams will contain any parameter defined in your url
            return User.getProfile($stateParams.userId)
              // .then is only necessary if you need to process returned data
              .then(function (data) {
                return doSomeProcessing(data);
              });
          }]
        }
      }]);
```

```
$urlRouterProvider.otherwise('/');
  });
```

profile.html

```
<ul>
  <li>姓名: {{vm.profile.name}}</li>
  <li>年龄: {{vm.profile.age}}</li>
  <li>性别: {{vm.profile.sex}}</li>
</ul>
```

查看UI-Router Wiki 中关于 resolves 的条目。

在$stateChangeSuccess事件触发之前，必须先解析解析函数，这意味着UI在状态上的所有解析函数完成之前不会加载。这是确保数据可用于控制器和UI的好方法。然而，你可以看到解析函数应该快速执行，以避免阻塞UI。

第33.4节：嵌套视图 / 状态

app.js

```
var app = angular.module('myApp',['ui.router']);

app.config(function($stateProvider,$urlRouterProvider) {

    $stateProvider

.state('home', {
url: '/home',
templateUrl: 'home.html',
        controller: function($scope){
            $scope.text = '这是主页'
        }
    })

.state('home.nested1',{
        url: '/nested1',
templateUrl:'nested1.html',
        controller: function($scope){
            $scope.text1 = '这是嵌套视图1'
        }
    })

.state('home.nested2',{
url: '/nested2',
templateUrl:'nested2.html',
        controller: function($scope){
            $scope.fruits = ['apple','mango','oranges'];
        }
    });

$urlRouterProvider.otherwise('/home');

});
```

index.html

```
<div ui-view></div>
```



```
    $urlRouterProvider.otherwise('/');
  });
```

profile.html

```
<ul>
  <li>Name: {{vm.profile.name}}</li>
  <li>Age: {{vm.profile.age}}</li>
  <li>Sex: {{vm.profile.sex}}</li>
</ul>
```

View UI-Router Wiki entry on resolves here.

Resolve functions must be resolved before the `$stateChangeSuccess` event is fired, which means that the UI will not load until *all* resolve functions on the state have finished. This is a great way to ensure that data will be available to your controller and UI. However, you can see that a resolve function should be fast in order to avoid hanging the UI.

Section 33.4: Nested Views / States

app.js

```
var app = angular.module('myApp',['ui.router']);

app.config(function($stateProvider,$urlRouterProvider) {

    $stateProvider

    .state('home', {
        url: '/home',
        templateUrl: 'home.html',
        controller: function($scope){
            $scope.text = 'This is the Home'
        }
    })

    .state('home.nested1',{
        url: '/nested1',
        templateUrl:'nested1.html',
        controller: function($scope){
            $scope.text1 = 'This is the nested view 1'
        }
    })

    .state('home.nested2',{
        url: '/nested2',
        templateUrl:'nested2.html',
        controller: function($scope){
            $scope.fruits = ['apple','mango','oranges'];
        }
    });

    $urlRouterProvider.otherwise('/home');

});
```

index.html

```
<div ui-view></div>
```

```
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js"></script>
    <script src="angular-ui-router.min.js"></script>
    <script src="app.js"></script>
```

home.html

```
<div>
<h1> {{text}} </h1>
<br>
    <a ui-sref="home.nested1">显示 nested1</a>
    <br>
    <a ui-sref="home.nested2">显示 nested2</a>
    <br>

    <div ui-view></div>
</div>
```

nested1.html

```
<div>
<h1> {{text1}} </h1>
</div>
```

nested2.html

```
<div>
    <ul>
      <li ng-repeat="fruit in fruits">{{ fruit }}</li>
    </ul>
</div>
```

belindoc.com


```
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js"></script>
    <script src="angular-ui-router.min.js"></script>
    <script src="app.js"></script>
```

home.html

```
<div>
<h1> {{text}} </h1>
<br>
    <a ui-sref="home.nested1">Show nested1</a>
    <br>
    <a ui-sref="home.nested2">Show nested2</a>
    <br>

    <div ui-view></div>
</div>
```

nested1.html

```
<div>
<h1> {{text1}} </h1>
</div>
```

nested2.html

```
<div>
    <ul>
      <li ng-repeat="fruit in fruits">{{ fruit }}</li>
    </ul>
</div>
```

第34章：内置辅助函数

第34.1节：angular.equals

angular.equals函数比较并判断两个对象或值是否相等，angular.equals执行深度比较，且仅当满足以下至少一条条件时返回true。

> angular.equals(value1, value2)

1. 如果对象或值通过了===比较
2. 如果两个对象或值类型相同，且它们的所有属性通过 angular.equals
3. 两个值都等于NaN
4. 两个值表示相同正则表达式的结果。

当你需要通过值或结果而不仅仅是引用来深度比较对象或数组时，此函数非常有用。

示例

```
angular.equals(1, 1) // true
angular.equals(1, 2) // false
angular.equals({}, {}) // true，注意 {}==={} 是 false
angular.equals({a: 1}, {a: 1}) // true
angular.equals({a: 1}, {a: 2}) // false
angular.equals(NaN, NaN) // true
```

第34.2节：angular.toJson

函数 angular.toJson 会接收一个对象并将其序列化为 JSON 格式的字符串。

与原生函数 JSON.stringify 不同，此函数会移除所有以 $$ 开头的属性（因为 angular 通常会用 $$ 作为内部属性的前缀）

```
angular.toJson(object)
```

由于数据需要在通过网络之前进行序列化，此函数可将您希望传输的任何数据转换为 JSON。

此函数对于调试也很有用，因为它的工作方式类似于.toString方法的作用。

示例：

```
angular.toJson({name: "barf", occupation: "mog", $$somebizzareproperty: 42})
// "{"name":"barf","occupation":"mog"}"
angular.toJson(42)
// "42"
angular.toJson([1, "2", 3, "4"])
// "[1,"2",3,"4"]"
var fn = function(value) {return value}
angular.toJson(fn)
// 未定义，函数在 JSON 中没有表示形式
```


Chapter 34: Built-in helper Functions

Section 34.1: angular.equals

The `angular.equals` function compares and determines if 2 objects or values are equal, `angular.equals` performs a deep comparison and returns true if and only if at least 1 of the following conditions is met.

> angular.equals(value1, value2)

1. If the objects or values pass the `===` comparison
2. If both objects or values are of the same type, and all of their properties are also equal by using `angular.equals`
3. Both values are equal to **`NaN`**
4. Both values represent the same regular expression's result.

This function is helpful when you need to deep compare objects or arrays by their values or results rather than just references.

Examples

```
angular.equals(1, 1) // true
angular.equals(1, 2) // false
angular.equals({}, {}) // true, note that {}=={} is false
angular.equals({a: 1}, {a: 1}) // true
angular.equals({a: 1}, {a: 2}) // false
angular.equals(NaN, NaN) // true
```

Section 34.2: angular.toJson

The function `angular.toJson` will take an object and serialize it into a JSON formatted string.

Unlike the native function `JSON.stringify`, This function will remove all properties beginning with $$ (as angular usually prefixes internal properties with $$)

```
angular.toJson(object)
```

As data needs to be serialized before passing through a network, this function is useful to turn any data you wish to transmit into JSON.

This function is also useful for debugging as it works similarly to a `.toString` method would act.

Examples:

```
angular.toJson({name: "barf", occupation: "mog", $$somebizzareproperty: 42})
// "{"name":"barf","occupation":"mog"}"
angular.toJson(42)
// "42"
angular.toJson([1, "2", 3, "4"])
// "[1,"2",3,"4"]"
var fn = function(value) {return value}
angular.toJson(fn)
// undefined, functions have no representation in JSON
```

第34.3节：angular.copy

angular.copy函数接受一个对象、数组或值，并创建其深拷贝。

> angular.copy()

示例：

对象：

```
let obj = {name: "vespa", occupation: "princess"};
let cpy = angular.copy(obj);
cpy.name = "yogurt"
// obj = {name: "vespa", occupation: "princess"}
// cpy = {name: "yogurt", occupation: "princess"}
```

数组：

```
var w = [a, [b, [c, [d]]]];
var q = angular.copy(w);
// q = [a, [b, [c, [d]]]]
```

在上述示例中，angular.equals(w, q)将返回true，因为.equals通过值来测试相等性。
然而 w === q 将返回 false，因为对象和数组之间的严格比较是通过引用进行的。

第34.4节：angular.isString

函数 angular.isString 如果传入的对象或值是 string 类型，则返回 true

> angular.isString(value1)

示例

```
angular.isString("hello") // true
angular.isString([1, 2]) // false
angular.isString(42) // false
```

这相当于执行

```
typeof someValue === "string"
```

第34.5节：angular.isArray

函数 angular.isArray 仅当传入的对象或值是 Array 类型时返回 true。

> angular.isArray(value)

示例

```
angular.isArray([]) // true
angular.isArray([2, 3]) // true
```


Section 34.3: angular.copy

The `angular.copy` function takes an object, array or a value and creates a deep copy of it.

> angular.copy()

Example:

Objects:

```
let obj = {name: "vespa", occupation: "princess"};
let cpy = angular.copy(obj);
cpy.name = "yogurt"
// obj = {name: "vespa", occupation: "princess"}
// cpy = {name: "yogurt", occupation: "princess"}
```

Arrays:

```
var w = [a, [b, [c, [d]]]];
var q = angular.copy(w);
// q = [a, [b, [c, [d]]]]
```

At the above example `angular.equals(w, q)` will evaluate to true because `.equals` tests equality by value.
however `w === q` will evaluate to false because strict comparison between objects and arrays is done by reference.

Section 34.4: angular.isString

The function `angular.isString` returns true if the object or value given to it is of the type `string`

> angular.isString(value1)

Examples

```
angular.isString("hello") // true
angular.isString([1, 2]) // false
angular.isString(42) // false
```

This is the equivalent of performing

```
typeof someValue === "string"
```

Section 34.5: angular.isArray

The `angular.isArray` function returns true if and only if the object or value passed to it is of the type `Array`.

> angular.isArray(value)

Examples

```
angular.isArray([]) // true
angular.isArray([2, 3]) // true
```

```
angular.isArray({}) // false
angular.isArray(17) // false
```

它相当于

```
数组。isArray(someValue)
```

第34.6节：angular.merge

函数 angular.merge 会将源对象的所有可枚举属性深度合并到目标对象中。

该函数返回对现已扩展的目标对象的引用

> angular.merge(destination, source)

示例

```
angular.merge({}, {}) // {}
angular.merge({name: "king roland"}, {password: "12345"})
// {name: "king roland", password: "12345"}
angular.merge({a: 1}, [4, 5, 6]) // {0: 4, 1: 5, 2: 6, a: 1}
angular.merge({a: 1}, {b: {c: {d: 2}}}) //{"a":1,"b":{"c":{"d":2}}}
```

第34.7节：angular.isDefined 和 angular.isUndefined

函数angular.isDefined用于测试一个值是否已定义

> angular.isDefined(someValue)

这相当于执行

```
value !== undefined; // 如果value已定义，则结果为true
```

示例

```
angular.isDefined(42) // true
angular.isDefined([1, 2]) // true
angular.isDefined(undefined) // false
angular.isDefined(null) // true
```

函数angular.isUndefined用于测试一个值是否未定义（实际上是angular.isDefined的相反）

> angular.isUndefined(someValue)

这相当于执行

```
value === undefined; // 如果value未定义，则结果为true
```

或者直接


```
angular.isArray({}) // false
angular.isArray(17) // false
```

It is the equivalent of

```
Array.isArray(someValue)
```

Section 34.6: angular.merge

The function angular.merge takes all the enumerable properties from the source object to deeply extend the destination object.

The function returns a reference to the now extended destination object

> angular.merge(destination, source)

Examples

```
angular.merge({}, {}) // {}
angular.merge({name: "king roland"}, {password: "12345"})
// {name: "king roland", password: "12345"}
angular.merge({a: 1}, [4, 5, 6]) // {0: 4, 1: 5, 2: 6, a: 1}
angular.merge({a: 1}, {b: {c: {d: 2}}}) // {"a":1,"b":{"c":{"d":2}}}
```

Section 34.7: angular.isDefined and angular.isUndefined

The function `angular.isDefined` tests a value if it is defined

> angular.isDefined(someValue)

This is the equivalent of performing

```
value !== undefined; // will evaluate to true is value is defined
```

Examples

```
angular.isDefined(42) // true
angular.isDefined([1, 2]) // true
angular.isDefined(undefined) // false
angular.isDefined(null) // true
```

The function `angular.isUndefined` tests if a value is undefined (it is effectively the opposite of `angular.isDefined`)

> angular.isUndefined(someValue)

This is the equivalent of performing

```
value === undefined; // will evaluate to true is value is undefined
```

Or just

```
!angular.isDefined(value)
```

示例

```
angular.isUndefined(42) // false
angular.isUndefined(undefined) // true
```

第34.8节：angular.isDate

angular.isDate函数仅当传入的对象是Date类型时返回true。

> angular.isDate(value)

示例

```
angular.isDate("lone star") // false
angular.isDate(new Date()) // true
```

第34.9节：angular.noop

angular.noop是一个不执行任何操作的函数，当你需要传入一个什么都不做的函数参数时，可以使用angular.noop。

> angular.noop()

angular.noop的一个常见用法是为某个函数提供一个空回调，否则当传入的参数不是函数时，该函数会抛出错误。

示例：

```
$scope.onSomeChange = function(model, callback) {
    updateTheModel(model);
    if (angular.isFunction(callback)) {
        callback();
    } else {
        throw new Error("错误：callback 不是一个函数！");
    }
};

$scope.onSomeChange(42, function() {console.log("hello callback")});
// 将更新模型并打印 'hello callback'
$scope.onSomeChange(42, angular.noop);
// 将更新模型
```

附加示例：

```
angular.noop() // undefined
angular.isFunction(angular.noop) // true
```

第34.10节：angular.isElement

如果传入的参数是DOM元素或jQuery包装的元素，angular.isElement将返回true。


```
!angular.isDefined(value)
```

Examples

```
angular.isUndefined(42) // false
angular.isUndefined(undefined) // true
```

Section 34.8: angular.isDate

The `angular.isDate` function returns true if and only if the object passed to it is of the type Date.

> angular.isDate(value)

Examples

```
angular.isDate("lone star") // false
angular.isDate(new Date()) // true
```

Section 34.9: angular.noop

The `angular.noop` is a function that performs no operations, you pass `angular.noop` when you need to provide a function argument that will do nothing.

> angular.noop()

A common use for `angular.noop` can be to provide an empty callback to a function that will otherwise throw an error when something else than a function is passed to it.

Example:

```
$scope.onSomeChange = function(model, callback) {
    updateTheModel(model);
    if (angular.isFunction(callback)) {
        callback();
    } else {
        throw new Error("error: callback is not a function!");
    }
};

$scope.onSomeChange(42, function() {console.log("hello callback")});
// will update the model and print 'hello callback'
$scope.onSomeChange(42, angular.noop);
// will update the model
```

Additional examples:

```
angular.noop() // undefined
angular.isFunction(angular.noop) // true
```

Section 34.10: angular.isElement

The `angular.isElement` returns true if the argument passed to it is a DOM Element or a jQuery wrapped Element.

> angular.isElement(elem)

此函数用于类型检查传入的参数是否为元素，然后再作为元素进行处理。

示例：

```
angular.isElement(document.querySelector("body"))
// true
angular.isElement(document.querySelector("#some_id"))
// 如果"some_id"未作为所选DOM中的id使用，则为false
angular.isElement("<div></div>")
// false
```

第34.11节：angular.isFunction

函数angular.isFunction 用于判断并返回true，当且仅当传入的值是对函数的引用时。

该函数返回对现已扩展的目标对象的引用

> angular.isFunction(fn)

示例

```
var onClick = function(e) {return e};
angular.isFunction(onClick); // true

var someArray = ["pizza", "the", "hut"];
angular.isFunction(someArray ); // false
```

第34.12节：angular.identity

angular.identity函数返回传入的第一个参数。

> angular.identity(argument)

此函数在函数式编程中非常有用，当预期的函数未传入时，可以将此函数作为默认值提供。

示例：

```
angular.identity(42) // 42
```

```
var mutate = function(fn, num) {
    return angular.isFunction(fn) ? fn(num) : angular.identity(num)
}

mutate(function(value) { return value - 7 }, 42) // 35
mutate(null, 42) // 42
mutate("mount. rushmore", 42) // 42
```


> angular.isElement(elem)

This function is useful to type check if a passed argument is an element before being processed as such.

Examples:

```
angular.isElement(document.querySelector("body"))
// true
angular.isElement(document.querySelector("#some_id"))
// false if "some_id" is not using as an id inside the selected DOM
angular.isElement("<div></div>")
// false
```

Section 34.11: angular.isFunction

The function `angular.isFunction` determines and returns true if and only if the value passed to is a reference to a function.

The function returns a reference to the now extended destination object

> angular.isFunction(fn)

Examples

```
var onClick = function(e) {return e};
angular.isFunction(onClick); // true

var someArray = ["pizza", "the", "hut"];
angular.isFunction(someArray ); // false
```

Section 34.12: angular.identity

The `angular.identity` function returns the first argument passed to it.

> angular.identity(argument)

This function is useful for functional programming, you can provide this function as a default in case an expected function was not passed.

Examples:

```
angular.identity(42) // 42
```

```
var mutate = function(fn, num) {
    return angular.isFunction(fn) ? fn(num) : angular.identity(num)
}

mutate(function(value) {return value-7}, 42) // 35
mutate(null, 42) // 42
mutate("mount. rushmore", 42) // 42
```


Beijing doc.com

第34.13节：angular.forEach

Angular 的 forEach 接受一个对象和一个迭代函数。然后它会对该对象的每个可枚举属性/值运行迭代函数。该函数也适用于数组。

像 JS 版本的Array.prototype.forEach一样，该函数不会遍历继承的属性（原型属性），但是函数不会尝试处理null或undefined值，而是直接返回它们。

> angular.forEach(object, function(value, key) { // function});

示例：

```
angular.forEach({"a": 12, "b": 34}, (value, key) => console.log("key: " + key + ", value: " + value))
// key: a, value: 12
// key: b, value: 34
angular.forEach([2, 4, 6, 8, 10], (value, key) => console.log(key))
// 将打印数组索引：1, 2, 3, 4, 5
angular.forEach([2, 4, 6, 8, 10], (value, key) => console.log(value))
// 将打印数组值：2, 4, 6, 7, 10
angular.forEach(undefined, (value, key) => console.log("key: " + key + ", value: " + value))
// undefined
```

第34.14节：angular.isNumber

angular.isNumber函数仅当传入的对象或值是Number类型时返回true，
这包括+Infinity、-Infinity和NaN

> angular.isNumber(value)

该函数不会引起类型强制转换，例如

```
"23" == 23 // true
```

示例

```
angular.isNumber("23") // false
angular.isNumber(23) // true
angular.isNumber(NaN) // true
angular.isNumber(Infinity) // true
```

该函数不会引起类型强制转换，例如

```
"23" == 23 // true
```

第34.15节：angular.isObject

angular.isObject仅当传入的参数是对象时返回true，该函数对数组也返回true，
但对null返回false，尽管typeof null是object。

> angular.isObject(value)

Section 34.13: angular.forEach

The `angular.forEach` accepts an object and an iterator function. It then runs the iterator function over each enumerable property/value of the object. This function also works on arrays.

Like the JS version of `Array.prototype.forEach` The function does not iterate over inherited properties (prototype properties), however the function will not attempt to process a **`null`** or an **`undefined`** value and will just return it.

> angular.forEach(object, function(value, key) { // function});

Examples:

```
angular.forEach({"a": 12, "b": 34}, (value, key) => console.log("key: " + key + ", value: " + value))
// key: a, value: 12
// key: b, value: 34
angular.forEach([2, 4, 6, 8, 10], (value, key) => console.log(key))
// will print the array indices: 1, 2, 3, 4, 5
angular.forEach([2, 4, 6, 8, 10], (value, key) => console.log(value))
// will print the array values: 2, 4, 6, 7, 10
angular.forEach(undefined, (value, key) => console.log("key: " + key + ", value: " + value))
// undefined
```

Section 34.14: angular.isNumber

The `angular.isNumber` function returns true if and only if the object or value passed to it is of the type Number, this includes +Infinity, -Infinity and NaN

> angular.isNumber(value)

This function will not cause a type coercion such as

```
"23" == 23 // true
```

Examples

```
angular.isNumber("23") // false
angular.isNumber(23) // true
angular.isNumber(NaN) // true
angular.isNumber(Infinity) // true
```

This function will not cause a type coercion such as

```
"23" == 23 // true
```

Section 34.15: angular.isObject

The `angular.isObject` return true if and only if the argument passed to it is an object, this function will also return true for an Array and will return false for **`null`** even though **`typeof null`** is `object` .

> angular.isObject(value)

当你需要一个已定义的对象进行处理时，该函数对于类型检查非常有用。

示例：

```
angular.isObject({name: "skroob", job: "president"})
// true
angular.isObject(null)
// false
angular.isObject([null])
// true
angular.isObject(new Date())
// true
angular.isObject(undefined)
// false
```

第34.16节：angular.fromJson

函数angular.fromJson将反序列化一个有效的JSON字符串并返回一个对象或数组。

> angular.fromJson(string|object)

注意该函数不仅限于字符串，它会输出传入的任何对象的表示形式。

示例：

```
angular.fromJson("{\"yogurt\": \"strawberries\"}")
// Object {yogurt: "strawberries"}
angular.fromJson('{jam: "raspberries"}')
// 会抛出异常，因为该字符串不是有效的 JSON
angular.fromJson(this)
// Window {external: Object, chrome: Object, _gaq: Y, angular: Object, ng339: 3...}
angular.fromJson([1, 2])
// [1, 2]
typeof angular.fromJson(new Date())
// "object"
```

beiyindoc.com

This function is useful for type checking when you need a defined object to process.

Examples:

```
angular.isObject({name: "skroob", job: "president"})
// true
angular.isObject(null)
// false
angular.isObject([null])
// true
angular.isObject(new Date())
// true
angular.isObject(undefined)
// false
```

Section 34.16: angular.fromJson

The function `angular.fromJson` will deserialize a valid JSON string and return an Object or an Array.

> angular.fromJson(string|object)

Note that this function is not limited to only strings, it will output a representation of any object passed to it.

Examples:

```
angular.fromJson("{\"yogurt\": \"strawberries\"}")
// Object {yogurt: "strawberries"}
angular.fromJson('{jam: "raspberries"}')
// will throw an exception as the string is not a valid JSON
angular.fromJson(this)
// Window {external: Object, chrome: Object, _gaq: Y, angular: Object, ng339: 3...}
angular.fromJson([1, 2])
// [1, 2]
typeof angular.fromJson(new Date())
// "object"
```

第35章：digest循环演练

第35.1节：$digest 和 $watch

实现双向数据绑定，以达到前面示例的效果，可以通过两个核心
函数来完成：

- **$digest** 在用户交互后调用（绑定 DOM=>变量）
- **$watch** 设置回调函数，在变量变化后调用（绑定变量=>DOM）

注意：此示例为演示用途，并非实际的 angular 代码

```
<input id="input"/>
<span id="span"></span>
```

我们需要的两个函数：

```
var $watches = [];
function $digest(){
    $watches.forEach(function($w){
        var val = $w.val();
        if($w.prevVal !== val){
            $w.callback(val, $w.prevVal);
            $w.prevVal = val;
        }
    })
}
function $watch(val, callback){
$watches.push({val:val, callback:callback, prevVal: val() })
}
```

现在我们可以使用这些函数将变量绑定到DOM（Angular自带内置指令
可以为你完成这项工作）：

```
var realVar;
//这通常由ng-model指令完成
input1.addEventListener('keyup',function(e){
    realVar=e.target.value;
$digest()
}, true);

//这通常通过{{表达式}}或ng-bind指令完成
$watch(function(){return realVar},function(val){
span1.innerHTML = val;
});
```

当然，实际的实现更复杂，并支持诸如绑定到哪个元素和使用哪个变量等参数

一个运行示例可以在这里找到：https://jsfiddle.net/azofxd4j/

第35.2节：$scope树

当我们需要将单个HTML元素绑定到单个变量时，前面的示例已经足够好。

Chapter 35: digest loop walkthrough

Section 35.1: $digest and $watch

Implementing two-way-data-binding, to achieve the result from the previous example, could be done with two core functions:

- **$digest** is called after a user interaction (binding DOM=>variable)
- **$watch** sets a callback to be called after variable changes (binding variable=>DOM)

note: this is example is a demonstration, not the actual angular code

```
<input id="input"/>
<span id="span"></span>
```

The two functions we need:

```
var $watches = [];
function $digest(){
    $watches.forEach(function($w){
        var val = $w.val();
        if($w.prevVal !== val){
            $w.callback(val, $w.prevVal);
            $w.prevVal = val;
        }
    })
}
function $watch(val, callback){
    $watches.push({val:val, callback:callback, prevVal: val() })
}
```

Now we could now use these functions to hook up a variable to the DOM (angular comes with built-in directives which will do this for you):

```
var realVar;
//this is usually done by ng-model directive
input1.addEventListener('keyup',function(e){
    realVar=e.target.value;
    $digest()
}, true);

//this is usually done with {{expressions}} or ng-bind directive
$watch(function(){return realVar},function(val){
    span1.innerHTML = val;
});
```

Off-course, the real implementations are more complex, and support parameters such as **which element** to bind to, and **what variable** to use

A running example could be found here: https://jsfiddle.net/azofxd4j/

Section 35.2: the $scope tree

The previous example is good enough when we need to bind a single html element, to a single variable.

实际上——我们需要将多个元素绑定到多个变量：

```
<span ng-repeat="number in [1,2,3,4,5]">{{number}}</span>
```

这个ng-repeat将5个元素绑定到5个名为number的变量，每个变量的值都不同！

Angular 实现这种行为的方式是为每个需要单独变量的元素使用一个独立的上下文。这个上下文称为作用域（scope）。

每个作用域包含属性，这些属性是绑定到DOM的变量，$digest 和 $watch 函数作为作用域的方法实现。

DOM是一个树状结构，变量需要在树的不同层级中使用：

```
<div>
    <input ng-model="person.name" />
    <span ng-repeat="number in [1,2,3,4,5]">{{number}} {{person.name}}</span>
</div>
```

但正如我们所见，ng-repeat 内部变量的上下文（或作用域）与其上方的上下文不同。为了解决这个问题——angular 将作用域实现为树结构。

每个作用域都有一个子作用域数组，调用其 $digest 方法会运行所有子作用域的 $digest 方法。

这样——在更改输入后——会调用该div作用域的 $digest 方法，随后运行其5个子作用域的 $digest 方法，从而更新内容。

一个简单的作用域实现可能如下所示：

```
function $scope(){
    this.$children = [];
    this.$watches = [];
}

$scope.prototype.$digest = function(){
    this.$watches.forEach(function($w){
        var val = $w.val();
        if($w.prevVal !== val){
            $w.callback(val, $w.prevVal);
          $w.prevVal = val;
        }
    });
    this.$children.forEach(function(c){
        c.$digest();
    });
}

$scope.prototype.$watch = function(val, callback){
    this.$watches.push({val:val, callback:callback, prevVal: val() })
}
```

注意：此示例为演示用途，并非实际的 angular 代码

第35.3节：双向数据绑定

Angular 有一些神奇的机制。它能够将 DOM 绑定到真实的 JavaScript 变量。

In reality - we need to bind many elements to many variables:

```
<span ng-repeat="number in [1,2,3,4,5]">{{number}}</span>
```

This ng-repeat binds 5 elements to 5 variables called number, with a different value for each of them!

The way angular achieves this behavior is using a separate context for each element which needs separate variables. This context is called a scope.

Each scope contains properties, which are the variables bound to the DOM, and the $digest and $watch functions are implemented as methods of the scope.

The DOM is a tree, and variables need to be used in different levels of the tree:

```
<div>
    <input ng-model="person.name" />
    <span ng-repeat="number in [1,2,3,4,5]">{{number}} {{person.name}}</span>
</div>
```

But as we saw, the context(or scope) of variables inside ng-repeat is different to the context above it. To solve this - angular implements scopes as a tree.

Each scope has an array of children, and calling its $digest method will run all of its children's $digest method.

This way - after changing the input - $digest is called for the div's scope, which then runs the $digest for its 5 children - which will update its content.

A simple implementation for a scope, could look like this:

```
function $scope(){
    this.$children = [];
    this.$watches = [];
}

$scope.prototype.$digest = function(){
    this.$watches.forEach(function($w){
        var val = $w.val();
        if($w.prevVal !== val){
            $w.callback(val, $w.prevVal);
          $w.prevVal = val;
        }
    });
    this.$children.forEach(function(c){
        c.$digest();
    });
}

$scope.prototype.$watch = function(val, callback){
    this.$watches.push({val:val, callback:callback, prevVal: val() })
}
```

note: this is example is a demonstration, not the actual angular code

Section 35.3: two way data binding

Angular has some magic under its hood. it enables binding DOM to real js variables.

Angular 使用一个名为“*digest loop*”的循环，在变量发生任何变化后调用——执行回调函数以更新 DOM。

例如，ng-model 指令会给该输入框绑定一个 keyup 事件监听器：

```
<input ng-model="variable" />
```

每次触发keyup事件时，digest循环就开始。

在某个时刻，digest 循环 会迭代一个回调函数，该函数更新此 span 的内容：

```
<span>{{variable}}</span>
```

此示例的基本生命周期，简要（非常示意性地）总结了 Angular 的工作原理：

1.Angular 扫描 html
   - `ng-model` 指令在输入框上创建一个 `keyup` 监听器
   - 表达式 在 span 内添加一个回调到 *digest 循环*
2.用户与输入框交互
   - keyup 监听器启动 digest 循环
   - digest 循环 调用回调函数
   - 回调函数更新 span 的内容

elindoc.com

Angular uses a loop, named the "*digest loop*", which is called after any change of a variable - calling callbacks which update the DOM.

For example, the `ng-model` directive attaches a `keyup` eventListener to this input:

```
<input ng-model="variable" />
```

Every time the `keyup` event fires, the *digest loop* starts.

At some point, the *digest loop* iterates over a callback which updates the contents of this span:

```
<span>{{variable}}</span>
```

The basic life-cycle of this example, summarizes (very Schematically) how angular works::

1. Angular scans html
   - `ng-model` directive creates a `keyup` listener on input
   - `expression` inside span adds a callback to *digest cycle*
2. User interacts with input
   - `keyup` listener starts *digest cycle*
   - *digest cycle* calles the callback
   - Callback updates span's contents

第36章：Angular $scopes

第36.1节：整个应用中可用的函数

请注意，这种方法可能被认为是 Angular 应用程序中的不良设计，因为它要求程序员既要记住函数在作用域树中的位置，又要了解作用域继承。在许多情况下，更倾向于注入服务（Angular 实践 - 使用作用域继承与注入）。

此示例仅展示了如何根据我们的需求使用作用域继承，以及如何利用它，而不是设计整个应用程序的最佳实践。

在某些情况下，我们可以利用作用域继承，将函数设置为 rootScope 的属性。这样，应用程序中的所有作用域（除了孤立作用域）都会继承此函数，且可以从应用程序的任何地方调用它。

```
angular.module('app', [])
.run(['$rootScope', function($rootScope){
    var messages = []
$rootScope.addMessage = function(msg){
        messages.push(msg);
    }
}]);

<div ng-app="app">
    <a ng-click="addMessage('hello world!')">它可以从这里访问</a><div ng-include="inner.html"></
    div>
</div>
```

inner.html:

```
<div>
    <button ng-click="addMessage('page')">从这里开始！</button>
</div>
```

第36.2节：避免继承原始值

在JavaScript中，赋值一个非原始值（例如对象、数组、函数以及许多其他类型）时，会保留对赋值对象的引用（内存地址）。

将一个原始值（字符串、数字、布尔值或符号）赋给两个不同的变量，并修改其中一个，不会同时改变两个变量：

```
var x = 5;
var y = x;
y = 6;
console.log(y === x, x, y); //false, 5, 6
```

但对于非原始值，由于两个变量只是保存对同一对象的引用，修改一个变量 **将** 改变另一个变量：

```
var x = { name : 'John Doe' };
var y = x;
y.name = 'Jhon';
console.log(x.name === y.name, x.name, y.name); //true, John, John
```


Chapter 36: Angular $scopes

Section 36.1: A function available in the entire app

Be careful, this approach might be considered as a bad design for angular apps, since it requires programmers to remember both where functions are placed in the scope tree, and to be aware of scope inheritance. In many cases it would be preferred to inject a service (Angular practice - using scope inheritance vs injection.

This example only show how scope inheritance could be used for our needs, and the how you could take advantage of it, and not the best practices of designing an entire app.

In some cases, we could take advantage of scope inheritance, and set a function as a property of the rootScope. This way - all of the scopes in the app (except for isolated scopes) will inherit this function, and it could be called from anywhere in the app.

```
angular.module('app', [])
.run(['$rootScope', function($rootScope){
    var messages = []
    $rootScope.addMessage = function(msg){
        messages.push(msg);
    }
}]);

<div ng-app="app">
    <a ng-click="addMessage('hello world!')">it could be accsessed from here</a>
    <div ng-include="inner.html"></div>
</div>
```

inner.html:

```
<div>
    <button ng-click="addMessage('page')">and from here to!</button>
</div>
```

Section 36.2: Avoid inheriting primitive values

In javascript, assigning a non-primitive value (Such as Object, Array, Function, and many more), keeps a reference (an address in the memory) to the assigned value.

Assigning a primitive value (String, Number, Boolean, or Symbol) to two different variables, and changing one, won't change both:

```
var x = 5;
var y = x;
y = 6;
console.log(y === x, x, y); //false, 5, 6
```

But with a non-primitive value, since both variables are simply keeping references to the same object, changing one variable **will** change the other:

```
var x = { name : 'John Doe' };
var y = x;
y.name = 'Jhon';
console.log(x.name === y.name, x.name, y.name); //true, John, John
```

在 Angular 中，当创建一个作用域时，它会被赋予其父作用域的所有属性。然而，之后更改属性只有在属性是非原始值时才会影响父作用域：

```
angular.module('app', [])
.controller('我的控制器', ['$scope', function($scope){
    $scope.person = { name: '约翰·多伊' }; //非原始值
    $scope.name = '约翰·多伊'; //原始值
}])
.controller('我的控制器1', ['$scope', function($scope){}]);

<div ng-app="app" ng-controller="我的控制器">
绑定到输入框有效: {{person.name}}<br/>
   绑定到输入框无效: {{name}}<br/>
    <div ng-controller="我的控制器1">
        <input ng-model="person.name" />
        <input ng-model="name" />
    </div>
</div>
```

请记住：在 Angular 中，作用域可以通过多种方式创建（例如内置或自定义指令，或 $scope.$new() 函数），并且跟踪作用域树几乎是不可能的。

仅使用非原始值作为作用域属性将使你处于安全状态（除非你需要某个属性不继承，或其他你已知作用域继承的情况）。

第36.3节：$scope继承的基本示例

```
angular.module('app', [])
.controller('我的控制器', ['$scope', function($scope){
    $scope.person = { name: '约翰·多伊' };
}]);

<div ng-app="app" ng-conroller="我的控制器">
    <input ng-model="person.name" />
    <div ng-repeat="number in [0,1,2,3]">
        {{person.name}} {{number}}
    </div>
</div>
```

在这个例子中，ng-repeat 指令为它新创建的每个子元素创建了一个新的作用域。

这些创建的作用域是它们父作用域的子作用域（在本例中是由我的控制器创建的作用域），因此它们继承了所有属性，比如 person。

第36.4节：你如何限制指令的作用域以及为什么要这样做？

作用域被用作我们在父控制器、指令和指令模板之间进行通信的“粘合剂”。每当 AngularJS 应用启动时，会创建一个 rootScope 对象。由控制器、指令和服务创建的每个作用域都是从 rootScope 原型继承的。

是的，我们可以限制指令的作用域。我们可以通过为指令创建一个孤立作用域来实现。

> 指令作用域有三种类型：
>
> 1.作用域：False（指令使用其父作用域）

In angular, when a scope is created, it is assigned all of its parent's properties However, changing properties afterwards will only affect the parent scope if it is a non-primitive value:

```
angular.module('app', [])
.controller('myController', ['$scope', function($scope){
    $scope.person = { name: 'John Doe' }; //non-primitive
    $scope.name = 'Jhon Doe'; //primitive
}])
.controller('myController1', ['$scope', function($scope){}]);

<div ng-app="app" ng-controller="myController">
    binding to input works: {{person.name}}<br/>
    binding to input does not work: {{name}}<br/>
    <div ng-controller="myController1">
        <input ng-model="person.name" />
        <input ng-model="name" />
    </div>
</div>
```

Remember: in Angular scopes can be created in many ways (such as built-in or custom directives, or the `$scope.$new()` function), and keeping track of the scope tree is probably impossible.

Using only non-primitive values as scope properties will keep you on the safe side (unless you need a property to not inherit, or other cases where you are aware of scope inheritance).

Section 36.3: Basic Example of $scope inheritance

```
angular.module('app', [])
.controller('myController', ['$scope', function($scope){
    $scope.person = { name: 'John Doe' };
}]);

<div ng-app="app" ng-conroller="myController">
    <input ng-model="person.name" />
    <div ng-repeat="number in [0,1,2,3]">
        {{person.name}} {{number}}
    </div>
</div>
```

In this example, the ng-repeat directive creates a new scope for each of its newly created children.

These created scopes are children of their parent scope (in this case the scope created by myController), and therfore, they inherit all of its proporties, such as person.

Section 36.4: How can you limit the scope on a directive and why would you do this?

Scope is used as the "glue" that we use to communicate between the parent controller, the directive, and the directive template. Whenever the AngularJS application is bootstrapped, a rootScope object is created. Each scope created by controllers, directives and services are prototypically inherited from rootScope.

Yes, we can limit the scope on a directive . We can do so by creating an isolated scope for directive.

> There are 3 types of directive scopes:
>
> 1. Scope : False (Directive uses its parent scope)

2.作用域：True（指令获得一个新的作用域）
3.作用域：{ }（指令获得一个新的隔离作用域）

具有新隔离作用域的指令：当我们创建一个新的隔离作用域时，它不会继承自父作用域。
这个新作用域被称为隔离作用域，因为它与父作用域完全分离。
为什么？我们应该使用隔离作用域：当我们想创建自定义指令时，应使用隔离作用域，因为它能确保我们的指令是通用的，可以放置在应用程序的任何位置。父作用域不会干扰指令作用域。

隔离作用域示例：

```
var app = angular.module("test",[]);

app.controller("Ctrl1",function($scope){
    $scope.name = "Prateek";
$scope.reverseName = function(){
        $scope.name = $scope.name.split('').reverse().join('');
    };
});
app.directive("myDirective", function(){
    return {
restrict: "EA",
scope: {},
template: "<div>你的名字是：{{name}}</div>"+
        "修改你的名字：<input type='text' ng-model='name'/>"
    };
});
```

AngularJS为隔离作用域提供了3种前缀类型，这些是：

1."@"（文本绑定 / 单向绑定）
2."="（直接模型绑定 / 双向绑定）
3."&"（行为绑定 / 方法绑定）

所有这些前缀都接收来自指令元素属性的数据，例如：

```
<div my-directive
 class="directive"
 name="{{name}}"
 reverse="reverseName()"
 color="color" >
</div>
```

第36.5节：使用$scope函数

虽然在$rootscope中声明函数有其优势，我们也可以在由$scope服务注入的代码的任何部分声明$scope函数。例如控制器。

控制器

```
myApp.controller('我的控制器', ['$scope', function($scope){
    $scope.myFunction = function () {
alert("你在myFunction中！");
    };
}]);
```


2. Scope : True (Directive gets a new scope)
3. Scope : { } (Directive gets a new isolated scope)

Directives with the new isolated scope: When we create a new isolated scope then it will not be inherited from the parent scope. This new scope is called Isolated scope because it is completely detached from its parent scope. Why? should we use isolated scope: We should use isolated scope when we want to create a custom directive because it will make sure that our directive is generic, and placed anywhere inside the application. Parent scope is not going to interfere with the directive scope.

Example of isolated scope:

```
var app = angular.module("test",[]);

app.controller("Ctrl1",function($scope){
    $scope.name = "Prateek";
    $scope.reverseName = function(){
        $scope.name = $scope.name.split('').reverse().join('');
    };
});
app.directive("myDirective", function(){
    return {
        restrict: "EA",
        scope: {},
        template: "<div>Your name is : {{name}}</div>"+
        "Change your name : <input type='text' ng-model='name'/>"
    };
});
```

There're 3 types of prefixes AngularJS provides for isolated scope these are :

1. "@" (Text binding / one-way binding)
2. "=" (Direct model binding / two-way binding)
3. "&" (Behaviour binding / Method binding)

All these prefixes receives data from the attributes of the directive element like :

```
<div my-directive
 class="directive"
 name="{{name}}"
 reverse="reverseName()"
 color="color" >
</div>
```

Section 36.5: Using $scope functions

While declaring a function in the $rootscope has it's advantages, we can also declare a $scope function any part of the code that is injected by the $scope service. Controller, for instance.

Controller

```
myApp.controller('myController', ['$scope', function($scope){
    $scope.myFunction = function () {
        alert("You are in myFunction!");
    };
}]);
```

现在你可以通过控制器调用你的函数：

```
$scope.myFunction();
```

或者通过该特定控制器下的HTML调用：

```
<div ng-controller="myController">
    <button ng-click="myFunction()"> 点击我！ </button>
</div>
```

指令

Angular 指令是你可以使用作用域的另一个地方：

```
myApp.directive('触发函数', function() {
    return {
scope: {
            触发函数: '&'
        },
link: function(scope, element) {
            element.bind('mouseover', function() {
                scope.触发函数();
            });
        }
    };
});
```

在相同控制器下的 HTML 代码中：

```
<div ng-controller="我的控制器">
    <button trigger-function="myFunction()"> 鼠标悬停我！ </button>
</div>
```

当然，你也可以使用 ngMouseover 来实现相同的功能，但指令的特别之处在于你可以根据自己的需求自定义它们。现在你知道如何在指令中使用 $scope 函数了，发挥你的创造力吧！

第36.6节：创建自定义 $scope 事件

像普通的 HTML 元素一样，$scope 也可以拥有自己的事件。可以通过以下方式订阅 $scope 事件：

```
$scope.$on('my-event', function(event, args) {
    console.log(args); // { custom: 'data' }
});
```

如果你需要注销一个事件监听器，$on 函数将返回一个解绑函数。继续以上示例：

```
var unregisterMyEvent = $scope.$on('my-event', function(event, args) {
    console.log(args); // { custom: 'data' }
unregisterMyEvent();
});
```

触发自定义 $scope 事件有两种方式：$broadcast 和 $emit。要通知父级作用域的特定事件，使用 $emit

Now you can call your function from the controller using:

```
$scope.myfunction();
```

Or via HTML that is under that specific controller:

```
<div ng-controller="myController">
    <button ng-click="myFunction()"> Click me! </button>
</div>
```

Directive

An angular directive is another place you can use your scope:

```
myApp.directive('triggerFunction', function() {
    return {
        scope: {
            triggerFunction: '&'
        },
        link: function(scope, element) {
            element.bind('mouseover', function() {
                scope.triggerFunction();
            });
        }
    };
});
```

And in your HTML code under the same controller:

```
<div ng-controller="myController">
    <button trigger-function="myFunction()"> Hover over me! </button>
</div>
```

Of course, you can use ngMouseover for the same thing, but what's special about directives is that you can customize them the way you want. And now you know how to use your $scope functions inside them, be creative!

Section 36.6: Creating custom $scope events

Like normal HTML elements, it is possible for $scopes to have their own events. $scope events can be subscribed to using the following manner:

```
 $scope.$on('my-event', function(event, args) {
    console.log(args); // { custom: 'data' }
});
```

If you need unregister an event listener, the **$on** function will return an unbinding function. To continue with the above example:

```
var unregisterMyEvent = $scope.$on('my-event', function(event, args) {
    console.log(args); // { custom: 'data' }
    unregisterMyEvent();
});
```

There are two ways of triggering your own custom $scope event **$broadcast** and **$emit**. To notify the parent(s) of a scope of a specific event, use **$emit**

belindoc.com

```
$scope.$emit('my-event', { custom: 'data' });
```

上述示例将触发父作用域中所有 my-event 的事件监听器，并会继续向上遍历作用域树直到 $rootScope，除非某个监听器调用了事件的 stopPropagation。只有通过 $emit 触发的事件可以调用 stopPropagation

$emit 的反向操作是 $broadcast，它会触发作用域树中所有调用 $broadcast 的作用域的子作用域上的事件监听器。

```
$scope.$broadcast('my-event', { custom: 'data' });
```

通过 $broadcast 触发的事件无法被取消。

belindoc.com


```
$scope.$emit('my-event', { custom: 'data' });
```

The above example will trigger any event listeners for `my-event` on the parent scope and will continue up the scope tree to **$rootScope** unless a listener calls `stopPropagation` on the event. Only events triggered with **$emit** may call `stopPropagation`

The reverse of **$emit** is **$broadcast**, which will trigger any event listeners on all child scopes in the scope tree that are children of the scope that called **$broadcast**.

```
$scope.$broadcast('my-event', { custom: 'data' });
```

Events triggered with **$broadcast** cannot be canceled.

第37章：在TypeScript中使用AngularJS

第37.1节：使用打包/压缩

在控制器的构造函数中注入$scope的方式，是演示和使用Angular依赖注入基本选项的一种方法，但不适合生产环境，因为它无法被压缩。原因是压缩系统会更改变量名，而Angular的依赖注入是通过参数名来确定需要注入的内容。例如，ExampleController的构造函数被压缩后变成了以下代码。

```
function n(n){this.setUpWatches(n)
```

并且$scope被改成了 n！
为了解决这个问题，我们可以添加一个$inject数组（string[]），这样Angular的依赖注入就知道在控制器构造函数的哪个位置注入什么。
因此，上述TypeScript代码变为

```
module App.Controllers {
    class Address {
        line1: string;
        line2: string;
        city: string;
        state: string;
    }
    export class SampleController {
        firstName: string;
lastName: string;
年龄: 数字;
address: 地址;
setUpWatches($scope: ng.IScope): void {
            $scope.$watch(() => this.firstName, (n, o) => {
                //n 是字符串，o 也是字符串
            });
        };
        static $inject : string[] = ['$scope'];
        constructor($scope: ng.IScope) {
            this.setUpWatches($scope);
        }
    }
}
```

第37.2节：Typescript中的Angular控制器

如AngularJS文档中定义

> 当通过ng-controller指令将控制器附加到DOM时，Angular将使用指定控制器的构造函数实例化一个新的控制器对象。将创建一个新的子作用域
> 并作为可注入参数以$scope的形式传递给控制器的构造函数。

控制器可以非常容易地使用typescript类来创建。

```
模块 App.Controllers {
```


Chapter 37: Using AngularJS with TypeScript

Section 37.1: Using Bundling / Minification

The way the $scope is injected in the controller's constructor functions is a way to demonstrate and use the basic option of angular dependency injection but is not production ready as it cannot be minified. Thats because the minification system changes the variable names and anguar's dependency injection uses the parameter names to know what has to be injected. So for an example the ExampleController's constructor function is minified to the following code.

```
function n(n){this.setUpWatches(n)
```

and `$scope` is changed to `n`!
to overcome this we can add an $inject array(`string[]`). So that angular's DI knows what to inject at what position is the controllers constructor function.
So the above typescript changes to

```
module App.Controllers {
    class Address {
        line1: string;
        line2: string;
        city: string;
        state: string;
    }
    export class SampleController {
        firstName: string;
        lastName: string;
        age: number;
        address: Address;
        setUpWatches($scope: ng.IScope): void {
            $scope.$watch(() => this.firstName, (n, o) => {
                //n is string and so is o
            });
        };
        static $inject : string[] = ['$scope'];
        constructor($scope: ng.IScope) {
            this.setUpWatches($scope);
        }
    }
}
```

Section 37.2: Angular Controllers in Typescript

As defined in the AngularJS Documentation

> When a Controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified Controller's constructor function. A new child scope will be created and made available as an injectable parameter to the Controller's constructor function as $scope.

Controllers can be very easily made using the typescript classes.

```
module App.Controllers {
```

```
    class Address {
        line1: string;
        line2: string;
        city: string;
        state: string;
    }
    export class SampleController {
        firstName: string;
lastName: string;
年龄: 数字;
address: 地址;
setUpWatches($scope: ng.IScope): void {
            $scope.$watch(() => this.firstName, (n, o) => {
                //n 是字符串，o 也是字符串
            });
        };
constructor($scope: ng.IScope) {
            this.setUpWatches($scope);
        }
    }
}
```

生成的 Javascript 是

```
var App;
(function (App) {
    var Controllers;
    (function (Controllers) {
        var Address = (function () {
            function Address() {
            }
            return Address;
        }());
        var SampleController = (function () {
            function SampleController($scope) {
                this.setUpWatches($scope);
            }
SampleController.prototype.setUpWatches = function ($scope) {
                var _this = this;
$scope.$watch(function () { return _this.firstName; }, function (n, o) {
                    //n 是字符串，o 也是字符串
                });
            };
            ;
            return SampleController;
        }());
Controllers.SampleController = SampleController;
    })(Controllers = App.Controllers || (App.Controllers = {}));
})(App || (App = {}));
//# sourceMappingURL=ExampleController.js.map
```

创建控制器类后，可以通过使用该类简单地完成关于控制器的 AngularJS 模块

```
app
.module('app')
.controller('exampleController', App.Controller.SampleController)
```



```
    class Address {
        line1: string;
        line2: string;
        city: string;
        state: string;
    }
    export class SampleController {
        firstName: string;
        lastName: string;
        age: number;
        address: Address;
        setUpWatches($scope: ng.IScope): void {
            $scope.$watch(() => this.firstName, (n, o) => {
                //n is string and so is o
            });
        };
        constructor($scope: ng.IScope) {
            this.setUpWatches($scope);
        }
    }
}
```

The Resulting Javascript is

```
var App;
(function (App) {
    var Controllers;
    (function (Controllers) {
        var Address = (function () {
            function Address() {
            }
            return Address;
        }());
        var SampleController = (function () {
            function SampleController($scope) {
                this.setUpWatches($scope);
            }
            SampleController.prototype.setUpWatches = function ($scope) {
                var _this = this;
                $scope.$watch(function () { return _this.firstName; }, function (n, o) {
                    //n is string and so is o
                });
            };
            ;
            return SampleController;
        }());
        Controllers.SampleController = SampleController;
    })(Controllers = App.Controllers || (App.Controllers = {}));
})(App || (App = {}));
//# sourceMappingURL=ExampleController.js.map
```

After making the controller class let the angular js module about the controller can be done simple by using the class

```
app
 .module('app')
 .controller('exampleController', App.Controller.SampleController)
```

第37.3节：使用带有ControllerAs语法的控制器

我们创建的控制器可以使用controller as语法实例化和使用。这是因为我们直接将变量放在控制器类上，而不是放在$scope上。

使用controller as someName是为了将控制器与$scope本身分离。因此，控制器中不需要注入$scope作为依赖。

传统方式：

```
// 我们使用的是$scope对象。
app.controller('MyCtrl', function ($scope) {
  $scope.name = 'John';
});

<div ng-controller="MyCtrl">
  {{name}}
</div>
```

现在，使用controller as语法：

```
// 我们使用的是"this"对象，而不是"$scope"
app.controller('我的控制器', function() {
  this.name = '约翰';
});

<div ng-controller="MyCtrl as info">
  {{info.name}}
</div>
```

如果你在JavaScript中实例化一个"类"，你可能会这样做：

```
var jsClass = function () {
  this.name = 'John';
}
var jsObj = new jsClass();
```

所以，现在我们可以使用jsObj实例来访问jsClass的任何方法或属性。

在Angular中，我们做同样的事情。我们使用controller as语法来实例化。

第37.4节：为什么使用ControllerAs语法？

控制器函数

控制器函数不过是一个JavaScript构造函数。因此，当视图加载时，函数上下文（this）被设置为控制器对象。

情况1：

```
this.constFunction = function() { ... }
```

它是在controller对象中创建的，而不是在$scope上。视图无法访问定义在controller

对象上的函数。

示例：

Section 37.3: Using the Controller with ControllerAs Syntax

The Controller we have made can be instantiated and used using `controller as` Syntax. That's because we have put variable directly on the controller class and not on the `$scope`.

Using `controller as someName` is to separate the controller from `$scope` itself.So, there is no need of injecting $scope as the dependency in the controller.

Traditional way :

```
// we are using $scope object.
app.controller('MyCtrl', function ($scope) {
  $scope.name = 'John';
});

<div ng-controller="MyCtrl">
  {{name}}
</div>
```

Now, with `controller as` Syntax :

```
// we are using the "this" Object instead of "$scope"
app.controller('MyCtrl', function() {
  this.name = 'John';
});

<div ng-controller="MyCtrl as info">
  {{info.name}}
</div>
```

If you instantiate a "class" in JavaScript, you might do this :

```
var jsClass = function () {
  this.name = 'John';
}
var jsObj = new jsClass();
```

So, now we can use **`jsObj`** instance to access any method or property of **`jsClass`**.

In angular, we do same type of thing.we use controller as syntax for instantiation.

Section 37.4: Why ControllerAs Syntax?

Controller Function

Controller function is nothing but just a JavaScript constructor function. Hence, when a view loads the **`function`** `context`(**`this`**) is set to the controller object.

Case 1 :

```
this.constFunction = function() { ... }
```

It is created in the `controller object`, not on `$scope`. views can not access the functions defined on controller object.

Example :

beli doc.com

```
<a href="#123" ng-click="constFunction()"></a> // 它将无法工作
```

情况2：

```
$scope.scopeFunction = function() { ... }
```

它是在$scope对象中创建的，而不是在controller对象上。视图只能访问定义在
$scope对象上的函数。

示例：

```
<a href="#123" ng-click="scopeFunction()"></a> // 它将可以工作
```

为什么使用ControllerAs？

- ControllerAs语法使得操作对象的位置更加清晰。拥有oneCtrl.name和anotherCtrl.name使得更容易识别你有一个name被多个不同的控制器分配用于不同的目的，但如果两者都使用相同的$scope.name，并且页面上有两个不同的HTML元素都绑定到{{name}}，那么就很难区分哪个是来自哪个控制器。

- 隐藏$scope，并通过中介对象将控制器中的成员暴露给视图。
通过设置this.*，我们可以只暴露想要从控制器到视图暴露的内容。

```
<div ng-controller="FirstCtrl">
   {{ name }}
    <div ng-controller="SecondCtrl">
      {{ name }}
       <div ng-controller="ThirdCtrl">
         {{ name }}
       </div>
    </div>
</div>
```

这里，在上述情况下{{ name }}的使用会非常混乱，我们也不知道哪个对应哪个
控制器。

```
<div ng-controller="FirstCtrl as first">
   {{ first.name }}
    <div ng-controller="SecondCtrl as second">
      {{ second.name }}
       <div ng-controller="ThirdCtrl as third">
         {{ third.name }}
       </div>
    </div>
</div>
```

为什么使用$scope？

- 当你需要访问 $scope 的一个或多个方法时，如 $watch、$digest、$emit、$http 等，请使用 $scope。
- 限制暴露给 $scope 的属性和/或方法，然后根据需要显式地将它们传递给 $scope。


```
<a href="#123" ng-click="constFunction()"></a> // It will not work
```

Case 2 :

```
$scope.scopeFunction = function() { ... }
```

It is created in the `$scope object`, not on `controller object`. views can only access the functions defined on $scope object.

Example :

```
<a href="#123" ng-click="scopeFunction()"></a> // It will work
```

Why ControllerAs ?

- **`ControllerAs`** syntax makes it much clearer where objects are being manipulated.Having `oneCtrl.name` and `anotherCtrl.name` makes it much easier to identify that you have an `name` assigned by multiple different controllers for different purposes but if both used same `$scope.name` and having two different HTML elements on a page which both are bound to `{{name}}` then it is difficult to identify which one is from which controller.

- Hiding the `$scope` and exposing the members from the controller to the view via an `intermediary object`. By setting `this.*`, we can expose just what we want to expose from the controller to the view.

```
<div ng-controller="FirstCtrl">
    {{ name }}
    <div ng-controller="SecondCtrl">
        {{ name }}
        <div ng-controller="ThirdCtrl">
            {{ name }}
        </div>
    </div>
</div>
```

Here, in above case `{{ name }}` will be very confusing to use and We also don't know which one related to which controller.

```
<div ng-controller="FirstCtrl as first">
    {{ first.name }}
    <div ng-controller="SecondCtrl as second">
        {{ second.name }}
        <div ng-controller="ThirdCtrl as third">
            {{ third.name }}
        </div>
    </div>
</div>
```

Why $scope ?

- Use `$scope` when you need to access one or more methods of $scope such as `$watch`, `$digest`, `$emit`, `$http` etc.
- limit which properties and/or methods are exposed to `$scope`, then explicitly passing them to `$scope` as needed.

第38章：$http 请求

第38.1节：$http 请求的时机

$http 请求需要时间，具体时间取决于服务器，有些可能只需几毫秒，有些可能需要几秒钟。通常，从请求中获取数据所需的时间是关键。
假设响应值是一个名字数组，考虑以下示例：

错误示范

```
$scope.names = [];

$http({
method: 'GET',
url: '/someURL'
}).然后(函数 successCallback(响应) {
            $scope.names = 响应.数据;
        },
        function errorCallback(response) {
            alert(response.status);
        });

alert("名字是: " + $scope.names[0]);
```

在 $http 请求下面直接访问 $scope.names[0] 通常会抛出错误——这行代码在服务器响应之前执行。

正确

```
$scope.names = [];

$scope.$watch('names', function(newVal, oldVal) {
    if(!(newVal.length == 0)) {
alert("名字是: " + $scope.names[0]);
    }
});

$http({
method: 'GET',
url: '/someURL'
}).然后(函数 successCallback(响应) {
            $scope.names = 响应.数据;
        },
        function errorCallback(response) {
            alert(response.status);
        });
```

使用 $watch 服务，我们仅在收到响应时访问 $scope.names 数组。在初始化期间，即使 $scope.names 已经初始化，函数仍会被调用，因此需要检查 newVal.length 是否不等于 0。请注意——对 $scope.names 的任何更改都会触发 watch 函数。

第38.2节：在控制器中使用 $http

$http 服务是一个生成 HTTP 请求并返回一个 promise 的函数。

Chapter 38: $http request

Section 38.1: Timing of an $http request

The $http requests require time which varies depending on the server, some may take a few milliseconds, and some may take up to a few seconds. Often the time required to retrieve the data from a request is critical. Assuming the response value is an array of names, consider the following example:

Incorrect

```
$scope.names = [];

$http({
    method: 'GET',
    url: '/someURL'
}).then(function successCallback(response) {
            $scope.names = response.data;
        },
        function errorCallback(response) {
            alert(response.status);
        });

alert("The first name is: " + $scope.names[0]);
```

Accessing `$scope.names[0]` right below the $http request will often throw an error - this line of code executes before the response is received from the server.

Correct

```
$scope.names = [];

$scope.$watch('names', function(newVal, oldVal) {
    if(!(newVal.length == 0)) {
        alert("The first name is: " + $scope.names[0]);
    }
});

$http({
    method: 'GET',
    url: '/someURL'
}).then(function successCallback(response) {
            $scope.names = response.data;
        },
        function errorCallback(response) {
            alert(response.status);
        });
```

Using the $watch service we access the `$scope.names` array only when the response is received. During initialization, the function is called even though `$scope.names` was initialized before, therefore checking if the `newVal.length` is different than 0 is necessary. Be aware - any changes made to `$scope.names` will trigger the watch function.

Section 38.2: Using $http inside a controller

The `$http` service is a function which generates an HTTP request and returns a promise.

通用用法

```
// 简单的 GET 请求示例：
$http({
method: 'GET',
url: '/someUrl'
}).then(function successCallback(response) {
    // 当响应可用时，这个回调函数将被异步调用

  }, function errorCallback(response) {
    // 如果发生错误，或者服务器返回错误状态的响应，将
    异步调用此函数。
  });
```

控制器内的用法

```
appName.controller('controllerName',
    ['$http', function($http){

    // 简单的 GET 请求示例：
$http({
method: 'GET',
url: '/someUrl'
    }).then(function successCallback(response) {
        // 当响应可用时，这个回调函数将被异步调用

    }, function errorCallback(response) {
        // 如果发生错误，或者服务器返回错误状态的响应，将
        异步调用此函数。
    });
}])
```

快捷方法

$http 服务也有快捷方法。请在此处阅读 http 方法

语法

```
$http.get('/someUrl', config).then(successCallback, errorCallback);
$http.post('/someUrl', data, config).then(successCallback, errorCallback);
```

快捷方法

- $http.get
- $http.head
- $http.post
- $http.put
- $http.delete
- $http.jsonp
- $http.patch

第38.3节：在服务中使用$http请求

HTTP请求在每个网络应用中被广泛且反复使用，因此为每个常用请求编写一个方法，然后在应用的多个地方使用它是明智的。

创建一个`httpRequestsService.js`

General Usage

```
// Simple GET request example:
$http({
  method: 'GET',
  url: '/someUrl'
}).then(function successCallback(response) {
    // this callback will be called asynchronously
    // when the response is available
  }, function errorCallback(response) {
    // called asynchronously if an error occurs
    // or server returns response with an error status.
  });
```

Usage inside controller

```
appName.controller('controllerName',
    ['$http', function($http){

    // Simple GET request example:
    $http({
        method: 'GET',
        url: '/someUrl'
    }).then(function successCallback(response) {
        // this callback will be called asynchronously
        // when the response is available
    }, function errorCallback(response) {
        // called asynchronously if an error occurs
        // or server returns response with an error status.
    });
}])
```

Shortcut Methods

`$http` service also has shortcut methods. Read about http methods here

Syntax

```
$http.get('/someUrl', config).then(successCallback, errorCallback);
$http.post('/someUrl', data, config).then(successCallback, errorCallback);
```

Shortcut Methods

- $http.get
- $http.head
- $http.post
- $http.put
- $http.delete
- $http.jsonp
- $http.patch

Section 38.3: Using $http request in a service

HTTP requests are widely used repeatedly across every web app, so it is wise to write a method for each common request, and then use it in multiple places throughout the app.

Create a `httpRequestsService.js`

httpRequestsService.js

```
appName.service('httpRequestsService', function($q, $http){

    return {
        // 执行基本的get请求的函数
       getName: function(){
            // 确保注入了$http
            return $http.get("/someAPI/names")
              .then(function(response) {
                    // 以promise形式返回结果
                    return response;
                }, function(response) {
                    // 拒绝promise
                    return $q.reject(response.data);
                });
        },

        // 为应用程序的其他请求添加函数
addName: function(){
            // 一些代码...
        }
    }
})
```

上述服务将在服务内部执行get请求。只要注入了该服务，任何控制器都可以使用它。

示例用法

```
appName.controller('controllerName',
    ['httpRequestsService', function(httpRequestsService){

        // 我们在这个控制器中注入了 httpRequestsService 服务
        // 使得 getName() 函数可以使用。
httpRequestsService.getName()
            .then(function(response){
                    // 成功
              }, function(error){
                    // 处理错误
              })
    }])
```

使用这种方法，我们现在可以在任何时候、任何控制器中使用 httpRequestsService.js。

httpRequestsService.js

```
appName.service('httpRequestsService', function($q, $http){

    return {
        // function that performs a basic get request
        getName: function(){
            // make sure $http is injected
            return $http.get("/someAPI/names")
                .then(function(response) {
                    // return the result as a promise
                    return response;
                }, function(response) {
                    // defer the promise
                    return $q.reject(response.data);
                });
        },

        // add functions for other requests made by your app
        addName: function(){
            // some code...
        }
    }
})
```

The service above will perform a get request inside the service. This will be available to any controller where the service has been injected.

Sample usage

```
appName.controller('controllerName',
    ['httpRequestsService', function(httpRequestsService){

        // we injected httpRequestsService service on this controller
        // that made the getName() function available to use.
        httpRequestsService.getName()
            .then(function(response){
                // success
            }, function(error){
                // do something with the error
            })
    }])
```

Using this approach we can now use **httpRequestsService.js** anytime and in any controller.

第39章：为生产环境准备 - Grunt

第39.1节：视图预加载

当第一次请求视图时，通常 Angular 会发起 XHR 请求来获取该视图。对于中型项目，
视图数量可能相当多，这会降低应用的响应速度。

对于小型和中型项目，良好的做法是一次性预加载所有视图。对于大型项目，最好将视图按有意义的批次进行聚合，但也可以使用其他方法来分担负载。为了自动化此任务，使用 Grunt 或 Gulp 任务非常方便。

要预加载视图，我们可以使用 $templateCache 对象。该对象是 Angular 用来存储从服务器接收的每个视图的地方。

可以使用 html2js 模块，将所有视图转换成一个模块 - js 文件。然后我们只需将该模块注入到应用中即可。

要创建所有视图的合并文件，我们可以使用此任务

```
module.exports = function (grunt) {
 //在这里设置视图的位置
 var viewLocation = ['app/views/**.html'];

 grunt.initConfig({
pkg: require('./package.json'),
            //设置将多个html文件合并成一个文件的配置部分
html2js: {
options: {
base: '',
                        module: 'app.templates', //新模块名称
                    singleModule: true,
useStrict: true,
                        htmlmin: {
collapseBooleanAttributes: true,
                            collapseWhitespace: true
                        }
                    },
main: {
                        src: viewLocation,
                        dest: 'build/app.templates.js'
                    }
                },
        //本节用于监视视图文件的更改，如果有更改，将重新生成生产文件。此任务在开发过程中非常有用。

watch: {
views:{
                    files: viewLocation,
                    tasks: ['buildHTML']
                },
            }
        });

        //自动生成一个视图文件
grunt.loadNpmTasks('grunt-html2js');

        //监听文件变化，如果文件被修改，则重新生成文件
grunt.loadNpmTasks('grunt-contrib-watch');
```

belindoc.com

Chapter 39: Prepare for Production - Grunt

Section 39.1: View preloading

When the first time view is requested, normally Angular makes XHR request to get that view. For mid-size projects, the view count can be significant and it can slow down the application responsiveness.

The **good practice is to pre-load** all the views at once for small and mid size projects. For larger projects it is good to aggregate them in some meaningful bulks as well, but some other methods can be handy to split the load. To automate this task it is handy to use Grunt or Gulp tasks.

To pre-load the views, we can use `$templateCache` object. That is an object, where angular stores every received view from the server.

It is possible to use `html2js` module, that will convert all our views to one module - js file. Then we will need to inject that module into our application and that's it.

To create concatenated file of all the views we can use this task

```
module.exports = function (grunt) {
 //set up the location of your views here
 var viewLocation = ['app/views/**.html'];

 grunt.initConfig({
       pkg: require('./package.json'),
           //section that sets up the settings for concatenation of the html files into one file
           html2js: {
                       options: {
                           base: '',
                           module: 'app.templates', //new module name
                           singleModule: true,
                           useStrict: true,
                           htmlmin: {
                               collapseBooleanAttributes: true,
                               collapseWhitespace: true
                           }
                       },
                       main: {
                           src: viewLocation,
                           dest: 'build/app.templates.js'
                       }
                   },
           //this section is watching for changes in view files, and if there was a change, it will
regenerate the production file. This task can be handy during development.
           watch: {
               views:{
                   files: viewLocation,
                   tasks: ['buildHTML']
               },
           }
       });

       //to automatically generate one view file
       grunt.loadNpmTasks('grunt-html2js');

       //to watch for changes and if the file has been changed, regenerate the file
       grunt.loadNpmTasks('grunt-contrib-watch');
```

```
        //仅是一个带有友好名称的任务，供watch引用
grunt.registerTask('buildHTML', ['html2js']);
};
```

要使用这种合并方式，你需要做两个更改：在你的`index.html`文件中需要引用合并后的视图文件

```
<script src="build/app.templates.js"></script>
```

在文件中，声明你的应用时，需要注入依赖

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

如果你使用流行的路由器如ui-router，引用模板的方式没有变化

```
.state('home', {
url: '/home',
views: {
            "@": {
controller: 'homeController',
                //这将从$templateCache中获取
templateUrl: 'app/views/home.html'
            },
        }

    })
```

第39.2节：脚本优化

将JS文件合并并压缩是良好的实践。对于较大的项目，可能有数百个JS文件，分别从服务器加载每个文件会增加不必要的延迟。

对于Angular的压缩，要求所有函数都必须注释。这是为了Angular依赖注入的正确压缩。（压缩过程中，函数名和变量会被重命名，如果不采取额外措施，会破坏依赖注入。）

在压缩过程中，`$scope` 和 `myService` 变量将被替换为其他值。Angular 依赖注入是基于名称工作的，因此，这些名称不应更改

```
.controller('我的控制器', function($scope, myService){
})
```

Angular 会理解数组表示法，因为压缩不会替换字符串字面量。

```
.controller('我的控制器', ['$scope','myService', function($scope, myService){
}])
```

- 首先我们会将所有文件依次连接起来。
- 其次我们会使用`ng-annotate`模块，该模块会为压缩准备代码
- 最后我们会应用`uglify`模块。

module.exports = function (grunt) { //设置脚本的位置以便在代码中复用 var scriptLocation = ['app/scripts/*.js'];


```
        //just a task with friendly name to reference in watch
        grunt.registerTask('buildHTML', ['html2js']);
};
```

To use this way of concatination, you need to make 2 changes: In your `index.html` file you need to reference the concatenated view file

```
<script src="build/app.templates.js"></script>
```

In the file, where you are declaring your app, you need to inject the dependency

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

If you are using popular routers like `ui-router`, there are no changes in the way, how you are referencing templates

```
    .state('home', {
        url: '/home',
        views: {
            "@": {
                controller: 'homeController',
                //this will be picked up from $templateCache
                templateUrl: 'app/views/home.html'
            },
        }

    })
```

Section 39.2: Script optimisation

It is **good practice to combine JS files together** and minify them. For larger project there could be hundreds of JS files and it adds unnecessary latency to load each file separately from the server.

For angular minification it is required to to have all functions annotated. That in necessary for Angular dependency injection proper minificaiton. (During minification, function names and variables will be renamed and it will break dependency injection if no extra actions will be taken.)

During minificaiton `$scope` and `myService` variables will be replaced by some other values. Angular dependency injection works based on the name, as a result, these names shouldn't change

```
.controller('myController', function($scope, myService){
})
```

Angular will understand the array notation, because minification won't replace string literals.

```
.controller('myController', ['$scope','myService', function($scope, myService){
}])
```

- Firstly we will concatinate all files end to end.
- Secondly we will use `ng-annotate` module, that will prepare code for minification
- Finally we will apply `uglify` module.

module.exports = function (grunt) { //set up the location of your scripts here for reusing it in code var scriptLocation = ['app/scripts/*.js'];

Bellindoc.com

```
grunt.initConfig({
pkg: require('./package.json'),
            //添加必要的注释以保证安全压缩
ngAnnotate: {
angular: {
src: ['staging/concatenated.js'],
                dest: 'staging/anotated.js'
            }
        },
        //将所有文件合并成一个文件
concat: {
js: {
src: scriptLocation,
                    dest: 'staging/concatenated.js'
                }
            },
        //最终混淆压缩
uglify: {
            options: {
                report: 'min',
                mangle: false,
                sourceMap:true
            },
my_target: {
文件: {
                    'build/app.min.js': ['staging/anotated.js']
                }
            }
        },

        //此部分用于监视JS文件的更改，如果有更改，将重新生成生产文件。你可以选择不这样做，但我喜欢保持合并版本的最新状态

watch: {
scripts: {
                files: scriptLocation,
                tasks: ['buildJS']
            }
        }

});

    //使文件更难阅读的模块
grunt.loadNpmTasks('grunt-contrib-uglify');

    //合并文件的模块
grunt.loadNpmTasks('grunt-contrib-concat');

    //使AngularJS文件准备好进行压缩的模块
grunt.loadNpmTasks('grunt-ng-annotate');

    //监听文件变化，如果文件被修改，则重新生成文件
grunt.loadNpmTasks('grunt-contrib-watch');

    // 任务，按顺序执行所有步骤以准备生产用的JS文件
    // 合并所有JS文件
    // 注释JS文件（为压缩做准备）
    // 压缩文件
grunt.registerTask('buildJS', ['concat:js', 'ngAnnotate', 'uglify']);
};
```

bellmdoc.com


```
grunt.initConfig({
        pkg: require('./package.json'),
            //add necessary annotations for safe minification
         ngAnnotate: {
            angular: {
                src: ['staging/concatenated.js'],
                dest: 'staging/anotated.js'
            }
        },
        //combines all the files into one file
        concat: {
                js: {
                    src: scriptLocation,
                    dest: 'staging/concatenated.js'
                }
            },
        //final uglifying
        uglify: {
            options: {
                report: 'min',
                mangle: false,
                sourceMap:true
            },
            my_target: {
                files: {
                    'build/app.min.js': ['staging/anotated.js']
                }
            }
        },

        //this section is watching for changes in JS files, and if there was a change, it will regenerate the production file. You can choose not to do it, but I like to keep concatenated version up to date
        watch: {
            scripts: {
                files: scriptLocation,
                tasks: ['buildJS']
            }
        }

});

    //module to make files less readable
    grunt.loadNpmTasks('grunt-contrib-uglify');

    //mdule to concatenate files together
    grunt.loadNpmTasks('grunt-contrib-concat');

    //module to make angularJS files ready for minification
    grunt.loadNpmTasks('grunt-ng-annotate');

    //to watch for changes and if the file has been changed, regenerate the file
    grunt.loadNpmTasks('grunt-contrib-watch');

    //task that sequentially executes all steps to prepare JS file for production
    //concatinate all JS files
    //annotate JS file (prepare for minification
    //uglify file
     grunt.registerTask('buildJS', ['concat:js', 'ngAnnotate', 'uglify']);
};
```

第40章：Grunt任务

第40.1节：本地运行应用程序

> 以下示例要求已安装Node.js且可用npm。
> 完整的工作代码可从GitHub分叉，地址为https://github.com/mikkoviitala/angular-grunt-run-local

通常，在开发新的网络应用时，首先要做的事情之一就是让它能在本地运行。

下面你会找到一个完整的示例，使用grunt（JavaScript任务运行器）、npm（Node包管理器）和bower（另一种包管理器）来实现这一点。

除了您的实际应用文件外，您还需要使用上述工具安装一些第三方依赖。在您的项目目录中，最好是在根目录，您需要三个（3）文件。

- package.json（由 npm 管理的依赖）
- bower.json（由 bower 管理的依赖）
- gruntfile.js（grunt 任务）

所以你的项目目录看起来是这样的：

beliindoc.com

package.json

我们将安装**grunt**本身，**matchdep**来简化我们的工作，允许我们按名称过滤依赖，**grunt-express**用于通过 grunt 启动 express 网络服务器，以及**grunt-open**用于从 grunt 任务打开 URL/文件。

所以这些包都是关于"基础设施"和我们将构建应用程序所依赖的辅助工具。

```
{
  "name": "app",
  "version": "1.0.0",
  "dependencies": {},
  "devDependencies": {
    "grunt": "~0.4.1",
    "matchdep": "~0.1.2",
    "grunt-express": "~1.0.0-beta2",
    "grunt-open": "~0.2.1"
  },
  "scripts": {
    "postinstall": "bower install"
  }
}
```

bower.json

Bower 是（或者至少应该是）专注于前端的工具，我们将使用它来安装angular。

Chapter 40: Grunt tasks

Section 40.1: Run application locally

> Following example requires that node.js is installed and npm is available.
> Full working code can be forked from GitHub @ https://github.com/mikkoviitala/angular-grunt-run-local

Usually one of the first things you want to do when developing new web application is to make it run locally.

Below you'll find complete example achieving just that, using grunt (javascript task runner), npm (node package manager) and bower (yet another package manager).

Beside your actual application files you'll need to install few 3rd party dependencies using tools mentioned above. In your project directory, **preferably root**, you'll need three (3) files.

- package.json (dependencies managed by npm)
- bower.json (dependencies managed by bower)
- gruntfile.js (grunt tasks)

So your project directory looks like so:

package.json

We'll be installing **grunt** itself, **matchdep** to make our life easier allowing us to filter dependencies by name, **grunt-express** used to start express web server via grunt and **grunt-open** to open urls/files from a grunt task.

So these packages are all about "infrastructure" and helpers we'll be building our application on.

```
{
  "name": "app",
  "version": "1.0.0",
  "dependencies": {},
  "devDependencies": {
    "grunt": "~0.4.1",
    "matchdep": "~0.1.2",
    "grunt-express": "~1.0.0-beta2",
    "grunt-open": "~0.2.1"
  },
  "scripts": {
    "postinstall": "bower install"
  }
}
```

bower.json

Bower is (or at least should be) all about front-end and we'll be using it to install **angular**.

```
{
  "name": "app",
  "version": "1.0.0",
  "dependencies": {
    "angular": "~1.3.x"
  },
  "devDependencies": {}
}
```

gruntfile.js

在 gruntfile.js 中，我们将拥有实际的“本地运行应用程序”的魔法，它会在新的浏览器窗口中打开我们的应用程序，运行地址为 http://localhost:9000/

```
'use strict';

// 参见 http://rhumaric.com/2013/07/renewing-the-grunt-livereload-magic/

module.exports = function(grunt) {
require('matchdep').filterDev('grunt-*').forEach(grunt.loadNpmTasks);

  grunt.initConfig({
express: {
all: {
options: {
          port: 9000,
          hostname: 'localhost',
          bases: [__dirname]
        }
      }
    },

open: {
all: {
path: 'http://localhost:<%= express.all.options.port%>'
        }
      }
   });

grunt.registerTask('app', [
    'express',
    'open',
    'express-keepalive'
  ]);
};
```

用法

要从零开始启动您的应用程序，请将上述文件保存到项目的根目录（任何空文件夹均可）。然后打开控制台/命令行，输入以下命令以安装所有所需的依赖项。

```
npm install -g grunt-cli bower
npm install
```

然后使用以下命令运行你的应用程序

```
grunt app
```


bellindoc.com

```
{
  "name": "app",
  "version": "1.0.0",
  "dependencies": {
    "angular": "~1.3.x"
  },
  "devDependencies": {}
}
```

gruntfile.js

Inside gruntfile.js we'll have the actual "running application locally" magic, which opens our application in new browser window, running on http://localhost:9000/

```
'use strict';

// see http://rhumaric.com/2013/07/renewing-the-grunt-livereload-magic/

module.exports = function(grunt) {
  require('matchdep').filterDev('grunt-*').forEach(grunt.loadNpmTasks);

  grunt.initConfig({
    express: {
      all: {
        options: {
          port: 9000,
          hostname: 'localhost',
          bases: [__dirname]
        }
      }
    },

    open: {
      all: {
        path: 'http://localhost:<%= express.all.options.port%>'
      }
    }
  });

  grunt.registerTask('app', [
    'express',
    'open',
    'express-keepalive'
  ]);
};
```

Usage

To get your application up & running from scratch, save above files to your project's root directory (any empty folder will do). Then fire up console/command line and type in the following to install all required dependencies.

```
npm install -g grunt-cli bower
npm install
```

And then run your application using

```
grunt app
```

请注意，是的，你也需要你的实际应用程序文件。
有关几乎最小示例，请浏览本示例开头提到的GitHub 仓库。

它们的结构并没有太大区别。只有一个`index.html`模板，angular 代码在`app.js`中，样式在`app.css`中。其他文件是用于 Git 和编辑器配置以及一些通用内容。试试看吧！

AngularJS application
Hello Stack Overflow Documentation (beta)

belindoc.com

Note that yes, you'll be needing your actual application files, too.
For almost-minimal example browse GitHub repository mentioned in beginning of this example.

There structure ain't that different. There's just `index.html` template, angular code in `app.js` and few styles in `app.css`. Other files are for Git and editor configuration and some generic stuff. Give it a try!

AngularJS application
Hello Stack Overflow Documentation (beta)

第41章：懒加载

第41.1节：为懒加载准备你的项目

在你的 index 文件中引入oclazyload.js后，在app.js中声明ocLazyLoad作为依赖

```
//确保你添加了正确的依赖！它的拼写与服务不同！
angular.module('app', [
 'oc.lazyLoad',
 'ui-router'
])
```

第41.2节：用法

为了实现文件的懒加载，将$ocLazyLoad服务注入到控制器或其他服务中

```
.controller('someCtrl', function($ocLazyLoad) {
 $ocLazyLoad.load('path/to/file.js').then(...);
});
```

Angular模块将自动加载到angular中。

其他变体：

```
$ocLazyLoad.load([
   'bower_components/bootstrap/dist/js/bootstrap.js',
   'bower_components/bootstrap/dist/css/bootstrap.css',
   'partials/template1.html'
]);
```

完整的变体列表请访问官方文档

第41.3节：与路由器的使用

UI-Router：

```
.state('profile', {
 url: '/profile',
 controller: 'profileCtrl as vm'
 resolve: {
module: function($ocLazyLoad) {
   return $ocLazyLoad.load([
    'path/to/profile/module.js',
    'path/to/profile/style.css'
   ]);
  }
 }
});
```

ngRoute：

```
.when('/profile', {
     controller: 'profileCtrl as vm'
     resolve: {
module: function($ocLazyLoad) {
       return $ocLazyLoad.load([
        'path/to/profile/module.js',
```


Chapter 41: Lazy loading

Section 41.1: Preparing your project for lazy loading

After including `oclazyload.js` in your index file, declare `ocLazyLoad` as a dependency in `app.js`

```
//Make sure you put the correct dependency! it is spelled different than the service!
angular.module('app', [
 'oc.lazyLoad',
 'ui-router'
])
```

Section 41.2: Usage

In order to lazily load files inject the `$ocLazyLoad` service into a controller or another service

```
.controller('someCtrl', function($ocLazyLoad) {
 $ocLazyLoad.load('path/to/file.js').then(...);
});
```

Angular modules will be automatically loaded into angular.

Other variation:

```
$ocLazyLoad.load([
  'bower_components/bootstrap/dist/js/bootstrap.js',
  'bower_components/bootstrap/dist/css/bootstrap.css',
  'partials/template1.html'
]);
```

For a complete list of variations visit the official documentation

Section 41.3: Usage with router

UI-Router:

```
.state('profile', {
 url: '/profile',
 controller: 'profileCtrl as vm'
 resolve: {
  module: function($ocLazyLoad) {
   return $ocLazyLoad.load([
    'path/to/profile/module.js',
    'path/to/profile/style.css'
   ]);
  }
 }
});
```

ngRoute:

```
.when('/profile', {
     controller: 'profileCtrl as vm'
     resolve: {
      module: function($ocLazyLoad) {
       return $ocLazyLoad.load([
        'path/to/profile/module.js',
```

```
        'path/to/profile/style.css'
      ]);
    }
  }
});
```

第41.4节：使用依赖注入

以下语法允许您在module.js中指定依赖项，而不是在使用服务时显式指定

```
//lazy_module.js
angular.module('lazy', [
 'alreadyLoadedDependency1',
 'alreadyLoadedDependency2',
 ...
 [
  'path/to/lazily/loaded/dependency.js',
  'path/to/lazily/loaded/dependency.css'
 ]
]);
```

注意：此语法仅适用于懒加载模块！

第41.5节：使用指令

```
<div oc-lazy-load="['path/to/lazy/loaded/directive.js', 'path/to/lazy/loaded/directive.html']">

<!-- myDirective 在此可用 -->
<my-directive></my-directive>

</div>
```

belindoc.com


```
        'path/to/profile/style.css'
      ]);
    }
  }
});
```

Section 41.4: Using dependency injection

The following syntax allows you to specify dependencies in your `module.js` instead of explicit specification when using the service

```
//lazy_module.js
angular.module('lazy', [
 'alreadyLoadedDependency1',
 'alreadyLoadedDependency2',
 ...
 [
  'path/to/lazily/loaded/dependency.js',
  'path/to/lazily/loaded/dependency.css'
 ]
]);
```

Note: this syntax will only work for lazily loaded modules!

Section 41.5: Using the directive

```
<div oc-lazy-load="['path/to/lazy/loaded/directive.js', 'path/to/lazy/loaded/directive.html']">

<!-- myDirective available here -->
<my-directive></my-directive>

</div>
```

第42章：HTTP 拦截器

AngularJS 的 $http 服务允许我们与后端通信并发起 HTTP 请求。有些情况下，我们希望捕获每个请求并在发送到服务器之前进行处理。另一些时候，我们希望捕获响应并在完成调用之前进行处理。全局 HTTP 错误处理也是这类需求的一个很好的例子。拦截器正是为这些情况而创建的。

第42.1节：通用 httpInterceptor 逐步讲解

创建一个包含以下内容的HTML文件：

```
<!DOCTYPE html>
<html>
<head>
  <title>Angular 拦截器示例</title>
  <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
  <script src="app.js"></script>
  <script src="appController.js"></script>
  <script src="genericInterceptor.js"></script>
</head>
<body ng-app="interceptorApp">
    <div ng-controller="appController as vm">
        <button ng-click="vm.sendRequest()">发送请求</button>
    </div>
</body>
</html>
```

添加一个名为 'app.js' 的JavaScript文件：

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

interceptorApp.config(function($httpProvider) {
$httpProvider.interceptors.push('genericInterceptor');
});
```

再添加一个名为 'appController.js' 的文件：

```
(function() {
    'use strict';

    function appController($http) {
        var vm = this;

vm.sendRequest = function(){
            $http.get('http://google.com').then(function(response){
                console.log(response);
            });
        };
    }

angular.module('interceptorApp').controller('appController',['$http', appController]);
})();
```

最后是包含拦截器本身的文件 'genericInterceptor.js'：

```
(function() {
    "use strict";
```


Chapter 42: HTTP Interceptor

The $http service of AngularJS allows us to communicate with a backend and make HTTP requests. There are cases where we want to capture every request and manipulate it before sending it to the server. Other times we would like to capture the response and process it before completing the call. Global http error handling can be also a good example of such need. Interceptors are created exactly for such cases.

Section 42.1: Generic httpInterceptor step by step

Create an HTML file with the following content:

```
<!DOCTYPE html>
<html>
<head>
  <title>Angular Interceptor Sample</title>
  <script src="https://code.angularjs.org/1.5.8/angular.min.js"></script>
  <script src="app.js"></script>
  <script src="appController.js"></script>
  <script src="genericInterceptor.js"></script>
</head>
<body ng-app="interceptorApp">
    <div ng-controller="appController as vm">
        <button ng-click="vm.sendRequest()">Send a request</button>
    </div>
</body>
</html>
```

Add a JavaScript file called 'app.js':

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

interceptorApp.config(function($httpProvider) {
        $httpProvider.interceptors.push('genericInterceptor');
});
```

Add another one called 'appController.js':

```
(function() {
    'use strict';

    function appController($http) {
        var vm = this;

        vm.sendRequest = function(){
            $http.get('http://google.com').then(function(response){
                console.log(response);
            });
        };
    }

    angular.module('interceptorApp').controller('appController',['$http', appController]);
})();
```

And finally the file containing the interceptor itself 'genericInterceptor.js':

```
(function() {
    "use strict";
```

```
    function genericInterceptor($q) {
        this.responseError = function (response) {
            return $q.reject(response);
        };

        this.requestError = function(request){
            if (canRecover(rejection)) {
                return responseOrNewPromise
            }
            return $q.reject(rejection);
        };

        this.response = function(response){
            return response;
        };

        this.request = function(config){
            return config;
        }
    }

angular.module('interceptorApp').service('genericInterceptor', genericInterceptor);
})();
```

“genericInterceptor”涵盖了我们可以重写以向应用程序添加额外行为的可能功能。

第42.2节：入门

Angular内置的$http服务允许我们发送HTTP请求。通常情况下，需要在请求之前或之后执行某些操作，例如为每个请求添加身份验证令牌或创建通用的错误处理逻辑。

第42.3节：使用http拦截器在响应上显示闪现消息

在视图文件中

在基础html（index.html）中，通常我们会包含Angular脚本或应用程序共享的html，留一个空的div元素，闪现消息将显示在该div元素内

```
<div class="flashmessage" ng-if="isVisible">
    {{flashMessage}}
</div>
```

脚本文件

在Angular模块的config方法中，注入httpProvider，httpProvider有一个interceptor数组属性，将自定义拦截器推入其中，在当前示例中，自定义拦截器仅拦截响应并调用附加到rootScope的方法。

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

    interceptorTest.config(['$httpProvider',function ($httpProvider) {

        $httpProvider.interceptors.push(["$rootScope",function ($rootScope) {
            return {     // 仅拦截响应
```



```
    function genericInterceptor($q) {
        this.responseError = function (response) {
            return $q.reject(response);
        };

        this.requestError = function(request){
            if (canRecover(rejection)) {
                return responseOrNewPromise
            }
            return $q.reject(rejection);
        };

        this.response = function(response){
            return response;
        };

        this.request = function(config){
            return config;
        }
    }

    angular.module('interceptorApp').service('genericInterceptor', genericInterceptor);
})();
```

The 'genericInterceptor' cover the possible functions which we can override adding extra behavior to our application.

Section 42.2: Getting Started

Angular's builtin $http service allows us to send HTTP requests. Oftentime, the need arise to do things before or after a request, for example adding to each request an authentication token or creating a generic error handling logic.

Section 42.3: Flash message on response using http interceptor

In the view file

In the base html (index.html) where we usually include the angular scripts or the html that is shared across the app, leave an empty div element, the flash messages will be appearing inside this div element

```
<div class="flashmessage" ng-if="isVisible">
    {{flashMessage}}
</div>
```

Script File

In the config method of angular module, inject the httpProvider, the httpProvider has an interceptor array property, push the custom interceptor, In the current example the custom interceptor intercepts only the response and calls a method attached to rootScope.

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

    interceptorTest.config(['$httpProvider',function ($httpProvider) {

        $httpProvider.interceptors.push(["$rootScope",function ($rootScope) {
            return {     //intercept only the response
```

```
                        'response': function (response)
                                    {

$rootScope.showFeedBack(response.status,response.data.message);

                                      return response;
                                    }
                        };
        }]);

    }])
```

由于只有提供者（providers）可以注入到 Angular 模块的 config 方法中（即 httpProvider 而不是 rootscope），因此将附加到 rootscope 的方法声明在 Angular 模块的 run 方法中。

同时在 $timeout 内显示消息，这样消息将具有闪现属性，即在阈值时间后消失。在我们的示例中为 3000 毫秒。

```
interceptorTest.run(["$rootScope","$timeout",function($rootScope,$timeout){
     $rootScope.showFeedBack = function(status,message){

$rootScope.isVisible = true;
        $rootScope.flashMessage = message;
        $timeout(function(){$rootScope.isVisible = false },3000)
    }
}]);
```

常见陷阱

尝试在 Angular 模块的 config 方法中注入 $rootScope 或任何其他服务，Angular 应用的生命周期不允许这样做，会抛出未知提供者错误。只有 providers 可以注入到 Angular 模块的 config 方法中

bindoc.com


```
                        'response': function (response)
                                    {

$rootScope.showFeedBack(response.status,response.data.message);

                                      return response;
                                    }
                        };
        }]);

    }])
```

Since only providers can be injected into the config method of an angular module (that is httpProvider and not the rootscope), declare the method attached to rootscope inside the run method of angular module.

Also display the message inside $timeout so that the message will have the flash property, that is disappearing after a threshold time. In our example its 3000 ms.

```
interceptorTest.run(["$rootScope","$timeout",function($rootScope,$timeout){
     $rootScope.showFeedBack = function(status,message){

        $rootScope.isVisible = true;
        $rootScope.flashMessage = message;
        $timeout(function(){$rootScope.isVisible = false },3000)
    }
}]);
```

Common pitfalls

Trying to inject **$rootScope or any other services** inside **config** method of angular module, the lifecycle of angular app doesn't allow that and unknown provider error will be thrown. Only **providers** can be injected in **config** method of the angular module

第43章：会话存储

第43.1节：使用 AngularJS 通过服务处理会话存储

会话存储服务：

通用工厂服务，根据键保存并返回已保存的会话数据。

```
'use strict';

/**
 * @ngdoc factory
 * @name app.factory:storageService
* @description 该函数将通过工厂服务与 HTML5 sessionStorage 进行通信。
 */

app.factory('storageService', ['$rootScope', function($rootScope) {

    return {
        get: function(key) {
            return sessionStorage.getItem(key);
        },
save: function(key, data) {
            sessionStorage.setItem(key, data);
        }
    };
}]);
```

在控制器中：

在控制器中注入storageService依赖，以便从会话存储中设置和获取数据。

```
app.controller('myCtrl',['storageService',function(storageService) {

  // 将会话数据保存到storageService
storageService.save('key', 'value');

  // 从storageService获取已保存的会话数据
  var sessionData = storageService.get('key');

});
```


Chapter 43: Session storage

Section 43.1: Handling session storage through service using angularjs

Session storage service :

Common factory service that will save and return the saved session data based on the key.

```
'use strict';

/**
 * @ngdoc factory
 * @name app.factory:storageService
 * @description This function will communicate with HTML5 sessionStorage via Factory Service.
 */

app.factory('storageService', ['$rootScope', function($rootScope) {

    return {
        get: function(key) {
            return sessionStorage.getItem(key);
        },
        save: function(key, data) {
            sessionStorage.setItem(key, data);
        }
    };
}]);
```

In controller :

Inject the storageService dependency in the controller to set and get the data from the session storage.

```
app.controller('myCtrl',['storageService',function(storageService) {

  // Save session data to storageService
  storageService.save('key', 'value');

  // Get saved session data from storageService
  var sessionData = storageService.get('key');

});
```

第44章：Angular MVC

在**AngularJS**中，**MVC**模式通过JavaScript和HTML实现。视图定义在HTML中，
而模型和控制器则通过JavaScript实现。在AngularJS中，这些组件可以通过多种方式组合，
但最简单的形式是从视图开始。

第44.1节：带控制器的静态视图

mvc演示
你好，世界

第44.2节：控制器函数定义

```
var indexController = myApp.controller("indexController", function ($scope) {
    // 应用逻辑写在这里
});
```

第44.3节：向模型添加信息

```
var indexController = myApp.controller("indexController", function ($scope) {
    // 控制器逻辑写在这里
$scope.message = "Hello Hacking World"
});
```

belindoc.com

Chapter 44: Angular MVC

In **AngularJS** the **MVC** pattern is implemented in JavaScript and HTML. The view is defined in HTML, while the model and controller are implemented in JavaScript. There are several ways that these components can be put together in AngularJS but the simplest form starts with the view.

Section 44.1: The Static View with controller

mvc demo
Hello World

Section 44.2: Controller Function Definition

```
var indexController = myApp.controller("indexController", function ($scope) {
    // Application logic goes here
});
```

Section 44.3: Adding information to the model

```
var indexController = myApp.controller("indexController", function ($scope) {
    // controller logic goes here
    $scope.message = "Hello Hacking World"
});
```

第45章：使用AngularJS的SignalR

本文重点介绍“如何使用AngularJS和SignalR创建一个简单项目”，在本教程中你需要
了解“如何使用AngularJS创建应用”、“如何创建/使用Angular服务”以及SignalR的基础知识，
为此我们推荐https://www.codeproject.com/Tips/590660/Introduction-to-SignalR)。

第45.1节：SignalR和AngularJS [聊天项目]

步骤1：创建项目

```
-应用程序
      -app.js
      -控制器
            -appController.js
      -工厂
            -SignalR-factory.js
-index.html
-脚本
      -angular.js
      -jquery.js
      -jquery.signalR.min.js
-集线器
```

> SignalR 版本使用：signalR-2.2.1

步骤 2：Startup.cs 和 ChatHub.cs

进入你的 "/Hubs" 目录并添加两个文件 [Startup.cs, ChatHub.cs]

Startup.cs

```
using Microsoft.Owin;
using Owin;
[assembly: OwinStartup(typeof(SignalR.Hubs.Startup))]

namespace SignalR.Hubs
{
public class Startup
    {
public void Configuration(IAppBuilder app)
        {
app.MapSignalR();
        }
    }
}
```

ChatHub.cs

```
using Microsoft.AspNet.SignalR;

namespace SignalR.Hubs
{
public class ChatHub : Hub
    {
public void Send(string name, string message, string time)
        {
```


Chapter 45: SignalR with AngularJS

In this article we focus on "How to create a simple project using AngularJS And SignalR", in this training you need to know about "how create app with angularjs", "how to create/use service on angular" And basic knowledge about SignalR" for this we recommend https://www.codeproject.com/Tips/590660/Introduction-to-SignalR).

Section 45.1: SignalR and AngularJS [ChatProject]

step 1: Create Project

```
- Application
    - app.js
    - Controllers
        - appController.js
    - Factories
        - SignalR-factory.js
- index.html
- Scripts
    - angular.js
    - jquery.js
    - jquery.signalR.min.js
- Hubs
```

> SignalR version use: signalR-2.2.1

Step 2: Startup.cs And ChatHub.cs

Go to your *"/Hubs"* directory and Add 2 files *[Startup.cs, ChatHub.cs]*

Startup.cs

```
using Microsoft.Owin;
using Owin;
[assembly: OwinStartup(typeof(SignalR.Hubs.Startup))]

namespace SignalR.Hubs
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.MapSignalR();
        }
    }
}
```

ChatHub.cs

```
using Microsoft.AspNet.SignalR;

namespace SignalR.Hubs
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message, string time)
        {
```

```
Clients.All.broadcastMessage(name, message, time);
        }
    }
}
```

步骤3：创建Angular应用

进入你的"/Application"目录并添加[app.js]文件

app.js

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

步骤4：创建SignalR工厂

进入你的"/Application/Factories"目录并添加[SignalR-factory.js]文件

SignalR-factory.js

```
app.factory("signalR", function () {
    var factory = {};

factory.url = function (url) {
        $.connection.hub.url = url;
    }

factory.setHubName = function (hubName) {
        factory.hub = hubName;
    }

factory.connectToHub = function () {
        return $.connection[factory.hub];
    }

factory.client = function () {
        var hub = factory.connectToHub();
        return hub.client;
    }

factory.server = function () {
        var hub = factory.connectToHub();
        return hub.server;
    }

factory.start = function (fn) {
        return $.connection.hub.start().done(fn);
    }

    return factory;
});
```

步骤5：更新 app.js

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

app.run(function(signalR) {
signalR.url("http://localhost:21991/signalr");
});
```



```
            Clients.All.broadcastMessage(name, message, time);
        }
    }
}
```

step 3: create angular app

Go to your *"/Application"* directory and Add *[app.js]* file

app.js

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

step 4: create SignalR Factory

Go to your *"/Application/Factories"* directory and Add *[SignalR-factory.js]* file

SignalR-factory.js

```
app.factory("signalR", function () {
    var factory = {};

    factory.url = function (url) {
        $.connection.hub.url = url;
    }

    factory.setHubName = function (hubName) {
        factory.hub = hubName;
    }

    factory.connectToHub = function () {
        return $.connection[factory.hub];
    }

    factory.client = function () {
        var hub = factory.connectToHub();
        return hub.client;
    }

    factory.server = function () {
        var hub = factory.connectToHub();
        return hub.server;
    }

    factory.start = function (fn) {
        return $.connection.hub.start().done(fn);
    }

    return factory;
});
```

step 5: update app.js

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

app.run(function(signalR) {
    signalR.url("http://localhost:21991/signalr");
});
```

> **localhost:21991/signalr | 这是你的 SignalR Hubs 地址**

步骤6：添加控制器

进入你的 "/Application/Controllers" 目录并添加 [appController.js] 文件

```
app.controller("ctrl", function ($scope, signalR) {
    $scope.messages = [];
$scope.user = {};

signalR.setHubName("chatHub");

signalR.client().broadcastMessage = function (name, message, time) {
        var newChat = { name: name, message: message, time: time };

$scope.$apply(function() {
            $scope.messages.push(newChat);
        });
    };

signalR.start(function () {
$scope.send = function () {
            var dt = new Date();
            var time = dt.getHours() + ":" + dt.getMinutes() + ":" + dt.getSeconds();

            signalR.server().send($scope.user.name, $scope.user.message, time);
        }
    });
});
```

> signalR.setHubName("chatHub") | [ChatHub] (public class) > ChatHub.cs

> 注意：不要使用大写字母插入HubName，首字母应为小写。

> **signalR.client()** | 此方法尝试连接到你的集线器并获取集线器中的所有函数，在此示例中我们有“chatHub”，用来获取“broadcastMessage()”函数；

步骤7：在目录路径中添加index.html

index.html

```
<!DOCTYPE html>
<html ng-app="app" ng-controller="ctrl">
<head>
    <meta charset="utf-8" />
    <title>SignalR 简易聊天</title>
</head>
<body>
    <form>
        <input type="text" placeholder="姓名" ng-model="user.name" />
        <input type="text" placeholder="消息" ng-model="user.message" />
        <button ng-click="send()">发送</button>
```


> localhost:21991/signalr | **this is your SignalR Hubs Urls**

step 6: add controller

Go to your *"/Application/Controllers"* directory and Add *[appController.js]* file

```
app.controller("ctrl", function ($scope, signalR) {
    $scope.messages = [];
    $scope.user = {};

    signalR.setHubName("chatHub");

    signalR.client().broadcastMessage = function (name, message, time) {
        var newChat = { name: name, message: message, time: time };

        $scope.$apply(function() {
            $scope.messages.push(newChat);
        });
    };

    signalR.start(function () {
        $scope.send = function () {
            var dt = new Date();
            var time = dt.getHours() + ":" + dt.getMinutes() + ":" + dt.getSeconds();

            signalR.server().send($scope.user.name, $scope.user.message, time);
        }
    });
});
```

> signalR.setHubName("chatHub") | **[ChatHub] (public class) >** *ChatHub.cs*

> **Note:** do not insert *HubName* with upper Case, **first letter** is lower Case.

> **signalR.client()** | this method try to connect to your hubs and get all functions in the Hubs, in this sample we have "chatHub", to get "broadcastMessage()" function;

step 7: add index.html in route of directory

index.html

```
<!DOCTYPE html>
<html ng-app="app" ng-controller="ctrl">
<head>
    <meta charset="utf-8" />
    <title>SignalR Simple Chat</title>
</head>
<body>
    <form>
        <input type="text" placeholder="name" ng-model="user.name" />
        <input type="text" placeholder="message" ng-model="user.message" />
        <button ng-click="send()">send</button>
```

```
        <ul>
            <li ng-repeat="item in messages">
                <b ng-bind="item.name"></b> <small ng-bind="item.time"></small> : {{item.message}}
            </li>
        </ul>
    </form>

    <script src="Scripts/angular.min.js"></script>
    <script src="Scripts/jquery-1.6.4.min.js"></script>
    <script src="Scripts/jquery.signalR-2.2.1.min.js"></script>
    <script src="signalr/hubs"></script>
    <script src="app.js"></script>
    <script src="SignalR-factory.js"></script>
</body>
</html
```

> 带图像的结果

> 用户 1（发送和接收）
>
> 用户 2（发送和接收）

belindoc.com


```
        <ul>
            <li ng-repeat="item in messages">
                <b ng-bind="item.name"></b> <small ng-bind="item.time"></small> : {{item.message}}
            </li>
        </ul>
    </form>

    <script src="Scripts/angular.min.js"></script>
    <script src="Scripts/jquery-1.6.4.min.js"></script>
    <script src="Scripts/jquery.signalR-2.2.1.min.js"></script>
    <script src="signalr/hubs"></script>
    <script src="app.js"></script>
    <script src="SignalR-factory.js"></script>
</body>
</html
```

> Result with Image

> User 1 (send and receive)
>
> User 2 (send and receive)

第46章：迁移到 Angular 2+

AngularJS 已完全使用 TypeScript 语言重写，并重命名为 Angular。

可以对 AngularJS 应用做很多工作以简化迁移过程。正如官方升级指南所说，可以执行若干“准备步骤”来重构您的应用，使其更好且更接近新的 Angular 风格。

第46.1节：将您的 AngularJS 应用转换为组件导向结构

在新的Angular框架中，组件是构成用户界面的主要构建块。因此，帮助AngularJS应用迁移到新Angular的第一步之一是将其重构为更面向组件的结构。

从旧的 AngularJS 版本 1.5+ 开始也引入了组件。在 AngularJS 应用中使用组件不仅会使其结构更接近新的 Angular 2+，还会使其更加模块化且更易于维护。

在深入之前，我建议查看 官方 AngularJS 文档页面关于组件，其中详细解释了它们的优点和用法。

我更愿意提一些关于如何将旧的 ng-controller 导向代码转换为新的 component 导向风格的技巧。

开始将你的应用拆分成组件

所有面向组件的应用通常都有一个或几个包含其他子组件的组件。那么为什么不创建第一个组件，简单地包含你的应用（或其大部分）呢？

假设我们有一段代码分配给一个名为 UserListController 的控制器，我们想把它做成一个组件，命名为 UserListComponent。

当前的 HTML：

```
<div ng-controller="UserListController as listctrl">
    <ul>
        <li ng-repeat="user in myUserList">
          {{ user }}
        </li>
    </ul>
</div>
```

当前 JavaScript：

```
app.controller("UserListController", function($scope, SomeService) {

    $scope.myUserList = ['Shin', 'Helias', 'Kalhac'];

    this.someFunction = function() {
        // ...
    }

    // ...
```


Chapter 46: Migration to Angular 2+

AngularJS has been totally rewritten using the TypeScript language and renamed to just Angular.

There is a lot that can be done to an AngularJS app to ease the migration process. As the official upgrade guide says, several "preparation steps" can be performed to refactor your app, making it better and closer to the new Angular style.

Section 46.1: Converting your AngularJS app into a componend-oriented structure

In the new Angular framework, **Components** are the main building blocks that compose the user interface. So one of the first steps that helps an AngularJS app to be migrated to the new Angular is to refactor it into a more component-oriented structure.

Components were also introduced in the old AngularJS starting from version **1.5+**. Using Components in an AngularJS app will not only make its structure closer to the new Angular 2+, but it will also make it more modular and easier to maintain.

Before going further I recommend to look at the official AngularJS documentation page about Components, where their advantages and usage are well explained.

I would rather mention some tips about how to convert the old `ng-controller` oriented code to the new `component` oriented style.

Start breaking your your app into components

All the component-oriented apps have typically one or few components that include other sub-components. So why not creating the first component which simply will contain your app (or a big piece of it).

Assume that we have a piece of code assigned to a controller, named `UserListController`, and we want to make a component of it, which we'll name `UserListComponent`.

current HTML:

```
<div ng-controller="UserListController as listctrl">
    <ul>
        <li ng-repeat="user in myUserList">
            {{ user }}
        </li>
    </ul>
</div>
```

current JavaScript:

```
app.controller("UserListController", function($scope, SomeService) {

    $scope.myUserList = ['Shin', 'Helias', 'Kalhac'];

    this.someFunction = function() {
        // ...
    }

    // ...
```

```
}
```

新的 HTML：

```
<user-list></user-list>
```

新的 JavaScript：

```
app.component("UserList", {
    templateUrl: 'user-list.html',
    controller: UserListController
});

function UserListController(SomeService) {

    this.myUserList = ['Shin', 'Helias', 'Kalhac'];

    this.someFunction = function() {
        // ...
    }

    // ...
}
```

注意我们不再将$scope注入控制器函数，而是声明了**this**。myUserList
而不是$scope.myUserList；

新的模板文件user-list.component.html：

```
<ul>
    <li ng-repeat="user in $ctrl.myUserList">
       {{ user }}
    </li>
</ul>
```

注意我们现在从html中使用$ctrl.myUserList来引用属于控制器的变量myUserList，而不是$scope.myUserList。

这是因为，正如你在阅读文档后可能已经猜到的，模板中的$ctrl现在指的是控制器函数。

控制器和路由呢？

如果你的控制器是通过路由系统绑定到模板的，而不是通过ng-controller绑定，那么如果你有如下内容：

```
$stateProvider
.state('users', {
    url: '/users',
templateUrl: 'user-list.html',
    controller: 'UserListController'
  })
  // ..
```

你可以将你的状态声明改为：


```
}
```

new HTML:

```
<user-list></user-list>
```

new JavaScript:

```
app.component("UserList", {
    templateUrl: 'user-list.html',
    controller: UserListController
});

function UserListController(SomeService) {

    this.myUserList = ['Shin', 'Helias', 'Kalhac'];

    this.someFunction = function() {
        // ...
    }

    // ...
}
```

Note how we are no longer injecting `$scope` into the controller function and we are now declaring **this**.`myUserList` instead of `$scope.myUserList`;

new template file `user-list.component.html`:

```
<ul>
    <li ng-repeat="user in $ctrl.myUserList">
        {{ user }}
    </li>
</ul>
```

Note how we are now referring to the variable `myUserList`, which belongs to the controller, using `$ctrl.myUserList` from the html instead of `$scope.myUserList`.

That is because, as you probably figured out after reading the documentation, `$ctrl` in the template now refers to the controller function.

What about controllers and routes?

In case your controller was bound to the template using the routing system instead of `ng-controller`, so if you have something like this:

```
$stateProvider
  .state('users', {
    url: '/users',
    templateUrl: 'user-list.html',
    controller: 'UserListController'
  })
  // ..
```

you can just change your state declaration to:

belindoc.com

```
$stateProvider
    .state('users', {
      url: '/',
template: '<user-list></user-list>'
    })
    // ..
```

接下来做什么？

现在你已经有了一个包含你的应用程序的组件（无论它包含整个应用程序还是其中的一部分，比如一个视图），你应该开始将你的组件拆分成多个嵌套组件，通过将其部分内容包装成新的子组件，依此类推。

你应该开始使用组件的功能，比如

- **输入**和**输出**绑定
- **生命周期钩子**例如`$onInit()`，`$onChanges()`等...

阅读了组件文档后，你应该已经知道如何使用所有这些组件功能，但如果你需要一个真实简单应用的具体示例，可以查看这个。

此外，如果你的组件控制器中有一些包含大量逻辑代码的函数，一个好主意是考虑将这些逻辑移到服务中。

结论

采用基于组件的方法使您的 AngularJS 更接近迁移到新的 Angular 框架，但这也使其更好且更加模块化。

当然，您还可以采取许多其他步骤，进一步向新的 Angular 2+ 方向发展，以下示例中我将列出这些步骤。

第46.2节：引入 Webpack 和 ES6 模块

通过使用模块加载器如Webpack，我们可以利用ES6（以及TypeScript）中内置的模块系统。然后我们可以使用import和export功能，指定哪些代码片段将在应用程序的不同部分之间共享。

当我们将应用程序投入生产时，模块加载器还使得将所有内容打包成包含所有依赖的生产包变得更容易。


```
$stateProvider
    .state('users', {
      url: '/',
      template: '<user-list></user-list>'
    })
    // ..
```

What's next?

Now that you have a component containing your app (whether it contains the entire application or a part of it, like a view), you should now start to break your component into multiple nested components, by wrapping parts of it into new sub-components, and so on.

You should start using the Component features like

- **Inputs** and **Outputs** bindings
- **lifecycle hooks** such as `$onInit()`, `$onChanges()`, etc...

After reading the Component documentation you should already know how to use all those component features, but if you need a concrete example of a real simple app, you can check this.

Also, if inside your component's controller you have some functions that hold a lot of logic code, a good idea can be considering to move that logic into services.

Conclusion

Adopting a component-based approach pushes your AngularJS one step closer to migrate it to the new Angular framework, but it also makes it better and much more modular.

Of course there are a lot of other steps you can do to go further into the new Angular 2+ direction, which I will list in the following examples.

Section 46.2: Introducing Webpack and ES6 modules

By using a ***module loader*** like Webpack we can benefit the built-in module system available in **ES6** (as well as in **TypeScript**). We can then use the import and export features that allow us to specify what pieces of code can we are going to share between different parts of the application.

When we then take our applications into production, module loaders also make it easier to package them all up into production bundles with batteries included.

第47章：带有数据过滤、分页等功能的 AngularJS

AngularJS 中关于提供者示例及使用过滤器、分页等显示数据的查询。

第47.1节：AngularJS 使用过滤器和分页显示数据

```
<div ng-app="MainApp" ng-controller="SampleController">
    <input ng-model="dishName" id="search" class="form-control" placeholder="过滤文本">
    <ul>
        <li dir-paginate="dish in dishes |  filter : dishName | itemsPerPage: pageSize" pagination-id="flights">{{dish}}</li>
    </ul>
    <dir-pagination-controls boundary-links="true" on-page-change="changeHandler(newPageNumber)" pagination-id="flights"></dir-pagination-controls>

<script type="text/javascript" src="angular.min.js"></script>
<script type="text/javascript" src="pagination.js"></script>
<script type="text/javascript">

var MainApp = angular.module('MainApp', ['angularUtils.directives.dirPagination'])
MainApp.controller('SampleController', ['$scope', '$filter', function ($scope, $filter) {

    $scope.pageSize = 5;

$scope.dishes = [
    '面条',
'香肠',
    '烤豆',
    '芝士汉堡',
    '炸火星棒',
    '薯片三明治',
'约克郡布丁',
    '维也纳炸肉排',
    '酸菜配蛋',
    '沙拉',
'洋葱汤',
    '白菜',
    '鳄梨卷'
     ];

$scope.changeHandler = function (newPage) { };
}]);
</script>
```

belindoc.com

Chapter 47: AngularJS with data filter, pagination etc

Provider example and query about display data with filter, pagination etc in Angularjs.

Section 47.1: AngularJS display data with filter, pagination

```
<div ng-app="MainApp" ng-controller="SampleController">
    <input ng-model="dishName" id="search" class="form-control" placeholder="Filter text">
    <ul>
        <li dir-paginate="dish in dishes |  filter : dishName | itemsPerPage: pageSize" pagination-id="flights">{{dish}}</li>
    </ul>
    <dir-pagination-controls boundary-links="true" on-page-change="changeHandler(newPageNumber)" pagination-id="flights"></dir-pagination-controls>
</div>
<script type="text/javascript" src="angular.min.js"></script>
<script type="text/javascript" src="pagination.js"></script>
<script type="text/javascript">

var MainApp = angular.module('MainApp', ['angularUtils.directives.dirPagination'])
MainApp.controller('SampleController', ['$scope', '$filter', function ($scope, $filter) {

    $scope.pageSize = 5;

    $scope.dishes = [
    'noodles',
    'sausage',
    'beans on toast',
    'cheeseburger',
    'battered mars bar',
    'crisp butty',
    'yorkshire pudding',
    'wiener schnitzel',
    'sauerkraut mit ei',
    'salad',
    'onion soup',
    'bak choi',
    'avacado maki'
    ];

    $scope.changeHandler = function (newPage) { };
}]);
</script>
```

第48章：性能分析与性能优化

第48.1节：7个简单的性能改进

1) 尽量少用 ng-repeat

在视图中使用ng-repeat通常会导致性能较差，尤其是在存在嵌套ng-repeat的情况下。

这非常慢！

```
<div ng-repeat="user in userCollection">
  <div ng-repeat="details in user">
    {{details}}
  </div>
</div>
```

尽量避免嵌套的 repeats。提高ng-repeat性能的一种方法是使用trackby $index（或其他某个id字段）。默认情况下，ng-repeat会跟踪整个对象。使用track by时，Angular只通过$index或对象id来监视对象。

```
<div ng-repeat="user in userCollection track by $index">
  {{user.data}}
</div>
```

尽可能使用其他方法如分页、虚拟滚动、无限滚动或limitTo: begin，避免遍历大型集合。

2) 绑定一次

Angular具有双向数据绑定功能。如果使用过多，会导致性能变慢。

性能变慢

```
<!-- 默认数据绑定有性能开销 -->
<div>{{ my.data }}</div>
```

更快的性能（AngularJS >= 1.3）

```
<!-- 单次绑定速度更快 -->
<div>{{ ::my.data }}</div>

<div ng-bind="::my.data"></div>

<!-- 在只需要列表显示的 ng-repeat 中使用单次绑定表示法 -->
<div ng-repeat="user in ::userCollection">
  {{::user.data}}
</div>
```

使用“单次绑定”表示法告诉 Angular 在第一次一系列 digest 循环后等待值稳定。Angular 会在 DOM 中使用该值，然后移除所有观察者，使其成为静态值，不再绑定到模型。

该{{}}要慢得多。

Chapter 48: Profiling and Performance

Section 48.1: 7 Simple Performance Improvements

1) Use ng-repeat sparingly

Using `ng-repeat` in views generally results in poor performance, particularly when there are nested `ng-repeat`'s.

This is super slow!

```
<div ng-repeat="user in userCollection">
  <div ng-repeat="details in user">
    {{details}}
  </div>
</div>
```

Try to avoid nested repeats as much as possible. One way to improve the performance of `ng-repeat` is to use `track by $index` (or some other id field). By default, `ng-repeat` tracks the whole object. With `track by`, Angular watches the object only by the `$index` or object id.

```
<div ng-repeat="user in userCollection track by $index">
  {{user.data}}
</div>
```

Use other approaches like pagination, virtual scrolls, infinite scrolls or limitTo: begin whenever possible to avoid iterating over large collections.

2) Bind once

Angular has bidirectional data binding. It comes with a cost of being slow if used too much.

Slower Performance

```
<!-- Default data binding has a performance cost -->
<div>{{ my.data }}</div>
```

Faster Performance (AngularJS >= 1.3)

```
<!-- Bind once is much faster -->
<div>{{ ::my.data }}</div>

<div ng-bind="::my.data"></div>

<!-- Use single binding notation in ng-repeat where only list display is needed  -->
<div ng-repeat="user in ::userCollection">
  {{::user.data}}
</div>
```

Using the "bind once" notation tells Angular to wait for the value to stabilize after the first series of digest cycles. Angular will use that value in the DOM, then remove all watchers so that it becomes a static value and is no longer bound to the model.

The {{}} is much slower.

这个ng-bind是一个指令，会在传入的变量上放置一个观察者。因此，只有当传入的值实际发生变化时，ng-bind才会生效。

另一方面，括号会在每次$digest中进行脏检查并刷新，即使没有必要。

3）作用域函数和过滤器会消耗时间

AngularJS 有一个 digest 循环。你的所有函数都在视图中，过滤器会在每次 digest 循环运行时执行。每当模型更新时，digest 循环都会执行，这可能会减慢你的应用（过滤器在页面加载前可能会被多次调用）。

避免这样做：

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.calculateMe()}}</p>
  <p>{{calc.data | heavyFilter}}</p>
</div>
```

更好的方法

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.preCalculatedValue}}</p>
  <p>{{calc.data | lightFilter}}</p>
</div>
```

控制器可以是：

```
app.controller('bigCalulations', function(valueService) {
    // 不好，因为这会在每个消化循环中调用
    this.calculateMe = function() {
        var t = 0;
        for(i = 0; i < 1000; i++) {
            t += i;
        }
        return t;
    }
    // 好，因为这只执行一次，且逻辑分离到服务中，保持
控制器轻量
    this.preCalulatedValue = valueService.valueCalculation(); // 返回 499500
});
```

4) 监听器

观察者会极大地降低性能。观察者越多，digest 循环所需时间越长，用户界面将会变慢。如果观察者检测到变化，它将启动 digest 循环并重新渲染视图。

在 Angular 中，有三种方法可以手动监听变量变化。

$watch() - 监听值的变化

`$watchCollection()` - 监听集合的变化（监听的范围比常规的`$watch`更多）

`$watch(..., true)` - **尽量避免使用**，因为它会执行“深度监听”，并且会降低性能（监听的范围比`watchCollection`更多）

注意，如果你在视图中绑定变量，就会创建新的监听器——使用{{::variable}}来防止

This `ng-bind` is a directive and will place a watcher on the passed variable. So the `ng-bind` will only apply, when the passed value does actually change.

The brackets on the other hand will be dirty checked and refreshed in every `$digest`, even if it's not necessary.

3) Scope functions and filters take time

AngularJS has a digest loop. All your functions are in a view and filters are executed every time the digest cycle runs. The digest loop will be executed whenever the model is updated and it can slow down your app (filter can be hit multiple times before the page is loaded).

Avoid this:

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.calculateMe()}}</p>
  <p>{{calc.data | heavyFilter}}</p>
</div>
```

Better approach

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.preCalculatedValue}}</p>
  <p>{{calc.data | lightFilter}}</p>
</div>
```

Where the controller can be:

```
app.controller('bigCalulations', function(valueService) {
    // bad, because this is called in every digest loop
    this.calculateMe = function() {
        var t = 0;
        for(i = 0; i < 1000; i++) {
            t += i;
        }
        return t;
    }
    // good, because this is executed just once and logic is separated in service to    keep the
controller light
    this.preCalulatedValue = valueService.valueCalculation(); // returns 499500
});
```

4) Watchers

Watchers tremendously drop performance. With more watchers, the digest loop will take longer and the UI will slow down. If the watcher detects change, it will kick off the digest loop and re-render the view.

There are three ways to do manual watching for variable changes in Angular.

`$watch()` - watches for value changes

`$watchCollection()` - watches for changes in collection (watches more than regular `$watch`)

`$watch(..., true)` - **Avoid this** as much as possible, it will perform "deep watch" and will decline the performance (watches more than `watchCollection`)

Note that if you are binding variables in the view you are creating new watches - use `{{::variable}}` to prevent

创建监听器，尤其是在循环中。

因此，你需要跟踪你使用了多少监听器。你可以用这个脚本来统计监听器数量（感谢@Words Like Jared Number of watchers）

```
(function() {
    var root = angular.element(document.getElementsByTagName('body')),
        watchers = [],
f = function(element) {
angular.forEach(['$scope', '$isolateScope'], function(scopeProperty) {
            if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
                angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
                watchers.push(watcher);
                });
            }
        });

angular.forEach(element.children(), function(childElement) {
            f(angular.element(childElement));
        });
    };

f(root);

    // 删除重复的观察者
    var watchersWithoutDuplicates = [];
    angular.forEach(watchers, function(item) {
        if(watchersWithoutDuplicates.indexOf(item) < 0) {
            watchersWithoutDuplicates.push(item);
        }
    });
console.log(watchersWithoutDuplicates.length);
})();
```

5) ng-if / ng-show

这些功能的行为非常相似。 ng-if 会从DOM中移除元素，而 ng-show 只是隐藏元素但保留所有处理程序。如果你有不想显示的代码部分，使用 ng-if。

这取决于使用类型，但通常其中一个比另一个更合适。

- 如果不需要该元素，使用 `ng-if`
- 快速切换开/关，使用 `ng-show/ng-hide`

```
<div ng-repeat="user in userCollection">
  <p ng-if="user.hasTreeLegs">我很特别<!-- 一些复杂的DOM --></p>
  <p ng-show="user.hasSubscribed">我很棒<!-- 开关此设置 --></p>
</div>
```

如果有疑问 - 使用ng-if并进行测试！

6) 禁用调试

默认情况下，绑定指令和作用域会在代码中保留额外的类和标记，以协助各种调试工具。禁用此选项意味着在消化周期中不再渲染这些元素。

creating a watch, especially in loops.

As a result you need to track how many watchers you are using. You can count the watchers with this script (credit to @Words Like Jared Number of watchers)

```
(function() {
    var root = angular.element(document.getElementsByTagName('body')),
        watchers = [],
        f = function(element) {
        angular.forEach(['$scope', '$isolateScope'], function(scopeProperty) {
            if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
                angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
                watchers.push(watcher);
                });
            }
        });

        angular.forEach(element.children(), function(childElement) {
            f(angular.element(childElement));
        });
    };

    f(root);

    // Remove duplicate watchers
    var watchersWithoutDuplicates = [];
    angular.forEach(watchers, function(item) {
        if(watchersWithoutDuplicates.indexOf(item) < 0) {
            watchersWithoutDuplicates.push(item);
        }
    });
    console.log(watchersWithoutDuplicates.length);
})();
```

5) ng-if / ng-show

These functions are very similar in behavior. `ng-if` removes elements from the DOM while `ng-show` only hides the elements but keeps all handlers. If you have parts of the code you do not want to show, use `ng-if`.

It depends on the type of usage, but often one is more suitable than the other.

- If the element is not needed, use `ng-if`
- To quickly toggle on/off, use `ng-show/ng-hide`

```
<div ng-repeat="user in userCollection">
  <p ng-if="user.hasTreeLegs">I am special<!-- some complicated DOM --></p>
  <p ng-show="user.hasSubscribed">I am awesome<!-- switch this setting on and off --></p>
</div>
```

If in doubt - use `ng-if` and test!

6) Disable debugging

By default, bind directives and scopes leave extra classes and markup in the code to assist with various debugging tools. Disabling this option means that you no longer render these various elements during the digest cycle.

```
angular.module('exampleApp', []).config(['$compileProvider', function ($compileProvider) {
    $compileProvider.debugInfoEnabled(false);
}]);
```

7) 使用依赖注入来暴露你的资源

依赖注入是一种软件设计模式，其中对象被赋予其依赖项，而不是对象自己创建它们。它旨在消除硬编码的依赖，使得可以在需要时更改它们。

你可能会担心对所有可注入函数进行字符串解析所带来的性能开销。
Angular 通过在第一次之后缓存 $inject 属性来处理这个问题。因此，这不会在每次调用函数时发生。

专业提示：如果你追求最佳性能，选择 $inject 属性注解方法。该方法完全避免了函数定义的解析，因为这段逻辑被包装在 annotate 函数中的以下检查内：if (!($inject = fn.$inject))。如果 $inject 已经存在，则无需解析！

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

var DemoController = function (s, h) {
h.get('https://api.github.com/users/angular/repos').success(function (repos) {
s.repos = repos;
    });
}
// $inject 属性注解
DemoController['$inject'] = ['$scope', '$http'];

app.controller('DemoController', DemoController);
```

专业提示 2：你可以在与 ng-app 相同的元素上添加 ng-strict-di 指令，以启用严格 DI 模式，该模式会在服务尝试使用隐式注解时抛出错误。示例：

```
<html ng-app="DemoApp" ng-strict-di>
```

或者如果你使用手动引导：

```
angular.bootstrap(document, ['DemoApp'], {
    strictDi: true
});
```

第48.2节：绑定一次

Angular 以其出色的双向数据绑定而闻名。默认情况下，Angular 会在模型或视图组件中的数据发生变化时，持续同步绑定在模型和视图组件之间的值。

如果使用过多，这会带来一些速度上的代价。性能影响会更大：

性能差：{{my.data}}

在变量名前添加两个冒号 : :以使用一次性绑定。在这种情况下，值只会在my.data被定义时更新一次。你明确表示不监视数据变化。Angular不会执行任何值检查，从而减少每个digest周期中被评估的表达式数量。


```
angular.module('exampleApp', []).config(['$compileProvider', function ($compileProvider) {
    $compileProvider.debugInfoEnabled(false);
}]);
```

7) Use dependency injection to expose your resources

Dependency Injection is a software design pattern in which an object is given its dependencies, rather than the object creating them itself. It is about removing the hard-coded dependencies and making it possible to change them whenever needed.

You might wonder about the performance cost associated with such string parsing of all injectable functions. Angular takes care of this by caching the $inject property after the first time. So this doesn't happen every time a function needs to be invoked.

PRO TIP: If you are looking for the approach with the best performance, go with the $inject property annotation approach. This approach entirely avoids the function definition parsing because this logic is wrapped within the following check in the annotate function: if (!($inject = fn.$inject)). If $inject is already available, no parsing required!

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

var DemoController = function (s, h) {
    h.get('https://api.github.com/users/angular/repos').success(function (repos) {
        s.repos = repos;
    });
}
// $inject property annotation
DemoController['$inject'] = ['$scope', '$http'];

app.controller('DemoController', DemoController);
```

PRO TIP 2: You can add an `ng-strict-di` directive on the same element as `ng-app` to opt into strict DI mode which will throw an error whenever a service tries to use implicit annotations. Example:

```
<html ng-app="DemoApp" ng-strict-di>
```

Or if you use manual bootstrapping:

```
angular.bootstrap(document, ['DemoApp'], {
    strictDi: true
});
```

Section 48.2: Bind Once

Angular has reputation for having awesome bidirectional data binding. By default, Angular continuously synchronizes values bound between model and view components any time data changes in either the model or view component.

This comes with a cost of being a bit slow if used too much. This will have a larger performance hit:

Bad performance: `{{my.data}}`

Add two colons : : before the variable name to use one-time binding. In this case, the value only gets updated once my.data is defined. You are explicitly pointing not to watch for data changes. Angular won't perform any value checks, resulting with fewer expressions being evaluated on each digest cycle.

使用一次性绑定的良好性能示例

```
{{::my.data}}
<span ng-bind="::my.data"></span>
<span ng-if="::my.data"></span>
<span ng-repeat="item in ::my.data">{{item}}</span>
<span ng-class="::{ 'my-class': my.data }"></div>
```

注意： 但是这会移除my.data的双向数据绑定，因此每当该字段在你的应用程序中发生变化时，视图中不会自动反映相应的变化。所以仅对在应用程序生命周期内不会变化的值使用它。

第48.3节：ng-if 与 ng-show

这些功能的行为非常相似。区别在于ng-if会从DOM中移除元素。如果有大量代码部分不会显示，那么ng-if是最佳选择。 ng-show只会隐藏元素，但会保留所有事件处理器。

ng-if

ngIf 指令根据表达式移除或重新创建 DOM 树的一部分。如果赋给 ngIf 的表达式计算结果为假值，则该元素会从 DOM 中移除，否则该元素的克隆会重新插入到 DOM 中。

ng-show

ngShow 指令根据提供给 ngShow 属性的表达式显示或隐藏指定的 HTML 元素。通过向元素添加或移除 ng-hide CSS 类来实现元素的显示或隐藏。

示例

```
<div ng-repeat="user in userCollection">
  <p ng-if="user.hasTreeLegs">我是特别的
    <!-- 一些复杂的 DOM -->
  </p>
  <p ng-show="user.hasSubscribed">我很棒
    <!-- 开关此设置 -->
  </p>
</div>
```

结论

这取决于使用类型，但通常其中一个比另一个更合适（例如，如果 95% 的时间不需要该元素，则使用 ng-if；如果需要切换 DOM 元素的可见性，则使用 ng-show）。

如果不确定，使用 ng-if 并进行测试！

注意: ng-if 会创建一个新的孤立作用域，而 ng-show 和 ng-hide 则不会。如果父作用域的属性在当前作用域中无法直接访问，请使用 $parent.property。

第48.4节：观察者

观察者用于监视某个值并检测该值是否发生变化。

调用$watch()或$watchCollection后，会在当前作用域的内部观察者集合中添加新的观察者。

那么，什么是观察者？

Good performance examples using one-time binding

```
{{::my.data}}
<span ng-bind="::my.data"></span>
<span ng-if="::my.data"></span>
<span ng-repeat="item in ::my.data">{{item}}</span>
<span ng-class="::{ 'my-class': my.data }"></div>
```

Note: This however removes the bi-directional data binding for `my.data`, so whenever this field changes in your application, the same won't be reflected in the view automatically. So **use it only for values that won't change throughout the lifespan of your application**.

Section 48.3: ng-if vs ng-show

These functions are very similar in behaviour. The difference is that `ng-if` removes elements from the DOM. If there are large parts of the code that will not be shown, then `ng-if` is the way to go. `ng-show` will only hide the elements but will keep all the handlers.

ng-if

The ngIf directive removes or recreates a portion of the DOM tree based on an expression. If the expression assigned to ngIf evaluates to a false value then the element is removed from the DOM, otherwise a clone of the element is reinserted into the DOM.

ng-show

The ngShow directive shows or hides the given HTML element based on the expression provided to the ngShow attribute. The element is shown or hidden by removing or adding the ng-hide CSS class onto the element.

Example

```
<div ng-repeat="user in userCollection">
  <p ng-if="user.hasTreeLegs">I am special
    <!-- some complicated DOM -->
  </p>
  <p ng-show="user.hasSubscribed">I am aweosme
    <!-- switch this setting on and off -->
  </p>
</div>
```

Conclusion

It depends from the type of usage, but often one is more suitable than the other (e.g., if 95% of the time the element is not needed, use `ng-if`; if you need to toggle the DOM element's visibility, use `ng-show`).

When in doubt, use `ng-if` and test!

Note: `ng-if` creates a new isolated scope, whereas `ng-show` and `ng-hide` don't. Use `$parent.property` if parent scope property is not directly accessible in it.

Section 48.4: Watchers

Watchers needed for watch some value and detect that this value is changed.

After call `$watch()` or `$watchCollection` new watcher add to internal watcher collection in current scope.

So, what is watcher?

观察者是一个简单的函数，它在每个消化周期（digest cycle）中被调用，并返回某个值。Angular会检查返回的值，如果与上一次调用时的值不同，则会执行传递给$watch()或$watchCollection函数的第二个参数中的回调函数。

```
(function() {
angular.module("app", []).controller("ctrl", function($scope) {
    $scope.value = 10;
$scope.$watch(
      function() { return $scope.value; },
      function() { console.log("value changed"); }
    );
  }
})();
```

观察者会影响性能。观察者越多，执行消化循环的时间越长，界面越慢。如果观察者检测到变化，它将触发消化循环（对整个屏幕进行重新计算）。

在Angular中，有三种手动监视变量变化的方法。

$watch() - 仅监视值的变化

`$watchCollection()` - 监视集合的变化（监视的内容比普通的 $watch 更多）

`$watch(..., true)` - **尽量避免使用**，因为它会执行“深度监视”，会严重影响性能（监视的内容比 watchCollection 更多）

注意，如果你在视图中绑定变量，就会创建新的监视器——使用`{{::variable}}`来避免创建监视器，尤其是在循环中

因此你需要跟踪你使用了多少个监视器。你可以用这个脚本来统计监视器数量（感谢@Words Like Jared - 如何统计页面上的监视器总数？）

```
(function() {
  var root = angular.element(document.getElementsByTagName("body")),
      watchers = [];

  var f = function(element) {

angular.forEach(["$scope", "$isolateScope"], function(scopeProperty) {
      if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
        angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
          watchers.push(watcher);
        });
      }
    });

angular.forEach(element.children(), function(childElement) {
      f(angular.element(childElement));
    });

  };

f(root);

  // 删除重复的观察者
  var watchersWithoutDuplicates = [];
  angular.forEach(watchers, function(item) {
    if(watchersWithoutDuplicates.indexOf(item) < 0) {
      watchersWithoutDuplicates.push(item);
```


Watcher is a simple function, which is called on every digest cycle, and returns some value. Angular checks the returned value, if it is not the same as it was on the previous call - a callback that was passed in second parameter to function `$watch()` or `$watchCollection` will be executed.

```
(function() {
  angular.module("app", []).controller("ctrl", function($scope) {
    $scope.value = 10;
    $scope.$watch(
      function() { return $scope.value; },
      function() { console.log("value changed"); }
    );
  }
})();
```

Watchers are performance killers. The more watchers you have, the longer they take to make a digest loop, the slower UI. If a watcher detects changes, it will kick off the digest loop (recalculation on all screen)

There are three ways to do manual watch for variable changes in Angular.

`$watch()` - just watches for value changes

`$watchCollection()` - watches for changes in collection (watches more than regular $watch)

`$watch(..., true)` - **Avoid this** as much as possible, it will perform "deep watch" and will kill the performance (watches more than watchCollection)

Note that if you are binding variables in the view, you are creating new watchers - use `{{::variable}}` not to create watcher, especially in loops

As a result you need to track how many watchers are you using. You can count the watchers with this script (credit to @Words Like Jared - How to count total number of watches on a page?

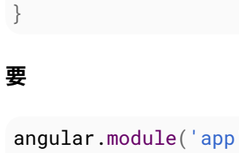
```
(function() {
  var root = angular.element(document.getElementsByTagName("body")),
      watchers = [];

  var f = function(element) {

    angular.forEach(["$scope", "$isolateScope"], function(scopeProperty) {
      if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
        angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
          watchers.push(watcher);
        });
      }
    });

    angular.forEach(element.children(), function(childElement) {
      f(angular.element(childElement));
    });

  };

  f(root);

  // Remove duplicate watchers
  var watchersWithoutDuplicates = [];
  angular.forEach(watchers, function(item) {
    if(watchersWithoutDuplicates.indexOf(item) < 0) {
      watchersWithoutDuplicates.push(item);
```

```
    }
  });

console.log(watchersWithoutDuplicates.length);

})();
```

如果你不想自己编写脚本，有一个名为ng-stats的开源工具，它使用嵌入页面的实时图表，帮助你了解Angular管理的观察者数量，以及消化周期的频率和持续时间。该工具暴露了一个名为showAngularStats的全局函数，你可以调用它来配置图表的工作方式。

```
showAngularStats({
  "position": "topleft",
  "digestTimeThreshold": 16,
  "autoload": true,
  "logDigest": true,
  "logWatches": true
});
```

上面的示例代码会自动在页面上显示以下图表（交互式演示）。



第48.5节：始终注销注册在当前作用域以外的其他作用域上的监听器

你必须始终注销除当前作用域以外的其他作用域，如下所示：

```
//始终注销这些
$rootScope.$on(...);
$scope.$parent.$on(...);
```

你不必注销当前作用域上的监听器，因为Angular会自动处理：

```
//不需要注销这个
$scope.$on(...);
```

`$rootScope.$on` 监听器如果你切换到另一个控制器将会保留在内存中。如果控制器超出作用域，这将导致内存泄漏。

不要

```
angular.module('app').controller('badExampleController', badExample);
badExample.$inject = ['$scope', '$rootScope'];

function badExample($scope, $rootScope) {
    $rootScope.$on('post:created', function postCreated(event, data) {});
}
```

要

```
angular.module('app').controller('goodExampleController', goodExample);
goodExample.$inject = ['$scope', '$rootScope'];
```



```
    }
  });

  console.log(watchersWithoutDuplicates.length);

})();
```

If you don't want to create your own script, there is an open source utility called ng-stats that uses a real-time chart embedded into the page to give you insight into the number of watches Angular is managing, as well as the frequency and duration of digest cycles over time. The utility exposes a global function named `showAngularStats` that you can call to configure how you want the chart to work.

```
showAngularStats({
  "position": "topleft",
  "digestTimeThreshold": 16,
  "autoload": true,
  "logDigest": true,
  "logWatches": true
});
```

The example code above displays the following chart on the page automatically (interactive demo).

Section 48.5: Always deregister listeners registered on other scopes other than the current scope

You must always unregister scopes other then your current scope as shown below:

```
//always deregister these
$rootScope.$on(...);
$scope.$parent.$on(...);
```

You don't have to deregister listners on current scope as angular would take care of it:

```
//no need to deregister this
$scope.$on(...);
```

`$rootScope.$on` listeners will remain in memory if you navigate to another controller. This will create a memory leak if the controller falls out of scope.

Don't

```
angular.module('app').controller('badExampleController', badExample);
badExample.$inject = ['$scope', '$rootScope'];

function badExample($scope, $rootScope) {
    $rootScope.$on('post:created', function postCreated(event, data) {});
}
```

Do

```
angular.module('app').controller('goodExampleController', goodExample);
goodExample.$inject = ['$scope', '$rootScope'];
```

```
function goodExample($scope, $rootScope) {
    var deregister = $rootScope.$on('post:created', function postCreated(event, data) {});

    $scope.$on('$destroy', function destroyScope() {
        deregister();
    });
}
```

第48.6节：作用域函数和过滤器

AngularJS 有消化循环，视图中的所有函数和过滤器都会在每次消化周期运行时执行。每当模型更新时，消化循环都会执行，这可能会减慢你的应用（过滤器可能在页面加载前被多次调用）。

你应该避免这样做：

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.calculateMe()}}</p>
  <p>{{calc.data | heavyFilter}}</p>
</div>
```

更好的方法

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.preCalculatedValue}}</p>
  <p>{{calc.data | lightFilter}}</p>
</div>
```

控制器示例是：

```
.controller("bigCalulations", function(valueService) {
  // 不好，因为这会在每个消化循环中调用
  this.calculateMe = function() {
    var t = 0;
    for(i = 0; i < 1000; i++) {
      t = t + i;
    }
    return t;
  }
  //好，因为它只执行一次，逻辑被分离到服务中以保持控制器轻量

  this.preCalulatedValue = valueService.caluclateSumm(); // 返回 499500
});
```

第48.7节：防抖你的模型

```
<div ng-controller="ExampleController">
    <form name="userForm">
姓名：
        <input type="text" name="userName"
          ng-model="user.name"
          ng-model-options="{ debounce: 1000 }" />
            <button ng-click="userForm.userName.$rollbackViewValue();
user.name=''">清除</button><br />
    </form>
    <pre>user.name = </pre>
</div>
```



```
function goodExample($scope, $rootScope) {
    var deregister = $rootScope.$on('post:created', function postCreated(event, data) {});

    $scope.$on('$destroy', function destroyScope() {
        deregister();
    });
}
```

Section 48.6: Scope functions and filters

AngularJS has digest loop and all your functions in a view and filters are executed every time the digest cycle is run. The digest loop will be executed whenever the model is updated and it can slow down your app (filter can be hit multiple times, before the page is loaded).

You should avoid this:

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.calculateMe()}}</p>
  <p>{{calc.data | heavyFilter}}</p>
</div>
```

Better approach

```
<div ng-controller="bigCalulations as calc">
  <p>{{calc.preCalculatedValue}}</p>
  <p>{{calc.data | lightFilter}}</p>
</div>
```

Where controller sample is:

```
.controller("bigCalulations", function(valueService) {
  // bad, because this is called in every digest loop
  this.calculateMe = function() {
    var t = 0;
    for(i = 0; i < 1000; i++) {
      t = t + i;
    }
    return t;
  }
  //good, because it is executed just once and logic is separated in service to    keep the
controller light
  this.preCalulatedValue = valueService.caluclateSumm(); // returns 499500
});
```

Section 48.7: Debounce Your Model

```
<div ng-controller="ExampleController">
    <form name="userForm">
        Name:
        <input type="text" name="userName"
          ng-model="user.name"
          ng-model-options="{ debounce: 1000 }" />
            <button ng-click="userForm.userName.$rollbackViewValue();
user.name=''">Clear</button><br />
    </form>
    <pre>user.name = </pre>
</div>
```


GoalKicker.com

上述示例中，我们设置了一个1000毫秒（即1秒）的防抖值。这是一个相当长的延迟，但可以防止输入频繁触发ng-model，导致多次$digest周期。

通过在输入字段以及其他不需要即时更新的地方使用防抖，你可以显著提升Angular应用的性能。你不仅可以通过时间来延迟，还可以延迟触发动作的时机。如果你不想在每次按键时更新ng-model，也可以选择在失焦时更新。

belindoc.com

The above example we are setting a debounce value of 1000 milliseconds which is 1 second. This is a considerable delay, but will prevent the input from repeatedly thrashing `ng-model` with many `$digest` cycles.

By using debounce on your input fields and anywhere else where an instant update is not required, you can increase the performance of your Angular apps quite substantially. Not only can you delay by time, but you can also delay when the action gets triggered. If you don't want to update your ng-model on every keystroke, you can also update on blur as well.

第49章：性能分析

第49.1节：关于性能分析

什么是性能分析？

根据定义，性能分析是一种动态程序分析形式，用于测量程序的空间（内存）或时间复杂度、特定指令的使用情况，或函数调用的频率和持续时间。

为什么有必要进行性能分析？

性能分析很重要，因为在不了解程序大部分时间花费在哪些操作上之前，你无法有效地进行优化。如果不测量程序的执行时间（性能分析），你就无法确定是否真的提升了性能。

工具和技术：

1.Chrome内置开发者工具

这包括一套用于性能分析的综合工具。您可以深入了解 JavaScript 文件、CSS 文件、动画、CPU 消耗、内存泄漏、网络、安全等方面的瓶颈。

制作时间线录制并查找异常长的Evaluate Script事件。如果发现，可以启用JS Profiler并重新录制，以获取关于具体调用了哪些JS函数及每个函数耗时的更详细信息。阅读更多...

2. FireBug（适用于Firefox）
3. Dynatrace（适用于IE）
4. Batarang（适用于Chrome）

这是一个过时的Chrome浏览器插件，虽然稳定，但可用于监控Angular应用的模型、性能和依赖关系。
它适用于小型应用，并能让你了解不同层级的scope变量内容。它会告诉你应用中的活动watcher、watch表达式和watch集合。

5. Watcher（适用于Chrome）

界面简洁，方便统计Angular应用中的watcher数量。

6. 使用以下代码手动查找你的Angular应用中的watcher数量（感谢@Words 类似Jared的watcher数量）

```
(function() {
    var root = angular.element(document.getElementsByTagName('body')),
        watchers = [],
f = function(element) {
angular.forEach(['$scope', '$isolateScope'], function(scopeProperty) {
            if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
                angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
```


Chapter 49: Performance Profiling

Section 49.1: All About Profiling

What is Profiling?

By definition Profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.

Why is it necessary?

Profiling is important because you can't optimise effectively until you know what your program is spending most of its time doing. Without measuring your program execution time (profiling), you won't know if you've actually improved it.

Tools and Techniques :

1. Chrome's in-built dev tools

   This includes a comprehensive set of tools to be used for profiling.You can go deep to find out bottlenecks in your javascript file, css files, animations, cpu consumption, memory leaks, network, security etc.

   Make a Timeline recording and look for suspiciously long Evaluate Script events. If you find any, you can enable the JS Profiler and re-do your recording to get more detailed information about exactly which JS functions were called and how long each took. Read more...

2. FireBug (use with Firefox)
3. Dynatrace (use with IE)
4. Batarang (use with Chrome)

   It's an outdated add-on for chrome browser though it's stable and can be used to monitor models, performance, dependencies for an angular application. It works fine for small scale application and can give you an insight of what does scope variable holds at various levels. It tells you about active watchers, watch expressions, watch collections in the app.

5. Watcher (use with Chrome)

   Nice and simplistic UI to count the number of watchers in a Angular app.

6. Use the following code to manually find out the number of watchers in your angular app (credit to @Words Like Jared Number of watchers)

```
(function() {
    var root = angular.element(document.getElementsByTagName('body')),
        watchers = [],
        f = function(element) {
        angular.forEach(['$scope', '$isolateScope'], function(scopeProperty) {
            if(element.data() && element.data().hasOwnProperty(scopeProperty)) {
                angular.forEach(element.data()[scopeProperty].$$watchers, function(watcher) {
```

```
watchers.push(watcher);
                });
            }
        });

angular.forEach(element.children(), function(childElement) {
            f(angular.element(childElement));
        });
    };

f(root);

    // 删除重复的观察者
    var watchersWithoutDuplicates = [];
    angular.forEach(watchers, function(item) {
        if(watchersWithoutDuplicates.indexOf(item) < 0) {
            watchersWithoutDuplicates.push(item);
        }
    });
console.log(watchersWithoutDuplicates.length);
})();
```

7. 有几个在线工具/网站提供了广泛的功能，帮助创建您的应用程序的性能分析报告。

   其中一个网站是：https://www.webpagetest.org/

   通过该网站，您可以从全球多个地点使用真实浏览器（IE 和 Chrome）以及真实的消费者连接速度运行免费的网站速度测试。您可以运行简单测试，也可以执行高级测试，包括多步骤事务、视频捕获、内容屏蔽等更多功能。

下一步：

性能分析完成了。这只是完成了一半的工作。接下来的任务是将您的发现转化为实际的优化措施，以提升您的应用性能。请参阅此文档，了解如何通过简单技巧提升您的 Angular 应用性能。

祝编码愉快 :)


```
                watchers.push(watcher);
                });
            }
        });

        angular.forEach(element.children(), function(childElement) {
            f(angular.element(childElement));
        });
    };

    f(root);

    // Remove duplicate watchers
    var watchersWithoutDuplicates = [];
    angular.forEach(watchers, function(item) {
        if(watchersWithoutDuplicates.indexOf(item) < 0) {
            watchersWithoutDuplicates.push(item);
        }
    });
    console.log(watchersWithoutDuplicates.length);
})();
```

7. There are several online tools/websites available which facilitates wide range of functionalities to create a profile of your application.

   One such site is : https://www.webpagetest.org/

   With this you can run a free website speed test from multiple locations around the globe using real browsers (IE and Chrome) and at real consumer connection speeds. You can run simple tests or perform advanced testing including multi-step transactions, video capture, content blocking and much more.

Next Steps:

Done with Profiling. It only brings you half way down the road. The very next task is to actually turn your findings into action items to optimise your application. See this documentation on how you can improve the performance of your angular app with simple tricks.

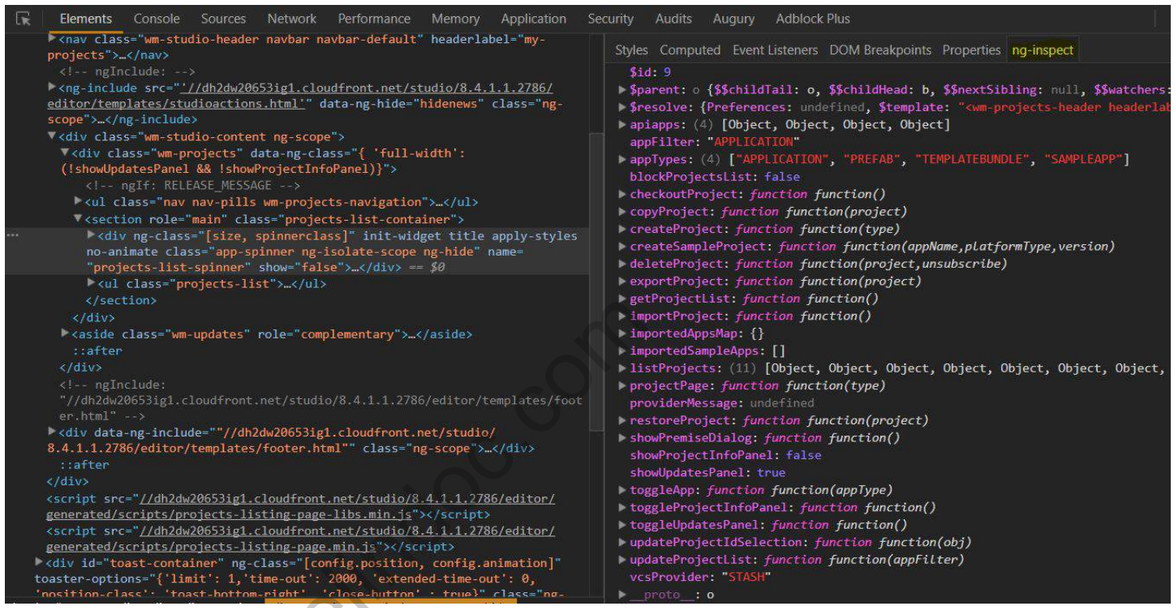
Happy Coding :)

第50章：调试

第50.1节：使用 ng-inspect Chrome 扩展

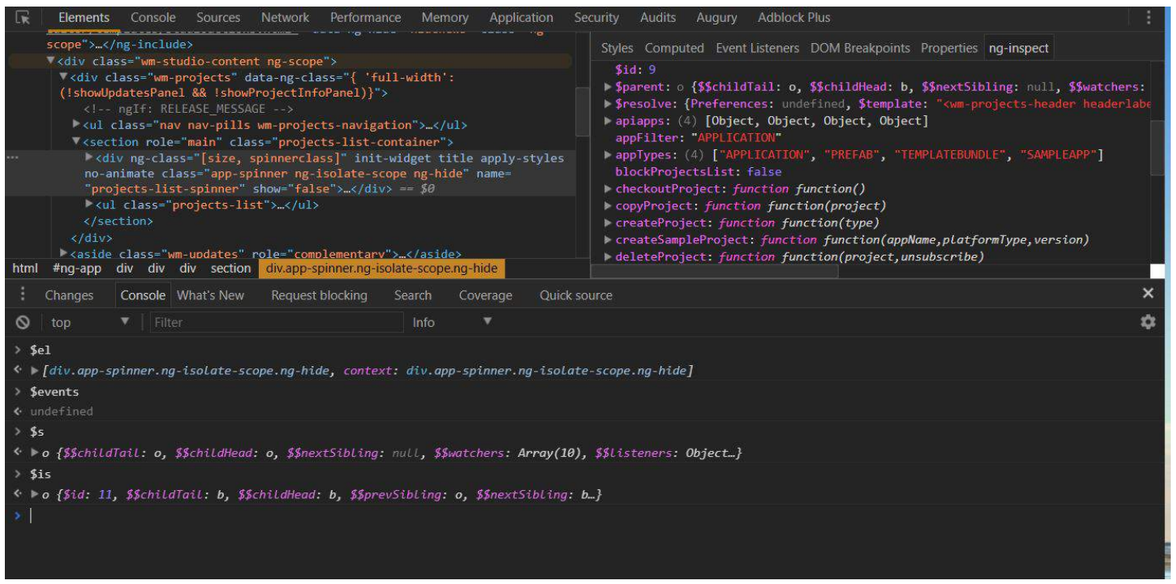
ng-inspect 是一个用于调试 AngularJS 应用程序的轻量级 Chrome 扩展。

当从元素面板中选择一个节点时，相关的作用域信息会显示在 ng-inspect 面板中。



暴露了一些全局变量，方便快速访问作用域/隔离作用域。

```
$s      -- 选中节点的作用域
$is     -- 选中节点的隔离作用域
$el     -- 选中节点的 jQuery 元素引用(需要 jQuery)
$events -- 选中节点上的事件(需要 jQuery)
```

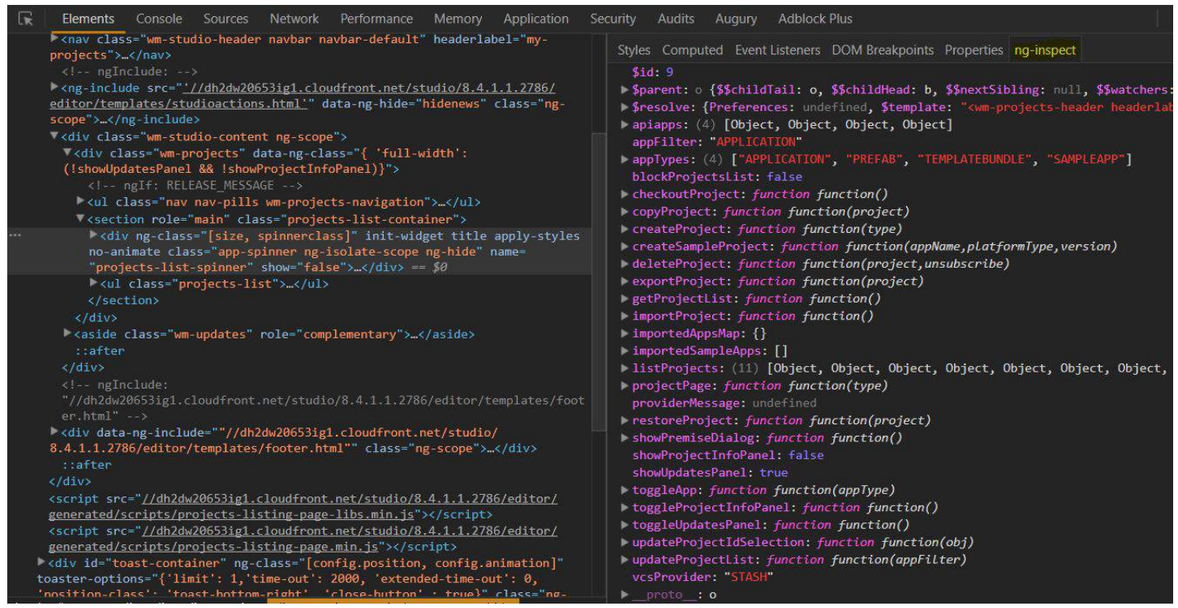


Chapter 50: Debugging

Section 50.1: Using ng-inspect chrome extension

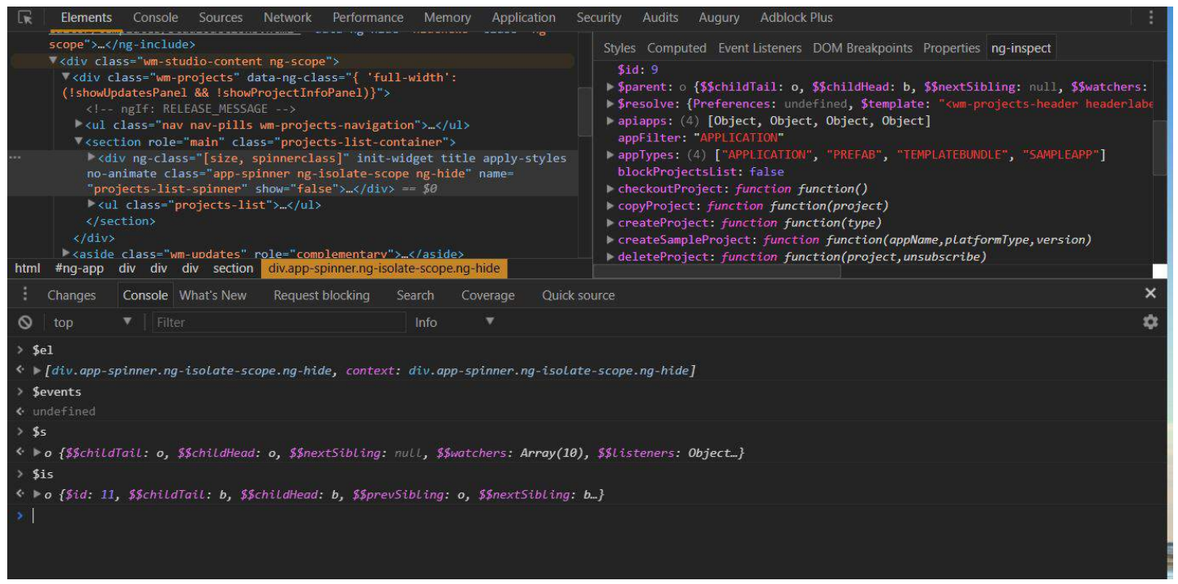
ng-inspect is a light weight Chrome extension for debugging AngularJS applications.

When a node is selected from the elements panel, the scope related info is displayed in the ng-inspect panel.



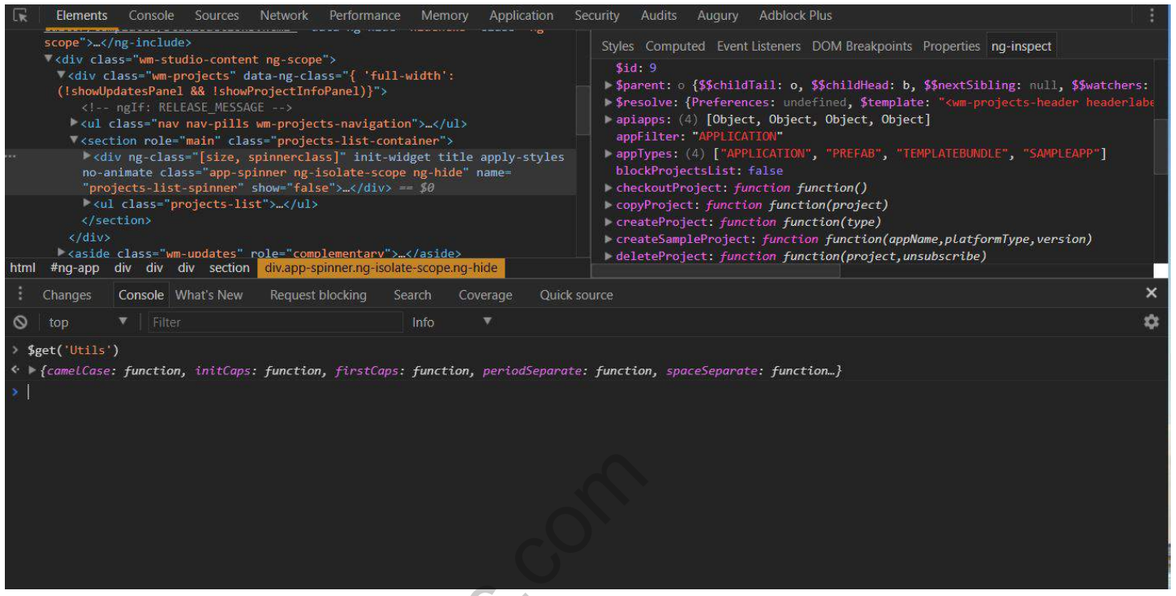
Exposes few global variables for quick access of `scope`/`isolateScope`.

```
$s      -- scope of the selected node
$is     -- isolateScope of the selected node
$el     -- jQuery element reference of the selected node (requiers jQuery)
$events -- events present on the selected node (requires jQuery)
```



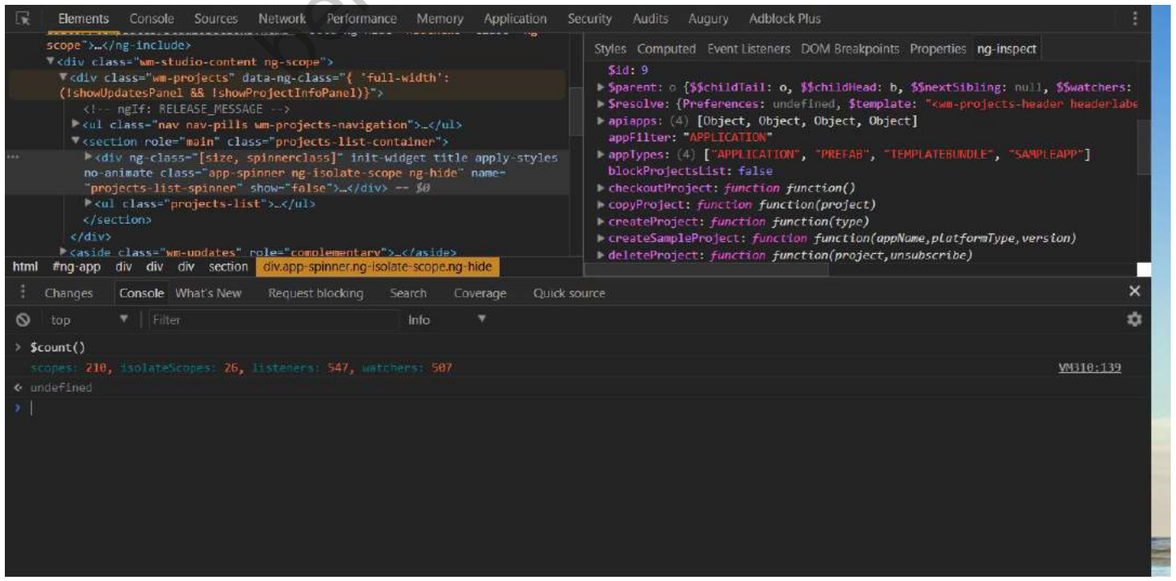
提供了对服务（Services）/工厂（Factories）的便捷访问。

使用$get()通过名称获取服务/工厂的实例。



可以通过统计应用中的作用域、隔离作用域、监视器和监听器的数量来监控应用性能。

使用$count()获取作用域、隔离作用域、监视器和监听器的数量。

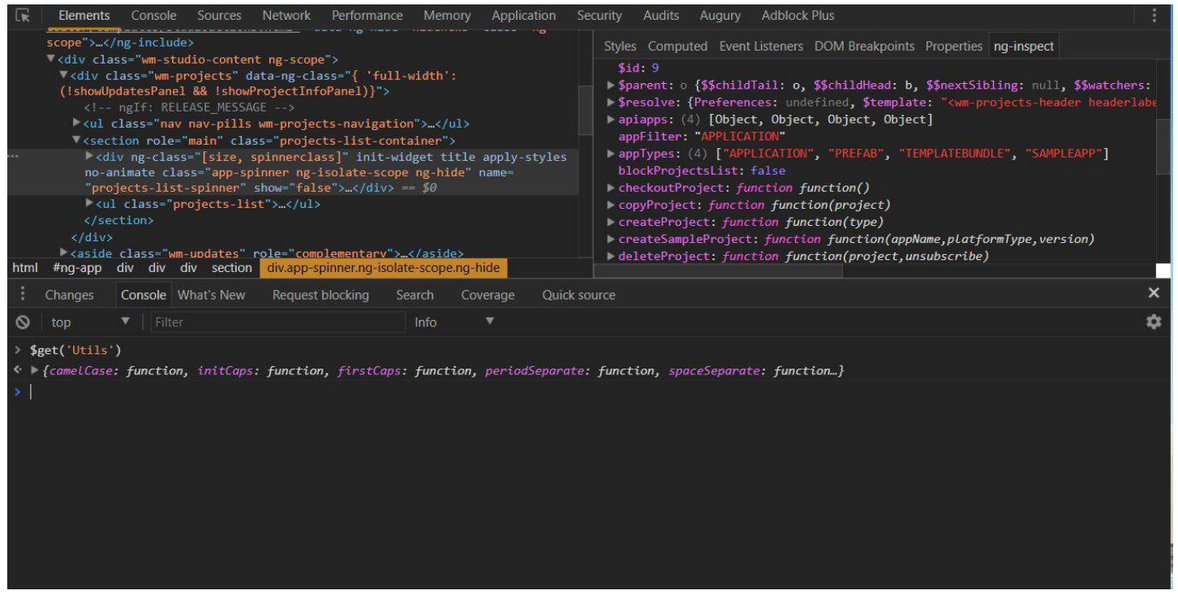


> 注意：此扩展仅在启用 debugInfo 时有效。

下载 ng-inspect here____

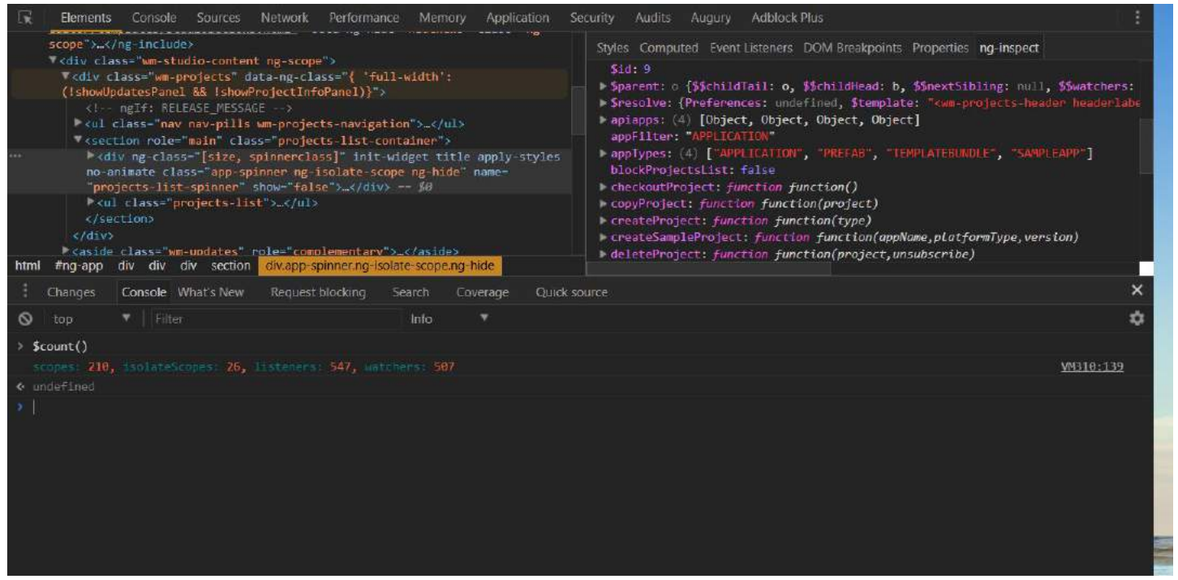
Provides easy access to Services/Factories.

Use `$get()` to retrieve the instance of a service/factory by name.



Performance of the application can be monitored by counting the no.of scopes,isolateScopes, watchers and listeners on the application.

Use `$count()` to get the count of scopes, isolateScopes, watchers and listeners.



> Note: This extension will work only when the debugInfo is enabled.

Download ng-inspect here

第50.2节：获取元素的作用域

在Angular应用中，一切都围绕作用域进行，如果我们能获取元素的作用域，那么调试Angular应用就很容易。如何访问元素的作用域：

```
angular.element(myDomElement).scope();
e.g.
angular.element(document.getElementById('yourElementId')).scope()   //通过ID访问
```

获取控制器的作用域：

```
angular.element('[ng-controller=ctrl]').scope()
```

另一种从控制台访问DOM元素的简单方法（如jm所述）是，在“元素”标签中点击它，它会自动存储为$0。

```
angular.element($0).scope();
```

第50.3节：标记中的基本调试

范围测试与模型输出

```
<div ng-app="demoApp" ng-controller="mainController as ctrl">
   {{$id}}
    <ul>
        <li ng-repeat="item in ctrl.items">
            {{$id}}<br/>
          {{item.text}}
        </li>
    </ul>
    {{$id}}
    <pre>
       {{ctrl.items | json : 2}}
    </pre>
</div>


angular.module('demoApp', [])
.controller('mainController', MainController);

function MainController() {
var vm = this;
    vm.items = [{
        id: 0,
        text: 'first'
    },
     {
id: 1,
        text: 'second'
    },
     {
id: 2,
        text: 'third'
    }];
}
```

有时查看是否存在新的作用域以解决作用域问题会有所帮助。 `$scope.$id` 可以在表达式中使用，在你的标记中任何地方查看是否有新的 `$scope`。

Section 50.2: Getting the Scope of element

In an angular app everything goes around scope, if we could get an elements scope then it is easy to debug the angular app. How to access the scope of element:

```
angular.element(myDomElement).scope();
e.g.
angular.element(document.getElementById('yourElementId')).scope()  //accessing by ID
```

Getting the scope of the controller:

```
angular.element('[ng-controller=ctrl]').scope()
```

Another easy way to access a DOM element from the console (as jm mentioned) is to click on it in the 'elements' tab, and it automatically gets stored as $0.

```
angular.element($0).scope();
```

Section 50.3: Basic debugging in markup

Scope testing & output of model

```
<div ng-app="demoApp" ng-controller="mainController as ctrl">
    {{$id}}
    <ul>
        <li ng-repeat="item in ctrl.items">
            {{$id}}<br/>
            {{item.text}}
        </li>
    </ul>
    {{$id}}
    <pre>
       {{ctrl.items | json : 2}}
    </pre>
</div>


angular.module('demoApp', [])
.controller('mainController', MainController);

function MainController() {
    var vm = this;
    vm.items = [{
        id: 0,
        text: 'first'
    },
    {
        id: 1,
        text: 'second'
    },
    {
        id: 2,
        text: 'third'
    }];
}
```

Sometimes it can help to see if there is a new scope to fix scoping issues. `$scope.$id` can be used in an expression everywhere in your markup to see if there is a new $scope.

在示例中你可以看到，ul 标签外是相同的作用域（$id=2），而在 ng-repeat 内部，每次迭代都有新的子作用域。

在 pre 标签中输出模型有助于查看模型的当前数据。 json 过滤器会创建一个漂亮的格式化输出。使用 pre 标签是因为该标签内的任何换行符

都会被正确显示。

演示

belindoc.com

In the example you can see that outside of the ul-tag is the same scope ($id=2) and inside the `ng-repeat` there are new child scopes for each iteration.

An output of the model in a pre-tag is useful to see the current data of your model. The `json` filter creates a nice looking formatted output. The pre-tag is used because inside that tag any new-line character `\n` will be correctly displayed.

demo

第51章：单元测试

第51.1节：单元测试组件（1.5及以上）

组件代码：

```
angular.module('myModule', []).component('myComponent', {
  bindings: {
myValue: '<'
  },
controller: function(MyService) {
    this.service = MyService;
    this.componentMethod = function() {
      return 2;
    };
  }
});
```

测试：

```
describe('myComponent', function() {
  var component;

  var MyServiceFake = jasmine.createSpyObj(['serviceMethod']);

beforeEach(function() {
module('myModule');
inject(function($componentController) {
      // 第1个参数 - 组件名称，第2个参数 - 控制器注入，第3个参数 - 绑定
      component = $componentController('myComponent', {
        MyService: MyServiceFake
      }, {
myValue: 3
      });
    });
  });

  /** 这里测试注入器。没用。 */

it('注入绑定', function() {
    expect(component.myValue).toBe(3);
  });

it('有一些很酷的行为', function() {
    expect(component.componentMethod()).toBe(2);
  });
});
```

运行！

第51.2节：单元测试过滤器

过滤器代码：

```
angular.module('myModule', []).filter('multiplier', function() {
  return function(number, multiplier) {
    if (!angular.isNumber(number)) {
      throw new Error(number + " 不是数字！");
```


Chapter 51: Unit tests

Section 51.1: Unit test a component (1.5+)

Component code:

```
angular.module('myModule', []).component('myComponent', {
  bindings: {
    myValue: '<'
  },
  controller: function(MyService) {
    this.service = MyService;
    this.componentMethod = function() {
      return 2;
    };
  }
});
```

The test:

```
describe('myComponent', function() {
  var component;

  var MyServiceFake = jasmine.createSpyObj(['serviceMethod']);

  beforeEach(function() {
    module('myModule');
    inject(function($componentController) {
      // 1st - component name, 2nd - controller injections, 3rd - bindings
      component = $componentController('myComponent', {
        MyService: MyServiceFake
      }, {
        myValue: 3
      });
    });
  });

  /** Here you test the injector. Useless. */

  it('injects the binding', function() {
    expect(component.myValue).toBe(3);
  });

  it('has some cool behavior', function() {
    expect(component.componentMethod()).toBe(2);
  });
});
```

Run!

Section 51.2: Unit test a filter

Filter code:

```
angular.module('myModule', []).filter('multiplier', function() {
  return function(number, multiplier) {
    if (!angular.isNumber(number)) {
      throw new Error(number + " is not a number!");
```

```
    }
    if (!multiplier) {
      multiplier = 2;
    }
    return number * multiplier;
  }
});
```

测试：

```
describe('multiplierFilter', function() {
  var filter;

beforeEach(function() {
    module('myModule');
    inject(function(multiplierFilter) {
      filter = multiplierFilter;
    });
  });

it('默认乘以2', function() {
    expect(filter(2)).toBe(4);
expect(filter(3)).toBe(6);
  });

it('允许指定自定义乘数', function() {
    expect(filter(2, 4)).toBe(8);
  });

it('对无效输入抛出错误', function() {
    expect(function() {
filter(null);
    }).toThrow();
  });
});
```

运行！

备注： 在测试中的 `inject` 调用中，您的过滤器需要通过其名称加上 *Filter* 来指定。原因是每当您为模块注册过滤器时，Angular 会在其名称后附加 `Filter` 来注册它。

第51.3节：对服务进行单元测试

服务代码

```
angular.module('我的模块', [])
  .service('我的服务', function() {
    this.doSomething = function(某个数字) {
      return 某个数字 + 2;
    }
  });
```

测试

```
describe('我的服务', function() {
  var 我的服务;
beforeEach(function() {
    module('我的模块');
    inject(function(_我的服务_) {
```



```
    }
    if (!multiplier) {
      multiplier = 2;
    }
    return number * multiplier;
  }
});
```

The test:

```
describe('multiplierFilter', function() {
  var filter;

  beforeEach(function() {
    module('myModule');
    inject(function(multiplierFilter) {
      filter = multiplierFilter;
    });
  });

  it('multiply by 2 by default', function() {
    expect(filter(2)).toBe(4);
    expect(filter(3)).toBe(6);
  });

  it('allow to specify custom multiplier', function() {
    expect(filter(2, 4)).toBe(8);
  });

  it('throws error on invalid input', function() {
    expect(function() {
      filter(null);
    }).toThrow();
  });
});
```

Run!

Remark: In the `inject` call in the test, your filter needs to be specified by its name + *Filter*. The cause for this is that whenever you register a filter for your module, Angular register it with a `Filter` appended to its name.

Section 51.3: Unit test a service

Service Code

```
angular.module('myModule', [])
  .service('myService', function() {
    this.doSomething = function(someNumber) {
      return someNumber + 2;
    }
  });
```

The test

```
describe('myService', function() {
  var myService;
  beforeEach(function() {
    module('myModule');
    inject(function(_myService_) {
```


belindoc.com

```
我的服务 = _我的服务_;
    });
  });
it('应该将 `num` 增加 2', function() {
    var 结果 = 我的服务.doSomething(4);
expect(结果).toEqual(6);
  });
});
```

运行！

第51.4节：单元测试控制器

控制器代码：

```
angular.module('我的模块', [])
  .controller('我的控制器', function($scope) {
    $scope.num = 2;
$scope.doSomething = function() {
      $scope.num += 2;
    }
  });
```

测试：

```
describe('我的控制器', function() {
  var $scope;
beforeEach(function() {
    module('我的模块');
    inject(function($controller, $rootScope) {
      $scope = $rootScope.$new();
$controller('我的控制器', {
          '$scope': $scope
        })
    });
  });
it('应该将 `num` 增加 2', function() {
    expect($scope.num).toEqual(2);
$scope.doSomething();
    expect($scope.num).toEqual(4);
  });
});
```

运行！

第51.5节：单元测试指令

指令代码

```
angular.module('我的模块', [])
  .directive('我的指令', function() {
    return {
template: '<div>{{greeting}} {{name}}!</div>',
      scope: {
name: '=',
        greeting: '@'
      }
    };
```



```
      myService = _myService_;
    });
  });
  it('should increment `num` by 2', function() {
    var result = myService.doSomething(4);
    expect(result).toEqual(6);
  });
});
```

Run!

Section 51.4: Unit test a controller

Controller code:

```
angular.module('myModule', [])
  .controller('myController', function($scope) {
    $scope.num = 2;
    $scope.doSomething = function() {
      $scope.num += 2;
    }
  });
```

The test:

```
describe('myController', function() {
  var $scope;
  beforeEach(function() {
    module('myModule');
    inject(function($controller, $rootScope) {
      $scope = $rootScope.$new();
      $controller('myController', {
        '$scope': $scope
      })
    });
  });
  it('should increment `num` by 2', function() {
    expect($scope.num).toEqual(2);
    $scope.doSomething();
    expect($scope.num).toEqual(4);
  });
});
```

Run!

Section 51.5: Unit test a directive

Directive code

```
angular.module('myModule', [])
  .directive('myDirective', function() {
    return {
      template: '<div>{{greeting}} {{name}}!</div>',
      scope: {
        name: '=',
        greeting: '@'
      }
    };
```


bellindoc.com

```
  });
```

测试

```
describe('myDirective', function() {
  var element, scope;
beforeEach(function() {
    module('myModule');
    inject(function($compile, $rootScope) {
      scope = $rootScope.$new();
element = angular.element("<my-directive name='name' greeting='Hello'></my-directive>");
      $compile(element)(scope);
      /* 请绝不要使用 scope.$digest()。scope.$apply 使用保护机制以避免在已有 digest 循环时再次运行，因此应使用 sc
ope.$apply() 代替。*/
scope.$apply();
    })
  });

it('has the text attribute injected', function() {
    expect(element.html()).toContain('Hello');
  });

it('should have proper message after scope change', function() {
    scope.name = 'John';
scope.$apply();
expect(element.html()).toContain("John");
    scope.name = 'Alice';
expect(element.html()).toContain("John");
    scope.$apply();
expect(element.html()).toContain("Alice");
  });
});
```

运行！


```
  });
```

The test

```
describe('myDirective', function() {
  var element, scope;
  beforeEach(function() {
    module('myModule');
    inject(function($compile, $rootScope) {
      scope = $rootScope.$new();
      element = angular.element("<my-directive name='name' greeting='Hello'></my-directive>");
      $compile(element)(scope);
      /* PLEASE NEVER USE scope.$digest(). scope.$apply use a protection to avoid to run a digest
loop when there is already one, so, use scope.$apply() instead. */
      scope.$apply();
    })
  });

  it('has the text attribute injected', function() {
    expect(element.html()).toContain('Hello');
  });

  it('should have proper message after scope change', function() {
    scope.name = 'John';
    scope.$apply();
    expect(element.html()).toContain("John");
    scope.name = 'Alice';
    expect(element.html()).toContain("John");
    scope.$apply();
    expect(element.html()).toContain("Alice");
  });
});
```

Run!

第52章：AngularJS的陷阱和注意事项

第52.1节：使用html5Mode时的注意事项

使用html5Mode([mode])时，必须：

1. 在index.html的头部用<base href="">指定应用程序的基础URL。
2. 重要的是，base标签必须出现在任何带有URL请求的标签之前。否则，可能会导致
   错误 - "资源被解释为样式表，但传输的MIME类型为text/html"。例如：

```
<head>
    <meta charset="utf-8">
    <title>求职者</title>

    <base href="/">

    <link rel="stylesheet" href="bower_components/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="/styles/main.css">
</head>
```

3. 如果您不想指定base标签，可以通过传递一个定义对象给$locationProvider.html5Mode()，并将requireBase设置为false来配置$locationProvider，使其不要求base标签，方法如下：
   将定义对象中requireBase设置为false传递给$locationProvider.html5Mode()，示例如下：

```
$locationProvider.html5Mode({
    enabled: true,
requireBase: false
});
```

4. 为了支持直接加载HTML5 URL，您需要启用服务器端的URL重写。摘自
   AngularJS / 开发者指南 / 使用 $location

> 使用此模式需要服务器端进行URL重写，基本上您必须将所有链接重写到应用程序的入口点（例如index.html）。在这种情况下，要求使用<base>标签也很重要，因为它允许Angular区分URL中作为应用程序基础的部分和应由应用程序处理的路径。

关于各种HTTP服务器实现的请求重写示例的优秀资源可以在ui-router FAQ - 如何配置服务器以支持html5Mode中找到。例如，Apache

```
 RewriteEngine on

# 不重写文件或目录
RewriteCond %{REQUEST_FILENAME} -f [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^ - [L]

# 将所有其他请求重写到 index.html 以支持 HTML5 状态链接
RewriteRule ^ index.html [L]
```

nginx

Chapter 52: AngularJS gotchas and traps

Section 52.1: Things to do when using html5Mode

When using `html5Mode([mode])` it is necessary that:

1. You specify the base URL for the application with a **`<base href="">`** in the head of your `index.html`.
2. It is important that the `base` tag comes before any tags with url requests. Otherwise, this might result in this error - `"Resource interpreted as stylesheet but transferred with MIME type text/html"`. For example:

```
<head>
    <meta charset="utf-8">
    <title>Job Seeker</title>

    <base href="/">

    <link rel="stylesheet" href="bower_components/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="/styles/main.css">
</head>
```

3. If you do no want to specify a `base` tag, configure `$locationProvider` to not require a `base` tag by passing a definition object with `requireBase:` **`false`** to `$locationProvider.html5Mode()` like this:

```
$locationProvider.html5Mode({
    enabled: true,
    requireBase: false
});
```

4. In order to support direct loading of HTML5 URLs, you need to enabler server-side URL rewriting. From
   AngularJS / Developer Guide / Using $location

> Using this mode requires URL rewriting on server side, basically you have to rewrite all your links to entry point of your application (e.g. `index.html`). Requiring a **`<base>`** tag is also important for this case, as it allows Angular to differentiate between the part of the url that is the application base and the path that should be handled by the application.

An excellent resource for request rewriting examples for various HTTP server implementations can be found in the ui-router FAQ - How to: Configure your server to work with html5Mode. For example, Apache

```
 RewriteEngine on

# Don't rewrite files or directories
RewriteCond %{REQUEST_FILENAME} -f [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^ - [L]

# Rewrite everything else to index.html to allow html5 state links
RewriteRule ^ index.html [L]
```

nginx

GoalKicker.com

```
server {
server_name my-app;

     root /path/to/app;

     location / {
try_files $uri $uri/ /index.html;
     }
 }
```

Express

```
 var express = require('express');
 var app = express();

app.use('/js', express.static(__dirname + '/js'));
 app.use('/dist', express.static(__dirname + '/../dist'));
 app.use('/css', express.static(__dirname + '/css'));
 app.use('/partials', express.static(__dirname + '/partials'));

 app.all('/*', function(req, res, next) {
     // 仅发送 index.html 以支持 HTML5 模式的其他文件
res.sendFile('index.html', { root: __dirname });
 });

app.listen(3006); //你想使用的端口
```

第52.2节：双向数据绑定停止工作

应当牢记：

1. Angular的数据绑定依赖于JavaScript的原型继承，因此它会受到变量遮蔽的影响。
2. 子作用域通常从其父作用域原型继承。该规则的一个例外是具有孤立作用域的指令，因为它不进行原型继承。
3. 有些指令会创建新的子作用域：ng-repeat，ng-switch，ng-view，ng-if，ng-controller，ng-include等。

这意味着当你尝试将某些数据双向绑定到子作用域内的原始值（或反之）时，可能不会按预期工作。这里是一个示例，说明“破坏”AngularJS是多么容易。

可以通过以下步骤轻松避免此问题：

1. 在HTML模板中绑定数据时，确保包含“.”
2. 使用`controllerAs`语法，因为它鼓励绑定到“点状”对象
3. $parent 可以用来访问父级作用域变量，而不是子级作用域。比如在ng-if内部我们可以使用 ng-model="$parent.foo"..

上述的另一种方法是将ngModel绑定到一个getter/setter函数，当带参数调用时，该函数会更新缓存的模型版本，未带参数调用时则返回该版本。为了使用getter/setter函数，需要在带有ngModel属性的元素上添加ng-model-options="{ getterSetter: true }"，并且如果想在表达式中显示其值，需要调用getter函数（工作示例）。

示例

视图：


```
server {
    server_name my-app;

    root /path/to/app;

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Express

```
var express = require('express');
var app = express();

app.use('/js', express.static(__dirname + '/js'));
app.use('/dist', express.static(__dirname + '/../dist'));
app.use('/css', express.static(__dirname + '/css'));
app.use('/partials', express.static(__dirname + '/partials'));

app.all('/*', function(req, res, next) {
    // Just send the index.html for other files to support HTML5Mode
    res.sendFile('index.html', { root: __dirname });
});

app.listen(3006); //the port you want to use
```

Section 52.2: Two-way data binding stops working

One should have in mind that:

1. Angular's data binding relies on JavaScript's prototypal inheritance, thus it's subject to variable shadowing.
2. A child scope normally prototypically inherits from its parent scope. One exception to this rule is a directive which has an isolated scope as it doesn't prototypically inherit.
3. There are some directives which create a new child scope: `ng-repeat`, `ng-switch`, `ng-view`, `ng-if`, `ng-controller`, `ng-include`, etc.

This means that when you try to two-way bind some data to a primitive which is inside of a child scope (or vice-versa), things may not work as expected. Here's an example of how easily is to *"break"* AngularJS.

This issue can easily be avoided following these steps:

1. Have a "." inside your HTML template whenever you bind some data
2. Use `controllerAs` syntax as it promotes the use of binding to a "dotted" object
3. $parent can be used to access parent `scope` variables rather than child scope. like inside `ng-if` we can use `ng-model="$parent.foo"`..

An alternative for the above is to bind `ngModel` to a getter/setter function that will update the cached version of the model when called with arguments, or return it when called without arguments. In order to use a getter/setter function, you need to add `ng-model-options="{ getterSetter: true }"` to the element with the `ngModal` attribute, and to call the getter function if you want to display its value in expression (Working example).

Example

View:

```
<div ng-app="myApp" ng-controller="MainCtrl">
    <input type="text" ng-model="foo" ng-model-options="{ getterSetter: true }">
    <div ng-if="truthyValue">
        <!-- 我是一个子作用域（在ng-if内部），但我与外部作用域的变化同步 -->
        <input type="text" ng-model="foo">
    </div>
    <div>$scope.foo: {{ foo() }}</div>
```

控制器：

```
angular.module('myApp', []).controller('MainCtrl', ['$scope', function($scope) {
    $scope.truthyValue = true;

    var _foo = 'hello'; // 这将用于缓存/表示'foo'模型的值

$scope.foo = function(val) {
        // 当函数被调用且无参数时返回内部的'_foo'变量，
        // 当带参数调用时更新内部的`_foo`变量
        return arguments.length ? (_foo = val) : _foo;
    };
}]);
```

最佳实践：最好保持 getter 方法的执行速度快，因为 Angular 可能比代码的其他部分更频繁地调用它们（参考）。

第52.3节：AngularJS的七宗罪

下面列出了一些开发者在使用 AngularJS 功能时常犯的错误，以及一些经验教训和解决方案。

1. 通过控制器操作 DOM

这是合法的，但应避免。控制器是定义依赖、将数据绑定到视图并执行业务逻辑的地方。技术上你可以在控制器中操作 DOM，但当你在应用的其他部分需要相同或类似的操作时，就需要另一个控制器。因此，这种做法的最佳实践是创建一个包含所有操作的指令，并在整个应用中使用该指令。这样，控制器保持视图不变，专注于它的职责。在指令中，链接函数是操作 DOM 的最佳位置。它可以完全访问作用域和元素，使用指令还能利用可重用性的优势。

```
link: function($scope, element, attrs) {
    // 操作 DOM 的最佳位置
}
```

你可以通过多种方式在链接函数中访问 DOM 元素，例如 element 参数、angular.element() 方法，或纯 JavaScript。

2. 在嵌入内容中进行数据绑定

AngularJS 以其双向数据绑定闻名。然而，有时你可能会遇到数据在指令内部只是单向绑定的情况。别急，AngularJS 没错，可能是你出了问题。指令是比较复杂的地方，因为涉及子作用域和孤立作用域。假设你有以下带有一个嵌入内容的指令

```
<my-dir>
```



```
<div ng-app="myApp" ng-controller="MainCtrl">
    <input type="text" ng-model="foo" ng-model-options="{ getterSetter: true }">
    <div ng-if="truthyValue">
        <!-- I'm a child scope (inside ng-if), but i'm synced with changes from the outside scope -->
        <input type="text" ng-model="foo">
    </div>
    <div>$scope.foo: {{ foo() }}</div>
</div>
```

Controller:

```
angular.module('myApp', []).controller('MainCtrl', ['$scope', function($scope) {
    $scope.truthyValue = true;

    var _foo = 'hello'; // this will be used to cache/represent the value of the 'foo' model

    $scope.foo = function(val) {
        // the function return the the internal '_foo' varibale when called with zero arguments,
        // and update the internal `_foo` when called with an argument
        return arguments.length ? (_foo = val) : _foo;
    };
}]);
```

Best Practice: It's best to keep getters fast because Angular is likely to call them more frequently than other parts of your code (reference).

Section 52.3: 7 Deadly Sins of AngularJS

Below is the list of some mistakes that developers often make during the use of AngularJS functionalities, some learned lessons and solutions to them.

1. Manipulating DOM through the controller

It's legal, but must be avoided. Controllers are the places where you define your dependencies, bind your data to the view and make further business logic. You can technically manipulate the DOM in a controller, but whenever you need same or similar manipulation in another part of your app, another controller will be needed. So the best practice of this approach is creating a directive that includes all manipulations and use the directive throughout your app. Hence, the controller leaves the view intact and does it's job. In a directive, linking function is the best place to manipulate the DOM. It has full access to the scope and element, so using a directive, you can also take the advantage of reusability.

```
link: function($scope, element, attrs) {
    //The best place to manipulate DOM
}
```

You can access DOM elements in linking function through several ways, such as the `element` parameter, `angular.element()` method, or pure Javascript.

2. Data binding in transclusion

AngularJS is famous with its two-way data binding. However you may encounter sometimes that your data is only one-way bound inside directives. Stop there, AngularJS is not wrong but probably you. Directives are a little dangerous places since child scopes and isolated scopes are involved. Assume you have the following directive with one transclusion

```
<my-dir>
```

```
    <my-transclusion>
    </my-transclusion>
</my-dir>
```

在 my-transclusion 内部，你有一些元素绑定到了外部作用域的数据。

```
<my-dir>
    <my-transclusion>
        <input ng-model="name">
    </my-transclusion>
</my-dir>
```

上述代码无法正常工作。这里，transclusion 创建了一个子作用域，你可以获取 name 变量，
没错，但你对该变量所做的任何更改都会停留在子作用域中。因此，你实际上可以通过
$parent.name访问该变量。然而，这种用法可能不是最佳实践。更好的方法是将
变量包装在一个对象中。例如，在控制器中你可以创建：

```
$scope.data = {
    name: 'someName'
}
```

然后在 transclusion 中，你可以通过 'data' 对象访问该变量，并且会发现双向绑定工作得非常好！

```
<input ng-model="data.name">
```

不仅在 transclusion 中，在整个应用中，使用点记法都是一个好主意。

3. 多个指令同时使用

实际上，在同一个元素内同时使用两个指令是合法的，只要遵守以下规则：同一个元素上不能存在两个孤立作用域
。一般来说，创建新的自定义指令时，会分配一个孤立作用域以便于参数传递。假设指令 myDirA 和 myDirB 有孤立
作用域，而 myDirC 没有，以下元素是有效的：

```
<input my-dir-a my-dirc>
```

而以下元素会导致控制台报错：

```
<input my-dir-a my-dir-b>
```

因此，指令的使用必须谨慎，需考虑作用域的情况。

4. $emit 的误用

$emit、$broadcast 和 $on 这三个方法遵循发送者-接收者原则。换句话说，它们是控制器之间通信的手段。
例如，下面这行代码从控制器 A 触发了 'someEvent'，由相关的控制器 B 捕获。

```
$scope.$emit('someEvent', args);
```

下面这行代码捕获了 'someEvent'

```
$scope.$on('someEvent', function(){});
```

到目前为止一切看起来都很完美。但请记住，如果控制器B尚未被调用，事件将不会被


```
    <my-transclusion>
    </my-transclusion>
</my-dir>
```

And inside my-transclusion, you have some elements which are bound to the data in the outer scope.

```
<my-dir>
    <my-transclusion>
        <input ng-model="name">
    </my-transclusion>
</my-dir>
```

The above code will not work correctly. Here, transclusion creates a child scope and you can get the name variable, right, but whatever change you make to this variable will stay there. So, you can truly acces this variable as **$parent.name**. However, this use might not be the best practice. A better approach would be wrapping the variables inside an object. For example, in the controller you can create:

```
$scope.data = {
    name: 'someName'
}
```

Then in the transclusion, you can access this variable via 'data' object and see that two-way binding works perfectly!

```
<input ng-model="data.name">
```

Not only in transclusions, but throughout the app, it's a good idea to use the dotted notation.

3. Multiple directives together

It is actually legal to use two directives together within the same element, as long as you obey by the rule: two isolated scopes cannot exist on the same element. Generally speaking, when creating a new custom directive, you allocate an isolated scope for easy parameter passing. Assuming that the directives myDirA and myDirB have isoleted scopes and myDirC has not, following element will be valid:

```
<input my-dir-a my-dirc>
```

whereas the following element will cause console error:

```
<input my-dir-a my-dir-b>
```

Therefore, directives must be used wisely, taking the scopes into consideration.

4. Misuse of $emit

$emit, $broadcast and $on, these work in a sender-receiver principle. In others words, they are a means of communication between controllers. For example, the following line emits the 'someEvent' from controller A, to be catched by the concerned controller B.

```
$scope.$emit('someEvent', args);
```

And the following line catches the 'someEvent'

```
$scope.$on('someEvent', function(){});
```

So far everything seems perfect. But remember that, if the controller B is not invoked yet, the event will not be

Bellindoc.com

捕获，这意味着发射器和接收器控制器都必须被调用才能使其工作。所以再次强调，如果你不确定是否必须使用$emit，构建一个服务似乎是更好的方式。

5. 误用$scope.$watch

$scope.$watch用于监听变量的变化。每当变量发生变化时，该方法会被调用。然而，一个常见的错误是在$scope.$watch内部修改变量。这会导致不一致并在某些时候引发无限的$digest循环。

```
$scope.$watch('myCtrl.myVariable', function(newVal) {
    this.myVariable++;
});
```

因此，在上述函数中，确保你没有对myVariable和newVal进行任何操作。

6. 将方法绑定到视图

这是最致命的错误之一。AngularJS具有双向绑定，每当某些内容发生变化时，视图会被多次更新。因此，如果你将一个方法绑定到视图的属性上，该方法可能会被调用上百次，这在调试时会让你抓狂。然而，只有某些属性是专门用于方法绑定的，比如ng-click、ng-blur、ng-on-change等，它们期望接收方法作为参数。例如，假设你的标记中有以下视图：

```
<input ng-disabled="myCtrl.isDisabled()" ng-model="myCtrl.name">
```

这里你通过方法 isDisabled 检查视图的禁用状态。在控制器 myCtrl 中，你有：

```
vm.isDisabled = function(){
    if(someCondition)
        return true;
    else
        return false;
}
```

理论上，这看起来是正确的，但从技术上讲，这会导致过载，因为该方法会运行无数次。为了解决这个问题，你应该绑定一个变量。在你的控制器中，必须存在以下变量：

```
vm.isDisabled
```

你可以在控制器的激活阶段重新初始化该变量

```
if(someCondition)
    vm.isDisabled = true
else
vm.isDisabled = false
```

如果条件不稳定，您可以将其绑定到另一个事件。然后，您应该将此变量绑定到视图：

```
<input ng-disabled="myCtrl.isDisabled" ng-model="myCtrl.name">
```

现在，视图的所有属性都具备了预期的值，方法也只会在需要时运行。

7. 不使用 Angular 的功能

AngularJS 提供了一些非常方便的功能，不仅简化了代码，还

caught, which means both emitter and receiver controllers have to be invoked to get this working. So again, if you are not sure that you definitely have to use $emit, building a service seems a better way.

5. Misuse of $scope.$watch

$scope.$watch is used for watching a variable change. Whenever a variable has changed, this method is invoked. However, one common mistake done is changing the variable inside $scope.$watch. This will cause inconsistency and infinite $digest loop at some point.

```
$scope.$watch('myCtrl.myVariable', function(newVal) {
    this.myVariable++;
});
```

So in the above function, make sure you have no operations on myVariable and newVal.

6. Binding methods to views

This is one of the deadlisest sins. AngularJS has two-way binding, and whenever something changes, the views are updated many many times. So, if you bind a method to an attribute of a view, that method might potentially be called a hundred times, which also drives you crazy during debugging. However, there are only some attributes that are built for method binding, such as ng-click, ng-blur, ng-on-change, etc, that expect methods as paremeter. For instance, assume you have the following view in your markup:

```
<input ng-disabled="myCtrl.isDisabled()" ng-model="myCtrl.name">
```

Here you check the disabled status of the view via the method isDisabled. In the controller myCtrl, you have:

```
vm.isDisabled = function(){
    if(someCondition)
        return true;
    else
        return false;
}
```

In theory, it may seem correct but technically this will cause an overload, since the method will run countless times. In order to resolve this, you should bind a variable. In your controller, the following variable must exist:

```
vm.isDisabled
```

You can initiate this variable again in the activation of the controller

```
if(someCondition)
    vm.isDisabled = true
else
    vm.isDisabled = false
```

If the condition is not stable, you may bind this to another event. Then you should bind this variable to the view:

```
<input ng-disabled="myCtrl.isDisabled" ng-model="myCtrl.name">
```

Now, all the attributes of the view have what they expect and the methods will run only whenever needed.

7. Not using Angular's functionalities

AngularJS provides great convenience with some of its functionalities, not only simplifying your code but also

提高了效率。以下列出了一些功能：

1. angular.forEach 用于循环（注意，不能使用“break;”，只能阻止进入循环体，因此这里需要考虑性能。）
2. angular.element 用于 DOM 选择器
3. angular.copy：当你不应该修改主对象时使用
4. **表单验证已经非常棒了。使用 dirty、pristine、touched、valid、required 等状态。**
5. 除了 Chrome 调试器外，移动开发时也使用远程调试。
6. 并且确保你使用Batarang。这是一个免费的 Chrome 扩展，可以轻松检查作用域。
   .

belindoc.com

making it more efficient. Some of these features are listed below:

1. **angular.forEach** for the loops (Caution, you can't "break;" it, you can only prevent getting into the body, so consider performance here.)
2. **angular.element** for DOM selectors
3. **angular.copy**: Use this when you should not modify the main object
4. **Form validations** are already awesome. Use dirty, pristine, touched, valid, required and so on.
5. Besides Chrome debugger, use **remote debugging** for mobile development too.
6. And make sure you use **Batarang**. It's a free Chrome extension where you can easily inspect scopes
   .

鸣谢

非常感谢所有来自 Stack Overflow 文档的人员，他们帮助提供了这些内容，
更多更改可以发送至web@petercv.com，以发布或更新新内容。

阿尤希·贾因	第28章
阿卜杜拉·阿劳伊	第17章
亚当·哈里森	第4章和第14章
Aeolingamenfel	第9章和第14章
阿吉特·拉卡尼	第48章
阿隆·埃坦	第3、4、6、23、24、48和52章
阿尔瓦罗·巴斯克斯	第17章和第24章
阿曼	第50章
安德烈亚	第20章
安费利普	第48章
阿尼鲁达	第48章
安基特	第2章
AnonDCX	第17章
阿隆	第4章
阿肖克·乔杜里	第44章
阿什温·拉马斯瓦米	第48章
阿图尔·米什拉	第48章
AWolf	第4章和第50章
阿扬	第4章
巴拉克D	第3章
邦·马卡林东	第2章、第4章和第14章
布拉维·卡塞姆	第12章和第15章
casraf	第6章
CENT1PEDE	第4章和第38章
chatuur	第14章
科斯明·阿巴贝伊	第52章
Dania	第48章
丹尼尔	第6章
丹尼尔·莫林	第48章
daniellmb	第48章
大卫·G.	第一章
迪帕克·班萨尔	第18章和第49章
developer033	第9章
迪隆·查尼斯	第4章
迪维娅·贾因	第4、26和27章
doctorsherlock	第52章
DotBot	第24章
酷博士	第4、14、25和48章
杜尔格帕尔·辛格	第48章
埃德·欣克利夫	第9章
埃利奥特	第36和51章
埃里克·西贝奈希	第4章
fantarama	第23章
法鲁克·亚兹提	第52章
菲利普·阿马拉尔	第6章
Flash	第17章
fracz	第51章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Aayushi Jain	Chapter 28
Abdellah Alaoui	Chapter 17
Adam Harrison	Chapters 4 and 14
Aeolingamenfel	Chapters 9 and 14
Ajeet Lakhani	Chapter 48
Alon Eitan	Chapters 3, 4, 6, 23, 24, 48 and 52
Alvaro Vazquez	Chapters 17 and 24
Aman	Chapter 50
Andrea	Chapter 20
Anfelipe	Chapter 48
Anirudha	Chapter 48
Ankit	Chapter 2
AnonDCX	Chapter 17
Aron	Chapter 4
Ashok choudhary	Chapter 44
Ashwin Ramaswami	Chapter 48
atul mishra	Chapter 48
AWolf	Chapters 4 and 50
Ayan	Chapter 4
BarakD	Chapter 3
Bon Macalindong	Chapters 2, 4 and 14
Bouraoui KACEM	Chapters 12 and 15
casraf	Chapter 6
CENT1PEDE	Chapters 4 and 38
chatuur	Chapter 14
Cosmin Ababei	Chapter 52
Dania	Chapter 48
Daniel	Chapter 6
Daniel Molin	Chapter 48
daniellmb	Chapter 48
David G.	Chapter 1
Deepak Bansal	Chapters 18 and 49
developer033	Chapter 9
DillonChanis	Chapter 4
Divya Jain	Chapters 4, 26 and 27
doctorsherlock	Chapter 52
DotBot	Chapter 24
Dr. Cool	Chapters 4, 14, 25 and 48
Durgpal Singh	Chapter 48
Ed Hinchliffe	Chapter 9
elliot	Chapters 36 and 51
Eric Siebeneich	Chapter 4
fantarama	Chapter 23
Faruk Yazıcı	Chapter 52
Filipe Amaral	Chapter 6
Flash	Chapter 17
fracz	Chapter 51

尼山特123	第17章
奥朱斯·库尔卡尼	第2章
奥姆里·阿哈龙	第20章
帕雷什·马戈迪亚	第47章
帕尔夫·夏尔马	第37章
帕特	第10章
pathe.kiran	第一章
帕特里克	第一章
菲尔	第52章
皮特	第4章
普拉蒂克·古普塔	第36章
普拉文·普尼亚	第14章和第19章
普什彭德拉	第6章
拉赫玛尼诺夫	第3章和第14章
拉维·辛格	第3章
redunderthebed	第4章
理查德·汉密尔顿	第1、4和23章
罗希特·金达尔	第19、21、22、23、37和43章
ronapelbaum	第51章
瑞安·哈姆利	第33章
RyanDawkins	第21和48章
萨桑克·桑卡瓦利	第7章
发送者	第26章
sgarcia.dev	第6、24、33和48章
沙恩	第21章
沙恩	第23章
沙尚克·维韦克	第21章
ShinDarth	第46章
苏尼尔·拉玛	第1章和第22章
超级巨星	第一章
斯瓦罗格（Svarog）	第4章、第19章、第23章、第24章和第34章
赛义德·普里奥姆	第一章
西尔万	第10章和第11章
盲眼先知	第4章和第48章
thegreenpizza	第14章
timbo	第1、4和23章
托米斯拉夫·斯坦科维奇	第4章
乌梅什·申德	第28章
user3632710	第48章
Ven	第一章
维奈·K	第50章
vincentvanjoe	第48章
维沙尔·辛格	第4章
亚辛·帕特尔	第1章和第48章
尤里·布兰克	第6章
泽·鲁比乌斯	第48章
ziaulain	第32章
zucker	第26章和第29章

Nishant123	Chapter 17
ojus kulkarni	Chapter 2
Omri Aharon	Chapter 20
Paresh Maghodiya	Chapter 47
Parv Sharma	Chapter 37
Pat	Chapter 10
pathe.kiran	Chapter 1
Patrick	Chapter 1
Phil	Chapter 52
Piet	Chapter 4
Prateek Gupta	Chapter 36
Praveen Poonia	Chapters 14 and 19
Pushpendra	Chapter 6
Rachmaninoff	Chapters 3 and 14
Ravi Singh	Chapter 3
redunderthebed	Chapter 4
Richard Hamilton	Chapters 1, 4 and 23
Rohit Jindal	Chapters 19, 21, 22, 23, 37 and 43
ronapelbaum	Chapter 51
Ryan Hamley	Chapter 33
RyanDawkins	Chapters 21 and 48
Sasank Sunkavalli	Chapter 7
Sender	Chapter 26
sgarcia.dev	Chapters 6, 24, 33 and 48
shaN	Chapter 21
shane	Chapter 23
Shashank Vivek	Chapter 21
ShinDarth	Chapter 46
Sunil Lama	Chapters 1 and 22
superluminary	Chapter 1
svarog	Chapters 4, 19, 23, 24 and 34
Syed Priom	Chapter 1
Sylvain	Chapters 10 and 11
theblindprophet	Chapters 4 and 48
thegreenpizza	Chapter 14
timbo	Chapters 1, 4 and 23
Tomislav Stankovic	Chapter 4
Umesh Shende	Chapter 28
user3632710	Chapter 48
Ven	Chapter 1
Vinay K	Chapter 50
vincentvanjoe	Chapter 48
Vishal Singh	Chapter 4
Yasin Patel	Chapters 1 and 48
Yuri Blanc	Chapter 6
Ze Rubeus	Chapter 48
ziaulain	Chapter 32
zucker	Chapters 26 and 29

你可能也喜欢

You may also like