**AngularJS教程**

**大纲**

1. 认识AngularJS
2. 数据绑定和第一个AngularJS Web应用
3. 模块
4. 作用域
5. 控制器
6. 表达式
7. 过滤器
8. 指令简介
9. 内置指令
10. 指令详解
11. AngularJS模块加载
12. 多重视图和路由
13. 依赖注入
14. 服务
15. 同外界通讯：XHR和服务器通讯
16. XHR实践
17. promise
18. 服务器通讯
19. 测试
20. 事件
21. 架构
22. Angular动画
23. digest循环和$apply
24. 揭秘Angular
25. Angular精华扩展
26. 移动应用
27. 本地化
28. 缓存
29. 安全性
30. AngularJS和IE浏览器
31. 构建AngularJS Chrome应用
32. 优化Angular应用
33. 调试AngularJS
34. 下一步
35. 总结
36. D
37. D
38. D
39. D

目录

[第一章 认识AngularJS 9](#_Toc424638323)

[1.1　浏览器如何获取网页 9](#_Toc424638324)

[1.2 　浏览器是什么 11](#_Toc424638325)

[1.3　AngularJS是什么 11](#_Toc424638326)

[1.3.1　AngularJS有什么不同 12](#_Toc424638327)

[1.3.1　AngularJS的特性 12](#_Toc424638328)

[第二章 数据绑定和第一个AngularJS Web应用 15](#_Toc424638329)

[2.1第一个AngularJS Web应用 15](#_Toc424638330)

[2.2 作用域 17](#_Toc424638331)

[2.2.1作用域的生成 17](#_Toc424638332)

[2.2.2作用域的继承 18](#_Toc424638333)

[2.2.3孤立作用域 19](#_Toc424638334)

[2.2.4 AngularJS 孤立作用域的数据绑定 20](#_Toc424638335)

[第三章 模块 23](#_Toc424638336)

[3.1 AngularJS模块化的优点 23](#_Toc424638337)

[3.2 AngularJS模块化的优化 24](#_Toc424638338)

[3.3 用于模块化的相关指令 27](#_Toc424638339)

[第四章 作用域 29](#_Toc424638340)

[4.1什么是作用域 29](#_Toc424638341)

[4.2作用域特点 29](#_Toc424638342)

[4.3作为数据模型的作用域 29](#_Toc424638343)

[4.4作用域的层级 30](#_Toc424638344)

[4.5作用域事件的传递 30](#_Toc424638345)

[4.6作用域的声明周期 31](#_Toc424638346)

[4.7作用域和指令 32](#_Toc424638347)

[4.8控制器和作用域 32](#_Toc424638348)

[4.9 作用域的类型 32](#_Toc424638349)

[4.10 创建子作用域的指令 33](#_Toc424638350)

[第五章 控制器 35](#_Toc424638351)

[第六章 表达式 36](#_Toc424638352)

[6.1 AngularJS表达式概述 36](#_Toc424638353)

[6.2 表达式的意义 36](#_Toc424638354)

[6.3 表达式的解析 37](#_Toc424638355)

[6.4 AngularJS表达式与Javascript表达式的不同 38](#_Toc424638356)

[\* 插值字符串 39](#_Toc424638357)

[第七章 过滤器 44](#_Toc424638358)

[7.1 过滤器的意义 44](#_Toc424638359)

[7.2 过滤器的使用方法 45](#_Toc424638360)

[7.3 AngularJS内置过滤器 48](#_Toc424638361)

[7.4 自定义filter（过滤器） 50](#_Toc424638362)

[第八章 指令 51](#_Toc424638363)

[8.1 指令的作用 52](#_Toc424638364)

[8.2 内置指令 53](#_Toc424638365)

[8.2.1 ng-model 53](#_Toc424638366)

[8.2.2 ng-init 53](#_Toc424638367)

[8.2.3 ng-app 53](#_Toc424638368)

[8.2.4 ng-controller 54](#_Toc424638369)

[8.2.5 ng-form 55](#_Toc424638370)

[8.2.6 ng-disabled 55](#_Toc424638371)

[8.2.7 ng-readonly 55](#_Toc424638372)

[8.2.8 ng-checked 56](#_Toc424638373)

[8.2.9 ng-selected 56](#_Toc424638374)

[8.2.10 ng-show/ng-hide 57](#_Toc424638375)

[8.2.11 ng-change、ng-options 57](#_Toc424638376)

[8.2.12 {{}} 58](#_Toc424638377)

[8.2.13 ng-bind 58](#_Toc424638378)

[8.2.14 ng-cloak 58](#_Toc424638379)

[8.2.15 ng-if 58](#_Toc424638380)

[8.2.16 ng-switch 59](#_Toc424638381)

[8.2.17 ng-repeat 59](#_Toc424638382)

[8.3 自定义指令 62](#_Toc424638383)

[8.4 例子 71](#_Toc424638384)

[第九章 AngularJS模块加载 75](#_Toc424638385)

[9.1 配置块 75](#_Toc424638386)

[9.2 运行块 76](#_Toc424638387)

[9.3 模块的依赖 77](#_Toc424638388)

[\* 模板加载 77](#_Toc424638389)

[\*.1 内存加载 77](#_Toc424638390)

[\*.2 AJAX加载 78](#_Toc424638391)

[\*.3 $templateCache 方法 79](#_Toc424638392)

[第十章 多重视图和路由 79](#_Toc424638393)

[10.1 多重视图和路由的执行步骤 79](#_Toc424638394)

[10.2 redirectTo的使用 81](#_Toc424638395)

[10.3 $routeParams 81](#_Toc424638396)

[10.4 $location服务 82](#_Toc424638397)

[10.5 嵌套路由 85](#_Toc424638398)

[第十一章 依赖注入 87](#_Toc424638399)

[11.1 什么是依赖注入 87](#_Toc424638400)

[11.2 可被依赖的组件 93](#_Toc424638401)

[11.2.1 值 93](#_Toc424638402)

[11.2.2 工厂 94](#_Toc424638403)

[11.2.3 服务 94](#_Toc424638404)

[11.2.4 提供者 95](#_Toc424638405)

[11.2.5 常量 97](#_Toc424638406)

[11.3 可被注入的组件 97](#_Toc424638407)

[11.4 声明依赖的方式 97](#_Toc424638408)

[11.4.1 推断式注入 97](#_Toc424638409)

[11.4.2 标记注入（Annotation） 98](#_Toc424638410)

[11.4.3 内联注入 98](#_Toc424638411)

[11.5 注入器 98](#_Toc424638412)

[第十二章 服务 99](#_Toc424638413)

[12.1 服务概述 99](#_Toc424638414)

[12.2 内置的服务 100](#_Toc424638415)

[12.2.1 $http 100](#_Toc424638416)

[12.2.2 $resource 101](#_Toc424638417)

[12.2.3 $q 101](#_Toc424638418)

[12.3 创建服务 101](#_Toc424638419)

[第十三章 与服务器通讯 105](#_Toc424638420)

[13.1通过$http实现 105](#_Toc424638421)

[13.1.1 $http的处理方法 105](#_Toc424638422)

[13.1.2 $http的快捷方法 107](#_Toc424638423)

[13.1.3 进一步配置请求 108](#_Toc424638424)

[13.1.4设置HTTP头 109](#_Toc424638425)

[13.1.5 缓存响应 110](#_Toc424638426)

[13.1.6转换请求和响应 110](#_Toc424638427)

[13.2采用RESTful资源实现 111](#_Toc424638428)

[13.2.1 什么时候及如何使用Angular Resources组件 111](#_Toc424638429)

[13.2.2 resource资源的声明 114](#_Toc424638430)

[13.2.3 定制方法 115](#_Toc424638431)

[13.2.4 不要使用回调函数机制!(除非你真的需要它们) 115](#_Toc424638432)

[第十四章 promise 119](#_Toc424638433)

[\* 卡通方式趣解 AngularJS 中的 promise 120](#_Toc424638434)

[14.1 promise是什么 123](#_Toc424638435)

[14.2 promise的基本用法 123](#_Toc424638436)

[14.2.1 deffered 对象的方法 124](#_Toc424638437)

[14.2.2 deffered 对象属性 124](#_Toc424638438)

[14.2.3 promise 对象的方法 124](#_Toc424638439)

[\* 英文参考文档（Promise in AngularJS） 124](#_Toc424638440)

[\*.1 What's promise 124](#_Toc424638441)

[\*.2 Promise in AngularJS 125](#_Toc424638442)

[\*.3 How to create a promise 126](#_Toc424638443)

[\*.4 interact with promise 127](#_Toc424638444)

[14.3 promise有与否的区别 130](#_Toc424638445)

[第十五章 测试 131](#_Toc424638446)

[15.1 测试环境的搭建 131](#_Toc424638447)

[15.2基于Karma和Jasmine的AngularJS单元测试 136](#_Toc424638448)

[**技巧** 139](#_Toc424638449)

[15.3 AngularJS端对端测试 140](#_Toc424638450)

[15.3.1 protractor测试环境的安装 140](#_Toc424638451)

[15.3.2 使用protractor进行端对端测试 141](#_Toc424638452)

[\* 参考文献 143](#_Toc424638453)

[\*.1Tutorial 143](#_Toc424638454)

[\*.2 Prerequisites 143](#_Toc424638455)

[\*.3 Setup 143](#_Toc424638456)

[\*.4 Step 0 - write a test 144](#_Toc424638457)

[\*.5 Step 1 - interacting with elements 145](#_Toc424638458)

[\*.6 Step 2 - writing multiple scenarios 146](#_Toc424638459)

[\*.7 Step 3 - changing the configuration 147](#_Toc424638460)

[\*.8 Step 4 - lists of elements 148](#_Toc424638461)

[\*.9 Where to go next 149](#_Toc424638462)

[15.4 AngularJS端对端测试详解 149](#_Toc424638463)

[15.4.1 详细配置文件 149](#_Toc424638464)

[15.4.2 测试文件的书写 158](#_Toc424638465)

[\* 参考文件 159](#_Toc424638466)

[\*.1 Using Locators 159](#_Toc424638467)

[\*.2 Overview 159](#_Toc424638468)

[\*.3 Locators 159](#_Toc424638469)

[\*.4 Actions 160](#_Toc424638470)

[\*.5 Finding Multiple Elements 161](#_Toc424638471)

[\*.6 Finding Sub-Elements 161](#_Toc424638472)

[\*.7 Behind the Scenes: ElementFinders versus WebElements 161](#_Toc424638473)

[第十六章 angular事件 162](#_Toc424638474)

[16.1 事件的作用 162](#_Toc424638475)

[16.2 事件的发布 164](#_Toc424638476)

[16.2.1 $emit() 165](#_Toc424638477)

[16.2.2 $broadcast() 167](#_Toc424638478)

[16.3 事件的监听 167](#_Toc424638479)

[16.4 事件的对象 167](#_Toc424638480)

[16.5 Angular内置事件 169](#_Toc424638481)

[第十七章 架构 170](#_Toc424638482)

[第十八章 动画 170](#_Toc424638483)

[18.1 创建动画的前提条件 170](#_Toc424638484)

[18.2 Angular自定义动画的创建方式 171](#_Toc424638485)

[18.2.1 基于CSS的动画效果 172](#_Toc424638486)

[18.2.2基于Javascript的动画效果 176](#_Toc424638487)

[18.3 为自定义的directives绑定动画 177](#_Toc424638488)

[\* 参考文献 179](#_Toc424638489)

[\*.1 Animating with AngularJS 179](#_Toc424638490)

[\*.2 How it Works… 179](#_Toc424638491)

[\*.3 Supported Events 180](#_Toc424638492)

[\*.4 CSS3 Transitions versus CSS3 Animations 181](#_Toc424638493)

[\*.5 CSS3 Transition Sample 181](#_Toc424638494)

[\*.6 CSS3 Animation Sample 182](#_Toc424638495)

[\*.7 Naming Conventions 183](#_Toc424638496)

[\*.8 Applying the CSS via ng-animate 184](#_Toc424638497)

[\*.9 Customizing Animation 184](#_Toc424638498)

[\*.10 Related Resources 184](#_Toc424638499)

[第十九章digest循环和$apply 185](#_Toc424638500)

[19.1 双向数据绑定 185](#_Toc424638501)

[19.2 双向数据绑定的实现原理 185](#_Toc424638502)

[19.2.1 $watch 185](#_Toc424638503)

[19.2.2 $digest 186](#_Toc424638504)

[19.2.3 $apply 187](#_Toc424638505)

[19.3什么时候需要手动调用$apply()方法 187](#_Toc424638506)

[\* $observe 189](#_Toc424638507)

[\* 参考资料 189](#_Toc424638508)

[\*.1 Understanding Angular’s $apply() and $digest() 189](#_Toc424638509)

[\*.2 $apply and $digest Explored 189](#_Toc424638510)

[\*.3 When Do You Call $apply() Manually? 191](#_Toc424638511)

[\*.4 How Many Times Does the $digest Loop Run? 192](#_Toc424638512)

[\*.5 Conclusion 192](#_Toc424638513)

[第二十章 揭秘Angular 192](#_Toc424638514)

[20.1 AngularJS双向数据绑定揭秘 193](#_Toc424638515)

[20.1.1 $scope.$watch 193](#_Toc424638516)

[20.1.2 $$watchers 194](#_Toc424638517)

[20.1.3 $scope.$apply 195](#_Toc424638518)

[20.1.4 digest cycle 195](#_Toc424638519)

[20.1.5 ng-keydown 198](#_Toc424638520)

[20.1.6 ng-model 198](#_Toc424638521)

[20.1.7 为什么我们的监听器没有被触发？ 199](#_Toc424638522)

[20.2依赖注入的实现原理 200](#_Toc424638523)

[20.2.1 function的参数 200](#_Toc424638524)

[20.2.2 $inject 201](#_Toc424638525)

[20.2. 3数组参数 202](#_Toc424638526)

[20.3 Factory和Service 202](#_Toc424638527)

[20.4从 $rootScope 中创建新的 $scope 203](#_Toc424638528)

[20.5 ANGULAR事件机制揭秘 204](#_Toc424638529)

[20.5.1视图工作原理 204](#_Toc424638530)

[20.5.2编译过程 204](#_Toc424638531)

[20.5.3编译过程 205](#_Toc424638532)

# 第一章 认识AngularJS

## 1.1　浏览器如何获取网页

我们把互联网想象成一个邮局：当你想给朋友写信时，首先要把内容写在一张信纸上，然后在信封上写上地址，再把信纸装进信封。

当你把信送到邮局，邮件分拣机会根据邮编和地址来判断你的朋友住在哪里。如果他住在一栋有很多房间的公寓大楼里面，邮局会把信件投递到大楼的前台，然后大楼的工作人员会根据房间号再次进行分拣。

互联网的工作原理和上面的过程很类似。不同的是，现实世界中由街道连接起来的楼房和公寓，在互联网世界中被路由器和网线连接起来的计算机所取代。每一台计算机都有一个唯一的地址，让网络可以定位到它。

多个公寓房间共享同一个街道地址，与此类似，多台计算机也可以共享同一个网络或路由器。比如，在使用星巴克提供的免费Wi-Fi时，多台计算机就会共享同一个公网IP地址。尽管如此，你的计算机依然可以通过路由器分配的内网IP地址被单独访问到，路由器就好比公寓大楼的工作人员，而内网IP地址就好比房间号。

IP是互联网协议（Internet Protocol）的缩写。IP地址是为每个接入到网络中的设备分配的数字标识符。计算机、打印机甚至手机都有自己的IP地址。

目前有IPv4和IPv6两种主要的IP地址类型，普遍使用的是IPv4地址，例如192.168.0.199这种形式，而IPv6地址是2001:0db8:0000:0000:0000:ff00:0042:8329这种形式的。

当你打开一个浏览器，并在地址栏输入[http://google.com](http://google.com/)后，浏览器会“询问”网络（更准确地说，是“询问”DNS服务器）google.com对应的IP地址是什么？如果DNS服务器知道你要找的IP地址，就会将其结果返回；如果不知道，它会将请求转发给其他DNS服务器，直到在某一台DNS服务器上找到对应的IP地址记录。

查看DNS服务器响应的内容：

如果你使用的是Mac操作系统，可以使用Terminal终端程序，它通常储存在/Applications/Utilities目录中。

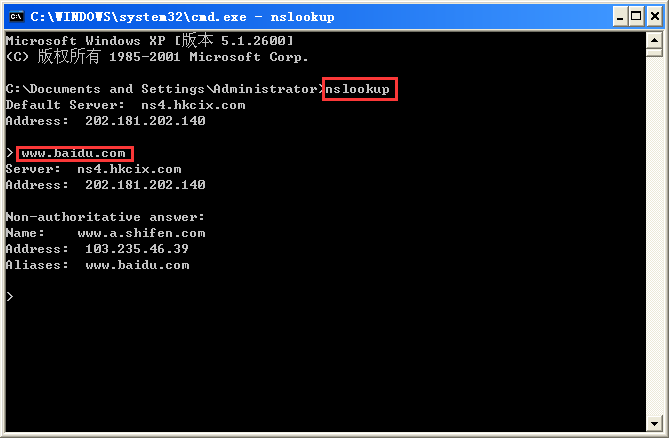
如果使用的是Windows操作系统，

（1）打开开始菜单，在运行中输入cmd打开终端；

（2）输入nslookup命令后回车，将进入DNS解析查询界面。命令行窗口中会显示出当前系统所使用的DNS服务器地址，例如笔者的DNS服务器IP为202.106.0.20。

（3）输入你要访问的站点对应的域名。假如不能访问的话，那么DNS解析应该是不能够正常进行的。我们会收到DNS request timed out，timeout was 2 seconds的提示信息。这说明我们的计算机确实出现了DNS解析故障；如果DNS解析正常的话，会反馈回正确的IP地址。

如：



现在，计算机已经知道了在哪个IP地址可以访问到http://google.com，它会向Google的服务器请求显示这个页面所需的HTML。

当远程服务器把HTML文档发送回来后，浏览器会对文档进行渲染。渲染就是通过一系列操作，使HTML页面按照设计之初的既定方式显示。

## 1.2 　浏览器是什么

在介绍AngularJS之前，我们需要先了解浏览器在渲染网页的过程中都做了些什么。

目前市场上有很多不同品牌的浏览器，常见的有Chrome、Safari、Firefox和IE。它们的核心功能基本上都是相同的：获取网页，并将它显示给用户。

浏览器获取页面对应的HTML文本，将其解析为一个在浏览器内部使用的结构，对页面的内容进行布局，并在内容显示到屏幕上之前加上样式，所有这些工作都是在浏览器内部进行的。

作为Web开发人员，我们的工作是构造网页的结构和内容，这样浏览器才能将它们转化成对用户来说比较美观的形式。

使用AngularJS，不仅可以构建页面的结构，而且可以构建用户和Web应用之间的交互。

## 1.3　AngularJS是什么

AngularJS，完全使用JavaScript编写的客户端技术，诞生于2009年，由Misko Hevery 等人创建，后为Google所收购。是一款优秀的前端JS框架。AngularJS有着诸多特性，最为核心的是：MVVM、模块化、自动化双向数据绑定、语义化标签、依赖注入、等等。

AngularJS通过使用我们称为标识符(directives)的结构，让浏览器能够识别新的语法。例如：

使用双大括号{{}}语法进行数据绑定；

使用DOM控制结构来实现迭代或者隐藏DOM片段；

支持表单和表单的验证；

能将逻辑代码关联到相关的DOM元素上；

能将HTML分组成可重用的组件。

AngularJS的一些出众之处如下：

构建一个CRUD（增加Create、查询Retrieve、更新Update、删除Delete）应用可能用到的全部内容包括：数据绑定、基本模板标识符、表单验证、路由、深度链接、组件重用、依赖注入。

测试方面包括：单元测试、端对端测试、模拟和自动化测试框架。

具有目录布局和测试脚本的种子应用作为起点。

## 1.3.1　AngularJS有什么不同

在其他JavaScript框架中，我们被迫从自定义的JavaScript对象中进行扩展，并从外到内操作DOM。以[jQuery](http://jquery.com/)为例，为了在DOM中插入一个按钮元素，我们必须**知道**要把元素放到何处，并在合适的位置插入它：

var btn = $("<button>Hi</button>");

btn.on('click', function(evt) {

console.log("Clicked button");

});

$("#checkoutHolder").append(btn);

尽管这个过程并不复杂，但是它要求开发者对整个DOM结构都有所了解，并强迫我们在JavaScript代码中加入复杂的控制逻辑，用以操作外部DOM。

而AngularJS则通过原生的Model-View-Controller（MVC，模型-视图-控制器）功能增强了HTML。结果表明，这个选择可以快捷和愉悦地构建出令人印象深刻并且极富表现力的客户端应用。

利用它，开发者可将页面的一部分封装为一个应用，并且不强迫整个页面都使用AngularJS进行开发。这个特质在某些情况下非常有用，比如你的工作流程中已经包含了另外一个框架，或者你只希望页面中的某一部分是动态的，而剩下的部分是静态的或者是由其他JavaScript框架来控制的。

此外，AngularJS团队非常重视框架文件压缩后的大小，这样使用它就不会付出太多的额外代价（写作本书时，文件压缩后的体积在90 KB左右）。这一特性使得AngularJS非常适合用于开发功能原型。

## 1.3.1　AngularJS的特性

AngularJS几个重量级特性：

**特性一：双向的数据绑定**

AngularJS的双向数据绑定，能够同步DOM和Model等等。

这里有一个非常简单的例子，用来演示一个input输入框和<h1>元素的双向绑定：

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8">

<title></title>

<script type="text/javascript" src="js/angular.min.js" ></script>

</head>

<body>

<div>

<label>Name:</label>

<input type="text" ng-model="yourName" placeholder="Enter a name here">

<hr>

<h1>Hello, {{yourName}}!</h1>

</div>

</body>

</html>

**特性二：模板**

在AngularJS中，一个模板就是一个HTML文件。但是HTML的内容扩展了，包含了很多帮助你映射model到view的内容。

这里有一个例子，我们使用ng-repeat指令来循环图片数组并且加入img模板，如下：

<!doctype html>

<html ng-app="Controller">

<head>

<meta charset="utf-8">

<title>模板－图片循环</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="AlbumCtrl">

<ul>

<li ng-repeat="image in images">

<img ng-src="{{image.image}}" alt="{{image.description}}">

</li>

</ul>

</div>

<script>

angular.module('Controller', [])

.controller('AlbumCtrl',function($scope){

$scope.images = [{

"image": "img/01.jpg",

"description": "Image 01 description"

}, {

"image": "img/02.jpg",

"description": "Image 02 description"

}, {

"image": "img/03.jpeg",

"description": "Image 03 description"

}, {

"image": "img/04.jpg",

"description": "Image 04 description"

}, {

"image": "img/05.jpeg",

"description": "Image 05 description"

}];

});

</script>

</body>

</html>

**特性三：MVC**

针对客户端应用开发AngularJS吸收了传统的MVC基本原则。MVC或者Model-View-Controller设计模式针对不同的人可能意味不同的东西。AngularJS并不执行传统意义上的MVC，更接近于MVVM（Model-View-ViewModel)。

（MVC框架可详见“MVC框架.docx”）

* Model

model是应用中的简单数据。一般是简单的javascript对象。这里没有必要继承框架的classes，使用proxy对象封装或者使用特别的setter/getter方法来访问。事实上我们处理vanilla javascript的方法就是一个非常好的特性，这种方法使得我们更少使用应用的原型。？？？？？？？？？？？

* ViewModel

viewmodel是一个用来提供特别数据和方法从而维护指定view的对象。

viewmodel是$scope的对象，只存在于AnguarJS的应用中。$scope只是一个简单的js对象，这个对象使用简单的API来侦测和广播状态变化。???

* Controller

controller负责设置初始状态和参数化$scope方法用以控制行为。需要指出的controller并不保存状态也不和远程服务互动。

* View

view是[AngularJS](http://angularjs.org/)解析后渲染和绑定后生成的HTML 。这个部分帮助你创建web应用的架构。$scope拥有一个针对数据的参考，controller定义行为，view处理布局和互动。

**特性四：依赖注入（Dependency Injection，即DI）**

AngularJS拥有内建的依赖注入子系统，可以帮助开发人员更容易的开发，理解和测试应用。

DI允许你请求你的依赖，而不是自己找寻它们。比如，我们需要一个东西，DI负责找创建并且提供给我们。

为了而得到核心的AngularJS服务，只需要添加一个简单服务作为参数，AngularJS会侦测并且提供给你：

你也可以定义自己的服务并且让它们注入。

**特性五：Directives（指令）**

指令可以用来创建自定义的标签。它们可以用来装饰元素或者操作DOM属性。

使用一系列的组件来创建你自己的应用将会让你更方便的添加，删除和更新功能。

**额外的特性：测试**

[AngularJS](http://angularjs.org/)内含了测试用例可以帮助你更方便的执行测试。

JS是一个动态的解析性语言，而不是编译类型的，因此非常的难写测试。AngularJS被开成一个可测试的框架。它甚至包含了点对点的单元测试runner。

AngularJS主要用于构建单页面Web应用。

# 第二章 数据绑定和第一个AngularJS Web应用

## 2.1第一个AngularJS Web应用

下面是一个包含了一个表单的典型CRUD应用。表单值先经过验证，然后用来计算总值，这个总值会被格式化成本地的样式。

<!doctype html>

<html ng-app="Controller">

<head>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="InvoiceCntl">

<b>Invoice:</b>

<br>

<br>

<table>

<tr>

<td>Quantity</td>

<td>Cost</td>

</tr>

<tr>

<td>

<input type="integer" min="0" ng-model="qty" required>

</td>

<td>

<input type="number" ng-model="cost" required>

</td>

</tr>

</table>

<hr>

<b>Total:</b> {{qty\*cost|currency}}

</div>

<script>

angular.module('Controller', [])

.controller('InvoiceCntl', function($scope) {

$scope.qty = 1;

$scope.cost = 19.95;

});

</script>

</body>

</html>

试一下上面这个例子。然后我们一起来看下这个例子的工作原理：

在<html>标签里, 我们用一个ng-app标识符标明这是一个AngularJS应用。这个ng-app标识符会使AngularJS自动初始化(auto initialize)你的应用。

我们用<script>标签来加载AngularJS脚本：

<script src="angular-1.1.0.min.js"></script>通过设置<input>标签里的ng-model属性, AngularJS会自动对数据进行双向绑定。我们还同时进行了一些简单的数据验证：

Quantity:<input type="integer"min="0"ng-model="qty"required >Cost:<input type="number"ng-model="cost"required >这个输入框的widget看起来很普通，但如果认识到以下几点那它就不普通了：

当页面加载完后，AngularJS会依照widget里的声明的模型名字（qty、cost）生成同名变量。你可以把这些变量认为是MVC设计模式中的M(Model)；

Total:{{qty \*cost |currency}}这个{{表达式}}标记是AngularJS的数据绑定。其中的表达式可以是表达式和过滤器({{ expression | filter }})的组合。AngularJS提供了过滤器来对输入输出数据格式化。

上面的这个例子里，{{}}里的表达式让AngularJS把从输入框中获得的数据相乘，然后把相乘结果格式化成本地货币样式，然后输出到页面上。

值得一提的是，我们既没有调用任何AngularJS的方法，也没有像用框架一样去编写某个具体逻辑，就是完成了上述功能。这个实现的背后是因为浏览器做了比以往生成 静态页面更多的工作，让它能满足动态WEB应用的需要。AngularJS使得动态WEB应用的开发门槛降到不需要类库或者框架的程度。

在前端html中使用{{ }}括起来的变量, 是跟AngularJS中对应的controller作用域内的属性绑定在一起的. 实际上,{{}}等同于ng-bind指令, 即ng-bind="myData"就能将NG中的myData数据跟前端对应元素绑定在一起.这样的话, 可以非常方便地做到从NG中获取任意数据并实时展示在页面上了.

另外, $scope对象还提供了一个$apply方法, 用于进行html页面上的更新, 使用方式为:

$scope.$apply(function(){

$scope.myValue = "NewValue";

}); （？？？？？？？？？？？）

## 2.2 作用域

$scope对象, 我们可以理解为NG框架中的一个作用域对象, 在该作用域内可以做到数据和视图的相互绑定, 同时又能与其他$scope对象的作用域隔离开来.

AngularJS 用一系列指令来代替 jQuery 的事件绑定代码。为了能够组织好各类指令之间的协调工作而不出现数据混乱，AngularJS 在模型层上引申出作用域的概念，以配合控制器来实现对视图层的展现工作。

AngularJS 中，作用域是一个指向应用模型的对象，它是表达式的执行环境。作用域有层次结构，这个层次和相应的 DOM 几乎是一样的。作用域能监控表达式和传递事件。

### 2.2.1作用域的生成

在 HTML 代码中，一旦一个 ng-app 指令被定义，那么一个作用域就产生了，由 ng-app 所生成的作用域比较特殊，它是一个根作用域（$rootScope），它是其他所有$Scope 的最顶层。

除了用 ng-app 指令可以产生一个作用域之外，其他的指令如 ng-controller，ng-repeat 等都会产生一个或者多个作用域。此外，还可以通过 AngularJS 提供的创建作用域的工厂方法来创建一个作用域。这些作用域都拥有自己的继承上下文，并且根作用域都为$rootScope。

在生成一个作用域之后，在编写 AngularJS 代码时，$scope 对象就代表了这个作用域的数据实体，我们可以在$scope 内定义各种数据类型，之后可以直接在 HTML 中以 {{变量名}} 方式来让 HTML 访问到这个变量。

如：

（BBB）

### 2.2.2作用域的继承

AngularJS 在创建一个作用域时，会检索上下文，如果上下文中已经存在一个作用域，那么这个新创建的作用域就会以 JavaScript 原型继承机制继承其父作用域的属性和方法（有个例外是孤立作用域，下文讨论）。

一些 AngularJS 指令会创建新的子作用域，并且进行原型继承： ng-repeat、ng-include、ng-switch、ng-view、ng-controller, 用 scope: true 和 transclude: true 创建的 directive。

如：

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body data-ng-app="app">

<div data-ng-controller="parentCtrl">

{{args}}

<div data-ng-controller="childCtrl">

{{args}}

<br />

{{args1}}

</div>

</div>

<script>

angular.module('app', [])

.controller('parentCtrl', function($scope) {

$scope.args = '父作用域';

$scope.args1 = '子作用域中没有，在父作用域中找';

})

.controller('childCtrl', function($scope) {

$scope.args = '子作用域';

});

</script>

</body>

</html>

以下 HTML 中定义了三个作用域，分别是由 ng-app 指令所创建的$rootScope，parentCtrl 和 childCtrl 所创建的子作用域，这其中 childCtrl 生成的作用域又是 parentCtrl 的子作用域。

继承作用域符合 JavaScript 的原型继承机制，这意味着如果我们在子作用域中访问一个父作用域中定义的属性，JavaScript 首先在子作用域中寻找该属性，没找到再从原型链上的父作用域中寻找，如果还没找到会再往上一级原型链的父作用域寻找。在 AngularJS 中，作用域原型链的顶端是$rootScope，AnguarJS 将会寻找到$rootScope 为止，如果还是找不到，则会返回 undefined。

### 2.2.3孤立作用域

孤立作用域是 AngularJS 中一个非常特殊的作用域，它只在 directive 中出现。在对 directive 的定义中，我们添加上一个 scope:{} 属性，就为这个 directive 创建出了一个隔离作用域。

孤立作用域最大的特点是不会原型继承其父作用域，对外界的父作用域保持相对的独立。因此，如果在定义了孤立作用域的 AngularJS directive 中想要访问其父作用域的属性，则得到的值为 undefined。

如：

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body data-ng-app="app">

<div data-ng-controller="ctrl">

<div data-isolate-directive></div>

</div>

<script>

angular.module('app', [])

.controller('ctrl', ['$scope',

function($scope) {

$scope.args = {};

}

])

.directive("isolateDirective", function() {

return {

scope: {},

link: function($scope, $element, $attr) {

console.log($scope.args); //输出 undefined

}

};

});

</script>

</body>

</html>

上面的代码中通过在 directive 中声明了 scope 属性从而创建了一个作用域，其父作用域为 ctrl 所属的作用域。但是，这个作用域是孤立的，因此，它访问不到父作用域的中的任何属性。存在这样设计机制的好处是：能够创建出一些列可复用的 directive，这些 directive 不会相互在拥有的属性值上产生串扰，也不会产生任何副作用。

### 2.2.4 AngularJS 孤立作用域的数据绑定

在继承作用域中，我们可以选择子作用域直接操作父作用域数据来实现父子作用域的通信，而在孤立作用域中，子作用域不能直接访问和修改父作用域的属性和值。为了能够使孤立作用域也能和外界通信，AngularJS 提供了三种方式用来打破孤立作用域“孤立”这一限制。

1. 单向绑定（@ 或者 @attr）

这是 AngularJS 孤立作用域与外界父作用域进行数据通信中最简单的一种，绑定的对象只能是父作用域中的字符串值，并且为单向只读引用，无法对父作用域中的字符串值进行修改，此外，这个字符串还必须在父作用域的 HTML 节点中以 attr（属性）的方式声明。

使用这种绑定方式时，需要在 directive 的 scope 属性中明确指定引用父作用域中的 HTML 字符串属性，否则会抛异常。示例代码如下：

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body data-ng-app="isolateScope">

<div data-ng-controller="ctrl">

<button>{{btns}}</button>

<div data-isolate-directive data-isolates="{{btns}}"></div>

</div>

<script>

angular.module('isolateScope', [])

.directive("isolateDirective", function() {

return {

replace: true,

template: '<button>{{isolates}}</button>',

scope: {

isolates: '@', //指定引用父作用域

},

link: function($scope, $element, $attr) {

$scope.isolates = "DeveloperWorks";

}

};

})

.controller("ctrl", function($scope) {

$scope.btns = 'IBM';

});

</script>

</body>

</html>

简单分析下上面的代码，通过在 directive 中声明了 scope:{isolates:'@'} 使得 directive 拥有了父作用域中 data-isolates 这个 HTML 属性所拥有的值，这个值在控制器 ctrl 中被赋值为'IBM'。所以，代码的运行结果是页面上有两个名为 IBM 的按钮。

我们还注意到 link 函数中对 isolates 进行了修改，但是最终不会在运行结果中体现。这是因为 isolates 始终绑定为父作用域中的 btns 字符串，如果父作用域中的 btns 不改变，那么在孤立作用域中无论怎么修改 isolates 都不会起作用。

1. 引用绑定（&或者&attr）

通过这种形式的绑定，孤立作用域将有能力访问到父作用域中的函数对象，从而能够执行父作用域中的函数来获取某些结果。这种方式的绑定跟单向绑定一样，只能以只读的方式访问父作用函数，并且这个函数的定义必须写在父作用域 HTML 中的 attr（属性）节点上。

这种方式的绑定虽然无法修改父作用域的 attr 所设定的函数对象，但是却可以通过执行函数来改变父作用域中某些属性的值，来达到一些预期的效果。示例代码如下：

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body data-ng-app="isolateScope">

<div data-ng-controller="ctrl">

<div data-isolate-directive data-isolates="func"></div>

</div>

<script>

angular.module('isolateScope', [])

.directive("isolateDirective", function() {

return {

replace: true,

scope: {

isolates: '&',

},

link: function($scope, $element, $attr) {

var func = $scope.isolates();

func();

}

};

})

.controller("ctrl", function($scope) {

$scope.func = function() {

console.log("IBM DeveloperWorks");

}

});

</script>

</body>

</html>

这个例子中，浏览器的控制台将会输出一段“IBM DeveloperWorks”文字。

上面的代码中我们在父作用域中指定了一个函数对象$scope.func，在孤立作用域中通过对 HTML 属性的绑定从而引用了 func。需要注意的是 link 函数中对 func 对象的使用方法，$scope.isolates 获得的仅仅是函数对象，而不是调用这个对象，因此我们需要在获取到$scope.isolates 之后再调用这个函数，才能得到真正的执行结果。

（3）双向绑定（=或者=attr）

双向绑定赋予 AngularJS 孤立作用域与外界最为自由的双向数据通信功能。在双向绑定模式下，孤立作用域能够直接读写父作用域中的属性和数据。和以上两种孤立作用域定义数据绑定一样，双向绑定也必须在父作用域的 HTML 中设定属性节点来绑定。

双向绑定非常适用于一些子 directive 需要频繁和父作用域进行数据交互，并且数据比较复杂的场景。不过，由于可以自由的读写父作用域中的属性和对象，所以在一些多个 directive 共享父作用域数据的场景下需要小心使用，很容易引起数据上的混乱。

示例代码如下：

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body data-ng-app="isolateScope">

<div data-ng-controller="ctrl">

<button>{{btns.dw}}</button>

<button>{{btns.ibm}}</button>

<div data-isolate-directive data-isolates="btns"></div>

</div>

<script>

angular.module('isolateScope', [])

.directive("isolateDirective", function() {

return {

replace: true,

template: '<button>{{isolates.ibm}}</button>',

scope: {

isolates: '=',

},

link: function($scope, $element, $attr) {

$scope.isolates.ibm = "IBM";//屏蔽后就可以看到明显的效果

}

};

})

.controller("ctrl", function($scope) {

$scope.btns = {

ibm: 'ibm',

dw: 'DeveloperWorks'

};

});

</script>

</body>

</html>

上面的代码运行的结果是浏览器页面上出现三个按钮，其中第一个按钮标题为“DeveloperWorks”，第二和第三个按钮的标题为“IBM”。

初始时父作用域中的$scope.btns.ibm 为小写的“ibm”，通过双向绑定，孤立作用域中将父作用域的 ibm 改写成为大写的“IBM”并且直接生效，父作用域的值被更改。

# 第三章 模块

## 3.1 AngularJS模块化的优点

大部分应用都有一个主方法用来实例化、组织、启动应用。AngularJS应用没有主方法，而是使用模块来声明应用应该如何启动。这种方式有以下几个优点：

1.启动过程是声明式的，所以更容易懂。

2.在单元测试是不需要加载全部模块的，因此这种方式有助于写单元测试。

3.可以在特定情况的测试中增加额外的模块，这些模块能更改配置，能帮助进行端对端的测试。

4.第三方代码可以打包成可重用的模块。

5.模块可以以任何先后或者并行的顺序加载（因为模块的执行本身是延迟的）。

下面是一个简单的例子：

<!doctype html>

<html ng-app="myApp">

<head>

<!--有时候因为复制其他代码而导致编码格式混乱从而使代码无法执行，此时要使用以下代码强制转换编码格式-->

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<script>

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

myAppModule.filter('greet', function() {

return function(name) {

return 'Hello, ' + name + '!';

};

});

</script>

</head>

<body>

<div>

{{ 'World' | greet }}

</div>

</body>

</html>

## 3.2 AngularJS模块化的优化

以上这个例子写法很简单，但是不适合用同样的写法来写大型应用。我们推荐是将你的应用拆分成以下几个模块：

1.一个服务模块，用来做服务的声明。

2.一个指令模块，用来做指令的声明。

3.一个过滤器模块，用来做过滤器声明。

4.一个依赖以上模块的应用级模块，它包含初始化代码。

例如：

<!doctype html>

<html ng-app="xmpl">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="XmplController">

{{ greeting }}!

</div>

<script>

angular.module('xmpl.service', []). //服务模块

value('greeter', { //自定义greeter对象

salutation: 'Hello',

localize: function(localization) {

this.salutation = localization.salutation;

},

greet: function(name) {

return this.salutation + ' ' + name + '!';

}

}).

value('user', { //自定义user对象

load: function(name) {

this.name = name;

}

});

angular.module('xmpl.directive', []); //指令模块

angular.module('xmpl.filter', []); //过滤器模块

angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter']). //模块xmpl依赖于数组中的模块

run(function(greeter, user) {

// 初始化代码，应用启动时，会自动执行

greeter.localize({

salutation: 'Bonjour'

});

user.load('World');

})

// A Controller for your app

.controller('XmplController', function($scope, greeter, user) {

$scope.greeting = greeter.greet(user.name);

});

</script>

</body>

</html>

这样拆分的原因是，在你的测试中常常需要忽略掉初始化代码，因为这些代码比较难测试。通过把它拆分出来就能在测试中方便地忽视掉它。通过只加载和当前测试相关的模块，也能使测试更专一。以上只是一个建议，你可以放心根据你的具体情况来调整。

一个模块就是一系列配置和代码块的集合，它们是在启动阶段就附加到应用上的。一个最简单的模块由两类代码块集合组成的：

配置代码块 - 在注入提供者注入和配置阶段执行。只有注入提供者和常量可以被注入到配置块中。这是为了防止服务在被配置好之前就被提前初始化。  
运行代码块 - 在注入器被创建后执行，被用来启动应用的。只有实例和常量能被注入到运行块中。这是为了防止在运行后还出现对系统的配置。

angular.module('myModule', [])

.config(function(injectables) { // provider-injector 配置代码块

　　　　 // This is an example of config block.

　　　　 // You can have as many of these as you want.

　　　　 // You can only inject Providers (not instances)

　　　　 // into the config blocks.

　　})

.run(function(injectables) { // instance-injector 运行代码块

　　　　 // This is an example of a run block.

　　　　 // You can have as many of these as you want.

　　　　 // You can only inject instances (not Providers)

　　　　 // into the run blocks

　　});

模块还有一些配置的简便方法，使用它们的效果等同于使用代码块。举个例子：

angular.module('myModule', [])

.value('a', 123)

.factory('a', function() {

return 123;

})

.directive('directiveName', ...)

.filter('filterName', ...);

// is same as

angular.module('myModule', [])

.config(function($provide, $compileProvider, $filterProvider) {

$provide.value('a', 123);

$provide.factory('a', function() {

return 123;

});

$compileProvider.directive('directiveName', ...);

$filterProvider.register('filterName', ...);

});

（？？？？？？？？？？待校验）

配置块会按照$provide, $compileProvider, $filterProvider，注册的顺序，依次被应用。唯一的例外是对常量的定义，它们应该始终放在配置块的开始处。

运行块是AngularJS中最像主方法的东西。一个运行块就是一段用来启动应用的代码。它在所有服务都被配置和所有的注入器都被创建后执行。运行块通常包含了一些难以测试的代码，所以它们应该写在单独的模块里，这样在单元测试时就可以忽略它们了。

模块可以把其他模块列为它的依赖。“依赖某个模块”意味着需要把这个被依赖的模块在本块模块之前被加载。换句话说被依赖模块的配置块会在本模块配置块前被执行。运行块也是一样。任何一个模块都只能被加载一次，即使它被多个模块依赖。

模块是一种用来管理$injector配置的方法，和脚本的加载没有关系。现在网上已有很多控制模块加载的库，它们可以和AngularJS配合使用。因为在加载期间模块不做任何事情，所以它们可以以任意顺序或者并行方式加载。

## 3.3 用于模块化的相关指令

模块可以说是AngularJS 的根本。它包含配置、控制、过滤、工厂模式、指令及其它模块。

如果你熟悉.NET平台，但初步学习Angular。下面的表格是一个简要的对比，帮助你理解Angular中的角色扮演情况：

|  |  |  |
| --- | --- | --- |
| **AngularJS** | **.NET** | **摘要** |
| module | Assembly | 应用开发模块 |
| controller | ViewModel | 控制器，启到不同层面间的组织作用 |
| scope | DataContext | 为视图提供绑定数据 |
| filter | ValueConverter | 数据传输到视图之前修改数据 |
| directive | Component | 可复用的UI元素，也可以理解为前端插件 |
| factory, service | Utility classes | 为其他模块元素提供服务 |

例如，下面的代码使用控制器、过滤器和指令创建了一个模块：

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-app="myApp" ng-controller="myCtrl">

<input ng-model="msg" />

<p my-dctv>

{{msg | myUpperFilter }}

</p>

<script>

// the main (app) module

var myApp = angular.module("myApp", []);

// add a controller

myApp.controller("myCtrl", function($scope) {

$scope.msg = "grapecity team blog";

});

// add a filter

myApp.filter("myUpperFilter", function() {

return function(input) {

return input.toUpperCase();

}

});

// add a directive

myApp.directive("myDctv", function() {

return function(scope, element, attrs) {

element.bind("mouseenter", function() {

element.css("background", "yellow");

});

element.bind("mouseleave", function() {

element.css("background", "none");

});

}

});

</script>

</body>

</html>

上面示例中module 方法的第一个参数为模块的名称，第二个参数为它的依赖模块列表。我们创建了一个独立的模块，不依赖于其它模块。所以第二个参数为空数组（注意：即使它为空，我们也必须填写这个参数。否则，该方法回去检索之前的同名模块）。

controller 构造函数获取$scope 对象，用于存储所有controller 暴露的接口和方法。scope 由Angular 传递到视图和指令层。在这个例子中， controller 添加了msg 属性给scope对象。一个应用模块可以包含多个controller，每个controller各司其职，控制一个或多个视图。

filter 构造函数返回一个方法用于更改input文本的显示方式。Angular 提供很多内置的filter，同时，你也可以添加自定义filter，操作方式Angular内置filter相同。在这个例子中，实现了小写到大写的转换。Filter不仅可以格式化文本值，还可以更改数组。AngularJS 内置的格式化Filter有number、date、currency、uppercase和lowercase。数组 filter有filter、orderBy和 limitTo。Filter需要设置参数，语法格式也是固定的： *someValue* |*filterName*:*filterParameter1*:*filterParameter2...*.

directive 构造函数返回了一个方法，该方法用于传递一个元素，并依据scope中的参数对其进行修改。示例中我们绑定了mouseenter 和mouseleave 事件用于切换文本高亮显示。

注意应用中module、controller和filter 作为特性值应用。它们代表JavaScript 对象，因此名称是区分大小写的。指令的名称同样也是属性值，它作为HTML标签被解析，所以也是区分大小写的。但AngularJS 会自动转换这些特性为小写，例如“myDctv" 指令变成"my-dctv" (就像内置的指令ngApp, ngController, 和ngModel会转换成 "ng-app", "ng-controller", 和"ng-model"。

# 第四章 作用域

## 4.1什么是作用域

作用域是一个指向应用模型的对象。它是表达式的执行环境。作用域有层次结构，这个层次和相应的DOM几乎是一样的。作用域能监控表达式和传递事件。

## 4.2作用域特点

（1）作用域提供APIs（$watch）来观察模型的变化。

（2）作用域提供APIs($apply)将任何模型的改变从"AngularJS领域( Angular realm)"通过系统映射到视图上。

（3）作用域能通过共享模型成员的方式嵌套到应用组件上。一个作用域从父作用域继承属性。

（4）作用域提供表达式执行的上下文。比如说表达式{{username}}本身是无意义的，除非把它放到指定username属性的作用域中。

## 4.3作为数据模型的作用域

作用域是控制器和视图之间的“胶水”。在模板链接阶段，指令设置好作用域的$watch表达式。$watch使得指令能知晓属性的改变，这使得指令能重新渲染和更新DOM中的值。

控制器和指令都持有作用域的引用，但是不持有对方的。这使得控制器能从指令和DOM中脱离出来。这很重要，因为这使得控制器完全不需要知道view的存在，这大大改善了应用的测试。

你可以把作用域和它的属性当做是用来渲染视图的数据。作用域是视图唯一相关联的变化来源。

## 4.4作用域的层级

每一个AngularJS应用都有一个绝对的根作用域。但是可能有多个子作用域。

一个应用可以有多个作用域，因为有一些指令会生成新的子作用域（参考指令的文档看看哪些指令会创建新作用域）。当新作用域被创建的时候，他们会被当成子作用域添加到父作用域下，这使得作用域会变成一个和相应DOM结构一个的树状结构。

当AngularJS执行表达式{{username}}，它会首先查找和当前节点相关的作用域中叫做username的属性。如果没找到，那就会继续向上层作用域搜索，直到根作用域。在Javascript中，这被称为原型类型的继承，子作用域以原型的形式继承自父作用域。

从DOM中获取作用域

作用域是作为$scope的数据属性关联到DOM上的，并且能在需要调试的时候被获取到。（这不像那些只能在应用内获取的框架）根作用关联的DOM就是ng-app指令定义的地方。一般来说ng-app都是放在<html>元素中的，但是也能放在其他元素中。

在调试器中检测作用域

在你要查看的元素上右键单击，选择菜单中的“审查元素”。你应该会看到浏览器的调试器，并且你选择的元素高亮了显示了。

调试器能让你在控制台中用变量$0获取到。

在控制台中获取想关联的作用域：angular.element($0).scope()

## 4.5作用域事件的传递

作用域中的事件传递是和DOM事件传递类似的。事件可以广播给子作用域或者传递给父作用域。

如：

<!doctype html>

<html ng-app='MyApp'>

<head>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="EventController">

Root scope <tt>MyEvent</tt> count: {{count}}

<ul>

<li ng-repeat="i in [1]" ng-controller="EventController">

<button ng-click="$emit('MyEvent')">$emit('MyEvent')</button>

<button ng-click="$broadcast('MyEvent')">$broadcast('MyEvent')</button>

<br> Middle scope <tt>MyEvent</tt> count: {{count}}

<ul>

<li ng-repeat="item in [1, 2]" ng-controller="EventController">

Leaf scope <tt>MyEvent</tt> count: {{count}}

</li>

</ul>

</li>

</ul>

</div>

<script>

angular.module('MyApp', [])

.controller('EventController', function($scope) {

$scope.count = 0;

$scope.$on('MyEvent', function() {

$scope.count++;

});

});

</script>

</body>

</html>

## 4.6作用域的声明周期

浏览器接收到事件后的一般工作流程是执行一个相应的Javascript回调。回调一执行完，浏览器就会重新渲染DOM并且重新回到等待事件的状态。

当浏览器调用AngularJS上下文之外的Javascript代码时，AngularJS是不知道模型的更改的。要正确处理模型的更改，就要使用$apply方法进入AngularJS的执行上下文。只有在$apply方法内执行的模型修改才会正确地被AngularJS处理。比如，一个指令监听DOM事件，比如ng-click，它必须在$apply方法中来执行表达式。

执行完表达式之后，$apply会进入$digest阶段。在$digest阶段作用域会检查所有的$watch表达式，并将它们和之前的值比较。这个检查工作是异步执行的。这意味着赋值语句，如$scope.username="angular"不会马上导致$watch被通知，取而代之的是它会等到$digest阶段才被通知。这种方式是合理的，因为它将多个模型的更新整合到一个$watch通知里，并且保证了一个$watch通知期间不会有其他$watch执行。如果一个$watch改变了模型的值，那么它会产生一个额外的$digest阶段。

**1.创建**

根作用域是在应用被$injector启动时创建的。在模板链接阶段，有些指令会创建新的子作用域。

**2.观察者注册**

在模板链接阶段，指令会在作用域上注册观察者。这些观察者是用来将模型的改变传递给DOM的。

**3.模型变化**

为了正确地观测到模型变化，你需要并且只能在scope.$apply()中改变他们。（AngularJS的API会隐式地这么做，所以在控制器或者在$http,$timeout等服务中你不需要额外的调用$apply）。

**4.变化的观测**

在$apply的最后，AngularJS会在根作用域中执行一个$digest循环，它会将变化传递给所有子作用域。在$digest循环中，所有的$watch表达式或者函数都会被检测，来观察模型的变化。如果有变化被检测到了，$watch的监听回调就会被调用。

**5.作用域的销毁**

如果子作用域不再有用了。那么子作用域的创建者就会负责用scope.$destroy() API来将它销毁。这会停止$digest再调用子作用域，并且让作用域占用的内容能够被回收。

## 4.7作用域和指令

在编译阶段，编译器在DOM中匹配指令。指令通常分为两种：

观察型的指令，例如双花括号表达式{{expression}}，会用$watch来注册一个监听者。无论表达式什么时候改变，这类型的指令都会被通知，并且能更新视图。

监听者型的指令，比如ng-click，会向DOM注册一个监听者。当DOM监听者触发，指令会执行相关的表达式并且使用$apply方法更新视图。

当一个外界事件（比如用户操作，计时器或者XHR）触发时，相应的表达式必须在$apply()方法内来由其相应的作用域调用，这样所有的监听者才会被正确地更新。

大部分情况下，指令和作用域交互但不会产生新的作用域实例。但是，有些指令，比如ng-controller和ng-repeat会创建新的作用域，并关联到相应的DOM元素上，你该可以使用angular.element(aDomElement).scope()方法来获得某一个DOM元素相关的作用域。

## 4.8控制器和作用域

作用域和控制器在以下几种情况下交互：

（1）控制器通过作用域来向模板暴露方法；

（2）控制器里定义能改变模型（作用域的属性）的方法（行为）；

（3）控制器在模型上注册了观察者。这些观察者会在控制器行为执行后立即被执行

## 4.9 作用域的类型

AngularJS存在四种作用域:

（1）普通的带原型继承的作用域 -- ng-include, ng-switch, ng-controller, directive with scope: true；

（2）普通的带原型继承的，并且有赋值行为的作用域 -- ng-repeat，**ng-repeat**为每一个迭代项创建一个普通的有原型继承的子作用域，但同时在子作用域中创建新属性存储迭代项；

（3）“Isolate”作用域 -- directive with scope: {...}， 该作用域没有原型继承，但可以通过'=', '@', 和 '&'与父作用域通信。

（4）“transcluded”作用域 -- directive with transclude: true，它也是普通的带原型继承的作用域，但它与“Isolate”作用域是相邻的好基友。

## 4.10 创建子作用域的指令

以下方式会创建新的子作用域，并且进行原型继承： ng-repeat、ng-include、ng-switch、ng-view、ng-controller, 用scope: true和transclude: true创建directive。

以下方式会创建新的独立作用域，不会进行原型继承：用scope: { ... }创建directive。这样创建的作用域被称为"Isolate"作用域。

**注意：**默认情况下创建directive使用了scope: false，不会创建子作用域。

进行原型继承即意味着父作用域在子作用域的原型链上，这是JavaScript的特性。AngularJS的作用域还存在如下内部定义的关系：

scope.$parent指向scope的父作用域；

**scope.$$childHead指向scope的第一个子作用域；**

**scope.$$childTail指向scope的最后一个子作用域；**

**scope.$$nextSibling指向scope的下一个相邻作用域；**

**scope.$$prevSibling指向scope的上一个相邻作用域；**

ng-include

ng-switch

ng-view

ng-repeat

ng-controller

directives

1. 默认 (scope: false) - directive使用原有作用域，所以也不存在原型继承，这种方式很简单，但也很容易出问题——除非该directive与html不存在数据绑定，否则一般情况建议使用第2条方式。
2. scope: true - directive创建一个子作用域, 并且会从父作用域进行原型继承。 如果同一个DOM element存在多个directives要求创建子作用域，那么只有一个子作用域被创建，directives共用该子作用域。
3. scope: { ... } - directive创建一个独立的“Isolate”作用域，没有原型继承。这是创建可复用directive组件的最佳选择。因为它不会直接访问/修改父作用域的属性，不会产生意外的副作用。这种directive与父作用域进行数据通信有如下四种方式（更详细的内容请参考[Developer Guide](http://docs.angularjs.org/guide/directive)）：
   1. **= or =attr** “Isolate”作用域的属性与父作用域的属性进行双向绑定，任何一方的修改均影响到对方，这是最常用的方式；
   2. **@ or @attr** “Isolate”作用域的属性与父作用域的属性进行单向绑定，即“Isolate”作用域只能读取父作用域的值，并且**该值永远的String类型**；
   3. **& or &attr** “Isolate”作用域把父作用域的属性包装成一个函数，从而以函数的方式读写父作用域的属性，包装方法是$parse，详情请见[API-$parse](http://docs.angularjs.org/api/ng.$parse)；

“Isolate”作用域的\_\_proto\_\_是一个标准[Scope](http://docs.angularjs.org/api/ng.$rootScope.Scope) object (the picture below needs to be updated to show an orange 'Scope' object instead of an 'Object'). “Isolate”作用域的$parent同样指向父作用域。它虽然没有原型继承，但它仍然是一个子作用域。

如下directive：

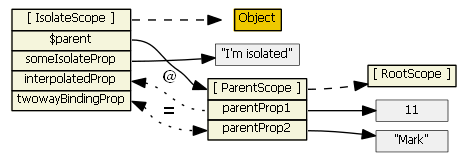
<my-directive interpolated="{{parentProp1}}" twowayBinding="parentProp2">

scope：

scope: { interpolatedProp: '@interpolated', twowayBindingProp: '=twowayBinding' }

link函数中：

scope.someIsolateProp = "I'm isolated"



请注意，我们在link函数中使用attrs.$observe('interpolated', function(value) { ... }来监测@属性的变化。

更多请参考： <http://onehungrymind.com/angularjs-sticky-notes-pt-2-isolated-scope/>

1. transclude: true - directive新建一个“transcluded”子作用域，并且会从父作用域进行原型继承。需要注意的是，“transcluded”作用域与“Isolate”作用域是相邻的关系（如果“Isolate”作用域存在的话） -- 他们的$parent属性指向同一个父作用域。“Isolate”作用域的$$nextSibling指向“transcluded”作用域。

# 第五章 控制器

 AngularJS 控制器是常规的 **JavaScript 对象**，由标准的 JavaScript **对象的构造函数** 创建。用来控制数据在AngularJS应用程序中的流动。

控制器采用ng-controller指令定义。控制器是一个包含属性/属性和JavaScript对象的功能。每个控制器接受$scope参数指定应用程序/模块，由控制器控制。

# 第六章 表达式

## 6.1 AngularJS表达式概述

AngularJS 表达式写在双大括号内：**{{ expression }}**。

AngularJS 表达式把数据绑定到 HTML，这与 **ng-bind** 指令有异曲同工之妙。

AngularJS 将在表达式书写的位置"输出"数据。

**AngularJS 表达式** 很像 **JavaScript 表达式**：它们可以包含文字、运算符和变量。

实例 {{ 5 + 5 }} 或 {{ firstName + " " + lastName }}

## 6.2 表达式的意义

表达式指的是javascript的一小片段代码，用于应用程序数据绑定到HTML。表达式都写在双括号内，就像{{表达式}}。表达式中的行为跟ng-bind指令方式相同。 AngularJS应用表达式是纯javascript表达式，并输出在它们被使用的数据那里。

使用数字

<p>Expense on Books : {{cost \* quantity}} Rs</p>

使用字符串

<p>Hello {{student.firstname + " " + student.lastname}}!</p>

使用对象

<p>Roll No: {{student.rollno}}</p>

使用数组

<p>Marks(Math): {{marks[3]}}</p>

下面的例子将展示上述所有表达式。

<html>

<title>AngularJS Expressions</title>

<body>

<h1>Sample Application</h1>

<div ng-app="" ng-init="quantity=1;cost=30; student={firstname:'Mahesh',lastname:'Parashar',rollno:101};marks=[80,90,75,73,60]">

<p>Hello {{student.firstname + " " + student.lastname}}!</p>

<p>Expense on Books : {{cost \* quantity}} Rs</p>

<p>Roll No: {{student.rollno}}</p>

<p>Marks(Math): {{marks[3]}}</p>

</div>

<script type="text/javascript" src="js/angular.min.js" ></script>

</body>

</html>

## 6.3 表达式的解析

在angularjs中，表达式是通过[$parse](http://docs.angularjs.org/api/ng.$parse) service解析的。

$parse用法： $parse(expression);

参数：javascript代码片段。

返回值：{function(context, locals)}。

表达式编译结果：

context：嵌入表达式执行的作用于scope。

locals：本地变量，常用于替换重写context。

返回值同样带有assign属性，允许为表达式赋值。

对于angularjs表达式，可以采用[$eval()](http://docs.angularjs.org/api/ng.$rootScope.Scope#$eval)方法解析执行。如：

<!doctype html>

<html ng-app='myApp'>

<head>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="Cntl2" class="expressions">

Expression:

<input type='text' ng-model="expr" size="80" />

<button ng-click="addExp(expr)">Evaluate</button>

<ul>

<li ng-repeat="expr in exprs">

[ <a href="" ng-click="removeExp($index)">X</a> ] <tt>{{expr}}</tt> => <span ng-bind="$parent.$eval(expr)"></span> </li>

</ul>

</div>

<script>

angular.module('myApp', [])

.controller('Cntl2', function($scope) {

var exprs = $scope.exprs = [];

$scope.expr = '3\*10|currency';

$scope.addExp = function(expr) {

exprs.push(expr);

};

$scope.removeExp = function(index) {

exprs.splice(index, 1);

};

});

</script>

</body>

</html>

## 6.4 AngularJS表达式与Javascript表达式的不同

angularjs视图表达式有点像javascript表达式，但是并不是利用javascript表达式eval()函数解析执行的，与javascript表达式区别如下：

（1）属性表达式

有属性都依赖于scope作用域。并不是javascript的全局作用域window。

angularjs所有的表达式执行都将依赖于一个作用域scope,不同于javascript的window全局作用域.如果你想引用全局作用于window,这必须引用$window service. 实例如下:

<!doctype html>

<html ng-app='myApp'>

<head>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div class="example2" ng-controller="Cntl1">

Name:

<input ng-model="name" type="text" />

<button ng-click="greet()">Greet</button>

</div>

<script>

angular.module('myApp', [])

.controller('Cntl1', function($window, $scope) {

$scope.name = 'World';

$scope.greet = function() {

($window.mockWindow || $window).alert('Hello ' + $scope.name);

}

});

</script>

</body>

</html>

（2）允许未定义值

angularjs表达式计算兼容处理null和undefined不会抛出任何异常，因为这将现实处理在view显示，而javascript则会抛出NullPointerExceptions异常。例如表达式{{a.b.c}}，与其同等效果的javascript代码将是{{((a||{}).b||{}).c}}。

（3）没有控制结构语句

在angularjs表达式中将不存在条件，循环，throw控制流程语句。因为angularjs作为mvc或者贴近pm模式要求表现层逻辑必须包含在controller中，而不是view，view应该足够的被动。在表达式模式中都尽力将表现层不变的表现逻辑和多样易变的UI view中抽离出来，便于更好的自动化测试构建等。

（4）过滤器

angularjs表达式计算结果可以通过过滤器转化格式，|表达式，比如时间，货币，数字格式等。

例如：name | uppercase ，123 | number:2，3\*10|currency。

filters支持链式写法，如powershell或者其他操作系统外壳语言一样的管道式模型，形如value | filter1 | filter2。

## \* 插值字符串

<!doctype html>

<html ng-app='myApp'>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="MyController">

<div style="margin-bottom: 1em;">

<input ng-model="to" type="email" placeholder="Recipient" />

</div>

<div id="emailEditor">

<textarea ng-model="emailBody"></textarea>

</div>

<div id="emailPreview">

<pre>\_\_previewText\_\_</pre>

</div>

</div>

<script>

angular.module('emailParser', []).config(['$interpolateProvider',

function($interpolateProvider) {

$interpolateProvider.startSymbol('\_\_');

$interpolateProvider.endSymbol('\_\_');

}

])

.factory('EmailParser', ['$interpolate',

function($interpolate) {

// 处理解析的服务

return {

parse: function(text, context) {

var template = $interpolate(text);

return template(context);

}

};

}

]);

angular.module('myApp', ['emailParser'])

.controller('MyController', ['$scope', 'EmailParser',

function($scope, EmailParser) {

// 设置监听

$scope.$watch('emailBody', function(body) {

if (body) {

$scope.previewText = EmailParser.parse(body, {

to: $scope.to

});

}

});

}

]);

</script>

</body>

</html>

在AngularJS中，我们的确有手动运行模板编译的能力。例如，插值允许基于作用域上的某个条件实时更新文本字符串。

要在字符串模板中做插值操作，需要在你的对象中注入$interpolate服务。在下面的例子中，我们将会将它注入到一个控制器中：

angular.module('myApp', []).controller('MyController',

function($scope, $interpolate) {

// 我们同时拥有访问$scope和$interpolate服务的权限

});

$interpolate服务是一个可以接受三个参数的函数，其中第一个参数是必需的。

$interpolate服务返回一个函数，用来在特定的上下文中运算表达式。   
设置好这些参数后，就可以在控制器中进行字符插值的操作了。例如，假设我们希望可以在电子邮件的正文中进行实时编辑，当文本发生变化时进行字符插值操作并将结果展示出来。

<div ng-controller="MyController">

<input ng-model="to"

type="email"

placeholder="Recipient" />

<textarea ng-model="emailBody"></textarea>

<pre>{{ previewText }}</pre>

</div>

由于控制器内部设置了一个需要每次变化都重新进行字符插值的自定义输入字段，因此需要设置一个$watch来监听数据的变化。

简而言之，$watch函数会监视scope上的某个属性。只要属性发生变化就会调用对应的函数。可以使用watch函数在scope上某个属性发生变化时直接运行一个自定义函数。

在控制器中，我们设置了$watch来监视邮件正文的变化，并将emailBody属性的值进行字符插值后的结果赋值给previewText属性。

angular.module('myApp', [])

.controller('MyController', function($scope, $interpolate) {

// 设置监听

$scope.$watch('emailBody', function(body) {

if (body) {

var template = $interpolate(body);

$scope.previewText =

template({to: $scope.to});

}

};

});

在

现在，在{{ previewText }}内部的文本中可以将{{ to }}当做一个变量来使用，并对文本的变化进行实时更新。

用startSymbol()方法可以修改标识开始的符号。这个方法接受一个参数。

用endSymbol()方法可以修改标识结束的符号。这个方法也接受一个参数。

如果要修改这两个符号的设置，需要在创建新模块时将$interpolateProvider注入进去。

angular.module('emailParser', [])

.config(['$interpolateProvider', function($interpolateProvider) {

$interpolateProvider.startSymbol('\_\_');

$interpolateProvider.endSymbol('\_\_');

}])

.factory('EmailParser', ['$interpolate', function($interpolate) {

// 处理解析的服务

return {

parse: function(text, context) {

var template = $interpolate(text);

return template(context);

}

};

}]);

现在，我们已经创建了一个模块，可以将它注入到应用中，并在邮件正文的文本中运行这个邮件解析器：

angular.module('myApp', ['emailParser'])

.controller('MyController', ['$scope', 'EmailParser',

function($scope, EmailParser) {

// 设置监听

$scope.$watch('emailBody', function(body) {

if (body) {

$scope.previewText = EmailParser.parse(body, {

to: $scope.to

});

}

});

}]);

现在用自定义的 \_\_ 符号取代默认语法中的 {{ }} 符号来请求插值文本。   
由于我们将表达式开始和结束的符号都设置成了\_\_，因此需要将HTML修改成用这个符号取代{{ }}的版本，

<div id="emailEditor">

<input ng-model="to"

type="email"

placeholder="Recipient" />

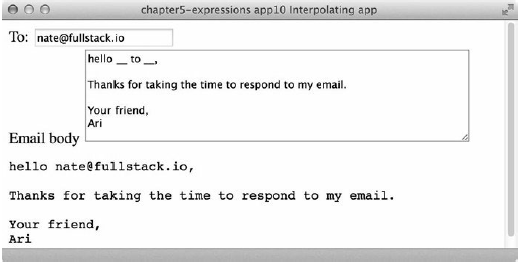
<textarea ng-model="emailBody"></textarea>

</div>

<div id="emailPreview">

<pre>\_\_ previewText \_\_</pre>

</div>



# 第七章 过滤器

## 7.1 过滤器的意义

在我们开发中经常需要在页面显示给用户的信息需要一定处理格式化，才能显示给用户。比如时间本地化，或者yyyy-MM-dd HH:mm:ss格式，数字精度格式化，本地化，人名格式化等等。在angularjs中为我们提供了叫filter的指令，让我们能够很轻易就能做到着一些列的功能，angularjs内部为我们提供了number等很多内置的filter。并且我们能够很轻易的自定义自己的领域filter。

如：

<!DOCTYPE html>

<html ng-app="app">

<head>

<meta charset="utf-8" />

<title>AngularJS Expressions</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="test">

num:

<input ng-model="num" />

<br/> {{num | number}}

<br/> {{num | number:2}}

<br/> first name:

<input ng-model="person.first" />

<br/> last name:

<input ng-model="person.last" />

<br/> name: {{person | fullname}}

<br/> name: {{person | fullname:"- -"}}

<br/> name: {{person | fullname:" " | uppercase }}

</div>

<script>

angular.module("app", [])

.controller("test", function() {

//

})

.filter("fullname", function() {

var filterfun = function(person, sep) {

sep = sep || " ";

person = person || {};

person.first = person.first || "";

person.last = person.last || "";

return person.first + sep + person.last;

};

return filterfun;

});

</script>

</body>

</html>

## 7.2 过滤器的使用方法

**（1）在表达式中应用Filters（过滤器）**

需要遵循格式如下：

{{ expression | filter }}

例如：

{{ 12 | currency }} 🡺 $12.00

**（2）在输出结果中应用Filters（过滤器）**

通俗点讲就是Filter的叠加，angularjs filters支持链式写法，和powershell或者其他操作系统外壳语言的管道式模型一样，形如 value | filter1 | filter2。也就是前一filter的输出结果作为后一filter的输入数据源.

需要遵循格式如下：

{{ expression | filter1 | filter2 | ... }}

即 表达式(expression)使用filter1过滤后再使用filter2过滤...

**（3）带参数的Filter**

Filter后面可以跟一个或多个参数，用来帮助实现特殊要求、需求的filter.

需要遵循格式如下：

{{ expression | filter:argument1:argument2:... }}

示例：

{{ 1234 | number:2 }} 🡺 1,234.00

**（4）在controllers， services和drictives中使用filter**

可以在AngularJS的controller, service或者driective中使用filter, 这时候你需要将依赖的filter名字加入到controller, service或者directive的依赖中去。

在controller中直接使用filter， 这样controller可以根据自身需要而适时调用filter，如：

<!doctype html>

<html ng-app="myApp">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js" ></script>

</head>

<body>

<div ng-controller="myAppCtrl">

<div>{{kkk}}</div>

</div>

<script type="text/javascript">

var myAppModule = angular.module("myApp", []);

myAppModule.filter("reverse", function() {

return function(input, uppercase) {

var out = "";

for (var i = 0; i < input.length; i++) {

out = input.charAt(i) + out;

}

if (uppercase) {

out = out.toUpperCase();

}

return out;

}

});

myAppModule.controller("myAppCtrl", function($scope,reverseFilter) {

$scope.kkk = reverseFilter('xingoo',true);

}

);

</script>

</body>

</html>

在服务、directive中使用filter也是同样的道理。

如果要在controller中使用多个filter，此时我们需要使用$filter服务，使用方法如下：

<!doctype html>

<html ng-app="myApp">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="myAppCtrl">

<div>{{kkk}}</div>

<div>{{ttt}}</div>

</div>

<script type="text/javascript">

var myAppModule = angular.module("myApp", []);

myAppModule.filter("reverse", function() {

return function(input, uppercase) {

var out = "";

for (var i = 0; i < input.length; i++) {

out = input.charAt(i) + out;

}

if (uppercase) {

out = out.toUpperCase();

}

return out;

}

});

myAppModule.controller("myAppCtrl", function($scope, $filter) {

$scope.kkk = $filter('reverse')('xingoo', true);

$scope.ttt = $filter('currency')(123534);

});

</script>

</body>

</html>

## 7.3 AngularJS内置过滤器

在angular中内置了几个常用的filter,可以简化我们的操作。

过滤器使用 '|' 符号，概念有点类似于linux中的管道。

**（1）filter（过滤）**

filter可以根据条件过滤数据，如：

<!DOCTYPE html>

<html ng-app>

<head>

<meta charset="utf-8" />

<title>AngularJS Filter</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div>{{[{name:'coolcao',age:23},{name:'lily',age:20},{name:'tom',age:22}] | filter:'tom'}}</div>

</body>

</html>

结果：



**（2）date : 日期格式化**

在[系统](http://www.2cto.com/os/)后台返回的数据中，时间字段，我们可能使用的是时间戳，Long型，在页面显示中肯定格式化为类似于‘2012-12-12 12:12:12’的字符串，使用date过滤器即可

{{1423130269432 | date:'yyyy-MM-dd HH:mm:ss'}}

显示结果：

2015-02-05 17:57:49

**（3）number : 数字格式化**

表示保留几位小数点，如：

{{ 3.1415926 | number:2 }}

显示结果：



|  |  |
| --- | --- |
|  |  |

**（4）orderBy 排列**

{{[{name:'coolcao',age:23},{name:'lily',age:20},{name:'tom',age:22}] | orderBy:'age'}}

显示结果：



默认是升序排列，如果要倒序：

{{[{name:'coolcao',age:23},{name:'lily',age:20},{name:'tom',age:22}] | orderBy:'age':true}}

显示结果：



另外，

{{[{name:'coolcao',age:23},{name:'lily',age:22},{name:'tom',age:22}] | orderBy:['age','name']}}

显示结果：



如果age相同，按照name进行排序

**（5）json格式化**

{{[{name:'coolcao',age:23},{name:'lily',age:22},{name:'tom',age:22}] | json}}

显示结果：



注意：

输入是js的对象（非标准json），输出的是标准的json字符串（属性名称会用双引号）

**（6）大小写转换：uppercase,lowercase**

{{'abc' | uppercase}}

显示结果：ABC

注意：uppercase,lowercase只能对字符串进行过滤转换

**（7）currency : 货币的格式化**

有时候我们需要把数字显示为货币的形式方便展示，可以使用currency进行格式化，如：

{{1000 | currency:'HK'}}

显示结果：



注意：currency默认为$.

**（8）limitTo(限制个数)**

{{[{name:'coolcao',age:23},{name:'lily',age:22},{name:'tom',age:22}] | limitTo:2}}

显示结果：



## 7.4 自定义filter（过滤器）

AngularJS编写自定义过滤器的形式和AngularJS的factory service非常相像，一定记得它返回一个对象或者是一个函数即可，编写的时候，只需要一个带有一个以上参数的函数即可。

格式大致如下：

app.filter('filter(过滤器)名称', function() {

return function(需要过滤的对象, 过滤器参数1, 过滤器参数2, ...) {

//...执行业务逻辑代码

return 处理后的对象;

}

});

内部返回的方法包含了两个参数，一个是输入的值，就是我们过滤器接受的值。

如：

如果想要实现下面的过滤器：

　　name | reverse

　　则input就是其中name代表的值。

　　后面的参数是可选的。

如：

<!doctype html>

<html ng-app="myApp">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js" ></script>

</head>

<body>

<div ng-controller="myAppCtrl">

name:{{ name }}

<br> reverse name:{{ name | reverse }}

<br> reverse&uppercase name:{{ name | reverse:true }}

</div>

<script type="text/javascript">

var myAppModule = angular.module("myApp", []);

myAppModule.controller("myAppCtrl", ["$scope",

function($scope) {

$scope.name = "xingoo";

}

]);

myAppModule.filter("reverse", function() {

return function(input, uppercase) {

var out = "";

for (var i = 0; i < input.length; i++) {

out = input.charAt(i) + out;

}

if (uppercase) {

out = out.toUpperCase();

}

return out;

}

});

</script>

</body>

</html>

# 第八章 指令

一个指令用来引入新的HTML语法。指令是DOM元素上的标记，使元素拥有特定的行为。举例来说，静态的HTML不知道如何来创建和展现一个日期选择器控件。让HTML能识别这个语法，我们需要使用指令。指令通过某种方法来创建一个能够支持日期选择的元素。如果你写过AngularJS的应用，那么你一定已经使用过指令，不管你有没有意识到。你肯定已经用过简单的指令，比如 ng-mode, ng-repeat, ng-show等。这些指令都赋予DOM元素特定的行为。例如，ng-repeat 重复特定的元素，ng-show 有条件地显示一个元素。如果你想让一个元素支持拖拽，你也需要创建一个指令来实现它。指令背后基本的想法很简单。它通过对元素绑定事件监听或者改变DOM而使HTML拥有真实的交互性。

## 8.1 指令的作用

指令的作用：实现语义化标签。如：

<!doctype html>

<html ng-app="app">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<hello></hello>

<script type="text/javascript">

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

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

return {

restrict: 'E',

template: '<div>Hi there</div>',

replace: true

};

});

</script>

</body>

</html>

对于以上代码里面的<hello>标签，浏览器显然是不认识的，它唯一能做的事情就是无视这个标签。那么，为了让浏览器能够认识这个标签，我们需要使用Angular来定义一个hello指令（本质上说就是自己来把<hello>替换成浏览器能识别的那些标准HTML标签）。

审查元素可以看到，<hello>标签已经被<div>Hi there</div>这个标签替换掉了，这也是以上JS代码里面replace:true这行配置的作用，代码里面的template配置项当然就是我们要的div标签啦，至于restrict:'E'这个配置项的含义，将在下文详细讲解。

## 8.2 内置指令

所有的内置指令的前缀都为ng，不建议自定义指令使用该前缀，以免冲突。

### 8.2.1 ng-model

将表单控件和当前作用域的属性进行绑定。如：

<!doctype html>

<html ng-app="myapp">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller="MainCtrl">

<div>{{text}}</div>

<script type="text/javascript">

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

app.controller('MainCtrl', function($scope) {

$scope.text = '绑定作用域';

});

</script>

</body>

</html>

### 8.2.2 ng-init

该指令被调用时会初始化内部作用域。这个指令一般会出现在比较小的应用中，比如：

<!doctype html>

<html ng-app>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-init="job='fighter'">

I'm a/an {{job}}

</div>

</body>

</html>

### 8.2.3 ng-app

每一次用AngularJS都离不开这个指令。

声明了ng-app的元素会成为$rootScope的起点，而$rootScope是作用域链的根，通常声明在<html>。也就是说根下的作用域都可以访问它。但是，不建议过度使用$rootScope，免得全局变量满天飞，效率又差又难管。下面是一个例子:

<!doctype html>

<html ng-app="myApp">

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

{{ someProperty }}

</body>

<script>

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

.run(function($rootScope) {

$rootScope.someProperty = 'hello computer';

});

</script>

</html>

### 8.2.4 ng-controller

我们用这个指令在一个DOM元素上装上controller。

记得AngularJS 1.2.x时还可以这样定义controller来着...

function ohMyController($scope) {

//...

}

AngularJS 1.3.x中禁止了这种方式，因为这种方式会让controller满天飞，分不清层次，所有东西都挂在$rootScope上...

ng-controller必须有一个表达式作为参数，另外通过$scope来继承上级$scope的方法和属性，$rootScope也包括在内。下面只是一个简单的例子，ancestor无法访问child的作用域。

<!doctype html>

<html ng-app="myApp">

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="AncestorController">

{{ ancestorName }} {{ childName }}

<div ng-controller="ChildController">

{{ ancestorName }} {{ childName }}

</div>

</div>

<script>

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

.controller('ChildController', function($scope) {//子控制器

$scope.childName = 'child';

})

.controller('AncestorController', function($scope) {//父控制器

$scope.ancestorName = 'ancestor';

});

</script>

</body>

</html>

### 8.2.5 ng-form

（？？？）

### 8.2.6 ng-disabled

像这种只要出现则生效的属性，我们可以在AngularJS中通过表达式返回值true/false令其生效。

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<textarea ng-disabled="1+1==2">1+1=?</textarea>

<br />

<textarea ng-disabled="false">1+1=?</textarea>

</body>

</html>

### 8.2.7 ng-readonly

通过表达式返回值true/false将表单输入字段设为只读。  
弄个例子，3秒后变成只读.

<!doctype html>

<html ng-app="myApp">

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<input type="text" ng-readonly="stopTheWorld" value="stop the world after 3s" />

<script>

angular.module('myApp',[])

.run(function($rootScope, $timeout) {

$rootScope.stopTheWorld = false;

$timeout(function() {

$rootScope.stopTheWorld = true;

}, 3000)

});

</script>

</body>

</html>

### 8.2.8 ng-checked

这个是给<input type="checkbox" />用的，如：

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<input type="checkbox" ng-checked="someProperty" ng-init="someProperty = true" ng-model="someProperty">

</body>

</html>

### 8.2.9 ng-selected

给<select>里面的<option>用的，如：

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<label>

<input type="checkbox" ng-model="isFullStack"> I'm Full Stack Engineer

</label>

<select>

<option>Front-End</option>

<option>Back-End</option>

<option ng-selected="isFullStack">Full Stack !!!</option>

</select>

</body>

</html>

### 8.2.10 ng-show/ng-hide

根据表达式显示/隐藏HTML元素，注意是隐藏，不是从DOM移除，例如:

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-show="1+1 == 2">

1+1=2

</div>

<div ng-hide="1+1 == 2">

you can't see me.

</div>

</body>

</html>

### 8.2.11 ng-change、ng-options

注意：不是HTML那套onXXX之类的，而是ng-XXX。结合ng-model使用，以ng-change为例:

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<input type="text" ng-model="calc.arg" ng-change="calc.result = calc.arg\*2" />

<code>{{ calc.result }}</code>

<!--

html中有code这个元素，表示代码范例

-->

</body>

</html>

### 8.2.12 {{}}

其实这个也是一个指令，也许觉得和ng-bind差不多，但页面渲染略慢时可能会被看到。另外，{{}}的performance远不如ng-bind，只是用起来很方便。

### 8.2.13 ng-bind

ng-bind的行为和{{}}差不多，只是我们可以用这个指令来避免FOUC(Flash Of Unrendered Content)，也就是未渲染导致的闪烁。

### 8.2.14 ng-cloak

ng-cloak也可以为我们解决FOUC。 ng-cloak会将内部元素隐藏，直到路由调用对应的页面。

### 8.2.15 ng-if

如果ng-if中的表达式为false，则对应的元素整个会从DOM中移除而非隐藏，但审查元素时你可以看到表达式变成注释了。如果相进行隐藏，可以使用ng-hide。

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-if="false">

无法审查到该元素

</div>

<div ng-hide="1+1==2">

可审查

</div>

</body>

</html>

### 8.2.16 ng-switch

单独使用没什么意思，下面是例子:

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-switch on="1+2">

<p ng-switch-default>0</p>

<p ng-switch-when="1">1</p>

<p ng-switch-when="2">2</p>

<p ng-switch-when="3">3</p>

</div>

</body>

</html>

### 8.2.17 ng-repeat

遍历集合，给每个元素生成模板实例，

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<ul>

<li ng-repeat="char in

[{'alphabet': 'K'},

{'alphabet': 'A'},

{'alphabet': 'V'},

{'alphabet': 'L'},

{'alphabet': 'E'},

{'alphabet': 'Z'}] " ng-show="$odd">{{char.alphabet}}</li>

</ul>

</body>

</html>

$index: 表示序号

$first: 第一个

$last: 最后一个

$middle

even: 偶数序号

odd: 奇数序号

**8.2.18 ng-href**其实这样href和ng-href看不出什么区别，所以我们可以试试这样:

<!doctype html>

<html ng-app='myApp'>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<ul ng-init="myHref=''">

<li><a ng-href="{{ myHref }}">{{linkText}}</a>

</li>

<li><a href="{{ myHref }}">可以点击，但不一定是正确的地址</a>

</li>

</ul>

</body>

<script>

angular.module('myApp', [])

.run(function($rootScope, $timeout) {

$rootScope.linkText = '尚未加载，您无法点击';

$timeout(function() {

$rootScope.linkText = '请点击';

$rootScope.myHref = 'http://www.baidu.com';

}, 2000);

});

</script>

</html>

**8.2.19 ng-src**

大同小异，即表达式生效前不要加载该资源。

<!doctype html>

<html ng-app='myApp'>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<img ng-src="{{imgSrc}}" />

</body>

<script>

angular.module('myApp', [])

.run(function($rootScope, $timeout) {

$timeout(function() {

$rootScope.imgSrc = 'img/1.jpg';

}, 2000);

});

</script>

</html>

**8.2.20 ng-class**

用作用域中的对象动态改变类样式，如:

<!doctype html>

<html ng-app='myApp'>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

<style>

.red {

background-color: red;

}

.blue {

background-color: blue;

}

</style>

</head>

<body>

<div ng-controller="CurTimeController">

<button ng-click="getCurrentSecond()">Get Time!</button>

<p ng-class="{red: x%2==0,blue: x%2!=0}">Number is: {{ x }}</p>

</div>

</body>

<script>

angular.module('myApp', [])

.controller('CurTimeController', function($scope) {

$scope.getCurrentSecond = function() {

$scope.x = new Date().getSeconds();

};

});

</script>

</html>

**8.2.21 ng-click**

ng-click指令属性给DOM元素注册了一个点击事件的监听器。当此DOM元素上有点击事件发生（即当此button或link被点击时），Angular.js就会执行表达式的内容，并相应地更新view。

<!doctype html>

<html ng-app>

<head>

<meta charset="utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<button ng-click="counter=counter + 1">Add one</button>

<span>Current counter: {{ counter }}</span>

</body>

</html>

## 8.3 自定义指令

创建一个自定义指令的完整格式如下：

angular.module('myApp', [])

.directive('myDirective', function() {

return {

restrict: 'EACM', //String，可选，默认为A

priority: 0, //Number，可选，默认为0

terminal: true, //Boolean，可选

template: function(tElement, tAttrs) {//String or Template Function，如'<h3>helloWorld</h3>'，可选

//返回代表模板的字符串

},

templateUrl: function(tElement, tAttrs) {//String or Template Function，如'/views/nprListItem.html'，可选

//返回一个外部HTML文件路径的字符串

},

replace: true, //Boolean，可选，默认为false

scope: true, //Boolean or Object，可选，默认为false，也可以是对象，如{}

transclude: true, //Boolean，可选，默认为false

controller: function(scope, element, attrs, transclude, otherInjectables) { //String or function，可以为string，即控制器名称，如'testCtrl'

//

},

controllerAs: 'textCtrl', //String,

require: 'hhh', //String,

link: function(scope, iElement, iAttrs) {

//

},

compile: function(tElement, tAttrs, transclude) { // 返回一个对象或连接函数，

return {

pre: function(scope, iElement, iAttrs, controller) {

//

},

post: function(scope, iElement, iAttrs, controller) {

//

}

}

return function postLink(/\*\*\*/) {

//

}

}

};

});

指令的构造函数会返回带有属性的JavaScript 对象。下面将对各个参数进行详细解析：

**1. restrict**

说明指令在HTML中的应用形式，值的选项有"A"、"E" 和 "C"， "M" ，分别代表 attribute、element、class和comment（默认值为"A"）。

具体使用方法如下：

作为元素（E）： <my-directive></my-directive>

作为属性（A）： <div my-directive="expression"></div>

作为类名（C）: <div class="my-directive:expression;"></div>

作为注释（M）： <!--directive:my-directive expression-->

**2. priority**

大多数指令会忽略这个参数，使用默认值0，但也有些场景设置高优先级是非常重要甚至是必须的。例如，ngRepeat将这个参数设置为1000，这样就可以保证在同一元素上，它总是在其他指令之前被调用。

**3. terminal**

这个参数用来停止运行当前元素上比本指令优先级低的指令。但同当前指令优先级相同的指令还是会被执行。例如：ngIf的优先级略高于ngView（它们操控的实际就是terminal参数），如果ngIf的表达式值为true，ngView就可以被正常执行，但如果ngIf表达式的值为false，由于ngView的优先级较低就不会被执行。

**4. template**

template参数是可选的，必须被设置为以下两种形式之一：

（1） 一段HTML文本；

（2）一个可以接受两个参数的函数，参数为tElement和tAttrs，并返回一个代表模板的字符串。tElement和tAttrs中的t代表template，是相对于instance的。

**5. templateUrl**

templateUrl是可选的参数，可以是以下类型：

（1）一个代表外部HTML文件路径的字符串；

（2）一个可以接受两个参数的函数，参数为tElement和tAttrs，并返回一个外部HTML文件路径的字符串。

无论哪种方式，模板的URL都将通过ng内置的安全层，特别是$getTrustedResourceUrl，这样可以保护模板不会被不信任的源加载。 默认情况下，调用指令时会在后台通过Ajax来请求HTML模板文件。加载大量的模板将严重拖慢一个客户端应用的速度。为了避免延迟，可以在部署应用之前对HTML模板进行缓存。

var expModule = angular.module('expanderModule', [])

expModule.directive('helloworld', function() {

return {

restrict: 'AE',

replace: true,

templateUrl: 'HelloWorld.html'

}

});

**6. replace**

这个属性指明生成的HTML内容是否会替换掉定义此指令的HTML元素。在下面的例子中，我们用 <hello-world></hello-world>的方式使用我们的指令，并且将 replace 设置成 true。所以，在指令被编译之后，生成的模板内容替换掉了 <hello-world></hello-world>。最终的输出是 <h3>Hello World!!</h3>。如果你将 replace 设置成 false，也就是默认值，那么生成的模板会被插入到定义指令的元素中。

<!doctype html>

<html ng-app="myapp">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<hello:world/>

<script type="text/javascript">

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

app.directive('helloWorld', function() {

return {

restrict: 'AE',

replace: 'true',

template: '<h3>Hello World!!</h3>'

};

});

</script>

</body>

</html>

**7. scope**

scope参数是可选的，可以被设置为true或一个对象。默认值是false。

当设置为布尔值时，指令获取它父节点的controller的scope。如：

<!doctype html>

<html ng-app="myapp">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller="MainCtrl">

父scope:

<input type="text" ng-model="text" />

<hello-world></hello-world>

<script type="text/javascript">

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

app.directive('helloWorld', function() {

return {

scope: true, //为false时，效果也是一样

restrict: 'AE',

replace: true,

template: '<p>{{text}}</p>'

};

});

app.controller('MainCtrl', function($scope) {

$scope.text = '';

});

</script>

</body>

</html>

从上面的例子中我们看不出scope为true或false时的区别，下面的例子将展示出它们的区别：

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

app.directive('helloWorld', function() {

return {

scope: false, // 或true

restrict: 'AE',

replace: true,

template: '<p>{{text}}</p>',

link: function(scope, elem, attr) {

scope.text = '为false时，我的改变会影响到父节点；为true时则不会';

}

};

});

app.controller('MainCtrl', function($scope) {

$scope.text = '';

});

综合上面两个例子我们可以用下面两句话来概括scope设置为true/false的区别：

**false:**继承但不隔离；**true:**继承并隔离。

另外，为了防止污染父节点上的作用域，我们还可以创建一个孤立存在的作用域，此时应将scope的值设置为{}，如：

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

app.directive('helloWorld', function() {

return {

scope: {}, // use a new isolated scope

restrict: 'AE',

replace: true,

template: '<p>{{text}}</p>',

link: function(scope, elem, attr) {

scope.text = '创建一个孤立的子scope';

}

}

});

app.controller('MainCtrl', function($scope) {

$scope.text = '';

});

上面的例子中创建了一个孤立的作用域，为了能够使孤立作用域也能和外界通信，AngularJS 提供了三种方式用来打破孤立作用域“孤立”这一限制。具体可看2.2.4.

**8. transclude**

transclude是一个可选的参数，默认值是false。表示自定义指令是否复制原始标记中的内容。例如，下面展示的“'hello'”指令设置了transclude 为 true，因为'hello'元素包含其他HTML 元素。

<!doctype html>

<html ng-app="app">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<hello>

<br/>

<span>原始的内容，</span>

<br/>

<span>还会在这里。</span>

</hello>

<hello>

</hello>

<script type="text/javascript">

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

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

return {

restrict: 'E',

template: '<div>Hi there <span ng-transclude></span></div>',

transclude: true

};

});

</script>

</body>

</html>

**9. controller**

controller参数可以是一个字符串或一个函数。当设置为字符串时，会以字符串的值为名字，来查找注册在应用中的控制器的构造函数.如果在指令内部通过匿名构造函数的方式来定义一个内联的控制器时, 有一些特殊的服务可以被注入到指令当中。这些服务有：

**（1）$scope**

与指令元素相关联的当前作用域。

**（2）$element**

当前指令对应的元素。

**（3） $attrs**

由当前元素的属性组成的对象。

<div id="aDiv"class="box"></div>

具有如下的属性对象：

{

id: "aDiv",

class: "box"

}

**（4） $transclude**

**10. controllerAs**

(???)

**11. require**

(???)

**12. link**

指令生成出的模板其实没有太多意义，除非它在特定的scope下编译。默认情况下，指令并不会创建新的子scope。更多的，它使用父scope。也就是说，如果指令存在于一个controller下，它就会使用这个controller的scope。 如何运用scope，我们要用到一个叫做 link 的函数。它由指令定义对象中的link属性配置，负责执行DOM 操作和注册事件监听器等。

下面的例子中，当用户在一个输入框中输入一种颜色的名称时，Hello World 文字的背景色自动发生变化。同时，当用户在 Hello World 文字上点击时，背景色变回白色。

<!doctype html>

<html ng-app="myapp">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller="MainCtrl">

<input type="text" ng-model="color" placeholder="Enter a color" />

<hello-world></hello-world>

<script type="text/javascript">

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

app.directive('helloWorld', function() {

return {

restrict: 'AE',

replace: true,

template: '<p style="background-color:{{color}}">Hello World</p>',

link: function($scope, $elem, $attrs) {

$elem.bind('click', function() {

$elem.css('background-color', 'white');

$scope.$apply(function() {

$scope.color = "white";

});

});

$elem.bind('mouseover', function() {

$elem.css('cursor', 'pointer');

});

}

};

});

app.controller('MainCtrl',function ($scope) {

$scope.color = '';

});

</script>

</body>

</html>

我们注意到指令定义中的 link 函数。 它有三个参数：

**（1） scope** – 指令的scope。在我们的例子中，指令的scope就是父controller的scope。scope 变量在初始化时是不被定义的，link 方法会注册监视器监视值变化事件。

**（2）elem** – 指令的jQLite(jQuery的子集)包装DOM元素。如果你在引入AngularJS之前引入了jQuery，那么这个元素就是jQuery元素，而不是jQLite元素。由于这个元素已经被jQuery/jQLite包装了，所以我们就在进行DOM操作的时候就不需要再使用 $()来进行包装。

link 方法一般通过jQuery 操作实例（如果没有加载jQuery，还可以使用Angular's jqLite ）。

**（3）attr** – 一个包含了指令所在元素的属性的标准化的参数对象。举个例子，你给一个HTML元素添加了一些属性：，那么可以在 link 函数中通过 attrs.someAttribute 来使用它。

link函数主要用来为DOM元素添加事件监听、监视模型属性变化、以及更新DOM。在上面的指令代码片段中，我们添加了两个事件，click，和 mouseover。click 处理函数用来重置 <p> 的背景色，而 mouseover 处理函数改变鼠标为 pointer。在模板中有一个表达式 {{color}}，当父scope中的 color 发生变化时，它用来改变 Hello World 文字的背景色。

**13. compile**

compile 函数在 link 函数被执行之前用来做一些DOM改造。它接收下面的参数：

**（1）tElement** – 指令所在的元素

**（2）attrs** – 元素上赋予的参数的标准化列表

要注意的是 compile 函数不能访问 scope，并且必须返回一个 link 函数。但是如果没有设置 compile 函数，你可以正常地配置 link 函数，（有了compile，就不能用link，link函数由compile返回）。compile函数可以写成如下的形式：

app.directive('test', function() {

return {

compile: function(tElem, attrs) {

//do optional DOM transformation here

return function(scope, elem, attrs) {

//linking function here

};

}

};

});

|  |  |
| --- | --- |
|  |  |

大多数的情况下，你只需要使用 link 函数。这是因为大部分的指令只需要考虑注册事件监听、监视模型、以及更新DOM等，这些都可以在 link 函数中完成。 但是对于像 ng-repeat 之类的指令，需要克隆和重复 DOM 元素多次，在 link 函数执行之前由 compile 函数来完成。这就带来了一个问题，为什么我们需要两个分开的函数来完成生成过程，为什么不能只使用一个？要回答好这个问题，我们需要理解指令在Angular中是如何被编译的！

**指令是如何被编译的**

当应用引导启动的时候，(1)Angular开始使用 $compile 服务遍历DOM元素。这个服务基于注册过的指令在标记文本中搜索指令。一旦所有的指令都被识别后，(2)Angular执行他们的 compile 方法。如前面所讲的，(3)compile 方法返回一个 link 函数，被添加到稍后执行的 link 函数列表中。这被称为编译阶段。

如果一个指令需要被克隆很多次（比如 ng-repeat），compile函数只在编译阶段被执行一次，复制这些模板，但是link 函数会针对每个被复制的实例被执行。所以分开处理，让我们在性能上有一定的提高。这也说明了为什么在 compile 函数中不能访问到scope对象。 在编译阶段之后，就开始了链接（linking）阶段。在这个阶段，所有收集的 link 函数将被一一执行。指令创造出来的模板会在正确的scope下被解析和处理，然后返回具有事件响应的真实的DOM节点。

## 8.4 例子

<!doctype html>

<html ng-app="expanderModule">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

<style>

.expander {

border: 1px solid black;

width: 250px;

}

.expander>.title {

background-color: black;

color: white;

padding: .1em .3em;

cursor: pointer;

}

.expander>.body {

padding: .1em .3em;

}

</style>

</head>

<body ng-controller='SomeController'>

<accordion>

<expander class='expander' ng-repeat='expander in expanders' expander-title='expander.title'>

{{expander.text}}

</expander>

</accordion>

<script type="text/javascript">

var expModule = angular.module('expanderModule', [])

expModule.directive('accordion', function() {

return {

restrict: 'EA',

replace: true,

transclude: true,

template: '<div ng-transclude></div>',

controller: function() {

var expanders = [];

this.gotOpened = function(selectedExpander) {

angular.forEach(expanders, function(expander) {

if (selectedExpander != expander) {

expander.showMe = false;

}

});

}

this.addExpander = function(expander) {

expanders.push(expander);

}

}

}

});

expModule.directive('expander', function() {

return {

restrict: 'EA',

replace: true,

transclude: true,

require: '^?accordion',

scope: {

title: '=expanderTitle'

},

template: '<div>' + '<div class="title" ng-click="toggle()">{{title}}</div>' + '<div class="body" ng-show="showMe" ng-transclude></div>' + '</div>',

link: function(scope, element, attrs, accordionController) {

scope.showMe = false;

accordionController.addExpander(scope);

scope.toggle = function toggle() {

scope.showMe = !scope.showMe;

accordionController.gotOpened(scope);

}

}

}

});

expModule.controller("SomeController", function($scope) {

$scope.expanders = [{

title: 'Click me to expand',

text: 'Hi there folks, I am the content that was hidden but is now shown.'

}, {

title: 'Click this',

text: 'I am even better text than you have seen previously'

}, {

title: 'Test',

text: 'test'

}];

});

</script>

</body>

</html>

这个例子主要的难点在于如何在子Expander里面访问外层Accordion的scope中的数据。

<!doctype html>

<html ng-app="expanderModule">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

<style>

.expander {

border: 1px solid black;

width: 250px;

}

.expander>.title {

background-color: black;

color: white;

padding: .1em .3em;

cursor: pointer;

}

.expander>.body {

padding: .1em .3em;

}

</style>

</head>

<body>

<div ng-controller='SomeController'>

<expander class='expander' expander-title='title'>

{{text}}

</expander>

</div>

<script type="text/javascript">

var expanderModule = angular.module('expanderModule', [])

expanderModule.directive('expander', function() {

return {

restrict: 'EA',

replace: true,

transclude: true,

scope: {

title: '=expanderTitle'

},

template: '<div>' + '<div class="title" ng-click="toggle()">{{title}}</div>' + '<div class="body" ng-show="showMe" ng-transclude></div>' + '</div>',

link: function(scope, element, attrs) {

scope.showMe = false;

scope.toggle = function toggle() {

scope.showMe = !scope.showMe;

}

}

}

});

expanderModule.controller('SomeController', function($scope) {

$scope.title = '点击展开';

$scope.text = '这里是内部的内容。';

});

</script>

</body>

</html>

<!doctype html>

<html ng-app="app">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller='MyController'>

<div ng-repeat='thing in things'>

{{thing}}.

<hello></hello>

</div>

<script type="text/javascript">

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

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

return {

restrict: 'E',

template: '<span>Hi there</span>',

replace: true

};

});

appModule.controller('MyController', function($scope) {

$scope.things = [1, 2, 3, 4, 5, 6];

});

</script>

</body>

</html>

# 第九章 AngularJS模块加载

一个模块就是一系列配置和代码块的集合，它们是在启动阶段就附加在到应用上的。一个最简单的模块由两类代码块集合组成的：

配置代码块 - 在提供程序注册和配置阶段执行。只有提供程序和常量可以被注入到配置块中。这是为了防止服务在被配置好之前就被提前初始化。

运行代码块 - 在注入器被创建后执行，被用来启动应用的。只有实例和常量能被注入到运行块中。这是为了防止在运行后还出现对系统的配置。

## 9.1 配置块

我们有时候希望在AngularJS应用启动执行之前,确切的说是在index页面加载之前就取到一个包含用户权限等数据.那么这个数据存放在什么地方?因为要在AngularJS启动之前就存在这些数据,route的路由等才能进行校验,不然等相应的一些Controller模块加载后才有permission,用户已经进入页面了,这岂不是为时已晚.

能让我在AngularJS启动之前就将数据存入到AngularJS应用中.的就是通过模块配置了,AngularJS模块可以在被加载和执行之前对自身进行配置.

在模块的加载流程中,AngularJS可以在模块注册时对其进行配置,这是AngularJS的工作流程中唯一可以在应用启动前对其进行修改的阶段.代码类似于:

angular.module('myApp', ['ngRoute'])

.config(function($provide) {

// do something

});

我们还可以在config中定义service用来存一个变量或者一个提前定义一个directive.如：

angular.module('myApp', [])

.factory('myFactory', function() {

var service = {};

return service;

})

.directive('myDirective', function() {

return {

template: '<button>Click me</button>'

}

});

AngularJS会在编译时执行这些辅助函数。它们在功能上等同于下面的写法：

angular.module('myApp', [])

.config(function($provide, $compileProvider) {

$provide.factory('myFactory', function() {

var service = {};

return service;

});

$compileProvider.directive('myDirective', function() {

return {

template: '<button>Click me</button>'

};

});

});

注意到这里多出一个$compileProvider,它辅助AngularJS模块完成了加载的过程.这一过程中AngularJS以书写和注册的顺序来执行函数,所以无法注入一个还没注册的提供者.

在使用config对模块进行配置的时候,不是所有的对象都可以被注入到config()函数中,只有少数几种可以注入:提供者和常量.如果将一个服务注入进去,造成的结构就是在真正对其配置之前就将它实例化了。

config()的代码会按照顺序执行,可以定义在多个配置模块,通过这些模块我们可以自定义一些服务,比如权限List,Url以及API加密.

1.3及之后的版本，所有的.config块都在.provider运行之后运行。这意味着不能使用.config来配置“下游”provider。

唯一例外的是constant()方法，这个方法总会在所有配置块之前被执行。

## 9.2 运行块

和配置块不同，运行块在注入器创建之后被执行，它是所有AngularJS应用中第一个被执行的方法。

运行块是AngularJS中与main方法最接近的概念。运行块中的代码块通常很难进行单元测试，它是和应用本身高度耦合的。

运行块通常用来注册全局的事件监听器。例如，我们会在.run()块中设置路由事件的监听器以及过滤未经授权的请求。

假设我们需要在每次路由发生变化时，都执行一个函数来验证用户的权限，放置这个功能唯一合理的地方就是run方法：

angular.module('myApp', ['ngRoute'])

.run(function($rootScope, AuthService) {

$rootScope.$on('$routeChangeStart', function(evt, next, current) {

// 如果用户未登录

if (!AuthService.userLoggedIn()) {

if (next.templateUrl === "login.html") {

// 已经转向登录路由因此无需重定向

} else {

$location.path('/login');

}

}

});

});

run()函数接受个参数：

initializeFn（函数）AngularJS在注入器创建后会执行这个函数。

## 9.3 模块的依赖

模块可以把其他模块列为它的依赖。“依赖某个模块”意味着需要把这个被依赖的模块需要在本块模块之前被加载。换句话说被依赖模块的配置块会在本模块配置块前被执行。运行块也是一样。任何一个模块都只能被加载一次，即使它被多个模块依赖。

## \* 模板加载

Angularjs作为mvc（或者说mvvm)框架，同样具备模板这一基本概念。

Angularjs加载模板的顺序为：内存加载---AJAX加载。

### \*.1 内存加载

模板本质上是字符串，把字符串直接写入内存，加载时直接从内存获取，速度会更快，有两种方式显式启用内存加载。

（1）通过使用$templateCache service来实现，如：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<title></title>

</head>

<body ng-controller="myCtrl">

<div ng-include="'lovestory.html'" class="well"></div>

<script>

angular.module('myApp', [])

.controller('myCtrl', ['$scope', '$templateCache',

function($scope, $templateCache) {

var tmp = '<h4>lovestory</h4>' + '<p>这是直接调用$templateCache服务获取模板文件的方式</p>' + '<a href="http://www.baidu.com">服务启用templateCache方式</a>';

$templateCache.put('lovestory.html', tmp);

}

]);

</script>

</body>

</html>

$templateCache服务put方法负责向内存写入模板内容。

（2）通过script标签引入，如：

<!DOCTYPE html>

<html ng-app>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-include="'lovestory.html'" class="well"></div>

<script type="text/ng-template" id="lovestory.html">

<h4>lovestory</h4>

<p>这是script标签获取模板文件的方式</p>

<a href="http://www.baidu.com">标签启用templateCache方式</a>

</script>

</body>

</html>

这里需要注意，type="text/ng-template"是指明这是ng模板，id属性是指实际使用模板时的一个引用，标签之间的内容才是实际的模板内容。而且，需要注意，id绝对不是URL，这个script标签绝对不会发出HTTP请求，具体讨论见最后。   
实际应用模板时候，使用ID属性，即可从内存中获取对应数据。

使用ng-include的时候，应该注意，id相当于一个字符串，不是ng-expression，所以不要忘了加单引号。

### \*.2 AJAX加载

### \*.3 $templateCache 方法

$templateCache基于cacheFactory而来，接口保持一致，可以认为 $templateCache = $cacheFactory('template');

|  |  |
| --- | --- |
| 方法 | 功能 |
| put | 向内存写入模板内容 |
| get | 从内存获取模板内容 |
| remove | 传入key值，删除对应模板内容 |
| removeAll | 删除所有模板内容 |
| destroy | 解除key-value对应关系，但不释放内存 |
| info | 模板缓存对象的信息 |

# 第十章 多重视图和路由

## 10.1 多重视图和路由的执行步骤

在开发过程中，都有许多的页面，而这些页面都有相同的部分，比如页面的头部和尾部通常都是一样的，变化的都是主体部分，还有就是一些后端管理的一些项目，通常不变的都是头部，尾部和菜单部分，变化的都是右边的内容部分，使用AngularJs的多重视图和路由就可以很方便的实现这样的效果。在实现之前需要准备两个文件，一个是angular的主JS文件，另一个是angular的路由JS文件，如下：

<script type="text/javascript" src="js/angular.min.js"></script>

<script type="text/javascript" src="js/angular-route.min.js"></script>

在页面中引入这两个JS文件就可以使用angular多重视图和路由了，这里将实现这效果分为如下几步：

1.创建一个简单的布局文件

2.创建一个模块，可以写在当前的布局文件中也可以新开一个js文件，为了方便管理还是分开好点

3.创建路由规则

4.如果有用到控制器，再创建控制器

接下来就具体来实现这个效果：

<!--创建一个简单布局文――开始-->

<html>

<head>

<title>Angular JS Views</title>

<script type="text/javascript" src="js/angular.min.js"></script>

<script type="text/javascript" src="js/angular-route.min.js"></script>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app="mainApp">

<p><a href="#addStudent">Add Student</a>

</p>

<p><a href="#viewStudents">View Students</a>

</p>

<div ng-view></div>

<script type="text/ng-template" id="addStudent.html">

<h2> Add Student </h2> {{message}}

</script>

<script type="text/ng-template" id="viewStudents.html">

<h2> View Students </h2> {{message}}

</script>

</div>

<!--创建一个简单布局文――结束-->

<script>

<!--创建一个模块-->

var mainApp = angular.module("mainApp", ['ngRoute']);

<!--创建路由规则及控制器-->

mainApp.config(['$routeProvider',

function($routeProvider) {

$routeProvider.

when('/addStudent', {

templateUrl: 'addStudent.html',

controller: 'AddStudentController'

}).

when('/viewStudents', {

templateUrl: 'viewStudents.html',

controller: 'ViewStudentsController'

}).

otherwise({

redirectTo: '/addStudent'

});

}

]);

mainApp.controller('AddStudentController', function($scope) {

$scope.message = "This page will be used to display add student form";

});

mainApp.controller('ViewStudentsController', function($scope) {

$scope.message = "This page will be used to display all the students";

});

</script>

</body>

</html>

## 10.2 redirectTo的使用

它的作用就是做转发，如下：

var mainApp = angular.module("mainApp", ['ngRoute']);

mainApp.config(['$routeProvider',

function($routeProvider) {

$routeProvider.

when('/addStudent', {

templateUrl: 'addStudent.html',

controller: 'AddStudentController'

}).

when('/viewStudents', {

templateUrl: 'viewStudents.html',

controller: 'ViewStudentsController',

redirectTo: "/addStudent"///////////////////////////////////

}).

otherwise({

redirectTo: '/addStudent'///////////////////////////////////////////

});

}

]);

mainApp.controller('AddStudentController', function($scope) {

$scope.message = "This page will be used to display add student form";

});

mainApp.controller('ViewStudentsController', function($scope) {

$scope.message = "This page will be used to display all the students";

});

## 10.3 $routeParams

可以使用它来获取路由参数，比如我们将路由配置成如下格式：

when('/viewStudents/:name', {

templateUrl: 'viewStudents.html',

controller: 'ViewStudentsController'

}).

那么访问的格式就是/viewStudents/zhangsan,这样通过控制器来获取的参数格式就是{"name":"zhangsan"},如：

mainApp.controller('ViewStudentsController', function($scope, $routeParams) {

$scope.message = "This page will be used to display all the students";

var route = $routeParams;

console.log(route);

});

## 10.4 $location服务

可以用它来解析请求的URL中的相关信息.

$location服务下有如下方法：

**（1）path()**

获取当前页面的路径，也可以设置一个新的路径path("/")，这样就修改为“/”的路由了

$location.path();//获取

$location.path("/");//设置

如：

<html>

<head>

<title>Angular JS Views</title>

<script type="text/javascript" src="js/angular.min.js"></script>

<script type="text/javascript" src="js/angular-route.min.js"></script>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app="mainApp">

<p><a href="#addStudent">Add Student</a>

</p>

<p><a href="#viewStudents/zhangsan">View Students</a>

</p>

<div ng-controller="AppCtrl">

<button ng-click="go()">location</button>

</div>

<div ng-view></div>

<script type="text/ng-template" id="addStudent.html">

<h2> Add Student </h2> {{message}}

</script>

<script type="text/ng-template" id="viewStudents.html">

<h2> View Students </h2> {{message}}

</script>

</div>

<script>

var mainApp = angular.module("mainApp", ['ngRoute']);

mainApp.config(['$routeProvider',

function($routeProvider) {

$routeProvider.

when('/addStudent', {

templateUrl: 'addStudent.html',

controller: 'AddStudentController'

}).

when('/viewStudents/:name', {

templateUrl: 'viewStudents.html',

controller: 'ViewStudentsController'

}).

otherwise({

redirectTo: '/addStudent'

});

}

]);

mainApp.controller('AddStudentController', function($scope) {

$scope.message = "This page will be used to display add student form";

});

mainApp.controller('ViewStudentsController', function($scope, $routeParams) {

$scope.message = "This page will be used to display all the students";

var route = $routeParams;

console.log(route);

});

mainApp.controller('AppCtrl',function ($scope, $location, $timeout) {

$scope.go = function () {

console.log($location.path());

$timeout(function () {

$location.path('/viewStudents/Lisi');

},2000);

}

});

</script>

</body>

</html>

**（2）replace()**

如果你希望跳转后用户不能点击后退按钮（对于登录之后的跳转这种发生在某个跳转之后的再次跳转很有用），AngularJS提供了replace()方法来实现这个功能：

$location.path('/home');

$location.replace();

// 或者

$location.path('/home').replace();

注：这只是把跳转之前的地址替换成跳转后的地址而已

**（3）absUrl()**

absUrl()方法用来获取编码后的完整URL：

$location.absUrl()

**（4）hash()**

hash()方法用来获取URL中的hash片段：

$location.hash(); // 返回当前的hash片段

**（5）host()**

host()方法用来获取URL中的主机：

$location.host();// 当前URL的主机

**（6）port()**

port()方法用来获取URL中的端口号：

$location.port();// 当前URL的端口

**（7）protocol()**

protocol()方法用来获取URL中的协议：

$location.protocol();// 当前URL的协议  
**（8）search()**

search()方法用来获取URL中的查询串：

$location.search();

我们可以向这个方法中传入新的查询参数，来修改URL中的查询串部分：

// 用对象设置查询

$location.search({name: 'Ari', username: 'auser'});

// 用字符串设置查询

$location.search('name=Ari&username=auser');

search方法可以接受两个参数。

search（可选，字符串或对象）

这个参数代表新的查询参数。hash对象的值可以是数组。

paramValue（可选，字符串）

如果search参数的类型是字符串，那么paramValue会做为该参数的值覆盖URL当中的对应值。如果paramValue的值是null，对应的参数会被移除掉。

**（9）url()**

url()方法用来获取当前页面的URL：

$location.url(); // 该URL的字符串

如果调用url()方法时传了参数，会设置并修改当前的URL，这会同时修改URL中的路径、查询串和hash，并返回$location。

// 设置新的URL

$location.url('/home?name=Ari#hashthing');

url()方法可以接受两个参数。

 url（可选，字符串）

新的URL的基础的前缀。

 replace（可选，字符串）

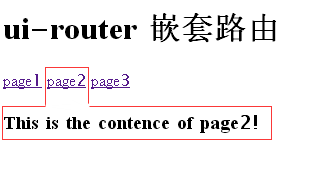
想要修改成的路径。

## 10.5 嵌套路由

ui-router和同属AngularJS框架一部分的ng-route一样强大. ui-router提供了让我们可以做路由嵌套和**视图命名**的特性。

它最大的作用是将web界面的设计分块了。不过从设计上已经有了变化。整个应用将变成只有一张网页，即单页应用。

下面的例子中我们将实现如下的效果：



**实现流程如下：**

（1）下载js文件，引入到main.html文件中。

（2）在html中，添加注入位置: <div ui-view="">。它是当被触发注入时，填充的位置。如：

<!--

main.html

-->

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>ui-router</title>

<script type="text/javascript" src="../js/angular.min.js" ></script>

<script type="text/javascript" src="../js/angular-ui-router.js" ></script>

<!--路由规则文件-->

<script type="text/javascript" src="App.js" ></script>

</head>

<body ng-app="myApp">

<h1>ui-router 嵌套路由</h1>

<!--标注位-->

<div ui-view = ""></div>

</body>

</html>

（3）创建其他块的相关html文件，并在相关位置添加触发器（<xxx ui-sref=”XX”>或<a href=”#/XX”>hhh</a>等），本例子中主要有四部分：pagetab、page1、page2、page3，如：

<!--

pagetab.html

-->

<div>

<div>

<span style="width: 100px;"><a ui-sref=".page1" href="">page1</a></span>

<span style="width: 100px;"><a href="#/pagetab/page2">page2</a></span>

<span style="width: 100px;"><a ui-sref=".page3" href="">page3</a></span>

</div>

<!--标注位-->

<div ui-view></div>

</div>

<!--

page1.html

-->

<div>

<h3>This is the contence of page1!</h3>

</div>

（4）在app.js文件中配置路由函数：

.config(function($stateProvider, $urlRouterProvider) {

});

如：

//创建模块，并添加依赖‘ui.router’

var myApp = angular.module("myApp", ['ui.router']);

//配置路由函数

myApp.config(function($stateProvider, $urlRouterProvider) {

$urlRouterProvider.when("", "/pagetab");//缺省时的页面

$stateProvider

.state("pagetab", {

url: '/pagetab',

templateUrl: 'pagetab.html'

})

//以下页面在pagetab页面显示

.state('pagetab.page1', {

url: '/page1',

templateUrl: 'page1.html'

})

.state('pagetab.page2', {

url: '/page2',

templateUrl: 'page2.html'

})

.state('pagetab.page3', {

url: '/page3',

templateUrl: 'page3.html'

})

});

# 第十一章 依赖注入

## 11.1 什么是依赖注入

依赖注入是一种软件设计模式，也叫控制反转。

所谓**依赖注入**，**是**指程序运行过程中，如果需要调用另一个对象协助时，无须在代码中创建被调用者，而是**依赖**于外部的**注入**。

AngularJS的依赖注入个人认为也可以这么理解：就是指通过给函数添加形参，使得caller传入的参数和callee接受的参数逻辑分离，使得函数通过依赖管理系统仅仅需要声明需要的协作对象，而不需要知道从哪里来，如何创建等。

原生Javascript/jquery是不支持依赖注入的，如：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

</head>

<body>

<button onclick="kkk()">OK</button>

</body>

<script>

var aa = {};

aa.alert\_t = function () {

console.log('kkk');

}

function kkk (aa) {

aa.alert\_t();

}

</script>

</html>

以上代码运行时是会报错的，因为原生的Javascript是不支持依赖注入的，而且在kkk这个方法的声明中aa只是一个形参，声明时kkk这个方法是不知道aa是从哪里来、如何创建的。如果要使用<script>元素中定义的aa对象那必须在调用kkk方法的地方引入实参。正确使用方法如下：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

</head>

<body>

<button onclick="kkk(aa)">OK</button>

</body>

<script>

var aa = {};

aa.alert\_t = function () {

console.log('kkk');

}

function kkk (aa) {

aa.alert\_t();

}

</script>

</html>

但是，如果使用AngluarJS那就简单多了，如：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js" ></script>

<title></title>

</head>

<body ng-controller="kkkCtrl">

<button ng-click="kkk()">OK</button>

</body>

<script>

angular.module('myApp',[])//声明模块，同时会自动创建本应用唯一的$inject 对象

.value('alert\_t',function () {//定义alert\_t常量，以便在后面的控制器中注入

console.log('kkk');

})

.controller('kkkCtrl', function ($scope, alert\_t) {//声明一个控制器，并注入alert\_t依赖

$scope.kkk = function () {

alert\_t();//调用alert\_t依赖

}

});

</script>

</html>

angularjs中与DI相关有angular.module()、angular.injector()、 [$injector](http://docs.angularjs.cn/api/auto/service/$injector)、[$provide](http://docs.angularjs.cn/api/auto/service/$provide)。对于一个DI容器来说，必须具备3个要素：**服务的注册**、**依赖关系的声明**、**对象的获取**。module和$provide相当于是服务的注册；injector用来获取对象（angular会自动完成依赖的注入）；依赖关系的声明在angular中有3种方式（将在11.4中详细讲解）。

**1、angular.module()创建、获取、注册angular中的模块**

The angular.module() is a global place for creating, registering and retrieving Angular modules.When passed two or more arguments, a new module is created. If passed only one argument, an existing module (the name passed as the first argument to module) is retrieved。

// 传递参数不止一个,代表新建模块;空数组代表该模块不依赖其他模块

var createModule = angular.module("myModule", []);

// 只有一个参数(模块名),代表获取模块

// 如果模块不存在,angular框架会抛异常

var getModule = angular.module("myModule");

// true,都是同一个模块

alert(createModule == getModule);

该函数既可以创建新的模块，也可以获取已有模块，是创建还是获取，通过参数的个数来区分。

angular.module(name, [requires], [configFn]);

name：字符串类型，代表模块的名称；

requires：字符串的数组，代表该模块依赖的其他模块列表，如果不依赖其他模块，用空数组即可；

configFn：用来对该模块进行一些配置。

现在我们知道如何创建、获取模块了，那么模块究竟是什么呢？官方的Developer Guide上只有一句话：You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.现在我还不太理解，大致就是说模块是一些功能的集合，如控制器、服务、过滤器、指令等子元素组成的整体。现在解释不了，先遗留。

**2、$provide和模块的关系**

The [$provide](http://docs.angularjs.cn/api/auto/service/$provide) service has a number of methods for registering components with the [$injector](http://docs.angularjs.cn/api/auto/service/$injector). Many of these functions are also exposed on [angular.Module](http://docs.angularjs.cn/api/ng/type/angular.Module).

之前提到过：module和provide是用来注册服务到injector中的。查看官方的API，可以看到$provide提供了provide()、constant()、value()、factory()、service()来创建各种不同性质的服务；angular.Module中也提供了这5个服务注册方法。其实2者功能是完全一样的，就是用来向DI容器注册服务到injector中。

官方API下的auto有[$provide](http://docs.angularjs.cn/api/auto/service/$provide) 和 [$injector](https://code.angularjs.org/1.2.21/docs/api/auto/service/$injector)，Implicit module which gets automatically added to each $injector.按照字面意思是说，每一个injector都有这2个隐含的服务。但1.2.25版本中，感觉没有办法获取injector中的[$provide](http://docs.angularjs.cn/api/auto/service/$provide)。不知道这是为什么?一般来说也不需要显示使用这个服务，直接使用module中提供的API即可。

var injector = angular.injector();

alert(injector.has("$provide"));//false

alert(injector.has("$injector"));//true

**3、angular.injector()**

使用angular.injector();也能获取到注入器，但是没有和模块绑定。这种做法是没有意义的，相当于是你创建了一个空的DI容器，里面都没有服务别人怎么用呢。正确的做法是，在创建注入器的时候，指定需要加载的模块。

// 创建myModule模块、注册服务

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

myModule.service('myService', function() {

this.my = 0;

});

// 创建herModule模块、注册服务

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

herModule.service('herService', function() {

this.her = 1;

});

// 加载了2个模块中的服务

var injector = angular.injector(["myModule","herModule"]);

alert(injector.get("myService").my);

alert(injector.get("herService").her);

如果加载了多个模块，那么通过返回的injector可以获取到多个模块下的服务。这个例子中如果只加载了myMoudle，那么得到的injector就不能访问herMoudle下的服务。这里**特别需要注意下：angular.injector()可以调用多次，每次都返回新建的injector对象**。

var injector1 = angular.injector(["myModule","herModule"]);

var injector2 = angular.injector(["myModule","herModule"]);

alert(injector1 == injector2);//false

**4、angular中三种声明依赖的方式**

angular提供了3种获取依赖的方式：inference、annotation、inline方式。

// 创建myModule模块、注册服务

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

myModule.service('myService', function() {

this.my = 0;

});

// 获取injector

var injector = angular.injector(["myModule"]);

// 第一种inference

injector.invoke(function(myService){alert(myService.my);});

// 第二种annotation

function explicit(serviceA) {alert(serviceA.my);};

explicit.$inject = ['myService'];

injector.invoke(explicit);

// 第三种inline

injector.invoke(['myService', function(serviceA){alert(serviceA.my);}]);

其中annotation和inline方式，对于函数参数名称没有要求，是推荐的做法；inference方式强制要求参数名称和服务名称一致，如果JS代码经过压缩或者混淆，那么功能会出问题，不建议使用这种方式。

## 11.2 可被依赖的组件

### 11.2.1 值

值是简单的JavaScript对象。

<!DOCTYPE html>

<html ng-app="mainApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<title></title>

</head>

<body ng-controller="CalcController">

<button ng-click="square()">OK</button>

</body>

<script>

//define a module

var mainApp = angular.module("mainApp", []);

//create a value object as "defaultInput" and pass it a data.

mainApp.value("defaultInput", 5);

//inject the value in the controller using its name "defaultInput"

mainApp.controller('CalcController', function($scope, defaultInput) {

$scope.square = function() {

console.log(defaultInput\*defaultInput);

}

});

</script>

</html>

### 11.2.2 工厂

工厂是用于返回函数的值。每当一个服务或控制器需求发生改变时，它通常使用一个工厂函数来计算并返回对应值，如11.2.3中的那个例子。

### 11.2.3 服务

如：

<!DOCTYPE html>

<html ng-app="mainApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<title></title>

</head>

<body ng-controller="CalcController">

<button ng-click="square()">OK</button>

</body>

<script>

//define a module

var mainApp = angular.module("mainApp", []);

//create a factory "MathService" which provides a method multiply to return multiplication of two numbers

mainApp.factory('MathService', function() {

var factory = {};

factory.multiply = function(a, b) {

return a \* b

}

return factory;

});

//inject the factory "MathService" in a service to utilize the multiply method of factory.

mainApp.service('CalcService', function(MathService) {

this.square = function(a) {

return MathService.multiply(a, a);

}

});

mainApp.controller('CalcController', function($scope, CalcService) {

$scope.square = function() {

console.log(CalcService.square(8));

}

});

</script>

</html>

在这个例子中，先是将工厂注入到服务中，然后再将服务注入到控制器中。

### 11.2.4 提供者

提供者所使用的AngularJS内部创建过程中配置阶段的服务，工厂等。提供者是一个特殊的工厂方法以及get()方法，用来返回值/服务/工厂。如：

<!DOCTYPE html>

<html ng-app="mainApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<title></title>

</head>

<body ng-controller="CalcController">

<button ng-click="square()">OK</button>

</body>

<script>

//define a module

var mainApp = angular.module("mainApp", []);

//create a service using provider which defines a method square to return square of a number.

mainApp.config(function($provide) {

$provide.provider('MathService', function() {

this.$get = function() {

var factory = {};

factory.multiply = function(a, b) {

return a \* b;

}

return factory;

};

});

});

mainApp.controller('CalcController', function($scope, MathService) {

$scope.square = function() {

console.log(MathService.multiply(8,9));

}

});

</script>

</html>

从上面的例子中可以看出使用provider这个服务要写的代码量比较大，因此AngularJS在模块对象上直接暴露了provider方法从而有了前面几种方法。

你可能会感到困惑：既然factorry和value能够节省那么多的代码，为什么还有人要使用provider。答案是provider允许我们进行一些配置，也就是说config阶段是你设置任何的provider的阶段。如：

<!DOCTYPE html>

<html ng-app="mainApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<title></title>

</head>

<body>

</body>

<script>

var mainApp = angular.module("mainApp", []);

mainApp.provider('greeting', function() {

var text = 'Hello, ';

this.setText = function(value) {

text = value;

};

this.$get = function() {

return function(name) {

console.log(text + name);

};

};

});

mainApp.config(function(greetingProvider) {

greetingProvider.setText("Howdy there, ");

});

mainApp.run(function(greeting) {

greeting('Ford Prefect');

});

</script>

</html>

需要注意的是在config阶段只能注入provider，以下的代码是错误的：

mainApp.config(function(greeting) {

greetingProvider.setText("Howdy there, ");

});

### 11.2.5 常量

## 11.3 可被注入的组件

* 控制器
* 控制器函数是可以被注入的，但是控制器本身是不能被注入到任何东西里面去的。这是因为控制器不是通过provider创建的。
* 指令
* factory
* 过滤器
* provider中的$get函数
* provider函数
* 服务

## 11.4 声明依赖的方式

### 11.4.1 推断式注入

最简单的获取依赖的方式,就是让函数参数名和依赖的名字一致。

虽然简单直接, 但这种方法在JavaScript压缩/混淆时会失效，因为压缩会重命名方法的参数名。

### 11.4.2 标记注入（Annotation）

为了可以在压缩代码后依然可以注入正确的 services, 函数需要通过 $inject 属性来注解. $inject 属性是一个数组,包含 需要注入的 service 名字.

在这种情况下,$inject数组中的值的顺序必须和要注入的参数的顺序一致。

### 11.4.3 内联注入

1. someModule.controller('MyController', ['$scope', 'dep1', 'dep2', **function**($scope, dep1, dep2) {
2. ...
3. $scope.aMethod = **function**() {
4. ...
5. }
6. ...
7. }]);

这避免了为 controllers 创建全局函数,并且在代码压缩时继续可用.

他们都是等价的，你可以在适当的地方互换使用.

## 11.5 注入器

注入器负责从我们通过$provide创建的服务中创建注入的实例。每一个AngularJS应用都有唯一一个$injector，当应用启动的时候它就被创造出来，你可以通过将$injector注入到任何可注入函数中来得到它（$injector知道如何注入它自己！）。

一旦你拥有了$injector，你可以通过调用get函数来获得任何一个已经被定义过的服务的实例。例如：

<!DOCTYPE html>

<html ng-app="mainApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<title></title>

</head>

<body ng-controller="CalcController">

<button ng-click="square()">OK</button>

</body>

<script>

//define a module

var mainApp = angular.module("mainApp", []);

//create a service using provider which defines a method square to return square of a number.

mainApp.config(function($provide) {

$provide.provider('MathService', function() {

this.$get = function() {

var factory = {};

factory.multiply = function(a, b) {

return a \* b;

}

return factory;

};

});

});

mainApp.controller('CalcController', function($scope, $injector) {

$scope.square = function() {

var aa = $injector.get('MathService');

console.log(aa);

console.log(aa.multiply(9,9));

}

});

</script>

</html>

# 第十二章 服务

## 12.1 服务概述

Angular 服务是为web应用执行特定任务的单例对象或方法。 注：如果组件是为了内容呈现的功能复用，那么服务就是为组件进行功能复用。

Angular有一些内建的服务（例如：[$http](http://docs.angularjs.org/api/ng.$http)），也可以创建自己的服务。内建的服务通常使用“$”开头（与jQuery类似）。

Services都是单例的，就是说在一个应用中，每一个Serice对象只会被实例化一次（用$injector服务)，主要负责提供一个接口把特定函数需要的方法放在一起。

在使用Angular服务的时候，只要将其注册成为Angular组件的一个依赖项目就可以了。Angular会自动对这个服务进行实例化、信赖处理等。（详细可参阅十一章 依赖注入）

## 12.2 内置的服务

Angular提供的标准服务组件有以下：

* $http：用于处理 XMLHttpRequest
* $location：提供当前URL的信息
* $q： 异步请求使用，promise/deferred模块
* $routeProvider：配置路由
* $log：日志服务

### 12.2.1 $http

我们看看最常用和调用后端的$http的用法：

$http({

method: 'GET',

url: 'dsds'

})

.success(function(data, status, headers, config) {

//

})

.error(function(data, status, headers, config) {

//

});

如：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<title>$http</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller="myCtrl">

<button ng-click="http()">$http</button>

<script>

angular.module('myApp', [])

.controller('myCtrl', function($scope, $http) {

$scope.http = function() {

$http({

method: 'GET',

url: 'http://www.w3cschool.cc/try/angularjs/data/Customers\_JSON.php'

})

.success(function(data, status, headers, config) {

console.log(data);

})

.error(function(data, status, headers, config) {

console.log('error');

});

}

});

</script>

</body>

</html>

$http有下面短方法：

* $http.get() 从web服务器上读取静态JSON数据。
* $http.head()
* $http.post()
* $http.put()
* $http.delete()
* $http.jsonp()

### 12.2.2 $resource

　　？？？？

### 12.2.3 $q

??????

…………………………………………

## 12.3 创建服务

创建一个新的服务，可以通过如下几个方式实现：

使用module提供的服务工厂注册：

angular.module('appName', [])

.factory('servicesName', function () {

var serviceInstance = {};

//定义服务的内容

return serviceInstance;

});

|  |  |
| --- | --- |
|  |  |

如：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<title>service</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller="myCtrl">

<input type="number" ng-model="num1" />

<span>

<select ng-model="calType">

<option value="1"> {{'+'}} </option>

<option value="2"> {{'-'}} </option>

<option value="3"> {{'\*'}} </option>

<option value="4"> {{'/'}} </option>

</select>

</span>

<input type="number" ng-model="num2" /><button ng-click="cal()"> = {{result}}</button>

</body>

<script>

angular.module('myApp', [])

.factory('calculator', function () {

var calculate = {};

calculate.add = function (num1, num2) {

return num1+num2;

}

calculate.minus = function (num1, num2){

return num1-num2;

}

calculate.multiplied = function (num1, num2){

return num1\*num2;

}

calculate.divided = function (num1, num2){

return num1/num2;

}

return calculate;

})

.controller('myCtrl', function ($scope, calculator) {

$scope.calType = '2';

$scope.cal = function () {

switch($scope.calType){

case '1':

$scope.result = calculator.add($scope.num1, $scope.num2);

break;

case '2':

$scope.result = calculator.minus($scope.num1, $scope.num2);

break;

case '3':

$scope.result = calculator.multiplied($scope.num1, $scope.num2);

break;

case '4':

$scope.result = calculator.divided($scope.num1, $scope.num2);

break;

}

}

});

</script>

</html>

上面这个应用中创建了一个'calculator'服务，……

使用$provide服务:

angular.module('appName', [])

.config(function($provide) {

$provide.factory('serviceName', function() {

var serviceInstance = {};

//定义服务的内容

return serviceInstance;

});

});

如：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<title>service</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller="myCtrl">

<input type="number" ng-model="num1" />

<span>

<select ng-model="calType">

<option value="1"> {{'+'}} </option>

<option value="2"> {{'-'}} </option>

<option value="3"> {{'\*'}} </option>

<option value="4"> {{'/'}} </option>

</select>

</span>

<input type="number" ng-model="num2" />

<button ng-click="cal()"> = {{result}}</button>

</body>

<script>

angular.module('myApp', [])

.config(function($provide) {

$provide.factory('calculator', function() {

var calculate = {};

calculate.add = function(num1, num2) {

return num1 + num2;

}

calculate.minus = function(num1, num2) {

return num1 - num2;

}

calculate.multiplied = function(num1, num2) {

return num1 \* num2;

}

calculate.divided = function(num1, num2) {

return num1 / num2;

}

return calculate;

});

})

.controller('myCtrl', function($scope, calculator) {

$scope.calType = '2';

$scope.cal = function() {

switch ($scope.calType) {

case '1':

$scope.result = calculator.add($scope.num1, $scope.num2);

break;

case '2':

$scope.result = calculator.minus($scope.num1, $scope.num2);

break;

case '3':

$scope.result = calculator.multiplied($scope.num1, $scope.num2);

break;

case '4':

$scope.result = calculator.divided($scope.num1, $scope.num2);

break;

}

}

});

</script>

</html>

# 第十三章 与服务器通讯

在上一章中，我们提到了$http，它其实就是一种为应用提供与服务器通讯的服务，本章中我们将详细讲解这个服务的使用已经angular与服务器通讯机制。

为了实现应用与服务器之间的交互，Angular提供了一个叫做$http的服务。它提供了一个可扩展的抽象方法列表，使得与服务器的交互更加容易。它支持HTTP、JSONP和CORS方式。它还包含了安全性支持，避免JSON格式的脆弱性和XSRF（跨站请求伪造）。它让你可以轻松的转换请求和响应数据，甚至还实现了简单的缓存。

## 13.1通过$http实现

### 13.1.1 $http的处理方法

对于AJAX应用（使用XMLHttpRequests）来说，向服务器发起请求的传统方式是：获取一个XMLHttpRequest对象的引用、发起请求、读取响应、检查状态码，最后处理服务端的响应。整个过程示例如下：

var xmlhttp = new XMLHttpRequest();

xmlhttp.onreadystatechange = function() {

if (xmlhttp.readystate == 4 && xmlhttp.status == 200) {

var response = xmlhttp.responseText;

} else if (xmlhttp.status == 400) { //或者可以是任何以4开头的状态码

//优雅地处理错误

}

};

//建立连接

xmlhttp.open("GET", "http://myserver/api", true);

//发起请求

xmlhttp.send();

对于简单、常用而且会经常重复的任务来说，这是一种很烦琐的工作。如果你想复用以上过程，你应该进行封装或者使用代码库。

AngularJS XHR API遵守一种通常被称为Promise的接口。由于XHR是异步调用的方法，所以服务端的响应会在未来某个不确定的时间点上返回（我们希望它立即能返回）。Promise接口规定了处理这种响应的方式，并且允许Promise的使用者以一种可预见的方式来使用它。

例如，我们要从服务端获取一个用户的信息，假设用来接受请求的后台接口位于/api/user路径上，此接口可以接受一个id属性作为URL参数，那么使用Angular的核心$http服务发起XHR请求的方法示例如下：

$http.get('api/user', {

params: {

id: '5'

}

})

.success(function(data, status, headers, config) {

//加载成功之后做一些事

})

.error(function(data, status, headers, config) {

//处理错误

});

如：

<!DOCTYPE html>

<html ng-app='myApp'>

<head>

<meta charset="utf-8" />

<title>Http实例</title>

</head>

<body ng-controller='ShoppingController'>

<h1>Shop!</h1>

<table>

<tr ng-repeat="item in items">

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

<td>{{item.description}}</td>

<td>{{item.price | currency}}</td>

</tr>

</table>

<script type="text/javascript" src="js/angular.min.js"></script>

<script>

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

myApp.controller('ShoppingController', function($scope, $http) {

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

$scope.items = data;

});

});

</script>

</body>

</html>

其中根据访问的路径可知，资源文件位于工程目录下的json文件夹中，具体的products.json如下：

[

{

"id": 0,

"title": "Paint pots",

"description": "Pots full of paint",

"price": 3.95

}, {

"id": 1,

"title": "Polka dots",

"description": "Dots with that polka groove",

"price": 12.95

}, {

"id": 2,

"title": "Pebbles",

"description": "Just little rocks,really",

"price": 6.95

}

]

如果你是jQuery使用者，你应该会发现，AngularJS和jQuery在对异步请求的处理方面非常类似。

### 13.1.2 $http的快捷方法

上面例子中使用的$http.get方法是AngularJS的核心服务$http所提供的众多快捷方法之一。类似地，如果你想使用AngularJS向同一个URL发送POST请求，同时带上一些POST数据，你可以像下面这样做：

var postData = {

text: 'long blob of text'

};

//下面这一行会被当成参数附加到URL后面，所以post请求最终会变成/api/user?id=5

var config = {

params: {

id: '5'

}

};

$http.post('api/user', postData, config)

.success(function(data, status, headers, config) {

//成功之后做一些事情

})

.error(function(data, status, headers, config) {

//处理错误

});

对于大多数常用的请求类型，都有类似的快捷方法，这些请求类型包括：GET、HEAD、POST、DELETE、PUT、JSONP。

### 13.1.3 进一步配置请求

虽然标准的请求方式使用起来比较简单，但是，有时候会存在可配置性不佳的缺点。如果你想要实现下面这些事情就会遇到困难：

a.给请求加上一些授权头。

b.修改对缓存的处理方式。

c.用一些特殊的方式来变换发送出去的请求，或者变换接收到的响应。

在这些情况下，你可以给请求传递一个可选的配置对象，从而对请求进行深度配置。在前面的例子中，我们使用config对象指定了一个可选的URL参数。但是那里的GET和POST方法是一些快捷方式。这种深度简化之后的方法调用示例如下：

$http(config)

下面是一个基本的伪代码模板，用来调用前面的这个方法：

$http({

method: string,

url: string,

params: object,

data: string or object,

headers: object,

transformRequest: function transform(data, headersGetter) or an array of functions,

transformResponse: function transform(data, headersGetter) or an array of functions,

cache: boolean or Cache object,

timeout: number,

withCredentials: boolean

});

GET、POST及其他快捷方法都会自动设置method参数，所以不需要手动设置。config对象会作为最后一个参数传递给$http.get和$http.post，所以，在所有的快捷方法内部都可以使用这个参数。你可以传递config对象来修改发送的请求，config对象可以设置以下键值。

method：一个字符串，表示HTTP请求的类型，例如GET或者POST。

url：URL字符串，表示请求的绝对或者相对资源路径。

params：一个键和值都是字符串的对象（确切来说是一个map），表示需要转换成URL参数的键和值。例如：

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

将会被转换成

?key1=**value**&key2=value2

并会被附加到URL后面。如果我们使用js对象（而不是字符串或者数值）作为map中的值，那么这个js对象会被转换成JSON字符串。

data：一个字符串或者对象，它会被当作请求数据发送。

timeout：在请求超时之前需要等待的毫秒数。

### 13.1.4设置HTTP头

AngularJS带有一些默认的请求头，Angular发出的所有请求上都会带有这些默认的请求头信息。默认请求头包括以下两个：

1.Accept:appliction/json,text/pain,/

2.X-Requested-With: XMLHttpRequest

如果想设置特殊的请求头，可以用如下两种方法实现。

第一种方法，如果你想把请求头设置到每一个发送出去的请求上，那么你可以把需要使用的特殊请求头设置成AngularJS的默认值。这些值可以通过$httpProvider.defaults.headers配置对象来设置，通常会在应用的配置部分来做这件事情。所以，如果你想对所有的GET请求使用“DO NOT TRACK”头，同时对所有请求删除Requested-With头，可以简单地操作如下：

angular.module('MyApp', [])

.config(function($httpProvider) {

//删除AngularJS默认的X-Request-With头

delete $httpProvider.default.headers.common['X-Requested-With'];

//为所有GET请求设置DO NOT TRACK

$httpProvider.default.headers.get['DNT'] = '1';

});

如果你只想对某些特定的请求设置请求头，但不把它们作为默认值，那么你可以把头信息作为配置对象的一部分传递给$http服务。同样的，自定义头信息也可以作为第二个参数的一部分传递给GET请求，第二个参数还可以同时接受URL参数。

$http.get('api/user', {

//设置Authorization(授权)头。在真实的应用中，你需要到一个服务里面去获取auth令牌

headers: {

'Authorization': 'Basic Qzsda231231'

},

params: {

id: 5

}

}).success(function() { //处理成功的情况 });

### 13.1.5 缓存响应

对于HTTP GET请求，AngularJS提供了一个开箱即用的简单缓存机制。默认情况下它对所有请求类型都不可用，为了启用缓存，你需要做一些配置：

$http.get('http://server/myapi', {

cache: true

}).success(function() { //处理成功的情况});

这样就可以启用缓存，然后AngularJS将会缓存来自服务器的响应。下一次向同一个URL发送请求的时候，AngularJS将会返回缓存中的响应内容。缓存也是智能的，所以即使你向同一个URL发送多次模拟的请求，缓存也只会向服务器发送一个请求，而且在收到服务端的响应之后，响应的内容会被分发给所有请求。

但是，这样做有些不太实用，因为用户会先看到缓存的旧结果，然后看到新的结果突然出现。例如，当用户即将点击一条数据时，它可能会突然发生变化。

注意，从本质上来说，响应（即使是从缓存中读取的）依然是异步的。换句话说，在第一次发出请求的时候，你应该使用处理异步请求的方式来编码。

### 13.1.6转换请求和响应

对于所有通过$http服务发出的请求和收到的响应来说，AngularJS都会进行一些基本的转换，包括如下内容。

1.转换请求

如果请求的配置对象属性中包含JS对象，那么就把这个对象序列化成JSON格式。

2.转换响应

如果检测到了XSRF（Cross Site Request Forgery的缩写，意为跨站请求伪造，这是跨站脚本攻击的一种方式）前缀，则直接丢弃。如果检测到了JSON响应，则使用JSON解析器对它进行反序列化。

如果你不需要其中的某些转换，或者想自已进行转换，可以在配置项里面传入自已的函数。这些函数会获取HTTP的request/response体以及协议头信息，然后输出序列化、修改之后的版本。可以使用transformLRequest和transformResponse作为key来配置这些转换函数，而这两个函数在模块的config函数中是用$httpProvider服务来配置的。

我们什么时候需要使用这些东西呢？假设我们有一个服务，它更适合用jQuery的方式来操作。POST数据使用key1=val1&key2=val2(也就是字符串)形式来代替{key1:val1, key2:val2}JSON格式。我们可以在每个请求中来进行这种转换，也可以添加一个独立transformRequest调用，对于当前这个例子来说，我们打算添加一个通用的transformRequest，这样所有发出的请求都会进行这种从JSON到字符串的转换。下面就是实现方式：

var module = angular.module('myApp');

module.config(function($httpProvider) {

$httpProvider.defaults.transformRequest = function(data) {

//使用jQuery的param方法把JSON数据转换成字符串形式

return $.param(data);

};

});

## 13.2采用RESTful资源实现

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

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

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

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

### 13.2.1 什么时候及如何使用Angular Resources组件

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

a.给地址/user/123/card发送一个GET请求，将会得到用户123的信用卡列表.

b.给地址/user/123/card/15发送一个GET请求,将会得到用户123本人的ID为15的信用卡信息

c.给地址/user/123/card发送一个在POST请求数据部分包含信用卡信息的POST请求,将会为用户123新创建一个信用卡

d.给地址/user/123/card/15发送一个在POST请求数据部分包含信用卡信息的POST请求,将会更新用户123的ID为15的信用卡的信息

e.给地址/user/123/card/15一个方法为DELETE类型的请求,将会删除掉用户123的ID为15的信用卡的数据.

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

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

（1）在你的HTML文件里面引用angular-resource.js的实际地址

（2）在你的模块依赖里面声明对它的依赖(例如,angular.module('myModule',['ngResource'])).

（3）在需要的地方，注入$resource这个依赖项.

如：

<!DOCTYPE html>

<html ng-app='myApp'>

<head>

<meta charset="utf-8" />

<title>Http实例</title>

</head>

<body ng-controller='ShoppingController'>

<h1>Shop!</h1>

<table>

<tr ng-repeat="item in items">

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

<td>{{item.description}}</td>

<td>{{item.price | currency}}</td>

</tr>

</table>

<script type="text/javascript" src="js/angular.min.js"></script>

<script type="text/javascript" src="js/angular-resource.js"></script>

<script>

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

myApp.controller('ShoppingController', function($scope, $http, Products) {

$scope.items = Products.query();

})

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

return $resource('json/products.json', {}, {

query: {

method: 'GET',

isArray: true

}

});

});

</script>

</body>

</html>

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

这儿是上述需求一个可能的实现：

myAppModule.factory('CreditCard', ['$http',

function($http) {

var baseUrl = '/user/123/card';

return {

get: function(cardId) {

return $http.get(baseUrl + '/' + cardId);

},

save: function(card) {

var url = card.id ? baseUrl + '/' + card.id : baseUrl;

return $http.post(url, card);

},

query: function() {

return $http.get(baseUrl);

},

charge: function(card) {

return $http.post(baseUrl + '/' + card.id, card, {

params: {

charge: true

}

});

}

};

}

]);

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

myAppModule.factory('CreditCard', ['$resource',

function($resource) {

return $resource('/user/:userId/card/:cardId', {

userId: 123,

cardId: '@id'

}, {

charge: {

method: 'POST',

params: {

charge: true

},

isArray: false

});

}]);

做到现在，你就可以任何时候从Angular注入器里面请求一个CreditCard依赖，你就会得到一个Angular资源,默认情况下，这个资源会提供一些初始的可用方法.

### 13.2.2 resource资源的声明

声明你自己的$resource非常简单，只要调用注入的$resource函数,并给他传入正确的参数即可。

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

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

函数的第二个参数则负责提供所有请求的默认参数变量信息.在这个案例中，我们给userId参数传递一个常量:123,cardId参数则更有意思,我们给cardId参数传递了一个"@id"字符串.这意味着如果我们使用一个从服务器返回的对象而且我们可以调用这个对象的任何方法(比如$save),那么这个对象的id属性将会被取出来赋给cardId字段.

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

### 13.2.3 定制方法

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

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

### 13.2.4 不要使用回调函数机制!(除非你真的需要它们)

第三个需要注意的事情是资源调用的返回类型.让我们再次关注一下CreditCard.query()这个函数.你将会看到不是在回调函数中给cards赋值,而是直接把它赋给card变量.在异步服务器请求的情况下唉，这样的代码是如何运作的哪?

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

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

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

简化的服务器端操作

无论你使用资源简化函数机制还是回调函数，关于返回对象都有几点问题需要我们注意.

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

$q和预期值(Promise)

目前为止，我们已经看到了AngulrJS是如何实现它的异步延迟API机制.

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

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

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

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

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

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

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

另外,这种情况对错误处理也有很大影响.错误处理的最好方法是什么?在每次都做错误处理?那代码结构就会非常乱.

为了解决上面这些问题,预期值建议(Promise proposal)机制提供了一个then函数的概念,这个函数会在响应成功返回的时候调用相关的函数去执行,另一方面，当产生错误的时候也会干相同的事，这样整个代码就有嵌套结构变为链式结构.所以之前那个例子用预期值API机制(至少在AngularJS中已经被实现的)改造一下,代码结构会平整许多： var deferred = $q.defer(); var fetchUser = function() { // After async calls, call deferred.resolve with the response value deferred.resolve(user);

// In case of error, calldeferred.reject(‘Reason for failure’);}// Similarly, fetchUserPermissions and fetchUserListData are handleddeferred.promise.then(fetchUser).then(fetchUserPermissions).then(fetchUserListData).then(function(list) {// Do something with the list of data}, function(errorReason) {// Handle error in any of the steps here in a single stop});

那个完全的横椎体代码一下子被优雅地平整了,而且提供了链式的作用域,以及一个单点的错误处理.你在你自己的应用中处理异步请求回调时也可以用相同的代码，只要调用Angular的$q服务.这种机制可以帮我做一些很酷的事情：比如响应拦截.

响应拦截处理

我们的讲解已经覆盖了怎样调用服务器端服务、怎样处理响应、怎样把响应优雅地抽象化封装、怎样处理异步回调.但是在真实世界的Web应用中，你最终还不得不对每个服务器端请求调用做一些通用的处理操作，比如错误处理、权限认证、以及其它考虑到安全问题的处理操作，比如对响应数据做裁剪处理(译注:有的Ajax响应为了安全需要，会添加一定约定好的噪声数据).

有着现在已经对$q API的深入理解,我们目前就可以利用响应拦截器机制来做上面所有提出过的功能.响应拦截器(正如其名)可以在响应数据被应用使用之前拦截他它,并且对它做数据转换处理,比如错误处理以及其它任何处理，包括厨房洗碗槽.(估计是指数据清洗)

让我们看一个代码例子，这个例子中的代码拦截响应，并对响应数据做了轻微的数据转换.

// register the interceptor as a servicemyModule.factory('myInterceptor', function($q, notifyService, errorLog) {return function(promise) {return promise.then(function(response) {// Do nothingreturn response;}, function(response) {// My notify service updates the UI with the error messagenotifyService(response);// Also log it in the console for debug purposeserrorLog(response);return $q.reject(response);});}});// Ensure that the interceptor we created is part of the interceptor chain$httpProvider.responseInterceptors.push('myInterceptor');

安全方面的考虑

目前我们开发Web应用的时候，安全是一个非常重要的关注点，在我们的考虑维度直中，它必须作为首位被考虑.AngularJS给我们提供了一些帮助，同时也带来了两个安全攻击的角度，下面这一节我们将会讲解这些内容.

JSON的安全脆弱性

当我们对服务器发送一个请求JSON数组数据的GET请求时(特别是当这些数据是敏感数据且需要登录验证或读取授权时),就会有一个不易察觉的JSON[安全漏洞](http://www.haodaima.net/tag/%E5%AE%89%E5%85%A8%E6%BC%8F%E6%B4%9E)被暴露出来.

当我们使用一个恶意站点时，站点可能会用<script>标签发起同样的请求而得到相同的信息.因为你仍旧是登录状态，恶意站点利用了你的验证信息而请求了JSON数据，并且得到了它.

你或许惊奇是如何做到的，因为信息仍旧在你客户端，服务器也得不到这个数组信息的引用.并且通常作为请求脚本返回响应JSO对象会导致一个执行错误，虽然数组是个列外.

但是漏洞真正的切入点是：在JavaScript里，你是可以对内建对象做重定义的.在这个漏洞里面，数组的构造函数可以被重定义，通过这种重定义，恶意站点脚本就可以得到对响应数据的引用，然后就可以把响应数据发回它自己的服务器喽.

有两种方法可以防止这个漏洞:一是通常要确保敏感数据信息只作为POST请求的响应被返回，二是返回一个不合法的JSON表达式,然后客户端用约定好的逻辑把不合法数据转换为可用的真实数据.

AngulaJS中你可以两种方法都用来阻止这个漏洞.在你的应用中，你可以而且应该选择敏感JSON信息只通过POST请求来获取.

进一步，你可以在服务器端给JSON响应数据配置一个前缀字符串:

")]}`,\n"

那么一个正常JSON响应比如：

['one','two']

通过前缀字符串设置，这个JSON响应就会变为

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

AngularJS将会自动的把前缀字符串过滤掉,然后仅仅处理真实JSON数据.

跨站请求伪造(XSRF)

跨站请求伪造攻击主要有以下特征：

它们影响的站点通常依赖于授权或者用户认证.它们往往利用漏洞站点保存登录或者授权信息这个事实.它们发起以假乱真的HTTP或者XMLHTTPRequest请求来制造副作用，这种副作用通常是有害的.

考虑依稀下面这个跨站请求伪造攻击的案例:

用户A登录进他的银行帐号(http:\/\/www.examplebank.com/)用户B意识到这点，然后诱导用户A访问用户B的个人主页

主页上有一个特殊手工生成的图片连接地址，这个图片的的指向地址将会导致一次跨站请求伪造攻击,比如如下代码：

<img src="http://www.examplebank.com/xfer?from=UserA&amount=10000&to=UserB" rel="[nofollow](http://www.haodaima.net/tag/nofollow)"/>

如果用户A的银行站点把授权信息保存在cookie里，且Cookie还没过期.当用户A打开用户B的站点时,就会导致非授权的用户A给用户B转账行为.

那么AngularJS是怎么帮助你防止这种事情发生?它提供一种双步机制来防止跨站请求伪造攻击.

在客户端，当发起XHR异步请求时,$http服务会从一个叫XSRF-TOKEN的cookie中读取令牌值，然后把它设置成X-XSRF-TOKEN头信息的值,因为只有你自己域的请求才能读取和设置这个令牌,你可以保证XHR请求只来自你自己的域.

同时，服务器端代码也需要一点轻微的修改，以便于你收到你的第一个HTTP GET请求时就设置一个可读取的对话Cookie，这个对话Cookie键叫XSRF-TOKEN。后续客户端发往服务器的请求就可以通过对比请求头信息的令牌值和之前第一个请求设置的Cookie令牌值来达到验证的目的.当然,令牌必须是一个用户一个唯一的令牌值.这个令牌值必须在服务器端验证(以防止恶意脚本捏造假令牌).

# 第十四章 promise

通过十三章的学习我们知道angularJS应用与服务器交互其实使用的就是promise模式，那promise是一种怎样的设计模式呢？在这一章中，我们将给大家详细解答。

## \* 卡通方式趣解 AngularJS 中的 promise

**一天早晨，爹对儿子说：“宝儿，出去看看天气如何！”**

每个星期天的早晨，爹都叫小宝拿着超级望远镜去家附近最高的山头上看看天气走势如何，小宝说没问题，我们可以认为小宝在离开家的时候给了他爹一个promise。

这时候，他爹就想了，如果明天艳阳高照，他就准备去钓鱼，如果天实在不行，就作罢，如果小宝对预报明天的天气也没底，他就在家宅一天哪也不去。

大概过了半小时，小宝回来了。每周的结果不尽相同：

**A计划 ：天气晴朗**

小宝不辱使命，说外面阳光明媚，万里无云，这个promise was resolved（小宝信守承诺），爹就可以收拾行装，钓鱼去鸟。



**B计划： 小宝日观天象，阴转小雨的节奏**

小宝依然不辱使命，但是天公不作美，promise was resolved，但是孩儿他爹觉得还是搁家呆着吧。



**C计划：天象诡谲，小宝无法做出天气走势判断**

小宝败兴而归，云雾重重，遮蔽了视线，不敢妄言天气走势，小宝走的时候立下承诺说要给他爹预报天气，但是没有成功，我们说promise was rejected！孩儿他爹决定小心驶得万年船，还是在家吧。



**上述种种，用代码写出来是什么样子呢？**

我们可以把孩儿他爹看成controller，小宝就是service。

整理逻辑：孩儿他爹让小宝去看天气，小宝不能立刻告诉他，但是孩儿他爹在等待结果的这段时间里可以抽抽烟，喝喝茶啥的，因为小宝承诺会把天气情况搞清楚。等小宝把天气预报带回来，他就可以决定下一步干啥。各位看官注意了：小宝登高望远看天气的时候并没有影响他爹干别的事情，这就是promise的妙处所在。

Angular里有个then()函数，我们可以决定孩儿他爹到底是用哪个计划(A，B，C)，then()接收两个functions作为参数，第一个在promise is resolved的时候执行，另一个在promise is rejected的时候执行

**Controller: FatherCtrl**

孩儿他爹掌控局面的代码如下：

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

myApp.controller('fatherCtrl', function($scope, SonService) {

$scope.makePromiseWithSon = function() {

// This service's function returns a promise, but we'll deal with that shortly

SonService.getWeather()

// then() called when son gets back

.then(function(data) {

// promise fulfilled

if (data.forecast === 'good') {

prepareFishingTrip();

} else {

prepareSundayRoastDinner();

}

}, function(error) {

// promise rejected, could log the error with: console.log('error', error);

prepareSundayRoastDinner();

});

}

//other function of fatherCtrl

});

|  |  |
| --- | --- |
|  |  |

**Service: SonService**

小宝的作用就是充当了一个service，他爬上山头看天象，有点类似调用weather API，而且还是异步调用，他得到的结果可能是个变量，也有可能出现异常情况(比如,返回500—>大雾弥漫)。

从”Fishing Weather API”返回一个promise，如果it was resolved，就格式化成{“forecase”:”good”}。

myApp.factory('SonService', function($http, $q) {

return {

getWeather: function() {

// the $http API is based on the deferred/promise APIs exposed by the $q service

// so it returns a promise for us by default

return $http.get('http://fishing-weather-api.com/sunday/afternoon')

.then(function(response) {

if (typeof response.data === 'object') {

return response.data;

} else {

// invalid response

return $q.reject(response.data);

}

}, function(response) {

// something went wrong

return $q.reject(response.data);

});

}

};

});

|  |  |
| --- | --- |
|  |  |

**总结**

这个比喻向我们展示了异步的实质，孩儿他爹不会倚门等待儿子的归来，这段时间他完全可以自由活动。 孩儿他爹到底用哪个计划取决于(天气好/坏，没有成功预报)，小宝在临走的时候给他爹一个promise，就等他回来的时候决定是resolve还是reject。

## 14.1 promise是什么

Promise是Js异步编程模式的一种模式，以同步操作的流程形式来操作异步事件，避免了层层嵌套，可以链式操作异步事件。

我们知道，在编写javascript异步代码时，callback是最最简单的机制，可是用这种机制的话必须牺牲控制流、异常处理和函数语义化为代价，甚至会让我们掉进出现callback大坑，而promise解决了这个问题。

ES6中Promise、angularJS内置的AngularJS内置Q，以及when采用的都是Promises/A规范，如下：

每个任务都有三种状态：未完成(pending)、完成(fulfilled)、失败(rejected)。

* pending状态：可以过渡到履行或拒绝状态。
* fulfilled状态：不能变为其他任何状态，而且状态不能改变，必须有value值。
* rejected状态：不能变为其他任何状态，而且状态不能改变，必须有reason。

状态的转移是一次性的，状态一旦变成fulfilled（已完成）或者failed（失败/拒绝），就不能再变了。

## 14.2 promise的基本用法

**1. 在service等中注入$q**

**2. 新建一个deffered实例**

var deferred = $q.defer();

A. 如果异步操作成功，则用resolve方法将Promise对象的状态变为“成功”（即从pending变为resolved），如：

deferred.resolve(result.value);

B. 如果异步操作失败，则用reject方法将状态变为“失败”（即从pending变为rejected），如：

deferred.reject("Can't update user");

C. 最后返回 deferred.promise，我们就可以链式调用then方法。如：

return deferred.promise;

### 14.2.1 deffered 对象的方法

resolve(value)：在声明resolve()处，表明promise对象由pending状态转变为resolve。

reject(reason)：在声明resolve()处，表明promise对象由pending状态转变为rejected。

notify(value) ：在声明notify()处，表明promise对象unfulfilled状态，在resolve或reject之前可以被多次调用。

### 14.2.2 deffered 对象属性

promise ：最后返回的是一个新的deferred对象 promise 属性，而不是原来的deferred对象。这个新的Promise对象只能观察原来Promise对象的状态，而无法修改deferred对象的内在状态可以防止任务状态被外部修改。

当创建 deferred 实例时会创建一个新的 promise 对象,并可以通过 deferred.promise 得到该引用。

promise 对象的目的是在 deferred 任务完成时,允许感兴趣的部分取得其执行结果。

### 14.2.3 promise 对象的方法

then(errorHandler, fulfilledHandler, progressHandler) —— then方法用来监听一个Promise的不同状态。errorHandler监听failed状态，fulfilledHandler监听fulfilled状态，progressHandler监听unfulfilled（未完成）状态。此外,notify 回调可能被调用 0到多次，提供一个进度指示在解决或拒绝（resolve和rejected）之前。

catch(errorCallback) —— promise.then(null, errorCallback) 的快捷方式

finally(callback) ——让你可以观察到一个 promise 是被执行还是被拒绝, 但这样做不用修改最后的 value值。 这可以用来做一些释放资源或者清理无用对象的工作,不管promise 被拒绝还是解决。

通过then()方法可以实现promise链式调用。

## \* 英文参考文档（Promise in AngularJS）

### \*.1 What's promise

Angular’s event system provides a lot of power to our Angular apps. One of the most powerful features that it enables is automatic resolution of promises.

**Promises** are a method of resolving a value or not in an asynchronous manner. *Promises are objects that represent the return value or a thrown exception that a function may eventually provide*.

Promises are incredibly useful in dealing with remote objects and we can think of them as a proxy for them.

Promises are first-class objects and carry with them a few **guarantees**:

Only one resolve or reject will ever be called

– resolve will be called with a single fulfillment value

– reject will only be called with a single rejection reason

If the promise has been resolved or rejected, any handlers depending upon them will still be called

Handlers will always be called asynchronously

Additionally, we can also chain promises and allow the code to process as it will normally would run. Exceptions from one promise bubble up through the entire promise chain.

*They are always asynchronous,* so we can use them in the flow of our code without worry that they will block the rest of the app.

### \*.2 Promise in AngularJS

Angular’s event-loop gives angular the unique ability to resolve promises in it’s $rootScope. $evalAsync stage (see under the hood for more detail on the run loop). *The promises will sit inert until the $digest run loop finishes*.

*bind promise and view directly*

This allows for Angular to turn the results of a promise into the view without any extra work. It enables us to assign the result of an XHR call directly to a property on a $scope object and think nothing of it. For instance, we might have a list of friends in a view, like so:

<**ul**>

<**li** ng-repeat="friend in friends">

{{ friend.**name** }}

</**li**>

</**ul**>

If we have a service that returns a promise , we can simply place the promise in the view and expect that Angular will resolve it for us:

angular.module('myApp', [])

.controller('DashboardController',['$scope', 'UserService', **function**($scope, UserService) {

// UserService's getFriends() method

// returns a promise

$scope.friends = User.getFriends(123);

}]);

When the asynchronous call to getFriends returns, the $scope.friends value will automatically update the view.

### \*.3 How to create a promise

In order to create a promise in Angular, we’ll use the built-in $q service. The $q service provides a few methods in it’s deferred API.

**1. inject $q service**

First, we’ll need to inject the $q service into our object where we want to use it.

angular.module('myApp', [])

.factory('UserService', ['$q', **function**($q) {

// Now we have access to the $q library

}]);

**2. $q.defer()**

To created a deferred object, we’ll call the method defer():

**var** deferred = $q.defer();

The deferred object exposes *three methods* and the *single promise property* that can be used in dealing with the promise.

**resolve(value)**

The resolve function will resolve the deferred promise with the value.

deferred.resolve({name: "Ari", username: "@auser"});

**reject(reason)**

This will reject the deferred promise with a reason. This is equivalent to resolving a promise with a rejection

deferred.reject("Can't update user");

// Equivalent to

deferred.resolve($q.reject("Can't update user"));

**notify(value)**

This will respond with the status of a promises execution.

TODO: Add notify example

**promise property**

We can get access to the promise as a property on the deferred object:

deferred.promise

A full example of creating a function that responds with a promise might look similar to the following method on the UserService as mentioned above.

angular.module('myApp', [])

.factory('UserService', ['$q', '$http', **function**($q) {

**var** result = {};

result.getFriends = **function**(id) {

**var** deferred = $q.defer();

// Get friends from a remote server

$http.get('/user/' + id + '/friends')

.success(**function**(data) {

deferred.resolve(data.friends);

})

.error(**function**(reason) {

deferred.reject(reason);

});

**return** deferred.promise;

}

**return** result;

}]);

### \*.4 interact with promise

Now we can use the promise API to interact with the getFriends() promise.

In the case of the above service, we can interact with the promise in two different ways.

**then(successFn, errFn, notifyFn)**

Regardless of the success or failure of the promise, then it will call either the successFn or the errFn asynchronously as soon as the result is available. The call backs are always called with a single argument: the result or the rejection reason.

The notifyFn callback may be called zero or more times to provide a progress status indication before the promise is resolved or rejected.

The then() method always returns a new promise which is either resolved or rejected through the return value of the successFn or the errFn. It also notifies through the notifyFn.

**catch(errFn)**

This is simply a helper function that allows for us to replace the err callback with *.catch(function(reason){}):*

$http.get('/user/' + id + '/friends')

.**catch**(**function**(reason) {

   deferred.reject(reason);

 });

**finally(callback)**

This allows you to observe the fulfillment or rejection of a promise, but without modifying the result value. This is useful for when we need to release a resource or run some clean-up regardless of the success/error of the promise.

*We cannot call this directly* due to finally being a reserved word in IE javascript. To use finally, we have to call it like:

promise['finally'](**function**() {});

Angular’s $q deferred objects are chainable in that even then returns a promise. As soon as the promise is resolved, the promise returned by then is resolved or rejected.

These promise chains are how Angular can support $http’s interceptors.

The $q service is similar to the original Kris Kowal’s Q library:

$q is integrated with the angular $rootScope model, so resolutions and rejections happen quickly inside the angular

$q promises are integrated with angular’s templating engine which means that any promises that are found in the views will be resolved/rejected in the view

$q is tiny and doesn’t contain the full functionality of the Q library

*$q library*

The $q library comes with several different useful methods.

**all(promises)**

If we have multiple promises that we want to combine into a single promise, then we can use the $q.all(promises) method to combine them all into a single promise. This method takes a single argument:

promises (array or object of promises)

Promises as an array or hash of promises

The all() method returns a single promise that will be resolved with an array or hash of values. Each value will correspond to the promises at the same index/key in the promises hash. If any of the promises are resolved with a rejection, then the resulting promise will be rejected as well.

**defer()**

The defer() method creates a deferred object. It takes no parameters. It returns a new instance of a single deferred object.

**reject(reason)**

This will create a promise that is resolved as rejected with a specific reason. *This is specifically designed to give us access to forwarding rejection in a chain of promises.*

This is akin to throw in javascript. In the same sense that we can catch an exception in javascript and we can forward the rejection, we’ll need to rethrow the error. We can do this with $q.reject(reason).

This method takes a single parameter:

reason (constant, string, exception, object)

The reasons for the rejection.

This reject() method returns a promise that has already been resolved as rejected with the reason.

**when(value)**

The when() function wraps an object that might be a value then-able promise into a $q promise. *This allows for us to deal with an object that may or may not be a promise.*

The when() function takes a single parameter:

value

This is the value or a promise

The when() function returns a promise that can be then used like any other promise.

## 14.3 promise有与否的区别

在下面的例子中，将创建两个服务，其中一个没使用promise，而另一个使用了promise.没有使用promise时将会在请求未返回时就将结果提前打印出来，此时只能打印出变量的初始值；而使用了promise的那个将会在请求返回后才把数据打印出来，因此打印出来的结果将是处理后的数据。

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<title>promise</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-controller="myCtrl">

<div>value: {{value}}</div>

<div>value1: {{value1}}</div>

<script>

angular.module('myApp', [])

.controller('myCtrl', function($scope, noPromise, hasPromise) {

$scope.value = noPromise.val();

var promise = hasPromise.val(); // 同步调用，获得承诺接口

promise.then(function(data) { // 调用承诺API获取数据 .resolve

$scope.value1 = data;

}, function(data) { // 处理错误 .reject

console.log('error');

});

})

.factory('noPromise', function($http) {

var result = {};

result.value = '初始值';

result.val = function() {

$http.get('json/products.json')

.success(function(data, status, headers, config) {

result.value = data.value;

//return result.value;//若在此处写返回则打印不出数据，个人认为是线程问题？？？？？

});

return result.value;//在此处写返回，打印出来的结果是“初始值”

}

return result;

})

.factory('hasPromise', function($http, $q) {

var result = {};

result.value = '初始值';

result.val = function() {

var deferred = $q.defer();

$http.get('json/products.json')

.success(function(data, status, headers, config) {

result.value = data.value;

deferred.resolve(result.value);

});

return deferred.promise;

}

return result;

});

</script>

</body>

</html>

{

"id": 0,

"title": "Paint pots",

"description": "Pots full of paint",

"price": 3.95,

"value": "已处理"

}

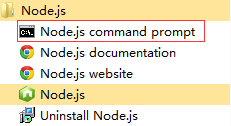
# 第十五章 测试

## 15.1 测试环境的搭建

Jasmine是一个Javascript的测试工具，在Karma上运行Jasmine可完成Javascript的自动化测试、生成覆盖率报告等。

Karma是运行在Node.js之上的，因此我们首先要安装Node.js。至于Node.js如何安装这里不讲。

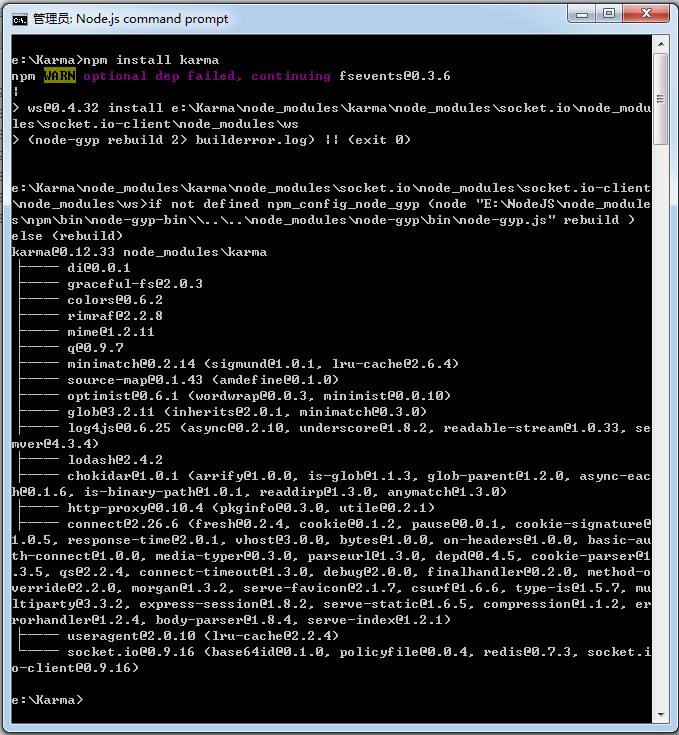
1. 运行Node.js的命令行程序：Node.js command prompt：



2. 安装Karma（我的安装路径为：E:\Karma）

运行命令：npm install –g karma

安装结束后结果如下：

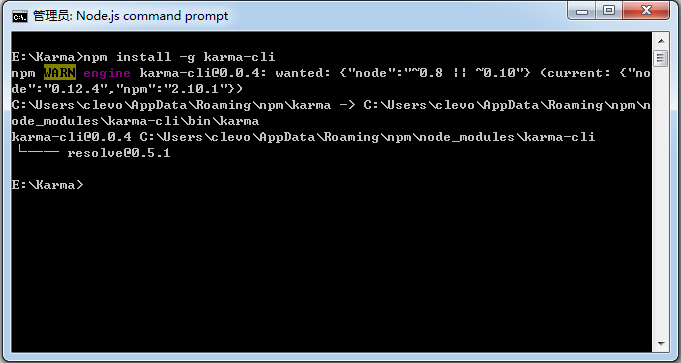


3. 安装karma-cli

karma-cli用来简化karma的调用，安装命令如下，其中-g表示全局参数，这样今后可以非常方便的使用karma了：

运行命令：npm install -g karma-cli

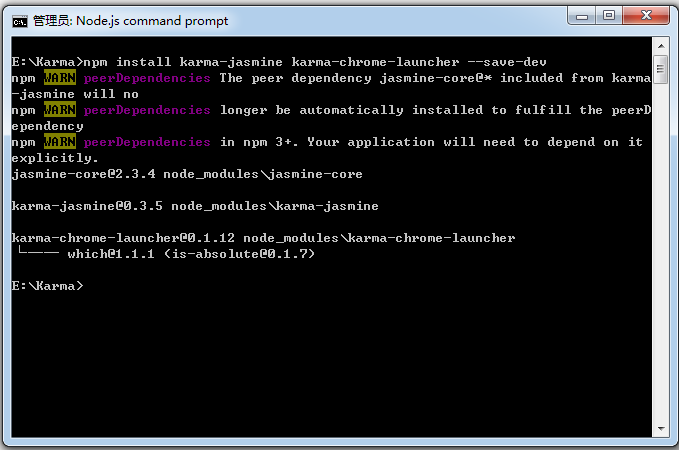
安装结果如下：



4. 安装karma-jasmine/karma-chrome-launcher插件

运行命令：npm install karma-jasmine karma-chrome-launcher --save-dev

安装结果如下：

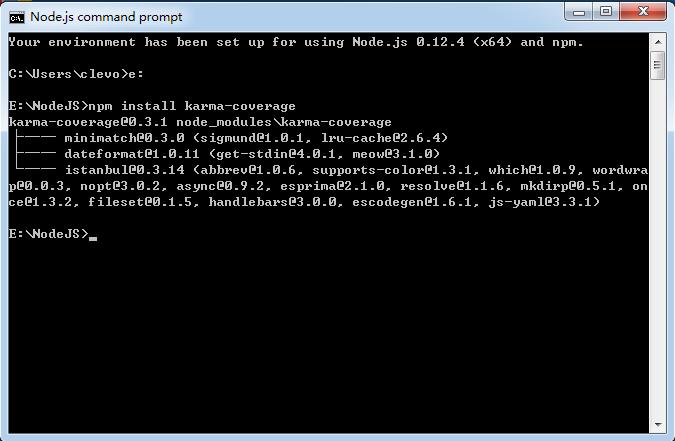


5. 安装karma-coverage插件

karma-coverage插件主要用于查看测试代码的覆盖率。（我当时好像是在NodeJs的安装目录下安装）

运行命令：npm install -g karma-coverage

安装结果：

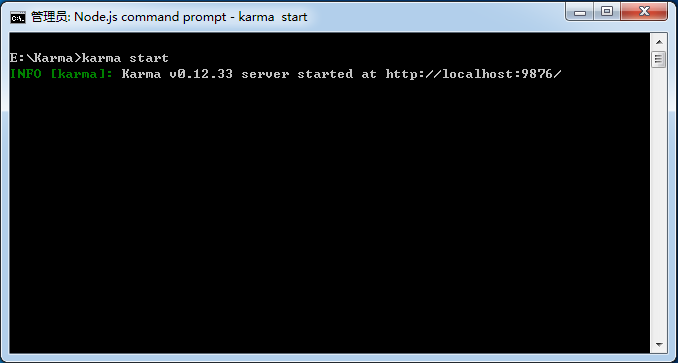


6. 检验：

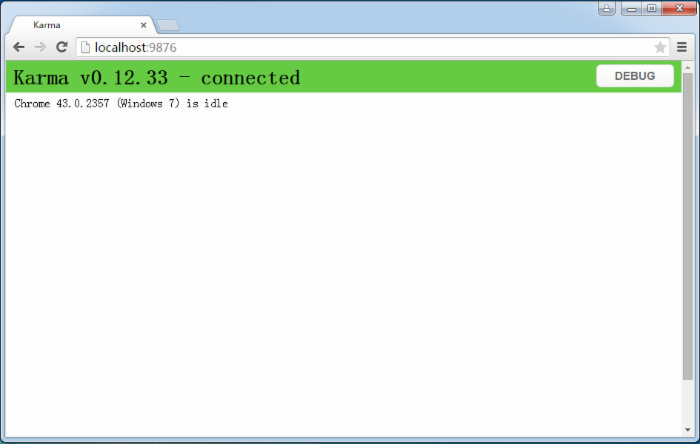
转到Karma的安装目录，运行以下命令：

karma start

执行结果：



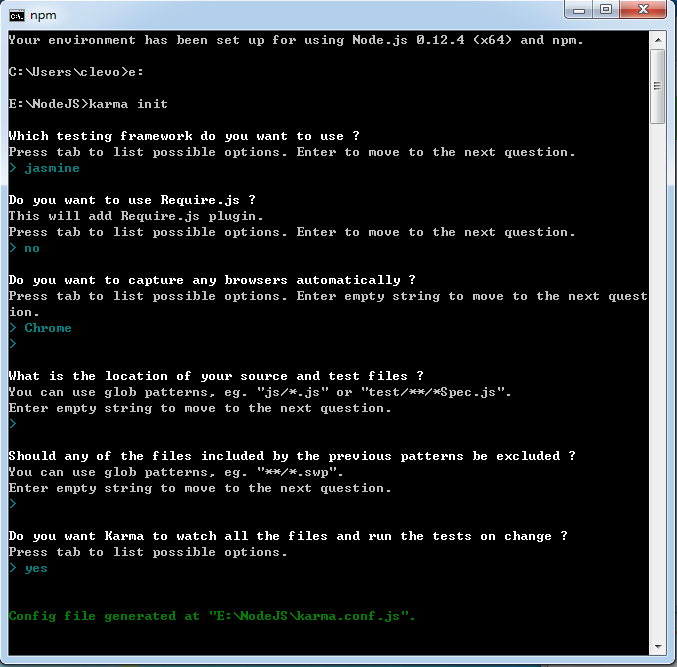
然后打开谷歌浏览器，输入localhost:9876，出现以下结果则说明安装成功：



7. 使用Karma进行Javascript测试

1）新建一个工程文件夹，并将被测文件及测试脚本添加到该文件夹；

2）使用命令跳转到工程目录，运行命令karma init生成karma.conf.js，具体配置我目前不是很懂，可参照下图：



3）修改karma.conf.js文件，配置files：

files: [

'svc.js',

'test.js'

],

若要查看测试代码的覆盖率，还应进行如下修改：

preprocessors: {

'svc.js':'coverage'

},

reporters: ['progress','coverage'],

coverageReporter:{

type:'html',

dir:'coverage/'

},

4）开启Karma，对代码进行测试：

运行命令：karma start karma.conf.js

此时谷歌浏览器会被自动打开，测试结果可点击debug查看（在consolelog中），生成的代码覆盖率报告在工程文件夹的coverage目录下。

## 15.2基于Karma和Jasmine的AngularJS单元测试

下面我们使用一个例子来讲一下基于Karma和Jasmine的AngularJS测试流程。

1）下我们要测试的文件：

home.js：

'use strict';

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

app.factory('UserFactory', function($resource) {

return $resource('Users/users.json');

});

app.controller('MainCtrl', function($scope, UserFactory) {

$scope.text = 'Hello World!';

$scope.users = UserFactory.get();

});

从以上代码中我们可以看出该应用使用了AngularJS框架，而且在创建模块时还注入了ngResource，因此在测试的时候好注意配置后所引用的库文件。

2）创建测试脚本

为了完成这个测试，我们需要模拟我们的 Application 模块以及 $scope 变量。我们会在Jasmine的 beforeEach 方法中做这个工作，这样的话我们在每个测试用例开始时可以有一个全新的（干净的）controler和scope对象。

具体脚本如下（home.test.js）：

'use strict';

describe('MainCtrl', function() {

var scope; //我们会在测试中使用这个scope

//模拟我们的Application模块并注入我们自己的依赖

beforeEach(angular.mock.module('Application'));

//模拟Controller，并且包含 $rootScope 和 $controller

beforeEach(angular.mock.inject(function($rootScope, $controller) {

//创建一个空的 scope

scope = $rootScope.$new();

//声明 Controller并且注入已创建的空的 scope

$controller('MainCtrl', {

$scope: scope

});

}));

// 测试从这里开始

it('should have variable text = "Hello World!"', function() {

expect(scope.text).toBe('Hello World!');

});

});

以上测试代码只测试了text的赋值语句；如果要测试users赋值语句，由于该语句需要发送$resource请求，因此我们需要对以上代码进行适当的修改。

要完成这个请求，我们需要使用到 $httpBackend，它一个模拟版本的Angular $http。我们会创建另一个叫做$httpBackend的变量，在第二个beforEach块中，注入 \_$httpBackend\_ 并将新创建的变量指向 \_$httpBackend\_。接下来我们会告诉 $httpBackend 如何对请求做出响应。

$httpBackend = \_$httpBackend\_;

$httpBackend.when('GET', 'Users/users.json').respond([{id: 1, name: 'Bob'}, {id:2, name: 'Jane'}]);

我们的测试：

it('should fetch list of users', function(){

$httpBackend.flush();

expect(scope.users.length).toBe(2);

expect(scope.users[0].name).toBe('Bob');

});

综合后home.test.js如下：

'use strict';

describe('MainCtrl', function() {

var scope, $httpBackend; //we'll use these in our tests

//mock Application to allow us to inject our own dependencies

beforeEach(angular.mock.module('Application'));

//mock the controller for the same reason and include $rootScope and $controller

beforeEach(angular.mock.inject(function($rootScope, $controller, \_$httpBackend\_) {

$httpBackend = \_$httpBackend\_;

$httpBackend.when('GET', 'Users/users.json').respond([{

id: 1,

name: 'Bob'

}, {

id: 2,

name: 'Jane'

}]);

//create an empty scope

scope = $rootScope.$new();

//declare the controller and inject our empty scope

$controller('MainCtrl', {

$scope: scope

});

}));

// tests start here

it('should have variable text = "Hello World!"', function() {

expect(scope.text).toBe('Hello World!');

});

it('should fetch list of users', function() {

$httpBackend.flush();

expect(scope.users.length).toBe(2);

expect(scope.users[0].name).toBe('Bob');

});

});

（这里有个问题尚未解决）

3）修改karma.conf.js，配置测试文件

从上面的文字中我们也可以知道，与本次测试有关的文件主要有：

* angular.js —— 应用的框架
* angular-resource.js —— 应用模块所注入的依赖
* home.js —— 被测试文件
* home.test.js —— 测试脚本

另外，我们还需要一个专门用于测试的一个脚本就是

* angular-mocks.js —— 用于测试的脚本

因此，我们需要在karma.conf.js文件中配置以上五个文件，具体如下：

files: [

'angular.js',

'angular-mocks.js',

'angular-resource.js',

'home.js',

'home.test.js'

],

4） 运行以下命令开始测试：

karma start karma.conf.js

## **技巧**

* Karma会运行所有文件中的所有测试用例，如果你只想运行所有测试的一个子集，修改 describe 或 it 为 ddescribe 或 iit 来运行个别的一些测试。如果有些测试你不想运行他们，那么修改 describe 或 it 为 xdescribe 或 xit 来忽略这些代码。
* 我建议通篇阅读一下 [Jasmine文档](http://pivotal.github.io/jasmine/)来获取更多的可用方法。
* 你也可以在html文件的页面上运行你的测试。举例的代码如下：  
  **home.runner.html**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <!DOCTYPE html>  <html>  <head>      <title>Partner Settings Test Suite</title>      <!-- include your script files (notice that the jasmine source files have been added to the project) -->      <script type="text/javascript" src="../jasmine/jasmine-1.3.1/jasmine.js"></script>      <script type="text/javascript" src="../jasmine/jasmine-1.3.1/jasmine-html.js"></script>      <script type="text/javascript" src="../angular-mocks.js"></script>      <script type="text/javascript" src="home.tests.js"></script>      <link rel="stylesheet" href="../jasmine/jasmine-1.3.1/jasmine.css"/>  </head>  <body>      <!-- use Jasmine to run and display test results -->      <script type="text/javascript">          var jasmineEnv = jasmine.getEnv();          jasmineEnv.addReporter(new jasmine.HtmlReporter());          jasmineEnv.execute();      </script>  </body> |

## 15.3 AngularJS端对端测试

AngularJS端对端测试有好几种方法，其中有基于karma的karma-ng-scenario方法，该方法跟单元测试的方法类似，但本人还是不是很精通，因此在这里不予介绍。

下面我们详细讲解一下AngularJS自己集成的一种方法——protractor。这种方法要使用到webdriver-manager，而webdriver-manager需要翻墙才能下载得到，因此本人弄了好久才安装成功。

### 15.3.1 protractor测试环境的安装

以下是protractor测试环境的详细安装步骤：

（1）安装protractor

同前面一样，protractor也是运行在node.js的基础上，因此在电脑中也要先装node.js。node.js的安装在这里不讲。

接下来我们使用以下命令安装protractor：

npm install -g protractor

其中，-g表示全局变量，使用它可以将它安装到默认位置。（若由于操作错误想要卸载它，可使用npm uninstall -g protractor命令卸载。

安装成功后我们可以使用以下命令测试安装是否成功：

protractor --version

若安装成功则会显示出protractor的版本号。

（2）安装webdriver-manager

在电脑可以翻墙的情况下，webdriver-manager的安装我们可以直接使用以下命令完成：

webdriver-manager update

若电脑不能翻墙，则会显示timeout下载超时。遇到这种情况我们可以手动安装：

首先，下载以下3个文件：

1）chromedriver.exe

2）chromedriver\_2.10.zip

3）selenium-server-standalone-2.42.2.jar

然后将他们复制到protractor根目录的selenium目录下，接着我们需要更改一下配置：

打开protractor根目录下的config.json文件，在相应的位置修改对应文件的版本号。

{

"webdriverVersions": {

"selenium": "2.42.2",

"chromedriver": "2.10",

"iedriver": "2.45.0"

}

}

到此为止，protractor测试环境可以说已经安装完成了。

### 15.3.2 使用protractor进行端对端测试

下面讲一下如何使用protractor进行端对端测试：

（1）打开webdriver-manager服务

在测试之前必须先打开webdriver-manager服务。可使用以下命令完成：

webdriver-manager start

（2）新建一个目录，编写应用以及测试文件，如：

index.html

<!DOCTYPE html>

<html data-ng-app="protractor">

<head lang="en">

<meta charset="UTF-8">

<title></title>

</head>

<body>

<div data-ng-controller="myAppController">

{{userName}}

<input id="userName" data-ng-model="userName" />

</div>

</body>

<script type="text/javascript" src="angular.min.js"></script>

<script>

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

app.controller('myAppController', ['$scope',

function($scope) {

$scope.userName = 'Jackey';

}

]);

</script>

</html>

test.js

describe('angularjs homepage', function() {

it('should greet the named user', function() {

browser.get('http://127.0.0.1:8020/TT/index.html');

expect(element(by.binding('userName')).getText()).

toEqual('Jackey');

element(by.id('userName')).clear().sendKeys('xiedr');

expect(element(by.binding('userName')).getText()).

toEqual('xiedr');

browser.sleep(2000);

});

});

其中browser.get( )中的地址表示应用主页的地址，由于本人使用的是HBuilder因此他的地址可以表示为这样。

在这里只是给出一个例子，至于上例中涉及到的函数我们将在后面讲解。

（3）新建配置文件

conf.js

exports.config = {

framework: 'jasmine2', //使用到的框架（？？不是很懂）

seleniumAddress: 'http://localhost:4444/wd/hub', //服务器地址，（如何使用也不是很懂）

specs: ['test.js'] //测试文件

}

这是一个很简单的配置文件，可供参考。

（4）开始测试

打开一个新的命令控制器，跳转到工程目录，然后使用以下命令开始测试：

protractor conf.js

到此protractor端对端测试也叫介绍完了。

## \* 参考文献

### \*.1Tutorial

This is a simple tutorial that shows you how to set up Protractor and start running tests.

### \*.2 Prerequisites

Protractor is a Node.js program. To run, you will need to have Node.js installed. You will download Protractor package using npm, which comes with Node.js. Check the version of Node.js you have by running node --version. It should be greater than v0.10.0.

By default, Protractor uses the Jasmine test framework for its testing interface. This tutorial assumes some familiarity with Jasmine, and we will use version 2.3.

This tutorial will set up a test using a local standalone Selenium Server to control browsers. You will need to have the Java Development Kit (JDK) installed to run the standalone Selenium Server. Check this by running java -version from the command line.

### \*.3 Setup

Use npm to install Protractor globally with:

npm install -g protractor

This will install two command line tools, protr actor and webdriver-manager. Try running protractor --version to make sure it's working.

The webdriver-manager is a helper tool to easily get an instance of a Selenium Server running. Use it to download the necessary binaries with:

webdriver-manager update

Now start up a server with:

webdriver-manager start

This will start up a Selenium Server and will output a bunch of info logs. Your Protractor test will send requests to this server to control a local browser. Leave this server running throughout the tutorial. You can see information about the status of the server at http://localhost:4444/wd/hub.

### \*.4 Step 0 - write a test

Open a new command line or terminal window and create a clean folder for testing.

Protractor needs two files to run, a spec file and a configuration file.

Let's start with a simple test that navigates to an example AngularJS application and checks its title. We’ll use the Super Calculator application at http://juliemr.github.io/protractor-demo/.

Copy the following into spec.js:

// spec.js

describe('Protractor Demo App', function() {

it('should have a title', function() {

browser.get('http://juliemr.github.io/protractor-demo/');

expect(browser.getTitle()).toEqual('Super Calculator');

});

});

The describe and it syntax is from the Jasmine framework. browser is a global created by Protractor, which is used for browser-level commands such as navigation with browser.get.

Now create the configuration file. Copy the following into conf.js:

// conf.js

exports.config = {

framework: 'jasmine2',

seleniumAddress: 'http://localhost:4444/wd/hub',

specs: ['spec.js']

}

This configuration tells Protractor where your test files (specs) are, and where to talk to your Selenium Server (seleniumAddress). It specifies that we will be using Jasmine version 2 for the test framework. It will use the defaults for all other configuration. Chrome is the default browser.

Now run the test with

protractor conf.js

You should see a Chrome browser window open up and navigate to the Calculator, then close itself (this should be very fast!). The test output should be 1 tests, 1 assertion, 0 failures. Congratulations, you've run your first Protractor test!

### \*.5 Step 1 - interacting with elements

Now let's modify the test to interact with elements on the page. Change spec.js to the following:

// spec.js

describe('Protractor Demo App', function() {

it('should add one and two', function() {

browser.get('http://juliemr.github.io/protractor-demo/');

element(by.model('first')).sendKeys(1);

element(by.model('second')).sendKeys(2);

element(by.id('gobutton')).click();

expect(element(by.binding('latest')).getText()).

toEqual('5'); // This is wrong!

});

});

This uses the globals element and by, which are also created by Protractor. The element function is used for finding HTML elements on your webpage. It returns an ElementFinder object, which can be used to interact with the element or get information from it. In this test, we use sendKeys to type into <input>s, click to click a button, and getText to return the content of an element.

element takes one parameter, a Locator, which describes how to find the element. The by object creates Locators. Here, we're using three types of Locators:

by.model('first') to find the element with ng-model="first". If you inspect the Calculator page source, you will see this is <input type=text ng-model="first">.

by.id('gobutton') to find the element with the given id. This finds <button id="gobutton">.

by.binding('latest') to find the element bound to the variable latest. This finds the span containing {{latest}}

Run the tests with

protractor conf.js

You should see the page enter two numbers and wait for the result to be displayed. Because the result is 3, not 5, our test fails. Fix the test and try running it again.

### \*.6 Step 2 - writing multiple scenarios

Let's put these two tests together and clean them up a bit. Change spec.js to the following:

// spec.js

describe('Protractor Demo App', function() {

var firstNumber = element(by.model('first'));

var secondNumber = element(by.model('second'));

var goButton = element(by.id('gobutton'));

var latestResult = element(by.binding('latest'));

beforeEach(function() {

browser.get('http://juliemr.github.io/protractor-demo/');

});

it('should have a title', function() {

expect(browser.getTitle()).toEqual('Super Calculator');

});

it('should add one and two', function() {

firstNumber.sendKeys(1);

secondNumber.sendKeys(2);

goButton.click();

expect(latestResult.getText()).toEqual('3');

});

it('should add four and six', function() {

// Fill this in.

expect(latestResult.getText()).toEqual('10');

});

});

Here, we've pulled the navigation out into a beforeEach function which is run before every it block. We've also stored the ElementFinders for the first and second input in nice variables that can be reused. Fill out the second test using those variables, and run the tests again to ensure they pass.

### \*.7 Step 3 - changing the configuration

Now that we've written some basic tests, let's take a look at the configuration file. The configuration file lets you change things like which browsers are used and how to connect to the Selenium Server. Let's change the browser. Change conf.js to the following:

// conf.js

exports.config = {

framework: 'jasmine2',

seleniumAddress: 'http://localhost:4444/wd/hub',

specs: ['spec.js'],

capabilities: {

browserName: 'firefox'

}

}

Try running the tests again. You should see the tests running on Firefox instead of Chrome. The capabilities object describes the browser to be tested against. For a full list of options, see the reference config file.（见下文）

You can also run tests on more than one browser at once. Change conf.js to:

// conf.js

exports.config = {

framework: 'jasmine2',

seleniumAddress: 'http://localhost:4444/wd/hub',

specs: ['spec.js'],

multiCapabilities: [{

browserName: 'firefox'

}, {

browserName: 'chrome'

}]

}

Try running once again. You should see the tests running on Chrome and Firefox simultaneously, and the results reported separately on the command line.

### \*.8 Step 4 - lists of elements

Let's go back to the test files. Feel free to change the configuration back to using only one browser.

Sometimes, you will want to deal with a list of multiple elements. You can do this with element.all, which returns an ElementArrayFinder. In our calculator application, every operation is logged in the history, which is implemented on the site as a table with ng-repeat. Let's do a couple of operations, then test that they're in the history. Change spec.js to:

// spec.js

describe('Protractor Demo App', function() {

var firstNumber = element(by.model('first'));

var secondNumber = element(by.model('second'));

var goButton = element(by.id('gobutton'));

var latestResult = element(by.binding('latest'));

var history = element.all(by.repeater('result in memory'));

function add(a, b) {

firstNumber.sendKeys(a);

secondNumber.sendKeys(b);

goButton.click();

}

beforeEach(function() {

browser.get('http://juliemr.github.io/protractor-demo/');

});

it('should have a history', function() {

add(1, 2);

add(3, 4);

expect(history.count()).toEqual(2);

add(5, 6);

expect(history.count()).toEqual(0); // This is wrong!

});

});

We've done a couple things here - first, we created a helper function, add. We've added the variable history. We use element.all with the by.repeater Locator to get an ElementArrayFinder. In our spec, we assert that the history has the expected length using the count method. Fix the test so that the second expectation passes.

ElementArrayFinder has many methods in addition to count. Let's use last to get an ElementFinder that matches the last element found by the Locator. Change the test to:

it('should have a history', function() {

add(1, 2);

add(3, 4);

expect(history.last().getText()).toContain('1 + 2');

expect(history.first().getText()).toContain('foo'); // This is wrong!

});

Since the Calculator reports the oldest result at the bottom, the oldest addition (1 + 2) be the last history entry. We're using the toContain Jasmine matcher to assert that the element text contains "1 + 2". The full element text will also contain the timestamp and the result.

Fix the test so that it correctly expects the first history entry to contain the text "3 + 4".

ElementArrayFinder also has methods each, map, filter, and reduce which are analogous to JavaScript Array methods.

### \*.9 Where to go next

This should get you started writing tests.

## 15.4 AngularJS端对端测试详解

### 15.4.1 详细配置文件

// Reference Configuration File

//

// This file shows all of the configuration options that may be passed

// to Protractor.

//

// Because this file shows all of the options, if used in practice some

// will be overridden or ignored. If you're looking for a seed configuration

// file, see example/conf.js

exports.config = {

// ---------------------------------------------------------------------------

// ----- How to connect to Browser Drivers -----------------------------------

// ---------------------------------------------------------------------------

//

// Protractor needs to know how to connect to Drivers for the browsers

// it is testing on. This is usually done through a Selenium Server.

// There are four options - specify one of the following:

//

// 1. seleniumServerJar - to start a standalone Selenium Server locally.

// 2. seleniumAddress - to connect to a Selenium Server which is already

// running.

// 3. sauceUser/sauceKey - to use remote Selenium Servers via Sauce Labs.

// 4. directConnect - to connect directly to the browser Drivers.

// This option is only available for Firefox and Chrome.

// ---- 1. To start a standalone Selenium Server locally ---------------------

// The location of the standalone Selenium Server jar file, relative

// to the location of this config. If no other method of starting Selenium

// Server is found, this will default to

// node\_modules/protractor/selenium/selenium-server...

seleniumServerJar: null,

// The port to start the Selenium Server on, or null if the server should

// find its own unused port. Ignored if seleniumServerJar is null.

seleniumPort: null,

// Additional command line options to pass to selenium. For example,

// if you need to change the browser timeout, use

// seleniumArgs: ['-browserTimeout=60']

// Ignored if seleniumServerJar is null.

seleniumArgs: [],

// ChromeDriver location is used to help find the chromedriver binary.

// This will be passed to the Selenium jar as the system property

// webdriver.chrome.driver. If null, Selenium will

// attempt to find ChromeDriver using PATH.

chromeDriver: './selenium/chromedriver',

// ---- 2. To connect to a Selenium Server which is already running ----------

// The address of a running Selenium Server. If specified, Protractor will

// connect to an already running instance of Selenium. This usually looks like

// seleniumAddress: 'http://localhost:4444/wd/hub'

seleniumAddress: null,

// ---- 3. To use remote browsers via Sauce Labs -----------------------------

// If sauceUser and sauceKey are specified, seleniumServerJar will be ignored.

// The tests will be run remotely using Sauce Labs.

sauceUser: null,

sauceKey: null,

// Use sauceAgent if you need customize agent for https connection to

// saucelabs.com (i.e. your computer behind corporate proxy)

sauceAgent: null,

// Use sauceSeleniumAddress if you need to customize the URL Protractor

// uses to connect to sauce labs (for example, if you are tunneling selenium

// traffic through a sauce connect tunnel). Default is

// ondemand.saucelabs.com:80/wd/hub

sauceSeleniumAddress: null,

// ---- 4. To connect directly to Drivers ------------------------------------

// Boolean. If true, Protractor will connect directly to the browser Drivers

// at the locations specified by chromeDriver and firefoxPath. Only Chrome

// and Firefox are supported for direct connect.

directConnect: false,

// Path to the firefox application binary. If null, will attempt to find

// firefox in the default locations.

firefoxPath: null,

// ---------------------------------------------------------------------------

// ----- What tests to run ---------------------------------------------------

// ---------------------------------------------------------------------------

// Spec patterns are relative to the location of this config.

specs: [

'spec/\*\_spec.js'

],

// Patterns to exclude.

exclude: [],

// Alternatively, suites may be used. When run without a command line

// parameter, all suites will run. If run with --suite=smoke or

// --suite=smoke,full only the patterns matched by the specified suites will

// run.

suites: {

smoke: 'spec/smoketests/\*.js',

full: 'spec/\*.js'

},

// ---------------------------------------------------------------------------

// ----- How to set up browsers ----------------------------------------------

// ---------------------------------------------------------------------------

//

// Protractor can launch your tests on one or more browsers. If you are

// testing on a single browser, use the capabilities option. If you are

// testing on multiple browsers, use the multiCapabilities array.

// For a list of available capabilities, see

// https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities

//

// In addition, you may specify count, shardTestFiles, and maxInstances.

capabilities: {

browserName: 'chrome',

// Name of the process executing this capability. Not used directly by

// protractor or the browser, but instead pass directly to third parties

// like SauceLabs as the name of the job running this test

name: 'Unnamed Job',

// User defined name for the capability that will display in the results log

// Defaults to the browser name

logName: 'Chrome - English',

// Number of times to run this set of capabilities (in parallel, unless

// limited by maxSessions). Default is 1.

count: 1,

// If this is set to be true, specs will be sharded by file (i.e. all

// files to be run by this set of capabilities will run in parallel).

// Default is false.

shardTestFiles: false,

// Maximum number of browser instances that can run in parallel for this

// set of capabilities. This is only needed if shardTestFiles is true.

// Default is 1.

maxInstances: 1,

// Additional spec files to be run on this capability only.

specs: ['spec/chromeOnlySpec.js'],

// Spec files to be excluded on this capability only.

exclude: ['spec/doNotRunInChromeSpec.js'],

// Optional: override global seleniumAddress on this capability only.

seleniumAddress: null

},

// If you would like to run more than one instance of WebDriver on the same

// tests, use multiCapabilities, which takes an array of capabilities.

// If this is specified, capabilities will be ignored.

multiCapabilities: [],

// If you need to resolve multiCapabilities asynchronously (i.e. wait for

// server/proxy, set firefox profile, etc), you can specify a function here

// which will return either `multiCapabilities` or a promise to

// `multiCapabilities`.

// If this returns a promise, it is resolved immediately after

// `beforeLaunch` is run, and before any driver is set up.

// If this is specified, both capabilities and multiCapabilities will be

// ignored.

getMultiCapabilities: null,

// Maximum number of total browser sessions to run. Tests are queued in

// sequence if number of browser sessions is limited by this parameter.

// Use a number less than 1 to denote unlimited. Default is unlimited.

maxSessions: -1,

// ---------------------------------------------------------------------------

// ----- Global test information ---------------------------------------------

// ---------------------------------------------------------------------------

//

// A base URL for your application under test. Calls to protractor.get()

// with relative paths will be prepended with this.

baseUrl: 'http://localhost:9876',

// CSS Selector for the element housing the angular app - this defaults to

// body, but is necessary if ng-app is on a descendant of <body>.

rootElement: 'body',

// The timeout in milliseconds for each script run on the browser. This should

// be longer than the maximum time your application needs to stabilize between

// tasks.

allScriptsTimeout: 11000,

// How long to wait for a page to load.

getPageTimeout: 10000,

// A callback function called once configs are read but before any environment

// setup. This will only run once, and before onPrepare.

// You can specify a file containing code to run by setting beforeLaunch to

// the filename string.

beforeLaunch: function() {

// At this point, global variable 'protractor' object will NOT be set up,

// and globals from the test framework will NOT be available. The main

// purpose of this function should be to bring up test dependencies.

},

// A callback function called once protractor is ready and available, and

// before the specs are executed.

// If multiple capabilities are being run, this will run once per

// capability.

// You can specify a file containing code to run by setting onPrepare to

// the filename string.

onPrepare: function() {

// At this point, global variable 'protractor' object will be set up, and

// globals from the test framework will be available. For example, if you

// are using Jasmine, you can add a reporter with:

// jasmine.getEnv().addReporter(new jasmine.JUnitXmlReporter(

// 'outputdir/', true, true));

//

// If you need access back to the current configuration object,

// use a pattern like the following:

// browser.getProcessedConfig().then(function(config) {

// // config.capabilities is the CURRENT capability being run, if

// // you are using multiCapabilities.

// console.log('Executing capability', config.capabilities);

// });

},

// A callback function called once tests are finished.

onComplete: function() {

// At this point, tests will be done but global objects will still be

// available.

},

// A callback function called once the tests have finished running and

// the WebDriver instance has been shut down. It is passed the exit code

// (0 if the tests passed). This is called once per capability.

onCleanUp: function(exitCode) {},

// A callback function called once all tests have finished running and

// the WebDriver instance has been shut down. It is passed the exit code

// (0 if the tests passed). This is called only once before the program

// exits (after onCleanUp).

afterLaunch: function() {},

// The params object will be passed directly to the Protractor instance,

// and can be accessed from your test as browser.params. It is an arbitrary

// object and can contain anything you may need in your test.

// This can be changed via the command line as:

// --params.login.user 'Joe'

params: {

login: {

user: 'Jane',

password: '1234'

}

},

// If set, protractor will save the test output in json format at this path.

// The path is relative to the location of this config.

resultJsonOutputFile: null,

// If true, protractor will restart the browser between each test.

// CAUTION: This will cause your tests to slow down drastically.

restartBrowserBetweenTests: false,

// ---------------------------------------------------------------------------

// ----- The test framework --------------------------------------------------

// ---------------------------------------------------------------------------

// Test framework to use. This may be one of:

// jasmine, jasmine2, cucumber, mocha or custom.

//

// When the framework is set to "custom" you'll need to additionally

// set frameworkPath with the path relative to the config file or absolute

// framework: 'custom',

// frameworkPath: './frameworks/my\_custom\_jasmine.js',

// See github.com/angular/protractor/blob/master/lib/frameworks/README.md

// to comply with the interface details of your custom implementation.

//

// Jasmine and Jasmine2 are fully supported as test and assertion frameworks.

// Mocha and Cucumber have limited support. You will need to include your

// own assertion framework (such as Chai) if working with Mocha.

framework: 'jasmine2',

// Options to be passed to minijasminenode.

//

// See the full list at https://github.com/juliemr/minijasminenode/tree/jasmine1

jasmineNodeOpts: {

// If true, display spec names.

isVerbose: false,

// If true, print colors to the terminal.

showColors: true,

// If true, include stack traces in failures.

includeStackTrace: true,

// Default time to wait in ms before a test fails.

defaultTimeoutInterval: 30000

},

// Options to be passed to jasmine2.

//

// See https://github.com/jasmine/jasmine-npm/blob/master/lib/jasmine.js

// for the exact options available.

jasmineNodeOpts: {

// If true, print colors to the terminal.

showColors: true,

// Default time to wait in ms before a test fails.

defaultTimeoutInterval: 30000,

// Function called to print jasmine results.

print: function() {},

// If set, only execute specs whose names match the pattern, which is

// internally compiled to a RegExp.

grep: 'pattern',

// Inverts 'grep' matches

invertGrep: false

},

// Options to be passed to Mocha.

//

// See the full list at http://mochajs.org/

mochaOpts: {

ui: 'bdd',

reporter: 'list'

},

// Options to be passed to Cucumber.

cucumberOpts: {

// Require files before executing the features.

require: 'cucumber/stepDefinitions.js',

// Only execute the features or scenarios with tags matching @dev.

// This may be an array of strings to specify multiple tags to include.

tags: '@dev',

// How to format features (default: progress)

format: 'summary'

}

};

去掉注释后完整的配置文件如下：

exports.config = {

seleniumServerJar: null,

seleniumPort: null,

seleniumArgs: [],

chromeDriver: './selenium/chromedriver',

seleniumAddress: null,

sauceUser: null,

sauceKey: null,

sauceAgent: null,

sauceSeleniumAddress: null,

directConnect: false,

firefoxPath: null,

specs: [

'spec/\*\_spec.js'

],

exclude: [],

suites: {

smoke: 'spec/smoketests/\*.js',

full: 'spec/\*.js'

},

capabilities: {

browserName: 'chrome',

name: 'Unnamed Job',

logName: 'Chrome - English',

count: 1,

shardTestFiles: false,

maxInstances: 1,

specs: ['spec/chromeOnlySpec.js'],

exclude: ['spec/doNotRunInChromeSpec.js'],

seleniumAddress: null

},

multiCapabilities: [],

getMultiCapabilities: null,

maxSessions: -1,

baseUrl: 'http://localhost:9876',

rootElement: 'body',

allScriptsTimeout: 11000,

getPageTimeout: 10000,

beforeLaunch: function() {},

onPrepare: function() {},

onComplete: function() {},

onCleanUp: function(exitCode) {},

afterLaunch: function() {},

params: {

login: {

user: 'Jane',

password: '1234'

}

},

resultJsonOutputFile: null,

restartBrowserBetweenTests: false,

framework: 'jasmine2',

jasmineNodeOpts: {

isVerbose: false,

showColors: true,

includeStackTrace: true,

defaultTimeoutInterval: 30000

},

jasmineNodeOpts: {

showColors: true,

defaultTimeoutInterval: 30000,

print: function() {},

grep: 'pattern',

invertGrep: false

},

mochaOpts: {

ui: 'bdd',

reporter: 'list'

},

cucumberOpts: {

require: 'cucumber/stepDefinitions.js',

tags: '@dev',

format: 'summary'

}

};

### 15.4.2 测试文件的书写

由于这部分涉及的内容较多，我就不详细的讲了，在这里给出两个链接，在使用的过程中若有什么问题，可点链接查看：

（1）元素选择器——用于定位要操作的元素（locators and ElementFinders）：

https://github.com/angular/protractor/blob/master/docs/locators.md

（2）比较全面的protractor帮助文档

https://github.com/angular/protractor/blob/master/docs/toc.md

## \* 参考文件

以下参考文件是15.4.2 （1）中的链接文件。

### \*.1 Using Locators

The heart of end-to-end tests for webpages is finding DOM elements, interacting with them, and getting information about the current state of your application. This doc is an overview of how to locate and perform actions on DOM elements using Protractor.

### \*.2 Overview

Protractor exports a global function element, which takes a Locator and will return an ElementFinder. This function finds a single element - if you need to manipulate multiple elements, use the element.all function.

The ElementFinder has a set of action methods, such as click(), getText(), and sendKeys. These are the core way to interact with an element and get information back from it.

When you find elements in Protractor all actions are asynchronous. Behind the scenes, all actions are sent to the browser being controlled using the JSON Webdriver Wire Protocol. The browser then performs the action as a user natively would.

### \*.3 Locators

A locator tells Protractor how to find a certain DOM element. Protractor exports locator factories on the global by object. The most common locators are:

// find an element using a css selector

by.css('.myclass')

// find an element with the given id

by.id('myid')

// find an element with a certain ng-model

by.model('name')

// find an element bound to the given variable

by.binding('bindingname')

For a list of Protractor-specific locators, see the Protractor API: ProtractorBy.

The locators are passed to the element function, as below:

element(by.css('some-css'));

element(by.model('item.name'));

element(by.binding('item.name'));

When using CSS Selectors as a locator, you can use the shortcut $() notation:

$('my-css');

// Is the same as

element(by.css('my-css'));

### \*.4 Actions

The element() function returns an ElementFinder object. The ElementFinder knows how to locate the DOM element using the locator you passed in as a parameter, but it has not actually done so yet. It will not contact the browser until an action method has been called.

The most common action methods are:

var el = element(locator);

// Click on the element

el.click();

// Send keys to the element (usually an input)

el.sendKeys('my text');

// Clear the text in an element (usually an input)

el.clear();

// Get the value of an attribute, for example, get the value of an input

el.getAttribute('value');

Since all actions are asynchronous, all action methods return a promise. So, to log the text of an element, you would do something like:

var el = element(locator);

el.getText().then(function(text) {

console.log(text);

});

Any action available in WebDriverJS on a WebElement is available on an ElementFinder. See a full list.

### \*.5 Finding Multiple Elements

To deal with multiple DOM elements, use the element.all function. This also takes a locator as its only parameter.

element.all(by.css('.selector')).then(function(elements) {

// elements is an array of ElementFinders.

});

element.all() has several helper functions:

// Number of elements.

element.all(locator).count();

// Get by index (starting at 0).

element.all(locator).get(index);

// First and last.

element.all(locator).first();

element.all(locator).last();

### \*.6 Finding Sub-Elements

To find sub-elements, simply chain element and element.all functions together as shown below.

Using a single locator to find:

//an element

element(by.css('some-css'));

//a list of elements:

element.all(by.css('some-css'));

Using chained locators to find:

//a sub-element:

element(by.css('some-css')).element(by.tagName('tag-within-css'));

//to find a list of sub-elements:

element(by.css('some-css')).all(by.tagName('tag-within-css'));

You can chain with get/first/last as well like so:

element.all(by.css('some-css')).first().element(by.tagName('tag-within-css'));

element.all(by.css('some-css')).get(index).element(by.tagName('tag-within-css'));

element.all(by.css('some-css')).first().all(by.tagName('tag-within-css'));

### \*.7 Behind the Scenes: ElementFinders versus WebElements

If you're familiar with WebDriver and WebElements, or you're just curious about the WebElements mentioned above, you may be wondering how they relate to ElementFinders.

When you call driver.findElement(locator), WebDriver immediately sends a command over to the browser asking it to locate the element. This isn't great for creating page objects, because we want to be able to do things in setup (before a page may have been loaded) like

var myButton = ??;

and re-use the variable myButton throughout your test. ElementFinders get around this by simply storing the locator information until an action is called.

var myButton = element(locator);

// No command has been sent to the browser yet.

The browser will not receive any commands until you call an action.

myButton.click();

// Now two commands are sent to the browser - find the element, and then click it

ElementFinders also enable chaining to find subelements, such as element(locator1).element(locator2).

All WebElement actions are wrapped in this way and available on the ElementFinder, in addition to a couple helper methods like isPresent.

You can always access the underlying WebElement using element(locator).getWebElement(), but you should not need to.

# 第十六章 angular事件

## 16.1 事件的作用

AngularJS中的作用域具有继承性，并且结构是嵌套的。也就是说，对于每个Angular应用或者ng-app来说，只有一个主作用域$rootScope，其他所有作用域都会继承这个主作用域，或者嵌套在这个主作用域内部。通常情况下，你会发现作用域之间不会共享变量，并且互相之间不会基于原型继承。

如果需要在两个作用域之间进行通信，应该怎么做呢？

其中一种方法是：创建一个服务，这个服务在整个应用范围内是一个单例，并且所有作用域之间的交互都通过它进行。

另一种方法就是使用本章所讲的事件：通过作用域上的事件进行交互。但是，这样做存在一些局限性。例如，你通常不能向正在监听的所有作用域同时广播事件。你必须选择是和父节点交互还是和子节点交互（事件的发布那里将会讲到）。

下面是一个简单的例子：

<!doctype html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<style>

\*{

margin-bottom: 10px;

}

</style>

</head>

<body>

<div ng-controller="grandpaCtrl">

<button ng-click="grandpaclick()">爷爷</button>

<div ng-controller="fatherCtrl">

<button ng-click="fatherclick()">爸爸</button>

<div ng-controller="myselfCtrl">

<button ng-click="myselfclick()">自己</button>

</div>

<div ng-controller="brotherCtrl">

<button ng-click="brotherclick()">兄弟</button>

</div>

</div>

</div>

<script>

var myApp = angular.module("myApp", []);

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

$scope.grandpaclick = function () {

$scope.$broadcast('grandpasaid', '爷爷说：我到了');

console.log('爷爷说：我到了');

}

$scope.$on('myselfsaid', function (event,data){

//console.log(data);

console.log('爷爷说：孙子，我去接你');

});

});

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

$scope.fatherclick = function () {

console.log('jjj');

}

$scope.$on('grandpasaid', function (event,data){

console.log('爸爸说：爸，我去接你');

});

$scope.$on('myselfsaid', function (event,data){

console.log('爸爸说：儿子，不用麻烦爷爷了，我去接你');

event.stopPropagation();//阻止事件冒泡

});

});

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

$scope.myselfclick = function () {

console.log('我说：我到了');

$scope.$emit('myselfsaid', '我说：我到了');

}

$scope.$on('grandpasaid', function (event,data){

console.log('我说：爷爷，我去接你');

});

});

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

$scope.brotherclick = function () {

console.log('ddd');

}

$scope.$on('grandpasaid', function (event,data){

console.log('弟弟说：爷爷，我去接你');

});

//平级收不到

$scope.$on('myselfsaid', function (event,data){

console.log('弟弟说：哥，你说什么？我听不见');

});

});

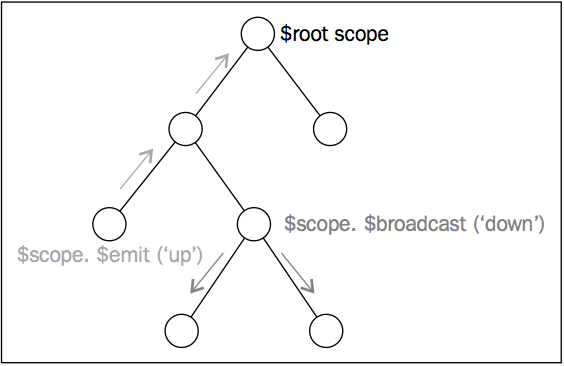
</script>

</body>

</html>

## 16.2 事件的发布

Angular事件的发布主要有两种方法：$emit()和$broadcast()。它们分别代表着事件传播的两个方向，即从下到上和从上到下，如下图：

[](http://www.leikunsheng.com/wp-content/uploads/2014/05/ch1_p20.png)

下面我将它们分开进行详细讲解。

### 16.2.1 $emit()

$emit()事件传播的方向是从子作用域冒泡到父作用域，产生事件的作用域之上的所有作用域都会收到这个事件的通知（除非在某个节点上阻止了冒泡，后面会介绍）。具体使用格式如下：

$emit(eventName, dataArgs);

其中，

eventName —— 表示向parentsController传递的事件名称；

dataArgs —— 表示向parentsController传递的数据。

下面是一个例子：

<!doctype html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<style>

\*{

margin-bottom: 10px;

}

</style>

</head>

<body>

<div ng-controller="grandpaCtrl">

<button ng-click="grandpaclick()">爷爷</button>

<div ng-controller="fatherCtrl">

<button ng-click="fatherclick()">爸爸</button>

<div ng-controller="myselfCtrl">

<button ng-click="myselfclick()">自己</button>

</div>

</div>

</div>

<script>

var myApp = angular.module("myApp", []);

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

$scope.grandpaclick = function () {

//

}

$scope.$on('myselfsaid', function (event,data){

console.log('爷爷听到: ' + data.toG);

});

});

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

$scope.fatherclick = function () {

//

}

$scope.$on('myselfsaid', function (event,data){

console.log('爸爸听到: ' + data.toF);

});

});

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

$scope.myselfclick = function () {

console.log('我说：我说了两句话');

var dataArg = {};

dataArg.toG = '我对爷爷说';

dataArg.toF = '我对爸爸说';

$scope.$emit('myselfsaid', dataArg);

}

});

</script>

</body>

</html>

### 16.2.2 $broadcast()

由$broadcast()发布的事件传播的方向是传播方向是从上至下，即从父作用域到子作用域。具体使用格式和$emit()一样：

$broadcast(eventName, dataArgs);

参数的意义也和$emit()一样，在这里也不在赘述。

## 16.3 事件的监听

事件发布了，接下来当然是要影响事件，此时我们要注册监听事件的服务。在Angular中我们我们可以通过$on()方法来注册事件监听事件。格式如下：

$scope.$on(event, listenerFn);

其中，

event —— 监听的事件，其所包含的属性在【6.4】中详细介绍；

listenerFn —— 事件触发后的响应函数。

具体例子可看上面中的两个例子。

## 16.4 事件的对象

所有的事件监听器第一个参数都代表了事件对象，该对象有以下属性：

* targetScope作用域对象，发送事件的作用域
* currentScope:当前处理事件的作用域
* name:当前事件的事件的名称
* stopPropagation:取消$emit触发的事件进一步的传播
* preventDefault：把defaultPrevented标志设置为true，告诉子作用域可以无需处理该事件。
* defaultPrevented：布尔值

具体请看下面的例子：

<!doctype html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

<style>

\*{

margin-bottom: 10px;

}

</style>

</head>

<body>

<div ng-controller="grandpaCtrl">

<button ng-click="grandpaclick()">爷爷</button>

<div ng-controller="fatherCtrl">

<button ng-click="fatherclick()">爸爸</button>

<div ng-controller="myselfCtrl">

<button ng-click="myselfclick()">自己</button>

</div>

</div>

</div>

<script>

var myApp = angular.module("myApp", []);

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

$scope.grandpaclick = function () {

//

}

$scope.$on('myselfsaid', function (event,data){

console.log('爷爷听到: ' + data.toG);

});

});

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

$scope.fatherclick = function () {

//

}

$scope.$on('myselfsaid', function (event,data){

console.log(event);

});

});

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

$scope.myselfclick = function () {

console.log('我说：我说了两句话');

var dataArg = {};

dataArg.toG = '我对爷爷说';

dataArg.toF = '我对爸爸说';

$scope.$emit('myselfsaid', dataArg);

}

});

</script>

</body>

</html>

在这里我只是把event的各个属性打印出来，具体怎么使用这些属性我还不是很明白。

下面是一个阻止ng-click冒泡的一个例子：

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body ng-app="myApp" ng-controller="myCtrl">

<div style="width: 300px; height: 300px; background-color: yellow;" ng-click="parents()">

<div style="width: 100px; height: 100px; background-color: gray;" ng-click="chirld($event)"></div>

</div>

<script>

var myApp = angular.module("myApp", []);

myApp.controller("myCtrl", function($scope) {

$scope.parents = function () {

console.log('parents');

}

$scope.chirld = function ($event) {

console.log('chirld');

$event.stopPropagation();

}

});

</script>

</body>

</html>

## 16.5 Angular内置事件

* $includeContentLoaded（$emit事件）

ngInclude内容重新加载的时候，从ngInclude指令触发

* $includeContentRequested（$emit事件）

从调用ngInclude的作用域上发送，每次ngInclude的内容被请求的时候，都会发布该事件

* $viewContentLoaded（$emit事件）

当ngView内容被重新加载时，从ngView作用域上发布

* $locationChangeStart($broadcast事件)

通过$location服务对浏览器的地址更新时会触发$locationChangeStart事件

* $locationChangeSuccess($broadcast事件)

当浏览器的地址成功变更时触发

* $routeChangeStart($broadcast事件)

在路由变更发生之前，该事件从$rootScope发布

**（例子？？？？？？？？）**

# 第十七章 架构

**（由于本人从事前端开发经验不足，本章暂不讨论）**

……

# 第十八章 动画

AngularJS自1.2版本起，可以在一些常用的directive上绑定animation，比如ngRepeat, ngSwitch和ngView，也可以在自定义的directive上绑定animation（使用$animation）。在使用了某些directives的元素的生命周期中，能够通过一些animation的钩子来触发animation事件，当触发时，会表现出动画效果（动画效果通过css或javascript定义）。可以使用纯css（遵循AngularJS中的一些规范）或者javascript代码（定义在服务中）来设置animation。

## 18.1 创建动画的前提条件

AngularJS中ngAnimate模块支持动画效果，但是ngAnimate模块并未包含在AngularJS核心库中，因此要使用动画，首先需要引入angular-animate.min.js库，如：

<script type="text/javascript" src="js/angular-animate.min.js" ></script>

另外，还需在定义Module时注入ngAnimate依赖，如：

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

除了以上的库和依赖外，使用ngAnimate常常会使用到Angular内部定义的一些Directive。

其中，官方给出的能支持动画效果的Directives有：

|  |  |
| --- | --- |
| Directive | Supported Animations |
| [*ngRepeat*](https://docs.angularjs.org/api/ng/directive/ngRepeat#animations) | enter, leave and move |
| [*ngView*](https://docs.angularjs.org/api/ngRoute/directive/ngView#animations) | enter and leave |
| [*ngInclude*](https://docs.angularjs.org/api/ng/directive/ngInclude#animations) | enter and leave |
| [*ngSwitch*](https://docs.angularjs.org/api/ng/directive/ngSwitch#animations) | enter and leave |
| [*ngIf*](https://docs.angularjs.org/api/ng/directive/ngIf#animations) | enter and leave |
| [*ngClass*](https://docs.angularjs.org/api/ng/directive/ngClass#animations) | add and remove (the CSS class(es) present) |
| [*ngShow*](https://docs.angularjs.org/api/ng/directive/ngShow#animations) &  [*ngHide*](https://docs.angularjs.org/api/ng/directive/ngHide#animations) | add and remove (the ng-hide class value) |
| [*form*](https://docs.angularjs.org/api/ng/directive/form#animation-hooks) &  [*ngModel*](https://docs.angularjs.org/api/ng/directive/ngModel#animation-hooks) | add and remove (dirty, pristine, valid, invalid & all other validations) |
| [*ngMessages*](https://docs.angularjs.org/api/ngMessages#animations) | add and remove (ng-active & ng-inactive) |
| [*ngMessage*](https://docs.angularjs.org/api/ngMessages#animations) | enter and leave |

这些属性具体怎么使用可参照下文。

## 18.2 Angular自定义动画的创建方式

AngularJS中实现动画效果有两大种方式：

* 基于CSS的动画效果

CSS Transition Animation

CSS Class-based Animation

* 基于Javascript的动画效果

### 18.2.1 基于CSS的动画效果

基于CSS 的动画效果又有两种使用方式：

**1. CSS Transition Animation**

使用CSS Transition时，ng-EVENT（动画开始前的样式）和ng-EVENT-active（动画执行完毕后的样式）这两组样式必须同时出现，且在ng-EVENT中必须包含transition的设置。如：

<!DOCTYPE >

<html>

<head>

<style type="text/css">

/\* 开始时的样式 \*/

.fade.ng-enter {

transition: 5s linear all;

/\* 当使用css transition实现动画效果时，在开始时的样式中必须包含transition的设置 \*/

opacity: 0;

}

/\* 结束时的样式 \*/

.fade.ng-enter.ng-enter-active {

opacity: 1;

}

</style>

<script type="text/javascript" src="js/angular.min.js" ></script>

<script type="text/javascript" src="js/angular-animate.min.js" ></script>

<meta charset="utf-8" />

<script type="text/javascript">

(function() {

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

})();

</script>

</head>

<body ng-app="cssBasedAnimationTest">

<div ng-if="bool" class="fade">

Fade me in out

</div>

<button ng-click="bool=true">Fade In!</button>

<button ng-click="bool=false">Fade Out!</button>

</body>

</html>

本例中，我们使用ngIf（ng-if）来控制class="fade"的元素是否加载到DOM。

注意：ngIf和ngShow/ngHide不同，ngIf在元素隐藏时DOM中并没有该元素，而使用ngShow/ngHide时元素依然存在于DOM中。

上面的例子执行过程如下：

点击"Fade In!"按钮时，加载div到DOM并触发enter，AngularJS对元素自动添加ng-enter和ng-enter-active的样式，并从ng-enter到ng-enter-active执行样式转换。（具体变化可打开控制台上的源码进行观察）

CSS transition的方法除了上面那种方式外，我们还可使用CSS的Keyframe关键字，具体操作只对样式文件进行了修改，如：

<!DOCTYPE >

<html>

<head>

<style type="text/css">

/\* 开始时的样式，使用keyframes不需要定义结束时的样式 \*/

.fade.ng-enter {

animation: my\_fade\_animation 5s linear;

-webkit-animation: my\_fade\_animation 5s linear;

}

@keyframes my\_fade\_animation {

from {

opacity: 0;

}

to {

opacity: 1;

}

}

@-webkit-keyframes my\_fade\_animation {

from {

opacity: 0;

}

to {

opacity: 1;

}

}

</style>

<script type="text/javascript" src="js/angular.min.js"></script>

<script type="text/javascript" src="js/angular-animate.min.js"></script>

<meta charset="utf-8" />

<script type="text/javascript">

(function() {

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

})();

</script>

</head>

<body ng-app="cssBasedAnimationTest">

<div ng-if="bool" class="fade">

Fade me in out

</div>

<button ng-click="bool=true">Fade In!</button>

<button ng-click="bool=false">Fade Out!</button>

</body>

</html>

**2. CSS Class-based Animation**

Class-based Animation即为通过ngClass、ngShow、ngHide等Directives执行动画效果。如：

<!DOCTYPE >

<html>

<head>

<style type="text/css">

.fade.ng-hide {

transition: 3s linear all;

opacity: 0;

}

.fade.ng-show {

transition: 3s linear all;

opacity: 1;

}

</style>

<script type="text/javascript" src="js/angular.min.js" ></script>

<script type="text/javascript" src="js/angular-animate.min.js" ></script>

<script type="text/javascript">

(function() {

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

})();

</script>

</head>

<body ng-app="cssClassBasedAnimationTest" ng-init="bool=true">

<div ng-show="bool" class="fade">

This is ng-show.

</div>

<div ng-hide="bool" class="fade">

This is ng-hide.

</div>

<button ng-click="bool=!bool">Toggle</button>

</body>

</html>

关于上面这个例子，当我们打开控制台时，我们可以看到与第一个例子不同，在本例中DOM节点一直存在（只是将元素隐藏而已）。观察他们的变化可以看到是由于元素的class属性的值在发生变化从而产生了动画效果。

另外，由于ngAminate能检测的行为是样式的add或者remove，因此我们还可以显式的指定add和remove的样式，如：

<!DOCTYPE >

<html>

<head>

<style type="text/css">

.highlight {

transition: 3s linear all;

}

.highlight.on-add {

background: white;

}

.highlight.on {

background: yellow;

}

.highlight.on-remove {

background: black;

}

</style>

<script type="text/javascript" src="js/angular.min.js" ></script>

<script type="text/javascript" src="js/angular-animate.min.js" ></script>

<script type="text/javascript">

(function() {

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

})();

</script>

</head>

<body ng-app="cssClassBasedAnimationTest" ng-init="onOff=true">

<div ng-class="{on:onOff}" class="highlight">

Highlight this box

</div>

<button ng-click="onOff=!onOff">Toggle</button>

</body>

</html>

我们让ng-class随着点击Toggle按钮变化，当onOff=true时样式on会被ngAnimate执行on-add的过程，反之则执行on-remove的过程。由于显式指定了样式，当我们运行示例3时，这个过程就一目了然了。

### 18.2.2基于Javascript的动画效果

使用基于Javascript的动画效果可以让你在脚本中使用其他的Service甚至引用第三方的脚本进行动画的制作，使动画效果更丰富多变。

与基于CSS的动画效果相似，基于Javascript的动画效果也会由AngularJS自动添加一些指定的样式到元素上，但基于Javascript的动画效果还需要使用 module . animation()添加动画脚本。如：

<!DOCTYPE >

<html>

<head>

<script type="text/javascript" src="js/angular.min.js" ></script>

<script type="text/javascript" src="js/angular-animate.min.js" ></script>

<script type="text/javascript" src="js/jquery-2.1.4.min.js" ></script>

<script type="text/javascript">

(function() {

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

app.animation('.slide', [

function() {

return {

enter: function(element, doneFn) {

jQuery(element).fadeIn(1000, doneFn);

},

leave: function(element, doneFn) {

jQuery(element).fadeOut(1000, doneFn);

}

}

}

]);

app.controller('myController', ['$scope',

function($scope) {

$scope.students = ["Tom", "Jack", "Alice", "May", "Thomas"];

}

]);

})();

</script>

</head>

<body ng-app="javascriptBasedAnimationTest" ng-controller="myController">

<div ng-if="isshow" ng-repeat="stu in students" class="slide">

{{ stu }}

</div>

<input type="button" value="Toggle" ng-click="isshow=!isshow" />

</body>

</html>

本例中，我们对class='slide'的元素的AngularJS中的默认行为添加了动画行为，当使用ngIf让元素enter或者leave时，将触发动画效果。

## 18.3 为自定义的directives绑定动画

在自定义的directives中使用animate，首先需要将$animate服务注入directives中，$animate服务在主Angular文件中有定义，因此不需要额外添加库文件（建议还是引入angular-animate库，这样动画效果会更佳），如：

app.directive('myCustomDirective', ['$animate',

function($animate) {

return {

//

}

}

]);

上面这个代码只是在定义directives注入了$animate服务，其他部分（如directives的功能）尚未进行声明。

动画的具体怎么实现我们一般写在directive服务的Link函数中，如：

app.directive('myCustomDirective', ['$animate',

function($animate) {

return {

link: function(scope, element, attrs) {

element.bind('click', function() {

if (element.hasClass('clicked')) {

$animate.removeClass(element, 'clicked');

} else {

$animate.addClass(element, 'clicked');

}

scope.$apply();

});

}

}

}

]);

从上面的代码中，我们不难看出这个directive的动画效果是：当点击该元素时，如果元素中没有添加clicked类则在元素的class属性添加clicked类；如果元素已经有了clicked类则在class属性中将clicked类移除。不过在这里需要注意一下就是此时需要刷新页面才能看得到页面变化，因此需要$apply()。在页面上具体怎么使用，如参照如下代码：

<!DOCTYPE >

<html>

<head>

<meta charset="utf-8" />

<style>

.clicked {

color: blue;

}

</style>

<script type="text/javascript" src="js/angular.min.js"></script>

<script type="text/javascript" src="js/angular-animate.min.js"></script>

<script type="text/javascript">

(function() {

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

app.controller('myController', ['$scope',

function($scope) {

$scope.students = 'I am a customer directives!'

}

]);

app.directive('myCustomDirective', ['$animate',

function($animate) {

return {

link: function(scope, element, attrs) {

element.bind('click', function() {

if (element.hasClass('clicked')) {

$animate.removeClass(element, 'clicked');

} else {

$animate.addClass(element, 'clicked');

}

scope.$apply();

});

}

}

}

]);

})();

</script>

</head>

<body ng-app="javascriptBasedAnimationTest" ng-controller="myController">

<div my-custom-directive>

{{ students }}

</div>

</body>

</html>

以上是一个比较简单的例子，由于时间关系本人尚未对此节内容进行深入研究。

## \* 参考文献

### \*.1 Animating with AngularJS

By[*Holly Schinsky*](http://modernweb.com/authors/holly-schinsky)

[AngularJS](http://angularjs.org/) recently came out with support for CSS3 transitions and animations, as well as JavaScript Animations. The support is part of version 1.1.4 (unstable build), but was changed and refined a bit in [version 1.1.5](http://code.angularjs.org/1.1.5/) so you should start with that version when you check it out. I definitely think it’s worth trying because it allows you to add some fun interactions to your application quickly.

In this article I’ll explain a bit about how it all works and include links to a [demo application](http://devgirl.org/files/AngularAnimationsDemo/#/) I created to try things our for yourself. The source to the demo application is located on my GitHub account [here](https://github.com/hollyschinsky/AngularAnimationsDemo) as well. I also included some great resources in the form of links at the end of the post. There’s currently not a lot of documentation on this subject since it is so new, so I encourage you to check those out as well.

### \*.2 How it Works…

ngAnimate is the name of the new [directive](http://docs.angularjs.org/guide/directive) for AngularJS animation support. The way it’s applied is by adding the ng-animate attribute to any element containing one of the following directives in the list below:

* [ngIf](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngIf)
* [ngInclude](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngInclude)
* [ngRepeat](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngRepeat)
* [ngShow](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngShow) / [ngHide](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngHide)
* [ngSwitch](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngSwitch)
* [ngView](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngView)

The only exception to the above is when you create your own custom directive with animation support using the*$animator*service. This is discussed later in the article.

Each of these directives causes a change to the DOM, which is how the transition or animation is triggered. For instance, on an ngRepeat they will occur when the items are repeating, forngShow/ngHide, when the element is being shown or hidden. You simply specify ng-animateon your element with the name of the classes you’ve defined in your CSS to perform the transitions or animations and they will be applied automatically. Of course, you need to ensure you specify the ng-animate on the same element where you have one of the directives mentioned above (ngRepeat, ngSwitch etc.) defined or nothing will happen.

Here’s a quick example of applying a scale type of transition the shorthand way (read on for notation details):

<ng-include ng-animate="'scale'" src="'partials/quote.html'"></ng-include>

Then in your CSS, you define a corresponding set of classes prefixed with scale that are used to trigger the transition or animation based on the type of event you want to animate for that directive. More info to come on this…

### \*.3 Supported Events

Certain events are supported for each of the directives you can animate. They vary per directive and it’s important to know which apply for a given directive when you’re defining your CSS classes to perform the animation.

Here’s the list:

|  |  |
| --- | --- |
| **Directive** | **Events** |
| ngIf | enter/leave |
| ngInclude | enter/leave |
| ngRepeat | enter/leave/move |
| ngShow/ngHide | show/hide |
| ngSwitch | enter/leave |
| ngView | enter/leave |

The [AngularJS docs](http://code.angularjs.org/1.1.5/docs/) describe exactly when those events occur for each of the directives supporting animations. So for instance the ngIf docs provide this description for the supportedenter and leave events:

**enter** – happens just after the ngIf contents change and a new DOM element is created and injected into the ngIf container

**leave** – happens just before the ngIf contents are removed from the DOM

You can use this information to determine when your transitions and animations will actually be triggered.

### \*.4 CSS3 Transitions versus CSS3 Animations

ngAnimate can be used for both CSS3 animations and transitions, as well as JavaScript animations but that is beyond the scope of this article. I wanted to take a moment to briefly discuss the difference between CSS3 transitions and animations.

CSS3 Transitions  
apply an effect to a style property for a certain duration. You can do things like fade, scale, move, slide, rotate, 3D effects etc.

CSS3 Animations are more complex than transitions and use keyframes to define different points to do things within the animation. These can be looped and auto-started. In addition, they don’t have to depend on a DOM change to trigger them.

The difference in using CSS3 transitions versus CSS3 animations with ngAnimate is all in the way the CSS classes are defined. A great example can be found in the [AngularJS 1.1.5 docs for ngAnimate](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngAnimate). I’m including it here too for easy reference:

### \*.5 CSS3 Transition Sample

.animate-enter {

-webkit-transition: 1s linear all; /\* Safari/Chrome \*/

-moz-transition: 1s linear all; /\* Firefox \*/

-o-transition: 1s linear all; /\* Opera \*/

transition: 1s linear all; /\* IE10+ and Future Browsers \*/

/\* The animation preparation code \*/

opacity: 0;

}

/\*

Keep in mind that you want to combine both CSS

classes together to avoid any CSS-specificity

conflicts

\*/

.animate-enter.animate-enter-active {

/\* The animation code itself \*/

opacity: 1;

}

### \*.6 CSS3 Animation Sample

.animate-enter {

-webkit-animation: enter\_sequence 1s linear; /\* Safari/Chrome \*/

-moz-animation: enter\_sequence 1s linear; /\* Firefox \*/

-o-animation: enter\_sequence 1s linear; /\* Opera \*/

animation: enter\_sequence 1s linear; /\* IE10+ and Future Browsers \*/

}

@-webkit-keyframes enter\_sequence {

from { opacity:0; }

to { opacity:1; }

}

@-moz-keyframes enter\_sequence {

from { opacity:0; }

to { opacity:1; }

}

@-o-keyframes enter\_sequence {

from { opacity:0; }

to { opacity:1; }

}

@keyframes enter\_sequence {

from { opacity:0; }

to { opacity:1; }

}

When using ngAnimate for either of the above, the same syntax would be used to apply the directive, such as:

<div ng-view ng-animate="{enter: 'animate-enter'}"></div>

I highly recommend reading through [the documentation](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngAnimate) for further details on how this is all handled.

### \*.7 Naming Conventions

When writing your CSS classes to apply your transitions or animations, you define two CSS classes for each supported event (enter, leave, show, hide, etc). One is used as a baseline class and the other is defined as an active class where the animation actually happens. There is a certain notation you need to use to define these classes. The base class is named as name-event (ie: slide-enter), whereas the active class is named the same with the keyword “active” appended to it (ie: slide-enter-active).

For example, using a fade type of effect on an ngShow or ngHide would require these 4 classes to be defined (since “show” and “hide” are the supported events for that directive):

.fade-show {

-webkit-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

-moz-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

-ms-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

-o-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

white-space:nowrap;

position:relative;

overflow: hidden;

text-overflow: clip;

}

.fade-show-active {

opacity:1;

}

.fade-hide {

-webkit-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

-moz-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

-ms-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

-o-transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

transition: 400ms cubic-bezier(0.250, 0.250, 0.750, 0.750) all;

white-space:nowrap;

position:relative;

overflow: hidden;

text-overflow: clip;

}

.fade-hide-active {

opacity:0;

}

### \*.8 Applying the CSS via ng-animate

Once you define all of your animations and transitions, you apply them via the ngAnimatedirective. There are two ways you can describe which CSS classes to use, explicitly or shorthand. Both are shown below:

Explicit:  
Here you explicitly name the class to apply for each supported event (enter, leave etc).

<div ng-view ng-animate="{enter: 'rotate-enter', leave: 'rotate-leave'}"></div>

Shorthand:  
The shorthand way of applying them is through a name only, and then that name with the event appended is implied as the class name. So for instance, in the following, the resulting class names implied will be name-event or rotate-enter and rotate-leave, just as above:

<div ng-view ng-animate="'rotate'">

Check out the [demo source](http://devgirl.org/files/AngularAnimationsDemo/#/) or links provided for specific examples to understand the syntax further.

### \*.9 Customizing Animation

You can customize your AngularJS animation in different ways using the $animator service. You could create your own custom animation events or use the built-in ones like “enter” and “leave” with ng-animate in your own custom directive by accessing them off the [$animator](http://code.angularjs.org/1.1.5/docs/api/ng.$animator)service.

You can also define custom events using the [$animator.animate(myEvent,element)](http://code.angularjs.org/1.1.5/docs/api/ng.$animator)function, where myEvent is your own String and element is what to apply the transition or animation to.

I included [an example](http://devgirl.org/files/AngularAnimationsDemo/#/customView) of defining a custom directive and custom event in the demo application if you are interested in seeing how it works.

### \*.10 Related Resources

* [Demo Application](http://devgirl.org/files/AngularAnimationsDemo/#/)
* [AngularJS 1.1.5 Docs](http://code.angularjs.org/1.1.5/docs/api/ng.directive:ngAnimate) for ngAnimate
* [nganimate.org](http://nganimate.org/) – Great samples here
* Really nice [detailed article](http://www.yearofmoo.com/2013/05/enhanced-animations-in-angularjs.html) about the new animations support in AngularJS
* [AngularJS and Animation Slides with demos](http://slid.es/gsklee/animation-in-angularjs) by Gias Kay Lee
* [Animate your AngularJS apps with CSS3 and jQuery article](http://gsklee.im/post/50254705713/nganimate-your-angularjs-apps-with-css3-and-jquery)
* [Misko Hevery video](http://www.youtube.com/watch?v=cF_JsA9KsDM) talking about the new animation features
* [Swipe Demo and code](https://github.com/jeffbcross/ngswipe-demo) using ngAnimate
* [Video showing swipe demo and code](https://plus.google.com/115279700532270609876/posts/L9q6HFFdrgj)
* [All About CSS3 Transitions](http://www.kirupa.com/html5/all_about_css_transitions.htm)
* [All About CSS3 Animations](http://www.kirupa.com/html5/all_about_css_animations.htm)

# 第十九章digest循环和$apply

这部分的内容主要与Angular双向数据绑定有关，一般情况下不需要我们显式的调用，因为Angular会自动为我们添加进去。但一些个别的情况下Angular监测不到数据的变化，这时就需要我们显式地去调用这部分的内容来实现数据的双向绑定。

## 19.1 双向数据绑定

在前面我们已经介绍过双向数据绑定，在这里我们再重复一遍双向数据绑定的定义。双向数据绑定就是意味着当View中有任何数据发生了变化，那么这个变化也会自动地反馈到scope的数据上，也即意味着scope模型会自动地更新。类似地，当scope模型发生变化时，view中的数据也会更新到最新的值。

## 19.2 双向数据绑定的实现原理

Angular是则么实现双向数据绑定的呢？在介绍这个原理之前我们首先介绍几个相关的函数。

### 19.2.1 $watch

$watch方法主要是用来判断model有没有发生变化。如果model发生了变化，关联到$watch上的回调函数就会被触发，从而更新scope或试图。

一般情况下，每次当我们绑定一些东西到UI上时，Angular就会自动往$watch队列里插入一条$watch。当有model发生变化时，响应的$watch就会被触发，从而及时的刷新了页面。

从上面的描述我们也可以看出，一般情况下我们是不需要显式地去调用$watch（因为Angular会自动调用）。但如果我们希望在model发生变化时做一些额外的操作，此时我们可以调用$watch实现，如下面的例子中，当name的值发生变化时，updated就会自动加1：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<title>$watch</title>

<script type="text/javascript" src="js/angular.min.js" ></script>

</head>

<body ng-controller="myCtrl">

<input ng-model="name" />

<br/>

Name updated: {{updated}} times.

<script>

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

app.controller('myCtrl', function($scope) {

$scope.name = "Angular";

$scope.updated = -1;

$scope.$watch('name', function() {

$scope.updated++;

});

});

</script>

</body>

</html>

$watch的调用格式一般如下：

$scope.$watch('aModel', function(newValue, oldValue) {

//update the DOM with newValue

});

### 19.2.2 $digest

前面我们已经介绍过当我们绑定一个model到UI上时，Angular会自动将一个$watch插入到$watch队列，然而当model发生变化时，相应的$watch不是马上就可以知道，因此我们就需要一个函数有规律地循环从而遍历各个$watch，监测有没有model发生变化。这个函数就是$digest。

$digest循环开始后，它会触发每个$watch。这些$watch会检查scope中的当前model值是否和上一次计算得到的model值不同。如果不同，那么对应的回调函数会被执行。调用该函数的结果，就是view中的表达式内容会被更新。

一般地，我们可以使用Angular的一些built-in指令（如ng-click）以及服务来更改models(比如ng-model，$timeout等)和自动触发一次$digest循环。

但是，AngularJS并不直接调用$digest()，而是调用$scope.$apply()，后者会调用$rootScope.$digest()。因此，一轮$digest循环在$rootScope开始，随后会访问到所有的children scope中的watchers。

### 19.2.3 $apply

$apply的作用其实和$digest的作用差不多。

$scope.$apply()会自动地调用$rootScope.$digest()。

$apply()方法有两种形式：

* 第一种会接受一个function作为参数，执行该function并且触发一轮$digest循环。
* 第二种会不接受任何参数，只是触发一轮$digest循环。我们马上会看到为什么第一种形式更好。

$apply的具体例子在这里没有给出，因为一般情况下Angular会自动触发，不需要我们显式去调用。下面我们看一下需要我们手动去调用$apply的情况。

## 19.3什么时候需要手动调用$apply()方法

实际上，AngularJS对这个问题有着非常明确的要求，就是它只负责对发生于AngularJS上下文环境中的变更会做出自动地响应(即，在$apply()方法中发生的对于models的更改)。AngularJS的built-in指令就是这样做的，所以任何的model变更都会被反映到view中。

但是，如果你在AngularJS上下文之外的任何地方修改了model，那么你就需要通过手动调用$apply()来通知AngularJS。这就像告诉AngularJS，你修改了一些models，希望AngularJS帮你触发watchers来做出正确的响应。

比如，如果你使用了JavaScript中的setTimeout()来更新一个scope model，那么AngularJS就没有办法知道你更改了什么。这种情况下，调用$apply()就是你的责任了，通过调用它来触发一轮$digest循环。类似地，如果你有一个指令用来设置一个DOM事件listener并且在该listener中修改了一些models，那么你也需要通过手动调用$apply()来确保变更会被正确的反映到view中。

具体让我们来看一个例子。假如你有一个页面，一旦该页面加载完毕了，你希望在两秒钟之后显示一条信息。你的实现可能是下面这个样子的：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<title>$watch</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="MessageController">

Delayed Message: {{message}}

</div>

<script>

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

/\* What happens without an $apply() \*/

app.controller('MessageController', function($scope) {

$scope.getMessage = function() {

setTimeout(function() {

$scope.$apply(function() {//如果屏蔽掉$apply你可以看到2秒钟后只有控制台打印出message，而页面上却没有打印出

$scope.message = 'Fetched after 2 seconds';

console.log('message:' + $scope.message);

});

}, 2000);

}

$scope.getMessage();

});

</script>

</body>

</html>

由于原生javascript的setTimeout()函数中不可能调用$apply()，因此我们需要额外的调用$apply()（jquery也是一样，因此使用jquery对象也是需要手动的调用$apply()）。

顺便提一下，对于上面的例子，如果你使用了Angular内置的$timeout函数，那就不需要显式的调用$apply了。

另外，上面的例子还可以使用$apply()的第二种形式：

<!DOCTYPE html>

<html ng-app="myApp">

<head>

<meta charset="utf-8">

<title>$watch</title>

<script type="text/javascript" src="js/angular.min.js"></script>

</head>

<body>

<div ng-controller="MessageController">

Delayed Message: {{message}}

</div>

<script>

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

/\* What happens without an $apply() \*/

app.controller('MessageController', function($scope) {

$scope.getMessage = function() {

setTimeout(function() {

$scope.message = 'Fetched after 2 seconds';

console.log('message:' + $scope.message);

$scope.$apply();

}, 2000);

}

$scope.getMessage();

});

</script>

</body>

</html>

以上的代码使用了$apply()的第二种形式，也就是没有参数的形式。

不过建议你使用接受一个function作为参数的$apply()方法。这是因为当你传入一个function到$apply()中的时候，这个function会被包装到一个try…catch块中，所以一旦有异常发生，该异常会被$exceptionHandler service处理。

## \* $observe

最后单独说一下$observe，他是在定义Directive的时候，针对link的第三个参数attr来做的监听attr，实际上就是定义的元素上的attribute参数，通过attr可以获取DOM中的参数。

通过$observe来进行监听，当参数的值发生变化了，就会触发回调函数，他和watch的不同就是，监听的宿主对象不同。

## \* 参考资料

### \*.1 Understanding Angular’s $apply() and $digest()

$apply() and $digest() are two core, and sometimes confusing, aspects of AngularJS. To understand how AngularJS works one needs to fully understand how $apply() and $digest() work. This article aims to explain what $apply() and $digest() really are, and how they can be useful in your day-to-day AngularJS programming.

### \*.2 $apply and $digest Explored

AngularJS offers an incredibly awesome feature known as two way data binding which greatly simplifies our lives. Data binding means that when you change something in the view, the scope model automagicallyupdates. Similarly, whenever the scope model changes, the view updates itself with the new value. How does does AngularJS do that? When you write an expression ({{aModel}}), behind the scenes Angular sets up a watcher on the scope model, which in turn updates the view whenever the model changes. Thiswatcher is just like any watcher you set up in AngularJS:

|  |  |
| --- | --- |
| 1  2  3 | $scope.$watch('aModel', function(newValue, oldValue) {    //update the DOM with newValue  }); |

The second argument passed to $watch() is known as a listener function, and is called whenever the value of aModel changes. It is easy for us to grasp that when the value of aModel changes this listener is called, updating the expression in HTML. But, there is still one big question! How does Angular figure out when to call this listener function? In other words, how does AngularJS know when aModel changes so it can call the corresponding listener? Does it run a function periodically to check whether the value of the scope model has changed? Well, this is where the $digest cycle steps in.

It’s the $digest cycle where the watchers are fired. When a watcher is fired, AngularJS evaluates thescope model, and if it has changed then the corresponding listener function is called. So, our next question is when and how this $digest cycle starts.

The $digest cycle starts as a result of a call to $scope.$digest(). Assume that you change a scopemodel in a handler function through the ng-click directive. In that case AngularJS automatically triggers a$digest cycle by calling $digest(). When the $digest cycle starts, it fires each of the watchers. These watchers check if the current value of the scope model is different from last calculated value. If yes, then the corresponding listener function executes. As a result if you have any expressions in the view they will be updated. 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.

So far, so good! But, there is a small gotcha. In the above cases, Angular doesn’t directly call $digest(). Instead, it calls $scope.$apply(), which in turn calls $rootScope.$digest(). As a result of this, a digest cycle starts at the $rootScope, and subsequently visits all the child scopes calling the watchers along the way.

Now, let’s assume you attach an ng-click directive to a button and pass a function name to it. When the button is clicked, AngularJS wraps the function call within $scope.$apply(). So, your function executes as usual, change models (if any), and a $digest cycle starts to ensure your changes are reflected in the view.

**Note**: $scope.$apply() automatically calls $rootScope.$digest(). The $apply() function comes in two flavors. The first one takes a function as an argument, evaluates it, and triggers a $digest cycle. The second version does not take any arguments and just starts a $digest cycle when called. We will see why the former one is the preferred approach shortly.

### \*.3 When Do You Call $apply() Manually?

If AngularJS usually wraps our code in $apply() and starts a $digest cycle, then when do you need to do call $apply() manually? Actually, AngularJS makes one thing pretty clear. It will account for only those model changes which are done inside AngularJS’ context (i.e. the code that changes models is wrapped inside $apply()). Angular’s built-in directives already do this so that any model changes you make are reflected in the view. However, if you change any model outside of the Angular context, then you need to inform Angular of the changes by calling $apply() manually. It’s like telling Angular that you are changing some models and it should fire the watchers so that your changes propagate properly.

For example, if you 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 your 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.

Let’s look at an example. Suppose you have a page, and once the page loads you want to display a message after a two second delay. Your implementation might look something like the JavaScript and HTML shown in the following listing.

By running the example, you will see that the delayed function runs after a two second interval, and updates the scope model message. Still, the view doesn’t update. The reason, as you may have guessed, is that we forgot to call $apply() manually. Therefore, we need to update our getMessage() function as shown below.

If you run this updated example, you can see the view update after two seconds. The only change is that we wrapped our code inside $scope.$apply() which automatically triggers $rootScope.$digest(). As a result the watchers are fired as usual and the view updates.

**Note**: By the way, you should use $timeout service whenever possible which is setTimeout() with automatic $apply() so that you don’t have to call $apply() manually.

Also, note that in the above code you could have done the model changes as usual and placed a call to$apply() (the no-arg version) in the end. Have a look at the following snippet:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $scope.getMessage = function() {    setTimeout(function() {      $scope.message = 'Fetched after two seconds';      console.log('message:' + $scope.message);      $scope.$apply(); //this triggers a $digest    }, 2000);  }; |

The above code uses the no-arg version of $apply() and works. Keep in mind that you should always use the version of $apply() that accepts a function argument. This is because when you pass a function to$apply(), the function call is wrapped inside a try...catch block, and any exceptions that occur will be passed to the $exceptionHandler service.

### \*.4 How Many Times Does the $digest Loop Run?

When a $digest cycle runs, the watchers are executed to see if the scope models have changed. If they have, then the corresponding listener functions are called. This leads to an important question. What if a listener function itself changed a scope model? How would AngularJS account for that change?

The answer is that the $digest loop doesn’t run just once. At the end of the current loop, it starts all over again to check if any of the models have changed. This is basically dirty checking, and is done to account for any model changes that might have been done by listener functions. So, the $digest cycle keeps looping until there are no more model changes, or it hits the max loop count of 10. It’s always good to stay idempotent and try to minimize model changes inside the listener functions.

**Note**: At a minimum, $digest will run twice even if your listener functions don’t change any models. As discussed above, it runs once more to make sure the models are stable and there are no changes.

### \*.5 Conclusion

I hope this article has clarified what $apply and $digest are all about. The most important thing to keep in mind is whether or not Angular **can** detect your changes. If it cannot, then you must call $apply()manually.

# 第二十章 揭秘Angular

这部分的内容主要讲述AngularJS内部执行的原理。这部分的内容我只是搜索网上的一些资料然后整理一下，但没有进行深入地去研究，可作为了解了解。

## 20.1 AngularJS双向数据绑定揭秘

AngularJS在$scope变量中使用脏值检查来实现了数据双向绑定。和Ember.js数据双向绑定中动态设施setter和getter不同，脏治检查允许AngularJS监视那些存在或者不存在的变量。

### 20.1.1 $scope.$watch

$scope.$watch( watchExp, listener, objectEquality );

为了监视一个变量的变化，你可以使用$scope.$watch函数。这个函数有三个参数，它指明了”要观察什么”(watchExp)，”在变化时要发生什么”(listener),以及你要监视的是一个变量还是一个对象。当我们在检查一个参数时，我们可以忽略第三个参数。例如下面的例子：

$scope.name = 'Ryan';

$scope.$watch(function() {

return $scope.name;

}, function(newValue, oldValue) {

console.log('$scope.name was updated!');

});

AngularJS将会在$scope中注册你的监视函数。你可以在控制台中输出$scope来查看$scope中的注册项目。

你可以在控制台中看到$scope.name已经发生了变化 – 这是因为$scope.name之前的值似乎undefined而现在我们将它赋值为Ryan!

当你调用 $scope.$watch 的时候, 实际上干了2件事情. watch的第一个参数是一个function, 这个function的返回你想监控的对象(如果你传递的是一个string, Angular会把他转换为一个function). digest cycle 运行的时候, 它会调用这个function. 第二个参数也是一个function, 当第一个function的值发生变化的时候它会被调用. 让我们看看他是怎么实现监控的:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

$watch: function(watchExp, listener, objectEquality) {

var get = $parse(watchExp);

if (get.$$watchDelegate) {

return get.$$watchDelegate(this, listener, objectEquality, get);

}

var scope = this,

array = scope.$$watchers,

watcher = {

fn: listener,

last: initWatchVal,

get: get,

exp: watchExp,

eq: !!objectEquality

};

lastDirtyWatch = null;

if (!isFunction(listener)) {

watcher.fn = noop;

}

if (!array) {

array = scope.$$watchers = [];

}

// we use unshift since we use a while loop in $digest for speed.

// the while loop reads in reverse order.

array.unshift(watcher);

return function deregisterWatch() {

arrayRemove(array, watcher);

lastDirtyWatch = null;

};

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

这个方法将会把参数添加到scope中的 $$watchers 数组中, 并且它会返回一个function, 以便于你想结束这个监控操作.

对于$wach的第一个参数，你也可以使用一个字符串。这和提供一个函数完全一样。在AngularJS的源代码中可以看到，如果你使用了一个字符串，将会运行下面的代码：

if (typeof watchExp == 'string' && get.constant) {

var originalFn = watcher.fn;

watcher.fn = function(newVal, oldVal, scope) {

originalFn.call(this, newVal, oldVal, scope);

arrayRemove(array, watcher);

};

}

这将会把我们的watchExp设置为一个函数，它也自动返回作用域中我们已经制定了名字的变量。

### 20.1.2 $$watchers

$scope中的$$watchers变量保存着我们定义的所有的监视器。如果你在控制台中查看$$watchers，你会发现它是一个对象数组。

$$watchers = [{

eq: false, // 表明我们是否需要检查对象级别的相等

fn: function(newValue, oldValue) {}, // 这是我们提供的监听器函数

last: 'Ryan', // 变量的最新值

exp: function() {}, // 我们提供的watchExp函数

get: function() {} // Angular's编译后的watchExp函数

}];

$watch函数将会返回一个deregisterWatch函数。这意味着如果我们使用$scope.$watch对一个变量进行监视，我们也可以在以后通过调用某个函数来停止监视。

### 20.1.3 $scope.$apply

当一个控制器/指令/等等东西在AngularJS中运行时，AngularJS内部会运行一个叫做$scope.$apply的函数。这个$apply函数会接收一个函数作为参数并运行它，在这之后才会在rootScope上运行$digest函数。

AngularJS的$apply函数代码如下所示：

$apply: function(expr) {

try {

beginPhase('$apply');

return this.$eval(expr);

} catch (e) {

$exceptionHandler(e);

} finally {

clearPhase();

try {

$rootScope.$digest();

} catch (e) {

$exceptionHandler(e);

throw e;

}

}

}

上面代码中的expr参数就是你在调用$scope.$apply()时传递的参数 – 但是大多数时候你可能都不会去使用$apply这个函数，要用的时候记得给它传递一个参数。

### 20.1.4 digest cycle

然后digest cycle会在每次调用 $scope.$apply 或者 $scope.$digest 的时候运行. $scope.$apply 实际上是一个rootScope的包装, 他会从根$rootScope向下广播. 而 $scope.$digest 只会在当前scope中运行(并向下级scope广播).

$digest: function() {

var watch, value, last,

watchers,

asyncQueue = this.$$asyncQueue,

postDigestQueue = this.$$postDigestQueue,

length,

dirty, ttl = TTL,

next, current, target = this,

watchLog = [],

logIdx, logMsg, asyncTask;

beginPhase('$digest');

lastDirtyWatch = null;

do { // "while dirty" loop

dirty = false;

current = target;

traverseScopesLoop:

do { // "traverse the scopes" loop

if ((watchers = current.$$watchers)) {

// process our watches

length = watchers.length;

while (length--) {

try {

watch = watchers[length];

// Most common watches are on primitives, in which case we can short

// circuit it with === operator, only when === fails do we use .equals

if (watch) {

if ((value = watch.get(current)) !== (last = watch.last) &&

!(watch.eq ? equals(value, last) : (typeof value === 'number' && typeof last === 'number' && isNaN(value) && isNaN(last)))) {

dirty = true;

lastDirtyWatch = watch;

watch.last = watch.eq ? copy(value, null) : value;

watch.fn(value, ((last === initWatchVal) ? value : last), current);

if (ttl < 5) {

logIdx = 4 - ttl;

if (!watchLog[logIdx]) watchLog[logIdx] = [];

logMsg = (isFunction(watch.exp)) ? 'fn: ' + (watch.exp.name || watch.exp.toString()) : watch.exp;

logMsg += '; newVal: ' + toJson(value) + '; oldVal: ' + toJson(last);

watchLog[logIdx].push(logMsg);

}

} else if (watch === lastDirtyWatch) {

// If the most recently dirty watcher is now clean, short circuit since the remaining watchers

// have already been tested.

dirty = false;

break traverseScopesLoop;

}

}

} catch (e) {

$exceptionHandler(e);

}

}

}

} while ((current = next));

// `break traverseScopesLoop;` takes us to here

if ((dirty || asyncQueue.length) && !(ttl--)) {

clearPhase();

throw $rootScopeMinErr('infdig',

'{0} $digest() iterations reached. Aborting!\n' +

'Watchers fired in the last 5 iterations: {1}',

TTL, toJson(watchLog));

}

} while (dirty || asyncQueue.length);

clearPhase();

while (postDigestQueue.length) {

try {

postDigestQueue.shift()();

} catch (e) {

$exceptionHandler(e);

}

}

}

这个实现非常有才, 虽然我没有进去看它是如何向下级广播的, 但这里的关键是循环遍历 $$watchers , 执行里面的函数(就是那个你通过 $scope.$watch 注册的第一个function), 然后如果得到和之前不同的值, 他又将调用listener(那个你传递的第二个function). 然后, 砰!我们得到了一个变量发生改变的通知. 关键是我们是如何知道一个值发生变化了的? 当一个值被更新的时候digest cycle会运行(尽管它可能不是必须的). 比如在 ng-model 上, 每一个keydown事件都会触发digest cycle.

### 20.1.5 ng-keydown

下面我们来看看ng-keydown是怎么来使用$scope.$apply的。为了注册这个指令，AngularJS会使用下面的代码。

var ngEventDirectives = {};

forEach(

'click dblclick mousedown mouseup mouseover mouseout mousemove mouseenter mouseleave keydown keyup keypress submit focus blur copy cut paste'.split(' '),

function(name) {

var directiveName = directiveNormalize('ng-' + name);

ngEventDirectives[directiveName] = ['$parse',

function($parse) {

return {

compile: function($element, attr) {

var fn = $parse(attr[directiveName]);

return function ngEventHandler(scope, element) {

element.on(lowercase(name), function(event) {

scope.$apply(function() {

fn(scope, {

$event: event

});

});

});

};

}

};

}

];

}

);

上面的代码做的事情是循环了不同的类型的事件，这些事件在之后可能会被触发并创建一个叫做ng-[某个事件]的新指令。在指令的compile函数中，它在元素上注册了一个事件处理器，它和指令的名字一一对应。当事件被出发时，AngularJS就会运行scope.$apply函数，并让它运行一个函数。

上面所说的ng-keydown只能够改变和元素值相关联的$scope中的值 – 这只是单向数据绑定。这是ng-keydown指令的原因，只有在keydown事件被触发时，才能够给与我们一个新值。

### 20.1.6 ng-model

我们现在来看一看ng-model。当你在使用ng-model时，你可以使用双向数据绑定 – 这正是我们想要的。AngularJS使用$scope.$watch（视图到模型）以及$scope.$apply（模型到视图）来实现这个功能。

ng-model会把事件处理指令(例如keydown)绑定到我们运用的输入元素上 – 这就是$scope.$apply被调用的地方！而$scope.$watch是在指令的控制器中被调用的。你可以在下面代码中看到这一点：

$scope.$watch(function ngModelWatch() {

var value = ngModelGet($scope);

//如果作用域模型值和ngModel值没有同步

if (ctrl.$modelValue !== value) {

var formatters = ctrl.$formatters,

idx = formatters.length;

ctrl.$modelValue = value;

while (idx--) {

value = formatters[idx](value);

}

if (ctrl.$viewValue !== value) {

ctrl.$viewValue = value;

ctrl.$render();

}

}

return value;

});

如果你在调用$scope.$watch时只为它传递了一个参数，无论作用域中的什么东西发生了变化，这个函数都会被调用。在ng-model中，这个函数被用来检查模型和视图有没有同步，如果没有同步，它将会使用新值来更新模型数据。这个函数会返回一个新值，当它在$digest函数中运行时，我们就会知道这个值是什么！

### 20.1.7 为什么我们的监听器没有被触发？

如果我们在$scope.$watch的监听器函数中停止这个监听，即使我们更新了$scope.name，该监听器也不会被触发。

正如前面所提到的，AngularJS将会在每一个指令的控制器函数中运行$scope.$apply。如果我们查看$scope.$apply函数的代码，我们会发现它只会在控制器函数已经开始被调用之后才会运行$digest函数 – 这意味着如果我们马上停止监听，$scope.$watch函数甚至都不会被调用！但是它究竟是怎样运行的呢？

$digest函数将会在$scope.$apply中被$rootScope所调用。它将会在$rootScope中运行digest循环，然后向下遍历每一个作用域并在每个作用域上运行循环。在简单的情形中，digest循环将会触发所有位于$$watchers变量中的所有watchExp函数，将它们和最新的值进行对比，如果值不相同，就会触发监听器。

当digest循环运行时，它将会遍历所有的监听器然后再次循环，只要这次循环发现了”脏值”，循环就会继续下去。如果watchExp的值和最新的值不相同，那么这次循环就会被认为发现了脏值。理想情况下它会运行一次，如果它运行超10次，你会看到一个错误。

因此当$scope.$apply运行的时候，$digest也会运行，它将会循环遍历$$watchers，只要发现watchExp和最新的值不相等，变化触发事件监听器。在AngularJS中，只要一个模型的值可能发生变化，$scope.$apply就会运行。这就是为什么当你在AngularJS之外更新$scope时，例如在一个setTimeout函数中，你需要手动去运行$scope.$apply()：这能够让AngularJS意识到它的作用域发生了变化。

## 20.2依赖注入的实现原理

依赖注入(DI)让我们可以不用自己实例化就能创建依赖对象的方法. 简单的来说, 依赖是以注入的方式传递的. 在Web应用中， Angular让我们可以通过DI来创建像Controllers和Directives这样的对象. 我们还可以创建自己的依赖对象, 当我们要实例化它们时, Angular能自动实现注入.

最常见的被注入对象应该是 $scope 对象. 它可以像下面这样被注入的:

function MainCtrl($scope) {

// access to $scope

}

angular

.module('app')

.controller('MainCtrl', MainCtrl);

对于从来没有接触过依赖注入的Javascript开发人员来说, 这样看起来只是像传递了一个参数. 而实际上, 他是一个依赖注入的占位符. Angular通过这些占位符, 把真正的对象实例化给我们, 让来看看他是怎么实现的.

### 20.2.1 function的参数

当你运行你代码的时候, 如果你把function声明中的参数换成一个其它字母, 那么Angular就无法找到你真正想实例化的对象. 因为Angular在我们的function上使用了toString() 方法, 他将把我们的整个function变成一个字符串， 然后解析function中声明的每一个参数.它使用下面4个正则(RegExps)来完成这件事情.

var FN\_ARGS = /^function\s\*[^\(]\*\(\s\*([^\)]\*)\)/m;

var FN\_ARG\_SPLIT = /,/;

var FN\_ARG = /^\s\*(\_?)(\S+?)\1\s\*$/;

var STRIP\_COMMENTS = /((\/\/.\*$)|(\/\\*[\s\S]\*?\\*\/))/mg;

Angular做的第一件事情就是把我们的整个function转换为字符串, 这确实是Javascript很强大的地方. 转换后我们将得到如下字符串:

'function MainCtrl ($scope) {...}'

然后, 他用正则移除了在 function() 中有可能的所有的注释.

fnText = fn.toString().replace(STRIP\_COMMENTS, '');

接着它提取其中的参数部分.

argDecl = fnText.match(FN\_ARGS);

最后它使用 .split() 方法来移除参数中的所有空格, 完美! Angular使用一个内部的forEach 方法来遍历这些参数, 然后把他们放入一个 $inject 数组中.

forEach(argDecl[1].split(FN\_ARG\_SPLIT), function(arg) {

arg.replace(FN\_ARG, function(all, underscore, name) {

$inject.push(name);

});

});

正如你现在想的, 这是一个很大的性能开销操作. 每个函数都要执行4个正则表达式还有大量的转换操作----这将给我们带来性能损失. 不过我们可以通过直接添加需要注入的对象到 $inject 数组中的方式来避免这个开销.

### 20.2.2 $inject

我们可以在function对象上添加一个 $inject 属性来告诉Angular我们的依赖对象. 如果对象是存在的, Angular将实例化它. 这样的语法更具有可读性, 因为我们可以这些对象是被注入的. 下面是一个例子:

function SomeCtrl($scope) {

//

}

SomeCtrl.$inject = ['$scope'];

angular

.module('app', [])

.controller('SomeCtrl', ['$scope', SomeCtrl]);

这将节省框架的大量操作, 它不用再解析function的参数, 也不用去操作数组(查看下一节数组参数), 它可以直接获取我们已经传递给他的 $inject 属性. 简单, 高效.

理想情况下我们应该使用构建工具, 比如 Grunt.js 或者 Gulp.js 来做这个事情: 让他们在编译时生成相应的 $injext 属性, 这样能让Web应用运行的更快.

注: 实际上上面介绍的内容并不涉如何实例化那些需要被注入的对象. 整个操作只是标记出需要的名字----实例化的操作将由框架的另一部分来完成.

### 20.2. 3数组参数

最后要提到的是数组参数. 数组的前面每个元素的名字和顺序, 刚是数组最后一个元素function的参数名字和顺序. 比如:

['$scope', function ($scope) {}]

这个顺序是非常重要的, 因为Angular是以这个顺序来实例化对象. 如果顺序不正确, 那么它可能将其它对象错误的实例化到你真正需要的对象上.

function SomeCtrl($scope, $rootScope) {

//

}

angular

.module('app', [])

.controller('SomeCtrl', ['$scope', '$rootScope', SomeCtrl]);

像上面一样, 我们需要做的就是把函数作为数组的最后一个元素. 然后Angular会遍历前面的每一个元素, 把它们添加到 $inject 数组中. 当Angular开始解析一个函数的时候, 它会先检查目标对象是不是一个数组类型, 如果是的话, 他将把最后一个元素作为真正的function, 其它的元素都作为依赖对象添加到 $inject 中.

} else if (isArray(fn)) {

last = fn.length - 1;

assertArgFn(fn[last], 'fn');

$inject = fn.slice(0, last);

}

## 20.3 Factory和Service

Factory和Service看起来非常相似, 以至于很多开发人员都无法理解它们有什么不同.

当实例化一个 .service() 的时候, 其实他将通过调用 new Service() 的形式来给我们创建一个新的实例, .service() 的方法像是一个构造函数.

服务(service)实际上来说是一个最基本的工厂(factory), 但是它是通过 new 来创建的, 你需要使用 this 来添加你需要的变量和函数, 最后返回这个对象.

工厂(factory)实际上是非常接近面向对象中的"工厂模式(factory pattern)". 当你调用时, 它会创建新的实例. 本质上来说, 那个实例是一个全新的对象.

下面是Angular内部实际执行的源码:

function factory(name, factoryFn) {

return provider(name, {

$get: factoryFn

});

}

function service(name, constructor) {

return factory(name, ['$injector',

function($injector) {

return $injector.instantiate(constructor);

}

]);

}

## 20.4从 $rootScope 中创建新的 $scope

所有的scope对象都继承于 $rootScope , $rootScope 又是通过 new Scope()来创建的. 所有的子scope都是用过调用 $scope.$new() 来创建的.

var $rootScope = new Scope();

它内部有一个 $new 方法, 让新的scope可以从原型链上引用它们的父scope, 子scope(为了digest cycle), 以及前后的scope.

从下面的代码可以看出, 如果你想创建一个独立的scope, 那么你应该使用 new Scope() , 否则它将以继承的方式来创建.

我省略了一些不必要的代码, 下面是他的核心实现

$new: function(isolate) {

var child;

if (isolate) {

child = new Scope();

child.$root = this.$root;

} else {

// Only create a child scope class if somebody asks for one,

// but cache it to allow the VM to optimize lookups.

if (!this.$$ChildScope) {

this.$$ChildScope = function ChildScope() {

this.$$watchers = null;

};

this.$$ChildScope.prototype = this;

}

child = new this.$$ChildScope();

}

child['this'] = child;

child.$parent = this;

return child;

}

理解这一点对写测试非常重要, 如果你想测试你的Controller, 那么你应该使用$scope.$new() 来创建$scope对象. 明白scope是如何创建的在测试驱动开发(TDD)中是十分重要的,这将更加有助于你mock module.

## 20.5 ANGULAR事件机制揭秘

和其他任何Javascript框架一样， 浏览器在构建DOM之后就载入了AngularJS库。

当浏览器触发DOMContentLoaded事件时，Angular就开始工作了。

第一件事就是找到ng-app，如果存在，Angular就会自动初始化运行环境，不然你就要手工运行Angular.bootstrap()来手工初始化运行环境。

然后，ng-app所赋给的值会用来配置$injector服务，当所有app全部载入以后，$injector会创建$compile服务和app的$rootScope(根上下文)。

这时，$compile会接过控制权，首先吧$rootScope链接上DOM，然后从ng-app标记的DOM根元素开始编译整个DOM树。

### 20.5.1视图工作原理

当浏览器接收到HTML的内容时，开始把Html解析成DOM树。当浏览器扫描的脚本文件时(<script>元素)，它会暂停解析，知道脚本接收完成。当接受到Angular.js脚本，该脚本的运行会建立一个事件侦听器，侦听DOMContentLoaded事件。这个事件会在所有HTML内容被扫描和解析完成以后被触发。

当DOMContentLoaded事件触发Angular的侦听器时，Angular就启动初始化。先找到ng-app，然后创建一些必要的组件(如$injector, $scompile和$rootScope等)，最后开始解析DOM树。

### 20.5.2编译过程

$compile服务遍历DOM树，收集所有的标识(Directives)，然后把所有标识的linking函数合并成一个linking函数。这个函数专用把编译完成的模板连接到$rootScope。标识(Directives)遍布DOM的所有地方，如属性，注释，类和元素。

<span my-directive></span>

<span class="my-directive"></span>

<my-directive>

<my-directive>

<!--directive:my-directive-->

在遍历过程中，当$compile在某个元素上找到了一个或多个Directives时，它先更具它们的Priority(优先级)排序，然后用$injector服务找到该Directive的compile函数并执行它。

Directive的compile函数主要功能是DOM转换货内嵌模板，因为它会产生一个模版的克隆。

在每个DOM节点都编译之后，$compile会调用linking函数。linking函数会把所有的Directive和其对应的Scope建立双向数据半丁，这个过程称为动态化视图。

当$compile完成其工作以后，Angular运行时开始启动。

### 20.5.3编译过程

浏览器的事件循环时刻在等待各种事件，如鼠标移动，点击或按键。当这些事件发生时，他们被存入浏览器事件队列中。如果有任何函数绑定到该事件时，这些函数会被调用，该事件作为调用传入的一个参数。

ele.addEventListener('click', function(event){});

事件循环在Angular中有点争议，因为Angular自己也有一个事件循环。Angular的事件循环被称为$digest循环，又有两个小循环组成：evalAsyn循环和$watch列表。

当事件触发时，它在Angular标识的上下文被调用。实际上，Angular是在$apply()方法中调用标识。整个流程开始于Angular在$rootScope级别启动$digest循环，然后递归的处理子scopes。

当Angular进入$digest循环中三，它需要等待$evalAsync队列全部处理清空后，才把回调的控制权传递给浏览器。 $evalAsync是用来保存当前工作栈之外的待执行任务，这些任务必须在浏览器回显之前执行。

另外$digest循环还在等待$watch表达式列表。上一次对$watch表达式的处理循环会导致一些数据的变化，而这些数据变化反过来又可能又会导致$watch表达式结果的进一步变动。因此$watch表达式还需要再一次计算，确保不再有新的变化产生。

当$digest循环完成，执行权交还给浏览器，浏览器将会刷新DOM.

整个流程在每一次浏览器实践触发以后，都会一次又一次的启动执行。这也是Angular的强大之处，也使得浏览器事件很容易的注入到Angular的流程中去。

# 第二十一章 Angular精华扩展

这部分的内容主要涉及基于AngularJS的一些插件，由于目前关于这方面的插件有很多，因此在这里暂时不做展开讲述。

如果有时间可以上网搜索一些资料了解了解。