--

[初始 1](#_Toc26204)

[学习网址 1](#_Toc32103)

[www 1](#_Toc9952)

[大漠穷秋 2](#_Toc724)

[第一步 2](#_Toc14671)

[一篇总结 3](#_Toc31884)

[概念 3](#_Toc28089)

[核心概念 3](#_Toc23735)

[Angular特性 3](#_Toc11930)

[ng模块 3](#_Toc26893)

[$scope 3](#_Toc17227)

[作用域 3](#_Toc2557)

[控制器 4](#_Toc7956)

[ng初始化加载流程 4](#_Toc10233)

[MVC的实现方式 4](#_Toc27073)

[如何使用controller[实现方式] 4](#_Toc12340)

[多个控制器间的关系 4](#_Toc5606)

[示例1 4](#_Toc25696)

[示例2 5](#_Toc14900)

[使用注意点 5](#_Toc29984)

[DC 5](#_Toc20113)

[scope中的Dirty Checking(脏数据检查) --- Digest Cycle 5](#_Toc9551)

[DC中的检查单元-Watcher 5](#_Toc13007)

[DC中判断数据是否变化的几种逻辑 5](#_Toc25682)

[DC的执行者$digest 6](#_Toc21555)

[Digest过程解析 6](#_Toc2)

[AngularJS的digest循环和$apply 6](#_Toc12337)

[DI 7](#_Toc21244)

[依赖注入基础概念 7](#_Toc9734)

[.module() 7](#_Toc14934)

[$provide 8](#_Toc22034)

[.injector() 8](#_Toc13828)

[.invoke() 9](#_Toc1321)

[控制器($controller)被注入 9](#_Toc29053)

[过滤器和指令 9](#_Toc27434)

[被注入 9](#_Toc7767)

[？如何配置provide 9](#_Toc20387)

[？ $rootScopeProvider <- $rootScope 10](#_Toc6389)

[创建服务的 3+2种方式 详解 10](#_Toc11342)

[provider 10](#_Toc20352)

[factory 10](#_Toc3237)

[service 10](#_Toc26043)

[ABC 10](#_Toc23261)

[value 11](#_Toc9337)

[constant 11](#_Toc3599)

[[decorator] 11](#_Toc31164)

[源码分析 11](#_Toc2650)

[实现案例 11](#_Toc14017)

[service/factory区别 11](#_Toc19254)

[JS依赖注入的简单实现（原理） 12](#_Toc25484)

[scope相关 13](#_Toc29018)

[angularJS中的$injector、$rootScope和$scope的概念和关联关系 13](#_Toc31662)

[scope的操作，如创建/注销/各种监听及scope间的通信等等 14](#_Toc22756)

[$new,$destroy,$watch使用 15](#_Toc18010)

[$on,$emit,$broadcast 15](#_Toc16801)

[angualr js中$parse/$eval和$observe/$watch如何区分? 15](#_Toc19461)

[$parse $eval 15](#_Toc8865)

[$observe $watch 15](#_Toc15908)

[使用场景 15](#_Toc19336)

[AngularJS 事件广播与接收 16](#_Toc2504)

[其他 16](#_Toc18737)

[使用angular.bootstrap完成模块的手动加载 16](#_Toc21543)

[AngularJS Promise Deferred实例 16](#_Toc28834)

[$compile 16](#_Toc22607)

[概念 16](#_Toc19038)

[+动态添加指令并编译 17](#_Toc6042)

[+使用$compile创建显示HTML的指令 17](#_Toc10654)

[$compile源码分析 17](#_Toc6810)

初始

学习网址

|  |  |
| --- | --- |
| www | angular基础入门概念解释（文章：前端网）：  <http://www.w3cfuns.com/notes/20332/7b34a55003dff88cd8d5f91b1ce3dada>  angular小知识点（博客：夜空中最亮的星星）  <http://www.cnblogs.com/lodingzone/tag/Angularjs/>  教程：  <http://www.ituring.com.cn/minibook/303>  <http://www.ituring.com.cn/article/13473>  <https://xdsnet.gitbooks.io/angular-phonecat-book-zhcn/content/index.html>  使用AngularJS  进行Hybrid App 开发已经有一年多时间了,这里做一个总结.  <http://www.cnblogs.com/hubcarl/p/4207909.html>  课程> 编程语言/框架> 其他> Learning AngularJS  <http://edu.csdn.net/course/detail/1857>  (十一)通过angularjs的ng-repeat指令看scope的继承关系  <http://blog.csdn.net/aitangyong/article/category/2602679>  AngularJS 开发者最常犯的 10 个错误  <http://www.oschina.net/translate/top-10-mistakes-angularjs-developers-make>  AngularJs 动态加载模块和依赖  <http://www.cnblogs.com/ys-ys/p/5119086.html>  开源中国-英文翻译  <http://www.oschina.net/translate/tag/angularjs>  segmentfault 提问  <https://segmentfault.com/search?q=angular+ui>  推酷  <http://www.tuicool.com/topics/11060094>  AngularJS 开发者最常犯的 10 个错误  <http://www.oschina.net/translate/top-10-mistakes-angularjs-developers-make>  我最喜欢的 20 个 Angular JS 开发工具  <http://www.tuicool.com/articles/niQ3i2a>  angular博客47  <http://www.cnblogs.com/whitewolf/category/404298.html>  让你的 AngularJS 应用动起来 ： ngView  <http://www.oschina.net/translate/animating-angularjs-apps-ngview>  Angular-UI-Router 学习笔记  <http://www.tuicool.com/articles/zeiy6ff>  angular好用的插件集合（持续更新中）  <https://segmentfault.com/a/1190000003858219>  Bootstrap components written in pure AngularJS by theAngularUI Team  <http://angular-ui.github.io/bootstrap/>  <http://angular-ui.github.io/bootstrap/versioned-docs/1.0.2/>  <http://angular-ui.github.io/>  <http://blog.csdn.net/zoutongyuan/article/details/39431299> |
|  | ng-template寄宿方式  <http://www.cnblogs.com/whitewolf/p/3601990.html>  csdn社区：  <http://lib.csdn.net/base/angularjs>  深入学习JavaScript的AngularJS框架中指令的使用方法  <http://www.jb51.net/article/80499.htm>  AngularJS开发指南12：作用域  <http://www.angularjs.cn/A00y>  [AngularJS面面观] 26  <http://blog.csdn.net/dm_vincent/article/category/2496175>  <http://blog.csdn.net/dm_vincent/article/category/2496171/2>  视屏教程：  <http://edu.csdn.net/course/detail/3181/53240?auto_start=1>  英文教程  <https://code.angularjs.org/1.0.2/docs/guide/overview>  源码分析  <http://www.ngnice.com/posts/1ba1b9b59cd3b5>  <http://www.cnblogs.com/web2-developer/category/751303.html>  zone.js  <http://www.tuicool.com/articles/jQfUJnu>  RESTful API 设计指南  <http://www.ruanyifeng.com/blog/2014/05/restful_api.html>  《基于React的前端工程实战》大纲，也是笔者心中的前端学习路线图  <https://segmentfault.com/a/1190000007730440>  # # # AngularJS PhoneCat代码分析---必学的工具  <http://www.cnblogs.com/mochaMM/p/5027107.html>  # # # angularjs 实战总结  <http://lib.csdn.net/article/angularjs/7215> |
|  | AngularJS按需动态加载template和controller?  <https://www.zhihu.com/question/30624377> |
| 大漠穷秋 | 专业AngularJS交流裙：286047042，口令：ng    关于AngularJS的其它相关文章：    1、《AngularJS》5个实例详解Directive（指令）机制    <http://damoqiongqiu.iteye.com/blog/1917971>    2、AngularJS表单基础    <http://damoqiongqiu.iteye.com/blog/1920191>    3、AngularJS Form 进阶：远程校验和自定义输入项    <http://damoqiongqiu.iteye.com/blog/1920993>    4、AngularJS：在Windows上安装Yeoman    <http://damoqiongqiu.iteye.com/blog/1885371>    5、对比Angular/jQueryUI/Extjs：没有一个框架是万能的    <http://damoqiongqiu.iteye.com/blog/1922004>    6、使用JsTestDriver实现JavaScript单元测试    <http://damoqiongqiu.iteye.com/blog/1924415>    7、JavaScript单元测试系列二：将Jasmine集成到JsTestDriver    <http://damoqiongqiu.iteye.com/blog/1925974> |
|  | <http://annn.me/webstorm/> |

第一步

|  |  |
| --- | --- |
| 简例1 | <**div** ng-init="name='zhongguo'">  <**span** ng-bind="name"></**span**> </**div**> |
| 简例2  $scope  -  input  -  {{ }}  ng-biWnd | ---定义的值赋值到input框中，再影响到显示  ---赋值通过，angualr.module("myapp",[]).controller("mycon",function($scope){})  **---中间：<input type="text" ng-model="name"/>** |
| ng-repeat | <**div** ng-repeat="x in name4.sss">{{$index+1}}---{{x.a}}</**div**> <**table**>  <**tr** ng-repeat="x in name4.sss">  <**td** ng-if="$odd" style="background:#red">{{ x.a }}</**td**>  <**td** ng-if="$even">{{ x.b }}</**td**>  </**tr**> </**table**>  ---x in arrName [$index|$even|$odd] |
| 表达式  过滤 | {{ (name0|lowercase)+"--"+(name1|uppercase)+"--"+(name1suzhi|currency) }}  ---注意：两个值的过滤要用()+""+()  <input ng-model="query"/>  <li nt-repeat="thing in things|filter:query">  ---关键字匹配实现方式 |
| 几个小问题 | 1. 关于页面数据展示方式"ng-bind"与"{{ }}"：   <**span** ng-bind="name"></**span**> <**span**>{{name}}</**span**> <**span** ng-bind="name">{{name}}</**span**>  ---三者的效果是一样的 |
| 1. 一个页面中只能存在一个np-app或者一个ng-controller，两者的存在关系：   [np-app="app"中若赋予了值，就要在js中体现出来,angular.module("app",[]).controller  ("con",function(){})]  [np-app="app" 与 angular.module().controller() 要同步，要么ng-app不进行赋值且下方不进行定义，或者ng-app=""进行赋值且下方进行定义]  [上方进行赋值，下方却不进行定义则有误] |
| 1. 页面值初始化方式 ng-init与 $scope的作用级别：   <**div** ng-app="app" ng-init="name='a'" ng-controller="con" >  {{name}} ---值为"a";说明ng-init的作用级别比较高，可以理解为就近原则 </**div**> <**script** type="text/javascript">  **angular**.module(**"app"**,[]).controller(**"con"**,**function**($scope){  $scope.**name**=**"b"**;  }) </**script**> |

一篇总结

|  |  |  |
| --- | --- | --- |
| 概念 | 传统前端开发思维  以JQuery为代表  以dom为中心  关注view层的变化和用户操作  [我有一个这个样的DOM，我想让他做什么] | 新一代前端开发思维  以angular为代表  以data为中心  关注数据的变更  MVM=model+view+whatever |
| 核心概念 | MVC：作为datamodel的$scope  依赖注入DI  模块化Module  Service  Filter  Two way DataBinding[双向数据绑定]  Directive  Unit Testing&2E Testing |  |
|  | “关于Angular 的起源，我可以追溯到2009 年的Google Feedback 项目。当时，对于项目的开发速度以及如何编写可测试代码的问题，我们已经经受了几个月的折磨。6 个月时，我们开发了差不多17000 行前端代码。这时候，团队中的一个成员Misko Hevery 做出了一个大胆的宣言：利用他自己业余时间所开发的一个开源库，他可以在两周之内把目前所有东西重写一遍。  我当时想，两周的时间并不会给我们造成太大的影响，同时我们也接受了Misko 努力构建一些东西的想法。然而Misko 最终还是估算错了时间，他用了三个星期。但是，我们所有人还是被他深深地震撼了，更让我们感到震撼的是，他所开发的新应用的代码量从原来的17000 行压缩到了1500 行。看起来，Misko 的东西值得深入推广。Misko 和我决定，围绕他所提倡的理念组建一个团队，这个简单的理念就是：简化对web 开发者的经验要求。” | |
| Angular特性 | 1、MVC(MVVM)模式：  MVC顾名思义：Model-view-Controll设计模式，其实这个有点类似于Java的思想，模型、视图、控制器的分离。但是Angular又有点不同于传统的MVC，严格意义上说它是MVVM（model-view-viewModel）。Model是简单的javascript对象；ViewModel是$scope对象，Angular依靠$scope检测状态变化；Controller负责设置初始状态和参数化$scope方法用以控制行为；View是Angular解析渲染后生成的HTML。   |  | | --- | | $scope是一个POJO(Plain Old JavaScript Object)  $scope提供了一些工具方法$watch()/$apply()  $scope是表达式的执行环境（或者叫作用域）  $scope是一个树型结构，与DOM标签平行  子$scope对象会继承父$scope上的属性和方法  每一个Angular应用只有一个根$scope对象（一般位于ng-app上）  $scope可以传播事件，类似DOM事件，可以向上也可以向下  $scope不仅是MVC的基础，也是后面实现双向数据绑定的基础  可以用angular.element($0).scope()进行调试 | | $scope充当MVC中的Data-Model角色  是一个POJO(plain old javascript object)  提供了一些工具方法： $watch() 、 $apply()  是表达式的执行环境，或者叫作用域  存在继承，子$scope对象会继承父$scope上的属性  每一个angular应用只有一个根$scope对象，一般位于ng-app  $scope可以传播事件，类似DOM事件，可以向上，亦可以向下 |     2、数据双向绑定：      这应该算Angular比较特有的属性，概括来说，Angular通过监听变量的变化来实现双向数据绑定。传统技术上，当model改变时，开发与人员需要手动处理DOM元素并将属性反映到变化中。而在Angular中，程序员可以轻轻松松解决这种问题，省掉了很多的代码。      {{}}与ng-bind指令：在脚本没有加载完成时，用户会看到{{}}，界面比较丑陋，一般的做法是在index.html里面使用ng-bind,其他动态的内容使用{{}}    3、指令：      指令比较有意思，可以帮助我们做很多的DOM操作，比如增删改查都可以用指令完成。指令自带的link方法有三个参数：scope、element、attrs。通过这三个参数我们可以完成很多功能，如：获取对象，改变CSS，变量传递等，这也是我用的比较多的方法。   1. 模板：      在AngularJS中，一个模板就是一个HTML文件。但是HTML的内容扩展了，包含了很多帮助你映射model到view的内容。                »  HTML模板将会被浏览器解析到DOM中。                »  DOM然后成为AngularJS编译器的输入。                »  AngularJS将会遍历DOM模板来生成一些指导，即，directive（指令）。                »  所有的指令都负责针对view来设置数据绑定。   5、服务和依赖注入：    Angular服务的作用就是对外提供某个特定的功能。Angular拥有内建的依赖注入（DI）子系统，可以帮助开发人员更容易开发，理解和测试应用。比如，很多时候我们会引入第三方指令，这时候需要在module中加入模块依赖，比如项目上我想用ng-bind-html，引入$sce，这时候需要在module写：  var app=angular.module('myapp',['ngSanitize']);  Angular的依赖注入实现：  每一个Angular应用都有一个injector  injector负责自动处理依赖关系、实例化对象  对用户代码来说，injector是透明的  injector会自动分析函数签名，注入所需要的对象  inJector注入有三种方式：http://www.cnblogs.com/lodingzone/p/4895275.html  依赖可以被使用者自己创建,通过new操作符.  依赖可以通过全局变量(如window)来查找并引用  依赖可以在需要的地方被传入  DI可以用在各种不同的地方，主要用在controller和factory中 | |
| ng模块 | [controllers][directives][services][routes][others]  任何一个ng应用都是由控制器、指令、服务、路由、过滤器等有限的模块类型构成的，用一个总的app模块作为入口点，它依赖其它所有模块。 | |
| $scope  作用域 | AngularJS 应用组成如下：  ---View(视图), 即 HTML；  ---Model(模型), 当前视图中可用的数据，$scope；  ---Controller(控制器), 即 JavaScript 函数，可以添加或修改属性；  scope 是模型,是一个 JavaScript 对象，带有属性和方法，这些属性和方法可以在视图和控制器中使用。  scope 是HTML (视图) 和 JavaScript (控制器)之间的纽带;当在控制器(controller)中添加 $scope 对象时，视图 (HTML) 可以通过{{name}}获取了这些属性。  **$scope 的使用贯穿整个 AngularJS App 应用,它与数据模型相关联,同时也是表达式执行的上下文.有了$scope 就在视图和控制器之间建立了一个通道,基于作用域视图在修改数据时会立刻更新 $scope,同样的$scope 发生改变时也会立刻重新渲染视图。**  有了 $scope 这样一个桥梁,应用的业务代码可以都在 controller 中,而数据都存放在controller 的$scope 中.    $rootScope  AngularJS 应用启动并生成视图时,会将根 ng-app 元素与 $rootScope 进行绑定.$rootScope 是所有$scope 的最上层对象,可以理解为一个 AngularJS 应用中得全局作用域对象,所以为它附加太多逻辑或者变量并不是一个好主意,和污染 Javascript 全局作用域是一样的.  所有的应用都有一个 $rootScope，它可以作用在 ng-app 指令包含的所有 HTML 元素中。  $rootScope 可作用于整个应用中。是各个 controller 中 scope 的桥梁。用 rootscope 定义的值，可以在各个 controller 中使用。  $scope 的作用  $scope 对象在 AngularJS 中充当数据模型的作用,也就是一般 MVC 框架中 Model 得角色.但又不完全与通常意义上的数据模型一样,因为 $scope 并不处理和操作数据,它只是建立了视图和 HTML 之间的桥梁,让视图和 Controller 之间可以友好的通讯.  再进一步系统的划分它的作用和功能:  1.提供了观察者可以监听数据模型的变化  2.可以将数据模型的变化通知给整个 App  3.可以进行嵌套,隔离业务功能和数据  4.给表达式提供上下文执行环境  在 Javascript 中创建一个新的执行上下文,实际就是用函数创建了一个新的本地上下文,在 AngularJS 中当为子 DOM 元素创建新的作用域时,其实就是为子 DOM 元素创建了一个新的执行上下文.  $scope 生命周期  AngularJS 中也有一个'事件'的概念,比如当一个绑定了 ng-model 的 input 值发生变化时,或者一个 ng-click 的 button 被点击时,AngularJS 的事件循环就会启动.事件循环是 AngularJS 中非常非常核心的一个概念,因为不是本文主旨所以不多说,感兴趣的可以自己看看资料.这里事件就在 AngularJS 执行上下文中处理,$scope 就会对定义的表达式求值.此时事件循环被启动, AngularJS 会监控应用程序内所有对象,脏值检查循环也会启动.  $scope 的生命周期有4个阶段:  1. 创建  控制器或者指令创建时, AngularJS 会使用 $injector 创建一个新的作用域,然后在控制器或指令运行时,将作用域传递进去.  2. 链接  AngularJS 启动后会将所有 $scope 对象附加或者说链接到视图上,所有创建 $scope 对象的函数也会被附加到视图上.这些作用域将会注册当 AngularJS 上下文发生变化时需要运行的函数.也就是 $watch 函数, AngularJS 通过这些函数或者何时开始事件循环.  3. 更新  一旦事件循环开始运行,就会开始执行自己的脏值检测.一旦检测到变化,就会触发 $scope 上指定的回调函数  4. 销毁  通常来讲如果一个 $scope 在视图中不再需要, AngularJS 会自己清理它.当然也可以通过 $destroy() 函数手动清理. | |
| 控制器 | ng-controller 指令定义了应用程序控制器，以控制 AngularJS 应用程序的数据。  控制器是 JavaScript 对象，由标准的 JavaScript 对象的构造函数创建。  ng-app 定义js应用程序  ng-controller 定义一个控制器  ng-controler="mycontrol" 其中mycontrol是一个js函数  使用$scope对象来调用控制器，$scope相当于作用域或者控制范围用来保存Module的对象 | |
| ng初始化加载流程 | 1、浏览器载入HTML，然后把它解析成DOM。 2、浏览器载入angular.js脚本。 3、AngularJS等到DOMContentLoaded事件触发。 4、AngularJS寻找ng-app指令，这个指令指示了应用的边界。 5、使用ng-app中指定的模块来配置注入器($injector)。 6、注入器($injector)是用来创建“编译服务($compile service)”和“根作用域($rootScope)”的。 7、编译服务($compile service)是用来编译DOM并把它链接到根作用域($rootScope)的。 8、ng-init指令将“World”赋给作用域里的name这个变量。 9、通过{{name}}的替换，整个表达式变成了“Hello World”。 | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |

## MVC的实现方式

|  |  |  |
| --- | --- | --- |
| 如何使用controller[实现方式] | | |
|  | angular中的MVC的实现全部都是借助于$scope(作用域)来实现的； | |
|  | 一个控制器控制多个视图 | 一个控制器控制一个视图 |
| angular的实现方式： | |
| 关于控制之间存在继承关系的实现方式：  将公共的部分抽成一个服务，来让控制器调用它，而不是让控制器去继承一个公共的控制器 | |
| 多个控制器间的关系  示例1 | **var mm** = **angular**.module(**"app"**, []);  <**div** ng-controller="con">  {{name}}  <**div** ng-controller="con1">{{name}}</**div**>  <**div** ng-controller="con2">{{name}}</**div**> </**div**> <**script** type="text/javascript">  **mm**.controller(**"con"**,**function**($scope){  $scope.**name**=**"con"** });  **function** con1($scope){  $scope.**name**=**"con1"**;  }  **function** con2($scope){  $scope.**name**=**"con2"**;  } </**script**> | |
|  | |
| 示例2 | <**div** ng-controller="con">  <**button** ng-click="test()">控制器</**button**>  <**div** ng-controller="con1">  <**button** ng-click="test1()">控制器1</**button**>  </**div**>  <**div** ng-controller="con2">  <**button** ng-click="test2()">控制器2</**button**>  </**div**> </**div**> <**script** type="text/javascript">  **mm**.controller(**"con"**,**function**($scope){  $scope.test=**function**(){  alert(0)  }  })  **function** con1($scope){  $scope.test1=**function**(){  alert(1)  }  }  **function** con2($scope){  $scope.test2=**function**(){  alert(2)  }  } </**script**> | |
| 其中“<**button** ng-click="test()">控制器</**button**>”在“con”中的任何位置都能实现功能，包括在“con1”、"con2"中；本质上时通过$scope的层级关系来实现的，在angular的语境中不推荐这种实现方式，而是希望能够将公共的东西抽成一个service。 | |
| 使用注意点 | 1. 不要试图去复用controller，一个控制器一般只负责一小块的视图 2. 不要在controller中区操作DOM，这不是控制器的职责[速度低] 3. 不要在controller中做数据格式化，ng有很好用的表单控件 4. 不要在controller中作数据过滤操作，ng有$filter服务 5. 一般来说，**controller是不会相互调用的**，控制器之间的交互应该通过$scope、数据模型上的事件进行交互;controller在内部来监控这些事件从而实现一些操作； | |

DC

[scope中的Dirty Checking(脏数据检查) --- Digest Cycle](http://blog.csdn.net/dm_vincent/article/details/51407292)

|  |  |
| --- | --- |
| DC中的检查单元-Watcher | 在angular中是通过Digest Cycle来完成脏数据检查从而完成双向绑定进而实现scope和view的同步的；脏数据检查(Dirty Checking)的机制实现了双向绑定。 |
|  | $watch: **function** (watchExp, listener, objectEquality, prettyPrintExpression) {  //调用了另一个服务$parse完成了从表达式到函数的转换  **var get** = $parse(watchExp);  //当编译得到的watch function中存在$$watchDelegate这个属性时，就会直接返回  **if** (**get**.$$watchDelegate) {  **return get**.$$watchDelegate(**this**, listener, objectEquality, **get**, watchExp);  }  **/\*  \* 声明一个数组变量来保存当前scope中的所有watchers;这个数组就是后面执行$digest时重点关注的对象;  \* 在创建watcher对象时，它有5个属性：  \* fn就是传入的listener，关于listener，它的形式是这样的：function(newVal, oldVal, scope)  \* 顾名思义在调用它的时候，会将当前的最新值，前值以及当前scope传入。  \* 但是listener也并不是必须的： if (!isFunction(listener)) { watcher.fn = noop; }  \* \*/  var** scope = **this**,  array = scope.$$watchers,  watcher = {  fn: listener,  last: initWatchVal,  **get**: **get**,  exp: prettyPrintExpression || watchExp,  eq: !!objectEquality  };   lastDirtyWatch = **null**;   **if** (!**isFunction**(listener)) {  watcher.fn = **noop**;  }   **if** (!array) {  array = scope.$$watchers = [];  }  // we use unshift since we use a while loop in $digest for speed.  // the while loop reads in reverse order.  array.unshift(watcher);  incrementWatchersCount(**this**, 1);   **/\*  \* 返回的这个函数是用来注销当前watcher：将当前watcher从数组中删除并减少计数器的值。  \* 以上就是DC中的检查单元：watcher。  \* 将不必要的复杂性剖离后，背后的逻辑也没有那么神秘。  \* \*/  return function deregisterWatch**() {  **if** (arrayRemove(array, watcher) >= 0) {  incrementWatchersCount(scope, -1);  }  lastDirtyWatch = **null**;  }; } |
| DC中判断数据是否变化的几种逻辑 | 声明好了watcher，那么在一轮DC中，是不是会将注册过的watcher中的listener统统调用一次呢？  很明显，答案是否定的。如果真要这么设计，任何程序员都会摇摇头：性能会有多差啊！  确实，不可能将所有watcher中的listener全部都调用一次。是否调用的关键就在于$watch中的第一个参数：watchExp。  只有在当前的值和上次检测的值不同时，才需要调用listener。基于这个逻辑，我们可以写下下面的示意代码：  forEach($$watchers, function(watcher) {  newVal = watcher.get(scope);  oldVal = watcher.last;  if (newVal !== oldVal) {  watcher.last = newValue;  watcher.listenerFn(newVal, oldVal, scope);  }  });  首先对watchExp求值得到当前值：newVal。  然后将该值和前值(oldVal)进行比较，如果不相等：  1. 将last设为当前值  2. 调用listener  从上面的逻辑中，我们能够发现几个问题：  1. 在一轮DC中，尽管每个watcher上的listener不一定会被调用，但是每个watcher上的watchExp是会被调用一次的。因此最为应用程序的开发者，对于它的性能我们需要做到心中有数，即在watchExp中不宜进行过于复杂的判断和操作。  2. 在比较当前值(newVal)和前值(oldVal)的时候，目前使用的是!==来进行比较。但是考虑下面的这种情况，又觉得有些不对：  [1, 2, 3] !== [1, 2, 3] // true  即使两个数组的元素一模一样，得到的判断仍然是：他俩不一样，需要调用listener！  好了，那么如何克服这个问题呢？即当元素内容一样的时候，不调用listener。这就涉及到了DC中判断数据是否变“脏”的几种逻辑：  1. 基于值的检查  2. 基于引用的检查  3. 基于集合的检查  对于第一种，基于值的检查。很好理解，就是我们在面对相同数组时需要的逻辑。如果数组元素相同，则判断它们是相同的。此时$watch方法的第三个参数eq就派上用场了，下面是文档中对它的解释：  @param {boolean=} [objectEquality=false] Compare for object equality using {@link angular.equals} instead of comparing for reference equality.  默认为false。使用基于引用的判断方式。也就是我们上面使用的!==。如果设置为true，那么就会使用angular.equals方法来进行比较。至于angular.equals这个方法，它定义在Angular.js这个文件中，其中定义的都是一些工具类方法。它会递归地对其中的所有属性进行比对。因此，只有两个对象的值一模一样时，才会判断它们是相同的。  那么第三种呢，它对应的方法是$watchCollection，现在介绍它还有点太早了，它的主要目的是对数组和对象的比较过程进行优化。在后续的文章中会进行介绍。 |
| DC的执行者$digest | 在上面执行DC的伪代码中，我们会调用watchExp，将当前的值和前值进行比对来确定是否需要调用listener。那么一个很直接的问题就是，第一次调用的时候，这个前值(oldVal)是如何确定的呢？  如果我们watch了scope上的一个并没有设置过的属性，那么在第一次DC的时候，该watch上的listener永远不会被执行。这是因为我们还没有设置watcher的last属性，没有设置的属性默认就是undefined，而undefined和undefined进行比较的时候，会返回true。因此，当我们需要在第一次DC的时候执行所有watcher的listener时，就必须对每个watcher设置last属性。  因此，也就有了代码中的这一段：  watcher = {  fn: listener,  last: initWatchVal,  get: get,  exp: prettyPrintExpression || watchExp,  eq: !!objectEquality  };  将last设置为了initWatchVal。  对代码进行搜索可以知道initWatchVal是这样定义的：  function initWatchVal() {}  比较巧妙的利用了[JavaScript](http://lib.csdn.net/base/javascript" \o "JavaScript知识库" \t "http://blog.csdn.net/dm_vincent/article/details/_blank)中引用相等判断(Reference Equality)规则。即只有该函数自身和自身进行比较的时候，才会相等。它和任何其它的值进行比较的时候，都会返回false。将它设置为一个watcher的last值再适合不过了。  另外一个需要注意的问题是，如果listener中对scope中的其它属性进行了修改，而恰巧该属性也有对应的watcher时，该如何是好？这样就造成了潜在的不一致，举个简单的例子。  现在我们有两个watcher A和B，分别针对属性a和属性b。针对a的watcher A首先执行，如果在B的listener中改变了属性a，那么由于A已经执行过了，就不会判断出a又变“脏”了的事实。这只是一种可能性，当watcher数量增多，各种各样的可能性都是存在的。因此，angular用最悲观的方式来看待这一问题：只要listener被执行了，就认为有可能存在有的数据又变“脏”了的情况。所以我们需要反复的执行DC，来确保再没有listener被执行。只有当一轮DC中一个listener都没有被执行时，DC才可以不用再被执行。  所以，DC并不是一蹴而就的过程。它需要反复确认所有的被检查值都“稳定”了，它才能休息。那么是不是意味着DC可能会执行非常多次，乃至于无限执行下去的情况呢？  答案显然是否定的。在实际开发过程中，我们应该遇到过这个异常：  10 $digest() iterations reached. Aborting!  它告诉我们DC似乎进入了一个无休止的循环状态中，因此为了不造成浏览器失去响应，angular主动放弃治疗！这个10是直接定义在angular的源码中的，当然我们也可以通过$rootScopeProvider来修改它：  this.digestTtl = function(value) {  if (arguments.length) {  TTL = value;  }  return TTL;  };  这里的TTL(可能是Time To Live的意思)就是DC的最大执行次数。  以上的逻辑反映到伪代码中是这样的：  var ttl = 10;  do {  var dirty = false;  var length = $$watchers.length;  var watcher;  for(var idx = 0; idx < length; idx++) {  watcher = $$watchers[idx];  newVal = watcher.watchFn(scope);  oldVal = watcher.last;  if (!newVal.equals(oldVal)) {  watcher.last = newVal;  watcher.listener(newVal, oldVal, scope);  dirty = true;  }  }  ttl -= 1;  if (dirty && ttl === 0) {  throw '10 $digest() iterations reached. Aborting!';  }  } while (dirty);  上面DC的逻辑离angular真正的实现还差的比较远。但是核心的逻辑已经初具雏形了。剩下的很多代码都是针对这个过程的各种优化措施。在下一篇文章中会对这些优化措施进行介绍。 |
| Digest过程解析 | <http://lib.csdn.net/article/angularjs/9446>  一、Digest 基本概念和原理  二、Digest 的应用场景 |
|  |  |

AngularJS的digest循环和$apply

|  |  |
| --- | --- |
|  | 一、传统事件触发  在标准的浏览器流程中，页面加载、$http请求返回响应、鼠标移动以及按钮被点击等情况都会触发事件。当事件被触发时（比如点击一个链接），JavaScript会创建一个事件对象，并执行这个事件对象所在的监听特定事件的所有函数。然后浏览器会执行注册给该事件的回调函数，更新DOM。  注意：同一时间不能运行两个事件。  当使用angular时，其会扩展这个标准的浏览器流程，创建一个angular上下文（angular事件循环内的特定代码，该angular事件循环通常被称为$digest循环）。  二、$digest循环  digest循环有两个主要部分组成：watch列表，$evalAsync列表。  1. $watch列表  angular跟踪变化，是通过给watch列表添加一个监控函数做到的，需要注意的是所有绑定给同一scope对象的UI元素，只会添加一个watch到watch列表中。这些watch列表会在digest循环中的“脏值检查”（检测值是否发生了变化，但整个应用还没有同步该变化）的程序解析。   （1）$watch(watchExpression, listener/callback, objectEquality)  （2）$watchCollection(obj/string, listener)  $scope.$watchCollection('names', function(newNames, oldNames, scope) {  // 发送变化的处理  });  2. $evalAsync列表  $evalAsync()方法是一种在当前作用域上调度表达式在未来某个时刻运行的方式。  指令、控制器调用$evalAsync()，会在angular操作DOM之后，浏览器渲染之前运行。所以，永远不要使用其来约定事件的顺序。  三、页面中的$digest循环  <input type="text" ng-model="user.name" ng-minlength="3" />  （1）angular会设置一个隐式的监控器，将输入字段的值绑定为当前的$scope对象；  （2）当用户输入字符，angular上下文就会生效并开始遍历$$watchers（$watch列表）；  （3）监控函数在$scope.user.name绑定上执行；  （4）退出$digest循环之前，会触发该值（ng-model）上运行的验证和格式化操作；  （5）由于在digest循环中值发生了变化，angular需要再次运行这一循环以确定它没有改变作用域对象上的其他值。（原因：如果有一个名为scope.user.fullName的属性由scope.user.firstName和$scope.user.lastName组成，那么这两个值的变化多会引起fullName的变化，因此需要再次确认）；  （6）$digest循环退出，浏览器重绘DOM以刷新视图。  四、$apply从外部进入上下文  所有指令ng-[event]指令（如ng-click）都会调用scope.apply()，以强制运行$digest循环。  apply()函数可以从angular框架的外部让表达式在angular上下文内部执行。当手动处理事件，使用第三框架（比如jquery）或者调用setTimeout都可以使用apply()函数将值传递到angular应用中。   1. 不建议在控制器中使用$apply()，因为这样会导致难以测试。   （2）jquery和angular同时使用被视为一个肮脏的行为。 |
|  | $apply会使ng进入$digest cycle, 并从$rootScope开始遍历(深度优先)检查数据变更。  $digest仅会检查该scope和它的子scope，当你确定当前操作仅影响它们时，用$digest可以稍微提升性能。  一些不必要的操作，放到$timeout里面延迟执行。  如果不涉及数据变更，还可以加上第三个参数false，避免调用$apply。  directive中执行的$evalAsync， 会在angular操作DOM之后，浏览器渲染之前执行。  controller中执行的$evalAsync， 会在angular操作DOM之前执行，一般不这么用。  而使用$timeout，会在浏览器渲染之后执行。  对时间有要求的，第二个参数可以设置为0。  $http.get('[http://path/to/url](http://path/to/url" \t "http://lib.csdn.net/article/angularjs/_blank)').success(function(data){    $scope.name = data.name;    $timeout(function(){      //do sth later, such as log    }, 0, false);  }); |

DI

依赖注入基础概念

|  |  |
| --- | --- |
|  | invoke:乞灵，祈求; 提出或授引…以支持或证明; 召鬼; 借助;  injector:注射器; 喷嘴;  provide:提供，供给，供应 ;规定;提供(+for);装备;预备 ;抚养，赡养(+for);做准备;预约 |
| [理解AngularJS中的依赖注入](http://sentsin.com/web/663.html)  <http://sentsin.com/web/663.html> |
| angularjs中与DI相关有[$provide](http://docs.angularjs.cn/api/auto/service/$provide" \t "http://blog.csdn.net/aitangyong/article/details/_blank)、angular.module()、angular.injector()、 [$injector](http://docs.angularjs.cn/api/auto/service/$injector" \t "http://blog.csdn.net/aitangyong/article/details/_blank)。  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。  在angular中module和injector，即依赖注入；  ---module和$provide相当于是服务的注册；  ---injector用来获取对象（angular会自动完成依赖的注入）； |
| 比如spring中，服务的注册是通过xml配置文件的<bean>标签或是注解@Repository、@Service、@Controller、@Component实现的；对象的获取可以ApplicationContext.getBean()实现；依赖关系的声明，即可以在xml文件中配置，也可以使用@Resource等注解在java代码中声明。在angular中，module和$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。下面从这3个方面，介绍下angular的DI。 |
| .module() | angular模块的声明与使用  （1）声明模块：  angular.module('MyApp', [])  ①第一个参数是模块的名称；  ②第二个参数是模块的依赖列表，是一个数组；依赖会在声明当前模块之前加载。  ③按照说明，相当于AngularJS模块的setter方法，用于定义模块的；创建一个新的module，覆盖已经存在的module。  // 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  var getModule = angular.module("myModule"); //如果模块不存在,抛出异常  alert(createModule == getModule); // true,都是同一个模块 |
| （2）引用模块：  angular.module('MyApp')  ①参数是模块的名称；指向已经存在的module  ②相当于AngularJS里的getter方法  ③然后可以通过在上面代码的返回的对象中，创建应用了。  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.  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。 |
|  | [Angular Module声明和获取重载](http://www.cnblogs.com/whitewolf/p/angular-module-declare-and-get.html)  <http://www.cnblogs.com/whitewolf/p/angular-module-declare-and-get.html>  module是angular中重要的模块组织方式，它提供了将一组内聚的业务组件（controller、service、filter、directive…）封装在一起的能力。这样做可以将代码按照业务领域问题分module的封装，然后利用module的依赖注入其关联的模块内容，使得我们能够更好的”分离关注点“，达到更好的”高内聚低耦合“。”高内聚低耦合“是来自面向对象设计原则。内聚是指模块或者对象内部的完整性，一组紧密联系的逻辑应该被封装在同一模块、对象等代码单元中，而不是分散在各处；耦合则指模块、对象等代码单元之间的依赖程度，如果一个模块的修改，会影响到另一个模块，则说明这两模块之间是相互依赖紧耦合的。  同时module也是我们angular代码的入口，首先需要声明module，然后才能定义angular中的其他组件元素，如controller、service、filter、directive、config代码块、run代码块等。  关于module的定义为：angular.module(‘com.ngbook.demo’, [])。关于module函数可以传递3个参数，它们分别为：  name：模块定义的名称，它应该是一个唯一的必选参数，它会在后边被其他模块注入或者是在ngAPP指令中声明应用程序主模块；  requires：模块的依赖，它是声明本模块需要依赖的其他模块的参数。特别注意：如果在这里没有声明模块的依赖，则我们是无法在模块中使用依赖模块的任何组件的；它是个可选参数。  configFn： 模块的启动配置函数，在angular config阶段会调用该函数，对模块中的组件进行实例化对象实例之前的特定配置，如我们常见的对$routeProvider配置应用程序的路由信息。它等同于”module.config“函数，建议用”module.config“函数替换它。这也是个可选参数。  对于angular.module方法，我们常用的方式有有种，分别为angular.module(‘com.ngbook.demo’, [可选依赖])和angular.module(‘com.ngbook.demo’)。请注意它是完全不同的方式，一个是声明创建module，而另外一个则是获取已经声明了的module。在应用程序中，对module的声明应该有且只有一次；对于获取module，则可以有多次。推荐将angular组件独立分离在不同的文件中，module文件中声明module，其他组件则引入module，需要注意的是在打包或者script方式引入的时候，我们需要首先加载module声明文件，然后才能加载其他组件模块。  在angular中文社区群中，有时会听见某些同学问关于”ng:areq“的错误：  [ng:areq] Argument 'DemoCtrl' is not a function, got undefined！  这往往是因为忘记定义controller或者是声明了多次module，多次声明module会导致前边的module定义信息被清空，所以程序就会找不到已定义的组件。这我们也能从angular源码中了解到（来自loader.js）：  function setupModuleLoader(window) {  ...  function ensure(obj, name, factory) {  return obj[name] || (obj[name] = factory());  }  var angular = ensure(window, 'angular', Object);  return ensure(angular, 'module', function() {  var modules = {};  return function module(name, requires, configFn) {  var assertNotHasOwnProperty = function(name, context) {  if (name === 'hasOwnProperty') {  throw ngMinErr('badname', 'hasOwnProperty is not a valid {0} name', context);  }  };  assertNotHasOwnProperty(name, 'module');  if (requires && modules.hasOwnProperty(name)) {  modules[name] = null;  }  return ensure(modules, name, function() {  if (!requires) {  throw $injectorMinErr('nomod', "Module '{0}' is not available! You either misspelled " +  "the module name or forgot to load it. If registering a module ensure that you " +  "specify the dependencies as the second argument.", name);  }  var invokeQueue = [];  var runBlocks = [];  var config = invokeLater('$injector', 'invoke');  var moduleInstance = {  \_invokeQueue: invokeQueue,  \_runBlocks: runBlocks,  requires: requires,  name: name,  provider: invokeLater('$provide', 'provider'),  factory: invokeLater('$provide', 'factory'),  service: invokeLater('$provide', 'service'),  value: invokeLater('$provide', 'value'),  constant: invokeLater('$provide', 'constant', 'unshift'),  animation: invokeLater('$animateProvider', 'register'),  filter: invokeLater('$filterProvider', 'register'),  controller: invokeLater('$controllerProvider', 'register'),  directive: invokeLater('$compileProvider', 'directive'),  config: config,  run: function(block) {  runBlocks.push(block);  return this;  }  };  if (configFn) {  config(configFn);  }  return moduleInstance;  function invokeLater(provider, method, insertMethod) {  return function() {  invokeQueue[insertMethod || 'push']([provider, method, arguments]);  return moduleInstance;  };  }  });  };  });  }  在代码中，我们能了解到angular在启动时，会设置全局的angular对象，然后在angular对象上发布module这个API。关于module API代码，能清晰的看见第一行谓语句，module名称不能以hasOwnProperty命名，否则会抛出”badname“的错误信息。紧接着，如果传入了name参数，其表示是声明module，则会删除已有的module信息，将其置为null。  从moduleInstance的定义，我们能够看出，angular.module为我们公开的API有：invokeQueue、runBlocks、requires、name、provider、factory、servic、value、constant、animation、filter、controller、directive、config、run。其中invokeQueue和runBlocks是按名约定的私有属性，请不要随意使用，其他API都是我们常用的angular组件定义方法，从invokeLater代码中能看到这类angular组件定义的返回依然是moduleInstance实例，这就形成了流畅API，推荐使用链式定义这些组件，而不是声明一个全局的module变量。  最后，如果传入了第三个参数configFn，则会将它配置到config信息中，当angular进入config阶段时，它们将会依次执行，进行对angular应用或者angular组件如service等的实例化前的配置。 |
| $provide | $provide服务负责告诉Angular如何创造一个新的可注入的东西：即服务(service)。  服务会被叫做provider的东西来定义，你可以使用$provide来创建一个provider。  你需要使用$provide中的provider方法来定义一个provider，同时你也可以通过要求改服务被注入到一个应用的config函数中来获得$provide服务。 |
|  | 服务的创建及两种写法  例子：  app.config(function($provide) {  $provide.provider('greeting', function() {  this.$get = function() {  return function(name) {  alert("Hello, " + name);  };  };  }); });  在上面的例子中我们为一个服务定义了一个叫做greeting的新provider；我么可以把一个叫做greeting的变量注入到任何可注入的函数中（例如控制器，在后面会讲到）然后Angular就会调用这个provider的$get函数来返回这个服务的一个实例。在上面的例子中，被注入的是一个函数，它接受一个叫做name的参数并且根据这个参数alert一条信息。我们可以像下面这样使用它：  app.controller('MainController', function($scope, greeting) {  $scope.onClick = function() {  greeting('Ford Prefect');  }; });  factory，service以及value全部都是用来定义一个providr的简写，它们提供了一种方式来定义一个provider而无需输入所有的复杂的代码。   |  | | --- | | 例如，你可以用下面的代码定义一个和前面完全相同的provider：  app.config(function($provide) {  $provide.factory('greeting', function() {  return function(name) {  alert("Hello, " + name);  };  }); });  这非常的重要，所以一定要记住：在幕后，AngularJS实际上是在调用前面出现的代码（就是$provide.provider的版本）。从字面上来说，这两种方法基本上没有差别。value方法也一样 – 如果我们需要从$get函数（也就是我们的factory函数）返回的值永远相同的话，我们可以使用value方法来编写更少的代码。例如如果我们的greeting服务总是返回相同的函数，我们可以使用value来进行定义：  app.config(function($provide) {  $provide.value('greeting', function(name) {  alert("Hello, " + name);  }); });  再一次申明，以上两种方式的效果完全一样 – 只不过是代码的量不同而已。 |   现在你可能已经注意到了我们使用的烦人的代码了。由于定义一个新的provider是如此的常用[app.config(function($provide){...})],AngularJS在模块对象上直接暴露了provider方法，以此来减少代码的输入量：  var myMod = angular.module('myModule', []);  myMod.provider("greeting", ...);  myMod.factory("greeting", ...);  myMod.value("greeting", ...);  上面的代码和前面[app.config(function($provide){...})]这样啰嗦的写法完全相同。 除了上面提到的可以注入的东西之外，还有一个constant方法。基本上，它和value的用法一致。我们会在后面来讨论两者的不同点。下面所有的代码所做的都是同一件事情：  myMod.provider('greeting', function() {  this.$get = function() {  return function(name) {  alert("Hello, " + name);  };  };  });  myMod.factory('greeting', function() {  return function(name) {  alert("Hello, " + name);  };  });  myMod.value('greeting', function(name) {  alert("Hello, " + name);  }); |
|  | $provide和模块的关系  The [$provide](http://docs.angularjs.cn/api/auto/service/$provide" \t "http://www.mamicode.com/_blank) service has a number of methods for registering components with the [$injector](http://docs.angularjs.cn/api/auto/service/$injector" \t "http://www.mamicode.com/_blank). Many of these functions are also exposed on [angular.Module](http://docs.angularjs.cn/api/ng/type/angular.Module" \t "http://www.mamicode.com/_blank).  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/$provide" \t "http://blog.csdn.net/aitangyong/article/details/_blank) 和 [$injector](https://code.angularjs.org/1.2.21/docs/api/auto/service/$injector" \t "http://blog.csdn.net/aitangyong/article/details/_blank)，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/$provide" \t "http://blog.csdn.net/aitangyong/article/details/_blank)。不知道这是为什么?一般来说也不需要显示使用这个服务，直接使用module中提供的API即可。  var injector = angular.injector();  alert(injector.has("$provide"));//false  alert(injector.has("$injector"));//true |
| .injector() | 注入器（injector）负责从我们通过$provide创建的服务中创建注入的实例。只要你编写了一个带有可注入性的参数，你都能看到注入器是如何运行的。每一个AngularJS应用都有唯一一个$injector，当应用启动的时候它被创造出来，你可以通过将$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，你可以动过调用get函数来获得任何一个已经被定义过的服务的实例。例如：  var greeting = $injector.get('greeting');  greeting('Ford Prefect'); |
|  | 注入器同样也负责将服务注入到函数中；例如，你可以魔法般的将服务注入到任何函数中，只要你使用了注入器的invoke方法：  var myFunction = function(greeting) {  greeting('Ford Prefect');  };  $injector.invoke(myFunction);  如果注入器只是创建一个服务的实例一次的话，那么它也没什么了不起的。它的厉害之处在于，他能够通过服务名称缓存从一个provider中返回的任何东西，当你下一次再使用这个服务时，你将会得到同一个对象。  因此，你可以通过调用$injector.invike将服务注入到任何函数中也是合情合理的了。包括：  -控制器定义函数  -指令定义函数  -过滤器定义函数  -provider中的$get方法（也就是factory函数）  由于constant和value总是返回一个静态值，它们不会通过注入器被调用，因此你不能在其中注入任何东西。 |
|  | 使用angular.injector();也能获取到注入器，但是没有和模块绑定。这种做法是没有意义的，相当于是你创建了一个空的DI容器，里面都没有服务别人怎么用呢。  正确的做法是，在创建注入器的时候，指定需要加载的模块。  // 创建myModule模块、注册服务  var myModule = angular.module('myModule', []);  myModule.service('myService', function() {              this.my = 0;  });    // 创建herModule模块、注册服务  var herModule = angular.module('herModule', []);  herModule.service('herService', function() {              this.her = 1;  });    // 加载了2个模块中的服务  var injector = angular.injector(["myModule","herModule"]);  alert(injector.get("myService").my);  alert(injector.get("herService").her);  如果加载了多个模块，那么通过返回的injector可以获取到多个模块下的服务。  这个例子中如果只加载了myMoudle，那么得到的injector就不能访问herMoudle下的服务。这里特别需要注意下：angular.injector()可以调用多次，每次都返回新建的injector对象。  var injector1 = angular.injector(["myModule","herModule"]);  var injector2 = angular.injector(["myModule","herModule"]);  alert(injector1 == injector2);//false |
| .invoke() | angular中三种声明依赖的方式 ：inference、annotation、inline方式。  // 创建myModule模块、注册服务  var myModule = angular.module('myModule', []);  myModule.service('myService', function() {              this.my = 0;  });  // 获取injector  var injector = angular.injector(["myModule"]);    // 第一种inference  injector.invoke(function(myService){alert(myService.my);});    // 第二种annotation  function explicit(serviceA) {alert(serviceA.my);};  explicit.$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。 |
| 控制器($controller)被注入 | 控制器函数是可以被注入的，但是控制器本身是不能被注入到任何东西里面去的。这是因为控制器不是通过provider创建的。然而，有一个内建的AngularJS服务叫做$controller，它负责设置你的控制器。当你调用myMod.controller(…)时，你实际上是访问了这个服务的provider，就像上面的例子一样。  例如，当你像下面一样定义了一个控制器时：  myMod.controller('MainController', function ($scope) {  // ... }); //你实际上做的是下面这件事： myMod.config(function ($controllerProvider) {  $controllerProvider.register('MainController', function ($scope) {  // ...  }); });  当Angular需要创建一个你的控制的实例时，它会使用$controller服务（它反过来会使用$injector来调用你的控制器以便它能够被注入依赖项）。 |
| 过滤器和指令  被注入 | filter和directive和controller的运行方式相同；filter会使用一个叫做$filter的服务以及它的provider $filterProvider，而directive使用一个叫做$compile的服务以及它的provider $compileProvidr。下面是相应的文档：  $filter: http://docs.angularjs.org/api/ng.$filter  $filterProvider: http://docs.angularjs.org/api/ng.$filterProvider  $compile: http://docs.angularjs.org/api/ng.$compile  $compileProvider: http://docs.angularjs.org/api/ng.$compileProvider  其中，myMod.filter和myMod.directive分别是这些服务的简写。 |
|  | 总结一下，任何能够被$injector.invoke调用的函数都是能被注入的。包括，但不限于下面的这些：  控制器  指令  factory  过滤器  provider中的$get函数  provider函数  服务  provider创建的新服务都可以用来注入。包括：  constant  factory  provider  service  value  另外，内建的服务$controller和$filter也可以被注入，同时你也可以使用这些服务来获得新的过滤器和控制器。 |
| 本文译自Understanding Dependency Injection，  原文地址https://github.com/angular/angular.js/wiki/Understanding-Dependency-Injection |
| ？如何配置provide | 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。在前面我们已经提到过当你通过provider（或者其他简写方法）创建一个服务时，你实际上创建了一个新的provider，它将定义你的服务如何被创建。我们没有提到的是，这些provider可以被注入到config函数中，你可以和它们进行一些交互。  首先，AngularJS分两个阶段运行你的应用 – config阶段和run阶段。  在config阶段是你设置任何的provider的阶段。它也是你设置任何的指令，控制器，过滤器以及其它东西的阶段。  在run阶段，AngularJS会编译你的DOM并启动你的应用。run方法初始化全局的数据 , 只对全局作用域起作用；如 $rootScope，局部的$scope不管用。  angular.module('myApp', []).run(['$rootScope', function ($rootScope) {  $rootScope.name = 'hello'; }]);  你可以在myMod.config和myMod.run中添加任何代码 – 这两个函数分别在两个阶段运行。正如我们看到的，这些函数都是可以被注入的 – 我们在第一个例子中注入了内建的$provide函数。然而，值得注意的是在config阶段，只有provider能被注入（只有两个例外是$provide和$injector)。  例如，下面的代码就是错误的写法：  myMod.config(function(greeting) {  //不会运行 -- greeting是一个服务的实例  //只有服务的provider能被注入到config中  });  但是你可以通过下面的方法注入provider：  myMod.config(function(greetingProvider) {  // 这下好了！  });  有一个例外：constant，由于它们不能被改变，因此它不能被注入到config中（这就是它和value之间的不同之处）。它们只能通过名字被获取。  无论何时你为一个服务定义了一个provider，这个provider的名字都是serviceProvider。在这里service是服务的名字。现在我们可以使用provider的力量来做一些更复杂的事情了！  myMod.provider('greeting', function() {  var text = 'Hello, ';  this.setText = function(value) {  text = value;  };  this.$get = function() {  return function(name) {  alert(text + name);  };  };  });  myMod.config(function(greetingProvider) {  greetingProvider.setText("Howdy there, ");  });  myMod.run(function(greeting) {  greeting('Ford Prefect');  });  现在我们在provider中拥有了一个叫做setText的函数，我们可以使用它来自定义我们alert的内容；我们可以再config中访问这个provider，调用setText方法并自定义我们的service。当我们最终运行我们的应用时，我们可以获取greeting服务，然后你会看到我们自定义的行为起作用了。 |
| ？ $rootScopeProvider <- $rootScope | [AngularJS Injector - Error: [$injector:unpr] Unknown provider: $rootScopeProvider <- $rootScope](http://stackoverflow.com/questions/21694615/angularjs-injector-error-injectorunpr-unknown-provider-rootscopeprovide)  May be it's trivial question, but for AngularJS newbie it's a matter ^\_^  What I'm trying to originally achieve is to make a dynamically inserted tag (by jQuery) with ng-click directive to work. I've searched and found that I've to get AngularJS Injector, then compile that code. So here it is the simplest form of the injector code which is NOT working for me, what's wrong with it?  Note #1: The dynamically inserted tag with ngDirective is done outside AngularJS scope.  angular.module('simpleExample', [])  .run(  [ '$rootScope',  function ($rootScope) {  $rootScope.test = "Test";  }]);  console.log(angular.injector(['simpleExample']));  // console.log(angular.injector(['simpleExample']).$compile('<a href="" ng-click="someFunctionOnRootScope()">Text</a>')); |
|  | The ng module  angular.bootstrap automatically adds the ng module to the dependencies when used (manually or with ngApp)  ---$rootScope / $compile services are part of the ng module.  ---You need to use injector.invoke if you want these services.  ---You should probably use angular in more traditional ways.  ---angular.module('simpleExample', ['ng']);  angular.injector(['simpleExample'])  .invoke(['$rootScope','$compile',  function($rootScope, $compile){  var elm = $compile('<a href="" ng-click="someFunctionOnRootScope()">Text</a>')($rootScope);  $rootScope.someFunctionOnRootScope = function(){  alert("Hello there!");  }  angular.element(document.body).append(elm);  }]); |

创建服务的 3+2种方式 详解

|  |  |
| --- | --- |
|  | <http://www.yiibai.com/angularjs/angularjs_dependency_injection.html>  <http://www.lxway.com/481248644.htm> |
| 依赖注入是一个在组件中给出的替代了硬的组件内的编码它们的依赖关系的软件设计模式。这减轻一个组成部分，从定位的依赖，依赖配置。这有助于使组件可重用，维护和测试。  AngularJS提供了一个至高无上的依赖注入机制。它提供了一个可注入彼此依赖下列核心组件。  在AngularJS应用中,factory()方法是用来注册服务的最常规方式,同时还有其他一些API可以在特定情况下帮助我们减少代码量。 共有5种方法用来创建服务:  provider() 提供者  factory() 工厂  service() 服务  value() 值  constant() 常值 |
| provider | 1. 简要说明   provider可以为应用提供通用的服务，形式可以是常量，也可以是对象。  比如我们在controller里常用的$http就是AngularJS框架提供的provider  myApp.controller(‘MainController', function($scope, $http) {  $http.get(…)  }  在上面的代码里，就可以直接使用$http包好的各种功能了～   1. 自定义一个provider   //定义: $provide.provider('age', {  start: 10,  $get: **function** () {  **return this**.start + 2;  } }); //或 $provide.provider('age', **function** () {  **this**.start = 10;  **this**.$get = **function** () {  **return this**.start + 2;  }; }); //调用: **app**.controller('MainCtrl', **function** ($scope, age) {  $scope.age = age; //12 });  provider的基本原则就是通过实现$get方法来在应用中注入单例，使用的时候拿到的age就是$get执行后的结果。 上面例子中的两种定义方法都可以～ |
| factory | 大哥provider每次出场太繁琐了，如果我就想定义个$get，没别的乱七八糟的呢？这时候该二哥factory出场了：直接第二个参数就是$get要对应的函数实现，代码简单了很多  **/\* \* module.factory("name",getFn) \* -name:需要注册的服务名 \* -getFn:这个函数会在AngularJS创建服务实例时被调用 \* \*/** angular.module('myApp').factory('myService', **function** () {  **return** {'name': 'angular'} }); **/\* \* 因为服务是单例对象,getFn在应用的生命周期内只会被调用一次。 \* 同其他AngularJS的服务 一样,在定义服务时,getFn可以接受一个包含可被注入对象的数组或函数。 \* getFn函数可以返回简单类型、函数乃至对象等任意类型的数据(同value()函数类似)。 \* \*/** angular.module('myApp').factory('githubService', ['$http', **function** ($http) {  **return** {  getUserEvents: **function** (username) {  // ...  }  }; }]); |
| service | 这时候我又来劲儿了，我不仅就想定义个$get，里面我还就返回个new出来的已有js类，三哥service闪亮登场～  **/\* \* 使用service()可以注册一个支持构造函数的服务,它允许我们为服务对象注册一个构造函数。 \* service()方法接受两个参数。 \* ---name(字符串) 要注册的服务名称。 \* ---constructor(函数)构造函数,我们调用它来实例化服务对象。 \* service()函数会在创建实例时通过new关键字来实例化服务对象。 \* \*/ var Person** = **function**($http) {  **this**.getName = **function**() {  **return** $http({ method: 'GET', url: '/api/user'});  }; }; angular.service('personService', **Person**); |
| ABC | //通过provider $provide.provider('myDate', {  $get: **function** () {  **return new Date**();  } }); //通过factory $provide.factory('myDate', **function** () {  **return new Date**(); }); //通过service $provide.service('myDate', **Date**); //调用: **app**.controller('MainCtrl', **function** ($scope, myDate) {  $scope.myDate = myDate; //current date }); |
| value  constant  [decorator] | 更直接的需求来了，我只想定义个$get，而且就返回个常量：这时候value和constant都可以做到  $provide.value('pageCount', 7);  $provide.constant('pageCount', 7); |
| 1. 两者的区别:   //区别一：value可以被修改，constant一旦声明无法被修改  //[decorator可以用来修改（修饰）已定义的provider们，除了constant] $provide.decorator('pageCount', **function** ($delegate) {  **return** $delegate + 1; });  //区别二：value不可在config里注入，constant可以；value不能在provider内访问，constant可以 myApp.config(**function** (pageCount) {  //可以得到constant定义的'pageCount' }); |
| **/\*值value是简单的JavaScript对象，它是用来将值传递过程中的配置相位控制器。\*/ var mainApp** = angular.module("mainApp", []); **mainApp**.value("defaultInput", 5); //inject the value in the controller using its name "defaultInput" **mainApp**.controller('CalcController', **function**($scope, CalcService, defaultInput) {  $scope.number = defaultInput;  $scope.result = CalcService.square($scope.number);  $scope.square = **function**() {  $scope.result = CalcService.square($scope.number);  } }); |
| **/\*常量constant用于通过配置相位值考虑事实，值不能使用期间的配置阶段被传递。\*/** mainApp.constant("configParam", "constant value"); |
| 源码分析 | **/\*通过底层实现代码看关系\*/ function provider**(name, provider\_) {  **if** (isFunction(provider\_)) {  provider\_ = providerInjector.instantiate(provider\_);  }  **if** (!provider\_.$get) {  **throw Error**('Provider ' + name + ' must define $get factory method.');  }  **return** providerCache[name + providerSuffix] = provider\_; } **function factory**(name, factoryFn) {  **return provider**(name, {$get: factoryFn}); } **function service**(name, constructor) {  **return factory**(name, ['$injector', **function** ($injector) {  **return** $injector.instantiate(constructor);  }]); } **function value**(name, value) {  **return factory**(name, valueFn(value)); } **function constant**(name, value) {  providerCache[name] = value;  instanceCache[name] = value; } **function decorator**(serviceName, decorFn) {  **var** origProvider = providerInjector.get(serviceName + providerSuffix),  orig$get = origProvider.$get;  origProvider.$get = **function** () {  **var** origInstance = instanceInjector.invoke(orig$get, origProvider);  **return** instanceInjector.invoke(decorFn, **null**, {$delegate: origInstance});  }; }  从上面的代码可以看出之前介绍的每种provider的特点，decorator比较特殊，除了constant，另外几个最终调用的都是provider。 |
| 实现案例 | <**html**> <**head**>  <**title**>AngularJS Dependency Injection</**title**> </**head**> <**body**> <**h2**>AngularJS Sample Application</**h2**>  <**div** ng-app="mainApp" ng-controller="CalcController">  <**p**>Enter a number: <**input** type="number" ng-model="number"/>  <**button** ng-click="square()">X<**sup**>2</**sup**></**button**>  <**p**>Result: {{result}}</**p**> </**div**> <**script** src="js/angular.min.js"></**script**> <**script**>   **var mainApp** = angular.module("mainApp", []);   **mainApp**.config(**function** ($provide) {  $provide.provider('MathService', **function** () {  **this**.$get = **function** () {  **var** factory = {};  factory.multiply = **function** (a, b) {  **return** a \* b;  }  **return** factory;  };  });  });   **mainApp**.value("defaultInput", 5);   **mainApp**.factory('MathService', **function** () {  **var** factory = {};  factory.multiply = **function** (a, b) {  **return** a \* b;  }  **return** factory;  });   **mainApp**.service('CalcService', **function** (MathService) {  **this**.square = **function** (a) {  **return** MathService.multiply(a, a);  }  });   **mainApp**.controller('CalcController', **function** ($scope, CalcService, defaultInput) {  $scope.number = defaultInput;  $scope.result = CalcService.square($scope.number);   $scope.square = **function** () {  $scope.result = CalcService.square($scope.number);  }  }); </**script**> </**body**> </**html**> |
|  |
| service/factory区别 | 几乎每一个刚接触AngularJS的开发者，都会对这两个东西产生困惑。 虽然它们（几乎）实现了同样的效果，但真的不是语法糖。  这里是它们在 AngularJS 源码中的定义:  function factory(name, factoryFn) { return provider(name, { $get: factoryFn }); }  function service(name, constructor) {      return factory(name, ['$injector', function($injector) {        return $injector.instantiate(constructor);  }]);  }  从源码上看显然 service 函数只是调用 factory 函数，然后 factory 函数再调用 provider 函数。事实上，value、constant和decorator 也是 AngularJS 提供的对 provider 的封装，但对它们使用场景不会有这种困惑，并且文档描述也非常清晰。  那么Service 仅仅是单纯的调用了一次 factory 函数吗？ 重点在 $injector.instantiate 中; 在这个函数里service会接收一个由$injector 使用new关键字去实例化的一个构造器对象。（原文：with in this function $injector creates a new instance of the service's constructor function.）  下面是完成同样功能的一个service和一个factory。  var app = angular.module('app',[]);  app.service('helloWorldService', function(){      this.hello = function() {          return "Hello World";  };  });  app.factory('helloWorldFactory', function(){      return {          hello: function() {              return "Hello World";          }  }  });  当 helloWorldService 或者 helloWorldFactory中的任何一个注入到controller里面, 他们都有一个返回字符串"Hello World"的名称为 hello方法。 这个service 的构造函数只在声明时被实例化一次，并且在这个 factory 对象每次被注入时各种互相引用， 但这个 factory还是只是被实例化了一次。 所有的 providers 都是单例的。  既然都完成同样的功能，为什么会有这两种格式存在？factory比service略微更灵活一些，因为它们可以使用new关键字返回函数（原文：Factories offer slightly more flexibility than services because they can return functions which can then be new'd）。 在其他地方，从面向对象编程的工厂模式来说。 一个factory可以是一个用于创建其他对象的对象。  app.factory('helloFactory', function() {      return function(name) {          this.name = name;            this.hello = function() {              return "Hello " + this.name;          };      };  });  这里有一个使用了前面提到的那个service和两个factory的controller 的例子。需要注意的是 helloFactory 返回的是一个函数，变量name的值是在对象使用new关键字的时候设置。  s  app.controller('helloCtrl', function($scope, helloWorldService, helloWorldFactory, helloFactory) {      init = function() {        helloWorldService.hello(); //'Hello World'        helloWorldFactory.hello(); //'Hello World'        new helloFactory('Readers').hello() //'Hello Readers'      }      init();  });  在刚入门时候最好只使用services.  Factory更加适用于当你在设计一个需要私有方法的类的时候使用：  app.factory('privateFactory', function(){      var privateFunc = function(name) {          return name.split("").reverse().join(""); //reverses the name      };        return {          hello: function(name){            return "Hello " + privateFunc(name);          }  };  });  在这个例子中privateFactory含有一个不能被外部访问的私有privateFunc函数。这种使用方式services也可以实现，但是使用Factory代码结构显得更加清晰。 |
|  |  |

JS依赖注入的简单实现（原理）

|  |  |
| --- | --- |
|  | http://lib.csdn.net/article/angularjs/32633 |
|  | 简化的实现原理  要实现注入，基本有三步：  ---得到模块的依赖项  ---查找依赖项所对应的对象  ---执行时注入 |
|  | 1. 得到模块的依赖项  javascript 实现DI的核心api是Function.prototype.toString,对一个函数执行toString，它会返回函数的源码字符串，这样我们就可以通过正则匹配的方式拿到这个函数的参数列表：  function extractArgs(fn) { //angular 这里还加了注释、箭头函数的处理  var args = fn.toString().match(/^[^\(]\*\(\s\*([^\)]\*)\)/m);  return args[1].split(',');  }  2. 查找依赖项所对应的对象  java与.net通过反射来获取依赖对象，js是动态语言，直接一个object[name]就可以直接拿到对象。所以只要用一个对象保存对象或函数列表就可以了  function createInjector(cache) {  this.cache = cache;  }  angular.module = function () {  modules = {};  injector = new createInjector(modules);  return {  injector: injector,  factory: function (name, fn) {  modules[name.trim()] = this.injector.invoke(fn);  return this;  }  }  };  3. 执行时注入  最后通过 fn.apply方法把执行上下文，和依赖列表传入函数并执行：  createInjector.prototype = {  invoke: function (fn, self) {  argsString = extractArgs(fn);  args = [];  argsString.forEach(function (val) {  args.push(this.cache[val.trim()]);  }, this);  return fn.apply(self, args);  }  };  简化的全部代码和执行效果见(可能需FQ)：[http://plnkr.co/edit/sJiIbzEXiqLLoQPeXBnR?p=preview](http://plnkr.co/edit/sJiIbzEXiqLLoQPeXBnR?p=preview" \t "http://lib.csdn.net/article/angularjs/_blank) 或查看[源码](https://github.com/etoah/Eg/blob/master/Angular/di.html" \t "http://lib.csdn.net/article/angularjs/_blank)  这里是简化的版本，实际angular的实现考虑了很多问题，如模块管理，延迟执行等  1. 得到模块的依赖项  var ARROW\_ARG = /^([^\(]+?)=>/;  var FN\_ARGS = /^[^\(]\*\(\s\*([^\)]\*)\)/m;  var STRIP\_COMMENTS = /((\/\/.\*$)|(\/\\*[\s\S]\*?\\*\/))/mg;  function extractArgs(fn) {  var fnText = fn.toString().replace(STRIP\_COMMENTS, ''),  args = fnText.match(ARROW\_ARG) || fnText.match(FN\_ARGS);  return args;  }  2. 查找依赖项所对应的对象  function getService(serviceName, caller) {  if (cache.hasOwnProperty(serviceName)) {  if (cache[serviceName] === INSTANTIATING) {  throw $injectorMinErr('cdep', 'Circular dependency found: {0}',  serviceName + ' <- ' + path.join(' <- '));  }  return cache[serviceName];  } else {  try {  path.unshift(serviceName);  cache[serviceName] = INSTANTIATING;  return cache[serviceName] = factory(serviceName, caller);  } catch (err) {  if (cache[serviceName] === INSTANTIATING) {  delete cache[serviceName];  }  throw err;  } finally {  path.shift();  }  }  }  3. 执行时注入  得到参数：  function injectionArgs(fn, locals, serviceName) {  var args = [],  $inject = createInjector.$$annotate(fn, strictDi, serviceName);  for (var i = 0, length = $inject.length; i < length; i++) {  var key = $inject[i];  if (typeof key !== 'string') {  throw $injectorMinErr('itkn',  'Incorrect injection token! Expected service name as string, got {0}', key);  }  args.push(locals && locals.hasOwnProperty(key) ? locals[key] :  getService(key, serviceName));  }  return args;  }  调用  function invoke(fn, self, locals, serviceName) {  if (typeof locals === 'string') {  serviceName = locals;  locals = null;  }  var args = injectionArgs(fn, locals, serviceName);  if (isArray(fn)) {  fn = fn[fn.length - 1];  }  if (!isClass(fn)) {  // http://jsperf.com/angularjs-invoke-apply-vs-switch  // #5388  return fn.apply(self, args);  } else {  args.unshift(null);  return new (Function.prototype.bind.apply(fn, args))();  }  }  4、angular模块管理，深坑  angular在每次应用启动时，初始化一个Injector实例：  var injector = createInjector(modules, config.strictDi);  由此代码可以看出对每一个Angular应用来说，无论是哪个模块，所有的"provider"都是存在相同的providerCache或cache中  所以会导致一个被誉为angular模块管理的坑王的问题：module 并没有什么命名空间的作用，当依赖名相同的时候，后面引用的会覆盖前面引用的模块。  具体的示例可以查看：  [http://plnkr.co/edit/TZ7hpMwuxk0surlcWDvU?p=preview](http://plnkr.co/edit/TZ7hpMwuxk0surlcWDvU?p=preview" \t "http://lib.csdn.net/article/angularjs/_blank) |
|  | 注：angular di用本文的调用方式压缩代码会出问题：可以用[g-annotate](https://www.npmjs.com/search?q=ng-annotate" \t "http://lib.csdn.net/article/angularjs/_blank)转为安全的调用方式。  到此angular di的实现原理已完成简单的介绍，angular用了项目中几乎不会用到的api：Function.prototype.toString 实现依赖注入，思路比较简单，但实际框架中考虑的问题较多，更加详细的实现可以直接看[angular的源码](https://github.com/angular/angular.js" \t "http://lib.csdn.net/article/angularjs/_blank)。  以后会逐步介绍angular其它原理。  转载时请注明源出处: [http://www.cnblogs.com/etoah/p/5460441.html](http://www.cnblogs.com/etoah/p/5460441.html" \t "http://lib.csdn.net/article/angularjs/_blank) |

scope相关

angularJS中的$injector、$rootScope和$scope的概念和关联关系

|  |  |
| --- | --- |
|  | <http://blog.csdn.net/aitangyong/article/details/40267583>  $injector、$rootScope和$scope是angularJS框架中比较重要的东西，理清它们之间的关系，对我们后续学习和理解angularJS框架都非常有用。  1、$injector其实是一个IOC容器，包含了很多服务(类似于spring框架中的bean)，其它代码能够通过 $injector.get("serviceName")的方式，从injector中获取所需要的服务。详情参考[这篇文章](http://blog.csdn.net/aitangyong/article/details/39937505" \t "http://blog.csdn.net/aitangyong/article/details/_blank)  2、scope是angularJS中的作用域(其实就是存储数据的地方)，很类似javascript的原型链。搜索的时候，优先找自己的scope，如果没有找到就沿着作用域链向上搜索，直至到达根作用域rootScope。  3、$rootScope是由angularJS加载模块的时候自动创建的，每个模块只会有1个rootScope。rootScope创建好会以服务的形式加入到$injector中。也就是说通过$injector.get("$rootScope");能够获取到某个模块的根作用域。更准确的来说，$rootScope是由angularJS的核心模块ng创建的。 |
|  | // 新建一个模块  var module = angular.module("app",[]);    // true说明$rootScope确实以服务的形式包含在模块的injector中  var hasNgInjector = angular.injector(['app','ng']);  console.log("has $rootScope=" + hasNgInjector.has("$rootScope"));//true    // 获取模块相应的injector对象,不获取ng模块中的服务  // 不依赖于ng模块,无法获取$rootScope服务  var noNgInjector = angular.injector(['app']);  console.log("no $rootScope=" + noNgInjector.has("$rootScope"));//false    // 获取angular核心的ng模块  var ngInjector = angular.injector(['ng']);  console.log("ng $rootScope=" + ngInjector.has("$rootScope"));//true |
| 上面的代码的确可以说明：$rootScope的确是由核心模块ng创建的，并以服务的形式存在于injector中。如果创建injector的时候，指定了ng模块，那么该injector中就会包含$rootScope服务；否则就不包含$rootScope。 |
|  | <**div** id="first" ng-controller="**FirstController**">               <**input** type="text" ng-model="name"><**br**>  {{**name**}}          </**div**>      <**script**>  **var module** = angular.module("app", []);  // 控制器里的$injector,是由angular框架自动创建的  **function FirstController**($scope, $injector, $rootScope) {  $rootScope.name = "aty";  }  //自己创建了个injector,依赖于app和ng模块  **var myInjector** = angular.injector(["app", "ng"]);  **var rootScope** = **myInjector**.get("$rootScope");  **alert**(**rootScope**.name);//udefined </**script**> |
| angular.injector()可以调用多次，每次都返回新建的injector对象。所以我们自己创建的myInjector和angular自动创建的$injector不是同一个对象，那么得到的rootScope也就不是同一个。更详细的可以看[另一篇文章](http://blog.csdn.net/aitangyong/article/details/39937505" \t "http://blog.csdn.net/aitangyong/article/details/_blank)中的angular.injector()章节。 |
|  | <**body** ng-app>    <**div** id="first" ng-controller="**FirstController**">  <**input** type="text" ng-model="name">  {{**name**}}    </**div**> </**body**> <**script**>  **function FirstController**($scope, $injector, $rootScope) {   console.log("scope parent :" + ($scope.$parent == $rootScope));// true   } </**script**> |
| ng-controller指令给所在的DOM元素创建了一个新的$scope对象,并作为rootScope的子作用域。$scope是由$rootScope创建的，$scope不会包含在$injector中。 |
|  | <body ng-app>   <div id="first" ng-controller="FirstController">   <input type="text" ng-model="name">   <br>   <div id="tips"></div>   </div>     <h2>outside of controller</h2>     <br>   <!--访问每一个应用(模块)的rootScope-->   {{$root.name}}   <div id="noControllerDiv"/>   <div ng-controller="SecondController"></div> </body> <script>  //记住rootScope，用来判断跨控制器是否相等  var first\_rootScope = null;  //记住scope，用来判断跨控制器是否相等  var first\_scope = null;  //记住injector，用来判断跨控制器是否相等  var first\_injectot = null;  // 第1个angular控制器  function FirstController($scope, $injector, $rootScope) {  $rootScope.name = "aty";  first\_rootScope = $rootScope;  first\_injectot = $injector;  first\_scope = $scope;  }  // 第2个angular控制器,主要是来测试跨controller时injector和scope的表现  function SecondController($scope, $injector, $rootScope) {  console.log("first\_rootScope==second\_rootScope:" + (first\_rootScope == $rootScope));//true  console.log("first\_injectot==second\_injector:" + (first\_injectot == $injector));//true  console.log("first\_scope==second\_scope:" + (first\_scope == $scope));//false  } </script> |
| ng-app定义了一个angular模块，每个模块只有一个$rootScope，只有一个$injector，但可以有多个$scope。  弄清了$injector、$rootScope和$scope这3者之间的关系，我们看下angular提供的2个API，一个是scope()，一个是injector()。使用angular.element()返回的DOM对象，都会包含这2个方法，用来获取与之关联的scope和injector。  由于每个模块的injector是唯一的，所以angular.element().injector()直接返回元素所在模块的injector。  angular.element().scope()可以获取到当前元素的scope或父scope。如果当前元素有scope，则返回自己的scope;如果没有则向父亲方向寻找,如果找不到返回rootScope。即返回作用域链上，距离该元素最近的scope。作用域是作为$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。 |
|  |  |
|  | <**body** ng-app>   <**div** id="first" ng-controller="**FirstController**">   <**input** type="text" ng-model="name">   <**br**>   <**div** id="tips"></**div**>   </**div**>     <**h2**>outside of controller</**h2**>     <**br**>   **<!--访问每一个应用(模块)的rootScope-->** {{$root.name}}   <**div** id="noControllerDiv"/>    </**body**> <**script**>  **function FirstController**($scope, $injector, $rootScope) {  //获取body对象   **var** domBody = document.getElementsByTagName('body')[0];   // 通过ng-app指令所在的DOM元素获取rootScope   **var** rtScope = angular.element(domBody).scope();   //当前元素没有新作用域,获取父作用域即rootScope   **var** noScope = angular.element("#noControllerDiv").scope();   // true   console.log("rtScope==noScope:" + (rtScope == noScope));   //ng-controller所在的元素,返回的scope   **var** scopeOnController = angular.element("#first").scope();   // ng-controller内部的元素返回所在的scope   **var** inController = angular.element("#tips").scope();   //true   console.log("scopeOnController==inController:" + (scopeOnController == inController));   //验证通过DOM获取的scope是否与注入的$scope和$rootScope一致   //true   console.log("result1:" + (rtScope == $rootScope));  //true   console.log("result2:" + (inController == $scope));   } </**script**> |

scope的操作，如创建/注销/各种监听及scope间的通信等等

|  |  |
| --- | --- |
|  | <http://www.cnblogs.com/ys-ys/p/4976128.html> |
| $rootScope.Scope  可以使用$injector通过$rootScope关键字检索的一个根作用域。  可以通过$new()方法创建子作用域。（大多子作用域是在HTML模板被执行编译时自动生成）  格式：$rootScope.Scope([Providers],[instanceCache])  [Providers]:当前作用域需要被提供的服务工厂地图。默认是ng。  [instanceCache]:为需要providers追加/重写的服务提供预实例化服务。 |
| $new(isolate);  创建一个新的子作用域。  父作用域将会广播$digest()和$digest()事件。作用域可以使用使用$destroy()从作用域的层级结构中移除。  $destroy()使其所需的范围和它的子作用域范围内永久地从父作用域分离从而停止参与模型变化检测和侦听通知调用。  isolate：boolean类型。如果值是true，那么这个scope不会从父scope继承原型。作用域是独立的，在这里不能看见父scope的属性。  当写小窗户组件的时候，这将是很实用的去防止不小心读取到其父级的状态。 |
| $watch(watchExpression,[listener],[objectEquality]);  注册一个监听器的回调函数，该函数在watchExpression变化的时候被执行。  watchExpressions表达式每次执行都会产生一次$digest()，并且返回一个将会被监听的值。（$digest()发现watchExpressions发生变化而执行多次，并且每次都是幂等的）  监听只有在当前的watchExpressions与之前的值不想等是被调用。变动是根据angular.equals函数判断的。需要保存对象比较后的值，也需要用到angular.copy。这也意味着看复杂的选项将不利于记忆和性能影响。  监听可能会改变模型，这可能会引发其他监听的变化。Ng会一直执行直到监听的值稳点。重播迭代极限是10，为了防止陷入无限循环的死锁。  watchExpressions：string或者function类型。每个$digest循环周期的表达式，返回值的变化会触发调用监听。  listener： watchexpression的返回值改变时发生回调。  objectEquality：使用angular.equals代替引用对象的相等性比较（对象的深度监听）。 |
| $watchGroup(watchExpressions,listener);  针对watchexpressions数组变量的$watch()。集合里的任何一个表达式变化都将引发监听的执行。  watchExpressions数组里的每一项都被标准的$watch()操作观察，并且审查每一次的$digest()去观察每一项是否变化。  当watchExpressions数组里的任何一项发生变化即执行。 |
| $watchCollection(obj,listener);  浅度的观察对象属性，并且在其变化时执行（对于数组，这意味着看数组项；对于对象，这意味着看属性）。如果检测到更改，则该侦听器将被触发。 |
| $digest();  处理所有的当前作用域和它的子作用域的监听。因为监听可能改变模型，所以$digest()会一直执行知道模型稳定。这意味这他可能进入无限循环。如果迭代次数超过10，这个函数将抛出“Maximum iteration limit exceeded”错误。 |
| $destroy();  从父域中删除当前的scope（及其所有的子scope）。删除意味着$digest()不再传播到目前作用域及其子作用域。删除也意味着目前的作用域符合垃圾集合的条件。 |
| $eval([expression],[locals]);  在当前作用域上执行表达式并返回结果。表达式的任何异常将传播（捕获）。在求Angular表达式的值的时候有用。 |
| $evalAsync([expression]);  在稍后的时间点上执行当前范围的表达式(异步)。 |
| $apply([exp]);  $apply() 用来在Angular框架外执行angular内部的表达式。（例如浏览器的DOM事件，setTimeout，XHR或第三方库）。 |
| $on(name,listener);  监听一个给定类型的事件。  name:监听的事件名。  listener:当事件发生时调用的函数。 |
| $emit(name,args);  向上级已注册的作用域传播指定的事件，直到根作用域。  name:发出的事件名称。  args:一个或多个可选参数，将传递到事件侦听器。 |
| $broadcast(name,args);  向下级已注册的作用域广播指定的事件。  name:发出的事件名称。  args:一个或多个可选参数，将传递到事件侦听器。 |
| $new,$destroy,$watch使用 | <**div** ng-app="Demo" ng-controller="testCtrl as ctrl">  <**input** ng-model="ctrl.text.words" />  <**input** ng-list="," ng-model="ctrl.list" /> </**div**> <**script**>  angular.module("Demo", [])  .run(["$rootScope",**rootScope**])  .controller("testCtrl",["$scope",**testCtrl**])  **function rootScope**($rootScope){  **var** rootScope = $rootScope;  **var** \_scope = rootScope.$new();  \_scope.value = "Hello World";  \_scope.$destroy();//$$destroyed:true  };  **function testCtrl**($scope){  **this**.text = { words:"Hello World",id:1};  $scope.$watch("ctrl.text",**function**(n,o){  console.log(n,o); // n 新值 o 旧值  },**true**);  **this**.list = ["a","b","c","d"];  $scope.$watchCollection("ctrl.list",**function**(n,o){  console.log(n,o); // n 新值 o 旧值  });  }; </**script**> |
| $on,$emit,$broadcast | <**div** ng-app="Demo">  <**div** ng-controller="testCtrl as ctrl">  {{ctrl.number||'Here where receive a number from childScope'}}  <**div** ng-controller="childCtrlOne as childOne">  <**input** type="button" ng-click="childOne.toFatherScope()" value="click me" />  </**div**>  <**div** ng-controller="childCtrlTwo as childTwo">  {{childTwo.number||'Here where receive a number from fatherScope'}}  </**div**>  </**div**> </**div**> <**script**>  angular.module("Demo", [])  .controller("testCtrl",["$scope",**testCtrl**])  .controller("childCtrlOne",["$scope",**childCtrlOne**])  .controller("childCtrlTwo",["$scope",**childCtrlTwo**])  **function testCtrl**($scope){  **var** vm = **this**;  $scope.$on("toFather",**function**(e,v){  vm.number = v;  $scope.$broadcast("toChild",v);  })  };  **function childCtrlOne**($scope){  **var** count = 0;  **this**.toFatherScope = **function**(){  count += 1;  $scope.$emit("toFather",count);  }  }  **function childCtrlTwo**($scope){  **var** vm = **this**;  $scope.$on("toChild",**function**(e,v){  vm.number = v;  })  } </**script**> |

scope作用域的 $destroy() 在指令中的使用

|  |  |
| --- | --- |
|  | mm.directive('myDirective', function () {  return {  scope: true,  link: function (scope, element, attrs) {  element.on('click', function () {  scope.$destroy();  element.remove();  });  }  }  })  //销毁需要注意scope的继承，只有当scope参数非false时才能去销毁 |

$parse/$eval和$observe/$watch如何区分?

|  |  |
| --- | --- |
|  | <http://www.w3cscript.com/Angular/2014-11-26/4.html> |
| 大家在看angular的时候，有时候偶尔会看到$parse,$eval和$observe,$watch这两对语法，随着深入使用angular，就不可避免使用到它。  文章从内部运行机制跟实际需求的角度来解释这两对语法的区别。 |
| $parse $eval | 首先，$parse跟$eval都是用来解析表达式的， 但是$parse是作为一个单独的服务存在的。$eval是作为scope的方法来使用的。  $parse典型的使用是放在设置字符串表达式映射在真实对象上的值。也可以从$parse上直接获取到表达式对应的值。  var getter = $parse('user.name');  var setter = getter.assign;  setter(scope, 'new name');  getter(context, locals) // 传入作用域，返回值  setter(scope,'new name') // 修改映射在scope上的属性的值为‘new value’  $eval 即scope.$eval，是执行当前作用域下的表达式，如：scope.$eval('a+b'); 而这个里的a,b是来自 scope = {a: 2, b:3};  看看源码它的实现是  $eval: function(expr, locals) {      return $parse(expr)(this, locals);  },  可以找到它也是基于$parse，不过它的参数已经被固定为this，就是当前的scope，所以$eval只是在$parse基础上的封装而已，是一种$parse快捷的API。 |
| $observe $watch | $observe和$watch都可以用来对属性进行监控的，先来看下$observe源码：  $observe: function(key, fn) {      var attrs = this,          $$observers = (attrs.$$observers || (attrs.$$observers = {})),          listeners = ($$observers[key] || ($$observers[key] = []));      listeners.push(fn);      $rootScope.$evalAsync(function() {        if (!listeners.$$inter) {         // no one registered attribute interpolation function, so lets call it manually          fn(attrs[key]);        }      });      return fn;  }  从这里可以看出来它是使用了$rootScope.$evalAsync()方法来监控的。什么是$evalAsync呢？是一个异步解析的操作，是在其他表达式都已经解析之后再解析，这样使它拥有了处理像插值字符串的机会。  $observe是属性对象上的方法，因此它是用来监控DOM属性上的值的变化，它仅用在指令内部，当你需要在指令内部监控包含有插值表达式的DOM属性的时候，就要用到这个方法，  比如，  attr1="Name:  然后在指令里 面：  attrs.$observe('attr1', ....),  但是假如你只用scope.$watch(attrs.attr1,...),这种情况下是无效的，  因为无法被解析，所以你得到的 是undefined,  在其他情况下用$watch。  $watch更复杂一点，它可以监视表达式，这个表达式可以是函数或者字符串，假如表达式是字符串的话，会 被封装成一个函数，然后在digest循环的时候被调用。 这个字符串表达式不能包含,$watch是一个scope对象上的方法，所以它可以在任何你可以访问 到作用域的地方被调用。比如，控制器中或者link函数中。因为字符串是被当做angular的表达式解析的，所以$watch经常被用在当你想要监控一 个模型或者作用域对象的时候，  比如:attr1="myModel.some\_prop",然后在控制器中或者link函数 中scope.$watch('myModel.some\_prop',...)或 者scope.$watch(attrs.attr1,...)或 者scope.$watch(attrs['attr1'],...)。假如你使 用attrs.$observe('attr1')你就只能得 到myModel.some\_prop字符串的值。  $observe和$watch都会在每个digest阶段被执行。  带有隔离作用域的指令的情况会复杂点。假如'@'语法被使用的时候，你可以使用$observe或 者$watch一个包含插值的DOM属性，之所以$watch在这里也能用，是因为'@'语法为我们 处理了，因此传给$watch是的时候是没有的字符串。这样子我们使用起来就更容易了。但是这种情况还是推荐使用$observe。  想要更好的理解，大家可以看这个例子[Plunker](http://plnkr.co/edit/HBha8sVdeCqhJtQghGxw?p=preview" \t "http://www.w3cscript.com/Angular/2014-11-26/_blank)。  注意到，当link函数被执行的时候，任何包含的DOM属性都还没被解析，所以此时假如你检查这种方式定义 的属性值的话，就是得到undefined, 唯一的方法可以得到内部的真实值的是使用$observe，或者在包含带有@语法的隔离作用域中使用$watch.因此获取这些属性的值是异步操作(因为 要内部的值是链接到其他地方，所以要等这些值稳定之后，才能得到属性的正确值，而在上文的源码中你可以看到使用 了$rootScope.$evalAsync())。  有时候我们并不需要$observe和$watch，比如你的属性一个boolen类型或者数字类型的值，只要解析他们一次就够了,你可以这么 做：attr1="22", 同时你的链接函数: var count = scope.$eval(attrs.attr1),假如只是包含字符串，attr1="my string",你就可以直接使用attrs.attr1而不需要 用$eval方法。 |
| 使用场景 | 想要在沒有isolate scope的directive中取出foo的属性值，存在以下几种情况：  1.foo属性值是固定的字串值，例如想要傳 class name，id 等。  <div foo="classname"></div>  因为这种情况是直接給定固定字串值，可以直接在 foo directive 中的 link function 直接取出属性值，attrs.foo。  2.foo 属性值是非字串的值，例如：boolean, number 等。  <div foo="true"></div>, <div foo="123"></div>  如果通过 attrs.foo 直接取值的话，就只能取到字符串值，但是我想要取值后就是正确的类型的話，可以使用 scope.$eval(attrs.foo)。  3.foo 属性是一個 scope property。  angular.controller('BarCtrl', function($scope) {  $scope.bar = {   test: ''  };  });  // html  <div ng-controller="BarCtrl">  <div foo="bar.test"></div>  </div>  如果想要在 foo directive 中去改变bar.test的值的話，可以使用$parse來取值，$parse(attrs.foo)，$parse后会一個 function，之後你可以通过这个function的assign property 去设置 bar.test 的值：  var model = $parse(attrs.foo);  model.assign(scope, 'Hello world');    // scope 为 link function 的 scope  4.foo 属性值是 interpolated attribute。  <div foo="2"></div>, <div foo=""></div>  如果直接在 link function 直接使用 attrs.foo 取值的話，得到的是 undefined，因为此时 interpolated attribute 还没被解析，所以我們可以通过 $observe 來取值，attrs.$observe。 |

AngularJS 事件广播与接收

|  |  |
| --- | --- |
|  | 发送消息： $scope.$emit(name, data) 或者 $scope.$broadcast(name, data);  接收消息： $scope.on(name,function(event,data){ });  区别： $emit 广播给父controller   $broadcast 广播给子controller  broadcast 是从发送者向他的子scope广播一个事件。  这里就是ParentController发送， ParentController 和 ChildController 会接受到, 而MainController是不会收到的  $emit 广播给父controller，父controller 是可以收到消息  $on 有两个参数function(event,msg)  第一个参数是事件对象，第二个参数是接收到消息信息 |
|  | 1. angularjs $broadcast $emit $on 事件触发controller间的值传递 |
|  | angular.module('onBroadcastEvent', ['ng'])  .controller('MainController', **function** ($scope) {  $scope.$on('To-MainController', **function** (event, msg) {  **console**.log('MainController received:'  + msg);  });  })  .controller('ParentController', **function** ($scope) {  $scope.click = **function** (msg) {  $scope.$emit('To-MainController', msg + ',from ParentController to MainController');  $scope.$broadcast('To-ChildController', msg + ',from ParentController to ChildController');  $scope.$broadcast('To-BrotherController', msg + ',from ParentController to BrotherController');  }  })  .controller('ChildController', **function** ($scope) {  $scope.$on('To-ChildController', **function** (event, msg) {  **console**.log('ChildController received:' + msg);  });  })  .controller('BrotherController', **function** ($scope) {  $scope.$on('To-BrotherController', **function** (event, msg) {  **console**.log('BrotherController received:' + msg);  });  }); |

其他

[使用angular.bootstrap完成模块的手动加载](http://blog.csdn.net/aitangyong/article/details/39698041)

|  |  |
| --- | --- |
|  | 之前我们看到使用ng-app指令，可以实现模块的自动加载。现在我们看下，angular中如何手动加载模块。需要使用到angular.bootstrap这个函数。 |
|  | <**body**>  <**div** id="div1" ng-controller="controller1">div1:{{**name**}}</**div**> <**div** id="div2" ng-controller="controller2">div2:{{**name**}}</**div**> </**body**> <**script**>   // 创建moudle1   **var rootMoudle** = angular.module('moudle1', []);  **rootMoudle**.controller("controller1", **function** ($scope) {  $scope.name = "aty"  });   // 创建moudle2   **var m2** = angular.module('moudle2', []);  **m2**.controller("controller2", **function** ($scope) {  $scope.name = "aty"  });    // 页面加载完成后,再加载模块   angular.element(document).ready(**function** () {  angular.**bootstrap**(document.getElementById("div1"), ["moudle1"]);  angular.**bootstrap**(document.getElementById("div2"), ["moudle2"]);  });  </**script**> |

AngularJS Promise Deferred实例

|  |  |
| --- | --- |
|  |  |
|  |  |
|  | 多个Promise实例：输出： ["111", "222"] 这个一个数组对象顺序与push的顺序一致  var app = angular.module('app', ['autocomplete']);  app.factory('Suggestion',  function($http, $q, $timeout){    var suggestion = new Object();    suggestion.getData = function(keyword) {      var deferred = $q.defer();      $http.get('http://codesearch.sinaapp.com/search.php',      }).success(function(data){        deferred.resolve(data);      });      return deferred.promise;    }    return suggestion;  });  app.controller('MySuggestionCtrl',  function($scope, $sce,Suggestion){    $scope.autoComplete = function(keyword){      if(keyword){          Suggestion.getData(keyword).then(function(data){          var dataList = data.split('|');          $scope.dataList = dataList;        });      }    }  }); |
|  | var data2="222";  var promises = [];  var deffered1  = $q.defer();  var deffered2  = $q.defer();  $timeout(function(){    deffered1.resolve(data1);  },2000);  $timeout(function(){    deffered2.resolve(data2);  },2000);  promises.push(deffered1.promise);  promises.push(deffered2.promise);   $q.all(promises).then(function(data){      console.log(data);  }); |
|  |  |
|  |  |
|  |  |
|  |  |

$compile

|  |  |
| --- | --- |
| 概念 | Compiler作为Angular的一个服务（Service），负责遍历DOM结构，寻找属性。编译过程分成两个阶段：  1. 编译（Compile）：遍历DOM节点树，收集所有directives。返回结果是一个链接函数（linking function）。  2. 链接（Link）：将directives绑定到一个作用域（scope）中，创建一个实况视图（live view）。在scope中的任何改变，将会在视图中得到体现（更新视图）；任何用户对模版的活动（改变），将会体现在scope model中（双向绑定）。这使得scope model能够反映正确的值。  一些directives，诸如ng-repeat，会为每一个在集合（collection）中的元素复制一次特定的元素（组合）。编译和链接两个阶段，使性能得以提升。因为克隆出来的模版(template)只需要编译一次，然后为每一个集合中的元素进行一次链接（类似模版缓存）。  Directive是一个行为，在编译过程中遇到特定的HTML结构时，它会被触发。Directives可以放置在元素的name、attribute、class甚至注释中。Directive只是一个当编译器在DOM中遇到时会执行的一个函数（function）。  许多模版系统，它们通常都通过模版字符串与数据进行连接，生成最终的HTML字符串，并将结果通过innerHTML属性写入某元素里。  IMG_256IMG_257  这意味着任何数据发生改变时，都需要重新将数据、模版合并成字符串，然后当作innerHTML写回对应元素中。这里存在一些问题：(这里直译实在没法懂..唯有YY)假设有这么一个场景，模版里包含输入框。用户对在输入框进行输入，模版同步更新。普通模版通过innerHTML、字符串与数据连接的方式更新视图，这样会打断用户的输入，体验不好。  Angular是与众不同的。Angular编译器（compiler）通过directives处理DOM，而不是通过处理字符串模版。处理结果是一个与scope model组合并生成实时模版的链接函数（linking function）。视图与scope model的绑定对我们来说是透明的。开发者无须为更新视图、model做任何动作。而且，因为没有使用innerHTML更新视图模版，所以用户输入不会被打断。此外，angular directives不仅可以绑定文本值，而且还可以是拥有行为的结构（behavioral constructs）。  Angular的这个处理方式，产生了一个稳定的DOM。这意味着在DOM元素的生命周期里，一直与某model的实例绑定着，这个关系不会发生改变。这也意味着代码可以保持对某DOM对象的引用，对其注册事件函数，并且这个引用不会被模版数据合并所销毁。 |
| +动态添加指令并编译 | 在directive的link中有一个$http请求，当请求完成后根据返回的值动态做element.append('......');这个操作，能显示没问题，可问题是我动态组的HTML里面有ng-click，发现显示出来的内容根本不执行ng-click绑定的方法！  这个是因为异步返回之后，指令的编译已经完成，因此需要动态编译。  使用$compile服务！ 在link方法中对返回后的html代码经过$compile服务处理后再插入：  var el=$compile("HTML代码")(scope);  element.append(el); |
| compile的启动关键代码:  初始化相关的依赖,然后执行全局编译,最后更新所有的$watch； |
| injector.**invoke**(['$rootScope', '$rootElement', '$compile', '$injector', '$animate',  **function** (scope, element, compile, injector, animate) {  scope.$apply(**function** () {  element.data('$injector', injector);  compile(element)(scope);  });  }] ); |
| 核心：compile(element)(scope);  1.compile(element) 收集完整个页面内的指令，然后返回publicLinkFn函数  2.执行publicLinkFn(scope) 此处的scope即为$rootScope |
| +使用$compile创建显示HTML的指令 | $compile 用于动态显示html内容：  <http://www.gsgundam.com/2014-12-13-angularjs-compile-to-show-dymanic-html-content/>  用$compile服务创建一个directive ‘compile’，这个complie会将传入的html字符串或者DOM转换为一个template，然后直接在html里调用compile即可。 |
| <div ng-controller="GreeterController">  <input ng-model="name"> <br>  <textarea ng-model="html"></textarea> <br>  <div compile="html"></div>  </div>  <script>  angular.module('compileExample', [], function($compileProvider) {  // configure new 'compile' directive by passing a directive  // factory function. The factory function injects the '$compile'  $compileProvider.directive('compile', function($compile) {  // directive factory creates a link function  return function(scope, element, attrs) {  scope.$watch(  function(scope) {  // watch the 'compile' expression for changes  return scope.$eval(attrs.compile);  },  function(value) {  // when the 'compile' expression changes  // assign it into the current DOM  element.html(value);  // compile the new DOM and link it to the current  // scope.  // NOTE: we only compile .childNodes so that  // we don't get into infinite loop compiling ourselves  $compile(element.contents())(scope);  }  );  };  });  })  .controller('GreeterController', ['$scope', function($scope) {  $scope.name = 'Angular';  $scope.html = 'Hello ';  }]);  </script> |
| $compile源码分析 | <http://www.jb51.net/article/78813.htm>  <https://segmentfault.com/a/1190000004373258?_ea=587358> |
| @(Angular)  $compile，在Angular中即“编译”服务，它涉及到Angular应用的“编译”和“链接”两个阶段，根据从DOM树遍历Angular的根节点（ng-app）和已构造完毕的 $rootScope对象，依次解析根节点后代，根据多种条件查找指令，并完成每个指令相关的操作（如指令的作用域，控制器绑定以及transclude等），最终返回每个指令的链接函数，并将所有指令的链接函数合成为一个处理后的链接函数，返回给Angluar的bootstrap模块，最终启动整个应用程序。  [TOC]  Angular的compileProvider  抛开Angular的MVVM实现方式不谈，Angular给前端带来了一个软件工程的理念－依赖注入DI。依赖注入从来只是后端领域的实现机制，尤其是javaEE的spring框架。采用依赖注入的好处就是无需开发者手动创建一个对象，这减少了开发者相关的维护操作，让开发者无需关注业务逻辑相关的对象操作。那么在前端领域呢，采用依赖注入有什么与之前的开发不一样的体验呢？  我认为，前端领域的依赖注入，则大大减少了命名空间的使用，如著名的YUI框架的命名空间引用方式，在极端情况下对象的引用可能会非常长。而采用注入的方式，则消耗的仅仅是一个局部变量，好处自然可见。而且开发者仅仅需要相关的“服务”对象的名称，而不需要知道该服务的具体引用方式，这样开发者就完全集中在了对象的接口引用上，专注于业务逻辑的开发，避免了反复的查找相关的文档。  前面废话一大堆，主要还是为后面的介绍做铺垫。在Angular中，依赖注入对象的方式依赖与该对象的Provider，正如小结标题的compileProvider一样，该对象提供了compile服务，可通过injector.invoke(compileProvider.$get,compileProvider)函数完成compile服务的获取。因此，问题转移到分析compileProvider.$get的具体实现上。  compileProvider.$get  this.$get = ['$injector', '$parse', '$controller', '$rootScope', '$http', '$interpolate',  function($injector, $parse, $controller, $rootScope, $http, $interpolate) {  ...  return compile;  }  上述代码采用了依赖注入的方式注入了injector,parse,controller,rootScope,http,interpolate五个服务，分别用于实现“依赖注入的注入器（injector），js代码解析器（parse），控制器服务（controller）,根作用域（rootScope）,服务和指令解析服务（http）”。  compileProvider通过这几个服务单例，完成了从抽象语法树的解析到DOM树构建，作用域绑定并最终返回合成的链接函数，实现了Angular应用的开启。  get方法最终返回compile函数，compile函数就是compile服务的具体实现。下面我们深入compile函数：  function compile($compileNodes, maxPriority) {  var compositeLinkFn = compileNodes($compileNodes, maxPriority);  return function publicLinkFn(scope, cloneAttachFn, options) {  options = options || {};  var parentBoundTranscludeFn = options.parentBoundTranscludeFn;  var transcludeControllers = options.transcludeControllers;  if (parentBoundTranscludeFn && parentBoundTranscludeFn.$$boundTransclude) {  parentBoundTranscludeFn = parentBoundTranscludeFn.$$boundTransclude;  }  var $linkNodes;  if (cloneAttachFn) {  $linkNodes = $compileNodes.clone();  cloneAttachFn($linkNodes, scope);  } else {  $linkNodes = $compileNodes;  }  \_.forEach(transcludeControllers, function(controller, name) {  $linkNodes.data('$' + name + 'Controller', controller.instance);  });  $linkNodes.data('$scope', scope);  compositeLinkFn(scope, $linkNodes, parentBoundTranscludeFn);  return $linkNodes;  };  }  首先，通过compileNodes函数，针对所需要遍历的根节点开始，完成指令的解析，并生成合成之后的链接函数，返回一个publicLinkFn函数，该函数完成根节点与根作用域的绑定，并在根节点缓存指令的控制器实例，最终执行合成链接函数。  合成链接函数的生成  通过上一小结，可以看出$compile服务的核心在于compileNodes函数的执行及其返回的合成链接函数的执行。下面，我们深入到compileNodes的具体逻辑中去：  function compileNodes($compileNodes, maxPriority) {  var linkFns = [];  \_.times($compileNodes.length, function(i) {  var attrs = new Attributes($($compileNodes[i]));  var directives = collectDirectives($compileNodes[i], attrs, maxPriority);  var nodeLinkFn;  if (directives.length) {  nodeLinkFn = applyDirectivesToNode(directives, $compileNodes[i], attrs);  }  var childLinkFn;  if ((!nodeLinkFn || !nodeLinkFn.terminal) &&  $compileNodes[i].childNodes && $compileNodes[i].childNodes.length) {  childLinkFn = compileNodes($compileNodes[i].childNodes);  }  if (nodeLinkFn && nodeLinkFn.scope) {  attrs.$$element.addClass('ng-scope');  }  if (nodeLinkFn || childLinkFn) {  linkFns.push({  nodeLinkFn: nodeLinkFn,  childLinkFn: childLinkFn,  idx: i  });  }  });  // 执行指令的链接函数  function compositeLinkFn(scope, linkNodes, parentBoundTranscludeFn) {  var stableNodeList = [];  \_.forEach(linkFns, function(linkFn) {  var nodeIdx = linkFn.idx;  stableNodeList[linkFn.idx] = linkNodes[linkFn.idx];  });  \_.forEach(linkFns, function(linkFn) {  var node = stableNodeList[linkFn.idx];  if (linkFn.nodeLinkFn) {  var childScope;  if (linkFn.nodeLinkFn.scope) {  childScope = scope.$new();  $(node).data('$scope', childScope);  } else {  childScope = scope;  }  var boundTranscludeFn;  if (linkFn.nodeLinkFn.transcludeOnThisElement) {  boundTranscludeFn = function(transcludedScope, cloneAttachFn, transcludeControllers, containingScope) {  if (!transcludedScope) {  transcludedScope = scope.$new(false, containingScope);  }  var didTransclude = linkFn.nodeLinkFn.transclude(transcludedScope, cloneAttachFn, {  transcludeControllers: transcludeControllers,  parentBoundTranscludeFn: parentBoundTranscludeFn  });  if (didTransclude.length === 0 && parentBoundTranscludeFn) {  didTransclude = parentBoundTranscludeFn(transcludedScope, cloneAttachFn);  }  return didTransclude;  };  } else if (parentBoundTranscludeFn) {  boundTranscludeFn = parentBoundTranscludeFn;  }  linkFn.nodeLinkFn(  linkFn.childLinkFn,  childScope,  node,  boundTranscludeFn  );  } else {  linkFn.childLinkFn(  scope,  node.childNodes,  parentBoundTranscludeFn  );  }  });  }  return compositeLinkFn;  }  代码有些长，我们一点一点分析。  首先，linkFns数组用于存储每个DOM节点上所有指令的处理后的链接函数和子节点上所有指令的处理后的链接函数，具体使用递归的方式实现。随后，在返回的compositeLinkFn中，则是遍历linkFns，针对每个链接函数，创建起对应的作用域对象（针对创建隔离作用域的指令，创建隔离作用域对象，并保存在节点的缓存中），并处理指令是否设置了transclude属性，生成相关的transclude处理函数，最终执行链接函数；如果当前指令并没有链接函数，则调用其子元素的链接函数，完成当前元素的处理。  在具体的实现中，通过collectDirectives函数完成所有节点的指令扫描。它会根据节点的类型（元素节点，注释节点和文本节点）分别按特定规则处理，对于元素节点，默认存储当前元素的标签名为一个指令，同时扫描元素的属性和CSS class名，判断是否满足指令定义。  紧接着，执行applyDirectivesToNode函数，执行指令相关操作，并返回处理后的链接函数。由此可见，applyDirectivesToNode则是&dollar;compile服务的核心，重中之重！  applyDirectivesToNode函数  applyDirectivesToNode函数过于复杂，因此只通过简单代码说明问题。上文也提到，在该函数中执行用户定义指令的相关操作。  首先则是初始化相关属性，通过遍历节点的所有指令，针对每个指令，依次判断$$start属性，优先级，隔离作用域，控制器，transclude属性判断并编译其模板，构建元素的DOM结构，最终执行用户定义的compile函数，将生成的链接函数添加到preLinkFns和postLinkFns数组中，最终根据指令的terminal属性判断是否递归其子元素指令，完成相同的操作。  其中，针对指令的transclude处理则需特殊说明：  if (directive.transclude === 'element') {  hasElementTranscludeDirective = true;  var $originalCompileNode = $compileNode;  $compileNode = attrs.$$element = $(document.createComment(' ' + directive.name + ': ' + attrs[directive.name] + ' '));  $originalCompileNode.replaceWith($compileNode);  terminalPriority = directive.priority;  childTranscludeFn = compile($originalCompileNode, terminalPriority);  } else {  var $transcludedNodes = $compileNode.clone().contents();  childTranscludeFn = compile($transcludedNodes);  $compileNode.empty();  }  如果指令的transclude属性设置为字符串“element”时，则会用注释comment替换当前元素节点，再重新编译原先的DOM节点，而如果transclude设置为默认的true时，则会继续编译其子节点，并通过transcludeFn传递编译后的DOM对象，完成用户自定义的DOM处理。  在返回的nodeLinkFn中，根据用户指令的定义，如果指令带有隔离作用域，则创建一个隔离作用域，并在当前的dom节点上绑定ng-isolate-scope类名，同时将隔离作用域缓存到dom节点上；  接下来，如果dom节点上某个指令定义了控制器，则会调用&dollar;cotroller服务，通过依赖注入的方式（&dollar;injector.invoke）获取该控制器的实例，并缓存该控制器实例；随后，调用initializeDirectiveBindings，完成隔离作用域属性的单向绑定（@），双向绑定（＝）和函数的引用（&），针对隔离作用域的双向绑定模式（＝）的实现，则是通过自定义的编译器完成简单Angular语法的编译，在指定作用域下获取表达式（标示符）的值，保存为lastValue，并通过设置parentValueFunction添加到当前作用域的$watch数组中，每次&dollar;digest循环，判断双向绑定的属性是否变脏（dirty），完成值的同步。  最后，根据applyDirectivesToNode第一步的初始化操作，将遍历执行指令compile函数返回的链接函数构造出成的preLinkFns和postLinkFns数组，依次执行，如下所示：  \_.forEach(preLinkFns, function(linkFn) {  linkFn(  linkFn.isolateScope ? isolateScope : scope,  $element,  attrs,  linkFn.require && getControllers(linkFn.require, $element),  scopeBoundTranscludeFn  );  });  if (childLinkFn) {  var scopeToChild = scope;  if (newIsolateScopeDirective && newIsolateScopeDirective.template) {  scopeToChild = isolateScope;  }  childLinkFn(scopeToChild, linkNode.childNodes, boundTranscludeFn);  }  \_.forEachRight(postLinkFns, function(linkFn) {  linkFn(  linkFn.isolateScope ? isolateScope : scope,  $element,  attrs,  linkFn.require && getControllers(linkFn.require, $element),  scopeBoundTranscludeFn  );  });  可以看出，首先执行preLinkFns的函数；紧接着遍历子节点的链接函数，并执行；最后执行postLinkFns的函数，完成当前dom元素的链接函数的执行。指令的compile函数默认返回postLink函数，可以通过compile函数返回一个包含preLink和postLink函数的对象设置preLinkFns和postLinkFns数组，如在preLink针对子元素进行DOM操作，效率会远远高于在postLink中执行，原因在于preLink函数执行时并未构建子元素的DOM，在当子元素是个拥有多个项的li时尤为明显。  end of compile－publicLinkFn  终于，到了快结束的阶段了。通过compileNodes返回从根节点（ng-app所在节点）开始的所有指令的最终合成链接函数，最终在publicLinkFn函数中执行。在publicLinkFn中，完成根节点与根作用域的绑定，并在根节点缓存指令的控制器实例，最终执行合成链接函数，完成了Angular最重要的编译，链接两个阶段，从而开始了真正意义上的双向绑定。 |