--

[filter 1](#_Toc5982)

[angular controller as syntax vs scope ---as 语法糖 1](#_Toc26443)

[说明 1](#_Toc12765)

[demo 1](#_Toc18888)

[angular.element方法汇总 1](#_Toc26386)

[AngularJS常用服务（$http、$location、$sce等） 2](#_Toc15430)

[UI闪烁消除 2](#_Toc24797)

[依赖注入使用在 js压缩 2](#_Toc17454)

[AngularJs 模块化编程实例 2](#_Toc11758)

[模块化写法 3](#_Toc24778)

[样例 3](#_Toc13903)

[编程实例 4](#_Toc28707)

[编程实例 4](#_Toc17773)

[在AngularJS框架中处理数据建模(服务应用实例) 4](#_Toc26479)

filter

|  |  |
| --- | --- |
|  | <http://www.cnblogs.com/wushangjue/p/4516107.html?utm_source=tuicool&utm_medium=referral> |
|  | <https://segmentfault.com/a/1190000002758481> |
|  |  |
|  |  |

[angular controller as syntax vs scope](http://blog.csdn.net/benweizhu/article/details/49633937) ---as 语法糖

|  |  |
| --- | --- |
|  | <http://www.tuicool.com/articles/Mfeimu>  <http://blog.csdn.net/benweizhu/article/details/49633937>  <http://lib.csdn.net/article/angularjs/42911> |
| 说明 | AngularJS: "Controller as" or "$scope"?  angular从1.2版本开始带来了新语法Controller as。再次之前我们对于angular在view上的绑定都必须使用直接的scope对象，对于controller来说我们也得必须注入$scope这个service。如下： |
| angular.module("app",[]).controller("demoController",["$scope",function($scope){      $scope.title = "angualr";   }])  <div ng-app="app" ng-controller="demoController">      hello : {{title}} !  </div> |
| 有些人觉得即使这样我们的controller还是不够POJO，以及对于coffescript爱好者不足够友好，所以在angular在1.2给我带来了一个新的语法糖这就是本文将要说的controller as的语法糖，修改上面的demo将会变成： |
| angular.module("app",[]).controller("demoController",[function(){      this.title = "angualr";  }])  <div ng-app="app" ng-controller="demoController as demo">       hello : {{demo.title}} !  </div> |
| 这里我们可以看见现在controller不再有$scope的注入了，感觉controller就是一个很简单的平面的JavaScript对象了，不存在任何的差别了。再则就是view上多增加了个demoController as demo，给controller起了一个别名，在此后的view模板中靠这个别名来访问数据对象。  或许看到这里你会问为什么需要如此啊，不就是个语法糖而已，先别着急，我们会在后边分析$scope和他的差别。在此之前我们先来看看angular源码的实现这样才会有助于我们的分析：  下面是一段来自angular的code：在1499行开始(行数只能保证在写作的时候有效) |
| if (directive.controllerAs) {           locals.$scope[directive.controllerAs] = controllerInstance;     } |
| <div ng-controller="MainController as main">  {{ main.someProp }}  </div>  app.controller('MainController', function () {  this.someProp = 'Some value.'  }); |
| 从上面的代码我们能看见的是：angular只是把controller这个对象实例以其as的别名在scope上创建了一个新的对象属性。靠，就这么一行代码搞定！     先别急，既然是语法糖，那么它肯定有他出现的原因，让我们来和直接用$scope对比下：     在此文之前我在angularjs的群中和大家讨论了下我的看法，得到大家不错的反馈，所以有了本文，记录和分享下来。     可以规定对于controller as的写法如下： |
| angular.module("app",[]) .controller("demoController",[function(){          var vm = this;          vm.title = "angualr";          return vm;   }]) |
|  |
| 其优势为：  定义vm这样会让我们更好的避免JavaScript的this的坑。  如果某个版本的angular不再支持controller as,可以轻易的注入$scope,修改为 var vm = $scope;  因为不再注入$scope了，controller更加的POJO，就是一个很普通的JavaScript对象。  也因为没有了$scope，而controller实例将会成为scope上的一个属性，所以在controller中我们再也不能使用$watch,$emit,$on之类的特殊方法，因为这些东西往往不该出现在controller中的，给大家一个警告，更好的控制。但是一旦如果没办法必须用的话，可以在征得项目组一致同意，将此controller退回$scope.  因为controller实例将会只是$scope的一个属性，所以view模板上的所有字段都会在一个引用的属性上，这可以避开JavaScript原型链继承对于值类型的坑。参加[https://github.com/angular/angular.js/wiki/Understanding-Scopes](https://github.com/angular/angular.js/wiki/Understanding-Scopes" \t "http://blog.csdn.net/benweizhu/article/details/_blank).  controller as 对于 coffescript,liveScript更友好。 |
| 模板上定义的每个字段方法都会在scope寄存在controller as别名上的引用上，所以在controller继承中，不会在出现命名冲突的问题。  注释:对于route，也有个controllerAs的属性可以设置，下面代码来自angular doc文档： |
| angular.module('ngViewExample', ['ngRoute', 'ngAnimate'], **function** ($routeProvider, $locationProvider) {  $routeProvider.when('/Book/:bookId', {  templateUrl: 'book.html',  controller: BookCntl,  controllerAs: 'book'  });  $routeProvider.when('/Book/:bookId/ch/:chapterId', {  templateUrl: 'chapter.html',  controller: ChapterCntl,  controllerAs: 'chapter'  });   // configure html5 to get links working on jsfiddle  $locationProvider.html5Mode(**true**); }); |
| demo | <**body** ng-controller="demo as d"> <**div** ng-click="d.click('parent',$event)">  given some text for click  <**hr**>  <**input** type="checkbox" ng-model="d.stopPropagation"/>Stop Propagation ?  <**hr**>  <**button** type="button" ng-click="d.click('button',$event)">button</**button**> </**div**> </**body**> |
| angular.module("app", []).controller("demo", [**function** () {  **var** vm = **this**;  vm.click = **function** (name, $event) {  console.log(name + " -----called");  **if** (vm.stopPropagation) {  $event.stopPropagation();  }  };  **return** vm; }]); |

angular.element方法汇总

|  |  |
| --- | --- |
|  | addClass()-为每个匹配的元素添加指定的样式类名 after()-在匹配元素集合中的每个元素后面插入参数所指定的内容，作为其兄弟节点 append()-在每个匹配元素里面的末尾处插入参数内容 attr() - 获取匹配的元素集合中的第一个元素的属性的值 bind() - 为一个元素绑定一个事件处理程序 children() - 获得匹配元素集合中每个元素的子元素，选择器选择性筛选 clone()-创建一个匹配的元素集合的深度拷贝副本 contents()-获得匹配元素集合中每个元素的子元素，包括文字和注释节点 css() - 获取匹配元素集合中的第一个元素的样式属性的值 data()-在匹配元素上存储任意相关数据 detach()-从DOM中去掉所有匹配的元素 empty()-从DOM中移除集合中匹配元素的所有子节点 eq()-减少匹配元素的集合为指定的索引的哪一个元素 find() - 通过一个选择器，jQuery对象，或元素过滤，得到当前匹配的元素集合中每个元素的后代 hasClass()-确定任何一个匹配元素是否有被分配给定的（样式）类 html()-获取集合中第一个匹配元素的HTML内容 next() - 取得匹配的元素集合中每一个元素紧邻的后面同辈元素的元素集合。如果提供一个选择器，那么只有紧跟着的兄弟元素满足选择器时，才会返回此元素 on() - 在选定的元素上绑定一个或多个事件处理函数 off() - 移除一个事件处理函数 one() - 为元素的事件添加处理函数。处理函数在每个元素上每种事件类型最多执行一次 parent() - 取得匹配元素集合中，每个元素的父元素，可以提供一个可选的选择器 prepend()-将参数内容插入到每个匹配元素的前面（元素内部） prop()-获取匹配的元素集中第一个元素的属性（property）值 ready()-当DOM准备就绪时，指定一个函数来执行 remove()-将匹配元素集合从DOM中删除。（同时移除元素上的事件及 jQuery 数据。） removeAttr()-为匹配的元素集合中的每个元素中移除一个属性（attribute） removeClass()-移除集合中每个匹配元素上一个，多个或全部样式 removeData()-在元素上移除绑定的数据 replaceWith()-用提供的内容替换集合中所有匹配的元素并且返回被删除元素的集合 text()-得到匹配元素集合中每个元素的合并文本，包括他们的后代 toggleClass()-在匹配的元素集合中的每个元素上添加或删除一个或多个样式类,取决于这个样式类是否存在或值切换属性。即：如果存在（不存在）就删除（添加）一个类 triggerHandler() -为一个事件执行附加到元素的所有处理程序 unbind() - 从元素上删除一个以前附加事件处理程序 val()-获取匹配的元素集合中第一个元素的当前值 wrap()-在每个匹配的元素外层包上一个html元素 |
|  |  |

AngularJS常用服务（$http、$location、$sce等）

|  |  |
| --- | --- |
|  | $http  $location  $cacheFactory  $timeout、$interval  $document  $log  $sce |
|  |  |

UI闪烁消除

|  |  |  |
| --- | --- | --- |
|  | <http://www.kuqin.com/shuoit/20131207/336826.html>  Angular的自动数据绑定功能是亮点，然而，他的另一面是：在Angular初始化之前，页面中可能会给用户呈现出没有解析的表达式。当DOM准备就绪，Angular计算并替换相应的值。这样就会导致出现一个丑陋的闪烁效果。  如果你做的是SPA(Single Page Application)，这个问题只会在第一次加载页面的时候出现，幸运的是，可以很容易杜绝这种情形发生: 放弃{{ }}表达式，改用ng-bind指令  如果你需要符合表达式，那就用[ng-bind-template](http://docs.angularjs.org/api/ng.directive:ngBindTemplate)吧，  如果用这个指令，为了区分字符串字面量和表达式，你需要使用大括号  另外一种方法就是完全隐藏元素，甚至可以隐藏整个应用，直到Angular就绪。  Angular为此还提供了[ng-cloak](http://docs.angularjs.org/api/ng.directive:ngCloak)指令，工作原理就是在初始化阶段inject了css规则，或者你可以包含这个css 隐藏规则到你自己的stylesheet。Angular就绪后就会移除这个cloak样式，让我们的应用(或者元素)立刻渲染。  Angular并不依赖jQuery。事实上，Angular源码里包含了一个内嵌的轻量级的jquery:[jqLite](http://docs.angularjs.org/api/angular.element). 当Angular检测到你的页面里有jQuery出现，他就会用这个jQuery而不再用jqLite，直接证据就是Angular里的元素抽象层。比如，在directive中访问你要应用到的元素。   |  | | --- | | angular.module('jqdependency', []).directive('failswithoutjquery', function () {  return {  restrict: 'A', link: function (scope, element, attrs) {  element.hide(4000)  }  } }); |   但是这个元素jqLite还是jQuery元素呢？取决于，手册上这么写的：  Angular中所有的元素引用都会被jQuery或者jqLite包装；他们永远不是纯DOM引用  所以Angular如果没有检测到jQuery，那么就会使用jqLite元素，hide()方法值能用于jQuery元素，所以说这个示例代码只能当检测到jQuery时才可以使用。如果你(不小心)修改了AngularJS和jQuery的出现顺序，这个代码就会失效！虽说没事挪脚本的顺序的事情不经常发生，但是在我开始模块化代码的时候确实给我造成了困扰。尤其是当你开始使用模块加载器(比如 [RequireJS](http://requirejs.org/" \t "http://blog.jobbole.com/52857/_blank)), 我的解决办法是在配置里显示的声明Angular确实依赖jQuery  另外一种方法就是你不要通过Angular元素的包装来调用jQuery特定的方法，而是使用$(element).hide(4000)来表明自己的意图。这样依赖，即使修改了script加载顺序也没事。 |

依赖注入使用在 js压缩

|  |  |
| --- | --- |
|  | 特别需要注意的是Angular应用压缩问题。否则错误信息比如 ‘Unknown provider:aProvider  <- a’ 会让你摸不到头脑。跟其他很多东西一样，这个错误在官方文档里也是无从查起的。简而言之，Angular依赖参数名来进行依赖注入。压缩器压根意识不到这个这跟Angular里普通的参数名有啥不同，尽可能的把脚本变短是他们职责。咋办？用“友好压缩法”来进行方法注入。看这里：  module.service('myservice', function($http, $q) {  // This breaks when minified  });  module.service('myservice', ['$http','$q',function($http, $q) {  // Using the array syntax to declare dependencies works with minification<b>!</b>  }]);  这个数组语法很好的解决了这个问题。我的建议是从现在开始照这个方法写，如果你决定压缩JavaScript，这个方法可以让你少走很多弯路。好像是一个[automatic rewriter](https://github.com/btford/ngmin" \t "http://www.kuqin.com/shuoit/20131207/_blank)机制，我也不太清楚这里面是怎么工作的。  最终一点建议：如果你想用数组语法复写你的functions，在所有Angular依赖注入的地方应用之。包括directives，还有directive里的controllers。别忘了逗号(经验之谈) |
| **/\* the directive itself needs array injection syntax:\*/ module**.directive('directive-with-controller', ['myservice', **function** (myservice) {  **return** {  controller: ['$timeout', **function** ($timeout) {  **/\* but this controller needs array injection syntax, too!\*/** }],  link: **function** (scope, element, attrs, ctrl) {  }  } }]); |
| 注意：link function不需要数组语法，因为他并没有真正的注入。这是被Angular直接调用的函数。Directive级别的依赖注入在link function里也是使用的。  Directive永远不会‘完成’  在directive中，一个令人掉头发的事就是directive已经‘完成’但你永远不会知道。当把jQuery插件整合到directive里时，这个通知尤为重要。假设你想用ng-repeat把动态数据以jQuery datatable的形式显示出来。当所有的数据在页面中加载完成后，你只需要调用$(‘.mytable).dataTable()就可以了。 但是，臣妾做不到啊！  为啥呢？Angular的数据绑定是通过持续的digest循环实现的。基于此，Angular框架里根本没有一个时间是‘休息’的。 一个解决方法就是将jQuery dataTable的调用放在当前digest循环外，用timeout方法就可以做到。  angular.module('table', []).directive('mytable', ['$timeout', **function** ($timeout) {  **return** {  restrict: 'E',  template: '<table class="mytable">' +  '<thead><tr><th>counting</th></tr></thead>' +  '<tr ng-repeat="data in datas"><td></td></tr>' +  '</table>',  link: **function** (scope, element, attrs, ctrl) {  scope.datas = ["one", "two", "three"]  // Doesn't work, shows an empty table:  // $('.mytable', element).dataTable()  // But this does:  $timeout(**function** () {  $('.mytable', element).dataTable();  }, 0)  }  } }]); |
|  |  |

AngularJs 模块化编程实例

|  |  |
| --- | --- |
|  | <http://www.jb51.net/article/91748.htm>  <http://www.jb51.net/article/60505.htm> |
|  | 在讲angularjs的模块之前，我们先介绍一下angular的一些知识点：  AngularJS是纯客户端技术，完全用Javascript编写的。它使用的是网页开发的常规技术(HTML,CSS,Javascript)，目的是让网页应用开发更快更容易。  AngularJS简化应用开发的一个重要方法是，将一个些通用的低级开发操作包装起来提供给开发者。AngularJS会自动处理好这些低级操作。它们包括：  1.DOM操作  2.设置事件的监听  3.输入验证，因为AngularJS会处理大部分这些操作，所以开发者就能更多的专注在应用的业务逻辑上，更少地编写那些重复性的、易错的、低级的代码。  在AngularJS简化开发的同时，它也为客户端带来了一些精巧的技术，它们包括：  1.数据、业务逻辑、视图的分离  2.数据和视图的自动绑定  3.通用服务  4.依赖注入（主要用于聚合服务）  5.可扩展的HTML编译器(完全由JavaScript编写)  6.易于测试  7.客户端对这些技术的需求其实已经存在很久了。  同时，你还可以用AngularJS来开发单页或者多页的应用，不过其主要还是用来开发单页的。 AngularJS支持浏览器的历史操作，向前，向后按钮，单页应用中的收藏操作。  接下来，我们来详细讲解angularJS的模块。  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）。 |
| 模块化写法 | 对于大规模的应用。我们建议将自己的应用按照如下建议，拆分为多个module：  1、service module，用于声明service。  2、directive module，用于声明directive。  3、filter module，用于声明filter。  4、应用级别的module，依赖上述的module，并且包含初始化的代码。  这样划分的理由是，当我们在测试的时候，往往需要忽略那些让测试变得困难的初始化代码。通过将代码分成独立的module，在测试中就可以很容易地忽略那些代码。这样，我们就可以更加专注在加载相应的module进行测试。  上面的只是一个建议，可以随意地按照自己的需求制定。 |
|  | Module Loading & Dependencies（模块加载和依赖）  　　module是配置（configuration）的集合，执行在启动应用的进程中应用的块（blocks）。在它的最简单的形式中，由两类block组成：  　　1.配置块（configuration blocks）：在provider注册和配置的过程中执行的。只有provider和constant（常量？）可以被注入（injected）到configuration blocks中。这是为了避免出现在service配置完毕之前service就被执行的意外。  2.运行块（run blocks）：在injector创建完成后执行，用于启动应用。只有实例（instances）和常量（constants）可以被注入到run block中。这是为了避免进一步的系统配置在程序运行的过程中执行。  3、ng自己的运行机制：  config阶段是给了ng上下文一个针对constant与provider修改其内部属性的一个阶段  而run阶段是在config之后的在运行独立的代码块，通常写法run Block  简单的说一下就是ng启动阶段是 config-->run-->compile/link  angular.module('myModule', []).config(**function** (injectables) { // provider-injector  // 这里是config block的一个例子  // 我们可以根据需要，弄N个这样的东东  // 我们可以在这里注入Providers (不是实例，not instances)到config block里面 }).run(**function** (injectables) { // instance-injector  // 这里是一个run block的例子  // 我们可以根据需要，弄N个这样的东东  // 我们只能注入实例（instances ）（不是Providers）到run block里面 }); |
| 样例 | a)Configuration Blocks（配置块）  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。例如：   |  | | --- | | angular.module('myModule', [])  .value('a', 123)  .factory('a', **function**() { **return** 123; })  .directive('directiveName', ...)  .**filter**('filterName', ...); | | // = = >  angular.module('myModule', [])  .config(**function**($provide, $compileProvider, $filterProvider) {  $provide.value('a', 123)  $provide.factory('a', **function**() { **return** 123; });  $compileProvider.directive('directiveName', ...);  $filterProvider.**register**('filterName', ...);  });  configuration blocks被应用的顺序，与它们的注册的顺序一致。对于常量定义来说，是一种额外的情况，即放置在configuration blocks开头的常量定义。  配置块会按照$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 | | 两个代码段的功能是一样的，不同写法主要有什么区别？$provide，provider,$compileProvider,$controllerProvider,$filterPrivoder，这几个的概念用法区分？  directive， factory 等module都是可以在多个项目中复用的， 然而每个项目的环境不同。调用Provider可以配置当前项目的环境。  打比方好了， 配置路由在config中调用了routeProvider来配置项目的路由， 每个项目是不同的; httpProvider配置$http的行为和返回内容， locationProvider来配置路由是否使用#，等等等等。  自定义的provider只需要在config中注入(provider名字)Provider就可以调用provider的方法了。  provider还有delegate方法， 如果当前项目的服务需要特别的function可以通过这个加入， 并不影响原先的服务。 | |
|  |
| b) Run Blocks（应用块）  run block是在angular中最接近main方法的东东。run block是必须执行，用于启动应用的代码。它将会在所有service配置、injector创建完毕后执行。run block通常包含那些比较难以进行单元测试的代码，就是因为这个原因，这些代码应该定义在一个独立的module中，让这些代码可以在单元测试中被忽略。  c) Dependencies（依赖）  一个module可以列出它所依赖的其他module。依赖一个module，意味着被请求（required）的module（被依赖的）必须在进行请求（requiring）module（需要依赖其他module的module，请求方）加载之前加载完成。换一种说法，被请求的module的configuration block会在请求的module的configuration block执行前执行（before the configuration blocks or the requiring module，这里的or怎么解释呢？）。对于run block也是这个道理。每一个module只能够被加载一次，即使有多个其他module需要（require）它。  d) Asynchronous Loading（异步加载）  module是管理$injector配置的方法之一，不用对加载脚本到VM做任何事情。现在已经有现成的项目专门用于处理脚本加载，也可以用到angular中。因为module在加载的过程中不做任何事情，它们可以按照任意的顺序被加载到VM中。脚本加载器可以利用这个特性，进行并行加载。 |
| 编程实例 | <**body**> <**div** ng-controller="**XmplController**">  {{ greeting }}! </**div**> <**script**>  //服务模块  //自定义greeter user对象  angular.module('xmpl.service', []).value('greeter', {  salutation: 'Hello',  localize: **function** (localization) {  **this**.salutation = localization.salutation;  },  greet: **function** (name) {  **return this**.salutation + ' ' + name + '!';  }  }).value('user', {  load: **function** (name) {  **this**.name = name;  }  });  angular.module('xmpl.directive', []); //指令模块  angular.module('xmpl.filter', []); //过滤器模块  // 模块xmpl依赖于数组中的模块  angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter']).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  **var XmplController** = **function** ($scope, greeter, user) {  $scope.greeting = greeter.greet(user.name);  } </**script**> |
| 这样拆分的原因是，在你的测试中常常需要忽略掉初始化代码，因为这些代码比较难测试。通过把它拆分出来就能在测试中方便地忽视掉它。通过只加载和当前测试相关的模块，也能使测试更专一。以上只是一个建议，你可以放心根据你的具体情况来调整。  Unit Testing（单元测试）  在单元测试的最简单的形式中，其中一个是在测试中实例化一个应用的子集，然后运行它们。重要的是，我们需要意识到对于每一个injector，每一个module只会被加载一次。通常一个应用只会有一个injector。但在测试中，每一个测试用例都有它的injector，这意味着在每一个VM中，module会被多次加载。正确地构建module，将对单元测试有帮助，正如下面的例子：  在这个例子中，我们准备假设定义如下的module：  angular.module('greetMod', [])  .factory('alert', function($window) {  return function(text) {  　　$window.alert(text);  };    })  .value('salutation', 'Hello')  .factory('greet', function(alert, salutation) {  return function(name) {  　　 alert(salutation + ' ' + name + '!');  };  });  让我们写一些测试用例：  describe('myApp', function() {  // 加载应用响应的module，然后加载指定的将$window重写为mock版本的测试module，  // 这样做，当进行window.alert()时，测试器就不会因被真正的alert窗口阻挡而停止  //这里是一个在测试中覆盖配置信息的例子  beforeEach(module('greetMod', function($provide) {//这里看来是要将真正的$window替换为以下的东东  　　$provide.value('$window', {  　　　　alert: jasmine.createSpy('alert')  　　});  }));    // inject()会创建一个injector，并且注入greet和$window到测试中。  // 测试不需要关心如何写应用，只需要关注如何测试应用。  it('should alert on $window', inject(function(greet, $window) {  　　greet('World');  　　expect($window.alert).toHaveBeenCalledWith('Hello World!');  }));    // 这里是在测试中通过行内module和inject方法来覆盖配置的方法  it('should alert using the alert service', function() {  　　var alertSpy = jasmine.createSpy('alert');  　　module(function($provide) {  　　　　$provide.value('alert', alertSpy);  　　});  　　inject(function(greet) {  　　　　greet('World');  　　　　expect(alertSpy).toHaveBeenCalledWith('Hello World!');  　　});  });  }); |
| 编程实例 | 加载配置和路由配置：  <script>  "use strict";  var tempApp = angular.module("templateApp", ["ui.router", "oc.lazyLoad"]);  tempApp.config(["$provide", "$compileProvider", "$controllerProvider", "$filterProvider", function ($provide, $compileProvider, $controllerProvider, $filterProvider) {  tempApp.controller = $controllerProvider.register;  tempApp.directive = $compileProvider.directive;  tempApp.filter = $filterProvider.register;  tempApp.factory = $provide.factory;  tempApp.service = $provide.service;  tempApp.constant = $provide.constant;  }]); </script>  以上代码对模块的注册，仅仅依赖了ui.router和oc.LazyLoad。配置也只是简单配置了模块，以便在后面的js能识别到tempApp上的方法。 |

在AngularJS框架中处理数据建模(服务应用实例)

|  |  |
| --- | --- |
|  | <http://developer.51cto.com/art/201401/426456.htm> |
|  | 我们知道，AngularJS并没有自带立等可用的数据建模方案。而是以相当抽象的方式，让我们在controller中使用JSON数据作为模 型。但是随着时间的推移和项目的成长，我意识到这种建模的方式不再能满足我们项目的需求。在这篇文章中我会介绍在我的AngularJS应用中处理数据建 模的方式。  为Controller定义模型  让我们从一个简单的例子开始。我想要显示一个书本(book)的页面。下面是控制器(Controller)：  BookController  app.controller('BookController', ['$scope', function($scope) {      $scope.book = {          id: 1,          name: 'Harry Potter',          author: 'J. K. Rowling',          stores: [              { id: 1, name: 'Barnes & Noble', quantity: 3},              { id: 2, name: 'Waterstones', quantity: 2},              { id: 3, name: 'Book Depository', quantity: 5}          ]      };  }]);  这个控制器创建了一个书本的模型，我们可以在后面的模板中(templage)中使用它。  template for displaying a book  <div ng-controller="BookController">      Id: <span ng-bind="book.id"></span>        Name:<input type="text" ng-model="book.name" />        Author: <input type="text" ng-model="book.author" />  </div>  假如我们需要从后台的api获取书本的数据，我们需要使用$http:  BookController with $http  app.controller('BookController', ['$scope', '$http', function($scope, $http) {      var bookId = 1;        $http.get('ourserver/books/' + bookId).success(function(bookData) {          $scope.book = bookData;      });  }]);  注意到这里的bookData仍然是一个JSON对象。接下来我们想要使用这些数据做一些事情。比如，更新书本信息，删除书本，甚至其他的一些不涉及到后台的操作，比如根据请求的图片大小生成一个书本图片的url，或者判断书本是否有效。这些方法都可以被定义在控制器中。  BookController with several book actions  app.controller('BookController', ['$scope', '$http', function($scope, $http) {      var bookId = 1;        $http.get('ourserver/books/' + bookId).success(function(bookData) {          $scope.book = bookData;      });        $scope.deleteBook = function() {          $http.delete('ourserver/books/' + bookId);      };        $scope.updateBook = function() {          $http.put('ourserver/books/' + bookId, $scope.book);      };        $scope.getBookImageUrl = function(width, height) {          return 'our/image/service/' + bookId + '/width/height';      };        $scope.isAvailable = function() {          if (!$scope.book.stores || $scope.book.stores.length === 0) {              return false;          }          return $scope.book.stores.some(function(store) {              return store.quantity > 0;          });      };  }]);  然后在我们的模板中：  template for displaying a complete book  <div ng-controller="BookController">      <div ng-style="{ backgroundImage: 'url(' + getBookImageUrl(100, 100) + ')' }"></div>      Id: <span ng-bind="book.id"></span>        Name:<input type="text" ng-model="book.name" />        Author: <input type="text" ng-model="book.author" />        Is Available: <span ng-bind="isAvailable() ? 'Yes' : 'No' "></span>        <button ng-click="deleteBook()">Delete</button>        <button ng-click="updateBook()">Update</button>  </div>  在controllers之间共享Model  如果书本的结构和方法只和一个控制器有关，那我们现在的工作已经可以应付。但是随着应用的增长，会有其他的控制器也需要和书本打交道。那些控制器很 多时候也需要获取书本，更新它，删除它，或者获得它的图片url以及看它是否有效。因此，我们需要在控制器之间共享这些书本的行为。我们需要使用一个返回 书本行为的factory来实现这个目的。在动手写一个factory之前，我想在这里先提一下，我们创建一个factory来返回带有这些book辅助 方法的对象，但我更倾向于使用prototype来构造一个Book类，我觉得这是更正确的选择：  Book model service  app.factory('Book', ['$http', function($http) {      function Book(bookData) {          if (bookData) {              this.setData(bookData):          }          // Some other initializations related to book      };      Book.prototype = {          setData: function(bookData) {              angular.extend(this, bookData);          },          load: function(id) {              var scope = this;              $http.get('ourserver/books/' + bookId).success(function(bookData) {                  scope.setData(bookData);              });          },          delete: function() {              $http.delete('ourserver/books/' + bookId);          },          update: function() {              $http.put('ourserver/books/' + bookId, this);          },          getImageUrl: function(width, height) {              return 'our/image/service/' + this.book.id + '/width/height';          },          isAvailable: function() {              if (!this.book.stores || this.book.stores.length === 0) {                  return false;              }              return this.book.stores.some(function(store) {                  return store.quantity > 0;              });          }      };      return Book;  }]);  这种方式下，书本相关的所有行为都被封装在Book服务内。现在，我们在BookController中来使用这个亮眼的Book服务。  BookController that uses Book model  app.controller('BookController', ['$scope', 'Book', function($scope, Book) {      $scope.book = new Book();      $scope.book.load(1);  }]);  正如你看到的，控制器变得非常简单。它创建一个Book实例，指派给scope，并从后台加载。当书本被加载成功时，它的属性会被改变，模板也随着 被更新。记住其他的控制器想要使用书本功能，只要简单地注入Book服务即可。此外，我们还要改变template使用book的方法。  template that uses book instance  <div ng-controller="BookController">      <div ng-style="{ backgroundImage: 'url(' + book.getImageUrl(100, 100) + ')' }"></div>      Id: <span ng-bind="book.id"></span>        Name:<input type="text" ng-model="book.name" />        Author: <input type="text" ng-model="book.author" />        Is Available: <span ng-bind="book.isAvailable() ? 'Yes' : 'No' "></span>        <button ng-click="book.delete()">Delete</button>        <button ng-click="book.update()">Update</button>  </div>  到这里，我们知道了如何建模一个数据，把他的方法封装到一个类中，并且在多个控制器中共享它，而不需要写重复代码。 |
|  | 在多个控制器中使用相同的书本模型  我们定义了一个书本模型，并且在多个控制器中使用了它。在使用了这种建模架构之后你会注意到有一个严重的问题。到目前为止，我们假设多个控制器对书本进行操作，但如果有两个控制器同时处理同一本书会是什么情况呢？  假设我们页面的一块区域我们所有书本的名称，另一块区域可以更新某一本书。对应这两块区域，我们有两个不同的控制器。第一个加载书本列表，第二个加 载特定的一本书。我们的用户在第二块区域中修改了书本的名称并且点击“更新”按钮。更新操作成功后，书本的名称会被改变。但是在书本列表中，这个用户始终 看到的是修改之前的名称！真实的情况是我们对同一本书创建了两个不同的书本实例——一个在书本列表中使用，而另一个在修改书本时使用。当用户修改书本名称 的时候，它实际上只修改了后一个实例中的属性。然而书本列表中的书本实例并未得到改变。  解决这个问题的办法是在所有的控制器中使用相同的书本实例。在这种方式下，书本列表和书本修改的页面和控制器都持有相同的书本实例，一旦这个实例发 生变化，就会被立刻反映到所有的视图中。那么按这种方式行动起来，我们需要创建一个booksManager服务（我们没有大写开头的b字母，是因为这是 一个对象而不是一个类）来管理所有的书本实例池，并且富足返回这些书本实例。如果被请求的书本实例不在实例池中，这个服务会创建它。如果已经在池中，那么 就直接返回它。请牢记，所有的加载书本的方法最终都会被定义在booksManager服务中，因为它是唯一的提供书本实例的组件。  booksManager service  app.factory('booksManager', ['$http', '$q', 'Book', function($http, $q, Book) {      var booksManager = {          \_pool: {},          \_retrieveInstance: function(bookId, bookData) {              var instance = this.\_pool[bookId];                if (instance) {                  instance.setData(bookData);              } else {                  instance = new Book(bookData);                  this.\_pool[bookId] = instance;              }                return instance;          },          \_search: function(bookId) {              return this.\_pool[bookId];          },          \_load: function(bookId, deferred) {              var scope = this;                $http.get('ourserver/books/' + bookId)                  .success(function(bookData) {                      var book = scope.\_retrieveInstance(bookData.id, bookData);                      deferred.resolve(book);                  })                  .error(function() {                      deferred.reject();                  });          },          /\* Public Methods \*/          /\* Use this function in order to get a book instance by it's id \*/          getBook: function(bookId) {              var deferred = $q.defer();              var book = this.\_search(bookId);              if (book) {                  deferred.resolve(book);              } else {                  this.\_load(bookId, deferred);              }              return deferred.promise;          },          /\* Use this function in order to get instances of all the books \*/          loadAllBooks: function() {              var deferred = $q.defer();              var scope = this;              $http.get('ourserver/books)                  .success(function(booksArray) {                      var books = [];                      booksArray.forEach(function(bookData) {                          var book = scope.\_retrieveInstance(bookData.id, bookData);                          books.push(book);                      });                        deferred.resolve(books);                  })                  .error(function() {                      deferred.reject();                  });              return deferred.promise;          },          /\*  This function is useful when we got somehow the book data and we wish to store it or update the pool and get a book instance in return \*/          setBook: function(bookData) {              var scope = this;              var book = this.\_search(bookData.id);              if (book) {                  book.setData(bookData);              } else {                  book = scope.\_retrieveInstance(bookData);              }              return book;          },        };      return booksManager;  }]);  下面是我们的EditableBookController和BooksListController两个控制器的代码：  EditableBookController and BooksListController that uses booksManager  app.factory('Book', ['$http', function($http) {      function Book(bookData) {          if (bookData) {              this.setData(bookData):          }          // Some other initializations related to book      };      Book.prototype = {          setData: function(bookData) {              angular.extend(this, bookData);          },          delete: function() {              $http.delete('ourserver/books/' + bookId);          },          update: function() {              $http.put('ourserver/books/' + bookId, this);          },          getImageUrl: function(width, height) {              return 'our/image/service/' + this.book.id + '/width/height';          },          isAvailable: function() {              if (!this.book.stores || this.book.stores.length === 0) {                  return false;              }              return this.book.stores.some(function(store) {                  return store.quantity > 0;              });          }      };      return Book;  }]);  需要注意的是，模块（template）中还是保持原来使用book实例的方式。现在应用中只持有一个id为1的book实例，它发生的所有改变都会被反映到使用它的各个页面上。  总结  在这片文章中，我建议了AngularJS中建模数据的一种架构。首先，我展示了AngularJS默认的数据模型绑定，然后讲了如何封装模型的方 法和操作从而可以在不同的控制其中重用它们，最后我解释了如何管理模型实例从而使得所有的改变都能被反映到应用中各个相关的视图上。  希望这篇文章能在如何实现数据建模上给你一些启示。  原文链接：<http://www.webdeveasy.com/angularjs-data-model/>  译文链接：<http://blog.jobbole.com/54817/> |