

AngularJS 学习笔记

2014-01-22 21:48 更新

邹业盛

1. 关于AngularJS
2. 关于本文档
3. 开始的例子
4. 依赖注入
5. 作用域
6. 数据绑定与模板
 - 6.1. 数据->模板
 - 6.2. 模板->数据
 - 6.3. 数据->模板->数据->模板
7. 模板
 - 7.1. 定义模板内容
 - 7.2. 内容渲染控制
 - 7.3. 节点控制
 - 7.4. 事件绑定
 - 7.5. 表单控件
8. 模板中的过滤器
 - 8.1. 排序 `orderBy`
 - 8.2. 过滤列表 `filter`
 - 8.3. 其它
 - 8.4. 例子: 表头排序
 - 8.5. 例子: 搜索
9. 锚点路由
 - 9.1. 路由定义
 - 9.2. 参数定义
 - 9.3. 业务处理
10. 定义模板变量标识标签
11. **AJAX**
 - 11.1. HTTP请求
 - 11.2. 广义回调管理
12. 工具函数
 - 12.1. 上下文绑定
 - 12.2. 对象处理
 - 12.3. 类型判定
13. 其它服务
 - 13.1. 日志
 - 13.2. 缓存
 - 13.3. 计时器
 - 13.4. 表达式函数化
 - 13.5. 模板单独使用
14. 自定义模块和服务
 - 14.1. 模块和服务的概念与关系
 - 14.2. 定义模块
 - 14.3. 定义服务
 - 14.4. 引入模块并使用服务
15. 附加模块 `ngResource`
 - 15.1. 使用引入与整体概念
 - 15.2. 基本定义

- 15.3. 基本使用
- 15.4. 定义和使用时的占位量
- 15.5. 实例
- 16. AngularJS与其它框架的混用(jQuery, Dojo)
- 17. 自定义过滤器
- 18. 自定义指令directive
 - 18.1. 指令的使用
 - 18.2. 指令的执行过程
 - 18.3. 基本的自定义方法
 - 18.4. 属性值类型的自定义
 - 18.5. Compile的细节
 - 18.6. transclude的细节
 - 18.7. 把节点内容作为变量处理的类型
 - 18.8. 指令定义时的参数
 - 18.9. Attributes的细节
 - 18.10. 预定义的 NgModelController
 - 18.11. 预定义的 FormController
 - 18.12. 示例: 文本框
 - 18.13. 示例: 模板控制语句 for
 - 18.14. 示例: 模板控制语句 if/else

阿里巴巴2016应届生招聘(内推流程)

本文的内容是在 1.0.x 版本之下完成的。

1. 关于AngularJS

AngularJS 是 Google 开源出来的一套 js 工具。下面简称其为 **ng**。这里只说它是“工具”，没说它是完整的“框架”，是因为它并不是定位于去完成一套框架要做的事。更重要的，是它给我们揭示了一种新的应用组织与开发方式。

ng 最让我称奇的，是它的数据双向绑定。其实想想，我们一直在提数据与表现的分离，但是这里的“双向绑定”从某方面来说，是把数据与表现完全绑定在一起——数据变化，表现也变化。反之，表现变化了，内在的数据也变化。有过开发经验的人能体会到这种机制对于前端应用来说，是很有必要的，能带来维护上的巨大优势。当然，这里的绑定与提倡的分离并不是矛盾的。

ng 可以和 jQuery 集成工作，事实上，如果没有 jQuery，ng 自己也做了一个轻量级的 jQuery，主要实现了元素操作部分的 API。

关于 ng 的几点：

- 对 IE 方面，它兼容 IE8 及以上的版本。
- 与 jQuery 集成工作，它的一些对象与 jQuery 相关对象表现是一致的。
- 使用 ng 时不要冒然去改变相关 DOM 的结构。

2. 关于本文档

这份文档如其名，是我自己学习 ng 的过程记录。只是过程记录，没有刻意像教程那样去做。所以呢，从前至后，中间不免有一些概念不清不明的地方。因为事实上，在某个阶段对于一些概念本来就不可能明白。所以，整个过程只求在形式上的能用即可——直到最后的“自定义”那几章，特别是“自定义指令”，那几章过完，你才能看清 ng 本来的面貌。前面就不要太纠结概念，本质，知道怎

么用就好。

3. 开始的例子

我们从一个完整的例子开始认识 ng :

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8" />
5
6  <title>试验</title>
7
8  <script type="text/javascript" src="jquery-1.8.3.js"></script>
9  <script type="text/javascript" src="angular.js"></script>
10
11 </head>
12 <body>
13   <div ng-controller="BoxCtrl">
14     <div style="width: 100px; height: 100px; background-color: red;"
15       ng-click="click()"></div>
16     <p>{{ w }} x {{ h }}</p>
17     <p>W: <input type="text" ng-model="w" /></p>
18     <p>H: <input type="text" ng-model="h" /></p>
19   </div>
20
21
22   <script type="text/javascript" charset="utf-8">
23
24
25   var BoxCtrl = function($scope, $element){
26
27     // $element 就是一个 jQuery 对象
28     var e = $element.children().eq(0);
29     $scope.w = e.width();
30     $scope.h = e.height();
31
32     $scope.click = function(){
33       $scope.w = parseInt($scope.w) + 10;
34       $scope.h = parseInt($scope.h) + 10;
35     }
36
37     $scope.$watch('w',
38       function(to, from){
39         e.width(to);
40       }
41     );
42
43     $scope.$watch('h',
44       function(to, from){
45         e.height(to);
46       }
47     );
48   }
49
50   angular.bootstrap(document, documentElement);
51 </script>
52 </body>
53 </html>
```

从上面的代码中，我们看到在通常的 HTML 代码当中，引入了一些标记，这些就是 ng 的模板机制，它不光完成数据渲染的工作，还实现了数据绑定的功能。

同时，在 HTML 中的本身的 DOM 层级结构，被 ng 利用起来，直接作为它的内部机制中，上下文结构的判断依据。比如例子中 p 是 div 的子节点，那么 p 中的那些模板标记就是在 div 的 Ctrl 的作用范围之内。

其它的，也同样写一些 js 代码，里面重要的是作一些数据的操作，事件的绑定定义等。这样，数据的变化就会和页面中的 DOM 表现联系起来。一旦这种联系建立起来，也即完成了我们所说的“双向绑定”。然后，这里说的“事件”，除了那些“点击”等通常的 DOM 事件之外，我们还更关注“数据变化”这个事件。

最后，可以使用：

```
angular.bootstrap(document, documentElement);
```

来把整个页面驱动起来了。（你可以看到一个可被控制大小的红色方块）

更完整的方法是定义一个 APP：

```
1  <!DOCTYPE html>
2  <html ng-app="MyApp">
3  <head>
4  <meta charset="utf-8" />
5
6  <title>数据正向绑定</title>
7
8  <script type="text/javascript" src="jquery-1.8.3.js"></script>
9  <script type="text/javascript" src="angular.js"></script>
10
11 </head>
12 <body>
13
14 <div ng-controller="TestCtrl">
15   <input type="text" value="" id="a" />
16 </div>
17
18
19 <script type="text/javascript">
20 var TestCtrl = function() {
21   console.log('ok');
22 }
23
24 //angular.bootstrap(document, documentElement);
25 angular.module('MyApp', [], function() {console.log('here')});
26 </script>
27
28 </body>
29 </html>
```

这里说的一个 App 就是 ng 概念中的一个 Module。对于 Controller 来说， 如果不想使用全局函数，也可以在 app 中定义：

```
var app = angular.module('MyApp', [], function() {console.log('here')});
app.controller('TestCtrl',
  function($scope) {
```

```
        console.log('ok');
    }
};
```

上面我们使用 **ng-app** 来指明要使用的 **App**，这样的话可以把显式的初始化工作省了。一般完整的过程是：

```
var app = angular.module('Demo', [], angular.noop);
angular.bootstrap(document, ['Demo']);
```

使用 **angular.bootstrap** 来显示地做初始化工具，参数指明了根节点，装载的模块（可以是多个模块）。

4. 依赖注入

injector，我从 **ng** 的文档中得知这个概念，之后去翻看源码时了解了一下这个机制的工作原理。感觉就是虽然与自己的所想仅差那么一点点，但就是这么一点点，让我感慨想象力之神奇。

先看我们之前代码中的一处函数定义：

```
var BoxCtrl = function($scope, $element) {}
```

在这个函数定义中，注意那两个参数：**\$scope**，**\$element**，这是两个很有意思的东西。总的来说，它们是参数，这没什么可说的。但又不仅仅是参数——你换个名字代码就不能正常运行了。

事实上，这两个参数，除了完成“参数”的本身任务之外，还作为一种语法糖完成了“依赖声明”的任务。本来这个函数定义，完整的写法应该像 **AMD** 声明一样，写成：

```
var BoxCtrl = ['$scope', '$element', function(s, e) {}];
```

这样就很明显，表示有一个函数，它依赖于两个东西，然后这两个东西会依次作为参数传入。

简单起见，就写成了一个函数定义原本的样子，然后在定义参数的名字上作文章，来起到依赖声明的作用。

在处理时，通过函数对象的 **toString()** 方法可以知道这个函数定义代码的字符串表现形式，然后就知道它的参数是 **\$scope** 和 **\$element**。通过名字判断出这是两个外部依赖，然后就去获取资源，最后把资源作为参数，调用定义的函数。

所以，参数的名字是不能随便写的，这里也充分利用了 **js** 的特点来尽量做到“反省”了。

在 **Python** 中受限于函数名的命名规则，写出来不太好看。不过也得利于反省机制，做到这点也很容易：

```
# -*- coding: utf-8 -*-

def f(Ia, Ib):
    print Ia, Ib

args = f.func_code.co_varnames
SRV_MAP = {
    'Ia': '123',
    'Ib': '456',
```

```

    }

    srv = {}
    for a in args:
        if a in SRV_MAP:
            srv[a] = SRV_MAP[a]
    f(**srv)

```

5. 作用域

这里提到的“作用域”的概念，是一个在范围上与 DOM 结构一致，数据上相对于某个 `$scope` 对象的属性的概念。我们还是从 HTML 代码上来入手：

```

<div ng-controller="BoxCtrl">
  <div style="width: 100px; height: 100px; background-color: red;"
    ng-click="click()">
  </div>
  <p>{{ w }} x {{ h }}</p>
  <p>W: <input type="text" ng-model="w" /></p>
  <p>H: <input type="text" ng-model="h" /></p>
</div>

```

上面的代码中，我们给一个 `div` 元素指定了一个 `BoxCtrl`，那么，`div` 元素之内，就是 `BoxCtrl` 这个函数运行时，`$scope` 这个注入资源的控制范围。在代码中我们看到的 `click()`，`w`，`h` 这些东西，它们本来的位置对应于 `$scope.click`，`$scope.w`，`$scope.h`。

我们在后面的 `js` 代码中，也可以看到我们就是在操作这些变量。依赖于 `ng` 的数据绑定机制，操作变量的结果直接在页面上表现出来了。

6. 数据绑定与模板

我纠结了半天，“数据绑定”与“模板”这两个东西还真没办法分开来说。因为数据绑定需要以模板为载体，离开了模板，数据还绑个毛啊。

`ng` 的一大特点，就是数据双向绑定。双向绑定是一体，为了描述方便，下面分别介绍。

6.1. 数据->模板

数据到表现的绑定，主要是使用模板标记直接完成的：

```

<p>{{ w }} x {{ h }}</p>

```

使用 `{{ }}` 这个标记，就可以直接引用，并绑定一个作用域内的变量。在实现上，`ng` 自动创建了一个 `watcher`。效果就是，不管因为什么，如果作用域的变量发生了改变，我们随时可以让相应的页面表现也随之改变。我们可以看一个更纯粹的例子：

```

<p id="test" ng-controller="TestCtrl">{{ a }}</p>

<script type="text/javascript">
var TestCtrl = function($scope) {
    $scope.a = '123';
}
angular.bootstrap(document, documentElement);

```

上面的例子在页面载入之后，我们可以在页面上看到 **123**。这时，我们可以打开一个终端控制器，输入：

```
$('#test').scope().a = '12345';
$('#test').scope().$digest();
```

上面的代码执行之后，就可以看到页面变化了。

对于使用 **ng** 进行的事件绑定，在处理函数中就不需要去关心 **\$digest()** 的调用了。因为 **ng** 会自己处理。源码中，对于 **ng** 的事件绑定，真正的处理函数不是指定名字的函数，而是经过 **\$apply()** 包装过的一个函数。这个 **\$apply()** 做的一件事，就是调用根作用域 **\$rootScope** 的 **\$digest()**，这样整个世界就清净了：

```
<p id="test" ng-controller="TestCtrl" ng-click="click()">{{ a }}</p>

<script type="text/javascript" charset="utf-8">
var TestCtrl = function($scope) {
    $scope.a = '123';

    $scope.click = function() {
        $scope.a = '456';
    }
}
angular.bootstrap(document, documentElement);
```

那个 **click** 函数的定义，绑定时变成了类似于：

```
function() {
    $scope.$apply(
        function() {
            $scope.click();
        }
    )
}
```

这里的 **\$scope.\$apply()** 中做的一件事：

```
$rootScope.$digest();
```

6.2. 模板->数据

模板到数据的绑定，主要是通过 **ng-model** 来完成的：

```
<input type="text" id="test" ng-controller="TestCtrl" ng-model="a" />

<script type="text/javascript" charset="utf-8">
var TestCtrl = function($scope) {
    $scope.a = '123';
}
```

这时修改 **input** 中的值，然后再在控制终端中使用：

```
$('#test').scope().a
```

查看，发现变量 **a** 的值已经更改了。

实际上，**ng-model** 是把两个方向的绑定都做了。它不光显示出变量的值，也把显示上的数值变化反映给了变量。这个在实现上就简单多了，只是绑定 **change** 事件，然后做一些赋值操作即可。不过 **ng** 里，还要区分对待不同的控件。

6.3. 数据->模板->数据->模板

现在要考虑的是一种在现实中很普遍的一个需求。比如就是我们可以输入数值，来控制一个矩形的长度。在这里，数据与表现的关系是：

- 长度数值保存在变量中
- 变量显示于某个 **input** 中
- 变量的值即是矩形的长度
- **input** 中的值变化时，变量也要变化
- **input** 中的值变化时，矩形的长度也要变化

当然，要实现目的在这里可能就不止一种方案了。按照以前的做法，很自然地会想法，绑定 **input** 的 **change** 事件，然后去做一些事就好了。但是，我们前面提到过 **ng-model** 这个东西，利用它就可以在不手工处理 **change** 的条件下完成数据的展现需求，在此基础上，我们还需要做的一点，就是把变化后的数据应用到矩形的长度之上。

最开始，我们面对的应该是这样一个东西：

```
<div ng-controller="TestCtrl">
  <div style="width: 100px; height: 10px; background-color: red"></div>
  <input type="text" name="width" ng-model="width" />
</div>

<script type="text/javascript" charset="utf-8">
var TestCtrl = function($scope) {
  $scope.width = 100;
}
angular.bootstrap(document, documentElement);
</script>
```

我们从响应数据变化，但又不使用 **change** 事件的角度来看，可以这样处理宽度变化：

```
var TestCtrl = function($scope, $element) {
  $scope.width = 100;
  $scope.$watch('width',
    function(to, from) {
      $element.children(':first').width(to);
    }
  );
}
```

使用 **\$watch()** 来绑定数据变化。

当然，这种样式的问题，有更直接有效的手段，**ng** 的数据绑定总是让人惊异：

```
<div ng-controller="TestCtrl">
  <div style="width: 10px; height: 10px; background-color: red" ng-style="style">
  </div>
  <input type="text" name="width" ng-model="style.width" />
</div>
```



```
</div>

<script type="text/javascript" charset="utf-8">
var TestCtrl = function($scope) {
  $scope.style = {width: 100 + 'px'};
}
angular.bootstrap(document.documentElement);
</script>
```

7. 模板

前面讲了数据绑定之后，现在可以单独讲讲模板了。

作为一套能称之为“模板”的系统，除了能干一些模板的常规的事之外（好吧，即使是常规的逻辑判断现在它也做不了的），配合作用域 **\$scope** 和 **ng** 的数据双向绑定机制，**ng** 的模板系统就变得比较神奇了。

7.1. 定义模板内容

定义模板的内容现在有三种方式：

1. 在需要的地方直接写字符串
2. 外部文件
3. 使用 **script** 标签定义的“内部文件”

第一种不需要多说。第二种和第三种都可以和 **ng-include** 一起工作，来引入一段模板。

直接引入同域的外部文件作为模板的一部分：

```
<div ng-include src="'tpl.html'">
</div>

<div ng-include="tpl.html">
</div>
```

注意，**src** 中的字符串会作为表达式处理（可以是 **\$scope** 中的变量），所以，直接写名字的话需要使用引号。

引入 **script** 定义的“内部文件”：

```
<script type="text/ng-template" id="tpl">
here, {{ 1 + 1 }}
</script>

<div ng-include src="'tpl'"></div>
```

配合变量使用：

```
<script type="text/ng-template" id="tpl">
here, {{ 1 + 1 }}
</script>

<a ng-click="v='tpl'">Load</a>
```

```
<div ng-include src="v"></div>
```

7.2. 内容渲染控制

7.2.1. 重复 ng-repeat

这算是唯一的一个控制标签么.....，它的使用方法类型于：

```
<div ng-controller="TestCtrl">
  <ul ng-repeat="member in obj_list">
    <li>{{ member }}</li>
  </ul>
</div>

var TestCtrl = function($scope) {
  $scope.obj_list = [1, 2, 3, 4];
}
```

除此之外，它还提供了几个变量可供使用：

- **\$index** 当前索引
- **\$first** 是否为头元素
- **\$middle** 是否为非头非尾元素
- **\$last** 是否为尾元素

```
<div ng-controller="TestCtrl">
  <ul ng-repeat="member in obj_list">
    <li>{{ $index }}, {{ member.name }}</li>
  </ul>
</div>

var TestCtrl = function($scope) {
  $scope.obj_list = [{name: 'A'}, {name: 'B'}, {name: 'C'}];
}
```

7.2.2. 赋值 ng-init

这个指令可以在模板中直接赋值，它作用于 **angular.bootstrap** 之前，并且，定义的变量与 **\$scope** 作用域无关。

```
<div ng-controller="TestCtrl" ng-init="a=[1, 2, 3, 4];">
  <ul ng-repeat="member in a">
    <li>{{ member }}</li>
  </ul>
</div>
```

7.3. 节点控制

7.3.1. 样式 ng-style

可以使用一个结构直接表示当前节点的样式：

```
<div ng-style="{width: 100 + 'px', height: 100 + 'px', backgroundColor: 'red'}">
</div>
```

同样地，绑定一个变量的话，威力大了。

7.3.2. 类 ng-class

就是直接地设置当前节点的类，同样，配合数据绑定作用就大了：

```
<div ng-controller="TestCtrl" ng-class="cls">
</div>
```

ng-class-even 和 **ng-class-odd** 是和 **ng-repeat** 配合使用的：

```
<ul ng-init="l=[1,2,3,4]">
  <li ng-class-odd="'odd'" ng-class-even="'even'" ng-repeat="m in l">{{ m }}</li>
</ul>
```

注意里面给的还是表示式，别少了引号。

7.3.3. 显示和隐藏 ng-show ng-hide ng-switch

前两个是控制 **display** 的指令：

```
<div ng-show="true">1</div>
<div ng-show="false">2</div>
<div ng-hide="true">3</div>
<div ng-hide="false">4</div>
```

后一个 **ng-switch** 是根据一个值来决定哪个节点显示，其它节点移除：

```
<div ng-init="a=2">
  <ul ng-switch on="a">
    <li ng-switch-when="1">1</li>
    <li ng-switch-when="2">2</li>
    <li ng-switch-default>other</li>
  </ul>
</div>
```

7.3.4. 其它属性控制

ng-src 控制 **src** 属性：

```

```

ng-href 控制 **href** 属性：

```
<a ng-href="{{ '#' + '123' }}">here</a>
```

总的来说：

- **ng-src** src属性
- **ng-href** href属性
- **ng-checked** 选中状态
- **ng-selected** 被选择状态
- **ng-disabled** 禁用状态

- **ng-multiple** 多选状态
- **ng-readonly** 只读状态

注意： 上面的这些只是单向绑定，即只是从数据到展示，不能反作用于数据。要双向绑定，还是要使用 **ng-model** 。

7.4. 事件绑定

事件绑定是模板指令中很好用的一部分。我们可以把相关事件的处理函数直接写在 **DOM** 中，这样做的最大好处就是可以从 **DOM** 结构上看出业务处理的形式，你知道当你点击这个节点时哪个函数被执行了。

- **ng-change**
- **ng-click**
- **ng-dblclick**
- **ng-mousedown**
- **ng-mouseenter**
- **ng-mouseleave**
- **ng-mousemove**
- **ng-mouseover**
- **ng-mouseup**
- **ng-submit**

对于事件对象本身，在函数调用时可以直接使用 **\$event** 进行传递：

```
<p ng-click="click($event)">点击</p>
<p ng-click="click($event.target)">点击</p>
```

7.5. 表单控件

表单控件类的模板指令，最大的作用是它预定义了需要绑定的数据的格式。这样，就可以对于既定的数据进行既定的处理。

7.5.1. form

form 是核心的一个控件。**ng** 对 **form** 这个标签作了包装。事实上，**ng** 自己的指令是叫 **ng-form** 的，区别在于，**form** 标签不能嵌套，而使用 **ng-form** 指令就可以做嵌套的表单了。

form 的行为中依赖它里面的各个输入控制的状态的，在这里，我们主要关心的是 **form** 自己的一些方法和属性。从 **ng** 的角度来说，**form** 标签，是一个模板指令，也创建了一个 **FormController** 的实例。这个实例就提供了相应的属性和方法。同时，它里面的控件也是一个 **NgModelController** 实例。

很重要的一点，**form** 的相关方法要生效，必须为 **form** 标签指定 **name** 和 **ng-controller**，并且每个控件都要绑定一个变量。**form** 和控件的名字，即是 **\$scope** 中的相关实例的引用变量名。

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="text" name="a" required ng-model="a" />
  <span ng-click="see()">{{ test_form.$valid }}</span>
</form>

var TestCtrl = function($scope) {
```

```

$scope.see = function() {
    console.log($scope.test_form);
    console.log($scope.test_form.a);
}
}

```

除去对象的方法与属性，**form** 这个标签本身有一些动态类可以使用：

- **ng-valid** 当表单验证通过时的设置
- **ng-invalid** 当表单验证失败时的设置
- **ng-pristine** 表单的未被动之前拥有
- **ng-dirty** 表单被动过之后拥有

form 对象的属性有：

- **\$pristine** 表单是否未被动过
- **\$dirty** 表单是否被动过
- **\$valid** 表单是否验证通过
- **\$invalid** 表单是否验证失败
- **\$error** 表单的验证错误

其中的 **\$error** 对象包含有所有字段的验证信息，及对相关字段的 **NgModelController** 实例的引用。它的结构是一个对象，**key** 是失败信息，**required**，**minlength** 之类的，**value** 是对应的字段实例列表。

注意，这里的失败信息是按序列取的一个。比如，如果一个字段既要求 **required**，也要求 **minlength**，那么当它为空时，**\$error** 中只有 **required** 的失败信息。只输入一个字符之后，**required** 条件满足了，才可能有 **minlength** 这个失败信息。

```

<form name="test_form" ng-controller="TestCtrl">
  <input type="text" name="a" required ng-model="a" />
  <input type="text" name="b" required ng-model="b" ng-minlength="2" />
  <span ng-click="see()">{{ test_form.$error }}</span>
</form>

var TestCtrl = function($scope) {
  $scope.see = function() {
    console.log($scope.test_form.$error);
  }
}

```

7.5.2. input

input 是数据的最主要入口。**ng** 支持 **HTML5** 中的相关属性，同时对旧浏览器也做了兼容性处理。最重要的，**input** 的规则定义，是所属表单的相关行为的参照（比如表单是否验证成功）。

input 控件的相关可用属性为：

- **name** 名字
- **ng-model** 绑定的数据
- **required** 是否必填
- **ng-required** 是否必填
- **ng-minlength** 最小长度

- **ng-maxlength** 最大长度
- **ng-pattern** 匹配模式
- **ng-change** 值变化时的回调

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="text" name="a" ng-model="a" required ng-pattern="/abc/" />
  <span ng-click="see()">{{ test_form.$error }}</span>
</form>
```

input 控件，它还有一些扩展，这些扩展有些有自己的属性：

- **input type="number"** 多了 **number** 错误类型，多了 **max**，**min** 属性。
- **input type="url"** 多了 **url** 错误类型。
- **input type="email"** 多了 **email** 错误类型。

7.5.3. checkbox

它也算是 **input** 的扩展，不过，它没有验证相关的东西，只有选中与不选中两个值：

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="checkbox" name="a" ng-model="a" ng-true-value="AA" ng-false-value="BB" />
  <span>{{ a }}</span>
</form>

var TestCtrl = function($scope) {
  $scope.a = 'AA';
}
```

两点：

1. **controller** 要初始化变量值。
2. **controller** 中的初始化值会关系到控件状态（双向绑定）。

7.5.4. radio

也是 **input** 的扩展。和 **checkbox** 一样，但它只有一个值了：

```
<form name="test_form" ng-controller="TestCtrl">
  <input type="radio" name="a" ng-model="a" value="AA" />
  <input type="radio" name="a" ng-model="a" value="BB" />
  <span>{{ a }}</span>
</form>
```

7.5.5. textarea

同 **input**。

7.5.6. select

这是一个比较牛B的控件。它里面的一个叫做 **ng-options** 的属性用于数据呈现。

对于给定列表时的使用。

最简单的使用方法， **x for x in list**：

```
<form name="test_form" ng-controller="TestCtrl" ng-init="o=[0,1,2,3]; a=o[1];">
  <select ng-model="a" ng-options="x for x in o" ng-change="show()">
```

```

    <option value="">可以加这个空值</option>
  </select>
</form>

<script type="text/javascript">
var TestCtrl = function($scope) {
  $scope.show = function() {
    console.log($scope.a);
  }
}

angular.bootstrap(document, documentElement);
</script>

```

在 **\$scope** 中，**select** 绑定的变量，其值和普通的 **value** 无关，可以是一个对象：

```

<form name="test_form" ng-controller="TestCtrl"
  ng-init="o=[{name: 'AA'}, {name: 'BB'}]; a=o[1];">
  <select ng-model="a" ng-options="x.name for x in o" ng-change="show()">
  </select>
</form>

```

显示与值分别指定，**x.v as x.name for x in o**：

```

<form name="test_form" ng-controller="TestCtrl"
  ng-init="o=[{name: 'AA', v: '00'}, {name: 'BB', v: '11'}]; a=o[1].v;">
  <select ng-model="a" ng-options="x.v as x.name for x in o" ng-change="show()">
  </select>
</form>

```

加入分组的，**x.name group by x.g for x in o**：

```

<form name="test_form" ng-controller="TestCtrl"
  ng-init="o=[{name: 'AA', g: '00'}, {name: 'BB', g: '11'}, {name: 'CC', g: '00'}];
a=o[1];">
  <select ng-model="a" ng-options="x.name group by x.g for x in o" ng-change="show()">
  </select>
</form>

```

分组了还分别指定显示与值的，**x.v as x.name group by x.g for x in o**：

```

<form name="test_form" ng-controller="TestCtrl" ng-init="o=[{name: 'AA', g: '00', v: '='},
{name: 'BB', g: '11', v: '+'}, {name: 'CC', g: '00', v: '!'}]; a=o[1].v;">
  <select ng-model="a" ng-options="x.v as x.name group by x.g for x in o" ng-change="show()">
  </select>
</form>

```

如果参数是对象的话，基本也是一样的，只是把遍历的对象改成 **(key, value)**：

```

<form name="test_form" ng-controller="TestCtrl" ng-init="o={a: 0, b: 1}; a=o.a;">
  <select ng-model="a" ng-options="k for (k, v) in o" ng-change="show()">
  </select>
</form>

<form name="test_form" ng-controller="TestCtrl"
  ng-init="o={a: {name: 'AA', v: '00'}, b: {name: 'BB', v: '11'}}; a=o.a.v;">
  <select ng-model="a" ng-options="v.v as v.name for (k, v) in o" ng-change="show()">
  </select>

```

```

</form>

<form name="test_form" ng-controller="TestCtrl"
      ng-init="o={a: {name: 'AA', v: '00', g: '=='}, b: {name: 'BB', v: '11', g: '=='}};
a=o.a;">
  <select ng-model="a" ng-options="v.name group by v.g for (k, v) in o" ng-change="show()">
  </select>
</form>

<form name="test_form" ng-controller="TestCtrl"
      ng-init="o={a: {name: 'AA', v: '00', g: '=='}, b: {name: 'BB', v: '11', g: '=='}};
a=o.a.v;">
  <select ng-model="a" ng-options="v.v as v.name group by v.g for (k, v) in o" ng-
change="show()">
  </select>
</form>

```

8. 模板中的过滤器

这里说的过滤器，是用于对数据的格式化，或者筛选的函数。它们可以直接在模板中通过一种语法使用。对于常用功能来说，是很方便的一种机制。

多个过滤器之间可以直接连续使用。

8.1. 排序 orderBy

orderBy 是一个排序用的过滤器标签。它可以像 **sort** 函数那样支持一个排序函数，也可以简单地指定一个属性名进行操作：

```

<div ng-controller="TestCtrl">
  {{ data | orderBy: 'age' }} <br />
  {{ data | orderBy: '-age' }} <br />
  {{ data | orderBy: '-age' | limitTo: 2 }} <br />
  {{ data | orderBy: ['-age', 'name'] }} <br />
</div>

<script type="text/javascript">
var TestCtrl = function($scope) {
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];
}

angular.bootstrap(document, [TestCtrl]);
</script>

```

8.2. 过滤列表 filter

filter 是一个过滤内容的标签。

如果参数是一个字符串，则列表成员中的任意属性值中有这个字符串，即为满足条件（忽略大小

写) :

```
<div ng-controller="TestCtrl">
  {{ data | filter: 'b' }} <br />
  {{ data | filter: '!B' }} <br />
</div>

<script type="text/javascript">
var TestCtrl = function($scope) {
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];
}

angular.bootstrap(document.documentElement);
</script>
```

可以使用对象, 来指定属性名, **\$** 表示任意属性:

```
{{ data | filter: {name: 'A'} }} <br />
{{ data | filter: {$: '3'} }} <br />
{{ data | filter: {$: '!3'} }} <br />
```

自定义的过滤函数也支持:

```
<div ng-controller="TestCtrl">
  {{ data | filter: f }} <br />
</div>

<script type="text/javascript">
var TestCtrl = function($scope) {
  $scope.data = [
    {name: 'B', age: 4},
    {name: 'A', age: 1},
    {name: 'D', age: 3},
    {name: 'C', age: 3},
  ];

  $scope.f = function(e) {
    return e.age > 2;
  }
}

angular.bootstrap(document.documentElement);
</script>
```

8.3. 其它

时间戳格式化 **date** :

```
<div ng-controller="TestCtrl">
  {{ a | date: 'yyyy-MM-dd HH:mm:ss' }}
</div>
```

```

</div>

<script type="text/javascript">
var TestCtrl = function($scope) {
    $scope.a = ((new Date().valueOf()));
}

angular.bootstrap(document.documentElement);
</script>

```

列表截取 **limitTo** , 支持正负数:

```

{{ [1,2,3,4,5] | limitTo: 2 }}
{{ [1,2,3,4,5] | limitTo: -3 }}

```

大小写 **lowercase** , **uppercase** :

```

{{ 'abc' | uppercase }}
{{ 'Abc' | lowercase }}

```

8.4. 例子: 表头排序

```

1  <div ng-controller="TestCtrl">
2    <table>
3      <tr>
4        <th ng-click="f='name'; rev=!rev">名字</th>
5        <th ng-click="f='age'; rev=!rev">年龄</th>
6      </tr>
7
8      <tr ng-repeat="o in data | orderBy: f : rev">
9        <td>{{ o.name }}</td>
10       <td>{{ o.age }}</td>
11     </tr>
12   </table>
13 </div>
14
15 <script type="text/javascript">
16 var TestCtrl = function($scope) {
17   $scope.data = [
18     {name: 'B', age: 4},
19     {name: 'A', age: 1},
20     {name: 'D', age: 3},
21     {name: 'C', age: 3},
22   ];
23 }
24
25 angular.bootstrap(document.documentElement);
26 </script>

```

8.5. 例子: 搜索

```

<div ng-controller="TestCtrl" ng-init="s=data[0].name; q=''">
  <div>
    <span>查找: </span> <input type="text" ng-model="q" />
  </div>
  <select ng-multiple="true" ng-model="s"
    ng-options="o.name as o.name + '(' + o.age + ')" for o in data | filter: {name: q} |

```

```
orderBy: ['age', 'name'] ">
</select>
</div>

<script type="text/javascript">
var TestCtrl = function($scope) {
    $scope.data = [
        {name: 'B', age: 4},
        {name: 'A', age: 1},
        {name: 'D', age: 3},
        {name: 'C', age: 3},
    ];
}

angular.bootstrap(document.documentElement);
</script>
```

9. 锚点路由

准确地说，这应该叫对 **hashchange** 事件的处理吧。

就是指 URL 中的锚点部分发生变化时，触发预先定义的业务逻辑。比如现在是 `/test#/x`，锚点部分的值为 `#` 后的 `/x`，它就对应了一组处理逻辑。当这部分变化时，比如变成了 `/test#/t`，这时页面是不会刷新的，但是它可以触发另外一组处理逻辑，来做一些事，也可以让页面发生变化。

这种机制对于复杂的单页面来说，无疑是一种强大的业务切分手段。就算不是复杂的单页面应用，在普通页面上善用这种机制，也可以让业务逻辑更容易控制。

ng 提供了完善的锚点路由功能，虽然目前我觉得相当重要的一个功能还有待完善（后面会说），但目前这功能的几部分内容，已经让我思考了很多种可能性了。

ng 中的锚点路由功能是由几部分 API 共同完成的一整套方案。这其中包括了路由定义，参数定义，业务处理等。

9.1. 路由定义

要使用锚点路由功能，需要在先定义它。目前，对于定义的方法，我个人只发现在“初始化”阶段可以通过 **\$routeProvider** 这个服务来定义。

在定义一个 **app** 时可以定义锚点路由：

```
<html ng-app="ngView">
... ..

<div ng-view></div>

<script type="text/javascript">

angular.module('ngView', [],
    function($routeProvider) {
        $routeProvider.when('/test',
            {
                template: 'test',
            }
        )
    }
);
```

```
    );  
  }  
);  
  
</script>
```

首先看 `ng-view` 这个 **directive**，它是一个标记“锚点作用区”的指令。目前页面上只能有一个“锚点作用区”。有人已经提了，“多个可命名”的锚点作用区的代码到官方，但是目前官方还没有接受合并，我觉得多个作用区这个功能是很重要的，希望下个发布版中能有。

锚点作用区的功能，就是让锚点路由定义时的那些模板，**controller** 等，它们产生的 **HTML** 代码放在作用区内。

比如上面的代码，当你刚打开页面时，页面是空白的。你手动访问 `/#/test` 就可以看到页面上出现了 **'test'** 的字样。

在 `angular.bootstrap()` 时也可以定义：

```
angular.bootstrap(document.documentElement, [  
  function($routeProvider){  
    $routeProvider.when('/test',  
      {  
        template: 'test'  
      }  
    );  
  }  
]);
```

9.2. 参数定义

在作路由定义时，可以匹配一个规则，规则中可以定义路径中的某些部分作为参数之用，然后使用 **\$routeParams** 服务获取到指定参数。比如 `/#/book/test` 中，**test** 作为参数传入到 **controller** 中：

```
<div ng-view></div>  
  
<script type="text/javascript">  
  
angular.module('ngView', [],  
  function($routeProvider){  
    $routeProvider.when('/book/:title',  
      {  
        template: '{{ title }}',  
        controller: function($scope, $routeParams){  
          $scope.title = $routeParams.title;  
        }  
      }  
    );  
  }  
);  
  
</script>
```

访问： `/#/book/test`

不需要预定义模式，也可以像普通 GET 请求那样获取到相关参数：

```
angular.module('ngView', [],
function($routeProvider){
  $routeProvider.when('/book',
  {
    template: '{{ title }}',
    controller: function($scope, $routeParams){
      $scope.title = $routeParams.title;
    }
  }
);
});
```

访问： `/#/book?title=test`

9.3. 业务处理

简单来说，当一个锚点路由定义被匹配时，会根据模板生成一个 `$scope`，同时相应的一个 `controller` 就会被触发。最后模板的结果会被填充到 `ng-view` 中去。

从上面的例子中可以看到，最直接的方式，我们可以在模板中双向绑定数据，而数据的来源，在 `controller` 中控制。在 `controller` 中，又可以使用到像 `$scope`，`$routeParams` 这些服务。

这里先提一下另外一种与锚点路由相关的服务，`$route`。这个服务里锚点路由在定义时，及匹配过程中的信息。比如我们搞怪一下：

```
angular.module('ngView', [],
function($routeProvider){
  $routeProvider.when('/a',
  {
    template: '{{ title }}',
    controller: function($scope){
      $scope.title = 'a';
    }
  }
);

$routeProvider.when('/b',
{
  template: '{{ title }}',
  controller: function($scope, $route){
    console.log($route);
    $route.routes['/a'].controller($scope);
  }
});
});
```

回到锚点定义的业务处理中来。我们可以以字符串形式写模板，也可以直接引用外部文件作为模板：

```
angular.module('ngView', [],
function($routeProvider){
```

```
$routeProvider.when('/test',
{
  templateUrl: 'tpl.html',
  controller: function($scope) {
    $scope.title = 'a';
  }
});
```

tpl.html 中的内容是：

```
{{ title }}
```

这样的话，模板可以预定义，也可以很复杂了。

现在暂时忘了模板吧，因为前面提到的，当前 **ng-view** 不能有多多的限制，模板的渲染机制局限性还是很大的。不过，反正会触发一个 **controller**，那么在函数当中我们可以尽量地干自己喜欢的事：

```
angular.module('ngView', [],
function($routeProvider) {
  $routeProvider.when('/test',
  {
    template: ' {} ',
    controller: function() {
      $('div').first().html('<b>OK</b>');
    }
  });
});
```

那个空的 **template** 不能省，否则 **controller** 不会被触发。

10. 定义模板变量标识标签

由于下面涉及动态内容，所以我打算起一个后端服务来做。但是我发现我使用的 **Tornado** 框架的模板系统，与 **ng** 的模板系统，都是使用 **{{ }}** 这对符号来定义模板表达式的，这太悲剧了，不过幸好 **ng** 已经提供了修改方法：

```
angular.bootstrap(document.documentElement,
[function($interpolateProvider) {
  $interpolateProvider.startSymbol('<[');
  $interpolateProvider.endSymbol('<]>');
}]);
```

使用 **\$interpolateProvider** 服务即可。

11. AJAX

ng 提供了基本的 **AJAX** 封装，你直接面对 **promise** 对象，使用起来还是很方便的。

11.1. HTTP 请求

基本的操作由 **\$http** 服务提供。它的使用很简单，提供一些描述请求的参数，请求就出去了，然后返回一个扩充了 **success** 方法和 **error** 方法的 **promise** 对象（下节介绍），你可以在这个对象中添加需要的回调函数。

```
var TestCtrl = function($scope, $http){
  var p = $http({
    method: 'GET',
    url: '/json'
  });
  p.success(function(response, status, headers, config){
    $scope.name = response.name;
  });
}
```

\$http 接受的配置项有：

- **method** 方法
- **url** 路径
- **params** GET请求的参数
- **data** post请求的参数
- **headers** 头
- **transformRequest** 请求预处理函数
- **transformResponse** 响应预处理函数
- **cache** 缓存
- **timeout** 超时毫秒，超时的请求会被取消
- **withCredentials** 跨域安全策略的一个东西

其中的 **transformRequest** 和 **transformResponse** 及 **headers** 已经有定义的，如果自定义则会覆盖默认定义：

```
1  var $config = this.defaults = {
2    // transform incoming response data
3    transformResponse: [function(data) {
4      if (isString(data)) {
5        // strip json vulnerability protection prefix
6        data = data.replace(PROTECTION_PREFIX, '');
7        if (JSON_START.test(data) && JSON_END.test(data))
8          data = fromJson(data, true);
9      }
10     return data;
11   }],
12
13   // transform outgoing request data
14   transformRequest: [function(d) {
15     return isObject(d) && !isFile(d) ? toJson(d) : d;
16   }],
17
18   // default headers
19   headers: {
20     common: {
21       'Accept': 'application/json, text/plain, */*',
22       'X-Requested-With': 'XMLHttpRequest'
23     },
24     post: {'Content-Type': 'application/json;charset=utf-8'},
```

```
25     put:  {'Content-Type': 'application/json;charset=utf-8'}
26   }
27   };
```

注意它默认的 **POST** 方法出去的 **Content-Type**

对于几个标准的 HTTP 方法，有对应的 **shortcut**：

- `$http.delete(url, config)`
- `$http.get(url, config)`
- `$http.head(url, config)`
- `$http.jsonp(url, config)`
- `$http.post(url, data, config)`
- `$http.put(url, data, config)`

注意其中的 **JSONP** 方法，在实现上会在页面中添加一个 `script` 标签，然后放出一个 **GET** 请求。你自己定义的，匿名回调函数，会被 **ng** 自己给一个全局变量。在定义请求，作为 **GET** 参数，你可以使用 `JSON_CALLBACK` 这个字符串来暂时代替回调函数名，之后 **ng** 会为你替换成真正的函数名：

```
var p = $http({
  method: 'JSONP',
  url: '/json',
  params: {callback: 'JSON_CALLBACK'}
});
p.success(function(response, status, headers, config){
  console.log(response);
  $scope.name = response.name;
});
```

\$http 有两个属性：

- **defaults** 请求的全局配置
- **pendingRequests** 当前的请求队列状态

```
$http.defaults.transformRequest = function(data) {console.log('here'); return data;}
console.log($http.pendingRequests);
```

11.2. 广义回调管理

和其它框架一样，**ng** 提供了广义的异步回调管理的机制。**\$http** 服务是在其之上封装出来的。这个机制就是 **ng** 的 **\$q** 服务。

不过 **ng** 的这套机制总的来说实现得比较简单，按官方的说法，够用了。

使用的方法，基本上是：

- 通过 **\$q** 服务得到一个 **deferred** 实例
- 通过 **deferred** 实例的 **promise** 属性得到一个 **promise** 对象
- **promise** 对象负责定义回调函数
- **deferred** 实例负责触发回调

```
var TestCtrl = function($q){
  var defer = $q.defer();
  var promise = defer.promise;
```



```
promise.then(function(data) {console.log('ok, ' + data)},
              function(data) {console.log('error, ' + data)});
//defer.reject('xx');
defer.resolve('xx');
```

了解了上面的东西，再分别看 **\$q**，**deferred**，**promise** 这三个东西。

11.2.1. \$q

\$q 有四个方法：

- **\$q.all()** 合并多个 **promise**，得到一个新的 **promise**
- **\$q.defer()** 返回一个 **deferred** 对象
- **\$q.reject()** 包装一个错误，以使回调链能正确处理下去
- **\$q.when()** 返回一个 **promise** 对象

\$q.all() 方法适用于并发场景很合适：

```
var TestCtrl = function($q, $http){
  var p = $http.get('/json', {params: {a: 1}});
  var p2 = $http.get('/json', {params: {a: 2}});
  var all = $q.all([p, p2]);
  p.success(function(res) {console.log('here')});
  all.then(function(res) {console.log(res[0])});
}
```

\$q.reject() 方法是在你捕捉异常之后，又要把这个异常在回调链中传下去时使用：

要理解这东西，先看看 **promise** 的链式回调是如何运作的，看下面两段代码的区别：

```
var defer = $q.defer();
var p = defer.promise;
p.then(
  function(data) {return 'xxx'}
);
p.then(
  function(data) {console.log(data)}
);
defer.resolve('123');
```

```
var defer = $q.defer();
var p = defer.promise;
var p2 = p.then(
  function(data) {return 'xxx'}
);
p2.then(
  function(data) {console.log(data)}
);
defer.resolve('123');
```

从模型上看，前者是“并发”，后者才是“链式”。

而 **\$q.reject()** 的作用就是触发后链的 **error** 回调：

```
var defer = $q.defer();
var p = defer.promise;
```

```
p.then(
  function(data) {return data},
  function(data) {return $q.reject(data)}
).
then(
  function(data) {console.log('ok, ' + data)},
  function(data) {console.log('error, ' + data)}
)
defer.reject('123');
```

最后的 `$q.when()` 是把数据封装成 `promise` 对象:

```
var p = $q.when(0, function(data) {return data},
                function(data) {return data});

p.then(
  function(data) {console.log('ok, ' + data)},
  function(data) {console.log('error, ' + data)}
);
```

11.2.2. deferred

`deferred` 对象有两个方法一个属性。

- `promise` 属性就是返回一个 `promise` 对象的。
- `resolve()` 成功回调
- `reject()` 失败回调

```
var defer = $q.defer();
var promise = defer.promise;
promise.then(function(data) {console.log('ok, ' + data)},
            function(data) {console.log('error, ' + data)});
//defer.reject('xx');
defer.resolve('xx');
```

11.2.3. promise

`promise` 对象只有 `then()` 一个方法, 注册成功回调函数和失败回调函数, 再返回一个 `promise` 对象, 以用于链式调用。

12. 工具函数

12.1. 上下文绑定

`angular.bind` 是用来进行上下文绑定, 参数动态绑定的工具函数。

```
var f = angular.bind({a: 'xx'},
  function() {
    console.log(this.a);
  }
);
f();
```

参数动态绑定:

```
var f = function(x) {console.log(x)}
angular.bind({}, f, 'x')();
```

12.2. 对象处理

对象复制: **angular.copy()**

```
var a = { 'x': '123' };
var b = angular.copy(a);
a.x = '456';
console.log(b);
```

对象聚合: **angular.extend()**

```
var a = { 'x': '123' };
var b = { 'xx': '456' };
angular.extend(b, a);
console.log(b);
```

空函数: **angular.noop()**

大小写转换: **angular.lowercase()** 和 **angular.uppercase()**

JSON转换: **angular.fromJson()** 和 **angular.toJson()**

遍历: **angular.forEach()** , 支持列表和对象:

```
var l = {a: '1', b: '2'};
angular.forEach(l, function(v, k) {console.log(k + ': ' + v)});

var l = ['a', 'b', 'c'];
angular.forEach(l, function(v, i, o) {console.log(v)});

var context = { 't': 'xx' };
angular.forEach(l, function(v, i, o) {console.log(this.t)}, context);
```

12.3. 类型判定

- angular.isArray
- angular.isDate
- angular.isDefined
- angular.isElement
- angular.isFunction
- angular.isNumber
- angular.isObject
- angular.isString
- angular.isUndefined

13. 其它服务

13.1. 日志

ng 提供 **\$log** 这个服务用于向终端输出相关信息:

- error()

- `info()`
- `log()`
- `warn()`

```
var TestCtrl = function($log) {
  $log.error('error');
  $log.info('info');
  $log.log('log');
  $log.warn('warn');
}
```

13.2. 缓存

`ng` 提供了一个简单封装了缓存机制 `$cacheFactory`，可以用来作为数据容器：

```
var TestCtrl = function($scope, $cacheFactory) {
  $scope.cache = $cacheFactory('s_' + $scope.$id, {capacity: 3});

  $scope.show = function() {
    console.log($scope.cache.get('a'));
    console.log($scope.cache.info());
  }

  $scope.set = function() {
    $scope.cache.put((new Date()).valueOf(), 'ok');
  }
}
```

调用时，第一个参数是 `id`，第二个参数是配置项，目前支持 `capacity` 参数，用以设置缓存能容留的最大条目数。超过这个个数，则自动清除较旧的条目。

缓存实例的方法：

- `info()` 获取 `id`，`size` 信息
- `put(k, v)` 设置新条目
- `get(k)` 获取条目
- `remove(k)` 删除条目
- `removeAll()` 删除所有条目
- `destroy()` 删除对本实例的引用

`$http` 的调用当中，有一个 `cache` 参数，值为 `true` 时为自动维护的缓存。值也可以设置为一个 `cache` 实例。

13.3. 计时器

`$timeout` 服务是 `ng` 对 `window.setTimeout()` 的封装，它使用 `promise` 统一了计时器的回调行为：

```
var TestCtrl = function($timeout) {
  var p = $timeout(function() {console.log('haha')}, 5000);
  p.then(function() {console.log('x')});
  //$timeout.cancel(p);
}
```

使用 `$timeout.cancel()` 可以取消计时器。

13.4. 表达式函数化

\$parse 这个服务，为 **js** 提供了类似于 **Python** 中 **@property** 的能力：

```
var TestCtrl = function($scope, $parse){
  $scope.get_name = $parse('name');
  $scope.show = function(){console.log($scope.get_name($scope))}
  $scope.set = function(){$scope.name = '123'}
}
```

\$parse 返回一个函数，调用这个函数时，可以传两个参数，第一个作用域，第二个是变量集，后者常用于覆盖前者的变量：

```
var get_name = $parse('name');
var r = get_name({name: 'xx'}, {name: 'abc'});
console.log(r);
```

\$parse 返回的函数，也提供了相应的 **assign** 功能，可以为表达式赋值（如果可以的话）：

```
var get_name = $parse('name');
var set_name = get_name.assign;
var r = get_name({name: 'xx'}, {name: 'abc'});
console.log(r);

var s = {}
set_name(s, '123');
var r = get_name(s);
console.log(r);
```

13.5. 模板单独使用

ng 中的模板是很重要，也很强大的一个机制，自然少不了单独运用它的方法。不过，即使是单独使用，也是和 **DOM** 紧密相关的程度：

- 定义时必须是有 **HTML** 标签包裹的，这样才能创建 **DOM** 节点
- 渲染时必须传入 **\$scope**

之后使用 **\$compile** 就可以得到一个渲染好的节点对象了。当然，**\$compile** 还要做其它一些工作，指令处理什么的。

```
var TestCtrl = function($scope, $element, $compile){
  $scope.a = '123';
  $scope.set = function(){
    var tpl = $compile('<p>hello {{ a }}</p>');
    var e = tpl($scope);
    $element.append(e);
  }
}
```

14. 自定义模块和服务

14.1. 模块和服务的概念与关系

总的来说，模块是组织业务的一个框框，在一个模块当中定义多个服务。当你引入了一个模块的时候，就可以使用这个模块提供的一种或多种服务了。

比如 **AngularJS** 本身的一个默认模块叫做 **ng**，它提供了 **\$http**，**\$q** 等等服务。

服务只是模块提供的多种机制中的一种，其它的还有命令（**directive**），过滤器（**filter**），及其它配置信息。

然后在额外的 **js** 文件中有一个附加的模块叫做 **ngResource**，它提供了一个 **\$resource** 服务。

定义时，我们可以在已有的模块中新定义一个服务，也可以先新定义一个模块，然后在新模块中定义新服务。

使用时，模块是需要显式地的声明依赖（引入）关系的，而服务则可以让 **ng** 自动地做注入，然后直接使用。

14.2. 定义模块

定义模块的方法是使用 **angular.module**。调用时声明了对其它模块的依赖，并定义了“初始化”函数。

```
var my_module = angular.module('MyModule', [], function() {  
    console.log('here');  
});
```

这段代码定义了一个叫做 **MyModule** 的模块，**my_module** 这个引用可以在接下来做其它的一些事，比如定义服务。

14.3. 定义服务

服务本身是一个任意的对象。但是 **ng** 提供服务的过程涉及它的依赖注入机制。在这里呢，就要先介绍一下叫 **provider** 的东西。

简单来说，**provider** 是被“注入控制器”使用的一个对象，注入机制通过调用一个 **provider** 的 **\$get()** 方法，把得到的东西作为参数进行相关调用（比如把得到的服务作为一个 **Controller** 的参数）。

在这里“服务”的概念就比较不明确，对使用而言，服务仅指 **\$get()** 方法返回的东西，但是在整体机制上，服务又要指提供了 **\$get()** 方法的整个对象。

```
//这是一个provider  
var pp = function() {  
    this.$get = function() {  
        return {'haha': '123'};  
    }  
}  
  
//我在模块的初始化过程当中，定义了一个叫 PP 的服务  
var app = angular.module('Demo', [], function($provide) {  
    $provide.provider('PP', pp);  
});  
  
//PP服务实际上就是 pp 这个 provider 的 $get() 方法返回的东西  
app.controller('TestCtrl',
```

```
function($scope, PP) {
    console.log(PP);
}
);
```

上面的代码是一种定义服务的方法，当然，**ng** 还有相关的 **shortcut**，**ng** 总有很多 **shortcut**。

第一个是 **factory** 方法，由 **\$provide** 提供，**module** 的 **factory** 是一个引用，作用一样。这个方法直接把一个函数当成是一个对象的 **\$get()** 方法，这样你就不用显式地定义一个 **provider** 了：

```
var app = angular.module('Demo', [], function($provide) {
    $provide.factory('PP', function() {
        return { 'hello': '123' };
    });
});
app.controller('TestCtrl', function($scope, PP) { console.log(PP) });
```

在 **module** 中使用：

```
var app = angular.module('Demo', [], function() { });
app.factory('PP', function() { return { 'abc': '123' } });
app.controller('TestCtrl', function($scope, PP) { console.log(PP) });
```

第二个是 **service** 方法，也是由 **\$provide** 提供，**module** 中有对它的同名引用。**service** 和 **factory** 的区别在于，前者是要求提供一个“构造方法”，后者是要求提供 **\$get()** 方法。意思就是，前者一定是得到一个 **object**，后者可以是一个数字或字符串。它们的关系大概是：

```
var app = angular.module('Demo', [], function() { });
app.service = function(name, constructor) {
    app.factory(name, function() {
        return (new constructor());
    });
}
```

这里插一句，js 中 **new** 的作用，以 **new a()** 为例，过程相当于：

1. 创建一个空对象 **obj**
2. 把 **obj** 绑定到 **a** 函数的上下文当中（即 **a** 中的 **this** 现在指向 **obj**）
3. 执行 **a** 函数
4. 返回 **obj**

service 方法的使用就很简单了：

```
var app = angular.module('Demo', [], function() { });
app.service('PP', function() {
    this.abc = '123';
});
app.controller('TestCtrl', function($scope, PP) { console.log(PP) });
```

14.4. 引入模块并使用服务

结合上面的“定义模块”和“定义服务”，我们可以方便地组织自己的额外代码：

```
angular.module('MyModule', [], function($provide) {
    $provide.factory('S1', function() {
```

```
        return 'I am S1';
    });
    $provide.factory('S2', function() {
        return {see: function() {return 'I am S2'}}
    });
});

var app = angular.module('Demo', ['MyModule'], angular.noop);
app.controller('TestCtrl', function($scope, S1, S2) {
    console.log(S1)
    console.log(S2.see())
});
```

15. 附加模块 ngResource

15.1. 使用引入与整体概念

ngResource 这个是 **ng** 官方提供的一个附加模块。附加的意思就是，如果你打算用它，那么你需要引入一人单独的 **js** 文件，然后在声明“根模块”时注明依赖的 **ngResource** 模块，接着就可以使用它提供的 **\$resource** 服务了。完整的过程形如：

```
<!DOCTYPE html>
<html ng-app="Demo">
<head>
<meta charset="utf-8" />
<title>AngularJS</title>
<script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js"></script>
<script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular-resource.js"></script>
</head>
<body>

    <div ng-controller="TestCtrl"></div>

<script type="text/javascript" charset="utf-8">

var app = angular.module('Demo', ['ngResource'], angular.noop);
app.controller('TestCtrl', function($scope, $resource) {
    console.log($resource);
});

</script>
</body>
</html>
```

\$resource 服务，整体上来说，比较像是使用类似 **ORM** 的方式来包装了 **AJAX** 调用。区别就是 **ORM** 是操作数据库，即拼出 **SQL** 语句之后，作 **execute** 方法调用。而 **\$resource** 的方式是构造出 **AJAX** 请求，然后发出请求。同时，**AJAX** 请求是需要回调处理的，这方面，**\$resource** 的机制可以使你在一些时候省掉回调处理，当然，是否作回调处理在于业务情形及容错需求了。

使用上 **\$resource** 分成了“类”与“实例”这两个层面。一般地，类的方法调用就是直观的调用形式，通常会返回一个对象，这个对象即为“实例”。

“实例”贯穿整个服务的使用过程。“实例”的数据是填充方式，即因为异步关系，回调函数没有执行

时，实例已经存在，只是可能它还没有相关数据，回调执行之后，相关数据被填充到实例对象当中。实例的方法一般就是在类方法名前加一个 **\$**，调用上，根据定义，实例数据可能会做一些自动的参数填充，这点是区别实例与类的调用上的不同。

好吧，上面这些话可能需要在看了接下来的内容之后再回过头来理解。

15.2. 基本定义

就像使用 ORM 一般要先定义 Model 一样，使用 **\$resource** 需要先定义“资源”，也就是先定义一些 HTTP 请求。

在业务场景上，我们假设为，我们需要操作“书”这个实体，包括创建**create**，获取详情**read**，修改**update**，删除**delete**，批量获取**multi**，共五个操作方法。实体属性有：唯一标识**id**，标题**title**，作者**author**。

我们把这些操作定义成 **\$resource** 的资源：

```
var app = angular.module('Demo', ['ngResource'], angular.noop);
app.controller('BookCtrl', function($scope, $resource) {
  var actions = {
    create: {method: 'POST', params: {_method: 'create'}},
    read: {method: 'POST', params: {_method: 'read'}},
    update: {method: 'POST', params: {_method: 'update'}},
    delete: {method: 'POST', params: {_method: 'delete'}},
    multi: {method: 'POST', params: {_method: 'multi'}}
  }
  var Book = $resource('/book', {}, actions);
});
```

定义是使用使用 **\$resource** 这个函数就可以了，它接受三个参数：

- url
- 默认的params（这里的 params 即是 GET 请求的参数，POST 的参数单独叫做“postData”）
- 方法映射

方法映射是以方法名为 key，以一个对象为 value，这个 value 可以有三个成员：

- method, 请求方法, 'GET', 'POST', 'PUT', 'DELETE' 这些
- params, 默认的 GET 参数
- isArray, 返回的数据是不是一个列表

15.3. 基本使用

在定义了资源之后，我们看如果使用这些资源，发出请求：

```
var book = Book.read({id: '123'}, function(response) {
  console.log(response);
});
```

这里我们进行 **Book** 的“类”方法调用。在方法的使用上，根据官方文档：

```
HTTP GET “class” actions: Resource.action([parameters], [success], [error])
non-GET “class” actions: Resource.action([parameters], postData, [success], [error])
```

```
non-GET instance actions: instance.$action([parameters], [success], [error])
```

我们这里是第二种形式，即类方法的非 GET 请求。我们给的参数会作为 `postData` 传递。如果我们需要 GET 参数，并且还需要一个错误回调，那么：

```
var book = Book.read({get: 'haha'}, {id: '123'},
  function(response) {
    console.log(response);
  },
  function(error) {
    console.log(error);
  }
);
```

调用之后，我们会立即得到的 `book`，它是 `Book` 类的一个实例。这里所谓的实例，实际上就是先把所有的 `action` 加一个 `$` 前缀放到一个空对象里，然后把发出的参数填充进去。等请求返回了，把除 `action` 以外的成员删除掉，再把请求返回的数据填充到这个对象当中。所以，如果我们这样：

```
var book = Book.read({id: '123'}, function(response) {
  console.log(book);
});
console.log(book)
```

就能看到 `book` 实例的变化过程了。

现在我们得到一个真实的实例，看一下实例的调用过程：

```
//响应的数据是 {result: 0, msg: '', obj: {id: 'xxx'}}
var book = Book.create({title: '测试标题', author: '测试作者'}, function(response) {
  console.log(book);
});
```

可以看到，在请求回调之后，`book` 这个实例的成员已经被响应内容填充了。但是这里有一个问题，我们返回的数据，并不适合一个 `book` 实例。格式先不说，它把 `title` 和 `author` 这些信息都丢了（因为响应只返回了 `id`）。

如果仅仅是格式问题，我们可以通过配置 `$http` 服务来解决（AJAX 请求都要使用 `$http` 服务的）：

```
$http.defaults.transformResponse = function(data) {return angular.fromJson(data).obj};
```

当然，我们也可以自己来解决一下丢信息的问题：

```
var p = {title: '测试标题', author: '测试作者'};
var book = Book.create(p, function(response) {
  angular.extend(book, p);
  console.log(book);
});
```

不过，始终会有一些不方便了。比较正统的方式应该是调节服务器端的响应，让服务器端也具有和前端一样的实例概念，返回的是完整的实例信息。即使这样，你也还要考虑格式的事。

现在我们得到了一个真实的 `book` 实例了，带有 `id` 信息。我们尝试一下实例的方法调用，先回过去头看一下那三种调用形式，对于实例只有第三种形式：

```
non-GET instance actions: instance.$action([parameters], [success], [error])
```

首先解决一个疑问, 如果一个实例是进行一个 **GET** 的调用会怎么样? 没有任何问题, 这当然没有任何问题的, 形式和上面一样。

如何实例是做 **POST** 请求的话, 从形式上看, 我们无法控制请求的 **postData**? 是的, 所有的 **POST** 请求, 其 **postData** 都会被实例数据自动填充, 形式上我们只能控制 **params**。

所以, 如果是在做修改调用的话:

```
book.$update({title: '新标题', author: '测试作者'}, function(response) {  
    console.log(book);  
});
```

这样是没有意义的并且错误的。因为要修改的数据只是作为 **GET** 参数传递了, 而 **postData** 传递的数据就是当前实例的数据, 并没有任何修改。

正确的做法:

```
book.title = '新标题'  
book.$update(function(response) {  
    console.log(book);  
});
```

显然, 这种情况下, 回调都可以省了:

```
book.title = '新标题'  
book.$update();
```

15.4. 定义和使用时的占位量

两方面。一是在定义时, 在其 **URL** 中可以使用变量引用的形式 (类型于定义锚点路由时那样)。第二时定义默认 **params**, 即 **GET** 参数时, 可以定义为引用 **postData** 中的某变量。比如我们这样改一下:

```
var Book = $resource('/book/:id', {}, actions);  
var book = Book.read({id: '123'}, {}, function(response) {  
    console.log(response);  
});
```

在 **URL** 中有一个 **:id**, 表示对 **params** 中 **id** 这个变量的引用。因为 **read** 是一个 **POST** 请求, 根据调用形式, 第一个参数是 **params**, 第二个参数是 **postData**。这样的调用结果就是, 我们会发一个 **POST** 请求到如下地址, **postData** 为空:

```
/book/123?_method=read
```

再看默认的 **params** 中引用 **postData** 变量的形式:

```
var Book = $resource('/book', {id: '@id'}, actions);  
var book = Book.read({title: 'xx'}, {id: '123'}, function(response) {  
    console.log(response);  
});
```

这样会出一个 POST 请求, `postData` 内容中有一个 `id` 数据, 访问的 URL 是:

```
/book?_method=read&id=123&title=xx
```

这两个机制也可以联合使用:

```
var Book = $resource('/book/:id', {id: '@id'}, actions);
var book = Book.read({title: 'xx'}, {id: '123'}, function(response) {
    console.log(response);
});
```

结果就是出一个 POST 请求, `postData` 内容中有一个 `id` 数据, 访问的 URL 是:

```
/book/123?_method=read&title=xx
```

15.5. 实例

`ngResource` 要举一个实例是比较麻烦的事。因为它必须要一个后端来支持, 这里如果我用 Python 写一个简单的后端, 估计要让这个后端跑起来对很多人来说都是问题。所以, 我在几套公共服务的 API 中纠结考察了一番, 最后使用 www.rememberthemilk.com 的 API 来做了一个简单的, 可用的例子。

例子见: <http://zouyesheng.com/demo/ng-resource-demo.html> (可以直接下载看源码)

先说一下 API 的情况。这里的请求调用全是跨域的, 所以交互上全部是使用了 JSONP 的形式。API 的使用有使用签名认证机制, 嗯, js 中直接算 md5 是可行的, 我用了一个现成的库 (但是好像不能处理中文吧)。

这个例子中的 `LoginCtrl` 大家就不用太关心了, 参见官方的文档, 走完流程拿到 `token` 完事。与 `ngResource` 相关的是 `MainCtrl` 中的东西。

其实从这个例子中就可以看出, 目前 `ngResource` 的机制对于服务端返回的数据的格式是严重依赖的, 同时也可以反映出 `$http` 对一些场景根本无法应对的局限。所以, 我现在的想法是理解 `ngResource` 的思想, 真正需要的人自己使用 `jQuery` 重新实现一遍也许更好。这应该也花不了多少时间, `ngResource` 的代码本来不多。

我为什么说 `$http` 在一些场景中有局限呢。在这个例子当中, 所有的请求都需要带一个签名, 签名值是由请求中带的参数根据规则使用 `md5` 方法计算出的值。我找不到一个 `hook` 可以让我在请求出去之前修改这个请求 (添加上签名)。所以在这个例子当中, 我的做法是根据 `ngResource` 的请求最后会使用 `$httpBackend` 这个底层服务, 在 `module` 定义时我自己复制官方的相关代码, 重新定义 `$httpBackend` 服务, 在需要的地方做我自己的修改:

```
script.src = sign_url(url);
```

不错, 我就改了这一句, 但我不得不复制了 50 行官方源码到我的例子中。

另外一个需要说的是对返回数据的处理。因为 `ngResource` 会使用返回的数据直接填充实例, 所以这个数据格式就很重要。

首先, 我们可以使用 `$http.defaults.transformResponse` 来统一处理一下返回的数据, 但是这并不能解决所有问题, 可目前 `ngResource` 并不提供对每一个 `action` 的单独的后处理回调函数项。

除非你的服务端是经过专门的适应性设计的，否则你用 **ngResource** 不可能爽。例子中，我为了获取当前列表的结果，我不得不自己去封装结果：

```
var list_list = List.getList(function() {
    var res = list_list[1];
    while(list_list.length > 0){list_list.pop()};
    angular.forEach(res.list, function(v){
        list_list.push(new List({list: v}));
    });
    $scope.list_list = list_list;
    $scope.show_add = true;
    return;
});
```

16. AngularJS与其它框架的混用(jQuery, Dojo)

这个问题似乎很多人都关心，但是事实是，如果了解了 **ng** 的工作方式，这本来就不是一个问题了。

在我自己使用 **ng** 的过程当中，一直是混用 **jQuery** 的，以前还要加上一个 **Dojo**。只要了解每种框架的工作方式，在具体的代码中每个框架都做了什么事，那么整体上控制起来就不会有问题。

回到 **ng** 上来看，首先对于 **jQuery** 来说，最开始说提到过，在 **DOM** 操作部分，**ng** 与 **jQuery** 是兼容的，如果没有 **jQuery**，**ng** 自己也实现了兼容的部分 **API**。

同时，最开始也提到过，**ng** 的使用最忌讳的一点就是修改 **DOM** 结构——你应该使用 **ng** 的模板机制进行数据绑定，以此来控制 **DOM** 结构，而不是直接操作。换句话说，在不动 **DOM** 结构的这个前提之下，你的数据随便怎么改，随便使用哪个框架来控制都是没问题的，到时如有必要使用 **\$scope.\$digest()** 来通知 **ng** 一下即可。

下面这个例子，我们使用了 **jQuery** 中的 **Deferred** (**\$.ajax** 就是返回一个 **Deferred**)，还使用了 **ng** 的 **\$timeout**，当然是在 **ng** 的结构之下：

```
1  <!DOCTYPE html>
2  <html ng-app="Demo">
3  <head>
4  <meta charset="utf-8" />
5  <title>AngularJS</title>
6  </head>
7  <body>
8
9  <div ng-controller="TestCtrl">
10    <span ng-click="go()">{{ a }}</span>
11  </div>
12
13  <script type="text/javascript">
14    src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
15  </script>
16  <script type="text/javascript">
17    src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
18  </script>
19
20  <script type="text/javascript">
21    var app = angular.module('Demo', [], angular.noop);
22    app.controller('TestCtrl', function($scope, $timeout) {
23      $scope.a = ' 点击我开始' ;
24
```

```

25     var defer = $.Deferred();
26     var f = function() {
27         if($scope.a == '') {$scope.a = '已停止'; return}
28         defer.done(function() {
29             $scope.a.length < 10 ? $scope.a += '>' : $scope.a = '>';
30             $timeout(f, 100);
31         });
32     }
33     defer.done(function() {$scope.a = '>'; f()});
34
35     $scope.go = function() {
36         defer.resolve();
37         $timeout(function() {$scope.a = ''}, 5000);
38     }
39 });
40 </script>
41 </body>
42 </html>

```

再把 Dojo 加进来看与 DOM 结构相关的例子。之前说过，使用 ng 就最好不要手动修改 DOM 结构，但这里说两点：

1. 对于整个页面，你可以只在局部使用 ng，不使用 ng 的地方你可以随意控制 DOM。
2. 如果 DOM 结构有变动，你可以在 DOM 结构定下来之后再初始化 ng。

下面这个例子使用了 AngularJS，jQuery，Dojo：

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8" />
5  <title>AngularJS</title>
6  <link rel="stylesheet"
7      href="http://ajax.googleapis.com/ajax/libs/dojo/1.9.1/dijit/themes/claro/claro.css"
media="screen" />
8  </head>
9  <body class="claro">
10
11  <div ng-controller="TestCtrl" id="test_ctrl">
12
13      <p ng-show="!btn_disable">
14          <button ng-click="change()">调用dojo修改按钮</button>
15      </p>
16
17      <p id="btn_wrapper">
18          <button data-dojo-type="dijit/form/Button" type="button">{{ a }}</button>
19      </p>
20
21      <p>
22          <input ng-model="dialog_text" ng-init="dialog_text='对话框内容'" />
23          <button ng-click="dialog(dialog_text)">显示对话框</button>
24      </p>
25
26      <p ng-show="show_edit_text" style="display: none;">
27          <span>需要编辑的内容:</span>
28          <input ng-model="text" />
29      </p>
30
31      <div id="editor_wrapper">
32          <div data-dojo-type="dijit/Editor" id="editor"></div>

```

```
33     </div>
34
35 </div>
36
37
38 <script type="text/javascript"
39     src="http://ajax.googleapis.com/ajax/libs/dojo/1.9.1/dojo/dojo.js">
40 </script>
41 <script type="text/javascript"
42     src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
43 </script>
44 <script type="text/javascript"
45     src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
46 </script>
47
48 <script type="text/javascript">
49
50 require(['dojo/parser', 'dijit/Editor'], function(parser) {
51     parser.parse($('#editor_wrapper')[0]).then(function() {
52         var app = angular.module('Demo', [], angular.noop);
53
54         app.controller('TestCtrl', function($scope, $timeout) {
55             $scope.a = '我是ng, 也是dojo';
56             $scope.show_edit_text = true;
57
58             $scope.change = function() {
59                 $scope.a = 'DOM结构已经改变(不建议这样做)';
60                 require(['dojo/parser', 'dijit/form/Button', 'dojo/domReady!'],
61                     function(parser) {
62                         parser.parse($('#btn_wrapper')[0]);
63                         $scope.btn_disable = true;
64                     }
65                 );
66             }
67
68             $scope.dialog = function(text) {
69                 require(["dijit/Dialog", "dojo/domReady!"], function(Dialog) {
70                     var dialog = new Dialog({
71                         title: "对话框哦",
72                         content: text,
73                         style: "width: 300px"
74                     });
75                     dialog.show();
76                 });
77             }
78
79             require(['dijit/registry'], function(registry) {
80                 var editor = registry.byId('editor');
81                 $scope.$watch('text', function(new_v) {
82                     editor.setValue(new_v);
83                 });
84             });
85
86         });
87
88         angular.bootstrap(document, ['Demo']);
89     });
90
91 });
92
93 </script>
```

```
94     </body>
95     </html>
```

17. 自定义过滤器

先来回顾一下 ng 中的一些概念：

- **module**，代码的组织单元，其它东西都是在定义在具体的模块中的。
- **app**，业务概念，可能会用到多个模块。
- **service**，仅在数据层面实现特定业务功能的代码封装。
- **controller**，与 DOM 结构相关联的东西，即是一种业务封装概念，又体现了项目组织的层级结构。
- **filter**，改变输入数据的一种机制。
- **directive**，与 DOM 结构相关联的，特定功能的封装形式。

上面的这几个概念基本上就是 ng 的全部。每一部分都可以自由定义，使用时通过各要素的相互配合来实现我们的业务需求。

我们从最开始一致打交道的东西基本上都是 **controller** 层面的东西。在前面，也介绍了 **module** 和 **service** 的自定义。剩下的会介绍 **filter** 和 **directive** 的定义。基本上这几部分的定义形式都是一样的，原理上是通过 **provider** 来做注入形式的声明，在实际操作过程中，又有很多 **shortcut** 式的声明方式。

过滤器的自定义是最简单的，就是一个函数，接受输入，然后返回结果。在考虑过滤器时，我觉得很重要的一点：**无状态**。

具体来说，过滤器就是一个函数，函数的本质含义就是确定的输入一定得到确定的输出。虽然 **filter** 是定义在 **module** 当中的，而且 **filter** 又是在 **controller** 的 DOM 范围内使用的，但是，它和具体的 **module**，**controller**，**scope** 这些概念都没有关系（虽然在这里你可以使用 js 的闭包机制玩些花样），它仅仅是一个函数，而已。换句话说，它没有任何上下文关联的能力。

过滤器基本的定义方式：

```
var app = angular.module('Demo', [], angular.noop);
app.filter('map', function() {
    var filter = function(input) {
        return input + '...';
    };
    return filter;
});
```

上面的代码定义了一个叫做 **map** 的过滤器。使用时：

```
<p>示例数据: {{ a|map }}</p>
```

过滤器也可以带参数，多个参数之间使用 **:** 分割，看一个完整的例子：

```
1 <div ng-controller="TestCtrl">
2 <p>示例数据: {{ a|map:map_value:'>>':'(no)' }}</p>
3 <p>示例数据: {{ b|map:map_value:'>>':'(no)' }}</p>
4 </div>
5
6
7 <script type="text/javascript">
```



```
8
9  var app = angular.module('Demo', [], angular.noop);
10 app.controller('TestCtrl', function($scope) {
11     $scope.map_value = {
12         a: '一',
13         b: '二',
14         c: '三'
15     }
16     $scope.a = 'a';
17 });
18
19 app.filter('map', function() {
20     var filter = function(input, map_value, append, default_value) {
21         var r = map_value[input];
22         if(r === undefined) { return default_value + append }
23         else { return r + append }
24     };
25     return filter;
26 });
27
28 angular.bootstrap(document, ['Demo']);
29 </script>
```

18. 自定义指令directive

这是 ng 最强大的一部分，也是最复杂最让人头疼的部分。

目前我们看到的所谓“模板”系统，只不过是官方实现的几个指令而已。这意味着，通过自定义各种指令，我们不但可以完全定义一套“模板”系统，更可以把 HTML 页面直接打造成为一种 DSL（领域特定语言）。

18.1. 指令的使用

使用指令时，它的名字可以有多种形式，把指令放在什么地方也有多种选择。

通常，指令的定义名是形如 `ngBind` 这样的“camel cased”形式。在使用时，它的引用名可以是：

- `ng:bind`
- `ng_bind`
- `ng-bind`
- `x-ng-bind`
- `data-ng-bind`

你可以根据你自己是否有“HTML validator”洁癖来选择。

指令可以放在多个地方，它们的作用相同：

- `` 作为标签的属性
- `` 作为标签类属性的值
- `<my-dir></my-dir>` 作为标签
- `<!-- directive: my-dir exp -->` 作为注释

这些方式可以使用指令定义中的 `restrict` 属性来控制。

可以看出，指令即可以作为标签使用，也可以作为属性使用。仔细考虑一下，这在类 XML 的结构当

中真算得上是一种神奇的机制。

18.2. 指令的执行过程

ng 中对指令的解析与执行过程是这样的:

- 浏览器得到 HTML 字符串内容, 解析得到 DOM 结构。
- ng 引入, 把 DOM 结构扔给 `$compile` 函数处理:
 - 找出 DOM 结构中有变量占位符
 - 匹配找出 DOM 中包含的所有指令引用
 - 把指令关联到 DOM
 - 关联到 DOM 的多个指令按权重排列
 - 执行指令中的 `compile` 函数 (改变 DOM 结构, 返回 `link` 函数)
 - 得到的所有 `link` 函数组成一个列表作为 `$compile` 函数的返回
- 执行 `link` 函数 (连接模板的 `scope`) 。

18.3. 基本的自定义方法

自定义一个指令可以非常非常的复杂, 但是其基本的调用形式, 同自定义服务大概是相同的:

```
<p show style="font-size: 12px;"></p>

<script type="text/javascript">

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

app.directive('show', function(){
  var func = function($scope, $element, $attrs){
    console.log($scope);
    console.log($element);
    console.log($attrs);
  }
  return func;
  //return {compile: function(){return func}}
});

angular.bootstrap(document, ['Demo']);
</script>
```

如果在 `directive` 中直接返回一个函数, 则这个函数会作为 `compile` 的返回值, 也即是作为 `link` 函数使用。这里说的 `compile` 和 `link` 都是一个指令的组成部分, 一个完整的定义应该返回一个对象, 这个对象包括了多个属性:

- name
- priority
- terminal
- scope
- controller
- require
- restrict
- template
- templateUrl
- replace
- transclude

- compile
- link

上面的每一个属性，都可以单独探讨的。

下面是一个完整的基本的指令定义例子：

```
<code lines>
//失去焦点使用 jQuery 的扩展支持冒泡
app.directive('ngBlur', function($parse){
  return function($scope, $element, $attr){
    var fn = $parse($attr['ngBlur']);
    $element.on('focusout', function(event){
      fn($scope, {$event: event});
    });
  }
});
</code>
```

```
<div code lines>
//失去焦点使用 jQuery 的扩展支持冒泡
app.directive('ngBlur', function($parse){
  return function($scope, $element, $attr){
    var fn = $parse($attr['ngBlur']);
    $element.on('focusout', function(event){
      fn($scope, {$event: event});
    });
  }
});
</div>
```

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('code', function(){
4    var func = function($scope, $element, $attrs){
5
6      var html = $element.text();
7      var lines = html.split('\n');
8
9      //处理首尾空白
10     if(lines[0] == ''){lines = lines.slice(1, lines.length - 1)}
11     if(lines[lines.length-1] == ''){lines = lines.slice(0, lines.length - 1)}
12
13     $element.empty();
14
15     //处理外框
16     (function(){
17       $element.css('clear', 'both');
18       $element.css('display', 'block');
19       $element.css('line-height', '20px');
20       $element.css('height', '200px');
21     })();
22
23     //是否显示行号的选项
24     if('lines' in $attrs){
25       //处理行号
26       (function(){
27         var div = $('<div style="width: %spx; background-color: gray; float: left; text-align: right; padding-right: 5px; margin-right: 10px;"></div>')
28         .replace('%s', String(lines.length).length * 10));
```

```

29     var s = '';
30     angular.forEach(lines, function(_, i) {
31         s += '<pre style="margin: 0;">%s</pre>\n'.replace('%s', i + 1);
32     });
33     div.html(s);
34     $element.append(div);
35     })();
36 }
37
38 //处理内容
39 (function() {
40     var div = $('<div style="float: left;"></div>');
41     var s = '';
42     angular.forEach(lines, function(l) {
43         s += '<span style="margin: 0;">%s</span><br />\n'.replace('%s', l.replace(/\s/g,
44 '<span>&nbsp;</span>'));
45     });
46     div.html(s);
47     $element.append(div);
48     })();
49
50     return {link: func,
51             restrict: 'AE'}; //以元素或属性的形式使用命令
52 });
53
54 angular.bootstrap(document, ['Demo']);

```

上面这个自定义的指令，做的事情就是解析节点中的文本内容，然后修改它，再把生成的新内容填充到节点当中去。其间还涉及了节点属性值 `lines` 的处理。这算是指令中最简单的一种形式。因为它是“一次性使用”，中间没有变量的处理。比如如果节点原来的文本内容是一个变量引用，类似于 `{{ code }}`，那上面的代码就不行了。这种情况麻烦得多。后面会讨论。

18.4. 属性值类型的自定义

官方代码中的 `ng-show` 等算是我说的这种类型。使用时主要是在节点加添加一个属性值以附加额外的功能。看一个简单的例子：

```

<p color="red">有颜色的文本</p>
<color color="red">有颜色的文本</color>

<script type="text/javascript">

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

app.directive('color', function() {
    var link = function($scope, $element, $attrs) {
        $element.css('color', $attrs.color);
    }
    return {link: link,
            restrict: 'AE'};
});

angular.bootstrap(document, ['Demo']);
</script>

```

我们定义了一个叫 `color` 的指令，可以指定节点文本的颜色。但是这个例子还无法像 `ng-show` 那

样工作的，这个例子只能渲染一次，然后就无法根据变量来重新改变显示了。要响应变化，我们需要手工使用 `scope` 的 `$watch` 来处理：

```
1
2   <div ng-controller="TestCtrl">
3     <p color="color">有颜色的文本</p>
4     <p color="'blue'">有颜色的文本</p>
5   </div>
6
7   <script type="text/javascript">
8
9     var app = angular.module('Demo', [], angular.noop);
10
11     app.directive('color', function() {
12       var link = function($scope, $element, $attrs) {
13         $scope.$watch($attrs.color, function(new_v) {
14           $element.css('color', new_v);
15         });
16       }
17       return link;
18     });
19
20     app.controller('TestCtrl', function($scope) {
21       $scope.color = 'red';
22     });
23
24     angular.bootstrap(document, ['Demo']);
25   </script>
```

18.5. Compile的细节

指令的处理过程，是 `ng` 的 `Compile` 过程的一部分，它们也是紧密联系的。继续深入指令的定义方法，首先就要对 `Compile` 的过程做更细致的了解。

前面说过，`ng` 对页面的处理过程：

- 浏览器把 HTML 字符串解析成 DOM 结构。
- `ng` 把 DOM 结构给 `$compile`，返回一个 `link` 函数。
- 传入具体的 `scope` 调用这个 `link` 函数。
- 得到处理后的 DOM，这个 DOM 处理了指令，连接了数据。

`$compile` 最基本的使用方式：

```
var link = $compile('<p>{{ text }}</p>');
var node = link($scope);
console.log(node);
```

上面的 `$compile` 和 `link` 调用时都有额外参数来实现其它功能。先看 `link` 函数，它形如：

```
function(scope[, cloneAttachFn])
```

第二个参数 `cloneAttachFn` 的作用是，表明是否复制原始节点，及对复制节点需要做的处理，下面这个例子说明了它的作用：

```
<div ng-controller="TestCtrl"></div>
<div id="a">A {{ text }}</div>
```

```
<div id="b">B </div>
```

```
app.controller('TestCtrl', function($scope, $compile) {
    var link = $compile($('#a'));

    //true参数表示新建一个完全隔离的scope,而不是继承的child scope
    var scope = $scope.$new(true);
    scope.text = '12345';

    //var node = link(scope, function() {});
    var node = link(scope);

    $('#b').append(node);
});
```

`cloneAttachFn` 对节点的处理是有限制的, 你可以添加 `class`, 但是不能做与数据绑定有关的其它修改(修改了也无效):

```
app.controller('TestCtrl', function($scope, $compile) {
    var link = $compile($('#a'));
    var scope = $scope.$new(true);
    scope.text = '12345';

    var node = link(scope, function(clone_element, scope) {
        clone_element.text(clone_element.text() + ' ...'); //无效
        clone_element.text('{{ text2 }}'); //无效
        clone_element.addClass('new_class');
    });

    $('#b').append(node);
});
```

修改无效的原因是, 像 `{{ text }}` 这种所谓的 `Interpolate` 在 `$compile` 中已经被处理过了, 生成了相关函数(这里起作用的是 `directive` 中的一个 `postLink` 函数), 后面执行 `link` 就是执行了 `$compile` 生成的这些函数。当然, 如果你的文本没有数据变量的引用, 那修改是会有效果的。

前面在说自定义指令时说过, `link` 函数是由 `compile` 函数返回的, 也就像前面说的, 应该把改变 `DOM` 结构的逻辑放在 `compile` 函数中做。

`$compile` 还有两个额外的参数:

```
$compile(element, transclude, maxPriority);
```

`maxPriority` 是指令的权重限制, 这个容易理解, 后面再说。

`transclude` 是一个函数, 这个函数会传递给 `compile` 期间找到的 `directive` 的 `compile` 函数(编译节点的过程中找到了指令, 指令的 `compile` 函数会接受编译时传递的 `transclude` 函数作为其参数)。

但是在实际使用中, 除我们手工在调用 `$compile` 之外, 初始化时的根节点 `compile` 是不会传递这个参数的。

在我们定义指令时, 它的 `compile` 函数是这个样子的:

```
function compile(tElement, tAttrs, transclude) { ... }
```

事实上，`transclude` 的值，就是 `directive` 所在的 **原始** 节点，把原始节点重新做了编译之后得到的 `link` 函数（需要 `directive` 定义时使用 `transclude` 选项），后面会专门演示这个过程。所以，官方文档上也把 `transclude` 函数描述成 `link` 函数的样子（如果自定义的指令只用在自己手动 `$compile` 的环境中，那这个函数的形式是可以随意的）：

```
{function(angular.Scope[, cloneAttachFn]}
```

所以记住，定义指令时，`compile` 函数的第三个参数 `transclude`，就是一个 `link`，装入 `scope` 执行它你就得到了一个节点。

18.6. transclude的细节

`transclude` 有两方面的东西，一个是使用 `$compile` 时传入的函数，另一个是定义指令的 `compile` 函数时接受的一个参数。虽然这里的一出一进本来是相互对应的，但是实际使用中，因为大部分时候不会手动调用 `$compile`，所以，在“默认”情况下，指令接受的 `transclude` 又会是一个比较特殊的函数。

看一个基本的例子：

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

app.directive('more', function() {
  var func = function(element, attrs, transclude) {
    var sum = transclude(1, 2);
    console.log(sum);
    console.log(element);
  }

  return {compile: func,
    restrict: 'E'};
});

app.controller('TestCtrl', function($scope, $compile, $element) {
  var s = '<more>123</more>';
  var link = $compile(s, function(a, b) {return a + b});
  var node = link($scope);
  $element.append(node);
});

angular.bootstrap(document, ['Demo']);
```

我们定义了一个 `more` 指令，它的 `compile` 函数的第三个参数，就是我们手工 `$compile` 时传入的。

如果不是手工 `$compile`，而是 `ng` 初始化时找出的指令，则 `transclude` 是一个 `link` 函数（指令定义需要设置 `transclude` 选项）：

```
<div more>123</div>
```

```
app.directive('more', function($rootScope, $document) {
  var func = function(element, attrs, link) {
    var node = link($rootScope);
```

```

node.removeAttr('more'); //不去掉就变死循环了
$('body', $document).append(node);
}

return {compile: func,
        transclude: 'element', // element是节点没, 其它值是节点的内容没
        restrict: 'A'};
});

```

18.7. 把节点内容作为变量处理的类型

回顾最开始的那个代码显示的例子，那个例子只能处理一次节点内容。如果节点的内容是一个变量的话，需要用另外的思路来考虑。这里我们假设的例子是，定义一个指令 `showLength`，它的作用是在一段文本的开头显示出这段节点文本的长度，节点文本是一个变量。指令使用的形式是：

```

<div ng-controller="TestCtrl">
  <div show-length>{{ text }}</div>
  <button ng-click="text='xx'">改变</button>
</div>

```

从上面的 HTML 代码中，大概清楚 ng 解析它的过程（只看 `show-length` 那一行）：

- 解析 `div` 时发现了一个 `show-length` 的指令。
- 如果 `show-length` 指令设置了 `transclude` 属性，则 `div` 的节点内容被重新编译，得到的 `link` 函数作为指令 `compile` 函数的参数传入。
- 如果 `show-length` 指令没有设置 `transclude` 属性，则继续处理它的子节点（`TextNode`）。
- 不管是上面的哪种情况，都会继续处理到 `{{ text }}` 这段文本。
- 发现 `{{ text }}` 是一个 `Interpolate`，于是自动在此节点中添加了一个指令，这个指令的 `link` 函数就是为 `scope` 添加了一个 `$watch`，实现的功能是当 `scope` 作 `$digest` 的时候，就更新节点文本。

与处理 `{{ text }}` 时添加的指令相同，我们实现 `showLength` 的思路，也就是：

- 修改原来的 DOM 结构
- 为 `scope` 添加 `$watch`，当 `$digest` 时修改指定节点的文本，其值为指定节点文本的长度。

代码如下：

```

app.directive('showLength', function($rootScope, $document){
  var func = function(element, attrs, link){

    return function(scope, ielement, iattrs, controller){
      var node = link(scope);
      ielement.append(node);
      var lnode = $('<span></span>');
      ielement.prepend(lnode);

      scope.$watch(function(scope){
        lnode.text(node.text().length);
      });
    };
  };

  return {compile: func,

```



```
    transclude: true, // element是节点没, 其它值是节点的内容没
    restrict: 'A' });
  });
```

上面代码中, 因为设置了 `transclude` 属性, 我们在 `showLength` 的 `link` 函数 (就是 `return` 的那个函数) 中, 使用 `func` 的第三个函数来重塑了原来的文本节点, 并放在我们需要的位置上。然后, 我们添加自己的节点来显示长度值。最后给当前的 `scope` 添加 `$watch`, 以更新这个长度值。

18.8. 指令定义时的参数

指令定义时的参数如下:

- `name`
- `priority`
- `terminal`
- `scope`
- `controller`
- `require`
- `restrict`
- `template`
- `templateUrl`
- `replace`
- `transclude`
- `compile`
- `link`

现在我们开始一个一个地吃掉它们....., 但是并不是按顺序讲的。

priority

这个值设置指令的权重, 默认是 `0`。当一个节点中有多个指令存在时, 就按着权重从大到小的顺序依次执行它们的 `compile` 函数。相同权重顺序不定。

terminal

是否以当前指令的权重为结束界限。如果这值设置为 `true`, 则节点中权重小于当前指令的其它指令不会被执行。相同权重的会执行。

restrict

指令可以以哪些方式被使用, 可以同时定义多种方式。

- E 元素方式 `<my-directive></my-directive>`
- A 属性方式 `<div my-directive="exp"> </div>`
- C 类方式 `<div class="my-directive: exp;"></div>`
- M 注释方式 `<!-- directive: my-directive exp -->`

transclude

前面已经讲过基本的用法了。可以是 `'element'` 或 `true` 两种值。

compile

基本的定义函数。 `function compile(tElement, tAttrs, transclude) { ... }`

link

前面介绍过了。大多数时候我们不需要单独定义它。只有 `compile` 未定义时 `link` 才会被尝试。

```
function link(scope, iElement, iAttrs, controller) { ... }
```

scope

scope 的形式。 `false` 节点的 **scope** , `true` 继承创建一个新的 **scope** , `{}` 不继承创建一个新的隔离 **scope** 。 `{@attr: '引用节点属性', =attr: '把节点属性值引用成scope属性值', &attr: '把节点属性值包装成函数'}`

controller

为指令定义一个 **controller** , `function controller($scope, $element, $attrs, $transclude) { ... }`

name

指令的 **controller** 的名字, 方便其它指令引用。

require

要引用的其它指令 **controller** 的名字, `?name` 忽略不存在的错误, `^name` 在父级查找。

template

模板内容。

templateUrl

从指定地址获取模板内容。

replace

是否使用模板内容替换掉整个节点, `true` 替换整个节点, `false` 替换节点内容。

```
<a b></a>
```

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

app.directive('a', function() {
  var func = function(element, attrs, link) {
    console.log('a');
  }

  return {compile: func,
    priority: 1,
    restrict: 'EA'};
});

app.directive('b', function() {
  var func = function(element, attrs, link) {
    console.log('b');
  }

  return {compile: func,
    priority: 2,
    //terminal: true,
    restrict: 'A'};
});
```

上面几个参数值都是比较简单且容易理想的。

再看 **scope** 这个参数:

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

```
<div a b></div>
</div>
```

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('a', function() {
4      var func = function(element, attrs, link) {
5          return function(scope) {
6              console.log(scope);
7          }
8      }
9
10     return {compile: func,
11             scope: true,
12             restrict: 'A'};
13 });
14
15 app.directive('b', function() {
16     var func = function(element, attrs, link) {
17         return function(scope) {
18             console.log(scope);
19         }
20     }
21
22     return {compile: func,
23           restrict: 'A'};
24 });
25
26 app.controller('TestCtrl', function($scope) {
27     $scope.a = '123';
28     console.log($scope);
29 });
```

对于 **scope** :

- 默认为 **false** , **link** 函数接受的 **scope** 为节点所在的 **scope** 。
- 为 **true** 时, 则 **link** 函数中第一个参数 (还有 **controller** 参数中的 **\$scope**) , **scope** 是节点所在的 **scope** 的 **child scope** , 并且如果节点中有多个指令, 则只要其中一个指令是 **true** 的设置, 其它所有指令都会受影响。

这个参数还有其它取值。当其为 **{}** 时, 则 **link** 接受一个完全隔离 (**isolate**) 的 **scope** , 于 **true** 的区别就是不会继承其它 **scope** 的属性。但是这时, 这个 **scope** 的属性却可以有很灵活的定义方式:

@attr 引用节点的属性。

```
<div ng-controller="TestCtrl">
  <div a abc="here" xx="{{ a }}" c="ccc"></div>
</div>
```

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

app.directive('a', function() {
    var func = function(element, attrs, link) {
        return function(scope) {
            console.log(scope);
        }
    }
})
```

```

    return {compile: func,
            scope: {a: '@abc', b: '@xx', c: '@'},
            restrict: 'A'};
  });

  app.controller('TestCtrl', function($scope) {
    $scope.a = '123';
  });

```

- **@abc** 引用 **div** 节点的 **abc** 属性。
- **@xx** 引用 **div** 节点的 **xx** 属性，而 **xx** 属性又是一个变量绑定，于是 **scope** 中 **b** 属性值就和 **TestCtrl** 的 **a** 变量绑定在一起了。
- **@** 没有写 **attr name**，则默认取自己的值，这里是取 **div** 的 **c** 属性。

=attr 相似，只是它把节点的属性值当成节点 **scope** 的属性名来使用，作用相当于上面例子中的 **@xx**：

```

<div ng-controller="TestCtrl">
  <div a abc="here"></div>
</div>

```

```

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

app.directive('a', function() {
  var func = function(element, attrs, link) {
    return function(scope) {
      console.log(scope);
    }
  }

  return {compile: func,
          scope: {a: '=abc'},
          restrict: 'A'};
});

app.controller('TestCtrl', function($scope) {
  $scope.here = '123';
});

```

&attr 是包装一个函数出来，这个函数以节点所在的 **scope** 为上下文。来看一个很爽的例子：

```

<div ng-controller="TestCtrl">
  <div a abc="here = here + 1" ng-click="show(here)">这里</div>
  <div>{{ here }}</div>
</div>

```

```

1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('a', function() {
4    var func = function(element, attrs, link) {
5      return function llink(scope) {
6        console.log(scope);
7        scope.a();
8        scope.b();
9
10       scope.show = function(here) {
11         console.log('Inner, ' + here);

```

```

12         scope.a({here: 5});
13     }
14 }
15 }
16
17     return {compile: func,
18             scope: {a: '&abc', b: '&ngClick'},
19             restrict: 'A'};
20 });
21
22 app.controller('TestCtrl', function($scope) {
23     $scope.here = 123;
24     console.log($scope);
25
26     $scope.show = function(here) {
27         console.log(here);
28     }
29 });

```

`scope.a` 是 `&abc`，即：

```
scope.a = function() {here = here + 1}
```

只是其中的 `here` 是 `TestCtrl` 的。

`scope.b` 是 `&ngClick`，即：

```
scope.b = function() {show(here)}
```

这里的 `show()` 和 `here` 都是 `TestCtrl` 的，于是上面的代码最开始会在终端输出一个 `124`。

当点击“这里”时，这时执行的 `show(here)` 就是 `link` 中定义的那个函数了，与 `TestCtrl` 无关。但是，其间的 `scope.a({here:5})`，因为 `a` 执行时是 `TestCtrl` 的上下文，于是向 `a` 传递的一个对象，里面的所有属性 `TestCtrl` 就全收下了，接着执行 `here=here+1`，于是我们会在屏幕上看到 `6`。

这里是一个上下文交错的环境，通过 `&` 这种机制，让指令的 `scope` 与节点的 `scope` 发生了互动。真是鬼斧神工的设计。而实现它，只用了几行代码：

```

case '&': {
    parentGet = $parse(attrs[attrName]);
    scope[scopeName] = function(locals) {
        return parentGet(parentScope, locals);
    }
    break;
}

```

再看 `controller` 这个参数。这个参数的作用是提供一个 `controller` 的构造函数，它会在 `compile` 函数之后，`link` 函数之前被执行。

```
<a>haha</a>
```

```

1 var app = angular.module('Demo', [], angular.noop);
2
3 app.directive('a', function() {
4     var func = function() {

```

```

5     console.log('compile');
6     return function() {
7         console.log('link');
8     }
9 }
10
11 var controller = function($scope, $element, $attrs, $transclude) {
12     console.log('controller');
13     console.log($scope);
14
15     var node = $transclude(function(clone_element, scope) {
16         console.log(clone_element);
17         console.log('--');
18         console.log(scope);
19     });
20     console.log(node);
21 }
22
23 return { compile: func,
24         controller: controller,
25         transclude: true,
26         restrict: 'E' }
27 });

```

`controller` 的最后一个参数， `$transclude`，是一个只接受 `cloneAttachFn` 作为参数的一个函数。

按官方的说法，这个机制的设计目的是为了让各个指令之间可以互相通信。参考普通节点的处理方式，这里也是处理指令 `scope` 的合适位置。

```
<a b>kk</a>
```

```

1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('a', function() {
4      var func = function() {
5      }
6
7      var controller = function($scope, $element, $attrs, $transclude) {
8          console.log('a');
9          this.a = 'xx';
10     }
11
12     return { compile: func,
13             name: 'not_a',
14             controller: controller,
15             restrict: 'E' }
16 });
17
18 app.directive('b', function() {
19     var func = function() {
20         return function($scope, $element, $attrs, $controller) {
21             console.log($controller);
22         }
23     }
24
25     var controller = function($scope, $element, $attrs, $transclude) {
26         console.log('b');
27     }

```

```

28
29     return {compile: func,
30             controller: controller,
31             require: 'not_a',
32             restrict: 'EA'}
33   });

```

name 参数在这里可以用以为 **controller** 重起一个名字，以方便在 **require** 参数中引用。

require 参数可以带两种前缀（可以同时使用）：

- **?**，如果指定的 **controller** 不存在，则忽略错误。即：

```
require: '?not_b'
```

如果名为 **not_b** 的 **controller** 不存在时，不会直接抛出错误，**link** 函数中对应的 **\$controller** 为 **undefined**。

- **^**，同时在父级节点中寻找指定的 **controller**，把上面的例子小改一下：

```
<a><b>kk</b></a>
```

把 **a** 的 **require** 改成（否则就找不到 **not_a** 这个 **controller**）：

```
require: '?^not_a'
```

还剩下几个模板参数：

template 模板内容，这个内容会根据 **replace** 参数的设置替换节点或只替换节点内容。

templateUrl 模板内容，获取方式是异步请求。

replace 设置如何处理模板内容。为 **true** 时为替换掉指令节点，否则只替换到节点内容。

```

<div ng-controller="TestCtrl">
  <h1 a>原始内容</h1>
</div>

```

```

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

app.directive('a', function() {
  var func = function() {
  }

  return {compile: func,
          template: '<p>标题 {{ name }} <button ng-click="name=\'hahaha\'">修改</button></p>',
          //replace: true,
          //controller: function($scope) {$scope.name = 'xxx'},
          //scope: {},
          scope: true,
          controller: function($scope) {console.log($scope)},
          restrict: 'A'}
});

app.controller('TestCtrl', function($scope) {
  $scope.name = '123';
  console.log($scope);
});

```

```
});
```

template 中可以包括变量引用的表达式，其 **scope** 遵循 **scope** 参数的作用（可能受继承关系影响）。

templateUrl 是异步请求模板内容，并且是获取到内容之后才开始执行指令的 **compile** 函数。

最后说一个 **compile** 这个参数。它除了可以返回一个函数用为 **link** 函数之外，还可以返回一个对象，这个对象能包括两个成员，一个 **pre**，一个 **post**。实际上，**link** 函数是由两部分组成，所谓的 **preLink** 和 **postLink**。区别在于执行顺序，特别是在指令层级嵌套的结构之下，**postLink** 是在所有的子级指令 **link** 完成之后才最后执行的。**compile** 如果只返回一个函数，则这个函数被作为 **postLink** 使用：

```
<a><b></b></a>
```

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('a', function() {
4    var func = function() {
5      console.log('a compile');
6      return {
7        pre: function() {console.log('a link pre')},
8        post: function() {console.log('a link post')},
9      }
10   }
11
12   return {compile: func,
13           restrict: 'E'}
14 });
15
16 app.directive('b', function() {
17   var func = function() {
18     console.log('b compile');
19     return {
20       pre: function() {console.log('b link pre')},
21       post: function() {console.log('b link post')},
22     }
23   }
24
25   return {compile: func,
26           restrict: 'E'}
27 });
```

18.9. Attributes的细节

节点属性被包装之后会传给 **compile** 和 **link** 函数。从这个操作中，我们可以得到节点的引用，可以操作节点属性，也可以为节点属性注册侦听事件。

```
<test a="1" b c="xxx"></test>
```

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

app.directive('test', function() {
  var func = function($element, $attrs) {
    console.log($attrs);
  }
});
```



```
return {compile: func,
        restrict: 'E'}
```

整个 **Attributes** 对象是比较简单的，它的成员包括了：

\$\$element 属性所在的节点。

\$attr 所有的属性值（类型是对象）。

\$normalize 一个名字标准化的工具函数，可以把 `ng-click` 变成 `ngClick`。

\$observe 为属性注册侦听器的函数。

\$set 设置对象属性，及节点属性的工具。

除了上面这些成员，对象的成员还包括所有属性的名字。

先看 **\$observe** 的使用，基本上相当于 **\$scope** 中的 **\$watch**：

```
<div ng-controller="TestCtrl">
  <test a="{{ a }}" b c="xxx"></test>
  <button ng-click="a=a+1">修改</button>
</div>
```

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

app.directive('test', function() {
  var func = function($element, $attrs) {
    console.log($attrs);

    $attrs.$observe('a', function(new_v) {
      console.log(new_v);
    });
  };

  return {compile: func,
          restrict: 'E'}
});

app.controller('TestCtrl', function($scope) {
  $scope.a = 123;
});
```

\$set 方法的定义是：`function(key, value, writeAttr, attrName) { ... }`。

- **key** 对象的成员名。
- **value** 需要设置的值。
- **writeAttr** 是否同时修改 DOM 节点的属性（注意区别“节点”与“对象”），默认为 `true`。
- **attrName** 实际的属性名，与“标准化”之后的属性名有区别。

```
<div ng-controller="TestCtrl">
  <test a="1" ys-a="123" ng-click="show(1)">这里</test>
</div>
```

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

```

app.directive('test', function() {
  var func = function($element, $attrs) {
    $attrs.$set('b', 'ooo');
    $attrs.$set('a-b', '11');
    $attrs.$set('c-d', '11', true, 'c_d');
    console.log($attrs);
  }

  return {compile: func,
    restrict: 'E'}
});

app.controller('TestCtrl', function($scope) {
  $scope.show = function(v) {console.log(v);}
});

```

从例子中可以看到，原始的节点属性值对，放到对象中之后，名字一定是“标准化”之后的。但是手动 `$set` 的新属性，不会自动做标准化处理。

18.10. 预定义的 NgModelController

在前面讲 `controller` 参数的时候，提到过可以为指令定义一个 `controller`。官方的实现中，有很多已定义的指令，这些指令当中，有两个已定义的 `controller`，它们是 `NgModelController` 和 `FormController`，对应 `ng-model` 和 `form` 这两个指令（可以参照前面的“表单控件”一章）。

在使用中，除了可以通过 `$scope` 来取得它们的引用之外，也可以在自定义指令中通过 `require` 参数直接引用，这样就可以在 `link` 函数中使用 `controller` 去实现一些功能。

先看 `NgModelController`。这东西的作用有两个，一是控制 `ViewValue` 与 `ModelValue` 之间的转换关系（你可以实现看到的是一个值，但是存到变量里变成了另外一个值），二是与 `FormController` 配合做数据校验的相关逻辑。

先看两个应该是最有用的属性：

\$formatters 是一个由函数组成的列表，串行执行，作用是把变量值变成显示的值。

\$parsers 与上面的方向相反，把显示的值变成变量值。

假设我们在变量中要保存一个列表的类型，但是显示的东西只能是字符串，所以这两者之间需要一个转换：

```

<div ng-controller="TestCtrl">
  <input type="text" ng-model="a" test />
  <button ng-click="show(a)">查看</button>
</div>

```

```

1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('test', function() {
4    var link = function($scope, $element, $attrs, $ctrl) {
5
6      $ctrl.$formatters.push(function(value) {
7        return value.join(',');
8      });

```

```
9
10     $ctrl.$parsers.push(function(value) {
11         return value.split(',');
12     });
13 }
14
15 return {compile: function() {return link},
16         require: 'ngModel',
17         restrict: 'A'}
18 });
19
20 app.controller('TestCtrl', function($scope) {
21     $scope.a = [];
22     //$scope.a = [1,2,3];
23     $scope.show = function(v) {
24         console.log(v);
25     }
26 });
```

上面在定义 `test` 这个指令，`require` 参数指定了 `ngModel`。同时因为 DOM 结构，`ng-model` 是存在的。于是，`link` 函数中就可以获取到一个 `NgModelController` 的实例，即代码中的 `$ctrl`。

我们添加了需要的过滤函数：

- 从变量(`ModelValue`)到显示值(`ViewValue`)的过程，`$formatters` 属性，把一个列表变成一个字符串。
- 从显示值到变量的过程，`$parsers` 属性，把一个字符串变成一个列表。

对于显示值和变量，还有其它的 API，这里就不细说了。

另一部分，是关于数据校验的，放到下一章同 `FormController` 一起讨论。

18.11. 预定义的 `FormController`

前面的“表单控制”那章，实际上讲的就是 `FormController`，只是那里是从 `scope` 中获取到的引用。现在从指令定义的角度，来更清楚地了解 `FormController` 及 `NgModelController` 是如何配合工作的。

先说一下，`form` 和 `ngForm` 是官方定义的两个指令，但是它们其实是同一个东西。前者只允许以标签形式使用，而后者允许 `EAC` 的形式。DOM 结构中，`form` 标签不能嵌套，但是 `ng` 的指令没有这个限制。不管是 `form` 还是 `ngForm`，它们的 `controller` 都被命名成了 `form`。所以 `require` 这个参数不要写错了。

`FormController` 的几个成员是很好理解的：

-
- `$pristine` 表单是否被动过
 - `$dirty` 表单是否没被动过
 - `$valid` 表单是否检验通过
 - `$invalid` 表单是否检验未通过
 - `$error` 表单中的错误
 - `$setDirty()` 直接设置 `$dirty` 及 `$pristine`

```
<div ng-controller="TestCtrl">
  <div ng-form test>
    <input ng-model="a" type="email" />
    <button ng-click="do()">查看</button>
  </div>
</div>
```

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

app.directive('test', function() {
  var link = function($scope, $element, $attrs, $ctrl) {
    $scope.do = function() {
      //$ctrl.$setDirty();
      console.log($ctrl.$pristine); //form是否没被动过
      console.log($ctrl.$dirty); //form是否被动过
      console.log($ctrl.$valid); //form是否被检验通过
      console.log($ctrl.$invalid); //form是否有错误
      console.log($ctrl.$error); //form中有错误的字段
    }
  }

  return {compile: function() {return link},
    require: 'form',
    restrict: 'A'}
});

app.controller('TestCtrl', function($scope) {
});
```

\$error 这个属性，是一个对象， **key** 是错误名， **value** 部分是一个列表，其成员是对应的 **NgModelController** 的实例。

FormController 可以自由增减它包含的那些，类似于 **NgModelController** 的实例。在 DOM 结构上，有 **ng-model** 的 **input** 节点的 **NgModelController** 会被自动添加。

\$addControl() 添加一个 **controller**

\$removeControl() 删除一个 **controller**

这两个手动使用机会应该不会很多。被添加的实例也可以手动实现所有的 **NgModelController** 的方法

```
<div ng-controller="TestCtrl">
  <bb />
  <div ng-form test>
    <input ng-model="a" type="email" />
    <button ng-click="add()">添加</button>
  </div>
</div>
```

```
1 var app = angular.module('Demo', [], angular.noop);
2
3 app.directive('test', function() {
4   var link = function($scope, $element, $attrs, $ctrl) {
5     $scope.add = function() {
```

```

6         $ctrl.$addControl($scope.bb);
7         console.log($ctrl);
8     }
9 }
10
11 return {compile: function() {return link},
12         require: 'form',
13         restrict: 'A' }
14 });
15
16 app.directive('bb', function() {
17     var controller = function($scope, $element, $attrs, $transclude) {
18         $scope.bb = this;
19         this.$name = 'bb';
20     }
21
22     return {compile: angular.noop,
23             restrict: 'E',
24             controller: controller}
25 });
26
27 app.controller('TestCtrl', function($scope) {
28 });

```

整合 **FormController** 和 **NgModelController** 就很容易扩展各种类型的字段:

```

<div ng-controller="TestCtrl">
  <form name="f">
    <input type="my" ng-model="a" />
    <button ng-click="show()">查看</button>
  </form>
</div>

```

```

1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('input', function() {
4      var link = function($scope, $element, $attrs, $ctrl) {
5          console.log($attrs.type);
6          var validator = function(v) {
7              if(v == '123') {
8                  $ctrl.$setValidity('my', true);
9                  return v;
10             } else {
11                 $ctrl.$setValidity('my', false);
12                 return undefined;
13             }
14         }
15
16         $ctrl.$formatters.push(validator);
17         $ctrl.$parsers.push(validator);
18     }
19
20     return {compile: function() {return link},
21             require: 'ngModel',
22             restrict: 'E' }
23 });
24
25 app.controller('TestCtrl', function($scope) {
26     $scope.show = function() {
27         console.log($scope.f);

```

```
28     }  
29   });
```

虽然官方原来定义了几种 `type`，但这不妨碍我们继续扩展新的类型。如果新的 `type` 参数值不在官方的定义列表里，那会按 `text` 类型先做处理，这其实什么影响都没有。剩下的，就是写我们自己的验证逻辑就行了。

上面的代码是参见官方的做法，使用格式化的过程，同时在里面做有效性检查。

18.12. 示例：文本框

这个例子与官网上那个例子相似。最终是要显示一个文本框，这个文本框由标题和内容两部分组成。而且标题和内容则是引用 `controller` 中的变量值。

HTML 部分的代码：

```
<div ng-controller="TestCtrl">  
  <ys-block title="title" text="text"></ys-block>  
  <p>标题: <input ng-model="title" /></p>  
  <p>内容: <input ng-model="text" /></p>  
  <ys-block title="title" text="text"></ys-block>  
</div>
```

从这个期望实现效果的 HTML 代码中，我们可以考虑设计指令的实现方式：

- 这个指令的使用方式是“标签”，即 `restrict` 这个参数应该设置为 `E`。
- 节点的属性值是对 `controller` 变量的引用，那么我们应该在指令的 `scope` 中使用 `=` 的方式来指定成员值。
- 最终的效果显示需要进行 DOM 结构的重构，那直接使用 `template` 就好了。
- 自定义的标签在最终效果中是多余的，所有 `replace` 应该设置为 `true`。

JS 部分的代码：

```
var app = angular.module('Demo', [], angular.noop);  
  
app.directive('ysBlock', function() {  
  return {compile: angular.noop,  
    template: '<div style="width: 200px; border: 1px solid black;"><h1 style="background-color: gray; color: white; font-size: 22px;">{{ title }}</h1><div>{{ text }}</div></div>',  
    replace: true,  
    scope: {title: '=title', text: '=text'},  
    restrict: 'E'};  
});  
  
app.controller('TestCtrl', function($scope) {  
  $scope.title = '标题在这里';  
  $scope.text = '内容在这里';  
});  
  
angular.bootstrap(document, ['Demo']);
```

可以看到，这种简单的组件式指令，只需要作 DOM 结构的变换即可实现，连 `compile` 函数都不需要写。

18.13. 示例：模板控制语句 for

这个示例尝试实现一个重复语句，功能同官方的 `ngRepeat`，但是使用方式类似于我们通常编程语言中的 `for` 语句：

```
<div ng-controller="TestCtrl" ng-init="obj_list=[1,2,3,4]; name='name'">
  <ul>
    <for o in obj_list>
      <li>{{ o }}, {{ name }}</li>
    </for>
  </ul>
  <button ng-click="obj_list=[1,2]; name='o?'">修改</button>
</div>
```

同样，我们从上面的使用方式去考虑这个指令的实现：

- 这是一个完全的控制指令，所以单个节点应该只有它一个指令起作用就好了，于是权重要比较高，并且“到此为止”—— `priority` 设置为 `1000`，`terminal` 设置为 `true`。
- 使用时的语法问题。事实上浏览器会把 `for` 节点补充成一个正确的 HTML 结构，即里面的属性都会变成类似 `o=""` 这样。我们通过节点的 `outerHTML` 属性取到字符串并解析取得需要的信息。
- 我们把 `for` 节点之间的内容作为一个模板，并且通过循环多次渲染该模板之后把结果填充到合适的位置。
- 在处理上面的那个模板时，需要不断地创建新 `scope` 的，并且 `o` 这个成员需要单独赋值。

注意：这里只是简单实现功能。官方的那个 `ngRepeat` 比较复杂，是做了专门的算法优化的。当然，这里的实现也可以是简单把 DOM 结构变成使用 `ngRepeat` 的形式：)

JS 部分代码：

```
1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('for', function($compile) {
4    var compile = function($element, $attrs, $link) {
5      var match = $element[0].outerHTML.match('<for (.*?)=.*? in=.*? (.*?)=.*?>');
6      if(!match || match.length !== 3) {throw Error('syntax: <for o in obj_list>')}
7      var iter = match[1];
8      var list = match[2];
9      var tpl = $compile($.trim($element.html()));
10     $element.empty();
11
12     var link = function($scope, $ielement, $iattrs, $controller) {
13
14       var new_node = [];
15
16       $scope.$watch(list, function(list) {
17         angular.forEach(new_node, function(n) {n.remove()});
18         var scp, inode;
19         for(var i = 0, ii = list.length; i < ii; i++) {
20           scp = $scope.$new();
21           scp[iter] = list[i];
22           inode = tpl(scp, angular.noop);
23           $ielement.before(inode);
24           new_node.push(inode);
25         }
26
27       });
28     }
29   }
```

```

29
30     return link;
31   }
32   return {compile: compile,
33           priority: 1000,
34           terminal: true,
35           restrict: 'E'};
36 });
37
38 app.controller('TestCtrl', angular.noop);
39 angular.bootstrap(document, ['Demo']);

```

18.14. 示例：模板控制语句 if/else

这个示例是尝试实现：

```

<div ng-controller="TestCtrl">
  <if true="a == 1">
    <p>判断为真, {{ name }}</p>
  <else>
    <p>判断为假, {{ name }}</p>
  </else>
</if>

  <div>
    <p>a: <input ng-model="a" /></p>
    <p>name: <input ng-model="name" /></p>
  </div>
</div>

```

考虑实现的思路：

- **else** 与 **if** 是两个指令，它们是父子关系。通过 **scope** 可以联系起来。至于 **scope** 是在 **link** 中处理还是 **controller** 中处理并不重要。
- **true** 属性的条件判断通过 **\$parse** 服务很容易实现。
- 如果最终效果要去掉 **if** 节点，我们可以使用注释节点来“占位”。

JS 代码：

```

1  var app = angular.module('Demo', [], angular.noop);
2
3  app.directive('if', function($parse, $compile){
4    var compile = function($element, $attrs){
5      var cond = $parse($attrs.true);
6
7      var link = function($scope, $ielement, $iattrs, $controller){
8        $scope.if_node = $compile($.trim($ielement.html()))($scope, angular.noop);
9        $ielement.empty();
10       var mark = $('<!-- IF/ELSE -->');
11       $element.before(mark);
12       $element.remove();
13
14       $scope.$watch(function(scope) {
15         if(cond(scope)) {
16           mark.after($scope.if_node);
17           $scope.else_node.detach();
18         } else {
19           if($scope.else_node !== undefined) {

```



```
20         mark.after($scope.else_node);
21         $scope.if_node.detach();
22     }
23 }
24 });
25 }
26 return link;
27 }
28
29 return {compile: compile,
30         scope: true,
31         restrict: 'E'}
32 });
33
34 app.directive('else', function($compile){
35     var compile = function($element, $attrs){
36
37         var link = function($scope, $element, $attrs, $controller){
38             $scope.else_node = $compile($.trim($element.html()))($scope, angular.noop);
39             $element.remove();
40         }
41         return link;
42     }
43
44     return {compile: compile,
45           restrict: 'E'}
46 });
47
48 app.controller('TestCtrl', function($scope){
49     $scope.a = 1;
50 });
51
52 angular.bootstrap(document, ['Demo']);
```

代码中注意一点,就是 `if_node` 在得到之时,就已经是做了变量绑定的了。错误的思路是,在 `$watch` 中再去不断地得到新的 `if_node`。

评论

149 条评论 进出自由, 我的分享

 dongguangming ▾

 Recommended 19  分享

按从新到旧排序 ▾



加入讨论...



hackrole • 2个月前

第一个例子就运行不起来.代码如下(完全复制下来的, 回复里转义有点问题. 只改了jquery和angularjs的地址):

```
<html>
<head>
<meta charset="utf-8"/>

<title>试验</title>

<script src="http://code.jquery.com/jquery-..."></script>
```

```
<script src="http://ajax.googleapis.com/aja..."></script>
</head>
<body>
<div ng-controller="BoxCtrl">
<div style="width: 100px; height: 100px; background-color: red;" ng-click="click()"></div>

{{ w }} x {{ h }}
```

查看更多

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 ➔ hackrole • 2个月前

<https://docs.angularjs.org/gui...>

1.3 之后, Controller 不允许全局定义了。

^ | v • 回复 • 分享 ›



TouKN • 4个月前

nice, 受益良多

^ | v • 回复 • 分享 ›



Mike • 4个月前

Great Article! Thanks!

^ | v • 回复 • 分享 ›



david • 5个月前

非常厉害 谢谢分享~还需要仔细研读理解下

^ | v • 回复 • 分享 ›



Elvis Macak • 6个月前

非常不方便打印, web版的代码背景色是黑色, pdf版的代码背景色和字体都是浅色,无法看清
建议更改一下代码的颜色风格

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 ➔ Elvis Macak • 6个月前

谢谢反馈。后面已经应用了新的颜色方案, 这里的还没有改。
我发了一份到你留的邮箱。

^ | v • 回复 • 分享 ›



magicdawn • 9个月前

问下用的angular版本是哪一版

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 ➔ magicdawn • 9个月前

目录后紧跟着就写了嘛——本文的内容是在 1.0.x 版本之下完成的。

^ | v • 回复 • 分享 ›



Reason • 9个月前

太强大了,非常清晰,但是还想问个问题:ng-repeat="o in list" 怎么能拿到序列号 0,1,2,3,4

^ | v • 回复 • 分享 ›



Reason → Reason • 9个月前

问题已找到答案，使用\$index便可以获取序列号

1 ^ | v • 回复 • 分享 >



Evan • 9个月前

hi,你好，拜读了你的大作，学到了不少的东西，非常感谢。。。

我想问下关于AngularJS的路由，请问有办法去掉其中'#'，总觉得这个'#'看起来怪怪的。

^ | v • 回复 • 分享 >



ZephyrZou 管理员 → Evan • 9个月前

开启 html5 model，浏览器的 api 支持的话，就可以没有 # 的。

（我个人觉得这样对于开发阶段不太直观的，因为那只是表面上的修改了浏览器地址栏的地址，和直接打开这个地址是两种行为）。

目前的相关文档在 <https://docs.angularjs.org/gui...>

1 ^ | v • 回复 • 分享 >



Evan → ZephyrZou • 9个月前

非常赞同你括号里面那句话。

我是这样想的：如果是用来开发手机端APP的，有没有'#'无所谓，但是来开发PC端应用的话，地址栏里面有个'#'，总觉得让人不舒服（我是强迫症患者，呵呵。。。），我看到AngularJS中文社区已经完美解决了这个问题。

我已经通过开启html5model加上Nginx做资源重定向解决了部分问题，例如我访问http://localhost:9000/phones这个页面的时候就已经能够脱离'#'，并且可以任意刷新或者在地址栏里面直接输入这个地址。

但是在三级页面就不行了，比如http://localhost:9000/phones/22，我在首页点击这个链接进去是可以的，如果我刷新这个页面，或者我在地址栏里面直接输入这个地址敲回车就不行了（这也正是验证了你括号里面的那句话），因此悲剧了。。。

^ | v • 回复 • 分享 >



ZephyrZou 管理员 → Evan • 9个月前

呵呵，我只是说不直观了。其实在后端的支持下，这本来就不是问题，只是后端要多做工作，麻烦一些。

以你说的例子，后端实现对 /phones/22 的正确渲染就可以了啊。（而在前端的 ng 中，可能对 #/phones/22 的处理，完全没有请求出去）

（你用 nginx 的 rewrite 要起作用，也需要后端真正实现页面渲染才行嘛）

更多的你可以看看 PJAX 的相关内容。在那里，才是真正的都是 /phones/22 的访问，但是要求不同情况响应的内容不同。

简单来说，带了一个头，比如 X-PJAX，则只响应 HTML 片断。否则，就响应完整的 HTML 页面。

^ | v • 回复 • 分享 >



Evan → ZephyrZou • 9个月前

不知道我理解是不是对的，既然用了AngularJS，那么就应该符合它本质的特性——单页面应用。后端仅仅只提供数据，所有的渲染都由前端来完成，我去掉'#'仅仅是为了地址栏更传统一点。

我感觉似乎是可以的，因为AngularJS中文社区是可以的，在浏览的时候我

我感觉似乎是可以的，因为AngularJS中又往区是可以的，在浏览的时候我注意到无论地址栏怎么变（都没有“#”），这个页面只是刷新了内容区的DOM，像head, footer等公共区域没有变，用户体验很好，这点很让人心动。

另外，你说的PJAX，我看了下，确实可以实现我的需求，谢谢！！

^ | v • 回复 • 分享 >



ZephyrZou 管理员 ➔ Evan • 9个月前

> 不知道我理解是不是对的，既然用了AngularJS，那么就应该符合它本质的特性——单页面应用。后端仅仅只提供数据，所有的渲染都由前端来完成...

我个人在这方面的观点是，无所谓“本质”与否。它只是工具，工具怎么用，何时用，则在于人。

对于前端来说，后端也是工具。后端是提供数据，但是数据的形式呢？json, xml, html, js 它们都是数据。所以也不存在说“所有的渲染都由前端来完成了”。

对于后端来说，前端也是工具。我后端的模块可以用于生成html，也自然可以用于生成js, json。

所以，在具体的项目，具体的场景之下，不要给自己无谓的约束，而看不到最合适的方式。为了单页面而单页面，就得不偿失了。

> 像head, footer等公共区域没有变

看到的与真正发生的可不总是一致。即使你刷新了整个页面，因为浏览器的处理，不变的部分你不一定感觉得到“刷新”行为。

回到angularjs.cn的例子，它的实现，我猜大概是前端全包的形式。后端针对用到的path，全部返回大致相同的html。前端初始化之后，根据location再触发逻辑（在html5mode下，ng应该实现了这个机制吧）。

对于页面上链接的处理，也完全可以在事件上干掉默认行为，而自己实现ajax的加载。

^ | v • 回复 • 分享 >



姓名 • 1年前

var link = \$compile(\$('#a')); 这句为什么总是说“\$”未定义？？

^ | v • 回复 • 分享 >



ZephyrZou 管理员 ➔ 姓名 • 1年前

\$是jQuery的API，你有引入jQuery么？

^ | v • 回复 • 分享 >



1 ➔ ZephyrZou • 1年前

谢谢，是的，解决了。

^ | v • 回复 • 分享 >



dsky.pw • 1年前

有几段代码中添加angular.bootstrap(document.documentElement);会报错

^ | v • 回复 • 分享 >



ZephyrZou 管理员 → dsky.pw • 1年前

能说具体一些么。或者给一个会报错，你又不明白为什么的小例子？

1 ^ | v • 回复 • 分享 ›



dsky.pw → ZephyrZou • 1年前

7.5.6的例子添加angular.bootstrap(document.documentElement);会报 Uncaught Error: [ng:btstrpd] <http://errors.angularjs.org/1....> 的错，而且select中没有下拉值，不添加就是正常的

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 → dsky.pw • 1年前

我试没有问题啊，完整的代码：<https://gist.github.com/yszou/...>

^ | v • 回复 • 分享 ›



dsky.pw → ZephyrZou • 1年前

懂了，知道问题在哪里，谢谢啦～

^ | v • 回复 • 分享 ›



xilou • 1年前

angular 1.2.x已经发布了很长时间了，楼主有没有想过，把教程的内容换成基于angular 1.2.x，毕竟1.0.x和1.2.x并不相互兼容

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 → xilou • 1年前

我计划是等 1.3 发布，然后在 1.3 上着手修订。

^ | v • 回复 • 分享 ›



mark • 1年前

虽然英文不是很好，但还是看完了官方网站的内容，而中文angular实在少咨询。这份学习笔记真是好！想法思路都和原文符合。有些部分直接说到点，而官网却希望看的人自己悟，自己想。

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 → mark • 1年前

还有重要的“自己试” :)，现在官网的文档应该好多了吧。

这里的一些内容在当时确实也是直接从源码理解出来的，特别是 **directive** 这块，本身内容也多，每一个点都要配合示例代码才好知道到底是什么东西，怎么用，然后还要想想在什么情况下用，用以解决什么问题。

这个过程对我自己，也是受益匪浅了。

^ | v • 回复 • 分享 ›



Bill • 1年前

博主能不能写一下单元测试方面的经验？

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 → Bill • 1年前

目前是真没什么经验的了。

可能在接下的几个月才会真正在前端做一些测试方面的事。

^ | v • 回复 • 分享 ›



yanchangyou • 1年前

✎ 7 回复



好义

· 回复 · 分享 ›



tyrion · 1年前

初学angular, 博主这篇文章写得真好!

· 回复 · 分享 ›



ZephyrZou 管理员 → tyrion · 1年前

谢谢。有什么建议或不爽都请留言在评论中, 以便新版本中参考改进。

· 回复 · 分享 ›



weweTom · 1年前

滚动太痛苦了, 能否加一个 jump top 功能.....

· 回复 · 分享 ›



ZephyrZou 管理员 → weweTom · 1年前

谢谢你, 你说的这个, 我后面考虑页面结构相关功能时会考虑的。

(HOME 键就是干这事的吧)

[http://checkcheckzz.gitbooks.i...](http://checkcheckzz.gitbooks.io/) 这里下面评论的朋友整理的此文, 其形式可能是你需要的。

· 回复 · 分享 ›



vampireslove · 1年前

博主您好, 刚学angularjs, 觉得写到特别好, 非常感谢, 有个问题想请教下, 在页面上, 如果只是输出后台的数据, 用ng-bind 绑定就可以了, 并且想格式化, 可以自己定义过滤, 来达到目录, 但是如果是<input/>控件, 需要双向绑定, 就得用ng-model绑定, 现在想对<input>展示出来的结果进行数据格式化, 好像这个ng-model 后面是不能用过滤器的, 但是不知道有什么好的办法?

· 回复 · 分享 ›



ZephyrZou 管理员 → vampireslove · 1年前

“预定义的 NgModelController” 这节是你需要的内容。

· 回复 · 分享 ›



zach · 1年前

初学angular, 虽然英文读的还行, 但是中文的还是看的爽, 我做一个gitbook的版本, 容易读, 而且有pdf等版本

[http://checkcheckzz.gitbooks.i...](http://checkcheckzz.gitbooks.io/)

repo在

<https://github.com/checkcheckz...>

我已把你加入了这个repo的合作者, 然后你可以直接用github更新, 若想这样的话

1 · 回复 · 分享 ›



wdq → zach · 1年前

pdf版本在哪里下载, 没有找到呀。

· 回复 · 分享 ›



zach → wdq · 1年前

<https://github.com/checkcheckz...>

电子书下载部分

1 · 回复 · 分享 ›



ZephyrZou 管理员 → zach • 1年前

这个工具看起来不错，稍后我会在文中加入链接指向。

^ | v • 回复 • 分享 ›



KendraZhi • 1年前

楼主啊，拷贝的代码都带行号的 有没有可以直接下载的demo包呀

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 → KendraZhi • 1年前

这个没有。那些行号都是对齐的吧，用编辑器敲几下就删了的。或者，你从 [github](https://github.com/zephyrzou/z...) 上下源文件从那里取代码？ <https://github.com/zephyrzou/z...>

^ | v • 回复 • 分享 ›



KendraZhi → ZephyrZou • 1年前

\$element 这个api里差不到呀，不知道什么意思 是说他跟jQuery类似的内容吗

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 → KendraZhi • 1年前

注意版本哦，上面写的内容都是1.0.8的 :)

它是跟 jQuery 类似的一个东西，是 ng-controller 所在的那个节点。

在官文的文档好像是没有，嗯，不过在源码中，有些 directive 中定义 ctrl 时有用到它，比如处理 select 标签时。

^ | v • 回复 • 分享 ›



KendraZhi → ZephyrZou • 1年前

哈哈，好的 我也是用编辑器敲几下删了。感谢大神提供demo的下载地址

^ | v • 回复 • 分享 ›



ビン テイ • 1年前

大神，我在日本搞开发的，最近日本上司要我用angularjs做个东西，但是以前都没用过，我想像你请教很多东西，请把你的QQ联系方式发给我好吗，我的邮箱是114575280@qq.com

^ | v • 回复 • 分享 ›



ZephyrZou 管理员 → ビン テイ • 1年前

QQ不怎么用的，你可以在这个站找到我的邮箱。方便的时候GTalk也能说上话吧。

^ | v • 回复 • 分享 ›



Joe • 1年前

更新个1.2.x的版本就更完美了：D

^ | v • 回复 • 分享 ›

加载更多评论

在 [进出自由, 我的分享](#) 上还有.....

这是什么？

在[Tornado中使用Django的ORM的注意事项](#)

2 条评论 • 3年前

异步调用中的上下文控制--[Tornado 中的 stack context](#)

3 条评论 • 2年前

[关于](#)

4 条评论 • 3年前

[使用邮件客户端整合日常信息](#)

4 条评论 • 2年前

 订阅

 在您的网站上使用 **Disqus**

 隐私

©2010-2014 zouyesheng.com All rights reserved. Powered by [GitHub](#) , [txt2tags](#) , [MathJax](#)