← 上一篇 下一篇 →

AngularJS开发指南01：AngularJS简介

发表于 2012年9月20日 rainer_H

什么是 AngularJS?

AngularJS 是一个为动态WEB应用设计的结构框架。它能让你使用HTML作为模板语言，通过扩展HTML的语法，让你能更清楚、简洁地构建你的应用组件。它的创新点在于，利用 **数据绑定** 和 **依赖注入**，它使你不用再写大量的代码了。这些全都是通过浏览器端的Javascript实现，这也使得它能够完美地和任何服务器端技术结合。

AngularJS是为了克服HTML在构建应用上的不足而设计的。HTML是一门很好的为静态文本展示设计的声明式语言，但要构建WEB应用的话它就显得乏力了。所以我做了一些工作（你也可以觉得是小花招）来让浏览器做我想要的事。

通常，我们是通过以下技术来解决静态网页技术在构建动态应用上的不足：

- **类库** - 类库是一些函数的集合，它能帮助你写WEB应用。起主导作用的是你的代码，由你来决定何时使用类库。类库有：`jQuery`等
- **框架** - 框架是一种特殊的、已经实现了的WEB应用，你只需要对它填充具体的业务逻辑。这里框架是起主导作用的，由它来根据具体的应用逻辑来调用你的代码。框架有：`knockout`、`sproutcore`等。

AngularJS使用了不同的方法，它尝试去补足HTML本身在构建应用方面的缺陷。AngularJS通过使用我们称为**标识符**(directives)的结构，让浏览器能够识别新的语法。例如：

- 使用双大括号`{{}}`语法进行数据绑定；
- 使用DOM控制结构来实现迭代或者隐藏DOM片段；
- 支持表单和表单的验证；
- 能将逻辑代码关联到相关的DOM元素上；
- 能将HTML分组成可重用的组件。

端对端的解决方案

AngularJS试图成为成为WEB应用中的一种端对端的解决方案。这意味着它不只是你的WEB应用中的一个小部分，而是一个完整的端对端的解决方案。这会让AngularJS在构建一个**CRUD**（增加Create、查询Retrieve、更新Update、删除Delete）的应用时显得很“固执”（原文为opinionated,意指没有太多的其他方式）。但是，尽管它很“固执”，它仍然能确保它的“固执”只是在你构建应用的起点，并且你仍能灵活变动。AngularJS的一些出众之处如下：

- 构建一个CRUD应用可能用到的全部内容包括：数据绑定、基本模板标识符、表单验证、路由、深度链接、组件重用、依赖注入。
- 测试方面包括：单元测试、端对端测试、模拟和自动化测试框架。
- 具有目录布局和测试脚本的种子应用作为起点。

AngularJS的可爱之处

AngularJS通过为开发者呈现一个更高层次的抽象来简化应用的开发。如同其他的抽象技术一样，这也会损失一部分灵活性。换句话说，并不是所有的应用都适合用AngularJS来做。AngularJS主要考虑的是构建CRUD应用。幸运的是，至少90%的WEB应用都是CRUD应用。但是要了解什么适合用AngularJS构建，就得了解什么不适合用AngularJS构建。

如游戏，图形界面编辑器，这种DOM操作很频繁也很复杂的应用，和CRUD应用就有很大的不同，它们不适合用AngularJS来构建。像这种情况用一些更轻量、简单的技术如`jQuery`可能会更好。

一个简单的AngularJS实例

下面是一个包含了一个表单的典型CRUD应用。表单值先经过验证，然后用来计算总值，这个总值会被格式化成本地的样式。下面有一些开发者常见的概念，你需要先了解一下：

- 将**数据模型**(data-model)关联到**视图**(UI)上；
- 写、读、验证用户的输入；
- 根据模型计算新的值；
- 将输出格式本地化。

index.html:

```
<!doctype html>
<html ng-app>
    <head>
        <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
        <script src="script.js"></script>
    </head>
    <body>
        <div ng-controller="InvoiceCntl">
            <b>Invoice:</b>
            <br>
            <br>
            <table>
                <tr><td>Quantity</td><td>Cost</td></tr>
                <tr>
                    <td><input type="integer" min="0" ng-model="qty" required ></td>
                    <td><input type="number" ng-model="cost" required ></td>
                </tr>
            </table>
            <hr>
            <b>Total:</b> {{qty * cost | currency}}
        </div>
    </body>
</html>
```

script.js:

```
function InvoiceCntl($scope) {
    $scope.qty = 1;
    $scope.cost = 19.95;
}
```

end-to-end test:

```
it('should show of angular binding', function() {
    expect(binding('qty * cost')).toEqual('$19.95');
    input('qty').enter('2');
    input('cost').enter('5.00');
    expect(binding('qty * cost')).toEqual('$10.00');
});
```

运行效果

Invoice:

Quantity | Cost

1 | 19.95

Total: $19.95

试一下上面这个例子，然后我们一起来看下这个例子的工作原理。

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

我们用`<script>`标签来加载AngularJS脚本：

```
<script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
```

通过设置`<input>`标签里的`ng-model`属性, AngularJS会自动对数据进行双向绑定。我们还同时进行了一些简单的数据验证：

```
Quantity: <input type="integer" min="0" ng-model="qty" required >
Cost: <input type="number" ng-model="cost" required >
```

这个输入框的widget看起来很普通，但如果认识到以下几点那它就不普通了：

- 当页面加载完后，AngularJS会依照widget里的声明的模型名字（`qty`、`cost`）生成同名变量。你可以把这些变量认为是MVC设计模式中的M(Model)；
- 注意上面widget里的`input`有着特殊的能力。如果你们没有输入数据或者输入的数据无效，这个`input`输入框会自动变红。输入框的这种新特性，能让开发者更容易实现CRUD应用里常见的字段验证功能。

终于，我们可以来看一下神秘的双大括号`{{}}`了:

```
Total: {{qty * cost | currency}}
```

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

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

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

AngularJS的“禅道(理念)”

Angular信奉的是，当组建视图(UI)同时又要写软件逻辑时，声明式的代码会比命令式的代码好得多，尽管命令式的代码非常适合用来表述业务逻辑。

- 将DOM操作和应用逻辑解耦是一种非常好的思路，它能大大改善代码的可调性；
- 将 **测试** 和 **开发** 同等看待是一种非常非常好的思路，测试的难度在很大程度上取决于代码的结构；
- 将客户端和服务器端解耦是一种特别好的做法，它能使两边并行开发，并且使两边代码都能实现重用；
- 如果框架能够在整个开发流程里都引导着开发者：从设计UI，到编写业务逻辑，再到测试，那对开发者将是极大的帮助；
- “化繁为简，化简为零”总是好的。

AngularJS能将你从以下的噩梦中解脱出来：

- **使用回调：** 回调的使用会打乱你的代码的可读性，让你的代码变得支离破碎，很难看清本来的业务逻辑。移除一些常见的代码，例如回调，是件好事。大幅度地减少你因为JavaScript这门语言的设计而不得不写的代码，能让你把自己应用的逻辑看得更清楚。
- **手动编写操作DOM元素的代码：**操作DOM是AJAX应用很基础的一部分，但它也总是很“笨重”并且容易出错。用声明的方式描述的UI界面可随着应用状态的改变而变化，能让你从编写低级的DOM操作代码中解脱出来。绝大部分用AngularJS写的应用里，开发者都不用再自己去写操作DOM的代码，不过如果你想的话还是可以去写。
- **对UI界面读写数据：** AJAX应用的很大一部是CRUD操作。一个经典的流程是把服务端的数据组建成内部对象，再把对象编成HTML表单，用户修改表单后再验证表单，如果有错再显示错误，然后将数据重新组建成内部对象，再返回给服务器。这个流程里有太多太多要重复写的代码，使得代码看起来总是在描述应用的全部执行流程，而不是具体的业务逻辑和业务细节。
- **开始前得写大量的基础性的代码：** 通常你需要写很多的基础性的代码才能实现一个“Hello World”的应用。用AngularJS的话，它会提供一些服务让你很容易地正式开始写你的应用，而这些服务都是以一种Guice-like dependency-injection式的依赖注入自动加入到你的应用中去的,这让你能很快的进入你应用的具体开发。特别的是，你还能全盘掌握自动化测试的初始化过程。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南02：引导程序

发表于 2012年9月22日 rainer_H

概览

这一节解释了AngularJS初始化的过程，以及需要的时候你该如何修改AngularJS的初始化。

AngularJS的 `<script>` 标签

这个示例展示了我们推荐的整合AngularJS的方法，它被称之为“自动初始化”。

```
<!doctype html>
<html xmlns:ng="http://angularjs.org" ng-app>
    <body>
        ...
    <script src="angular.js"><script>
    </body>
</html>
```

* 将上面代码中的 `script` 标签放在页面的底部。将 `script` 标签放在底部缩短应用加载的时间，因为这样HTML的加载不会被 `angular.js` 脚本的加载阻塞。你可以从http://code.angularjs.org获得最新的版本。请不要在你的代码里面引用这个URL，因为它会暴露你的站点的安全隐患。如果只是实验性质的开发，那链接到我们的站点没有什么问题。
* `angular-[version].js` 是具有可读性的版本, 适合开发和调试。
* `angular-[version].min.js` 是压缩和混淆后的版本, 适合部署到成型产品中。
* 请将 `ng-app` 指令 放到你的应用的标签根节点中, 如果你想要AngularJS自动执行整个 `<html>` 程序就把它放在 `<html>` 标签中。

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

- 如果你想使用旧版的指令语法：`ng:`，那还需要将 `xml-namespace` 写在 `<html>` 中才能使AngularJS在IE下正常工作。(这样做是因为一些历史原因, 我们不推荐继续使用 `ng:` 的语法。)

```
<html xmlns:ng="http://angularjs.org">
```

自动初始化

AngularJS会在 `DOMContentLoaded` 事件触发时执行，并通过 `ng-app` 指令 寻找你的应用根作用域。如果 `ng-app` 指令找到了，那么AngularJS将会：

- 载入和 指令 内容相关的模块。
- 创建一个应用的“注入器(injector)”。
- 已拥有 `ng-app` 指令 的标签为根节点来编译其中的DOM。这使得你可以只指定DOM中的一部分作为你的AngularJS应用。

```
<!doctype html>
<html ng-app="optionalModuleName">
    <body>
        I can add: {{ 1+2 }}.
        <script src="angular.js"></script>
    </body>
</html>
```

手动初始化

如果你需要主动控制一下初始化的过程，你可以使用手动执行引导程序的方法。比如当你使用“脚本加载器(script loader)”，或者需要在AngularJS编译页面之前做一些操作，你就会用到它了。

下面的例子演示了手动初始化AngularJS的方法。它的效果等同于使用 `ng-app` 指令 。

```
<!doctype html>
<html xmlns:ng="http://angularjs.org">
```

```
    <body>
        Hello {{'World'}}!
        <script src="http://code.angularjs.org/angular.js"></script>
        <script>
            angular.element(document).ready(function() {
            angular.bootstrap(document);
            });
        </script>
    </body>
</html>
```

下面是一些你的代码必须遵守的顺序：

1. 等页面和所有的脚本加载完之后，找到HTML模板的根节点——通常就是文档的根节点。
2. 调用 api/angular.bootstrap将模板编译成可执行的、数据双向绑定的应用程序。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南03：HTML编译器

发表于 2012年9月22日 rainer_H

概览

AngularJS的HTML编译器能让浏览器识别新的HTML语法。它能让你将行为关联到HTML元素或者属性上，甚至能让你创造具有自定义行为的新元素。AngularJS称这种行为扩展为“**指令**”

HTML在编写静态页面时，有很多声明式的结构来控制格式。比如你要把某个内容居中，你不必告诉浏览器“去找到窗口的中点位置，然后跟内容的中间结合”。你只需要添加一个 align="center" 的属性给需要内容居中的元素就行了。这就是声明式语言的强大之处。

但是声明式语言也有力所不能及的地方，原因之一在于你不能用它来让浏览器识别新的语法。比如说，你不要内容居中，而是居左到1/3,这时它就做不到了。所以我们需要一个办法让浏览器能学会新的HTML语法。

AngularJS生来自带一些对创建APP非常有用的**指令**。我们也希望你能自己创造一些对你自己的应用有用的**指令**。这些扩展的**指令**就是你创建APP的 “**特定领域语言（Domain Specific Language）**”。

编译的过程都会在浏览器端发生；服务器端不会参与到其中的任何步骤，也不会做预编译。

编译器(complier)

编译器是AngularJS提供的一项服务，它通过遍历DOM来查找和它相关的属性。整个编译的过程分为两个阶段。

- **编译**： 遍历DOM并且收集所有的相关指令，生成一个链接函数。
- **链接**： 给指令绑定一个作用域，生成一个动态的视图。作用域模型的任何改变都会反映到视图上，并且视图上的任何用户操作也都会反映到作用域模型。这使得作用域模型成为你的业务逻辑里唯一要关心的东西。

有一些指令，比如 `ng-repeat` 会为数据集合里的每一项DOM元素都克隆一次。将整个编译过程分为**编译**和**链接**两个阶段的作法改善了整体的性能，因为克隆出来的模板总共只需要被编译一次，然后链接到各自的模型实例上就行了。

指令

指令指示的是**“当关联的HTML结构进入编译阶段时应该执行的操作”**。指令可以写在元素的名称里，属性里，css类名里，注释里。下面有几个功能相同的使用 `ng-bind` 指令的例子。

```
<span ng-bind="exp"></span>
<span class="ng-bind: exp;"></span>
<ng-bind></ng-bind>
<!-- directive: ng-bind exp -->
```

指令本质上只是一个当编译器编译到相关DOM时需要执行的函数。你可以在指令API文档中找到更详尽的关于指令的资料。

下面是一条能让元素变得可拖拽的指令。注意 `<span>` 元素里的那个 `draggable` 属性。

index.html:

```
<!doctype html>
<html ng-app="drag">
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <span draggable>Drag ME</span>
  </body>
</html>
```

script.js:

```
angular.module('drag', []).
directive('draggable', function($document) {
```

```
    var startX=0, startY=0, x = 0, y = 0;
    return function(scope, element, attr) {
      element.css({
       position: 'relative',
       border: '1px solid red',
       backgroundColor: 'lightgrey',
       cursor: 'pointer'
      });
      element.bind('mousedown', function(event) {
        startX = event.screenX - x;
        startY = event.screenY - y;
        $document.bind('mousemove', mousemove);
        $document.bind('mouseup', mouseup);
      });

      function mousemove(event) {
        y = event.screenY - startY;
        x = event.screenX - startX;
        element.css({
          top: y + 'px',
          left:  x + 'px'
        });
      }

      function mouseup() {
        $document.unbind('mousemove', mousemove);
        $document.unbind('mouseup', mouseup);
      }
    }
});
```

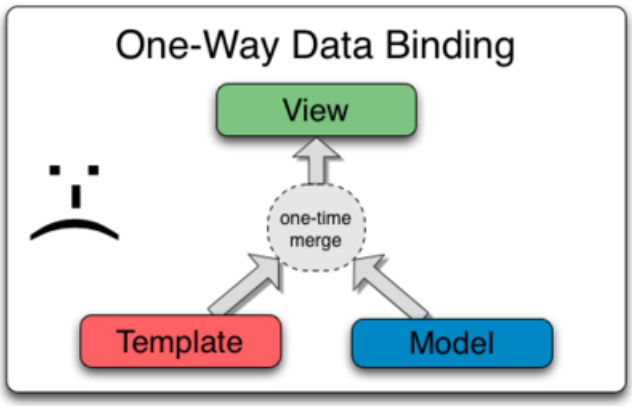
运行效果

Drag ME

通过加入`draggable`属性可以让任何HTML元素都实现这个新的行为。我们这种改进的优美之处在于我们给了浏览器新能力。我们用了一种只要开发者熟悉HTML规则，就会举得很自然的方式扩展了浏览器理解新行为新语法的能力。

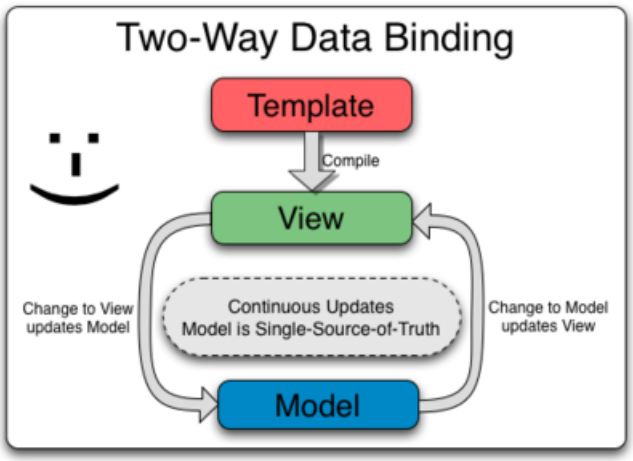
理解视图

网上有很多的模板系统。他们大多数都是**“将静态的字符模板和数据绑定，生成新字符，然后通过innerHTML插入到页面元素中”**。



这意味着数据上的任何改变，都会导致数据要重新和模板结合生成新字符，再插入到DOM里。这其中会出现的问题有：需要读取用户输入并和模型的数据结合，需要覆写用户的输入，需要手动管理整个更新过程，缺少丰富的表现形式。

AngularJS则不同，AngularJS编译器使用的是带指令的DOM，而不是字符串模板。它返回的是一个链接函数，这个函数和作用域模型结合就会生成一个动态视图。这个视图和模型的绑定过程是“透明的”。开发者不需要做任何关于更新视图的工作。并且应用没有用到innerHTML，所以我们也不用覆写用户的输入。更特别的是，Angular的指令不仅仅能使用字符串形式的绑定，还可以使用一些指示行为的结构体。



AngularJS的编译会生成一个**“稳定的DOM”**。这意味绑定了数据模型的DOM元素的实例不会在绑定的生命周期中发生改变。这也意味着代码中可以获取到DOM元素的实例引用并注册事件，不用担心这用引用会在模板和数据的结合时丢失。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南04：核心概览

发表于 2012年9月23日 rainer_H

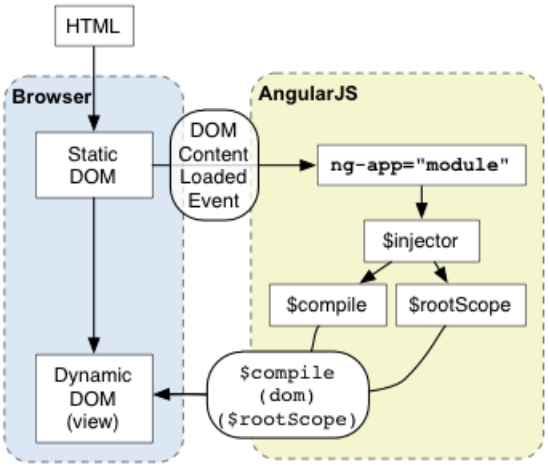
概览

这篇文档描述了AngularJS的主要组成部分，以及它们如何协同工作。它们是：

- 启动(startup) - 展示“hello world!”
- 执行期(runtime) - AngularJS 执行期概览
- 作用域(scope) - 视图和控制器的集合区
- 控制器(controller) - 应用的行为
- 模型(model) - 应用的数据
- 视图(view) - 用户能看到的
- 指令(directives) - 扩展HTML语法
- 过滤器(filters) - 数据本地化
- 注入器(injector) - 聚合你的应用
- 模块(module) - 配置注入器
- $ - AngularJS的命名空间(namespace)

启动

下面解释了我们是如何把这一切运转起来的(用一张图和一个例子来解释)：



1. 浏览器载入HTML，然后把它解析成DOM。
2. 浏览器载入angular.js脚本。
3. AngularJS等到 `DOMContentLoaded` 事件触发。
4. AngularJS寻找 `ng-app` 指令，这个指令指示了应用的边界。
5. 使用 `ng-app` 中指定的模块来配置注入器($injector)。
6. 注入器($injector)是用来创建“编译服务($compile service)”和“根作用域($rootScope)”的。
7. 编译服务($compile service)是用来编译DOM并把它链接到根作用域($rootScope)的。
8. `ng-init` 指令将“World”赋给作用域里的name这个变量。
9. 通过 `{{name}}` 的替换，整个表达式变成了“Hello World”。

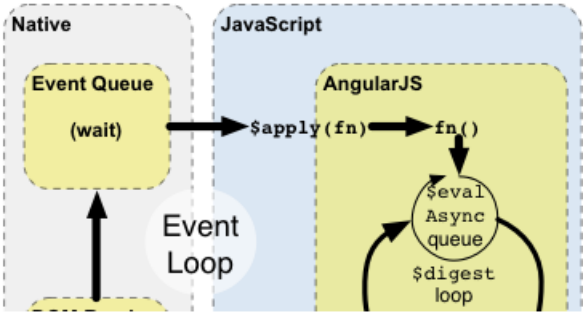
index.html:

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
  </head>
  <body>
    <p ng-init=" name='World' ">Hello {{name}}!</p>
  </body>
</html>
```

运行效果

执行期

下面的图和例子解释了AngularJS如何和浏览器的事件回路(event loop)交互。



1. 浏览器的事件循环等待事件的触发。所谓事件包括用户的交互操作、定时事件、或者网络事件(服务器的响应)。
2. 事件触发后，回调会被执行。此时会进入Javascript上下文。通常回调可以用来修改DOM结构。
3. 一旦回调执行完毕，浏览器就会离开Javascript上下文，并且根据DOM的修改重新渲染视图。

AngularJS通过使用自己的事件处理循环，改变了传统的Javascript工作流。这使得Javascript的执行被分成原始部分和拥有AngularJS执行上下文的部分。只有在AngularJS执行上下文中运行的操作，才能享受到AngularJS提供的数据绑定，异常处理，资源管理等功能和服务。你可以使用`$apply()`来从普通Javascript上下文进入AngularJS执行上下文。记住，大部分情况下（如在控制器，服务中），$apply都已经被用来处理当前事件的相应指令执行过了。只有当你使用自定义的事件回调或者是使用第三方类库的回调时，才需要自己执行`$apply`。

1. 通过调用`scope.$apply(stimulusFn)`来进入AngularJS的执行上下文，这里的stimulusFn是你希望在AngularJS执行上下文中执行的函数。
2. AngularJS会执行`stimulusFn()`，这个函数一般会改变应用的状态。
3. AngularJS进入$digest循环。这个循环是由两个小循环组成的，这两个小循环用来处理处理$evalAsync队列和$watch列表。这个$digest循环直到模型“稳定”前会一直迭代。这个稳定具体指的是$evalAsync对表为空，并且$watch列表中检测不到任何改变了。
4. 这个$evalAsync队列是用来管理那些“视图渲染前需要在当前栈框架外执行的操作的”。这通常使用`setTimeout(0)`来完成的。用`setTimeout(0)`会有速度慢的问题。并且，因为浏览器是根据事件队列按顺序渲染视图的，还会造成视图的抖动。
5. $watch列表是一个表达式的集合，这些表达式可能是自上次迭代后发生了改变的。如果有检测到了有改变，那么$watch函数就会被调用，它通常会把新的值更新到DOM中。
6. 一旦AngularJS的$digest循环结束，整个执行就会离开AngularJS和Javascript的上下文。这些都是在浏览器为数据改变而进行重渲染之后进行的。

下面解释了"hello world"的例子是怎么样实现“将用户输入绑定到视图上”的效果。

1. 在编译阶段：
   A. input上的`ng-model`指令给`<input>`输入框绑定了keydown事件；
   B. `{{name}}`这个变量替换表单式建立了一个 $watch 来接受 name 变量改变的通知。
2. 在执行期阶段：
   A. 按下任何一个键(以X键为例)，都会触发一个 input 输入框的keydown事件；
   B. input 上的指令捕捉到 input 里值得改变，然后调用 $apply("name = 'X';")来更新处于AngularJS执行上下文中的模型；
   C. AngularJS将`name='X'`应用到模型上；
   D. $digest 循环开始；
   E. $watch 列表检测到了name值的变化，然后通知`{{name}}`变量替换的表达式，这个表达式负责将DOM进行更新；
   F. AngularJS退出执行上下文，然后退出Javascript上下文中的keydown事件；
   G. 浏览器以更新的文本重渲染视图。

index.html:

```
<!doctype html>
<html ng-app>
  <head>
```

```
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
  </head>
  <body>
    <input ng-model="name">
    <p>Hello {{name}}!</p>
  </body>
</html>
```

运行效果

作用域(Scope)

作用域是用来检测模型的改变和为表达式提供执行上下文的。它是分层组织起来的，并且层级关系是紧跟着DOM的结构的。(请参考单个指令的文档，有一些指令会导致新的作用域的创建。)

下面这个例子演示了`{{name}}`表达式在不同的作用域下解析成不同的值。这个例子下面的图片显示作用域的边界。

`index.html:`

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="GreetCtrl">
      Hello {{name}}!
    </div>
    <div ng-controller="ListCtrl">
      <ol>
        <li ng-repeat="name in names">{{name}}</li>
      </ol>
    </div>
  </body>
</html>
```

`style.css:`

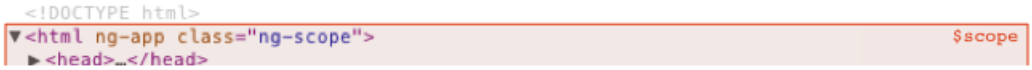
```
.show-scope .doc-example-live.ng-scope,
.show-scope .doc-example-live .ng-scope  {
  border: 1px solid red;
  margin: 3px;
}
```

`script.js:`

```
function GreetCtrl($scope) {
  $scope.name = 'World';
}

function ListCtrl($scope) {
  $scope.names = ['Igor', 'Misko', 'Vojta'];
}
```

运行效果



控制器

视图背后的控制代码就是控制器。它的主要工作内容是构造模型，并把模型和回调方法一起发送到视图。 视图可以看做是作用域在模板(HTML)上的“投影(projection)”。而作用域是一个中间地带，它把模型整理好传递给视图，把浏览器事件传递给控制器。控制器和模型的分离非常重要，因为：

- 控制器是由Javascript写的。Javascript是命令式的，命令式的语言适合用来编写应用的行为。控制器不应该包含任何关于渲染代码（DOM引用或者片段）。
- 视图模板是用HTML写的。HTML是声明是的，声明式的语言适合用来编写UI。视图不应该包含任何行为。
- 因为控制器和视图没有直接的调用关系，所以可以使多个视图对应同一个控制器。这对“换肤(re-skinning)”、适配不同设备（比如移动设备和台式机）、测试，都非常重要。

index.html:

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="MyCtrl">
      Hello {{name}}!
      <button ng-click="action()">
        OK
      </button>
    </div>
  </body>
</html>
```

script.js:

```
function MyCtrl($scope) {
  $scope.action = function() {
    $scope.name = 'OK';
  }

  $scope.name = 'World';
}
```

运行效果

模型

模型就是用来和模板结合生成视图的数据。模型必须在作用域中时可以被引用，这样才能被渲染生成视图。和其他框架不一样的是，Angularjs对模型本身没有任何限制和要求。你不需要继承任何类也不需要实现指定的方法以供调用或者改变模型。 模型可以是原生的对象哈希形式的，也可以是完整对象类型的。简而言之，模型可以是原生的Javascript对象。

视图

所谓视图，就是指用户所看见的。 视图的生命周期由作为一个模板开始，它将和模型合并并最终渲染到浏览器的DOM中。与其他模板系统不同的是，AngularJS使用一种独特的形式来渲染视图。

- 其他模板 - 大部分模板系统工作原理，都是一开始获取一个带有特殊标记的HTML形式字符串。通常情况下模板的特殊标记破换了HTML的语法，以至于模板是不能用HTML编辑器编辑的。然后这个字符串会被送到模板引擎那里解析，并和数据合并。合并的结果是一个可以被浏览器解析的HTML字符串。这个字符串会被 `.innerHTML` 方法写到DOM中。使用innerHTML会造成浏览器的重新渲染。当模型改变时，这整个流程又要重复一遍。模板的生存周期就是DOM的更新周期。这里我想强调是，这些模板系统模板的基础是字符串。
- AngularJS - AngularJS和其它模板系统不同。它使用的是DOM而不是字符串。模板仍然是用HTML字符串写的，并且它仍然是HTML。浏览器将它解析成DOM， 然后这个DOM会作为输入传递给模板引擎，也就是我们的编译器。编译器查看其中的指令，找到的指令后，会开始监视指令内容中相应的模型。 这样做，就使得视图能“连续地”更新，不需要模板和数据的重新合并。你的模型也就成了你视图变化的唯一动因。

index.html:

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
  </head>
  <body>
    <div ng-init="list = ['Chrome', 'Safari', 'Firefox', 'IE'] ">
      <input ng-model="list" ng-list> <br>
      <input ng-model="list" ng-list> <br>
      <pre>list={{list}}</pre> <br>
      <ol>
        <li ng-repeat="item in list">
          {{item}}
        </li>
      </ol>
    </div>
  </body>
</html>
```

运行效果

指令

一个指令 就是一种**“由某个属性、元素名称、css类名出现而导致的行为，或者说是DOM的变化”**。指令能让你以一种声明式的方法来扩展HTML表示能力。下面演示了一个增加了数据绑定的“内容可编辑”HTML。

index.html:

```
<!doctype html>
<html ng-app="directive">
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div contentEditable="true" ng-model="content">Edit Me</div>
    <pre>model = {{content}}</pre>
  </body>
</html>
```

script.js:

```
<!doctype html>
<html ng-app="directive">
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div contentEditable="true" ng-model="content">Edit Me</div>
    <pre>model = {{content}}</pre>
  </body>
</html>
```

style.css:

```
div[contentEditable] {
  cursor: pointer;
```

```
  background-color: #D0D0D0;
  margin-bottom: 1em;
  padding: 1em;
}
```

运行效果

Filters过滤器

过滤器扮演着数据翻译的角色。一般他们主要用在数据需要格式化成本地格式的时候。它参照了UNIX过滤的规则，并且也实现了“|”（管道）语法。

index.html:

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
  </head>
  <body>
    <div ng-init="list = ['Chrome', 'Safari', 'Firefox', 'IE'] ">
      Number formatting: {{ 1234567890 | number }} <br>
      array filtering <input ng-model="predicate">
      {{ list | filter:predicate | json }}
    </div>
  </body>
</html>
```

运行效果

模块和注入器

每个AngularJS应用都有一个唯一的注入器。注入器提供一个通过名字查找对象实例的方法。它将所有对象缓存在内部，所以如果重复调用同一名称的对象，每次调用都会得到同一个实例。如果调用的对象不存在，那么注入器就会让**实例工厂(instance factory)**创建一个新的实例。

一个模块就是一种配置注入器实例工厂的方式，我们也称为**“提供者(provider)”**。

```
// Create a module
var myModule = angular.module('myModule', [])

// Configure the injector
myModule.factory('serviceA', function() {
  return {
    // instead of {}, put your object creation here
  };
});

// create an injector and configure it from 'myModule'
var $injector = angular.injector('myModule');

// retrieve an object from the injector by name
var serviceA = $injector.get('serviceA');

// always true because of instance cache
$injector.get('serviceA') === $injector.get('serviceA');
```

注入器真正强大之处在于它可以用来调用方法和实例化的类型。这个精妙的特性让方法和类型能够通过注入器请求到他们依赖的组件，而不需要自己加载依赖。

```
// You write functions such as this one.
function doSomething(serviceA, serviceB) {
  // do something here.
}

// Angular provides the injector for your application
var $injector = ...;

////////////////////////////////////////////
// the old-school way of getting dependencies.
var serviceA = $injector.get('serviceA');
var serviceB = $injector.get('serviceB');

// now call the function
doSomething(serviceA, serviceB);

////////////////////////////////////////////
// the cool way of getting dependencies.
// the $injector will supply the arguments to the function automatically
```

```
$injector.invoke(doSomething); // This is how the framework calls your functions
```

注意，你唯一需要写的就是上例中指出的函数，并把需要的依赖写在函数参数里。当AngularJS调用这个函数时，它会自动体充好需要的参数。

看看下面的动态时间的例子，注意它是如何在构造器中列举出依赖的。当`ng-controller`实例化构造器的时候，它自动提供了指明的依赖。没有必要去创建依赖、查找依赖、或者提供一个依赖的引用给注入器。

index.html:

```
<!doctype html>
<html ng-app="timeExampleModule">
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="ClockCtrl">
      Current time is: {{ time.now }}
    </div>
  </body>
</html>
```

script.js:

```
angular.module('timeExampleModule', []).
  // Declare new object call time,
  // which will be available for injection
  factory('time', function($timeout) {
    var time = {};

    (function tick() {
      time.now = new Date().toString();
      $timeout(tick, 1000);
    })();
    return time;
  });

// Notice that you can simply ask for time
// and it will be provided. No need to look for it.
function ClockCtrl($scope, time) {
  $scope.time = time;
}
```

运行效果

AngularJS 命名空间

为了防止意外的命名冲突， AngularJS为可能冲突的对象名加以前缀"$"。所以请不要在你自己的代码里用"$"做前缀，以免和AngularJS代码发生冲突。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

《AngularJS开发指南04：核心概览》上有 1 条评论

defims 在 2013年1月28日08:59 说道： 登录以回复

指令的script.js部分有错误，复制了index.html部分。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南05：指令

发表于 2012年10月4日 rainer_H

指令使我们用来扩展浏览器能力的技术之一。在DOM编译期间，和HTML关联着的指令会被检测到，并且被执行。这使得指令可以为DOM指定行为，或者改变它。

AngularJS有一套完整的、可扩展的、用来帮助web应用开发的指令集，它使得HTML可以转变成“**特定领域语言(DSL)**”。

从HTML中调用指令

指令遵循驼峰式命名，如`ngBind`。指令可以通过使用指定符号转化成链式风格的的名称来调用，特定符号包括`:`,`-`,`_`。你好可以选择给指令加上前缀，比如“x-”，“data-”来让它符合html的验证规则。这里有以下可以用的指令名称例子：`ng:bind`, `ng-bind`, `ng_bind`, `x-ng-bind`, `data-ng-bind`。

指令可以做为元素名，属性名，类名，或者注释。下面是一些等效调用myDir指令的例子：

```
<span my-dir="exp"></span>
<span class="my-dir: exp;"></span>
<my-dir></my-dir>
<!-- directive: my-dir exp -->
```

指令可以通过很多不同的方式调用，但最后结果都是等效的。下例中对此做了演示。

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl1">
      Hello <input ng-model='name'> <hr/>
      &ltspan ng:bind="name"&gt <span ng:bind="name"></span> <br/>
      &ltspan ng_bind="name"&gt <span ng_bind="name"></span> <br/>
      &ltspan ng-bind="name"&gt <span ng-bind="name"></span> <br/>
      &ltspan data-ng-bind="name"&gt <span data-ng-bind="name"></span> <br/>
      &ltspan x-ng-bind="name"&gt <span x-ng-bind="name"></span> <br/>
    </div>
  </body>
</html>
```

script.js：

```
function Ctrl1($scope) {
  $scope.name = 'angular';
}
```

End to end test：

```
it('should show off bindings', function() {
  expect(element('div[ng-controller="Ctrl1"] span[ng-bind]').text()).toBe('angular');
});
```

字符串替换式

在编译期间，编译器会用$interpolate服务去检查文本中是否嵌入了表达式。这个表达式会被当成一个监视器一样注册，并且在 作为$digest循环中的一部分，它会自动更新。下面是一个替换式的例子：

```
<img src="img/{{username}}.jpg">Hello {{username}}!</img>
```

编译过程和指令匹配

HTML的编译分为三个阶段：

1. 首先浏览器会用它的标准API将HTML解析成DOM。 你需要认清这一点，因为我们的模板必须是可被解析的HTML。这是AngularJS和那些“以字符串为基础而非以DOM元素为基础的”模板系统的区别之处。
2. DOM的编译是有`$compile`方法来执行的。 这个方法会遍历DOM并找到匹配的指令。一旦找到一个，它就会被加入一个指令列表中，这个列表是用来记录所有和当前DOM相关的指令的。 一旦所有的指令都被确定了，会按照优先级被排序，并且他们的`compile`方法会被调用。 指令的`$compile()`函数能修改DOM结构，并且要负责生成一个link函数（后面会提到）。$compile方法最后返回一个合并起来的链接函数，这是链接函数是每一个指令的compile函数返回的链接函数的集合。
3. 通过调用一步所说的链接函数来将模板与作用域链接起来。这会轮流调用每一个指令的链接函数，让每一个指令都能对DOM注册监听事件，和建立对作用域的的监听。这样最后就形成了作用域的DOM的动态绑定。任何一个作用域的改变都会在DOM上体现出来。
   var $compile = ...; // injected into your code
   var scope = ...;

```
var html = '<div ng-bind='exp'></div>';

// Step 1: parse HTML into DOM element
var template = angular.element(html);

// Step 2: compile the template
var linkFn = $compile(template);

// Step 3: link the compiled template with the scope.
linkFn(scope);
```

编译和链接分离的合理性分析

你可能会疑惑为什么编译过程和链接过程要分离。要明白其中的原因，你可以先看下面这个带有“重复指令”的例子：

```
Hello {{user}}, you have these actions:
<ul>
  <li ng-repeat="action in user.actions">
    {{action.description}}
  </li>
</ul>
```

简短的说，编译和链接的分离是模型和DOM结构能够动态关联的一种需要。

当上面的例子被编译后，编译器会遍历所有节点来寻找指令。例如{{user}}是一个替换式指令，`ngRepeat`是另一个指令。但是`ngRepeat`有一个难题。他需要为`user.actions`中的每一个`action`构造一个li。这以为着它先要保存一个“干净”的li元素来用作克隆，然后等新的action插入进来时，克隆一个li并插入到ul中。但是仅仅克隆li的话工作还没完。他还需要编译这个li才能把其中的像是{{action.descriptions}}的替换式替换成相应作用域下的值。我们可以用一个简单地方法来克隆和插入li元素然后编译它。但是要编译每一个li的话，使用克隆会速度很慢， 因为编译的工程需要我们遍历DOM树，并找到对应的指令并执行它们。如果我们在一个需要循环100次循环体内执行编译的话，性能问题就会马上凸现出来。

而我们的解决方案就是将编译工程分为两个阶段。编译阶段将指令识别出来并按优先级排序，编译阶段将作用域中的实例和li进行链接。

`ngRepeat`会阻止li子元素的编译。取而代之的是`ngRepeat`指令会单独对li进行编译。这个编译结束后会生成一个链接函数，这个函数包含了准备li元素上的所有指令，并等待被绑定到相应克隆出来的li元素上。在执行期，`ngRepeat`之指令会监视表达式，当有新的元素增加到对应的数组之后，它就会新克隆一个li元素，为它创建一个新作用域，并使用链接函数把它和对应作用域链接上。

总结

编译函数 - 编译函数在指令中是很少的， 因为大部分指令都只是为了处理相应的DOM元素实例，而不是改变模板DOM元素。考虑到性能问题，任何指令的实例见能被共享的操作都应该移到编译函数中。

链接函数 - 指令很少不带有链接函数，链接函数可以让指令对相应克隆元素注册事件，还可以将作用域中的内容复制到DOM中。

如何写指令(短版)

下面这个例子演示一个获取当前时间的指令。

index.html：

```
<!doctype html>
<html ng-app="time">
  <head>
    <script src="http://code.angularjs.org/angular-1.1.0.min.js"></script>
```

```
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl2">
      Date format: <input ng-model="format"> <hr/>
      Current time is: <span my-current-time="format"></span>
    </div>
  </body>
</html>
```

script.js :

```
function Ctrl2($scope) {
  $scope.format = 'M/d/yy h:mm:ss a';
}

angular.module('time', [])
  // Register the 'myCurrentTime' directive factory method.
  // We inject $timeout and dateFilter service since the factory method is DI.
  .directive('myCurrentTime', function($timeout, dateFilter) {
    // return the directive link function. (compile function not needed)
    return function(scope, element, attrs) {
      var format,  // date format
          timeoutId; // timeoutId, so that we can cancel the time updates

      // used to update the UI
      function updateTime() {
        element.text(dateFilter(new Date(), format));
      }

      // watch the expression, and update the UI on change.
      scope.$watch(attrs.myCurrentTime, function(value) {
        format = value;
        updateTime();
      });

      // schedule update in one second
      function updateLater() {
        // save the timeoutId for canceling
        timeoutId = $timeout(function() {
          updateTime(); // update DOM
          updateLater(); // schedule another update
        }, 1000);
      }

      // listen on DOM destroy (removal) event, and cancel the next UI update
      // to prevent updating time after the DOM element was removed.
      element.bind('$destroy', function() {
        $timeout.cancel(timeoutId);
      });

      updateLater(); // kick off the UI update process.
    }
  });
```

如何写指令(长版)

下面是写一个完整的指令的例子。

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  var directiveDefinitionObject = {
    priority: 0,
    template: '<div></div>',
    templateUrl: 'directive.html',
    replace: false,
    transclude: false,
    restrict: 'A',
    scope: false,
    compile: function compile(tElement, tAttrs, transclude) {
      return {
        pre: function preLink(scope, iElement, iAttrs, controller) { ... },
        post: function postLink(scope, iElement, iAttrs, controller) { ... }
      }
    },
    link: function postLink(scope, iElement, iAttrs) { ... }
  };
  return directiveDefinitionObject;
});
```

大部分情况下你不需要控制这么多细节。 这其中的具体内容下面会有完整的解释。 这一节里我们只关注大部分指令相同的地方。

要简化上面的代码，我们首先要依赖一下基本选项的默认值。使用默认值的话，上面的代码可以简化成：

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  var directiveDefinitionObject = {
    compile: function compile(tElement, tAttrs) {
      return function postLink(scope, iElement, iAttrs) { ... }
    }
  };
  return directiveDefinitionObject;
});
```

大部分指令只关心实例，并不需要将模板进行变形，所以我们还可以简化：

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  return function postLink(scope, iElement, iAttrs) { ... }
});
```

Factory method工厂函数

工厂函数是用来创建指令的。它只会被调用一次：就是当编译器第一次匹配到相应指令的时候。你可以在其中进行任何初始化的工作。调用它时使用的是 `$injector.invoke` ，所以它遵循所有注入器的规则。

Directive Definition Object 指令定义对象

指令定义对象给编译器提供了生成指令需要的细节。这个对象的属性有：

- **名称name** - 当前作用域的名称，在注册是可选的。
- **优先级priority** - 当一个DOM上有多个指令时，有会需要指定指令执行的顺序。 这个**优先级**就是用来在执行指令的compile函数前先排序的。高优先级的先执行。 相同优先级的指令顺序没有被指定谁先执行。
- **终端terminal** - 如果被设置为true，那么该指令就会在同一个DOM的指令集和中最后被执行。任何其他“terminal”的指令也仍然会执行，因为同级的指令顺序是没有被定义的。
- **作用域scope**- 如果被定义成：
  - 那么就会为当前指令创建一个新的作用域。如果有多个在同一个DOM上的指令要求创建新作用域，那么只有一个新的会被创建。 这一创建新作用域的规则不适用于模板的根节点，因为模板的根节点总是会得到一个新的作用域。
  - {} 对象哈希 - 那么一个新的“孤立的”作用域就会被创建。这个“孤立的”作用域区别于一般作用域的地方在于，它不会以原型继承的方式直接继承自父作用域。这对于创建可重用的组件是非常有用的，因为可重用的组件一般不应该读或写父作用域的数据。

    这个“孤立的”作用域使用一个对象哈希来表示，这个哈希定义了一系列本地作用域属性， 这些本地作用域属性是从父作用域中衍生出来的。这些属性主要用来分析模板的值。这个哈希的键值对是本地属性为键，它的来源为值。
    - @ 或 @attr - 将本地作用域成员成员和DOM属性绑定。绑定结果总是一个字符串，因为DOM的属性就是字符串。如果DOM属性的名字没有被指定，那么就和本地属性名一样。比如说 `<widget my-attr="hello {{name}}">` 和作用域对象: `{ localName:'@myAttr' }` 。当 `name` 值改变的时候， 作用域中的LocalName也会改变。这个 `name` 是从父作用域中读来的（而不是组件作用域）。
    - = 或 =expression(表达式) - 在本地作用域属性和父作用域属性间建立一个双向的绑定。如果没有指定父作用域属性名称，那就和本地名称一样。 比如 `<widget my-attr="parentModel">` 和作用域对象: `{ localModel:'=myAttr' }` , 本地属性 `localModel` 会反映父作用域中 `parentModel` 的值。localModel和parentModel的任一方改变都会影响对方。
    - & 或 &attr - 提供了一种能在父作用域下执行表达式的方法。如果没有指定父作用域属性名称，那就和本地名称一样。 比如 `<widget my-attr="count = count + value">` 和作用域对象： `{ localFn:'increment()' }` 。本地作用域成员 `localFn` 会指向一个 `increment` 表达式的函数包装。通常你可以通过这个表达式从本地作用域给父作用域传值， 操作方法是将本地变量名和值得对应关系传给这个表达式的包装函数。比如说，这个表达式是 `increment(amount)` ，那么你就可以用调用 `localFn({amount:22})` 的方式指定amount的值。
- **控制器controller** - 控制器的构造对象。这个控制器函数是在预编译阶段被执行的，并且它是共享的，其他指令可以通过它的名字得到（参考依赖属性[require attribute]）。这就使得指令间可以互相交流来扩大自己的能力。会传递给这个函数的参数有：
  - $scope - 当前元素关联的作用域。
  - $element - 当前元素
  - $attrs - 当前元素的属性对象。

    - $transclude - 模板链接功能前绑定到正确的模板作用域：`function(cloneLinkingFn)`。
- **请求require** - 请求将另一个控制器作为参数传入到当前链接函数。 这个请求需要传递被请求指令的控制器的名字。如果没有找到，就会触发一个**错误**。请求的名字可以加上下面两个前缀：
    - `?` - 不要触发错误，这只是一个可选的请求。
    - `^` - 没找到的话，在父元素的controller里面也查找有没有。
- **限制restrict** - EACM中的任意一个之母。它是用来限制指令的声明格式的。如果没有这一项。那就只允许使用属性形式的指令。
    - E - 元素名称：`<my-directive></my-directive>`
    - A - 属性：`<div my-directive="exp"> </div>`
    - C - 类名：`<div class="my-directive: exp;"></div>`
    - M - 注释：`<!-- directive: my-directive exp -->`
- `模板template` - 将当前的元素替换掉。 这个替换过程会自动将元素的属性和css类名添加到新元素上。更多细节请查考章节“创建widgets”。
- 模板templateUrl - 和template属性一样，只不过这里指示的是一个模板的URL。因为模板加载是异步的，所有编译和链接都会等到加载完成后再执行。
- **替换replace** - 如果被设置成true那么现在的元素会被模板替换，而不是被插入到元素中。
- **编译模板transclude** - 将元素编译好，使得指令可以开始使用它。一般情况下需要和ngTransclude指令一起使用。 使用嵌入的好处在于链接好书可以获取到预绑定在作用域上的函数。在一个典型的初始化过程中，widget会创建一个孤立的作用域，但是嵌入并不是其中一个子成员，而是这孤立作用域的兄弟成员。这使得widget可以有一个私有的状态，并且嵌入被绑定在父作用于上。
    - true - 嵌入指令的内容。
    - 'element' - 嵌入整个元素，包括优先级较低的指令。
- **编译compile** - 这就是后面将要讲到的编译函数。
- **链接link** - 这就是后面将要讲到的链接函数。只有没有提供编译函数时才会用到这个值。

编译函数 Compile function

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

编译函数是用来处理需要修改模板DOM的情况的。因为大部分指令都不需要修改模板，所以这个函数也不常用。需要用到的例子有`ngTrepeat`，这个是需要修改模板的，还有`ngView`这个是需要异步载入内容的。编译函数接受以下参数。

- tElement - template element - 指令所在的元素。对这个元素及其子元素进行变形之类的操作是安全的。
- tAttrs - template attributes - 这个元素上所有指令声明的属性，这些属性都是在编译函数里共享的，参考章节“属性”。
- transclude - 一个嵌入的链接函数`function(scope, cloneLinkingFn)`。

注意：如果模板被克隆了，那么模版实例和链接实例可能不是同一个对象。 所以在编译函数不要进行任何DOM变形之外的操作。 更重要的，DOM监听事件的注册应该在链接函数中做，而不是编译函数中。

编译函数可以返回一个对象或者函数。

- 返回函数 - 等效于在编译函数不存在时，使用配置对象的`link`属性注册的链接函数。
- 返回对象 - 返回一个通过`pre`或`post`属性注册了函数的对象- 使你能更具体的链接函数的执行点。参考下面`pre-linking`和`post-liking`函数的解释。

链接函数 Linking function

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

链接函数负责注册DOM事件和更新DOM。它是在模板被克隆之后执行的。 它也是大部分指令逻辑代码编写的地方。

- scope - 指令需要监听的作用域。
- iElement - instance element - 指令所在的元素。只有在`postLink`函数中对元素的子元素进行操作才是安全的，因为那时它们才已经全部连接好。
- iAttrs - instance attributes - 实例属性，一个标准化的、所有声明在当前元素上的属性列表，这些属性在所有链接函数间是共享的。参考“属性”。
- controller - 控制器实例，如果至少有一个指令定义了控制器，那么这个控制器就会被传递。控制器也是指令间共享的，指令可以用它来相互通信。

Pre-linking function

在子元素被链接前执行。不能用来进行DOM的变形，以为可能导致链接函数找不到正确的元素来链接。

Post-linking function

所有元素都被链接后执行。可以操作DOM的变形。

属性 Attributes

The Attributes object属性对象 - 作为参数传递给链接函数和编译函数。这使得下列资源可以被使用。

- 标准化的属性名: 因为指令的名称，如`ngBind`可以有很多种变形表示，如`ng:bind`，或者`x-ng-bind`，这个对象使得可以用标准的名称获取到相应的属性。
- 指令间通信：所有指令间共享同一个属性对象的实例，这使得指令可以通过这个属性对象通信。
- 支持替换式：属性中若包含替换式，那么其他指令能够独到替换式的值。
- 监视替换式属性：使用$observe 能监视使用了替换式的属性(比如`src="{{bar}}"`)。这是一种高效的，也是为唯一的方法来获取变量的值。因为在链接阶段替换式还没有被替换成值，所有变量此时是undefined。
  function linkingFn(scope, elm, attrs, ctrl) {
  // get the attribute value
  console.log(attrs.ngModel);

```
// change the attribute
attrs.$set('ngModel', 'new value');

// observe changes to interpolated attribute
attrs.$observe('ngModel', function(value) {
    console.log('ngModel has changed value to ' + value);
});
}
```

理解嵌入和作用域

开发者总是希望组件能尽量重用。下面是一个简单地对话框组件的伪代码。

```
<button ng-click="show=true">show</button>
<dialog title="Hello {{username}}."
    visible="show"
    on-cancel="show = false"
    on-ok="show = false; doSomething()">
    Body goes here: {{username}} is {{title}}.
</dialog>
```

点击`show`按钮会打开对话框，对话框有一个标题， 标题绑定了一个`username`。还包含了一个我们希望嵌入对话框的`body`。

下面是一个模板对话框组件模板的例子。

```
<div ng-show="show">
  <h3>{{title}}</h3>
  <div class="body" ng-transclude></div>
  <div class="footer">
    <button ng-click="onOk()">Save changes</button>
    <button ng-click="onCancel()">Close</button>
  </div>
</div>
```

在我们正确设置好作用域前，它不会被正确渲染。

第一个要处理的问题就是，对话框模板希望标题是被定义好的，但是初始化的时候需要绑定成`username`。然后，onOK和onCancel函数要在作用域里被定义好，才能让button按钮起作用。这限制了组件的重用。要解决这个映射的问题，我们需要创建一个本地的作用域：

```
scope: {
  title: '=',             // set up title to accept data-binding
  onOk: '&',              // create a delegate onOk function
  onCancel: '&',          // create a delegate onCancel function
  show: '='
}
```

在组件上创建本地作用域会造成两个问题：

- 孤立 - 如果用户忘了给`title`属性赋值，那么模板就会绑定到父作用域的值上。这是一个比较讨厌的问题。
- 嵌入 - 嵌入的DOM可以获取到组件的本地变量， 这可能是的嵌入需要的用来做数据绑定的属性被覆

写。在我们的例子里，组件的`title`属性覆写了嵌入的`title`属性。

要解决作用域不独立的问题，指令需要声明一个新的孤立作用域。它不从父作用域作原型继承，这样我们也不用担心会发生属性的覆盖。

但是孤立作用域也会导致一个新问题：如果嵌入的DOM是孤立作用域的一个子元素，那它就不会被绑定任何东西。为了解决这个问题，需要在组件创建本地作用域前， 让嵌入的作用域成为子作用域的一个属性，这让组件的作用域和嵌入内容成为兄弟节点。

这可能感觉有点复杂，但是却能给widget的使用者和开发者惊喜。

所以，指令最后会定义成这个样子：

```
transclude: true,
scope: {
  title: 'bind',          // set up title to accept data-binding
  onOk: 'expression',     // create a delegate onOk function
  onCancel: 'expression', // create a delegate onCancel function
  show: 'accessor'        // create a getter/setter function for visibility.
}
```

创建组件

通常需要使用更复杂的DOM结构替换单个指令。这允许指令成为一个可以生成应用程序可重用组件的短标志。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

《AngularJS开发指南05：指令》上有 1 条评论

dark 在 2013年1月24日11:58 说道： 登录以回复

作用域scope- 如果被定义成：
那么就会为当前指令创建一个新的作用域。

————————

少了一个单词：true，

应该为：

作用域scope- 如果被定义成：
true 那么就会为当前指令创建一个新的作用域。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南06：表达式

发表于 2012年10月4日 rainer_H

表达式是类似Javascript的代码片段，通常在绑定中用到，写在双大括号中如`{{表达式}}`。表达式是用$parse方法来处理的。

下面是一些合法的AngularJS表达式

- `1+2`
- `3*10 | currency`
- `user.name`

AngularJS表达式 与Javascript表达式的比较

你可能会认为AngularJS视图中的表达式就是Javascript表达式，这种认识不完全对，因为AngularJS不会用Javascript的`eval()`函数去执行表达式。 不过除了以下几个需要区别的地方以外，你可以把AngularJS表达式看成是Javascript表达式：

- **属性表达式**：属性表达式是对应于当前的作用域的，不像Javascript对应的是全局window对象。
- **允许未定义值**：执行表达式时，AngularJS能够允许undefined或者null，不像Javascript会抛出一个异常。
- **没有控制结构**： 你不能在AngularJS表达式中使用“条件判断”、“循环”、“抛出异常”等控制结构。
- **过滤器(类似unix中的管道操作符)**： 你可以通过过滤器链来传递表达式的结果。例如将日期对象转变成指定的阅读友好的格式。

如果你想要在表达式中使用标准的Javascript，那么你应该把它写成一个控制器的方法，然后在表达式中调用这个方法。如果你想在Javascript中执行AngularJS表达式，你可以使用$eval()方法。

例子

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
  </head>
  <body>
    1+2={{1+2}}
  </body>
</html>
```

端对端测试：

```
it('should calculate expression in binding', function() {
  expect(binding('1+2')).toEqual('3');
});
```

你可以用下面的例子试试别的表达式：

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Cntl2" class="expressions">
      Expression:
      <input type='text' ng-model="expr" size="80"/>
      <button ng-click="addExp(expr)">Evaluate</button>
      <ul>
       <li ng-repeat="expr in exprs">
         [ <a href="" ng-click="removeExp($index)">X</a> ]
         <tt>{{expr}}</tt> => <span ng-bind="$parent.$eval(expr)"></span>
```

```
        </li>
      </ul>
    </div>
  </body>
</html>
```

script.js：

```
function Cntl2($scope) {
  var exprs = $scope.exprs = [];
  $scope.expr = '3*10|currency';
  $scope.addExp = function(expr) {
     exprs.push(expr);
  };

  $scope.removeExp = function(index) {
    exprs.splice(index, 1);
  };
}
```

端对端测试：

```
it('should allow user expression testing', function() {
   element('.expressions :button').click();
   var li = using('.expressions ul').repeater('li');
   expect(li.count()).toBe(1);
   expect(li.row(0)).toEqual(["3*10|currency", "$30.00"]);
});
```

属性表达式

属性表达式计算是发生在作用域中的。Javascript默认是以window为作用域的。AngularJS要使用window作用域的话得用$window来指向全局window对象。比如说，你使用window中定义的`alert()`方法，在AngularJS表达式中必须写成`$window.alert()`才行。这是为了防止意外进入全局作用域（各种bug的来源）而设计的。

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div class="example2" ng-controller="Cntl1">
      Name: <input ng-model="name" type="text"/>
      <button ng-click="greet()">Greet</button>
    </div>
  </body>
</html>
```

script.js：

```
function Cntl1($window, $scope){
  $scope.name = 'World';

  $scope.greet = function() {
    ($window.mockWindow || $window).alert('Hello ' + $scope.name);
  }
}
```

端对端测试：

```
it('should calculate expression in binding', function() {
  var alertText;
  this.addFutureAction('set mock', function($window, $document, done) {
    $window.mockWindow = {
      alert: function(text){ alertText = text; }
    };
    done();
  });
  element(':button:contains(Greet)').click();
  expect(this.addFuture('alert text', function(done) {
    done(null, alertText);
  })).toBe('Hello World');
});
```

允许未定义值

表达式在执行时是可以允许`undifined`和`null`的。 在Javascript中，计算a.b.c会抛出一个异常，如果这不是一个对象的话。这对大部分的语言来说是有意义的，但是如果大多数时候表达式是用来作数绑定，像下面这种形式：

```
{{a.b.c}}
```

那么表达式返回一个空值会比触发异常更有意义。因为通常我们是在等待服务器的响应，并且变量马上就会被定义和赋值。如果表达式不能容忍未定义的值，那么我们绑定的代码就不得不写成形如：

```
{{((a||{}).b||{}).c}}
```

执行未定义的函数`a.b.c()`也会返回undefined，不会触发异常。

没有流程控制结构

你不能在表达式中使用控制结构。这样设计得原因在于AngularJS的设计理念之一就是逻辑代码都应该在控制器里。如果你需要使用条件、循环、或者处理异常，你就应该写在控制器的方法里。

过滤器

当将数据呈献给用户时，你很可能需要将数据转换为阅读友好的格式。比方说，你可能需要在显示之前将一个日期对象转换为用户本地的时间格式。你可以用链式的过滤器来传递表达式，像下面这样：

```
name | uppercase
```

这个表达式会将`name`的值传递给`uppercase`这个过滤器。

链式过滤器使用的是下面这样的语法：

```
value | filter1 | filter2
```

你也可以通过冒号来给过滤器传递参数，比如，将123显示成带有两位小数的形式：

```
123 | number:2
```

$符号

你可能会好奇，这个$的前缀有什么用?其实这只是一个标记AngularJS专有属性方法的符号，用来表示区别于其他用户自定义函数的符号。如果AngularJS不用$，那么执行`a.length()`会返回undeifined，因为a和angular都没有这样一个属性。

但是考虑到我们以后可能会增加一个length的方法（这会改变表达式的行为），同时你也想要自己增加一个length方法的话，那就会产生冲突。这种冲突可能出现都是因为，AngularJS的设计是在已有的对象上添加行为。使用$做前缀的话，就能使得开发者的代码和AngularJS和谐共处了。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

《AngularJS开发指南06：表达式》上有 2 条评论

zhangdi 在 2013年1月7日10:49 说道： 登录以回复

这篇文章中文版和英文版我反复看了很多遍了。这里要特别注意表达式与普通JS语句的不同。

Jerry.M 在 2012年10月4日11:37 说道： 登录以回复

支持Node.js+AngularJs，关注。。。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南07：表单

发表于 2012年10月10日 rainer_H

表单控件(input, select, textarea)是用来获取用户输入的。表单则是一组有联系的表单控件的集合。

用户能通过表单和表单控件提供验证的服务，知道自己的输入是否合法。这样能让用户交互变得友好，因为用户能通过反馈来修正自己的错误。不过，虽然客户端的验证能够起到很大作用，但也很容易被绕过，所以不能完全依靠客户端验证。 要建立安全的应用，服务器端验证还是必不可少的。

简单表单

了解AngularJS双向绑定的关键在于了解`ngModel`指令。这个指令通过动态将model和view互相映射，来实现双向绑定。此外它还提供API给其它指令来增强它们的行为。

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form novalidate class="simple-form">
        Name: <input type="text" ng-model="user.name" /><br />
        E-mail: <input type="email" ng-model="user.email" /><br />
        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />
        <button ng-click="reset()">RESET</button>
        <button ng-click="update(user)">SAVE</button>
      </form>
      <pre>form = {{user | json}}</pre>
      <pre>master = {{master | json}}</pre>
    </div>
  </body>
</html>
```

script.js :

```
function Controller($scope) {
  $scope.master= {};

  $scope.update = function(user) {
    $scope.master= angular.copy(user);
  };

  $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
  };

  $scope.reset();
}
```

注意例子中的`novalidate`是用来屏蔽浏览器本身的验证功能的。

使用css的类名

为了能美化表单和表单元素，ngModel指令会自动为元素添加以下css类：

- ng-valid
- ng-invalid
- ng-pristine
- ng-dirty

下面的例子演示了如何使用CSS来显示表单的验证结果。其中的user.name和user.email是必填项，它们没有被填写就提交时，底色会变红。这能避免用户注意力一开始就被分散。只有在与界面交互之后才显示错误。

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form novalidate class="css-form">
        Name:
          <input type="text" ng-model="user.name" required /><br />
        E-mail: <input type="email" ng-model="user.email" required /><br />
        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />
        <button ng-click="reset()">RESET</button>
        <button ng-click="update(user)">SAVE</button>
      </form>
    </div>

    <style type="text/css">
      .css-form input.ng-invalid.ng-dirty {
        background-color: #FA787E;
      }

      .css-form input.ng-valid.ng-dirty {
        background-color: #78FA89;
      }
    </style>
  </body>
</html>
```

script.js：

```
function Controller($scope) {
  $scope.master= {};

  $scope.update = function(user) {
    $scope.master= angular.copy(user);
  };

  $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
  };

  $scope.reset();
}
```

与表单的状态或者表单元素状态绑定

一个表单就是一个`FormController`的实例。表单实例可以通过name属性选择性地公开到作用域中。同样的，一个表单控件也是一个`NgModelController`的实例。表单控件也能同样的被公开到作用域中。这意味视图能通过绑定的基本功能获取表单或者表单控件的状态。

这些特点让我们能将上面的例子改造成下面这样：

- 只有当表单发生改变时，重置按钮才会被显示出来。
- 只有当表单有改变并且改变的值都是合法的，保存按钮才会被显示出来。
- 能自定义user.email和user.agree的错误信息。

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form name="form" class="css-form" novalidate>
        Name:
          <input type="text" ng-model="user.name" name="uName" required /><br />
        E-mail:
          <input type="email" ng-model="user.email" name="uEmail" required/><br />
        <div ng-show="form.uEmail.$dirty && form.uEmail.$invalid">Invalid:
          <span ng-show="form.uEmail.$error.required">Tell us your email.</span>
          <span ng-show="form.uEmail.$error.email">This is not a valid email.</span>
        </div>

        Gender: <input type="radio" ng-model="user.gender" value="male" />male
```

```
        <input type="radio" ng-model="user.gender" value="female" />female<br />

        <input type="checkbox" ng-model="user.agree" name="userAgree" required />
        I agree: <input ng-show="user.agree" type="text" ng-model="user.agreeSign"
                 required /><br />
        <div ng-show="!user.agree || !user.agreeSign">Please agree and sign.</div>

        <button ng-click="reset()" ng-disabled="isUnchanged(user)">RESET</button>
        <button ng-click="update(user)"
                ng-disabled="form.$invalid || isUnchanged(user)">SAVE</button>
      </form>
    </div>
  </body>
</html>
```

script.js :

```
function Controller($scope) {
  $scope.master= {};

  $scope.update = function(user) {
    $scope.master= angular.copy(user);
  };

  $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
  };

  $scope.isUnchanged = function(user) {
    return angular.equals(user, $scope.master);
  };

  $scope.reset();
}
```

自定义验证

AngularJS实现了大部分常见的html5表单输入元素(text, number, url, email, radio, checkbox)，也实现了很多用于验证的指令(required, pattern, minlength, maxlength, min, max)。

想要定义你自己的验证器的话，可以通过在你自己的指令中添加一个验证函数给ngModel的控制器来实现。要想获得控制器的引用，需要在指令中指定依赖，像下面的例子一样。验证函数可以写在两个地方。

- 模型到视图的更新中- 只要绑定的模型改变了，NgModelController#$formatters数组中的函数就会被轮流调用，所以每一个函数都有机会对数据进行格式化或者通过NgModelController#$setValidity来改变表单的验证状态。
- 视图到模型的更新中- 相同的，只要用户与表单实现了就会，NgModelController#$setViewValue就会被调用。 这次是NgModelController#$parsers数组中的函数会被依次调用，每个函数都有机会来改变值或者通过NgModelController#$setValidity来改变表单的验证状态。

下面的例子中，我们创建了两个指令。

- 第一个要验证的是输入是否是整数。例如，1.23就不是符合验证的值，因为它包含了分数部分。注意，我们是将验证数组中的项从头取出，而不是压入。这是因为我们要在输入值被改变（格式化）前，先验证输入的合法性。
- 第二个指令是一个“智能浮点(smart-float)”。它能把"1.2"或者"1,2"都转化为合法的浮点数"1.2"。注意，这里我们不能使用“数字输入类型”，因为H支持TML5的浏览器不允许用户输入像"1,2"这样的非法值。

index.html :

```
<!doctype html>
<html ng-app="form-example1">
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form name="form" class="css-form" novalidate>
        <div>
          Size (integer 0 - 10):
          <input type="number" ng-model="size" name="size"
                 min="0" max="10" integer />{{size}}<br />
          <span ng-show="form.size.$error.integer">This is not valid integer!</span>
          <span ng-show="form.size.$error.min || form.size.$error.max">
```

```
            The value must be in range 0 to 10!</span>
        </div>

        <div>
          Length (float):
          <input type="text" ng-model="length" name="length" smart-float />
          {{length}}<br />
          <span ng-show="form.length.$error.float">
            This is not a valid float number!</span>
        </div>
      </form>
    </div>
  </body>
</html>
```

script.js :

```
var app = angular.module('form-example1', []);

var INTEGER_REGEXP = /^\-?\d*$/;
app.directive('integer', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (INTEGER_REGEXP.test(viewValue)) {
          // it is valid
          ctrl.$setValidity('integer', true);
          return viewValue;
        } else {
          // it is invalid, return undefined (no model update)
          ctrl.$setValidity('integer', false);
          return undefined;
        }
      });
    }
  };
});

var FLOAT_REGEXP = /^\-?\d+((\.|\,)\d+)?$/;
app.directive('smartFloat', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (FLOAT_REGEXP.test(viewValue)) {
          ctrl.$setValidity('float', true);
          return parseFloat(viewValue.replace(',', '.'));
        } else {
          ctrl.$setValidity('float', false);
          return undefined;
        }
      });
    }
  };
});
```

实现自定义的表单控件(用ngModel)

AngularJS已经实现了所有基本的HTML表单控件（input,select,textarea），对于大部分情况应该已经够了。但是，你还是可以通过写指令来实现你自己的表单控件。

要和ngModel指令协同工作实现自定义控件，并且实现双向绑定的话，需要：

- 实现render方法。这个方法负责在数据传递给NgModelController#$formatters后来渲染数据。
- 在用户与控件交互并且模型需要被更新时，调用$setViewValue方法。这通常是在DOM的监听事件中完成的。（查看$compileProvider.directive来获取更多信息）

下面的例子演示了如何添加一个“内容可编辑”的数据双向绑定的元素。

index.html :

```
<!doctype html>
<html ng-app="form-example2">
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div contentEditable="true" ng-model="content" title="Click to edit">Some</div>
    <pre>model = {{content}}</pre>

    <style type="text/css">
```

```
      div[contentEditable] {
        cursor: pointer;
        background-color: #D0D0D0;
      }
    </style>
  </body>
</html>
```

script.js :

```
angular.module('form-example2', []).directive('contenteditable', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      // view -> model
      elm.bind('blur', function() {
        scope.$apply(function() {
          ctrl.$setViewValue(elm.html());
        });
      });

      // model -> view
      ctrl.$render = function(value) {
        elm.html(value);
      };

      // load init value from DOM
      ctrl.$setViewValue(elm.html());
    }
  };
});
```


rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南08：i8n和l10n

发表于 2012年10月13日 rainer_H

什么是i18n和l10n?

国际化，简写为i18，指的是使产品快速适应不同语言和文化。本地化，简称l10n，是指使产品在特定文化和语言市场中可用。对开发者来说，国际化一个应用意味着将所有的文字和其他因地区而异的数据从应用中抽离出来。本地化意味着为这些抽离的数据和文字提供翻译和转变成本地的格式。

AngularJS目前对本地化和国际化的支持程度

目前，AngularJS支持日期，数字和货币的国际化和本地化。

另外，AngularJS还通过`ngPluralize`指令支持本地多元化。

所有的AngularJS本地化组件都依赖于$locale服务管理本地化规则(locale-specific rule sets)。

如果你想直接看例子，我们提供了一些来展示使用了丰富本地化规则的过滤器。你可以在Github或者AngularJS开发包的i18n/e2e文件夹下找到这些例子。

什么是地区 id

一个地区(`locale`)是指一个按地理上，政治上，文化上划分的区域。常用的地区ID由两部分组成:语言代号和国家代号。比如，en-US, en-AU, zh-CN就是合法的地区ID，它们都含有语言代号和国家代号。 因为国家代号是可选的。所以像en, zh, and sk这样的id也是合法的。查看ICU网站来获取更多的地区ID信息。

对于AngularJS支持的地区，AngularJS将数字，事件格式等规则分隔放在不同的文件里，每个文件对应一个指定地区。你在AngularJS文件夹里可以找到一系列支持的的地区。

使用新的地区规则

有两种方法能使用新的地区规则：

1. 预捆绑规则

你可以通过将地区文件与angular.js或者angular.min.js捆绑来实现对你想要使用的地区文件的预绑定。

比如在*nix系统中，你可以按下面这样创建一个包含德国地区规则的angular.js文件：

```
cat angular.js i18n/angular-locale_de-ge.js > angular_de-ge.js
```

当应用使用angular_de-ge.js脚本而不是angular.js脚本时，AngularJS或自动使用德国本地化规则。

2. 将地区脚本载入到index.html

你也可以将指定的地区文件载入到index.html中。比如，当有客户端请求来自德国，你可以返回index_de-ge.html，这个文件看起来像下面这样：

```
<html ng-app>
 <head>
….
   <script src="angular.js"></script>
   <script src="i18n/angular-locale_de-ge.js"></script>
….
 </head>
</html>
```

两种方法的对比

以上两种都需要你能先准备好适合不同地区的页面。你同时也要配置好服务器来返回正确配置好的文件。

但是，第二种方式（在index.html中加载脚本）要更慢一些，因为需要额外加载脚本。

其他细节

货币符号

AngularJs的货币过滤器允许你使用地区服务里的默认货币符号，但是你也可以使用你自己定义了货币符号的过滤器。如果你的应用只在一个地区使用，那么使用默认的钱币符很好。但是如果你的应

用在多地区使用， 你应该提供你自己的钱币符来确保钱币值能被正确理解。

比如说，如果你想显示1000美元的账户余额，使用下面这种带有钱币过滤器的表达式{{1000|currency}},并且你的应用目前是在en-US地区，那么'$1000.00'会被显示出来。但是，如果用户在其他地方使用（比如说，中国的钓鱼岛）你的应用，它的浏览器会将地区指定成CN，'￥1000.00'就会被显示出来。这会迷惑你的用户。

在这个例子中，你需要通过自己指定钱币过滤器来替换默认的钱币符。这个过滤器要有一个你自己指定的钱币符，比如USD$1,000.00。这样，AngularJS就总是会显示'USD$1000'并且忽略掉地区的转换。

译文的长度

要记住，译文的长度可能和原文有很大区别。比如说，`June 3, 1977`在西班牙会被翻译成`3 de junio de 1977`。而且还有更多极端的情况。所以，在对你的应用进行国际化时，你需要好好的写CSS规则并且做好测试。

时区

注意，AngularJS使用的是浏览器的时区设置。所以一个相同的应用会根据不同的设备显示不同的时间格式。Javascript和AngularJS都不能支持使用统一的开发者制定的时区。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南09：IE浏览器兼容性

发表于 2012年10月15日 rainer_H

概览

本章描述了IE在处理自定义的HTML属性和标签时的一些独特之处。如果你要让你的AngularJS应用兼容IE8和IE8以下的版本的话，你需要仔阅读本章。

简易版

要让你的AngularJS应用在IE中正常运行你必须：

1. 确保JSON字符串能被正常解析（IE7需要），你可以使用JSON2或者JSON3来实现。
2. 你不能使用自定义的元素标签，如`<ng:view>`（你只能使用属性的形式，如`<div ng-view>`），或者，
3. 如果你用了自定义的标签名，那你必须按照以下步骤做才能保证IE正常运行：

```
<html xmlns:ng="http://angularjs.org">
 <head>
  <!--[if lte IE 8]>
    <script>
       document.createElement('ng-include');
       document.createElement('ng-pluralize');
       document.createElement('ng-view');
       // Optionally these for CSS
      document.createElement('ng:include');
      document.createElement('ng:pluralize');
      document.createElement('ng:view');
     </script>
  <![endif]-->
</head>
<body>
    ...
</body>
</html>
```

其中**重要的部分**是:

- `xmlns:ng` - 命名空间 - 你需要为你使用或者准备使用的每一个自定义标签准备一个命名空间。
- `document.createElement`(你的标签名) - 自定义标签的创建 - 因为这只是老版本IE的一个问题，所以你需要根据情况使用。对于每一个你没有使用命名空间或者HTML中没有定义的标签，你需要预先声明它彩色能使IE正常工作。

详细版

IE处理非标准标签名会产生问题。问题可以分为两类，每类都有自己的解决方法。

- 如果标签名是以`my:`前缀开始的: 这会被当成是一个XML的命名空间，并且必须使用`来声明。
- 如果标签没有`:`符号，但它又不是一个标准的HTML标签。那么就必须预先使用`document.createElement('my-tag')`来创建它。
- 如果你准备使用css选择器来对自定义标签添加样式，那么你就必须先用`document.createElement('my-tag')`来创建一下，不管有没有XML命名空间。

好消息

值得庆幸的是，IE的这种限制只存在在标签名上，标签属性名没有限制。所以，当在IE上使用`<div my-tag your:tag> </div>.`这样的形式时没有特殊要求。

如果我没按上面说的做会怎么样？

假设你用了一个非HTML标准的标签，暂时称为`mytag`(称为my:tag 或者 my-tag都可以)：

```
<html>
  <body>
    <mytag>some text</mytag>
  </body>
</html>
```

它本该被解析成下面这样的DOM：

```
#document
  +- HTML
     +- BODY
        +- mytag
           +- #text: some text
```

期望的结果是BODY元素包含一个叫做`mytag`的子元素。这个子元素包含文本"some text"。

但是IE不会这样解析（如果没有按之前强调修复IE的步骤做的话）：

```
#document
  +- HTML
     +- BODY
        +- mytag
        +- #text: some text
        +- /mytag
```

在IEE中，BODY元素有了三个子元素：

1. 一个自闭和的标签mytag。和自闭和标签
一样。最后的关闭符/是可选的，但是
   标签并不允许包含子元素。所以浏览器会将
   some text
   解析成三个兄弟节点，而不是一个包含文本的节点。
2. 一个纯文字的节点。这本来应该是上面mytag的子节点，而不是兄弟节点。
3. 一个不合法的自闭和标签`/mytag`。 因为标签名不允许含有`/`，所以说它是非法的。另外这个关闭标签不应该是DOM的一部分，因为它是用来描述标签的结尾位置的。

自定义标签名元素的CSS样式

要让CSS选择器能正确使用自定义元素的话，先要用`document.createElement('my-tag')`来创建一下元素，不管有没有XML命名空间。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

《AngularJS开发指南09：IE浏览器兼容性》上有 1 条评论

lovexiaobai 在 2013年2月2日09:22 说道： 登录以回复

非自定义标签,如果要ie7以下跑起来.单单引入json2 .js是不够的哇，还要在ng-app 这个地方加上 id=“ng-app”才行呢。希望站长可以加上去呢？

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南10：介绍

发表于 2012年10月17日 rainer_H

AngularJS是纯客户端技术，完全用Javascript编写的。它使用的是网页开发的常规技术(HTML,CSS,Javascript)，目的是让网页应用开发更快更容易。

AngularJS简化应用开发的一个重要方法是，将一个些通用的低级开发操作包装起来提供给开发者。AngularJS会自动处理好这些低级操作。它们包括：

- DOM操作
- 设置好事件的监听和通知
- 输入验证

  因为AngularJS会处理大部分这些操作，所以开发者就能更多的专注在应用逻辑上，更少地编写那些重复性的、易错的、低级的代码。

在AngularJS简化开发的同时，它也为客户端带来了一些精巧的技术，它们包括：

- 数据、逻辑、视图的分离
- 数据和视图间的自动绑定
- 通用服务(用可替换的对象实现的通用的应用操作)
- 依赖注入（主要用于聚合服务）
- 可扩展的HTML编译器(完全由JavaScript编写)
- 易于测试

客户端对这些技术的需求其实已经存在很久了。

单页或者多页应用

你可以用AngularJS来开发单页或者多页的应用，不过主要是用来开发单页的。 AngularJS支持浏览器历史操作，向前向后按钮，单页应用中的收藏操作。

你不会想在每一次页面切换时都载入AngularJS。比如多页应用就常需要切换页面。 但是，如果你只是给已有的多页应用上加上一些AngularJS提供的功能（比如使用AngularJS模板的数据绑定功能）的话，那就有意义了。你可以按照后面的教程来将已有的应用整合成单页应用。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南11：模块

发表于 2012年10月18日 rainer_H

什么是模块?

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

- 启动过程是声明式的，所以更容易懂。
- 在单元测试是不需要加载全部模块的，因此这种方式有助于写单元测试。
- 可以在特定情况的测试中增加额外的模块，这些模块能更改配置，能帮助进行端对端的测试。
- 第三方代码可以打包成可重用的模块。
- 模块可以以任何先后或者并行的顺序加载（因为模块的执行本身是延迟的）。

最基本的

好吧，我赶时间，快告诉我怎么写个 `hello world` 的模块。

你需要了解以下两条重要的信息：

- Module API模块API
- 注意我们是通过在 `<html ng-app="myApp">` 中进行指定，来实现使用myApp这个模块启动应用的。

index.html：

```
<!doctype html>
<html ng-app="myApp">
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div>
      {{ 'World' | greet }}
    </div>
  </body>
</html>
```

script.js：

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

// configure the module.
// in this example we will create a greeting filter
simpleAppModule.filter('greet', function() {
 return function(name) {
    return 'Hello, ' + name + '!';
  };
});
```

推荐写法

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

- 一个服务模块，用来做服务的声明。
- 一个指令模块，用来做指令的声明。
- 一个过滤器模块，用来做过滤器声明。
- 一个依赖以上模块的应用级模块，它包含初始化代码。

这样拆分的原因是，在你的测试中常常需要忽略掉初始化代码，因为这些代码比较难测试。通过把它拆分出来就能在测试中方便地忽视掉它。通过只加载和当前测试相关的模块，也能使测试更专一。

以上只是一个建议，你可以放心根据你的具体情况来调整。

index.html：

```
<!doctype html>
<html ng-app="xmpl">
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="XmplController">
      {{ greeting }}!
    </div>
  </body>
</html>
```

script.js：

```
angular.module('xmpl.service', []).
  value('greeter', {
    salutation: 'Hello',
    localize: function(localization) {
      this.salutation = localization.salutation;
    },
    greet: function(name) {
      return this.salutation + ' ' + name + '!';
    }
  }).
  value('user', {
    load: function(name) {
      this.name = name;
    }
  });

angular.module('xmpl.directive', []);

angular.module('xmpl.filter', []);

angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter']).
  run(function(greeter, user) {
    // This is effectively part of the main method initialization code
    greeter.localize({
      salutation: 'Bonjour'
    });
    user.load('World');
  })


// A Controller for your app
var XmplController = function($scope, greeter, user) {
  $scope.greeting = greeter.greet(user.name);
}
```

模块的加载和依赖

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

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

```
angular.module('myModule', []).
config(function(injectables) { // provider-injector
    // This is an example of config block.
    // You can have as many of these as you want.
    // You can only inject Providers (not instances)
    // into the config blocks.
  }).
  run(function(injectables) { // instance-injector
    // This is an example of a run block.
    // You can have as many of these as you want.
    // You can only inject instances (not Providers)
    // into the run blocks
  });
```

配置块

模块有一些配置的简便方法，使用它们效果等同于使用代码块。举个例子：

```
angular.module('myModule', []).
  value('a', 123).
  factory('a', function() { return 123; }).
```

```
  directive('directiveName', ...).
  filter('filterName', ...);

// is same as

angular.module('myModule', []).
  config(function($provide, $compileProvider, $filterProvider) {
    $provide.value('a', 123)
    $provide.factory('a', function() { return 123; })
    $compileProvider.directive('directiveName', ...).
    $filterProvider.register('filterName', ...);
  });
```

配置块按照它们注册的顺序依次被应用。唯一的例外是对常量的定义，它们应该放在配置块的开始处。

运行块

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

依赖

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

异步加载

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

单元测试

单元测试最简单的一种形式是实例化一个应用的子集，并且运行它。对每个注入器来说任何一个模块都只能被夹在一次。一般来说一个应用只有一个注入器。但是在测试中，每个测试都可以有自己的注入器，这也意味着模块可以再每个VM中被加载多次。结构良好的模块能很好的进行单元测试，就像下面这样：

我们假设在所有的例子都是使用下面这种定义：

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

  factory('alert', function($window) {
    return function(text) {
      $window.alert(text);
    }
  }).

  value('salutation', 'Hello').

  factory('greet', function(alert, salutation) {
    return function(name) {
      alert(salutation + ' ' + name + '!');
    }
  });
```

让我们来一些测试：

```
describe('myApp', function() {
  // load the application relevant modules then load a special
  // test module which overrides the $window with mock version,
  // so that calling window.alert() will not block the test
  // runner with a real alert box. This is an example of overriding
  // configuration information in tests.
  beforeEach(module('greetMod', function($provide) {
    $provide.value('$window', {
      alert: jasmine.createSpy('alert')
    });
  }));

  // The inject() will create the injector and inject the greet and
  // $window into the tests. The test need not concern itself with
  // wiring of the application, only with testing it.
  it('should alert on $window', inject(function(greet, $window) {
    greet('World');
    expect($window.alert).toHaveBeenCalledWith('Hello World!');
  }));

  // this is another way of overriding configuration in the
```

```
  // tests using an inline module and inject methods.
  it('should alert using the alert service', function() {
    var alertSpy = jasmine.createSpy('alert');
    module(function($provide) {
      $provide.value('alert', alertSpy);
    });
    inject(function(greet) {
      greet('World');
      expect(alertSpy).toHaveBeenCalledWith('Hello World!');
    });
  });
});
```


rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南12：作用域

发表于 2012年10月19日 rainer_H

什么是作用域?

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

作用域特点

- 作用域提供APIs（$watch）来观察模型的变化。
- 作用域提供APIs($apply)将任何模型的改变从"AngularJS领域(Angular realm)"通过系统映射到视图上。
- 作用域能通过共享模型成员的方式嵌套到应用组件上。一个作用域从父作用域继承属性。
- 作用域提供表达式执行的上下文。比如说表达式 `{{username}}` 本身是无意义的，除非把它放到指定username属性的作用域中。

作为数据模型的作用域

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

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

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="MyController">
      Your name:
        <input type="text" ng-model="username">
        <button ng-click='sayHello()'>greet</button>
      <hr>
      {{greeting}}
    </div>
  </body>
</html>
```

script.js：

```
function MyController($scope) {
  $scope.username = 'World';

  $scope.sayHello = function() {
   $scope.greeting = 'Hello ' + $scope.username + '!';
  };
}
```

上例中MyController将值World赋给了作用域中的username。然后作用域将这个赋值的操作通知给input，然后input就会被渲染成预填充了值得样子。这展示控制器如何将数据写入到作用域。

同样的，控制器能将行为添加到作用域，正如你看到的sayHello方法，这个方法是在用户点击'greet'按钮时被调用的。这展示了当属性绑定到HTML input元素上时，它会自动更新。

逻辑上来说，表达式 `{{greeting}}` 的渲染需要：

- 获取和模板中定义了 `{{greeting}}` DOM节点相关的作用域。在这个例子里，就是传入到MyController的作用域。
- 结合上一步获取到的作用域来计算表达式的值，将该值在DOM中替换掉表达式。

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

从测试角度来说，控制器和分离是我们想要的结果，因为这使得我们能在测试行为的时候排除掉

渲染细节的干扰。

```
it('should say hello', function() {
  var scopeMock = {};
  var cntl = new MyController(scopeMock);

  // Assert that username is pre-filled
  expect(scopeMock.username).toEqual('World');

  // Assert that we read new username and greet
  scopeMock.username = 'angular';
  scopeMock.sayHello();
  expect(scopeMock.greeting).toEqual('Hello angular!');
});
```

作用域层级

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

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

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

下面这个例子展示了应用中的作用域，它们的继承关系。

index.html：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="EmployeeController">
      Manager: {{employee.name}} [ {{department}} ]<br>
      Reports:
        <ul>
          <li ng-repeat="employee in employee.reports">
            {{employee.name}} [ {{department}} ]
          </li>
        </ul>
      <hr>
      {{greeting}}
    </div>
  </body>
</html>
```

style.css：

```
/* remove .doc-example-live in jsfiddle */
.doc-example-live .ng-scope {
  border: 1px dashed red;
}
```

script.js：

```
function EmployeeController($scope) {
  $scope.department = 'Engineering';
  $scope.employee = {
    name: 'Joe the Manager',
    reports: [
      {name: 'John Smith'},
      {name: 'Mary Run'}
    ]
  };
}
```

注意当作用域和元素相关联的时候，AngularJS会自动给相应元素添加ng-scope类名。这个例子中的作用域范围突出显示了。子作用域的存在是很有必要的，因为迭代器要执行`{{employee.name}}`表达式，它会根据不同的作用域生成不同的值。同样的，`{{department}}`的执行是继承自根作用域的，因为只有根作用域中定义了它。

从DOM中获取作用域

作用域是作为$scope的数据属性关联到DOM上的，并且能在需要调试的时候被获取到。（这不

像那些只能在应用内获取的框架）根作用关联的DOM就是ng-app指令定义的地方。一般来说ng-app都是放在 `<html>` 元素中的，但是也能放在其他元素中，比如，只有

在调试器中检测作用域：

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

调试器能让你在控制台中用变量 `$0` 获取到。

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

作用域事件的传递

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

`index.html:`

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="EventController">
      Root scope <tt>MyEvent</tt> count: {{count}}
      <ul>
        <li ng-repeat="i in [1]" ng-controller="EventController">
          <button ng-click="$emit('MyEvent')">$emit('MyEvent')</button>
          <button ng-click="$broadcast('MyEvent')">$broadcast('MyEvent')</button>
          <br>
          Middle scope <tt>MyEvent</tt> count: {{count}}
          <ul>
            <li ng-repeat="item in [1, 2]" ng-controller="EventController">
              Leaf scope <tt>MyEvent</tt> count: {{count}}
            </li>
          </ul>
        </li>
      </ul>
    </div>
  </body>
</html>
```

`script.js:`

```
function EventController($scope) {
  $scope.count = 0;
  $scope.$on('MyEvent', function() {
    $scope.count++;
  });
}
```

作用域的声明周期

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

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

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

1. **创建**——根作用域是在应用被$injector启动时创建的。在模板链接阶段，有些指令会创建新的子作用域。
   2.**观察者注册**——在模板链接阶段，指令会在作用域上注册观察者。这些观察者是用来将模型的改变传递给DOM的。
2. **模型变化**——为了正确地观测到模型变化，你需要并且只能在scope.$apply()中改变他们。
   （AngularJS的API会隐式地这么做，所以在控制器或者在$http,$timeout等服务中你不需要额外的调用$apply）。
3. **变化的观测**——在$apply的最后，AngularJS会在根作用域中执行一个$digest循环，它会将变化传递给所有子作用域。在$digest循环中，所有的$watch表达式或者函数都会被检测，来观察模型的变化。

如果有变化被检测到了，$watch的监听回调就会被调用。

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

作用域和指令

在编译阶段，编译器在DOM中匹配指令。指令通常分为两种：

- 观察型的指令，例如双花括号表达式 `{{expression}}` ，会用$watch来注册一个监听者。无论表达式什么时候改变，这类型的指令都会被通知，并且能更新视图。
- 监听者型的指令，比如 `ng-click` ，会向DOM注册一个监听者。当DOM监听者触发，指令会执行相关的表达式并且使用$apply方法更新视图。

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

会创建作用域的指令

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

控制器和作用域

作用域和控制器在以下几种情况下交互：

- 控制器通过作用域来向模板暴露方法（参考 `ng-controller` ）
- 控制器定义里能改变模型（作用域的属性）的方法（行为）
- 控制器在模型上注册了观察者。这些观察者会在控制器行为执行后立即被执行

参考 `ng-controller` 来获取更多信息。

作用域的$watch操作要注意的

检测属性的改变是AngularJS中一项常用的操作，所以它应该是高效的。要注意的是，执行检测的方法不应该包含任何DOM操作，因为在Javascript对象中，DOM获取要比属性获取慢很多很多。

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南13：暂无内容

发表于 2012年10月20日 rainer_H

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南14：依赖注入

发表于 2012年10月20日 rainer_H

依赖注入

依赖注入是一种软件设计模式，用来处理代码的依赖关系。

要更多地了解依赖注入，你可以到wikipedia上查找依赖注入，Martin Fowler写的Inversion of Control，或者到你喜欢的讲设计模式的书中查找。

简单说说依赖注入

一般来说有三种方法让函数获得它的需要的依赖：

1. 它的依赖是能被创建的，一般用new操作符就行。
2. 能够通过全局变量查找依赖。
3. 依赖能在需要时被导入。

前两种方式都不是很好，因为它们需要对依赖硬编码，使得修改依赖的时候变得困难。特别是在测试的时候不好办，因为对某个部分进行孤立的测试常常需要模拟它的依赖。

第三种方式是最好的，因为它不必在组件中去主动需找和获取依赖，而是由外界将依赖传入。

```
function SomeClass(greeter) {
  this.greeter = greeter
}

SomeClass.prototype.doSomething = function(name) {
  this.greeter.greet(name);
}
```

上面的例子中SomeClass不关心去哪里获得叫greeter的依赖，而只是在运行时处理一下greeter。

这是我们想要的结果，但是这也同时把获取依赖的任务交给了负责创建SomeClass的代码。

每一个AngularJS应用都有一个注入器(injector)用来处理依赖的创建。注入器是一个负责查找和创建依赖的服务定位器。

下面是使用注入器服务的例子：

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

  // Teach the injector how to build a 'greeter'
  // Notice that greeter itself is dependent on '$window'
  factory('greeter', function($window) {
    // This is a factory function, and is responsible for
    // creating the 'greet' service.
    return {
      greet: function(text) {
        $window.alert(text);
      }
    };
  }).

// New injector is created from the module.
// (This is usually done automatically by angular bootstrap)
var injector = angular.injector('myModule');

// Request any dependency from the injector
var greeter = injector.get('greeter');
```

通过请求依赖的方式解决了硬编码的问题，但是同样也意味着注入器需要再应用中传递，这违反了迪米特法则。我们通过使用下面这个例子中的声明的方式来将依赖查找都给注入器来解决。

```
<!-- Given this HTML -->
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
// And this controller definition
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter('Hello World');
  };
}

// The 'ng-controller' directive does this behind the scenes
injector.instantiate(MyController);
```

注意，通过使用`ng-controller`来实例化控制器类，是的控制器和注入器不再相关联。这是最好的结果。应用代码可以简单的请求依赖而不必处理注入器。这种方式就没有破坏迪米特法则。

依赖标记

注入器怎么知道需要注入什么依赖呢？

注入器需要应用提供一些标记来表示自己需要的依赖。在关于AngularJS的某些API文档中你会看到函数都是被注入器调用的。注入器需要知道函数需要什么依赖。下面有三个等效的表示的自己需要的依赖的方法。这些方法可以互相替换，并且是等效的。

推断依赖

最简单的处理依赖的方法，就是假设函数的参数名就是依赖的名字

```
function MyController($scope, greeter) {
  ...
}
```

给出一个注入器可以通过检查声明来获取函数名，从而知道需要的依赖的函数。在上面的例子中，`$scope`和`greeter`是需要注入到函数中的依赖。

坦白的来讲，用了这种方法就不能使用`JavaScript minifiers/obfuscators`（一些用来缩短的JS的类库）了，因为它们会改变变量名。这使得这种方法只适合于`pretotyping`和做demo。

$inject 标记

要允许压缩类库重命名函数参数，同时注入器又能正确处理依赖的话，函数需要使用$inject属性。这个属性是一个包含依赖的名称的数组。

```
var MyController = function(renamed$scope, renamedGreeter) {
  ...
}
MyController.$inject = ['$scope', 'greeter'];
```

注意$inject标记里的值和函数声明的参数是对应的。

这种方式适合用于控制器的声明，因为控制器有了明确的声明标记。

行内标记

有时候用$inject标记不是很方便，比如用来声明指令的时候。

比如：

```
someModule.factory('greeter', function($window) {
  ...;
});
```

使用$inject会导致代码膨胀：

```
var greeterFactory = function(renamed$window) {
  ...;
};

greeterFactory.$inject = ['$window'];

someModule.factory('greeter', greeterFactory);
```

这种情况我们就推荐使用第三种方式

```
someModule.factory('greeter', ['$window', function(renamed$window) {
  ...;
}]);
```

记住这三种方式是等效的，并且在AngularJS应用中注入器支持的情况下可以随处用。

哪里可以使用依赖注入

依赖注入再AngularJS中很普遍。一般用在控制器和工场方法中。

控制器中的依赖注入

控制器是负责应用行为的类。推荐的控制器声明方法如下：

```
var MyController = function(dep1, dep2) {
  ...
}
MyController.$inject = ['dep1', 'dep2'];

MyController.prototype.aMethod = function() {
  ...
}
```

工场方法

工场方法负责创建AngularJS中的大部分对象。比如指令，服务，过滤器。工厂方法一般在模块中使用，推荐的方法如下：

```
angular.module('myModule', []).
  config(['depProvider', function(depProvider){
    ...
  }]).
  factory('serviceId', ['depService', function(depService) {
    ...
  }]).
  directive('directiveName', ['depService', function(depService) {
    ...
  }]).
  filter('filterName', ['depService', function(depService) {
    ...
  }]).
  run(['depService', function(depService) {
    ...
  }]);
```


rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南15：MVC模式

发表于 2012年10月22日 rainer_H

自MVC模式第一次发表出来，它已经吸收了很多新的含义。AngularJS将它原本的宗旨整合进自己的模式中，更好的帮助开发浏览器端应用。

总的概括MVC模式：

- 将应用分解成独立的表现、数据、逻辑三种组件。
- 鼓励三者间的解耦。

和服务、依赖注入这两项技术一样，MVC让AngularJS应用更好地结构化，更容易实现和更容易测试。

下面的几篇解释了AngularJS如何将MVC模式整合到自己的模式中：

- 理解Model组件
- 理解Controller组件
- 理解View组件

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南16：Model组件

发表于 2012年10月23日 rainer_H

`model` 这个词在AngularJS中既可以表示一个用来表示实体（比如，一个叫做 `phones` 的 `model` ，它的值是一个包含多个 `phone` 的数组）的对象，也可以表示应用中的整个数据模型，这取决于我们所讨论的AngularJS文档中的上下文。

在AngularJS中，一个模型就是一AngularJS作用于对象的任何一个可取的属性。属性的名字就是模型的标示符。它的值可以是任意的Javascript对象（包括数组和原始对象）。

将Javascript对象编程模型的唯一要求是这个对象必须被AngularJS作用域的一个属性引用。这个引用既可以显式也可以隐式地创建。

你可以像下面这样显式地创建一个作用域属性，引用Javascript对象：

- 在Javascript代码中直接将一个对象赋给作用域对象属性；这种情况常见于控制器中：

```
function MyCtrl($scope) {
    // create property 'foo' on the MyCtrl's scope
    // and assign it an initial value 'bar'
    $scope.foo = 'bar';
}
```

- 在模板中使用表达式：
  `<button ng-click="{{foos='ball'}}">Click me</button>`
- 在模板中使用 `ngInit` 指令（只适用于实例，不推荐在实际应用中使用）：
  `<body ng-init=" foo = 'bar' ">`

当处理下面这样的模板机构时，AngularJS会隐式地（通过创建一个作用域对象的属性，并将合适地值赋给它来实现）创建模型。

- 从input, select, textarea或者其他表单元素中：
  `<input ng-model="query" value="fluffy cloud">`

  上面的代码再当前作用域中创建了一个叫做 `query` 的模型，值被设置成"fluffy cloud"。
- 在 `ngRepeater` 的迭代声明中：
  `<p ng-repeat="phone in phones"></p>`

上面代码为 `phones` 数组中的每一项都创建了一个子作用域。并且在自作用于中创建了一个叫做 `phone` 的对象（模型），它的值被设置成数组中当前的值。

在AngularJS中的下列情况中，Javascript对象也可以变成模型：

- 没有个属性里引用了Javascript对象的作用域对象。
- 所有的包含对象应用属性的作用域对象都已过期并且是会被回收的。

下面的图中描述了隐式地从模板中创建数据库模型：

相关主题

- AngularJS中的MVC
- 理解Controller组件
- 理解view组件

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

《AngularJS开发指南16：Model组件》上有 1 条评论

chaos.forfun 在 2013年2月4日14:32 说道： 登录以回复

在AngularJS中的下列情况中，Javascript对象也可以变成模型：
没有个属性里引用了Javascript对象的作用域对象。
所有的包含对象应用属性的作用域对象都已过期并且是会被回收的。

这段看蒙了，怎么看都不明白。看了原文才知道翻译错了。
原文是：In Angular, a JavaScript object stops being a model when:
在Angular中，在下列情况下JavaScript对象不再是一个模型：

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南17：Controller组件

发表于 2012年10月26日 rainer_H

在AngularJS中，控制器是一个Javascript函数（类型/类），用来增强除了根作用域意外的作用域实例的。当你或者AngularJS本身通过 `scope.$new` 俩创建一个新的子作用域对象时，有一个选项能让你将它当做参数传递给控制器。这能使AngularjS将控制器和这个作用域联系起来，增强作用域的行为。

控制器用于：

- 设置好作用域对象的初始状态。
- 给作用域对象增加行为。

给作用于对象设置初始状态

一般来说，当你创建应用时，你需要对它的作用域设置初始状态。

AngularJS将对作用域对象调用控制器的构造函数（从某种意义上来说就像使用的Javascript的apply方法），以此来设置作用域的初始状态。这意味着AngularJS不会创建控制器类型的实例（不会使用new方法来调用控制器构造函数）。控制器总是对某个已存在的作用域对象调用。

你可以通过创建一个模型属性来设置初始作用域的初始状态。 比如:

```
function GreetingCtrl($scope) { $scope.greeting = 'Hola!'; }
```

GreetingCtrl控制器创建了一个模板中可以调用的叫 `greeting` 的模型。

给作用域对象增加行为

AngularJS作用域对象的行为是由作用域的方法来表示的。这些方法是可以在模板或者说视图中调用的。这些方法和应用模型交互，并且能改变模型。

如我们在模型那一章所说的，任何对象（或者原生的类型）被赋给作用域后就会变成模型。任何赋给作用域的方法，都能在模板或者说视图中被调用，并且能通过表达式或者 `ng` 事件指令调用。（比如，ngClick）

正确地使用控制器

总的来说，一个控制器不应该做太多工作。它应该只包含单个视图的业务逻辑。

保持控制器职责单一的最常见做法是将那些不属于控制器的工作抽离到服务中，然后通过依赖注入在控制器中使用这些服务。这在依赖注入服务的章节中会详细讨论。

不要用控制器干下面的事情：

- 控制器应该只关心业务逻辑。DOM操作（表现层逻辑）通常会把测试弄得很难。将任何表现层逻辑放到控制器中都会显著地增加对业务逻辑的测试难度。AngularJS提供 `dev_guide.templates.databinding` 用来自动进行DOM操作。如果你需要手动操作DOM，将表现层的逻辑抽离到指令中。
- 对输入格式化 — 你应该用AngularJS的表单控制来实现格式化。.
- 对输出格式化 — 该用AngularJS的过滤器实现。.
- 在控制器中运行无状态或者有状态但在控制器中共享的代码 — 该用服务来实现.
- 实例化组件或者控制其它组件的生命周期（比如创建一个服务的实例）.

将控制器和AngularJS的作用域对象联系起来

你可以显示地用 `scope.$new` 来将控制器和作用域对象显示地联系起来，或者隐式地通过 `ngController` 指令或者 `$route` 服务来联系。

控制器构造函数和方法的例子

为了阐述AngularJS的控制器组件的运行原理，让我们来创建一个拥有下面这些组件的小应用：

- 一个有两个按钮和一条消息的模板
- 一个叫 `spice` 的字符串模型。
- 一个拥有两个方法的控制器。方法是用来设置 `spice` 的值得。

模板中的消息包含了一个对`spice`模型的绑定，它初始的字符串是“very”。这个spice模型会被设置成 chili 或者 jalapeno，这取决于哪个按钮会被点击。消息会通过`data-binding`自动更新。

一个 spice 控制器例子

```
<body ng-controller="SpicyCtrl">
 <button ng-click="chiliSpicy()">Chili</button>
 <button ng-click="jalapenoSpicy()">Jalapeño</button>
 <p>The food is {{spice}} spicy!</p>
</body>

function SpicyCtrl($scope) {
 $scope.spice = 'very';
 $scope.chiliSpicy = function() {
   $scope.spice = 'chili';
 }
 $scope.jalapenoSpicy = function() {
  $scope.spice = 'jalapeño';
 }
}
```

例子中有下面这些需要注意：

- `ngController`指令是用来（隐式地）为模板创建作用域的。并且使用命令中指定的`spicyCtrl`控制器来增强这个作用域。
- `spicyCtrl`只是一个纯Javascript函数。使用了驼峰式命名法（可选）命名并以**Ctrl**或者**Controller**结尾。
- 对作用域对象赋予一个新的属性会创建或者更新模型。
- 控制器方法能够直接通过赋格作用域对象这个方式创建（如例子中的`chiliSpicy`方法）。
- 控制器中的所用方法都能在模板中调用（在body元素或者子元素中）.
- NB:AngularJS的老版本（1.0RC之前的）能让你用它来替换$scope方法。但是现在不行了。在作用域中定义的方法中，他和$scope是可以互换的（AngularJS将它设置成$scope），那是在你的控制器构造函数中就不行了。
- NB: AngularJS的老版本（1.0RC之前的）自动给作用域对象原型添加方法，现在不会了。所有的方法都必须手动添加到作用域。

控制器方法可以接受参数，像下面里中演示的：

控制器方法参数的例子

```
<body ng-controller="SpicyCtrl">
 <input ng-model="customSpice" value="wasabi">
 <button ng-click="spicy('chili')">Chili</button>
 <button ng-click="spicy(customSpice)">Custom spice</button>
 <p>The food is {{spice}} spicy!</p>
</body>

function SpicyCtrl($scope) {
 $scope.spice = 'very';
 $scope.spicy = function(spice) {
   $scope.spice = spice;
 }
}
```

注意`SpicyCtrl`控制器只定义了一个叫spicy的方法，它接受一个叫做spice的参数。和这个控制器相关的模板在第一个按钮事件中传递了一个`chili`常量给控制器方法，在第二个按钮中传递一个模型属性。

控制器继承示例

AngularJS中的控制器继承是基于作用域的继承的。让我们看下面这个例子：

```
<body ng-controller="MainCtrl">
 <p>Good {{timeOfDay}}, {{name}}!</p>
 <div ng-controller="ChildCtrl">
   <p>Good {{timeOfDay}}, {{name}}!</p>
   <p ng-controller="BabyCtrl">Good {{timeOfDay}}, {{name}}!</p>
</body>

function MainCtrl($scope) {
 $scope.timeOfDay = 'morning';
 $scope.name = 'Nikki';
}

function ChildCtrl($scope) {
 $scope.name = 'Mattie';
}
```

```
function BabyCtrl($scope) {
 $scope.timeOfDay = 'evening';
 $scope.name = 'Gingerbreak Baby';
}
```

注意我们是如何在模板中嵌套我们的`ngController`指令的。这个模板结构会使得AngularJS为视图创建四个作用域：

- 根作用域
- MainCtrl作用域， 它包含了模型timeOfDay和模型name。
- ChildCtrl作用域，它继承了上层作用域的timeOfDay，复写了name。
- BabyCtrl作用域，复写了MainCtrl中定义的timeOfDay和ChildCtrl中的name。

控制器的继承和模型继承是同一个原理。所以在我们前面的例子中，所有的模型都用返回相应字符串的控制器方法代替。

注意：常规的原型继承对控制器来说不起作用。因为正如我们之前提到的，控制器不是直接实例化的，而是对作用域对象调用的。

测试控制器

尽管有很多测试控制器的方法。但最好的是像我们下面这样展示的。这个例子注入了$rootScope和$controller。

控制器函数：

```
function myController($scope) {
   $scope.spices = [{"name":"pasilla", "spiciness":"mild"},
                    {"name":"jalapeno", "spiceiness":"hot hot hot!"},
                    {"name":"habanero", "spiceness":"LAVA HOT!!"}];

   $scope.spice = "habanero";
}
```

控制器测试：

```
describe('myController function', function() {

  describe('myController', function() {
    var scope;

    beforeEach(inject(function($rootScope, $controller) {
      scope = $rootScope.$new();
      var ctrl = $controller(myController, {$scope: scope});
    }));

    it('should create "spices" model with 3 spices', function() {
      expect(scope.spices.length).toBe(3);
    });

    it('should set the default value of spice', function() {
      expect(scope.spice).toBe('habanero');
    });
  });
});
```

如果你需要测试嵌套的控制器，你需要创建和DOM中相同相同的作用域层级。

```
describe('state', function() {
    var mainScope, childScope, babyScope;

    beforeEach(inject(function($rootScope, $controller) {
        mainScope = $rootScope.$new();
        var mainCtrl = $controller(MainCtrl, {$scope: mainScope});
        childScope = mainScope.$new();
        var childCtrl = $controller(ChildCtrl, {$scope: childScope});
        babyScope = childCtrl.$new();
        var babyCtrl = $controller(BabyCtrl, {$scope: babyScope});
    }));

    it('should have over and selected', function() {
        expect(mainScope.timeOfDay).toBe('morning');
        expect(mainScope.name).toBe('Nikki');
        expect(childScope.timeOfDay).toBe('morning');
        expect(childScope.name).toBe('Mattie');
        expect(babyScope.timeOfDay).toBe('evening');
        expect(babyScope.name).toBe('Gingerbreak Baby');
    });
});
```

相关主题

- AngularJS中的MVC
- 理解Controller组件
- 理解view组件

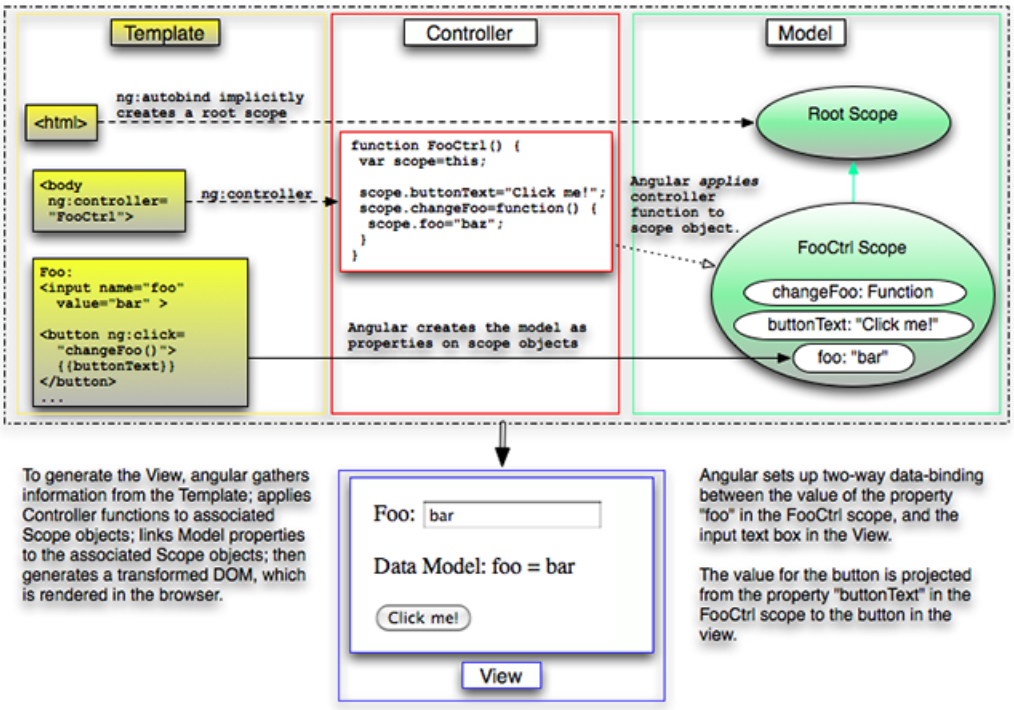
rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

AngularJS开发指南18：View组件

发表于 2012年10月27日 rainer_H

在AngularJS中，视图(view)指的是浏览器加载和渲染之后，并且在AngularJS根据模板、控制器、模型信息修改之后的DOM。



在AngularJS对MVC的实现中，视图是知道模型和控制器的。视图知道模型的双向绑定。视图通过指令知道的控制器，比如 `ngController` 和 `ngView` 指令，也可以通过绑定知道，比如 `{{someControllerFunction()}}`。通过这些方式，视图可以调用相应控制器中的方法。

相关主题

- AngularJS中的MVC
- 理解Controller组件
- 理解view组件

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南19：端到端测试

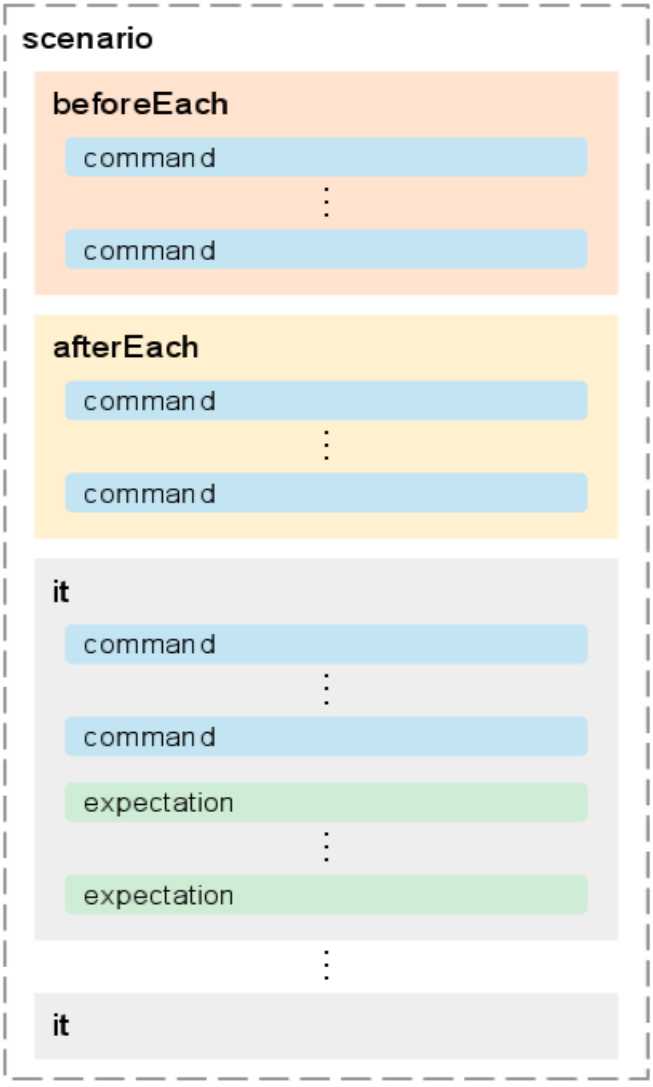
发表于 2012年10月30日 rainer_H

随着软件的规模和复杂度增长。依靠人工来进行测试越来越不现实。

为了解决这个问题，我们建立了**Angular Scenario Runner**来模拟用户交互，以此来帮助你对你的应用进行测试。

概览

方案测试(scenario tests)使用Javascript写的，你在其中描述在某一个状态的某一个交互中你的应用应该运作。一个方案包括一个或多个 `it` 代码块（你可以把这些当成你应用的需求），代码块它由命令和期望结果组成。命令告诉*Runner*让应用执行某个操作（比如翻页或者点击按钮），期望告诉*Runner*验证执行后的应用状态（比如字段的值或者当前URL）。如果某个期望不符合，那么**Runner**就会把它标记为“失败”然后继续验证下一个。方案还能有 `beforeEach` 和 `afterEach` 代码块，他们会在 `it` 代码块前（后）执行，不管代码块是否成功。



除了上面所列的，方案里还能使用一些辅助函数来避免代码块中的重复代码。

下面是一个简单的方案：

```
describe('Buzz Client', function() {
it('should filter results', function() {
  input('user').enter('jacksparrow');
  element(':button').click();
  expect(repeater('ul li').count()).toEqual(10);
  input('filterText').enter('Bees');
```

```
    expect(repeater('ul li').count()).toEqual(1);
  });
  });
```

这个方案描述了**Buzz Client**应该能过滤用户的输入。它一开始模拟用户输入，然后点击了按钮，然后它验证应该出现10个列表项。再然后它模拟在`filterText`输入框内输入`Bees`，并验证列表项减少到1个。
下面的API是命令和期望中可以用到的：

API

参阅 https://github.com/angular/angular.js/blob/master/src/ngScenario/dsl.js

pause()

暂停测试执行，知道你在命令行中执行`resume()`（或者在Runner UI中点击继续按钮）。

sleep(seconds)

将测试暂停一段指定的事件，以秒计。

browser().navigateTo(url)

将url加载到测试框架中。

browser().navigateTo(url, fn)

将函数返回的url加载到测试框架中。参数中的url是用来测试输出的。在url是动态地时候，你可以用这个api（意思是在你写测试的时候并不知道url是什么的时候）。

browser().reload()

在测试框架中刷新加载的页面。

browser().window().href()

返回测试框架页面中的window.location.href。

browser().window().path()

返回测试框架页面中的window.location.pathname。

browser().window().search()

返回测试框架页面中的window.location.search.

browser().window().hash()

返回测试框架页面中的window.location.hash.

browser().location().url()

返回测试框架页面中的$location.url().

browser().location().path()

返回测试框架页面中的$location.path().

browser().location().search()

返回测试框架页面中的$location.search().

browser().location().hash()

返回测试框架页面中的$location.hash().

expect(future).{matcher}

验证当前的future对象是否满足当前的匹配.所有的api都会返回一个future对象，它没在执行后就会对这个对象赋值。**匹配**是用angular.scenario.matcher定义的， 它用future对象的值来验证是否复合期望，比
如：`expect(browser().location().href()).toEqual('http://www.google.com')`。

expect(future).not().{matcher}

验证当前future对象值是否不满足当前匹配。

using(selector, label)

审查下一个DSL元素选项。

binding(name)

返回符合给定名字的第一个绑定值。

input(name).enter(value)

在指定输入框中输入指定值。

input(name).check()

勾选或者反勾选指定名字的checkbox。

input(name).select(value)

选择指定名字的单选框。

input(name).val()

返回指定名字的input元素的值。

repeater(selector, label).count()

返回用指定jquery选择器选择到的元素的个数。参数中label是用来检测输出的。

repeater(selector, label).row(index)

返回用指定jquery选择器选择到的元素的指定索引的绑定（数组）。参数中label是用来检测输出的。

repeater(selector, label).column(binding)

返回用指定jquery选择器选择到的元素的列的值（数组）。参数中label是用来检测输出的。

select(name).option(value)

返回指定名字的select中指定值的option。

select(name).option(value1, value2...)

返回指定名字的select中的和给定任意一个值匹配的option。

element(selector, label).count()

返回符合指定jquery选择器的元素的个数。参数中label是用来检测输出的。

element(selector, label).click()

模拟指定jquery选择器的元素的点击事件。参数中label是用来检测输出的。

element(selector, label).query(fn)

执行函数fn(selectedElements, done)，selectedElements是符合jquery选择器的元素，done是函数执行后的回调。参数中label是用来检测输出的。

element(selector, label).{method}()

返回指定jquery选择器选择的元素的方法。方法可以是如下的jquery方法：val, text, html, height, innerHeight, outerHeight, width, innerWidth, outerWidth, position, scrollLeft, scrollTop, offset.参数中label是用来检测输出的。

element(selector, label).{method}(value)

执行指定jquery选择器选择的元素的方法。方法可以是如下的jquery方法：val, text, html, height, innerHeight, outerHeight, width, innerWidth, outerWidth, position, scrollLeft, scrollTop, offset.参数中label是用来检测输出的。

element(selector, label).{method}(key)

返回执行指定jquery选择器选择的元素执行方法的结果，传递key做参数。方法可以是：attr, prop, css。参数中label是用来检测输出的。

element(selector, label).{method}(key, value)

返回执行指定jquery选择器选择的元素执行方法的结果，传递key和value做参数。方法可以是：attr, prop, css。参数中label是用来检测输出的。

Javascript是动态类型的语言，它在表达式上很强大，但也使得编译器对它没有任何帮助。因此，我们十分强烈地觉得它需要强大的测试工具。我们已经建立这个强大的工具，所以，没有利用不用它了。

← 上一篇 下一篇 →

AngularJS开发指南20：模板

发表于 2012年10月31日 rainer_H

AngularJS模板是一种声明式的规则。它包含了模型和控制器的信息，最后会被渲染成用户在浏览器中看到的视图。它是静态的DOM，包含HTML，CSS和AngularJS指定的元素和属性。AngularJS元素和属性让angular给模板DOM添加行为，或者变形，成为动态地DOM。

下面是你能在模板中用到的AngularJS元素和属性：

- 指令 — 一个用来扩张已存在的DOM元素或者表现可重用DOM组件的属性或者元素。也可称为widget。
- 混合（Markup） — 双花括号是angular内置的一种混合，它会将表达式绑定到元素。
- 过滤器 — 格式化输出给用户的数据。
- 表单控制 — 让你能验证用户输入

注意：除了在模板中声明元素，你也可以在代码中获取到这些元素。

下面的例子展示了一个简单的模板。它包含标准的HTML标记、AngularJS指令和用双花括号进行绑定的表达式。

```
<html ng-app>
 <!-- Body tag augmented with ngController directive  -->
 <body ng-controller="MyController">
   <input ng-model="foo" value="bar">
   <!-- Button tag with ng-click directive, and
          string expression 'buttonText'
          wrapped in "{{ }}" markup -->
   <button ng-click="changeFoo()">{{buttonText}}</button>
   <script src="angular.js">
 </body>
</html>
```

在一个简单的单页应用中，模板由HTML，CSS和包含在一个HTML页面（通常是index.html）中的AngularJS指令组成。在更复杂的应用中，你可以通过“**局部模板**”来在一个页面中显示多个视图，这个局部模板是指定义在单独HTML文件中的模板片段。你在主页面中通过结合使用$route服务和ngView指令来导入这些局部模板。AngularJS入门中第7步和第8步展示了如何使用。

相关主题

- Angular过滤器
- Angular表单

相关API

- API Reference

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南21：使用CSS

发表于 2012年11月2日 rainer_H

AngularJS设置了下面这些CSS类名，你可以方便地用来给你的应用添加样式。

AngularJS使用的CSS类名

- ng-invalid, ng-valid
  - 当元素中的输入值没有通过验证时，AngularJS会将这个类名加入到元素上。（参考input指令）。
- ng-pristine, ng-dirty
  - input指令会给新的widget中的input元素（没有用户交互过）添加ng-pristine类名，交互之后会改为`ng-dirty`。

相关主题

- Angular过滤器
- Angular表单

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

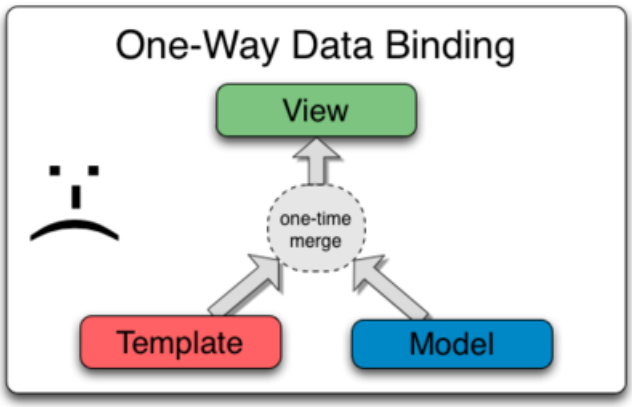
← 上一篇 下一篇 →

AngularJS开发指南22：AngularJS中的数据绑定

发表于 2012年11月3日 rainer_H

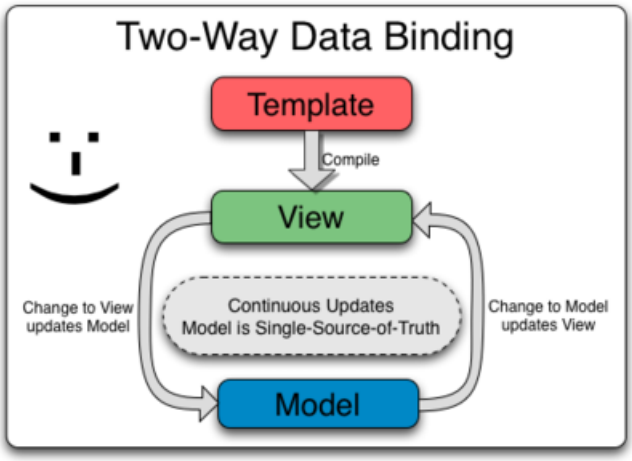
AngularJS中的数据绑定就是模型与视图间的自动同步。这种实现方式让你能专心地处理你的模型。视图总是模型的投影。当模型改变，视图就会反映这种改变，反之亦然。

传统的模板系统数据绑定



大部分模板系统都是这种绑定方式。
方向：如图所示，它们将模型和模板结合生成视图。这个结合过程产生的视图不是动态更新的。更糟的是，任何用户和视图的交互都不会反映到模型。这意味着开发者要自动写视图和模型双向的同步代码。

AngularJS模板中的数据绑定



如图，AngularJS模板的工作原理不同。不同之处在于：第一，模板（附加了自定义属性等标记的未经编译的HTML）是由浏览器编译的；第二，编译最后产生的是一个动态的视图。这里动态指的是视图的任何变化都会直接反应到模型，反之亦然。这使得模型总是应用状态的唯一标识，这大大简化了开发人员的编程工作。你可以简单地认为视图只是模型的投影。

因为视图只是模型的投影，控制器完全和视图分离了，并且视图对它来说是透明的。这使得测试变得更简单，因为你不需要关心相关的DOM或者浏览器变化了。

相关主题

AngularJS作用域
AngularJS模板

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南23：理解AngularJS过滤器

发表于 2012年11月5日 rainer_H

AngularJS过滤器用来格式化输出给用户的数据。除了格式化数据，过滤器还能修改DOM。这使得过滤器通常用来做些如“适时地给输出加入CSS样式”等工作。

比如，你可能有些数据在输出之前需要根据进行本地化。你可以向下面这样使用链式的过滤器来传递表达式：

```
name | uppercase
```

这个表达式执行时会将name的值传递给uppercase过滤器。

相关主题

- 使用AngularJS过滤器
- 创建AngularJS过滤器

相关API

- Angular Filter API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南24：创建AngularJS过滤器

发表于 2012年11月6日 rainer_H

写一个你自己的过滤器非常容易：在你的模块中注册一个新的过滤器（可注入的）工厂函数就行了。这个工厂函数必须放回一个新的过滤器函数，这个过滤函数的第一个参数接受的是输入。任何过滤器参数都会被当成附加的参数传递给过滤器。

下面的例子展示了逆转字符串文本。另外，它有条件地将文本大写并填上颜色。

index.html:

```
<!doctype html>
<html ng-app="MyReverseModule">
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl">
      <input ng-model="greeting" type="greeting"><br>
      No filter: {{greeting}}<br>
      Reverse: {{greeting|reverse}}<br>
      Reverse + uppercase: {{greeting|reverse:true}}<br>
    </div>
  </body>
</html>
```

script.js:

```
angular.module('MyReverseModule', []).
  filter('reverse', function() {
    return function(input, uppercase) {
      var out = "";
      for (var i = 0; i < input.length; i++) {
        out = input.charAt(i) + out;
      }
      // conditional based on optional argument
      if (uppercase) {
        out = out.toUpperCase();
      }
      return out;
    }
  });

function Ctrl($scope) {
  $scope.greeting = 'hello';
}
```

End to end test:

```
it('should reverse greeting', function() {
  expect(binding('greeting|reverse')).toEqual('olleh');
  input('greeting').enter('ABC');
  expect(binding('greeting|reverse')).toEqual('CBA');
});
```

Demo

相关主题

- 理解AngularJS过滤器
- AngularJS HTML编译器

相关API

- Angular Filter API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南25：使用AngularJS过滤器

发表于 2012年11月8日 rainer_H

过滤器可用在任何api或者ng.$rootScoe.Scope的执行过程中，不过一般用来格式化绑定在模板中的表达式。

```
{{ expression | filter }}
```

过滤器一般在处理过程中将数据转变成新的格式。它能使用链式风格，还能接受附加参数。

你可以像下面这样使用链式风格：

```
{{ expression | filter1 | filter2 }}
```

你也可以使用“:”来传递额外的参数给过滤器，比如，将数字123格式化成带有2为小数的形式：

```
123 | number:2
```

下面有些例子，展示了使用不同过滤器格式化之前和之后的样子：

- 无过滤器: {{1234.5678}} => 1234.5678
- 数字过滤器: {{1234.5678|number}} => 1,234.57. 注意“,”号和四舍五入后的后两位。
- 带参数的过滤器: {{1234.5678|number:5}} => 1,234.56780. 过滤器可以接受额外的参数，参数写在“:”的后面。比如，`number`过滤器接受数值型参数来制定需要展示几位小数。

相关主题

- 理解AngularJS过滤器
- 创建AngularJS过滤器

相关API

- Angular Filter API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南26：AngularJS服务

发表于 2012年11月10日 rainer_H

服务是一种由服务器端带到客户端的特性，它由来已久。AngularJS应用中的服务是一些用依赖注入捆绑在一起的可替换的对象。服务是最常和依赖注入一起用的，它也是AngularJS中的关键特性。

相关主题

- 理解AngularJS服务
- 创建AngularJS服务
- 管理服务依赖关系
- 将服务注入到控制器中
- 测试AngularJS服务

相关API

- Angular Service API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南27：使用$location

发表于 2012年11月12日 rainer_H

它是干什么的?

$location服务解析地址栏中的URL（基于window.location），让你在应用代码中能获取到。改变地址栏中的URL会反应$location服务中，反之亦然。

$location服务:

- 暴露当前地址栏的URL，这样你就能
  - 获取并监听URL。
  - 改变URL。
- 当出现以下情况时同步URL
  - 改变地址栏
  - 点击了后退按钮（或者点击了历史链接）
  - 点击了一个链接
- 一系列方法来获取URL对象的具体内容用（protocol, host, port, path, search, hash）.

Comparing $location to window.location

	window.location	$location.service
目的	允许对当前浏览器位置的读写	同左
API	暴露一个“裸聊”的能被读写的对象	暴露jquery风格的读写器
是否在AngularJS应用生命周期中和应用整合	否	可获取到应用声明周期内的每一个阶段，并且和$watch整合
是否和HTML5 API的无缝整合	否	是（对低级浏览器优雅降级）
和应用的上下文是否相关	否，window.location.path返回"/docroot/actual/path"	是，$location.path()返回"/actual/path"

什么时候该用$location

在你想对URL的改变做出响应是，或者在你想改变当前URL时。

它不能用来干什么

在URL改变时，不要刷新整个页面。一定要的话，用低级的API，`$window.location.href`。

API的总览

$location服务的具体行为取决于它初始化时的配置。默认设置对大多数应用都是适合的，你也可以自定义配置来增加些新特性。

$location服务初始化好以后，你就可以使用jquery风格的读写器和它交互了，你可以获取或者改变当前URL。

$location服务的配置

要配置$location服务，检索$locationProvider并把参数设置成以下这样：

- html5Mode(模式): {boolean}
  true - 参阅HTML5模式

  false - 参阅Hashbang模式

  default: false
-

  hashPrefix(前缀): {string}

  Hashbang URLs的前缀 (在Hashbang模式中或者低级浏览器中使用)

default: '!'

配置示例

```
$locationProvider.html5Mode(true).hashPrefix('!');
```

读写器(getter and setter)

$location服务为URL只读部分(absUrl, protocol, host, port)提供读方法，为可读写部分（url, path, search, hash）提供读写方法：

```
// get the current path
$location.path();

// change the path
$location.path('/newValue')
```

所有的写方法返回同一个$location对象来支持链式风格。比如，要在一条语句中改变URL的多个部分：

```
$location.path('/newValue').search({key: value});
```

$location服务有一个特殊的`replace`方法可以用来告诉$lacation服务下一次自动和浏览器同步，上一条浏览记录应该被替换而不是创建一个新的。这在重定向的时候很好用。不这样的话容易使后退按钮失效（点后退时会又触发重定向）。要改变URL而不添加新的历史记录，你可以这样做：

```
$location.path('/someNewPath');
$location.replace();
// or you can chain these as: $location.path('/someNewPath').replace();
```

注意写方法并不会马上更新window.location，而是在作用域的$digest阶段将多个$location操作合并成一个对windiow.location对象的`commit`操作。因为多个操作会后对浏览器来说都会只是一个，所以只要调用一次`replace()`方法就能实现浏览器记录的替换操作。一旦浏览器更新了，$location服务就会将replace方法的标志重置，以后的改变就会创建新的历史记录，知道再次调用replace方法。

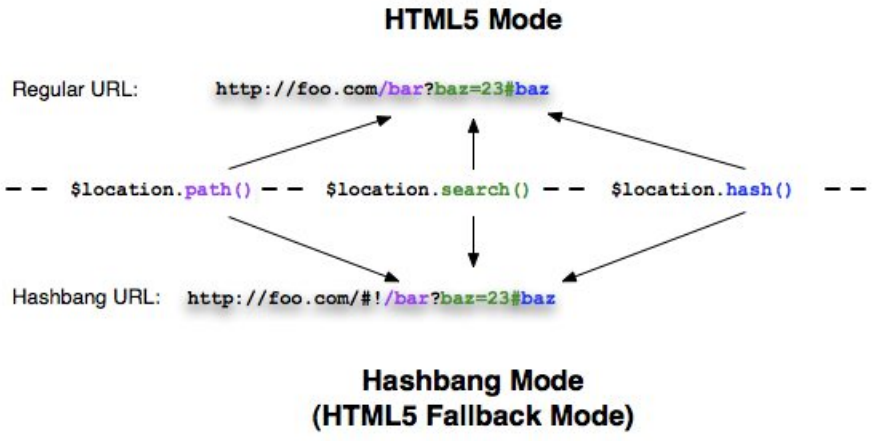
写方法和字符编码

你可以给$location服务传递特殊字符，它会根据RFC 3986规则来编码。当你调用写方法时：

- 所有传递给写方法（如path(), search(), hash()）的值都会被编码。
- 读方法（path(), search(), hash()不带参数的调用）返回解码后的值。
- 当你调用absUrl()时，会返回各部分经过了编码的完整url。
- 当你调用url()时，返回的值是path, search 和hash，形式是`/path?search=a&b=c#hash`。

Hashbang和HTML5模式

$location服务有两种用来控制地址栏URL格式的配置：Hashbang模式（默认）和HTML5模式（使用HTML5历史API）。应用会使用两种模式中相同的API，并且$location服务会使用需要的URL片段和浏览器API来帮助改变URL或者进行历史管理。



	Hashbang模式	HTML5模式
配置	默认	{ html5Mode: true }

URL格式	所有浏览器都支持hashbang URLs	在高级浏览器中使用regular URLs，低级浏览器使用hashbang URLs
<a href=""> 链接重写	否	是
需要服务器端配置	否	是

Hashbang模式(默认mode)

使用这个模式的话，$location会在所有浏览器中使用Hashbang URLs。

示例

```
it('should show example', inject(
  function($locationProvider) {
    $locationProvider.html5mode = false;
    $locationProvider.hashPrefix = '!';
  },
  function($location) {
    // open http://host.com/base/index.html#!/a
    $location.absUrl() == 'http://host.com/base/index.html#!/a'
    $location.path() == '/a'

    $location.path('/foo')
    $location.absUrl() == 'http://host.com/base/index.html#!/foo'

    $location.search() == {}
    $location.search({a: 'b', c: true});
    $location.absUrl() == 'http://host.com/base/index.html#!/foo?a=b&c'

    $location.path('/new').search('x=y');
    $location.absUrl() == 'http://host.com/base/index.html#!/new?x=y'
  }
));
```

支持网络爬虫

你需要添加特别的meta标记在你的文档的头部才能支持对你的AJAX应用的索引。

```
<meta name="fragment" content="!" />
```

这能让网络爬虫请求带有`_escaped_fragment_`形式的参数链接，这样你就能识别爬虫并且返回一个HTML的快照了。更多信息请参考 **Making AJAX Applications Crawlable**。

HTML5模式

在HTML5模式中，$location服务的读写器和浏览器的URL地址通过HTML5历史API交互，这使你能用**regular URL path**并且搜索各组成部分，和hashbang是等效的。 如果浏览器不支持HTML5历史API， $location服务会自动回退成使用hashbang URLs。你就不用担心浏览器的支持性了。$location服务总是会用最好的选择。

- 在低级浏览器中使用了regular URL -> 重定向成hashbang URL
- 在现代浏览器中打开了一个hashbang URL -> 重写成regular URL

example

```
it('should show example', inject(
  function($locationProvider) {
    $locationProvider.html5mode = true;
    $locationProvider.hashPrefix = '!';
  },
  function($location) {
    // in browser with HTML5 history support:
    // open http://host.com/#!/a -> rewrite to http://host.com/a
    // (replacing the http://host.com/#!/a history record)
    $location.path() == '/a'

    $location.path('/foo');
    $location.absUrl() == 'http://host.com/foo'

    $location.search() == {}
    $location.search({a: 'b', c: true});
    $location.absUrl() == 'http://host.com/foo?a=b&c'

    $location.path('/new').search('x=y');
    $location.url() == 'new?x=y'
    $location.absUrl() == 'http://host.com/new?x=y'

    // in browser without html5 history support:
    // open http://host.com/new?x=y -> redirect to http://host.com/#!/new?x=y
```

```
    // (again replacing the http://host.com/new?x=y history item)
    $location.path() == '/new'
    $location.search() == {x: 'y'}

    $location.path('/foo/bar');
    $location.path() == '/foo/bar'
    $location.url() == '/foo/bar?x=y'
    $location.absUrl() == 'http://host.com/#!/foo/bar?x=y'
  }
));
```

低级浏览器使用的降级

在支持HTML5 历史 API的浏览器中，$location服务的读写器和浏览器的URL地址通过HTML5历史API交互。 如果浏览器不支持HTML5 历史API， $location服务会自动降级成使用hashbang URLs。你就不用担心浏览器的支持性了。$location服务总是会用最好的选择。

Html链接重写

当你使用历史API模式时，在不同的浏览器中你需要使用不同的链接，但是你需要做的仅仅是指定好regular URL形式的链接，如 `<a href="/some?foo=bar">link</a>`。

当用户点击这个链接时

- 在低级浏览器中，URL转换成 `/index.html#!/some?foo=bar`
- 在现代浏览器中转换成 `/some?foo=bar`

如果是下面的这集中形式，连接不会被重写。取而代之的是，浏览器会根据链接重新加载页面。

- 包含target的链接
  Example: `link`
-

指向其他域的绝对路径
Example: `link`

- 当base被定义时，使用'/'开头指向一个不同的base路径。
  Example: `link`

服务器端

使用这种模式需要开启服务器端的URL重写功能，基本上你需要重写所有指向你应用的链接(如index.html)。

相对链接

记住要检查所有的相对连接、图片、脚本等。你必须指定你主页面的base url(`<base href="/my-base">`)，或者你使用绝对路径也行，因为相对路径会结合文档的初始绝对路径转换成绝对路径。文档初始路径通常和应用的根路径不一样。

我们强烈推荐应用使用文档根节点开始的历史API，因为它能处理好所有相对路径的问题。

不同浏览器中的链接

因为HTML模式的重写能力，你的用户能在低级浏览器中使用regualr url，在现代浏览器中使用hashbang url。

- 在高级浏览器中会将hashbang URLs冲写成regular URLs。
- 在低级浏览器中使用了regular URL会被重定向成hashbang URL

例子

这里你会看到两个$location实例，都是在Html5模式下，但是在不同浏览器中，这样你就能看出区别了。这两个$location服务是连接在虚拟的浏览器上的。每个input表示了一个浏览器地址栏。

注意，当你输入hashbang url到第一个浏览器的时候（或者反过来）,它不会马上重写成regular URL的形式（或者反过来），这个转换只发生在页面加载对初始URL解析的时候。

例子中我们使用 `<base href="/base/index.html" />`

Source

index.html:

```
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-non-bindable class="html5-hashbang-example">
      <div id="html5-mode" ng-controller="Html5Cntl">
        <h4>Browser with History API</h4>
```

```
        <div ng-address-bar browser="html5"></div><br><br>
        $location.protocol() = {{$location.protocol()}}<br>
        $location.host() = {{$location.host()}}<br>
        $location.port() = {{$location.port()}}<br>
        $location.path() = {{$location.path()}}<br>
        $location.search() = {{$location.search()}}<br>
        $location.hash() = {{$location.hash()}}<br>
        <a href="http://www.host.com/base/first?a=b">/base/first?a=b</a> |
        <a href="http://www.host.com/base/sec/ond?flag#hash">sec/ond?flag#hash</a> |
        <a href="/other-base/another?search">external</a>
      </div>

      <div id="hashbang-mode" ng-controller="HashbangCntl">
        <h4>Browser without History API</h4>
        <div ng-address-bar browser="hashbang"></div><br><br>
        $location.protocol() = {{$location.protocol()}}<br>
        $location.host() = {{$location.host()}}<br>
        $location.port() = {{$location.port()}}<br>
        $location.path() = {{$location.path()}}<br>
        $location.search() = {{$location.search()}}<br>
        $location.hash() = {{$location.hash()}}<br>
        <a href="http://www.host.com/base/first?a=b">/base/first?a=b</a> |
        <a href="http://www.host.com/base/sec/ond?flag#hash">sec/ond?flag#hash</a> |
        <a href="/other-base/another?search">external</a>
      </div>
    </div>
  </body>
</html>
```

script.js:

```
function FakeBrowser(initUrl, baseHref) {
  this.onUrlChange = function(fn) {
    this.urlChange = fn;
  };

  this.url = function() {
    return initUrl;
  };

  this.defer = function(fn, delay) {
    setTimeout(function() { fn(); }, delay || 0);
  };

  this.baseHref = function() {
    return baseHref;
  };

  this.notifyWhenOutstandingRequests = angular.noop;
}

var browsers = {
  html5: new FakeBrowser('http://www.host.com/base/path?a=b#h', '/base/index.html'),
  hashbang: new FakeBrowser('http://www.host.com/base/index.html#!/path?a=b#h', '/base/index.h
};

function Html5Cntl($scope, $location) {
  $scope.$location = $location;
}

function HashbangCntl($scope, $location) {
  $scope.$location = $location;
}

function initEnv(name) {
  var root = angular.element(document.getElementById(name + '-mode'));
  angular.bootstrap(root, [function($compileProvider, $locationProvider, $provide){
    $locationProvider.html5Mode(true).hashPrefix('!');

    $provide.value('$browser', browsers[name]);
    $provide.value('$document', root);
    $provide.value('$sniffer', {history: name == 'html5'});

    $compileProvider.directive('ngAddressBar', function() {
      return function(scope, elm, attrs) {
        var browser = browsers[attrs.browser],
            input = angular.element('<input type="text">').val(browser.url()),
            delay;

        input.bind('keypress keyup keydown', function() {
          if (!delay) {
            delay = setTimeout(fireUrlChange, 250);
          }
        });

        browser.url = function(url) {
```

```
          return input.val(url);
        };

        elm.append('Address: ').append(input);

        function fireUrlChange() {
          delay = null;
          browser.urlChange(input.val());
        }
      };
    });
  }]);
  root.bind('click', function(e) {
    e.stopPropagation();
  });
}

initEnv('html5');
initEnv('hashbang');
```

Demo

注意

页面的重新加载

$location服务职能让你改变URL；不能让你重新加载页面。但你需要重新加载页面或者跳转到另外的页面时，请使用更低级别的API，$window.location.href。

在作用域生命周期外使用$location

$location知道应用作用域的声明周期。但URL改变时，它会更新$location，并且调用$apply，这样所有的监听它的程序都会收到。当你在$digest阶段改变URL，那么没什么问题。$location会将改变传递给浏览器，并且通知所有的监听者。但是如应用之外使用$location的话（比如，在DOM事件中或者测试中），你就要手动调用它$apply来传递改变。

$location.path() 和 "!" "/" 前缀

一个路径应该总是以斜杠开始；$location.path()写方法会在没有前缀/时自动添加。

注意，hashbang模式中的"!"前缀实际上不是$location.path()的一部分，它其实是hashPrefix。

使用$location服务测试

当你在测试中使用$location服务时，你是处在作用域生命周期之外的，所以你要手动调用scope.$apply().

```
describe('serviceUnderTest', function() {
  beforeEach(module(function($provide) {
    $provide.factory('serviceUnderTest', function($location){
      // whatever it does...
    });
  });

  it('should...', inject(function($location, $rootScope, serviceUnderTest) {
    $location.path('/new/path');
    $rootScope.$apply();

    // test whatever the service should do...

  }));
});
```

和之前的AngularJS版本整合

在之前版本中，$location使用hashPath或者hashSearch来处理path和搜索。在这些版本中，$location服务处理path和搜索方法，然后在需要时用它收集到的信息将hashbang URL(如http://server.com/#!/path?search=a)暴露出来。

将你的代码修改为

Navigation inside the app	Change to
$location.href = value $location.hash = value $location.update(value) $location.updateHash(value)	$location.path(path).search(search)
$location.hashPath = path	$location.path(path)

$location.hashSearch = search	$location.search(search)
Navigation outside the app	Use lower level API
$location.href = value $location.update(value)	$window.location.href = value
$location[protocol | host | port | path | search]	$window.location[protocol | host | port | path | search]
Read access	Change to
$location.hashPath	$location.path()
$location.hashSearch	$location.search()
$location.href $location.protocol $location.host $location.port $location.hash	$location.absUrl() $location.protocol() $location.host() $location.port() $location.path() + $location.search()
$location.path $location.search	$window.location.path $window.location.search

$location的双向绑定

AngularJS的编译器目前不支持对$location对象的双向绑定（参看问题列表）。如果你需要对$location对象（在input元素上使用ngModel指令）进行双向绑定，你需要指定一个带有两个监听者的额外的模型属性（比如locationPath），这两个监听者各负责一个方向。

```
<!-- html -->
<input type="text" ng-model="locationPath" />
// js - controller
$scope.$watch('locationPath', function(path) {
  $location.path(path);
});

$scope.$watch('$location.path()', function(path) {
  scope.locationPath = path;
});
```

相关API

$location API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南28：创建服务

发表于 2012年11月13日 rainer_H

虽然AngularJS提过了很多有用的服务，但是如果你要创建一个很棒的应用，你可能还是要写自己的服务。你可以通过在模块中注册一个服务工场函数，或者通过Modeul#factory api或者直接通过模块配置函数中的$provide api来实现。

所有的服务都符合依赖注入的原则。它们用一个唯一的名字将自己注册进AngularJS的依赖注入系统（injector），并且声明需要提供给工场函数的依赖。它们的依赖在测试中可以是虚拟的，这使得它们能很好地被测试。

注册服务

要注册服务，你首先要有一个包含该服务的模块。然后你就能通过模块的api或者使用模块配置函数中的$provide服务来注册你的服务了。下面的伪代码显示了这两种方法。

使用angular.Module api:

```
var myModule = angular.module('myModule', []);
myModule.factory('serviceId', function() {
  var shinyNewServiceInstance;
  //factory function body that constructs shinyNewServiceInstance
  return shinyNewServiceInstance;
});
```

使用$provide服务:

```
angular.module('myModule', [], function($provide) {
  $provide.factory('serviceId', function() {
    var shinyNewServiceInstance;
    //factory function body that constructs shinyNewServiceInstance
    return shinyNewServiceInstance;
  });
});
```

注意，你不应该注册一个服务实例，而是一个会在被调用时创建实例的工场函数。

依赖

服务不仅可以被依赖，还可以有自己的依赖。依赖可以在工场函数的参数中指定。参阅AngularJS的依赖注入系统，和使用依赖的数组表示法和$inject属性来让依赖表示精简化。

下面是一个很简单的服务的例子。这个服务依赖于$window服务（会被当成参数传递给工场函数），并且只是个函数。这个服务的任务是存储所有的通知；在第三个通知以后，服务会用window的alert来输出所有的通知。

```
angular.module('myModule', [], function($provide) {
  $provide.factory('notify', ['$window', function(win) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        win.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  }]);
});
```

实例化AngularJS的服务

所有服务都是延迟实例化的。这意味着所有的服务只有在需要时，或者被依赖时才会实例化。换句话说，AngularJS不会实例化服务，除非被要请求了或者被应用直接或间接依赖了。

作为单例的服务

最好，要注意的是所有AngularJS服务都是单例的。这意味着在每一个注入器中都只有一个需要的服务的实例。因为AngularJS极度讨厌全局的东西，所以是可以创建多个注入器的，并且每个住一起有自己的服务实例。但这种情况很少，除非在测试中，这样的特性才极度重要。

相关主题

- 理解AngularJS服务
- 管理服务以来
- 将服务注入控制器
- 测试AngularJS服务

相关API

- Angular Service API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南29：将服务注入到控制器中

发表于 2012年11月14日 rainer_H

将服务用作控制器的依赖和将服务用作其他服务的依赖很类似。

因为Javascript是一种动态语言，依赖注入系统无法通过静态类型来知道应该注入什么样的服务（静态类型语言就可以）。所以，你应该$inject的属性来指定服务的名字，这个属性是一个包含这需要注入的服务的名字字符串的数组。名字要和服务注册到系统时的名字匹配。服务的名称的顺序也很重要：当执行工场函数时传递的参数是依照数组里的顺序的。但是工场函数中参数的名字不重要，但最好还是和服务本身的名字一样，下面展示了这样的好处:

```
function myController($loc, $log) {
this.firstMethod = function() {
 // use $location service
 $loc.setHash();
};
this.secondMethod = function() {
 // use $log service
 $log.info('...');
};
}
// which services to inject ?
myController.$inject = ['$location', '$log'];
```

index.html:

```
<!doctype html>
<html ng-app="MyServiceModule">
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="myController">
      <p>Let's try this simple notify service, injected into the controller...</p>
      <input ng-init="message='test'" ng-model="message" >
      <button ng-click="callNotify(message);">NOTIFY</button>
    </div>
  </body>
</html>
```

script.js:

```
angular.
 module('MyServiceModule', []).
 factory('notify', ['$window', function(win) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        win.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  }]);

function myController(scope, notifyService) {
  scope.callNotify = function(msg) {
    notifyService(msg);
  };
}

myController.$inject = ['$scope','notify'];
```

end to end test :

```
it('should test service', function() {
    expect(element(':input[ng\\:model="message"]').val()).toEqual('test');
});
```

隐式依赖注入

AngularJS依赖注入系统的新特性使得AngularJS可以通过参数名称来判断依赖。让们重写上面的例子，展示一下隐式地依赖$window, $scope：

index.html:

```
<!doctype html>
<html ng-app="MyServiceModuleDI">
  <head>
    <script src="http://code.angularjs.org/angular-1.0.2.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="myController">
      <p>Let's try the notify service, that is implicitly injected into the controller...</p>
      <input ng-init="message='test'" ng-model="message">
      <button ng-click="callNotify(message);">NOTIFY</button>
    </div>
  </body>
</html>
```

script.js:

```
angular.
 module('MyServiceModuleDI', []).
 factory('notify', function($window) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        $window.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  });

function myController($scope, notify) {
  $scope.callNotify = function(msg) {
    notify(msg);
  };
```

但是如你要压缩你的代码，你的变量名会被重命名，你就只能显示地指定依赖了。

相关主题

- 理解AngularJS服务
- 创建AngularJS服务
- 管理服务依赖
- 测试AngularJS服务

相关API

Angular Service API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

← 上一篇 下一篇 →

AngularJS开发指南30：管理服务依赖

发表于 2012年11月18日 rainer_H

AngularJS允许服务声明它的实例和构造依赖的服务。

要声明依赖，你可以在工场方法参数中隐式指明他们，也可以将$inject属性设置成包含了依赖名称的数组，或者是使用数组表示法。不推荐使用$inject属性的这种方法。

使用数组表示法:

```
function myModuleCfgFn($provide) {
  $provide.factory('myService', ['dep1', 'dep2', function(dep1, dep2) {}]);
}
```

使用$inject属性:

```
function myModuleCfgFn($provide) {
  var myServiceFactory = function(dep1, dep2) {};
  myServiceFactory.$inject = ['dep1', 'dep2'];
  $provide.factory('myService', myServiceFactory);
}
```

使用隐式依赖(使用代码压缩是会失效):

```
function myModuleCfgFn($provide) {
  $provide.factory('myService', function(dep1, dep2) {});
}
```

下面是一个互相依赖的两个服务的例子。它们也依赖了其他AngularJS提供的服务。

```
/**
 * batchLog service allows for messages to be queued in memory and flushed
 * to the console.log every 50 seconds.
 *
 * @param {*} message Message to be logged.
 */
  function batchLogModule($provide){
    $provide.factory('batchLog', ['$timeout', '$log', function($timeout, $log) {
      var messageQueue = [];

      function log() {
        if (messageQueue.length) {
          $log('batchLog messages: ', messageQueue);
          messageQueue = [];
        }
        $timeout(log, 50000);
      }

      // start periodic checking
      log();

      return function(message) {
        messageQueue.push(message);
      }
    }]);

    /**
     * routeTemplateMonitor monitors each $route change and logs the current
     * template via the batchLog service.
     */
    $provide.factory('routeTemplateMonitor',
                ['$route', 'batchLog', '$rootScope',
         function($route,   batchLog,   $rootScope) {
      $rootScope.$on('$routeChangeSuccess', function() {
        batchLog($route.current ? $route.current.template : null);
      });
    }]);
  }

  // get the main service to kick of the application
  angular.injector([batchLogModule]).get('routeTemplateMonitor');
```

上例中有几点要注意：

- batchLog服务依赖内建的$timeout和$log服务，并且允许消息批量地使用console.log记录。
- 和batchLog服务一样，routeTemplateMonitor服务依赖内建的$route服务。
- 自定义的服务都使用隐式表示和数组法来表示自己的依赖。最重要的是数组中的服务的名字顺序要和工厂函数参数的名字顺序对应。除非依赖是隐式地通过函数参数名表示的，那么就是有声明依赖的数组名称顺序决定依赖注入的顺序。

相关主题

- 理解AngularJS服务
- 创建AngularJS服务
- 管理服务依赖
- 测试AngularJS服务

相关API

Angular Service API
Angular Injector API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南31：测试服务

发表于 2012年11月20日 rainer_H

下面是“创建AngularJS服务”一章中“依赖”例子里的“notify”服务的单元测试。测试用使用了Jasmine spy替代了真实浏览器的alert。

```
var mock, notify;

beforeEach(function() {
  mock = {alert: jasmine.createSpy()};

  module(function($provide) {
    $provide.value('$window', mock);
  });

  inject(function($injector) {
    notify = $injector.get('notify');
  });
});

it('should not alert first two notifications', function() {
  notify('one');
  notify('two');

  expect(mock.alert).not.toHaveBeenCalled();
});

it('should alert all after third notification', function() {
  notify('one');
  notify('two');
  notify('three');

  expect(mock.alert).toHaveBeenCalledWith("one\ntwo\nthree");
});

it('should clear messages after alert', function() {
  notify('one');
  notify('two');
  notify('third');
  notify('more');
  notify('two');
  notify('third');

  expect(mock.alert.callCount).toEqual(2);
  expect(mock.alert.mostRecentCall.args).toEqual(["more\ntwo\nthird"]);
```

相关主题

- 理解AngularJS服务
- 创建AngularJS服务
- 管理服务依赖
- 将AngularJS注入到控制器

相关API

Angular Service API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南32：理解服务

发表于 2012年11月22日 rainer_H

AngularJS服务是一种能执行一个常见操作的单例，比如$http服务是用来操作浏览器的XMLHttpRequest对象的。

要使用AngularJS服务，你只需要在需要的地方（控制器，或其他服务）指出依赖就行了。AngularJS的依赖注入系统会帮你完成剩下的事情。它负责实例化，查找左右依赖，并且按照工场函数要求的样子传递依赖。

AngularJS通过“构造器注入”来注入依赖（通过工场函数来传递依赖）。以为Javascript是动态类型语言，AngularJS无法通过静态类型来识别服务，所以你必须使用$inject属性来显式地指定依赖。比如:

```
myController.$inject = ['$location'];
```

AngularJS web框架提供了一组常用操作的服务。和其他的内建变量或者标识符一样，内建服务名称也总是以"$"开头。另外你也可以创建你自己的服务。

相关主题

- 理解AngularJS服务
- 创建AngularJS服务
- 管理服务依赖
- 测试AngularJS服务

相关API

Angular Service API
Angular Injector API

rainer_H 发表在 开发指南 分类，标签 AngularJS、rainer_H、开发指南。将本文加入收藏夹。

要发表评论，您必须先登录。

← 上一篇 下一篇 →

AngularJS开发指南33：单元测试

发表于 2012年11月23日 rainer_H

Javascript是动态类型语言，它的表达式有强大的威力，但也因此使得编译器对它没有任何帮助。所以我们强烈的觉得它需要强大的测试框架。我们已经建好了这样一个框架，所以没有理由不用它吧。

不要把问题都搅在一起

单元测试，如名字一样就是单独测试每个部分的代码。单元测试视图解决的问题是：我的逻辑正确了吗？排序函数正确排序了吗？为了解决这样的问题我们非常需要将它们鼓励出来。因为当我们解决排序问题的时候，我们不希望还要考虑到其他的如果DOM操作的问题，或者需要执行异步请求来获取数据之类的。但是在通常的项目中，几乎很难去调用一个单独的纯粹的函数。因为开发者常常将问题搅在一起，最后写出的代码职责太多。比如异步读取数据同时又要排序又要操作DOM。在AngularJS中，我们试图让开发者正容易地使用正确的开发方法，所以我们通过依赖注入来让你异步获取数据（你也可以模拟），我们创建了数据抽象让你操作模型数据的时候不用去管DOM。所以，最后你可以写出纯粹的排序函数了，并且能去测试它。测试不需要等待异步请求，也不需要去管理DOM或者验证DOM是否也正确反应。AngularJS是以可测试为宗旨之一写的，但是仍然需要你自己正确地操作。我们试图让事情变得简单，但是如果你不照章办事，你还是会把你的应用搞的完全没办法测。

依赖注入

有几种你可以获取依赖的方式：1.你直接使用new运算符建立实例。2.你查找已知的域，比如全局对象。3.你使用一个已有的注册系统来获取（但你如何来获取注册系统呢，貌似你还是必须查找一个已知域）。4.等待依赖被传递给你。

上面的选项只有最后是可测的。让我们看看为什么：

使用new操作符

new操作符本身来说并没有什么错误，但是待用它的地方会将自己和被调用者永久绑定。比如说，我们现在实例化一个XHR对象来取数据。

```
function MyClass() {
  this.doWork = function() {
    var xhr = new XHR();
    xhr.open(method, url, true);
    xhr.onreadystatechange = function() {...}
    xhr.send();
  }
}
```

问题来了，在测试中，我们很可能想要模拟一个XHR来返回虚拟的数据。但是调用`new XHT()`使得我们被绑定在了这个真实的对象上，别且没有很好地方法能替换它。呃，其实有一些方法，但是都很丑陋并且会导致其他问题，这个已经超出本章范围，暂不讨论。

上面的类很难用于测试，我们不得不使用一些歪招：

```
var oldXHR = XHR;
XHR = function MockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that MockXHR got called with the right arguments
XHR = oldXHR; // if you forget this bad things will happen
```

全局查找

另一种解决上述问题的方法是在一个已知域查找服务

```
function MyClass() {
  this.doWork = function() {
    global.xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
  }
}
```

虽然没有使用new来创建实例，但本质上和new是一样。在代码中还是没有办法为了测试的需要而去拦截global.xhr的调用，除非又使用些怪招。这里面最基本的问题在于我们需要有一个方法来模拟依赖。更多这方面的知识请参阅 Brittle Global State & Singletons ：

上面的类难测试是因为我们需要改变全局状态：

```
var oldXHR = global.xhr;
global.xhr = function mockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.xhr = oldXHR; // if you forget this bad things will happen
```

服务注册

看起来好像我们可以用一个注册系统来解决这个问题，在需要时替换掉不需要测试的部分就行了。

```
function MyClass() {
  var serviceRegistry = ????;
  this.doWork = function() {
    var xhr = serviceRegistry.get('xhr');
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

但是，这个serviceRegistry是哪里来的呢？貌似又要new一个。而且在测试全局的时候，我们无法重置服务。

上面的例子难以测试是因为我们必须改变全局状态：

```
var oldServiceLocator = global.serviceLocator;
global.serviceLocator.set('xhr', function mockXHR() {});
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.serviceLocator = oldServiceLocator; // if you forget this bad things will happen
```

传递依赖

终于说到这个了。

```
function MyClass(xhr) {
  this.doWork = function() {
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

这是最好的一种方式，因为代码不关心xhr从哪里来，谁穿件来的谁负责实例化它。因为它的创建者和用它的地方肯定是不一样的，所以它在逻辑上分离了两者，简言之这就是依赖注入。

上面的例子是很好测的，我们可以这样写：

```
function xhrMock(args) {...}
var myClass = new MyClass(xhrMock);
myClass.doWork();
// assert that xhrMock got called with the right arguments
```

注意我们没有用到任何全局变量。

AngularJS已经内建好依赖注入系统来让事情比那的简单，但是你仍然需要按照规矩来，才能让你的应用更好测试。

控制器

应用的本质区别在于它的逻辑，这也正是我们希望测试的。如果你应用的逻辑里混杂着DOM操作，那就会很难测试，像下面这样：

```
function PasswordController() {
  // get references to DOM elements
  var msg = $('.ex1 span');
  var input = $('.ex1 input');
  var strength;

  this.grade = function() {
```

```
    msg.removeClass(strength);
    var pwd = input.val();
    password.text(pwd);
    if (pwd.length > 8) {
      strength = 'strong';
    } else if (pwd.length > 3) {
      strength = 'medium';
    } else {
      strength = 'weak';
    }
    msg
     .addClass(strength)
     .text(strength);
  }
}
```

上面的代码难测在于你需要同时去测试DOM是否也正确反应了。你的测试可能会像下面这样：

```
var input = $('<input type="text"/>');
var span = $('<span>');
$('body').html('<div class="ex1">')
  .find('div')
    .append(input)
    .append(span);
var pc = new PasswordController();
input.val('abc');
pc.grade();
expect(span.text()).toEqual('weak');
$('body').html('');
```

在AngularJS控制器是和DOM操作完全分离的，这使得我们能像下面这样更好的测试：

```
function PasswordCntrl($scope) {
  $scope.password = '';
  $scope.grade = function() {
    var size = $scope.password.length;
    if (size > 8) {
      $scope.strength = 'strong';
    } else if (size > 3) {
      $scope.strength = 'medium';
    } else {
      $scope.strength = 'weak';
    }
  };
}
```

测试会变得这样直白：

```
var pc = new PasswordController();
pc.password('abc');
pc.grade();
expect(span.strength).toEqual('weak');
```

注意我们的测试不只是变短了，而是更加清楚了。这样的测试代码告诉了你究竟是怎么运行的，而不是一堆看不懂的符号。

过滤器

过滤器是用来将输出给用户的数据变得更可读的。它们的重要性在于它们将格式化的工作从应用逻辑中抽离出来了，进一步的简化了应用逻辑。

```
myModule.filter('length', function() {
  return function(text){
    return (''+(text||'')).length;
  }
});

var length = $filter('length');
expect(length(null)).toEqual(0);
expect(length('abc')).toEqual(3);
```

指令

指令是在当模型数据改变时负责更新DOM的。

要发表评论，您必须先登录。