# angularJS进阶

### jQuery：用更少的代码，实现更强悍的功能

托互联网日新月异发展的福，浏览器变成了人们接入互联网的入口，而JavaScript 这个曾经的小语种，终于成功地站到了舞台的中央，唤起了开发者的兴趣。

浏览器里原生的JavaScript有点像汇编语言，不同的浏览器就像不同的CPU架构， 汇编语言各有千秋，这让前端开发者很恼火。聪明人很快发现了这个痛点，于是， 抹平浏览器差异的jQuery库出现了。



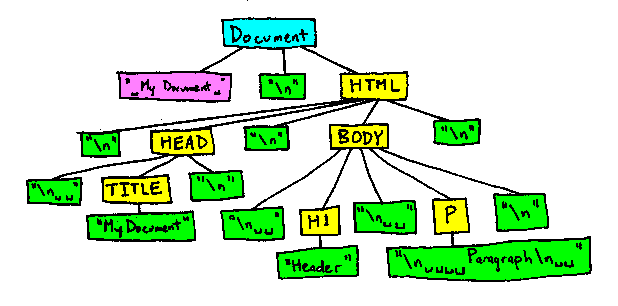
jQuery由一小撮对浏览器极其熟稔的极客负责抹平不同浏览器的差异，其他开发 者只需要基于jQuery进行开发，可以更好地关注业务实现，而不是把时间花在 适配不同的浏览器上。

这样的分工符合经济学原理，开启了一个不可忽视的jQuery时代。

### 满眼的全是DOM

jQuery使得开发无刷新动态页面（AJAX）或者单页应用（SPA）变得 相当简单。

标准的HTML页面是静态的，被浏览器渲染后就产生了一个DOM树：



jQuery让静态的文档动起来，通过提供一系列的选择符，jQuery使开发者能够 极其方便地选中一组DOM节点，对其进行操作。

这就是jQuery的开发范式。jQuery没有引入什么新的概念，只是朴素地，让你能够更简单 地、低成本地操作DOM：

1. 用选择符选定一组DOM节点
2. 操作选中的DOM节点，比如：修改文本、改变属性、挂接事件监听函数、变换DOM等等。
3. 基本不用考虑跨浏览器的兼容性

jQuery的API符合大多数开发者的预期，因此，很容易上手。

### jQuery缺失的环节

jQuery有点像C语言，威力很大，不过要弄出点像样的前端界面，还得花不少功夫 处理琐碎的事情。

还能再简单些吗？Misko Hevery认为在某些应用场景下可以。于是，AngularJS诞生了：



AngularJS引入了三个主要的概念，期望让前端开发更系统化一些：

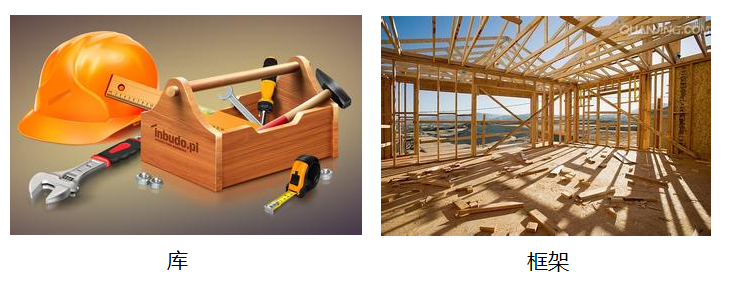
1. **声明式界面开发**
2. **双向数据绑定**
3. **使用依赖注入解耦**

很多人在初次接触AngularJS时，都有些吃惊，因为它把前端开发搞的突然严肃起来 了。考虑到Misko曾经是一个Java程序员，这一切就好理解了。

Java程序员擅长引入复杂的架构来解决简单的问题，对吧？

### 库 vs. 框架

和jQuery不同，AngularJS是一个框架。



jQuery是一个库，库总是被动的，就像工具，应用的开发逻辑是你的，在 某一点上需要用一下工具，就用好了。

框架则非常不同，这意味着AngularJS为应用已经搭起了一个架子，约定了 一些组成部分，并且实现了这些部分的拼装运行。换句话说， 应用的开发逻辑是AngularJS的，你得跟着它走。

所以，AngularJS难学一些，因为它有一个架子在那，你不了解这个架子， 基本没法下手。

### 用jQuery实现一个小时钟

我们试着用jQuery实现一个简单的时钟页面，实现思路很简单：

* **引入jquery库**

jquery库将创建一个全局对象：$。开发者需要的API都挂接在这个对象上。 它其实是一个类工厂，负责将指定的DOM对象转化为jquery对象。

* **在DOM文档就绪后，启动一个定时器**

使用$(document).ready()可以监听DOM文档就绪事件，通常简写成$(...)。

* **在定时器里更新div#clock的文本**

"div#clock"是一个选择符，使用这个选择符，jquery库可以找到那个 特定的DOM对象，并将其转化为jQuery对象。

jquery封装后的DOM对象有一堆的方法供你调用，我们使用text()方法更新其文本。

### 重写示例：模板、指令和视图

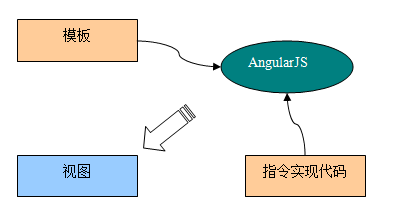
AngularJS最显著的特点是用静态的HTML文档，就可以生成具有动态行为的页面。

还是前面的小时钟示例，我们使用AngularJS模板来重写，示例已经嵌入→\_→：

HTML文件看起来像普通的HTML，只是其中多了一些特别的标记 （比如：ng-app,ez-clock等等）。在Angular中，这个HTML文件被称为模板。

ng-app这样的标记我们称之为指令。模板通过指令指示AngularJS进行必要的操作。 比如：ng-app指令用来通知AngularJS自动引导应用；ez-clock 指令用来通知AngularJS生成指定的时钟组件。

当AngularJS启动应用时，它会通过一个编译器解析处理这个模板文件，生成的结果就是： 视图：



我们定义了两个部件：模板（包含指令的HTML文件）和指令实现 （JavaScript文件），AngularJS将这两部分拼装起来，生成了最终的视图。

有点理解框架的含义了吗？

angular.module("ezstuff",[])//创建模块ezstuff

.directive("ezClock",function(){//在模块上注册指令ezClock的类工厂

return {

restrict : "E",

replace : true,

template : "<div class='clock'></div>",

link : function(scope,element,attrs){

setInterval(function(){

//获取当前时间

var d = new Date();

//element对应引用该指令的DOM对象的jqLite封装

element.text(d.toString());

},1000);

}

}

})

### 使用指令封装JavaScript代码

我们在模板中使用了一个自定义的标签ez-clock，而它变成了一个会动的时钟， 这期间发生了什么事情?

肯定不是浏览器干的，它不认识ez-clock是什么东西。

angular.min.js引入了基本的angularJS库，它会在浏览器载入HTML文档并且 建立好DOM树后，执行以下操作：

1. 找到有ng-app属性的DOM节点
2. 以这个节点为根节点，搜索自定义指令，发现ez-clock
3. 调用ez-clock指令的实现函数（指令类工厂）进行展开

根据我们的定义，ez-clock的展开操作如下：

1. 使用一个div元素替换这个自定义标签
2. 创建一个定时器，在定时器触发时刷新div元素的innerText

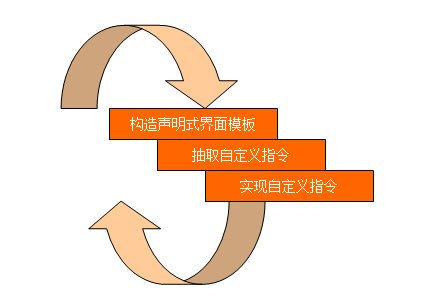
ez-clock这样的非HTML标准标签，在AngularJS中之所以称为指令/directive， 就是指看到它时，基础框架需要对其进行解释，以便展开成浏览器可以理解 的东西（HTML元素和脚本），而这个解释的过程被称为：编译。

可见，AngularJS框架要求将HTML文档和JavaScript代码分割的更清晰，通常混杂在 HTML文档中的JavaScript代码，需要以指令的形式进行封装，而模板、指令 实现代码这两个部件，则由基础框架负责拼装运行。

### 起点：声明化

基于前面的示例，我们容易感受到使用AngularJS进行应用开发的一个重要的思维模式： 从构造声明式界面入手。

事实上，我猜测这也是Misko开发AngularJS最初的动机。稍早一些的Flex、WPF和Firefox 的XUL，或多或少给了Misko启发。

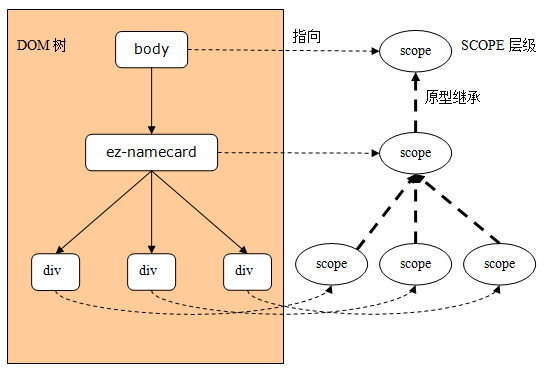


在使用AngularJS进行前端开发时，始终应该从构造声明式界面模板开始，如果现成的指令不够 用，那么就定义自己的指令、实现自己的指令。这是一个迭代的过程。

记住，指令是新型的API，用界面的声明化作为需求，来指导我们的代码封装。

### 层级的作用域

在AngularJS中，ng-app开始的DOM子树上，每个DOM对象都有一个对应的scope对象。 比如，在我们的示例中，body对象对应一个scope对象（因为body元素有ng-app属性，所以 这个scope就是$rootScope对象），ez-namecard对象也对应一个scope对象......



在默认情况下，一个DOM子元素不会创建新的作用域，也就是说，这个子元素所对应的 scope对象，其实就是它的最近一级的祖先对象对应的scope对象。比如，在我们的例子中， ez-namecard对应的scope对象，就是它的父对象即body对象的scope对象，恰好也就是 $rootScope对象；而ez-namecard有3个div子元素对应的scope对象，也是$rootScope对象。

有些指令会导致创建新的作用域，比如ng-controller。如果在一个DOM对象上创建了新 的作用域，那么这个scope对象的原型是其最近一级的组件对象的scope对象。

比如在我们的例子中，如果在ez-namecard上使用ng-controller指令，那么ez-namecard 对应的scope对象就不再是body对应的$rootScope对象，但是由于是原型继承，所以通过 这个scope依然可以访问到sb变量。

### 监听数据的变化

我们已经实现了将数据显示到界面上，不过这还不够。

由于编译仅仅在启动引导时执行一次，这意味着我们的link函数只会被调用一次，那么， 如果数据变化，在界面上将不会有任何反馈，即界面和数据将变得不同步了。

这需要持续监听数据的变化。

好在AngularJS的scope对象可以使用$watch()方法，对建立在其上的变量的变化进行监听：

1. $watch(watchExpression, listener, [objectEquality]);

$watch方法要求传入三个参数：

* watchExpression - 要监听的表达式，比如："sb"
* listener - 变化发生时的回调函数，AngularJS将向这个函数传入新值和旧值
* objectEquality - 如果监听表达式的值是一个对象，应当将这个参数置为true。

→\_→是经过改进后的代码，当数据被改变时，界面会自动得到更新。这时，我们称，建立了从 数据到界面的单向绑定。为了验证这一点，在代码中我们追加了一个定时器自动 更新数据的age值。

你可以试着去掉监听部分代码，看有什么效果。

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body ng-app="ezstuff" ng-init="sb = {name:'somebody',gender:'male',age:28}">

<ez-namecard data="sb"></ez-namecard>

</body>

</html>

angular.module("ezstuff",[])

.directive("ezNamecard",function($rootScope){

return {

restrict : "E",

template : "<div class='namecard'>",

replace : true,

link : function(scope,element,attrs){

element.append("<div>name : <span class='name'></span></div>")

.append("<div>gender : <span field='gender'></span></div>")

.append("<div>age : <span field='age'></span></div>")

//监听sb变量的变化，并在变化时更新DOM

scope.$watch(attrs.data,function(nv,ov){

var fields = element.find("span");

fields[0].textContent = nv.name;

fields[1].textContent = nv.gender;

fields[2].textContent = nv.age;

},true);

//验证代码，1秒改变1次age的值

setInterval(function(){

scope.$apply("sb.age=sb.age+1;")

},1000);

}

};

});

### 如何修改数据

一旦在指令的实现代码中可以访问数据模型，那么使用声明式模板实现数据 修改也非常简单了。

我们定义一个新的指令：ez-namecard-editor，意图让其展开成这样：

1. <ul>
2. <li>name : <input type='text'> </li>
3. <li>gender : <input type='text'> </li>
4. <li>age : <input type='text'></li>
5. </ul>

在ez-namecard-editor的指令实现中，为了用input中的值自动更新 sb变量中的值，我们需要在给input对象挂接上监听函数（示例中使用keyup事件）， 在监听函数中实现对sb变量的修改。

最终的效果是，用户在界面上进行的操作，自动地同步到了我们的数据。这时，我们称， 已经建立了从界面到数据的单向绑定。

→\_→的示例代码中，为了验证绑定的效果，我们增加了一个ez-logger指令。这个指令 将一个DOM元素的内容绑定到指定的变量上。

这样，当我们通过ez-namecard-editor修改数据时，可以同步地看到变化后的内容。

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body ng-app="ezstuff" ng-init="sb = {name:'somebody',gender:'male',age:28}">

<!-- 下面两个指令都绑定到变量sb上-->

<ez-namecard-editor data="sb"></ez-namecard-editor>

<div ez-logger data="sb"></div></body>

</html>

angular.module("ezstuff",[])

.directive("ezNamecardEditor",function(){

return {

restrict : "E",

template : "<ul class='nceditor'></ul>",

replace : true,

link : function(scope,element,attrs){

//获得变量名称

var model = attrs.data;

//展开HTML模板，使用field属性标记对应字段

element.append("<li>name : <input type='text' field='name'></li>")

.append("<li>gender : <input type='text' field='gender'></li>")

.append("<li>age : <input type='text' field='age'></li>");

//监听DOM事件，变化时修改变量值

element.find("input").on("keyup",function(ev){

var field = ev.target.getAttribute("field");

scope[model][field] = ev.target.value;

//将对scope的修改进行传播

scope.$apply("");

});

}

};

})

.directive("ezLogger",function(){

return {

restrict : "A",

link : function(scope,element,attrs){

var model = attrs.data;

scope.$watch(model,function(nv){

var cnt = JSON.stringify(nv,null," ");

element.html("<pre>"+cnt+"</pre ");

},true);

}

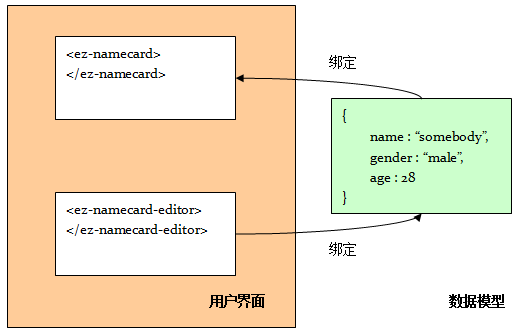
};

});

### 数据变化的传播

数据绑定有两个方向：

* 数据 → 界面：我们使用scope对象的$watch()方法监听数据的变化，来更新界面。
* 界面 → 数据：我们在界面的DOM对象上监听变化事件，来更新数据，并通过$apply()方法传播变化。



上面的图中，我们把ez-namecard和ez-namecard-editor都绑定到同一个sb对象上，那么在 ez-namecard-editor上进行编辑，将导致sb对象发生变化；由于ez-namecard监听了这个变化， 所以，ez-namecard的显示也应该变化。

* **$watch()**

每个scope对象都维护了一个私有的监听队列，每次当我们在scope上执行一次$watch方法，就相当于 向这个监听队列里塞入一个监听函数。

* **$apply()**

为了捕捉对数据的修改，AngularJS要求开发者使用scope对象的$apply方法对数据进行修改， $apply方法内部会自动地调用监听队列里的监听函数，比如：

1. //方法1：直接修改sb对象. 不会自动触发监听函数
2. scope.sb.name = 'Tonny';
4. //方法2：使用scope的$apply方法，在数据修改后会自动触发监听函数
5. scope.$apply("sb.name = 'Tonny'");
7. //方法3：直接修改sb对象，然后调用$apply方法来传播变化。
8. scope.sb.name = 'Tonny';
9. scope.$apply("");

在有些情况下，AngularJS会自动调用$apply方法，比如在初次编译的时候。但无论哪种情况， 希望你能了解，对数据的变化监听，总是需要通过$apply方法的调用而被激活，如果 AngularJS没有获得一个机会来调用$apply，就需要你手工的调用它。

这里面没有魔法。

### 找不到的API？

AngularJS提供了一些功能的封装，但是当你试图通过全局对象angular去 访问这些功能时，却发现与以往遇到的库大不相同。

* **$http**

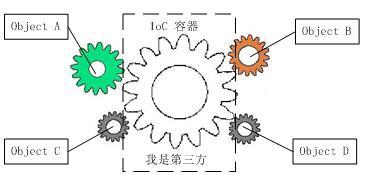
比如，在jQuery中，我们知道它的API通过一个全局对象：$ 暴露出来，当你需要 进行ajax调用时，使用$.ajax()就可以了。这样的API很符合思维的预期。

AngularJS也暴露了一个全局对象：angular，也对ajax调用进行封装提供了一个 $http对象，但是，但是，当你试图沿用旧经验访问angular.$http时，发现不是 那么回事！

仔细地查阅$http的文档，也找不到一点点的线索，从哪里可以把这个$http拿到。

* **依赖注入/DI**

事实上，AngularJS把所有的功能组件都以依赖注入的方式组织起来：



在依赖注入的模式下，所有的组件必须通过容器才能相互访问，这导致了在AngularJS中， 你必须通过一个中介才能获得某个组件的实例对象：

1. var injector = angular.injector(['ng']);
2. injector.invoke(function($http){
3. //do sth. with $http
4. });

这个中介，就是依赖注入模式中的容器，在AngularJS中，被称为：注入器。

在→\_→的示例中，我们可以看到，我们已经拿到了$http对象，它其实是一个函数。

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<!--在这里显示$http的内容-->

<div id="logger"></div>

</body>

</html>

angular.element(document).ready(function(){

angular.injector(["ng"]).invoke(function($http){

//将$http对象转成字符串显示出来

var e = document.querySelector("#logger");

angular.element(e).text($http.toString());

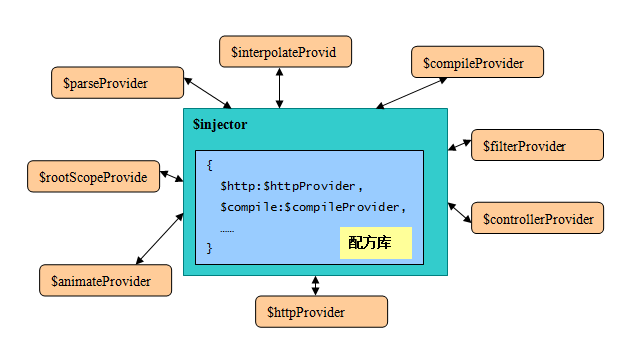
});

});

### 注入器/injector

注入器是AngularJS框架实现和应用开发的关键，这是一个DI/IoC容器的实现。

AngularJS将功能分成了不同类型的组件分别实现，这些组件有一个统称 - 供给者/provider， 下图中列出了AngularJS几个常用的内置服务：



AngularJS的组件之间不可以互相直接调用，一个组件必须通过注入器才 可以调用另一个组件。这样的好处是组件之间相互解耦，对象的整个生命周期的管理 甩给了注入器。

注入器实现了两个重要的功能：

1. 集中存储所有provider的配方

配方其实就是：名称+类构造函数。AngularJS启动时，这些provider首先使用其配方在注入器 内注册。比如，http请求服务组件封装在$httpProvider类内，它通过"$http"这个名字在注入 器内注册。

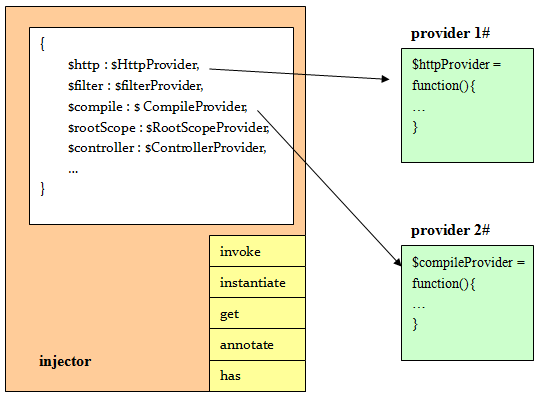
1. 按需提供功能组件的实例

其他组件，比如一个用户的控制器，如果需要使用http功能，使用"$http"这个名字 向注入器请求，就可以获得一个http服务实例了。

试着修改→\_→的代码，查看下$compile服务到底是什么？

### 注册服务组件

从injector的角度看，组件就是一个功能提供者，因此被称为供给者/Provider。 在AngularJS中，provider以JavaScript类（构造函数）的形式封装。



服务名称通常使用一个字符串标识，比如"$http"代表http调用服务、"$rootScope"代表根 作用域对象、"$compile"代表编译服务...

Provider类要求提供一个$get函数（类工厂），injector通过调用该函数， 就可以获得服务组件的实例。

名称和类函数的组合信息，被称为配方。injector中维护一个集中的配方库， 用来按需创建不同的组件。这个配方库，其实就是一个Hash对象，key就是服务名称，value 就是类定义。

在→\_→的示例中，我们定义了一个简单的服务类，这个服务类的实例就是一个字符串：“hello,world!”。 我们使用"ezHello"作为其服务名在注入器里注册，并通过注入器将这个实例显示出来。

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<!--在这里显示ezHello实例的内容-->

<div id="logger"></div>

</body>

</html>

//在ezstuff模块上登记一个服务ezHello

angular.module("ezstuff",[])

.provider("ezHello",function(){

//$get方法是一个类工厂，返回服务的实例

this.$get = function(){

return "hello,world!";

};

});

angular.element(document).ready(function(){

angular.injector(["ng","ezstuff"]).invoke(function(ezHello){

//将ezHello实例对象转成字符串显示出来

var e = document.querySelector("#logger");

angular.element(e).text(ezHello);

});

});

### 获得注入器对象

要使用AngularJS的功能，必须首先获取注入器。有两种方法取得注入器。

* **创建一个新的注入器**

可以使用angular.injector()创建一个新的注入器：

1. angular.injector(modules, [strictDi]);

* **获取已经创建的注入器**

如果AngularJS框架已经启动，那么可以使用DOM对象的injector()方法获 得已经创建的注入器：

1. var element = angular.element(dom\_element);
2. var injector = element.injector();

### 通过注入器调用API

注入器有两个方法可供进行API调用：invoke()和get()。

* **invoke()**

使用注入器的invoke()方法，可以直接调用一个用户自定义的函数体，并通过函数参数 注入所依赖的服务对象，这是AngularJS推荐和惯例的用法：

1. angular.injector(['ng'])
2. .invoke(function($http){
3. //do sth. with $http
4. });

* **get()**

也可以使用注入器的get()方法，获得指定名称的服务实例：

1. var my$http = angular.injector(['ng']).get('$http');
2. //do sth. with my$http

→\_→的示例这次使用了get()方法直接获取一个服务实例，感受一下！

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<!--在这里显示ezHello实例的内容-->

<div id="logger"></div>

</body>

</html>

//在ezstuff模块上登记一个服务ezHello

angular.module("ezstuff",[])

.provider("ezHello",function(){

//$get方法是一个类工厂，返回服务的实例

this.$get = function(){

return "hello,world!";

};

});

angular.element(document).ready(function(){

//直接通过注入器获取ezHello实例对象

var myHello = angular.injector(["ng","ezstuff"]).get("ezHello");

//将ezHello实例对象转成字符串显示出来

var e = document.querySelector("#logger");

angular.element(e).text(myHello);

});

### 注入的方式和原理

有两种方法告知注入器需要注入的服务对象：参数名注入和依赖数组注入。

* **参数名注入**

AngularJS在执行invoke()函数时，将待注入函数定义转化为字符串，通过 正则表达式检查其参数表，从而发现并注入所所依赖的服务对象：

1. //myfunc通过参数表声明这个函数依赖于"$http"服务
2. var myfunc = function($http){
3. //do sth. with $http
4. };
5. injector.invoke(myfunc);//myfunc的定义将被转化为字符串进行参数名检查

这样有一个问题，就是当我们对JavaScript代码进行压缩处理时，$http可能会被 变更成其他名称，这将导致注入失败。

* **依赖数组注入**

AngularJS采用依赖项数组的方法解决代码压缩混淆产生的问题。这时传入invoke()的 是一个数组，数组的最后一项是实际要执行的函数，其他项则指明需要向该函数注入 的服务名称。注入器将按照数组中的顺序，依次向函数注入依赖对象。

采用这种方法，待注入函数的参数表的名称就无关紧要了：

1. //myfunc依赖于"$http"和"$compile"服务
2. var myfunc = ["$http","$compile",function(p1,p2){
3. //do sth. with p1($http),p2($compile)
4. }];
5. injector.invoke(myfunc);

→\_→的实例这次采用依赖数组的方法注入了ezHello服务实例，可以改改参数名称 看有没有影响结果？

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<!--在这里显示ezHello实例的内容-->

<div id="logger"></div>

</body>

</html>

//在ezstuff模块上登记一个服务ezHello

angular.module("ezstuff",[])

.provider("ezHello",function(){

//$get方法是一个类工厂，返回服务的实例

this.$get = function(){

return "hello,world!";

};

});

angular.element(document).ready(function(){

angular.injector(["ng","ezstuff"]).invoke(["ezHello",function(hhh){

//将ezHello实例对象转成字符串显示出来

var e = document.querySelector("#logger");

angular.element(e).text(hhh);

}]);

});

### 引导之前：库阶段

在右边的示例中，我们定义了一个指令ez-duang， 它应该会展开成一个动画 显示出来。

但是，看起来没有什么动画显示出来。AngularJS似乎没有工作，为什么？

有点像操作系统，AngularJs也有一个启动引导的概念。

当你在HTML文件中引入angular.min.js时，AngularJS只是建立了一个全局的 angular对象，这个对象有一些方法可供开发者调用，但应用的框架还没有建立。

在这个阶段，AngularJS还只是一个库，和jQuery类似，你可以使用angular.element() 操作DOM，也可以使用angular.injector()创建注入器... 但是，你定义的指令，你 创建的控制器，你封装的服务，你开发的模板...所有这些组件，还静静地躺在那里， 没有被整合在一起。

我们说，框架还没有运转起来，现在还是库阶段。

只有通过启动引导，AngularJS框架才开始将那些组件拼接在一起，应用才真正 开始运转。

像下面这样，试着给html元素增加一个ng-app指令，再重新运行！

1. <html ng-app="ezstuff">
2. ....
3. </html>

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<p>应该在下面看到一幅动画才对！</p>

<ez-duang></ez-duang>

</body>

</html>

angular.module("ezstuff",[])

.directive("ezDuang",function(){

return {

restrict : "E",

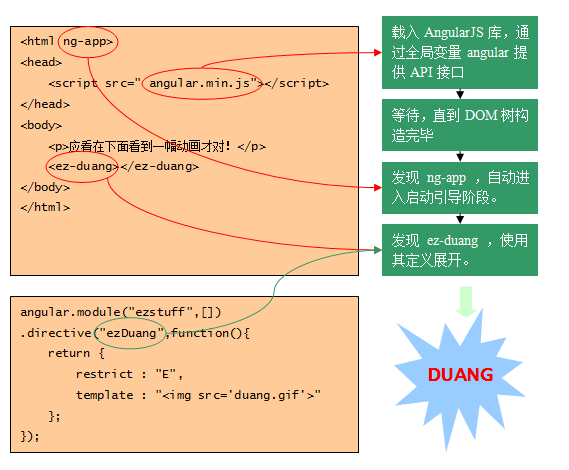
template : "<img src='http://ww4.sinaimg.cn/bmiddle/757eb2ffjw1eptcr4qobjg209205dthh.gif'>"

};

});

### 自动引导启动框架

就像你看到的那样，如果HTML模板中有某个标签有ng-app属性，那么当DOM树建立成功后， AngularJS就会自动进入引导过程，启动整个框架：



试着把ng-app指令挪到body元素上，看看有什么不同？

### 手工引导启动框架

在大多数情况下，我们都使用ng-app指令来进行自动引导启动，但是如果一个HTML文件中 有多个ng-app，AngularJS只会自动引导启动它找到的第一个ng-app应用，这是需要手工引导 的一个应用场景。

我们可以利用angular.bootstrap()方法进行手动引导:

1. angular.bootstrap(element, [modules], [config]);

bootstrap方法有三个参数：

* element ： 一个DOM元素，以这个元素为Angular应用的根，等同自动引导时ng-app所在 的元素。这个参数是必须的。比如：document、document.body等。
* modules ： 引导时需要载入的模块数组。比如：[]、["ezstuff"]等。由于我们的HTML中引用 了ezstuff模块中定义的ez-duang指令，所以，我们需要指定载入ezstuff模块。
* config ：引导配置项，可选。我们先忽略。

最终，我们使用如下的形式进行手动引导：

1. angular.bootstrap(document,["ezstuff"]);

改进的代码已经预置于→\_→。请点击【手动引导】按钮启动引导过程！

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<button id="bootstrap">手动引导</button>

<p>应该在下面看到一幅动画才对！</p>

<ez-duang></ez-duang>

</body>

</html>

angular.element(document).ready(function(){

var e = document.querySelector("#bootstrap");

angular.element(e)

.on("click",function(){

angular.bootstrap(document,["ezstuff"]);

})

})

;

angular.module("ezstuff",[])

.directive("ezDuang",function(){

return {

restrict : "E",

template : "<img src='http://ww4.sinaimg.cn/bmiddle/757eb2ffjw1eptcr4qobjg209205dthh.gif'>"

};

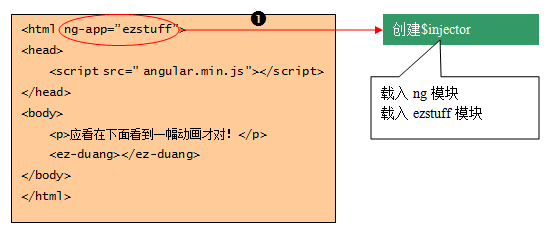
});

### 引导第1步：创建注入器

引导过程使AngularJS从库转变成了一个框架。

回忆我们之前提到，AngularJS深入骨髓地使用着依赖注入，那么，在引导过程 之初，首先需要创建一个注入器就毫不奇怪了。

注入器是通向AngularJS所有功能的入口，而AngularJS的功能实现，是通过模块的方式组织的。所以， 在创建注入器的时候，需要告诉AngularJS载入哪些模块（ng模块是内置载入的，不需要显式指定）。



在自动启动引导的场景下，可以给ng-app赋值以指定一个需要载入的模块，比如：

1. ng-app = "ezstuff"

在手动启动引导的场景下，通过bootstrap方法的第二个参数指定需要载入的模块，比如：

1. angular.bootstrap(document,["ezstuff"]);

INSIDE：无论自动启动还是手工启动，最终都是调用angular对象上的injector()方法创建了一个 注入器，然后把这个注入器存入了根对象的data里：

1. var injector = angular.injector(["ng","ezstuff"]);
2. angular.element(document).data("$injector",injector);

我们在→\_→的代码中，开始模拟引导启动过程的第一步：创建注入器。

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<p>现在还看不到动画！</p>

<ez-duang></ez-duang>

</body>

</html>

//模拟引导启动过程

angular.element(document).ready(function(){

//第一步：创建注入器并保存到根对象的data中

var injector = angular.injector(["ng","ezstuff"]);

angular.element(document).data("$injector",injector);

})

;

angular.module("ezstuff",[])

.directive("ezDuang",function(){

return {

restrict : "E",

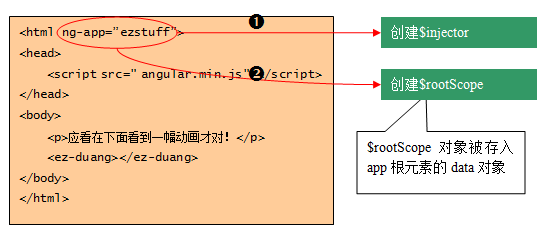
template : "<img src='http://ww4.sinaimg.cn/bmiddle/757eb2ffjw1eptcr4qobjg209205dthh.gif'>"

};

});

### 引导第2步：创建根作用域

scope对象是AngularJS实现数据绑定的重要服务，所以，在引导启动建立了注入器之后， AngularJS马上在应用的根节点上创建一个根作用域：$rootScope对象。



如果是自动引导启动，那么ng-app所在的DOM节点对应着根作用域。如果是手工引导启动， 那么在bootstrap方法中指定的第一个参数就对应着根作用域。

无论哪一种情况，一旦$rootScope对象创建成功，AngularJS就将这个对象存储到根节点 的data中，我们可以使用如下的方法查看这个对象：

1. angular.element(approot).data("$rootScope");

你可以摆弄一下代码，看看$rootScope到底是什么东西。

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<p>现在还看不到动画！</p>

<ez-duang></ez-duang>

</body>

</html>

//模拟引导启动过程

angular.element(document).ready(function(){

//第一步：创建注入器并保存到根对象的data中

var injector = angular.injector(["ng","ezstuff"]);

angular.element(document).data("$injector",injector);

//第二步：创建根作用域并保存到根对象的data中

var rootScope = injector.get("$rootScope");

angular.element(document).data("$rootScope",rootScope);

})

;

angular.module("ezstuff",[])

.directive("ezDuang",function(){

return {

restrict : "E",

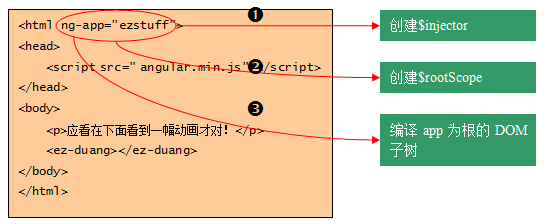
template : "<img src='http://ww4.sinaimg.cn/bmiddle/757eb2ffjw1eptcr4qobjg209205dthh.gif'>"

};

});

### 引导第3步：编译DOM子树

引导过程的最后一步，是以ng-app所在DOM节点为根节点，对这棵DOM子树进行编译。



编译过程通常借助于指令，完成这几种操作：

1. 对DOM对象进行变换。
2. 在DOM对象上挂接事件监听。
3. 在DOM对象对应的scope对象上挂接数据监听。

编译过程是AngularJS相当有特点的一个存在，我们将在下一节继续深入。

现在你应该看到结果了吧？

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<p>应该在下面看到一幅动画才对！</p>

<ez-duang></ez-duang>

</body>

</html>

//模拟引导启动过程

angular.element(document).ready(function(){

//第一步：创建注入器并保存到根对象的data中

var injector = angular.injector(["ng","ezstuff"]);

angular.element(document).data("$injector",injector);

//第二步：创建根作用域并保存到根对象的data中

var rootScope = injector.get("$rootScope");

angular.element(document).data("$rootScope",rootScope);

//第三步：编译DOM树

var compile = injector.get("$compile")

compile(document)(rootScope);

})

;

angular.module("ezstuff",[])

.directive("ezDuang",function(){

return {

restrict : "E",

template : "<img src='http://ww4.sinaimg.cn/bmiddle/757eb2ffjw1eptcr4qobjg209205dthh.gif'>"

};

});

### 编译器/$compile

编译器$compile是一个AngularJS的内置服务，它负责遍历DOM树来查找匹配指令， 并调用指令的实现代码进行处理。

HTML编译包括3个步骤：

* **匹配指令**

$compile遍历DOM树，如果发现有元素匹配了某个指令，那么这个指令将被加入 该DOM元素的指令列表中。一个DOM元素可能匹配多个指令。

* **执行指令的编译函数**

当一个DOM元素的所有指令都找齐后，编译器根据指令的优先级/priority指令进行排序。 每个指令的compile函数被依次执行。每个compile执行的结果产生一个link函数，这些 link函数合并成一个复合link函数。

* **执行生成的链接函数**

$compile通过执行指令的link函数，将模板和scope链接起来。结果就是一个DOM视图和scope对象模型 之间的动态数据绑定。

为何将编译和连接两个步骤分开?

简单说，当数据模型的变化会导致DOM结构变化时，指令就需要分别定义compile()函数和link函数。 例如，ng-repeat指令需要为数据集合中的每个成员复制DOM元素。将编译和链接过程分开可以有效 地提高性能，因为DOM的复制放在compile()里，仅需要执行一次，但链接则发生在每个生成的DOM元素 上，所以指令的link()函数会执行多次。

指令很少需要compile函数，因为大多数指令考虑的是作用于特定的DOM元素实例，而不是改变DOM 的结构。所以link函数更常用。

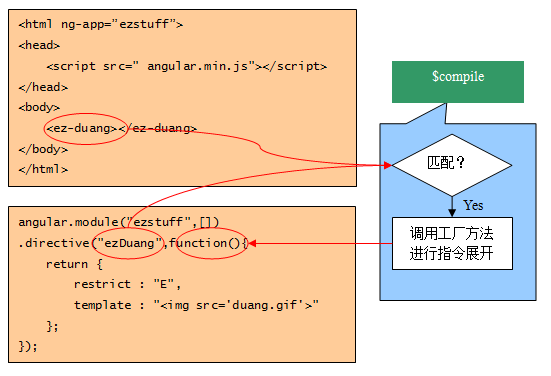
### 指令/directive

笼统地说，指令是DOM元素（例如属性、元素、CSS类等）上的标记符，用来告诉AngularJS的HTML编译器 （$compile服务）将特定的行为绑定到DOM元素，或者改变DOM元素。

指令可以放置在元素名、属性、CSS类名称及备注中。下面是一些等效的触发"ng-bind"指令的写法：

1. <span ng-bind="exp"></span>
2. <span class="ng-bind: exp;"></span>
3. <ng-bind></ng-bind>
4. <!-- directive: ng-bind exp -->

指令的实现本质上就是一个类工厂，它返回一个指令定义对象，编译器根据这个指令定义对象进行操作。



问题是，HTML中的ez-duang，怎么就匹配到了JavaScript中的ezDuang？

### 指令的规范化

AngularJS在进行匹配检测之前，首先对HTML元素的标签和属性名转化成规范的驼峰式字符串：

1. 去除名称前缀的x-和data-
2. 以: , - 或 \_ 为分割符，将字符串切分成单词，除第一个单词外，其余单词首字母大写
3. 重新拼接各单词

例如，下面的写法都等效地匹配ngBind指令：

1. <span ng-bind="name"></span> <br/>
2. <span ng:bind="name"></span> <br/>
3. <span ng\_bind="name"></span> <br/>
4. <span data-ng-bind="name"></span> <br/>
5. <span x-ng-bind="name"></span> <br/>

所以，在前面的课程中，我们在HTML中使用的ez-duang指令，将被规范为ezDuang， 编译器使用这个规范化的名称与注册的指令进行匹配。

### 控制器的作用

我们知道，在AngularJS中，实现数据绑定的核心是scope对象。那么控制器又有什么用呢？

简单地说，没有控制器/controller，我们没有地方定义业务模型。

回忆下ng-init指令。我们可以使用ng-init指令在scope对象上定义数据，比如：

1. <div ng-init="sb={name:'somebody',gender:'male',age:28}">

但是，ng-init的值是一个AngularJS表达式，没有办法定义方法。

控制器让我们有机会在scope上定义我们的业务逻辑，具体说，可以使用控制器：

1. 对scope对象进行初始化
2. 向scope对象添加方法

### 在模板中声明控制器

在一个HTML元素上使用ng-controller指令，就可以引入一个控制器对象：

1. <div ng-controller="myController">...</div>

### 控制器的实现

控制器实际上就是一个JavaScript的类/构造函数：

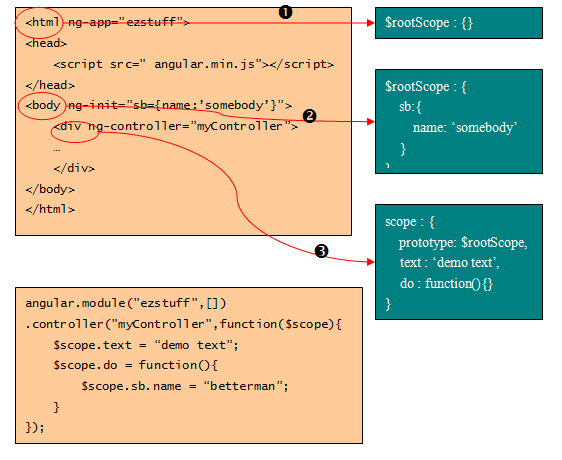
1. //控制器类定义
2. var myControllerClass = function($scope){
3. //模型属性定义
4. $scope.text = "...";
5. //模型方法定义
6. $scope.do = function(){...};
7. };
8. //在模块中注册控制器
9. angular.module('someModule',[])
10. .controller("myController",myControllerClass);

### 控制器的一次性

控制器构造函数仅在AngularJS对HTML文档进行编译时被执行一次。从这个角度看， 就更容易理解为何将控制器称为对scope对象的增强：一旦控制器创建完毕，就意味着scope对 象上的业务模型构造完毕，此后就不再需要控制器了- scope对象接管了一切。

### 控制器对scope的影响

ng-controller指令总是创建一个新的scope对象：



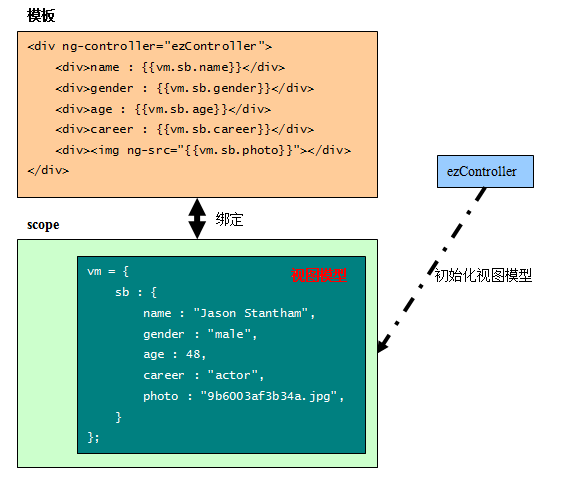
在图中，我们看到：

1. ng-app指令引发$rootScope对象的创建。开始时，它是一个空对象。
2. body元素对应的scope对象还是$rootScope。ng-init指令将sb对象挂在了$rootScope上。
3. div元素通过ng-controller指令创建了一个新的scope对象，这个对象的原型是$rootScope。
4. 因为原型继承的关系，在do函数中对sb的引用指向$rootScope.sb。

### 初始化$scope对象

通常在应用启动时，需要初始化scope对象上的数据模型。我们之前曾使用ng-init指令进行初始化， 而使用控制器则是更为规范的做法。

右边的示例定义了一个ezController，利用这个控制器，我们对业务模型进行了初始化赋值：



请注意，控制器仅仅负责在编译时在scope对象上建立视图对象vm，视图对象和模板的绑定则是由 scope负责管理的。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezController">

<div>name : {{vm.sb.name}}</div>

<div>gender : {{vm.sb.gender}}</div>

<div>age : {{vm.sb.age}}</div>

<div>career : {{vm.sb.career}}</div>

<div><img ng-src="{{vm.sb.photo}}"></div>

</div>

</body>

</html>

var ezControllerClass = function($scope){

//view model

$scope.vm = {

sb : {

name : "Jason Stantham",

gender : "male",

age : 48,

career : "actor",

photo : "http://b.hiphotos.baidu.com/baike/w%3D268/sign=a03742145bee3d6d22c680cd7b176d41/359b033b5bb5c9eae4c45250d739b6003af3b34a.jpg"

}

};

};

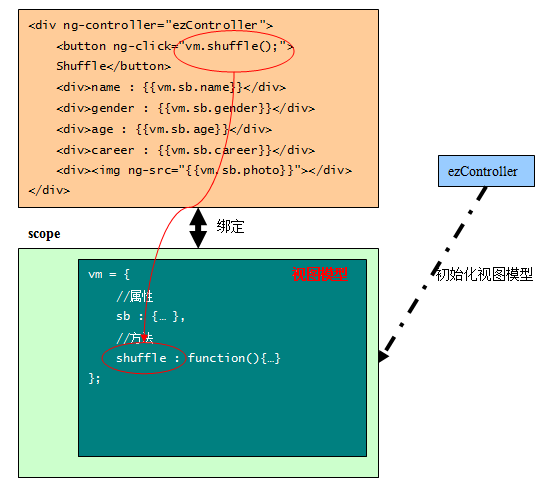
angular.module("ezstuff",[])

.controller("ezController",ezControllerClass);

### 向scope对象添加方法

业务模型是动态的，在数据之外，我们需要给业务模型添加动作。

在之前建立的业务模型上，我们增加一个随机挑选的方法：shuffle，这个方法负责 从一个小型的名人库中随机的选择一个名人来更新模型的sb属性：



通过在button上使用ng-click指令，我们将模型的shuffle方法绑定到了鼠标点击 事件上。试着用鼠标点击【shuffle】按钮，我们的模型将从库里随机的选出一个 名人，显示在视图里。

示例代码参见→\_→。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezController">

<button ng-click="vm.shuffle();">shuffle</button>

<div>name : {{vm.sb.name}}</div>

<div>gender : {{vm.sb.gender}}</div>

<div>age : {{vm.sb.age}}</div>

<div>career : {{vm.sb.career}}</div>

<div><img ng-src="{{vm.sb.photo}}"></div>

</div>

</body>

</html>

var ezControllerClass = function($scope){

//view model

$scope.vm = {

sb : {

name : "Jason Stantham",

gender : "male",

age : 48,

career : "actor",

photo : "http://b.hiphotos.baidu.com/baike/w%3D268/sign=a03742145bee3d6d22c680cd7b176d41/359b033b5bb5c9eae4c45250d739b6003af3b34a.jpg"

},

shuffle : function(){

var repo = [

{name:"Jason Stantham",gender:"male",age:48,career:"actor",photo:"http://b.hiphotos.baidu.com/baike/w%3D268/sign=a03742145bee3d6d22c680cd7b176d41/359b033b5bb5c9eae4c45250d739b6003af3b34a.jpg"},

{name:"Jessica Alba",gender:"female",age:32,career:"actress",photo:"http://h.hiphotos.baidu.com/baike/w%3D268/sign=ce8cdcb43bdbb6fd255be2203125aba6/b219ebc4b74543a91d7092831c178a82b9011411.jpg"},

{name:"Nicolas Cage",gender:"male",age:53,career:"actor",photo:"http://f.hiphotos.baidu.com/baike/w%3D268/sign=e97412d2359b033b2c88fbdc2dcf3620/4a36acaf2edda3cc4187b7f600e93901203f9280.jpg"},

{name:"崔永元",gender:"male",age:48,career:"independent journalist",photo:"http://e.hiphotos.baidu.com/baike/w%3D268/sign=856e3aab34d3d539c13d08c50286e927/8c1001e93901213ff48a548956e736d12f2e952d.jpg"},

{name:"Sheetal Sheth",gender:"female",age:36,career:"actress",photo:"http://h.hiphotos.baidu.com/baike/c0%3Dbaike80%2C5%2C5%2C80%2C26/sign=f3627d0333fa828bc52e95b19c762a51/060828381f30e924f7c565374c086e061d95f757.jpg"},

{name:"Barack Obama",gender:"male",age:58,career:"president",photo:"http://a.hiphotos.baidu.com/baike/w%3D268/sign=2a0045f7f1d3572c66e29bdab2126352/f7246b600c338744cb293d62520fd9f9d72aa03b.jpg"},

{name:"Владимир Владимирович Путин",gender:"male",age:63,career:"president",photo:"http://h.hiphotos.baidu.com/baike/w%3D268/sign=657e210bb17eca8012053ee1a9239712/8435e5dde71190efa1a915f7cf1b9d16fdfa604c.jpg"}

];

var idx = Math.floor(Math.random()\*repo.length);

$scope.vm.sb = repo[idx];

}

};

};

angular.module("ezstuff",[])

.controller("ezController",ezControllerClass);

### 别把任何代码都塞到控制器里！



控制器的设计出发点是封装单个视图的业务逻辑，因此，不要进行以下操作：

* DOM操作

应当将DOM操作使用指令/directive进行封装。

* 变换输出形式

应当使用过滤器/filter对输出显示进行转化。

* 跨控制器共享代码

对于需要复用的基础代码，应当使用服务/service进行封装

### 创建服务组件

在AngularJS中创建一个服务组件很简单，只需要定义一个具有$get方法的构造函数， 然后使用模块的provider方法进行登记：

1. //定义构造函数
2. var myServiceProvider = function(){
3. this.$get = function(){
4. return ....
5. };
6. };
7. //在模块中登记
8. angular.module("myModule",[])
9. .provider("myService",myServiceProvider);

右边的示例定义了一个支持四则运算的服务：ezCalculator。你可以点击按钮【3+4=?】 查看效果。

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<button onclick="doCalc();">3+4=?</button>

<div id="result"></div>

</body>

</html>

function doCalc(){

var injector = angular.injector(["ezstuff"]),

mycalculator = injector.get("ezCalculator"),

ret = mycalculator.add(3,4);

document.querySelector("#result").textContent = ret;

}

angular.module("ezstuff",[])

.provider("ezCalculator",function(){

this.$get = function(){

return {

add : function(a,b){return a+b;},

subtract : function(a,b){return a-b;},

multiply : function(a,b){return a\*b;},

divide: function(a,b){return a/b;}

}

};

})

### 可配置的服务

有时我们希望服务在不同的场景下可以有不同的行为，这意味着服务可以进行配置。

比如，我们希望小计算器可以根据不同的本地化区域，给计算结果追加货币符号前缀， 那么需要在这个服务创建之前，首先配置本地化区域的值，然后在具体的计算中， 根据这个值选择合适的货币符号。

AngularJS使用模块的config()方法对服务进行配置，需要将实例化的服务提供者 （而不是服务实例）注入到配置函数中：

1. angular.module("myModule",[])
2. .config(["myServiceProvider",function(myServiceProvider){
3. //do some configuration.
4. }]);

注意：服务提供者provider对象在注入器中的登记名称是：服务名+Provider。 例如： $http的服务提供者实例名称是"$httpProvider"。

右边的示例嵌入了可配置的ezCalculator服务，查看代码体会一下！

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<button onclick="doCalc();">3+4=?</button>

<div id="result"></div>

</body>

</html>

function doCalc(){

var injector = angular.injector(["ezstuff"]),

mycalculator = injector.get("ezCalculator"),

ret = mycalculator.add(3,4);

document.querySelector("#result").textContent = ret;

}

angular.module("ezstuff",[])

.provider("ezCalculator",function(){

var currency = "$";

this.setLocal = function(l){

var repo = {

"CN":"¥",

"US":"$",

"JP":"¥",

"EN":"€"

};

if(repo[l]) currency = repo[l];

};

this.$get = function(){

return {

add : function(a,b){return currency + (a+b);},

subtract : function(a,b){return currency + (a-b);},

multiply : function(a,b){return currency + (a\*b);},

divide: function(a,b){return currency + (a/b);}

}

};

})

.config(function(ezCalculatorProvider){

ezCalculatorProvider.setLocal("CN");

});

### 服务定义语法糖

使用模块的provider方法定义服务组件，在有些场景下显得有些笨重。AngularJS友好 地提供了一些简化的定义方法，这些方法通常只是对provider方法的封装， 分别适用于不同的应用场景：

* **factory**

使用一个对象工厂函数定义服务，调用该工厂函数将返回服务实例。

* **service**

使用一个类构造函数定义服务，通过new操作符将创建服务实例。

* **value**

使用一个值定义服务，这个值就是服务实例。

* **constant**

使用一个常量定义服务，这个常量就是服务实例。

### factory方法

factory方法要求提供一个对象工厂，调用该类工厂将返回服务实例。

1. var myServiceFactory = function(){
2. return ...
3. };
4. angular.module("myModule",[])
5. .factory("myService",myServiceFactory);

INSIDE：AngularJS会将factory方法封装为provider，上面的示例 等同于：

1. var myServiceFactory = function(){
2. return ...
3. };
4. angular.module("myModule",[])
5. .provider("myService",function(){
6. this.$get = myServiceFactory;
7. });

右边预置了使用factory方法改写的ezCalculator示例，感受下和provider方法的区别！

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<button onclick="doCalc();">3+4=?</button>

<div id="result"></div>

</body>

</html>

function doCalc(){

var injector = angular.injector(["ezstuff"]),

mycalculator = injector.get("ezCalculator"),

ret = mycalculator.add(3,4);

document.querySelector("#result").textContent = ret;

}

angular.module("ezstuff",[])

.factory("ezCalculator",function(){

return {

add : function(a,b){return a+b;},

subtract : function(a,b){return a-b;},

multiply : function(a,b){return a\*b;},

divide: function(a,b){return a/b;}

}

})

### service方法

service方法要求提供一个构造函数，AngularJS使用这个构造函数创建服务实例：

1. var myServiceClass = function(){
2. this.method1 = function(){...}
3. };
4. angular.module("myModule",[])
5. .service("myService",myServiceClass);

INSIDE：AngularJS会将service方法封装为provider，上面的示例 等同于：

1. var myServiceClass = function(){
2. //class definition.
3. };
4. angular.module("myModule",[])
5. .provider("myService",function(){
6. this.$get = function(){
7. return new myServiceClass();
8. };
9. });

右边预置了使用service方法改写的ezCalculator示例，感受下和factory方法的区别！

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<button onclick="doCalc();">3+4=?</button>

<div id="result"></div>

</body>

</html>

function doCalc(){

var injector = angular.injector(["ezstuff"]),

mycalculator = injector.get("ezCalculator"),

ret = mycalculator.add(3,4);

document.querySelector("#result").textContent = ret;

}

var ezCalculatorClass = function(){

this.add = function(a,b){return a+b;};

this.subtract = function(a,b){return a-b;};

this.multiply = function(a,b){return a\*b;};

this.divide = function(a,b){return a/b;};

};

angular.module("ezstuff",[])

.service("ezCalculator",ezCalculatorClass);

### value方法

有时我们需要在不同的组件之间共享一个变量，可以将这种情况视为一种服务： provider返回的总是变量的值。

value方法提供了对这种情况的简化封装：

1. angular.module("myModule",[])
2. .value("myValueService","cx129800123");

INSIDE：AngularJS会将value方法封装为provider，上面的示例 等同于：

1. angular.module("myModule",[])
2. .provider("myService",function(){
3. this.$get = function(){
4. return "cx129800123";
5. };
6. });

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<button onclick="showUserName();">show user name</button>

<div id="result"></div>

</body>

</html>

var showUserName = function(){

var injector = angular.injector(["ezstuff"]),

username = injector.get("ezUserName");

document.querySelector("#result").textContent = username;

};

angular.module("ezstuff",[])

.value("ezUserName","whoami");

### constant方法

有时我们需要在不同的组件之间共享一个常量，可以将这种情况视为一种服务： provider返回的总是常量的值。

constant方法提供了对这种情况的简化封装：

1. angular.module("myModule",[])
2. .constant("myConstantService","Great Wall");

和value方法不同，AngularJS并没有将constant方法封装成一个provider，而仅仅 是在内部登记这个值。这使得常量在AngularJS的启动配置阶段就可以使用（创建任何 服务之前）：你可以将常量注入到模块的config()方法中。

在右边的示例中，你可以试着将constant()改成value()，看看还能正常运行吗？

<html>

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<button onclick="doCalc();">3+4=?</button>

<div id="result"></div>

</body>

</html>

function doCalc(){

var injector = angular.injector(["ezstuff"]),

mycalculator = injector.get("ezCalculator"),

ret = mycalculator.add(3,4);

document.querySelector("#result").textContent = ret;

}

angular.module("ezstuff",[])

.constant("ezCurrency","CN")

.provider("ezCalculator",function(){

var currency = "$";

this.setLocal = function(l){

var repo = {

"CN":"¥",

"US":"$",

"JP":"¥",

"EN":"€"

};

if(repo[l]) currency = repo[l];

};

this.$get = function(){

return {

add : function(a,b){return currency + (a+b);},

subtract : function(a,b){return currency + (a-b);},

multiply : function(a,b){return currency + (a\*b);},

divide: function(a,b){return currency + (a/b);}

}

};

})

.config(function(ezCurrency,ezCalculatorProvider){

ezCalculatorProvider.setLocal(ezCurrency);

});

### 创建指令

指令也是一种服务，只是这种服务的定义有几个特殊要求：

1. 必须使用模块的directive()方法注册服务
2. 必须以对象工厂/factory()方法定义服务实现
3. 对象工厂必须返回一个指令定义对象
4. //定义指令的类工厂
5. var directiveFactory = function(injectables){
6. //指令定义对象
7. var directiveDefinationObject = {
8. ...
9. };
10. return directiveDefinationObject;
11. };
12. //在模块上注册指令
13. angular.module("someModule",[])
14. .directive("directiveName",directiveFactory);

INSIDE：指令在注入器中的登记名称是：指令名+Directive。 例如，ng-app指令的服务名称是："ngAppDirective"。

右边的示例定义一个简单的指令ez-hoverable，这个指令被限制只能 出现在属性的位置，每个具有这个指令的HTML元素，将在鼠标移入 时以虚线边框突出显示。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<span ez-hoverable>我是SPAN</span>

<button ez-hoverable>我是BUTTON</button>

<div ez-hoverable>我是DIV</div>

<textarea ez-hoverable>我是TEXTAREA</textarea>

</body>

</html>

var ezHoverableFactory = function(){

return {

restrict : "A",

link : function(scope,element,attrs){

element.on("mouseover",function(){

element.css({outline:"#ff0000 dotted thick"});

})

.on("mouseout",function(){

element.css({outline:"none"});

})

}

};

};

angular.module("ezstuff",[])

.directive("ezHoverable",ezHoverableFactory);

### 指令定义对象

每个指令定义的工厂函数，需要返回一个指令定义对象。指令定义对象就是 一个具有约定属性的JavaScript对象，编译器/$compile在编译时就根据这 个定义对象对指令进行展开。

指令定义对象的常用属性如下：

* **template** : *string*

使用template指定的HTML标记替换指令内容（或指令自身）

* **restrict** : *string*

用来限定指令在HTML模板中出现的位置。

* **replace** : *true|false*

使用这个属性指明template的替换方式。

* **scope** : *true|false|{...}*

scope属性为指令创建私有的作用域，这在创建可复用的Widget时非常有用。

* **link** : *function(..){...}*

link属性是一个函数，用来在指令中操作DOM树、实现数据绑定。

* **transclude** : *true|false|'element'*

允许指令包含其他HTML元素，这通常用于实现一个容器类型的Widget。

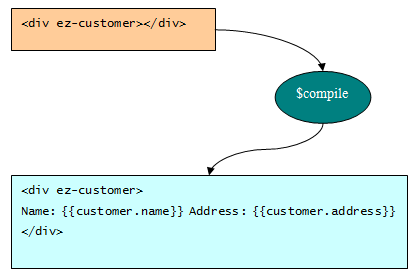
### template：定义替换模板

最简单的指令只需要使用template属性进行模板替换就可以实现。

template指明一个HTML片段，可以用来：

* 替换指令的内容。这是默认的行为，可以使用replace属性更改。
* 如果replace = true，那么用HTML片段替换指令本身。
* 包裹指令的内容，如果transclue属性为true。

右边的示例实现了一个ezCustomer指令，这个指令只是简单的使用template指定的 模板替换ez-customer的内容：



<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezCtrl">

<div ez-customer></div>

</div>

</body>

</html>

angular.module("ezstuff",[])

.controller("ezCtrl", ["$scope", function($scope) {

$scope.customer = {

name: "Naomi",

address: "1600 Amphitheatre"

};

}])

.directive("ezCustomer", function() {

return {

template: "Name: {{customer.name}} Address: {{customer.address}}"

};

});

### restrict：限制指令的出现位置

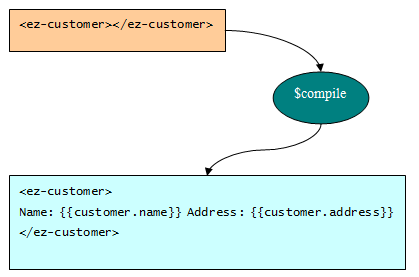
restict属性可以是EACM这四个字母的任意组合，用来限定指令的应用场景。 如果不指定这个属性，默认情况下，指令将仅允许被用作元素名和属性名：

* E - 指令可以作为HTML元素使用
* A - 指令可以作为HTML属性使用
* C - 指令可以作为CSS类使用
* M - 指令可以在HTML注释中使用

我们对之前的示例，增加一个restrict属性，限制这个只能作为元素名使用。 代码已经预置到右边，你可以看到，现在唯一合法的方式是使用如下方式应用指令：

1. <ez-customer></ez-customer>

考查编译后的DOM结构，你会发现ez-customer这个”伪“HTML标签还被保留着，这有时让完美 主义者有点闹心：



<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezCtrl">

<ez-customer></ez-customer>

</div>

</body>

</html>

angular.module("ezstuff",[])

.controller("ezCtrl", ["$scope", function($scope) {

$scope.customer = {

name: "Naomi",

address: "1600 Amphitheatre"

};

}])

.directive("ezCustomer", function() {

return {

restrict:"E",

template: "Name: {{customer.name}} Address: {{customer.address}}"

};

});

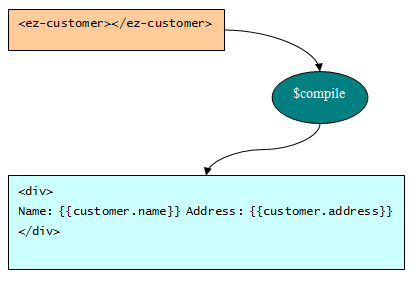
### replace：模板的使用方式

我们希望使用template完整地替换原始的DOM对象，而不是填充其内容，replace 属性负责这件事。

replace属性指明使用template时，如何替换指令元素：

* true - 编译时，将使用template替换指令元素
* false - 编译时，将使用template替换指令元素的内容

右边的示例增加了replace属性，值为true意味着这个指令要求编译器使用template 替换原始的DOM元素:



你可能注意到模板的内容稍微修改了一下，这是因为replace为true时，要求模板有 一个根节点。

angular.module("ezstuff",[])

.controller("ezCtrl", ["$scope", function($scope) {

$scope.customer = {

name: "Naomi",

address: "1600 Amphitheatre"

};

}])

.directive("ezCustomer", function() {

return {

restrict:"E",

replace:true,

template: "<div>Name: {{customer.name}} Address: {{customer.address}}</div>"

};

});

### 作用域问题

默认情况下，指令没有自己的scope对象，换句话说，它使用所在DOM对象对应的scope对象。

那么问题来了，如果一个指令在同一个scope内出现多次，会怎样？

1. <div ng-controller="ezCtrl">
2. <ez-customer></ez-customer>
3. <ez-customer></ez-customer>
4. </div>

没错，由于两个ez-customer指令都处在ezCtrl开辟的作用域内，所以两个指令绑定到了同样的 数据模型上，得到的是重复的结果。

显然，我们可以将每个ez-customer指令置于不同的作用域下，这意味着我们给每个ez-customer 一个不同的控制器：

1. <div ng-controller="ezCtrl1">
2. <ez-customer></ez-customer>
3. </div>
4. <div ng-controller="ezCtrl2">
5. <ez-customer></ez-customer>
6. </div>

看起来很怪异，对吗？

### scope：使用隔离的作用域

通过设置scope属性，指令的每个实例都将获得一个隔离的本地作用域：

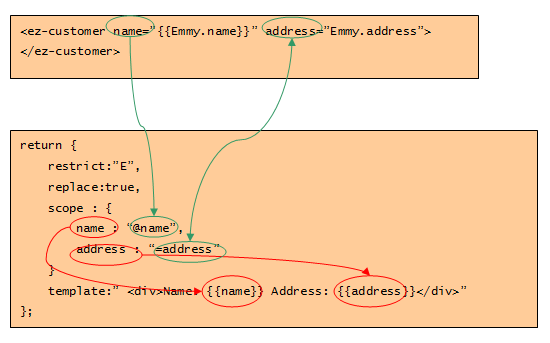
1. var ezCustomerDirectiveFactory = function(){
2. return {
3. restrict:"E",
4. replace:true,
5. scope:{
6. name : "@name",
7. address : "=address"
8. },
9. template:"<div>name:{{name}} address:{{address}}</div>"
10. }
11. }

在上面的例子中，我们在本地scope上定义了两个属性：name和address，这样在 模板中就可以使用name和address了。

你应该已经注意到，name属性的值之前有一个@符号，这是一个约定好的标记，它 告诉编译器，本地scope上的name值需要从应用这个指令的DOM元素的name属性值 读取，如果DOM元素的name属性值变了，那么本地scope上的name值也会变化。

同样，address属性之前的=符号也是一个约定好的标记，它告诉编译器，本地scope 上的address属性值和DOM元素的address属性值指定的外部scope对象上的模型需要 建立双向连接：外部scope上模型的变化会改变本地scope上的address属性，本地 scope上address属性的变化也会改变外部scope上模型的变化。

有点绕，上个图：



从图中可以看出：

1. 指令的template绑定的是本地scope上的name和address。
2. 本地scope的name属性的值始终是ez-customer对象上name属性的值
3. 本地scope的address属性值始终和ez-customer对应的scope对象上的Emmy.address 保持同步。

再送个示例→\_→，多琢磨下。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezCtrl">

<ez-customer sb="Emmy"></ez-customer>

<ez-customer sb="Edison"></ez-customer>

</div>

</body>

</html>

angular.module("ezstuff",[])

.controller("ezCtrl", ["$scope", function($scope) {

$scope.Emmy = {

name: "Emmy",

address: "1600 Amphitheatre"

};

$scope.Edison = {

name: "Edison",

address: "2500 Amphitheatre"

};

}])

.directive("ezCustomer", function() {

return {

restrict:"E",

replace:true,

scope:{

customer:"=sb",

},

template: "<div>Name: {{customer.name}} Address: {{customer.address}}</div>"

};

});

### link:在指令中操作DOM

如果需要在指令中操作DOM，我们需要在对象中定义link属性，link函数的定义如下：

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

注意link函数的参数，AngularJS在编译时负责传入正确的值：

* **scope**

指令对应的scope对象。如果指令没有定义自己的本地作用域，那么传入的就是外部的 作用域对象。

* **iElement**

指令所在DOM对象的jqLite封装。如果使用了template属性，那么iElement对应 变换后的DOM对象的jqLite封装。

* **iAttrs**

指令所在DOM对象的属性集。这是一个Hash对象，每个键是驼峰规范化后 的属性名。

后两个参数我们先略过。

### 示例

在右边的示例中，我们实现了一个可以指定显示格式的小时钟指令：ezCurrentTime。和原来一样， 我们在link函数中启动定时器，并在定时器中更新DOM。有几点解释下：

1. 我们在scope上使用$watch()方法对format的值进行监听，并使用这个值调整显示格式
2. 我们监听element的$destroy事件，这个事件是在DOM对象销毁时触发。我们在这个事件触发时 销毁定时器以释放资源
3. 我们使用了AngularJS内置的$interval服务，而不是setInterval()函数创建定时器。
4. 我们使用了AngularJS内置的dateFilter过滤器服务，对时间的显示进行格式化。 和$interval一样，dateFilter服务也是通过注入器注入的。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezCtrl">

Date format: <input ng-model="format"> <hr/>

Current time is: <span ez-current-time="format"></span>

</div>

</body>

</html>

angular.module("ezstuff", [])

.controller("ezCtrl", ["$scope", function($scope) {

$scope.format = "M/d/yy h:mm:ss a";

}])

.directive("ezCurrentTime", ["$interval", "dateFilter", function($interval, dateFilter) {

//定义link函数

function link(scope, element, attrs) {

var format,

timeoutId;

//更新DOM内容

function updateTime() {

element.text(dateFilter(new Date(), format));

}

//监听时钟格式

scope.$watch(attrs.ezCurrentTime, function(value) {

format = value;

updateTime();

});

//在DOM对象销毁时注销定时器

element.on("$destroy", function() {

$interval.cancel(timeoutId);

});

//启动定时器

timeoutId = $interval(function() {

updateTime(); //update DOM

}, 1000);

};

//返回指令定义对象

return {

link: link

};

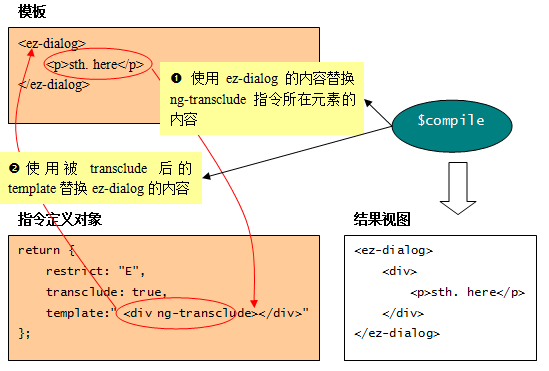
}]);

### transclude:包含其他元素

有些指令需要能够包含其他未知的元素。比如我们定义一个指令ez-dialog，用来 封装对话框的样式和行为，它应当允许在使用期（也就是在界面模板文件里）才指 定其内容：

1. <ez-dialog>
2. <p>对话框的内容在我们开发ez-dialog指令的时候是无法预计的。这部分内容需要
3. 被转移到展开的DOM树中适当的位置。</p>
4. </ez-dialog>

transclude属性可以告诉编译器，利用所在DOM元素的内容，替换template中包含 ng-transclude指令的元素的内容：



从上图中可以看到，使用transclude有两个要点：

1. 需要首先声明transclude属性值为true，这将告诉编译器，使用我们这个指令的 DOM元素，其内容需要被复制并插入到编译后的DOM树的某个点。
2. 需要在template属性值中使用ng-transclude指明插入点。

右边嵌入了ez-dialog的实现实例。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezCtrl">

<ez-dialog>

<p>Check out the contents, {{name}}!</p>

<p><button>Ok,I will</button></p> </ez-dialog>

</div>

</body>

</html>

angular.module("ezstuff", [])

.controller("ezCtrl", ["$scope", function($scope) {

$scope.name = "Mr. Whoami";

}])

.directive("ezDialog", function() {

return {

restrict: "E",

replace : true,

transclude: true,

template: "<div class='ez-dialog'><div class='header'>alert</div><div class='content' ng-transclude></div></div>"

};

});

### 在视图模板中使用过滤器

过滤器也是一种服务，负责对输入的内容进行处理转换，以便更好地向用户显示。

过滤器可以在模板中的{{}}标记中使用：

1. {{ expression | filter:arg1:arg2}}

* 预置的过滤器

AngularJS的ng模块实现了一些预置的过滤器，如：currency、number等等，可以直接 使用。例如下面的示例将对数字12使用currency过滤器，结果将是"$12.00"：

1. {{12|currency}}

* 带参数的过滤器

过滤器也可以有参数，例如下面的示例将数字格式化为"1,234.00":

1. {{1234|number:2}}

* 过滤器流水线

过滤器可以应用于另一个过滤器的输出，就像流水线，语法如下：

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

右边是使用过滤器的示例。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezCtrl">

Total salary is {{total|number:2|currency}}

</div>

</body>

</html>

angular.module("ezstuff",[])

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

$scope.total = 123;

});

### 在代码中使用过滤器

别忘了过滤器也是一种服务，所以你可以将它注入你的代码中。

和普通的服务不同，过滤器在注入器中注册时，名称被加了一个后缀：Filter。 例如，number过滤器的服务名称是：numberFilter，而currency过滤器的服务名称是： currencyFilter。

通常我们的代码被封装在三个地方：控制器、服务、指令。这些地方都支持服务的直接 注入，例如：

1. angular.module('myModule',[])
2. .controller(function($scope,numberFilter){
3. //...
4. })

有时你需要显式的通过注入器调用过滤器，那么使用注入器的invoke()方法：

1. angular.injector(['ng'])
2. .invoke(function(numberFilter){
3. //...
4. })

总之，记住过滤器是一种服务，除了名字需要追加Filter后缀，和其他服务的调用方法没 什么区别。

右边的示例在控制器中注入了number和currency过滤器，实现对total的格式化。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body>

<div ng-controller="ezCtrl">

Total salary is {{total}}

</div>

</body>

</html>

angular.module("ezstuff",[])

.controller("ezCtrl",function($scope,numberFilter,currencyFilter){

$scope.total = currencyFilter(numberFilter(123,2));

});

### 创建过滤器

和指令类似，过滤器也是一种特殊的服务，与创建一个普通的服务相比较：

1. 必须使用模块的filter()接口注册服务
2. 必须提供对象工厂/factory方法
3. 对象工程必须返回一个过滤器函数，其第一个参数为输入变量
4. //定义过滤器类工厂
5. var filterFactory = function(){
6. //定义过滤器函数
7. var filter = function(input,extra\_args){
8. //process input and generate output
9. return output
10. }
11. };
12. //在模块上注册过滤器
13. angular.module("someModule",[])
14. .filter("filterName",filterFactory);

右边的示例定义了一个将字符串格式化为大写的过滤器。

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body ng-init="text='just a demo!'">

<p>{{text|ezUC}}</p></body>

</html>

//过滤器对象工厂定义

var ezUCFilterFactory = function(){

//过滤器对象返回的是一个过滤函数

var filter = function(input){

return input.toUpperCase();

}

return filter;

};

angular.module("ezstuff",[])

//使用模块的filter()接口注册过滤器

.filter("ezUC",ezUCFilterFactory);

### 为过滤器增加参数

过滤器的行为可以通过额外的参数来调整。比如，我们希望改进上一节的示例，使其可以 支持仅大写每个单词的首字母。

* **实现**

通过在过滤器类工厂返回的过滤器函数中传入额外的参数，就可以实现这个功能。

1. var filter = function(input,argument1,argument2){...}

* **使用**

在使用过滤器时，额外的参数通过前缀:引入，比如

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

右边的示例实现了支持参数的过滤器ezUC，试着去掉HTML模板中过滤器ezUC的参数， 看看显示输出的区别！

<html ng-app="ezstuff">

<head>

<script src="http://lib.sinaapp.com/js/angular.js/angular-1.2.19/angular.min.js"></script>

</head>

<body ng-init="text='just a demo!'">

<p>{{text|ezUC:true}}</p></body>

</html>

//过滤器对象工厂定义

var ezUCFilterFactory = function(){

//过滤器对象返回的是一个过滤函数

return function(input,cap){

if(!cap) return input.toUpperCase();

var output = input.replace(/\b\w+\b/g, function(word) {

return word.substring(0,1).toUpperCase( ) +word.substring(1);

});

return output;

}

};

angular.module("ezstuff",[])

//使用模块的filter()接口注册过滤器

.filter("ezUC",ezUCFilterFactory);